
Manuel 2

Optimisation des sous-routines en langage assembleur

Un guide d'optimisation pour les plateformes x86

par Agner Fog

Copyright © 1996 - 2025. Dernière mise à jour le 18-12-2025.

Texte original faisant référence à www.agner.org/optimize

Traduction française n° 1 du 30/09/2026 (14h00)
par Calendos (France)

Sommaire

1 Introduction.....	1
1.1 Les raisons d'apprendre le langage assembleur.....	1
1.2 Les raisons de NE PAS UTILISER le langage assembleur.....	2
1.3 Systèmes d'exploitation couverts par ce manuel.....	3
2 Prérequis.....	3
2.1 Points à clarifier avant de commencer à programmer	3
2.2 Élaboration d'une stratégie de test.....	4
2.3 Pièges courants en programmation	5
3 Bases de la programmation en assembleur.....	7
3.1 Assembleurs disponibles	7
MASM.....	7
GAS	7
NASM	7
YASM	7
FASM	7
UASM	7
HLA.....	8
TASM.....	8
Assembleurs Inline	8
Fonctions intrinsèques en C++	8
Quel assembleur choisir ?	8
3.2 Jeu de registres et instructions de base.....	8
Registres en mode 16 bits	8
Registres en mode 32 bits	9
Registres en mode 64 bits	10
3.3 Modes d'adressage.....	12
Adressage en mode 16 bits.....	12
Adressage en mode 32 bits.....	13
Code indépendant de la position en mode 32 bits	13
Adressage en mode 64 bits.....	15
3.4 Format du code d'instruction	18
3.5 Préfixes d'instruction.....	19
4 Normes ABI.....	20
4.1 Utilisation des registres	20
4.2 Stockage de données.....	21
4.3 Conventions d'appel de fonction.....	21
Convention d'appel en mode 16 bits sous DOS et Windows 3.x.....	21
Convention d'appel sous Windows 32 bits, Linux, BSD et Mac OS X	22
Conventions d'appel sous Windows 64 bits	22
Conventions d'appel sous Linux, BSD et Mac OS X 64 bits	23
4.4 Altération et décoration de noms	23
4.5 Exemples de fonctions.....	23

5 Utilisation des fonctions intrinsèques en C++	25
5.1 Utilisation de fonctions intrinsèques pour le code système	26
5.2 Utilisation de fonctions intrinsèques pour des instructions non disponibles en C++ standard	27
5.3 Utilisation de fonctions intrinsèques pour les opérations vectorielles	27
5.4 Disponibilité des fonctions intrinsèques	27
6 Utilisation de l'assembleur en ligne.....	27
6.1 Assembleur en ligne de style MASM	28
Accès aux variables de registre	29
Accès aux membres de classe et aux membres de structure	30
Appel de fonctions	31
Vue d'ensemble de la syntaxe	31
Compilateurs utilisant un assembleur en ligne de style MASM	32
6.2 Assemblage en ligne de style GNU	32
Syntaxe AT&T	32
Syntaxe Intel	33
7 Utilisation d'un assembleur	34
7.1 Bibliothèques de liens statiques	35
7.2 Bibliothèques de liens dynamiques	36
7.3 Bibliothèques d'objets partagés.....	36
7.4 Bibliothèques sous forme de code source	36
7.5 Création de classes en langage assembleur	37
7.6 Fonctions thread-safe.....	38
7.7 Makefiles.....	38
8 Rendre les bibliothèques de fonctions compatibles avec plusieurs compilateurs et plateformes	39
8.1 Prise en charge de plusieurs schémas de transformation de noms.....	40
8.2 Prise en charge de plusieurs conventions d'appel en mode 32 bits.....	41
8.3 Prise en charge de plusieurs conventions d'appel en mode 64 bits.....	44
8.4 Prise en charge des différents formats de fichiers objets.....	45
8.5 Prise en charge d'autres langages de haut niveau	46
9 Optimisation de la vitesse.....	46
9.1 Identification des parties les plus critiques de votre code.	46
9.2 Exécution dans le désordre	47
Registres partiels	47
Micro-opérations.....	48
Unités d'exécution.....	48
Instructions traitées en pipeline	48
Résumé	48
9.3 Récupération, décodage et validation des instructions.....	49
9.4 Latence et débit des instructions	49
9.5 Rupture des chaînes de dépendance.....	50
9.6 Jumps et calls	51
Faites en sorte que les sauts conditionnels ne soient le plus souvent pas effectués.	52
Appels terminaux	53

Élimination des sauts inconditionnels	53
Remplacement des sauts conditionnels par des déplacements conditionnels (CMOV)	54
Remplacement des sauts conditionnels par des instructions SET conditionnel	55
Remplacement des sauts conditionnels par des instructions de manipulation de bits	56
10 Optimisation de la taille	57
10.1 Choix d'instructions plus courtes	57
10.2 Utilisation de constantes et d'adresses plus courtes.....	58
10.3 Réutilisation de constantes	59
10.4 Constantes en mode 64 bits	59
10.5 Adresses et pointeurs en mode 64 bits	60
10.6 Allongement des instructions pour assurer l'alignement	61
Utilisation de la forme générale plutôt que la forme abrégée d'une instruction	62
Utilisation d'une instruction équivalente plus longue.....	62
Utilisation d'un opérande immédiat de 4 octets	62
Ajout d'un déplacement nul au pointeur.....	62
Utilisation de l'octet SIB	62
Utilisation de préfixes.....	62
10.7 Utilisation de NOPs multi-octets pour l'alignement.....	63
11 Optimisation de l'accès à la mémoire	64
11.1 Fonctionnement de la mise en cache.....	64
11.2 Cache de traces.....	64
11.3 Cache de micro-opérations (μop).....	65
11.4 Alignement des données	65
11.5 Alignement du code.....	67
11.6 Organisation des données pour une mise en cache améliorée	68
11.7 Organisation du code pour une mise en cache améliorée.....	68
11.8 Instructions de contrôle du cache.....	69
12 Boucles.....	69
12.1 Minimisation de la surcharge de la boucle.....	69
12.2 Variables d'induction	71
12.3 Déplacement du code invariant de boucle.....	72
12.4 Identification des goulots d'étranglement.....	72
12.5 Optimisation pour l'exécution dans le désordre	72
12.6 Récupération, décodage et validation des instructions dans une boucle.....	73
12.7 Répartition uniforme des micro-opérations entre les unités d'exécution	73
12.8 Un exemple d'analyse des goulots d'étranglement dans les boucles vectorielles.....	73
12.9 Même exemple avec FMA3.....	75
12.10 Même exemple avec AVX512.....	75
12.11 Déroulage de boucle	76
12.12 Boucles vectorielles utilisant des registres de masque (AVX-512).....	78
12.13 Optimisation de la mise en cache.....	80
12.14 Parallélisation	80
12.15 Boucles de macros	82

13 Programmation vectorielle	84
13.1 Utilisation du jeu d'instructions AVX et des registres YMM ou ZMM.....	85
13.2 Mélange de code VEX et SSE	85
Recommandations pour mélanger du code VEX et non-VEX	87
Temps de préchauffage	88
Prise en charge des systèmes d'exploitation.....	88
YMM / ZMM et code système.....	88
Utilisation d'instructions non destructives à trois opérandes	88
Prise en charge par le compilateur.....	89
Instructions de multiplication-addition fusionnée (FMA)	89
Exemples	89
13.3 Utilisation du jeu d'instructions AVX-512 et des registres ZMM	89
13.4 Déplacements conditionnels dans les registres xmm et ymm.....	90
13.5 Déplacements conditionnels avec AVX-512	92
13.6 Utilisation d'instructions vectorielles avec des types de données autres que ceux pour lesquels elles sont conçues.....	94
En utilisant l'instruction la plus courte	95
En utilisant l'instruction la plus efficace	96
Utilisation d'une instruction qui n'est pas disponible pour d'autres types de données	96
13.7 Permutation de données.....	96
13.8 Génération de constantes	100
13.9 Accès à des données non alignées et à des vecteurs partiels	101
Utilisation d'instructions de lecture non alignées	102
Utilisation d'instructions préfixées par VEX	102
Lectures se chevauchant partiellement.....	102
Lecture à partir de la limite de 16 octets précédente la plus proche	102
Combinaison de 2 parties de vecteur non alignées en un seul vecteur aligné	103
Utilisation d'instructions d'écriture non alignées	104
Écritures se chevauchant partiellement	104
Écrire du début et de la fin séparément	104
Utilisation de l'écriture masquée.....	104
13.10 Opérations vectorielles dans les registres d'usage général	104
14 Multithread.....	106
14.1 Multithreading simultané.....	106
15 Ordonnancement du processeur	107
15.1 Vérification de la prise en charge des registres XMM, YMM et ZMM par le système d'exploitation ..	108
16 Instructions problématiques	109
16.1 Instruction LEA (tous les processeurs)	109
16.2 INC et DEC	110
16.3 XCHG (tous les processeurs).....	110
16.4 Rotation via la retenue (tous les processeurs).....	110
16.5 Test de bit (tous les processeurs)	110
16.6 LAHF et SAHF (tous les processeurs).....	110
16.7 Multiplication d'entiers (tous les processeurs)	110
16.8 Division (tous les processeurs)	110

Division entière par une puissance de 2 (tous les processeurs)	110
Division entière par une constante (tous les processeurs)	111
Division entière répétée par la même valeur (tous les processeurs)	112
Division en virgule flottante (tous les processeurs)	113
Utilisation de l'instruction réciproque pour la division rapide	113
16.9 Instructions de manipulation de chaînes (tous les processeurs)	113
16.10 Instructions de chaîne vectorisées (processeurs avec SSE4.2)	114
16.11 Instruction WAIT (tous les processeurs)	114
16.12 FCOM + FSTSW AX (tous les processeurs)	115
16.13 FPREM (tous les processeurs)	115
16.14 FRNDINT (tous les processeurs)	115
16.15 FSCALE et fonction exponentielle (tous les processeurs)	115
16.16 FPTAN (tous les processeurs)	116
16.17 FSQRT, SQRSS	116
16.18 FLDCW	116
16.19 Instructions MASKMOV	117
16.20 Instructions RDRAND et RDSEED	117
17 Sujets particuliers	117
17.1 XMM contre registres à virgule flottante	117
17.2 Registres MMX contre XMM	118
17.3 Registres XMM par rapport aux registres YMM et ZMM	118
17.4 Libération des registres à virgule flottante	118
17.5 Transitions entre les instructions à virgule flottante et les instructions MMX	118
17.6 Conversion de virgule flottante en entier	118
17.7 Utilisation d'instructions sur entiers pour des opérations en virgule flottante	119
Test si une valeur à virgule flottante est égale à zéro	119
Manipulation du bit de signe	120
Manipulation de l'exposant	120
Manipulation de la mantisse	120
Comparaison de nombres	121
17.8 Déplacement de blocs de données	121
Préchargement (Prefetching)	123
Transfert de données sur les futurs processeurs	123
17.9 Code auto-modifiable	123
18 Mesure de performance	124
18.1 Test de vitesse	124
18.2 Les pièges des tests unitaires	125
19 Bibliographie	125
20 Copyright	125

1 Introduction

Il s'agit du deuxième d'une série de cinq manuels :

1. Optimisation des logiciels en C++ : guide d'optimisation pour les plates-formes Windows, Linux et Mac.
2. Optimisation de sous-programmes en langage assembleur : guide d'optimisation pour les plateformes x86.
3. La microarchitecture des processeurs Intel, AMD et VIA : guide d'optimisation pour les programmeurs en assembleur et les concepteurs de compilateurs.
4. Tableaux d'instructions : listes des latences d'instructions, des débits et des pannes de micro-opérations pour les processeurs Intel, AMD et VIA.
5. Conventions d'appel pour différents compilateurs C++ et systèmes d'exploitation.

Les versions les plus récentes de ces manuels sont toujours disponibles sur www.agner.org/optimize. Les conditions relatives aux droits d'auteur sont décrites à la page 125 à la fin de ce document.

Ce manuel explique comment combiner du code assembleur avec un langage de programmation de haut niveau et comment optimiser la vitesse d'un code sollicitant fortement le processeur en utilisant de l'assembleur.

Il s'adresse aux programmeurs en assembleur expérimentés ainsi qu'aux concepteurs de compilateurs. Il est supposé ici que le lecteur possède une bonne maîtrise du langage assembleur et une certaine expérience de la programmation dans ce langage. Il est conseillé aux débutants de se documenter par ailleurs et d'acquérir de l'expérience en programmation avant de tenter de mettre en œuvre les techniques d'optimisation décrites ici. Je recommande de consulter les diverses introductions, tutoriels, forums de discussion et groupes de nouvelles disponibles sur Internet (voir les liens sur www.agner.org/optimize), ainsi que des ouvrages d'introduction tels que "Introduction to 80x86 Assembly Language and Computer Architecture" de R. C. Detmer (3e édition, 2014).

Le présent manuel couvre toutes les plateformes utilisant les jeux d'instructions x86 et x86-64. Ces jeux d'instructions sont utilisés par la plupart des microprocesseurs d'Intel, d'AMD et de VIA. Parmi les systèmes d'exploitation compatibles, on compte DOS, Windows, Linux, FreeBSD/OpenBSD et Mac OS pour processeurs Intel. Ce manuel traite des microprocesseurs et des jeux d'instructions les plus récents. Consultez les manuels 3 et 4 pour plus de détails sur des modèles de microprocesseurs spécifiques.

Les techniques d'optimisation qui ne sont pas propres au langage assembleur sont abordées dans le manuel 1 : "Optimizing software in C++". Les détails spécifiques à un microprocesseur donné sont traités dans le manuel 3 : "The microarchitecture of Intel, AMD, and VIA CPUs". Des tableaux indiquant les temps d'exécution des instructions, etc., figurent dans le manuel 4 : "Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs". Les conventions d'appel propres aux différents systèmes d'exploitation et compilateurs sont détaillées dans le manuel 5 : "Calling conventions for different C++ compilers and operating systems".

La programmation en langage assembleur est bien plus complexe que celle en langage de haut niveau. Il est très facile d'introduire des bugs, mais très difficile de les déboguer. Vous voilà prévenus ! Merci de ne pas m'envoyer vos questions de programmation ; je n'y répondrai pas. Il existe divers forums de discussion sur Internet où vous pourrez trouver des réponses à vos questions si vous ne les trouvez pas dans les ouvrages et manuels de référence.

Bonne chance dans votre quête de nanosecondes !

1.1 Les raisons d'apprendre le langage assembleur

La programmation en langage assembleur n'est plus aussi répandue qu'auparavant. Il existe toutefois de bonnes raisons d'apprendre et d'utiliser ce langage. Les principales sont les suivantes :

1. Raisons pédagogiques. Il est important de comprendre le fonctionnement des microprocesseurs et des compilateurs au niveau des instructions afin de pouvoir prédire quelles techniques de codage sont les plus efficaces, de saisir le fonctionnement des diverses constructions des langages de haut niveau et de traquer les erreurs difficiles à déceler.
2. Débogage et vérification. L'examen du code assembleur généré par le compilateur ou de la fenêtre de désassemblage d'un débogueur est utile pour repérer des erreurs et évaluer l'efficacité avec laquelle le compilateur optimise un segment de code donné.
3. Création de compilateurs. La maîtrise des techniques de programmation en langage assembleur est nécessaire pour concevoir des compilateurs, des débogueurs et d'autres outils de développement.
4. Systèmes embarqués. Les petits systèmes embarqués disposent de moins de ressources que les PC et les ordinateurs centraux. La programmation en langage d'assemblage peut s'avérer nécessaire pour optimiser le code en termes de vitesse ou de taille dans ces systèmes.
5. Pilotes de matériels et code système. L'accès aux ports matériels, aux registres de contrôle du système, etc., s'effectue traditionnellement en langage assembleur. Toutefois, cela est rarement nécessaire aujourd'hui.

d'hui, car la plupart des instructions système sont désormais accessibles depuis du code C++ via des fonctions intrinsèques (voir p. 25).

6. Code auto-modifiable. L'utilisation de code auto-modifiable est généralement déconseillée, car elle nuit à l'efficacité de la mise en cache du code. Il peut toutefois s'avérer avantageux, par exemple, d'intégrer un compilateur minimal dans des programmes de calcul mathématique lorsque l'on doit évaluer à maintes reprises une fonction définie par l'utilisateur.
7. Optimisation de la taille du code. L'espace de stockage et la mémoire sont aujourd'hui si peu coûteux qu'il ne vaut pas la peine de recourir au langage d'assemblage pour réduire la taille du code. Toutefois, la taille du cache reste une ressource si critique qu'il peut s'avérer utile, dans certains cas, d'optimiser la taille d'une portion de code critique afin de la faire tenir dans le cache d'instructions ou le cache de micro-opérations (cache μop).
8. Optimisation du code pour la vitesse. Auparavant, il était possible d'améliorer considérablement les performances d'un segment de code en recourant à l'assembleur. Aujourd'hui, toutefois, les meilleurs compilateurs C++, tels que GCC et Clang, optimisent généralement le code mieux qu'un programmeur assembleur expérimenté ne saurait le faire. Quoi qu'il en soit, l'apprentissage de l'assembleur reste utile pour examiner le code généré par le compilateur et vérifier si l'optimisation s'est effectuée comme prévu. Il existe souvent des obstacles empêchant le compilateur d'optimiser le code, tels que l'aliasing de pointeurs, les problèmes d'alignement, ou encore la gestion des exceptions qui entrave la vectorisation (voir le manuel 1 : "Optimizing software in C++"). Ces problèmes peuvent être identifiés en examinant le code assembleur produit par le compilateur ou le désassemblage généré par un débogueur. Une fois identifiés, la plupart de ces obstacles à l'optimisation peuvent être levés grâce à des directives de compilation appropriées.
9. Utilisation des registres vectoriels. Le code assembleur permet d'utiliser des registres et des instructions vectoriels. Toutefois, il est souvent plus efficace de recourir à des fonctions intrinsèques ou à une bibliothèque de classes vectorielles pour la programmation vectorielle (voir le manuel 1 : Optimisation des logiciels en C++).
10. Bibliothèques de fonctions. L'avantage total de l'optimisation du code est plus élevé dans les bibliothèques de fonctions utilisées par de nombreux programmes.
11. Résolution de problèmes de compatibilité. Différents compilateurs, plateformes ou systèmes d'exploitation peuvent employer des conventions d'appel de fonctions ou des mécanismes de « name mangling » (codage des noms) distincts (voir le manuel 5 : Conventions d'appel). Il est parfois possible de résoudre ces problèmes de compatibilité en créant, en langage assembleur, une interface assurant la liaison entre l'appelant et la fonction appelée. L'assembleur permet également de concevoir une fonction dotée de plusieurs points d'entrée correspondant à différentes conventions d'appel. Toutefois, la programmation en assembleur ne permet pas de résoudre les problèmes liés à des jeux d'instructions ou à des fonctions système incompatibles.
12. Rétro-ingénierie. Dans des cas simples, il est parfois possible d'étudier le fonctionnement d'un segment de code en examinant son désassemblage lorsque le code source n'est pas disponible. (La rétro-ingénierie peut être considérée comme une violation du droit d'auteur dans certains cas, selon l'objectif visé et la juridiction. Si elle a pour but de créer un code similaire ou de contourner une protection contre la copie, elle sera probablement jugée illégale. Toutefois, la rétro-ingénierie visant à assurer l'« interopérabilité » entre différents produits est explicitement autorisée dans plusieurs juridictions.)

Ce manuel met principalement l'accent sur l'optimisation du code en termes de vitesse, bien que d'autres sujets soient également abordés.

1.2 Les raisons de NE PAS UTILISER le langage assembleur

La programmation en langage assembleur présente tellement d'inconvénients et de problèmes qu'il est conseillé d'envisager les alternatives avant de décider d'utiliser du code assembleur pour une tâche donnée. Les principales raisons de ne pas recourir à la programmation en assembleur sont les suivantes :

1. Durée de développement. L'écriture de code en langage assembleur prend beaucoup plus de temps qu'en langage de haut niveau.
2. Fiabilité et sécurité. Il est facile de commettre des erreurs dans le code assembleur. L'assembleur ne vérifie pas si les conventions d'appel et les conventions de sauvegarde des registres sont respectées. Personne ne vérifie pour vous si le nombre d'instructions PUSH et POP est identique dans toutes les branches et tous les chemins d'exécution possibles. Les possibilités d'erreurs cachées dans le code assembleur sont si nombreuses qu'elles affectent la fiabilité et la sécurité du projet, à moins d'adopter une approche très systématique en matière de test et de vérification.
3. Débogage et vérification. Le code assembleur est plus difficile à déboguer et à vérifier, car les sources d'erreurs y sont plus nombreuses que dans le code de haut niveau.

4. **Maintenabilité.** Le code assembleur est plus difficile à modifier et à maintenir, car le langage permet une structure désordonnée (type « code spaghetti ») ainsi que toutes sortes d'astuces peu orthodoxes difficiles à comprendre pour autrui. Une documentation approfondie et un style de programmation cohérent sont nécessaires.
5. Le code système peut utiliser des fonctions intrinsèques plutôt que de l'assembleur. Les compilateurs C++ modernes disposent de fonctions intrinsèques pour accéder aux registres de contrôle du système et à d'autres instructions système (voir p. 25). Le recours à l'assembleur n'est plus nécessaire pour les pilotes de périphériques et autres codes système lorsque des fonctions intrinsèques sont disponibles.
6. Le code applicatif peut utiliser des fonctions intrinsèques ou des classes vectorielles plutôt que de l'assembleur. Les compilateurs C++ modernes proposent des fonctions intrinsèques pour les opérations vectorielles et d'autres instructions spéciales qui nécessitaient auparavant une programmation en assembleur. Il n'est plus nécessaire d'écrire du code assembleur pour tirer parti des instructions SIMD (*Single Instruction, Multiple Data* / instruction unique, données multiples) (voir page 25).
7. **Portabilité.** Le code assembleur est très spécifique à une plateforme donnée. Le portage vers une autre plateforme est difficile. Le code utilisant des fonctions intrinsèques plutôt que de l'assembleur est portable sur toutes les plateformes x86 et x86-64.
8. Les compilateurs ont considérablement progressé au cours des dernières décennies. Dans la plupart des situations, les meilleurs compilateurs surpassent désormais un programmeur assembleur expérimenté.
9. Le code compilé peut être plus rapide que le code assembleur, car les compilateurs sont capables d'effectuer des optimisations inter-procédurales et des optimisations à l'échelle du programme entier. En général, pour rendre le code testable et vérifiable, le programmeur assembleur doit concevoir des fonctions bien définies, dotées d'interfaces d'appel claires et respectant toutes les conventions d'appel en vigueur. Cela empêche l'application de nombreuses méthodes d'optimisation utilisées par les compilateurs, telles que l'expansion en ligne des fonctions, l'allocation de registres, la propagation de constantes, l'élimination des sous-expressions communes entre fonctions, l'ordonnancement entre fonctions, etc. Ces avantages peuvent être obtenus en utilisant du code C++ avec des fonctions intrinsèques ou des classes vectorielles plutôt que du code assembleur.

1.3 Systèmes d'exploitation couverts par ce manuel

Les systèmes d'exploitation suivants peuvent utiliser des microprocesseurs de la famille x86 :

- 16 bits : DOS, Windows 3.x.
- 32 bits : Windows, Linux, FreeBSD, OpenBSD, NetBSD, Intel-based Mac OS X.
- 64 bits : Windows, Linux, FreeBSD, OpenBSD, NetBSD, Intel-based Mac OS X.

Tous les systèmes d'exploitation de type UNIX (Linux, BSD, Mac OS) utilisent les mêmes conventions d'appel, à de très rares exceptions près. Tout ce qui est dit dans ce manuel à propos de Linux s'applique également aux autres systèmes de type UNIX, y compris, éventuellement, à des systèmes non mentionnés ici.

2 Prérequis

2.1 Points à clarifier avant de commencer à programmer

Avant de commencer à programmer en assembleur, vous devez réfléchir aux raisons pour lesquelles vous souhaitez utiliser ce langage, à la partie de votre programme que vous devez développer en assembleur et à la méthode de programmation à adopter. Si votre stratégie de développement n'est pas clairement définie, vous risquez rapidement de perdre du temps à optimiser les mauvaises parties du programme, à réaliser en assembleur des opérations qui auraient pu l'être en C++, à tenter d'optimiser des éléments qui ne peuvent plus l'être, ou encore à produire un code « spaghetti » difficile à maintenir, voire un code truffé d'erreurs et ardu à déboguer.

Voici une liste de points à prendre en compte avant de commencer la programmation :

- Ne développez jamais l'intégralité d'un programme en langage assembleur ; ce serait une perte de temps. L'assembleur ne doit être utilisé que lorsque la vitesse est critique et qu'une amélioration significative des performances peut être obtenue. La majeure partie du programme doit être écrite en C ou en C++, car ce sont les langages qui s'interfacent le plus facilement avec l'assembleur.
- Si vous utilisez l'assembleur pour écrire du code système ou pour exploiter des instructions spécifiques indisponibles en C++ standard, isolez la partie du programme nécessitant ces instructions dans une fonction ou une classe distincte aux fonctionnalités bien définies. Privilégiez l'utilisation de fonctions intrinsèques (voir p. 24) lorsque cela est possible.
- Si l'objectif est d'optimiser la vitesse d'exécution, identifiez la partie du programme la plus gourmande en ressources processeur, éventuellement à l'aide d'un profileur. Déterminez si le goulot d'étranglement provient des accès aux fichiers, des accès mémoire, des instructions processeur ou d'un autre facteur, comme

indiqué dans le manuel 1 : “Optimizing software in C++”. Isolez ensuite cette partie critique dans une fonction ou une classe aux fonctionnalités bien définies.

- Si vous utilisez l'assembleur pour créer une bibliothèque de fonctions, définissez-en clairement les fonctionnalités. Choisissez entre une bibliothèque de fonctions et une bibliothèque de classes. Décidez également du mode de liaison : statique (fichiers .lib sous Windows, .a sous Linux) ou dynamique (fichiers .dll sous Windows, .so sous Linux). La liaison statique est généralement plus efficace, mais la liaison dynamique peut s'avérer nécessaire si la bibliothèque doit être appelée depuis des langages autres que le C ou le C++. Il est également possible de fournir les deux versions (statique et dynamique) de la bibliothèque.
- Si l'objectif du recours à l'assembleur est d'optimiser une application embarquée en termes de taille ou de vitesse, choisissez un outil de développement prenant en charge à la fois le C/C++ et l'assembleur, et développez la majeure partie du code en C ou C++.
- Déterminez si le code est réutilisable ou spécifique à l'application. Une optimisation minutieuse se justifie davantage si le code est réutilisable. Il est préférable d'implémenter un code réutilisable sous forme de bibliothèque de fonctions ou de classes.
- Déterminez si le code doit prendre en charge le multithreading. Une application multithread peut tirer parti des microprocesseurs multi-cœurs. Les données devant être conservées d'un appel de fonction à l'autre pour un thread donné doivent être stockées dans une classe C++ ou dans une mémoire tampon spécifique au thread, fournie par le programme appelant.
- Déterminez si la portabilité est importante pour votre application. L'application doit-elle fonctionner sous Windows, Linux et macOS (sur processeurs Intel) ? Doit-elle fonctionner en modes 32 bits et 64 bits ? Doit-elle fonctionner sur des plateformes autres que x86 ? Ces questions sont déterminantes pour le choix du compilateur, de l'assembleur et de la méthode de programmation.
- Déterminez si votre application doit fonctionner sur d'anciens microprocesseurs. Si tel est le cas, vous pouvez créer une version pour les microprocesseurs dotés, par exemple, du jeu d'instructions AVX512, et une autre compatible avec les anciens modèles. Vous pouvez même produire plusieurs versions, chacune optimisée pour un processeur spécifique. Il est recommandé de mettre en place une sélection automatique du processeur (voir page 100).
- Il existe trois méthodes de programmation en assembleur au choix : (1) utiliser des fonctions intrinsèques et des classes vectorielles dans un compilateur C++ ; (2) utiliser de l'assembleur en ligne (*inline assembly*) dans un compilateur C++ ; (3) utiliser un assembleur distinct. Ces trois méthodes, ainsi que leurs avantages et inconvénients respectifs, sont décrites dans les chapitres 5, 6 et 7 (pages 25, 27 et 34, respectivement).
- Si vous utilisez un assembleur, vous devrez choisir entre différents dialectes de syntaxe. Il est souvent préférable d'utiliser un assembleur compatible avec le code assembleur que votre compilateur C++ est capable de générer.
- Commencez par écrire votre code en C++ et optimisez-le autant que possible en appliquant les méthodes décrites dans le manuel n° 1 : “Optimizing software in C++”. Demandez au compilateur de traduire le code en assembleur. Examinez le code généré par le compilateur pour identifier d'éventuelles pistes d'amélioration.
- Un code hautement optimisé est souvent très difficile à lire et à comprendre, tant pour les autres que pour vous-même lorsque vous y revenez après un certain temps. Pour garantir la maintenabilité du code, il est important de l'organiser en petites unités logiques (procédures ou macros) dotées d'interfaces et de conventions d'appel bien définies, ainsi que de commentaires appropriés. Adoptez une stratégie cohérente pour les commentaires et la documentation du code.
- Conservez le compilateur, l'assembleur et tous les autres outils de développement, ainsi que le code source et les fichiers du projet, en vue d'une maintenance ultérieure. Il est possible que des outils compatibles ne soient plus disponibles dans quelques années, au moment où des mises à jour ou des modifications du code s'avéreront nécessaires.

2.2 Élaboration d'une stratégie de test

Comme je l'ai déjà mentionné, le code assembleur est sujet aux erreurs, difficile à déboguer, ardu à structurer clairement, difficile à lire et complexe à maintenir. Une stratégie de test cohérente peut atténuer certains de ces problèmes et vous faire gagner beaucoup de temps.

Je recommande de concevoir le code assembleur sous forme de module, de fonction, de classe ou de bibliothèque isolé, doté d'une interface bien définie pour communiquer avec le programme appelant. Commencez par tout réaliser en C++. Créez ensuite un programme de test capable de vérifier tous les aspects du code que vous souhaitez optimiser. Il est plus simple et plus sûr d'utiliser un programme de test plutôt que de tester le module directement au sein de l'application finale.

Le programme de test remplit deux objectifs. Le premier consiste à vérifier que le code assembleur fonctionne correctement dans toutes les situations. Le second est de mesurer la vitesse d'exécution du code assembleur sans faire intervenir l'interface utilisateur, les accès aux fichiers ou d'autres composants de l'application finale susceptibles de rendre les mesures de vitesse moins précises et moins reproductibles.

Vous devrez utiliser ce programme de test de manière répétée, après chaque étape du processus de développement ainsi qu'après chaque modification du code.

Assurez-vous que le programme de test fonctionne correctement. Il est fréquent de passer beaucoup de temps à chercher une erreur dans le code testé alors qu'elle se trouve en réalité dans le programme de test.

Il existe différentes méthodes pour vérifier le bon fonctionnement du code. Le test en "boîte blanche" utilise une série soigneusement sélectionnée de jeux de données d'entrée pour garantir que toutes les branches, tous les chemins d'exécution et tous les cas particuliers du code sont testés. Le test en "boîte noire" utilise une série de données d'entrée aléatoires et vérifie l'exactitude de la sortie. Une très longue série de données aléatoires, générée par un bon générateur de nombres aléatoires, peut parfois révéler des erreurs rares que le test en boîte blanche n'avait pas détectées.

Le programme de test peut comparer la sortie du code assembleur à celle d'une implémentation en C++ pour en vérifier l'exactitude. Le test doit couvrir tous les cas limites et, de préférence, inclure des données d'entrée invalides afin de vérifier si le code génère les réponses d'erreur appropriées.

Le test de performance doit utiliser un jeu de données d'entrée réaliste. Une part importante du temps processeur peut être consacrée à la gestion des erreurs de prédiction de branchement dans un code comportant de nombreuses instructions de branchement. La fréquence de ces erreurs dépend du degré d'aléa des données d'entrée. Vous pouvez faire varier ce degré d'aléa pour observer son impact sur le temps de calcul, puis choisir une valeur réaliste correspondant à une application réelle typique.

Un programme de test automatisé fournissant un flux important de données de test permettra généralement de détecter davantage d'erreurs, et ce, beaucoup plus rapidement qu'un test effectué au sein de l'application finale. Bien qu'un bon programme de test permette de déceler la plupart des erreurs, il ne garantit pas leur détection exhaustive ; certaines erreurs peuvent en effet n'apparaître que lors de l'intégration au sein de l'application finale.

2.3 Pièges courants en programmation

La liste suivante met en évidence certaines des erreurs de programmation les plus courantes dans un code assembleur :

1. Oubli de sauvegarder les registres. Certains registres doivent être sauvegardés par la fonction appelée (dits "callee-save"), comme EBX. Ces registres doivent être sauvegardés dans le prologue de la fonction et restaurés dans l'épilogue s'ils sont modifiés au sein de la fonction. N'oubliez pas que l'ordre des instructions POP doit être l'inverse de celui des instructions PUSH. Consultez la page 20 pour obtenir la liste des registres de ce type.
2. Instructions PUSH et POP non appariées. Le nombre d'instructions PUSH et POP doit être identique pour tous les chemins d'exécution possibles au sein d'une fonction. Exemple :

Exemple 2.1. PUSH/POP non appariés

```
push ebx
test ecx, ecx
jz Finished
...
pop ebx
Finished ; Faux ! L'étiquette doit être avant pop ebx
ret
```

Ici, la valeur d'EBX qui a été empilée n'est pas dépilée si ECX est nul. Il en résulte que l'instruction RET dépilera l'ancienne valeur d'EBX et sautera vers une adresse erronée.

3. Utilisation d'un registre réservé à un autre usage. Certains compilateurs réservent l'utilisation de EBP ou de EBX pour un pointeur de trame ou à d'autres fins. L'utilisation de ces registres à une autre fin dans de l'assembleur en ligne peut provoquer des erreurs.
4. Adressage relatif à la pile après une opération PUSH. Lors de l'adressage d'une variable par rapport au pointeur de pile, vous devez prendre en compte toutes les instructions précédentes qui modifient ce pointeur. Exemple :

Exemple 2.2. Adressage relatif de la pile

```
mov [esp+4], edi
push ebp
push ebx
cmp esi, [esp+4] ; Faux !
```

Ici, le programmeur avait probablement l'intention de comparer ESI à EDI, mais la valeur d'ESP utilisée pour l'adressage a été modifiée par les deux instructions PUSH ; par conséquent, ESI est comparé en fait à EBP.

5. Confusion entre la valeur et l'adresse d'une variable. Exemple :

Exemple 2.3. Valeur contre adresse — section .data

; Syntaxe NASM:

MyVariable DD 0 ; Définition de la variable

section .text

mov eax, [MyVariable] ; Récupération de la valeur de MyVariable

mov eax, MyVariable ; Récupération de l'adresse de MyVariable

lea eax, [MyVariable] ; Récupération de l'adresse de MyVariable

mov ebx, [eax] ; Récupération de la valeur de MyVariable au travers du pointeur

; La syntaxe MASM est encore plus déroutante :

mov eax, MyVariable ; Récupération de la valeur de MyVariable

mov eax, offset MyVariable ; Récupération de l'adresse de MyVariable

lea eax, MyVariable ; Récupération de l'adresse de MyVariable

mov ebx, [eax] ; Récupération de la valeur de MyVariable au travers du pointeur

mov ebx, [100] ; Gets the constant 100 despite brackets

mov ebx, ds:[100] ; Récupération de la valeur à partir de l'adresse 100

6. Absence de prise en compte des conventions d'appel. Il est important de respecter les conventions d'appel des fonctions, telles que l'ordre des paramètres, le mode de transfert des paramètres (sur la pile ou dans des registres) et la responsabilité du nettoyage de la pile (incombant à l'appelant ou à la fonction appelée). Voir page 20.
7. Décoration de noms de fonctions. Un code C++ appelant une fonction en assembleur doit utiliser extern "C" pour éviter la décoration des noms. Certains systèmes exigent qu'un symbole de soulignement (_) soit placé devant le nom dans le code assembleur. Voir page 23.
8. Omission de l'instruction de retour. Une déclaration de procédure doit se terminer à la fois par RET et par ENDP ; l'utilisation d'une seule de ces instructions ne suffit pas. En l'absence de RET, l'exécution se poursuivra dans le code situé après la procédure.
9. Oubli de l'alignement de la pile. Le pointeur de pile doit pointer vers une adresse divisible par 16 avant toute instruction call, sauf dans les systèmes 16 bits et sous Windows 32 bits. Voir page 20.
10. Omission du "shadow space" sous Windows 64 bits. Il est nécessaire de réserver 32 octets d'espace vide sur la pile avant tout appel de fonction sous Windows 64 bits. Voir page **Erreur ! Signet non défini.**
11. Mélange de conventions d'appel. Les conventions d'appel sous Windows 64 bits et Linux 64 bits sont différentes. Voir page 21.
12. Omission du nettoyage de la pile de registres à virgule flottante. Tous les registres de la pile à virgule flottante utilisés par une fonction doivent être libérés — généralement à l'aide de l'instruction FSTP ST(0) — avant que la fonction ne rende la main, sauf si ST(0) sert à renvoyer une valeur. Il est nécessaire de suivre précisément le nombre de registres à virgule flottante utilisés. Si une fonction empile plus de valeurs sur la pile de registres qu'elle n'en dépile, la taille de la pile augmentera à chaque appel de la fonction. Une exception est générée lorsque la pile est pleine ; cette exception peut survenir à un autre endroit du programme.
13. Omission de la réinitialisation de l'état MMX. Une fonction utilisant des registres MMX doit les réinitialiser à l'aide de l'instruction EMMS avant tout appel ou retour.
14. Omission de la réinitialisation de l'état YMM. Une fonction utilisant des registres YMM ou ZMM doit les réinitialiser à l'aide de l'instruction VZERoupper ou VZEROALL avant tout appel ou retour.
15. Oubli de remise à zéro du flag de direction. Toute fonction qui met à 1 le flag de direction à l'aide de l'instruction STD doit le remettre à zéro à l'aide de l'instruction CLD avant tout appel ou retour.
16. Mélange d'entiers signés et non signés. Les entiers non signés sont comparés à l'aide des instructions JB et JA. Les entiers signés sont comparés à l'aide des instructions JL et JG. Le fait de mélanger des entiers signés et non signés peut entraîner des conséquences imprévues.
17. Oubli d'ajustement de l'échelle d'un index de tableau. Un index de tableau doit être multiplié par la taille d'un élément du tableau. Par exemple : mov eax, MyIntegerArray[ebx*4].
18. Dépassement des limites d'un tableau. Un tableau de n éléments est indexé de 0 à $n - 1$, et non de 1 à n . Une boucle défectueuse écrivant en dehors des limites d'un tableau peut provoquer, ailleurs dans le programme, des erreurs difficiles à localiser.
19. Boucle avec ECX = 0. Une boucle se terminant par l'instruction LOOP s'exécutera 232 fois si ECX est égal à zéro. Veillez à vérifier si ECX est nul avant la boucle.

20. Lecture du flag de retenue après INC ou DEC. Les instructions INC et DEC ne modifient pas le flag de retenue. N'utilisez pas d'instructions qui lisent le flag de retenue (telles que ADC, SBB, JC, JBE, SETA, etc.) après INC ou DEC. Utilisez ADD et SUB en lieu et place de INC et DEC pour éviter ce problème.

3 Bases de la programmation en assembleur

3.1 Assembleurs disponibles

Il existe plusieurs assembleurs pour le jeu d'instructions x86. Les programmeurs en langage d'assemblage sont confrontés à une situation regrettable : il n'existe pas de syntaxe universellement reconnue pour l'assembleur x86. Différents assembleurs utilisent des variantes syntaxiques différentes. Les assembleurs les plus courants sont énumérés ci-dessous.

MASM

L'assembleur de Microsoft est inclus avec les compilateurs C++ de Microsoft. Il est parfois possible d'obtenir des versions gratuites en téléchargeant le kit de développement de pilotes (WDK) ou le kit de développement logiciel (SDK) pour Windows, ou encore en tant que module complémentaire de l'édition gratuite Visual C++ Express. Bien que MASM ait longtemps constitué un standard de fait dans l'univers Windows, il tombe aujourd'hui en désuétude. Le code assembleur généré par plusieurs compilateurs Windows continue toutefois d'utiliser la syntaxe MASM. Cette syntaxe est quelque peu confuse et incohérente, héritage des tout premiers assembleurs conçus pour le processeur 8086. Microsoft continue d'assurer la maintenance de MASM afin de proposer une suite complète d'outils de développement pour Windows ; toutefois, cette activité n'est manifestement pas rentable, et cette maintenance semble être réduite au strict minimum. Les versions les plus récentes ne peuvent fonctionner que si le compilateur est installé.

GAS

L'assembleur GNU fait partie du package GNU Binutils, inclus dans la plupart des distributions Linux, BSD et Mac OS X. Les compilateurs GNU et Clang produisent un code source assembleur qui est traité par l'assembleur GNU avant l'étape d'édition de liens. L'assembleur GNU utilise traditionnellement la syntaxe AT&T ; bien qu'efficace pour le code généré par machine, celle-ci s'avère peu pratique pour le code écrit manuellement. La syntaxe AT&T adopte un ordre des opérandes qui diffère de celui de tous les autres assembleurs x86 ainsi que de la documentation des instructions publiée par Intel et AMD. Elle emploie divers préfixes, tels que “%” et “\$”, pour spécifier les types d'opérandes. L'assembleur GNU est disponible pour toutes les plateformes x86.

Heureusement, l'assembleur GNU offre la possibilité d'utiliser la syntaxe Intel. La syntaxe Intel de GNU est similaire à celle de MASM. Elle ne définit que la syntaxe des instructions machine, et non celle des directives, fonctions, macros, etc. ; les directives continuent d'utiliser l'ancienne syntaxe GNU-AT&T. Pour utiliser la syntaxe Intel, il convient de spécifier `.intel_syntax noprefix`. Pour revenir à la syntaxe AT&T — par exemple avant de quitter un bloc d'assembleur en ligne dans du code C ou C++ — il faut spécifier `.att_syntax prefix`.

NASM

NASM est un assembleur libre et open source prenant en charge plusieurs plateformes et formats de fichiers objets. Sa syntaxe est plus claire et plus cohérente que celle de MASM. NASM est régulièrement mis à jour pour inclure de nouveaux jeux d'instructions. C'est actuellement le meilleur assembleur multiplateforme, largement utilisé par les programmeurs en langage assembleur.

YASM

YASM est très similaire à NASM et utilise exactement la même syntaxe. YASM et NASM peuvent être utilisés de manière interchangeable. YASM n'a pas été mis à jour depuis 2014.

FASM

FASM est un autre assembleur open source multiplateforme. Sa syntaxe n'est pas compatible avec celle des autres assembleurs. FASM est lui-même écrit en langage assembleur — une idée séduisante, mais qui, malheureusement, rend son développement et sa maintenance moins efficaces.

UASM

UASM est la dernière évolution d'une série d'initiatives menées par différentes personnes. Tout a commencé avec l'assembleur WASM inclus dans le compilateur Watcom C++. Le développement de WASM s'est poursuivi sous la forme d'un assembleur indépendant nommé JWASM, puis HJWASM et enfin UASM. Il est compatible avec la syntaxe MASM, y compris les macros avancées et les directives de haut niveau. Il peut être utilisé comme plugin pour l'environnement de développement (IDE) Visual Studio.

HLA

HLA (High Level Assembler) est en fait un compilateur de langage de haut niveau qui permet des instructions de type assembleur et produit une sortie d'assembleur. C'était probablement une bonne idée au moment où il a été inventé, mais aujourd'hui, alors que de nombreux compilateurs C++ prennent en charge les fonctions intrinsèques, je pense que HLA n'est plus nécessaire. HLA n'est apparemment plus maintenu.

TASM

Le Turbo Assembleur Borland était inclus avec le compilateur Borland C++, autrefois populaire ; son développement a été poursuivi par la suite par les sociétés CodeGear et Embarcadero. TASM est obsolète et n'est plus disponible.

Assembleurs Inline

Les compilateurs C++ d'Intel prennent en charge l'assembleur en ligne en utilisant un sous-ensemble de la syntaxe MASM.

Les compilateurs Microsoft ne prennent en charge l'assembleur en ligne qu'en mode 32 bits. Il est possible d'accéder aux variables, fonctions et étiquettes C++ simplement en insérant leurs noms dans le code assembleur. Cette méthode est simple, mais ne prend pas en charge les variables C++ stockées dans des registres. Voir page 27.

Les compilateurs GNU et Clang prennent en charge l'assembleur en ligne en offrant un accès à l'ensemble des instructions et directives de l'assembleur GNU, aussi bien avec la syntaxe Intel qu'avec la syntaxe AT&T. L'accès aux variables C++ depuis le code assembleur repose sur une syntaxe polyvalente mais complexe.

Les compilateurs Intel pour les systèmes Linux et Mac prennent en charge l'assembleur en ligne selon les styles Microsoft et GNU.

Fonctions intrinsèques en C++

C'est le moyen le plus pratique de combiner du code de bas niveau et de haut niveau. Les fonctions intrinsèques sont des représentations, dans un langage de haut niveau, d'instructions machine. Par exemple, vous pouvez effectuer une addition de vecteurs en C++ en appelant la fonction intrinsèque équivalente à une instruction d'assemblage pour l'addition de vecteurs. De plus, il est possible de définir une classe de vecteurs avec un opérateur "+" surchargé, de sorte qu'une addition de vecteurs s'obtienne simplement en écrivant "+". Les fonctions intrinsèques sont prises en charge par les compilateurs Microsoft, Intel, GNU et Clang. Voir la page 25 et le manuel 1 : "Optimizing software in C++".

Quel assembleur choisir ?

Dans la plupart des cas, la solution la plus simple consiste à utiliser des fonctions intrinsèques dans le code C++. Le compilateur peut s'occuper de l'essentiel de l'optimisation afin que le programmeur puisse se concentrer sur le choix du meilleur algorithme et sur l'organisation des données en vecteurs. Les programmeurs système peuvent accéder aux instructions système en utilisant des fonctions intrinsèques sans avoir à utiliser le langage assembleur.

Lorsqu'une véritable programmation de bas niveau est nécessaire, comme dans des bibliothèques de fonctions ou des pilotes de périphériques hautement optimisés, vous pouvez utiliser un assembleur.

Il peut être préférable d'utiliser un assembleur compatible avec le compilateur C++ que vous utilisez. Cela vous permet d'utiliser le compilateur pour traduire le C++ en assembleur, d'optimiser davantage le code assembleur, puis de l'assembler. Si l'assembleur n'est pas compatible avec la syntaxe générée par le compilateur, vous pouvez alors générer un fichier objet avec le compilateur, puis désassembler le fichier objet selon la syntaxe d'assemblage dont vous avez besoin. Le désassembleur [objconv](#) prend en charge plusieurs dialectes de syntaxe différents.

L'assembleur NASM constitue un excellent choix pour de nombreux usages : il prend en charge une grande variété de plateformes et de formats de fichiers objets, bénéficie d'une bonne maintenance et est généralement à jour concernant les derniers jeux d'instructions.

Sauf indication contraire, les exemples présentés dans ce manuel utilisent la syntaxe NASM.

Consultez le site www.agner.org/optimize pour accéder à des liens vers divers manuels de syntaxe et de programmation ainsi qu'à des forums de discussion.

3.2 Jeu de registres et instructions de base

Registres en mode 16 bits

Registres à usage général et registres d'entiers

Registre complet bits 0 – 15	Registre partiel bits 8 – 15	Registre partiel bits 0 – 7
---------------------------------	---------------------------------	--------------------------------

AX	AH	AL
BX	BH	BL
CX	CH	CL
DX	DH	DL
SI		
DI		
BP		
SP		
Flags		
IP		
Tableau 3.1. Registres à usage général en mode 16 bits.		

Les registres 32 bits sont également disponibles en mode 16 bits, sous réserve que le microprocesseur et le système d'exploitation le prennent en charge. Le mot de poids fort du registre ESP ne doit pas être utilisé, car il n'est pas sauvegardé lors des interruptions.

Registres à virgule flottante

Registre complet bits 0 – 79
ST(0)
ST(1)
ST(2)
ST(3)
ST(4)
ST(5)
ST(6)
ST(7)
Tableau 3.2. Registres de pile à virgule flottante

Les registres MMX peuvent être disponibles s'ils sont pris en charge par le microprocesseur. Les registres XMM peuvent être disponibles s'ils sont pris en charge par le microprocesseur et le système d'exploitation.

Registres de segment

Registre complet bits 0 – 15
CS
DS
ES
SS
Tableau 3.3. Registres de segment en mode 16 bits

Les registres FS et GS peuvent être disponibles.

Registres en mode 32 bits

Registres à usage général et registres d'entiers

Registre complet bits 0 – 31	Registre partiel bits 0 – 15	Registre partiel bits 8 – 15	Registre partiel bits 0 – 7
EAX	AX	AH	AL
EBX	BX	BH	BL
ECX	CX	CH	CL
EDX	DX	DH	DL
ESI	SI		
EDI	DI		
EBP	BP		
ESP	SP		
EFlags	Flags		
EIP	IP		
Tableau 3.4. Registres à usage général en mode 32 bits			

Registres à virgule flottante et registres vectoriels 64 bits

Registre complet	Registre partiel
------------------	------------------

bits 0 – 79	bits 0 – 63
ST(0)	MM0
ST(1)	MM1
ST(2)	MM2
ST(3)	MM3
ST(4)	MM4
ST(5)	MM5
ST(6)	MM6
ST(7)	MM7
Tableau 3.5. Registres à virgule flottante et MMX	

Les registres MMX ne sont disponibles que s'ils sont pris en charge par le microprocesseur. Les registres ST et MMX ne peuvent pas être utilisés dans la même partie du code. Une section de code utilisant des registres MMX doit être séparée de toute section ultérieure utilisant des registres ST par l'exécution d'une instruction EMMS.

Registres vectoriels d'entiers et de nombres à virgule flottante de 128 bits et plus

Registre complet ou partiel bits 0 – 127	Registre complet ou partiel bits 0 – 255	Registre complet bits 0 – 511
XMM0	YMM0	ZMM0
XMM1	YMM1	ZMM1
XMM2	YMM2	ZMM2
XMM3	YMM3	ZMM3
XMM4	YMM4	ZMM4
XMM5	YMM5	ZMM5
XMM6	YMM6	ZMM6
XMM7	YMM7	ZMM7
Tableau 3.6. Registres XMM, YMM et ZMM en mode 32 bits		

Les registres XMM ne sont disponibles que s'ils sont pris en charge à la fois par le microprocesseur et par le système d'exploitation. Les instructions scalaires à virgule flottante utilisent uniquement 32 ou 64 bits des registres XMM pour une simple ou une double précision, respectivement. Les registres YMM ne sont disponibles que si le processeur et le système d'exploitation prennent en charge le jeu d'instructions AVX. Les registres ZMM sont disponibles si le processeur prend en charge le jeu d'instructions AVX-512.

Registres de segment

Registre complet bits 0 – 15
CS
DS
ES
FS
GS
SS
Tableau 3.7. Registres de segment en mode 32 bits

Registres en mode 64 bits

Registres à usage général et registres d'entiers

Registre complet bits 0 – 63	Registre partiel bits 0 – 31	Registre partiel bits 0 – 15	Registre partiel bits 8 – 15	Registre partiel bits 0 – 7
RAX	EAX	AX	AH	AL
RBX	EBX	BX	BH	BL
RCX	ECX	CX	CH	CL
RDX	EDX	DX	DH	DL
RSI	ESI	SI	SIL	
RDI	EDI	DI	DIL	
RBP	EBP	BP	BPL	
RSP	ESP	SP	SPL	
R8	R8D	R8W	R8B	
R9	R9D	R9W	R9B	
R10	R10D	R10W	R10B	

R11	R11D	R11W	R11B	
R12	R12D	R12W	R12B	
R13	R13D	R13W	R13B	
R14	R14D	R14W	R14B	
R15	R15D	R15W	R15B	
RFlags		Flags		
RIP				

Tableau 3.8. Registres en mode 64 bits

Les registres 8 bits de poids fort AH, BH, CH et DH ne peuvent être utilisés que dans des instructions dépourvues de préfixe REX.

Notez que la modification d'un registre partiel de 32 bits met à zéro le reste du registre (bits 32 à 63), alors que la modification d'un registre partiel de 8 ou 16 bits n'affecte pas le reste du registre. Cela peut être illustré par la séquence suivante :

```
; Exemple 3.1. Registres de 8, 16, 32 et 64 bits
mov rax, 1111111111111111H ; rax = 1111111111111111H
mov eax, 22222222H ; rax = 0000000022222222H
mov ax, 3333H ; rax = 0000000022223333H
mov al, 44H ; rax = 0000000022223344H
```

Cette incohérence s'explique par une bonne raison. Mettre à zéro la partie inutilisée d'un registre est plus efficace que de la laisser inchangée, car cela supprime une fausse dépendance vis-à-vis des valeurs précédentes. Toutefois, le principe de remise à zéro de la partie inutilisée d'un registre ne peut pas être étendu aux registres partiels de 16 et 8 bits, car cela romprait la compatibilité ascendante avec les modes 32 bits et 16 bits.

La seule instruction pouvant accepter un opérande de donnée immédiate sur 64 bits est MOV. Les autres instructions sur entiers ne peuvent accepter qu'un opérande sur 32 bits avec extension de signe. Exemples :

```
; Exemple 3.2. Opérandes immédiats, avec extension à zéro ou extension de signe
mov rax, 1111111111111111H ; opérande immédiat 64 bits complet
mov rax, -1 ; opérande 32 bits avec extension de signe
mov eax, 0ffffffffH ; opérande 32 bits à extension par des zéros
add rax, 1 ; opérande 8 bits avec extension de signe
add rax, 100H ; opérande 32 bits avec extension de signe
add eax, 100H ; opérande 32 bits. Le résultat est étendu par des zéros.
mov rbx, 100000000H ; opérande immédiat 64 bits
add rax, rbx ; utilisez un registre supplémentaire si l'opérande est volumineux.
```

Il n'est pas possible d'utiliser un opérande étendu par signe de 16 bits. Si vous devez ajouter une valeur immédiate à un registre de 64 bits, il est alors nécessaire de d'abord déplacer la valeur dans un autre registre si la valeur est trop grande pour tenir dans un opérande de signe étendu de 32 bits.

Registres à virgule flottante et registres vectoriels 64 bits

Registre complet bits 0 – 79	Registre partiel bits 0 – 63
ST(0)	MM0
ST(1)	MM1
ST(2)	MM2
ST(3)	MM3
ST(4)	MM4
ST(5)	MM5
ST(6)	MM6
ST(7)	MM7

Tableau 3.9. Registres à virgule flottante et MMX

Les registres ST et MMX ne peuvent pas être utilisés dans la même partie du code. Une section de code utilisant des registres MMX doit être séparée de toute section ultérieure utilisant des registres ST en exécutant une instruction EMMS. Les registres ST et MMX ne peuvent pas être utilisés dans les pilotes de périphérique pour Windows 64 bits.

Registres vectoriels d'entiers et de nombres à virgule flottante de 128 bits ou plus

Registre complet ou partiel bits 0 – 127	Registre complet ou partiel bits 0 – 255	Registre complet bits 0 – 511
XMM0	YMM0	ZMM0
XMM1	YMM1	ZMM1

XMM2	YMM2	ZMM2
XMM3	YMM3	ZMM3
XMM4	YMM4	ZMM4
XMM5	YMM5	ZMM5
XMM6	YMM6	ZMM6
XMM7	YMM7	ZMM7
XMM8	YMM8	ZMM8
XMM9	YMM9	ZMM9
XMM10	YMM10	ZMM10
XMM11	YMM11	ZMM11
XMM12	YMM12	ZMM12
XMM13	YMM13	ZMM13
XMM14	YMM14	ZMM14
XMM15	YMM15	ZMM15
ZMM16		
ZMM17		
ZMM18		
ZMM19		
ZMM20		
ZMM21		
ZMM22		
ZMM23		
ZMM24		
ZMM25		
ZMM26		
ZMM27		
ZMM28		
ZMM29		
ZMM30		
ZMM31		
Tableau 3.10. Registres XMM, YMM et ZMM en mode 64 bits		

Les instructions scalaires en virgule flottante n'utilisent que 32 ou 64 bits des registres XMM, respectivement pour la simple ou la double précision. Les registres YMM ne sont disponibles que si le processeur et le système d'exploitation prennent en charge le jeu d'instructions AVX. Les registres ZMM ne sont disponibles que si le processeur prend en charge le jeu d'instructions AVX-512. Il est possible d'utiliser les registres XMM16–31 et YMM16–31 lorsque le processeur prend en charge l'extension AVX-512 VL.

Registres de segment

Registre complet bits 0 – 15
CS
FS
GS
Tableau 3.11. Registres de segment en mode 64 bits

Les registres de segment ne sont utilisés qu'à des fins particulières.

3.3 Modes d'adressage

Adressage en mode 16 bits

Le code 16 bits utilise un modèle de mémoire segmentée. Un opérande mémoire peut comporter l'un quelconque de ces composants :

- Une spécification de segment. Il peut s'agir de n'importe quel registre de segment ou d'un nom de segment (ou de groupe) associé à un registre de segment. (Le segment par défaut est DS, sauf si BP est utilisé comme registre de base). Le segment peut être déduit d'une étiquette définie au sein d'un segment.
- Une étiquette définissant un déplacement relogeable. Le déplacement par rapport au début du segment est calculé par l'éditeur de liens.
- Un déplacement immédiat. Il s'agit d'une constante. Si un déplacement relogeable est également présent, les valeurs sont additionnées.
- Un registre de base. Il ne peut s'agir que de BX ou de BP.

- Un registre d'index. Il ne peut s'agir que de SI ou de DI. Aucun facteur d'échelle n'est autorisé.

Un opérande mémoire peut comporter toutes ces composantes. Un opérande ne contenant qu'un déplacement immédiat n'est pas interprété comme un opérande mémoire par l'assembleur MASM, même s'il est entouré de crochets []. Exemples :

; **Exemple 3.3.** Opérandes mémoire en mode 16 bits, syntaxe MASM
 MOV AX, DS:[100H] ; L'adresse comporte un segment et un décalage immédiat.
 ADD AX, MEM[SI]+4 ; Possède un décalage et un index relocalisables et immédiats

Les structures de données dépassant 64 Ko sont gérées de la manière suivante. En mode réel et en mode virtuel (DOS) : ajouter 1 au registre de segment revient à ajouter 10H à l'offset. En mode protégé (Windows 3.x) : ajouter 8 au registre de segment revient à ajouter 10000H à l'offset. La valeur ajoutée au segment doit être un multiple de 8.

Adressage en mode 32 bits

Le code 32 bits utilise un modèle de mémoire *flat* (plat) dans la plupart des cas. La segmentation est possible, mais elle n'est utilisée qu'à des fins spécifiques (par exemple, le bloc d'environnement de thread dans le registre FS).

Un opérande mémoire peut comporter l'un quelconque des éléments suivants :

- Une spécification de segment. Non utilisée en mode *flat* (mode plat).
- Une étiquette définissant un déplacement relogeable. Le déplacement par rapport au groupe de segments FLAT est calculé par l'éditeur de liens.
- Un déplacement immédiat. Il s'agit d'une constante. S'il existe également un déplacement relogeable, les valeurs sont additionnées.
- Un registre de base. Il peut s'agir de n'importe quel registre 32 bits.
- Un registre d'index. Il peut s'agir de n'importe quel registre 32 bits, à l'exception d'ESP.
- Un facteur d'échelle appliqué au registre d'index. Les valeurs autorisées sont 1, 2, 4 et 8.

Un opérande mémoire peut comporter tous ces composants. Exemples :

; **Exemple 3.4.** Opérandes mémoire en mode 32 bits
 mov eax, [fs:10H] ; L'adresse comporte un segment et un décalage immédiat.
 add eax, [mem+esi] ; Dispose d'un décalage et d'un index déplaçables.
 add eax, [esp+ecx*4+8] ; Base, index, échelle et déplacement immédiat

Code indépendant de la position en mode 32 bits

Un code indépendant de la position est nécessaire pour créer des objets partagés (*.so) sur les systèmes de type Unix 32 bits. La méthode couramment utilisée pour générer ce type de code sous Linux et BSD 32 bits consiste à employer une table de décalage global (GOT, *Global Offset Table*) contenant les adresses de tous les objets statiques. Cette méthode est assez inefficace, car le code doit récupérer une adresse dans la GOT à chaque lecture ou écriture de données dans le segment de données. Une méthode plus rapide consiste à utiliser un point de référence arbitraire, comme le montre l'exemple suivant. Notez que cette technique fonctionne avec l'assembleur NASM, mais pas avec plusieurs autres assembleurs :

; **Exemple 3.5a.** Code indépendant de la position, 32 bits, syntaxe NASM
 SECTION .data
 alpha: dd 1
 beta: dd 2
 SECTION .text
 funca: ; Cette fonction renvoie alpha + bêta.
 call get_thunk_ecx ; obtention de ecx = eip
 refpoint: ; ecx pointe ici
 mov eax, [ecx+alpha-refpoint] ; adresse relative
 add eax, [ecx+beta-refpoint] ; adresse relative
 ret
 get_thunk_ecx: ; Fonction de lecture du pointeur d'instructions
 mov ecx, [esp]
 ret

La seule instruction capable de lire le pointeur d'instruction en mode 32 bits est l'instruction `call`. Dans l'exemple 3.5, nous utilisons `call get_thunk_ecx` pour charger le pointeur d'instruction (eip) dans le registre ecx. ecx pointera alors vers la première instruction suivant le `call`. Il s'agit de notre point de référence, nommé `refpoint`.

Le compilateur GNU pour Mac OS X 32 bits utilise une version légèrement différente de cette méthode :

Exemple 3.5b. Mauvaise méthode !

```
funca:                ; Cette fonction renvoie alpha + bêta.
call retpoint         ; obtention de eip sur la pile
retpoint:
pop ecx               ; pop eip à partir de la pile
mov eax, [ecx+alpha-retpoint] ; adresse relative
add eax, [ecx+beta-retpoint]  ; adresse relative
ret
```

La méthode utilisée dans l'exemple 3.5b est mauvaise car elle comporte une instruction `call` qui ne correspond pas à une instruction `return`. Cela entraînera une mauvaise prédiction des retours ultérieurs sur de nombreux processeurs. (Consultez le manuel 3 : “The microarchitecture of Intel, AMD and VIA CPUs” pour une explication sur la prédiction des retours).

Cette méthode est couramment utilisée sur les systèmes Mac, où le format de fichier Mach-O prend en charge les références relatives à un point arbitraire. D'autres formats de fichier ne permettent pas ce type de référence, mais il est possible d'utiliser une référence relative à l'adresse courante (*self-relative*) avec un décalage. Les assembleurs NASM et GNU effectuent cette opération automatiquement, alors que la plupart des autres assembleurs ne sont pas capables de gérer cette situation. Il est donc nécessaire d'utiliser NASM ou l'assembleur GNU si vous souhaitez générer du code indépendant de la position en mode 32 bits avec cette méthode. Le code peut sembler étrange dans un débogueur ou un désassembleur, mais il s'exécute sans problème sous Linux, BSD et Windows en 32 bits. Sur les systèmes Mac 32 bits, il est possible que le chargeur n'identifie pas correctement la section cible, à moins d'utiliser un assembleur prenant en charge la méthode du point de référence (je ne connais aucun autre assembleur que celui de GNU capable de le faire correctement).

La méthode utilisant la GOT emploierait la même technique de point de référence que dans l'exemple 3.5a pour accéder à la GOT, puis utiliserait celle-ci pour charger les adresses de alpha et beta dans d'autres registres de pointeur. Il s'agit d'un gaspillage inutile de temps et de registres, car si l'on peut accéder à la GOT par rapport au point de référence, on peut tout aussi bien accéder à alpha et beta par rapport à ce même point.

Les tables de pointeurs utilisées dans les instructions `switch/case` peuvent employer le même point de référence pour la table elle-même et pour les pointeurs qu'elle contient :

```
; Exemple 3.6. Switch indépendant de la position, 32 bits, syntaxe NASM
SECTION .data

jumptable: dd case1-retpoint, case2-retpoint, case3-retpoint

SECTION .text

global funcb:function

funcb: ; Cette fonction implémente une instruction switch.
    mov eax, [esp+4] ; paramètre de fonction
    call get_thunk_ecx ; obtention de ecx = eip
retpoint: ; ecx pointe ici
    cmp eax, 3
    jnb case_default ; index hors limites
    mov eax, [ecx+eax*4+jumptable-retpoint] ; Lecture de l'entrée de la table
; Les adresses de saut sont relatives à retpoint
; obtention de l'adresse absolue :
    add eax, ecx
    jmp eax ; Saut au cas souhaité
case1: ...
    ret
case2: ...
    ret
case3: ...
    ret
case_default:
    ...
    ret
get_thunk_ecx: ; Fonction de lecture du pointeur d'instruction
    mov ecx, [esp]
    ret
```

Adressage en mode 64 bits

Le code 64 bits utilise toujours un modèle mémoire plat. La segmentation est impossible, à l'exception des registres FS et GS, qui sont réservés à des usages spécifiques (bloc d'environnement de thread, etc.).

Il existe plusieurs modes d'adressage en mode 64 bits : relatif à RIP, absolu 32 bits, absolu 64 bits et relatif à un registre de base.

Adressage relatif à RIP

C'est le mode d'adressage privilégié pour les données statiques. L'adresse contient un déplacement de 32 bits avec extension de signe par rapport au pointeur d'instruction. L'adresse ne peut comporter ni registre de segment ni registre d'index, ni aucun registre de base autre que RIP, qui est implicite. Exemple :

- ; **Exemple 3.7a.** Opérande mémoire relatif à RIP, syntaxe NASM
default rel
mov eax, [mem]
- ; **Exemple 3.7b.** Opérande mémoire relatif à RIP, syntaxe MASM
mov eax, [mem]
- ; **Exemple 3.7c.** Opérande mémoire relatif à RIP, syntaxe GAS/Intel
mov eax, [mem+rip]

L'assembleur MASM génère toujours des adresses relatives au registre RIP pour les données statiques lorsqu'aucun registre de base ou d'index explicite n'est spécifié. Avec d'autres assembleurs, vous devez penser à spécifier l'adressage relatif.

