Architecture des Ordinateurs
Première partie

Cécile Germain Daniel Étiemble

Licence d'Informatique - IUP Miage - FIIFO

Table des matières

Chapitre 1

Introduction

Un cours sur l'Architecture des Ordinateurs demande de définir ce que sont, d'une part un ordinateur, d'autre part l'architecture. Le lecteur qui s'apprête à aborder le troiseme millénaire a certainement une intuition sur l'ordinateur, mais peut-être pas d'idée précise sur la notion d'architecture.

Un ordinateur est un **machine** qui traite une information fournie par un **organe d'entrée** suivant un **programme** et délivre une information sur un **organe de sortie**. La partie de l'ordinateur chargée du traitement de l'information est appelée Unité Centrale (UC) ou Central Processing Unit (CPU).

L'explosion des jeux informatiques et videos a certainement diffusé auprès du "grand public" une connaissance fonctionnelle de certains composants architecturaux : chacun sait que, pour jouer raisonnablement sur un PC, il vaut mieux qu'il soit équipé d'une carte 3D, et beaucoup connaissent la hiérarchie en termes de rapport qualité/prix de ces cartes. En tant que sous-discipline scientifique de l'informatique, aussi bien qu'en tant que compétence professionnelle, l'architecture des ordinateurs est plus que cette connaissance encyclopédique.

Matériellement, un ordinateur est composé de cartes et de périphériques (écran, clavier, disques etc.) ; chaque carte est elle-même construite à partir de composants électroniques, qui utilisent depuis les années 60 la technologie des transistors et depuis les années 70 la technologie des circuits intégrés.

Ces divers organes doivent être conçus et organisés pour trouver un optimum suivant les critères de *fonctionnalité*, de *performances* et de *prix*. Le plus important de ces organes est le processeur, qui est composé d'un ou d'un petit nombre de circuits intégrés.

Dans le sens le plus général, l'architecture est donc la conception et l'organisation des composants matériels de l'ordinateur basés sur la technologie des circuits intégrés, et plus particulièrement du processeur.

Le terme de matériel dans la définition ci-dessus ne signifie pas que l'architecte dessine des transistors.

Le fonctionnement d'un ordinateur peut s'envisager suivant la hiérarchie des niveaux

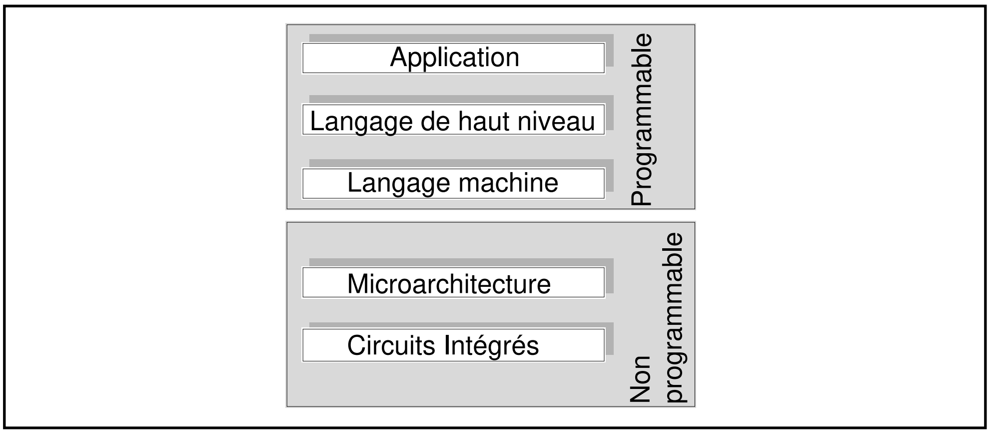

Figure 1.1: Les niveaux d'un système informatique

décrite dans la fig 1.1. L'utilisateur final voit une application plus ou moins interactive, par exemple un jeu, ou un logiciel de traitement de texte. Cette couche est réalisée essentiellement suivant un modèle de calcul figé dans un langage de haut niveau, qui utilise des mécanismes génériques. Par exemple, presque tous les langages connaissent la notion d'appel fonctionnel récursif, qui requiert sur les processeurs un mécanisme de pile et un mécanisme de rupture de séquence (ceci sera expliqué plus loin).

Chaque processeur dispose d'un langage spécifique, son *jeu d'instructions*, qui implémente ces mécanismes communs. Le jeu d'instruction est également appelé langage-machine, ou encore architecture logicielle. De façon précise, le jeu d'instruction est une abstraction programmable (i.e. utilisable pour la programmation) du processeur. Les niveaux suivants sont figés dans le matériel, et ne sont pas reprogrammables. Le jeu d'instruction est donc l'interface de programmation, l'API, du processeur.

Le jeu d'instruction est implémenté par divers circuits logiques : registres, UAL etc. L'organisation et la coordination de ces organes constitue l'architecture matérielle du processeur. De même qu'une API est destinée à recevoir plusieur implémentations, ou une voiture plusieurs motorisations, un jeu d'instruction correspond en général à plusieurs architectures matérielles différentes. Finalement, les circuits logiques abstraits sont instanciés dans une technologie donnée.

L'interaction entre les niveaux de la hérarchie n'est pas simplement descendante. Une architecture logicielle est d'une part conçue pour permettre une compilation commode et efficace des langages de haut niveau, mais aussi en fonction des possibilités d'implantation matérielle. L'exemple le plus élémentaire est la taille des mots que le processeur peut traiter : 8 bits dans les annéess 70, 32 et plus souvent 64 maintenant. Plus profondément, la période récente (années 85- 90) a vu une convergence forte des jeux d'instructions des processeurs non Intel vers un noyau RISC. Un des objectifs de ce cours est de montrer comment ce type d'architecture logicielle découle de l'état de la technologie à cette période.

Le terme d'architecture a été initialement défini comme èquivalent à celui de jeu d'instructions. La première "architecture" en ce sens est l'IBM 360, dont le cycle de vie s'est étendu de 64 à 86. De même, l'architecture 8086 est incluse dans toutes les architectures Intel qui lui ont succédé. Auparavant, les niveaux de spécification (jeu d'instruction) et de réalisation n'étaient pas distingués. L'intérêt évident est d'assurer la compatibilité totale sans recompilation des applications. C'est un aspect supplémentaire, et très important, de la fonctionnalité. En particulier, l'architecture logicielle x86 possède de nombreuses caractéristiques (nombre de registres, modes d'adressage) qui expriment les contraintes technologiques des années 70, et ne sont plus du tout adaptées à la technologie actuelle. Cependant, Intel a maintenu la compatibilité binaire jusqu'au P3, à cause de l'énorme base installée.

La convergence vers les jeux d'instruction RISC a ensuite mis l'accent sur l'aspect organisation, qui est alors devenu le principal enjeu scientifique et commercial de l'architecture. La période qui s'ouvre verra probablement un renouveau de la variabilité des jeux d'instruction. En particulier l'IA-64 d'Intel offre une fonctionnalité nouvelle dans le jeu d'instruction, qui n'a jamais existé prćédemment dans des ordinateurs commerciaux.

Le cours traite les deux aspects de l'architecture. La première partie étudie l'architecture logicielle, à travers le codage de l'information, et les jeux d'instruction. On étudiera en particulier la relation entre les structures des langages de haut niveau et le langage machine. La deuxième partie étudie l'architecture matérielle, du processeur, de la hiérarchie mémoire, et des organes d'entrées-sortie. Cette partie étudiera en particulier les conséqences des contraintes technologiques sur les modèles d'exécution, la gestion de la mémoire et les entrées-sorties.

Deux excellents livres correspondent à ce cours [2, 1]. Ils ont été tous deux écrits par les architectes des premiers microprocesseurs RISC. [2] est un synthèse abordable pour un étudiant de licence, mais quand même assez difficile. Ce livre a très fortement marqué la conception d'architectures, en imposant une discipline d'approche quantitative. [1] est plus élémentaire. [3] est une synthèse qui présente les caractéristiques fondamentales des architectures RISC.

Chapitre 2

Les composantes de l'ordinateur

2.1 Le modèle de Von Neumann

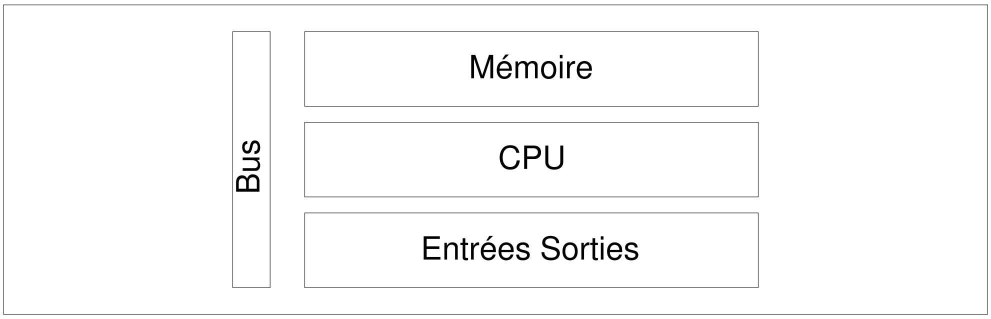

Figure 2.1: Principe de l'organisation d'un ordinateur

Définition 1 *L'ordinateur est une machine électronique, qui traite l'information dans une unité centrale (UC, ou CPU pour Central Processing Unit), selon un programme qui est enregistré en mémoire. Les données fournies en entrée par un organe d'entrée (par exemple de type clavier) sont traitées par l'unité centrale en fonction du programme pour délivrer les résultats en sortie, via un organe de sortie (par exemple un écran).*

Cette définition très générale est résumée dans la fig. 2.1. Elle implique de définir ce que sont l'information et le traitement. L'information est numérisée, c'est à dire limitée à des valeurs discrètes, en l'occurence binaires ; les détails du codage binaire de l'information sont traités au chapitre suivant. Par ailleurs, la définition implique que le comportement d'un programme qui ne fait pas d'entrées-sorties ne peut être spécifié.

Le traitement suit le *modèle d'exécution de Von Neumann.*

- La mémoire contient les instructions et les données.

- La mémoire est formée d'un ensemble de mots de longueur fixe, chaque mot contenant une information codée en binaire. Chaque mot de la mémoire est accessible par l'intermédiaire de l'adresse mémoire. Le temps d'accès à un mot est le même quelle que soit la place du mot en mémoire : ce type d'accès est appelé aléatoire, et les mémoires sont appelées RAM (random access memory).

- Les instructions sont exécutées en séquence. Le CPU conserve l'adresse de la prochaine instruction à exécuter dans un registre, appelé PC (Program Counter) ; pour chaque instruction, le CPU effectue les actions suivantes :

  - lire l'instruction a l'adresse PC et incrémenter PC ;

  - exécuter l'instruction.

Le premier ordinateur a été la machine ENIAC, construite à l'université de Pennsylvannie (Moore School) pendant la seconde guerre mondiale, par les ingénieurs J. P. Eckert et J. Mauchly. Elle avait été commandée par l'armée américaine pour le calcul des tables de tir pour l'artillerie. L'ENIAC a été la première machine *programmable*, c'est à dire dont la séquence d'opérations n'était pas prédéfinie, contrairement à un automate du type caisse enregisteuse. Mais la nouveauté radicale de cette approche n'était pas complètement perçue, ce dont témoigne le nom de la machine : ENIAC signifie *Electronic Numerical Integrator And Calculator*. C'est le mathématicien John Von Neuman, intégré en 1944 au projet ENIAC, qui formalisa les concepts présents dans ce premier ordinateur, et les développa sous le nom de projet EDVAC : *Electronic Discrete Variable Automatic Computer*. La création d'un nouveau terme, Computer et non Calculator, correspond à une rupture théorique fondamentale. Comme cela se reproduira souvent dans l'histoire de l'informatique, ce progrès théorique majeur aboutit dans le très court terme à un semi-échec pratique : l'équipe se dispersa, et l'EDVAC ne fut opérationnel qu'en 1952. Von Neumann contribua à la construction d'une machine prototype universitaire à Princeton, L'IAS. Eckert et J. Mauchly fondèrent une entreprise qui réalisa le premier ordinateur commercial, l'UNIVAC-1, dont 48 exemplaires furent vendus. On trouvera un résumé de l'évolution des calculateurs et des ordinateurs dans [4].

La réalisation de l'organisation abstraite de la fig. 2.1 a fortement varié depuis l'origine des ordinateurs. La fig. 2.2 décrit l'organisation matérielle typique d'un ordinateur monoprocesseur (ie avec un seul CPU). Les composants matériels ne recoupent pas exactement l'organisation abstraite : la mémoire est réalisée par un ensemble de composants matériels, depuis un circuit intégrés dans le composant processeur (cache L1), jusqu'aux disques. Les disques appartiennent au domaine des entrées-sorties par leur gestion, et en tant que support des systèmes de fichiers, mais aussi à la hiérarchie mémoire, car ils contiennent une partie des données et des instructions adressables par le processeur. Les entrées-sorties incluent des interfaces, qui sont elles-mêmes des processeurs programmables.

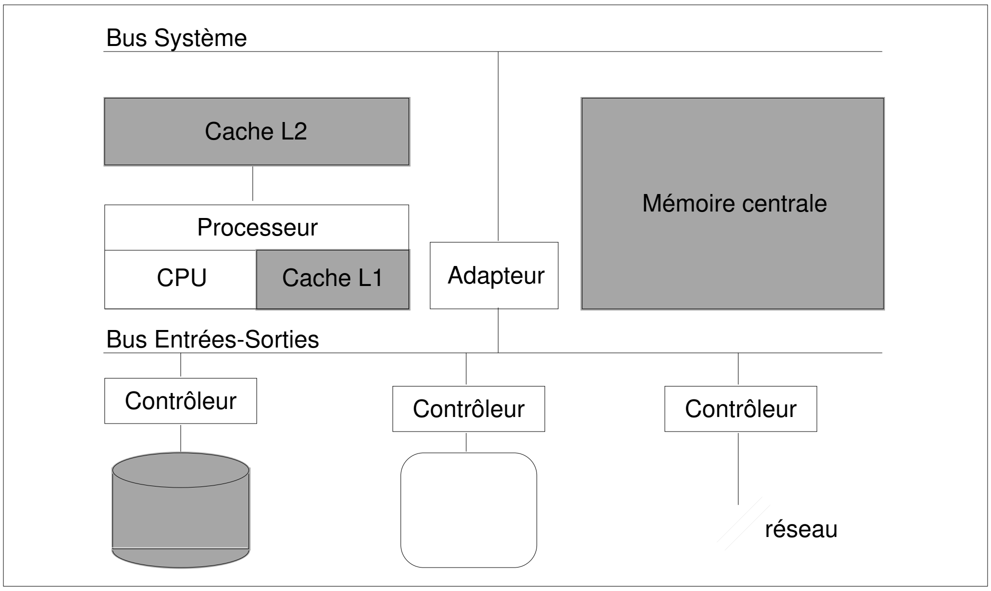

Figure 2.2: Architecture d'un ordinateur monoprocesseur en 99

2.2 Les faits technologiques

L'évolution des architectures repose sur la *loi de Moore* :

Définition 2 *Le nombre de transistors dans un circuit double approximativement tous les 18 mois.*

Cette évolution exponentielle de la capcité de calcul est accompagné d'une augmentation, également exponentielle, de la fréquence de fonctionnement, qui est multipliée par un facteur 1,24 tous les ans (fig. 2.3).

La loi de Moore est très souvent incorrectement interprétée comme un doublement des *performances* tous les 18 mois, ce qui n'est pas exact : on verra plus loin que la performance progresse exponentiellement, mais moins vite.

La loi de Moore est reliée à deux phénomènes : d'une part l'augmentation de la densité d'intégration, c'est à dire du nombre de transistors intégrables par unité de surface, d'autre part, l'augmentation de la taille des circuits intégrés. Elle traduit globalement le nombre d'opérateurs matériels, et donc de fonctionnalités, qui peuvent être intégrées dans un processeur. Le produit du nombre d'opérateurs par la fréquence d'activation traduit donc le nombre d'opérations potentiellement disponibles par seconde.

2.3 L'unité centrale

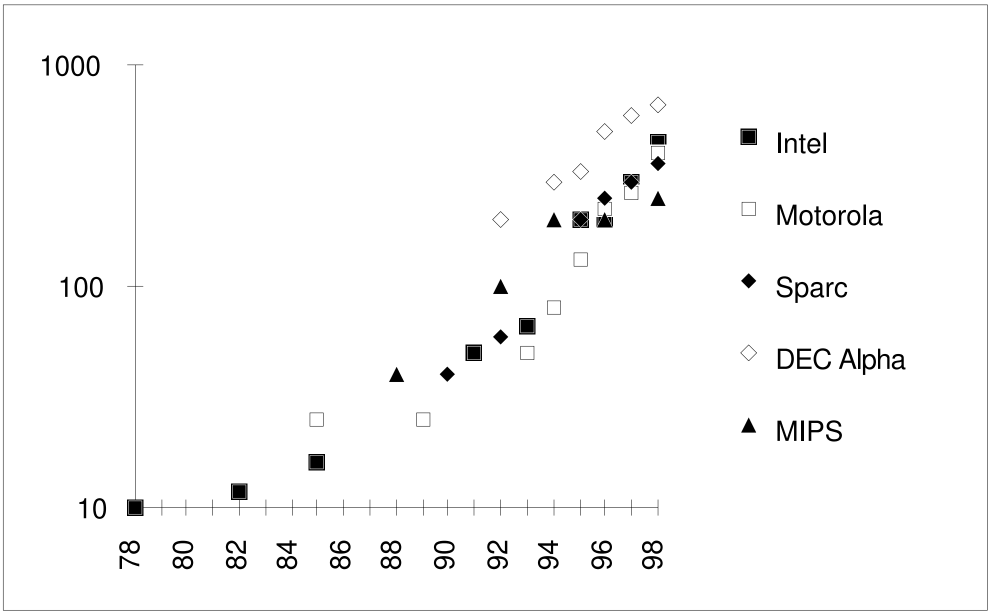

Figure 2.3: Evolution de la fréquence des processeurs

Pour exécuter un programme, l'unité centrale appelle les instructions depuis la mémoire, et les données que traitent ces instructions. En effet, dans l'état actuel de la technologie, le CPU ne peut travailler que sur des informations physiquement stockées "près" des organes de calcul qu'elle contient.

