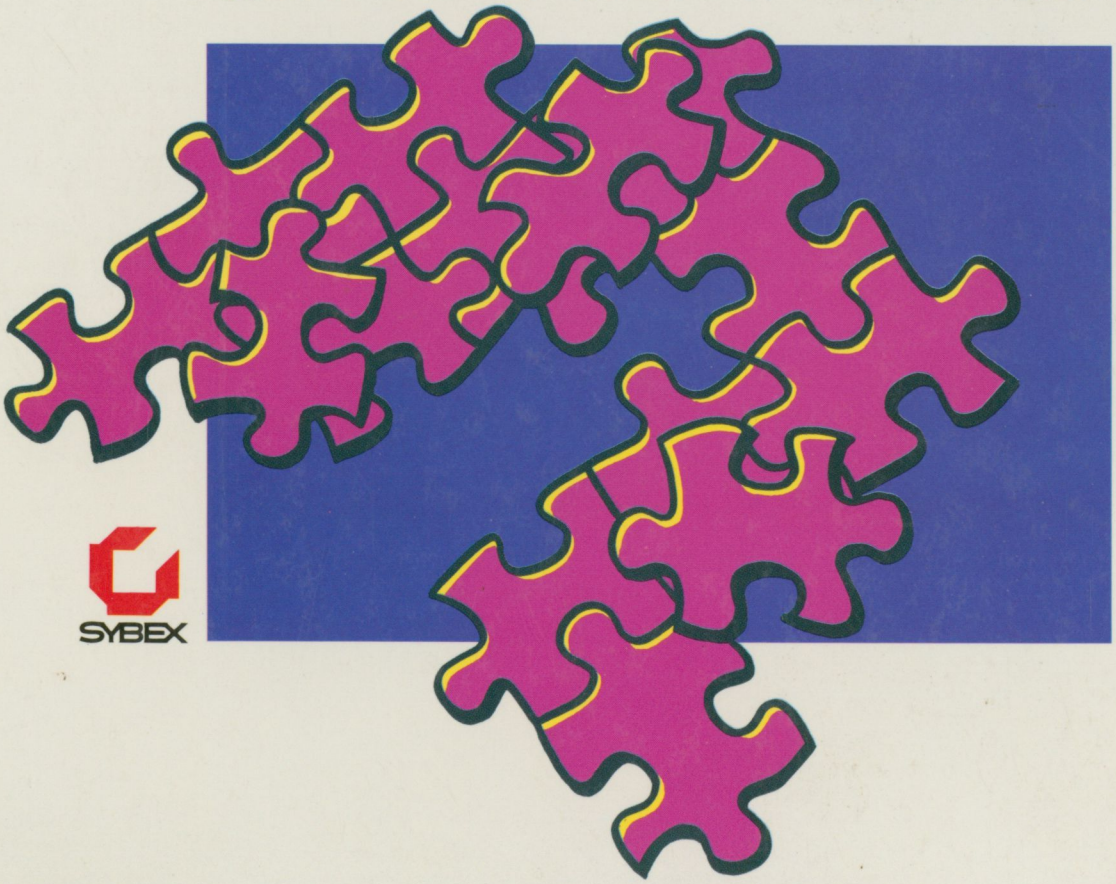
micro passion

COLLECTION

80386 Fonctionnement et programmation

J.H. CRAWFORD ET P.P. GELSINGER



SYBEX

80386
FONCTIONNEMENT
ET PROGRAMMATION

John H. Crawford
Patrick P. Gelsinger

80386
FONCTIONNEMENT ET PROGRAMMATION

Paris • San Francisco • Düsseldorf • Londres • Amsterdam

Collection dirigée par Lucien Lamarre

Traduction : Franck FRANCHIN

Ashton-Tate et dBASE sont des marques déposées par Ashton-Tate.
IBM, PC-AT et PS/2 sont des marques déposées de International Machines Corporation.
Intel est une marque de Intel Corporation.
Lotus et 1-2-3 sont des marques déposées de Lotus Development Corporation.
MS-DOS est une marque de Microsoft Corporation.
MuitiMate est une marque déposée par Multimate International.
UNIX est une marque déposée de AT&T Bell Laboratories.

Sybex n'est lié à aucun constructeur.

Tous les efforts ont été faits pour fournir dans ce livre une information complète et exacte. Néanmoins, Sybex n'assume de responsabilités, ni pour son utilisation ni pour les contrefaçons de brevets ou atteintes aux droits de tierces personnes qui pourraient résulter de cette utilisation.

Copyright (c) 1987 Sybex Inc
Copyright (c) 1990 Sybex

Tous droits réservés. Toute reproduction, même partielle, par quelque procédé que ce soit, est interdite sans autorisation préalable. Une copie par xérographie, photographie, film, bande magnétique ou autre, constitue une contrefaçon passible des peines prévues par la loi sur la protection des droits d'auteur.

ISBN 2-7361-0632-6
(Version originale, ISBN 0-89588-381-

ISSN 0990-2856

SOMMAIRE

Introduction XVII

1 : Concepts de base 1

Chronologie des microprocesseurs Intel 3
Compatibilité avec le 8086 et le 80286 4
Les formats des données 5
La mémoire 5
Notation 6
Nombres non signés 8
Entiers signés 8
Nombres avec offset 10
Signe magnitude 10
Complément à un 10
Complément à deux 11
Chaînes 13
ASCII 13
Bits 14
BCD 17
Types de données à virgule flottante 18
Introduction à la notion de virgule flottante 18
La norme virgule flottante IEEE 20
Que faire sans le 80387 ? 21
Formats des données 22
Types de données entières 22
BCD 23
Les formats réels 24
Réels temporaires 28
Cas particuliers 30
Exceptions 36

2 : Etat de la machine et adressage mémoire 37

Les registres 39
Les registres généraux 41
Caractérisation d'un registre général 42
Les registres de contrôle du processeur 43
Le registre pointeur d'instruction - EIP 43
Le registre des indicateurs d'état et de contrôle - EIP 43
Les registres segments 50
Les concepts d'adressage de la mémoire 51
Adressage en deux parties 51
Notation 52
Le mécanisme d'adressage de la mémoire 53
La partie segment : le registre segment 54
La partie déplacement : les modes d'adressage 56
La pile programme 59
Stockage temporaire dans la pile 60
Les sous-programmes et la pile 60
Type de données pointeur 64
Les modes d'adressage et les structures de données 66
Stratégies de segmentation 67
Codage des instructions 72
Champs d'une instruction 73
Les constantes immédiates 75
Les opérandes registre 78
Les opérandes mémoire 82
Codage des modes d'adressage à un octet 83
Codage des modes d'adressage à deux octets 89
Octet préfixe de recouvrement de segment 94
L'espace E/S 96
Les registres à virgule flottante 97
Pile accumulateur à virgule flottante 98
Registres de contrôle et d'état 16 bits 103
Le registre d'étiquette 103
Le registre de contrôle 103
Le registre d'état 108
Les registres pointeurs d'erreurs 109
Le format des opérandes de FLDENV, FSTENV, FNSTENV 111

3 : Le jeu d'instructions 113

Table des matières 115
Introduction 122
Index alphabétique des instructions 123
Format de description d'une instruction 132
Format d'une instruction 132
Description d'une instruction 135
Fonctionnement d'une instruction 135
Exemples d'une instruction 136
Exceptions associées à une instruction 137
Notes sur une instruction 139
Syntaxe de description d'une instruction 139
Fonctions 146
Entier 149
Segments multiples 348
Définitions des fonctions 349
Routines d'exception 349
Résumé du jeu d'instructions 350
Système d'exploitation 373
Définitions de fonctions 373
Routines d'exception 374
Résumé du jeu d'instructions 375
Virgule flottante 418

4 : Exemples du jeu d'instructions 535

Syntaxe 537
Types 538
Allocation des données 540
Déplacement 541
Etiquettes 541
PROC/ENDP 542
EQU 542
Exemples sur les entiers 543
Division signée 543
Tri 545
Factorielle 547
Sémaphore 550
Recherche de chaîne 553

Transfert de blocs binaires 557
Exemples d'instructions à virgule flottante 565
Indicateurs à virgule flottante 565
Reste partiel 567
Calculs exponentiels 568
Multiplication de matrices 569
Statistiques 573

5 : Gestion mémoire, protection et tâches 577

Gestion mémoire 581
Translation d'adresses 581
Segmentation et pagination 582
Mémoire virtuelle 585
Protection 586
Protection entre les tâches 587
Protection dans une tâche 588
Niveaux de protection combinés 591
Segmentation 593
Les tables des descripteurs de segments 599
Sélecteurs de segments 602
Sélecteur nul 604
Utilisation du champ RPL 605
Descripteurs de segments 605
Base et limite 606
Attributs de segment 608
Format des descripteurs de segments mémoire 609
Segments système et guichets 613
Les registres cachés des descripteurs 616
Pagination 618
Structure de la table des pages 620
Structure à deux niveaux de la table des pages 621
Tables des pages non présentes 623
Table des pages locales / table des pages globales 624
Format d'un élément de la table des pages 627
Mémoire virtuelle 631
Protection au niveau pagination 632
Modification des éléments d'une table des pages 633
Cohérence du cache de pagination 634

Multiprocesseurs 635
Segments système et registre de contrôle du processeur 636
Registres de contrôle du processeur 637
CR0 - Bits de contrôle du coprocesseur 638
CR0 - Bits de contrôle de la protection 640
CR2 et CR3 641
Registres de base des tables de segmentation 642
Format du segment d'état de la tâche 644
Champ de liens 647
Pointeurs de pile de niveaux internes 648
Registres de base d'adressage 649
Zone de sauvegarde des registres 650
Champs divers 650
Instructions sensibles au niveau de privilège 651
Instructions privilégiées 651
Protection de l'espace des E/S 652
Table d'autorisations E/S 654
Instructions qui modifient EFLAGS 660
Méthodes de transfert de contrôle 661
Même niveau, même tâche 663
Niveaux différents, même tâche 664
Les guichets d'appel 665
Retours de sous-programmes vers un niveau externe 671
La segmentation 672
Résumé sur les exceptions 674
Exceptions du type segment 674
Exceptions du type page 676
Arrêt de l'exécution des descriptions d'instructions 678
L'accès aux données en mémoire 678
Notations, définitions des structures et variables globales 679
Références à des données 682
Chargement d'un registre segment 690
Chargement de LDTR par l'instruction LLDT 700
Chargement de TR par l'instruction LTR 702
Transfert de contrôle 705
L'instruction JMP 708
L'instruction CALL 713
L'instruction RET 722

Commutations de tâches 727
Le guichet de tâche 729
La commutation de tâches 735

6 : Interruptions et exceptions 743

Interruptions 746
Interruptions INTR 747
Interruptions NMI 747
Exceptions 748
Relance d'une instruction 749
Types d'exception 750
Exception 0 - Erreur de division 750
Exception 1 - Exceptions du type mise au point 750
Exception 3 - Point d'arrêt INT 3 751
Exception 4 - Débordement 751
Exception 5 - Test de limite 751
Exception 6 - CodeOp non valide 752
Exception 7 - Processeur d'extension non disponible (DNA) 752
Exception 8 - Double faute 753
Exception 9 - Dépassement segment par le processeur d'extension 754
Exception 10 - TSS non valide 754
Exception 11 - Segment non présent 755
Exception 12 - Segment pile 755
Exception 13 - Protection générale 756
Exception 14 - Exceptions du type page 757
Exception 16 - Erreur du coprocesseur 758
Résumé des exceptions 758
Priorité des interruptions et des exceptions 758
Masquage des interruptions et des exceptions 761
Méthodes de transfert des interruptions/exceptions 762
Les guichets d'interruption et de trappe 765
IRET avec NT = 0 769
Transferts à travers des guichets de tâche 771
Guichets de tâche ou guichets d'interruption/trappe ? 771
Interruptions et exceptions 773
Description des interruptions 773
Guichets d'interruption et de trappe 776

Instruction IRET 779
Génération des exceptions 782
Routine SegmentException() 783
Routine PageException() 785
Exceptions du type erreur du coprocesseur 787
Exceptions masquées / Exceptions non masquées 787
Les catégories d'erreurs du coprocesseur 789
Exceptions du type opération non valide 790
Exceptions du type dénormalisé 791
Exceptions du type division par zéro 792
Exceptions du type overflow 792
Exceptions du type underflow 793
Exceptions du type précision (IEEE inexact) 794
Priorité des exceptions du type erreur du coprocesseur 795

7 : Mise en oeuvre du système d'exploitation 797

Syntaxe 799
SEGMENT/ENDS 799
USE32/USE16 799
ORG 800
ASSUME 800
Exemple de syntaxe 800
Exemple d'initialisation 801
Vue d'ensemble de l'exemple 1 801
Hypothèses 801
Gestion multitâche et protection 802
Espace adressable virtuel 804
Espace adressable linéaire 806
Pagination 808
Adresses physiques 810
Commutation de tâches 812
Description de l'exemple d'initialisation 813
Equivalences 813
Définitions des segments 813
Séquence de Code 1 - Démarrage à froid 814
Séquence de Code 2 - Divers 815
Séquence de Code 3 - Construction de GDT, LGDT 815
Séquence de Code 4 - Passage en mode protégé 816

Séquence de Code 5 - Chargement des descripteurs de segments 817
Séquence de Code 6 - Construction de la LDT, LLDT 817
Séquence de Code 7 - Construction de l'IDT 818
Séquence de Code 8 - Construction du catalogue des pages 819
Séquence de Code 9 - Construction des tables des pages 821
Séquence de Code 10 - Activation de la pagination 824
Séquence de Code 11 - Chargement du pointeur de pile du système d'exploitation 825
Séquence de Code 12 - Construction du TSS 825
Chargement de la tâche utilisateur 826
Préparation de l'appel à la tâche utilisateur 827
Appel de la tâche utilisateur 827
Handler d'exception du coprocesseur 844
Vue générale de l'exemple 2 844
Handler d'exception 846
Séquence de Code 1 - Emulation 846
Séquence de Code 2 - Base de la tâche courante 846
Séquence de Code 3 - Sauvegarde de l'ancien contexte 847
Séquence de Code 4 - Chargement du nouveau contexte 847

8 : Aide à la mise au point 851

Terminologie 853
Points d'arrêt 854
Les registres de mise au point 855
DR6 857
DR7 858
Reconnaissance de l'adresse d'un point d'arrêt 860
Points d'arrêt sur code / points d'arrêt sur donnée 862
Autres possibilités de mise au point 863
Trappe de mise au point dans le TSS 863
INT3 864
Mode pas à pas 865

9 : Exécution de programmes 8086 et 80286 867

Registres 16 bits et modes d'adressage 870
Exécution de programmes 8086 872
Segmentation et adressage 874

Exceptions lors de références mémoire 877
Références mémoire en mode réel et en mode virtuel 8086 878
Emulation de l'arithmétique 20 bits (adresses 8086) 880
Instructions non valides 880
Format 8086 pour FLDENV, FSTENV et FNSTENV 882
Mode virtuel 8086 883
IOPL en mode virtuel 8086 884
Entrer et quitter le mode virtuel 8086 886
Mode réel 905
Etat du 80386 à l'initialisation 905
Transferts du mode réel au mode protégé 908
Transferts du mode protégé au mode réel 909
Traitement des interruptions en mode réel 910
Exécution de programmes 80286 en mode protégé 912

Annexe A : Comparaison des 80386, 80286 et 8086 917

Le 8086 comparé au 80386 919
Exceptions du type division 919
CodeOps non définis 921
Valeur de PUSH SP 922
Décalages et rotations 922
Préfixes redondants 922
Accès au-delà des 64 Ko 922
LOCK 923
Mode pas à pas 923
Registre FLAGS 923
Interruption NMI 923
Erreur du coprocesseur 923
Préfixes sur les instructions du coprocesseur 924
Adressage d'un méga-octet 924
Répétition d'instructions opérant sur des chaînes 924
Le 80286 comparé au 80386 924
LOCK 925

Annexe B : Comparaison des 80387, 80287 et 8087 927

Le 80287 (et le 8087) comparé au 80387 929
Exécution des instructions 929
Intervalles étendus 929

FPTAN 929
Opérations sur les dénormalisés 930
Codes de condition FPREM 930
FLD de dénormalisé 930
FXTRACT 931
FLD d'une constante 931
FLD d'un réel court ou d'un réel long 931
FSETPM 931
Fonctions transcendantes 931
Autres différences 932
RESET 932
NaN silencieux 932
Formats non autorisés 932
Opération non valide 933
Exception du type dénormalisé 933
Exception du type débordement (overflow) 933
Exception du type débordement (underflow) 933
Priorité des exceptions 934
Contrôle de l'infini 934
Débordement de pile 934
Bits de condition du mot d'état 934
Mot d'étiquette 935
Le 8087 comparé au 80387 935
FENI/FNENI et FDISI/FNDISI 935
Exceptions du type numérique 935
Pointeur d'exception 936
Segmentation 936
Exception 7 936
Préfixes d'instructions du coprocesseur 936
Instruction WAIT 937

Annexe C : Table binaire, hexadécimale et décimale 939

Annexe D : Puissances de deux 943

Annexe E : Table des codes ASCII 947

Annexe F : Table des CodeOp 80386 951

Abréviations 953
Codage de l'adressage 953
Codage du type d'opérande 955
Codage des registres 955

Annexe G : Format des instructions du 80386 et timing 961

Codage des instructions 80386 et nombre de cycles 963
Codage des instructions 978
Extensions 32 bits du jeu d'instructions 980
Codage des champs d'instructions 981
Codage du champ (w) de longueur d'opérande 981
Codage du champ (reg) de registre général 982
Codage du champ (sreg) de registre segment 983
Codage du mode d'adressage 985
Codage du champ (d) de direction 990
Codage du champ (s) d'extension de signe 991
Codage du champ (tttn) de test conditionnel 991
Codage du champ (eee) des registres de contrôle ou de mise au point 991

Annexe H : Guide de désassemblage 995

Annexe I : Extensions du 80387 au jeu d'instructions du 80386 1003

Index 1011

INTRODUCTION

Cet ouvrage traite du 80386, le dernier membre de la fameuse famille Intel 86, et s'adresse au programmeur en assembleur. Tout au long de ce livre, nous nous intéresserons aux caractéristiques 32 bits de ce circuit. Le 80386 est entièrement compatible avec le 8086 et avec le 80286, et nous résumerons ces caractéristiques dans le Chapitre 9. En outre, afin de compléter l'étude du 80386, nous étudierons aussi le 80387, le coprocesseur numérique du 80386. Plutôt que de présenter le 80387 dans une annexe ou dans un chapitre à part, comme beaucoup le font, nous l'intégrerons dans l'ensemble.

Après avoir passé quelques années à développer ce circuit, nous avons la satisfaction de vous expliquer *de l'intérieur* comment programmer et utiliser le 80386. Tout au long de ce livre, nous nous sommes efforcés de travailler avec précision et exactitude, comme seuls ceux qui conçoivent les "puces" peuvent le faire.

Il existe une question importante à laquelle nous devons répondre : pourquoi lire ce livre ? En quoi le fait de comprendre et de savoir programmer le 80386 pourra-t-il vous être utile dans votre carrière de programmeur pour les dix ou vingt ans à venir ? La réponse se trouve dans le fantastique investissement cumulatif de la famille 86. Pour concevoir et utiliser un ordinateur, on investit sans cesse. On achète des ordinateurs (IBM PC, PC/AT, PS/2), des systèmes d'exploitation (UNIX, MS-DOS), des langages de programmation (C, FORTRAN), des programmes d'application (Lotus 1-2-3, MultiMate, dBASE III), et puis d'autres matériels (matériels graphiques, autres disques durs, connexions en réseaux, mémoires supplémentaires). Ces investissements incluent aussi les programmes que vous écrivez vous-même, et, bien sûr, le temps que vous passez à étudier. Ainsi, les familles d'ordinateurs, comme celle du 86, maintiennent et partagent la compatibilité d'une génération à l'autre. Cette compatibilité permet d'assurer la pérennité d'importants investissements qui ont déjà été effectués dans une famille d'ordinateurs.

Nous présumons que vous connaissez les principes de base du fonctionnement des ordinateurs. Nous présumons aussi que ce n'est pas votre première expérience dans le langage d'assembleur. Nous avons donc omis délibérément ces principes de base et nous conseillons aux lecteurs inexpérimentés de combler ces lacunes en lisant d'abord un ouvrage

d'introduction. On peut décomposer sommairement ce livre en trois parties. Dans les Chapitres 1 à 4, nous exposerons le point de vue des programmeurs d'applications du 80386. Ces derniers peuvent se limiter à ces chapitres sans y perdre beaucoup. Dans les Chapitres 5 à 7, nous expliquerons ce qui peut intéresser un programmeur de systèmes d'exploitation sur le 80386. Ces chapitres sont moins scolaires que les Chapitres 1 à 4 et décrivent précisément le fonctionnement du système d'exploitation. Les Chapitres 8 et 9 achèvent cette description : mise au point des programmes et compatibilité du 80386 avec le 8086 et le 80286. Voici une description plus précise de chaque chapitre.

Dans le **Chapitre 1**, nous introduisons brièvement la famille de processeurs 8086. Nous présentons aussi d'autres principes de base, comme l'organisation de la mémoire et les différentes représentations des nombres. Ce chapitre est surtout consacré aux types de données que traitent le 80386 et le 80387.

Dans le **Chapitre 2**, nous présentons l'état interne du 80386, les registres généraux, les registres de contrôle du processeur et les registres segments du 80386, suivis d'une introduction à l'adressage mémoire. Ensuite sont présentés le codage des instructions et l'adressage de l'espace E/S. Le chapitre se conclut sur l'état interne du 80387, les registres généraux et les registres de contrôle.

Dans le **Chapitre 3**, le plus volumineux du livre, nous présentons toutes les instructions du 80386 et du 80387 en quatre parties : les instructions travaillant sur des entiers, les instructions du type segments multiples, les instructions pour écrire un système d'exploitation et les instructions qui opèrent sur des données à virgule flottante.

Dans le **Chapitre 4**, on trouve plusieurs exemples mettant en oeuvre les instructions classiques. Ce chapitre intéressera plus particulièrement les programmeurs d'applications des 80386 et 80387.

Dans le **Chapitre 5**, nous présentons la gestion mémoire, la protection et les possibilités de gestion multitâche du 80386. Nous introduisons ici plusieurs registres et segments système plutôt que dans le Chapitre 2. On y trouve aussi la sémantique exacte du fonctionnement de la segmentation,

de l'accès mémoire, des transferts de contrôle et des commutations de tâches.

Dans le **Chapitre 6**, nous présentons les interruptions et les exceptions du 80386, ainsi que leur traitement. Cela inclut les priorités des interruptions, la façon de les masquer, et tous les détails sur les transferts de contrôle lors du traitement des interruptions. Comme au Chapitre 5, nous donnons la sémantique exacte de ces fonctionnements. Le chapitre se conclut sur les causes des exceptions du 80387 et la façon de les traiter.

Au **Chapitre 7**, nous trouvons des exemples illustrant les possibilités du 80386 quant à l'écriture de systèmes d'exploitation. Ces exemples permettent d'expliquer l'emploi des techniques de segmentation, de pagination et d'exception (voir les Chapitres 5 et 6) et des instructions du type système d'exploitation et segments multiples (voir le Chapitre 3).

Au **Chapitre 8**, nous présentons les possibilités du 80386 qui facilitent la mise au point des programmes.

Dans le **Chapitre 9** nous traitons de l'exécution de programmes 16 bits sur le 80386. Cela comprend la description du mode réel (8086), du mode virtuel 8086 et du mode protégé 16 bits.

Puisque nous ne faisons mention d'aucune caractéristique matérielle du 80386, nous conseillons la lecture des références bibliographiques 2 et 3 dans la liste qui suit.

1 *80386 Programmer's Reference Manual*, Intel Corporation, Order No 230985.

2 *80386 Hardware Reference Manual*, Intel Corporation, Order No 231732.

3 *80386 Data Sheet*, Intel Corporation, Order No 231630.

4 *80386 Assembly Language Reference Manual*, Intel Corporation, Order No 122332.

5 *80387 Data Sheet*, Intel Corporation, Order No 231920.

Après cette brève introduction, commençons notre étude du 80386. Ce travail sera complexe mais fort enrichissant, puisque nous allons étudier une famille de processeurs très populaires.

1

Concepts de base

Ce chapitre commence par un bref rappel historique de la famille des processeurs 86 développés et fabriqués par Intel Corporation. Le 80386 est le dernier-né et le plus puissant de la famille de processeurs 86. Après cet historique, nous décrirons les types de données disponibles sur le 80386. Le chapitre se conclura avec la description des types de données à virgule flottante que traite le 80387.

CHRONOLOGIE DES MICROPROCESSEURS INTEL

Le premier microprocesseur, le 4004, fut développé en 1971 par Intel et fut rapidement amélioré en devenant le 8008. Ces microprocesseurs, bien que banals par rapport aux standards actuels, étaient si singuliers qu'on les prenait à peine au sérieux en tant qu'ordinateurs. En 1974, Intel lança le 8080, un microprocesseur de deuxième génération. C'était le premier d'usage général et cette innovation fut très importante pour l'industrie du microprocesseur. En 1978, on lança la troisième génération : le 8086. Il marqua le début de la carrière des microprocesseurs en tant que véritables ordinateurs. Ce fut l'origine de la famille 86.

Le 8086 est un 16 bits, le 8080 un 8 bits et le 4004 un 4 bits. Le 8088, le petit frère du 8086, fut utilisé dans le Personal Computer d'IBM (commercialisé en 1981) et cela lança la révolution des ordinateurs personnels. Ainsi, d'autres projets aidant, l'architecture à base de 8086 devint la plus importante.

Mais cette famille ne devait pas s'arrêter au 8086. En 1982, Intel lança le 80186. Ce composant a une architecture identique à celle du 8086 mais il inclut plusieurs autres parties communes à un système dans le même composant. De plus, en 1982, on lança le 80286. Le 80286 a une *architecture englobant* celle du 8086. Cela signifie qu'il peut opérer exactement comme un 8086, mais qu'il peut aussi en faire beaucoup plus. En particulier, il permet le fonctionnement en multitâche, qui consiste à faire exécuter plus d'un programme d'application, ou tâche, en même temps. Le fonctionnement en multitâche exige du 80286 qu'il gère les protections

entre chaque tâche et zone mémoire de chaque tâche. Les 80186 et 80286 sont tous les deux des 16 bits.

Enfin, le 80386 a été introduit sur le marché en 1985. Le 80386 apporte deux améliorations majeures aux 80286 et 8086, plus beaucoup d'autres mineures. Celles-ci sont résumées dans l'Annexe A. Les deux améliorations les plus importantes sont les types de données et opérations sur 32 bits, et la technique de gestion de la mémoire par pagination, en plus de la technique de segmentation existant déjà chez tous les membres de la famille 86. On approfondira les notions de pagination et de segmentation dans les Chapitres 5, 6, et 7. De plus, le 80386 étend les possibilités de fonctionnement en multitâche du 80286. Le 80386 permet l'exécution simultanée de tâches 8086, 80286, 80386, et de systèmes d'exploitation.

Enfin, nous allons parler des coprocesseurs arithmétiques. Ces derniers ont été associés à chaque génération majeure de la famille 86. Les éléments importants de la famille 86 que sont le 8086, le 80286 et le 80386 ont pour associés respectivement les coprocesseurs arithmétiques 8087, 80287 et 80387. Ceux-ci sont étroitement couplés avec le processeur pour offrir une architecture d'ordinateur qui supporte les types de données et opérations à virgule flottante. Les petites différences qui existent entre ces générations de composants à virgule flottante sont indiquées dans l'Annexe B.

Compatibilité avec le 8086 et le 80286

Chaque génération de la famille 86 a maintenu la compatibilité avec chaque membre précédent. Ainsi, le 80386 peut exécuter tout programme d'origine 8086 ou 80286. Dans le Chapitre 9, nous expliquerons comment lancer des programmes 8086 et 80286 sur le 80386. Mais avant de parler de compatibilité ascendante, cet ouvrage va mettre en évidence les possibilités 32 bits réelles du 80386, et présenter la programmation du 80386 en tant que machine 32 bits. Ce livre ne veut pas traiter de la programmation du 8086 ou du 80286 (si vous vouliez apprendre à programmer le 8086 ou le 80286, vous liriez un autre ouvrage !).

Cependant, comme de nombreux lecteurs doivent avoir l'expérience de la programmation du 8086 ou du 80286, nous soulignerons en temps utile les différences entre les membres de la famille 86 et le 80386. L'Annexe A les résume.

LES FORMATS DES DONNEES

La finalité première d'un ordinateur est de stocker, retrouver et traiter des données. Ainsi, comprendre les types de données que traite une machine est un bon moyen de commencer à apprendre à programmer un nouvel ordinateur. Les différents types de données sont les entiers signés ou non, les BCD compactés ou non, les chaînes de caractères, les octets et les réels à virgule flottante. On trouve la plupart de ces types de données dans les ordinateurs courants. Cependant, nous allons souligner les différences qui existent entre les types de données du 80386 et ceux de la plupart des autres ordinateurs, ou, mieux, ceux du 8086 ou du 80286.

La mémoire

Avant de se lancer dans une description détaillée des types de données, il est nécessaire de comprendre l'organisation de la mémoire. Comme dans tous les ordinateurs traditionnels, la mémoire est la première source et destination de toute information. On peut considérer que la mémoire est semblable à un simple tableau séquentiel d'octets qui possèdent chacun une adresse unique. Normalement, les adresses commencent à 0 et sont incrémentées jusqu'au maximum admis par l'ordinateur. Le 80386 est une machine 32 bits et dispose d'un espace total adressable *physiquement* de 2^{32} octets ou 4 giga-octets (Go) de mémoire physique. Notez cette insistance sur l'espace adressable physique. Dans les Chapitres 2, 5 et 7, vous en apprendrez plus sur la mémoire virtuelle segmentée et paginée. Vous verrez que le maximum d'espace mémoire adressable peut être supérieur à 2^{32}.

Si on a besoin de plus de 8 bits pour représenter les valeurs dans un type de données, on utilise des octets adjacents. Un *mot* représente 2 octets adjacents et peut prendre 2^{16} valeurs différentes. Un *dmot*, ou *double-mot*, représente 4 octets consécutifs et peut stocker 2^{32} valeurs différentes.

Une conséquence simple mais importante de cette capacité multioctet apparaît quand le bit de poids fort se trouve à l'adresse la plus basse (numériquement la plus petite) ou à la plus haute. La Figure 1.1 illustre les deux possibilités pour ranger un mot dans deux octets de mémoire. La méthode "big endian", à gauche, place les 8 bits de poids fort dans l'octet de plus basse adresse du mot, et les 8 bits de poids faible dans l'octet de plus haute adresse. Si cela est contraire à vos habitudes, c'est parce que le 80386 utilise la méthode "little endian", à droite. Sur le 80386, les 8 bits de poids faible d'un mot sont placés dans l'octet de plus faible adresse (m), qui est aussi l'adresse du mot. Les 8 bits de poids fort du mot sont stockés à l'adresse la plus haute (m+1).

Nous n'essaierons pas de polémiquer sur le problème quasi religieux et si épineux du choix entre le "little endian" et le "big endian", mais vous pouvez facilement deviner notre opinion. La Figure 1.2 montre comment les octets sont rangés en mémoire pour un *dmot* dans chacun des cas. Nous le répétons : le 80386 est un microprocesseur du type "little endian".

Notation

Tout au long de cet ouvrage, nous allons introduire des notations fonctionnelles qui permettent des descriptions concises et sans ambiguïté. Un nombre suivi d'un *h* sera un nombre hexadécimal (base 16). Un nombre suivi d'un *b* sera un nombre binaire (base 2). Si le nombre n'est suivi ni du suffixe *h*, ni du suffixe b, il sera présumé décimal. Ainsi, 100 est un décimal, 100b un binaire (avec une valeur décimale de 4), et 100h un hexadécimal (avec une valeur décimale de 256). L'Annexe C donne une liste complète des équivalences binaires, hexadécimales et décimales.

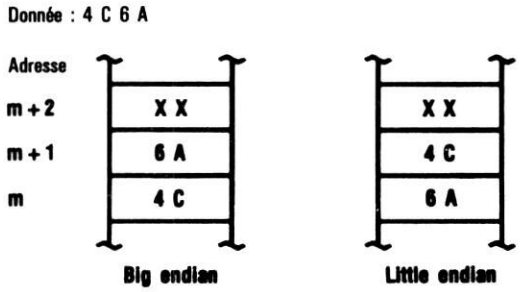


Figure 1.1 : *Big endian et little endian.*

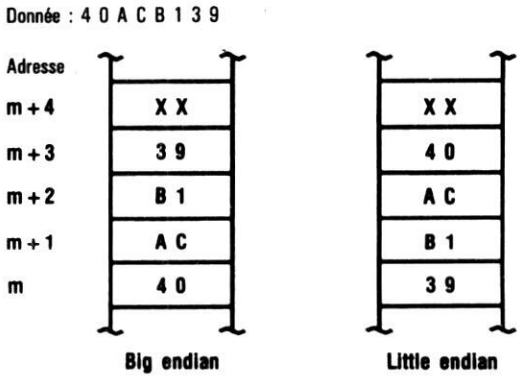


Figure 1.2 : *Big endian et little endian pour des dmots.*

Nombres non signés

Comme c'est habituellement le cas pour les ordinateurs conventionnels, le 80386 traite le type de données de base, le nombre non signé, dans des longueurs octets, mots, dmots (8, 16, et 32 bits). Les dmots sont un plus qu'offre le 80386 par rapport aux autres membres de la famille 86. L'Annexe D fournit une liste complète des puissances de 2 jusqu'à 232.

Le Tableau 1.1 définit quelques abréviations usuelles que nous utiliserons dans cet ouvrage. Ainsi, les types de données non signées peuvent représenter des nombres compris entre 0 et 256, 0 et 64Ko et 0 et 4Go pour respectivement un octet, un mot et un dmot.

La Figure 1.3 montre l'organisation des bits dans le format des données non signées. Au-dessus des trois formats de données, on trouve l'adresse mémoire commençant à l'adresse *m* qui a été vue plus haut. Le bit 0 est le bit de poids faible (LSB). Le bit de poids fort (MSB) d'un nombre non signé est respectivement le bit 7, 15 et 31 pour des représentations d'octets, de mots et de dmots.

Entiers signés

Dans le paragraphe précédent, nous avons vu le type de données de base, le nombre non signé. De tels nombres ont cependant un inconvénient majeur : ils ne peuvent pas représenter des nombres négatifs. Si vous essayez de suivre la balance de votre compte bancaire, vous savez que les nombres signés sont indispensables.

Ainsi, comme les autres microprocesseurs, le 80386 peut représenter les nombres négatifs. Il existe plusieurs façons de le faire, dont la *notation à offset*, le *signe magnitude*, *le complément à un* et le *complément à deux*. Le 80386, ainsi que tous les membres de la famille 86, emploie le complément à deux pour représenter les entiers signés. On décrira la notation à complément à deux après une brève digression où l'on va expliquer les différentes méthodes offertes pour représenter les nombres négatifs. On aura affaire ultérieurement avec certaines de ces possibilités quand les types de

Abrévation	Puis de 2	Valeur décimale
1K	2^{10}	1024
4K	2^{12}	4096
16K	2^{14}	16384
32K	2^{15}	32768
64K	2^{16}	65536
2G	2^{31}	2147483648
4G	2^{32}	4294967296

Tableau 1.1 : *Abréviations des puissances de 2.*

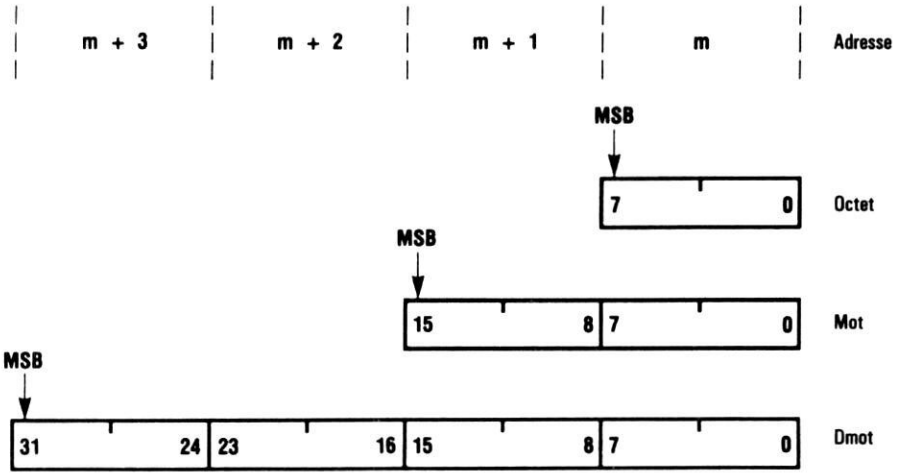


Figure 1.3 : *Octet, mot, dmot non signés.*

données à virgule flottante se présenteront. Cela vaut donc la peine de les approfondir maintenant.

Le Tableau 1.2 donne une liste des différents nombres sous chacun de ces formats. La liste n'est pas complète mais elle suffira dans le cadre de ce livre. Ce sont bien sûr des données 8 bits.

Nombres avec offset

On utilise les nombres avec offset pour exprimer l'exposant d'un nombre à virgule flottante, parce qu'on peut facilement les comparer numériquement (plus petit que ou plus grand que). Un nombre avec offset est traité en prenant la partie positive ou négative et en lui ajoutant un offset. Habituellement, l'offset est tel que le nombre le plus négatif permis dans cette représentation devient 0 et que le plus positif prend la valeur la plus importante. Le Tableau 1.2 le montre bien avec -127 étant le nombre le plus négatif, et un offset de 127 faisant de 0 la représentation de -127.

Signe magnitude

Le signe magnitude possède un bit représentant le signe (0 si positif et 1 si négatif), tandis que le reste des bits donnant un nombre non signé représente la magnitude, ou valeur absolue, du nombre. La mantisse d'un nombre à virgule flottante est exprimée dans cette notation avec un bit de signe qui indique le signe du nombre à virgule flottante complet.

Complément à un

Dans un nombre sous forme complément à un, le MSB indique le signe du nombre (0 si positif et 1 si négatif). On calcule le complément à un négatif d'un nombre positif en inversant simplement chaque bit (en incluant le MSB) de ce nombre. La notation à complément à un était couramment utilisée par les premiers ordinateurs, parce qu'elle est facile à appliquer

Décimal	Complément à deux	Complément à un	Offset =127	Signe magnitude
128	NR	NR	11111111b	NR
127	01111111b	01111111b	11111110b	01111111b
.	.	.	.	.
.	.	.	.	.
2	00000010b	00000010b	10000001b	00000010b
1	00000001b	00000001b	10000000b	00000001b
0	00000000b	00000000b	01111111b	00000000b
-0	NR	11111111b	NR	10000000b
-1	11111111b	11111110b	01111110b	10000001b
-2	11111110b	11111101b	01111101b	10000010b
.		.	.	.
.	.	.	.	.
-126	10000010b	10000001b	00000001b	11111110b
-127	10000001b	10000000b	00000000b	11111111b
-128	10000000b	NR	NR	NR

Note : NR indique que la donnée n'est pas représentable dans ce format.

Tableau 1.2 : *Formats des nombres négatifs.*

(une simple inversion de chaque bit). Cependant, elle n'est plus tellement employée aujourd'hui.

Complément à deux

La notation à complément à deux est décrite ci-après. Elle est communément choisie pour représenter les entiers signés car elle possède une propriété agréable : la simple addition binaire qui est utilisée pour les

nombres non signés opérera aussi sur deux nombres à complément à deux et ce sans modifications. Cela est important pour un processeur comme le 80386, qui traite des données à la fois non signées et complémentées à deux. On obtient le complément à deux en calculant le complément à un et en ajoutant 1 au résultat. Comme c'était le cas pour le complément à un, le MSB est le bit de signe. MSB = 0 indique un nombre positif, et MSB = 1 un nombre négatif. La Figure 1.4 montre les formes de complément à deux du 80386.

Le 80386 peut faire des calculs arithmétiques divers sur des nombres à complément à deux. Dans le Chapitre 3, nous décrirons les opérations disponibles. De plus, comme c'est habituellement le cas en notation complément à deux, les opérations arithmétiques (l'addition, par exemple) peuvent provoquer des dépassements qui indiquent un résultat erroné. Cela arrive quand on additionne deux grands nombres : le résultat est un petit

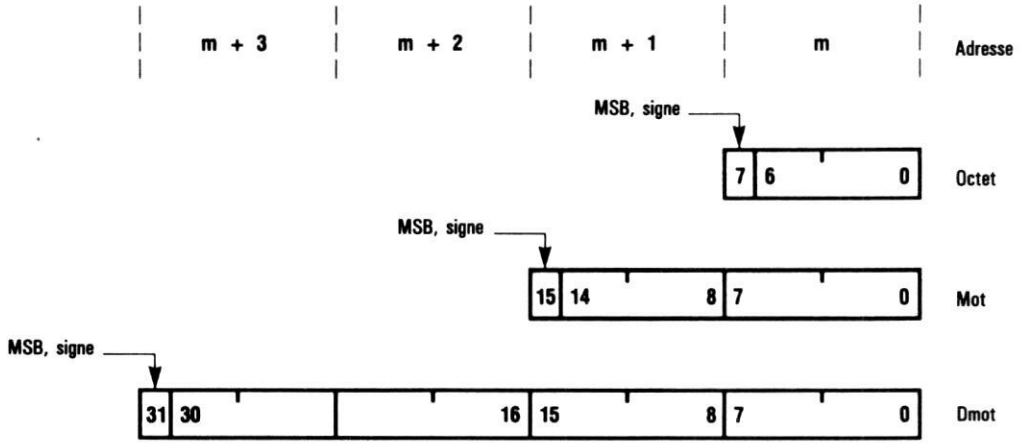


Figure 1.4 : *Octet, mot, dmot en complément à deux.*

nombre négatif. Dans le Chapitre 2, nous expliquerons comment sont traitées de telles erreurs. Ces conditions sont les mêmes que pour les autres membres de la famille 86.

Le 80386 peut traiter des types de données à complément à deux sous 8, 16, et 32 bits. Cela permet de représenter des nombres compris entre -128 et 127, -32 Ko et 32 Ko-1, et -2 Go et 2 Go-1, pour respectivement un octet, un mot, et un dmot.

Chaînes

Comme pour les membres précédents de la famille 86, le 80386 permet de travailler sur des chaînes de données. Une chaîne est une séquence continue d'octets, de mots, ou de dmots. Traiter des chaînes de dmots est une nouveauté du 80386. La longueur d'une chaîne va de 1 à 2^{32} (4Go) éléments. La Figure 1.5 montre ces trois types de chaînes.

Le 80386 possède des instructions pour déplacer des chaînes d'une région de la mémoire vers une autre, pour comparer deux chaînes, pour remplir une chaîne avec un élément déterminé, pour lire ou écrire des chaînes via les ports d'entrée/sortie, et pour chercher des chaînes de valeur déterminée.

ASCII

Une des formes les plus communes de chaînes est la chaîne de caractères ASCII. Les données ASCII sont courantes dans un système à base de 80386, puisque la plupart des données en provenance d'un terminal sont sous format ASCII (*American Standard Code for Information Interchange*). Ainsi, comme les autres membres de la famille 86, le 80386 doit pouvoir traiter l'ASCII. L'Annexe E donne un tableau ASCII complet qui comprend les entiers, les caractères alphabétiques, les caractères spéciaux et les caractères de contrôle.

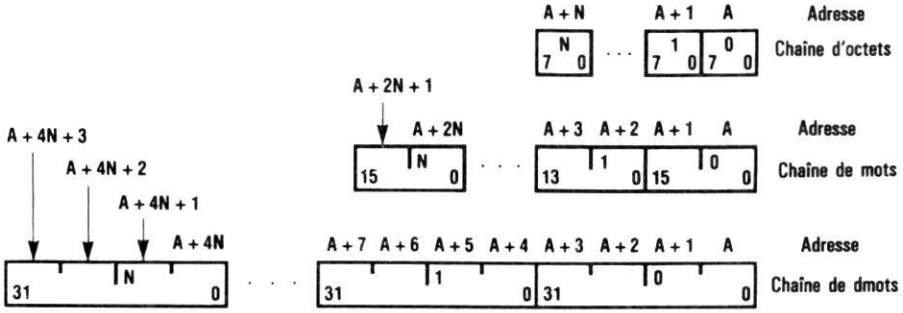


Figure 1.5 : *Chaînes d'octets, de mots et de dmots.*

Le 80386 effectue des opérations arithmétiques telles que l'addition et la division de nombres ASCII. Ces opérations sont décrites plus précisément dans le Chapitre 3.

Bits

Dans les paragraphes précédents, nous avons montré que le 80386 opère sur des données d'au moins 8 bits et souvent plus. Le traitement de chaînes de bits est une nouveauté du 80386, puisque aucun autre membre de la famille 86 ne le fait.

Le traitement du binaire est important, puisque les données peuvent souvent être représentées par un simple bit. Un exemple courant est un af-

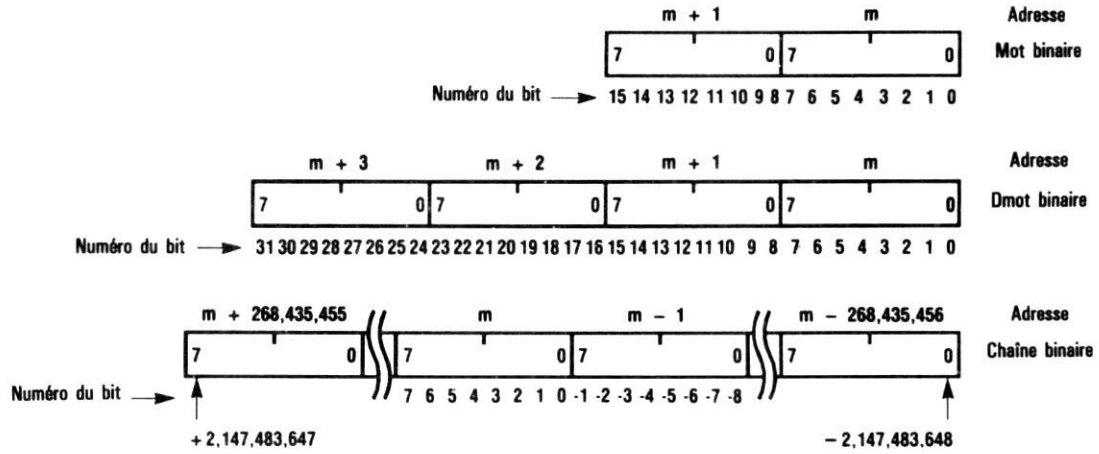
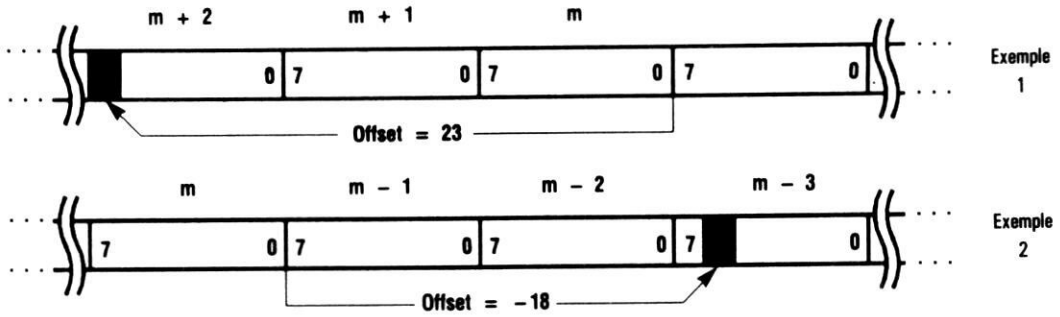


Figure 1.6 : *Types de données binaires.*

fichage bitmap. Dans ce cas, chaque *pixel*, un simple point sur l'écran, est représenté par un simple bit dans la mémoire, dont la valeur est soit 1 (le point luminophore est allumé), soit 0 (le point est éteint). Un autre exemple est celui du sémaphore où un simple bit indique s'il est libre (0) ou occupé (1). Il est possible de consacrer un octet entier à chaque pixel ou sémaphore, mais cela gaspillerait beaucoup d'espace mémoire. Effectivement, on perdrait 7 bits pour chaque élément de donnée, c'est-à-dire 87,5 % de la mémoire. Pour cette raison, le 80386 effectue aussi des opérations sur les bits.

Le 80386 reconnaît des chaînes de bits qui peuvent contenir jusqu'à 2^{32} bits indexés par un dmot signé. Ainsi, une chaîne binaire ne doit pas avoir son bit de poids faible référencé à 0. Dans un octet, le bit le plus bas est le bit 0 et le bit le plus haut, le bit 7. Cette numérotation des bits est logique dans l'esprit du 80386 en tant que machine "little endian".

On utilise un entier signé sur 32 bits pour adresser un bit particulier dans une chaîne binaire. Cet entier est appelé *bit offset* et peut avoir une



Exemple 1		Exemple 2	
Octet d'index	Bit restant	Octet d'index	Bit restant
23	23	-18	-18
÷ 8	mod 8	÷ 8	mod 8
2	7	-3	6

Figure 1.7 : *Exemples d'offset de chaînes binaires.*

valeur comprise entre -2Go et 2Go-1. Le bit offset est divisé en une adresse d'octet et un bit restant. L'adresse d'octet est le bit offset spécifié divisé par 8, et le bit désiré se trouve dans cet octet. Le bit restant est déterminé par le bit offset modulo 8. La Figure 1.7 donne deux exemples : un bit offset de 23 et un bit offset de -18 dans une chaîne binaire d'adresse N. Dans le Chapitre 4, les Listings 4.5, 4.7 et 4.8 donnent des exemples plus approfondis sur le traitement des données binaires.

BCD

Comme tout membre de la famille 86, le 80386 peut effectuer des opérations sur des types de données BCD (*Binary-Coded Decimal*). Le 80386 contient des instructions (dont on parle dans le Chapitre 3) qui lui permettent d'additionner et de soustraire des données BCD. Le Tableau 1.3 rappelle le codage BCD.

Binaire	Décimal
0000 0000b	00
0000 0001b	01
0000 0010b	02
0000 0011b	03
0000 0100b	04
0000 0101b	05
0000 0110b	06
0000 0111b	07
0000 1000b	08
0000 1001b	09
0000 1010b	0X
0000 1011b	0X
0000 1100b	0X
0000 1101b	0X
0000 1110b	0X
0000 1111b	0X
0001 0000b	10
0001 0001b	11
.	.
.	.
.	.

Note : X indique une valeur illégale dans la représentation BCD.

Tableau 1.3 : *Représentations BCD.*

Directement, sous format BCD, le 80386 ne peut traiter que des octets. Dans le Chapitre 3, nous montrerons comment sont traités les nombres BCD sur plusieurs octets. Deux chiffres BCD occupent un octet, avec le chiffre inférieur dans les bits 0 à 3, et le chiffre supérieur dans les bits 4 à 7. Le BCD non compacté range un chiffre BCD par octet, dans les bits 0 à 3.

TYPES DE DONNEES A VIRGULE FLOTTANTE

Le coprocesseur arithmétique du 80386, le 80387, traite les types de données à virgule flottante. Il est très semblable aux 8087 et 80287 et les éventuelles différences sont notées dans l'Annexe B.

Introduction à la notion de virgule flottante

Nous avons parlé de types de données variées : entiers signés ou non, chaînes, chaînes binaires, ASCII et BCD. Aucun d'eux ne peut représenter des nombres à partie décimale - par exemple, un nombre décimal tel que 1,21F. Aucun ne peut représenter des constantes physiques essentielles telles que 3,1416, $8{,}854*10^{-14}$, $6{,}02*10^{23}$, 9,81, $9{,}11*10^{-31}$, ou $1{,}38*10^{-23}$. Des nombres qui possèdent une partie entière et une partie décimale, comme dans les exemples précédents, sont mathématiquement classés comme nombres réels ou nombres rationnels. Dans cet ouvrage, on les considérera simplement comme des *nombres réels*, même si cela n'est pas très précis mathématiquement. Ainsi, dans le cas présent, un nombre est réel s'il peut être écrit avec une partie entière et une partie décimale. Notons que l'ensemble des nombres entiers défini dans le précédent paragraphe est un sous-ensemble de l'ensemble des nombres réels, dans lequel la partie décimale est toujours nulle.

Un ordinateur représente les nombres au moyen d'un nombre fixe de bits. Ainsi, un ordinateur ne peut pas quantifier de manière précise tous les nombres réels. En fait, il en représente un très petit sous-ensemble. Heu-

reusement cependant, ce sous-ensemble couvre la majorité des cas pratiques importants, si bien que l'utilisation de ce sous-ensemble n'entraîne qu'une perte d'information minime. La représentation du type de données associé aux nombres réels est appelée *virgule flottante. Nombres réels* est un terme mathématique qui décrit l'ensemble des nombres, et *virgule flottante* décrit le type de données représentant un sous-ensemble des nombres réels dans un ordinateur.

Considérons maintenant plusieurs possibilités pour la notation à virgule flottante, ce qui va nous permettre de comprendre le pourquoi de cette notation. On peut opérer par exemple comme on le fait avec l'argent. Vous pouvez facilement multiplier tous vos nombres par 100 et convertir ainsi toutes les informations en entiers. Les chiffres 23,41 seraient alors lus du clavier et convertis à partir des caractères ASCII en une représentation interne de 2341. Toutes les opérations internes seraient assurées sous cette forme, et, au niveau des résultats, les nombres seraient donc divisés par 100 avant affichage. Cette méthode présente plusieurs inconvénients, en particulier le manque de précision. Par exemple, si vous percevez un intérêt de 7 % sur 23,41 F, obtenez-vous 25,04 F ou 25,05 F ? Sans précautions supplémentaires, vous aurez une perte de 0,01 avec 25,04 F.

Il existe une autre solution pour le traitement des nombres : la virgule fixe. Imaginons qu'on assigne les 3 bits de poids faible d'un octet à la partie décimale. Par exemple :

$$10110110b = 2^4 + 2^2 + 2^1 + 2^{-1} + 2^{-2} = 22.75$$

C'est valable, mais on obtient ainsi un petit ensemble fixe de nombres représentables (de 0 à 2^6) par incréments de 2^{-3}.

La solution à ces problèmes est apportée par la représentation à virgule flottante. On obtient ainsi un point décimal mobile. Une partie de la représentation des données est consacrée à la spécification de la position du point décimal, et le reste représente les bits significatifs des données. Comme vous le verrez, cela permet d'élargir de façon notable la quantité

de nombres représentables tout en conservant une bonne précision sur toute l'échelle.

Vous pouvez représenter des nombres à virgule flottante à l'aide de deux entiers : le premier pour exprimer les bits significatifs des données, et le second pour renseigner sur la position du point décimal. On peut traiter les opérations à virgule flottante sur ces paires d'entiers en utilisant une séquence d'instructions sur des entiers telles que additions, multiplications et décalages. Cette émulation logicielle de la virgule flottante peut être relativement lente, ce qui rend souhaitable l'emploi d'un matériel spécialisé dans la représentation à virgule flottante. Intel utilise un coprocesseur arithmétique, le 80387, pour travailler dans cette notation. Le reste du chapitre traitera des types de données du 80387.

La norme virgule flottante IEEE

Avant de voir en détail les types de données du 80387, nous allons faire quelques commentaires sur la norme virgule flottante IEEE. Avant 1979, beaucoup d'unités centrales et de mini-ordinateurs traitaient les types de données à virgule flottante. Cependant, il n'existait pas de norme pour cette représentation. En fait, vous pouviez exécuter un programme FORTRAN sur un ordinateur A et ultérieurement sur un ordinateur B et obtenir des résultats sensiblement différents.

Au cours du développement du 8087 (le coprocesseur arithmétique du 8086 et 8088), Intel s'était déjà intéressé à l'élaboration d'une norme pour la représentation à virgule flottante, et tout spécialement d'un standard que le 8087 puisse employer. Juste après la présentation de cette norme par Intel, l'IEEE confia à une commission le soin de développer une norme pour l'arithmétique sur des données à virgule flottante. Le résultat de plusieurs années de travail de la part des membres de cette commission est le Projet 10.0 de l'IEEE Task P754, *A Standard for Binary Floating-Point Arithmetic*. Voici les grandes lignes de cette norme telle qu'elle est établie dans le projet :

1 Faciliter le transfert des programmes existants à partir d'ordinateurs divers vers ceux qui suivent cette norme.

2 Augmenter les possibilités et la sécurité offertes aux programmeurs qui, sans qualification sur les méthodes numériques, pourront produire aisément des programmes sophistiqués.

3 Encourager des experts à développer et distribuer des programmes numériques solides et efficaces, portables sur tout ordinateur se conformant à la norme, et cela avec des modifications mineures et une simple recompilation.

On commence à accepter la norme IEEE dans l'industrie. Non seulement Intel produit des circuits qui suivent ce standard, mais beaucoup d'autres microprocesseurs, mini-ordinateurs, et unités centrales en font autant. Le 80387 suit cette norme. En résumé, le 80387 permet une grande étendue de données avec des résultats fiables et une bonne performance.

Que faire sans le 80387 ?

Il est malheureusement possible que votre système ne possède pas de 80387. Rappelez-vous en effet que le 80386 et le 80387 sont des composants distincts. Un système qui contient un 80386 peut ne pas avoir de coprocesseur arithmétique. Ainsi qu'il en a été fait mention dans l'introduction, plusieurs solutions sont possibles.

1 Vous pourriez ajouter un 80387 dans votre système. Si vous envisagez de faire beaucoup de calculs numériques, c'est probablement un choix judicieux. Presque tous les fabricants d'ordinateurs proposent le 80387 en option si votre matériel n'a pas été livré avec.

2 Votre matériel doit disposer d'un module mathématique d'émulation (c'est l'émulation logicielle dont on a parlé auparavant). Si vous n'en n'avez pas, vous pouvez en acquérir un ou l'écrire vous-même.

Donc, même si vous n'avez pas de coprocesseur, si vous voulez faire de l'arithmétique à virgule flottante, il vous faudra utiliser un module d'émulation. Dans ce cas, vous devrez employer les types de données à virgule flottante et les instructions destinées au 80387.

Votre système peut aussi comporter un 80287 (le coprocesseur arithmétique du 80286) au lieu du 80387. Le 80386 peut travailler avec les deux. Si vous possédez un 80287, reportez-vous à l'Annexe B qui résume les différences entre le 80287 et le 80387.

Formats des données

Le 80387 traite sept types de données qui sont résumés dans le Tableau 1.4. Notez comment un nombre conséquent peut être représenté. Rappelez-vous que le nombre le plus important qui peut être traité par le 80386 est 2^{32} ou $4{,}29*10^9$ en décimal. Ainsi, nous avons étendu de façon *substantielle* la plage de variations des types de données dont nous parlions plus tôt.

Notez aussi que le 80387 traite les types de données entières. Cela peut paraître trivial puisque, à l'exception du type entier long sur 64 bits, ils sont déjà gérés par le 80386. Mais, si un calcul s'effectue à la fois sur des données réelles et entières, le calcul entier doit être traité par le 80387. Le transfert des données entre le 80387 et le 80386 ne serait alors pas exigé.

Types de données entières

Les types de données entières sont représentés dans la notation complément à deux utilisée pour les types de données entières du 80386. Tous les types entiers sont communs aux deux circuits, avec cependant une exception : le 80386 gère des entiers 8 bits et le 80387 des entiers 64 bits. Le tableau suivant résume les types de données permis par le 80387 et le

Types de données	Bits	Chiffres Significatifs	Echelle Approximative
Mot entier	16	4	-32768 <= X <= 32767
Entier court	32	9	$-2*10^{9}$ <= X <= $+2*10^{9}$
Entier long	64	18	$-9*10^{18}$ <= X <= $+9*10^{18}$
BCD compacté	80	18	-99..99 <= X <= +99..99
Réel court	32	6-7	$-3,39*10^{-38}$ <= X <= $3,39*10^{38}$
Réel long	64	15-16	$-1,80*10^{-308}$ <= X <= $1,80*10^{308}$
Réel tempo.	80	19	$-1,19*10^{-4932}$ <= X <= $1,19*10^{4932}$

Tableau 1.4 : *Types de données supportés par le 80387.*

80386. Dans ce tableau, NR indique qu'il n'y a pas de représentation possible dans le processeur.

Bits	80386	80387
8	Octet signé	NR
16	Mot signé	Mot entier
32	Dmot signé	Entier court
64	NR	Entier long

La Figure 1.8 nous montre les types de données entières. Ces trois types de données peuvent représenter des nombres compris entre -32768 et 32767, $-2,147*10^{9}$ et $2,147*10^{9}$, et $-9,233*10^{18}$ et $9,233*10^{18}$ pour des formats de mots, d'entiers courts et d'entiers longs.

BCD

Encore une fois, puisque le 80386 traite le BCD, pourquoi utiliser le 80387 ? Le 80387 utilise ce type de données pour les mêmes raisons qui

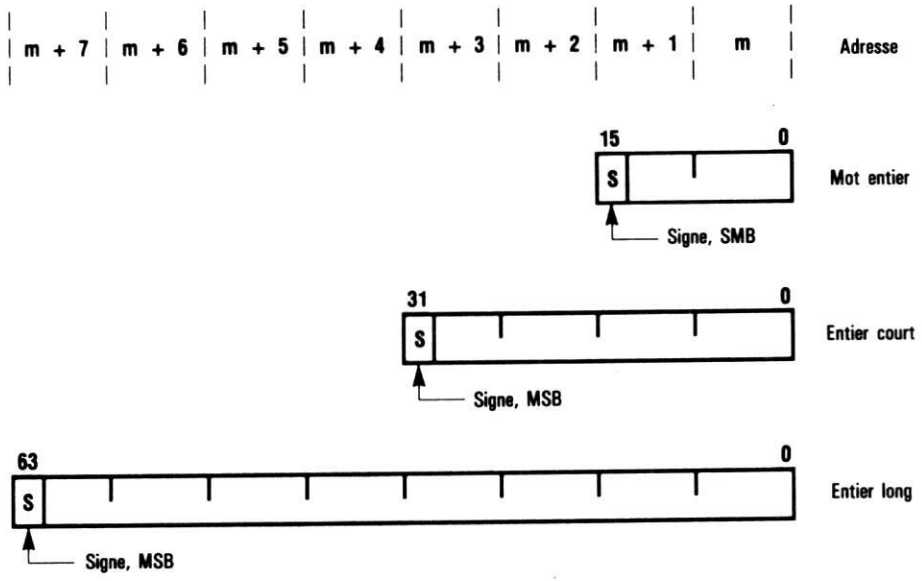
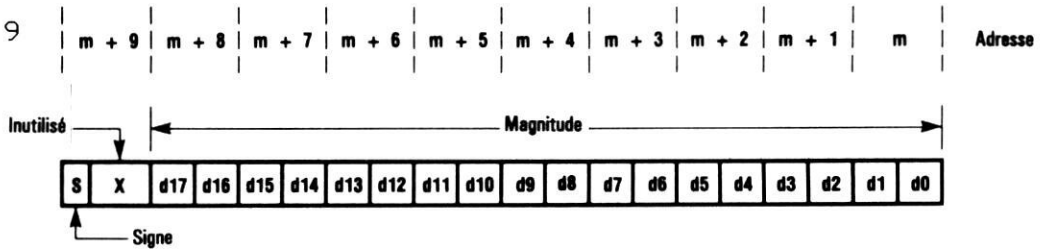


Figure 1.8 : *Types de données entières du 80387 (complément à deux).*

l'ont conduit à employer les types de données entières. Le type BCD du 80387 est le type décimal compacté sur 80 bits, qui consiste en 18 chiffres décimaux et un bit de signe. C'est ce que montre la Figure 1.9. Pourquoi le BCD compacté s'arrête-t-il à 18 chiffres et laisse-t-il 7 bits inutilisés dans cette représentation ? Avec ces 18 chiffres, il répond à la norme COBOL (un des tout premiers langages à utiliser le BCD), et il n'y avait pas de raison de continuer au-delà de 18 bits.

Les formats réels

Les formats réels et les formats à virgule flottante dont on a parlé auparavant sont exactement les mêmes. La Figure 1.10 donne le format réel général. Il est composé de trois parties : mantisse, exposant et signe.



d_i

Décimal	Bits
0	0 0 0 0
1	0 0 0 1
.	.
.	.
.	.
8	1 0 0 0
9	1 0 0 1
Inutilisés	1 0 1 0
	.
	.
	.
	1 1 1 1

Figure 1.9 : *Type de données BCD du 80387.*

La Figure 1.11 décrit en détail la place des bits dans les trois types de données. Commençons par le plus facile. C'est le *signe* : un simple bit de signe. Si le bit est à 1, le nombre est négatif. Si le bit est à 0, le nombre est positif. On avait déjà parlé de ce codage du signe dans le chapitre traitant des entiers signés.

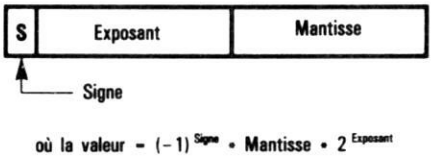


Figure 1.10 : *Format réel général.*

Le tableau suivant indique quels bits sont utilisés pour chaque partie des trois formats de données réelles.

Type de données	Total	Signe	Exposant	Mantisse
Réel court	32	1	8	23
Réel long	64	1	11	52
Réel temporaire	80	1	15	64

La mantisse représente les bits significatifs du nombre. Une brève remarque vous convaincra qu'il est possible de stocker un même nombre de différentes façons. Par exemple (en décimal), $100{,}0*10^{-2}$, $10{,}0*10^{-1}$, $1{,}0*10^{0}$, $0{,}1*10^{1}$, $0{,}01*10^{2}$, $0{,}001*10^{3}$ représentent tous 1. Maintenant, afin de permettre des calculs avec un maximum de précision, les résultats sont *normalisés*. En d'autres termes, les nombres ont toujours leurs exposants tels que le bit le plus significatif (le plus à gauche) de la mantisse soit à 1, avec la virgule binaire juste "à droite" de ce bit à 1. Ainsi, le résultat correctement normalisé pour l'exemple précédent est $1{,}0*10^{0}$.

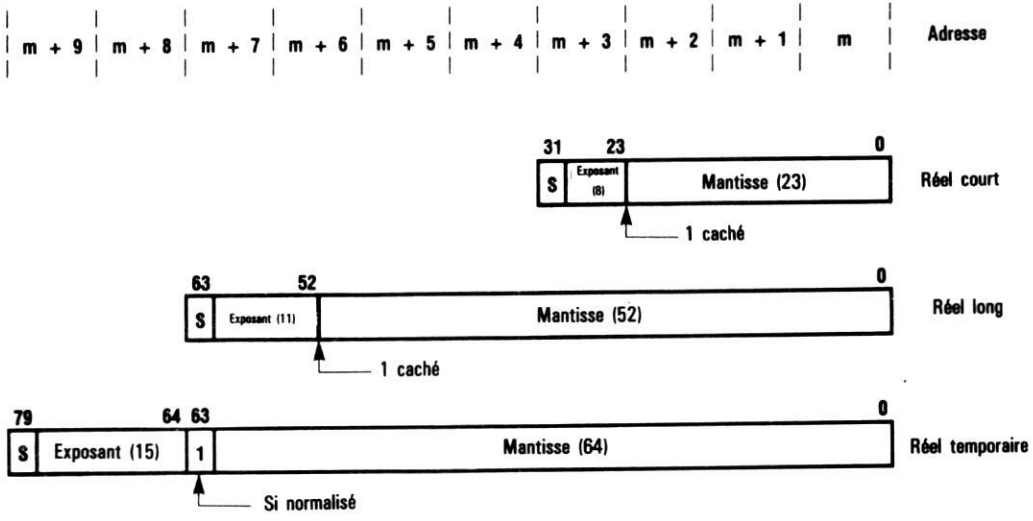


Figure 1.11 : *Types de données réelles du 80387.*

Si vous avez toujours 1,XXXXX*2^n (X étant un 1 ou un 0), vous savez que le bit à gauche de la virgule binaire est toujours à 1. Le 80387 élimine ce bit dans les formats réel court et réel long pour obtenir un bit supplémentaire de précision.

La partie exposant contient la puissance de deux qui est nécessaire pour obtenir le bon résultat à partir de la mantisse. L'exposant est stocké sous forme offset (se reporter au chapitre traitant des entiers signés). Ceci permet de faciliter les comparaisons numériques, puisqu'une grande valeur

numérique est toujours un grand nombre - ce n'est pas le cas dans les autres représentations des réels signés. Les offsets sont 127, 1023, et 16383 pour les formats réel court, réel long et réel temporaire. Ainsi, un exposant de 10000000b (réel court) est en vérité 2^1.

Maintenant que nous avons décrit les trois parties du format à virgule flottante, nous sommes prêts pour le résumé de la Figure 1.12 et quelques exemples. Comparez avec la Figure 1.10.

A la Figure 1.12 on peut représenter des nombres plus importants, positifs ou négatifs. La *précision* est aussi sensiblement améliorée. On parlera ultérieurement des *dénormalisés*. Tous les exemples suivants sont des réels courts. Leur nombre de bits est déjà suffisant pour entraîner des difficultés de compréhension.

Signe	Exposant	Mantisse	Somme	Valeur
0	10001110b	000100...0b	$(1 + 2^{-4})*2^{15}$	34816
1	01111100b	011000...0b	$-(1 + 2^{-2} + 2^{-3})*2^{-3}$	-0,17188
0	01111111b	000000...0b	$(1)*2^0$	1

Réels temporaires

Jusqu'à maintenant, la différence entre les trois types de formats réels résidait dans leur nombre de bits. C'est presque vrai, mais le format réel temporaire demande quelques explications.

1 Le format réel temporaire est une forme interne utilisée par le 80387. Quel que soit le type de données que vous lui attribuez (entier court, BCD, réel court, etc.), le 80387 le convertit immédiatement en format réel temporaire.

2 Le format réel temporaire n'a pas de bit caché. Ainsi, le bit 63 d'un nombre réel temporaire est toujours à 1 dans un nombre normalisé.

1 caché

↓ Mantisse

$(-1)^{\text{Signe}} \cdot (1.S_{23}S_{22}S_{21}...S_0) \cdot 2^{(\text{Exposant} - 127)}$ **Réel court**

Échelle : $\pm\ 3.39 \cdot 10^{38}$

Précision : $\pm\ 1.18 \cdot 10^{-38}$ (Dénormalisé : $1.40 \cdot 10^{-45}$)

1 caché

↓ Mantisse

$(-1)^{\text{Signe}} \cdot (1.S_{51}S_{50}S_{49}...S_0) \cdot 2^{(\text{Exposant} - 1023)}$ **Réel long**

Échelle : $\pm\ 1.80 \cdot 10^{308}$

Précision : $\pm\ 2.23 \cdot 10^{-308}$ (Dénormalisé : $4.94 \cdot 10^{-324}$)

$(-1)^{\text{Signe}} \cdot (1.S_{63}S_{62}S_{61}...S_0) \cdot 2^{(\text{Exposant} - 16383)}$

Échelle : $\pm\ 1.19 \cdot 10^{4932}$ **Réel temporaire**

Précision : $\pm\ 3.36 \cdot 10^{-4932}$ (Dénormalisé : $3.65 \cdot 10^{-4951}$)

Figure 1.12 : *Nombres réels courts, longs, temporaires.*

Pourquoi ce format est-il employé uniquement par le 80387 ? C'est pour optimiser la précision et l'étendue des calculs. Même si vous effectuez un calcul sur des réels courts, vous pouvez avoir des résultats intermédiaires qui excèdent de beaucoup l'échelle normale d'un réel court, même si le résultat final est un réel court.

Il n'y a pas de bit caché, puisque le 80387 utilise le format réel temporaire de façon interne et que le bit *est* utilisé pour le calcul. Quand les réels longs ou courts donnés au 80387 sont convertis en réels temporaires, un bit explicite 1 est inséré.

Cas particuliers

Les Tableaux 1.5 à 1.7 montrent toutes les possibilités de représentation pour les trois formats. Nous allons parler des cas particuliers sui-vants : zéros, infinis, dénormalisés, pseudo-dénormalisés, et NaN. Notez que tous les nombres de ces trois tableaux sont binaires.

Le *zéro* est simplement un exposant (avec offset) de 00...00b et une mantisse de 00...00b dans les trois formats. Remarquons qu'un exposant nul est réservé. Cela signifie que l'exposant (avec offset) nul ne peut être employé pour représenter un nombre réel normal. On peut représenter un zéro positif et un zéro négatif. La division par zéro est le seul cas où le 80387 distingue les deux formes de zéro (Voir le Chapitre 3 pour l'instruction FDIV).

Signe	Exposant	Mantisse	Catégorie
0/1	11...11	11...11	NaN silencieux
0/1	11...11	.	
0/1	11...11	10...00	
0/1	11...11	01...11	NaN signalés
0/1	11...11	.	
0/1	11...11	00...01	
0/1	11...11	00...00	Infinis
0/1	11...10	11...11	Normalisés
0/1	.	.	
0/1	00...01	00...00	
0/1	00...00	11...11	Dénormalisés
0/1	00...00	.	
0/1	00...00	00...01	
0/1	00...00	00...00	Zéros

Tableau 1.5 : *Représentations des réels courts et longs.*

Second cas particulier : les *infinis*. L'exposant (avec offset) à valeur 11..11b est réservé. Comme ci-dessus, cela signifie qu'il ne peut pas être utilisé pour un nombre réel normal. Si la mantisse est égale à zéro, c'est l'infini (positif ou négatif selon le cas). Nous reparlerons bientôt des infinis .

Les *dénormalisés* sont un cas particulier dans la représentation des très petits nombres. Ils permettent d'avoir une perte de précision progressive. Beaucoup d'ordinateurs ne le font pas et tronquent brutalement, au contraire du 80387. Cela signifie que lorsqu'ils atteignent le plus petit nombre normalisable, la représentation suivante est égale à zéro. Au contraire, le 80387 utilise des représentations qui ne sont pas normalisées.

Signe	Exposant	Mantisse	Catégorie
0/1	11...11	111...11	NaN silencieux
0/1	11...11	.	
0/1	11...11	110...00	
0/1	11...11	101...11	NaN signalés
0/1	11...11	.	
0/1	11...11	100...01	
0/1	11...11	100...00	Infinis
0/1	11...10	111...11	Normalisés
0/1	.	.	
0/1	00...01	100...00	
0/1	00...00	111...11	Pseudo-dénormalisés
0/1	00...00	.	
0/1	00...00	100...00	
0/1	00...00	011...11	Dénormalisés
0/1	00...00	.	
0/1	00...00	000...01	
0/1	00...00	000...00	Zéros

Tableau 1.6 : *Représentations des réels temporaires.*

Cela entraîne une perte de précision mais étend de façon considérable la représentation des très petits nombres. Il est évident qu'une troncature progressive est une meilleure solution qu'une troncature abrupte, même si on perd un peu de précision.

Normalement, les nombres doivent être normalisés (décalés à gauche jusqu'à ce que le bit le plus significatif de la mantisse soit égal à 1). Cependant, les dénormalisés n'ont pas le bit le plus significatif de la mantisse à 1. L'exposant 00...00 est une représentation particulière avec des valeurs de 2^{-126}, 2^{-1022}, 2^{-16382} pour des réels courts, longs et temporaires. Ceci est bien particulier puisque l'exposant 00...01 représente aussi 2^{-126}, 2^{-1022} et 2^{-16382}. Les dénormalisés entraînent toujours une perte de précision ; d'un autre côté, un nombre normalisé serait impossible.

Comme vous pouvez le voir sur la Figure 1.12, l'emploi des dénormalisés autorise la représentation des très petits nombres (on parle parfois de *petitesse*).

Signe	Exposant	Mantisse	Catégorie
0/1	11...11	011...11	Pseudo-NaN
0/1	11...11	.	
0/1	11...11	000...01	
0/1	11...11	000...00	Pseudo-infinis
0/1	11...10	011...11	Anormalisés
0/1	.	.	
0/1	00...01	000...01	
0/1	11...10	000...00	Pseudo-zéros
0/1	.	.	
0/1	00...01	000...00	

Tableau 1.7 : *Représentations des réels temporaires non permis.*

Les *pseudo-normalisés* sont admis par le 80387 mais ne peuvent jamais être générés par le 80387. Généralement, le bit le plus significatif de la mantisse des dénormalisés est égal à 0. Les pseudo-normalisés ont ce bit égal à 1, comme on le voit au Tableau 1.6. Ils sont peu communs puisqu'ils peuvent être représentés sous forme normalisée mais ne le sont pas. Ceci est d'autant plus vrai que l'exposant particulier 00...00 a la même valeur que l'exposant "normalisé" 00...01. De plus, la mantisse d'un dénormalisé est normalisée ; c'est-à-dire que le MSB est égal à 1. Par conséquent, tous les pseudo-dénormalisés peuvent être représentés comme des normalisés.

Le cas particulier suivant est celui des NaN : NaN pour *Not a Number* (pas un nombre). Il existe deux formes de NaN : le signalé et le silencieux. Un NaN signalé entraîne une exception du type opération non valide lorsqu'on en utilise un dans une opération. Un NaN silencieux ne génère pas cette exception. D'où la terminologie de signalé et de silencieux. Nous expliquerons le fonctionnement des exceptions dans le Chapitre 6.

Le NaN signalé a le bit le plus significatif de la mantisse égal à 0. Ceci est vrai sauf pour un réel temporaire, où le second bit le plus significatif est un 0 et le bit le plus significatif (le bit caché) un 1. Par exemple, le NaN signalé peut être employé pour s'assurer qu'un programme initialise bien toutes les variables avant de se lancer. Le programmeur ou le compilateur pourrait initialiser chaque variable d'un programme à un NaN signalé pour qu'une exception survienne si une valeur non initialisée est employée. Un NaN peut avoir n'importe quelle valeur dans le reste de la partie décimale, ce qui permet de l'utiliser pour stocker des informations : par exemple, où et pourquoi le NaN a été créé, si on le désire. Le 80387 ne génère jamais de NaN signalés.

Le bit le plus significatif de la mantisse d'un NaN silencieux est égal à 1 sauf dans le cas d'un réel temporaire où le second bit le plus significatif est 1 et le bit le plus significatif (le bit caché) est aussi 1. Un NaN silencieux apparaît lors d'une exception du type opération non valide. Quand cela se produit, le résultat est indéterminé (voir plus bas). Dans tous les cas (sauf pour FCOM, FIST, et FBSTP dont on parle dans le Chapitre 3), les NaN silencieux et signalés sont préservés par toutes les opérations. Il y a plusieurs cas particuliers qui se présentent avec des NaN :

1 Toute opération qui générerait une exception du type opération non valide mais qui implique des NaN produira un NaN silencieux indéfini. On donne les entiers, BCD et réels indéfinis ultérieurement.

2 Quand les opérations impliquent des NaN silencieux et signalés (QNaN et SNaN), le QNaN (NaN silencieux donc) sera présenté comme résultat.

3 Les opérations entre deux SNaN donnent le plus grand SNaN après conversion en QNaN. La conversion d'un SNaN en QNaN s'effectue simplement en mettant à 1 le bit le plus significatif de la mantisse.

4 Les opérations entre deux QNaN produisent le plus grand QNaN.

5 Les opérations entre un SNaN et un nombre normal aboutissent à un SNaN après conversion en QNaN.

6 Les opérations entre un QNaN et un nombre normal donnent un résultat QNaN.

Cette description s'avère un peu confuse. Le Tableau 1.8 résume les opérations sur les NaN. Ce tableau se réfère à des opérations avec les opérandes 1 et 2 (op1 et op2).

Les *indéfinis* sont un cas particulier de NaN silencieux (décrits ci-dessus). Chaque type de données admis par le 80387 (mot, entiers court et long ; décimal compacté et réels court, long et temporaire) a une représentation indéfinie unique. Cette représentation est générée par le 80387 toutes les fois qu'une exception du type opération non valide survient et que les opérandes d'entrée ne sont pas des NaN. On donne dans le Chapitre 3 les opérations qui génèrent des exceptions du type opération non valide ; le processus exact de génération de ces exceptions est expliqué dans le Chapitre 6. Le Tableau 1.9 résume les codages indéfinis pour chaque type de données. Le bit de gauche du codage est le bit le plus significatif. Pour la représentation des réels, de gauche à droite, les champs sont le signe, l'exposant et la mantisse.

		op2		
		Q	**S**	**R**
	Q	>Q	Q	Q
op1	**S**	Q	CQ(>S)	CQ(S)
	R	Q	CQ(S)	QI

Clés

Q	NaN silencieux
S	NaN signalé
R	Réel normal (pas un NaN, nombre légal)
>Q	Le plus grand NaN silencieux de op1 et op2
>S	Le plus grand NaN signalé de op1 et op2
CQ(X)	Conversion de X en QNaN
QI	Le NaN silencieux par défaut - un réel indéfini

Tableau 1.8 : *Opérations impliquant des NaN.*

Format	Bits	Codage
Mot entier	16	100...000b
Entier court	32	100...000b
Entier long	64	100...000b
BCD	80	1 1111111 1111 1111 XX...XXb
Réel court	32	1 11...11 10...00b
Réel long	64	1 11...11 10...00b
Réel tempo.	80	1 11...11 110...00b

Note : X indique "quelconque".

Tableau 1.9 : *Codages indéfinis.*

Notez bien que, quand nous parlions de types de données BCD et entières, nous avons omis les indéfinis. Quand un indéfini doit être généré pour un type de donnée entière, c'est la représentation la plus négative qui est générée. Pour les BCD, par contre, c'est le codage contraire qui est retenu.

Les indéfinis peuvent être chargés et stockés pour les formats réels. Puisqu'un réel indéfini est un NaN silencieux, il n'y a pas d'exception quand on charge un indéfini. Pour les entiers et les BCD, le chargement n'est pas possible. Pour les entiers, les indéfinis chargés sont traités comme les plus grands nombres négatifs. Pour les BCD, le résultat du chargement est indéterminé.

Le Tableau 1.7 donne les représentations des réels temporaires non admis par le 80387 (certains cependant le sont par le 80287). Ces formats (pseudo-zéro, anormalisé, pseudo-infini, et pseudo-NaN) entraîneront des exceptions du type opération non valide si le 80387 les rencontre.

Exceptions

Les exceptions signalent qu'une erreur quelconque est détectée pendant une opération. Par exemple, quand vous faites démarrer votre voiture, un bip peut se faire entendre si votre porte est ouverte ou si vos ceintures de sécurité ne sont pas bouclées. Dans les deux cas, une exception est détectée.

Il existe plusieurs exceptions possibles. La perte de précision peut être considérée comme une exception, cela dépend de l'état de la machine. Nous avons aussi parlé des NaN signalés. Travailler sur un NaN signalé entraînera aussi une exception. Il existe beaucoup d'autres exceptions possibles et pour la plupart, nous n'avons pas encore discuté de leur contexte. Les conditions des exceptions et leur prise en compte seront détaillées dans le Chapitre 6.

2

Etat de la machine et adressage mémoire

Dans le Chapitre 1, nous avons présenté les types de données reconnus par le 80386 et le 80387, et comment ils sont stockés en mémoire. Les instructions du 80386 et 80387 opèrent directement sur ces types de données de base. Chaque instruction détermine l'*opération* qui doit être traitée, et l'emplacement des données en entrée et en sortie qui participent à l'opération. Les données en entrée/sortie sont appelées *opérandes*.

Dans ce chapitre, nous allons indiquer où les types de données de base peuvent être stockés et récupérés par le 80386 en tant qu'opérandes. Au sein du 80386, un opérande peut spécifier des données dans un *registre*, ou à l'extérieur du processeur, dans la *mémoire principale* ou *mémoire E/S*, ou directement dans une instruction comme pour une *constante immédiate*. La mémoire principale représente un important espace de stockage pour les opérandes, mais on y accède plus lentement qu'aux registres ou à des constantes immédiates. En outre, pour certaines opérations, il faut placer certains opérandes dans des registres plutôt que dans la mémoire.

Dans ce chapitre, nous allons d'abord parler des trois types de registres dont peuvent disposer les programmeurs d'applications : pour le stockage de nombres 32 bits, pour le contrôle d'exécution du processeur, et pour l'adressage de segments mémoires. Puis, nous décrirons la méthode employée pour former les adresses mémoire et le codage binaire des opérandes dans le jeu d'instructions. Ensuite, nous introduirons la notion d'espace mémoire E/S. Pour finir, nous examinerons les registres disponibles sur le coprocesseur à virgule flottante 80387 pour le stockage des types de données à virgule flottante et pour le contrôle des opérations sur ces types de données.

LES REGISTRES

Le programmeur d'applications dispose d'un jeu de 16 registres répartis en trois catégories (voir la Figure 2.1). Les programmeurs système disposent d'autres registres : Le Chapitre 5 contient une étude des registres qui gèrent la mémoire, et le Chapitre 8 une description des registres permettant la mise au point des programmes.

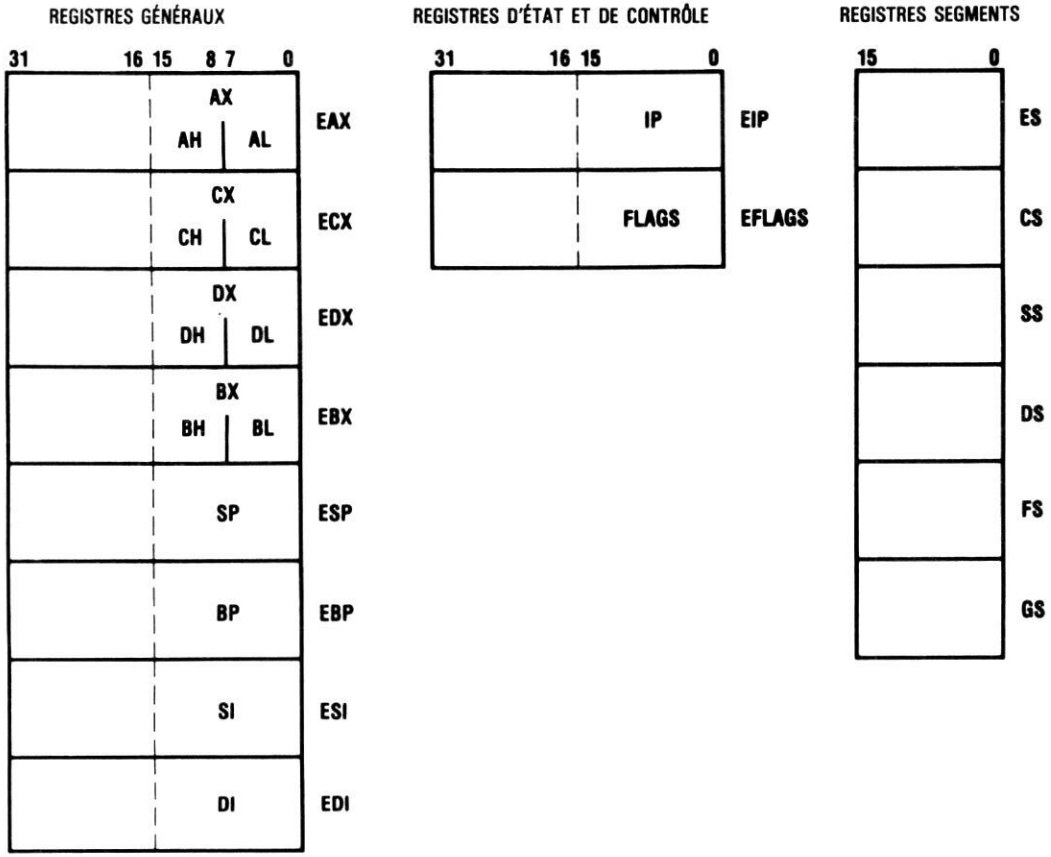


Figure 2.1 : *Les registres du 80387.*

Les trois catégories sont les suivantes :

1 Huit registres 32 bits d'usage général pour les opérations arithmétiques et logiques, et pour les parties base et index des adresses mémoire.

2 Deux registres 32 bits pour le contrôle du processeur.

3 Six registres 16 bits pour l'adressage des segments mémoire. Chaque registre permet un accès immédiat à un segment de la mémoire à un instant donné. Nous parlerons bientôt des segments dans ce chapitre et plus précisément au Chapitre 5.

Le jeu de registres du 80386 est compatible de façon ascendante avec les jeux de registres des autres processeurs de la famille 86. Quand un registre 16 bits est étendu à un registre 32 bits du 80386, on donne simplement à ce dernier l'ancien nom du registre 16 bits avec pour préfixe E (pour étendu). Par exemple, le registre 16 bits AX du 8086 devient le registre 32 bits EAX du 80386, le registre 16 bits IP (pointeur d'instruction) devient le registre 32 bits EIP sur le 80386, etc.

Les registres généraux

Le 80386 comprend huit registres 32 bits généraux, qu'on utilise pour des opérations arithmétiques telles que addition, soustraction, multiplication et division, et aussi pour former les adresses mémoire, comme cela sera expliqué ultérieurement dans ce chapitre. Ces huit registres se nomment EAX, ECX, EDX, EBX, ESP, EBP, ESI, et EDI (voir la Figure 2.1).

On peut accéder aux 16 bits de poids faible de ces registres de façon indépendante, comme des registres 16 bits, qui sont appelés AX, CX, DX, BX, SP, BP, Si et DI. Ce sont les huit registres 16 bits d'usage général des précédents processeurs de la famille 86. On a ainsi un jeu de registres compatible qui permet d'exécuter du code 8086 et 80286 (voir le Chapitre 9). Si on accède à un de ces registres 16 bits, les 16 bits de poids fort des registres 32 bits d'usage général ne sont pas modifiés.

On accède aux parties hautes et basses des registres X - AX, BX, CX et DX - de façon indépendante, comme pour les registres 8 bits. Le suffixe H ou L est ajouté à la première lettre du nom du registre 16 bits pour former le nom du registre 8 bits, partie haute ou basse. Par exemple, le registre AL est la partie basse 8 bits du registre AX, et aussi les 8 bits de poids faible

du registre EAX. Le registre AH est la partie haute 8 bits du registre AX et représente aussi les bits 8 à 15 du registre EAX. Si on accède à un de ces registres 8 bits, les bits restants du registre général ne sont pas modifiés. On peut voir cette imbrication des registres 8 bits, 16 bits et 32 bits sur la Figure 2.1.

Caractérisation d'un registre général

Les opérandes peuvent utiliser le contenu d'un registre général pour des calculs d'adresses et pour les opérations arithmétiques simples telles que addition et soustraction. Certaines opérations plus complexes, telles que les opérations sur les chaînes et la multiplication et division en double précision, doivent prendre un ou plusieurs opérandes dans des registres déterminés. La technique qui consiste à utiliser certains registres spécifiques pour certaines opérations s'appelle la *caractérisation.* Le 80386 l'emploie pour traiter des opérations complexes qui demandent plus de deux opérandes (c'est généralement le cas pour le jeu d'instructions du 80386). Puisque la référence à un registre est interne à l'instruction, il n'est pas utile de la rappeler dans un opérande. Par exemple, les registres ECX, ESI, et EDI sont utilisés par les instructions de traitement sur les chaînes pour contenir respectivement la longueur de la chaîne, le pointeur de source, et le pointeur de destination. Les registres EAX et EDX sont destinés aux instructions de multiplication en double précision pour contenir le résultat sur 64 bits. On trouvera des exemples plus détaillés en relation avec ces instructions au Chapitre 3.

A part ces exceptions, un opérande peut représenter tout registre ou toute adresse mémoire ; la sélection d'un opérande 80386 est symétrique.

Les registres de contrôle du processeur

Deux registres contrôlent le fonctionnement du processeur 80386 : EIP, le registre pointeur d'instruction, et EFLAGS, le registre des indicateurs d'état et de contrôle.

Le registre pointeur d'instruction - EIP

Le registre 32 bits EIP n'a qu'une fonction : pointer la prochaine instruction exécutable par le processeur. Lorsque le 80386 exécute un programme, il va chercher et exécute chaque instruction du programme. Il va donc chercher l'instruction pointée par le registre EIP, incrémente le registre EIP de la longueur de l'instruction, puis exécute l'instruction. L'incrémentation laisse le registre EIP pointer sur la prochaine instruction de la séquence d'instructions enregistrées en mémoire. Ainsi, EIP pointe toujours vers la prochaine instruction séquentiellement exécutable. Les instructions de transfert de contrôle (voir le Chapitre 3) peuvent altérer la suite séquentielle d'instructions en chargeant le registre EIP avec une nouvelle valeur.

Les 16 bits inférieurs du registre EIP contiennent le registre 16 bits IP qui donne un pointeur d'instruction 16 bits permettant l'exécution de code 8086 et 80286 (voir le Chapitre 9).

Le registre des indicateurs d'état et de contrôle - EIP

Dans le registre 32 bits EFLAGS, on trouve plusieurs indicateurs d'état et des bits de contrôle que le programme positionne afin de contrôler les opérations sur certaines fonctions du 80386. Le processeur positionne lui-même les bits d'état, qui sont testés par le programme après des opérations arithmétiques. La Figure 2.2 montre la répartition des bits dans le registre EFLAGS. Nous allons décrire brièvement chaque champ.

Les bits marqués 0 ou 1 sur la Figure 2.2 sont *réservés*. On doit charger ces bits réservés à 0 ou à 1 et on doit les ignorer quand on examine le registre EFLAGS. Ceci assure la compatibilité avec les futurs processeurs si Intel décide d'employer ces bits pour définir d'autres indicateurs dans le registre EFLAGS.

Il existe un moyen de changer les bits du registre EFLAGS tout en garantissant une compatibilité avec les processeurs à venir : stocker le registre EFLAGS, modifier seulement les bits en cause, puis recharger EFLAGS avec la valeur modifiée. En ne changeant que les bits désirés, les bits restants ne sont pas modifiés (en incluant tous ceux qui seront définis sur les processeurs futurs).

Indicateurs d' état arithmétiques

Les bits CF, PF, AF, ZF, SF et OF sont positionnés par la plupart des instructions arithmétiques et logiques.

CF — CF est l'indicateur de retenue (*Carry Flag*). Il est levé (CF = 1) si une opération arithmétique génère une retenue positive ou négative (*borrow*) à partir du bit le plus significatif, sinon il est baissé (CF = 0). Cela donne une indication de débordement pour l'arithmétique non signée et traite l'arithmétique multiprécision.

PF — PF est l'indicateur de parité (*Parity Flag*), et renseigne sur la parité des huit bits inférieurs du résultat. PF est à 1 s'il y a un nombre pair de 1 dans les huit bits inférieurs du résultat, et il est à 0 dans le cas contraire. Cela s'appelle la parité *impaire*, puisqu'il y a toujours un nombre impair de 1 dans les données et les bits de parité.

AF — AF est l'indicateur de retenue auxiliaire (*Auxiliary Flag*, ou demi-retenue), qui est levé s'il y a retenue ou emprunt au-dessus du bit 3. Il est utilisé pour traiter l'arithmétique BCD.

Figure 2.2 : *Le registre EFLAGS.*

ZF — ZF est l'indicateur de zéro (*Zero Flag*). Il est positionné si le résultat est nul.

SF — SF est l'indicateur de signe (*Sign Flag*). Il représente le bit le plus significatif du résultat, qui est le bit de signe en notation complément à deux.

OF — OF est l'indicateur de débordement en notation complément à deux (*Overflow Flag*). OF est positionné si le résultat d'une opération arithmétique est trop grand ou trop petit pour être représenté par un entier en complément à deux d'un nombre de bits donné. Il est donc positionné par la plupart des instructions arithmétiques s'il y a une retenue pour le bit supérieur sans retenue extérieure, ou s'il n'y a pas de retenue pour le bit supérieur mais avec retenue extérieure.

Ce jeu d'indicateurs d'état a été choisi de telle façon qu'un seul jeu d'instructions arithmétiques puisse traiter des nombres binaires non signés, des nombres binaires en notation complément à deux, et des chiffres BCD. On peut détecter des dépassements d'échelle en examinant le bit CF après avoir travaillé sur des nombres non signés, le bit OF pour des entiers si-

gnés, et le bit AF pour des chiffres BCD. Une relation d'infériorité entre deux nombres peut être déterminée en effectuant une soustraction puis en examinant :

- CF pour les nombres non signés.

- SF XOR OF (opération OU exclusif sur SF et OF) pour les entiers signés.

- AF pour les chiffres BCD.

En effectuant un OU exclusif sur OF et SF pour déterminer la relation d'infériorité pour des entiers signés, on obtient une indication correcte même si le résultat de la soustraction signale un débordement.

L'exemple suivant illustre comment ces indicateurs traitent les nombres non signés, les entiers signés en notation complément à deux, et les chiffres BCD avec un seul jeu d'instructions. Dans le Chapitre 3, nous décrirons l'effet de chaque instruction sur les indicateurs. Ici, nous allons nous intéresser à l'instruction CMP, qui compare deux nombres en les soustrayant l'un à l'autre, et en positionnant convenablement les indicateurs d'état. Supposons que nous devions comparer deux nombres de 8 bits. Le premier a une valeur de 85h (133 en base 10 si on le considère comme non signé, -123 s'il est signé ou même 5 si les quatre bits inférieurs sont des chiffres BCD). Le second a une valeur de 49h (73 en non signé, 73 en signé et 9 en BCD). Quand on soustrait 49h à 85h, on obtient un résultat de 3Ch sur le 80386 (voir la Figure 2.3), et les indicateurs sont positionnés de la manière suivante :

- CF est à 0, puisqu'il n'y a pas de retenue négative à partir du bit 7.

- PF est à 1, puisqu'il y a un nombre pair de 1 dans le résultat sur 8 bits.

- AF est à 1, puisqu'il y a une retenue négative à partir du bit 3.

- ZF est à 0, puisque le résultat n'est pas 0.

	Représentation 386		Résultat réel		
	Hex	Binaire	Non signé	Compl.à 2	BCD
Emprunt	1	1111	1		1
	85h	10000101b	133	-123	5
Soustr.	-49h	-01001001b	-73	- 73	-9
Total	3Ch	00111100b	60	-196	6
Indicateur de débordement			CF=0	OF=1	AF=1
Indicateur d'infériorité			CF=0	(SF XOR OF)=1	AF=1

Figure 2.3 : *Les indicateurs d'état permettent des types de données multiples.*

- SF est à 0, puisque le bit le plus significatif est 0.

- OF est à 1, puisqu'il y a une retenue négative à partir du bit 6 mais pas du bit 7.

A la Figure 2.3, le résultat binaire produit par le 80386 se trouve à gauche. A droite, on a les entrées binaires et les résultats sous les formats nombres non signés, entiers en notation complément à deux, ou chiffres BCD.

On obtient un résultat correct si les entrées sont des nombres non signés, puisque CF, l'indicateur de débordement pour les nombres non signés, est égal à 0. CF est aussi un indicateur d'infériorité. Ainsi, dans cet exemple, vous pouvez remarquer que le 80386 a bien déterminé que 133 n'est pas plus petit que 73 !

Si les entrées sont des entiers signés, la valeur produite (-196) est trop grande pour tenir dans 8 bits. Le 386 positionne OF, l'indicateur de débordement en complément à deux, pour indiquer que le résultat dépasse le

nombre de bits disponibles. En dépit de ce dépassement, le 386 indique toujours correctement que -123 est plus petit que 73, puisque l'indicateur d'infériorité est dans ce cas SF XOR OF, et est égal à 1.

Si les entrées sont des nombres BCD, la valeur produite par la soustraction binaire est un chiffre décimal non valide (C), comme l'indique AF, l'indicateur BCD de dépassement. AF est aussi un indicateur BCD d'infériorité, et ici il montre que 5 est inférieur à 9.

Le positionnement de CF, ZF, SF et OF dépend de la taille des données utilisées dans l'opération. Des opérations sur des données 32 bits (resp. 16, 8 bits) donnent à SF la valeur du bit 31 (resp. 15, 7) du résultat, positionnent ZF si le résultat 32 bits (resp. 16, 8 bits) est nul, et lèvent CF et OF en accord avec la retenue positive ou négative associée au bit 31 (resp. 15, 7). Le positionnement des indicateurs pour des données 8 bits est illustré par la Figure 2.3.

Le positionnement de PF et AF est indépendant de la taille des opérations mises en jeu. PF est toujours égal au bit de parité impaire pour les 8 bits inférieurs du résultat. AF est levé s'il y a une retenue positive ou négative à partir du bit 3 du résultat.

Indicateurs de contrôle du processeur

Les bits TF, IF, DF, IOPL, NT, RF et VM peuvent être positionnés par le programme pour contrôler le travail du processeur 80386. La plupart de ces indicateurs sont associés à des caractéristiques du 80386 que nous décrirons dans les chapitres suivants. Dans ce paragraphe, nous décrivons rapidement ces indicateurs, et donnons des références sur les chapitres suivants qui détaillent l'utilisation de ces indicateurs. La plupart des indicateurs de contrôle du processeur sont accessibles à tous les programmes, sauf trois : ce sont les indicateurs VM, IOPL et IF. Dans le Chapitre 5, nous expliquerons comment sont protégés ces trois indicateurs de contrôle.

TF L'indicateur du mode pas à pas (*Trap Flag*) contrôle la génération d'interruptions du type pas à pas pour aider à la mise au

point des programmes (voir le Chapitre 8). Une interruption de type pas à pas apparaîtra après chaque instruction lorsque TF = 1.

IF — L'indicateur de validation des interruptions (*Interrupt Flag*) permet la reconnaissance des interruptions externes (Voir le Chapitre 6) qui arrivent sur une broche du processeur. Les interruptions externes sont acceptées si IF = 1, et sont mises en attente si IF = 0.

DF — L'indicateur de direction (*Direction Flag*) détermine si les instructions de chaînes sont en mode postincrémenté (DF = 0), ou postdécrémenté (DF = 1) relativement aux registres pointeurs de chaîne et cela après chaque instruction. Se reporter au Chapitre 3 pour une description détaillée des instructions sur les chaînes.

IOPL — Ce champ de deux bits (*I/O Privilege Level*) contrôle le modèle de protection décrit au Chapitre 5. Le champ IOPL représente le niveau de privilège requis pour accéder aux instructions E/S. Si le niveau de privilège courant est numériquement inférieur ou égal à IOPL, on peut exécuter des instructions E/S, sinon une exception du type protection est générée.

NT — Le bit NT (*Nested Task)* contrôle le traitement de l'instruction IRET (voir le Chapitre 3). Si NT = 0, c'est un retour normal d'interruption qui se produit avec récupération des valeurs de EFLAGS, CS et EIP dans la pile. Si NT = 1, le retour d'interruption s'effectue via une commutation de tâches (voir le Chapitre 5).

RF — L'indicateur de relance (*Restart Flag*) contrôle l'acceptation (RF = 0) ou l'ignorance (RF = 1) des défauts de mise au point. Le processeur met le bit RF à zéro après l'exécution d'une instruction et le positionne si un défaut autre qu'un défaut de mise au point est signalé.

VM Le bit VM est l'indicateur de mode virtuel 8086 (*Virtual Mode*). S'il est positionné, le processeur travaillera en mode virtuel 8086 (voir le Chapitre 9). S'il est baissé, le processeur travaillera en mode protégé classique (voir le Chapitre 5).

Les bits RF, NT, DF, et TF peuvent être modifiés par un programme qui s'exécute à tout niveau de privilège. Seul un programme s'exécutant au niveau de privilège 0 (privilège maximum) peut modifier les champs VM et IOPL. Seul un programme s'exécutant avec le privilège E/S peut changer le bit IF (voir le Chapitre 5).

Seule une instruction IRET ou une commutation de tâches peut modifier les bits RF et VM. Les autres bits de contrôle peuvent être positionnés par l'instruction POPF.

Les registres segments

Le 80386 possède six registres segments 16 bits qui adressent les segments mémoire. Ces registres s'appellent ES, CS, SS, DS, FS et GS. Ils sont représentés à la Figure 2.1. FS et GS sont des nouveautés du 80386.

Pour accéder à des données dans un segment précis, un programme doit charger un des registres segments avec une valeur spécifique qui identifie ce segment. Cette valeur est appelée sélecteur, et on en parlera rapidement dans la suite de ce chapitre et plus précisément au Chapitre 5. Les six registres segments permettent au programme d'accéder à six segments à tout moment. Lorsque le programme en a fini avec un segment et désire travailler sur un autre, il peut charger le sélecteur du nouveau segment dans le registre segment utilisé pour adresser l'ancien segment.

LES CONCEPTS D'ADRESSAGE DE LA MEMOIRE

Le 80386 utilise une technique d'adressage de la mémoire appelée *segmentation*, qui divise l'espace mémoire en une ou plusieurs régions contiguës distinctes nommées *segments*. Une adresse mémoire est définie par deux parties : une partie segment qui identifie le segment contenant, et une partie déplacement (offset) qui associe à un octet unique son déplacement dans le segment. On doit préciser les deux parties pour toute référence mémoire. Dans ce chapitre, nous allons décrire les mécanismes de base de l'adressage mémoire employés dans chaque instruction faisant appel à un opérande mémoire. Nous détaillerons ultérieurement les options disponibles pour la spécification des segments et des déplacements.

Le programmeur d'applications peut accéder à une grande partie des possibilités de structuration de la mémoire qu'apporte la segmentation. Nous allons donc discuter de ces aspects maintenant. Il faut principalement étudier comment on assigne un espace d'un ou plusieurs segments à des unités de programme telles que du code exécutable, des zones de données et la pile du programme.

Dans le Chapitre 1, nous avons introduit la notion de mémoire physique comme un tableau d'octets à une dimension (structure linéaire), et nous avons expliqué comment les types de données de plus d'un octet étaient placés en mémoire. Dans ce chapitre, nous allons voir que la mémoire est divisée en segments, ce qui apporte une seconde dimension à l'espace mémoire. Dans le Chapitre 5, nous traiterons de l'adressage mémoire dans l'optique du système d'exploitation, en décrivant les mécanismes de gestion de la mémoire qui protègent les segments et les réallouent dans l'espace mémoire physique à une dimension.

Adressage en deux parties

A cause de la segmentation, une adresse mémoire du 80386 possède deux parties, une partie segment et une partie déplacement. On doit spécifier ces deux parties dans une instruction qui traite un opérande mémoire.

La partie segment est le *sélecteur* de segment sur 16 bits, qui contient un champ de 14 bits, lequel identifie un des 16 384 segments possibles. La partie déplacement sur 32 bits donne le déplacement associé à l'octet dans le segment. On accède à des octets d'adresses consécutives en incrémentant la partie déplacement de 0 jusqu'à la limite du segment. Puisque le déplacement est sur 32 bits, le déplacement maximal permis, et donc la taille maximale d'un segment, est de 4 Go.

Notation

La notation que nous allons employer dans ce chapitre pour l'adressage mémoire consiste à placer la partie segment à gauche, la partie déplacement à droite et à séparer les deux par deux-points (:). La partie segment peut être soit un nom de segment, soit un registre segment. Si on donne un registre segment, le sélecteur de segment contenu dans ce registre identifie le segment. Les déplacements sont soit des nombres, soit des noms de registres entourés de crochets. Ils peuvent être aussi formés par des expressions mettant en jeu des noms de registres, des facteurs d'échelle, et des constantes. Ceci pour donner une idée des modes d'adressage décrits ultérieurement dans ce chapitre.

Par exemple, l'adresse associée au déplacement 1234 dans le segment nommé NEWDATA sera notée NEWDATA:[1234]. Si le registre segment ES contenait le sélecteur du segment NEWDATA, et le registre général EAX la valeur 1234, la même adresse serait notée ES:[EAX]. Cette notation est dérivée de la notation assembleur pour les adresses mémoire que nous définirons et utiliserons dans le Chapitre 4.

La Figure 2.4 illustre comment les adresses en deux parties sont employées pour accéder à la mémoire. Il y a deux segments sur la figure, le Segment A et le Segment B. Le Segment A (resp. B) contient 11 octets (resp. 17), donc le déplacement associé à A (resp. B) est compris entre 0 et 10 (resp. 16). L'octet à l'adresse A:[5] - c'est-à-dire un déplacement de 5 dans le Segment A - contient la valeur 77. L'octet à l'adresse B:[12] - c'est-à-dire un déplacement de 12 dans le Segment B - contient la valeur 88.

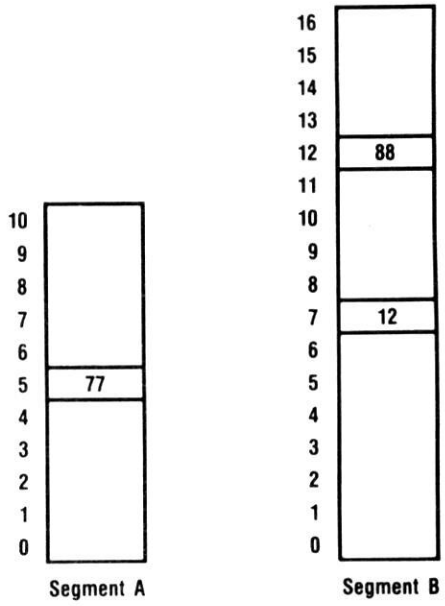


Figure 2.4 : *Adressage en deux parties.*

LE MECANISME D'ADRESSAGE DE LA MEMOIRE

On doit spécifier à la fois les parties segment et déplacement pour toute instruction qui emploie un opérande mémoire. La plupart des programmes travaillent en général sur peu de segments à la fois, mais génèrent de nombreux déplacements différents dans ces segments. Le mécanisme d'adressage du 80386 est optimisé pour ce genre de programmes.

La partie segment : le registre segment

Des registres spéciaux - les six registres segments présentés au début du chapitre - sont employés pour contenir les sélecteurs de segment utilisés dans la partie segment des adresses. Ces registres permettent d'accéder jusqu'à six segments à n'importe quel stade du programme.

Toute référence mémoire spécifie implicitement, explicitement ou par défaut le registre segment contenant le sélecteur pour la partie segment de l'adresse. Les références au code programme utilisent toujours le registre CS, les références à la pile le registre SS, et certaines instructions sur les chaînes le registre ES pour l'opérande destination. On peut employer n'importe lequel de ces six registres segments pour d'autres références. Par exemple, si le sélecteur pour le Segment B de la Figure 2.4 est dans le registre FS, la référence mémoire FS:[12] accédera aux données stockées au déplacement 12 de ce segment, c'est-à-dire à l'adresse B:[12]. Si le sélecteur pour B est transféré du registre FS au registre DS, on accédera aux mêmes données à l'adresse DS:[12].

Le segment adressé par le registre CS est appelé le *segment code* (*Code Segment*). Le registre EIP contient le déplacement de la prochaine instruction exécutable dans le segment adressé par le registre CS, si bien que l'adresse de cette instruction est CS:[EIP]. Si le segment code courant contient à la fois du code et des données, ces dernières peuvent être référencées par le registre CS comme partie segment de l'adresse. Par exemple, une donnée qui se trouve à un déplacement de 8 plus la valeur du registre EAX peut être adressée par CS:[EAX+8] par des instructions dans le même segment.

Toute référence à du code programme utilise le registre CS, si bien qu'un seul segment code est adressable à la fois. Les instructions qui contrôlent les transferts intersegments sont décrites au Chapitre 3. On peut les employer pour charger une nouvelle valeur dans les registres CS et EIP, ce qui permet de changer le lieu d'exécution en passant à un segment code différent. Cela donne une méthode efficace de transferts de contrôle entre des procédures qui appartiennent à des segments différents. Par exemple, le code d'un programme d'application est habituellement dans un segment différent de celui du code du système d'exploitation. Ainsi, par des

transferts intersegments, les applications peuvent efficacement employer les ressources du système d'exploitation.

Le segment adressé par le registre SS est appelé le *segment pile* courant (*Stack Segment*). Les opérations sur la pile telles que PUSH, POP, CALL et RETURN utilisent une pile programme contenue dans le segment adressé par SS. Le sommet de la pile se trouve à un déplacement donné par le registre général ESP qui est réservé à cet effet. L'adresse du sommet de la pile est SS:[ESP]. Nous reparlerons plus précisément de la pile programme dans ce chapitre.

Les registres DS, ES, FS et GS sont disponibles pour adresser des segments de données quelconques requis par le programme. Le registre DS est le registre segment données "principal", puisque toutes les références autres qu'à la pile s'effectuent via ce segment. Comme vous le verrez plus loin dans ce chapitre, les instructions qui font référence à des données dans des segments adressés par CS, SS, ES, FS et GS ont un octet de plus que celles relatives à DS. Ainsi, les références à des segments autres que DS sont légèrement plus gourmandes en mémoire et en temps d'exécution, mais pas beaucoup plus. En général, c'est une bonne idée de disposer les données dans le programme de façon que le registre DS puisse adresser le segment données le plus souvent référencé, et d'utiliser les registres ES, FS, et GS pour adresser les segments auxquels on fait le moins souvent référence.

La Figure 2.5 illustre l'utilisation des registres segments. CS pointe vers le segment code courant, et EIP fournit le déplacement de la prochaine instruction dans le segment. SS pointe vers le segment pile, et ESP identifie le sommet de la pile programme dans le segment. Comme on l'a déjà vu, la mémoire pour la pile est allouée en déplaçant ESP vers les adresses basses, si bien que la partie segment pile au-dessus de ESP est déjà allouée pour la pile, et que la mémoire au-dessous de ESP est libre pour une éventuelle allocation. DS pointe vers le segment données primaire, appelé $DATA_1$. ES, FS et GS sont disponibles pour adresser d'autres segments données, et pourraient pointer vers $DATA_2$, $DATA_3$ ou $DATA_4$ comme l'indiquent les pointillés.

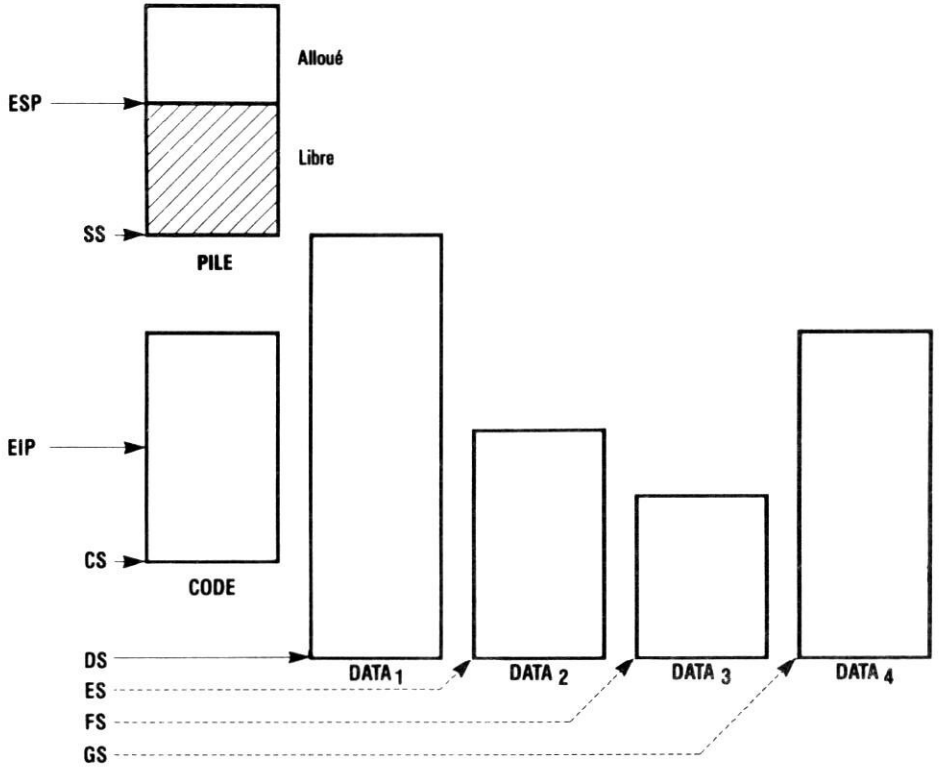


Figure 2.5 : *Utilisation des registres segments.*

La partie déplacement : les modes d'adressage

Dans le chapitre précédent, nous avons vu les mécanismes de spécification de la partie segment d'une adresse. Puisque chaque référence mé-

moire contient à la fois une partie segment et une partie déplacement, nous devons donc décrire maintenant la spécification des parties déplacement.

Le mécanisme de formation de la partie déplacement d'une adresse du 80386 est très souple. Chaque instruction qui fait référence à un opérande mémoire contient la méthode de spécification du calcul de la partie déplacement. Cette spécification s'appelle le *mode d'adressage* de l'instruction. Les modes d'adressage du 80386 spécifient jusqu'à 3 composantes qu'on peut associer pour former le déplacement. Un registre de base, un registre d'index multiplié par 1, 2, 4, ou 8 (facteur d'échelle), et un déplacement constant peuvent être additionnés pour former le déplacement.

N'importe lequel des huit registres généraux 32 bits peut servir de registre de *base*. La composante base peut être omise, elle aussi. N'importe lequel des huit registres généraux 32 bits, excepté le registre pointeur de pile, ESP, peut être utilisé comme registre d'*index*. La composante index peut être omise. Si on spécifie un registre d'index, la valeur contenue dans ce registre peut être multipliée par 1, 2, 4, ou 8 avant d'être ajoutée au déplacement. Une constante de *déplacement* sur 8 ou 32 bits peut être spécifiée ou omise.

Ces modes d'adressage base + (index * échelle) + déplacement procurent un mécanisme d'adressage puissant et flexible qui répond aux besoins des structures de données employées par les langages de haut niveau. Dans le paragraphe suivant, nous allons parler des structures de données telles que : enregistrements (*records*), tableaux, et même des tableaux d'enregistrements employant ces modes d'adressage.

L'utilisation d'un opérande mémoire est rendue plus rapide si cet opérande est *aligné*. C'est-à-dire si le déplacement associé à l'opérande est un multiple de sa taille. Par exemple, un opérande dmot (resp. mot) est aligné si son déplacement est un multiple de 4 (resp. 2). On peut tout de même accéder aux opérandes non alignés, mais cela peut prendre plus de temps. Nous recommandons l'alignement de tous les opérandes.

Le registre segment par défaut pour les références de données est dépendant du registre de base sélectionné. Si le registre de base est ESP ou EBP, le registre segment par défaut passe du classique DS à SS, puisque

les registres ESP et EBP sont destinés à être employés avec la pile. Pour tous les autres choix de registres de base, y compris les registres autres que les registres de base, DS reste le registre segment par défaut. L'accès à des données d'un segment adressé par un registre segment différent du registre par défaut rend nécessaire l'emploi d'un octet supplémentaire d'instruction pour préciser le registre segment désiré. L'emploi de EBP comme registre d'index (ESP ne peut pas être registre d'index) n'affecte pas le choix du registre segment par défaut. Seule la sélection du registre de base affecte le choix du registre segment par défaut.

Le Tableau 2.1 décrit la variété des modes d'adressage disponibles. Dans les paragraphes suivants, nous donnerons des exemples d'utilisation des modes d'adressage.

BASE + (INDEX * ÉCHELLE) + DÉPLACEMENT

BASE	INDEX	ÉCHELLE	DÉPLACEMENT
Aucun	Aucun		
EAX	EAX		
ECX	ECX		
EDX	EDX	1	Aucun
EBX	EBX	2	8-bit
ESP[1]	---[2]	4	32-bit
EBP[1]	EBP	8	
ESI	ESI		
EDI	EDI		

Notes :

1 SS est le registre segment par défaut si ESP ou EBP est le registre de base.

2 Le symbole (---) signifie que ESP ne peut pas être employé comme registre d'index.

Tableau 2.1 : *Les modes d'adressage du 80386.*

La pile programme

Une *Pile Programme* est un système d'allocation LIFO (*last in, first out*, dernier entré, premier sorti). Cette structure est bien appropriée au stockage sous forme d'empilement des résultats de sous-expressions et de sous-programmes. Les deux opérations de base sur la pile sont PUSH et POP. L'opération PUSH ajoute un nouvel élément à la pile, et l'opération POP enlève le dernier élément empilé. Un PUSH suivi d'un POP laisse la pile inchangée.

Les opérations PUSH et POP emploient la pile pour stocker des valeurs temporaires - par exemple, les parties intermédiaires d'un calcul complexe. Lorsque ces parties intermédiaires sont calculées, leurs résultats peuvent être empilés dans la pile. Pour calculer la valeur de l'expression finale, les résultats intermédiaires seront dépilés et combinés pour former le résultat final. Le stockage dans la pile reflète l'emboîtement naturel du calcul des sous-expressions.

Cette façon de stocker dans la pile offre un mécanisme efficace et flexible pour gérer des appels et des retours aux sous-programmes. Ce qui permet bien sûr des procédures emboîtées. Une instruction CALL empile une adresse de retour dans la pile où elle reste jusqu'à ce qu'une instruction RET la dépile. Un appel à un sous-programme empile l'adresse de retour, cachant l'adresse de retour de la procédure extérieure. Le retour d'un sous-programme dépile l'adresse de retour, mettant en évidence l'adresse de la procédure extérieure au sommet de la pile. L'utilisation de la pile programme est la base du mécanisme des appels aux sous-programmes comme nous le verrons ultérieurement.

A l'intérieur d'un segment mémoire, on adresse la pile par le registre SS. Le registre ESP contient le déplacement du sommet de la pile programme à l'intérieur du segment. L'adresse SS:[ESP] pointe vers le sommet courant de la pile programme.

Stockage temporaire dans la pile

Sur le 80386, la pile croît vers les adresses inférieures, et chaque élément de la pile est composé de 4 octets (32 bits). Pour empiler un élément dans la pile du 80386, le registre ESP est tout d'abord décrémenté de 4, puis le nouvel élément est écrit à l'adresse mémoire SS:[ESP]. Pour dépiler un élément, on lit l'adresse mémoire SS:[ESP] pour retrouver l'élément, et le registre ESP est incrémenté de 4. A tout moment, on adresse l'élément courant du sommet de la pile par SS:[ESP]. Le second élément est adressé par SS:[ESP + 4], le troisième par SS:[ESP + 8], etc. On peut faire rapidement référence à des opérandes dans la pile si celle-ci est alignée. Comme ses éléments sont composés de quatre octets, cela s'avère facile si on s'assure que le registre ESP contient toujours un multiple de 4.

La Figure 2.6 illustre comment PUSH et POP travaillent sur la pile programme du 80386. Le schéma de la pile la plus à gauche montre la pile après empilement de trois éléments aux déplacements FCh, F8h, et F4h dans le segment de pile dont la taille est 100h octets. Le sommet de la pile est situé au bas de la figure car la pile croît vers les adresses inférieures. Le schéma de droite montre la pile après empilement de la valeur 12345678h. ESP contient maintenant la valeur F0h, c'est-à-dire 4 de moins que la valeur initiale. L'élément qui était au sommet de la pile à l'origine est maintenant le second élément de la pile et peut être adressé par SS:[ESP + 4]. Une opération POP retournera la pile dans le même état que la pile de gauche, en incrémentant de 4 la valeur de ESP.

Les sous-programmes et la pile

La pile programme du 80386 propose un mécanisme efficace des appels et retours aux sous-programmes. L'instruction CALL empile l'adresse de retour (l'adresse de l'instruction suivant le CALL) dans la pile programme, et saute au début du sous-programme. L'instruction RET dépile l'adresse de retour, puis saute à cette adresse pour exécuter l'instruction suivant le CALL. La pile programme est aussi employée pour passer des paramètres

aux sous-programmes, et pour stocker des variables locales du sous-programme. En utilisant une pile, le 80386 gère efficacement les sous-programmes emboîtés, en incluant les sous-programmes récursifs et réentrants.

Une optimisation de cette technique permet d'employer les registres généraux pour quelques paramètres ou pour les variables locales utilisées le plus fréquemment. Puisqu'on peut accéder aux registres plus facilement qu'à la mémoire, cette optimisation peut accélérer l'exécution du programme. On peut aussi utiliser la pile pour stocker le reste des paramètres et des variables locales.

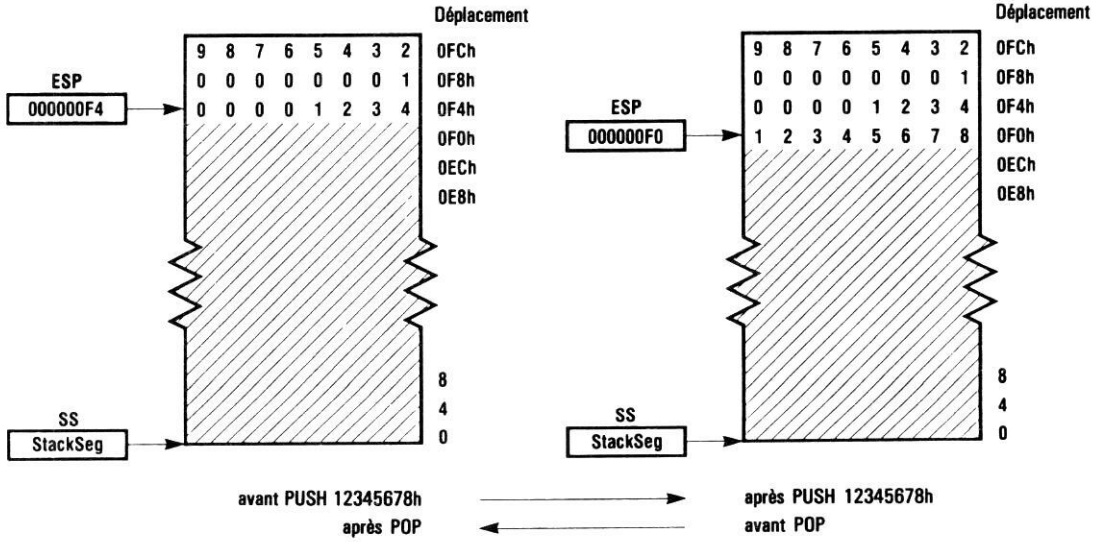


Figure 2.6 : *Les Opérations sur la Pile PUSH/POP.*

Liens de base entre les sous-programmes

Le lien le plus simple permet d'appeler un sous-programme avec un passage de paramètres et une allocation des variables locales dans la pile du programme. Un appel à un sous-programme se décompose en trois étapes :

1 Empiler les paramètres dans la pile programme.

2 Utiliser une instruction CALL pour empiler l'adresse de retour dans la pile et sauter au sous-programme.

3 Soustraire une constante du registre ESP pour réserver de l'espace dans la pile pour les variables locales.

A l'intérieur du sous-programme, on peut adresser les paramètres et les variables locales avec des déplacements constants à partir du registre ESP. Le mode d'adressage utilisant ESP comme base plus un déplacement signé sur 8 bits peut être employé lorsque la taille des variables locales et des paramètres n'excède pas 127 octets. Sinon, on doit utiliser un déplacement complet de 32 bits à partir de ESP. Ces derniers doivent suivre tout changement de ESP à l'intérieur du sous-programme, que ce soit des valeurs temporaires empilées ou dépilées, ou des paramètres empilés pour être transmis à un autre sous-programme. Cette technique peut être facile à traiter sur un compilateur, mais est relativement confuse lors de la programmation en assembleur ! Une autre possibilité consiste à employer le registre EBP comme registre de base fixe pour pointer sur la zone de la pile allouée à la procédure active. Cela permet d'adresser des variables et paramètres à des déplacements fixés à partir de EBP, mais empêche d'employer EBP comme registre général.

Les registres ESP et EBP sont destinés à adresser efficacement la pile, puisque le choix de ESP ou EBP comme registre de base fixe SS comme registre de segment par défaut. Cela permet à ESP ou EBP d'adresser la zone de pile pour la procédure courante, sans ajouter un octet d'instruction supplémentaire pour préciser le segment de pile.

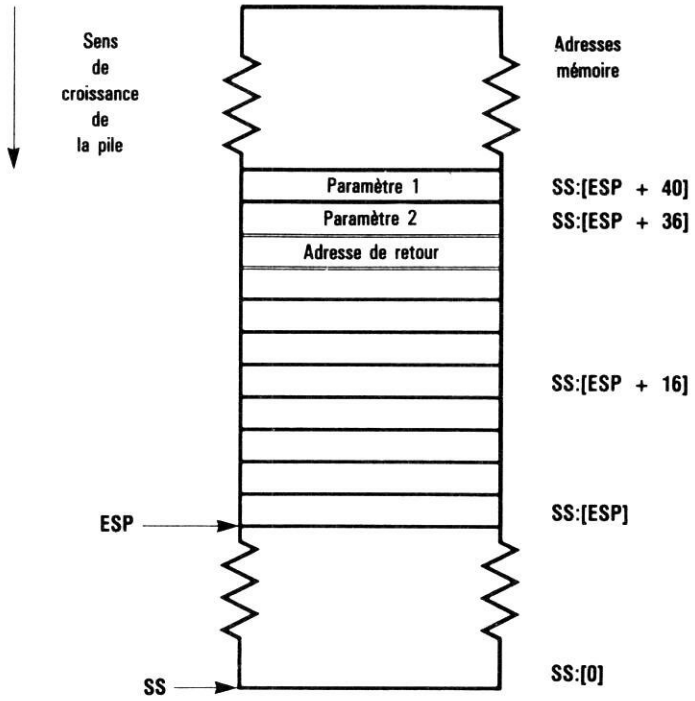


Figure 2.7 : *Paramètres, adresses de retour et variables locales stockés dans la pile programme.*

La Figure 2.7 illustre l'état de la pile programme après l'appel à un sous-programme avec 2 paramètres et 32 octets de variables locales. Le sous-programme se trouve dans le même segment que la procédure appe-

lante, si bien qu'un CALL intrasegment (voir le Chapitre 3) est employé pour empiler les quatre octets de l'adresse de retour. Les deux paramètres empilés occupent 8 bits et l'adresse de retour 4 octets. Avec les 32 octets de stockage local, on arrive à un total de 44 octets.

A l'intérieur du sous-programme, on adresse les variables locales par SS:[ESP + *n*], où *n* varie de 0 à 31. Le premier paramètre est à l'adresse SS:[ESP + 40], le second à l'adresse SS:[ESP + 36].

Pour quitter le sous-programme, on effectue les trois étapes du CALL dans le sens inverse :

1 Ajouter une valeur à ESP pour dépiler les variables locales de la pile programme.

2 Quitter le sous-programme en dépilant l'adresse de retour et en sautant à cette adresse.

3 Ajouter une valeur à ESP pour dépiler les paramètres.

Une forme d'instruction RET, décrite au Chapitre 3, combine ces deux dernières étapes.

Type de données pointeur

Dans le Chapitre 1, nous avons décrit la plupart des types de données. Nous avons différé l'introduction de la notion de type de données pointeur jusqu'à maintenant, si bien que nous avons pu introduire d'abord quelques concepts de base sur l'adressage. Un type de données pointeur contient une valeur qui donne l'adresse d'une donnée. Les pointeurs sont utiles pour construire des structures de données complexes telles que des listes ou des arbres dont les structures varient de façon conséquente lors de l'exécution du programme. Chaque élément d'une structure liste ou arbre contient un ou plusieurs pointeurs vers d'autres éléments. Ainsi, les éléments sont liés et déliés simplement en stockant les adresses d'autres éléments dans ces pointeurs. D'autres constructions telles que les tableaux et les enregistre-

ments sont plutôt des structures peu variantes. C'est pourquoi elles sont généralement plus efficaces pour le stockage et le temps d'exécution que les structures liées par des pointeurs.

Le 80386 gère deux types de pointeurs : un pointeur 48 bits qui contient les deux parties d'une adresse, et un pointeur 32 bits qui contient uniquement la partie déplacement d'une adresse. On emploie le pointeur 48 bits quand des éléments de segments différents sont liés par des pointeurs. Le pointeur de déplacement est plus efficace, mais on peut l'employer seulement quand tous les éléments liés sont stockés dans le même segment.

Un pointeur 48 bits (voir la Figure 2.8) contient les deux parties d'une adresse. La partie déplacement est représentée par les 32 bits de poids faible, et le sélecteur de segment se situe dans les 16 bits de poids fort. Pour adresser des données avec un pointeur 48 bits, on doit charger les deux parties dans les registres. Le sélecteur de segment, les 16 bits de poids fort, se charge dans un des registres segments. La partie déplacement, les 32 bits de poids faible, se charge dans un des registres généraux où il peut être employé comme registre de base. Dans le Chapitre 3, nous décrirons les instructions LDS, LES, LFS, et LGS qui permettent de charger ces pointeurs 48 bits en une seule instruction dans des paires registre segment/registre général.

Si toutes les adresses qui doivent être stockées dans des pointeurs ont la même partie segment, on peut optimiser judicieusement en conservant le sélecteur pour ce segment dans un registre segment réservé à cet emploi, et en stockant seulement les parties déplacement dans des pointeurs 32 bits (voir la Figure 2.8). Pour adresser des données avec un pointeur 32 bits, seule la partie déplacement doit être chargée dans un des registres généraux pour être employée comme registre de base. La partie segment est alors prise dans le registre segment qui contient le sélecteur commun.

Un pointeur de déplacement 32 bits est plus efficace au niveau stockage et temps d'exécution qu'un pointeur 48 bits. Si on stocke toutes les données d'une application dans un segment, y compris la pile programme, on peut alors employer des pointeurs 32 bits. Dans ce modèle, les registres segments SS, DS et ES sont chargés avec le sélecteur pour le segment commun contenant toutes les données et la pile programme. Ainsi, toutes

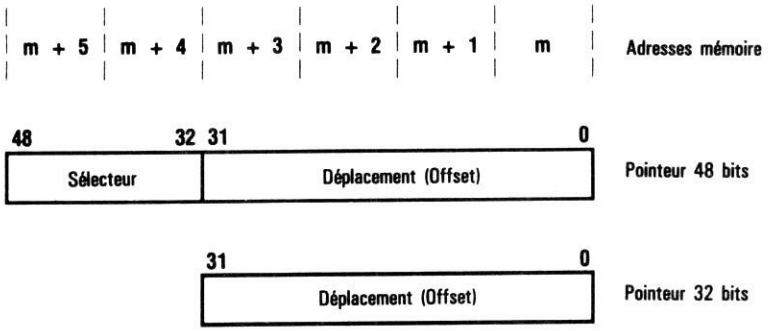


Figure 2.8 : *Pointeurs 48 bits et 32 bits.*

les références mémoire peuvent utiliser le registre segment données par défaut. Bien qu'il n'y ait qu'un seul segment dans ce modèle, on considère que les adresses mémoire ont leur partie segment qui est implicitement le segment unique contenant l'ensemble.

Les modes d'adressage et les structures de données

Les modes d'adressage base + index + déplacement du 80386 répondent directement aux besoins des langages de haut niveau. Les variables scalaires, les enregistrements, les tableaux, et même les tableaux d'enregistrements et les enregistrements de tableaux sont directement traités par les modes d'adressage du 80386. Ces structures de données peuvent être localisées en stockage *statique*, comme l'exige le FORTRAN, ou *allouées de façon dynamique* dans la pile programme (et référencées relati-

vement à SS:[ESP]) ou dans le tas (*heap*) (et référencées indirectement par des pointeurs) comme le demandent le Pascal ou C. Les registres de base et d'index apportent deux composantes dynamiques pour le mode d'adressage, et le déplacement une composante statique. Les données allouées statiquement sont adressées en utilisant simplement un déplacement constant à l'intérieur d'un segment données. Les données allouées dans la pile sont adressées en employant un déplacement constant à partir du registre ESP ou EBP, comme on l'a vu dans un paragraphe précédent. Les données allouées dans le tas sont adressées en chargeant un pointeur sur les données dans une paire registre segment/registre général, puis en employant le registre général comme registre de base. Le Tableau 2.2 donne une synthèse des modes d'adressage du 80386 répondant aux besoins des langages, et un exemple en notation assembleur.

Pour simplifier l'allocation des registres pour les tables de traduction des langages, tout registre parmi les huit registres généraux peut être employé comme registre de base ou comme registre d'index (sauf ESP dans ce cas). On peut utiliser la valeur du registre d'index directement (facteur d'échelle 1), ou avec un facteur d'échelle de 2, 4 ou 8 pour indexer directement des données 16, 32 et 64 bits sans instructions de décalage ni registre supplémentaire.

Stratégies de segmentation

Le 80386 permet une multitude de stratégies de segmentation, qui déterminent comment les unités de programme sont allouées au niveau stockage dans un ou plusieurs segments mémoire. Une *unité de programme* est une procédure de code indépendante, ou zone de données, ou pile programme. Chaque unité de programme peut être contenue dans son propre segment pour un maximum de flexibilité dans la réallocation, le partage et la protection des unités indépendantes. Au contraire, le code entier d'une application, les données et la pile peuvent être stockés dans un segment unique.

Type stockage	Type structure	Mode d'adressage	Exemple
Statique	Scalaire	Déplacement	DS:[1000]
	Tableau	Index + Dépl.	DS:[ESI*4+1004]
	Enregistrement	Déplacement	DS:[1234h]
	Tableau d'enregistrements	Index + Dépl.	DS:[EDX*2+1000h]
	Enregistrement de tableaux	Index + Dépl.	DS:[EBX*2+1048h]
Pile	Scalaire	Base + Dépl.	SS:[ESP+24]
	Tableau	Base + Index + Dépl.	SS:[ESP+ESI*4+120]
	Enregistrement	Base + Dépl.	SS:[ESP+48]
	Tableau d'enregistrements	Base + Index + Dépl.	SS:[EBP+EDX*8-256]
	Enregistrement de tableaux	Base + Index + Dépl.	SS:[ESP+EDI*2+1000h]
Tas	Scalaire	Base	ES:[EBX]
	Tableau	Base + Index	FS:[EBX+ESI*4]
	Enregistrement	Base + Dépl.	DS:[EAX+12]
	Tableau d'enregistrements	Base + Index + Dépl.	ES:[EBX+EAX*8+6]
	Enregistrement de tableaux	Base + Index + Dépl.	DS:[EAX+EDI*2+1800h]

Tableau 2.2 : *Les modes d'adressage du 80386 et structures de données.*

Il y a deux caractéristiques qui font des segments un outil très puissant pour l'adressage, la protection et le partage des procédures de codes, des zones de données et de la pile programme.

- Un segment peut avoir une taille quelconque.

- Le mécanisme de gestion mémoire décrit au Chapitre 5 réalloue, partage et protège un segment comme une unité unique et indivisible, quelle que soit la taille du segment.

Grâce à la taille variable des segments, toute unité programme peut être installée dans un segment unique, où elle sera protégée et réallouée par le système d'exploitation en tant qu'unité unique. Une petite procédure de 20 octets peut être stockée dans un segment, tout comme un tableau de données de 100 Mo, ou le code et les données d'une application entière.

Les Figures 2.9 et 2.10 illustrent deux stratégies de stockage des unités programme dans des segments. C'est le nombre de segments employés pour stocker les données qui diffère pour chaque modèle. Le modèle de la Figure 2.10 utilise un seul segment pour le stockage des données, si bien qu'on l'appelle le modèle à segment unique (même si un second segment comprend le code). Le modèle de la Figure 2.9 enregistre les données d'une application dans plusieurs segments. D'autre part, la taille des pointeurs est différente dans chaque cas. Le modèle à segment unique peut employer des pointeurs 32 bits, puisque toutes les données et toute la pile sont stockées dans un segment. Le modèle à segments multiples rend nécessaire l'emploi de pointeurs 48 bits puisque les zones de données sont dans différents segments, et que chaque pointeur de données doit préciser explicitement sur quel segment il pointe.

La Figure 2.9 montre comment on peut stocker le code, la pile et les données de deux applications distinctes dans des segments séparés. Le code de l'application 1 se trouve dans le segment appelé $Code_1$, sa pile dans le segment $Pile_1$, et ses données dans $Données_{1a}$ et $Données_{1b}$. Il partage un segment de données $Données_S$ avec l'application 2 dont le code, la pile et les données se trouvent dans les segments $Code_2$, $Pile_2$ et $Données_2$.

Dans la stratégie à segments multiples, les deux applications sont stockées dans huit segments différents, avec un segment partagé entre les deux applications. On peut fixer des limitations d'accès pour chaque segment. Par exemple, on peut réduire l'accès des deux segments de code, $Code_1$ et $Code_2$, à leur seule exécution. Le segment de données partagé $Data_S$ peut avoir un accès en écriture limité à l'une ou les deux applications, afin de prévenir toute modification non autorisée des données. Etant donné qu'il s'agit de segments données multiples, on doit utiliser des pointeurs 48 bits.

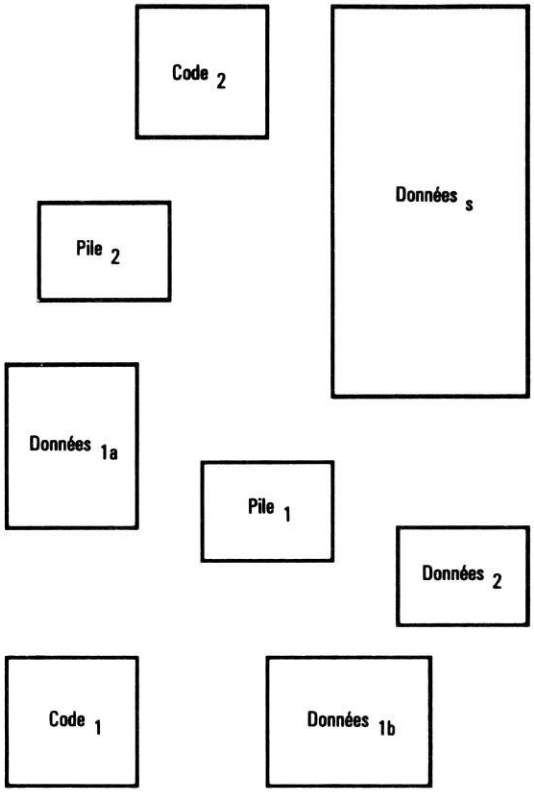


Figure 2.9 : *Stratégie de segmentation à segments multiples.*

Il est donc intéressant de placer chaque unité programme dans un segment distinct pour une meilleure souplesse dans la réallocation et la protection. D'un autre côté, stocker toutes les données et la pile d'une

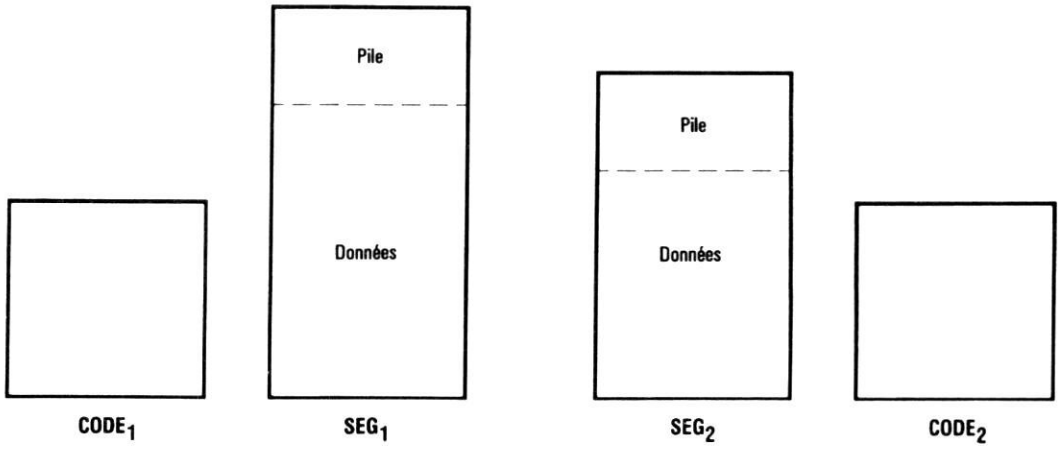


Figure 2.10 : *Stratégie de segmentation à segment unique.*

application dans un segment unique est plus efficace, puisque cela permet d'utiliser des pointeurs 32 bits. Cette stratégie à segment unique correspond aussi au modèle d'application offert par la plupart des mini-ordinateurs ou unités centrales (qui n'ont pas de segment), assurant ainsi une portabilité aisée des applications vers ces environnements.

Même si la segmentation n'est pas employée pour une application sur ce modèle, les logiciels des systèmes d'exploitation peuvent aussi utiliser la segmentation pour séparer et protéger les applications entre elles, en environnement multitâche. Chaque application est alors stockée dans deux segments différents : l'un contenant tout le code pour l'application, l'autre comprenant toutes les données et la pile. Le système d'exploitation est stocké dans deux segments ou plus, et gère les segments de façon que les applications entre elles soient isolées, et que lui-même le soit de toutes les applications.

La Figure 2.10 illustre la stratégie de segmentation à segment unique. $CODE_1$ (resp. $CODE_2$) contient le code pour l'application 1 (resp. 2), SEG_1 (resp. SEG_2) la pile et les données pour l'application 1 (resp. 2). Puisqu'il y a seulement un segment de données par application, on peut utiliser efficacement un pointeur de déplacement 32 bits pour chaque application. Ainsi les deux applications ne peuvent plus partager un segment de données (voir la Figure 2.9). Aussi, le segment commun doit permettre des accès lecture et écriture, sans protection au niveau de l'écriture comme cela était possible pour le modèle à segments multiples.

Il existe des avantages et des inconvénients dans ces deux stratégies de segmentation. Dans de nombreux cas, le logiciel du système d'exploitation dictera l'emploi d'un modèle plutôt que de l'autre. Si on doit faire un choix, nous conseillons le modèle à segment unique pour la plupart des applications, pour sa simplicité inhérente. Le modèle à segment unique offre aussi une meilleure efficacité grâce à l'emploi de pointeurs 32 bits. Pour quelques systèmes, la souplesse du partage et la protection apportée par le modèle à segments multiples sont nécessaires. Mais le modèle à segment unique, plus simple, convient à la plupart des applications.

CODAGE DES INSTRUCTIONS

Dans ce paragraphe, nous allons décrire le codage des instructions, et insister sur le codage des opérandes. Nous aborderons en premier lieu le codage général des instructions, puis nous donnerons la structure détaillée bit par bit du codage des opérandes immédiat, registre et mémoire.

Il n'est pas nécessaire de comprendre ces lignes pour programmer le 80386, même au niveau du langage d'assemblage. Dans les paragraphes précédents, nous avons décrit les opérandes par leur représentation symbolique dans un langage d'assemblage. Dans cette partie, nous allons détailler les blocs de bits qui servent à représenter ces opérandes en instructions du 80386. Si vous n'avez pas besoin d'une information aussi détaillée, vous pouvez passer au paragraphe décrivant l'espace mémoire E/S.

Ce paragraphe est avant tout destiné aux développeurs de langages tels que assembleurs, compilateurs et debuggers. Pour ces derniers, la connaissance du codage des instructions est nécessaire. En fait, la fonction première de ces langages est de cacher ces détails au programmeur.

Champs d'une instruction

La Figure 2.11 montre le format général d'une instruction du 80386. Chaque instruction peut comporter jusqu'à cinq champs, avec les champs stockés aux adresses les plus basses à partir de la gauche.

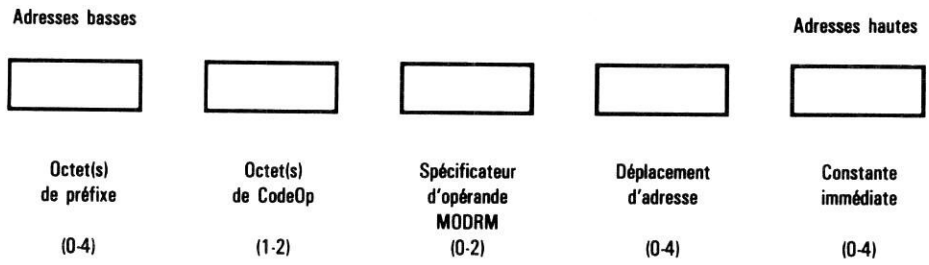


Figure 2.11 : *Format général d'une instruction.*

Octets préfixes

On peut spécifier jusqu'à quatre octets préfixes, ou même aucun octet préfixe dans une instruction. S'ils sont présents, c'est dans les premiers octets d'une instruction qu'on les trouve, d'où leur nom. On emploie les préfixes seulement pour modifier l'interprétation de l'instruction suivante, et elle seule. Ils sont utilisés pour associer des paramètres peu fréquents. Ainsi, le code reste plus compact dans les cas classiques. Le recouvrement d'un registre segment est un exemple de préfixe d'instruction, qu'on détaillera ultérieurement. Il est employé pour déterminer le registre segment d'un opérande mémoire dans le cas où le registre segment par défaut n'est pas approprié. On trouvera dans le Chapitre 3 d'autres exemples (préfixes LOCK et REP).

Octets CodeOp

Les octets CodeOp suivent immédiatement le dernier octet préfixe. On doit trouver au moins un octet CodeOp dans chaque instruction, et quelques instructions en demandent deux. Une forme particulière d'instruction sur un octet combine un CodeOp avec un spécificateur d'opérande registre sur 3 bits.

Spécificateur d' opérande MODRM

Un spécificateur d'opérande MODRM suit le ou les octets CodeOp. Le CodeOp détermine si ce spécificateur est présent et si ce champ spécifie deux opérandes, ou un opérande plus des octets supplémentaires de CodeOp. Ce champ peut spécifier comme opérande un registre, une mémoire, deux registres, un registre et une mémoire. Le champ MODRM peut avoir un ou deux octets.

Tableau G.1 : 80386 Jeu d'instructions Compteur de cycles

INSTRUCTION	FORMAT	COMPTEUR		NOTES	
		Mode adresse réelle ou mode virtuel 8086	Mode protégé adresse virtuelle	Mode adresse réelle ou mode virtuel 8086	Mode protégé adresse virtuelle
INTERRUPT INSTRUCTIONS (Continued)					
BOUND:					
Via Interrupt or Trap Gate to Same Privilege Level			59		g, j, k, r
Via Interrupt or Trap Gate to Different Privilege Level			99		g, j, k, r
From 286 Task to 286 TSS via Task Gate			254		g, j, k, r
From 286 Task to 386 TSS via Task Gate			284		g, j, k, r
From 286 Task to virt 8086 Mode via Task Gate			231		g, j, k, r
From 386 Task to 286 TSS via Task Gate			264		g, j, k, r
From 386 Task to 386 TSS via Task Gate			294		g, j, k, r
From 386 Task to virt 8086 Mode via Task Gate			243		g, j, k, r,
From virt 8086 Mode to 286 TSS via Task Gate			264		g, j, k, r
From virt 8086 Mode to 386 TSS via Task Gate			294		g, j, k, r
From virt 8086 md to priv level 0 via Trap Gate or Interrupt Gate			119		
INTERRUPT RETURN					
IRET = Interrupt Return	11001111	22			g, h, j, k, r
Protected Mode Only (IRET)					
To the Same Privilege Level (within task)			38		g, h, j, k, r
To Different Privilege Level (within task)			82		g, h, j, k, r
From 286 Task to 286 TSS			232		h, j, k, r
From 286 Task to 386 TSS			265		h, j, k, r
From 286 Task to Virtual 8086 Task			214		h, j, k, r
From 286 Task to Virtual 8086 Mode (within task)			60		
From 386 Task to 286 TSS			271		h, j, k, r
From 386 Task to 386 TSS			275		h, j, k, r
From 386 Task to Virtual 8086 Task			224		h, j, k, r
From 386 Task to Virtual 8086 Mode (within task)			60		
PROCESSOR CONTROL					
HLT = HALT	11110100	5	5		l
MOV = Move to and From Control/Debug/Test Registers					
CR0/CR2/CR3 from register	00001111 00100010 11 eee reg	10/4/5	10/4/5		l
Register From CR0–3	00001111 00100000 11 eee reg	6	6		l
DR0–3 From Register	00001111 00100011 11 eee reg	22	22		l
DR6–7 From Register	00001111 00100011 11 eee reg	16	16		l
Register from DR6–7	00001111 00100001 11 eee reg	14	14		l
Register from DR0–3	00001111 00100001 11 eee reg	22	22		l
TR6–7 from Register	00001111 00100110 11 eee reg	12	12		l
Register from TR6–7	00001111 00100100 11 eee reg	12	12		l
NOP = No Operation	10010000	3	3		
WAIT = Wait until BUSY # pin is negated	10011011	6	6		

Tableau G.1 : 80386 Jeu d'instructions Compteur de cycles

INSTRUCTION	FORMAT	COMPTEUR		NOTES	
		Mode adresse réelle ou mode virtuel 8086	Mode protégé adresse virtuelle	Mode adresse réelle ou mode virtuel 8086	Mode protégé adresse virtuelle
PROCESSOR EXTENSION INSTRUCTIONS					
Processor Extension Escape	11011TTT mod LLL r/m TTT and LLL bits are opcode information for coprocessor.	See 80287/80387 data sheets for clock counts			h
PREFIX BYTES					
Address Size Prefix	01100111	0	0		
LOCK – Bus Lock Prefix	11110000	0	0		m
Operand Size Prefix	01100110	0	0		
Segment Override Prefix					
CS:	00101110	0	0		
DS:	00111110	0	0		
ES:	00100110	0	0		
FS:	01100100	0	0		
GS:	01100101	0	0		
SS:	00110110	0	0		
PROTECTION CONTROL					
ARPL – Adjust Requested Privilege Level					
From Register/Memory	01100011 mod reg r/m	N/A	20/21	a	h
LAR – Load Access Rights					
From Register/Memory	00001111 00000010 mod reg r/m	N/A	15/16	a	g, h, j, p
LGDT – Load Global Descriptor					
Table Register	00001111 00000001 mod 010 r/m	11	11	b, c	h, l
LIDT – Load Interrupt Descriptor					
Table Register	00001111 00000001 mod 011 r/m	11	11	b, c	h, l
LLDT – Load Local Descriptor					
Table Register to Register/Memory	00001111 00000000 mod 010 r/m	N/A	20/24	a	g, h, j, l
LMSW – Load Machine Status Word					
From Register/Memory	00001111 00000001 mod 110 r/m	10/13	10/13	b, c	h, l
LSL – Load Segment Limit					
From Register/Memory	00001111 00000011 mod reg r/m				
Byte-Granular Limit		N/A	20/21	a	g, h, j, p
Page-Granular Limit		N/A	25/26	a	g, h, j, p
LTR – Load Task Register					
From Register/Memory	00001111 00000000 mod 001 r/m	N/A	23/27	a	g, h, j, l
SGDT – Store Global Descriptor					
Table Register	00001111 00000001 mod 000 r/m	9	9	b, c	h

Tableau G.1 : 80386 Jeu d'instructions Compteur de cycles

INSTRUCTION		FORMAT			COMPTEUR		NOTES	
					Mode adresse réelle ou mode virtuel 8086	Mode protégé adresse virtuelle	Mode adresse réelle ou mode virtuel 8086	Mode protégé adresse virtuelle
SIDT	= Store Interrupt Descriptor Table Register	00001111	00000001	mod001 r/m	9	9	b, c	h
SLDT	= Store Local Descriptor Table Register To Register/Memory	00001111	00000000	mod000 r/m	N/A	2/2	a	h
SMSW	= Store Machine Status Word	00001111	00000001	mod100 r/m	10/13	10/13	b, c	h, l
STR	= Store Task Register To Register/Memory	00001111	00000000	mod001 r/m	N/A	2/2	a	h
VERR	= Verify Read Access Register/Memory	00001111	00000000	mod100 r/m	N/A	10/11	a	g, h, j, p
VERW	= Verify Write Access	00001111	00000000	mod101 r/m	N/A	15/16	a	g, h, j, p

NOTES

Notes a through c apply to 80386 real address mode only:

a. This is a protected mode instruction. Attempted execution in real mode will result in exception 6 (invalid opcode).

b. Exception 13 fault (general protection violation) will occur in real mode if an operand reference is made that partially or fully extends beyond the maximum CS, DS, ES, FS or GS limit, FFFFh. Exception 12 fault (stack segment limit violation or not present) will occur in real mode if an operand reference is made that partially or fully extends beyond the maximum SS limit.

c. This instruction may be executed in real mode. In real mode, its purpose is primarily to initialize the CPU for protected mode.

Notes d through g apply to 80386 real address mode and 80386 protected virtual address mode:

d. The 80386 uses an early-out multiply algorithm. The actual number of clocks depends on the position of the most significant bit in the operand (multiplier).

Clock counts given are minimum to maximum. To calculate actual clocks, use the formula

Actual Clock = if m <> 0 then max ([$\log_2$ | m |],3) + 6 clocks:
if m = 0 then 9 clocks

where m is the multiplier.

e. An exception may occur, depending on the value of the operand.

f. LOCK# is automatically asserted, regardless of the presence or absence of the LOCK# prefix.

g. LOCK# is asserted during descriptor table accesses.

Notes h through r apply to 80386 protected virtual address mode only:

h. Exception 13 fault (general protection violation) will occur if the memory operand in CS, DS, ES, FS, or GS cannot be used due to either a segment limit violation or access rights violation. If a stack limit is violated, an exception 12 (stack segment limit violation or not present) occurs.

i. For segment load operations, the CPL, RPL, and DPL must agree with the privilege rules to avoid an exception 13 fault (general protection violation). The segment's descriptor must indicate "present" or exception 11 (CS, DS, ES, FS, GS not present) will occur. If the SS register is loaded and a stack segment not present is detected, an exception 12 (stack segment limit violation or not present) occurs.

j. All segment descriptor accesses in the GDT or LDT made by this instruction will automatically assert LOCK# to maintain descriptor integrity in multiprocessor systems.

k. JMP, CALL, INT, RET, and IRET instructions referring to another code segment will cause an exception 13 (general protection violation) if an applicable privilege rule is violated.

l. An exception 13 fault occurs if CPL is greater than 0 (0 is the most privileged level).

m. An exception 13 fault occurs if CPL is greater than IOPL.

n. The IF bit of the flag register is not updated if CPL is greater than IOPL. The IOPL and VM fields of the flag register are updated only if CPL = 0.
o. The PE bit of the MSW (CR0) cannot be reset by this instruction. Use MOV into CR0 if desiring to reset the PE bit.
p. Any violation of privilege rules as applied to the selector operand does not cause a protection exception; rather, the zero flag is cleared.
q. If the coprocessor's memory operand violates a segment limit or segment access rights, an exception 13 fault (general protection violation) will occur before the ESC instruction is executed. An exception 12 fault (stack segment limit violation or not present) will occur if the stack limit if violated by the operand's starting address.
r. The destination of a JMP, CALL, INT, RET, or IRET must be in the defined limit of a code segment, or an exception 13 (general protection violation) will occur.

Nous utilisons les conventions suivantes :

1 Si nous donnons deux nombres de cycles, le plus petit est associé à un opérande registre et le plus grand à un opérande mémoire.

2 n = nombre de répétitions.

3 m = nombre de composantes de la prochaine instruction exécutée, où le déplacement complet compte pour une composante, la donnée immédiate complète pour une composante, et chacun des *autres* octets de l'instruction et préfixes comme une composante.

CODAGE DES INSTRUCTIONS

Le codage des instructions est un sous-ensemble du format général de la Figure G.1. Une instruction consiste en un ou deux octets primaires de CodeOp, plus un éventuel spécificateur d'adresse consistant en un champ de un ou deux octets et un déplacement, plus un éventuel champ de données immédiates.

D'autres champs de codage peuvent être définis à l'intérieur des CodeOp. Ces champs varient selon la classe de l'opération. Ils définissent la

direction de l'opération, la taille du déplacement, le codage des registres ou l'extension de signe.

La plupart des instructions qui font référence à un opérande en mémoire ont un champ MODRM d'un ou deux octets suivant le ou les octets primaires.

Les modes d'adressage peuvent inclure un déplacement suivant immédiatement le champ MODRM. Si ce déplacement est présent, sa taille est de 8, 16 ou 32 bits.

Si l'instruction spécifie un opérande immédiat, ce dernier suit tous les octets de déplacement. L'opérande immédiat, s'il est spécifié, est toujours le dernier champ de l'instruction.

La Figure G.1 montre plusieurs champs qui peuvent apparaître dans une instruction, tels que le champ mod et le champ R/M. Plusieurs autres champs plus petits peuvent apparaître dans certaines instructions, quelquefois à l'intérieur des octets de CodeOp eux-mêmes. Le Tableau G.2 donne une liste complète de tous ces champs.

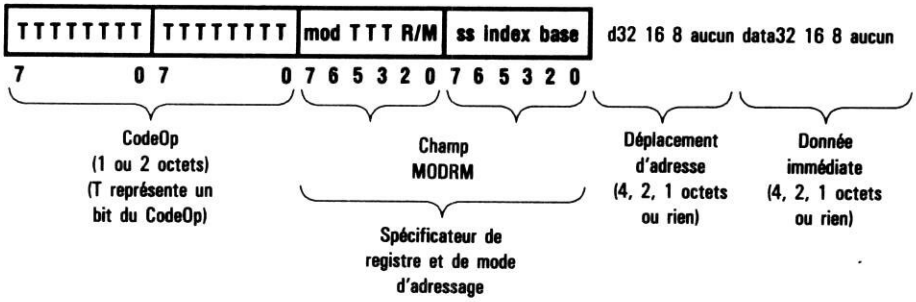


Figure G.1 : *Format général des instructions.*

CHAMP	DESCRIPTION	Nb de BITS
w	Spécifie si la donnée est d'un octet ou de taille normale	1
d	Spécifie la direction	1
s	Spécifie l'extension de signe	1
reg	Spécifie le registre	3
MODRM	Spécifie le mode d'adressage	2 pour mod; 3 pour R/M
ss	Facteur d'échelle	2
base	Registre de base	3
index	Registre d'index	3
sreg2	Registre segment	2
sreg3	Idem + FS et GS	3
tttn	Condition	4

Note : Le Tableau G.1 donne le codage des instructions.