Adressage absolu 32 bits en mode 64 bits

Une adresse constante sur 32 bits fait l'objet d'une extension de signe pour passer à 64 bits. Ce mode d'adressage ne fonctionne que s'il est garanti que toutes les adresses sont inférieures à 2^{31} (ou supérieures à -2^{31} pour le code système).

L'utilisation de l'adressage absolu sur 32 bits est sans risque pour les exécutables principaux sous Linux et BSD, où toutes les adresses sont par défaut inférieures à 2^{31} , mais elle est impossible pour les objets partagés. Les adresses absolues sur 32 bits fonctionnent souvent aussi avec les exécutables principaux sous Windows (mais pas avec les DLL) ; toutefois, aucun compilateur Windows n'exploite cette possibilité, car les fichiers .exe peuvent être relocalisés à des adresses plus élevées dans certains cas particuliers.

Les adresses absolues sur 32 bits ne peuvent pas être utilisées sous Mac OS X, où les adresses sont par défaut supérieures à 2^{32} .

À noter que les assembleurs NASM et GNU peuvent générer des adresses absolues sur 32 bits si l'on ne spécifie pas explicitement un adressage relatif au registre RIP. Il est nécessaire de préciser `default rel` avec NASM ou `[mem+rip]` avec Gas pour éviter l'utilisation d'adresses absolues sur 32 bits.

Il n'y a absolument aucune raison d'utiliser des adresses absolues pour des opérandes mémoire simples. Les adresses relatives au registre RIP permettent de réduire la taille des instructions, suppriment le besoin de relocation au moment du chargement et peuvent être utilisées en toute sécurité sur tous les systèmes.

Les adresses absolues ne sont nécessaires que pour accéder à des tableaux impliquant un registre d'index, par exemple :

- ; **Exemple 3.8.** Adresses absolues 32 bits en mode 64 bits
mov al, [abs chararray + rsi]
mov ebx, [abs intarray + rsi*4]

Cette méthode ne peut être utilisée que s'il est garanti que l'adresse est $< 2^{31}$, comme expliqué ci-dessus. Voir ci-dessous pour d'autres méthodes d'adressage des tableaux statiques.

L'assembleur MASM génère des adresses absolues uniquement lorsqu'un registre de base ou d'index est spécifié avec une étiquette de mémoire comme dans l'exemple 3.8 ci-dessus.

Le registre d'index doit de préférence être un registre de 64 bits et non un registre de 32 bits. La segmentation n'est possible qu'avec FS ou GS.

Adressage absolu 64 bits

Cela utilise une adresse virtuelle absolue de 64 bits. L'adresse ne peut contenir aucun registre de segment, registre de base ou d'index. Les adresses absolues 64 bits ne peuvent être utilisées qu'avec l'instruction MOV, et uniquement avec AL, AX, EAX ou RAX comme source ou destination.

- ; **Exemple 3.9.** Adresse absolue 64 bits, syntaxe NASM
mov eax, dword [qword a]

Ce mode d'adressage n'est pas pris en charge par l'assembleur MASM, mais il l'est par la plupart des autres assembleurs.

Adressage relatif au registre de base 64 bits

Un opérande mémoire dans ce mode peut comporter l'un quelconque de ces composants :

- Un registre de base. Il peut s'agir de n'importe quel registre d'entiers 64 bits.
- Un registre d'index. Il peut s'agir de n'importe quel registre d'entiers 64 bits, à l'exception de RSP.
- Un facteur d'échelle appliqué au registre d'index. Les seules valeurs possibles sont 1, 2, 4 et 8.
- Un déplacement immédiat. Il s'agit d'un déplacement constant par rapport au registre de base.

Un registre de base est toujours nécessaire pour ce mode d'adressage. Les autres composants sont facultatifs.

Exemples :

```
; Exemple 3.10. Adressage par registre de base en mode 64 bits
mov eax, [rsi]
add eax, [rsp + 4*rcx + 8]
```

Ce mode d'adressage est utilisé pour les données situées sur la pile, pour les membres de structures et de classes, ainsi que pour les tableaux.

Adressage de tableaux statiques en mode 64 bits

Il n'est pas possible d'accéder à des tableaux statiques en utilisant un adressage relatif au RIP combiné à un registre d'index. Plusieurs alternatives sont possibles.

Les exemples suivants portent sur des tableaux statiques. Le code C++ correspondant à cet exemple est le suivant :

```
// Exemple 3.11a. Tableaux statiques en mode 64 bits
// C++ code:
static int a[100], b[100];
for (int i = 0; i < 100; i++) {
    b[i] = -a[i];
}
```

La solution la plus simple consiste à utiliser des adresses absolues sur 32 bits. Cela est possible tant que les adresses sont inférieures à 2^{31} .

```
; Exemple 3.11b. Utilisation d'adresses absolues 32 bits
; 64 bit Linux seulement
; On suppose que la base de l'image est inférieure à 80000000H.
```

```
SECTION .bss
A: resd 100          ; Définition du tableau statique A.
B: resd 100          ; Définition du tableau statique B

SECTION .text
xor ecx, ecx        ; i = 0

TOPOFLOOP:          ; Haut de la boucle
mov eax, [abs A+rcx*4] ; Adresse 32 bits + index mis à l'échelle
neg eax
mov [abs B+rcx*4], eax ; Adresse 32 bits + index mis à l'échelle
add ecx, 1
cmp ecx, 100        ; i < 100
jb TOPOFLOOP        ; Boucle
```

L'assembleur générera une adresse relogeable sur 32 bits pour A et B dans l'exemple 3.11b, car il ne peut pas combiner une adresse relative à RIP avec un registre d'index.

Cette méthode est utilisée sous Linux 64 bits pour accéder aux tableaux statiques. Aucun compilateur pour Windows 64 bits ne l'utilise, car elle n'est pas fiable à 100 %. Sous Windows, la base de l'image est généralement de 2^{22} pour les programmes d'application et se situe entre 2^{28} et 2^{29} pour les DLL ; cette méthode fonctionnera donc dans la plupart des cas, mais pas dans tous. Elle ne peut pas être utilisée sur les systèmes Mac 64 bits, car toutes les adresses y sont supérieures à 2^{32} par défaut.

La seconde méthode consiste à utiliser un adressage relatif à l'image. La solution suivante charge la base de l'image dans le registre RBX à l'aide d'une instruction LEA avec une adresse relative à RIP :

```
; Exemple 3.11c. Adresse relative à la base de l'image
; 64 bits, Windows uniquement, assembleur MASM
```

```
.data
A DD 100 dup (?)
B DD 100 dup (?)
```

```

extern __ImageBase:byte

.code
lea rbx, [__ImageBase] ; Utilisation de l'adresse relative à RIP de l'image de base
xor ecx, ecx           ; i = 0

TOPOFLOOP:             ; Haut de la boucle
; imagerel(A) = adresse de A par rapport à la base de l'image :
mov eax, [(imagerel A) + rbx + rcx*4]
neg eax
mov [(imagerel B) + rbx + rcx*4], eax
add ecx, 1
cmp ecx, 100
jb TOPOFLOOP

```

Cette méthode est utilisée uniquement sous Windows 64 bits et nécessite l'assembleur MASM. Sous Linux, l'adresse de base de l'image est accessible via `__executable_start`, mais les adresses relatives à l'image ne sont pas prises en charge par le format de fichier ELF. Le format Mach-O autorise les adresses relatives à un point de référence arbitraire, y compris l'adresse de base de l'image, accessible via

`__mh_execute_header`.

La troisième solution charge l'adresse du tableau A dans le registre RBX à l'aide d'une instruction LEA utilisant un adressage relatif à RIP. L'adresse de B est calculée par rapport à celle de A.

```

; Exemple 3.11d.
; Chargement de l'adresse du tableau dans le registre de base
; Fonctionne sur tous les systèmes 64 bits. Syntaxe NASM.

default rel           ; Utilisation de l'adressage relatif à RIP lorsque cela est possible.

SECTION .bss
A: resd 100           ; Définition du tableau statique A
B: resd 100           ; Définition du tableau statique B

SECTION .text

lea rbx, [A]          ; Chargement de l'adresse relative à RIP de A
xor ecx, ecx          ; i = 0
TOPOFLOOP:           ; Haut de la boucle
mov eax, [rbx + 4*rcx] ; A[i]
neg eax
mov [(B-A) + rbx + 4*rcx], eax ; Utilisation du décalage de B par rapport à A.
add ecx, 1
cmp ecx, 100
jb TOPOFLOOP

```

Notez que nous pouvons utiliser une instruction 32 bits pour incrémenter l'indice (`ADD ECX, 1`), même si nous utilisons le registre 64 bits (RCX) pour cet indice. Cela fonctionne car nous avons la certitude que l'indice est positif ou nul et inférieur à 2^{32} . Cette méthode permet d'utiliser n'importe quelle adresse du segment de données comme point de référence et de calculer d'autres adresses par rapport à ce point.

Si un tableau se trouve à une distance supérieure à 2^{31} octets du pointeur d'instruction, nous devons alors charger l'adresse complète sur 64 bits dans un registre de base. Par exemple, dans l'exemple 3.11d, nous pouvons remplacer `lea rbx, [A]` par `mov rbx, qword A`.

Code indépendant de la position en mode 64 bits

Il est facile de générer du code indépendant de la position en mode 64 bits. L'accès aux données statiques peut s'effectuer via un adressage relatif au registre RIP. Les tableaux statiques sont accessibles comme dans l'exemple 3.11d.

Les tables de pointeurs des instructions switch peuvent être définies par rapport à un point de référence arbitraire. Il est commode d'utiliser la table elle-même comme point de référence :

```

; Exemple 3.12. instruction switch avec pointeurs relatifs, 64 bits, syntaxe NASM

SECTION .data
jumptable: dd case1-jumptable, case2-jumptable, case3-jumptable

SECTION .text
default rel           ; utilisation d'adresses relatives
global funcb:function

```

```

funcb:                                ; cette fonction implémente une instruction switch.
                                        ; le premier paramètre de la fonction est ECX sous Windows,
                                        ; et EDI sous Linux.
    mov eax, ecx                      ; paramètre de fonction
    cmp eax, 3
    jnb case_default                 ; index hors limites
    lea rdx, [jump table]             ; adresse de la table
    movsxd rax, dword [rdx+rax*4]     ; lecture de l'entrée de la table
; Les adresses de saut sont relatives à la table de saut ; obtention de l'adresse absolue :
    add rax, rdx
    jmp rax                          ; saut au cas souhaité
case1:                               ...
    ret
case2:                               ...
    ret
case3:                               ...
    ret
case_default:
    ...
ret

```

Cette méthode peut être utile pour réduire la taille des tables de pointeurs longues, car elle utilise des pointeurs relatifs de 32 bits plutôt que des pointeurs absolus de 64 bits.

L'assembleur MASM ne peut pas générer les tables relatives de l'exemple 3.12 à moins que la table de saut ne soit placée dans le segment de code. Il est préférable de placer la table de sauts dans le segment de données pour une mise en cache et une prélecture optimales du code, et cela peut être fait avec les assembleurs NASM ou Gnu.

3.4 Format du code d'instruction

Le format des codes d'instruction est décrit en détail dans les manuels d'Intel et d'AMD. Les principes fondamentaux du codage des instructions sont exposés ici en raison de leur incidence sur les performances du microprocesseur. En règle générale, vous pouvez laisser l'assembleur générer le codage le plus compact possible pour une instruction.

Chaque instruction peut se composer des éléments suivants, dans l'ordre indiqué :

1. **Préfixes** (0 à 5 octets)
Il s'agit de préfixes qui modifient la signification de l'opcode qui suit. Il existe plusieurs types de préfixes, comme indiqué dans le tableau 3.12 ci-dessous.
2. **Opcode** (1 à 3 octets)
Il s'agit du code de l'instruction. Il peut prendre les formes suivantes :
Un seul octet : XX
Deux octets : 0F XX
Trois octets : 0F 38 XX ou 0F 3A XX
Les opcodes de trois octets de la forme 0F 38 XX comportent toujours un octet mod-reg-r/m et aucun déplacement. Les opcodes de trois octets de la forme 0F 3A XX comportent toujours un octet mod-reg-r/m et un déplacement d'un octet.
3. **Octet mod-reg-r/m** (0 ou 1 octet)
Cet octet, qui définit les opérandes, se compose de trois champs. Le champ *mod* (2 bits) spécifie le mode d'adressage ; le champ *reg* (3 bits) désigne le registre du premier opérande (le plus souvent l'opérande de destination) ; le champ *r/m* (3 bits) spécifie le second opérande (le plus souvent l'opérande source), qui peut être un registre ou un opérande en mémoire. Le champ *reg* peut faire partie du code opération (opcode) si l'instruction ne comporte qu'un seul opérande.
4. **Octet SIB** (0 à 1 octet)
Cet octet est utilisé pour les opérandes mémoire employant des modes d'adressage complexes, et uniquement en présence d'un octet mod-reg-r/m. Il comprend deux bits pour un facteur d'échelle, trois bits spécifiant un registre d'index mis à l'échelle et trois bits spécifiant un registre de base. Un octet SIB est requis dans les cas suivants :
 - a. Si un opérande mémoire utilise deux registres de pointeur ou d'index,
 - b. Si un opérande mémoire utilise un registre d'index mis à l'échelle,
 - c. Si un opérande mémoire utilise le pointeur de pile (ESP ou RSP) comme registre de base,
 - d. Si un opérande mémoire, en mode 64 bits, utilise une adresse mémoire directe sur 32 bits avec extension de signe plutôt qu'une adresse relative au RIP.

Un octet SIB ne peut pas être utilisé en mode d'adressage 16 bits.

5. **Déplacement** (0, 1, 2, 4 ou 8 octets)

Il s'agit d'une partie de l'adresse d'un opérande mémoire. Elle est ajoutée à la valeur des registres de pointeur (base, index ou les deux), le cas échéant.

Un déplacement d'un octet avec extension de signe est possible dans tous les modes d'adressage si un registre de pointeur est spécifié.

Un déplacement de 2 octets n'est possible qu'en mode d'adressage 16 bits.

Un déplacement de 4 octets est possible en mode d'adressage 32 bits.

Un déplacement de 4 octets avec extension de signe est possible en mode d'adressage 64 bits. Si des registres de pointeur sont spécifiés, le déplacement est ajouté à ceux-ci. Si aucun registre de pointeur ni octet SIB n'est spécifié, le déplacement est ajouté au registre RIP. En présence d'un octet SIB mais sans registre de pointeur, la valeur avec extension de signe constitue une adresse directe absolue.

Une adresse directe absolue de 8 octets est possible en mode d'adressage 64 bits pour certaines instructions MOV ne comportant pas d'octet mod-reg-r/m.

6. **Opérande immédiat** (0, 1, 2, 4 ou 8 octets)

Il s'agit d'une constante de données qui, dans la plupart des cas, sert d'opérande source pour l'opération.

Un opérande immédiat d'un octet avec extension de signe est possible dans tous les modes pour toutes les instructions pouvant accepter des opérandes immédiats, à l'exception de MOV, CALL et RET.

Un opérande immédiat de 2 octets est possible pour les instructions utilisant des opérandes de 16 bits.

Un opérande immédiat de 4 octets est possible pour les instructions utilisant des opérandes de 32 bits.

Un opérande immédiat de 4 octets avec extension de signe est possible pour les instructions utilisant des opérandes de 64 bits.

Un opérande immédiat de 8 octets n'est possible que pour les opérations de transfert vers un registre de 64 bits.

3.5 Préfixes d'instruction

Le tableau suivant résume l'utilisation des préfixes d'instruction.

Préfixe pour :	mode 16 bits	mode 32 bits	mode 64 bits
taille d'opérande de 8 bits	aucun	aucun	aucun
taille d'opérande de 16 bits	aucun	66h	66h
taille d'opérande de 32 bits	66h	aucun	aucun
taille d'opérande de 64 bits	n.a.	n.a.	REX.W (48h)
entiers juxtaposés dans un registre MMX	aucun	aucun	aucun
entiers juxtaposés dans un registre XMM	66h	66h	66h
flottants simple-précision juxtaposés dans un registre XMM	aucun	aucun	aucun
flottants double-précision juxtaposés dans un registre XMM	66h	66h	66h
flottants simple-précision scalaires dans un registre XMM	F3h	F3h	F3h
flottants double-précision scalaires dans un registre XMM	F2h	F2h	F2h
taille d'adresse de 16 bits	aucun	67h	n.a.
taille d'adresse de 32 bits	67h	aucun	67h
taille d'adresse de 64 bits	n.a.	n.a.	aucun
segment CS	2Eh	2Eh	n.a.
segment DS	3Eh	3Eh	n.a.
segment ES	26h	26h	n.a.
segment SS	36h	36h	n.a.
segment FS	64h	64h	64h
segment GS	65h	65h	65h
opération sur chaîne REP ou REPE	F3h	F3h	F3h
opération sur chaîne REPNE	F2h	F2h	F2h
opérande mémoire verrouillé	F0h	F0h	F0h
registres R8 – R15, XMM8 – XMM15 dans le champ reg	n.a.	n.a.	REX.R (44h)
registres R8 – R15, XMM8 – XMM15 dans le champ r/m	n.a.	n.a.	REX.B (41h)
registres R8 – R15 dans le champ SIB. base	n.a.	n.a.	REX.B (41h)
registres R8 – R15 dans le champ SIB. index	n.a.	n.a.	REX.X (42h)
registres SIL, DIL, BPL, SPL	n.a.	n.a.	REX (40h)
réduction de branchement pris (Intel NetBurst unique-	3Eh	3Eh	3Eh

ment)			
rrédiction de branchement NON pris (Intel NetBurst uni- quement)	2Eh	2Eh	2Eh
préservation du registre de bornes lors d'un saut (MPX)	F2h	F2h	F2h
préfixe VEX, 2 octets	C5h	C5h	C5h
préfixe VEX, 3 octets	C4h	C4h	C4h
préfixe XOP, 3 octets (AMD seulement)	8Fh	8Fh	8Fh
préfixe EVEX, 4 octets (AVX-512)	62h	62h	62h
préfixe MVEX, 4 octets (Intel Knights Corner seulement)	n.a.	62h	62h

Tableau 3.12. Préfixes d'instruction

Les préfixes de segment sont rarement nécessaires dans un modèle de mémoire *flat*. Le préfixe de segment DS n'est requis que si un opérande mémoire utilise un registre de base BP, EBP ou ESP et que l'on souhaite cibler le segment DS plutôt que le segment SS.

Le préfixe de verrouillage (*lock*) n'est autorisé qu'avec certaines instructions effectuant une opération de lecture, modification et écriture sur un opérande mémoire.

Les préfixes de prédiction de branchement ne fonctionnent que sur l'architecture Intel NetBurst (Pentium 4) et sont rarement nécessaires.

Il ne peut y avoir qu'un seul préfixe REX. Si plusieurs préfixes REX sont nécessaires, leurs valeurs sont combinées par un OR logique pour former un octet unique compris entre 40h et 4Fh. Ces préfixes ne sont disponibles qu'en mode 64 bits. Les octets 40h à 4Fh correspondent à des codes d'instruction en modes 16 bits et 32 bits ; ces instructions (INC r et DEC r) sont codées différemment en mode 64 bits.

Les préfixes peuvent être insérés dans n'importe quel ordre, à l'exception des préfixes REX et des préfixes multi-octets (VEX, XOP, EVEX, MVEX), qui doivent être placés après tous les autres préfixes.

Le jeu d'instructions AVX utilise des préfixes de 2 ou 3 octets appelés préfixes VEX. Ces préfixes VEX intègrent des bits destinés à remplacer les préfixes 66, F2, F3 et REX, ainsi que les octets d'échappement 0F, 0F 38 et 0F 3A des opcodes multi-octets. Ils comportent également des bits pour spécifier les registres YMM et un opérande de registre supplémentaire, ainsi que des bits réservés aux extensions futures. Les préfixes EVEX et MVEX sont similaires aux préfixes VEX, mais incluent des bits supplémentaires pour prendre en charge davantage de registres, les opérations avec masque et d'autres fonctionnalités. Aucun préfixe supplémentaire n'est autorisé après un préfixe VEX, EVEX ou MVEX. Les seuls préfixes autorisés avant un préfixe VEX, EVEX ou MVEX sont les préfixes de segment et les préfixes de taille d'adresse.

Les préfixes inutiles, redondants ou mal placés sont ignorés, à l'exception des préfixes LOCK et VEX. Toutefois, des préfixes sans effet dans un contexte donné pourraient en avoir un sur les processeurs futurs.

Des préfixes superflus peuvent être utilisés à la place d'instructions NOP pour aligner le code, mais un nombre excessif de préfixes risque de ralentir le décodage des instructions sur certains processeurs.

Il est possible d'utiliser un nombre quelconque de préfixes, tant que la longueur totale de l'instruction ne dépasse pas 15 octets. Par exemple, une instruction MOV EAX, EBX précédée de dix préfixes de segment ES fonctionnera toujours, mais son décodage pourra prendre plus de temps.

4 Normes ABI

ABI signifie *Application Binary Interface* (interface binaire d'application). Une ABI définit une norme concernant l'appel des fonctions, le transfert des paramètres et des valeurs de retour, ainsi que les registres qu'une fonction est autorisée à modifier. Il est important de respecter la norme ABI appropriée lorsque l'on combine du code assembleur avec un langage de haut niveau. Les détails relatifs aux conventions d'appel, entre autres, sont traités dans le manuel n° 5 : “Conventions d'appel pour différents compilateurs C++ et systèmes d'exploitation”. Les règles les plus importantes sont résumées ici pour plus de commodité.

4.1 Utilisation des registres

	16 bits DOS, Windows	32 bits Windows, Unix	64 bits Windows	64 bits Unix
--	----------------------------	-----------------------------	--------------------	-----------------

Registres qui peuvent être utilisés librement	AX, BX, CX, DX, ES, ST(0)-ST(7)	EAX, ECX, EDX, ST(0)-ST(7), XMM0-XMM7, YMM0-YMM7, ZMM0-ZMM7	RAX, RCX, RDX, R8-R11, ST(0)-ST(7), XMM0-XMM5, YMM0-YMM5, YMM6H-YMM15H, ZMM16-ZMM31	RAX, RCX, RDX, RSI, RDI, R8-R11, ST(0)-ST(7), XMM0-XMM15, YMM0-YMM15, ZMM16-ZMM31
Registres devant être sauvegardés et restaurés	SI, DI, BP, DS	EBX, ESI, EDI, EBP	RBX, RSI, RDI, RBP, R12-R15, XMM6-XMM15	RBX, RBP, R12-R15
Registres qui ne peuvent pas être modifiés		DS, ES, FS, GS, SS		
Registres utilisés pour le transfert de paramètres		(ECX)	RCX, RDX, R8, R9, XMM0-XMM3, YMM0-YMM3, ZMM0-ZMM3	RDI, RSI, RDX, RCX, R8, R9, XMM0-XMM7, YMM0-YMM7, ZMM0-ZMM7
Registres utilisés pour les valeurs de retour	AX, DX, ST(0)	EAX, EDX, ST(0)	RAX, XMM0, YMM0, ZMM0	RAX, RDX, XMM0, XMM1, YMM0, ZMM0, ST(0), ST(1)

Tableau 4.1. Utilisation des registres

Les registres à virgule flottante ST(0) à ST(7) doivent être vides avant tout appel ou retour, sauf lorsqu'ils sont utilisés pour renvoyer une valeur de fonction. Les registres MMX doivent être réinitialisés via l'instruction EMMS avant tout appel ou retour. Les registres YMM doivent être réinitialisés via l'instruction VZERoupper avant tout appel ou retour vers du code non-VEX.

Les flags arithmétiques peuvent être modifiés librement. Le flag de direction peut être positionné temporairement, mais doit être réinitialisé avant tout appel ou retour dans les systèmes 32 bits et 64 bits. Le flag d'interruption ne peut pas être désactivé dans les systèmes d'exploitation fonctionnant en mode protégé. Le mot de contrôle de virgule flottante ainsi que les bits 6 à 15 du registre MXCSR doivent être sauvegardés et restaurés par les fonctions qui les modifient.

Les registres FS et GS sont utilisés pour les blocs d'informations de thread (TIB), etc., et ne doivent pas être modifiés. Les autres registres de segment ne doivent pas être modifiés, sauf dans le cas des modèles segmentés 16 bits.

4.2 Stockage de données

Les variables et objets déclarés dans une fonction en C ou C++ sont stockés sur la pile et adressés par rapport au pointeur de pile ou à un cadre de pile. Il s'agit du moyen le plus efficace de stocker des données, pour deux raisons. Premièrement, l'espace de pile utilisé pour le stockage local est libéré au retour de la fonction et peut être réutilisé par la prochaine fonction appelée. L'utilisation répétée de la même zone mémoire améliore la mise en cache des données. La deuxième raison est que les données stockées sur la pile peuvent souvent être adressées avec un décalage de 8 bits par rapport à un pointeur plutôt que les 32 bits requis pour adresser les données dans le segment de données. Cela rend le code plus compact afin qu'il prenne moins de place dans le cache de code ou le cache de trace.

Les données globales et statiques en C++ sont stockées dans le segment de données et adressées avec des adresses absolues de 32 bits dans les systèmes 32 bits et avec des adresses relatives RIP de 32 bits dans les systèmes 64 bits. Une troisième façon de stocker des données en C++ consiste à allouer de l'espace avec `new` ou `malloc`. Cette méthode doit être évitée si la vitesse est critique.

4.3 Conventions d'appel de fonction

Les conventions d'appel de fonctions sont décrites en détail dans le manuel 5 : "Conventions d'appel pour différents compilateurs C++ et systèmes d'exploitation". Les règles les plus importantes sont exposées ci-après.

Convention d'appel en mode 16 bits sous DOS et Windows 3.x

Les paramètres de fonction sont transmis via la pile, le premier paramètre étant placé à l'adresse la plus basse. Cela revient à empiler le dernier paramètre en premier. Le nettoyage de la pile est effectué par l'appelant.

Les paramètres de 8 ou 16 bits occupent un mot d'espace sur la pile. Les paramètres de plus de 16 bits sont stockés au format “little-endian”, c'est-à-dire avec le mot de poids faible à l'adresse la plus basse. Tous les paramètres de la pile sont alignés sur une frontière de 2 octets.

Dans la plupart des cas, les valeurs de retour des fonctions sont transmises dans des registres. Les entiers de 8 bits sont renvoyés dans AL, les entiers de 16 bits et les pointeurs “near” dans AX, les entiers de 32 bits et les pointeurs “far” dans DX:AX, les booléens dans AX et les valeurs à virgule flottante dans ST(0).

Convention d'appel sous Windows 32 bits, Linux, BSD et Mac OS X

Les paramètres de fonction sont transmis sur la pile selon les conventions d'appel suivantes :

Convention d'appel	Ordre des paramètres sur la pile	Paramètres supprimés par
<code>__cdecl</code>	Premier param. à l'adresse basse	l'appelant
<code>__stdcall</code>	Premier param. à l'adresse basse	le sous-programme
<code>__fastcall</code> Microsoft et Gnu	Les 2 premiers paramètres dans ECX et EDX. Les autres sont traités par <code>__stdcall</code> .	le sous-programme
<code>__fastcall</code> Borland	Les 3 premiers paramètres dans EAX, EDX, ECX. Le reste comme <code>__stdcall</code> .	le sous-programme
<code>_pascal</code>	Premier param. à l'adresse haute	le sous-programme
<code>__thiscall</code> Microsoft	this dans ECX. Le reste comme <code>__stdcall</code> .	le sous-programme

Tableau 4.2. Conventions d'appel en mode 32 bits

La convention d'appel `__cdecl` est celle utilisée par défaut sous Linux. Sous Windows, la convention `__cdecl` est également la norme par défaut, sauf pour les fonctions membres, les fonctions système et les DLL. Les modules liés statiquement (fichiers .obj et .lib) devraient de préférence utiliser `__cdecl`, tandis que les bibliothèques de liens dynamiques (fichiers .dll) devraient utiliser `__stdcall`.

La convention d'appel la plus rapide pour les fonctions prenant des paramètres entiers est `__fastcall`, mais cette convention n'est pas normalisée.

Rappelez-vous que le pointeur de pile est décrémenté lorsqu'une valeur est empilée. Cela signifie que le premier paramètre empilé se trouvera à l'adresse la plus élevée, conformément à la convention `_pascal`. Pour respecter les conventions `__cdecl` et `__stdcall`, vous devez empiler les paramètres dans l'ordre inverse.

Les paramètres d'une taille égale ou inférieure à 32 bits occupent 4 octets d'espace sur la pile. Les paramètres de plus de 32 bits sont stockés au format “little-endian” (le DWORD de poids faible à l'adresse la plus basse) et sont alignés sur 4 octets.

Sous Mac OS X, le compilateur GNU (version 3 et ultérieures) aligne la pile sur une frontière de 16 octets avant chaque instruction `call`, bien que ce comportement ne soit pas systématique. Il est possible de spécifier des alignements différents, ce qui peut entraîner des incompatibilités. Consultez le manuel n° 5, “Calling conventions for different C++ compilers and operating systems”, pour plus de détails.

Dans la plupart des cas, les valeurs de retour des fonctions sont transmises via des registres. Les entiers de 8 bits sont renvoyés dans AL, les entiers de 16 bits dans AX, les entiers de 32 bits, les pointeurs, les références et les booléens dans EAX, les entiers de 64 bits dans EDX:EAX et les valeurs à virgule flottante dans ST(0).

Consultez le manuel n° 5, “Calling conventions for different C++ compilers and operating systems”, pour plus de détails concernant les paramètres de types composés (struct, class, union) et de types vectoriels (`__m64`, `__m128`, `__m256`, `__m512`).

Conventions d'appel sous Windows 64 bits

Le premier paramètre est transmis dans RCX s'il s'agit d'un entier, ou dans XMM0 s'il s'agit d'un nombre float ou double. Le deuxième paramètre est transmis dans RDX ou XMM1. Le troisième paramètre est transmis dans R8 ou XMM2. Le quatrième paramètre est transmis dans R9 ou XMM3. Notez que RCX n'est pas utilisé pour la transmission de paramètres si XMM0 est utilisé, et inversement. Au maximum quatre paramètres peuvent être transmis via des registres, quel que soit leur type. Les paramètres supplémentaires sont transmis sur la pile, le premier paramètre étant placé à l'adresse la plus basse et aligné sur 8 octets. Pour les fonctions membres, le pointeur “this” constitue le premier paramètre.

L'appelant doit allouer 32 octets d'espace libre sur la pile, en plus des paramètres éventuellement transmis sur la pile. Il s'agit d'une zone de réserve (“shadow space”) où la fonction appelée peut sauvegarder les quatre registres de paramètres si nécessaire. Cette zone correspond à l'emplacement où les quatre premiers paramètres auraient été stockés s'ils avaient été transmis sur la pile selon la convention `__cdecl`. Cette zone appartient à la fonction appelée, qui est libre d'y stocker les paramètres (ou toute autre donnée). L'appelant doit réserver ces 32 octets de zone de réserve, même pour les fonctions ne prenant aucun paramètre. L'appelant est responsable du nettoyage de la pile, y compris de la zone de réserve. Les valeurs de retour sont placées dans RAX ou XMM0.

Le pointeur de pile doit être aligné sur 16 octets avant toute instruction CALL, de sorte que la valeur de RSP soit égale à 8 modulo 16 à l'entrée de la fonction. La fonction peut se fier à cet alignement lorsqu'elle sauvegarde des registres XMM dans la pile.

Consultez le manuel n° 5, “Calling conventions for different C++ compilers and operating systems”, pour plus de détails sur les paramètres de types composés (struct, class, union) et de types vectoriels (__m64, __m128, __m256, __m512).

Conventions d'appel sous Linux, BSD et Mac OS X 64 bits

Les six premiers paramètres de type entier sont transmis respectivement dans les registres RDI, RSI, RDX, RCX, R8 et R9. Les huit premiers paramètres en virgule flottante sont transmis dans les registres XMM0 à XMM7. Tous ces registres peuvent être utilisés, permettant ainsi de transmettre jusqu'à quatorze paramètres via les registres. Les paramètres supplémentaires sont transmis sur la pile, les premiers étant placés à l'adresse la plus basse et alignés sur 8 octets. C'est l'appelant qui nettoie la pile si des paramètres y ont été déposés. Il n'y a pas de zone de réserve (“shadow space”). Pour les fonctions membres, le pointeur “this” constitue le premier paramètre. Les valeurs de retour sont placées dans RAX ou XMM0.

Le pointeur de pile doit être aligné sur 16 octets avant toute instruction CALL, de sorte que la valeur de RSP soit égale à 8 modulo 16 à l'entrée de la fonction. La fonction peut se fier à cet alignement lorsqu'elle sauvegarde les registres XMM sur la pile.

La plage d'adresses allant de [RSP-1] à [RSP-128] est appelée “zone rouge” (*red zone*). Une fonction peut stocker des données en toute sécurité au-dessus de la pile, dans la zone rouge, tant que celles-ci ne sont pas écrasées par des instructions PUSH ou CALL.

Consultez le manuel 5, “Calling conventions for different C++ compilers and operating systems” (Conventions d'appel pour différents compilateurs C++ et systèmes d'exploitation), pour plus de détails sur les paramètres de types composés (struct, class, union) et de types vectoriels (__m64, __m128, __m256, __m512).

4.4 Altération et décoration de noms

La prise en charge de la surcharge de fonctions en C++ nécessite de fournir à l'éditeur de liens des informations sur les paramètres d'une fonction. Cela s'effectue en ajoutant au nom de la fonction des codes correspondant aux types de ses paramètres ; ce procédé est appelé “name mangling” (ou décoration de noms). Traditionnellement, ces codes de décoration étaient spécifiques à chaque compilateur. Heureusement, on observe une tendance croissante à la normalisation dans ce domaine afin d'améliorer la compatibilité entre les différents compilateurs. Les codes de décoration utilisés par les divers compilateurs sont décrits en détail dans le manuel n° 5 : “Conventions d'appel pour différents compilateurs C++ et systèmes d'exploitation”.

La manière la plus simple de résoudre le problème d'incompatibilité des codes de décoration consiste à utiliser des déclarations extern “C”. Les fonctions déclarées avec extern “C” ne font pas l'objet de cette décoration de nom ; la seule modification apportée au nom est l'ajout d'un préfixe de type “underscore” (trait de soulignement) sous Windows (versions 16 et 32 bits) et sous Mac OS (versions 32 et 64 bits). Une décoration supplémentaire du nom s'applique toutefois aux fonctions déclarées avec __stdcall et __fastcall.

La déclaration extern “C” ne peut pas être utilisée pour les fonctions membres, les fonctions surchargées, les opérateurs et autres constructions non prises en charge par le langage C. Dans ces cas, vous pouvez éviter la décoration de noms en définissant une fonction dont le nom est décoré, laquelle appelle une fonction dont le nom ne l'est pas. Si la fonction au nom décoré est définie comme *inline*, le compilateur remplacera simplement l'appel à cette fonction par un appel à la fonction au nom non décoré. Par exemple, pour définir un opérateur C++ surchargé en assembleur sans décoration de nom :

```
class C1;
// fonction d'assemblage au nom non altéré ;
extern "C" C1 cplus (C1 const & a, C1 const & b);
// opérateur C++ dont le nom a été décoré
inline C1 operator + (C1 const & a, C1 const & b) {
    // opérateur + remplacé en ligne par la fonction cplus
    return cplus(a, b);
}
```

Les fonctions surchargées peuvent être rendues “inline” de la même manière. Les fonctions membres de classe peuvent être traduites en fonctions amies, comme l'illustre l'exemple 7.1b à la page 38.

4.5 Exemples de fonctions

Les exemples suivants montrent comment coder une fonction en assembleur qui obéit aux conventions d'appel. D'abord le code en C++ :

```
// Exemple 4.1a
extern "C" double sinxpnx (double x, int n) {
```

```

        return sin(x) + n * x;
    }

```

La même fonction peut être codée en langage assembleur. Les exemples suivants présentent la même fonction codée pour différentes plateformes.

; **Exemple 4.1b.** Windows 32 bits

```

extern _sin
global _sinxpnx:function
align 4
_sinxpnx:
; paramètre x = [esp+4]
; paramètre n = [esp +12]
; valeur de retour = st(0)
    fld qword [esp+4]      ; x
    sub esp, 8             ; on fait de la place pour le paramètre x
    fstp qword [esp]       ; stockage du paramètre pour sin ; effacement de st(0)
    call _sin              ; fonction de bibliothèque pour sin()
    add esp, 8             ; nettoyage de la pile après le CALL
    fild dword [esp+12]    ; n
    fmul qword [esp+4]     ; n*x
    fadd                   ; sin(x) + n*x
    ret                   ; la valeur de retour est dans st(0)

```

Ici, j'ai choisi d'utiliser la fonction de bibliothèque `_sin` plutôt que `FSIN`. Cela peut s'avérer plus rapide dans certains cas, car `FSIN` offre une précision supérieure aux besoins. Le paramètre de `_sin` est transmis sur la pile sous la forme de 8 octets.

; **Exemple 4.1c.** Linux 32 bits

```

extern _sin
global sinxpnx:function
align 4
sinxpnx:
; paramètre x = [esp+4]
; paramètre n = [esp +12]
; valeur de retour = st(0)
    fld qword [esp+4]      ; x
    sub esp, 12            ; alignement de la pile sur 16 octets avant le CALL
    fstp qword [esp]       ; stockage du paramètre pour sin ; effacement de st(0)
    call _sin              ; fonction de bibliothèque pour sin()
    add esp, 12            ; nettoyage de la pile après le CALL
    fild dword [esp+12]    ; n
    fmul qword [esp+4]     ; n*x
    fadd                   ; sin(x) + n*x
    ret                   ; la valeur de retour est dans st(0)

```

Sous Linux 32 bits, les noms de fonction ne comportent pas de souligné. Sous Linux (avec GCC version 3 ou ultérieure), la pile doit être alignée sur 16 octets. L'appel à `sinxpnx` soustrait 4 à ESP. Nous soustrayons 12 octets supplémentaires à ESP afin que la valeur totale soustraite soit de 16. Il est possible de soustraire davantage, à condition que le total soit un multiple de 16. Dans l'exemple 4.1b ci-dessus, nous n'avons soustrait que 8 à ESP car, sous Windows 32 bits, la pile n'est alignée que sur 4 octets.

; **Exemple 4.1d.** Windows 64 bits

```

extern sin
global sinxpnx:function
align 4
sinxpnx:
; paramètre x = xmm0
; paramètre n = edx
; valeur de retour = xmm0
    push rbx               ; rbx doit être sauvegardé
    movaps [rsp+16], xmm6   ; sauvegarde de xmm6 dans mon espace sombre
    sub rsp, 32             ; espace sombre pour l'appel à sin
    mov ebx, edx            ; sauvegarde de n
    movsd xmm6, xmm0        ; sauvegarde de x
    call sin                ; xmm0 = sin(xmm0)

```

```

cvtsi2sd xmm1, ebx      ; conversion de n en un double
mulsd xmm1, xmm6        ; n * x
addsd xmm0, xmm1        ; sin(x) + n * x
add rsp, 32             ; restauration du pointeur de pile
movaps xmm6, [rsp+16]   ; restauration de xmm6
pop rbx                 ; restauration de rbx
ret                     ; la valeur de retour est dans xmm0

```

Sous Windows 64 bits, les paramètres des fonctions sont transmis via des registres. Le registre ECX n'est pas utilisé pour la transmission des paramètres car le premier paramètre n'est pas un entier. Nous utilisons RBX et XMM6 pour conserver les valeurs de n et x durant l'appel à `sin`. Pour ce faire, nous devons utiliser des registres dont la sauvegarde incombe à la fonction appelée (“callee-save”) et les sauvegarder sur la pile avant de les utiliser. La pile doit être alignée sur 16 octets avant l'appel à `sin`; nous pouvons supposer qu'elle l'est déjà avant l'appel à `sinxpnx`. L'appel à `sinxpnx` soustrait 8 à RSP, l'instruction `PUSH RBX` en soustrait 8 et l'instruction `SUB` en soustrait 32. La valeur totale soustraite est de $8 + 8 + 32 = 48$, ce qui constitue un multiple de 16, garantissant ainsi le maintien de l'alignement requis. Les 32 octets supplémentaires sur la pile constituent la zone de réserve (“shadow space”) pour l'appel à `sin`. À noter que l'exemple 4.1d ne prévoit pas la gestion des exceptions. Il est nécessaire d'ajouter des tables contenant les informations de dépilement (“stack unwind”) si le programme doit intercepter des exceptions générées au sein de la fonction `sin`.

```

; Exemple 4.1e. Linux 64 bits
extern sin
global sinxpnx: function
align 4
sinxpnx:
; paramètre x = xmm0
; paramètre n = edi
; valeur de retour = xmm0

    push rbx                ; rbx doit être sauvegardé
    sub rsp, 16             ; Création de l'espace local et alignement de la pile sur 16
    movaps [rsp], xmm0      ; sauvegarde de x
    mov ebx, edi            ; sauvegarde de n
    call sin                ; xmm0 = sin(xmm0)
    cvtsi2sd xmm1, ebx      ; conversion de n en un double
    mulsd xmm1, [rsp]       ; n * x
    addsd xmm0, xmm1        ; sin(x) + n * x
    add rsp, 16             ; restauration du pointeur de pile
    pop rbx                 ; restauration de rbx
    ret                     ; la valeur de retour est dans xmm0

```

Sous Linux 64 bits, les registres utilisés pour le passage des paramètres diffèrent de ceux utilisés sous Windows 64 bits. Il n'existe pas de registres XMM devant être préservés par la fonction appelée (*callee-save*) ; il est donc nécessaire de sauvegarder x dans la pile durant l'appel à `sin`, bien qu'une sauvegarde dans un registre puisse s'avérer plus rapide (sauvegarder x dans un registre d'entiers 64 bits est possible, mais lent). En revanche, n peut être sauvegardé dans un registre d'usage général devant être préservé par la fonction appelée. La pile est alignée sur 16 octets. Aucune zone de réserve (*shadow space*) n'est requise dans la pile. La “zone rouge” (*red zone*) ne peut pas être utilisée, car elle serait écrasée lors de l'appel à `sin`. À noter que l'exemple 4.1e ne prend pas en charge la gestion des exceptions. Il est nécessaire d'ajouter des tables contenant les informations de déroulement de la pile (*stack unwind information*) si le programme doit intercepter des exceptions générées au sein de la fonction `sin`.

5 Utilisation des fonctions intrinsèques en C++

Comme indiqué précédemment, il existe plusieurs façons de combiner l'assembleur avec du code écrit dans un langage de haut niveau : l'utilisation de l'assembleur en ligne en C++, des fonctions intrinsèques en C++, des classes vectorielles en C++, ainsi que la création de modules d'assembleur distincts. L'assembleur en ligne est décrit au chapitre 6, page 25. Les fonctions intrinsèques sont décrites dans ce chapitre. Les classes vectorielles sont décrites dans le manuel 1 : “Optimizing software in C++”.

L'utilisation de fonctions intrinsèques et de classes vectorielles est vivement recommandée, car elles sont bien plus simples et sûres à employer que la syntaxe de l'assembleur. Les compilateurs C++ de Microsoft, Intel, GNU et Clang prennent en charge les fonctions intrinsèques. La plupart de ces fonctions génèrent chacune une instruction machine unique ; une fonction intrinsèque équivaut donc à une instruction assembleur.

La programmation à l'aide de fonctions intrinsèques s'apparente à de l'assembleur de haut niveau. Elle peut aisément être combinée avec des constructions du langage C++, telles que les instructions conditionnelles, les

boucles, les fonctions, les classes et la surcharge d'opérateurs. L'utilisation de fonctions intrinsèques constitue une méthode plus simple pour effectuer de la programmation en assembleur de haut niveau que le recours aux directives de contrôle (comme `.if`) dans un assembleur classique ou à l'utilisation d'un assembleur dit de haut niveau (HLA).

L'introduction des fonctions intrinsèques a grandement facilité la réalisation de tâches de programmation qui, auparavant, nécessitaient l'utilisation de la syntaxe assembleur. Les avantages liés à l'utilisation de fonctions intrinsèques sont les suivants :

- Il n'est pas nécessaire d'apprendre la syntaxe de l'assembleur.
- Intégration transparente dans le code C++.
- Les branchements, boucles, fonctions, classes, etc., s'écrivent facilement grâce à la syntaxe C++.
- Le compilateur gère les conventions d'appel, les conventions d'utilisation des registres, etc.
- Le code est portable sur la quasi-totalité des plateformes x86 : Windows (32 et 64 bits), Linux, Mac OS, etc.
- Le code est compatible avec les compilateurs Microsoft, GNU, Clang et Intel.
- Le compilateur gère les variables en registre, l'allocation des registres et le débordement de registres (*spilling*). Le programmeur n'a pas à se soucier de savoir quel registre est utilisé pour quelle variable.
- Différentes instances d'une même fonction ou d'un même opérateur **inline** peuvent utiliser des registres différents pour leurs paramètres. Cela élimine le besoin de transferts de registre à registre. La même fonction écrite en assembleur utiliserait généralement un registre spécifique pour chaque paramètre ; une instruction de transfert serait donc nécessaire si la valeur se trouvait dans un registre différent.
- Il est possible de définir des opérateurs surchargés pour les fonctions intrinsèques. Par exemple, l'instruction additionnant deux vecteurs de quatre nombres à virgule flottante (*floats*) s'écrit `ADDPS` en assembleur et `_mm_add_ps` avec les fonctions intrinsèques. Toutefois, on peut définir un opérateur surchargé pour cette dernière afin qu'elle s'écrive simplement `+` lors de l'utilisation de classes vectorielles. Le code ressemble alors à du C++ standard.
- Le compilateur peut optimiser davantage le code, par exemple via l'élimination des sous-expressions communes, le déplacement de code invariant hors des boucles, l'ordonnancement et la réorganisation des instructions, etc. Ces opérations devraient être effectuées manuellement si l'on utilisait la syntaxe assembleur. Les compilateurs Clang et GNU offrent les meilleures optimisations.

Les inconvénients de l'utilisation des fonctions intrinsèques sont :

- Toutes les instructions d'assemblage ne possèdent pas d'équivalents sous forme de fonctions intrinsèques.
- Les noms des fonctions sont parfois longs et difficiles à mémoriser.
- Une expression faisant appel à de nombreuses fonctions intrinsèques peut paraître lourde et difficile à lire.
- Cela nécessite une bonne compréhension des mécanismes sous-jacents.
- Certains compilateurs ne parviennent pas à optimiser le code contenant des fonctions intrinsèques aussi efficacement que le reste du code, notamment lorsque la propagation de constantes est requise.
- Une utilisation maladroite des fonctions intrinsèques peut rendre le code moins performant qu'un code C++ standard.
- Le compilateur peut modifier le code ou l'implémenter d'une manière moins efficace que ce que le programmeur avait prévu. Il peut s'avérer nécessaire d'examiner le code généré par le compilateur pour vérifier s'il est optimisé conformément aux intentions du programmeur.
- Le mélange des types `_m128` et `_m256` peut entraîner des ralentissements importants si certaines règles ne sont pas respectées. Appelez `_mm256_zeroupper()` avant toute transition entre des modules compilés avec AVX activé et des modules ou fonctions de bibliothèque compilés sans AVX.

5.1 Utilisation de fonctions intrinsèques pour le code système

Les fonctions intrinsèques sont utiles pour écrire du code système et accéder à des registres système inaccessibles via le C++ standard. Certaines de ces fonctions sont énumérées ci-dessous.

Fonctions d'accès aux registres système :

```
_rdtsc, __readpmc, __readmsr, __readcr0, __readcr2, __readcr3, __readcr4,
__readcr8, __writecr0, __writecr3, __writecr4, __writecr8, __writemsr,
__mm_getcsr, __mm_setcsr, __getcalleseflags.
```

Fonctions d'entrée et de sortie :

```
__inbyte, __inword, __indword, __outbyte, __outword, __outdword.
```

Fonctions pour les opérations atomiques de lecture/écriture en mémoire :

```
_InterlockedExchange, etc.
```

Fonctions d'accès aux segments FS et GS :

`__readfsbyte`, `__writefsbyte`, etc.

Instructions de contrôle du cache (nécessitent le jeu d'instructions SSE ou SSE2) :

`_mm_prefetch`, `_mm_stream_si32`, `_mm_stream_pi`, `_mm_stream_si128`, `_ReadBarrier`,
`_WriteBarrier`, `_ReadWriteBarrier`, `_mm_sfence`.

Autres fonctions système :

`__cpuid`, `__debugbreak`, `_disable`, `_enable`.

5.2 Utilisation de fonctions intrinsèques pour des instructions non disponibles en C++ standard

Certaines instructions simples, indisponibles en C++ standard, peuvent être implémentées à l'aide de fonctions intrinsèques, comme par exemple les fonctions de rotation de bits, de recherche de bits, etc. :

`_rotl8`, `_rotr8`, `_rotl16`, `_rotr16`, `_rotl`, `_rotr`, `_rotl64`, `_rotr64`, `_BitScanForward`,
`_BitScanReverse`.

5.3 Utilisation de fonctions intrinsèques pour les opérations vectorielles

Les instructions vectorielles sont très utiles pour accélérer le code présentant un parallélisme inhérent. Il existe des fonctions intrinsèques pour la quasi-totalité des instructions opérant sur des registres vectoriels.

L'utilisation de ces fonctions intrinsèques pour les opérations vectorielles est décrite en détail dans le manuel 1 : "Optimizing software in C++".

5.4 Disponibilité des fonctions intrinsèques

Les fonctions intrinsèques sont disponibles dans les compilateurs Microsoft, GNU, Clang et Intel. La plupart d'entre elles portent le même nom quel que soit le compilateur. Pour y accéder, il est nécessaire d'inclure le fichier d'en-tête approprié. Certaines fonctions intrinsèques ne sont pas prises en charge par tous ces compilateurs.

Ces fonctions sont répertoriées dans la documentation d'aide de chaque compilateur, dans les fichiers d'en-tête correspondants, sur docs.microsoft.com, dans le manuel "Intel 64 and IA-32 Architectures Software Developer's Manual" (developer.intel.com) ainsi que dans le "Intel Intrinsic Guide" disponible à l'adresse <https://software.intel.com/sites/landingpage/IntrinsicsGuide/>

6 Utilisation de l'assembleur en ligne

L'assembleur en ligne (*inline*) est un autre moyen d'insérer du code assembleur dans un fichier C++. Les mots-clés `asm`, `_asm`, `__asm` ou `__asm__` indiquent au compilateur qu'il s'agit de code assembleur. Les différents compilateurs utilisent des syntaxes variées pour l'assembleur en ligne ; ces syntaxes sont décrites ci-dessous.

Les avantages de l'utilisation de l'assembleur en ligne sont les suivants :

- Il est facile de l'associer au C++.
- Les variables et autres symboles définis dans le code C++ sont accessibles depuis le code assembleur.
- Seule la partie du code impossible à écrire en C++ est codée en assembleur.
- Toutes les instructions assembleur sont disponibles.
- Le code généré correspond exactement à ce que vous écrivez.
- Il permet une optimisation très fine.
- Le compilateur gère les conventions d'appel, la transformation des noms (*name mangling*) et la sauvegarde des registres.
- Le compilateur peut intégrer (*inline*) une fonction contenant de l'assembleur en ligne.
- Le code est portable entre différentes plateformes x86 lors de l'utilisation des compilateurs GNU, Clang ou Intel.

Les inconvénients de l'utilisation de l'assembleur en ligne sont :

- Différents compilateurs utilisent des syntaxes différentes.
- Nécessite des connaissances en langage d'assemblage.
- Nécessite une bonne compréhension du fonctionnement du compilateur. Il est facile de commettre des erreurs.
- L'allocation des registres s'effectue principalement manuellement. Le compilateur peut allouer des registres différents pour les mêmes variables.
- Le compilateur ne peut pas optimiser efficacement le code d'assemblage en ligne.
- Il peut être difficile de contrôler le code de prologue et d'épilogue des fonctions.

- Il peut s'avérer impossible de définir des données.
- Il peut s'avérer impossible d'utiliser des macros et d'autres directives.
- Il peut s'avérer impossible de créer des fonctions à points d'entrée multiples.
- Vous risquez de mélanger par inadvertance des instructions VEX et non-VEX, ce qui entraîne des pénalités importantes sur certains processeurs (voir [paragraphe 13.1](#)).
- Le compilateur Microsoft ne prend pas en charge l'assemblage en ligne sur les plateformes 64 bits.

Les sections suivantes montrent comment utiliser l'assembleur en ligne avec différents compilateurs.

6.1 Assembleur en ligne de style MASM

La syntaxe de l'assembleur en ligne pour les compilateurs Microsoft et Intel sous Windows est de type MASM. Malheureusement, cette syntaxe est mal documentée, voire pas du tout, dans les manuels des compilateurs. Je vais donc la décrire brièvement ici.

Les exemples suivants présentent une fonction qui élève un nombre à virgule flottante x à une puissance entière n . L'algorithme consiste à multiplier x^1, x^2, x^4, x^8 , etc., en fonction de chaque bit de la représentation binaire de n . En réalité, il n'est pas nécessaire de coder cela en assembleur, car un bon compilateur effectuera une optimisation quasi équivalente si l'on se contente d'écrire `pow(x, n)`. Mon objectif ici est simplement d'illustrer la syntaxe de l'assembleur en ligne.

Voici d'abord le code C++ illustrant l'algorithme :

```
// Exemple 6.1a. Élevation du nombre à virgule flottante x à la puissance de l'entier n.
double ipow (double x, int n) {
    unsigned int nn = abs(n); // valeur absolue de n
    double y = 1.0;          // utilisé pour la multiplication
    while (nn != 0) {         // boucle pour chaque bit dans nn
        if (nn & 1) y *= x;    // multiplication si le bit est égal à 1
        x *= x;               // Élevation au carré x
        nn >>= 1;             // Obtention du bit suivant de nn
    }
    if (n < 0) y = 1.0 / y;    // réciproque si n est négatif
    return y;                 // retour de y = pow(x,n)
}
```

Et voici le code optimisé utilisant de l'assembleur en ligne avec une syntaxe de type MASM pour les compilateurs Microsoft ou Intel 32 bits :

```
// Exemple 6.1b. Assembleur en ligne de style MASM, mode 32 bits
double ipow (double x, int n) {
    __asm {
        mov eax, n                // Déplacement de n dans eax
        // abs(n) est calculé en inversant tous les bits
        // et en ajoutant 1 si n < 0 :
        cdq // Propagation du bit de signe à tous les bits de EDX.
        xor eax, edx              // Inversion des bits si la valeur est négative
        sub eax, edx              // Ajout de 1 si le nombre est négatif.
        // Maintenant, eax = abs(n).
        fldl // st(0) = 1.0
        jz L9                     // Fin if n = 0
        fld qword ptr x           // st(0) = x, st(1) = 1.0
        jmp L2                    // Saut dans la boucle
L1:                               // Haut de la boucle
        fmul st(0), st(0)         // Élevation au carré de x
L2:                               // Entrée de la boucle ici
        shr eax, 1               // Chaque bit de n est placé dans le flag de retenue.
        jnc L1                   // Pas de retenue. On ignore la multiplication
        //et on passe à l'étape suivante.
        fmul st(1), st(0)        // Multiplication par x au carré i fois pour le bit n° i.
        jnz L1                   // Fin de la boucle. Arrêt lorsque nn = 0.
        fstp st(0)               // Élimination de st(0)
        test edx, edx            // Vérification si n était négatif.
        jns L9                   // Terminaison si n n'était pas négatif
        fldl // st(0) = 1.0, st(1) = x^abs(n)
        fdivr                    // Réciproque
    }
```

```

    L9:      // Terminaison
    }        // Le résultat se trouve dans st(0).
#pragma warning(disable:1011)      // Ne pas avertir en cas de valeur de retour manquante
}

```

Notez que l'entrée de la fonction et ses paramètres sont déclarés selon la syntaxe C++. Le corps de la fonction, ou une partie de celui-ci, peut ensuite être codé en assembleur “inline” (intégré). Les paramètres `x` et `n`, déclarés en C++, sont directement accessibles dans le code assembleur sous ces mêmes noms. Le compilateur remplace simplement `x` et `n` dans le code assembleur par les opérandes mémoire appropriés, probablement `[esp+4]` et `[esp+12]`. Si le code assembleur intégré doit accéder à une variable située dans un registre, le compilateur la transfère vers une variable en mémoire (sur la pile) et insère l'adresse de cette variable dans le code assembleur.

Le résultat est renvoyé dans `st(0)`, conformément à la convention d'appel 32 bits. Le compilateur émet généralement un avertissement car l'instruction `return y;` est absente à la fin de la fonction. Cette instruction n'est pas nécessaire si l'on sait dans quel registre la valeur doit être renvoyée. La directive `#pragma warning(disable:1011)` permet de supprimer cet avertissement. Si vous souhaitez que le code fonctionne avec différentes conventions d'appel (par exemple sur des systèmes 64 bits), il est nécessaire de stocker le résultat dans une variable temporaire au sein du bloc assembleur :

```

// Exemple 6.1c. Style MASM, indépendant de la convention d'appel
double ipow (double x, int n) {
    double result; // Définition d'une variable temporaire pour le résultat
    __asm {
        mov eax, n
        cdq
        xor eax, edx
        sub eax, edx
        fldl
        jz L9
        fld qword ptr x
        jmp L2
    L1: fmul st(0), st(0)
    L2: shr eax, 1
        jnc L1
        fmul st(1), st(0)
        jnz L1
        fstp st(0)
        test edx, edx
        jns L9
        fldl
        fdivr
    L9: fstp qword ptr result // stockage du résultat dans une variable temporaire
    }
    return result;
}

```

Désormais, le compilateur gère tous les aspects de la convention d'appel et le code fonctionne sur toutes les plateformes x86.

Le compilateur examine le code assembleur en ligne pour déterminer quels registres sont modifiés. Il sauvegarde et restaure automatiquement ces registres si la convention d'utilisation des registres l'exige. Certains compilateurs interdisent la modification des registres `ebp` ou `ebx` dans le code assembleur en ligne, car ces registres sont nécessaires à la gestion de la trame de pile (*stack frame*) ; le compilateur émet généralement un avertissement dans ce cas.

Il est possible de supprimer le code de prologue et d'épilogue généré automatiquement en ajoutant `__declspec(naked)` à la déclaration de la fonction. Dans ce cas, il incombe au programmeur d'ajouter le code de prologue et d'épilogue nécessaire ainsi que de sauvegarder les registres modifiés, le cas échéant. La seule opération prise en charge par le compilateur pour une fonction “naked” est la décoration des noms (*name mangling*). La substitution automatique des noms de variables peut ne pas fonctionner avec ces fonctions, car elle dépend de la manière dont le prologue de la fonction est implémenté. Une fonction “naked” ne peut pas faire l'objet d'une introduction en ligne (*inlining*).

Accès aux variables de registre

Dans l'assembleur en ligne de style MASM, il n'est pas possible d'accéder directement aux variables stockées dans des registres en utilisant leurs noms symboliques. Accéder à une variable par son nom au sein d'un code assembleur en ligne contraint le compilateur à la placer dans un emplacement mémoire temporaire.

Si vous connaissez le registre dans lequel se trouve une variable, il vous suffit d'indiquer le nom de ce registre. Cette approche rend le code plus efficace, mais moins portable.

Par exemple, si le code de l'exemple précédent est utilisé sous Windows 64 bits, la variable `x` se trouvera dans le registre `XMM0` et la variable `n` dans le registre `EDX`. En tirant parti de cette information, nous pouvons optimiser le code ainsi :

```
// Exemple 6.1d. Style MASM, Windows 64 bits, compilateur Intel
double ipow (double x, int n) {
    const double one = 1.0;    // définir la constante à 1,0
    __asm {                    // x est dans XMM0
        mov eax, edx           // chargement de n dans EAX
        cdq
        xor eax, edx
        sub eax, edx
        movsd xmm1, one        // chargement de 1,0
        jz L9
        jmp L2

L1: mulsd xmm0, xmm0           // élévation au carré de x
L2: shr eax, 1
    jnc L1
    mulsd xmm1, xmm0           // multiplication par x au carré. i fois
    jnz L1
    movsd xmm0, xmm1           // placement du résultat dans XMM0
    test edx, edx
    jns L9
    movsd xmm0, one
    divsd xmm0, xmm1           // réciproque
L9: }
#pragma warning(disable:1011) // ne pas avertir en cas de valeur de retour manquante
}
```

Sous Linux 64 bits, la valeur de `n` se trouvera dans le registre `EDI` ; par conséquent, l'instruction `mov eax, edx` devra être remplacée par `mov eax, edi`.

Accès aux membres de classe et aux membres de structure

Prenons pour exemple une classe C++ contenant une liste d'entiers :

```
// Exemple 6.2a. Accès aux membres de données d'une classe
// définition de classe C++
class MyList {
protected:
    int length;                // Nombre d'éléments dans la liste
    int buffer[100];           // Stockage des éléments
public:
    MyList();                  // Constructeur
    void AttItem(int item);     // Ajout de l'élément à la liste
    int Sum();                  // Calcul de la somme des éléments
};

MyList::MyList() {             // Constructeur
    length = 0;
}

void MyList::AttItem(int item) { // Ajout de l'élément à la liste
    if (length < 100) {
        buffer[length++] = item;
    }
}

int MyList::Sum() {             // Fonction membre Sum
    int i, sum = 0;
    for (i = 0; i < length; i++) sum += buffer[i];
    return sum;
}
```

Je vais montrer ci-après comment coder la fonction membre `MyList::Sum` en assembleur "inline". Je n'ai pas cherché à optimiser le code ; mon objectif est simplement d'en illustrer la syntaxe.

L'accès aux membres de la classe s'effectue en chargeant “this” dans un registre de pointeur et en adressant les données membres par rapport à ce pointeur, à l'aide de l'opérateur point (.).

// **Exemple 6.2b.** Accès aux membres de classe, Windows 32 bits

```
int MyList::Sum() {
    __asm {
        mov ecx, this           // pointeur 'this'
        xor eax, eax           // sum = 0
        xor edx, edx           // boucle d'index, i = 0
        cmp [ecx].length, 0     // si (this->length != 0)
        je L9

    L1: add eax, [ecx].buffer[edx*4] // sum += buffer[i]
        add edx, 1              // i++
        cmp edx, [ecx].length   // while (i < length)
        jb L1

    L9:
    }                          // la valeur de retour se trouve dans EAX.
    #pragma warning(disable:1011)
}
```

Ici, on accède au pointeur “this” par son nom, et tous les membres de données de la classe sont référencés par rapport à “this”. Le décalage (offset) d'un membre de la classe par rapport à “this” s'obtient en écrivant le nom du membre précédé de l'opérateur point. L'indice du tableau nommé “buffer” doit être multiplié par la taille de chaque élément du tableau (soit [edx*4]).

Certains compilateurs 32 bits pour Windows placent “this” dans le registre ecx ; l'instruction `mov ecx, this` peut donc être omise. Les systèmes 64 bits nécessitent des pointeurs 64 bits ; il faut par conséquent remplacer ecx par rcx et edx par rdx. Sous Windows 64 bits, “this” se trouve dans rcx, tandis que sous Linux 64 bits, il se trouve dans rdi.

L'accès aux membres d'une structure s'effectue de la même manière : on charge un pointeur vers la structure dans un registre et on utilise l'opérateur point. Aucune vérification syntaxique n'est effectuée concernant l'accès aux membres privés ou protégés. Il n'existe aucun moyen de lever l'ambiguïté si plusieurs structures ou classes possèdent un membre portant le même nom. L'assembleur MASM permet de résoudre ce type d'ambiguïté grâce à la directive `assume` ou en plaçant le nom de la structure avant le point, mais cela n'est pas possible avec l'assembleur en ligne (*inline assembly*).

Appel de fonctions

Avec l'assembleur en ligne, les fonctions sont appelées par leur nom. Les fonctions membres ne peuvent être appelées que depuis d'autres fonctions membres de la même classe. Les fonctions surchargées ne peuvent pas être appelées, car il n'existe aucun moyen de lever l'ambiguïté. Il est impossible d'utiliser des noms de fonctions “décorés” (*mangled names*) dans de l'assembleur en ligne de style MASM. Il incombe au programmeur de placer les paramètres de la fonction sur la pile ou dans les registres appropriés avant l'appel, et de nettoyer la pile après celui-ci. C'est également au programmeur qu'il revient de sauvegarder les registres devant être préservés lors de l'appel, à moins que ces registres ne soient de type “callee-save” (sauvegardés par la fonction appelée).

En raison de ces complications, je vous recommande de sortir du bloc assembleur et d'utiliser la syntaxe C++ pour effectuer les appels de fonction.

Vue d'ensemble de la syntaxe

La syntaxe de l'assembleur en ligne de style MASM n'est correctement décrite dans aucun manuel de compilateur que j'ai consulté. Je vais donc en résumer ici les règles les plus importantes :

Dans la plupart des cas, l'assembleur en ligne de style MASM est interprété par le compilateur sans faire appel à un assembleur externe. Par conséquent, vous ne pouvez pas supposer que le compilateur acceptera tout ce que l'assembleur comprend.

Le code assembleur en ligne est délimité par le mot-clé `__asm`. Certains compilateurs autorisent l'alias `_asm`. Le code assembleur doit être placé entre accolades {}, sauf s'il ne comporte qu'une seule ligne. Les instructions assembleur sont séparées par des sauts de ligne. Il est également possible de séparer les instructions assembleur par le mot-clé `__asm`, sans utiliser de point-virgule.

Les instructions et les étiquettes s'écrivent de la même manière qu'en MASM. La taille des opérandes mémoire peut être spécifiée à l'aide de l'opérateur PTR ; par exemple : `INC DWORD PTR [ESI]`. Les noms d'instructions et de registres ne sont pas sensibles à la casse.