L'unité centrale se décompose en une *partie opérative*, ou *chemin de données* et *une partie contrôle* (fig. 2.4).

Le chemin de données est organisé autour d'une unité arithmétique et logique (UAL) et d'opérateurs arithmétiques flottants qui effectuent les opérations sur les données. Les

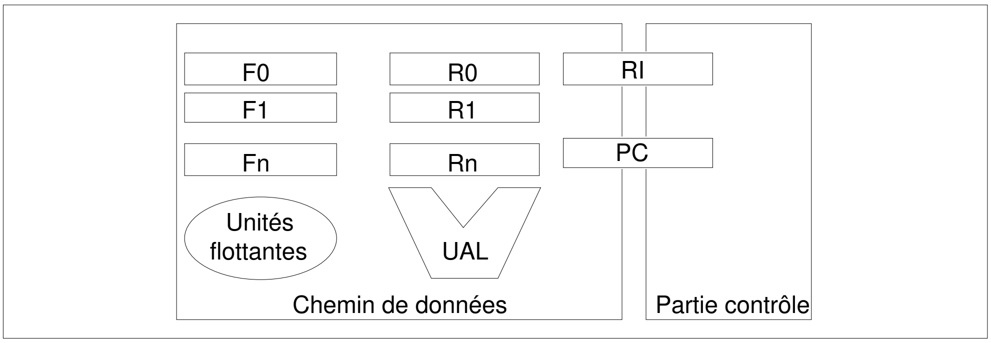

Figure 2.4: Organisation de l'unité centrale

opérandes, initialement lus en mémoire, et les résultats des opérations, sont stockés dans des organes de mémorisations internes au CPU, les *registres*. Certains de ces registres sont visibles : leur nom (ou leur numéro) est présent dans l'instruction, et le programmeur en langage machine peut décider quel registre lire ou écrire ; par exemple, l'instruction ADD R1, R2, R3 nomme les trois registres R1, R2 et R3. D'autres registres ne sont pas visibles, et servent à des besoins de stockage temporaire. Le chemin de données détermine les deux caractéristiques fondamentales d'une unité centrale, le *temps de cycle* et la *largeur du chemin de données*.

- Le temps de cycle T_c est essentiellement le temps d'une opération de l'UAL ; l'inverse du temps de cycle est la fréquence ; si l'unité de temps de cycle est la seconde, l'unité de fréquence est le Hertz. Par exemple, un processeur cadencé à 500MHz a un temps de cycle de $\frac{1}{500 \times 10^6} = 2ns = 2 \times 10^{-9}s$.
- La largeur du chemin de données est la taille de l'information traitée par la partie opérative. Les premiers microprocesseurs étaient des 8 bits, les processeurs généralistes actuels sont 32 ou 64 bits, le processeur de la Playstation 2 est un 128 bits.

On a mentionné ci-dessus "l'instruction ADD R1, R2, R3", alors que les instructions sont des mots binaires. En fait, le codage binaire des instructions n'étant pas très pratique pour l'utilisateur humain, les instructions peuvent être décrites par un langage rudimentaire, le langage d'assemblage. Celui-ci comporte des mnémoniques, qui décrivent l'opération, et une description des opérandes. C'est aussi le langage qu'utilise un programmeur humain. Le langage d'assemblage n'a aucune réalité au niveau de la machine ; la traduction en binaire est réalisée par un utilitaire, *l'assembleur*. La correspondance entre instruction binaire et langage d'assemblage est très élémentaire, du type "traduction mot-à-mot" (contrairement à la compilation). Le codage est si immédiat qu'il est réversible : les debogueurs (dbx, gdb) contiennent des désassembleurs, qui effectuent le codage inverse, du binaire vers le langage d'assemblage. Dans la suite, on utilisera un langage d'assemblage générique, où les registres seront notés R0 à R31, et la syntaxe est du type dest-source, par exemple

ADD R1, R2, R3 ; une instruction d'addition

signifie R1 $\leftarrow$ R2 + R3 , le ";" signalant le début d'un commentaire.

La partie contrôle effectue le séquencement des différentes instructions en fonction des résultats des opérations et actionne les circuits logiques de base pour réaliser les transferts et traitements de données. Elle contrôle également la synchronisation avec les autres composants.

A cheval entre le chemin de données et la partie contrôle, on trouve deux registres spécialisés : le compteur de programme PC, dont on a déjà parlé, et le *registre instruction* RI. Comme les autres informations, l'instruction ne peut être traitée par le CPU que si elle y est physiquement présente. La lecture de l'instruction consiste donc à copier l'instruction depuis la mémoire vers le registre RI. Comme le registre RI peut contenir une constante opérande, il appartient au chemin de données ; comme il contient l'instruction, que la partie contrôle décode pour activer les circuits logiques qui exécutent l'instruction, il appartient aussi à la partie contrôle. PC contrôle la lecture de l'instruction, mais il peut aussi être modifié par les instructions qui réalisent des branchements.

2.4 La mémoire

Les mémoires contiennent le programme (instructions et données). Les informations mémorisées sont repérées par une adresse. On distingue les mémoires à accès aléatoires (RAM) réalisées avec des technologies semiconducteurs, et les mémoires secondaires, réalisées essentiellement avec des supports magnétiques.

Les principes de fonctionnement des mémoires secondaires, comme les disques et les disquettes, seront présentés dans le chapitre sur les Entrées-Sorties. La caractéristique importante est un temps d'accès très grand par rapport aux temps de cycle de l'UC : pour les disques, il se chiffre en dizaines de μs.

Les RAM

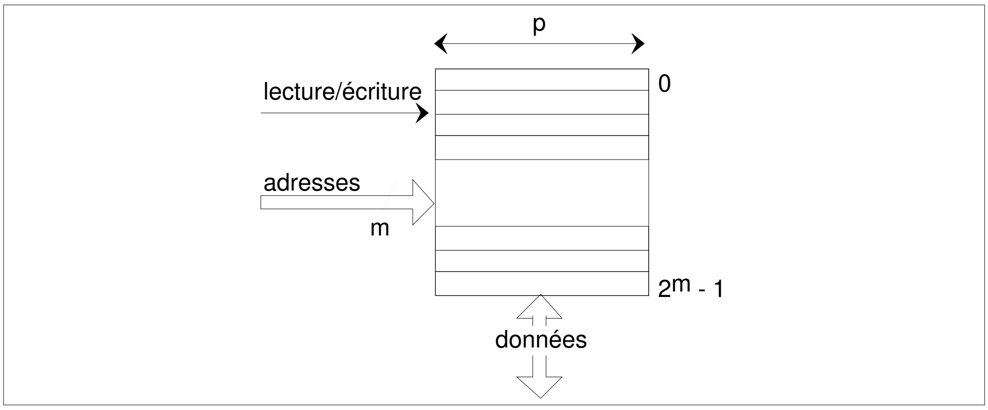

Figure 2.5: Schéma fonctionnel d'une mémoire RAM

Les RAM sont caractérisées par le fait qu'un mot peut être lu ou écrit en un temps identique quelle que soit son adresse. Un mot est repéré par une adresse : une adresse sur m bits permet de repérer un mot de p bits parmi 2^m (fig. 2.5). Généralement, la plus petite unité adressable est l'octet (8 bits), mais la mémoire peut lire ou écrire un mot de plusieurs octets (4 ou 8 pour manipuler en un accès 32 ou 64 bits). Les mémoires RAM sont caractérisées par plusieurs paramètres.

Introduction	Capacité	Temps de cycle
1980	64 Kbits	250 ns
1983	256 Kbits	220 ns
1986	1 Mbits	190 ns
1989	4 Mbits	165 ns
1992	16 Mbits	120 ns
1995	64 Mbits	90 ns

Table 2.1: Évolution des mémoires dynamiques.

- *Taille*, *organisation* et *capacité* : nombre de mots, nombre de bits par mot et nombre total de bits.

- *Temps d'accès*, le temps entre l'envoi de l'adresse et l'obtention de la donnée, et *temps de cycle*, le temps entre deux opérations mémoire successives.

Les mémoires RAM sont réalisées avec les technologies à semiconducteurs, le plus souvent à base de transistors MOS

Il existe deux grandes classes de circuit réalisant une RAM, les RAM *statiques* (SRAM) et les RAM *dynamiques* (DRAM)

Les RAM statiques

Leur point mémoire est constitué avec des portes logiques. La mémorisation est permanente, tant que la mémoire est alimentée. La lecture n'est pas destructrice et le temps d'accès est identique au temps de cycle. La complexité du point mémoire est d'environ 6 transistors MOS. En première approximation, on peut dire que le temps d'accès des SRAM décroît exponentiellement en fonction des années, les temps d'accès étant d'autant plus grand que la taille mémoire utilisée est grande.

Cette information est importante. En effet, le temps de cycle des processeurs décroît exponentiellement en fonction des années. Il est donc possible, en jouant sur la taille des mémoires SRAM utilisées, d'adapter le temps d'accès des mémoires SRAM au temps de cycle des processeurs. L'adaptation est encore plus facile si l'on implante la mémoire SRAM sur la même puce que le processeur.

Les RAM dynamiques

Leur point mémoire est constitué avec une capacité et un transistor, et a une complexité équivalente à 1,5 transistor MOS. A surface égale, leur capacité est quatre fois plus élevée qu'une SRAM. Il existe une très grande variété de mémoires dynamiques, qu'on ne peut décrire dans le cadre de ce cours. On trouvera une synthèse récente dans [6].

La mémorisation est fondée sur un phénomène électrique (conservation de charges dans un condensateur). La première conséquence est que la lecture étant destructrice, il faut réécrire la valeur que l'on lit. Le temps de cycle d'une mémoire dynamique est donc au moins le double de son temps d'accès. La seconde conséquence est que la mémorisation n'est que transitoire, et il faut rafraîchir périodiquement tous les points mémoire. Ce rafraîchissement est fait automatiquement dans la plupart des boîtiers DRAM. La table. 2.1, d'après [2], montre l'évolution des temps de cycle et de la capacité des DRAM en fonction des années. On constate une décroissance quasi-linéaire en fonction des années, au lieu d'une décroissance exponentielle pour les SRAM. D'autre part, la capacité quadruple tous les trois ans (le taux annuel correspondant est 60%, mais les générations successives de DRAM s'étagent en fait par tranches de trois ans). C'est exactement le taux de la loi de Moore.

En résumé, à une période donnée, les boîtiers mémoire DRAM disponibles contiennent 4 fois plus de bits que les mémoires SRAM de technologie équivalente et sont moins chers, mais beaucoup plus lents.

La hiérarchie mémoire

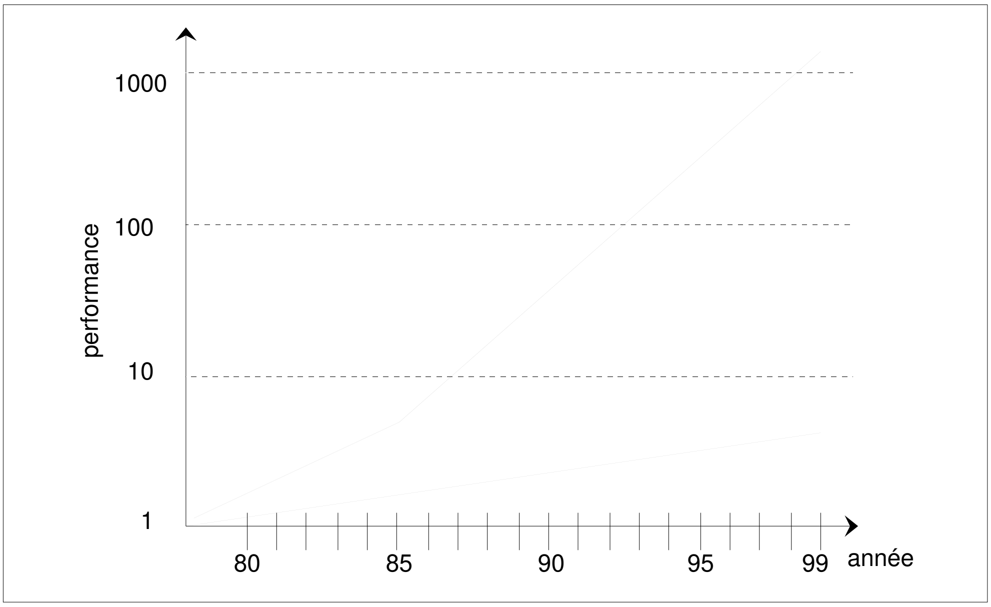

Figure 2.6: L'écart de performances entre processeur et DRAM

Les caractéristiques (quantité d'information stockée, temps d'accès et de cycle, coût par bit) des différentes mémoires conduisent à l'utilisation d'une hiérarchie de mémoires, allant de mémoires petites et rapides vers des mémoires plus grosses et plus lentes. Deux niveaux sont importants : la mémoire cache (SRAM) à côté de la mémoire principale (DRAM),

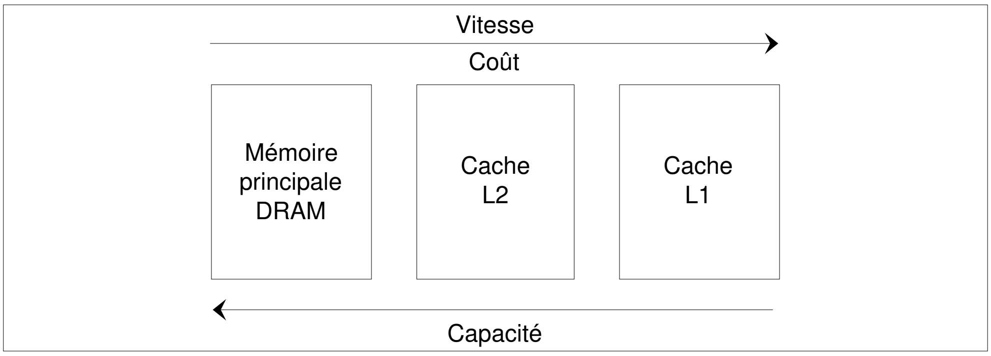

Figure 2.7: Hiérarchie caches - mémoire principale

	temps d'accès	Débit
L1 8 KO, intégré	6,7 ns ($2T_c$)	4800 MO/s
L2 96 KO, intégré	20 ns ($6T_c$)	4800 MO/s
L3 4 MO, externe	26 ns ($8T_c$)	960 MO/s
Mémoire principale	253 ns ($76T_c$)	1200 MO/s
Composant DRAM	60 ns ($18T_c$)	30 - 100 M0/s

Table 2.2: Performances de la hiérarchie mémoire de l'AlphaServer 8200 ; le processeur est un Alpha 21164 à 300 MHz.

et la mémoire virtuelle, constitué de la mémoire principale (DRAM) à côté de la mémoire secondaire (disque).

L'évolution des performances des processeurs et des DRAM diverge, car l'une est exponentielle et l'autre linéaire (fig. 2.6, d'après [5]). Mais la réalisation de la mémoire principale d'un ordinateur uniquement à partir de SRAM est exclue, pour des raisons économiques : le coût des mémoires constituant l'essentiel du coût des ordinateurs, l'utilisation de mémoires statiques à la place des mémoires dynamiques se traduirait par un facteur multiplicatif de l'ordre de 3 ou 4 du prix des machines.

Les architectures actuelles résolvent cette divergence par l'introduction de *caches*. Les caches sont des RAM statiques, qui contiennent un extrait, qu'on espère utile, du contenu de la mémoire principale, réalisée en DRAM (fig. 2.7). Idéalement, les caches permettront de soutenir le débit mémoire, en instruction et données, égal à celui du processeur. Le cache de premier niveau est en général intégré avec le CPU dans le circuit processeur ; le cache de second niveau peut être dans le même circuit (Alpha 21164), ou bien un autre circuit, qui peut être dans le même boîtier que le processeur (Pentium Pro), mais est le plus souvent à l'extérieur. Le passage à l'extérieur du circuit a pour conséquence que le temps d'accès à la mémoire principale peut être encore plus grand que le temps

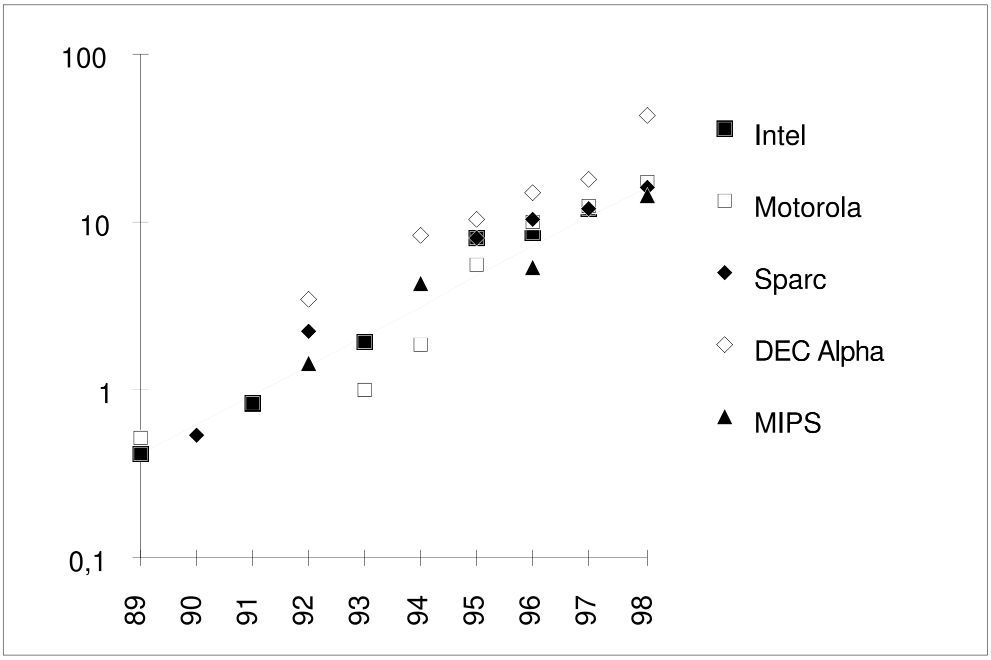

Figure 2.8: Evolution des performances des ordinateurs sur les benchmark SPEC. La ligne continue est une droite de pente 1,5.

d'accès au circuit DRAM correspondant. En revanche, une connexion large entre DRAMs et processeur permet d'équilibrer mieux les débits. La table 2.2 donne les performances d'une hiérachie mémoire agressive.