Tableau G.2 : *Champs à l'intérieur des instructions 80386.*

Extensions 32 bits du jeu d'instructions

Avec le 80386, le jeu d'instructions des 86/186/286 est étendu dans deux directions orthogonales : formes 32 bits de toutes les instructions 16 bits et modes d'adressage 32 bits. Cette extension est effectuée par le bit de défaut (D) dans le descripteur de segment code, et par deux préfixes d'instructions.

Ce bit D donne la longueur par défaut (32 bits ou 16 bits) des opérandes et des adresses effectives. Le mode réel ou le mode virtuel 8086 n'utilisent pas des descripteurs de segment code, mais le 80386 se fixe une valeur

de D nulle lorsqu'il travaille dans un de ces modes (ce qui assure la compatibilité 16 bits avec les 8086, 80186, et 80286).

Deux préfixes, le préfixe de taille d'opérande et celui de taille d'adresse effective, permettent de changer localement la taille par défaut des opérandes ou des adresses. Ces préfixes peuvent précéder n'importe quel octet de CodeOp et affectent seulement l'instruction qu'ils précèdent. Si nécessaire, on peut placer un ou plusieurs de ces préfixes après les octets de CodeOp. La présence d'un tel préfixe modifiera la taille de l'opérande ou de l'adresse avec la valeur opposée à la valeur par défaut. Par exemple, si la taille des opérandes par défaut est 32 bits, la présence d'un préfixe de taille d'opérande la fixera à 16 bits. De même, si la taille des adresses est 16 bits, la présence d'un préfixe de taille d'adresse la fixera à 32 bits.

Ces extensions 32 bits sont disponibles pour tous les modes du 80386, même le mode virtuel 8086 et le mode réel. Pour ces modes, le défaut est 16 bits, si bien qu'on utilise des préfixes pour spécifier des tailles 32 bits.

Dans la plupart des cas, des instructions sur des opérandes 8 bits ou 16 bits ne modifient pas les bits de poids fort des registres étendus.

Codage des champs d'instructions

A l'intérieur d'une instruction, il y a plusieurs champs qui indiquent les modes d'adressage, les registres, etc. Nous allons définir le codage exact de ces champs.

Codage du champ (w) de longueur d'opérande

Toute instruction effectuant une opération sur des données travaille sur 16 bits ou 32 bits. Le champ w code la taille de l'opérande d'après la table suivante :

CHAMP W	TAILLE OPERANDE OPERATIONS SUR 16 BITS	TAILLE OPERANDE OPERATIONS SUR 32 BITS
0	8 bits	8 bits
1	16 bits	32 bits

Codage du champ (reg) de registre général

Le champ reg spécifie le registre général. Ce champ peut apparaître dans les octets de CodeOp primaires, comme champ reg de l'octet MODRM, ou comme champ R/M de l'octet MODRM. Le Tableau G.3 montre le codage du champ reg lorsqu'il n'y a pas de champ reg dans l'instruction (voir le Tableau G.4 dans le cas contraire).

CHAMP REG	REG SELECT. OPERATIONS SUR 16 BITS	REG SELECT. OPERATIONS SUR 32 BITS
000	AX	EAX
001	CX	ECX
010	DX	EDX
011	BX	EBX
100	SP	ESP
101	BP	EBP
110	SI	ESI
111	DI	EDI

Tableau G.3 : *Codage du champ reg s'il n'y a pas de champ w.*

CHAMP REG	REG SELECT. OPERATIONS SUR 16 BITS W = 0	W = 1
000	AL	AX
001	CL	CX
010	DL	DX
011	BL	BX
100	AH	SP
101	CH	BP
110	DH	SI
111	BH	DI

CHAMP REG	REG SELECT. OPERATIONS SUR 32 BITS W = 0	W = 1
000	AL	EAX
001	CL	ECX
010	DL	EDX
011	BL	EBX
100	AH	ESP
101	CH	EBP
110	DH	ESI
111	BH	EDI

Tableau G.4 : *Codage du champ reg s'il y a un champ w.*

Codage du champ (sreg) de registre segment

Dans certaines instructions, le champ sreg est un champ de 2 bits qui permet de spécifier un des quatre registres segments du 80286 (voir le Ta-

bleau G.5). Dans d'autres instructions, le champ sreg est un champ de 3 bits qui permet de spécifier les registres segments FS et GS du 80386 (voir le Tableau G.6).

CHAMP sreg2	REGISTRE SEGMENT
00	ES
01	CS
10	SS
11	DS

Tableau G.5 : *Champ sreg2 de 2 bits.*

CHAMP sreg3	REGISTRE SEGMENT
000	ES
001	CS
010	SS
011	DS
100	FS
101	GS
110	ne pas utiliser
111	ne pas utiliser

Tableau G.6 : *Champ sreg3 de 3 bits.*

Déplacement d'adresse

Un déplacement d'adresse suit le champ MODRM. Le sous-champ mod du champ MODRM indique si ce déplacement existe et donne éventuellement sa longueur. Les déplacements n'existent que pour les opérandes mémoire.

Constante immédiate

Si elle existe, une constante immédiate est le dernier champ d'une instruction. C'est-à-dire qu'elle est représentée par les octets d'adresses supérieures de l'instruction. Le CodeOp détermine si une constante immédiate est présente, et définit sa longueur. On peut avoir jusqu'à quatre octets de données immédiates dans ce champ.

Les constantes immédiates

Un opérande *constante immédiate* est le type le plus simple de spécificateur d'opérande, où la valeur de l'opérande est directement donnée dans l'instruction. Le nombre de bits de la constante immédiate dépend de la taille de l'opérande de l'instruction et du CodeOp. Si elle existe, une constante immédiate est toujours le dernier champ d'une instruction, après tout champ CodeOp ou champ de mode d'adressage.

La Figure 2.12 illustre les différentes formes de constantes immédiates. Les constantes peuvent avoir une longueur de 8, 16, ou 32 bits, pour des opérandes de 8, 16, 32 bits. Les formes immédiates 16 et 32 bits sont stockées avec les octets de poids faible aux adresses basses, et les octets de poids fort aux adresses hautes. On peut représenter à la fois des octets, des mots, et des dmots, tous signés ou non, sous ces trois formes. En plus, on peut représenter une petite valeur d'un opérande 16 ou 32 bits comme une valeur 8 bits avec extension de signe, afin d'économiser de l'espace pour

Codage du mode d'adressage

Sauf pour certaines instructions, telles que PUSH ou POP, où le mode d'adressage est prédéterminé, le mode d'adressage de l'instruction courante est spécifié par un champ MODRM d'un ou deux octets, qui suit le ou les octets de CodeOp.

Le mode d'adressage sur 32 bits utilise un champ MODRM sur 2 octets, avec R/M = 100 et mod = 00, 01 ou 10. Le mode d'adressage sur 32 bits est donc une fonction des champs mod, ss, index et base.

Le champ MODRM contient aussi 3 bits (notés TTT à la Figure G.1), utilisés quelquefois comme extension du CodeOp primaire. Cependant, les 3 bits peuvent aussi être employés comme champ registre (reg).

Lors du calcul d'une adresse effective, le processeur utilise soit le mode d'adressage sur 16 bits, soit le mode d'adressage sur 32 bits. Le mode d'adressage sur 16 bits emploie des composantes d'adresses sur 16 bits pour calculer l'adresse effective, alors que le mode d'adressage sur 32 bits utilise des composantes sur 32 bits. En mode d'adressage sur 16 bits (resp. 32 bits), l'octet MODRM est interprété comme un spécificateur de mode d'adressage sur 16 bits (resp. 16 bits).

Le Tableau G.7 définit les codages de tous les modes d'adressage sur 16 bits, et les Tableaux G.8 et G.9 les codages de tous les modes d'adressage sur 32 bits.

MODRM	ADRESSE EFFECTIVE	MODRM	ADRESSE EFFECTIVE
00 000	DS:[BX+SI]	10 000	DS:[BX+SI+d16]
00 001	DS:[BX+DI]	10 001	DS:[BX+DI+d16]
00 010	SS:[BP+SI]	10 010	SS:[BP+SI+d16]

Tableau G.7 : *Codage du mode d'adressage sur 16 bits avec octet MODRM.*

MODRM	ADRESSE EFFECTIVE	MODRM	ADRESSE EFFECTIVE
00 011	SS:[BP+DI]	10 011	SS:[BP+DI+d16]
00 100	DS:[SI]	10 100	DS:[SI+d16]
00 101	DS:[DI]	10 101	DS:[DI+d16]
00 110	DS:d16	10 110	SS:[BP+d16]
00 111	DS:[BX]	10 111	DS:[BX+d16]
01 000	DS:[BX+SI+d8]	11 000	registre
01 001	DS:[BX+DI+d8]	11 001	registre
01 010	SS:[BP+SI+d8]	11 010	registre
01 011	SS:[BP+DI+d8]	11 011	registre
01 100	DS:[SI+d8]	11 100	registre
01 101	DS:[DI+d8]	11 101	registre
01 110	SS:[BP+d8]	11 110	registre
01 111	DS:[BX+d8]	11 111	registre

MODRM	REG SELECT.PAR R/M OPERATIONS SUR 16 BITS W = 0	W = 1
11 000	AL	AX
11 001	CL	CX
11 010	DL	DX
11 011	BL	BX
11 100	AH	SP
11 101	CH	BP
11 110	DH	SI
11 111	BH	DI

Tableau G.7 : *Codage du mode d'adressage sur 16 bits avec octet MODRM (suite).*

MODRM	REG SELECT.PAR R/M OPERATIONS SUR 32 BITS W = 0	W = 1
11 000	AL	EAX
11 001	CL	ECX
11 010	DL	EDX
11 011	BL	EBX
11 100	AH	ESP
11 101	CH	EBP
11 110	DH	ESI
11 111	BH	EDI

Tableau G.7 : *Codage du mode d'adressage sur 16 bits avec octet MODRM (fin).*

MODRM	ADRESSE EFFECTIVE	MODRM	ADRESSE EFFECTIVE
00 000	DS:[EAX]	10 000	DS:[EAX+d32]
00 001	DS:[ECX]	10 001	DS:[ECX+d32]
00 010	DS:[EDX]	10 010	DS:[EDX+d32]
00 011	DS:[EBX]	10 011	DS:[EBX+d32]
00 100	2 octets	10 100	2 octets
00 101	DS:d32	10 101	SS:[EBP+d32]
00 110	DS:[ESI]	10 110	DS:[ESI+d32]
00 111	DS:[EDI]	10 111	DS:[EDI+d32]

Tableau G.8 : *Codage du mode d'adressage sur 32 bits avec un champ MODRM d'un octet.*

01 000	DS:[EAX+d8]	11 000	registre
01 001	DS:[ECX+d8]	11 001	registre
01 010	DS:[EDX+d8]	11 010	registre
01 011	DS:[EBX+d8]	11 011	registre
01 100	2 octets	11 100	registre
01 101	SS:[EBP+d8]	11 101	registre
01 110	DS:[ESI+d8]	11 110	registre
01 111	DS:[EDI+d8]	11 111	registre

MODRM	REG SELECT.PAR R/M ET REG OPERATIONS SUR 16 BITS W = 0	W = 1
11 000	AL	AX
11 001	CL	CX
11 010	DL	DX
11 011	BL	BX
11 100	AH	SP
11 101	CH	BP
11 110	DH	SI
11 111	BH	DI

MODRM	REG SELECT.PAR R/M ET REG OPERATIONS SUR 32 BITS W = 0	W = 1
11 000	AL	EAX
11 001	CL	ECX
11 010	DL	EDX
11 011	BL	EBX
11 100	AH	ESP
11 101	CH	EBP
11 110	DH	ESI
11 111	BH	EDI

Tableau G.8 : *Codage du mode d'adressage sur 32 bits avec un champ MODRM d'un octet (fin).*

BASE MOD	ADRESSE EFFECTIVE
00 000	DS:[EAX+(index à l'échelle)]
00 001	DS:[ECX+(index à l'échelle)]
00 010	DS:[EDX+(index à l'échelle)]
00 011	DS:[EBX+(index à l'échelle)]
00 100	SS:[ESP+(index à l'échelle)]
00 101	DS:[d32+(index à l'échelle)]
00 110	DS:[ESI+(index à l'échelle)]
00 111	DS:[EDI+(index à l'échelle)]
01 000	DS:[EAX+(index à l'échelle)+d8]
01 001	DS:[ECX+(index à l'échelle)+d8]
01 010	DS:[EDX+(index à l'échelle)+d8]
01 011	DS:[EBX+(index à l'échelle)+d8]
01 100	SS:[ESP+(index à l'échelle)+d8]
01 101	DS:[EBP+(index à l'échelle)+d8]
01 110	DS:[ESI+(index à l'échelle)+d8]
01 111	DS:[EDI+(index à l'échelle)+d8]
10 000	DS:[EAX+(index à l'échelle)+d32]
10 001	DS:[ECX+(index à l'échelle)+d32]
10 010	DS:[EDX+(index à l'échelle)+d32]
10 011	DS:[EBX+(index à l'échelle)+d32]
10 100	SS:[ESP+(index à l'échelle)+d32]
10 101	DS:[EBP+(index à l'échelle)+d32]
10 110	DS:[ESI+(index à l'échelle)+d32]
10 111	DS:[EDI+(index à l'échelle)+d32]

SS	FACTEUR D'ECHELLE
00	x 1
01	x 2
10	x 4
11	x 8

Tableau G.9 : *Codage du mode d'adressage sur 32 bits avec un champ MODRM de deux octets.*

INDEX	REGISTRE D'INDEX
000	EAX
001	ECX
010	EDX
011	EBX
100	**
101	EBP
110	ESI
111	EDI

** *Note importante :*
Lorsqu'un champ d'index est égal à 100, indiquant "pas de registre d'index", alors le champ ss DOIT être égal à 00. Si l'index est égal à 100 et ss différent de 00, l'adresse effective est indéfinie.

Tableau G.9 : *Codage du mode d'adressage sur 32 bits avec un champ MODRM de deux octets (fin).*

Codage du champ (d) de direction

Dans de nombreuses instructions à deux opérandes, le champ d sert à indiquer quel opérande est l'origine et lequel est la destination.

d	Direction de l'opération
0	Registre/Mémoire ← Registre Le champ "reg" indique l'opérande d'origine "mod R/M" ou "mod ss index base" indique l'opérande de destination
1	Registre ← Registre/Mémoire Le champ "reg" indique l'opérande de destination "mod R/M" ou "mod ss index base" indique l'opérande d'origine

Codage du champ (s) d'extension de signe

Le champ s intervient dans les instructions comportant des champs immédiats. Il n'a d'effet que si la taille de la donnée immédiate est de 8 bits et qu'elle est placée dans une destination 16 bits ou 32 bits.

s	Effet sur Data8 immédiate	Effet sur Data16/32 immédiate
0	Aucun	Aucun
1	Extension de signe	Aucun

Codage du champ (tttn) de test conditionnel

Pour les instructions conditionnelles (sauts conditionnels, etc.), tttn est codé avec n qui représente la condition (n = 0) ou la négation de cette condition (n = 1) et ttt qui donne la condition à tester (voir le Tableau G.10).

Codage du champ (eee) des registres de contrôle ou de mise au point

Pour le chargement et le rangement des registres de test, de mise au point et de contrôle, le codage est le suivant :

1 Si champ de registre de contrôle :

Code eee	Nom du reg
000	CR0
010	CR2
011	CR3

MNEMONIQUE	CONDITION	tttn
O	Débordement	0000
NO	Pas de débordement	0001
B/NAE	Inférieur	0010
NB/AE	Supérieur ou égal	0011
E/Z	Egal/Nul	0100
NE/NZ	Différent/Non nul	0101
BE/NA	Inférieur ou égal	0110
NBE/A	Supérieur	0111
S	Signe	1000
NS	Pas de signe	1001
P/PE	Parité	1010
NP/PO	Pas de parité	1011
L/NGE	Plus petit	1100
NL/GE	Plus grand ou égal	1101
LE/NG	Plus petit ou égal	1110
NLE/G	Plus grand	1111

Tableau G.10 : *Codage du champ de test conditionnel.*

2 Si champ de registre de mise au point :

Code eee	Nom du reg
000	DR0
001	DR1
010	DR2
011	DR3
110	DR6
111	DR7

3 Si champ de registre de test :

Code eee	Nom du reg
110	TR6
111	TR7

N'utilisez aucun autre codage.

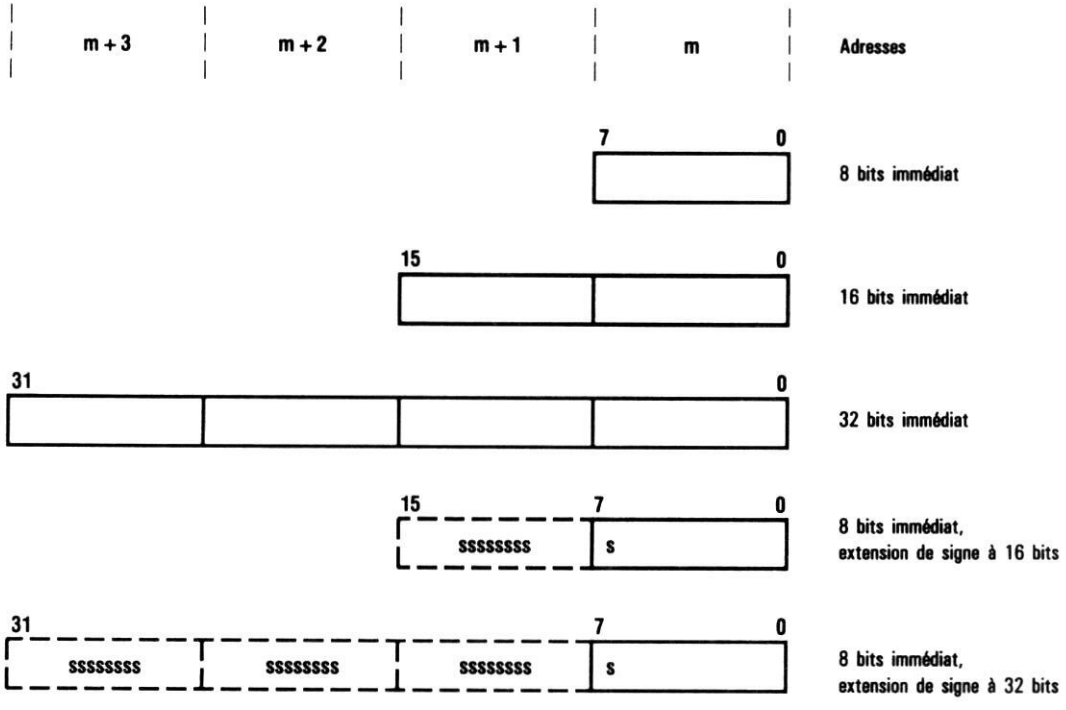


Figure 2.12 : *Codage d'un opérande immédiat.*

l'instruction. Dans ce cas, une constante 8 bits est donnée directement dans l'instruction, et est étendue à 16 ou 32 bits en copiant le bit de signe, le bit 7, vers les 8 ou 24 bits de poids fort. Cela économise de l'espace dans les nombreux cas de mots ou dmots signés avec des valeurs entre -128 et

ANNEXE H

Guide de désassemblage

Premier octet		Second octet	Octets 3-7	Format de l'instruction ASM386	
Hex	Binaire				
D8	1101 1000	MOD 000 R/M	SIB, displ	FADD	single-real
D8	1101 1000	MOD 001 R/M	SIB, displ	FMUL	single-real
D8	1101 1000	MOD 010 R/M	SIB, displ	FCOM	single-real
D8	1101 1000	MOD 011 R/M	SIB, displ	FCOMP	single-real
D8	1101 1000	MOD 100 R/M	SIB, displ	FSUB	single-real
D8	1101 1000	MOD 101 R/M	SIB, displ	FSUBR	single-real
D8	1101 1000	MOD 110 R/M	SIB, displ	FDIV	single-real
D8	1101 1000	MOD 111 R/M	SIB, displ	FDIVR	single-real
D8	1101 1000	1100 0 REG		FADD	ST,ST(i)
D8	1101 1000	1100 1 REG		FMUL	ST,ST(i)
D8	1101 1000	1101 0 REG		FCOM	ST(i)
D8	1101 1000	1101 1 REG		FCOMP	ST(i)
D8	1101 1000	1110 0 REG		FSUB	ST,ST(i)
D8	1101 1000	1110 1 REG		FSUBR	ST,ST(i)
D8	1101 1000	1111 0 REG		FDIV	ST,ST(i)
D8	1101 1000	1111 1 REG		FDIVR	ST,ST(i)
D9	1101 1001	MOD 000 R/M	SIB, displ	FLD	single-real
D9	1101 1001	MOD 001 R/M		reserved	
D9	1101 1001	MOD 010 R/M	SIB, displ	FST	single-real
D9	1101 1001	MOD 011 R/M	SIB, displ	FSTP	single-real
D9	1101 1001	MOD 100 R/M	SIB, displ	FLDENV	14 or 28 bytes***
D9	1101 1001	MOD 101 R/M	SIB, displ	FLDCW	2 bytes
D9	1101 1001	MOD 110 R/M	SIB, displ	FSTENV	14 or 28 bytes***
D9	1101 1001	MOD 111 R/M	SIB, displ	FSTCW	2 bytes
D9	1101 1001	1100 0 REG		FLD	ST(i)

Reprinted by permission of Intel Corporation, copyright 1987.

Premier octet		Second octet	Octets 3-7	Format de l'instruction ASM386
Hex	Binaire			
D9	1101 1001	1100 1 REG		FXCH ST(i)
D9	1101 1001	1101 0000		FNOP
D9	1101 1001	1101 0001		reserved
D9	1101 1001	1101 001–		reserved
D9	1101 1001	1101 01––		reserved
D9	1101 1001	1101 1 REG		*(1)
D9	1101 1001	1110 0000		FCHS
D9	1101 1001	1110 0001		FABS
D9	1101 1001	1110 001–		reserved
D9	1101 1001	1110 0100		FTST
D9	1101 1001	1110 0101		FXAM
D9	1101 1001	1110 011–		reserved
D9	1101 1001	1110 1000		FLD1
D9	1101 1001	1110 1001		FLDL2T
D9	1101 1001	1110 1010		FLDL2E
D9	1101 1001	1110 1011		FLDPI
D9	1101 1001	1110 1100		FLDLG2
D9	1101 1001	1110 1101		FLDLN2
D9	1101 1001	1110 1110		FLDZ
D9	1101 1001	1110 1111		reserved
D9	1101 1001	1111 0000		F2XM1
D9	1101 1001	1111 0001		FYL2X
D9	1101 1001	1111 0010		FPTAN
D9	1101 1001	1111 0011		FPATAN
D9	1101 1001	1111 0100		FXTRACT
D9	1101 1001	1111 0101		FPREM1
D9	1101 1001	1111 0110		FDECSTP
D9	1101 1001	1111 0111		FINCSTP
D9	1101 1001	1111 1000		FPREM
D9	1101 1001	1111 1001		FYL2XP1
D9	1101 1001	1111 1010		FSQRT

Premier octet		Second octet	Octets 3-7	Format de l'instruction ASM386	
Hex	Binaire				
D9	1101 1001	1111 1011		FSINCOS	
D9	1101 1001	1111 1100		FRNDINT	
D9	1101 1001	1111 1101		FSCALE	
D9	1101 1001	1111 1110		FSIN	
D9	1101 1001	1111 1111		FCOS	
DA	1101 1010	MOD 000 R/M	SIB, displ	FIADD	short-integer
DA	1101 1010	MOD 001 R/M	SIB, displ	FIMUL	short-integer
DA	1101 1010	MOD 010 R/M	SIB, displ	FICOM	short-integer
DA	1101 1010	MOD 011 R/M	SIB, displ	FICOMP	short-integer
DA	1101 1010	MOD 100 R/M	SIB, displ	FISUB	short-integer
DA	1101 1010	MOD 101 R/M	SIB, displ	FISUBR	short-integer
DA	1101 1010	MOD 110 R/M	SIB, displ	FIDIV	short-integer
DA	1101 1010	MOD 111 R/M	SIB, displ	FIDIVR	short-integer
DA	1101 1010	110- ----		reserved	
DA	1101 1010	1110 0---		reserved	
DA	1101 1010	1110 1000		reserved	
DA	1010 1010	1110 1001		FUCOMPP	
DA	1101 1010	1110 101-		reserved	
DA	1101 1010	1110 11--		reserved	
DA	1101 1010	1111 ----		reserved	
DB	1101 1011	MOD 000 R/M	SIB, displ	FILD	short-integer
DB	1101 1011	MOD 001 R/M	SIB, displ	reserved	
DB	1101 1011	MOD 010 R/M	SIB, displ	FIST	short-integer
DB	1101 1011	MOD 011 R/M	SIB, displ	FISTP	short-integer
DB	1101 1011	MOD 100 R/M	SIB, displ	reserved	
DB	1101 1011	MOD 101 R/M	SIB, displ	FLD	extended-real
DB	1101 1011	MOD 110 R/M	SIB, displ	reserved	
DB	1101 1011	MOD 111 R/M	SIB, displ	FSTP	extended-real
DB	1101 1011	110- ----		reserved	
DB	1101 1011	1110 0000		**(1)	
DB	1101 1011	1110 0001		**(2)	

Premier octet		Second octet	Octets 3-7	Format de l'instruction ASM386	
Hex	Binaire				
DB	1101 1011	1110 0010		FCLEX	
DB	1101 1011	1110 0011		FINIT	
DB	1101 1011	1110 0100		**(3)	
DB	1101 1011	1110 0101		reserved	
DB	1101 1011	1110 011-		reserved	
DB	1101 1011	1110 1---		reserved	
DB	1101 1011	1111 ----		reserved	
DC	1101 1100	MOD 000 R/M	SIB, displ	FADD	double-real
DC	1101 1100	MOD 001 R/M	SIB, displ	FMUL	double-real
DC	1101 1100	MOD 010 R/M	SIB, displ	FCOM	double-real
DC	1101 1100	MOD 011 R/M	SIB, displ	FCOMP	double-real
DC	1101 1100	MOD 100 R/M	SIB, displ	FSUB	double-real
DC	1101 1100	MOD 101 R/M	SIB, displ	FSUBR	double-real
DC	1101 1100	MOD 110 R/M	SIB, displ	FDIV	double-real
DC	1101 1100	MOD 111 R/M	SIB, displ	FDIVR	double-real
DC	1101 1100	1100 0 REG		FADD	ST(i),ST
DC	1101 1100	1100 1 REG		FMUL	ST(i),ST
DC	1101 1100	1101 0 REG		*(2)	
DC	1101 100	1101 1 REG		*(3)	
DC	1101 1100	1110 0 REG		FSUBR	ST(i),ST
DC	1101 1100	1110 1 REG		FSUB	ST(i),ST
DC	1101 1100	1111 0 REG		FDIVR	ST(i),ST
DC	1101 1100	1111 1 REG		FDIV	ST(i),ST
DD	1101 1101	MOD 000 R/M	SIB, displ	FLD	double-real
DD	1101 1101	MOD 001 R/M		reserved	
DD	1101 1101	MOD 010 R/M	SIB, displ	FST	double-real
DD	1101 1101	MOD 011 R/M	SIB, displ	FSTP	double-real
DD	1101 1101	MOD 100 R/M	SIB, displ	FRSTOR	94 or 108 bytes***
DD	1101 1101	MOD 101 R/M	SIB, displ	reserved	
DD	1101 1101	MOD 110 R/M	SIB, displ	FSAVE	94 or 108 bytes***
DD	1101 1101	MOD 111 R/M	SIB, displ	FSTSW	2 bytes

Premier octet		Second octet	Octets 3-7	Format de l'instruction ASM386	
Hex	Binaire				
DD	1101 1101	1100 0 REG		FFREE	ST(i)
DD	1101 1101	1100 1 REG		*(4)	
DD	1101 1101	1101 0 REG		FST	ST(i)
DD	1101 1101	1101 1 REG		FSTP	ST(i)
DD	1101 1101	1110 0 REG		FUCOM	ST(i)
DD	1101 1101	1110 1 REG		FUCOMP	ST(i)
DD	1101 1101	1111 ----		reserved	
DE	1101 1110	MOD 000 R/M	SIB, displ	FIADD	word-integer
DE	1101 1110	MOD 001 R/M	SIB, displ	FIMUL	word-integer
DE	1101 1110	MOD 010 R/M	SIB, displ	FICOM	word-integer
DE	1101 1110	MOD 011 R/M	SIB, displ	FICOMP	word-integer
DE	1101 1110	MOD 100 R/M	SIB, displ	FISUB	word-integer
DE	1101 1110	MOD 101 R/M	SIB, displ	FISUBR	word-integer
DE	1101 1110	MOD 110 R/M	SIB, displ	FIDIV	word-integer
DE	1101 1110	MOD 111 R/M	SIB, displ	FIDIVR	word-integer
DE	1101 1110	1100 0 REG		FADDP	ST(i),ST
DE	1101 1110	1100 1 REG		FMULP	ST(i),ST
DE	1101 1110	1101 0---		*(5)	
DE	1101 1110	1101 1000		reserved	
DE	1101 1110	1101 1001		FCOMPP	
DE	1101 1110	1101 101-		reserved	
DE	1101 1110	1101 11--		reserved	
DE	1101 1110	1110 0 REG		FSUBRP	ST(i),ST
DE	1101 1110	1110 1 REG		FSUBP	ST(i),ST
DE	1101 1110	1111 0 REG		FDIVRP	ST(i),ST
DE	1101 1110	1111 1 REG		FDIVP	ST(i),ST
DF	1101 1111	MOD 000 R/M	SIB, displ	FILD	word-integer
DF	1101 1111	MOD 001 R/M	SIB, displ	reserved	
DF	1101 1111	MOD 010 R/M	SIB, displ	FIST	word-integer
DF	1101 1111	MOD 011 R/M	SIB, displ	FISTP	word-integer
DF	1101 1111	MOD 100 R/M	SIB, displ	FBLD	packed-decimal

Premier octet		Second octet	Octets 3-7	Format de l'instruction ASM386	
Hex	Binaire				
DF	1101 1111	MOD 101 R/M	SIB, displ	FILD	long-integer
DF	1101 1111	MOD 110 R/M	SIB, displ	FBSTP	packed-decimal
DF	1101 1111	MOD 111 R/M	SIB, displ	FISTP	long-integer
DF	1101 1111	1100 0 REG		*(6)	
DF	1101 1111	1100 1 REG		*(7)	
DF	1101 1111	1101 0 REG		*(8)	
DF	1101 1111	1101 1 REG		*(9)	
DF	1101 1111	1110 0000		FSTSW AX	
DF	1101 1111	1110 0001		reserved	
DF	1101 1111	1110 001–		reserved	
DF	1101 1111	1110 01––		reserved	
DF	1101 1111	1110 1–––		reserved	
DF	1101 1111	1111 ––––		reserved	

NOTES

* The marked encodings are ***not*** generated by the language translators. If, however, the 80387 encounters one of these encodings in the instruction stream, it will execute it as follows:

(1) FSTP ST(i)
(2) FCOM ST(i)
(3) FCOMP ST(i)
(4) FXCH ST(i)
(5) FCOMP ST(i)
(6) FFREE ST(i) and pop stack
(7) FXCH ST(i)
(8) FSTP ST(i)
(9) FSTP ST(i)

** The marked encodings can be generated by the language translators; however, the 80387 treats them as FNOP. They correspond to the following 8087 or 80287 instructions:

(1) FENI
(2) FDISI
(3) FSETPM

*** The size of operand transferred depends on the 80386 operand size attribute in effect for the instruction.

ANNEXE I

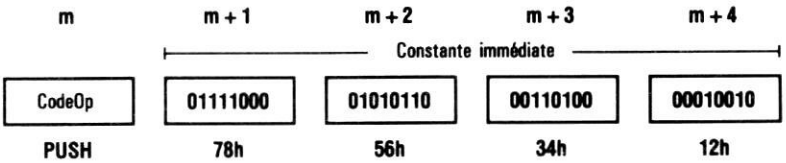
Extensions du 80387 au jeu d'instructions du 80386

127, et des mots (resp. dmots) non signés avec des valeurs entre 0 et 127 ou FF80h (resp. FFFFFF80h) et FFFFh (resp. FFFFFFFFh).

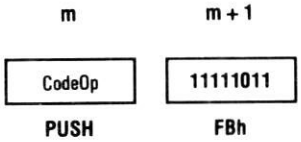
Exemple

L'instruction qui empile une constante immédiate emploie une forme d'instruction qui possède un CodeOp d'un octet et un ou quatre octets de données immédiates. Cette instruction est un bon exemple d'illustration de ce point particulier, parce qu'il n'a qu'un opérande et qu'une constante immédiate. Dans cet exemple, les adresses mémoires sont indiquées à partir du sommet avec m, m + 1, ..., m + 4. Une interprétation des bits de l'instruction est donnée juste au-dessous de ces derniers. Notons que le CodeOp est l'octet d'adresse inférieure et que la constante est stockée dans les bits de poids faible aux adresses inférieures.

Ainsi, PUSH 12345678h serait codé de la façon suivante :



Et PUSH -5 serait codé ainsi :



Notons l'emploi d'une constante immédiate courte d'un octet dans l'exemple précédent, ce qui permet de placer une constante courte de -128 à 127 dans un octet unique de l'instruction. Lorsque cette dernière est exécutée, la constante sur un octet FBh est étendue (extension de signe) à FFFFFFFBh, qui est la représentation en complément à deux de -5.

Les instructions du 80387 suivent une des cinq formes du Tableau I.1. Dans tous les cas, les instructions ont une longueur minimale de deux octets et commencent avec le mot binaire 11011b, qui identifie une instruction à virgule flottante. Les instructions qui font référence à des opérandes mémoire spécifient des adresses qui utilisent les modes d'adressage du 80386.

Le Tableau I.2 suit les règles suivantes :

1 L'instruction a été traitée et décodée. Elle est prête pour l'exécution.

2 Les cycles de bus ne nécessitent pas d'états d'attente.

3 Aucune demande HOLD du bus local ne ralentit les accès du processeur au bus.

4 Aucune exception n'est détectée lors de l'exécution.

5 Si une adresse effective est calculée, elle n'emploie pas deux registres généraux. Les cycles d'horloge sont donnés pour une adresse formée par un registre, un facteur d'échelle et un déplacement. Cependant, si le calcul de l'adresse effective utilise deux registres généraux, ajoutez 1 cycle d'horloge au nombre de cycles donné.

Tableau I.1 : Formes des instructions du 80387

	Instruction										Champs optionnels	
	Premier octet				Second octet							
1	11011	OPA		1	MOD		1	OPB		R/M	SIB	DISP
2	11011	MF		OPA	MOD		OPB			R/M	SIB	DISP
3	11011	d	P	OPA	1	1	OPB			ST(i)		
4	11011	0	0	1	1	1	1	OP				
5	11011	0	1	1	1	1	1	OP				
	15–11	10	9	8	7	6	5	4	3	2 1 0		

NOTES

OP = Instruction opcode, possibly split into two fields, OPA and OPB

MF = Memory Format
00—32-bit real
01—32-bit integer
10—64-bit real
11—64-bit integer

P = Pop
0—Do not pop stack
1—Pop stack after operation

ESC = 11011

d = Destination
0—Destination is ST(0)
1—Destination is ST(i)

R XOR d = 0—Destination (op) Source

R XOR d = 1—Source (op) Destination

ST(i) = Register stack element *i*
000 = Stack top
001 = Second stack element
⋮
111 = Eighth stack element

MOD (mode field) and R/M (register/memory specifier) have the same interpretation as the corresponding fields of 80386 instructions (refer to Appendix G).

SIB (Scale Index Base) byte and DISP (displacement) are optionally present in instructions that have MOD and R/M fields. Their presence depends on the values of MOD and R/M, as for 80386 instructions.

Tableau I.2 : Extension 80387 au jeu d'instructions du 80386

Instruction	Codage			Cycles d'horloge			
	Octet 0	Octet 1	Octets-opt 2-6	Réel 32 bits	Entier 32 bits	Réel 64 bits	Entier 16 bits
DATA TRANSFER							
FLD = Load[a]							
Integer/real memory to ST(0)	ESC MF 1	MOD 000 R/M	SIB/DISP	20	45–52	25	61–65
Long integer memory to ST(0)	ESC 111	MOD 101 R/M	SIB/DISP		56–67		
Extended real memory to ST(0)	ESC 011	MOD 101 R/M	SIB/DISP		44		
BCD memory to ST(0)	ESC 111	MOD 100 R/M	SIB/DISP		266–275		
ST(i) to ST(0)	ESC 001	11000 ST(i)			14		
FST = Store							
ST(0) to integer/real memory	ESC MF 1	MOD 010 R/M	SIB/DISP	44	79–93	45	82–95
ST(0) to ST(i)	ESC 101	11010 ST(i)			11		
FSTP = Store and Pop							
ST(0) to integer/real memory	ESC MF 1	MOD 011 R/M	SIB/DISP	44	79–93	45	82–95
ST(0) to long integer memory	ESC 111	MOD 111 R/M	SIB/DISP		80–97		
ST(0) to extended real	ESC 011	MOD 111 R/M	SIB/DISP		53		
ST(0) to BCD memory	ESC 111	MOD 110 R/M	SIB/DISP		512–534		
ST(0) to ST(i)	ESC 101	11001 ST (i)			12		
FXCH = Exchange							
ST(i) and ST(0)	ESC 001	11001 ST(i)			18		
COMPARISON							
FCOM = Compare							
Integer/real memory to ST(0)	ESC MF 0	MOD 010 R/M	SIB/DISP	26	56–63	31	71–75
ST(i) to ST(0)	ESC 000	11010 ST(i)			24		
FCOMP = Compare and pop							
Integer/real memory to ST	ESC MF 0	MOD 011 R/M	SIB/DISP	26	56–63	31	71–75
ST(i) to ST(0)	ESC 000	11011 ST(i)			26		
FCOMPP = Compare and pop twice							
ST(1) to ST(0)	ESC 110	1101 1001			26		
FTST = Test ST(0)	ESC 001	1110 0100			28		
FUCOM = Unordered compare	ESC 101	11100 ST(i)			24		
FUCOMP = Unordered compare and pop	ESC 101	11101 ST(i)			26		
FUCOMPP = Unordered compare and pop twice	ESC 010	1110 1001			26		
FXAM = Examine ST(0)	ESC 001	11100101			30-38		
CONSTANTS							
FLDZ = Load +0.0 into ST(0)	ESC 001	1110 1110			20		
FLD1 = Load +1.0 into ST(0)	ESC 001	1110 1000			24		
FLDPI = Load pi into ST(0)	ESC 001	1110 1011			40		
FLDL2T = Load $\log_2(10)$ into ST(0)	ESC 001	1110 1001			40		

Shaded areas indicate instructions not available in 80287/8087.

NOTE

a. When loading single- or double-precision zero from memory, add 5 clocks.

Tableau I.2 : Extension 80387 au jeu d'instructions du 80386

Instruction	Codage			Cycles d'horloge			
	Octet 0	Octet 1	Octets-opt 2-6	Réel 32 bits	Entier 32 bits	Réel [redacted]	Entier 16 bits
CONSTANTS (Continued)							
FLDL2E = Load $\log_2(e)$ into ST(0)	ESC 001	1110 1010		40			
FLDLG2 = Load $\log_{10}(2)$ into ST(0)	ESC 001	1110 1100		41			
FLDLN2 = Load $\log_e(2)$ into ST(0)	ESC 001	1110 1101		41			
ARITHMETIC							
FADD = Add							
Integer/real memory with ST(0)	ESC MF 0	MOD 000 R/M	SIB/DISP	24–32	57–72	29–37	71–85
ST(i) and ST(0)	ESC d P 0	11000 ST(i)		23–31[b]			
FSUB = Subtract							
Integer/real memory with ST(0)	ESC MF 0	MOD 10 R R/M	SIB/DISP	24–32	57–82	28–36	71–83[c]
ST(i) and ST(0)	ESC d P 0	1110 R R/M		26–34[d]			
FMUL = Multiply							
Integer/real memory with ST(0)	ESC MF 0	MOD 001 R/M	SIB/DISP	27–35	61–82	32–57	76–87
ST(i) and ST(0)	ESC d P 0	1100 1 R/M		29–57[e]			
FDIV = Divide							
Integer/real memory with ST(0)	ESC MF 0	MOD 11 R R/M	SIB/DISP	89	120–127[f]	94	136–140[g]
ST(i) and ST(0)	ESC d P 0	1111 R R/M		88[h]			
FSQRT[i] = Square root	ESC 001	1111 1010		122–129			
FSCALE = Scale ST(0) by ST(1)	ESC 001	1111 1101		67–86			
FPREM = Partial remainder	ESC 001	1111 1000		74–155			
FPREM1 = Partial remainder (IEEE)	ESC 001	1111 0101		[redacted]			
FRNDINT = Round ST(0) to integer	ESC 001	1111 1100		66–80			
FXTRACT = Extract components of ST(0)	ESC 001	1111 0100		70–76			
FABS = Absolute value of ST(0)	ESC 001	1110 0001		22			
FCHS = Change sign of ST(0)	ESC 001	1110 0000		24–25			

Shaded areas indicate instructions not available in 80287/8087.

NOTES

b. Add 3 clocks to the range when d = 1.
c. Add 1 clock to *each* range when R = 1.
d. Add 3 clocks to the range when d = 0.
e. typical = 52 (When d = 0, 46–54, typical = 49).
f. Add 1 clock to the range when R = 1.
g. 135–141 when R = 1.
h. Add 3 clocks to the range when d = 1.
i. $-0 \leqslant ST(0) \leqslant +\infty$.

Tableau I.2 : Extension 80387 au jeu d'instructions du 80386

Instruction	Codage			Cycles d'horloge
	Octet 0	Octet 1	Octets-opt 2-6	
TRANSCENDENTAL				
FCOS[k] = Cosine of ST(0)	ESC 001	1111 1111		123–772[j]
FPTAN[k] = Partial tangent of ST(0)	ESC 001	1111 0010		191–497[j]
FPATAN = Partial arctangent	ESC 001	1111 0011		314–487
FSIN[k] = Sine of ST(0)	ESC 001	1111 1110		122–771[j]
FSINCOS[k] = Sine and cosine of ST(0)	ESC 001	1111 1011		194–809[j]
F2XM1[l] = $2^{ST(0)} - 1$	ESC 001	1111 0000		211–476
FYL2X[m] = ST(1) * $\log_2$(ST(0))	ESC 001	1111 0001		120–538
FYL2XP1[n] = ST(1) * $\log_2$(ST(0) + 1.0)	ESC 001	1111 1001		257–547
PROCESSOR CONTROL				
FINIT = Initialize NPX	ESC 011	1110 0011		33
FSTSW AX = Store status word	ESC 111	1110 0000		13
FLDCW = Load control word	ESC 001	MOD 101 R/M	SIB/DISP	19
FSTCW = Store control word	ESC 101	MOD 111 R/M	SIB/DISP	15
FSTSW = Store status word	ESC 101	MOD 111 R/M	SIB/DISP	15
FCLEX = Clear exceptions	ESC 011	1110 0010		11
FSTENV = Store environment	ESC 001	MOD 110 R/M	SIB/DISP	103–104
FLDENV = Load environment	ESC 001	MOD 100 R/M	SIB/DISP	71
FSAVE = Save state	ESC 101	MOD 110 R/M	SIB/DISP	375–376
FRSTOR = Restore state	ESC 101	MOD 100 R/M	SIB/DISP	308
FINCSTP = Increment stack pointer	ESC 001	1111 0111		21
FDECSTP = Decrement stack pointer	ESC 001	1111 0110		22
FFREE = Free ST(i)	ESC 101	1100 0 ST(i)		18
FNOP = No operations	ESC 001	1101 0000		12

Shaded areas indicate instructions not available in 80287/8087.

NOTES

j. These timings hold for operands in the range $|x| < \pi/4$. For operands not in this range, up to 76 additional clocks may be needed to reduce the operand.
k. $0 \leq |ST(0)| < 2^{63}$.
l. $-1.0 \leq ST(0) \leq 1.0$.
m. $0 \leq ST(0) < -\infty$, $-\infty < ST(1) < +\infty$
n. $0 \leq |ST(0)| < (2 - SQRT(2))/2$, $-\infty < ST(1) < +\infty$.

INDEX

$GP, 675, 690, 697, 711, 755, 775
$NP, 674, 700, 755, 775
$SS, 674, 700, 755
$TS, 674, 690, 697, 711, 754
80186, 869
80286, 869, 912, 924
80287, 929
80387, 20, 23, 27, 97
8086, 869, 872, 880, 919
8087, 929, 935

A, 628
AAA, 155
AAD, 157
AAM, 159
AAS, 161
Abandon de la procédure, 238
ADC, 163
ADD, 165
Addition, 165, 428
    avec retenue, 163
Adresse
    de base, 597
    physique, 581, 810
    virtuelle, 582
AF, 44, 137, 143
AH, 142
Ajustement ASCII
    après l'addition, 155
    après la multiplication, 159
    après soustraction, 161
    avant la division, 157
Ajustement décimal
    après l'addition, 203
    après soustraction, 205
Ajustement du niveau de privilège requis, 377
AL, 141
Alias, 644
Alignement, 57
Allocation, 540
AND, 167
Anormal, 932
Appel
    à procédure d'interruption sur débordement, 357
    à un sous-programme, 187
    à une procédure d'interruption, 354
    de procédure, 351
Arctangente partiel, 477
ARPL, 377
Arrondi à l'entier, 487
ASCII, 13
ASSUME, 800
Attente pour le coprocesseur, 532
Attributs de segment, 597
AVL, 610, 614, 629
AX, 41, 141, 870

B0-B3, 857
Base, 90-91, 606
BCD, 17, 23
BD, 855, 857
Benchmarks, 553
BH, 142
Big endian, 6
Bit
    Available-to-Software, 610
    Communication de tâches, 638
    de granulité, 613
    Dirty, 629
    Emulation, 639
    Math présent, 639
    offset, 15
    présent, 608, 613-615, 627
    Type, 612, 764
    Type d'extension, 638
BL, 142
Blocs binaires, 557
Boucle avec compteur ECX, 245
Boucle avec ECX et test si non nul, 247
Boucle avec ECX et test si nul, 249
BOUND, 169, 751
BP, 41, 141, 870
Branchement, 361
BS, 858
BSF, 171
BSR, 173
BT, 175, 858
BTC, 178
BTR, 181
BTS, 184
Bubble sort, 545
Busy waiting, 552
BX, 41, 141, 870

C0, 108, 143
C1, 108, 143, 790, 794
C2, 108, 143
C3, 108, 143
Cache de pagination, 634
CALL, 59, 60, 187, 351, 713

Call gate, 665
Caractérisation, 42
Catalogue de pages, 621, 819
CBW/CWDE, 189
CF, 44, 137, 143
CH, 142
Chaînes, 13
Champ
de liens, 647
eee, 991
reg, 982
s, 991
sreg, 983
tttn, 991
w, 981
Changement de signe, 435
Chargement
constante, 468
d'entier, 456
d'un pointeur, 363
d'une chaîne, 242
de BCD, 431
de l'adresse effective, 236
de l'environnement, 471
de la limite de segment, 394
de la table des descripteurs d'interruptions, 388
de la table des descripteurs globaux, 386
de la table des descripteurs locaux, 390
de réel, 466
des droits d'accès, 382
des indicateurs dans AH, 234
du mot d'état machine, 392
du mot de contrôle, 470
du registre de tâche, 398
Chargeur, 826
CL, 142
CLC, 191
CLI, 193, 653, 660, 884
CLTS, 379
CMC, 195
CMP, 196
CMPS/CMPSB/CMPSW/CMPSD, 198, 297, 554
Codage des instructions, 72
CodeOp, 74
Commutation de piles, 667
Commutation de tâches, 727, 735, 812
Comparaison, 196, 438
d'entiers, 453
de chaînes, 198
logique, 340
non ordonnée, 519
Complément à deux, 8, 11, 262
Complément à un, 8, 10
Complémentation du bit de retenue, 195
Constante immédiate, 75
Contexte, 847
Contrôle
d'arrondi, 105
de précision, 104
Conversion
d'octet en mot, 189
de mot en dmot, 201
Coprocesseur arithmétique, 20
Cosinus, 441
CPL, 143, 589
CR0, 142, 379, 392, 408, 636-640, 685
CR1, 636
CR2, 142, 636, 641, 676
CR3, 142, 623, 629, 635-636, 641, 738
Création du cadre de pile, 212
CS, 50, 142
CW, 142, 462, 470
CWD/CDQ, 201
CX, 41, 141, 870

D, 610, 629, 875, 980
DAA, 203
DAS, 205
DE, 791
Débordement, 751
DEC, 207
Décalage
à droite double, 329
à gauche double, 324
arithmétique à droite, 312
arithmétique à gauche, 309
logique à droite, 327
Décrémentation, 207
Décrémentation du pointeur de pile, 443
Dénormalisé, 28-31, 791, 933
Dépilage, 275
dans le registre segment, 367
de tous les registres, 277
des indicateurs, 279
Déplacement, 57, 541
Descripteur de segment, 599, 605
DF, 48, 143
DH, 142
DI, 41, 141, 870

Les opérandes registres

Un opérande d'une instruction peut être situé dans un registre du processeur. Le spécificateur d'un opérande registre sélectionne un des huit registres généraux, ou un des six registres segments, en fonction du CodeOp. La taille de l'opérande détermine si on accède à un registre 8, 16, 32 bits pour un registre général. On accède toujours aux registres segments par des registres 16 bits.

La Figure 2.13 montre comment les spécificateurs d'un opérande registre sont codés dans une instruction. Les formes d'instruction à un octet utilisent les trois bits de poids faible de l'instruction pour coder un des huit registres généraux. Les formes d'instructions à deux ou trois octets emploient un octet MODRM qui suit le CodeOp pour coder un ou deux opérandes registre. Cet octet est composé de trois champs. Le champ R/M est représenté par les trois bits de poids faible, et indique si on a un opérande registre ou mémoire, selon l'état du champ mod. Le champ mod comprend deux bits de poids fort. Il détermine si le champ R/M doit être interprété comme un nom de registre sous 3 bits ou comme un opérande mémoire. Si le champ mod a la valeur 11b (les deux bits à 1), le champ R/M détermine un opérande registre, sinon il s'agit d'un opérande mémoire. Le terme R/M provient de la double interprétation de ce champ en tant qu'opérande *Registre* ou *Mémoire*. On appelle cet octet l'octet MODRM, selon les noms du premier et du dernier champ, qui existent toujours. Les trois bits du centre peuvent déterminer un opérande registre, ou apporter 3 bits de CodeOp supplémentaires, selon le CodeOp du premier octet de l'instruction.

Dans tous ces cas, les opérandes registre sont spécifiés par un code 3 bits dans l'instruction. Le Tableau 2.3 donne le codage des registres généraux 8, 16, 32 bits basé sur la taille de l'opérande et le spécificateur de registre. Le Tableau 2.4 fournit un tableau identique pour les registres segments.

DIV, 209, 750, 878
Division, 445
arrière, 448
non signée, 209
par zéro, 792
signée, 214, 543
DL, 142
Dmot, 6
DmotCompt, 616, 669, 719, 764
DNA, 638, 752, 844
Double faute, 753
DPL, 143, 608-615, 763
DR0, 142, 855
DR1, 142, 855
DR2, 142, 855
DR3, 142, 855
DR6, 142, 750, 855-857
DR7, 142, 855-858
DS, 50, 65, 142
DT, 611, 616, 764
DType, 609, 613, 708
DUP, 540
DX, 41, 141, 870
Dynamique, 66

EAX, 41, 141
EBP, 41, 141
EBX, 41, 141
Echange, 342
Echange de registres, 524
Echec, 748, 750
Echelle de puissance de deux, 493
ECX, 41, 141, 293-297
EDI, 41, 141
EDX, 41, 141
Effacement
de l'indicateur de commutation de tâches, 379
des exceptions, 436
EFLAGS, 43, 142, 660
EIP, 43, 142
EM, 638, 639, 752, 844
Empilage, 281, 369
de tous les registres, 283
des indicateurs, 285
Emulation, 846, 880
ENDP, 542
ENDS, 799
ENTER, 212, 554, 564
Entiers signés, 8
Entrée
E/S, 221
d'une chaîne, 225
EQU, 542
Erreur
de division, 750
de pile, 108
du coprocesseur, 758, 787, 923
ES, 50, 65, 142, 932
ESI, 41, 141
ESP, 41, 59, 141
Espace adressable
global, 587
linéaire, 583, 806
local, 588
Espace E/S, 96
ET, 638
ET logique, 167
Etiquettes, 541
Examen, 522
Exception, 36, 137, 580, 745, 748, 773
du type page, 676
du type segment, 674
Expand down, 608, 684
Expand up, 608, 684
Exponentielles, 568
Exposant, 24
EXT, 675
Extraction de l'exposant et de la mantisse, 526

F2XM1, 425, 568, 929
FABS, 427, 790
Factorielle, 547
FADD/FIADD/FADDP, 428
Faute, 748
Faute du type mise au point, 854
FBLD, 431
FBSTP, 433, 788-790, 930
FCHS, 435, 790
FCLEX/FNCLEX, 436, 789
FCOM/FCOMP/FCOMPP, 438, 512, 567
FCOS, 441
FCS, 98, 109, 788-789
FCX, 142
FDECSTP, 443
FDISI/FNDISI, 935
FDIV/FIDIV/FDIVP, 445, 792, 930, 933
FDIVR/FIDIVR/FDIVRP, 448
FENI/FNENI, 935
FFREE, 451

FICOM/FICOMP, 453, 567
FILD, 456
FINCSTP, 458
FINIT/FNINIT, 460, 754, 847, 934
FIP, 98, 109, 142, 788, 789
FIST(P), 462-464, 788-790, 930
FLAGS, 923
FLD, 466, 789, 930-931
FLDcon, 468
FLDCW, 470
FLDENV, 110-111, 471, 789, 882, 915, 935
FMUL/FIMUL/FMULP, 473
FNINIT, 848
FNOP, 476
FNSAVE, 110-112, 882, 915
FNSTENV, 110-111, 882, 915
FOO, FOS, 98, 109, 143, 788
FPATAN, 477, 929
FPREM, 479, 567, 930, 933-934
FPREM1, 482, 567
FPTAN, 485, 929
FRNDINT, 487
FRSTOR, 110-112, 489, 789, 847, 882, 915, 935
FS, 50, 142, 870
FSAVE/FNSAVE, 110-112, 489, 491, 788, 882, 915
FSCALE, 493, 929
FSETPM, 931
FSIN, 495
FSINCOS, 497
FSQRT, 499, 930, 933
FST(P), 501, 506,788-792
FSTCW/FNSTCW, 503
FSTENV/FNSTENV, 110-111, 504, 788, 882, 915
FSTP, 506
FSTSW AX/FNSTSW AX, 509, 565
FSTSW/FNSTSW, 508
FSUB/FISUB/FSUBP, 511
FSUBR/FISUBR/FSUBRP, 514
FTST, 517, 567
FUCOM/FUCOMP/FUCOMPP, 519, 567
FXAM, 522, 790
FXCH, 524, 790
FXTRACT, 526, 792, 931-933
FYL2X, 528, 792
FYL2XP1, 530

G, 610, 613
GD, 855, 860
GDT, 386, 402, 599, 815
GDTR, 142, 636, 642
GE/LE, 859
Gestion
    mémoire, 579
    multitâches, 579
Granulité, 394
GS, 50, 142, 870
Guichet, 613
    d'appel, 665, 716, 720
    d'interruption, 765, 776
    de tâche, 729
    de trappe, 765, 776

Halte, 381
Handler, 354, 357, 745, 844
Handling, 745
Heap, 67
HLT, 381
HOLD, 963, 1005

ID, 907
IDIV, 214, 543, 750, 878
IDT, 388, 404, 642, 675, 762, 818
IDTR, 142, 636, 642, 762
IE, 932
IF, 48, 143, 660, 747, 761
IM, 932
IMUL, 218
IN, 221, 653
INC, 223
Incrémentation, 223
Incrémentation du pointeur de pile, 458
Indéfinis, 34
Indicateur
    de débordement, 45
    de direction, 49
    de mode virtuel, 50
    de parité, 44
    de relance, 49
    de retenue, 44
    de retenue auxiliaire, 44
    de signe, 45
    de validation des interruptions, 49
    de zéro, 45
    du mode pas à pas, 48
    à virgule flottante, 565
Infinis, 30-31, 107, 501
Initialisation du processeur, 460
INS/INSB/INSW/INSD, 225, 663
Instructions de saut conditionnel, 228
INT, 354, 745, 763, 766

INT3, 751, 864
Interruption, 745, 773
INTO, 357, 745, 763, 766
INTR, 747, 760
IOPL, 48, 143, 651, 660, 884
IRET, 359, 660, 769, 779

Jcc, 228
JMP, 231, 361, 708

Kernel, 589

L0-L3/G0-G3, 860
Labels, 541
LAHF, 234
LAR, 382
LDS, 65
LDT, 390, 406, 599, 817
LDTR, 142, 636, 642, 700, 738
LEA, 236
LEAVE, 238, 564
LEN, 858
LES, 65
LFS, 65
LGDT, 386, 815
LGS, 65
Libération de registre, 451
LIDT, 388
Limite, 606
Limite de segment, 597
Little endian, 6
LLDT, 390, 700, 817
LMSW, 392
LOCK, 143, 240, 342, 551-552, 635, 752, 923-925
LODS/LODSB/LODSW/LODSD, 242, 554
LOOP, 245
LOOPNZ/LOOPNE, 247
LOOPZ/LOOPE, 249
LSL, 394
Lsr, 363
LTR, 398, 702

Mantisse, 24
Map, 587
Mapping, 582
Masquage, 761, 787
Mécanisme
  de pagination, 581
  de translation, 585
Mémoire
  E/S, 39
  principale, 39
  virtuelle, 585, 631
  Mise à zéro
  de l'indicateur d'interruption, 193
  de l'indicateur de retenue, 191
Mise au point, 853
Mod, 84-86, 88, 90, 91
Mode
  d'adressage, 56, 57
  pas à pas, 865, 923
  protégé, 816, 908, 912
  réel, 814, 873, 880, 905
  virtuel, 8086, 873, 880, 883-886
MODRM, 74, 78, 82-83, 953, 979
Mot, 6
Mot étiquette, 935
MOV, 251, 365, 400
MOVS/MOVSB/MOVSW/MOVSD, 253
MOVSX, 256
MOVZX, 258
MP, 638-639, 753
MSW, 142, 379, 392, 408
MUL, 260
Multiplication, 473
  de matrices, 569
  non signée, 260
  signée, 218
Multiprocesseurs, 635

NaN, 30, 33, 501, 517, 519
  signalé, 33, 36, 932
  silencieux, 33, 932
NEG, 262
Niveau de privilège
  courant, 589
  du descripteur, 608, 763
  externe, 589
  interne, 589
NMI, 143, 747, 760-761, 923
Nombres
  non signés, 8
  réels, 18
NON logique, 266
NOP, 264
Normalisés, 26
Norme virgule flottante IEEE, 20
NOT, 266
Notation à offset, 8, 10

NT, 48, 143, 647, 729, 779, 896

OE, 792
OF, 44, 137, 143, 751
Offset, 541
Opérande, 39
Opération, 39
OR, 268
ORG, 800
OS, 589
OSZAPC, 137
OU
    exclusif logique, 346
    inclusif logique, 268
OUT, 270, 653
OUTS/OUTSB/OUTSW/OUTSD, 272, 653
Overflow, 792, 933

P, 608, 611-615, 635, 676, 755, 763
Pages
    globales, 624
    locales, 624
Pagination, 582, 586, 618, 808, 824
Pas d'opération, 264, 476
PC, 104
PE, 143, 640, 794, 816
PF, 44, 137, 143
PG, 618, 640, 685
Pile
    accumulateur à virgule flottante, 98
    programme, 59
Point d'arrêt, 751, 854
Pointeur, 64, 363
POP, 59, 275, 367
POPA/POPAD, 277
POPF/POPFD, 279, 660
Positionnement
    de l'indicateur d'interruption, 334
    de l'indicateur de direction, 333
    de l'indicateur de retenue, 332
Préfixes, 74, 94
PROC, 542
Protection, 579, 586, 632
    dans une tâche, 588
    entre les tâches, 587
    mutuelle des tâches, 580
Pseudo-dénormalisés, 30, 33
Pseudo-infini, 932
Pseudo-NaN, 932
Pseudo-zéro, 932
PUSH, 59, 281, 369, 922
PUSHA/PUSHAD, 283
PUSHF/PUSHFD, 285

QNaN, 34

R/M, 78, 84-89
R/W, 627, 632
Racine carrée, 499
Rangement de AH dans les indicateurs, 307
RC, 105,106, 462, 464, 501, 792
RCL, 287
RCR, 290
Recherche de bit
    en arrière, 173
    de bit en avant, 171
Recherche de chaîne, 318, 553
Récupération de l'état, 489
Récursivité, 547
Réel
    court, 26
    long, 26
    temporaire, 26, 28
Registre, 39
    caché des descripteurs, 616
    d'étiquette, 103
    d'index, 57
    de base, 57
    de contrôle, 43, 103
    des indicateurs d'état et de contrôle, 43
    général, 41-42
    physique, 100
    pointeur d'instruction, 43
    segment, 50, 54, 593, 690
REP, 225, 272, 293, 336
REPE/REPZ, 198, 285, 318, 554
Répétition, 293
    tant qu'égal, 295
    tant que pas égal, 297
REPNE/REPNZ, 198, 297, 318, 554
Reste partiel, 479, 567
Reste partiel IEEE, 482
RET, 59-60, 299, 371, 722
Retour d'une procédure, 299, 371
Retour d'une procédure d'interruption, 359
RF, 48, 143, 751, 761
ROL, 301
ROR, 304
Rotation
    droite, 304

à droite avec retenue, 290
à gauche, 301
à gauche avec retenue, 287
RPL, 143, 377, 605
RWE, 859

SAHF, 307, 567
SAL/SHL, 309
SAR, 312, 543
Saut, 231
Sauvegarde de l'état, 491
SBB, 315
SCAS/SCASB/SCASW/SCASD, 297, 318, 554
Scheduler, 552
SEGMENT, 799
Segment, 51, 593
code, 54
d'état de la tâche, 642
de données, 55
pile, 55
système, 613
Segmentation, 51, 582, 586, 593, 672
Segments multiples, 579
Sélecteur, 52, 694
nul, 604
de segment, 602
Sémaphore, 550
SETcc, 321
SF, 44, 108, 137, 143, 790
SGDT, 402
Shadow register, 616
SHLD, 324
SHR, 327
SHRD, 329
Shutdown, 753
SI, 41, 141, 870
SIDT, 404
Signe, 24-25
Signe magnitude, 8, 10
Sinus, 495
Sinus et cosinus, 497
SLDT, 406
SMSW, 408
SNaN, 34
Sortie E/S, 270
Sortie E/S d'une chaîne, 272
Soustraction, 338, 511
arrière, 514
avec retenue, 315
SP, 41, 141, 870
SS, 50, 65, 142
ST, 98, 100-102, 142
ST(n), 142
Statique, 66
Statistiques, 573
STC, 332
STD, 333
STI, 334, 653, 660, 761, 884
Stockage
d'entier, 462
d'entier et dépilage, 464
de BCD et Dépilage, 433
de chaîne, 336
de l'environnement, 504
de la table des descripteurs d'interruptions, 404
de la table des descripteurs globaux, 402
de la table des descripteurs locaux, 406
de réel, 501
de réel et dépilage
du mot d'état, 508
du mot d'état dans AX, 509
du mot d'état machine, 408
du mot de contrôle, 503
du registre de tâche, 410
STOS/STOSB/STOSW/STOSD, 336
STR, 410
Structures de données, 66, 676
SUB, 338
SW, 142, 460, 470
Swapping, 626
Systèmes d'exploitation multitâches, 579

T, 863
Table
d'autorisations E/S, 650-654
de pagination, 583
des descripteurs d'interruptions, 642
des descripteurs globaux, 599
des descripteurs locaux, 599
des pages, 620-622, 821
des pages globales, 624
des pages locales, 624
Tâche, 579
Tangente partielle, 485
Tas, 67
TEST, 340
Test, 517
de bit, 175
de bit et remise à zéro, 181
de limite, 751

et complémentation de bit, 178
et positionnement de bit, 184
TF, 48, 143, 865
TI, 602, 675
TLB, 634
TOP, 100, 142
TR, 142, 410, 636, 642, 647, 702
Transfert
avec extension de signe, 256
avec extension de zéro, 258
d'opérandes, 251
d'un registre spécial, 400
de chaînes, 253
de registre segment, 365
Translation Lookaside Buffer, 634
Trappe, 748
de mise au point, 650, 854
Tri à bulles, 545
Troncature, 107
TS, 638, 738, 752
TSS, 398, 642, 825, 863
TW, 142, 460
Type, 538, 612-616

U, 676
U/S, 627, 632
UE, 793
Underflow, 793, 933
USE16, 799
USE32, 799

Valeur absolue, 427
Vérification
des limites d'un tableau, 169
du segment en écriture, 415
du segment en lecture, 412
VERR, 412
Verrouillage du bus, 240
VERW, 415
Virgule flottante, 18-19
VM, 48, 143, 660, 883, 887-889, 901

W, 676
WAIT, 532, 639, 753, 758, 845, 935
Watchdog timer, 886

XCHG, 342
XOR, 346

ZE, 792
Zéros, 30
ZF, 44, 137, 143
Zone de sauvegarde des registres, 650

S Y B E X

DANS LE MONDE ENTIER

FRANCE

6-8, Impasse du Curé
75881 PARIS CEDEX 18
Tél. : (1) 42.03.95.95
Télex : 211801
Télécopie : 42.03.01.45

U.S.A.

2021 Challenger Drive, 100
ALAMEDA - CALIFORNIA 94501
Tél. : (415) 523-8233
(800) 227-2346 (Outside CA)
Télex : 336311

R.F.A.

Vogelsanger Weg 111
Postfach 300961
4000 DÜSSELDORF 30
Tel. : (211) 61802-0
Telex : 8588163

PAYS-BAS

Birkstraat 95
Postbus 3177
3760 DD SOEST
Tél. : (31) 2155 276 25
Télécopie : (31) 2155 265 56

DISTRIBUTION BELGIQUE

Presses de Belgique
96, rue Gray
1040 BRUXELLES

DIFFUSION CANADA

Diffulivre
817, rue Mac Caffrey
SAINT-LAURENT
QUÉBEC H4T 1N3

DIFFUSION SUISSE

Office du Livre
Route de Villars, 101
B.P. 1061
1701 FRIBOURG

SYBEX

Paris • San Francisco • Düsseldorf • Londres • Amsterdam

Achevé d'imprimer le 13 avril 1990 sur les presses de l'Imprimerie «La Source d'Or»
63200 Marsat - Dépôt légal : 2e trimestre 1990 - Imprimeur n° 3621

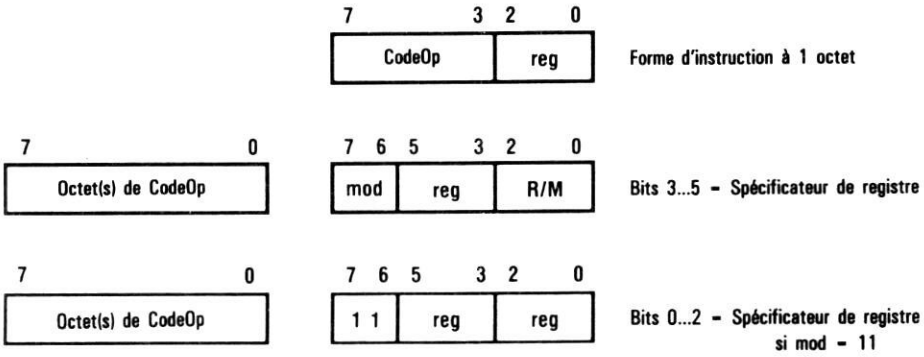


Figure 2.13 : *Codage d'un opérande registre.*

Code reg	Registre 32 Bits	Registre 16 Bits	Registre 8 bits
000b	EAX	AX	AL
001b	ECX	CX	CL
010b	EDX	DX	DL
011b	EBX	BX	BL
100b	ESP	SP	AH
101b	EBP	BP	CH
110b	ESI	SI	DH
111b	EDI	DI	BH

Tableau 2.3 : *Codage des registres généraux.*

micro passion

COLLECTION

80386

Fonctionnement & Programmation

JOHN H. CRAWFORD, architecte en chef du 80386, et PATRICK P. GELSINGER, un des concepteurs de la logique du 80386, traitent dans cet ouvrage des possibilités de programmation de la nouvelle génération de microprocesseurs Intel : le 80386 et le 80387, coprocesseur arithmétique.

Ce livre est une référence indispensable de programmation pour les ingénieurs, les concepteurs de systèmes et les utilisateurs expérimentés de systèmes à base de 80386 - en fait, toute personne qui cherche des informations détaillées sur les possibilités du 80386 et du 80387. On y trouvera :

- Registres, types de données et classes d'instructions ;
- Gestion de la mémoire (segmentation et pagination) ;
- Gestion multitâche, interruptions et exceptions ;
- Système de soutien au debuggage ;
- Exécution des programmes 8086 et 80286.

De plus, l'ouvrage contient :

- Un guide de référence sur les jeux d'instructions du 80386 et du 80387, complètement indexé, expliqué à l'aide d'exemples et de renvois. Les opérations sur chaque instruction sont clairement présentées pour chaque condition possible. Les rubriques sont classées alphabétiquement sous quatre références : instructions sur des entiers, sur la multiple segmentation, sur le système d'exploitation et sur les réels à virgule flottante.
- Deux chapitres entiers donnant des exemples de programmation aussi indispensables que des routines d'initialisation ou des handlers d'exception du coprocesseur.
- Des annexes techniques complètes, incluant des comparaisons détaillées entre les processeurs et coprocesseurs de première, deuxième et troisième génération ; plus un résumé du jeu d'instructions et la table des codes opérations (opcodes) du 80386.

0632 0490 398 F

9 782736 106324

SYBEX 6-8, impasse du Curé, 75018 Paris

Paris • San Francisco • Düsseldorf • Londres • Amsterdam

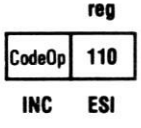
Code reg	Registre Segment
000b	ES
001b	CS
010b	SS
011b	DS
100b	FS
101b	GS
110b	Réservé
111b	Réservé

Tableau 2.4 : *Codage des registres segments.*

Exemple

Le codage le plus simple d'un opérande registre emploie la forme d'instruction sur un octet, qui place un code registre dans les 3 bits de poids faible et le CodeOp dans les 5 bits de poids fort de l'octet.

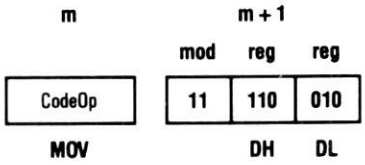
INC ESI serait alors codé comme :



Exemple

Les instructions qui emploient deux opérandes registre utilisent des formats d'instruction qui ont un octet MODRM. Dans ce cas, le champ mod est 11b, indiquant que le champ R/M contient un second opérande registre.

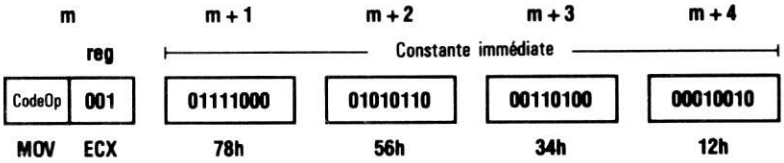
MOV DH,DL se coderait de la façon suivante :



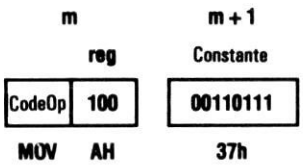
Exemple

L'instruction qui transfère une constante immédiate dans un registre est utilisée pour initialiser des compteurs de boucle qui sont alloués à des registres au cours de l'exécution de cette boucle. On utilise la forme de CodeOp sur un octet qui a un spécificateur d'opérande registre sur les 3 bits de poids faible. Dans ces exemples, on note les adresses mémoire par m, m + 1, ..., m + 4. Remarquons que le CodeOp se trouve dans l'octet d'adresse inférieure, et que la constante est stockée dans le reste des octets.

MOV ECX,12345678h serait codé de la façon suivante :



MOV AH,37h comme :



Les opérandes mémoire

Une instruction 80386 ne peut spécifier qu'un opérande mémoire de façon explicite. On doit préciser les deux parties, segment et déplacement, d'une adresse mémoire pour un opérande mémoire. La partie segment est déterminée en donnant le registre segment qui contient le sélecteur du segment de l'opérande. Le plus souvent, on peut utiliser le registre segment par défaut, si bien qu'aucun bit d'instruction supplémentaire n'est nécessaire. Si on emploie un registre segment différent, on a besoin d'un octet préfixe supplémentaire dans l'instruction.

La partie déplacement est déterminée par le mode d'adressage, qui indique le registre de base, le registre d'index, le facteur d'échelle pour l'index, et le déplacement constant. Le mode d'adressage existe toujours dans une instruction qui spécifie un opérande mémoire, et est contenu dans un ou plusieurs octets suivant les octets de CodeOp. Un champ MODRM d'un ou deux octets détermine le registre de base, le registre d'index et le facteur d'échelle, et donne la taille du déplacement. Si elle est indiquée, une constante de déplacement d'un, de deux, ou de quatre octets suit le champ MODRM. Les octets inférieurs des déplacements de deux ou de quatre octets sont stockés aux adresses inférieures, avec les octets supérieurs aux adresses supérieures.

Codage des modes d'adressage à un octet

On utilise un spécificateur spécialisé d'adresse courte dans les cas particuliers importants où il n'y a pas de registre d'index, c'est-à-dire quand le mode d'adressage précise au plus le registre de base et un déplacement. Cela économise un octet quand on emploie des modes d'adressage simples contenant juste un registre de base, un déplacement, ou les deux.

La Figure 2.14 montre le format des spécificateurs sur un octet. C'est l'octet MODRM dont on a parlé dans le paragraphe sur le codage de

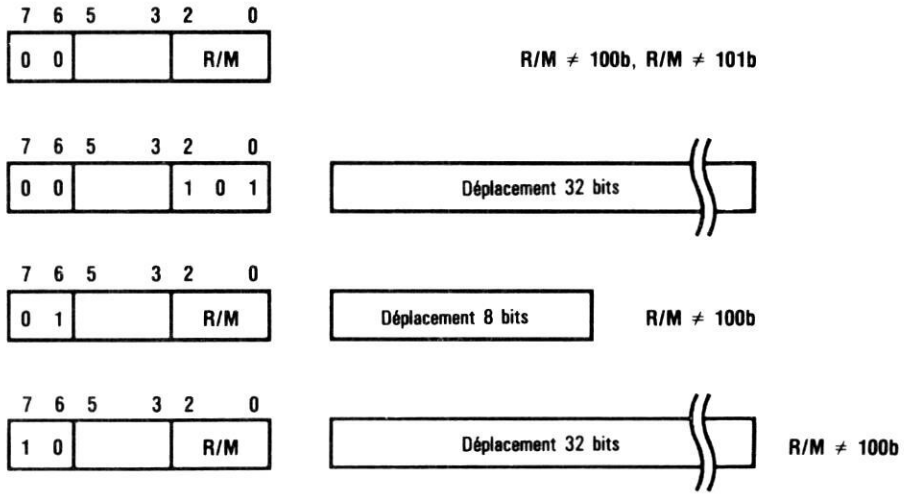


Figure 2.14 : *Format du mode d'adressage à un octet.*

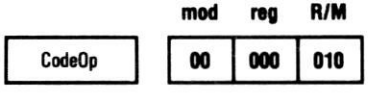
l'opérande registre. Les trois bits de poids faible (R/M) déterminent le registre de base. Un code de 100b dans ce champ est un code d'échappement spécial qui indique que la notation à deux octets est employée, avec le second octet du mode d'adressage sur deux octets dans l'octet suivant. Sinon, les 3 octets codent les registres de base 32 bits en utilisant le même code que pour déterminer les opérandes registre (voir le Tableau 2.3). Notons que seuls les registres 32 bits peuvent être employés dans les adresses.

Les 2 bits supérieurs (*mod*) déterminent la longueur du déplacement. Un code de 00b indique qu'il n'y a pas de déplacement. Un code de 01b indique la présence d'un déplacement de 8 bits. Le déplacement de 8 bits est étendu à 32 bits par duplication du bit de signe (bit 7) sur les 24 bits supérieurs (bits 8 à 31). Un code de 10b indique la présence d'un déplacement complet de 32 bits. Un code de 11b dans ce champ indique que le champ R/M détermine un opérande registre plutôt qu'un opérande mémoire. Ceci a été décrit dans le paragraphe précédent.

Exemple

Si mod = 00b, un mode d'adressage indirect par registre est sélectionné, avec le registre de base donné par le champ R/M.

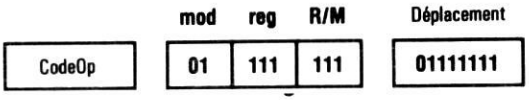
ADD EAX,DC:[EDX] serait codé comme :



Exemple

Si mod = 01b, un mode d'adressage indirect par registre plus déplacement est sélectionné, avec le registre de base donné par le champ R/M, plus un déplacement sur 8 bits. Le déplacement 8 bits est étendu (extension du bit de signe) à un déplacement 32 bits pour être ajouté à la valeur du registre de base.

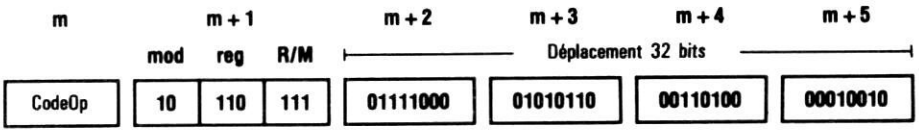
SUB EDI,DS:[EDI + 127] serait codé comme :



Exemple

Si mod = 10b, un mode d'adressage indirect par registre plus déplacement est sélectionné, avec le registre de base donné par le champ R/M, plus un déplacement sur 32 bits. Dans ce cas, les adresses mémoire sont notées m, m + 1, ..., m + 5 pour indiquer que les octets inférieur du déplacement 32 bits sont stockés aux adresses mémoire basses.

ADD ESI,DS:[EDI + 12345678h] serait codé comme :



Il y a un cas particulier qui demande plus d'explications. La combinaison de mod = 00b et R/M = 101b n'indique pas que EBP est le registre de base, et sans déplacement. Au contraire, il code le cas du seul déplace-

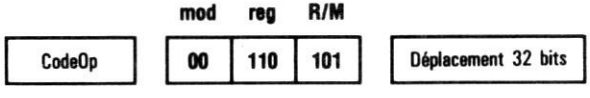
ment, c'est-à-dire sans registre de base et sans registre d'index, mais avec un déplacement de 32 bits.

Ces cas particuliers sont utiles puisqu'il existe dix codes de "registres de base" : un pour chacun des huit registres généraux, un pour coder l'échappement pour la forme à deux octets, et un pour indiquer l'absence de registre de base. Puisqu'il y a seulement 3 bits disponibles pour le code du registre de base, on a choisi pour le code de ESP l'échappement de la forme à deux octets (mod <> 11b et R/M = 100b). De plus, un peu d'espace a été "volé" au champ mod pour permettre au champ base de préciser "pas de registre base" (mod = 00b et R/M = 101b). Ce codage signifie que le mode d'adressage employant EBP comme base sans déplacement n'est pas permis. Au contraire, le mode d'adressage qui emploie EBP comme base avec un déplacement 8 bits de 0 est utilisé.

Exemple

On peut adresser une variable allouée statiquement en utilisant le mode d'adressage avec seulement un déplacement (mod = 00b et R/M = 101b).

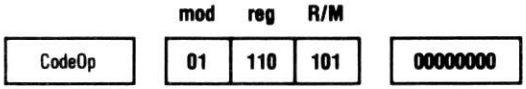
MOV ESI,VarScalaire serait codé comme :



Exemple

mod = 01b, R/M = 101b et un déplacement de 8 bits contenant 0 détermine une adresse mémoire avec EBP comme registre de base, plus un déplacement de 0.

MOV ESI,SS:[EBP] serait codé comme :



Si une instruction exige à la fois un champ de déplacement et une constante immédiate, le déplacement vient en premier, comme le montre l'exemple suivant.

Exemple

On peut initialiser une variable allouée statiquement par une instruction MOV qui a un mode d'adressage ne contenant qu'un déplacement et un champ 32 bits immédiat. Notons que le champ immédiat vient après (à une adresse mémoire supérieure) le champ déplacement.

MOV VarScalaire,32bitsImm serait codé comme :

Le Tableau 2.5 rappelle le codage des formes des modes d'adressage sur un octet. Les champs mod et R/M déterminent le registre de base et la taille du déplacement, et implicitement le registre segment par défaut. La colonne de droite donne la syntaxe en langage assembleur pour préciser le mode d'adressage. Puisque généralement le programmeur en assembleur ignore le nombre de bits du déplacement, Dépl32 (resp. Dépl8) représente un déplacement sur 32 bits (resp. 8 bits).

Mod	R/M	Mode d'adressage
00b	000b	DS:[EAX]
	001b	DS:[ECX]
	010b	DS:[EDX]
	011b	DS:[EBX]
	100b	échappement vers 2 octets
	101b	DS:Dépl32
	110b	DS:[ESI]
	111b	DS:[EDI]
01b	000b	DS:Dépl8[EAX]
	001b	DS:Dépl8[ECX]
	010b	DS:Dépl8[EDX]
	011b	DS:Dépl8[EBX]
	100b	échappement vers 2 octets
	101b	DS:Dépl8[EBP]
	110b	DS:Dépl8[ESI]
	111b	DS:Dépl8[EDI]
10b	000b	DS:Dépl32[EAX]
	001b	DS:Dépl32[ECX]
	010b	DS:Dépl32[EDX]
	011b	DS:Dépl32[EBX]
	100b	échappement vers 2 octets
	101b	DS:Dépl32[EBP]
	110b	DS:Dépl32[ESI]
	111b	DS:Dépl32[EDI]

Tableau 2.5 : *Codage des modes d'adressage à un octet.*

Codage des modes d'adressage à deux octets

Les formes d'adressage les plus générales sont codées sur deux octets, ce qui laisse de la place pour préciser le registre index ainsi que son facteur d'échelle. La forme à deux octets permet aussi l'emploi du registre ESP comme registre de base pour l'adressage par rapport au sommet de la pile programme. La Figure 2.15 illustre cette forme à deux octets. Comme on l'a vu pour la forme d'adressage à un octet, la forme à deux octets est indiquée par le code d'échappement 100b dans le champ R/M du premier octet. Le champ mod associé est exactement le même que dans le cas de la forme à un octet.

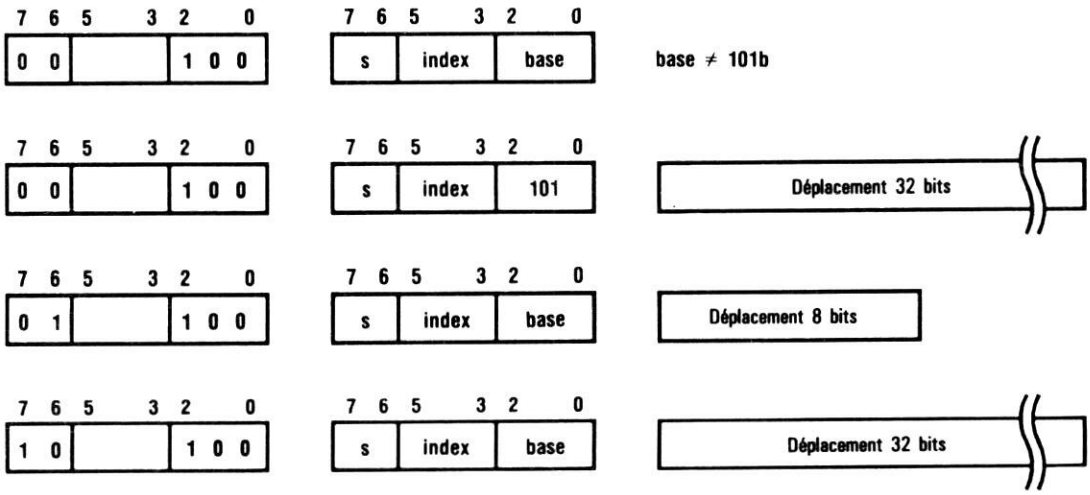


Figure 2.15 : *Format du mode d'adressage à deux octets.*

Le champ de base est situé dans les bits 0...2 du second octet, et il code un registre de base 32 bits, en utilisant le même codage que pour les opérandes registre du Tableau 2.3. Comme pour l'instruction sur un octet, la combinaison de mod = 00b et base = 101b n'indique pas que EBP est le registre de base, et cela sans déplacement. Au contraire, il code le cas index + déplacement, c'est-à-dire sans registre de base, mais avec un registre d'index et un déplacement complet sur 32 bits.

Le champ "index" est situé dans les bits 3...5, et code le registre d'index 32 bits (voir le Tableau 2.3). Un code de 100b dans le champ index ne donne pas ESP comme index ; au contraire, cela indique qu'on n'utilise pas de registre d'index. Le champ se situe dans les bits 6...7 et détermine le facteur d'échelle pour le registre d'index, un peu comme un compteur de décalage. Si on ne précise pas de registre d'index, le facteur d'échelle doit être 00b.

Le Tableau 2.6 rappelle le codage des formes d'adressage à deux octets. Les champs base et mod déterminent le registre de base, la taille du déplacement, et le registre segment par défaut. Les champs index et échelle modifient de façon indépendante l'adresse en ajoutant le registre d'index avec son facteur d'échelle. On donne le mode d'adressage en langage assembleur (voir le Tableau 2.5). Le modificateur de registre d'index est concaténé à la fraction segment/base/déplacement pour respecter la syntaxe de l'assembleur. Par exemple, une base de EDX, un déplacement de 8 bits, plus un index de ECX à l'échelle 4 donneraient une syntaxe d'assembleur DS:Dépl8[EDX][ECX*4]. D'un autre côté, la partie index pourrait être "ajoutée", avec la notation DS:Dépl8[EDX+ECX*4].

Mod	Base	Mode d'adressage
00b	000b	DS:[EAX]
	001b	DS:[ECX]
	010b	DS:[EDX]

Tableau 2.6 : *Codage du mode d'adressage à deux octets.*

Mod	Base	Mode d'adressage
	011b	DS:[EBX]
	100b	SS:[ESP]
	101b	DS:Dépl32
	110b	DS:[ESI]
	111b	DS:[EDI]
01b	000b	DS:Dépl8[EAX]
	001b	DS:Dépl8[ECX]
	010b	DS:Dépl8[EDX]
	011b	DS:Dépl8[EBX]
	100b	DS:Dépl8[ESP]
	101b	DS:Dépl8[EBP]
	110b	DS:Dépl8[ESI]
	111b	DS:Dépl8[EDI]
10b	000b	DS:Dépl32[EAX]
	001b	DS:Dépl32[ECX]
	010b	DS:Dépl32[EDX]
	011b	DS:Dépl32[EBX]
	100b	DS:Dépl32[ESP]
	101b	DS:Dépl32[EBP]
	110b	DS:Dépl32[ESI]
	111b	DS:Dépl32[EDI]

Index	Mode d'adressage
000b	[EAX*s]
001b	[ECX*s]
010b	[EDX*s]
011b	[EBX*s]
100b	pas d'index
101b	[EBP*s]
110b	[ESI*s]
111b	[EDI*s]

Tableau 2.6 : *Codage du mode d'adressage à deux octets (suite).*

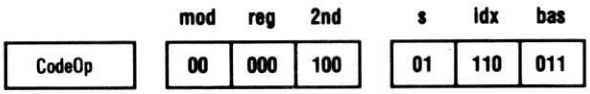
s	Facteur d'échelle index
00b	1
01b	2
10b	4
11b	8

Tableau 2.6 : *Codage du mode d'adressage à deux octets (suite).*

Exemple

Le mode d'adressage base + index est déterminé par mod = 00b, et les champs base et index donnent respectivement les registres base et index.

MOV EAX,DS:[EBX+ESI*2] serait codé comme :



Exemple

Un mode d'adressage complet base + index + déplacement est déterminé par mod = 01b pour un déplacement de 8 bits, et mod = 10b pour un déplacement de 32 bits.

MOV EAX,SS:[ESP+24][ESI*8] serait codé comme :

	mod	reg	2nd	s	idx	bas	Déplacement
CodeOp	01	000	100	11	110	100	00011000

MOV EAX,DS:[9999999+EDI+EAX*4] serait codé comme :

	mod	reg	2nd	s	idx	bas	
CodeOp	10	000	100	10	000	111	Déplacement 32 bits

Exemple

En précisant un index à l'échelle en association avec un mode d'adressage à déplacement, on peut traiter le cas important d'un index unique à l'échelle dans un tableau alloué statiquement. On utilise mod = 00b avec base = 101b pour indiquer un déplacement sur 32 bits sans registre de base. Les champs index et échelle sont employés pour préciser respectivement le registre d'index et le facteur d'échelle.

MOV EAX,TABLE[ESI*4] serait codé comme :

	mod	reg	2nd	s	idx	bas	
CodeOp	00	000	100	10	110	101	Déplacement 32 bits

Exemple

L'adressage relatif à la pile est traité avec la forme à deux octets en employant ESP comme registre de base avec un code d'index nul (100b), et un choix de déplacement nul (sommet de la pile) ou de 8 bits, ou de 32 bits (références à un élément à l'intérieur de la pile).

MOV EDI,0[ESP] serait codé comme :

	mod	reg	2nd	s	idx	bas
CodeOp	00	111	100	00	100	100

MOV EDI,24[ESP] serait codé comme :

	mod	reg	2nd	s	idx	bas	
CodeOp	01	111	100	00	100	100	00011000

Octet préfixe de recouvrement de segment

Un octet préfixe d'instruction peut préciser le registre segment qui doit être employé pour adresser l'opérande mémoire quand le registre segment par défaut n'est pas approprié. Comme son nom l'indique, c'est un champ de 8 bits qui est ajouté devant l'instruction. Puisqu'il y a six registres segments, il y a six codes pour les octets préfixes de recouvrement de segment (voir le Tableau 2.7).

Code	Registre
26h	ES:
2Eh	CS:
36h	SS:
3Eh	DS:
64h	FS:
65H	GS:

Tableau 2.7 : *Codage du préfixe de recouvrement de segment.*

Exemple

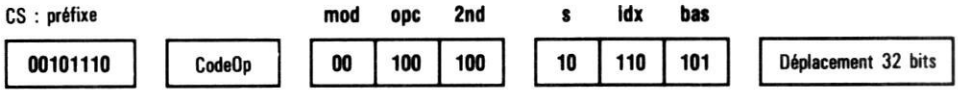
Pour adresser une mémoire via un pointeur, on charge d'abord le pointeur dans la paire registre segment/registre général. Puis un mode d'adressage employant le registre général comme base, combiné avec le préfixe de recouvrement de segment précisant le registre segment, donne accès à la donnée pointée par le pointeur.

MOV EDX,ES:[ESI] serait codé comme :

Exemple

Pour adresser des données stockées dans un segment code, on peut spécifier le préfixe de recouvrement CS. Par exemple, on peut stocker une table de déplacements associés à des labels dans le segment code et l'utiliser pour implémenter les fonctions CASE ou SWITCH. Un mode d'adressage index (à l'échelle) + déplacement, combiné avec un préfixe de changement CS, forme le mode d'adressage nécessaire à une instruction de branchement indirect (JMP) qui fait référence à cette table.

JMP CS:24[ESI*4] serait codé comme :



L'ESPACE E/S

Dans le chapitre précédent, nous avons étudié la structure de l'espace mémoire principal, et la façon de référencer dans une instruction des opérandes en mémoire principale. Un second espace mémoire, appelé espace E/S, est aussi disponible. Comme son nom l'indique, il est optimisé pour la gestion des ports de contrôle d'unités d'entrées/sorties (E/S) telles que claviers, unités de disque, affichages vidéo, imprimantes, etc. Un espace indépendant pour les ports E/S est intéressant parce que l'adressage et la protection des unités E/S sont tout à fait différents de ceux des codes programme et des données. En effet, une unité E/S a seulement quelques ports de contrôle, demandant un petit nombre d'octets adressables. De plus, il y a peu d'unités dans un système. Les ports E/S ne peuvent pas être réalloués dans l'espace E/S, et ils doivent être protégés individuellement pour contrôler l'accès aux unités distinctes. D'un autre côté, le code programme et les données demandent quelques milliers, voire quelques millions d'octets adressables, et ont besoin d'un mécanisme de protection différent.

Le 80386 permet de disposer d'un espace E/S indépendant. Ce dernier n'est pas segmenté. C'est un espace adressable à dimension unique, c'est-à-dire 64 K. Comme la mémoire principale, on peut adresser l'espace E/S octet par octet. Un port E/S peut avoir un, deux, ou quatre octets de longueur. Comme dans la mémoire principale, c'est la méthode "little endian" qui est employée pour adresser les ports à deux ou à quatre octets. Les octets d'adresses basses contiennent les bits les moins significatifs, et les octets d'adresses hautes, les bits les plus significatifs.

Le 80386 dispose d'instructions spéciales, décrites au Chapitre 3, pour transférer des données entre l'espace E/S et les registres ou la mémoire principale. Les adresses E/S sont spécifiées comme suit :

- Si l'adresse E/S est plus petite que 256, on peut la placer dans un champ immédiat 8 bits de l'instruction.

- Des adresses E/S plus grandes, ou des adresses qui sont calculées au cours de l'exécution du programme, sont prises à partir du registre DX.

L'espace E/S 64 K est protégé par un mécanisme, décrit au Chapitre 5, qui permet des protections distinctes pour chaque octet de l'espace E/S. Ce mécanisme de protection est totalement indépendant et complètement différent de celui de la mémoire principale.

LES REGISTRES A VIRGULE FLOTTANTE

Comme on l'a vu dans le Chapitre 1, le coprocesseur 80387 peut être associé au 80386 pour effectuer des opérations complexes sur des données dans un des formats à virgule flottante. Même si vous n'avez pas de 80387, vous devez utiliser une émulation logicielle afin de travailler avec des données à virgule flottante. Quel que soit le cas (avec le circuit ou avec l'émulateur), vous devez être familiarisé avec l'architecture du 80387.

Quand il est employé avec le 80386, le 80387 ajoute les trois ensembles de registres de la Figure 2.16 pour traiter les calculs en virgule flottante :

1 Une pile de huit accumulateurs de 80 bits pour prendre en compte jusqu'à huit opérandes à virgule flottante.

2 Trois registres 16 bits d'état et de contrôle : un pour le mot d'état, un pour le mot de contrôle, et un pour le mot d'étiquette.

3 Quatre registres 32 bits pointeurs d'erreurs (FIP, FCS, FOO et FOS) pour identifier l'instruction et l'opérande mémoire causant l'exception.

Pile accumulateur à virgule flottante

Les instructions à virgule flottante traitent les registres 80 bits comme une pile d'accumulateurs plutôt que comme un simple tableau de huit registres. Le sommet de cette pile est appelé ST. Les accumulateurs situés sous ST sont adressés relativement à ST avec des noms de la forme ST(i), où i varie de 1 à 7. ST(0) est un substitut de ST pour nommer le sommet de la pile accumulateur.

ST se comporte comme un accumulateur parce qu'il est implicitement employé comme un opérande pour toutes les opérations à virgule flottante. Cela pour avoir 64 CodeOp en virgule flottante qui représentent seulement huit codes pour l'espace CodeOp du 80386. C'est un champ registre de 3 bits consacré aux CodeOp supplémentaires qui permet cette astuce. Afin de contourner la plupart des limites du modèle à accumulateur, on a créé une pile de huit accumulateurs à accès direct au niveau opérande. Bien que les opérations à virgule flottante doivent employer ST comme opérande, elle peuvent nommer tout accumulateur à virgule flottante ou tout opérande mémoire comme second opérande.

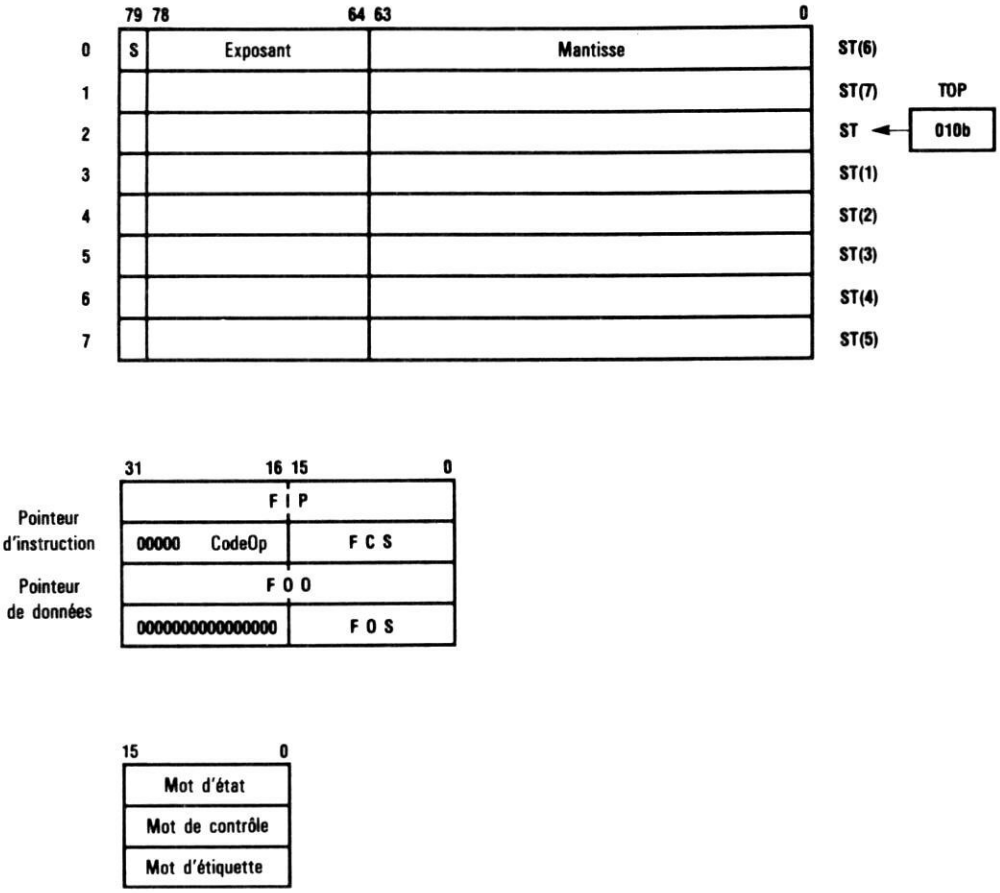


Figure 2.16 : *Registres du 80387.*

La pile accumulateur est réellement un tableau de huit registres physiques, avec un champ indépendant de 3 bits pour identifier le sommet courant de la pile. Un emploi judicieux des termes registre et accumulateur

permet de distinguer les deux interprétations des opérandes à virgule flottante. Le terme *registre physique* se réfère à une organisation physique de la pile. Le terme *accumulateur* décrit une localisation d'un opérande à virgule flottante. Cependant, nous avons à la fois une pile accumulateur et un tableau de registres physiques.

Le champ de 3 bits nommé TOP dans le mot d'état contient le numéro absolu du registre du sommet courant de la pile accumulateur, ST ou ST(0). Si on empile, on décrémente TOP et on stocke la nouvelle valeur dans le nouvel ST. Après cela, le précédent ST devient ST(1), et tous les accumulateurs empilés ont leurs noms relatifs à ST incrémentés de 1. Si on dépile pour lire une valeur à partir du ST courant, TOP est incrémenté. Puis, l'accumulateur ST(1) devient le nouveau ST, et tous les accumulateurs empilés ont leurs noms relatifs à ST décrémentés de 1.

L'incrémentation et la décrémentation du champ TOP à 3 bits ignorent le recouvrement. Si TOP = 000b, un empilage décrémentera TOP à 111b et stockera la nouvelle valeur dans le registre 7. Si TOP = 111b, un dépilage lira la valeur du registre 7 puis incrémentera TOP à 000b.

La Figure 2.16 illustre la structure de la pile accumulateur à virgule flottante. Si le champ TOP contient 010b, les noms des accumulateurs relatifs à ST (à droite) correspondent aux numéros des registres physiques (à gauche). Au sommet courant de la pile accumulateur, ST, correspond un registre physique 2. Chaque accumulateur 80 bits de la pile peut stocker un nombre réel sous le format réel temporaire (voir le Chapitre 1). Le bit de poids fort est le bit de signe. Les bits 64..78 donnent le champ exposant sur 15 bits. Les bits 0..63 donnent la mantisse sur 64 bits.

Les instructions à virgule flottante, décrites au Chapitre 3, sont étudiées pour optimiser un modèle de pile accumulateur. Les instructions de chargement lisent un opérande de la mémoire et l'empilent dans la pile accumulateur. Les instructions de rangement prennent la valeur du sommet courant de la pile et la rangent en mémoire, et peuvent optionnellement dépiler la pile accumulateur si on n'a pas besoin tout de suite de cette valeur pour un autre calcul. Les opérations arithmétiques telles que addition ou multiplication prennent un opérande du registre ST et l'autre opérande d'un autre registre ou mémoire, et rangent le résultat dans le registre ST.

Une forme particulière avec dépilage est disponible pour travailler sur deux opérandes sommet de pile, ST et ST(1). Cette instruction dépile une fois, et place le résultat dans le nouvel ST. Cette séquence est équivalente à l'opération sur une pile classique, qui dépile deux opérandes, ST et ST(1), traite l'opération et empile le résultat.

L'exemple suivant illustre les opérations sur la pile accumulateur en montrant différents états de la pile à chaque séquence d'instruction. On trouvera un autre exemple au Chapitre 4.

Supposons que vous ayez besoin de calculer le produit scalaire de deux vecteurs, donc de former une somme de produits. Vous pouvez accumuler la somme dans un des registres du 80387, et former chaque produit dans un autre registre. Si ST contient la somme partielle des produits, vous pouvez traiter une étape du produit scalaire en chargeant le prochain élément d'un vecteur dans la pile accumulateur, en multipliant l'autre vecteur par le prochain élément, puis en additionnant le résultat à la somme partielle.

La Figure 2.17 montre plusieurs états de la pile accumulateur du 80387 tout au long du traitement du produit scalaire. Le premier état montre la pile accumulateur avec la somme partielle 101,237 stockée dans ST, et les autres registres vides (illustrés par les hachures). Les numéros des registres physiques sont à gauche, et les noms des accumulateurs sont à droite. ST est le registre physique 5 au début de l'exemple, si bien que le champ TOP contient 101b.

L'élément suivant du premier vecteur (dont la valeur est 1,21) est empilé avec une instruction de chargement dans la pile accumulateur, laissant deux éléments dans la pile, comme on peut le voir sur le dessin. La nouvelle valeur est chargée dans le registre physique 4, qui devient le registre sommet de pile, ST. On pourrait maintenant adresser la somme partielle par ST(1), somme qui, bien sûr, se trouve encore dans le registre physique 5. Le troisième dessin montre la pile après la multiplication de ST par l'élément suivant du second vecteur (dont la valeur est 5,0). Cette opération change la valeur de ST en 6,05, mais n'empile ni ne dépile. Le dernier schéma montre la pile après une addition et une instruction de dépilage (FADDP), qui ajoute le nouveau produit dans le registre contenant la

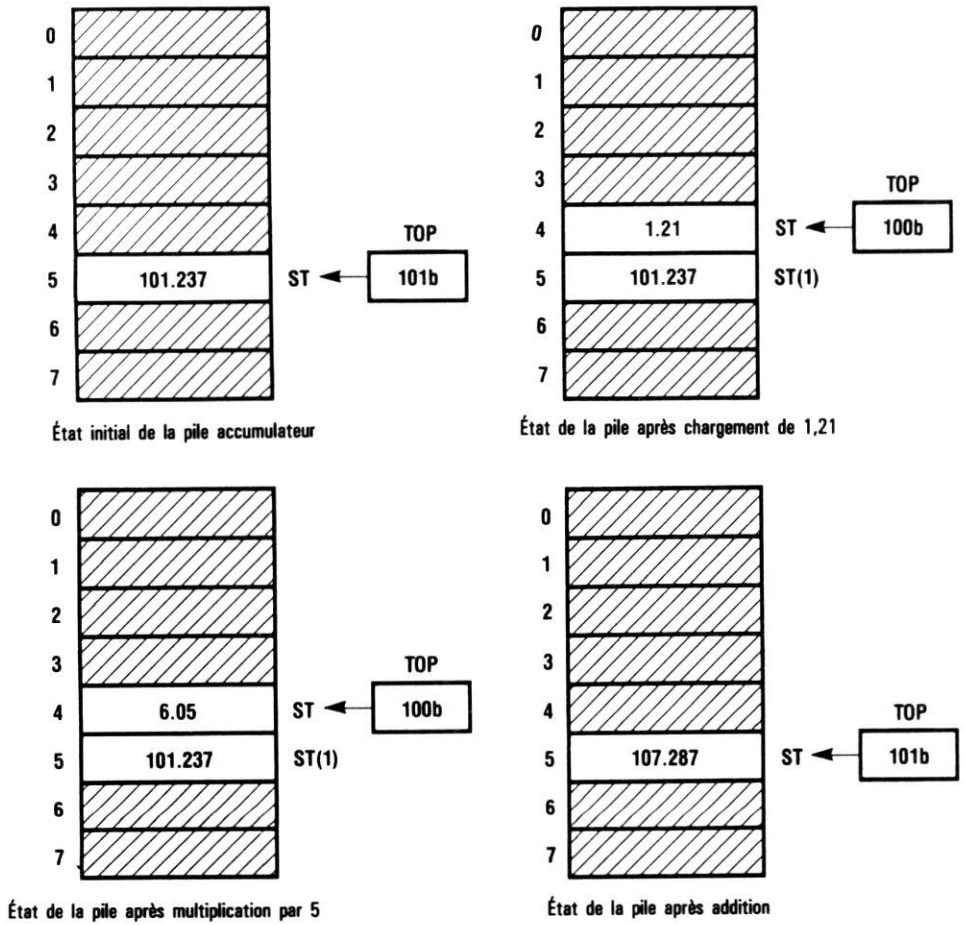


Figure 2.17 : *Opérations sur la pile accumulateur du 80387.*

somme partielle. Cette instruction ajoute ST (le nouveau produit) à ST(1) (l'ancienne somme partielle), dépile un élément, et remplace ST par le résultat (107,287), la nouvelle somme partielle. Cela laisse la pile dans un

état semblable à l'état initial, avec la somme partielle dans ST, qui est prête pour une nouvelle itération.

Registres de contrôle et d'état 16 bits

Trois registres 16 bits (mot d'étiquette, mot de contrôle et mot d'état) contrôlent le traitement des instructions à virgule flottante et renseignent sur l'état du coprocesseur.

Le registre d'étiquette

Le registre d'étiquette 16 bits (voir la Figure 2.18) contient huit champs de 2 bits, un pour chacun des huit registres physiques à virgule flottante. Ces champs indiquent si le registre physique correspondant contient un nombre à virgule flottante valide, nul ou particulier, ou même s'il est vide. Les champs du registre d'étiquette correspondent aux registres physiques plutôt qu'au sommet de pile pour éviter l'emploi d'une rotation chaque fois que la pile accumulateur est dépilée ou empilée !

On emploie ces champs pour détecter des dépassements de la pile accumulateur. Une sorte de dépassement apparaît si, quand on empile, TOP pointe sur un registre non vide. Un autre dépassement survient si on tente de lire ou de dépiler un registre vide. Ces dépassements entraînent une opération du type non valide et génèrent une exception.

Le registre de contrôle

Le registre de contrôle peut être modifié par le programme afin de contrôler les opérations du 80387. On dispose de trois champs (voir la Figure 2.19).

Nous allons décrire brièvement chaque champ du registre de contrôle :

15 14	13 12	11 10	9 8	7 6	5 4	3 2	1 0	
Etq 7	Etq 6	Etq 5	Etq 4	Etq 3	Etq 2	Etq 1	Etq 0	TW (Étiquette)

Codage du champ Etq	
Etq i	Sens
00b	Valide
01b	Zéro
10b	Spécial (Infini, Non, Dénormalisé)
11b	Vide

Figure 2.18 : *Registre d'étiquette.*

- Les bits 0..5 contiennent les masques d'exception pour les conditions dont la liste est fournie dans le Tableau 2.8. Si une exception est détectée par le 80387, le bit de masque associé à cette exception est testé pour déterminer si l'exception doit être passée à une routine de traitement des erreurs (masque = 0), ou traitée comme erreur dans le 80387 par défaut (masque = 1). Une exception est dite *masquée* si son bit de masque est à 1.

- Les bits 8..9 contiennent le champ de *contrôle de la précision*, PC (*Precision Control*). Les résultats d'une addition, soustraction, multiplication, division, ou racine carrée à virgule flottante, sont arrondis dans une des trois précisions citées à la Figure 2.19 avant d'être stockés dans leur destination. Toute autre opération emploie la précision des réels temporaires, ou une précision déterminée par l'instruction.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	0	0	0	RC		RC		0	1	PM	UM	OM	ZM	DM	IM	CW (Contrôle)

Contrôle de la précision	
PC	Précision de la mantisse
00b	24 bits (Réel court)
01b	(Réservé)
10b	53 bits (Réel long)
11b	64 bits (Réel temporaire)

Contrôle de l'arrondi	
RC	Mode d'arrondi
00b	Au plus près
01b	Vers – l'infini
10b	Vers + l'infini
11b	Troncature

Figure 2.19 : *Registre de contrôle.*

- Les bits 10..11 contiennent le champ de *contrôle d'arrondi*, RC (*Rounding Control*). Les quatre modes d'arrondi sont indiqués à la Figure 2.19 et sont disponibles pour arrondir des résultats d'opérations à virgule flottante.

Les bits restant dans le mot de contrôle sont câblés aux valeurs de la Figure 2.19. Quand on charge une valeur dans ces registres, ces bits doivent avoir les valeurs précédentes, ou alors ils seront rechargés à partir d'une image sauvegardée du mot de contrôle. Cela afin d'assurer la compatibilité

Etat	Masque	Condition
IE	IM	Opération non valide
DE	DM	Dénormalisé
ZE	ZM	Division par zéro
OE	OM	Débordement (overflow)
UE	UM	Débordement (underflow)
PE	PM	Précision

Tableau 2.8 : *Bits de masque et d'état des exceptions.*

avec d'autres coprocesseurs numériques pouvant définir des bits de contrôle supplémentaires.

Arrondi

Chaque fois que cela est possible, le 80387 donne un résultat correct. Il est possible cependant qu'au cours des calculs intermédiaires ou du stockage dans un type de données plus petit, une perte de précision apparaisse. Il est évident qu'on perd en précision quand on passe d'un format de stockage interne sous forme réel temporaire à un format entier court. Quand de telles situations se présentent, le 80387 arrondit le résultat afin qu'il puisse tenir dans un opérande plus petit.

Il y a quatre modes d'arrondi sur le 80387, déterminés par le champ RC du registre de contrôle.

- Arrondi au plus près.
- Arrondi vers - l'infini.
- Arrondi vers + l'infini.

- Troncature.

La troncature est utile quand on travaille en arithmétique entière, puisqu'elle permet à une instruction de division entière d'arrondir le quotient.

L'arrondi au plus près s'emploie dans la plupart des applications. C'est le mode d'arrondi par défaut, puisque le 80387 initialise le champ RC à 00b lors d'une réinitialisation. Si vous connaissez mal les méthodes numériques, ce mode devrait suffire. Il permet d'obtenir les résultats les plus précis et n'introduit jamais de biais dans les calculs.

L'arrondi vers +/- l'infini gère l'arithmétique sur intervalles. C'est utile quand vous suspectez une erreur fantaisiste et que vous voulez quantifier cette erreur. Une utilisation judicieuse du mode d'arrondi permet d'établir correctement l'intervalle d'erreur d'un calcul.

Les infinis

Les premiers coprocesseurs arithmétiques de la famille 86 (8087 et 80287) pouvaient traiter les infinis de deux modes, affine ou projeté, et employaient le bit 12 du mot de contrôle pour préciser le mode. Le 80387 traite seulement le mode affine pour suivre la norme IEEE dont on a parlé au Chapitre 1.

Le 8087 et le 80287 mettent 1 dans le bit 12 du registre de contrôle pour déterminer le mode affine, et 0 pour le mode projeté. Ces anciens coprocesseurs initialisent ce bit à 0 (mode projeté donc) à la réinitialisation, ou après une instruction FINIT. Ce bit est aussi initialisé à 0 sur le 80387, donc le mot de contrôle est initialisé à la même valeur sur tous les coprocesseurs. L'Annexe B montre de façon plus détaillée ces différences (et d'autres encore) entre le 80387 et les coprocesseurs 8087/80287.

Dans les calculs, le mode affine fait la distinction entre + l'infini et - l'infini. Le signe indique vers quelle direction de l'infini on tend, ce qui peut être utile dans certains cas. Cependant, un algorithme peut conduire à

un résultat correct à partir de plus ou moins zéro. Si vous traitez 1/0 ou 1/-0, vous obtenez respectivement plus ou moins l'infini. Cela donne une indication supplémentaire, mais peut s'avérer inutile, et même incorrect dans certains cas. Ainsi, il est important d'interpréter le signe de l'infini avec prudence.

Le registre d'état

Le 80387 positionne des bits dans le registre d'état afin qu'ils puissent être testés par le programme pour contrôler certaines conditions. La Figure 2.20 détaille le registre d'état.

Nous allons décrire brièvement chaque champ du registre d'état.

- Les bits 0..5 sont levés si une exception est détectée lors de l'exécution d'une instruction à virgule flottante. Ces bits sont peu agréables à utiliser. Ils sont positionnés par le 80387 chaque fois qu'une condition d'exception est détectée et doivent être explicitement remis à zéro par le programme. Le Tableau 2.8 donne une liste des correspondances entre ces bits et les exceptions.

- Le bit 6 (SF) est l'indicateur d'*erreur de pile* (*Stack Fault*). Il est levé si une exception d'opération du type non valide est causée par un dépassement de la pile accumulateur. Sinon, il est à zéro.

- Les bits 8 (C0), 9 (C1), 10 (C2), 14 (C3) contiennent les codes de condition en virgule flottante. On dispose d'un jeu d'instructions de comparaison et celui-ci permet de positionner ces bits sur la validité de la comparaison. Ces bits de code de condition sont positionnés de façon à être liés aux 8 bits de poids faible du registre EFLAGS du 80386. Ainsi, le même jeu d'instructions de sauts conditionnels peut être employé aussi bien pour les conditions en virgule flottante que pour les conditions détectées par le 80386. Dans le Chapitre 4, on trouve un exemple qui permet de tester les codes de condition en virgule flottante.

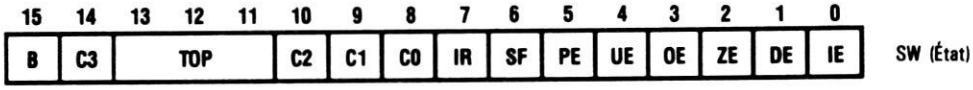


Figure 2.20 : *Registre d'état.*

- les bits 11..13 contiennent le champ TOP, qui indique à quel registre physique correspond le sommet de la pile accumulateur.

- Le bit 7 (IR) et le bit (15) sont égaux à 1 si une quelconque exception non masquée est indiquée par les 6 bits d'exception du registre d'état. Sinon, IR et B sont à 0.

Les registres pointeurs d'erreurs

Le dernier jeu de registres du 80387 (voir la Figure 2.16) est constitué par l'ensemble des quatre registres 32 bits qui contiennent les pointeurs sur la dernière instruction 80387 exécutée, et ses données. Les deux premiers registres, FIP et FCS, contiennent un pointeur sur l'instruction 80387 qui vient d'être exécutée relativement aux deux octets de cette instruction (en ignorant les préfixes). FCS contient le sélecteur et le CodeOp, et FIP le déplacement. La seconde paire de registres, FOO et FOS, pointe sur l'opérande mémoire de cette dernière instruction. FOS contient le sélecteur et FOO le déplacement. Si la dernière instruction du coprocesseur ne

contient pas d'opérande mémoire, les valeurs de FOO et FOS sont indéterminées. Les instructions FLDENV, FSTENV, FNSTENV, FRSTOR, FSAVE et FNSAVE (voir le Chapitre 3) permettent de charger et de ranger ces registres pointeurs d'erreurs.

Le champ CodeOp est formé par la combinaison des 3 bits de poids faible du premier octet d'instruction (après les préfixes) et du second octet d'instruction (voir la Figure 2.21). Les deux octets sélectionnés suivent d'éventuels préfixes, et contiennent donc le champ CodeOp et le premier octet du champ MODRM. A la Figure 2.21, on trouve en haut les deux octets d'instruction comme un nombre 16 bits, avec les bits de poids faible à l'adresse inférieure. A droite, le premier octet d'instruction est à l'adresse m et contient les bits I0 à I7. Le second octet d'instruction à gauche est à l'adresse m+1 et contient les bits I8 à I15. Les trois bits de poids faible du premier octet d'instruction (I0, I1, et I2) sont placés dans les bits 8 à 10 du champ CodeOp. Les 8 bits du second octet d'instruction (I8 à I15) sont placés dans les bits 0 à 7 du champ CodeOp. Puisque les 5 bits de poids fort du premier octet d'instruction (I3, I4, I5, I6, et I7) sont les mêmes pour toutes les instructions à virgule flottante, ils ne sont pas stockés dans le champ CodeOp. Le format du CodeOp sépare les octets CodeOp afin que les bits écartés soient en position haute, plutôt qu'au centre du champ CodeOp (notons que la version précédente B1 du 386 ne stockait pas l'information CodeOp).

Les pointeurs d'erreurs servent à analyser et à rendre compte des exceptions qui surviennent durant des instructions à virgule flottante. Ces pointeurs sont nécessaires puisque le 80387 et le 80386 peuvent travailler en parallèle. Dès qu'une instruction à virgule flottante est envoyée au 80387, ce dernier peut poursuivre son travail pendant que le 80386 continue avec une autre instruction. Si une erreur apparaît pour une instruction 80387, les registres CS et EIP du 80386 peuvent ne pas être liés à l'instruction 80387 qui a causé l'exception. Les registres pointeurs d'erreurs permettent un traitement parallèle par le 80386 et le 80387, tout en apportant encore l'information nécessaire au diagnostic et à la gestion des exceptions en virgule flottante.

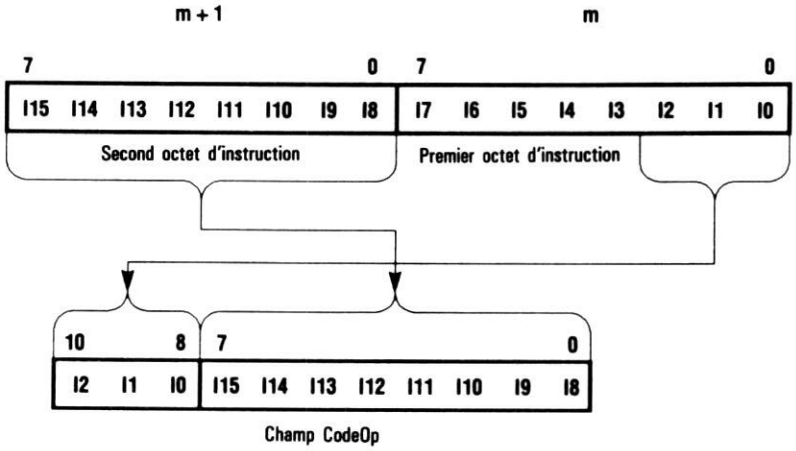


Figure 2.21 : *Format du champ CodeOp.*

Le format des opérandes de FLDENV, FSTENV, FNSTENV

Les instructions FLDENV, FSTENV, FNSTENV, décrites au Chapitre 3, chargent et rangent le mot de contrôle, le mot d'état, le mot étiquette, et les registres pointeurs d'erreurs à partir du bloc de mémoire de 28 octets de la Figure 2.22. Les octets qui ont la valeur 0 sur la figure sont *réservés*. On doit charger à 0 ces bits réservés et les ignorer ensuite.

Les instructions FRSTOR, FSAVE et FNSAVE chargent et rangent tous les registres 80387 à partir d'un bloc mémoire de 108 octets. Les 28 octets inférieurs de ce bloc suivent le format de la Figure 2.22. ST(0)

31 … 16	15 … 0	
0000000000000000	FOS	24
FOO		20
00000 CodeOp	FCS	16
FIP		12
0000000000000000	Mot d'étiquette	8
0000000000000000	Mot d'état	4
0000000000000000	Mot de contrôle	0

Figure 2.22 : *Le format des opérandes de FLDENV, FSTENV, FNSTENV*

est chargé ou rangé à partir du déplacement 28, ST(1) à partir du déplacement 38, et ainsi de suite, jusqu'à ST(7) à partir du déplacement 98. Les places mémoire qui correspondent aux accumulateurs considérés comme vides sont indéfinies après l'exécution d'un FSAVE ou FNSAVE, et sont ignorées par l'instruction FRSTOR.

3

Le jeu d'instructions

TABLE DES MATIERES

Introduction 122

Index alphabétique des instructions 123
Format de description d'une instruction 132
Format d'une instruction 132
Description d'une instruction 135
Fonctionnement d'une instruction 135
Exemples d'une instruction 136
Exceptions associées à une instruction 137
Notes sur une instruction 139
Syntaxe de description d'une instruction 139
Fonctions 146

Entier 149

AAA - Ajustement ASCII après l'addition 155
AAD - Ajustement ASCII avant la division 157
AAM - Ajustement ASCII après la multiplication 159
AAS - Ajustement ASCII après la soustraction 161
ADC - Addition avec retenue 163
ADD - Addition 165
AND - "ET" logique 167
BOUND - Vérification des limites d'un tableau 169
BSF - Recherche de bit en avant 171
BSR - Recherche de bit en arrière 173
BT - Test de bit 175

BTC - Test et complémentation de bit 178
BTR - Test de bit et remise à zéro 181
BTS - Test et positionnement de bit 184
CALL - Appel à un sous-programme 187
CBW/CWDE - Conversion d'octet en mot 189
CLC - Mise à zéro de l'indicateur de retenue 191
CLD - Mise à zéro de l'indicateur de direction 192
CLI - Mise à zéro de l'indicateur d'interruption 193
CMC - Complémentation du bit de retenue 195
CMP - Comparaison 196
CMPS/CMPSB/CMPSW/CMPSD - Comparaison de chaînes 198
CWD/CDQ - Conversion de mot en dmot 201
DAA - Ajustement décimal après addition 203
DAS - Ajustement décimal après soustraction 205
DEC - Décrémentation 207
DIV - Division non signée 209
ENTER - Création du cadre de pile 212
IDIV - Division signée 214
IMUL - Multiplication signée 218
IN - Entrée E/S 221
INC - Incrémentation 223
INS/INSB/INSW/INSD - Entrée E/S d'une chaîne 225
Jcc - Instructions de saut conditionnel 228
JMP - Saut 231
LAHF - Chargement des indicateurs dans AH 234
LEA - Chargement de l'adresse effective 236
LEAVE - Abandon de la procédure 238
LOCK - Verrouillage du bus 240

LODS/LODSB/LODSW/LODSD - Chargement d'une chaîne 242
LOOP - Boucle avec compteur ECX 245
LOOPNZ/LOOPNE - Boucle avec ECX et test si non nul 247
LOOPZ/LOOPE - Boucle avec ECX et test si non nul 249
MOV - Transfert d'opérandes 251
MOVS/MOVSB/MOVSW/MOVSD - Transfert de chaînes 253
MOVSX - Transfert avec extension de signe 256
MOVZX - Transfert avec extension de zéro 258
MUL - Multiplication non signée 260
NEG - Complémentation à deux 262
NOP - Pas d'opération 264
NOT - "NON" logique 266
OR - "OU inclusif" logique 268
OUT - Sortie E/S 270
OUTS/OUTSB/OUTSW/OUTSD - Sortie E/S d'une chaîne 272
POP - Dépilage 275
POPA/POPAD - Dépilage de tous les registres 277
POPF/POPFD - Dépilage des indicateurs 279
PUSH - Empilage 281
PUSHA/PUSHAD - Empilage de tous les registres 283
PUSHF/PUSHFD - Empilage des indicateurs 285
RCL - Rotation à gauche avec retenue 287
RCR - Rotation à droite avec retenue 290
REP - Répétition 293
REPE/REPZ - Répétition tant qu'égal 295
REPNE/REPNZ - Répétition tant que pas égal 297
RET - Retour d'une procédure 299
ROL - Rotation à gauche 301

ROR - Rotation à droite 304
SAHF - Rangement de AH dans les indicateurs 307
SAL/SHL - Décalage arithmétique à gauche 309
SAR - Décalage arithmétique à droite 312
SBB - Soustraction avec retenue 315
SCAS/SCASB/SCASW/SCASD - Recherche de chaîne 318
SETcc - Assignation conditionnelle d'octet 321
SHLD - Décalage à gauche double 324
SHR - Décalage logique à droite 327
SHRD - Décalage à droite double 329
STC - Positionnement de l'indicateur de retenue 332
STD - Positionnement de l'indicateur de direction 333
STI - Positionnement de l'indicateur d'interruption 334
STOS/STOSB/STOSW/STOSD - Stockage de chaîne 336
SUB - Soustraction 338
TEST - Comparaison logique 340
XCHG - Echange 342
XLAT/XLATB - Translation avec table 344
XOR - "OU exclusif" logique 346

Segments multiples 348

CALL - Appel de procédure 351
INT - Appel à une procédure d'interruption 354
INTO - Appel à procédure d'interruption sur débordement 357
IRET - Retour d'une procédure d'interruption 359
JMP - Branchement 361
Lsr - Chargement d'un pointeur 363

MOV - Transfert de registre segment 365
POP - Dépilage dans le registre segment 367
PUSH - Empilage 369
RET - Retour d'une procédure 371

Système d'Exploitation 373

ARPL - Ajustement du niveau de privilège requis 377
CLTS - Effacement de l'indicateur de commutation de tâches 379
HLT - Halte 381
LAR - Chargement des droits d'accès 382
LGDT - Chargement de la table des descripteurs globaux 386
LIDT - Chargement de la table des descripteurs d'interruptions 388
LLDT - Chargement de la table des descripteurs locaux 390
LMSW - Chargement du mot d'état machine 392
LSL - Chargement de la limite de segment 394
LTR - Chargement du registre de tâche 398
MOV - Transfert de registre spécial 400
SGDT - Stockage de la table des descripteurs globaux 402
SIDT - Stockage de la table des descripteurs d'interruptions 404
SLDT - Stockage de la table des descripteurs locaux 406
SMSW - Stockage du mot d'état machine 408
STR - Stockage du registre de tâche 410
VERR - Vérification du segment en lecture 412
VERW - Vérification du segment en écriture 415

Virgule flottante 418

F2XM1 - 2^x - 1 425
FABS - Valeur absolue 427
FADD/FIADD/FADDP - Addition 428
FBLD - Chargement de BCD 431
FBSTP - Stockage de BCD et dépilage 433
FCHS - Changement de signe 435
FCLEX/FNCLEX - Effacement des exceptions 436
FCOM/FCOMP/FCOMPP - Comparaison 438
FCOS - Cosinus 441
FDECSTP - Décrémentation du pointeur de pile 443
FDIV/FIDIV/FDIVP - Division 445
FDIVR/FIDIVR/FDIVPR - Division arrière 448
FFREE - Libération de registre 451
FICOM/FICOMP - Comparaison d'entiers 453
FILD - Chargement d'entier 456
FINCSTP - Incrémentation du pointeur de pile 458
FINIT/FNINIT - Initialisation du processeur 460
FIST - Stockage d'entier 462
FISTP - Stockage d'entier et dépilage 464
FLD - Chargement de réel 466
FLDcon - Chargement de constante 468
FLDCW - Chargement du mot de contrôle 470
FLDENV - Chargement de l'environnement 471
FMUL/FIMUL/FMULP - Multiplication 473
FNOP - Pas d'opération 476
FPATAN - Arctangente partiel 477

FPREM - Reste partiel 479
FPREM1 - Reste partiel - IEEE 482
FPTAN - Tangente partielle 485
FRNDINT - Arrondi à l'entier 487
FRSTOR - Récupération de l'état 489
FSAVE/FNSAVE - Sauvegarde de l'état 491
FSCALE - Echelle de puissance de deux 493
FSIN - Sinus 495
FSINCOS - Sinus et cosinus 497
FSQRT - Racine carrée 499
FST - Stockage de réel 501
FSTCW/FNSTCW - Stockage du mot de contrôle 503
FSTENV/FNSTENV - Stockage de l'environnement 504
FSTP - Stockage de réel et dépilage 506
FSTSW/FNSTSW - Stockage du mot d'état 508
FSTSW AX/FNSTSW AX - Stockage du mot d'état dans AX 509
FSUB/FISUB/FSUBP - Soustraction 511
FSUBR/FISUBR/FSUBRP - Soustraction arrière 514
FTST - Test 517
FUCOM/FUCOMP/FUCOMPP - Comparaison non ordonnée 519
FXAM - Examen 522
FXCH - Echange de registres 524
FXTRACT - Extraction de l'exposant et de la mantisse 526
FYL2X - $y * \log_2 x$ 528
FYL2XP1 - $y * \log_2(x + 1)$ 530
WAIT - Attente pour le coprocesseur 532

INTRODUCTION

Félicitations ! Vous avez ingurgité toute la partie préliminaire des Chapitres 1 et 2, et vous voici prêt à comprendre exactement ce que le processeur 80386 peut vous apporter. Vous vous êtes familiarisé avec les types de données, l'état interne du processeur, et la méthode d'adressage mémoire. Maintenant, vous êtes prêt à vous plonger dans une description détaillée de chaque instruction du 80386.

Vous verrez dans ce chapitre que vous devez fournir un travail important. Le 80386 et le 80387 ont des jeux d'instructions très riches. Nous avons fait le compte : le 80386 possède 152 instructions et le 80387 en possède 74. Le jeu d'instructions du 80386 a suivi l'évolution de la micro-informatique : on est passé du 8086 au 80386 qui est un processeur 32 bits multitâche, multitraitement, et à mémoire virtuelle. On retrouve donc les instructions familières du 8086, ainsi que des instructions qui ont été ajoutées à chaque génération pour permettre à la machine d'accéder à quelques-uns des concepts d'architecture d'ordinateurs les plus avancés, comme la mémoire virtuelle.

Cette évolution nous a conduits à réorganiser la présentation du jeu d'instructions. Contrairement à la plupart des ouvrages qui ne donnent qu'une simple présentation alphabétique, nous avons choisi de regrouper les instructions en catégories d'utilisation. Puis nous présenterons les instructions classées par ordre alphabétique dans chacune de ces catégories. Ces dernières sont résumées ainsi :

Première page	**Catégorie**
115	Entier
118	Segments multiples
119	Système d'exploitation
120	Virgule flottante

Les programmeurs d'applications voudront lire et employer la description du jeu d'instructions du type entier. Toutes les instructions arithmétiques, logiques, de transfert de données et de contrôle simple qu'un pro-

grammeur d'applications utilise sont décrites dans cette partie. C'est la catégorie d'instructions la plus importante.

La deuxième partie décrit les instructions qui sont spécialement destinées aux programmeurs employant la notion de segments multiples dans leurs programmes. Le programmeur d'applications qui utilise un modèle de programmation à une dimension ou à segment unique (notion présentée brièvement dans le Chapitre 2) n'a pas besoin de se perdre dans de telles complexités.

La troisième partie est uniquement destinée aux personnes qui écrivent des systèmes d'exploitation. Ces instructions ont été spécifiquement ajoutées au 80386 pour faciliter le travail d'écriture d'un système d'exploitation. Encore une fois, le programmeur d'applications n'a pas besoin d'instructions aussi complexes. En fait, la plupart des systèmes 80386 ne lui permettront pas du tout d'utiliser ces instructions !

La quatrième et dernière partie comprend les instructions à virgule flottante du 80387. Comme c'est le cas dans tout le livre, nous incluons la description du 80387 comme une partie intégrante d'une architecture à base de 80386. Cependant, nous séparons la description des instructions à virgule flottante, puisque beaucoup de programmeurs n'en ont pas besoin.

Dans chacune de ces parties, chaque instruction (ou un petit groupe d'instructions très liées) est présentée sur une page complète, ou plus si c'est nécessaire, pour offrir une description à la fois complète et concise. Ce format, aussi bien que l'organisation précédemment décrite, facilite la lecture et la compréhension des instructions du processeur. Et surtout, cette présentation fait de ce livre un excellent manuel de référence.

Index alphabétique des instructions

Le tableau suivant résume les jeux d'instructions du 80386 et du 80387 par ordre alphabétique. Il donne le numéro de page et la partie du chapitre où on peut trouver chaque instruction, ainsi qu'une brève description.

Instruction	Section	Page	Description
AAA	Entier	155	Ajustement ASCII après l'addition
AAD	Entier	157	Ajustement ASCII avant la division
AAM	Entier	159	Ajustement ASCII après la multiplication
AAS	Entier	161	Ajustement ASCII après la soustraction
ADC	Entier	163	Addition avec retenue
ADD	Entier	165	Addition
AND	Entier	167	"ET" logique
ARPL	S.E.	377	Ajustement du niveau de privilège requis
BOUND	Entier	169	Vérification des limites d'un tableau
BSF	Entier	171	Recherche de bit en avant
BSR	Entier	173	Recherche de bit en arrière
BT	Entier	175	Test de bit
BTC	Entier	178	Test et complémentation de bit
BTR	Entier	181	Test de bit et remise à zéro
BTS	Entier	184	Test et positionnement de bit
CALL	Entier	187	Appel à un sous-programme
CALL	S.M.	351	Appel de procédure
CBW/CWDE	Entier	189	Conversion d'octet en mot
CLC	Entier	191	Mise à zéro de l'indicateur de retenue
CLD	Entier	192	Mise à zéro de l'indicateur de direction
CLI	Entier	193	Mise à zéro de l'indicateur d'interruption
CLTS	S.E.	379	Effacement de l'indicateur de commutation de tâches
CMC	Entier	195	Complémentation du bit de retenue
CMP	Entier	196	Comparaison
CMPS	Entier	198	Comparaison de chaînes
CMPSB	Entier	198	Comparaison de chaînes
CMPSW	Entier	198	Comparaison de chaînes
CMPSD	Entier	198	Comparaison de chaînes
CWD/CDQ	Entier	201	Conversion de mot en dmot
DAA	Entier	203	Ajustement décimal après addition
DAS	Entier	205	Ajustement décimal après soustraction
DEC	Entier	207	Décrémentation

Instruction	Section	Page	Description
DIV	Entier	209	Division non signée
ENTER	Entier	212	Création du cadre de pile
F2XM1	V.F.	425	$2^x - 1$
FABS	V.F.	427	Valeur absolue
FADD	V.F.	428	Addition
FIADD	V.F	428	Addition
FADDP	V.F	428	Addition
FBLD	V.F.	431	Chargement BCD
FBSTP	V.F.	433	Stockage de BCD et dépilage
FCHS	V.F.	435	Changement de signe
FCLEX	V.F.	436	Effacement des exceptions
FNCLEX	V.F	436	Effacement des exceptions
FCOM	V.F.	438	Comparaison
FCOMP	V.F	438	Comparaison
FCOMPP	V.F	438	Comparaison
FCOS	V.F	441	Cosinus
FDECSTP	V.F.	443	Décrémentation du pointeur de pile
FDIV	V.F.	445	Division
FIDIV	V.F	445	Division
FDIVP	V.F	445	Division
FDIVR	V.F.	448	Division arrière
FIDIVR	V.F	448	Division arrière
FDIVPR	V.F	448	Division arrière
FFREE	V.F.	451	Libération de registre
FICOM	V.F.	453	Comparaison d'entiers
FICOMP	V.F	453	Comparaison d'entiers
FILD	V.F.	456	Chargement d'entier
FINCSTP	V.F.	458	Incrémentation du pointeur de pile
FINIT	V.F.	460	Initialisation du processeur
FNINIT	V.F	460	Initialisation du processeur
FIST	V.F.	462	Stockage d'entier
FISTP	V.F.	464	Stockage d'entier et dépilage
FLD	V.F.	466	Chargement de réel
FLD1	V.F.	468	Chargement de 1
FLDCW	V.F.	470	Chargement du mot de contrôle
FLDENV	V.F.	471	Chargement de l'environnement

Instruction	Section	Page	Description
FLDL2E	V.F.	468	Chargement de $\log_2 e$
FLDL2T	V.F.	468	Chargement de $\log_2 10$
FLDLG2	V.F.	468	Chargement de $\log_{10} 2$
FLDLN2	V.F.	468	Chargement de $\log_e 2$
FLDPI	V.F.	468	Chargement de π
FLDZ	V.F.	468	Chargement du zéro
FMUL	V.F.	473	Multiplication
FIMUL	V.F	473	Multiplication
FMULP	V.F	473	Multiplication
FNOP	V.F.	476	Pas d'opération
FPATAN	V.F.	477	Arctangente partiel
FPREM	V.F.	479	Reste partiel
FPREM1	V.F.	482	Reste partiel - IEEE
FPTAN	V.F.	485	Tangente partielle
FRNDINT	V.F.	487	Arrondi à l'entier
FRSTOR	V.F.	489	Récupération de l'état
FSAVE	V.F.	491	Sauvegarde de l'état
FNSAVE	V.F	491	Sauvegarde de l'état
FSCALE	V.F.	493	Echelle de puissance de deux
FSIN	V.F.	495	Sinus
FSINCOS	V.F.	497	Sinus et cosinus
FSQRT	V.F.	499	Racine carrée
FST	V.F.	501	Stockage de réel
FSTCW	V.F.	503	Stockage du mot de contrôle
FNSTCW	V.F	503	Stockage du mot de contrôle
FSTENV	V.F.	504	Stockage de l'environnement
FNSTENV	V.F	504	Stockage de l'environnement
FSTP	V.F.	506	Stockage de réel et dépilage
FSTSW	V.F.	508	Stockage du mot d'état
FNSTSW	V.F	508	Stockage du mot d'état
FSTSW AX	V.F.	509	Stockage du mot d'état dans AX
FNSTSW AX	V.F	509	Stockage du mot d'état dans AX
FSUB	V.F.	511	Soustraction
FISUB	V.F	511	Soustraction
FSUBP	V.F	511	Soustraction
FSUBR	V.F.	514	Soustraction arrière

Instruction	Section	Page	Description
FISUBR	V.F	514	Soustraction arrière
FSUBRP	V.F	514	Soustraction arrière
FTST	V.F.	517	Test
FUCOM	V.F.	519	Comparaison non ordonnée
FUCOMP	V.F	519	Comparaison non ordonnée
FUCOMPP	V.F	519	Comparaison non ordonnée
FXAM	V.F.	522	Examen
FXCH	V.F.	524	Echange de registres
FXTRACT	V.F.	526	Extraction de l'exposant et de la mantisse
FYL2X	V.F.	528	$y*\log_2 x$
FYL2XP1	V.F.	530	$y*\log_2(x+1)$
HLT	S.E.	381	Halte
IDIV	Entier	214	Division signée
IMUL	Entier	218	Multiplication signée
IN	Entier	227	Entrée E/S
INC	Entier	223	Incrémentation
INS/INSB	Entier	225	Entrée E/S d'une chaîne
INSW/INSD	Entier	225	Entrée E/S d'une chaîne
INT	S.M.	354	Appel à une procédure d'interruption
INTO	S.M.	357	Appel à procédure d'interruption sur débordement
IRET	S.M.	359	Retour d'une procédure d'interruption
JB/JNAE/JC	Entier	228	Saut si inférieur
JBE/JNA	Entier	228	Saut si inférieur ou égal
JCXZ	Entier	228	Saut si CX nul
JECXZ	Entier	228	Saut si ECX nul
JE/JZ	Entier	228	Saut si égal
JL/JNGE	Entier	228	Saut si plus petit
JLE/JNG	Entier	228	Saut si plus petit ou égal
JMP	Entier	231	Saut (intrasegment)
JMP	S.M.	361	Saut (intersegment)
JNB/JAE/JNC	Entier	228	Saut si supérieur ou égal
JNBE/JA	Entier	228	Saut si supérieur
JNE/JNZ	Entier	228	Saut si différent
JNL/JGE	Entier	228	Saut si plus grand ou égal

Instruction	Section	Page	Description
JNLE/JG	Entier	228	Saut si plus grand
JNO	Entier	228	Saut si pas débordement
JNP/JPO	Entier	228	Saut si pas parité
JNS	Entier	228	Saut si pas signe
JO	Entier	228	Saut si débordement
JP/JPE	Entier	228	Saut si parité
JS	Entier	228	Saut si signe
LAHF	Entier	234	Chargement des indicateurs dans AH
LAR	S.E.	382	Chargement des droits d'accès
LDS	S.M.	363	Chargement du pointeur en DS
LEA	Entier	236	Chargement de l'adresse effective
LEAVE	Entier	238	Abandon de la procédure
LES	S.M.	363	Chargement du pointeur en ES
LFS	S.M.	363	Chargement du pointeur en FS
LGDT	S.E.	386	Chargement de la table des descripteurs globaux
LGS	S.M.	363	Chargement du pointeur en GS
LIDT	S.E.	388	Chargement de la table des descripteurs interruptions
LLDT	S.E.	390	Chargement de la table des descripteurs 392locaux
LMSW	S.E.	240	Chargement du mot d'état machine
LOCK	Entier	242	Verrouillage du bus
LODS	Entier	242	Chargement d'une chaîne
LODSB	Entier	242	Chargement d'une chaîne
LODSW	Entier	242	Chargement d'une chaîne
LODSD	Entier	242	Chargement d'une chaîne
LOOP	Entier	245	Boucle avec compteur ECX
LOOPNZ	Entier	247	Boucle avec ECX et test si non nul
LOOPNE	Entier	247	Boucle avec ECX et test si non nul
LOOPZ	Entier	249	Boucle avec ECX et test si nul
LOOPE	Entier	249	Boucle avec ECX et test si nul
LSL	S.E.	394	Chargement de la limite de segment
LSS	S.M.	363	Chargement du pointeur en SS
LTR	S.E.	398	Chargement du registre de tâche

Instruction	Section	Page	Description
MOV	Entier	251	Transfert d'opérandes
MOV	S.M.	365	Transfert deregistre segment
MOV	S.E.	400	Transfert de registre spécial
MOVS	Entier	253	Transfert de chaînes
MOVSB	Entier	253	Transfert de chaînes
MOVSW	Entier	253	Transfert de chaînes
MOVSD	Entier	253	Transfert de chaînes
MOVSX	Entier	256	Transfert avec extension de signe
MOVZX	Entier	258	Transfert avec extension de zéro
MUL	Entier	260	Multiplication non signée
NEG	Entier	262	Complémentation à deux
NOP	Entier	264	Pas d'opération
NOT	Entier	266	"NON" logique
OR	Entier	268	"OU inclusif" logique
OUT	Entier	270	Sortie E/S
OUTS	Entier	272	Sortie E/S d'une chaîne
OUTSB	Entier	272	Sortie E/S d'une chaîne
OUTSW	Entier	272	Sortie E/S d'une chaîne
OUTSD	Entier	272	Sortie E/S d'une chaîne
POP	Entier	275	Dépilage
POP	S.M.	367	Dépilage dans le registre segment
POPA	Entier	277	Dépilage de tous les registres
POPAD	Entier	277	Dépilage de tous les registres
POPF	Entier	279	Dépilage des indicateurs
POPFD	Entier	279	Dépilage des indicateurs
PUSH	Entier	281	Empilage
PUSH	S.M.	369	Empilage
PUSHA	Entier	283	Empilage de tous les registres
PUSHAD	Entier	283	Empilage de tous les registres
RCL	Entier	287	Rotation à gauche avec retenue
RCR	Entier	290	Rotation à droite avec retenue
REP	Entier	293	Répétition
REPE	Entier	295	Répétition tant qu'égal
REPZ	Entier	295	Répétition tant qu'égal

Instruction	Section	Page	Description
REPNE	Entier	297	Répétition tant que pas égal
REPNZ	Entier	297	Répétition tant que pas égal
RET	Entier	299	Retour d'une procédure
RET	S.M.	371	Retour d'une procédure
ROL	Entier	301	Rotation à gauche
ROR	Entier	304	Rotation à droite
SAHF	Entier	307	Rangement de AH dans les indicateurs
SAL	Entier	305	Décalage arithmétique à gauche
SHL	Entier	309	Décalage arithmétique à gauche
SAR	Entier	312	Décalage arithmétique à droite
SBB	Entier	315	Soustraction avec retenue
SCAS	Entier	318	Recherche de chaîne
SCASB	Entier	318	Recherche de chaîne
SCASW	Entier	318	Recherche de chaîne
SCASD	Entier	318	Recherche de chaîne
SETB	Entier	321	Assignation si inférieur
SETNAE	Entier	321	Assignation si inférieur
SETC	Entier	321	Assignation si inférieur
SETBE	Entier	321	Assignation si inférieur ou égal
SETNA	Entier	321	Assignation si inférieur ou égal
SETE	Entier	321	Assignation si égal
SETZ	Entier	321	Assignation si égal
SETL	Entier	321	Assignation si plus petit
SETNGE	Entier	321	Assignation si plus petit
SETLE	Entier	321	Assignation si plus petit ou égal
SETNG	Entier	321	Assignation si plus petit ou égal
SETNB	Entier	321	Assignation si supérieur ou égal
SETAE	Entier	321	Assignation si supérieur ou égal
SETNC	Entier	321	Assignation si supérieur ou égal
SETNBE	Entier	321	Assignation si supérieur
SETA	Entier	321	Assignation si supérieur
SETNE	Entier	321	Assignation si différent
SETNZ	Entier	321	Assignation si différent
SETNL	Entier	321	Assignation si plus grand ou égal
SETGE	Entier	321	Assignation si plus grand ou égal
SETNLE	Entier	321	Assignation si plus grand

Instruction	Section	Page	Description
SETG	Entier	321	Assignation si plus grand
SETNO	Entier	321	Assignation si pas débordement
SETNP	Entier	321	Assignation si pas parité
SETPO	Entier	321	Assignation si pas parité
SETNS	Entier	321	Assignation si pas signe
SETO	Entier	321	Assignation si débordement
SETP	Entier	321	Assignation si parité
SETPE	Entier	321	Assignation si parité
SETS	Entier	321	Assignation si signe
SGDT	S.E.	402	Stockage de la table des descripteurs "é'globaux
SHLD	Entier	324	Décalage à gauche double
SHR	Entier	327	Décalage logique à droite
SHRD	Entier	329	Décalage à droite double
SIDT	S.E.	404	Stockage de la table des descripteurs d'interruptions
SLDT	S.E.	406	Stockage de la table des descripteurs locaux
SMSW	S.E.	408	Stockage du mot d'état machine
STC	Entier	332	Positionnement de l'indicateur de retenue
STD	Entier	333	Positionnement de l'indicateur de direction
STI	Entier	334	Positionnement de l'indicateur d'interruption
STOS	Entier	336	Stockage de chaîne
STOSB	Entier	336	Stockage de chaîne
STOSW	Entier	336	Stockage de chaîne
STOSD	Entier	336	Stockage de chaîne
STR	S.E.	410	Stockage du registre de tâche
SUB	Entier	338	Soustraction
TEST	Entier	340	Comparaison logique
VERR	S.E.	412	Vérification du segment en lecture
VERW	S.E.	415	Vérification du segment en écriture
WAIT	V.F.	532	Attente pour le coprocesseur
XCHG	Entier	342	Echange

Instruction	Section	Page	Description
XLAT	Entier	344	Translation avec table
XLATB	Entier	344	Translation avec table
XOR	Entier	346	"OU exclusif" logique

Format de description d'une instruction

Voici maintenant un lexique de notre description du jeu d'instructions. Plusieurs possibilités se présentent. Par exemple, vous pouvez décrire un marteau en expliquant comment l'utiliser. Vous pouvez aussi en faire une description matérielle en précisant son poids, sa forme, ou son apparence. Il en va de même pour chaque instruction qui est décrite de plusieurs façons dans le jeu d'instructions qui va suivre.

Nous allons tout d'abord donner la syntaxe en assembleur de chaque instruction. Puis suivra une description littérale plus facile à comprendre qu'une définition formelle dans bien des cas. On donnera ensuite cette dernière en fournissant une définition sémantique de l'instruction. Ensuite, nous proposerons un exemple d'emploi de l'instruction et de ses effets. Pour finir, nous fournirons une liste des exceptions (ce qui peut s'être mal passé) pour chaque instruction. On ajoutera à l'occasion une partie notes avec une liste comprenant quelques particularités intéressantes. Cette partie pourra aussi comporter des références à d'autres pages du livre où l'instruction est présentée de manière plus approfondie.

Format d'une instruction

Donnons un exemple de format d'instruction.

ADD	op_1	op_2
	reg	reg
	reg	mem
	reg	imm
	mem	reg
	mem	imm

Ceci indique que l'instruction ADD a cinq formes possibles. Elle peut avoir deux opérandes registre (reg,reg), un registre et une mémoire (reg,mem), etc. Les abréviations op_1 et op_2 sont employées dans la description formelle qui suit. Ainsi, dans la partie fonctionnement, on fait simplement référence à $op_1 = op_1 + op_2$ au lieu de décrire spécifiquement chaque possibilité (reg = reg + reg, reg = reg + mem, etc.).

Nous allons donner une liste de tous les mnémoniques d'opérandes possibles et décrire brièvement chacun d'eux.

reg — Tout registre général 8, 16, 32 bits décrit dans le Chapitre 2, Figure 2.1. L'abréviation *reg* se rapporte à un registre de taille quelconque. Une taille particulière sera notée par reg8, reg16, reg32. Certaines instructions emploient des registres particuliers qui sont explicitement ou implicitement précisés. D'autres instructions utilisent plusieurs registres avec un registre spécifié.

mem — Toute forme d'adressage mémoire décrite dans le Chapitre 2 peut être employée quand *mem* fait partie du format de l'instruction. On peut ainsi faire référence à des opérandes mémoire de 8, 16, 32, 48, 64, 80 bits et 28 et 108 octets. On notera ces formats ainsi : mem8, mem16, mem32, etc. L'abréviation *mem* se rapporte à mem8, mem16, mem32. Toute autre taille de mémoire est explicitement précisée.

imm — Toute constante immédiate décrite aussi dans le Chapitre 2 et permise dans certaines instructions. L'abréviation *imm* fait référence à une constante immédiate de taille quelconque (8, 16, 32 bits). Toute taille particulière sera notée imm8, imm16, imm32.

sreg — On fait référence à un registre segment par *sreg* (voir le Chapitre 2).

ST — De nombreux types de données à virgule flottante emploient la pile accumulateur (voir le Chapitre 2). Le sommet de la pile

s'appelle ST ou ST(0), et les éléments à l'intérieur de la pile sont notés ST(n). Se reporter au Chapitre 2.

; On emploie le ; (point-virgule) comme directive de commentaire en langage assembleur. Ainsi :

ADD EAX,EBX ; Ceci est une addition

est une ligne en langage assembleur avec un commentaire. Vous trouverez le point-virgule dans les parties format et exemple.

Notons que l'écriture d'une ligne suivant ce format (ADD reg,reg par exemple) implique que les opérandes soient de la même taille (à moins que la taille des opérandes ne soit spécifiée). Ainsi, le format :

ADD reg,reg

permet : reg8,reg8, reg16,reg16 ou reg32,reg32, mais jamais reg8,reg32, reg16,reg8, etc. La même notion d'opérandes équivalents se retrouve dans les mnémoniques mémoire. Ainsi :

ADD reg,mem

permet reg8,mem8 ; reg16,mem16 ; reg32,mem32.

Notons aussi que les opérations à deux opérandes (binaires) sont généralement de la forme :

$op_1 = op_1 + op_2$

où + est l'opérateur approprié de l'instruction. Cependant, la destination de l'opération est généralement op_1. Par exemple :

ADD reg,mem

fait que reg prend la valeur de reg + mem.

Description d'une instruction

Nous allons donner un exemple de description d'une instruction.

> AAA effectue un ajustement BCD (non compacté) du contenu du registre AL après une addition d'octets. Normalement, AAA suit une addition d'octets du contenu BCD non compacté de AL, mais on peut l'employer pour toute autre conversion BCD.
>
> Si le résultat de l'addition donne une retenue décimale, ou si le demi-octet inférieur de AL est supérieur à 9, alors AL est incrémenté de 6, AH de 1, et les bits CF et AF sont levés. S'il n'y a pas eu de retenue décimale, alors AH n'est pas modifié, CF et AF sont mis à 0, et les 4 bits inférieurs de AL ne sont pas modifiés. Qu'il y ait eu retenue ou non, les quatre bits supérieurs de AL sont mis à zéro.

Cette description littérale est tirée de l'instruction AAA. Quand cela est possible, on donne un algorithme différent de celui de la partie fonctionnement. On indique aussi le but et l'emploi de l'instruction si nécessaire.

Fonctionnement d'une instruction

Nous allons donner un exemple de la partie fonctionnement de l'instruction.

```
AL = AL + (10 * AH);
AH = 0;
SF = B(7,AL);
ZF = AL == 0;
PF = ~(B(7,AL)^B(6,AL)^...^B(0,AL));
AF = OF = CF = INDEFINI;
```

Ceci est la description de l'algorithme associé à l'instruction AAD. L'ordre des lignes fait partie de la sémantique de l'instruction. Ainsi, la ligne :

```
ZF = AL == 0
```

se rapporte à la valeur de AL après que la première ligne :

```
AL = AL + (10 * AH)
```

a été exécutée. La partie fonctionnement décrit tous les effets de l'instruction. Si un registre, un indicateur, ou un opérande n'est pas inclus dans cette description, c'est qu'il n'est pas affecté par l'instruction.

Nous allons donner une description de la syntaxe utilisée dans cette partie fonctionnement (aussi bien que dans les autres parties de la description du jeu d'instructions).

Exemples d'une instruction

Nous donnons des exemples pour clarifier le fonctionnement de chaque instruction. Nous avons testé tous les exemples sur un système 386, afin de nous assurer qu'ils sont corrects. Voici un exemple représentatif :

```
                                                        OSZAP
ADD EAX,EBX          Avant:  EAX:  01234567  EBX:  76543210  XXXXXX
                     Après:  EAX:  77777777  EBX:  76543210  000010

ADD mem32,70000000h  Avant:  Mem:  7FFFFFFF                  XXXXXX
                     Après:  Mem:  EFFFFFFF                  110010
```

Cet exemple tiré de l'instruction ADD comporte deux parties. A gauche, on a le code en langage machine de l'instruction. Cette syntaxe est celle de notre système 386. Cependant, la vôtre peut être légèrement différente. A droite, on a l'état avant et après l'exécution de l'instruction. L'état du registre est donné en hexadécimal, même s'il n'a pas de suffixe h. L'état des indicateurs est donné en binaire. La colonne abrégée OSZAPC se rapporte aux bits OF, SF, ZF, AF, PF et CF. Dans les exemples, un X signifie "ne pas tenir compte" ou "inconnu". Dans l'exemple précédent, l'état des indicateurs était inconnu avant l'exécution de l'instruction ADD. Cependant, leur état ne peut pas modifier l'exécution de l'instruction ADD. Ainsi, on ne tient pas compte des indicateurs pour cette instruction.

U indique qu'il n'y a pas eu de changement (*unchanged*). Dans l'exemple suivant de l'instruction JO (saut si débordement), les indicateurs (excepté pour le débordement) sont inconnus ou non pris en compte avant l'exécution de JO, et l'instruction ne les modifie pas.

```
JO   near label1   ;label1 se trouve 68h octets plus loin dans le programme
                                        OSZAPC
          Avant:   EIP:   00000300      1XXXXX
          Après:   EIP:   00000368      UUUUUU
```

Après cette instruction, tous les indicateurs demeurent inchangés, comme nous l'indiquent les U dans l'exemple précédent.

Exceptions associées à une instruction

Chaque instruction a plusieurs conditions d'exception possibles. Dans la description du jeu d'instructions, on emploie l'appel à une fonction pour résumer une condition d'exception potentielle. Par exemple, toute instruction comprenant un opérande mémoire peut générer un type différent d'exception selon l'opérande mémoire. Ainsi, un "appel" à la fonction Memory(), décrite ultérieurement, testera les conditions d'erreur possibles.

Décrire les conditions d'erreur sous cette forme abrégée devrait éliminer des descriptions répétitives qui n'ajouteraient que peu d'intérêt à la description de l'instruction.

Si une exception doit survenir, le résultat sera une exception du type décrit plus bas. Notons que si aucune exception n'est précisée, l'instruction ne peut pas générer d'exception. Les exceptions sont décrites dans le Chapitre 6.

```
Memory(){
   /* Toutes les exceptions qui sont possibles lors d'un accès
      mémoire sont contenues dans la routine AccessVirtual() définie
      dans le Chapitre 5. S'y reporter pour plus de précisions.
   */
}
Stack(){
   /* Toutes les exceptions qui sont possibles lors d'un accès à la
      pile sont contenues dans la routine AccessVirtual() définie
      dans le Chapitre 5 avec le SReg égal à SS. S'y reporter pour
      plus de précisions.
   */
}
CodeReference(){
   /* La destination d'une instruction de transfert est vérifiée. Une
      exception segment est générée dans l'instruction de transfert
      si le premier octet de la destination n'est pas compris dans
      les limites du segment. Tant que l'instruction n'a pas été
      traitée, les exceptions pages ne sont pas générées. Sinon,
      elles sont générées de la même façon que les exceptions
      segments.
   */
si (EIP > CS.Limit)
   /* La routine SegmentException() est définie dans le Chapitre 6.
      S'y reporter pour plus de précisions.
   */
   SegmentException($GP,0);
}
AccessIO(){
   /* Définie dans le Chapitre 5 avec toutes les conditions
```

```
      d'exception possibles. S'y reporter pour plus de précisions.
   */
}
IOPLSensitive(){
   /* La routine SegmentException() est définie dans le Chapitre 6.
      On parle plus précisément de CPL et IOPL dans le Chapitre 5.
   */
   si (CPL > IOPL){
      SegmentException($GP,0)
   }
}
InvalidOpCode(){
   /* Défini dans le Chapitre 6 */
}
```

Notes sur une instruction

Le but de la partie notes est de soulever certains points particuliers que vous devez connaître. Par exemple, on mentionnera toute instruction nouvelle du 386 ou du 387, ou toute instruction qui travaille différemment par rapport aux autres circuits de la famille 86. On relèvera aussi tout résultat inattendu ou aspect particulier d'une instruction. Souvent aussi, on notera des références à une description ou à un exemple de l'ouvrage.

Syntaxe de description d'une instruction

Dans ce paragraphe, nous donnons une description de la syntaxe utilisée pour décrire les instructions dans les parties fonctionnement et exception. Cette syntaxe est basée sur le langage de programmation C. Nous allons rappeler la notation C et ajouter certaines extensions qui nous sont propres. Nous emploierons souvent une syntaxe différente du C réglementaire. Ceci pour une meilleure clarté et concision.