Les variables, fonctions et étiquettes goto déclarées en C++ sont accessibles sous les mêmes noms dans le code assembleur en ligne. Ces noms sont sensibles à la casse.

L'accès aux membres de données des structures, classes et unions s'effectue par rapport à un registre de pointeur, en utilisant l'opérateur point (.).

Les commentaires commencent par un point-virgule (;) ou une double barre oblique (//).

Les opcodes codés en dur sont définis à l'aide de `_emit` suivi d'une constante d'octet, là où l'on utiliserait `DB` dans MASM. Par exemple, `_emit 0x90` correspond à `NOP`.

Les directives et les macros ne sont pas autorisées.

Le compilateur gère les conventions d'appel et la sauvegarde des registres pour la fonction contenant l'assembleur en ligne, mais pas pour les appels de fonction effectués depuis ce code assembleur.

Compilateurs utilisant un assembleur en ligne de style MASM

L'assemblage en ligne de style MASM est pris en charge par les compilateurs Microsoft C++ 16 bits et 32 bits, mais le compilateur Microsoft 64 bits n'a pas de fonctionnalité d'assemblage en ligne intégrée.

Le compilateur Intel C++ prend en charge l'assemblage en ligne de style MASM sous Windows 32 bits et 64 bits ainsi que sous Linux 32 bits et 64 bits (et Mac OS ?). Le compilateur Intel sous Linux nécessite l'option de ligne de commande `-use-msasm` pour reconnaître cette syntaxe pour l'assemblage en ligne. Seul le mot-clé `__asm` fonctionne dans ce cas, pas les synonymes `asm` ou `__asm__`. Le compilateur Intel convertit la syntaxe MASM en syntaxe AT&T avant de l'envoyer à l'assembleur. Le compilateur Intel peut donc être utile comme convertisseur de syntaxe.

6.2 Assemblage en ligne de style GNU

L'assembleur en ligne fonctionne de manière très différente avec les compilateurs GNU et Clang, car ces derniers passent par une étape d'assemblage plutôt que de produire directement du code objet. Le code assembleur est saisi sous forme de chaîne de caractères constante, transmise à l'assembleur avec très peu de modifications. La syntaxe par défaut est la syntaxe AT&T, utilisée par l'assembleur GNU.

L'assembleur en ligne de style GNU présente l'avantage de permettre la transmission de n'importe quelle instruction ou directive à l'assembleur ; tout est possible. En revanche, sa syntaxe est très complexe et difficile à maîtriser, comme l'illustrent les exemples ci-dessous.

L'assembleur en ligne de style GNU est pris en charge par les compilateurs GNU, Clang et Intel (pour Linux), aussi bien en mode 32 bits qu'en mode 64 bits.

Syntaxe AT&T

Revenons à l'exemple 6.1b et traduisons-le en assembleur en ligne de style GNU avec la syntaxe AT&T :

```
// Exemple 6.1e. Assembleur en ligne de style GNU, syntaxe AT&T
double ipow (double x, int n) {
    double y;
    __asm__ (
        "cld \n"                // cdq
        "xorl %%edx, %%eax \n"
        "subl %%edx, %%eax \n"
        "fldl \n"
        "jz 9f \n"                // Saut vers le 9: le plus proche
        "fldl %[xx] \n"          // Remplacé par x
        "jmp 2f \n"              // Saut vers le 2: le plus proche
        "1: \n"                  // Étiquette locale 1:
        "fmul %%st(0), %%st(0) \n"
        "2: \n"                  // Étiquette locale 1:
        "shrl $1, %%eax \n"
        "jnc 1b \n"              // Saut en arrière vers le 1: le plus proche
        "fmul %%st(0), %%st(1) \n"
        "jnz 1b \n"              // Saut en arrière vers le 1: le plus proche
        "fstp %%st(0) \n"
        "testl %%edx, %%edx \n"
        "jns 9f \n"              // Saut vers le 9: le plus proche
        "fldl \n"
        "fdivp %%st(0), %%st(1) \n"
        "9: \n"                  // La chaîne d'assemblage se termine ici
        // Utilisation de la syntaxe d'assembleur étendu pour spécifier les opérandes :
        // Opérandes de sortie :
        : "=t" (y)              // Affichage du sommet de la pile dans y
    );
}
```

```

// Opérandes d'entrée :
: [xx] "m" (x), "a" (n) // Opérandes d'entrée %[xx] = x, eax = n
// Registres écrasés :
: "%edx", "%st(1)" // Écrasement de edx et st(1)
); // L'instruction __asm__ se termine ici.
return y;
}

```

On remarque immédiatement que la syntaxe diffère grandement des exemples précédents. De nombreux codes d'instruction comportent des suffixes précisant la taille de l'opérande. Les instructions sur les entiers utilisent `b` pour `BYTE`, `w` pour `WORD`, `d` pour `DWORD` et `q` pour `QWORD`. Les instructions en virgule flottante utilisent `s` pour `DWORD` (float), `l` pour `QWORD` (double) et `t` pour `TBYTE` (long double). Certains codes d'instruction sont totalement différents ; par exemple, `CDQ` devient `CLTD`. L'ordre des opérandes est l'inverse de celui de la syntaxe MASM : l'opérande source précède l'opérande de destination. Les noms de registres sont précédés du préfixe `%%`, qui devient `%` avant que la chaîne ne soit transmise à l'assembleur. Les constantes sont précédées du préfixe `$`. La notation des opérandes mémoire diffère également ; par exemple, `[ebx+ecx*4+20h]` devient `0x20(%%ebx, %%ecx, 4)`.

Les étiquettes de saut peuvent être définies comme dans MASM (par exemple `L1:`, `L2:`), mais j'ai opté pour la syntaxe des étiquettes locales, consistant en un nombre décimal suivi d'un deux-points. Les instructions de saut peuvent alors utiliser `jmp 1b` pour revenir en arrière vers l'étiquette 1 : la plus proche, ou `jmp 1f` pour avancer vers l'étiquette 1 : suivante la plus proche. J'ai choisi ce type d'étiquettes car le compilateur génère plusieurs instances du code assembleur “inline” si la fonction est intégrée directement au code appelant (inlining), ce qui est très probable. Si l'on utilisait des étiquettes classiques comme `L1:`, le fichier assembleur final contiendrait plusieurs étiquettes portant le même nom, ce qui provoquerait inévitablement une erreur. Pour utiliser des étiquettes classiques, il faut ajouter `__attribute__((noinline))` à la déclaration de la fonction afin d'empêcher son intégration directe.

L'assembleur en ligne de style GNU ne permet pas d'insérer directement les noms de variables locales C++ dans le code assembleur. Seuls les noms de variables globales fonctionnent, car ils sont identiques dans le code assembleur. Vous pouvez en revanche utiliser la syntaxe dite “étendue”, comme illustré ci-dessus. Cette syntaxe étendue se présente comme suit :

```

__asm__ ("assembly code string" : [output list] : [input list] :
[clobbered registers list] );

```

La chaîne de code assembleur est une constante de type chaîne contenant des instructions assembleur séparées par des caractères de saut de ligne (`\n`).

Dans l'exemple ci-dessus, la liste de sortie est `"=t" (y)`. “`t`” désigne le registre à virgule flottante situé au sommet de la pile, `st(0)`, et “`y`” indique que la valeur doit être stockée dans la variable C++ nommée `y` après l'exécution du code assembleur.

La liste d'entrée contient deux opérandes. `[xx] "m" (x)` signifie qu'il faut remplacer `%[xx]` dans la chaîne assembleur par un opérande mémoire correspondant à la variable C++ `x`. “`a`” (`n`) signifie qu'il faut charger la variable C++ `n` dans le registre `eax` avant l'exécution du code assembleur. Il existe de nombreux symboles de contrainte pour spécifier différents types d'opérandes et de registres en entrée et en sortie ; consultez le manuel de GCC pour plus de détails.

La liste des registres modifiés (ou “*clobbered registers*”) “`%edx`”, “`%st(1)`” indique que les registres `edx` et `st(1)` sont altérés par le code assembleur en ligne. Si ces registres ne figuraient pas dans cette liste, le compilateur supposerait à tort qu'ils sont restés inchangés.

Syntaxe Intel

L'exemple ci-dessus sera un peu plus facile à coder si nous utilisons la syntaxe Intel pour la chaîne d'assemblage. L'assembleur GNU accepte la syntaxe Intel grâce à la directive `.intel_syntax noprefix`. L'option `noprefix` signifie que les registres ne nécessitent pas de signe “`%`” comme préfixe.

```

// Exemple 6.1f. Assembleur en ligne de style GNU, syntaxe Intel
double ipow (double x, int n) {
    double y;
    __asm__ (
        ".intel_syntax noprefix \n" // utilisation de la syntaxe Intel par commodité
        "cdq \n"
        "xor eax, edx \n"
        "sub eax, edx \n"
        "fildl \n"
        "jz 9f \n"
        ".att_syntax prefix \n" // Syntaxe AT&T requise pour %[xx]
        "fildl %[xx] \n" // opérande mémoire remplacé par x
    );
}

```

```

        ".intel_syntax noprefix \n" // basculer à nouveau sur la syntaxe Intel
        "jmp 2f \n"
        "1: \n"
        "fmul st(0), st(0) \n"
        "2: \n"
        "shr eax, 1 \n"
        "jnc 1b \n"
        "fmul st(1), st(0) \n"
        "jnz 1b \n"
        "fstp st(0) \n"
        "test edx, edx \n"
        "jns 9f \n"
        "fldl \n"
        "fdivrp \n"
        "9: \n"
        ".att_syntax prefix \n" // retour à la syntaxe AT&T
        // output operands:
        : "=t" (y) // la sortie en haut de la pile est dirigée vers y
        // input operands:
        : [xx] "m" (x), "a" (n) // entrée mémoire %[x] pour x, eax pour n
        // clobbered registers:
        : "%edx", "%st(1)"; // edx et st(1) sont modifiés
    return y;
}

```

Ici, j'ai inséré `.intel_syntax noprefix` au début de la chaîne d'assembleur, ce qui me permet d'utiliser la syntaxe Intel pour les instructions. La chaîne doit se terminer par `.att_syntax prefix` pour revenir à la syntaxe AT&T par défaut, car cette dernière est utilisée dans le code généré ultérieurement par le compilateur. L'instruction qui charge l'opérande mémoire `x` doit utiliser la syntaxe AT&T, car le mécanisme de substitution d'opérande emploie cette syntaxe pour l'opérande remplaçant `%[xx]`. L'instruction `fldl %[xx]` doit donc être écrite en syntaxe AT&T. Il reste possible d'utiliser des étiquettes locales de style AT&T. Les listes d'opérands de sortie, d'opérands d'entrée et de registres modifiés sont identiques à celles de l'exemple 6.1e.

7 Utilisation d'un assembleur

L'utilisation de fonctions intrinsèques et de l'assembleur en ligne comporte certaines limitations. Ces limitations peuvent être surmontées en faisant appel à un assembleur. Le principe consiste à écrire un ou plusieurs fichiers assembleur contenant les fonctions les plus critiques du programme, tout en rédigeant les parties moins critiques en C++. Les différents modules sont ensuite liés pour former un fichier exécutable.

Les avantages de l'utilisation d'un assembleur sont les suivants :

- Il n'y a pratiquement aucune limite à ce que vous pouvez faire.
- Vous maîtrisez entièrement tous les détails du code exécutable final.
- Tous les aspects du code peuvent être optimisés, notamment le prologue et l'épilogue des fonctions, les méthodes de passage de paramètres, l'utilisation des registres, l'alignement des données, etc.
- Il est possible de créer des fonctions comportant plusieurs points d'entrée.
- Il est possible de produire du code compatible avec plusieurs compilateurs et systèmes d'exploitation (voir page 39).
- Les bons assembleurs disposent de fonctionnalités de macro-instructions puissantes, ouvrant la voie à des possibilités de méta-programmation absentes de la plupart des langages compilés de haut niveau (voir page 82).

Les inconvénients de l'utilisation d'un assembleur sont :

- Le langage assembleur est difficile à apprendre. Il y a de nombreuses instructions à mémoriser.
- Coder en assembleur prend plus de temps que de coder dans un langage de haut niveau.
- La syntaxe du langage assembleur n'est pas normalisée.
- Le code assembleur a tendance à devenir mal structuré et à ressembler à du "code spaghetti". Il faut beaucoup de discipline pour produire un code assembleur bien structuré et lisible par les autres.
- Le code assembleur n'est pas portable d'une architecture de microprocesseur à une autre.
- Le programmeur doit connaître tous les détails des conventions d'appel et les respecter dans le code.

- L'assembleur effectue très peu de vérifications syntaxiques. De nombreuses erreurs de programmation ne sont pas détectées par l'assembleur.
- Il existe de nombreuses façons de commettre des erreurs en assembleur, et ces erreurs peuvent avoir de graves conséquences.
- Il est difficile de se rappeler quelles instructions appartiennent à quelles extensions du jeu d'instructions. utilisation d'une instruction inappropriée peut provoquer le plantage du programme.
- On peut mélanger par inadvertance des instructions vectorielles VEX et non-VEX. Cela entraîne une perte de performance importante sur certains processeurs Intel (voir [paragraphe 13.1](#)).
- Les erreurs dans le code assembleur peuvent être difficiles à localiser. Par exemple, l'oubli de sauvegarde d'un registre peut provoquer un dysfonctionnement dans une partie totalement différente du programme.
- Le langage assembleur ne convient pas à la réalisation d'un programme complet. La majeure partie du programme doit être écrite dans un autre langage de programmation.

La meilleure façon de procéder pour écrire du code assembleur est de commencer par réaliser l'intégralité du programme en C ou en C++. Optimisez le programme en appliquant les méthodes décrites dans le manuel 1 : "Optimizing software in C++". Si une partie du programme nécessite une optimisation supplémentaire, isolez-la dans un module distinct. Traduisez ensuite ce module critique du C++ vers l'assembleur. Il n'est pas nécessaire d'effectuer cette traduction manuellement ; la plupart des compilateurs C++ peuvent générer du code assembleur. Activez toutes les options d'optimisation pertinentes du compilateur lors de la conversion du code C++ en assembleur. Le code assembleur produit par le compilateur constitue un excellent point de départ pour une optimisation ultérieure. Le code généré par le compilateur respecte nécessairement les conventions d'appel appropriées. (Notez toutefois que le code produit par les compilateurs 64 bits pour Windows peut ne pas être entièrement compatible avec tous les assembleurs).

Examinez le code assembleur produit par le compilateur pour identifier d'éventuelles possibilités d'optimisation supplémentaire. Il arrive que les compilateurs soient très performants et produisent un code mieux optimisé que celui qu'un programmeur assembleur moyen pourrait écrire. À l'inverse, ils peuvent parfois se montrer incroyablement inefficaces, en effectuant certaines opérations de manière maladroite. C'est dans ce dernier cas qu'il est justifié de consacrer du temps à la programmation en assembleur.

La plupart des environnements de développement intégrés (IDE) permettent d'inclure des modules en langage assembleur dans un projet C++. Par exemple, dans Microsoft Visual Studio, vous pouvez définir une "étape de génération personnalisée" pour un fichier source assembleur. La spécification de cette étape peut, par exemple, se présenter comme suit : Ligne de commande : `ml /c /Cx /Zi /coff $(InputName).asm`. Sorties : `$(InputName).obj`. Vous pouvez également utiliser un fichier make (voir page 38) ou un fichier batch.

Les fichiers C++ qui appellent les fonctions du module assembleur doivent inclure un fichier d'en-tête (*.h) contenant les prototypes de ces fonctions. Il est recommandé d'ajouter `extern "C"` aux prototypes afin d'éliminer les codes de "décoration de noms" (*name mangling*) spécifiques au compilateur.

Des exemples de fonctions écrites en assembleur pour différentes plateformes sont fournis au paragraphe 4.5, à partir de la page 23.

7.1 Bibliothèques de liens statiques

Il est pratique de collecter le code assembleur à partir de plusieurs fichiers assembleur dans une bibliothèque de fonctions. Les avantages de l'organisation des modules d'assemblage en bibliothèques de fonctions sont les suivants :

- La bibliothèque peut contenir de nombreuses fonctions et de nombreux modules. L'éditeur de liens sélectionnera automatiquement les modules nécessaires à un projet donné et ignorera les autres, évitant ainsi l'ajout de code superflu au projet.
- Il est simple et pratique d'intégrer une bibliothèque de fonctions dans un projet C++. Tous les compilateurs et environnements de développement (IDE) C++ prennent en charge les bibliothèques de fonctions.
- Une bibliothèque de fonctions est réutilisable. Le temps supplémentaire consacré au codage et au test d'une fonction en langage assembleur se justifie davantage si le code peut être réutilisé dans différents projets.
- La création d'une bibliothèque de fonctions réutilisables vous oblige à produire un code rigoureusement testé et documenté, doté d'une fonctionnalité et d'une interface avec le programme appelant clairement définies.
- Une bibliothèque de fonctions réutilisables offrant une fonctionnalité générale est plus facile à maintenir et à vérifier qu'un code assembleur spécifique à une application, dont le rôle est moins clairement défini.
- Une bibliothèque de fonctions peut être utilisée par d'autres programmeurs ne possédant aucune connaissance du langage assembleur.

Une bibliothèque de fonctions à liaison statique pour Windows est créée en utilisant le gestionnaire de bibliothèques (par ex. `lib.exe`) pour combiner un ou plusieurs fichiers `*.obj` en un fichier `*.lib`.

Une bibliothèque de fonctions à liaison statique pour Linux est créée en utilisant le gestionnaire d'archives (`ar`) pour combiner un ou plusieurs fichiers `*.o` en un fichier `*.a`.

Une bibliothèque de fonctions doit être accompagnée d'un fichier d'en-tête (`*.h`) contenant les prototypes des fonctions qu'elle renferme. Ce fichier d'en-tête est inclus dans les fichiers C++ qui appellent les fonctions de la bibliothèque (par ex. `#include "mylibrary.h"`).

Il est pratique d'utiliser un fichier `make` (voir page 38) pour gérer les commandes nécessaires à la création et à la mise à jour d'une bibliothèque de fonctions.

7.2 Bibliothèques de liens dynamiques

La différence entre l'édition de liens statique et dynamique réside dans le fait que la bibliothèque statique est intégrée au fichier exécutable, de sorte que ce dernier ne contient qu'une copie des parties nécessaires de la bibliothèque. Une bibliothèque à liaison dynamique (`*.dll`) est distribuée sous forme de fichier distinct, chargé lors de l'exécution par le fichier exécutable dans les systèmes Windows. Linux et Mac OS utilisent un type de bibliothèque dynamique légèrement différent, appelé « objet partagé » (`*.so`). Ces derniers sont décrits dans la section suivante.

Les avantages des bibliothèques de liens dynamiques sont :

- Une seule instance de la bibliothèque de liens dynamiques est chargée en mémoire lorsque plusieurs programmes s'exécutant simultanément utilisent la même bibliothèque.
- La bibliothèque de liens dynamiques peut être mise à jour sans modifier le fichier exécutable.
- Une bibliothèque de liens dynamiques peut être appelée depuis la plupart des langages de programmation, tels que Pascal, C# et Visual Basic. Un appel depuis Java est possible mais difficile.

Les inconvénients des bibliothèques de liens dynamiques sont :

- La bibliothèque entière est chargée en mémoire, même si seule une petite partie est nécessaire.
- Le chargement d'une bibliothèque à liaison dynamique (DLL) entraîne un délai supplémentaire lors du chargement du fichier exécutable.
- L'appel d'une fonction dans une bibliothèque à liaison dynamique est moins efficace que dans une bibliothèque statique, en raison de la surcharge liée à l'appel et d'une utilisation moins optimale du cache de code.
- La bibliothèque à liaison dynamique doit être distribuée conjointement avec le fichier exécutable.
- Plusieurs programmes installés sur le même ordinateur doivent utiliser la même version d'une bibliothèque à liaison dynamique si des versions différentes portent le même nom. Cela peut engendrer de nombreux problèmes de compatibilité.

Une DLL pour Windows est générée à l'aide de l'éditeur de liens Microsoft (`link.exe`). Il est nécessaire de fournir à l'éditeur de liens un ou plusieurs fichiers `.obj` ou `.lib` contenant les fonctions de bibliothèque requises ainsi qu'une fonction `DllEntry`, laquelle se contente de renvoyer la valeur 1. Un fichier de définition de module (`*.def`) est également requis. À noter que, sous Windows 32 bits, les fonctions de DLL utilisent la convention d'appel `__stdcall`, tandis que les fonctions des bibliothèques statiques utilisent par défaut la convention `__cdecl`. Un exemple de code source est disponible à l'adresse www.agner.org/random/randoma.zip.

7.3 Bibliothèques d'objets partagés

Linux, BSD et Mac OS utilisent des objets partagés (`*.so`) plutôt que des DLL. Les objets partagés sont très similaires aux DLL et utilisés de la même manière.

Il est possible de remplacer un nom de symbole dans un objet partagé comme dans une bibliothèque statique. Cela le rend différent d'une DLL. Cette fonctionnalité, appelée interposition de symboles, est rarement utilisée et son coût est élevé. Toutes les fonctions avec visibilité publique sont accessibles via une table de liaison de procédures (PLT), et toutes les variables et objets de données avec visibilité publique sont accessibles via une table de décalage globale (GOT). Les variables statiques avec uniquement une visibilité locale sont également accessibles via le GOT en mode 32 bits. Ces complications rendent les objets partagés moins efficaces que les bibliothèques de liens statiques. Voir la section « Code indépendant de la position » dans le manuel 1 : « Optimisation du logiciel en C++ » pour une discussion plus approfondie.

7.4 Bibliothèques sous forme de code source

L'un des problèmes liés aux bibliothèques de sous-programmes fournies sous forme binaire est que le compilateur ne peut pas optimiser l'appel de fonction. Ce problème peut être résolu en fournissant les fonctions de la bibliothèque sous forme de code source C++.

Lorsque les fonctions de la bibliothèque sont fournies sous forme de code source C++, le compilateur peut éliminer la surcharge liée à l'appel de fonction en procédant à l'intégration en ligne (*inlining*) de la fonction. Il peut optimiser l'allocation des registres à travers la fonction, effectuer une propagation de constantes, déplacer du code invariant lorsque la fonction est appelée au sein d'une boucle, etc.

Le compilateur ne peut réaliser ces optimisations qu'avec du code source C++, et non avec du code purement assembleur. Le code peut contenir de l'assembleur en ligne ou des appels à des fonctions intrinsèques. Le compilateur peut effectuer davantage d'optimisations si le code utilise des fonctions intrinsèques, mais moins s'il fait appel à de l'assembleur en ligne. L'efficacité de l'optimisation du code variera selon le compilateur utilisé.

Si le compilateur prend en charge l'optimisation à l'échelle du programme entier (*whole program optimization*), les fonctions de la bibliothèque peuvent simplement être fournies dans un fichier source C++. Dans le cas contraire, le code de la bibliothèque doit être inclus à l'aide de directives `#include` afin de permettre l'optimisation au-delà des appels de fonction. Une fonction définie dans un fichier inclus doit être déclarée `static` et/ou `inline` pour éviter les conflits entre plusieurs instances de cette fonction.

Certains compilateurs dotés de fonctionnalités d'optimisation à l'échelle du programme entier peuvent générer des fichiers objets partiellement compilés, permettant une optimisation supplémentaire lors de l'étape d'édition de liens. Malheureusement, le format de ces fichiers n'est pas normalisé, pas même entre différentes versions d'un même compilateur. Il est possible que les futures technologies de compilation permettent l'adoption d'un format standardisé pour le code partiellement compilé. Ce format devrait, au minimum, spécifier quels registres sont utilisés pour le passage des paramètres et lesquels sont modifiés par chaque fonction. Idéalement, il devrait également permettre l'allocation des registres lors de l'édition de liens, la propagation des constantes, l'élimination des sous-expressions communes entre les fonctions ainsi que le déplacement de code invariant.

Tant que de telles possibilités ne seront pas disponibles, on peut envisager une stratégie alternative consistant à placer la totalité de la boucle la plus interne dans une fonction de bibliothèque optimisée, plutôt que d'appeler cette fonction depuis une boucle C++. Cette solution est utilisée dans la bibliothèque Math Kernel Library d'Intel. Si, par exemple, vous devez calculer un millier de logarithmes, vous pouvez transmettre un tableau de mille arguments à une fonction de calcul vectoriel de logarithmes de la bibliothèque et récupérer un tableau de mille résultats en retour. Cette approche présente toutefois l'inconvénient de nécessiter le stockage des résultats intermédiaires en mémoire vive plutôt que leur transfert via des registres.

7.5 Création de classes en langage assembleur

En langage assembleur, les classes sont codées sous forme de structures, et les fonctions membres sont codées comme des fonctions recevant en paramètre un pointeur vers la classe ou la structure.

Il n'est pas possible d'appliquer la déclaration `extern "C"` à une fonction membre en C++, car `extern "C"` fait référence aux conventions d'appel du langage C, lequel ne comporte ni classes ni fonctions membres. La solution la plus logique consiste à utiliser le nom de fonction "décoré" (ou *mangled*). En reprenant les exemples 6.2a et b de la page 30, nous pouvons écrire la fonction membre `int MyList::Sum()` avec un nom décoré de la manière suivante :

```
; Exemple 7.1a (Exemple 6.2b traduit intégralement en langage assembleur)
; Fonction membre, Windows 32 bits

section .text
global ?Sum@MyList@@QAEHXZ:function

; Définition d'une structure correspondant à la classe MyList :
struc MyList
length: resd 1           ; int length
buffer: resd 100         ; int buffer[100];
endstruc

; int MyList::Sum()
; Nom de fonction décoré compatible avec le compilateur Microsoft (32 bits) :
?Sum@MyList@@QAEHXZ:
; Sous Windows 32 bits, le pointeur "this" se trouve dans le registre ECX.
xor eax, eax             ; sum = 0
xor edx, edx             ; Index de boucle i = 0
cmp dword [ecx+length], 0 ; this->length
je L9                   ; Ignorer si length = 0
L1: add eax, [ecx+buffer+edx*4] ; sum += buffer[i]
add edx, 1               ; i++
cmp edx, [ecx+length]    ; while (i < length)
jb L1                   ; Boucle
L9: ret                  ; La valeur de retour se trouve dans EAX.
```

Le nom de fonction mutilé `?Sum@MyList@@QAEHXZ` est spécifique au compilateur. Les compilateurs Gnu et Clang ont d'autres codes de modification de noms. De plus, d'autres compilateurs peuvent mettre 'this' sur la pile plutôt que dans un registre. Ces incompatibilités peuvent être résolues en utilisant une fonction friend plutôt qu'une fonction membre. Cela résout le problème selon lequel une fonction membre ne peut pas être déclarée `extern "C"`. La déclaration dans le fichier d'en-tête C++ doit alors être modifiée comme suit :

```
// Exemple 7.1b. Fonction membre transformée en fonction friend :
// Une déclaration de classe incomplète est nécessaire ici :
class MyList;

// Prototype de fonction pour une fonction friend avec "this" comme paramètre :
extern "C" int MyList_Sum(MyList * ThisP);

// Déclaration de classe:
class MyList {
protected:
    int length;           // Data members:
    int buffer[100];
public:
    MyList();             // Constructeur
    void AttItem(int item); // Ajout de l'élément à la liste
    // Faire MyList_Sum un friend:
    friend int MyList_Sum(MyList * ThisP);
    // Traduire Sum en MyList_Sum via un call inline :
    int Sum() {return MyList_Sum(this);}
};
```

Le prototype de la fonction friend doit précéder la déclaration de la classe, car certains compilateurs n'autorisent pas `extern "C"` à l'intérieur de celle-ci. Une déclaration incomplète de la classe est nécessaire, car la fonction friend requiert un pointeur vers la classe.

Les déclarations ci-dessus amèneront le compilateur à remplacer tout appel à `MyList::Sum` par un appel à `MyList_Sum`, car cette dernière fonction est incluse (*inlinée*) dans la première. L'implémentation en assembleur de `MyList_Sum` ne nécessite pas de nom décoré. Le pointeur se trouvera sur la pile en mode 32 bits, et dans le registre `rcx` ou `rdi` en mode 64 bits.

7.6 Fonctions thread-safe

Une fonction “thread-safe” (sûre pour le multithreading) ou réentrante est une fonction qui fonctionne correctement lorsqu'elle est appelée simultanément par plusieurs threads. Le multithreading permet de tirer parti des ordinateurs dotés de plusieurs cœurs de processeur. Il est légitime d'exiger qu'une bibliothèque de fonctions destinée à des applications critiques en termes de performance soit “thread-safe”.

Les fonctions sont “thread-safe” lorsqu'aucune variable n'est partagée entre les threads, sauf dans le cadre d'une communication délibérée entre eux. Les données constantes peuvent être partagées entre les threads sans problème. Les variables stockées sur la pile sont “thread-safe” car chaque thread possède sa propre pile. Le problème ne survient qu'avec les variables statiques stockées dans un segment de données. Les variables statiques sont utilisées lorsque des données doivent être conservées d'un appel de fonction à l'autre. Il est possible de créer des variables statiques locales à un thread, mais cette approche est inefficace et dépend du système.

La meilleure façon de conserver des données d'un appel de fonction à l'autre tout en garantissant la sécurité des threads consiste à laisser la fonction appelante allouer l'espace de stockage nécessaire. L'approche la plus élégante consiste à encapsuler les données et les fonctions qui les utilisent au sein d'une classe. Chaque thread doit alors créer une instance de cette classe et appeler ses fonctions membres. Le paragraphe précédent explique comment implémenter des fonctions membres en langage assembleur.

Si la fonction assembleur sécurisée pour les threads doit être appelée depuis le langage C ou un autre langage ne prenant pas en charge les classes (ou le faisant de manière incompatible), la solution consiste à allouer un tampon de stockage dans chaque thread et à transmettre un pointeur vers ce tampon à la fonction.

7.7 Makefiles

L'utilitaire “make” est un outil universel de gestion de projets logiciels. Il assure le suivi de l'ensemble des fichiers sources, fichiers objets, bibliothèques, fichiers exécutables, etc., d'un projet logiciel. Il procède ainsi en s'appuyant sur un ensemble de règles générales fondées sur les horodatages des fichiers. Si un fichier source est plus récent que le fichier objet correspondant, ce dernier doit être régénéré ; de même, si le fichier objet est plus récent que le fichier exécutable, ce dernier doit être régénéré.

Tout environnement de développement intégré (IDE) inclut un utilitaire “make” activable depuis une interface graphique ; toutefois, dans la plupart des cas, il est également possible d'utiliser une version de cet utilitaire en

ligne de commande. L'utilitaire “make” en ligne de commande (appelé make ou nmake) repose sur un ensemble de règles que vous pouvez définir dans un fichier appelé “makefile”. L'avantage d'utiliser un makefile est qu'il permet de définir des règles pour tout type de fichier : fichiers sources (quel que soit le langage de programmation), fichiers objets, bibliothèques, fichiers de définition de module, fichiers de ressources, exécutables, archives ZIP, etc. La seule condition requise est l'existence d'un outil capable de convertir un type de fichier en un autre et pouvant être appelé en ligne de commande avec les noms de fichiers comme paramètres.

La syntaxe utilisée pour définir des règles dans un makefile est pratiquement identique pour tous les utilitaires “make” fournis avec les différents compilateurs pour Windows et Linux.

De nombreux IDE fournissent également des fonctionnalités pour les règles de création définies par l'utilisateur pour les types de fichiers inconnus de l'EDI, mais ces utilitaires sont souvent moins généraux et moins flexibles qu'un utilitaire de création autonome.

Ce qui suit est un exemple de makefile pour créer une bibliothèque de fonctions mylibrary.lib à partir de trois fichiers sources assembleur func1.asm, func2.asm, func3.asm et l'emballer avec le fichier d'en-tête correspondant mylibrary.h dans un fichier zip mylibrary.zip.

```
# Example 7.2. Makefile pour mylibrary, syntaxe NMAKE
mylibrary.zip: mylibrary.lib mylibrary.h
    zip $@ $?

mylibrary.lib: func1.obj func2.obj func3.obj
    lib /out:$@ $**

.asm.obj
    nasm -f win32 -o $@.obj $.asm
    # or MASM: ml /c /Cx /coff /Fo$@ $.asm
```

La ligne mylibrary.zip: mylibrary.lib mylibrary.h indique que le fichier mylibrary.zip est construit à partir de mylibrary.lib et mylibrary.h, et qu'il doit être reconstruit si l'un de ces fichiers a été modifié plus récemment que le fichier zip. La ligne suivante, qui doit être indentée par une tabulation, spécifie la commande nécessaire pour construire le fichier cible mylibrary.zip à partir de ses dépendances mylibrary.lib et mylibrary.h. Les deux lignes suivantes indiquent comment construire le fichier de bibliothèque mylibrary.lib à partir des trois fichiers objets. La ligne .asm.obj est une règle générique ; elle indique que tout fichier ayant l'extension .obj peut être construit à partir d'un fichier portant le même nom mais l'extension .asm, en utilisant la règle définie à la ligne indentée suivante.

Les règles de construction des makefiles peuvent utiliser les macros suivantes pour spécifier les noms de fichiers :

	make Gnu	nmake MS
Nom complet de la cible actuelle	\$@	\$@
Nom complet du fichier dépendant	\$<	\$<
Nom de base de la cible actuelle sans l'extension	\$*	\$*
Toutes les dépendances de la cible actuelle, séparées par des espaces	^ \$+	**
Toutes les dépendances dont l'horodatage est postérieur à celui de la cible actuelle	\$?	\$?

Tableau 7.1. Macros Makefile

L'utilitaire make est lancé à l'aide du batch

```
nmake /Fmakefile or
make -f makefile.
```

Consultez le manuel de l'utilitaire “make” spécifique pour plus de détails.

8 Rendre les bibliothèques de fonctions compatibles avec plusieurs compilateurs et plateformes

Il convient de résoudre un certain nombre de problèmes de compatibilité si l'on souhaite créer une bibliothèque de fonctions compatible avec plusieurs compilateurs, langages de programmation et systèmes d'exploitation. Les principaux problèmes de compatibilité concernent :

1. Décoration de noms
2. Conventions d'appel
3. Formats de fichiers objets

La solution la plus simple pour résoudre ces problèmes de portabilité consiste à écrire le code dans un langage de haut niveau tel que le C++ et à implémenter les constructions de bas niveau nécessaires à l'aide de fonctions intrinsèques ou d'assembleur en ligne. Le code peut ensuite être compilé à l'aide de différents compilateurs adap-

tés aux diverses plateformes. Les compilateurs C++ GNU et Clang permettent de générer du code pour la quasi-totalité des plateformes, en utilisant une syntaxe identique pour l'assembleur en ligne et les fonctions intrinsèques sur toutes les plateformes x86 ; toutefois, d'autres compilateurs peuvent différer à cet égard.

Si une programmation entièrement en langage assembleur s'avère nécessaire ou souhaitable, il existe diverses méthodes pour surmonter les problèmes de compatibilité entre les différentes plateformes x86. Ces méthodes sont abordées dans les paragraphes suivants.

8.1 Prise en charge de plusieurs schémas de transformation de noms

La manière la plus simple de gérer les problèmes liés aux mécanismes de décoration de noms (*name mangling*) spécifiques au compilateur consiste à désactiver ce mécanisme à l'aide de la directive `extern "C"`, comme expliqué à la page 23.

La directive `extern "C"` ne peut pas être utilisée pour les fonctions membres de classes, les fonctions surchargées ni les opérateurs. On peut contourner ce problème en créant une fonction inline (dont le nom sera décoré) qui appelle une fonction en assembleur (dont le nom ne sera pas décoré) :

```
// Exemple 8.1. Éviter la transformation des noms (name mangling) des fonctions surchargées
// en C++
// Prototypes pour les fonctions d'assemblage sans "mangling" :
extern "C" double power_d (double x, double n);
extern "C" double power_i (double x, int n);

// Encapsulage (Wrapping) dans des fonctions surchargées :
inline double power (double x, double n) {return power_d(x, n);}
inline double power (double x, int n) {return power_i(x, n);}
```

Le compilateur remplacera simplement un appel à la fonction mutilée par un appel à la fonction d'assemblage non décorée appropriée sans aucun code supplémentaire. La même méthode peut être utilisée pour les fonctions membres de classe, comme expliqué à la page 37.

Cependant, dans certains cas, il est souhaité de conserver le nom de mutilation. Soit parce que cela simplifie le code C++, soit parce que les noms mutilés contiennent des informations sur les conventions d'appel et d'autres problèmes de compatibilité.

Une fonction d'assemblage peut être rendue compatible avec plusieurs schémas de modification de noms simplement en lui attribuant plusieurs noms publics. En revenant à l'exemple 4.1b page 23, nous pouvons ajouter des noms mutilés pour plusieurs compilateurs de la manière suivante :

```
; Exemple 8.2. (Exemple 4.1b réécrit)
; Fonction avec plusieurs noms décorés (mode 32 bits)

; double sinxpnx (double x, int n) {return sin(x) + n*x;}
section .text

global _sinxpnx:function, ?sinxpnx@@YANNH@Z:function
global __Z7sinxpnxdi:function, __Z7sinxpnxdi:function

ALIGN 4
; Définissez des noms publics pour chaque schéma de transformation de noms :
_sinxpnx:                ; extern "C" name
?sinxpnx@@YANNH@Z:       ; Compilateur Microsoft
__Z7sinxpnxdi:           ; Compilateur Gnu pour Linux
__Z7sinxpnxdi:           ; Compilateur Gnu pour Windows et Mac OS

; paramètre x = [ESP+4]
; paramètre n = [ESP+12]
; valeur de retour = ST(0)

    fild dword [esp+12]    ; n
    fld qword [esp+4]      ; x
    fmul st1, st0          ; n*x
    fsin                  ; sin(x)
    fadd                  ; sin(x) + n*x
    ret
```

L'exemple 8.2 fonctionne avec la plupart des compilateurs, aussi bien sous Windows 32 bits que sous Linux 32 bits, car les conventions d'appel sont identiques. Une fonction peut posséder plusieurs noms publics, et l'éditeur de liens recherchera simplement le nom correspondant à l'appel effectué depuis le fichier C++. En revanche, un appel de fonction ne peut pas correspondre à plus d'un nom externe.

La syntaxe de codage des noms (*name mangling*) propre aux différents compilateurs est décrite dans le manuel n° 5 : “Calling conventions for different C++ compilers and operating systems” (Conventions d'appel pour différents compilateurs C++ et systèmes d'exploitation). Appliquer cette syntaxe manuellement est une tâche ardue. Il est bien plus simple et sûr de générer chaque nom codé en compilant la fonction en C++ avec le compilateur approprié pour obtenir un fichier source en assembleur. Une autre méthode consiste à compiler pour obtenir un fichier objet, puis à le désassembler. Les versions en ligne de commande de la plupart des compilateurs sont disponibles gratuitement ou sous forme de versions d'essai.

Le compilateur Intel pour Windows est compatible avec le schéma de codage des noms de Microsoft. Les compilateurs Intel et Clang pour Linux sont compatibles avec le schéma de codage des noms de GNU. Les compilateurs GNU et Clang pour Windows peuvent, quant à eux, être capables d'utiliser le schéma de codage des noms de Microsoft.

8.2 Prise en charge de plusieurs conventions d'appel en mode 32 bits

Sous Windows 32 bits, les fonctions membres n'utilisent pas toujours la même convention d'appel. Les compilateurs compatibles Microsoft utilisent la convention `__thiscall`, avec le pointeur `this` dans le registre `ecx`, tandis que les compilateurs Borland et GNU utilisent la convention `__cdecl`, avec `this` placé sur la pile. Une solution consiste à utiliser des fonctions amies (*friend*), comme expliqué à la page 37. Une autre possibilité est de créer une fonction comportant plusieurs points d'entrée. L'exemple suivant est une réécriture de l'exemple 7.1a (page 37.) incluant deux points d'entrée correspondant aux deux conventions d'appel différentes :

```
; Exemple 8.3a (Exemple 7.1a avec deux entrées)
; Fonction membre, mode 32 bits
; int MyList::Sum()

section .text

global _MyList_Sum:function, @MyList@Sum$qv:function
global _ZN6MyList3SumEv:function, __ZN6MyList3SumEv:function
global ?Sum@MyList@@QAEHXZ:function

; Définition de la structure correspondant à la classe MyList :
struc MyList
length: resd 1          ; int length
buffer: resd 100        ; int buffer[100];
endstruc

_MyList_Sum:            ; pour la fonction amie extern "C"

; Génération des noms décorés pour les compilateurs utilisant la convention __cdecl :
_ZN6MyList3SumEv:       ; comp Gnu. pour Linux
__ZN6MyList3SumEv:      ; comp Gnu. pour Windows et Mac OS
; Déplacement du pointeur "this" hors de la pile pour le registre ECX:
mov ecx, [esp+4]

; Génération des noms décorés pour les compilateurs utilisant la convention __thiscall :
?Sum@MyList@@QAEHXZ:    ; compilateur Microsoft
xor eax, eax             ; sum = 0
xor edx, edx             ; Index de boucle i = 0
cmp dword [ecx+length], 0 ; this->length
je L9                   ; Ignorer si length = 0
L1: add eax, [ecx+buffer+edx*4] ; sum += buffer[i]
add edx, 1              ; i++
cmp edx, [ecx+length]   ; while (i < length)
jb L1                   ; Boucle
L9: ret                 ; la valeur de retour est dans EAX
```

La différence entre les mécanismes de décoration de noms (*name mangling*) constitue ici un avantage, car elle permet à l'éditeur de liens d'orienter l'appel vers le point d'entrée correspondant à la convention d'appel appropriée.

La méthode se complexifie si la fonction membre comporte davantage de paramètres. Prenons l'exemple de la fonction `void MyList::AttItem(int item)` figurant à la page 30. La convention `__thiscall` place le paramètre `this` dans le registre `ecx` et le paramètre `item` sur la pile à l'adresse `[esp+4]`, tout en exigeant que la fonction elle-même nettoie la pile. À l'inverse, la convention `__cdecl` place les deux paramètres sur la pile — `this` à `[esp+4]` et `item` à `[esp+8]` — et confie le nettoyage de la pile à l'appelant. Une solution impliquant deux points d'entrée nécessite un saut :

```
; Exemple 8.3b
```

```

; void MyList::AttItem(int item);

section .text

global _MyList_AttItem:function, _ZN6MyList7AttItemEi:function
global __ZN6MyList7AttItemEi:function,
?AttItem@MyList@@QAEXH@Z:function

; Définition de la structure correspondant à la classe MyList :
struc MyList
length: resd 1                ; int length
buffer: resd 100              ; int buffer[100];
endstruc

; Entrées de fonction
_MyList_AttItem:              ; pour une fonction amie extern "C"

; Génération des noms décorés pour les compilateurs utilisant la convention __cdecl :
_ZN6MyList7AttItemEi:         ; comp Gnu. pour Linux
__ZN6MyList7AttItemEi:        ; comp Gnu. pour Windows et Mac OS

    ; Déplacement des paramètres dans les registres :
    mov ecx, [esp+4]           ; ecx = pointeur this
    mov edx, [esp+8]           ; edx = item

    jmp L0                    ; jump into common section

; Génération des noms décorés pour les compilateurs utilisant la convention __thiscall :
?AttItem@MyList@@QAEXH@Z:     ; Microsoft compiler
    ; __thiscall nécessite un nettoyage de la pile par la fonction :
    pop eax                   ; Retire l'adresse de retour de la pile
    pop edx                   ; Récupération du paramètre « item » de la pile
    push eax                  ; Remise de l'adresse de retour sur la pile.

L0: ; section commune où les paramètres se trouvent dans des registres
    ; ecx = this, edx = item

    mov eax, [ecx+length]      ; eax = this->length
    cmp eax, 100               ; Vérification si c'est trop élevé
    jnb L9                    ; La liste est complète. Quitter.
    mov [ecx+buffer+eax*4], edx ; buffer[length] = item
    add eax, 1                 ; length++
    mov [ecx+length], eax
L9: ret

```

Dans l'exemple 8.3b, les deux points d'entrée de la fonction chargent chacun tous les paramètres dans des registres, puis effectuent un saut vers une section commune qui ne nécessite pas de lire les paramètres depuis la pile. Le point d'entrée `__thiscall` doit retirer les paramètres de la pile avant d'atteindre la section commune, car la convention `__thiscall` exige que la fonction elle-même nettoie la pile.

Un autre problème de compatibilité survient lorsque l'on souhaite disposer à la fois d'une version à liaison statique et d'une version à liaison dynamique d'une même bibliothèque de fonctions sous Windows 32 bits. La bibliothèque à liaison statique utilise par défaut la convention `__cdecl`, tandis que la bibliothèque à liaison dynamique utilise par défaut la convention `__stdcall`. La bibliothèque à liaison statique constitue la solution la plus efficace pour les programmes C++, mais la bibliothèque à liaison dynamique est nécessaire pour plusieurs autres langages de programmation.

Une solution à ce problème consiste à spécifier la convention `__cdecl` ou `__stdcall` pour les deux bibliothèques. Une autre solution consiste à réaliser des fonctions à deux entrées.

L'exemple suivant montre la fonction de l'exemple 8.2 avec deux entrées pour les conventions d'appel `__cdecl` et `__stdcall`. Les deux conventions ont les paramètres sur la pile. La différence est que la pile est nettoyée par l'appelant dans la convention `__cdecl` et par la fonction appelée dans la convention `__stdcall`.

; **Exemple 8.4a** (Exemple 8.2 avec les entrées `__stdcall` et `__cdecl`)
; Fonction avec des entrées pour `__stdcall` et `__cdecl` (Windows 32 bits) :

```

section .text

global _sinxpnx@12:function, _sinxpnx:function

align    4

```

```

; __stdcall entry:
; extern "C" double __stdcall sinxpnx (double x, int n);
_sinxpnx@12:
    ; Chargement de tous les paramètres dans les registres
    fild dword [esp+12]    ; n
    fld qword [esp+4]     ; x

    ; Suppression des paramètres de la pile :
    pop eax               ; Pop de l'adresse de retour
    add esp, 12           ; suppression de 12 octets de paramètres
    push eax              ; Remise de l'adresse de retour sur la pile.
    jmp L0

; entrée __cdecl:
; extern "C" double __cdecl sinxpnx (double x, int n);
_sinxpnx:
    ; Chargement de tous les paramètres dans les registres
    fild dword [esp+12]    ; n
    fld qword [esp+4]     ; x
    ; Ne retirez pas les paramètres de la pile. C'est l'appelant qui s'en charge.
L0: ; Point d'entrée standard avec tous les paramètres dans des registres
    ; paramètre x = st(0)
    ; paramètre n = st(1)
    fmul st1, st0         ; n*x
    fsin                  ; sin(x)
    fadd                  ; sin(x) + n*x
    ret                   ; la valeur de retour est dans st(0)

```

La méthode consistant à retirer les paramètres de la pile dans le prologue de la fonction plutôt que dans l'épilogue est, il faut l'admettre, assez peu élégante. Une solution plus efficace consiste à utiliser l'assemblage conditionnel :

```

; Exemple 8.4b
; Fonction avec des versions pour __stdcall et __cdecl (Windows 32 bits)
section .text

; Choix du prologue de fonction en fonction de la convention d'appel :
#ifdef STDCALL           ; si STDCALL_ est définie
global _sinxpnx@12:
function _sinxpnx@12:
    ; extern "C" __stdcall nom de fonction
#else
global _sinxpnx:
function _sinxpnx:
    ; extern "C" __cdecl nom de fonction
#endif

; Corps de fonction commun aux deux conventions d'appel :
    fild dword [esp+12]    ; n
    fld qword [esp+4]     ; x
    fmul st1, st0         ; n*x
    fsin                  ; sin(x)
    fadd                  ; sin(x) + n*x

; Choix de l'épilogue de la fonction en fonction de la convention d'appel :
#ifdef STDCALL           ; si STDCALL est définie
    ret 12                ; Clean up stack if __stdcall
#else
    ret                   ; Ne pas nettoyer la pile si __cdecl
#endif

```

Cette solution nécessite de créer deux versions du fichier objet : l'une utilisant la convention d'appel `__cdecl` pour la bibliothèque de liaison statique, et l'autre utilisant la convention `__stdcall` pour la bibliothèque de liaison dynamique. Cette distinction s'effectue au niveau de la ligne de commande de l'assembleur. La version `__stdcall` est assemblée avec l'option `/DSTDCALL` sur la ligne de commande afin de définir la macro `STDCALL_`, laquelle est détectée par la directive conditionnelle `IFDEF`.

8.3 Prise en charge de plusieurs conventions d'appel en mode 64 bits

Les conventions d'appel sont mieux normalisées dans les systèmes 64 bits que dans les systèmes 32 bits. Malheureusement, ces deux ensembles de conventions d'appel diffèrent considérablement. Les différences les plus importantes sont :

- Les paramètres de fonction sont transmis dans des registres différents selon les deux systèmes.
- Les registres RSI, RDI et XMM6 à XMM15 doivent être préservés par la fonction appelée (*callee-save*) sous Windows 64 bits, mais pas sous Linux 64 bits.
- L'appelant doit réserver un “espace fantôme” (*shadow space*) de 32 octets sur la pile pour la fonction appelée sous Windows 64 bits, mais pas sous Linux.
- Une “zone rouge” (*red zone*) de 128 octets située sous le pointeur de pile est disponible pour le stockage sous Linux 64 bits, mais pas sous Windows.
- Le mécanisme Microsoft de transformation des noms (*name mangling*) est utilisé sous Windows 64 bits, tandis que le mécanisme GNU est utilisé sous Linux 64 bits.

Dans les deux systèmes, la pile est alignée sur 16 octets avant tout appel et c'est l'appelant qui se charge de la nettoyer.

Il est possible de concevoir des fonctions compatibles avec les deux systèmes, à condition de tenir compte de leurs différences. La fonction doit préserver les registres dont la sauvegarde incombe à la fonction appelée (*callee-saved*) sous Windows, ou bien les laisser intacts. Elle ne doit utiliser ni la “shadow space” ni la “red zone”, mais doit réserver une “shadow space” pour toute fonction qu'elle appelle. Enfin, si elle comporte des paramètres de type entier, elle nécessite deux points d'entrée pour gérer les différences concernant les registres utilisés pour le passage des paramètres.

Reprenons l'exemple 4.1 (page 23) et réalisons une implémentation fonctionnant aussi bien sous Windows 64 bits que sous Linux 64 bits.

```
; Exemple 8.5a (Exemples 4.1d/e combinés).
; Prise en charge de Windows 64 bits et de Linux 64 bits
; double sinxpnx (double x, int n) {return sin(x) + n * x;}

section .text
global _Z7sinxpnxdi, ?sinxpnx@@YANNH@Z
extern sin

align      8

; 64-bit Linux entry:
_Z7sinxpnxdi:                                ; Transformation de noms GNU

        ; Linux a n dans EDI, Windows a n dans EDX. Déplaçons-le :
        mov edx, edi

; Entrée Windows 64 bits :
?sinxpnx@@YANNH@Z:                            ; Transformation de noms Microsoft

        ; parameter x = xmm0
        ; parameter n = edx
        ; return value = xmm0

        push rbx                                ; RBX doit être sauvegardé
        sub rsp, 48                            ; espace pour x, espace d'ombre f. sin, alignement
        movapd [rsp+32], xmm0                 ; sauvegarde de x à travers l'appel à sin
        mov ebx, edx                          ; sauvegarde de n à travers l'appel à sin
        call sin                               ; xmm0 = sin(xmm0)
        cvtsi2sd xmm1, ebx                    ; conversion de n en double
        mulsd xmm1, [rsp+32]                  ; n * x
        addsd xmm0, xmm1                      ; sin(x) + n * x
        add rsp, 48                            ; restauration du pointeur de pile
        pop rbx                                ; restauration de RBX
        ret                                    ; la valeur de retour est dans xmm0
```

Nous n'utilisons pas ici la déclaration extern "C", car nous nous appuyons sur les différents mécanismes de décoration de noms (*name mangling*) pour distinguer les versions Windows et Linux. Ces deux entrées servent à gérer les différences dans la passation des paramètres. Si la déclaration de la fonction plaçait *n* avant *x* (par exemple : double sinxpnx (int n, double x);), la version Windows placerait *x* dans XMM1 et *n* dans ECX, tandis que la version Linux placerait toujours *x* dans XMM0 et *n* dans EDI.

La fonction sauvegarde x sur la pile pendant l'appel à `sin`, car il n'existe pas de registres XMM devant être préservés par la fonction appelée (*callee-save*) sous Linux 64 bits. Elle réserve également 32 octets de “shadow space” pour l'appel à `sin`, bien que cela ne soit pas nécessaire sous Linux.

8.4 Prise en charge des différents formats de fichiers objets

Un autre problème de compatibilité provient des différences de format des fichiers objets.

Les anciens compilateurs Borland, Digital Mars et Microsoft 16 bits utilisent le format OMF pour les fichiers objets. Les compilateurs modernes pour Windows 32 bits utilisent le format COFF, également appelé PE32. Sous Linux 32 bits, les compilateurs GNU et Intel privilégient le format ELF32. Les compilateurs GNU et Intel pour Mac OS X utilisent le format Mach-O (en versions 32 et 64 bits). Tous les compilateurs pour Windows 64 bits utilisent le format COFF/PE32+, tandis que ceux pour Linux 64 bits utilisent le format ELF64.

L'assembleur NASM peut générer des fichiers aux formats OMF, COFF/PE32, ELF32/64, COFF/PE32+ et MachO32/64. L'assembleur GNU (Gas) peut produire les formats ELF32/64 et MachO32/64. L'assembleur MASM peut générer des fichiers objets aux formats OMF, COFF/PE32 et COFF/PE32+, mais pas au format ELF.

Il est possible d'effectuer un développement multiplateforme si vous disposez d'un assembleur prenant en charge tous les formats de fichiers objets nécessaires ou d'un utilitaire de conversion de fichiers objets adapté. Cela s'avère utile pour créer des bibliothèques de fonctions fonctionnant sur plusieurs plateformes. Un convertisseur de fichiers objets et gestionnaire de bibliothèques multiplateformes nommé `objconv` est disponible sur le site www.agner.org/optimize.

L'utilitaire `objconv` permet de modifier les noms de fonctions dans le fichier objet tout en effectuant une conversion vers un format de fichier objet différent. Cela peut aider à résoudre les problèmes liés à des schémas d'encodage des noms (*name mangling*) incompatibles. Reprenons l'exemple 8.5 sans encodage des noms :

```
; Exemple 8.5b.
; Prise en charge des systèmes Windows et Unix 64 bits.
; double sinxpnx (double x, int n) {return sin(x) + n * x;}

section .text
global Unix_sinxpnx, Win_sinxpnx
extern sin

align      8

; Entrée Linux 64 bits :
Unix_sinxpnx:

    ; Linux a n dans EDI, Windows a n dans EDI. Déplaçons-le :
    mov     edx, edi

; Entrée Windows 64 bits :
Win_sinxpnx:
    ; paramètre x = xmm0
    ; paramètre n = edx
    ; valeur de retour = xmm0
    push    rbx                ; RBX doit être sauvegardé
    sub     rsp, 48             ; espace pour x, espace d'ombre f. sin, alignement
    movapd  [rsp+32], xmm0      ; sauvegarde de x à travers l'appel à sin
    mov     ebx, edx            ; sauvegarde de n à travers l'appel à sin
    call    sin                 ; xmm0 = sin(xmm0)
    cvtsi2sd xmm1, ebx          ; conversion de n en double
    mulsd   xmm1, [rsp+32]      ; n * x
    addsd   xmm0, xmm1          ; sin(x) + n * x
    add     rsp, 48             ; restauration du pointeur de pile
    pop     rbx                 ; restauration de RBX
    ret                                ; la valeur de retour est dans xmm0
```

Cette fonction peut désormais être assemblée et convertie en plusieurs formats de fichier à l'aide des commandes suivantes :

```
nasm -f win64 -o sinxpnx.obj sinxpnx.nasm
objconv -cof64 -np:Win_ sinxpnx.obj sinxpnx_win.obj
objconv -elf64 -np:Unix_ sinxpnx.obj sinxpnx_linux.o
objconv -mac64 -np:Unix:_ sinxpnx.obj sinxpnx_mac.o
```

La première ligne assemble le code à l'aide de l'assembleur NASM et produit un fichier objet au format COFF.

La deuxième ligne supprime le préfixe “Win_” au début des noms de fonctions dans le fichier objet. Le résultat est un fichier objet COFF pour Windows 64 bits, dans lequel le point d'entrée Windows de notre fonction est disponible sous la forme extern "C" double sinxpx(double x, int n). Le nom Unix_sinxpx correspondant au point d'entrée Unix demeure inchangé dans le fichier objet, mais n'est pas utilisé. La troisième ligne convertit le fichier au format ELF pour Linux et BSD 64 bits et supprime le préfixe “Unix_” au début des noms de fonctions dans le fichier objet. Ainsi, le point d'entrée Unix de la fonction devient sinxpx, tandis que le point d'entrée Windows inutilisé est Win_sinxpx. La quatrième ligne effectue la même opération pour le format de fichier Mach-O et ajoute un préfixe de soulignement (“underscore”) au nom de la fonction, comme l'exigent les compilateurs Mac.

Objconv permet également de créer et de convertir des fichiers de bibliothèque statique (*.lib, *.a). Cela rend possible la création d'une bibliothèque de fonctions multiplateforme à partir d'une seule plateforme source.

La bibliothèque de fonctions multiplateformes asmlib.zip, disponible sur le site www.agner.org/optimize/, constitue un exemple d'utilisation de cette méthode. asmlib.zip inclut un fichier makefile (voir page 38) permettant de générer plusieurs versions de la même bibliothèque grâce au convertisseur de fichiers objets objconv.

Pour plus de détails sur les formats de fichiers objets, on peut consulter l'ouvrage “Linkers and Loaders” de J. R. Levine (Morgan Kaufmann Publ., 2000).

8.5 Prise en charge d'autres langages de haut niveau

Si vous utilisez un langage de haut niveau autre que le C++ et que le manuel du compilateur ne précise pas comment effectuer une liaison avec du code assembleur, vérifiez s'il indique comment lier des modules C ou C++. Ces informations devraient vous permettre de déduire la procédure pour l'assembleur.

En règle générale, il est préférable d'utiliser des fonctions simples, sans décoration de noms (*name mangling*), compatibles avec les conventions extern "C" et __cdecl ou __stdcall du C++. Cette approche fonctionne avec la plupart des langages compilés. Notez que les tableaux et les chaînes de caractères sont généralement implémentés différemment selon les langages.

De nombreux langages de programmation modernes, tels que C# et Visual Basic.NET, ne permettent pas la liaison avec des bibliothèques statiques ; vous devez alors créer une bibliothèque à liaison dynamique (DLL). Delphi Pascal peut également rencontrer des difficultés lors de la liaison avec des fichiers objets ; l'utilisation d'une DLL est alors plus simple.

L'appel de code assembleur depuis Java est une opération assez complexe. Vous devez compiler le code sous forme de DLL ou d'objet partagé, puis utiliser l'interface JNI (Java Native Interface) ou JNA (Java Native Access).

9 Optimisation de la vitesse

9.1 Identification des parties les plus critiques de votre code.

L'optimisation logicielle ne se résume pas à peaufiner les instructions assembleur. De nombreuses applications modernes consacrent bien plus de temps au chargement de modules, de fichiers de ressources, de bases de données, de frameworks d'interface, etc., qu'à l'exécution des calculs pour lesquels le programme a été conçu. Optimiser le temps de calcul est inutile si le programme passe 99,9 % de son temps à effectuer des tâches autres que le calcul. Il est crucial d'identifier les opérations les plus chronophages avant d'entreprendre toute optimisation. Parfois, la solution consiste à passer de Java ou C# à C++, à utiliser un autre framework d'interface utilisateur, à réorganiser les entrées/sorties de fichiers, à mettre en cache les données réseau ou à éviter l'allocation dynamique de mémoire, plutôt que de recourir au langage assembleur. Pour plus de détails, consultez le manuel 1 : “Optimizing software in C++”.

Le recours au code assembleur pour optimiser un logiciel ne se justifie que pour des programmes sollicitant intensément le processeur, tels que le traitement du son et de l'image, le chiffrement, le tri, la compression de données et les calculs mathématiques complexes.

Dans les logiciels sollicitant fortement le processeur, on constate souvent que plus de 99 % du temps de calcul est consommé par la boucle la plus imbriquée. Il est donc nécessaire d'identifier la partie la plus critique du code si l'on souhaite en améliorer la vitesse d'exécution. Optimiser des sections moins critiques constituerait non seulement une perte de temps, mais rendrait également le code moins lisible, ainsi que plus difficile à déboguer et à maintenir. La plupart des compilateurs incluent un outil de profilage (ou *profiler*) capable d'identifier la partie du code la plus critique. Si vous ne disposez pas d'un tel outil et que la section critique n'est pas évidente, mettez en place des compteurs incrémentés à différents endroits du code pour déterminer quelle partie est la plus fréquemment exécutée. Utilisez les méthodes décrites à la page 124 pour mesurer la durée d'exécution de chaque partie du code.

Il est important d'examiner l'algorithme utilisé dans la partie critique du code pour voir s'il peut être amélioré. Souvent, le simple choix de l'algorithme optimal permet d'obtenir un gain de vitesse supérieur à celui de toute autre méthode d'optimisation.

9.2 Exécution dans le désordre

Tous les processeurs x86 modernes peuvent exécuter des instructions dans le désordre. Prenons cet exemple :

; **Exemple 9.1a.** Exécution dans le désordre

```
mov eax, [mem1]
imul eax, 6
mov [mem2], eax
mov ebx, [mem3]
add ebx, 2
mov [mem4], ebx
```

Ce fragment de code effectue deux opérations sans aucun lien entre elles : multiplier [mem1] par 6 et ajouter 2 à [mem3]. Si [mem1] ne se trouve pas dans le cache, le processeur doit attendre de nombreux cycles d'horloge pendant que cet opérande est récupéré depuis la mémoire principale. Entre-temps, le processeur cherche une autre tâche à accomplir. Il ne peut pas exécuter la deuxième instruction, `imul eax, 6`, car elle dépend du résultat de la première. En revanche, la quatrième instruction, `mov ebx, [mem3]`, est indépendante des instructions précédentes ; il est donc possible d'exécuter `mov ebx, [mem3]` et `add ebx, 2` pendant l'attente de [mem1]. Les processeurs disposent de nombreuses fonctionnalités pour permettre une exécution dans le désordre (*out-of-order execution*) efficace. La plus importante est, bien entendu, la capacité à détecter si une instruction dépend du résultat d'une instruction antérieure. Une autre fonctionnalité clé est le renommage de registres. Supposons que nous utilisions le même registre pour la multiplication et l'addition dans l'exemple 9.1a, faute de registres disponibles :

; **Exemple 9.1b.** Exécution dans le désordre avec renommage de registres

```
mov eax, [mem1]
imul eax, 6
mov [mem2], eax
mov eax, [mem3]
add eax, 2
mov [mem4], eax
```

L'exemple 9.1b fonctionnera exactement aussi vite que l'exemple 9.1a car le CPU est capable d'utiliser différents registres physiques pour le même registre logique `eax`. Cela fonctionne de manière très élégante. Le processeur attribue un nouveau registre physique pour contenir la valeur de `eax` à chaque fois que `eax` est écrit. Cela signifie que le code ci-dessus est modifié à l'intérieur du processeur en un code qui utilise quatre registres physiques différents pour `eax`. Le premier registre est utilisé pour la valeur chargée depuis [mem1]. Le deuxième registre est utilisé pour la sortie de l'instruction `imul`. Le troisième registre est utilisé pour la valeur chargée depuis [mem3]. Et le quatrième registre est utilisé pour la sortie de l'instruction `ajout`. L'utilisation de registres physiques différents pour le même registre logique permet au CPU de rendre les trois dernières instructions de l'exemple 9.1b indépendantes des trois premières instructions. Le nombre de registres physiques est différent selon les microprocesseurs. Les meilleurs microprocesseurs disposent aujourd'hui de plusieurs centaines de registres physiques. Cela permet de réorganiser les instructions à grande échelle.

Registres partiels

Certains processeurs peuvent traiter séparément différentes parties d'un registre, tandis que d'autres considèrent toujours le registre dans sa globalité. Si nous modifions l'exemple 9.1b de manière à ce que la seconde partie utilise des registres de 16 bits, nous sommes confrontés à un problème de fausse dépendance :

; **Exemple 9.1c.** Fausse dépendance de registre partiel

```
mov eax, [mem1] ; opérande mémoire 32 bits
imul eax, 6
mov [mem2], eax
mov ax, [mem3] ; opérande mémoire 16 bits
add ax, 2
mov [mem4], ax
```

Ici, l'instruction `mov ax, [mem3]` ne modifie que les 16 bits de poids faible du registre EAX, tandis que les 16 bits de poids fort conservent la valeur issue de l'instruction `imul`. Certains processeurs (qu'ils soient d'Intel, d'AMD ou de VIA) sont incapables de renommer un registre partiel. Par conséquent, l'instruction `mov ax, [mem3]` doit attendre l'achèvement de l'instruction `imul`, car elle doit combiner les 16 bits de poids faible provenant de [mem3] avec les 16 bits de poids fort issus de l'instruction `imul`.

D'autres processeurs sont capables de scinder le registre en plusieurs parties afin d'éviter cette fausse dépendance ; toutefois, cette approche présente un autre inconvénient si les deux parties doivent être réunies ultérieu-

rement. Supposons, par exemple, que le code de l'exemple 9.1c soit suivi d'une instruction `PUSH EAX`. Sur certains processeurs, cette instruction doit attendre que les deux parties d'EAX soient validées (*retired*) pour pouvoir les fusionner, ce qui entraîne un coût de 5 à 6 cycles d'horloge. D'autres processeurs généreront une micro-opération (μop) supplémentaire pour assembler les deux parties du registre.