2.5 Performances

L'évolution exponentielle des performances

La performances des ordinateurs dépend de l'ensembles des composantes matérielles (CPU, mémoires, entrées-sorties) et logicielles (programme, système). La performance des composantes matérielles est elle-même le résultat de l'évolution exponentielle de la technologie (loi de Moore) et de l'architecture des processeurs et des ordinateurs. Le résultat net est actuellement une croissance exponentielle des performances (fig. 2.8) : la performance est multipliée d'un facteur environ 1,5 par an (augmentation de 50%).

Mesure de performances

La *définition* et la *mesure* des performances d'un ordinateur sont un problème difficile. Tout d'abord, les performances d'un ordinateur dépendent de l'utilisation visée. la mesure la plus courante est le temps d'exécution, mais le débit peut être plus important, par exemple pour une machine qui joue un rôle de serveur.

Ensuite, le temps d'exécution d'une tâche, de son début à sa fin, comprend le temps d'exécution du programme par le CPU, mais aussi les accès mémoire, les accès disque, les activités d'entrée-sortie et le temps utilisé pour les besoins du système d'exploitation. Il dépend donc des performances de l'ensemble des composantes matérielles et logicielles du système.

Les unités de mesure les plus simples sont les MIPS (million d'instructions par seconde) ou le MegaFLOP (million d'instructions flottantes par seconde). Ces deux performance a souvent été calculée de façon parfaitement trompeuse, comme le nombre d'instructions (ou d'instructions flottantes) que l'architecture permet d'exécuter en 1 cycle divisée par le temps de cycle. Ce chiffre représente alors seulement la *puissance crête*, c'est à dire ce que l'architecture ne peut en aucun cas dépasser. La première unité ne permet de toutes façons pas de comparer entre elles les performances de façon absolue, car elle dépend de la "quantité de travail" qu'effectue chaque instruction. Les MFlops *mesurés* sur une application sont plus significatifs, car ils mesurent de façon normalisée la quantité de travail utile que doit effectuer un algorithme numérique : additions et multiplications flottantes comptent pour 1 opération flottante, et toutes les autres opérations sont normalisées (par exemple, une racine carrée est comptée pour un certain nombre d'additions flottantes).

Pour comparer entre eux divers CPU ou divers ordinateurs, il est généralement admis que la seule manière correcte est la mesure du temps d'exécution sur des programmes réels pour des entrées déterminées. La définition de programmes tests représentatifs des applications, donc prédictifs des performances sur des applications réelles, Les programmes spécifiques d'évaluation de performance ou benchmark, n'est pas un problème complètement résolu actuellement. De tels programmes, appelés *benchmarks*, sont spécifiés par des consortiums industriels ou des organisations scientifiques. Certains, comme la suite des programmes SPEC, sont utilisés par les constructeurs pour évaluer la puissance de calcul brute des processeurs, sur du calcul entier (SPECint) ou flottant (SPECfp). Les mesures exprimées en SPEC n'ont de valeur que si les conditions d'expérimentation (fréquence d'horloge, hiérarchie mémoire utilisée, etc.) sont précisées. D'autres types de benchmarks sont plus orientés vers des applications utilisateurs. C'est le cas par exemple des programmes d'évaluation comme TP-1 qui mesurent le nombre de transactions par seconde (TPS) caractéristique de systèmes transactionnels. Dans ce cas, c'est l'ensemble des performances, incluant notamment le débit d'accès aux disques et le système d'exploitation, qui est évaluée sur une classe d'application caractéristique.

CPI et IPC

Pour mesurer et comparer les performances d'architectures, il faut donc d'abord travailler à programme utilisateur constant. Les performances d'une architecture, pour un programme donné, sont caractérisées par le temps écoulé sur l'exécution du programme utilisateur (hors temps système), selon la formule suivante :

$$\text{Temps}_{\text{exe}} = \text{NI} \times \text{CPI} \times T_c$$

où NI est le nombre d'instructions du programme, CPI est le nombre moyen de cycles d'horloge pour exécuter une instruction, et Tc est le temps de cycle. NI est fonction des instructions machine utilisables par compilateur, donc décrit la fonctionnalité de l'architecture logicielle. En première approximation, T_c est fonction de la technologie utilisée qui détermine le temps d'exécution des étapes élémentaires, par exemple la traversée de l'UAL, et décrit donc les contraintes technologiques.

On définit également le nombre d'instructions par cycle, IPC : IPC = 1/CPI. Le produit CPI$\times T_c$ est relié au MIPS par la relation :

$$\text{Nombre de MIPS} = \frac{\text{NI} \times 10^{-6}}{\text{Temps}_{\text{exe}}} = F \times \text{IPC}$$

si la fréquence F est exprimée en MHz.

CPI traduit les performances des architectures matérielles. Il permet d'abord de comparer des architectures matérielles qui implémentent la même architecture logicielle, en éliminant l'influence de la technologie (T_c) : typiquement, le CPI a doublé entre le pentium Pro et le PII.

CPI permet également de comparer les performances réelles avec les performances crêtes. Par exemple, l'étude [5] montre qu'un système processeur-mémoire à base de 21164 (l'AlphaServer 8200) atteint seulement une performance de 3,0 à 3,6 CPI, alors que le 21164 est capable d'exécuter 4 instructions par cycle, donc un CPI de 0,25. La valeur du CPI moyen mesuré sur des programmes réels, comparé au CPI minimum ou CPI optimal (CPIopt), va traduire la différence entre les performances réelles et les performances maximales de la machine. La valeur du CPI moyen mesuré sur des programmes réels, comparé au CPI minimum ou CPI optimal (CPIopt), va traduire la différence entre les performances réelles et les performances maximales de la machine, suivant la relation :

$$\text{CPI} = \text{CPI}_{\text{opt}}(1 + a + c + v + s)$$

CPI_{opt} correspond au fonctionnement idéal du séquencement des instructions qu'on étudiera aux chapitres 4 et 5 ; le terme a est lié à des aléas de fonctionnement à l'intérieur de l'exécution des instructions qu'on étudiera au chapitre 5 ; c et v sont des facteurs correctifs qui tiennent compte des limitations liées à l'existence de la hiérarchie mémoire constituée de caches, d'une mémoire principale et de mémoires secondaires (chapitres 6 et 7) ; s traduit l'impact du système d'exploitation, notamment pour les opérations d'entrées-sorties (chapitre 8).

CPI permet enfin de comparer des architectures de *processeur*, donc indépendamment du système mémoire : on compare alors plus précisément CPI_{opt}, ou $\text{CPI}_{\text{opt}}(1 + a)$. L'élément décisif qui a conduit au succès des architectures RISC est une valeur du CPI moyen bien meilleure à celle des architectures CISC. Elle est inférieure à 1,5 pour les premiers RISC commerciaux (SPARC de la société Sun, ou R2000 et R3000 de la société MIPS), alors qu'une architecture CISC comme celle du VAX-11 de Digital avait un CPI moyen de l'ordre de 6 à 8. $\mathrm{CPI}_{\mathrm{opt}}$ est de l'ordre de 1/4 à 1/8 actuellement.

Chapitre 3

Représentation de l'information

Les types d'informations traitées directement par un processeur ne sont pas très nombreux.

- Les données :
  - les entiers, avec deux sous-types, entiers naturels et relatifs ;
  - les flottants, qui décrivent les réels, avec également deux sous-types, simple et double précision.
  - les caractères, qui sont plutôt traités au niveau du logiciel de base.

  Le codage de ces trois types est actuellement définie formellement par des *standards*, c'est à dire des normes contraignantes spécifiées par des organisations internationales.

- Les instructions, dont le codage est spécifique d'un processeur.

Toute cette section traite de l'information par rapport aux mécanisme d'un ordinateur, et n'a donc pas de rapport avec la théorie de l'information utilisée dans la théorie du codage.

3.1 L'information

Dans un ordinateur, l'information est numérisée (digitale) :

Définition 3 *L'information est la connaissance d'un état parmi un nombre fini d'états possibles.*

Une information non numérisée est une information *analogique* : une grandeur physique continue, par exemple tension ou courant.

Etat	b_2	b_1	b_0
France	0	0	0
GB	0	0	1
Allemagne	0	1	0
Espagne	0	1	1
Italie	1	0	0
Portugal	1	0	1
Grèce	0	1	0
Irlande	1	1	1

Table 3.1: Représentation de huit Etats

Quantité d'information

L'unité de mesure de l'information est le *bit.*

Définition 4 *1 bit est quantité d'information liée à la connaissance d'un état parmi deux.*

1 bit d'information peut être représenté commodément par un digit binaire, prenant les valeurs 0 ou 1.

Avec n bits, on peut représenter 2^n configurations. La table 3.1 montre comment on peut représenter 8 états avec 3 bits.

Proposition 1 *La quantité d'information contenue dans la connaissance d'un état parmi N est $\lceil \log_2(N) \rceil$ bits.*

$$I = \lceil \log_2(N) \rceil$$

Pour avoir une idée intuitive des grandeurs mises en jeu, il suffit de remarquer que $2^{10} = 1024$, encore noté 1K. 2^{20} est donc de l'ordre du million : $2^{20} \approx (10^2)^3 = 10^6$.

La caractéristique la plus fondamentale, mais aussi la plus élémentaire d'un ordinateur est la taille de l'information qu'il est capables de manipuler. Le nombre d'états distincts représentés dans un ordinateur moderne est grand. En 74, le premier microprocesseur, le 8080 d'Intel, était un ordinateur 8 bits, correspondant à 256 états. En 99, tous les processeurs existants sont 32 bits ou 64 bits. 32 bits correspondent à $2^{32} = 2^2 \times 2^{30}$, soit 4 milliards.

Bien que ce nombre soit grand, il n'est pas infini. Une partie des applications informatiques travaille sur des ensembles finis : un éditeur de texte traite les caractères, un programme de gestion d'écran des pixels. A l'inverse, les applications numériques travaillent sur des ensembles non bornés : entiers naturels ou relatifs, rationnels, réels.

Les calculs d'ordinateurs sont donc **par construction** susceptibles d'erreur. On verra plus loin comment contrôler ces erreurs, mais les supprimer est impossible.

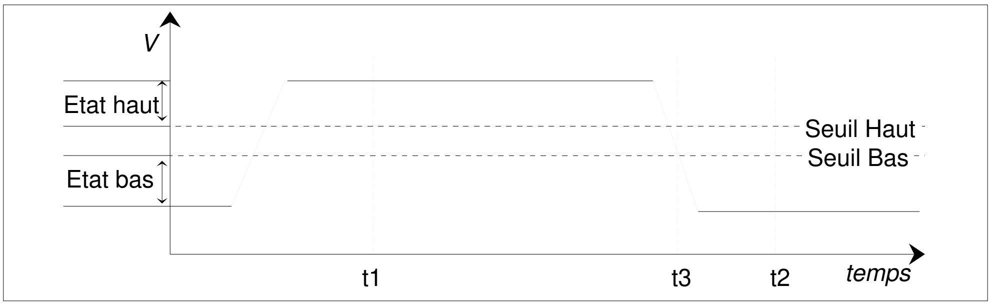

Figure 3.1: Information : état et temps.

Support de l'information

Bien que l'information soit numérisée, son support est physique, donc analogique : typiquement, dans la technologie actuelle des CI, la tension. La connaissance d'un état dépend donc de deux seuils, haut et bas.

La figure 3.1 illustre cette notion avec un signal électrique. Elle montre qu'il y a deux états significatifs, l'état bas lorsque la tension est inférieure à une référence basse, et un état haut lorsque la tension est supérieure à une référence haute. Le troisième état, situé entre les références basse et haute, ne peut être utilisé comme support d'information.

Donc, pour qu'il y ait information, il faut préciser l'instant auquel on regarde l'état du signal. : par exemple, en t_1 le signal est haut et en t_2, le signal est bas. En revanche, à l'instant t_3, le signal ne fournit aucune information et ne doit donc pas être échantilloné.

Notations

Mots binaires

Définition 5 *Un mot de n bits est une suite* $(a_i), 0 \leq i \leq n-1$ *;* a_0 *est le bit de* poids faible*,* a_n *est le bit de* poids fort*.*

La notation hexadécimale

La notation hexadécimale est une manière simplifiée d'écrire des mots binaires. Un mot binaire de n bits peut être écrit à l'aide de $\lceil n/4 \rceil$ digits hexadécimaux, en remplaçant chaque groupe de 4 digits binaires par le digit hexadécimal correspondant (table 3.2). Actuellement, l'usage de la notation hexdécimale ne correspond à aucun support matériel : il n'existe pas d'additionneur travaillant sur des digits hexadécimaux. En revanche, il est plus agréable d'ecrire dans un programme C 0x1234 que son équivalent binaire.

Chiffre Hexa	0	1	2	3	4	5	6	7
Code binaire	0000	0001	0010	0011	0100	0101	0110	0111
Chiffre Hexa	8	9	A	B	C	D	E	F
Code binaire	1000	1001	1010	1011	1100	1101	1110	1111

Table 3.2: Correspondance entre chiffres hexadécimaux et quartets binaires

3.2 Représentation des caractères

De nombreux standards existent. Le plus simple est le code ASCII (American Standard Code for Information Interchange) [7]. Il permet de représenter sur un octet des données alphanumériques, dont les caractères latins et les chiffres décimaux, et quelques autres informationscomme le retour chariot. Par exemple, la lettre "A" est codée par 41_H et le chiffre "9" par 39_H La représentation utilise en fait 7 bits, plus un bit de parité. Ce code a été établi par l'ANSI. Il a évolué vers le standard ISO 8859-1 (Latin-1), qui utilise les 8 bits pour représenter entre autres les caractères accentués : par exemple, le code CA_H représente Ê.

D'autres standards existent, en particulier Unicode [8]. Il utilise deux octets pour encoder aussi des jeux de caractères non latins, cyrilliques, hébreu, asiatiques. Unicode a été établi par le "Unicode consortium". Un encodage proche d'Unicode, l'UCS (Universal Character Set), est l'objet de la norme ISO 10646.

3.3 Représentation des entiers

La plupart des langages de programmation permettent de distinguer le type entier naturel du type entier relatif. Par exemple, C propose *unsigned int* et *int*. Pourquoi cette distinction ? Certains objets sont intrinsèquement des entiers non signés : une adresse mémoire ou un âge ne peuvent prendre des valeurs négatives.

La capacité de représentation est limitée, comme on l'a vu ci-dessus. Sur un budget de 4 bits, par exemple, on peut représenter soit 16 entiers naturels, soit 16 entiers relatifs, donc 8 positifs et 8 négatifs si on souhaite une repésentation raisonnablement équilibrée. Le codage des naturels et des relatifs est donc différent.

Code	0000	0001	0010	0011	0100	0101	0110	0111
Entier	0	1	2	3	4	5	6	7
Code	1000	1001	1010	1011	1100	1101	1110	1111
Entier	8	9	10	11	12	13	14	15

Table 3.3: Représentation des entiers [0 15] sur un quartet

Entiers naturels

Représentation

Un système de représentation des nombres fondé sur la numération de position utilise une base b, par exemple 10 ou 2, et b symboles qui représentent les nombres entre 0 et $b-1$: en base 2, on n'a donc que deux symboles, 0 et 1.

Définition 6 *La représentation en base 2 est fondée sur l'égalité suivante :*

$$\overline{a_{n-1}a_{n-2}\dots a_0}^2 = \sum_{i=0}^{n-1} a_i 2^i$$

Par exemple, la table 3.3 donne la représentation des entiers naturels de 0 à 15 sur 4 bits.

Proposition 2 *Sur n bits, on peut représenter les entiers naturels N tels que $0 \leq N < 2^n - 1$.*

En effet, le plus grand nombre représentable est

$$\sum_{i=0}^{n-1} 2^i = 2^n - 1.$$

Avec un octet, on peut donc représenter tous les entiers naturels entre 0 et 255. Un mot de 32 bits permet de représenter tous les entiers naturels entre 0 et 4 294 967 295

L'additionneur

L'additionneur d'une UAL est capable d'effectuer l'opération d'addition de deux entiers naturels. Il faut souligner ici que l'additionneur est l'opérateur fondamental de l'UAL : toutes les autres opérations, soustraction de naturels, addition et soustraction de nombres relatifs devront être conçues pour exploiter efficacement l'additionneur naturel. On note $A \bigoplus B$ l'interprétation en entier naturel du résultat de l'addition UAL.

L'addition UAL fournit également une retenue, qui ne fait pas partie du résultat, mais est conservée. Dans la suite, on note C la retenue. On a donc :

Proposition 3 *Si A et B sont des entiers naturels,*

$$A + B = A \bigoplus B + C2^n.$$

Si l'addition de deux naturels produit une retenue, le résultat est faux : le résultat est trop grand pour être représenté. Par exemple, sur 4 bits, l'addition UAL de 0111 et 1100 donne 0011, avec retenue 1. En effet, 7 + 12 = 19, qui ne peut être représenté sur 4 bits.

Entiers relatifs

Une représentation raisonnable des entiers relatifs doit être symétrique, en permettant de représenter autant d'entiers positifs que négatifs. Ceci pose un problème lié au zéro. En effet, il y a un nombre pair de configurations associées à n bits, à répartir entre nombres positifs, nombres négatifs et la valeur 0. La représentation ne peut donc pas être complètement symétrique.