Opérateurs logiques

&	ET
|	OU inclusif
^	OU exclusif
<<	Décalage à gauche
>>	Décalage à droite
~	Complément à un

Opérateurs arithmétiques

+	Addition
-	Soustraction
/	Division
%	Modulo
++	Incrémentation
--	Décrémentation

Opérateurs relationnels et connecteurs logiques

>	Plus grand que
<	Plus petit que
>=	Plus grand que ou égal à
<=	Plus petit que ou égal à
==	Egal
!=	Différent
||	Ou
&&	Et

Autres

=	Assignation
/*	Début de commentaire
*/	Fin de commentaire
{	Ouverture d'un bloc
}	Fermeture d'un bloc
&	Adresse d'un opérande

[a]	L'opérande est à l'adresse a
a is b	Vrai si a est du type b
c in (s)	Vrai si a appartient à l'ensemble de s

Constantes

DENORMAL	Nombre dénormalisé à virgule flottante
EMPTY	Registre à virgule flottante vide
+INFINITY	Représentation à virgule flottante de +infini
-INFINITY	Représentation à virgule flottante de -infini
MSB	Le bit le plus significatif
NaN	Non représentable en virgule flottante
NORMAL	Nombre normalisé à virgule flottante
NUMOPS	Nombre d'opérandes spécifiés
OPCODE	CodeOp de l'instruction
QUIETNaN	NaN à virgule flottante silencieux
UNDEFINED	Résultat indéfini
UNSUPPORTED	Représentation impossible

Mnémoniques des registres

EAX	Registre 32 bits EAX
ECX	Registre 32 bits ECX
EDX	Registre 32 bits EDX
EBX	Registre 32 bits EBX
ESP	Registre 32 bits ESP
EBP	Registre 32 bits EBP
ESI	Registre 32 bits ESI
EDI	Registre 32 bits EDI
AX	Registre 16 bits AX
CX	Registre 16 bits CX
DX	Registre 16 bits DX
BX	Registre 16 bits BX
SP	Registre 16 bits SP
BP	Registre 16 bits BP
SI	Registre 16 bits SI
DI	Registre 16 bits DI
AL	Registre 8 bits AL

CL	Registre 8 bits CL
DL	Registre 8 bits DL
BL	Registre 8 bits BL
AH	Registre 8 bits AH
CH	Registre 8 bits CH
DH	Registre 8 bits DH
BH	Registre 8 bits BH
CS	Registre 16 bits CS (segment code)
DS	Registre 16 bits DS (segment donnée)
SS	Registre 16 bits SS (segment pile)
ES	Registre 16 bits ES (extra segment)
FS	Registre 16 bits FS (extra segment)
GS	Registre 16 bits GS (extra segment)
EIP	Pointeur 32 bits d'instruction
CR0	Registre de contrôle 32 bits n°0
CR2	Registre de contrôle 32 bits n°2
CR3	Registre de contrôle 32 bits n°3
DR0	Registre de mise au point 32 bits n°0
DR1	Registre de mise au point 32 bits n°1
DR2	Registre de mise au point 32 bits n°2
DR3	Registre de mise au point 32 bits n°3
DR6	Registre de mise au point 32 bits n°6
DR7	Registre de mise au point 32 bits n°7
EFLAGS	Registre 32 bits des indicateurs
MSW	Mot d'état machine
LDTR	Registre pointant la table des descripteurs locaux
IDTR	Registre pointant la table des descripteurs d'interruption
GDTR	Registre pointant la table des descripteurs globaux
TR	Registre de tâche
TW	Mot étiquette en virgule flottante
SW	Mot d'état en virgule flottante
CW	Mot de contrôle en virgule flottante
ST	Sommet de pile virgule flottante
ST(n)	Nème registre de la pile à virgule flottante
TOP	Pointeur de sommet de pile
FIP	Pointeur d'instruction en virgule flottante
FCS	Segment code en virgule flottante

FOO	Déplacement de l'opérande en virgule flottante
FOS	Segment de l'opérande en virgule flottante

Mnémoniques des indicateurs

CF	Indicateur de retenue
PF	Indicateur de parité
AF	Indicateur de retenue auxiliaire
ZF	Indicateur de zéro
SF	Indicateur de signe
TF	Indicateur d'autorisation de trappe
IF	Indicateur d'interruption
DF	Indicateur de direction
OF	Indicateur de débordement
IOPL	Niveau de privilège E/S
NT	Indicateur de tâche
RF	Autorisation d'erreur de mise au point
VM	Mode virtuel 8086
C0	Code de condition 0 en virgule flottante
C1	Code de condition 1 en virgule flottante
C2	Code de condition 2 en virgule flottante
C3	Code de condition 3 en virgule flottante

Autres

CPL	Niveau de privilège courant
DPL	Niveau de privilège descripteur
LOCK	Le signal LOCK du bus
NMI	Le signal d'interruption non masquable
RPL	Niveau de privilège requis
PE	Autorisation de protection

Autres mnémoniques

imm8	Immédiat 8 bits
imm16	Immédiat 16 bits
imm32	Immédiat 32 bits
imm	Immédiat 8, 16, ou 32 bits

mem8	Pointeur mémoire 8 bits
mem16	Pointeur mémoire 16 bits
mem32	Pointeur mémoire 32 bits
mem48	Pointeur mémoire 48 bits
mem64	Pointeur mémoire 64 bits
mem80	Pointeur mémoire 80 bits
mem	Pointeur mémoire 8, 16, ou 32 bits
reg8	Registre 8 bits
reg16	Registre 16 bits
reg32	Registre 32 bits
reg	Registre 8, 16, ou 32 bits
sreg	Registre segment

Variables générales

cnt	Variable utilisée comme compteur de bits
delta	Variable utilisée comme descripteur de chaîne
disp	Déplacement d'octet
i	Variable de boucle
offset	Déplacement de bit
quotient	Variable utilisée pour stocker un quotient temporaire
tmp,temp	Variable diverse

Types de données

shortReal	Réel court en virgule flottante (32 bits)
longReal	Réel long en virgule flottante (64 bits)
tempReal	Réel temporaire en virgule flottante (80 bits)
wordInt	Mot entier 80387 (16 bits)
shortInt	Entier court 80387 (32 bits)
longInt	Entier long 80387 (64 bits)
int	Entier

Structures de contrôle

if (e) b1	Si l'expression e est vraie, exécuter le bloc b1
else b2	Sinon exécuter le bloc b2

while (e) b3	Tant que l'expression e est vraie, exécuter le bloc b3
for (e1;e2;e3), b4	Exécuter e1; tant que e2 exécuter b4 et e3
switch (e1) { }	Conditions multiples
case c1:	Vrai si la constante c1 est rencontrée
break:	Sortie immédiate d'un niveau de programme

La plupart de ces abréviations doivent vous être familières, ou, du moins, elles sont faciles à comprendre. Quelques-unes, cependant, peuvent nécessiter une explication plus détaillée.

Les *définitions de types* sont très proches de celles du C. Elles permettent la conversion d'un type de données en un autre. Par exemple :

```
X = (shortInt) Y;
```

entraîne que X est assigné à la valeur de Y après que Y a été converti en type de données entier court.

L'opérateur *is* permet de tester les types de données. Par exemple

```
if (op1 is mem){
whatever...
}
```

exécute "whatever..." si op_1 (le premier opérande) est une référence à la mémoire.

L'opérateur *in* apporte la notion d'ensemble. Par exemple :

```
if (op1 in (mem32, mem64, mem80)){
Memory();
}
```

appelle la routine d'exception Memory() si op_1 appartient à l'ensemble donné : mem32, mem64, mem80.

Fonctions

On emploie plusieurs procédures prédéfinies dans la description du jeu d'instructions. Nous allons définir chacune d'elles :

```
/* Retourne le bit "N" déterminé par le bitN de op */
B(bitN,op){
      return((op >> bitN) & 01b);
}

/* Retourne les bits de num au bit "N" de op déterminé par bitN et num */
Bits(bitN, num, op){
      for (mask=0;i=0;i< num;i++){
          mask = (mask << 1) | 1;
      }
      return((op >> bitN) & mask);
}

/* Retourne la retenue 0 du bitN à partir de la dernière opération de l'ALU.*/
/* Cela assure l'existence d'une variable particulière qui contient les     */
/* bits de retenue de l'addition la plus récente                            */
Carry0(bitN){
      return((CarryChain >> bitN) & 01b);
}

/* Retourne la retenue 0 négative du bitN à partir de la dernière opération */
/* de l'ALU. Cela assure l'existence d'une variable particulière qui        */
/* contient les bits de retenue négative de la soustraction la plus récente */
```

```
Borrow0(bitN){
      return((BorrowChain >> bitN) & 01b);
}

/* Retourne la longueur en bits de op                                    */
/* Notez que sizeof est une fonction C qui retourne la taille d'un type de*/
/* données en nombre d'octets                                            */
Length(op){
      return(sizeof(op) * 8);
}

/* On utilise une extension de signe pour étendre op de longueur OpLen à  */
/* longueur déterminée par DesLen                                         */
SignEx(DesLen,OpLen,op){
      sign = B(OpLen - 1,op);
      tmp = op;
      for (i=OpLen;i< DesLen;i++){
         tmp = (sign << i) | tmp;
      }
      return(tmp);
}

/* op1 est concaténé avec op2, lenOfRes étant la longueur finale, lenOfOp1*/
/* la longueur de op1. lenOfRes moins lenOfOp1 donne la longueur de op2   */
ConCat(lenOfRes, op2, lenOfOp1, op1){
      mask = 0
      for (i=0;i<lenOfRes;i++){
         mask = (1 << i) | mask;
      }
return(mask & ((op2 << lenOfOp1) | op1));
}

Push(op){
      len = Length(op) / 8;
      SS: [ESP] ;
      ESP = ESP - len;
}

Pop(op){
      len = Length(op) / 8;
      tmp = SS: [ESP] ;
```

```
        ESP = ESP + len;
        return(tmp);
}

Port(op){
        /* La routine port est un appel simplifié à la routine AccessIO()*/
        /* définie dans le Chapitre 5.                                   */
        /* AccessIO(numéro du port, longueur de l'opération, adresse de  */
        /* la donnée, écriture ou lecture);                              */
        /* Se reporter au Chapitre 5 pour plus de précisions             */
        if (OPCODE in (OUT,OUTS)){
           AccessIO(op, Length(op)/8, &op , 1 /* écriture */);
        } else { /* lecture à partir d'un port */
           AccessIO(op, Length(op)/8, &tmp, 0 /* lecture */);
           return(*tmp);
        }
}

/* Employé par les instructions sur les chaînes pour déterminer le segment*/
/* à employer pour les références à un élément d'une chaîne (ESI)         */
SegReg(op){
        if (op donne un recouvrement de segment){
           return(segment de op)
        } else {
           return(DS);
        }
}

/* La boucle infini */
StopExecution(){
        while(1);
}
```

Maintenant que vous connaissez la syntaxe de notre langage de description du jeu d'instructions, vous pouvez vous lancer dans cette description du 80386 instruction par instruction.

ENTIER

La partie concernant les instructions du type entier fournit le jeu complet d'instructions utiles au programmeur d'applications. Vous y trouverez toutes les instructions arithmétiques, logiques, de manipulation de bits, de transfert de données, d'opérations sans privilège sur les indicateurs, de langages de haut niveau, et de traitements sur les chaînes.

Chaque instruction comprend un ou deux exemples d'utilisation. Dans le Chapitre 4, nous donnerons plusieurs exemples, des segments de programme, qui sont constitués de plusieurs instructions et qui effectuent des fonctions utiles. Grâce à cette description, aux exemples donnés ici et à ceux plus compliqués du Chapitre 4, vous devriez mieux comprendre l'utilité et le fonctionnement de ces instructions.

Les tableaux suivants résument toutes les instructions de cette partie en les divisant en sous-groupes. Ensuite, on trouvera la description page à page de chaque instruction.

Voici les instructions entières arithmétiques qui travaillent sur les entiers signés et non signés, les données ASCII et BCD :

Arithmétique

AAA	Ajustement ASCII après l'addition
AAD	Ajustement ASCII avant la division
AAM	Ajustement ASCII après la multiplication
AAS	Ajustement ASCII après la soustraction
ADC	Addition avec retenue
ADD	Addition
CBW	Conversion d'octet en mot
CWDE	""
CMP	Comparaison
CWD	Conversion de mot en dmot
CDQ	""
DAA	Ajustement décimal après addition
DAS	Ajustement décimal après soustraction

DEC	Décrémentation
DIV	Division non signée
IDIV	Division signée
IMUL	Multiplication signée
INC	Incrémentation
MUL	Multiplication non signée
NEG	Complémentation à deux
SBB	Soustraction avec retenue
SUB	Soustraction

Voici les instructions qui travaillent sur les types de données binaires :

	Bit
BSF	Recherche de bit en avant
BSR	Recherche de bit en arrière
BT	Test de bit
BTC	Test et complémentation de bit
BTR	Test de bit et remise à zéro
BTS	Test et positionnement de bit

Voici les instructions qui positionnent conditionnellement un octet relativement à l'état d'un indicateur.

	Assignation conditionnelle
SETB/SETNAE/SETC	Assignation si inférieur
SETBE/SETNA	Assignation si inférieur ou égal
SETE/SETZ	Assignation si égal
SETL/SETNGE	Assignation si plus petit
SETLE/SETNG	Assignation si plus petit ou égal
SETNB/SETAE/SETNC	Assignation si supérieur ou égal
SETNBE/SETA	Assignation si supérieur
SETNE/SETNZ	Assignation si différent
SETNL/SETGE	Assignation si plus grand ou égal
SETNLE/SETG	Assignation si plus grand
SETNO	Assignation si pas de débordement
SETNP/SETPO	Assignation si pas parité

SETNS	Assignation si pas signe
SETO	Assignation si débordement
SETP/SETPE	Assignation si parité
SETS	Assignation si signe

Les instructions suivantes permettent le transfert conditionnel ou inconditionnel de contrôle à partir d'une suite séquentielle d'instructions.

Transfert de contrôle

CALL	Appel d'une procédure
JB/JNAE/JC	Saut si inférieur
JBE/JNA	Saut si inférieur ou égal
JCXZ/JECXZ	Saut si CX nul / ECX nul
JE/JZ	Saut si égal
JL/JNGE	Saut si plus petit
JLE/JNG	Saut si plus petit ou égal
JMP	Branchement
JNB/JAE/JNC	Saut si supérieur ou égal
JNBE/JA	Saut si supérieur
JNL/JGE	Saut si plus grand ou égal
JNLE/JG	Saut si plus grand
JNO	Saut si pas débordement
JNP/JPO	Saut si pas parité
JNS	Saut si pas signe
JO	Saut si débordement
JP/JPE	Saut si parité
JS	Saut si signe
LOOP	Boucle avec ECX comme compteur
LOOPNZ/LOOPNE	Boucle avec ECX si pas zéro/différent
LOOPZ/LOOPE	Boucle avec ECX si zéro/égal
RET	Retour d'une procédure

Le prochain jeu d'instructions permet le transfert de données de (ou à partir de) la mémoire, de la pile, des ports d'entrée/sortie, et des registres.

Transfert de données

IN	Entrée à partir d'un port
LEA	Chargement de l'adresse effective
MOV	Transfert d'opérandes
MOVSX	Transfert avec extension de signe
MOVZX	Transfert avec extension de zéro
OUT	Sortie E/S
POP	Dépilage
POPA/POPAD	Dépilage de tous les registres
PUSH	Empilage
PUSHA	Empilage de tous les registres
XCHG	Echange

Les instructions suivantes permettent la modification individuelle et le transfert en bloc des indicateurs du (ou à partir du) registre AH et de la pile.

Contrôle des indicateurs

CLC	Remise à zéro de l'indicateur de retenue
CLD	Mise à zéro de l'indicateur de direction
CLI	Mise à zéro de l'indicateur d'interruption
CMC	Complémentation du bit de retenue
LAHF	Chargement des indicateurs dans AH
POPF/POPFD	Dépilage des indicateurs
PUSHF	Empilage des indicateurs
SAHF	Rangement de AH dans les indicateurs
STC	Positionnement de l'indicateur de retenue
STD	Positionnement de l'indicateur de direction
STI	Positionnement de l'indicateur d'interruption

Les instructions suivantes facilitent l'écriture de langages de haut niveau.

Langage de haut niveau

BOUND	Vérification des limites d'un tableau
ENTER	Création du cadre de pile
LEAVE	Abandon de la procédure

Ces instructions sont des opérateurs logiques standard.

Logique

AND	Opérateur "ET"
NOT	Opérateur "NON"
OR	Opérateur "OU inclusif"
RCL	Rotation à gauche avec retenue
RCR	Rotation à droite avec retenue
ROL	Rotation à gauche
ROR	Rotation à droite
SAL/SHL	Décalage arithmétique/logique à gauche
SAR	Décalage arithmétique à droite
SHLD	Décalage à gauche double
SHR	Décalage logique à droite
SHRD	Décalage à droite double
TEST	Comparaison logique
XOR	Opérateur "OU exclusif"

Les instructions suivantes effectuent des opérations sur les types de données chaînes.

Chaîne

CMPS/CMPSB/CMPSW/CMPSD	Comparaison de chaînes
INS/INSB/INSW/INSD	Entrée E/S d'une chaîne
LODS/LODSB/LODSW/LODSD	Chargement d'une chaîne
MOVS/MOVSB/MOVSW/MOVSD	Transfert de chaînes
OUTS/OUTSB/OUTSW/OUTSD	Sortie E/S d'une chaîne
REP	Répétition
REPE/REPZ	Répétition tant qu'égal
REPNE/REPNZ	Répétition tant que pas égal

SCAS/SCASB/SCASW/SCASD	Recherche de chaîne
STOS/STOSB/STOSW/STOSD	Stockage de chaîne
XLAT/XLATB	Translation avec table

Pour finir, les dernières instructions n'entrent dans aucune des catégories précédentes.

Autres

LOCK	Verrouillage du bus
NOP	Pas d'opération

AAA

Ajustement ASCII après l'addition

Format :

AAA

Description :

AAA effectue un ajustement BCD (non compacté) du contenu du registre AL après une addition d'octets. Normalement, AAA suit une addition d'octets du contenu BCD non compacté de AL, mais on peut l'employer pour toute autre conversion BCD.

Si le résultat de l'addition donne une retenue décimale, ou si le demi-octet inférieur de AL est supérieur à 9, alors AL est incrémenté de 6, AH de 1, et les bits CF et AF sont levés. S'il n'y a pas eu de retenue décimale, alors AH n'est pas modifié, CF et AF sont mis à 0, et les quatre bits inférieurs de AL ne sont pas modifiés. Qu'il y ait eu retenue ou non, les quatre bits supérieurs de AL sont mis à zéro.

Fonctionnement :

```
if ((AL & 0Fh) > 9) || (AF == 1) {
   AL = AL + 6;
   AH = AH + 1;
   AF = 1;
   CF = 1;
} else {
   AF = 0;
   CF = 0;
}
AL = AL & 0Fh;

OF = SF = ZF = PF = UNDEFINED;
```

Exemples :

					OSZAPC
ADD AL,BL	Avant:	AH: 05	AL: 07	BL: 09	XXXXXX
	Après:	AH: 05	AL: 10	BL: 09	000100
AAA	Avant:	AH: 05	AL: 10	BL: 09	000100
	Après:	AH: 06	AL: 06	BL: 09	XXX1X1
ADD AL,BL	Avant:	AH: 05	AL: 07	BL: 02	XXXXXX
	Après:	AH: 05	AL: 09	BL: 02	000010
AAA	Avant:	AH: 05	AL: 09	BL: 02	000010
	Après:	AH: 05	AL: 09	BL: 02	XXX0X0

Exceptions :

Aucune.

AAD

Ajustement ASCII avant la division

Format :

AAD

Description :

AAD ajuste le numérateur dans AL avant la division de deux nombres BCD non compactés. L'ajustement est fait de façon que le quotient de la division soit un résultat BCD valide.

Le registre AL est chargé à : AL + (10 * AH). Le registre AH est mis à zéro. L'opération de division laissera un quotient BCD valide dans AL et le reste dans AH.

Fonctionnement :

```
AL = AL + (10 * AH);
AH = 0;
SF = B (7,AL)
ZF = AL == 0
PF = ~(B(7,AL)^B(6,AL)^...^B(0,AL));
AF = OF= CF = UNDEFINED;
```

Exemples :

					OSZAPC
AAD	Avant:	AH: 05	AL: 07	BL: 09	XXXXXX
	Après:	AH: 00	AL: 39	BL: 09	X00X1X
DIV BL	Avant:	AH: 00	AL: 39	BL: 09	X00X1X
	Après:	AH: 03	AL: 06	BL: 09	XXX1X1

Exceptions :

Aucune.

AAM

Ajustement ASCII après la multiplication

Format :

AAM

Description :

Une MUL de deux chiffres BCD peut donner un résultat BCD non valide. AAM convertit ce résultat en une paire de chiffres BCD, qui sont placés dans le registre AX (respectivement AH et AL).

Ainsi, AAM suit généralement l'opération de multiplication de deux chiffres BCD. Le registre AH reçoit la valeur de AL divisée par 10. La nouvelle valeur de AL est égale à l'ancienne valeur de AL modulo 10.

Fonctionnement :

```
AH = AH / 10;
AL = AL % 10;
SF = B(7,AL);
ZF = AL == 0;
PF = ~(B(7,AL)^B(6,AL)^...^B(0,AL));
OF = AF = CF = UNDEFINED;
```

Exemples :

					OSZAPC
MUL BL	Avant:	AH: 00	AL: 07	BL: 09	XXXXXX
	Après:	AH: 00	AL: 3F	BL: 09	0XXXX0
AAM	Avant:	AH: 00	AL: 3F	BL: 09	0XXXX0
	Après:	AH: 06	AL: 03	BL: 09	X00X1X

Exceptions :

Aucune.

AAS

Ajustement ASCII après soustraction

Format :

AAS

Description :

AAS opérera un ajustement BCD (non compacté) du contenu du registre AL après une soustraction d'octets. Ainsi, on devrait utiliser cette instruction uniquement après une soustraction d'octets sur le contenu non compacté du registre AL.

Si la soustraction donne une retenue négative décimale, alors AL est décrémenté de 6, AH de 1, et les bits CF et AF sont levés. S'il n'y a pas eu de retenue décimale, alors AH n'est pas changé, CF et AL sont mis à zéro, et les 4 bits de poids faible de AL sont mis à zéro.

Fonctionnement :

```
if ((AL & 0Fh) > 9) || (AF == 1) {
        AL = AL - 6;
        AH = AH - 1;
        AF = 1;
        CF = 1;
} else {
        AF = 0;
        CF = 0;
}
AL = AL & 0Fh;
OF = SF = ZF = PF = UNDEFINED;
```

Exemples :

					OSZAPC
SUB AL,BL	Avant:	AH: 05	AL: 07	BL: 09	XXXXXX
	Après:	AH: 05	AL: FE	BL: 09	010101
AAS	Avant:	AH: 05	AL: FE	BL: 09	010101
	Après:	AH: 04	AL: 08	BL: 09	XXX1X1
SUB AL,BL	Avant:	AH: 05	AL: 09	BL: 07	XXXXXX
	Après:	AH: 05	AL: 02	BL: 07	000000
AAS	Avant:	AH: 05	AL: 02	BL: 07	000000
	Après:	AH: 05	AL: 02	BL: 07	XXX0X0

Exceptions :

Aucune.

ADC

Addition avec retenue

Formats :

ADC	op_1	,	op_2
	reg	,	reg
	reg	,	mem
	reg	,	imm
	mem	,	reg
	mem	,	imm

Description :

Cette instruction effectue une addition sur op_1, op_2 et c_i et place le résultat dans op_1. On emploie généralement ADC comme élément de plusieurs additions successives sur octets, mots, ou dmots.

Fonctionnement :

```
op1 = op1 + op2 + CF;
OF = Carry0(MSB)^Carry0(MSB - 1);
SF = B(MSB, op1);
ZF = op1 == 0;
AF = Carry0(3);
PF = ~(B(7,op1)^B(6,op1)^...^B(0,op1));
CF = Carry0(MSB);
/*
      On définit + par le tableau logique binaire suivant.
      a^i et b^i sont les i^èmes bits de op1 et op2. c_{i-1} est
      la retenue sortante du bit i-1 (qui est la même que
      la retenue entrante du bit i). r_i est le i^ème bit de
      résultat et c^i est la retenue sortante du i^ème bit.
```

```
        La variable binaire i prend les valeurs de 0 à MSB.
        c^i pour i-1 == -1 est CF pour ADC (valeur importante
        pour ADC)
        a_i   b_i   c_i-1      r_i   c_i
        0     0     0          0     0
        0     0     1          1     0
        0     1     0          1     0
        0     1     1          0     1
        1     0     0          1     0
        1     0     1          0     1
        1     1     0          0     1
        1     1     1          1     1
*/
```

Exemples :

```
                                                          OSZAPC
ADC EAX,EBX          Avant:   EAX:   01234567   EBX:   76543210   XXXXX1
                     Après:   EAX:   77777778   EBX:   76543210   000010

ADC mem32,70000000H  Avant:   Mem:   7FFFFFFF                     XXXXX0
                     Après:   Mem:   EFFFFFFF                     110010

ADC AL,AH            Avant:   AL: F9            AH: 65            XXXXX1
                     Après:   AL: 5F            AH: 65            000011
```

Exception :

Memory();

ADD

Addition

Formats :

ADD	op_1	,	op_2
	reg	,	reg
	reg	,	mem
	reg	,	imm
	mem	,	reg
	mem	,	imm

Description :

Cette instruction effectue une addition sur op_1 et op_2 et place le résultat dans op_1.

Fonctionnement :

```
op1 = op1 + op2;
OF = Carry0(MSB)^Carry0(MSB - 1);
SF = B(MSB, op1);
ZF = op1 == 0;
AF = Carry0(3);
PF = ~(B(7,op1)^B(6,op1)^...^B(0,op1));
CF = Carry0(MSB);
/*
      On définit + par le tableau logique binaire suivant.
      ai et bi sont les ièmes bits de op1 et op2. ci-1 est
      la retenue sortante du bit i-1 (qui est la même que
      la retenue entrante du bit i). ri est le ième bit de
      résultat et ci est la retenue sortante du ième bit.
      La variable binaire i prend les valeurs de 0 à MSB.
```

```
        c^i pour i-1 == -1 est 0 pour ADD.
        a_i   b_i   c_i-1      r_i   c_i
        0     0     0          0     0
        0     0     1          1     0
        0     1     0          1     0
        0     1     1          0     1
        1     0     0          1     0
        1     0     1          0     1
        1     1     0          0     1
        1     1     1          1     1
    */
```

Exemples :

```
                                                        OSZAPC
ADD EAX,EBX          Avant:   EAX:   01234567   EBX:   76543210   XXXXXX
                     Après:   EAX:   77777777   EBX:   76543210   000010

ADD mem32,70000000H  Avant:   Mem:   7FFFFFFF                     XXXXX0
                     Après:   Mem:   EFFFFFFF                     110010

ADD AL,AH            Avant:   AL: F9            AH:   65          XXXXX1
                     Après:   AL: 5E            AH:   65          000001
```

Exception :

Memory();

AND

ET logique

Formats :

AND	op_1	,	op_2
	reg	,	reg
	reg	,	mem
	reg	,	imm
	mem	,	reg
	mem	,	imm

Description :

Cette instruction effectue un ET logique entre op_1 et op_2 et range le résultat dans op_1.

Fonctionnement :

```
op1 = op1 & op2;
OF = 0;
SF = B(MSB, op1);
ZF = op1 == 0;
AF = UNDEFINED;
PF = ~(B(7,op1)^B(6,op1)^...^B(0,op1));
CF = 0;
/*
      On définit & par le tableau logique binaire suivant.
      ai et bi sont les bits de op1 et op2.
      ri est le bit de résultat.
      La variable binaire i prend les valeurs de 0 à MSB.
```

```
        ai  bi     ri
        0   0      0
        0   1      0
        1   0      0
        1   1      1
*/
```

Exemples :

```
                                                          OSZAPC
AND EAX,EBX     Avant :   EAX :   01234567    EBX :   76543210   XXXXXX
                Après :   EAX :   00000000    EBX :   76543210   001X10

AND mem8,70h    Avant :   Mem :   7F                             XXXXXX
                Après :   Mem :   70                             000X00

AND AX,DI       Avant :   AX  :    A5A5        DI :   FFFF       XXXXXX
                Après :   AX  :    A5A5        DI :   FFFF       010X10
```

Exception :

Memory();

BOUND

Vérification des limites d'un tableau

Formats :

BOUND	op_1	,	op_2
	reg16	,	mem32
	reg32	,	mem64

Description :

Cette instruction compare l'index signé d'un tableau contenu dans op_1 (op_1 doit être un registre) aux limites haute et basse de la structure de données déterminée par op_2. Si l'index de tableau n'est pas compris dans ces limites, une exception de niveau 5 sera générée.

Cette structure de données permet d'avoir deux opérandes contiguës (un mot ou un dmot), avec le premier opérande qui est la limite inférieure et le second la limite supérieure.

Fonctionnement :

```
if ((op1 < op2) || (op1 > [&op2 + sizeof(op1)])) {
      Interrupt(5);
}
```

Exemple :

```
BOUND EAX, mem64      Avant :  EAX      :  000092FD   mem :  00000000
                              [&mem+4] :  00000064
                      Après :  Interrupt(5)
```

Exceptions :

Interrupt(5);
Memory();

BSF

Recherche de bit en avant

Formats :

BSF	op_1	,	op_2
	reg16	,	reg16
	reg16	,	mem16
	reg32	,	reg32
	reg32	,	mem32

Description :

Le mot ou dmot déterminé par op_2 est parcouru de droite à gauche (du bit 0 au bit 15 ou 31) pour chercher le premier bit égal à 1. L'index sur le premier bit égal à 1 est stocké dans op_1 pendant la recherche de droite à gauche.

Si le mot ou dmot entier est nul, l'instruction lève ZF et op_1 est indéfini. Si elle trouve un bit égal à 1, elle baisse ZF.

Fonctionnement :

```
if (op2 == 0) {
      ZF = 1;
      op1 = UNDEFINED;
} else {       /* op2 n'est pas nul : recherche */
      ZF = 0;
      temp = 0;
      while (B(temp, op2) == 0) {
         temp = temp + 1;
      }
      op1 = temp;
}
OF = SF = AF = PF = CF = UNDEFINED;
```

Exemples :

```
                                                                OSZAPC
BSF EAX, EBX    Avant :   EAX :   XXXXXXXX    EBX :   76543210    XXXXXX
                Après :   EAX :   00000004    EBX :   76543210    XX0XXX

BSF SI, mem16   Avant :   SI  :   XXXX        mem :   0000        XXXXXX
                Après :   SI  :   XXXX        mem :   0000        XX1XXX
```

Exception :

Memory();

Note :

Cette instruction est une nouveauté du 80386.

BSR

Recherche de bit en arrière

Formats :

BSR	op_1	,	op_2
	reg16	,	reg16
	reg16	,	mem16
	reg32	,	reg32
	reg32	,	mem32

Description :

Le mot ou dmot déterminé par op_2 est parcouru de gauche à droite (du bit 15 ou 31 au bit 0) pour chercher le premier bit égal à 1. L'index sur le premier bit égal à 1 est stocké dans op_1 pendant la recherche de gauche à droite.

Si le mot ou dmot entier est nul, l'instruction lève ZF et op_1 est indéfini. Si elle trouve un bit égal à 1, elle baisse ZF.

Fonctionnement :

```
if (op2 == 0) {
      ZF = 1;
      op1 = UNDEFINED;
} else {       /* op2 n'est pas nul : recherche */
      ZF = 0;
      temp = Length(op2) - 1;
      while (B(temp, op2) == 0) {
         temp = temp - 1;
      }
      op1 = temp;
}
OF = SF = AF = PF = CF = UNDEFINED;
```

Exemples :

```
                                                          OSZAPC
BSR BX, CX       Avant :   BX  :   XXXX        CX :   0030      XXXXXX
                 Après :   BX  :   0005        CX :   0030      XX0XXX

BSR ESI, mem32   Avant :   ESI :   XXXXXXXX   mem :   000F0000  XXXXXX
                 Après :   ESI :   00000013   mem :   000F0000  XX0XXX
```

Exception :

Memory();

Note :

Cette instruction est une nouveauté du 80386.

BT

Test de bit

Formats :

BT	op_1	,	op_2
	reg16	,	reg16
	mem16	,	reg16
	reg16	,	imm
	mem16	,	imm
	reg32	,	reg32
	mem32	,	reg32
	reg32	,	imm
	mem32	,	imm

Description :

Cette instruction assigne le bit d'op_1 déterminé par op_2 à l'indicateur de retenue.

op_2 est considéré comme une valeur signée. Ainsi, on peut faire référence à des chaînes binaires de -32 K à 32 K - 1 et de -2 G à 2 G - 1.

Si op_1 est un registre, le bit assigné à CF est égal à op_2 modulo la taille du registre. Si op_1 est une chaîne binaire en mémoire, le mot ou dmot représentant le reste est calculé en additionnant op_2 (l'index binaire), divisé par la taille de l'opérande (16 ou 32), à l'adresse mémoire de op_1. Le bit du mot ou dmot est déterminé par op_2 modulo la taille de l'opérande (16 ou 32).

Fonctionnement :

```
offset = op2 % Length(op1);
disp = op2 / Length(op1);
if (op1 in (reg16,reg32)){
      CF = B(offset, op1);
} else {   /* opérande mémoire */
      CF = B(offset, [&op1 + disp]);
}
OF = SF = ZF = AF = PF = UNDEFINED;
```

Exemples :

```
                                                        OSZAPC
BT BX,CX      Avant :   BX  :   0A50   CX : 4218        XXXXXX
              Après :   BX  :   0A50   CX : 4218        XXXXX0

BT mem,EAX    Avant :   EAX :                 004A6B48  XXXXXX
                        [&mem+04A6B48/20] : 54BBD231
              Après :   EAX :                 004A6B48  XXXXX0
                        [&mem+04A6B48/20] : 54BBD231
```

Exception :

Memory();

Notes :

Cette instruction est une nouveauté du 80386.

Si op_1 est un opérande mémoire, le processeur peut accéder à deux ou quatre octets en mémoire commençant à :

```
&mem + sizeof(op1) * disp
```

Le programmeur doit donc veiller à ne point utiliser BT à côté de zones interdites de l'espace mémoire. En particulier, cette instruction ne doit pas être employée pour faire référence directement à des registres E/S paginés en mémoire. Dans ce cas, nous recommandons l'emploi d'instructions MOVE dans le logiciel pour charger ou ranger les registres associés à des unités paginées en mémoire, et l'utilisation du format registre de BT pour manipuler l'image de ce type de registre chargée dans le registre du processeur.

Si op_2 est une constante immédiate, seul un champ immédiat de 8 bits est pris en compte dans l'instruction. Cette constante doit être plus petite que la taille de l'opérande (32 ou 16 bits), sinon le déplacement résultant sera indéfini. Cela permet de déterminer tout déplacement dans un registre, et restreint les déplacements immédiats dans les chaînes binaires en mémoire à l'intérieur d'un dmot (ou mot) à l'emplacement mémoire précisé. Un assembleur peut admettre des déplacements immédiats plus grands dans une chaîne binaire en mémoire en plaçant les 5 bits de poids faible (4 bits pour les opérandes 16 bits) dans le champ immédiat op_2, et en ajustant le champ de déplacement par addition dans les bits de poids fort décalés pour former un déplacement d'octet.

BTC

Test et complémentation de bit

Formats :

BTC	*op*$_1$	,	*op*$_2$
	reg16	,	reg16
	mem16	,	reg16
	reg16	,	imm
	mem16	,	imm
	reg32	,	reg32
	mem32	,	reg32
	reg32	,	imm
	mem32	,	imm

Description :

Cette instruction assigne le bit d'op$_1$ déterminé par op$_2$ à l'indicateur de retenue. Puis elle complémente le même bit d'op$_1$ qui a été déterminé par op$_2$.

op$_2$ est considéré comme une valeur signée. Ainsi, on peut faire référence à des chaînes binaires de -32 K à 32 K - 1 et de -2 G à 2 G - 1.

Si op$_1$ est un registre, le bit assigné à CF (et complémenté) est op$_2$ modulo la taille du registre. Si op$_1$ est une chaîne binaire en mémoire, le mot ou dmot représentant le reste est calculé en additionnant op$_2$ (l'index binaire) divisé par la taille de l'opérande (16 ou 32) à l'adresse mémoire de op$_1$. Le bit du mot ou dmot est déterminé par op$_2$ modulo la taille de l'opérande (16 ou 32).

Fonctionnement :

```
offset = op2 % Length(op1);
disp = op2 / Length(op1);
if (op1 in (reg16,reg32)){
      CF = B(offset, op1);
      B(offset,op1) = ~B(offset,op1);
} else {   /* opérande mémoire */
      CF = B(offset, [&op1 + disp]);
      B(offset, [&op1 + disp]) = ~B(offset, [&op1 + disp]);
}
OF = SF = ZF = AF = PF = UNDEFINED;
```

Exemples :

```
                                                                OSZAPC
BTC EAX,EDX   Avant :   EAX :   0A50FF14    EDX :   00A4CCD4    XXXXXX
              Après :   EAX :   0A40FF14    EDX :   00A4CCD4    XXXXX1

BTC mem,CX    Avant :   CX              : F4B8                  XXXXXX
                      [&mem+F4B8/10]    : D231
              Après :   CX              : F4B8                  XXXXX0
                      [&mem+F4B8/10]    : D331
```

Exception :

Memory();

Notes :

Cette instruction est une nouveauté du 80386.

Si op_1 est un opérande mémoire, le processeur peut accéder à deux ou quatre octets en mémoire commençant à :

```
&mem + sizeof(op1) * disp
```

Le programmeur doit donc veiller à ne point utiliser BTC à côté de zones interdites de l'espace mémoire. En particulier, cette instruction ne doit pas être employée pour faire référence directement à des registres E/S paginés en mémoire. Dans ce cas, nous recommandons l'emploi d'instructions MOVE dans le logiciel pour charger ou ranger les registres associés à des unités paginées en mémoire, et l'utilisation du format registre de BTC pour manipuler l'image de ce type de registre chargée dans le registre du processeur.

Si op_2 est une constante immédiate, seul un champ immédiat de 8 bits est pris en compte dans l'instruction. Cette constante doit être plus petite que la taille de l'opérande (32 ou 16 bits), sinon le déplacement résultant sera indéfini. Cela permet de déterminer tout déplacement dans un registre, et restreint les déplacements immédiats dans les chaînes binaires en mémoire à l'intérieur d'un dmot (ou mot) à la place mémoire précisée. Un assembleur peut admettre des déplacements immédiats plus grands dans une chaîne binaire en mémoire en plaçant les 5 bits de poids faible (4 bits pour les opérandes 16 bits) dans le champ immédiat op_2, et en ajustant le champ de déplacement par addition dans les bits de poids fort décalés pour former un déplacement d'octet.

BTR

Test de bit et remise à zéro

Formats :

BTR	op_1	,	op_2
	reg16	,	reg16
	mem16	,	reg16
	reg16	,	imm
	mem16	,	imm
	reg32	,	reg32
	mem32	,	reg32
	reg32	,	imm
	mem32	,	imm

Description :

Cette instruction assigne le bit d'op_1 déterminé par op_2 à l'indicateur de retenue, puis remet à zéro le même bit d'op_1 qui a été déterminé par op_2.

op_2 est considéré comme une valeur signée. Ainsi, on peut faire référence à des chaînes binaires de -32 K à 32 K - 1 et de -2 G à 2 G - 1.

Si op_1 est un registre, le bit assigné à CF (et complémenté) est op_2 modulo la taille du registre. Si op_1 est une chaîne binaire en mémoire, le mot ou dmot représentant le reste est calculé en additionnant op_2 (l'index binaire) divisé par la taille de l'opérande (16 ou 32) à l'adresse mémoire de op_1. Le bit du mot ou dmot est déterminé par op_2 modulo la taille de l'opérande (16 ou 32).

Fonctionnement :

```
offset = op2 % Length(op1);
disp = op2 / Length(op1);
if (op1 in (reg16,reg32)){
      CF = B(offset, op1);
      B(offset,op1) = 0;
} else {   /* opérande mémoire */
      CF = B(offset, [&op1 + disp]);
      B(offset, [&op1 + disp]) = 0;
}
OF = SF = ZF = AF = PF = UNDEFINED;
```

Exemples :

```
                                                        OSZAPC
BTR AX,23h     Avant :   AX :   53FE                    XXXXXX
               Après :   AX :   53F6                    XXXXX1

BTR mem,ESI    Avant :   ESI                 :  532011D4     XXXXXX
                       [&mem+532011D4/20]  :  FFFFFFFF
               Après :   ESI                 :  532011D4     XXXXX1
                       [&mem+532011D4/20]  :  FFEFFFFF
```

Exception :

Memory();

Notes :

Cette instruction est une nouveauté du 80386. Voir l'exemple 4, page 553.

Si op_1 est un opérande mémoire, le processeur peut accéder à deux ou quatre octets en mémoire commençant à :

```
&mem + sizeof(op1) * disp
```

Le programmeur doit donc veiller à ne point utiliser BTR à côté de zones interdites de l'espace mémoire. En particulier, cette instruction ne doit pas être employée pour faire référence directement à des registres E/S paginés en mémoire. Dans ce cas, nous recommandons l'emploi d'instructions MOVE dans le logiciel pour charger ou ranger les registres associés à des unités paginées en mémoire, et l'utilisation du format registre de BTR pour manipuler l'image de ce type de registre chargée dans le registre du processeur.

Si op_2 est une constante immédiate, seul un champ immédiat de 8 bits est pris en compte dans l'instruction. Cette constante doit être plus petite que la taille de l'opérande (32 ou 16 bits), sinon le déplacement résultant sera indéfini. Cela permet de déterminer tout déplacement dans un registre, et restreint les déplacements immédiats dans les chaînes binaires en mémoire à l'intérieur d'un dmot (ou mot) à la place mémoire précisée. Un assembleur peut admettre des déplacements immédiats plus grands dans une chaîne binaire en mémoire en plaçant les 5 bits de poids faible (4 bits pour les opérandes 16 bits) dans le champ immédiat op_2, et en ajustant le champ de déplacement par addition dans les bits de poids fort décalés pour former un déplacement d'octet.

BTS

Test et positionnement de bit

Formats :

BTS	op_1	,	op_2
	reg16	,	reg16
	mem16	,	reg16
	reg16	,	imm
	mem16	,	imm
	reg32	,	reg32
	mem32	,	reg32
	reg32	,	imm
	mem32	,	imm

Description :

Cette instruction assigne le bit d'op_1 déterminé par op_2 à l'indicateur de retenue, puis met à 1 le même bit d'op_1 qui a été déterminé par op_2.

op_2 est considéré comme une valeur signée. Ainsi, on peut faire référence à des chaînes binaires de -32 K à 32 K - 1 et de -2 G à 2 G - 1.

Si op_1 est un registre, le bit assigné à CF (et complémenté) est op_2 modulo la taille du registre. Si op_1 est une chaîne binaire en mémoire, le mot ou dmot représentant le reste est calculé en additionnant op_2 (l'index binaire) divisé par la taille de l'opérande (16 ou 32) à l'adresse mémoire de op_1. Le bit du mot ou dmot est déterminé par op_2 modulo la taille de l'opérande (16 ou 32).

Fonctionnement :

```
offset = op2 % Length(op1);
disp = op2 / Length(op1);
if (op1 in (reg16,reg32)){
      CF = B(offset, op1);
      B(offset,op1) = 1;
} else {   /* opérande mémoire */
      CF = B(offset, [&op1 + disp]);
      B(offset, [&op1 + disp]) = 1;
}
OF = SF = ZF = AF = PF = UNDEFINED;
```

Exemples :

```
                                                    OSZAPC
BTS EAX,69h     Avant :   EAX :   0459B820          XXXXXX
                Après :   EAX :   0459BA20          XXXXX0

BTS mem16,SI    Avant :   SI              : 3014     XXXXXX
                          [&mem+3014/10]  : 0023
                Après :   SI              : 3014     XXXXX0
                          [&mem+3014/10]  :0033
```

Exception :

Memory();

Notes :

Cette instruction est une nouveauté du 80386. Voir l'exemple 4, page 553.

Si op_1 est un opérande mémoire, le processeur peut accéder à deux ou quatre octets en mémoire commençant à :

```
&mem + sizeof(op1) * disp
```

Le programmeur doit donc veiller à ne point utiliser BTS à côté de zones interdites de l'espace mémoire. En particulier, cette instruction ne doit pas être employée pour faire référence directement à des registres E/S paginés en mémoire. Dans ce cas, nous recommandons l'emploi d'instructions MOVE dans le logiciel pour charger ou ranger les registres associés à des unités paginées en mémoire, et l'utilisation du format registre de BTS pour manipuler l'image de ce type de registre chargée dans le registre du processeur.

Si op_2 est une constante immédiate, seul un champ immédiat de 8 bits est pris en compte dans l'instruction. Cette constante doit être plus petite que la taille de l'opérande (32 ou 16 bits), sinon le déplacement résultant sera indéfini. Cela permet de déterminer tout déplacement dans un registre, et restreint les déplacements immédiats dans les chaînes binaires en mémoire à l'intérieur d'un dmot (ou mot) à la place mémoire précisée. Un assembleur peut admettre des déplacements immédiats plus grands dans une chaîne binaire en mémoire en plaçant les 5 bits de poids faible (4 bits pour les opérandes 16 bits) dans le champ immédiat op_2, et en ajustant le champ de déplacement par addition dans les bits de poids fort décalés pour former un déplacement d'octet.

CALL

Appel à un sous-programme

Formats :

CALL op_1
near_label
reg32
mem32

Description :

CALL force le processeur à poursuivre l'exécution de l'instruction située à l'adresse donnée par le déplacement dans le segment code courant. Ici, nous parlerons uniquement des appels à des sous-programmes situés dans le même segment code. Nous décrirons les appels à des sous-programmes situés dans des segments différents à la partie Segments multiples.

Les déplacements intrasegments sont déterminés par un "near label" (étiquette proche), un reg32, ou un mem32, où le contenu de la mémoire mem32 fournit le déplacement.

Avant le transfert de contrôle, le pointeur courant (qui pointe donc sur la prochaine instruction exécutable) est empilé. Cette information peut être employée par l'instruction RET pour revenir à cet endroit du programme.

Notons que c'est EIP qui est sauvegardé dans la pile, et non EIP plus la longueur de l'instruction courante d'appel. Quand le 80386 traite une instruction, il incrémente le pointeur d'instruction de la longueur de l'instruction avant de l'exécuter. Ainsi, EIP peut être sauvegardé directement.

Fonctionnement :

```
if (op1 est un near_label) {
      Push(EIP);
      EIP = near_label; /*   */
} else if (op1 est un reg32) {
      Push(EIP);
      EIP = reg32;
} else if (op1 est une mem32) {
      Push(EIP);
      EIP = mem32;
}
/*
        la forme near_label est constituée d'un simple déplacement
      immédiat. Ce déplacement est relatif au pointeur courant
      d'instruction. Ainsi, on aurait :
         EIP = EIP + imm32
*/
```

Exemple :

```
CALL near_label1 ;   label1 se trouve 32 octets plus loin
          Avant :   EIP :   300   ESP :   200
          Après :   EIP :   332   ESP :   1FC   SS :[ESP] : 300
```

Exceptions :

Memory();
Stack();
CodeReference();

Note :

Voir l'exemple 3a, page 549.

CBW/CWDE

Conversion d'octet en mot

Formats :

CBW
CWDE

Description :

L'octet ou le mot signé contenu dans AL ou AX subit une extension de signe pour remplir le registre AX ou EAX. La valeur du bit 7 de AX ou du bit 15 de EAX est recopiée dans chaque bit de AH ou les 16 bits supérieurs de EAX.

Ainsi, la valeur de l'octet signé dans AL ou du mot signé dans AX devient un mot signé dans AX ou un dmot signé dans EAX.

Fonctionnement :

```
if (CodeOp == CWDE) {
      EAX = SignEx(32,16,AX);
} else {    /* CBW */
      AX = SignEx(16,8,AL);
}
```

Exemples :

```
CBW     Avant :   AX  :   XX7F
        Après :   AX  :   007F

CWDE    Avant :   EAX :   XXXXFDF3
        Après :   EAX :   FFFFFDF3
```

Exceptions :

Aucune.

Note :

CWDE est différent de CWD car CWD utilise la paire de registres DX et AX plutôt que EAX.

CLC

Mise à zéro de l'indicateur de retenue

Format :

CLC

Description :

L'indicateur de retenue est mis à zéro.

Fonctionnement :

```
CF = 0;
```

Exemple :

```
CLC      Avant :   CF :   1
         Après :   CF :   0
```

Exceptions :

Aucune.

CLD

Mise à zéro de l'indicateur de direction

Format :

CLD

Description :

L'indicateur de direction est mis à 0. Après cette opération, les instructions de chaînes incrémenteront leurs registres d'index (ESI et EDI).

Fonctionnement :

```
DF = 0;
```

Exemple :

```
CLD       Avant :   DF :   1
          Après :   DF :   0
```

Exceptions :

Aucune.

Note :

Voir l'exemple 5, page 555.

CLI

Mise à zéro de l'indicateur d'interruption

Format :

CLI

Description :

L'indicateur d'autorisation d'interruption est mis à 0. Le 80386 ignorera toute interruption à partir de la fin de l'exécution de l'instruction suivante jusqu'à ce que le bit IF soit remis à 1.

Une exception est générée si le programme ne possède pas le privilège E/S (voir le Chapitre 5).

Fonctionnement :

```
IF = 0;
```

Exemple :

```
CLI      Avant :   IF :   1
         Après :   IF :   0
```

Exception :

IOPLSensitive();

Note :

Voir l'exemple 1, page 842.

CMC

Complémentation du bit de retenue

Format :

CMC

Description :

L'indicateur de retenue est complémenté.

Fonctionnement :

```
CF = ~CF;
```

Exemples :

```
CMC      Avant :   CF :   1
         Après :   CF :   0

CMC      Avant :   CF :   0
         Après :   CF :   1
```

Exceptions :

Aucune.

CMP

Comparaison

Formats :

CMP	op_1	,	op_2
	reg	,	reg
	reg	,	mem
	reg	,	imm
	mem	,	reg
	mem	,	imm

Description :

Cette instruction soustrait op_2 à op_1 mais ne sauvegarde pas le résultat. Seuls les indicateurs sont modifiés. CMP est souvent suivi d'une instruction de saut conditionnel ou de positionnement d'octet.

Fonctionnement :

```
temp = op1 - op2;
/* - est défini par l'instruction SUB */

OF = Borrow0(MSB)^Borrow0(MSB-1);
SF = B(MSB,temp);
ZF = temp == 0;
AF = Borrow0(3);
PF = ~(B(7,temp)^B(6,temp)^...^B(0,temp));
CF = Borrow0(MSB);
```

Exemples :

```
                                                                OSZAPC
CMP EAX,EBX      Avant :   EAX :   01234567    EBX :   01234568   XXXXX
                 Après :   EAX :   01234567    EBX :   01234568   10111
JZ EQUAL         ; Le saut n'est pas pris en compte.

CMP AX,mem16     Avant :   AX  :   0000      mem16 :   0000       XXXXXX
                 Après :   AX  :   0000      mem16 :   0000       001010
JZ EQUAL         ; Le saut EST pris en compte.
```

Exception :

Memory();

Note :

Voir le Chapitre 4 pour des exemples, comme l'exemple 5, page 555.

CMPS/CMPSB/CMPSW/CMPSD

Comparaison de chaînes

Formats :

```
CMPSB
CMPSB
CMPSW
CMPSD  mem1   ,  mem2
       mem8   ,  mem8
       mem16  ,  mem16
       mem32  ,  mem32
```

Fonctionnement :

CMPS soustrait ES:[EDI] de [ESI] et ne sauvegarde pas le résultat, puisque seuls les indicateurs sont modifiés. Les registres ESI et EDI sont corrigés pour pointer sur le prochain élément de la chaîne. Cette correction est basée sur l'indicateur de direction (DF) et la longueur des opérandes (8, 16, ou 32 bits) indiquée par mem_1, mem_2, ou par les CODEOP eux-mêmes (CMPSB, CMPSW et CMPSD). Si l'indicateur DF est à 0 (resp. 1), les registres sont corrigés par 1, 2 ou 4 (resp. -1, -2, -4).

REPE (REPZ), ou REPNE (REPNZ) peuvent précéder l'opération CMPS. Si REPE (resp. REPNE) précède l'instruction CMPS, cette dernière se répète tant que ECX n'est pas nul et que les éléments de la chaîne sont égaux (ZF == 1) (resp. différents (ZF == 0)). Ainsi, on utilise CMPS pour trouver des similitudes ou des différences entre des chaînes. Se reporter à REPE et REPNE pour de plus amples détails sur les préfixes.

L'assembleur trouvera dans mem_1 et mem_2 des renseignements pour la longueur du traitement. Il considère toujours les chaînes dans [ESI] et ES:[EDI]. Il est possible que CMPS inclue un préfixe de recouvrement de

segment qui affecte le déplacement associé à mem_1 ([ESI]). mem_2 provient toujours de ES:[EDI].

Fonctionnement :

```
if (CODEOP == CMPSB) {
      delta = 1;
      temp = (octet)  DS:[ESI] - (octet)  ES:[EDI];
} else if (CODEOP == CMPSB) {
      delta = 2;
      temp = (mot)    DS:[ESI] - (mot)    ES:[EDI];
} else if (CODEOP == CMPSB) {
      delta = 4;
      temp = (dmot)   DS:[ESI] - (dmot)   ES:[EDI];
} else {      /* CMPS */
      /* Notons que le défaut pour SegReg(mem1) est DS */
      delta = Length(mem1);
      temp = SegReg(mem1):[ESI] - ES:[EDI]
}
/* - est defini par l'instruction SUB */
OF = Borrow0(MSB)^Borrow(MSB-1);
SF = B(MSB,temp);
ZF = temp == 0;
AF = Borrow0(3);
PF = ~(B(7,temp)^B(6,temp)^...^B(0,temp));
CF = Borrow0(MSB);
if (DF == 1) {
      delta = - delta;
}
ESI = ESI + delta;
EDI = EDI + delta;
```

Exemples :

```
                                                                    OSZAPC
CMPSB   Avant :   ESI          :   0008   EDI :   0016   [ESI]    :   67   XXXXXX
                  ES :[EDI]    :     65   DF  :   0
        Après :   ESI          :   0009   EDI :   0017   [ESI-1] :   67   000010
                  ES :[EDI-1]  :     65   DF  :   0

REPE CMPS S1,S2       ; Notons que S1 et S2 sont des Chaînes 16 bits
                      ; [ESI] pointe vers S1, ES :[EDI] pointe vers S2.
                           OSZAPC
      Avant :   ESI     :   0008   EDI     :   0016   [ESI]    : 0023   XXXXXX
                [ESI-2] :   7923   [ESI-4] :   7214   [ESI-6] : AA6D
                [EDI]   :   0023   [EDI-2] :   7923   [EDI-4] : 8215
                [EDI-6] :   AA6D   ECX     :      4   DF      :    1
      Après :   ESI     :   0002   EDI     :   0010   [ESI+6] : 0023   001010
                [ESI+4] :   7923   [ESI+2] :   7214   [ESI]   : AA6D
                [EDI+6] :   0023   [EDI+4] :   7923   [EDI+2] : 8215
                [EDI]   :   AA6D   ECX     :      1   DF      :    1
```

Exception :

Memory();

Note :

Voir l'exemple 5, page 555.

CWD/CDQ

Conversion de mot en dmot

Formats :

CWD
CDQ

Description :

Le mot signé contenu dans AX ou le dmot contenu dans EAX subit une extension de signe pour remplir les registres DX ou EDX. La valeur du bit 15 ou du bit 31 de AX ou de EAX est copié dans tous les bits de DX ou EDX.

Ainsi, la valeur du mot signé contenu dans AX devient un dmot signé dans DX:AX. Pour des opérandes 32 bits, le dmot signé contenu dans EAX devient un qmot signé dans EDX:EAX.

Fonctionnement :

```
if (CODEOP == CDQ) {
      if (B(31,EAX) == 1) {
         EDX = FFFFFFFFh;
      } else {
         EDX = 00000000h;
      }
} else {
      if (B(15,AX) == 1) {
         DX = FFFFh;
      } else {
         DX = 0000h;
      }
}
```

Exemples :

```
CWD     Avant :   AX :   7615   DX :   XXXX
        Après :   AX :   7615   DX :   0000
CWD     Avant :   AX :   F103   DX :   XXXX
        Après :   AX :   F103   DX :   FFFF
```

Exceptions :

Aucune.

Note :

CWD est différent de CWDE car CWDE utilise le registre EAX plutôt que la paire de registres DX et AX.

DAA

Ajustement décimal après addition

Format :

DAA

Description :

On ne devrait employer DAA qu'après une instruction ADD qui a travaillé sur deux nombres BCD compactés (un octet compacté) et qui a placé le résultat dans AL.

DAA va prendre le contenu du registre AL et le convertir en un nombre BCD à 2 chiffres compactés (le résultat est donc dans AL).

Si le demi-octet inférieur de AL est > 9 ou si AF est levé, alors l'instruction ajoute 6 à AL et positionne AF. Sinon, AF est mis à zéro. Si le demi-octet supérieur de AL est >9 F ou si CF est levé, alors elle ajoute 60h à AL et positionne CF. Sinon, CF est mis à zéro.

Fonctionnement :

```
if (((AL & 0Fh) > 9) || AF ) {
      AL = AL + 6;
      CF = CF | Carry0(7);
      AF = 1;
} else {
      AF = 0;
}
if ((AL > 9Fh) || CF) {
      AL = AL + 60h;
      CF = 1;
```

```
} else {
      CF == 0;
}
OF = UNDEFINED;
SF = B(7,AL);
ZF = (AL == 0);
PF = ~(B(7,AL)^B(6,AL)^...^B(0,AL));
```

Exemples :

							OSZAPC
ADD AL,BL	Avant :	AL :	77	BL :	13		XXXXXX
	Après :	AL :	8A	BL :	13		110000
DAA	Avant :	AL :	8A	BL :	13		110000
	Après :	AL :	90	BL :	13		X10110
ADD AL,BL	Avant :	AL :	79	BL :	35		XXXXXX
	Après :	AL :	AE	BL :	35		110000
DAA	Avant :	AL :	AE	BL :	35		110000
	Après :	AL :	14	BL :	35		X00111

Exceptions :

Aucune.

DAS

Ajustement décimal après soustraction

Format :

DAS

Description :

On ne devrait employer DAS qu'après une instruction SUB qui a travaillé sur deux nombres BCD compactés (un octet compacté) et qui a placé le résultat dans AL.

DAS va prendre le contenu du registre AL et le convertir en un nombre BCD à 2 chiffres compactés (le résultat est donc dans AL).

Si le demi-octet inférieur de AL est > 9 ou si AF est levé, alors l'instruction soustrait 6 de AL et positionne AF. Sinon, AF est mis à zéro. Si le demi-octet supérieur de AL est > 9F ou si CF est levé, alors elle soustrait 60h de AL et positionne CF. Sinon, CF est mis à zéro.

Fonctionnement :

```
if (((AL & 0Fh) > 9) || AF ) {
      AL = AL - 6;
      CF = CF | Borrow0(7);
      AF = 1;
} else {
      AF = 0;
}
if ((AL > 9Fh) || CF) {
      AL = AL + 60h;
      CF = 1;
```

```
} else {
      CF == 0;
}
OF = UNDEFINED;
SF = B(7,AL);
ZF = (AL == 0);
PF = ~(B(7,AL)^B(6,AL)^...^B(0,AL));
```

Exemples :

							OSZAPC
SUB AL,BL	Avant :	AL :	71	BL :	13		XXXXXX
	Après :	AL :	5E	BL :	13		000100
DAS	Avant :	AL :	5E	BL :	13		000100
	Après :	AL :	58	BL :	13		X00100
SUB AL,BL	Avant :	AL :	35	BL :	47		XXXXXX
	Après :	AL :	EE	BL :	47		010111
DAS	Avant :	AL :	EE	BL :	47		010111
	Après :	AL :	88	BL :	47		X10111

Exceptions :

Aucune.

DEC

Décrémentation

Formats :

DEC op_1
mem
reg

Description :

L'instruction soustrait 1 de op_1, et le résultat est replacé dans op_1.

Fonctionnement :

```
op1 = op1 - 1;
/* - est défini par l'instruction SUB   */

OF = Carry0(MSB)^Carry0(MSB-1);
SF = B(MSB, op1);
ZF = op1 == 0;
AF = Carry0(3);
PF = ~(B(7,op1)^B(6,op1)^...^B(0,op1));
CF = CF;
```

Exemples :

```
                                                OSZAPC
DEC EAX        Avant :   EAX :   01234567       XXXXXX
               Après :   EAX :   01234567       00001U

DEC mem16      Avant :   Mem :   0000           XXXXXX
               Après :   Mem :   FFFF           01011U
```

Exception :

Memory();

Note :

L'indicateur de retenue *n'est pas* affecté par l'instruction DEC. On doit employer l'instruction SUB avec un opérande de 1 si on désire modifier l'indicateur de retenue.

DIV

Division non signée

Formats :

DIV	op_1
	reg
	mem

Description :

Cette instruction effectue une division non signée avec reste sur un opérande donné (op_1) et un registre implicite. Ce dernier est lié à la longueur de l'opérande (8-AX, 16-AX, ou 32-EAX:EDX).

L'instruction place le quotient dans AL, AX, ou EAX pour des opérandes 8, 16, 32 bits. Le reste se trouve dans AH, DX, ou EDX pour des opérandes 8, 16, 32 bits. Ainsi, le diviseur est déterminé par op_1, et le dividende, le quotient et le reste sont toujours dans des registres implicites. Le tableau suivant résume les registres et les opérandes employés :

Longueur(op1)	Dividende	Diviseur	Quotient	Reste
8	AX	op_1	AL	AH
16	DX:AX	op_1	AX	DX
32	EDX:EAX	op_1	EAX	EDX

Les quotients non entiers sont tronqués à 0. Les restes sont toujours plus petits que les dividendes.

Si le quotient ne tient pas dans un des registres quotient (AL, AX, EAX), une erreur du type division par zéro est générée, c'est-à-dire l'interruption 0 du 80386. Le quotient et le reste sont alors indéfinis.

Fonctionnement :

```
if (op1 = 0) Interrupt(0);
if (Length(op1) == 8){
      if ((AX / op1) > FFh) Interrupt(0);
      temp = AX;
      AL = temp / op1
      AH = temp % op1
} else if (Length(op1) == 16){
      if ((ConCat(32,DX,16,AX) / op1) > FFFFh) Interrupt(0);
      temp = AX;
      AX = ConCat(32,DX,16,temp) / op1;
      DX = ConCat(32,DX,16,temp) % op1;
} else { /* Length(op1) == 32 */
      if ((ConCat(64,EDX,32,EAX) / op1) > FFFFFFFFh) Interrupt(0);
      temp = EAX;
      EAX = ConCat(64,EDX,32,temp) / op1;
      EDX = ConCat(64,EDX,32,temp) % op1;
}
OF = SF = ZF = AF = PF = CF = UNDEFINED
/*
      / est défini comme la division non signée.
A la fois AX ou ConCat(DX,AX) ou ConCat(EDX,EAX)
et op1 sont considérés comme des nombres entiers
non signés.

      %, de même, est défini comme le modulo non signé
*/
```

Exemples :

							OSZAPC
DIV EBX	Avant :	EDX :	76543210	EAX :	01234567		XXXXXX
		EBX :	89014573				
	Après :	EDX :	34DCEE8F	EAX :	DD1A57C8		XXXXXX
		EBX :	89014573				

```
DIV mem8   Avant :   AH :   78   AL :   21   mem :   5C       XXXXXX
           Après :   Interrupt(0) ;
                     AH :   XX   AL :   XX   mem :   5C       XXXXXX
```

Exceptions :

Memory();
Interrupt(0);

ENTER

Création du cadre de pile

Formats :

ENTER	op_1	,	op_2
	imm16	,	imm8

Description :

ENTER génère un cadre de pile. La plupart des langages de haut niveau le font à chaque appel de procédure.

op_2 détermine le niveau d'imbrication de la routine. Ce dernier détermine le nombre de pointeurs de cadre de pile qui sont copiés depuis le cadre de pile courant vers le nouveau qui vient d'être construit par l'instruction ENTER. On emploie EBP pour copier les pointeurs depuis le cadre de la pile courant vers le nouveau. EBP pointe vers le pointeur du nouveau cadre à la fin de l'instruction ENTER.

L'espace pile est automatiquement alloué pour un nombre d'octets de variables locales déterminé par op_1.

Fonctionnement :

```
level = op2 % 32;
Push(EBP);
frame = ESP;
if (level > 0) {
      while ((--level > 0){
         EBP = EBP -4;
         Push([EBP]);
      }
```

```
        Push(frame);
    }
    EBP = frame;
    ESP = ESP - Concat(32,0000h,16,op1);
```

Exemple :

```
ENTER 0Ch,4
      Avant :   EBP          :   00000F38   SS  : [EBP]    :   00000F4C
                SS :[EBP-4]  :   0000100B   SS  : [EBP-8]  :   000010A4
                SS :[EBP-12] :   00001102   ESP            :   00000F04
      Après :   SS :[ESP+28] :   00000F38   SS  : [ESP+24] :   0000100B
                SS :[ESP+20] :   00000F00   EBP            :   00000F00
                ESP          :   00000EE4
```

Exception :

Stack();

Note :

Voir l'exemple 3b, page 551.

IDIV

Division signée

Formats :

IDIV op_1
reg
mem

Description :

Cette instruction effectue une division signée avec reste sur un opérande donné (op_1) et un registre implicite. Ce dernier est lié à la longueur de l'opérande (8-AX, 16-AX, ou 32-EAX:EDX).

L'instruction place le quotient dans AL, AX, ou EAX pour des opérandes 8, 16, 32 bits. Le reste se trouve dans AH, DX, ou EDX pour des opérandes 8, 16, 32 bits. Le tableau suivant résume les registres et les opérandes employés :

Longueur(op1)	Dividende	Diviseur	Quotient	Reste
8	AX	op_1	AL	AH
16	DX:AX	op_1	AX	DX
32	EDX:EAX	op_1	EAX	EDX

Les quotients non entiers sont tronqués à 0. Les restes sont toujours du même signe que les dividendes et sont toujours plus petits, en valeur absolue, que les diviseurs. Le tableau suivant donne toutes les combinaisons de signes possibles du dividende et du quotient, et les quotients et les restes obtenus, pour l'opération : 4 divisé par 3.

Dividende	op1	Quotient	Reste
+4	+3	+1	+1
+4	-3	-1	+1
-4	+3	-1	-1
-4	-3	+1	-1

La division obéit toujours à l'équation :

```
dividende = quotient * diviseur + reste
```

Si $op_1 = 0$ ou si le quotient ne tient pas dans un des registres quotient (AL, AX, EAX), une erreur du type division par zéro est générée, c'est-à-dire l'interruption 0 du 80386. Le quotient et le reste sont alors indéfinis.

Fonctionnement :

```
if (op1 = 0) Interrupt(0);
if (Length(op1) == 8){
      if ( ((AX > 0) && (AX / op1) > 7Fh) ||
         ((AX < 0) && (AX / op1) < 80h) )
         Interrupt(0);
      temp = AX;
      AL = temp / op1
      AH = temp % op1
} else if (Length(op1) == 16){
      if ( ((ConCat(32,DX,16,AX) > 0) &&
         (ConCat(32,DX,16,AX) / op1) > 7FFFh) ||
         ((ConCat(32,DX,16,AX) < 0) &&
         (ConCat(32,DX,16,AX) / op1) < 8000h) )
         Interrupt(0);
      temp = AX;
      AX = ConCat(32,DX,16,temp) / op1;
      DX = ConCat(32,DX,16,temp) % op1;
```

```
} else { /* Length(op1) == 32 */
      if ( ((ConCat(64,EDX,32,EAX) > 0) &&
         (ConCat(64,EDX,32,EAX) / op1) > 7FFFFFFFh) ||
         ((ConCat(64,EDX,32,EAX) < 0) &&
         (ConCat(64,EDX,32,EAX) / op1) < 80000000h) )
         Interrupt(0);
      temp = EAX;
      EAX = ConCat(64,EDX,32,temp) / op1;
      EDX = ConCat(64,EDX,32,temp) % op1;
}
OF = SF = ZF = AF = PF = CF = UNDEFINED
/*
      / est défini comme la division signée.
A la fois AX ou ConCat(DX,AX) ou ConCat(EDX,EAX)
et op1 sont considérés comme des nombres entiers
signés.

      %, de même, est défini comme le modulo signé
*/
```

Exemples :

								OSZAPC
IDIV BX	Avant :	DX :	3210	AX :	4567			XXXXXX
		BX :	6773					
	Après :	DX :	496E	AX :	7BE3			XXXXXX
		BX :	6773					
IDIV mem8	Avant :	AH :	D8	AL :	D7	mem :	AD	XXXXXX
	Après :	AH :	BF	AL :	78	mem :	AD	XXXXXX
IDIV BL	Avant :	AH :	87	AL :	B2	BL :	19	XXXXXX
	Après :	Interrupt(0);						
		AH :	XX	AL :	XX	BL :	19	XXXXXX

Exceptions :

Memory();
Interrupt(0);

Note :

Voir l'exemple 1 et description, page 544.

IMUL

Multiplication signée

Formats :

IMUL	op_1				
	reg				
	mem				
IMUL	op_1	,	op_2		
	reg16	,	reg16		
	reg16	,	mem16		
	reg16	,	imm		
	reg32	,	reg32		
	reg32	,	mem32		
	reg32	,	imm		
IMUL	op_1	,	op_2	,	op_3
	reg16	,	reg16	,	imm
	reg16	,	mem16	,	imm
	reg32	,	reg32	,	imm
	reg32	,	mem32	,	imm

Description :

Si IMUL n'a qu'un opérande, le second est implicitement pris dans AL, AX, ou EAX. L'instruction range les résultats signés 16, 32, 64 bits respectivement dans AX, DX:AX, EDX:EAX. Si AH, DX, ou EDX ne sont que des extensions de signe de AL, AX, EAX, alors CF et OF sont mis à zéro. Sinon, ils sont positionnés à 1.

Quand on a deux opérandes, IMUL range le produit op_1 * op_2 dans op_1. Si op_1 peut contenir le résultat, alors CF et OF sont positionnés à 0. Sinon, ils sont mis à 1.

Quand on a trois opérandes, l'instruction range le produit $op_2 * op_3$ dans op_1. Si op_1 peut contenir le résultat, alors les indicateurs CF et OF sont positionnés à 0. Sinon, ils sont mis à 1.

Fonctionnement :

```
if (NUMOPS == 1){
      if (Length(op1 == 8){
         AX = AL * op1;
         if ((AH == 00h) || (AH == FFh)){
            CF = 0; OF = 0;
         } else {
            CF = 1; OF = 1;
         }
      } else if (Length(op1) == 16){
         DX = (AX * op1) >> 16;
         AX = (AX * op1) & 0FFFFh;
         if ((DX == 0000h) || (DX == FFFFh)){
            CF = 0; OF = 0;
         } else {
            CF = 1; OF = 1;
      } else {       /* Length(op1) == 32 */
         EDX = (EAX * op1) >> 32;
         EAX = (EAX * op1) & 0FFFFFFFFh;
         if ((EDX == 00000000h) || (EDX == FFFFFFFFh)){
            CF = 0; OF = 0;
         } else {
            CF = 1; OF = 1;
         }
      }
} else if (NUMOPS == 2){
      doubleTemp = op1 * op2;
      op1 = op1 * op2;
      if (doubleTemp != op1){
         CF = 1; OF = 1;
      } else {
         CF = 0; OF = 0;
      }
} else {       /* NUMOPS == 3 */
```

```
            op1 = op2 * op3;
            doubleTemp = op2 * op3;
            if (doubleTemp != op1){
               CF = 1; OF = 1;
            } else {
               CF = 0; OF = 0;
            }
         }
         SF = ZF = AF = PF = UNDEFINED;
         /* * est défini comme la multiplication signée */
```

Exemples :

```
                                                                OSZAPC
IMUL BX          Avant :   DX  :   XXXX   AX :   1862           XXXXXX
                           BX  :   8536
                 Après :   DX  :   F44E   AX :   0EAC           1XXXX1
                           BX  :   8536

IMUL ECX,mem32   Avant :   ECX :   00015792   mem32 :   00052692   XXXXXX
                 Après :   ECX :   E99D9D44   mem32 :   00052692   1XXXX1

IMUL CX,mem16,8  Avant :   CX  :   XXXX   mem16 :   002A         XXXXXX
        Après :            CX  :   0150   mem16 :   002A         0XXXX0
```

Exception :

Memory();

Note :

Voir les exemples 3a et 3b, pages 548 et 551.

IN

Entrée E/S

Formats :

IN	op_1	,	op_2
	AL	,	imm8
	AX	,	imm8
	EAX	,	imm8
	AL	,	DX
	AX	,	DX
	EAX	,	DX

Description :

IN transfère une donnée octet, mot, ou dmot d'un port déterminé respectivement vers AL, AX, ou EAX.

Les ports E/S se trouvent dans l'intervalle 0,64 K-1. Une instruction peut accéder à un port de deux façons : par un octet immédiat ou indirectement par le registre DX.

La première solution (octet immédiat) permet d'accéder aux ports 0 à 255. Les bits de poids fort de l'adresse du port sont donc toujours nuls dans ce cas. La seconde solution (registre DX) permet d'accéder à tous les ports.

Une exception est générée si la tâche courante ne dispose pas du privilège E/S. Se reporter au Chapitre 6 pour plus de précisions sur les niveaux de privilège.

Fonctionnement :

```
op1 = port(op2);
```

Exemples :

```
IN AL,20h       Avant :   AL :   XX   port(20) :   4A
                Après :   AL :   4A   port(20) :   XX

IN EAX,DX       Avant :   EAX :   XXXXXXXX   DX :   4680
                          port(4680)             :   FFA78201
                Après :   EAX :   FFA78201   DX :   4680
                          port(4680)             :   XXXXXXXX
```

Exception :

AccessIO();

INC

Incrémentation

Formats :

INC	op_1
	mem
	reg

Description :

L'instruction ajoute 1 à op_1, et place le résultat dans op_1. Tous les indicateurs sauf CF se positionnent en accord avec le résultat.

Fonctionnement :

```
op1 = op1 + 1;
/* + est défini par l'instruction ADD   */

OF = Carry0(MSB)^Carry0(MSB-1);
SF = B(MSB, op1);
ZF = op1 == 0;
AF = Carry0(3);
PF = ~(B(7,op1)^B(6,op1)^...^B(0,op1));
CF = CF;
```

Exemples :

```
                                          OSZAPC
INC AX       Avant :   AX  :   22D9       XXXXXX
             Après :   AX  :   22DA       00000U

INC mem32    Avant :   Mem :   05621340   XXXXXX
             Après :   Mem :   05621341   00001U
```

Exception :

Memory();

Note :

L'indicateur de retenue *n'est pas* affecté par l'instruction INC. On doit employer l'instruction ADD avec un opérande de 1 si on désire modifier l'indicateur de retenue.

INS/INSB/INSW/INSD

Entrée E/S d'une chaîne

Formats:

INSB			
INSW			
INSD			
INS	mem_1	,	reg_2
	mem8	,	DX
	mem16	,	DX
	mem32	,	DX

Description :

Ces instructions transfèrent un, deux ou quatre octets de données d'un port déterminé par DX vers ES:[EDI]. Après ce transfert, EDI pointe vers l'emplacement de la chaîne suivante. EDI est corrigé de 1, 2, ou 4 (resp. -1, -2, ou -4) si DF = 0 (resp. DF = 1).

Le registre DX doit déterminer quel port est utilisé. Un opérande immédiat est interdit dans ce cas.

Notez que l'emplacement mémoire est toujours déterminé par ES:[EDI]. INS n'autorise pas les recouvrements de segments. mem_1 donne uniquement une indication sur la longueur de l'opérande.

Un préfixe REP peut précéder l'instruction INS. Dans ce cas, on peut transférer ECX octets, mots ou dmots. Nous décrirons REP ultérieurement.

Une exception est générée si la tâche courante n'a pas un privilège E/S suffisant (voir le Chapitre 5).

Fonctionnement :

```
if (CODEOP == INSB){
      ES:[EDI] = (byte) port(DX);
      delta = 1;
} else if (CODEOP == INSW) {
      ES:[EDI] = (word) port(DX);
      delta = 2;
} else if (CODEOP == INSD) {
      ES:[EDI] = (dword) port(DX);
      delta = 4;
} else {      /* INS */
      ES:[EDI] = (Length(mem1)) port(DX);
      delta = Length(mem1) / 8;
}
if (DF == 1) {
      delta = -delta;
}
EDI = EDI + delta;
```

Exemples :

```
INSD   Avant :   EDI       :   00000052   ES : [EDI]   :   XXXXXXXX
                 DX        :   004C       DF :   0
                 port(DX)  :   FFFFD832
       Après :   EDI       :   00000056   ES :[EDI-4]  :   FFFFD832
                 DX        :   004C       DF :   0
                 port(DX)  :   FFFFD832

REP INS S1,DX ;      S1 est une chaîne 16 bits
      Avant :   EDI           :   0374   ES :[EDI]    :   XXXX
                ES : [EDI-2]  :   XXXX   ES :[EDI-4]  :   XXXX
                DX            :   4220   DF           :   1
                port(DX,1)    :   6000   port(DX,2)   :   0455
                port(DX,3)    :   6001   ECX          :   2
```

```
Après :   EDI          :   0370   ES :[EDI+4] :   6000
          ES : [EDI+2] :   0455   ES :[EDI]   :   XXXX
          DX           :   4220   DF          :   1
          port(DX,1)   :   6000   port(DX,2)  :   0455
          port(DX,3)   :   6001   ECX         :   0
```

Exceptions :

Memory();
AccessIO();

Jcc

Instructions de saut conditionnel

Formats :

Jcc near_label (branchement court)

Où le code de condition (cc) est un des suivants :

Mnémoniques	Condition	Description
JB/JNAE/JC	CF==1	Saut si inférieur ou si retenue
JBE/JNA	CF==1 || ZF==1	Saut si inférieur ou égal
JCXZ	CX==0	Saut si CX==0
JE/JZ	ZF==1	Saut si égal ou si zéro
JECXZ	ECX==0	Saut si ECX==0
JL/JNGE	SF!=OF	Saut si plus petit
JLE/JNG	SF!=OF || ZF==1	Saut si plus petit ou égal
JNB/JAE/JNC	CF==0	Saut si supérieur ou égal ou si pas de retenue
JNBE/JA	CF==0 && ZF==0	Saut si supérieur
JNE/JNZ	ZF==0	Saut si différent ou pas nul
JNL/JGE	SF==OF	Saut si plus grand ou égal
JNLE/JG	ZF==0 && SF==OF	Saut si plus grand
JNO	OF==0	Saut si pas de débordement
JNP/JPO	PF==0	Saut si pas parité
JNS	SF==0	Saut si pas signe
JO	OF==1	Saut si débordement
JP/JPE	PF==1	Saut si parité
JS	SF==1	Saut si signe

Notez que *plus petit* et *plus grand* se rapportent à des comparaisons sur des entiers signés, tandis que *supérieur* et *inférieur* traitent de comparaisons sur des entiers non signés.

Description :

Les indicateurs sont testés pour les conditions décrites ci-dessus. Si les indicateurs remplissent ces conditions, le branchement s'effectue à l'étiquette déterminée dans le segment code courant (la destination du saut doit se trouver dans le même segment). Sinon, le processeur passe à l'instruction suivante.

On doit avoir positionné les indicateurs par n'importe quel moyen avant le saut conditionnel (cependant, il n'est pas nécessaire que cela soit effectué par l'instruction précédente).

L'assembleur accepte de multiples mnémoniques qui facilitent l'interprétation des indicateurs. Par exemple, JA (saut si supérieur) et JNBE (saut si pas inférieur ou égal) sont synonymes. L'assembleur accepte les deux termes.

Fonctionnement :

```
if (CodeCondition) {
      EIP =  Offset(near_label); /*   */
}
/*
        L'étiquette proche (near_label) est
      codée par un déplacement immédiat, relatif
      au pointeur d'instruction courant.
      Ainsi, EIP devient :
         EIP = EIP + imm;
      Le déplacement immédiat peut être de
      8 bits ou 32 bits.
*/
```

Exemples :

```
                          OSZAPC
JO     near_label1   ; label1 se trouve 68 octets plus loin
          Avant :   EIP :   00000300      1XXXXX
          Après :   EIP :   00000368      UUUUUU

JAE    near_label1   ; label1 se trouve 537 octets avant
          Avant :   EIP :   00000300      XXXXX1
          Après :   EIP :   00000300      UUUUUU
       ; l'instruction suivante sera alors
       ; traitée, le registre EIP chargé
       ; avec sa nouvelle valeur et l'instruction
       ; exécutée.
```

Exception :

CodeReference();

Notes :

Voir l'exemple 10, page 571.

JMP

Saut

Formats :

JMP op_1
near_label
reg32
mem32

Description :

Cette instruction force le processeur à poursuivre l'exécution à l'adresse associée au déplacement précisé dans l'opérande, et cela dans le segment code courant.

Une étiquette proche (near_label), un reg32, ou une mem32 peuvent déterminer le déplacement intrasegment associé à ce saut. L'opérande du type near_label détermine un saut direct à l'étiquette donnée. Les opérandes reg32 et mem32 donnent des sauts indirects, où le déplacement est pris dans le registre ou l'emplacement mémoire choisis.

Notez que l'instruction JMP ne sauvegarde aucune information, alors que CALL le fait.

Fonctionnement :

```
if (op1 est un near_label){
      EIP = Offset(near_label);  /*   */
} else if (op1 est un reg32){
      EIP = reg32;
} else if (op1 est une mem32){
```

```
        EIP = mem32;
}
/*
          L'étiquette proche (near_label) est
        codée par un déplacement immédiat, relatif
        au pointeur d'instruction courant.
        Ainsi, EIP devient :
           EIP = EIP + imm;
        Le déplacement immédiat peut être de
        8 bits ou 32 bits.
*/
```

Exemples :

```
JMP near_label1          ; label1 se trouve 42h octets plus loin
          Avant :   EIP :   300
          Après :   EIP :   342

JMP CASETABLE[EAX*4]     ; CASETABLE est une table de branchement
                         ; EAX*4 fournit un index dans cette
                         ; table
          Avant :   EIP :   5022AC
                    CASETABLE[EAX*4] :   60AA40
          Après :   EIP :   60AA40
```

Exceptions :

Memory();
CodeReference();

Note :

Voir aussi la forme multisegment de l'instruction JMP et l'exemple 2, page 546.

LAHF

Chargement des indicateurs dans AH

Format :

LAHF

Description :

LAHF transfère l'octet de poids faible du mot d'état dans AH. Les indicateurs sont (du bit 7 au bit 0) : signe, zéro, valeur 0, retenue auxiliaire, valeur 0, parité, valeur 1 et retenue. Le dessin suivant montre les 8 bits de poids faible du registre EFLAGS.

7	6	5	4	3	2	1	0
S	Z		A		P		C
F	F	0	F	0	F	1	F

Fonctionnement :

```
AH = (EFLAGS & 0FFh);
```

Exemple :

```
                                        OSZAPC
LAHF    Avant :   AH :   XXXXXXXXb    X01110
        Après :   AH :   01010110b    U01110
```

Exceptions :

Aucune.

LEA

Chargement de l'adresse effective

Formats :

LEA	op_1	,	op_2
	reg32	,	mem
	reg16	,	mem

Description :

L'instruction extrait la partie déplacement de l'adresse (et non la valeur de cette adresse) de op_2 et la place dans op_1. op_2 doit être une adresse mémoire. op_1 est toujours un registre. Si on précise un registre 16 bits, les 16 bits de poids faible du déplacement sont placés dans ce registre (ce n'est pas très utile !).

L'adresse mémoire est formée conformément à la description du Chapitre 2.

Fonctionnement :

```
op1 = &op2;
```

Exemples :

```
LEA AX,[EBP+20h]        Avant :    AX :   XXXX          EBP :   FFFA0246
                        Après :    AX :   0266          EBP :   FFFA0246
```

```
LEA ECX,[EAX*2+14Ch]   Avant :   EAX :   0548901A     ECX :   XXXXXXXX
                       Après :   EAX :   0548901A     ECX :   0A912180
```

Exceptions :

Aucune.

Notes :

On utilise LEA pour calculer des adresses ainsi que des déplacements dans un tableau multidimensionnel. C'est aussi une instruction arithmétique puissante. Elle permet d'effectuer une addition sur trois registres généraux :

```
r1 = r2 + r3 + constante
```

La mise à l'échelle (voir le Chapitre 2) peut être employée pour ajouter de petits coefficients multiplicateurs.

LEAVE

Abandon de la procédure

Format :

LEAVE

Description :

LEAVE détruit le cadre de la pile qui a été créé par l'instruction associée ENTER. L'espace pile est récupéré en assignant EBP (le pointeur du cadre) à ESP. L'ancien pointeur de cadre est dépilé dans EBP pour positionner le pointeur de cadre pour la procédure appelante.

On emploie généralement une instruction RET pour compléter le transfert du contrôle vers la procédure appelante. RET suit LEAVE et détruit tout paramètre empilé pour la procédure existante.

Fonctionnement :

```
ESP = EBP;
EBP = Pop();
```

Exemple :

```
LEAVE   Avant :   EBP :   00000F00   SS :[EBP] :   0000F38
                  ESP :   00000EB4
        Après :   EBP :   00000F38
                  ESP :   00000F04
```

Exception :

Stack();

Note :

Voir l'exemple 3a, page 547.

LOCK

Verrouillage du bus

Format :

LOCK *instruction*

Description :

Une instruction précédée de LOCK verrouille la mémoire (plus exactement la broche LOCK#) pour la durée de l'instruction. Ceci permet de garantir l'indivisibilité du cycle lecture/modification/écriture. Quand plusieurs processeurs sont sur le bus, ce signal important est nécessaire pour assurer un accès total au bus et à toute mémoire liée à celui-ci pendant certaines opérations. La gestion d'un sémaphore en est un bon exemple.

On ne peut employer LOCK qu'avec les instructions suivantes :

ADC, ADD, AND, BT	mem, reg/imm
BTS, BTR, BTC, OR	mem, reg/imm
SBB, SUB, XOR	mem, reg/imm
XCHG	reg, mem
XCHG	mem, reg
DEC, INC, NEG, NOT	mem

Si on spécifie un LOCK avec les instructions qui ne sont pas dans cette liste, une exception du type CodeOp non valide est générée (voir le Chapitre 6).

Fonctionnement :

```
LOCK# = 0;
instruction;
LOCK# = 1;
```

Exemple :

```
;La broche LOCK# est verrouillée pendant l'exécution de BTS
LOCK BTS dmot ptr [ESI],ECX
```

Exceptions :

InvalidOpCode();

Des exceptions Memory() peuvent être générées par l'instruction quand elle est verrouillée.

Notes :

La liste des instructions verrouillables du 80386 est différente de celle des autres circuits de la famille 86. Voir l'Annexe A.

Voir l'exemple 4, page 553.

LODS/LODSB/LODSW/LODSD

Chargement d'une chaîne

Formats :

LODSB
LODSW
LODSD
LODS *mem*$_1$
mem8
mem16
mem32

Description :

Ces instructions transfèrent un, deux, ou quatre octets de données (pour des opérations 8, 16 et 32 bits) de [ESI] vers les registres AL, AX, ou EAX. Après ce transfert, ESI pointe vers le prochain élément de la chaîne et ESI est corrigé par 1, 2 ou 4 (resp. -1, -2, ou -4) si DF est égal à 0 (resp. 1).

mem$_1$ détermine la longueur de l'opérande. Ce transfert est toujours effectué avec l'adresse déterminée par le registre ESI. Cependant, un recouvrement de segment peut être précisé dans mem$_1$ relativement à [ESI].

Fonctionnement :

```
if (CODEOP == LODSB){
      AL = (byte) DS:[ESI];
      delta = 1;
} else if (CODEOP == LODSW){
      AX = (word) DS:[ESI];
```

```
        delta = 2;
} else if (CODEOP == LODSD){
        EAX = (dword) DS:[ESI];
        delta = 4;
} else {        /* LODS */
        /* Notez que SegReg(mem1) par défaut est DS */
        if (Length(mem1) == 8){
           AL = SegReg(mem1):[ESI];
           delta = 1;
        } else if (Length(mem1) == 16){
           AX = SegReg(mem1):[ESI];
           delta = 2;
        } else if (Length(mem1) == 32){
           EAX = SegReg(mem1):[ESI];
           delta = 4;
        }
}
if (DF == 1) {
        delta = - delta;
}
ESI = ESI + delta;
```

Exemple :

```
LODSB   Avant :   ESI :   00000052   SS : [ESI]    :   BF
                  AL  :   XX         DF            :   0
        Après :   ESI :   00000053   SS : [ESI-1]  :   BF
                  Al  :   BF         DF            :   0
```

Exception :

Memory();

Note :

Voir les exemples 6a et 6b, pages 558 et 559.

LOOP

Boucle avec Compteur ECX

Formats :

LOOP short_label

Description :

L'instruction décrémente le registre ECX sans affecter les indicateurs. Si après cette opération, la valeur de ECX n'est pas nulle, l'exécution se poursuit à l'adresse déterminée par l'étiquette courte (short_label). Si le registre ECX est nul, c'est l'instruction suivant le LOOP qui est exécutée. Une étiquette courte se situe dans un intervalle de +127 octets à -128 octets autour de l'instruction LOOP (le déplacement varie de -128 à 127 octets à partir du EIP courant).

Fonctionnement :

```
ECX = ECX - 1;     /* Pas d'altération des indicateurs */
if (ECX != 0){
      EIP = short_label;   /*   */
}
/*
        L'étiquette courte est assemblée en un
      simple déplacement immédiat 8b.
      Ce déplacement est relatif au pointeur
      d'instruction courant, qui pointe vers la
      prochaine instruction exécutable. Ainsi
      EIP devient :
         EIP = EIP + SignEx(32,8,imm8)
*/
```

Exemple :

```
LOOP    near_label1    ; label1 se trouve 4Eh octets plus loin
          Avant :    EIP :    000DE300    ECX :    00000002
          Après :    EIP :    000DE2B2    ECX :    00000001
LOOP    near_label1    ;label1 se trouve 4Eh octets plus loin
          Avant :    EIP :    000DE300    ECX :    00000001
          Après :    EIP :    000DE300    ECX :    00000000
                       ; Exécute la prochaine instruction
```

Exception :

CodeReference();

Notes :

L'instruction LOOP devrait être placée en fin de boucle, et l'étiquette courte au début.

Voir l'exemple 2, page 546.

LOOPNZ/LOOPNE

Boucle avec ECX et test si non nul

Formats :

LOOPNZ	short_label
LOOPNE	short_label

Description :

L'instruction décrémente le registre ECX sans affecter les indicateurs. Si, après cette opération, la valeur de ECX n'est pas nulle et si ZF est nul, l'exécution se poursuit à l'adresse déterminée par l'étiquette courte (short_label). Si le registre ECX est nul ou si l'indicateur ZF est positionné, c'est l'instruction suivant le LOOPNE ou LOOPNZ qui est exécutée. Une étiquette courte se situe dans un intervalle de +127 octets à -128 octets autour de l'instruction LOOPNZ/LOOPNE (le déplacement varie de -128 à 127 octets à partir du EIP courant).

Fonctionnement :

```
ECX = ECX - 1;     /* Pas d'altération des indicateurs */
if ((ECX != 0) && (ZF == 0)){
      EIP = short_label;    /*    */
}
/*
        L'étiquette courte est assemblée en un
      simple déplacement immédiat 8b.
      Ce déplacement est relatif au pointeur
      d'instruction courant, qui pointe vers la
      prochaine instruction exécutable. Ainsi
      EIP devient :
```

```
        EIP = EIP + SignEx(32,8,imm8)
*/
```

Exemple :

```
                                OSZAPC
LOOPNE  near_label2       ; label2 se trouve 4Eh octets plus loin
        Avant :   EIP :   000DE300      ECX :   00000008   XX1XXX
        Après :   EIP :   000DE300      ECX :   00000007   UU1UUU
                        ; Exécute la prochaine instruction

LOOPNZ  near_label3       ; label3 se trouve 3Ch octets plus loin
        Avant :   EIP :   000C8300      ECX :   00000002   XX0XXX
        Après :   EIP :   000C82C4      ECX :   00000001   UU0UUU
```

Exception :

CodeReference();

Notes :

LOOPNE et LOOPNZ sont synonymes.

L'instruction LOOPNE/LOOPNZ devrait être placée en fin de boucle, et l'étiquette courte au début.

Voir l'exemple 11, page 574.

LOOPZ/LOOPE

Boucle avec ECX et test si nul

Formats :

LOOPZ short_label
LOOPE short_label

Description :

L'instruction décrémente le registre ECX sans affecter les indicateurs. Si la valeur de ECX après cette opération n'est pas nulle et si ZF est positionné, l'exécution se poursuit à l'adresse déterminée par l'étiquette courte (short_label). Si le registre ECX est nul ou si l'indicateur ZF est nul, c'est l'instruction suivant le LOOPE ou LOOPZ qui est exécutée. Une étiquette courte se situe dans un intervalle de +127 octets à -128 octets autour de l'instruction LOOPNZ/LOOPNE (le déplacement varie de -128 à 127 octets à partir du EIP courant).

Fonctionnement :

```
ECX = ECX - 1;     /* Pas d'altération des indicateurs */
if ((ECX != 0) && (ZF == 1)){
      EIP = short_label;    /*    */
}
/*
        L'étiquette courte est assemblée en un
      simple déplacement immédiat 8b.
      Ce déplacement est relatif au pointeur
      d'instruction courant, qui pointe vers la
      prochaine instruction exécutable. Ainsi
      EIP devient :
```

```
        EIP = EIP + SignEx(32,8,imm8)
*/
```

Exemple :

```
                                OSZAPC
LOOPE   near_label2      ; label2 se trouve 4Eh octets plus loin
         Avant :   EIP :   000DE300      ECX :   00000001   XX0XXX
         Après :   EIP :   000DE300      ECX :   00000000   UU0UUU
                         ; Exécute la prochaine instruction

LOOPZ   near_label3      ; label3 se trouve 3Ch octets plus loin
         Avant :   EIP :   000C8300      ECX :   00000002   XX1XXX
         Après :   EIP :   000C82C4      ECX :   00000001   UU1UUU
```

Exception :

CodeReference();

Notes :

LOOPE et LOOPZ sont synonymes.

L'instruction LOOPE/LOOPZ devrait être placée en fin de boucle, et l'étiquette courte au début.

MOV

Transfert d'opérandes

Formats :

MOV	op_1	,	op_2
	reg	,	reg
	reg	,	mem
	mem	,	reg
	reg	,	imm
	mem	,	imm

Description :

MOV copie le contenu de op_2 dans op_1. op_1 et op_2 peuvent être des octets, des mots ou des dmots.

Fonctionnement :

```
op1 = op2;
```

Exemples :

```
MOV AX,0A80h     Avant :   AX  :   XXXX
                 Après :   AX  :   0A80

MOV EAX,mem32    Avant :   EAX :   XXXXXXXX      mem :   0892ABDF
                 Après :   EAX :   0892ABDF      mem :   0892ABDF
```

```
MOV mem8,AL     Avant :   AL  :   4A          mem :   XX
                Après :   AL  :   4A          mem :   4A
```

Exception :

Memory();

MOVS/MOVSB/MOVSW/MOVSD

Transfert de chaînes

Formats :

```
MOVSB
MOVSW
MOVSD
MOVS   mem1    ,  mem2
       mem8    ,  mem8
       mem16   ,  mem16
       mem32   ,  mem32
```

Description :

MOVS transfère l'octet, le mot ou le dmot déterminé par [ESI] dans ES:[EDI]. Les registres ESI et EDI sont mis à jour pour pointer sur le prochain élément de la chaîne, selon l'état de l'indicateur de direction (DF) et la longueur des opérandes (8, 16, ou 32 bits) qui est précisée par mem_1 et mem_2 ou par le CODEOP lui-même (MOVSB, MOVSW, ou MOVSD). Si DF est à 0 (resp. 1), les valeurs 1, 2 ou 4 (resp. -1, -2, ou -4) corrigent les registres.

L'instruction MOVS peut être précédée du préfixe REP. S'il en est ainsi, le MOVS est répété ECX fois. Ainsi, une chaîne de ECX octets, mots ou dmots est transférée de [ESI] dans ES:[EDI].

L'assembleur utilise mem_1 et mem_2 pour déterminer la longueur de l'opération, et un éventuel recouvrement de segment pour mem_2. On transfère toujours les chaînes de [ESI] vers ES:[EDI]. MOVS peut comporter un préfixe de recouvrement segment, qui affecte le déplacement employé pour mem_2 ([ESI]). mem_1 provient toujours de ES:[EDI].

Fonctionnement :

```
if (CODEOP == MOVSB){
      delta = 1;
      ES:[EDI] = (byte) DS:[ESI];
} else if (CODEOP == MOVSW){
      delta = 2;
      ES:[EDI] = (word) DS:[ESI];
} else if (CODEOP == MOVSD){
      delta = 4;
      ES:[EDI] = (dword) DS:[ESI];
} else {      /* MOVS */
      /* Notez que par défaut Segreg(mem2) est DS */
      delta = Length(mem1);
      ES:[EDI] = SegReg(mem2):[ESI];
}
if (DF == 1) {
      delta = -delta;
}
ESI = ESI + delta;
EDI = EDI + delta;
```

Exemples :

```
MOVSD   Avant :  ESI      :     06533A40   EDI         :  07822CD4
                 [ESI]    :     FFFFFFFF   ES :[EDI]    :  XXXXXXXX
                 DF       :     0
        Après :  ESI      :     06533A44   EDI         :  07822CD8
                 [ESI-4]  :     FFFFFFFF   ES :[EDI-4]  :  FFFFFFFF
                 DF       :     0

REP MOVSD
        Avant :  ESI          :     00000008   EDI     :  00000010
                 ECX          :     00000002   [ESI]    :  01324567
                 ES :[EDI]    :     XXXXXXXX   [ESI-4]  :  F421890A
                 ES :[EDI-4]  :     XXXXXXXX   DF       :  1
```

```
Après :   ESI          :     00000000   EDI      :  00000008
          ECX          :     00000000   [ESI+8]  :  01324567
          ES :[EDI+8]  :     01324567   [ESI+4]  :  F421890A
          ES :[EDI+4]  :     F421890A   DF       :  1
```

Exception :

Memory();

Note :

Voir l'exemple 1, page 544.

MOVSX

Transfert avec extension de signe

Formats :

MOVSX	op_1	,	op_2
	reg16	,	reg8
	reg16	,	mem8
	reg16	,	reg16
	reg16	,	mem16
	reg32	,	reg8
	reg32	,	mem8
	reg32	,	reg16
	reg32	,	mem16

Description :

Le mot ou dmot op_1 devient une extension de signe de l'octet ou du mot op_2. Si op_1 et op_2 sont tous les deux des mots, l'instruction équivaut à un classique MOV.

Fonctionnement :

```
if (Length(op1) == 16){
      if (Length(op2) == 8){
         op1 = SignEx(16,8,op2);
      } else {
         op1 = op2;    /* MOV classique */
      }
} else {       /* Length(op1) == 32 */
      if (Length(op2) == 8){
         op1 = SignEx(32,8,op2);
      } else {
```

```
            op1 = SignEx(32,16,op2);
        }
    }
```

Exemples :

```
MOVSX AX,BH        Avant :   AX  :   XXXX    BH :   9E
                   Après :   AX  :   FF9E    BH :   9E

MOVSX EDX,mem16    Avant :   EDX :   XXXXXXXX       mem : 742D
                   Après :   EDX :   0000742D       mem : 742D
```

Exception :

Memory();

Note :

Cette instruction est une nouveauté du 80386.

MOVZX

Transfert avec extension de zéro

Formats :

MOVZX	op_1	,	op_2
	reg16	,	reg8
	reg16	,	mem8
	reg16	,	reg16
	reg16	,	mem16
	reg32	,	reg8
	reg32	,	mem8
	reg32	,	reg16
	reg32	,	mem16

Description :

Le mot ou dmot op_1 devient une extension de zéro de l'octet ou du mot op_2. Si op_1 et op_2 sont tous les deux des mots, l'instruction équivaut à un classique MOV.

Fonctionnement :

```
if (Length(op1) == 16){
      if (Length(op2) == 8){
         op1 = ConCat(16,00h,8,op2);
      } else {
         op1 = op2;    /* MOV classique ! */
      }
} else {       /* Length(op1) == 32 */
      if (Length(op2) == 8){
         op1 = ConCat(32,00h,8,op2);
      } else {
```

```
            op1 = ConCat(32,00h,16,op2);
        }
    }
```

Exemples :

```
MOVZX AX,BH        Avant :   AX  :   XXXX   BH :   9E
                   Après :   AX  :   009E   BH :   9E

MOVZX EDX,mem16    Avant :   EDX :   XXXXXXXX       mem : 742D
                   Après :   EDX :   0000742D       mem : 742D
```

Exception :

Memory();

Note :

Cette instruction est une nouveauté du 80386.

MUL

Multiplication non signée

Formats :

MUL op_1
reg
mem

Description :

L'instruction MUL ne comprend qu'un opérande ; le second opérande est toujours implicite. op_1 est multiplié par les registres AL, AX, ou EAX pour des opérations 8, 16, ou 32 bits. Les résultats non signés 16, 32, ou 64 bits sont rangés dans AX, DX:AX, ou EDX:EAX. Si AH, DX, ou EDX sont nuls pour des opérations 8, 16, ou 32 bits, CF et OF sont positionnés à 0. Sinon, ils sont mis à 1.

Fonctionnement :

```
if (Length(op1) == 8){
      AX = AL * op1;
      if (AH == 00h){
         CF = 0; OF = 0;
      } else {
         CF = 1; OF = 1;
      }
} else if (Length(op1) == 16){
      DX = (AX * op1) >> 16;
      AX = FFFFh & (AX * op1);
      if (DX == 0000h){
         CF = 0; OF = 0;
      } else {
```

```
            CF = 1; OF = 1;
         }
   } else {       /* Length(op1) == 32 */
         EDX = (EAX * op1) >> 32;
         EAX = FFFFFFFFh & (EAX * op1);
         if (EDX == 00000000h){
            CF = 0; OF = 0;
         } else {
            CF = 1; OF = 1;
         }
   }
   SF = ZF = AF = PF = UNDEFINED;
   /* * est définie comme la multiplication non signée */
```

Exemples :

			OSZAPC						
MUL BX	Avant :	DX :	XXXX	AX :	229A	BX	:	A431	XXXXXX
	Après :	DX :	1631	AX :	477A	BX	:	A431	1XXXX1
MUL mem8	Avant :	AH :	XX	AL :	0A	mem	:	12	XXXXXX
	Après :	AH :	00	AL :	B4	mem	:	12	0XXXX0

Exception :

Memory();

NEG

Complémentation à deux

Formats :

NEG	op_1
	mem
	reg

Description :

NEG forme le complément à deux d'un opérande donné. L'instruction soustrait op_1 de 0 et place le résultat dans op_1.

L'indicateur de retenue est positionné à 1 sauf si op_1 (la valeur initiale de op_1) est égal à 0.

Fonctionnement :

```
CF = op1 != 0;
op1 = 0 - op1;
/* - est défini par l'instruction SUB */

OF = Carry0(MSB)^Carry0(MSB-1);
SF = B(MSB,op1);
ZF = op1 == 0;
AF = Carry0(3);
PF = ~(B(7,op1)^B(6,op1)^...^B(0,op1));
```

Exemples :

```
                                          OSZAPC
NEG EAX     Avant :   EAX :   01234567    XXXXXX
            Après :   EAX :   FEDCBA99    010111

NEG mem8    Avant :   Mem :   00          XXXXXX
            Après :   Mem :   00          001010
```

Exception :

Memory();

NOP

Pas d'opération

Format :

NOP

Description :

NOP ne fait rien. Le seul effet de cette instruction est de modifier l'adresse de la prochaine instruction exécutable.

Fonctionnement :

```
;
```

Exemple :

```
NOP       ; Rien ne se passe
```

Exceptions :

Aucune.

Notes :

NOP est employé pour des boucles de temporisation (il dure 3 cycles horloge) et pour aligner des étiquettes. Cet alignement sur des frontières de quatre octets peut accélérer l'exécution, tout particulièrement pour du code souvent exécuté, comme les boucles.

NOT

NON logique

Formats :

NOT op_1
reg
mem

Description :

L'instruction effectue un NON (NOT) logique ou complément à un sur op_1 et place le résultat dans op_1. Les indicateurs ne sont pas modifiés.

Fonctionnement :

```
op1 = ~op1;
/*
~ est défini par la table logique suivante.
ai est le ième bit de op1. ri est le ième bit du résultat.
La variable binaire i prend les valeurs  0 à MSB.
ai      ri
0       1
1       0
*/
```

Exemples :

```
NOT EAX     Avant :   EAX  :   01234567
            Après :   EAX  :   FEDCBA98

NOT mem8    Avant :   mem8 :   72
            Après :   mem8 :   8D
```

Exception :

Memory();

OR

OU inclusif logique

Formats :

OR	op_1	,	op_2
	reg	,	reg
	reg	,	mem
	reg	,	imm
	mem	,	reg
	mem	,	imm

Description :

L'instruction effectue un OU logique inclusif entre op_1 et op_2 et place le résultat dans op_1.

Fonctionnement :

```
op1 = op1 | op2;
OF = 0;
SF = B(MSB,op1);
ZF = op1 == 0;
AF = UNDEFINED;
PF = ~(B(7,op1)^B(6,op1)^...^B(0,op1));
C» = 0;
/*
   | est défini par la table logique suivante.
   ai, bi sont les ièmes bits de op1 et op2.
   ri est le ième bit du résultat.
   La variable binaire i prend les valeurs 0 à MSB.
   ai   bi   ri
   0    0    0
```

```
    0    1    1
    1    0    1
    1    1    1
*/
```

Exemples :

```
                                                                OSZAPC
OR EAX,EBX        Avant :   EAX :   01234567      EBX :   76543210   XXXXXX
                  Après :   EAX :   77777777      EBX :   76543210   000X10

OR mem16,7878h    Avant :   Mem :   1F90                             XXXXXX
                  Après :   Mem :   7FF8                             000X00

OR AL,0FFh        Avant :   AL  :   A2                               XXXXXX
                  Après :   AL  :   FF                               010X10
```

Exception :

Memory();

OUT

Sortie E/S

Formats :

OUT	op_1	,	op_2
	imm8	,	AL
	imm8	,	AX
	imm8	,	EAX
	DX	,	AL
	DX	,	AX
	DX	,	EAX

Description :

L'instruction OUT transfère une donnée octet, mot ou dmot de AL, AX, ou EAX vers le port spécifié.

On peut préciser le port par un octet immédiat permettant d'accéder aux ports 0 à 255 ; dans ce cas les bits de poids fort de l'adresse du port sont toujours nuls. On peut aussi placer le numéro du port dans le registre DX. On peut alors accéder à tous les ports (0 à 64 K-1).

Précisons qu'Intel se réserve les adresses E/S 00F8 à 00FF.

Fonctionnement :

```
port(op1) = op2;
```

Exemples :

```
OUT 0C3h,AX    Avant :   AX :   1C9B   port(00C3) :   XXXX
               Après :   AX :   1C9B   port(00C3) :   1C9B

OUT DX,EAX     Avant :   EAX :   96238887   DX :   4684
                         port(4684)           :   XXXXXXXX
               Après :   EAX :   96238887   DX :   4684
                         port(4684)           :   96238887
```

Exception :

AccessIO();

OUTS/OUTSB/OUTSW/OUTSD

Sortie E/S d'une chaîne

Formats :

```
OUTSB
OUTSW
OUTSD
OUTS    reg1    ,  mem2
        DX      ,  mem8
        DX      ,  mem16
        DX      ,  mem32
```

Description :

Ces instructions transfèrent un, deux ou quatre octets de données (pour des opérations 8, 16, et 32 bits) selon mem_2 ou le CODEOP lui-même (OUTSB, OUTSW, ou OUTSD) de la mémoire déterminée par [ESI] vers le port déterminé par DX. Après ce transfert, ESI pointe vers la chaîne suivante. ESI est corrigé par 1, 2 ou 4 (resp. -1, -2, ou -4) si DF est 0 (resp. 1).

On doit préciser le port utilisé dans le registre DX. On n'a pas le droit de choisir un port par l'adressage immédiat.

mem_2 détermine la longueur de l'opérande. Le transfert s'effectue toujours avec l'adresse dans le registre ESI. Cependant, un recouvrement de segment peut être appliqué à [ESI] via mem_2.

Un préfixe REP peut précéder OUTS. Dans ce cas, l'instruction transfère ECX octets, mots ou dmots. Dans la description de l'instruction REP, on expliquera cette technique plus précisément.

Fonctionnement :

```
if (CODEOP == OUTSB){
      port(DX) = (byte) DS:[ESI];
      delta = 1;
} else if (CODEOP == OUTSW){
      port(DX) = (word) DS:[ESI];
      delta = 2;
} else if (CODEOP == OUTSD){
      port(DX) = (dword) DS:[ESI];
      delta = 4;
} else {      /* OUTS */
      /* Notez que par défaut SegReg(mem2) == DS */
      if (Length(mem2) == 8){
         port(DX) = (byte) SegReg(mem2):[ESI];
         delta = 1;
      } else if (Length(mem2) == 16){
         port(DX) = (word) SegReg(mem2):[ESI];
         delta = 2;
      } else if (Length(mem2) == 32){
         port(DX) = (dword) SegReg(mem2):[ESI];
         delta = 4
      }
}
if (DF == 1) {
      delta = -delta;
}
ESI = ESI + delta;
```

Exemples :

```
OUTSW   Avant :  ESI      :  6240   [ESI]      :  FFF6
                 DX       :  004C   DF         :  0
                 port(DX) :  XXXX
        Après :  ESI      :  6244   [ESI-4]    :  FFF6
                 DX       :  004C   DF         :  0
                 port(DX) :  FFF6
```

```
REP OUTS DX,S1        ; S1 est une chaîne d'octet
        Avant :   ESI :   00000040    [ESI]        :   F2
                  [ESI+1]     :   FF   [ESI+2]      :   00
                  DX          :   20   DF           :   0
                  port(DX)    :   XX   CX           :   3
        Après :   ESI :   00000043    [ESI-3]      :   F2
                  [ESI-2]     :   FF   [ESI-1]      :   00
                  DX          :   20   DF           :   0
                  port(DX,1)  :   F2   port(DX,2)   :   FF
                  port(DX,3)  :   00   CX           :   0
```

Exceptions :

Memory();
AccessIO();

POP

Dépilage

Formats :

POP	op_1
	mem16
	reg16
	mem32
	reg32

Description :

POP transfère un mot ou dmot du sommet de la pile dans op_1. SS:[ESP] pointe sur le sommet de la pile. Après ce transfert, l'instruction ajoute 2 ou 4 au pointeur de pile selon la longueur de l'opérande (16 ou 32 bits).

Fonctionnement :

```
if (Length(op1) == 32){
      op1 = SS:[ESP];
      ESP = ESP + 4;
} else {
      op1 = SS:[ESP];
      ESP = ESP + 2;
}
```

Exemple :

```
POP AX   Avant :   AX :   XXXX   SS :[ESP]   :   F340   ESP :   F4320530
         Après :   AX :   F340   SS :[ESP-2] :   F340   ESP :   F4320532
```

Exceptions :

Memory();
Stack();

Note :

Prenez garde lorsque vous dépilez des opérandes 16 bits d'éviter un mauvais alignement de la pile qui dégraderait les performances !

POPA/POPAD

Dépilage de tous les registres

Formats :

POPA
POPAD

Description :

POPA/POPAD transfèrent huit mots ou dmots du sommet de la pile vers les huit registres généraux. Ainsi, POPA/POPAD nous évitent d'employer huit instructions POP consécutives. L'ordre des registres empilés est : DI, SI, BP, SP, BX, DX, CX, et AX pour les mots (POPA) ; ou EDI, ESI, EBP, ESP, EBX, EDX, ECX, et EAX pour les dmots (POPAD).

Notez que la valeur empilée de ESP est laissée de côté.

Fonctionnement :

```
if (CODEOP == POPAD){
      EDI = SS:[ESP+0];
      ESI = SS:[ESP+4];
      EBP = SS:[ESP+8];
      tmp = SS:[ESP+12];    /* la valeur de ESP est mise de côté */
      EBX = SS:[ESP+16];
      EDX = SS:[ESP+20];
      ECX = SS:[ESP+24];
      EAX = SS:[ESP+28];
      ESP = ESP + 32;
} else {
      DI = SS:[ESP+0];
      SI = SS:[ESP+2];
```

```
        BP = SS:[ESP+4];
        tmp = SS:[ESP+6];   /* la valeur de ESP est mise de côté */
        BX = SS:[ESP+8];
        DX = SS:[ESP+10];
        CX = SS:[ESP+12];
        AX = SS:[ESP+14];
        ESP = ESP + 16;
    }
```

Exemple :

```
                                OSZAPC
POPA   Avant :   AX : XXXX   BX : XXXX   CX : XXXX   DX : XXXX   XXXXXX
                 SP : 7236   BP : XXXX   SI : XXXX   DI : XXXX
                 SS : [07236] :   42FF   SS : [07238] :   AA2C
                 SS : [0723A] :   B290   SS : [0723C] :   8861
                 SS : [0723E] :   0000   SS : [07240] :   0002
                 SS : [07242] :   FFFF   SS : [07244] :   2133
       Après :   AX : 2133   BX : FFFF   CX : 0002   DX : 0000   UUUUUU
                 SP : 7236   BP : B290   SI : AA2C   DI : 42FF
                 SS : [07236] :   42FF   SS : [07238] :   AA2C
                 SS : [0723A] :   B290   SS : [0723C] :   8861
                 SS : [0723E] :   0000   SS : [07240] :   0002
                 SS : [07242] :   FFFF   SS : [07244] :   2133
```

Exception :

Stack();

POPF/POPFD

Dépilage des indicateurs

Formats :

POPF
POPFD

Description :

Ces instructions copient le contenu courant du sommet de la pile dans le registre 32 bits des indicateurs, puis incrémente le pointeur de pile de 2 ou 4 (opérations 16 ou 32 bits) pour pointer sur le nouveau sommet de pile.

La figure suivante montre le mot d'état 32 bits.

31	30	...	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
		...			V	R		N	IOP		O	D	I	T	S	Z		A		P		C
0	0	...	0	0	M	F	0	T	L		F	F	F	F	F	F	0	F	0	F	1	F

POPF modifie uniquement les 16 bits de poids faible du registre des indicateurs.

Si le niveau de privilège courant est égal à 0, seul l'indicateur de niveau de privilège E/S sera modifié. Sinon, les indicateurs de niveau de privilège E/S ne seront pas modifiés et aucune exception ne sera générée.

Si le niveau de privilège courant est égal ou supérieur au niveau de privilège E/S courant, l'indicateur d'autorisation des interruptions sera modifié. Sinon, il ne sera pas modifié et aucune exception ne sera générée.

Fonctionnement :

```
RF = 0;
NT = ([ESP] & 00004000h) >> 14;
if (CPL == 00b){
      IOPL = ([ESP] & 00003000h) >> 12;
}
OF = ([ESP] & 00000800h) >> 11;
DF = ([ESP] & 00000400h) >> 10;
if (CPL <= IOPL){
      IF = ([ESP] & 00000200h) >> 9;
}
TF = ([ESP] & 00000100h) >> 8;
SF = ([ESP] & 00000080h) >> 7;
ZF = ([ESP] & 00000040h) >> 6;
AF = ([ESP] & 00000010h) >> 4;
PF = ([ESP] & 00000004h) >> 2;
CF = ([ESP] & 00000001h) >> 0;
if (CODEOP == POPFD){
      ESP = ESP + 4;
} else {
      ESP = ESP + 2;
}
```

Exemple :

```
                                                          II
                                                          OO
                                                       VRNPPODITSZ A P C
                                                       MFTLLFFFFFF0F0F1F
POPFD      ; Le niveau de privilège est 0
             Avant :    ESP :   20143110   [ESP] :   FFF601A5o   OXXXXXXXXXX0X0X1X
             Après :    ESP :   20143114                          U0000000110000111
```

Exception :

Stack();

PUSH

Empilage

Formats :

PUSH op_1
imm
mem
reg

Description :

L'instruction place op_1 au sommet de la pile après avoir décrémenté le pointeur de pile ESP de 2 ou 4 (16 ou 32 bits). L'instruction considère toujours qu'une donnée immédiate a une taille de 32 bits, même si le codage de l'instruction spécifie un opérande de 8 bits (voir le Chapitre 2).

Fonctionnement :

```
if (Length(op1) == 32){
      SS:[ESP-4] = op1;
      ESP = ESP - 4;
} else {       /* Length(op1) == 16 */
      SS:[ESP-2] = op1;
      ESP = ESP -2;
```

Exemples :

```
PUSH 24h   Avant :   ESP :   FFFFF320   SS : [FFFFF31C] :   XXXXXXXX
           Après :   ESP :   FFFFF31C   SS : [FFFFF31C] :   00000024

PUSH AX    Avant :   ESP :   FFFFF31C   SS : [FFFFF31A] :   XXXX
                     AX  :   F799
           Après :   ESP :   FFFFF31A   SS : [FFFFF31A] :   F799
                     AX  :   F799
```

Exceptions :

Stack();
Memory();

Note :

Lorsque vous empilez des opérandes 16 bits, prenez garde de bien aligner la pile, sinon les performances se dégraderont !

PUSHA/PUSHAD

Empilage de tous les registres

Formats :

PUSHA
PUSHAD

Description :

PUSHA/PUSHAD copient les huit registres mots ou dmots au sommet de la pile. Ainsi, PUSHA/PUSHAD nous évitent d'employer huit instructions PUSH consécutives. L'ordre d'empilage des registres est le suivant : AX, CX, DX, BX, SP, BP, SI, et DI pour des mots ; ou EAX, ECX, EDX, EBX, ESP, EBP, ESI, et EDI pour des dmots.

Précisons que la valeur empilée pour SP ou ESP est la valeur originale.

L'ordre d'empilage des registres par un PUSHA permet un futur POPA.

Fonctionnement :

```
if (CODEOP == PUSHAD) {
      SS:[ESP-4] = EAX;
      SS:[ESP-8] = ECX;
      SS:[ESP-12] = EDX;
      SS:[ESP-16] = EBX;
      SS:[ESP-20] = ESP;    /* Note : valeur originale de ESP */
      SS:[ESP-24] = EBP;
      SS:[ESP-28] = ESI;
      SS:[ESP-32] = EDI;
```

```
            ESP = ESP - 32;
        } else {
            SS:[ESP-2] = AX;
            SS:[ESP-4] = CX;
            SS:[ESP-6] = DX;
            SS:[ESP-8] = BX;
            SS:[ESP-10] = SP;    /* Note : valeur originale de ESP */
            SS:[ESP-12] = BP;
            SS:[ESP-14] = SI;
            SS:[ESP-16] = DI;
            ESP = ESP - 16;
        }
```

Exemple :

```
                                OSZAPC
PUSHA   Avant :   AX : 0000   BX : 1111   CX : 2222   DX : 3333   XXXXXX
                  SP : 4444   BP : 5555   SI : 6666   DI : 7777
                  SS : [04442] :   XXXX   SS :[04440] :   XXXX
                  SS : [0443E] :   XXXX   SS :[0443C] :   XXXX
                  SS : [0443A] :   XXXX   SS :[04438] :   XXXX
                  SS : [04436] :   XXXX   SS :[04434] :   XXXX
        Après :   AX : 0000   BX : 1111   CX : 2222   DX : 3333   UUUUUU
                  SP : 4434   BP : 5555   SI : 6666   DI : 7777
                  SS : [04442] :   0000   SS :[04440] :   1111
                  SS : [0443E] :   2222   SS :[0443C] :   3333
                  SS : [0443A] :   4444   SS :[04438] :   5555
                  SS : [04436] :   6666   SS :[04434] :   7777
```

Exception :

Stack();

PUSHF/PUSHFD

Empilage des Indicateurs

Formats :

PUSHF
PUSHFD

Description :

Ces instructions décrémentent le pointeur de pile de 2 ou 4 (mot d'état de 16 ou 32 bits) pour former le nouveau sommet de pile. Ce qui est dans le registre d'état 16 ou 32 bits est copié dans ce nouveau sommet.

Le schéma suivant montre le registre EFLAGS.

31	30	...	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	...	0	0	VM	RF	0	NT	IOPL		OF	DF	IF	TF	SF	ZF	0	AF	0	PF	1	CF

Fonctionnement :

```
if (PUSHF){
      ESP = ESP - 2;
      SS:[ESP] =        (0 <<15) | (NT <<14) |
                   (IOPL <<12) | (OF <<11) | (DF <<10) | (IF <<9) |
                      (SF <<7) | (ZF <<6)  | (0 <<5)   | (AF <<4) |
             (0 <<3) | (PF <<2) | (1 <<1)  | (CF <<0);
} else {       /* PUSHFD */
      ESP = ESP - 4;
```

```
SS:[ESP] = (0000h <<16) | (0 <<15)  | (NT <<14) |
           (IOPL <<12) | (OF <<11) | (DF <<10) | (IF <<9) |
              (SF <<7) | (ZF <<6)  | (0 <<5)   | (AF <<4) |
               (0 <<3) | (PF <<2)  | (1 <<1)   | (CF <<0);
}
```

Exemple :

```
                                                  II
                                                  OO
                                               VRNPPODITSZ A P C
                                                MFTLLFFFFFF0F0F1F
PUSHFD
   Avant :    [ESP-4] :   XXXXXXXX   ESP :   FFF42104   00100011011000011
   Après :    [ESP]   :   000046C3   ESP :   FFF42100   00100011011000011
```

Exception :

Stack();

Note :

Voir l'exemple 1, page 544.

RCL

Rotation à gauche avec retenue

Formats :

RCL	op_1	,	op_2
	reg	,	imm8
	mem	,	imm8
	reg	,	CL
	mem	,	CL

Description :

op_2 détermine le nombre de rotations. Ce nombre est masqué à 5 bits (0-31) qui sont considérés modulo la longueur de l'opérande plus 1 pour former le nombre de rotations.

La rotation s'effectue avec op_1 et l'indicateur de retenue. Ainsi, l'instruction fait une rotation sur 33, 17, ou 9 bits pour un dmot, un mot ou un octet.

On peut considérer l'instruction de rotation comme n décalages à gauche (n étant représenté par le nombre de rotations), avec le bit de poids fort qui tombe dans la retenue et la retenue qui remplace le bit de poids faible.

Si le nombre de rotations est 1, l'indicateur de débordement est positionné à 0 si l'indicateur de retenue final (après l'instruction RCL) est égal au bit de poids fort du résultat. Sinon, il est à 1. S'il y a plus d'une rotation, l'indicateur de débordement est indéfini.

Fonctionnement :

```
if (Length(op1) == 8){
      cnt = (op2 & 01Fh) % 9;
      if (cnt>0){
         temp = (op1 << cnt)    |
            (CF << (cnt - 1))    |
            (op1 >> (9 - cnt));
         CF = (op1 >> (8 - cnt)) & 01;
         op1 = temp;
      }
} else if (Length(op1) == 16){
      cnt = (op2 & 01Fh) % 17;
      if (cnt>0){
         temp = (op1 << cnt)    |
            (CF << (cnt - 1))    |
            (op1 >> (17 - cnt));
         CF = (op1 >> (16 - cnt)) & 01;
         op1 = temp;
      }
} else if (Length(op1) == 32){
      cnt = (op2 & 01Fh);
      if (cnt>0){
         temp = (op1 << cnt)    |
            (CF << (cnt - 1))    |
            (op1 >> (33 - cnt));
         CF = (op1 >> (32 - cnt)) & 01;
         op1 = temp;
      }
}
if (op2 == 1){
      OF = B(MSB,op1)^CF;
} else {
      OF = UNDEFINED;
}
```

Exemples :

```
                                                OSZAPC
RCL EAX,1       Avant :   EAX :   01234567      XXXXX1
                Après :   EAX :   02468ACF      0UUUU0

RCL mem8,CL     Avant :   mem :   9B   CL :  11 XXXXX0
                Après :   mem :   4D   CL :  11 XUUUU1
```

Exception :

Memory();

Notes :

Un nombre de rotations nul ne modifie pas l'indicateur de retenue !

On ne peut effectuer un RCL de 32 ou 33 bits (dmot), puisque le nombre de rotations varie de 0 à 31.

RCR

Rotation à droite avec retenue

Formats :

RCR	op_1	,	op_2
	reg	,	imm8
	mem	,	imm8
	reg	,	CL
	mem	,	CL

Description :

op_2 détermine le nombre de rotations. Ce nombre est masqué à 5 bits (0-31) qui sont considérés modulo la longueur de l'opérande plus 1 pour former le nombre de rotations.

La rotation s'effectue avec op_1 et l'indicateur de retenue. Ainsi, l'instruction fait une rotation sur 33, 17, ou 9 bits pour un dmot, un mot ou un octet.

On peut considérer l'instruction de rotation comme n décalages à droite (n étant représenté par le nombre de rotations), avec le bit de poids faible qui tombe dans la retenue et la retenue qui remplace le bit de poids fort.

Si le nombre de rotations est égal à 1, l'indicateur de débordement est positionné à 0 si les deux bits de poids fort du résultat sont égaux. Sinon, il est à 1. S'il y a plus d'une rotation, l'indicateur de débordement est indéfini.

Fonctionnement :

```
if (op2 == 1){
      OF = B(MSB,op1)^CF;
} else {
      OF = UNDEFINED;
}
if (Length(op1) == 8){
      cnt = (op2 & 01Fh) % 9;
      if (cnt>0){
         temp = (op1 >> cnt)    |
            (CF << (cnt - 1))    |
            (op1 << (9 - cnt));
         CF = (op1 >> (8 - cnt)) & 01;
         op1 = temp;
      }
} else if (Length(op1) == 16){
      cnt = (op2 & 01Fh) % 17;
      if (cnt>0){
         temp = (op1 >> cnt)    |
            (CF << (cnt - 1))    |
            (op1 << (17 - cnt));
         CF = (op1 >> (16 - cnt)) & 01;
         op1 = temp;
      }
} else if (Length(op1) == 32){
      cnt = (op2 & 01Fh);
      if (cnt>0){
         temp = (op1 >> cnt)    |
            (CF << (cnt - 1))    |
            (op1 << (33 - cnt));
         CF = (op1 >> (32 - cnt)) & 01;
         op1 = temp;
      }
}
```

Exemples :

```
                                                    OSZAPC
RCR mem32,1   Avant :   mem :   01234567            XXXXX0
              Après :   mem :   0091A2B3            1UUUU1

RCR EDI,CL   Avant :   EDI  :   A3214551   CL : 11  XXXXX0
             Après :   EDI  :   45515190   CL : 11  XUUUU1
```

Exception :

Memory();

Notes :

Un nombre de rotations nul ne modifie pas l'indicateur de retenue !

On ne peut pas effectuer un RCL de 32 ou 33 bits (dmot), puisque le nombre de rotations varie de 0 à 31.

REP

Répétition

Formats :

REP *Instruction sur une chaîne*
INS
MOVS
OUTS
STOS

Description :

REP est le préfixe d'instruction sur une chaîne dont la présence entraîne la répétition de l'instruction ECX fois.

Fonctionnement :

```
while (ECX != 0){
      ECX = ECX - 1;
      Instruction_Chaîne;
}
OF, SF, ZF, AF, PF, CF = définis par l'instruction sur les chaînes
```

Exemples :

```
REP MOVSD
      Avant :   ESI          :   00000008   EDI      : 0000008
                ECX          :   00000002   [ESI]    : 01324567
                ES :[EDI]    :   XXXXXXXX   [ESI-4]  : F421890A
                ES :[ESI-4]  :   XXXXXXXX   DF       : 1
```

```
Après :   ESI          :   00000000   EDI      : 00000010
          ECX          :   00000000   [ESI+8]  : 01324567
          ES :[EDI+8]  :   01324567   [ESI+4]  : F421890A
          ES :[EDI+4]  :   F421890A   DF       : 1
```

Exceptions :

Aucune pour REP, mais l'instruction sur les chaînes peut en générer une.

REPE/REPZ

Répétition tant qu'égal

Formats :

REPE/REPZ *Instruction sur une chaîne*
CMPS
SCAS

Description :

REPE est le préfixe d'une instruction travaillant sur les chaînes. Sa présence entraîne la répétition de cette instruction ECX fois, ou jusqu'à ce que le bit ZF passe à 0. REPZ est synonyme de REPE.

Fonctionnement :

```
while (ECX != 0){
      ECX = ECX - 1;
      Instruction_Chaîne;
      if (ZF == 0) break;
}
OF, SF, ZF, AF, PF, CF = définis par l'instruction sur les chaînes
```

Exemples :

```
REPE CMPSB
Avant :   ESI :       00000008   EDI          :   16   ECX          :   04
          [ESI]         :   01   [ESI+1]       :   23   [ESI+2]      :   45
          [ESI+3]       :   67   ES :[EDI]     :   01   ES :[EDI+1]  :   23
          ES :[EDI+2]   :   44   ES :[EDI+3]   :   67   DF           :   0
```

```
Après :   ESI :       0000000B   EDI          :   19   ECX          :   01
          [ESI-3]     :   01     [ESI-2]      :   23   [ESI-1]      :   45
          [ESI]       :   67     ES :[EDI-3]  :   01   ES :[EDI-2]  :   23
          ES :[EDI-1] :   44     ES :[EDI]    :   67   DF           :   0
```

Exceptions :

Aucune pour REPE/REPZ, mais l'instruction sur les chaînes peut générer une exception.

Note :

Voir l'exemple 5, page 555.

REPNE/REPNZ

Répétition tant que pas égal

Formats :

REPNE/REPNZ *Instruction sur une chaîne*
CMPS
SCAS

Description :

REPNE est le préfixe d'une instruction travaillant sur les chaînes. Sa présence entraîne la répétition de cette instruction ECX fois ou jusqu'à ce que le bit ZF passe à 1. REPNZ est synonyme de REPNE.

Fonctionnement :

```
while (ECX != 0){
      ECX = ECX - 1;
      Instruction_Chaîne;
      if (ZF == 1) break;
}
OF, SF, ZF, AF, PF, CF = définis par l'instruction sur les chaînes
```

Exemples :

```
REPNE CMPSB
    Avant :   ESI :       00000008   EDI          :   16   ECX          :   04
              [ESI]         :   01   [ESI+1]       :   23   [ESI+2]      :   45
              [ESI+3]       :   67   ES :[EDI]     :   01   ES :[EDI+1]  :   23
              ES :[EDI+2]   :   44   ES :[EDI+3]   :   67   DF           :   0
```

```
Après :   ESI :       00000009   EDI          :   17   ECX          :   03
          [ESI-1]       :   01   [ESI]        :   23   [ESI+1]      :   45
          [ESI+2]       :   67   ES :[EDI-1]  :   01   ES :[EDI]    :   23
          ES :[EDI+1]   :   44   ES :[EDI+2]  :   67   DF           :   0
```

Exceptions :

Aucune pour REPE/REPZ, mais l'instruction de chaîne peut générer une exception.

Note :

Voir l'exemple 5, page 555.

RET

Retour d'une procédure

Formats :

NEAR	RET	
	RET	imm16

Description :

RET redonne le contrôle à l'instruction qui suivait le CALL associé. Le CALL a sauvegardé dans la pile l'adresse de cette instruction et cette dernière se retrouve donc au sommet de la pile. En d'autres termes, RET effectue un saut indirect via le pointeur situé au sommet de la pile, où on trouve EIP.

On peut préciser une valeur imm16 optionnelle qui est additionnée à ESP. Cela est utile si vous voulez dépiler plusieurs octets qui servent de paramètres.

Fonctionnement :

```
EIP = Pop();
if (NUMOPS == 1) {
      ESP = ESP + imm16;
}
```

Exemples :

```
RET   Avant :   EIP :   804231AA   SS :[ESP] :   032BBD4C
                ESP :   04260988
      Après :   EIP :   032BBD4C
                ESP :   0426098C
```

Exceptions :

Stack();
CodeReference();

ROL

Rotation à gauche

Formats :

ROL	op_1	,	op_2
	reg	,	imm8
	mem	,	imm8
	reg	,	CL
	mem	,	CL

Description :

op_2 détermine le nombre de rotations. Ce nombre est masqué à 5 bits. Ainsi, on peut effectuer des rotations de 0 à 31 bits.

L'instruction effectue la rotation sur op_1 et range le résultat dans op_1.

On peut considérer l'instruction de rotation comme n décalages à gauche (n étant le nombre de rotations), avec le bit de poids fort remplaçant le bit de poids faible et tombant dans l'indicateur de retenue après chaque décalage. Ainsi, la retenue finale est égale au bit de poids faible du résultat.

Si le nombre de rotations est égal à 1, l'indicateur de débordement est positionné à 0 si l'indicateur de retenue (après l'instruction ROL) est égal au bit de poids fort de op_1 (le résultat). Sinon, il est à 1. S'il y a plus d'une rotation, l'indicateur de débordement est indéfini.

Fonctionnement :

```
if (Length(op1) == 8){
      cnt = op2 % 8;
      if (cnt>0){
         op1 = (op1 << cnt) |
            (op1 >> (8 - cnt));
         CF = B(0,op1);
      }
} else if (Length(op1) == 16){
      cnt = op2 % 16;
      if (cnt>0){
         op1 = (op1 << cnt) |
            (op1 >> (16 - cnt));
         CF = B(0,op1);
      }
} else {   /* (Length(op1) == 32) */
      cnt = op2 % 32;
      if (cnt>0){
         op1 = (op1 << cnt) |
            (op1 >> (32 - cnt));
         CF = B(0,op1);
      }
}
if (op2 == 1){
      OF = (MSB,op1)^CF;
} else {
      OF = UNDEFINED;
}
```

Exemples :

					OSZAPC
ROL AL,1	Avant :	AL :	A6		XXXXXX
	Après :	AL :	4D		1UUUU1

```
ROL BP,CL     Avant :   BP :   6279   CL :   0A   XXXXXX
              Après :   BP :   E589   CL :   0A   XUUUU1
```

Exception :

Memory();

Note :

ROL avec un nombre de rotations nul ne modifie ni la retenue ni l'indicateur de débordement !

ROR

Rotation à droite

Formats :

ROR	op_1	,	op_2
	reg	,	imm8
	mem	,	imm8
	reg	,	CL
	mem	,	CL

Description :

op_2 détermine le nombre de rotations. Ce nombre est masqué à 5 bits. Ainsi, on peut effectuer des rotations de 0 à 31 bits.

L'instruction effectue la rotation sur op_1 et range le résultat dans op_1.

On peut considérer l'instruction de rotation comme n décalages à droite (n étant le nombre de rotations), avec le bit de poids faible remplaçant le bit de poids fort et tombant dans l'indicateur de retenue après chaque décalage. Ainsi, la retenue finale est égale au bit de poids fort du résultat.

Si le nombre de rotations est égal à 1, l'indicateur de débordement est positionné à 0 si les deux bits de poids fort de op_1 (le résultat) sont égaux. Sinon, il est à 1. S'il y a plus d'une rotation, l'indicateur de débordement est indéfini.

Fonctionnement :

```
if (Length(op1) == 8){
      cnt = op2 % 8;
```

```
        if (cnt>0){
            op1 = (op1 >> cnt) |
                (op1 >> (8 - cnt));
            CF = B(7,op1);
        }
} else if (Length(op1) == 16){
        cnt = op2 % 16;
        if (cnt>0){
            op1 = (op1 >> cnt) |
                (op1 >> (16 - cnt));
            CF = B(15,op1);
        }
} else {    /* (Length(op1) == 32) */
        cnt = op2 % 32;
        if (cnt>0){
            op1 = (op1 >> cnt) |
                (op1 >> (32 - cnt));
            CF = B(31,op1);
        }
}
if (op2 == 1){
        OF = (MSB,op1)^(MSB-1,op1);
} else {
        OF = UNDEFINED;
}
```

Exemples :

				OSZAPC
ROR mem16,1	Avant :	mem :	E4A2	XXXXXX
	Après :	mem :	7251	1UUUU0
ROR EDI,7	Avant :	EDI :	07BDAF21	XXXXXX
	Après :	EDI :	420F7B5E	XUUUU0

Exception :

Memory();

Note :

ROL avec un nombre de rotations nul ne modifie ni la retenue ni l'indicateur de débordement !

SAHF

Rangement de AH dans les indicateurs

Formats :

SAHF

Description :

L'instruction transfère AH dans l'octet de poids faible du mot d'état. Elle range les bits 7, 6, 4, 2 et 0 du registre AH respectivement dans les indicateurs de signe, de zéro, de retenue auxiliaire, de parité et de retenue. Le schéma suivant montre les 8 bits de poids faible du registre EFLAGS :

Fonctionnement :

```
SF = (AH & 80h) >> 7;
ZF = (AH & 40h) >> 6;
AF = (AH & 10h) >> 4;
PF = (AH & 04h) >> 2;
CF = (AH & 01h) >> 0;
```

Exemples :

```
                                        OSZAPC
SAHF    Avant :   AH :   10X1X0X0b      XXXXXX
        Après :   AH :   10X1X0X0b      U1O1OO
```

Exceptions :

Aucune.

Note :

Voir l'exemple 7, page 566.

SAL/SHL

Décalage arithmétique à gauche

Formats :

SAL/SHL	op_1	,	op_2
	reg	,	imm8
	mem	,	imm8
	reg	,	CL
	mem	,	CL

Description :

op_2 détermine le nombre de décalages qui est masqué à 5 bits. Ainsi, on peut effectuer des décalages de 0 à 31 bits.

L'instruction effectue le décalage sur op_1 et range le résultat dans op_1. Elle décale op_1 n fois à gauche (n étant le nombre de décalages), avec les bits de poids faible mis à 0. L'indicateur de retenue représente le bit sortant.

Si le nombre de décalages est égal à 1, l'indicateur de débordement est positionné à 0 si la retenue (après l'instruction SAL/SHL) est égale au bit de poids fort de op_1 (le résultat). Sinon, il est à 1. S'il y a plus d'un décalage, l'indicateur de débordement est indéfini.

Notez que SAL et SHL sont synonymes et que les décalages à gauche arithmétiques et logiques sont identiques.

Fonctionnement :

```
cnt = op2 & 01Fh;
if (cnt < Length(op1)){
      (if (cnt > 0){
         CF = B(Length(op1) - cnt), op1);
      }
      op1 = op1 << cnt;
} else {
      CF = 0;
      op1 = 0;
}
if (op2 == 1){
      OF = (MSB,op1)^CF;
} else {
      OF = UNDEFINED;
}
SF = B(MSB,op1);
ZF = op1 == 0;
AF = UNDEFINED;
PF = ~(B(7,op1)^B(6,op1)^...^B(0,op1));
```

Exemples :

							OSZAPC
SAL BX,1	Avant :	BX :	ADF3				XXXXXX
	Après :	BX :	5BE6				100X01
SHL mem32,CL	Avant :	mem :	A6341209	CL :	09		XXXXXX
	Après :	mem :	68241200	CL :	09		X00X10

Exception :

Memory();

Note :

Des décalages de zéro ne modifient pas les indicateurs !

SAR

Décalage arithmétique à droite

Formats :

SAR	op_1	,	op_2
	reg	,	imm8
	mem	,	imm8
	reg	,	CL
	mem	,	CL

Description :

op_2 détermine le nombre de décalages qui est masqué à 5 bits. Ainsi, on peut effectuer des décalages de 0 à 31 bits.

L'instruction effectue le décalage sur op_1 et range le résultat dans op_1. Elle décale op_1 n fois à droite (n étant le nombre de décalages), avec les bits de poids fort remplis avec le bit de signe de op_1 (le bit le plus significatif). L'indicateur de retenue représente le bit sortant.

Si le nombre de décalages est égal à 1, l'indicateur de débordement est positionné à 0. Sinon, il est inchangé.

Notez que dans cette instruction, op_1 est divisé par 2 à chaque décalage. Dans ce cas, on obtient un arrondi vers moins l'infini, contrairement à l'instruction IDIV pour les nombres négatifs.

Fonctionnement :

```
cnt = op2 & 01Fh;
if (cnt > 0){
```

```
        CF = B(cnt,op1);
    }
    if (cnt < Length(op1){
        if (Length(op1) == 8){
            if (B(7,op1) == 1){
                op1 = (op1 >> cnt) | (FFh << (8-cnt));
            } else {
                op1 = (op1 >> cnt);
            }
        } else if (Length(op1) == 16){
            if (B(15,op1) == 1){
                op1 = (op1 >> cnt) | (FFFFh << (16-cnt));
            } else {
                op1 = (op1 >> cnt);
            }
        } else {        /* Length(op1) == 32) */
            if (B(31,op1) == 1){
                op1 = (op1 >> cnt) | (FFFFFFFFh << (32-cnt));
            } else {
                op1 = (op1 >> cnt);
            }
        }
    } else {
        if (B(Length(op1),op1) == 1){
            CF = 1;
            if (Length(op1) == 8){
                op1 = FFh;
            } else {        /* doit être 16 */
                op1 = FFFFh;
            }
        } else {
            CF = 0;
            if (Length(op1) == 8){
                op1 = 00h;
            } else {        /* doit être 16 */
                op1 = 0000h;
            }
        }
    }
    if (op2 == 1){
        OF = 0;
```

```
} else {
      OF = OF;
}
SF = B(MSB,op1);
ZF = op1 == 0;
AF = UNDEFINED;
PF = ~(B(7,op1)^B(6,op1)^...^B(0,op1));
```

Exemples :

```
                                                    OSZAPC
SAR mem8,1    Avant :   mem :   84                  XXXXXX
              Après :   mem :   C2                  010X00

SAR EDI,CL    Avant :   EDI :   A6341209   CL : 13  XXXXXX
              Après :   EDI :   FFFFF4C6   CL : 13  X10X11
```

Exception :

Memory();

Notes :

Les décalages de zéro ne modifient pas les indicateurs !

On peut employer SAR pour diviser rapidement des entiers non signés par des puissances de deux. Voir l'exemple 1, page 544.

SBB

Soustraction avec retenue

Formats :

SBB	op_1	,	op_2
	reg	,	reg
	reg	,	mem
	reg	,	imm
	mem	,	reg
	mem	,	imm

Description :

SBB soustrait op_2 plus la retenue de op_1 et place le résultat dans op_1. L'indicateur de retenue (positive ou négative) indique une retenue négative (*borrow*) pour la soustraction. Ce qui équivaut à :

```
op1 = op1 - (op2 + CF);
```

On emploie l'instruction SBB pour des soustractions multiples sur des octets, des mots ou des dmots.

Fonctionnement :

```
op1 = op1 - (op2 + CF);
OF = Borrow0(MSB)^Borrow0(MSB-1);
SF = B(MSB,op1);
ZF = op1 == 0;
```

```
AF = Borrow0(3);
PF = ~(B(7,op1)^B(6,op1)^...^B(0,op1));
CF = Borrow0(MSB);
/*
   - est défini par la table logique suivante.
   xi, yi sont les ièmes bits de op1 et op2.
   bi-1 est la retenue négative sortant du bit i-1
   (c'est la même que la retenue négative entrant dans le bit i)
   ri est le ième bit du résultat
   bi est la  retenue sortant du ième bit.
   la variable binaire i prend les valeurs 0 à MSB
   bi pour i-1 == -1 est CF (la retenue négative) pour SBB.

   xi  yi  bi-1   ri  bi
   0   0   0      0   0
   0   0   1      1   1
   0   1   0      1   1
   0   1   1      0   1
   1   0   0      1   0
   1   0   1      0   0
   1   1   0      0   0
   1   1   1      1   1

   Notez qu'on peut aussi définir la soustraction par une addition.
   Comparez avec la table logique de l'addition et avec :
      xi =  ai
      yi = ~bi
      bi =  ci
   Ainsi, on peut aussi identifier l'indicateur de retenue du 80386
   à un indicateur de retenue négative.
*/
```

Exemples :

```
                                                          OSZAPC
SBB EAX,EBX    Avant :   EAX :   0743CE21   EBX :   8D4BC956   XXXXX1
               Après :   EAX :   79F804CA   EBX :   8D4BC956   010111
```

```
SBB mem8,1Ah    Avant :    mem :    72                    XXXXX0
                Après :    mem :    58                    000100
```

Exception :

Memory();

SCAS/SCASB/SCASW/SCASD

Recherche de chaîne

Formats :

SCASB
SCASW
SCASD
SCAS *mem*$_1$
mem8
mem16
mem32

Description :

SCAS soustrait ES:[EDI] de AL, AX, ou EAX (opérations sur des octets, des mots, ou des dmots). L'instruction ne sauvegarde pas le résultat de la soustraction : elle change uniquement les indicateurs. Elle modifie le registre EDI pour qu'il pointe sur le prochain élément de la chaîne, selon l'indicateur de direction (DF) et la longueur des opérandes (8, 16, ou 32 bits) comme l'indique mem$_1$ ou le CODEOP lui-même (SCASB, SCASW, SCASD). Si l'indicateur DF est à 0 (resp. 1), EDI est corrigé de 1, 2, ou 4 (resp. -1, -2, ou -4).

Un préfixe REPE (REPZ) ou REPNE (REPNZ) peut précéder l'instruction SCAS. Dans le cas de REPE, l'instruction SCAS se répète tant que ECX n'est pas égal à 0 et que les éléments de la chaîne sont égaux à AL, AX, ou EAX (ZF==1). Dans le cas de REPNE, l'instruction SCAS se répète tant que ECX n'est pas égal à 0 et que les éléments de la chaîne ne sont pas égaux à AL, AX, ou EAX (ZF==0). De cette façon, on utilise SCAS pour trouver la première différence (REPE) ou égalité (REPNE) à AL, AX, ou EAX dans la chaîne. Se reporter à REPE et à REPNE pour des précisions supplémentaires sur les préfixes.

L'assembleur utilise mem_1 (s'il est présent) uniquement pour déterminer la longueur de l'opération. Il considère toujours la chaîne dans ES:[EDI]. SCAS n'autorise pas de recouvrement de segment.

Fonctionnement :

```
if ((Length(mem1) == 8) || (CODEOP == SCANSB)){
      delta = 1;
      temp = AL - (byte) ES:[EDI];
} else if ((Length(mem1) == 16) || (CODEOP == SCANSW)){
      delta = 2;
      temp = AX - (word) ES:[EDI];
} else {      /* (Length(mem1) == 32) || (CODEOP == SCANSD) */
      delta = 4;
      temp = EAX - (dword) ES:[EDI];
}
/* - est défini par l'instruction SUB */
OF = Borrow0(MSB)^Borrow0(MSB-1);
SF = B(MSB,temp);
ZF = temp == 0;
AF = Borrow0(3);
PF = ~(B(7,temp)^B(6,temp)^...^B(0,temp));
CF = Borrow0(MSB);
if (DF == 1) {
      delta = - delta;
}
EDI = EDI + delta;
```

Exemples :

```
                                                                  OSZAPC
SCASW
   Avant :   AX :   0123   EDI :   0016   ES :[EDI]   :   3210     XXXXXX
             DF :   0
```

```
   Après :   AX :   0123   EDI :   0018   ES :[EDI-2] :   3210          010001
             DF :   0

REPNE SCAS S1   ;   Notez que S1 est une chaîne d'octets
   Avant :   EDI          : 0008   AL          : 72   ES :[EDI] :   23     XXXXXX
             ES :[EDI-1] : 23     ES :[EDI-2] : 72   ES :[EDI-3] : AA
             DF           : 0      ECX :   00000008
   Après :   EDI          : 0005   AL          : 72   ES :[EDI+3] :   23   001010
             ES :[EDI+2] : 23     ES :[EDI+1] : 72   ES :[EDI] : AA
             DF           : 0      ECX :   00000005
```

Exception :

Memory();

Note :

Voir l'exemple 5, page 555.

SETcc

Assignation conditionnelle d'octet

Formats :

SETcc op_1
reg8
mem8

Où le code de condition est un des suivants :

Mnémoniques	Condition	Description
SETB/SETNAE/	CF == 1	Assignation si inférieur ou si retenue
SETC	"	""
SETBE/SETNA	CF == 1 ‖ ZF == 1	Assignation si inférieur ou égal
SETE/SETZ	ZF == 1	Assignation si égal
SETL/SETNGE	SF != OF	Assignation si plus petit
SETLE/SETNG	SF != OF ‖ ZF == 1	Assignation si plus petit ou égal
SETNB/SETAE/	CF == 0	Assignation si supérieur ou égal
SETNC	"	ou si pas de retenue
SETNBE/SETA	CF == 0 && ZF == 0	Assignation si supérieur
SETNE/SETNZ	ZF == 0	Assignation si différent ou si non nul
SETNL/SETGE	SF == OF	Assignation si plus grand ou égal
SETNLE/SETG	ZF == 0 && SF == OF	Assignation si plus grand
SETNO	OF == 0	Assignation si pas débordement
SETNP/SETPO	PF == 0	Assignation si pas parité
SETNS	SF == 0	Assignation si pas signe
SETO	OF == 1	Assignation si débordement
SETP/SETPE	PF == 1	Assignation si parité
SETS	SF == 1	Assignation si signe

Notez que *plus grand* et *plus petit* se rapportent à des comparaisons sur des entiers signés, tandis que *supérieur* et *inférieur* traitent de comparaisons sur des entiers non signés.

Description :

L'instruction teste les indicateurs selon les conditions décrites ci-dessus. Si les indicateurs remplissent ces conditions, op_1 est mis à 1. Sinon, op_1 est mis à 0.

Notez que les instructions SETcc ne travaillent que sur des opérandes d'un octet.

L'assembleur accepte de nombreux mnémoniques identiques pour faciliter l'interprétation des indicateurs. Par exemple, SETA (assignation si supérieur, *set byte above*) et SETNBE (positionnement si pas inférieur ou égal, *set byte not below or equal*) sont synonymes. L'assembleur accepte les deux syntaxes.

Fonctionnement :

```
if (ConditionCode) {
      op1 = 1;
} else {
      op1 = 0;
}
```

Exemples :

```
                                          OSZAPC
SETNO mem8    Avant :   mem8 :      XX    1XXXXX
              Après :   mem8 :      00    1UUUUU
```

```
SETNBE AL    Avant :   AL   :      XX   XX1XX0
             Après :   AL   :      01   UU1UU0
```

Exceptions :

CodeReference();
Memory();

Note :

Cette instruction est une nouveauté du 80386.

SHLD

Décalage à gauche double

Formats :

SHLD	op_1	,	op_2	,	op_3
	reg16	,	reg16	,	imm8
	mem16	,	reg16	,	imm8
	reg16	,	reg16	,	CL
	mem16	,	reg16	,	CL
	reg32	,	reg32	,	imm8
	mem32	,	reg32	,	imm8
	reg32	,	reg32	,	CL
	mem32	,	reg32	,	CL

Description :

op_3 détermine le nombre de décalages, qui est masqué à 5 bits. Ainsi, on peut effectuer des décalages de 0 à 31 bits.

L'instruction effectue le décalage sur op_1:op_2 et range le résultat dans op_1. Elle décale op_1 n fois à gauche (n étant le nombre de décalages), avec les bits de poids faible qui sont remplacés par les bits de poids fort de op_2. L'indicateur de retenue devient le bit sortant provenant du décalage du bit de poids fort.

Si le nombre de décalages est égal à 0, l'instruction équivaut à un NOP. S'il est égal à 1, l'indicateur de débordement est mis à 1 si le bit le plus significatif du résultat est différent de l'indicateur de retenue. Sinon, il est mis à 0. Si on spécifie un nombre de décalages plus grand que la longueur de l'opérande, les indicateurs et le résultat de op_1 sont indéfinis.

Fonctionnement :

```
cnt = op3 & 01Fh;
if (cnt == 0){
      /* nop */
} else if (cnt > Length(op1)){
      op1 = OF = SF = ZF = PF = CF = AF = UNDEFINED;
} else {
      CF = B(Length(op1) - cnt,op1);
      op1 = (op1 << cnt) | (op2 >> (Length(op2) - cnt));
      OF = B(MSB,op1)^CF);
      SF = B(MSB,op1);
      ZF = op1 == 0;
      PF = (B(7,op1)^B(6,op1)^...^B(0,op1));
      AF = UNDEFINED;
}
```

Exemples :

								OSZAPC
SHLD mem16,AX,3	Avant :	mem :	0084	AX :	00F3			XXXXXX
	Après :	mem :	0420	AX :	00F3			000X00
SHLD DI,SI,CL	Avant :	DI :	1209	SI :	8552	CL :	0B	XXXXXX
	Après :	DI :	4C2A	SI :	8552	CL :	0B	00X000

Exception :

Memory();

Notes :

Des décalages de zéro ne modifient pas les indicateurs !

Cette instruction est une nouveauté du 80386.

Voir l'exemple 6b, page 559.

SHR

Décalage logique à droite

Formats :

SHR	op_1	,	op_2
	reg	,	imm8
	mem	,	imm8
	reg	,	CL
	mem	,	CL

Description :

op_3 détermine le nombre de décalages, qui est masqué à 5 bits. Ainsi, on peut effectuer des décalages de 0 à 31 bits.

L'instruction effectue le décalage sur op_1 et range le résultat dans op_1. Elle décale op_1 n fois à droite (n étant le nombre de décalages), avec les bits de poids fort qui sont remplacés par des 0. L'indicateur de retenue devient le bit sortant provenant du décalage du bit de poids faible.

S'il est égal à 1, l'indicateur de débordement est mis à 1 si les deux bits les plus significatifs du résultat sont différents. Sinon, il est mis à 0. Si le nombre de décalages n'est pas égal à 1, l'indicateur de débordement est indéfini.

Fonctionnement :

```
cnt = op3 & 01Fh;
if (cnt =< Length(op1)){
      if (cnt > 0){
         CF = B((cnt),op1);
```

```
        }
        op1 = op1 >> cnt;
} else {
        op1 = 0;
        CF = 0;
}
if (op2 == 1){
        OF = B(MSB,op1)^B(MSB-1,op1);
} else {
        OF = UNDEFINED;
}
SF = B(MSB,op1);
ZF = op1 == 0;
PF = ~(B(7,op1)^B(6,op1)^...^B(0,op1));
AF = UNDEFINED;
```

Exemples :

							OSZAPC
SHR DX,1	Avant :	DX	:	84			XXXXXX
	Après :	DX	:	42			000X10
SHR mem32,CL	Avant :	mem	:	F42BBD51	CL	: 0C	XXXXXX
	Après :	mem	:	000F42BB	CL	: 0C	X00X11

Exception :

Memory();

Notes :

Des décalages de zéro ne modifient pas les indicateurs !

Voir l'exemple 6b, page 559.

SHRD

Décalage à droite double

Formats :

SHRD	op_1	,	op_2	,	op_3
	reg16	,	reg16	,	imm8
	mem16	,	reg16	,	imm8
	reg16	,	reg16	,	CL
	mem16	,	reg16	,	CL
	reg32	,	reg32	,	imm8
	mem32	,	reg32	,	imm8
	reg32	,	reg32	,	CL
	mem32	,	reg32	,	CL

Description :

op_3 détermine le nombre de décalages, qui est masqué à 5 bits. Ainsi, on peut effectuer des décalages de 0 à 31 bits.

L'instruction effectue le décalage sur op_1:op_2 et range le résultat dans op_1. Elle décale op_1 n fois à droite (n étant le nombre de décalages), avec les bits de poids fort qui sont remplacés par les bits de poids faible de op_2. L'indicateur de retenue devient le bit sortant provenant du décalage du bit de poids faible.

Si le nombre de décalages est égal à 0, l'instruction équivaut à un NOP. S'il est égal à 1, l'indicateur de débordement est mis à 1 si les deux bits les plus significatifs du résultat sont différents. Sinon, il est mis à 0.

Si on spécifie un nombre de décalages plus grand que la longueur de l'opérande, les indicateurs et le résultat de op_1 sont indéfinis.

Fonctionnement :

```
cnt = op3 & 01Fh;
if (cnt == 0){
      /* nop */
} else if (cnt > Length(op1)){
      op1 = OF = SF = ZF = PF = CF = AF = UNDEFINED;
} else {
      CF = B((cnt-1),op1);
      op1 = (op1 >> cnt) | (op2 << (Length(op2) - cnt));
      OF = B(MSB,op1)^B(MSB-1,op1);
      SF = B(MSB,op1);
      ZF = op1 == 0;
      PF = ~((B(7,op1)^B(6,op1)^...^B(0,op1));
      AF = UNDEFINED;
}
```

Exemples :

							OSZAPC
SHRD mem16,BP,2	Avant :	mem :	7AC0	BP :	F43D3	XXXXXX
	Après :	mem :	5EB0	AX :	F43D	100X00
SHRD EAX,mem32,CL	Avant :	EAX :	663B982F			
		mem :	9963120A	CL :	0A	XXXXXX
	Après :	EAX :	82998EE6			
		mem :	9963120A	CL :	OA	110X10

Exception :

Memory();

Notes :

Des décalages de zéro ne modifient pas les indicateurs !

Cette instruction est une nouveauté du 80386.

Voir l'exemple 6b, page 559.

STC

Positionnement de l'indicateur de retenue

Format :

STC

Description :

L'instruction positionne l'indicateur de retenue à 1.

Fonctionnement :

```
CF = 1;
```

Exemple :

```
STC      Avant :   CF :   0
         Après :   CF :   1
```

Exceptions :

Aucune.

STD

Positionnement de l'indicateur de direction

Format :

STD

Description :

L'instruction positionne l'indicateur de direction à 1. Après cette opération, les instructions de chaînes décrémenteront les registres d'index associés (ESI et EDI).

Fonctionnement :

```
DF = 1;
```

Exemples :

```
STD   Avant :   DF :   0
      Après :   DF :   1
```

Exceptions :

Aucune.

STI

Positionnement de l'indicateur d'interruption

Format :

STI

Description :

L'instruction positionne l'indicateur d'autorisation d'interruption à 1. Le 80386 répondra aux interruptions après l'exécution de la prochaine instruction.

Si la tâche courante n'a pas un privilège suffisant pour modifier cet indicateur, il y aura génération d'une exception du type CodeOp indéfini. Le Chapitre 5 décrit plus précisément les niveaux de privilège.

Fonctionnement :

```
IF = 1;
```

Exemples :

```
STI      Avant :   IF :   0
         Après :   IF :   1
```

Exception :

IOPLSensitive();

STOS/STOSB/STOSW/STOSD

Stockage de chaîne

Formats :

STOSB
STOSW
STOSD
STOS *mem*$_1$
mem8
mem16
mem32

Description :

Ces instructions transfèrent un, deux, ou quatre octets de données (pour des opérations 8, 16, et 32 bits) des registres AL, AX ou EAX vers ES:[EDI]. Après cette opération, EDI est modifié pour pointer sur le prochain élément de la chaîne, selon l'indicateur de direction (DF) et la longueur de l'opérande (8,16,32). Cette dernière est déterminée par mem$_1$ ou par le CODEOP lui-même (STOS, STOSW, ou STOSD). Si DF est égal à 0 (resp. 1), EDI est corrigé de 1, 2, ou 4(resp. -1, -2, ou -4).

mem$_1$ détermine la longueur de l'opérande. Normalement, le transfert s'effectue à l'adresse spécifiée par ES:[EDI]. Le recouvrement de segment n'est pas possible.

Le préfixe REP peut précéder l'instruction STOS. Cela permet d'initialiser une chaîne avec le contenu de AL, AX, ou EAX.

Fonctionnement :

```
if ((Length(mem1) == 8) || (CODEOP == STOSB)){
      delta = 1;
      ES:[EDI] = AL;
} else if ((Length(mem1) == 16) || (CODEOP == STOSW)){
      delta = 2;
      ES:[EDI] = AX;
} else {   /* Length(mem1) == 32) || (CODEOP == STOSD)) */
      delta = 4;
      ES:[EDI] = EAX;
}
if (DF ==1) {
      delta = - delta;
}
EDI = EDI + delta;
```

Exemples :

```
STOSW      Avant :   EDI :   0052   ES :[EDI]    :  XXXX
                     AX  :   FFFF   DF           :  0
           Après :   EDI :   0054   ES :[EDI-2]  :  FFFF
                     AX  :   FFFF   DF           :  0
```

Exception :

Memory();

Note :

Voir l'exemple 1, page 544.

SUB

Soustraction

Formats :

SUB	op_1	,	op_2
	reg	,	reg
	reg	,	mem
	reg	,	imm
	mem	,	reg
	mem	,	imm

Description :

L'instruction soustrait op_2 de op_1 et place le résultat dans op_1.

Fonctionnement :

```
op1 = op1 - op2;
OF = Borrow0(MSB)^Borrow0(MSB-1);
SF = B(MSB,op1);
ZF = op1 == 0;
AF = Borrow0(3);
PF = ~(B(7,op1)^B(6,op1)^...^B(0,op1));
CF = Borrow0(MSB);
/*
   - est défini par la table logique suivante.
   xi, yi sont les ièmes bits de op1 et op2.
   bi-1 est la retenue négative sortant du bit i-1
   (c'est la même que la retenue négative entrant dans le bit i)
   ri est le ième bit du résultat
   bi est la  retenue sortant du ième bit.
   la variable binaire i prend les valeurs 0 à MSB
```

```
   bi pour i-1 == -1 est CF (la retenue négative) pour SBB.

   xi  yi  bi-1   ri  bi
   0   0   0      0   0
   0   0   1      1   1
   0   1   0      1   1
   0   1   1      0   1
   1   0   0      1   0
   1   0   1      0   0
   1   1   0      0   0
   1   1   1      1   1

   Notez qu'on peut aussi définir la soustraction par une addition.
   Comparez la table logique de l'addition avec :
      xi =  ai
      yi = ~bi
      bi =  ci
   Ainsi, on peut identifier l'indicateur de retenue du 80386
   à un indicateur de retenue négative.
*/
```

Exemples :

```
                                                          OSZAPC
SUB EAX,EBX     Avant :   EAX :   0743CE21      EBX :   8D4BC956   XXXXXX
                Après :   EAX :   79F804CB      EBX :   8D4BC956   000101

SUB mem8,3Ah    Avant :   mem :   72                               XXXXXX
                Après :   mem :   38                               000100
```

Exception :

Memory();

TEST

Comparaison logique

Formats :

TEST	op_1	,	op_2
	reg	,	reg
	reg	,	mem
	reg	,	imm

Description :

L'instruction effectue un ET (AND) logique entre op_1 et op_2, et ne sauvegarde pas le résultat. Seuls les indicateurs sont modifiés.

Fonctionnement :

```
temp = op1 & op2;
OF = 0;
SF = B(MSB,temp);
ZF = temp == 0;
AF = UNDEFINED;
PF = ~(B(7,temp)^B(6,temp)^...^B(0,temp));
CF = 0;
/*
   & est défini par la table logique suivante.
   ai et bi sont les bits de opi et de op2.
   ri est la ième variable du résultat.
   la variable binaire i prend les valeurs 0 à MSB
   ai  bi     ri
   0   0      0
   0   1      0
   1   0      0
   1   1      1
*/
```

Exemples :

```
                                                                OSZAPC
TEST EAX,EBX   Avant :   EAX :   01234567      EBX :   76543210   XXXXXX
               Après :   EAX :   01234567      EBX :   76543210   001X10

TEST mem8,75   Avant :   Mem :   7F                               XXXXXX
               Après :   Mem   : 7F                               000X00
```

Exception :

Memory();

XCHG

Echange

Formats :

XCHG	op_1	,	op_2
	reg	,	reg
	mem	,	reg
	reg	,	mem

Description :

L'instruction échange les contenus de op_2 et de op_1. op_1 ou op_2 peuvent être des octets, des mots ou des dmots mais doivent toujours être de longueur égale.