Ces problèmes sont évités en remplaçant `mov ax, [mem3]` par `movzx eax, [mem3]`. Cette opération réinitialise les bits de poids fort du registre `eax` et supprime toute dépendance vis-à-vis d'une valeur antérieure de ce registre. En mode 64 bits, il suffit d'écrire dans le registre 32 bits, car cela réinitialise systématiquement la partie supérieure du registre 64 bits correspondant ; ainsi, `movzx eax, [mem3]` et `movzx rax, [mem3]` produisent exactement le même résultat. La version 32 bits de l'instruction est plus courte d'un octet que la version 64 bits. Il convient d'éviter l'utilisation des registres 8 bits de poids fort (AH, BH, CH, DH), car elle peut engendrer des dépendances artificielles et rendre le code moins efficace.

Le registre de flags peut poser des problèmes similaires avec les instructions qui modifient certains flags tout en laissant les autres inchangés. Par exemple, les instructions `INC` et `DEC` laissent le flag de retenue (*carry flag*) inchangé, mais modifient les flags de zéro et de signe.

Micro-opérations

Une autre caractéristique importante est la décomposition des instructions en micro-opérations (abrégées en μops ou $uops$). L'exemple suivant illustre l'avantage :

```
; Exemple 9.2. Découpage des instructions en micro-opérations
push eax
call SomeFunction
```

L'instruction `push eax` effectue deux opérations : elle soustrait 4 au pointeur de pile et enregistre la valeur de `eax` à l'adresse pointée par ce dernier. Supposons maintenant que `eax` soit le résultat d'un calcul long et complexe ; cela retarde l'exécution de l'instruction `push`. L'instruction `call` dépend de la valeur du pointeur de pile, laquelle est modifiée par l'instruction `push`. Si les instructions n'étaient pas décomposées en micro-opérations (μops), l'instruction `call` devrait attendre que l'instruction `push` soit entièrement terminée. Or, le processeur décompose l'instruction `push eax` en `sub esp, 4` suivie de `mov [esp], eax`. La micro-opération `sub esp, 4` pouvant être exécutée avant que `eax` ne soit disponible, l'instruction `call` n'attendra que la fin de `sub esp, 4`, et non celle de `mov [esp], eax`.

Unités d'exécution

L'exécution dans le désordre gagne encore en efficacité lorsque le processeur est capable d'effectuer plusieurs tâches simultanément. Les processeurs modernes peuvent exécuter jusqu'à quatre, six, voire huit instructions en même temps, à condition que les données d'entrée de chaque instruction soient indépendantes des résultats des autres instructions et qu'elles ne sollicitent pas les mêmes unités d'exécution au sein du processeur. Ils disposent d'au moins deux unités arithmétiques et logiques (ALU) pour les entiers, ce qui leur permet de réaliser deux additions d'entiers (ou plus) par cycle d'horloge. Ils intègrent également deux unités de calcul en virgule flottante (ou davantage), rendant possible l'exécution simultanée de plusieurs opérations de ce type. La présence de plusieurs unités de lecture mémoire et d'une ou deux unités d'écriture mémoire permet d'effectuer des opérations de lecture et d'écriture en mémoire de manière concomitante. Il est ainsi possible, par exemple, de lancer deux opérations sur des entiers, une opération en virgule flottante et une opération mémoire au cours d'un même cycle d'horloge. Le manuel 3, intitulé "The microarchitecture of Intel, AMD, and VIA CPUs", détaille le nombre d'unités d'exécution de chaque type présentes dans un microprocesseur donné. Le manuel 4, "Instruction tables", indique le débit de chaque instruction pour les différents microprocesseurs.

Instructions traitées en pipeline

Les opérations en virgule flottante nécessitent généralement plus d'un cycle d'horloge, mais elles sont souvent traitées en pipeline, de sorte qu'une nouvelle addition en virgule flottante peut, par exemple, débuter avant que la précédente ne soit terminée. Sur la plupart des processeurs, les instructions vectorielles utilisent les unités d'exécution en virgule flottante, même pour les instructions sur entiers. Les détails concernant les instructions pouvant être exécutées simultanément ou en pipeline, ainsi que le nombre de cycles d'horloge requis pour chaque instruction, sont spécifiques à chaque processeur. Ces détails sont exposés dans le manuel n° 3 (« The microarchitecture of Intel, AMD and VIA CPUs ») et le manuel n° 4 (« Instruction tables »).

Résumé

Les points les plus importants à connaître pour tirer le meilleur parti de l'exécution dans le désordre sont les suivants :

- Au minimum, les registres suivants peuvent être renommés : tous les registres d'usage général, le pointeur de pile, le registre de flags, les registres à virgule flottante et les registres vectoriels. Certains processeurs permettent également de renommer les registres de segment et le mot de contrôle de l'unité à virgule flottante.

- Évitez les fausses dépendances en écrivant dans un registre complet plutôt que dans une partie de registre.
- Les instructions INC et DEC sont inefficaces sur certains processeurs car elles ne modifient qu'une partie du registre de flags (à l'exclusion du flag de retenue). Utilisez plutôt ADD ou SUB pour éviter les fausses dépendances ou une fragmentation inefficace du registre de flags, surtout si ces instructions sont suivies d'une instruction qui lit les flags.
- Une chaîne d'instructions dans laquelle chaque instruction dépend de la précédente ne peut pas être exécutée dans le désordre. Évitez les longues chaînes de dépendance. (Voir page 50).
- Les opérandes en mémoire ne peuvent pas être renommés.
- Une lecture mémoire peut s'exécuter avant une écriture mémoire précédente à une adresse différente dans la plupart des cas. Tout registre de pointeur ou d'index doit être calculé le plus tôt possible afin que le processeur puisse vérifier que les adresses des opérandes mémoire sont différentes.
- Il est possible qu'une écriture en mémoire ne puisse pas s'exécuter avant une écriture précédente, mais les tampons d'écriture peuvent contenir un certain nombre d'écritures en attente, généralement quatre ou plus.
- Une lecture mémoire peut s'exécuter avant ou simultanément avec une autre lecture précédente sur la plupart des processeurs.
- Le CPU peut faire plus de choses simultanément si le code contient un bon mélange d'instructions de différentes catégories, telles que : instructions entières simples, instructions à virgule flottante ou vectorielles, lecture mémoire, écriture mémoire.

9.3 Récupération, décodage et validation des instructions

La récupération (*fetching*) des instructions peut constituer un goulot d'étranglement. De nombreux processeurs ne peuvent pas récupérer plus de 16 octets de code d'instruction par cycle d'horloge. Il peut être nécessaire de rendre les instructions les plus courtes possibles si cette limite s'avère critique. Une façon de raccourcir les instructions est de remplacer les opérandes mémoire par des pointeurs (voir chapitre 10 page 57). L'adresse des opérandes mémoire peut éventuellement être chargée dans des registres de pointeurs en dehors d'une boucle si la récupération des instructions constitue un goulot d'étranglement. De grandes constantes peuvent également être chargées dans des registres.

La récupération des instructions est retardée par les sauts sur la plupart des processeurs. Il est important de minimiser le nombre de sauts dans le code critique. Les branches qui ne sont pas prises et correctement prédites ne retardent pas la récupération des instructions. Il est donc avantageux d'organiser les branches *if-else* de manière à ce que la branche la plus couramment suivie soit celle où le saut conditionnel n'est pas effectué.

La plupart des processeurs récupèrent les instructions par blocs alignés de 16 ou 32 octets. Il peut être avantageux d'aligner sur 16 octets les points d'entrée des boucles critiques et des sous-programmes afin de minimiser le nombre de frontières de blocs de 16 octets dans le code. À défaut, veillez à ce qu'aucune frontière de bloc de 16 octets ne se trouve parmi les premières instructions suivant l'entrée d'une boucle critique ou d'un sous-programme.

Le décodage des instructions constitue souvent un goulot d'étranglement. L'organisation des instructions permettant un décodage optimal dépend du processeur. Sur les anciens processeurs AMD, il est préférable d'éviter les instructions générant plus de deux micro-opérations (μ ops).

Les instructions comportant plusieurs préfixes peuvent ralentir le décodage. Selon le processeur, il existe une limite au nombre de préfixes qu'une instruction peut comporter sans ralentir le décodage. Évitez les préfixes de taille d'adresse. Évitez les préfixes de taille d'opérande sur les instructions utilisant un opérande immédiat ; par exemple, il vaut mieux remplacer `MOV AX, 2` par `MOV EAX, 2`.

Les processeurs modernes disposent d'un cache de micro-opérations ou d'un petit tampon de boucle (*loopback buffer*) qui permet d'éliminer le goulot d'étranglement lié au décodage des instructions dans les boucles courtes. Maintenez la taille des boucles réduite afin qu'elles puissent tenir dans le cache de micro-opérations ou le tampon de boucle. Évitez le déroulage de boucle inutile.

Tous ces détails sont spécifiques au processeur. Consultez le manuel n° 3, "The microarchitecture of Intel, AMD and VIA CPUs", pour plus d'informations.

9.4 Latence et débit des instructions

La latence d'une instruction correspond au nombre de cycles d'horloge nécessaires entre le début de son exécution et la disponibilité du résultat. Le temps d'exécution d'une chaîne de dépendances est égal à la somme des latences de toutes les instructions qui la composent.

Le débit d'une instruction correspond au nombre maximal d'instructions de même type pouvant être exécutées par cycle d'horloge, à condition que ces instructions soient indépendantes. Je préfère indiquer l'inverse du débit, car cela facilite la comparaison entre latence et débit. Cette valeur représente le temps moyen écoulé entre le début de l'exécution d'une instruction et le moment où une autre instruction indépendante de même type peut commencer, ou encore le nombre de cycles d'horloge par instruction au sein d'une série d'instructions indépen-

dantes de même type. Par exemple, l'addition en virgule flottante sur un processeur Core 2 présente une latence de 3 cycles d'horloge et un inverse de débit d'un cycle par instruction. Cela signifie que le processeur consomme 3 cycles d'horloge par addition si chaque opération dépend du résultat de la précédente, mais seulement 1 cycle par addition si les opérations sont indépendantes.

Le manuel n° 4, intitulé “Instruction tables” (Tables d'instructions), contient des listes détaillées des latences et des débits pour la quasi-totalité des instructions de nombreux microprocesseurs Intel, AMD et VIA.

La liste suivante présente quelques valeurs typiques.

Instruction	Latence typique	Débit réciproque typique
Integer move	1	0.17-0.5
Integer addition	1	0.17-0.5
Integer Boolean	1	0.17-0.5
Integer shift	1	0.33-1
Integer multiplication	3-10	1
Integer division	12-80	4-40
Floating point addition	2-6	0.5-1
Floating point multiplication	3-8	0.5-2
Floating point division	10-45	3-45
Integer vector addition	1-2	0.25-2
Integer vector multiplication	3-7	0.5-2
Floating point vector addition	2-5	0.5-2
Floating point vector multiplication	3-7	0.5-2
Floating point vector division	10-60	4-60
Memory read (cached)	3-4	0.25-1
Memory write (cached)	3-4	0.5-1
Jump or call	0	0.5-2

Tableau 9.1. Latences et débits typiques des instructions

9.5 Rupture des chaînes de dépendance

Une chaîne de dépendances est une série d'instructions dans laquelle le résultat de chaque instruction sert de donnée d'entrée pour l'instruction suivante. Chaque instruction doit attendre que la précédente soit terminée. Cela empêche l'exécution de plusieurs instructions en parallèle ou dans le désordre.

Il est important d'éviter les longues chaînes de dépendances afin de tirer parti de l'exécution dans le désordre. Prenons l'exemple suivant en C++, qui calcule la somme de 100 nombres :

```
// Exemple 9.3a. Chaîne de dépendances inter-itérations
double list[100], sum = 0.;
for (int i = 0; i < 100; i++) sum += list[i];
```

Ce code effectue une centaine d'additions, chacune dépendant du résultat de la précédente. Il s'agit d'une chaîne de dépendances inter-itérations. Une telle chaîne peut être très longue et empêcher totalement l'exécution dans le désordre pendant une durée prolongée. Seul le calcul de *i* peut être effectué en parallèle des additions en virgule flottante.

En supposant que l'addition en virgule flottante présente une latence de 4 cycles et un débit inverse de 1 cycle, l'implémentation optimale utilisera quatre accumulateurs afin de maintenir en permanence quatre additions dans le pipeline de l'unité d'addition en virgule flottante. En C++, cela se présentera comme suit :

```
// Exemple 9.3b. Accumulateurs multiples
double list[100], sum1 = 0., sum2 = 0., sum3 = 0., sum4 = 0.;
for (int i = 0; i < 100; i += 4) {
    sum1 += list[i];
    sum2 += list[i+1];
    sum3 += list[i+2];
    sum4 += list[i+3];
}
sum1 = (sum1 + sum2) + (sum3 + sum4);
```

Nous avons ici quatre chaînes de dépendance s'exécutant en parallèle, chacune représentant un quart de la longueur de la chaîne d'origine. Le nombre optimal d'accumulateurs correspond à la latence de l'instruction (ici, une addition en virgule flottante) divisée par l'inverse du débit. L'optimisation des boucles est abordée plus en détail à la page 69.

Il n'est pas toujours possible d'atteindre le débit maximal théorique. Plus le nombre de chaînes de dépendance parallèles est élevé, plus il est difficile pour le processeur d'ordonnancer et de réorganiser les micro-opérations de manière optimale. Cette difficulté est accrue lorsque les chaînes de dépendance présentent des branchements ou sont entrelacées.

Les microprocesseurs modernes peuvent exécuter quatre instructions ou plus par cycle d'horloge. Plus le microprocesseur prend en charge un parallélisme au niveau des instructions élevé, plus il est important d'éviter les longues chaînes de dépendance.

Les chaînes de dépendance ne se limitent pas aux boucles ; elles apparaissent également dans le code linéaire. Il est possible de rompre ces chaînes. Par exemple, l'expression $y = a + b + c + d$ peut être réécrite sous la forme $y = (a + b) + (c + d)$, permettant ainsi de calculer simultanément les deux groupes entre parenthèses.

Il existe parfois plusieurs façons d'effectuer un même calcul, chacune impliquant une latence différente. On peut, par exemple, avoir le choix entre une instruction de branchement et une instruction de déplacement conditionnel (*conditional move*). Le branchement offre la latence la plus faible, mais le déplacement conditionnel évite le risque d'une mauvaise prédiction de branchement (voir page 51). Le choix de l'implémentation optimale dépend de la prévisibilité du branchement et de la longueur de la chaîne de dépendance.

Une méthode courante pour mettre un registre à zéro consiste à utiliser les instructions `XOR EAX, EAX` ou `SUB EAX, EAX`. Certains processeurs reconnaissent que ces instructions sont indépendantes de la valeur précédente du registre. Ainsi, toute instruction utilisant la nouvelle valeur du registre n'aura pas à attendre que la valeur antérieure à l'instruction `XOR` ou `SUB` soit disponible. Ces instructions sont utiles pour rompre une dépendance inutile. La plupart des processeurs modernes considèrent qu'une instruction `XOR` utilisant deux registres d'entrée identiques est indépendante de la valeur précédente, qu'il s'agisse de registres d'usage général (32 ou 64 bits) ou de registres vectoriels (128 bits ou plus).

Il ne faut pas tenter de rompre une dépendance en utilisant une partie 8 ou 16 bits d'un registre. Par exemple, l'instruction `XOR AX, AX` rompt la dépendance sur certains processeurs, mais pas sur tous. En revanche, `XOR EAX, EAX` suffit à rompre la dépendance sur RAX en mode 64 bits.

L'instruction `SBB EAX, EAX` dépend naturellement du flag de retenue (*carry flag*), même lorsqu'elle ne dépend pas de EAX.

Vous pouvez également utiliser ces instructions pour rompre des dépendances liées aux flags d'état (flags). Par exemple, les instructions de rotation présentent une fausse dépendance aux flags sur certains processeurs Intel. Il est possible de supprimer cette dépendance de la manière suivante :

```
; Exemple 9.4. Rupture de la dépendance aux flags
ror eax, 1
xor edx, edx ; Élimination de la dépendance artificielle vis-à-vis des flags.
ror ebx, 1
```

Vous ne pouvez pas utiliser l'instruction `CLC` pour rompre les dépendances vis-à-vis du flag de retenue (*carry flag*).

9.6 Jumps et calls

Les sauts, branchements, appels et retours n'augmentent pas nécessairement le temps d'exécution d'un code, car ils sont généralement exécutés en parallèle avec d'autres opérations. Il convient néanmoins de réduire au minimum le nombre de ces instructions dans les sections de code critiques, pour les raisons suivantes :

- La récupération du code après un saut inconditionnel ou un saut conditionnel effectué est généralement retardée de 1 à 3 cycles d'horloge, en fonction du microprocesseur. Le délai est pire si la cible est proche de la fin d'un bloc de récupération de code de 16 ou 32 octets (c'est-à-dire avant une adresse divisible par 16).
- Le cache de code devient fragmenté et moins efficace lors du saut entre des sous-programmes non contigus.
- Les microprocesseurs dotés d'un cache μ op ou d'un cache de trace sont susceptibles de stocker plusieurs instances du même code dans ce cache lorsque le code contient de nombreux sauts.
- Le tampon de cible de branchement (BTB) ne peut stocker qu'un nombre limité d'adresses de cible de saut. Un échec BTB coûtera plusieurs cycles d'horloge.
- Les sauts conditionnels sont prédits selon des mécanismes avancés de prédiction de branchement. Les erreurs de prédiction coûtent cher, comme expliqué ci-dessous.
- Sur la plupart des processeurs, les branches peuvent interférer les unes avec les autres dans la table historique globale des modèles de branchement et dans le registre de l'historique des branches. Une branche peut donc réduire le taux de prédiction des autres branches.
- Les retours sont prédits par l'utilisation d'un tampon de pile de retour, qui ne peut contenir qu'un nombre limité d'adresses de retour, généralement 16.

- Les sauts indirects et les appels indirects sont mal prédits sur les processeurs plus anciens.

Tous les processeurs modernes disposent d'un pipeline d'exécution comprenant des étapes telles que la prélecture des instructions (*prefetching*), le décodage, le renommage des registres, le réordonnancement et la planification des micro-opérations (μops), l'exécution, la validation (ou "retirement"), etc. Le nombre d'étapes du pipeline varie de 12 à 22, selon la microarchitecture spécifique. Lorsqu'une instruction de branchement pénètre dans le pipeline, le processeur ne sait pas avec certitude quelle instruction devra être chargée ensuite. Il faut attendre 12 à 22 cycles d'horloge supplémentaires pour que l'instruction de branchement soit exécutée et que l'on connaisse avec certitude la direction prise par le branchement. Cette incertitude risque d'interrompre le flux au sein du pipeline. Plutôt que d'attendre 12 cycles d'horloge ou plus pour obtenir une réponse, le processeur tente de deviner quelle direction prendra le branchement. Cette prédiction repose sur le comportement antérieur du branchement : si celui-ci a suivi la même voie lors des dernières occurrences, on prédit qu'il en ira de même cette fois-ci ; s'il a alterné régulièrement entre les deux voies, on prédit qu'il continuera à alterner.

Si la prédiction est correcte, le processeur gagne un temps précieux en chargeant la bonne branche dans le pipeline et en commençant à décoder et à exécuter de manière spéculative les instructions de cette branche. Si la prédiction est erronée, l'erreur est détectée après plusieurs cycles d'horloge et doit être corrigée en vidant le pipeline et en rejetant les résultats des exécutions spéculatives. Le coût d'une mauvaise prédiction de branchement varie de 12 à plus de 50 cycles d'horloge, selon la longueur du pipeline et d'autres caractéristiques de la microarchitecture. Ce coût est si élevé que des algorithmes très sophistiqués ont été mis en œuvre pour affiner la prédiction de branchement. Ces algorithmes sont expliqués en détail dans le manuel n° 3 : "La microarchitecture des processeurs Intel, AMD et VIA".

En général, vous pouvez supposer que les branchements sont correctement prédits la plupart du temps dans ces cas :

- • Si la branche emprunte toujours le même chemin.
- • Si la branche suit un schéma répétitif simple et se trouve à l'intérieur d'une boucle comportant peu ou pas d'autres branches.
- • Si la branche est corrélée à une branche précédente.
- • Si la branche constitue une boucle avec un nombre de répétitions constant et faible, et qu'il y a peu ou pas de sauts conditionnels à l'intérieur de la boucle.

Le pire des cas est celui d'un branchement qui prend l'une ou l'autre direction environ 50 % du temps, sans suivre de schéma régulier ni présenter de corrélation avec un branchement précédent. Un tel branchement sera mal prédit dans 50 % des cas. Cette situation est si coûteuse qu'il convient, si possible, de remplacer le branchement par des instructions de déplacement conditionnel ou par une consultation de table.

En règle générale, il est conseillé de minimiser le nombre de branchements mal prédits ainsi que le nombre de branchements à l'intérieur d'une boucle. Il peut être utile de scinder ou de dérouler une boucle si cela permet de réduire le nombre de branchements qu'elle contient.

Les sauts et appels indirects sont souvent mal prédits. Les processeurs plus anciens prédisent simplement que le saut ou l'appel indirect suivra la même direction que lors de sa dernière exécution. De nombreux processeurs récents sont capables de reconnaître des schémas répétitifs simples pour les sauts indirects.

Les retours de fonction sont prédits à l'aide d'une structure appelée "tampon de pile de retour" (*return stack buffer*) ; il s'agit d'une mémoire tampon de type premier entré, dernier sorti (LIFO) qui reflète les adresses de retour empilées. Un tampon de pile de retour comportant 16 entrées permet de prédire correctement tous les retours de sous-programmes jusqu'à un niveau d'imbrication de 16. Si le niveau d'imbrication des sous-programmes dépasse la taille du tampon, les échecs de prédiction surviendront aux niveaux d'imbrication les plus externes, et non aux niveaux internes, qui sont généralement plus critiques. Une taille de tampon de 8 entrées ou plus suffit donc dans la plupart des cas, sauf pour les fonctions récursives fortement imbriquées.

Le tampon de pile de retour (*return stack buffer*) provoquera une erreur en cas d'appel sans instruction de retour correspondante ou de retour sans appel préalable. Il est donc important de toujours faire correspondre les appels et les retours. Ne sortez pas d'une sous-routine par un autre moyen qu'une instruction RET, sauf dans le cas d'appels terminaux (*tail calls*), comme expliqué ci-dessous. De plus, n'utilisez pas l'instruction RET comme un saut indirect. Les appels lointains (*far calls*) doivent être associés à des retours lointains (*far returns*).

Faites en sorte que les sauts conditionnels ne soient le plus souvent pas effectués.

Sur la plupart des processeurs, l'efficacité et le débit sont meilleurs pour les branchements non pris que pour les branchements pris. Il est donc avantageux de placer le branchement le plus fréquent en premier :

; **Exemple 9.5.** Placez la branche la plus fréquente en premier.

```
Func1:
    cmp eax, 567
    je L1
    ; branche fréquente
```

```

    ret
L1:  ; branche rare
    ret

```

Appels terminaux

Il est possible de remplacer un CALL suivi d'un RET par un saut :

```

; Exemple 9.6a. Séquence CALL/RET (32 bits)
Func1:
    ...
    call Func2
    ret

```

Cela peut être modifié comme suit :

```

; Exemple 9.6b. CALL+RET remplacé par JMP
Func1:
    ...
    jmp Func2      ; Appel terminal

```

Cette modification n'entre pas en conflit avec le mécanisme de tampon de pile de retour car l'appel à Func1 correspond au retour de Func2. Dans les systèmes avec alignement de pile, il est nécessaire de restaurer le pointeur de pile avant le JMP :

```

; Exemple 9.7a. Séquence CALL/RET (Windows ou Linux 64 bits)
Func1:
    sub rsp, 8      ; Alignement de la pile par 16
    ...
    call Func2      ; Ce CALL peut être supprimé.
    add rsp, 8
    ret

```

Cela peut être modifié comme suit :

```

; Exemple 9.7b. Remplacement de CALL+RET par JMP avec alignement de la pile
Func1:
    sub rsp, 8
    ...
    add rsp, 8      ; Restauration du pointeur de pile avant le JMP
    jmp Func2

```

Élimination des sauts inconditionnels

Il est souvent possible d'éliminer un saut en copiant le code vers lequel il pointe. Le code ainsi copié peut généralement correspondre à l'épilogue d'une boucle ou d'une fonction. L'exemple suivant présente une fonction contenant une structure conditionnelle (if-else) au sein d'une boucle :

```

; Exemple 9.8a. Fonction avec JMP pouvant être éliminé
FuncA:
    push ebp
    mov  ebp, esp
    sub  esp, StackSpaceNeeded
    lea  edx, [EndOfSomeArray]
    xor  eax, eax

Loop1:                                ; la boucle commence ici
    cmp  [edx+eax*4], eax             ; if-else
    je   ElseBranch
    ...                               ; Première branche
    jmp  End_If
ElseBranch:
    ...                               ; Deuxième branche
End_If:
    add  eax, 1                       ; Boucle d'épilogue
    jnz  Loop1
    mov  esp, ebp                     ; Fonction épilogue
    pop  ebp
    ret

```

Le saut vers End_If peut être éliminé en dupliquant l'épilogue de la boucle :

; **Exemple 9.8b.** Épilogue de boucle copié pour éliminer le saut

```
FuncA:
    push ebp
    mov  ebp, esp
    sub  esp, StackSpaceNeeded
    lea  edx, [EndOfSomeArray]
    xor  eax, eax

Loop1:                                ; La boucle commence ici
    cmp  [edx+eax*4], eax             ; if-else
    je   ElseBranch
    ...                                ; Première branche
    add  eax, 1                       ; Épilogue de boucle pour la première branche
    jnz  Loop1
    jmp  AfterLoop

ElseBranch:
    ...                                ; Deuxième branche
    add  eax, 1                       ; Épilogue de boucle pour la seconde branche
    jnz  Loop1

AfterLoop:
    Mov  esp, ebp                    ; Fonction épilogue
    pop  ebp
    ret
```

Dans l'exemple 9.8b, le saut inconditionnel à l'intérieur de la boucle a été éliminé en dupliquant l'épilogue de la boucle. La branche la plus fréquemment exécutée doit être placée en premier, car c'est la plus rapide. Le saut inconditionnel vers AfterLoop peut également être supprimé ; pour ce faire, on duplique l'épilogue de la fonction :

; **Exemple 9.8b.** L'épilogue de la fonction a été copié pour éliminer le JMP.

```
FuncA:
    push ebp
    mov  ebp, esp
    sub  esp, StackSpaceNeeded
    lea  edx, [EndOfSomeArray]
    xor  eax, eax

Loop1:                                ; La boucle commence ici
    cmp  [edx+eax*4], eax             ; if-else
    je   ElseBranch
    ...                                ; Première branche
    add  eax, 1                       ; Épilogue de boucle pour la première branche
    jnz  Loop1
    mov  esp, ebp                    ; Fonction épilogue 1
    pop  ebp
    ret

ElseBranch:
    ...                                ; Deuxième branche
    add  eax, 1                       ; Épilogue de boucle pour la seconde branche
    jnz  Loop1

    mov  esp, ebp                    ; Fonction épilogue 2
    pop  ebp
    ret
```

Le gain obtenu en supprimant le saut vers AfterLoop est inférieur à celui obtenu en supprimant le saut vers End_If, car ce dernier se situe en dehors de la boucle. Toutefois, je l'ai présenté ici pour illustrer la méthode générale de duplication de l'épilogue d'une fonction.

Remplacement des sauts conditionnels par des déplacements conditionnels (CMOV)

Les sauts les plus importants à éliminer sont les sauts conditionnels, surtout s'ils sont mal prédits. Exemple :

```
// Exemple 9.9a. Branche C++ à optimiser
a = b > c ? d : e;
```

Cela peut être mis en œuvre soit par un saut conditionnel, soit par un déplacement conditionnel :

; **Exemple 9.9b.** Branchement implémenté par un saut conditionnel (CMOV)

```
mov  eax, [b]
cmp  eax, [c]
jng  L1
mov  eax, [d]
jmp  L2
L1:  mov  eax, [e]
L2:  mov  [a], eax
```

; **Exemple 9.9c.** Branchement implémenté à l'aide d'un déplacement conditionnel (CMOV)

```
mov  eax, [b]
cmp  eax, [c]
mov  eax, [d]
cmovng eax, [e]
mov  [a], eax
```

L'avantage d'une instruction de déplacement conditionnel (*conditional move*) est qu'elle évite les erreurs de prédiction de branchement. Toutefois, elle présente l'inconvénient d'allonger la chaîne de dépendance, alors qu'un branchement prédit rompt cette chaîne. Si le code de l'exemple 9.9c s'inscrit dans une chaîne de dépendance, l'instruction `cmov` accroît la longueur de ladite chaîne. Si ce même code est implémenté à l'aide d'un saut conditionnel (comme dans l'exemple 9.9b) et que le branchement est correctement prédit, le résultat n'a pas besoin d'attendre que `b` et `c` soient prêts ; il lui suffit d'attendre soit `d`, soit `e`, selon l'opérande sélectionné. Cela signifie que la chaîne de dépendance est rompue par le branchement prédit, tandis que l'implémentation par déplacement conditionnel impose d'attendre la disponibilité de `b`, `c`, `d` et `e`. Si `d` et `e` correspondent à des expressions complexes, les deux doivent être calculées en cas d'utilisation du déplacement conditionnel, alors qu'une seule d'entre elles doit l'être si l'on opte pour un saut conditionnel.

En règle générale, on peut affirmer qu'un saut conditionnel est plus rapide qu'un déplacement conditionnel si le code fait partie d'une chaîne de dépendance et que le taux de prédiction dépasse 75 %. Le saut conditionnel est également préférable lorsqu'il permet d'éviter un calcul long de `d` ou de `e` alors que l'autre opérande est celui qui est retenu.

Les chaînes de dépendance portées par une boucle sont particulièrement sensibles aux inconvénients des instructions de déplacement conditionnel (*conditional moves*). Par exemple, le code de l'exemple 12.14a (page 83) s'exécute plus efficacement avec un branchement à l'intérieur de la boucle qu'avec un déplacement conditionnel, même si le branchement est mal prédit. Cela s'explique par le fait que le déplacement conditionnel en virgule flottante allonge la chaîne de dépendance portée par la boucle et que l'implémentation utilisant cette instruction nécessite de calculer toutes les valeurs `power*xp`, même lorsqu'elles ne sont pas utilisées.

La recherche dichotomique dans une liste triée constitue un autre exemple de chaîne de dépendance portée par une boucle. Si les éléments recherchés sont répartis de manière aléatoire sur l'ensemble de la liste, le taux de prédiction des branchements avoisinera les 50 % et l'utilisation de déplacements conditionnels sera plus rapide. En revanche, si les éléments sont souvent proches les uns des autres — ce qui améliore le taux de prédiction —, il est plus efficace d'utiliser des sauts conditionnels plutôt que des déplacements conditionnels, car la chaîne de dépendance est rompue à chaque prédiction de branchement correcte.

Il est également possible d'effectuer des transferts conditionnels dans des registres vectoriels, élément par élément. Voir la page 90 et les suivantes pour plus de détails. Il existe des instructions vectorielles spécifiques pour déterminer le minimum ou le maximum de deux nombres. L'utilisation de registres vectoriels peut s'avérer plus rapide que celle de registres d'entiers ou de nombres à virgule flottante pour rechercher des minimums ou des maximums.

Remplacement des sauts conditionnels par des instructions SET conditionnel

Si un saut conditionnel est utilisé pour attribuer la valeur 0 ou 1 à une variable booléenne, il est souvent plus efficace d'utiliser l'instruction de positionnement (SET) conditionnel. Exemple :

// **Exemple 9.10a.** Définition (SET) d'une variable booléenne selon une certaine condition.

```
int b, c;
bool a = b > c;
```

; **Exemple 9.10b.** Mise en œuvre avec SET conditionnel

```
mov  eax, [b]
cmp  eax, [c]
setg al
mov  [a], al
```

L'instruction de définition conditionnelle (SET) écrit uniquement dans des registres de 8 bits. Si un résultat sur 32 bits est nécessaire, mettez à zéro le reste du registre avant la comparaison :

```

; Exemple 9.10c. Implémentation avec SET conditionnel, 32 bits
mov  eax, [b]
xor  ebx, ebx      ; Mettre le registre à zéro avant l'instruction CMP pour éviter
                   ; de modifier les flags.

cmp  eax, [c]
setg bl
mov  [a], ebx

```

S'il n'y a pas de registre libre, utilisez alors `movzx` :

```

; Exemple 9.10d. Implémentation avec SET conditionnel, 32 bits
mov  eax, [b]
cmp  eax, [c]
setg al
movzx eax, al
mov  [a], eax

```

Si une valeur composée uniquement de bits à 1 est requise pour représenter VRAI, utilisez `neg eax`.

Une implémentation utilisant des sauts conditionnels peut s'avérer plus rapide que des SET conditionnels si le taux de prédiction est bon et que le code s'inscrit dans une longue chaîne de dépendances, comme expliqué dans la section précédente (page 54).

Remplacement des sauts conditionnels par des instructions de manipulation de bits

Il est parfois possible d'obtenir le même effet qu'une instruction de branchement grâce à une manipulation ingénieuse de bits et de flags. Le flag de retenue est particulièrement utile pour ce genre d'astuces de manipulation de bits :

```

; Exemple 9.11. Positionnement du flag de retenue si EAX est nul:
cmp  eax, 1

; Exemple 9.12. Activation du flag de retenue si EAX n'est pas nul :
neg  eax

; Exemple 9.13. Incrémentation de EAX si le flag de retenue est positionné:
adc  eax, 0

; Exemple 9.14. Copie du flag de retenue dans tous les bits de EAX :
sbb  eax, eax

; Exemple 9.15. Copie des bits un par un depuis la retenue vers un vecteur de bits :
rcl  eax, 1

```

Il est possible de calculer la valeur absolue d'un entier signé sans branchement :

```

; Exemple 9.16. Calcul de la valeur absolue de EAX
cdq      ; Copie du bit de signe de EAX dans tous les bits de EDX.
xor  eax, edx  ; Inversion de tous les bits si la valeur est négative
sub  eax, edx  ; Add 1 si négatif

```

L'exemple suivant identifie le minimum de deux nombres non signés : if (b > a) b = a;

```

; Exemple 9.17a. Recherche du minimum de EAX et EBX (non signé) :
sub  eax, ebx  ; = a-b
sbb  edx, edx  ; = (b > a) ? 0xFFFFFFFF : 0
and  edx, eax  ; = (b > a) ? a-b : 0
add  ebx, edx  ; le résultat est dans EBX

```

Ou, pour les nombres signés, en ignorant le dépassement de capacité (overflow) :

```

; Exemple 9.17b. Identification du minimum de EAX et EBX (signés) :
sub  eax, ebx  ; Ne fonctionnera pas en cas de débordement ici.
cdq      ; = (b > a) ? 0xFFFFFFFF : 0
and  edx, eax  ; = (b > a) ? a-b : 0
add  ebx, edx  ; le résultat est dans EBX

```

L'exemple suivant choisit entre deux nombres : if (a < 0) d = b; else d = c;

```

; Exemple 9.18a. Choisissez entre deux nombres.
test eax, eax
mov  edx, ecx
cmovs edx, ebx ; = (a < 0) ? b : c

```

Les instructions de déplacement conditionnel ne sont pas très efficaces sur les anciens processeurs Intel. D'autres implémentations peuvent s'avérer plus rapides dans certains cas. L'exemple suivant produit le même résultat que l'exemple 9.18a :

```
; Exemple 9.18b. Choisir entre deux nombres sans instruction de saut conditionnel :
cdq                ; = (a < 0) ? 0xFFFFFFFF : 0
xor ebx, ecx       ; b ^ c = bits qui diffèrent entre b et c
and edx, ebx       ; = (a < 0) ? (b ^ c) : 0
xor edx, ecx       ; = (a < 0) ? b : c
```

L'exemple 9.18b peut être plus rapide que le 9.18a sur les processeurs où les déplacements conditionnels sont inefficaces. L'exemple 9.18b détruit la valeur de ebx.

La question de savoir si ces astuces sont plus rapides qu'un saut conditionnel dépend du taux de prédiction, comme expliqué ci-dessus.

10 Optimisation de la taille

Le cache d'instructions peut contenir entre 8 et 64 Ko de code, comme expliqué au chapitre 11 (page 64). Si vous rencontrez des difficultés pour maintenir la boucle interne critique du code dans le cache d'instructions, le cache de micro-opérations (μ op cache) ou le tampon de boucle (*loopback buffer*), vous pouvez envisager de réduire la taille du code. Réduire la taille du code peut également améliorer le décodage des instructions. Les boucles très courtes sont particulièrement rapides car elles peuvent tenir dans un tampon de boucle, ou parce que le décodeur peut réutiliser des lignes du cache d'instructions déjà chargées.

Vous pouvez même choisir de réduire la taille du code au prix d'une vitesse moindre, si la performance n'est pas une priorité.

Le code 64 bits ne nécessite pas davantage d'octets pour les adresses que le code 32 bits, car il peut utiliser des adresses 32 bits relatives au registre RIP. Le code 64 bits peut être légèrement plus volumineux que le code 32 bits en raison des préfixes REX et d'autres différences mineures ; toutefois, il peut aussi être plus compact, car le nombre accru de registres réduit le recours aux variables en mémoire.

10.1 Choix d'instructions plus courtes

Certaines instructions possèdent des formes courtes. Les instructions PUSH et POP utilisant un registre d'entiers n'occupent qu'un seul octet. L'instruction XCHG EAX, reg32 tient également sur un octet et occupe donc moins d'espace qu'une instruction MOV, bien que XCHG soit plus lente que MOV. Il en va de même pour INC et DEC avec un registre 32 bits en mode 32 bits. La forme courte de INC et DEC n'est pas disponible en mode 64 bits.

Les instructions suivantes occupent un octet de moins lorsqu'elles utilisent l'accumulateur plutôt qu'un autre registre : ADD, ADC, SUB, SBB, AND, OR, XOR, CMP et TEST avec un opérande immédiat sans extension de signe. Cela s'applique aussi à l'instruction MOV avec un opérande mémoire et sans registre de pointeur en modes 16 et 32 bits, mais pas en mode 64 bits. Exemples :

```
; Exemple 10.1. Tailles des instructions
add eax, 1000      est plus petit que add ebx, 1000
mov  eax, [mem]    est plus petit que mov  ebx, [mem], sauf en mode 64 bits.
```

Les instructions utilisant des pointeurs occupent un octet de moins lorsqu'elles comportent uniquement un pointeur de base (autre que ESP, RSP ou R12) et un déplacement, par rapport aux cas où elles utilisent un registre d'index avec facteur d'échelle, à la fois un pointeur de base et un registre d'index, ou ESP, RSP ou R12 comme pointeur de base. Exemples :

```
; Exemple 10.2. Tailles des instructions
mov  eax, array[ebx] est plus petit que mov  eax, array[ebx*4]
mov  eax, [ebp+12]   est plus petit que mov  eax, [esp+12]
```

Les instructions utilisant EBP, RBP ou R13 comme pointeur de base, sans déplacement ni index, occupent un octet de plus que celles utilisant d'autres registres :

```
; Exemple 10.3. Tailles des instructions
mov  eax, [ebx]      est plus petit que mov  eax, [ebp], mais
mov  eax, [ebx+4]    est de la même taille que mov  eax, [ebp+4].
```

Les instructions avec un pointeur d'index mis à l'échelle et sans pointeur de base doivent avoir un déplacement de quatre octets, même lorsqu'il vaut 0 :

```
; Exemple 10.4. Tailles des instructions
lea  eax, [ebx+ebx]  est plus petit que lea  eax, [ebx*2].
```

En mode 64 bits, les instructions nécessitent un préfixe REX si au moins l'un des registres R8 à R15 ou XMM8 à XMM15 est utilisé. Les instructions utilisant ces registres sont donc plus longues d'un octet que celles qui utilisent d'autres registres, à moins qu'un préfixe REX ne soit déjà requis pour d'autres raisons :

; **Exemple 10.5a.** Tailles des instructions (mode 64 bits)

```
mov  eax, [rbx]  est plus petit que mov  eax, [r8].
```

; **Exemple 10.5b.** Tailles des instructions (mode 64 bits)

```
mov  rax, [rbx]  est de la même taille que mov  rax, [r8].
```

Dans l'exemple 10.5a ci-dessus, on peut éviter un préfixe REX en utilisant le registre RBX comme pointeur au lieu de R8. En revanche, dans l'exemple 10.5b, un préfixe REX est nécessaire de toute façon pour la taille d'opérande de 64 bits, et l'instruction ne peut pas comporter plus d'un préfixe REX.

Les calculs en virgule flottante peuvent être effectués soit à l'aide des anciennes instructions de type x87 utilisant les registres de pile ST(0)-à ST(7), soit à l'aide des instructions plus récentes de type SSE utilisant les registres XMM. Les instructions de type x87 sont plus compactes que ces dernières ; par exemple :

; **Exemple 10.6.** Tailles des instructions en virgule flottante

```
fadd st0, st1      ; 2 octets
```

```
addsd xmm0, xmm1   ; 4 octets
```

L'utilisation de code de type x87 peut s'avérer avantageuse, même si elle nécessite des instructions FXCH supplémentaires. Sur les processeurs actuels, la différence de vitesse d'exécution entre les deux types d'instructions en virgule flottante est minime. Toutefois, il est probable que les instructions de type x87 soient considérées, à l'avenir, comme obsolètes et deviennent moins efficaces sur les futurs processeurs.

Les processeurs prenant en charge le jeu d'instructions AVX permettent d'encoder les instructions XMM de deux manières différentes : soit avec un préfixe VEX, soit avec les anciens préfixes. Selon les cas, c'est soit la version VEX, soit l'ancienne version qui s'avère la plus courte. Cependant, sur la plupart des processeurs Intel, le mélange d'instructions XMM sans préfixe VEX et d'instructions utilisant des registres YMM entraîne une pénalité de performance importante.

Le jeu d'instructions AVX-512 utilise un nouveau préfixe de 4 octets appelé EVEX. Bien que le préfixe EVEX soit plus long d'un ou deux octets que le préfixe VEX, il permet un encodage plus efficace des opérandes mémoire impliquant un pointeur et un décalage. Grâce au préfixe EVEX, les opérandes mémoire comportant un décalage de 4 octets peuvent parfois être remplacés par un décalage mis à l'échelle (*scaled offset*) d'un seul octet, réduisant ainsi la longueur totale de l'instruction.

10.2 Utilisation de constantes et d'adresses plus courtes

De nombreuses adresses de saut, adresses de données et constantes de données peuvent être exprimées sous forme de constantes sur 8 bits avec extension de signe. Cela permet un gain d'espace considérable. Un octet avec extension de signe ne peut être utilisé que si la valeur est comprise dans l'intervalle allant de -128 à +127.

Pour les adresses de saut, cela signifie que les sauts courts occupent deux octets de code, tandis que les sauts dépassant 127 octets en occupent 5 s'ils sont inconditionnels et 6 s'ils sont conditionnels.

De même, les adresses de données occupent moins d'espace si elles peuvent être exprimées sous la forme d'un pointeur et d'un déplacement compris entre -128 et +127. L'exemple suivant suppose que [mem1] et [mem2] sont des adresses mémoire statiques situées dans le segment de données et que la distance qui les sépare est inférieure à 128 octets :

; **Exemple 10.7a.** Opérandes mémoire statiques

```
mov  ebx, [mem1]   ; 6 octets
```

```
add  ebx, [mem2]   ; 6 octets
```

Que l'on peut réduire ainsi :

; **Exemple 10.7b.** Remplacement des adresses par des pointeurs

```
mov  eax, offset mem1      ; 5 octets
```

```
mov  ebx, [eax]            ; 2 octets
```

```
add  ebx, [eax] + (mem2 - mem1) ; 3 octets
```

En mode 64 bits, il faut remplacer `mov eax, offset mem1` par `lea rax, [mem1]`, instruction qui est plus longue d'un octet. L'avantage d'utiliser un pointeur s'accroît évidemment si l'on peut réutiliser ce même pointeur à plusieurs reprises. Stocker des données sur la pile et utiliser EBP ou ESP comme pointeur permet donc de réduire la taille du code par rapport à l'utilisation d'emplacements mémoire statiques et d'adresses absolues, à condition bien sûr que les données se situent dans une plage de +/- 127 octets autour du pointeur. L'utilisation de PUSH et POP pour écrire et lire des données entières temporaires permet une écriture encore plus concise.

Les constantes peuvent également occuper moins d'espace si elles sont comprises entre -128 et +127. La plupart des instructions acceptant des opérandes immédiats disposent d'une forme courte où l'opérande est codé sur un seul octet avec extension de signe. Exemples :

```

; Exemple 10.8. Opérandes à extension de signe
push 200      ; 5 octets
push 100      ; 2 octets, signe étendu

add ebx, 128   ; 6 octets
sub ebx, -128  ; 3 octets, signe étendu

```

Les seules instructions avec opérande immédiat ne disposant pas d'une forme courte à constante de 8 bits avec extension de signe sont MOV, TEST, CALL et RET. Une instruction TEST avec un opérande immédiat de 32 bits peut être remplacée par diverses alternatives plus courtes, selon la logique concernée. Quelques exemples :

```

; Exemple 10.9. Alternatives à tester avec une constante 32 bits
test eax, 8    ; 5 octets
test ebx, 8    ; 6 octets
test al, 8     ; 2 octets
test bl, 8     ; 3 octets
and  ebx, 8    ; 3 octets
bt   ebx, 3    ; 4 octets (utilisation du flag de retenue)
cmp  ebx, 8    ; 3 octets

```

Il est déconseillé d'utiliser les versions avec des constantes sur 16 bits en mode 32 ou 64 bits (comme TEST AX, 800H), car cela entraîne une pénalité de décodage due au préfixe de modification de taille sur certains processeurs, comme l'explique le manuel 3 : “The microarchitecture of Intel, AMD and VIA CPUs”.

Il est souvent utile d'employer des alternatives plus courtes pour les instructions MOV *registre, constante*. Exemples :

```

; Exemple 10.10. Chargement de constantes dans des registres 32 bits
mov  eax, 0      ; 5 octets
xor  eax, eax    ; 2 octets
mov  eax, 1      ; 5 octets
xor  eax, eax / inc eax ; 3 octets
push 1 / pop eax ; 3 octets
mov  eax, -1     ; 5 octets
or   eax, -1     ; 3 octets

```

Vous pouvez également envisager de réduire la taille des données statiques. Il est évidemment possible de réduire la taille d'un tableau en utilisant des éléments de plus petite taille ; par exemple, en employant des entiers sur 16 bits au lieu de 32 bits, à condition que les données tiennent effectivement dans ce format réduit. Le code nécessaire pour accéder à des entiers sur 16 bits est légèrement plus volumineux que celui utilisé pour des entiers sur 32 bits, mais cette augmentation reste négligeable face à la réduction de la taille des données dans le cas d'un tableau volumineux. Il convient d'éviter les instructions utilisant des opérandes immédiats sur 16 bits en modes 32 et 64 bits, en raison des problèmes liés au décodage des préfixes modifiant la longueur de l'instruction.

10.3 Réutilisation de constantes

Si la même adresse ou la même constante est utilisée plus d'une fois, vous pouvez la charger dans un registre. Une instruction MOV avec un opérande immédiat de 4 octets peut parfois être remplacée par une instruction arithmétique si la valeur du registre avant le MOV est connue. Exemple :

```

; Exemple 10.11a. Chargement de constantes 32 bits
mov  [mem1], 200 ; 10 octets
mov  [mem2], 201 ; 10 octets
mov  eax, 100    ; 5 octets
mov  ebx, 150    ; 5 octets

```

A remplacer par :

```

; Exemple 10.11b. Réutilisation des constantes
mov  eax, 200    ; 5 octets
mov  [mem1], eax ; 5 octets
inc  eax         ; 1 octet
mov  [mem2], eax ; 5 octets
sub  eax, 101    ; 3 octets
lea  ebx, [eax+50] ; 3 octets

```

10.4 Constantes en mode 64 bits

En mode 64 bits, il existe trois façons de charger une constante dans un registre 64 bits : avec une constante 64 bits, avec une constante 32 bits étendue par le signe et avec une constante 32 bits étendue par des zéros :

```

; Exemple 10.12. Chargement de constantes dans des registres 64 bits

```

```

mov rax, 123456789abcdef0h ; 10 octets (constante 64-bits)
mov rax, -100                ; 7 octets (32-bits en signe étendu)
mov eax, 100                 ; 5 octets (32-bits en extension à zéro)

```

Certains assembleurs utilisent la version avec extension de signe plutôt que la version plus courte avec extension de zéro, même lorsque la constante se situe dans la plage compatible avec une extension de zéro. Vous pouvez forcer l'assembleur à utiliser la version avec extension de zéro en spécifiant un registre de destination de 32 bits. Les écritures dans un registre de 32 bits entraînent toujours une extension de zéro au sein du registre de 64 bits.

10.5 Adresses et pointeurs en mode 64 bits

Le code 64 bits devrait de préférence utiliser des registres de 64 bits pour la base et l'index dans les adresses, et des registres de 32 bits pour tout le reste. Exemple :

```

; Exemple 10.13. Registres 64 bits contre registres 32 bits
mov  eax, [rbx + 4*rcx]
inc  rcx

```

Ici, vous pouvez économiser un octet en remplaçant `inc rcx` par `inc ecx`. Cela fonctionnera car la valeur du registre d'index sera nécessairement inférieure à 2^{32} . En revanche, le registre de base peut dépasser 2^{32} sur certains systèmes ; vous ne pouvez donc pas remplacer `add rbx, 4` par `add ebx, 4`. N'utilisez jamais de registres 32 bits tels que ceux de base ou d'index entre crochets en mode 64 bits.

La règle consistant à utiliser des registres 64 bits entre crochets pour une adresse indirecte et des registres 32 bits partout ailleurs s'applique également à l'instruction `LEA`. Exemples :

```

; Exemple 10.14. LEA en mode 64 bits
lea  eax, [ebx + ecx] ; 4 octets (nécessite un préfixe de taille d'adresse)
lea  eax, [rbx + rcx] ; 3 octets (aucun préfixe)
lea  rax, [ebx + ecx] ; 5 octets (taille d'adresse et préfixe REX)
lea  rax, [rbx + rcx] ; 4 octets (nécessite un préfixe REX)

```

La forme utilisant une destination sur 32 bits et une adresse sur 64 bits est préférable, sauf si un résultat sur 64 bits est requis. Cette version ne nécessite pas plus de temps d'exécution que celle utilisant une destination sur 64 bits. Les formes incluant un préfixe de taille d'adresse ne doivent jamais être utilisées.

Il est possible de réduire la taille d'un tableau de pointeurs 64 bits dans un programme 64 bits en utilisant des pointeurs 32 bits définis par rapport à la base de l'image ou à un point de référence. Cette approche réduit la taille du tableau de pointeurs, mais au prix d'une augmentation de la taille du code qui les utilise, celui-ci devant ajouter la base de l'image. L'intérêt global de cette méthode dépend de la taille du tableau.

Exemple :

```

; Exemple 10.15a. Table de sauts en mode 64 bits
section .data
JumpTable DQ Label1, Label2, Label3, ..

section .text
mov  eax, [n]          ; Index
lea  rdx, [JumpTable] ; Adresse de la table de saut
jmp  near [rdx+rax*8] ; Saut à JumpTable[n]

```

L'implémentation utilisant des pointeurs relatifs à l'image n'est disponible que sous Windows :

```

; Exemple 10.15b. Table de sauts relative à l'image sous Windows 64 bits
; Cela ne fonctionne qu'avec l'assembleur MASM.
section .data
JumpTable DD imagerel(Label1), imagerel(Label2), imagerel(Label3), ..
extrn __ImageBase:byte

section .text
mov  eax, [n]          ; Index
lea  rdx, [__ImageBase] ; Base de l'image
mov  eax, [rdx+rax*4+imagerel(JumpTable)] ; Chargement de l'adresse relative de l'image
add  rax, rdx          ; Ajout de la base de l'image à l'adresse
jmp  rax               ; Saut à l'adresse calculée

```

Une autre possibilité consiste à utiliser la table de saut elle-même comme point de référence. Cette méthode est couramment employée par les compilateurs destinés au système d'exploitation Mac OS, mais elle peut également être utilisée dans d'autres systèmes 64 bits. Il convient de noter que tous les assembleurs ne permettent pas d'établir une référence relative entre la section de données et la section de code. Cet exemple fonctionne avec l'assembleur NASM.

```

; Exemple 10.15c. Table de sauts à adressage relatif sur Mac 64 bits

```

```

section .data
JumpTable: DD Label1-JumpTable, Label2-JumpTable, Label3-JumpTable

section .text
default rel
mov  eax, [n]           ; Index
lea  rdx, [JumpTable]   ; Tableau et point de référence
movsxd rax, [rdx + rax*4] ; Chargement de l'adresse relative de la table
add  rax, rdx           ; Ajout de la base du tableau à l'adresse
jmp  rax               ; Saut à l'adresse calculée

```

Une alternative simple consiste à utiliser des pointeurs absolus de 32 bits. Cette méthode ne peut être utilisée que si l'on a la certitude que toutes les adresses sont inférieures à 2^{31} :

- ; **Exemple 10.15d.** Table de sauts absolus 32 bits sous Linux 64 bits
- ; Nécessite que les adresses soient inférieures à 2^{31}

```

section .data
JumpTable DD Label1, Label2, Label3      ; adresses 32-bits

section .text
default rel
mov  eax, [n]           ; Index
mov  eax, [JumpTable+rax*4] ; Chargement de l'adresse 32-bits
jmp  rax               ; Saut à l'adresse étendue par des zéros

```

Dans l'exemple 10.15d, l'adresse de JumpTable est une adresse relogeable sur 32 bits, étendue à 64 bits par extension de signe. Cela fonctionne si l'adresse est inférieure à 2^{31} . Les adresses de Label1, etc., font l'objet d'une extension par des zéros ; cela fonctionnera donc si ces adresses sont inférieures à 2^{32} . La méthode de l'exemple 10.15d peut être utilisée s'il est certain que la somme de la base de l'image et de la taille du programme est inférieure à 2^{31} . Cette approche fonctionne pour les programmes d'application sous Linux et BSD, ainsi que dans certains cas sous Windows, mais pas sous Mac OS X (voir page 16).

Il est même possible de remplacer les pointeurs sur 64 ou 32 bits par des déplacements sur 16 bits par rapport à un point de référence approprié :

- ; **Exemple 10.15d.** Décalages sur 16 bits par rapport à un point de référence

```

section .data
; adresses 16 bits relatives à Label1 :
JumpTable: DW 0, Label2-Label1, Label3-Label1

section .text
default rel
mov  eax, [n]           ; Index
lea  rdx, [JumpTable]   ; Adresse du tableau (relatif-RIP)
movsx rax, word [rdx+rax*2] ; Extension du signe du décalage de 16 bits
lea  rdx, [Label1]      ; Utilisation de Label1 comme point de référence.
add  rax, rdx           ; Ajout d'un décalage au point de référence
jmp  rax               ; Saut à l'adresse calculée

```

L'exemple 10.15d utilise Label1 comme point de référence. Il ne fonctionne que si toutes les étiquettes se situent dans l'intervalle $\text{Label1} \pm 2^{15}$. La table contient les déplacements sur 16 bits qui, après extension de signe, sont ajoutés au point de référence.

Les exemples ci-dessus présentent différentes méthodes de stockage des pointeurs de code. Ces mêmes méthodes peuvent être appliquées aux pointeurs de données. Un pointeur peut être stocké sous la forme d'une adresse absolue sur 64 bits, d'une adresse relative sur 32 bits, d'une adresse absolue sur 32 bits, ou encore d'un déplacement sur 8 ou 16 bits par rapport à un point de référence approprié. Les méthodes utilisant des pointeurs relatifs à la base de l'image ou à un point de référence ne justifient le code supplémentaire que si plusieurs pointeurs sont impliqués. C'est généralement le cas pour les grandes instructions de type `switch` et pour les listes chaînées.

10.6 Allongement des instructions pour assurer l'alignement

Il existe des situations où il peut être avantageux d'aller à l'encontre des recommandations formulées dans les paragraphes précédents afin d'allonger les instructions. Le cas le plus important concerne l'alignement du point d'entrée d'une boucle (voir p. 67). Plutôt que d'insérer des instructions NOP pour aligner l'étiquette d'entrée de la boucle, on peut choisir des versions des instructions précédentes plus longues que leur taille minimale, de manière à ce que le point d'entrée de la boucle se retrouve correctement aligné. Ces versions allongées ne nécessitent pas davantage de temps d'exécution ; on économise ainsi le temps qu'auraient pris les instructions NOP.

En règle générale, l'assembleur choisit la forme la plus courte possible pour une instruction. Il est souvent possible d'opter pour une forme plus longue de la même instruction ou d'une instruction équivalente. Cela peut se faire de plusieurs manières.

Utilisation de la forme générale plutôt que la forme abrégée d'une instruction

Les formes courtes de INC, DEC, PUSH, POP, XCHG, ADD et MOV ne comportent pas d'octet mod-reg-r/m (voir p. 18). Ces mêmes instructions peuvent être codées sous leur forme générale, avec un octet mod-reg-r/m. Exemples :

```
; Exemple 10.16. Allongement des instructions
inc  eax          ; format court (short). 1 octet (en mode 32 bits uniquement)
DB   0FFH, 0C0H   ; forme longue (long) de INC EAX, 2 octets
push ebx          ; forme courte (short). 1 octet
DB   0FFH, 0F3H   ; forme longue de PUSH EBX, 2 octets
```

Utilisation d'une instruction équivalente plus longue.

Exemples:

```
; Exemple 10.17. Allongement des instructions
inc  eax          ; 1 octet (en mode 32 bits uniquement)
add  eax, 1        ; Remplacement sur 3 octets de INC EAX
mov  eax, ebx      ; 2 octets
lea  eax, [ebx]    ; peut avoir une longueur comprise entre 2 et 8 octets ; voir ci-dessous
```

Utilisation d'un opérande immédiat de 4 octets

Les instructions comportant un opérande immédiat de 8 bits avec extension de signe peuvent être remplacées par la version utilisant un opérande immédiat de 32 bits :

```
; Exemple 10.18. Allongement des instructions
add  ebx, 1        ; 3 octets. Utilise un opérande de 8 bits avec extension de signe.
DB   0x81, 0xC3
DD   1             ; 6 octets. add ebx, 1 avec un opérande 32 bits.
```

Ajout d'un déplacement nul au pointeur

Une instruction avec un pointeur peut avoir un déplacement de 1 ou 4 octets en mode 32 bits ou 64 bits (1 ou 2 octets en mode 16 bits). Un déplacement factice de zéro peut être utilisé pour allonger l'instruction :

```
; Exemple 10.19. Allongement des instructions
mov  eax, [ebx]    ; 2 octets
; mov eax, [ebx+00] ; 3 octets
DB   0x8B, 0x43, 0x00 ; Ajout d'un déplacement d'un octet
; mov eax, [ebx+00000000] ; 6 octets
DB   0x8B, 0x83, 0, 0, 0, 0 ; Ajout d'un déplacement de 4 octets
```

On peut faire de même avec LEA EAX, [EBX+0] pour remplacer MOV EAX, EBX.

Utilisation de l'octet SIB

Une instruction comportant un opérande mémoire peut inclure un octet SIB (voir p. 18). Un octet SIB peut être ajouté à une instruction qui n'en possède pas déjà, augmentant ainsi la longueur de l'instruction d'un octet. Un octet SIB ne peut pas être utilisé en mode 16 bits ni en mode 64 bits avec un adressage relatif au RIP. Exemple :

```
; Exemple 10.20. Allongement des instructions
mov  eax, [ebx]    ; Longueur = 2 octets
DB   0x8B, 0x04, 0x23 ; Il en va de même pour l'octet SIB. Longueur = 3 octets.
DB   0x8BH, 0x44H, 0x23, 0 ; Avec octet SIB et déplacement. 4 octets.
```

Utilisation de préfixes

Un moyen simple d'allonger une instruction consiste à ajouter des préfixes inutiles. Toutes les instructions comportant un opérande mémoire peuvent recevoir un préfixe de segment. Le préfixe de segment DS est rarement nécessaire, mais il peut être ajouté sans modifier la signification de l'instruction :

```
; Exemple 10.21. Allongement des instructions
DB   3EH           ; Préfixe de segment DS
mov  eax, [ebx]    ; préfixe + instruction = 3 octets
```

Toutes les instructions comportant un opérande mémoire peuvent être dotées d'un préfixe de segment, y compris l'instruction LEA. Il est même possible d'ajouter un préfixe de segment à des instructions ne comportant pas d'opérande mémoire ; ces préfixes sans effet sont simplement ignorés. Toutefois, rien ne garantit absolument qu'un tel préfixe n'aura pas de signification particulière sur les futurs processeurs. Par exemple, le Pentium 4 utilisait des

préfixes de segment sur les instructions de branchement comme indices pour la prédiction de branchement. La probabilité que des préfixes de segment aient un effet indésirable sur les futurs processeurs pour les instructions susceptibles de comporter un opérande mémoire — c'est-à-dire celles incluant un octet `mod-reg-r/m` — est, à mon avis, très faible.

Les préfixes de segment CS, DS, ES et SS sont sans effet en mode 64 bits, mais ils restent autorisés, conformément au document “AMD64 Architecture Programmer's Manual, Volume 3: General-Purpose and System Instructions” (2003).

En mode 64 bits, vous pouvez également utiliser un préfixe REX vide pour allonger les instructions :

```
; Exemple 10.22. Allongement des instructions
DB 40H          ; préfixe REX vide
mov eax,[rbx]   ; préfixe + instruction = 3 octets
```

Les préfixes REX “vides” peuvent être appliqués sans risque à presque toutes les instructions en mode 64 bits ne possédant pas déjà de préfixe REX, à l'exception de celles utilisant les registres AH, BH, CH ou DH. Les préfixes REX ne peuvent pas être utilisés en mode 32 bits ou 16 bits. Un préfixe REX doit être placé après tout autre préfixe, et aucune instruction ne peut comporter plus d'un préfixe REX.

Le manuel d'optimisation d'AMD recommande d'utiliser jusqu'à trois préfixes de taille d'opérande (66H) comme éléments de remplissage. Toutefois, ce préfixe ne peut être employé qu'avec des instructions sur lesquelles il n'a aucun effet, comme les instructions NOP ou les instructions en virgule flottante x87. Les préfixes de segment sont plus polyvalents et produisent le même résultat — ou plutôt, la même absence d'effet.

Il est possible d'ajouter plusieurs préfixes identiques à une instruction, à condition que la longueur totale de celle-ci n'excède pas 15 octets. Par exemple, une instruction peut comporter deux ou trois préfixes de segment DS. Cependant, sur de nombreux processeurs, les instructions comportant plusieurs préfixes nécessitent davantage de temps pour être décodées. Il est déconseillé d'utiliser des préfixes de taille d'adresse comme éléments de remplissage, car cela risque de ralentir le décodage des instructions.

Ne placez pas de préfixes factices juste avant une étiquette de saut pour l'aligner :

```
; Exemple 10.23. Une mauvaise façon d'allonger les instructions
L1: mov ecx,1000
    DB 0x3E          ; Préfixe de segment DS. Erreur !
L2: mov eax,[esi]    ; Exécuté avec et sans préfixe
```

Dans cet exemple, l'instruction `MOV EAX, [ESI]` sera décodée avec un préfixe de segment DS si l'on arrive depuis L1, mais sans ce préfixe si l'on arrive depuis L2. Cela fonctionne en principe, mais certains microprocesseurs mémorisent les limites des instructions ; ces processeurs seront perturbés lorsqu'une même instruction débutera à deux emplacements différents. Cela peut entraîner une baisse des performances.

Les processeurs prenant en charge le jeu d'instructions AVX utilisent des préfixes VEX, d'une longueur de 2 ou 3 octets. Un préfixe VEX de 2 octets peut toujours être remplacé par un préfixe VEX de 3 octets. Un préfixe VEX peut être précédé de préfixes de segment, mais d'aucun autre type de préfixe. Aucun autre préfixe n'est autorisé après le préfixe VEX. La plupart des instructions utilisant des registres XMM ou YMM peuvent comporter un préfixe VEX. Il convient de ne pas mélanger les instructions vectorielles YMM avec préfixe VEX et les instructions XMM sans préfixe VEX (voir [chapitre 13.1](#)). Quelques instructions récentes opérant sur des registres d'usage général utilisent également un préfixe VEX.

Le jeu d'instructions AVX512 utilise un préfixe de 4 octets appelé EVEX. Les instructions peuvent être allongées en remplaçant un préfixe VEX par un préfixe EVEX lorsque l'extension AVX512VL est prise en charge.

Il est recommandé de vérifier les instructions codées manuellement à l'aide d'un débogueur ou d'un désassembleur afin de s'assurer de leur exactitude.

10.7 Utilisation de NOPs multi-octets pour l'alignement

L'instruction NOP multi-octets possède le code opération `0F 1F` suivi d'un opérande mémoire fictif. La longueur de cette instruction peut être ajustée en ajoutant, de manière optionnelle, un déplacement de 1 ou 4 octets ainsi qu'un octet SIB à l'opérande mémoire fictif, et en incluant un ou plusieurs préfixes 66H. Un nombre excessif de préfixes peut entraîner un ralentissement sur les microprocesseurs plus anciens, mais l'utilisation d'au moins deux préfixes est acceptable sur la plupart des processeurs. Il est ainsi possible de construire des NOP de toute longueur jusqu'à 10 octets en utilisant au maximum deux préfixes. Si le processeur gère les préfixes multiples sans pénalité de performance, la longueur peut atteindre 15 octets.

Le NOP multi-octets est plus efficace que les pseudo-NOP couramment utilisés, tels que `'MOV EAX, EAX'` ou `LEA RAX, [RAX+0]`. L'instruction NOP multi-octets est prise en charge par tous les processeurs de la famille Intel P6 et les modèles ultérieurs, ainsi que par les processeurs AMD Athlon, K7 et ultérieurs ; autrement dit, par la quasi-totalité des processeurs prenant en charge les instructions de déplacement conditionnel.