La représentation universelle actuelle est la représentation en complément à 2.

Complément à 2

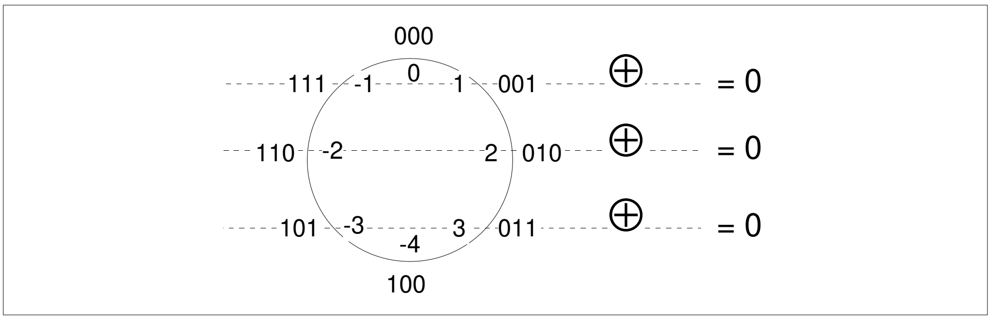

Figure 3.2: Représentation des entiers relatifs sur 3 bits

Un exemple du codage en complément à 2 sur 3 bits est décrit fig. 3.2. On pourra sûrement décrire 0, 1, 2 et 3 et -1, -2, -3. Pour utiliser efficacement l'additionneur, la représentation des ces nombres en tant qu'entiers positifs doit être identique à leur représentation en tant qu'entiers naturels. Pour la même raison, il faut définir la représentation des négatifs de telle sorte que l'addition UAL de N et $-N$ ait pour résultat 0. Il n'y a qu'une solution possible. En fait l'addition UAL donne 0 avec retenue, c'est à dire que l'addition mathématique donne $8 = 2^3$

On remarque que le bit de poids fort des positifs est à 0, et celui des négatifs à 1. Cette caractérisation peut être exploitée facilement en matériel pour tester le signe d'un nombre entier relatif. Il reste une configuration non attribuée : 100. Elle peut être attribuée, soit à 4, soit à -4. Pour rester cohérent avec la caractérisation du signe par le bit de poids fort,elle est associée à -4.

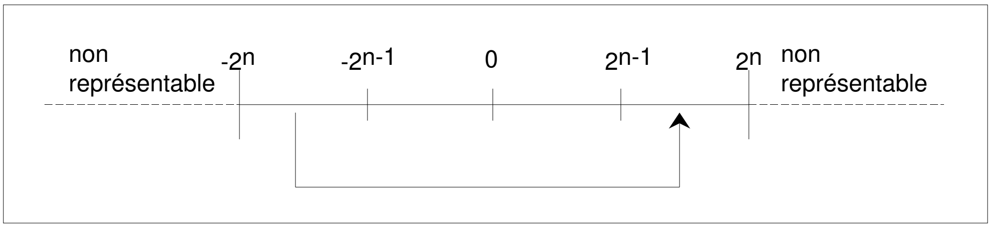

Figure 3.3: Représentation des entiers relatifs sur n bits

Définition 7 *La représentation en complément à 2 sur n bits est définie par :*

1. *les entiers représentés sont* $[-2^{n-1}, 2^{n-1}-1]$*;*
2. *la représentation des entiers positifs est identique à celle des entiers naturels ;*
3. *la représentation d'un entier négatif* $-N$ *est la représentation en naturel de* $2^n - N$

Cette définition est illustrée fig. 3.3. Par exemple, la représentation sur 3 bits de -3est la représentation du naturel $8-3=5$ sur 3 bits, soit 101. La définition est consistante : si $-N$ est négatif et représentable, $-2^{n-1} \leq -N < 0$, donc $2^{n-1} \leq 2^n - N < 2^n$; $2^n - N$ est donc un entier naturel et représentable sur n bits, puiqu'il est plus petit que 2^n.

On a les propriétés suivantes :

Proposition 4 *En complément à 2*

1. *L'addition UAL des représentations d'un entier relatif* N *et de* $-N$ *est égale à 0.*
2. *Le bit de poids fort de la représentation des positifs est 0, le bit de poids fort de la représentation des négatifs est 1.*
3. *La représentation de -1 est le mot où tous les bits sont à 1*

La première propriété vient du fait que l'addition UAL des représentations de N et de $-N$ est celle (en supposant $N \geq 0$) des naturels N et $2^n - N$. Le résultat est donc 0 avec retenue 1.

Soit $N \geq 0$ un positif représentable. $0 < N < 2^{n-1}$, donc $a_{n-1} = 0$. Soit $-N$ un négatif représentable. $0 < N \leq 2^{n-1}$, donc $2^n - N \geq 2^{n-1}$, ce qui prouve la deuxième propriété. □

La définition précédente décrit comment coder les nombres relatifs, ce qui est par exemple la tâche d'un compilateur. L'opération inverse, d'interprétation d'une chaîne de bit vers un nombre relatif, est décrite par la propriété suivante :

Proposition 5 *Si un entier relatif* N *est représenté sur n bits par* $a_{n-1}a_{n-2}\ldots a_0$*, alors*

$$N = -a_{n-1}2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i.$$

Soit $M = -a_{n-1}2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i$. On va montrer que M a pour représentation $a_{n-1}a_{n-2}\dots a_0$. D'abord, M est représentable :

$$(-1)2^{n-1} + \sum_{i=0}^{n-2} 0.2^i \le M \le 0 + \sum_{i=0}^{n-2} 2^i,$$

donc $-2^{n-1} \le M \le 2^{n-1} - 1$. Si $a_{n-1} = 0$, M est représenté par $a_{n-1}a_{n-2}\dots a_0$ d'après la définition 7.2. Si $a_{n-1} = 1$, $M < 0$, donc sa représentation est celle du naturel $2^n + M$.

$$2^n + M = 2^n - 2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i = 2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i,$$

qui se représente comme $1a_{n-2}\dots a_0$.□

Autres représentations

Plusieurs représentations des entiers relatifs ont été utilisées dans le lointain passé de l'informatique, mais la seule représentation utilisée actuellement pour les entiers relatifs est la représentation en complément à 2. En revanche, d'autres représentations sont utilisées indirectement, en particulier pour les exposants des flottants (cf 3.4). Dans toutes les représentations, les nombres positifs sont représentés par leur écriture en base 2, qui est compatible avec l'addition UAL.

Opérations arithmétiques

On a vu que l'UAL effectue l'addition de deux entiers naturels. Le calcul de l'opposé d'un nombre relatif, l'addition et la soustraction de nombres relatifs, doivent être des opérations réalisables facilement avec l'additionneur de l'UAL.

Opposé

Proposition 6 *Si N est un entier dans $[-2^{n-1} + 1, 2^{n-1} - 1]$, la représentation de $-N$ s'obtient en complémentant bit à bit la représentation de N et en ajoutant 1. L'opération de complémentation bit à bit et ajout de 1 est appelée la complémentation à 2.*

Par exemple, sur 4 bits, la représentation de 3 est 0011. La reprsentation de -3 s'obtient en calculant sur 4 bits $1100 \bigoplus 1 = 1101$. En recommençant l'opération, on retrouve 0011.

Supposons $N > 0$ (sinon, on inverse les rôles de N et $-N$). Si N est noté $a_{n-1}a_{n-2}\dots a_0$, alors

$$N = \sum_{i=0}^{n-1} a_i 2^i.$$

Le nombre M obtenu par complémentation à 2 est

$$M = 1 \bigoplus \sum_{i=0}^{n-1} \overline{a}_i 2^i = 1 + \sum_{i=0}^{n-1} \overline{a}_i 2^i,$$

car l'opération ne produit jamais de retenue pour $N > 0$. Donc

$$N + M = 1 + \sum_{i=0}^{n-1} (a_i + \overline{a}_i) = 2^n,$$

donc M est la représentation de $-N$.

Extension de signe

Proposition 7 *Si N est un entier relatif représenté sur n bits par $a_{n-1}a_{n-2}\ldots a_0$, N est représenté sur m ($m > n$) bits par $a_{n-1}\ldots a_{n-1}a_{n-2}\ldots a_0$.*

Cette opération est appelée *extension de signe* : le bit de poids fort a_{n-1} est recopié sur les $m - n$ bits manquants. Par exemple, 7 est représenté par 0111 sur 4 bits, et par 00000111 sur un octet ; -2 est représenté par 1110 sur 4 bits, et par 11111110 sur un octet.

La proposition est évidente si $N \geq 0$. Si $N < 0$, les deux nombres ont le même complément à 2.

Addition et soustraction

La première question est de savoir si l'addition UAL est compatible avec la représentation en complément à 2, *ie* si

$$\text{codage}(A) \bigoplus \text{codage}(B) = \text{codage}(A + B)$$

Ce n'est pas toujours vrai : le nombre de configurations possibles étant limité, le résultat peut être erroné. Par exemple, sur 4 bits, 7 est représenté par 0111, 2 est représenté par 0010, et l'addition donne 1001, qui représente $-8 + 1 = -7$ et non 9. De même l'addition UAL de -7 (1001) et -6 (1010) donne 0011, soit 3 et non -13. Le problème est que le résultat est trop grand en valeur absolue pour être représenté sur 4 bits.

Le problème est donc de déterminer un algorithme simple, qui sera implémenté en matériel, pour détecter l'erreur. On a vu que, pour l'addition des naturels, l'indicateur est le bit de retenue. Pour l'addition des relatifs, la proposition suivante fournit un critère de correction.

Proposition 8 *L'addition UAL de deux relatifs N et M fournit toujours un résultat correct si N et M ne sont pas de même signe. Si N et M sont de même signe, le résultat est correct si le bit de signe est égal à la retenue.*

"Addition dans l'UAL des relatifs N et M" signifie interprétation en relatif du résultat de l'addition UAL des codages de N et M. On veut donc vérifier que

$$\text{codage}(N) \bigoplus \text{codage}(M) = \text{codage}(N + M), \tag{3.1}$$

si et seulement si la condition de la proposition est vraie. Pour pouvoir effectuer des calculs, on choisit une interprétation univoque des deux membres de cette égalité, celle des entiers naturels.

Premier cas : $N \geq 0$ *et* $M < 0$

Alors, $-2^{n-1} \leq N + M < 2^{n-1}$, car $0 \leq N < 2^{n-1}$ et $-2^{n-1} \leq M < 0$ Donc, $N + M$ est représentable. $\text{codage}(N) = N, \text{codage}(M) = 2^n + M$. Donc,

$$\text{codage}(N) \bigoplus \text{codage}(M) = 2^n + N + M - C2^n,$$

où C est la retenue (prop 3).

Il y a retenue si $N + 2^n + M \geq 2^n$, donc si $N + M \geq 0$; dans ce cas, $\text{codage}(N + M) = N + M$ car $N + M \geq 0$ et représentable; $\text{codage}(N) \bigoplus \text{codage}(M) = N + M$ car $C = 1$.

Il n'y a pas retenue si $N + 2^n + M < 2^n$, donc si $N + M < 0$; dans ce cas, $\text{codage}(N + M) = 2^n + N + M$ car $N + M < 0$ et représentable; $\text{codage}(N) \bigoplus \text{codage}(M) = N + M + 2^n$ car $C = 1$.

Deuxième cas : $N \geq 0$ *et* $M \geq 0$

Alors, $0 \leq N + M < 2^n$. Donc, $N + M$ est représentable si $0 \leq N + M < 2^{n-1}$ et non représentable sinon.
$\text{codage}(N) = N, \text{codage}(M) = M$. L'addition UAL ne produit jamais de retenue : les bits de poids fort sont tous deux à 0 ; donc

$$\text{codage}(N) \bigoplus \text{codage}(M) = N + M - C2^n = N + M,$$

puisque $C = 0$. D'autre part, $\text{codage}(N + M) = N + M$ si $N + M$ est représentable, puisque $N + M \geq 0$. Donc, l'égalité (3.1) est vraie si et seulement si $N + M < 2^{n-1}$, donc si le bit poids fort est 0, comme la retenue.

Troisième cas : $N < 0$ *et* $M < 0$

Alors, $-2^n \leq N + M < 0$. Donc, $N + M$ est représentable si $2^{n-1} \leq N + M < 0$ et non représentable sinon.

$\text{codage}(N) = 2^n + N, \text{codage}(M) = 2^n + M$. L'addition UAL produit toujours une retenue : les bits de poids fort sont tous deux à 1 ; donc

$$\text{codage}(N) \bigoplus \text{codage}(M) = 2^n + N + 2^n + M - C2^n = 2^n + N + M,$$

puisque $C = 1$. D'autre part, $\text{codage}(N + M) = 2^n + N + M$ si $N + M$ est représentable, puisque $N + M < 0$. Donc, l'égalité (3.1) est vraie si et seulement si $N + M > -2^{n-1}$, donc si $2^n + N = M > 2^{n-1}$, donc si le bit de poids fort est à 1, comme la retenue. □

Décalages

Un opérateur de l'UAL indépendant de l'additionneur effectue les *décalages logiques*, sll(shift logical left) et slr(shift logical right) et le *décalage arithmétique*, sar(shift arithmetic right).

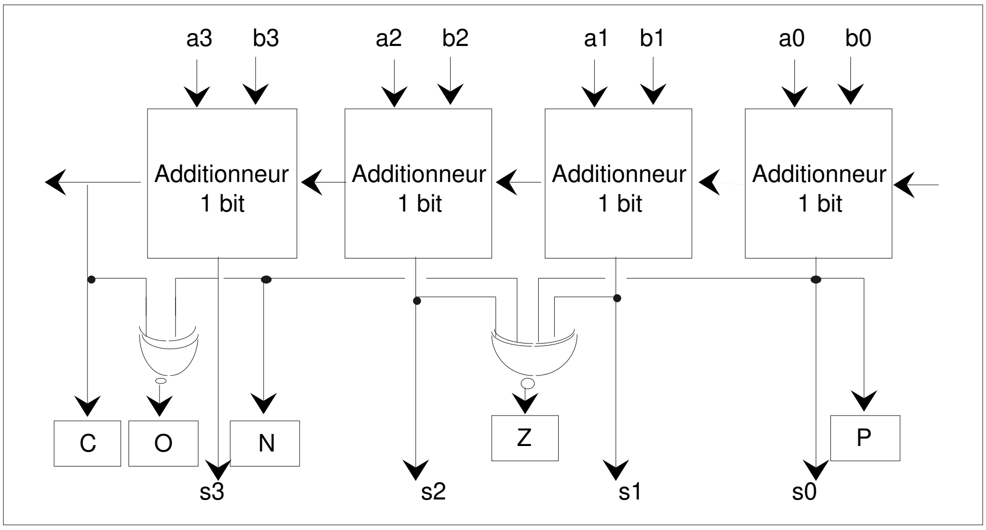

Figure 3.4: Génération des code-conditions

Définition 8

$$sll(a_{n-1}a_{n-2}\dots a_0) = a_{n-2}\dots a_0$$

$$slr(a_{n-1}a_{n-2}\dots a_0) = 0a_{n-1}\dots a_1$$

$$sar(a_{n-1}a_{n-2}\dots a_0) = a_{n-1}a_{n-1}\dots a_1$$

Le décalage arithmétique droit étend le bit de poids fort, le décalage logique droit remplit le bit manquant par un 0.

Proposition 9 *L'interprétation arithmétique des décalages est donné par les propriétés suivantes:*

1. *Si un entier naturel N est représenté par $a_{n-1}a_{n-2}\dots a_0$, $sll(a_{n-1}a_{n-2}\dots a_0) = 2N$ et $slr(a_{n-1}a_{n-2}\dots a_0) = N/2$, le quotient étant le quotient entier.*

2. *Si un entier relatif N est représenté par $a_{n-1}a_{n-2}\dots a_0$, $sll(a_{n-1}a_{n-2}\dots a_0) = 2N$ et $sar(a_{n-1}a_{n-2}\dots a_0) = N/2$, le quotient étant le quotient entier.*

Par exemple, sur 4 bits, 7 est représenté par 0111 ; sll(0111) = 1110, qui représente 14 ; slr(0111) = 0011, qui représente 3. -3 est représenté par 1101 ; sll(1101) = 1010, qui représente -6 ; sar(1101) = 1110, qui représente -2 (rappel : le quotient entier de a par b est le plus grand entier inférieur ou égal au réel a/b, donc $(-3)/2 = -2$, et non -1.

Traitement des erreurs

Une opération UAL génère, outre le résultat sur n bits, plusieurs informations que le code peut vouloir tester par la suite (fig. 3.4). Ces informations sont : C : retenue (carry)
P : parité, résultat pair ou impair
Z : résultat 0
N : résultat négatif
0 : dépassement de capacité (overflow)

C correspond à un dépassement de capacité quand l'opération est interprétée dans le champ des entiers naturels. 0 correspond à un dépassement de capacité quand l'opération est interprétée dans le champ des entiers relatifs.

Ces informations sont appelées *Code-conditions*, ou *drapeaux* (flags). Elles sont souvent conservées dans un registre d'état du processeur, le *registre code-conditions* RCC, et accessibles par des instructions conditionnelles, par exemple BE = brancher si drapeau Z positionné. D'autres architectures (MIPS, Alpha, IA64) conservent ce résultat dans un registre adressable. Ce problème sera étudié plus en détail au chapitre suivant.

Que se passe-t-il lors d'un dépassement de capacité ? Pour beaucoup d'architectures (x86, SPARC) , rien d'autre que le positionnement du code-condition associé. D'autres (MIPS) peuvent déclencher une *exception* (trap), qui déclenche elle-même l'exécution d'un *gestionnaire d'exception* (trap handler) c'est à dire l'exécution d'une routine associée à l'évènement. Le processeur fournit un support matériel en lançant l'exécution de la routine sans intervention de l'application. Le logiciel de base fournit le contenu de la routine. Dans ce cas précis, la routine va en général signaler l'erreur et arrêter l'exécution du programme. Le comportement de ces architecture est donc très différent, par exemple sur le calcul de 100! : les premières fournissent un résultat faux, 100! modulo 2^n ; les secondes signalent l'erreur. Certains langages, comme ADA, requièrent la détection de toutes les erreurs arithmétiques. Sur les architectures qui ne déclenchent pas d'exception, le compilateur doit insérer un code de test de débordement après chaque opération arithmétique susceptible d'en produire.