Si op_1 ou op_2 sont du type mem, le transfert au niveau du bus est traité comme s'il y avait un préfixe LOCK, même si LOCK n'a pas été spécifié. Ainsi, il est utile d'employer XCHG pour des opérations sur des sémaphores.

Fonctionnement :

```
temp = op1;
op1 = op2;
op2 = temp;
```

Exemples :

```
XCHG AX,BX       Avant :   AX :   0A8F   BX :   9042
                 Après :   AX :   9042   BX :   0A8F

XCHG mem8,AL     Avant :   AL :   4A   mem :   0F
                 Après :   AL :   0F   mem :   4A
```

Exception :

Memory();

XLAT/XLATB

Translation avec table

Formats :

XLATB
XLAT mem8

Description :

Au début de l'instruction, AL est censé représenter l'index non signé de la table dont la base est à [EBX]. XLAT charge AL avec [EBX+AL]. AL est toujours considéré comme une valeur non signée. La base de la table est toujours [EBX]. Cependant, mem8 permet d'utiliser un recouvrement de segment plutôt que la valeur par défaut DS:[EBX].

Fonctionnement :

```
if (CODEOP == XLATB) {
      AL = DS:[EBX + ConCat(000000h,AL)];
} else {
      AL = SegReg(mem8):[EBX + ConCat(000000h,AL)];
}
```

Exemple :

```
XLATB   Avant :   EBX      :   F000   AL :   30
                  [EBX+AL] :   7D
        Après :   EBX      :   F000   AL :   7D
```

Exception :

Memory();

XOR

OU exclusif logique

Formats :

XOR	op_1	,	op_2
	reg	,	reg
	reg	,	mem
	reg	,	imm
	mem	,	reg
	mem	,	imm

Description :

L'instruction effectue un OU (OR) exclusif logique entre op_1 et op_2 et place le résultat dans op_1.

Fonctionnement :

```
op1 = op1 ^ op2;
OF = 0;
SF = B(MSB,op1);
ZF = op1 == 0;
AF = UNDEFINED;
PF = ~(B(7,op1)^B(6,op1)^...^B(0,op1));
CF = 0;
/*
   ^ est défini par la table logique suivante.
   ai et bi sont les ièmes bits de op1 et op2,.
   ri est le ième bit du résultat.
   La variable binaire i prend les valeurs 0 à MSB.
   ai  bi     ri
   0   0      0
```

```
   0   1      1
   1   0      1
   1   1      0
*/
```

Exemples :

```
                                                          OSZAPC
XOR EAX,EBX   Avant :   EAX :   01234567   EBX :   F0F0F0F0   XXXXXX
              Après :   EAX :   F1D3B597   EBX :   F0F0F0F0   010X00

XOR mem8,70   Avant :   Mem :   7F                            XXXXXX
              Après :   Mem :   0F                            000X10
```

Exception :

Memory();

SEGMENTS MULTIPLES

Dans le Chapitre 2, nous avons parlé de la segmentation. Vous vous rappelez donc que le système d'exploitation peut dicter un modèle de programmation à segment unique (tous les segments sont regroupés en un segment unique) ou permettre un modèle de programmation à segments multiples. Ce choix est laissé au programmeur. Ceci est vrai sauf pour le traitement des interruptions et des exceptions qui obligent les programmeurs système à employer le modèle à segments multiples. Nous décrirons de façon précise les interruptions et les exceptions dans le Chapitre 6.

Si vous avez choisi d'employer le modèle à segments multiples, cette partie qui décrit des instructions 80386 particulières vous sera utile. Ces instructions permettent de gérer le modèle de programmation à segments multiples. Sinon vous pouvez laisser cette partie de côté.

Dans les Chapitres 5 et 7, nous expliquerons comment employer ces instructions à travers quelques exemples. Leur description fera donc souvent référence au Chapitre 5.

Toutes les instructions de cette partie traitent du chargement des sélecteurs dans les registres segments. Dans le Chapitre 2, nous avons déjà décrit les sélecteurs et les registres segments. Le chargement d'un sélecteur de segment dans un registre segment rend le segment adressable. On peut faire référence à une donnée dans un segment par le déplacement associé au registre segment (voir Chapitre 2). Le segment code est implicite pour toutes les instructions, et le registre EIP donne le déplacement dans ce segment.

Définitions des fonctions

Au début de ce chapitre, nous avons défini plusieurs fonctions qui nous ont permis d'abréger les descriptions des interruptions et des exemples. Nous allons donner une routine similaire. La définition qui va suivre est propre à cette partie de la description du jeu d'instructions.

```
/* La routine IntExecTable retourne soit le Segment            */
/* soit le Déplacement de la routine de traitement             */
/* des interruptions qui a été appelée. La syntaxe             */
/* normale de gestion d'une interruption est plus              */
/* compliquée (guichets, commutations de tâches, changement    */
/* de niveau de privilège) et le Chapitre 6 l'explique         */
/* plus précisément. Un appel à cette routine serait :         */
/* Interrupt(imm,1)                        */
/* où le second opérande indique que les tests sur les niveaux */
/* de privilège doivent être effectués et où le premier        */
/* opérande est une constante immédiate passée par l'appel     */
/* à IntTable.                                                 */

IntExecTable(type,imm){
      if (type==SEGMENT){
         return((word) IDTR.Base+4);
      } else {       /* OFFSET */
         return((dword) IDTR.Base+0);
      }
}
```

Routines d'exception

Au début de ce chapitre, nous avons défini plusieurs routines d'exception qui nous ont permis d'abréger la description des conditions d'exception. Nous allons donner plusieurs routines d'exception que nous avions omises, puisqu'elles sont spécifiques aux instructions à segments multiples.

```
CodeSegmentLoad(){
      /* Toutes les exceptions segments possibles              */
      /* lors du chargement d'un descripteur CS sont           */
      /* contenues dans la routine CSDescriptorLoad()          */
      /* définie au Chapitre 5. Se reporter à ce chapitre      */
      /* pour plus de détails.                                 */
}
SegmentLoad(){
      /* Toutes les exceptions segments possibles              */
      /* lors du chargement d'un descripteur sont              */
      /* contenues dans la routine DescriptorLoad()            */
      /* définie dans le Chapitre 5. Se reporter à ce chapitre */
      /* pour plus de détails.            */
}
```

Résumé du jeu d'instructions

Le tableau suivant résume les instructions qu'on trouvera dans cette partie de la description du jeu d'instructions.

Instructions à segments multiples

CALL	Appel de procédure
INT	Appel à une procédure d'interruption
INTO	Appel à une procédure d'interruption sur débordement
IRET	Retour d'une procédure d'interruption
JMP	Branchement
LDS	Chargement du pointeur dans DS
LES	Chargement du pointeur dans ES
LFS	Chargement du pointeur dans FS
LGS	Chargement du pointeur dans GS
LSS	Chargement du pointeur dans SS
MOV	Transfert de registre segment
POP	Dépilage dans le registre segment
PUSH	Empilage
RET	Retour d'une procédure

CALL

Appel de procédure

Formats :

CALL op_1
far_label
mem48

Description :

CALL effectue un branchement à l'instruction située au déplacement précisé dans le nouveau segment code. Le pointeur donné (un pointeur 48 bits composé d'un sélecteur de segment 16 bits et d'un déplacement 32 bits ; voir le Chapitre 2) pointe sur une procédure d'un segment différent (appel intersegment).

Avant que le transfert s'effectue, le pointeur 48 bits sur la prochaine instruction exécutable est sauvegardé dans la pile. L'instruction RET pourra employer cette information.

Notez que la valeur de EIP sauvegardée dans la pile pointe sur l'instruction qui suit directement le CALL. Le 80386 traite une instruction, puis, avant son exécution, incrémente le pointeur d'instruction. Ainsi, on peut sauvegarder directement EIP.

Notez aussi que lorsque CS est sauvegardé dans la pile, c'est un mot de 32 bits qui est empilé, afin de conserver l'alignement de la pile (voir le Chapitre 2). Les 16 bits de poids fort sont alors indéfinis. L'alignement de la pile améliore les performances du 80386.

Fonctionnement :

```
/* far_label (étiquette lointaine) est assemblée
     en un pointeur 48 bits composé d'un sélecteur
     de segment code 16 bits (donné par la routine
     Segment) et d'un déplacement d'instruction
     à partir du début du segment (donné par la
     routine Offset).
*/
if (op1 is far_label) {   /*   */
      ESP = ESP - 4;
      [ESP] = CS;
      CS = Segment(far_label);
      ESP = ESP - 4;
      [ESP] = EIP;
      EIP = Offset(far_label);
} else if (op1 is mem48) {   /*   */
      ESP = ESP - 4;
      [ESP] = CS;
      CS = [&mem+4];
      ESP = ESP - 4;
      [ESP] = EIP;
      EIP = [&mem];
}
/*
        Puisque l'instruction d'appel (far call)
      charge le registre CS, ce qui peut entraîner
      des sauts de niveaux de privilège, des passages
      de guichets, et des commutations de tâches,
      on ne peut pas décrire exactement tous les
      effets de l'instruction ici. On en reparlera
      plus précisément dans le Chapitre 5.
*/
```

Exemples :

```
CALL FAR label    ;label = 6320 :0F03298A
       Avant :   EIP     :   00000300      ESP    :   00000200   CS :   5028
       Après :   EIP     :   0F03298A      ESP    :   00000198   CS :   6320
                 [ESP+4] :   XXXX5028      [ESP]  :   00000300
RET
       Avant :   EIP     :   0F03298A      ESP    :   00000198   CS :   6320
                 [ESP+4] :   XXXX5028      [ESP]  :   00000300
       Après :   EIP     :   00000300      ESP    :   00000200   CS :   5028
```

Exceptions :

Memory();
Stack();
CodeSegmentLoad();
CodeReference();

Note :

Pour plus de précisions, voir la description de la page 713.

Se reporter aussi à la forme intrasegment de l'instruction CALL (page 187).

INT

Appel à une procédure d'interruption

Formats :

INT imm8

Description :

Une instruction INT est un appel logiciel à une routine de traitement d'exception. imm8, qui est le numéro de l'exception, précise quelle procédure d'interruption particulière doit être appelée.

Le registre étendu des indicateurs, le registre de segment code, et le pointeur d'instruction sont empilés. Le contrôle est donné à la routine de traitement des exceptions (*handler*) en accord avec le numéro de l'exception. A la fin du traitement de l'interruption, on emploie généralement une instruction IRET pour rendre la main au programme appelant.

Notez aussi que lorsque CS est sauvegardé dans la pile, c'est un mot de 32 bits qui est empilé, afin de conserver l'alignement de la pile (voir le Chapitre 2). Les 16 bits de poids fort sont alors indéfinis. L'alignement de la pile améliore les performances du 80386.

On peut employer cette instruction comme "appel" au système d'exploitation.

Fonctionnement :

```
ESP = ESP - 4;
[ESP] = EFLAGS;
ESP = ESP - 4;
```

```
[ESP] = CS;    /*    */
ESP = ESP - 4;
[ESP] = EIP;
CS =  IntExecTable(SEGMENT,imm8);
EIP = IntExecTable(OFFSET, imm8);
IF = 0;
TF = 0;
NT = 0;
/*
        Puisque l'instruction INT
      charge le registre CS, ce qui peut entraîner
      des sauts de niveaux de privilège, des passages
      de guichets, et des commutations de tâches,
      on ne peut pas décrire exactement tous les
      effets de l'instruction ici. On en reparlera
      plus précisément dans les Chapitres 5 et 6.
*/
```

Exemples :

```
INT 5
   Avant :  ESP      :   FFFFF388  EIP      :   0000FA04
            EFLAGS   :   00004302  CS       :   F244
            IntExecTable(SEGMENT,5)          :   0400
            IntExecTable(OFFSET,5)           :   004AFDDC
   Après :  ESP      :   FFFFF37C  EIP      :   004AFDDC
            EFLAGS   :   00000002  CS       :   0400
            [ESP+8]  :   00004302  [ESP+4]  :   XXXXF244
            [ESP]    :   0000FA04
```

Exceptions :

Stack();
CodeSegmentLoad();
CodeReference();

Note :

L'instruction INT 3 est une forme spéciale à octet unique (l'instruction point d'arrêt), qui est utile pour la mise au point. Voir le Chapitre 8 pour plus de précisions.

INTO

Appel à procédure d'interruption sur débordement

Format :

INTO

Description :

Une instruction INTO est un appel logiciel conditionnel à la procédure d'interruption 4. Cet appel s'effectue si l'indicateur de débordement est positionné.

L'instruction empile le registre étendu des indicateurs, le registre de segment code, et le pointeur d'instruction, puis donne le contrôle à la routine de traitement des exceptions (*handler*) en accord avec le numéro de l'exception. A la fin du traitement de l'interruption, on emploie généralement une instruction IRET pour rendre la main au programme appelant.

Notez aussi que lorsque CS est sauvegardé dans la pile, c'est un mot de 32 bits qui est empilé, afin de conserver l'alignement de la pile (voir le Chapitre 2). Les 16 bits de poids fort sont alors indéfinis. L'alignement de la pile améliore les performances du 80386.

Fonctionnement :

```
if (OF==0){        /*    */
      ESP = ESP - 4;
      [ESP] = EFLAGS;
      ESP = ESP - 4;
      [ESP] = CS;
      ESP = ESP - 4;
      [ESP] = EIP;
```

```
        CS =  IntExecTable(SEGMENT,4);
        EIP = IntExecTable(OFFSET, 4);
IF = 0;
TF = 0;
NT = 0;
}
/*
          Puisque l'instruction INTO
        charge le registre CS, ce qui peut entraîner
        des sauts de niveaux de privilège, des passages
        de guichets, et des commutations de tâches,
        on ne peut pas décrire exactement tous les
        effets de l'instruction ici. On en reparlera
        plus précisément dans les Chapitres 5 et 6.
*/
```

Exemples :

```
INTO
   Avant : ESP      :   FFFFF388  EIP      :   0F822DDB
           EFLAGS   :   00006FA7      CS   :   0008
           IntExecTable(SEGMENT,4)         :   4A00
           IntExecTable(OFFSET,4)          :   004AFDDC
   Après : ESP      :   FFFFF37C  EIP      :   004AFDDC
           EFLAGS   :   00002CA7  CS       :   4A00
           [ESP+8]  :   00006FA7  [ESP+4]  :   XXXX0008
           [ESP]    :   0F822DDB
```

Exceptions :

Stack();
CodeSegmentLoad();
CodeReference();

IRET

Retour d'une procédure d'interruption

Format :

IRET

Description :

L'instruction IRET rend la main après l'appel à une procédure d'interruption. IRET dépile les registres EIP, CS, et EFLAGS, puis relance l'exécution à l'endroit où cette dernière avait été interrompue. Ainsi, on emploie IRET à la fin d'une routine de traitement des interruptions pour rendre le contrôle au programme interrompu.

Notez aussi que lorsque CS est sauvegardé dans la pile, c'est un mot de 32 bits qui est empilé afin de conserver l'alignement de la pile (voir le Chapitre 2). Les 16 bits de poids fort sont alors indéfinis. L'alignement de la pile améliore les performances du 80386.

Fonctionnement :

```
EIP = [ESP];
ESP = ESP + 4;
CS = [ESP];    /*    */
ESP = ESP + 4;
EFLAGS = [ESP];
ESP = ESP + 4;
/*
        Puisque l'instruction IRET
      charge le registre CS, ce qui peut entraîner
      des sauts de niveaux de privilège, des passages
      de guichets, et des commutations de tâches,
```

```
        on ne peut pas décrire exactement tous les
        effets de l'instruction ici. On en reparlera
        plus précisément dans les Chapitres 5 et 6.
*/
```

Exemples :

```
IRET
   Avant : ESP      :  FFFFF37C      EIP      :  004B0245
           EFLAGS   :  00000002      CS       :  0400
           [ESP+8]  :  00004302      [ESP+4]  :  XXXXF244
           [ESP]    :  0000FA04
   Après : ESP      :  FFFFF388      EIP      :  0000FA04
           EFLAGS   :  00004302      CS       :  F244
```

Exceptions :

Stack();
CodeSegmentLoad();
CodeReference();

Note :

Voir l'exemple 1 à la page 544. Voir aussi page 779.

JMP

Branchement

Formats :

JMP op_1
far_label
mem48

Description :

L'instruction de branchement (ou saut) permet de continuer l'exécution à un déplacement déterminé d'un autre segment. Le pointeur donné (un pointeur 48 bits composé d'un sélecteur 16 bits et d'un déplacement 32 bits (voir le Chapitre 2)) pointe sur une étiquette d'un segment différent (saut intersegment).

On peut déterminer directement le pointeur 48 bits par une étiquette lointaine (far_label) ou par un opérande mémoire (mem48).

Fonctionnement :

```
/* far_label (étiquette lointaine) est assemblée
     en un pointeur 48 bits composé d'un sélecteur
     de segment code 16 bits (donné par la routine
     Segment) et d'un déplacement d'instruction
     à partir du début du segment (donné par la
     routine Offset).
*/
if (op1 is far_label) {
      CS = Segment(far_label);
      EIP = Offset(far_label);
} else if (op1 is mem48) {    /*    */
```

```
        CS = [&mem+4];
        EIP = [&mem];
}
/*
          Puisque l'instruction d'appel (far call)
        charge le registre CS, ce qui peut entraîner
        des sauts de niveaux de privilège, des passages
        de guichets, et des commutations de tâches,
        on ne peut pas décrire exactement tous les
        effets de l'instruction ici. On en reparlera
        plus précisément dans le Chapitre 5.
*/
```

Exemple :

```
JMP FAR label1        ; label1 = AC40 :000000FB
                Avant :   EIP :   00000300   CS :   2DF0
                Après :   EIP :   000000FB   CS :   AC40
```

Exceptions :

Memory();
CodeSegmentLoad();
CodeReference();

Note :

Voir les exemples aux pages 709 et 844.

Lsr

Chargement d'un pointeur

Formats :

```
Lsr   reg   ,  mem48
LDS   reg   ,  mem48   ; Chargement d'un pointeur dans DS
LES   reg   ,  mem48   ; Chargement d'un pointeur dans ES
LFS   reg   ,  mem48   ; Chargement d'un pointeur dans FS
LGS   reg   ,  mem48   ; Chargement d'un pointeur dans GS
LSS   reg   ,  mem48   ; Chargement d'un pointeur dans SS
```

Description :

Une instruction Lsr charge un pointeur 48 bits dans une paire registre segment/registre général. Le dmot situé à l'adresse mem48 est chargé dans le registre déterminé par reg. Le mot situé à l'adresse (mem48+4) est chargé dans le registre segment déterminé par le mnémonique de l'instruction (DS, ES, FS, GS, ou SS). Ainsi, mem48 détermine un pointeur 48 bits ; le premier dmot est un déplacement (qui est chargé dans le registre précisé), et le dernier mot est un descripteur de segment (qui est chargé dans le registre segment précisé).

Charger un registre segment avec un sélecteur rend le segment adressable (voir le Chapitre 2). Dans le Chapitre 5, nous donnerons plus de détails sur le chargement des registres segments.

Une instruction Lsr ne peut pas charger CS (LCS). Seules des instructions de contrôle de transfert intersegment le peuvent (CALL far_label, IRET, JMP far_label, INT).

Fonctionnement :

```
reg = (dword)[&mem48];
sr  = (mot)  [&mem48+4];
```

Exemple :

```
LGS EAX,mem48   Avant :   EAX   :   XXXXXXXX      GS :   XXXX
                          mem48 :   8034 :0F4BC319
                Après :   EAX   :   0F4BC319      GS :   8034
                          mem48 :   8034 :0F4BC319
```

Exceptions :

Memory();
SegmentLoad();

Notes :

LSS, LFS, et LGS sont des nouveautés du 80386.

Employez :

LSS ESP,mem48

pour pointer sur une nouvelle pile. Puisque cette instruction assigne à la fois SS et ESP, tout problème potentiel qui serait lié à des assignations par deux instructions distinctes est éliminé.

Voir l'exemple 1, page 544.

MOV

Transfert de registre segment

Formats :

MOV	op_1	,	op_2
	sreg	,	reg16
	sreg	,	mem16
	reg16	,	sreg
	mem16	,	sreg

Description :

Cette instruction copie le contenu de op_2 dans op_1.

Si op_1 est un registre segment, l'instruction rend adressable un nouveau segment en chargeant son sélecteur dans sreg. Pour accéder à des données dans un segment, on doit charger un sélecteur, qui identifie ce segment, dans un des registres segments.

Si op_1 est SS, les interruptions sont inhibées jusqu'à la fin de l'instruction suivante.

Dans le Chapitre 5, nous traiterons du chargement des registres segments, lors de la description de la segmentation.

Fonctionnement :

```
op1 = op2;
```

Exemples :

```
MOV AX, GS    Avant :   AX :   XXXX   GS :   F433
              Après :   AX :   F433   GS :   F433

MOV AX,Segment1    ; Chargement du sélecteur de Segment1 dans AX
MOV GS,AX          ; Puis, le sélecteur est transféré dans GS.
ADD GS :sum,DX     ; La donnée située à 'sum' dans le segment GS
                   ; (Segment1) est maintenant adressable
                   ; et utilisée par l'instruction ADD
```

Exceptions :

SegmentLoad();
Memory();

Notes :

op_1 ne peut pas être CS.

Voir la page 690.

POP

Dépilage dans le registre segment

Formats :

POP	op_1
	DS
	ES
	SS
	FS
	GS

Description :

POP transfère un mot (sélecteur) de la pile dans un registre segment déterminé (DS, ES, FS, ou GS). Charger le registre segment rend le segment adressable par le sélecteur au sommet de la pile. Le sommet de la pile est pointé par SS:[ESP]. Après le transfert de la pile vers op_1, le pointeur de pile est automatiquement incrémenté de 4.

Comme on l'a vu, le chargement du registre segment peut occasionner le chargement d'un descripteur de segment. Dans le Chapitre 5, nous décrirons la sémantique du chargement du registre segment.

POP SS inhibe les interruptions jusqu'à ce que l'instruction suivante soit exécutée.

Donner comme opérande CS dans une instruction POP est illégal.

Fonctionnement :

```
op1 = SS:[ESP];
ESP = ESP + 4;
```

Exemple :

```
POP ES   Avant :   ES :   XXXX   SS : [ESP]   :   F340   ESP :   F4320530
         Après :   ES :   F430   SS : [ESP-4] :   F340   ESP :   F4320534
```

Exceptions :

SegmentLoad();
Stack();

PUSH

Empilage

Formats :

PUSH op_1
DS
ES
CS
SS
FS
GS

Description :

Cette instruction place op_1 au nouveau sommet de la pile. On obtient ce dernier en décrémentant de 4 le pointeur de pile.

Fonctionnement :

```
SS : [ESP-2] = op1;
ESP = ESP - 4;
```

Exemple :

PUSH SS	Avant :	SS :	8042	ESP :	0424	[SS :420] :	XXXX
	Après :	SS :	8042	ESP :	0420	[SS :420] :	8042

Exception :

Stack();

RET

Retour d'une procédure

Formats :

RET
RET imm16

Description :

RET permet de rendre le contrôle à l'instruction suivant directement l'instruction CALL correspondante. RET emploie le pointeur de retour (un pointeur 48 bits composé de CS et EIP voir le Chapitre 2) ; qu'elle trouve au sommet de la pile. Le pointeur est dépilé dans les registres EIP et CS. Une autre façon de considérer l'instruction RET est de remarquer qu'elle effectue un saut indirect intersegment (far) via le pointeur mémoire à SS:[ESP] plus la mise à jour appropriée de la pile.

On peut préciser une valeur optionnelle imm16 dans l'instruction RET. Cette valeur est ajoutée à ESP pour dépiler plusieurs octets de paramètres.

Notez aussi que lorsque CS est sauvegardé dans la pile, c'est un mot de 32 bits qui est empilé afin de conserver l'alignement de la pile (voir le Chapitre 2). Les 16 bits de poids fort sont alors indéfinis. L'alignement de la pile améliore les performances du 80386.

Fonctionnement :

```
EIP = [ESP];
ESP = ESP + 4;
CS = [ESP];    /*    */
ESP = ESP + 4;
```

```
if (NUMOPS == 1){
      ESP = ESP + imm16;
}
/*
        Puisque l'instruction RET
      charge le registre CS, ce qui peut entraîner
      des sauts de niveaux de privilège, des passages
      de guichets, et des commutations de tâches,
      on ne peut pas décrire exactement tous les
      effets de l'instruction ici. On en reparlera
      plus précisément dans les Chapitres 5 et 6.
*/
```

Exemples :

```
FAR RET
   Avant :   EIP     :   0F03429A     ESP   :   00000198   CS :   6320
             [ESP+4] :   XXXX5028     [ESP] :   00000300
   Après :   EIP     :   00000300     ESP   :   00000200   CS :   5028
```

Exceptions :

Stack();
CodeSegmentLoad();
CodeReference();

Note :

Voir page 722 pour une description plus détaillée.

SYSTEME D'EXPLOITATION

Comme nous l'avons déjà mentionné plusieurs fois au cours de cet ouvrage, l'architecture matérielle du 80386 lui apporte plusieurs fonctions évoluées pour faciliter la mise en oeuvre d'un système d'exploitation. Pour tirer avantage de cette architecture matérielle, les programmeurs de systèmes d'exploitation disposent des instructions nécessaires à l'implantation de systèmes d'exploitation multitâches, multi-utilisateurs, et à mémoire virtuelle.

Dans cette partie, nous allons décrire les instructions qui sont explicitement mises à la disposition du programmeur de systèmes d'exploitation. Si vous êtes un programmeur d'applications, vous pouvez vous passer de cette partie de la description du jeu d'instructions. En fait, la plupart des systèmes d'exploitation ne permettent pas aux programmeurs d'applications d'utiliser ces instructions.

Toute justification de l'existence de ces instructions se trouve dans le Chapitre 5, où nous vous expliquons pourquoi ce jeu d'instructions est nécessaire. Ainsi, nous vous recommandons de lire le Chapitre 5 avant cette partie. Après cette lecture, les instructions liées aux systèmes d'exploitation vous sembleront plus claires et plus faciles à comprendre. Cependant, nous incluons à ce stade de l'ouvrage la description complète de ces instructions.

Toutes ces instructions peuvent s'exécuter à n'importe quel niveau de privilège, sauf si elles sont liées à une exception Level(0) (niveau de privilège 0).

Définitions de fonctions

Au début de ce chapitre, nous avons défini plusieurs fonctions qui nous ont permis d'abréger les descriptions des instructions et des exemples. C'est pour des raisons similaires que nous donnons la routine ci-après.

Nous la définissons maintenant car son usage est limité à cette partie de la description du jeu d'instructions.

```
/* La routine FetchDescriptor() est définie plus          */
/* précisément dans le Chapitre 5, ainsi que la structure */
/* des attributs.                                         */
Descr(num,op){
      FetchDescriptor(op,&dword1,&dword2,&attributes);
      if (num == 1) return(dword1);
      else          return(dword2);
}
```

Routines d'exception

Au début du chapitre, nous avons défini plusieurs routines d'exception qui nous ont permis d'abréger la description des conditions d'exception. Nous allons maintenant donner plusieurs routines que nous n'avions pas définies au début du chapitre, puisqu'elles sont particulières aux instructions de cette partie.

```
SegmentLoad(){
      /* Toutes les exceptions segments possibles              */
      /* lors du chargement d'un descripteur sont              */
      /* contenues dans la routine DescriptorLoad()            */
      /* définie dans le Chapitre 5. Voir le Chapitre 5 pour   */
      /* plus d'informations.                                  */
}
Level0(){
      /* CPL est défini dans le Chapitre 5.                   */
      /* La routine SegmentException() est définie             */
      /* dans le Chapitre 6.                                   */
      if (CPL != 0){
         SegmentException($GP,0);
      }
}
```

```
8086Mode(){
      /* 8086Mode() signale une exception InvalidCodeOp()        */
      /* si le 80386 est en train de travailler                  */
      /* en mode réel ou virtuel 8086,                           */
      /* (voir le Chapitre 9). Le processeur est en mode         */
      /* réel si le bit PE dans CR0 est à 0. Le                  */
      /* processeur est en mode virtuel 8086 si                  */
      /* PE est à 1 et que le bit VM du registre                 */
      /* EFLAGS est à 1                                          */

if ( (CR0 & 1b == 0)          /* mode réel */
      || ( (CR0 & 1b) ==1) &&
         (EFLAGS.VM == 1) )    /mode virtuel 8086 */ )
      InvalidCodeOp();
}
```

Résumé du jeu d'instructions

Le tableau suivant résume les instructions qu'on trouvera dans cette partie de la description du jeu d'instructions.

Instructions liées au système d'exploitation

ARPL	Ajustement du niveau de privilège requis
CLTS	Effacement de l'indicateur de commutation de tâches
HLT	Halte
LAR	Chargement des droits d'accès
LGDT	Chargement de la table des descripteurs globaux
LIDT	Chargement de la table des descripteurs d'interruptions
LLDT	Chargement de la table des descripteurs locaux
LMSW	Chargement du mot d'état machine
LSL	Chargement de la limite de segment
LTR	Chargement du registre de tâche
MOV	Transfert de registre spécial
SGDT	Stockage de la table des descripteurs globaux
SIDT	Stockage de la table des descripteurs d'interruptions
SLDT	Stockage de la table des descripteurs locaux

SMSW	Stockage du mot d'état machine
STR	Stockage du registre de tâche
VERR	Vérification du segment en lecture
VERW	Vérification du segment en écriture

ARPL

Ajustement du niveau de privilège requis

Formats :

ARPL	op_1	,	op_2
	reg16	,	reg16
	mem16	,	reg16

Description :

ARPL teste le RPL (niveau de privilège requis) de op_1 en le comparant avec celui de op_2. Le RPL est déterminé par les deux bits de poids faible de op_1 et de op_2. Si le RPL de op_1 est plus petit que celui de op_2, l'indicateur de zéro est positionné à 1 et les deux bits de poids faible de op_1 deviennent égaux aux deux bits de poids faible de op_2. op_1 ou op_2 peuvent être le sélecteur nul.

Les programmes d'application n'emploient pas ARPL. Normalement, seul un système d'exploitation utilise cette instruction. Généralement, op_2 est le sélecteur du segment code du programme appelant. op_1 est le sélecteur de segment d'une routine à laquelle le programme appelant veut accéder. Cette comparaison est effectuée pour s'assurer que le programme appelant ne requiert pas plus de privilège qu'il n'en a le droit.

Fonctionnement :

```
if ((op1 & 03h) < (op2 & 03h)){
      ZF = 1;
      op1 = (op1 & FFFCh) | (op2 & 03h);
} else {
      ZF = 0;
}
```

Exemples :

```
                                                OSZAPC
ARPL mem16,AX
      Avant :   mem16 :   6DD3   AX :   456A   XXXXXX
      Après :   mem16 :   6DD3   AX :   456A   UU0UUU

ARPL CX,AX
      Avant :   CX    :   FF44   AX :   62F1   XXXXXX
      Après :   CX    :   FF45   AX :   6251   UU1UUU
```

Exceptions :

Memory();
8086Mode();

CLTS

Effacement de l'indicateur de commutation de tâches

Formats :

CLTS

Description :

Cette instruction met à 0 l'indicateur de commutation de tâches du mot d'état machine (MSW ou CR0). Chaque fois qu'il y a commutation de tâches, cet indicateur est automatiquement positionné à 1 (voir le Chapitre 5 pour plus d'informations sur la commutation de tâches). On emploie l'indicateur TS pour réduire les problèmes de changement de contexte lorsqu'on se trouve en présence d'un 80387. Si une commutation de tâches s'effectue (TS = 1) et si la nouvelle tâche tente d'utiliser le 80387, il y aura génération d'une faute. La routine d'erreur correspondante doit ensuite sauver le contexte du 80387 et exécuter l'instruction CLTS. La nouvelle tâche peut alors employer le 80387 sans détruire une seule donnée de la tâche précédente.

Ainsi, seules les tâches qui utilisent le 80387 entraîneront la sauvegarde du contexte de 80387.

CLTS apparaît seulement dans des programmes de systèmes d'exploitation. On ne peut l'exécuter qu'au niveau de privilège 0.

Fonctionnement :

```
TS = 0;
```

Exemple :

```
CLTS   Avant :   TS :   1
       Après :   TS :   0
```

Exception :

Level0();

HLT

Halte

Format :

HLT

Description :

HLT entraîne l'arrêt du 80386. La poursuite de l'exécution ne s'effectue alors que si le 80386 reçoit une interruption autorisée (notez que NMI est toujours autorisée) ou si on effectue une réinitialisation de l'ordinateur. Nous parlerons plus précisément des interruptions et de la façon dont elles peuvent être masquées dans le Chapitre 6.

Exemple :

```
HLT    ;    l'exécution s'arrête
```

Exception :

Level0();

LAR

Chargement des droits d'accès

Formats :

LAR	op_1	,	op_2
	reg16	,	reg16
	reg16	,	mem16
	reg32	,	reg32
	reg32	,	mem32

Description :

LAR charge le premier opérande avec le champ des attributs du segment (droits d'accès) à partir du descripteur du segment déterminé par le sélecteur du second opérande.

Le champ des attributs du segment est représenté simplement par les quatre octets de poids fort (descripteur ET 00FxFF00h), où x indique que les bits 16 à 19 sont indéfinis lors de ce chargement.

Le descripteur déterminé par op_1 doit être compris dans les limites de la table des descripteurs, avoir un champ Type valide, et être accessible à la fois par le CPL et par le RPL du sélecteur dans op_2 comparé au DPL. S'il en est ainsi, ZF est mis à 1 et op_1 est chargé avec le champ des attributs du segment. Sinon, ZF est mis à 0 et op_1 n'est pas modifié.

Tout descripteur de segment mémoire (noté par un 1 dans le champ DType) peut avoir n'importe quelle valeur dans son champ Type et pourtant être valide pour LAR. Nous allons donner les types valides pour des segments système et des guichets (DType = 0) dans le tableau suivant :

Type de descripteur	Valide ?
0 Indéfini	Non
1 TSS 286 disponible	Oui
2 LDT	Oui
3 TSS 286 occupé	Oui
4 Guichet d'appel 286	Oui
5 Guichet de tâche	Oui
6 Guichet d'interruption 286	Non
7 Guichet de trappe 286	Non
8 Indéfini	Non
9 TSS 386 disponible	Oui
A Indéfini	Non
B TSS 386 occupé	Oui
C Guichet d'appel 386	Oui
D Indéfini	Non
E Guichet d'interruption 386	Non
F Guichet de trappe 386	Non

Fonctionnement :

```
if ( (op2& 0FFFch) == 0    /* Sélecteur nul */ )
      goto ClearZF;

/* La routine FetchDescriptor est définie dans le Chapitre 5. */
/* Retourne 0 si le descripteur est au-delà des limites de la table
*/
if (FetchDescriptor(op2,&dword1,&dword2,&Attributes) == 0)
      goto ClearZF;

/* Sinon le descripteur est compris dans les limites de la table */
if (Attributes.DType == 1    /* segment mémoire */ )
      switch (Attributes.Type){
         case 0: case 1:       /* Lecture */
         case 2: case 3:       /* Lecture/Ecriture */
         case 4: case 5:       /* Lecture, expand_down */
         case 6: case 7:       /* Lecture/Ecriture, expand_down */
```

```
            case 8: case 9:       /* Exécution */
            case 10: case 11:   /* Exécution/Lecture */
               /* Teste le privilège seulement pour des segments */
               /* non conformes */
               if ( (Attributes.DPL < CPL) ||
                    (Attributes.DPL < op2.RPL))
                  goto ClearZF;
               break;
               /* On ignore DPL pour un segment conforme */
            case 12: case 13:   /* Exécution, conforme */
            case 14: case 15:   /* Exécution/Lecture, conforme */
               break;
         } /* fin du switch */

else   /* DType == 0, segment système ou guichet */
         switch(Attributes.Type){
            Case 1:   /* TSS 286 Disponible */
            Case 2:   /* LDT                */
            Case 3:   /* TSS 286 Occupé     */
            Case 4:   /* Guichet appel 286  */
            Case 5:   /* Guichet de tâche   */
            Case 9:   /* TSS 386 Disponible */
            Case 11:   /* TSS 386 Occupé     */
            Case 12:   /* Guichet appel 386  */
               /* Type est valide, teste DPL */
               if ( Attributes.DPL < CPL ||
                    Attributes.DPL < op2.RPL )
                  goto ClearZF;
               break;

            Default:   /* Autres types non valides */
                  goto ClearZF;
         } /* fin du switch */

/* On arrive ici seulement si tous les tests sont OK */
/* Met ZF à 1, charge les Attributs dans op1        */
         ZF = 1;
         op1 = Attributes & 0FxFF00h;
         return;

/* Saut à ClearZF si quelque chose s'est mal passé  */
```

```
/* Met ZF à 0 et ne change pas op1                    */
ClearZF:
      ZF = 0;
      return;
```

Exemples :

```
                                                  OSZAPC
LAR EAX,mem32    ; Descripteur visible
      Avant :    Descr(0,mem32) :   1F0FACFE      XXXXXX
                 EAX            :   XXXXXXXX
      Après :    Descr(0,mem32) :   1F0FACFE      XX1XXX
                 EAX            :   000XAC00

LAR EAX,mem32    ; Descripteur non visible
      Avant :    Descr(0,mem32) :   1F0FACFE      XXXXXX
                 EAX            :   XXXXXXXX
      Après :    Descr(0,mem32) :   1F0FACFE      XX0XXX
                 EAX            :   UUUUUUUU
```

Exceptions :

Memory();
8086Mode();

LGDT

Chargement de la table des descripteurs globaux

Formats :

LGDT op_1
mem48

Description :

LGDT charge le registre de la table des descripteurs globaux à partir du pseudo-descripteur 48 bits (mem48) donné dans l'instruction. Ce dernier possède deux composantes : la limite et la base. La limite 16 bits est stockée dans le mot de poids faible, et la base 32 bits dans le dmot de poids fort.

LGDT apparaît seulement dans du code de système d'exploitation ; on ne devrait jamais la trouver dans des programmes d'application.

Fonctionnement :

```
GDTR.Limit = mem48;          /* limite GDT 16 bits */
GDTR.Base  = [&mem48+2]      /* base GDT 32 bits */
```

Exemple :

```
LGDT mem48
      Avant :   mem48       :   4132    [&mem48+2] :   AC405B10
                GDTR.Limit  :   XXXX    GDTR.Base  :   XXXXXXXX
```

```
Après :   mem48       :   4132   [&mem48+2] :   AC405B10
          GDTR.Limit  :   4132   GDTR.Base  :   AC405B10
```

Exceptions :

Memory();
Level0();

Note :

Voir la page 843.

LIDT

Chargement de la table des descripteurs d'interruptions

Format :

LIDT op_1
mem48

Description :

LIDT charge le registre de la table des descripteurs d'interruptions à partir du pseudo-descripteur 48 bits (mem48) donné dans l'instruction. Ce dernier possède deux composantes : la limite et la base. La limite 16 bits est stockée dans le mot de poids faible, et la base 32 bits dans le dmot de poids fort.

LIDT apparaît seulement dans du code de système d'exploitation ; on ne devrait jamais la trouver dans des programmes d'application.

Fonctionnement :

```
IDTR.Limit = mem48;          /* limite IDT 16 bits */
IDTR.Base  = [&mem48+2]      /* base IDT 32 bits */
```

Exemple :

```
LIDT mem48
      Avant :   mem48       :   AAC3    [&mem48+2] :   0BF4103D
                IDTR.Limit  :   XXXX    IDTR.Base  :   XXXXXXXX
```

```
Après : mem48      :  AAC3   [&mem48+2] :  0BF4103D
        IDTR.Limit :  AAC3   IDTR.Base  :  0BF4103D
```

Exceptions :

Memory();
Level0();

Note :

Voir les pages 836 et 837.

LLDT

Chargement de la table des descripteurs locaux

Formats :

LLDT op_1
reg16
mem16
reg32
mem32

Description :

LLDT charge le registre de la table des descripteurs locaux avec le sélecteur contenu dans op_1, adressant ainsi un nouveau segment LDT.

Le sélecteur donné doit pointer sur un élément de la table des descripteurs globaux (GDT) qui est un descripteur du LDT. S'il en est ainsi, le registre LDT est chargé. Dans le Chapitre 5, nous décrirons plus précisément ces tests.

op_1 peut être un sélecteur nul, qui marquera le LDT comme non valide. Charger un sélecteur associé à un segment LDT générera une exception du type chargement de segment si LDTR contient un sélecteur nul.

LLDT apparaît uniquement dans du code de système d'exploitation ; on ne devrait jamais la trouver dans un programme d'application.

Fonctionnement :

```
LDTR = op1;
```

Exemple :

```
LLDT AX   Avant :   AX :   001C   LDTR :   XXXX
          Après :   AX :   001C   LDTR :   001C
```

Exceptions :

SegmentLoad();
Memory();
Level0();
8086Mode();

Note :

Voir la page 702.

LMSW

Chargement du mot d'état machine

Formats :

LMSW	op_1
	reg16
	mem16
	reg32
	mem32

Description :

LMSW charge le registre MSW à partir de op_1. MSW (les 16 bits de poids faible de CR0) est décrit dans le Chapitre 2.

On peut employer LMSW pour entrer en mode protégé en positionnant le bit PE de MSW à 1. Dans ce cas, l'instruction LMSW doit immédiatement être suivie d'une instruction de saut intrasegment.

Notez aussi que pour des raisons de stricte compatibilité avec le 80286, l'instruction LMSW ne modifie pas le bit ET de MSW.

Notez que le bit PE de MSW est lié à l'instruction LMSW. Cela signifie qu'après avoir été mis à 1 (on est en mode protégé), le bit PE de MSW ne peut pas être mis à 0 par l'instruction LMSW. Se reporter au Chapitre 5 pour plus d'informations sur le mode protégé et au Chapitre 9 pour plus de précisions sur le mode réel.

LMSW apparaît uniquement dans du code de système d'exploitation ; on ne devrait jamais la trouver dans un programme d'application.

Fonctionnement :

```
MSW = (op1 & FFEFh) | (MSW & 0011h);
      /* CR0 = (CR0 & FFFF0011h) | (0000FFEFh & op1); */
```

Exemples :

```
LMSW AX    Avant :   AX :   0007   MSW :   0010
           Après :   AX :   0007   MSW :   0017

LMSW AX    Avant :   AX :   0010   MSW :   0007
           Après :   AX :   0010   MSW :   0001
```

Exceptions :

Memory();
Level0();

Note :

L'instruction LMSW permet une compatibilité stricte avec le 80286. Quand on programme le 80386, on devrait employer MOV CR0 au lieu de LMSW.

LSL

Chargement de la limite de segment

Formats :

LSL	op_1	,	op_2
	reg16	,	reg16
	reg16	,	mem16
	reg32	,	reg32
	reg32	,	mem32

Description :

LSL charge le premier opérande avec le champ limite (limite de segment) du descripteur du segment déterminé par le sélecteur du second opérande.

Cette limite placée dans op_1 est la limite d'octet. Ainsi, si le descripteur précisé a une limite à granulité de page (G = 1, le bit G du descripteur, décrit dans le Chapitre 5), les limites sont décalées à gauche de 12 bits et ces 12 bits sont remplis de 1.

Le descripteur déterminé par le sélecteur dans op_2 doit être compris dans les limites de la table des descripteurs, avoir un champ Type valide pour LSL (voir le tableau), et être accessible à la fois par le CPL et par le RPL du sélecteur de op_2 comparé au DPL. S'il en est ainsi, ZF est mis à 1 et op_1 est modifié. Sinon, ZF est mis à 0 et op_1 n'est pas modifié.

Tout descripteur de segment mémoire (noté par un 1 dans le champ DType) peut avoir n'importe quelle valeur dans son champ Type et pourtant être valide pour LSL. Nous allons donner les types valides pour des segments système et des guichets (DType = 0) dans le tableau suivant :

Type du descripteur	Valide ?
0 Indéfini	Non
1 TSS 286 disponible	Oui
2 LDT	Oui
3 TSS 286 occupé	Oui
4 Guichet appel 286	Non
5 Guichet de Tâche	Non
6 Guichet d'interruption 286	Non
7 Guichet de trappe 286	Non
8 Indéfini	Non
9 TSS 386 disponible	Oui
A Indéfini	Non
B TSS 386 occupé	Oui
C Guichet d'appel 386	Non
D Indéfini	Non
E Guichet d'interruption 386	Non
F Guichet de trappe 386	Non

Fonctionnement :

```
if ( (op2& 0FFFch) == 0    /* Sélecteur nul */ )
      goto ClearZF;

/* La routine FetchDescriptor est définie au Chapitre 5. */
/* Retourne 0 si le descripteur est au-delà des limites de la table
*/
if (FetchDescriptor(op2,&dword1,&dword2,&Attributes) == 0)
      goto ClearZF;

/* Sinon le descripteur est compris dans les limites de la table */
if (Attributes.DType == 1    /* segment mémoire */ )
      switch (Attributes.Type){
         case 0: case 1:       /* Lecture */
         case 2: case 3:       /* Lecture/Ecriture */
         case 4: case 5:       /* Lecture expand_down */
         case 6: case 7:       /* Lecture/Ecriture, expand_down */
```

```
            case 8: case 9:       /* Exécution */
            case 10: case 11:    /* Exécution/Lecture */
               /* Teste le privilège seulement pour des segments */
               /* non conformes */
               if ( (Attributes.DPL < CPL) ||
                    (Attributes.DPL < op².RPL))
                  goto ClearZF;
               break;
               /* On ignore DPL pour un segment conforme */
            case 12: case 13:    /* Exécution, conforme */
            case 14: case 15:    /* Exécution/Lecture, conforme */
               break;
         } /* fin du switch */

else   /* DType == 0, segment système ou guichet */
         switch(Attributes.Type){
            Case 1:   /* TSS 286 Disponible */
            Case 2:   /* LDT                */
            Case 3:   /* TSS 286 Occupé     */
            Case 9:   /* TSS 386 Disponible */
            Case 11:   /* TSS 386 Occupé     */
               break;

            Default:   /* Autres types non valides */
                  goto ClearZF;
         } /* fin du switch */

/* On arrive ici seulement si tous les tests sont OK */
/* Met ZF à 1, charge la limite du segment dans op¹        */
         ZF = 1;
         if (B(23,dword1) == 0){   /* un segment à granulité d'octet */
            op¹ = (dword2 & 000F0000h) |
                  (dword1 & 0000FFFFh) ;
         } else {          /* un segment à granulité de page */
            op¹ = (dword2 & 000F0000h) |
                  (dword1 & 0000FFFFh) ) << 12) | FFFh;
         }
         return;

/* Saut à ClearZF si quelque chose s'est mal passé  */
/* Met ZF à 0 et ne change pas op₁                  */
```

```
ClearZF:
      ZF = 0;
      return;
```

Exemples :

```
                                OSZAPC
LSL EAX,mem32      ; Descripteur visible
      Avant :   Descr(0,mem32) :   1F0FACFE      XXXXXX
                Descr(1,mem32) :   008F8100
                EAX :   XXXXXXXX
      Après :   Descr(0,mem32) :   1F0FACFE      UU1UUU
                Descr(1,mem32) :   008F8100
                EAX            :   FACFEFFF
```

Exceptions :

Memory();
8086Mode();

LTR

Chargement du registre de tâche

Formats :

LTR op_1
reg16
mem16
reg32
mem32

Description :

LTR charge le registre de tâche à partir du sélecteur dans op_1, rendant un segment TSS adressable. Le TSS (segment d'état de tâche, *Task State Segment*) chargé est noté occupé.

Le sélecteur donné doit pointer sur un élément de table de descripteurs globaux (GDT), c'est-à-dire un descripteur du type TSS. S'il en est ainsi, le registre de tâche est chargé. Dans le Chapitre 5, nous décrirons ces tests plus précisément.

LTR apparaît seulement dans du code de système d'exploitation ; on devrait pas la trouver dans des programmes d'application.

Fonctionnement :

```
TR = op1;
```

Exemple :

```
LTR mem32      Avant :   mem32 :   XXXX0014   TR :   XXXX
               Après :   mem32 :   UUUU0014   TR :   0014
```

Exceptions :

SegmentLoad();
Memory();
8086Mode();
Level0();

Note :

Voir les pages 704 et 969.

MOV

Transfert d'un registre spécial

Formats :

MOV	op_1	,	op_2
	reg32	,	CR0
	reg32	,	CR2
	reg32	,	CR3
	CR0	,	reg32
	CR2	,	reg32
	CR3	,	reg32
	reg32	,	DR0
	reg32	,	DR1
	reg32	,	DR2
	reg32	,	DR3
	reg32	,	DR6
	reg32	,	DR7
	DR0	,	reg32
	DR1	,	reg32
	DR2	,	reg32
	DR3	,	reg32
	DR6	,	reg32
	DR7	,	reg32

Description :

L'instruction copie le contenu de op_2 dans op_1.

Dans le Chapitre 5 (resp. Chapitre 8), nous décrirons les registres de contrôle (resp. de mise au point) et les effets exacts de leur chargement.

Notez qu'on ne peut utiliser que des opérandes 32 bits avec cette instruction MOV.

Cette instruction peut être employée uniquement au niveau de privilège 0.

Fonctionnement :

```
op1 = op2;    /*    */
/*
        Puisque le chargement de ces registres
      spéciaux peut avoir des effets de bord,
      cette description n'est pas complète pour
      tous les cas.
*/
```

Exemple :

```
MOV CR0,EAX   Avant :   EAX :   80000003   CR0 :   XXXXXXXX
              Après :   EAX :   80000003   CR0 :   80000003
```

Exception :

Level0();

Note :

Voir la page 838.

SGDT

Stockage de la table des descripteurs globaux

Format :

SGDT op_1
mem48

Description :

Le contenu du registre de la table des descripteurs globaux est rangé dans un pseudo-descripteur de 6 octets déterminé par op_1. La limite 16 bits est rangée dans le mot de poids faible et la base 32 bits dans le dmot de poids fort.

SGDT apparaît seulement dans du code de système d'exploitation ; on ne devrait jamais la trouver dans un programme d'application.

Fonctionnement :

```
mem48   = GDTR.Limit;
[&mem48+2]= GDTR.Base;
```

Exemple :

```
SGDT mem48   Avant :   GDTR.Limit :   0032   GDTR.Base   :   0BD542FF
                       mem48      :   XXXX   [&mem48+2] :   XXXXXXXX
             Après :   GDTR.Limit :   0032   GDTR.Base   :   0BD542FF
                       mem48      :   0032   [&mem48+2] :   0BD542FF
```

Exception :

Memory();

SIDT

Stockage de la table des descripteurs d'interruptions

Format :

SIDT op_1
mem48

Description :

Le contenu du registre de la table des descripteurs d'interruptions est rangé dans un pseudo-descripteur de 6 octets déterminé par op_1. La limite 16 bits est rangée dans le mot de poids faible et la base 32 bits dans le dmot de poids fort.

SIDT apparaît seulement dans du code de système d'exploitation ; on ne devrait jamais la trouver dans un programme d'application.

Fonctionnement :

```
mem48      = IDTR.Limit;
[&mem48+2]= IDTR.Base;
```

Exemple :

```
SIDT mem48   Avant :   IDTR.Limit :   0032   IDTR.Base   :   0BD542FF
                       mem48      :   XXXX   [&mem48+2] :   XXXXXXXX
             Après :   IDTR.Limit :   0032   IDTR.Base   :   0BD542FF
                       mem48      :   0032   [&mem48+2] :   0BD542FF
```

Exception :

Memory();

SLDT

Stockage de la table des descripteurs locaux

Formats :

SLDT op_1
reg16
mem16
reg32
mem32

Description :

Le contenu du registre de la table des descripteurs locaux est rangé dans op_1. Ce registre contient un sélecteur qui pointe sur le LDT courant.

SLDT apparaît seulement dans du code de système d'exploitation ; on ne devrait pas la trouver dans des programmes d'application.

Fonctionnement :

```
op1 = LDTR;
```

Exemple :

```
SLDT mem16   Avant :   LDTR :   5569   [&mem16] :   XXXX
             Après :   LDTR :   5569   [&mem16] :   5569
```

Exceptions :

Memory();
8086Mode();

SMSW

Stockage du mot d'état machine

Formats :

SMSW op_1
reg16
mem16
reg32
mem32

Description :

SMSW place le contenu du registre MSW dans op_1. MSW (qui correspond aux 16 bits de poids faible de CR0) est décrit dans le Chapitre 2.

Fonctionnement :

```
op1 : MSW;
```

Exemple :

```
SMSW AX   Avant :   AX :   XXXX   MSW :   FF10
          Après :   AX :   FF10   MSW :   FF10
```

Exception :

Memory();

Note :

L'instruction SMSW permet d'assurer une compatibilité stricte avec le 80286. Quand on programme le 80386, on devrait employer MOV CR0 au lieu de SMSW.

STR

Stockage du registre de tâche

Formats :

STR	op_1
	reg16
	mem16

Description :

Le contenu du registre de tâche est rangé dans op_1.

STR apparaît dans du code de système d'exploitation ; on ne devrait jamais la trouver dans des programmes d'application.

Fonctionnement :

```
op1 = TR;
```

Exemple :

```
STR mem16      Avant :   TR :   FFD5   mem :   XXXX
               Après :   TR :   FFD5   mem :   FFD5
```

Exceptions :

Memory();
8086Mode();

Note :

Voir l'exemple 2, page 848.

VERR

Vérification du segment en lecture

Formats :

VERR op_1
reg16
mem16
reg32
mem32

Description :

L'instruction VERR détermine si le segment identifié par le sélecteur contenu dans op_1 peut être lu.

Le sélecteur doit être un descripteur valide (dans le LDT ou le GDT), un descripteur de segment mémoire, et pouvoir être lu et être visible au niveau de privilège courant. Si le segment identifié par le sélecteur de op_1 remplit ces conditions, l'indicateur de zéro est mis à 1. Sinon, il est mis à 0.

Ces étapes de vérification (avec les résultats dans l'indicateur de zéro) sont similaires à celles du chargement du descripteur référencé et d'une lecture de ce segment.

Fonctionnement :

```
if ( (op2& 0FFFch) == 0    /* Sélecteur nul */ )
     goto ClearZF;

/* La routine FetchDescriptor est définie dans le Chapitre 5. */
```

```
/* Retourne 0 si le descripteur est au-delà des limites de la table */
if (FetchDescriptor(op2,&dword1,&dword2,&Attributes) == 0)
      goto ClearZF;

/* Sinon le descripteur est compris dans les limites de la table */
if (Attributes.DType == 1   /* segment mémoire */ )
      switch (Attributes.Type){
         case 0: case 1:       /* Lecture */
         case 2: case 3:       /* Lecture/Ecriture */
         case 4: case 5:       /* Lecture, expand_down */
         case 6: case 7:       /* Lecture/Ecriture, expand_down */
         case 10: case 11:    /* Exécution/Lecture */
            /* Teste le privilège seulement pour des segments */
            /* non conformes */
            if ( (Attributes.DPL < CPL) ||
                (Attributes.DPL < op2.RPL))
               goto ClearZF;
            break;
            /* On ignore DPL pour un segment conforme */
         case 14: case 15:    /* Exécution/Lecture, conforme */
            break;
         Default:   /* Les autres types sont non valides */
            goto ClearZF;
      } /* fin du switch */

else   /* DType == 0, segment système ou guichet */
      goto ClearZF; /* Tous les segments système sont non valides */

/* On arrive ici seulement si tous les tests sont OK */
/* Met ZF à 1, charge la limite du segment dans op1        */
      ZF = 1;
      return;

/* Saut à ClearZF si quelque chose s'est mal passé  */
/* Met ZF à 0 et ne change pas op1                  */
ClearZF:
      ZF = 0;
      return;
```

Exemple :

```
VERR mem16    ; Le segment est lisible    OSZAPC
                      Avant :             XXXXXX
                      Après :             XX1XXX
```

Exceptions :

Memory();
8086Mode();

VERW

Vérification du segment en écriture

Formats :

VERW	op_1
	reg16
	mem16
	reg32
	mem32

Description :

L'instruction VERW détermine si le segment identifié par le sélecteur contenu dans op_1 peut être écrit.

Le sélecteur doit être un descripteur valide (dans le LDT ou le GDT), un descripteur de segment mémoire, et pouvoir être écrit et être visible au niveau de privilège courant. Si le segment identifié par le sélecteur de op_1 remplit ces conditions, l'indicateur de zéro est mis à 1. Sinon, il est mis à 0.

Ces étapes de vérification (avec les résultats dans l'indicateur de zéro) sont similaires à celles du chargement du descripteur référencé et d'une écriture de ce segment.

Fonctionnement :

```
if ( (op2& 0FFFch) == 0    /* Sélecteur nul */ )
      goto ClearZF;

/* La routine FetchDescriptor est définie dans le Chapitre 5. */
```

```
/* Retourne 0 si le descripteur est au-delà des limites de la table */
if (FetchDescriptor(op2,&dword1,&dword2,&Attributes) == 0)
      goto ClearZF;

/* Sinon le descripteur est compris dans les limites de la table */
if (Attributes.DType == 1   /* segment mémoire */ )
      switch (Attributes.Type){
         case 2: case 3:       /* Lecture/Ecriture */
         case 6: case 7:       /* Lecture/Ecriture, expand_down */
            /* Teste le privilège seulement pour des segments */
            /* non conformes */
            if ( (Attributes.DPL < CPL) ||
                (Attributes.DPL < op2.RPL))
               goto ClearZF;
            break;
         Default:   /* Les autres types sont non valides */
            goto ClearZF;
      } /* fin du switch */

else   /* DType == 0, segment système ou guichet */
      goto ClearZF; /* Tous les segments système sont non valides */

/* On arrive ici seulement si tous les tests sont OK */
/* Met ZF à 1, charge la limite du segment dans op1        */
      ZF = 1;
      return;

/* Saut à ClearZF si quelque chose s'est mal passé  */
/* Met ZF à 0 et ne change pas op1                  */
ClearZF:
      ZF = 0;
      return;
```

Exemple :

```
VERW mem16    ; Le segment est lisible    OSZAPC
                       Avant :            XXXXXX
                       Après :            XX1XXX
```

Exceptions :

Memory();
8086Mode();

VIRGULE FLOTTANTE

Dans cette partie, nous allons présenter le jeu complet des instructions à virgule flottante disponibles sur les systèmes 80386/80387. Toutes les instructions arithmétiques, de transfert de données, de comparaison, transcendantales, et de contrôle du 80387 sont décrites ci-après.

Chaque instruction contient un ou deux exemples d'utilisation. Vous trouverez davantage d'exemples dans la seconde moitié du Chapitre 4. Ces exemples sont des morceaux de programmes composés de plusieurs instructions qui exécutent quelques fonctions intéressantes. Entre la description, les exemples donnés ici, et le Chapitre 4 présentant des exemples plus complexes, vous devriez pouvoir mieux comprendre le but et le fonctionnement de ces instructions.

Etant donné que la syntaxe des instructions à virgule flottante est quelquefois sensiblement différente de celle qui est employée pour décrire les instructions du 80386, nous consacrons quelques paragraphes à la révision de cette syntaxe.

Mnémoniques des instructions

Les mnémoniques de l'assembleur pour les instructions à virgule flottante suivent une notation standardisée.

F — Toutes les instructions à virgule flottante commencent par F (FADD, FLD, etc.). Aucune autre instruction de la famille 86 ne commence par F.

FI — Toutes les instructions qui travaillent sur des types de données entiers commencent par FI (FIADD, FILD, etc.).

FB — Toutes les instructions qui travaillent sur des types de données BCD commencent par FB (FBLD, FBST, etc.).

FxxP — Toutes les instructions qui dépilent une fois se terminent par P (FSTP, FADDP, etc.).

FxxPP — Toutes les instructions qui dépilent deux fois se terminent par PP (FCOMPP, FUCOMPP, etc.).

FNxx — Toutes les instructions, sauf celles qui commencent par FN, testent les exceptions numériques non masquées avant de s'exécuter. Les instructions FNxx n'effectuent pas ce test (FNINIT, FNSAVE, etc.).

Pile

Comme nous l'avons mentionné dans le Chapitre 2, le 80387 emploie une pile accumulateur et presque toutes les opérations doivent utiliser le sommet de la pile comme opérande. Le sommet de la pile est noté *ST*. On fera référence aux autres accumulateurs en employant la notation ST(n) (relativement à ST donc), qui représente le n^{ième} élément sous le sommet de la pile. Ainsi, ST(3) représente le troisième accumulateur sous le sommet de la pile.

Puisque plusieurs opérations à virgule flottante empilent et dépilent des éléments, il est souvent utile d'indiquer ce qui se passe au sommet de la pile. Pour cela, nous emploierons la notation TOP dans les exemples. Ainsi, TOP = 3 indique que le registre courant de sommet de pile est le troisième registre physique à virgule flottante.

Exemple

Voici un exemple d'instruction à virgule flottante. C'est l'instruction FCOMP. Dans ce cas, elle compare numériquement le sommet de la pile ST avec ST(3), le troisième élément qui se trouve au-dessous du pointeur

de pile. Après cette comparaison, le sommet de la pile est dépilé : TOP passe de 4 à 5.

```
FCOMP ST(3)   Avant :   ST    :   2.4560 * 10⁶
                        ST(3) :   9.4102 * 10⁴
                        TOP   :   4   C3 :   X   C2 :   X   C0 :   X
              Après :   ST(2) :   9.4102 * 10⁴
                        TOP   :   5   C3 :   0   C2 :   0   C0 :   0
```

Notez que les empilages entraînent la décrémentation du pointeur de sommet de pile (on met un nouvel élément dans la pile), et les dépilages l'incrémentation du même pointeur (on enlève un élément de la pile). Nous avons expliqué ce mécanisme dans le Chapitre 2, et vous en avez une représentation visuelle à la Figure 2.24.

Définitions de fonctions

Au début de ce chapitre, nous avons défini plusieurs fonctions qui nous ont permis d'abréger les descriptions d'instructions et les exemples. Nous allons donner des routines dans un but identique. Elles sont définies maintenant car leur usage est limité à cette partie de la description du jeu d'instructions.

```
/* Dépile de la pile accumulateur à virgule flottante        */
/* Le pointeur de sommet de pile TOP est alors incrémenté    */
FPop(){
      TOP = TOP + 1;
      if (TOP > 7) {
         TOP = 0;
      }
}
/* Empile dans la pile à virgule flottante    */
```

```
/* Le pointeur est décrémenté        */
FPush(){
      TOP = TOP - 1;
      if (TOP < 0) {
         TOP = 7;
      }
}
/* Notez que FIP (Floating-point Instruction Pointer,        */
/* FCS (Floating-point Code Segment), FOO (Floating          */
/*-point Operand Offset), et FOS (Floating-point Operand     */
/* Segment) sont chargés en commençant à [memp+12]           */
/* Notez que FIP et FCS sont juste des copies de EIP et CS   */
/* à partir de la plus récente instruction à virgule         */
/* flottante. FOO et FOS sont simplement le déplacement      */
/* et le segment de "mem" (de la plus récente instruction    */
/* à virgule flottante si elle existe).                      */
LdErrorPointer(memp){
      FIP = [memp + 12];
      FCS = [memp + 16];
      FOO = [memp + 20];
      FOS = [memp + 24];
}

StErrorPointer(memp){
      [memp + 12] = FIP;
      [memp + 16] = FCS;
      [memp + 20] = FOO;
      [memp + 26] = FOS;
}

/* Les routines suivantes correspondent à leurs            */
/* noms mathématiques                                      */
abs(op);       /* retourne la valeur absolue de op         */
arctan(op);    /* retourne l'arctan de op (op en radians)  */
cos(op);       /* retourne le cos de op (op en radians)    */
log2(op);       /* retourne le log base 2 de op            */
sin(op);       /* retourne le sin de op (op en radians)    */
sqrt(op);       /* retourne la racine carrée de op         */
tan(op);       /* retourne la tan de op (op en radians)    */
```

Routines d'exception

Au début de ce chapitre, nous avons défini plusieurs routines d'exception qui nous ont permis d'abréger les conditions d'exception. Nous allons donner une autre routine d'exception que nous n'avons pas définie au début du chapitre car elle est spécifique à cette partie.

```
Float(Src){
      if(exception Src not masked){
         Interrupt(16);
      }
      /* case Src:
         IS:   Opération non valide causée par débordement de la pile
         I :   Opération non valide
         D :   Opérande dénormalisé
         Z :   Division par zéro
         O :   Débordement (Overflow)
         U :   Débordement (Underflow)
         P :   Précision
```

Résumé des instructions

Le tableau suivant donne un résumé de toutes les instructions de cette partie en les divisant en sous-groupes. Ensuite, viendra la description instruction par instruction.

Transfert de données

FBLD	Chargement de BCD
FBSTP	Stockage de BCD et dépilage
FILD	Chargement d'entier
FIST	Stockage d'entier
FISTP	Stockage d'entier et dépilage
FLD	Chargement de réel
FST	Stockage de réel

FSTP	Stockage d'entier et dépilage
FXCH	Echange de registres

Arithmétique

FABS	Valeur absolue
FADD/FIADD/FADDP	Addition
FCHS	Changement de signe
FDIV/FIDIV/FDIVP	Division
FDIVR/FIDIVR/FDIVRP	Division arrière
FMUL/FIMUL/FMULP	Multiplication
FPREM	Reste partiel
FPREM1	Reste partiel - IEEE
FRNDINT	Arrondi à l'entier
FSCALE	Echelle de puissance de deux
FSUB/FISUB/FSUBP	Soustraction
FSUBR/FISUBR/FSUBRP	Soustraction arrière
FSQRT	Racine carrée
FXTRACT	Extraction de l'exposant et de la mantisse

Comparaison

FCOM/FCOMP/FCOMPP	Comparaison
FICOM/FICOMP	Comparaison d'entiers
FTST	Test
FUCOM/FUCOMP/FUCOMPP	Comparaison non ordonnée
FXAM	Examen

Transcendantale

F2XM1	$2^x - 1$
FCOS	Cosinus
FPATAN	Arctangente partiel
FPTAN	Tangente partielle
FSIN	Sinus
FSINCOS	Sinus et cosinus
FYL2X	$y * \log_2 x$
FYL2XP1	$y * \log_2(x+1)$

Note : Les opérandes de FCOS, FPTAN, FSIN, et FSINCOS doivent être en radians.

Constantes

FLD1	Chargement de 1
FLDL2E	Chargement de $\log_2 e$
FLDL2T	Chargement de $\log_2 10$
FLDLG2	Chargement de $\log_{10} 2$
FLDLN2	Chargement de $\log_e 2$
FLDPI	Chargement de π
FLDZ	Chargement de zéro

Contrôle

FCLEX/FNCLEX	Effacement des exceptions
FDECSTP	Décrémentation du pointeur de pile
FFREE	Libération du registre
FINCSTP	Incrémentation du pointeur de pile
FINIT/FNINIT	Initialisation du processeur
FLDCW	Chargement du mot de contrôle
FLDENV	Chargement de l'environnement
FNOP	Pas d'opération
FRSTOR	Récupération de l'état
FSAVE/FNSAVE	Sauvegarde de l'état
FSTCW/FNSTCW	Stockage du mot de contrôle
FSTENV/FNSTENV	Stockage de l'environnement
FSTSW/FNSTSW	Stockage du mot d'état
FSTSW AX/FNSTSW AX	Stockage du mot d'état dans AX
WAIT	Attente pour le coprocesseur

F2XM1

2^X-1

Format :

F2XM1

Description :

L'élément situé au sommet de la pile est ST. Le calcul est donc 2ST - 1. Le résultat remplace le ST initial.

Notez que les valeurs de l'opérande d'entrée peuvent varier de -0.5 à 0.5. Dans le cas contraire, le résultat est indéfini.

Fonctionnement :

```
if (-0.5 <= ST <= 0.5){
      ST = 2^ST - 1;
} else {
      UNDEFINED;
}
```

Exemple :

```
F2XM1   Avant :   ST :   3.49921 * 10^-2
        Après :   ST :   2.45512 * 10^-2
```

Exception :

Float(IS,I,D,U,P);

Note :

Le 80387 permet des opérandes variant de -0.5 à 0.5 tandis que le 80287 se limite à des variations de 0 à 0.5.

FABS

Valeur absolue

Format :

FABS

Description :

L'élément situé au sommet de la pile est remplacé par sa valeur absolue. Cet élément est donc toujours positif après l'instruction FABS.

Fonctionnement :

```
if (ST < 0 ){
      ST = - ST;
}
```

Exemple :

```
FABS    Avant :    ST :    -7.324 * 10^-402
        Après :    ST :     7.324 * 10^-402
```

Exception :

Float(IS);

FADD/FIADD/FADDP

Addition

Formats :

```
FADD                          ; Addition réelle ST(1), ST
FADD    op1                   ; Addition réelle
        ST(n)
        mem32                 ; réel court
        mem64                 ; réel long

FIADD   op1                   ; Addition entière
        mem16                 ; mot entier
        mem32                 ; entier court

FADDP   op1     , op2         ; Addition réelle et dépilage
        ST      , ST(n)
        ST(n)   , ST

FADD    op1     , op2         ; Addition réelle
        ST      , ST(n)
        ST(n)   , ST
```

Description :

Ces instructions ajoutent les opérandes à virgule flottante définis explicitement ou implicitement (ST(1)+ST, ST+op_1, ou op_1+op_2, si 0, 1, ou 2 opérandes sont précisés). Le résultat est rangé au sommet de la pile dans tous les cas sauf pour la forme à deux opérandes, où la destination peut être ST(n).

Si op_1 est un opérande mémoire (mot entier ou entier court, réel court ou réel long), il est automatiquement converti en réel temporaire (le format interne) avant tout traitement.

Un FADDP ou un FADD sans opérandes entraîne un dépilage. Ainsi, FADD sans opérandes est synonyme de FADDP ST(1),ST.

Fonctionnement :

```
if (NUMOPS == 0){
      ST(1) = ST(1) + ST;
      FPop();   /* dépile et résultat dans ST */
      } else if (NUMOPS == 1){
         ST = ST + (tempReal) op1;
} else {       /* NUMOPS == 2 */
      op1 = op1 + op2;
      if (OPCODE == FADDP){
         FPop();   /* Dépile */
      }
}
```

Exemple :

FADD mem32	Avant :	ST :	$4.51200 * 10^5$	mem :	$3.664 * 10^3$
	Après :	ST :	$4.54864 * 10^5$	mem :	$3.664 * 10^3$

Exceptions :

Memory();
Float(IS,I,D,U,O,P);

Note :

Le tableau suivant résume toutes les additions possibles avec les infinis.

Opérandes		Résultats (op_1)	
op_1	op_2	Signe	Valeur
$+\infty$	$+\infty$	+	∞
$-\infty$	$-\infty$	-	∞
$-\infty$	$+\infty$	Opération non valide	
$+\infty$	$-\infty$	Opération non valide	
$\pm z$	$\pm\infty$	Signe de ∞	∞
$\pm\infty$	$\pm\infty$	Signe de ∞	∞

Où $0 \leq z < \infty$.

FBLD

Chargement de BCD

Format :

FBLD mem80

Description :

mem80 pointe sur un opérande mémoire du type BCD compacté de 10 octets. Il est converti en réel temporaire et empilé. La conversion est toujours exacte. Les chiffres BCD de mem80 sont censés varier de 0h à 9h.

Fonctionnement :

```
Fpush();
ST = (tempReal) mem80;
```

Exemple :

```
FBLD mem80   Avant :   mem80 :   -000000000000005698   TOP :   4
             Après :   mem80 :   -000000000000005698   TOP :   3
                       ST    :   -5.698 * 10^3
```

Exceptions :

Float(IS);
Memory();

Note :

Toute tentative pour charger des chiffres BCD non valides place un réel temporaire indéfini dans ST.

FBSTP

Stockage de BCD et dépilage

Format :

FBSTP mem80

Description :

Le sommet de la pile est converti en un nombre du type BCD de 18 chiffres, qui est rangé dans les 10 octets pointés par mem80. Ensuite, on dépile. ST est arrondi à un entier en employant le mode d'arrondi déterminé par RC. Ainsi, FBSTP n'a pas besoin d'un FRNDINT.

Fonctionnement :

```
tmp = (BCD) ST;
FPop();
mem80 = tmp;
```

Exemple :

```
FBLD mem80   Avant :   ST    :   8.99033 * 10³      TOP :   7   RC :   RD
             Après :   mem80 :   XXXXXXXXXXXXXXXXXX
             Après :   ST    :   8.99033 * 10³      TOP :   0   RC :   RD
             Après :   mem80 :   000000000000008990
```

Exceptions :

```
Float(IS,I);
Memory();
if (ST is QUIETNaN){
        Float(I);
}
```

Notes :

Pour FBSTP, ST peut être un dénormalisé, mais pas pour un 80287.

Cette instruction est très lente et peut poser des problèmes au niveau du traitement des interruptions.

Si ST est un NaN silencieux, l'instruction générera une exception d'opération non valide. En général, les opérations sur des NaN silencieux ne génèrent pas d'exceptions.

FCHS

Changement de signe

Format :

FCHS

Description :

L'instruction complémente le signe de l'élément situé au sommet de la pile. Si ce dernier est égal à +0.0, le résultat de FCHS est -0.0.

Fonctionnement :

```
ST = - ST;
```

Exemple :

```
FCHS    Avant :    ST :    6.221 * 10^37
        Après :    ST :   -6.221 * 10^37
```

Exception :

Float(IS);

FCLEX/FNCLEX

Effacement des exceptions

Formats :

FCLEX
FNCLEX

Description :

FCLEX/FNCLEX permettent d'effacer toutes les exceptions (80387) ainsi que le bit Occupé (Busy) du registre d'état.

FCLEX vérifie les exceptions numériques non masquées, alors que FNCLEX ne le fait pas.

Fonctionnement :

```
SW = SW & 7F00h;
```

Exemple :

```
FCLEX    Avant :    SW :    F450
         Après :    SW :    7400
```

Exceptions :

Aucune.

FCOM/FCOMP/FCOMPP

Comparaison

Formats :

FCOM	op_1	;comparaison
	mem32	
	mem64	
	ST(i)	
FCOMP	op_1	;comparaison et dépilage
	mem32	
	mem64	
	ST(i)	
FCOMPP		;comparaison et dépilage deux fois

Description :

Ces instructions comparent numériquement l'opérande donné avec le sommet de la pile, puis positionnent les codes de condition en accord avec le tableau suivant. Si le CodeOp est FCOMPP, l'opérande est alors ST(1).

	C3	C2	C0
ST > op$_1$	0	0	0
ST < op$_1$	0	0	1
ST == op$_1$	1	0	0
Sinon ( )	1	1	1

Si le CodeOp est FCOMP, le 80387 dépile après la comparaison. Si le CodeOp est FCOMPP, le 80387 dépile deux fois après la comparaison.

( ) Si les deux opérandes sont des NaN, *il n'y a pas comparaison.*

Le 80387 ignore le signe du zéro dans les comparaisons.

Fonctionnement :

```
if (CODEOP == FCOMPP){
      op1 = ST(1);
}
if ((ST is NaN) || (op1 is NaN)){
      C3 = 1; C2 = 1; C0 = 1;
} else if (ST > op1){
      C3 = 0; C2 = 0; C0 = 0;
} else if (ST < op1){
      C3 = 0; C2 = 0; C0 = 1;
} else if (ST == op1){
      C3 = 1; C2 = 0; C0 = 0;
}
if (CODEOP == FCOMP){
      FPop();
} else if (CODEOP == FCOMPP){
      FPop();
      FPop();
}
```

Exemple :

FCOMP ST(3)	Avant :	ST	:	$2.4560 * 10^6$										
		ST(3)	:	$9.4102 * 10^4$										
		TOP	:	4	C3	:	X	C2	:	X	C0	:	X
	Après	TOP	:	5	C3	:	0	C2	:	0	C0	:	0

Exceptions :

```
Float(IS,I,D);
if ((ST is QUIETNaN) || (op1 is QUIETNaN)){
      Float(I);
}
```

Notes :

FCOM génère une exception du type opération non valide si l'un des opérandes est un NaN silencieux. Normalement, les opérations sur des NaN silencieux n'entraînent pas d'exceptions de ce type.

Un exemple de branchement conditionnel sur les codes de condition du 80387 est donné à la page 566 (exemple 7).

FCOS

Cosinus

Format :

FCOS

Description :

Le 80387 calcule le cosinus du ST. Ensuite, le sommet de la pile est égal au cosinus de ST et 1.0 est empilé. Ainsi :

ST(1)/ST = cos(ST')

où ST' est le sommet de pile précédant l'instruction FCOS. ST est toujours égal à 1.0, et ST(1) est le cosinus de ST'.

L'opérande d'entrée de FCOS doit varier de 0 à $\pi * 2^{62}$. Si l'opérande source respecte ces bornes, C2 est mis à 0 . Sinon C2 est mis à 1 et ST et le pointeur de pile (TOP) ne sont pas modifiés.

Fonctionnement :

```
if (ST < π * 2^62) {
      ST = cos(ST);
      FPush();
      ST = 1.0;
      C2 = 0;
} else {
      /* Note : ST non modifié */
      C2 = 1;
}
```

Exemple :

```
FCOS   Avant :   ST    :   4.51200 * 10^5      TOP :   4   C2 :   X
       Après :   ST    :   1.00000 * 10^0      TOP :   3   C2 :   0
                 ST(1) :   4.53137 * 10^-1
```

Exception :

Float(IS,I,D,U,P);

Note :

Cette instruction est une nouveauté du 80387.

FDECSTP

Décrémentation du pointeur de pile

Format :

FDECSTP

Description :

L'instruction soustrait 1 du pointeur de sommet de pile TOP (dans le mot d'état). Ni le mot d'étiquette ni même le sommet de la pile ne sont modifiés. Si on exécute un FDECSTP lorsque TOP est égal à 0, ST devient 7. Quand on empile un nouvel élément, le pointeur de pile est décrémenté comme le fait FDECSTP.

Fonctionnement :

```
TOP = TOP - 1;
if (TOP < 0){
      TOP = 7;
}
```

Exemple :

```
FDECSTP    Avant :    TOP :    4
           Après :    TOP :    3
```

Exceptions :

Aucune.

Note :

Cette instruction permet un contrôle direct du pointeur de pile. Cela peut s'avérer utile si on a besoin d'une pile accumulateur "virtuelle" plus grande que les huit registres réels dont dispose le 80387. Les registres de la pile "virtuelle" pourraient alors se trouver en mémoire, et le logiciel pourrait gérer la pile virtuelle lorsque des exceptions d'opérations non valides sur la pile sont détectées. FDECSTP, FINCSTP, et FFREE permettent de gérer une pile accumulateur virtuelle.

FDIV/FIDIV/FDIVP

Division

Formats :

```
FDIV                                ; Division réelle et dépilage ST(1), ST
FDIV    op1                         ; Division réelle
        ST(n)
        mem32                       ; réel court
        mem64                       ; réel long

FIDIV   op1                         ; Division entière
        mem16                       ; mot entier
        mem32                       ; entier court

FDIVP   op1     ,   op2             ; Division réelle et dépilage
        ST      ,   ST(n)
        ST(n)   ,   ST

FDIV    op1     ,   op2             ; Division réelle
        ST      ,   ST(n)
        ST(n)   ,   ST
```

Description :

Ces instructions effectuent une division sur les opérandes à virgule flottante définis explicitement ou implicitement (ST(1)/ST, ST/op_1, ou op_1/op_2, si 0, 1, ou 2 opérandes sont précisés). Le résultat est rangé au sommet de la pile dans tous les cas sauf pour la forme à deux opérandes où la destination peut être ST(n).

Si op_1 est un opérande mémoire (mot entier ou entier court, réel court ou réel long), il est automatiquement converti en réel temporaire (le format interne) avant tout traitement.

Un FDIV sans opérandes entraîne un dépilage. Ainsi, FDIV sans opérandes est synonyme de FDIVP ST(1),ST.

Fonctionnement :

```
if (NUMOPS == 0){
      ST(1) = ST(1) / ST;
      FPop();   /* dépile et résultat dans ST */
      } else if (NUMOPS == 1){
         ST = ST / (tempReal) op1;
} else {      /* NUMOPS == 2 */
      op1 = op1 / op2;
      if (OPCODE == FADDP){
         FPop();   /* Dépile */
      }
}
```

Exemple :

FDIV ST,ST(3)	Avant :	ST	:	$2.240 * 10^{-2}$
		ST(3)	:	$3.664 * 10^{3}$
	Après :	ST	:	$6.113 * 10^{-6}$
		ST(3)	:	$3.664 * 10^{3}$

Exceptions :

Memory();
Float(IS,I,D,U,O,P);

Notes :

Le tableau suivant résume toutes les possibilités de division avec les infinis.

Opérandes		Résultats (op_1)	
op_1	op_2	Signe	Valeur
$\pm\infty$	$\pm\infty$	Opération non valide	
$\pm\infty$	$\pm z$	Ou exclusif des signes	∞
$\pm z$	$\pm\infty$	Ou exclusif des signes	0
$\pm z$	± 0	Ou exclusif des signes	∞
$\pm\infty$	± 0	Ou exclusif des signes	∞

où $0 \leq z < \infty$.

Voir l'exemple 11, page 574.

FDIVR/FIDIVR/FDIVRP

Division arrière

Formats :

```
FDIVR                          ; Division réelle arrière ST(1), ST
FDIVR   op1                    ; Division réelle arrière
        ST(n)
        mem32                  ; réel court
        mem64                  ; réel long

FIDIVR  op1                    ; Division entière arrière
        mem16                  ; mot entier
        mem32                  ; entier court

FDIVRP  op1     ,  op2         ; Division réelle arrière et dépilage
        ST      ,  ST(n)
        ST(n)   ,  ST

FDIVR   op1     ,  op2         ; Division réelle arrière
        ST      ,  ST(n)
        ST(n)   ,  ST
```

Description :

Ces instructions effectuent une division sur les opérandes à virgule flottante définis explicitement ou implicitement (ST/ST(1), op_1/ST, ou op_2/op_1, si 0, 1, 2 opérandes sont précisés). Le résultat est rangé au sommet de la pile dans tous les cas sauf pour la forme à deux opérandes où la destination peut être ST(n).

Ces opérations sont équivalentes aux FDIV/FIDIV/FDIVP, si ce n'est que le numérateur et le diviseur sont inversés (op_2/op_1 pour FDIVR, plutôt que op_1/op_2 pour FDIV, par exemple).

Si op_1 est un opérande mémoire (mot entier ou entier court, réel court ou réel long), il est automatiquement converti en réel temporaire (le format interne) avant tout traitement.

Un FDIVR sans opérandes entraîne un dépilage. Ainsi, FDIVR sans opérandes est synonyme de FDIVRP ST(1),ST.

Fonctionnement :

```
if (NUMOPS == 0){
      ST(1) = ST / ST(1);
      FPop();   /* dépile et résultat dans ST */
      } else if (NUMOPS == 1){
         ST = (tempReal) op1 / ST;
} else {       /* NUMOPS == 2 */
      op1 = op2 / op1;
      if (OPCODE == FADDP){
         FPop();   /* Dépile */
      }
}
```

Exemple :

FDIVR ST,ST(3)	Avant :	ST	:	$2.240 * 10^{-2}$
		ST(3)	:	$3.664 * 10^{3}$
	Après :	ST	:	$1.636 * 10^{5}$
		ST(3)	:	$3.664 * 10^{3}$

Exceptions :

Memory();
Float(IS,I,Z,U,P,D,O);

Note :

Le tableau suivant résume toutes les possibilités de division arrière avec les infinis.

Opérandes		Résultats (op_1)	
op_1	op_2	Signe	Valeur
$\pm\infty$	$\pm\infty$	Opération non valide	
$\pm\infty$	$\pm z$	Ou exclusif des signes	0
$\pm z$	$\pm\infty$	Ou exclusif des signes	∞
± 0	$\pm z$	Ou exclusif des signes	∞
± 0	$\pm\infty$	Ou exclusif des signes	∞

où $0 \leq z < \infty$.

FFREE

Libération de registre

Format :

FFREE ST(n)

Description :

Les bits du mot d'étiquette associé au registre précisé sont mis à 11b. Cela signifie que cet élément de la pile devient vide. L'instruction ne modifie ni la pile à virgule flottante ni son pointeur.

Fonctionnement :

```
tmp = ((TOP + n) % 8);    /* numéro du registre physique */
TW = TW | (11b << (tmp * 2));
```

Exemple :

```
FFREE ST(3)    Avant :   TOP :   3   TW :   0FC0
               Après :   TOP :   3   TW :   3FC0
```

Exceptions :

Aucune.

Note :

Le mot d'étiquette décrit les registres physiques de la pile, et FFREE fournit une référence à la pile accumulateur qui est relative au sommet de la pile.

FICOM/FICOMP

Comparaison d'entiers

Formats :

FICOM	op_1	;	Comparaison entière
	mem16	;	mot entier
	mem32	;	entier court
FICOMP	op_1	;	Comparaison entière et dépilage
	mem16	;	mot entier
	mem32	;	entier court

Description :

Ces instructions convertissent l'opérande donné en réel temporaire puis le comparent numériquement avec le sommet de la pile. Elles positionnent les codes de condition en accord avec le tableau suivant :

	C3	C2	C0
ST > op$_1$	0	0	0
ST < op$_1$	0	0	1
ST == op$_1$	1	0	0
Sinon ( )	1	1	1

Si le CodeOp est FICOMP, le 80387 dépile après la comparaison.

( ) Si les deux opérandes sont des NaN, *il n'y a pas comparaison. Le 80387 ignore le signe du zéro dans les comparaisons.*

Fonctionnement :

```
tmp = (tempReal) op1;
if ((ST is NaN) {
      C3 = 1; C2 = 1; C0 = 1;
} else if (ST > tmp){
      C3 = 0; C2 = 0; C0 = 0;
} else if (ST < tmp){
      C3 = 0; C2 = 0; C0 = 1;
} else if (ST == tmp){
      C3 = 1; C2 = 0; C0 = 0;
}
if (CODEOP == FICOMP){
      FPop();
}
```

Exemple :

```
FICOMP mem16   Avant :   mem16 :   94320   ST :   6.0059 * 10^3
                         TOP   :   4   C3 :   X   C2 :   X   C0 :   X
               Après :   mem16 :   94320
                         TOP   :   5   C3 :   0   C2 :   0   C0 :   1
```

Exceptions :

```
Float(IS,I,D);
if ((ST is QUIETNaN) {
        Float(I);
}
```

Note :

Un exemple de branchement conditionnel sur les codes de condition du 80387 est donné à la page 566 (exemple 7).

FILD

Chargement d'entier

Formats :

FILD	op_1	
	mem16	;mot entier
	mem32	;entier court
	mem64	;entier long

Description :

L'instruction lit un entier (mot, entier court ou long) dans la mémoire, le convertit en réel temporaire, puis l'empile. Il devient ainsi le nouveau sommet de la pile.

Fonctionnement :

```
FPush();
ST == (tempReal) op1;
```

Exemple :

```
FILD mem32   Avant :   mem32 :   F234D9A1   TOP :   6
             Après :   mem32 :   F234D9A1   TOP :   5
                       ST    :   -2.31417439 * 10^8
```

Exceptions :

Float(IS);
Memory();

FINCSTP

Incrémentation du pointeur de pile

Format :

FINCSTP

Description :

L'instruction ajoute 1 au pointeur de sommet de pile TOP. Ni le mot d'étiquette (TW) ni le contenu de la pile ne sont modifiés. Si on exécute un FINCSTP lorsque le pointeur du sommet de pile (TOP) est égal à 7, ST devient 0. Quand on dépile un nouvel élément, le pointeur de pile est incrémenté comme le fait FINCSTP.

Fonctionnement :

```
TOP = (TOP + 1) % 8;
```

Exemple :

```
FINCSTP    Avant :   TOP :   7
           Après :   TOP :   0
```

Exceptions :

Aucune.

Note :

Cette instruction permet un contrôle direct du pointeur de pile. Cela peut s'avérer utile si on a besoin d'une pile accumulateur "virtuelle" plus grande que les huit registres réels dont dispose le 80387. Les registres de la pile "virtuelle" pourraient alors se trouver en mémoire, et le logiciel pourrait gérer la pile virtuelle lorsque des exceptions du type opération non valide sur la pile sont détectées. FDECSTP, FINCSTP, et FFREE permettent de gérer une pile accumulateur virtuelle.

FINIT/FNINIT

Initialisation du processeur

Formats :

FINIT
FNINIT

Description :

FINIT/FNINIT positionnent tous les registres de contrôle, d'état, et d'étiquette à leurs valeurs par défaut. Après cela, le mode d'arrondi devient l'arrondi au plus près, toutes les exceptions sont masquées, la précision est sur 64 bits, le mot d'état (SW) est mis à 0 sauf pour les quatre bits de code de condition qui sont indéfinis, et tous les registres de la pile à virgule flottante (TW) seront considérés comme vides.

FINIT teste les exceptions numériques non masquées, alors que FNINIT ne le fait pas.

Fonctionnement :

```
CW = 037Fh;    /* Arrondi au plus près, masque toutes les exceptions */
               /* Précision 64 bits */
SW = SW & 4700h;
TW = FFFFh;
```

Exemple :

```
FINIT   Avant :   CW :   1D7E   SW :   D401   TW :   3FFC
        Après :   CW :   037F   SW :   4400   TW :   FFFF
```

Exceptions :

Aucune.

FIST

Stockage d'entier

Formats :

FIST	op_1	
	mem16	;mot entier
	mem32	;entier court

Description :

L'instruction arrondit le sommet de la pile à un entier dont la longueur correspond à celle de op_1 : mot entier ou entier court. Les bits de contrôle d'arrondi (RC) du mot de contrôle (CW) déterminent le type d'arrondi. Puis, l'entier est rangé dans op_1. Un zéro négatif est converti en un zéro positif en notation complément à deux avant d'être stocké.

Fonctionnement :

```
if (op1 is mem16){
      op1 = (wordInt) ST;
} else {   /* mem32 */
      op1 = (shortInt) ST;
}
```

Exemple :

```
                                                        ; Arrondi au plus près
FIST mem16   Avant :   ST    :   8.90133 * 10¹      TOP :   5   RC :   RU
             Après :   ST    :   8.90133 * 10¹      TOP :   5   RC :   RU
                       mem16 :   005A
```

Exceptions :

```
Float(IS,I,P);
Memory();
if (ST is QUIETNaN){
        Float(I);
}
```

Notes :

FIST ne peut pas traiter un entier long.

Si ST est un NaN silencieux, une exception du type opération non valide est générée. Habituellement, un NaN silencieux n'entraîne pas une exception de ce type.

FISTP

Stockage d'entier et dépilage

Formats :

FISTP	op_1	
	mem16	;mot entier
	mem32	;entier court
	mem64	;entier long

Description :

L'instruction arrondit le sommet de la pile à un entier dont la longueur correspond à celle de op_1 : mot entier, entier court, ou entier long. RC détermine le type d'arrondi. Puis, l'entier est rangé dans op_1. Un zéro négatif est converti en un zéro positif en notation complément à deux avant d'être stocké. Il y a dépilage.

Fonctionnement :

```
if (op1 is mem16){
      op1 = (wordInt) ST;
} else if (op1 is mem32){
      op1 = (shortInt) ST;
} else {    /* mem64 */
      op1 = (LongInt) ST;
}
FPop();
```

Exemple :

```
                                                          ; Arrondi au plus près
FISTP mem64   Avant :   ST  :   -4.32990 * 10³     TOP :   5   RC :   RN
              Après :   mem :   FFFFFFFFFFFFEF16   TOP :   6   RC :   RN
```

Exceptions :

Float(IS,I,P);
Memory();

Notes :

FISTP peut traiter un entier long, alors que FIST ne le peut pas.

FLD

Chargement de réel

Formats :

FLD	op_1	
	mem32	;réel court
	mem64	;réel long
	mem80	;réel temporaire
	ST(i)	

Description :

Il y a empilement. Le réel court, long ou temporaire, ou l'opérande pile déterminé par op_1 est rangé au nouveau sommet de la pile. Si op_1 est un réel court ou long, l'instruction convertit l'opérande en réel temporaire avant de l'empiler.

Fonctionnement :

```
Fpush();
ST = (tempReal) op1;
```

Exemple :

```
FLD mem32          ; mem32 est un réel court
        Avant :   mem32 :   3.2611 * 10²   TOP :   2
        Après :   mem32 :   3.2611 * 10²   TOP:   1
                  ST    :   3.2611 * 10²
```

Exceptions :

```
Memory();
if (op1 in (mem32,mem64)){
        Float(I,D);
}
FLoat(IS);
```

Note :

Le 80287 transmettra toutes les exceptions Float(D) dans tous les formats (y compris le format étendu), tandis que le 80387 ne transmettra que les Float(D) pour les réels courts et longs.

FLDcon

Chargement de constante

Formats :

FLD1	;	Charge 1.0
FLDL2E	;	Charge $\log_2 e$
FLDL2T	;	Charge $\log_2 10$
FLDLG2	;	Charge $\log_{10} 2$
FLDLN2	;	Charge $\log_e 2$
FLDPI	;	Charge π
FLDZ	;	Charge 0.0

Description :

Il y a empilement. La valeur constante déterminée par l'instruction elle-même est rangée dans le nouveau sommet de pile.

Fonctionnement :

```
FPush();
switch (CODEOP) {
      case FLD1     :   ST = +1.0;      break;
      case FLDL2E   :   ST = log2e;     break;
      case FLDL2T   :   ST = log210;    break;
      case FLDLG2   :   ST = log102;    break;
      case FLDLN2   :   ST = loge2;     break;
      case FLDPI    :   ST = π;         break;
      case FLDZ     :   ST = +0.0;      break;
}
```

Exemples :

```
FLD1      Avant :   TOP :   6
          Après :   TOP :   5   ST :   1.0 * 10^0

FLDPI     Avant :   TOP :   0
          Après :   TOP :   7   ST :   3.1416 * 10^0
```

Exception :

Float(IS);

Note :

Voir l'exemple 9, page 568.

FLDCW

Chargement du mot de contrôle

Format :

FLDCW mem16

Description :

FLDCW charge le mot de contrôle avec la valeur trouvée dans mem16. Il y a génération d'une exception si un des indicateurs d'exception de SW est démasqué par le nouveau CW.

Fonctionnement :

```
CW = mem16;
```

Exemple :

```
FLDCW mem16   Avant :   CW :   XXXX   mem16 :   1F4F
              Après :   CW :   1F4F   mem16 :   1F4F
```

Exception :

Float(I,IS,D,Z,O,U,P,S)

FLDENV

Chargement de l'environnement

Format :

FLDENV mem

Description :

FLDENV charge l'environnement de la machine à partir d'une zone donnée de la mémoire. L'environnement comprend : le mot de contrôle, le mot d'état, le mot d'étiquette et les registres pointeurs d'erreur pour la dernière instruction à virgule flottante exécutée. Dans le Chapitre 2, nous donnons un descriptif détaillé du format des registres pointeurs d'erreur. Ces derniers contiennent des informations sur le dernier CodeOp et sur les données référencées par ce CodeOp.

Fonctionnement :

```
CW = mem;
SW = [&mem+4];
TW = [&mem+8];
LdErrorPoint(&mem);
```

Exemple :

```
FLDENV mem
   Avant :  CW  :   XXXX   SW  :   XXXX        TW :   XXXX   FIP :   XXXXXXXX
            FCS :   XXXX   FDP :   XXXXXXXX   FDS :   XXXX
```

```
        [&mem+0 ] :   XXXX137F      [&mem+4 ] :   XXXX0400
        [&mem+8 ] :   XXXXA0FE      [&mem+12] :   0000FFD0
        [&mem+16] :   XXXX0241      [&mem+20] :   005F3DD0
        [&mem+24] :   XXXX3015
Après : CW  :   137F   SW  :   0400        TW  :   A0FE   FIP :   0000FFD0
        FCS :   0241   FDP :   005F3DD0    FDS :   3015
        [&mem+0 ] :   XXXX137F      [&mem+4 ] :   XXXX0400
        [&mem+8 ] :   XXXXA0FE      [&mem+12] :   0000FFD0
        [&mem+16] :   XXXX0241      [&mem+20] :   005F3DD0
        [&mem+24] :   XXXX3015
```

Exception :

Float(I,IS,D,Z,O,U,P,S)

FMUL/FIMUL/FMULP

Multiplication

Formats :

```
FMUL                          ; Multiplication réelle ST(1), ST
FMUL    op1                   ; Multiplication réelle
        ST(n)
        mem32                 ; réel court
        mem64                 ; réel long

FIMUL   op1                   ; Multiplication entière
        mem16                 ; mot entier
        mem32                 ; entier court

FMULP   op1     , op2         ; Multiplication réelle et dépilage
        ST      , ST(n)
        ST(n)   , ST

FMUL    op1     , op2         ; Multiplication réelle
        ST      , ST(n)
        ST(n)   , ST
```

Description :

L'instruction effectue une multiplication sur les opérandes à virgule flottante définis explicitement ou implicitement (ST*ST(1), op_1*ST, ou op_2*op_1, si 0, 1, ou 2 opérandes sont précisés). Le résultat est rangé au sommet de la pile dans tous les cas sauf pour la forme à deux opérandes où la destination peut être ST(n).

Si op_1 est un opérande mémoire (mot entier ou entier court, réel court ou réel long), il est automatiquement converti en réel temporaire (le format interne) avant tout traitement.

Un FMUL sans opérandes entraîne un dépilage. Ainsi, FMUL sans opérandes est synonyme de FMULP ST(1),ST.

Fonctionnement :

```
if (NUMOPS == 0){
      ST(1) = ST * ST(1);
      FPop();    /* dépile et résultat dans ST */
      } else if (NUMOPS == 1){
         ST = (tempReal) op1 * ST;
} else {       /* NUMOPS == 2 */
      op1 = op2 * op1;
      if (OPCODE == FMULP){
         FPop();    /* Dépile */
      }
}
```

Exemple :

FMULP ST,ST(3)	Avant :	ST	:	$8.410 * 10^{-12}$	TOP :	4
		ST(3)	:	$9.934 * 10^{9}$		
	Après :	ST	:	$8.354 * 10^{-2}$	TOP :	3
		ST(2)	:	$9.934 * 10^{9}$		

Exceptions :

Memory();
Float(IS,I,O,P,D,U);

Notes :

Le tableau suivant résume toutes les possibilités de multiplication avec les infinis.

Opérandes		Résultats (op_1)	
op_1	op_2	Signe	Valeur
$\pm\infty$	$\pm\infty$	Ou exclusif des signes	∞
$\pm x$	$\pm\infty$	Ou exclusif des signes	∞
± 0	$\pm\infty$	Opération non valide	
$\pm\infty$	± 0	Opération non valide	

où $0 \leq x < \infty$.

Voir l'exemple 10, page 570.

FNOP

Pas d'opération

Format :

FNOP

Description :

FNOP range le sommet de la pile dans le sommet de la pile (*sic* !), ce qui, effectivement, est un NOP (*resic* !).

Fonctionnement :

```
ST = ST;
```

Exemple :

```
FNOP
```

Exceptions :

Aucune.

FPATAN

Arctangente partiel

Format :

FPATAN

Description :

L'instruction calcule l'arctangente de ST(1)/ST. Puis, elle dépile une fois et place le résultat au nouveau sommet de la pile.

Le 80387 n'impose aucune limite à l'opérande d'une instruction FPATAN. Cependant, les résultats respectent les bornes données par le tableau suivant :

Signe(ST(1))	Signe(ST)	|ST(1)|<|ST|	Résultat
+	+	Oui	ATAN(ST(1)/ST
+	+	Non	π/2 - ATAN(ST(1)/ST)
+	-	Oui	π - ATAN(ST(1)/-ST)
+	-	Non	π/2 - ATAN(-ST(1)/ST)
-	+	Oui	-ATAN(ST(1)/ST)
-	+	Non	-π/2 + ATAN(ST(1)/-ST)
-	-	Oui	-π + ATAN(-ST(1)/-ST)
-	-	Non	-π/2 - ATAN(-ST(1)/-ST)

Fonctionnement :

```
ST(1) = arctan(ST(1) / ST);
FPop(); /* Résultat dans ST après FPop */
```

Exemple :

FPATAN	Avant :	ST	:	$4.51200 * 10^5$	TOP :	4
		ST(1)	:	$3.664 * 10^3$		
	Après :	ST	:	$8.12039 * 10^{-3}$	TOP :	5

Exception :

Float(IS,I,D,U,P);

Notes :

Les opérandes du 80287 doivent être compris dans les limites suivantes :

$0 \leq |ST(1)| < |ST(0)| < +\infty$

alors que le 80387 n'a pas ces restrictions.

FPREM

Reste partiel

Format :

FPREM

Description :

FPREM calcule le reste partiel de ST/ST(1). Ce reste est exact (pas d'erreur de précision possible). Le reste est calculé par une série de soustractions successives. Quand les opérandes ont des valeurs absolues très différentes, ces séries de soustractions peuvent prendre beaucoup de temps. Pour éviter une dégradation possible du traitement des interruptions, l'instruction ne fait que calculer partiellement le reste, et une boucle logicielle doit finir le travail. D'où le mnémonique : reste partiel.

L'instruction permet de réduire une différence de grandeur de 2^{64} au mieux en une exécution. Si la réduction est complète, le code de condition 2 est mis à 0 et les bits de code de condition 0, 3, et 1 représentent les 3 bits les moins significatifs du quotient. Si la réduction est incomplète, le code de condition est mis à 1.

Le résultat suit la relation :

REM = ST - ST(1) * quotient;

où le reste a toujours le signe du ST initial.

FPREM était une instruction importante sur le 80287, puisque la plupart des fonctions périodiques transcendantales y sont restreintes à

l'intervalle 0 et $\pi/4$. FPREM réduisait alors l'argument à une valeur acceptable. Puisque de telles limites ont été supprimées sur le 80387 (hourra !), cette instruction perd de son importance.

Fonctionnement :

```
if(|ST| < |ST(1)| * 2^64){
      C2 = 0;
      quotient = ST / ST(1);
      ST = ST % ST(1);
      if ((quotient % 8)   4) C0 = 1;
      if ((quotient % 4)   2) C3 = 1;
      if ((quotient % 2)   1) C1 = 1;
} else {
      if (ST>0){
         ST = ST - ST(1) * (2^64 - 1);
      } else { ST = ST + ST(1) * (2^64 - 1); }
      C2 = 1;
}
```

Exemple :

```
                                                CCCC
                                                3210
FPREM   Avant :   ST    :      -4.317 * 10^2    XXXX
                  ST(1) :      6.283 * 10^0
        Après :   ST    :      -4.443 * 10^0    0000
                  ST(1) :      6.283 * 10^0
```

Exception :

Float(IS,I,U,D);

Notes :

Les codes de condition (C0,C3,C1) sont positionnés sur le 80387, mais pas sur le 80287.

Voir aussi FPREM1.

Si ST(1) avait initialement la valeur ∞, ST reste inchangé et le quotient (codes de condition) est mis à 0.

Voir l'exemple 8, page 567.

FPREM1

Reste partiel - IEEE

Format :

FPREM1

Description :

FPREM1 calcule le reste partiel de ST/ST(1). Ce reste est exact (pas d'erreur de précision possible). Le reste est calculé par une série de soustractions successives. Quand les opérandes ont des valeurs absolues très différentes, ces séries de soustractions peuvent prendre beaucoup de temps. Pour éviter une dégradation possible du traitement des interruptions, l'instruction ne fait que calculer partiellement le reste, et une boucle logicielle doit finir le travail. D'où le mnémonique : reste partiel.

L'instruction permet de réduire une différence de grandeur de 2^{64} au mieux en une exécution. Si la réduction est complète, le code de condition 2 est mis à 0 et les bits de code de condition 0, 3, et 1 représentent les 3 bits les moins significatifs du quotient. Si la réduction est incomplète, le code de condition est mis à 1.

Le résultat suit la relation :

REM = ST - ST(1) * quotient;

où le quotient est l'entier le plus près de la valeur exacte de ST/ST(1). De plus :

$|Q - ST/ST(1)| = 1/2$

Q est pair. Le mode d'arrondi et le contrôle de la précision n'affectent pas les résultats sauf si ST(1) divise exactement ST. Dans ce cas, le résultat est un zéro négatif pour un arrondi à l'entier inférieur et un zéro positif pour les autres modes d'arrondi.

FPREM était une instruction importante sur le 80287, puisque la plupart des fonctions périodiques transcendantales y sont restreintes à l'intervalle 0 et $\pi/4$. FPREM réduisait alors l'argument à une valeur acceptable. Puisque de telles limites ont été supprimées sur le 80387 (hourra !), cette instruction perd de son importance.

Fonctionnement :

```
if(|ST| < |ST(1)| * 2^64){
      C2 = 0;
      quotient = ST / ST(1);
      ST = ST % ST(1);
      if ((quotient % 8)   4) C0 = 1;
      if ((quotient % 4)   2) C3 = 1;
      if ((quotient % 2)   1) C1 = 1;
} else {
      if (ST>0){
         ST = ST - ST(1) * (2^64 - 1);
      } else { ST = ST + ST(1) * (2^64 - 1); }
      C2 = 1;
}
```

Exemple :

```
                                                CCCC
                                                3210
FPREM1   Avant :   ST    :      -4.317 * 10²     XXXX
                   ST(1) :      6.283 * 10⁰
         Après :   ST    :      1.827 * 10⁰      0010
                   ST(1) :      6.283 * 10⁰
```

Exception :

Float(IS,I,U,D);

Notes :

FPREM1 est une nouveauté du 80387. Il permet la compatibilité IEEE. Les exemples de FPREM et FPREM1 montrent les différences qui existent entre les deux instructions.

Notez la différence avec FPREM. Le reste de FPREM1 est compris dans l'intervalle abs(ST(1)/2) et abs(ST(1)/2), alors que FPREM donne un reste dans l'intervalle 0 et abs(ST(1)) (ou abs(ST(1)) pour des dividendes négatifs ST) pour des dividendes positifs ST. Cela peut donner une différence de 1 entre les codes de condition de FPREM et FPREM1.

Si ST(1) avait initialement la valeur ∞, ST reste inchangé et le quotient (codes de condition) est mis à 0.

FPTAN

Tangente partielle

Format :

FPTAN

Description :

L'instruction calcule la tangente de ST, place le résultat dans le sommet de la pile, et empile 1,0. Ainsi :

ST(1)/ST = tan(ST')

où ST' est le sommet de la pile avant l'exécution de l'instruction FPTAN. ST est toujours 1,0 et ST(1) est la tangente de ST'.

Tout opérande de FPTAN doit varier de 0 à $\pi * 2^{62}$. Si l'opérande source est compris dans ces bornes, C2 est mis à 0. Sinon, C2 est mis à 1, ST n'est pas modifié ainsi que le pointeur de sommet de pile (TOP).

Fonctionnement :

```
if (ST < π * 2^62) {
      ST = tan(ST);
      FPush();
      ST = 1.0;
      C2 = 0;
} else {
      /* Notez que ST et TOP ne sont pas modifiés */
```

```
        C2 = 1;
}
```

Exemple :

FPTAN	Avant :	ST	:	$9.47103 * 10^{22}$	TOP :	4	C2 :	X
	Après :	ST	:	$1.00000 * 10^{0}$	TOP :	3	C2 :	0
		ST(1)	:	$1.51273 * 10^{0}$				

Exception :

Float(IS,I,D,U,P);

FRNDINT

Arrondi à l'entier

Format :

FRNDINT

Description :

L'élément situé au sommet de la pile est arrondi à un entier. La procédure d'arrondi suit la valeur du champ RC (contrôle d'arrondi) du mot de contrôle. Les quatre modes d'arrondi sont : troncature (11), moins l'infini (01), plus l'infini (10), et au plus près (00).

Fonctionnement :

```
ST = (int) ST;
```

Exemples :

```
                                ; Arrondi au plus près
FRNDINT   Avant :   ST :   -4.317 * 10⁰   RC :   00
          Après :   ST :   -4.000 * 10⁰   RC :   00
                                ; Arrondi vers moins l'infini
FRNDINT   Avant :   ST :   -4.317 * 10⁰   RC :   01
          Après :   ST :   -5.000 * 10⁰   RC :   01
```

Exception :

Float(IS,I,D,P);

FRSTOR

Récupération de l'état

Format :

FRSTOR mem

Description :

FRSTOR recharge l'état complet du 80387 à partir de l'emplacement de 108 octets donné par mem. Sont inclus l'environnement du 80387 et les huit registres de la pile à virgule flottante.

Le format de la mémoire est identique à celui de l'instruction FSAVE.

Une exception peut être générée si la combinaison du mot d'état et du mot de contrôle donne une exception.

Fonctionnement :

```
CW = mem;
SW = [&mem+4];
TW = [&mem+8];
LdErrorPoint(&mem);
for (i=0;i<8;i++){
      STi = [&mem+28+i*10];
}
```

Exemple :

```
FRSTOR mem        ; mem est un emplacement mémoire de 108 octets.
```

Exceptions :

Float(I,IS,D,Z,O,U,P,S)
Memory();

FSAVE/FNSAVE

Sauvegarde de l'état

Formats :

FSAVE mem
FNSAVE mem

Description :

FSAVE/FNSAVE sauvegardent l'état complet du 80387 dans l'emplacement mémoire mem. L'environnement comprend : le mot de contrôle, le mot d'état, le mot d'étiquette, les registres pointeurs d'erreur et la pile à virgule flottante complète (8 registres 80b). Dans le Chapitre 2, nous donnons le format détaillé des pointeurs d'erreur et de la zone de sauvegarde. Les pointeurs d'erreur contiennent des informations sur la dernière instruction exécutée et sur toute adresse de données utilisée.

Après avoir sauvé l'état du 80387, FSAVE initialise le 80387, comme le fait FINIT.

FSAVE teste les exceptions numériques non masquées, alors que FNSAVE ne le fait pas.

Fonctionnement :

```
mem = CW;
[&mem+4] = SW;
[&mem+8] = TW;
StErrorPointer(&mem);
for (i=0;i<8;i++){
      [&mem+28+i*10] = STi;
```

```
}
FINIT;
```

Exemple :

```
FSAVE mem        ; mem est un emplacement mémoire de 108 octets.
```

Exception :

Memory();

FSCALE

Echelle de puissance de deux

Format :

FSCALE

Description :

Le sommet de la pile subit une mise à l'échelle à la puissance de deux donnée par ST(1). La valeur dans ST(1) est traitée comme un entier et est ajoutée à l'exposant de ST. Cette instruction est très utile pour multiplier rapidement par des puissances de deux.

FSCALE utilise le plus petit entier le plus proche de ST(1) en valeur absolue (ST(1) est donc tronqué).

Fonctionnement :

$$ST = ST * 2^{ST(1)}$$

Exemples :

FSCALE	Avant :	ST	:	$-1.0111b * 2^{6}$
		ST(1)	:	$-1.0110b * 2^{4}$
	Après :	ST	:	$-1.0111b * 2^{-16}$
		ST(1)	:	$-1.0110b * 2^{4}$

Exception :

Float(IS,I,D,O,U,P);

Notes :

Le 80387 n'impose pas de restrictions sur les valeurs de ST(1) au contraire du 80287. Les opérandes compris entre les bornes :

$0 < |ST(1)| < 1$

sont considérés comme nuls, alors qu'ils donnaient un résultat indéfini sur le 80287. Les opérandes peuvent engendrer des exceptions du type débordement.

Le tableau suivant résume les opérations FSCALE avec les infinis :

Opérandes		Résultats (ST)	
ST	**ST(1)**	**Signe**	**Valeur**
$\pm\infty$	$-\infty$	Opération non valide	
$\pm\infty$	$+\infty$	Signe de ST	∞
$\pm\infty$	$\pm z$	Signe de ST	∞
± 0	$\pm\infty$	Signe de ST	0
$\pm x$	$+\infty$	Débordement	
$\pm x$	$-\infty$	Débordement	

où $0 < x < \infty$ et $0 \leq z < \infty$

FSIN

Sinus

Format :

FSIN

Description :

L'instruction calcule le sinus de ST, range le résultat au sommet de la pile, et empile 1. Ainsi

ST(1)/ST = sin(ST')

où ST' est l'ancien sommet de la pile, ST est toujours égal à 1, et ST(1) le sinus de ST'.

L'opérande du FSIN doit être compris entre 0 et $\pi * 2^{62}$. Si l'opérande source remplit ces conditions, C2 est mis à 0. Sinon, C2 est mis à 1, et ST et le pointeur de sommet de pile (TOP) ne sont pas modifiés.

Fonctionnement :

```
if (ST < π * 2^62) {
      ST = sin(ST);
      FPush();
      ST = 1.0;
      C2 = 0;
} else {
      /* Notez que ST et TOP ne sont pas modifiés */
```

```
        C2 = 1;
}
```

Exemple :

FSIN	Avant :	ST	:	$8.41229 * 10^{3}$	TOP :	4	C2 :	X
	Après :	ST	:	$1.00000 * 10^{0}$	TOP :	3	C2 :	0
		ST(1)	:	$-7.80288 * 10^{-1}$				

Exception :

Float(IS,I,D,U,P);

Note :

Cette instruction est une nouveauté du 80387.

FSINCOS

Sinus et cosinus

Format :

FSINCOS

Description :

L'instruction calcule le sinus et le cosinus de ST, range le sinus au sommet de la pile, et empile le cosinus. Ainsi :

ST(1) = sin(ST')
ST = cos(ST')

où ST' est l'ancien sommet de la pile, ST est toujours égal à 1, et ST(1) est le sinus de ST'.

L'opérande du FSINCOS doit être compris entre 0 et $\pi * 2^{62}$. Si l'opérande source remplit ces conditions, C2 est mis à 0. Sinon, C2 est mis à 1, et ST et le pointeur de sommet de pile (TOP) ne sont pas modifiés. Le sinus calculé par FSINCOS peut être moins précis que celui calculé par les instructions FSIN et FCOS.

Fonctionnement :

```
if (ST < π * 2^62) {
      tmp = cos(ST)
      ST = sin(ST);
```

```
        FPush();
        ST = tmp;
        C2 = 0;
} else {
        /* Notez que ST et TOP ne sont pas modifiés */
        C2 = 1;
}
```

Exemple :

FSINCOS	Avant :	ST	:	$2.97421 * 10^{6}$	TOP :	4	C2 :	X
	Après :	ST	:	$9.86024 * 10^{-1}$	TOP :	3	C2 :	0
		ST(1)	:	$1.66602 * 10^{-1}$				

Exception :

Float(IS,I,D,U,P);

Note :

Cette instruction est une nouveauté du 80387.

FSQRT

Racine carrée

Format :

FSQRT

Description :

Le sommet de la pile est remplacé par sa racine carrée. Notez que :

FSQRT(-0) = -0
FSQRT(+∞) = +∞

et que FSQRT(-∞) est une opération non valide.

Fonctionnement :

```
ST = sqrt(ST);
```

Exemple :

```
FSQRT   Avant :   ST :   8.410 * 10⁴
        Après :   ST :   2.900 * 10²
```

Exception :

Float(IS,I,U,P,D);

Note :

Voir l'exemple 11, page 574.

FST

Stockage de réel

Formats :

FST	op_1	
	mem32	;réel court
	mem64	;réel long
	ST(i)	

Description :

L'instruction range le sommet de la pile dans op_1. Si op_1 est un réel court ou long, le sommet de la pile est tout d'abord converti au format associé à RC (contrôle d'arrondi). Si le sommet de la pile est un NaN ou un infini, l'exposant et la mantisse du sommet de la pile sont tronqués plutôt qu'arrondis, afin de respecter la taille de op_1.

Fonctionnement :

```
if (op1 is mem32){
      op1 = (shortReal) ST;
} else if (op1 is mem64){
      op1 = (longReal) ST;
} else {
      op1 = ST;
}
```

Exemple :

```
FST mem64        ; mem64 est un réel long
      Avant :   ST     :   2.9921 * 10^-9      TOP :   2
                mem64  :   X.XXXX * 10^X
      Après :   ST     :   2.9921 * 10^-9      TOP :   2
                mem64  :   2.9921 * 10^-9
```

Exceptions :

Memory();
Float(IS,I,O,U,P);

Notes :

L'exception D n'est pas transmise durant FST afin d'assurer la compatibilité avec le 80287.

FST ne peut pas ranger un réel temporaire en mémoire, alors que FSTP le peut.

FSTCW/FNSTCW

Stockage du mot de contrôle

Formats :

FSTCW mem16
FNSTCW mem16

Description :

FSTCW/FNSTCW rangent le mot de contrôle dans mem16. FSTCW teste les exceptions numériques non masquées, alors que FNSTCW ne le fait pas.

Fonctionnement :

```
mem16 = CW;
```

Exemple :

```
FSTCW mem16   Avant :   CW :   1F4D   mem16 :   XXXX
              Après :   CW :   1F4D   mem16 :   1F4D
```

Exceptions :

Aucune.

FSTENV/FNSTENV

Stockage de l'environnement

Formats :

FSTENV	mem
FNSTENV	mem

Description :

FSTENV/FNSTENV sauvegardent l'environnement de la machine à l'emplacement mémoire donné. L'environnement mémoire comprend : le mot de contrôle, le mot d'état, le mot d'étiquette et les pointeurs d'erreur associés à la dernière instruction à virgule flottante exécutée. Dans le Chapitre 2, nous donnons le format détaillé des pointeurs d'erreur. Ils contiennent des informations sur la dernière instruction exécutée et sur toutes les données utilisées dans cette instruction.

FSTENV teste les exceptions numériques non masquées, alors que FNSTENV ne le fait pas.

Fonctionnement :

```
mem = CW;
[&mem+4] = SW;
[&mem+8] = TW;
StErrorPointer(&mem);
```

Exemples :

```
FSTENV mem
   Avant :  CW  :  137F   SW  :   0400       TW  :   A0FE   FIP :   0000FFD0
            FCS :  0241   FOS :   005F3DD0   FDS :   3015
            [&mem+0]  :   XXXXXXXX     [&mem+4]  :   XXXXXXXX
            [&mem+8]  :   XXXXXXXX     [&mem+12] :   XXXXXXXX
            [&mem+16] :   XXXXXXXX     [&mem+20] :   XXXXXXXX
            [&mem+24] :   XXXXXXXX
   Après :  CW  :  137F   SW  :   0400       TW  :   A0FE   FIP :   0000FFD0
            FCS :  0241   FOS :   005F3DD0   FDS :   3015
            [&mem+0]  :   XXXX137F     [&mem+4]  :   XXXX0400
            [&mem+8]  :   XXXXA0FE     [&mem+12] :   0000FFD0
            [&mem+16] :   XXXX0241    [&mem+20]  :   005F3DD0
            [&mem+24] :   XXXX3015
```

Exception :

Memory();

FSTP

Stockage de réel et dépilage

Formats :

FSTP	op_1	
	mem32	;réel court
	mem64	;réel long
	ST(i)	

Description :

L'instruction range le sommet de la pile dans op_1. Si op_1 est un réel court ou long, le sommet de la pile est tout d'abord converti au format associé à RC (contrôle d'arrondi). Si le sommet de la pile est un NaN ou un infini, l'exposant et la mantisse du sommet de la pile sont tronqués plutôt qu'arrondis, afin de respecter la taille de op_1.

FSTP peut stocker des nombres réels temporaires en mémoire, alors que FST ne le peut pas.

Fonctionnement :

```
if (op1 is mem32){
      op1 = (shortReal) ST;
} else if (op1 is mem64){
      op1 = (longReal) ST;
} else {
      op1 = ST;
}
FPop();
```

Exemple :

```
FSTP mem64        ; mem32 est un réel court
      Avant :   ST    :   8.9901 * 10^-102   TOP :   7
                mem64 :   X.XXXX * 10^X
      Après :   ST    :   8.9901 * 10^-102   TOP :   0
                mem64 :   8.9901 * 10^-102
```

Exceptions :

Memory();
Float(IS,I,O,U,P);

Notes :

L'exception D n'est pas transmise durant FST afin d'assurer la compatibilité avec le 80287.

Voir l'exemple 10, page 570.

FSTSW/FNSTSW

Stockage du mot d'état

Formats :

FSTSW mem16
FNSTSW mem16

Description :

FSTSW/FNSTSW rangent le mot d'état courant dans mem16. FSTSW teste les exceptions numériques non masquées, alors que FNSTSW ne le fait pas.

Fonctionnement :

```
mem16 = SW;
```

Exemple :

```
FSTSW mem16   Avant :   SW :   FF81   mem16 :   XXXX
              Après :   SW :   FF81   mem16 :   FF81
```

Exception :

Memory();

FSTSW AX/FNSTSW AX

Stockage du mot d'état dans AX

Formats :

FSTSW AX
FNSTSW AX

Description :

FSTSW/FNSTSW rangent le mot d'état courant dans le registre AX du 80386. Ces instructions sont utiles pour travailler sur les indicateurs du 80387.

FSTSW teste les exceptions numériques non masquées, alors que FNSTSW ne le fait pas.

Fonctionnement :

```
AX = SW;
```

Exemple :

```
FSTSW AX   Avant :   SW :   0900   AX :   XXXX
           Après :   SW :   0900   AX :   0900
```

Exceptions :

Aucune.

Note :

Voir l'exemple 7, page 566.

FSUB/FISUB/FSUBP

Soustraction

Formats :

```
FSUB                          ;Soustraction réelle ST(1), ST
FSUB    op1                   ;Soustraction réelle
        ST(n)
        mem32                 ;réel court
        mem64                 ;réel long

FISUB   op1                   ;Soustraction entière
        mem16                 ;mot entier
        mem32                 ;entier court

FSUBP   op1     ,  op2        ;Soustraction réelle et dépilage
        ST      ,  ST(n)
        ST(n)   ,  ST

FSUB    op1     ,  op2        ;Soustraction réelle
        ST      ,  ST(n)
        ST(n)   ,  ST
```

Description :

Ces instructions soustraient les opérandes à virgule flottante définis explicitement ou implicitement (ST(1)-ST, ST-op_1, ou op_1-op_2, si 0, 1, ou 2 opérandes sont précisés). Le résultat est rangé au sommet de la pile dans tous les cas sauf pour la forme à deux opérandes où la destination peut être ST(n).

Si op_1 est un opérande mémoire (mot entier ou entier court, réel court ou réel long), il est automatiquement converti en réel temporaire (le format interne) avant tout traitement.

Un FSUB sans opérandes entraîne un dépilage. Ainsi, FSUB sans opérandes est synonyme de FSUBP ST(1),ST.

Fonctionnement :

```
if (NUMOPS == 0){
      ST(1) = ST(1) - ST;
      FPop();    /* dépile et résultat dans ST */
      } else if (NUMOPS == 1){
         ST = ST - (tempReal) op1;
} else {       /* NUMOPS == 2 */
      op1 = op1 - op2;
      if (OPCODE == FSUBP){
         FPop();    /* Dépile */
      }
}
```

Exemple :

```
FSUB ST,ST(3)    ;   ST = ST - ST(3);
      Avant :   ST     :   8.410 * 10^12   TOP :   7
                ST(3)  :   9.934 * 10^11
      Après :   ST     :   7.419 * 10^12   TOP :   7
                ST(3)  :   9.934 * 10^11
```

Exceptions :

Memory();
Float(IS,I,O,P,D,U);

Note :

Le tableau suivant résume toutes les possibilités de soustraction avec les infinis :

Opérandes		Résultats (op_1)	
op_1	op_2	Signe	Valeur
$+\infty$	$-\infty$	+	∞
$-\infty$	$+\infty$	-	∞
$+\infty$	$+\infty$	Opération non valide	
$-\infty$	$-\infty$	Opération non valide	
$\pm z$	$\pm\infty$	Signe de ∞	∞
$\pm\infty$	$\pm z$	Signe de z	∞

où $0 \leq z < \infty$.

FSUBR/FISUBR/FSUBRP

Soustraction arrière

Formats :

```
FSUBR                          ; Soustraction réelle arrière ST - ST(1)
FSUBR   op1                    ; Soustraction réelle arrière
        ST(n)
        mem32                  ; réel court
        mem64                  ; réel long

FISUBR  op1                    ; Soustraction entière
        mem16   ; Mot entier
        mem32   ; Entier court

FSUBRP op1      ,  op2         ; Soustraction réelle arrière et dépilage
        ST      ,  ST(n)
        ST(n)   ,  ST

FSUBR   op1     ,  op2         ; Soustraction réelle arrière
        ST      ,  ST(n)
        ST(n)   ,  ST
```

Description :

Ces instructions soustraient les opérandes à virgule flottante définis explicitement ou implicitement (ST-ST(1), op_1-ST, ou op_2-op_1, si 0, 1, ou 2 opérandes sont précisés). Le résultat est rangé au sommet de la pile dans tous les cas sauf pour la forme à deux opérandes où la destination peut être ST(n).

Si op_1 est un opérande mémoire (mot entier ou entier court, réel court ou réel long), il est automatiquement converti en réel temporaire (le format interne) avant tout traitement.

Un FSUBR sans opérandes entraîne un dépilage. Ainsi, FSUBR sans opérandes est synonyme de FSUBRP ST(1),ST.

Fonctionnement :

```
if (NUMOPS == 0){
      ST(1) = ST - ST(1);
      FPop();   /* dépile et résultat dans ST */
      } else if (NUMOPS == 1){
         ST = (tempReal) op1 - ST;
} else {      /* NUMOPS == 2 */
      op1 = op2 - op1;
      if (OPCODE == FSUBRP){
         FPop();   /* Dépile */
      }
}
```

Exemple :

```
FSUBR mem   ;   ST = mem - ST
      Avant :   ST   :   8.410 * 10^4
                mem  :   9934
      Après :   ST   :   7.74166 * 10^4
                mem  :   9934
```

Exceptions :

Memory();
Float(IS,I,O,P,D,U);

Note :

Le tableau suivant résume toutes les possibilités de soustraction arrière avec les infinis :

Opérandes		Résultats (op_1)	
op_1	op_2	Signe	Valeur
$+\infty$	$-\infty$	-	∞
$-\infty$	$+\infty$	+	∞
$+\infty$	$+\infty$	Opération non valide	
$-\infty$	$-\infty$	Opération non valide	
$\pm z$	$\pm\infty$	Signe de ∞	∞
$\pm\infty$	$\pm z$	Signe de z	∞

où $0 \leq z < \infty$.

FTST

Test

Format :

FTST

Description :

L'instruction compare numériquement le sommet de la pile avec 0 puis positionne les bits de code de condition du mot d'état selon le tableau suivant.

	C3	C2	C0
ST > 0.0	0	0	0
ST < 0.0	0	0	1
ST == 0.0	1	0	0
Non valide	1	1	1

On obtient une comparaison non valide lorsque ST est un NaN. Les comparaisons ignorent le signe du zéro.

Fonctionnement :

```
if (ST is NaN){
      C3 = 1;   C2 = 1;   C0 = 1;
} else if (ST > 0.0){
      C3 = 0;   C2 = 0;   C0 = 0;
} else if (ST < 0.0){
      C3 = 0;   C2 = 0;   C0 = 1;
} else if (ST == 0.0){
```

```
        C3 = 1;   C2 = 0;   C0 = 0;
}
```

Exemple :

FTST	Avant :	ST :	2.4560 * 10^6			TOP :	4		
		C3 :	X	C2 :	X	C0 :	X		
	Après :	ST :	2.4560 * 10^6			TOP :	4		
		C3 :	0	C2 :	0	C0 :	0		

Exceptions :

```
Float(IS,I,D);
if (ST is QUIETNaN) {
        FLoat(I);
}
```

FUCOM/FUCOMP/FUCOMPP

Comparaison non ordonnée

Formats :

```
FUCOM  op1      ; Comparaison non ordonnée
       mem32    ; réel court
       mem64    ; réel long
       ST(i)

FUCOMP op1      ; Comparaison non ordonnée et dépilage
       mem32    ; réel court
       mem64    ; réel long
       ST(i)
FUCOMPP         ; Comparaison non ordonnée et dépilage deux fois
```

Description :

L'instruction compare le sommet de la pile avec l'opérande donné, puis positionne les codes de condition selon le tableau suivant. Si le CodeOp est FUCOMPP, l'opérande comparé au sommet de la pile est ST(1).

	C3	C2	C0
ST > op$_1$	0	0	0
ST < op$_1$	0	0	1
ST == op$_1$	1	0	0
Non valide	1	1	1

FUCOMP dépile une fois, tandis que FUCOMPP dépile deux fois.

On obtient une comparaison non valide lorsque l'un des deux opérandes est un NaN. Les comparaisons ignorent le signe du zéro.

Fonctionnement :

```
if (CODEOP == FUCOMPP){
      op1 = ST(1);
}
if ((ST is NaN) || (op1 is NaN)){
      C3 = 1;   C2 = 1;   C0 = 1;
} else if (ST > op1){
      C3 = 0;   C2 = 0;   C0 = 0;
} else if (ST < op1){
      C3 = 0;   C2 = 0;   C0 = 1;
} else if (ST == op1){
      C3 = 1;   C2 = 0;   C0 = 0;
}
if (CODEOP == FUCOMP){
      FPop();
} else if (CODEOP == FUCOMPP){
      FPOP();
      FPop();
}
```

Exemple :

FUCOMPP	Avant :	ST	:	$6.1224 * 10^{-4}$							
		ST(1)	:	$5.3210 * 10^{-11}$							
		TOP	:	4	C3 :	X	C2 :	X	C0 :	X	
	Après :	TOP	:	6							
		C3	:	0	C2 :	0	C0 :	0			

Exception :

Float(IS,I,D);

Note :

FUCOM (et toutes les variantes qui dépilent) est identique à FCOM (et à ses variantes), sauf que FUCOM ne génère pas d'exception si l'opérande est un NaN silencieux, contrairement à FCOM.

FXAM

Examen

Format :

FXAM

Description :

L'instruction examine le sommet de la pile et modifie les codes de condition du mot d'état selon le tableau suivant.

C3	C2	C1	C0	Valeur de ST
0	0	0	0	+ Non permis
0	0	0	1	+ NaN
0	0	1	0	- Non permis
0	0	1	1	- NaN
0	1	0	0	+ Normalisé
0	1	0	1	+ ∞
0	1	1	0	- Normalisé
0	1	1	1	- ∞
1	0	0	0	+ 0
1	0	0	1	+ Vide
1	0	1	0	- 0
1	0	1	1	- Vide
1	1	0	0	+ Dénormalisé
1	1	0	1	+ Vide
1	1	1	0	- Dénormalisé
1	1	1	1	- Vide

Dans ce tableau, la notation *non permis* représente les entités suivantes : pseudo-NaN, pseudo-infini, anormal, ou pseudo-zéro (voir le Chapitre 1).

Fonctionnement :

```
if (ST < 0.0){
        C1 = 1;
}
if (ST is UNSUPPORTED){ C3 = 0; C2 = 0 ; C0 = 0; }
else if (ST is NaN){ C3 = 0 ; C2 = 0; C0 = 1; }
else if (ST is NORMAL){ C3 = 0 ; C2 = 1; C0 = 0; }
else if (ST is INFINITY){ C3 = 0 ; C2 = 1; C0 = 1; }
else if (ST is 0.0){ C3 = 1 ; C2 = 0; C0 = 0; }
else if (ST is DENORMAL){ C3 = 1 ; C2 = 1; C0 = 0; }
else if (ST is EMPTY){ C3 = 1 ; C2 = 0; C0 = 1; }
```

Exemple :

FXAM	Avant :	ST :	$3.21950 * 10^{-131}$			TOP :	2				
		C3 :	X	C2 :	X	C1 :	X	C0 :	X		
	Après :	ST :	$3.21950 * 10^{-131}$			TOP :	2				
		C3 :	0	C2 :	1	C1 :	0	C0 :	0		

Exceptions :

Aucune.

Note :

Voir la partie sur la notation à virgule flottante du Chapitre 1 pour des précisions complémentaires sur ces types de données.

FXCH

Echange de registres

Format :

FXCH
FXCH op_1
ST(n)

Description :

L'instruction échange le contenu de op_1 avec celui du sommet de la pile. Puisque plusieurs instructions du 80387 travaillent uniquement avec le sommet de la pile (par exemple : FSQRT, FSIN, FPATAN), FXCH permet d'employer ces instructions avec d'autres éléments de la pile.

Fonctionnement :

```
if (NUMOPS == 1){
      tmp = ST;
      ST = ST(n);
      St(n) = tmp;
} else {
      tmp = ST;
      ST = ST(1);
      ST(1) = tmp;
```

Exemple :

```
FXCH ST(3)   Avant :   ST    :   9.99321 * 10^-92   TOP :   3
                       ST(3) :   -7.22964 * 10^75
             Après :   ST    :   -7.22964 * 10^75   TOP :   3
                       ST(3) :   9.99321 * 10^-92
```

Exception :

Float(IS);

Note :

Voir l'exemple 11, page 574.

FXTRACT

Extraction de l'exposant et de la mantisse

Format :

FXTRACT

Description :

L'instruction place l'exposant (sans biais) du sommet initial de la pile au sommet de la pile (comme un nombre réel), puis empile la mantisse et le signe. Ainsi, l'ancien sommet de la pile est décomposé en une partie exposant ST(1) et une partie mantisse ST. La mantisse a un exposant zéro.

Fonctionnement :

```
tmp = ST;
if (tmp == 0){
      ST = -INFINITY;
} else {
      /* Notez que le biais (16383) est soustrait */
      ST = (tempReal) ((tmp >> 64) & 7FFFh) -16383;
}
FPush();
            /* 64 bits */
ST = (tempReal) (tmp & FFFFFFFFFFFFFFFFh/2^63);
if (tmp < 0.0){
      ST = -ST;
}
```

Exemple :

```
FXTRACT   Avant :   ST     :   1.1001101b * 2^39   TOP = 3
          Après :   ST     :   1.1001101b * 2^0    TOP = 2
                    ST(1)  :   1.00111b * 2^5
```

Exception :

Float(IS,I,Z,D);

Notes :

Si l'opérande est égal à 0, ST(1) (exposant) est mis à -∞ et ST (mantisse) à 0 avec le même signe que le sommet initial de la pile. Le 80287 se comporte différemment. Si l'opérande est égal à ∞, ST(1) (exposant) est mis à + ∞ et ST (mantisse) à ∞ avec le signe du sommet initial de la pile.

FYL2X

$y*\log_2 x$

Format :

FYL2X

Description :

L'instruction emploie les deux éléments du sommet de la pile pour calculer la fonction :

$$ST(1) * \log_2 ST$$

Puis, elle dépile et remplace le nouveau sommet de la pile par le résultat du calcul.

ST doit être compris entre 0 et $+\infty$ et l'opérande ST(1) entre $-\infty$ et $+\infty$. Sinon, l'instruction peut donner un résultat indéfini sans qu'une exception soit détectée.

On peut employer FYL2X pour calculer le log dans toute base puisqu'il y a multiplication. On peut le voir dans l'exemple suivant qui calcule $\log_n x$:

$$\log_n x = (1/\log_2 n) * \log_2 x$$

Fonctionnement :

```
if ((0    ST < +INFINITY) &&
      ( -INFINITY < ST(1) < +INFINITY)){
      tmp = ST(1) * log2(ST);
      FPop();
      ST = tmp;
} else {
      UNDEFINED;
}
```

Exemple :

FYL2X	Avant :	ST	:	$9.87311 * 10^{9}$
		ST(1)	:	$-2.79022 * 10^{-3}$
	Après :	ST	:	$-9.26377 * 10^{-2}$

Exception :

Float(IS,I,D,U,P);

Note :

Voir l'exemple 9, page 568.

FYL2XP1

$y*\log_2(x+1)$

Format :

FYL2XP1

Description :

L'instruction utilise les deux éléments du sommet de la pile pour calculer la fonction :

$$ST(1)*\log_2(ST+1.0)$$

puis dépile et range le résultat au nouveau sommet de la pile.

ST doit être compris entre :

$-(1-\sqrt{2}/2)$ et $+(1-\sqrt{2}/2)$

ST(1) peut être compris entre $-\infty$ et $+\infty$. Sinon, l'instruction donnera un résultat indéfini sans qu'une exception soit détectée.

Fonctionnement :

```
if ((-1-sqrt(2)/2) < ST < (1-sqrt(2)/2)) &&
      (( -INFINITY < ST(1) < +INFINITY )) {
      tmp = ST(1) * log2(ST + 1.0);
      FPop();
      ST = tmp;
} else {
      UNDEFINED;
}
```

Exemple :

FYL2XP1	Avant :	ST	:	$1.17230 * 10^{0}$
		ST(1)	:	$2.99013 * 10^{13}$
	Après :	ST	:	$-7.58600 * 10^{13}$

Exception :

Float(IS,I,D,U,P);

WAIT

Attente pour le coprocesseur

Format :

WAIT

Description :

WAIT entraîne l'arrêt du fonctionnement du 80386 qui se met en attente du coprocesseur numérique.

Cette instruction est utile pour synchroniser le processeur principal et le coprocesseur numérique.

Fonctionnement :

```
while (80387 is busy)
{
}
```

Exemple :

```
WAIT    ; Rien n'apparaît
```

Exceptions :

Aucune.

Note :

Cette instruction est utile uniquement après des rangements en mémoire d'opérandes à virgule flottante (FST(P), FIST(P) et FBSTP(P)), avant de les employer comme opérandes d'une instruction 80386. Cela permet de s'assurer que les données sont bien stockées par le 80387 avant que le 80386 n'y accède. On n'a pas besoin d'un WAIT si une autre instruction 80387 suit le rangement en mémoire. Les 8086/8087 et 80286/80287 ont besoin de l'instruction WAIT dans la plupart des cas.

4

Exemples du jeu d'instructions

Dans ce chapitre, nous allons présenter un ensemble d'exemples qui vont nous aider à mieux comprendre les instructions décrites dans le Chapitre 3 ainsi que des notions des Chapitres 1 et 2. Nous ne donnerons que des exemples sur des entiers et des nombres à virgule flottante. On trouvera dans le Chapitre 7 des exemples sur les instructions du type segments multiples et système d'exploitation.

Cela peut sembler surprenant, mais en onze exemples - six sur les entiers et cinq sur les nombres à virgule flottante - nous allons aborder chaque groupe d'instructions du 80386. Instructions du type entier : instructions arithmétiques, binaires, de transfert de contrôle, de transfert de données, de contrôle des indicateurs, de langage de haut niveau, instructions logiques et opérant sur les chaînes. Instructions du type virgule flottante : instructions arithmétiques, de comparaison, transcendantales, à constante et de contrôle.

SYNTAXE

Dans le Chapitre 2, nous avons présenté la syntaxe du langage d'assemblage qu'on emploie pour adresser les registres et la mémoire. Nous avons étendu cette syntaxe à des formes d'instructions à plusieurs opérandes (pour le 80386 et le 80387). Dans ce chapitre, nous allons donner des séquences d'instructions complètes qui implémentent quelques fonctions classiques. Ainsi, nous avons besoin d'étendre la syntaxe du langage d'assemblage que nous avions développée auparavant.

Cette syntaxe peut être différente de celle que vous employez sur votre machine.

Types

Toutes les variables et les références à la mémoire sont associées à des *types*. Le type d'une variable ou d'une référence détermine le nombre d'octets qui y est associé. Cette information permet à l'assembleur de générer des instructions correctes. Quelquefois, l'information sur le type peut être comprise dans l'opérande ou être liée à d'autres opérandes. Sinon, c'est au programmeur de donner explicitement le type. Par exemple :

```
INC    [ESI]
```

est ambigu, puisque le pointeur [ESI] ne précise pas si l'opérande est un octet (*byte*), un mot (*word*) ou un dmot (*dword*). On appelle cela un *pointeur anonyme*. Par contre, les exemples suivants sont valides :

```
INC    byte ptr [ESI]
INC    word ptr [ESI]
INC    dword ptr [ESI]
```

L'exemple suivant aussi est un cas ambigu :

```
MOV    [EBX],5
```

puisque la constante immédiate ne donne pas le type. Voici la forme valide :

```
MOV    word ptr [EBX],5
```

Cependant, les formes à deux opérandes n'ont besoin du type que pour un seul des opérandes. Par exemple :

```
MOV    [EBX],EAX
MOV    [EBX],AX
MOV    [EBX],AL
```

sont toutes valides, puisque le type du premier opérande donne implicitement celui du second. Le Tableau 4.1 donne une liste des types valides.

Notez que ptr, abréviation de pointeur, est omis. Par exemple, *byte ptr* donne le type de l'opérande mémoire (par exemple, [EBX]), mais ne signifie en aucun cas que [EBX] est un pointeur mémoire. C'est la même chose pour une forme à deux opérandes : l'opérande mémoire est anonyme dans MOV [EBX],EAX.

Type	Description
byte	1 octet
word	1 mot = 2 octets
dword	1 dmot = 4 octets
pword	6 octets
qword	8 octets
tword	10 octets
short	(court) étiquette dans un segment comprise entre +127 et -128 octets
near	(proche) étiquette dans un segment (courte ou longue)
far	(lointain) étiquette dans un segment différent

Tableau 4.1 : *Types valides.*

Allocation des données

Pour réserver de l'espace mémoire, vous pouvez utiliser db, dw, et dd pour des octets, des mots et des dmots. La forme générale vous permet de spécifier une variable (nom symbolique qui y fait référence), une taille, et une valeur initiale lors du stockage.

```
fred    db    1       ; définit 1 octet appelé fred
                      ; avec la valeur initiale 1.
joe     dw    10,8,7  ; définit 3 mots appelés joe
                      ; avec les valeurs initiales
                      ; 10,8,7.
        dd    ?       ; définit 1 dmot sans nom et
                      ; sans valeur initiale.
```

Il existe deux formes spéciales d'allocation de données. La première permet d'initialiser une suite d'octets (db) avec une chaîne de caractères. La seconde forme utilise la construction spéciale DUP pour initialiser plusieurs unités de stockage. La directive DUP peut être implémentée, comme dans l'exemple suivant :

```
string1   db    'Hello World'
stack     dd    50    dup (0)    ; 50 dmots, tous
                                 ; initialisés à 0
funny     dw    2     dup (3, 2 dup (?), 4)
                                 ; initialise 8 mots à
                                 ;3,?,?,4,3,?,?,4
```

Déplacement

L'opérateur offset retourne le déplacement d'une variable (ou d'un label) à partir de la base du segment dans lequel elle est définie. Voici un exemple :

```
string1      db   'Hello World'
string1_end  db   0
string2      db   'Goodbye World'
string2_end  db   0
<...>
MOV   ESI,offset string1
MOV   EAX,offset string1 - offset string2
```

Ce segment de code transfère le déplacement de string1 depuis la base du segment dans lequel string1 est définie vers le registre ESI. Le second exemple vous montre que vous pouvez employer l'opérateur offset dans des calculs arithmétiques pour former une constante immédiate.

Etiquettes

Les étiquettes (labels) sont des références symboliques à des adresses mémoire de la séquence de code. Elles permettent de se brancher facilement à n'importe quel endroit du programme.

```
label1:   MOV    EAX,1
             .
             .
             .
          LOOPNZ label1
```

PROC/ENDP

La directive PROC est un autre moyen pour définir des étiquettes et permet en outre de transformer une séquence d'instructions en un sous-programme. La séquence d'instructions qui représente ce sous-programme se termine par la directive ENDP. Habituellement, on accède au sous-programme grâce à l'instruction CALL.

```
JunkProc   PROC
      .
      .
JunkProc   ENDP
      .
      .
      .
call   JunkProc
```

EQU

La directive EQU vous permet d'associer un symbole à une expression. Vous pouvez utiliser cette directive pour donner un mnémonique différent à une constante, une étiquette, une adresse, ou un registre. Par exemple :

```
reg1   EQU   EAX
reg2   EQU   EBX
```

vous permet d'employer reg1 et reg2 au lieu des noms traditionnels des registres. Voici un autre exemple :

```
Length    EQU    100
My_word   EQU    word ptr [ESI+EBP]
```

Dans ce chapitre, vous remarquerez que nous utilisons la directive EQU pour faire référence à des variables locales ou temporaires relativement à la pile. Par exemple :

```
Error    EQU    SS: byte ptr [EBP-1]
```

Ceci définit un octet Error qui se trouve dans le segment pile adressé par SS au déplacement donné par le registre EBP moins 1.

EXEMPLES SUR LES ENTIERS

Dans ce paragraphe, nous allons donner des exemples d'instructions qui travaillent sur des entiers.

Division signée

Dans la description de l'instruction SAR (décalage arithmétique à droite) du Chapitre 3, nous notons que SAR est équivalent à une division signée par une puissance de deux, si ce n'est que l'arrondi du quotient se fait vers moins l'infini au lieu de 0 (cas de l'instruction IDIV). Par exemple, diviser - 17 par 4 donne un quotient de -4 en utilisant l'instruction IDIV. Cependant, si on utilise SAR pour diviser - 17 par 4, le quotient sera -5. Cette différence est expliquée dans l'exemple suivant :

```
IDIV ECX      Avant :   EDX :   FFFFFFFF   EAX :   FFFFFFEF
                        ECX :   00000004
              Après :   EDX :   FFFFFFFF   EAX :   FFFFFFFC
                        ECX :   00000004
SAR EAX,CL    Avant :   EAX :   FFFFFFEF   ECX :   00000002
              Après :   EAX :   FFFFFFFB   ECX :   00000002
```

Puisque IDIV est une instruction plutôt lente, vous pouvez écrire du code plus efficace en évitant de l'utiliser pour des divisions par des puissances de deux. Cependant, il est important que tous les diviseurs entiers

```
; Utilise SAR pour des divisions par puissances de deux
; Vérifie l'arrondi du quotient. Le premier exemple
; est le plus rapide si un ajustement n'est pas nécessaire.
; Le second est le plus rapide s'il y a ajustement
;   N:   la puissance de deux du diviseur
;   EAX: le dividende
;
; Routine de division par SAR N°1
   OR    EAX,EAX
   LEA   EAX,[EAX+2^N-1]
   JL    Adj
   SUB   EAX,2^N-1
Adj:
   SAR   EAX,N
;
; Routine de division par SAR N°2
;
   OR    EAX,EAX
   JGE   NoAdj
   ADD   EAX,2^N-1
NoAdj:
   SAR   EAX,N
```

Listing 4.1 : *Exemple 1 - Division signée.*

agissent de la même façon ! Le code du Listing 4.1 détecte d'éventuelles erreurs d'arrondi (un dividende négatif qui n'est pas une puissance de deux). De plus, il ajuste éventuellement le dividende, afin d'obtenir un arrondi correct après le décalage.

L'instruction IDIV sur un opérande du type registre dure 43 cycles d'horloge. Les routines du Listing 4.1 calculent le quotient en 12 ou 15 cycles si le dividende est positif, et en 17 ou 10 cycles s'il est négatif. Ainsi, les routines SAR sont bien plus rapides qu'un IDIV et elles arrondissent correctement.

Notez que nous ne considérons pas la division par SAR comme un sous-programme. A cause des cycles d'horloge demandés par les instructions d'appel et de retour associées au sous-programme, on perd pratiquement tous les avantages de la routine précitée. Ainsi, pour obtenir les performances désirées, on doit exécuter directement la division par SAR. Pour réduire le temps de saisie et pour disposer d'une forme plus concise, vous pouvez employer des EQU ou des macros (si votre assembleur vous le permet).

Tri

Le tri d'une liste de nombres est un classique du genre. Le Listing 4.2 donne une fonction qui effectue le tri d'une liste de nombres pointée par ESI. La longueur de la liste est donnée par ECX. L'algorithme employé ici est le tri à bulles (*bubble sort*).

Un *bubble sort* (tri à bulles) comporte deux boucles imbriquées (le temps d'exécution est un $O(n^2)$). La boucle interne compare chaque élément avec celui qui se situe immédiatement au-dessus dans la liste. Si ce dernier est le plus petit, les deux éléments sont permutés. Ainsi, le plus grand monte telle une bulle au sommet de la liste. Cette boucle interne se répète autant de fois qu'il y a d'éléments. A chaque itération, elle n'a besoin de travailler que sur un nombre de données égal à la valeur du compteur de la boucle externe moins 1. Ce compteur de boucle externe est décrémenté à partir de la longueur initiale de la liste. Après la première

```
; Tri : Trie la liste de nombres pointée par ESI. La
; longueur est donnée par ECX. La longueur est égale
; au compteur - 1 puisqu'on compte à partir de 0
; (ex: [0:ECX]). Le plus petit nombre de la liste est
; rangé dans ESI, et le plus grand nombre dans [ESI+ECX*4].
; L'algorithme de tri employé est le tri à bulles.
; On considère une liste de dmots.
;   ESI: Pointeur sur le début de la chaîne
:   ECX: Compteur du nombre d'éléments de la liste.
SORT   PROC
OutLoop:
   MOV   DX,0
   InnerLoop:
   CMP   EDX,ECX
   JGE   Bottom
   MOV   EAX,[ESI+EDX*4+4]
   CMP   [ESI+EDX*4],EAX
   JGE   NoSwap
   ; [ESI+EDX*4]>[ESI+EDX*4+4] permutation
   XCHG  [ESI+EDX*4+4],EAX
   MOV   [ESI+EDX*4],EAX
   NoSwap
   INC   EDX
   JMP   InnerLoop
   Bottom:
   LOOP  OutLoop
Done:
RET
SORT   ENDP
```

Listing 4.2 : *Exemple 2 - Tri.*

exécution de la boucle interne, la donnée la plus grande doit se trouver à l'adresse la plus haute. Après la seconde itération, la donnée la plus grande doit être à l'adresse la plus haute, la seconde plus grande à la seconde adresse la plus haute, etc.

Ainsi, la seconde itération de la boucle interne n'a pas besoin de comparer la seconde adresse la plus haute avec l'adresse la plus haute, puisqu'on sait que cette dernière est la plus grande, etc.

Factorielle

L'exemple du Listing 4.3 est une routine dont le paramètre se trouve dans le registre EAX. Cette routine calcule la factorielle de ce nombre et retourne le résultat dans le registre EAX. L'équation mathématique qui décrit une factorielle est :

$$n! = n*(n-1)*(n-2)*...*(1)$$

La routine présentée utilise un appel récursif à une procédure. La procédure initiale (Fact) effectue quelques tests puis appelle la procédure récursive _Fact. _Fact teste si l'argument transmis est égal à 0. S'il n'est pas nul, _Fact est appelé récursivement avec l'argument moins 1. Si l'argument est nul, alors la routine met à 1 le registre résultat (EAX moins les résultats intermédiaires calculés auparavant) et retourne au programme appelant. A chaque retour à l'appelant, le produit intermédiaire est multiplié par la valeur de l'argument transmis à _Fact lors de l'appel récursif. Ainsi, le programme appelle _Fact n+1 fois, avec le n+1ème appel recevant 0 comme argument. Au retour, le programme effectue la multiplication (1*1, 1*2, 2*3, 6*4,...).

```
; Fact: Calcule récursivement la factorielle de la valeur
; donnée par le registre EAX. Le résultat de ce calcul
; est retourné dans EAX. Si une erreur apparaît lors
; du calcul, le programme retourne la valeur -1 dans EAX.
;   EAX: nombre dont on veut calculer la factorielle
```

Listing 4.3 : *Exemple 3a - Factorielle récursive.*

```
;        Après l'exécution, EAX contient le résultat
; Procédure récursive 'dummy' de Fact : _Fact
;
Error   EQU   SS:byte ptr [EBP-1]
_Fact   PROC
   CMP   ECX,0
   JZ    done
   PUSH  ECX
   DEC   ECX
   CALL  _Fact
   POP   ECX
   IMUL  EAX,ECX
   SETO  BL
   OR    Error,BL
   RET
done:
   MOV   EAX,1
   RET
_Fact   ENDP

; La procédure principale de calcul de la factorielle
; qui teste l'argument et appelle la procédure
; récursive _Fact.
FACT   PROC
   ENTER 4,0
   PUSH  EBX
   PUSH  ECX
   ; Sauve EBX et ECX, puisqu'ils sont employés dans
   ; cette routine et que nous ne voulons pas modifier
   ; l'état des registres de l'appelant.
   MOV   Error,0
   MOV   ECX,EAX
   CMP   ECX,0
   JB    Fail
   CALL  _Fact
```

Listing 4.3 : *Exemple 3a - Factorielle récursive (suite).*

```
; Fini avec le calcul de la factorielle
   CMP    Error,1
   JNZ    Done
Fail:
   MOV    EAX,-1
Done:
   POP    ECX
   POP    EBX
   LEAVE
   RET
FACT   ENDP
```

Listing 4.3 : *Exemple 3a - Factorielle récursive (fin).*

Dans cet exemple, nous appelons _Fact procédure "dummy" de Fact (en anglais : *dummy* = le mort au bridge). Le programme appelant Fact ne voit jamais _Fact. Fact facilite l'écriture du code pour la procédure récursive, _Fact. Une procédure "dummy" comme _Fact est très courante en programmation récursive.

La factorielle est une fonction mathématique puissante. En fait, il apparaît que 13! est la plus grande factorielle calculable dans une représentation du type entier signé sur 32 bits. Ainsi, il est important de surveiller tout débordement, ce que fait cette routine.

Notez le PUSH et le POP de EBX et ECX au début et à la fin du sous-programme du Listing 4.3. Cette opération fait que l'appelé ne modifie pas l'état des registres de l'appelant. En général, deux choix sont possibles :

1. L'appelant sauvegarde les registres qu'il veut garder intacts.

2. L'appelé sauvegarde tous les registres qu'il utilise.

Chacune de ces possibilités peut sauvegarder plus de registres que nécessaire, puisque l'appelant peut sauvegarder des registres que l'appelé

n'emploie pas ou l'appelé des registres dont l'appelant ne tient pas compte. Vous pouvez aussi vouloir garder intact un autre état de la machine ; l'état des indicateurs, par exemple. Les exemples de ce chapitre ne prendront pas en compte de telles considérations. Vous devez donc vous en occuper.

Souvent, la récursivité n'est pas obligatoire et vous pouvez convertir quelques programmes récursifs en simples itérations (voir l'exemple du Listing 4.4).

La factorielle itérative s'avère beaucoup plus rapide que la factorielle récursive. Le gain réel en vitesse dépend de la valeur dont on veut calculer la factorielle. Sur deux tests, nous avons noté un gain de vitesse d'environ 50 pour cent. Certains compilateurs sophistiqués de langages de haut niveau analysent les appels récursifs à des procédures et déterminent s'ils peuvent être remplacés par des itérations.

Sémaphore

L'allocation des ressources système entre les tâches est un problème classique des systèmes multitâches. Par exemple, si deux tâches veulent accéder à une unité de disque, à un dérouleur ou à un fichier, vous devez vous assurer qu'elles n'essaient pas de les utiliser simultanément. Si ce partage des ressources n'est pas garanti, le résultat peut être désastreux. Les sémaphores apportent une solution à ce problème. Ce sont des indicateurs spécialisés qu'on ne peut modifier qu'en une opération indivisible.

Imaginez qu'on utilise un octet de la mémoire comme sémaphore. Un processeur lira cet octet lorsqu'il aura besoin d'utiliser une ressource système, déterminera si cette ressource est libre, puis modifiera le sémaphore pour indiquer que la ressource n'est plus libre. Ceci marche tant que la lecture/modification/écriture de l'indicateur est une opération indivisible. Si ce n'était pas le cas, deux processeurs pourraient lire le même indicateur, et chacun pourrait tester si l'indicateur est libre et en conclure qu'il pourrait utiliser la ressource système : un désastre ! Ainsi, lorsque le sémaphore est lu, aucun autre processeur du système ne peut être autorisé à accéder à

```
; Fact: Calcule itérativement la factorielle de la valeur
; donnée dans le registre EAX. Le résultat de ce calcul
; est retourné dans EAX. Si une erreur survient
; lors du calcul, le programme retourne -1 dans EAX.
;   EAX: Nombre dont on veut calculer la factorielle.
;
Error   EQU SS:byte ptr[EBP-1]
FACT    PROC
   ENTER 4,0
   MOV   Error,0
   MOV   ECX,EAX
   CMP   ECX,0
   JB    Fail
   MOV   EAX,1
   JE    Done
top:
   IMUL  EAX,ECX
   SETO  BL
   OR    Error,BL
   LOOP  top
Done:
; Fini avec le calcul de la factorielle
   CMP   Error,1
   JZ    Fail
   LEAVE
   RET
Fail:
   MOV   EAX,-1
   LEAVE
   RET
FACT   ENDP
```

Listing 4.4 : *Exemple 3b - Factorielle itérative.*

la mémoire tant que le cycle lecture/modification/écriture n'est pas terminé. Le 80386 permet cette technique grâce au préfixe d'instruction LOCK (voir le Chapitre 3). Une instruction précédée de LOCK verrouille

la mémoire pour la durée de l'instruction. Ceci permet de garantir l'indivisibilité du cycle lecture/modification/écriture. Aucun autre processeur du système ne peut accéder à la mémoire tant que l'instruction n'est pas terminée. Dans le Listing 4.5, nous montrons l'emploi du préfixe LOCK. Ces routines fournissent un exemple de fonctions *Get* (le processeur devient le maître) et *Free* (permet à un autre processeur de devenir le maître).

L'exemple suivant montre l'utilisation des opérateurs de base GetSem et FreeSem :

```
MOV   ESI,offset SemWord
MOV   ECX,SemBitWeNeed
CALL  GetSem

< code critique demandant un usage >
< exclusif de la ressource sémaphorée >

MOV   ESI,offset SemWord
MOV   ECX,SemBitWeNeed
CALL  FreeSem
```

Cet exemple fonctionne correctement mais malheureusement de façon inefficace. Si le sémaphore n'est pas libre lors du Get, il boucle sans fin jusqu'à ce que le sémaphore le soit. On appelle cela du *busy waiting*.

Ainsi, une boucle de ce genre est coûteuse puisqu'elle ne fait rien d'intéressant si ce n'est occuper le CPU et les ressources mémoire. Souvent vous pouvez redéfinir les opérateurs sur les sémaphores (ce que nous ne ferons pas ici) pour placer ce genre de boucle dans une file d'attente (attente de la libération du sémaphore) puis appeler le *scheduler* pour exécuter une autre tâche pendant ce temps.

```
; GetSem: le bit sémaphore est égal à 0 si le sémaphore
; est occupé, ainsi le programme boucle tant que CF n'est
; pas levé lors du BTR
;   Sémaphore=0   Occupé
;   Sémaphore=1   Libre
;
;   ECX:   emplacement du bit dans le sémaphore
;   ESI:   adresse (dans DS) du sémaphore
GetSem   PROC
LoopGet:
   LOCK  BTR dword ptr[ESI],ECX
   JNC   LoopGet
   RET
GetSem   ENDP
; FreeSem: libère le sémaphore. S'il l'était déjà
; la routine renvoie -1 dans EAX.
FreeSem  PROC
   LOCK  BTS dword ptr[ESI],ECX
   JNC   Done
   MOV   EAX,-1   ; erreur: libération d'un sémaphore libre
Done:    RET
FreeSem   ENDP
```

Listing 4.5 : *Exemple 4 - Sémaphore.*

Recherche de chaîne

Etudions maintenant un programme (un autre grand classique) qui recherche une sous-chaîne à l'intérieur d'une chaîne. Les interpréteurs de commande, les éditeurs de texte, ou les bases de données disposent souvent de telles fonctions. Cette fonction est si classique qu'elle appartient à un ensemble de *benchmarks* qu'on appelle les benchmarks EDN. La plupart des fabricants d'ordinateurs utilisent ces benchmarks pour comparer les performances de leurs machines à celles de leurs concurrents. Le Lis-

ting 4.6 donne la routine codée en 80386 correspondant au benchmark EDN.

L'algorithme utilise l'instruction SCAS pour rechercher le premier octet de la sous-chaîne à l'intérieur de la chaîne. L'instruction REPNE SCASB est tout à fait efficace dans ce cas, puisqu'elle effectue la recherche par comparaison dans toute la chaîne. Lorsque le premier octet de la sous-chaîne est trouvé, la chaîne entière est comparée à la sous-chaîne rencontrée par une instruction REPE CMPSB. Si ce n'est pas la bonne chaîne (même si le premier octet est le même), la recherche recommence à partir du premier octet de la sous-chaîne et se poursuit jusqu'à ce que la sous-chaîne soit trouvée ou que la chaîne soit intégralement parcourue. Une sous-chaîne à octet unique est un cas particulier. Dans ce cas, la partie CMPS de la routine est inutile. Si la recherche aboutit, le déplacement associé au début de la sous-chaîne est retourné dans le registre EAX. Sinon, -1 est placé dans EAX.

Cette procédure est un bon exemple des instructions opérant sur les chaînes (LODS, SCAS, et CMPS). Si vous travaillez sur des chaînes d'octets, de mots, ou de dmots, vous trouverez ces instructions (LODS, CMPS, SCAS, STOS, MOVS) utiles et efficaces.