11 Optimisation de l'accès à la mémoire

La lecture à partir du cache de niveau 1 prend environ 3 cycles d'horloge. La lecture à partir du cache de niveau 2 prend de l'ordre de 10 cycles d'horloge. La lecture à partir de la mémoire principale prend de l'ordre de 100 cycles d'horloge. Le temps d'accès est encore plus long si la limite d'une page DRAM est franchie, et extrêmement long si la zone mémoire a été transférée sur le disque. Je ne peux pas donner ici les temps d'accès exacts car cela dépend de la configuration matérielle. Les chiffres ne cessent de changer grâce au développement technologique rapide.

Cependant, il ressort clairement de ces chiffres que la mise en cache du code et des données est extrêmement importante pour les performances. Si le code présente de nombreux échecs de cache et si chaque échec de cache coûte plus d'une centaine de cycles d'horloge, cela peut constituer un goulot d'étranglement très sérieux pour les performances.

Plus de conseils sur la façon d'organiser les données pour une mise en cache optimale sont donnés dans le manuel 1 : "Optimisation des logiciels en C++". Des détails spécifiques au processeur sont donnés dans le manuel 3 : "La microarchitecture des processeurs Intel, AMD et VIA" et dans les manuels d'optimisation logicielle d'Intel et d'AMD.

11.1 Fonctionnement de la mise en cache

Une mémoire cache est un dispositif de stockage temporaire situé plus près du microprocesseur que la mémoire principale. Les données et le code fréquemment utilisés, ou dont l'utilisation est imminente, sont stockés dans le cache afin d'accélérer leur accès. Les microprocesseurs peuvent comporter un, deux ou trois niveaux de cache. Le cache de niveau 1 (L1) est proche du cœur du microprocesseur et son accès ne nécessite que quelques cycles d'horloge. Un cache de niveau 2 (L2), de plus grande capacité, est intégré à la même puce ou, du moins, au même boîtier.

Le cache de données de niveau 1 du processeur Intel Skylake, par exemple, peut contenir 32 Ko de données. Il est organisé en 512 lignes de 64 octets chacune. Ce cache est de type associatif par ensembles à 8 voies (*8-way set-associative*). Cela signifie que les données provenant d'une adresse mémoire spécifique ne peuvent pas être affectées à une ligne de cache arbitraire, mais uniquement à l'une des huit lignes possibles. La longueur de ligne étant de 64 octets dans cet exemple, chaque ligne doit être alignée sur une adresse divisible par 64. Les 6 bits de poids faible (bits 0 à 5) de l'adresse mémoire servent à désigner un octet spécifique parmi les 64 octets de la ligne de cache. Comme chaque ensemble comprend 8 lignes, on compte $512 / 8 = 64$ ensembles distincts. Les six bits suivants (bits 6 à 11) de l'adresse mémoire permettent donc de sélectionner l'un de ces 64 ensembles. Les bits restants peuvent prendre n'importe quelle valeur. Il ressort de ce raisonnement mathématique que si les bits 6 à 11 de deux adresses mémoire sont identiques, ces adresses seront stockées dans le même ensemble de lignes de cache. Les blocs mémoire de 64 octets entrant en concurrence pour le même ensemble de lignes de cache sont espacés de $2^{12} = 4096$ octets. Au maximum huit de ces adresses peuvent être présentes simultanément dans le cache.

Les tailles de cache, les tailles de ligne de cache et l'associativité par ensemble des différents microprocesseurs sont décrites dans le manuel 3 : "The microarchitecture of Intel, AMD and VIA CPUs". La pénalité de performance liée aux conflits d'accès aux lignes de cache de niveau 1 peut être considérable sur les anciens microprocesseurs ; en revanche, sur les processeurs plus récents, la perte se limite à quelques cycles d'horloge, car les données sont probablement préchargées (*prefetched*) depuis le cache de niveau 2, lequel est accessible très rapidement via un bus de données 256 bits fonctionnant à pleine vitesse.

Les lignes de cache sont toujours alignées sur des adresses physiques divisibles par la taille de la ligne de cache (généralement 64 octets). Lorsqu'un octet est lu à une adresse divisible par 64, les 63 octets suivants sont également mis en cache et peuvent être lus ou écrits sans coût supplémentaire notable. Il est possible d'en tirer parti en regroupant les données utilisées conjointement au sein de blocs mémoire alignés de 64 octets.

Le cache d'instructions (code) de niveau 1 fonctionne de la même manière que le cache de données, sauf sur les processeurs dotés d'un cache de traces (*trace cache*). Le cache de niveau 2 est généralement partagé entre les instructions et les données.

11.2 Cache de traces

Les anciens processeurs Intel dotés de la microarchitecture NetBurst (Pentium 4) utilisaient un cache de traces plutôt qu'un cache d'instructions classique. Le cache de traces stocke le code après sa traduction en micro-opérations (μ ops), alors qu'un cache d'instructions ordinaire conserve le code brut, sans traduction. Ce type de cache élimine le goulot d'étranglement lié au décodage des instructions et tente de stocker le code dans l'ordre de son exécution plutôt que dans celui de sa disposition en mémoire. Le principal inconvénient du cache de traces est qu'il nécessite davantage d'espace pour stocker le code qu'un cache d'instructions classique. Les processeurs Intel ultérieurs ne disposent pas de cache de traces, mais éventuellement d'un cache de micro-opérations (cache μ op).

11.3 Cache de micro-opérations (μop)

Les microprocesseurs Intel depuis l'architecture Sandy Bridge, ainsi que les processeurs AMD depuis Zen 1, disposent d'un cache d'instructions traditionnel en amont des décodeurs et d'un cache de micro-opérations (μop) en aval. Ce cache de μops constitue un avantage majeur, car le décodage des instructions représente souvent un goulot d'étranglement. Sa capacité maximale est de 1 536 μops pour les processeurs Intel Skylake et ultérieurs, de 2 048 μops pour l'architecture AMD Zen 1 et de 4 096 μops pour les architectures AMD Zen 2 et Zen 3. Le cache de μops est une ressource si critique que le programmeur doit en optimiser l'usage et veiller à ce que la partie critique du code y tienne entièrement ; il convient notamment d'éviter le déroulage de boucles inutile.

Les processeurs Intel plus récents intègrent également un petit tampon de rebouclage (*loopback buffer*) capable de stocker les 30 à 60 dernières micro-opérations. Une boucle très courte tenant dans ce tampon s'exécute avec une rapidité accrue, en l'absence d'autres goulots d'étranglement.

Sur les anciens processeurs AMD, les limites des instructions sont marquées directement dans le cache d'instructions. Cette approche lève le verrou critique lié au décodage de la longueur des instructions et peut donc être considérée comme une alternative au cache de μops. J'ignore pourquoi cette technique est rarement employée dans les processeurs Intel.

11.4 Alignement des données

Toutes les données en mémoire vive (RAM) doivent être alignées sur des adresses divisibles par une puissance de 2, selon le schéma suivant :

Taille de l'opérande	Alignement
1 (byte)	1
2 (word)	2
4 (dword)	4
6 (fword)	8
8 (qword)	8
10 (tbyte)	16
16 (oword, xmmword)	16
32 (ymmword)	32
64 (zmmword)	64
Tableau 11.1. Alignement des données privilégié	

L'exemple suivant illustre l'alignement des données statiques :

```
; Exemple 11.1. alignement des données statiques
section .data align=16
A:    DQ 0, 0      ; A est aligné par 16
B:    times 32 DB 0
C:    DD 0
D:    DW 0
ALIGN 16          ; E doit être aligné par 16
E:    DQ 0, 0

section .text
default rel
movdqa xmm0, [A]
movdqa [E], xmm0
```

Dans l'exemple ci-dessus, A, B et C commencent tous à des adresses divisibles par 16. D commence à une adresse divisible par 4, ce qui est largement suffisant car il ne nécessite qu'un alignement sur 2 octets. Une directive d'alignement doit être insérée avant E, car l'adresse suivant D n'est pas divisible par 16, comme l'exige l'instruction MOVDQA. Une autre solution consisterait à placer E après A ou B pour assurer son alignement.

Certains microprocesseurs subissent une pénalité de plusieurs cycles d'horloge lors de l'accès à des données mal alignées chevauchant une limite de ligne de cache.

La plupart des instructions XMM dépourvues de préfixe VEX, qui lisent ou écrivent des opérandes mémoire de 16 octets, exigent que ces opérandes soient alignés sur 16 octets. Les instructions acceptant des opérandes de 16 octets non alignés peuvent s'avérer très inefficaces sur les processeurs plus anciens. Toutefois, cette contrainte est largement levée avec les jeux d'instructions AVX et ultérieurs. Les instructions AVX ne nécessitent pas l'alignement des opérandes mémoire, sauf pour les instructions spécifiquement conçues pour des données alignées. Les processeurs prenant en charge le jeu d'instructions AVX gèrent généralement très efficacement les opérandes mémoire mal alignés.

L'alignement des données stockées sur la pile peut être effectué en arrondissant par défaut la valeur du pointeur de pile. L'ancienne valeur du pointeur de pile doit bien entendu être restaurée avant le retour de la fonction. Une fonction utilisant des données locales alignées se présente généralement comme suit :

; **Exemple 11.2a.** Alignement explicite de la pile (Windows 64 bits)

```
FuncWithAlign:
    push    rbp                ; code du Prologue
    mov     rbp, rsp           ; Sauvegarde de la valeur du pointeur de pile
    sub     rsp, LocalSpace    ; Allocation d'espace pour les données locales
    and     rsp, -32           ; Alignement de RSP sur un multiple de 32
    mov     eax, [rbp+8]       ; Paramètre de fonction = tableau
    vmovdqu ymm0, [rax]        ; Chargement (load) depuis un tableau non aligné
    vmovdqa [rsp], ymm0        ; Stockage dans un espace aligné
    call    SomeOtherFunction  ; Appel d'une autre fonction
    ...
    mov     rsp, rbp           ; code d'Épilogue. Restauration de RSP
    pop     rbp                ; Restauration de RSP
    ret
```

Cette fonction utilise RBP pour traiter les paramètres de fonction et RSP pour traiter les données locales alignées. RSP est arrondi à la valeur la plus proche divisible par 32 simplement en lui ajoutant -32. Vous pouvez aligner la pile par n'importe quelle puissance de 2 en exécutant AND sur le pointeur de pile avec la valeur négative.

Tous les systèmes d'exploitation 64 bits et certains systèmes d'exploitation 32 bits (Mac OS, facultatif sous Linux) maintiennent la pile alignée par 16 pour toutes les instructions CALL. Cela élimine le besoin de l'instruction AND et du pointeur de trame. Il est nécessaire de propager cet alignement d'une instruction CALL à la suivante par un ajustement rigoureux du pointeur de pile dans chaque fonction :

; **Exemple 11.2b.** Propagation de l'alignement de la pile (Linux 32 bits)

```
FuncWithAlign:
    sub     rsp, 24            ; Allocation d'espace pour les données locales
    mov     rax, [rsp+32]      ; Paramètre de fonction = tableau
    movdqu xmm0, [rax]        ; Chargement (load) depuis un tableau non aligné
    movdqa [rsp], xmm0        ; Stockage dans un espace aligné
    call    SomeOtherFunction  ; Ce CALL doit être aligné.
    ...
    retFuncWithAlign ENDP
```

Dans l'exemple 11.2b, nous nous insistons sur le fait que le pointeur de pile est aligné sur 16 octets avant le CALL à FuncWithAlign. L'instruction CALL FuncWithAlign (non représentée ici) a empilé l'adresse de retour, ce qui a eu pour effet de soustraire 8 au pointeur de pile. Il faut soustraire 8 octets supplémentaires au pointeur de pile pour rétablir l'alignement sur 16. Ces 8 octets ne suffisent pas pour la variable locale qui en nécessite 16 ; nous devons donc soustraire 24 octets pour maintenir l'alignement du pointeur de pile sur 16. Les 8 octets de l'adresse de retour plus les 24 octets donnent 32, une valeur divisible par 16. N'oubliez pas d'inclure les éventuelles instructions PUSH dans le calcul. Si, par exemple, le prologue de la fonction contenait une instruction PUSH, il faudrait soustraire 16 ou 32 à RSP pour préserver l'alignement sur 16. L'exemple 11.2b nécessite un alignement de la pile pour deux raisons : l'instruction MOVDQA requiert un opérande aligné, et l'appel CALL SomeOtherFunction doit être effectué avec un alignement correct afin de transmettre cet alignement à SomeOtherFunction.

Les questions d'alignement sont également importantes lorsque l'on combine du C++ et du langage assembleur. Prenons l'exemple de cette structure C++ :

// **Exemple 11.3a.** Structure C++

```
struct abcd {
    unsigned char a; // occupe 1 octet de stockage
    int b;           // 4 octets de stockage
    short int c;     // 2 octets de stockage
    double d;        // 8 octets de stockage
} x;
```

La plupart des compilateurs (mais pas tous) inséreront trois octets vides entre a et b, et six octets vides entre c et d, afin d'assurer l'alignement naturel de chaque élément. Vous pouvez modifier la définition de la structure comme suit :

// **Exemple 11.3b.** Structure C++

```
struct abcd {
    double d;        // 8 octets de stockage
    int b;           // 4 octets de stockage
    short int c;     // 2 octets de stockage
```

```

        unsigned char a;    // 1 octet de stockage
        char unused[1];    // remplissage jusqu'à 16 octets
    } x;

```

Cela présente plusieurs avantages : l'implémentation est identique pour les compilateurs avec ou sans alignement automatique, la structure se traduit aisément en langage assembleur, tous les membres sont correctement alignés et le nombre d'octets inutilisés est réduit. L'octet supplémentaire inutilisé à la fin garantit que tous les éléments d'un tableau de structures sont correctement alignés. Reportez-vous à la page 121 pour découvrir comment déplacer efficacement des blocs de données non alignés.

11.5 Alignement du code

La plupart des microprocesseurs récupèrent le code par blocs alignés de 16 ou 32 octets. Si le point d'entrée d'un sous-programme important ou une étiquette de saut se trouve vers la fin d'un bloc de 16 octets, le microprocesseur ne récupérera que quelques octets de code utiles lors du chargement de ce bloc. Il devra peut-être charger également les 16 octets suivants avant de pouvoir décoder les premières instructions suivant l'étiquette. Ce problème peut être évité en alignant sur 16 octets les points d'entrée des sous-programmes et des boucles importants. Un alignement sur 8 octets garantit le chargement d'au moins 8 octets de code dès la première opération de récupération, ce qui peut suffire si les instructions sont courtes. On peut également aligner les points d'entrée des sous-programmes sur la taille de la ligne de cache (généralement 64 octets) si le sous-programme fait partie d'une zone critique très sollicitée (*hot spot*) et qu'il est peu probable que le code précédent soit exécuté dans le même contexte.

L'alignement du code présente toutefois un inconvénient : une partie de l'espace cache est perdue au niveau des zones vides précédant les points d'entrée alignés.

Dans la plupart des cas, l'impact de l'alignement du code est négligeable. Je recommande de n'aligner le code que dans les situations les plus critiques, comme pour les sous-programmes et les boucles internes les plus sollicités.

L'alignement du point d'entrée d'une sous-routine est simple : il suffit d'insérer autant d'instructions NOP que nécessaire avant ce point pour rendre l'adresse divisible par 8, 16, 32 ou 64, selon le besoin. L'assembleur réalise cette opération grâce à la directive ALIGN. Les instructions NOP insérées n'affectent pas les performances, car elles ne sont jamais exécutées.

L'alignement du point d'entrée d'une boucle est plus problématique, car le code qui le précède est également exécuté. Aligner l'entrée d'une boucle sur une frontière de 16 octets peut nécessiter jusqu'à 15 instructions NOP. Ces instructions seront exécutées avant l'entrée dans la boucle, ce qui consommera du temps processeur. Certains assembleurs utilisent des pseudo-NOP (comme MOV RAX, RAX ou LEA RBX, [RBX+0]) comme instructions de remplissage. Cette méthode présente toutefois l'inconvénient de créer une fausse dépendance au registre et de mobiliser des ressources d'exécution. Il est préférable d'utiliser l'instruction NOP multi-octets, dont la longueur peut être ajustée selon les besoins.

Une autre façon d'aligner l'entrée d'une boucle consiste à coder les instructions précédentes sous une forme plus longue que nécessaire. Dans la plupart des cas, cela n'augmente pas le temps d'exécution, bien que cela puisse accroître le temps de récupération des instructions. Voir la page 61 pour plus de détails sur la manière de coder des instructions dans des versions plus longues.

La méthode la plus efficace pour aligner la boucle la plus imbriquée consiste à déplacer le point d'entrée de la sous-routine précédente. L'exemple suivant illustre cette technique :

; **Exemple 11.4.** Alignement de l'entrée de la boucle. Assembleur NASM.

```

section .text
align 16
; Calcul où placer innerfunction afin d'aligner innerloop :
times -(innerloop - innerfunction) % 16 nop

innerfunction:                ; Cette adresse sera modifiée.
    mov rax, [rsp+8]          ; zone
    mov ecx, 0x10000          ; taille de la zone
    add rax, rcx               ; fin de la zone
    xor edx, edx              ; remise à zéro de sum
    neg rcx                   ; negation de l'index

innerloop:                    ; Cette entrée de boucle sera alignée sur 16
    add edx, [rax+rcx]         ; Calcul de la somme des éléments d'une zone
    inc rcx
    jnz innerloop
    ret

```

Ce code ajuste l'adresse de début de innerfunction de manière à ce que la boucle interne soit alignée sur une frontière de 16 octets. Cette méthode fonctionne avec l'assembleur NASM, mais d'autres assembleurs pourraient ne pas être en mesure d'effectuer les calculs nécessaires impliquant des références vers l'avant.

Le coût d'un mauvais alignement de la fonction est négligeable comparé au gain obtenu par l'alignement de la boucle interne, car cette dernière est la cible d'un saut 0x4000 fois plus souvent.

11.6 Organisation des données pour une mise en cache améliorée

La mise en cache des données est plus efficace lorsque les données critiques sont regroupées dans une zone mémoire contiguë de petite taille. L'emplacement idéal pour stocker ces données est la pile. L'espace de pile alloué à une sous-routine est libéré au retour de celle-ci, puis réutilisé par la sous-routine suivante appelée. Cette réutilisation de la même zone mémoire optimise la mise en cache. Par conséquent, il est préférable de stocker les variables sur la pile plutôt que dans le segment de données, lorsque cela est possible.

Les constantes en virgule flottante sont généralement stockées dans le segment de données. Cela pose problème, car il est difficile de maintenir la contiguïté des constantes utilisées par différentes sous-routines. Une alternative consiste à stocker les constantes directement dans le code. En mode 64 bits, il est possible de charger une constante en double précision via un registre d'entiers afin d'éviter d'utiliser le segment de données. Exemple :

; **Exemple 11.5a.** Chargement d'une constante de type double depuis le segment de données

```
section .data
C1 DQ SomeConstant

section .text
movsd xmm0, C1
```

Cela peut être modifié comme suit :

; **Exemple 11.5b.** Chargement d'une constante double depuis un registre (mode 64 bits)

```
section .text
mov rax, SomeConstant
movq xmm0, rax ; Certains assembleurs utilisent MOVD pour cette instruction.
```

Consultez la page 100 pour découvrir diverses méthodes permettant de générer des constantes sans charger de données depuis la mémoire. Cette approche est avantageuse si l'on prévoit des défauts de cache de données, mais elle ne l'est pas si la mise en cache des données est efficace.

Les tables de constantes sont généralement stockées dans le segment de données. Il peut être avantageux de copier une telle table du segment de données vers la pile, en dehors de la boucle la plus imbriquée, si cela permet d'améliorer l'efficacité du cache au sein de la boucle.

Les variables statiques sont des variables dont la valeur est conservée d'un appel de fonction à l'autre. Elles sont généralement stockées dans le segment de données. Une alternative préférable peut consister à encapsuler la fonction et ses données dans une classe C++. Cette classe peut être déclarée dans la partie C++ du code, même si la fonction membre est écrite en langage assembleur.

Pour les structures de données trop volumineuses pour le cache de données, il est préférable d'effectuer des accès linéaires et progressifs afin d'optimiser la prélecture et la mise en cache. Les accès non séquentiels peuvent entraîner des conflits au niveau des lignes de cache si le pas d'accès (*stride*) correspond à une puissance de 2 élevée. Le manuel 1, intitulé "Optimizing software in C++", présente des exemples montrant comment éviter les pas d'accès correspondant à des puissances de 2 élevées.

11.7 Organisation du code pour une mise en cache améliorée

La mise en cache du code est plus efficace lorsque la partie critique du code est située dans une zone mémoire contiguë dont la taille n'excède pas celle du cache d'instructions. Évitez de disperser les sous-programmes critiques à des adresses mémoire aléatoires. Le code rarement exécuté, comme les routines de gestion des erreurs, doit être séparé du code critique (ou "zone chaude").

Il peut être judicieux de diviser le segment de code en plusieurs segments distincts selon les parties du programme. Par exemple, vous pouvez créer un segment pour le code "chaud" (exécuté le plus fréquemment) et un autre pour le code "froid" (pour lequel la vitesse d'exécution n'est pas critique).

Une autre approche consiste à contrôler l'ordre d'édition des liens des modules, afin que ceux utilisés dans une même partie du programme soient placés à des adresses mémoire proches les uns des autres.

L'édition de liens dynamique des bibliothèques de fonctions (DLL ou objets partagés) réduit l'efficacité de la mise en cache du code. En effet, ces bibliothèques sont généralement chargées à des adresses mémoire "rondes" (alignées sur des puissances de 2 élevées). Cela peut engendrer des conflits de cache, car les distances séparant plusieurs DLL sont alors divisibles par des puissances de 2 élevées.

11.8 Instructions de contrôle du cache

Les opérations d'écriture en mémoire sont plus coûteuses que les lectures en cas de défaut de cache dans un système à écriture différée (*write-back*). En effet, lorsqu'un défaut de cache survient, une ligne de cache entière doit être lue depuis la mémoire, modifiée, puis réécrite. Il est possible d'éviter cela en utilisant les instructions d'écriture non temporelle MOVNTI, MOVNTQ, MOVNTDQ, MOVNTPD et MOVNTPS. Ces instructions doivent être employées lors de l'écriture vers un emplacement mémoire qui a peu de chances d'être mis en cache ou d'être relu avant que la ligne de cache correspondante ne soit évacuée. En règle générale, il est conseillé de n'utiliser les écritures non temporelles que pour l'écriture d'un bloc mémoire dont la taille dépasse la moitié de celle du cache de plus haut niveau.

La prélecture explicite des données à l'aide des instructions PREFETCH peut parfois améliorer les performances du cache, mais, dans la plupart des cas, la prélecture automatique s'avère plus efficace.

12 Boucles

Le point critique d'un programme sollicitant fortement le processeur est presque toujours une boucle. La fréquence d'horloge des ordinateurs modernes est si élevée que même les instructions les plus coûteuses en temps, les défauts de cache et les exceptions inefficaces s'exécutent en une fraction de microseconde. Le ralentissement dû à un code inefficace ne devient perceptible que lorsqu'il se répète des millions de fois. De telles fréquences de répétition ne se rencontrent généralement qu'au niveau le plus interne d'une série de boucles imbriquées. Ce chapitre traite des mesures permettant d'améliorer les performances des boucles.

12.1 Minimisation de la surcharge de la boucle

La surcharge liée à la boucle correspond aux instructions nécessaires pour revenir au début de la boucle et déterminer quand en sortir. L'optimisation de ces instructions est une technique assez générale, applicable dans de nombreuses situations. Toutefois, cette optimisation n'est pas nécessaire si un autre goulot d'étranglement limite la vitesse d'exécution. Voir la page 72 et les suivantes pour une description des goulots d'étranglement possibles au sein d'une boucle.

Une boucle typique en C++ peut se présenter comme suit :

```
// Exemple 12.1a. Boucle for typique en C++
for (int i = 0; i < n; i++) {
    // (loop body)
}
```

Sans optimisation, l'implémentation en assembleur ressemblera à ceci :

```
; Exemple 12.1b. Boucle for, non optimisée
mov ecx, n      ; Chargement de n
xor eax, eax    ; i = 0

LoopTop:
    cmp eax, ecx ; i < n
    jge LoopEnd  ; Sortie lorsque i >= n
    ; (loop body) ; Le corps de la boucle se place ici
    add eax, 1   ; i++
    jmp LoopTop  ; Saut en arrière

LoopEnd:
```

Le problème majeur de la boucle de l'exemple 12.1b est qu'elle contient deux instructions de saut. Nous pouvons éliminer un saut de la boucle en plaçant l'instruction de branchement à la fin :

```
; Exemple 12.1c. Boucle "for" avec un branchement à la fin
mov ecx, n      ; Chargement de n
test ecx, ecx   ; Test de n
jng LoopEnd     ; Ignorer si n <= 0
xor eax, eax    ; i = 0

LoopTop:
    ; (loop body) ; Le corps de la boucle se place ici
    add eax, 1    ; i++
    cmp eax, ecx  ; i < n
    jl LoopTop    ; Boucle arrière si i < n

LoopEnd:
```

Nous avons désormais éliminé l'instruction de saut inconditionnel de la boucle en précisant le branchement de sortie à la fin. Il est nécessaire d'ajouter une vérification préalable pour traiter le cas où la boucle ne doit pas s'exécuter (zéro itération). Sans cette vérification, la boucle s'exécuterait une fois lorsque $n = 0$.

Cette technique, consistant à placer le branchement de sortie à la fin, peut même s'appliquer à des structures de boucle complexes dont la condition de sortie se situe au milieu. Prenons l'exemple d'une boucle C++ avec une condition de sortie intermédiaire :

```
// Exemple 12.2a. Boucle C++ avec sortie en milieu de boucle
int i = 0;
while (true) {
    FuncA();           // Corps de la boucle supérieure
    if (++i >= n) break; // Condition de sortie ici
    FuncB();           // Corps de la boucle inférieure
}
```

Cela peut être implémenté en langage assembleur en réorganisant la boucle de manière à ce que la sortie se situe à la fin et l'entrée au milieu :

```
; Exemple 12.2b. Boucle en langage assembleur avec entrée au milieu
xor eax, eax    ; i = 0
jmp LoopEntry   ; Saut au milieu de la boucle

LoopTop:
    call FuncB   ; La partie inférieure de la boucle du corps vient en premier.
LoopEntry:
    call FuncA   ; La partie supérieure de la boucle du corps vient en dernier.
    add eax, 1
    cmp eax, n
    jge LoopTop  ; Condition de sortie finale
```

L'instruction `cmp` des exemples 12.1c et 12.2b peut être supprimée si le compteur s'arrête à zéro, car on peut s'appuyer sur l'instruction `add` pour positionner le flag de zéro. Cela peut être réalisé en décomptant de `n` vers zéro plutôt qu'en comptant de zéro vers `n` :

```
; Exemple 12.3. Boucle à compte à rebours
mov ecx, n      ; Chargement de n dans ECX
test ecx, ecx   ; Test de n
jng LoopEnd     ; Ignorer si n <= 0

LoopTop:
    ; (loop body) ; Le corps de la boucle se place ici
    sub ecx, 1    ; n--
    jnz LoopTop   ; Boucle arrière si non zéro
LoopEnd:
```

Désormais, la surcharge liée à la boucle est réduite à seulement deux instructions, ce qui constitue l'optimum. Il convient d'éviter les instructions `jecxz` et `loop`, car elles sont moins efficaces sur certains processeurs.

La solution présentée dans l'exemple 12.3 ne convient pas si la variable `i` doit être utilisée à l'intérieur de la boucle, par exemple comme indice de tableau. L'exemple suivant ajoute 1 à tous les éléments d'un tableau d'entiers :

```
; Exemple 12.4a. Boucle for avec un tableau
section .text
default rel
    mov ecx, n      ; Chargement de n dans ECX
    test ecx, ecx   ; Test de n
    jng LoopEnd     ; Ignorer si n <= 0
    xor eax, eax     ; i = 0
    lea rsi, [Array] ; Adresse du tableau

LoopTop:
    ; Corps de la boucle : ajout de 1 à tous les éléments du tableau :
    add dword [rsi+4*rax], 1
    add eax, 1       ; i++
    cmp eax, ecx     ; i < n
    jl LoopTop       ; Boucle arrière si i < n
LoopEnd:
```

L'adresse de début du tableau se trouve dans `rsi` et l'indice dans `rax`. L'indice est multiplié par 4 lors du calcul de l'adresse, car la taille de chaque élément du tableau est de 4 octets.

Il est possible de modifier l'exemple 12.4a pour effectuer un décompte décroissant plutôt que croissant ; toutefois, le cache de données est optimisé pour un accès aux données dans le sens de la progression (vers l'avant) et

non en sens inverse. Il est donc préférable d'effectuer un décompte croissant en partant de valeurs négatives, de -n jusqu'à zéro. Cela peut être réalisé en créant un pointeur vers la fin du tableau et en utilisant un décalage négatif par rapport à cette fin :

; **Exemple 12.4b.** Boucle for avec un indice négatif partant de la fin du tableau

```
section .text
default rel
    mov ecx, n          ; Chargement de n dans ECX
    lea rsi, [Array]    ; Adresse du tableau
    lea rsi, [rsi+4*rcx] ; Pointeur vers la fin du tableau
    neg rcx             ; i = -n
    jnl LoopEnd         ; Ignorer si (-n) >= 0
LoopTop:
    ; Corps de la boucle : ajout de 1 à tous les éléments du tableau :
    add dword [rsi+4*rcx], 1
    add rcx, 1          ; i++
    js LoopTop          ; Boucle arrière si i < 0
LoopEnd:
```

Une solution légèrement différente consiste à multiplier n par 4 et à compter de -4*n jusqu'à zéro :

; **Exemple 12.4c.** Boucle for avec indice négatif multiplié par la taille de l'élément

```
section .text
default rel
    mov ecx, n          ; Chargement de n dans ECX
    shl ecx, 2          ; n * 4
    jng LoopEnd         ; Ignorer si (4*n) <= 0
    lea rsi, [Array]    ; Adresse du tableau
    add rsi, rcx         ; Pointeur vers la fin du tableau
    neg rcx             ; i = -4*n
LoopTop:
    ; Corps de la boucle : ajout de 1 à tous les éléments du tableau :
    add dword [rsi+rcx], 1
    add rcx, 4          ; i += 4
    js LoopTop          ; Boucle arrière si i < 0
LoopEnd:
```

Il n'y a pas de différence de vitesse significative entre les exemples 12.4b et 12.4c ; toutefois, cette dernière méthode s'avère utile lorsque la taille des éléments du tableau n'est pas de 1, 2, 4 ou 8 octets, empêchant ainsi l'utilisation de l'adressage indexé avec mise à l'échelle (*scaled index addressing*).

Le compteur de boucle doit toujours être un entier, car les instructions de comparaison en virgule flottante sont bien moins efficaces que celles portant sur des entiers.

Certaines boucles reposent intrinsèquement sur une condition de sortie en virgule flottante. Par exemple, les développements en série de Taylor et les itérations de Newton-Raphson s'achèvent généralement lorsque les termes d'erreur résiduelle deviennent suffisamment faibles. Dans de tels cas, il peut être avantageux d'utiliser systématiquement un nombre maximal d'itérations correspondant au pire des scénarios, en s'appuyant sur un compteur entier. Le coût d'itérations supplémentaires est souvent inférieur aux économies réalisées en évitant de calculer la condition de sortie en virgule flottante à l'intérieur de la boucle.

Un autre avantage de cette méthode réside dans la prévisibilité accrue du branchement de sortie de boucle. Même en cas de mauvaise prédiction de ce branchement, le coût associé est moindre avec un compteur entier distinct ; en effet, les instructions sur entiers sont susceptibles d'être exécutées bien en amont des instructions en virgule flottante (plus lentes), permettant ainsi de résoudre l'erreur de prédiction beaucoup plus tôt. Il est préférable d'éviter les conditions de sortie dépendant de données calculées au sein même du corps de la boucle.

12.2 Variables d'induction

Les premiers compilateurs calculaient souvent une expression à l'intérieur d'une boucle à partir de la valeur obtenue lors de l'itération précédente. Par exemple, si la valeur en virgule flottante du compteur de boucle est requise à une autre fin, il peut être utile de disposer à la fois d'un compteur entier et d'un compteur en virgule flottante. Prenons l'exemple d'une boucle générant une table de sinus :

```
// Exemple 12.5a. Boucle C++ pour générer une table de sinus
double Table[100]; int i;
for (i = 0; i < 100; i++) Table[i] = sin(0.01 * i);
```

Cela peut être changé en :

```
// Exemple 12.5b. Boucle C++ pour générer une table de sinus
double Table[100], x; int i;
for (i = 0, x = 0.; i < 100; i++, x += 0.01) Table[i] = sin(x);
```

Cependant, les variables d'induction ne sont généralement pas efficaces sur les microprocesseurs modernes, car elles impliquent une chaîne de dépendance portée par la boucle et rendent la vectorisation plus difficile.

12.3 Déplacement du code invariant de boucle

Le calcul de toute expression dont la valeur ne varie pas au sein de la boucle doit être effectué en dehors de celle-ci.

Il en va de même pour les structures conditionnelles (if-else) dont la condition reste inchangée à l'intérieur de la boucle. On peut éviter une telle structure en créant deux boucles distinctes — une pour chaque branche — et en mettant en place une condition pour choisir entre les deux.

12.4 Identification des goulots d'étranglement

Il existe plusieurs goulots d'étranglement potentiels susceptibles de limiter les performances d'une boucle. Les plus probables sont :

- Défauts de cache et contentions de cache
- Chaînes de dépendance inter-itérations
- Récupération d'instructions
- Décodage d'instructions
- Validation (retrait) d'instructions
- Débit des ports d'exécution
- Débit des unités d'exécution
- Réordonnancement et ordonnancement sous-optimaux des micro-opérations (μops)
- Mauvaises prédictions de branchement
- Exceptions en virgule flottante et opérandes subnormaux

Si un goulot d'étranglement spécifique limite les performances, optimiser d'autres éléments ne sera d'aucune utilité. Il est donc crucial d'analyser minutieusement la boucle afin d'identifier quel goulot d'étranglement constitue le facteur limitant. Ce n'est qu'une fois le goulot d'étranglement le plus critique éliminé qu'il devient pertinent de s'intéresser au suivant. Les différents goulots d'étranglement sont abordés dans les sections suivantes. Tous ces détails sont spécifiques au processeur. Consultez le manuel 3, “The microarchitecture of Intel, AMD and VIA CPUs”, pour obtenir des explications sur les particularités liées aux processeurs mentionnées ci-dessous.

Il est parfois nécessaire de procéder à de nombreux essais pour identifier et résoudre le goulot d'étranglement limitant. Gardez à l'esprit qu'une solution trouvée par expérimentation est spécifique au processeur utilisé et ne sera pas nécessairement optimale sur des processeurs dotés d'une microarchitecture différente.

12.5 Optimisation pour l'exécution dans le désordre

Les microprocesseurs les plus performants dotés de capacités d'exécution dans le désordre (*out-of-order*) peuvent exécuter quatre, six, voire huit instructions simultanément. Cela permet au microprocesseur d'entamer l'exécution de chaque itération du corps de la boucle avant que l'itération précédente ne soit terminée. Ces processeurs disposent généralement de plus d'une centaine de registres temporaires capables de stocker simultanément les résultats intermédiaires de plusieurs itérations de la boucle. Les considérations suivantes peuvent s'avérer utiles pour optimiser une boucle critique située au niveau le plus interne, afin de tirer parti d'un parallélisme d'exécution dans le désordre intensif :

- Utilisez une condition de sortie de boucle simple, telle qu'un compteur entier.
- Évitez toute condition de sortie de boucle dépendant de données calculées au sein du corps de la boucle. Cela permet de résoudre le branchement de la boucle plusieurs itérations à l'avance.
- Évitez les branchements à l'intérieur du corps de la boucle s'ils sont difficilement prévisibles et ne peuvent être résolus à l'avance.
- Évitez toute chaîne de dépendance inter-itérations (portant sur plusieurs itérations) autre que celle liée au compteur de boucle.
- Si le corps de la boucle est volumineux et contient de longues chaînes de dépendance (ne portant pas sur plusieurs itérations) ainsi que de nombreuses variables susceptibles de consommer un nombre excessif de registres temporaires, il peut être avantageux de scinder la boucle en plusieurs boucles plus petites, chacune effectuant une partie du travail.

12.6 Récupération, décodage et validation des instructions dans une boucle

Les détails concernant l'optimisation de la récupération, du décodage et de la validation (retirement) des instructions, entre autres, sont spécifiques au processeur, comme indiqué à la page 49.

Si la récupération du code constitue un goulot d'étranglement, il est utile d'aligner le point d'entrée de la boucle sur une frontière de 16 octets et de réduire la taille des instructions afin de minimiser le nombre de frontières de 16 octets traversées par la boucle.

Si le décodage des instructions constitue un goulot d'étranglement, il peut être utile de respecter les règles spécifiques au processeur concernant les schémas de décodage. Évitez les instructions complexes générant plus de deux micro-opérations (μ ops), telles que LOOP, JECXZ, LODS, STOS, etc.

Il convient d'éviter les sauts et les appels au sein de la boucle, car ils retardent la récupération du code. Les sous-programmes appelés dans une boucle critique peuvent, si possible, faire l'objet d'une expansion en ligne (*inlining*).

Dans la mesure du possible, évitez les branchements à l'intérieur de la boucle, car ils peuvent perturber la prédiction du branchement de sortie de boucle. Toutefois, ne remplacez pas les branchements par des instructions de déplacement conditionnel (*conditional moves*) si cela allonge une chaîne de dépendance portée par la boucle (*loop-carried dependency chain*).

12.7 Répartition uniforme des micro-opérations entre les unités d'exécution

Le manuel 4, intitulé “Instruction tables” (Tables d'instructions), contient des tableaux indiquant le nombre de micro-opérations (μ ops) générées par chaque instruction ainsi que le port d'exécution vers lequel chaque micro-opération est dirigée. Ces informations sont, bien entendu, spécifiques au processeur. Elles sont nécessaires pour calculer le nombre total de micro-opérations générées par la boucle, ainsi que la répartition de ces micro-opérations entre les différents ports et unités d'exécution.

Le temps requis pour finaliser l'exécution (*retirement*) de toutes les instructions de la boucle correspond au nombre total de micro-opérations divisé par le taux de finalisation. Pour les processeurs récents, ce taux est d'au moins 4 micro-opérations par cycle d'horloge. Le temps de finalisation ainsi calculé représente la durée d'exécution minimale de la boucle. Cette valeur sert de référence pour évaluer d'éventuels autres goulots d'étranglement.

Sur la plupart des processeurs Intel, le débit d'un port d'exécution est d'une micro-opération par cycle d'horloge. La charge d'un port d'exécution donné se calcule en divisant le nombre de micro-opérations qui lui sont affectées par son débit. Si cette valeur dépasse le temps de finalisation calculé précédemment, ce port d'exécution risque fort de constituer un goulot d'étranglement. Les processeurs AMD ne disposent pas de ports d'exécution au sens strict, mais ils possèdent plusieurs pipelines aux débits comparables.

Sur les processeurs Intel, chaque port d'exécution peut comporter plusieurs unités d'exécution. La plupart de ces unités présentent le même débit que le port d'exécution associé ; dans ce cas, l'unité d'exécution ne peut pas constituer un goulot d'étranglement plus restrictif que le port lui-même. Toutefois, une unité d'exécution peut devenir un goulot d'étranglement dans les situations suivantes : (1) si son débit est inférieur à celui du port d'exécution (par exemple, pour les opérations de multiplication et de division) ; (2) si elle est accessible via plusieurs ports d'exécution ; et (3) sur les processeurs AMD, qui ne disposent pas de ports d'exécution.

La charge d'une unité d'exécution donnée se calcule en multipliant le nombre total de micro-opérations (μ ops) qui lui sont destinées par l'inverse de son débit. Si cette valeur dépasse le temps de validation (*retirement time*) calculé précédemment, il est probable que cette unité d'exécution constitue un goulot d'étranglement.

12.8 Un exemple d'analyse des goulots d'étranglement dans les boucles vectorielles

Nous allons ici décrire la théorie de l'analyse des goulots d'étranglement au sein d'un microprocesseur. Notez que cette analyse dépend fortement du processeur utilisé. Il est possible de créer plusieurs versions d'une partie critique du code, chacune optimisée pour un processeur spécifique, comme expliqué dans la section “CPU dispatching” (aiguillage selon le processeur) à la page 107. Toutefois, il n'est pas forcément judicieux d'optimiser le code pour un ou plusieurs microprocesseurs particuliers, car celui-ci pourrait être amené à s'exécuter sur des modèles plus récents, inexistant au moment de la programmation. Les écueils liés à cette technique d'aiguillage sont détaillés dans le manuel 1 (“Optimizing software in C++”), dans la section intitulée “CPU dispatch strategies”.

Les exemples présentés ci-dessous visent uniquement à illustrer la théorie ; ils n'ont pas pour but d'encourager l'optimisation pour un modèle de processeur spécifique. Le type d'optimisation décrit ici n'est pertinent que si le débit des unités d'exécution constitue un goulot d'étranglement.

La méthode de calcul de l'utilisation des différentes unités d'exécution est illustrée par l'exemple suivant, qui présente l'algorithme dit “DAXPY”, utilisé en algèbre linéaire :

```
// Exemple 12.6a. Code C++ pour l'algorithme DAXPY
int i; const int n = 100;
```

```
double X[n]; double Y[n]; double DA;
for (i = 0; i < n; i++) Y[i] = Y[i] - DA * X[i];
```

L'implémentation suivante concerne un processeur doté du jeu d'instructions SSE2 en mode 32 bits, en supposant que X et Y sont alignés par 16 :

```
; Exemple 12. 6b. Algorithme DAXPY, mode 32 bits
n equ 100                ; Définition de la constante n (paire et positive)
mov ecx, n * 8           ; Chargement de n * sizeof(double)
xor eax, eax             ; i = 0
lea rsi, [X]             ; X doit être aligné par 16
lea rdi, [Y]             ; Y doit être aligné par 16
movsd xmm2, [DA]         ; Chargement de DA
shufpd xmm2, xmm2, 0     ; Intégration de DA dans les deux qwords de xmm2

; Cette boucle effectue deux calculs DAXPY par itération, en utilisant des vecteurs :
L1: movapd xmm1, [rsi+rax] ; X[i], X[i+1]
mulpd xmm1, xmm2          ; X[i] * DA, X[i+1] * DA
movapd xmm0, [rdi+rax]    ; Y[i], Y[i+1]
subpd xmm0, xmm1          ; Y[i]-X[i]*DA, Y[i+1]-X[i+1]*DA
movapd [rdi+rax], xmm0    ; Enregistrement du résultat
add eax, 16               ; Ajout de la taille de deux éléments à l'index
cmp eax, ecx              ; Comparaison avec n*8
jl L1                     ; Boucle arrière
```

Analysons maintenant ce code pour identifier les goulots d'étranglement sur un processeur Pentium M, en supposant qu'il n'y ait aucun défaut de cache. Les détails spécifiques au processeur auxquels je fais référence sont expliqués dans le manuel 3 : “La microarchitecture des processeurs Intel, AMD et VIA”.

Nous ne nous intéressons qu'à la boucle, c'est-à-dire au code situé après l'étiquette L1. Il nous faut lister la décomposition en micro-opérations (μops) de toutes les instructions de la boucle, en utilisant le tableau du manuel 4 : “Tableaux d'instructions”. Cette liste se présente comme suit :

Instruction	μops fusionnées	ports d'exécution					unités d'exécution	
		port 0/1	port 0/1/5/6	port 6	port 2/3/7	port 4	FADD	FMUL
movapd xmm1, [rsi+rax]	1				1			
mulpd xmm1, xmm2	1	1						1
movapd xmm0, [rdi+rax]	1				1			
subpd xmm0, xmm1	1	1					1	
movapd [rdi+rax], xmm0	1				1	1		
add eax, 16	1		1					
cmp eax, ecx	1		1					
jl L1	1			1				
Total	8	2	2	1	3	1	1	1

Le nombre total de μops fusionnées destinées à l'ensemble des ports est de 8. Le débit global est de quatre μops par cycle d'horloge. Cela implique un temps d'exécution minimal de 2 cycles d'horloge par itération.

Les ports arithmétiques 0, 1, 5 et 6 reçoivent 5 μops par itération à répartir entre ces 4 ports, ce qui fixe une limite de 1,25 cycle d'horloge par itération.

Les ports de calcul d'adresse et d'entrée 2, 3 et 7 reçoivent 3 μops par itération, soit une limite de 1 cycle par itération. Le port d'écriture (port 4) reçoit un μop.

L'unité de calcul en virgule flottante (FADD/FMUL) reçoit 2 μops ; un débit de 2 μops par cycle d'horloge correspond à une limite de 2 cycles par itération.

Le temps nécessaire à la récupération des instructions peut être calculé à partir de la taille de ces dernières. La taille totale des huit instructions de la boucle est de 30 octets. Le processeur doit récupérer deux blocs de 16 octets si le point d'entrée de la boucle est aligné sur 16 octets, ou au maximum trois blocs de 16 octets dans le pire des cas. La récupération des instructions ne constitue donc pas un goulot d'étranglement.

La conclusion est que le goulot d'étranglement limitant réside dans les décodeurs ou le cache de micro-opérations (μop), qui sont limités à 4 μops fusionnées par cycle d'horloge.

Il existe une chaîne de dépendance au sein de la boucle. Les latences sont les suivantes : 2 pour la lecture en mémoire, 4 pour la multiplication, 4 pour la soustraction et 3 pour l'écriture en mémoire, soit un total de 13 cycles d'horloge. C'est trois fois supérieur au temps de validation (*retirement*), mais il ne s'agit pas d'une dépendance portée par la boucle (*loop-carried dependence*), car les résultats de chaque itération sont enregistrés en mémoire et ne sont pas réutilisés lors de l'itération suivante. Le mécanisme d'exécution dans le désordre (*out-of-*

order) et le pipeline permettent à chaque calcul de débiter avant que le précédent ne soit terminé. La seule chaîne de dépendance portée par la boucle est l'instruction `add eax, 16`, dont la latence n'est que de 1.

La durée de 2 cycles par itération constitue un minimum théorique. Le temps d'exécution mesuré sera souvent plus long en raison d'une planification dans le désordre sous-optimale, d'une répartition sous-optimale des μ ops entre les ports d'exécution et de contentions au niveau du cache. Le temps d'exécution sera évidemment bien plus long en cas de défauts de cache (*cache misses*).

Il est possible de réduire le nombre d'instructions dans la boucle en utilisant un index négatif partant de la fin des tableaux. Cela peut améliorer la vitesse de calcul car cela réduit le nombre de μ ops :

```
; Exemple 12.6c. Boucle de l'algorithme DAXPY avec des indices négatifs
section .data
align 16
SignBit DD 0, 80000000H      ; mot quadruple avec bit de signe positionné
n equ 100                    ; Définition de la constante n (paire et positive)

section .text
default rel
    mov eax, n * 8           ; Taille = n * sizeof(double)
    lea rsi, [X]              ; Adresse du tableau X (aligné)
    lea rdi, [Y]              ; Adresse du tableau Y (aligné)
    add rsi, rax               ; pointe vers la fin du tableau X
    add rdi, rax               ; pointe vers la fin du tableau Y
    neg rax                    ; index négatif
    movsd xmm2, [DA]          ; Chargement de DA
    xorpd xmm2, [SignBit]     ; Changement de signe
    shufpd xmm2, xmm2, 0      ; Intégration de -DA dans les deux qwords de xmm2
L1:  movapd xmm1, [rsi+rax]    ; X[i], X[i+1]
     mulpd xmm1, xmm2          ; X[i] * (-DA), X[i+1] * (-DA)
     addpd xmm1, [rdi+rax]     ; Y[i]-X[i]*DA, Y[i+1]-X[i+1]*DA
     movapd [rdi+rax], xmm1    ; Enregistrement du résultat
     add rax, 16               ; Ajout de la taille de deux éléments à l'index
     js L1                    ; Boucle arrière
```

Cela supprime deux instructions et deux μ ops de la boucle, de sorte que le minimum théorique est de 1,5 cycle d'horloge par itération.

12.9 Même exemple avec FMA3

Les processeurs Intel Haswell et ultérieurs prennent en charge la forme à trois opérandes des instructions de multiplication-accumulation fusionnée, appelée FMA3.

```
; Exemple 12.6d. Boucle DAXPY avec FMA3, utilisant des registres XMM
section .text
default rel
    mov eax, n * 8           ; Taille = n * sizeof(double)
    lea rsi, [X]              ; Adresse du tableau X (aligné)
    lea rdi, [Y]              ; Adresse du tableau Y (aligné)
    add rsi, rax               ; pointe vers la fin du tableau X
    add rdi, rax               ; pointe vers la fin du tableau Y
    neg rax                    ; index négatif
    vmovddup xmm2, [DA]       ; Chargement de DA x 2
L1:  vmovapd xmm1, [rdi+rax]   ; Y[i]
     vfmadd231pd xmm1, xmm2, [rsi+rax] ; Y[i]-X[i]*DA
     vmovapd [rdi+rax], xmm1  ; Enregistrement
     add rax, 16               ; Ajout de la taille de deux éléments à l'index
     jl L1                    ; Boucle arrière
```

Cette boucle nécessite 1 à 2 cycles d'horloge par itération sur les processeurs dotés de FMA3. Il peut être utile d'aligner le point d'entrée de la boucle, comme expliqué dans l'[exemple 11.4](#).

12.10 Même exemple avec AVX512

Vous pouvez doubler le débit en utilisant les registres YMM à l'aide du jeu d'instructions AVX, ou vous pouvez quadrupler le débit en utilisant les registres ZMM avec le jeu d'instructions AVX512 :

```
; Exemple 12.6e. Boucle DAXPY avec AVX-512, utilisant des registres ZMM
section .text
```

```

default rel
n equ 128                                ; n doit être divisible par 8
mov eax, n * 8                          ; Taille = n * sizeof(double)
lea rsi, [X]                            ; Adresse du tableau X (aligné)
lea rdi, [Y]                            ; Adresse du tableau Y (aligné)
add rsi, rax                            ; pointe vers la fin du tableau X
add rdi, rax                            ; pointe vers la fin du tableau Y
neg rax                                ; index négatif
vbroadcastsd zmm2, [DA]                ; Chargement de DA x 8
L1: vmovupd zmm1, [rdi+rax]             ; Y[i]
vfmadd231pd zmm1, zmm2, [rsi+rax]      ; Y[i]-X[i]*DA
vmovupd [rdi+rax], zmm1               ; Enregistrement
add rax, 64                           ; Ajout de la taille de 8 éléments à l'index
jl L1                                  ; Boucle arrière

```

L'exemple 12.6e ne suppose pas que les tableaux sont alignés sur une frontière de 64 octets. Par conséquent, l'instruction `VMOVPD` a été remplacée par `VMOVUPD`. Le temps d'exécution de cette boucle a été mesuré à 1,5 cycle d'horloge sur l'architecture Skylake. Ici, le goulot d'étranglement ne réside pas dans les unités d'exécution, mais dans les effets liés à la mémoire cache et, peut-être, dans l'instruction de branchement. On suppose que la taille du tableau, n , est divisible par 8, car nous effectuons 8 opérations par cycle d'horloge. Voir la section suivante pour savoir comment procéder lorsque n n'est pas divisible par la taille du vecteur.

12.11 Déroulage de boucle

Une boucle effectuant n répétitions peut être remplacée par une boucle qui se répète n/r fois et effectue r calculs à chaque répétition, où r est le facteur de déroulement. Il est préférable que n soit divisible par r .

Le déroulement de boucle peut être utilisé aux fins suivantes :

- Réduction de la surcharge liée à la boucle. La surcharge par calcul est divisée par le facteur de déroulage r . Cela n'est utile que si cette surcharge contribue de manière significative au temps de calcul. Il n'y a aucune raison de dérouler une boucle si un autre goulot d'étranglement limite la vitesse d'exécution. Par exemple, la boucle de l'exemple 12.6e ci-dessus ne tirerait aucun avantage d'un déroulage supplémentaire.
- Vectorisation. Une boucle doit être déroulée selon un facteur r (ou un multiple de r) pour permettre l'utilisation de registres vectoriels contenant r éléments. La boucle de l'exemple 12.6e est déroulée par un facteur de 8 afin d'utiliser des vecteurs de huit nombres en double précision. Si nous avions utilisé des nombres en simple précision, nous aurions pu dérouler la boucle par un facteur de 16 et utiliser des vecteurs de 16 éléments.
- Amélioration de la prédiction de branchement. La prédiction du branchement de sortie de boucle peut être améliorée en déroulant la boucle de telle sorte que le nombre de répétitions n/r ne dépasse pas la limite maximale prédictible par le processeur concerné. Toutefois, la pénalité liée à une mauvaise prédiction de branchement sera probablement masquée par le fait que, dans un processeur à exécution dans le désordre (*out-of-order*), le compteur de boucle est calculé bien en amont du corps de la boucle.
- Amélioration de l'utilisation du cache. Si la boucle génère de nombreux défauts de cache de données ou des contentions au niveau du cache, il peut être avantageux d'ordonnancer les lectures et écritures mémoire de manière optimale pour le processeur spécifique. Cela s'avère rarement nécessaire sur les processeurs modernes. Consultez le manuel d'optimisation fourni par le fabricant du microprocesseur pour plus de détails.
- Élimination des divisions entières. Si la boucle contient une expression impliquant la division du compteur i par un entier r ou le calcul du modulo de i par r , il est possible d'éviter la division entière en déroulant la boucle selon le facteur r .
- Élimination des branchements internes à la boucle. Si la boucle contient un branchement ou une instruction `switch` suivant un schéma répétitif de période r , ce branchement peut être éliminé en déroulant la boucle selon le facteur r . Par exemple, si une structure `if-else` alterne entre ses deux branches une fois sur deux, ce branchement peut être supprimé en déroulant la boucle par un facteur de deux.
- Rupture de la chaîne de dépendance inter-itérations. Dans certains cas, il est possible de rompre une chaîne de dépendance inter-itérations en utilisant plusieurs accumulateurs. Le facteur de déroulage r correspond alors au nombre d'accumulateurs utilisés. Voir l'[exemple 9.3b](#) à la page 50.
- Réduire la dépendance d'une variable d'induction. Si le délai de calcul d'une variable d'induction à partir de sa valeur lors de l'itération précédente est tel qu'il crée un goulot d'étranglement, il est possible de résoudre ce problème en effectuant un déroulage de facteur r et en calculant chaque valeur de la variable d'induction à partir de la valeur située r rangs en arrière dans la séquence.
- Déroulage complet. Une boucle est entièrement déroulée lorsque $r = n$, où n est une constante connue. Cela supprime totalement la surcharge liée à la gestion de la boucle. Toute expression fonction du comp-

teur de boucle peut être remplacée par des constantes. Tout branchement ne dépendant que du compteur de boucle peut être éliminé. Voir la page 82 pour des exemples.

Le déroulage de boucle présente également des inconvénients importants. Il ne doit être utilisé que lorsqu'il existe une bonne raison de le faire et qu'un gain de vitesse significatif peut être obtenu. Il convient d'éviter un déroulage excessif des boucles. Les inconvénients du déroulage de boucle sont les suivants :

- Les microprocesseurs modernes peuvent exécuter quatre instructions indépendantes ou plus par cycle d'horloge. La surcharge liée à la boucle comprend généralement un compteur, une comparaison et un saut conditionnel. Ces instructions peuvent s'exécuter simultanément avec le corps de la boucle, de sorte que cette surcharge n'ajoute aucun cycle d'horloge supplémentaire au temps d'exécution. Le microprocesseur peut même exécuter plusieurs itérations de la boucle en parallèle. Dans ce cas, le déroulage de boucle a peu de chances d'améliorer significativement les performances.
- Le code s'agrandit et occupe davantage d'espace dans le cache d'instructions. Cela peut entraîner des défauts de cache d'instructions dont le coût dépasse le gain obtenu par le déroulage. Il est à noter que ces défauts de cache ne sont pas détectés lorsque la boucle est testée de manière isolée.
- Certains processeurs disposent d'un tampon de boucle (*loopback buffer*) pour accélérer les très petites boucles. La capacité de ce tampon est limitée à 20-50 instructions ou 64 octets de code, selon le processeur. Le déroulage de boucle risque de réduire les performances s'il fait dépasser la taille de ce tampon. Des détails spécifiques aux processeurs sont fournis dans le manuel 3 : "The microarchitecture of Intel, AMD and VIA CPUs".
- De nombreux processeurs modernes possèdent un cache de micro-opérations (*cache μ op*) de taille limitée (voir [section 11.3](#)). Ce cache est si précieux qu'il convient d'en optimiser l'utilisation. Les boucles déroulées occupent plus d'espace dans le cache μ op. Notez que les défauts de cache μ op ne sont pas toujours détectés lors d'un test isolé de la boucle.
- Une boucle déroulée pour améliorer la mise en cache des données sur un microprocesseur donné peut ne pas être optimale lorsqu'elle s'exécute sur un autre processeur doté d'une structure de cache différente.
- La nécessité d'effectuer des calculs supplémentaires en dehors de la boucle déroulée (lorsque n n'est pas divisible par r) rend le code plus complexe et plus lourd, tout en augmentant le nombre de branchements, comme expliqué ci-dessous.
- La boucle déroulée peut nécessiter davantage de registres, par exemple pour gérer plusieurs accumulateurs.

Le déroulage de boucle pose un problème particulier lorsque le nombre de répétitions n n'est pas divisible par le facteur de déroulage r . Il subsiste alors un reste de n modulo r correspondant à des calculs supplémentaires qui ne sont pas effectués au sein de la boucle. Ces calculs doivent être réalisés soit avant, soit après la boucle principale.

Il peut s'avérer délicat de gérer correctement ces calculs supplémentaires lorsque le nombre de répétitions n'est pas divisible par le facteur de déroulage. La difficulté est accrue si l'on utilise un indice négatif, comme dans les exemples 12.6c, d et e. L'exemple suivant présente à nouveau l'algorithme DAXPY, cette fois déroulé par un facteur de 4 et utilisant AVX2 ; ici, n est une variable qui peut être divisible ou non par 4.

; **Exemple 12.7.** Boucle DAXPY déroulée, simple précision.

```
section .text
default rel
    mov eax, [n]                ; Taille du tableau
    shl eax, 3                  ; Taille du tableau en octets
    sub rax, 32                  ; On ignore tout reste.
    lea rsi, [X]                 ; Adresse du tableau X
    lea rdi, [Y]                 ; Adresse du tableau Y
    add rsi, rax                  ; pointe vers la fin du tableau X - 4*8
    add rdi, rax                  ; pointe vers la fin du tableau Y - 4*8
    neg rax                      ; index négatif
    vbroadcastsd ymm2, [DA]      ; Chargement de DA x 4
    jg L2                        ; Ignorer la boucle principale si n < 4

L1:                               ; boucle principale traitant tous les éléments
                                ; sauf le reste
    vmovupd ymm1, [rdi+rax]      ; Y[i]
    vfmadd231pd ymm1, ymm2, [rsi+rax] ; Y[i]-X[i]*DA
    vmovupd [rdi+rax], ymm1      ; Enregistrement
    add rax, 32                  ; Quatre éléments par itération
    jl L1                        ; Boucle arrière

L2:                               ; Vérifiez s'il reste des éléments.
```

```

sub rax, 32          ; = -reste
jns L4              ; Ignorer en cas de reste = 0
L3:                 ; Boucle supplémentaire pour les (n % 4)
                   ; itérations restantes
vmovsd xmm1, [rdi+rax+32] ; Y[i]
vfmadd231sd xmm1, xmm2, [rsi+rax+32] ; Y[i]-X[i]*DA
vmovsd [rdi+rax+32], xmm1 ; Enregistrement
add rax, 8           ; Un élément par itération
jl L3               ; Boucler tant que la valeur est négative
L4:

```

Une solution alternative pour une boucle déroulée effectuant des calculs sur des tableaux consiste à étendre ces derniers en y ajoutant jusqu'à $r-1$ emplacements inutilisés et à arrondir le nombre d'itérations n au multiple du facteur de déroulage r immédiatement supérieur. Cela supprime la nécessité de calculer le reste (n modulo r) ainsi que celle d'une boucle supplémentaire pour traiter les itérations restantes. Les éléments inutilisés des tableaux doivent être initialisés à zéro ou à une autre valeur en virgule flottante valide afin d'éviter l'apparition de nombres dénormalisés, de valeurs NaN, de dépassements de capacité (supérieurs ou inférieurs) ou de toute autre condition susceptible de ralentir les calculs en virgule flottante. Si les tableaux contiennent des entiers, la seule condition à éviter est la division par zéro.

En résumé : le déroulage de boucle présente des avantages et des inconvénients. Il ne faut dérouler une boucle que si les avantages l'emportent clairement sur les inconvénients. Une erreur courante consiste à mesurer les performances d'une boucle déroulée de manière isolée, sans prendre en compte la concurrence pour l'accès au cache d'instructions ou au cache de micro-opérations (*μop cache*) avec d'autres parties du programme. Certains compilateurs déroulent les boucles de manière excessive sans tenir compte des inconvénients ; il s'agit peut-être d'un vestige d'une époque où le surcoût lié à la gestion des boucles était plus élevé.

12.12 Boucles vectorielles utilisant des registres de masque (AVX-512)

Le jeu d'instructions AVX512 propose des opérations masquées. Cette fonctionnalité est utile pour masquer les éléments vectoriels excédentaires lorsque le nombre de boucles n'est pas divisible par la taille du vecteur :

```

; Exemple 12.8. Boucle DAXPY vectorisée utilisant des registres de masque
section .data
align 64
countdown dd 7, 6, 5, 4, 3, 2, 1, 0
eight      dd 8
section .text
default rel
    lea rsi, [X]          ; RSI pointe le début de X
    lea rdi, [Y]          ; RDI pointe le début de Y
    mov edx, [n]          ; nombre d'éléments, n
    vmovd xmm0, edx
    vpbroadcastd ymm0, xmm0 ; diffusion de n
    vpaddb ymm0, ymm0, [countdown] ; counts = n + countdown
    vpbroadcastd ymm1, [eight] ; diffusion de eight
    vbroadcastsd zmm2, [DA] ; diffusion de DA
    xor eax, eax          ; compteur de boucle i = 0
L1: ; Boucle déroulée 8 fois, éléments inutilisés masqués.
    vpcmpub k1, ymm0, ymm1, 5 ; mask k1 = (counts >= 8)
    vpsubd ymm0, ymm0, ymm1 ; counts -= 8
    vmovupd zmm3 {k1}{z}, [rdi+rax*8] ; Chargement de Y[i]
    vfmadd231pd zmm3 {k1}, zmm2, [rsi+rax*8] ; Y[i]-DA*X[i]
    vmovupd [rdi+rax*8] {k1}, zmm3 ; Y[i] masqué
    add rax, 8 ; 8 éléments par itération
    cmp rax, rdx ; Tant que i < n
    jb L1 ; poursuivre la boucle

```

Ici, le masque du registre `k1` comporte un bit à 1 pour tous les éléments valides et un bit à 0 pour les éléments excédentaires au-delà de n . Le nombre d'éléments restants est décompté dans `YMM0`.

Les instructions de chargement et de stockage sont masquées par `k1` afin d'éviter toute opération au-delà des n éléments de `Y`. L'instruction de chargement masquée utilise l'option de mise à zéro `{z}` pour éviter une fausse dépendance vis-à-vis de la valeur précédente du registre vectoriel `ZMM3`. Il est également judicieux d'appliquer le même masque aux instructions de calcul. Le masquage des calculs permet d'économiser de l'énergie et d'éviter

les exceptions ainsi que les pénalités liées aux éventuelles valeurs sous-normales, etc. L'ajout d'un masque à une instruction vectorielle de 512 bits n'entraîne aucun surcoût temporel.

Les instructions de décompte utilisées pour générer le masque doivent comporter le même nombre d'éléments par vecteur que les calculs, mais pas nécessairement le même nombre de bits par élément. Nous décomptons la valeur n dans le registre YMM0 sous forme de huit entiers non signés de 32 bits, tandis que les calculs DAXPY utilisent huit nombres à virgule flottante double précision de 64 bits dans le registre ZMM3.

La boucle de l'exemple 12.9 est dominée par des instructions vectorielles. Il est probable que les unités d'exécution vectorielle constituent ici un goulot d'étranglement. Il peut être préférable d'utiliser des instructions sur entiers plutôt que des instructions vectorielles pour générer le masque si les unités d'exécution dédiées à l'addition vectorielle représentent un facteur limitant. C'est ce qu'illustre l'exemple suivant :

- ; **Exemple 12.10.** DAXPY vectorisé, masque calculé à l'aide d'instructions sur entiers.
- ; Remarque : cette version ne fonctionne que pour $n < 256$!

```
section .text
default rel

    lea rsi, [X]                ; RSI pointe le début de X
    lea rdi, [Y]                ; RDI pointe le début de Y
    vbroadcastsd zmm2, [DA]     ; diffusion de DA
    mov edx, [n]                ; nombre d'éléments, n
    mov ecx, -1                 ; remplissage avec uniquement des 1
    xor eax, eax                ; compteur de boucle i = 0
L1:  ; Boucle déroulée 8 fois, éléments inutilisés masqués.
    bzhi ecx, ecx, edx          ; positions bit 0 > edx
    kmovw kl, ecx               ; copie des bits dans le registre de masque
    vmovupd zmm3 {kl}{z}, [rdi+rax*8] ; chargement de Y[i]
    vfmadd231pd zmm3 {kl}, zmm2, [rsi+rax*8] ; Y[i]-DA*X[I]
    vmovupd [rdi+rax*8] {kl}, zmm3 ; enregistrement de Y[i]
    add rax, 8                  ; 8 éléments par itération
    sub edx, 8                  ; décompte de n
    ja L1                       ; répétition si n est positif
```

L'exemple 12.10 décrémente le nombre d'éléments restants dans `edx` et utilise `bzhi` pour mettre à zéro les bits restants dans `ecx` lors de la dernière itération si $edx < 16$. L'instruction `bzhi` appartient au jeu d'instructions BMI2, pris en charge par tous les processeurs connus dotés de l'AVX512. Notez que cela ne fonctionne que pour $n < 256$, car `bzhi` ne lit que les 8 bits de poids faible de `edx`. Nous utilisons l'instruction `kmovw` plutôt que `kmovb`, car cette dernière nécessite le jeu d'instructions AVX512DQ.