L'exemple le plus célèbre d'erreur arithmétique catastrophique est la perte de la fusée Ariane 5 après quarante secondes de vol en 96 [9]. La défaillance était due à un comportement aberrant du système de guidage de la fusée. Ce système était forme d'un système de référence inertiel (SRI) alimenté par des capteurs et fournissant des informations à un calculateur embarqué (ORB, on board computer) qui commandait les mouvements de rotation de la fusée. L'ensemble SRI plus ORB était répliqué en deux exemplaires pour assurer la redondance. A environ 37 secondes, les SRI1 et SRI2 ont envoyé un message d'erreur aux OBC1 et 2 ; l'arrivée d'un tel message n'était pas prévue dans le code exécuté par les OBC, qui l'ont interprété comme des coordonnées. La fusée a tangué brutalement, puis a commencé à se briser. Le dispositif d'autodestruction, destiné à éviter que la fusée retombe dangereusement sur la terre, a détecté la rupture fusée-boosters, et détruit la fusée en vol.

Le message d'erreur est intervenu lorsque les systèmes de réfénces inertiels effectuaient

Chiffre décimal	code
0	0 0 0 0
1	0 0 0 1
2	0 0 1 0
3	0 0 1 1
4	0 1 0 0
5	0 1 0 1
6	0 1 1 0
7	0 1 1 1
8	1 0 0 0
9	1 0 0 1

Table 3.4: Décimal codé binaire

la conversion d'un flottant x vers un entier 16 bits, avec arrondi évidemment (par exemple 10,3 $\rightarrow$ 10). La valeur de x était trop grande pour tenir sur un entier 16 bits, et le code ADA a signalé le dépassement de capacité.... Or, ce fragment de code de conversion, hérité d'Ariane 4, était inutile pour Ariane 5 ; dans Ariane 4, la valeur de l'entier était représentable sur 16 bits. Cette erreur élémentaire a coûté plus de 4 milliards de francs...

Représentation BCD

Certaines applications, notamment en gestion, exigent des calculs décimaux exacts, sans arrondi, ce qui implique de travailler avec des nombres décimaux. En effet, avec un nombre fixé de bits, il est impossible de convertir de manière exacte des nombres binaires en nombres décimaux et réciproquement. On utilise alors la représentation *décimale codée binaire* (binary coded decimal, BCD), dans laquelle chaque chiffre décimal (0 - 9) est codé avec 4 chiffres binaires, en codage naturel 3.4. On distingue le BCD non compacté, avec 1 octet par chiffre, et le BCD compacté, avec 1 octet pour 2 chiffres.

Le BCD n'est pas compatible avec l'addition UAL :

$$\mathrm{codage}_{\mathrm{BCD}}(1) \bigoplus \mathrm{codage}_{\mathrm{BCD}}(2) = \mathrm{codage}_{\mathrm{BCD}}(3)$$

car $0001 \bigoplus 1010 = 0011$, mais

$$\mathrm{codage}_{\mathrm{BCD}}(6) \bigoplus \mathrm{codage}_{\mathrm{BCD}}(8) = 1110$$

, qui n'est pas un code BCD.

3.4 Nombres réels

La représentation et le traitement des nombres réels sont appelés représentation et traitement flottants. L'explication est que la virgule n'a pas une position fixe, comme on le verra par la suite.

Représentation des réels

Mantisse et exposant

La représentation flottante est basée sur la représentation d'un nombre sous forme mantisse et exposant :

$$x = \pm m \times B^e$$

Par exemple, en notation décimale, $2,76$ peut s'écrire $2,76 \times 10^0$, et aussi 276×10^{-2}.

Dans la suite, certains exemples seront présentés en décimal (m, e en base 10, $B = 10$) pour alléger les notations. En machine, tout est en base 2.

Un nombre décimal a une infinité de représentations mantisse-exposant, par exemple

$$2,76 = 0,276 \times 10^1 = 27,6 \times 10^{-1} etc.$$

La représentation normalisée se définit par l'existence d'un seul chiffre, qui ne doit pas être 0, avant la virgule. Dans l'exemple précédent, cette représentation est donc $2,76 \times 10^0$.

En base 2, le chiffre avant la virgule est nécéssairement 1. Il n'est donc pas utile de gaspiller un bit pour l'indiquer. La mantisse comportera donc souvent un 1 implicite.

Difficultés de la représentation des réels

La représentation des nombres réels pose un problème plus difficile que celle des nombres entiers : les nombres réels sont continus. L'ordinateur ne travaillant que sur un nombre limité de digits, il doit *arrondir* les représentations. Il ne peut pas non plus représenter des réels abitrairement grands ou petits en valeur absolue.

Tous les nombres rationnels ne peuvent pas s'écrire sous forme mantisse-exposant. Par exemple en base 10 et en base 2, 1/3. Les nombres rationnels qui s'écrivent sous cette forme en base 10 sont les nombres décimaux, mais le terme équivalent en base 2, deuxcimaux, n'est guère usité. Les nombres représentables en machine sont donc un sous-ensemble des nombres deucimaux.

La représentation des réels, les critères de précision (arrondi) requis des processeurs, et le traitement des dépassements de capacité (nombres trop grands ou trop petits) ont été un sujet de longues polémique parmi les constructeurs de machines et les utilisateurs du calcul scientifique : le standard actuel, la norme IEEE 754, fut mis en chantier en 1977, mais son adoption définitive date de 1985.

La norme IEEE 754 comporte deux aspects : la définition d'une représentation commune des réels, et des contraintes sur la précision des calculs.

On mesure mieux la difficulté du problème si l'on sait que Von Neumann était opposé à l'idée même de l'arithmétique flottante en matériel. Cette opposition était fondée sur la complexité du matériel requis, et sur l'encombrement mémoire supplémentaire.

Simple précision

1	8	23
s	E (exposant)	f (mantisse)

Double précision

1	11	52
s	E (exposant)	f (mantisse)

Table 3.5: Formats IEEE 754 simple et double précision.

Représentation IEEE 754

Il existe quatre formats : simple et double précision, et simple et double précision étendue. Nous ne présentons que simple et double précision. La fig. 3.5 présente les formats correspondants.

En simple précision e est l'interprétation de E en excès à 128 :

e = interprétation de E en naturel - 127.

Donc, en simple précision, $e_{\text{min}} = 0 - 127 = -127, e_{\text{max}} = 255 - 127 = 128$.

En double précision, e est l'interprétation de E en excès à 1023 : $e_{\text{min}} = -1023, e_{\text{max}} = 1024$

Cas normalisé

Pour $e_{\text{min}} < e < e_{\text{max}}$, le codage s'interprète de la façon suivante :

- le bit de plus fort poids donne le signe du nombre ;
- la mantisse utilise un 1 implicite ; donc, pour une partie fractionnaire $f_1 f_2 \dots f_n$, la mantisse m est définie par :

$$m = 1, f_1 f_2 \dots f_n = 1 + \sum_{i=1}^{n} f_i 2^{-i} = \overline{1 f_1 f_2 \dots f_n}^2 2^{-n};$$

- au total, le nombre s'évalue par :

$$x = (-1)^s \times 1, f_1 f_2 \dots f_n \times 2^e.$$

Par exemple, C8900000 code $-2^{18}(1 + 2^{-3})$. En effet, on a :

bit de signe 1
$E = 10010001 = 145$, donc $e = 17$
$f = 0010\dots0$, donc $m = \overline{1,001}^2 = 1 + 2^{-3}$

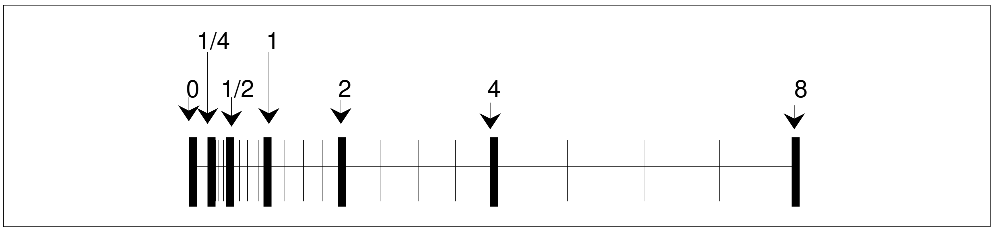

Figure 3.5: Echelle de représentation des flottants

Nom	e	f	valeur
Normalisé	$e_{\text{min}} < e < e_{\text{max}}$	quelconque	$(-1)^s \times 1, f \times 2^e$
Dénormalisé	$e = e_{\text{min}}$	$\neq 0$	$(-1)^s \times 0, f \times 2^{e_{\text{min}}}$
Zéro	$e = e_{\text{min}}$	0	$(-1)^s \times 0$
Infini	$e = e_{\text{max}}$	0	$(-1)^s \times \infty$
NaN	$e = e_{\text{max}}$	$\neq 0$	NaN

Table 3.6: Interprétation des champs dans le format IEEE 754

La représentation normalisée permet de représenter les nombres de façon équilibrée. Considérons un exemple plus simple, avec 2 bits de mantisse et 3 bits d'exposant (ce qui n'est pas un choix raisonnable !). L'exposant est donc interprété en excès à 3, et $-2 \leq e \leq 3$. La figure 3.5 décrit les nombres positifs représentés. Entre ces nombres, on a les réels non deucimaux de l'intervalle, qui devront être approximés. Les barres épaisses correspondent aux cas où $m = 1$, donc $x = 2^{-2}, 2^{-1}, 2^0, 2^1, 2^2, 2^3$. On voit que dans chaque intervalle, l'espacement est double de celui du précédent. La différence entre deux nombres représentés consécutifs n'est pas constante, mais l'erreur relative créée par un arrondi est constante.

Cas exceptionnels

La table 3.6 montre que les valeurs extrêmes sont réservées pour une représentation synthétique des nombres non représentables en normalisé.

Il existe un plus grand nombre exactement représentable en normalisé :

$$x_m = 1, 1 \ldots 1 \times 2^{e_{\text{max}}-1}.$$

Les nombres plus grands sont représentés par les mots où le signe est positif, $e = e_{\text{max}}$ et $f \neq 0$. L'interprétation de tous ces mots est identique : $+\infty$. Pour les nombres négatifs, on a une représentation analogue de $-\infty$.

Il existe un plus petit nombre strictement positif représentable en normalisé. Cependant, la représentation dénormalisée permet de représenter des nombres plus petits. On notera que le bit implicite n'est plus 1, mais 0, d'où l'appellation dénormalisé.

Il existe deux représentations de 0, suivant le bit de signe.

Enfin, un code est réservé pour représenter le résultat d'une opération aberrante, par exemple 0/0. Ce code est noté NaN.

L'objectif de cette représentation est décrit en 3.4.

Opérations flottantes

Les opérations flottantes impliquent un traitement simultané des mantisses (c'est à dire des parties fractionnaires) et des exposants. Les principales opérations sont les comparaisons et les opérations arithmétiques : addition, soustraction, multiplication, division. Dans tous les microprocesseurs généralistes actuels, les opérations d'addition, soustraction, multiplication et division sont réalisés par des opérateurs matériels spécifiques, intégrés dans le CPU. On trouvera une présentation de ces opérateurs dans [2, 10]. L'ensemble de ces opérateurs est applelée *Floating Point Unit* (FPU). En revanche, les opérations plus complexes, racine carrée, fonctions trigonométriques, sont réalisés par logiciel.

Le standard IEEE 754 décrit dans cette section ne prescrit rien sur l'implémentation. Il spécifie simplement les bits d'une représentation, et du résultat des opération arithmétiques flottante. Ainsi, le langage Java offre un calcul flottant conforme IEEE 754 indépendamment de toute plate-forme matérielle.

Comparaisons

L'utilisation du bit de signe permet le test rapide du signe. La notation en excès pour l'exposant permet de comparer les flottants en utilisant la comparaison de entiers naturels.

Addition et soustraction

L'addition implique une dénormalisation du nombre le plus petit pour que les exposants deviennent égaux, suivie d'une addition des mantisses, qui est suivie éventuellement d'une renormalisation.

La dénormalisation et la renormalisation peuvent entraîner une perte d'information. L'addition peut entraîner la sortie du champ des nombres représentable en normalisé : par exemple $x_m + x_m$.

Multiplication

Soit deux nombres flottants $x_1 = s_1 m_1 2^{e_1}$ et $x_2 = s_2 m_2 2^{e_2}$. Le produit $x_1 x_2$ est donné par $s_1 s_2 m_1 m_2 2^{e_1+e_2}$.

Il y a multiplication des mantisses, ce qui correspond à une multiplication entière, où l'on arrondit pour obtenir un résultat correspondant au nombre de bits de la partie fractionnaire. Compte tenu du codage en excès, l'addition des exposants correspond à l'opération $E_1 + E_2 - c$, où les E_i sont interprétés comme des entiers naturels, $c = 127$

en simple précision et 1023 en double précision. Là encore, il peut il y avoir perte d'information, lors de l'arrondi, ou sortie du champ des nombres représentables, lorsque l'exposant est trop grand ou trop petit.

Traitement des situations anormales

Comme on vient de le voir, les calculs peuvent entraîner soit un dépassement de capacité, le résultat n'est pas représentable en format normalisé, ou erreur, le résultat est arrondi. La norme IEEE 754 vise à permettre à un programme d'adopter un comportement adéquat dans ce type de situation anormale. Cette section en présente les aspects élémentaires. On trouvera un traitement exhaustif dans [11].

Les résultats non représentables sont pris en compte par l'arithmétique étendue et la représentation dénormalisée. Les erreurs sont prises en compte par une contrainte de précision.

Arithmétique étendue

L'idée de l'arithmétique étendue est qu'il peut être profitable de laisser survivre un programme qui a, par exemple, effectué une division par 0. L'exemple plus simple est un solveur qui cherche les zeros d'une fonction dont l'ensemble de définition n'est pas commodément calculable. le solveur travaille en évaluant la fonction en divers points. S'il tombe sur une valeur hors de l'ensemble de définition de la fonction, il peut fort bien calculer $0/0$ ou $\sqrt{-1}$. Cette erreur n'est pas nécessairement gênante, si le solveur peut recommencer en un autre point indépendamment.

La norme définit d'abord une arithmétique étendue à trois valeurs supplémentaires, $\pm\infty$ et *NaN*. Ce sont les règles "usuelles", par exemple :

$(+\infty) + (+\infty) = (+\infty)$

$(+\infty) + (-\infty) = NaN$

$(\pm\infty) \times (\pm\infty) = (\pm\infty)$ avec règle des signes.

Toute opération dont un des opérandes est *NaN* a pour résultat *NaN*.

Du point de vue de l'application, si une opération produit un résultat trop grand en valeur absolue, ou bien est mathématiquement incorrecte ($0/0$), le résultat tombe dans l'ensemble $\{+\infty, -\infty, NaN\}$, et les calculs peuvent se poursuivre en utilisant l'arithmétique étendue.

Tout FPU conforme IEEE 754 doit implanter cette arithmétique étendue.

Représentation dénormalisée

La différence de deux nombres normalisés peut ne pas être représentable en format normalisé. Par exemple, en simple précision ($e_{\text{min}} = -127$), $x = 1,1\ldots11 \times 2^{-126}$ et $y = 1,1\ldots10 \times 2^{-126}$ sont représentables en normalisé (par 00FFFFFF et 00FFFFFE). Mais $x - y = 0,0\ldots01 \times 2^{-126}$ n'est pas représentable en normalisé.

On pourrait tout simplement arrondir le résultat, au plus proche représentable, soit 0. Mais supposons que le FPU réalise correctement la comparaison de deux flottants, en testant l'égalité de tous leurs bits. Le test $x = y$ donne faux, et le test $x - y = 0$ donne vrai. Deux codes identiques du point de vue mathématique auront des résultats différents : le code if not (x=y) then z = 1/(x-y) pourrait aboutir à une division par 0, et le code if not (x-y = 0) then z = 1/(x-y) ne produit pas d'erreur.

La représentation dénormalisée permet précisément la représentation de ces nombres trop petits : $0,0\ldots 01 \times 2^{-126} = 0,0\ldots 10 \times 2^{-127}$, qui se représente comme 00000002. Dans ce cas, les deux tests ont le même résultat.

La différence de deux nombres dénormalisée peut elle-même être trop petite pour être représentée, même en dénormalisé, et donc être arrondie à 0. On verra plus loin le traitement de ce type d'évènement.

Drapeaux et gestionnaires d'exceptions

Le standard IEEE 754 fournit un fonctionnement par défaut, qui est de continuer le calcul dans l'arithmétique étendue. Continuer l'exécution est souvent la solution appropriée, mais pas toujours. Sur erreur arithmétique flottante, une application peut donc souhaiter trois comportements : arrêt immédiat, ou contrôle par elle-même ou non-traitement.

Un exemple typique d'application qui demande un arrêt immédiat est celui du calcul de $\frac{x}{1+x^2}$. En simple précision, quand $x = 2^{65}$ (représentable par 60000000), x^2 produit ∞ et le résultat est 0, alors qu'il est de l'ordre de $1/x$, qui est parfaitement représentable ; le résultat est donc complètement erroné et continuer l'exécution en général sans intérêt (et éventuellement coûteux en temps machine). Un exemple typique d'application qui demande un traitement nuancé est une application qui génère des nombres dénormalisés : elle ne souhaite pas s'interrompre prématurément à chaque résultat dénormalisé, mais elle veut s'interrompre si elle obtient un 0 comme résultat de la soustraction de nombres dénormalisés.

Le standard prescrit que les évènements anormaux soient enregistrés, dans des drapeaux (flag), et recommande fortement que des gestionnaires d'exceptions soient installés. Les drapeaux permettent un traitement personnel à l'utilisateur (éventuellement rien) ; les gestionnaires d'exception fournissent un traitement par le logiciel de base ("système"), qui permet en particulier l'arrêt immédiat.