Notez aussi un moyen "rapide" pour coder ENTER :

```
PUSH  EBP       ;Code Enter 0 rapide.....
MOV   EBP,ESP
SUB   ESP,4     ;Taille du cadre local
```

L'instruction ENTER, lorsqu'elle est utilisée au niveau 0, a besoin de 10 cycles d'horloge pour son exécution. L'astuce précédente, qui fonctionne uniquement au niveau 0, ne demande que 6 cycles d'horloge. Ce gain en vitesse est possible parce que ENTER est une instruction d'usage général. Le 80386 doit d'abord tester le niveau 0. Vous - le programmeur - pouvez "sauter" ces tests comme on l'a fait ici puisque seul le niveau 0 est nécessaire.

```
; EDN Benchmark E: Recherche de Chaîne.
; Trouve une sous-chaîne dans une chaîne et retourne
; la position du début de la sous-chaîne dans EAX
; EAX vaut -1 si la sous-chaîne n'est pas trouvée
;
; Paramètres :
;   ESI:   PTR (dans DS) sur la sous-chaîne
;   EDI:   PTR (dans ES) sur la chaîne
;   ECX:   Longueur de la chaîne
;   EDX:   Longueur de la sous-chaîne
;
Srch_Len   EQU   SS: dword ptr [EBP-4]   ;StringLen-SubStrLen

BENCHE   PROC
   PUSH  EBP
   MOV   EBP,ESP
   SUB   ESP,4   ; Taille du cadre local
   CLD

; Doit rechercher le premier octet seulement pour
; StrLen-SubStrLen+1 (>0)
   LODSB          ;AL = premier octet de SubStr
   MOV   EBX,ESI ;EBX pointe sur le second octet de SubStr
   DEC   EDX     ;Longueur du reste de la sous-chaîne
   SUB   ECX,EDX ;Longueur pour SCAS
   MOV   Srch_Len,ECX   ;Sauve pour le calcul final du
                        ; déplacement

; Recherche le premier octet de la chaîne
Scan_Loop:
   JECXZ SHORT NotFnd   ;Teste si itérations précédentes
                        ; finies
   REPNE SCASB
   JNE   SHORT NotFnd   ;N'a pas rencontré le premier
                        ; caractère
```

Listing 4.6 : *Exemple 5 - Recherche de chaîne.*

```
; Premier octet rencontré. Si SubStrLen = 1, c'est fini,
; Sinon, on fait un CMPS sur le reste de la chaîne
   CMP    EDX,0
   JE     SHORT Match

; Sauve le pointeur courant (pointe sur le prochain caractère
; pour SCAS si CMPS n'a pas marché)
   PUSH   ECX
   PUSH   EDI

; Charge ESI à partir du second octet de SubStr
; (stocké dans EBX).
   MOV    ESI,EBX          ;Adr. de la source pour CMPS
   MOV    ECX,EDX          ;Longueur pour CMPS
   REPE   CMPSB            ;Vérifie si SubStr repérée
   POP    ECX
   POP    EDI    ;Récupère les pointeurs et compteurs pour SCAS
   JNE    Scan_Loop    ;Continue avec SCAS
; ECX pointe sur le second octet de SubStr
; Srch_Len contient la longueur initiale -1
; si bien que SRCH_LEN-ECX est le déplacement.
Match:
   MOV    EAX,Srch_Len
   SUB    EAX,ECX
   JMP    Epilogue
; Si sous-chaîne pas trouvée, met -1 dans EAX et rend la main
NotFnd:
   MOV    EAX,-1

Epilogue:
   LEAVE
   RET

BENCHE   ENDP
```

Listing 4.6 : *Exemple 5 - Recherche de chaîne (fin).*

Notez aussi l'emploi des directives SHORT dans le Listing 4.6. La plupart des assembleurs ne parcourent qu'une fois les instructions pour générer le code qui implémentera la fonction associée. Mais les références forward, telles que :

```
JNE    SHORT NotFnd
```

sont des sauts à des étiquettes (ici NotFnd) que l'assembleur n'a pas encore rencontrées, si bien que ce dernier ne peut pas déterminer si l'étiquette est réellement courte (+127 à -128 octets à partir du JNE). Ainsi, un assembleur à simple passe insérera un saut à déplacement sur 32 bits plutôt que de tenter un saut court. La directive SHORT permet au programmeur de forcer l'assembleur à générer du code pour un saut court.

Transfert de blocs binaires

Les fonctions graphiques, comme bien d'autres, traitent souvent des données constituées de simples séquences binaires. Une des opérations les plus classiques consiste à transférer une chaîne (ou bloc) binaire depuis un emplacement mémoire vers un autre. La routine du Listing 4.7 effectue un transfert binaire depuis une chaîne d'origine pointée par ESI vers une chaîne de destination pointée par EDI. Cette routine exige que la destination soit alignée sur une frontière de dmot, et que la longueur de la chaîne binaire soit exactement un multiple de 32 bits. Ceci simplifie l'exemple qui illustre l'essentiel du fonctionnement de BitBlt. La chaîne d'origine peut être située à n'importe quel déplacement arbitraire à l'intérieur de la chaîne d'origine. La chaîne binaire est traitée à partir des adresses hautes vers les adresses basses. Le double décalage à droite est plus efficace de cette façon.

```
; BitBlt: transfère arbitrairement la chaîne binaire
; source pointée par ESI vers la chaîne destination
; (alignée) pointée par EDI. La quantité qui est transférée
; doit être un multiple de 32 bits.
;   ESI: déplacement de la source (DS)
;   EDI: déplacement de la destination (ES)
;   EBX: compteur dmot des bits transférés
;        (multiples de 32 bits donc)
;   ECX: déplacement à partir de 0 dans la chaîne.
BitBlt   PROC
   MOV   EDX,dword ptr [ESI]
   ADD   ESI,4
   CLD
BitLoop:
   LODS                ;nouvelle partie de poids fort
   SHRD  EDX,EAX,CL    ;EDX reçoit le dmot aligné
   XCHG  EAX,EDX       ;permute poids fort et poids faible
   STOS
   DEC   EBX
   JA    BitLoop
   RET
BitBlt   ENDP
```

Listing 4.7 : *Exemple 6a - Transfert de bloc binaire simplifié.*

La routine du Listing 4.7 montre comment utiliser l'instruction de double décalage à droite du 80386, ainsi que son efficacité. Dans cet exemple, la boucle peut traiter 32 bits à chaque itération. Si le 80386 ne possédait pas d'instruction de double décalage, on pourrait au mieux traiter 16 bits à chaque itération en employant un décalage sur 32 bits à la place du double décalage. Ainsi, le double décalage permet de doubler la performance de ce programme.

Cet exemple démontre donc comment l'instruction de double décalage du 80386 facilite le transfert. Malheureusement, cette procédure simplifiée n'est pas très utile, puisque habituellement les chaînes binaires n'ont pas

une longueur multiple de 32 bits, et que les chaînes de destination ne sont pas toujours alignées sur une frontière de dmot. La procédure du Listing 4.8 est une routine de transfert de bloc binaire plus générale et plus utile. Celle-ci travaille sur des chaînes binaires de longueur quelconque. Les chaînes d'origine et de destination peuvent être alignées sur n'importe quel déplacement arbitraire. Malheureusement, cette procédure plus générale est beaucoup plus compliquée !

En effet, cette procédure est composée de trois parties : le prélude, la partie principale, et le postlude. Le prélude aligne la destination sur une frontière de dmot. C'était aussi une contrainte pour la procédure de transfert simplifiée. Cet alignement rend nécessaire le transfert des premiers bits de la source vers la destination tout en gardant intact chaque bit de la chaîne de destination situé avant le déplacement initial de la chaîne de destination. Le prélude, aussi bien que le postlude, suit des algorithmes différents selon que le déplacement de la destination est plus grand ou non que celui de l'origine. Ceci est nécessaire puisque le prélude "consomme" tous les bits du premier dmot de la chaîne d'origine si le déplacement de la destination est plus grand que le déplacement de l'origine.

Après le prélude, on entre dans la boucle principale. Celle-ci calcule tout d'abord le nombre de décalages, qui est égal à la différence de déplacement entre l'origine et la destination. Elle est identique à la routine du Listing 4.7. Les deux contraintes de la routine simplifiée sont les mêmes que celles de la boucle principale. Le prélude et le postlude prennent en compte l'alignement de la destination sur une frontière dmot pour pouvoir transférer des blocs non multiples de 32 bits. Chaque itération de la boucle principale traite 32 bits de données.

```
; BitBlt général : transfère une chaîne binaire de longueur
; quelconque vers n'importe quelle destination.
;   EAX: (compteur binaire) le nombre de bits à transférer.
;   ESI: adresse dmot de la chaîne d'origine.
```

Listing 4.8 : *Exemple 6b - Transfert de bloc binaire.*

```
;   EDI: adresse dmot de la chaîne de destination.
;   ECX: (SrcOff) déplacement binaire
;        dans le dmot (Source) déterminé par ESI.
;   EDX: (DestOff) déplacement binaire
;        dans le dmot (Destination) déterminé par EDI.
;
BitCount    EQU SS:dword ptr [EBP-4]
SrcOff      EQU SS:dword ptr [EBP-8]
DestOff     EQU SS:dword ptr [EBP-12]
SrcOffFlg   EQU SS:dword ptr [EBP-16]

BitBlt   PROC
   ENTER 16,0     ;16 octets de variables locales temporaires
   CLD

;****
;**PRELUDE:            ;Partie initiale de BitBlt
;****
   MOV   BitCount,EAX  ;Sauvegarde dans la pile
                       ;les paramètres
   MOV   SrcOff,ECX    ;passés par les registres.
   MOV   DestOff,EDX
   MOV   EDX,dword ptr [ESI]   ;charge les bits poids fort et
   MOV   EAX,dword ptr [ESI+4] ;poids faible de la
                               ;chaîne d'origine.
;Détermine le premier dmot de la destination
   SHRD  EDX,EAX,CL   ;EDX contient les 32 premiers bits
                      ;de l'origine.
   MOV   EAX,dword ptr [EDI]   ;prend les bits restants
   MOV   ECX,DestOff           ;et les transfère dans le dmot
   ROR   EAX,CL                ;de la destination
   SHLD  EDX,EAX,CL
   MOV   EBX,32            ;teste le cas où la chaîne entière
   SUB   EBX,DestOff       ;est contenue dans un dmot unique
   SUB   EBX,BitCount      ;Si elle est contenue dans le
```

Listing 4.8 : *Exemple 6b - Transfert de bloc binaire (suite).*

```
                                ;premier dmot
   JBE    DestMoreThan1         ;de la destination, on saute à la
                                ;fin de
   MOV    ECX,32                ;la routine qui inclura tous les
                                ;bits restants
   SUB    ECX,EBX               ;de poids fort.
   MOV    EBX,ECX
   JMP    CommonEnd
DestMoreThan1:
   MOV    dword ptr [EDI],EDX   ;range le premier résultat
   ADD    EDI,4
   JZ     Done          ;rentre exactement dans le premier dmot
                        ;de  la destination.

;Calcule le nombre correct de décalages
   SUB    ECX,SrcOff;Si SrcOff>DestOff alors CL=SrcOff-DestOff
   JL     SrcLrgStr ;Sinon              CL=32-(DestOff-SrcOff)
   MOV    BL,32
   SUB    BL,CL
   MOV    CL,BL
   JMP    BeginBitBlt
SrcLrgStr:
   MOV    ECX,SrcOff
   SUB    ECX,DestOff
   ADD    ESI,4  ;Puisque DestOff <= Srcoff le programme a
                 ;traité tous les premiers dmots de l'origine.
   MOV    SrcOffFlg,1  ;Simple indicateur notant que
                       ;SrcOff > DestOff
;****
;** Programme Principal : Corps de la boucle BitLoop
;****
;
; Corps de BitBlt. A ce stade, Dest est aligné sur une
; frontière 32 bits. CL contient le nombre de décalages,
; ESI pointe sur l'origine, et EDI sur la destination
```

Listing 4.8 : *Exemple 6b - Transfert de bloc binaire (suite).*

```
; à l'endroit correct pour commencer le transfert
BeginBitBlt:
   MOV   EBX,BitCount
   ADD   EBX,DestOff
   SUB   EBX,32      ;EBX contient le nb de bits à transférer.
   MOV   EDX,dword ptr [ESI]
   ADD   ESI,4
; La boucle de BitBlt. Transfère 32 bits depuis l'origine
; vers la destination à chaque itération. La boucle
; s'arrête quand il reste moins de 32 bits à transférer.
BitLoop:
   SUB   EBX,32
   JLE   FixEnd
   LODSD
   SHRD  EDX,EAX,CL
   XCHG  EDX,EAX        ;EAX: partie alignée de Dest,
                        ;EDX: partie restante de l'origine
                        ;pour la prochaine itération
   STOSD                ;Range EAX dans la chaîne Dest
   JMP   BitLoop

;****
;**POSTLUDE: Partie finale de BitBlt
;****
FixEnd:
   JZ    Done           ; Si pas de bits à traiter, c'est fini !
   CMP   SrcOffFlg,1
   JZ    SrcLrgEnd
   LODSD
   SHRD  EDX,EAX,CL
   JMP   CommonEnd
SrcLrgEnd:
   SHR   EDX,CL
```

Listing 4.8 : *Exemple 6b - Transfert de bloc binaire (suite).*

```
CommonEnd:          ; A ce stade, EDX contient les bits restants
                    ; de la chaîne d'origine alignés sur la
                    ; destination
   MOV    EAX,dword ptr [EDI]
   MOV    CL,BL
   ROR    EDX,CL
   ROR    EAX,CL
   MOV    BL,32
   SUB    BL,CL
   MOV    CL,BL
   SHRD   EDX,EAX,CL     ;Effectue le décalage final
                         ;en plaçant la destination
                         ;après la fin de la chaîne binaire
   XCHG   EAX,EDX
   STOSD                 ;Range le dmot final dans la destination
Done:
   LEAVE
   RET
BitBlt    ENDP
```

Listing 4.8 : *Exemple 6b - Transfert de bloc binaire (fin).*

La partie postlude prend en compte le transfert des derniers bits dans la chaîne de destination, tout en gardant intacts tous les bits situés après la chaîne binaire de destination. Le programme doit considérer le cas particulier (dont le prélude s'occupe) d'une chaîne binaire limitée à un dmot unique. Le nombre de bits transférés vers la destination par le postlude est donné par :

```
(BitCount - DestOffset) % 32
```

Même si la routine de transfert binaire générale du Listing 4.8 est beaucoup plus compliquée que celle du Listing 4.7, décomposer la routine en

trois parties (prélude, partie principale et postlude) et analyser chacune de ces parties séparément facilite la compréhension de sa fonction.

Comme c'est souvent le cas, cette procédure utilise le stockage local en pile. Ceci permet la récursivité et en général facilite la gestion des variables temporaires locales, puisqu'on ne stocke que les variables qu'on emploie. Comme on l'a mentionné dans le Chapitre 2, on emploie souvent le registre EBP comme pointeur de cadre de pile, et ESP comme pointeur dynamique du sommet de la pile. Ainsi, EBP adresse souvent les variables locales temporaires :

```
BitCount   EQU   SS: dword ptr [EBP-4]
```

Ainsi, vous pouvez employer un mnémonique BitCount qui équivaut à une variable temporaire (basée sur la pile) référencée indirectement par EBP.

Les instructions ENTER et LEAVE permettent d'utiliser EBP comme pointeur de cadre de pile. L'instruction ENTER 16,0 crée dans notre exemple un cadre de pile de niveau 0 avec 16 octets disponibles pour des variables locales. Un cadre de pile de niveau 0 indique qu'aucune image n'est construite dans le nouveau cadre de pile. Un modèle de programmation avec des procédures imbriquées et des variables libres dans ces procédures imbriquées (c'est le cas de langages évolués tels que le Pascal) utilise souvent une image. Le complément d'un ENTER est une instruction LEAVE qui détruit le cadre de pile juste avant de rendre la main au programme appelant.

EXEMPLES D'INSTRUCTIONS A VIRGULE FLOTTANTE

Dans ce paragraphe, nous allons donner plusieurs exemples d'utilisation des instructions à virgule flottante.

Indicateurs à virgule flottante

Nous avons parlé des codes de condition à virgule flottante dans le Chapitre 2. Ces quatre codes de condition font partie du mot d'état du 80387. Dans le Chapitre 3, lorsque nous avons parlé des instructions à virgule flottante du 80387, nous avons aussi présenté plusieurs instructions qui modifient ces codes de condition. Par exemple, l'instruction :

```
FCOM    op1
```

compare numériquement l'opérande donné (op_1) avec le sommet de la pile, puis positionne les codes de condition comme le montre le tableau suivant :

	C3	C2	C0
ST > op_1	0	0	0
ST < op_1	0	0	1
ST = op_1	1	0	0
Non classé	1	1	1

Voyons maintenant comment employer ces codes de condition pour modifier le cours de l'exécution d'un programme. Il existe une instruction particulière des 80386/80387 : FSTSW AX. Cette instruction copie le mot d'état du 80387 dans le registre AX du 80386. Si l'instruction FSTSW est

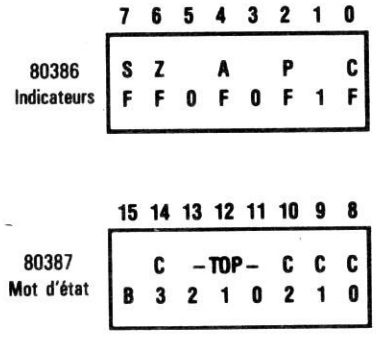


Figure 4.1 : *Transfert des indicateurs du 80386 vers le 80387.*

```
; Comparaison et table de branchement
FCOM  ST(1)    ; Effectue la comparaison
FSTSW AX       ; Transfère le mot d'état dans le registre
               ;AX du 80386
SAHF           ; Transfère les codes de condition
               ; dans le registre des indicateurs
JP    Error    ;ST ou ST(1) est un NaN - Condition d'erreur
JE    STeqST1  ;ST = ST(1)
JB    STltST1  ;ST < ST(1)
JA    STgtST1  ;ST > ST(1)
```

Listing 4.9 : *Exemple 7 - Indicateurs de virgule flottante.*

suivie de l'instruction SAHF (range AH dans les indicateurs), on peut utiliser les sauts conditionnels du 80386 sur les indicateurs du 80387. Si FSTSW AX est suivie de SAHF, le diagramme de la Figure 4.1 montre la partie haute du mot d'état et les indicateurs associés du 80386.

La séquence de code du Listing 4.9 montre une table de branchement basée sur l'état des codes de condition du 80387 après une instruction FCOM. Le premier accumulateur est comparé avec le sommet de la pile et le programme effectue le branchement approprié selon le résultat de la comparaison.

On peut employer un exemple similaire pour les instructions FTST, FICOM, FUCOM et toutes les variantes de ces instructions.

Reste partiel

Dans la présentation des instructions qui calculent le reste partiel (Chapitre 3) (FPREM, FPREM1), nous avions mis en évidence leur lenteur. Elles pouvaient donc être interrompues au cours de leur réduction. La routine du Listing 4.10 est constituée d'une boucle courte qui teste si la réduction est terminée et réitère l'opération si nécessaire.

```
;Calcul complet du reste partiel
RemLoop:
   FPREM
   FSTSW AX
   SAHF
   JP    RemLoop    ; Si C2=1 la réduction est incomplète,
                    ; Réessaye.
```

Listing 4.10 : *Exemple 8 - Reste partiel.*

Dans cet exemple, notez l'emploi de la séquence d'instructions de transfert des indicateurs dont on a parlé auparavant.

Calculs exponentiels

Rappelez-vous que l'instruction F2XM1 calcule :

$$ST = 2^{ST} - 1$$

```
; Calcul exponentiel autre que 2^x-1

; 10^x                  ; ST = X
   FLDL2T
   FMULP ST,ST(1)       ; ST = X * log2 10
   F2XM1                ; ST = 2^x * log2 10 - 1
   FLD1
   FADDP ST,ST(1)       ; ST = 2^x * log2 10 = 10^x

; e^x                  ; ST = X
   FLDL2E
   FMULP ST,ST(1)       ; ST = X * log2 e
   F2XM1                ; ST = 2^x * log2 e - 1
   FLD1
   FADDP ST,ST(1)       ; ST = 2^x * log2 e = e^x

; Y**X                  ; ST = Y, ST(1) = X
   FYL2X                ; ST= X * log2 Y
   F2XM1                ; ST = Y^X-1
   FLD1
   FADDP ST,ST(1)       ; ST = Y^X
```

Listing 4.11 : *Exemple 9 - Calculs exponentiels.*

Notez que l'instruction F10XM1 n'existe pas. Les trois exemples du Listing 4.11 fournissent des algorithmes de calcul exponentiel avec F2XM1. Les équations liées à ces exemples sont :

$$10^X = 2^x * \log_2 10$$
$$e^X = 2^x * \log_2 e$$
$$Y^X = 2^x * \log_2 Y$$

Notez l'emploi bien agréable de l'instruction FYL2X dans le dernier exemple. Rappelez-vous que FYL2X implémente la fonction :

$$ST = ST(1) * \log_2 ST \quad ; pop ;$$

Multiplication de matrices

On arrange souvent des données mathématiques sous forme de matrices. Un traitement classique sur des matrices consiste à en multiplier deux pour en former une troisième. La multiplication de matrices prend deux matrices, X et Y, où X est de dimension i*k et Y de dimension k*j. Le résultat, la matrice Z, est de dimension i*j. Les dimensions k de X et Y doivent être identiques. Les éléments de la matrice Z obéissent à la relation :

$$Z_{ij} = \sum_{n=1}^{k} (X_{in} * Y_{nj})$$

pour tous les éléments de Z.

La routine du Listing 4.12 effectue une multiplication matricielle. Les adresses des deux matrices opérandes sont passées par les registres ESI et EDI. L'adresse des résultats est passée par le registre EDX. Toutes les trois sont dans le segment adressé par le registre DS. Les trois matrices sont stockées par rangées.

```
; Cette routine effectue une multiplication de
; matrices en prenant deux matrices X et Y pour
; donner une matrice Z. La matrice X est de dimension
; i*k, la matrice Y de dimension k*j et le résultat
; de dimension i*j. Notez que les matrices sont
; stockées par rangées.
;   ESI: Déplacement de X dans le segment DS
;   EDO: Déplacement de Y dans le segment DS
;   EAX: Di, i dimension
;   ECX: Dk, k dimension
;   EBX: Dj, j dimension
;   EDX: Déplacement de Z dans le segment DS
;
; Les boucles sont :
; for (iCnt = 0; iCnt < Di; iCnt ++){
;    for (jCnt = 0; jCnt < Dj; jCnt ++){
;       for (ECX = Dk; ECX   0; ECX--){
;          charge élément depuis X
;          charge élément depuis Y
;          multiplication; addition à la somme partielle
;       }
;       range dans Z
;    }
; }
Di   EQU   SS: dword ptr [EBP-4]
Dk   EQU   SS: dword ptr [EBP-8]
Dj   EQU   SS: dword ptr [EBP-12]
Zoff EQU   SS: dword ptr [EBP-16]
iCnt EQU   SS: dword ptr [EBP-20]
jCnt EQU   SS: dword ptr [EBP-24]

MatMult   PROC
   ENTER 24,0       ; Pas d'image copiée, 24 octets
                    ; de variables locales
```

Listing 4.12 : *Exemple 10 - Multiplication de matrices.*

```
MOV   Di,EAX     ; Sauvegarde le contenu des
                 ; registres avant
                 ; utilisation
MOV   Dk,ECX     ;  "
MOV   Dj,EBX     ;  "
MOV   Zoff,EDX   ;  "
CMP   EAX,0      ; Test si dimension = 0
JZ    Done       ;  "
CMP   ECX,0      ;  "
JZ    Done       ;  "
CMP   EBX,0      ;  "
JZ    Done       ;  "
MOV   iCnt,0     ; i est la variable de boucle
MOV   EDX,0      ; z est le pointeur d'élément
Loop_i:
   MOV   jCnt,0   ; j est la variable de boucle
   Loop_j:
      MOV   EAX,iCnt
      IMUL  EAX,Dk
      MOV   EBX,jCnt
      FLDZ
      MOV   ECX,Xk
      Loop_k:
         DEC   ECX
         FLD   qword ptr [ESI+EAX*8]   ; traite l'élt de X
         INC   EAX
         FLD   qword ptr [EDI+EBX*8]   ; traite l'élt de Y
         ADD   EBX,Dj
         FMULP ST(1),ST
         FADDP ST(1),ST
         CMP   ECX,0
         JA    Loop_k
      MOV   EAX,Zoff
      FSTP  qword ptr [EAX+EDX*8]   ; Range dans Z
      INC   EDX                     ;Passe à l'élt suivant de Z
```

Listing 4.12 : *Exemple 10 - Multiplication de matrices (suite).*

```
        INC     jCnt
        MOV     EAX,Dj
        CMP     EAX,jCnt
        JA      Loop_j
    INC     iCnt
    MOV     EAX,Di
    CMP     EAX,iCnt
    JA      Loop_i
Done:
   LEAVE
   RET
MatMult    ENDP
```

Listing 4.12 : *Exemple 10 - Multiplication de matrices (fin).*

Ce programme met en évidence le problème de la représentation d'un tableau à deux dimensions dans une mémoire à une dimension (rappelez-vous la discussion du Chapitre 1). Les matrices sont stockées par rangées. Si vous avez un tableau A avec des éléments A_{ij}, $A_{0,0}$ est-il suivi de $A_{1,0}$ ou de $A_{0,1}$? Dans le format de stockage par rangées, les éléments d'une même rangée sont stockés séquentiellement en mémoire. Notez que dans le format en rangées :

$A_{0,0}, A_{0,1}, A_{0,2},...A_{0,n}$ (rangée 0)

sont stockés séquentiellement et suivis par :

$A_{1,0}, A_{1,1}, A_{1,2},...A_{1,n}$ (rangée 1)

etc.

Les dimensions *i* et *k* de la matrice X sont placées dans les registres EAX et ECX, et les dimensions *k* et *j* dans les registres ECX et EBX. Les dimensions *k* des matrices X et Y doivent être identiques pour pouvoir multiplier les matrices.

Notez que cette routine peut multiplier des matrices de toutes tailles et qu'elle ne connaît pas leurs tailles avant d'être appelée. On peut coder de façon plus efficace un programme qui effectue des multiplications de matrices de taille fixe. Chaque élément de X et Y est chargé *k* fois. Une routine qui connaîtrait la taille des matrices pourrait charger les éléments d'une matrice (ceux de X par exemple) une fois pour toutes, et utiliser l'élément à partir de la pile accumulateur plutôt que de le charger *k* fois.

La procédure emploie trois boucles imbriquées. La première se répète *i* fois, la seconde *j* fois, et la troisième *k* fois. La boucle interne calcule les éléments de Z_{ij}. La seconde parcourt les rangées de X et remplit ainsi les rangées de Z à chaque colonne de Y. La troisième se répète pour chaque rangée de X et de Z. Ainsi, le programme remplit séquentiellement en mémoire la matrice Z.

Statistiques

On doit souvent effectuer des statistiques sur des recueils de données. Par exemple, un instructeur peut donner la moyenne et l'écart type d'un test lorsqu'il distribue les notes d'une classe. La moyenne statistique donne la moyenne arithmétique des notes de la classe. L'équation de la moyenne est :

$$\mu = \frac{\sum_{i=0}^{n-1} x_i}{n}$$

où Σ représente l'opérateur de sommation qui calcule la somme de tous les x_i pour *i* variant de 0 à *n*-1. Nous emploierons la notation : Σx.

L'écart type fournit une mesure de la dispersion moyenne autour de la moyenne. L'équation de l'écart type est :

$$s = \sqrt{\frac{\Sigma(x-\mu)^2}{n-1}}$$

Si vous voulez calculer l'écart type d'un ensemble de nombres, cette équation vous obligera à calculer d'abord la moyenne des nombres, puis vous effectuerez le calcul de l'écart type. Ceci se ferait en deux étapes. Il existe une meilleure équation qui permet d'effectuer le calcul en un seul passage (on ne démontre pas la formule ici). On a donc une formule équivalente :

$$s = \sqrt{\frac{\sum x^2 - \frac{(\Sigma x)^2}{n}}{n-1}}$$

La fonction du Listing 4.13 utilise cette seconde équation pour calculer l'écart type. La boucle principale calcule Σx^2 et Σx et place les résultats dans ST(1) et ST.

```
; Stat : la procédure suivante calcule l'écart-type et
; la moyenne d'une liste de nombres. Cette dernière
; est une liste de réels longs en mémoire pointés
; par ESI. ECX donne la longueur de la liste.
; L'écart-type est calculé et retourné au sommet de la pile
; De même pour la moyenne dans ST(1).
; L'équation suivante donne l'écart-type :
;   s = sqrt((sum(x²) - sum(x)²/n)/(n-1))
; L'équation suivante donne la moyenne :
;   u = sum(x)/n
;   ECX: nombre de réels longs
;   ESI: pointeur (dans DS) sur la liste des réels longs
;
Count   EQU   SS: dword ptr [EBP-4]
STAT    PROC
   ENTER  4,0
   MOV    Count,ECX       ;Sauvegarde le compteur
```

Listing 4.13 : *Exemple 11 - Statistiques.*

```
   FLDZ                    ;sum(x) nulle
   FLDZ                    ;sum(x²) nulle
; La boucle suivante calcule sum(x), le place dans ST(1),
; et sum(x²) et le place dans ST.
LoopTop:
   FLD     qword ptr [ESI+ECX*8]
   FADD    ST(2),ST        ;additionne à sum(x)
   FMUL    ST,ST(0)        ;crée x²
   FADDP   ST(1),ST        ;additionne à sum(x²)
   LOOPNZ  LoopTop

; A ce stade ST=sum(x²) et ST(1)=sum(x)
   FLD     ST(1)
   FMUL    ST,ST(1)        ; calcule sum(x)²
   FILD    Count           ; Après : ST=n
   FDIV    ST(1),ST        ; Après: ST(1)=sum(x)²/n
   FXCH    ST,ST(1)
   FSUBP   ST(2),ST        ; Après: ST(1)=sum(x²)-sum(x)²/n
   FLD1
   FSUBR   ST,ST(1)        ; Après : ST=n-1
   FDIVP   ST(2),ST        ; Après : ST(1)=(sum(x²)-sum(x)²/n)/(n-1)
   FDIVP   ST(2),ST        ; ST(1)=u=sum(x)/n
   FSQRT                   ; Après : ST=s
; ST=s, ST(1)=u
   LEAVE
   RET
STAT   ENDP
```

Listing 4.13 : *Exemple 11 - Statistiques (fin).*

La partie finale de la procédure permet l'emploi de la pile accumulateur à virgule flottante. En fait, il n'est pas facile de suivre ce qui se passe dans la pile à chaque instruction. Le Tableau 4.2 donne une trace horizontale de la pile, qui pourra vous aider à en comprendre le fonctionnement exact. A droite, on a le contenu de la pile accumulateur à virgule flottante après

l'opération de gauche sur la même ligne. Pour des raisons de concision, on utilise l'identité suivante dans le tableau :

$$a = \sum x^2 - \frac{(\sum x)^2}{n}$$

		PILE APRES EXECUTION				
Instruction		**ST**	**ST(1)**	**ST(2)**	**ST(3)**	**ST(4)**
		$\sum x^2$	$\sum x$			
FLD	ST(1)	$\sum x$	$\sum x^2$	$\sum x$		
FMUL	ST,ST(0)	$(\sum x)^2$	$\sum x^2$	$\sum x$		
FILD	Count	n	$(\sum x)^2$	$\sum x^2$	$\sum x$	
FDIV	ST(1),ST	n	$(\sum x)^2/n$	$\sum x^2$	$\sum x$	
FXCH	ST,ST(1)	$(\sum x)^2/n$	n	$\sum x^2$	$\sum x$	
FSUBP	ST(2),ST	n	a	$\sum x$		
FLD1	1	n	a	$\sum x$		
FSUBR	ST,ST(1)	n-1	n	a	$\sum x$	
FDIVP	ST(2),ST	n	s^2	$\sum x$		
FDIVP	ST(2),ST	s^2	μ			
FSQRT		s	μ			

Table 4.2 : *Trace horizontale de la pile.*

5

Gestion mémoire, protection et tâches

Dans ce chapitre et le Chapitre 6, nous allons décrire les fonctionnalités du 80386 qui facilitent la mise en oeuvre des systèmes d'exploitation. Ces chapitres sont moins pédagogiques que ceux qui décrivaient les fonctionnalités que le 80386 mettait à la disposition des programmeurs d'applications. Dans ce chapitre, nous considérerons que les concepts de base des systèmes d'exploitation multitâches qui utilisent la mémoire virtuelle vous sont familiers. Nous allons rappeler dans une brève introduction quelques concepts pertinents, et vous indiquer notre terminologie. Ce chapitre s'adresse directement aux développeurs de systèmes d'exploitation. Cependant, les programmeurs d'applications pourront trouver dans la première partie un résumé des concepts de gestion mémoire, de protection et de gestion multitâche, qui ne sont pas directement liés au 80386. Ceux d'entre eux qui voudraient employer le modèle à segments multiples (voir le Chapitre 2) seront tout particulièrement concernés.

Les systèmes d'exploitation, même ceux des ordinateurs personnels mono-utilisateurs, doivent pouvoir travailler sur plusieurs tâches à la fois afin d'optimiser les possibilités du matériel ainsi que le temps CPU. Un ordinateur à base de 80386 possède une puissance de calcul suffisante pour effectuer plusieurs tâches en arrière-plan, tout en vous permettant d'exécuter pendant ce temps votre traitement de texte ou un programme de jeu. Pendant que vous utilisez le traitement de texte, vous pouvez avoir besoin d'appeler le programme de feuille de calcul afin de traiter certaines données pour votre rapport. Vous pouvez alors suspendre le traitement de texte dans une fenêtre en arrière-plan pendant que vous exécutez le programme de feuille de calcul. Dès que le traitement est effectué, vous pouvez transférer les données de votre feuille de calcul dans votre traitement de texte, puis continuer à éditer votre rapport. Vous pouvez avoir besoin d'effectuer plusieurs aller et retour entre ces deux programmes.

Chaque programme constitue une *tâche* distincte, que ce soit une mise à jour de la feuille de calcul, un chargement d'une base de données à partir d'un ordinateur central, un traitement de texte ou un jeu demandant une interaction continue. Chaque tâche est "une partie séquentielle de l'exécution" d'un programme unique. A l'intérieur d'une tâche, le programme s'exécute séquentiellement. Cependant, les tâches elles-mêmes se partagent le temps CPU de telle façon qu'elles semblent s'exécuter simultanément tout en restant asynchrones. L'exécution d'une tâche peut être

suspendue en attendant qu'un accès à une unité de disque soit terminé ; ainsi, une autre tâche peut utiliser le processeur. Ensuite, l'apparition d'une interruption nous indiquera la fin de l'accès à l'unité de disque. La tâche en cours d'exécution est alors suspendue afin que l'interruption soit traitée, ce qui peut signifier qu'une autre tâche doit poursuivre son exécution. Ou alors, vous pouvez contrôler la commutation des tâches, comme lorsque vous passez d'une fenêtre à une autre pour suspendre une application et en poursuivre une autre.

Le système d'exploitation utilise les possibilités décrites dans ce chapitre et dans le Chapitre 6 pour allouer et protéger les ressources du système : espace mémoire principal, temps d'exécution, et périphériques. Il alloue les ressources aux tâches du système afin qu'elles puissent s'exécuter efficacement. Les règles d'allocation des ressources et de leur protection profitent des mécanismes de gestion mémoire et de protection du 80386. Le système d'exploitation emploie ces mécanismes pour assurer la protection mutuelle des tâches. Par exemple, il utilise habituellement les mécanismes de gestion mémoire pour vérifier que les zones mémoire allouées à des tâches différentes ne se recouvrent pas (à moins d'un partage voulu de la mémoire). Il emploie en outre les mécanismes de protection pour s'assurer qu'aucune tâche ne peut accéder à la mémoire allouée au système d'exploitation.

La plupart des possibilités décrites dans ce chapitre et dans le Chapitre 6 sont employées pour allouer et protéger des ressources. Elles peuvent être décomposées en trois groupes qui sont associés aux trois catégories de ressources suivantes : espace mémoire principal, temps CPU, et périphériques. Les possibilités de gestion mémoire permettent l'allocation et la protection de l'espace mémoire. Associées au mécanisme d'interruption et d'exception (voir le Chapitre 6), elles permettent l'allocation du temps CPU. La gestion des E/S autorise l'allocation et la protection des périphériques. Dans le reste de ce chapitre, nous allons traiter de la gestion mémoire et de la gestion des E/S.

Les exceptions présentent des caractéristiques intéressantes, mais on ne les décrira de façon précise que dans le Chapitre 6. Dans ce chapitre, nous présenterons cependant quelques aspects des exceptions afin de faciliter la description du reste du système d'exploitation. Une exception est en

quelque sorte une instruction CALL invisible qui est exécutée chaque fois qu'une opération non valide est décelée. CALL appelle une procédure prédéfinie du système d'exploitation écrite spécialement pour traiter les exceptions. Cette procédure peut soit décider de mettre fin au programme ayant généré l'exception, soit enregistrer la cause de cette exception et laisser le programme se poursuivre. On emploie les exceptions pour communiquer au système d'exploitation les éventuelles opérations non valides. Comme vous le verrez, les exceptions sont un bon moyen de rendre le contrôle des ressources au système d'exploitation. Si un programme tente d'utiliser ou d'accéder à une ressource qui n'est pas la sienne, une exception sera générée pour faire part de cette tentative au système d'exploitation.

GESTION MEMOIRE

Tout système de gestion mémoire se décompose en deux parties : la protection et la translation d'adresses. La protection permet d'empêcher qu'une tâche accède à de la mémoire qui appartient à une autre tâche ou au système d'exploitation. La translation d'adresses apporte au système d'exploitation une certaine flexibilité pour l'allocation mémoire des tâches et pour le mécanisme de protection.

Translation d'adresses

Comme nous l'avons défini dans le Chapitre 1, la mémoire physique d'un ordinateur est un tableau linéaire d'octets, chaque octet ayant une adresse unique connue comme son *adresse physique*. Dans le Chapitre 2, nous avons défini les adresses employées par les programmes pour accéder à la mémoire. Ces adresses ont deux composantes pour adresser la mémoire segmentée à deux dimensions employée dans le 80386. Cette adresse n'est pas directement employée pour accéder à la mémoire physique. Au contraire, les adresses mémoire sont transformées en adresses mémoire physiques par le mécanisme de pagination des adresses. Ce der-

nier introduit la notion d'*adresse virtuelle*, qui est une adresse à deux composantes générées par un programme. On emploie le terme d'*adresse virtuelle* car ces adresses ne correspondent pas directement aux emplacements physiques de la mémoire, mais seulement indirectement via la fonction transformant une adresse virtuelle en adresse physique (*mapping*).

Notez que chaque adresse générée par un programme est une adresse virtuelle, qu'on se trouve au coeur du système d'exploitation ou dans un programme d'application. Quel que soit le type de programme, le mécanisme de gestion mémoire traduit toujours une adresse virtuelle en adresse physique.

Cette opération est intéressante pour la protection de la mémoire, puisque vous pouvez faire en sorte que certaines adresses physiques ne soient pas transformées en adresses virtuelles. En plus de la protection de base, cette fonction étend la protection de la mémoire puisqu'elle peut rendre une adresse virtuelle non valide. Plutôt que de générer une adresse physique à partir d'une adresse virtuelle non valide, elle génère une exception qui permet au système d'exploitation de prendre les mesures nécessaires.

Segmentation et pagination

Pour réduire la quantité d'informations que demande la gestion de la mémoire, on transformera des blocs séquentiels importants en unités distinctes. Cela permet de paginer en une fois un bloc complet de mémoire plutôt que de le faire pour chaque octet. La segmentation et la pagination sont deux techniques largement employées. Dans le cas de la pagination, la mémoire virtuelle est organisée en blocs. On peut noter certaines différences entre ces deux techniques pour transformer l'information. De plus, les programmeurs voient leurs résultats de façon différente.

La segmentation et la pagination emploient toutes deux des tables qui résident en mémoire. Seul le système d'exploitation peut accéder à ces tables, afin d'éviter toute modification de leur contenu par des programmes d'application, ce qui changerait tout l'espace mémoire virtuel. Normale-

ment, le système d'exploitation gère un jeu de tables différent pour chaque tâche. Ceci permet à chaque tâche d'utiliser les adresses de tout l'espace mémoire virtuel. Ainsi, chaque tâche a un espace mémoire virtuel différent, et les tâches sont isolées les unes des autres.

Le 80386 utilise à la fois la segmentation et la pagination dans son mécanisme de translation des adresses. Ce dernier est composé de deux étapes. La première emploie la segmentation pour transformer une adresse à deux composantes dans l'espace adressable virtuel en une adresse dans un espace adressable intermédiaire, appelé l'*espace adressable linéaire*. La seconde étape utilise la pagination pour transformer cette adresse linéaire en adresse physique. La segmentation est obligatoire, mais on peut activer ou non la pagination. Si la pagination n'est pas activée, le processeur utilise directement les adresses linéaires produites par la segmentation comme adresses physiques. Tout se passe comme si la fonction de pagination était la fonction identité (voir la Figure 5.1). Donc, s'il n'y a pas de pagination, l'adresse physique est égale à l'adresse linéaire. Sinon, elle est transformée par la table de pagination.

L'espace adressable linéaire a la même structure que l'espace adressable physique. Tous les deux sont des espaces mémoire à une dimension, tandis que l'espace mémoire virtuel est à deux dimensions. Ce dernier contient jusqu'à 16 Ko de segments, chacun pouvant avoir une taille égale à 4 Go, donnant ainsi un espace adressable virtuel de 64 To (tera octets) (2^{46}). Les espaces mémoire linéaire et physique ont une taille maximale de 4 Go (2^{32}). En fait, s'il n'y a pas de pagination, l'espace adressable linéaire *est* égal à l'espace adressable physique !

Les deux mécanismes de translation, la segmentation et la pagination, sont distincts et permettent deux niveaux d'adressage. Bien qu'ils emploient tous les deux des tables stockées en mémoire, les structures de ces dernières sont différentes. Plus exactement, les tables de segments sont placées dans l'espace adressable linéaire, tandis que les tables des pages se trouvent dans l'espace adressable physique. Ainsi, les tables des segments peuvent être déplacées par le mécanisme de pagination sans qu'il y ait interaction entre les deux mécanismes ! Le mécanisme de segmentation traduit les adresses virtuelles en adresses linéaires, et accède à ses propres tables dans la mémoire linéaire, sans savoir que le mécanisme de pagina-

tion est en train de transformer ces adresses linéaires en adresses physiques.

De même, le mécanisme de pagination ne sait rien de l'espace adressable virtuel que les programmes utilisent pour générer des adresses. La pagination transforme simplement les adresses linéaires en adresses physiques et accède à ses propres tables en mémoire physique, sans savoir que l'espace adressable virtuel existe ou même que la segmentation existe.

Le paragraphe suivant est une brève digression sur la mémoire virtuelle, et inclut une discussion rapide sur les avantages relatifs de la segmentation et de la pagination. Puis, nous introduirons quelques concepts de base sur la protection avant de revenir à une description plus approfondie de la segmentation et de la pagination.

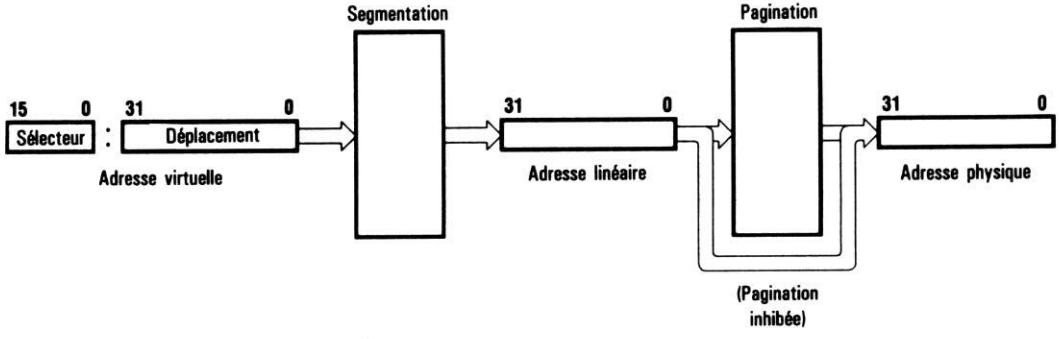


Figure 5.1 : *Fonction de translation des adresses virtuelles en adresses physiques.*

Mémoire virtuelle

On emploie la technique de la mémoire virtuelle pour donner l'illusion d'un espace mémoire beaucoup plus grand que la mémoire principale physique disponible sur le système. Ceci permet d'écrire des programmes sans se préoccuper de la taille exacte de la mémoire physique. Ainsi, on peut exécuter sans problème un programme sur un grand nombre de configurations mémoire d'un même système. De plus, vous utilisez la taille de la mémoire virtuelle qui est beaucoup plus importante que la mémoire physique de n'importe quelle configuration.

Ceci est possible grâce à un mécanisme de translation de la mémoire, associé à des possibilités importantes de stockage sur disque dur rapide. A tout moment, une petite partie de l'espace adressable virtuel est stockée dans la mémoire centrale et le reste sur disque. Comme seule la partie en mémoire centrale est utilisable par le processeur, cette technique doit s'assurer qu'au cours de son exécution le programme n'a besoin que d'une petite quantité de la mémoire virtuelle. Au cours de l'exécution, les références mémoire changent. Ainsi, le système doit charger à partir du disque dur certaines parties de la mémoire virtuelle et en sauver d'autres à partir de sa mémoire centrale.

Par exemple, un programme d'application conséquent peut offrir un large choix de fonctions. Si vous sélectionnez une fonction à partir du menu, vous ferez exécuter plusieurs sous-programmes spécifiques à cette fonction, mais vous n'utiliserez pas les sous-programmes associés aux autres fonctions. Un système à mémoire virtuelle chargera la partie code et les données associées à cette fonction en mémoire centrale (si ce n'est pas déjà fait). Le code et les données des autres fonctions seront transférés sur le disque (ou y resteront). Tant que la mémoire physique sera suffisamment importante pour contenir le code et les données d'une fonction unique, la mémoire physique pourra avoir une taille beaucoup plus petite que celle de l'application.

Le mécanisme de translation des adresses offre deux possibilités de gestion de la mémoire virtuelle. Tout d'abord, il n'autorise l'accès qu'aux parties de la mémoire virtuelle qui sont réellement en mémoire centrale, et transforme les adresses virtuelles (correspondant aux parties résidant en

mémoire) en adresses mémoire physiques. Si un programme fait appel à une adresse virtuelle correspondant à une partie de la mémoire virtuelle qui ne réside pas en mémoire centrale, il y aura génération d'une exception. Le système d'exploitation peut traiter cette exception en transférant du disque dans la mémoire principale la partie manquante, et en mettant à jour les tables appropriées. Ainsi, le programme peut se poursuivre lorsque la routine de traitement de cette interruption lui rend la main. L'instruction qui a engendré l'exception sera exécutée à nouveau, et le programme pourra continuer sans problème.

Ce mécanisme permet aussi d'effectuer des statistiques sur les parties de la mémoire virtuelle qui résident en mémoire centrale. Ainsi, le système d'exploitation pourra optimiser et choisir quelle partie doit être renvoyée sur le disque lorsque la mémoire centrale devient insuffisante.

Sur le 80386, c'est la pagination qui offre la meilleure gestion de la mémoire virtuelle. Elle utilise des blocs de taille fixe pour gérer la mémoire, alors que la segmentation travaille sur des blocs de taille variable. Il s'avère que les blocs de taille fixe employés par la pagination conviennent mieux à la gestion de la mémoire physique, qu'elle soit en mémoire centrale ou sur disque. Cette supériorité dans le traitement de l'aspect physique de la mémoire virtuelle en fait la technique idéale pour gérer la mémoire virtuelle.

D'un autre côté, les blocs à taille variable employés par la segmentation conviennent mieux au partitionnement logique d'un système complexe. On peut alors définir des unités mémoire conformément à leur utilisation logique sans s'embarrasser des contraintes artificielles imposées par des pages de taille fixe. Chaque segment peut être considéré comme une unité unique, ce qui simplifie la protection et le partage des segments.

Protection

Le 80386 gère deux types importants de protection. Le premier offre la possibilité de séparer totalement deux tâches en donnant à chacune d'elles un espace adressable virtuel différent. On effectue cette opération en asso-

ciant à chaque tâche une table de transformation (adresse virtuelle en adresse physique) (*map*) différente. Le second mécanisme de protection est utilisé dans une tâche afin de protéger l'accès des programmes d'application aux segments mémoire du système d'exploitation et à certains registres du processeur.

Protection entre les tâches

Le mécanisme de protection permet de protéger des tâches du type application les unes des autres. Pour cela, le 80386 place chaque tâche dans un espace adressable virtuel différent, et attribue à chaque tâche une *map* différente. La fonction de transformation des adresses de chaque tâche lie les adresses virtuelles d'une tâche à une partie de la mémoire physique, alors que celles d'une autre tâche sont liées à des zones différentes de la mémoire physique. Puisqu'une tâche ne peut pas générer une adresse virtuelle associée à une partie de la mémoire physique utilisée par une autre tâche, les tâches sont bien isolées les unes des autres.

On attribue à chaque tâche une fonction différente de transformation des adresses en disposant simplement de plusieurs tables de pagination pour chaque tâche. Sur le 80386, chaque tâche possède ses propres tables des segments et des pages. Quand le processeur commute sur une autre tâche, il y a commutation simultanée des tables de transformation.

Le système d'exploitation pourra être stocké dans une tâche distincte afin de l'isoler de toutes les autres applications. Cependant, le mécanisme de protection (voir le paragraphe suivant), qui opère sur une tâche, est plus approprié à la protection du système d'exploitation. Ce mécanisme permet à chaque tâche d'accéder au système d'exploitation, tout en conservant la protection du système d'exploitation. Ce dernier est partagé entre toutes les tâches en s'arrangeant pour avoir une partie de la map identique pour toutes les tâches, et en stockant le système d'exploitation dans la partie commune à toutes les tâches. Cette partie commune s'appelle l'*espace adressable global.*

La partie de l'espace adressable virtuel qui est spécifique à chaque tâche - c'est-à-dire la partie qui n'est partagée avec aucune autre tâche - s'appelle l'*espace adressable local.* Celle-ci contient le code et les données "privés" d'une tâche et doit être isolée des autres tâches du système.

Avoir un espace adressable local différent pour chaque tâche implique qu'une référence à une même adresse virtuelle dans deux tâches distinctes soit convertie en deux adresses physiques distinctes. Ceci permet au système d'exploitation de donner à chaque tâche de la mémoire des adresses communes tout en conservant l'isolement des tâches. En d'autres termes, une référence à la même adresse virtuelle dans l'espace adressable global donnera la même adresse physique pour toutes les tâches. Ceci permet de partager des codes et des données communes (par exemple le système d'exploitation). Nous approfondirons les notions d'espaces adressables local et global quand nous parlerons de la segmentation et de la pagination.

L'exemple d'initialisation du Chapitre 7 démontre comment on peut définir les espaces adressables local et global conformément à ce qui vient d'être expliqué. Dans cet exemple, le système d'exploitation se trouve dans l'espace adressable global et est partagé entre toutes les tâches. Le code et les données des autres tâches sont stockés dans l'espace adressable local.

Protection dans une tâche

Dans une tâche, on définit quatre niveaux de privilège pour restreindre l'accès aux segments de cette tâche selon la confidentialité des données contenues dans le segment et du degré d'interaction des différentes parties du programme d'application. Les données les plus confidentielles sont assignées au niveau de privilège le plus haut, auquel seules les parties autorisées du programme peuvent accéder. Les données les moins confidentielles sont assignées au niveau de privilège le plus bas, ce qui permet aux autres parties du programme d'y accéder.

Les niveaux sont numérotés de 0 à 3, 0 étant le plus privilégié et 3 le moins privilégié. Mais vous devrez vous rappeler qu'un niveau avec un

numéro élevé correspond à un privilège faible ! Pour éviter toute ambiguïté, nous n'emploierons pas les termes de *plus grand que* ou *plus petit que* lorsque nous comparerons les niveaux de privilège. Au contraire, nous utiliserons les termes d'*interne* pour parler du plus privilégié et d'*externe* pour le moins privilégié. Le niveau 0 est donc le niveau de privilège le plus interne, et le niveau 3 le niveau le plus externe. La Figure 5.2 illustre cette hiérarchie à quatre niveaux par des cercles concentriques.

Chaque segment mémoire est associé à un niveau de privilège. Ce dernier limite l'accès aux seuls programmes ayant un privilège suffisant. Rappelez-vous (voir le Chapitre 2) que le processeur traite et exécute les instructions à partir du segment adressé par le registre CS. Le *niveau de privilège courant* (Current Privilege Level), ou CPL, est simplement le niveau de privilège du programme qui est en train de s'exécuter. Le CPL détermine à quels segments le programme peut accéder.

Si un programme tente d'accéder à un segment, le niveau de privilège courant est comparé au niveau de privilège du segment afin de déterminer si l'accès est autorisé. On autorise cet accès à un programme de CPL donné pour un segment de données du même niveau, ou d'un niveau externe. Toute tentative d'accès à un segment de niveau interne est illégale, et engendre une exception pour rapporter la violation au système d'exploitation.

Chaque niveau de privilège possède sa propre pile programme (voir le Chapitre 2), afin d'éviter des problèmes de protection qu'entraînerait une pile partagée. Lorsqu'un programme passe d'un niveau à un autre au cours de son exécution, le segment pile est changé pour correspondre à la pile du nouveau niveau. Cette méthode de changement de niveau de privilège sera décrite ultérieurement dans ce chapitre, dans le paragraphe traitant des méthodes de transfert de contrôle.

En général, on utilise les niveaux de privilège en plaçant le noyau (*kernel*) du système d'exploitation au niveau 0, le reste du système d'exploitation au niveau 1, et les applications au niveau 3. De cette façon, le niveau 2 est libre et peut être utilisé par des niveaux de logiciels intermédiaires. Le noyau du système d'exploitation (OS) peut ainsi accéder à

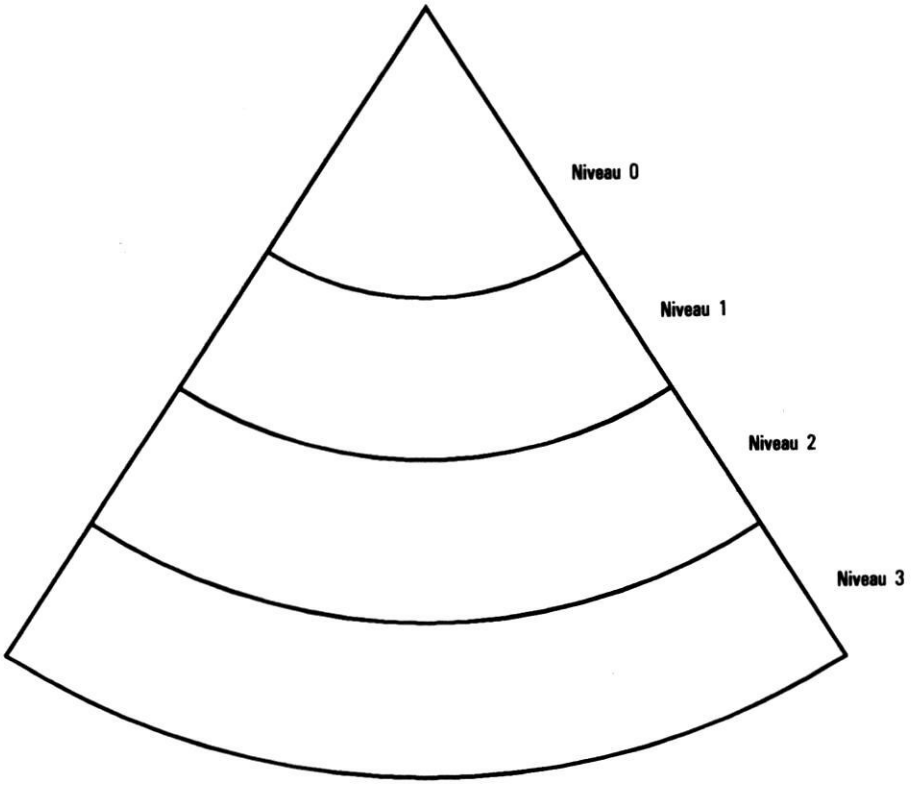


Figure 5.2 : *Quatre niveaux de privilège.*

tous les segments d'une tâche. Le reste du système d'exploitation accédera à tous les segments, sauf ceux du niveau 0. Les programmes d'application du niveau 3 ne pourront accéder qu'à leurs propres segments. Ce type d'allocation protège le noyau du système d'exploitation et empêche les programmes d'application d'accéder à ses segments.

Les Figures 5.3, 5.4 et 5.5 illustrent les quatre niveaux de privilège, l'assignation des segments à ces niveaux, et les règles d'accès aux segments, basées sur le CPL. Le noyau de l'OS au niveau 0 est stocké dans un segment code de niveau 0 appelé $Code_k$ et dans un segment données de niveau 0 nommé $Data_k$. Le reste du système d'exploitation au niveau 1 est stocké dans $Code_{os}$ et $Data_{os}$, et les applications au niveau 3 dans $Code_{ap}$ et $Data_{ap}$. Dans ces exemples, on n'utilise pas le niveau 2. Quand le code du programme d'application du segment $Code_{ap}$ est exécuté, CPL est égal à 3, et seuls les segments $Code_{ap}$ et $Data_{ap}$ sont accessibles (voir la Figure 5.3).

La Figure 5.4 illustre comment s'exécute le système d'exploitation avec CPL = 1, et comment il accède à ses propres segments aussi bien qu'aux segments de l'application.

La Figure 5.5 illustre comment le noyau de l'OS peut accéder aux six segments de ce petit système avec CPL = 0.

Niveaux de protection combinés

Dans les deux paragraphes précédents, nous avons décrit les deux aspects de la protection que permet le 80386 : protection entre les tâches, et protection dans une tâche. Dans ce paragraphe, nous allons décrire comment on peut combiner les deux fonctions pour protéger les applications les unes des autres, pour permettre à toutes les applications de partager le système d'exploitation, et pour protéger le système d'exploitation de toutes les applications.

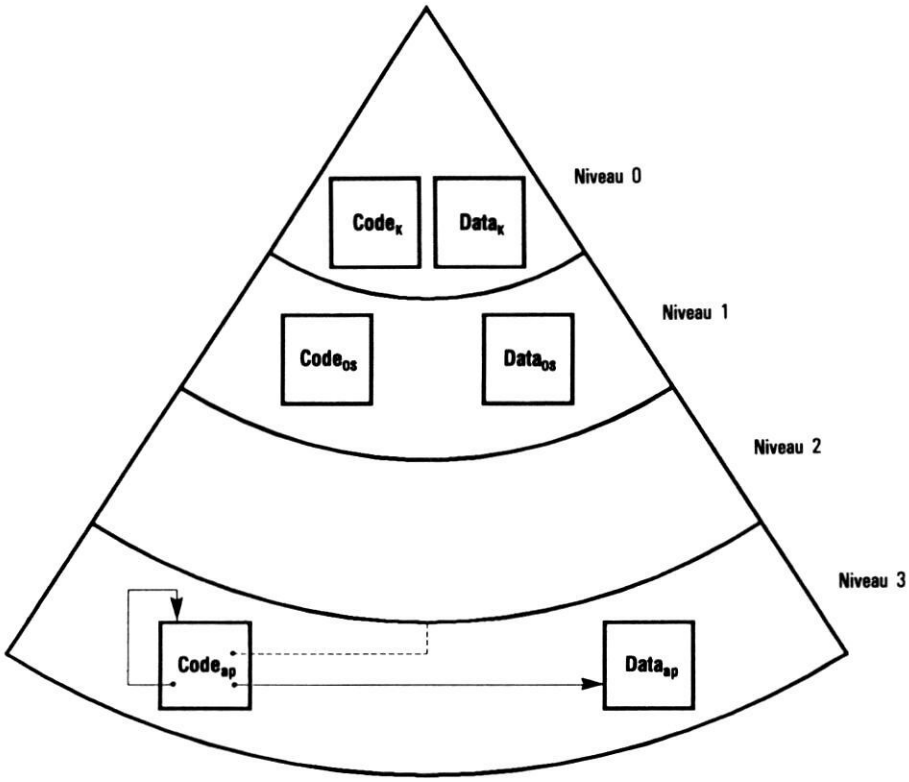


Figure 5.3 : *Segments accessibles au niveau de privilège 3.*

Chaque application est considérée comme une tâche distincte, avec le code et les données (spécifiques à cette application) stockés au niveau 3. Le système d'exploitation est stocké dans des segments au niveau 1. Le noyau est stocké au niveau 0.

Tout ceci est illustré à la Figure 5.6 : les cercles concentriques représentent les niveaux de protection et les lignes radiales les limites de tâche. Ainsi, le système d'exploitation peut être partagé entre toutes les tâches, puisqu'il se trouve dans l'espace adressable global (absence de lignes radiales dans les niveaux 0 et 1). Par conséquent, le système d'exploitation est toujours protégé des programmes d'applications puisque les applications se trouvent au niveau 3, et le système d'exploitation au niveau 0 et 1. Bien que toutes les applications soient au niveau 3, elles sont stockées dans des espaces adressables virtuels différents, dans différentes tâches pour permettre la protection entre les applications.

SEGMENTATION

La segmentation décompose la mémoire virtuelle en un ensemble d'unités de taille variable, appelées *segments*. Le modèle de segmentation du 80386 utilise un adressage virtuel à deux composantes : une partie segment et une partie déplacement. Dans le Chapitre 2, nous avons introduit ce type d'adressage en décrivant les registres segments qui contiennent la partie segment de l'adresse, et en détaillant les modes d'adressage disponibles pour générer la partie déplacement des adresses. Dans ce chapitre, nous allons décrire de façon précise la partie segment d'une adresse virtuelle en expliquant comment sont définis les segments et comment les adresses virtuelles sont transformées en adresses linéaires. Dans le Chapitre 3, nous avons décrit les instructions qui permettent de charger les sélecteurs dans les registres segments (ceci afin de rendre les segments adressables). Dans ce chapitre, nous allons expliquer comment rendre un segment adressable lorsqu'un registre segment est chargé par un sélecteur.

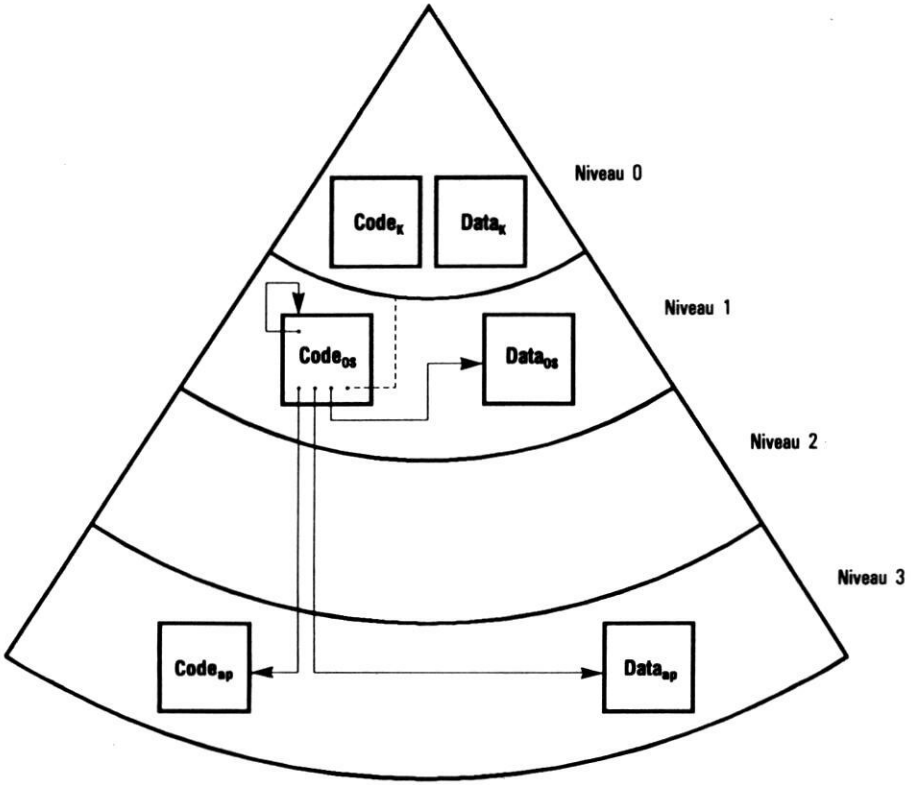


Figure 5.4 : *Segments accessibles au niveau de privilège 1.*

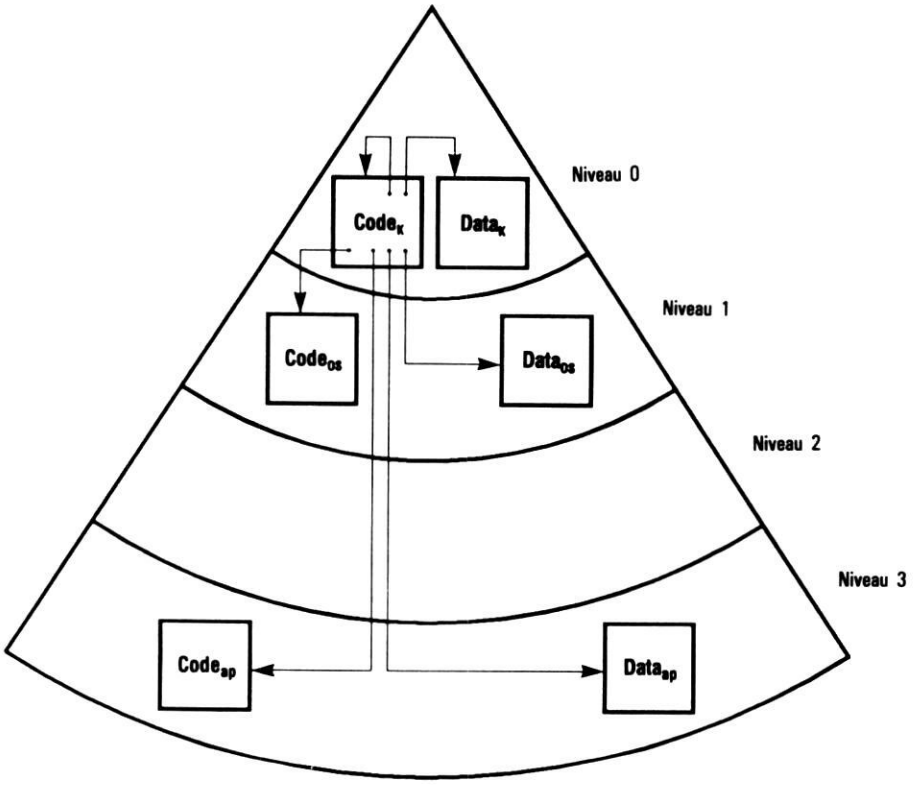


Figure 5.5 : *Segments accessibles au niveau de privilège 0.*

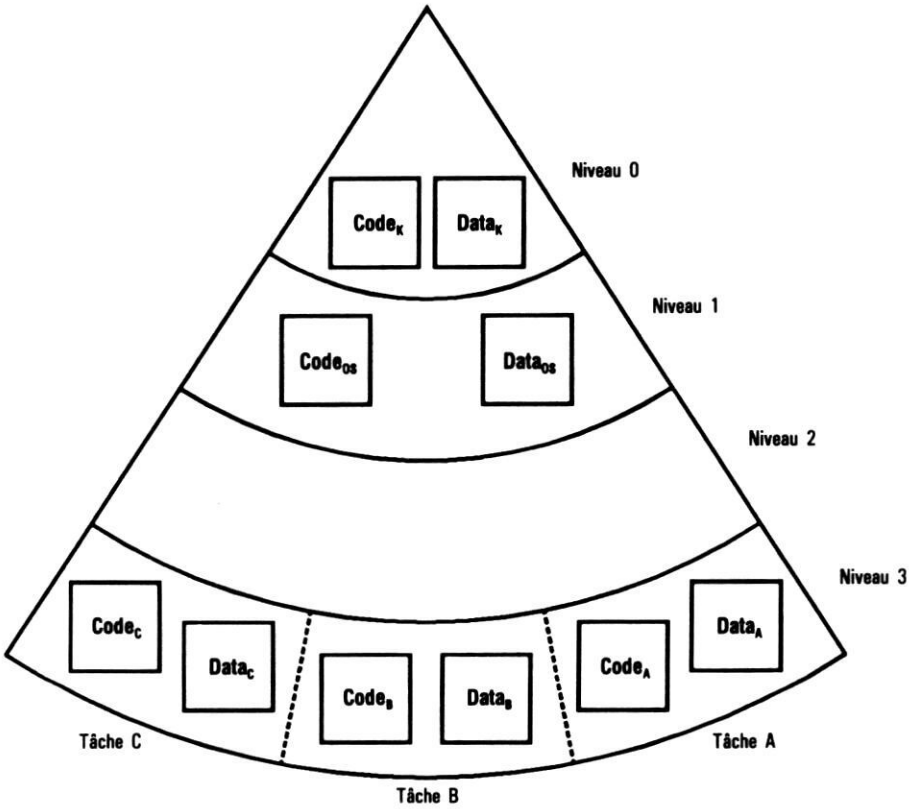


Figure 5.6 : *Protection entre les tâches et dans une tâche.*

Les segments sont à la base du mécanisme de transformation des adresses (virtuelles en linéaires). Chaque segment est défini par trois paramètres :

1 *L'adresse de base* du segment détermine l'adresse de départ du segment dans l'espace adressable linéaire. L'adresse de base est une adresse linéaire correspondant à une adresse virtuelle de déplacement nul dans le segment.

2 *La limite* du segment, qui est le plus grand déplacement qu'on peut employer dans le segment pour une adresse virtuelle, définit la taille du segment.

3 *Les attributs* du segment donnent les caractéristiques du segment (si le segment peut être lu, écrit ou exécuté), le niveau de privilège du segment, etc.

La limite du segment définit la taille du segment dans l'espace adressable linéaire. L'adresse de base et la limite définissent le nombre d'adresses linéaires contenues dans un segment. Les adresses virtuelles dans un segment (adresses variant de 0 à la *limite*) correspondent à des adresses linéaires variant de la *base* à *base* + *limite*. Une adresse virtuelle dans un segment avec un déplacement supérieur à la limite du segment n'a aucun sens, et si elle est utilisée, elle générera une exception. On aura aussi une exception si les attributs du segment ne permettent pas l'accès au segment. Par exemple, le 80386 détecte une exception si vous tentez d'écrire dans un segment du type lecture.

La Figure 5.7 montre comment les segments sont relogés depuis l'espace adressable virtuel (à gauche) dans l'espace adressable linéaire. Trois segments sont définis - A, B, C - avec leurs tailles données par $Limite_A$, $Limite_B$, et $Limite_C$. La transformation est définie par les adresses de bases de segments $Base_A$, $Base_B$, et $Base_C$. Cette transformation est représentée symboliquement par des hachures connectant les segments A, B, C de l'espace virtuel aux régions correspondantes de l'espace linéaire défini par $Base_A$ à $Base_A + Limite_A$ pour le segment A, par $Base_B$ à $Base_B + Limite_B$ pour le segment B, $Base_C$ à $Base_C + Limite_C$ pour le

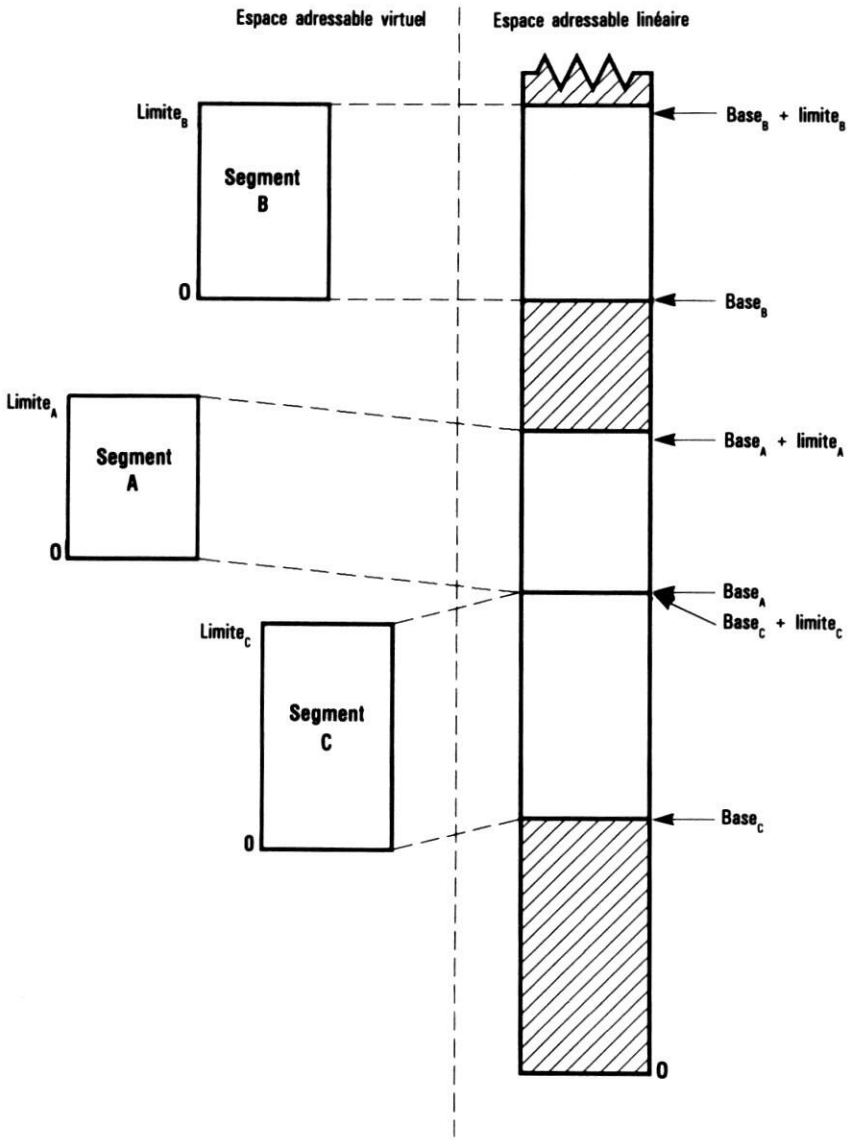


Figure 5.7 : *Transformation des adresses virtuelles en adresses physiques.*

segment C. Notez que le segment A est stocké juste au-dessus du segment C. Cela signifie que $Base_A = Base_C + Limite_C + 1$. Il y a un espace de séparation entre la fin du segment A et le début du segment B.

Chaque référence mémoire vérifie si la partie déplacement d'une adresse est bien comprise dans les limites du segment. Par conséquent, un très grand déplacement dans le segment C (plus grand que $Limite_C$) n'adressera pas de la mémoire dans le segment A, mais au contraire générera une exception pour communiquer cette violation de limites de segment au système d'exploitation. Ce test des limites permet au système d'exploitation de lier le segment A au privilège 3 et le segment C à un privilège interne, et de s'assurer que le programme de niveau 3 ne peut pas employer un déplacement "illégal" dans un segment A pour accéder à une mémoire du segment C, même si les segments se touchent dans l'espace adressable linéaire.

L'adresse de base la limite, et les attributs de protection d'un segment sont stockés dans un *descripteur de segment* qui est utilisé au cours du mécanisme de transformation des adresses. Les descripteurs de segments sont rangés en mémoire dans des *tables des descripteurs*, qui sont de simples tableaux de descripteurs de segments. Le sélecteur de segment (voir le Chapitre 2) identifie un segment en précisant l'emplacement du descripteur de ce segment. Dans les paragraphes suivants, nous allons décrire plus précisément les tables des descripteurs, les sélecteurs et les descripteurs.

Les tables des descripteurs de segments

La *table des descripteurs globaux* (Global Descriptor Table) (GDT) et la *table des descripteurs locaux* (Local Descriptor Table) (LDT) sont des segments particuliers qui contiennent les tables des descripteurs de segments. Ces segments sont gérés par le système d'exploitation et les circuits de gestion mémoire du processeur. Ils devraient se trouver dans de la mémoire protégée accessible uniquement par le système d'exploitation afin d'éviter qu'un programme d'application ne puisse modifier les tables de transformation des adresses.

L'espace adressable virtuel est divisé en deux parties égales : une moitié est associée à la GDT, l'autre à la LDT. L'espace adressable virtuel total contient 2^{14} segments. La moitié de cet espace, soit 2^{13} segments, représente l'espace adressable virtuel global associé à la GDT. L'autre moitié représente l'espace adressable local associé à la LDT. On fait référence à un descripteur de segment en indiquant une table des descripteurs (GDT ou LDT) et un numéro de descripteur dans cette table.

Lors d'une commutation de tâches, la LDT courante devient la LDT de la nouvelle tâche, mais la GDT n'est pas modifiée. Ainsi, la moitié de l'espace adressable virtuel associée à la GDT est commune à toutes les tâches du système, mais l'autre moitié associée à la LDT change à chaque commutation de tâches. Les segments qui sont partagés entre toutes les tâches du système sont associés à la GDT. Dans ces segments, sont inclus les segments comprenant le système d'exploitation, et les segments des LDT de toutes les tâches du système. On peut considérer les segments des LDT comme des données appartenant au système d'exploitation.

La LDT contient les descripteurs des segments réservés à une tâche unique. Plusieurs tâches peuvent se partager une LDT commune. Dans ce cas, les mêmes segments sont à la disposition de toutes ces tâches, puisqu'elles ont la même LDT, et toutes les tâches se partagent une GDT unique. Deux tâches peuvent aussi avoir un descripteur pour un segment présent dans leurs deux LDT, afin de partager ce segment sans devoir placer son descripteur dans la GDT (il deviendrait alors accessible à toutes les tâches). Dans ce cas, le système d'exploitation traite différemment ce segment partagé, puisqu'il est associé à deux descripteurs dans deux LDT différentes qui doivent être mis à jour simultanément.

La Figure 5.8 illustre comment les segments d'une tâche peuvent être répartis entre la GDT et la LDT. Il y a huit segments qui contiennent deux applications (A et B) plus le système d'exploitation et son noyau. Il y a deux tâches, une pour chaque application, et chaque tâche possède sa propre LDT. L'application A (resp. B) s'exécute en Tâche A (resp. B), dont la LDT_A (resp. LDT_B) est associée aux segments $Code_A$ et $Data_A$ (resp. $Code_B$ et $Data_B$), qui contiennent le code et les données de cette application. Les deux segments qui contiennent le système d'exploitation,

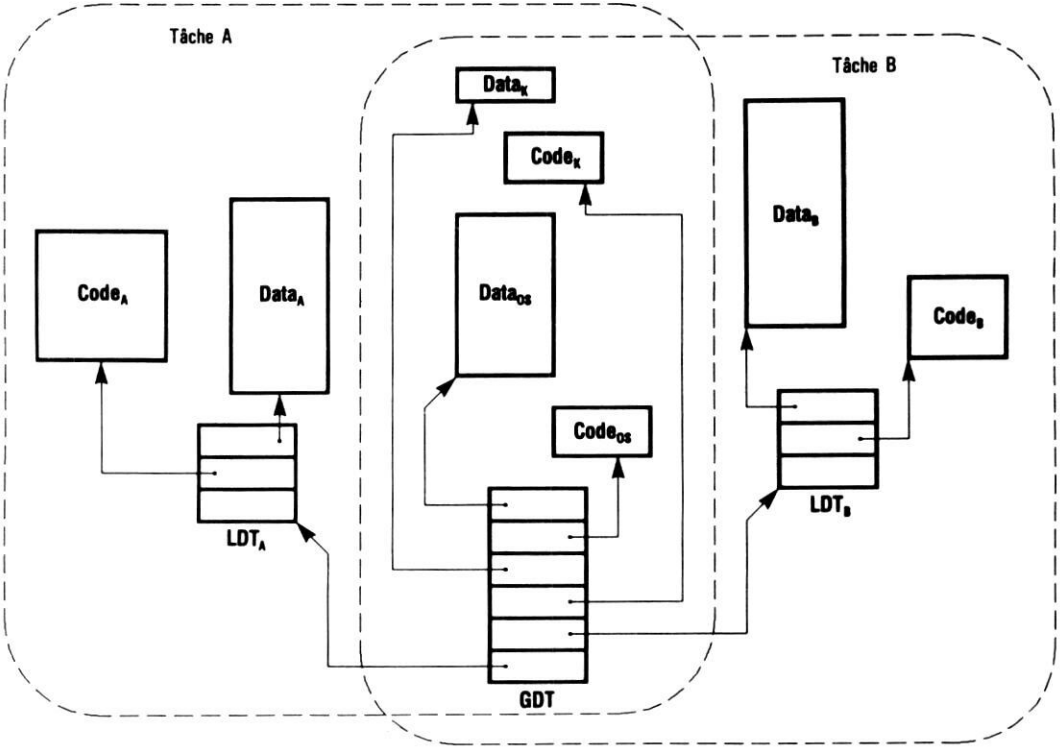


Figure 5.8 : *Espaces adressables local et global.*

$Code_{OS}$ et $Data_{OS}$, et le noyau, $Code_K$ et $Data_K$, sont associés à la GDT afin d'être partagés entre les autres tâches. Les segments LDT_A et LDT_B sont associés à la GDT.

L'ensemble des segments accessibles lors de l'exécution de la Tâche A comprend $Code_A$ et $Data_A$ associés à la LDT_A, plus les segments du système d'exploitation $Code_{OS}$, $Data_{OS}$, $Code_K$ et $Data_K$ associés à la GDT. Lorsque c'est la Tâche B qui s'exécute, l'ensemble des segments accessibles devient $Code_B$ et $Data_B$ plus les segments du système d'exploitation $Code_{OS}$, $Data_{OS}$, $Code_K$ et $Data_K$ associés à la GDT.

Cet exemple montre comment on peut structurer l'espace adressable virtuel pour isoler les tâches d'application entre elles, en employant des LDT distinctes pour chacune d'elles. Lorsque la Tâche A s'exécute, les segments de la Tâche B ne font pas partie de l'espace adressable virtuel, si bien que la Tâche A n'a aucun moyen d'y accéder. De même, lorsque la Tâche B s'exécute, on ne peut adresser les segments de la Tâche A. Cette utilisation des LDT permet le cloisonnement des tâches d'application. Au début de ce chapitre, nous avions déjà parlé de cette contrainte nécessaire à une protection efficace.

Sélecteurs de segment

Dans le Chapitre 2, nous avons introduit la notion de sélecteur de segment en tant que composante segment d'une adresse virtuelle à deux composantes. Un sélecteur de segment identifie un segment ; c'est en quelque sorte le nom du segment. Un sélecteur de segment a une taille de 16 bits et est composé de trois sous-champs (voir la Figure 5.9). Le champ RPL (Requested Privilege Level, niveau de privilège requis) se trouve dans les 2 bits de poids faible. Il fait partie du modèle de protection des segments et on en parlera ultérieurement. Le bit TI (bit 2) identifie la table des descripteurs dans laquelle se trouve le descripteur de ce segment. TI = 0 indique que le descripteur du segment est dans la GDT. TI = 1 indique que le descripteur se trouve dans la LDT. Le champ d'index est contenu dans les 13 bits de poids fort du sélecteur, et fournit l'index du descripteur de ce segment dans les GDT ou LDT.

Figure 5.9 : *Format du sélecteur.*

Ainsi que vous pouvez le voir, un sélecteur identifie un segment en localisant son descripteur. Le bit TI détermine la table des descripteurs contenant le descripteur, et le champ d'index donne le descripteur dans cette table. Le descripteur contient toutes les informations nécessaires pour accéder à un segment, telles que l'adresse de base du segment, sa taille et ses attributs. On peut considérer que les sélecteurs de segments déterminent indirectement le descripteur.

Par exemple, le sélecteur suivant identifie le segment 3 dans la GDT, avec RLP = 0. Le champ d'index est égal à 11b (ou 3), et le bit TI est à 0 (indiquant la GDT).

index	TI	RPL
0000000000011	0	00

Le sélecteur suivant identifie le segment 8191 dans la LDT, avec RPL = 3. Le champ d'index est égal à 1111111111111b (ou 8191), et le bit

TI est à 1 (indiquant la LDT). Notez qu'on peut donner une valeur d'index plus grande.

index	TI	RPL
1111111111111	1	11

Sélecteur nul

Le sélecteur suivant est une valeur particulière qu'on appelle le *sélecteur nul* : TI = 0 et index = 0, mais le champ RPL peut avoir une valeur quelconque. Comme on le verra dans le paragraphe traitant des accès à la mémoire, toute référence mémoire qui utilise un registre segment contenant le sélecteur nul générera une exception. Le sélecteur nul est utile lorsqu'on a besoin d'un sélecteur. Notez qu'un sélecteur qui a TI = 1 et index = 0 n'est pas un sélecteur nul, mais au contraire il identifie le segment associé au premier descripteur de la LDT.

index	TI	RPL
0000000000000	0	RPL

Comme le sélecteur nul est caractérisé par TI = 0 et index = 0, il occupe le premier emplacement de la GDT. Lorsque le système d'exploitation construit les tables des descripteurs, il doit mettre un sélecteur "dummy" dans le premier emplacement de la GDT pour prendre en compte le sélecteur nul. Le processeur ne fait jamais référence à ce descripteur.

Utilisation du champ RPL

Chaque fois qu'un programme tente d'accéder à un segment, le processeur compare le niveau de privilège courant (CPL) avec le niveau de privilège du segment afin de déterminer s'il doit en autoriser l'accès. Le champ RPL du sélecteur modifie ce test du niveau de privilège : il simule l'exécution du programme à un niveau de privilège qui serait le plus externe des CPL et RPL.

RPL permet au système d'exploitation d'affaiblir son CPL lorsqu'il travaille avec des sélecteurs qui ont été passés comme paramètres par des programmes externes. Il positionne tout simplement le RPL de tous les sélecteurs (passés en paramètres) à la valeur du CPL du programme qui a passé le sélecteur. Cette opération s'effectue efficacement grâce à l'instruction ARPL (voir le Chapitre 3). Ensuite, lorsque le système d'exploitation utilise le sélecteur pour accéder à un segment, il teste les niveaux de privilège en employant le CPL du programme appelant (stocké dans le RPL) plutôt que le CPL du système d'exploitation, afin de s'assurer que le système d'exploitation n'accède pas à un segment au nom d'un programme appelant (à moins que le programme lui-même n'ait accès à ce segment).

Le CPL est rangé dans le champ RPL du registre CS. Chaque fois qu'un sélecteur de segment code est chargé dans le registre CS, le processeur range automatiquement le CPL dans le champ RPL de CS. Un programme peut examiner le CPL en rangeant le sélecteur de CS dans un registre général ou dans la mémoire.

Descripteurs de segments

Dans les paragraphes précédents, nous avons parlé des tables des descripteurs et des sélecteurs de segments qui identifient les segments en localisant leurs descripteurs. Il est temps maintenant de décrire les descripteurs eux-mêmes. Avant de donner le format exact d'un descripteur de segment, nous allons énoncer de façon abstraite les champs clés d'un des-

cripteur (si vous êtes impatient, vous pouvez toujours vous reporter à la Figure 5.10 qui décrit le format d'un descripteur de segment). Chaque descripteur de segment a une taille de 8 octets, et contient trois champs : une adresse de base du segment, une limite de segment, et les attributs du segment.

Base et limite

L'adresse de base est constituée de 32 bits, ce qui permet à un segment de commencer à n'importe quelle adresse d'octet dans l'espace adressable linéaire. La limite est aussi codée sur 32 bits, mais seuls 20 bits se trouvent dans le descripteur. Les limites de segment peuvent avoir une granulité d'un octet ou de 4 Ko, ce que définit un des 12 bits d'attributs. Par conséquent, la limite sur 20 bits permet de spécifier des tailles de segments variant de 1 octet à 1 Mo (par incréments de 1 octet) ou de 4 Ko à 4 Go (par incréments de 4 Ko). L'emploi d'une granulité de 4 Ko permet de spécifier des limites de segments plus grandes, mais laisse 12 bits disponibles pour les attributs décrits ultérieurement.

Par exemple, un segment avec :

Base =	012345678
Limite =	0
Granulité =	0 (granulité d'octet)

décrit un segment ayant une longueur d'un octet, à l'adresse linéaire 12345678. Dans ce segment, le seul déplacement valide est 0 et il permet d'accéder à l'octet unique de ce segment. Tous les autres déplacements généreront des exceptions.

Quand on spécifie une limite de granulité 4 Ko, le système calcule la limite complète à partir de la limite 20 bits stockée dans le descripteur en utilisant la formule suivante :

```
Limite Segment = Limite Descripteur * 4K + 4095
               = Limite Descripteur << 12 + 0FFFh
```

En d'autres termes, les 20 bits du champ limite représentent les 20 bits de poids fort de la limite du segment. Les 12 bits supérieurs de la limite du segment sont fixés à 0FFFh, si bien qu'une limite de 0 donne un segment de 4 Ko. Et surtout, la limite de descripteur la plus grande, 2^{20}-1 (0FFFFFh), indique un segment de taille 4 Go.

Par exemple, un segment avec :

Base =	12345000h
Limite =	0
Granulité =	1 (granulité de 4 Ko)

décrit un segment qui commence à l'adresse linéaire 12345000h, et d'une longueur de 4 Ko. Des déplacements de 0 à 0FFFh sont valides dans ce segment, et donnent des adresses linéaires variant de 12345000h à 12345FFFh.

Un autre bon exemple est un segment avec :

Base =	0
Limite =	0FFFFFh
Granulité =	1 (granulité de 4 Ko)

décrivant un segment qui occupe la totalité des 4 Go de l'espace linéaire. La limite est 0FFFFFh, si bien que le plus grand déplacement qui puisse être spécifié est 0FFFFFh * 1000h + 0FFFh (c'est-à-dire 0FFFFFFFFh). Si vous comptez correctement le nombre de F dans ce déplacement, vous verrez qu'on obtient exactement un segment de taille 4 Go. Puisque l'adresse de base est 0, ce segment est directement implanté au sommet de l'espace adressable linéaire de 4 Go.

On peut étendre la taille des segments en augmentant leur limite (ce qui peut nécessiter un déplacement des segments dans l'espace adressable linéaire pour faire de la place pour ce segment plus important !). Cela

marche très bien pour des segments de données qui peuvent "croître" en mémoire (*expand-up*), mais ne sert à rien dans le cas d'un segment contenant une pile. Rappelez-vous (voir le Chapitre 2) que la pile programme est implantée dans le segment adressé par le registre segment SS, que le registre ESP contient le déplacement du sommet de la pile, et que cette dernière croît vers le bas (*expand-down*) lorsqu'on empile. Par conséquent, on place habituellement le début de la pile au sommet du segment (à la limite du segment), et elle croît vers le déplacement 0 du segment. Des segments qui grandissent vers les adresses inférieures donnent des piles croissant vers leur direction naturelle : le bas !

Ce type de segment joue le rôle d'une limite inversée. Dans un segment classique, le champ limite divise les 4 Go de déplacements possibles en deux catégories : de 0 à la limite de segment (déplacements légaux), et de limite + 1 à 4 Go - 1 (déplacements illégaux). Pour ces segments, la signification légal/illégal est inversée. Les déplacements de 0 à la limite de segment sont illégaux (ce qui engendrera une exception), et les déplacements de limite + 1 à 4 Go - 1 sont légaux.

Un des attributs du segment sert à indiquer dans quel sens le segment peut croître. Le processeur fait référence à cet attribut pour déterminer comment utiliser le champ limite afin de tester toute violation de limites.

Attributs de segment

Le niveau de privilège du segment est rangé dans le champ attribut de chaque type de descripteurs de segments, et c'est pourquoi on l'appelle *niveau de privilège du descripteur* ou DPL (Descriptor Privilege Level). Ce champ est constitué de deux bits contenant le niveau de privilège, qui est un nombre compris entre 0 et 3.

Le bit *Présent* (P) est le second attribut commun à tous les types de descripteurs. S'il est levé, il indique qu'on peut employer le descripteur dans le mécanisme de transformation des adresses virtuelles en adresses linéaires. S'il est baissé, le descripteur n'est pas valide pour cette transfor-

mation, et toute tentative d'utilisation du descripteur générera une exception.

Un autre bit d'attribut commun à tous les descripteurs est le bit *DType*, qui permet de distinguer deux sortes de descripteurs. Si le bit DType est égal à 1, le descripteur est associé à un segment mémoire. S'il est égal à 0, le descripteur est associé à un segment système ou à un guichet. La LDT est un des segments système dont nous avons déjà parlé. Nous décrirons les guichets plus tard.

Dans les paragraphes suivants, nous allons parler du codage des descripteurs pour des segments mémoire, des segments système et des guichets. Bien que les segments mémoire et les segments système aient un format similaire, ils ont des attributs différents. Nous en reparlerons. Le format des descripteurs de guichets est tout à fait différent de celui des descripteurs de segments mémoire ou système.

Format des descripteurs de segments mémoire

Les segments mémoire contiennent tout le code et toutes les données auxquels un programme peut accéder directement. La Figure 5.10 illustre le format des descripteurs des segments mémoire. Le descripteur est constitué de huit octets avec l'octet de poids faible (à l'adresse *m*) à droite. Considérons les attributs stockés dans les octets aux déplacements 5 et 6. Le bit DType est le bit 4 au déplacement 5. Il est égal à 1 pour les segments mémoire (appelé DT1 à la Figure 5.10) et à 0 pour les segments système et les guichets.

L'adresse de base est divisée en deux parties : les bits 23...0 sont placés dans trois octets commençant au déplacement 2, et les bits 31...24 dans l'octet au déplacement 7. Le champ limite aussi est divisé en deux parties : les bits 15...0 sont rangés dans deux octets au déplacement 0, et les bits 19...16 dans les 4 bits de poids faible de l'octet se trouvant au déplacement 6. Les attributs des segments sont placés dans deux octets aux déplacements 5 et 6, et sont traités séparément dans la liste suivante :

G — Le bit G est l'attribut *de granulité* de la limite dont nous avons déjà parlé. G = 0 indique une granulité d'un octet. G = 1 indique une granulité de 4 Ko. Notez que le bit G affecte seulement la granulité de la limite du segment. La base du segment a toujours une granulité d'un octet.

D — Le bit D devrait toujours être égal à 1 pour des logiciels 80386. Les logiciels 80286 devraient avoir ce bit positionné à 1 (voir le Chapitre 9). Ce bit donne un *Défaut* pour les segments qui sont exécutables "expand-down" ou adressables par le registre SS :

1. Les segments exécutables utilisent le bit D pour fixer par défaut la taille des adresses et des opérandes référencés par les instructions de ce segment. D = 1 indique que la taille par défaut est de 32 bits pour les adresses et de 32 bits ou de 8 bits pour les opérandes (normal pour les programmes 80386). D = 0 indique que la taille par défaut est de 16 bits pour les adresses ou de 8 bits pour les opérandes, pour assurer la compatibilité avec le 286. On peut employer un préfixe d'instructions pour obtenir une taille différente de celle prise par défaut.

2. Les segments "expand-down" utilisent le bit D pour déterminer la frontière supérieure du segment. D = 1 indique une limite supérieure de 4 Go pour le segment. D = 0 indique une limite supérieure de 64 Ko pour assurer la compatibilité avec le 286.

3. Les segments adressés par le registre SS emploient le bit D pour déterminer s'il faut utiliser le registre 32 bits ESP (si D = 1) pour les références implicites à la pile (telles que PUSH et POP), ou le registre 16 bits SP (D = 0) pour assurer la compatibilité avec le 286.

AVL — Le bit *Available-to-Software* peut être utilisé par tous les logiciels. Le 80386 n'interprète pas ce bit, et Intel assure que tous

les futurs processeurs compatibles avec le 80386 ne l'emploieront pas.

P — Nous avons déjà parlé du bit *Présent*. P = 1 indique que le descripteur est valide pour la transformation des adresses. P = 0 indique que le descripteur n'est pas valide, et son emploi générera une exception.

DPL — Le *niveau de privilège du descripteur* (Descriptor Privilege Level) définit le niveau de privilège associé au segment.

DT — Le bit *DType* fait la distinction entre les segments mémoire (DType = 1) et les segments système et les guichets (DType = 0). Ce champ s'appelle DT1 dans la Figure 5.10.

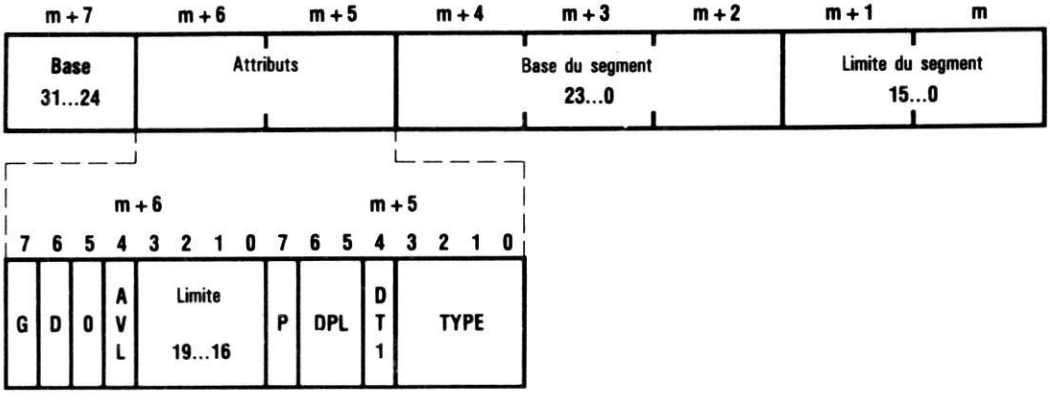


Figure 5.10 : *Format des descripteurs de segments mémoire.*

Type

Le champ de 4 bits *Type* définit le type de descripteur mémoire. Les codes sont donnés dans le Tableau 5.1. Les attributs de lecture, d'écriture et d'exécution du Tableau 5.1 ne demandent pas plus d'explications. Nous avons déjà parlé de l'attribut de limite "expand-down", et nous parlerons ultérieurement des attributs conformes pour les segments codes. Un descripteur est marqué "accessed" par le 80386 lorsque le sélecteur correspondant est chargé dans le segment registre. Le logiciel du système d'exploitation peut examiner l'attribut "accessed" pour voir si le processeur a accédé à un descripteur depuis la dernière fois où le logiciel à mis le bit à 0.

TYPE	DEFINITION
0	Lecture
1	Lecture, "accessed"
2	Lecture/Ecriture
3	Lecture/Ecriture, "accessed"
4	Lecture, "expand-down"
5	Lecture, "expand-down", "accessed"
6	Lecture/Ecriture, "expand-down"
7	Lecture/Ecriture, "expand-down", "accessed"
8	Exécution
9	Exécution, "accessed"
A	Exécution/Lecture
B	Exécution/Lecture, "accessed"
C	Exécution
D	Exécution, conforme, "accessed"
E	Exécution/Lecture, conforme
F	Exécution/Lecture, conforme, "accessed"

Tableau 5.1 : *Types de descripteurs de segments mémoire.*

Segments système et guichets

Les segments système sont des segments particuliers utilisés par le mécanisme de segmentation du 80386. Les guichets ne décrivent pas des segments, mais contiennent des pointeurs. On parlera plus tard dans ce chapitre des segments système et des guichets. On définit ici le format de leurs descripteurs pour référence.

La Figure 5.11 illustre le format des descripteurs de segments système. Les champs de base et de limite sont les mêmes que ceux des descripteurs de segments mémoire, c'est-à-dire qu'on retrouve le bit de granulité (G), le bit Présent (P), le champ DPL, le bit DType, et même le champ

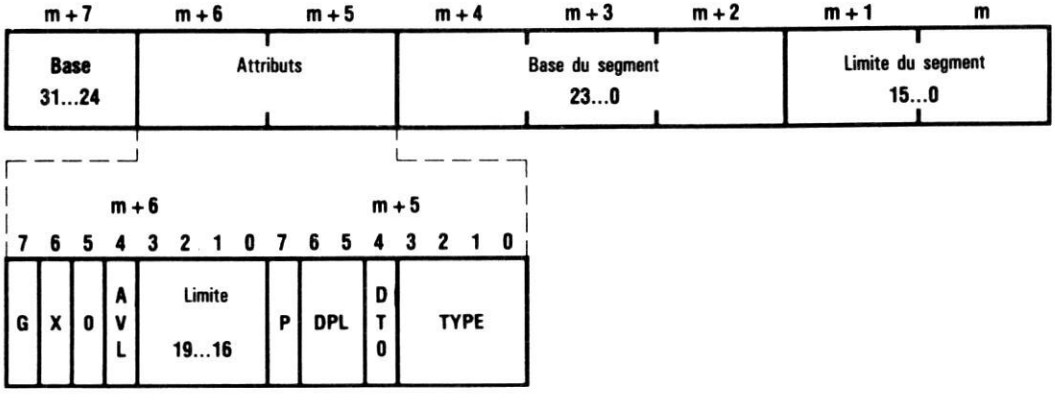


Figure 5.11 : *Format des descripteurs de segments système.*

Available-to-Software (AVL). Le champ Type est présent, mais codé différemment. Il ne manque que le bit Défaut (D), qu'on ignore ici (on l'appelle X dans la Figure 5.11). Le bit DType est égal à 0 (DT0), ce qui indique que ce segment est un segment système ou un descripteur de guichet.

Comme pour les descripteurs de segments mémoire, l'octet situé au déplacement 5 contient un champ Type de 4 bits, qui détermine si c'est un descripteur de segment système ou de guichet. Le champ Type code les deux types de descripteurs (voir le Tableau 5.2). On a déjà parlé du type de segment système LDT. Nous décrirons ultérieurement les types de segments système suivants : les TSS 386 Disponibles et les TSS 386 Occupés, les guichets d'appel 386 et les guichets de tâche ; dans le Chapitre 6, les guichets de trappe et d'interruption 386 ; dans le Chapitre 9, les TSS 286 et les guichets 286 qui sont les contreparties 16 bits des types 386.

TYPE	DEFINITION
0	Indéfini
1	TSS 286 Disponible
2	LDT
3	TSS 286 Occupé
4	Guichet d'Appel 286
5	Guichet de Tâche
6	Guichet d'Interruption 286
7	Guichet de Trappe 286
8	Indéfini
9	TSS 386 Disponible
A	Indéfini
B	TSS 386 Occupé
C	Guichet d'Appel 386
D	Indéfini
E	Guichet d'Interruption 386
F	Guichet de Trappe 386

Tableau 5.2 : *Codage du champ Type pour les segments système et les guichets.*

La Figure 5.12 donne le format des descripteurs de guichets. Les guichets sont composés d'un pointeur de 48 bits et de 16 bits d'attributs. Le pointeur 48 bits représente la même information que le type de donnée pointeur introduit au Chapitre 2, mais, pour le stockage, son déplacement est séparé en deux parties. La partie sélecteur du pointeur est rangée dans les octets aux déplacements 2 et 3. Le déplacement 32 bits est placé dans deux endroits différents. Les 16 bits de poids faible du déplacement sont rangés au déplacement 0, et les 16 bits de poids fort au déplacement 6.

Dans un descripteur de guichet, les attributs sont rangés dans les octets situés aux déplacements 4 et 5 :

P — C'est le bit *Présent*. P = 1 indique que le guichet est valide. P = 0 indique qu'il n'est pas valide, et son usage générera une exception.

DPL — C'est *le niveau de privilège du descripteur* et il définit le niveau de privilège associé au guichet.

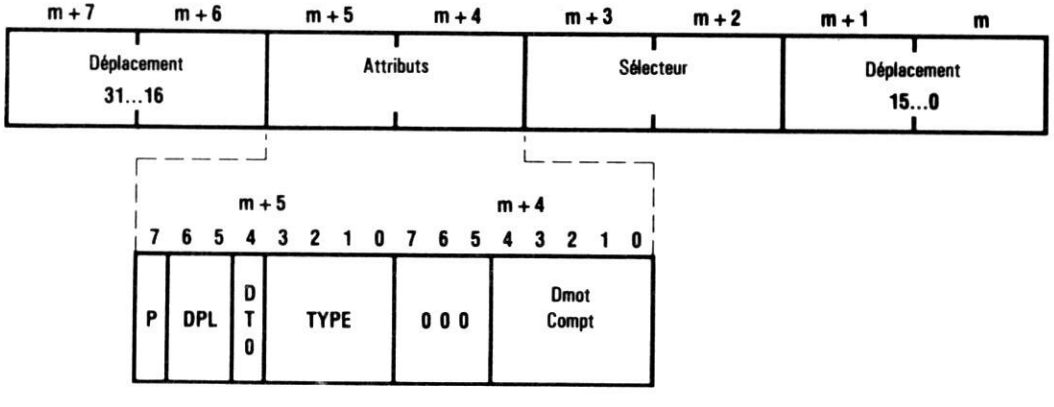


Figure 5.12 : *Format d'un descripteur de guichet.*

DT	C'est le bit *DType*. Il fait la distinction entre les segments mémoire (DType = 1), les segments système et les guichets (DType = 0).
Type	Ce champ de 4 bits définit le type du guichet (voir le Tableau 5.2).
DmotCompt	Ce champ donne le nombre des paramètres dmots (4 octets) à copier d'une pile vers une autre. Cette information est valable si le guichet est employé pour une transition de niveau ou pour un changement de pile. Comme nous l'expliquerons ultérieurement, ceci permet de passer des paramètres par la pile programme (voir le Chapitre 2), même s'il y a transition de niveau ou changement de pile.

Les registres cachés des descripteurs

Pour éviter de faire appel à la table des descripteurs à chaque accès mémoire (lecture et décodage du descripteur de segment), chaque registre segment est associé à un jeu de registres cachés des descripteurs. Ces registres contiennent les informations du descripteur associé au segment identifié par le sélecteur dans le registre segment. Ce dernier est visible par le programmeur, soit en tant que registre segment (voir le Chapitre 2), soit en tant que registre (défini dans ce chapitre). Le programmeur ne voit pas les registres associés qui contiennent le descripteur. C'est pour cela qu'on les appelle les *registres cachés* (Shadow Registers). La Figure 5.13 montre les six registres segments (en traits pleins) avec leurs registres cachés (en pointillé).

Les registres cachés contiennent la base, la limite et les attributs du segment adressé par le registre segment correspondant. Le système les charge avec les informations contenues dans le descripteur chaque fois que le registre segment correspondant est chargé par un sélecteur. Grâce à cela, le descripteur est chargé une fois pour toutes. Puisque les registres cachés

	Registres segments visibles par le programmeur	Registres cachés des descripteurs		
ES	Sélecteur	Base	Limite	Attributs
CS	Sélecteur	Base	Limite	Attributs
SS	Sélecteur	Base	Limite	Attributs
DS	Sélecteur	Base	Limite	Attributs
FS	Sélecteur	Base	Limite	Attributs
GS	Sélecteur	Base	Limite	Attributs

Figure 5.13 : *Registres cachés des descripteurs.*

sont stockés dans le 80386, les circuits de segmentation du 80386 y accèdent très rapidement. On obtient ainsi une exécution performante puisque la plupart des instructions font référence à des données contenues dans des segments dont les sélecteurs ont déjà été chargés dans les registres segments.

Chaque instruction de la partie "Segments multiples" (voir le Chapitre 3) charge un nouveau sélecteur dans le registre segment. Ces instructions chargeront aussi dans les registres cachés les informations contenues dans le descripteur correspondant. Dans le Chapitre 3, nous décrivons ces instructions du point de vue du programmeur, c'est-à-dire en tant qu'instructions qui chargent des sélecteurs dans les registres segments. Les paragraphes suivants vont étendre cette définition en expliquant comment ces instructions testent les descripteurs correspondants, et, si tous

les tests sont positifs, comment elles les chargent dans les registres cachés. Si un des tests est négatif, le système génère une exception et ne modifie ni le sélecteur ni les registres cachés.

Puisque les registres cachés contiennent une copie des informations du descripteur, le système d'exploitation doit veiller à ce que tout changement dans la table des descripteurs se reflète dans les registres cachés. Sinon, il se pourrait qu'un segment voie son adresse de base ou sa limite changer dans la table des descripteurs mais pas dans les registres cachés. Le moyen le plus simple de prendre en compte ce problème consiste à recharger tous les registres segments après chaque modification de descripteurs dans la table des descripteurs. Cette opération permettra de recharger les registres cachés avec les dernières informations de la table des descripteurs.

PAGINATION

La pagination est le second mécanisme de gestion de la mémoire disponible sur le 80386. Elle travaille à un niveau inférieur à celui de la segmentation afin de compléter le mécanisme de transformation des adresses virtuelles en adresses physiques. La segmentation transforme les adresses virtuelles en adresses linéaires. La pagination transforme les adresses linéaires issues de la segmentation en adresses physiques.

Le mécanisme de pagination est activé par le bit PG de CR0. Si PG = 1, la pagination est activée et les adresses linéaires sont transformées en adresses physiques par le mécanisme décrit dans ce paragraphe. Si PG = 0, il n'y a pas pagination et les adresses linéaires générées par le mécanisme de segmentation sont utilisées directement par les adresses physiques.

La segmentation travaille sur des blocs de mémoire de taille variable, alors que la pagination traite des blocs de mémoire de taille fixe appelés *pages*. La pagination divise à la fois les espaces adressables linéaire et physique en pages. Toute page de l'espace adressable linéaire peut être associée à une page de l'espace adressable physique. La Figure 5.14 illustre comment la pagination divise à la fois les espaces adressables linéaire et

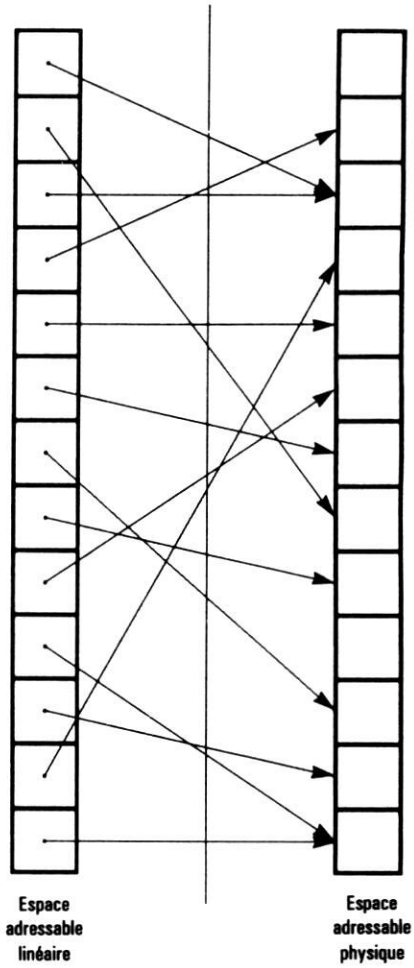


Figure 5.14 : *La pagination transforme les adresses linéaires en adresses physiques.*

physique en pages et reloge arbitrairement ces espaces. A gauche, l'espace adressable linéaire est représenté comme une séquence de blocs de mémoire représentant les pages. A droite, on a l'espace adressable physique. Les flèches associent une page de l'espace linéaire à une page de l'espace physique. Notez que cette correspondance est arbitraire.

La taille des pages utilisées par le 80386 est de 4 Ko. Chaque page est alignée sur une frontière de 4 Ko. Cela signifie que le mécanisme de pagination divise les 2^{32} octets (4 Go) de l'espace adressable linéaire en 2^{20} pages de 2^{12} octets (4 Ko) chacune. Puisqu'une page entière de 4 Ko est considérée comme une unité et que les pages sont alignées sur des frontières de 4 Ko, les 12 bits de poids faible de l'adresse linéaire passent directement par le mécanisme de pagination : ils sont égaux aux 12 bits de poids faible de l'adresse physique. On peut considérer la pagination comme une fonction qui transforme les 20 bits de poids faible de l'adresse linéaire en 20 bits de poids fort de l'adresse physique.

Cette fonction de transformation a été modifiée afin de marquer une adresse comme non valide plutôt que de générer une adresse physique. Une page peut être marquée comme non valide soit parce qu'elle représente simplement une adresse linéaire non permise par le système d'exploitation, soit parce qu'elle correspond à une page de la mémoire virtuelle qui est rangée sur disque plutôt qu'en mémoire physique. Dans le premier cas, le programme générant l'adresse non valide doit être stoppé. Dans le second cas, l'adresse non valide est réellement une requête au système de gestion de la mémoire virtuelle (du système d'exploitation) pour déplacer la page du disque vers la mémoire physique. Donc, le programme doit y accéder. Le système de mémoire virtuelle reconnaît ces pages non valides grâce à l'attribut *présent* se trouvant dans la table des pages.

Structure de la table des pages

La fonction de pagination est décrite dans une table qui réside en mémoire. On appelle cette table la *table des pages*. Elle est placée dans l'espace adressable physique. On peut considérer la table des pages

comme un simple tableau de 2^{20} adresses physiques. La fonction de mapping devient une simple consultation du tableau. Les 20 bits de poids fort de l'adresse linéaire constituent un index dans ce tableau. Cet index permet de sélectionner l'adresse physique correspondant à la page. Les 12 bits de poids faible donnent un déplacement (dans cette page), qui est additionné à l'adresse de base pour constituer l'adresse physique finale. Puisque les adresses de base des pages sont alignées sur des frontières de 4 Ko, les 12 bits de poids faible de l'adresse de base de la page sont nuls. Cela signifie qu'il n'y a pas vraiment d'addition mais simplement une concaténation du déplacement de 12 bits avec les 20 bits de poids fort de l'adresse de base de la page. C'est-à-dire que l'adresse de la base fournit les 20 bits de poids fort de l'adresse physique et le déplacement les 12 bits de poids faible.

Chaque élément de la table des pages est constitué de 32 bits. Puisque les adresses physiques sont codées sur 20 bits, il reste 12 bits pour les attributs de page (par exemple : page présente ou non). Si l'élément de la table des pages, indexé par l'adresse linéaire, est présent, il sera valide et le système obtiendra l'adresse physique à partir de cet élément. Si ce n'est pas le cas, une exception du type page sera générée pour indiquer au système d'exploitation que l'adresse n'est pas valide.

Structure à deux niveaux de la table des pages

La table des pages contient 2^{20} éléments, chacun d'eux étant constitué de 4 octets. Si elle était stockée en une seule table, elle occuperait 4 Mo de mémoire physique continue ! Plutôt que de réserver tant d'espace mémoire pour la table des pages, cette dernière est stockée sous deux niveaux. Ainsi, la transformation des 20 bits d'adresses de poids fort s'effectue en deux étapes, par blocs de 10 bits.

Le premier niveau de la page est appelé le *catalogue de la page* (page directory). Il est stocké dans une seule page de 4 Ko et est constitué de 2^{10} (1 Ko) éléments de 4 octets qui pointent sur les tables de second niveau. On emploie les 10 bits de poids fort des adresses linéaires (bits 31...22)

pour indexer la table de premier niveau qui sélectionne une des 2^{10} tables de second niveau.

Les tables de second niveau sont appelées *tables des pages* (page tables) et ont exactement la même taille qu'une page avec 1 Ko d'éléments de 4 octets. Chacun de ses éléments contient l'adresse de base physique d'une page. Les tables de second niveau sont indexées par les 10 bits 21...12 des adresses linéaires pour obtenir un élément de la table des pages contenant l'adresse de base physique d'une page. Les 20 bits de poids fort de cette adresse physique sont combinés avec les 12 bits de poids faible (le déplacement) de l'adresse linéaire pour former l'adresse physique finale qui sera le résultat du mécanisme de pagination.

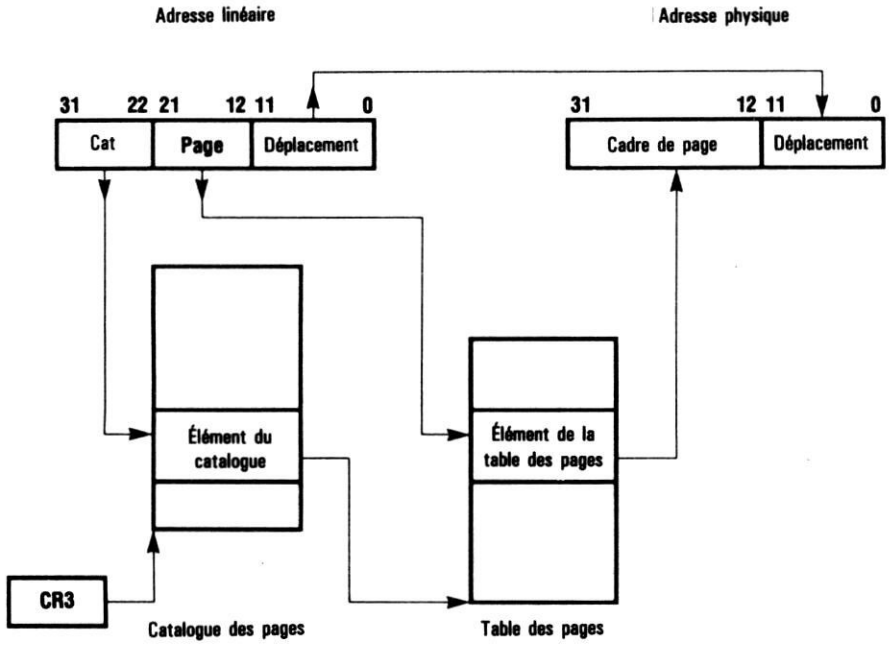


Figure 5.15 : *Structure de la table des pages à deux niveaux.*

La Figure 5.15 explique comment s'effectue la consultation de la table à deux niveaux. Le registre CR3, décrit dans un paragraphe ultérieur, donne une origine à la structure de la table des pages en pointant sur la page de catalogue. Les 10 bits de poids fort d'une adresse linéaire sont employés pour indexer ce catalogue afin d'obtenir un pointeur sur la table appropriée de second niveau. Les 10 bits du centre de l'adresse linéaire sont employés pour indexer la table de second niveau afin d'obtenir les 20 bits de poids fort de l'adresse physique finale. Les 12 bits de poids faible de l'adresse linéaire ne sont pas modifiés par le mécanisme de pagination, et sont concaténés aux 20 bits de poids fort de l'adresse physique (obtenue à partir de la table des pages) pour former l'adresse physique complète.

Tables des pages non présentes

En utilisant une structure de table à deux niveaux, nous n'avons pas résolu le problème des 4 Mo de mémoire nécessaires pour stocker la table des pages. En fait, nous avons même légèrement compliqué le problème, puisque nous avons maintenant besoin d'une page supplémentaire pour le catalogue ! Cependant, la structure à deux niveaux permet de scinder la table des pages en plusieurs pages dans la mémoire, plutôt que d'avoir un morceau continu de 4 Mo. De plus, on n'a pas besoin d'allouer des tables de second niveau pour des parties de l'espace adressable physique qu'on n'utilise pas ou qui n'existent pas. La page de catalogue doit toujours être présente dans la mémoire physique, mais on n'implantera que les tables de second niveau nécessaires. Ceci permet de faire correspondre la taille des tables des pages à la taille de l'espace adressable linéaire effectivement employé.

Chaque élément du catalogue possède un attribut *présent* qui est analogue à celui des éléments de la table des pages. Dans un élément du catalogue, l'attribut présent indique que la table de second niveau correspondante est disponible pour la pagination. Si cet attribut indique que la table de second niveau est présente, la consultation de cette dernière se déroulera conformément à ce que nous avons expliqué précédemment. Si ce n'est pas le cas, il y aura génération d'une exception du type page pour

communiquer au système d'exploitation cet emploi d'une adresse linéaire non valide. L'attribut présent du catalogue permet ainsi au système d'exploitation de n'allouer que le nombre de tables de second niveau nécessaires pour couvrir l'espace adressable linéaire utilisé.

On peut aussi l'employer pour ranger les tables des pages de second niveau dans la mémoire virtuelle. Cela signifie que seul un sous-ensemble des tables de second niveau doit se trouver dans la mémoire physique, le reste étant stocké sur disque. Les éléments du catalogue qui correspondent à des tables de la mémoire physique seront marqués *présents* pour indiquer qu'ils peuvent être employés par la pagination. Une exception du type page causée par l'absence d'une table des pages de second niveau signalera au système d'exploitation qu'il doit charger la table manquante à partir du disque. Ranger les tables des pages en mémoire virtuelle permet d'optimiser la quantité de mémoire physique nécessaire pour stocker les tables de pagination.

Table des pages locales / Table des pages globales

Puisqu'il ne s'agit pas de la structure de table de segments, il n'est pas nécessaire de diviser la table des pages en une table globale et une table locale. Cependant, en s'arrangeant pour que chaque tâche partage une partie du mapping, on peut définir une partie globale de l'espace adressable linéaire. Le résultat de la pagination de cette partie de l'espace adressable linéaire est le même pour chaque tâche dans le système parce que ce sont les mêmes informations qui sont utilisées par le mapping à chaque fois. Les pages de cette partie de l'espace adressable linéaire sont donc appelées *pages globales*. Chaque tâche emploie différemment le reste de l'espace adressable linéaire. La structure de la table fait que chaque tâche emploie une fonction de mapping différente. Les pages de ces parties de l'espace adressable linéaire sont appelées *pages locales*.

La structure de table des pages à deux niveaux permet un partage efficace de l'espace adressable linéaire en partitionnant les tables des pages de second niveau aussi bien que les pages globales elles-mêmes.

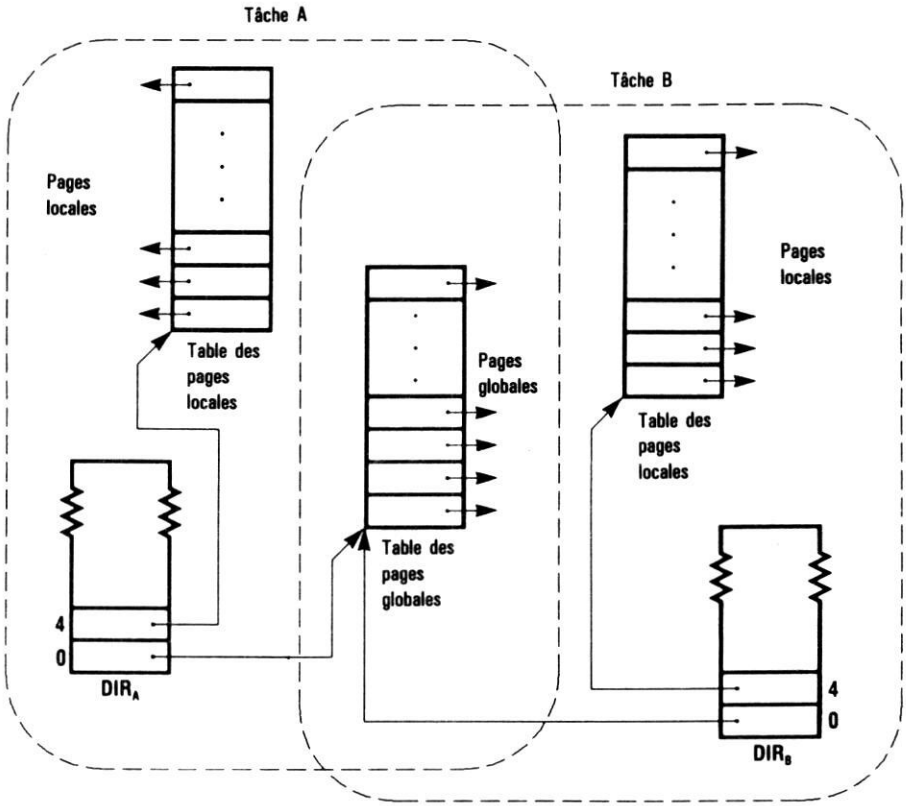


Figure 5.16 : *Tables des pages globales et locales.*

La Figure 5.16 illustre cette technique. Elle montre la structure des tables des pages pour deux tâches A et B. DIR_A (resp. DIR_B) est le catalogue de la tâche A (resp. B). Le premier élément (au déplacement 0) de chaque catalogue pointe sur une table des pages globales, ce qui permet de gérer jusqu'à 2^{10} pages globales (1 Ko). Le second élément (au déplacement 4) de chaque catalogue pointe sur une table des pages locales propre à chaque tâche, ce qui permet de gérer pour chacune d'elles 2^{10} pages locales.

Cet exemple simple est structuré de la façon suivante : les 4 Mo inférieurs de l'espace adressable linéaire sont globaux à toutes les tâches, alors que les 4 Mo suivants sont locaux à chaque tâche. On peut allouer des tables des pages supplémentaires à la zone globale par tranches de 4 Mo. De plus, notez que ce sont les mêmes éléments du catalogue qui pointent sur les tables des pages partagées dans les deux catalogues. C'est important, puisque ainsi cette zone globale sera associée aux mêmes adresses linéaires pour les deux tâches. Dans cet exemple, les quatre premiers Mo de l'espace adressable linéaire (adresses de 0 à 3FFFFFh) se trouvent dans l'espace adressable linéaire global.

Il est très intéressant d'avoir des tables de second niveau partagées, associées à l'espace adressable linéaire global. Tout d'abord, un seul jeu de tables des pages de second niveau doit être alloué. Ces tables sont ensuite partagées entre toutes les tâches, ce qui évite de les recopier à chaque fois. Un second avantage est que, si l'état d'une page partagée change, seul un élément de la table des pages doit être mis à jour. Si une page globale est transférée du disque, ou à partir de celui-ci (*swapping*), en mémoire virtuelle, un seul élément de la table des pages doit être changé.

Comme dans le cas de la segmentation, le code et les données du système d'exploitation et les tables des descripteurs doivent être normalement stockés dans des pages globales. Le code et les données propres à une tâche doivent être stockés dans des pages locales. Les tables des pages et des segments doivent alors être modifiées de façon que les segments associés à la GDT soient rangés dans des pages globales et les segments associés à la LDT dans des pages locales.

Format d'un élément de la table des pages

Les éléments du catalogue et des tables des pages suivent le format de la Figure 5.17. Les bits 31...12 contiennent les 20 bits de poids fort de l'adresse physique pour déterminer un "cadre" de page dans l'espace adressable physique. Les 12 bits restants contiennent les attributs des pages. Nous avons déjà parlé de l'attribut présent. Nous allons donner rapidement une liste des autres attributs, puis dans les prochains paragraphes nous expliquerons leur utilisation. Les bits notés 0 sur la Figure 5.17 sont réservés par Intel pour les processeurs à venir et on doit les positionner à 0 pour assurer la compatibilité ascendante.

P — Le bit 0 est le bit *Présent* dont nous avons déjà parlé. Il indique si un élément est valide pour la transformation d'adresse (P = 1) ou non (P = 0). Une exception est générée si un élément non valide est rencontré soit dans le catalogue, soit dans la table des pages lors de la pagination. Si P = 0, le logiciel peut tout de même utiliser les autres bits de l'élément, comme le montre la Figure 5.18. Le 80386 n'interprète aucun de ces bits si P = 0.

R/W — Le bit 1 est le bit R/W (*Lecture/Ecriture*) (Read/Write). S'il est égal à 1, on peut lire, écrire ou exécuter la page. S'il est égal à 0, on ne peut pas écrire dans la page. Le processeur ignore le bit R/W s'il exécute une tâche au niveau de privilège superviseur (0, 1 ou 2). Le bit R/W d'un élément du catalogue est valable pour toutes les pages associées à cet élément.

U/S — Le bit 2 est le bit U/S (*Utilisateur/Superviseur*). S'il est égal à 1, n'importe quel programme peut accéder à cette page, quel que soit son niveau de privilège (même le niveau utilisateur : niveau 3). S'il est égal à 0, seuls les programmes ayant un privilège superviseur (niveaux 0, 1 et 2) peuvent y accéder. Le bit U/S d'un élément du catalogue est valable pour toutes les pages associées à cet élément.

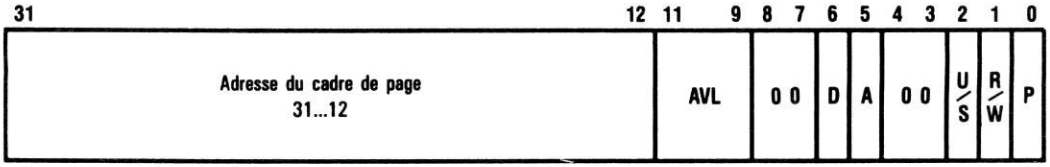


Figure 5.17 : *Format des éléments du catalogue ou des tables des pages.*

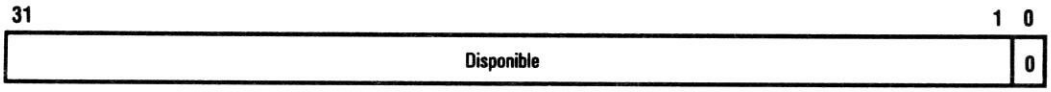


Figure 5.18 : *Format d'un élément non présent du catalogue ou des tables des pages.*

A — Le bit 5 est le bit A (*Accessed*). Le bit A d'un élément d'une table des pages est positionné à 1 par le processeur avant chaque accès à la page associée à cet élément. Le bit A d'un élément du catalogue est positionné à 1 par le processeur avant chaque accès à une des pages associées à cet élément. Le processeur n'efface jamais le bit A, mais le système d'exploitation peut le faire périodiquement pour établir des statistiques sur l'emploi des pages.

D — Le bit 6 est le bit D ou bit *Dirty*. Le processeur met à 1 le bit D d'un élément d'une table des pages de second niveau avant toute écriture d'une page associée à cet élément. Il ne modifie pas le bit D d'éléments du catalogue.

AVL — Le champ AVL est mis à la disposition du logiciel. Le processeur ne le modifie pas, et Intel ne l'emploiera pas dans ses prochains processeurs.

L'exemple suivant illustre de façon précise comment une adresse linéaire est transformée en adresse physique en traversant les deux niveaux de la table des pages. Supposons que vous vouliez transformer l'adresse linéaire 11111678h en adresse physique, et que CR3 contienne la valeur 8000h. On décompose d'abord l'adresse linéaire 32 bits 11111678h en trois champs :

31 — 22	21 — 12	11 — 0
0001000100	0100010001	011001111000

Les bits 31...22 sont utilisés pour indexer le catalogue. Les bits 21...12 indexent la table des pages de second niveau pointée par l'élément du catalogue sélectionné ci-dessus. Les bits 11...0 donnent le déplacement dans la page pointée par l'élément de la table des pages.

Pour adresser l'élément du catalogue (4 octets), on extrait l'index du catalogue (les bits 31...22 de l'adresse linéaire), puis on les décale à gauche deux fois (multiplication par 4) pour obtenir le déplacement dans le catalogue : 0001000100b = 44h décalé deux fois à gauche donne 0100010000b = 110h. Ce déplacement est ajouté à l'adresse de base physique de la page catalogue contenue dans CR3 - dans ce cas, la valeur 8000h. Ainsi, l'élément du catalogue est à l'adresse physique 8110h.

Ce sont les 20 bits de poids fort de l'élément du catalogue qui adressent la table des pages. Si l'élément du catalogue à l'adresse 8110h contient la valeur 00023021h, la base de la table des pages est 00023000h, et les

bits P et A sont égaux à 1, indiquant que la table des pages de second niveau est bien présente, et qu'on a déjà accédé à cet élément du catalogue. Les bits 21..12 de l'adresse linéaire sont utilisés comme index dans la table des pages à l'adresse 00023000h et fournissent donc l'adresse de l'élément de la table des pages. L'index de table est égal à 100010001b (111h). On le décale à gauche 2 fois pour former un déplacement d'octets valant 444h dans la table des pages. On peut alors lire l'élément de la table des pages à l'adresse physique 00023444h. S'il contient la valeur 12345021H, l'adresse du cadre de page sera 12345000h, et la page sera présente (on y aura déjà accédé).

L'adresse physique correspondant à l'adresse linéaire initiale est obtenue en additionnant les 12 bits de poids faible de l'adresse linéaire à l'adresse du cadre de page :

```
adresse du cadre de page   12345000h
déplacement                 +   678h = 011001111000b
                        -------------
adresse physique           12345678h
```

Dans cet exemple, l'adresse linéaire 11111678h est devenue l'adresse physique 12345678h en traversant les deux niveaux de la table des pages. Le registre CR3 pointait sur le catalogue des pages à l'adresse physique 8000h. L'élément du catalogue a été lu à l'adresse 8110h, et il pointait sur une table des pages à l'adresse physique 23000h. On a lu l'élément de la table des pages à l'adresse 23444h, et il pointait sur un cadre de page à l'adresse physique 12345000h. L'adresse physique finale a été formée en ajoutant le déplacement de page obtenu par l'adresse linéaire à l'adresse du cadre de page. On a abouti ainsi à une adresse physique finale de 12345678h.

Mémoire virtuelle

Le bit P est un attribut important pour la pagination associée à la mémoire virtuelle. Des pages de l'espace adressable linéaire qui sont présentes dans la mémoire physique seront marquées présentes (P = 1), avec l'adresse physique correspondante disponible à partir de la table des pages. Evidemment, les pages non présentes en mémoire physique seront marquées non présentes (P = 0). Si un programme désire accéder à ce dernier type de page, il y aura génération d'une exception du type page afin d'avertir le système d'exploitation qu'il doit charger la page manquante à partir du disque, la ranger dans l'adresse physique correspondante, puis la marquer présente avant de rendre la main au programme qui a causé l'exception.

Les bits A et D facilitent l'implantation de la mémoire virtuelle. En examinant et en effaçant périodiquement tous les bits A, le système d'exploitation peut déterminer quelles pages n'ont pas été utilisées récemment. Ces pages ont alors de bonnes chances d'être transférées sur le disque. Si le bit D est positionné à 0 lorsqu'une page est lue à partir du disque, et qu'il est encore égal à 0 lorsque la page est transférée sur le disque, on n'a pas besoin de la sauvegarder. Si le bit D est égal à 1, on doit réécrire la page, puisqu'elle a été modifiée au moins une fois depuis son dernier chargement à partir du disque.

On doit considérer un autre problème important pour les systèmes à mémoire virtuelle qui n'apparaît pas dans la structure des tables des pages. Toutes les instructions peuvent être réexécutées après une exception du type page. Une fois que le système a résolu le problème lié à cette exception, par exemple en lisant une page non présente à partir du disque et en la marquant comme *présente*, la routine de traitement de cette interruption rend la main à l'instruction en cause. Ce mécanisme est expliqué au Chapitre 6. Nous parlerons bientôt des exceptions du type page, de leurs causes et des informations disponibles pour la routine de traitement de l'exception.

Protection au niveau pagination

Les bits R/W et U/S sont un nouveau jeu d'attributs de protection. Ces attributs sont résumés au Tableau 5.3. La pagination ne reconnaît que deux niveaux de privilège.

Les niveaux de privilège 0, 1 et 2 sont regroupés pour former le niveau de privilège superviseur. Le niveau de privilège 3 est appelé le niveau de privilège utilisateur. Au niveau utilisateur, les pages peuvent être du type lecture/exécution ou lecture/écriture/exécution. Au niveau superviseur, elles sont toujours du type lecture/écriture/exécution, mais aucun accès du niveau utilisateur n'est permis. Comme pour la segmentation, un programme s'exécutant au niveau de privilège utilisateur ne peut accéder qu'à des pages de niveau utilisateur, mais s'il s'exécute à un des niveaux superviseur (0, 1, ou 2), il pourra accéder aussi bien aux pages de niveau utilisateur qu'aux pages de niveau superviseur. Au contraire de la segmentation, un programme s'exécutant à un niveau interne superviseur peut lire/écrire/exécuter toute page, même celles qui sont du type lecture/exécution au niveau utilisateur. Ceci permet de restreindre l'accès d'une page au niveau utilisateur mais autorise des accès lecture/-écriture/exécution au superviseur.

Puisque la pagination opère après la segmentation dans le mécanisme de transformation des adresses du 80386, la protection au niveau des pages intervient après la protection apportée par la segmentation. D'abord, le système teste toutes les protections au niveau segment. S'il n'y a pas de problème, on passe aux tests de la protection au niveau pagination. Par exemple, un programme qui s'exécute au niveau de privilège 3 peut accéder à un octet de mémoire faisant partie d'un segment accessible au niveau 3, et d'une page de niveau utilisateur. On peut écrire dans cet octet seulement si la segmentation *et* la pagination autorisent l'écriture. Si le segment est du type lecture/écriture, mais que la page est du type lecture/exécution, on n'aura pas accès à l'écriture (de même si le segment est du type lecture/exécution, quelle que soit la protection associée à la pagination).

U/S	R/W	ACCES UTILISATEUR	ACCES SUPERVISEUR
0	0	Aucun	Lecture/écriture/exécution
0	1	Aucun	Lecture/écriture/exécution
1	0	Lecture/exécution	Lecture/écriture/exécution
1	1	Lecture/écriture/exécution	Lecture/écriture/exécution

Tableau 5.3 : *Attributs de protection au niveau pagination.*

Les attributs de protection d'une page sont calculés à partir des attributs du catalogue et de la page. Les bits U/S et R/W d'un élément d'une table des pages s'appliquent à la page associée à cet élément. Les bits U/S et R/W d'un élément d'un catalogue s'appliquent à la totalité du Ko de pages associées à cet élément. Le Tableau 5.4 définit les attributs combinés à partir des attributs des deux niveaux de tables des pages. Ces attributs combinés sont obtenus en effectuant un ET entre les attributs de deux niveaux et sont donc plus restrictifs. Par exemple, supposez que l'élément du catalogue ait U/S = 1 et R/W = 1 (page utilisateur du type lecture/écriture/exécution), mais que l'élément de la table des pages ait U/S = 1 et R/W = 0 (page utilisateur du type lecture/exécution). Dans ce cas, les attributs de protection combinés seront U/S = 1 et R/W = 0 (page utilisateur du type lecture/exécution).

Modification des éléments d'une table des pages

Dans ce paragraphe, nous allons donner des règles de programmation que devra suivre le programmeur de systèmes d'exploitation lors des modifications des éléments des tables des pages. L'emploi de caches de pagination implique que tous les systèmes suivent un même protocole. De plus, des systèmes multiprocesseurs dans lesquels un processeur peut changer les éléments d'une table des pages d'un autre processeur doivent suivre des règles supplémentaires.

U/S CATALOGUE	U/S PAGE	U/S COMBINES
0	0	0
0	1	0
1	0	0
1	1	1

R/W CATALOGUE	R/W PAGE	R/W COMBINES
0	0	0
0	1	0
1	0	0
1	1	1

Tableau 5.4 : *Attributs combinés de protection au niveau page.*

Cohérence du cache de pagination

Pour augmenter la vitesse en évitant d'accéder aux tables des pages qui résident en mémoire, les dernières transformations d'adresses linéaires en adresses physiques sont rangées dans un cache de pagination à l'intérieur du processeur. Ce dernier consulte ce cache avant chaque référence à des tables des pages. S'il n'y trouve pas le renseignement, alors il se reporte aux tables des pages à deux niveaux. Ce cache de pagination est équivalent aux registres cachés des descripteurs (décrits précédemment dans ce chapitre), qui sont employés pour accélérer la segmentation. Sur d'autres ordinateurs, on emploie souvent le terme de *Translation Lookaside Buffer*, ou TLB.

La cohérence entre les données du cache de pagination et les données des tables des pages n'est pas assurée par le 80386, mais elle doit être garantie par le système d'exploitation. En effet, le processeur ne sait pas si les tables des pages ont été modifiées par le logiciel, par exemple pour changer l'adresse de base d'une table ou pour marquer une page comme non présente. Dans un système bien conçu, seul le système d'exploitation peut modifier les tables des pages, et celui-ci peut assurer efficacement la cohérence en purgeant le cache après chaque modification logicielle des tables des pages. Cela s'effectue simplement en chargeant le registre de contrôle du processeur CR3, dont nous parlerons ultérieurement. La séquence de code suivante recharge CR3 avec sa valeur courante :

```
MOV   EAX,CR3    ;EAX <- CR3
MOV   CR3,EAX    ;CR3 <- EAX pour RAZ cache
```

Un cas particulier important de modification d'un élément d'une table des pages ne demande pas que le cache soit vidé. C'est lorsqu'une partie d'un élément non présent est modifiée, même si le bit P passe de 0 à 1 pour noter que l'élément est valide pour la pagination. Puisque les éléments non valides ne sont jamais dans le cache, on n'a pas besoin de purger ce cache lorsqu'on en modifie un. Cela signifie aussi que vous n'avez pas besoin de vider le cache après avoir lu une page du disque pour la rendre présente.

Multiprocesseurs

Dans un système à multiprocesseurs, on doit veiller à ce qu'un programme qui s'exécute sur un processeur ne modifie pas une table des pages à laquelle un second processeur peut accéder simultanément. Le processeur 80386 résout ce problème en utilisant des cycles du type lecture/modification/écriture indivisibles chaque fois qu'il met à jour un élément d'une table des pages pour positionner les bits A ou D. Un logiciel qui modifie la table des pages travaillera correctement s'il comporte des préfixes d'instruction LOCK (voir le Chapitre 3). Ceci permet d'assurer

l'indivisibilité des cycles lecture/modification/écriture pour des instructions qui modifient la table des pages. Avant de modifier un élément de la table des pages qui pourrait être utilisé par un autre processeur, le logiciel devra employer un LOCK AND pour effacer le bit P (P = 0). Puis, il modifiera l'élément et le rendra disponible en positionnant P (P = 1).

A un certain stade de la modification d'un élément de la table des pages, tous les processeurs du système susceptibles d'avoir cet élément dans leur cache devront savoir (habituellement par une interruption) qu'ils doivent purger leurs caches de pagination pour effacer les anciennes valeurs de l'élément. Tant que ce n'est pas fait, ces processeurs continuent d'accéder à l'ancienne page, et peuvent aussi positionner le bit D dans l'élément qui a été modifié. Même si cela peut ne pas modifier l'élément, les caches de pagination doivent être purgés avant toute modification de l'élément.

SEGMENTS SYSTEME ET REGISTRES DE CONTROLE DU PROCESSEUR

Dans cette partie, nous allons décrire les registres et les segments mémoire qui contrôlent le fonctionnement de la pagination et de la segmentation. Les registres contiennent les adresses de base des tables de segmentation et de pagination, et certains bits qui contrôlent le fonctionnement du processeur. Seuls les programmes ayant le niveau de privilège 0 (le plus interne) peuvent accéder à ces registres. Si un programme d'un niveau de privilège externe tente d'écrire dans ces registres, le processeur génère une exception pour que le noyau du système d'exploitation puisse prendre les mesures nécessaires.

La Figure 5.19 illustre les registres de contrôle du 80386. Il y a quatre registres de contrôle 32 bits : CR0, CR1, CR2, et CR3. Il y a aussi deux registres 48 bits, GDTR et IDTR, et deux registres 16 bits sélecteurs, LDTR et TR. La Figure 5.19 montre aussi les registres cachés des descripteurs associés à LDTR et TR, qui sont dessinés en pointillé pour indiquer que le programmeur ne les voit pas.

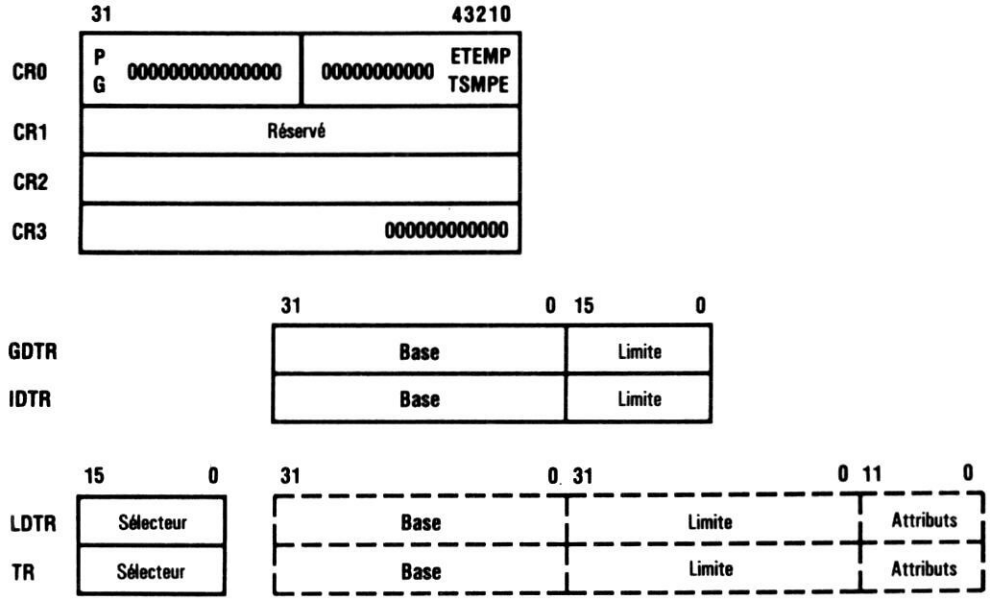


Figure 5.19 : *Registres de contrôle du processeur.*

Registres de contrôle du processeur

CR0, CR1, CR2, et CR3 sont les quatre registres de contrôle 32 bits. CR1 est réservé aux futurs processeurs et n'est donc pas défini pour le 80386. Employer une instruction qui traite CR1 comme un registre engendrera une exception du type CodeOp non valide (voir le Chapitre 6). CR0 contient des bits qui activent ou non la pagination et la protection, et des bits qui contrôlent le fonctionnement du coprocesseur arithmétique. CR2 et CR3 sont employés pour la pagination. Les bits 30 à 5 de CR0 et les bits 11 à 0 de CR3 sont réservés et doivent être chargés avec des 0.

Seuls des programmes au niveau de privilège 0 peuvent accéder aux registres de contrôle du processeur, en utilisant des formes particulières de l'instruction MOV (voir le Chapitre 3).

CR0 - Bits de contrôle du coprocesseur

Quatre bits de CR0 (ET, TS, EM et MP) contrôlent le fonctionnement du coprocesseur à virgule flottante 80387. Le bit ET sélectionne le protocole de communication. Les bits TS, MP, et EM déterminent si des instructions à virgule flottante ou une instruction WAIT doivent générer une exception du type *unité non disponible* (Device Not Available) (DNA) (voir le Chapitre 6). L'exemple de routine de traitement des exceptions du coprocesseur du Chapitre 7 montre comment on peut employer l'exception du type DNA pour sauver et récupérer les registres à virgule flottante uniquement pour des tâches employant l'arithmétique sur des nombres à virgule flottante. Ceci permet de simplifier le traitement de la commutation des tâches qui n'utilisent pas l'arithmétique à virgule flottante.

ET Le bit *Type d'extension* contrôle le protocole employé pour transmettre des instructions à virgule flottante au coprocesseur. ET = 1 indique la présence d'un coprocesseur 80387, donc l'emploi du protocole 32 bits qui donne les meilleures performances. ET = 0 indique l'emploi d'un protocole 16 bits pour communiquer avec un processeur 80287. Le processeur ignore ce champ si le bit EM est égal à 1.

TS Le bit *Commutation de tâches* (Task-Switched) est employé pour accélérer les commutations de tâches en autorisant la sauvegarde des registres du coprocesseur uniquement quand cela est nécessaire. Le processeur met TS à 1 chaque fois qu'il y a commutation de tâches. Une instruction à virgule flottante générera une exception du type DNA si TS = 1. L'instruction WAIT générera une exception si TS = 1 et MP = 1.

MP Le bit *Math présent* contrôle la génération d'une exception du type DNA par l'instruction WAIT si TS = 1. Si MP = 1, une instruction

WAIT avec TS = 1, génère une exception. Si MP = 0, l'instruction WAIT ignore le bit TS.

EM Le bit *Emulation* contrôle si les instructions à virgule flottante génèrent une exception du type DNA (EM = 1) ou sont transmises au coprocesseur (EM = 0). Notez que l'instruction WAIT ignore la valeur du bit EM.

Le Tableau 5.5 rappelle l'emploi des bits EM, TS et MP pour la génération d'une exception du type DNA. Les instructions WAIT utilisent le bit MP pour valider le bit TS et ne sont pas affectées par le bit EM. Les instructions à virgule flottante ignorent le bit MP, mais sont affectées par le bit EM.

Le bit ET est initialisé lors de l'initialisation du processeur lui-même pour indiquer quel type de coprocesseur arithmétique est employé dans le système. Si c'est un 80387, le bit ET est mis à 1. Sinon, si c'est un 80287 ou s'il n'y en a pas, ET est mis à 0. Reportez-vous aux documentations

BITS DE CR0			TYPE D'INSTRUCTIONS	
EM	TS	MP	VIRGULE FLOTTANTE	WAIT
0	0	0	Exécution	Exécution
0	0	1	Exécution	Exécution
0	1	0	Exception DNA	Exécution
0	1	1	Exception DNA	Exception DNA
1	0	0	Exception DNA	Exécution
1	0	1	Exception DNA	Exécution
1	1	0	Exception DNA	Exécution
1	1	1	Exception DNA	Exception DNA

Tableau 5.5 : *Résumé des bits EM, TS et MP.*

Intel sur le 80386, le 80387 et le 80287 si vous désirez des informations complémentaires sur l'initialisation du matériel.

CR0 - Bits de contrôle de la protection

Le bit PE (bit 0) et le bit PG (bit 31) contrôlent le fonctionnement de la segmentation et de la pagination. PE contrôle la segmentation. Si PE = 1, le processeur travaille avec la segmentation. Lorsque PE = 1, on dit que le processeur travaille en mode protégé. Si PE = 0, la segmentation n'est pas activée et le processeur travaille en mode réel comme un 8086 (voir le Chapitre 9). PG contrôle la pagination. Si PG = 1, le processeur travaille avec la pagination. Si PG = 0, la pagination n'est pas activée et les adresses linéaires produites par la segmentation deviennent les adresses physiques.

Le Tableau 5.6 résume les modes de fonctionnement du processeur qu'on peut sélectionner à partir des bits PE et PG. Notez que seules trois des quatre combinaisons possibles sont autorisées. Charger CR0 avec PG = 1 et PE = 0 générera une exception du type protection.

Faites attention lorsque vous modifiez les bits PG et PE. On doit modifier le bit PG uniquement lorsque s'exécute un programme qui a son code et au moins certaines de ses données dans des pages qui ont les

PG	PE	MODE D'EXECUTION
0	0	Mode Réel (voir le Chapitre 9)
0	1	Mode Protégé,pas de pagination
1	0	Ne pas utiliser
1	1	Mode Protégé + Pagination

Tableau 5.6 : *Modes de fonctionnement du processeur sélectionnés par les bits PG et PE.*

mêmes adresses dans l'espace adressable linéaire et dans l'espace adressable physique (voir le Chapitre 7). Ce code apporte une passerelle entre les mondes paginés et non paginés. De plus, le cache de pagination doit être purgé avant de positionner PG à 1.

Le programme doit exécuter une instruction de saut immédiatement après avoir changé la valeur du bit PE afin de vider le tampon d'instructions (*pipeline*). Ceci afin d'éviter qu'une instruction ne soit traitée dans le mauvais mode. Avant de mettre le bit PE à 1, le programme doit initialiser les segments système et les registres de contrôle. Le processeur est initialisé avec PE = 0 et PG = 0 (mode réel) pour pouvoir exécuter du code qui initialise les registres et les structures de données nécessaires à la pagination et à la segmentation avant que ces mécanismes soient valides. Reportez-vous au Chapitre 7 pour avoir un exemple d'initialisation du processeur, et au Chapitre 9 pour plus d'informations sur le mode réel.

CR2 et CR3

La pagination utilise CR2 et CR3. CR3 contient l'adresse physique de la page contenant le premier niveau de la table des pages, le catalogue. Puisque le catalogue est aligné sur une page, seuls les 20 bits supérieurs de ce registre sont significatifs. Les 12 bits restants sont réservés aux processeurs futurs. Ils doivent être égaux à 0 lorsqu'on charge une nouvelle valeur dans CR3, sinon on les ignore.

Charger CR3 par l'instruction *MOV CR3, reg* (voir le Chapitre 3) a pour conséquence désagréable d'invalider le cache de pagination. On peut charger CR3 même si le bit PG de CR0 n'est pas positionné. Cela permet d'initialiser la pagination. Les commutations de tâches modifient aussi CR3, mais si la nouvelle tâche a le même CR3 que l'ancienne, le processeur n'a pas besoin de purger le cache de pagination. Ceci accélère l'exécution lorsque des tâches se partagent des tables des pages.

CR2 donne des informations sur le type d'erreur lorsque survient une exception du type page. Le processeur range dans CR2 l'adresse linéaire

qui a causé l'exception. La routine de traitement des exceptions du système d'exploitation peut alors examiner le contenu de CR2 afin de déterminer quelle est la page de l'espace adressable linéaire qui a causé l'exception.

Registres de base des tables de segmentation

GDTR, IDTR, LDTR et TR sont les registres de base des segments qui contiennent les tables importantes de la segmentation. GDTR, IDTR et LDTR adressent des segments qui contiennent les tables des descripteurs, et TR adresse un segment particulier, le *segment d'état de la tâche* (Task State Segment) (TSS), qui contient des informations importantes sur la tâche courante.

Le GDTR pointe sur la GDT. Le GDTR 48 bits définit la base et la limite de la GDT, directement par une adresse linéaire 32 bits et une limite 16 bits. Le GDTR permet de donner une "origine" à la structure de la table des segments. Les descripteurs pour la LDT et pour la tâche courante TSS sont stockés dans la GDT, si bien qu'on peut identifier ces segments système par des sélecteurs, exactement comme les autres segments. Puisqu'un descripteur de la GDT ne peut pas définir la GDT, le GDTR offre un pseudo-descripteur pour ce segment système particulier afin de donner une origine à la structure de table des segments de l'espace adressable linéaire.

L'IDTR contient un pseudo-descripteur 48 bits pour la *table des descripteurs d'interruptions* (Interrupt Descriptor Table) (IDT). L'IDT prend son origine directement dans l'espace des adresses linéaires grâce à un pseudo-descripteur plutôt qu'à un descripteur de la GDT, ceci afin de gagner un niveau d'indirection lors de l'accès à des descripteurs d'interruptions. L'IDT et les types de descripteurs spécifiques à l'IDT seront détaillés au Chapitre 6.

La Figure 5.20 donne le format d'un pseudo-descripteur 48 bits. Les 32 bits de poids fort contiennent l'adresse de base du segment dans l'espace adressable linéaire. Les 16 bits de poids faible contiennent le

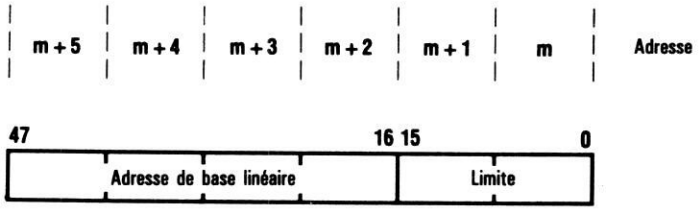


Figure 5.20 : *Format d'un pseudo-descripteur.*

champ limite, qui est une limite (granulité d'octet) donnant la taille de la table (1 octet à 64 Ko). Puisque les descripteurs sont constitués de huit octets, le système d'exploitation doit s'assurer que le champ limite d'une table contenant N descripteurs a bien la valeur 8*N - 1.

Le registre LDTR contient le sélecteur pour la LDT de la tâche courante. Ce sélecteur doit identifier un segment du type LDT associé à la GDT. C'est-à-dire que le bit TI de ce sélecteur doit être égal à 0 pour indiquer que le descripteur associé est stocké dans la GDT, et en outre le champ type de ce descripteur doit indiquer un segment du type LDT.

Il est possible de charger LDTR avec un sélecteur nul pour indiquer que la tâche courante n'a pas de LDT. Dans ce cas, une exception survient si une adresse est générée avec un sélecteur qui a le bit TI égal à 1, indiquant qu'il est rangé dans la LDT. Si LDTR est chargé avec un sélecteur nul, toutes les adresses doivent employer des sélecteurs pour des segments associés à la GDT.

Notez que tous ces segments système sont définis comme des régions contiguës de l'espace adressable linéaire, exactement comme tous les autres segments. Parce qu'elles sont dans l'espace adressable linéaire, ces tables système peuvent être relogées par la pagination. Par conséquent, dans un système à mémoire virtuelle comprenant plusieurs tâches, les segments LDT et TSS des tâches non actives peuvent être placés sur le disque pour être rappelés à la demande. Puisqu'on fait référence à l'IDT et la GDT à chaque tâche, ces segments système doivent toujours se trouver en mémoire physique.

Aucun programme ne peut faire référence directement à ces segments système, pas même le noyau du système d'exploitation au niveau 0. Les descripteurs particuliers qui sont utilisés pour les segments LDT et TSS causeraient une erreur si on tentait de charger dans un des six registres segments un sélecteur qui les identifie. Les segments GDT et IDT n'ont même pas de sélecteurs ou de descripteurs de segments ! Pour permettre au noyau du système d'exploitation d'examiner et de modifier le contenu de ces segments, on doit définir des segments (du type lecture ou lecture/écriture) aux mêmes adresses de base et limites, mais avec des attributs qui autorisent le chargement dans un des registres DS, ES, FS, ou GS. On appelle ces segments supplémentaires des *alias*, puisqu'ils donnent un nom différent à la même zone de l'espace adressable linéaire.

Format du segment d'état de la tâche

Le segment d'état de la tâche, ou TSS, est un segment particulier qui contient des informations importantes sur cette tâche. Le registre TR adresse le TSS d'une tâche active. Le TSS d'une tâche non active contient une image "figée" de la tâche. Le TSS permet la suspension d'une tâche puisqu'il contient une image complète du registre d'état de la tâche. Lorsqu'une tâche est suspendue, les valeurs courantes des registres du processeur sont sauvegardées dans les champs du TSS. Lorsqu'on reprend l'exécution de la tâche, les registres sont rechargés à partir des valeurs sauvegardées dans le TSS pour rétablir l'état de la tâche, et permettre ainsi de poursuivre l'exécution comme si rien ne s'était passé. La Figure 5.21

31	0		
Déplacement de la table d'autorisations E/S	000000000000000	T	64h
0000000000000000	LDT		60h
0000000000000000	GS		5Ch
0000000000000000	FS		58h
0000000000000000	DS		54h
0000000000000000	SS		50h
0000000000000000	CS		4Ch
0000000000000000	ES		48h
EDI			44h
ESI			40h
EBP			3Ch
ESP			38h
EBX			34h
EDX			30h
ECX			2Ch
EAX			28h
EFLAGS			24h
EIP			20h
CR3			1Ch
0000000000000000	SS2		18h
ESP2			14h
0000000000000000	SS1		10h
ESP1			0Ch
0000000000000000	SS0		8
ESP0			4
0000000000000000	LIEN		0

Figure 5.21 : *Format du segment d'état d'une tâche.*

donne le format d'un TSS. Ce dernier est composé de cinq types différents d'information :

- Champ de liens
- Pointeurs de piles de niveaux internes
- Registres de base d'adressage
- Zone de sauvegarde des registres
- Champs divers

Le processeur 80386 définit le format des 104 premiers octets d'un TSS. Le système d'exploitation peut sauvegarder d'autres informations sur la tâche (par exemple environnement matériel). Comme on l'a vu, on doit définir un segment mémoire du type alias pour le TSS pour permettre au système d'exploitation de lire ou d'écrire les champs du TSS.

Puisque le TSS est stocké dans l'espace adressable linéaire, la pagination peut le reloger n'importe où et même sur le disque, en mémoire virtuelle. L'unique contrainte est qu'il ne peut y avoir génération d'une exception du type page au cours d'une commutation de tâches. Une exception du type page peut être traitée à la première référence à une page provenant du disque, si bien que des astuces simples permettent de passer outre cette restriction. Evidemment, on peut par exemple ne jamais marquer un TSS présent. Une autre possibilité consiste à s'assurer que si un TSS est chargé à partir du disque, le TSS complet est soit présent, soit non présent (aucun TSS ne doit déborder d'une frontière de page dans ce cas). Cela peut se faire facilement en alignant chaque TSS sur une frontière de 128 octets de l'espace adressable linéaire, puisque le processeur ne fait référence qu'aux seuls 104 premiers octets d'un TSS lors d'une commutation de tâches. Si la base d'un TSS est alignée sur une frontière de 128 octets, les 104 premiers octets ne peuvent déborder d'une frontière de page (4 Ko).

Champ de liens

Le champ de liens est situé au déplacement 0 dans le TSS, et est constitué de 32 bits dont un sélecteur dans les 16 bits de poids faible (les 16 bits de poids fort étant indéfinis). On l'utilise conjointement avec le bit NT du registre EFLAGS pour lier entre eux les TSS des tâches suspendues par des instructions CALL ou par des interruptions. On peut suspendre une tâche soit par une instruction CALL qui fait référence à une tâche (voir le paragraphe sur la commutation des tâches dans ce chapitre), soit par une interruption qui demande une commutation de tâches (voir le Chapitre 6). Si la tâche courante a été activée par une instruction CALL ou une interruption, le champ de liens de son TSS contient le sélecteur du TSS de la tâche suspendue, et le bit NT est mis à 1 pour indiquer que le champ de liens est valide. Une instruction IRET exécutée avec le bit NT du registre EFLAGS égal à 1 suivra pour rendre la main à la tâche précédente dans la chaîne (voir le Chapitre 6).