Si le nombre d'itérations est élevé et que les instructions de calcul du masque ralentissent l'exécution de la boucle, il vaut mieux avoir une boucle principale sans masquage et n'effectuer les opérations nécessitant un masque qu'après cette boucle principale :

- ; **Exemple 12.11.** DAXPY vectorisé, masquage uniquement après la boucle principale

```
section .text
default rel

    lea rsi, [X]                ; RSI pointe le début de X
    lea rdi, [Y]                ; RDI pointe le début de Y
    vbroadcastsd zmm2, [DA]     ; diffusion DA
    mov edx, [n]                ; nombre d'éléments, n
    and edx, -8                 ; arrondi de n à un multiple de 8 (par valeurs inf.)
    lea rsi, [rsi+rdx*8]        ; RSI pointe la fin de X
    lea rdi, [rdi+rdx*8]        ; RDI pointe la fin de Y
    neg rdx                     ; utilisation d'un indice négatif à partir de la fin
L1:  ; Boucle principale déroulée par 8
    vmovupd zmm3, [rdi+rdx*8]   ; chargement de Y[i]
    vfmadd231pd zmm3, zmm2, [rsi+rdx*8] ; Y[i]-DA*X[I]
    vmovupd [rdi+rdx*8], zmm3   ; enregistrement de Y[i]
    add rdx, 8                  ; 8 éléments suivants
    jnz L1                      ; comptage jusqu'à zéro
    mov edx, [n]                ; Calcul des éléments restants
    and edx, 7                  ; n modulo 8
    ; Omission de la ligne suivante si la branche L2 est difficile à prédire.
    jz L2                       ; ignorer éventuellement si zéro
    mov eax, -1                 ; tous les bits à 1
```

```

    bzhi eax, eax, edx          ; positions de bits à zéro > edx
    kmovw kl, eax              ; copie des bits dans le registre de masque
; RSI et RDI pointent vers l'endroit où la boucle principale s'est terminée.
; Traitement, maintenant, des éléments 0 à 7 en utilisant le masque kl.
    vmovupd zmm3 {kl}{z}, [rdi] ; chargement de Y[i]
    vfmadd231pd zmm3 {kl}, zmm2, [rsi] ; Y[i]-DA*X[I]
    vmovupd [rdi] {kl}, zmm3    ; enregistrement de Y[i]

```

L2:

L'exemple 12.11 contient deux fois le corps de la boucle. Le premier corps est répété $n/8$ fois dans la boucle L1. Le second corps traite les éléments restants (n modulo 8) lorsque n n'est pas divisible par 8. Il n'est pas nécessaire de sauter le dernier corps à l'aide d'une instruction `jz L2` si ce branchement est difficile à prédire.

Le corps de la boucle de l'exemple 12.11 ne comporte que 5 instructions. Cette solution sera la plus rapide si le nombre d'itérations est élevé. Les exemples 12.8 et 12.10 sont utiles si le corps de la boucle est trop volumineux pour être dupliqué, si le nombre d'itérations est faible, ou si le goulot d'étranglement réside dans les accès mémoire ou dans une chaîne de dépendances entre itérations. L'exemple 12.10 ne peut être utilisé que si $n < 256$.

12.13 Optimisation de la mise en cache

Dans une boucle accédant à de la mémoire non mise en cache, l'accès à la mémoire est généralement l'opération la plus coûteuse en temps. Dans la mesure du possible, les données doivent être stockées de manière contiguë et faire l'objet d'un accès séquentiel, comme expliqué au chapitre 11 (page 64).

Le nombre de tableaux auxquels on accède au sein d'une boucle ne doit pas dépasser le nombre de tampons de lecture/écriture du microprocesseur. Une méthode pour réduire le nombre de flux de données consiste à regrouper plusieurs tableaux en un tableau de structures, de sorte que les différents flux soient entrelacés en un flux unique.

Les microprocesseurs modernes disposent de mécanismes avancés de préchargement (*prefetching*) des données. Ces mécanismes sont capables de détecter des régularités dans les schémas d'accès aux données, comme un accès avec un pas (*stride*) constant. Il est recommandé de tirer parti de ces mécanismes en limitant au maximum le nombre de flux de données distincts et en maintenant, si possible, un pas d'accès constant. Le préchargement automatique s'avère souvent plus efficace que le préchargement explicite lorsque le schéma d'accès aux données est suffisamment régulier.

Un préchargement explicite, au moyen d'instructions `prefetch`, peut s'avérer nécessaire lorsque le schéma d'accès aux données est trop irrégulier pour être anticipé par les mécanismes automatiques. La détermination de la stratégie de préchargement optimale pour un programme accédant aux données de manière irrégulière nécessite souvent une phase d'expérimentation approfondie.

Il est possible de placer la prélecture (*prefetching*) des données dans un thread distinct si le microprocesseur est capable d'exécuter deux threads par cœur de processeur. Le compilateur Intel C++ propose une fonctionnalité permettant de le faire.

L'accès aux données avec un pas (*stride*) correspondant à une puissance de 2 élevée risque de provoquer des contentions au niveau des lignes de cache. Ce problème peut être évité en modifiant le pas ou en utilisant le découpage de boucle (*loop blocking*). Pour plus de détails, consultez le chapitre consacré à l'optimisation des accès mémoire dans le manuel 1 : "Optimizing software in C++".

Les instructions d'écriture non temporelles sont utiles pour écrire dans une mémoire non mise en cache qui ne sera probablement pas sollicitée à nouveau prochainement. Vous pouvez utiliser des instructions vectorielles afin de réduire le nombre d'instructions d'écriture non temporelles.

12.14 Parallélisation

Le moyen le plus important d'améliorer les performances d'un code sollicitant fortement le processeur consiste à effectuer les tâches en parallèle. Les principales méthodes pour y parvenir sont les suivantes :

- Amélioration des possibilités du CPU d'effectuer une exécution dans le désordre. Cela se fait en brisant les longues chaînes de dépendances (voir page 50) et en répartissant les μ ops uniformément entre les différentes unités d'exécution ou ports d'exécution (voir page 73).
- Utilisation des instructions vectorielles. Voir chapitre 13 page 84.
- Utilisation de plusieurs threads. Voir chapitre 14 page 106.

Il est possible de rompre les chaînes de dépendance inter-itérations en utilisant plusieurs accumulateurs, comme expliqué à la page 50. Le nombre maximal d'accumulateurs utile correspond à la latence de l'instruction la plus critique de la chaîne de dépendance divisée par l'inverse du débit de cette instruction. Par exemple, si la latence d'une addition en virgule flottante est de 4 cycles d'horloge et que l'inverse du débit est de 1, le nombre maximal d'accumulateurs utile est alors de 4. Un nombre d'accumulateurs inférieur à ce maximum peut suffire à garantir que la chaîne de dépendance inter-itérations ne constitue pas un facteur limitant.

```
// Exemple 12.12a. Chaîne de dépendances inter-itérations.
// Code C sans optimisation
// (Identique à l'exemple 9.3a, page 50)
double list[100], sum = 0.0;
for (int i = 0; i < 100; i++) sum += list[i];
```

L'exemple 12.12b présente une boucle qui additionne cent nombres en utilisant trois registres vectoriels AVX2 comme accumulateurs.

; **Exemple 12.12b.** Trois accumulateurs vectoriels

```
section .text
default rel

; Addition de 100 nombres en utilisant trois accumulateurs de 4 nombres chacun.
    lea rsi, [list]                ; Pointeur vers la liste
    vmovapd ymm0, [rsi]            ; chargement des 4 premiers éléments
    vmovapd ymm1, [rsi+32]         ; chargement des 4 éléments suivants
    vmovapd ymm2, [rsi+64]         ; chargement des 4 éléments suivants
    add rsi, 96                    ; incrémentation du pointeur
    mov ecx, 7                     ; répétition de la boucle 7 fois

L1:
    vaddpd ymm0, ymm0, [rsi]       ; addition de 4 éléments
    vaddpd ymm1, ymm1, [rsi+32]    ; addition de 4 éléments
    vaddpd ymm2, ymm2, [rsi+64]    ; addition de 4 éléments
    add rsi, 96
    dec ecx
    jnz L1                         ; boucle
    vaddpd ymm0, ymm0, [rsi]       ; addition des 4 derniers éléments
    vaddpd ymm1, ymm1, ymm2        ; combinaison de 2 accumulateurs
    vaddpd ymm0, ymm0, ymm1        ; combinaison des 2 derniers accumulateurs

; calcule de la somme des 4 éléments de ymm0 :
    vextractf128 xmm1, ymm0, 1     ; Placement de la moitié supérieure de ymm0 dans xmm1
    vaddpd xmm0, xmm1, xmm1        ; Combinaison de 4 valeurs en deux

; on évite l'instruction vhaddpd, qui est lente
    vunpckhpd xmm1, xmm0, xmm0     ; obtention de la valeur supérieure
    vaddsd xmm0, xmm1              ; combinaison des 2 dernières valeur

; The sum is now in xmm0
```

Dans l'exemple 12.12b, j'ai chargé les trois registres vectoriels avec les 3×4 premières valeurs de la liste. La boucle effectue 12 additions par itération. Nous devons réaliser quatre additions supplémentaires après la boucle, car 100 n'est pas divisible par 12. Les trois accumulateurs, contenant chacun quatre sommes partielles, sont additionnés pour former un vecteur de quatre valeurs après la boucle. Enfin, ces quatre valeurs sont additionnées pour obtenir la somme finale.

La boucle de l'exemple 12.12b comporte quatre chaînes de dépendance inter-itérations s'exécutant en parallèle : ymm0, ymm1, ymm2 et ecx. La vitesse est limitée par la latence de l'instruction vaddpd, qui est supérieure à celle de l'instruction dec ecx. Il serait peut-être possible d'améliorer encore la vitesse en utilisant davantage d'accumulateurs, mais d'autres facteurs risqueraient de ralentir l'exécution à mesure que l'on atteint la limite de débit de l'unité d'addition en virgule flottante. Le nombre optimal d'accumulateurs est souvent inférieur au maximum théorique. Avec seulement 7 itérations dans cet exemple, l'ajout d'accumulateurs supplémentaires ne justifie probablement pas le code additionnel requis. Il n'est pas nécessaire d'utiliser des indices négatifs dans cette boucle, car la surcharge liée à la boucle elle-même ne constitue pas un goulot d'étranglement limitant.

La situation se complique si l'on souhaite utiliser la précision "long double" afin de réduire l'accumulation des erreurs d'arrondi. Cette précision n'est disponible qu'avec les anciens registres en virgule flottante de type x87. Ces registres sont organisés sous forme de pile tournante. Il est nécessaire d'utiliser des instructions fxch pour placer le registre souhaité au sommet de la pile.

; **Exemple 12.12c.** Quatre accumulateurs en précision double étendue (long double)

```
section .text
default rel

; Addition de 100 nombres à l'aide de quatre accumulateurs en double précision étendue.
    lea rsi, [list]                ; Pointeur vers la liste
    fld qword [rsi]                ; accum1 = list[0]
    fld qword [rsi+8]              ; accum2 = list[1]
    fld qword [rsi+16]              ; accum3 = list[2]
    fld qword [rsi+24]              ; accum4 = list[3]
```

```

    fxch st3                ; Amener accum1 au sommet
    mov eax, 32              ; Index de list[4]
    mov ecx, 32              ; Nombre de repetitions du comptage
L1:
    fadd qword [rsi+rax]     ; Addition de list[i]
    fxch st1                ; Permutation des accumulateurs
    fadd qword [rsi+rax+8]   ; Addition de list[i+1]
    fxch st2                ; Permutation des accumulateurs
    fadd qword [rsi+rax+16]  ; Addition de list[i+2]
    fxch st3                ; Permutation des accumulateurs
    add eax, 24              ; i += 3
    dec ecx
    jnz L1                  ; Boucle
    faddp st1, st0           ; Addition de 2 accumulateurs.
    fxch st1                ; Permutation des accumulateurs
    faddp st2, st0           ; Addition des 2 autres accumulateurs.
    faddp st1, st0           ; Additionnez de ces sommes.
    ; La somme finale est maintenant en st0

```

Dans l'exemple 12.12c, nous avons chargé les quatre accumulateurs avec les quatre premières valeurs de la liste. Une particularité de l'utilisation des registres x87 comme accumulateurs est que leur nombre correspond au facteur de déroulement *augmenté de un*. Cela découle de la manière dont les instructions `fxch` sont employées pour permuter les accumulateurs. Il faut se mettre à la place du processeur et suivre la position de chaque accumulateur dans la pile de registres à virgule flottante pour vérifier qu'ils subissent bien une rotation d'une position après chaque itération de la boucle ; ainsi, chaque accumulateur est utilisé pour une addition sur quatre, bien que la boucle ne soit déroulée que de trois itérations.

12.15 Boucles de macros

Si le nombre d'itérations d'une boucle est faible et constant, il est possible de dérouler entièrement la boucle. L'avantage est que les calculs ne dépendant que du compteur de boucle peuvent être effectués lors de l'assemblage plutôt qu'à l'exécution. L'inconvénient est, bien entendu, que cela occupe davantage d'espace dans le cache d'instructions si le nombre de répétitions est élevé.

La syntaxe de NASM inclut un langage de macros qui peut s'avérer très utile. D'autres assembleurs offrent des fonctionnalités similaires, mais avec une syntaxe différente.

Si, par exemple, nous avons besoin d'une liste de carrés parfaits, le code C++ pourrait ressembler à ceci :

```

// Exemple 12.13a. Boucle permettant de créer une liste de carrés
int squares[10];
for (int i = 0; i < 10; i++) squares[i] = i*i;

```

La même liste peut être générée par une boucle de macro en syntaxe NASM :

```

; Exemple 12.13b. Boucle de macro pour générer des données
section .data
    squares:                ; étiquette au début du tableau
    %assign i 0              ; compteur de boucle temporaire
    %rep 10                  ; répétition 10 fois
    DD i*i                  ; définition d'un élément de tableau
    %assign i i+1            ; incrémentation du compteur de boucle
    %endrep                  ; fin de la boucle de répétition

```

Ici, *i* est une variable de préprocesseur. La boucle sur *i* est exécutée lors de l'assemblage, et non lors de l'exécution. La variable *i* et l'instruction `%assign i i+1` n'apparaissent jamais dans le code final et ne nécessitent donc aucun temps d'exécution. En fait, l'exemple 12.13b ne génère aucun code exécutable, uniquement des données. Le préprocesseur de macros traduira le code ci-dessus comme suit :

```

; Exemple 12.13c. Résultats du développement de la macro-boucle
squares :                ; étiquette du début du tableau
    DD 0
    DD 1
    DD 4
    DD 9
    DD 16
    DD 25
    DD 36
    DD 49

```

DD 64
DD 81

Les boucles de macro sont également utiles pour générer du code. L'exemple suivant calcule x^n , où x est un nombre à virgule flottante et n est un entier positif. Cela se fait le plus efficacement en mettant x au carré à plusieurs reprises et en multipliant ensemble les facteurs qui correspondent aux chiffres binaires de n . L'algorithme peut être exprimé par le code C++ suivant :

```
// Exemple 12.14a. Calcul de pow(x, n) où n est un entier positif.
double x, xp, power;
unsigned int n, i;
xp = x; power = 1.0;
for (i = n; i != 0; i >>= 1) {
    if (i & 1) power *= xp;
    xp *= xp;
}
```

Si n est connu au moment de l'assemblage, la fonction de puissance peut être implémentée à l'aide de la boucle de macro suivante :

```
; Exemple 12.14b.
; La macro INTPOWER calcule la puissance entière N d'une valeur en virgule
; flottante double précision X, en utilisant le jeu d'instructions SSE2.
; INTPOWER possède trois paramètres :
; X : registre XMM utilisé en entrée. Ce registre n'est pas préservé.
; Y : registre XMM utilisé pour le résultat. Doit être différent de X.
; N : une constante entière positive.
; Le résultat sera Y = pow(X, N).

%macro INTPOWER 3
; variables temporaires locales :
; IPI : Utilisé pour le décalage N
; UTILISE : Mémoriser si Y contient des données
    %assign IPI %3          ; IPI = N
    %assign YUSED 0        ; n'oubliez pas de vérifier si Y contient
                           ; des données valides.
    %rep 32                ; le nombre maximal de répétitions est de 32.
        %if IPI & 1        ; test du bit 0 de l'IPI
            %if YUSED      ; si Y contient déjà des données
                mulsd %2, %1 ; multiplication de Y par une puissance de X
            %else          ; si c'est la première fois que Y est utilisé :
                movsd %2, %1 ; copie des données vers Y
                %assign YUSED 1 ; n'oubliez pas que Y contient désormais des données.
            %endif
        %endif
        %assign IPI (IPI >> 1) ; décalage de IPI d'une position vers la droite
        %if IPI == 0          ; arrêt lorsque IPI = 0
            %exitrep          ; sortie prématurée de la boucle REP 32
        %else
            mulsd %1, %1      ; elevation de X au carré
        %endif
    %endrep                ; fin de la boucle REP 32
%endmacro                  ; fin de la définition de la macro INTPOWER

; s'il s'agit du jeu d'instructions AVX :
; changement de mulsd %2, %1 par vmulsd %2, %2, %1
; changement de movsd %2, %1 par vmovsd %2, %1
```

Cette macro génère le nombre minimum d'instructions nécessaires pour effectuer le travail. Il n'y a pas de surcharge de boucle, de prologue ou d'épilogue dans le code final. Et surtout, pas de branchements. Toutes les branches ont été résolues par le préprocesseur de macro. Pour calculer XMM0 à la puissance 12, vous devez écrire :

```
; Exemple 12.14c. Appel de macro
INTPOWER xmm0, xmm1, 12
```

Cela sera étendu à :

```
; Exemple 12.14d. Résultat de l'expansion de la macro
```

```

mulsd xmm0, xmm0 ; x^2
mulsd xmm0, xmm0 ; x^4
movsd xmm1, xmm0 ; sauvegarde x^4
mulsd xmm0, xmm0 ; x^8
mulsd xmm1, xmm0 ; x^4 * x^8 = x^12

```

Cela comporte même moins d'instructions qu'une boucle en assembleur optimisée sans déroulage. La macro peut également fonctionner sur des vecteurs si l'on remplace `mulsd` par `mulpd` et `movsd` par `movapd`.

13 Programmation vectorielle

Il existe des limites physiques et technologiques à la fréquence d'horloge maximale des microprocesseurs. Par conséquent, la tendance est à l'augmentation du débit de traitement en traitant plusieurs données en parallèle.

Lors de l'optimisation du code, il est important de déterminer si certaines données peuvent être traitées en parallèle. Le principe de la programmation SIMD (*Single-Instruction-Multiple-Data*) consiste à regrouper un vecteur ou un ensemble de données dans un registre de grande taille et à les traiter simultanément en une seule opération. Il existe plus d'un millier d'instructions SIMD différentes. Ces instructions sont répertoriées dans les manuels “Intel 64 and IA-32 Architectures Software Developer's Manual” et “AMD64 Architecture Programmer's Manual”.

Plusieurs données peuvent être regroupées dans des registres MMX de 64 bits, XMM de 128 bits, YMM de 256 bits ou ZMM de 512 bits, selon les modalités suivantes :

type de données	données par pack	taille de registre	jeu d'instructions
8 bit integer	8	64 bit (MMX)	MMX
16 bit integer	4	64 bit (MMX)	MMX
32 bit integer	2	64 bit (MMX)	MMX
64 bit integer	1	64 bit (MMX)	SSE2
32 bit float	2	64 bit (MMX)	3DNow (obsolète)
8 bit integer	16	128 bit (XMM)	SSE2
16 bit integer	8	128 bit (XMM)	SSE2
32 bit integer	4	128 bit (XMM)	SSE2
64 bit integer	2	128 bit (XMM)	SSE2
16 bit float	8	512 bit (ZMM)	AVX512-FP16
32 bit float	4	128 bit (XMM)	SSE
64 bit float	2	128 bit (XMM)	SSE2
8 bit integer	32	256 bit (YMM)	AVX2
16 bit integer	16	256 bit (YMM)	AVX2
32 bit integer	8	256 bit (YMM)	AVX2
64 bit integer	4	256 bit (YMM)	AVX2
16 bit float	16	512 bit (ZMM)	AVX512-FP16
32 bit float	8	256 bit (YMM)	AVX
64 bit float	4	256 bit (YMM)	AVX
8 bit integer	64	512 bit (ZMM)	AVX-512BW
16 bit integer	32	512 bit (ZMM)	AVX-512BW
32 bit integer	16	512 bit (ZMM)	AVX-512
64 bit integer	8	512 bit (ZMM)	AVX-512
16 bit float	32	512 bit (ZMM)	AVX512-FP16
32 bit float	16	512 bit (ZMM)	AVX-512
64 bit float	8	512 bit (ZMM)	AVX-512

Tableau 13.1. Types vectoriels

Il est possible de déterminer si un microprocesseur donné prend en charge différents jeux d'instructions à l'aide de l'instruction `CPUID`, comme expliqué à la page 107. Les registres MMX 64 bits ne peuvent pas être utilisés conjointement avec les registres à virgule flottante de type `x87`. Les registres vectoriels ne peuvent être utilisés que s'ils sont pris en charge par le système d'exploitation. Tous les systèmes d'exploitation modernes prennent en charge les registres vectoriels (voir page 108 pour savoir comment vérifier si leur utilisation est activée par le système d'exploitation).

Il est avantageux de choisir la taille de données la plus petite adaptée à l'objectif visé, afin de pouvoir regrouper un maximum de données dans un seul registre vectoriel. Les calculs mathématiques peuvent nécessiter des nombres à virgule flottante en double précision (64 bits) pour éviter toute perte de précision lors des étapes intermédiaires, même si la simple précision suffit pour le résultat final.

Avant d'opter pour des instructions vectorielles, il convient de déterminer si le code résultant sera plus rapide que celui utilisant des instructions classiques (non vectorielles). Le code vectoriel consacre parfois plus d'instructions à des opérations annexes — telles que la conversion et le placement des données aux bons endroits dans les registres, ou l'émulation de branchements par des déplacements conditionnels — qu'aux calculs proprement dits. L'exemple 13.9 ci-dessous illustre ce cas de figure. Si les instructions vectorielles sont relativement lentes sur les processeurs anciens, de nombreux processeurs récents sont capables, dans bien des cas, d'effectuer un calcul vectoriel aussi rapidement qu'un calcul scalaire (sur une seule donnée).

Pour les calculs en virgule flottante, il est préférable d'utiliser des registres vectoriels plutôt que les anciens registres x87, même s'il n'existe aucune possibilité de gérer les données en parallèle. Les registres vectoriels sont gérés de manière plus simple que l'ancienne pile de registres x87, et certains processeurs ont de mauvaises performances pour les instructions x87.

Les opérandes mémoire pour les instructions XMM sans préfixe VEX doivent être alignés par 16. Les opérandes mémoire pour les instructions YMM sont de préférence alignés par 32 et ZMM par 64, mais cela n'est pas nécessaire. Voir page 67 pour savoir comment aligner les données en mémoire.

Les opérations arithmétiques et logiques les plus courantes peuvent être effectuées dans des registres vectoriels. L'exemple suivant illustre l'ajout de deux tableaux :

```
; Exemple 13.1a. Addition de deux tableaux à l'aide de vecteurs
; float a[128], b[128], c[128];
; for (int i = 0; i < 128; i++) a[i] = b[i] + c[i];
; a, b et c sont de préférence alignés sur 64.

xor ecx, ecx           ; Compteur de boucle i = 0
L: vmovups zmm0, [b+rcx] ; Chargement de 16 éléments depuis b
vaddps zmm0, zmm0, [c+rcx] ; Addition de 16 éléments de c
vmovups [a+rcx], zmm0   ; Stockage des 16 résultats en a
add ecx, 64             ; 16 éléments * 4 octets = 64
cmp ecx, 512            ; 128 éléments * 4 octets = 512
jb L                   ; Boucle
```

Il n'existe pas d'instruction pour la division entière. La division entière par un diviseur constant peut être implémentée sous forme de multiplication et de décalage, en utilisant la méthode décrite à la page 137 ou les fonctions de la bibliothèque de classes vectorielles disponible sur github.com/vectorclass, ou encore la bibliothèque d'assembleur disponible sur www.agner.org/optimize/asmlib.zip.

13.1 Utilisation du jeu d'instructions AVX et des registres YMM ou ZMM

Les registres XMM de 128 bits sont étendus en registres YMM de 256 bits et en registres ZMM de 512 bits lorsque les diverses extensions du jeu d'instructions AVX sont disponibles. D'autres extensions à 1024 bits, voire 2048 bits, sont envisageables à l'avenir, mais aucun projet de ce type n'a encore été publié (en 2025).

On compte 16 registres YMM ou 32 registres ZMM en mode 64 bits. Il n'y a que 8 registres XMM/YMM/ZMM en mode 32 bits.

La plupart des instructions AVX acceptent trois opérandes : un opérande de destination et deux opérandes source. Cela présente l'avantage qu'aucun registre d'entrée n'est écrasé par le résultat.

Toutes les instructions XMM possèdent deux versions dans le jeu d'instructions AVX : une version héritée (*legacy*) qui laisse inchangée la moitié supérieure (bits 128 à 511) du registre ZMM cible, et une version utilisant le préfixe VEX qui met à zéro les bits de poids fort, rendant ainsi le résultat indépendant de la valeur précédente de ces bits dans le registre cible.

La version utilisant le préfixe VEX se distingue par un préfixe “V” ajouté au nom de l'instruction et, dans la plupart des cas, par l'utilisation de trois opérandes :

```
; Exemple 13.2. Versions Legacy et VEX de la même instruction
addps xmm1, xmm2       ; xmm1 = xmm1 + xmm2
vaddps xmm1, xmm2, xmm3 ; xmm1 = xmm2 + xmm3
```

13.2 Mélange de code VEX et SSE

Un problème survient si un programme mélange des instructions utilisant des tailles de vecteur différentes, comme des instructions SSE (avec registres XMM) et des instructions VEX (avec registres YMM ou ZMM).

Le mélange d'instructions YMM ou ZMM avec des instructions SSE classiques (128 bits) dépourvues de préfixe VEX amène certains processeurs Intel à faire basculer le banc de registres entre différents états, une opération coûteuse en cycles d'horloge.

Ce qui suit s'applique aux processeurs Intel Sandy Bridge, Ivy Bridge, Haswell et Broadwell. Ces processeurs disposent de registres YMM, mais pas de registres ZMM.

Pour assurer la compatibilité avec le code SSE existant, l'ensemble des registres peut se trouver dans trois états différents sur les processeurs Sandy Bridge, Ivy Bridge, Haswell et Broadwell :

- A. (Clean state). Les bits de poids fort de tous les registres YMM sont inutilisés et connus pour être à zéro.
- B. (Modified state). Les bits 128 à 255 d'au moins un registre YMM sont utilisés et contiennent des données.
- C. (Saved state). Tous les registres YMM sont scindés en deux. La moitié inférieure est utilisée par les instructions SSE héritées, qui laissent la partie supérieure inchangée. Toutes les moitiés supérieures sont stockées dans une zone de sauvegarde temporaire. Les deux parties de chaque registre seront réunies si nécessaire lors d'une transition vers l'état B.

Deux instructions sont disponibles pour permettre une transition rapide entre ces états : VZEROALL, qui met à zéro tous les registres YMM, et VZERoupper, qui met à zéro la partie supérieure des registres YMM0–YMM15 ou ZMM0–ZMM15. Ces deux instructions laissent le processeur dans l'état A. Les transitions d'état peuvent être illustrées par le tableau de transition d'état suivant :

État courant →	A	B	C
Instruction ↓			
VZEROALL / UPPER	A	A	A
XMM A C C	A	C	C
VEX XMM A B B	A	B	B
VEX YMM B B B	B	B	B

Tableau 13.2. Transitions d'état YMM

L'état A est l'état initial neutre. L'état B est requis lorsque les registres YMM sont utilisés dans leur intégralité. L'état C est nécessaire pour garantir l'exécution rapide du code XMM hérité et l'absence de fausses dépendances lors d'un appel depuis l'état B. Une transition de l'état B vers l'état C est coûteuse car tous les registres YMM doivent être scindés en deux moitiés stockées séparément. Une transition de l'état C vers l'état B est tout aussi coûteuse, car tous les registres doivent être à nouveau fusionnés. Une transition de l'état C vers l'état A est également coûteuse, car la zone de stockage temporaire contenant la partie supérieure des registres ne prend pas en charge l'accès dans le désordre (*out-of-order*) et doit être protégée contre l'exécution spéculative lors de branchements dont la prédiction pourrait être erronée. Les transitions B → C, C → B et C → A sont très chronophages sur ces processeurs Intel, car elles nécessitent d'attendre que tous les registres soient validés (retire). Les transitions A → B et B → A sont rapides et ne prennent au plus qu'un cycle d'horloge. L'état C doit être considéré comme un état indésirable, et la transition A → C est impossible. D'après mes mesures, ces transitions indésirables prennent environ 70 cycles d'horloge sur les processeurs Intel Sandy Bridge, Ivy Bridge, Haswell et Broadwell.

Les exemples suivants illustrent les transitions d'état :

; **Exemple 13.3a.** Transition entre les états YMM

```
vaddps ymm0, ymm1, ymm2 ; État B
addss xmm3, xmm4        ; État C
vmulps ymm0, ymm0, ymm5 ; État B
```

L'exemple 13.3a comporte deux transitions d'état coûteuses : de B vers C, puis un retour à l'état B. Cette transition d'état peut être évitée en remplaçant l'instruction ADDSS héritée par sa version encodée VEX, VADDSS :

; **Exemple 13.3b.** Transition entre les états YMM évitée

```
vaddps ymm0, ymm1, ymm2 ; État B
vaddss xmm3, xmm3, xmm4 ; État B
vmulps ymm0, ymm0, ymm5 ; État B
```

Cette méthode ne peut pas être utilisée lors de l'appel d'une fonction de bibliothèque qui utilise des instructions XMM. La solution consiste ici à sauvegarder les registres YMM utilisés et à passer à l'état A :

; **Exemple 13.3c.** Transition vers l'état A

```
vaddps ymm0, ymm1, ymm2 ; État B
vmovaps [mem], ymm0      ; Sauvegarde de ymm0
vzeroupper                ; État A
call XMM_Function        ; Fonction héritée
vmovaps ymm0, [mem]      ; Restauration de ymm0
vmulps ymm0, ymm0, ymm5 ; État B
vzeroupper                ; Passez à l'état A avant de revenir.
ret
...
XMM_Function proc near
addss xmm3, xmm4          ; État A
```

ret

Les processeurs Intel Skylake et ultérieurs se comportent différemment. Ils disposent des états A et B, mais pas de l'état C. Cela permet d'éviter les transitions coûteuses vers et depuis l'état C, mais engendre un autre problème.

Les processeurs Intel Skylake et AMD Zen 5 sont incapables de scinder un registre vectoriel en deux. Si une instruction non-VEX écrit dans un registre XMM alors que le processeur est dans l'état B, cette instruction devra attendre la valeur précédente du registre ; en effet, elle pourrait avoir à combiner la partie basse issue de l'instruction non-VEX avec la partie haute provenant d'une instruction VEX précédente. Par exemple :

```
; Exemple 13.4. Mélange d'instructions VEX et SSE sur Skylake
vaddps zmm0, zmm1, zmm2    ; État B
mulps xmm3, xmm4
movaps [mem], xmm3          ; Sauvegarde de xmm3
movps xmm3, xmm5            ; Il faut attendre mulps.
```

L'état ne fait pas de distinction entre les différents registres. Lorsqu'il se trouve dans l'état B, le processeur considère que la partie supérieure de tous les registres est "sale" (modifiée). Par conséquent, la dernière instruction doit attendre le résultat de la multiplication précédente pour pouvoir fusionner les deux parties du registre zmm3.

Le processeur AMD Ryzen scinde toutes les instructions de 256 bits en deux micro-opérations (μops) de 128 bits, de sorte que ces problèmes ne surviennent jamais.

Recommandations pour mélanger du code VEX et non-VEX

Le mélange de code VEX et non-VEX est fréquent lors de l'utilisation de bibliothèques de fonctions. Par exemple, un programme compilé pour le jeu d'instructions AVX512 peut appeler une fonction de bibliothèque compilée pour SSE2.

Il convient de respecter les directives suivantes pour obtenir des performances optimales dans les programmes mêlant code SSE et code VEX.

- Une fonction utilisant des instructions YMM ou ZMM doit exécuter une instruction VZEROALL ou VZERoupper avant de rendre la main, s'il existe une possibilité qu'elle retourne vers du code SSE.
- Une fonction utilisant des instructions YMM ou ZMM doit sauvegarder tout registre YMM ou ZMM utilisé et exécuter une instruction VZEROALL ou VZERoupper avant d'appeler une fonction susceptible de contenir du code SSE.
- Une fonction disposant d'un répartiteur (*dispatcher*) pour choisir entre du code YMM ou ZMM (si disponible) et du code XMM doit exécuter une instruction VZEROALL ou VZERoupper avant de quitter la section de code YMM ou ZMM.

Cette recommandation permet à toute fonction d'utiliser les registres vectoriels les plus larges disponibles ; toutefois, une fonction utilisant des registres YMM ou ZMM doit laisser ces registres dans l'état A avant d'appeler une fonction dont l'état VEX est inconnu, ainsi qu'avant de rendre la main à une fonction dont l'état VEX est inconnu.

Cela ne s'applique évidemment pas aux fonctions utilisant des registres YMM ou ZMM pour le passage de paramètres ou la valeur de retour. Une fonction utilisant des registres YMM ou ZMM pour le passage de paramètres peut supposer que l'état B est actif à l'entrée. Une fonction utilisant un registre YMM ou ZMM pour la valeur de retour ne peut se trouver qu'à l'état B au moment du retour. L'état C doit toujours être évité.

L'instruction VZERoupper est plus rapide que VZEROALL sur la plupart des processeurs. Il est donc recommandé d'utiliser VZERoupper plutôt que VZEROALL, sauf si une initialisation complète est requise. VZEROALL ne peut pas être utilisé sous Windows 64 bits, car l'ABI stipule que les registres XMM6 à XMM15 doivent être préservés par la fonction appelée (*callee-save*). En d'autres termes, la fonction appelante peut supposer que les registres XMM6 à XMM15 — mais pas les parties supérieures des registres YMM ou ZMM — restent inchangés après le retour. Aucun registre vectoriel n'est soumis à cette règle de préservation sous Windows 32 bits ou sur les systèmes Unix (Linux, BSD, Mac). Il est donc acceptable d'utiliser VZEROALL sous Linux, par exemple. Bien entendu, VZEROALL ne peut pas être utilisé si un registre XMM contient un paramètre de fonction ou une valeur de retour ; dans ces cas, VZERoupper doit être utilisé.

Malheureusement, les instructions VZERoupper et VZEROALL sont coûteuses sur le processeur Knights Landing (le premier processeur à prendre en charge l'AVX512). Il est déconseillé d'utiliser ces instructions sur Knights Landing. La question de savoir quand utiliser VZERoupper (ou non) est abordée à l'adresse suivante : <https://software.intel.com/en-us/forums/intel-saextensions/topic/704023>.

Une solution alternative consiste à n'utiliser que les registres ZMM16 à ZMM31. Ces registres ne sont pas accessibles au code SSE et n'affectent pas l'état du processeur. Bien que cette approche puisse être optimale sur les processeurs prenant en charge l'AVX512, elle ne peut être retenue lorsque les registres YMM0, ZMM0, etc., sont nécessaires pour les paramètres ou la valeur de retour d'une fonction. Notez que les instructions VZERoupper et VZEROALL n'affectent que les registres ZMM0 à ZMM15, et non les registres ZMM16 à ZMM31.

Mélanger des instructions VEX et non-VEX est une erreur courante, très facile à commettre et très difficile à détecter. Le code continuera de fonctionner, mais avec des performances réduites sur la plupart des processeurs Intel. Il est vivement recommandé de vérifier minutieusement votre code afin de détecter tout mélange d'instructions VEX et non-VEX.

Temps de préchauffage

De nombreux processeurs sont capables de désactiver les bits de poids fort des unités d'exécution de 256 ou 512 bits afin d'économiser de l'énergie lorsque ces unités ne sont pas utilisées.

La réactivation de cette partie supérieure après une période d'inactivité prend généralement entre 10 et 20 μ s. Le débit des instructions vectorielles de 256 bits est nettement inférieur durant cette phase de réactivation, car le processeur utilise deux fois les unités de 128 bits (bits de poids faible) pour exécuter une opération de 256 bits. Il est possible d'anticiper la réactivation des unités de 256 ou 512 bits en exécutant une instruction fictive de 256 ou 512 bits à un moment opportun, avant que l'unité de plus grande taille ne soit requise. La partie supérieure des unités de 256 ou 512 bits sera de nouveau désactivée après environ 1 ms sans exécution d'instructions vectorielles de grande taille. Ce phénomène est décrit dans le manuel n° 3 : "The microarchitecture of Intel, AMD and VIA CPUs".

Prise en charge des systèmes d'exploitation

Le code utilisant des registres YMM ou ZMM ou des instructions encodées VEX ne peut s'exécuter que sur un système d'exploitation prenant en charge cette taille de registre, car le système d'exploitation doit sauvegarder les registres YMM ou ZMM lors des changements de tâche.

Les versions de système d'exploitation suivantes prennent en charge l'AVX et les registres YMM : Windows 7, Windows Server 2008 SP2, ainsi que le noyau Linux version 2.6.30 et ultérieures.

Les versions de système d'exploitation suivantes prennent en charge l'AVX-512 et les registres ZMM : Windows 10 et ultérieures, ainsi que le noyau Linux version 3.15 et ultérieures.

YMM / ZMM et code système

Une situation dans laquelle des transitions entre les états B et C doivent avoir lieu apparaît lorsque le code YMM est interrompu et que le gestionnaire d'interruption contient du code XMM hérité qui enregistre les registres XMM mais pas les registres YMM complets. En fait, l'état C a été inventé précisément dans le but de préserver la partie supérieure des registres YMM dans cette situation.

Il est très important de se conformer à certaines règles lors de l'écriture de pilotes de périphériques et d'autres codes système pouvant être appelés à partir des gestionnaires d'interruptions. Si un registre vectoriel est modifié par une instruction VEX dans le code système, il est alors nécessaire de sauvegarder l'intégralité de l'état du registre avec XSAVE et de le restaurer avec XRESTOR avant de revenir. Il ne suffit pas de sauvegarder les registres YMM individuels car les futurs processeurs, qui pourraient étendre davantage les registres YMM à des registres ZMM de 512 bits ou encore plus grands, étendront à zéro les registres YMM à ZMM lors de l'exécution des instructions YMM et détruiront ainsi la partie la plus élevée des registres ZMM. XSAVE/XRESTOR est le seul moyen de sauvegarde de ces registres compatible avec les futures extensions au-delà de 256 ou 512 bits. Les futures extensions n'utiliseront pas la méthode compliquée consistant à avoir deux versions de chaque instruction vectorielle.

Si un pilote de périphérique n'utilise pas XSAVE/XRESTOR, il existe un risque qu'il utilise par inadvertance du code VEX même si le programmeur n'en avait pas l'intention. Un compilateur qui n'est pas destiné au code système peut insérer des appels implicites aux fonctions de bibliothèque telles que memset et memcpy. Ces fonctions ont généralement leur propre répartiteur de CPU qui peut sélectionner la plus grande taille de registre disponible. Il est donc nécessaire d'utiliser un compilateur et une bibliothèque de fonctions destinés à réaliser du code système.

Ces règles sont décrites plus en détail dans le manuel 5 : "Conventions d'appel".

Utilisation d'instructions non destructives à trois opérandes

Le jeu d'instructions AVX qui définit les instructions YMM définit également un codage alternatif de toutes les instructions XMM existantes en remplaçant les préfixes et codes d'échappement existants par le nouveau préfixe VEX. Le préfixe VEX présente l'avantage supplémentaire de définir un opérande de registre supplémentaire. Presque toutes les instructions XMM qui avaient auparavant deux opérandes en ont désormais trois lorsque le préfixe VEX est utilisé.

La version à deux opérandes d'une instruction utilise généralement le même registre pour la destination et pour l'un des opérandes source :

```
; Exemple 13.5a. Instruction à 2 opérandes
movsd xmm0, xmm1    ; copie de xmm1 pour éviter d'écraser la valeur.
addsd xmm0, xmm2     ; xmm0 = xmm0 + xmm2
```

Cela présente l'inconvénient que le résultat écrase la valeur de l'un des opérandes sources. L'instruction MOV de l'exemple 13.5a peut être évitée si l'on utilise la variante de l'addition à trois opérandes :

; **Exemple 13.5b.** Instruction à 3 opérandes

```
vaddsd xmm0, xmm1, xmm2 ; xmm0 = xmm1 + xmm2
```

Ici, aucun des opérandes sources n'est détruit, car le résultat peut être stocké dans un registre de destination différent. Cela peut s'avérer utile pour éviter les transferts de registre à registre. Les instructions `addsd` et `vaddsd` des exemples 13.5a et 13.5b ont exactement la même longueur. Par conséquent, l'utilisation de la version à trois opérandes n'entraîne aucune pénalité. Les instructions dont le nom se termine par "ps" (simple précision groupée/"packed") sont plus courtes d'un octet dans leur version à deux opérandes que dans leur version VEX à trois opérandes, à condition que le registre de destination ne soit pas compris entre `xmm8` et `xmm15`. Dans quelques cas, la version à trois opérandes est plus courte que celle à deux opérandes. Toutefois, la plupart du temps, les versions à deux et trois opérandes ont la même longueur.

Il est possible de mélanger des instructions à deux et trois opérandes dans un même code, tant que le jeu de registres se trouve dans l'état A. En revanche, si le jeu de registres passe à l'état C pour une raison quelconque, l'alternance entre des instructions XMM avec et sans VEX entraînera un changement d'état coûteux à chaque modification du type d'instruction. Il est donc préférable d'utiliser soit uniquement des versions VEX, soit uniquement des versions sans VEX. Si des registres YMM ou ZMM sont utilisés (état B), vous devez utiliser exclusivement la version avec préfixe VEX pour toutes les instructions XMM, jusqu'à ce que la section de code VEX se termine par une instruction `VZEROALL` ou `VZERoupper`.

Les instructions MMX 64 bits et la plupart des instructions utilisant des registres d'usage général ne disposent pas de versions à trois opérandes. Le mélange d'instructions MMX et VEX n'entraîne aucune pénalité. Seules quelques instructions utilisant des registres d'usage général possèdent également trois opérandes.

Accès mémoire non aligné

Toutes les instructions vectorielles encodées en VEX comportant un opérande mémoire autorisent les accès mémoire non alignés, à l'exception des instructions explicitement alignées `VMOVAPS`, `VMOVAPD`, `VMOVDQA`, `VMOVNTPS`, `VMOVNTPD` et `VMOVNTDQ`. Par conséquent, il est possible de stocker des opérandes YMM et ZMM sur la pile sans maintenir un alignement de la pile sur 32 octets.

Prise en charge par le compilateur

Les jeux d'instructions AVX sont pris en charge par les compilateurs Microsoft, Intel, GNU et Clang.

Ces compilateurs utilisent la version avec préfixe VEX pour toutes les instructions XMM, y compris les fonctions intrinsèques, lors de la compilation pour un jeu d'instructions AVX. Il incombe au programmeur d'insérer une instruction `VZERoupper` avant toute transition d'un module compilé avec AVX vers un module ou une bibliothèque compilé sans AVX.

Instructions de multiplication-addition fusionnée (FMA)

Une instruction de multiplication-accumulation fusionnée (FMA) permet d'effectuer une multiplication en virgule flottante suivie d'une addition ou d'une soustraction en virgule flottante dans le même laps de temps qu'une simple multiplication. L'opération FMA se présente sous la forme suivante :

$$d = a * b + c$$

Il existe deux variantes distinctes d'instructions FMA, appelées FMA3 et FMA4. Les instructions FMA4 peuvent utiliser quatre registres différents pour les opérandes a, b, c et d. Les instructions FMA3 ne comportent que trois opérandes, la destination d devant utiliser le même registre que l'un des opérandes d'entrée a, b ou c. Intel a initialement conçu le jeu d'instructions FMA4, mais ne prend actuellement en charge que le FMA3. Les derniers processeurs AMD prennent en charge à la fois le FMA3 et le FMA4.

Voir, sur ce sujet : en.wikipedia.org/wiki/FMA_instruction_set/History.

Exemples

L'exemple 12.8, page 78, illustre l'utilisation du jeu d'instructions AVX512 pour un calcul DAXPY, au moyen d'instructions FMA3.

13.3 Utilisation du jeu d'instructions AVX-512 et des registres ZMM

Les registres vectoriels sont étendus à 512 bits avec le jeu d'instructions AVX512. Le nombre de registres vectoriels passe à 32 en mode 64 bits, alors qu'il n'y en a que 8 en mode 32 bits.

Il existe 8 nouveaux registres de masque, nommés `k0` à `k7`. Les registres de masque `k1` à `k7` peuvent être utilisés pour effectuer des opérations conditionnelles sur chaque élément du vecteur, comme expliqué à la page 92. Les opérations conditionnelles sont également utiles pour les boucles sur des tableaux lorsque la taille du tableau n'est pas un multiple de la taille des registres vectoriels. Ce point est expliqué à la page 78.

Il existe plusieurs extensions supplémentaires à l'AVX512. Tous les processeurs compatibles AVX512 intègrent certaines de ces extensions, mais aucun processeur ne les possède toutes à ce jour (en 2020). Les extensions connues et prévues pour l'AVX512 sont les suivantes :

- AVX512F. Fondation. Tous les processeurs AVX512 en sont dotés. Inclut les opérations sur des entiers 32 et 64 bits, ainsi que sur des nombres à virgule flottante (simple et double précision) au sein de vecteurs de 512 bits, y compris des opérations avec masquage.
- AVX512VL. Comprend les mêmes opérations sur les vecteurs 128 bits et 256 bits, y compris les opérations masquées et 32 registres vectoriels.
- AVX512BW. Opérations sur des entiers 8 bits et 16 bits dans des vecteurs 512 bits.
- AVX512DQ. Instructions de multiplication et de conversion avec des entiers 64 bits. Diverses autres instructions sur float et double.
- AVX512ER. Fonction réciproque rapide, racine carrée réciproque et fonction exponentielle. Précis sur float ; approximatif sur double.
- AVX512CD. Détection de conflits. Recherchez les éléments en double dans un vecteur.
- AVX512PF. Instructions de prefetch (*prélecture*) avec logique gather/scatter (*collecte/diffusion*).
- AVX512VBMI. Permutation et décalage avec une granularité de 8 bits.
- AVX512VBMI2. Compression et expansion avec une granularité de 8 bits et 16 bits.
- AVX512IFMA. Multiplication et addition fusionnées sur des entiers de 52 bits.
- AVX512_4VNNIW. Produit scalaire itéré sur des entiers 16 bits.
- AVX512_4FMAPS. Multiplication et addition fusionnées itérées, simple précision.

13.4 Déplacements conditionnels dans les registres xmm et ymm

Considérez ce code C++ dont l'objectif est de trouver les plus grandes valeurs parmi quatre paires de valeurs :

```
// Exemple 13.6a. Boucle pour trouver les maximums
float a[4], b[4], c[4];
for (int i = 0; i < 4; i++) {
    c[i] = a[i] > b[i] ? a[i] : b[i];
}
```

Si nous voulons implémenter ce code à l'aide de registres XMM, nous ne pouvons pas utiliser de saut conditionnel pour la bifurcation à l'intérieur de la boucle, car la condition de branchement n'est pas la même pour les quatre éléments. Heureusement, il existe une instruction de maximum qui permet d'obtenir le même résultat :

```
; Exemple 13.6b. Maximum dans XMM
movaps xmm0, [a] ; Chargement d'un vecteur
maxps xmm0, [b] ; max(a,b)
movaps [c], xmm0 ; c = a > b ? a : b
```

Il existe des instructions vectorielles de calcul de minimum et de maximum pour les nombres à virgule flottante en simple et double précision, ainsi que pour les entiers sur 8 et 16 bits. Des instructions vectorielles permettent également de déterminer la valeur absolue d'entiers. La valeur absolue d'éléments vectoriels en virgule flottante s'obtient en annulant le bit de signe via une opération AND, comme illustré dans l'exemple 13.17 (page 101). Les instructions vectorielles d'addition entière avec saturation (par exemple PADDSW) peuvent aussi servir à déterminer un maximum ou un minimum, ou à limiter des valeurs à une plage donnée.

Ces méthodes manquent toutefois de généralité. La manière la plus générale d'effectuer des transferts conditionnels dans des registres vectoriels consiste à utiliser des instructions vectorielles booléennes. L'exemple suivant est une variante de l'exemple précédent, adaptée au cas où l'on ne peut pas utiliser l'instruction MAXPS :

```
// Exemple 13.7a. Branchement dans une boucle
float a[4], b[4], c[4], x[4], y[4];
for (int i = 0; i < 4; i++) {
    c[i] = x[i] > y[i] ? a[i] : b[i];
}
```

La méthode utilisée pour effectuer des déplacements conditionnels dépend du jeu d'instructions. Le premier jeu d'instructions SSE implémente les déplacements conditionnels en créant un masque composé uniquement de 1 lorsque la condition est vraie et uniquement de 0 lorsqu'elle est fausse. On applique une opération AND logique entre $a[i]$ et ce masque, et entre $b[i]$ et le masque inversé :

```
; Exemple 13.7b. Déplacement conditionnel avec le jeu d'instructions SSE2
movaps xmm1, [y] ; Chargement du vecteur y
cmpltps xmm1, [x] ; Comparaison avec x. XMM1 = masque pour y < x
movaps xmm0, [a] ; Chargement du vecteur a
andps xmm0, xmm1 ; a AND mask
andnps xmm1, [b] ; b AND NOT mask
orps xmm0, xmm1 ; (a AND mask) OR (b AND NOT mask)
movaps [c], xmm0 ; c = x > y ? a : b
```

Les vecteurs définissant la condition (x et y dans l'exemple 13.7b ci-dessus) et les vecteurs sélectionnés (a et b dans l'exemple 13.7b) ne sont pas nécessairement du même type. Par exemple, x et y peuvent être des entiers. Toutefois, ils doivent avoir le même nombre de bits par élément. Si a et b sont des nombres à virgule flottante double précision (doubles) comportant deux éléments par vecteur, et que x et y sont des entiers 32 bits comportant quatre éléments par vecteur, il est alors nécessaire de dupliquer chaque élément de x et y pour obtenir la taille de masque appropriée (voir l'exemple 13.9b ci-dessous).

Notez que l'instruction AND-NOT (andnps, andnpd, pandn) inverse l'opérande de destination, et non l'opérande source. Cela signifie qu'elle détruit le masque. Par conséquent, l'instruction andps doit précéder andnps dans l'exemple 13.7b. Si le jeu d'instructions SSE4.1 est pris en charge, on peut utiliser à la place l'instruction BLENDVPS :

```
; Exemple 13.7c. Déplacement conditionnel avec le jeu d'instructions SSE4.1
movaps xmm0, [y]          ; Chargement du vecteur y
cmpltps xmm0, [x]          ; Comparaison avec x. XMM0 = masque pour y < x
movaps xmm1, [a]           ; Chargement du vecteur a
blendvps xmm1, [b], xmm0   ; Mélange (Blend) de a et b.
movaps [c], xmm0           ; c = x > y ? a : b
```

Si le masque doit être utilisé plus d'une fois, il peut être plus efficace d'effectuer un AND logique entre ce masque et le résultat d'un XOR (OU exclusif) entre a et b. C'est ce qu'illustre l'exemple suivant, qui réalise une permutation conditionnelle de a et b :

```
// Exemple 13.8a. Permutation conditionnelle dans une boucle
float a[4], b[4], x[4], y[4], temp;
for (int i = 0; i < 4; i++) {
    if (x[i] > y[i]) {
        temp = a[i]; // Permute a[i] et b[i] si x[i] > y[i]
        a[i] = b[i];
        b[i] = temp;
    }
}
```

Et voici maintenant le code assembleur utilisant les registres XMM :

```
; Exemple 13.8b. Permutation conditionnelle dans les registres XMM, SSE
movaps xmm2, [y] ; Chargement du vecteur y
cmpltps xmm2, [x] ; Comparaison avec x. XMM2 = masque pour y < x
movaps xmm0, [a] ; Chargement du vecteur a
movaps xmm1, [b] ; Chargement du vecteur b
xorps xmm0, xmm1 ; a XOR b
andps xmm2, xmm0 ; (a XOR b) AND mask
xorps xmm1, xmm2 ; b XOR ((a XOR b) AND mask)
xorps xmm2, [a] ; a XOR ((a XOR b) AND mask)
movaps [b], xmm1 ; (x[i] > y[i]) ? a[i] : b[i]
movaps [a], xmm2 ; (x[i] > y[i]) ? b[i] : a[i]
```

L'instruction xorps xmm0, xmm1 génère une configuration de bits correspondant aux différences entre a et b. Cette configuration est combinée par un AND logique avec le masque, de sorte que xmm2 contienne les bits à modifier si a et b doivent être échangés, ou des zéros dans le cas contraire. Les deux dernières instructions xorps inversent les bits devant être modifiés si l'échange doit avoir lieu, et laissent les valeurs inchangées sinon.

Le masque utilisé pour les déplacements conditionnels peut également être généré en décalant le bit souhaité vers la position de poids fort, en vue d'une utilisation avec des instructions de fusion (*blend*). Si l'on utilise la méthode basée sur les instructions AND/ANDN, on copie alors le bit de poids fort dans toutes les positions de bits à l'aide de l'instruction de décalage arithmétique à droite psrad.

La méthode par décalage est illustrée dans l'exemple suivant, où les éléments d'un vecteur sont élevés à différentes puissances entières. Nous appliquons ici la méthode décrite dans l'exemple 12.14a (page 83) pour le calcul des puissances.

```
// Exemple 13.9a. Élévation des éléments d'un vecteur à différentes puissances entières
double x[2], y[2]; unsigned int n[2];
for (int j = 0; j < 2; j++) {
    y[j] = pow(x[j], n[j]);
}

// Cela peut être optimisé en
double x[2], xp[2], y[2];
unsigned int n[2], i[2];
```

```

for (int j = 0; j < 2; j++) {
    xp[j] = x[j];
    y[j] = 1.0;
    for (i[j] = n[j]; i[j] != 0; i[j] >>= 1) {
        if (i[j] & 1) y[j] *= xp[j];
        xp[j] *= xp[j];
    }
}

```

Si les éléments de n sont égaux, la solution la plus simple consiste à utiliser un branchement. En revanche, si les puissances diffèrent, il faut recourir à des déplacements conditionnels :

; **Exemple 13.9b.** Élever un vecteur à des puissances à l'aide d'un masque entier (SSE4.1)

```

section .data
align 16

one DQ 1.0, 1.0          ; Constitution de la constante 1.0
X   DQ 2.0, 3.0          ; x[0], x[1]
Y   DQ 0, 0              ; y[0], y[1]
N   DD 4, 5              ; n[0], n[1]

section .text
default rel

; xmm0 = sélecteur pour déplacement conditionnel
; xmm1 = xp
; xmm2 = i (i0 et i1 sont chacun stockés deux fois sous forme d'entiers DWORD.)
; xmm3 = y
; xmm4 = (i & 1) ? xp : 1.0
    movq xmm2, [N]        ; Chargement de n0, n1
    punpckldq xmm2, xmm2   ; Obtention de chaque valeur deux fois : n0, n0, n1, n1
    movapd xmm1, [X]       ; Chargement de x0, x1
    movapd xmm3, [one]     ; y initialisé à 1.0
    mov eax, [N]           ; n0
    or eax, [N+4]          ; n0 OR n1 pour obtenir le bit de poids fort
    xor ecx, ecx           ; 0 si n0 et n1 sont tous les deux à zéro
    bsr ecx, eax           ; Calcul du nombre de répétitions pour max(n0, n1)
L1: movdqa xmm0, xmm2       ; Copie de i
    pslld xmm0, 31         ; Placement du bit de poids faible de i
                           ; dans le bit de plus fort poids
    movapd xmm4, [one]     ; 1.0
    blendvpd xmm4, xmm1, xmm0 ; (i & 1) ? xp : 1.0
    psrld xmm2, 1          ; i >>= 1
    mulpd xmm3, xmm4       ; y *= (i & 1) ? xp : 1.0
    mulpd xmm1, xmm1       ; xp = xp * xp
    sub ecx, 1             ; Décrémenter le compteur de boucle
    jns L1                 ; Répétition ECX+1 fois
    movapd [Y], xmm3       ; Stockage du résultat

```

Le nombre d'itérations de la boucle est calculé séparément, en dehors de celle-ci, afin de réduire le nombre d'instructions à l'intérieur de la boucle.

Les transferts conditionnels vers des registres d'usage général (via `CMOVcc`) et vers des registres à virgule flottante (via `FCMOVcc`) ne sont pas plus rapides que ceux effectués vers des registres XMM.

13.5 Déplacements conditionnels avec AVX-512

Le jeu d'instructions AVX512 prend en charge des opérations masquées contrôlées par un registre de masque. Une instruction vectorielle masquée n'effectue son opération que sur les éléments du vecteur pour lesquels le bit correspondant dans le registre de masque est à 1. Nous pouvons réécrire l'exemple 13.7 pour illustrer cela :

```

// Exemple 13.10a. Branchement dans une boucle utilisant AVX-512
float a[16], b[16], c[16], x[16], y[16];
for (int i = 0; i < 16; i++) {
    c[i] = x[i] > y[i] ? a[i] : b[i];
}

```

L'implémentation de ceci avec AVX-512 est très efficace :

```
; Exemple 13.10b. Déplacement conditionnel avec AVX-512
section .text
default rel

vmovaps zmm1, [Y]          ; Chargement du vecteur y
vcmpps k1, zmm1, [X], 1    ; Comparaison avec x. k1 = masque pour y < x
vmovaps zmm0, [b]          ; Chargement du vecteur b
vmovaps zmm0{k1}, [a]      ; Chargement du vecteur a pour les éléments avec le bit de masque 1
vmovaps [c], zmm0          ; c = x > y ? a : b
```

La plupart des instructions vectorielles peuvent être masquées avec AVX-512, par exemple :

```
// Exemple 13.11a. Addition conditionnelle avec AVX-512
float a[16], b[16];
for (int i = 0; i < 16; i++) {
    if (a[i] < 0) {
        a[i] += b[i];
    }
}
```

Cela peut être mis en œuvre à l'aide d'une addition avec masque :

```
; Exemple 13.11b. Addition conditionnelle avec AVX-512
section .text
default rel

vmovaps zmm1, [a]          ; Chargement du vecteur a
vpxord zmm0, zmm0, zmm0    ; provoque ZMM0 = 0
vcmpps k1, zmm1, zmm0, 1   ; Comparaison avec zéro. k1 = masque pour a < 0
vaddps zmm1{k1}, zmm1, [b] ; Addition avec masque
vmovaps [a], zmm1          ; mémorisation du résultat
```

Les instructions avec masque peuvent également être utilisées sur les registres XMM et YMM de plus petite taille si l'extension du jeu d'instructions AVX512VL est prise en charge. L'instruction vpxord peut être remplacée par vxorps si le jeu d'instructions AVX512DQ est pris en charge.

Dans les exemples ci-dessus, les éléments du vecteur pour lesquels le bit de masque est à 0 restent inchangés. Il est possible de mettre à zéro les éléments désactivés, plutôt que de les laisser inchangés, en spécifiant l'option {z} :

```
; Exemple 13.12. Masquage et mise à zéro
section .text
default rel

mov eax, 000FH              ; définition d'une constante
kmovw k1, eax              ; copie de la constante dans le registre de masque
vmovaps zmm1{k1}, [a]      ; déplacement conditionnel
vmovaps zmm2{k1}{z}, [b]   ; déplacement conditionnel avec mise à zéro
```

Dans cet exemple, les quatre premiers éléments du tableau a sont chargés dans zmm1, tandis que les éléments restants de zmm1 demeurent inchangés. Les quatre premiers éléments du tableau b sont chargés dans zmm2, et les éléments restants de zmm2 sont mis à zéro. L'option de mise à zéro est utile car elle supprime la dépendance de zmm2 vis-à-vis de sa valeur précédente. Il est recommandé d'utiliser cette option lorsque cela est approprié, car elle améliore l'efficacité de l'exécution dans le désordre (*out-of-order*) des instructions masquées.

Le masquage n'entraîne aucun surcoût temporel, dans la mesure où les latences et les débits de la plupart des instructions restent identiques avec ou sans masquage, quelle que soit la valeur du masque. En revanche, vous risquez de perdre du temps à effectuer des calculs alors que le masque ne contient que des zéros. Il peut s'avérer avantageux d'ignorer des calculs conditionnels coûteux en temps si le masque est entièrement constitué de zéros. Par exemple :

```
// Exemple 13.13a. Ignorer les calculs si le masque ne contient que des zéros.
float a[16], b[16];
for (int i = 0; i < 16; i++) {
    if (a[i] >= 0) {
        b[i] = sqrt(a[i]);
    }
}
```

S'il arrive souvent que tous les éléments du tableau a soient négatifs, on peut alors éviter l'opération coûteuse en temps de calcul qu'est la racine carrée :

; **Exemple 13.13b.** Ignorer les calculs si le masque ne contient que des zéros.

```
section .text
default rel

vmovaps zmm1, [a]          ; Chargement du vecteur a
vpxord zmm0, zmm0, zmm0    ; provoque ZMM0 = 0
vcmpps k1, zmm1, zmm0, 1   ; Comparaison avec zéro. k1 = masque pour a < 0
kortestw k1, k1            ; Test des bits de k1
jz L1                     ; saut si tout est à zéro
vmovaps zmm2, [b]          ; Chargement du vecteur b
vsqrtps zmm2{k1}, zmm1     ; racine carrée d'éléments non négatifs
vmovaps [b], zmm2          ; mémorisation de b modifié
L1:
```

Une instruction avec masque peut souvent remplacer deux instructions, car elle effectue deux opérations. Même les instructions de comparaison dont la sortie est dirigée vers un registre de masque peuvent être masquées. Les instructions de comparaison et de test utilisant un registre de masque en sortie et soumises à un masque supplémentaire disposent toujours d'une option de mise à zéro implicite ; ainsi, pour les bits où le masque d'entrée est nul, le masque de sortie est mis à zéro plutôt que de rester inchangé. Le résultat correspond à un AND logique entre le résultat de la comparaison et le masque d'entrée. Cela signifie que les opérations de type AND peuvent être implémentées plus efficacement que celles de type OR. L'exemple suivant l'illustre :

// **Exemple 13.14a.** Utilisation de masques

```
float a[16], b[16], c[16];
for (int i = 0; i < 16; i++) {
    if (a[i] > 0 || b[i] > 0) {
        c[i] = a[i] + b[i];
    }
    else {
        c[i] = a[i] - b[i];
    }
}
```

Dans cet exemple, nous pouvons transformer l'opération OR (||) en AND (&&) en inversant toutes les entrées et sorties selon la formule : $a \ || \ b = \neg(\neg a \ \&\& \ \neg b)$. Le code devient alors :

// **Exemple 13.14b.** Utilisation de masques : conversion du OR en AND.

```
float a[16], b[16], c[16];
for (int i = 0; i < 16; i++) {
    if (a[i] <= 0 && b[i] <= 0) {
        c[i] = a[i] - b[i];
    }
    else {
        c[i] = a[i] + b[i];
    }
}
```

C'est utile car l'opération AND peut être optimisée en masquant la seconde instruction de comparaison avec le résultat de la première :

; **Exemple 13.14c.** Utilisation efficace des masques

```
section .text
default rel

vmovups zmm1, [a]          ; Chargement du vecteur a
vmovups zmm2, [b]          ; Chargement du vecteur b
vpxord zmm0, zmm0, zmm0    ; provoque ZMM0 = 0
vcmpps k1, zmm1, zmm0, 2   ; k1 = masque pour a <= 0
vcmpps k2{k1}, zmm2, zmm0, 2 ; k2 = masque pour a <= 0 && b <= 0
vaddps zmm0, zmm1, zmm2    ; a + b
vsubps zmm0{k2}, zmm1, zmm2 ; a - b if masque
vmovups [c], zmm0          ; mémorisation du résultat
```

13.6 Utilisation d'instructions vectorielles avec des types de données autres que ceux pour lesquels elles sont conçues

La plupart des instructions XMM, YMM et ZMM sont “typées”, au sens où elles sont conçues pour un type de données spécifique. Par exemple, il est illogique d'utiliser une instruction d'addition d'entiers sur des données en

virgule flottante. Toutefois, les instructions qui se contentent de déplacer des données fonctionnent avec n'importe quel type de données, même si elles sont initialement prévues pour un type particulier. Cela peut s'avérer utile lorsqu'il n'existe pas d'instruction équivalente pour le type de données utilisé, ou lorsqu'une instruction destinée à un autre type de données s'avère plus efficace.

Toutes les instructions vectorielles de déplacement, de permutation (*shuffle*), de fusion (*blend*) ou de décalage (*shift*), ainsi que les instructions booléennes, peuvent être utilisées avec des types de données autres que ceux pour lesquels elles ont été conçues. En revanche, les instructions effectuant des opérations arithmétiques, des conversions de type ou des conversions de précision sont strictement réservées au type de données pour lequel elles sont prévues. Par exemple, l'instruction FLD ne se contente pas de déplacer des données en virgule flottante ; elle effectue également une conversion de précision. Si vous tentez d'utiliser FLD et FSTP pour déplacer des données entières, vous risquez de déclencher des exceptions liées aux opérandes “subnormaux” (*denormals*) si la valeur entière ne correspond pas à un nombre normal en virgule flottante. Dans certains cas, l'instruction peut même modifier la valeur des données. À l'inverse, l'instruction MOVAPS, bien qu'elle soit également destinée au déplacement de données en virgule flottante, n'effectue aucune conversion de précision ni aucune autre transformation ; elle se borne à déplacer les données. Il est donc possible d'utiliser MOVAPS pour déplacer des données entières.