La figure 3.7 présente ces drapeaux. Des instructions du processeur permettent de les tester. Finalement, des routines des bibliothèques numériques offrent une interface utilisateur vers ces instructions, par exemple la *libm*, qui est la librairie numérique standard associée au langage C. L'utilisateur peut donc choisir de ne pas tester les drapeaux, ou de les tester et d'appliquer un algorithme de son choix. La norme prescrit que les drapeaux soient persistants (sticky) : à la différence des drapeaux entiers, un drapeau positionné ne sera remis à à que par une instruction explicite. Les drapeaux et les librairies numériques permettent donc le contrôle par l'application.

Un traitement générique, en général l'arrêt immédiat est rendu possible si chaque

Flag	Condition	Resultat
Underflow	Nombre trop petit	0, $\pm e_{\mathrm{min}}$ ou nombre denormalise
Overflow	Nombre trop grand	$\pm\infty$ ou x_{max}
Division par 0	division par 0	$\pm\infty$
Invalid Operation	Le resultat est NaN et les opérandes $\neq$ NaN	NaN
Inexact	Resultat arrondi	Resultat arrondi

Table 3.7: Evènements anormaux dans la norme IEEE 754. la première colonne décrit le drapeau, la seconde l'opération qui crée cet évènement, la troisième le résultat dans le comportement par défaut.

dépassement de capacité flottant peut déclencher une exception. Le drapeau n'est alors pas positionné.

Le choix entre positionnement des drapeaux (avec poursuite de l'exécution du programme utilisateur) et exception est programmable. Les microprocesseurs offrent des bits de contrôle, qui font généralement partie d'un mot de contrôle du processeur, et des instructions pour les positionner, qui permet ce choix. Il y a un bit de contrôle par exception. Lorsque l'erreur ne produit pasl'appel d'un gestionnaire d'exception, on dit que l'exception est *masquée*. Par exemple, l'exception *Résultat Inexact* est presque toujours masquée : la plupart des calculs effectuent des arrondis et le programme doit continuer. L'interface d'un langage de haut niveau vers les instructions de masquage est réalisé par des options du compilateur.

Précision

Les opérations d'alignement et d'arrondi perdent de l'information, ce qui peut être sans effet ou catastrophique.

Par exemple, sur 3 digits décimaux de mantisse, soient $x = 2,15 \times 10^{12}$ et $y = 1,25 \times 10^{-5}$. En alignant y sur x et en arrondissant, $y = 0$ et $x \ominus y = 2,15 \times 10^{12}$, ce qui est le résultat arrondi correct ($\ominus$ est l'opérateur de soustraction sur 3 digits).

Mais, pour $x = 1,01 \times 10^1$ et $y = 9,93$, en alignant y sur x et en arrondissant, $y = 0,99 \times 10^1$ et $x \ominus y = 0,02$, alors que le résultat exact sur 3 digits est $10,1 - 9,93 = 0,17$. Deux digits sont donc faux : l'erreur sur le dernier digit s'est propagée.

Une mesure de la précision est le *ulp* (units in last postion). Dans le calcul précédent, l'erreur est 15 ulp.

Le standard IEEE impose que les opérations d'addition, soustraction, multiplication et division soient *arrondies exactement* : tout se passe comme si le résulat était calculé exactement, puis arrondi au plus près. Ceci requiert que l'unité de calcul dispose de plus de bits que le format, pour stocker temporairement des informations. On a montré que 3 bits suffisent (garde, garde supplémentaire et sticky bit).

Le contrat assuré par le standard IEEE ne garantit évidemment rien sur le cumul des erreurs dans une séquence de calculs. L'étude de la la qualité numérique des résultats d'un algorithme est une branche de l'analyse numérique.

L'exemple le plus simple est l'associativité, que le calcul flottant ne respecte pas. Considérons l'exemple suivant, toujours avec une mantisse 3 digits décimaux :

$-2,15 \times 10^{12} \oplus (2,15 \times 10^{12} \oplus 1,25 \times 10^5) = -2,15 \times 10^{12} \oplus 2,15 \times 10^{12} = 0$

$(-2,15 \times 10^{12} \oplus 2,15 \times 10^{12}) \oplus 1,25 \times 10^5 = 1,25 \times 10^5$

3.5 Placement mémoire

Big Endian et Little Endian

La mémoire est en général organisée par octets, c'est à dire que la plus petite unité adressable est l'octet. Ainsi, la déclaration :

char c = 'A';

correspond à la réservation d'un octet, dont l'adresse est par définition celle de la variable c, et qui est initialisé à 0x41 (réservation signifie simplement que le compilateur n'allouera pas cet octet pour une autre variable ; on verra dans le chapitre suivant comment s'effectue cette réservation). On notera dans la suite @x l'adresse mémoire d'une variable x

Comme la mémoire est organisée par octets, les types représentés sur plus d'un octet est plus difficile à gérer ; par exemple, la déclaration :

int x = 0x12345678 ;

demande que l'ensemble d'octets {12, 34, 56, 78} soit placé en mémoire dans l'ensemble d'adresses {@x, 1+@x,2+@x, 3+@x}, mais dans quel ordre ? En fait, deux solutions existent (fig. 3.6).

- *Big Endian* : l'octet de poids fort est à l'adresse @x ; on commence par le gros bout.
- *Little Endian* : l'octet de poids faible est à l'adresse @x ; on commence par le petit bout.

Initialement, l'un des formats était caractéristique des architectures x86 (Intel), et l'autre des architectures 68x00 (Motorola). Actuellement, les processeurs peuvent être configurés pour supporter les deux formats. Les noms Big Endian et Little Endian sont une allusion littéraire. Dans les années 80, les news ont été encombrées par d'interminables polémiques portant sur les mérites respectifs des deux formats, pour lesquels aucun argument décisif n'existe. Le nom fait allusion à un épisode des voyages de Gulliver, où une guerre est déclarée entre les partisans du gros bout et du petit bout pour entamer les œufs à la coque.

Adresse	Big Endian	Little Endian
@x	12	78
1+@x	34	56
2+@x	56	34
3+@x	78	12

Figure 3.6: Plan mémoire en Big Endian et Little Endian

Alignement

Les deux déclarations suivantes sont sémantiquement équivalentes.

(a)	(b)
int x = 0x12345678; char c = 'A'	char c = 'A'; int x = 0x12345678;

Cependant, elles produisent deux occupations mémoire différentes (fig. 3.7, en Big Endian, en supposant que les variables sont implantées à partir de l'adresse 0x1000). Dans les deux cas, la variable x, qui occupe quatre octets, est implantée à une adresse multiple de 4 ; dans le deuxième cas, le compilateur est contraint de laisser inoccupés quelques octets pour respecter cette contrainte. Plus généralement,

Définition 9 *On dit qu'une donnée de taille p mots mémoire est alignée si son adresse est multiple de p.*

Adresse	(a)	(b)
1000	12	41
1001	34	-
1002	56	-
1003	78	-
1004	41	12
1005	-	34
1006	-	56
1007	-	78

Figure 3.7: Alignement mémoire

L'alignement découle de l'architecture matérielle : les accès aux circuits qui sont le support matériel de la mémoire sont organisés pour permettre un accès rapide aux ensembles d'octets alignés. Le surcoût en encombrement mémoire est compensé par la rapidité. La plupart des processeurs, à l'exception des architectures Intel, n'autorisent pas l'accès non aligné, le surcoût en encombrement mémoire étant compensé par la rapidité. Un accès non aligné provoque alors une exception, qui conduit en général à la fin prématurée du programme. En particulier, une erreur d'exécution interviendra si un pointeur sur un type multi-octets (par exemple sur un entier), se retrouve pour une raison quelconque contenir une adresse non alignée. Lorsque l'accès non aligné est permis, il est souvent plus coûteux en temps qu'un accès aligné. Les compilateurs actuels créent donc toujours un code aligné.

Types

Les LHN définissent des types, qui permettent un calcul sur le type des expressions, donc sur le résultat des affectations (par exemple entier + flottant → flottant). Le type d'une variable définit aussi son encombrement mémoire. On ne considèrera ici que les structures de données les plus courantes, scalaires et tableaux.

Types scalaires

On a vu ci-dessus la représentation des *types scalaires*.

Tableaux

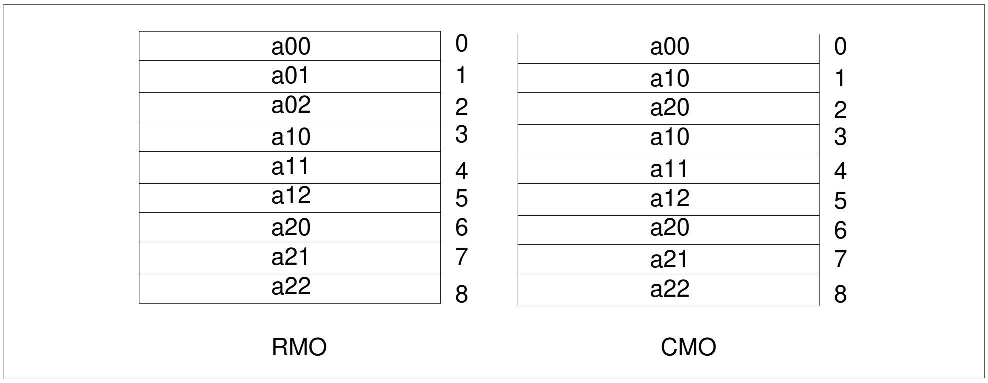

Figure 3.8: Allocation des tableaux en mémoire

Les *tableaux* sont réalisés en mémoire par une séquence implantée à partir de l'adresse du premier élément du tableau. Pour un tableau unidimensionnel A, l'adresse de $A[i]$ est $@A + i \times s_a$, où s_a est la taille en octets d'un élément de a.

Les tableaux multidimensionnels sont linéarisés par le compilateur (fig. 3.8). Les compilateurs C et Pascal utilisent l'ordre *ligne d'abord* (Row Major Order, RMO) ; par exemple, pour un tableau bidimensionnel, les éléments d'une ligne sont alloués consécutivement en mémoire ; les compilateurs Fortran utilisent l'ordre *colonne d'abord* (Colum Major Order, CMO) où les éléments d'une colonne sont alloués consécutivement en mémoire.

Chapitre 4

Architecture logicielle du processeur

L'architecture logicielle est la *spécification* d'un processeur. Elle décrit les unités fonctionnelles (UAL, Unités de calcul flottant etc.), les registres visibles, les types de données manipulés et les opérations le processeur effectue sur ces données. Cette spécification est décrite par le *jeu d'instructions*. C'est la machine que voit un compilateur, ou un programmeur en langage machine.

Le but de l'architecture logicielle d'un processeur généraliste est de permettre une exécution efficace des constructions des langages de haut niveau (LHN). Ces constructions ont une sémantique riche, par exemple une boucle, ou un appel de fonction.

L'introduction a souligné que la prise en compte de l'ensemble des contraintes technologique a des conséquences majeures à tous les niveaux de l'architecture, matérielle et logicielle. Dans ce chapitre, nous ne considèrerons que deux contraintes :

- le nombre des opérateurs matériels est limité;
- les calculs ne peuvent s'effectuer que sur des données présentes dans le processeur.

Ce chapitre présente donc une typologie des composants des architecture logicielles, par rapport aux structures des LHN impératifs. L'objectif n'est ni de présenter de manière exhaustive le jeu d'instructions d'une machine (plus de 200 pour le PowerPC !), ni de présenter toutes les variantes de jeux d'instructions que l'on peut trouver dans les architectures actuelles, mais d'étudier les caractéristiques des jeux d'instruction qui permettent l'exécution des langages de haut niveau. Les chapitres suivants discuteront de façon plus approfondie l'interaction avec les contraintes technologiques.

4.1 Modèle d'exécution

Définition

Le choix du nombre d'opérandes sur lesquels travaille une instruction machine est une question clé. Il résulte d'un compromis entre le temps de calcul, le nombre d'opérateurs matériels utilisés et le nombre de registres disponibles.

Soit par exemple l'instruction Y := A + B + C + D. Elle peut être exécutée en une seule instruction si l'on dispose de trois opérateurs matériels d'addition (*additionneurs*), ou en trois instructions avec un seul additionneur. La fréquence d'une telle instruction dans les codes LHN ne nécessite certainement pas le coût de trois additionneurs, alors que l'instruction suivante sera typique des langages de haut niveau :

$$\text{Resultat} = \text{operande}_1 \text{ OP } \text{operande}_2.$$

Cette instruction correspond à ce que peut réaliser une UAL qui effectue une opération sur deux opérandes et délivre un résultat.

Trois opérandes spécifiés par instruction (deux opérandes source et un résultat) constituent donc un bon compromis entre efficacité et coût du matériel. Cependant, ce compromis suppose qu'on peut effectivement accéder à trois opérandes, ce que le nombre de registres disponible n'a pas toujours permis. La première caractéristique du modèle d'exécution est donc le nombre d'opérandes.

Les opérandes considérés peuvent être situés, soit dans un registre interne à l'unité centrale, soit dans la mémoire. Pour chaque opérande source ou résultat, l'instruction doit contenir le numéro de registre pour les opérandes dans l'UC, ou l'ensemble des informations pour calculer l'adresse mémoire de l'opérande.

Définition 10 *Soit n le nombre total d'opérandes spécifiés par instruction, et m le nombre d'opérandes mémoire. Le couple (n, m), avec $m \leq n$, définit le modèle d'exécution du processeur.*

Les différents modèles d'exécution

La figure 4.1 illustre les différents modèles d'exécution des instructions, sur l'exemple A = B + C.

La partie (a) présente le modèle mémoire-mémoire (3,3), qui est caractéristique du VAX-11 de Digital. Une instruction peut spécifier les deux opérandes source et le résultat, et chaque opérandes peut être situé en mémoire. L'instruction A = B + C est exécutée par une instruction assembleur : ADD @A, @B, @C.

La partie (b) présente le modèle mémoire-accumulateur (1,1), typique des premiers microprocesseurs 8 bits (Intel 8080 ou Motorola 6800). L'instruction ne spécifie qu'un opérande ; les autres ssont implicites, contenus dans le registre *accumulateur*. L'instruction A = B + C est alors exécutée par une suite de trois instructions machine :

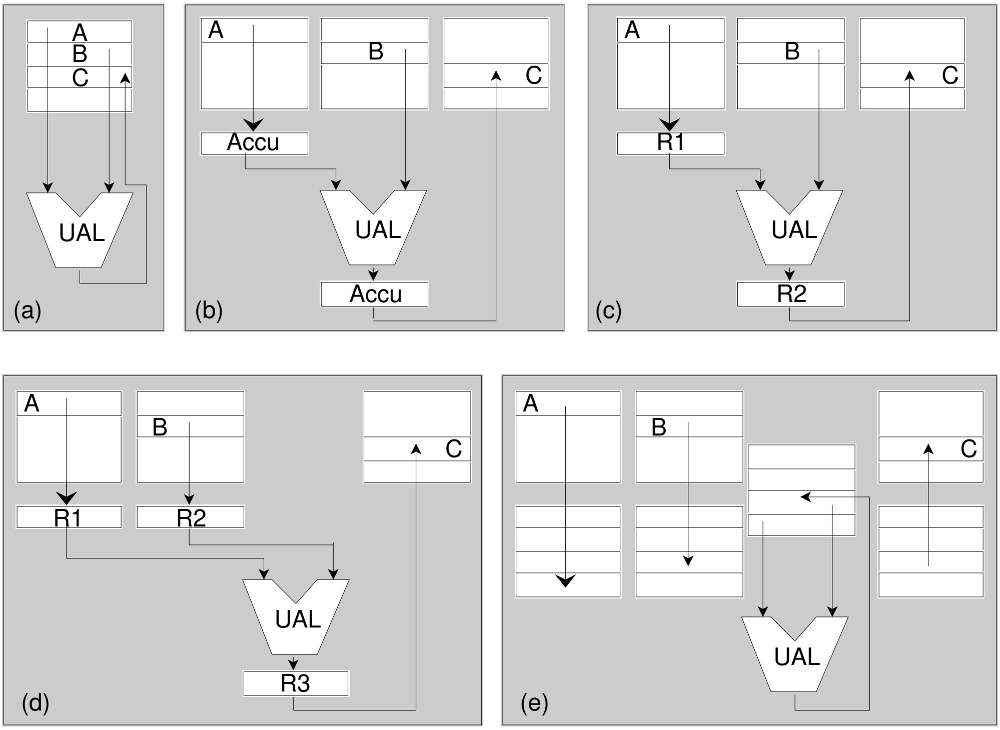

Figure 4.1: Modèles d'exécution

```
LD @B
ADD @C
ST @A
```

chargement du premier opérande dans l'accumulateur, addition du deuxième opérande avec l'accumulateur dans lequel est rangé le résultat, et rangement de l'accumulateur en mémoire.

La partie (c) présente le modèle mémoire-registre (2,1), où l'accumulateur unique est remplacé par un ensemble de registres généraux. Il est caractéristique des microprocesseurs 16 bits (Intel 8086 ou Motorola 68000). Dans les processeurs avec modèle (2,1), il existera aussi la variante (2,0) où les deux opérandes sources sont dans des registres, et le résultat dans un des deux registres source.

```
LD R1, @B
ADD R2, R1, @C
ST R2, @A
```

La partie (d) présente le modèle registre-registre (3,0), où les trois opérandes des opérations UAL sont obligatoirement en registre (0 opérande mémoire). Seules les instructions de chargement (LD) et rangement (ST) accèdent à la mémoire pour charger ou ranger le contenu d'un registre. L'instruction A = B + C est exécutée par une suite de quatre instructions machine : chargement du premier opérande dans un premier registre, chargement du deuxième opérande dans un deuxième registre, addition sur registres, et rangement du résultat en mémoire.

```
LD R1, @B
LD R2, @C
ADD R3, R1, R2
ST R3, @A
```

Les architectures avec modèle (3,0) ont donc été appelées architecture *chargement-rangement* ou encore *registre-registre*.