La Figure 5.22 montre une chaîne de tâches reliées entre elles. Le TSS de la tâche courante (TSS_C) est adressé par le registre TR. Les TSS des tâches suspendues (TSS_A et TSS_B) sont liés par le champ de liens. Le bit NT du registre EFLAGS sera égal à 1 lorsque la tâche C s'exécutera pour indiquer qu'il y a imbrication de tâches. De même, le bit NT de l'image du registre EFLAGS dans TSS_B sera égal à 1 pour indiquer que la tâche B est imbriquée dans une autre tâche. Le bit NT dans TSS_A sera égal à 0 pour indiquer que la tâche A n'est pas imbriquée dans une autre tâche. Notez que NT = 0 signifie que la tâche A correspond à la fin de cette chaîne, et que le champ de liens de TSS_A n'est pas référencé.

La chaîne de TSS liés (voir la Figure 5.22) a été construite par une série de commutations de tâches imbriquées causées par des instructions CALL ou des interruptions. Un CALL ou une interruption reçue pendant l'exécution de la tâche A entraîne une commutation sur la tâche B. Ainsi, la tâche A est suspendue, la tâche B est activée, et le champ de liens de la tâche B pointe sur TSS_A. Lors de l'exécution de la tâche B, un autre CALL ou une autre interruption a causé une commutation sur la tâche C, entraînant la suspension de la tâche B, activant la tâche C et faisant pointer le champ de liens de TSS_C sur TSS_B. La Figure 5.22 montre le résultat de cet enchaînement.

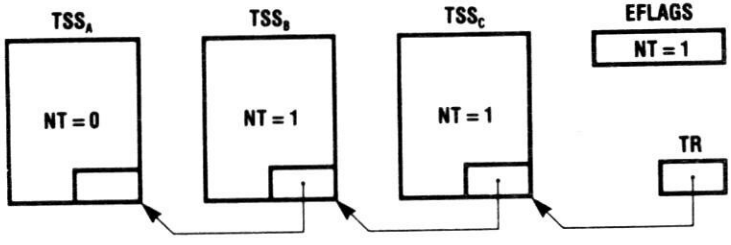


Figure 5.22 : *Chaîne de TSS liés.*

Pointeurs de pile de niveaux internes

Dans les paragraphes précédents, nous avons expliqué que chaque niveau de privilège possédait une pile distincte afin d'éviter tout problème de protection lié à une pile commune. Lorsqu'un changement de niveau s'effectue vers un niveau interne, la pile associée à ce niveau interne est initialisée en chargeant le pointeur de pile approprié dans cette zone du TSS. Le pointeur de pile du niveau externe est empilé dans la pile interne si bien que la pile externe peut être restituée au retour. Dans le paragraphe sur les méthodes de transfert de contrôle, nous expliquerons les changements de niveau de privilège et les commutations de piles associées.

Les pointeurs de pile sont simplement des pointeurs 48 bits sur les sommets des piles de niveaux 0, 1 et 2. Ils sont rangés aux déplacements 4, 12 et 20 dans le TSS. Lorsqu'il y a une transition vers un niveau interne, le pointeur approprié est chargé dans les registres SS et ESP pour commuter sur la pile interne. Il n'y a pas de pointeur de pile pour le niveau 3, le plus

externe, puisqu'il est impossible d'entrer dans ce niveau en allant à un niveau interne ! Si une tâche est suspendue au niveau 3, le pointeur de la pile associée sera contenu dans les images des registres SS et ESP, qui sont sauvegardés à un autre endroit du TSS.

Le processeur 80386 lit dans cette zone les pointeurs de pile internes, mais n'y écrit jamais. Cela signifie que les transitions vers des niveaux internes initialisent toujours la pile interne au même endroit, commençant toujours avec une pile "fraîche" à chaque transition interne. Ceci est correct puisqu'il n'est pas possible d'avoir des transitions récursives. Lorsqu'il y a une transition vers un niveau interne, le seul moyen de revenir au niveau externe est de rencontrer un retour au niveau externe, qui dépile la pile de niveau interne pour lui redonner sa configuration initiale.

Registres de base d'adressage

Lorsqu'il y a commutation de tâches, les registres LDTR et CR3 sont chargés à partir des champs du TSS de la nouvelle tâche. Le registre LDTR est chargé avec un sélecteur situé au déplacement 60h dans le TSS, et CR3 est chargé avec le dmot situé au déplacement 1Ch.

Charger CR3 avec une valeur nouvelle remplace la table des pages par celle de la nouvelle tâche, ce qui modifie la fonction de transformation des adresses linéaires en adresses physiques. De même, charger le registre LDTR remplace la LDT par celle de la nouvelle tâche, ce qui change la fonction de mapping (adresses virtuelles en adresses linéaires) de la moitié de l'espace adressable virtuel associé à la LDT. Comme on l'a déjà vu, cette possibilité de changer la fonction de transformation des adresses entre les tâches fait partie du mécanisme de protection qui isole les tâches les unes des autres.

Notez que le processeur 80386 lit ces champs dans le TSS afin de charger les registres appropriés lors d'une commutation de tâches, mais qu'il n'écrit jamais dans ces champs, même lorsqu'il y a commutation de tâches. Ainsi, si un programme modifie soit LDTR, soit CR3, il doit aussi ranger la nouvelle valeur dans le TSS de la tâche courante.

Zone de sauvegarde des registres

Cette zone du TSS est une zone de sauvegarde des registres généraux, des registres de contrôle du processeur, et des registres segments. Lorsque la tâche décrite par ce TSS est la tâche courante, ces champs sont indéfinis. Lorsqu'il y a commutation de tâches, les valeurs courantes de ces registres sont sauvegardées dans cette zone, si bien qu'à la prochaine commutation sur l'ancienne tâche le système pourra récupérer dans cette zone les valeurs des registres pour poursuivre l'exécution.

La zone de sauvegarde des registres occupe les déplacements 20h à 5Fh (voir la Figure 5.21). Les registres segments sont sauvegardés dans les champs 32 bits dont les 16 bits de poids fort sont indéfinis, et le sélecteur est sauvegardé dans les 16 bits de poids faible.

Champs divers

Le déplacement dans le TSS de la table (bitmap) d'autorisations E/S est stocké dans le mot situé au déplacement 66h. La table (bitmap) d'autorisations E/S est stockée dans le TSS, et définit les adresses E/S auxquelles la tâche peut accéder.

Le mot situé au déplacement 64h est destiné à des attributs particuliers de la tâche. Dans le 80386, un seul attribut est défini, l'attribut de *trappe de mise au point* (Debug Trap), qui est stocké dans le bit de poids faible de ce mot. Les bits restants doivent être positionnés à 0 pour assurer la compatibilité avec les futurs processeurs. S'il y a commutation dans une tâche dont ce bit est égal à 1, une trappe de mise au point est prise en considération lorsque la commutation de tâches est terminée, mais avant que la première instruction de la nouvelle tâche soit exécutée. Ce bit permet au logiciel de partager efficacement les registres de mise au point entre les tâches, sans alourdir la commutation de tâches classique. Dans le Chapitre 8, nous parlerons des trappes et des registres de mise au point.

INSTRUCTIONS SENSIBLES AU NIVEAU DE PRIVILEGE

Certaines instructions du 80386 s'exécutent différemment selon le niveau de privilège du programme qui les contient. Les instructions *privilégiées* ne peuvent être exécutées qu'au niveau de privilège 0, et elles généreront une exception si on tente de les exécuter à un autre niveau de privilège. On ne peut exécuter les instructions *sensibles aux E/S* qu'au niveau de privilège correspondant au champ IOPL du registre EFLAGS (ou à un niveau plus interne) (voir le Chapitre 2), ou si elles accèdent à des adresses de l'espace E/S marquées accessibles à la tâche courante par la table (bitmap) d'autorisations E/S (une structure de données particulières du TSS). Finalement, les instructions qui modifient le registre EFLAGS ne changeront pas les valeurs de certains champs à moins qu'elles ne soient exécutées au niveau de privilège 0 ou, pour d'autres champs, au niveau de privilège relatif à IOPL(ou à un niveau plus interne) .

Instructions privilégiées

Les instructions qui accèdent aux registres de protection sont des instructions privilégiées. Elles n'autorisent l'accès à ces registres qu'aux programmes qui s'exécutent au niveau 0. On doit protéger ces registres de tout accès non autorisé afin d'assurer l'intégrité du modèle de protection. Le Tableau 5.7 donne une liste des instructions privilégiées du 80386. Notez que les instructions qui *chargent* les GDTR, IDTR, LDTR, TR et MSW (les 16 bits de poids faible de CR0) sont privilégiées contrairement aux instructions qui *rangent* ces mêmes registres dans la mémoire. Cela signifie que tout programme peut stocker ces registres, mais que seuls des programmes de niveau 0 peuvent les modifier. Au contraire, les

Mnémonique	Fonction
CLTS	Efface le bit TS dans CR0
HLT	Halte
LGDT	Charge GDTR
LIDT	Charge IDTR
LLDT	Charge LDTR
LMSW	Charge MSW
LTR	Charge TR
MOV CRn,reg	Charge le registre de contrôle n
MOV reg,CRn	Range le registre de contrôle n
MOV DRn,reg	Charge le registre de mise au point n
MOV reg,DRn	Range le registre de mise au point n

Tableau 5.7 : *Instructions privilégiées.*

instructions qui rangent en mémoire les registres de contrôle et les registres de mise au point (voir le Chapitre 8) sont privilégiées, de même que les instructions qui les chargent.

Protection de l'espace des E/S

Deux mécanismes contrôlent l'accès à l'espace adressable E/S et aux instructions relatives aux E/S :

1 Le champ IOPL du registre EFLAGS.

2 La table (bitmap) d'autorisations E/S dans le TSS.

Le champ IOPL du registre EFLAGS définit le niveau de privilège le plus externe qui permet d'exécuter toutes les instructions relatives aux E/S et d'accéder à toutes les adresses de l'espace E/S. La table (bitmap)

Mnémonique	Fonction
CLI	Efface le bit IF de EFLAGS
STI	Positionne le bit IF de EFLAGS
IN	Lit une donnée sur l'adresse E/S
INS	Lit une chaîne sur l'adresse E/S
OUT	Ecrit une donnée sur l'adresse E/S
OUTS	Ecrit une chaîne sur l'adresse E/S

Tableau 5.8 : *Instructions sensibles aux E/S.*

d'autorisations E/S contenue dans le TSS définit les adresses auxquelles un programme s'exécutant à tout niveau de privilège peut accéder.

Les programmes qui s'exécutent au niveau de privilège E/S (IOPL) ou à un niveau interne peuvent exécuter toutes les instructions du Tableau 5.8. Les instructions CLI et STI généreront une exception si elles sont exécutées à un niveau externe. Les instructions IN, INS, OUT et OUTS généreront une exception à des niveaux externes uniquement si l'adresse E/S est marquée inaccessible dans la table (bitmap) d'autorisations E/S. Si une de ces instructions est exécutée à un niveau externe relativement au IOPL, il y a appel à la table (bitmap) d'autorisations E/S avant toute génération d'exception. Si l'accès aux adresses E/S particulières demandées par l'instruction est permis, l'instruction s'effectue normalement, sinon il y a génération d'une exception.

Puisque chaque tâche possède son propre TSS et son propre registre EFLAGS, elle peut avoir un IOPL différent et définir une table (bitmap) d'autorisations E/S différente. Par exemple, une tâche exécutant un programme de jeu peut avoir une table qui autorise l'accès au joystick. Une tâche exécutant un programme de communications peut avoir une table qui autorise l'accès au réseau.

Table d'autorisations E/S

La table d'autorisations E/S définit les adresses de l'espace E/S de 64 Ko auxquelles des programmes de n'importe quel niveau de privilège peuvent accéder. Une chaîne binaire de 64 Ko est donc stockée dans le TSS courant. Chaque bit de la chaîne binaire correspond à l'adresse E/S d'un octet unique. Le bit 0 correspond à l'adresse E/S 0, le bit 1 à l'adresse 1, etc. Dans la table, un 0 signifie que l'adresse E/S correspondante est accessible aux programmes de tout niveau de privilège. Un 1 indique que seuls les programmes qui s'exécutent à IOPL (ou à un niveau interne) peuvent accéder à l'adresse E/S correspondante. Une exception sera générée si un programme tente d'accéder à une adresse E/S correspondante à un 1 dans la table alors qu'il s'exécute à un privilège externe relativement à IOPL.

Puisque l'espace E/S est adressable par octets et qu'une instruction peut accéder jusqu'à quatre octets, on doit donc tester chaque fois tous les bits des adresses auxquelles l'instruction fait référence. Si tous les bits de la table sont égaux à 0, le système autorise l'accès aux E/S. Si un seul des bits est égal à 1, l'accès aux E/S générera une exception.

Ces tests multiples doivent marcher pour toutes les combinaisons de longueur et d'alignement possibles. Ceci implique qu'au moins deux octets soient lus dans la table. Pour accéder à la table aussi rapidement que possible, le 80386 lit toujours deux octets, que cela soit nécessaire ou non. Pour éviter les problèmes posés par les adresses E/S supérieures, il doit toujours y avoir un octet ne contenant que des 1 après le dernier octet de la table qui contient une information valide mais avant la limite du TSS. Ceci donne un octet de "remplissage" qui permet au processeur de lire deux octets dans la table, même au sommet de l'espace adressable E/S.

La table d'autorisations E/S peut être stockée n'importe où dans les 64 premiers Ko du TSS, et avoir n'importe quelle longueur, pourvu que ce soit un multiple de 8 bits. Le mot situé au déplacement 66h du TSS courant définit le déplacement d'origine de cette table, lequel doit être inférieur à 56 Ko Ainsi, la table peut commencer n'importe où dans les 56 premiers Ko du TSS. La fin de cette table est située soit à ce déplacement plus 8 Ko, soit à la limite du TSS. On peut définir une table complète de 8 Ko en ren-

dant la limite du TSS plus grande que le déplacement initial plus 8 Ko. On peut aussi ramener la limite du TSS au déplacement initial pour donner moins d'espace à la table. Les bits qui seront en dehors de la limite du TSS seront considérés comme des 1 et généreront des exceptions si on accède aux adresses E/S correspondantes avec un niveau de privilège externe à IOPL. On peut vider la table en rendant la limite du TSS égale au déplacement initial de la table.

Par exemple, en positionnant la limite à :

TableDépl + 32

on obtient une table pour les 256 premières adresses E/S. Il y aura génération d'une exception si on utilise une adresse E/S supérieure à 255. Ceci permet de ne pas occuper 8 Ko de mémoire quand ce n'est pas nécessaire, tout en maintenant cette possibilité.

Références à l'espace adressable E/S

La routine AccessIO() du Listing 5.1 (en pseudo-C) donne une description détaillée des références à l'espace adressable E/S. Cette description utilise la routine AccessLinear() (définie au Listing 5.4) qui lit des données dans le TSS. On fait référence au registre TR.Base pour obtenir l'adresse linéaire de la base du TSS courant. Le programme appelle la routine SegmentException() pour générer une exception du type segment si nécessaire. Reportez-vous à la description des références mémoire et des exceptions donnée ultérieurement dans ce chapitre.

La routine AccessIO() compare tout d'abord CPL avec IOPL, puisqu'on peut accéder à toutes les adresses E/S et que la table d'autorisations E/S n'est pas testée si CPL est un niveau identique ou interne à IOPL. Sinon, le test est effectué.

Le déplacement du début de la table (bitmap) est lu à partir du mot situé au déplacement 66h du TSS courant. L'emplacement des bits nécessaires est calculé à partir du déplacement de bit et d'octet de l'adresse E/S à tes-

ter. Le déplacement d'octet est l'adresse E/S décalée 3 fois à droite. Le déplacement de bit dans cet octet est l'adresse E/S modulo 8. Deux octets sont lus au déplacement d'octet indiqué dans la table (bitmap) pour assurer que tous les bits nécessaires sont lus. Si ces octets sont situés en dehors de la limite du segment, il y a génération d'une exception. Ceci permet de définir une table (bitmap) plus petite que 8 Ko, la taille normale.

Le masque de la longueur est basé sur la taille de la référence E/S. Il peut être de 1 ou 4 bits pour des références E/S à quatre octets. Le masque est décalé à gauche *n* fois (*n* représentant le déplacement de bit) pour aligner le bit de poids faible du masque sur le bit de la table (bitmap) correspondant à l'adresse E/S la plus basse. Le ET s'effectue entre le masque et les octets lus dans la table pour effacer les bits inintéressants. Si le résultat est égal à 0, l'accès E/S est permis. Si le résultat n'est pas nul, une ou plusieurs adresses ne sont pas accessibles, et il y a génération d'une exception.

```
AccessIO(IOAddress, Lenght, RW, Data)
   int    IOAddress,    /* Adresse E/S à tester pour */
                        /* accessibilité */
          Length,       /* nombre de bits à tester */
          RW,           /* 0 si lecture, 1 si écriture */
          *Data;        /* pointeur sur les données à lire */
                        /* ou à écrire*/
{
   short int     ptr2, ptrbits;
   int           BitOffset, ByteOffset, Mask, Result;

/* Si CPL IOPL, toutes les adresses E/S sont accessibles. */
/* Si CPL>IOPL, teste la table d'autorisation E/S         */
if (CPL>IOPL)    {
   /* Lit le déplacement de la table à partir */
   /*  du déplacement 66h dans TSS            */
   AccessLinear(TR.Base+66h,2,0 /* PL 0 */, 0 /* Ecriture */,
                &ptr2);
```

Listing 5.1 : *Sous-programme AccessIO().*

```
   /* Calcule le déplacement du bit et de l'octet */
   /* dans la table */
   ByteOffset = IOAddress >>3;
   BitOffset = IOAddress & 0111b;

   /* Le masque est obtenu en décalant un champ de 1, 2 */
   /* ou 4 octets à gauche de BitOffset dans la table   */
   Mask = (01111b >> (4-Length)) << BitOffset;

   /* Lecture de deux octets (1 à 4 bits) dont nous avons */
   /* besoin pour le test */
   /* Test de la limite TSS pour voir si les bits indiqués */
   /* se trouvent au-delà de la fin de la table.*/
   if (TR.Limit < ptr2+ByteOffset+1)
      SegmentException($GP, 0);
   AccessLinear (TR.Base+ptr2+ByteOffset,
                 2, 0 /* PL 0 */, 0 /* Lecture */,
                 &ptrbits);

/* Si ET(masque,bits d'autorisation) = 0 */
/* l'accès n'est pas possible, il y a génération d'une */
/* exception.*/
/* Sinon, on autorise l'accès et on retourne à l'appelant */
   Result=ptrbits & Mask;
   if (Result !=0)
      SegmentException($GP, 0);
   } /* fin CPL>IOPL */

   /* On arrive ici uniquement si l'accès E/S est permis */
   AccessPhysicalIO(IOAddress, Lenght, RW, &Data);
}
```

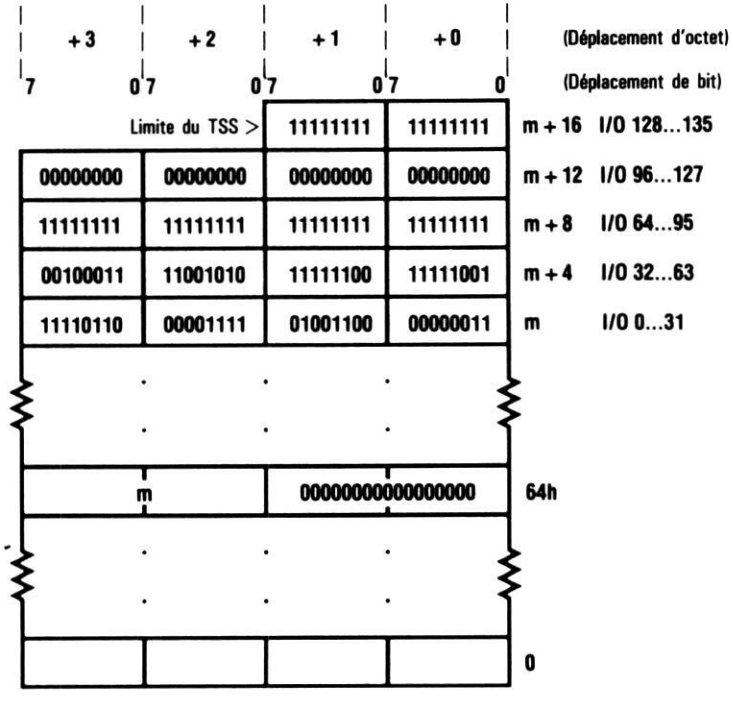
Listing 5.1 : *Sous-programme AccessIO() (fin).*

Si l'accès E/S est permis, il est pris en compte par un appel à la routine AccessPhysicalIO(). Cette routine effectue l'accès aux adresses E/S physiques. Ce livre ne la décrit pas car le fonctionnement de la mémoire est indépendant du processeur 80386. La routine reprend les mêmes paramètres que ceux de la routine AccessIO().

La table de la Figure 5.23 est utilisée dans les exemples suivants pour illustrer le fonctionnement de la table d'autorisations E/S. La table est rangée dans le TSS de la tâche courante commençant au déplacement donné par le mot situé au déplacement 66h dans TSS. Dans cet exemple, le déplacement est *m*. Les adresses E/S associées à chaque dmot sont à droite, à côté de l'adresse du dmot relative au début de la table (déplacement *m*). Les déplacements de bits dans les octets sont en haut, comme les déplacements d'octets dans les dmots. Cette table contient 17 octets plus un octet de "remplissage" à l'adresse la plus haute. Les 17 octets significatifs s'associent donc aux adresses E/S de 0 à 135.

Cette table permet d'accéder aux adresses E/S 2...9, 12...13, 15, 20...24, 27, 33...34, 40...41, 50, 52...53, 58...60, 62...63 et 96...127. Les autres adresses E/S sont accessibles uniquement par des programmes qui s'exécutent à un niveau identique à IOPL (ou plus interne). Notez que le dernier octet avant la limite de TSS est un octet de remplissage qui doit être rempli de 1.

Par exemple, supposez qu'un programme qui s'exécute à un niveau de privilège externe (relativement à IOPL) possède la table d'autorisations de la Figure 5.23. Ce programme tente d'accéder à un dmot à l'adresse E/S 7. Cela signifie que le système doit tester les bits des adresses E/S 7, 8, 9 et 10. En utilisant l'algorithme précédent, nous formons un masque qui a quatre bits égaux à 1 décalés par 7 modulo 8, soit 7. Le ET s'effectue sur ce masque et les deux octets lus dans la table au déplacement 7/8 (0). Puisque l'adresse E/S 10 n'est pas accessible (comme l'indique le bit 1 à l'emplacement 10 dans la table), le résultat du ET n'est pas nul et une exception est donc générée. Le traitement est alors le suivant :



Exemple de TSS

Figure 5.23 : *Exemple de table d'autorisations E/S.*

```
Dépl.       = Byte 0, bit 7
Masque      = (1111b >> (4-Length)) << 7
            = (1111b >> 0) << 7
            = 1111b << 7 = 78h

      BitString 0100110000000011 (voir la table)
AND   MASK      0000011110000000
      Résultat  0000010000000000 ( <> 0)

Exception !
```

Cet exemple illustre le cas où les bits à tester couvrent deux octets dans la table. Le bit associé à l'adresse E/S 7 se trouve dans l'octet situé au déplacement 0 à partir du début de la table, et les bits associés aux adresses 8, 9 et 10 dans l'octet suivant.

Pour l'autre exemple, supposez qu'un programme qui s'exécute à un niveau de privilège externe (relativement à IOPL) possède la table d'autorisations de la Figure 5.23. Ce programme tente d'accéder à un dmot à l'adresse E/S 33. Cela signifie que le système doit tester les bits des adresses E/S 33 et 34. En utilisant l'algorithme précédent, nous formons un masque qui a deux bits égaux à 1 décalés par 33 modulo 8, soit 1. Le ET s'effectue sur ce masque et les deux octets lus dans la table au déplacement 33/8 (4). Dans cet exemple, les deux bits sont égaux à 0, et donc l'accès est autorisé.

```
Dépl.       = Byte 4, bit 1
Masque      = (1111b >> (4-Length)) << 1
            = (1111b >> 2) << 1
            = 11b << 1 = 110h

      BitString 1111110011111001 (voir la table)
AND   MASK      0000000000000110
      Résultat  0000000000000000 ( <> 0)

E/S permises !
```

Instructions qui modifient EFLAGS

Certains champs du registre EFLAGS sont traités différemment selon le niveau de privilège du programme qui tente d'y accéder ou de les modifier. Les bits IF, IOPL, et VM sont traités différemment des autres champs de EFLAGS (voir le Chapitre 2). Les instructions IRET, CLI, STI et POPF (voir le Chapitre 3) permettent de modifier ces champs de EFLAGS.

		CHAMPS EFLAGS	
PRIVILEGE	**VM**	**IOPL**	**IF**
CPL=0	Modifié*	Modifié	Modifié
0<CPL IOPL	----	----	Modifié
IOPL<CPL	----	----	----

---- signifie que l'indicateur n'est pas modifié.

* VM n'est pas modifié par l'instruction POPF.

Tableau 5.9 : *Traitement particulier des champs EFLAGS.*

Seuls des programmes qui s'exécutent au niveau de privilège 0 peuvent modifier les bits VM et IOPL. Le bit IF peut être modifié uniquement par des programmes qui s'exécutent au même niveau ou à un niveau interne relativement à IOPL. Un programme à un niveau de privilège externe qui exécute une instruction POPF ou IRET et qui tente de modifier un de ces champs ne génère pas une exception. Au contraire, les champs ne sont pas modifiés, tout simplement. Le Tableau 5.9 résume le traitement de ces indicateurs particuliers.

De plus, l'instruction POPF ne modifie pas le bit VM, et l'instruction PUSHF empile toujours un 0 dans l'emplacement du bit VM. Ceci est nécessaire pour empêcher des programmes de tester le bit VM pour savoir s'ils sont en mode virtuel 8086 ou en mode réel. Dans le Chapitre 9, nous décrirons ces modes.

METHODES DE TRANSFERT DE CONTROLE

Dans ce paragraphe, nous allons décrire les méthodes employées pour effectuer des sauts intersegments, des appels à des sous-programmes, et des retours de sous-programmes entre des segments de codes différents.

Nous proposons deux façons de considérer les transferts de contrôle. Nous allons parler de la première dans ce paragraphe. Les différentes méthodes pour transférer le contrôle incluent un transfert direct vers un déplacement d'un autre segment de code, un transfert à travers un guichet vers le point d'entrée d'un autre segment, ou un transfert à travers un guichet de tâche vers une autre tâche. Nous allons introduire les concepts nécessaires à la compréhension des différentes manières de contrôler l'environnement extérieur du segment de code courant. Dans les paragraphes suivants, nous allons considérer le problème de façon différente. Nous parlerons des instructions JMP, CALL et RET qu'on peut utiliser pour contrôler le transfert selon ces méthodes. Dans le Chapitre 6, nous expliquerons comment les interruptions et les instructions IRET emploient les mêmes méthodes pour gérer les interruptions.

Nous décrirons les "méthodes" de commutation de tâches et de transfert de contrôle d'une tâche vers une autre dans le paragraphe suivant. La commutation de tâches est la partie la plus complexe du modèle de segmentation du 80386, mais elle n'est pas employée par tous les systèmes 80386. En remettant à plus tard la description de la commutation des tâches, nous espérons clarifier l'explication des autres méthodes de contrôle de transfert qui doivent être utilisées sur tous les systèmes 80386.

Le Chapitre 3 comportait une description des instructions JMP, CALL et RET dans laquelle le transfert vers un autre segment code s'effectuait en changeant les registres CS et EIP pour les faire pointer sur une instruction d'un autre segment. C'est ce que voit le programmeur d'applications : le programme exécute l'instruction de transfert de contrôle puis poursuit son exécution dans un segment code différent. Le programmeur système en voit beaucoup plus. Les registres cachés du descripteur de CS sont chargés avec le descripteur du nouveau segment code après qu'un certain nombre de tests de protection ont été appliqués au sélecteur CS et au descripteur associé. Le transfert peut même s'effectuer vers un segment code d'un niveau de privilège différent dans la même tâche, ou vers un segment code d'une tâche différente.

Même niveau, même tâche

Le transfert intersegment le plus simple utilise une instruction JMP, CALL ou RET pour passer à un segment code au même niveau de priorité dans la même tâche. Ce transfert s'effectue en identifiant le nouveau sélecteur CS à un segment mémoire présent et exécutable avec DPL = CPL, ou à un segment conforme, exécutable et présent (décrit plus bas) avec DPL CPL. Ce type de transfert de contrôle est identique au chargement d'un registre segment données avec un sélecteur pour un segment données. C'est un peu la méthode employée pour charger le registre CS. Le nouveau sélecteur CS est testé plusieurs fois et si ces tests sont positifs, c'est le descripteur associé qui est lu dans la table des descripteurs et qui est testé à son tour plusieurs fois. Si ces tests sont positifs, le descripteur est chargé dans les registres cachés de CS, le nouveau sélecteur dans CS, et le nouveau déplacement dans EIP pour achever le transfert de contrôle.

Les segments exécutables conformes sont un type particulier de segments mémoire qui permettent de partager des sous-programmes entre des programmes de niveaux de privilège différents sans changer ces privilèges. Par exemple, des programmes s'exécutant à des niveaux de privilège différents peuvent se partager une bibliothèque mathématique en la plaçant dans un segment conforme. Ainsi, un programme de niveau quelconque pourra appeler un sous-programme de la bibliothèque en employant un appel intersegment, et la routine s'exécutera au niveau de privilège de l'appelant.

Un transfert de contrôle vers un segment conforme exécutera ce segment au niveau de privilège de l'appelant plutôt qu'au DPL du segment. Au contraire, le DPL d'un segment conforme est employé pour déterminer le niveau de privilège le plus interne qui permet d'accéder à ce segment. Cette interprétation du DPL est contraire à l'interprétation habituelle du DPL. Normalement, on emploie DPL pour déterminer le niveau de privilège le plus externe qui permet d'accéder au segment. Ainsi, un programme au niveau 3 peut accéder à n'importe quel segment conforme, mais des routines au niveau 0 ne pourront accéder qu'à des segments conformes de DPL = 0.

Niveaux différents, même tâche

La possibilité d'effectuer des transferts entre des segments de même niveau de privilège est importante mais ne suffit pas, puisqu'il existe quatre niveaux de privilège sur le 80386, et non un seul ! L'instruction CALL permet d'effectuer un transfert vers un niveau interne en passant par un guichet. L'instruction RET autorise un transfert vers un segment de niveau externe pour revenir d'un appel à un niveau interne. L'instruction JMP ne permet pas d'accéder à un niveau différent.

Il est ainsi possible pour des programmes d'application d'appeler directement des sous-programmes du système d'exploitation de niveau interne pour accéder à des fichiers ou à l'allocation mémoire. Seuls ces types d'appel sont permis, puisque le système d'exploitation n'a pas besoin d'appeler les programmes d'application. Quand le système d'exploitation doit accéder à un niveau externe, plutôt que de rendre la main après un appel, on peut empiler le pointeur sur le segment approprié et le déplacement au niveau externe, puis exécuter l'instruction RET. Cette technique est utile pour accéder à un programme qui vient d'être chargé pour commencer son exécution (voir l'exemple d'initialisation du Chapitre 7).

On doit contrôler avec soin les transferts vers des niveaux internes pour assurer l'intégrité du mécanisme de protection. Les niveaux externes ont le droit d'accéder aux niveaux internes uniquement par des points d'entrée définis par le système d'exploitation. Ce dernier doit contrôler à la fois le segment et le déplacement de ces points d'entrée. Autrement, si seule la partie segment était contrôlée, une application pourrait accéder à n'importe quel déplacement du segment. Par exemple, elle pourrait spécifier un déplacement juste après le code qui teste les paramètres passés au niveau interne. Ou pire encore, le niveau externe pourrait spécifier un déplacement qui tomberait au milieu d'une instruction.

Les guichets d'appel

Les guichets d'appel nous apportent le mécanisme nécessaire pour contrôler l'accès aux routines internes. Ce sont des types particuliers de descripteurs qui contiennent des pointeurs sur les points d'entrée. Le descripteur guichet contient un pointeur 48 bits sur le point d'entrée, qui inclut ainsi à la fois la composante segment et la composante déplacement. Un CALL à travers un guichet d'appel (*call gate*) est identique à un CALL indirect à travers un guichet. Pour employer le guichet, la routine externe détermine un sélecteur pour le guichet comme le sélecteur qui doit être "chargé" dans CS. Lorsqu'une instruction CALL teste le nouveau descripteur de CS et découvre un guichet d'appel au lieu d'un segment mémoire exécutable, le pointeur est extrait du guichet et utilisé comme pointeur pour le transfert de contrôle. Le déplacement provenant du guichet est utilisé à la place du déplacement donné dans l'instruction pour contrôler le point d'entrée du segment interne. Le sélecteur provenant du guichet est employé pour lire un autre descripteur, qui doit être un segment exécutable, destination véritable de l'appel. Le transfert de guichet à guichet est interdit !

La Figure 5.24 montre comment le guichet d'appel détermine indirectement la destination de l'appel. Notez que le déplacement provenant de l'instruction est mis à l'écart et remplacé par le déplacement provenant du guichet. Le sélecteur dans le guichet est utilisé pour accéder à un autre descripteur qui doit être celui d'un segment exécutable.

Le champ DPL du descripteur de guichet contrôle l'accès au guichet utilisant les mêmes règles de privilège que celles de l'accès aux données. Seules les procédures qui s'exécutent au même niveau ou à un niveau interne relativement au DPL du guichet peuvent employer ce guichet. Le champ RPL du sélecteur pour le guichet est aussi testé pour s'assurer qu'il est au même niveau ou à un niveau interne relativement au DPL du guichet. Ainsi, le guichet est placé au niveau le plus externe pour lequel son accès est autorisé.

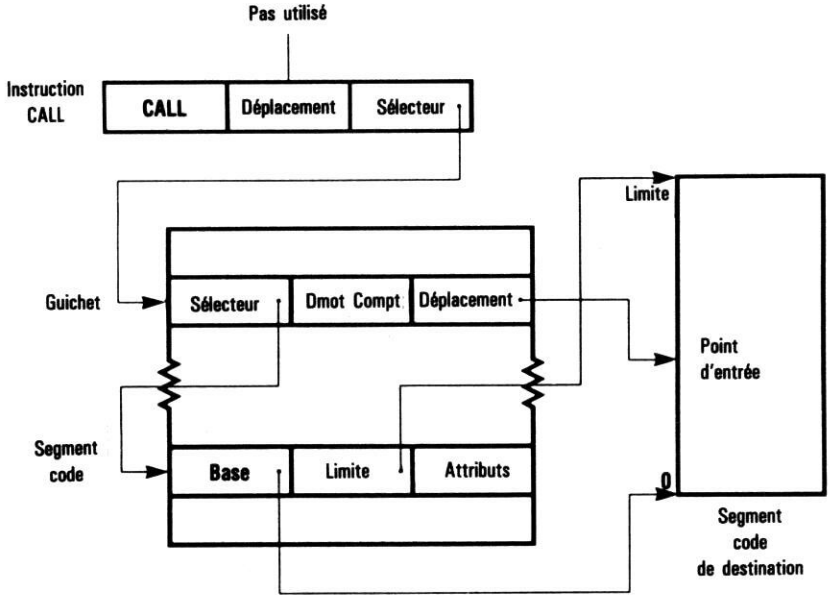


Figure 5.24 : *Accès à travers un guichet d'appel.*

Le système applique un jeu différent de tests sur les niveaux de privilège au descripteur du segment exécutable sur lequel le sélecteur du guichet pointe. Le champ RPL du sélecteur dans le guichet est ignoré. Seul le champ DPL du segment exécutable est employé pour ces tests de niveaux de privilège. Si DPL = CPL, ou si le segment est conforme avec DPL ≤ CPL, les transferts de guichet s'effectuent au même niveau. Les instructions JMP et CALL peuvent employer les guichets d'appel pour

accéder à un segment de même niveau. Si DPL est plus petit que CPL, un accès à un niveau interne est effectué. Seules les instructions CALL peuvent employer des guichets d'appel pour accéder à des niveaux internes. DPL > CPL est interdit pour éviter des appels à des niveaux externes. Les instructions RET peuvent utiliser des guichets d'appel quels que soient les niveaux de privilège.

La Figure 5.25 montre (pour une instruction CALL) les relations permises entre CPL, le niveau du guichet et le niveau du segment exécutable pointé par le guichet. Un programme qui s'exécute au niveau 2 dans le segment $Code_A$ peut accéder aux guichets des niveaux 2 et 3. Ces guichets doivent pointer sur des segments exécutables au niveau 2 ou à des niveaux internes (0 et 1). Les relations possibles sont illustrées par des flèches pleines, tandis que les relations interdites sont représentées par des pointillés.

Le $Guichet_B$ au niveau 2 pointe sur un segment exécutable au niveau 2 ($Code_B$), donc l'utilisation du $Guichet_B$ à partir du niveau 2 ne change pas le niveau de privilège. En fait, $Code_B$ est directement accessible sans utiliser le guichet (voir la Figure 5.25). Le $Guichet_C$ au niveau 3 pointe sur un segment $Code_C$ au niveau 0, donc l'emploi du $Guichet_C$ à partir du niveau 2 entraîne un changement de niveau de privilège. $Code_C$ n'est pas accessible directement mais indirectement à partir du $Guichet_C$. De même, le $Guichet_D$ au niveau 2 pointe sur un segment $Code_D$ au niveau 1, qui n'est pas directement accessible à partir du niveau 2. L'utilisation de $Guichet_D$ à partir du niveau 2 entraîne aussi un changement de niveau de privilège. Le $Guichet_Z$ au niveau 1 n'est pas accessible à partir du niveau 2, de même que le segment $Code_Z$ à partir du niveau 0.

Commutation de piles

Un CALL à travers un guichet d'appel vers un niveau interne commute le niveau de privilège et accède au nouveau segment de code, mais commute aussi le segment pile pour le niveau interne. La pile reste inchangée si le CALL est effectué au même niveau, que ce soit à travers un

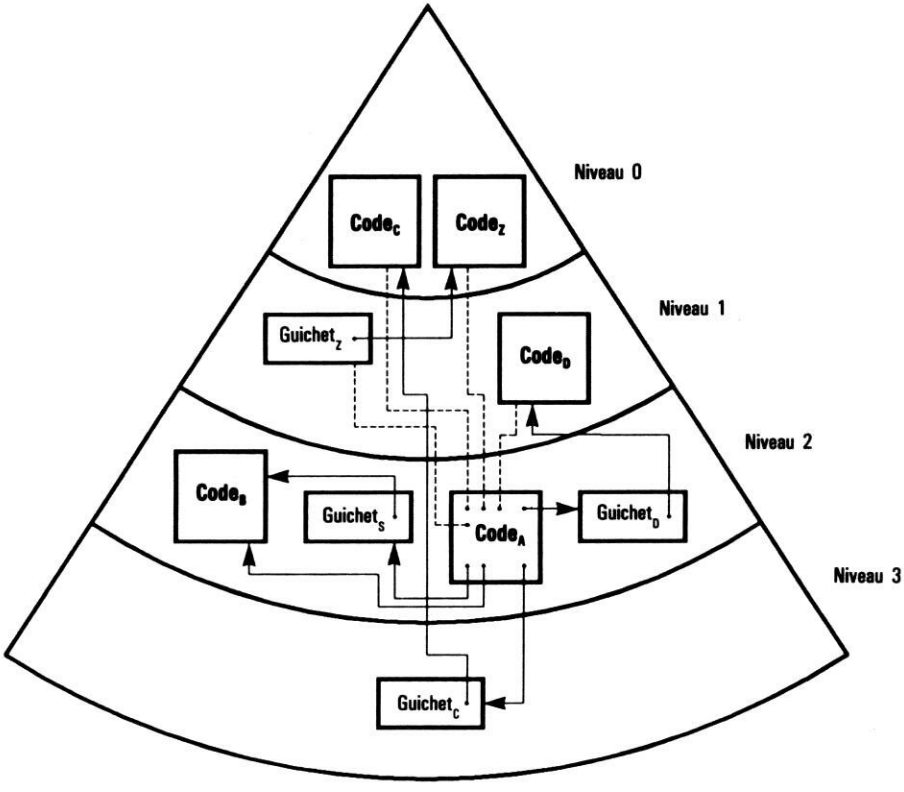


Figure 5.25 : *Niveaux de privilège et guichets.*

guichet ou non. Rappelez-vous que le TSS contient des pointeurs sur des piles de niveaux 0, 1 et 2. Au cours d'un CALL interniveau, les registres SS et ESP sont initialisés avec le pointeur approprié de TSS. On obtient ainsi une pile vide au nouveau niveau, puisque le pointeur ESP dans TSS

pointe généralement sur la limite supérieure du nouveau segment de pile. Cependant, la pile ne reste pas vide très longtemps !

Quand le système charge le registre SS avec le sélecteur pour le segment pile de niveau interne, il applique les mêmes tests de privilège que pour une instruction MOV SS,sélecteur exécutée au niveau de privilège interne. Cela signifie que le segment doit être du type lecture/écriture, et qu'il doit avoir RPL = DPL = CPL au niveau interne.

Après l'initialisation de la pile avec le pointeur de TSS, les anciennes valeurs des registres SS et ESP sont empilées au niveau interne pour permettre de récupérer la pile de niveau externe. Ensuite, on peut copier de 0 à 31 dmots (124 octets) depuis la pile de niveau externe vers la pile de niveau interne. Ceci inclut les paramètres qui ont été empilés au niveau externe avant l'appel, mais dont la procédure de niveau interne a besoin dans la nouvelle pile. Le champ DmotCompt du descripteur de guichet d'appel donne le nombre de dmots à copier.

La Figure 5.26 montre le fonctionnement de la commutation de piles lors d'un CALL à travers un guichet vers un niveau interne. La pile externe est à droite de la figure et contient quatre paramètres dmots (P1, P2, P3, et P4) empilés avant l'instruction CALL. La pile interne est à gauche de la figure. Les valeurs SS et ESP de la pile externe sont empilées en premier dans la pile interne. Ensuite, les quatre dmots de paramètres sont copiés de la pile externe vers la pile interne, puisque le guichet a un champ DmotCompt égal à 4. Finalement, l'adresse de retour est empilée comme les valeurs de CS et EIP pour le programme de niveau externe. Après l'instruction CALL, SS adresse le segment pile pour le niveau interne, et ESP pointe dans ce segment sur la valeur EIP empilée par le CALL.

Bien que les guichets soient plus pratiques pour accéder à un niveau interne, ils peuvent spécifier un segment code qui est au niveau de privilège courant, ou qui est conforme. Par cet emploi des guichets d'appel, on n'obtient ni une commutation de niveaux de privilège, ni une commutation de piles. Les instructions CALL et JMP peuvent employer des guichets pour accéder à un segment à un même niveau de privilège. Cet emploi du guichet équivaut à une instruction CALL ou JMP indirecte.

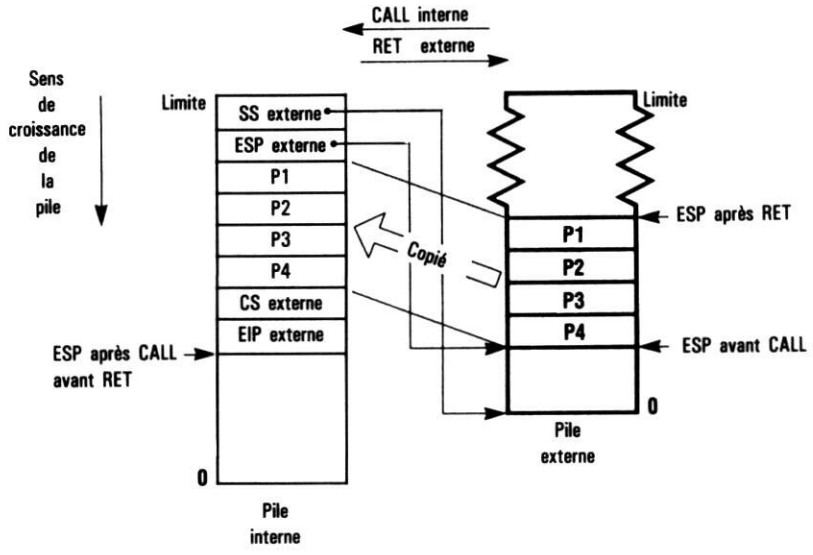


Figure 5.26 : *Commutation sur une pile de niveau interne.*

Transparence

Les guichets sont transparents pour le programme appelant : c'est une propriété importante. En effet, le programme appelant utilise l'instruction intersegment standard CALL pour accéder à un segment différent et ne peut pas déterminer si le sélecteur donné dans l'appel identifie le segment mémoire, ou s'il identifie un guichet qui spécifie indirectement le nouveau segment et substitue un nouveau déplacement. Puisque le système d'exploitation contrôle le contenu des tables des descripteurs, il peut em-

ployer des guichets pour intercepter des accès à des routines de différents niveaux de privilège.

L'emploi de guichets pour appeler directement des procédures du système d'exploitation convient parfaitement au modèle de programmation à segments multiples et améliore son utilisation. Cependant, puisque le modèle de programmation à segment unique n'emploie pas d'appels intersegments, les bienfaits de la transparence deviennent moins évidents. Dans le modèle à segment unique, un appel à une procédure du système d'exploitation est généralement un appel intrasegment à une routine de la bibliothèque utilitaire qui contient elle-même un appel intersegment à travers un éventuel guichet.

Retours de sous-programmes vers un niveau externe

Des CALL à travers des guichets d'appel transfèrent le contrôle de routines de niveau externe à des routines de niveau interne. Dans ce paragraphe, nous allons expliquer comment on peut utiliser l'instruction RET pour retourner d'un niveau interne vers un niveau externe. L'instruction RET intersegment dépile le pointeur de retour, et peut aussi modifier ESP pour retirer des paramètres empilés avant l'appel correspondant. La partie sélecteur de ce pointeur identifie le segment de destination. Le champ RPL de ce sélecteur identifie le niveau de privilège de ce segment. Notez que c'est le champ RPL de ce sélecteur qui est utilisé et non le champ DPL du descripteur correspondant. Ceci est nécessaire puisque le retour peut s'effectuer vers un segment conforme qui s'exécutera à un niveau différent de celui donné par son champ DPL.

Si le champ RPL du sélecteur spécifie un niveau externe relativement au CPL, un retour vers un niveau externe s'effectue. Le pointeur sur la pile externe est dépilé de la pile interne, et est chargé dans SS et ESP pour récupérer la pile externe avant que le CPL soit ajusté au niveau externe. Puis ESP est modifié pour retirer les paramètres empilés dans la pile externe avant le CALL correspondant. Avant de poursuivre l'exécution au niveau externe, les registres de segments données DS, ES, FS et GS sont testés pour s'assurer que les segments qu'ils adressent sont accessibles au niveau

externe. Si un registre segment adresse un segment qui n'est pas accessible au niveau externe, il est chargé avec un sélecteur nul pour éviter un problème de protection au retour.

Notez que les octets de paramètres sont dépilés à la fois de la pile de niveau interne et de la pile de niveau externe. Ceci a un sens puisqu'il existe deux copies des paramètres ! Les paramètres copiés dans la pile de niveau interne doivent être dépilés afin de permettre l'accès au pointeur de pile vers la pile de niveau externe. Les paramètres doivent être dépilés de la pile externe pour mettre à jour la pile après le retour.

La Figure 5.26, qui décrivait la commutation des piles pour un appel intersegment, la montre aussi lors d'un retour de sous-programme à un niveau externe. Juste avant que l'instruction RET soit exécutée, la pile interne contient l'adresse de retour, les quatre dmots de paramètres copiés lors de l'appel au niveau interne, et le pointeur SS:ESP sur la pile de niveau externe. La pile de niveau externe est composée des quatre dmots de paramètres empilés avant que l'instruction d'appel au niveau interne ait été exécutée. Une instruction RET avec un nombre de paramètres égal à 16 effacera les effets d'un appel interniveau. L'adresse de retour CS:EIP est dépilée de la pile interne, et le ESP interne incrémenté de 16 pour retirer les octets paramètres. Par conséquent, ESP pointe sur le pointeur SS:ESP qui pointe lui-même sur la pile externe. Le champ RPL du sélecteur CS de retour indique un niveau externe : les valeurs SS et ESP sont dépilées, CPL est modifié pour le niveau externe, puis le ESP externe est incrémenté de 16 pour retirer les paramètres initiaux. Après le retour, la pile de niveau interne est écartée, SS adresse la pile de niveau externe, et ESP pointe juste au-dessus des quatre paramètres, comme le montre la figure.

LA SEGMENTATION

Dans les prochains paragraphes, nous allons décrire de façon précise le fonctionnement du mécanisme de segmentation en utilisant notre notation pseudo-C. Dans ces paragraphes et dans d'autres du Chapitre 6, nous allons présenter un ensemble de sous-programmes qui décrivent précisément

le modèle de gestion mémoire du 80386. L'ordre de présentation de ces sous-programmes permet une bonne progression des concepts employés, mais ne définit pas toujours les symboles avant leur utilisation (ce qui déplairait à tout compilateur C !). Ces routines dépendent les unes des autres. Par exemple, la plupart d'entre elles appellent d'autres sous-programmes pour traiter des références mémoire. Les sous-programmes qui décrivent les commutations de tâches font référence à plusieurs niveaux de sous-programmes pour décrire ces opérations complexes. Nous pensons que les sous-programmes constituent une bonne approche de la présentation de ces notions parce qu'ils vous permettent de "digérer" un concept à la fois, et de l'employer ensuite pour construire des entités plus complexes. Cet ensemble de sous-programmes couvre toutes les actions du 80386 lorsqu'il fait référence à la mémoire, qu'il charge les registres segments, et qu'il exécute des transferts de contrôle intersegments.

Dans la description qui va suivre, nous allons surtout parler des exceptions qui sont générées pour communiquer toute violation de protection au système d'exploitation. Bien que le Chapitre 6 décrive de façon précise les exceptions, ce chapitre va indiquer les conditions qui causent une exception, et le type d'information que procure l'exception. Nous allons rapidement expliquer de quelle manière les exceptions sont décrites avant de nous plonger dans les détails.

Nous avons décomposé la description en trois parties principales :

1 Accès aux données dans des segments données.

2 Transferts de contrôle intersegments.

3 Commutations de tâches.

Dans le premier paragraphe, nous allons compléter notre notation pseudo-C et y inclure les définitions de structure, les opérateurs sur les pointeurs, et une définition des types de structures et des variables globales employés dans ce chapitre. On donnera aussi une description du mécanisme de pagination.

Dans ces paragraphes, nous allons décrire formellement le fonctionnement du modèle de gestion de la mémoire du 80386, qui gère tous les concepts présentés dans ce chapitre. Nous allons y donner une description concise et exacte du mécanisme de gestion mémoire du 80386 en reprenant avec moins de verbiage mais plus de détails pratiques les notions abordées au début de ce chapitre. Si vous lisez ceci pour la première fois, vous voudrez peut-être lire auparavant les Chapitres 6 et 7 pour vous faire une meilleure idée de toutes les possibilités du 80386. Les autres lecteurs trouveront dans les lignes qui suivent un bon manuel de référence.

Résumé sur les exceptions

Dans les descriptions suivantes, on peut trouver plusieurs types différents d'exceptions segments qui se produisent lorsqu'un programme tente une opération qui viole le modèle de protection de la segmentation. Si la pagination est activée, on peut aussi avoir des exceptions du type page même lorsqu'on utilise le mécanisme de segmentation. Dans le Chapitre 6, nous décrirons comment les exceptions interrompent le programme. Nous allons maintenant parler des conditions qui génèrent des exceptions du type segment et page, et des informations dont peut disposer un logiciel à la suite d'une exception.

Exceptions du type segment

Les notations employées pour décrire les exceptions du type segment sont les suivantes :

Commutation de tâches	$TS
Non présent	$NP
Segment de pile	$SS

Protection générale $GP

Une exception du type segment renvoie un code d'erreur que peut utiliser une routine de traitement des exceptions pour découvrir et traiter la cause de l'exception, ou pour donner un diagnostic lorsque le programme principal se terminera. La Figure 5.27 donne le format du code d'erreur. C'est une valeur 16 bits soit formée à partir du sélecteur qui a entraîné l'exception, soit égale à 0 si aucun sélecteur n'était en cause. Les opérations qui chargent un sélecteur dans un registre segment prennent directement dans le sélecteur les 14 bits supérieurs : le bit 2 est le bit TI et les bits 15...3 représentent le champ d'index. D'autres opérations, par exemple les accès aux données, emploient un code d'erreur nul, puisque aucun sélecteur n'intervient directement. Le champ RPL du sélecteur est écarté et remplacé par deux bits qui indiquent l'opération qui a engendré l'exception. Le bit 0 (bit EXT) est positionné à 1 si l'exception a été générée lors du traitement d'une autre exception, ou d'une interruption externe. Le bit 1 (bit IDT) est positionné si l'exception a été générée lors de la lecture d'un élément de l'IDT, ce qui se passe uniquement lors du traitement d'une interruption ou d'une exception (voir le Chapitre 6). Nous allons préciser comment le code d'erreur est formé pour chaque exception possible.

Nous faisons référence à une exception en appelant le sous-programme SegmentException() (voir le Chapitre 6) qui reçoit deux paramètres :

1 Le numéro du vecteur à employer. On utilise les mnémoniques $TS, $NP, $SS et $GP. Dans le Chapitre 6, nous donnerons la correspondance entre les numéros des vecteurs et les exceptions.

2 Le sélecteur à employer dans le code d'erreur renvoyé par l'exception.

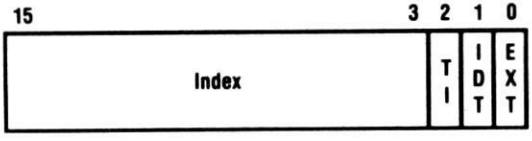


Figure 5.27 : *Format d'un code d'erreur d'une exception du type segment.*

Exceptions du type page

Des exceptions du type page sont aussi générées par les opérations décrites dans ce chapitre. Bien qu'il existe plusieurs sortes d'exceptions du type segment, il n'y a qu'une seule sorte d'exception du type page. Quand une exception du type page est générée, le processeur charge CR2 avec l'adresse linéaire causant l'exception. Les exceptions du type page renvoient aussi un code d'erreur, pour aider le système d'exploitation à diagnostiquer l'exception.

Le code d'erreur d'une exception du type page est constitué seulement de 3 bits significatifs (voir la Figure 5.28). Le bit 0 (bit P) indique si l'exception était causée par une page non présente (P = 0) ou par une violation de protection de page (P = 1). Les bits 1 et 2 donnent le type d'accès qui a causé l'exception. Le bit 1 (bit W) est égal à 1 s'il y a eu écriture, et à 0 s'il y a eu lecture. Le bit 2 (bit U) est égal à 1 si c'est un

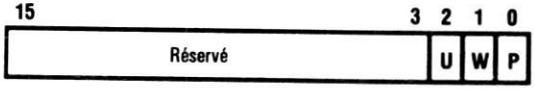


Figure 5.28 : *Format d'un code d'erreur d'une exception du type page.*

programme qui s'exécute au niveau de privilège 3 (niveau utilisateur) qui a effectué l'accès, et est égal à 0 si ce sont des programmes aux niveaux 0, 1, et 2.

Nous faisons référence à une exception du type page en appelant une routine du type PageException() (voir le Chapitre 6) qui reçoit quatre paramètres :

1 L'adresse linéaire ayant causé l'exception.

2 Le niveau de privilège de l'accès tenté pour former le bit U du code d'erreur.

3 Le bit W du code d'erreur (1 si écriture, 0 si lecture).

4 Le bit P du code d'erreur (1 si exception du type protection, 0 si exception du type non présent).

Arrêt de l'exécution des descriptions d'instructions

Les routines PageException() et SegmentException() ne rendent pas la main à l'appelant alors que les autres sous-programmes le font. En effet, elles interrompent l'"exécution" de la description de l'instruction, mettant fin à toutes les procédures imbriquées. Ceci nous permet d'être plus concis dans la description du traitement des exceptions tout en conservant cette concision pour les cas généraux.

L'accès aux données en mémoire

Comme on l'a vu dans le Chapitre 2, on doit spécifier la composante segment et la composante déplacement d'une adresse virtuelle pour pouvoir accéder à une donnée en mémoire. La composante segment est donnée par un registre segment qui a été chargé par le sélecteur du segment contenant les données désirées. Le déplacement est un simple déplacement d'octets dans ce segment, et il peut être généré à partir de la somme d'un registre de base, d'un registre d'index, et d'un déplacement constant. Cela signifie que l'accès à une donnée en mémoire s'effectue en deux étapes. Tout d'abord, un registre doit être chargé avec le sélecteur du segment contenant la donnée. Puis on peut faire référence aux données en précisant le registre segment avec les composantes du déplacement (registre de base, registre d'index, déplacement constant).

On décompose la description d'un accès mémoire en trois parties. Dans la première, on introduit les types et les variables globales qui sont employées pour décrire les registres. Dans la seconde, on donne une référence mémoire utilisant un registre segment qui est censé avoir déjà été chargé. Finalement, nous décrirons de façon précise le processus de chargement d'un registre segment pour conclure sur ce paragraphe. Nous décrivons d'abord les références mémoire, puisque le chargement d'un registre segment demande lui-même plusieurs références mémoire.

Notations, définitions des structures et variables globales

Avant de nous lancer dans les descriptions, il nous faut ajouter deux nouvelles constructions à la notation C introduite dans le Chapitre 3. Nous utiliserons la construction *struct* pour élaborer des structures ou des enregistrements de types de données variés tels que *int* (pour entier), *char* (pour caractère), etc. L'opérateur struct nous permet de définir nos propres types de données pour des structures complexes telles que des descripteurs de segments ou des sélecteurs. Nous utiliserons aussi l'opérateur pointeur (*) pour faire référence à des données en mémoire dont l'adresse est obtenue à partir d'une variable pointeur. Si *ptr* est un pointeur sur un entier, la construction *ptr représente l'entier pointé par ptr. Dans les définitions de variables, on emploie l'opérateur * pour définir un pointeur (voir l'exemple suivant). Le programme déclare I et J comme des entiers et PTR comme un pointeur sur un entier, puis assigne PTR à l'adresse de J et donne la valeur de I à J en passant par PTR.

```
int I,J,*PTR;    /* I,J sont des entiers        */
                 /* PTR pointe sur un entier    */
PTR = &J;        /* PTR pointe maintenant sur J */
*PTR = I;        /* Idem que J = I;             */
```

Les descriptions détaillées des accès aux données et des chargements des registres segments font appel aux définitions de structures et aux variables globales du Listing 5.2. Les définitions de structures emploient deux types qui ne sont pas des constructions standard du C. Le type *Bit* définit un champ par un nombre donné de bits. Dans une définition de structure, les champs de bits sont classés du poids faible au poids fort. Le type *filler* réserve des bits dans la structure pour représenter des bits qui ne sont pas employés.

SelectorType définit la structure d'un sélecteur, qui décrit la place des bits du champ RPL, du bit TI et du champ index. Cette définition rejoint la représentation graphique du format d'un sélecteur (voir la Figure 5.9). SegAttributes décrit la structure des attributs d'un segment. Elle donne la

configuration du second dmot (au déplacement 4) d'un descripteur. Elle inclut tous les champs attributs possibles, et la combinaison des champs de guichets et de descripteurs de segments (voir les Figures 5.10, 5.11 et 5.12). Les champs DType et Type donnent les champs attributs valides dans les champs restants.

SegmentRegister décrit la structure d'un registre segment et des registres cachés associés. Elle contient quatre sous-champs qui suivent la structure de la Figure 5.13 : un sélecteur qui a le type de structure SelectorType, un champ attribut qui a le type de structure SegAttributes, et deux entiers 32 bits qui décrivent la base et la limite. La limite du segment est stockée comme une limite 32 bits de granulité d'un octet. Les chargements des registres segments transforment des limites de granulité 4 Ko en limites de granulité d'un octet avant de les ranger dans le registre limite.

```
Struct SelectorType{
   Bit(2)       RPL;
   Bit(1)       TI;       /* 0 si GDT, 1 si LDT */
   Bit(13)      Index;    /* Index dans GDT ou LDT du
                          /* descripteur */
}
/* La structure suivante représente le second dmot */
/* du descripteur */
/* Les champs Dtype et Type indiquent quels sont   */
/* les autres champs valides                       */
struct SegAttributes {
   Bit(5)       DwordCount; /* Le nombre de paramètres */
                  /* pour un guichet d'appel seulement */
   filler(3);
   Bit(4)       Type;
   Bit(1)       DType;  /* 0 si Segment Système ou Guichet, */
                        /*1 si Segment Mémoire */
   Bit(2)       DPL;
```

Listing 5.2 : *Définitions des structures et variables globales.*

```
   Bit(1)       P;       /* Bit présent */
   filler(6);
   Bit(1)       D;       /* Taille par défaut (16 vs. 32),  */
                         /* voir le Chapitre 9 */
   Bit(1)       G;       /* Granulité de la limite. */
                         /* 0=octet, 1=4K */
}
struct SegmentRegister {
   SelectorType   Selector;  /* Registre sélecteur visible */
   SegAttributes  Attributes;/* Registres cachés invisibles*/
   int    Base,
          Limite;
}
SegmentRegister   CS, SS, DS, ES, FS, GS,
                  /* Registres segments */
                  TR, LDTR,
                  /* Registres segment système */
                  IDTR, GDTR;
                  /* Utilise seulement les */
                  /* champs de base et de limite*/

int   CRO, CR2, CR3;   /* Registres 32 bits de contrôle */
                       /* du processeur                 */
int   CPL;             /* Niveau de Privilège Courant   */

/* Registres généraux 32 bits */
int   EAX, EBX, ECX, EDX, ESP, EBP, ESI, EDI;

/* Registres de contrôle du processeur */
struct EflagsType   {
   Bit(1)   CF;
   filler(1);
   Bit(1)   PF;
   filler(1);
   Bit(1)   AF;
```

Listing 5.2 : *Définitions des structures et variables globales (suite).*

```
    filler(1);
    Bit(1)    ZF, SF, TF, IF, DF, OF;
    Bit(2)    IOPL;
    Bit(1)    NT;
    filler(1);
    Bit(1)    RF, VM;
    filler(14);
}

EflagsType    EFLAGS;
int           EIP;
```

Listing 5.2 : *Définitions des structures et variables globales (fin).*

Le type de structure SegmentRegister définit les six registres segments CS, SS, DS, ES, FS et GS. C'est cette même structure qui définit les registres système LDTR, TR, IDTR et GDTR. IDTR et GDTR n'emploient ni sélecteur ni champs attributs. CPL est défini comme un entier 32 bits. Les registres de contrôle CR0, CR2, et CR3 sont définis comme des entiers 32 bits, ainsi que les registres généraux EAX, EBX, ECX, EDX, ESP, EBP, ESI et EDI.

EflagsType décrit la structure du registre EFLAGS et rejoint la définition graphique de la Figure 2.2. Le registre EIP est défini comme un entier 32 bits.

Références à des données

Si un sélecteur a été chargé dans un registre segment et le descripteur correspondant dans les registres cachés, on peut faire référence à des données contenues dans le segment adressé par ce registre. Chaque référence à des données entraîne une suite de tests de protection, décrits de façon précise par la routine AccessVirtual() du Listing 5.3. Le système effectue la

plupart des tests de protection lors du chargement du registre segment. Il réduit le nombre de tests effectués à chaque référence mémoire pour améliorer les performances puisqu'il y a habituellement beaucoup plus de références mémoire que de chargements de registres segments.

Dans un paragraphe du Chapitre 3, nous avons fait référence à des opérandes stockés en mémoire, qui sont tous des "calls" à la routine AccessVirtual(), même si les notations du Chapitre 3 n'en font pas mention. La routine AccessVirtual() décrit donc les références à la mémoire.

```
/* Accès à des données dans un segment adressé */
/* par un Registre Segment SReg                */
AccessVirtual(SReg, Offset, Length, RW, IntNumber, Data)
   SegmentRegister   SReg;
   int   Offset, Length,   /* déplacement dans un segment, */
                           /* et longueur de la donnée     */
   RW,                     /* 0 si lecture, 1 si écriture  */
   IntNumber,              /* $SS pour accès SS, $GP pour  */
                           /*les autres registres          */
   *Data;                  /* Pointeur sur donnée pour     */
                           /*lecture ou écriture           */
{
   if (SReg.Attributes.P==0)
/* registre Segment Non valide(Nul) */
      SegmentException(IntNumber, 0);

   switch   (SReg.Attributes.Type)
      case 0:   case 1:   /* Lecture */
      case 10:  case 11:  /* Lecture/Exécution */
      case 14:  case 15:  /* Lecture/Exécution, Conforme */
         /* Test si écriture d'un segment du type Lecture */
         if (RW==1)
            SegmentException(IntNumber, 0);
      case 2:   case 3:   /* Lecture/Ecriture */
         /* Test pour limite de segment classique */
```

Listing 5.3 : *Sous-programme AccessVirtual().*

```
         if ( (Offset+Length-1) > SReg.Limit )
            SegmentException(IntNumber, 0);
         break;

      case 4:   case 5:   /* Ecriture, Expand-down*/
         /* Test si écriture d'un segment du type Lecture */
         if (RW==1)
            SegmentException(IntNumber, 0);
      case 6:   case 7:   /* Ecriture/Lecture, Expand-down */

   /* Test pour la limite d'un segment du type expand-down*/
         if ( (Offset <= SReg.Limit) || ((Offset+Length-1)>=4G) )
            SegmentException(IntNumber, 0);
         break;

      case 8:   case 9:   /* Exécution */
      case 12:  case 13:  /* Exécution , Conforme */
/*Ne peut pas lire ou écrire un segment du type Exécution */
         SegmentException(IntNumber, 0);
         break;
   }

   AccesLinear(SReg.Base+Offset, Length, CPL, RW, Data);
} /* fin d'AccessVirtual() */
```

Listing 5.3 : *Sous-programme AccessVirtual() (fin).*

Cette routine teste d'abord le registre segment pour vérifier qu'il ne contient pas un sélecteur nul (test du bit P dans les registres descripteurs cachés). Puis elle teste le déplacement pour s'assurer qu'il est compris dans la limite du segment. Il y a deux tests de la limite : un pour les segments classiques (expand-up) et un second pour les segments à croissance vers le bas (expand-down). Elle compare ensuite le type de référence avec les attributs du segment pour empêcher toute écriture dans un segment du type lecture ou toute lecture ou écriture dans un segment du type exécution. Si un des tests échoue, il y a génération d'une exception et le système n'effectue pas la référence mémoire. Sinon, la segmentation puis la pagination traitent cette référence mémoire en faisant appel à la routine Ac-

cessLinear() avec l'adresse linéaire obtenue par l'addition de la base du segment au déplacement de la donnée. Si tous les tests d'accès à la pagination sont positifs, l'instruction peut se terminer normalement. Notez que le système considère CPL comme le niveau de privilège d'accès pour ce type de références directement générées par le programme.

Pagination

Le mécanisme de pagination est décomposé en deux sous-programmes : AccessLinear() et TranslateLinear(). AccessLinear(), décrit dans le Listing 5.4, appelle TranslateLinear() pour transformer les adresses linéaires en adresses physiques, puis la routine AccessPhysical() pour faire référence à la mémoire physique. AccessPhysical() traite aussi le cas d'une référence à cheval sur deux pages. Par exemple, faire référence au dmot à l'adresse 0FFFh : le premier octet du dmot est dans la page 0, à l'adresse 0FFFh et les trois octets suivants sont dans la page 1 aux adresses 1000h à 1003h. Dans ces cas particuliers, AccessLinear() décompose la référence en deux références dans deux pages différentes.

Le coeur du mécanisme de pagination se trouve dans la routine TranslateLinear() du Listing 5.5. Cette routine teste d'abord le bit PG de CR0 pour voir si la pagination est activée. Sinon, elle retourne simplement l'adresse linéaire comme adresse physique. Si la pagination est activée, elle lit les deux niveaux de la table des pages pour transformer l'adresse linéaire en adresse physique. Elle teste à la fois les éléments du catalogue et ceux de la table des pages, et détecte une exception si un élément est marqué *non présent*, ou s'il y a une violation de protection de la page. La routine a été écrite pour toujours lire les deux niveaux de la table des pages. Comme on l'a vu dans ce chapitre, le 80386 possède un cache de pagination qui contient les tables des pages les plus couramment employées.

Accès à la mémoire physique

Le Sous-programme AccessPhysical() résume le fonctionnement de tout accès à la mémoire physique. Nous ne détaillerons pas cette routine puisque le système mémoire est indépendant du processeur 80386. Cette routine reçoit quatre paramètres :

1 L'adresse physique à laquelle on veut accéder.

2 La longueur de la donnée à laquelle on fait référence.

3 Lecture/Ecriture.

4 Un pointeur sur la donnée à laquelle on veut accéder.

```
/* Les routines suivantes, AccessLinear et TranslateLinear,*/
/* simulent le mécanisme de translation de page.           */
/* TranslateLinear transforme les adresses linéaires en    */
/* adresses physiques.                                     */
/* AccessLinear utilise cette fonction pour permettre      */
/* l'accès en lecture et en écriture à l'espace d'Adresse  */
/* Linéaire.                                               */

AccessLinear(LAddress, Length, PL, RW, Data)
   int   LAddress, Length,
/* Adresse linéaire et longueur de donnée */
         PL,              /* Niveau de privilège d'accès */
         RW,              /* 0 si lecture, 1 si écriture */
         *Data;           /* Pointeur sur donnée pour    */
                          /*lecture ou écriture          */
{
```

Listing 5.4 : *Sous-programme AccessLinear().*

```
   int    PAddress,
          LAddress2, PAddress2,
          Len1,
          Len2;

/* Test si référence sur deux pages */
if ( (Laddress % 4096) + Length) > 4096)
   {
   /* Accès divisé en deux parties sur deux pages */
   /* La première partie est à l'adresse LAdress  */
   Len1 = 4096 - (LAddress % 4096);
   PAddress = TranslateLinear(LAddress, PL, RW);

   /* La seconde partie à la page suivante (LADress+Len1) */
   /* de longueur Length-Len1 */
   LAddress2 = LAddress + Len1;
   Len2 = Length-Len1;
   PAddress2 = TranslateLinear(LAddress2, PL, RW);

   /* Les deux parties sont accessibles donc accès */
   /* mémoire physique */
   AccessPhysical(PAddress, Len1, RW, Data);
   AccessPhysical(Paddress2, Len2, RW, Data+Len1);

}

else   { /* Accès sur une page classique */
   PAddress = TranslateLinear(LAddress, PL, RW);
   AccessPhysical(PAddress, Length, RW, Data);
   }
}   /* fin d'AccessLinear */
```

Listing 5.4 : *Sous-programme AccessLinear() (fin).*

```
TranslateLinear(LAddress, PL, RW)
   int   LAddress,
   PL,       /* Niveau de Privilège d'accès */
   RW;       /* 0 si lecture, 1 si écriture */

{
   /* La structure PTE définit le format des éléments de la */
   /* table des pages aux deux niveaux de cette table des   */
   /*  pages.*/
   struct   PTE {
      bit(1)   P;   /* 1 si valide pour la translation */
      bit(1)   RW;  /* 1 si on peut écrire             */
      bit(1)   SU;  /* 1 si accessible au PL 3         */
      filler(2);
      bit(1)   A;   /* Bit "Accessed"  */
      bit(1)   D;   /* Bit "Dirty"     */
      filler(5);
      Bit(20)  PageFrame;  /* Adresse physique 31..12  */

   PTE   DEntry,    /* Elément du catalogue (1er niveau)*/
   PEntry;          /* Elément de la table des pages    */
                    /*(2ème niveau) */
   int   PAddress;

/* Test du bit PG dans CRO pour voir si pagination.   */
/* Sinon retourne simplement l'adresse linéaire comme */
/* adresse physique                                   */
if ((CRO & 80000000h) == 0)
   return(LAddress);

/* Si PG = 1, lit l'élément du catalogue en utilisant */
/* les 10 bits de poids fort de Ladress comme index   */
/* dans le catalogue pointé par CR3.                  */
/* Notez que les attributs de page ne sont testés que */
/* lorsque les deux niveaux de la table des pages ont */
```

Listing 5.5 : *Sous-programme TranslateLinear ().*

```
/* été lus                                                    */
PAddress = (CR3        & OFFFFF000h) + ( (LAddress>>20) & OFFCh);
AccessPhysical(PAddress, 4, 0   /* Lecture */, &DEntry);
if (DEntry.P == 0)       /* Non Présent */
      PageException(LAddress, PL, RW, DEntry.P);

/* mise à jour du bit A pour les éléments présents */
DEntry.A = 1;
AccessPhysical(PAddress, 4, 1   /* Ecriture */, &DEntry);

/* Lit l'élément de la table des pages en utilisant les */
/* 10 bits du centre de LAdress comme un index dans la  */
/* table des pages pointée par l'élément du catalogue   */
PAddress = (DEntry & OFFFFF000h) + ( (LAddress>>10) & OFFCh);
AccessPhysical(PAddress, 4, 0   /* Lecture */, &PEntry);
if ( (PEntry.P == 0)       /* Non Présent */
   ||    ( (PL == 3) &&
         ( /* Test si violation de protection de page */
            (DEntry.U == 0)
         ||    (PEntry.U == 0)
         ||    ( (DEntry.RW == 0) && (RW == 1) )
         ||    ( (PEntry.RW == 0) && (RW == 1) )
         )
      )
   )
   PageException(LAddress, PL, RW, PEntry.P);
   /* Si transformation valide, mise à jour des bits D et A */
   PEntry.A = 1;
   if (RW == 1)    /* Ecriture */
      PEntry.D = 1;
   AccessPhysical(PAddress, 4, 1   /* Ecriture */, &PEntry);

   Return( (PEntry & OFFFFF000h)  /* bits 31..12 de la table */
                                  /* des pages */
      + (LAddress & OFFFh)        /* bits 11..0 de l'adresse */
                                  /*linéaire   */

} /* fin de TranslateLinear */
```

Listing 5.5 : *Sous-programme TranslateLinear () (fin).*

Chargement d'un registre segment

Avant qu'un programme puisse accéder à des données dans un segment mémoire, ce dernier doit être rendu adressable par le chargement de son sélecteur du segment dans un des registres segments (par exemple, par un MOV). Dans ce paragraphe, nous allons expliquer comment un registre segment est chargé et en particulier comment le descripteur de segment est chargé dans les registres cachés des descripteurs.

Les instructions de chargement décrites ici ne peuvent charger que les registres DS, ES, FS, GS et SS. Seuls les transferts de contrôle intersegments chargent CS. Notez qu'on peut faire référence à une donnée dans le segment de code courant en donnant CS comme registre segment et en précisant le déplacement de la donnée dans ce segment. Puisque les segments chargés dans CS doivent être soit du type exécution, soit du type exécution/lecture, on ne peut effectuer que des accès lecture dans le segment code courant.

La routine SRegLoad() du Listing 5.6 donne une description du chargement d'un registre segment. Elle reçoit trois paramètres :

1 SReg, le registre segment qu'on doit charger.

2 Selector, le sélecteur qu'on doit charger.

3 GP ou TS, qui sélectionnent le type des exceptions générées : exceptions $GP ou $TS. Les chargements de segment dans les instructions MOV, POP et Lsr emploient $GP. Ceux dans les commutations de tâches (voir le TaskSwitch()) utilisent $TS.

La routine SRegLoad() charge uniquement les registres cachés des descripteurs si tous les tests de protection sont positifs. C'est à l'appelant de charger la partie visible du registre segment : le sélecteur. La routine utilise deux jeux différents de tests de protection : un pour les sélecteurs chargés dans DS, ES, FS et GS, et un second pour les sélecteurs chargés dans le registre SS (segment pile).

```
/* SRegLoad charge SS, DS, ES, FS, ou les registres */
/* cachés de GS avec un descripteur. */
SRegLoad(SReg, Selector, GPorTS)
   SegmentRegister   SReg;
   SelectorType      Selector;
   int   GPorTS;
/* 13 ou 10 pour sélectionner les exceptions */
/* GP ou TS sauf pour les exceptions du type */
/* non présent */
{
   SegAttributes     Attributes;
   SelectorType      GSelector;
   int   Base, Limit, GOffset;

/* Un jeu de tests pour SS, un autre pour DS, ES, FS, GS */
if (SReg == SS)   {
   if ( ((Selector & OFFFCh) == 0   /* Nul */)
      ||   (Selector.RPL != CPL) )
      SegmentException(GPorTS, 0);

/* Si les tests de Selector sont positifs, lit Descriptor */
/* et le teste */
   ReadDescriptor(Selector,&Attributes,&Base,&Limit,&GSelector,
                &GOffset);
   if ( (Attributes.DPL != CPL)
      ||   (Attributes.DType == 0)
/* Segment système ou Guichet */)
      SegmentException(GPorTS, 0);
   switch   (Attributes.Type) {   /* Ignore l'attribut A  */
      case 2:   case 3:   /* Lecture/Ecriture             */
      case 6:   case 7:   /* Lecture/Ecriture expand-down */
         /* Type OK, teste le bit Présent */
         if (Attributes.P == 0)
            SegmentException(12, Selector);
         break;
```

Listing 5.6 : *Sous-programme SRegLoad().*

```
      case 0:   case 1:   /* Lecture                  */
      case 4:   case 5:   /* Lecture, expand-down     */
      case 8:   case 9:   /* Exécution                */
      case 10:  case 11:  /* Exécution/Lecture        */
      case 12:  case 13:  /* Exécution, Conforme      */
      case 14:  case 15:  /* Exécution/Lecture, Conforme */
/* Type n'est pas OK, violation de segment */
         SegmentException(GPorTS, 0);
      }
   }   /* fin des tests SS */

else   {/* chargement de DS, ES, FS, ou GS */

   if ( (Selector & OFFFCH) == 0 /* Nul */)
      {/* Marque le descripteur non valide (non présent) */
      SReg.Attributes.P = 0;
      Return;
      }
   /* Si le sélecteur n'est pas le sélecteur nul, */
   /* chargement du Descripteur et test */
   ReadDescriptor(Selector,&Attributes,&Base,&Limit,&GSelector,
                &GOffset);
   if (Attributes.DType == 0)/* Segment système ou Guichet */
      SegmentException(GPorTS, 0);
   switch   (Attributes.Type){       /*Ignore l'attribut A */
      case 0:   case 1:   /* Lecture                  */
      case 2:   case 3:   /* Lecture/Ecriture         */
      case 4:   case 5:   /* Lecture, expand-down */
      case 6:   case 7:   /* Lecture/Ecriture, expand-down*/
      case 10:  case 11:  /* Exécution/Lecture        */
      /* Type OK, compare DPL avec RPL et CPL  */
         if ( (Attributes.DPL<CPL) || (Attributes.DPL<Selector.RPL) )
            SegmentException(GPorTS, Selector);
```

Listing 5.6 : *Sous-programme SRegLoad() (suite).*

```
   /* DPL ignoré lors du chargement d'un segment conforme */
   /* comme segment de données.                           */
      case 14:  case 15:  /* Exécution/Lecture, Conforme */

         /* Test du bit Présent */
         if (Attributes.P == 0)
            SegmentException(11, Selector);
      break;

      case 8:   case 9:   /* Exécution           */
      case 12:  case 13:  /* Exécution, Conforme */
         /* Type pas OK, violation segment */
         SegmentException(GPorTS, 0);
      }

   }   /* fin de DS, ES, FS, GS */

/* On arrive ici si tous les tests sont OK             */
/* Positionne le bit A dans le descripteur             */
/* Chargement du descripteur dans les registres cachés.*/

SetAccessed(Selector);

SReg.Attributes = Attributes;
SReg.Base       = Base;
SReg.Limit      = Limit;

}   /* fin de SRegLoad */
```

Listing 5.6 : *Sous-programme SRegLoad() (fin).*

Lorsqu'un sélecteur est chargé dans le registre segment, c'est d'abord sa validité qui est testée. S'il y a un problème, le système génère une exception et le registre segment n'est pas modifié. Un sélecteur nul est valide s'il est chargé dans DS, ES, FS et GS mais pas dans SS. Dans le cas d'un sélecteur nul, aucun descripteur n'est chargé et le bit P est mis à 0 dans le

champ attribut du registre caché des descripteurs pour indiquer que ce n'est pas un descripteur valide. Par la suite, toute référence mémoire qui utilisera ce registre segment générera une exception.

Si le sélecteur est valide et non nul, le système retrouve le descripteur correspondant dans la table des descripteurs (qui se trouve en mémoire) et lui applique une nouvelle série de tests. Si un des tests échoue, le système génère une exception et ne modifie pas le registre segment. Sinon, il range le descripteur dans les registres cachés correspondants. Ensuite, lorsqu'une adresse fera référence à ce registre segment, le descripteur sera déjà disponible dans des registres particuliers internes au 80386 pour une meilleure efficacité.

Le chargement d'un registre segment peut aboutir à trois résultats différents. Tout d'abord, une exception peut être générée s'il y a un problème soit avec le sélecteur, soit avec le descripteur. Deuxièmement, un sélecteur nul peut être chargé dans DS, ES, FS ou GS (mais pas dans SS). Finalement, le registre segment et le descripteur ont pu être convenablement chargés. S'il y a génération d'une exception, le registre segment et les registres cachés ne sont pas modifiés.

Tests du sélecteur

Rappelons que les sélecteurs sont constitués de trois champs : 1 bit qui détermine la table (LDT ou GDT) contenant le descripteur associé, 13 bits qui déterminent l'index dans cette table pour avoir le descripteur, et 2 bits qui donnent le RPL, qui est employé pour "affaiblir" le niveau de privilège courant (CPL).

Le système teste d'abord le sélecteur pour s'assurer qu'il n'est pas nul. Les sélecteurs nuls peuvent être chargés dans les registres DS, ES, FS et GS. Le système ne génère pas d'exception lorsqu'un de ces registres est chargé avec un sélecteur nul, mais toute référence mémoire qui utilisera ce registre engendrera une exception. Si on tente de charger le registre SS avec un sélecteur nul, une exception sera générée lors du chargement plutôt qu'à la première référence à la pile.

```
/* Routine pour lire un descripteur à partir d'une table */
/* de descripteurs. Renvoie l'attribut, plus la base, la */
/* limite et le sélecteur et le déplacement de Guichet   */
/* L'appelant doit examiner le champ attribut pour       */
/* déterminer s'il faut utiliser les champs base et      */
/* limite (segments mémoire ou système), ou les champs   */
/* sélecteur et déplacement de Guichet                   */
ReadDescriptor(Selector, Attributes, Base, Limit, GSelector, GOffset)

   SelectorType     Selector;    /* Sélecteur à charger */
   SegAttributes    *Attributes;/* Les attributs */
   int              *Base,       /* La base de segment */
                    *Limit;      /* La limite de segment */
   SelectorType     *GSelector; /* Le sélecteur de guichet */
   int              *GOffset; /* Le déplacement de guichet */
{
int   DTBase, Dword1, Dword2;

if (Selector.TI == 1   /* LDT */) {
   if (LDTR.Attributes.P == 0   /* LDT Nulle */
      || (((Selector.Index * 8) + 7) > LDTR.Limit)
      )
      SegmentException($GP, Selector);
   DTBase = LDTR.Base;
   }
else   /* GDT */ {
   if ( ((Selector.Index * 8) + 7) > GDTR.Limit)
      SegmentException($GP, Selector);
   DTBase = GDTR.Base;
   }
/* Les limites de la table sont OK, lecture des éléments */
/* du descripteur dans l'espace linéaire */
/* Lecture de la table des descripteurs en employant */
/* le niveau de privilège 0 quel que soit CPL */
```

Listing 5.7 : *Routine ReadDescriptor().*

```
AccessLinear(DTBase+Selector.Index*8, 4, 0/* PL */,
              0 /* Lecture */,&Dword1);
AccessLinear(DTBase+Selector.Index*8 + 4,4,0 /* PL */,
              0 /* Lecture */,&Dword2);

/* Champs base et limite */
*Limit = (Dword1 & OFFFFh) | (Dword2 & OF0000h);
*Base = (Dword1>>16) | ((Dword2 & OFFh)<<16)
                     | (Dword2 & OFF000000h);
*Attributes = Dword2;
if (*Attributes.G == 1)
 *Limit = (*Limit << 12) | OFFFh;/* limite de granulité 4K */

/* Sélecteur de guichet et déplacement */
*GSelector = Dword1 >> 16;
*GOffset = (Dword1 & OFFFFh) | (Dword2 & OFFFF0000h);

}   /* fin de ReadDescriptor */
```

Listing 5.7 : *Routine ReadDescriptor() (fin).*

Les sélecteurs chargés dans le registre SS sont testés afin de s'assurer que le RPL est égal au CPL. Les autres registres segments ne nécessitent pas ce test du RPL.

Tests du descripteur

Si le sélecteur est valide et non nul, le champ index est utilisé pour lire le descripteur dans la table des descripteurs. La routine ReadDescriptor() du Listing 5.7 décrit de façon précise le processus de lecture du descripteur. Cette lecture peut engendrer une exception pour les raisons suivantes :

1 Si LDT était indiqué et que le registre LDTR contient un sélecteur nul, une exception du type $GP est générée, avec le nouveau sélecteur comme code d'erreur.

2 Si le champ index spécifie un descripteur qui est partiellement ou totalement en dehors de la limite de la table des descripteurs, une exception du type $GP est générée, avec le nouveau sélecteur comme code d'erreur.

3 Si une partie ou la totalité du descripteur est stockée dans une page qui n'est pas présente, une exception du type page est générée.

La routine ReadDescriptor() effectue les deux premiers tests. Notez qu'au contraire de la routine SRegLoad(), ReadDescriptor() n'a pas de paramètres qui lui permettent de choisir entre une exception du type $GP et une exception du type $TS. Seules des exceptions du type $GP peuvent être générées lors de la lecture des descripteurs, y compris lors de commutations de tâches.

Il est possible d'obtenir des exceptions du type page lors de la lecture des descripteurs, puisque les tables des descripteurs sont stockées dans l'espace adressable linéaire comme tous les autres segments. Cela signifie que la lecture d'un descripteur fait appel à la routine AccessLinear() qui peut générer une exception du type page. Notez que lorsque ReadDescriptor() appelle la routine AccessLinear() pour lire un descripteur, elle utilise le niveau de privilège 0 lors de la lecture. Cela signifie que toute lecture de tables des descripteurs s'effectue au niveau de privilège 0, quel que soit le niveau de privilège (CPL) du programme courant. Ceci permet de placer les tables des descripteurs dans des pages qui ne sont pas accessibles à des programmes de niveau 3, tout en autorisant ces derniers à charger les registres segments.

La routine FetchDescriptor() du Listing 5.8 est identique à la routine ReadDescriptor(). Elle apparaît au Chapitre 3 lors de la description des instructions qui testent les attributs de segment. Au lieu de générer une exception si le descripteur se trouve en dehors des limites du segment, FetchDescriptor() renvoie la valeur 0 pour indiquer qu'on ne peut pas lire le descripteur. Sinon, c'est la valeur 1 qui est retournée avec les deux dmots

contenant le descripteur (le second dmot a une structure du type SegAttributes).

Après être retourné à la routine SRegLoad(), mais avant de lire le descripteur, le système effectue d'autres tests pour s'assurer que le segment peut être employé au niveau de privilège courant. Ces tests sont effectués une fois pour toutes lors du chargement du registre segment, et non à chaque référence mémoire à ce segment. Comme pour les tests du sélecteur, les tests du descripteur dépendent des registres segments chargés : un jeu de tests pour les registres segments DS, ES, FS et GS et un jeu plus contraignant pour SS.

```
/* Routine de lecture d'un descripteur à partir */
/* d'une table des descripteurs.                */
/* Retourne les deux dmots du descripteur, plus */
/* le second dmot en un type de structure       */
/* SegAttributes. Retourne 0 s'il y a des       */
/* problèmes de lecture du descripteur, sinon   */
/* retourne 1. Il ne peut pas y avoir           */
/* d'exception du type segment, mais il peut y  */
/* avoir des exceptions du type pagination (voir*/
/* la routine AccessLinear).                    */
FetchDescriptor(Selector, Dword1, Dword2, Attributes)

SelectorType   Selector;   /* Donne le descripteur désiré */
int   *Dword1, *Dword2;    /* Les deux dmots du descripteur*/
SegAttributes  *Attributes;/* Les attributs */
{

int   DTBase;

if (Selector.TI == 1   /* LDT */) {
   if (LDTR.Attributes.P == 0   /* LDT Nul */)
```

Listing 5.8 : *Sous-programme FetchDescriptor().*

```
      || (((Selector.Index * 8) + 7) > LDTR.Limit)
      )
      return(0);
   DTBase = LDTR.Base;
   }
else   /* GDT */ {
   if ( ((Selector.Index * 8) + 7) > GDTR.Limit)
      return(0);
   DTBase = GDTR.Base;
   }

/* Limites de la table OK, lecture des éléments du */
/* descripteur dans l'espace linéaire. Lecture de  */
/* la table des descripteurs en employant le niveau*/
/* de privilège 0 quel que soit CPL.               */
AccessLinear(DTBase+Selector.Index*8, 4, 0 /* PL */,
             0  /* Lecture */,&Dword1);
AccessLinear(DTBase+Selector.Index*8 + 4,4,0 /* PL */,
             0 /* Lecture */,&Dword2);

*Attributes = Dword2;
return(1);

} /* fin de FetchDescriptor */
```

Listing 5.8 : *Sous-programme FetchDescriptor() (fin).*

Le système effectue un test du niveau de privilège pour tous les descripteurs chargés. Le registre SS exige que le niveau de privilège du descripteur (DPL) soit égal au CPL. Puisque RPL a été testé auparavant pour vérifier qu'il était égal au CPL, lors de chargements dans SS nous avons besoin de DPL = RPL = CPL. Pour les autres registres segments, on doit avoir CPL et RPL à un niveau interne relativement au DPL. C'est-à-dire :

DPL ≥ CPL && DPL ≥ RPL

Etant donné qu'une exception est générée lorsque cette condition n'est pas remplie, la routine SRegLoad() inverse ce test pour voir s'il y a eu exception :

DPL < CPL || DPL < RPL

Un segment du type exécution/lecture peut être chargé dans les registres DS, ES, FS ou GS mais pas dans SS. Les segments conformes omettent le test de niveau de privilège. Si ces derniers sont du type exécution/lecture, ils peuvent être lus à n'importe quel niveau de privilège.

Le système teste le champ Type du descripteur. Les chargements dans SS exigent que le segment soit du type écriture/lecture, ce qui est la moindre des choses pour un segment pile ! Pour les autres registres segments, il suffit que le segment puisse être lu.

Si un des tests précédents échoue, le système génère une exception soit du type $GP, soit du type $TS selon la valeur du troisième paramètre passé à SRegLoad(). Si tous les tests sont positifs, le bit présent est testé. Si le segment n'est pas présent (P == 0), le système génère une exception du type $SS pour des chargements dans le registre SS et une exception du type $NP pour des chargements dans les autres registres.

Si tous les tests de protection sont positifs, l'attribut "accessed" du champ Type est positionné par la routine SetAccessed() du Listing 5.9. Puis les registres cachés sont chargés avec les informations du descripteur qu'on vient de lire. Le chargement du registre segment est effectué !

Chargement de LDTR par l'instruction LLDT

La routine LLDT() du Listing 5.10 décrit de façon précise l'instruction LLDT. Elle reçoit un paramètre, le sélecteur du nouveau segment de la LDT. L'instruction LLDT est une instruction privilégiée. On ne peut donc l'exécuter qu'au niveau 0. Tout d'abord, elle teste le sélecteur pour s'assurer qu'il identifie bien un segment avec un descripteur dans la GDT. Il est possible de charger un sélecteur nul dans le LDTR. Si c'est le cas,

```
/* Routine pour positionner le bit A dans les descripteurs */
/* de segment.*/
SetAccessed(Selector)
   SelectorType   Selector;
/* Sélecteur du descripteur à changer */

{int    DTBase, Dword2;

/* Simple lecture du second dmot d'un descripteur,*/
/* positionne le bit de poids faible du champ Type à 1, */
/* puis écrit le dmot modifié. Pas de test de type : */
/* c'est à l'appelant de s'en occuper */

/* Détermine l'adresse linéaire de la base de la table des */
/* descripteurs contenant le descripteur à modifier.       */
if (Selector.TI == 1    /* LDT */) {
   if (LDTR.Attributes.P == 0    /* LDT Nul */)
      || (((Selector.Index * 8) + 7) > LDTR.Limit)
      )
      SegmentException($GP, Selector);
   DTBase = LDTR.Base;
   }
else    /* GDT */ {
   if ( ((Selector.Index * 8) + 7) > GDTR.Limit)
      SegmentException($GP, Selector);
   DTBase = GDTR.Base;
   }

AccessLinear(DTBase+Selector.Index*8 + 4,4,0 /* PL */,
             0 /* Lecture */,&Dword2);
Dword2 = Dword2 | 100h;
/* positionne le bit de poids faible du champ type */
AccessLinear(DTBase+Selector.Index*8 + 4,4,0 /* PL */,
             1 /* Ecriture*/,&Dword2);
}
```

Listing 5.9 : *Sous-programme SetAccessed().*

tout chargement de registre segment avec des sélecteurs identifiant des segments dans la LDT générera une exception du type segment. Sinon, l'instruction lit puis teste le descripteur du segment de la nouvelle LDT. Ce doit être un segment système ou un guichet, de type LDT, et il doit être présent. Si tous ces tests sont positifs, l'instruction charge le descripteur dans les registres cachés de LDTR et le sélecteur dans le registre sélecteur LDTR.

Chargement de TR par l'instruction LTR

La routine LTR() du Listing 5.11 décrit le fonctionnement de l'instruction LTR. Elle reçoit un paramètre, le sélecteur du nouveau segment du TSS. L'instruction LTR est une instruction privilégiée. On ne peut donc l'exécuter qu'au niveau 0. L'instruction teste le sélecteur pour s'assurer qu'il identifie bien un segment avec un descripteur dans la GDT, et qu'il n'est pas nul. Puis elle lit et teste le descripteur du nouveau TSS. Ce doit être un segment système ou un guichet du type TSS 286 Disponible ou TSS 386 Disponible et il doit être présent. Si tous ces tests sont positifs, cette routine charge le descripteur dans les registres cachés de TR, et le sélecteur dans le registre sélecteur de TR après avoir changé le type du descripteur en TSS 286 Occupé ou TSS 386 Occupé.

```
/* Description de l'instruction LLDT */
LLDT(Selector)
   SelectorType       Selector;
{
   SegAttributes      Attributes;
   SelectorType       GSelector;
int   Base, Limit, GOffset;

/* Niveau 0 obligatoire pour exécuter l'instruction LLDT */
```

Listing 5.10 : *Description de l'instruction LLDT.*

```
if (CPL != 0)
   SegmentException($GP, 0);

/* Tests du sélecteur */
if (Selector.TI == 1   /* descripteur dans LDT */)
   SegmentException($GP, 0);

if ( (Selector & OFFFCh) == 0   /* Nul */)
   /* Possibilité de charger un sélecteur Nul dans LDTR */
   /* pour indiquer pas de LDT                          */
   {LDTR.Selector = Selector;
   LDTR.Attributes.P = 0;
   return;
   }

/* Après les tests du Sélecteur, lecture et test du    */
/* Descripteur. Segment système ou guichet obligatoire,*/
/*  du type LDT.*/
ReadDescriptor(Selector,&Attributes,&Base,&Limit,&GSelector,
               &GOffset);
if ( (Attributes.DType == 1)   /* Segment mémoire */
      || (Attributes.Type != 2       /* LDT */) )
   SegmentException($GP, 0);

if (Attributes.P == 0)
   SegmentException($NP, 0);

/* Chargement du descripteur dans les registres cachés, */
/* et du sélecteur dans la partie visible de LDTR.      */
LDTR.Attributes   = Attributes;
LDTR.Base         = Base;
LDTR.Limit        = Limit;
LDTR.Selector     = Selector;
}   /* fin de LLDT */
```

Listing 5.10 : *Description de l'instruction LLDT (fin).*

```
/* Description de l'instruction LTR */
LTR(Selector)
   SelectorType   Selector;
{
   SegAttributes  Attributes;
   SelectorType   GSelector;
   int            Base, Limit, GOffset;

/* Niveau 0 obligatoire pour exécuter l'instruction LTR */
if (CPL != 0)
   SegmentException($GP, 0);

/* Tests de sélecteurs */
if ( (Selector.TI == 1   /* descripteur dans LDT */)
   || ((Selector & OFFFCh) == 0   /* Nul */) )
   SegmentException($GP, 0);

/* Si les tests de Sélecteur réussissent, lecture et test */
/* du Descripteur.                                        */
/* Doit être un segment système ou guichet, et du type    */
/* TSS 286 Disponible ou TSS 386 Disponible               */
ReadDescriptor(Selector,&Attributes,&Base,&Limit,&GSelector,
               &GOffset);
if ( (Attributes.DType == 1)       /* Segment mémoire */
   || ( (Attributes.Type != 1      /* TSS 286 Disponible */
      &&(Attributes.Type != 9      /* TSS 386 Disponible */
   SegmentException($GP, 0);

if (Attributes.P == 0)
   SegmentException($NP, 0);

/* Marque le TSS Occupé en changeant le champ type */
/* du descripteur */
AccessLinear(GDT.Base+Selector.Index*8+4,4,0 /* PL */,
             0 /*Lecture */,&Dword2);
```

Listing 5.11 : *Description de l'instruction LTR.*

```
Dword2 = Dword2 | 200h;
/* positionnement du bit 1 du champ type */
AccessLinear(GDT.Base+Selector.Index*8+4,4,0 /* PL */,
            1 /*Ecriture*/,&Dword2);

/* Chargement du descripteur dans les registres cachés, */
/* et du sélecteur dans la partie visible de TR.        */
TR.Attributes    = Attributes;
TR.Base          = Base;
TR.Limit         = Limit;
TR.Selector      = Selector;
} /* fin de LTR */
```

Listing 5.11 : *Description de l'instruction LTR (fin).*

Transfert de contrôle

Ce paragraphe décrit chaque instruction de transfert de contrôle intersegment à l'aide de programmes en pseudo-C. Cette description donne une vue orthogonale des méthodes de transfert de contrôle décrites au début de ce chapitre. Elle utilise certaines routines données dans les paragraphes précédents sur la pagination et le chargement des registres segments.

Le Sous-programme CSDescriptorLoad() du Listing 5.12 est employé par toutes les instructions de transfert de contrôle pour tester le descripteur qui doit être chargé dans les registres cachés de CS. CSDescriptorLoad() charge uniquement les registres cachés de CS. C'est à la routine appelante de charger la partie sélecteur du registre CS.

Le Sous-programme CSDescriptorLoad() compare le champ Type et le DPL du descripteur avec le CPL et le champ RPL du sélecteur pour déterminer si le descripteur est valide. Ces comparaisons sont effectuées pour un transfert direct dans un segment mémoire exécutable et présent, avec un

test du niveau de privilège différent si le segment donné est normal ou conforme. C'est un test approprié pour les instructions JMP et CALL directement sur des segments exécutables ou pour des instructions RET qui ne modifient pas les niveaux de privilège. Des transferts plus complexes changeront les niveaux de privilège ou les tâches avant d'appeler cette routine, si bien que les tests restent valides.

CSDescriptorLoad() comprend un test du RPL du sélecteur CS. Puisque les transferts à travers des guichets ignorent le champ RPL du sélecteur lu dans le guichet, les programmes simulant ces transferts doivent positionner le champ RPL à 0 avant tout appel à CSDescriptorLoad().

```
/* Routine courante pour charger un descripteur dans les   */
/* registres cachés de CS, et définit un segment mémoire   */
/* exécutable avec DPL=CPL et DPL   RPL, ou un segment     */
/* conforme avec DPL   CPL. Seuls les registres cachés de  */
/* CS sont chargés (base, limite, attributs). Un sélecteur */
/* CS doit être chargé par l'appelant.                     */
CSDescriptorLoad(Selector, Attributes, Base, Limit, GPorTS)
   SelectorType   Selector;
   SegAttributes  Attributes;
int   Base, Limit,
      GPorTS;   /* exception $GP ou $ST  */
                /* autre que non-présent.*/
{
switch   (Attributes.Type  /* Ignore l'attribut "Accessed" */
   case 8:   case 9:   /* Exécution */
   case 10:  case 11:  /* Lecture   */
/* Segment non conforme, teste DPL relativement */
/* à CPL et RPL */
      if ((Attributes.DPL !=CPL)
         || (Attributes.DPL < Selector.RPL))
         SegmentException(GPorTS, Selector);
```

Listing 5.12 : *Sous-programme CSDescriptorLoad().*

```
      break;
   case 12:  case 13:  /* Exécution, conforme          */
   case 14:  case 15:  /* Exécution/lecture, conforme */
      /* Segment conforme, teste DPL relativement à CPL    */
      if (Attributes.DPL > CPL)
         SegmentException(GPorTS, Selector);
      break;

   case 0:   case 1:   /* Lecture                 */
   case 2:   case 3:   /* Lecture/Ecriture        */
   case 4:   case 5:   /* Lecture, Expand-down */
   case 6:   case 7:   /* Lecture/Ecriture        */
    /* Type pas OK, rapporte violation segment */
      SegmentException(GPorTS, 0);
   } /*

/* Test du bit Présent.*/
if (Attributes.P == 0)
   SegmentException($NP, Selector);

/* On arrive ici seulement si tous les tests OK.           */
/* Positionne le bit "Accessed" puis charge le descripteur */
/* dans les registres cachés, et retour.                   */

SetAccessed(Selector);

CS.Attributes   = Attributes;
CS.Base         = Base;
CS.Limit        = Limit;

} /* fin de CSDescriptorLoad */
```

Listing 5.12 : *Sous-programme CSDescriptorLoad() (fin).*

L'instruction JMP

La routine JMP() du Listing 5.13 décrit l'instruction JMP. Elle reçoit deux paramètres, le sélecteur et le déplacement de la destination du saut. Si le sélecteur n'est pas nul, elle lit le descripteur associé grâce à la routine ReadDescriptor() du Listing 5.7. Les tests du descripteur sont séparés en deux parties, selon le bit DType.

```
/* Description de l'instruction JMP.*/
JMP(Selector, Offset)
   SelectorType   Selector;
   int   Offset;
{
   SegAttributes   Attributes;
   SelectorType    GSelector;
   int   Base, Limit, GOffset;

/* Tests du sélecteur pour JMP */
if ( (Selector & OFFFCh) == 0)   /* Nul */
   SegmentException($GP, 0);

/* Lecture et test du descripteur */
ReadDescriptor(Selector,&Attributes,&Base,&Limit,&GSelector,&GOffset);
if (Attributes.DType == 1)   /* Segment Mémoire */ {
   /* Appel classique pour compléter le chargement */
   /* du Descripteur de CS.                        */
   CSDescriptorLoad(Selector, Attributes, Base, Limit, $GP);
   /* Vérifie que la destination est comprise dans */
   /* la limite du segment                         */
   if (Offset > CS.Limit)
      SegmentException($GP, 0);
   /* On arrive à ce stade seulement si tous les   */
   /* tests de protection réussissent.             */
   /* Achève l'instruction avec les chargements du */
   /* registre "visible".                          */
```

Listing 5.13 : *Description de l'instruction JMP.*

```
   CS.Selector        Selector;
   CS.Selector.RPL    = CPL;
   EIP                = Offset;
   } /* fin du Segment Mémoire */

else   /* Segment système ou Guichet */ {
   /* Teste le DPL relativement au CPl et au RPL */
   if ( (Attributes.DPL < CPL) || (Attributes.DPL < Selector.RPL) )
   SegmentException($GP, Selector);

   switch (Attributes.Type){
   case 1:   /* TSS 286 Disponible */
      if (Attributes.P == 0)
         SegmentException($NP, Selector);
      TaskSwitch286(Selector, Attributes, Base, Limit, 0
               /* Pas de lien */);
      /* Voir Chapitre 9*/
      break;

   case 5:   /* Guichet de tâche */
      if (Attributes.P == 0)
         SegmentException($NP, Selector);
         TaskGate(GSelector, 0   /* Pas de lien */ );
      break;

   case 9:   /* TSS 386 Disponible */
      if (Attributes.P == 0)
         SegmentException($NP, Selector);
      TaskSwitch(Selector, Attributes,Base,Limit,0
                /* Pas de lien */);
      break;
   case 4:   /* Guichet d'Appel 286 */
      if (Attributes.P == 0)
         SegmentException($NP, Selector);
      JumpGate286(GSelector, GOffset, $GP);
```

Listing 5.13 : *Description de l'instruction JMP (suite).*

```
            /* voir le Chapitre 9.*/
         break;

      case 12:   /* Guichet d'Appel 386 */
         if (Attributes.P == 0)
            SegmentException($NP, Selector);
         JumpGate(GSelector, GOffset, $GP);
         break;

      Default:
         /* D'autres Types génèrent des exceptions $GP */
         SegmentException($GP, Selector);
      }
   } /* fin du segment Système ou Guichet */

   } /* fin de JMP */
```

Listing 5.13 : *Description de l'instruction JMP (fin).*

Si DType = 1, on a affaire à un segment mémoire, et on appelle la routine CSDescriptorLoad() du Listing 5.12 pour achever les tests du descripteur. Elle rend la main si elle a pu charger le descripteur. Si le nouveau déplacement est compris dans la limite de CS, l'instruction s'achève sur le chargement du registre sélecteur de CS, en positionnant le RPL de ce sélecteur à CPL, et en chargeant EIP avec le déplacement précisé par l'instruction.

Si DType = 0, on a affaire à un segment système ou à un guichet. Tout d'abord, l'instruction teste le champ DPL pour s'assurer qu'il est au même niveau ou à un niveau interne relativement à CPL et à RPL. Puis, elle teste le champ Type, et agit différemment selon chaque type valide (le bit P doit être égal à 1 pour indiquer que l'élément du descripteur est valide).

Si le champ Type du descripteur indique qu'on a un segment du type TSS 286 Disponible, TSS 386 Disponible, ou guichet de tâche, il y aura commutation de tâches sans action sur la chaîne des tâches suspendues.

Trois autres routines détaillent les commutations de tâches. TaskGate() et TaskSwitch() seront décrites ultérieurement dans ce chapitre. Nous ne détaillerons pas le Sous-programme TaskSwitch286() qui effectue une commutation de tâches vers un TSS 286. Dans le Chapitre 9, nous expliquerons les différences entre une commutation de tâches sur un TSS 286 et la commutation de tâches 386 décrite par la routine TaskSwitch().

Saut à travers un guichet d'appel

Un descripteur du type guichet d'appel 286 ou guichet d'appel 386 spécifie un saut à travers un guichet d'appel. La routine JumpGate() du Listing 5.14 décrit un saut à travers un guichet d'appel 386. Elle reçoit trois paramètres :

1 Le sélecteur du guichet.

2 Le déplacement du guichet.

3 GP ou TS, qui indiquent si on doit générer des exceptions du type $GP ou $TS. Ce paramètre permet au programme de simulation des commutations de tâches d'utiliser cette routine.

Tout d'abord, la routine teste le sélecteur pour s'assurer qu'il n'est pas nul. Puis elle lit le descripteur dans la table des descripteurs. Si son champ DType indique que c'est un segment mémoire, il est transmis à la routine CSDescriptorLoad() du Listing 5.12. Puisque le champ RPL du sélecteur obtenu à partir du guichet est ignoré, il est positionné à 0 pour annuler le test RPL de cette routine. Si CSDescriptorLoad() charge le descripteur, elle rend la main. Si le nouveau déplacement est compris dans la limite de CS, l'instruction s'achève en chargeant le registre sélecteur de CS, en positionnant le RPL du sélecteur de CS à CPL, et en chargeant EIP avec le déplacement obtenu à partir du guichet.

```
JumpGate(Selector, Offset, GPorTS)
   SelectorType   Selector;
   int   Offset,
   int   GPorTS;
/*13 ou 10 pour sélectionner les exceptions GP ou TS*/
/* sauf pour les exceptions du type non présent     */
{
   SegAttributes      Attributes;
   SelectorType       Gselector;
   int   Base, Limit, GOffset;

/* Tests du sélecteur pour JMP */
if ( (Selector & OFFFCh) == 0)   /* Nul */
   SegmentException(GPorTS, 0);

/* Lecture et test du descripteur */
ReadDescriptor(Selector,&Attributes,&Base,&Limit,&GSelector,&GOffset);

if (Attributes.DType == 0)
/* Ne peut pas être un segment Système */
                        /* ou un guichet */
   SegmentException(GPorTS, Selector);

/* Appel classique pour compléter le chargement du */
/* descripteur de CS  */
Selector.RPL = 0;
/* ignore RPL dans la lecture du sélecteur */
CSDescriptorLoad(Selector, Attributes, Base, Limit, GPorTS);
/* Vérifie si la destination est comprise dans les */
/* limites du segment */
if (Offset > CS.Limit)
   SegmentException(GPorTS, 0);

/* Partie visible de l'instruction JMP */
CS.Selector     = Selector;
CS.Selector.RPL = CPL;
EIP     = Offset;
} /* fin de JumpGate */
```

Listing 5.14 : *Sous-programme JumpGate().*

Nous ne donnons pas la description de la routine JumpGate286(), qui effectue un saut à travers le guichet d'appel 286. Dans le Chapitre 9, nous expliquerons en quoi un saut à travers un guichet d'appel 286 diffère d'un saut à travers un guichet d'appel 386 (voir la routine JumpGate()).

L'instruction CALL

La routine CALL() du Listing 5.15 décrit l'instruction CALL. Sa structure est identique à la routine JMP(). Elle reçoit deux paramètres : le sélecteur et le déplacement de la destination de l'appel. Si le sélecteur n'est pas nul, le descripteur associé est lu puis testé d'après l'état du type DType.

```
/* Description de l'instruction CALL Inter-Segment.*/
CALL(Selector, Offset)
   SelectorType   Selector;
   int   Offset;
{
   SegAttributes   Attributes;
   SelectorType    GSelector;
   int   Base, Limit, GOffset;

/* Tests du sélecteur pour CALL */
if ( (Selector & OFFFCh) == 0)    /* Nul */
   SegmentException($GP, 0);

/* Lecture et test du descripteur */
ReadDescriptor(Selector,&Attributes,&Base,&Limit,&GSelector,&GOffset);
if (Attributes.DType == 1)    /* Segment Mémoire */ {
/* Appel classique pour compléter le chargement */
/* du Descripteur CS    */
```

Listing 5.15 : *Description de l'instruction CALL.*

```
   CSDescriptorLoad(Selector, Attributes, Base, Limit, $GP);
/* Vérifie si la destination est comprise dans  */
/* la limite du segment */
   if (Offset > CS.Limit)
      SegmentException($GP, 0);

   /* On arrive ici seulement si tous les tests de */
   /* protection réussissent.                      */
   /* Empile le pointeur de retour, puis charge    */
   /* le sélecteur de CS et EIP                    */
   ESP = ESP-4;
   /* Empile 4 octets, avec le sélecteur dans les  */
   /* 2 octets inférieurs                          */
   AccessVirtual(SS, ESP, 4, 1  /* Ecriture */, $SS,
                 &CS.Selector);
   ESP = ESP-4;
   AccessVirtual(SS, ESP, 4, 1  /* Ecriture */, $SS, &EIP);

   CS.Selector      = Selector;
   CS.Selector.RPL  = CPL;
   EIP        = Offset;
   } /* Fin de Segment Mémoire */
else /* Segment système ou Guichet */ {
   /* Teste DPL relativement à CPL et RPL */
   if( (Attributes.DPL < CPL)
        || (Attributes.DPL < Selector.RPL) )
       SegmentException($GP, Selector);

   switch   (Attributes.Type){
   case 1:   /* TSS 286 Disponible */
      if (Attributes.P == 0)
         SegmentException($NP, Selector);
      TaskSwitch286(Selector, Attributes, Base, Limit, 1
                    /* Lien */);
              /* voir le Chapitre 9.*/
```

Listing 5.15 : *Description de l'instruction CALL (suite).*

```
      break;

   case 5:   /* Guichet de Tâche */
      if (Attributes.P == 0)
         SegmentException($NP, Selector);
      TaskGate(GSelector, 1 /* Lien */);
      break;

   case 9:   /* TSS 386 Disponible */
      if (Attributes.P == 0)
         SegmentException($NP, Selector);
      TaskSwitch(Selector, Attributes, Base, Limit, 1
                 /* Lien */);
      break;

   case 4:   /* Guichet d'Appel 286 */
      if (Attributes.P == 0)
         SegmentException($NP, Selector);
      CallGate286(GSelector, GOffset, Attributes.DwordCount);
         /* voir le Chapitre 9.*/
      break;

   case 12:  /* Guichet d'Appel 386 */
      if (Attributes.P == 0)
         SegmentException($NP, Selector);
      CallGate(GSelector, GOffset, Attributes.DwordCount);
      break;

   Default:   /* D'autres génèrent des exceptions $GP */
      SegmentException($GP, Selector);
   }
} /* fin du segment Système ou Guichet */

} /* fin de CALL */
```

Listing 5.15 : *Description de l'instruction CALL (fin).*

Si DType = 1, on a affaire à un segment mémoire, et on appelle la routine CSDescriptorLoad() du Listing 5.12 pour poursuivre les tests de descripteur. Elle rend la main si elle a réussi à charger le descripteur. Si le nouveau déplacement est compris dans la limite de CS, l'instruction s'achève par la partie "visible" de l'instruction. Le sélecteur de l'ancien CS est empilé, puis EIP est empilé pour avoir l'adresse de retour. Finalement, le registre sélecteur de CS est chargé, son RPL est positionné à CPL, et EIP est chargé avec le déplacement donné par l'instruction.

Si DType = 0, on a affaire à un segment système ou à un guichet. Tout d'abord, l'instruction teste le champ DPL pour s'assurer qu'il est au même niveau ou à un niveau interne relativement à CPL et à RPL. Puis, elle teste le champ Type, et agit différemment selon chaque type valide (bit P égal à 1), indiquant que l'élément du descripteur est valide.

Si le champ Type du descripteur indique qu'on a un segment du type TSS 286 Disponible, TSS 386 Disponible, ou guichet de tâche, il y aura commutation de tâches sans influence sur la chaîne des tâches suspendues.

Nous ne détaillerons pas le Sous-programme TaskSwitch286() qui effectue une commutation de tâches vers un TSS 286. Dans le Chapitre 9, nous expliquerons les différences entre une commutation de tâches sur un TSS 286 et la commutation de tâches 386 (voir la routine TaskSwitch()).

Appel à travers un guichet d'appel

Un descripteur du type guichet d'appel 286 ou guichet d'appel 386 spécifie un saut à travers un guichet d'appel. La routine CallGate() du Listing 5.16 décrit un saut à travers un guichet d'appel 80386. Elle reçoit trois paramètres :

1 Le sélecteur du guichet.

2 Le déplacement du guichet.

3 Le nombre de dmots à partir du guichet, qui donne le nombre de dmots à copier s'il y a commutation de tâches.

Tout d'abord, la routine teste le sélecteur pour s'assurer qu'il n'est pas nul. Puis, elle lit le descripteur dans la table des descripteurs, et teste son champ DType pour s'assurer que c'est un segment mémoire (et non un segment système ou un guichet). Puis elle teste les attributs du descripteur pour déterminer s'il doit y avoir un changement de niveau de privilège. Si le descripteur indique un segment exécutable, non conforme et présent avec un DPL à un niveau interne relativement au CPL, il y a changement du niveau de privilège.

Nous ne donnons pas la description de la routine CallGate286(), qui effectue un saut à travers le guichet d'appel 286. Dans le Chapitre 9, nous expliquerons en quoi un saut à travers un guichet d'appel 286 diffère d'un saut à travers un guichet d'appel 386 (voir la routine CallGate()).

```
CallGate(Selector, Offset, DwordCount)
   SelectorType   Selector;
   int   Offset,
         DwordCount;/* Nombre de paramètres dmots à copier */
                    /* pour des calls inter-niveaux */
{
   SegAttributes      Attributes;
   SelectorType       GSelector;
   int   Base, Limit, GOffset;

/* Tests du sélecteur pour CALL */
if ( (Selector & OFFFCh) == 0)    /* Nul */
   SegmentException($GP, 0);

/* Lecture et test du descripteur */
ReadDescriptor(Selector,&Attributes,&Base,&Limit,&GSelector,
               &GOffset);
```

Listing 5.16 : *Sous-programme CallGate().*

```
if (Attributes.DType == 0)
/* Ne peut pas être un segment Système ni un guichet */
   SegmentException($GP, Selector);

/* le CALL à un segment présent, exécutable, non conforme */
/* de niveau interne est OK, mais requiert commutation    */
/* vers pile interne et CPL interne                       */
if ( (Attributes.Type>=8) && (Attributes.Type<=11)
      && (Attributes.DPL < CPL) && (Attributes.P == 1) )
          InnerStack(Attributes.DPL, DwordCount, Selector);

/* Appel classique pour terminer le chargement */
/* du descripteur de CS */
Selector.RPL = 0;
/* ignore RPL lors de la lecture du sélecteur */
CSDescriptorLoad(Selector, Attributes, Base, Limit, $GP);
/* Vérifie si la destination est comprise dans  */
/* la limite du segment */
if (Offset > CS.Limit)
   SegmentException($GP, 0);

/* On arrive ici seulement si tous les tests de protection */
/* réussissent. */
/* Empile le pointeur de retour, puis charge le sélecteur  */
/* de CS et EIP */
ESP = ESP-4;
/* Empile 4 octets, avec le sélecteur CS dans les 2 octets */
/* inférieurs */
AccessVirtual(SS, ESP, 4, 1  /* Ecriture */, $SS, &CS.Selector);
ESP = ESP-4;
AccessVirtual(SS, ESP, 4, 1  /* Ecriture */, $SS, &EIP);

CS.Selector     = Selector;
CS.Selector.RPL = CPL;
EIP             = Offset;
} /* fin de CallGate */
```

Listing 5.16 : *Sous-programme CallGate() (fin).*

Niveaux de privilège et commutation de piles

La routine InnerStack() du Listing 5.17 décrit les changements de niveaux de privilège et les commutations de piles associées. Elle reçoit trois paramètres :

1 NewCPL donne le nouveau niveau de privilège.

2 Le champ DmotCompt du guichet donne le nombre de paramètres dmots qui doivent être copiés.

3 Le NewCSSelector fournit le sélecteur de CS à partir du guichet, qui est passé comme code d'erreur pour les exceptions.

Tout d'abord, la routine initialise la pile de niveau interne en lisant le pointeur de pile de ce niveau à partir du TSS. Avant de charger SS avec le sélecteur et le descripteur de la pile interne, elle sauvegarde l'ancien sélecteur de SS, les registres descripteurs, et le CPL externe pour copier la pile. Puis elle donne au CPL la valeur du niveau interne, et charge SS en appelant la routine SRegLoad(). Elle doit modifier CPL avant d'appeler SRegLoad(), pour que les tests de protection de cette routine soient efficaces.

Le pointeur sur la pile de niveau externe est sauvegardé dans la pile interne en empilant les anciennes valeurs de SS et ESP. Puis la routine copie les paramètres dmots depuis la pile externe vers la pile interne. Cette opération se déroule dans le même ordre que l'empilage initial dans la pile externe, mais dans ce cas l'ordre n'est pas significatif. Ce qui l'est, par contre, c'est que les lectures à partir de la pile externe s'effectuent avec un CPL égal au CPL externe, et les écritures dans la pile interne avec un CPL égal au CPL interne. Ceci est indispensable pour assurer le bon fonctionnement des tests de la protection de la pagination. En outre, on évite ainsi toute défaillance de la protection qui se produirait lors de la lecture de la pile externe avec un niveau de privilège interne. Une fois que la routine a fini de copier les paramètres, ESP est positionné pour pointer sur le dernier paramètre empilé et la routine InnerStack() s'achève en rendant la main.

Le guichet d'appel

Une fois que le niveau de privilège et la pile ont été changés, on appelle la routine CSDescriptorLoad() du Listing 5.12 pour effectuer les tests du descripteur. Mais avant, le champ RPL du sélecteur obtenu par le guichet est positionné à 0. Par conséquent, RPL sera ignoré dans la suite des opérations. La routine rend la main si elle a réussi à charger le descripteur. Si le nouveau déplacement est compris dans la limite de CS, l'instruction s'achève en effectuant la partie "visible" de l'instruction. Elle empile l'ancien sélecteur de CS, puis EIP pour sauvegarder l'adresse de retour. Finalement, elle charge le registre sélecteur de CS, positionne son RPL à CPL, puis charge EIP avec le déplacement obtenu à partir du guichet.

```
/* Commutation de piles pour des changements inter-niveaux */
InnerStack(NewCPL, DwordCount, NewCSSelector)
   int    NewCPL,
/* Commutation de piles pour ce niveau de privilège */
         DwordCount;/* Nombre de paramètres dmots à copier */
                    /*dans la nouvelle pile */
   SelectorType   NewCSSelector;
/* sélecteur CS obligatoire */
/* pour code d'erreur.*/
{
   SelectorType      NewSSSelector;
   SegmentRegister   OldSS;
   int    TSSOffset, NewESP, tempESP, OldCPL, Dword1;

/* Lecture du nouveau SS et de ESP à partir de TSS.*/
TSSOffset = NewCPL*8 + 4;
if ( (TSSOffset + 7) > TR.Limit)
   SegmentException($TS, NewCSSelector);
AccessLinear(TR.Base+TSSOFFSET,0 /* PL 0 */,0 /* Lecture */,
             &NewESP);
```

Listing 5.17 : *Sous-programme InnerStack().*

```
AccessLinear(TR.Base+TSSOffset+4, 0 /* PL 0 */,
             0  /* Lecture */, &NewSSSelector);

/* Sauve les anciens SS et CPL pour utilisation future */
OldSS = SS;
OldCPL = CPL;

/* Charge SS avec le sélecteur de la nouvelle pile après */
/* passage au nouveau CPL et chargement du descripteur   */
/* dans les registres cachés.*/
CPL = NewCPL;
SRegLoad(SS, NewSSSelector, &TS);
/* retour si le chargement de SS a réussi.*/
SS.Selector = NewSSSelector;

/* Empile les anciens SS et ESP */
NewESP = NewESP - 4;
AccessVirtual(SS, NewESP, 4, 1 /* Ecriture */, $SS, &OldSS.Selector);
NewESP = NewESP - 4;
AccessVirtual(SS, NewESP, 4, 1 /* Ecriture */, $SS, &ESP);

/* Copie les paramètres. L'écriture dans la nouvelle pile */
/* utilise le nouveau CPL. La lecture dans l'ancienne pile*/
/* utilise l'ancien CPL */
/* L'ordre de copie des paramètres n'est pas important,   */
/* mais on  emploie l'ordre initial de la pile externe.   */
/* Avec cet ordre, nous allons sortir de la boucle avec   */
/* NewESP pointant sur le sommet de la nouvelle pile      */
tempESP = ESP + DwordCount*4;
for (i=1; i<=DwordCount; i++) {
   CPL = OldCPL;
   tempESP = tempESP - 4;
   AccessVirtual(OldSS, tempESP, 4, 0  /* Lecture */, $SS,
                 &Dword1);
   CPL = NewCPL;
```

Listing 5.17 : *Sous-programme InnerStack() (suite).*

```
   NewESP = NewESP - 4;
   AccessVirtual(SS, NewESP, 4, 1  /* Ecriture */, $SS,
                 &Dword1);
   }

ESP = NewESP;
/* Sortie de boucle avec NewESP pointant */
/* sur le nouveau sommet de pile*/
} /* fin de InnerStack */
```

Listing 5.17 : *Sous-programme InnerStack() (fin).*

L'instruction RET

La routine RET() du Listing 5.18 décrit l'instruction RET. Elle reçoit comme paramètre le nombre de paramètres octets à enlever de la pile après dépilage de l'adresse de retour. Il existe deux formes d'instruction RET intersegment. La première comprend un champ immédiat de 16 bits contenant ce nombre de paramètres octets. La seconde n'a pas de champ immédiat (nombre de paramètres octets nul).

```
/* Description de l'intersegment RET */
RET(ParmCount)
   int   ParmCount;
   /* nombre de paramètres octets à retirer */
                              /* de la pile.*/
{
```

Listing 5.18 : *Description de l'instruction RET.*

```
   SelectorType       Selector, GSelector;
   SegAttributes      Attributes;
   int    Base, Limit, Offset, GOffset;

/* Dépile l'adresse de retour */
AccessVirtual(SS, ESP, 4, 0  /* Lecture */, $SS, &Offset);
ESP = ESP + 4;
AccessVirtual(SS, ESP, 4, 0  /* Lecture */, $SS, &Selector);
ESP = ESP + 4;
/* Retire les paramètres octets de la pile */
ESP = ESP + ParmCount;

/* Tests du sélecteur pour RET */
if ( ((Selector & 0FFFCh) == 0) || (Selector.RPL < CPL))
   SegmentException($GP, 0);
if (Selector.RPL > CPL) {
/* RET inter-niveaux est requis si Selector.RPL > CPL.  */
/* Appelle le Sous-programme pour récupérer la pile de  */
/* niveau externe à partir du pointeur de pile SS:ESP   */
/* situé maintenant au sommet de la pile de niveau      */
/* interne */
   OuterStack(Selector.RPL, ParmCount);
   }

/* Lecture et test du descripteur de CS */
ReadDescriptor(Selector,&Attributes,&Base,&Limit,&GSelector,
               &GOffset);
if (Attributes.DType == 0)
/* Ne peut pas être un segment Système */
                      /* ni un guichet */
   SegmentException($GP, Selector);

/* Appel classique pour compléter le chargement */
/* du descripteur de CS.*/
CSDescriptorLoad(Selector, Attributes, Base, Limit, $GP);
```

Listing 5.18 : *Description de l'instruction RET (suite).*

```
/* Vérifie si la destination est comprise */
/* dans la limite du segment */
if (Offset > CS.Limit)
   SegmentException($GP, 0);

/* On arrive ici seulement si tous les tests de   */
/* protection réussissent.                        */
/* Termine sur la partie visible de l'instruction */
/* en chargeant le sélecteur de CS et EIP.        */
CS.Selector = Selector;
EIP = Offset;

} /* fin de RET */
```

Listing 5.18 : *Description de l'instruction RET (fin).*

Tout d'abord, l'instruction dépile le pointeur de retour, et incrémente ESP du nombre de paramètres octets. Puis, elle teste le sélecteur de CS qu'elle vient de dépiler. Le champ RPL définit le niveau de privilège auquel il faut retourner. Une exception est générée si le sélecteur est nul, ou si son champ RPL spécifie un niveau interne relativement au CPL. Si son champ RPL indique un niveau externe relativement au CPL, l'instruction effectue un retour au niveau externe indiqué.

Commutation de niveaux de privilège et de piles

La routine OuterStack() du Listing 5.19 est appelée lors de commutations de piles et de niveaux de privilège. Elle reçoit deux paramètres :

1 NewCPL donne le nouveau niveau de privilège.

2 ParmCount donne le nombre de paramètres octets à enlever de la pile externe après la commutation des piles.

Tout d'abord, le pointeur de la pile de niveau externe est dépilé de la pile de niveau interne. Puis la routine donne à CPL la valeur du niveau externe, et charge le registre SS en appelant la routine SRegLoad(). CPL est modifié avant d'appeler SRegLoad() pour que les tests de protection de cette routine soient efficaces. Puis elle charge ESP avec la valeur dépilée de la pile interne après y avoir ajouté le nombre de paramètres octets (pour enlever les paramètres de la pile externe). Pour finir, elle teste les segments adressés par DS, ES, FS et GS pour s'assurer qu'ils sont accessibles au niveau externe. Chaque registre qui n'est pas accessible est chargé avec un sélecteur nul, et le bit P du registre caché est effacé pour interdire tout accès à ce registre avant qu'un nouveau sélecteur soit chargé.

Fin de l'instruction RET

Une fois que le niveau de privilège et la pile ont été changés, on appelle la routine CSDescriptorLoad() du Listing 5.12 pour effectuer les tests du descripteur. Puisque CPL a la valeur du champ RPL du sélecteur, le test du RPL dans CSDescriptorLoad() ne pose pas de problème. La routine rend la main si elle a réussi à charger le descripteur. Si le nouveau déplacement est compris dans la limite de CS, l'instruction s'achève en effectuant la partie "visible" de l'instruction, c'est-à-dire en chargeant CS et EIP avec le sélecteur et le déplacement qui ont été dépilés.

```
/* Sous-programme de commutation vers une pile */
/* de niveau externe */
OuterStack(NewCPL, ParmCount)
   int   NewCPL,
```

Listing 5.19 : *Sous-programme OuterStack().*

```
/* CPL du niveau externe auquel on retourne */
        ParmCount;   /* nombre de paramètres à dépiler*/
{
   SelectorType   NewSSSelector;
   int   NewESP;

/* Dépile les SS et ESP externes de la pile */
/* de niveau interne */
   AccessVirtual(SS, ESP, 4, 0  /* Lecture */, $SS, $NewESP);
   ESP = ESP + 4;
   AccessVirtual(SS, ESP, 4, 0  /* Lecture */, $SS,
                 $NewSSSelector);
   ESP = ESP + 4;

   /* Charge SS avec le sélecteur de la pile externe */
   /* après changement du CPL */
   CPL = NewCPL;
   SRegLoad(SS, NewSSSelector, $GP);
   /* retour si le chargement de SS s'est bien passé */
   SS.Selector = NewSSSelector;

   /* Charge ESP avec le ESP externe incrémenté du nombre */
   /* de paramètres octets */
   ESP = NewESP + ParmCount;

   /* Vérifie que les segments adressés par DS, ES, FS */
   /* sont accessibles au nouveau niveau de privilège. */
   /* Un segment est accessible s'il est conforme      */
   /* quel que soit DPL, ou sinon si DPL ≥ au nouveau   */
   /* CPL. Dans le cas contraire, charge Null pour     */
   /* éviter tout problème de protection. Pas besoin de*/
   /* tester RPL, puisqu'on ne peut pas charger le */
   /* registre au niveau interne si RPL > au nouveau   */
   /* CPL > à l'ancien CPL                             */
```

Listing 5.19 : *Sous-programme OuterStack() (suite).*

```
if ( (DS.Attributes.Type < 12)
      && (DS.Attributes.DPL < CPL) )
   {DS.Selector = 0; DS.Attributes.P = 0;}
if ( (ES.Attributes.Type < 12)
      && (ES.Attributes.DPL < CPL) )
   {ES.Selector = 0; ES.Attributes.P = 0;}
if ( (FS.Attributes.Type < 12)
      && (FS.Attributes.DPL < CPL) )
   {FS.Selector = 0; FS.Attributes.P = 0;}
if ( (GS.Attributes.Type < 12)
      && (GS.Attributes.DPL < CPL) )
   {GS.Selector = 0; GS.Attributes.P = 0;}
} /* fin de OuterStack */
```

Listing 5.19 : *Sous-programme OuterStack() (fin).*

Commutations de tâches

Les instructions CALL et JMP peuvent aussi effectuer un transfert de contrôle vers une tâche différente, comme les interruptions et l'instruction IRET (voir le Chapitre 6). Le processeur 80386 peut traiter une commutation complète de tâches en exécutant ces instructions. Le type de segment système TSS qui gère l'état d'une tâche le permet. Le TSS contient une zone de sauvegarde des registres, un champ qui permet de lier les tâches entre elles, et des pointeurs sur les tables de segmentation et de pagination. Ainsi, le 80386 sauvegarde ou charge les registres du processeur, lie les tâches suspendues, et modifie le mapping (adresse virtuelle en adresse physique), ce qui permet de gérer totalement la commutation de tâches.

Cette capacité à commuter des tâches en une instruction unique n'ajoute rien de plus aux possibilités du jeu d'instructions du 80386. Les fonctions de commutation de tâches décrites ici peuvent être implémentées par un logiciel en utilisant une séquence d'instructions très simples. Cependant, la

gestion matérielle de la commutation des tâches est bien plus efficace qu'une simulation logicielle si le modèle de tâche du 80386 correspond exactement à celui du système d'exploitation. Si la différence entre les deux modèles de tâche est trop importante, il vaut mieux opter pour la solution logicielle.

Si le sélecteur de la destination d'une interruption, ou une instruction JMP, CALL, ou IRET identifie un TSS ou un guichet de tâche, il y a commutation de tâches vers le TSS indiqué ou vers le TSS pointé par le guichet de tâche. Les guichets de tâche sont identiques à des guichets d'appel puisqu'ils spécifient indirectement la destination du transfert de contrôle. Seule la partie sélecteur du pointeur d'un guichet d'appel est utilisée, ce qui doit identifier un TSS.

Les tests de privilège s'appliquent aux commutations de tâches pour s'assurer que le TSS ou le guichet d'appel sont accessibles. Ces tests sont les mêmes que ceux des guichets d'appel ou des chargements de segments données. Le TSS ou le guichet de tâche doit avoir son DPL au même niveau ou à un niveau externe relativement à CPL et à RPL. Le DPL du TSS pointé par un guichet de tâche n'est pas testé.

Le type du descripteur de TSS doit être approprié au type de transfert. Un TSS peut être soit occupé, soit disponible selon le champ Type de son descripteur. Une tâche est du type occupé si elle est la tâche courante ou si elle est chaînée à la tâche courante dans la liste des tâches suspendues selon le champ de lien de chaque TSS. Sinon, une tâche est du type disponible. Un CALL, un JMP ou une interruption doivent s'effectuer sur un TSS du type disponible. Un IRET rend la main à la tâche précédente dans la liste des tâches chaînées. Cette tâche doit être du type occupé.

Comme nous l'avons vu auparavant dans ce chapitre, on utilise le champ de liens du TSS pour construire la liste des tâches suspendues. La tâche courante est suspendue et est ajoutée au début de la liste des tâches suspendues s'il y a eu un appel ou une interruption vers une autre tâche. Une tâche est retirée de la liste des tâches suspendues lorsque le processeur rencontre un IRET correspondant. Cet IRET entraîne la poursuite de l'exécution de la dernière tâche suspendue.

Les commutations de tâches ne peuvent pas être récursives. Un CALL ou une interruption doivent s'effectuer sur un TSS du type disponible, et ces opérations laissent l'ancien TSS occupé, ce qui empêche toute récursivité. Une interruption qui peut générer des interruptions imbriquées ne peut pas être traitée par une commutation de tâches. Après avoir reçu une interruption traitée par une commutation de tâches, la tâche d'interruption doit s'achever et exécuter une instruction JMP ou IRET. Il faut rendre la tâche d'interruption disponible, avant que tout autre interruption de même origine puisse être acceptée.

Le bit NT du registre EFLAGS indique la présence d'une tâche suspendue liée au TSS courant. L'instruction IRET fait référence à ce bit pour déterminer s'il doit y avoir commutation de tâches avant le retour, ou si le retour peut s'effectuer vers un programme de la tâche courante. Si NT = 1, IRET suivra le champ de liens pour reprendre l'exécution de la dernière tâche suspendue. Si NT = 0, IRET dépile l'adresse de retour pour revenir à un programme de la tâche courante.

Un programme peut modifier la valeur du bit NT quel que soit son niveau de privilège. S'il positionne le bit NT à 1 puis exécute un IRET, il y aura commutation de tâches vers le TSS pointé par le champ de liens du TSS courant. Si ce champ de liens n'a pas été positionné par un appel ou une interruption correspondants, les résultats sont imprévisibles. Par conséquent, le système d'exploitation doit initialiser le champ de liens de chaque TSS avant d'effectuer un saut à ce TSS. De cette façon, un programme, par malice ou par erreur, pourra modifier le bit NT et exécuter un IRET, mais le système d'exploitation gardera toujours le contrôle.

Le guichet de tâche

La routine TaskGate() du Listing 5.20 décrit les transferts effectués à travers des guichets de tâche. Elle reçoit deux paramètres :

1 Le sélecteur à partir du descripteur de guichet de tâche.

2 Une information sur le chaînage, qui contrôle le traitement de la liste des tâches suspendues.

Le sélecteur ne doit pas être nul, et doit déterminer un descripteur dans la GDT. Dans ce cas, la routine lit puis teste le descripteur. Ce dernier doit déterminer un segment système ou un descripteur de guichet dont le champ DType doit être nul. Puis cette routine teste le champ Type selon le paramètre de lien. Si le champ Type est valide, et si le bit P est égal à 1, elle appelle la routine TaskSwitch() pour effectuer la commutation de tâches.

Nous ne donnons pas la description du Sous-programme TaskSwitch286() qui effectue une commutation de tâches à travers un TSS 286. Dans le Chapitre 9, nous expliquerons en quoi une commutation de tâches à travers un TSS 286 diffère d'une commutation de tâches 386 (voir TaskSwitch()).

```
TaskGate(Selector, Linkage)
   SelectorType   Selector;
   int      Linkage;   /* 0 : pas de lien, 1 : lien */
                       /* -1 : libre */
{
   SegAttributes   Attributes;
   SelectorType    GSelector;
   int   Base, Limit, GOffset;

/* Le sélecteur doit identifier un TSS, rangé dans la GDT */
if ( ((Selector & OFFFCh) == 0  /* Nul */)
      || (Selector.TI == 1  /* dans LDT */) )
   SegmentException($GP, Selector);

ReadDescriptor(Selector,&Attributes,&Base,&Limit,&GSelector,
               &GOffset);
```

Listing 5.20 : *Sous-programme TaskGate().*

```
if (Attributes.DType == 1)  /* Segment Mémoire */
   SegmentException($GP, Selector);

switch (Attributes.Type) {
case 1:   /* TSS 286 Disponible */
   if (Linkage == -1   /* Libre */
      SegmentException($GP, Selector);
   if (Attributes.P == 0)
      SegmentException($NP, Selector);
   TaskSwitch286(Selector, Attributes, Base, Limit, Linkage);
   break;

case 3:   /* TSS 286 occupé */
   if (Linkage != -1   /* Libre */
     SegmentException($GP, Selector);
   if (Attributes.P == 0)
     SegmentException($NP, Selector);
   TaskSwitch286(Selector, Attributes, Base, Limit, Linkage);
   break;

case 9:   /* TSS 386 Disponible */
   if (Linkage == -1   /* Libre */
      SegmentException($GP, Selector);
   if (Attributes.P == 0)
      SegmentException($NP, Selector);
   TaskSwitch(Selector, Attributes, Base, Limit, Linkage);
   break;

case 11:   /* TSS 386 occupé */
   if (Linkage != -1   /* Libre */
      SegmentException($GP, Selector);
   if (Attributes.P == 0)
      SegmentException($NP, Selector);
   TaskSwitch(Selector, Attributes, Base, Limit, Linkage);
   break;
```

Listing 5.20 : *Sous-programme TaskGate() (suite).*

```
Default: /* D'autres segments système ou types de guichets */
         /* sont illégaux */
   SegmentException($GP, Selector);
}
} /* fin de TaskGate */
```

Listing 5.20 : *Sous-programme TaskGate() (fin).*

```
TaskSwitch(Selector, Attributes, Base, Limit, Linkage)
   SelectorType      Selector;
   SegAttributes     Attributes;
   int  Base, Limit,
            Linkage.  /* 0 : Pas de lien, 1 : lien */
                      /* -1 : Libre */
{
if (Limit < 103)
   SegmentException($TS, Selector);

/* Sauve le contexte dans le TSS de l'ancienne tâche */
AccessTSSState(1 /* Ecriture */);

/* TR et les registres cachés du nouveau TSS */
TR.Base       = Base;
TR.Limit      = Limit;
TR.Attributes = Attributes;
OldTSS        = TR.Selector; /* Sauve le sélecteur du TSS */
                             /* de l'ancienne tâche */
TR.Selector   = Selector;

/* Charge le contexte à partir du TSS de la nouvelle tâche */
```

Listing 5.21 : *Sous-programme TaskSwitch().*

```
AccessTSSState(0 /* Lecture */);

/* Traite les différents liens */
if (Linkage == 1   /* Lien */) {
/* Sauve le sélecteur de l'ancien TSS dans le nouveau TSS. */
/* Lève le bit NT. Laisse le descripteur de l'ancien TSS   */
/* occupé */
   AccessLinear(TR.Base,2,0 /* Niveau 0 */, 1 /* Ecriture */,
                &OldTS);
   EFLAGS.NT = 1;
   SetTSSBusy(Selector, 1);
   /* descripteur du nouveau TSS occupé */
   } /* fin de Link */
else if (Linkage == -1   /* Libre */) {
   SetTSSBusy(OldTSS, 0);
   /* descripteur de l'ancien TSS libre */
   } /* fin de Unlink */
else if   (Linkage = 0   /* pas de lien */) {
   SetTSSBusy(OldTSS, 0);
   /* descripteur de l'ancien TSS libre */
   SetTSSBusy(Selector, 1);
   /* descripteur du nouveau TSS occupé */
   } /* fin de NoLink */

CRO.TS = 1;       /* Positionne le bit TS */
CPL = CS.Selector.RPL;
   /* CPL à partir du RPL du sélecteur de CS */

/* Contexte visible récupéré. Charge les descripteurs dans */
/* les registres cachés pour LDTR et les registres         */
/* segments. Rend tous les descripteurs non valides        */
LDTR.Attributes.Present = 0;
CS.Attributes.Present = 0;
SS.Attributes.Present = 0;
DS.Attributes.Present = 0;
```

Listing 5.21 : *Sous-programme TaskSwitch() (suite).*

```
ES.Attributes.Present = 0;
FS.Attributes.Present = 0;
GS.Attributes.Present = 0;
/* Doit charger d'abord LDTR */
if (LDTR.Selector.TI == 1   /* dans la LDT */)
   SegmentException($TS, LDTR.Selector);
if ((LDTR.Selector & OFFFCh) = 0   /* Nul */) {
   /* OK si LDTR est nul */
   LDTR.Attributes.P = 0;
   }

else {
/* Lecture et test du descripteur si le sélecteur */
/* n'est pas nul */
   ReadDescriptor(LDTR.Selector,&Attributes,&Base,&Limit,
                  &GSelector,&GOffset);
   if ( (Attributes.DType == 1)
      || (Attributes.Type != 2   /* LDT */)
      || (Attributes.Present == 0) )
         SegmentException($TS, LDTR.Selector);
/* Charge les registres cachés de LDTR si tous les tests */
/* réussissent */
   SetAccessed(LDTR.Selector);
   LDTR.Attributes   = Attributes;
   LDTR.Base         = Base;
   LDTR.Limit        = Limit;
   } /* fin de LDTR.Selector non Nul */

/* Chargement du reste des registres cachés */
/* Le chargement de CS est identique à un   */
/* saut via un guichet d'appel              */
JumpGate(CS.Selector, EIP, $TS);

SRegLoad(SS, SS.Selector,$TS);
SRegLoad(DS, DS.Selector, $TS);
```

Listing 5.21 : *Sous-programme TaskSwitch() (suite).*

```
SRegLoad(ES, ES.Selector, $TS);
SRegLoad(FS, FS.Selector, $TS);
SregLoad(GS, GS.Selector, $TS);

/* Efface les bits locaux de DR7. Voir le Chapitre 8 */
DR7.L0 = 0;
DR7.L1 = 0;
DR7.L2 = 0;
DR7.L3 = 0;
DR7.L0 = 0;

} /* fin de TaskSwitch */
```

Listing 5.21 : *Sous-programme TaskSwitch() (fin).*

La commutation de tâches

La routine TaskSwitch() du Listing 5.21 décrit une commutation de tâches à travers un TSS 386. Avant d'appeler cette routine, il faut vérifier que le programme courant a un privilège suffisant pour commuter sur cette nouvelle tâche, et que cette dernière a un descripteur valide. Toute exception détectée à ce niveau sera prise en compte par la tâche courante et non par la nouvelle tâche. Les étapes restantes sont décrites dans TaskSwitch() :

1 Appel de la routine AccessTSSState() du Listing 5.22, pour sauver les valeurs courantes des registres généraux, des registres segments, de EIP et de EFLAGS dans le TSS courant. La valeur sauvegardée de EIP pointe sur l'instruction suivant celle qui a entraîné la commutation de tâches. Ceci sauvegarde le contexte de la tâche courante afin qu'il puisse être récupéré ultérieurement.

```
/* Sauve ou récupère le contexte dans le TSS */
AccessTSSState(RW)
   int   RW;   /* 0 si Lecture, 1 si Ecriture */
{
if (TR.Limit < 103)
   SegmentException($TS, TR.Selector);

AccessLinear(TR.Base+ 20h, 4, 0      /* Niveau 0 */,
             RW, &EIP);
AccessLinear(TR.Base+ 24h, 4, 0      /* Niveau 0 */,
             RW, &EFLAGS);
AccessLinear(TR.Base+ 28h, 4, 0      /* Niveau 0 */,
             RW, &EAX);
AccessLinear(TR.Base+ 2Ch, 4, 0      /* Niveau 0 */,
             RW, &ECX);
AccessLinear(TR.Base+ 30h, 4, 0      /* Niveau 0 */,
             RW, &EDX);
AccessLinear(TR.Base+ 34h, 4, 0      /* Niveau 0 */,
             RW, &EBX);
AccessLinear(TR.Base+ 38h, 4, 0      /* Niveau 0 */,
             RW, &ESP);
AccessLinear(TR.Base+ 3Ch, 4, 0      /* Niveau 0 */,
             RW, &EBP);
AccessLinear(TR.Base+ 40h, 4, 0      /* Niveau 0 */,
             RW, &ESI);
AccessLinear(TR.Base+ 44h, 4, 0      /* Niveau 0 */,
             RW, &EDI);
AccessLinear(TR.Base+ 48h, 2, 0      /* Niveau 0 */,
             RW, &ES.Selector);
AccessLinear(TR.Base+ 4Ch, 2, 0      /* Niveau 0 */,
             RW, &CS.Selector);
AccessLinear(TR.Base+ 50h, 2, 0      /* Niveau 0 */,
             RW, SS.Selector);
AccessLinear(TR.Base+ 54h, 2, 0      /* Niveau 0 */,
             RW, &DS.Selector);
```

Listing 5.22 : *Sous-programme AccessTSSState().*

```
AccessLinear(TR.Base+ 58h, 2, 0       /* Niveau 0 */,
             RW, &FS.Selector);
AccessLinear(TR.Base+ 5Ch, 2, 0       /* Niveau 0 */,
             RW, &GS.Selector);

/* Accès à CR3 et LDTR uniquement en lecture */
if (RW == 0   /* Lecture */) {
   AccessLinear(TR.Base+ 60h, 2, 0  /* Niveau 0 */,
                RW, &LDTR.Selector);
   AccessLinear(TR.Base+ 1Ch, 4, 0  /* Niveau 0 */,
                RW, &New CR3);
   if (NewCR3 != CR3) {
      /* Evite de purger TLB si le nouveau CR3 */
      /* est égal à l'ancien */
      FlushTLB;
      CR3 = NewCR3;
      } /* fin du chargement de CR3 */
   }

} /* fin de AccessTSSState */
```

Listing 5.22 : *Sous-programme AccessTSSState() (fin).*

2 Chargement de TR (le registre de tâche) avec le sélecteur du TSS de la nouvelle tâche, et chargement du descripteur dans les registres cachés. A ce stade, toute exception qui pourrait survenir serait prise en compte par la nouvelle tâche.

3 Appel de AccessTSSState() pour charger les registres généraux, les registres segments, EFLAGS, et EIP du nouveau TSS. Pour pouvoir traiter correctement les exceptions qui surviendraient lors du chargement des registres segments de la nouvelle tâche, tous les registres sélecteurs sont chargés avant les descripteurs. A ce stade, tous les registres cachés des descripteurs sont marqués *non présents*, puisque aucun descripteur n'a encore été chargé.

4 AccessTSSState() chargera aussi LDTR et CR3 du nouveau TSS. On travaille alors avec le mapping (adresse virtuelle en adresse physique) de la nouvelle tâche. Notez qu'aucun de ces registres n'a été écrit dans l'ancien TSS lors de la sauvegarde du contexte de l'étape 1. Puisque CR3 est modifié pendant l'accès au nouveau TSS, le mapping doit s'assurer qu'il traite les TSS de la même façon pour toutes les tâches.

5 Appel de la routine SetTSSBusy() du Listing 5.23 pour mettre à jour les champs Type dans les descripteurs des deux TSS. Ceux-ci seront du type occupé ou disponible selon le type de chaînage demandé. Si le paramètre de chaînage indique qu'il y a lien avec la chaîne des tâches suspendues, le bit NT du registre EFLAGS (chargé à partir du nouveau TSS) est positionné à 1 et le sélecteur de l'ancien TSS est placé dans le champ de liens du nouveau TSS.

6 Positionnement du bit TS dans CR0. La prochaine instruction du coprocesseur générera une trappe, si bien que les registres du coprocesseur pourront être sauvegardés dans l'ancienne tâche, et rechargés depuis la nouvelle tâche. La sauvegarde/récupération du contexte du coprocesseur est expliquée au Chapitre 7. Le CPL de la nouvelle tâche est égal au champ RPL du sélecteur de CS dans la nouvelle tâche. On peut avoir une commutation de tâches depuis une tâche de niveau quelconque vers une tâche de niveau quelconque (si le nouveau TSS ou le guichet de tâche du nouveau TSS sont accessibles).

7 Chargement des descripteurs correspondant aux sélecteurs précédemment chargés. Si une exception survient lors du chargement d'un descripteur, elle peut être générée avant que tous les sélecteurs soient chargés. Dans ce cas, ces registres segments contiendront les sélecteurs, mais le bit P sera égal à 0 pour en interdire l'accès. Les routines qui traitent les exceptions doivent traiter convenablement ce cas.

```
/* Routine pour changer le type dans un descripteur TSS.*/
SetTSSBusy(Selector, Busy)
   SelectorType   Selector;
   /* Sélecteur pour un descripteur TSS.*/
   int      Busy;
   /* 0 si Disponible, 1 si Occupé */
{
   int      dword1;

/* L'appelant a vérifié que le sélecteur n'est pas nul      */
/* et qu'il est dans la GDT. Lecture simple du descripteur */
/* et modification du bit 2 du champ Type                  */
AccessLinear(GDT.Base+Selector.Index*8 + 4, 4,
             0  /* PL 0 */,0 /* Lecture */, &dword1);
dword1 = dword1 & OFFFFFDFFh;
   /* Efface le bit 2 du champ type */
dword1 = dword1 | (Busy << 9);
AccessLinear(GDT.Base+Selector.Index*8 + 4, 4,
             0  /* PL 0 */,1 /* Ecriture */, &dword1);
} /* fin de SetTSSBusy */
```

Listing 5.23 : *Sous-programme SetTSSBusy().*

8 Les bits d'autorisation du registre de mise au point DR7 sont positionnés à 0 pour effacer tous les points d'arrêt, ou le mode exact propre à l'ancienne tâche. Dans le Chapitre 8, nous expliquerons l'emploi des registres de mise au point.

Chargement des registres segments lors d'une commutation de tâches

Comme on l'a déjà vu, le chargement des registres segments se déroule en deux étapes. Ce fonctionnement est nécessaire puisqu'une exception survenue lors du chargement d'un registre segment ne doit pas interrompre

la commutation de tâches, laissant le reste des registres segments avec des valeurs indéfinies.

Le premier niveau de l'étape 3 charge les registres sélecteurs avec les valeurs lues dans le TSS sans tester ces sélecteurs ni les descripteurs associés. Pour éviter d'obtenir des valeurs incongrues, le bit P est mis à 0 dans tous les registres cachés. Ainsi, on ne peut pas utiliser les registres segments pour adresser de la mémoire tant que la commutation de tâches n'est pas terminée, ou que le nouveau sélecteur n'est pas chargé avec succès.

Le second niveau de l'étape 6 charge les descripteurs pour chaque registre segment et effectue les tests de protection sur les sélecteurs et les descripteurs. A ce stade, toute exception mettra fin à la commutation de tâches. On obtient alors un ou plusieurs registres segments dont la partie sélecteur contient un sélecteur lu dans le TSS et non testé, mais avec P = 0 pour interdire tout accès. Ces sélecteurs non testés se comportent comme des sélecteurs nuls mais peuvent avoir des valeurs non nulles.

La plupart des systèmes 80386 mettront simplement fin à une tâche si le TSS contient une valeur de sélecteur fausse. Seuls les systèmes qui peuvent marquer des segments *non présents* devront s'occuper des exceptions du type segment lors d'une commutation de tâches.

Il existe deux méthodes pour poursuivre ou achever la commutation de tâches. La plus simple consiste à traiter ces exceptions par des guichets de tâche (voir le Chapitre 6). Puis, après avoir déterminé la cause de l'exception, l'IRET rendra la main à la tâche interrompue pour relancer la commutation de tâches. Ces routines de traitement des interruptions devraient vérifier toutes les valeurs des registres segments de la tâche interrompue pour que la commutation de tâches puisse se poursuivre correctement.

La seconde méthode consiste à traiter les exceptions par des guichets de trappe (voir le Chapitre 6), et à laisser la routine d'interruption empiler toutes les valeurs des registres segments et les dépiler juste après l'IRET. Lorsque le mauvais sélecteur est dépilé après le retour d'interruption, il est chargé avec succès, puisque la cause de l'exception a été déterminée. D'autres sélecteurs peuvent engendrer des exceptions lorsqu'ils sont dépi-

lés sans avoir été traités avant l'exception. Dans ce cas, le système d'exploitation doit être capable de tolérer des exceptions récursives juste avant l'IRET de ces exceptions.

6

Interruptions et exceptions

Outre les instructions classiques, les interruptions et les exceptions sont des méthodes particulières de transfert de contrôle. Plutôt que de passer d'une procédure à une autre par une instruction CALL, elles effectuent un transfert de contrôle sans employer d'instruction, comme si un CALL avait été inséré entre deux instructions. Le contrôle est donné à une procédure qui traite les exceptions ou les interruptions (*handler*) et qui peut revenir au programme interrompu grâce à une instruction IRET (voir le Chapitre 3). La prise en compte de l'interruption par le handler s'appelle le *handling* de l'interruption ou de l'exception.

Les *interruptions* sont générées par des événements externes et n'ont aucun rapport avec l'instruction qui s'exécute lorsque l'interruption est reçue. Généralement, on emploie une interruption pour indiquer qu'une unité E/S (disque, clavier, etc) a terminé la dernière opération ou a bien reçu des données provenant du processeur. Ces événements externes n'ont aucun lien avec l'instruction qui s'exécute lors de l'apparition de l'interruption. Cette dernière entraîne un transfert de contrôle vers un handler d'interruptions du système d'exploitation qui lance une nouvelle opération vers l'unité E/S, lit des données, effectue des tests, etc. Lorsque le handler d'interruptions a fini son travail, il rend la main par une instruction IRET au programme à l'endroit où ce dernier a été interrompu.

Les *exceptions* sont des conditions inhabituelles ou non valides détectées lors de l'exécution d'une instruction, et elles sont donc directement liées à cette instruction. Par exemple, les exceptions du type page ou segment (voir le Chapitre 5), qui sont détectées au cours d'une instruction, empêchent son exécution correcte. Les instructions INT *n* et INTO ("interruptions logicielles") sont considérées comme des exceptions plutôt que comme des interruptions, puisque c'est l'exécution de ces instructions qui cause l'exception. Dans le Chapitre 8, nous donnerons d'autres exemples d'exceptions qui facilitent la mise au point des programmes.

En général, les interruptions et les exceptions sont prises en compte entre deux instructions. Les instructions qui travaillent sur des chaînes avec un préfixe de répétition (voir le Chapitre 3) sont définies de façon particulière. En effet, elles mettent à jour les registres pointeur et compteur et prennent en compte les interruptions après chaque boucle. Ceci permet de traiter convenablement les interruptions tout en assurant le bon dérou-

lement de l'instruction. Ces instructions peuvent aussi traiter des exceptions sans perdre les résultats des itérations précédentes. Si une exception survient à la 1902e itération d'une boucle de 2 000 itérations, l'exception est transmise avec les registres pointeur et compteur de la 1901e itération, dernière itération valide. Une fois qu'on a trouvé la cause de l'exception, le programme peut se poursuivre et effectuer les 99 itérations restantes.

Chaque interruption et chaque exception sont associées à un nombre de 8 bits qu'on appelle un numéro de vecteur. Les exceptions ont des numéros de vecteurs prédéterminés variant de 0 à 31. Par exemple, l'exception du type erreur de division a un numéro de vecteur égal à 0 et l'exception du type page un numéro de vecteur égal à 14. Vous pouvez assigner les interruptions à n'importe quel numéro compris entre 0 et 255, mais cela peut provoquer des conflits avec les numéros de vecteurs des exceptions. On assignera donc les interruptions à des numéros compris entre 32 et 255. Le numéro de vecteur sélectionne le handler d'une interruption ou d'une exception donnée à partir de la table des descripteurs d'interruptions.

En général, les interruptions et les exceptions peuvent survenir entre deux instructions quelconques. Cependant, on peut employer certains mécanismes (décrits dans ce chapitre) pour masquer les interruptions et certaines exceptions. Ceci signifie que le système les ignore jusqu'à ce qu'elles soient démasquées. On ne peut masquer que les exceptions qui servent à la mise au point des programmes (voir le Chapitre 8). Les autres exceptions, y compris les exceptions du type segment ou page (voir le Chapitre 5), ne peuvent être masquées et peuvent donc survenir à tout moment.

INTERRUPTIONS

Comme nous l'avons déjà vu, les interruptions sont causées par des événements externes asynchrones. L'électronique externe utilise des broches particulières du 80386 pour signaler les interruptions au processeur. Le 80386 a deux sortes d'interruptions, correspondant chacune à une broche du processeur : INTR et NMI.

Interruptions INTR

Les interruptions INTR sont des interruptions masquables et on peut leur attribuer n'importe quel numéro de vecteur. Lorsque l'électronique externe signale une interruption INTR, elle apporte aussi un numéro de vecteur d'interruption codé sur 8 bits. Le Contrôleur d'Interruptions Programmable 8259A d'Intel a été étudié pour travailler avec le 80386 et peut multiplexer jusqu'à 64 lignes d'interruptions sur l'unique broche INTR du 80386. Un seul 8259A peut traiter jusqu'à huit interruptions. On peut employer jusqu'à neuf 8259A pour traiter jusqu'à 64 lignes d'interruptions, chacune d'elles étant assignée à un numéro de vecteur différent. En plus de ces fonctions, le 8259A peut attribuer et arbitrer des priorités aux interruptions. Ceci est une fonction importante, puisque le 80386 ne reconnaît qu'une seule priorité pour les interruptions INTR.

Ces dernières peuvent être masquées par le bit IF du registre EFLAGS. Si IF = 1, des interruptions INTR peuvent survenir entre deux instructions. Si IF = 0, elles sont masquées. Si une interruption INTR est signalée lorsque IF = 0, elle est mise en attente soit jusqu'à ce qu'il soit égal à 1, soit jusqu'à ce que l'électronique externe retire sa requête INTR. On peut employer IF pour masquer les interruptions lors de séquences critiques qui doivent s'exécuter sans être interrompues.

Interruptions NMI

L'interruption non masquable (*Non Maskable Interrupt*) (NMI) n'est pas masquée par le bit IF, et est associée au numéro de vecteur 2. Ce numéro n'est donc pas fixé par un circuit extérieur. On emploie généralement un NMI pour signaler tout problème sérieux du système tel que *time-out* du bus, défaillance de l'alimentation, etc. Les interruptions NMI sont masquées uniquement lors de l'exécution du handler de ces NMI. Contrairement aux interruptions INTR qui sont masquables par le bit IF, un logiciel ne peut pas masquer des NMI. Dès réception d'un NMI, c'est le 80386 qui masque de façon interne les NMI suivants jusqu'à ce qu'une instruction IRET soit exécutée pour retourner au programme interrompu.

EXCEPTIONS

Les exceptions sont des conditions inhabituelles ou non valides associées à l'exécution d'une instruction particulière. Le 80386 reconnaît plusieurs classes différentes d'exceptions, et assigne à chacune d'elles un numéro de vecteur différent. Chaque exception est ainsi classée en faute (*fault*), en trappe (*trap*) ou en échec (*abort*), selon le moment et l'endroit où elle a été signalée.

Une *faute* est une exception qui a été signalée avant l'instruction qui a causé l'exception. Les valeurs de CS et de EIP, sauvegardées lors du transfert de contrôle au handler, pointeront sur l'instruction causant la faute. Une fois que la cause de l'exception a été déterminée, on peut rendre la main au programme interrompu par une instruction IRET et retourner ainsi à l'instruction qui a engendré la faute.

Le système détecte les fautes soit avant qu'il n'exécute l'instruction, soit au cours de son exécution. Si une faute est détectée au milieu d'une instruction, les opérandes de cette dernière sont sauvegardés avec les valeurs qu'ils avaient avant le début de l'exécution de l'instruction. Ainsi, lorsque le programme aura de nouveau la main, l'instruction en cause sera réexécutée avec exactement les mêmes opérandes pour garantir une exécution correcte.

Une *trappe* est une exception qui est signalée après l'instruction qui a causé l'interruption. Les valeurs de CS et de EIP, sauvegardées lors du transfert de contrôle au handler, pointent sur l'instruction située après celle qui a généré l'exception. Etant donné que CS et EIP pointent sur l'instruction suivante, on ne peut généralement pas identifier l'instruction qui a généré la trappe. Avant que la trappe ne soit signalée, l'instruction qui l'a engendrée s'achève normalement, ce qui peut entraîner des changements dans les registres ou la mémoire.