En cas de doute quant à la compatibilité d'une instruction donnée avec un type de données spécifique, consultez le manuel technique d'Intel ou d'AMD. Si l'instruction est susceptible de générer une “exception en virgule flottante”, elle ne doit pas être utilisée avec des données autres que celles pour lesquelles elle a été conçue.

L'utilisation de types d'instructions inappropriés peut entraîner une baisse de performances sur certains processeurs. Cela s'explique par le fait que le processeur peut disposer de bus de données ou d'unités d'exécution distincts pour les nombres entiers et les nombres en virgule flottante. Le transfert de données entre les unités de traitement des entiers et celles des nombres en virgule flottante peut nécessiter un ou plusieurs cycles d'horloge selon le processeur, comme indiqué dans le tableau 13.3.

Processeur	Délai de contournement, cycles d'horloge
Intel Core 2 et précédents	1
Intel Nehalem	2
Intel Sandy Bridge et suivants	0-1
Intel Atom	0
AMD	0-2
VIA Nano	2-3

Tableau 13.3. Délais de contournement de données entre les unités d'exécution d'entiers et en virgule flottante

Sur certains processeurs Intel, quelques instructions en virgule flottante sont exécutées par les unités de traitement des entiers. C'est notamment le cas des instructions de transfert (move) XMM, des opérations booléennes ainsi que de certaines instructions de permutation (*shuffle*) et de regroupement (*pack*) sur les processeurs Intel Core 2. Ces instructions entraînent un délai de transfert (*bypass delay*) lorsqu'elles sont mêlées à des instructions utilisant l'unité de virgule flottante. Sur la plupart des autres processeurs, l'unité d'exécution utilisée correspond à la nature de l'instruction : par exemple, MOVAPS XMM1, XMM2 utilise l'unité de virgule flottante, tandis que MOVDQA XMM1, XMM2 utilise l'unité de traitement des entiers.

Les instructions effectuant des lectures ou des écritures en mémoire utilisent une unité distincte. Sur certains processeurs, le délai de transfert entre l'unité mémoire et l'unité de virgule flottante peut être plus long que celui vers l'unité de traitement des entiers, mais ce délai ne dépend pas du type d'instruction. Ainsi, il n'y a pas de différence de latence entre MOVAPS XMM0, [MEM] et MOVDQA XMM0, [MEM] sur les processeurs actuels, bien qu'une telle différence ne puisse être exclue pour les processeurs futurs.

De plus amples détails concernant les unités d'exécution des différents processeurs figurent dans le manuel n° 3, intitulé “The microarchitecture of Intel, AMD and VIA CPUs”. Le manuel n° 4, “Instruction tables”, contient la liste de toutes les instructions ainsi que les unités d'exécution qu'elles utilisent.

L'utilisation d'une instruction d'un type inapproprié peut s'avérer avantageuse lorsqu'il n'y a pas de délai de transfert ou lorsque le débit est plus important que la latence. Certains de ces cas sont décrits ci-dessous.

En utilisant l'instruction la plus courte

Les instructions pour les nombres à virgule flottante en simple précision “packed” (regroupés), dont les noms se terminent par “PS”, sont plus courtes d'un octet que les instructions équivalentes pour la double précision ou les entiers. Par exemple, vous pouvez utiliser MOVAPS au lieu de MOVAPD ou MOVDQA pour transférer des données vers ou depuis la mémoire, ou entre des registres. Un délai de contournement (*bypass delay*) survient sur certains processeurs lorsque l'on utilise MOVAPS pour transférer le résultat d'une instruction sur des entiers vers un autre registre, mais pas lors du transfert de données vers ou depuis la mémoire.

En utilisant l'instruction la plus efficace

Une méthode efficace pour mettre un registre vectoriel à zéro consiste à utiliser l'instruction PXOR XMM0, XMM0. La plupart des processeurs considèrent cette instruction comme indépendante de la valeur précédente de XMM0, alors que ce n'est pas le cas pour XORPS et XORPD sur tous les processeurs. L'instruction PXOR est donc privilégiée pour la mise à zéro d'un registre.

Sur certains processeurs AMD et Intel, les versions pour entiers des instructions vectorielles booléennes (PAND, PANDN, POR, PXOR) peuvent exploiter une plus grande variété d'unités d'exécution que leurs équivalents pour nombres à virgule flottante (ANDPS, etc.).

Utilisation d'une instruction qui n'est pas disponible pour d'autres types de données

Il existe de nombreuses situations où il est avantageux d'utiliser une instruction conçue pour un type de données différent, simplement parce qu'il n'existe pas d'instruction équivalente pour le type de données dont vous disposez.

De nombreuses instructions utiles pour le réarrangement et le mélange de données ne sont disponibles que pour un seul type de données. Ces instructions peuvent toutefois être aisément utilisées avec des types de données autres que ceux pour lesquels elles ont été conçues. Le délai de contournement (*bypass delay*), le cas échéant, peut s'avérer inférieur au coût des solutions alternatives. Les instructions de réarrangement de données (*shuffling instructions*) sont énumérées dans le paragraphe suivant.

13.7 Permutation de données

Le code vectorisé nécessite parfois un grand nombre d'instructions pour échanger et copier des éléments vectoriels, ainsi que pour placer les données aux positions adéquates au sein des vecteurs. Le recours à ces instructions supplémentaires réduit l'avantage lié à l'utilisation d'opérations vectorielles. Il est possible d'utiliser une instruction de permutation conçue pour un type de données différent de celui que vous manipulez, comme expliqué dans le paragraphe précédent. Certaines instructions utiles pour la permutation de données sont énumérées ci-dessous.

Permutation des données au sein de chaque voie de 128 bits

Instruction	Taille de bloc, en bits	Description	Jeu d'instructions
PSHUFD	32	Permutation générale	SSE2
PSHUFLW	16	Permutation uniquement de la moitié inférieure du registre	SSE2
PSHUFW	16	Permutation uniquement de la moitié supérieure du registre	SSE2
SHUFPS	32	Permutation	SSE
SHUFPD	64	Permutation	SSE2
PSLLDQ	8	Décalage à une position différente et réinitialisation de la position d'origine à zéro	SSE2
PSHUFB	8	Permutation générale (supplément)	SSE3
PALIGNR	8	Rotation d'un vecteur de 8 ou 16 octets (Suppl.)	SSE3
PINSRB	8	Insertion d'un octet dans le vecteur	SSE4.1
PINSRW	16	Insertion d'un mot dans le vecteur	SSE
PINSRD	32	Insertion d'un double-mot (DWORD) dans le vecteur	SSE4.1
PINSRQ	64	Insertion d'un quadruple-mot (QWORD) dans le vecteur	SSE4.1
INSERTPS	32	Insertion d'un double-mot (DWORD) dans le vecteur	SSE4.1
VPERMILPS	32	Permutation avec le sélecteur de variable	AVX
VPERMILPD	64	Permutation avec le sélecteur de variable	AVX
VPPERM	8	Permutation avec le sélecteur de variable	AMD XOP

Tableau 13.4. Instructions de permutation

Permutation des données en croisant les voies de 128 bits

Instruction	Taille de bloc, en bits	Description	Jeu d'instructions
VPERMW	16	Permutation avec le sélecteur de variable	AVX512BW
VPERMD	32	Permutation avec le sélecteur de variable	AVX2
VPERMQ	64	Permutation avec le sélecteur de variable	AVX2
VPERMPD	32	Permutation avec le sélecteur de variable	AVX2
VPERMPD	64	Permutation avec le sélecteur de variable	AVX2
VPERMI/T2B	8	Permutation à partir de deux sources	AVX512VBMI
VPERMI/T2W	16	Permutation à partir de deux sources	AVX512BW

Instruction	Taille de bloc, en bits	Description	Jeu d'instructions
VPERMI/T2D	32	Permutation à partir de deux sources	AVX512
VPERMI/T2Q	64	Permutation à partir de deux sources	AVX512
VPERMI/T2PS	32	Permutation à partir de deux sources	AVX512
VPERMI/T2PD	64	Permutation à partir de deux sources	AVX512
VPERM21128	128	Permutation à partir de deux sources	AVX2
VPERM2F128	128	Permutation à partir de deux sources	AVX

Tableau 13.5. Instructions complètes pour la permutation

Combinaison des données provenant de deux sources différentes

Instruction	Taille de bloc, en bits	Description	Jeu d'instructions
SHUFPS	32	Les 2 dwords de poids faible provenant de n'importe quelle position de la destination Les 2 dwords de poids fort provenant de n'importe quelle position de la source	SSE
SHUFPD	64	Mot de 64 bits de poids faible depuis n'importe quelle position de destination Mot de 64 bits de poids fort depuis n'importe quelle position de source	SSE2
MOVLPS/D	64	Mot de 64 bits de poids faible issu de la mémoire, mot de 64 bits de poids fort inchangé	SSE/SSE2
MOVHPS/D	64	Mot de 64 bits de poids fort issu de la mémoire, mot de 64 bits de poids faible inchangé	SSE/SSE2
MOVLHPS	64	Mot de 64 bits de poids faible inchangé, mot de 64 bits de poids fort provenant de la partie de poids faible de la source	SSE
MOVHLPS	64	Mot de 64 bits de poids faible issu de la partie de poids fort de la source, mot de 64 bits de poids fort inchangé	SSE
MOVSS	32	Mot double de poids faible de la source (registre uniquement), bits 32 à 127 inchangés	SSE
MOVSD	64	Quadruple-mot (QWORD) inférieur de la source (registre uniquement), Quadruple-mot supérieur inchangé	SSE2
PUNPCKLBW	8	8 octets de poids faible de la source et de la destination entrelacés	SSE2
PUNPCKLWD	16	4 octets de poids faible de la source et de la destination entrelacés	SSE2
PUNPCKLDQ	32	2 DWORD de poids faible de la source et de la destination entrelacés	SSE2
PUNPCKLQDQ	64	Mot 64 bits de poids faible inchangé, mot 64 bits de poids fort provenant de la partie de poids faible de la source	SSE2
PUNPCKHBW	8	8 octets de poids fort de la source et de la destination entrelacés	SSE2
PUNPCKHWD	16	4 octets de poids fort de la source et de la destination entrelacés	SSE2
PUNPCKHDQ	32	Double-mots de poids fort de la source et de la destination entrelacés	SSE2
PUNPCKHQDQ	64	Mot de 64 bits de poids faible issu de la partie de poids fort de la source, mot de 64 bits de poids fort issu de la partie de poids fort de la source,	SSE2
PACKUSWB	8	Les 8 octets de poids faible issus de 8 mots de la destination, et les 8 octets de poids fort issus de 8 mots de la source. Conversion avec saturation non signée.	SSE2
PACKSSWB	8	Les 8 octets de poids faible issus de 8 mots de destination, les 8 octets de poids fort issus de 8 mots de source. Conversion avec saturation signée.	SSE2
PACKSSDW	16	Les 4 mots de poids faible issus des 4 doubles-mots de la destination, et les 4 mots de poids fort issus des 4 doubles-mots de la source. Conversion avec saturation signée.	SSE2
PINSRB	8	Insertion d' un octet dans le vecteur	SSE4.1

Instruction	Taille de bloc, en bits	Description	Jeu d'instructions
PINSRW	16	Insertion d' un mot dans le vecteur	SSE
PINSRD	32	Insertion d' un double-mot dans le vecteur	SSE4.1
INSERTPS	32	Insertion d' un double-mot dans le vecteur	SSE4.1
PINSRQ	64	Insertion d' un quadruple-mot dans le vecteur	SSE4.1
VINSERTF128	128	Insertion xmmword dans le vecteur	AVX
PALIGNR	8	Double décalage (analogue à SHRD) Suppl.	SSE3
PBLENDW	16	Assemblage (Blend) issu de deux sources différentes	SSE4.1
BLENDPS	32	Assemblage (Blend) issu de deux sources différentes	SSE4.1
BLENDPD	64	Assemblage (Blend) issu de deux sources différentes	SSE4.1
PBLENDVB	8	Multiplexage	SSE4.1
BLENDVPS	32	Multiplexage	SSE4.1
BLENDVPD	64	Multiplexage	SSE4.1
VPCMOV	1	Multiplexage	AMD XOP
VPBLEND	32	Multiplexage	AVX2
VPBLENDMB	8	Multiplexage utilisant un masque comme sélecteur	AVX512BW
VPBLENDMW	16	Multiplexage utilisant un masque comme sélecteur	AVX512BW
VPBLENDMD	32	Multiplexage utilisant un masque comme sélecteur	AVX512
VPBLENDMQ	64	Multiplexage utilisant un masque comme sélecteur	AVX512
VBLENDMPS	32	Multiplexage utilisant un masque comme sélecteur	AVX512
VBLENDMPD	64	Mélanges (Blend) avec le masque comme sélecteur	AVX512

Tableau 13.6. Combinaison de données

Diffusion de données (Broadcasting) vers tous les éléments d'un registre

Instruction	Taille de bloc, en bits	Description	Jeu d'instructions
PSHUFD xmm2, xmm1, 0	32	diffusion (broadcast) d'un dword	SSE2
PSHUFD xmm2, xmm1, 0EEH	64	diffusion (broadcast) d'un qword	SSE2
MOVDDUP	64	diffusion (broadcast) d'un qword	SSE3
MOVSLDUP	32	2 copies de chaque dword 0 et 2	SSE3
MOVSHDUP	32	2 copies de chaque dword 1 et 3	SSE3
VBROADCASTSS	32	diffusion dword à partir de la mémoire	AVX
VBROADCASTSS	32	diffusion dword à partir d'un registre	AVX2
VBROADCASTSD	64	diffusion qword à partir de la mémoire	AVX
VBROADCASTSD	64	diffusion qword à partir d'un registre	AVX2
VBROADCASTF128	128	diffusion 16 octets à partir de la mémoire	AVX
VPBROADCASTB	8	diffusion d'octet à partir d'un registre ou de la mémoire	AVX2
VPBROADCASTW	16	diffusion de mot à partir d'un registre ou de la mémoire	AVX2
VPBROADCASTD	32	diffusion de dword à partir d'un registre ou de la mémoire	AVX2
VPBROADCASTQ	64	diffusion de qword à partir d'un registre ou de la mémoire	AVX2
VBROADCASTF32X2	64	diffusion de qword à partir d'un registre ou de la mémoire	AVX512DQ
VBROADCASTF32X4	128	diffusion de qword à partir de la mémoire	AVX512
VPBROADCASTMB2Q	8	diffusion de qword à partir de la mémoire	AVX512CD
VPBROADCASTMW2D	16	diffusion du registre de masque au vecteur	AVX512CD

Tableau 13.7. Déplacement et diffusion de données

Outre ces instructions, de nombreuses instructions AVX-512 offrent la possibilité d'utiliser un opérande mémoire de type “broadcast”.

Fusion des données provenant de différents emplacements mémoire en un seul vecteur (gather)

Instruction	Taille de bloc, en bits	Description	Jeu d'instructions
VPGATHERDD	32	rassemblement de dwords (gather) à l'aide d'indices dword	AVX2
VPGATHERQD	32	rassemblement de dwords (gather) à l'aide d'indices qword	AVX2

Instruction	Taille de bloc, en bits	Description	Jeu d'instructions
VPGATHERDQ	64	rassemblement de qwords à l'aide d'indices dword	AVX2
VPGATHERQQ	64	rassemblement de qwords à l'aide d'indices qword	AVX2
VGATHERDPS	32	rassemblement de dwords à l'aide d'indices dword	AVX2
VGATHERQPS	32	rassemblement de dwords à l'aide d'indices qword	AVX2
VGATHERDPD	64	rassemblement de qwords à l'aide d'indices dword	AVX2
VGATHERQPD	64	rassemblement de qwords à l'aide d'indices qword	AVX2

Tableau 13.8. Instructions de rassemblement (Gather)

Recherche dans une table vectorisée

Les instructions de permutation peuvent être utilisées pour la consultation de tables si la table entière tient dans un ou quelques registres vectoriels. Si la table est trop volumineuse, il faut alors recourir aux instructions de regroupement (*gather*), plus lentes, pour effectuer la consultation.

Addition horizontale

Les exemples suivants montrent comment additionner tous les éléments d'un vecteur.

; **Exemple 13.15a.** Addition de 16 éléments au vecteur d'entiers non signés sur 8 bits.
; (SSE2)

```
movaps    xmm1, [source]      ; Vecteur source, 16 entiers 8 bits
pxor      xmm0, xmm0          ; 0
psadbw    xmm1, xmm0          ; Somme de 8 différences
pshufd     xmm0, xmm1, 0EH     ; Extraction des bits 64 à 127 de xmm1
paddb     xmm0, xmm1          ; Somme
movd      [sum], xmm0         ; Mémorisation de la somme
```

; **Exemple 13.15b.** Addition de 8 éléments dans un vecteur d'entiers
; 16 bits en utilisant l'instruction d'addition horizontale (SSSE3).

```
movaps    xmm0, [source]      ; Vecteur source, 8 entiers 16 bits
phaddw    xmm0, xmm0
phaddw    xmm0, xmm0
phaddw    xmm0, xmm0
movq      [sum], xmm0         ; Mémorisation de la somme
```

; L'instruction PHADDW est lente. L'alternative qui suit pourrait être plus rapide :

; **Exemple 13.15c.** Addition de 8 éléments dans un vecteur d'entiers
; 16 bits en évitant l'instruction d'addition horizontale (SSE2).

```
movaps    xmm0, [source]      ; Vecteur source, 8 entiers 16 bits
punpckhqdq xmm1, xmm0         ; Récupération des éléments 4, 5, 6 et 7.
paddw     xmm0, xmm1          ; Addition de 4 et 4 éléments.
pshufd     xmm1, xmm0, 1      ; Récupération des éléments 2, 3
paddw     xmm0, xmm1          ; Addition de 2 et 2 éléments.
pshufbw    xmm1, xmm0, 1      ; Récupération de l'élément 1
paddw     xmm0, xmm1          ; Addition d'un élément et d'un autre.
movq      [sum], xmm0         ; Mémorisation de la somme
```

; **Exemple 13.15d.** Addition de 8 huit éléments dans un vecteur de nombres virgule flottante (AVX)

```
vmovaps    ymm0, [source]     ; Vecteur source, 8 nombres à virgule flottante 32 bits
vextractf128 xmm1, ymm0, 1     ; Obtention de la moitié supérieure
vaddps     xmm0, xmm0, xmm1    ; Addition
vhaddps     xmm0, xmm0, xmm0
vhaddps     xmm0, xmm0, xmm0
vmoovsd     [sum], xmm0        ; Mémorisation de la somme
```

; L'instruction VHADDPS est lente. L'alternative qui suit pourrait être plus rapide :

; **Exemple 13.15e.** Addition de 8 huit éléments dans un vecteur de nombres
; à virgule flottante 32 bits en évitant l'instruction d'addition horizontale (AVX).

```
vmovaps    ymm0, [a]          ; Vecteur source, 8 nombres à virgule flottante 32 bits
vextractf128 xmm1, ymm0, 1     ; Récupération des éléments 4, 5, 6, 7
vaddps     xmm0, xmm0, xmm1    ; Addition de 4 éléments
vmovhlps    xmm1, xmm0, xmm0   ; Récupération des éléments 2, 3
vaddps     xmm0, xmm0, xmm1    ; Addition de 4 éléments
```

vshufps	xmm1, xmm0, xmm0, 1	; Récupération de l'élément 1
vaddss	xmm0, xmm0, xmm1	; Addition d'un élément
vmovsd	[b], xmm0	; Mémorisation de la somme

13.8 Génération de constantes

Il n'existe pas d'instruction permettant de charger directement une constante dans un registre vectoriel. La méthode standard consiste à charger la constante depuis la mémoire. C'est également l'approche la plus efficace lorsque les défauts de cache sont rares ; toutefois, si ces défauts sont fréquents, il peut être judicieux d'envisager des alternatives.

Une première solution consiste à copier la constante depuis une zone mémoire statique vers la pile, en dehors de la boucle la plus imbriquée. Un emplacement mémoire situé dans la pile est moins susceptible de provoquer des défauts de cache qu'un emplacement situé dans un segment de données constantes. Cette option peut cependant s'avérer impossible dans le cas de fonctions de bibliothèque.

Une seconde solution consiste à écrire la constante dans la pile à l'aide d'instructions sur entiers, puis à charger cette valeur depuis la pile vers le registre vectoriel.

Une troisième solution consiste à générer la constante en combinant astucieusement diverses instructions. Cette méthode n'utilise pas le cache de données mais occupe davantage d'espace dans le cache d'instructions. Le cache d'instructions est moins susceptible de provoquer des défauts de cache, car le code est contigu.

Les constantes peuvent être réutilisées à maintes reprises, tant que le registre n'est pas requis pour une autre tâche.

Le tableau 13.9 ci-dessous indique comment générer diverses constantes entières dans des registres XMM. La même valeur est générée dans tous les éléments du vecteur :

Création de constantes pour des vecteurs d'entiers dans des registres XMM

Valeur	8 bits	16 bits	32 bits	64 bits
0	pxor xmm0, xmm0	pxor xmm0, xmm0	pxor xmm0, xmm0	pxor xmm0, xmm0
1	pcmpeqw xmm0, xmm0 pabsb xmm0, xmm0	pcmpeqw xmm0, xmm0 psrlw xmm0, 15	pcmpeqd xmm0, xmm0 psrld xmm0, 31	pcmpeqw xmm0, xmm0 psrlq xmm0, 63
2	pcmpeqw xmm0, xmm0 pabsb xmm0, xmm0 paddb xmm0, xmm0	pcmpeqw xmm0, xmm0 psrlw xmm0, 15 psllw xmm0, 1	pcmpeqd xmm0, xmm0 psrld xmm0, 31 pslld xmm0, 1	pcmpeqw xmm0, xmm0 psrlq xmm0, 63 psllq xmm0, 1
3	pcmpeqw xmm0, xmm0 psrlw xmm0, 14 packuswb xmm0, xmm0	pcmpeqw xmm0, xmm0 psrlw xmm0, 14	pcmpeqd xmm0, xmm0 psrld xmm0, 30	pcmpeqw xmm0, xmm0 psrlq xmm0, 62
4	pcmpeqw xmm0, xmm0 pabsb xmm0, xmm0 psllw xmm0, 2	pcmpeqw xmm0, xmm0 psrlw xmm0, 15 psllw xmm0, 2	pcmpeqd xmm0, xmm0 psrld xmm0, 31 pslld xmm0, 2	pcmpeqw xmm0, xmm0 psrlq xmm0, 63 psllq xmm0, 2
-1	pcmpeqw xmm0, xmm0	pcmpeqw xmm0, xmm0	pcmpeqd xmm0, xmm0	pcmpeqw xmm0, xmm0
-2	pcmpeqw xmm0, xmm0 paddb xmm0, xmm0	pcmpeqw xmm0, xmm0 psllw xmm0, 1	pcmpeqd xmm0, xmm0 pslld xmm0, 1	pcmpeqw xmm0, xmm0 psllq xmm0, 1
Autre valeur	mov eax, value*01010101H movd xmm0, eax pshufd xmm0, xmm0, 0	mov eax, value*10001H movd xmm0, eax pshufd xmm0, xmm0, 0	mov eax, value movd xmm0, eax pshufd xmm0, xmm0, 0	mov rax, value movq xmm0, rax punpcklqdq xmm0, xmm0 (64 bit mode only)

Tableau 13.9. Générer des constantes de vecteur d'entiers

Le tableau 13.10 ci-dessous montre comment créer diverses constantes à virgule flottante dans les registres XMM. La même valeur est générée dans un ou tous les éléments du vecteur :

Création de constantes à virgule flottante dans les registres XMM

Valeur	scalaire simple	scalaire double	vecteur simple	vecteur double
0.0	xorps xmm0, xmm0	xorps xmm0, xmm0	xorps xmm0, xmm0	xorps xmm0, xmm0
0.5	pcmpeqw xmm0, xmm0 pslld xmm0, 26 psrld xmm0, 2	pcmpeqw xmm0, xmm0 psllq xmm0, 55 psrlq xmm0, 2	pcmpeqw xmm0, xmm0 pslld xmm0, 26 psrld xmm0, 2	pcmpeqw xmm0, xmm0 psllq xmm0, 55 psrlq xmm0, 2
1.0	pcmpeqw xmm0, xmm0 pslld xmm0, 25 psrld xmm0, 2	pcmpeqw xmm0, xmm0 psllq xmm0, 54 psrlq xmm0, 2	pcmpeqw xmm0, xmm0 pslld xmm0, 25 psrld xmm0, 2	pcmpeqw xmm0, xmm0 psllq xmm0, 54 psrlq xmm0, 2

1.5	pcmpeqw xmm0, xmm0 pslld xmm0, 24 psrld xmm0, 2	pcmpeqw xmm0, xmm0 psllq xmm0, 53 psrlq xmm0, 2	pcmpeqw xmm0, xmm0 pslld xmm0, 24 psrld xmm0, 2	pcmpeqw xmm0, xmm0 psllq xmm0, 53 psrlq xmm0, 2
2.0	pcmpeqw xmm0, xmm0 pslld xmm0, 31 psrld xmm0, 1	pcmpeqw xmm0, xmm0 psllq xmm0, 63 psrlq xmm0, 1	pcmpeqw xmm0, xmm0 pslld xmm0, 31 psrld xmm0, 1	pcmpeqw xmm0, xmm0 psllq xmm0, 63 psrlq xmm0, 1
-2.0	pcmpeqw xmm0, xmm0 pslld xmm0, 30	pcmpeqw xmm0, xmm0 psllq xmm0, 62	pcmpeqw xmm0, xmm0 pslld xmm0, 30	pcmpeqw xmm0, xmm0 psllq xmm0, 62
sign bit	pcmpeqw xmm0, xmm0 pslld xmm0, 31	pcmpeqw xmm0, xmm0 psllq xmm0, 63	pcmpeqw xmm0, xmm0 pslld xmm0, 31	pcmpeqw xmm0, xmm0 psllq xmm0, 63
not sign bit	pcmpeqw xmm0, xmm0 psrld xmm0, 1	pcmpeqw xmm0, xmm0 psrlq xmm0, 1	pcmpeqw xmm0, xmm0 psrld xmm0, 1	pcmpeqw xmm0, xmm0 psrlq xmm0, 1
Autre valeur (mode 32 bits)	mov eax, value movd xmm0, eax	mov eax, value>>32 movd xmm0, eax psllq xmm0, 32	mov eax, value movd xmm0, eax shufps xmm0, xmm0, 0	mov eax, value>>32 movd xmm0, eax pshufd xmm0, xmm0, 22H
Autre valeur (mode 64 bits)	mov eax, value movd xmm0, eax	mov rax, value movq xmm0, rax	mov eax, value movd xmm0, eax shufps xmm0, xmm0, 0	mov rax, value movq xmm0, rax shufpd xmm0, xmm0, 0

Tableau 13.10. Génération de constantes vectorielles à virgule flottante

Le “bit de signe” est une valeur dont le bit de signe est positionné à 1 et tous les autres bits sont à 0. Il est utilisé pour modifier ou définir le signe d'une variable. Par exemple, pour changer le signe d'un vecteur de nombres à double précision (doubles) dans xmm0 :

;**Exemple 13.16.** Changement du signe du vecteur 2*double
 pcmpeqw xmm7, xmm7 ; Tous les bits à 1
 psllq xmm7, 63 ; Décalage des 63 bits de poids faible (tous à 1) vers l'extérieur.
 xorpd xmm0, xmm7 ; Inversion du bit de signe de XMM0

Le “not sign bit” (bit de signe inversé) correspond à la valeur inversée du bit de signe. Il se caractérise par un bit de signe à 0 et tous les autres bits à 1. Cette valeur est utilisée pour obtenir la valeur absolue d'une variable. Par exemple, pour obtenir la valeur absolue d'un vecteur de nombres à double précision (doubles) situé dans le registre xmm0 :

;**Exemple 13.17.** Valeur absolue du vecteur de 2*doubles
 pcmpeqw xmm6, xmm6 ; Tous les bits à 1
 psrlq xmm6, 1 ; Décalage du bit de poids fort vers l'extérieur
 andpd xmm0, xmm6 ; Mise à zéro du bit de signe

La génération d'une valeur arbitraire en double précision en mode 32 bits est plus complexe. La méthode présentée dans le tableau 13.10 n'utilise que les 32 bits de poids fort de la représentation 64 bits du nombre, en supposant que les bits de poids faible de la partie fractionnaire sont nuls ou qu'une approximation est acceptable. Par exemple, pour générer la valeur “double” 9,25, on utilise d'abord un compilateur ou un assembleur pour déterminer que la représentation hexadécimale de 9,25 est 4022800000000000H. Les 32 bits de poids faible pouvant être ignorés, on peut procéder comme suit :

;**Exemple 13.18a.** Définition d'un vecteur 2*double sur une valeur arbitraire (mode 32 bits)
 mov eax, 40228000H ; Bits de poids fort (32 bits) de 9,25
 movd xmm0, eax ; Déplacement dans XMM0
 pshufd xmm0, xmm0, 22H ; Placement de la valeur dans les dwords 1 et 3.

En mode 64 bits, nous pouvons utiliser des registres d'entiers 64 bits :

;**Exemple 13.18b.** Définition d'un vecteur 2*double sur une valeur arbitraire (mode 64 bits)
 mov rax, 4022800000000000H ; Représentation complète de 9,25
 movq xmm0, rax ; Déplacement dans XMM0
 shufpd xmm0, xmm0, 0 ; Diffusion

Notez que certains assembleurs utilisent le nom très trompeur movd au lieu de movq pour l'instruction qui transfère 64 bits entre un registre d'usage général et un registre XMM.

13.9 Accès à des données non alignées et à des vecteurs partiels

Toutes les données lues ou écrites à l'aide de registres vectoriels doivent de préférence être alignées sur la taille du vecteur. Consultez la page 65 pour savoir comment aligner les données.

L'accès à des données non alignées entraîne une pénalité de performance sur les anciens processeurs, alors que les processeurs modernes traitent ces données presque aussi rapidement que les données alignées.

Il existe des situations où l'alignement n'est pas possible ; par exemple, lorsqu'une fonction de bibliothèque reçoit un pointeur vers un tableau sans savoir si celui-ci est aligné ou non. Les méthodes suivantes peuvent être utilisées pour lire des vecteurs non alignés :

Utilisation d'instructions de lecture non alignées

Les instructions MOVDBQ, MOVUPS, MOVUPD et LDDQU permettent toutes de lire des vecteurs non alignés. Ces instructions de lecture non alignée sont relativement lentes sur les anciens processeurs Intel et sur les processeurs Intel Atom, mais rapides sur les processeurs Intel de l'architecture Nehalem et ultérieurs, ainsi que sur les processeurs AMD et VIA.

```
; Exemple 13.19. Lecture de vecteur non aligné
; ESI contient un pointeur vers un tableau non aligné
movdbq xmm0, [esi] ; Lecture de vecteur non aligné
```

Sur les processeurs modernes, l'utilisation de l'instruction MOVDBQ (pour données non alignées) plutôt que MOVDBA (pour données alignées) n'entraîne aucune pénalité temporelle si les données sont effectivement alignées. Il est donc avantageux d'utiliser MOVDBQ lorsque l'on n'est pas certain de l'alignement des données.

Utilisation d'instructions préfixées par VEX

Les instructions dotées du préfixe VEX autorisent les opérandes mémoire non alignés, tandis que les mêmes instructions sans ce préfixe génèrent une exception si l'opérande n'est pas correctement aligné :

```
; Exemple 13.20. Accès VEX par rapport à l'accès non-VEX aux données non alignées
movups xmm1, [esi] ; l'opérande non aligné doit être lu en premier.
addps xmm0, xmm1
; Le préfixe VEX autorise les opérandes non alignés.
vaddps xmm0, [esi] ; opérande non aligné autorisé ici
```

Il ne faut jamais mélanger du code VEX et du code non-VEX si les registres YMM ou ZMM sont utilisés. Il est toutefois acceptable de mélanger des instructions AVX et AVX-512. Voir la [section 13.1](#).

Lectures se chevauchant partiellement

Effectuez la première lecture à partir de l'adresse non alignée et la lecture suivante à partir de la limite de 16 octets immédiatement supérieure. Les deux premières lectures pourront donc se chevaucher :

```
; Exemple 13.21. La première lecture non alignée chevauche la lecture alignée suivante
; ESI contient un pointeur vers un tableau non aligné
movdbq xmm1, [esi] ; Lecture du vecteur non aligné
add esi, 10H
and esi, -10H ; = limite 16 octets suivante la plus proche
movdbq xmm2, [esi] ; Lecture du vecteur aligné suivant
```

Ici, les données contenues dans xmm1 et xmm2 se chevaucheront partiellement si esi n'est pas divisible par 16. Cela ne fonctionne, bien entendu, que si l'algorithme qui suit accepte la présence de données redondantes. La dernière lecture vectorielle peut également chevaucher l'avant-dernière si la fin du tableau n'est pas alignée.

Lecture à partir de la limite de 16 octets précédente la plus proche

Il est possible de commencer la lecture à partir de la limite de 16 octets précédente la plus proche pour un tableau non aligné. Cela placera des données non pertinentes dans une partie du registre vectoriel, et ces données devront être ignorées. De même, la dernière lecture peut dépasser la fin du tableau jusqu'à la limite de 16 octets suivante :

```
; Exemple 13.22. Lecture à partir de la limite de 16 octets précédente la plus proche
; ESI contient un pointeur vers un tableau non aligné
mov eax, esi ; Copie du pointeur
and esi, -10H ; Arrondi à la valeur inférieure divisible par 16
and eax, 0FH ; Le tableau est décalé de cette valeur.
movdbq xmm1, [esi] ; Lecture à partir de la limite 16 octets précédente
movdbq xmm2, [esi+10H] ; Lecture du bloc suivant
```

Dans l'exemple ci-dessus, xmm1 contient eax octets de données parasites suivis de (16-eax) octets utiles. xmm2 contient les eax octets restants du vecteur, suivis de (16-eax) octets qui sont soit des données parasites, soit des données appartenant au vecteur suivant. La valeur contenue dans eax doit alors servir à masquer ou à ignorer la partie du registre contenant les données parasites.

Nous tirons ici parti du fait que les tailles de vecteurs, de lignes de cache et de pages mémoire sont toujours des puissances de 2. Les limites des lignes de cache et des pages mémoire coïncideront donc avec les limites des vecteurs. Bien que cette méthode conduise à lire certaines données non pertinentes, elle n'entraîne jamais le

chargement d'une ligne de cache inutile, car les lignes de cache sont toujours alignées sur un multiple de 16. La lecture de données non pertinentes n'engendre donc aucun surcoût. De plus, et c'est un point crucial, nous ne rencontrerons jamais d'erreur de lecture due à l'accès à des adresses mémoire inexistantes ; en effet, la mémoire de données est toujours allouée par pages de 4096 octets (soit 2^{12}) ou plus, de sorte qu'une lecture de vecteur aligné ne peut jamais chevaucher la limite d'une page mémoire.

Un exemple utilisant cette méthode est présenté dans l'exemple `strlenSSE2.asm` disponible dans l'annexe : www.agner.org/optimize/asmexamples.zip.

Combinaison de 2 parties de vecteur non alignées en un seul vecteur aligné

La méthode ci-dessus peut être étendue en combinant les parties valides de deux registres pour former un registre complet. Si l'on décale `xmm1` (dans l'exemple 13.22) vers la droite de la valeur de décalage (`eax`) et `xmm2` vers la gauche de (`16-eax`), les deux registres peuvent alors être fusionnés en un vecteur ne contenant que des données valides.

Malheureusement, il n'existe pas d'instruction permettant de décaler un registre vectoriel entier vers la droite ou vers la gauche d'un nombre variable de positions (par analogie avec `shr eax, cl`) ; il faut donc recourir à une instruction de permutation. L'exemple suivant utilise une instruction de permutation d'octets, `PSHUFb` avec des masques appropriés pour décaler un registre vectoriel vers la droite ou vers la gauche :

```
; Exemple 13.23. Combinaison de 2 parties non alignées en un seul vecteur.
; (Jeu d'instructions Supplementary-SSE3 requis)
; Cet exemple calcule la racine carrée de n nombres à virgule flottante contenus
; dans un tableau non aligné "src" et stocke le résultat dans un tableau aligné "dest".
; Code C++:
; const int n = 100;
; float * src;
; float dest[n];
; for (int i=0; i<n; i++) dest[i] = sqrt(src[i]);

; Définition des masques pour utiliser l'instruction PSHUFb comme instruction de décalage :
; Les 16 octets de SMask[16+a] décaleront de a octets vers la droite
; Les 16 octets de SMask[16-a] décaleront de a octets vers la gauche
```

section .data

align 64

SMask:

```
DB -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1
DB 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15
DB -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1
```

section .text

default rel

```
mov esi, [src]           ; Pointeur src non aligné
lea edi, [dest]          ; Tableau aligné dest
mov eax, esi
and eax, 0FH             ; Récupération du désalignement, a
movdqu xmm4, [SMask+10H+eax] ; Masque pour décalage à droite de a
movdqu xmm5, [SMask+eax]  ; Masque pour décalage à gauche de 16-a
and esi, -10H            ; Limite 16 octets précédente la plus proche
xor ecx, ecx             ; Compteur de boucle i = 0

L:                        ; Boucle
movdqa xmm1, [esi+ecx]    ; Lecture à partir de la limite précédente
movdqa xmm2, [esi+ecx+10H] ; Lecture du bloc suivant
pshufb xmm1, xmm4         ; décalage à droite de a
pshufb xmm2, xmm5         ; décalage à gauche de 16-a
por xmm1, xmm2            ; combinaison des blocs
sqrtps xmm1, xmm1         ; calcul de 4 racines carrées
movaps [edi+ecx], xmm1    ; Enregistrement du résultat aligné
add ecx, 10H              ; Boucle sur les 4 valeurs suivantes
cmp ecx, 400              ; 4*n
jb L ; Loop
```

Alignez `SMask` sur une frontière de 64 octets si possible, afin d'éviter les lectures à cheval sur une limite de ligne de cache. Cette opération est facilitée si la valeur du décalage d'alignement (dans le registre `eax`) est une cons-

tante connue. Le nombre d'instructions peut être réduit en utilisant l'instruction `PALIGNR` lorsque la valeur de décalage est constante.

Nous avons abordé plusieurs méthodes pour lire des vecteurs non alignés. Il convient désormais d'examiner comment écrire des vecteurs à des adresses non alignées. Certaines de ces méthodes sont pratiquement identiques.

Utilisation d'instructions d'écriture non alignées

Les instructions `movdqu`, `movups` et `movupd` permettent toutes d'écrire des vecteurs non alignés. Ces instructions d'écriture non alignée sont relativement lentes sur les anciens processeurs Intel, mais rapides sur les processeurs Intel de l'architecture Nehalem et ultérieurs, ainsi que sur les processeurs AMD et VIA contemporains.

```
; Exemple 13.24. Écriture d'un vecteur non aligné
; EDI contient un pointeur vers un tableau non aligné
movdqu [edi], xmm0      ; Écriture d'un vecteur non aligné
```

Écritures se chevauchant partiellement

Effectuez la première écriture à l'adresse non alignée et la suivante à la limite de 16 octets immédiatement supérieure. Les deux premières écritures risquent donc de se chevaucher :

```
; Exemple 13.25. La première écriture non alignée chevauche l'écriture alignée suivante.
; EDI contient un pointeur vers un tableau non aligné
movdqu [edi], xmm1      ; Écriture d'un vecteur non aligné
add edi, 10H
and edi, 0FH             ; = limite 16 octets suivante la plus proche
movdqa [edi], xmm2      ; Écriture du vecteur aligné suivant
```

Ici, les données de `xmm1` et `xmm2` se chevaucheront partiellement si `edi` n'est pas divisible par 16. Cela ne fonctionne évidemment que si l'algorithme est capable de générer les données qui se chevauchent. La dernière écriture vectorielle peut également chevaucher l'avant-dernière si la fin du tableau n'est pas alignée. Voir l'exemple `memset` dans l'annexe www.agner.org/optimize/asmexamples.zip.

Écrire du début et de la fin séparément

Utilisez des instructions non vectorielles pour effectuer l'écriture depuis le début d'un tableau non aligné jusqu'à la première limite de 16 octets, puis de la dernière limite de 16 octets jusqu'à la fin du tableau. Consultez l'exemple `memcpy` dans l'annexe disponible sur www.agner.org/optimize/asmexamples.zip.

Utilisation de l'écriture masquée

Les instructions `VPMASKMOVD` (AVX2) et `VMASKMOVPS` (AVX), etc. peuvent être utilisées pour écrire dans la première partie d'un tableau non aligné jusqu'à la première limite de 16 octets ainsi que dans la dernière partie après la dernière limite de 16 octets.

Les versions non VEX des instructions de déplacement masquées, telles que `MASKMOVDQU`, sont extrêmement lentes car elles contournent le cache et écrivent dans la mémoire principale (ce qu'on appelle une écriture non temporelle). Les versions VEX sont plus rapides que les versions non VEX sur les processeurs Intel, mais pas les processeurs AMD Bulldozer et Piledriver. Ces instructions doivent absolument être évitées.

Les versions de mouvements masqués dans AVX512 sont rapides et utiles.

13.10 Opérations vectorielles dans les registres d'usage général

Il est parfois possible de traiter des données multiples (*packed data*) dans des registres à usage général de 32 ou 64 bits. Cette méthode peut être utilisée sur des processeurs où les opérations sur entiers sont plus rapides que les opérations vectorielles, ou lorsque les opérations vectorielles appropriées ne sont pas disponibles.

Un registre de 64 bits peut contenir deux entiers de 32 bits, quatre entiers de 16 bits, huit entiers de 8 bits ou 64 booléens. Lors de calculs sur des entiers multiples dans des registres de 32 ou 64 bits, il convient de veiller tout particulièrement à éviter qu'une retenue provenant d'un opérande ne se propage à l'opérande suivant en cas de risque de dépassement de capacité (*overflow*). Une telle retenue ne se produit pas si tous les opérandes sont positifs et suffisamment petits pour exclure tout dépassement. Par exemple, vous pouvez juxtaposer quatre entiers positifs de 16 bits dans le registre `RAX` et utiliser l'instruction `ADD RAX, RBX` au lieu de `PADDW MM0, MM1` si vous avez la certitude qu'aucun dépassement ne surviendra. Si le risque de retenue ne peut être écarté, il est nécessaire de masquer le bit de poids fort, comme dans l'exemple suivant qui ajoute 2 à chacun des quatre octets du registre `EAX` :

```
; Exemple 13.26. Addition de vecteurs d'octets dans un registre 32 bits
mov eax, [esi]      ; lecture d'un opérande de 4 octets
mov ebx, eax        ; copie dans EBX
and eax, 7f7f7f7fh  ; récupération des 7 bits de poids faible de chaque octet dans EAX.
```

```

xor ebx, eax          ; récupération du bit de poids fort de chaque octet
add eax, 02020202h    ; Ajout de la valeur souhaitée aux 4 octets.
xor eax, ebx          ; combinaison des bits à nouveau
mov [edi], eax        ; mémorisation du résultat

```

Ici, le bit de poids fort de chaque octet est masqué afin d'éviter toute retenue d'un octet vers le suivant lors de l'addition. Le code utilise l'opération XOR plutôt que l'addition pour rétablir le bit de poids fort, évitant ainsi la propagation d'une retenue. Si le second opérande présente également un bit de poids fort positionné à 1, celui-ci doit aussi être masqué. Aucun masquage n'est nécessaire si aucun des deux opérandes n'a son bit de poids fort positionné à 1.

Il est également possible de rechercher un octet spécifique. Ce code en langage C illustre le principe en vérifiant si un entier 32 bits contient au moins un octet nul :

```

// Exemple 13.27. Renvoie une valeur non nulle si le dword contient un octet nul.
inline int dword_has_nullbyte(int w) {
    return ((w - 0x01010101) & ~w & 0x80808080);}

```

Le résultat est nul si les quatre octets de `w` sont non nuls. Le résultat présentera la valeur 0x80 à la position du premier octet nul. Si plusieurs octets sont nuls, les octets suivants ne sont pas nécessairement signalés. (Cette méthode a été inventée par Alan Mycroft et publiée en 1987. J'ai publié la même méthode en 1996 dans la première version de ce manuel, sans savoir que Mycroft avait fait cette découverte avant moi).

Ce principe peut être utilisé pour déterminer la longueur d'une chaîne terminée par un caractère nul en recherchant le premier octet nul. C'est plus rapide que l'utilisation de `REPNE SCASB` :

```

; Exemple 13.28. Procédure strlen optimisée (32 bits) :
_strlen PROC NEAR
; extern "C" int strlen (const char * s);

    push ebx          ; EBX doit être sauvegardé
    mov ecx, [esp+8]   ; obtention d'un pointeur vers une chaîne de caractères
    mov eax, ecx       ; copie du pointeur
    and ecx, 3         ; 2 bits de poids faible, vérification de l'alignement
    jz L2              ; s est aligné sur 4. Passer à la boucle.
    and eax, -4        ; alignement du pointeur sur 4
    mov ebx, [eax]     ; lecture à partir de la limite précédente
    shl ecx, 3         ; *8 = déplacement en bits
    mov edx, -1
    shl edx, cl        ; constitution d'un masque d'octet
    not edx            ; masque = 0FFH pour les octets erronés
    or ebx, edx        ; masquage des octets erronés

; Vérification si les quatre premiers octets sont nuls.
    lea ecx, [ebx-01010101H] ; soustraction de 1 à chaque octet
    not ebx            ; inversion de tous les octets
    and ecx, ebx       ; ainsi que ces deux-là
    and ecx, 80808080H ; test de tous les bits de signe sur les 4 octets
    jnz L3             ; saut car aucun octet n'est trouvé

; Boucle principale, lecture de 4 octets alignés
L1:  add eax, 4         ; incrémentation du pointeur
L2:  mov ebx, [eax]     ; lecture de 4 octets de la chaîne
    lea ecx, [ebx-01010101H] ; soustraction de 1 à chaque octet
    not ebx            ; inversion de tous les octets
    and ecx, ebx       ; ainsi que ces deux-là
    and ecx, 80808080H ; test de tous les bits de signe sur les 4 octets
    jz L1              ; aucun octet n'est trouvé, poursuite de la boucle
L3:  bsf ecx, ecx       ; identification du bit à 1 le plus à droite
    shr ecx, 3         ; division par 8 = index d'octet
    sub eax, [esp+8]   ; soustraction de l'adresse de début
    add eax, ecx       ; ajouter d'un index à l'octet
    pop ebx            ; restauration de EBX
    ret               ; la valeur de retour est dans EAX
_strlen ENDP

```

La vérification de l'alignement garantit que nous ne lisons qu'à partir d'adresses alignées sur 4 octets. La fonction peut lire aussi bien avant le début de la chaîne qu'après sa fin ; toutefois, comme toutes les lectures sont alignées, nous ne franchirons inutilement aucune limite de cache ni de page mémoire. Plus important encore, nous

ne provoquerons aucune erreur de page (*page fault*) en lisant au-delà de la mémoire allouée, car les limites de page sont toujours alignées sur des multiples de 2^{12} (4096) ou plus.

Si le jeu d'instructions SSE2 est disponible, il peut être plus rapide d'utiliser des instructions XMM en appliquant la même astuce d'alignement. L'exemple 13.28, ainsi qu'une fonction similaire utilisant des registres XMM, sont fournis dans l'annexe disponible sur www.agner.org/optimize/asmexamples.zip et dans ma bibliothèque d'assembleur sur www.agner.org/optimize/asmlib.zip.

D'autres fonctions courantes recherchant un octet spécifique, telles que `strcpy`, `strchr` ou `memchr`, peuvent utiliser la même astuce. Pour rechercher un octet différent de zéro, il suffit d'effectuer une opération XOR avec l'octet recherché, comme l'illustre ce code C :

```
// Exemple 13.29. Renvoi d'une valeur non nulle si l'octet b est contenu dans le mot double w
inline int dword_has_byte(int w, unsigned char b) {
    w ^= b * 0x01010101;
    return ((w - 0x01010101) & ~w & 0x80808080);}
```

Notez que si vous effectuez une recherche en partant de la fin (par exemple avec `strrchr`), la méthode ci-dessus indiquera la position de la première occurrence de `b` dans `w` en cas de présence de plusieurs occurrences.

14 Multithread

La puissance de calcul pouvant être obtenue à partir d'un processeur unique est limitée. C'est pourquoi de nombreux systèmes informatiques modernes sont dotés de plusieurs cœurs de processeur. Pour exploiter ces multiples cœurs, on répartit la tâche de calcul entre plusieurs threads. Le nombre optimal de threads correspond généralement au nombre de cœurs du processeur. Idéalement, la charge de travail doit être répartie équitablement entre les threads.

Le multithreading est utile lorsque le code présente un parallélisme intrinsèque qui est à gros grain (*coarse-grained*). Le multithreading ne peut pas être utilisé pour un parallélisme à grain fin (*fine-grained*), car le démarrage et l'arrêt des threads, ainsi que leur synchronisation, entraînent des coûts supplémentaires considérables.

La communication entre les threads peut également être assez coûteuse, bien que ces coûts soient réduits sur les processeurs plus récents.

Il est préférable de diviser le travail de calcul en threads au niveau le plus élevé possible. Si la boucle externe peut être parallélisée, elle peut alors être répartie en une boucle pour chaque thread, chacun exécutant sa part de l'ensemble du travail.

Le stockage local aux threads (Thread-Local Storage) doit de préférence utiliser la pile (*stack*). L'utilisation d'une mémoire statique locale aux threads est inefficace et doit être évitée.

14.1 Multithreading simultané

La plupart des microprocesseurs modernes possèdent plusieurs cœurs de processeur, ce qui leur permet d'exécuter plusieurs threads simultanément. Par ailleurs, de nombreux processeurs Intel et AMD sont capables d'exécuter deux threads par cœur. Les processeurs Knights Corner et Knights Landing peuvent même en exécuter quatre. "Hyper-Threading" est le terme utilisé par Intel pour désigner le multithreading simultané.

Deux threads (ou plus) s'exécutant sur un même cœur entrent inévitablement en concurrence pour les mêmes ressources, telles que la mémoire cache, le décodeur d'instructions et les unités d'exécution. Si l'une de ces ressources partagées constitue un facteur limitant pour les performances d'un thread unique, l'exécution de deux threads par cœur n'offre aucun avantage. Au contraire, chaque thread risque de s'exécuter à une vitesse inférieure à la moitié de sa vitesse nominale en raison des expulsions de cache et d'autres conflits d'accès aux ressources. En revanche, si une part importante du temps d'exécution est consacrée à la gestion des défauts de cache, des erreurs de prédiction de branchement ou de longues chaînes de dépendance, chaque thread fonctionnera à une vitesse supérieure à la moitié de celle d'un thread unique. Dans ce cas, le recours au multithreading simultané présente un avantage, sans toutefois doubler les performances. Un thread partageant les ressources d'un cœur avec un autre thread s'exécutera toujours plus lentement qu'un thread occupant seul ce cœur.

Plus la capacité d'un cœur de processeur est élevée, plus le multithreading simultané présente un avantage. Le processeur Knights Landing possède une capacité relativement faible ; il est donc rarement avantageux d'y exécuter deux threads — et encore moins quatre. Le multithreading simultané s'avère plus utile sur l'ancien processeur Knights Corner, car celui-ci ne dispose pas de capacités d'exécution dans le désordre (*out-of-order processing*).

Il peut s'avérer nécessaire d'effectuer des tests pour déterminer si l'utilisation du multithreading simultané est avantageuse ou non pour une application donnée.

L'un des inconvénients du multithreading simultané est qu'un thread de faible priorité peut accaparer des ressources au détriment d'un thread de haute priorité s'exécutant sur le même cœur de processeur. Les systèmes d'exploitation actuels ne gèrent pas toujours ce problème de manière optimale.

Consultez le manuel 1, “Optimizing software in C++”, pour plus de détails sur le multithreading et le multithreading simultané.

15 Ordonnancement du processeur

On souhaite souvent tirer parti de jeux d'instructions avancés tels qu'AVX2 ou AVX512. Toutefois, un problème survient si l'on veut que le même programme puisse s'exécuter sur des microprocesseurs dépourvus de ces jeux d'instructions. La solution consiste à disposer de plusieurs versions de la partie la plus critique du programme, chacune optimisée pour un jeu d'instructions différent. La version du code appropriée est sélectionnée lors de l'exécution, après vérification des capacités du processeur hôte. C'est ce que l'on appelle la répartition automatique selon le processeur (*CPU dispatching*).

Il ne vaut pas la peine d'optimiser le code pour chaque modèle de processeur spécifique, car le logiciel finira souvent par s'exécuter sur un modèle qui n'existait pas au moment de l'écriture du code. Il est préférable d'écrire une version optimisée pour un jeu d'instructions récent que le programme peut exploiter, ainsi qu'une ou plusieurs versions pour des jeux d'instructions plus anciens. Consultez le chapitre 13 du manuel 1, “Optimizing software in C++”, qui explique les pièges liés à une répartition spécifique au modèle de processeur.

La répartition selon le processeur peut être implémentée à l'aide d'instructions de branchement ou d'un pointeur de fonction, comme le montre l'exemple suivant.

Exemple 15.1. Fonction avec répartition sur processeur

```
section .data
MyFunctionPoint: DQ MyFunctionDispatch      ; Pointeur de fonction

section .text
default rel
extern InstructionSet

MyFunction:
    ; Saut via le pointeur. Le pointeur de code pointe initialement
    ; vers MyFunctionDispatch. MyFunctionDispatch modifie le pointeur
    ; pour qu'il pointe vers la version appropriée de MyFunction.
    ; Lors du prochain appel à MyFunction, le programme saute directement
    ; à la bonne version de la fonction.

    jmp [MyFunctionPoint]

; Code pour chaque version :

MyFunctionAVX512:
    ; Version AVX512 de MyFunction
    ...
    ret

MyFunctionAVX2:
    ; Version AVX2 de MyFunction
    ...
    ret

MyFunctionSSE2:
    ; Version générique/SSE2 de MyFunction
    ; (tous les microprocesseurs x64 prennent également en charge SSE2.
    ; Si le programme s'exécute en mode 32 bits, une version générique
    ; compatible 80386 pourrait être nécessaire).
    ...
    ret

MyFunctionDispatch:
    ; Détectez quel jeu d'instructions est pris en charge.
    ; La fonction InstructionSet est dans asmlib

    push rcx                ; Sauvegardez tous les registres susceptibles
                            ; d'être utilisés pour les paramètres de MyFunction.

    push rdx
    call InstructionSet      ; EAX indique le jeu d'instructions du processeur.
    lea rdx, [MyFunctionSSE2]
    cmp eax, 13             ; EAX >= 13 si AVX2
```

```

    jb DispEnd
    lea rdx, [MyFunctionAVX2]
    cmp eax, 16          ; EAX >= 16 si AVX512BW/DQ/VL
    jb DispEnd
    lea rdx, [MyFunctionAVX512]

DispEnd:
    ; Enregistrement du pointeur vers la version appropriée de MyFunction
    mov [MyFunctionPoint], rdx
    mov rax, rdx          ; adresse de la fonction
    pop rdx               ; restauration des registres
    pop rcx
    jmp rax               ; saut à cette version de la fonction

```

La fonction `InstructionSet` détecte quel jeu d'instructions est pris en charge. Cette fonction est fournie dans la bibliothèque téléchargeable à l'adresse www.agner.org/optimize/asmlib.zip. La plupart des systèmes d'exploitation disposent également de fonctions à cet effet.

15.1 Vérification de la prise en charge des registres XMM, YMM et ZMM par le système d'exploitation

Malheureusement, les informations obtenues via l'instruction `CPUID` ne suffisent pas à déterminer si l'utilisation des registres vectoriels est possible. Le système d'exploitation doit sauvegarder ces registres lors d'un changement de tâche et les restaurer lorsque la tâche reprend. Le microprocesseur peut désactiver l'utilisation des registres vectoriels afin d'empêcher leur emploi sous d'anciens systèmes d'exploitation qui ne les sauvegardent pas. Les systèmes d'exploitation prenant en charge les registres XMM doivent positionner le bit 9 du registre de contrôle CR4 pour activer ces registres et signaler leur capacité à les sauvegarder et à les restaurer lors des changements de tâche (la sauvegarde et la restauration des registres sont d'ailleurs plus rapides lorsque les registres XMM sont activés).

Malheureusement, le registre CR4 ne peut être lu qu'en mode privilège. Les programmes d'application ont donc du mal à déterminer s'ils sont autorisés ou non à utiliser les registres XMM. Selon la documentation officielle d'Intel, le seul moyen pour une application de savoir si le système d'exploitation prend en charge les registres XMM consiste à tenter d'exécuter une instruction XMM et à vérifier si une exception de type “code opération invalide” (*invalid opcode*) se produit. Cette approche est absurde, car tous les systèmes d'exploitation, compilateurs et langages de programmation ne permettent pas aux applications d'intercepter ce type d'exception. L'avantage lié à l'utilisation des registres XMM s'annule totalement s'il est impossible de savoir si l'on peut les employer sans provoquer le plantage du logiciel.

Ces problèmes sérieux m'ont amené à rechercher une méthode alternative pour vérifier si le système d'exploitation prend en charge l'utilisation des registres XMM et, heureusement, j'ai trouvé une méthode qui fonctionne de manière fiable. Si les registres XMM sont activés, alors les instructions `FXSAVE` et `FXRSTOR` peuvent lire et modifier les registres XMM. Si les registres XMM sont désactivés, alors `FXSAVE` et `FXRSTOR` ne peuvent pas accéder à ces registres. Il est donc possible de vérifier si les registres XMM sont activés, en essayant de lire et d'écrire ces registres avec `FXSAVE` et `FXRSTOR`. Les sous-programmes de www.agner.org/optimize/asmlib.zip utilisent cette méthode. Ces sous-programmes peuvent être appelés depuis l'assembleur ainsi que depuis des langages de haut niveau et fournissent un moyen simple de détecter si les registres XMM peuvent être utilisés.

Afin de vérifier que cette méthode de détection fonctionne correctement avec tous les microprocesseurs, j'ai d'abord consulté différents manuels. La version 1999 du manuel du développeur de logiciels d'Intel indique à propos de l'instruction `FXRSTOR` : “*Les champs d'extension Streaming SIMD dans l'image de sauvegarde (XMM0-XMM7 et MXCSR) ne seront pas chargés dans le processeur si le bit CR4.OSFXSR n'est pas défini.*” Le manuel du programmeur AMD dit en fait la même chose. Cependant, la version 2003 du manuel d'Intel indique que ce comportement dépend de l'implémentation. Afin de clarifier cela, j'ai contacté le support technique Intel et j'ai obtenu cette réponse : “*Si le bit OSFXSR dans CR4 n'est pas défini, alors les registres XMMx ne sont pas restaurés lorsque FXRSTOR est exécuté.*” Ils ont en outre confirmé que cela est vrai pour toutes les versions des microprocesseurs Intel et toutes les mises à jour du microcode. Je considère cela comme une garantie d'Intel que ma méthode de détection fonctionnera sur tous les microprocesseurs Intel. Nous pouvons également compter sur le bon fonctionnement de la méthode sur les processeurs AMD puisque le manuel AMD est sans ambiguïté sur cette question. Il semble sûr de pouvoir compter sur cette méthode pour qu'elle fonctionne correctement également sur les futurs microprocesseurs, car tout microprocesseur s'écartant de la spécification ci-dessus introduirait un problème de sécurité et empêcherait d'exécuter les programmes existants. La compatibilité avec les programmes existants est une grande préoccupation pour les producteurs de microprocesseurs.

La méthode de détection recommandée dans les manuels d'Intel présente l'inconvénient de dépendre de la capacité du compilateur et du système d'exploitation à intercepter les exceptions liées aux codes opération (opcodes) invalides. Ainsi, une application Windows utilisant la méthode de détection d'Intel devrait être testée sur tous les

systèmes d'exploitation compatibles, y compris les divers émulateurs Windows fonctionnant sous d'autres systèmes d'exploitation. Ma méthode de détection ne présente pas cet inconvénient, car elle est indépendante du compilateur et du système d'exploitation. Elle offre en outre l'avantage de faciliter la programmation modulaire : un module, une bibliothèque de sous-programmes ou une DLL utilisant des instructions XMM peut intégrer la procédure de détection, de sorte que la prise en charge des instructions XMM ne concerne pas le programme appelant — lequel peut même être écrit dans un langage de programmation différent. Certains systèmes d'exploitation fournissent des fonctions permettant d'identifier les jeux d'instructions pris en charge, mais la méthode susmentionnée, elle, ne dépend pas du système d'exploitation.

Il est plus simple de vérifier la prise en charge par le système d'exploitation des registres YMM et ZMM. Exécutez l'instruction `CPUID` avec `eax = 1`. Vérifiez que les bits 27 et 28 du registre `ecx` sont tous deux à 1 (flags de fonctionnalité `OSXSAVE` et `AVX`). Si tel est le cas, exécutez l'instruction `XGETBV` avec `ecx = 0` pour obtenir le masque `XFEATURE_ENABLED_MASK`. Vérifiez que les bits 1 et 2 du registre `eax` sont tous deux activés (prise en charge des états XMM et YMM). Si c'est le cas, l'utilisation des registres YMM est sans risque. Les registres ZMM sont activés si les bits 5, 6 et 7 sont également tous activés.

Les explications ci-dessus s'appuient sur les documents suivants :

Intel application note AP-900: "Identifying support for Streaming SIMD Extensions in the Processor and Operating System". 1999.

Intel application note AP-485: "Intel Processor Identification and the CPUID Instruction". 2002.
"Aria-ItalicMT 99.

"IA-32 Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference", 2003.

"Intel 64 and IA-32 Architectures Software Developer's Manual", 2018.

"AMD64 Architecture Programmer's Manual, Volume 4: 128-Bit Media Instructions", 2003.

"Intel Advanced Vector Extensions Programming Reference", 2008, 2010.

16 Instructions problématiques

16.1 Instruction LEA (tous les processeurs)

L'instruction `LEA` est utile à bien des égards, car elle permet d'effectuer une opération de décalage, deux additions et un transfert de données en une seule instruction. Exemple :

```
; Exemple 16.1a. Instruction LEA
lea rax, [rbx+8*rcx-1000]
```

est beaucoup plus rapide que

```
; Exemple 16.1b
mov rax, rcx
shl rax, 3
add rax, rbx
sub rax, 1000
```

Une utilisation courante de l'instruction `LEA` consiste à effectuer une addition impliquant trois registres : `lea rax, [rbx+rcx]`. L'instruction `LEA` peut également servir à réaliser une addition ou un décalage sans modifier les flags.

Les processeurs ne disposent d'aucun mode d'adressage documenté utilisant uniquement un registre d'index avec facteur d'échelle. Par conséquent, une instruction telle que `lea rax, [rbx*2]` est en réalité codée sous la forme `lea rax, [rbx*2+00000000H]`, avec un déplacement immédiat de 4 octets. La taille de cette instruction peut être réduite en écrivant `lea rax, [rbx+rbx]`. Si vous disposez d'un registre contenant la valeur zéro (comme un compteur de boucle après l'exécution de celle-ci), vous pouvez l'utiliser comme registre de base pour réduire la taille du code :

```
; Exemple 16.2. Instruction LEA sans pointeur de base
lea rax, [rbx*4] ; 8 octets
lea rax, [rcx+rbx*4]; 4 octets
```

La taille des registres de base et d'index peut être modifiée à l'aide d'un préfixe de taille d'adresse. La taille du registre de destination peut être modifiée à l'aide d'un préfixe de taille d'opérande (voir "Préfixes", page 19). Si la taille de l'opérande est inférieure à la taille de l'adresse, le résultat est tronqué. Si la taille de l'opérande est supérieure à la taille de l'adresse, le résultat fait l'objet d'une extension par zéros.

La version la plus courte de l'instruction `LEA` en mode 64 bits utilise une taille d'opérande de 32 bits et une taille d'adresse de 64 bits (par exemple, `LEA EAX, [RBX+RCX]` ; voir page 60). Utilisez cette version lorsque le résultat est garanti inférieur à 2^{32} ; dans ce cas, la partie supérieure des registres `RBX` et `RCX` est ignorée. Utilisez la version avec un registre de destination de 64 bits pour le calcul d'adresse en mode 64 bits lorsque l'adresse est susceptible de dépasser 2^{32} .

Sur certains processeurs, l'instruction LEA est plus lente qu'une addition. Les formes plus complexes de LEA, incluant un facteur d'échelle et un déplacement, sont plus lentes que la forme simple sur de nombreux processeurs ; consultez le manuel 4 "Instruction tables" pour plus de détails concernant chaque processeur.

En mode 32 bits, la version privilégiée utilise des opérandes et des adresses de 32 bits. L'instruction LEA avec des opérandes de 16 bits est lente sur les processeurs AMD. Il convient d'éviter l'utilisation de LEA avec des adresses de 16 bits en mode 32 bits, car le décodage du préfixe de taille d'adresse est lent sur de nombreux processeurs.

L'instruction LEA peut également être utilisée en mode 64 bits pour charger une adresse relative au RIP. Une adresse relative au RIP ne peut pas être combinée avec des registres de base ou d'index.

16.2 INC et DEC

Les instructions INC et DEC ne modifient pas le flag de retenue (*carry flag*), mais elles modifient les autres drapeaux arithmétiques. L'écriture dans une seule partie du registre des flags entraîne une micro-opération (μ op) supplémentaire sur certains processeurs. Cela peut provoquer des délais d'attente liés à une mise à jour partielle des flags sur certains anciens processeurs Intel si une instruction ultérieure lit le flag de retenue ou l'ensemble des bits de flags. Sur tous les processeurs, cela peut engendrer une fausse dépendance vis-à-vis du flag de retenue d'une instruction précédente.