La partie (e) présente le modèle pile (0,0) pour lequel les opérandes sont successivement empilés en sommet de pile. L'instruction ADD ne spécifie aucun opérande, puisqu'elle prend implicitement ses opérandes d'entrée dans la pile et range le résultat dans la pile. Le résultat est ensuite dépilé.

```
PUSH @B
PUSH @C
ADD
POP @A
```

Dans tous les cas, le processeur effectue *le même nombre d'accès mémoire*. Dans les cas (a) (b) et (c), les transferts mémoire-registre sont seulement masqués par la syntaxe, qui est plus concise.

(n)	(m)	Exemples
3	0	SPARC, MIPS, HP PA, POWER, PowerPC, ALPHA
2	1	Motorola 68000, IBM 360, Intel 8086
2	2	PDP-11
3	3	VAX-11

Table 4.1: Les modèles caractéristiques de différentes machines classiques

La table 4.1 présente les modèles caractéristiques de différentes machines classiques. Cependant, il faut souligner très fortement que *les architectures des microprocesseurs actuels ont totalement convergé vers le modèle (3,0) et sont donc toutes des architectures chargement-rangement.* L'exception - de taille - a longtemps été les architectures Intel. Héritières du 8086, elles ont suivi le modèle (2,1), jusqu'à la gamme Pentium (architecture IA-32). Depuis 99, la nouvelle architecture Intel, l'IA-64, est une architecture chargement-rangement. Les raisons de cette convergence, ainsi que le fait que les architectures chargement-rangement sont plus connues sous le nom d'architectures RISC, seront discutées dans les chapitres suivants.

4.2 Format des instructions

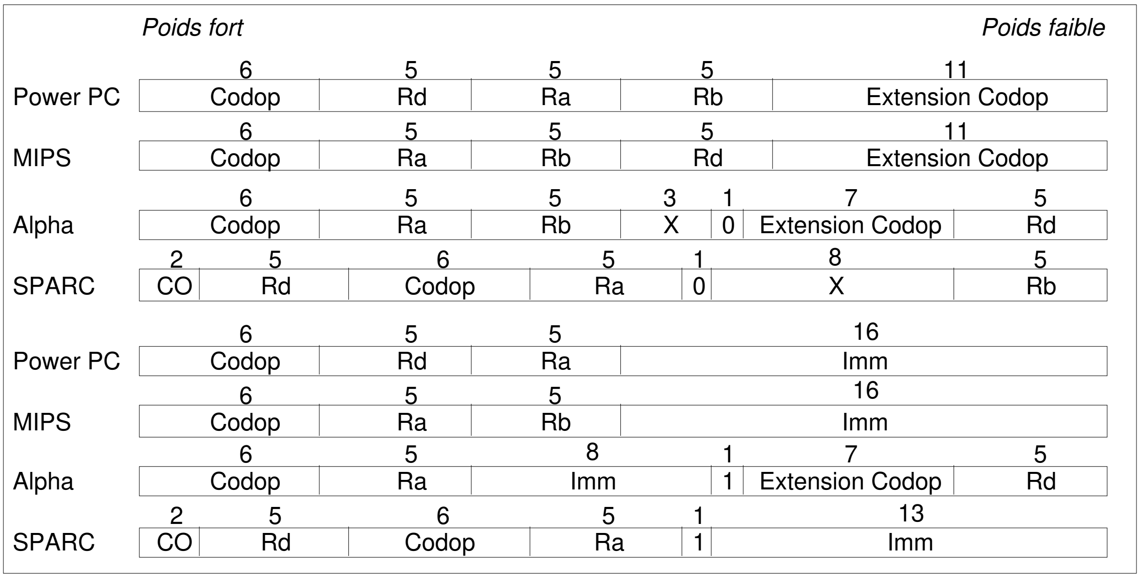

Figure 4.2: Les formats d'instructions entières pour quelques processeurs RISC

Le format des instructions est leur codage binaire. Il est absolument spécifique d'une architecture logicielle, mais les formats des machines RISC présentent tous la même structure.

Le format doit décrire l'opération et fournir le moyen d'accéder aux opérandes. La fig. 4.2 présente trois exemples de formats. On trouve plusieurs champs : le code opération (CODOP), qui décrit l'opération effectuée par l'instruction (addition, soustraction, chargement, etc.) et des champs registres qui décrivent les opérandes. Dans l'exemple, seuls l'emplacement des champs numéro de registre, et le codage des opérations, diffèrent. On voit comment le format décrit l'architecture logicielle : la largeur des champs numéros de registres est 5 bits ; le compilateur dispose donc de $2^5 = 32$ registres.

Les architectures RISC définissent toutes un format fixe : toutes les instructions sont codées sur un mot de taille fixe, de la taille d'un mot mémoire (32 ou 64 bits). L'architecture IA-32 utilse un format variable, où la taille est adptée à la quantité d'information nécessaire : typiquement, un NOP (non-opération, instruction qui ne fait rien) demande un octet ; d'autres instructions en demanderont beaucoup plus.

Dans la suite, tous les exemples utiliseront une architecture 32 bits, et un format d'instruction également 32 bits.

On notera que nous n'avons présenté ci-dessus que les formats des machines chargement-rangement. En effet, ces formats sont assez simples, plus précisément *orthogonaux* : les champs CODOP et opérandes sont distincts et indépendants. Le formats des architectures Intel jusqu'à la gamme Pentium sont hautement non orthogonaux, et donc excessivement compliqués, car ils devaient rester compatibles avec les formats très contraints du 8086.

4.3 Langage de haut niveau et langage machine

La fig. 4.3 résume les principales structures des LHN impératifs (Pascal, C ou Fortran). Les caractéristiques des langages plus évolués (fonctionnels, objets etc.) n'étant pas significatives dans le contexte des architectures logicielles, ne seront pas considérées ici.

Les constantes des LHN sont intégrées au programme. Sinon, on pourrait les modifier par erreur. Le code n'est en principe pas modifiable... Les variables des LHN sont ultimement réalisées par un ensemble de cases mémoire.

Une affectation *variable = expression* donne lieu à la séquence

```
Inst1
...
Instn
STxx Registre resultat, @variable
```

où Inst1, ..., Instn calculent l'expression. Si les instruction Inst1, ..., Instn font intervenir des variables, la séquence contient des chargements. Ainsi, l'instruction LHN A = B + C; où A, B et C sont des entiers, se compile (on suppose R1 = @A, R2 = @B, R3 = @C) en :

- Structures de données : Variables et constantes.
  - scalaire: entiers, caractères, flottants etc.
  - tableaux
  - enregistrements
  - pointeurs
- Instructions
  - Affectation : variable = expression
  - Contrôle interne
    - * Séquence : debut Inst1 ; Inst2 ;...;Instn fin
    - * Conditionnelle : Si ExpBool alors Inst_vrai sinon Inst_faux
    - * Repetition : Tant que ExpBool faire Inst
      Repeter Inst jusqu'a ExpBool
  - contrôle externe : appel de fonction

Figure 4.3: Les composantes des LHN impératifs

```
LD R12, 0(R2)          ; R12 ← B
LD R13, 0(R3)          ; R13 ← C
ADD R11, R12, R13      ; R11 ← B + C
ST R11, 0(R1)          ; A ← R11
```

En réalité, le compilateur n'effectue pas toujours une traduction "mot-à-mot" du programme source. Le degré de fidélité est contrôlé par les options d'optimisation de la commande de compilation, dont le but est d'améliorer les performances. L'exemple le plus simple est la séquence :

```
A = B + C ;
A = A + D;
```

La fidélité absolue implique le code suivant :

```
LD R12, 0(R2)          ; R12 ← B
LD R13, 0(R3)          ; R13 ← C
ADD R11, R12, R13      ; R11 ← B + C
ST R11 0(R1)           ; A ← R11
LD R11 0(R1)           ; R11 ← A
LD R14 0(R4)           ; R14 ← D
ADD R11 R11 R14        ; R11 ← A+D
ST R14 0(R4)           ; A ← R11
```

Les deux accès mémoire intermédiaires (ST R11 suivi de LD R11) sont inutiles, et sont supprimés par un compilateur optimisant. Ici et dans la suite, l'allocation des registres n'est pas conforme à celle que génèrerait un compilateur, pour une meilleure lisibilité.

Plus généralement, dans les architectures chargement-rangement, un compilateur optimisant tentera de supprimer toutes les transactions mémoire inutile : idéalement, une variable est chargée lors de sa première lecture, et écrite en mémoire lorsqu'elle n'est plus utilisée, ou aux frontières de procédure. En pratique, comme le nombre de registres disponibles n'est pas suffisant, le compilateur doit effectuer un choix judicieux, c'est l'*allocation de registres*. L'importance des options d'optimisation est donc considérable.

Les compilateur proposent en général une option (-S sous Unix), qui est compatible avec les options d'optimisation, et qui permet de récupérer une version du code généré en langage machine, dans un fichier suffixé *.s*. On peut utiliser cette option pour étudier l'effet des optimisations sur de petits programmes.

L'évaluation des expressions entières des LHN utilise les instructions entières, qui comprennent toutes les instructions qui accèdent aux registres entiers, donc les instructions de chargement-rangement et les instructions arithmétiques et logiques. Elles travaillent toutes sur les registres entiers généraux, en format registre-registre ou registre-immédiat. Les instructions de contrôle des LHN utilisent les instructions de comparaison et de branchement.

4.4 Instructions arithmétiques et logiques

Ce sont les instructions d'addition (ADD) et de soustraction (SUB), de multiplication (MUL), de division (DIV), etc.

Mode d'adressage

Le *mode d'adressage* définit la manière dont les instructions accèdent aux opérandes. Le terme d'adressage ne correspond pas bien au modèle d'exécution (3,0), puisque les opérandes sont forcément situés en registre ; nous le conservons cependant, car il est traditionnel.

Mode registre

Dans ce mode, l'opérande est situé dans un registre général. Par exemple, les trois opérandes de l'instruction

ADD Rd, Rs1, Rs2 ; Rd $<-$ Rs1 + Rs2

sont adressés en mode registre. Plus généralement, dans la première partie de la fig. 4.2, les deux opérandes sont adressés en mode registre.

Mode immédiat

Dans le mode immédiat, l'opérande est contenu dans l'instruction. L'immédiat est disponible dans l'instruction, donc dans le registre RI, mais il n'est pas de la taille requise : en général l'instruction a la même largeur que le chemin de données, et l'immédiat ne peut donc l'occuper tout entière. L'opérande est obtenu soit en ajoutant des 0 en tête de l'immédiat (Alpha), soit par extension de signe (SPARC, Power, MIPS). Dans la deuxième partie de la fig. 4.2, le deuxième opérande est un immédiat.

ADD Rd Rs1 imm ; Rd $\leftarrow$ Rs1 + ES(imm)

Le mode d'adressage immédiat permet, entre autres, de compiler les constantes. Si R1 = x et R2 = y,

x = y + 0x12,

se compile en

ADD R1, R2, Ox12.

Mais, pour compiler

x = y + 0x12348765,

on ne peut PAS écrire ADD R1, R2, 0x12348765. En effet, l'immédiat est limité à moins que la taille du plus grand entier représentable, car en général, l'instruction est de la même taille que le chemin de données. Par exemple, les immédiats Power sont compris entre -2^{15} et $2^{15}-1$, les immédiats SPARC entre -2^{12} et $2^{12}-1$, alors que dans les deux cas, les entiers représentables sont compris entre -2^{31} et $2^{31}-1$. La valeur 0x12348765 doit donc être préalablement écrite dans un registre. Suivant les processeurs et les formats, des instructions spécifiques pour l'écriture en registre d'un immédiat de la taille du mot-processeur, sont définies. Il faudra néammoins toujours deux instructions au moins pour un immédiat trop large. Il est donc utile d'avoir un champ immédiat aussi grand que le format le permet, pour limiter les situations où deux instructions sont nécessaires.

Les deux opérandes source ne peuvent pas être tous deux des immédiats : la taille de l'instruction est trop petite pour en accepter deux. En outre, une telle instruction serait inutile : le compilateur peut calculer directement le résultat d'une affectaion du type x = cst1 + cst2.

Les modes registre et immédiat sont également disponibles, respectivement pour (source et destination) et (source seulement) sur les architectures Intel.

Instructions

Les instructions d'addition et de soustraction peuvent avoir plusieurs variantes :

- positionnement ou non de certains drapeaux : une version positionne les drapeaux, et l'autre ne les positionne pas. L'existence des deux versions est utile pour les tests, comme on le verra plus loin.
- signé ou non signé : l'addition signée positionne le drapeau d'overflow, l'addition non signée ne le positionne pas. En effet, lorsque les opérandes représentent un entier naturel, le drapeau de retenue suffit à indiquer un dépassement de capacité, alors que la situation est plus compliquée lorque les opérandes représentent un entier relatif, comme on l'a vu au chapitre précédent. L'architecture x86 ne calcule qu'en signé.
- addition des opérandes et du drapeau de retenue ; cette instruction est par exemple utile pour additionner des opérandes en multiple longueur.

La table 4.2 montre les diverses versions de l'instruction d'addition dans le jeu d'instruction PowerPC.

Mémonique	Syntaxe	Effet
ADDx	ADDx Rd, Ra, Rb	Rd $\leftarrow$ Ra + Rb
ADD		Aucun flag affecté
ADDo		Flags Z, E, N affectés
ADD.		Flag O affecté
ADDo.		Flags 0, Z, E, N affectés
ADDcx	ADDcx Rd, Ra, Rb	Rd $\leftarrow$ Ra + Rb C Flag affecté , x commme pour ADD
ADDEx	ADDcx Rd, Ra, Rb	Rd $\leftarrow$ Ra + Rb + C C Flag affecté , x commme pour ADD
ADDI	ADDcx Rd, Ra, Imm$_{16}$	Rd $\leftarrow$ Ra + Imm$_{16}$ Aucun flag affecté

Table 4.2: Instructions d'addition du PowerPC

On trouve également des instructions logiques, qui effectuent bit à bit les opérations logiques Et (AND), Ou (OR), Ou exclusif (XOR), Complément (NEG), etc. et des instructions de décalage arithmétique ou logique, circulaires avec ou sans la retenue d'entrée, à droite ou à gauche.

L'addition et les décalages sont évidemment utilisées pour les compiler les instructions de calcul. Elles sont aussi fortement utilisés pour les calculs d'adresse (fig. 4.4).

4.5 Instructions d'accès mémoire

On ne traitera ici que les architectures RISC, où les accès mémoires sont limités aux transferts entre la mémoire et les registres du processeur. Des instructions spécifiques

```
int v [N], k ;
...
temp = v[k] ;
v[k] = v [k+1];
v[k+1] = temp ;
```

Initialement, R1 = @v[0], R2 = k

```
SLL  R2, 4, R2      ;deplacement k← k*4
ADD  R2, R2, R1     ; R2 ← @ v[k]
LD   R3, 0(R2)      ; temp = R3 ← v[k]
LD   R4, 4(R2)      ; R4 ← v[k+1]
LD   R4, 0(R2)      ; v[k] ← R4
ST   R3, 4(R2)      ; v[k+1] ← temp
```

Figure 4.4: Utilisation des instructions arithmétiques pour le calcul d'adresse

assurent ce transfert.

Accès entiers

Entiers signifie ici tous les accès qui echangent entre la mémoire et un registre entier, ie un registre sur lequel opère l'UAL, indépendamment du type de la donnée.

La taille des registres entiers d'un processeur est fixe (32 ou 64 bits). Il existe donc des instructions de chargement et de rangement, qui copient un mot de la mémoire vers le registre, ou l'inverse.

```
LD Rd, @variable ; Rd ← Mem[@variable]
ST Rs, @variable  ; Mem[@variable] ← Rs
```

Les instructions arithmétiques et logiques travaillent sur le format correspondant au nombre de bits des registres entiers, soit 32 bits ou 64 bits. Mais les variables du LHN peuvent être plus petites, par exemple un caractère sur 8 bits. Il peut être donc être nécessaire de convertir un type vers un autre, lors d'un accès mémoire en lecture, par exemple le chargement d'un octet vers un registre 32 bits.

L'alternative principale pour la conversion porte sur l'extension : l'octet étant placé dans l'octet bas du registre, les bits restants sont, soit forcés à 0, soit positionnés par extension de signe. Les formats de données les plus couramment accessibles en mémoire de cette façon sont les octets, les demi-mots (2 octets), les mots (4 octets) et, pour les microprocesseurs 64 bits, les doubles mots (64 bits). On a ainsi, pour un microprocesseur 32 bits, les instructions (fig. 4.5) :

- LDUB (Load Unsigned Byte) pour charger un octet sans extension de signe, On suppose @A = 1000, et le plan mémoire :

1000	82
1001	AB
1002	56
1003	78

```
LDUB   R1, @A  ; R1 ← 00000082
LDB    R2, @A  ; R2 ← FFFFFF82
LDUH   R3, @A  ; R3 ← 0000AB82 si LE, 000082AB si BE
LDH    R4, @A  ; R4 ← FFFF AB82 si LE, FFFF82AB si BE
LD     R5, @A  ; R5 ← 7856AB82 si LE, 82AB5678 si BE
```

Figure 4.5: Les instructions de chargement

- LDB (Load Byte) pour charger un octet avec extension de signe,
- LDUH (Load Unsigned Half-Word) pour charger un demi-mot sans extension de signe,
- LDH (Load Half-Word) pour charger un demi-mot avec extension de signe.

Le rangement ne pose pas un problème symétrique : on veut seulement, par exemple, pouvoir ranger un octet sans perturber les octets voisins. On a donc seulement trois instructions de rangement : ST pour un mot, STH pour un demi-mot et STB pour un octet ; dans tous les cas, c'est la partie basse du registre qui est rangée.