Un *échec* est une exception qui est signalée lors de conditions critiques telles que des défaillances du matériel ou des valeurs illégales dans les tables du système. On ne peut pas identifier l'instruction qui a engendré cet échec. De plus, on ne peut pas poursuivre l'exécution d'un programme qui

a été interrompu par un échec. Dès réception d'un échec, le handler peut avoir besoin de relancer le système d'exploitation après avoir restructuré les tables du système.

Les exceptions du type segment et page (voir le Chapitre 5) sont des exemples de fautes. Si une de ces exceptions est détectée lors de l'exécution d'une instruction, elle est signalée et CS et EIP pointent sur l'instruction en cause. Les exceptions du type pas à pas et point d'arrêt (Chapitre 8) sont des bons exemples de trappes. Si ces exceptions sont détectées lors de l'exécution d'une instruction, cette dernière s'achève normalement et signale la trappe. Dans ce cas, CS et EIP pointent sur l'instruction suivante.

Les instructions d'interruptions logicielles INT *n* et INTO (voir le Chapitre 3) sont plutôt des instructions de trappes programmées que des interruptions. Elles génèrent une trappe, ce qui effectue un transfert de contrôle vers le handler associé. Dans ce cas, CS et EIP pointent sur l'instruction suivante.

Relance d'une instruction

Après une faute, on peut relancer toutes les instructions. Le système d'exploitation n'a pas besoin d'intervenir dans ce cas. En effet, le processeur va signaler une faute alors que le contexte permet de relancer l'exécution de l'instruction dès correction de la faute. Les valeurs de CS et de EIP sauvées lors de la faute pointeront sur l'instruction en cause. Une fois que le système d'exploitation a corrigé la cause de la faute, l'instruction interrompue peut être réexécutée simplement par un IRET dans le handler.

Types d'exception

Dans ce paragraphe, nous allons donner la liste des types d'exceptions avec leur numéro de vecteur, et nous décrirons brièvement chaque type. Chaque exception est du type faute, trappe ou échec. Certaines exceptions apportent une information supplémentaire en transmettant un code d'erreur au handler correspondant.

La plupart des exceptions sont faciles à employer, ce qui permet de les décrire rapidement. On se reportera au Chapitre 5 (exceptions du type page et segment) et au Chapitre 8 (exceptions du type mise au point) pour plus de précisions. Nous parlerons des exceptions liées au coprocesseur dans un paragraphe particulier de ce chapitre.

Exception 0 - Erreur de division

Lors d'une instruction IDIV ou DIV, cette exception survient si le quotient est trop grand pour être représenté par l'opérande résultat ou si le diviseur est égal à 0. Au début du handler associé, les valeurs sauvegardées de CS et de EIP pointent sur l'instruction en cause. Aucun code d'erreur n'est transmis.

Exception 1 - Exceptions du type mise au point

Les exceptions du type mise au point (voir le Chapitre 8) utilisent le vecteur n° 1. Certaines sont des fautes, d'autres des trappes. Le handler associé peut faire référence au registre DR6 (voir le Chapitre 8) pour déterminer la ou les conditions qui ont causé cette exception et si ces exceptions sont des fautes ou des trappes. On peut détecter plus d'une exception en une instruction, puisque plusieurs bits de DR6 peuvent être positionnés. Les fautes du type mise au point (mais pas les trappes) sont masquées si le bit RF du registre EFLAGS est égal à 1. Aucun code d'erreur n'est transmis.

Exception 3 - Point d'arrêt INT 3

INT 3 est une forme particulière (un octet) de l'instruction INT *n*. Les logiciels de mise au point (*debuggers*) peuvent l'utiliser pour introduire des points d'arrêt dans un programme (voir le Chapitre 8). C'est une trappe programmée, et non une interruption. Au début du handler d'exception, les valeurs de CS et de EIP qui ont été sauvegardées pointent sur l'instruction suivant directement INT 3, donc un octet après cette instruction. Aucun code d'erreur n'est transmis.

Exception 4 - Débordement

Une trappe du type débordement est associée au vecteur 4. L'instruction INTO associe une trappe conditionnelle au vecteur 4. Elle générera une trappe si le bit OF du registre EFLAGS est égal à 1. Si OF est égal à 0, rien ne se passe et l'exécution se poursuit avec l'instruction suivant INTO. Au début du handler de l'exception du type débordement, les valeurs de CS et de EIP qui ont été sauvegardées pointent sur l'instruction suivant INTO. Aucun code d'erreur n'est transmis.

Exception 5 - Test de limite

L'instruction BOUND permet de tester si une valeur signée est inférieure à une limite inférieure, ou supérieure à une limite supérieure. Dans ce cas, il y a génération d'une exception. Au début du handler associé, les valeurs de CS et de EIP qui ont été sauvegardées pointent sur l'instruction BOUND en cause. Aucun code d'erreur n'est transmis.

Exception 6 - CodeOp non valide

On obtient une faute du type CodeOp non valide si CS et EIP pointent sur un bloc binaire qui n'est pas reconnu comme une instruction par le 80386. Ceci peut arriver si le champ CodeOp spécifie un code qui n'est pas une instruction du 80386, ou si un opérande registre est spécifié pour une instruction qui requiert un opérande mémoire, ou si le préfixe LOCK est utilisé pour une instruction qui ne le permet pas. Au début du handler associé, les valeurs de CS et de EIP qui ont été sauvegardées pointent sur le premier octet de l'instruction non valide. Aucun code d'erreur n'est transmis.

Exception 7 - Processeur d'extension non disponible (DNA)

Le processeur d'extension considéré est le coprocesseur arithmétique 80387. Vous pouvez utiliser cette exception (du type faute) pour lui substituer un logiciel de simulation arithmétique dans des systèmes qui ne comportent pas de coprocesseur 80387. Vous pouvez aussi employer cette exception pour retarder le changement de contexte du 80387 (simulé ou non) après une commutation de tâches jusqu'à ce qu'une autre tâche fasse appel à une instruction à virgule flottante. Les valeurs de CS et de EIP qui ont été sauvegardées pointent sur l'instruction en cause. Aucun code d'erreur n'est transmis.

Une exception 7 est générée par les conditions suivantes :

1 Une instruction à virgule flottante est exécutée et soit le bit EM, soit le bit TS de CR0 est égal à 1.

2 Une instruction WAIT est exécutée et les deux bits TS et MP de CR0 sont égaux à 1.

Reportez-vous au Chapitre 5 pour tout renseignement complémentaire sur les bits EM, TS et MP de CR0.

Exception 8 - double faute

Si le processeur détecte une exception du type page ou segment lors d'une autre exception, il tentera de signaler une double faute plutôt que la seconde exception. Les doubles fautes sont considérées comme des échecs. Ainsi, les valeurs de CS et de EIP qui ont été sauvegardées peuvent ne pas pointer sur l'instruction en cause, et il n'est donc pas possible de relancer cette instruction par la suite. Un code d'erreur 0 est transmis.

En général, les doubles fautes indiquent qu'il y a de sérieux problèmes au niveau des tables du système : les tables des descripteurs de segments, les tables des pages ou la table des descripteurs d'interruptions. Cette exception est bien utile lorsqu'on met au point un système d'exploitation qui ne gère pas encore convenablement les tables du système. Dans un système de production, le handler associé devra probablement relancer le système d'exploitation après avoir reconstruit les tables du système.

Il est possible d'obtenir une faute du type page pendant que le système tente de générer une faute du type segment. Dans ce cas, c'est une faute du type page qui est générée, et non une double faute. Cependant, une faute du type segment détectée lors de la transmission d'une faute du type page ou segment donnera une double faute, et une faute du type page détectée lors d'une faute de type page donnera aussi une double faute.

Si une exception du type segment ou page est détectée lors du traitement de la double faute, le processeur arrêtera l'exécution de l'instruction courante et passera en mode "shutdown" (arrêt catastrophique). Le mode shutdown laisse le processeur dans le même état que l'exécution d'une instruction HLT : il s'arrête complètement. Il restera dans cet état jusqu'à ce qu'il reçoive un NMI ou qu'il soit réinitialisé. Pendant que le processeur est en mode shutdown, les interruptions INTR sont masquées.

Exception 9 - Dépassement segment par le processeur d'extension

Cette exception qui est du type échec est générée lorsque le coprocesseur dépasse les limites d'un segment alors qu'il est en train de lire ou d'écrire une partie d'un opérande. Aucun code d'erreur n'est transmis.

Par exemple, si une instruction à virgule flottante a placé un opérande de 8 octets au déplacement 0FFFFFFFCh dans un segment de taille 0FFFFFFFDh, cette exception sera générée.

Puisque cette exception est du type échec, l'instruction en cause ne peut pas être relancée. Le coprocesseur 80387 doit être réinitialisé avec l'instruction FINIT avant que le handler associé ne rende la main au programme interrompu. Les valeurs de CS et de EIP qui ont été sauvegardées pointeront sur l'instruction en cause, ce qui facilite le diagnostic du problème. Cette exception affecte uniquement le programme qui s'exécutait lorsque la condition a été détectée.

Exception 10 - TSS Non valide

Le chargement d'un sélecteur à partir du TSS peut générer une exception du type segment. Si cette exception n'est pas du type non présent, on a génération de l'exception 10. Cette dernière apporte un code d'erreur contenant le sélecteur du segment qui a causé l'exception. Dans le Chapitre 5, nous décrivons le format de ce code d'erreur et les conditions qui ont engendré cette exception (exception référencée par $TS).

Puisque cette exception est une faute, les valeurs de CS et de EIP qui ont été sauvegardées pointent sur l'instruction en cause, ou sur la première instruction d'une tâche si l'exception a été générée lors d'une commutation de tâches.

Exception 11 - Segment non présent

Cette exception survient lorsque le processeur trouve un bit P nul lorsqu'il accède à un descripteur valide qui ne doit pas être chargé dans le registre SS. Si dans le descripteur de SS le bit P est égal à 0, on aura une exception du type segment pile plutôt qu'une exception du type segment non présent. Cette dernière transmet un code d'erreur qui contient le sélecteur du segment en cause. Dans le Chapitre 5, nous décrivons le format de ce code d'erreur et les conditions qui ont engendré cette exception (exception référencée par $NP).

Puisque cette exception est une faute, les valeurs de CS et de EIP sauvegardées pointent sur l'instruction en cause ou sur la première instruction d'une tâche si l'exception a été générée lors d'une commutation de tâches.

Exception 12 - Segment pile

Cette exception survient lorsque le processeur détecte certains problèmes dus au segment adressé par le registre segment SS. Elle transmet un code d'erreur dont la valeur dépend de la condition détectée.

Dans le Chapitre 5, nous décrivons le format de ce code d'erreur et les conditions qui ont engendré cette exception (exception référencée par $SS). On peut classer en trois catégories les conditions qui peuvent causer cette exception :

1 S'il y a violation des limites du segment adressé par le registre SS (code d'erreur nul).

2 S'il y a violation des limites de la pile interne lors d'un appel interniveau ou d'une interruption (code d'erreur contenant le sélecteur de la pile interne).

3 Si le descripteur qui doit être chargé dans le registre SS a son bit P égal à 0 (code d'erreur contenant le sélecteur du segment non présent).

Les deux dernières conditions engendrent des exceptions dont les codes d'erreur ne sont pas nuls. Le handler associé peut examiner le descripteur du segment identifié par le sélecteur contenu dans le code d'erreur, afin de différencier les deux conditions. Si le bit P de ce descripteur est égal à 1, on a affaire à une violation des limites de pile interne. Sinon, c'est une exception du type segment pile non présent.

Cette exception est une faute. Ainsi, les valeurs de CS et de EIP qui ont été sauvegardées pointent sur l'instruction en cause ou sur la première instruction de la nouvelle tâche si l'exception a été détectée lors d'une commutation de tâches. L'instruction est réexécutable.

On peut utiliser des fautes avec des codes d'erreur nuls lorsqu'on veut faire croître le segment pile de systèmes qui le permettent.

Exception 13 - Protection générale

Toute exception du type segment qui n'appartient pas aux catégories précédentes est une exception du type protection générale. Elle transmet un code d'erreur dont les valeurs dépendent de la condition détectée. Dans le Chapitre 5, nous donnons le format de ce code d'erreur et les conditions qui ont engendré cette exception (exception référencée par $GP).

Puisque cette exception est une faute, les valeurs de CS et de EIP qui ont été sauvegardées pointent sur l'instruction en cause ou sur la première instruction d'une tâche si l'exception a été générée lors d'une commutation de tâches.

Les conditions qui génèrent cette exception peuvent être rangées en deux classes selon les réponses possibles du handler associé :

1 L'exception peut indiquer une violation du modèle de protection par un programme d'application exécutant une instruction privilégiée ou une référence E/S. Les systèmes qui acceptent les programmes 8086 (voir le Chapitre 9) ou les références mémoire en mode virtuel doivent émuler ces instructions et relancer le programme interrompu après l'instruction en cause. Ces exceptions transmettent un code d'erreur nul.

2 L'exception peut indiquer une violation du modèle de protection qui met fin au programme en cause. On peut alors avoir un code d'erreur nul ou comportant un sélecteur.

On peut distinguer ces deux cas en examinant le code d'erreur. Puisque cette exception est une faute, les valeurs de CS et de EIP qui ont été sauvegardées pointent sur l'instruction en cause. Si le code d'erreur est nul et si l'instruction peut être émulée, le handler le fera et rendra la main à l'instruction suivante. Si ce n'est pas possible, on doit pouvoir mettre fin au programme en cause.

Exception 14 - Exceptions du type page

Cette exception survient lorsque la pagination est activée (PG = 1 dans le registre CR0) et qu'une instruction fait référence à une adresse linéaire qui ne se trouve pas dans une page présente ni dans une page dont les attributs ne permettent pas ce type d'accès. Le processeur charge le registre CR2 avec l'adresse linéaire en cause, et transmet un code d'erreur qui fournit le type d'accès mémoire. Dans le Chapitre 5, nous décrivons le format de ce code d'erreur et les conditions qui ont engendré cette exception.

Puisque cette exception est une faute, les valeurs de CS et de EIP qui ont été sauvegardées pointent sur l'instruction en cause. Plus important encore, on peut exécuter à nouveau l'instruction, si bien qu'une fois la cause de cette exception déterminée, on peut relancer cette instruction par un simple IRET dans le handler pour rendre la main au programme appelant.

Exception 16 - Erreur du coprocesseur

Cette exception survient s'il y a eu une erreur numérique non masquée, par exemple un débordement. Elle est générée lors de l'instruction à virgule flottante suivante ou du WAIT suivant. Aucun code d'erreur n'est transmis. Nous expliquerons ce type d'exception plus tard.

Résumé des exceptions

Le Tableau 6.1 résume les conditions d'exception dont nous avons déjà parlé. Elles sont classées selon leur numéro de vecteur. Le tableau donne une liste des exceptions et des instructions qui les génèrent, classe les exceptions en trois catégories, et indique s'il y a ou non code d'erreur.

PRIORITE DES INTERRUPTIONS ET DES EXCEPTIONS

Dans ce paragraphe, nous allons expliquer dans quel ordre sont traitées plusieurs interruptions générées par la même instruction. Le Tableau 6.2 donne une liste des interruptions et des exceptions reconnues par le 80386 et classées selon leur priorité. Si plusieurs exceptions sont détectées, c'est celle qui a la priorité la plus haute qui est transmise. Les exceptions de priorité plus basse sont ignorées et les interruptions de priorité plus basse sont mises en attente. Une exception qui a été ignorée sera probablement détectée à nouveau lorsque la cause de l'exception de priorité la plus haute aura été déterminée et que l'instruction sera relancée.

NUMERO VECTEUR	TYPE d'EXCEPTION	CLASSE	CODE ERREUR	INSTRUCTION
0	Erreur de division	Faute	NON	DIV, IDIV
1	Mise au point	F/T	NON	Toute instruction
3	INT 3	Trappe	NON	INT 3
4	Débordement	Trappe	NON	INTO
5	Test de limites	Faute	NON	BOUND
6	CodeOp non valide	Faute	NON	Instruction/opérande non valide
7	DNA	Faute	NON	Instruction à virgule flottante ou WAIT
8	Double faute	Echec	OUI	Toute instruction
9	Dépassement segment	Echec	NON	Toute instruction à virgule flottante
10	TSS non valide	Faute	OUI	JMP, CALL, IRET et interruption
11	Segment non présent	Faute	OUI	Toute instruction qui charge un registre segment
12	Segment pile	Faute	OUI	Toute instruction qui charge SS ou qui fait référence à un segment adressé par SS
13	Protection générale	Faute	OUI	Toute instruction privilégiée ou qui fait référence à la mémoire
14	Type page	Faute	OUI	Toute instruction qui fait référence à la mémoire
16	Erreur coprocesseur	Faute	NON	Instruction à virgule flottante ou WAIT
0-255	Inter. logicielle	Trappe	NON	Int n

Tableau 6.1 : *Résumé des exceptions.*

Par exemple, si NMI et INTR sont signalées en même temps, c'est NMI qui est prise en compte puisqu'elle est plus prioritaire. INTR est alors mise en attente. Considérons maintenant un exemple plus complexe : une trappe de mise au point et une faute de type page sont détectées dans la même instruction. Dans ce cas, la faute du type page est signalée et la trappe du type mise au point est ignorée. Une fois la cause de la faute déterminée, l'instruction sera relancée et la trappe du type mise au point surviendra à nouveau. Elle sera transmise puisqu'il n'y aura pas d'exception de priorité plus haute.

Notez que ce cas (exceptions et interruptions simultanées) est tout à fait différent de celui de l'exception du type double faute. Le système détermine quelle exception doit être transmise si plusieurs exceptions sont générées au cours de l'exécution d'une instruction. Une seule de ces exceptions ou interruptions sera transmise. Si une exception du type segment ou page survient lorsqu'il y a génération de l'exception ou de l'interruption de plus haute priorité, il y aura génération d'une double faute. Les doubles fautes ne sont pas seulement générées quand il y a plusieurs exceptions simultanées. En effet, on a génération d'une double faute si une exception du type segment ou page survient lors du transfert de contrôle au handler associé.

TYPE D'INTERRUPTION/EXCEPTION	PRIORITE
Fautes de mise au point	La plus haute
Autres fautes	
INT *n* , INTO	
Trappes de mise au point	
Interruption NMI	
Interruption INTR	La plus basse

Tableau 6.2 : *Priorité des exceptions et des interruptions.*

MASQUAGE DES INTERRUPTIONS ET DES EXCEPTIONS

Certaines conditions et états des indicateurs du processeur permettent de masquer les interruptions et les exceptions du type mise au point. Dans ce cas, le processeur met en attente les interruptions et ignore les exceptions du type mise au point.

1 Les interruptions INTR sont masquées si le bit IF du registre EFLAGS est égal à 0.

2 Une instruction STI exécutée avec IF = 0 masquera les interruptions INTR jusqu'à ce que la prochaine instruction soit exécutée.

3 Les fautes de mise au point (mais pas les trappes) sont masquées si le bit RF du registre EFLAGS est égal à 1.

4 Si un NMI survient après un autre NMI, il est masqué jusqu'à ce qu'une instruction IRET soit exécutée.

5 Une instruction MOV ou POP sur le registre SS masquera les interruptions et les exceptions du type mise au point jusqu'à la fin de la prochaine instruction.

Nous avons déjà fait mention des trois premiers cas dans ce chapitre. La condition 5 permet d'effectuer en deux instructions un chargement indivisible du nouveau pointeur de pile dans les registres SS et ESP. Si un MOV ou un POP dans SS est suivi par un MOV ou un POP dans ESP, aucune interruption et aucune exception de mise au point ne seront transmises entre les deux instructions. Ceci permet au système de mettre à jour convenablement le pointeur de pile. Tant qu'il n'y aura pas d'exceptions du type page ou segment lors du chargement de ESP dans la seconde instruction, les deux instructions seront exécutées sans interruption. L'instruction LSS (voir le Chapitre 3) est une nouveauté du 80386 qui permet de charger à la fois SS et un registre général (habituellement ESP) à partir d'un pointeur 48 bits en mémoire. On doit si possible utiliser l'instruction LSS pour charger un nouveau pointeur de pile plutôt que

l'ancienne méthode, puisqu'il effectue cette opération en une seule instruction.

METHODES DE TRANSFERT DES INTERRUPTIONS/EXCEPTIONS

Dans ce chapitre, nous présentons deux façons d'appréhender les transferts de contrôle des interruptions et des exceptions. Ces transferts consistent à exécuter le handler associé au numéro de vecteur de l'interruption ou de l'exception. La première approche, donnée dans ce paragraphe, est une vue générale des méthodes employées lors de ces transferts. La seconde, donnée au paragraphe suivant, est une description détaillée du mécanisme en notation pseudo-C. On peut considérer que ces méthodes et ces mécanismes sont identiques à ceux utilisés par les CALL intersegments via des guichets de tâche ou d'appel (voir le Chapitre 5). Vous devez maîtriser les notions de ce chapitre avant de lire les lignes suivantes.

Les interruptions et les exceptions utilisent la table des descripteurs d'interruptions (IDT), qui est pointée par le registre IDTR (voir le Chapitre 5). On utilise le numéro de vecteur pour indexer l'IDT et obtenir ainsi un descripteur de guichet sur 8 octets. La Figure 6.1 donne le format d'un descripteur de guichet. Les guichets sont constitués d'un pointeur 48 bits plus 16 bits d'attributs. Le pointeur 48 bits contient la même information que le type de données pointeur (voir le Chapitre 2), mais il est stocké avec le déplacement divisé en deux parties. La partie sélecteur du pointeur est rangée dans les octets situés aux déplacements 2 et 3. Le déplacement 32 bits est stocké à deux endroits différents. Ses 16 bits de poids faible sont rangés au déplacement 0 et ses 16 bits de poids fort au déplacement 6.

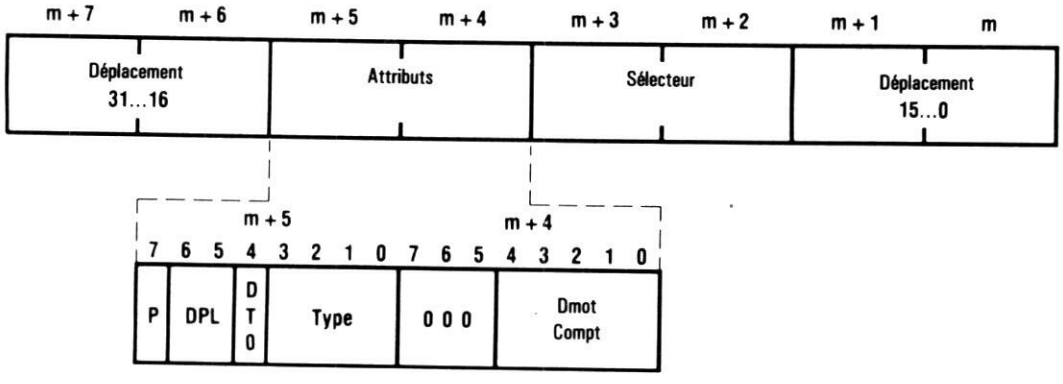


Figure 6.1 : *Format d'un descripteur de guichet.*

Dans un descripteur de guichet, les attributs sont placés dans les octets situés aux déplacements 4 et 5 :

P — P est le bit *présent*. P = 1 indique que le guichet est valide. P = 0 indique qu'il n'est pas valide et l'emploi de ce guichet générera une exception.

DPL — DPL est le *niveau de privilège du descripteur*, qui définit le niveau de privilège associé au guichet. Le DPL d'un guichet est testé uniquement par les instructions INT *n* et INTO pour éviter qu'un programme d'applications n'exécute un INT n avec un numéro de vecteur quelconque, par exemple celui assigné aux interruptions disque. Il est ignoré par toutes les autres exceptions et interruptions.

DT — DT est le bit DType, qui distingue les segments mémoire (DType = 1) des segments système et des guichets (DType = 0). Il est à 0 pour indiquer qu'on se trouve en présence d'un guichet.

Type — Le champ de 4 bits Type définit le type du guichet (voir le Tableau 6.3).

Les types des descripteurs dans la IDT sont les suivants :

- Guichet de tâche
- Guichet d'interruption 286
- Guichet de trappe 286
- Guichet d'interruption 386
- Guichet de trappe 386

DmotCompt — Le champ DmotCompt n'est pas utilisé dans les guichets de tâche, d'interruption ou de trappe.

Dans le Chapitre 9, nous parlerons des guichets d'interruption 286 et des guichets de trappe. Nous décrirons les transferts à travers des guichets de tâche, des guichets d'interruption 386 et des guichets de trappe. Les transferts à travers des guichets de tâche appellent un handler qui se trouve dans une tâche différente. Les transferts à travers des guichets d'interruption et de trappe appellent un handler de la tâche courante. Comme nous le verrons, les noms de ces guichets ne correspondent pas exactement à leur emploi, ce qui est normal puisqu'il n'existe pas de guichet du type faute ou échec ! Les interruptions, les fautes, les trappes et les échecs peuvent être traités par un des trois types de guichets suivants : tâche, interruption ou trappe.

Les transferts à travers des guichets de tâche effectuent une commutation complète de tâches pour exécuter le handler d'une tâche différente.

TYPE	DEFINITION
0	Indéfini
1	TSS 286 Disponible
2	LDT
3	TSS 286 Occupé
4	Guichet d'appel 286
5	Guichet de tâche
6	Guichet d'interruption 286
7	Guichet de trappe 286
8	Indéfini
9	TSS 386 Disponible
A	Indéfini
B	TSS 386 Occupé
C	Guichet d'appel 386
D	Indéfini
E	Guichet d'interruption 386
F	Guichet de trappe 386

Tableau 6.3 : *Types de descripteurs guichets et segments système.*

Cette commutation est identique à celle d'un CALL à travers un guichet de tâche. Les guichets d'interruption et de trappe effectuent un transfert beaucoup plus simple vers un handler de la tâche courante. Ce transfert est similaire, mais non identique, à celui employé lors d'un CALL à travers un guichet d'appel.

Les guichets d'interruption et de trappe

Si le descripteur de l'IDT, indexé par le numéro de vecteur, est un guichet du type guichet d'interruption 386 ou guichet de trappe 386, le transfert s'effectuera vers un handler dans la tâche courante. Comme pour un

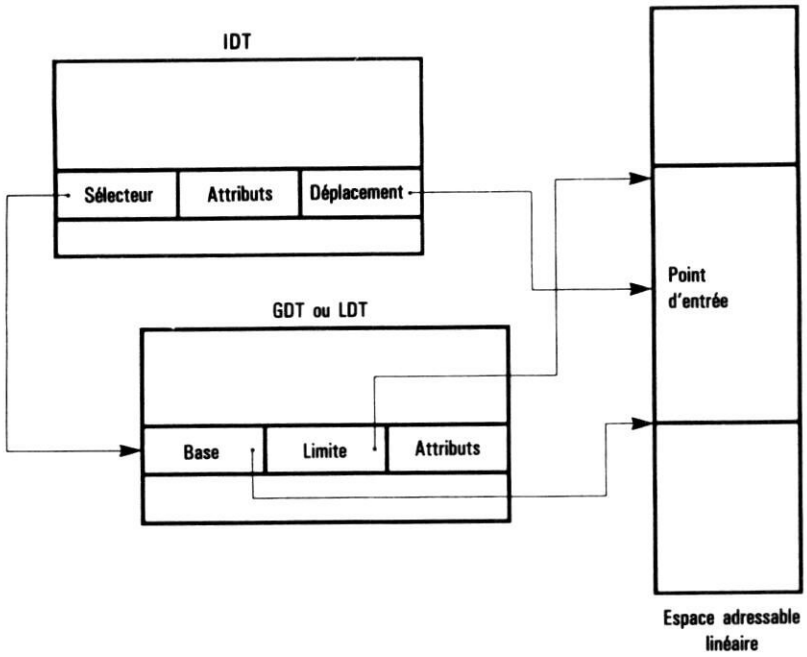


Figure 6.2 : *Transfert à travers un guichet d'interruption ou de trappe.*

CALL à travers un guichet d'appel, le pointeur 48 bits sur le handler est obtenu à partir du guichet d'interruption ou de trappe. La Figure 6.2 montre comment un guichet de trappe ou d'interruption dans l'IDT détermine le handler d'interruption. Le sélecteur obtenu à partir du guichet est employé pour accéder au descripteur dans la GDT ou la LDT. Ce descripteur doit être associé à un segment mémoire exécutable. Le déplacement du descripteur de guichet spécifie le déplacement du point d'entrée du handler dans le segment.

Seules les instructions INT *n* et INTO testent le DPL du guichet pour éviter qu'un programme d'application n'exécute un INT *n* avec un numéro

de vecteur quelconque, par exemple celui associé aux interruptions disque. Toutes les autres exceptions et interruptions ignorent le DPL du guichet.

Le sélecteur obtenu à partir du guichet doit spécifier un segment mémoire exécutable et présent. Les champs Type et DPL du descripteur de ce segment déterminent si l'interruption ou l'exception se font sur une procédure au niveau de privilège courant ou s'il doit y avoir transfert vers un nouveau niveau de privilège. Les interruptions et les exceptions peuvent produire un transfert vers une procédure au même niveau de privilège ou à un niveau interne. L'instruction IRET peut effectuer le transfert au même niveau de privilège ou à un niveau de privilège externe. On retrouve les règles des instructions CALL et RET.

Si le sélecteur obtenu à partir du guichet identifie soit un segment conforme dont le DPL est identique ou d'un niveau interne relativement à CPL, soit un segment non conforme avec DPL = CPL, alors il y aura transfert à une procédure du niveau de privilège courant. Un segment non conforme dont le DPL est d'un niveau interne relativement à CPL effectue un transfert au niveau interne DPL de ce segment. Comme pour un CALL à travers un guichet d'appel, le segment pile est échangé avec celui du niveau interne. Les valeurs initiales de SS et de ESP pour la nouvelle pile sont obtenues à partir des champs du TSS courant (voir le Chapitre 5). Les anciennes valeurs de SS et de ESP sont empilées dans la nouvelle pile; un IRET peut donc récupérer le pointeur de pile externe. Au contraire du CALL à travers un guichet d'appel, aucun paramètre n'est copié de l'ancienne pile vers la nouvelle pile. Par conséquent, le système ignore le champ DmotCompt des guichets d'interruption et de trappe.

Une fois que le niveau de privilège et la pile ont été commutés, si cela était nécessaire, le reste du traitement de l'interruption ou de l'exception se poursuit. Le registre EFLAGS est empilé. Une instruction IRET peut donc le récupérer. Puis les bits NT et TF du registre EFLAGS sont mis à 0. TF = 0 indique que le handler n'est pas en mode pas à pas (voir le Chapitre 8). NT = 0 indique qu'un IRET rendra la main à la même tâche et non à une tâche imbriquée. Si le transfert s'effectue à travers un guichet d'interruption, le bit IF de EFLAGS aussi est mis à 0 pour que les interruptions INTR soient masquées dans le handler. Si le transfert s'effectue à travers un guichet de trappe, le bit IF n'est pas modifié.

Puis le pointeur de retour (valeurs courantes de CS et de EIP) est empilé. CS est ensuite chargé avec le sélecteur obtenu à partir du guichet, après avoir positionné son champ RPL à CPL. EIP est chargé avec le déplacement obtenu à partir du guichet pour achever le transfert au handler.

La dernière partie du traitement d'une exception consiste à empiler un code d'erreur si nécessaire. Certaines exceptions fournissent un code d'erreur (voir le Tableau 6.1). C'est une valeur 16 bits dont le format est décrit au Chapitre 5. Pour conserver l'alignement de la pile, le système empile le code d'erreur comme une valeur 32 bits dont les 16 bits de poids fort sont indéfinis.

Les guichets d'interruption et de trappe fonctionnent de manière identique, sauf pour le positionnement du bit IF du registre EFLAGS après le transfert à travers le guichet. Les guichets d'interruption positionnent IF à 0 pour masquer les interruptions INTR au début du handler, mais les guichets de trappe ne le modifient pas. Comme leur nom l'indique, les guichets d'interruption se prêtent mieux au traitement des interruptions INTR, puisqu'ils masquent les interruptions avant d'entrer dans le handler associé. Les guichets de trappe sont étudiés pour traiter les exceptions (trappes logicielles comprises), puisqu'ils ne modifient pas l'état du bit IF.

La Figure 6.3 montre une commutation de pile lors d'un transfert à un niveau interne à travers un guichet d'interruption ou de trappe dans le cas d'une exception qui fournit un code d'erreur. La pile de niveau externe adressée par les registres SS et ESP est à droite sur la figure. La nouvelle pile de niveau interne est à gauche. Le premier élément empilé (à l'adresse la plus haute) est un pointeur sur la pile de niveau externe donné par les anciennes valeurs de SS et de ESP. Puis l'ancienne valeur de EFLAGS est empilée. Ensuite, c'est la valeur de l'adresse de retour qui est empilée avec les valeurs externes de CS et de EIP. Le code d'erreur est empilé en dernier. Au début du handler, SS adresse le segment de pile interne et ESP donne le déplacement du code d'erreur qui se trouve au sommet de la pile.

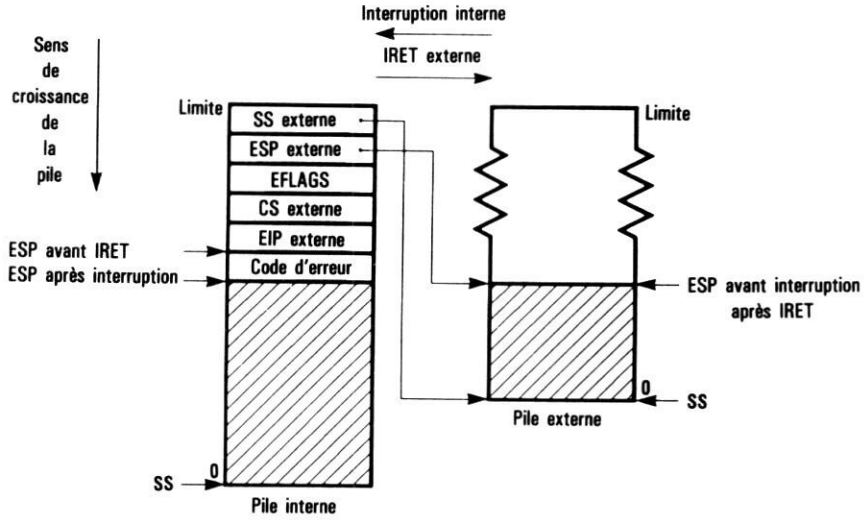


Figure 6.3 : *Guichet de trappe ou d'interruption vers un niveau interne.*

IRET avec NT = 0

On emploie l'instruction IRET pour revenir d'un handler d'interruption ou d'exception. Un handler associé à un guichet d'interruption ou de trappe exécutera IRET avec NT = 0 dans le registre EFLAGS, ce qui signifie que IRET peut trouver l'information de retour au sommet de la pile courante. IRET avec NT = 0 peut retourner à une procédure de même niveau de privilège ou de privilège externe.

IRET dépile le pointeur de retour (EIP et CS) du sommet de la pile puis la valeur de EFLAGS. Le champ RPL du sélecteur du CS dépilé identifie

le niveau de privilège de la tâche vers laquelle on retourne. Comme avec l'instruction RET (voir le Chapitre 5), le système emploie le RPL du sélecteur de retour plutôt que le DPL du segment identifié par ce sélecteur. Ainsi, on peut retourner à un segment conforme qui peut s'exécuter à un niveau autre que celui donné par son champ DPL.

Si le RPL du sélecteur de retour est égal au CPL, il n'y aura pas de changement de niveau de privilège. Si le RPL détermine un niveau externe, il doit y avoir changement de niveau de privilège. Dans ce cas, les valeurs de ESP et de SS, qui sont nécessaires pour récupérer la pile externe, sont dépilées de la pile interne.

IRET s'attend à trouver le pointeur de retour et la valeur de EFLAGS au sommet de la pile, si bien que les handlers associés à des exceptions qui fournissent des codes d'erreur doivent retirer le code d'erreur de la pile avant d'exécuter un IRET.

La Figure 6.3 nous a servi à illustrer un transfert vers un niveau interne à travers un guichet de trappe ou d'interruption. Elle va aussi nous présenter la pile avant et après l'exécution d'une instruction IRET. La pile de niveau externe est à droite sur la figure, et la pile courante de niveau interne à gauche. Avant d'exécuter l'instruction IRET, le handler doit dépiler le code d'erreur, si bien que ESP pointe sur la partie EIP du pointeur de retour empilé lors du transfert à travers le guichet d'interruption ou de trappe correspondant. Le pointeur de retour (EIP et CS) est dépilé pour rendre la main au programme interrompu, et le registre EFLAGS est récupéré par dépilage. Le système examine le champ RPL du sélecteur de CS dépilé pour déterminer s'il a affaire à un retour à un niveau externe. Dans cet exemple, RPL spécifie un niveau externe relativement au CPL, si bien que la pile de niveau externe est récupérée en dépilant les valeurs de ESP et de SS dans les registres correspondants. Le programme peut se poursuivre avec les valeurs de SS, ESP, EFLAGS, CS et EIP qui avaient été sauvegardées lors de l'interruption ou de l'exception correspondante. La figure montre que SS et ESP adressent la pile externe après la fin de l'instruction IRET.

Transferts à travers des guichets de tâche

Si le descripteur de l'IDT, indexé par le numéro de vecteur, est un guichet du type guichet de tâche, on a affaire à un transfert vers un handler d'une tâche différente. Comme pour un CALL via un guichet de tâche, le guichet contient un sélecteur 16 bits du segment TSS décrivant la tâche du handler. Ce sélecteur doit identifier un segment du type TSS 286 Disponible ou TSS 386 Disponible. Dans le Chapitre 9, nous parlerons du transfert à travers un TSS 286. Un transfert à travers un guichet de tâche vers un TSS 386 Disponible est identique à un CALL à travers un guichet de tâche vers un TSS 386 Disponible (voir le Chapitre 5). La seule différence est que les exceptions qui apportent un code d'erreur l'empileront dans la pile de la nouvelle tâche après la commutation. Pour conserver l'alignement de la pile, le code d'erreur 16 bits est empilé comme une valeur 32 bits dont les 16 bits de poids fort sont indéfinis.

Le transfert à travers un guichet d'appel entrera dans le handler avec NT = 1 dans le registre EFLAGS pour indiquer que la prochaine instruction IRET doit retourner à une tâche imbriquée. IRET fera référence au champ de liens du TSS courant pour obtenir le sélecteur du TSS de la tâche vers laquelle il retourne. Reportez-vous au Chapitre 5 pour plus de détails sur les commutations de tâches et sur les tâches imbriquées définies par le champ de liens du TSS.

Guichets de tâche ou guichets d'interruption/trappe ?

Le 80386 permet de traiter une interruption ou une exception par une procédure dans la tâche courante en employant un guichet de trappe ou d'interruption, ou par une tâche distincte en utilisant un guichet de tâche. Les handlers dans la tâche courante sont plus simples et donnent des transferts plus rapides, mais ils doivent sauvegarder et récupérer les registres du processeur. Les tâches handler prennent plus de temps mais sauvegardent et récupèrent les registres du processeur grâce au mécanisme de commutation de tâches. L'emploi d'une procédure handler permet d'accéder directement à l'état de la tâche qui s'exécutait au moment de l'interruption ou

de l'exception, mais exige que chaque tâche contienne une procédure handler. L'utilisation d'une tâche distincte permet d'isoler le handler, mais peut rendre l'accès à l'état de la tâche interrompue plus difficile.

Les exceptions du type TSS non valide doivent être prises en compte par un guichet de tâche pour assurer au handler un contexte de tâche valide. D'autres exceptions seront traitées plus facilement dans le contexte de la tâche où elles ont été détectées et n'ont pas besoin de masquer les interruptions, elles doivent donc employer des guichets de trappe. Le handler d'exception sur lequel pointe le guichet de trappe est une procédure partagée entre toutes les tâches et doit donc se trouver dans l'espace adressable global. Si chaque tâche a besoin d'un handler différent, le handler global d'exception peut conserver une table de ces handlers et appeler celui qui convient à la tâche ayant causé l'exception.

Habituellement, les interruptions n'ont pas de lien avec la tâche qu'elles ont interrompue, et peuvent ainsi bénéficier de l'isolement qu'apporte un guichet de tâche. Les interruptions qui demandent une réponse très rapide seront par contre traitées par un guichet d'interruption. Les systèmes qui n'utilisent pas le mécanisme de commutation de tâches, propre au 80386, devront aussi passer par les guichets d'interruption. Puisqu'une interruption peut survenir à n'importe quel moment, les handlers associés doivent se trouver dans l'espace adressable local, pour être adressable par toutes les tâches. Dans le Chapitre 7, nous donnerons un long exemple qui illustrera l'emploi des guichets d'interruption et la façon de placer le handler dans l'espace adressable global.

L'emploi d'un guichet de tâche permet de dispatcher automatiquement le handler lors de l'interruption ou de l'exception. C'est le 80386 qui effectue directement ce dispatching et contourne le système d'exploitation. Ceci permet de commuter les tâches plus rapidement, mais le handler doit appeler le dispatcher du système d'exploitation pour notifier cette commutation de tâches.

INTERRUPTIONS ET EXCEPTIONS

Dans les paragraphes suivants, nous allons présenter différemment les méthodes de transfert de contrôle des interruptions et des exceptions (notation pseudo-C). Nous ferons référence aux Sous-programmes, aux variables et aux types du Chapitre 5.

Description des interruptions

La routine Interrupt() (voir le Listing 6.1) décrit le traitement d'une interruption ou d'une exception. Elle reçoit deux paramètres :

1 Le numéro du vecteur associé à l'interruption ou à l'exception.

2 Un indicateur qui indique si la routine doit tester le DPL du guichet dans l'IDT.

```
/* Génère une interruption ou une exception.*/
Interrupt(VecNumber, CheckDPL)

int    VecNumber,    /* Numéro du Vecteur */
       CheckDPL;     /* 0 = ignore DPL du guichet de IDT */
                     /* 1 = teste DPL du guichet de IDT  */
                     /* relativement à CPL */
{   SegAttributes       Attributes;
    SelectorType        Selector;
   int   Dword1, Dword2, Offset, Base, Limit;

/* Lecture de l'élément de l'IDT pour le numéro de */
/* vecteur indiqué */
```

Listing 6.1 : *Description du traitement d'une interruption.*

```
if (VecNumber*8+7 > IDTR.Limit)
   SegmentException($GP, VecNumber*8+2);
AccessLinear(IDTR.Base+VecNumber*8, 4, 0 /* PL 0 */, 0
             /* Lecture */,&Dword1);
AccessLinear(IDTR.Base+VecNumber*8+4, 4, 0  /* PL 0 */, 0
             /* Lecture */,&Dword2);

Attributes = Dword2;  /* Les attributs du guichet dans le */
                      /* dmot de poids fort */
/* Sélecteur et déplacement de guichet */
Selector = Dword1 >> 16;
Offset   = (Dword1 & OFFFFh) | (Dword2 * OFFFF0000h);

/* Test des champs DType et Type du descripteur.*/
if (
      (Attributes.DType == 0)   /* Segment Mémoire */
       || /* Test DPL du guichet si CheckPL = 1 */
      ( (CheckDPL == 1) && (Attributes.DPL < CPL) )
       || (Attributes.Type < 5)
          /* Le type doit être 5, 6, 7, 14 ou 15 */
       || ( (Attributes.Type > 7) && (Attributes.Type < 14) )
      )
   SegmentException($GP, VecNumber*8+2);

/* Violations du type protection générale testées avant tout*/
if (Attributes.P == 0  /* Elément de l'IDT non présent */)
   SegmentException($NP, VecNumber*8+2);

switch (Attributes.Type) {
5:   /* TaskGate */
   TaskGate(Selector, 1  /* Lien */);/* voir le Chapitre 5 */
   break;
6:   /* IntGate286 */
   intTrapGate286(Selector, Offset, 1  /* Efface IF */);
   break;
```

Listing 6.1 : *Description du traitement d'une interruption (suite).*

```
7:   /* TrapGate286 */
   intTrapGate286(Selector, Offset, 0  /* IF non modifié */);
   break;
14:   /* IntGate386 */
   IntTrapGate(Selector, Offset, 1  /* Efface IF */)
   break;
15:   /*TrapGate386 */
   IntTrapGate(Selector, Offset, 0  /* IF non modifié */);
   break;

}
} /* fin de Interrupt */
```

Listing 6.1 : *Description du traitement d'une interruption (fin).*

On obtient le descripteur de guichet en indexant l'IDT par le numéro de vecteur. Le champ Dtype doit être égal à 0 pour indiquer un segment système ou un descripteur de guichet. Le champ Type doit être égal à 5, 6, 7, 14 ou 15 pour indiquer que le guichet est du type guichet de tâche, guichet d'interruption 286, guichet de trappe 286, guichet d'interruption 386 ou guichet de trappe 386. Si le paramètre TestDPL est égal à 1, le DPL du guichet de la IDT est testé pour vérifier s'il est identique ou d'un niveau externe relativement à CPL. S'il y a un problème, une exception du type segment $GP (protection générale) est générée. Le bit P doit être égal à 1 pour indiquer un descripteur valide. Sinon, il y aura génération d'une exception du type segment $NP (non présent). Les exceptions $GP et $NP fournissent un code d'erreur qui contient le numéro de vecteur dans le champ index du sélecteur, avec le bit 1 du code d'erreur égal à 1 pour signaler que le code d'erreur se rapporte à l'IDT.

Le champ Type dans le guichet détermine comment l'interruption est traitée. Un guichet de tâche entraîne une commutation de tâches vers la tâche décrite par le TSS identifié par le sélecteur du guichet. La routine TaskGate() (voir le Chapitre 5) décrit cette commutation de tâches. Un guichet d'interruption ou de trappe 286 rend nécessaire l'emploi d'un

handler dans la tâche courante et l'utilisation d'un transfert 16 bits compatible avec le traitement des interruptions du 286. Dans le Chapitre 9, nous parlerons des transferts à travers ce type de guichet.

Guichets d'interruption et de trappe

Un guichet d'interruption ou de trappe 386 est associé à un handling dans la tâche courante avec un transfert de contrôle similaire à celui d'un CALL via un guichet d'appel. La routine IntTrapGate() (voir le Listing 6.2) décrit ce transfert de contrôle. Elle reçoit trois paramètres :

1 Un sélecteur pour le segment code contenant le handler.

2 Le déplacement du handler dans ce segment.

3 Un indicateur pour déterminer s'il faut mettre IF à 0 ou non avant le transfert au handler.

```
IntTrapGate(Selector, Offset, ClearIF)
/* Similaire à la routine CallGate du Chapitre 5.*/
   SelectorType    Selector;
   int             Offset,
   ClearIF;
/* si 1, efface IF avant d'entrer dans le handler */
/* si 0, ne modifie pas IF */
{
   SegAttributes   Attributes;
   SelectorType    GSelector;
   int             Base, Limit, GOffset;
```

Listing 6.2 : *Sous-programme IntTrapGate().*

```
/* Test du sélecteur */
if ( (Selector & OFFFCh) == 0)    /* Nul */
   SegmentException($GP, 0);

/* Lecture et test du descripteur */
ReadDescriptor(Selector,&Attributes,&Base,&Limit,&GSelector,
               &GOffset);

if (Attributes.DType == 0)
/* Ne peut pas être un segment système ni un guichet */
   SegmentException($GP, Selector);

/* Interruption vers un segment présent, exécutable, */
/* non conforme  de niveau interne OK, mais demande  */
/*  commutation vers pile interne et CPL interne     */
if ( (Attributes.Type>=8) && (Attributes.Type<=11)
      && (Attributes.DPL < CPL) && (Attributes.P = 1) )
         InnerStack(Attributes.DPL, 0, Selector);

/* Appel classique pour achever le chargement */
/* du descripteur de CS*/
Selector.RPL = 0;
/* Ignore RPL dans la lecture du sélecteur */
CSDescriptorLoad(Selector, Attributes, Base, Limit, $GP);

/* Vérifie que la destination est comprise dans */
/* les limites du segment    */
if (Offset > CS.Limit)
   SegmentException($GP, 0);

/* On arrive ici seulement si tous les tests de       */
/* protection réussissent.                            */
/* Empile EFLAGS et le pointeur de retour, puis       */
/* modifie EFLAGS et charge le sélecteur de CS et EIP.*/
ESP = ESP-4;
```

Listing 6.2 : *Sous-programme IntTrapGate() (suite).*

```
AccessVirtual(SS, ESP, 4, 1  /* Ecriture */, $SS, &EFLAGS);
ESP = ESP-4;
/* Empile 4 octets avec le sélecteur de CS dans les 2  */
/* octets de poids faible.                             */
AccessVirtual(SS, ESP, 4, 1  /* Ecriture */, $SS, &CS.Selector);
ESP = ESP-4;
AccessVirtual(SS, ESP, 4, 1  /* Ecriture */, $SS, &EIP);

EFLAGS.TF = 0;
/* Désactive le mode pas à pas (voir le Chapitre 8)*/
EFLAGS.NT = 0;
/* Interruption non traitée par une tâche imbriquée */
if (ClearIF == 1)
   EFLAGS.IF = 0;

CS.Selector       = Selector;
CS.Selector.RPL   = CPL;
EIP               = Offset;
} /* fin de IntTrapGate */
```

Listing 6.2 : *Sous-programme IntTrapGate() (fin).*

Tout d'abord, la routine teste le sélecteur pour s'assurer qu'il n'est pas nul. Puis elle appelle la routine ReadDescriptor() (voir le Chapitre 5) pour lire le descripteur dans la table des descripteurs. Le champ DType doit être égal à 1 pour indiquer un segment mémoire. Si le descripteur indique un segment exécutable, non conforme et présent, avec DPL d'un niveau interne relativement au CPL, il y a changement de niveau de privilège. Ceci inclut un changement de pile : on appelle donc la routine InnerStack() (voir le Chapitre 5).

Une fois que le niveau de privilège et la pile ont été changés, si cela était nécessaire, IntTrapGate() appelle la routine CSDescriptorLoad() (voir le Chapitre 5) pour poursuivre les tests du descripteur. Cette routine ignore le RPL du sélecteur du guichet puisqu'il a été positionné à 0 avant. Elle rend la main à l'appelant si elle a réussi à charger le descripteur. Si le nou-

veau déplacement est compris dans la limite de CS, la partie "visible" du handling empile EFLAGS CS, et EIP, efface certains champs de EFLAGS et charge CS et EIP avec le sélecteur et le déplacement obtenus à partir du guichet.

Instruction IRET

La routine IRET() (voir le Listing 6.3) décrit l'instruction IRET, qui complète la description du Chapitre 3. D'abord, cette routine teste le bit NT du registre EFLAGS pour déterminer si le retour doit s'effectuer vers une tâche imbriquée (NT = 1) ou dans la tâche courante (NT = 0). Si NT = 1, elle effectue un retour à la tâche imbriquée en lisant le sélecteur du TSS de cette tâche à partir du champ de liens situé au déplacement 0 du TSS courant. Elle appelle ensuite la routine TaskGate() pour effectuer la commutation de tâches.

```
/* Description de l'instruction IRET.*/
IRET()
{
   SelectorType       Selector, GSelector;
   SegAttributes      Attributes;
   int                Base, Limit, Offset, GOffset;

/* Test du bit NT si on fait un retour de tâche */
if (EFLAGS.NT == 1)
   { /* Retour vers une tâche identifiée par le champ */
     /* de lien du TSS courant*/
   AccessLinear(TR.Base, 2, 0 /* PL 0 */, 0 /* Lecture */,
                &Selector);
   TaskGate(Selector, -1 /* Libre */ );
```

Listing 6.3 : *Description de IRET.*

```
      /* voir le Chapitre 5 */
   }

else   {
/* Sinon, dépile les informations de retour d'interruption */
   AccessVirtual(SS, ESP, 4, 0  /* Lecture */, $SS, &Offset);
   ESP = ESP + 4;
   AccessVirtual(SS, ESP, 4, 0  /* Lecture */, $SS,
                 &Selector);
   ESP = ESP + 4;
   AccessVirtual(SS, ESP, 4, 0  /* Lecture */, $SS,
                 &NewEFLAGS);
   ESP = ESP + 4;

   /* Tests du sélecteur pour IRET */
   if ( ((Selector & OFFFCh) == 0) || (Selector.RPL < CPL))
      SegmentException($GP, 0);
   if (Selector.RPL > CPL) {
      /* IRET inter-niveaux si RPL > CPL                    */
      /* Appel à Sous-programme pour récupérer la pile      */
      /* externe à partir du pointeur SS:ESP qui est au     */
      /* sommet de la pile interne.                         */
      OuterStack(Selector.RPL, 0);
      }

   /* Lecture et test du descripteur de CS */
   ReadDescrip-
tor(Selector,&Attributes,&Base,&Limit,&GSelector,&GOffset);
   if (Attributes.DType == 0)
   /* Ne peut pas être un segment Système */
   /*ni un guichet */
      SegmentException($GP, Selector);

   /* Appel classique pour compléter le chargement */
   /* du descripteur de CS */
```

Listing 6.3 : *Description de IRET (suite).*

```
   CSDescriptorLoad(Selector, Attributes, Base, Limit, $GP);

   /* Vérifie si la destination est comprise dans  */
   /*la limite du segment  */
   if (Offset > CS.Limit)
      SegmentException($GP, 0);

   /* On arrive ici seulement si tous les tests de   */
   /* protection sont OK.                            */
   /* On complète la partie visible de l'instruction */
   /* en chargeant EFLAGS, le sélecteur de CS, et EIP.*/
   EFLAGS = NewEFLAGS;
   CS.Selector = Selector;
   EIP = Offset;
   } /* fin NT=0 */
} /* fin de IRET */
```

Listing 6.3 : *Description de IRET (fin).*

Si NT = 0, elle effectue un retour à la tâche courante, et dépile les registres EIP, CS et EFLAGS. Le champ RPL du sélecteur du CS dépilé détermine le niveau de privilège vers lequel le retour s'effectue. S'il indique un niveau de privilège externe relativement à CPL, on aura un changement de niveau de privilège entraînant un changement de segment pile qui s'effectue par un appel à la routine OuterStack() (voir le Chapitre 5). Puis IRET() lit le descripteur de CS, le teste et le charge dans les registres cachés en appelant deux routines décrites au Chapitre 5 : ReadDescriptor() et CSDescriptorLoad(). Finalement, si le déplacement de retour est compris dans la limite de CS, la partie "visible" de l'instruction IRET charge EFLAGS, CS et EIP avec les valeurs précédemment dépilées.

Génération des exceptions

Dans ces paragraphes, nous allons décrire les routines SegmentException() et PageException() utilisées dans le Chapitre 5 pour générer des exceptions du type segment et page. Avant de commencer cette description, nous allons introduire quelques variables globales supplémentaires et une boucle de traitement d'instruction (voir le Listing 6.4). Nous emploierons la variable globale EXT pour former le code d'erreur des exceptions du type segment. Elle est généralement égale à 0 et est mise à 1 lors du traitement d'une interruption externe ou d'une exception autre que INT *n* ou INTO. OldEIP sauvegarde le déplacement de l'instruction courante dans le segment code courant adressé par le registre CS. Si une faute survient, OldEIP est replacé dans EIP, si bien que l'exception est transmise avec l'EIP sauvegardé (dans la pile ou dans le TSS pour une tâche imbriquée) pointant sur l'instruction en cause.

La boucle DispatchInstruction est une boucle de traitement de base. Avant chaque instruction, elle met EXT à 0 et copie EIP dans OldEIP. Puis elle appelle la routine FetchInstruction() (cet ouvrage ne la donne pas) pour traiter l'instruction pointée par EIP. Elle incrémente aussi EIP pour le faire pointer sur l'instruction suivante. Après ce traitement, elle exécute l'instruction par un appel à la routine Execute() (toute routine de ce chapitre ou du Chapitre 5 qui décrit une instruction par exemple IRET()). Cette routine rendra la main si elle a réussi à exécuter l'instruction (sans faute), et on bouclera.

Les routines PageException() et SegmentException() ne retournent pas à l'appelant, mais au contraire, interrompent l'"exécution" de la routine qui décrit l'instruction. Ceci s'effectue par un saut à l'étiquette DispatchInstruction. Ce saut va "purger" la pile des appels de procédure pour achever toute procédure imbriquée avant de relancer l'exécution à l'étiquette DispatchInstruction.

Cette description ne comprend pas la détection ni la transmission des doubles fautes par souci de simplification. Nous en parlerons bientôt.

```
/* EXT et OldEIP sont des variables globales qui sont    */
/* assignées chaque fois qu'une nouvelle instruction est */
/* traitée et exécutée. Cette partie gère le traitement  */
/* de EXT, EIP, et OldEIP à une frontière d'instruction. */

int   EXT,
/* 1 si traitement d'une interruption externe ou d'une */
/* exception, utilisé pour former le code d'erreur des */
/* exceptions du type segment.                         */
      OldEIP;
/* Sauve le EIP de l'instruction courante pour         */
/* permettre le traitement de l'exception.             */

/* Actions lors du traitement et de l'exécution d'une  */
/* instruction. Les routines de traitement des exceptions */
/* arrivent ici pour mettre fin  à une instruction et  */
/* continuer sur la première instruction du handler.   */
DispatchInstruction:
   EXT = 0;
   OldEIP = EIP;
/* Sauve le pointeur de l'instruction courante           */
   FetchInstruction();
/* Incrémente le EIP de la prochaine instruction         */
   Execute();
/* Appels à la bonne routine pour exécuter l'instruction  */
   goto DispatchInstruction;
/* Boucle de traitement et d'exécution */
```

Listing 6.4 : *Variables globales et boucle de traitement.*

Routine SegmentException()

La routine SegmentException() du Listing 6.5 génère des exceptions du type segment. Elle reçoit deux paramètres :

1 Le numéro de vecteur associé.

2 Le sélecteur employé pour former le code d'erreur fourni par l'exception.

```
SegmentException(VecNumber, ErrorCode)
/* Génère une exception du type segment */

   int    VecNumber,    /* Numéro de l'exception */
          ErrorCode;    /* Les 15 bits du code d'erreur */
{
/* Calcule le code d'erreur avant le traitement de  */
/* l'interruption ce qui permet de positionner EXT  */
/* à 1 à ce stade sans affecter le code d'exception.*/
ErrorCode = (ErrorCode & OFFFEh) | EXT;
EXT = 1;

/* Sauvegarde EIP : il pointe sur l'instruction en cause   */
/* EIP est incrémenté après chaque traitement d'instruction*/
/* EIP de l'instruction courante est sauvé dans OldEIP     */
EIP = OldEIP;

/* Signale l'exception avec un numéro de vecteur donné */
/* par VecNumber */
Interrupt(VecNumber, 0
          /* Ne teste pas le DPL du guichet de l'IDT */);

/* Empile le code d'erreur après le traitement de */
/* l'interruption */
ESP = ESP-4;
AccessVirtual(SS, ESP, 4, 1  /*Ecriture */, $SS, &ErrorCode);

/* Termine le traitement de l'instruction en cause         */
/* Continue à la première instruction du handler           */
/* d'exception */
GOTO   DispatchInstruction;
} /* fin de SegmentException */
```

Listing 6.5 : *Sous-programme SegmentException().*

Le code d'erreur est obtenu en concaténant les 15 bits de poids fort (second paramètre) avec le bit EXT. Dans le Chapitre 5, nous donnons le format du code d'erreur pour les exceptions du type segment. Puis la routine copie EIP pour qu'il pointe sur l'instruction en cause en sauvegardant OldEIP (par la boucle de traitement) dans EIP. Après que l'exception a été traitée par la routine Interrupt(), elle empile le code d'erreur. Pour finir, elle interrompt cette instruction en sautant à l'étiquette DispatchInstruction qui poursuivra l'exécution à la première instruction du handler associé.

Routine PageException()

La routine PageException() (voir le Listing 6.6) est similaire à SegmentException() sauf sur deux points : la façon de former le code d'erreur et le numéro de vecteur passé à la routine Interrupt(). Dans le Chapitre 5,

```
PageException(LAddress, U, W, Present) {
   int   LAddress,
   U,        /* 1 si accès utilisateur, 0 si superviseur   */
   W,        /* 1 si accès écriture, 0 si lecture          */
   Present; /* 1 si élément présent, 0 si non présent      */

{int   ErrorCode;

/* Range l'adresse linéaire dans CR2 et prépare le code  */
/* d'erreur basé sur l'accès tenté et sur le bit présent */

CR2= LAddress;
if (U==3)
   ErrorCode = 4 + 2*W + Present;
else   ErrorCode = 2*W + Present;
EXT = 1;
```

Listing 6.6 : *Sous-programme PageException().*

```
/* Sauvegarde EIP pour qu'il pointe sur l'instruction en  */
/* cause. EIP est incrémenté après chaque traitement      */
/* d'instruction. EIP de l'instruction courante est sauvé */
/* dans OldEIP                                            */
EIP = OldEIP;

/* Signale l'exception numéro 14 */
Interrupt(14, 0 /* Ne teste pas le DPL du guichet d'IDT */);

/* Empile le code d'erreur après traitement */
/* de l'interruption */

ESP = ESP-4;
AccessVirtual(SS, ESP, 4, 1 /* Ecriture */, $SS, &ErrorCode);

/* Achève le traitement de l'instruction    */
/* Continue avec la première instruction du */
/* handler d'exception   */
GOTO   DispatchInstruction;
} /* fin de PageException */
```

Listing 6.6 : *Sous-programme PageException().*

nous décrivons le format du code d'erreur pour des exceptions du type page. La routine PageException() reçoit quatre paramètres :

1 L'adresse linéaire en cause.

2 Le niveau de privilège du programme qui a tenté l'accès (pour former le bit U du code d'erreur).

3 Le bit W du code d'erreur (1 si écriture, 0 si lecture).

4 Le bit P du code d'erreur (1 si exception du type protection, 0 si exception du type non présent).

EXCEPTIONS DU TYPE ERREUR DU COPROCESSEUR

Dans les paragraphes suivants, nous allons décrire les exceptions détectées par le coprocesseur arithmétique 80387. Dans le Chapitre 2, nous avons décrit le mot d'état et les registres de contrôle du 80387 qui contiennent des champs importants pour le traitement de ces exceptions. Dans le Chapitre 3, nous avons donné une liste des exceptions qui peuvent survenir au cours d'une instruction.

Nous avons regroupé ces exceptions en six catégories (voir le Tableau 6.4). Nous allons détailler chacune d'elles. Ce tableau donne le code utilisé par les routines du Chapitre 3, le nom du bit correspondant dans le registre d'état, et le bit de masque correspondant dans le registre de contrôle.

Exceptions masquées / Exceptions non masquées

Le bit de masque du registre de contrôle détermine quelle action choisir si une condition exceptionnelle est détectée. Si le bit de masque est égal à 1, l'exception est masquée. Dans ce cas, elle est prise en compte par le 80387 qui transmet un résultat raisonnable et continue sans générer d'exception. Si le bit de masque est égal à 0, l'exception est non masquée et le 80387 la génère avec le numéro de vecteur 16.

Notez que le bit d'état est positionné pour les exceptions masquées, même si le coprocesseur ne génère pas d'exception. Le logiciel peut ainsi déterminer si une exception masquée est survenue lors d'une séquence de calcul numérique en effaçant les bits d'état, en positionnant les bits de masque, en effectuant les calculs, puis en examinant les bits d'état des exceptions masquées.

Si une exception non masquée survient, elle est transmise comme une faute lors de l'instruction à virgule flottante suivante ou du prochain

CODE	ETAT	MASQUE	CONDITION
I,IS	IE	IM	Opération non valide
D	DE	DM	Dénormalisé
Z	ZE	ZM	Division par zéro
O	OE	OM	Débordement (Overflow)
U	UE	UM	Débordement (Underflow)
P	PE	PM	Précision

Tableau 6.4 : *Résumé des exceptions du type erreur du coprocesseur.*

WAIT. Cela signifie que les valeurs de CS et de EIP qui ont été sauvegardées au début du handler pointent sur l'instruction à virgule flottante suivante, et non sur l'instruction à virgule flottante en cause. On peut lire les registres pointeurs d'erreur du 80387 grâce aux instructions FSTENV et FSAVE pour obtenir un pointeur (dans FCS ou FIP) sur l'instruction en cause (sur le CodeOp de cette instruction dans FOS et sur l'opérande mémoire dans FOO).

Les exceptions non masquées vont être classées en exceptions du type préexécution et exceptions du type postexécution. Une exception du type préexécution est générée avant qu'un résultat soit sauvegardé. Elle ne modifie donc pas les opérandes initiaux. Une exception du type postexécution est générée après que le résultat a été sauvegardé, ce qui peut modifier un ou plusieurs opérandes initiaux. Une exception du type préexécution est analogue à une faute, sauf que les valeurs de CS et de EIP qui ont été sauvegardées ne pointent pas sur l'instruction en cause. De même, une exception du type postexécution est analogue à une trappe.

Les exceptions I, IS, D et Z sont toujours des exceptions du type préexécution. L'exception P est toujours une exception du type postexécution. Les exceptions U et O sont du type préexécution pour les instructions de stockage en mémoire (FST(P), FIST(P) et FBST(P)). Sinon,

U et O sont du type postexécution, ce qui englobe les instructions de stockage dans la pile accumulateur.

Le handler associé peut simplement transmettre l'erreur à l'écran ou au debugger et mettre fin au programme en cause. Au contraire, si le handler désire relancer le programme en cause, il doit achever l'instruction à virgule flottante en cours soit en interprétant l'instruction (pour des exceptions du type préexécution), soit en ajustant le résultat (pour des exceptions du type postexécution). Il peut lire le registre d'état pour déterminer la nature de l'exception non masquée survenue, et lire et décoder l'instruction en employant les registres pointeurs d'instruction FCS et FIP. Avant de rendre la main, le handler doit effacer les bits d'état de l'exception non masquée en exécutant une instruction FCLEX, FLDENV, ou FRSTOR.

Dans la plupart des cas, la réponse du 80387 à une exception masquée est la meilleure. Nous recommandons de masquer toutes les exceptions, sauf celle du type opération non valide. Cette dernière survient uniquement lorsque le 80387 ne peut pas fournir un résultat raisonnable qui permettrait de continuer le calcul (addition de + et - , ou division de 0 par 0).

Les catégories d'erreurs du coprocesseur

Dans les paragraphes suivants, nous allons décrire les six catégories d'exceptions du type erreur du coprocesseur. Ces descriptions classent les instructions à virgule flottante en deux groupes : calculatoire ou non calculatoire. Les instructions calculatoires sont les instructions de transfert de données, arithmétique, de comparaison, et de calcul transcendantal (voir le Chapitre 3). Les instructions suivantes sont du type non calculatoire :

1 FLD avec un opérande d'origine mémoire du type réel temporaire ou registre.

2 FST(P) avec un opérande destination mémoire du type réel temporaire ou registre.

3 FXCH, FABS, FCHS, et FXAM.

Exceptions du type opération non valide

Il existe deux catégories d'exceptions du type opération non valide : celles qui sont dues à des erreurs numériques et celles qui sont dues à des débordements de la pile accumulateur du 80387. Le bit SF du registre d'état permet de distinguer les exceptions dues à un débordement de pile (SF = 1) de celles dues à une erreur numérique (SF = 0). Ces exceptions sont du type préexécution, si bien que les exceptions non masquées sont transmises avant que le résultat soit sauvegardé.

Opération numérique non valide (SF = 0)

Tous les cas du Tableau 6.5 génèrent une exception du type opération numérique non valide. Si cette dernière est masquée, c'est une valeur indéfinie du type NaN silencieux qui est retournée (sauf pour FIST(P) qui donne un entier indéfini et FBSTP qui donne un BCD indéfini). Pour les opérations qui donnent deux résultats, les deux seront du type NaN silencieux indéfini.

Débordement de la pile accumulateur (SF = 1)

Une exception du type opération non valide est générée quand une instruction tente de lire un accumulateur vide ou d'empiler un résultat dans la pile accumulateur lorsque le nouveau sommet de pile n'est pas vide. Notez que les instructions FST(P) et FXCH avec un opérande destination du type registre ne sont pas considérées comme des opérations sur pile, si bien qu'elles peuvent écrire dans un registre non vide. Le bit C1 du registre d'état détermine si on a eu un underflow (C1 = 0) ou un overflow (C1 = 1).

CONDITIONS D'OPERATIONS NON VALIDES
Opérande d'une instruction calculatoire dans un format non permis
Opérande d'une instruction calculatoire du type NaN signalé
Opérande de FCOM ou FTST du type NaN silencieux ou signalé
Opérande de FUCOM du type NaN signalé [1]
Opérandes de FADD du type infini avec signes opposés
Opérandes de FSUB du type infini avec même signe
Opérandes de FMUL du type zéro et infini
Opérandes de FDIV tous les deux du type infini ou zéro
Diviseur nul ou dividende infini pour FPREM ou FPREM1 [2]
Opérande de FCOS, FPTAN, FSIN ou FSINCOS du type infini
Opérande de FSQRT ou FYL2X plus petit que 0
Opérande de FYL2XP1 plus petit que -1
Opérande de FIST ou FBSTP du type NaN ou infini ou ne rentre pas dans la destination

[1] Si l'exception est masquée, les codes de condition sont positionnés C3, C2, C0 à 1.

[2] Si l'exception est masquée, C2 est mis à 1.

Tableau 6.5 : *Résumé des opérations numériques non valides.*

Exceptions du type dénormalisé

Si l'opérande source d'une instruction calculatoire est du type dénormalisé, il y a génération d'une exception du type dénormalisé. Si cette dernière est masquée, le bit DE du registre d'état est positionné et l'instruction est exécutée avec le nombre dénormalisé. Dans ce cas, l'emploi d'un opérande mémoire dénormalisé du type réel long ou court provoquera une conversion de l'opérande en réel temporaire normalisé. En effet, un réel temporaire normalisé peut contenir un réel dénormalisé court ou long.

Si l'exception est non masquée, le bit DE du registre d'état est positionné, une exception du type coprocesseur est générée et aucun résultat n'est sauvegardé, puisque cette exception est du type préexception.

Puisque les dénormalisés ont beaucoup moins de bits significatifs que les nombres normalisés, vous devrez éviter l'emploi de dénormalisés dans certains algorithmes. Cependant, dans la plupart des cas, cette exception devra être masquée pour que le calcul s'effectue avec un opérande dénormalisé.

Exceptions du type division par zéro

Une instruction FDIV ou FYL2X qui tente de diviser par zéro un opérande fini, ou une instruction FXTRACT qui a un opérande nul génèrent une exception du type division par zéro. Si l'exception est masquée, le résultat de FDIV ou de FYL2X est égal à l'infini avec un signe qui est le *ou* exclusif des signes des opérandes initiaux. Pour FXTRACT, on obtient dans ce cas - ∞. Dans tous les cas, le bit ZE du mot d'état est positionné. Si l'exception n'est pas masquée, le bit ZE est positionné et une exception est générée. Dans ce cas, aucun résultat n'est sauvegardé puisque cette exception est du type préexécution.

Exceptions du type overflow

Si le résultat réel est trop grand pour tenir dans l'opérande de destination, il y a transmission d'une exception du type overflow (débordement). Si cette exception est masquée, le bit OE du registre d'état est positionné. De plus, le résultat sauvegardé dépend du mode d'arrondi sélectionné par le champ RC des registres de contrôle (voir le Tableau 6.6).

Si l'exception n'est pas masquée, le bit OE est positionné et il y a génération d'une exception du type coprocesseur. Si un overflow survient au cours d'une instruction FST qui range son opérande en mémoire, aucun ré-

MODE D'ARRONDI	SIGNE RESULTAT	RESULTAT MASQUE
Au plus près	+	+∞
	-	-∞
Vers -∞	+	Le plus grand nombre >0
	-	-∞
Vers +∞	+	+∞
	-	Le plus grand nombre <0
Vers 0	+	Le plus grand nombre >0
	-	Le plus grand nombre <0

Tableau 6.6 : *Résultats masqués lors d'un overflow.*

sultat n'est sauvegardé. Ceci permet au handler associé de traiter convenablement la donnée initiale qui se trouve encore dans la pile accumulateur.

Si la destination est la pile accumulateur, le résultat est sauvegardé avant que l'exception ne soit générée. Cette valeur est un résultat réel divisé par 2^{24576} après arrondi de la mantisse. Notez que $24576 = 3*2^{13}$. Ceci permet de repousser les limites de l'exposant.

Exceptions du type underflow

Si le résultat réel est trop petit pour tenir dans l'opérande de destination, il y a transmission d'une exception du type underflow (débordement). Si cette exception est masquée, le résultat est dénormalisé et rangé dans l'opérande destination. Le bit UE du registre d'état est positionné uniquement si ce résultat dénormalisé provoque une perte de précision, auquel cas une exception du type précision est aussi générée.

Si l'exception n'est pas masquée, le bit UE est positionné (qu'il y ait perte de précision ou non) et il y a génération d'une exception du type coprocesseur. Si un débordement survient au cours d'une instruction FST qui range son opérande en mémoire, aucun résultat n'est sauvegardé. Ainsi, le handler associé peut traiter convenablement la donnée initiale qui se trouve encore dans la pile accumulateur.

Si la destination est la pile accumulateur, le résultat est sauvegardé avant que l'exception ne soit générée. Cette valeur est un résultat réel divisé par 2^{24576} après arrondi de la mantisse. Notez que $24576 = 3*2^{13}$. Ceci permet de repousser les limites de l'exposant.

Exceptions du type précision (IEEE inexact)

Si le résultat ne peut pas tenir dans le format de l'opérande destination, ou si un underflow a entraîné une perte de précision, une exception du type précision est générée. Notez que le bit C1 du registre d'état indique le sens du dernier arrondi dans les instructions calculatoires. C1 = 0 indique un arrondi vers le bas, si bien que le résultat final est plus petit que le résultat réel. C1 = 1 indique un arrondi vers le haut, si bien que le résultat final est plus grand que le résultat réel. Si l'exception est masquée, le bit PE est positionné dans le registre d'état, et le résultat arrondi est sauvé dans l'opérande destination. Si l'exception est non masquée, le bit PE est positionné, le résultat arrondi est sauvé, et une exception du type coprocesseur est générée.

Ce type d'exception permet de savoir si un calcul est inexact. Ceci est important dans de nombreux cas, par exemple lorsqu'on travaille avec des unités monétaires. Cette exception devrait être masquée dans la plupart des applications.

Priorité des exceptions du type erreur du coprocesseur

Si plus d'une condition d'exception est détectée lors d'une instruction, une seule est générée. Le Tableau 6.7 détermine l'ordre de priorité des exceptions. Notez qu'une exception du type précision peut être générée avec une exception du type underflow ou overflow.

EXCEPTION	PRIORITE
Opération non valide	La plus haute
- UnderFlow de pile	
- OverFlow de pile	
- Format non autorisé	
- NaN signalé	
- Autres	
Division par zéro	
Dénormalisé	
Overflow/Underflow	
Précision	La plus basse

Tableau 6.7 : *Priorité des exceptions du type erreur du coprocesseur.*

7

Mise en oeuvre du système d'exploitation

Les programmeurs d'applications ont pu trouver dans le Chapitre 4 quelques exemples des possibilités du 80386 qui les intéressent. Dans les Chapitres 1, 2, et en partie dans le Chapitre 3, nous avions décrit ces possibilités. Quant aux programmeurs de systèmes d'exploitation, ils ont été plus particulièrement intéressés par les Chapitres 5, 6, et en partie par le Chapitre 3. Dans ce chapitre, nous allons donner quelques exemples qui mettent en évidence les caractéristiques du 80386 propres aux systèmes d'exploitation.

SYNTAXE

Comme dans le Chapitre 4, il nous faut compléter la syntaxe assembleur que nous avions définie. La plupart de ces structures sont liées aux définitions de segments.

SEGMENT/ENDS

SEGMENT et ENDS sont des directives assembleur qui définissent le début et la fin d'un segment. De telles directives ne génèrent pas de code, mais renseignent l'assembleur sur la façon de générer le code.

USE32/USE16

USE est une directive assembleur qui indique si le contenu du segment associé est constitué de données ou de code sur 32 bits ou sur 16 bits. L'assembleur générera du code dont le descripteur de segment aura le bit D égal à 1 si ce segment est du type USE32. Notez que le programmeur peut avoir fait une erreur (l'assembleur n'effectue pas de test de validité) et ne pas avoir positionné le bit D du descripteur. Dans ce cas, le programme

ne pourra pas s'exécuter correctement puisque la taille du code ou des données sera incorrecte.

ORG

ORG est une directive assembleur qui définit le déplacement dans un segment.

ASSUME

ASSUME est la dernière directive que nous présenterons. Elle n'a de sens que pour les segments code. Elle indique quel est le registre segment associé au segment. Sans cette directive, l'assembleur générerait des messages d'avertissement indiquant que le segment adressé n'est pas couramment adressable. Cette directive ne génère pas le code qui charge le registre segment indiqué dans le segment spécifié.

Exemple de syntaxe

Voyons un bref exemple qui emploie les directives précédentes :

```
Test_Code_Seg   SEGMENT USE32
ASSUME DS:Mon_Segment
      MOV   AX,Mon_Segment_Sel
      MOV   DS,AX
      ;  Code1
      ORG   0FF0h
      ;  Code2
Test_Code_Seg   ENDS
```

Un segment appelé Test_Code_Seg est défini par les directives SEGMENT et ENDS. C'est un segment 32 bits d'après la directive USE32. Ce segment de code est assemblé avec DS qui adresse le segment Mon_Segment. Les deux premières lignes de code du segment remplissent cette condition. Puis l'origine d'une partie du code est située au déplacement 0FF0h dans le segment.

EXEMPLE D'INITIALISATION

Le premier exemple sera un exemple d'initialisation qui transforme une machine réinitialisée en machine 32 bits, avec activation de la pagination pour travailler en multitâche.

Vue d'ensemble de l'exemple 1

Avant de commencer la description du programme, nous allons fixer l'état final de la machine auquel nous voulons aboutir. Ensuite, toutes les explications sur le fonctionnement du programme seront plus faciles à comprendre.

Cet exemple est une simple partie d'un système d'exploitation. Un système d'exploitation réel est composé de milliers, voire de millions de lignes de code (et nous n'allons pas décrire ici un million de lignes...). Notre exemple n'est pas complet, mais il permet de présenter la plupart des instructions et des fonctions nécessaires à un système d'exploitation.

Hypothèses

Nous allons faire quelques hypothèses sur le matériel :

1 Le programme du système d'exploitation est en ROM (mémoire morte) à l'adresse physique la plus haute. La ROM a une taille de 64 Ko. (elle commence donc à FFFF0000h et finit à FFFFFFFFh).

2 La RAM (mémoire vive) commence à l'adresse 0, et la machine possède au moins 48 Ko de mémoire. Notre exemple ne détermine pas la quantité de mémoire disponible, ce qui est obligatoire pour un système d'exploitation complet. Il considère seulement que 48 Ko sont disponibles.

Hormis ces hypothèses, notre exemple est complet.

Gestion multitâche et protection

Notre exemple devra construire les tables nécessaires au système d'exploitation, et pour terminer exécutera une tâche utilisateur. Cependant, nous avons implanté le concept de gestion multitâche. Cet exemple gère deux niveaux de protection : 0 pour le système d'exploitation et 3 pour le programme d'application. Cependant, le système d'exploitation est typiquement partagé entre toutes les tâches du système. Chaque tâche possède ses propres zones code et données, qui sont placées dans l'espace adressable local et sont donc invisibles pour les autres tâches du système. Le système d'exploitation est partagé entre toutes les tâches et se trouve donc dans l'espace adressable global. La Figure 7.1 résume ces notions. Les cercles concentriques représentent les niveaux de protection (nous n'employons que 0 et 3) et les lignes radiales séparent les tâches du système.

Chaque tranche radiale délimite une tâche. Il n'existe aucun lien entre une adresse de la tâche 1 et une même adresse de la tâche 2, si ce n'est dans les régions globales de l'espace adressable virtuel de chaque tâche.

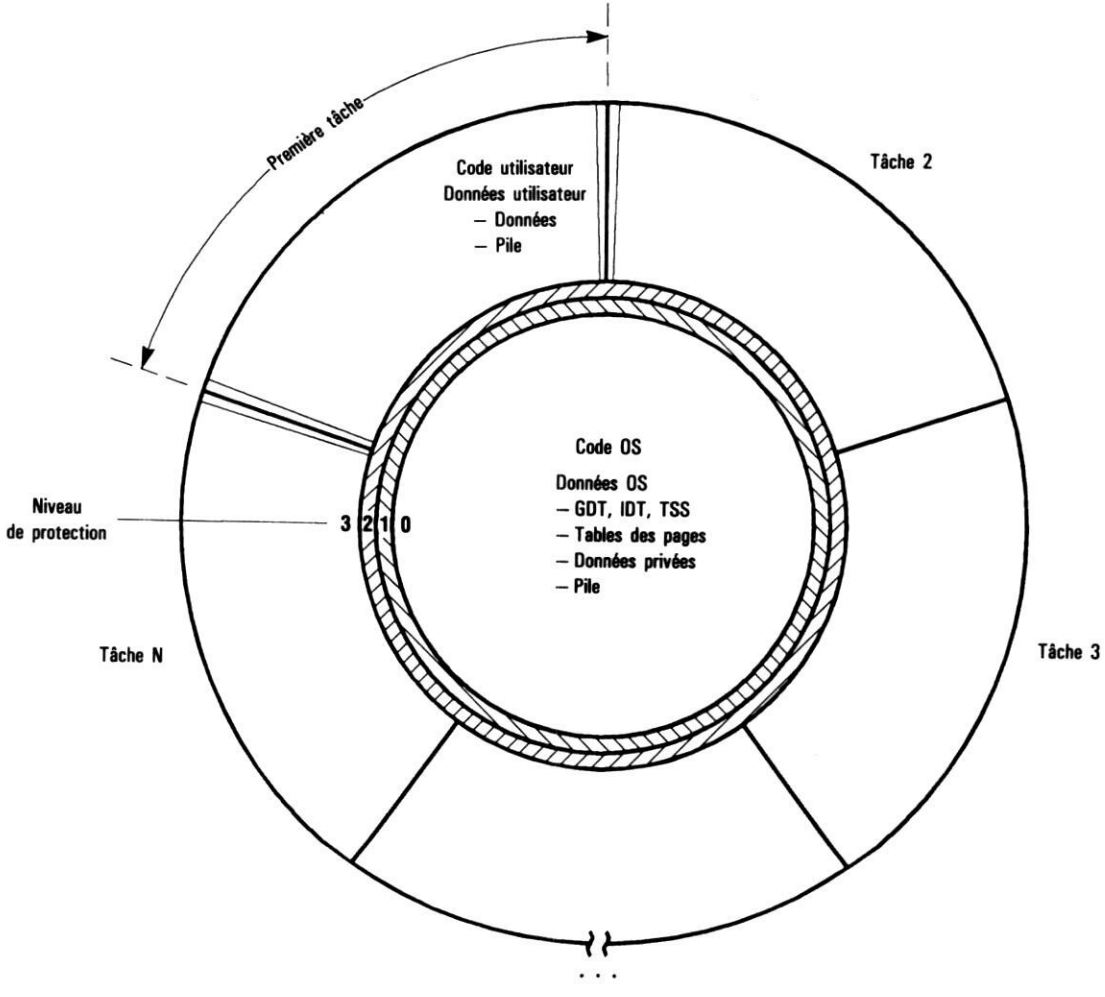


Figure 7.1 : *Système multitâche.*

Espace adressable virtuel

La Figure 7.2 décrit l'espace adressable virtuel de chaque tâche. Ce dernier est composé de l'espace adressable global associé à la GDT et de l'espace adressable local associé à la LDT.

L'espace adressable global contient cinq segments :

1 La GDT elle-même, qui contient les descripteurs définissant les quatre autres segments de cet espace adressable global.

2 Le TSS de la première tâche. Lorsque plusieurs tâches s'exécutent, le système définit un TSS pour chacune d'elles.

3 Le segment des données du système d'exploitation.

4 Le segment code du système d'exploitation.

5 La LDT de la première tâche, qui est associée aux segments de l'espace adressable local. Lorsque plusieurs tâches s'exécutent, le système définit une LDT pour chacune d'elles.

L'espace adressable local contient deux segments :

1 Le segment code utilisateur.

2 Le segment des données utilisateur qui contient les données et la pile.

Les applications ne voient qu'un modèle à segment unique contenant un segment de données unique et un segment code unique (voir le Chapitre 5).

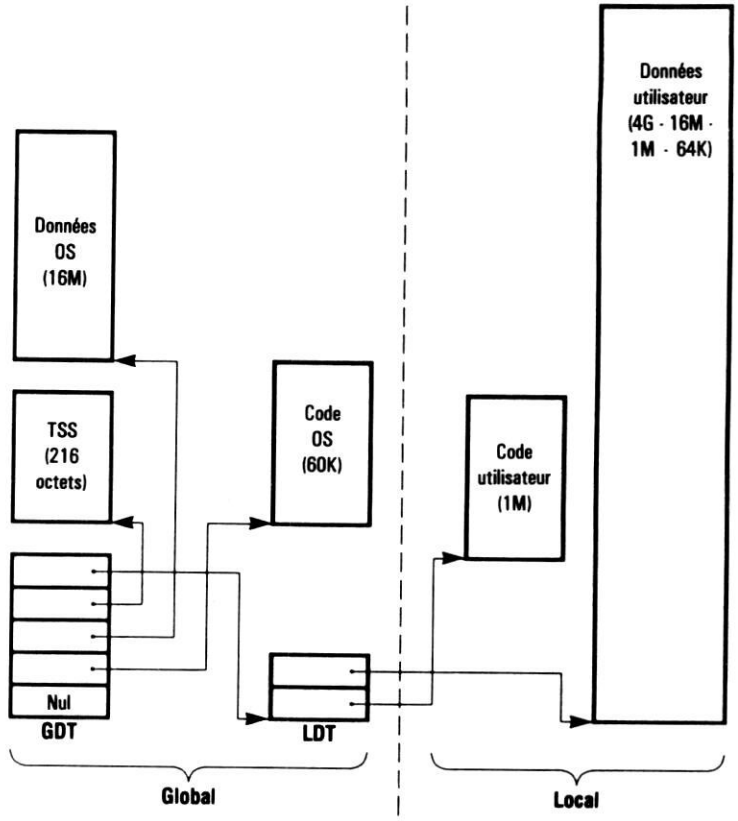


Figure 7.2 : *Espace adressable virtuel.*

Espace adressable linéaire

La Figure 7.3 montre l'espace adressable linéaire. Ce dernier est constitué de quatre segments :

1 Les données du système d'exploitation (OS Data), un segment de 16 Mo commençant à l'adresse linéaire 0 et finissant à 00FFFFFFh. Ce segment contient les tables du système d'exploitation (GDT, IDT, TSS), les tables de pagination (catalogue et tables de second niveau), des données locales, et la pile du système d'exploitation.

2 Le code utilisateur, un segment de 1 Mo commençant à l'adresse linéaire 01000000h et finissant à 010FFFFFh. Le choix d'une taille de 1 Mo est arbitraire. Lorsque la tâche sera chargée de façon effective, le système connaîtra exactement la taille du code segment et ajustera ainsi la limite du segment.

3 Les données utilisateur. Toute la mémoire qui n'est pas utilisée par le système d'exploitation ou par le code utilisateur est donnée au programme utilisateur comme un gigantesque segment de données. Le segment commence à l'adresse linéaire 01100000h et finit à FFFEFFFh. Ainsi, la taille de ce segment est :

4 Go - 16 Mo - 1 Mo - 64 Ko

(un grand tableau, n'est-ce pas ?). Dans ce segment, la pile commence au sommet et croît vers le bas. Les données commencent en bas et croissent vers le haut.

4 Le code du système d'exploitation (OS Code). Ce segment commence à l'adresse linéaire FFFF0000h et finit à FFFFEFFFh (le sommet de l'espace mémoire virtuel d'une tâche). Dans cet exemple, tout le code du système d'exploitation tient dans une ROM de 64 Ko (ce qui n'est pas vrai pour un système d'exploitation réel, mais c'est bien pratique pour notre exemple).

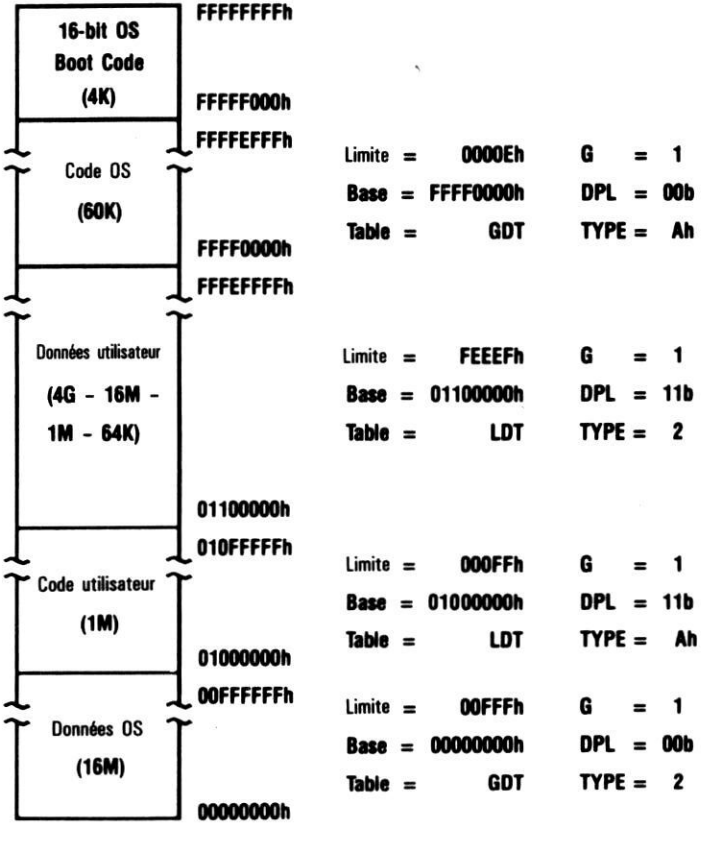


Figure 7.3 : *Espace adressable linéaire de chaque tâche.*

5 Le segment code du programme de lancement (*boot*) commence à l'adresse FFFFF000h et finit à FFFFFFFFh. Ainsi, les 4 Ko supérieurs de la ROM sont consacrés au code 16 bits de lancement. Notez que, dans notre exemple, on ne peut pas adresser ce segment une fois que la protection a été mise en place.

Le choix de l'emplacement de ces segments dans l'espace adressable linéaire est totalement arbitraire, sauf pour le code de lancement qui doit toujours commencer à l'adresse physique FFFFFFF0h. On a choisi le mapping du code et des données du système d'exploitation afin que les adresses linéaires et physiques soient identiques. Nous justifierons bientôt tous ces choix. Les adresses linéaires des segments code et données des applications ont été choisies arbitrairement.

Pagination

L'espace adressable linéaire - l'espace des adresses obtenues après transformation par le mécanisme de segmentation - est associé aux tables des pages de la Figure 7.4. Cette dernière montre l'espace adressable linéaire décomposé en régions de 4 Mo. C'est la quantité de mémoire qui est associée à une table des pages (1 Ko d'éléments de pages avec 4 Ko par page). Nous avons créé cinq tables dont les pointeurs proviennent du catalogue des pages. Un sixième élément du catalogue pointe sur le catalogue lui-même. C'est un moyen simple d'accéder aux tables des pages en les associant elles-mêmes à l'espace adressable linéaire. Ces six éléments du catalogue sont :

1 Les données communes du système d'exploitation. En particulier les tables GDT, IDT et TSS.

2 Le catalogue des pages et les tables des pages.

3 Les données propres au système d'exploitation. La pile du système d'exploitation et quelques données.

4 Le code et les données utilisateur.

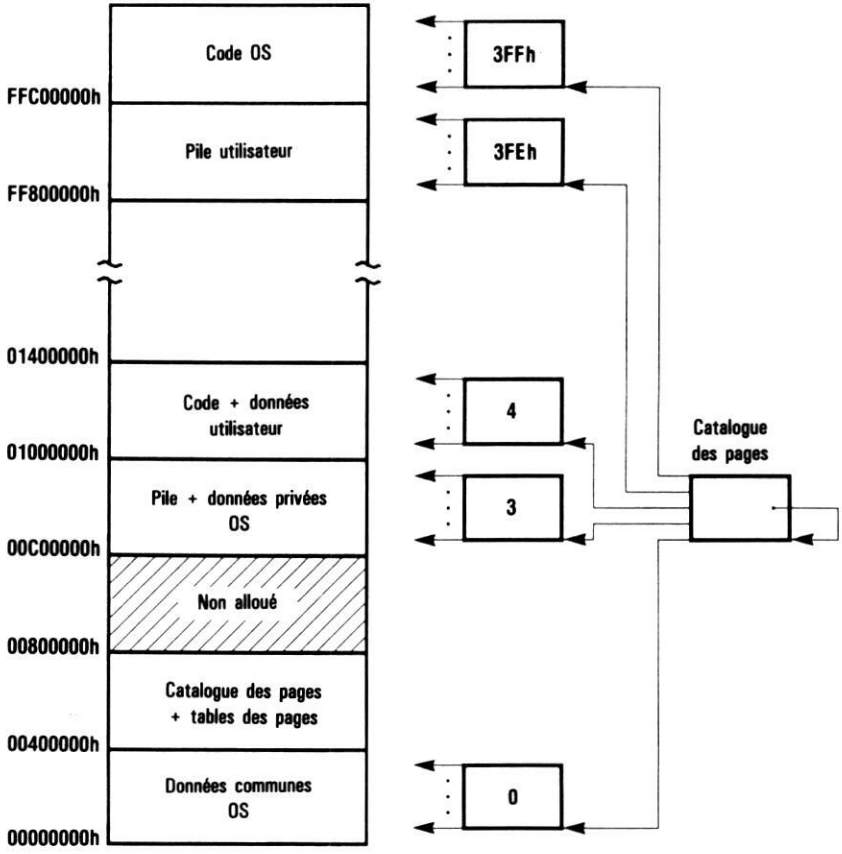


Figure 7.4 : *Tables des pages associées à l'espace adressable linéaire.*

5 La pile utilisateur.

6 Le code du système d'exploitation.

Adresses physiques

La Figure 7.5 représente l'espace adressable physique, soit un total de 28 pages (chacune de 4 Ko). Les 12 pages inférieures sont définies dans la zone RAM (mémoire vive) qui commence à l'adresse 0 et finit à la page débutant à l'adresse 0B000h. Il reste 16 pages qui commencent à l'adresse 0FFFF0000h. Elles représentent exactement les 64 Ko de ROM dont nous avons déjà parlé.

Comparons maintenant l'espace adressable linéaire et l'espace adressable physique.

1 Le code du système d'exploitation est situé dans les 64 Ko supérieurs dans les deux cas, ce qui est bien utile puisque la machine est en cours d'initialisation. Le code aurait pu se trouver à des adresses différentes dans les deux espaces adressables. Mais l'exemple aurait alors été beaucoup moins compréhensible et moins facile à écrire ! C'est surtout vrai pour la protection et la pagination.

2 Les tables GDT, IDT, TSS et LDT sont situées aux mêmes adresses dans les deux cas. Sinon, la mise en place de la protection puis de la pagination nous obligerait à recréer ou à déplacer ces tables, puis à recharger les descripteurs. Pour éviter cette complication, nous les avons gardées aux mêmes adresses. Ainsi, nous les construisons puis les chargeons une fois pour toutes.

3 Les autres pages ne sont pas situées aux mêmes adresses, afin de réduire la quantité de mémoire physique nécessaire au mapping de l'espace adressable linéaire. C'est un des avantages importants de la pagination : l'espace adressable linéaire est plus grand que l'espace

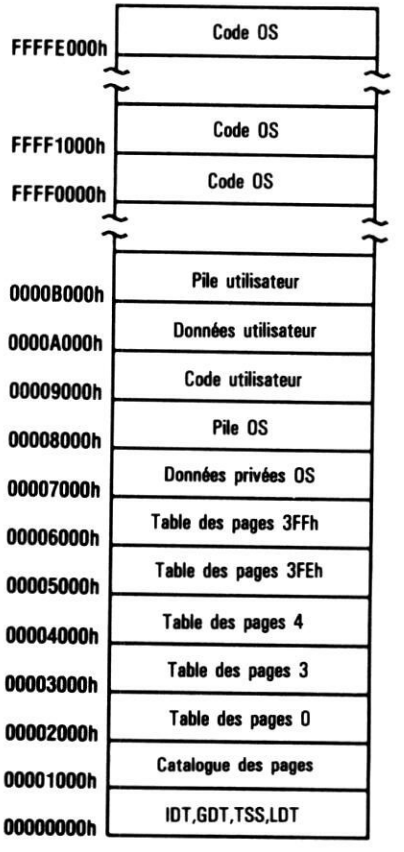


Figure 7.5 : *Espace adressable physique.*

mémoire physique. Cependant, nous avons gardé approximativement le même ordre d'apparition des pages. Sauf pour les cas précédents, l'emplacement des pages en mémoire physique est arbitraire. En fait, on pourra déplacer ces pages puisque la pagination remplace certaines pages inutilisées par d'autres pages. Non seulement nous pouvons permuter certaines parties de l'espace adressable linéaire par la pagination, mais nous pouvons aussi paginer les tables des pages (ce qui peut être nécessaire pour réduire la quantité de mémoire physique consacrée au système d'exploitation, puisque les tables des pages peuvent être très importantes). Pour paginer ces tables des pages, nous devons faire attention lors des mises à jour des éléments non valides et les distinguer des éléments classiques. Le catalogue des pages doit rester au même endroit et être toujours présent. On doit modifier le registre de base du catalogue (CR3) si on désire modifier l'emplacement de ce catalogue.

Commutation de tâches

Nous n'allons pas approfondir la notion de construction et de commutation d'autres tâches (par exemple, ajouter une nouvelle tâche au système), mais il est utile d'en connaître les grandes lignes :

1 On doit construire une LDT pour la nouvelle tâche. Cette table sera identique à celle de l'exemple, à part la taille du code. Un élément de la GDT sera associé (et donc créé) à cette LDT.

2 On doit construire un TSS. Il est identique à celui de l'exemple, si ce n'est que la LDT est identique à la LDT précédente. Un élément de la GDT sera associé (et donc créé) à ce TSS.

3 On doit construire un catalogue des pages. Les tables des pages associées à l'espace adressable global sont identiques à celles de l'exemple et les éléments du catalogue sont partagés entre toutes les tâches. Les éléments du catalogue associés aux tables des pages locales (code, données et pile utilisateur) sont bien sûr différents.

4 On doit construire les tables des pages de l'espace adressable local pour la nouvelle tâche. Elles sont identiques à celles de la première tâche, si ce n'est qu'elles sont associées à des pages physiques différentes.

5 Pour commuter sur cette nouvelle tâche, on doit charger TR, LDTR, et CR0 (voir le Chapitre 5).

Description de l'exemple d'initialisation

Nous pouvons maintenant nous plonger dans la description de cet exemple (voir le Listing 7.1) et le décomposer en plusieurs exemples plus courts qui peuvent être exécutés séquentiellement.

Equivalences

La première partie de l'exemple définit simplement quelques mnémoniques. Les sept sélecteurs (cinq dans la GDT et deux dans la LDT) y sont définis, ainsi que les adresses physiques de toutes les pages. Puisque les adresses physiques et les adresses linéaires ne correspondent pas toutes, nous définissons aussi les mnémoniques des adresses physiques. Notez que les adresses physiques correspondent à la Figure 7.5 qui représente l'espace mémoire physique.

Définitions des segments

Nous définissons ici les quatre segments de l'exemple (OS_Data_Seg, User_Code_Seg, User_Data_Seg et OS_Code_Seg). Ces définitions correspondent à l'image des espaces adressables virtuel et linéaire (voir les Figures 7.2 et 7.3). Elles nous permettent d'utiliser des références symbo-

liques à ces segments. Cependant, ce type de déclaration n'initialise pas ces segments mémoire.

Notez aussi que le segment OS_Code ne couvre pas les 4 Ko supérieurs de la mémoire. Ils sont réservés à un segment particulier (BootRom16). Ce segment est un segment 16 bits alors que OS_Code_Seg est un segment 32 bits. Puisque le 80386 démarre en mode 16 bits, cette petite région de code est indispensable pour passer en mode 32 bits. Après être passé en mode protégé, ce segment 16 bits n'est plus adressable. L'exemple minimise le temps passé à assurer la compatibilité 16 bits puisque le but de cet ouvrage est de décrire les possibilités 32 bits du 80386.

Séquence de Code 1 - Démarrage à froid

A l'initialisation, le 80386 commence à fonctionner en *mode réel* (voir le Chapitre 9). Le mode réel est simplement une machine 16 bits avec une taille maximale de 64 Ko pour un segment. En mode réel, le sélecteur est décalé quatre fois à gauche et ajouté au déplacement dans le segment pour former l'adresse physique (voir la Figure 9.2). Ainsi, la plus grande adresse physique est 0FFFFFh, ce qui donne un espace adressable total de 1 Mo. Le Tableau 9.3 donne l'état de la machine au démarrage. Nous ferons référence à cet état initial dans les lignes qui suivent.

Au début, le registre CS est initialisé à F000h et IP à FFF0h. Ainsi, l'adresse de la première instruction qui sera traitée est FFFF0h. De plus, le 80386 positionne les bits d'adresses A31 à A20 de façon que la première instruction traitée soit celle de l'adresse FFFFFFF0h. Ainsi, le segment code 16 bits est situé à l'adresse FFFF0000h pour correspondre à la valeur initiale de CS et l'ORG du code de démarrage à froid (JMP Start16) est situé au déplacement 0FFF0h. Les bits d'adresses A31 à A20 restent verrouillés jusqu'au premier transfert de contrôle intersegment. La plupart des machines commencent l'exécution à l'adresse la plus haute, ce qui permet d'y placer la ROM de lancement.

Séquence de Code 2 - Divers

Cette séquence commence à l'étiquette *Start16*. La première instruction efface le bit IF (autorisation des interruptions), masquant ainsi toutes les interruptions masquables. Il n'est pas vraiment nécessaire d'effacer IF puisque la séquence de démarrage le fait aussi. Le programme positionne la limite de la IDT (table des descripteurs d'interruptions) à 0. Ainsi, toute interruption non masquable générera un shutdown (voir le Chapitre 6). Puisque l'instruction LIDT demande un opérande mémoire, nous devons d'abord initialiser le registre DS pour le faire pointer sur OS_Data_Seg. Puisque le registre DS est initialisé à 0 et que OS_Data_Seg se trouve à l'adresse 0, ce n'était pas nécessaire. Mais dans la plupart des cas, il faut charger un registre segment avant d'adresser le segment associé.

Séquence de Code 3 - Construction de GDT, LGDT

Cette troisième partie de code construit la GDT (table des descripteurs globaux) et charge le pseudo-descripteur de la GDT dans le processeur.

La première étape consiste à remplir la GDT avec l'information associée. Dans cet exemple, nous avons cinq éléments dans la GDT (Nul, OS Code, OS Data, LDT et TSS). Le Tableau 7.1 résume le contenu de ces cinq descripteurs. Le programme montre quelle est la décomposition adéquate de cette information selon le format du descripteur (voir la Figure 5.10). Le Tableau 7.2 rappelle les différents types de descripteurs utilisés dans cet exemple.

La dernière étape consiste à construire un pseudo-descripteur en mémoire. Le mot de poids faible est la limite de la GDT. La limite est égale à huit fois le nombre d'éléments de la GDT moins 1, le plus grand déplacement valide dans la GDT. Le dmot de poids fort du pseudo-descripteur est l'adresse linéaire de la GDT. Ce pseudo-descripteur est ensuite chargé dans le processeur par une instruction LGDT.

NOM	TABLE	LIMITE	BASE	TYPE	DT	DPL	P	D	G
Nul	GDT	00000	00000000	0	0	0	0	0	0
OS Code	GDT	0000E	FFFF0000	A	1	0	1	1	1
OS Data	GDT	FFFFF	00000000	2	1	0	1	1	1
Util Code	LDT	000FF	01000000	A	1	3	1	1	1
Util Data	LDT	FEEEF	01100000	2	1	3	1	1	1
TSS1	GDT	000D7	000000B0	9	0	0	1	0*	0
LDT1	GDT	0000F	00000188	2	0	0	1	0*	0

* *Note :* Pour un segment système, le bit D n'a aucun sens.

Tableau 7.1 : *Résumé du contenu des descripteurs.*

Séquence de Code 4 - Passage en mode protégé

La prochaine étape positionne le bit PE (activation de la protection) du mot d'état. Après cette opération, il faut effectuer immédiatement un saut. Ceci permet de vider la file des instructions prétraitées en mode réel. La machine est maintenant en mode protégé 16 bits. Pour nous, ce mode est identique à celui dont nous avons déjà parlé dans cet ouvrage, si ce n'est que toutes les données et déplacements sont sous 16 bits. Dans le Chapitre 9, nous présenterons plus précisément ce mode de fonctionnement.

Après être passé en mode protégé, le programme exécute un JMP lointain (far). Le JMP fait référence à Start32 dans le segment OS_Code. Puisque l'élément de la GDT du segment OS_Code définit un segment code 32 bits (voir le bit D du descripteur), la machine passera en mode 32 bits après ce JMP. A ce stade, la machine est (enfin) dans l'état adéquat.

TYPE	DT	DESCRIPTION
2	1	Segment mémoire - Lecture/Ecriture
A	1	Segment mémoire - Lecture/Exécution
9	0	Segment système - TSS 386 Disponible
2	0	Segment système - LDT

Tableau 7.2 : *Types de descripteurs.*

Séquence de Code 5 - Chargement des descripteurs de segments

Cette séquence commence à l'étiquette *Start32*. Puisque la machine est maintenant en mode protégé, il faut recharger les registres segments pour rendre les segments de données adressables. Notez la différence entre le chargement du registre DS à ce stade et celui qui a été effectué à la séquence de code 2, lorsque la machine était encore en mode réel. Lors du chargement du registre DS en mode réel, la valeur du sélecteur chargée dans DS est simplement décalée de 4 et ajoutée au déplacement pour former l'adresse physique finale. Maintenant que nous sommes en mode protégé, charger un sélecteur dans DS entraîne le chargement du descripteur obtenu dans la GDT (voir le Chapitre 5). Le sélecteur est aussi chargé dans ES pour être utilisé par les instructions travaillant sur les chaînes. Plus tard, SS sera chargé avec ce sélecteur.

Séquence de Code 6 - Construction de la LDT, LLDT

L'espace adressable local de la tâche est associé à la LDT (table des descripteurs locaux). La LDT contient deux éléments, un pour le segment code utilisateur et un pour le segment de données utilisateur (voir la Figure 7.2). Lorsque nous avons construit la GDT, nous avons inclus un

élément de la LDT qui définissait une LDT à deux éléments. Le segment LDT est une partie du segment OS_Data. L'exemple utilise la technique des alias dans le segment OS_Data pour accéder à la LDT, alors que le 80386 emploie le véritable segment. La première partie de l'exemple remplit la LDT avec l'information appropriée pour ces deux segments (code et données utilisateur). Le programme montre la décomposition de l'information contenue dans le descripteur selon le format de ce descripteur. Le Tableau 7.1 résume cette information pour les deux segments associés à la LDT. Après la construction de la LDT, le programme charge le sélecteur de la LDT dans LDTR.

Une bonne partie de cette initialisation peut être effectuée soit avant, soit après le passage en mode protégé. Puisque notre but était de réduire la quantité de code 16 bits, nous avons effectué toutes les initialisations, excepté celle de la GDT, après être passé en mode protégé. Puisqu'on ne peut charger LDTR qu'en mode protégé, nous n'avions pas le choix dans ce cas-là.

Séquence de Code 7 - Construction de la l'IDT

Construire l'IDT (table des descripteurs d'interruptions) est une phase très importante de l'initialisation. Dans notre exemple, après passage en mode utilisateur (niveau 3), le mécanisme de traitement des exceptions est le seul moyen dont dispose la tâche pour accéder au système d'exploitation. Les guichets (voir le Chapitre 5) permettent à des niveaux de privilège interne d'accéder à des niveaux de privilège externe, mais nous n'avons pas défini de guichets dans notre exemple.

Les éléments de la l'IDT sont eux-mêmes des guichets d'interruption. Puisque dans notre modèle de machine la tâche courante peut "voir" le système d'exploitation complet, nous n'avons pas besoin d'utiliser un guichet de tâche pour les éléments de l'IDT. La Figure 5.12 donne le format des descripteurs de guichets. Le tableau suivant résume le contenu de notre descripteur de guichets d'interruption. Si une exception ou une interruption est prise en compte au niveau 3, on aura un changement de niveau de privilège vers le niveau 0. Chaque niveau de protection (0-3) possède une pile

distincte (voir le Chapitre 5). Ainsi, lors d'une transition du niveau 3 au niveau 0, on a une commutation de pile vers la pile de niveau 0.

Champ	Valeur
Déplacement	déplacement Default_Hand
Sélecteur	OS_Code_Sel
DmotCompt.	0, ne copie aucun paramètre
Type	E, Guichet d'interruption 386
DPL	0

Nous employons les alias du segment OS_Data pour accéder à l'IDT. Initialement, les 17 guichets d'interruption sont remplis avec la même information. Ce sont les 17 niveaux d'interruption prédéfinis par le 80386, et nous n'en définissons pas d'autres. Après avoir positionné les 17 descripteurs par défaut, nous remplaçons explicitement cette information pour les interruptions qui sont associées à des handlers spécifiques gérés par le système d'exploitation. Ainsi, le système d'interruption n'a pas intérêt à distinguer certaines interruptions et les traite par le même handler. Les handlers d'exception situés au début du OS_Code_Seg sont triviaux, et permettent de donner un exemple de déclaration de handler d'exception. Dans le second exemple de ce chapitre, nous donnerons une explication beaucoup plus élaborée du handler d'exception (Exception 7 - Coprocesseur non disponible (DNA)).

Séquence de Code 8 - Construction du catalogue des pages

Maintenant que le modèle de segmentation a été construit, nous sommes prêts à mettre en place le mécanisme de pagination du 80386. Ce dernier (voir le Chapitre 5) possède deux niveaux de tables de transformation, qui résident en mémoire : le catalogue des pages et les tables des pages (voir la Figure 5.15). Cette séquence de code va construire le catalogue des pages. La Figure 5.17 donne le format des éléments d'une table des pages.

Chaque élément du catalogue des pages (parmi les 1 024 éléments possibles) pointe sur une table des pages. Puisque chaque table des pages pos-

sède 1 024 éléments, chacun pointant sur une page de 4 096 octets, chaque élément du catalogue des pages peut gérer jusqu'à 4 Mo de mémoire. Si un élément du catalogue des pages est marqué non présent, les 4 Mo correspondants de l'espace adressable linéaire ne sont pas présents. La première partie de cette séquence marque toutes les tables non présentes (le bit 0 de chaque élément est le bit P, qui est positionné à 0). Puis, elle initialise chaque élément du catalogue des pages qui est associé à une partie valide de l'espace adressable linéaire. L'information d'initialisation est :

1 Le bit présent (=1).

2 Niveau de privilège utilisateur/superviseur.

3 Lecture/Ecriture.

4 Adresse physique de la table des pages sur laquelle cet élément pointe.

Tous les éléments seront du type lecture (privilège superviseur), sauf les catalogues de code, de données et de pile utilisateur qui sont du type écriture. L'adresse de table des pages contenue dans chacun des éléments est l'adresse physique de la table des pages.

Le deuxième élément de la table des pages est particulièrement intéressant. Cet élément du catalogue pointe sur le catalogue lui-même. Au premier abord, cela peut paraître confus, mais c'est un moyen judicieux d'associer le catalogue des pages et les tables des pages à l'espace adressable linéaire. La Figure 7.6 représente cette opération. L'élément 0000000001b du catalogue des pages pointe sur le catalogue des pages.

Une adresse linéaire de 00C00010h :

31 — 22	21 — 12	11 — 0
0000000011	0000000000	000000010000

représente un accès classique par l'élément 3 du catalogue des pages, puis par la table des pages 0 pour adresser 10h dans cette page (voir le premier exemple de la Figure 7.6).

Le deuxième exemple montre que l'adresse linéaire 00403000h :

31 22	21 12	11 0
0000000001	0000000011	000000000000

donnera un accès au premier élément (dmot) de la table des pages 3 (voir le second exemple de la Figure 7.6).

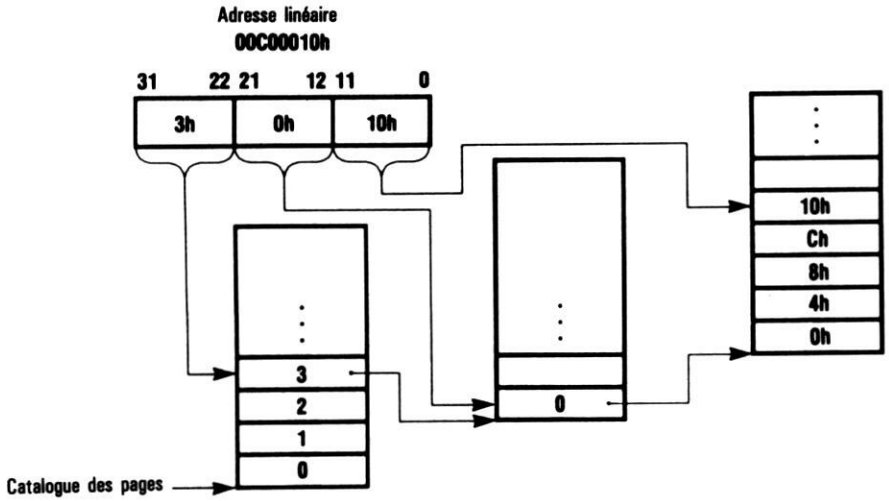
Le troisième exemple montre comment l'adresse linéaire 0040100Ch :

31 22	21 12	11 0
0000000001	0000000001	000000001100

accédera au quatrième élément (dmot) du catalogue des pages (voir le troisième exemple de la Figure 7.6).

Séquence de Code 9 - Construction des tables des pages

Après le catalogue des pages, nous devons maintenant construire les tables associées à chaque élément valide du catalogue. Comme pour le catalogue, la première chose à faire est de remplir toute la table avec des éléments non présents puis d'y insérer sélectivement des éléments valides. On a vu qu'il était important de rendre l'espace adressable linéaire identique à l'espace physique lors de la construction des tables des pages. A ce stade, nous devons conserver l'égalité (espace physique et espace linéaire) des adresses suivantes :



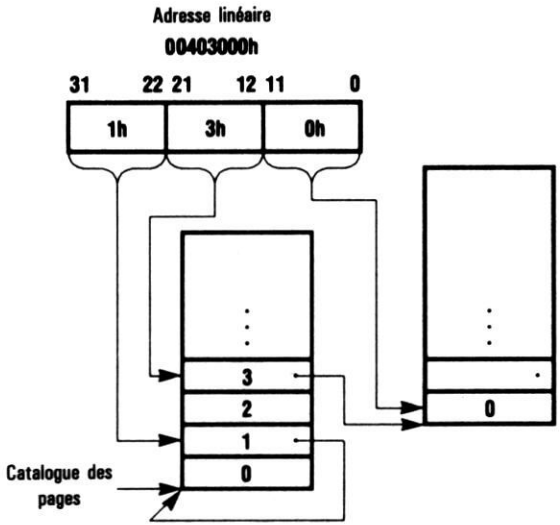


Figure 7.6 : *Catalogue des pages pointant sur lui-même.*

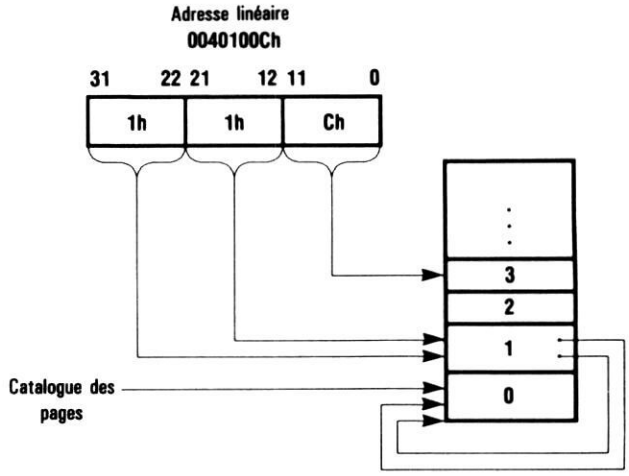


Figure 7.6 : *Catalogue des pages pointant sur lui-même (fin).*

1 Le système d'exploitation s'exécute depuis la ROM. Ainsi, les 16 pages qui commencent aux adresses physiques FFFF0000h, FFFF1000h...FFFFF000h sont placées au même endroit dans l'espace physique et dans l'espace linéaire. Sinon, on aboutirait à l'exécution d'un *saut magique* lors de la mise en place de la pagination. *Magique* signifie que l'instruction suivante serait traitée à partir d'une adresse linéaire associée à une adresse physique différente de celle qui existait avant la pagination.

ELEMENT CATALOGUE	ELEMENT TABLE	UTIL.	ECRIT.	REM
0	0	N	N	IDT,GDT,TSS
1	0	N	N	Table des pages 0
1	1	N	N	Catalogue des pages
1	3	N	N	Table des pages 3
1	4	N	N	Table des pages 4
1	3FEh	N	N	Table des pages 3FE
1	3FFh	N	N	Table des pages 3FF
3	0	N	N	Données OS
3	3FFh	N	N	Pile OS
4	0	O	N	Code utilisateur
4	100h	O	O	Données utilisateur
3FEh	3FEh	O	O	Pile utilisateur
3FFh	3F0h-3FFh	N	N	Code OS

Tableau 7.3 : *Pages allouées.*

2 Les GDT, IDT et TSS sont rangés dans la page commençant à l'adresse physique 00000000h, qui est égale à l'adresse linéaire. Sinon, nous devrions recharger ces registres et reconstruire les tables associées après avoir activé la pagination.

Reportez-vous à la Figure 7.5 (l'espace adressable physique) et comparez les éléments remplis et les pages allouées sur la figure. Le Tableau 7.3 résume les pages allouées et leur niveau d'accès.

Séquence de Code 10 - Activation de la pagination

Le catalogue des pages et les tables des pages sont prêts. L'activation de la pagination s'effectue en positionnant simplement le bit PG de CR0.

Après cette opération, il est important d'effectuer un saut qui vide à la fois la file de prétraitement et la file de décodage des instructions. Comme pour la protection, l'information contenue dans ces deux files a été traitée avant l'activation de la pagination et n'est donc plus valide.

Séquence de Code 11 - Chargement du pointeur de pile du système d'exploitation

Ce segment de code charge simplement le pointeur de pile du système d'exploitation. Chaque niveau de protection (0-3) possède une pile distincte (voir le Chapitre 5). Puisque notre exemple n'emploie que les niveaux 0 et 3, il n'a besoin que de deux piles. Remarquez l'utilisation de l'instruction LSS. Elle permet d'éviter toutes les difficultés que pourraient engendrer les chargements consécutifs de SS et de ESP. Dans notre exemple, les interruptions sont masquées et aucune exception ne peut survenir. L'emploi de LSS n'est donc pas nécessaire, mais c'est une bonne habitude à prendre.

Séquence de Code 12 - Construction du TSS

Tout changement de niveau de privilège ou toute commutation de tâches fera référence au TSS pour retrouver ou ranger de l'information. Par exemple, la première exception (comme une faute du type page) rendra nécessaire un passage du niveau de privilège 3 (le niveau utilisateur) au niveau de privilège 0 (le niveau du système d'exploitation). Il y aura donc changement de pile. Le 80386 s'attend à trouver le sélecteur et le déplacement de la pile de niveau 0 dans le TSS. Nous devons donc placer les informations suivantes dans le TSS :

1 Déplacement de la pile de niveau 0.

2 Sélecteur de la pile de niveau 0.

3 Base du catalogue des pages (CR3).

4 LDT.

5 Déplacement de la table d'autorisations E/S.

6 Bit de trappe de mise au point (mis à 0).

Nous donnons la valeur 8000h au déplacement de la table d'autorisations E/S, c'est-à-dire au-delà de la limite du TSS, pour spécifier une table vide.

Ensuite, le programme charge le TR (registre de tâche) pour que le processeur reconnaisse le TSS.

Notez aussi que le TSS dispose de place pour le mode virgule flottante. L'octet de poids fort du TSS est mis à 0 pour indiquer que cette tâche n'a pas encore utilisé le coprocesseur.

Chargement de la tâche utilisateur

Maintenant, nous pouvons charger la tâche utilisateur. Cet exemple suppose l'existence d'une procédure *chargeur* (loader) qui prend les sélecteurs des segments code et données de la tâche. Cette routine lira ensuite le code et les données de la tâche sur le disque (par exemple) puis les placera en mémoire. Etant donné que la tâche peut prendre beaucoup de place en mémoire, la première page du code et les données sont chargées en mémoire tandis que le reste de la tâche est gardé en mémoire virtuelle (sur disque donc) jusqu'à ce qu'une faute du type page entraîne son chargement. Cette routine renvoie le déplacement du début de la tâche (la première instruction) dans le registre EAX.

Lorsque nous avons construit les descripteurs locaux (séquence de Code 6), nous avons fixé arbitrairement la zone de code utilisateur à 1 Mo et laissé le reste de la mémoire (presque 4 Go de l'espace adressable linéaire) aux données utilisateur. Le chargeur devra donc réajuster la limite

de User_Code_Seg et la base de User_Data_Seg lorsque la tâche sera chargée.

Préparation de l'appel à la tâche utilisateur

Les appels à des niveaux externes sont interdits (voir le Chapitre 5). Au contraire, les retours s'effectuent vers des niveaux externes. Ainsi, nous allons placer une image de la tâche utilisateur dans la pile. Cette image est composée de :

- Sélecteur de la pile
- Déplacement de la pile
- Indicateurs
- Sélecteur du code
- Déplacement du code (donné par le chargeur)

Lorsqu'un IRET est effectué, il *retournera* au programme utilisateur au niveau 3 pour commencer l'exécution de la première tâche utilisateur du système.

Notez que les interruptions sont autorisées quand on positionne le bit IF dans l'image des indicateurs dans la pile. Ainsi, lors de l'IRET, les interruptions seront autorisées. Un STI dans le système d'exploitation aurait effectué la même opération mais prématurément.

Appel de la tâche utilisateur

Finalement, nous arrivons au fameux IRET. Il appelle la première tâche utilisateur et c'est la fin de notre exemple. L'IRET va retrouver

l'information qui vient d'être empilée et va changer l'état de la machine. Dans notre exemple, le niveau de protection, la pile, CS:EIP et les indicateurs sont modifiés.

```
; ***********************************************
; * Initialisation complète de la machine et    *
; * appel de la première tâche utilisateur.     *
; * La plupart des parties de ce programme      *
; * construisent les tables de segmentation et *
; * les tables de pagination.                   *
; *                                             *
; * L'exemple se compose de 15 étapes:        *
; * 1)  Démarrage à froid                       *
; * 2)  Divers : Inhibe les interruptions,      *
; * 3)           charge DS, annule IDTR         *
; * 4)  Passage en mode protégé 32 bits         *
; * 5)  Chargement des registres segments       *
; * 6)  Construction de LDT,   LLDT             *
; * 7)  Construction de IDT,   LIDT             *
; * 8)  Construction du catalogue des pages     *
; * 9)  Construction des tables des pages       *
; * 10) Activation de la pagination             *
; * 11) Chargement de la pile de l'OS           *
; * 12) Construction de TSS et LTR              *
; * 13) Chargement de la tâche utilisateur      *
; * 14) Préparation à l'appel de la tâche util.*
; * 15) Appel de la tâche utilisateur !         *
; ***********************************************

                NAME   Initial
; Constantes Diverses
IDT_Entries    EQU      17
GDT_Entries    EQU      5        ; NUL, OS Data, LDT
                                 ; OS Code, TSS
LDT_Entries    EQU      2        ; Code Util. Data Util.
```

Listing 7.1 : *Code d'initialisation.*

```
Page_Size      EQU      4096
Page_Entries   EQU      1024
Not_Present    EQU      0
PE             EQU      1
PG             EQU      80000000h
Int_Flag       EQU      0200h

; Sélecteurs de la GDT
Null           EQU      0h
OS_Code_Sel    EQU      8h      ; 1,GDT,RPL=00
OS_Data_Sel    EQU      10h     ; 2,GDT,RPL=00
LDT1_Sel       EQU      18h     ; 3,GDT,RPL=00
TSS1_Sel       EQU      20h     ; 4,GDT,RPL=00

; Sélecteurs de la LDT
User_Code_Sel EQU       7h      ; 0,LDT,RPL=11
User_Data_Sel EQU       OFh     ; 1,LDT,RPL=11

; Adresses Physiques :
; Initialisation des tables des pages
P_OS_GData     EQU      00000h
P_Page_Dir     EQU      01000h
P_Page_Tab0    EQU      02000h
P_Page_Tab3    EQU      03000h
P_Page_Tab4    EQU      04000h
P_Page_Tab3FE EQU       05000h
P_Page_Tab3FF EQU       06000h
P_OS_PData     EQU      07000h
P_OS_Stack     EQU      08000h
P_User_Code    EQU      09000h
P_User_Data    EQU      0A000h
P_User_Stack   EQU      0B000h

TSS_Size       EQU      104     ; Taille minimum de TSS
FP_Save        EQU      112     ; 108 + 4 (mot supplémentaire
```

Listing 7.1 : *Code d'initialisation (suite).*

```
                                ; pour indicateur (FP))

; ******************
; ******************
; ** Segment OS Data
; ******************
; ******************
; Contient les tables de pagination et de segmentation.
; Place ce segment à l'adresse linéaire 0.
OS_Data_Seg   SEGMENT RW USE32   ; Placé à 00000000h

; IDT_Table au Déplacement 0, GDT immédiatement après
; TSS immédiatement après GDT, LDT immédiatement après TSS
IDT_Table     db   IDT_Entries*8 DUP (?)
GDT_Table     db   GDT_Entries$8 DUP (?)
TSS1_Table    db   (TSS_Size+FP_Save) DUP (?)
LDT1_Table    db   LDT_Entries$8 DUP (?)

pdescr        db   6 dup(?) ; Construit un pseudo-descripteur
                            ;pour IDTR et GDTR

; *****
; ** Zone du catalogue des pages et des tables des pages
; ** dans l'espace physique, utilisée pour initialiser les
; ** tables des pages avant activation de la pagination.
; *****
            ORG 00001000h    ; Adresse Physique
Page_Dir      dd   Page_Entries DUP(?)
Page_Tab0     dd   Page_Entries DUP(?)
                   ; Données de l'OS (IDT,GDT,TSS)
Page_Tab3     dd   Page_Entries DUP(?)
                   ; Données privées de l'OS
                   ; (pile)
Page_Tab4     dd   Page_Entries DUP(?)
                   ; Code et Données Utilisateur
```

Listing 7.1 : *Code d'initialisation (suite).*

```
Page_Tab3FE    dd    Page_Entries DUP(?)
                     ; Pile Utilisateur
Page_Tab3FF    dd    Page_Entries DUP(?)
                     ; Code OS

; *****
; ** Zone du catalogue des pages et des tables des pages
; ** dans l'espace linéaire (-> 4 Mégaoctets)
; *****
             ORG 00400000h
L_Page_Tab0    dd    Page_Entries DUP(?)
                     ; Données de l'OS (IDT,GDT,TSS)
L_Page_Dir     dd    Page_Entries DUP(?)
                     ; Catalogue des pages
             ORG 00403000h
L_Page_Tab3    dd    Page_Entries DUP(?)
                     ; Données privées de l'OS
                     ; (pile)
L_Page_Tab4    dd    Page_Entries DUP(?)
                     ; Code et Données Utilisateur
             ORG 007FE000h
L_Page_Tab3FE dd     Page_Entries DUP(?)
                     ; Pile Utilisateur
L_Page_Tab3FF dd     Page_Entries DUP(?)
                     ; Code OS

; *****
; * Données OS. Espace linéaire de 4 Mo.
; * N'est pas utilisé dans l'exemple.
; *****
ORG       0800000h

; *****
; ** Données OS.
; ** Espace linéaire de 4 Mo pour les données
```

Listing 7.1 : *Code d'initialisation (suite).*

```
; ** et la pile privées
; *****
ORG       0C00000h       ; Adresse Linéaire
Unshared_OS_Data   dd   ?
; Zone non partagée
temp          db   10 dup (?)
fptss         dw   ?

; *****
; ** Pile OS.
; *****
ORG       OFFFFFCh       ; Sommet de la pile OS
Top_OS_Stack  dd   ?
OS_Data_Seg   ENDS

; ****************************
; ****************************
; ** Segment Code Utilisateur
; ****************************
; ****************************
; Place ce segment à l'adresse linéaire 1000000h.
User_Code_Seg   SEGMENT ER USE32 ;
   ; Début du segment code utilisateur dans
   ; l'espace linéaire. Définit une région de 1 Mégaoctet.
User_Code_Seg   ENDS

; ********************************
; ********************************
; ** Segment de Données Utilisateur
; ********************************
; ********************************
; Place ce segment à l'adresse linéaire 1100000h.
User_Data_Seg   SEGMENT RW USE32 ; à 1100000h
   ; Début du segment données utilisateur dans
   ; l'espace linéaire. Le reste de la mémoire
```

Listing 7.1 : *Code d'initialisation (suite).*

```
   ; est laissé aux données utilisateur
   ORG   OFEEEFFFCh
User_Data_Seg   ENDS

; ******************************
; ******************************
; ** Segment Code 32 bits de l'OS
; ******************************
; ******************************
; Code OS 32 bits.
; Place ce segment à l'adresse linéaire OFFFFOOOOh.
OS_Code_Seg   SEGMENT ER USE32 ; à OFFFF0000h
      Extrn   loader:NEAR

Default_Hand:
      ; <Code Handler par défaut>
      IRET
DNA_Hand:
      ; <Code handler du type DNA : exemple 2>
      IRET
Page_Fault_Hand:
      ; <Code handler du type faute de page>
      IRET
Debug_Hand:
      ; <Code handler du type mise au point>
      IRET
Start32:

; *****
; ** [5] Rend le segment OS_Data adressable en chargeant
; ** les registres segments DS et ES.
; *****
   MOV   AX, OS_Data_Sel
   MOV   DS, AX
   MOV   ES, AX
```

Listing 7.1 : *Code d'initialisation (suite).*

```
   ASSUME DS: OS_Data_Seg

; *****
; ** [6] Construction de LDT,  LLDT
; *****
   MOV   EAX, Offset LDT1_Table
   ; élément 0 --> Descripteur du Code Utilisateur
   ; Base=00100000h, Limite=000FFh,G=1,D=1,Type=A,DPL=3
   MOV   word ptr [EAX],000FFh   ; Limite[15..0]
   MOV   word ptr [EAX+2],0000h  ; Base[15..0]
   MOV   byte ptr [EAX+4],000h   ; Base[23..16]
   MOV   byte ptr [EAX+5],0FAh   ; Type = descripteur
                                 ; de données,DPL=11
   MOV   byte ptr [EAX+6],0C0h   ; G,D,Limite[19..16]
   MOV   byte ptr [EAX+7],01h    ; Base[31..24]
   ADD   EAX,8
   ; élément 1 -> Descripteur des Données Utilisateur
   ; Base=00110000h, Limite=FEEEFh,G=1,D=1,Type=A,DPL=3
   MOV   word ptr [EAX],0EEEFh   ; Limite[15..0]
   MOV   word ptr [EAX+2],0000h  ; Base[15..0]
   MOV   byte ptr [EAX+4],010h   ; Base[23..16]
   MOV   byte ptr [EAX+5],0F2h   ; Type = Donnée
                                 ; descripteur, DPL=11
   MOV   byte ptr [EAX+6],0CFh   ; G,D,Limite[19..16]
   MOV   byte ptr [EAX+7],01h    ; Base[31..24]

   MOV   AX, LDT1_Sel
   LLDT  AX

; *****
; ** [7] Construction de IDT
; *****
 ; Remplit les 16 premiers éléments de la IDT.
 ; Tout d'abord le handler
 ; par défaut, puis continue sur les autres
```

Listing 7.1 : *Code d'initialisation (suite).*

```
  MOV   EAX, offset IDT_Table
  MOV   EBX, offset Default_Hand
  MOV   word ptr [EAX], BX        ; Déplacement[15..0]
  MOV   word ptr [EAX+2], OS_Code_Sel
  MOV   byte ptr [EAX+4], 00h     ; WordCnt=0
  MOV   byte ptr [EAX+5], 8Eh     ; Type=E, DPL=00
  SHR   EBX, 16
  MOV   word ptr [EAX+6], BX      ; Déplacement[31..16]
; Copie dans le reste
  MOV   ESI, offset IDT_Table
  MOV   EDI, (offset IDT_Table + 8)
  MOV   ECX, (IDT_Entries-1)*2
  REP   MOVSD
; Revient et remplit les autres pointeurs
; élément 1: debugger
  MOV   EAX, (offset IDT_Table + (1 * 8))
  MOV   EBX, offset Debug_Hand
  MOV   word ptr [EAX], BX        ;Déplacement[15..0]
  SHR   EBX, 16
  MOV   word ptr [EAX+6], BX      ;Déplacement[30..16]

; élément 7: DNA
  MOV   EAX, (offset IDT_Table + ( 7 * 8))
  MOV   EBX, offset DNA_Hand
  MOV   word ptr [EAX], BX        ;Déplacement[15..0]
  SHR   EBX, 16
  MOV   word ptr [EAX+6], BX      ;Déplacement[30..16]

; élément 14: Faute du type page
  MOV   EAX, (offset IDT_Table + (14 * 8))
  MOV   EBX, offset Page_Fault_Hand
  MOV   word ptr [EAX], BX        ;Déplacement[15..0]
  SHR   EBX, 16
  MOV   word ptr [EAX+6], BX      ;Déplacement[30..16]
; charge IDTR
```

Listing 7.1 : *Code d'initialisation (suite).*

```
   MOV    word ptr pdescr, (IDT_Entries * 8 - 1)
   MOV    dword ptr pdescr[2], offset IDT_Table
   LIDT   pword ptr pdescr

; *****
; ** [8] Construction du catalogue des pages
; *****
   MOV    EAX,Not_Present
   MOV    ECX, Page_Entries
   CLD
   MOV    EDI, offset Page_Dir
   REP    STOSD        ; marque non présent tout le catalogue

 ; Elément 0: Table des données partagées par l'OS.
 ; pas d'accès utilisateur
 ; Placé à 0 si bien que linéaire = physique et GDT,
 ; IDT et TSS chargés
 ; encore valides après activation de la pagination
   MOV    Page_dir[0], P_Page_Tab0 OR 001h
 ; Elément 1: Pointeur sur le Catalogue.
 ; Pas d'accès utilisateur.
 ; PageDir et PageTables dans l'espace linéaire
   MOV    Page_Dir[4], P_Page_Dir OR 001h

 ; Elément 3: Table des données privées de l'OS.
 ; Pas d'accès utilisateur
   MOV    Page_Dir[12], P_Page_Tab3 OR 001h

 ; Elément 4: Code et Données Utilisateur. Accès utilisateur
 ; du type écriture/lecture
   MOV    Page_Dir[16], P_Page_Tab4 OR 007h

 ; Elément 3FEh: Pile Utilisateur, à l'adresse
 ; la plus haute de l'espace
 ; des données utilisateur. Accès utilisateur en écriture
```

Listing 7.1 : *Code d'initialisation (suite).*

```
 ; et lecture
   MOV    Page_Dir[4088], P_Page_Tab3FE OR 007h

 ; Elément 3FFh: Table des zones Code partagées de l'OS.
 ; Pas d'accès utilisateur
 ; Il est important que cet élément soit placé
 ; à l'adresse la plus haute pour être associé
 ; à la même adresse lors de l'exécution
   MOV    Page_Dir[4092], P_Page_Tab3FF OR 001h

; *****
; ** [9] Construction des Tables : Table des Pages
; *****
   ; Table des Pages 0 : Données générales de l'OS
   ; On doit penser à effacer la table entière
   MOV    EAX, Not_Present
   MOV    ECX, Page_Entries
   CLD
   MOV    EDI, offset Page_Tab0
   REP    STOSD
   ; Maintenant alloue les pages pour l'IDT, GDT, TSS...
   MOV    Page_Tab0[0h], P_OS_GData OR 001h
   ; Table des Pages 3: données privées de l'OS
   ; On doit penser à effacer la table entière
   MOV    EAX, Not_Present
   MOV    ECX, Page_Entries
   CLD
   MOV    EDI, offset Page_tab3
   REP    STOSD
   ; Maintenant alloue les pages pour les données
   ; privées et la pile
   MOV    Page_Tab3[0000h], (P_OS_PData OR 001h)
   MOV    Page_Tab3[OFFCh], (P_OS_Stack OR 001h)

   ; Table des Pages 4: Code, Données et Pile Utilisateur
```

Listing 7.1 : *Code d'initialisation (suite).*

```
  ; On doit penser à effacer la table entière
  MOV   EAX, Not_present
  MOV   ECX, Page_entries
  CLD
  MOV   EDI, offset Page_tab4
  REP   STOSD
  ; Maintenant alloue les pages du code
  ; et données utilisateur
  MOV   Page_tab4[0000], (P_User_Code OR 005h)
  MOV   Page_Tab4[1024], (P_User_Data OR 007h)

  ; Table des Pages 3FE: Pile Utilisateur
  ; On doit penser à effacer la table entière
  MOV   EAX, Not_Present
  MOV   ECX, Page_Entries
  CLD
  MOV   EDI, offset Page_Tab3FE
  REP   STOSD
  ; Maintenant alloue les pages pour pile utilisateur
  MOV   Page_tab3FE[4092], (P_UserStack OR 007h)

  ; Table des Pages 3FFh: Code OS.
  ; Accès utilisateur en lecture
  MOV   EAX, 0FFFFF005h      ; PTE le plus haut
  MOV   ECX, Page_Size
  LoopTop:
     SUB   ECX, 4
     MOV   Page_Tab3FF[ECX], EAX
     SUB   EAX, 1000h
     CMP   ECX, 4032
     JNZ   LoopTop
; *****
; ** [10] Activation de la Pagination
; *****
   MOV   EAX, P_Page_Dir
```

Listing 7.1 : *Code d'initialisation (suite).*

```
   MOV    CR3, EAX
   MOV    EAX, CR0
   OR     EAX, PG
   MOV    CR0, EAX
   JMP    pflush
pflush:
; *****
; ** [11] Charge le pointeur de pile OS
; *****
   MOV    dword ptr temp[0],offset Top_OS_stack
   MOV    word ptr temp[4],OS_Data_Sel
   LSS    ESP,pword ptr temp

; *****
; ** [12] Construction du TSS.
; ** Charge uniquement les parties référencées par l'OS.
; ** Les autres parties sont écrites lors de la première
; ** transition à un niveau interne.
; ** Charge aussi le registre de tâche
; *****
   ; Déplacement de la pile de niveau 0
   MOV    dword ptr TSS1_Table[4], offset Top_OS_stack
   ; Sélecteur de la pile de niveau 0
   MOV    dword ptr TSS1_Table[8],OS_Data_Sel
   ; CR3
   MOV    EAX, CR3
   MOV    dword ptr TSS1_Table[1Ch],EAX
   ; LDT
   MOV    dword ptr TSS1_Table[60h],0
   ; Déplacement de la table d'autorisations E/S
   MOV    word ptr TSS1_Table[66h], 8000h
   ; Bit de trappe de mise au point
   MOV    word ptr TSS1_Table[64h], 0
   MOV    AX, TSS1_Sel
   LTR    AX
```

Listing 7.1 : *Code d'initialisation (suite).*

```
   ; Initialise les octets d'état
   ; (en virgule flottante)(exemple 2)
   ; Ceux-ci sont décrits précisément dans cet exemple.
   MOV   byte ptr TSS1_Table[TSS_Size + FP_Save - 1], 0
   MOV   fptss, 0

; *****
; ** [13] Chargement de la Tâche Utilisateur
; *****
   MOV   EAX, User_Code_Sel
           ; Sélecteur du code utilisateur
   MOV   ECX, User_Data_Sel
           ; Sélecteur des données utilisateur
   ; Le chargeur traite la première tâche
   ; utilisateur en la chargeant
   ; à partir du disque dans les emplacements
   ; mémoire déterminés par les sélecteurs.
   ; Puis, il retourne dans EAX le déplacement du
   ; début de la tâche utilisateur.
   CALL  loader

; *****
; ** [14] Prépare l'appel de la routine utilisateur.
; ** Empile SS:ESP, les indicateurs, CS:EIP, et IRET.
; *****
   MOV   ECX, User_Data_Sel; Pile pour la routine utilisateur
   PUSH  ECX
                           ; Pointeur de pile utilisateur
   MOV   ECX, offset Top_User_Stack
   PUSH  ECX

   PUSHF             ; Positionne IF sur la pile.
   OR    SS: dword ptr [ESP],Int_Flag

   MOV   ECX, User_Code_Sel
```

Listing 7.1 : *Code d'initialisation (suite).*

```
; Segment Code de la routine utilisateur
   PUSH  ECX
   PUSH  EAX
; Déplacement du début de la tâche utilisateur

; *****
; ** [15] Appel de la tâche utilisateur
; *****

   ; IRET magique, commutation sur la routine utilisateur
   IRET

OS_Code_Seg ENDS

; Code d'initialisation 16 bits
; Origine du segment à 0FFFF0000h
; ORG au déplacement 0F000h pour placer
; le code de lancement aux 4K
; situés au sommet de la mémoire.
BootRom16   SEGMENT EO   USE16 ; à 0FFFF0000h
      ORG   0F000h         ; Code OS 16 bits aux 4K supérieurs
Start16:
   ; *****
   ; ** [2] Divers: CLI, DS, IDT nul
   ; *****
   CLI            ; Inhibe les interruptions
   MOV   AX, 0    ; Segment initial des données en mode réel
   MOV   DS, AX
   ASSUME DS:OS_Data_Seg ; dit à l'assembleur que DS adresse
                         ; les données de l'OS

   ; shutdown pour toutes les interruptions
   MOV   word ptr pdsecr[0],0
   MOV   dword ptr pdescr[2],0
   LIDT  pword ptr pdescr
```

Listing 7.1 : *Code d'initialisation (suite).*

```
; *****
; ** [3] GDT
; *****
MOV   EAX, offset GDT_Table
; élément 0 --> nul
MOV   dword ptr [EAX],0
MOV   dword ptr [EAX+4],0
ADD EAX,8
; élément 1 --> Descripteur du Code de l'OS
; Base=FFFF0000h, Limite=0000Eh, G=1, D=1, Type=A, DPL=0
MOV   word ptr [EAX],0000Eh      ; Limite[15..0]
MOV   word ptr [EAX+2],0000h     ; Base[15..0]
MOV   byte ptr [EAX+4],0FFh      ; Base[23..16]
MOV   byte ptr [EAX+5],09Ah      ; Type = descripteur de
                                 ; code, DPL=0
MOV   byte ptr [EAX+6],0C0h      ; G,D,Limite[19..16]
MOV   byte ptr [EAX+7],0FFh      ; Base[31..24]
ADD   EAX,8

; élément 2 --> Descripteur des Données de l'OS
; Base=00000000h, Limite=0FFFFFh, G=1, D=1, Type=9, DPL=0
MOV   word ptr [EAX],0FFFFh  ; Limite[15..0]
MOV   word ptr [EAX+2],0000h ; Base[15..0]
MOV   byte ptr [EAX+4],00h   ; Base[23..16]
MOV   byte ptr [EAX+5],092h  ; Type = descripteur de
                             ; données, DPL=0
MOV   byte ptr [EAX+6],0CFh  ; G,D,Limite[19..16]
MOV   byte ptr [EAX+7],00h   ; Base[31..24]
ADD   EAX,8

; élément 3 --> LDT
; Base=(offset LDT1_Table), Limite=(LDT_Entries * 8),
; G=0, Type=LDT, DPL=00
MOV   word ptr [EAX], (LDT_Entries * 8 - 1)
; Limite[15..0]
```

Listing 7.1 : *Code d'initialisation (suite).*

```
MOV   EBX, offset LDT1_Table
MOV   word ptr [EAX+2], BX    ; Base[15..0]
SHR   EBX, 16
MOV   byte ptr [EAX+4], BL    ; Base[23..16]
MOV   byte ptr [EAX+5],082h   ; Type=LDT
MOV   byte ptr [EAX+6],000h   ; Limite[19..16]
MOV   byte ptr [EAX+7], BH    ; Base[31..24]
ADD   EAX,8

; élément 4 --> TSS pour cette tâche
; Base=(offset TSS1_Table), Limite=(TSS_Size + FP_Save),
; G=0, Type=TQQ 386 Disponible, DPL=00
MOV   word ptr [EAX], (TSS_Size + FP_Save - 1)
; Limite[15..0]
MOV   EBX, offset TSS1_Table
MOV   word ptr [EAX+2], BX    ;Base[15..0]
SHR   EBX, 16
MOV   byte ptr [EAX+4], BL    ; Base[23..16]
MOV   byte ptr [EAX+5],089h   ;Type=TSS 386 Disponible
MOV   byte ptr [EAX+6],000h   ; Limite[19..16]
MOV   byte ptr [EAX+7], BH    ; Base[31..24]

; Pseudo-descripteur de GDT (limite et base)
; et le charge dans GDTR
MOV   word ptr pdescr[0], (GDT_Entries*8 - 1)
MOV   dword ptr pdescr[2],offset GDT_Table
LGDT  pword ptr pdescr

; *****
; ** [4] Entre en Mode Protégé
; *****
SMSW  AX
OR    AX,PE
LMSW  AX          ; valide le bit de protection
JMP   Flush
```

Listing 7.1 : *Code d'initialisation (suite).*

```
   Flush:
   JMP    far ptr Start32       ; Saut via GDT vers
                                ; Code OS 32 bits
   ; *****
   ; ** [1] Code du démarrage à froid !
   ; *****
   ORG    0FFF0h
   JMP    Start16
BootRom16 ENDS

END
```

Listing 7.1 : *Code d'initialisation (fin).*

HANDLER D'EXCEPTION DU COPROCESSEUR

Le second exemple présente un handler d'exception du type *coprocesseur non disponible* (DNA). Le Listing 7.2 donne le programme de ce handler.

Vue générale de l'exemple 2

Dans cet exemple, nous allons détailler un handler de faute et tout spécialement le handler associé à l'exception du type coprocesseur non disponible (vecteur n° 7). Les conditions suivantes peuvent générer cette exception (voir le Chapitre 6).

1 Une instruction à virgule flottante est exécutée et le bit EM du registre de contrôle CR0 est à 1. EM indique qu'on doit émuler le coprocesseur mathématique.

2 Une instruction à virgule flottante est exécutée et le bit TS du registre de contrôle CR0 est à 1. TS indique que le système est en mode commutation de tâches rapide.

3 Une instruction WAIT est exécutée et les bits TS et MP de CR0 sont à 1.

Dans cet exemple, nous allons parler du mode commutation de tâches rapide. Si le système doit émuler le coprocesseur (voir le Chapitre 1), le programme effectue un simple appel à la routine DNA_Emulate, qui doit simuler le fonctionnement du coprocesseur.

Le mode commutation de tâches rapide optimise le fonctionnement du système puisque le contexte du coprocesseur n'est sauvegardé que lorsqu'une tâche y accède. Si la plupart des tâches n'emploient pas le coprocesseur, ce mode rend bien service, puisque les sauvegardes et récupérations du contexte du coprocesseur sont gourmandes en temps CPU.

Lorsqu'une exception DNA survient, le handler associé sera appelé. Nous avons écrit ce handler de façon qu'on puisse l'inclure directement dans notre premier exemple. En fait, vous pouvez insérer le handler DNA exactement à l'emplacement marqué :

```
<Code Handler du type DNA - Exemple 2>
```

Nous avons défini plusieurs éléments de l'exemple 1 uniquement pour le handler DNA. Par exemple, nous avons alloué suffisamment d'espace mémoire au TSS pour lui permettre de contenir le contexte du coprocesseur.

Lors d'une exception, si une tâche avait employé le coprocesseur, le système sauvegarde le contexte du coprocesseur dans le TSS de la tâche en question. Le contexte du coprocesseur pour la tâche courante (si cette tâche a déjà employé le coprocesseur) est ensuite chargé dans le coproces-

seur. Le système d'exploitation conserve une variable contenant le sélecteur du TSS de la dernière tâche utilisant le coprocesseur. Elle est mise à jour chaque fois que le système charge un nouveau contexte dans le coprocesseur. Nous parlerons du problème posé par les interruptions ultérieurement.

Handler d'exception

Avant chaque opération, les interruptions sont autorisées. Puisque charger et sauvegarder le contexte du coprocesseur peut prendre un bon moment, nous n'inhiberons les interruptions que lorsque cela sera absolument nécessaire. On doit autoriser les interruptions puisque tous les handlers de l'exemple 1 sont des guichets d'interruption qui inhibent les interruptions.

Séquence de Code 1 - Emulation

Le programme appelle l'émulateur si l'exception a été causée par le bit EM de CR0 à 1. Notre exemple ne comprend pas l'émulateur. Après l'émulation, on retourne simplement au programme principal.

Séquence de Code 2 - Base de la tâche courante

La variable *fptss* contient le sélecteur pour le TSS de la tâche dont le contexte est le contexte courant du coprocesseur. Ce dernier est sauvegardé dans la partie supérieure du TSS. Ainsi, nous devons déterminer le sélecteur et la base du TSS de la tâche courante, ce que nous faisons habilement à partir du TR. Le TR donne le sélecteur du TSS courant, accédant à la GDT et y extirpant la base à partir du descripteur. Le sélecteur est gardé dans SI et la base dans ECX pour toute utilisation future.

Séquence de Code 3 - Sauvegarde de l'ancien contexte

La séquence d'initialisation met à zéro la variable fptss, afin d'indiquer qu'aucune tâche n'a encore employé le coprocesseur. Si c'est le cas, on n'a pas besoin de sauvegarder le contexte du coprocesseur et il sera initialisé par l'instruction FNINIT. Si fptss n'est pas nul, le contexte du coprocesseur est sauvegardé dans le TSS de l'ancienne tâche, à l'endroit déterminé par le sélecteur de fptss qui donne la base du TSS. Si la dernière tâche qui a utilisé le coprocesseur est la tâche courante, rien ne se passe et la routine rend la main.

Séquence de Code 4 - Chargement du nouveau contexte

Pendant que le nouveau contexte est chargé dans le coprocesseur, il faut inhiber les interruptions, puisque la machine n'est pas alors dans un état valide. Considérons le cas où le contexte du coprocesseur a été sauvegardé dans le TSS de la tâche précédente, que le contexte de la nouvelle tâche est chargé dans le coprocesseur mais que fptss n'a pas été mis à jour. Dans ce cas, fptss et le contexte du coprocesseur sont contradictoires. Si une interruption survient, tout appel à ce handler peut modifier le contexte du coprocesseur de l'ancienne tâche.

Il est ennuyeux d'inhiber les interruptions lors du chargement d'un nouveau contexte de coprocesseur : la rapidité du traitement des interruptions s'en ressent, puisque FRSTOR est une instruction très longue. Vous pouvez éviter d'inhiber les interruptions lors de l'emploi de cette instruction, mais vous avez alors besoin de deux sémaphores, et l'écriture du handler se complique beaucoup. Nous avons préféré laisser cet exercice au lecteur habile et intéressé.

L'octet supérieur de la partie contexte du TSS indique si le contexte a été sauvegardé dans ce TSS. La partie du système d'exploitation qui initialise les tâches doit annuler cet octet (notre exemple le fait). Si cet octet est nul, le programme n'a pas besoin de récupérer le contexte du coprocesseur pour la tâche courante puisque cette dernière n'a jamais utilisé le co-

processeur. Dans ce cas, le coprocesseur est simplement initialisé par une instruction FNINIT. Si l'octet est à 1, le programme doit charger le contexte associé à cette tâche dans le coprocesseur. Finalement, la variable fptss est mise à jour et le fameux octet est mis à 1 pour indiquer le nouveau contexte du coprocesseur et son emploi par cette tâche.

```
; ********************************************
; * Exemple 2 - Handler DNA                  *
; *                                          *
; * Plusieurs caractéristiques :             *
; * 1.    fptss est un sélecteur du TSS de la *
; *       tâche la plus récente qui a utilisé *
; *       le coprocesseur. Sinon, il est nul. *
; * 2.    Le TSS comporte une zone de         *
; *       sauvegarde (virgule flottante)      *
; * 3.    Cette zone de sauvegarde du TSS est *
; *       constituée de 108 octets (classique) *
; *       + 4 octets. L'octet supérieur est un *
; *       indicateur d'utilisation du         *
; *       coprocesseur.                       *
; ********************************************

DNA_Hand:
      STI
   ; *****
   ; * [1] Détermine si une faute est causée par l'Emulation
   ; *****
      ;   CR0          3
      ;                1 ...4 3 2 1 0
      ;                P    E T E M P
      ;                G    T S M P E
      MOV   EAX, CR0
      BT    EAX, 2
      JNC   DNA_Task_EX
      CALL  DNA_Emulate
```

Listing 7.2 : *Handler d'exception du type coprocesseur non disponible.*

```
   IRET

; *****
; * [2] Détermine la base du TSS de la tâche courante
; *****
DNA_Task_Ex:
   XOR   ESI,ESI
   STR   SI       ; ESI contient le sélecteur du TSS de
                  ; la tâche courante
   MOV   ECX,[ESI + offset GDT_Table + 0]
   MOV   EDX,[ESI + offset GDT_Table + 4]
   SHRD  ECX,EDX,16
   AND   ECX,000FFFFFFh
   AND   EDX,0FF000000h
   OR    ECX,EDX    ;ECX contient la base de la tâche courante
; *****
; * [3] Faute causée par la commutation de tâches.
; * Si l'état de l'ancienne tâche doit être sauvegardé, le fait
; *****
   XOR   EDI,EDI
   MOV   DI, fptss
   CMP   DI,0
   JE    DNA_Restor
   CMP   DI,SI
   ;Si le dernier utilisateur a été la tâche courante
   ; on n'a pas besoin de sauvegarder et récupérer
   JNE   DNA_Save
   IRET
DNA_Save:
   ; détermine la base du TSS de la dernière tâche
   MOV   EAX,[EDI + offset GDT_Table + 0]
   MOV   EDX,[EDI + offset GDT_Table + 4]
   SHRD  EAX,EDX,16
   AND   EAX,000FFFFFFh
   AND   EDX,0FF000000h
```

Listing 7.2 : *Handler d'exception du type coprocesseur non disponible (suite).*

```
   OR    EAX,EDX
   ; EAX contient la base de la dernière tâche

   FNSAVE [EAX + TSS_Size]
   ; Sauvegarde l'état du coprocesseur
   ; pour l'ancienne tâche

; *****
; * [4] Positionne maintenant fptss sur cette tâche.
; * Si nécessaire charge l'état précédent du coprocesseur
; * pour cette tâche. On doit inhiber alors les
; * interruptions.
; *****
DNA_Restor:
   CLI
   ; Cet octet indique que cette tâche est la précédente
   CMP   byte ptr [ECX + (TSS_Size + FP_Save - 1)],1
   JNE   DNA_Init
   FRSTOR [ECX + TSS_Size]
   JMP   DNA_Done
DNA_Init:
   FNINIT
DNA_Done:
   MOV   fptss, SI
   ; Range ce sélecteur du TSS de cette tâche dans
   ; fptss, positionne l'octet indiquant que le
   ; coprocesseur a été utilisé par cette tâche,
   ; efface TS, et positionne IF.
   MOV   byte ptr [ECX + (TSS_SIZE + FP_Save - 1)],1
   CLTS
   STI
   IRET
```

Listing 7.2 : *Handler d'exception du type coprocesseur non disponible (fin).*

8

Aide à la mise au point

Comme chacun sait, les programmes doivent être mis au point (ou *déboggués* pour les franglais !). Jusqu'à présent, cela relevait de l'art plutôt que de la science. Pour enlever un peu de magie à cette opération, des logiciels de mise au point ont été développés pour assister les programmeurs dans la localisation des erreurs (nous savons tous qu'il y en a toujours plusieurs !). Ces logiciels possèdent certaines caractéristiques bien agréables comme le mode pas à pas, la pose de point d'arrêt à certains endroits du programme, ou l'arrêt du programme lorsqu'une variable prend une valeur prédéfinie. La part que prennent ces logiciels dans le développement d'un programme devient de plus en plus importante.

Dans ce chapitre, nous n'allons pas présenter un programme de mise au point (debugger) spécifique au 80386, mais nous allons plutôt donner les caractéristiques de base, propres au 80386, qui facilitent la conception d'un outil d'aide à la mise au point très performant.

L'infrastructure du 80386 inclut une instruction de trappe sur un octet (INT 3), une trappe sur une commutation de tâches, le mode pas à pas, et la détection de quatre conditions simultanées de point d'arrêt (instruction, écriture de données, et écriture ou lecture de données). Chacune de ces possibilités va être détaillée dans la suite de ce chapitre. Les points d'arrêt du 80386 représentent une extension considérable des possibilités de mise au point qu'on trouvait chez les processeurs précédents de la famille 86.

TERMINOLOGIE

Avant de commencer notre description, nous devons clarifier la terminologie que nous allons employer.

Dans le Chapitre 6, nous avons décrit les exceptions du 80386. Dans ce chapitre, nous allons détailler *l'exception du type mise au point* (debug exception), qui survient lors de l'exécution d'une instruction. La routine qui va traiter une interruption ou une exception du type mise au point est un *handler de mise au point* (debug handler).

On peut considérer une exception du type mise au point comme une faute ou comme une trappe selon le moment où l'exception est survenue. Une *faute du type mise au point* (debug fault) est générée immédiatement avant l'exécution de l'instruction en cause. Ainsi, le handler d'exception débute avec le CS:EIP sauvegardé qui pointe sur cette instruction. Une *trappe du type mise au point* (debug trap) est générée immédiatement après l'exécution de l'instruction en cause. Ainsi, le handler d'exception débute avec le CS:EIP sauvegardé qui pointe sur l'instruction suivant l'instruction en cause. Les fautes surviennent "avant", et les trappes "après" l'instruction ayant causé l'exception. Résumons les différents types de fautes et de trappes :

1 Trappes

- Commutation de tâches
- Pas à pas
- Points d'arrêt sur donnée

2 Fautes

- Protection des registres de mise au point
- Points d'arrêt sur instructions

POINTS D'ARRET

Un point d'arrêt permet au programmeur d'associer une condition spécifique à une adresse linéaire particulière. Si elle est vérifiée, cette condition entraînera un saut au handler d'exception. Le 80386 dispose de quatre conditions simultanées de points d'arrêt. Le programmeur peut ainsi positionner quatre adresses mémoire dans un programme. Chacun de ces quatre points d'arrêt peut être de trois types différents :

- Exécution de l'instruction
- Ecriture de données
- Ecriture ou lecture de données (mais pas exécution)

Les registres de mise au point

Pour gérer ces quatre points d'arrêt, huit nouveaux registres sont ajoutés au 80386. On ne peut y accéder qu'au niveau de privilège 0. Toute tentative d'accès à un autre niveau de privilège générera une exception du type CodeOp non valide. De plus, on peut protéger l'accès de ces registres même au niveau 0 à l'aide des bits BD et GD de DR6 et de DR7. Les instructions de transfert de registres peuvent accéder à ces registres de mise au point,

```
MOV   reg,DRi
MOV   DRi,reg
```

où DRi représente DR0, DR1, DR2, DR3, DR6 ou DR7. Dans le chapitre 3, nous détaillons l'emploi de ces instructions.

La Figure 8.1 représente les huit registres de mise au point. Détaillons ces registres :

DR0-DR3 Les registres DR0 à DR3 contiennent l'adresse linéaire associée à chacune des quatre conditions de point d'arrêt. DR7 donne des renseignements supplémentaires sur ces points d'arrêt. Puisque c'est l'adresse linéaire qui est employée, le système de points d'arrêt fonctionne correctement, qu'il y ait pagination ou non.

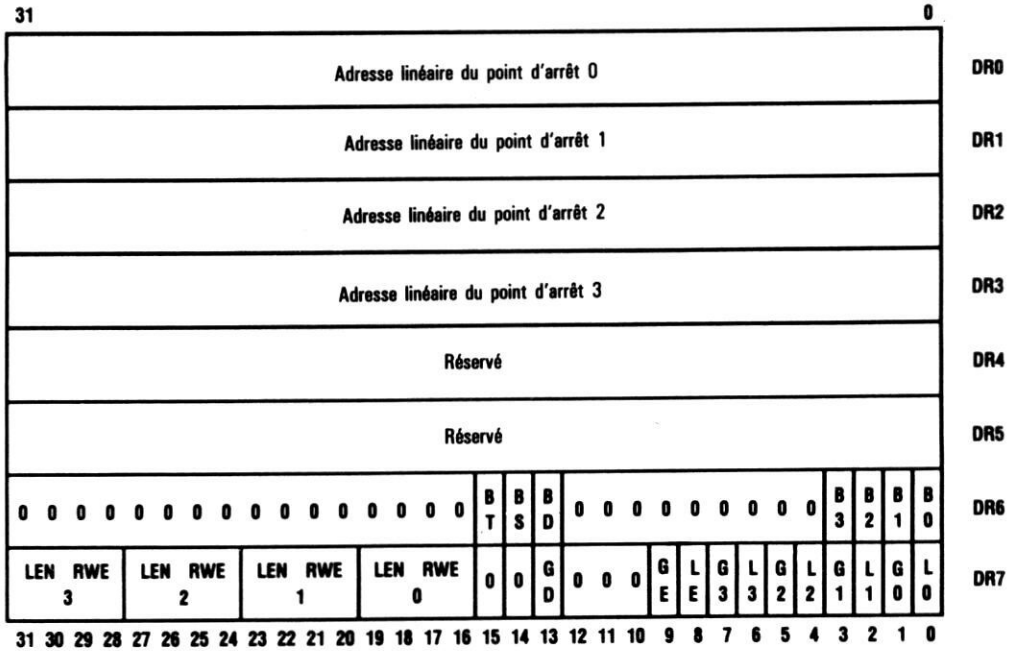


Figure 8.1 : *Registres de mise au point.*

DR4-DR5 — Les registres DR4 et DR5 sont réservés par Intel pour utilisation future.