Utilisez ADD et SUB pour optimiser la vitesse. Utilisez INC et DEC pour optimiser la taille du code ou lorsqu'aucune pénalité n'est à prévoir.

16.3 XCHG (tous les processeurs)

L'instruction XCHG register, [memory] est dangereuse. Elle comporte toujours un préfixe LOCK implicite qui impose une synchronisation avec les autres processeurs ou cœurs. Cette instruction est donc très coûteuse en temps d'exécution et doit toujours être évitée, sauf si ce verrouillage est délibérément recherché.

L'instruction XCHG utilisant des opérandes de type registre peut s'avérer utile pour optimiser la taille du code, comme expliqué à la page 57.

16.4 Rotation via la retenue (tous les processeurs)

Les instructions RCR et RCL utilisant CL ou un compteur différent de 1 sont lentes sur tous les processeurs et doivent être évitées.

16.5 Test de bit (tous les processeurs)

Sur les processeurs plus anciens, il est préférable de remplacer les instructions BT, BTC, BTR et BTS par des instructions telles que TEST, AND, OR, XOR ou des décalages. Il convient d'éviter les tests de bits avec un opérande mémoire sur les processeurs Intel. Les instructions BTC, BTR et BTS utilisent 2 μ ops sur les processeurs AMD. Les instructions de test de bits sont utiles pour optimiser la taille du code.

16.6 LAHF et SAHF (tous les processeurs)

LAHF est lent sur P4 et P4E. Utilisez plutôt SETcc pour stocker la valeur d'un flag.

SAHF est lent sur les processeurs P4E et AMD plus anciens. Utilisez plutôt TEST pour tester un bit dans AH. Utilisez FCOMI si disponible en remplacement de la séquence FCOM / FNSTSW AX / SAHF.

LAHF et SAHF ne sont pas disponibles en mode 64 bits sur certains des processeurs Intel 64 bits les plus anciens.

16.7 Multiplication d'entiers (tous les processeurs)

Une multiplication d'entiers nécessite entre 3 et 14 cycles d'horloge, selon le processeur. Il est donc souvent avantageux de remplacer une multiplication par une constante par une combinaison d'autres instructions telles que SHL, ADD, SUB et LEA. Par exemple, l'instruction IMUL EAX, 5 peut être remplacée par LEA EAX, [RAX+4*RAX] en mode 64 bits ou par LEA EAX, [EAX+4*EAX] en mode 32 bits.

16.8 Division (tous les processeurs)

La division entière comme la division en virgule flottante sont très coûteuses en temps de calcul sur tous les processeurs. Diverses méthodes permettant de réduire le nombre de divisions sont exposées dans le manuel 1 : "Optimizing software in C++". Plusieurs méthodes pour optimiser le code contenant des divisions sont proposées ci-dessous.

Division entière par une puissance de 2 (tous les processeurs)

La division entière par une puissance de deux peut être effectuée par un décalage vers la droite. Division d'un entier non signé par 2^N :

; **Exemple 16.3.** Division de l'entier non signé dans EAX par 2^N
shr eax, N

Division d'un entier signé par 2^N :

```
; Exemple 16.4. Division de l'entier signé dans EAX par  $2^N$ 
cdq
and edx, (1 shl N) - 1 ; (Ou: shr edx, 32-N)
add eax, edx
sar eax, N
```

Il est évidemment préférable d'utiliser la version non signée si l'on a la certitude que le dividende n'est pas négatif.

Division entière par une constante (tous les processeurs)

Il est possible de diviser un nombre à virgule flottante par une constante en le multipliant par son inverse. Pour effectuer la même opération avec des entiers, il faut multiplier l'inverse par 2^n , puis décaler le produit vers la droite de n positions. Il existe divers algorithmes permettant de déterminer une valeur appropriée de n et de compenser les erreurs d'arrondi. L'algorithme décrit ci-dessous a été mis au point par le Norvégien Terje Mathisen et n'a pas été publié ailleurs. D'autres méthodes sont présentées dans l'ouvrage "Hacker's Delight" de Henry S. Warren (Addison-Wesley, 2003) ainsi que dans l'article de T. Granlund et P. L. Montgomery intitulé "Division by Invariant Integers Using Multiplication", paru dans les actes de la conférence SIGPLAN 1994 sur la conception et l'implémentation des langages de programmation (*Programming Language Design and Implementation*).

L'algorithme suivant donne le résultat correct pour la division d'entiers non signés avec troncature, c'est-à-dire le même résultat que l'instruction DIV :

$b = (\text{nombre de bits significatifs dans } d) - 1$

$r = w + b$

$f = 2^r / d$

Si f est un entier, alors d est une puissance de 2: aller au cas A.

Si f n'est pas un entier, alors vérifier si la partie fractionnaire de f est $< 0,5$.

Si la partie fractionnaire de $f < 0,5$: aller au cas B.

Si la partie fractionnaire de $f > 0,5$: aller au cas C.

cas A ($d = 2^b$):

result = $x \text{ SHR } b$

cas B (partie fractionnaire de $f < 0,5$):

Arrondir f à l'entier inférieur le plus proche.

result = $((x+1) * f) \text{ SHR } r$

cas C (partie fractionnaire de $f > 0,5$):

Arrondir f à l'entier supérieur le plus proche

result = $(x * f) \text{ SHR } r$

Exemple:

Supposons que vous vouliez diviser par 5.

$5 = 101\text{B}$.

$w = 32$.

$b = (\text{nombre de chiffres binaires significatifs}) - 1 = 2$

$r = 32 + 2 = 34$

$f = 2^{34} / 5 = 3435973836,8 = 0\text{CCCCCCCC.CCC...}(\text{hexadécimal})$

La partie fractionnaire est supérieure à 0,5 : utilisez donc le cas C.

Soit, arrondir 0CCCCCCCCDH à l'entier supérieur le plus proche.

Le code suivant divise EAX par 5 et renvoie le résultat dans EDX :

```
; Exemple 16.5a. Division de l'entier non signé dans EAX par 5
mov edx, 0CCCCCCDH
mul edx
shr edx, 2
```

Après la multiplication, EDX contient le produit décalé de 32 positions vers la droite. Dans la mesure où $r = 34$, il faut effectuer un décalage supplémentaire de 2 positions pour obtenir le résultat. Pour diviser par 10, il suffit de modifier la dernière ligne en `SHR EDX, 3`.

Dans le cas B, nous aurions :

```
; Exemple 16.5b. Division de l'entier non signé dans EAX, cas B
add eax, 1
mov edx, f
```

```
mul edx
shr edx, b
```

Ce code fonctionne pour toutes les valeurs de x , sauf 0FFFFFFFH, qui donne zéro en raison d'un dépassement de capacité (*overflow*) dans l'instruction ADD EAX, 1. Si la valeur $x = 0FFFFFFFH$ est possible, modifiez alors le code comme suit :

```
; Exemple 16.5c. Division d'un entier non signé, cas B, vérifier le dépassement de capacité.
mov edx, f
add eax, 1
jc DOVERFL
mul edx
DOVERFL: shr edx, b
```

Si la valeur de x est limitée, vous pouvez utiliser une valeur de r inférieure, c'est-à-dire moins de chiffres. Il peut y avoir plusieurs raisons d'utiliser une valeur de r inférieure :

- Vous pouvez choisir $r = w = 32$ pour éviter l'instruction SHR EDX, b à la fin.
- Vous pouvez choisir $r = 16 + b$ et utiliser une instruction de multiplication produisant un résultat sur 32 bits plutôt que sur 64 bits. Cela libérera le registre EDX :

```
; Exemple 16.5d. Division d'un entier non signé par 5, plage limitée
imul eax, 0CCCDH
shr eax, 18
```
- Vous pouvez choisir une valeur de r conduisant au cas C plutôt qu'au cas B afin d'éviter l'instruction ADD EAX, 1.

Dans ces cas, la valeur maximale de x est d'au moins $2^{r-b} - 1$, voire supérieure. Il est nécessaire d'effectuer un test systématique pour déterminer la valeur maximale exacte de x pour laquelle le code fonctionne correctement.

Si l'opération de multiplication est lente, vous pouvez remplacer l'instruction de multiplication par des instructions de décalage et d'addition, comme expliqué à la page 110.

L'exemple suivant divise EAX par 10 et renvoie le résultat dans EAX. J'ai choisi $r = 17$ plutôt que 19 car cela permet d'obtenir un code plus facile à optimiser, tout en couvrant la même plage de valeurs pour x . $f = 2^{17} / 10 = 3333H$, cas B : $q = (x+1) * 3333H$:

```
; Exemple 16.5e. Division d'un entier non signé par 10, plage limitée
lea ebx, [eax+2*eax+3]
lea ecx, [eax+2*eax+3]
shl ebx, 4
mov eax, ecx
shl ecx, 8
add eax, ebx
shl ebx, 8
add eax, ecx
add eax, ebx
shr eax, 17
```

Un test systématique montre que ce code fonctionne correctement pour tout $x < 10004H$.

La méthode de division peut également être utilisée pour des opérandes vectoriels. L'exemple 16.5f divise huit entiers non signés de 16 bits par 10 :

```
; Exemple 16.5f. Division d'un vecteur d'entiers non signés par 10
section .data
align 16
RECIPDIV times 8 DW 0CCCDH ; Vecteur des diviseurs réciproques
section .text
pmulhuw xmm0, [RECIPDIV]
psrlw xmm0, 3
```

Division entière répétée par la même valeur (tous les processeurs)

Si le diviseur n'est pas connu au moment de l'assemblage, mais que vous effectuez des divisions répétées avec le même diviseur, il est avantageux d'utiliser la méthode décrite ci-dessus. La division n'est réalisée qu'une seule fois, tandis que les opérations de multiplication et de décalage sont effectuées pour chaque dividende. Une variante de l'algorithme sans branchement est présentée dans l'article de Granlund et Montgomery cité plus haut.

Cette méthode est implémentée dans la bibliothèque de fonctions [asmlib](#) ainsi que dans la [bibliothèque de classes de vecteurs C++](#), tant pour les entiers signés et non signés que pour les registres vectoriels.

Division en virgule flottante (tous les processeurs)

Il est possible de combiner plusieurs divisions en virgule flottante en une seule opération, en utilisant la méthode décrite dans le manuel 1 : “Optimizing software in C++”.

La durée d'une division en virgule flottante dépend de la précision. Lorsque l'on utilise des registres en virgule flottante de type x87, il est possible d'accélérer la division en spécifiant une précision moindre dans le mot de contrôle de l'unité de calcul en virgule flottante. Cela accélère également l'instruction FSQRT, mais aucune autre instruction. En revanche, l'utilisation de registres XMM ne nécessite aucune modification du mot de contrôle ; il suffit d'employer des instructions en simple précision si l'application le permet.

Il est impossible d'effectuer simultanément une division en virgule flottante et une division entière, car elles utilisent la même unité d'exécution sur la plupart des processeurs.

Utilisation de l'instruction réciproque pour la division rapide

Les instructions suivantes calculent les inverses approximatifs de nombres à virgule flottante en simple précision, avec divers niveaux de précision.

instruction	précision	Jeu d'instructions
rcpss, rcpps	12 bits	SSE
vrcp14ss, vrcp14ps	14 bits	AVX512
vrcp28ss, vrcp28ps	28 bits	AVX512ER

Tableau 16.1. Instructions réciproques approximatives

Il est possible d'effectuer des divisions approximatives en utilisant l'une des instructions de calcul rapide de l'inverse sur le diviseur, puis en multipliant le résultat par le dividende. La précision peut être portée de 12 à 23 bits en appliquant le principe de l'itération de Newton-Raphson :

```
x0 = rcpss(d);  
x1 = x0 * (2 - d * x0) = 2 * x0 - d * x0 * x0;
```

où x_0 est la première approximation de l'inverse du diviseur d , et x_1 est une meilleure approximation. Vous devez utiliser cette formule avant de multiplier par le dividende.

```
; Exemple 16.6. Division rapide, simple précision (SSE)  
movaps xmm1, [divisors] ; chargement des diviseurs  
rcpps xmm0, xmm1        ; inverse approximatif  
mulps xmm1, xmm0        ; formule de Newton-Raphson  
mulps xmm1, xmm0  
addps xmm0, xmm0  
subps xmm0, xmm1  
mulps xmm0, [dividends] ; résultats dans XMM0
```

Il est possible d'accroître encore la précision en appliquant de nouveau la formule de Newton-Raphson en double précision, mais cela ne présente guère d'avantage.

16.9 Instructions de manipulation de chaînes (tous les processeurs)

Les instructions de manipulation de chaînes dépourvues de préfixe de répétition sont trop lentes et devraient être remplacées par des instructions plus simples. Il en va de même pour l'instruction LOOP et, sur certains processeurs, pour JECXZ.

Les instructions REP MOVSD et REP STOSD sont assez rapides si le nombre de répétitions n'est pas trop faible. Il est préférable d'utiliser la taille de mot la plus grande possible (DWORD en mode 32 bits, QWORD en mode 64 bits). La source et la destination doivent être alignées sur la taille du mot (ou mieux). Toutefois, dans de nombreux cas, l'utilisation de registres vectoriels est plus rapide. Le transfert de données via les plus grands registres disponibles est plus rapide que l'utilisation de REP MOVSD et REP STOSD dans la plupart des situations, en particulier sur les processeurs plus anciens. Voir la page 121 pour plus de détails.

Notez que si l'instruction REP MOVSD écrit un mot dans la destination, elle lit simultanément le mot suivant de la source au cours du même cycle d'horloge. Cela peut entraîner des conflits de bancs de cache sur certains processeurs anciens. Le moyen le plus simple d'éviter ces conflits consiste à aligner la source et la destination sur une frontière de 8 octets.

Sur de nombreux processeurs, les instructions REP MOVSD et REP STOSD peuvent s'exécuter très rapidement en transférant 16 octets ou une ligne de cache entière à la fois. Cela ne se produit toutefois que si certaines conditions sont réunies. Selon le processeur, les conditions requises pour une exécution rapide de ces instructions de manipulation de chaînes sont généralement les suivantes : un nombre d'itérations élevé, un alignement des adresses source et destination, un sens de transfert croissant, une distance entre la source et la destination au moins égale à la taille d'une ligne de cache, et un type de mémoire (pour la source comme pour la destination) de type *write-back* ou *write-combining* (cette dernière condition est généralement remplie).

Dans ces conditions, la vitesse atteinte est comparable à celle obtenue avec des transferts utilisant des registres vectoriels.

Bien que les instructions de manipulation de chaînes soient très pratiques, il convient de souligner que d'autres solutions s'avèrent plus rapides dans de nombreux cas. Si les conditions susmentionnées pour un transfert rapide ne sont pas réunies, l'utilisation d'autres méthodes peut s'avérer bien plus avantageuse. Consultez la page 151 pour découvrir des alternatives à REP MOVS.

Les instructions REP LOADS, REP SCAS et REP CMPS nécessitent plus de temps par itération que les boucles simples. Consultez la page 105 pour des alternatives à REPNE SCASB.

16.10 Instructions de chaîne vectorisées (processeurs avec SSE4.2)

The SSE4.2 instruction set contains four very powerful instructions for string search and compare operations: PCMPSTRI, PCMPISTRI, PCMPSTRM, PCMPISTRM. There are two vector operands each containing a string and an immediate operand determining what operation to do on these strings. These four instructions can all do each of the following kinds of operations, differing only in the input and output operands:

Le jeu d'instructions SSE4.2 comprend quatre instructions très puissantes pour les opérations de recherche et de comparaison de chaînes : PCMPSTRI, PCMPISTRI, PCMPSTRM et PCMPISTRM. Elles utilisent deux opérandes vectoriels contenant chacun une chaîne, ainsi qu'un opérande immédiat déterminant l'opération à effectuer sur ces chaînes. Ces quatre instructions peuvent toutes réaliser chacun des types d'opérations suivants, ne différant que par les opérandes d'entrée et de sortie :

- Recherche de caractères dans un ensemble : identifie les octets spécifiés par le second opérande vectoriel présents parmi les octets du premier opérande vectoriel, en comparant les 256 combinaisons possibles en une seule opération.
- Recherche de caractères dans une plage : identifie les octets du second opérande vectoriel compris dans la plage définie par le premier opérande vectoriel.
- Comparaison de chaînes : détermine si deux chaînes sont identiques.
- Recherche de sous-chaîne : trouve toutes les occurrences d'une sous-chaîne définie par le premier opérande vectoriel au sein du second opérande vectoriel.

La longueur des deux chaînes peut être spécifiée explicitement (PCMPSTRx) ou implicitement via un caractère de fin nul (PCMPISTRx). Cette dernière méthode est plus rapide sur les processeurs Intel actuels car elle nécessite moins d'opérandes en entrée. Les éventuels caractères nuls terminaux ainsi que les octets situés au-delà de la fin des chaînes sont ignorés. Le résultat peut prendre la forme d'un masque de bits ou d'octets (PCMPxSTRM) ou d'un index (PCMPxSTRI). Des informations supplémentaires sont fournies par les flags suivants : flag de retenue (*Carry flag*) : résultat non nul ; flag de signe (*Sign flag*) : fin de la première chaîne ; flag de zéro (*Zero flag*) : fin de la seconde chaîne ; flag de dépassement (*Overflow flag*) : premier bit du résultat.

Ces instructions sont très efficaces pour diverses tâches d'analyse et de traitement de chaînes, car elles permettent d'effectuer des opérations complexes et de grande envergure en une seule étape. Toutefois, les implémentations actuelles sont plus lentes que les meilleures alternatives pour des tâches simples telles que strlen et strcpy.

Des exemples sont disponibles dans ma bibliothèque de fonctions à l'adresse www.agner.org/optimize/asmlib.zip.

16.11 Instruction WAIT (tous les processeurs)

L'instruction WAIT (également appelée FWAIT) a trois fonctions :

A. L'ancien coprocesseur 8087 nécessite une instruction WAIT avant chaque instruction en virgule flottante, afin de s'assurer que le coprocesseur est prêt à la recevoir.

B. WAIT est utilisé pour coordonner l'accès à la mémoire entre l'unité de calcul en virgule flottante et l'unité de calcul en nombres entiers. Exemples :

; **Exemple 16.7.** Usages de WAIT :

```
B1:  fistp [mem32]
      wait                ; attente que la FPU écrive au préalable..
      mov eax,[mem32]     ; lecture du résultat avec l'unité entière

B2:  fild [mem32]
      wait                ; attente que la FPU lise la valeur..
      mov [mem32],eax     ; avant de l'écraser par une unité entière

B3:  fld qword [ESP]
      wait                ; empêchement d'une interruption accidentelle de..
      add esp,8           ; écrasement d'une valeur sur la pile
```

C. L'instruction WAIT est parfois utilisée pour vérifier la présence d'exceptions. Elle génère une interruption si un bit d'exception non masqué, situé dans le mot d'état en virgule flottante, a été positionné par une instruction en virgule flottante précédente.

Concernant A :

La fonctionnalité mentionnée au point A n'est jamais requise sur des processeurs autres que l'ancien 8087. À moins que vous ne souhaitiez que votre code 16 bits soit compatible avec le 8087, vous devriez demander à votre assembleur de ne pas inclure ces instructions WAIT en spécifiant un processeur plus récent. Un émulateur de virgule flottante 8087 sous DOS insère également des instructions WAIT ; vous devez donc indiquer à votre assembleur de ne pas générer de code d'émulation, sauf si vous en avez besoin.

Concernant B:

Les instructions WAIT servant à coordonner les accès mémoire sont absolument nécessaires sur les coprocesseurs 8087 et 80287, mais pas sur les processeurs ultérieurs.

Concernant C:

À cette fin, l'assembleur insère automatiquement une instruction WAIT avant les instructions suivantes : FCLEX, FINIT, FSAVE, FSTCW, FSTENV et FSTSW. Vous pouvez omettre le WAIT en écrivant FNCLEX, etc. Mes tests montrent que le WAIT est inutile dans la plupart des cas, car ces instructions, même sans WAIT, déclenchent tout de même une interruption en cas d'exception (à l'exception de FNCLEX et FNINIT sur le 80387). Il existe une certaine incohérence quant à savoir si l'instruction IRET de l'interruption renvoie à l'instruction FN... ou à l'instruction suivante.

La quasi-totalité des autres instructions x87 en virgule flottante déclenchera également une interruption si une instruction x87 précédente a positionné un bit d'exception non masqué ; l'exception sera donc probablement détectée tôt ou tard. Vous pouvez insérer une instruction WAIT après la dernière instruction x87 de votre programme pour vous assurer de détecter toutes les exceptions.

16.12 FCOM + FSTSW AX (tous les processeurs)

L'instruction FNSTSW est très lente sur tous les processeurs. La plupart des processeurs disposent d'instructions FCOMI permettant d'éviter l'utilisation de la lente FNSTSW. L'utilisation de FCOMI à la place de la séquence classique FCOM / FNSTSW AX / SAHF permet d'économiser entre 4 et 8 cycles d'horloge. Il est donc recommandé d'utiliser FCOMI pour éviter FNSTSW chaque fois que cela est possible, même si cela nécessite un peu plus de code.

Il est parfois plus rapide d'utiliser des instructions sur entiers pour comparer des valeurs en virgule flottante, comme indiqué aux pages 119 et **Erreur ! Signet non défini.**

16.13 FPREM (tous les processeurs)

Les instructions FPREM et FPREM1 sont lentes sur tous les processeurs. Vous pouvez les remplacer par l'algorithme suivant : multiplier par l'inverse du diviseur, obtenir la partie fractionnaire en soustrayant la valeur tronquée, puis multiplier par le diviseur.

Certains documents indiquent que ces instructions peuvent conduire à des réductions incomplètes et qu'il est donc nécessaire de les répéter jusqu'à ce que la réduction soit totale. J'ai effectué des tests sur plusieurs processeurs, en commençant par l'ancien 8087, et je n'ai rencontré aucune situation nécessitant une répétition de FPREM ou FPREM1.

16.14 FRNDINT (tous les processeurs)

Cette instruction est lente sur tous les processeurs. Remplacez-la par :

```
; Exemple 16.8.  
fistp qword [TEMP]  
fild qword [TEMP]
```

Ce code est plus rapide, malgré une pénalité potentielle liée à la tentative de lecture depuis [TEMP] avant la fin de l'écriture sur les processeurs plus anciens. Vous pouvez insérer d'autres instructions entre les deux pour éviter cette pénalité.

Sur les processeurs dotés des jeux d'instructions SSE, il convient plutôt d'utiliser les instructions de conversion telles que CVTSS2SI et CVTTSS2SI.

16.15 FSCALE et fonction exponentielle (tous les processeurs)

L'instruction FSCALE est lente sur tous les processeurs. Le calcul de puissances entières de 2 peut être effectué beaucoup plus rapidement en insérant la puissance souhaitée dans le champ exposant du nombre à virgule flottante. Pour calculer 2^N , où N est un entier signé, choisissez parmi les exemples ci-dessous celui qui correspond à la plage de valeurs de N :

Simple précision, pour $|N| < 2^7 - 1$:

```
; Exemple 16.9a. 2 à la puissance i, simple précision
mov eax, [N]
shl eax, 23
add eax, 3f800000h
movd xmm0, eax
```

Simple précision, pour $|N| < 2^{10}-1$:

```
; Exemple 16.9b. 2 à la puissance i, double précision
mov eax, [N]
shl eax, 20
add eax, 3ff00000h
movd xmm0, eax
psllq xmm0, 32
```

16.16 FPTAN (tous les processeurs)

Selon les manuels, FPTAN renvoie deux valeurs, X et Y , et laisse au programmeur le soin de diviser Y par X pour obtenir le résultat ; mais en fait, il renvoie toujours 1 dans X pour que vous puissiez sauvegarder la division. Mes tests montrent que sur tous les processeurs Intel 32 bits avec unité à virgule flottante ou coprocesseur, FPTAN renvoie toujours 1 dans X quel que soit l'argument. Si vous voulez être absolument sûr que votre code s'exécutera correctement sur tous les processeurs, vous pouvez tester si X vaut 1, ce qui est plus rapide que de diviser par X . La valeur Y peut être très élevée, mais jamais infinie, vous n'avez donc pas besoin de tester si Y contient un nombre valide si vous savez que l'argument est valide.

Lors de l'optimisation de la vitesse, il est généralement plus rapide d'utiliser une fonction de bibliothèque avec SSE2.

16.17 FSQRT, SQRSS

Les instructions suivantes sont disponibles pour le calcul des inverses de racines carrées :

instruction	précision	Jeu d'instructions
rsqrtss, rsqrtps	12 bits, float	SSE
vrsqrt14ss, vrsqrt14ps	14 bits, float	AVX512
vrsqrt14sd, vrsqrt14pd	14 bits, double	AVX512
vrsqrt28ss, vrsqrt28ps	28 bits, float	AVX512ER
vrsqrt28sd, vrsqrt28pd	28 bits, double	AVX512ER
Tableau 16.2. Instructions pour l'approximation de l'inverse de la racine carrée		

Une méthode rapide pour calculer une approximation de la racine carrée consiste à multiplier l'inverse de la racine carrée de x par x :

$$\text{sqrt}(x) = x * \text{rsqrt}(x)$$

Les instructions de racine carrée inverse fournissent la racine carrée inverse avec une précision d'au moins 12 bits. Vous pouvez porter cette précision à 23 bits en utilisant la formule d'itération de Newton-Raphson :

$$\begin{aligned} x_0 &= \text{rsqrtss}(a) \\ x_1 &= 0.5 * x_0 * (3 - (a * x_0) * x_0) \end{aligned}$$

où x_0 est la première approximation de l'inverse de la racine carrée de a , et x_1 est une meilleure approximation. L'ordre d'évaluation est important. Vous devez utiliser cette formule avant de multiplier par x pour obtenir la racine carrée.

16.18 FLDCW

De nombreux processeurs subissent un ralentissement important après l'instruction FLDCW si celle-ci est suivie d'une instruction en virgule flottante lisant le mot de contrôle (ce qui est le cas de la quasi-totalité de ces instructions).

Lorsque du code C ou C++ est compilé sans prise en charge de SSE2, il génère souvent un grand nombre d'instructions FLDCW, car la conversion de nombres à virgule flottante en entiers s'effectue par troncature, alors que les autres instructions en virgule flottante utilisent l'arrondi. Après traduction en langage assembleur, il est possible d'optimiser ce code en privilégiant l'arrondi à la troncature lorsque cela est possible, ou en sortant l'instruction FLDCW d'une boucle si la troncature est nécessaire à l'intérieur de celle-ci.

Une meilleure solution consiste à compiler pour les jeux d'instructions SSE ou SSE2 et à utiliser des instructions de troncature telles que CVTTSS2SI.

16.19 Instructions MASKMOV

Les instructions d'écriture en mémoire avec masque MASKMOVQ et MASKMOVDQU sont extrêmement lentes car elles contournent le cache pour écrire directement dans la mémoire principale (ce que l'on appelle une écriture non temporelle).

Les variantes utilisant le codage VEX — VPMASKMOVD, VPMASKMOVQ, VMASKMOVPS et VMASKMOVPD — écrivent dans le cache et sont donc beaucoup plus rapides sur les processeurs Intel. Toutefois, ces versions VEX restent extrêmement lentes sur les processeurs AMD Bulldozer et Piledriver.

Il convient d'éviter ces instructions à tout prix. Le jeu d'instructions AVX-512 propose des opérations d'écriture avec masque bien plus performantes. Voir également la [section 13.9](#) pour d'autres méthodes.

16.20 Instructions RDRAND et RDSEED

Ces instructions échantillonnent du bruit physique sur une certaine durée afin de générer un nombre aléatoire imprévisible. Si elles sont appelées de manière répétée, ces instructions peuvent nécessiter des centaines, voire des milliers, de cycles d'horloge. Un programme nécessitant un grand nombre de valeurs aléatoires devrait utiliser un générateur de nombres pseudo-aléatoires et ne recourir à RDRAND ou RDSEED que pour générer la valeur initiale (graine) de ce générateur.

17 Sujets particuliers

17.1 XMM contre registres à virgule flottante

Les processeurs dotés du jeu d'instructions SSE peuvent effectuer des calculs en virgule flottante simple précision dans les registres XMM. Les processeurs dotés du jeu d'instructions SSE2 peuvent également effectuer des calculs double précision dans les registres XMM. La vitesse d'exécution des calculs en virgule flottante est sensiblement identique, que l'on utilise les registres XMM ou les anciens registres x87 (ST(0) à ST(7)). Le choix entre l'utilisation des registres x87 ST(0) à ST(7) et celle des registres XMM dépend des facteurs suivants :

Avantages de l'utilisation des registres x87 :

- Compatible avec les anciens processeurs sans SSE ou SSE2.
- Compatible avec les anciens systèmes d'exploitation sans support XMM.
- Prend en charge la double précision longue.
- Les résultats intermédiaires sont calculés avec une double précision longue.
- Les conversions de précision sont gratuites dans le sens où elles ne nécessitent aucune instruction supplémentaire et ne prennent pas de temps supplémentaire. Vous pouvez utiliser les registres ST() pour les expressions où les opérandes ont une précision mixte.
- Les fonctions mathématiques telles que les logarithmes et les fonctions trigonométriques sont prises en charge par les instructions matérielles. Ces fonctions sont utiles lors de l'optimisation de la taille, mais pas plus rapides que les fonctions de bibliothèque utilisant des registres XMM.
- Les conversions vers et depuis des nombres décimaux peuvent utiliser les instructions FBSTP et FBSTP lors de l'optimisation de la taille, mais ces instructions sont lentes.
- Les instructions à virgule flottante utilisant les registres ST() sont plus petites que les instructions correspondantes utilisant les registres XMM. Par exemple, FADD ST(0),ST(1) fait 2 octets, tandis que ADDSD XMM0, XMM1 fait 4 octets.

Avantages de l'utilisation des registres XMM, YMM ou ZMM :

- Permet d'effectuer plusieurs opérations avec une seule instruction vectorielle.
- Évite d'avoir recours à l'instruction FXCH pour placer le registre souhaité au sommet de la pile.
- Ne nécessite pas de nettoyage de la pile de registres après utilisation.
- Peut être utilisé conjointement avec des instructions MMX.
- Ne nécessite pas de zones mémoire intermédiaires lors de la conversion entre entiers et nombres à virgule flottante.
- Les systèmes 64 bits disposent de 16 registres XMM/ymm ou 32 registres ZMM, contre seulement 8 registres ST().
- Les registres ST() ne peuvent pas être utilisés dans les pilotes de périphériques sous Windows 64 bits.
- Le jeu d'instructions associé aux registres ST() n'évolue plus. Ces instructions resteront probablement prises en charge pendant de nombreuses années pour garantir la compatibilité ascendante, mais leur efficacité pourrait diminuer sur les futurs processeurs.

17.2 Registres MMX contre XMM

Les instructions vectorielles sur entiers peuvent utiliser soit les registres MMX de 64 bits, soit les registres XMM de 128 bits sur les processeurs dotés de la technologie SSE2.

Avantages de l'utilisation des registres MMX :

- Compatible avec les microprocesseurs antérieurs au PMMX.
- Compatible avec les anciens systèmes d'exploitation dépourvus de prise en charge XMM.
- Aucun alignement des données requis.

Avantages de l'utilisation des registres XMM :

- Le nombre d'éléments par vecteur est doublé dans les registres XMM par rapport aux registres MMX, et augmenté encore dans les registres YMM et ZMM.
- Les registres MMX ne peuvent pas être utilisés avec les registres ST().
- Une série d'instructions MMX doit se terminer par EMMS.
- Les systèmes 64 bits ont 16 ou 32 registres XMM, mais seulement 8 registres MMX.
- Les registres MMX ne peuvent pas être utilisés dans les pilotes de périphérique sous Windows 64 bits.
- Le jeu d'instructions pour les registres MMX n'est plus développé et n'est plus utilisé. Les registres MMX seront probablement encore pris en charge dans de nombreuses années pour des raisons de compatibilité ascendante.

17.3 Registres XMM par rapport aux registres YMM et ZMM

Les registres XMM de 128 bits sont étendus à 256 ou 512 bits dans les registres YMM et ZMM lorsque les jeux d'instructions les plus récents sont disponibles. Voir page 85 pour plus de détails.

Avantages de l'utilisation du jeu d'instructions AVX et des registres YMM ou ZMM :

- Davantage d'éléments par vecteur
- Version non destructive à trois opérandes pour toutes les instructions XMM, YMM et ZMM
- Instructions avec masquage disponibles via AVX-512.

Avantages de l'utilisation des registres XMM :

- Compatibilité avec les processeurs plus anciens
- Une pénalité de performance survient lors du basculement entre des instructions VEX et des instructions XMM sans préfixe VEX (voir page 87). Le programmeur peut mélanger par inadvertance des instructions VEX et non-VEX.
- Les registres YMM et ZMM ne peuvent pas être utilisés dans les pilotes de périphériques sans sauvegarder l'état complet via XSAVE / XRESTOR (voir page 88).

17.4 Libération des registres à virgule flottante

Vous devez libérer tous les registres à virgule flottante de type x87 utilisés avant de quitter une sous-routine, à l'exception de celui servant à stocker le résultat.

La méthode la plus rapide pour libérer un registre est l'instruction `FSTP ST`. Pour en libérer deux, vous pouvez utiliser soit `FCOMPP`, soit deux fois `FSTP ST`, selon ce qui s'intègre le mieux dans la séquence de décodage ou la charge des ports.

L'utilisation de `FFREE` est déconseillée.

17.5 Transitions entre les instructions à virgule flottante et les instructions MMX

Il n'est pas possible d'utiliser simultanément des registres MMX de 64 bits et des registres à virgule flottante ST() de 80 bits dans une même section de code. Vous devez exécuter une instruction EMMS après la dernière instruction utilisant des registres MMX si le code ultérieur est susceptible d'utiliser des registres à virgule flottante. Vous pouvez éviter ce problème en utilisant plutôt des registres XMM de 128 bits.

Sur le processeur PMMX, le basculement entre les instructions à virgule flottante et les instructions MMX entraîne une pénalité importante. La première instruction à virgule flottante suivant une instruction EMMS nécessite environ 58 cycles d'horloge supplémentaires, tandis que la première instruction MMX suivant une instruction à virgule flottante en nécessite environ 38 de plus. Cette pénalité n'existe pas sur les processeurs dotés d'une exécution dans le désordre (*out-of-order execution*).

17.6 Conversion de virgule flottante en entier

Toutes les conversions entre les registres à virgule flottante de type x87 et les registres d'entiers doivent transiter par un emplacement mémoire :

; **Exemple 17.1.**

```
fistp dword [TEMP]
mov eax, [TEMP]
```

Sur les processeurs plus anciens, et en particulier le P4, ce code risque de subir une pénalité si l'on tente de lire depuis [TEMP] avant que l'écriture dans [TEMP] ne soit terminée. L'insertion d'une instruction WAIT n'y change rien. Il est recommandé d'insérer d'autres instructions entre l'écriture dans [TEMP] et la lecture depuis [TEMP], si possible, afin d'éviter cette pénalité. Cela s'applique à tous les exemples qui suivent.

Les spécifications des langages C et C++ imposent que la conversion de nombres à virgule flottante en entiers s'effectue par troncature plutôt que par arrondi. La méthode utilisée par la plupart des bibliothèques C consiste à modifier le mot de contrôle de l'unité de calcul en virgule flottante pour activer la troncature avant d'exécuter une instruction FISTP, puis à le rétablir ensuite. Cette méthode est très lente sur tous les processeurs. Sur de nombreux processeurs, le mot de contrôle de l'unité de calcul en virgule flottante ne peut pas être renommé ; par conséquent, toutes les instructions en virgule flottante ultérieures doivent attendre que l'instruction FLDCW soit achevée (retirée du pipeline). Voir page 116.

Vous pouvez éviter tous ces problèmes en utilisant des registres XMM au lieu des registres en virgule flottante et en employant les instructions CVT... pour s'affranchir du passage par la mémoire sur les processeurs compatibles SSE2.

17.7 Utilisation d'instructions sur entiers pour des opérations en virgule flottante

Il est souvent avantageux d'utiliser des instructions sur les entiers pour effectuer des opérations simples en virgule flottante, car ces instructions sont généralement plus rapides que celles traitant les nombres à virgule flottante. L'exemple le plus évident est le transfert de données. Par exemple :

```
; Exemple 17.2a. Déplacement de données en virgule flottante
fld qword [esi]
fstp qword [edi]
```

peut être remplacé par :

```
; Exemple 17.2b
mov rax, [rsi]
mov [rdi], rax
```

De nombreuses autres manipulations sont possibles si vous savez comment les nombres à virgule flottante sont représentés en format binaire. Consultez le chapitre “Utilisation d'opérations sur les entiers pour manipuler des variables à virgule flottante” dans le manuel 1 : “Optimisation de logiciels en C++”.

Les positions des bits sont indiquées dans ce tableau :

Précision	mantisse	toujours à 1	exposant	signe
single (32 bits)	bit 0 – 22		bit 23 – 30	bit 31
double (64 bits)	bit 0 – 51		bit 52 – 62	bit 63
long double (80 bits)	bit 0 – 62	bit 63	bit 64 – 78	bit 79

Tableau 17.1. Formats à virgule flottante

Ce tableau montre que la valeur 1,0 est représentée par 3F80,0000H en simple précision, 3FF0,0000,0000,0000H en double précision et 3FFF,8000,0000,0000,0000H en précision “long double”. Il est possible de générer des constantes à virgule flottante simples sans recourir à des données en mémoire, comme expliqué à la page 100.

Test si une valeur à virgule flottante est égale à zéro

Pour vérifier si un nombre à virgule flottante est égal à zéro, il faut examiner tous les bits à l'exception du bit de signe, qui peut valoir 0 ou 1. Par exemple :

```
; Exemple 17.3a. Test si une valeur à virgule flottante est égale à zéro
fld dword [ebx]
ftst
fnstsw ax
and ah, 40h
jnz IsZero
```

peut être remplacé par

```
; Exemple 17.3b. Test si une valeur à virgule flottante est égale à zéro
mov eax, [ebx]
add eax, eax
jz IsZero
```

où ADD EAX, EAX décale le bit de signe. Les flottants à double précision ont 63 bits à tester, mais si les nombres anormaux peuvent être exclus, alors vous pouvez être certain que la valeur est nulle si les bits des exposants sont tous nuls. Exemple :

```
; Exemple 17.3c. Test si une valeur double-précision est égale à zéro
fld qword [ebx]
ftst
fnstsw ax
and ah, 40h
jnz IsZero
```

peut être remplacé par

```
; Exemple 17.3d. Test si une valeur double-précision est égale à zéro
mov eax, [ebx+4]
add eax, eax
jz IsZero
```

Manipulation du bit de signe

Un nombre à virgule flottante est négatif si le bit de signe est positionné (à 1) et qu'au moins un autre bit est positionné. Exemple (simple précision) :

```
; Exemple 17.4. Test si une valeur à virgule flottante est négative
mov eax, [NumberToTest]
cmp eax, 80000000H
ja IsNegative
```

Vous pouvez changer le signe d'un nombre à virgule flottante simplement en inversant le bit de signe. Cela s'avère utile lors de l'utilisation de registres XMM, car il n'existe pas d'instruction XMM pour changer le signe. Exemple :

```
; Exemple 17.5. Inversion du signe de 4 nombres à virgule flottante simple précision dans xmm0.
cmpeqd xmm1, xmm1 ; génération uniquement de 1
pslld xmm1, 31 ; 1 uniquement dans le bit le plus à gauche de chaque dword
xorps xmm0, xmm1 ; changement du signe de xmm0
```

Vous pouvez obtenir la valeur absolue d'un nombre à virgule flottante en éliminant le bit de signe par une opération AND :

```
; Exemple 17.6. Valeur absolue de 4 nombres à virgule flottante simple précision dans xmm0
cmpeqd xmm1, xmm1 ; génération uniquement de 1
psrld xmm1, 1 ; 1 uniquement dans le bit le plus à gauche de chaque dword
andps xmm0, xmm1 ; mise à zéro des bits de signe
```

Vous pouvez extraire le bit de signe d'un nombre à virgule flottante :

```
; Exemple 17.7. Génération d'un masque de bits si les nombres flottants sont en simple précision.
; xmm0 sont négatives ou -0.0
psrad xmm0, 31 ; Copie du bit de signe dans toutes les positions de bit.
```

Manipulation de l'exposant

Vous pouvez multiplier un nombre non nul par une puissance de 2 en ajoutant simplement une valeur à l'exposant :

```
; Exemple 17.8. Multiply vector by power of 2
movaps xmm0, [x] ; 4 nombres à virgule flottante simple précision
movdqa xmm1, [n] ; 4 entiers positifs de 32 bits
pslld xmm1, 23 ; décalage des entiers vers le champ exposant
paddb xmm0, xmm1 ;  $x * 2^n$ 
```

De même, vous pouvez diviser par une puissance de 2 en soustrayant une valeur à l'exposant. Notez que ce code ne fonctionne pas si x est nul ou si un dépassement de capacité (positif ou négatif) est possible.

Manipulation de la mantisse

Vous pouvez convertir un entier en un nombre à virgule flottante appartenant à un intervalle de longueur 1,0 en plaçant des bits dans le champ de la mantisse. Le code suivant calcule $x = n / 2^{32}$, où n est un entier non signé compris dans l'intervalle $0 \leq n < 2^{32}$, et où le résultat x se situe dans l'intervalle $0 \leq x < 1,0$.

```
; Exemple 17.9. Conversion des bits en une valeur comprise entre 0 et 1
section .data
one dq 1.0
x dq 0
n dd 0

section .text
movsd xmm0, [one] ; 1.0, double précision
```

```

movd xmm1, [n]      ; n, entier non signé de 32 bits
psllq xmm1, 20      ; alignement à gauche dans le champ de la mantisse
orpd xmm1, xmm0     ; combinaison de la mantisse et de l'exposant
subsd xmm1, xmm0    ; soustraction de 1,0
movsd [x], xmm1     ; mémorisation du résultat

```

Dans le code ci-dessus, l'exposant de 1,0 est combiné avec une mantisse contenant les bits de n . On obtient ainsi une valeur en double précision située dans l'intervalle $1,0 \leq x < 2,0$. L'instruction SUBSD soustrait 1,0 pour placer x dans l'intervalle souhaité. Cette technique est utile pour les générateurs de nombres aléatoires.

Comparaison de nombres

Grâce au fait que l'exposant est stocké au format décalé et placé à gauche de la mantisse, il est possible d'utiliser des instructions sur les entiers pour comparer des nombres à virgule flottante positifs. Exemple (simple précision) :

```

; Exemple 17.10a. Comparaison de nombres en virgule flottante simple précision
fld dword [a]
fcomp dword [b]
fnstsw ax
and ah, 1
jnz ASmallerThanB

```

peut être remplacé par

```

; Exemple 17.10b. Comparaison de nombres en virgule flottante simple précision
mov eax, [a]
mov ebx, [b]
cmp eax, ebx
jb ASmallerThanB

```

Cette méthode ne fonctionne que si vous avez la certitude qu'aucun des nombres n'a son bit de signe activé. Vous pouvez comparer les valeurs absolues en éliminant le bit de signe des deux nombres par décalage. Pour les nombres en double précision, il est possible d'effectuer une comparaison approximative en comparant les 32 bits de poids fort à l'aide d'instructions pour entiers 32 bits.

17.8 Déplacement de blocs de données

Il existe plusieurs façons de déplacer de grands blocs de données. Les méthodes les plus courantes sont :

1. Instruction REP MOVSB.
2. Si les données sont alignées : lecture et écriture en boucle en utilisant la plus grande taille de registre disponible.
3. Si la taille est constante : insertion en ligne (*inlining*) des instructions de transfert.
4. Si les données sont désalignées : transférer d'abord le nombre d'octets nécessaire pour aligner la destination. Ensuite, effectuer des lectures non alignées et des écritures alignées en boucle, en utilisant la plus grande taille de registre disponible.
5. Si les données sont désalignées : lecture alignée, décalage pour compenser le désalignement et écriture alignée.
6. Si la taille des données est trop importante pour le cache, utiliser des écritures non temporelles pour contourner le cache. Effectuer un décalage pour compenser le désalignement, si nécessaire.

La méthode la plus rapide dépend du microprocesseur, de la taille des données et de l'alignement. Si une part importante du temps processeur est consacrée au transfert de données, il est alors crucial de choisir la méthode la plus rapide. J'aborderai donc ici les avantages et les inconvénients de chaque méthode.

L'instruction REP MOVSB (1) constitue une solution simple, utile lorsque l'on cherche à optimiser la taille du code plutôt que la vitesse d'exécution. Cette instruction est implémentée sous forme de microcode au sein du processeur. Il arrive que cette implémentation en microcode utilise en interne l'une des autres méthodes ; son niveau d'optimisation varie selon les cas. En règle générale, l'instruction REP MOVSB entraîne une surcharge importante liée au choix et à la configuration de la méthode appropriée. Elle n'est donc pas optimale pour les petits blocs de données. En revanche, pour les blocs volumineux, elle peut s'avérer très efficace si certaines conditions — notamment d'alignement — sont réunies. Ces conditions dépendent du processeur spécifique (voir page 113). Sur les processeurs Intel Nehalem et les modèles ultérieurs, cette méthode est parfois aussi rapide que les autres lorsque le bloc de mémoire est volumineux.

Les déplacements alignés utilisant les plus grands registres disponibles (2) constituent une méthode efficace sur tous les processeurs. Il est donc avantageux d'aligner les grands blocs de données et, si nécessaire, d'ajouter un remplissage (*padding*) à la fin pour que la taille de chaque bloc soit un multiple de la taille des registres.

L'utilisation d'instructions de transfert "inlinées" (3) est souvent la méthode la plus rapide lorsque la taille du bloc de données et son alignement sont connus à la compilation. Une partie du bloc peut être transférée via des registres plus petits pour s'adapter à l'alignement et à la taille des données spécifiques. La boucle d'instructions de transfert peut être entièrement déroulée si la taille est très réduite.

La lecture non alignée combinée à une écriture alignée (4) est rapide sur la plupart des processeurs récents. En général, les anciens processeurs effectuent les lectures non alignées lentement, tandis que les processeurs plus récents disposent d'une meilleure interface de cache qui optimise ce type d'opération. La méthode de lecture non alignée est raisonnablement rapide sur tous les processeurs récents et constitue même la méthode la plus rapide sur bon nombre d'entre eux.

La méthode "lecture alignée – décalage – écriture alignée" (5) est la plus rapide pour déplacer des données non alignées sur les processeurs plus anciens. Sur les processeurs dotés du jeu d'instructions SSE2, utilisez PSRLDQ, PSLLDQ et POR. Sur ceux dotés du jeu SSSE3, utilisez PALIGNR (sauf sur l'architecture Bobcat, où PALIGNR est particulièrement lent). Il est nécessaire de prévoir 16 branches distinctes pour couvrir les 16 valeurs de décalage possibles. Sur les processeurs AMD prenant en charge les instructions XOP, il est possible d'éviter cela en utilisant l'instruction VPPERM ; toutefois, la méthode de lecture non alignée (4) peut s'avérer tout aussi rapide sur ces processeurs.

Le recours aux écritures non temporelles (6) pour contourner le cache peut s'avérer avantageux lors du déplacement de très grands blocs de données. Cette approche n'est pas systématiquement plus rapide que les autres méthodes, mais elle permet de préserver le cache pour d'autres usages. En règle générale, il est judicieux d'utiliser des écritures non temporelles pour déplacer des blocs de données dont la taille dépasse la moitié de celle du cache de dernier niveau.

Comme vous pouvez le constater, il est parfois très difficile de déterminer la méthode optimale dans une situation donnée. D'après mes tests, voici le meilleur conseil que je puisse donner pour la conception d'une fonction `memcpy` universelle :

- Sur les architectures Intel Wolfdale et antérieures, Intel Atom, AMD K8 et antérieures, ainsi que VIA Nano, utilisez la méthode "lecture alignée – décalage – écriture alignée" (5).
- Sur les architectures Intel Nehalem et ultérieures, la méthode (4) est jusqu'à 33 % plus rapide que la méthode (5).
- Sur les architectures AMD K10 et ultérieures ainsi que Bobcat, utilisez la méthode "lecture non alignée – écriture alignée" (4).
- La méthode d'écriture non temporelle (6) peut être utilisée pour des blocs de données dont la taille dépasse la moitié de celle du cache de plus haut niveau.

Un autre aspect à prendre en compte est la taille des registres utilisés pour le transfert de données. La taille des registres vectoriels est passée de 64 bits à 128, 256, puis 512 bits grâce à diverses extensions du jeu d'instructions. Historiquement, les premiers processeurs prenant en charge une nouvelle taille de registre offraient souvent une bande passante limitée pour les opérations impliquant ces registres plus larges. Certaines unités d'exécution, registres physiques, bus de données ou ports de lecture et d'écriture (voire tous ces éléments) conservaient la taille de la génération précédente ; par conséquent, une opération portant sur la taille de registre maximale était scindée en interne en deux opérations de moitié cette taille. Ainsi, les premiers processeurs dotés de la technologie SSE divisaient une lecture de 128 bits en deux lectures de 64 bits, tandis que les premiers processeurs AVX divisaient une lecture de 256 bits en deux lectures de 128 bits. L'utilisation des registres les plus larges sur ces processeurs n'entraînait donc aucune augmentation de la bande passante. Dans certains cas, comme avec l'architecture AMD Bulldozer, l'utilisation de registres de 256 bits entraînait même une baisse significative des performances. Dans ces situations, il n'est pas avantageux d'utiliser une nouvelle taille de registre pour le transfert de données avant l'arrivée de la deuxième génération de processeurs la prenant en charge. En revanche, l'utilisation de registres de 512 bits s'avère avantageuse sur les premiers processeurs Intel compatibles avec cette taille.

La copie de blocs mémoire présente une complication supplémentaire qualifiée de *dépendance mémoire fallacieuse*. Lorsque le processeur lit une donnée en mémoire, il vérifie si cette opération chevauche une écriture précédente. Si tel est le cas, la lecture doit soit attendre la fin de l'écriture précédente, soit recourir au transfert direct de données d'écriture à lecture (*store forwarding*). Malheureusement, il arrive que le processeur ne puisse pas vérifier l'adresse physique complète, mais seulement le décalage au sein de la page mémoire, c'est-à-dire les 12 bits de poids faible de l'adresse. Si les 12 bits de poids faible de l'adresse de lecture chevauchent ceux d'une écriture précédente en cours, la lecture est retardée. Ce phénomène se produit lorsque la différence entre l'adresse source et l'adresse de destination est légèrement inférieure à un multiple de 4 Ko. Dans le pire des cas, cette dépendance mémoire fallacieuse peut ralentir la copie mémoire d'un facteur 3 à 4. Il est possible d'éviter ce problème en effectuant la copie en sens inverse — de la fin vers le début du bloc mémoire — lorsqu'une telle dépendance existe. Cette méthode n'est toutefois envisageable que si les blocs mémoire source et destination ne présentent aucun chevauchement réel.

La bibliothèque [asmlib](#) inclut des fonctions optimisées pour le transfert de données, capables de sélectionner automatiquement la méthode la plus efficace pour le processeur utilisé.

Préchargement (Prefetching)

Il est parfois avantageux de précharger des données avant qu'elles ne soient requises, une pratique recommandée par certains manuels. Toutefois, les processeurs modernes disposent de mécanismes de préchargement automatique. Leur architecture leur permet de prédire les prochaines adresses de données nécessaires et de les précharger automatiquement lorsque les accès s'effectuent de manière linéaire, que ce soit dans le sens de la progression ou en sens inverse. Le préchargement explicite est rarement nécessaire. La stratégie de préchargement optimale dépendant du processeur, il est préférable de s'en remettre au préchargement automatique. Le préchargement explicite s'avère particulièrement inefficace sur les processeurs Intel Ivy Bridge et AMD Jaguar.

Transfert de données sur les futurs processeurs

Il faut garder à l'esprit que le code écrit aujourd'hui s'exécutera sur les processeurs de demain. Il est donc important d'optimiser le code pour les futurs processeurs, comme expliqué dans la section “CPU dispatch strategies” du manuel 1 : “Optimizing software in C++”. J'examinerai ici quelle méthode est susceptible d'être la plus rapide sur les processeurs à venir. Tout d'abord, on peut prévoir que les jeux d'instructions AVX-512 (et ultérieurs) seront probablement pris en charge par tous les futurs processeurs Intel x86 haut de gamme. Les processeurs plus modestes et à faible consommation, dotés d'unités d'exécution de 128 ou 256 bits, peuvent exécuter une instruction de 512 bits sous la forme de deux micro-opérations de 256 bits ou de quatre micro-opérations de 128 bits. Tous les processeurs compatibles AVX offriront des opérations de lecture non alignée relativement rapides, car les instructions AVX imposent peu de contraintes d'alignement. La tendance étant à l'adoption de deux ports de lecture, il est peu probable que les opérations de lecture en mémoire constituent un goulot d'étranglement, même en cas d'absence d'alignement. Par conséquent, on peut s'attendre à ce que la méthode de lecture non alignée (méthode 4) soit performante sur les futurs processeurs.

De nombreux processeurs modernes ont optimisé l'instruction `REP MOVSB` (méthode 1) pour utiliser la plus grande taille de registre disponible ainsi que la méthode la plus rapide, du moins dans les cas simples. Il subsiste toutefois des situations où la méthode `REP MOVSB` s'avère lente, par exemple en cas de mauvais alignement ou de fausse dépendance mémoire. L'instruction `REP MOVSB` présente néanmoins l'avantage d'être susceptible d'exploiter la taille de registre maximale sur les futurs processeurs dotés de registres dépassant 512 bits. Comme il est encore impossible de coder et de tester des instructions adaptées aux futures tailles de registre, `REP MOVSB` constitue le seul moyen d'écrire aujourd'hui du code capable de tirer parti des futures extensions de taille de registre. Il peut donc s'avérer judicieux d'utiliser l'instruction `REP MOVSB` pour les cas favorables impliquant de grands blocs de mémoire alignés.

Il serait utile que les fabricants de processeurs fournissent des informations via l'instruction `CPUID` pour faciliter le choix du code optimal ou, à défaut, qu'ils implémentent la meilleure méthode sous forme de microcode au sein de l'instruction `REP MOVSB`.

Des exemples des méthodes susmentionnées sont disponibles dans la bibliothèque de fonctions accessible à l'adresse www.agner.org/optimize/asmlib.zip. Consultez le manuel 1, “Optimizing software in C++”, pour une analyse de la sélection automatique du code selon le processeur (*CPU dispatching*) et une comparaison des bibliothèques de fonctions disponibles.

Des conseils spécifiques aux processeurs pour améliorer les accès mémoire figurent dans les documents “Intel 64 and IA-32 Architectures Optimization Reference Manual” d'Intel et “Software Optimization Guide for AMD Family xx Processors” d'AMD.

17.9 Code auto-modifiable

La pénalité liée à l'exécution d'un segment de code immédiatement après sa modification est d'environ 19 cycles d'horloge pour le P1, 31 pour le PMMX et entre 150 et 300 pour les PPro, P2, P3 et PM. Le P4 vide entièrement le cache de traces (trace cache) après l'exécution d'un code auto-modifiant. Les processeurs 80486 et antérieurs nécessitent une instruction de saut entre le code modificateur et le code modifié afin de vider le cache d'instructions.

Pour obtenir l'autorisation de modifier du code sous un système d'exploitation protégé, il est nécessaire d'appeler des fonctions système spécifiques : `ChangeSelector` sous Windows 16 bits, et `VirtualProtect` ainsi que `FlushInstructionCache` sous Windows 32 et 64 bits. L'astuce consistant à placer le code dans un segment de données ne fonctionne pas sur les systèmes récents qui utilisent le bit de désactivation de l'exécution (NX bit).

Le code auto-modifiant n'est pas considéré comme une bonne pratique de programmation. Il ne doit être utilisé que si le gain de vitesse est substantiel et si le code modifié est exécuté un nombre de fois suffisant pour que cet avantage compense les pénalités inhérentes à l'utilisation de cette technique.

Le code auto-modifiant peut s'avérer utile, par exemple, dans un programme de calcul mathématique où une fonction définie par l'utilisateur doit être évaluée à de nombreuses reprises. Le programme peut alors intégrer un compilateur léger chargé de convertir cette fonction en code binaire.

18 Mesure de performance

18.1 Test de vitesse

De nombreux compilateurs intègrent un profileur permettant de mesurer la fréquence d'appel de chaque fonction d'un programme ainsi que sa durée d'exécution. Cet outil est très utile pour identifier les zones critiques ("hot spots") du programme. Si une zone critique particulière accapare une part importante du temps d'exécution total, c'est sur elle que vous devriez concentrer vos efforts d'optimisation.

Beaucoup de profileurs manquent de précision, et ne sont certainement pas assez précis pour affiner une petite portion de code. Une méthode plus fiable pour mesurer la vitesse d'exécution d'un segment de code consiste à utiliser le compteur d'horodatage (*time stamp counter*). Il s'agit d'un compteur d'horloge interne de 64 bits dont la valeur peut être lue dans le couple de registres EDX:EAX à l'aide de l'instruction RDTSC (*Read Time Stamp Counter*). Ce compteur s'incrémente à la fréquence d'horloge du processeur ; ainsi, une unité de comptage correspond à un cycle d'horloge, ce qui constitue la plus petite unité de temps pertinente.

De nombreux microprocesseurs sont capables de modifier la fréquence d'horloge du cœur du processeur en fonction des variations de la charge de travail. La fréquence d'horloge est augmentée lorsque la charge est élevée et réduite lorsqu'elle est faible, afin d'économiser de l'énergie. Les processeurs Intel disposent d'un "compteur d'horloge du cœur" (*core clock counter*) qui s'incrémente à la fréquence réelle de l'horloge du cœur. Ce compteur fournit des résultats plus constants et reproductibles, ce qui est utile pour comparer différentes implémentations d'un même code, tandis que le compteur d'horodatage (*time stamp counter*) offre une mesure plus réaliste de la durée réelle d'exécution du code. L'activation du compteur d'horloge du cœur nécessite un pilote de périphérique, car cette opération requiert un accès privilégié. Une fois activé, le compteur peut être lu sans mode privilège. Un programme de test permettant d'activer ce compteur est présenté ci-dessous.

Il est nécessaire d'insérer la séquence d'instructions XOR EAX, EAX / CPUID avant et après chaque lecture des compteurs afin d'empêcher toute exécution parallèle avec d'autres opérations sur les processeurs à exécution dans le désordre (*out-of-order execution*). L'instruction CPUID est une instruction de sérialisation ; elle vide le pipeline et attend l'achèvement de toutes les opérations en attente avant de poursuivre. Cette caractéristique est très utile pour les tests.

L'un des problèmes majeurs lors du comptage des cycles d'horloge est la nécessité d'éviter les interruptions. Les systèmes d'exploitation protégés ne permettent pas d'effacer le flag d'interruption ; il est donc impossible d'éviter les interruptions et les changements de tâche pendant le test. Cela rend les résultats imprécis et non reproductibles. Il existe plusieurs solutions alternatives pour remédier à ce problème :

1. Exécutez le code de test avec une priorité élevée afin de minimiser les risques d'interruptions et de changements de tâche.
2. Si le segment de code testé est suffisamment court, vous pouvez répéter le test plusieurs fois et considérer que la valeur la plus faible relevée correspond à une situation sans interruption.
3. Si le segment de code testé est d'une longueur telle que les interruptions sont inévitables, vous pouvez répéter le test à plusieurs reprises et calculer la moyenne des mesures effectuées.
4. Créez un pilote de périphérique virtuel pour effacer le flag d'interruption.
5. Utilisez un système d'exploitation permettant d'effacer le flag d'interruption (par exemple, Windows 98 sans réseau, en mode console).
6. Lancez le programme de test en mode réel sous l'ancien système d'exploitation DOS.

J'ai élaboré une série de programmes de test utilisant les méthodes 1 et 2. Ces programmes sont disponibles à l'adresse www.agner.org/optimize/testp.zip.

Une complication supplémentaire survient sur les processeurs multicœurs si un thread est susceptible de basculer d'un cœur à un autre. Les compteurs d'horodatage des différents cœurs ne sont pas nécessairement synchronisés. Cela ne pose pas de problème lors du test de courts segments de code, à condition de prendre les précautions susmentionnées pour minimiser les interruptions. En revanche, cela peut s'avérer problématique lors de la mesure d'intervalles de temps plus longs. Il peut être nécessaire de verrouiller le processus sur un seul cœur de processeur, par exemple en utilisant la fonction `SetProcessAffinityMask` sous Windows. Ce point est abordé dans le document "Game Timing and Multicore Processors" (Microsoft, 2005) : <http://msdn.microsoft.com/enus/library/ee417693.aspx>. L'instruction RDTSCP permet de détecter si la tâche a basculé vers un autre cœur de processeur.

En mesurant les cycles d'horloge, vous constaterez rapidement qu'un segment de code met toujours plus de temps à s'exécuter lors de sa première exécution, lorsqu'il ne figure pas encore dans le cache. De plus, il peut falloir deux ou trois itérations pour que le prédicteur de branchement s'adapte au code. La première mesure indique le temps d'exécution lorsque le code et les données ne sont pas en cache, tandis que les mesures suivantes reflètent le temps d'exécution avec une mise en cache optimale.

Sur certains anciens processeurs Intel, les effets d'alignement peuvent compliquer les mesures temporelles. Supposons que vous souhaitiez modifier un segment de code pour le rendre plus rapide de quelques cycles d'horloge. Le code modifié n'ayant pas exactement la même taille que l'original, le code situé après le point de modification

se retrouvera avec un alignement différent, modifiant ainsi les blocs de récupération des instructions. Si la récupération et le décodage des instructions constituent un goulot d'étranglement, ce changement d'alignement peut accélérer ou ralentir l'exécution du code de plusieurs cycles d'horloge. Des effets d'alignement peuvent également être observés sur les processeurs AMD lors de l'exécution de très petites boucles.

La plupart des processeurs x86 disposent également d'un ensemble de compteurs dits de surveillance des performances. Ces compteurs peuvent compter des événements tels que des échecs de cache, des désalignements, des erreurs de prédiction de branche, etc. Ils sont très utiles pour diagnostiquer les problèmes de performances. Les compteurs du moniteur de performances sont spécifiques au processeur. Vous avez besoin d'une configuration de test différente pour chaque type de processeur.

Des détails sur les compteurs du moniteur de performances sont disponibles dans le manuel du développeur de logiciels d'Intel et dans le guide du développeur du BIOS et du noyau d'AMD.

Vous avez besoin d'un accès privilégié pour configurer les compteurs du moniteur de performances. Cela se fait plus facilement avec un pilote de périphérique. Les programmes de test sur www.agner.org/optimize/testp.zip donnent accès aux compteurs du moniteur de performances sous Linux et Windows. Ce programme de test prend en charge les différents types de compteurs de surveillance de performances dans la plupart des processeurs Intel, AMD et VIA.

Intel et AMD proposent des profileurs qui utilisent les compteurs de performances de leurs processeurs respectifs. Le profileur d'Intel s'appelle Vtune et celui d'AMD s'appelle CodeXL.

18.2 Les pièges des tests unitaires

Pour déterminer quelle version d'une fonction offre les meilleures performances, il ne suffit pas de mesurer les cycles d'horloge au sein d'un programme de test restreint qui appelle la fonction à maintes reprises. Un tel test a peu de chances de fournir une mesure réaliste des défauts de cache, car le programme de test pourrait utiliser moins de mémoire que la capacité du cache. Par exemple, un test isolé pourrait indiquer qu'il est avantageux de dérouler une boucle, alors qu'un test intégrant la fonction au programme final révélerait un nombre élevé de défauts de cache avec ce même déroulage.

Par conséquent, lors de l'évaluation des performances d'une fonction, il est important non seulement de comptabiliser les cycles d'horloge, mais aussi de prendre en compte l'espace qu'elle occupe dans le cache d'instructions, le cache de données et le tampon de cibles de branchement (BTB).

Pour plus de détails sur cette question, consultez la section intitulée "The pitfalls of unit-testing" (Les pièges des tests unitaires) dans le manuel 1 : "Optimizing software in C++".

19 Bibliographie

Ce manuel fait partie d'une série disponible sur www.agner.org/optimize, comme indiqué dans l'introduction (page 1). Consultez le manuel n° 3, "The microarchitecture of Intel, AMD, and VIA CPUs", pour une liste de documents pertinents concernant des processeurs spécifiques.

De nombreuses autres sources contiennent également des informations utiles. Certaines ressources intéressantes sont répertoriées sur www.agner.org/optimize.

Voici quelques ouvrages utiles :

R. C. Detmer: Introduction to 80x86 Assembly Language and Computer Architecture, 2nd ed. Jones & Bartlett, 2006. Jones & Bartlett, 2006.

Une bonne introduction à la programmation en assembleur.

R. Hyde: The Art of 64-Bit Assembly, vol. 1+2.

Programmation avancée en langage assembleur. Windows et Microsoft uniquement.

J. Duntemann: Assembly Language Step By Step. Various versions.

J. L. Hennessy and D. A. Patterson: Computer Architecture: A Quantitative Approach, 3rd ed. 2002.

Bon manuel sur l'architecture et la microarchitecture des ordinateurs

John R. Levine: Linkers and Loaders. Morgan Kaufmann, 2000.

Explique le fonctionnement des éditeurs de liens et des chargeurs.

Henry S. Warren, Jr.: "Hacker's Delight". Addison-Wesley, 2003.

Contient de nombreuses astuces de manipulation de bits.

20 Copyright

Cette série de cinq manuels est protégée par le droit d'auteur d'Agner Fog. Sa diffusion publique et sa reproduction sur d'autres sites (*mirroring*) sont interdites. La diffusion non publique auprès d'un public restreint, à des

fins éducatives, est autorisée. Les exemples de code contenus dans ces manuels peuvent être utilisés sans restriction. Une licence Creative Commons CC-BY-SA entrera automatiquement en vigueur à mon décès.

Voir <https://creativecommons.org/licenses/by-sa/4.0/legalcode>