DR6 — Le registre DR6 est le registre d'état. Lorsqu'une exception du type mise au point est générée, le processeur positionne DR6 pour indiquer le type d'exception et/ou les points d'arrêt qui ont généré cette exception.

DR7 — Le registre DR7 est le registre de contrôle. Les bits de ce registre permettent de valider chaque registre de points d'arrêt, aussi bien que le type de point d'arrêt (instruction, écriture de données, écriture ou lecture de données). Ce registre détermine la protection de tous les registres de points d'arrêt.

DR6

Dès le début du handler de mise au point, le registre DR6 indique qu'une exception a été détectée. Notez que les bits de DR6 n'indiquent pas si la condition du point d'arrêt est valide (c'est le rôle de DR7). Ils indiquent simplement que la condition a été détectée. Ainsi, le programmeur doit vérifier qu'elle a été détectée (DR6) et que la condition n'est pas inhibée (DR7).

Les exceptions de mise au point de deux instructions différentes peuvent être transmises simultanément. Le système peut signaler une faute sur une instruction, et en même temps une trappe sur l'instruction précédente. Par exemple, un point d'arrêt sur une instruction peut être transmis en même temps qu'un point d'arrêt sur des données pour l'instruction précédente. Le handler de mise au point peut examiner les bits de DR6 pour déterminer quelle condition ou combinaison de conditions a été détectée au moment de la génération de l'exception du type mise au point.

B0-B3 — Les bits B0 à B3 indiquent que la condition de point d'arrêt spécifiée par le registre de points d'arrêt correspondant a été détectée. Notez que ces bits sont positionnés même si la condition de point d'arrêt a été inhibée par DR7.

BD — Le bit BD est positionné sur une frontière d'instruction si l'instruction suivante va lire ou écrire dans un des huit registres de mise au point (protection des registres de mise au point). BD sera positionné chaque fois qu'il y aura écriture ou lecture des registres de mise au point.

La condition ne doit pas être validée par le bit GD de DR7.

BS — Le bit BS est positionné si une exception du type mode pas à pas survient. La condition associée est validée par le bit TF du registre EFLAGS (voir le Chapitre 2). Le bit BS de DR6 est positionné si le programme entre dans le handler de mise au point à cause d'une condition de mode pas à pas. Au contraire des autres bits de DR6, le bit BS est positionné uniquement par une trappe de pas à pas et non par la détection d'une condition de mode pas à pas.

BT — Le bit BT indique que la cause de l'exception de mise au point était une commutation vers une tâche où le bit de trappe de mise au point dans le TSS est validé. Il n'existe pas de bit correspondant dans DR7.

Notez que les bits de DR6 ne sont jamais effacés par le processeur. Ainsi, le handler de mise au point doit effacer DR6 avant de rendre la main pour éviter une mauvaise interprétation de DR6 lors de la prochaine condition d'exception.

DR7

Nous allons donner la définition des bits du registre de contrôle DR7 :

LEN — Les quatre champs de deux bits LEN déterminent la longueur du point d'arrêt pour chacun des quatre registres de points d'arrêt. Le codage est le suivant :

LEN	Description
00	Un octet
01	Deux octets

LEN	Description
10	Réservé
11	Quatre octets

Notez que l'adresse linéaire du point d'arrêt doit être alignée sur un multiple de la valeur de ce champ. Si le point d'arrêt correspondant est du type instruction (RWE = 00), LEN doit être égal à 00.

RWE

Les quatre champs de deux bits RWE déterminent le type d'accès qui engendrera une exception du type point d'arrêt. Le codage est le suivant :

RWE	Description
00	Instruction
01	Ecriture de données
10	Réservé
11	Ecriture ou lecture de données

GE/LE

Les bits GE/LE déterminent l'exactitude des points d'arrêt du type donnée (globaux et locaux). Si GE ou LE sont positionnés, le processeur ralentira l'exécution de façon que les points d'arrêt du type donnée soient transmis exactement lors de l'instruction en cause. Si ces bits ne sont pas positionnés, le processeur peut retarder légèrement la transmission d'une condition de point d'arrêt pour des instructions qui effectuent des écritures de données vers la fin de leur exécution. Nous recommandons que soit LE, soit GE soient validés chaque fois que des points d'arrêt du type donnée sont posés. Ce ralentissement du processeur ne devrait pas poser de problèmes, à part une légère perte de performance. Pour les cas où la vitesse est un facteur critique, on devra inhiber GE et LE, et tolérer quelques imprécisions mineures dans la mise au point.

L0-L3/G0-G3 Les bits L0 à L3 et G0 à G3 sont les signaux de validation locaux et globaux des quatre registres de points d'arrêt. Si l'un des Li ou Gi est positionné, le point d'arrêt déterminé par le registre DRi correspondant est valide.

GD Le bit GD valide la condition de protection des registres de mise au point qui est associée à BD de DR6. Notez que GD est effacé par le processeur au début du handler de mise au point. Le handler a donc libre accès aux registres de mise au point.

Notez que les bits L (LE, L0, L1, L2, L3) sont des bits locaux à une tâche. Ils permettent d'établir les conditions de mise au point pour une tâche particulière, tandis que les bits G (GD, GE, G0, G1, G2, G3) sont globaux et doivent être utilisés pour des conditions de mise au point qui sont valides pour toutes les tâches du système. Les bits L sont automatiquement effacés par le processeur à chaque commutation de tâches (voir le Chapitre 5).

Reconnaissance de l'adresse d'un point d'arrêt

La combinaison du champ LEN et de l'adresse linéaire du point d'arrêt détermine les valeurs des quatre adresses linéaires qui sont testées par le mécanisme d'aide à la mise au point. Les adresses linéaires des points d'arrêt doivent être alignées sur des adresses qui sont des multiples de la longueur spécifiée par LEN. En fait, le processeur ignore les bits de poids faible de l'adresse linéaire lorsqu'il teste la présence de points d'arrêt, selon la valeur du LEN. Par exemple, si LEN = 11 (longueur 4), il ignore les 2 bits de poids faible de l'adresse linéaire. Si LEN = 01 (longueur 2), c'est le bit de poids faible qu'il ignore.

Tout accès correct à un octet d'une valeur spécifiée par l'adresse linéaire d'un point d'arrêt et par LEN engendrera une exception. Le processeur teste pour chaque accès à des données et pour chaque instruction les valeurs des quatre adresses de points d'arrêt. Si un octet est compris dans

les valeurs déterminées par les points d'arrêt et que le type d'accès associé correspond, il y a transmission d'une exception du type point d'arrêt.

Le Tableau 8.1 donne plusieurs exemples de reconnaissance de points d'arrêt du type donnée. Vérifiez que tous les points d'arrêt sont valides et associés au type d'accès correct.

Pour positionner un point d'arrêt sur une adresse non alignée, vous devez définir deux conditions de points d'arrêt qui couvrent correctement les adresses désirées. Par exemple, si vous voulez vous arrêter à l'adresse 00FFFEh pour une donnée de quatre octets, vous devez positionner un point d'arrêt à l'adresse 00FFFEH avec LEN = 01 (deux octets) et un second point d'arrêt à l'adresse 010000h avec LEN = 01.

		ADRESSE	LONGUEUR	POINT d'ARRET
	DR0	00FF02	1, LEN = 00	
CONTENU des	DR1	00CC32	2, LEN = 01	
REGISTRES	DR2	0D0004	3, LEN = 11	
	DR4	01FF00	4, LEN = 11	
ACCES		00FF02	1	B0 = 1
CAUSANT UNE		00CC33	2	B1 = 1
EXCEPTION		0D0007	2	B2 = 1
		00FEFF	4	B0 = 1
		01FF00	4	B3 = 1
		01FF03	4	B3 = 1
ACCES NE		00FF01	1	-
CAUSANT PAS		00FF00	2	-
D'EXCEPTION		00CC34	1	-
		01FEFF	1	-
		0D0000	4	-

Tableau 8.1 : *Exemples de reconnaissance de points d'arrêt.*

Points d'arrêt sur code / Points d'arrêt sur donnée

Il existe plusieurs différences entre les points d'arrêt sur instruction (code) et les points d'arrêt sur donnée. Tout d'abord, le champ RWE est évidemment différent pour chacun d'eux. Ensuite, le LEN des points d'arrêt sur instruction ne peut être égal qu'à 00 (un octet). Le point d'arrêt sur instruction doit être posé sur le premier octet de l'instruction (qui peut avoir une longueur de 1 octet à 15 octets). Le *premier octet* inclut tous les octets préfixes qui peuvent être associés à cette instruction.

La troisième différence, la plus significative, est que les points d'arrêt sur instruction sont des fautes, alors que les points d'arrêt sur donnée sont des trappes. Ceci signifie que les points d'arrêt sur instruction sont détectés et transmis avant l'exécution de l'instruction, tandis que les points d'arrêt sur donnée sont transmis après la lecture ou l'écriture à l'adresse spécifiée. Cette différence est importante pour deux raisons. D'abord, un point d'arrêt sur donnée ne protège pas l'adresse de la donnée contre l'écriture. Si vous désirez cette protection, le handler de mise au point peut effectuer une copie de l'ancienne donnée, puisqu'une écriture de donnée survient avant que le handler soit appelé.

Puisque les points d'arrêt sur instruction sont transmis avant l'exécution de l'instruction, il est évident que l'instruction ne peut pas être "relancée" simplement ; à la seconde exécution, il y aura à nouveau génération d'une faute. On aboutit ainsi à une boucle sans fin, à moins que le handler n'inhibe le point d'arrêt. Mais le handler n'a pas besoin de le faire parce que le 80386 possède le bit RF du registre EFLAGS (voir le Chapitre 2). Le bit RF, lorsqu'il est positionné, fait ignorer toute faute du type point d'arrêt sur instruction. Cependant, notez que les autres fautes seront encore transmises. Le bit RF est automatiquement effacé par le processeur à la fin de chaque instruction. Ainsi, le bit RF va obliger le processeur à ne pas transmettre une faute de ce type lors de la seconde (ou n^{ième}) exécution de l'instruction en cause, mais le 80386 recommencera à les transmettre pour les instructions suivantes. RF permet aussi la transmission d'autres types de fautes lors de la seconde exécution de cette instruction. Notez que RF restera positionné jusqu'à ce que l'instruction s'achève. Il est alors automatiquement effacé par le processeur.

Mais comment RF est-il positionné ? Le handler de mise au point doit le positionner avant de rendre la main, ainsi que toutes les autres fautes du processeur. Chaque fois qu'un handler de faute (et pas seulement un handler de faute du type mise au point) est appelé, le processeur positionne ce bit dans les indicateurs qui sont empilés. Ainsi, lorsqu'on arrive à l'IRET du handler, l'image des indicateurs est dépilée et placée dans le registre EFLAGS avec RF positionné.

Notez aussi que le bit RF, lorsqu'il est positionné, inhibe l'état de tous les bits de DR6 pour les instructions de points d'arrêt.

AUTRES POSSIBILITES DE MISE AU POINT

Les points d'arrêt ne sont pas les seules possibilités du 80386 pour la mise au point, bien qu'ils soient les plus importants. Nous allons résumer les autres possibilités.

Trappe de mise au point dans le TSS

Le bit T qui se trouve dans le TSS (voir le Chapitre 5) permet d'appeler le handler de mise au point chaque fois que survient une commutation de tâches via ce TSS. Ceci permet de contrôler facilement les activités de certaines tâches. Le bit BT de DR6 indique que cette condition a été détectée. Il n'existe pas de bit spécifique qui valide cette condition dans DR7.

Si le handler de mise au point (interruption 1) est associé à un guichet de tâche (c'est un bon choix), il est important de ne pas positionner l'indicateur de trappe du type mise au point du TSS correspondant. Sinon, on obtiendra une boucle sans fin qui tentera d'appeler le handler.

INT3

L'instruction de point d'arrêt (1 octet) INT 3 offre d'autres outils de mise au point. Le premier octet de l'instruction où on désire s'arrêter peut être remplacé par l'instruction INT 3. Ainsi, lorsque le processeur rencontre l'instruction INT 3, il appelle le handler associé à l'interruption 3.

Souvent, INT 3 n'est pas suffisant. Il faut alors opter pour les registres de points d'arrêt :

1 Vous ne pouvez pas insérer une instruction INT 3 dans du code en ROM.

2 Lorsqu'un INT 3 est exécuté, le code est modifié, et ainsi *toute* tâche exécutant ce code est interrompue. Avec des points d'arrêt, vous pouvez déterminer l'arrêt sur une instruction pour une tâche particulière même si le code est partagé.

3 INT 3 ne peut évidemment pas simuler un point d'arrêt sur donnée.

Toutefois, INT 3 est intéressant dans plusieurs cas :

1 Le mode pas à pas et les points d'arrêt appellent le handler de mise au point qui est associé à l'interruption 1. Comment mettre au point ce handler ? L'instruction INT 3 le permet.

2 Vous pouvez insérer un nombre illimité d'instructions INT 3 dans du code, alors qu'il n'existe que quatre registres de points d'arrêt.

3 Les anciens processeurs de la famille 86 ne possédaient pas de registres de mise au point. Ainsi, l'instruction INT 3 jouait un rôle important pour ces processeurs, puisqu'elle représentait le seul moyen d'effectuer un point d'arrêt.

En résumé, nous recommandons (dans la limite des cas particuliers précédents) un emploi illimité des instructions INT 3 pour poser des points d'arrêt. Nous pensons aussi qu'il vaut mieux consacrer les registres de mise au point aux points d'arrêt sur donnée.

Mode pas à pas

Le mode pas à pas est un moyen pratique pour se déplacer dans un programme. La trappe de pas à pas est validée par le positionnement du bit TF du registre EFLAGS (voir le Chapitre 2). Si TF = 1 au début de l'instruction, il y a génération d'une exception du type mise au point à la fin de cette instruction (trappe) qui appelle le handler de mise au point.

Notez que le bit TF doit être positionné lors de l'exécution de l'instruction en cause. Ainsi, la trappe n'est pas transmise lors de l'instruction qui a positionné l'indicateur TF. Le bit TF est effacé avant d'entrer dans le handler. Il est aussi effacé lorsqu'il y a interruption ou exception. Si une interruption externe survient en même temps qu'une interruption du type pas à pas, c'est cette dernière qui est traitée en premier, ce qui efface le bit TF. Avant l'exécution de la première instruction du handler, l'interruption est transmise si elle est encore en attente. Ainsi, le traitement de l'interruption est effectué sans mode pas à pas. Si vous voulez travailler avec ce mode dans le handler, vous pouvez positionner un point d'arrêt sur sa première instruction, puis passer en mode pas à pas lors de l'arrêt.

9

Exécution de programmes 8086 et 80286

Dans les huit chapitres précédents de cet ouvrage, nous avons considéré le 80386 comme un processeur 32 bits. Dans ce chapitre, nous allons nous préoccuper de la compatibilité du 80386 qui permet d'exécuter sur ce processeur des programmes conçus pour les processeurs précédents : 8086, 8088, 80186, 80188, et 80286.

Nous avons vu au Chapitre 1 que les architectures des 8086, 8088, 80186 et 80188 sont identiques. Le 80286 possède en plus une gestion complète de la mémoire et de la protection. Il travaille selon deux modes de fonctionnement qui se distinguent par leur modèle de protection. Il est possible d'utiliser le 80286 sans le modèle de protection. Dans ce mode, appelé le mode réel, le 80286 se comporte exactement comme un 8086. Lorsque le modèle de protection est activé, on passe en mode protégé et le processeur se comporte tout à fait différemment.

Etant donné que les 8086, 8088, 80186, 80188 et le 80286 en mode réel peuvent paraître identiques, nous allons les décrire dans ce chapitre en choisissant le 8086 comme modèle. L'Annexe A rappelle les quelques différences mineures entre ces processeurs. Nous parlerons du 80286 en mode protégé dans un paragraphe distinct puisque son fonctionnement est alors différent.

Tout d'abord, nous allons décrire les registres de base et les modes d'adressage qu'utilisent les programmes 16 bits. Ceci nous permettra de faire un bref parallélisme avec le Chapitre 2. Puis nous expliquerons comment des programmes 8086 peuvent s'exécuter sur le 80386. Deux modes permettent d'exécuter des programmes 8086 : le mode virtuel 8086 et le mode réel. Le mode réel du 80386, comme celui du 80286, est un mode d'exécution qui n'utilise pas le mécanisme de protection défini au Chapitre 5. Le mode virtuel 8086 est une nouveauté du 80386 qui permet d'exécuter des programmes 8086 tout en restant en mode protégé. Pour finir, nous parlerons du cas particulier des programmes 80286.

REGISTRES 16 BITS ET MODES D'ADRESSAGE

Les programmes sur 16 bits emploient les 16 bits de poids faible des registres définis au Chapitre 2. La Figure 9.1 illustre ces registres. Il y a huit registres 16 bits : AX, CX, DX, BX, SP, BP, SI et DI. Ils sont constitués des 16 bits de poids faible des registres 32 bits généraux décrits au Chapitre 2. Les six registres segments 16 bits décrits au Chapitre 2 sont employés par les programmes 16 bits. Les registres FS et GS sont une nouveauté du 80386. Ainsi, les programmes 16 bits écrits pour s'exécuter sur un 8086 ou un 80286 n'utiliseront pas ces registres.

L'adressage mémoire est identique au mécanisme d'adressage mémoire décrit au Chapitre 2. Chaque référence mémoire se décompose en une composante segment et une composante déplacement. Puisque les programmes 16 bits utilisent des déplacements 16 bits, la taille maximale d'un segment est égale à 64 Ko. Ils peuvent générer des déplacements de segments qui correspondent aux modes d'adressage du Tableau 9.1. Un déplacement 16 bits est formé en additionnant un registre de base 16 bits, un registre d'index 16 bits et un déplacement 8 ou 16 bits. Ces additions emploient une arithmétique 16 bits, si bien que la retenue provenant du bit 15 est ignorée. L'Annexe G décrit le codage des modes d'adressage 16 bits.

Les modes d'adressage 16 bits sont moins généraux que les modes d'adressage 32 bits, bien qu'ils opèrent sur les mêmes composantes. On peut employer uniquement les registres BX et BP comme registres de base et les registres SI et DI comme registres d'index. Lorsque BP est utilisé comme registre de base, SS est le registre segment par défaut. Sinon, le registre segment par défaut est DS.

L'emploi de registres 16 bits est contrôlé par la taille de l'opérande d'une instruction. L'emploi des modes d'adressage 16 bits est contrôlé par la taille de l'adresse d'une instruction. Rappelez-vous (voir le Chapitre 5) que le bit D du descripteur de segment code détermine la taille par défaut des opérandes et des adresses. Les programmes 80286 utilisent une taille par défaut de 16 bits en effaçant le bit D des descripteurs de tous les segments code. Les programmes qui s'exécutent en mode réel ou en mode virtuel ont une taille par défaut de 16 bits. On peut utiliser un préfixe

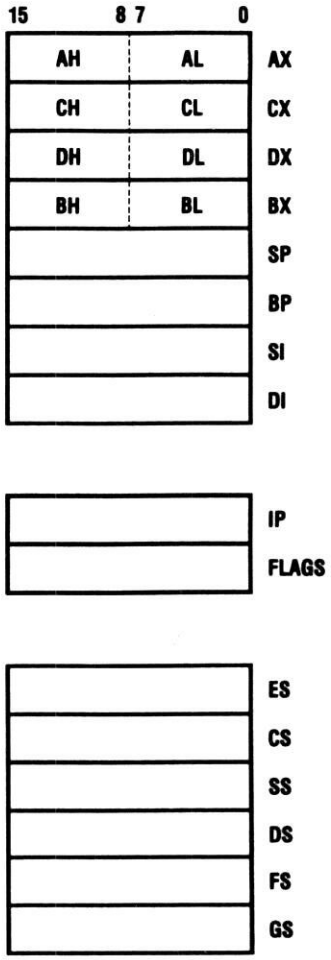


Figure 9.1 : *Registres 16 bits.*

BASE	+	INDEX	+	DEPLACEMENT
Aucun		Aucun		Aucun
BX	+	SI	+	8 bits
BP[1]		DI		16 bits

[1] SS est le registre segment par défaut si BP est le registre de base.

Tableau 9.1 : *Modes d'adressage 16 bits.*

d'instruction pour remplacer la taille par défaut des opérandes ou des adresses de toute instruction (voir l'Annexe G).

Les registres à virgule flottante sont les mêmes pour le 80387, le 8087 et le 80287. Tout programme 16 bits écrit pour les 8086/8087 ou les 80286/80287 utilise le jeu de registres à virgule flottante décrit au Chapitre 2.

EXECUTION DE PROGRAMMES 8086

Les programmes 8086 (et 8088, 80186, 80188 et 80286 en mode réel) disposent de deux modes d'exécution : le mode réel et le mode virtuel 8086. Lorsqu'il travaille dans un de ces modes, le 80386 se programme comme un 8086 (à de rares exceptions près : voir l'Annexe A).

Le mode réel du 80386 est identique à celui du 80286. Lorsqu'il travaille en mode réel, le 80386 se comporte exactement comme un 8086, ce qui lui permet d'exécuter directement des programmes 8086. Le 80386 passe dans ce mode lorsque les mécanismes de gestion mémoire, de la

protection et des tâches (voir le Chapitre 5) sont inhibés en effaçant le bit PE du registre CR0. Puisque le mécanisme de protection est inhibé en mode réel, le 80386 se comporte comme s'il opérait au niveau de privilège 0. En mode réel, c'est un mécanisme différent de celui décrit au Chapitre 6 qui gère les interruptions et les exceptions. Nous allons bientôt en parler.

Lorsque le 80386 est électroniquement réinitialisé, il débute en mode réel. Le code de lancement peut ainsi initialiser les tables du système avant de mettre en place la protection (voir le Chapitre 7).

Le mode virtuel 8086 est une nouveauté du 80386. Il permet l'exécution de programmes 8086 tout en restant dans le contexte des mécanismes de gestion mémoire, de protection et de tâches décrits au Chapitre 5. Un logiciel particulier du système d'exploitation, connu sous le nom de moniteur de machine virtuelle, peut créer une tâche qui exécute un programme 8086 en mode virtuel 8086. On a ainsi l'illusion que le programme s'exécute sur un 8086 avec toutes les ressources de ce processeur. Il est possible d'exécuter un programme 8086 dans une tâche en mode virtuel 8086, un programme en mode protégé 16 bits 80286 dans une autre tâche, et un programme en mode protégé 32 bits 80386 dans une troisième tâche. C'est le mécanisme de gestion multitâche du 80386 qui permet cette flexibilité dans l'exécution des programmes.

Un programme en mode virtuel s'exécute au niveau de privilège 3. On peut employer la pagination pour allouer et protéger la mémoire pour des programmes en mode virtuel 8086. Si une interruption est reçue ou une exception générée lorsqu'un programme s'exécute en mode virtuel 8086, le processeur commute en mode normal protégé pour prendre en compte l'interruption ou l'exception (voir les Chapitres 5 et 6). Dans le paragraphe suivant, nous allons décrire la commutation du mode virtuel 8086 en mode protégé.

Un moniteur de machine virtuelle peut traiter plusieurs programmes 8086 "à la fois" en créant plusieurs tâches qui s'exécutent en mode virtuel 8086. Grâce à l'illusion donnée par la combinaison du moniteur de machine virtuelle et des possibilités matérielles du mode virtuel 8086, chacun de ces programmes 8086 s'exécutera comme sur un 8086. D'autres tâches

du système peuvent exécuter des programmes 80286 16 bits en mode protégé, ou des programmes 80386 32 bits en mode protégé et des commutations classiques de tâches. Ainsi, le 80386 peut exécuter des tâches très diverses, certaines en mode virtuel 8086 et d'autres en mode protégé, et passer facilement de l'une à l'autre. C'est un mécanisme très puissant qui permet d'employer l'énorme bibliothèque de programmes pour compatibles PC 16 bits (8086 et 80286) tout comme des applications 32 bits très sophistiquées écrites initialement pour des miniordinateurs ou pour des unités centrales. Le mode virtuel 8086 et les possibilités de gestion multitâche du 80386 sont les clés matérielles qui permettent une telle puissance logicielle.

Contrairement à ces puissants outils, le mode réel ne permet pas la gestion multitâche. Il ne gère qu'un programme 8086. Ce dernier accédera directement au processeur 80386, sans protection, sans gestion mémoire, et sans possibilités multitâches.

Dans les paragraphes suivants, nous allons traiter des mécanismes communs au mode réel et au mode virtuel 8086. Nous parlerons ensuite des particularités du mode réel et du mode virtuel 8086.

Segmentation et adressage

Le modèle de programmation du 8086 utilise la segmentation pour obtenir un modèle d'adressage mémoire à deux dimensions qui est le même que celui décrit au Chapitre 2. Les programmes 8086 emploient les mêmes adresses mémoire à deux composantes que les programmes en mode protégé. Ces adresses comportent une composante segment et une composante déplacement. La partie déplacement est générée à partir d'un des modes d'adressage 16 bits du Tableau 9.1. La composante segment d'une adresse est obtenue à partir d'un registre segment qu'on doit avoir chargé auparavant pour adresser le segment désiré. Bien que la composante segment d'une adresse soit obtenue à partir d'un registre segment en mode protégé, elle est traitée d'une manière tout à fait différente dans les programmes 8086.

L'adressage mémoire du 8086 n'emploie pas les tables des descripteurs de segments (voir le Chapitre 5) pour définir l'adresse de base, la limite, et les attributs du segment. Au contraire, les segments ont une taille arbitraire de 64 Ko, et autorisent des accès en lecture, écriture ou exécution. Les segments en mode réel ont implicitement un DPL égal à 0, puisque les programmes en mode réel s'exécutent comme s'ils étaient au niveau de privilège 0. Les segments en mode virtuel 8086 ont un DPL égal à 3, puisque les programmes en mode virtuel 8086 s'exécutent au niveau de privilège 3. Les segments adressés par CS ont implicitement leur bit D (*Défaut*) égal à 0, si bien que les programmes 8086 ont des opérandes et des adresses sur 16 bits. Les segments adressés par SS ont implicitement leur bit D égal à 0, si bien que toute référence à la pile emploie le registre 16 bits ESP.

La valeur du registre segment est utilisée pour déterminer l'adresse de base du segment dans l'espace adressable linéaire comme la valeur numérique du registre segment multipliée par 16 (ou décalée 4 fois à gauche). Puisque la valeur d'un registre segment de 16 bits est limitée à 2^{16}, la plus grande adresse de base de segment est égale à 2^{20} (après multiplication par 16). Ceci signifie qu'un segment peut commencer à n'importe quelle frontière de 16 octets dans le premier méga-octet de l'espace adressable linéaire.

L'adresse de base du segment est ajoutée au déplacement déterminé par le mode d'adressage de l'instruction pour obtenir l'adresse linéaire de l'emplacement mémoire désiré. Ce mécanisme utilise une arithmétique 32 bits (voir la Figure 9.2). Cette addition conduit à un résultat sur 21 bits sur le 80386, et le vingt et unième bit est la retenue sortant du bit 20. Puisque les limites des segments sont fixées à 64 Ko, le plus grand déplacement qui peut être généré est 0FFFFh (voir la Figure 9.2).

Puisque les valeurs des registres segments sont traitées simplement comme des nombres 16 bits, on ne peut obtenir une exception lorsqu'on charge une valeur dans un registre segment. Aucun test n'est effectué sur la valeur chargée, si bien qu'on peut utiliser n'importe quel nombre 16 bits. Par exemple, on peut employer la valeur 0, qui serait considérée comme un sélecteur nul par le mécanisme de protection.

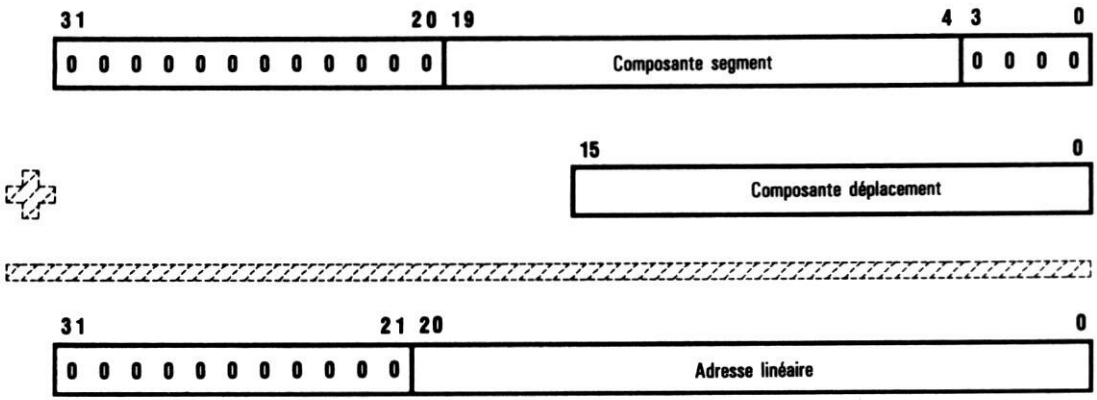


Figure 9.2 : *Formation d'une adresse linéaire en mode réel et en mode virtuel 8086.*

Par exemple, considérons qu'une référence mémoire spécifie un registre segment qui contient la valeur 1234h, et un mode d'adressage qui donne un déplacement égal à 5. Le calcul de l'adresse linéaire résultante est le suivant :

```
   12340h      la valeur du registre segment multipliée par 16
+      5h      le déplacement donné par le mode d'adressage
   ------
   12345h      adresse linéaire résultante
```

Une valeur de registre segment de 0FFFFh et un déplacement de 0FFFFh génèrent la plus grande adresse linéaire. Le calcul est le suivant :

```
   FFFF0h      la valeur du registre segment multipliée par 16
+   FFFFh      le déplacement donné par le mode d'adressage
  -------
  10FFEFh      adresse linéaire résultante
```

L'ajout de la base du segment conduit à l'adresse linéaire d'un opérande mémoire. En mode réel, c'est aussi l'adresse physique, puisqu'on ne peut pas utiliser la pagination dans ce mode. En mode virtuel 8086, on peut employer la pagination pour transformer cette adresse linéaire en adresse physique, si la pagination est activée. Lorsque la pagination n'est pas activée en mode virtuel 8086 ou en mode réel, la plus grande adresse physique qu'on peut générer est égale à 10FFEFh, c'est-à-dire presque 1 Mo. Si la pagination est activée en mode virtuel 8086, n'importe quelle page de 1 Mo dans l'espace adressable linéaire peut être associée à une page de l'espace adressable physique de 4 Go.

Exceptions lors de références mémoire

Lorsque le 80386 travaille en mode réel ou en mode virtuel 8086, il peut générer des exceptions du type segment pile et protection générale lorsque les références mémoire dépassent les limites des segments fixées à 64 Ko. Un déplacement généré à partir d'un mode d'adressage 32 bits peut excéder 64 Ko. L'emploi d'un mode d'adressage 32 bits nécessite l'utilisation d'un préfixe de taille d'adresse. Les programmes 8086 n'emploient que des modes d'adressage 16 bits. Ceux-ci peuvent générer une référence qui dépasse les limites fixées de 64 Ko si le déplacement est proche de 64 Ko et qu'on fait référence à un opérande sur deux ou quatre octets. Par exemple, un opérande sur quatre octets au déplacement 0FFFEh générera une exception, puisque les deux derniers octets sont au-delà des 64 Ko.

Notez que le 8086 ne génère pas d'exception dans ce cas. Au contraire, on se retrouve cycliquement au déplacement 0.

L'exception transmise lorsqu'une référence mémoire dépasse la limite fixée de 64 Ko dépend du registre segment utilisé. Si SS est le registre segment, on aura une exception du type segment pile. Sinon, c'est une exception du type protection générale qui sera transmise.

Si la pagination est activée, un programme qui s'exécute en mode virtuel 8086 peut aussi générer une exception du type page s'il génère une adresse linéaire non valide ou qui viole le mécanisme de protection (voir le Chapitre 5).

On peut obtenir d'autres exceptions lorsqu'un programme s'exécute en mode réel ou en mode virtuel 8086. Par exemple, le processeur générera une erreur du type division si une instruction DIV ou IDIV a un opérande nul. La plupart des exceptions décrites au Chapitre 6 peuvent survenir avec un programme 8086. Seules les exceptions associées au chargement des registres segments sont impossibles, puisque ces registres sont traités de façon tout à fait différente (valeurs 16 bits) dans ce cas.

Références mémoire en mode réel et en mode virtuel 8086

Les instructions qui chargent les registres segments s'exécutent différemment suivant qu'on se trouve en mode réel, en mode virtuel 8086 ou en mode protégé. Au lieu d'utiliser la sémantique du Chapitre 5, les descriptions plus simples du Chapitre 3 suffisent.

La routine AccessVirtual() du Chapitre 5 devient la routine Access8086Virtual() du Listing 9.1, lorsque le processeur travaille en mode réel ou en mode virtuel 8086. Dans ces modes, aucune référence n'est faite aux registres cachés des descripteurs. Au contraire, la base du segment est obtenue en multipliant la valeur du registre sélecteur de segment par 16. De plus, le programme teste le déplacement et la longueur du segment

```
/* Accès de données dans un segment adressé par un */
/* Registre Segment SReg                           */
/* Version en mode réel 8086 ou mode virtuel 8086  */
Access8086Virtual(SReg, Offset, Length, RW, IntNumber, Data)
   SegmentRegister SReg;
   int Offset, Length,    /* déplacement dans le segment, */
                          /* et longueur de la donnée */
       RW,                /* 0 si lecture, 1 si écriture */
       IntNumber,         /* $SS pour un accès de SS, $GP */
                          /* pour les autres registres */
       *Data;             /* Pointeur sur Donnée pour */
                          /* lecture ou écriture */
{
   /* Teste pour vérifier que l'accès ne dépasse pas */
   /* les 64 K du segment */
   if   (Offset+Length > 10000h)
      SegmentException(IntNumber, 0);

   /* Si le déplacement est dans la limite, accès à */
   /*l'espace linéaire */
   AccessLinear(SReg.Selector*16 + Offset, Length, CPL, RW,
                Data);

} /* fin de Access8086Virtual */
```

Listing 9.1 : *Sous-programme Access8086Virtual().*

pour vérifier que la référence est bien compatible avec les limites de segments 64 Ko. Dans ces modes, tous les accès sont autorisés (lecture, écriture, exécution).

Emulation de l'arithmétique 20 bits (adresses 8086)

Le 8086 travaille sur un espace adressable physique de 20 bits, alors que le 80386 emploie une arithmétique 32 bits pour calculer les adresses linéaires. Il est possible que des programmes 8086 fonctionnent mal si on ne traite pas convenablement le vingt et unième bit. En mode réel, ou en mode virtuel sans pagination, le processeur utilise directement l'adresse linéaire sur 21 bits comme adresse physique. Dans ce cas, on doit utiliser des circuits externes qui simulent l'espace adressable sur 20 bits du 8086. La solution matérielle la plus simple consiste à employer uniquement les 20 broches d'adresses de poids faible, et à ignorer les 12 bits d'adresses de poids fort. Une autre solution consiste à forcer à 0 par un circuit logique externe le bit d'adresse 20 lors de l'exécution de programmes 8086.

On peut employer la pagination en mode virtuel 8086 pour simuler l'espace linéaire sur 20 bits. Dans ce cas, on associe les pages d'adresses linéaires comprises entre 100000h et 10FFEFh aux mêmes adresses physiques que les pages d'adresses linéaires comprises entre 0 et FFEFh. Cette technique simule une arithmétique 20 bits en associant les adresses des pages contenues dans 1 Mo aux pages correspondantes qui ont la même adresse linéaire modulo 2^{20}. Cette technique permet de simuler l'adressage du 8086 sans matériel externe.

La Figure 9.3 illustre l'emploi de cette technique. Pour simplifier, on s'est restreint au premier méga-octet de l'espace adressable linéaire. En effet, une adresse linéaire comprise entre 0 et 00FFFFh (resp. 010000h et 0FFFFFh) est associée à une adresse physique comprise entre 0 et 00FFFFh (resp. 010000 à 0FFFFFh). Une adresse linéaire comprise entre 100000h et 10FFFFh est associée à une adresse physique comprise entre 0 et 00FFFFh pour simuler l'adressage sur 20 bits du 8086.

Instructions non valides

Les instructions du Tableau 9.2 ne sont pas permises en mode réel ou en mode virtuel 8086. Si on emploie une de ces instructions dans un de ces

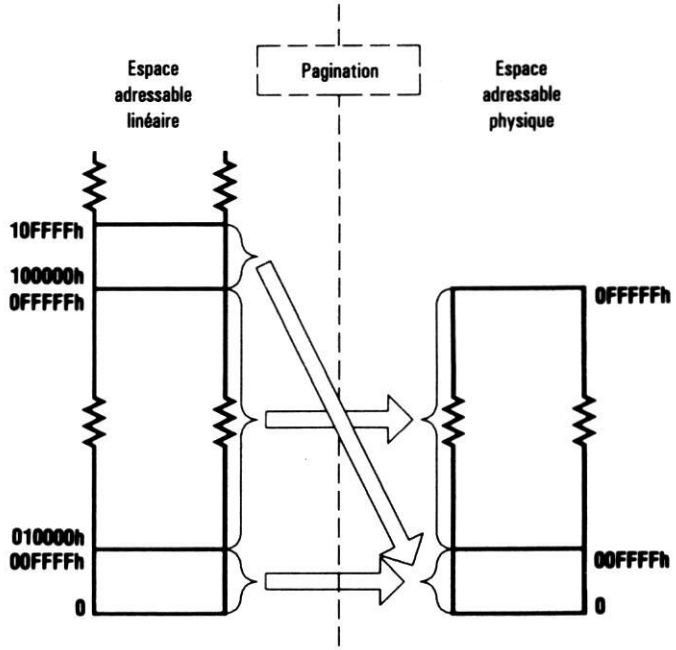


Figure 9.3 : *La pagination simule l'adressage sur 20 bits du 8086.*

```
ARPL
LAR
LLDT
LSL
LTR
SLDT
STR
VERR
VERW
```

Tableau 9.2 : *Instructions non valides en mode réel et en mode virtuel 8086.*

modes, il y aura génération d'une exception du type CodeOp non valide (voir le Chapitre 6). Le 8086 ne permet aucune de ces instructions. En effet, elles manipulent les sélecteurs et les descripteurs du mode protégé, ce qui n'a aucun sens dans les modes considérés.

Format 8086 pour FLDENV, FSTENV et FNSTENV

Les instructions FLDENV, FSTENV et FNSTENV (voir le Chapitre 3) chargent ou sauvent le mot de contrôle, le mot d'état, le mot d'étiquette, et les registres pointeurs d'erreur. Dans le Chapitre 2, nous avions détaillé le format mémoire employé par ces instructions en mode protégé. La Figure 9.4 nous montre le format de ces instructions avec des opérandes 16 bits (la taille par défaut) en mode réel ou en mode virtuel 8086. Les programmes qui s'exécutent sur un 8086 spécifient des opérandes 16 bits pour ces instructions. On peut utiliser des opérandes 32 bits avec le 80386, mais ce n'est ni utile ni recommandé.

Le format du mot de contrôle, du mot d'état et du mot d'étiquette est le même que celui décrit au Chapitre 2, mais les pointeurs d'erreur sont rangés différemment. Au lieu d'avoir deux composantes, ils sont considérés comme des adresses linéaires 20 bits. Pour le format du CodeOp, reportez-vous au Chapitre 2.

Les instructions FRSTOR, FSAVE, et FNSAVE chargent et sauvent les accumulateurs à virgule flottante en plus des registres traités par FLDENV, FSTENV et FNSTENV. Si on exécute ces instructions avec des opérandes 16 bits, le mot de contrôle, le mot d'état, le mot d'étiquette et les registres pointeurs d'erreur seront stockés sous le format de la Figure 9.4. Puis ces instructions chargeront ou rangeront ST(0) au déplacement 14, ST(1) au déplacement 24, ST(2) au déplacement 34, et ainsi jusqu'à ST(7) au déplacement 84. Dans tous les modes d'exécution, ST(0) à ST(7) sont traités comme des réels temporaires 80 bits.

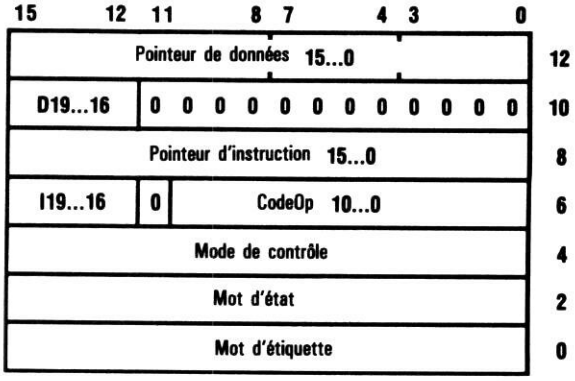


Figure 9.4 : *Format 8086 pour FLDENV, FSTENV et FNSTENV.*

Mode virtuel 8086

Dans les paragraphes suivants, nous allons décrire les caractéristiques spécifiques au mode virtuel 8086. On a vu que le processeur travaille en mode virtuel 8086 lorsque le bit VM du registre EFLAGS est égal à 1. Tout d'abord, nous allons décrire les changements effectués sur les instructions liées au niveau de privilège E/S. Puis nous expliquerons la méthode employée pour commuter du mode protégé au mode virtuel 8086.

IOPL en mode virtuel 8086

Lorsque le 80386 travaille en mode virtuel 8086, les instructions qui effectuent des accès au registre EFLAGS sont liées au niveau de privilège E/S (IOPL) (voir le Chapitre 5). Les instructions CLI et STI sont liées à IOPL pour tous les modes (réel, virtuel 8086 et protégé) du 80386. Les instructions PUSHF, POPF, INT n et IRET sont modifiées afin de générer une exception du type protection générale avec un code d'erreur nul si elles sont exécutées en mode virtuel 8086 lorsque IOPL est inférieur à 3 (le niveau de privilège des programmes en mode virtuel 8086). Ainsi, toutes les instructions qui peuvent modifier ou stocker le bit IF de EFLAGS sont liées à IOPL. Ceci permet aussi au logiciel du système d'exploitation de gérer un registre "EFLAGS virtuel".

Lorsque les instructions E/S IN, OUT, INS et OUTS sont exécutées en mode virtuel 8086, elles ne testent pas IOPL avant de tester la table E/S (voir le Chapitre 5) comme le font les programmes en mode protégé. Au contraire, ces instructions E/S font toujours référence à la table E/S rangée dans le TSS courant pour déterminer si la référence E/S est permise (bits égaux à 0), ou pour générer une exception du type protection générale (un ou plusieurs bits égaux à 1 ou en deçà des limites du TSS). Les instructions E/S ignorent le champ IOPL en mode virtuel 8086.

Puisque les instructions E/S ne testent pas IOPL en mode virtuel 8086 et que les instructions PUSHF, POPF, INT n, et IRET testent IOPL (comme CLI et STI), ce champ est interprété de façon tout à fait différente suivant qu'on se trouve en mode virtuel 8086 ou en mode protégé. En mode virtuel 8086, IOPL est employé pour protéger le bit IF du registre EFLAGS, et n'a rien à voir avec les instructions E/S. En mode protégé, IOPL affecte les instructions E/S aussi bien que certaines instructions qui peuvent modifier IF (CLI et STI).

Mode virtuel 8086 avec IOPL < 3

Le système d'exploitation peut positionner IOPL à une valeur inférieure à 3, si bien qu'un programme en mode 8086 générera une exception chaque fois qu'il exécutera une instruction qui modifie ou stocke le bit IF. Le handler associé à l'exception du type protection générale peut alors gérer un registre EFLAGS virtuel qui est différent du registre EFLAGS réel du 80386. C'est le programme en mode virtuel 8086 qui positionne le bit IF du registre EFLAGS virtuel, alors que, en général, le registre EFLAGS réel a son bit IF égal à 1 (interruptions activées) pour des programmes en mode virtuel 8086. Le système d'exploitation peut utiliser le bit IF du registre EFLAGS virtuel pour déterminer si les interruptions doivent être transmises à la tâche virtuelle 8086 (IF virtuel = 1), ou mises en attente (IF virtuel = 0). L'instruction PUSHF génère une exception, si bien que le handler associé peut émuler cette instruction en empilant le registre EFLAGS virtuel plutôt que le registre EFLAGS réel.

Mode virtuel 8086 avec IOPL = 3

Vous pouvez exécuter un programme en mode virtuel 8086 avec IOPL égal à 3. Ceci permet aux instructions qui accèdent au bit IF de s'exécuter sans générer d'exception, et au programme en mode virtuel 8086 de garder le contrôle des interruptions INTR. Ainsi, on peut accélérer la vitesse d'exécution d'un programme 8086 qui exécute un grand nombre d'instructions liées à IOPL. Si de tels programmes étaient exécutés avec IOPL inférieur à 3, le système générerait de nombreuses exceptions du type protection générale, causant une perte de temps importante lors des nombreux appels au handler associé. Avec IOPL égal à 3, ces instructions s'exécutent à la vitesse normale.

Cette approche est dangereuse. En effet, un programme virtuel 8086 pourrait, par erreur ou par malice, masquer les interruptions INTR pendant un temps conséquent. Ceci bloquerait le système d'exploitation qui ne pourrait traiter les interruptions à temps.

Pour éviter ce problème, vous pouvez employer un *watchdog timer* pour générer une interruption NMI si les interruptions INTR sont désactivées trop longtemps. Par exemple, le timer NMI pourra être réglé de façon à générer une interruption au bout d'une durée égale à deux fois la période du timer système lié à une interruption. Le handler associé au timer système devra réinitialiser le watchdog chaque fois que ce dernier répondra à une interruption INTR. Dans cette configuration, le watchdog générera une NMI uniquement si une INTR liée au timer système n'a pas été traitée pendant une durée dangereusement longue.

Un autre problème se pose lorsqu'on exécute un programme virtuel 8086 avec IOPL = 3. En effet, l'instruction INT *n* ne générera pas une exception du type protection générale mais, au contraire, une trappe avec un numéro de vecteur n, à moins que le DPL du guichet d'IDT soit inférieur à 3 (auquel cas, on aura une exception du type protection générale). Pour les numéros de vecteurs qui doivent avoir leurs DPL égaux à 3, l'emploi d'une instruction INT *n* en mode virtuel 8086 pour entrer dans le handler peut être simplement détecté en utilisant le bit VM de l'image sauvegardée de EFLAGS.

Entrer et quitter le mode virtuel 8086

Le 80386 peut commuter entre le mode protégé et le mode virtuel 8086 en utilisant les mécanismes décrits dans les paragraphes suivants. Comme dans le cas des interruptions et des exceptions, ces transferts peuvent survenir dans une tâche unique ou en multitâche. Il y a commutation du mode protégé au mode virtuel 8086 dans les cas suivants :

1 Une instruction IRET en mode protégé avec NT = 0 dans le registre EFLAGS dépilera EIP, CS et EFLAGS pour retourner à une procédure interrompue dans la même tâche. Si le bit VM est égal à 1 dans la valeur dépilée de EFLAGS, et si l'IRET est exécuté avec CPL = 0, la procédure interrompue se poursuit en mode virtuel 8086. Si IRET est exécuté avec CPL différent de 0, le bit VM n'est pas modifié, si bien que le retour s'effectue en mode protégé (voir le Chapitre 5). Notez aussi que la taille de l'opérande de l'IRET doit

être de 32 bits. Un IRET avec un opérande 16 bits récupérera uniquement les 16 bits de poids faible de EFLAGS et laissera VM à 0.

2 Une commutation de tâches vers une tâche décrite par un TSS 386 entraînera la poursuite de l'exécution de la nouvelle tâche en mode virtuel 8086 si l'image de EFLAGS chargée à partir du nouveau TSS est telle que VM = 1. Une instruction JMP ou CALL vers un guichet de tâche ou un TSS, une interruption vers un guichet de tâche ou un IRET avec NT = 1 dans le registre EFLAGS peuvent entraîner une commutation de tâches. Puisqu'un TSS 286 sauvegarde uniquement les 16 bits de poids faible de EFLAGS, le bit VM est chargé à 0 lors d'une commutation de tâches utilisant un TSS 286.

Lorsqu'une interruption ou une exception surviennent, on obtient une commutation du mode virtuel 8086 au mode protégé. L'interruption ou l'exception est traitée par cette commutation de mode, puis en mode protégé par le mécanisme décrit au Chapitre 6. Ce mécanisme emploie l'IDT (table des descripteurs d'interruptions) pour se brancher au handler associé, selon une des deux méthodes suivantes :

1 L'interruption ou l'exception peuvent être traitées dans la même tâche si l'élément de l'IDT associé contient un guichet de trappe 386 ou un guichet d'interruption 386. On doit employer les types de guichets 386 pour que le registre 32 bits EFLAGS sauvegardé (avec VM = 1) soit récupéré par le prochain IRET. Le handler d'interruption ou d'exception doit être au niveau de privilège 0, sinon on obtiendra une exception du type protection générale avec le nouveau sélecteur de CS comme code d'erreur. Dans ce cas, l'IRET sera exécuté au niveau de privilège 0 pour permettre de changer le bit VM en 1.

2 L'interruption ou l'exception sont traitées dans une tâche distincte si l'élément de l'IDT contient un guichet de tâche. Dans ce cas, le TSS courant doit être du format TSS 386 pour s'assurer que les 32 bits de EFLAGS peuvent être sauvegardés. La nouvelle tâche peut être d'un type quelconque. Elle peut contenir un programme 16 bits en mode

protégé décrit par un TSS 286, ou un programme 32 bits en mode protégé décrit par un TSS 386 avec VM = 0, ou même une autre tâche virtuelle 8086 décrite par un TSS 386 avec VM = 1.

Puisque le système ignore le bit NT du registre EFLAGS en mode virtuel 8086, il n'est pas possible d'utiliser IRET pour effectuer une commutation de tâches à partir d'une tâche en mode virtuel 8086. Dans ce cas, l'IRET dépilera les registres sauvegardés IP, CS et FLAGS pour poursuivre l'exécution du programme interrompu, quelle que soit la valeur du bit NT.

Changement de mode dans une tâche

Si le processeur en mode virtuel 8086 reçoit une interruption ou une exception traitée par un guichet de trappe ou d'interruption, le 80386 passe du mode virtuel 8086 au mode protégé dans la même tâche. Le prochain IRET poursuivra l'exécution du programme en mode virtuel 8086 interrompu en commutant du mode protégé au mode virtuel 8086. Ces changements de modes font aussi appel au mécanisme de protection. Le programme en mode virtuel 8086 s'exécute au niveau de privilège 3, et le handler doit s'exécuter au niveau de privilège 0.

Interruption/Exception via un guichet 386 d'interruption ou d'exception

Le transfert vers le handler utilise le même mécanisme que celui décrit au Chapitre 6 pour passer par des guichets de trappe ou d'interruption. Toutefois, les registres GS, FS, DS et ES des programmes virtuels 8086 sont empilés dans la nouvelle pile (de niveau 0) avant que le traitement classique des interruptions ne commence. Pour conserver l'alignement de la pile, ils sont empilés comme des valeurs 32 bits avec le registre segment dans les 16 bits de poids faible et les 16 bits de poids fort indéfinis. Ces registres segments sont ensuite chargés avec des sélecteurs nuls avant d'entrer dans le handler d'interruption ou d'exception.

Les quatre registres segments GS, FS, DS et ES sont sauvegardés dans la pile puis chargés avec des sélecteurs nuls avant d'entrer dans le handler d'interruption (en mode protégé). Ainsi, le handler peut sauver et récupérer les registres segments en tant que valeurs valides dans le mode protégé, quel que soit le mode du programme interrompu. Si ce dernier est en mode protégé, les registres segments contiendront des sélecteurs valides qui peuvent être empilés au début du handler puis dépilés avant l'IRET. Cependant, si le programme interrompu s'exécutait en mode virtuel 8086, les registres segments pourraient contenir des valeurs qui ne sont pas des sélecteurs valides dans le mode protégé. En sauvegardant dans la pile les valeurs des segments virtuels 8086, et en chargeant les registres segments (GS, FS, DS et ES) avec des sélecteurs nuls, les registres segments contiendront des valeurs valides en mode protégé au début du handler.

La Figure 9.5 illustre les manipulations de la pile lors d'une exception ou d'une interruption à travers un guichet 386 d'interruption ou d'exception et à partir d'un programme en mode virtuel 8086 au niveau 3 vers un handler d'interruption en mode protégé au niveau 0. Les registres SS et SP adressent la pile externe. SS contient un registre segment 8086 et non un sélecteur de segment en mode protégé. A gauche sur la figure est représentée la nouvelle pile interne. Les registres GS, FS, DS et ES sont empilés dans la nouvelle pile. Puis le pointeur de la pile externe est sauvé en empilant SS et ESP. Ensuite, l'ancienne valeur de EFLAGS est empilée (VM égal à 1). Puis l'adresse de retour est sauvée en empilant CS et EIP. En fin de compte, c'est un code d'erreur qui est empilé.

Au début du handler, SS adresse le segment de pile interne, et ESP pointe sur le code d'erreur. Puisque le handler travaille en mode protégé, le registre SS contient un sélecteur pour la pile interne, et les registres des descripteurs cachés de SS sont chargés à partir du descripteur de ce segment. De même, le nouveau sélecteur de CS adresse un segment code en mode protégé, et les registres des descripteurs cachés de CS sont chargés à partir du descripteur de ce segment.

Le handler d'exception ou d'interruption peut examiner l'image de EFLAGS sauvegardée dans la pile pour déterminer le mode de fonctionnement du programme interrompu. Si le bit VM de cette image est égal

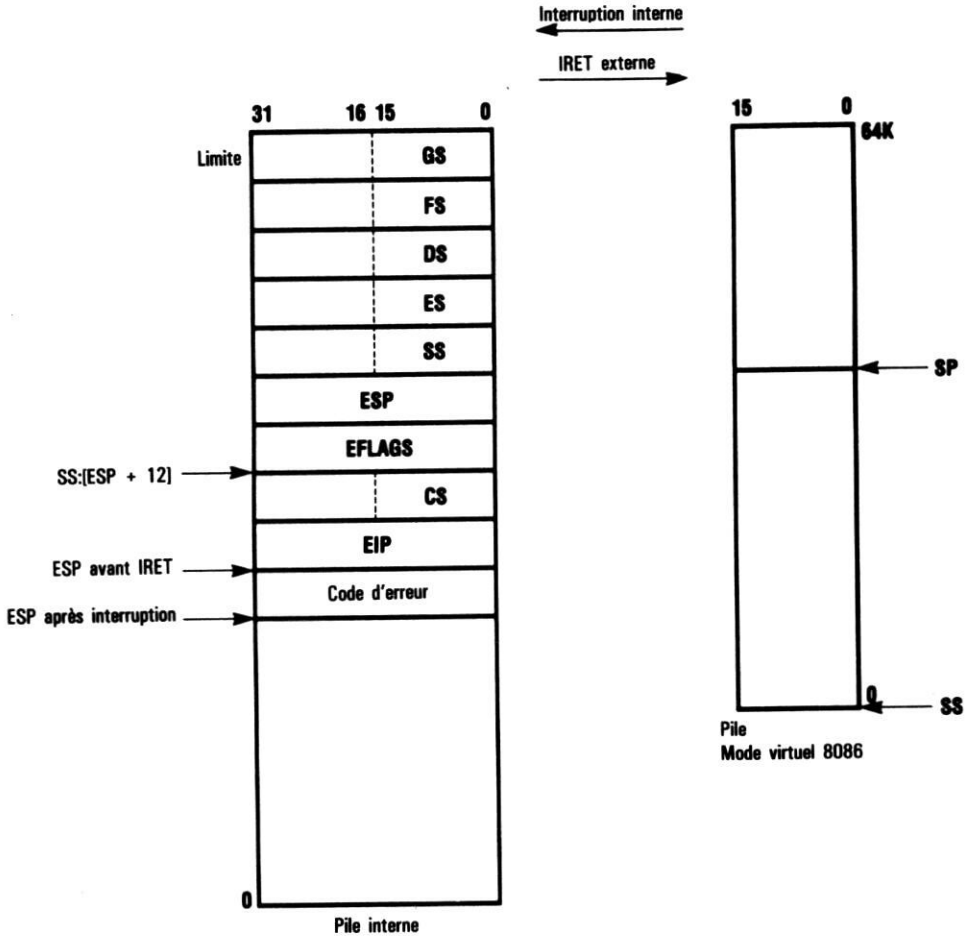


Figure 9.5 : *Commutation de pile à partir du mode virtuel 8086 via un guichet d'interruption ou de trappe.*

à 1, c'est que le programme interrompu s'exécutait en mode virtuel 8086. Sinon, il s'exécutait en mode protégé. Le registre EFLAGS est toujours situé au même déplacement à partir du registre ESP, quel que soit le mode de fonctionnement du programme interrompu. L'image de EFLAGS est située au déplacement 12 à partir de ESP si le code d'erreur a été empilé (voir la Figure 9.5). Sinon, EFLAGS est situé au déplacement 8.

```
/* Routine IntTrapGate complète, y compris le mode virtuel */
/* 8086. La version originale est au Chapitre 6            */
IntTrapGate'(Selector, Offset, ClearIF)
   SelectorType   Selector;
   int   Offset,
         ClearIF;   /* Si 1, efface IF avant le handler */
                    /* Si 0, ne modifie pas */
{
   SegAttributes  Attributes;
   SelectorType   GSelector;
   int   Base, Limit, GOffset;

/* Test du sélecteur */
if ( (Selector & 0FFFCh) == 0)   /* Nul */
   SegmentException($GP, 0);

/* Lecture et test du descripteur */
ReadDescriptor(Selector,&Attributes,&Base,&Limit,&GSelector,&GOffset);

if (Attributes.DType == 0)
   /* Ne peut pas être un segment Système */
                          /* ni un guichet */
   SegmentException($GP, Selector);

/* Interruption vers un segment présent, exécutable,*/
/* non conforme de niveau interne OK, mais demande  */
/*  commutation vers pile interne et CPL interne.   */
```

Listing 9.2 : *Sous-programme IntTrapGate'().*

```
if ( (Attributes.Type>=8) && (Attributes.Type<=11)
   &&(Attributes.DPL < CPL) && (Attributes.P == 1) )
      InnerStack(Attributes.DPL, 0, Selector);

/* Si provient du mode  virtuel 8086, transition vers le */
/* niveau 0 */
if ( (CPL != 0) && (EFLAGS.VM == 1) )
   SegmentException($GP, Selector);

/* Appel classique pour achever le chargement */
/* du descripteur de CS. */
Selector.RPL = 0;
   /* Ignore RPL dans la lecture du sélecteur      */
CSDescriptorLoad(Selector, Attributes, Base, Limit, $GP);

/* Vérifie si la destination est comprise dans les limites */
/* du segment*/
if (Offset > CS.Limit)
   SegmentException($GP, 0);

/* On arrive ici seulement si tous les tests de protection */
/* réussissent */
/* Empile EFLAGS et le pointeur de retour, puis modifie    */
/* EFLAGS et charge */
/* le sélecteur de CS et EIP.                              */
ESP = ESP-4;
AccessVirtual(SS, ESP, 4, 1  /* Ecriture */, $SS, &EFLAGS);
ESP = ESP-4;
/* Empile 4 octets avec le sélecteur de CS dans les */
/* 2 octets de poids faible.                        */
AccessVirtual(SS, ESP, 4, 1  /* Ecriture */, $SS, $CS.Selector);
ESP = ESP-4;
AccessVirtual(SS, ESP, 4, 1  /* Ecriture */, $SS, &EIP);
EFLAGS.VM = 0;   /* Handler en mode protégé          */
```

Listing 9.2 : *Sous-programme IntTrapGate'() (suite).*

```
EFLAGS.TF = 0;    /* Désactive le mode pas à pas      */
                  /* (voir le Chapitre 8) */
EFLAGS.NT = 0;    /* Interruption non traitée par une*/
                  /* tâche imbriquée */
if (ClearIF == 1)
   EFLAGS.IF = 0;

CS.Selector      = Selector;
CS.Selector.RPL = CPL;
EIP              = Offset;
} /* fin de IntTrapGate' */
```

Listing 9.2 : *Sous-programme IntTrapGate'() (fin).*

Les Listings 9.2 et 9.3 illustrent les changements opérés sur le mécanisme d'interruption (voir le Chapitre 6) pour adapter ce mécanisme au mode virtuel 8086. Le Listing 9.2 contient la version modifiée de la routine IntTrapGate() du Chapitre 6. Les parties qui ont été modifiées sont en gris. La première modification consiste à tester la valeur du CPL juste après le test habituel du passage au niveau interne. Le handler doit s'exécuter au niveau 0 si le programme interrompu travaillait en mode virtuel 8086. A l'endroit où le test a été inséré, CPL peut avoir été modifié lors de l'appel de la routine InnerStack(). C'est à ce moment qu'on peut tester le CPL du handler. S'il est différent de 0, une exception $GP sera générée avec le sélecteur de CS du handler comme code d'erreur.

L'autre modification consiste à positionner le bit VM de EFLAGS à 0 avant d'entrer dans le handler. Ceci permet au handler de travailler en mode protégé, même si le programme interrompu s'exécutait en mode virtuel 8086.

Le Listing 9.3 montre le Sous-programme InnerStack() modifié pour adapter le mécanisme d'interruption au mode virtuel 8086. Les changements sont en gris. Dans cette routine, les registres GS, FS, DS et ES sont empilés après que la nouvelle pile a été adressée par les registres SS et

ESP, mais avant que l'ancien pointeur de pile soit empilé. Quand les registres segments sont empilés, ils sont chargés avec des sélecteurs nuls.

```
/* Commutation de piles pour des changements inter-niveaux */
/* Modification de la version originale du Chapitre 5 pour */
/* inclure les modifications du mode virtuel 8086          */
InnerStack'(NewCPL, DwordCount, NewCSSelector)
   int   NewCPL,
/* Commutation de piles pour ce niveau de privilège */
         DwordCount;
/* Nombre de paramètres dmots à copier */
              /* dans la nouvelle pile */
         SelectorType NewCSSelector;
/* sélecteur de CS obligatoire */
         /* pour code d'erreur */
{
   SelectorType NewSSSelector;
   SegmentRegister OldSS;
   int TSSOffset, NewESP, tempESP, OldCPL, Dword1;

/* Lecture du nouveau SS et de ESP à partir de TSS.*/
TSSOffset = NewCPL*8 + 4;
if ( (TSSOffset + 7) > TR.Limit)
   SegmentException($TS, NewCSSelector);
AccessLinear(TR.Base+TSSOffset, 0  /* PL 0 */,
             0  /* Lecture */, &NewESP);
AccessLinear(TR.Base+TSSOffset+4, 0  /* PL 0 */,
             0  /* Lecture */, &NewSSSelector);

/* Sauve les anciens SS et CPL pour utilisation future */
OldSS = SS;
OldCPL = CPL;

/* Charge SS avec le sélecteur de la nouvelle pile après */
/* passage au nouveau CPL et chargement du descripteur   */
```

Listing 9.3 : *Sous-programme InnerStack'().*

```
/* dans les registres cachés.                                  */
CPL = NewCPL;
SRegLoad(SS, NewSSSelector, $TS);
/* retour si le chargement de SS a réussi */
SS.Selector = NewSSSelector;

/* Empile les registres segments des données et charge    */
/* avec des sélecteurs nuls, si on vient du mode virtuel */
/* 8086 */

if (EFLAGS.VM == 1)
   {NewESP = NewESP-4;
   AccessVirtual(SS, NewESP, 4, 1 /* Ecriture */, $SS,
                 &GS.Selector);
   GS.Selector = 0; GS.Attributes.P = 0;
   NewESP = NewESP-4;
   AccessVirtual(SS, NewESP, 4, 1 /* Ecriture */, $SS,
                 &FS.Selector);
   FS.Selector = 0; FS.Attributes.P = 0;
   NewESP = NewESP-4;
   AccessVirtual(SS, NewESP, 4, 1 /* Ecriture */, $SS,
                 &DS.Selector);
   DS.Selector = 0; DS.Attributes.P = 0;
   NewESP = NewESP-4;
   AccessVirtual(SS, NewESP, 4, 1 /* Ecriture */, $SS,
                 &ES.Selector);
   ES.Selector = 0; ES.Attributes.P = 0;
   }

/* Empile les anciens SS et ESP */
NewESP = NewESP - 4;
AccessVirtual(SS, NewESP, 4, 1  /* Ecriture */, $SS,
              &OldSS.Selector);
NewESP = NewESP - 4;
AccessVirtual(SS, NewESP, 4, 1  /* Ecriture */, $SS, &ESP);
```

Listing 9.3 : *Sous-programme InnerStack'() (suite).*

```
/* Copie les paramètres. L'écriture dans la nouvelle pile  */
/* utilise le nouveau CPL. La lecture dans l'ancienne pile */
/* utilise l'ancien  CPL */
/* L'ordre de copie des paramètres n'est pas important,    */
/* mais on emploie l'ordre initial de la pile externe. Avec*/
/* cet ordre, nous allons sortir de la boucle avec NewESP  */ /*
pointant sur le sommet de la nouvelle pile */
tempESP = ESP + DwordCount*4;
for (i=1; i<=DwordCount; i++){
   CPL = OldCPL;
   tempESP = tempESP - 4;
   AccessVirtual(OldSS, tempESP, 4, 0  /* Lecture */, $SS,
                 &Dword1);
   CPL = NewCPL;
   NewESP = NewESP - 4;
   AccessVirtual(SS, NewESP, 4, 1  /* Ecriture */, $SS,
                 &Dword1);
   }

ESP = NewESP;
   /* Sortie de boucle avec NewESP pointant sur le */
                        /* nouveau sommet de pile */
} /* fin de InnerStack' */
```

Listing 9.3 : *Sous-programme InnerStack'() (fin).*

IRET avec NT = 0 vers un programme en mode virtuel 8086

L'instruction IRET, si elle est exécutée avec NT = 0 dans le registre EFLAGS, dépilera les informations de retour vers le programme interrompu dans la même tâche. Si IRET retourne à un programme en mode virtuel 8086, les registres ES, DS, FS et GS sont dépilés après que l'IRET classique s'est achevé. Ces valeurs, ainsi que les autres valeurs chargées

dans SS et CS, sont chargées comme des valeurs 8086, et non comme des sélecteurs en mode protégé.

La Figure 9.5 montre aussi la pile avant et après un IRET au niveau interne vers un programme en mode virtuel 8086 dans la même tâche. La pile courante (niveau interne) est à gauche. Elle est adressée comme un segment en mode protégé avec un sélecteur dans SS. Avant d'exécuter l'instruction IRET, le handler doit ajuster la pile en dépilant le code d'erreur pour faire pointer ESP sur le pointeur de retour. L'instruction IRET dépile EIP, CS et EFLAGS de la pile interne. Si VM = 1 dans la valeur de EFLAGS dépilée, et que IRET s'exécute au niveau de privilège 0, le retour s'effectue vers un programme en mode virtuel 8086 et CS est chargé comme un registre segment 8086. Puisqu'un retour vers le mode virtuel 8086 équivaut à un changement de niveau de privilège (vers le niveau 3) et exige un changement de pile, le pointeur de pile externe est récupéré par dépilage de ESP et SS en tant que segments 8086. Finalement, les registres ES, DS, FS et GS sont dépilés comme des segments 8086 pour achever l'IRET.

Le Listing 9.4 décrit IRET avec les modifications ajoutées qui permettent de retourner à un programme en mode virtuel 8086. Si IRET s'exécute au niveau 0 et que le bit VM est à 1 dans la nouvelle image de EFLAGS qui vient d'être dépilée, le retour s'effectue vers le mode virtuel 8086. Au lieu que CS et SS soient chargés comme des segments en mode protégé, seuls les registres sélecteurs de segments visibles sont chargés. De plus, les registres ES, DS, FS et GS sont dépilés comme des registres segments 8086.

```
/* Description de l'instruction IRET, modifiée pour  */
/* inclure le retour à un programme en mode virtuel. */

IRET'()
{
```

Listing 9.4 : *Description de IRET'.*

```
   SelectorType Selector, GSelector;
   SegAttributes Attributes;
   EFLAGSType NewEFLAGS;
int Base, Limit, Offset, GOffset, NewESP, NewSS;

/* Test du bit NT si on fait un retour de tâche */
if (EFLAGS.NT == 1)
   {/* Retour vers une tâche identifiée par le champ */
    /* de lien du TSS courant */
   AccessLinear(TR.Base, 2, 0  /* PL 0 */, 0  /* Lecture */,
                &Selector);
   TaskGate(Selector, -1  /* Libre */);
        /* voir le Chapitre 5.*/
   }

else {
/* Sinon, dépile les informations de retour d'interruption */
   AccessVirtual(SS, ESP, 4, 0  /*Lecture */, $SS, &Offset);
   ESP = ESP + 4;
   AccessVirtual(SS, ESP, 4, 0  /* Lecture */, $SS,
                &Selector);
   ESP = ESP + 4;
   AccessVirtual(SS, ESP, 4, 0  /* Lecture */, $SS,
                &NewEFLAGS);
   ESP = ESP + 4;

if ( (NewEFLAGS.VM == 1) && (CPL == 0) )
   {/* Retour à un programme en mode virtuel 8086     */
    /* Dépile l'ancien pointeur de pile, charge SS:ESP */
   AccessVirtual(SS, ESP, 4, 0  /* Lecture */, $SS, &NewESP);
   ESP = ESP - 4;
   AccessVirtual(SS, ESP, 4, 0  /* Lecture */, $SS, &NewSS);
   ESP = ESP - 4;

/* Dépile les valeurs des segments ES, DS, FS et GS (8086) */
```

Listing 9.4 : *Description de IRET' (suite).*

```
   AccessVirtual(SS, ESP, 4, 0  /* Lecture */, $SS,
                 &ES.Selector);
   ESP = ESP - 4;
   AccessVirtual(SS, ESP, 4, 0  /* Lecture */, $SS,
                 &DS.Selector);
   ESP = ESP - 4;
   AccessVirtual(SS, ESP, 4, 0  /* Lecture */, $SS,
                 &FS.Selector);
   ESP = ESP - 4;
   AccessVirtual(SS, ESP, 4, 0  * Lecture */, $SS,
                 &GS.Selector);

   /* Pointe sur la pile de niveau externe en mode */
   /* virtuel 8086 */
   ESP = NewESP;
   SS.Selector = NewwSS;      /* segment 8086 */
   CPL = 3;
   /* Le programme en mode virtuel 8086 se poursuit */
                                    /* au niveau 3 */
   }
else {
   /* Tests de sélecteur pour IRET en mode protégé */
   if ( ((Selector & 0FFFCh) == 0) || (Selector.RPL < CPL))
      SegmentException($GP, 0);
   if (Selector.RPL > CPL) {
      /* IRET inter-niveaux si RPL > CPL       */
      /* Appel à Sous-programme pour récupérer */
      /* la pile externe à partir du pointeur  */
      /* SS:ESP qui est au sommet de la pile   */
      /* interne.                              */
      OuterStack(Selector.RPL, 0);
      }

   /* Lecture et test du descripteur de CS */
```

Listing 9.4 : *Description de IRET' (suite).*

```
   ReadDescriptor(Selector,&Attributes,&Base,&Limit,&GSelector,
                &GOffset);
   if (Attributes.DType == 0)
   /* Ne peut pas être un segment Système */
                        /* ni un guichet  */
      SegmentException($GP, Selector);

   /* Appel classique pour compléter le chargement */
   /* du descripteur de CS.*/
   CSDescriptorLoad(Selector, Attributes, Base, Limit, $GP);

/* Vérifie si la destination est comprise dans la  */
/* limite du segment */
   if (Offset > CS.Limit)
      SegmentException($GP, 0);
} /* fin de IRET en mode protégé.*/

/* On arrive ici seulement si tous les tests de protection */
/* sont OK. On complète la partie visible de l'instruction */
/* en chargeant EFLAGS, le sélecteur CS, et EIP.           */
   EFLAGS      = NewEFLAGS;
   CS.Selector = Selector;
   EIP         = Offset;
   } /* fin de NT=0 */
} /* fin de IRET' */
```

Listing 9.4 : *Description de IRET' (fin).*

Transferts vers des tâches différentes

Une interruption ou une exception reçues en mode virtuel entraîneront un transfert vers une tâche différente si l'élément de l'IDT contient un guichet d'appel. Ainsi, une instruction JMP, CALL ou IRET (avec NT = 1) exécutée en mode protégé peut effectuer une commutation de tâches. Elle

peut s'effectuer d'un programme en mode virtuel 8086 vers un programme en mode protégé ou vers un autre programme en mode virtuel 8086. De la même façon, une tâche en mode protégé peut commuter vers une autre tâche en mode protégé ou une tâche en mode virtuel 8086. Si on a une commutation de tâches depuis une tâche en mode virtuel 8086, l'image de EFLAGS sauvée dans l'ancien TSS sera telle que son bit VM sera égal à 1. Une commutation de tâches vers une tâche dont le bit VM de l'image de EFLAGS dans le nouveau TSS est à 1 effectuera une commutation vers une tâche en mode virtuel 8086.

Une commutation d'une tâche ou vers une tâche en mode virtuel 8086 est identique à une commutation vers une tâche en mode protégé, sauf pour le chargement des registres segments. Une commutation vers une tâche en mode virtuel 8086 chargera les registres segments (CS, SS, DS, ES, FS et GS) comme des registres segments 8086, et non comme des sélecteurs en mode protégé. Sinon, la commutation de tâches est identique, y compris le chargement du LDTR avec un sélecteur en mode protégé pour le segment de la LDT de la tâche.

Le Listing 9.5 contient la routine TaskSwitch() introduite au Chapitre 5, avec les modifications apportées par les commutations de tâches vers des tâches en mode virtuel 8086. La première modification consiste à fixer le niveau de privilège de la nouvelle tâche à 3 si on est en mode virtuel 8086, au lieu de lui donner la valeur du champ RPL du sélecteur de CS. Une nouvelle LDT est adressée même si la tâche s'exécute en mode virtuel 8086. Ceci est nécessaire puisque la tâche peut commuter hors du mode virtuel 8086, par exemple, pour traiter une interruption ou une exception via un guichet de trappe ou d'interruption. Ces handlers peuvent adresser des segments associés à la LDT.

L'autre modification de la routine TaskSwitch() consiste à ne pas charger les registres cachés des descripteurs si la nouvelle tâche est en mode virtuel 8086. Au contraire, seuls les sélecteurs de segments sont nécessaires et sont chargés par la routine AccessTSSState().

```
/* Routine TaskSwitch du Chapitre 5 modifiée pour inclure */
/* les tâches du mode virtuel-8086.                       */
TaskSwitch'(Selector, Attributes, Base, Limit, Linkage)
   SelectorType Selector;
   SegAttributes Attributes;
   int    Base, Limit,
          Linkage;       /* 0 : Pas de lien, 1: lien */
                         /* -1 : Libre */
{
if (Limit < 103)
   SegmentException($TS, Selector);

/* Sauve le contexte dans le TSS de l'ancienne tâche       */
AccessTSSState(1  /* Ecriture */);

/* TR et les registres Cachés du nouveau TSS */
TR.Base       = Base;
TR.Limit      = Limit;
TR.Attributes = Attributes;
OldTSS        = TR.Selector;  /* Sauve le sélecteur du TSS */
                              /* de l'ancienne tâche */
TR.Selector   = Selector;

/* Charge le contexte à partir du TSS de la nouvelle tâche */
AccessTSSState(0  /* Lecture */);

/* Traite les différents liens */
if (Linkage == 1  /* Lien */) {
/* Sauve le sélecteur de l'ancien TSS dans le nouveau TSS. */
/* Lève le bit NT. Laisse le descripteur de l'ancien TSS   */
/* occupé */
   AccessLinear(TR.Base, 2, 0  /* Niveau 0 */,
                1  /* Ecriture */, &OldTSS);
   EFLAGS.NT = 1;
   SetTSSBusy(Selector, 1);
```

Listing 9.5 : *Sous-programme TaskSwitch'().*

```
   /* descripteur du nouveau TSS occupé */
   } /* fin de Link */
else if (Linkage == -1   /* Libre */){
   SetTSSBusy(OldTSS, 0);
   /* descripteur de l'ancien TSS libre */
   } /* fin de Unlink */
else if (Linkage == 0 /* pas de lien */) {
   SetTSSBusy(OldTSS, 0);
   /* descripteur de l'ancien TSS libre */
   SetTSSBusy(Selector, 1);
   /* descripteur du nouveau TSS occupé */
   } /* fin de Nolink */

CR0.TS = 1;   /* Positionne le bit TS */

if (EFLAGS.VM == 1)
   CPL = 3;   /* Mode virtuel 8086 au niveau 3 */
   else CPL = CS.Selector.RPL;/* CPL à partir du champ RPL */
                              /* du sélecteur CS */

/* Contexte visible récupéré. Charge les descripteurs      */
/* dans les registres cachés pour LDTR et les registres    */
/* segment. Rend tous les descripteurs non valides.        */
/* L'Attribut Présent est ignoré dans le mode virtuel 8086.*/
LDTR.Attributes.Present = 0;
CS.Attributes.Present = 0;
SS.Attributes.Present = 0;
DS.Attributes.Present = 0;
ES.Attributes.Present = 0;
FS.Attributes.Present = 0;
GS.Attributes.Present = 0;

/* Doit charger d'abord LDTR */
if (LDTR.Selector.TI == 1   /* dans LDT */)
   SegmentException($TS, LDTR.Selector);
```

Listing 9.5 : *Sous-programme TaskSwitch'() (suite).*

```
if ((LDTR.Selector & 0FFFCh) = 0   /* Nul */) {
   /* OK si LDTR est nul.*/
   LDTR.Attributes.P = 0;
   }
else {/* Lecture et test du descripteur si le sélecteur */
      /* n'est pas nul */
   ReadDescriptor(LDTR.Selector,&Attributes,&Base,&Limit,
                  &GSelector,&GOffset);
   if ( (Attributes.DType == 1)
      || (Attributes.Type != 2   /* LDT */)
      || (Attributes.Present == 0) )
      SegmentException($TS, LDTR.Selector);
/* Chargement des registres cachés de LDTR si tous */
/* les tests sont OK */
   SetAccessed(LDTR.Selector);
   LDTR.Attributes   = Attributes;
   LDTR.Base         = Base;
   LDTR.Limit        = Limit;
   } /* fin de LDTR. Sélecteur non Nul */

if (EFLAGS.VM == 0) {
/* Chargement des registres cachés restants du descripteur */
/* si on se trouve en mode protégé. Le chargement de CS est*/
/* identique à un saut à travers un guichet d'appel.       */
JumpGate(CS.Selector, EIP, $TS);

SRegLoad(SS, SS.Selector, $TS);
SRegLoad(DS, DS.Selector, $TS);
SRegLoad(ES, ES.Selector, $TS);
SRegLoad(FS, FS.Selector, $TS);
SRegLoad(GS, GS.Selector, $TS);
}
/* fin des chargements en mode protégé de registres cachés */
/* du descripteur.*/
```

Listing 9.5 : *Sous-programme TaskSwitch'() (suite).*

```
DR7.LE0 = 0;
DR7.LE1 = 0;
DR7.LE2 = 0;
DR7.LE3 = 0;

} /* fin de TaskSwitch' */
```

Listing 9.5 : *Sous-programme TaskSwitch'() (fin).*

Mode réel

Dans les paragraphes suivants, nous allons décrire les caractéristiques spécifiques aux programmes qui s'exécutent en mode réel. Tout d'abord, nous allons décrire l'état de la machine au moment de l'initialisation. Le 80386 est en mode réel à chaque mise sous circuit. Un court programme de lancement (voir le Chapitre 7) permet d'initialiser les segments système et les registres de contrôle avant d'entrer en mode protégé. Puis nous expliquerons les méthodes qui permettent de passer du mode réel au mode protégé et réciproquement. Le mode réel présente aussi quelques différences pour le traitement des interruptions, et utilise un mécanisme différent.

Etat du 80386 à l'initialisation

Lorsque le 80386 est initialisé, le processeur est en mode réel, avec les registres initialisés par les valeurs du Tableau 9.3. Ceux qui ne sont pas dans le Tableau 9.3 sont indéfinis lors de l'initialisation et c'est au logiciel de les initialiser. La *Data Sheet 80386* d'Intel explique de façon précise le processus d'initialisation du processeur.

REGISTRE	VALEUR INITIALE
EAX	Signature d'autotest
ECX	Indéfini
EDX	ID composant/version
EBX	Indéfini
ESP	Indéfini
EBP	Indéfini
ESI	Indéfini
EDI	Indéfini
EFLAGS	00000002h
EIP	0FFF0h
ES	0
CS	F000h
SS	0
DS	0
FS	0
GS	0
IDTR	Base = 0, Limite = 3FFh
CR0	0 ou 10h
DR7	0

Tableau 9.3 : *Valeurs initiales des registres.*

Le registre EAX contient une signature d'autotest, si le processus d'initialisation comprend un autotest. Cette signature est égale à 0 si tout s'est bien passé. La valeur de EAX est indéfinie s'il n'y a pas eu d'autotest. Reportez-vous à la *Data Sheet 80386* pour savoir comment effectuer cet autotest et ce qu'il teste.

Le registre EDX est initialisé avec un ID (numéro d'identification) de version et de composant. L'ID de composant est placé dans le registre DH et est égal à 3 pour le 80386. Les futurs processeurs de la famille 80386 utiliseront des ID différents. Les anciens processeurs de la famille 86 ne possèdent pas cette caractéristique. Au contraire, ils initialisent DX avec une valeur indéfinie (aléatoire). Le ID de version est placé dans le registre DL. Il existe un numéro unique pour chaque modification importante du processeur.

Les registres CS et EIP sont initialisés à 0F000h et 0FFF0h. Donc, jusqu'au premier transfert de contrôle intersegment, toute référence au registre CS donne une adresse physique dont les 12 bits d'adresses de poids fort (A31 à A20) sont égaux à 1. Cela signifie qu'après une initialisation, la première instruction est traitée à l'adresse physique 0FFFFFFF0h, et qu'on peut employer des sauts intrasegments pour exécuter du code situé aux 64 plus hauts Ko de l'espace adressable physique (adresses 0FFFF0000h à 0FFFFFFFFh). Si le processeur est encore en mode réel lors du premier transfert intersegment (une instruction intersegment CALL, JMP, RET ou IRET, ou une interruption ou une exception), le transfert rechargera le registre CS et mettra à 0 les 12 bits d'adresses de poids fort. La plupart des systèmes 80386 comprennent une ROM de lancement au sommet de la mémoire, qui contient le programme de lancement ainsi que certaines parties du système d'exploitation.

Les registres segments données SS, DS, ES, FS et GS sont initialisés à 0 pour adresser des données situées dans les 64 plus bas Ko de la mémoire (adresses 0 à 0FFFFh).

Le registre EFLAGS est initialisé à 2, si bien que les interruptions (IF = 0) et le mode pas à pas (TF = 0) sont désactivés. IDTR est initialisé pour avoir une adresse de base nulle et une limite égale à 3FFh. Ceci assure la compatibilité avec le 8086, qui range sa table d'interruptions à l'adresse 0. Cette table contient 256 éléments de 4 octets. CR0 est initialisé à 0 ou à 10h, avec tous ses bits (sauf ET) égaux à 0. ET est mis à 1 si le 80386 a détecté un 80387 lors de l'initialisation. Toutefois, s'il a affaire à un 80287 ou s'il ne détecte aucun coprocesseur, le 80386 met le bit ET à 0. La valeur initiale de CR0 est telle que PE = 0 (protection désactivée), et

que PG = 0 (pagination désactivée), si bien que le processeur se trouve en mode réel. DR7 est initialisé à 0 pour désactiver tous les points d'arrêt.

Le logiciel doit initialiser la pile en chargeant SS et SP avant toute instruction (ou interruption) qui utilise la pile. De plus, avant de valider les interruptions, la table des interruptions doit être initialisée. Bien que le 80386 soit initialisé avec les interruptions désactivées, les NMI sont toujours valides et peuvent survenir immédiatement après l'initialisation, à moins qu'un circuit externe ne les masque.

Transferts du mode réel au mode protégé

Après l'initialisation, on peut utiliser un programme de lancement pour initialiser les segments système et les registres de contrôle et pour mettre à 1 le bit PE de CR0 (mode protégé). Reportez-vous au Chapitre 7 pour avoir un exemple complet de transfert du mode réel au mode protégé. Immédiatement après avoir mis à 1 le bit PE, on doit effectuer un saut intrasegment pour vider la file d'exécution des instructions qui ont été traitées et décodées en mode réel. En général, ce saut s'effectue à l'instruction suivante.

Après être passé en mode protégé, on doit recharger les registres segments avec des sélecteurs valides en mode protégé (ou des sélecteurs nuls). On peut "recharger" CS avec un transfert de contrôle intersegment (une instruction JMP ou CALL). On doit charger le registre TR avec un sélecteur de segment système du type TSS disponible, et LDTR avec un sélecteur nul ou un segment système du type LDT.

On doit charger GDTR et IDTR avec les adresses de base et de limite de la GDT et de l'IDT. Au contraire des registres segments et des registres LDTR et TR, on peut les charger avant d'entrer en mode protégé. L'IDT doit suivre le format employé par les interruptions en mode protégé (voir le Chapitre 6).

Transferts du mode protégé au mode réel

Il existe deux possibilités pour effectuer un transfert du mode protégé au mode réel :

1 Utiliser des circuits externes pour réinitialiser le 80386.

2 Charger CR0 avec une valeur qui place 0 dans le bit PE.

La première méthode est décrite dans les paragraphes précédents. La seconde, qui est une méthode logicielle, doit obéir aux règles suivantes :

1 Le transfert doit s'effectuer à partir d'un programme qui s'exécute dans un segment de code de taille 64 Ko exactement (limite = 0FFFFh).

2 Avant de charger CR0 pour mettre le bit PE à 0, on doit charger les registres segments SS, DS, ES, FS et GS avec des sélecteurs d'un segment données du type présent, lecture/écriture, expand-up, et DPL = 0, qui soit exactement de taille 64 Ko.

3 On doit charger CR0 avec une valeur qui mette le bit PE à 0.

4 Finalement, on doit exécuter une instruction JMP intersegment directe immédiatement après le chargement de CR0. Ce JMP videra la file d'attente du processeur et effectuera le transfert au programme en mode réel dans le méga-octet inférieur de la mémoire physique.

Lors de ce changement de mode, on doit désactiver les interruptions (et NMI) jusqu'au chargement de IDTR qui pointera sur une table d'interruptions conforme au mode réel. Pour retourner au mode réel, on peut aussi utiliser le mode virtuel 8086 pour exécuter des programmes 8086.

Avant d'effectuer le transfert du mode protégé au mode réel, on doit désactiver la pagination. On effectuera de préférence cette opération dans un segment code qui a la même adresse dans les espaces linéaires et physiques et en employant des données possédant les mêmes caractéristiques.

Dans ce contexte, on peut désactiver la pagination en chargeant CR0 avec une valeur qui mette 0 dans le bit PG, en exécutant un saut intrasegment pour vider la file d'attente du processeur, et pour finir en chargeant CR3 pour vider le cache de pagination.

Traitement des interruptions en mode réel

C'est un mécanisme différent de celui du mode protégé (voir le Chapitre 6) qui va traiter toute interruption ou exception qui survient lorsque le processeur travaille en mode réel. Ce mécanisme est identique à celui du 8086. L'IDT suit un format différent en mode réel, et les interruptions et les exceptions sont traitées en employant une méthode quelque peu similaire à celle des guichets d'interruption (voir le Chapitre 6).

En mode réel, la table des descripteurs d'interruptions est un tableau d'éléments de 4 octets adressé par le registre IDTR. Chaque élément contient un pointeur 32 bits sur la première instruction du handler d'interruption. Chaque pointeur contient un déplacement dans les 16 bits de poids faible et une valeur de registre segment dans les 16 bits de poids fort (voir la Figure 9.6). Ce pointeur donne les valeurs à charger dans IP et CS pour commencer l'exécution du handler en mode réel.

En mode réel, les interruptions et les exceptions sont traitées en utilisant une version 16 bits de la sémantique (voir le Chapitre 3) de l'instruction INT. Tout d'abord, le registre 16 bits FLAGS est empilé, suivi par les registres CS et IP. Puis le processeur consulte la table des interruptions pour trouver le pointeur à charger dans CS et IP qui lui permet de commencer l'exécution du handler.

Finalement, les bits TF et IF sont mis à 0 pour désactiver le mode pas à pas et les interruptions au début du handler. Si une exception survient lors du mode réel, aucun code d'erreur n'est empilé.

On peut utiliser l'instruction IRET du Chapitre 3 pour retourner du traitement des exceptions et des interruptions en mode réel. La sémantique de IRET sera alors celle du Chapitre 3 et non celle du Chapitre 6 pour la

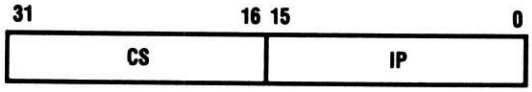


Figure 9.6 : *Format d'un élément de la table des interruptions en mode réel.*

version de IRET en mode protégé. IRET dépile IP, CS et FLAGS pour poursuivre l'exécution du programme interrompu en mode réel.

La Figure 9.7 montre les valeurs empilées lors d'une interruption en mode réel, et dépilées lors d'un IRET en mode réel. La pile est identique à la pile 16 bits du 8086. Le registre FLAGS est empilé en premier à l'adresse la plus haute, puis c'est au tour des registres CS et IP. Au début du handler d'interruption ou d'exception, SP pointe sur l'image de IP empilée. Aucun code d'erreur n'est empilé lors du traitement des exceptions en mode réel. Le prochain IRET dépilera IP, CS et FLAGS pour poursuivre l'exécution, avec SP à la valeur qu'il avait avant l'interruption ou l'exception.

Le Listing 9.6 décrit le traitement des interruptions ou des exceptions en mode réel. Notez l'emploi de la routine Access8086Virtual pour accéder à la mémoire dans la pile en mode réel. Si l'élément de la table des interruptions pour l'interruption ou l'exception donnée est au-delà de la limite de IDTR, il y aura génération d'une exception du type protection générale. Il peut se produire des doubles fautes si la limite de IDTR est trop petite pour accéder à l'élément de la table des interruptions. Si la limite de IDTR est inférieure à 35, le processeur ne pourra pas accéder à l'élément "double faute" et entrera en mode shutdown (voir le Chapitre 6).

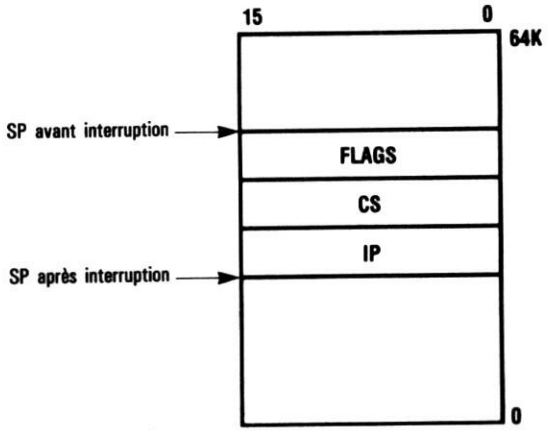


Figure 9.7 : *Utilisation de la pile par les exceptions et les interruptions en mode réel, et par IRET.*

EXECUTION DE PROGRAMMES 80286 EN MODE PROTEGE

Les programmes qui s'exécutent en mode protégé sur le 80286 peuvent aussi s'exécuter sur le 80386. Vous pouvez lancer des programmes d'application 80286 et des systèmes d'exploitation 80286 sans modification. Deux options sont disponibles :

```
/* Traitement d'une interruption ou d'une exception */
/* en mode réel */
InterruptREAL(VecNumber)

int     VecNumber;    /* Numéro du Vecteur */

{
/* Empile 16 bits FLAGS, CS, et IP */
SP = SP-2;
Access8086Virtual(SS, SP, 2, 1  /* Ecriture */, $SS, &FLAGS);
SP = SP-2;
Access8086Virtual(SS, SP, 2, 1  /* Ecriture */, $SS,
                  &CS.Selector);
SP = SP-2;
Access8086Virtual(SS, SP, 2, 1  /* Ecriture */, $SS, &EIP);

EFLAGS.TF = 0;   /* Inhibe le mode pas à pas (Chapitre 8) */
EFLAGS.IF = 0;   /* Inhibe les interruptions INTR         */

/* Lecture de l'élément de la table d'interruptions indexé */
/* par le numéro de vecteur et chargement de CS et IP avec */
/* le pointeur du handler.                                 */
if (VecNumber*4+3 > IDTR.Limit)
   SegmentException($GP, 0);

AccessLinear(IDTR.Base+VecNumber*4, 2, 0  /* PL 0 */,
             0 /* Lecture */, &IP);
AccessLinear(IDTR.Base+VecNumber*4+2, 2,0 /* PL 0 */,
             0 /* Lecture */, &CS.Selector);

} /* fin de InterruptREAL */
```

Listing 9.6 : *Traitement des interruptions et des exceptions en mode réel.*

1 Exécuter un système d'exploitation 80286 sur le 80386, et exécuter le programme d'application 80286 sur ce système d'exploitation.

Cette méthode utilise uniquement la partie 16 bits du 80386, ce qui équivaut à un 80286 rapide.

2 Exécuter un système d'exploitation 32 bits sur le 80386 et exécuter le programme d'application en interposant une bibliothèque d'interface 16 bits entre ce programme et le système d'exploitation 32 bits. Cette bibliothèque transforme les fonctions d'appels au système d'exploitation 16 bits utilisées par le programme d'application 80286 en appels à des fonctions 32 bits du système d'exploitation du 80386.

Le 80386 peut exécuter des systèmes d'exploitation 80286 parce que le modèle de gestion mémoire, de protection et de tâches du 80286 est un sous-ensemble du modèle du 80386 (voir le Chapitre 5), de même pour les registres 80286, les modes d'adressage, et les instructions. Le 80286 utilise les mêmes segments mémoire que le 80386, mais avec des adresses de base et des limites de segments de taille limitée. Le 80286 utilise 8 types de segments système et de guichets (TSS 286 disponible, LDT, TSS 286 occupé, guichet de tâche, guichet de trappe 286, et guichet d'interruption 286).

Le 80286 utilise des segments mémoire dont les descripteurs ont un format identique à celui des descripteurs du 80386 (voir la Figure 5.10), sauf que les 16 bits de poids fort sont nuls. Ainsi :

1 Les huit bits de poids fort de l'adresse de base sont nuls, limitant l'espace adressable physique à 24 bits.

2 Le bit G est nul (limite à granulité d'un octet).

3 Le bit D est nul (segments code, segments pile et segments du type expand-down du type 16 bits).

4 Les 4 bits de poids fort de la limite sont nuls. Ce fait, ajouté à la restriction de la granulité à un octet, donne une limite de segment maximale de 0FFFFh (taille d'un segment : 64 Ko).

Les descripteurs de segments système ont les mêmes restrictions que les segments mémoire (voir le Figure 5.11). Les guichets (voir la Figure 5.12) sont aussi limités par leurs 16 bits de poids fort nuls, ce qui limite le champ déplacement à 64 Ko. Les guichets d'appel 286 interprètent le champ DmotCompt comme un compteur de mots et non comme un compteur de dmots, si bien qu'un guichet d'appel dans un CALL interniveau copiera 2*DmotCompt octets de paramètres.

Les guichets d'appel, de trappe et d'interruption 286 empilent des données 16 bits qui sont les 16 bits de poids faible des données 32 bits empilées par les guichets d'appel, de trappe et d'interruption 386. Sinon, ils fonctionnent de la même façon que les guichets 386 (voir le Chapitre 5). On peut utiliser les instructions RET et IRET avec des opérandes 16 bits pour retourner à des procédures ayant appelé ces guichets 286.

Le mécanisme de gestion de tâches du 286 est le même que celui du 386, si ce n'est que seul l'état des registres 16 bits est sauvé et récupéré à chaque commutation de tâches décrites par un TSS 286. Le format d'un TSS 286 correspond aux 16 bits de poids faible des champs d'un TSS 386, et omet aussi le champ CR3 et la table E/S.

Les instructions FLDENV, FRSTOR, FSTENV, FNSTENV, FSAVE et FNSAVE décrites au Chapitre 3 chargent et stockent les registres de contrôle, d'état, d'étiquette et de pointeur d'erreur. Si ces instructions sont exécutées avec des opérandes 16 bits, le format utilisé pour sauver et récupérer ces données est simplement constitué des 16 bits de poids faible de chaque dmot du format 32 bits de la Figure 2.22. Les composantes déplacement des pointeurs d'erreur sont donc limitées à 16 bits, et le champ CodeOp omis.

ANNEXE A

Comparaison des 80386 80286 et 8086

Dans cette annexe, nous allons décrire les différences entre le 8086, le 80286 et le 80386. Nous n'allons pas redonner le détail des registres, des modes d'adressage, du codage des instructions, de la segmentation du 80286, de la segmentation en mode réel et du traitement des interruptions en mode réel (voir le Chapitre 9).

LE 8086 COMPARE AU 80386

Le Tableau A.1 donne la liste des instructions du 80386 qui n'existent pas sur le 8086.

Toutes les instructions liées au mécanisme de protection (quatre niveaux de protection, descripteurs et leurs tables, tâches et guichets) et de pagination n'existent pas sur le 8086 (voir le Tableau A.2).

De plus, la sémantique de toutes les instructions qui affectent les registres segments (PUSH, POP, MOV, LES, LDS) et de celles qui modifient le déroulement d'un programme (CALL, INT, INTO, IRET, JMP, RET) est très différente de la sémantique des instructions du 80386 en mode protégé.

Nous allons examiner les différences entre le 8086 et le 80386 pour certaines instructions.

Exceptions du type division

Pour le 8086, la valeur CS:IP d'une erreur du type division pointe sur l'instruction suivante. Pour le 80386, cette valeur pointe sur l'instruction en cause.

INSTRUCTION	DESCRIPTION
PUSH imm	Empilage de données immédiates
PUSHA et POPA	Empilage et Dépilage des huit registres généraux
IMUL reg,imm	Multiplication signée en immédiat
IMUL reg,reg/mem,imm	Multiplication signée en immédiat
IMUL reg,reg/mem	Multiplication signée
RCR/RCL reg/mem,imm	Rotation en immédiat
ROR/ROL reg/mem,imm	Rotation en immédiat
SAL/SAR reg/mem,imm	Décalage en immédiat
SHL/SHR reg/mem,imm	Décalage en immédiat
INS/OUTS	E/S chaîne
ENTER/LEAVE	Entrée et sortie de procédure
BOUND	Vérification des limites d'un tableau
LSS/LFS/LGS	Chargement de segment
Jcc 32 bits	Saut conditionnel long
BT/BTC/BTR/BTS	Opérations binaires
BSF/BSR	Recherche binaire
SHRD/SHLD	Double décalage
SETcc	Assignation conditionnelle de bit
MOVSX/MOVZX	MOV avec extension de signe
MOV DRx,reg;reg,DRx	MOV sur registre de mise au point
MOV CRx,reg;reg,CRx	MOV sur registre de contrôle
Préfixes FS et GS	Préfixes segments pour FS et GS

Tableau A.1 : *Instructions générales du 80386 qui n'existent pas sur le 8086.*

INSTRUCTION	DESCRIPTION
ARPL	Ajustement du niveau de privilège requis
CLTS	Effacement de l'indicateur de commutation de tâches
LAR	Chargement des droits d'accès
LGDT	Chargement de la table des descripteurs globaux
LIDT	Chargement de la table des descripteurs d'interruption
LLDT	Chargement de la table des descripteurs locaux
LMSW	Chargement du mot d'état machine
LSL	Chargement de la limite de segment
LTR	Chargement du registre de tâche
SGDT	Stockage de la table des descripteurs globaux
SIDT	Stockage de la table des descripteurs d'interruption
SLDT	Stockage de la table des descripteurs locaux
SMSW	Stockage du mot d'état machine
STR	Stockage du registre de tâche
VERR	Vérification du segment en lecture
VERW	Vérification du segment en écriture

Tableau A.2 : *Instructions de protection du 80386 qui n'existent pas sur le 8086.*

CodeOps non définis

Les CodeOp qui ne sont pas définis sur le 8086 génèrent une exception du type CodeOp non valide (exception 6) sur le 80386 ou exécutent une des nouvelles instructions du 80386.

Valeur de PUSH SP

La valeur empilée par PUSH SP est la version postincrémentée pour le 8086 et la version préincrémentée pour le 80386.

Décalages et rotations

Le compteur de décalages du 8086 a une taille de 8 bits pour toutes les instructions de décalage et de rotation. Ce compteur a une taille de 5 bits pour le 80386 (c'est-à-dire que le nombre de décalages est tronqué modulo 32 sur le 80386).

Préfixes redondants

Le 8086 n'impose pas de limite à la longueur d'une instruction, alors que le 80386 limite la longueur d'une instruction à 15 octets. Ainsi, des préfixes redondants peuvent générer des exceptions du type CodeOp non valide.

Accès au-delà des 64 Ko

Le 8086 gère des segments de mémoire cyclique et génère une adresse nulle si on dépasse la limite de segment de 64 Ko. Sur le 80386, on aboutit dans ce cas à une violation de limite si le segment est de taille 64 Ko et que le processeur travaille en mode réel. Ceci est vrai pour les accès à des opérandes mémoire et pour des accès séquentiels à du code.

LOCK

L'emploi de LOCK est limité à certaines instructions sur le 80386 (voir le Chapitre 3), mais pas sur le 8086.

Mode pas à pas

La priorité d'une exception du type mode pas à pas est plus haute que celle d'une interruption externe. Ainsi, on ne pourra pas exécuter en mode pas à pas un handler d'interruption externe.

Registre FLAGS

Le registre FLAGS est différent pour les bits 12 à 15 (voir le Chapitre 2).

Interruption NMI

NMI ne peut pas interrompre un handler NMI sur le 80386, alors que cela est possible sur le 8086.

Erreur du coprocesseur

Les erreurs du coprocesseur utilisent le numéro de vecteur 16 sur le 80386. Le 8086 peut employer n'importe quel numéro de vecteur.

Préfixes sur les instructions du coprocesseur

Lorsqu'un handler d'exception du type coprocesseur est appelé, la valeur sauvée de CS:IP par le 8086 n'inclut pas d'éventuels préfixes. Le 80386 pointe sur le début de l'instruction du coprocesseur, englobant donc d'éventuels préfixes.

Adressage d'un méga-octet

Le 8086 ne dispose que d'un espace adressable physique sur 20 bits. Les adresses physiques (segment + déplacement) qui dépassent 1 méga-octet sont tronquées en adresses sur 20 bits. Le 80386, qui dispose d'adresses physiques sur 32 bits, n'a pas ce problème.

Répétition d'instructions opérant sur des chaînes

Le CS:IP sauvé par le 8086 pour des instructions sur chaînes répétées (Préfixe REPEAT) n'inclut pas d'éventuels préfixes. Le 80386 pointe sur le début de ces instructions, englobant donc d'éventuels préfixes.

LE 80286 COMPARE AU 80386

Les différences les plus importantes entre le 80286 et le 80386 portent sur les types de données et adresses 32 bits, et sur la gestion mémoire et la pagination. En effet, plusieurs instructions ont été ajoutées au 80386. Dans le Chapitre 9, nous avons présenté les différences existant pour la segmentation et l'adressage. Le Tableau A.3 donne une liste des instructions du 80386 qui n'existent pas sur le 80286.

INSTRUCTION	DESCRIPTION
LSS/LFS/LGS	Chargement de segment
Jcc 32 bits	Saut conditionnel long
BT/BTC/BTR/BTS	Opérations binaires
BSF/BSR	Recherche binaire
SHRD/SHLD	Double décalage
SETcc	Positionnement conditionnel de bit
MOVSX/MOVZX	MOV avec extension de signe
IMUL reg,reg/mem	Multiplication signée
MOV DRx,reg;reg,DRx	MOV sur registre de mise au point
MOV CRx,reg;reg,CRx	MOV sur registre de contrôle
Préfixes FS et GS	Préfixes segments pour FS et GS

Tableau A.3 : *Instructions du 80386 qui n'existent pas sur le 80286.*

Voici quelques différences entre le 80286 et le 80386 dans l'exécution de certaines instructions.

LOCK

L'emploi de LOCK est limité à certaines instructions sur le 80386 (voir le Chapitre 3). Il n'est pas limité sur le 80286.

ANNEXE B

Comparaison des 80387 80287 et 8087

Cette annexe décrit les différences qui existent entre le 8087, le 80287 et le 80387.

LE 80287 (ET LE 8087) COMPARE AU 80387

Dans ce paragraphe, nous allons énoncer les différences qui existent entre le 80387 et le 80287. Ce sont les mêmes qui existent entre le 80387 et le 8087. Dans le paragraphe suivant, nous décrirons les différences supplémentaires entre le 8087 et le 80287. Le Tableau B.1 donne une liste des instructions du 80387 qui n'existent ni sur le 8087 ni sur le 80287.

Exécution des instructions

L'exécution des instructions peut aussi être différente dans les cas suivants.

Intervalles étendus

Les intervalles de définition des instructions FSCALE, FPTAN, FPATAN et F2XM1 ont été étendus (voir le Chapitre 3).

FPTAN

Le bit 10 du mot d'état (C2) est positionné pour indiquer un FPTAN inachevé, alors que ce bit n'est pas défini pour le 80287 et le 8087.

INSTRUCTION	DESCRIPTION
FPREM1	Reste partiel - IEEE
FUCOM/FUCOMP/FUCOMPP	Comparaison
FSIN	Sinus
FCOS	Cosinus
FSINCOS	Sinus et cosinus

Tableau B.1 : *Instructions du 80387 qui n'existent ni sur le 80287 ni sur le 8087.*

Opérations sur les dénormalisés

FBSTP, FDIV, FIST, FISTP, FPREM et FSQRT peuvent travailler sur des opérandes dénormalisés sur le 80387, alors que ce n'est pas possible sur le 80287 ni sur le 8087.

Codes de condition FPREM

Les codes de condition (CC3-CC0) ne sont pas positionnés convenablement sur les 80287 et 8087, alors qu'ils le sont sur le 80387.

FLD de dénormalisé

Un dénormalisé ne génère pas une exception du type dénormalisé sur le 80387 mais pas sur le 80287 ni sur le 8087.

FXTRACT

Un opérande nul génère une exception du type division par zéro et positionne ST(1) à moins l'infini sur le 80387, alors que les 80287 et 8087 positionnent ST(1) à 0 et ne génèrent pas d'exception. Un opérande plus l'infini ne génère pas d'exception sur le 80387, alors que les 80287 et 8087 génèrent une exception du type opération non valide.

FLD d'une constante

Le contrôle d'arrondi est activé sur le 80387, mais pas sur le 80287 ni sur le 8087.

FLD d'un réel court ou d'un réel long

Le chargement d'un dénormalisé entraîne la conversion d'un nombre en précision étendue sur le 80387, alors qu'on obtient un anormalisé sur les 80287 et 8087. Charger un NaN signalé génère une exception du type opération non valide sur le 80387 mais pas sur le 80287 ni sur le 8087.

FSETPM

FSETPM est une instruction 80287 qui est un FNOP sur le 80387.

Fonctions transcendantes

Le bit d'arrondi du mot d'état est généré par le 80387, alors qu'il est indéfini sur le 80287 et le 8087.

Autres différences

Nous allons énoncer d'autres différences qui ne concernent pas des instructions particulières mais plutôt l'initialisation, les types de données et les mots de contrôle, d'état et d'étiquette.

RESET

A la réinitialisation (reset), les bits IE et ES du mot d'état sont positionnés et le bit IM du mot de contrôle est initialisé pour indiquer la présence du 80387. Le 80386 peut ainsi déterminer s'il a affaire à un 80387 ou à un 80287.

NaN silencieux

Le 80287 ne gère qu'un type de NaN, le NaN signalé. Le 80387 gère en plus le NaN silencieux, qui ne génère pas une exception.

Formats non autorisés

Le 80387 génère une exception du type opération non valide chaque fois qu'il rencontre un des formats non autorisés suivants (Tableau 1.7) : pseudo-NaN, pseudo-zéro, pseudo-infini, et anormal. Le 80287 et le 8087 autorisent ces formats mais ne génèrent pas d'exception. Ainsi, les formats pseudo-zéro et anormal sont considérés comme valides par le 80287 (00b dans le mot étiquette), alors que le 80387 les considère comme particuliers (10b dans le mot étiquette).

Opération non valide

Un FSQRT, FDIV, et FPREM ou une conversion en BCD ou en entier ne génèrent pas une exception du type opération non valide sur le 80387, mais pas les 80287/8087.

Exception du type dénormalisé

Les instructions transcendantales et FXTRACT positionnent une exception du type dénormalisé sur le 80387, mais pas sur le 80287.

Exception du type débordement (overflow)

Lorsque le mode d'arrondi est du type troncature, le résultat masqué d'une exception du type overflow est le nombre valide le plus positif ou négatif pour le 80387, alors qu'il est égal à plus ou moins l'infini pour les 80287 et 8087. Si l'exception n'est pas masquée, on aura une erreur du type précision sur le 80387 et pas sur le 80287. De plus, dans ce cas, le résultat rangé en pile est arrondi par le 80387 mais pas par le 80287 ni le 8087.

Exception du type débordement (underflow)

Lorsque l'exception du type underflow est masquée, le 80387 génère une exception du type underflow uniquement quand ce débordement entraîne une perte de précision. Le 80287 génère une exception du type underflow si le débordement s'est effectué avec un contrôle d'arrondi du type arrondi à zéro, même s'il n'y a pas perte de précision. Lorsque l'exception du type underflow n'est pas masquée et que le résultat est rangé en pile, ce dernier est arrondi par le 80387 mais pas par le 80287 ni le 8087.

Priorité des exceptions

Pour le 80387, l'exception du type dénormalisé a la même priorité, masquée ou non. Pour les 80287 et 8087, elle devient la plus prioritaire si elle est non masquée. Ceci permet au 80387 d'économiser la normalisation inutile d'un dénormalisé.

Contrôle de l'infini

Le 80387 permet uniquement l'interprétation affine de l'infini, alors que les 80287 et 8087 emploient l'interprétation affine et projetée.

Débordement de pile

Dans le cas d'une faute du type pile, le bit 6 du mot d'état est positionné ainsi que le bit 10 pour indiquer un débordement de pile, en plus de l'exception (bit 0) pour le 80387. Les 80287 et 8087 ne signalent que l'exception.

Bits de condition du mot d'état

Les bits de condition du mot d'état sont mis à 0 par FINIT, FPREM et par une réinitialisation du 80387. Le 80287 ne modifie pas ces bits lors de ces opérations.

Mot d'étiquette

Il est possible de charger des valeurs du mot d'étiquette qui ne sont pas en accord avec le contenu du registre (instructions FLDENV et FRSTOR). Le 80387 traite uniquement les bits vide/non vide (valeurs 11 pour vide, et 00, 01, et 10 pour non vide) du mot d'étiquette et analyse le contenu du registre non vide pour les autres cas. Les 80287 et 8087 traitent la valeur du mot d'étiquette pour déterminer la classe d'opérande et n'analysent pas le contenu du registre.

LE 8087 COMPARE AU 80387

En plus des différences existant entre les 80287/8087 et le 80387, le 8087 présente certaines différences supplémentaires avec le 80387. Le traitement des exceptions tout particulièrement est différent. Les handlers d'exceptions du 8087 devront pour la plupart être réécrits pour pouvoir s'exécuter sur le 80387. Dans la majorité des cas, les programmes d'application pourront s'exécuter sans changement.

FENI/FNENI et FDISI/FNDISI

Le 80387 ignore les instructions FENI/FNENI et FDISI/FNDISI (l'état du 80387 n'est pas modifié). Les handlers d'exceptions du 8087 qui utilisent ces instructions ne seront certainement pas compatibles.

Exceptions du type numérique

Le 80387 et le 8087 ne traitent pas une exception de la même manière (voir l'Annexe A). Le 8087 peut traiter des interruptions via n'importe quel vecteur d'interruption, alors que le 80387 doit les traiter via le vecteur

d'interruption 16. De plus, les interruptions du 8087 sont gérées par un contrôleur d'interruption séparé. Cette façon de traiter les exceptions numériques n'est pas possible avec le 80387.

Pointeur d'exception

Les formats des pointeurs d'instructions et de données sauvegardés sont différents pour le 8087 et le 80387 (voir le Chapitre 9).

Segmentation

Il est possible de générer une faute du type segmentation sur le 80387 si l'instruction dépasse la limite du segment (exception 13) ou si l'opérande déborde du segment (exceptions 9 et 13). Le 8087 ne permet pas cela.

Exception 7

Lors de l'exécution d'une instruction à virgule flottante, il est possible de générer une exception indiquant l'émulation ou une commutation de tâches (bits EM et TS du registre CR0 du 80386) sur le 80387. Ce n'est pas possible sur le 8087.

Préfixes d'instructions du coprocesseur

Lors de l'appel à un handler d'exceptions du coprocesseur, la valeur de CS:IP sauvée par le 8087 ne comprend pas d'éventuels préfixes. Le 80387 pointe sur le début de l'instruction, ce qui inclut d'éventuels préfixes.

Instruction WAIT

Le 80387 et le 80386 sont automatiquement synchronisés. Le programmeur n'a donc pas besoin d'inclure des instructions WAIT, sauf après des instructions qui rangent dans la mémoire des données en provenance du 80387. Les programmes qui comprennent des WAIT inutiles s'exécutent normalement, mais gaspillent du code et du temps lors de l'exécution.

ANNEXE C

Table binaire, hexadécimale et décimale

HEX	BINAIRE	DECIMAL
0	0	0
1	1	1
2	10	2
3	11	3
4	100	4
5	101	5
6	110	6
7	111	7
8	1000	8
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15
10	10000	16
11	10001	17
.	.	.
.	.	.
.	.	.

ANNEXE D

Puissances de deux

PUISSANCE	VALEUR DECIMALE
2^{0}	1
2^{1}	2
2^{2}	4
2^{3}	8
2^{4}	16
2^{5}	32
2^{6}	64
2^{7}	128
2^{8}	256
2^{9}	512
2^{10}	1.024
2^{11}	2.048
2^{12}	4.096
2^{13}	8.192
2^{14}	16.384
2^{15}	32.768
2^{16}	65.536
2^{17}	131.072
2^{18}	262.144
2^{19}	524.288
2^{20}	1.048.576
2^{21}	2.097.152
2^{22}	4.194.304
2^{23}	8.388.608
2^{24}	16.777.216
2^{25}	33.554.432
2^{26}	67.108.864
2^{27}	134.217.728
2^{28}	268.435.456
2^{29}	536.870.912
2^{30}	1.073.741.824
2^{31}	2.147.483.648
2^{32}	4.294.967.296

ANNEXE E

Table des codes ASCII

	3 BITS SUP							
4 BITS INF	**000**	**0001**	**010**	**011**	**100**	**101**	**110**	**111**
0000	NUL	DLE	SP	0	@	P	`	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	"	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	'	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	|
1101	CR	GS	-	=	M	]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	_	o	DEL

Clés

NUL	Nul	DLE	Echap. du lien de données
SOH	Début d'en-tête	DC1	Commande 1 du disp.
STX	Début du texte	DC2	Commande 2 du disp.
ETX	Fin du texte	DC3	Commande 3 du disp.
EOT	Fin de transmission	DC4	Commande 4 du disp.
ENG	Requête	NAK	Accusé de réception négatif
ACK	Accusé de réception	SYN	Synchrone au repos
BEL	Signal sonore	ETB	Fin du bloc transmis
BS	Retour arrière	EM	Fin de média
HT	Tabulation horiz.	SUB	Remplace
LF	Saut de ligne	ESC	Echappement
VT	Tabulation vert.	FS	Séparateur de fichiers
FF	Saut de page	GS	Séparateur de groupes
CR	Retour chariot	RS	Séparateur d'enregistrements
SO	Sortie du mode maj	US	Séparateur d'unités
SI	Entrée en mode maj	DEL	Effacement

ANNEXE F

Table des CodeOp 80386

Les tables de CodeOp de cette annexe servent à interpréter du code objet 80386. Utilisez les 4 bits de poids fort du CodeOp comme un index sur les rangées de la table ; utilisez les 4 bits de poids faible comme un index sur les colonnes. Si le CodeOp est égal à 0FH, reportez-vous à la table des CodeOp sur deux octets et utilisez le second octet du CodeOp pour indexer les rangées et les colonnes de cette table.

ABREVIATIONS

Les opérandes sont identifiés par un code sur deux caractères de la forme Zz. Le premier caractère, en majuscule, détermine l'adressage. Le second caractère, en minuscule, détermine le type de l'opérande.

Codage de l'adressage

A Adressage direct : l'instruction n'a pas de champ MODRM ; l'adresse de l'opérande est codée dans l'instruction ; il n'y a pas de registre de base, de registre d'index, de facteur d'échelle ; exemple : far JMP (EA).

C Le champ reg du champ MODRM sélectionne un registre de contrôle ; exemple : MOV (0F20, 0F22).

D Le champ reg du champ MODRM sélectionne un registre de mise au point ; exemple : MOV (0F21, 0F23).

E Un champ MODRM suit le CodeOp et détermine l'opérande. L'opérande est soit un registre général, soit une adresse mémoire. Si c'est une adresse mémoire, l'adresse est calculée à partir du registre segment et des valeurs suivantes : un registre de base, un registre d'index, un facteur d'échelle, un déplacement.

F Indique le registre.

G Le champ reg du champ MODRM sélectionne un registre général ; exemple : ADD (00).

I Donnée immédiate. La valeur de l'opérande est codée dans les octets suivant l'instruction.

J L'instruction contient un déplacement relatif qui est ajouté au registre pointeur d'instruction ; exemple : JMP short, LOOP.

M Le champ MODRM peut faire référence uniquement à la mémoire; exemple : BOUND, LES, LDS, LSS, LFS, LGS.

O L'instruction n'a pas de champ MODRM ; le déplacement de l'opérande est codé comme un mot ou un dmot (selon l'attribut de la taille d'adresse) dans l'instruction. Il n'y a pas de registre de base, de registre d'index, ni de facteur d'échelle ; exemple : MOV (A0-A3).

R Le champ mod du champ MODRM peut faire uniquement référence à un registre général ; exemple : MOV (0F20-0F24, 0F26).

S Le champ reg du champ MODRM détermine un registre segment ; exemple : MOV (8C, 8E).

T Le champ reg du champ MODRM détermine un registre de test ; exemple : MOV (0F24, 0F26).

X Mémoire adressée par DS:SI ; exemple : MOVS, COMPS, OUTS, LODS, SCAS.

Y Mémoire adressée par ES:DI ; exemple : MOVS, CMPS, INS, STOS.

Codage du type d'opérande

a Deux opérandes mots en mémoire ou deux opérandes dmots en mémoire, selon l'attribut de taille de l'opérande (employé uniquement par BOUND).

b Octet (quel que soit l'attribut de taille de l'opérande).

c Octet ou mot, selon l'attribut de taille de l'opérande.

d Dmot (quel que soit l'attribut de taille de l'opérande).

p Pointeur 32 bits ou 48 bits, selon l'attribut de taille de l'opérande.

s Pseudo-descripteur de six octets.

v Mot ou dmot, selon l'attribut de taille de l'opérande.

w Mot (quel que soit l'attribut de taille de l'opérande).

Codage des registres

Lorsqu'un opérande est un registre spécifique codé dans le CodeOp, le registre est identifié par son nom ; exemple : AX, CL, ou ESI. Le nom du registre indique si ce registre est de taille 32, 16 ou 8 bits. Un identificateur de registre de la forme eXX est employé lorsque la taille du registre dépend de l'attribut de taille de l'opérande. Par exemple, eAX indique que le registre AX est employé lorsque l'attribut de taille de l'opérande est égal à 16, et que c'est le registre EAX qui est utilisé lorsque cet attribut est égal à 32.

Table des CodeOp sur un octet

	0	1	2	3	4	5	6	7
0	ADD						PUSH ES	POP ES
	Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev	AL,Ib	eAX,Iv		
1	ADC						PUSH SS	POP SS
	Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev	AL,Ib	eAX,Iv		
2	AND						SEG =ES	DAA
	Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev	AL,Ib	eAX,Iv		
3	XOR						SEG =SS	AAA
	Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev	AL,Ib	eAX,Iv		
4	INC general register							
	eAX	eCX	eDX	eBX	eSP	eBP	eSI	eDI
5	PUSH general register							
	eAX	eCX	eDX	eBX	eSP	eBP	eSI	eDI
6	PUSHA	POPA	BOUND Gv,Ma	ARPL Ew,Rw	SEG =FS	SEG =GS	Operand Size	Address Size
7	Short-displacement jump on condition (Jb)							
	JO	JNO	JB	JNB	JZ	JNZ	JBE	JNBE
8	Immediate Grpl			Grpl Ev,Ib	TEST		XCHG	
	Eb,Ib	Ev,Iv			Eb,Gb	Ev,Gv	Eb,Gb	Ev,Gv
9	NOP	XCHG word or double-word register with eAX						
		eCX	eDX	eBX	eSP	eBP	eSI	eDI
A	MOV				MOVSB Xb,Yb	MOVSW/D Xv,Yv	CMPSB Xb,Yb	CMPSW/D Xv,Yv
	AL,Ob	eAX,Ov	Ob,AL	Ov,eAX				
B	MOV immediate byte into byte register							
	AL	CL	DL	BL	AH	CH	DH	BH
C	Shift Grp2		RET near		LES Gv,Mp	LDS Gv,Mp	MOV	
	Eb,Ib	Ev,Ib	Iw				Eb,Ib	Ev,Iv
D	Shift Grp2				AAM	AAD		XLAT
	Eb,1	Ev,1	Eb,CL	Ev,CL				
E	LOOPNE Jb	LOOPE Jb	LOOP Jb	JCXZ Jb	IN		OUT	
					AL,Ib	eAX,Ib	Ib,AL	Ib,eAX
F	LOCK		REPNE	REP REPE	HLT	CMC	Unary Grp3	
							Eb	Ev

Table des CodeOp sur un octet

	8	9	A	B	C	D	E	F
0	OR						PUSH CS	2-byte escape
	Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev	AL,Ib	eAX,Iv		
1	SBB						PUSH DS	POP DS
	Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev	AL,Ib	eAX,Iv		
2	SUB						SEG =CS	DAS
	Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev	AL,Ib	eAX,Iv		
3	CMP						SEG =CS	AAS
	Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev	AL,Ib	eAX,Iv		
4	DEC general register							
	eAX	eCX	eDX	eBX	eSP	eBP	eSI	eDI
5	POP into general register							
	eAX	eCX	eDX	eBX	eSP	eBP	eSI	eDI
6	PUSH Ib	IMUL GvEvIv	PUSH Ib	IMUL GvEvIb	INSB Yb,DX	INSW/D Yv,DX	OUTSB DX,Xb	OUTSW/D DX,Xv
7	Short-displacement jump on condition (Jb)							
	JS	JNS	JP	JNP	JL	JNL	JLE	JNLE
8	MOV				MOV Ew,Sw	LEA Gv,M	MOV Sw,Ew	POP Ev
	Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev				
9	CBW	CWD	CALL Ap	WAIT	PUSHF Fv	POPF Fv	SAHF	LAHF
A	TEST		STOSB Yb,AL	STOSW/D Yv,eAX	LODSB AL,Xb	LODSW/D eAX,Xv	SCASB AL,Xb	SCASW/D eAX,Xv
	AL,Ib	eAX,Iv						
B	MOV immediate word or double into word or double register							
	eAX	eCX	eDX	eBX	eSP	eBP	eSI	eDI
C	ENTER Iw,Ib	LEAVE	RET far		INT 3	INT Ib	INTO	IRET
			Iw					
D	ESC (Escape to coprocessor instruction set)							
E	CALL Av	JMP			IN		OUT	
		Jv	Ap	Jb	AL,DX	eAX,DX	DX,AL	DX,eAX
F	CLC	STC	CLI	STI	CLD	STD	INC/DEC Grp4	Indirct Grp5

Table des CodeOp sur deux octets (premier octet égal à 0FH)

	0	1	2	3	4	5	6	7
0	Grp6	Grp7	LAR Gv,Ew	LSL Gv,Ew			CLTS	
1								
2	MOV Cd,Rd	MOV Dd,Rd	MOV Rd,Cd	MOV Rd,Dd	MOV Td,Rd		MOV Rd,Td	
3								
4								
5								
6								
7								
8	Long-displacement jump on condition (Jv)							
	JO	JNO	JB	JNB	JZ	JNZ	JBE	JNBE
9	Byte Set on condition (Eb)							
	SETO	SETNO	SETB	SETNB	SETZ	SETNZ	SETBE	SETNBE
A	PUSH FS	POP FS		BT Ev,Gv	SHLD EvGvIb	SHLD EvGvCL		
B			LSS Mp	BTR Ev,Gv	LFS Mp	LGS Mp	MOVZX	
							Gv,Eb	Gv,Ew
C								
D								
E								
F								

Table des CodeOp sur deux octets (premier octet égal à 0FH)

	8	9	A	B	C	D	E	F
0								
1								
2								
3								
4								
5								
6								
7								
8	Long-displacement jump on condition (Jv)							
	JS	JNS	JP	JNP	JL	JNL	JLE	JNLE
9	SETS	SETNS	SETP	SETNP	SETL	SETNL	SETLE	SETNLE
A	PUSH GS	POP GS		BTS Ev,Gv	SHRD EvGvIb	SHRD EvGvCL		IMUL Gv,Ev
B			Grp-8 Ev,Ib	BTC Ev,Gv	BSF Gv,Ev	BSR Gv,Ev	MOVSX	
							Gv,Eb	Gv,Ew
C								
D								
E								
F								

CodeOp déterminés par les bits 5, 4, 3 du champ MODRM

mod	nnn	R/M

Group	000	001	010	011	100	101	110	111
1	ADD	OR	ADC	SBB	AND	SUB	XOR	CMP
2	ROL	ROR	RCL	RCR	SHL	SHR		SAR
3	TEST Ib/Iv		NOT	NEG	MUL AL/eAX	IMUL AL/eAX	DIV AL/eAX	IDIV AL/eAX
4	INC Eb	DEC Eb						
5	INC Ev	DEC Ev	CALL Ev	CALL eP	JMP Ev	JMP Ep	PUSH Ev	

CodeOp déterminés par les bits 5, 4, 3 du champ MODRM

mod	nnn	R/M

Group	000	001	010	011	100	101	110	111
6	SLDT Ew	STR Ew	LLDT Ew	LTR Ew	VERR Ew	VERW Ew		
7	SGDT Ms	SIDT Ms	LGDT Ms	LIDT Ms	SMSW Ew		LMSW Ew	
8					BT	BTS	BTR	BTC

ANNEXE G

Format des instructions du 80386 et timing

Dans cette annexe, nous décrivons le jeu d'instructions du 80386. Le Tableau G.1 donne une liste de toutes les instructions avec leur codage et leur nombre de cycles. Puis nous expliquerons la structure du codage des instructions et définirons tous les champs pouvant apparaître dans les instructions 80386.

CODAGE DES INSTRUCTIONS 80386 ET NOMBRE DE CYCLES

Pour calculer le temps écoulé lors d'une instruction, multipliez le nombre de cycles de cette instruction (voir le Tableau G.1) par la période d'horloge du processeur (exemple : 62,5 ns pour un 80386-16 fonctionnant à 16 MHz).

Reportez-vous au paragraphe qui traite du codage des instructions pour plus de détails.

Le Tableau G.1 suit les règles suivantes :

1 L'instruction a été traitée et décodée. Elle est prête pour l'exécution.

2 Les cycles de bus ne nécessitent pas d'états d'attente.

3 Aucune demande HOLD du bus local ne ralentit les accès du processeur au bus.

4 Aucune exception n'est détectée lors de l'exécution.

5 Si une adresse effective est calculée, elle n'emploie pas deux registres généraux. Les cycles d'horloge sont donnés pour une adresse formée par un registre, un facteur d'échelle et un déplacement. Cependant, si le calcul de l'adresse effective utilise deux registres généraux, ajoutez 1 cycle d'horloge au nombre de cycles donné.

Tableau G.1 : 80386 Jeu d'instructions Compteur de cycles

INSTRUCTION	FORMAT			COMPTEUR		NOTES	
				Mode adresse réelle ou mode virtuel 8086	Mode protégé adresse virtuelle	Mode adresse réelle ou mode virtuel 8086	Mode protégé adresse virtuelle
GENERAL DATA TRANSFER							
MOV = Move:							
Register to Register/Memory	1000100w	mod reg r/m		2/2	2/2	b	h
Register/Memory to Register	1000101w	mod reg r/m		2/4	2/4	b	h
Immediate to Register/Memory	1100011w	mod000 r/m	immediate data	2/2	2/2	b	h
Immediate to Register (short form)	1011 w reg	immediate data		2	2		
Memory to Accumulator (short form)	1010000w	full displacement		4	4	b	h
Accumulator to Memory (short form)	1010001w	full displacement		2	2	b	h
Register Memory to Segment Register	10001110	mod sreg3 r/m		2/5	18/19	b	h, i, j
Segment Register to Register/Memory	10001100	mod sreg3 r/m		2/2	2/2	b	h
MOVSX = Move With Sign Extension							
Register From Register/Memory	00001111	1011111w	mod reg r/m	3/6	3/6	b	h
MOVZX = Move With Zero Extension							
Register From Register/Memory	00001111	1011011w	mod reg r/m	3/6	3/6	b	h
PUSH = Push:							
Register/Memory	11111111	mod110 r/m		5	5	b	h
Register (short form)	01010 reg			2	2	b	h
Segment Register (ES, CS, SS or DS) (short form)	000sreg2110			2	2	b	h
Segment Register (ES, CS, SS, DS, FS or GS)	00001111	10sreg3000		2	2	b	h
Immediate	011010s0	immediate data		2	2	b	h
PUSHA = Push All	01100000			18	18	b	h
POP = Pop							
Register/Memory	10001111	mod000 r/m		5	5	b	h
Register (short form)	01011 reg			4	4	b	h
Segment Register (ES, CS, SS or DS) (short form)	000sreg2111			7	21	b	h, i, j
Segment Register (ES, CS, SS or DS FS or GS)	00001111	10sreg3001		7	21	b	h, i, j
POPA = Pop All	01100001			24	24	b	h
XCHG = Exchange							
Register/Memory With Register	1000011w	mod reg r/m		3/5	3/5	b, f	f, h
Register With Accumulator (short form)	10010 reg			3	3		
IN = Input from:			**Clk Count Virtual 8086 Mode**				
Fixed Port	1110010w	port number	†26	12	6*/26**		m
Variable Port	1110110w		†27	13	7*/27**		m
OUT = Output to:							
Fixed Port	1110011w	port number	†24	10	4*/24**		m
Variable Port	1110111w		†25	11	5*/25**		m
LEA = Load EA to Register	10001101	mod reg r/m		2	2		

* If CPL ≤ IOPL ** If CPL > IOPL

Tableau G.1 : 80386 Jeu d'instructions Compteur de cycles

INSTRUCTION	FORMAT	COMPTEUR		NOTES	
		Mode adresse réelle ou mode virtuel 8086	Mode protégé adresse virtuelle	Mode adresse réelle ou mode virtuel 8086	Mode protégé adresse virtuelle
SEGMENT CONTROL					
LDS = Load Pointer to DS	11000101 | mod reg r/m	7	22	b	h, i, j
LES = Load Pointer to ES	11000100 | mod reg r/m	7	22	b	h, i, j
LFS = Load Pointer to FS	00001111 | 10110100 | mod reg r/m	7	25	b	h, i, j
LGS = Load Pointer to GS	00001111 | 10110101 | mod reg r/m	7	25	b	h, i, j
LSS = Load Pointer to SS	00001111 | 10110010 | mod reg r/m	7	22	b	h, i, j
FLAG CONTROL					
CLC = Clear Carry Flag	11111000	2	2		
CLD = Clear Direction Flag	11111100	2	2		
CLI = Clear Interrupt Enable Flag	11111010	3	3		m
CLTS = Clear Task Switched Flag	00001111 | 00000110	5	5	c	l
CMC = Complement Carry Flag	11110101	2	2		
LAHF = Load AH into Flag	10011111	2	2		
POPF = Pop Flags	10011101	5	5	b	h, n
PUSHF = Push Flags	10011100	4	4	b	h
SAHF = Store AH into Flags	10011110	3	3		
STC = Set Carry Flag	11111001	2	2		
STD = Set Direction Flag	11111001	2	2		
STI = Set Interrupt Enable Flag	11111011	3	3		m
ARITHMETIC					
ADD = Add					
Register to Register	000000dw | mod reg r/m	2	2		
Register to Memory	0000000w | mod reg r/m	7	7	b	h
Memory to Register	0000001w | mod reg r/m	6	6	b	h
Immediate to Register/Memory	100000sw | mod 000 r/m | immediate data	2/7	2/7	b	h
Immediate to Accumulator (short form)	0000010w | immediate data	2	2		
ADC = Add With Carry					
Register to Register	000100dw | mod reg r/m	2	2		
Register to Memory	0001000w | mod reg r/m	7	7	b	h
Memory to Register	0001001w | mod reg r/m	6	6	b	h
Immediate to Register/Memory	100000sw | mod 010 r/m | immediate data	2/7	2/7	b	h
Immediate to Accumulator (short form)	0001010w | immediate data	2	2		
INC = Increment					
Register/Memory	1111111w | mod 000 r/m	2/6	2/6	b	h
Register (short form)	01000 reg	2	2		
SUB = Subtract					
Register from Register	001010dw | mod reg r/m	2	2		

Tableau G.1 : 80386 Jeu d'instructions Compteur de cycles

INSTRUCTION	FORMAT	COMPTEUR		NOTES	
		Mode adresse réelle ou mode virtuel 8086	Mode protégé adresse virtuelle	Mode adresse réelle ou mode virtuel 8086	Mode protégé adresse virtuelle
ARITHMETIC (Continued)					
Register from Memory	0010100w | mod reg r/m	7	7	b	h
Memory from Register	0010101w | mod reg r/m	6	6	b	h
Immediate from Register/Memory	100000sw | mod 101 r/m | immediate data	2/7	2/7	b	h
Immediate from Accumulator (short form)	0010110w | immediate data	2	2		
SBB = Subtract with Borrow					
Register from Register	000110dw | mod reg r/m	2	2		
Register from Memory	0001100w | mod reg r/m	7	7	b	h
Memory from Register	0001101w | mod reg r/m	6	6	b	h
Immediate from Register/Memory	100000sw | mod 011 r/m | immediate data	2/7	2/7	b	h
Immediate from Accumulator (short form)	0001110w | immediate data	2	2		
DEC = Decrement					
Register/Memory	1111111w | reg 001 r/m	2/6	2/6	b	h
Register (short form)	01001 reg	2	2		
CMP = Compare					
Register with Register	001110dw | mod reg r/m	2	2		
Memory with Register	0011100w | mod reg r/m	5	5	b	h
Register with Memory	0011101w | mod reg r/m	6	6	b	h
Immediate with Register/Memory	100000sw | mod 111 r/m | immediate data	2/5	2/5	b	h
Immediate with Accumulator (short form)	0011110w | immediate data	2	2		
NEG = Change Sign	1111011w | mod 011 r/m	2/6	2/6	b	h
AAA = ASCII Adjust for Add	00110111	4	4		
AAS = ASCII Adjust for Subtract	00111111	4	4		
DAA = Decimal Adjust for Add	00100111	4	4		
DAS = Decimal Adjust for Subtract	00101111	4	4		
MUL = Multiply (unsigned)					
Accumulator with Register/Memory	1111011w | mod 100 r/m				
Multiplier-Byte		9–14/12-17	9–14/12–17	b, d	d, h
-Word		9–22/12-25	9–22/12–25	b, d	d, h
-Doubleword		9–38/12-41	9–38/12–41	b, d	d, h
IMUL = Integer Multiply (signed)					
Accumulator with Register/Memory	1111011w | mod 100 r/m				
Multiplier-Byte		9–14/12-17	9–14/12–17	b, d	d, h
-Word		9–22/12-25	9–22/12–25	b, d	d, h
-Doubleword		9–38/12-41	9–38/12–41	b, d	d, h
Register with Register/Memory	00001111 | 10101111 | mod reg r/m				
-Word		9–22/12-25	9–22/12–25	b, d	d, h
-Doubleword		9–38/12-41	9–38/12–41	b, d	d, h
Register/Memory with Immediate to Register	011010s1 | mod reg r/m | immediate data				
-Word		9–22/12-25	9–22/12–25	b, d	d, h
-Doubleword		9–38/12-41	9–38/12–41	b, d	d, h

Tableau G.1 : 80386 Jeu d'instructions Compteur de cycles

INSTRUCTION	FORMAT	COMPTEUR		NOTES	
		Mode adresse réelle ou mode virtuel 8086	Mode protégé adresse virtuelle	Mode adresse réelle ou mode virtuel 8086	Mode protégé adresse virtuelle
ARITHMETIC (Continued)					
DIV = Divide (Unsigned)					
Accumulator by Register/Memory	1111011w mod110 r/m				
Divisor—Byte		14/17	14/17	b,e	e,h
—Word		22/25	22/25	b,e	e,h
—Doubleword		38/41	38/41	b,e	e,h
IDIV = Integer Divide (Signed)					
Accumulator By Register/Memory	1111011w mod111 r/m				
Divisor—Byte		19/22	19/22	b,e	e,h
—Word		27/30	27/30	b,e	e,h
—Doubleword		43/46	43/46	b,e	e,h
AAD = ASCII Adjust for Divide	11010101 00001010	19	19		
AAM = ASCII Adjust for Multiply	11010100 00001010	17	17		
CBW = Convert Byte to Word	10011000	3	3		
CWD = Convert Word to Double Word	10011001	2	2		
LOGIC					
Shift Rotate Instructions					
Not Through Carry (**ROL, ROR, SAL, SAR, SHL,** and **SHR**)					
Register/Memory by 1	1101000w mod TTT r/m	3/7	3/7	b	h
Register/Memory by CL	1101001w mod TTT r/m	3/7	3/7	b	h
Register/Memory by Immediate Count	1100000w mod TTT r/m immed 8-bit data	3/7	3/7	b	h
Through Carry (**RCL** and **RCR**)					
Register/Memory by 1	1101000w mod TTT r/m	9/10	9/10	b	h
Register/Memory by CL	1101001w mod TTT r/m	9/10	9/10	b	h
Register/Memory by Immediate Count	1100000w mod TTT r/m immed 8-bit data	9/10	9/10	b	h
	TTT Instruction 000 ROL 001 ROR 010 RCL 011 RCR 100 SHL/SAL 101 SHR 111 SAR				
SHLD = Shift Left Double					
Register/Memory by Immediate	00001111 10100100 mod reg r/m immed 8-bit data	3/7	3/7		
Register/Memory by CL	00001111 10100101 mod reg r/m	3/7	3/7		
SHRD = Shift Right Double					
Register/Memory by Immediate	00001111 10101100 mod reg r/m immed 8-bit data	3/7	3/7		
Register/Memory by CL	00001111 10101101 mod reg r/m	3/7	3/7		
AND = And					
Register to Register	001000dw mod reg r/m	2	2		

Tableau G.1 : 80386 Jeu d'instructions Compteur de cycles

INSTRUCTION	FORMAT			Clk Count Virtual 8086 Mode	COMPTEUR: Mode adresse réelle ou mode virtuel 8086	COMPTEUR: Mode protégé adresse virtuelle	NOTES: Mode adresse réelle ou mode virtuel 8086	NOTES: Mode protégé adresse virtuelle
LOGIC (Continued)								
Register to Memory	0010000w	mod reg r/m			7	7	b	h
Memory to Register	0010001w	mod reg r/m			6	6	b	h
Immediate to Register/Memory	1000000w	mod 100 r/m	immediate data		2/7	2/7	b	h
Immediate to Accumulator (Short Form)	0010010w	immediate data			2	2		
TEST – And Function to Flags, No Result								
Register/Memory and Register	1000010w	mod reg r/m			2/5	2/5	b	h
Immediate Data and Register/Memory	1111011w	mod 000 r/m	immediate data		2/5	2/5	b	h
Immediate Data and Accumulator (Short Form)	1010100w	immediate data			2	2		
OR – Or								
Register to Register	000010dw	mod reg r/m			2	2		
Register to Memory	0000100w	mod reg r/m			7	7	b	h
Memory to Register	0000101w	mod reg r/m			6	6	b	h
Immediate to Register/Memory	1000000w	mod 001 r/m	immediate data		2/7	2/7	b	h
Immediate to Accumulator (Short Form)	0000110w	immediate data			2	2		
XOR – Exclusive Or								
Register to Register	001100dw	mod reg r/m			2	2		
Register to Memory	0011000w	mod reg r/m			7	7	b	h
Memory to Register	0011001w	mod reg r/m			6	6	b	h
Immediate to Register/Memory	1000000w	mod 110 r/m	immediate data		2/7	2/7	b	h
Immediate to Accumulator (Short Form)	0011010w	immediate data			2	2		
NOT – Invert Register/Memory	1111011w	mod 010 r/m			2/6	2/6	b	h
STRING MANIPULATION								
CMPS – Compare Byte Word	1010011w				10	10	b	h
INS – Input Byte/Word from DX Port	0110110w			†29	15	9*/29**	b	h, m
LODS – Load Byte/Word to AL/AX/EAX	1010110w				5	5	b	h
MOVS – Move Byte Word	1010010w				7	7	b	h
OUTS – Output Byte/Word to DX Port	0110111w			†28	14	8*/28**	b	h, m
SCAS – Scan Byte Word	1010111w				7	7	b	h
STOS – Store Byte/Word from AL/AX/EX	1010101w				4	4	b	h
XLAT – Translate String	11010111				5	5		h
REPEATED STRING MANIPULATION Repeated by Count in CX or ECX								
REPE CMPS – Compare String (Find Non-Match)	11110011	1010011w			5+9n	5+9n	b	h

* If CPL ≤ IOPL ** If CPL > IOPL

Tableau G.1 : 80386 Jeu d'instructions Compteur de cycles

INSTRUCTION	FORMAT					COMPTEUR		NOTES	
						Mode adresse réelle ou mode virtuel 8086	Mode protégé adresse virtuelle	Mode adresse réelle ou mode virtuel 8086	Mode protégé adresse virtuelle
REPEATED STRING MANIPULATION (Continued)									
REPNE CMPS = Compare String					**Clk Count Virtual 8086 Mode**				
(Find Match)	11110010	1010011w				5+9n	5+9n	b	h
REP INS = Input String	11110010	0110110w			†27+6n	13+6n	7+6n*/27+6n**	b	h, m
REP LODS = Load String	11110010	1010110w				5+6n	5+6n	b	h
REP MOVS = Move String	11110010	1010010w				7+4n	7+4n	b	h
REP OUTS = Output String	11110010	0110111w			†26+5n	12+5n	6+5n*/26+5n**	b	h, m
REPE SCAS = Scan String									
(Find Non-AL/AX/EAX)	11110011	1010111w				5+8n	5+8n	b	h
REPNE SCAS = Scan String									
(Find AL/AX/EAX)	11110010	1010111w				5+8n	5+8n	b	h
REP STOS = Store String	11110010	1010101w				5+5n	5+5n	b	h
BIT MANIPULATION									
BSF = Scan Bit Forward	00001111	10111100	mod reg r/m			10+3n	10+3n	b	h
BSR = Scan Bit Reverse	00001111	10111101	mod reg r/m			10+3n	10+3n	b	h
BT = Test Bit									
Register/Memory, Immediate	00001111	10111010	mod 100 r/m	immed 8-bit data		3/6	3/6	b	h
Register/Memory, Register	00001111	10100011	mod reg r/m			3/12	3/12	b	h
BTC = Test Bit and Complement									
Register/Memory, Immediate	00001111	10111010	mod 111 r/m	immed 8-bit data		6/8	6/8	b	h
Register/Memory, Register	00001111	10111011	mod reg r/m			6/13	6/13	b	h
BTR = Test Bit and Reset									
Register/Memory, Immediate	00001111	10111010	mod 110 r/m	immed 8-bit data		6/8	6/8	b	h
Register/Memory, Register	00001111	10110011	mod reg r/m			6/13	6/13	b	h
BTS = Test Bit and Set									
Register/Memory, Immediate	00001111	10111010	mod 101 r/m	immed 8-bit data		6/8	6/8	b	h
Register/Memory, Register	00001111	10101011	mod reg r/m			6/13	6/13	b	h
CONTROL TRANSFER									
CALL = Call									
Direct Within Segment	11101000	full displacement				7+m	7+m	b	r
Register/Memory									
Indirect Within Segment	11111111	mod010 r/m				7+m/ 10+m	7+m/ 10+m	b	h, r
Direct Intersegment	10011010	unsigned full offset, selector				17+m	34+m	b	j,k,r

Notes

† Clock count shown applies if I/O permission allows I/O to the port in virtual 8086 mode. If I/O bit map denies permission exception 13 fault occurs; refer to clock counts for INT 3 instruction.

* If CPL ≤ IOPL ** If CPL > IOPL

Tableau G.1 : 80386 Jeu d'instructions Compteur de cycles

INSTRUCTION	FORMAT	COMPTEUR		NOTES	
		Mode adresse réelle ou mode virtuel 8086	Mode protégé adresse virtuelle	Mode adresse réelle ou mode virtuel 8086	Mode protégé adresse virtuelle
CONTROL TRANSFER (Continued)					
Protected Mode Only (Direct Intersegment)					
Via Call Gate to Same Privilege Level			52 + m		h,j,k,r
Via Call Gate to Different Privilege Level, (No Parameters)			86 + m		h,j,k,r
Via Call Gate to Different Privilege Level, (x Parameters)			94 + 4x + m		h,j,k,r
From 286 Task to 286 TSS			273		h,j,k,r
From 286 Task to 386 TSS			298		h,j,k,r
From 286 Task to Virtual 8086 Task (386 TSS)			217		h,j,k,r
From 386 Task to 286 TSS			273		h,j,k,r
From 386 Task to 386 TSS			300		h,j,k,r
From 386 Task to Virtual 8086 Task (386 TSS)			217		h,j,k,r
Indirect Intersegment	11111111 mod 011 r/m	22 + m	38 + m	b	h,j,k,r
Protected Mode Only (Indirect Intersegment)					
Via Call Gate to Same Privilege Level			56 + m		h,j,k,r
Via Call Gate to Different Privilege Level, (No Parameters)			90 + m		h,j,k,r
Via Call Gate to Different Privilege Level, (x Parameters)			98 + 4x + m		h,j,k,r
From 286 Task to 286 TSS			278		h,j,k,r
From 286 Task to 386 TSS			303		h,j,k,r
From 286 Task to Virtual 8086 Task (386 TSS)			221		h,j,k,r
From 386 Task to 286 TSS			278		h,j,k,r
From 386 Task to 386 TSS			305		h,j,k,r
From 386 Task to Virtual 8086 Task (386 TSS)			221		h,j,k,r
JMP = Unconditional Jump					
Short	11101001 8-bit displacement	7 + m	7 + m		r
Direct within Segment	11101001 full displacement	7 + m	7 + m		r
Register/Memory Indirect within Segment	11111111 mod 100 r/m	7 + m/ 10 + m	7 + m/ 10 + m	b	h,r
Direct Intersegment	11101010 unsigned full offset, selector	12 + m	27 + m		j,k,r
Protected Mode Only (Direct Intersegment)					
Via Call Gate to Same Privilege Level			45 + m		h,j,k,r
From 286 Task to 286 TSS			274		h,j,k,r
From 286 Task to 386 TSS			301		h,j,k,r
From 286 Task to Virtual 8086 Task (386 TSS)			218		h,j,k,r
From 386 Task to 286 TSS			270		h,j,k,r
From 386 Task to 386 TSS			303		h,j,k,r
From 386 Task to Virtual 8086 Task (386 TSS)			220		h,j,k,r
Indirect Intersegment	11111111 mod 101 r/m	17 + m	31 + m	b	h,j,k,r
Protected Mode Only (Indirect Intersegment)					
Via Call Gate to Same Privilege Level			49 + m		h,j,k,r
From 286 Task to 286 TSS			279		h,j,k,r
From 286 Task to 386 TSS			306		h,j,k,r
From 286 Task to Virtual 8086 Task (386 TSS)			222		h,j,k,r
From 386 Task to 286 TSS			275		h,j,k,r
From 386 Task to 386 TSS			308		h,j,k,r
From 386 Task to Virtual 8086 Task (386 TSS)			224		h,j,k,r

Tableau G.1 : 80386 Jeu d'instructions Compteur de cycles

INSTRUCTION	FORMAT			COMPTEUR		NOTES	
				Mode adresse réelle ou mode virtuel 8086	Mode protégé adresse virtuelle	Mode adresse réelle ou mode virtuel 8086	Mode protégé adresse virtuelle
CONTROL TRANSFER (Continued)							
RET = Return from CALL:							
Within Segment	11000011			10 + m	10 + m	b	g, h, r
Within Segment Adding Immediate to SP	11000010	16-bit displ		10 + m	10 + m	b	g, h, r
Intersegment	11001011			18 + m	32+m	b	g, h, j, k, r
Intersegment Adding Immediate to SP	11001010	16-bit displ		18 + m	32+m	b	g, h, j, k, r
Protected Mode Only (RET):							
to Different Privilege Level							
Intersegment					68		h, j, k, r
Intersegment Adding Immediate to SP					68		h, j, k, r
CONDITIONAL JUMPS							
NOTE: Times Are Jump "Taken or Not Taken"							
JO = Jump on Overflow							
8-Bit Displacement	01110000	8-bit displ		7 + m or 3	7 + m or 3		r
Full Displacement	00001111	10000000	full displacement	7 + m or 3	7 + m or 3		r
JNO = Jump on Not Overflow							
8-Bit Displacement	01110001	8-bit displ		7 + m or 3	7 + m or 3		r
Full Displacement	00001111	10000001	full displacement	7 + m or 3	7 + m or 3		r
JB/JNAE = Jump on Below/Not Above or Equal							
8-Bit Displacement	01110010	8-bit displ		7 + m or 3	7 + m or 3		r
Full Displacement	00001111	10000010	full displacement	7 + m or 3	7 + m or 3		r
JNB/JAE = Jump on Not Below/Above or Equal							
8-Bit Displacement	01110011	8-bit displ		7 + m or 3	7 + m or 3		r
Full Displacement	00001111	10000011	full displacement	7 + m or 3	7 + m or 3		r
JE/JZ = Jump on Equal/Zero							
8-Bit Displacement	01110100	8-bit displ		7 + m or 3	7 + m or 3		r
Full Displacement	00001111	10000100	full displacement	7 + m or 3	7 + m or 3		r
JNE/JNZ = Jump on Not Equal/Not Zero							
8-Bit Displacement	01110101	8-bit displ		7 + m or 3	7 + m or 3		r
Full Displacement	00001111	10000101	full displacement	7 + m or 3	7 + m or 3		r
JBE/JNA = Jump on Below or Equal/Not Above							
8-Bit Displacement	01110110	8-bit displ		7 + m or 3	7 + m or 3		r
Full Displacement	00001111	10000110	full displacement	7 + m or 3	7 + m or 3		r
JNBE/JA = Jump on Not Below or Equal/Above							
8-Bit Displacement	01110111	8-bit displ		7 + m or 3	7 + m or 3		r
Full Displacement	00001111	10000111	full displacement	7 + m or 3	7 + m or 3		r
JS = Jump on Sign							
8-Bit Displacement	01111000	8-bit displ		7 + m or 3	7 + m or 3		r
Full Displacement	00001111	10001000	full displacement	7 + m or 3	7 + m or 3		r

Tableau G.1 : 80386 Jeu d'instructions Compteur de cycles

INSTRUCTION	FORMAT				COMPTEUR		NOTES	
					Mode adresse réelle ou mode virtuel 8086	Mode protégé adresse virtuelle	Mode adresse réelle ou mode virtuel 8086	Mode protégé adresse virtuelle
CONDITIONAL JUMPS (Continued)								
JNS = Jump on Not Sign								
8-Bit Displacement	01111001	8-bit displ			7 + m or 3	7 + m or 3		r
Full Displacement	00001111	10001001	full displacement		7 + m or 3	7 + m or 3		r
JP/JPE = Jump on Parity/Parity Even								
8-Bit Displacement	01111010	8-bit displ			7 + m or 3	7 + m or 3		r
Full Displacement	00001111	10001010	full displacement		7 + m or 3	7 + m or 3		r
JNP/JPO = Jump on Not Parity/Parity Odd								
8-Bit Displacement	01111011	8-bit displ			7 + m or 3	7 + m or 3		r
Full Displacement	00001111	10001011	full displacement		7 + m or 3	7 + m or 3		r
JL/JNGE = Jump on Less/Not Greater or Equal								
8-Bit Displacement	01111100	8-bit displ			7 + m or 3	7 + m or 3		r
Full Displacement	00001111	10001100	full displacement		7 + m or 3	7 + m or 3		r
JNL/JGE = Jump on Not Less/Greater or Equal								
8-Bit Displacement	01111101	8-bit displ			7 + m or 3	7 + m or 3		r
Full Displacement	00001111	10001101	full displacement		7 + m or 3	7 + m or 3		r
JLE/JNG = Jump on Less or Equal/Not Greater								
8-Bit Displacement	01111110	8-bit displ			7 + m or 3	7 + m or 3		r
Full Displacement	00001111	10001110	full displacement		7 + m or 3	7 + m or 3		r
JNLE/JG = Jump on Not Less or Equal/Greater								
8-Bit Displacement	01111111	8-bit displ			7 + m or 3	7 + m or 3		r
Full Displacement	00001111	10001111	full displacement		7 + m or 3	7 + m or 3		r
JCXZ = Jump on CX Zero	11100011	8-bit displ			9 + m or 5	9 + m or 5		r
JECXZ = Jump on ECX Zero	11100011	8-bit displ			9 + m or 5	9 + m or 5		r
(Address Size Prefix Differentiates JCXZ from JECXZ)								
LOOP = Loop CX Times	11100010	8-bit displ			11 + m	11 + m		r
LOOPZ/LOOPE = Loop with Zero/Equal	11100001	8-bit displ			11 + m	11 + m		r
LOOPNZ/LOOPNE = Loop While Not Zero	11100000	8-bit displ			11 + m	11 + m		r
CONDITIONAL BYTE SET								
NOTE: Times Are Register/Memory								
SETO = Set Byte on Overflow								
To Register/Memory	00001111	10010000	mod 000 r/m		4/5	4/5		h
SETNO = Set Byte on Not Overflow								
To Register/Memory	00001111	10010001	mod 000 r/m		4/5	4/5		h
SETB/SETNAE = Set Byte on Below/Not Above or Equal								
To Register/Memory	00001111	10010010	mod 000 r/m		4/5	4/5		h

Tableau G.1 : 80386 Jeu d'instructions Compteur de cycles

INSTRUCTION	FORMAT				COMPTEUR		NOTES	
					Mode adresse réelle ou mode virtuel 8086	Mode protégé adresse virtuelle	Mode adresse réelle ou mode virtuel 8086	Mode protégé adresse virtuelle
CONDITIONAL BYTE SET (Continued)								
SETNB = Set Byte on Not Below/Above or Equal								
To Register/Memory	00001111	10010011	mod000 r/m		4/5	4/5		h
SETE/SETZ = Set Byte on Equal/Zero								
To Register/Memory	00001111	10010100	mod000 r/m		4/5	4/5		h
SETNE/SETNZ = Set Byte on Not Equal/Not Zero								
To Register/Memory	00001111	10010101	mod000 r/m		4/5	4/5		h
SETBE/SETNA = Set Byte on Below or Equal/Not Above								
To Register/Memory	00001111	10010110	mod000 r/m		4/5	4/5		h
SETNBE/SETA = Set Byte on Not Below or Equal/Above								
To Register/Memory	00001111	10010111	mod000 r/m		4/5	4/5		h
SETS = Set Byte on Sign								
To Register/Memory	00001111	10011000	mod000 r/m		4/5	4/5		h
SETNS = Set Byte on Not Sign								
To Register/Memory	00001111	10011001	mod000 r/m		4/5	4/5		h
SETP/SETPE = Set Byte on Parity/Parity Even								
To Register/Memory	00001111	10011010	mod000 r/m		4/5	4/5		h
SETNP/SETPO = Set Byte on Not Parity/Parity Odd								
To Register/Memory	00001111	10011011	mod000 r/m		4/5	4/5		h
SETL/SETNGE = Set Byte on Less/Not Greater or Equal								
To Register/Memory	00001111	10011100	mod000 r/m		4/5	4/5		h
SETNL/SETGE = Set Byte on Not Less/Greater or Equal								
To Register/Memory	00001111	01111101	mod000 r/m		4/5	4/5		h
SETLE/SETNG = Set Byte on Less or Equal/Not Greater								
To Register/Memory	00001111	10011110	mod000 r/m		4/5	4/5		h
SETNLE/SETG = Set Byte on Not Less or Equal/Greater								
To Register/Memory	00001111	10011111	mod000 r/m		4/5	4/5		h
ENTER = Enter Procedure	11001000	16-bit displacement, 8-bit level						
L = 0					10	10	b	h
L = 1					12	12	b	h
L > 1					15 + 4(n − 1)	15 + 4(n − 1)	b	h
LEAVE = Leave Procedure	11001001				4	4	b	h

Tableau G.1 : 80386 Jeu d'instructions Compteur de cycles

INSTRUCTION	FORMAT	COMPTEUR		NOTES	
		Mode adresse réelle ou mode virtuel 8086	Mode protégé adresse virtuelle	Mode adresse réelle ou mode virtuel 8086	Mode protégé adresse virtuelle
INTERRUPT INSTRUCTIONS					
INT = Interrupt:					
Type Specified	11001101 type	37		b	
Type 3	11001100	33		b	
INTO = Interrupt 4 If Overflow Flag Set	11001110				
If OF = 1		35		b, e	
If OF = 0		3	3	b, e	
Bound = Interrupt 5 If Detect Value Out of Range	01100010 mod reg r/m				
If Out of Range		44		b, e	e, g, h, j, k, r
If In Range		10	10	b, e	e, g, h, j, k, r
Protected Mode Only (INT)					
INT: Type Specified					
Via Interrupt or Trap Gate to Same Privilege Level			59		g, j, k, r
Via Interrupt or Trap Gate to Different Privilege Level			99		g, j, k, r
From 286 Task to 286 TSS via Task Gate			282		g, j, k, r
From 286 Task to 386 TSS via Task Gate			309		g, j, k, r
From 286 Task to virt 8086 md via Task Gate			226		g, j, k, r
From 386 Task to 286 TSS via Task Gate			284		g, j, k, r
From 386 Task to 386 TSS via Task Gate			311		g, j, k, r
From 386 Task to virt 8086 md via Task Gate			228		g, j, k, r
From virt 8086 md to 286 TSS via Task Gate			289		g, j, k, r
From virt 8086 md to 386 TSS via Task Gate			316		g, j, k, r
From virt 8086 md to priv level 0 via Trap Gate or Interrupt Gate			119		
INT: TYPE 3					
Via Interrupt or Trap Gate to Same Privilege Level			59		g, j, k, r
Via Interrupt or Trap Gate to Different Privilege Level			99		g, j, k, r
From 286 Task to 286 TSS via Task Gate			278		g, j, k, r
From 286 Task to 386 TSS via Task Gate			305		g, j, k, r
From 286 Task to Virt 8086 md via Task Gate			222		g, j, k, r
From 386 Task to 286 TSS via Task Gate			280		g, j, k, r
From 386 Task to 386 TSS via Task Gate			307		g, j, k, r
From 386 Task to Virt 8086 md via Task Gate			224		g, j, k, r
From virt 8086 md to 286 TSS via Task Gate			285		g, j, k, r
From virt 8086 md to 386 TSS via Task Gate			312		g, j, k, r
From virt 8086 md to priv level 0 via Trap Gate or Interrupt Gate			119		
INTO:					
Via Interrupt or Trap Grate to Same Privilege Level			59		g, j, k, r
Via Interrupt or Trap Gate to Different Privilege Level			99		g, j, k, r
From 286 Task to 286 TSS via Task Gate			280		g, j, k, r
From 286 Task to 386 TSS via Task Gate			307		g, j, k, r
From 286 Task to virt 8086 md via Task Gate			224		g, j, k, r
From 386 Task to 286 TSS via Task Gate			282		g, j, k, r
From 386 Task to 386 TSS via Task Gate			309		g, j, k, r
From 386 Task to virt 8086 md via Task Gate			226		g, j, k, r
From virt 8086 md to 286 TSS via Task Gate			287		g, j, k, r
From virt 8086 md to 386 TSS via Task Gate			314		g, j, k, r
From virt 8086 md to priv level 0 via Trap Gate or Interrupt Gate			119		