Accès flottants

Les accès flottants sont les transferts entre la mémoire et les registres flottants. Les unités flottantes travaillent en général en double précision, et le codage est la standard IEEE 754. On trouvera donc deux instructions de chargement : LDFS , charger un flottant simple précision (4 octets), avec conversion codage simple précision vers codage double précision, et LDF, charger un flottant double précision (8 octets). Ici, le rangement est symétrique : on peut vouloir arrondir un résultat double précision vers la simple précision, le standard définissant l'algorithme d'arrondi.

Modes d'adressage

On appelle *adresse effective*, et on note EA, l'adresse de la case mémoire lue ou écrite.

Inclure une adresse mémoire explicite dans une instruction est exclu: l'adresse est de la taille du chemin de données, donc doublerait la taille au moins des instructions mémoire. D'autre part, un programme utilisateur accède rarement à des données qui ont une adresse absolue, mais plutôt à des variables locales à une procédure. Ces variables sont placées en

	6	5	5	16
Alpha	Codop	Ra	Rb	Imm

Figure 4.6: Format des instructions d'accès mémoire pour l'architecture Alpha

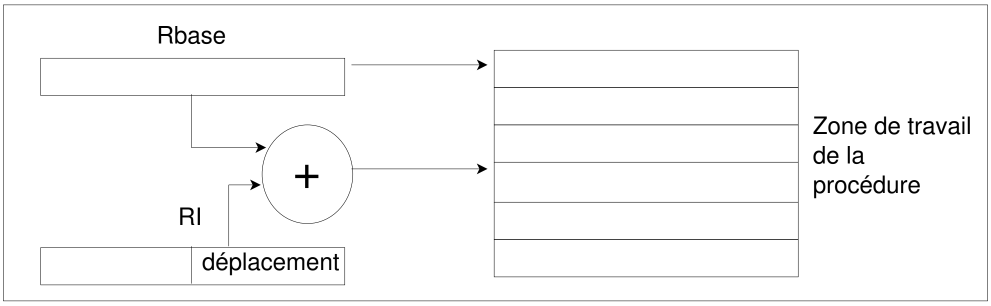

Figure 4.7: Accès à une variable de procédure

mémoire à partir d'une adresse associée à la procédure et contenue dans un registre (ce mécanisme sera détaillé dans la partie appel de sous-programmes).

Pour ces deux raisons, les adresses mémoires sont calculées comme somme d'un registre et d'un immédiat ou d'un registre ; on parle d'adressage indirect.

La table 4.3 résume les modes d'adressage que l'on trouve dans un certain nombre d'architectures RISC, et qui sont détaillées dans la suite.

Architecture	Base + Dep.	Base + Index	Auto-incrémenté
Alpha	oui		
Power	oui	oui	oui
MIPS jusqu'à R3000	oui		
MIPS à partir R10000	oui	oui	
SPARC	oui	oui	

Table 4.3: Modes d'adressages des architectures RISC

Les formats des instructions mémoire sont souvent identiques à ceux des instructions UAL, car les champs sont identiques. L'architecture Alpha fait exception : l'immédiat étant petit et interprété positif, les instructions mémoire ont le format de la fig. 4.6, où l'immédiat est obtenu par extension de signe de Imm.

Base + Déplacement

L'adresse est la somme d'un registre (*registre de base*) et d'un immédiat, le *déplacement*, positif ou négatif :

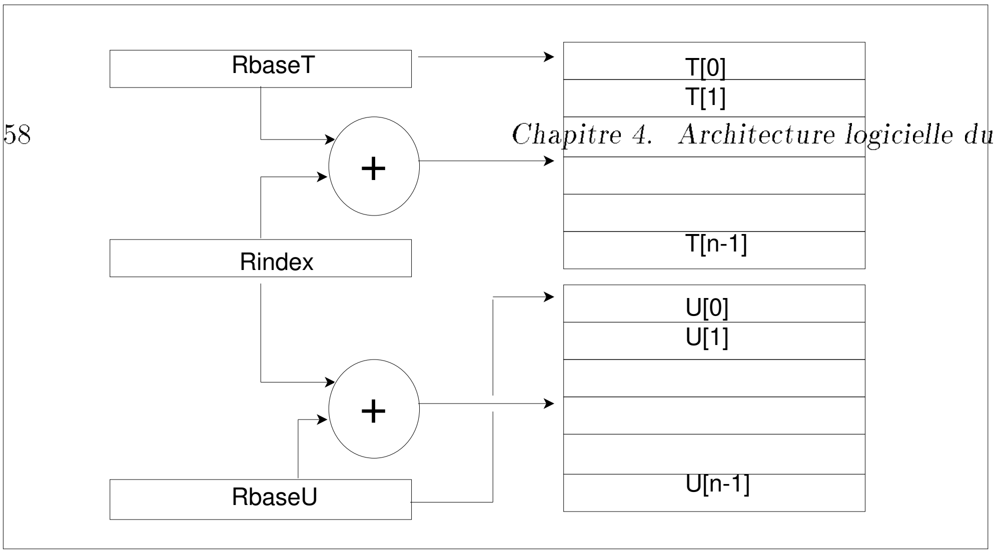

Figure 4.8: Accès à un élément de tableau

EA = Rs1 + ES(Imm)

On utilisera dans la suite la syntaxe Imm (Rs1), par exemple :

LD R1, 4(R2) ; R1 ← Mem [R2 +4].

Ce mode d'adressage est fondamental. Il permet d'accéder aux variables d'une procédure (fig. 4.7) : le registre de base contient l'adresse de début de la zone des variables de la procédure ; le déplacement donne l'adresse de la variable par rapport à la zone de données de la procédure.

Il permet également d'accéder aux variables statiques, dont l'adresse est absolue. Il faut pour cela que l'adresse de la variable soit préalablement chargée dans le registre de base Rs1 ; on accède alors à la variable par LD/ST 0(Rs1).

Base + Index

Certaines architectures ont un mode d'adressage mémoire registre-registre.

EA = Rs1 + Rs2

On utilisera la syntaxe (Rs1 + Rs2), par exemple

LD R1, (R2 + R3) ; R1 ← Mem[R2 + R3]

L'utilisation la plus courante est l'accès à un élement de tableau (fig. 4.8) : le premier registre est utilisé comme registre de base qui contient l'adresse de début d'un tableau, et le second registre contient l'indice i de l'élément du tableau multipliée par la taille de l'élément. Pour accéder aux éléments successifs d'un tableau, il suffit d'incrémenter le registre d'index.Par exemple, le code

```
int i ;
for( i = 0 ; i¡ n-1 ; i++)
   x(i) = ...;
```

se traduit, en supposant que R1 soit initialisé à @x[0] et R2 à 0, par une boucle dont le corps est :

LD R10, (R1 + R2) ; R10 ← x [i]
...
ADD R2, R2, 4 ; incrementation index
...

Base + Index + Déplacement

Certaines architectures non chargement-rangement, en particulier les architectures Intel, proposent la combinaison des deux modes précédents

Il faut noter une différence majeure avec les cas précédents : le déplacement peut s'étendre jusqu'à 32 bits. Ce MA autorise alors l'adressage absolu, donc une compilation particulièrement simple et un mode d'accès efficace aux variables statiques. La combinaison avec le registre d'index et le registre de base s'applique aux accès aux tableaux statiques à 1 et 2 dimensions respectivement.

Pour un tableau variable de procédure, ce mode d'adressage permet de cumuler les avantages de base + déplacement et base + index. Le registre de base contient l'adresse de base de la procédure, le déplacement contient l'adresse relative d'un élément du tableau par rapport au premier élément du tableau, et le registre d'index stocke l'indice, à un facteur d'échelle près. En fait, l'architecture Pentium autorise même à introduire le facteur d'échelle, avec le mode d'adressage

Base + Index * scale + deplacement, avec scale = 1,2,3 ou 4
EA = Base + Index * /1, 2, 4, 8/ + deplacement

L'inconvénient de ce mode d'adressage est qu'il nécessite d'additionner trois adresses et d'effectuer un décalage, ce qui prend beaucoup plus de temps que la simple addition de deux adresses dans un additionneur rapide. Pour cette raison, liée aux contraintes temporelles pour la réalisation de pipelines avec des fréquences d'horloge élevées, ce mode d'adressage n'est pas spécifié dans les architectures RISC.

Le mode auto-incrémenté

Il s'agit d'une variante des modes base + déplacement et base + index. La seule différence est que le registre Rs1 est aussi modifié. On peut par exemple définir une instruction :

LDX Rd, (Rs1 + Rs2) ; Rd ← Mem[Rs1 + Rs2], puis Rd ← Rs1 + Rs2].

Ce mode permet de balayer successivement les éléments d'un tableau sans avoir à incrémenter le registre d'index par une instruction explicite.

Dans des machines des années 60, on trouvait couramment des adressages indirects via la mémoire : le contenu d'une case mémoire contenait l'adresse de l'opérande. Ces modes d'adressage ont disparu au profit des modes indirects via registre.

4.6 Instructions de comparaison

Une instruction de comparaison (CMP) effectue la soustraction de ses opérandes. Elle ne conserve pas le résultat arithmétique, mais seulement les code-conditions (CC).

Syntaxe : CMP Ra, Rb

Il s'agit en fait d'instructions UAL, mais qui sont utilisées pour compiler les structures de contrôle. Les instructions de comparaison évaluent des prédicats des LHN ; un prédicat est une variable à valeur booléenne fonction des autres variables du programme, par exemple le résulat de l'évaluation $x = y$. Les prédicats élémentaires sont :

- l'égalité (de variables de types quelconque) ;
- les relations d'inégalité arithmétiques ($>, \geq, <, \leq$) sur les
  - naturels
  - relatifs;
  - flottants
- les évènement exceptionnels arithmétiques (Overflow entier, drapeaux IEEE754)

Un tel prédicat sur un couple (x, y) est en fait un prédicat sur $x - y$:

$$x = y \rightarrow x - y = 0$$
$$x \geq y \rightarrow x - y \geq 0$$

etc. On a donc besoin de soustraire les registres contenant le valeurs de x et y. Le résultat n'est pas la valeur de la soustraction, sur n bits, mais un ensemble de booléens, les code-conditions, sur quelques bits. Ces booléens sont rangés dans un registre implicite, le Registre Code Condition.

Un autre type d'instruction de comparaison calcule directement les prédicats, et positionne le résultat dans un registre utilisateur. C'est le cas du jeu d'instruction du MIPS, avec des instruction du type "set conditionnel" :

S*cond* Rd, Ra, Rb ; Rd $\leftarrow$ 0 si Ra *cond* Rb, 1 sinon

cond exprime le prédicat. L'inconvénient est de consommer un registre 32 bits pour stocker un booléen. L'architecture Alpha a des instructions analogues.

Enfin, l'architecture IA 64 définit 64 registres 1 bit, en plus des registres de travail arithmétiques, pour mémoriser le résultat d'instructions de type S*cond* (CMPS*cond* dans la syntaxe Intel).

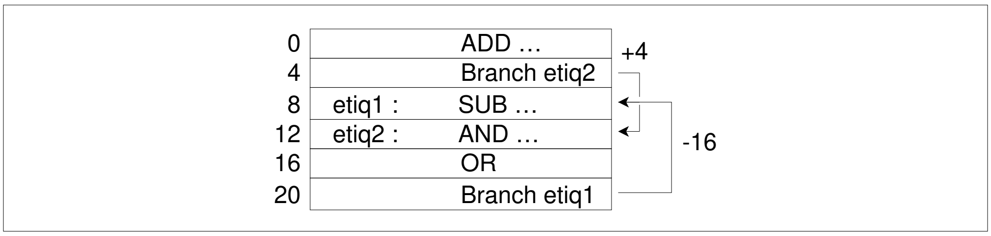

Figure 4.9: Calcul des adresses des branchements relatifs

4.7 Instructions de saut et de branchement

Les instructions en langage machine s'exécutent en séquence. Des instructions particulières, les *branchements*, permettent d'implémenter toutes les structures de contrôle interne des LHN, conditionnelles et boucles. L'effet d'un branchement est PC $\leftarrow$ nouvelle valeur. Les branchements de séquence peuvent être :

- conditionnels ou inconditionnels ;
- *absolus* ou *relatifs*
  - absolu : PC est chargé avec une adresse complète, présente dans un registre ; l'effet est PC $\leftarrow$ Adresse
  - relatif : la nouvelle adresse est calculée à partir de la valeur courante de PC plus un déplacement : PC $\leftarrow$ PC + déplacement

Pour les branchements relatifs, la valeur de PC à laquelle s'ajoute le déplacement est celle de l'instruction qui suit le branchement, et non celle du branchement : PC est incrémenté en même temps que l'instruction est lue. Pour éviter des calculs fastidieux, les programmes en langage d'assemblage utilisent des étiquettes symboliques (fig. 4.9).

Branchements conditionnels sur registre code-conditions

L'instruction conditionnelle

Si condition alors Bloc I1 sinon Bloc I2

se compile classiquement par le schéma de la fig. 4.10.

Les instructions de comparaison ont précisément pour fonction de permette l'évaluation des conditions les plus courantes sans polluer les registres de calcul. Cependant, une architecture à RCC permet de calculer implicitement des conditions, sans instruction de comparaison ; par exemple, l'instruction PowerPC ADDo. positionne tous les bit du RCC. En revanche, MIPS, Alpha et IA-64 imposent une instruction explicite de comparaison.

Les instructions de branchement conditionnel n'exécutent la rupture de séquence que si la condition booléenne testée est vraie. Dans le cas contraire, l'exécution des instructions

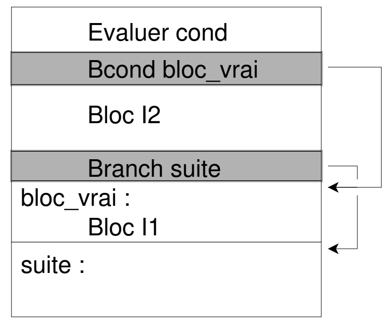

Figure 4.10: Schéma de compilation d'une conditionnelle

continue en séquence. La forme précise de ces instructions conditionnelles dépend de la manière dont sont mémorisés les codes conditions. Dans une architecture à RCC, les instructions conditionnelles testent les code-conditions contenues dans ce registre, qui est implicite dans l'instruction. La forme générale des instructions est alors

B*cond* deplacement

cond représente la condition testée.

La fig. 4.11 donne un exemple de de réalisation des conditionnelles.

```
       CMP R1, R2         ; 1 effectue R1 - R2 et positionne RCC
       BGT vrai           ; 2 Saute les deux instructions suivantes si R1>R2
       ADD R3, R0, R2     ; 3 Transfere le contenu de R2 dans R3
       BA suite           ; 4 Saute l'instruction suivante
vrai : ADD R3, R0, R1     ; 5 Transfere le contenu de R1 dans R3
suite :                   ; 6 Suite du programme
```

Figure 4.11: Code de la conditionnelle *Si R1 > R2, alors R3 = R1 sinon R3 = R2* sur une architecture à RCC. Si R1 > R2, la suite d'instructions exécutées est 1-2-5-6 ; sinon 1-2-3-4-6.

Les conditions testables dépendent de l'architecture, mais on trouve typiquement les mnémoniques de la table 4.4. La sémantique exacte de ces mnémoniques est fournie par la combinaison des codes conditions qu'ils testent, par exemple E correspond au code condition Z = 1, mais leur interprétation est intuitive ; typiquement, la séquence du type :

```
CMP R1, R2
BGT cible
```

branche si R1 > R2, lorsque R1 et R2 sont interprétés comme des entiers relatifs.

BE	Egal
BGT	Supérieur, signé
BLE	Inférieur ou égal, signé
BGTU	Supérieur, non signé
BLEU	Inférieur ou égal, non signé
BA	Toujours (branchement inconditionnel
BC	Retenue (code condition C = 1)
BO	Overflow (code condition O = 1)

Table 4.4: Mnémoniques des branchements conditionnels

Branchements conditionnels sur registre général

Les architectures qui ne mémorisent pas les codes condition dans un RCC, mais dans des registres généraux, ont des instructions de branchement plus simples, du type

B*cond* Rd, deplacement

qui branche si Rd vérifie la condition. C'est le cas des architectures Alpha et MIPS. L'architecture MIPS propose une instruction de branchement-comparaison : BE/BNE Rd, Ra, deplacment, qui branche si Rd = Ra (resp si Rd $\neq$ Ra).

Discussion

Les branchements conditionnels sur code-condition posent un problème de programmation, et un problème d'efficacité.

- Programmation : il faut pouvoir conserver la condition, c'est à dire ne pas écraser le contenu de RCC par une opération arithmétique qui positionne les CC, ou par une nouvelle comparaison.
- Efficacité : on verra par la suite qu'avec le modèle d'exécution pipeliné, les ruptures de séquence sont potentiellement coûteuses.

Pour résoudre le premier problème, les architectures à RCC définissent deux versions de chaque opération arithmétique, l'une positionnant le RCC et l'autre pas, comme on l'a vu pour le PowerPC. Les instructions qui ne positionnent pas le RCC sont massivement utilisées par les compilateurs, en particulier pour les calculs d'adresses.

En outre, dans l'architecture PowerPC, le registre code condition est décomposé en 8 champs de 4 bits (CR0 à CR7) pour les comparaisons, ce qui revient à disposer de 8 sous registres code condition. Enfin, les instructions arithmétiques entières (ADD., ADDo.) ne positionnent que le champ CR0.

En revanche, dans l'architecture x86, qui utilise un RCC, toutes les instructions arithmétques positionnent le RCC. Le jeu d'instruction contient donc des instructions spécifiques, qui ne le positionnent pas : DEC, décrémentation par une constante ; NEG, calcul de l'opposé ; MOV, transfert.

Ce problème n'existe pas dans les architectures ou les code-conditions sont mémorisés dans un registre général, au prix de l'utilisation d'un registre général. Un avantage important est un moins grand nombre de branchement pour compiler les conditionnelles complexes (fig. 4.12).

(a)

```
        R10 ← cond1
        R20 ← cond2
        AND R30, R10, R20      ; R30 ← (cond1 et cond2)
        BR R30 vrai            ;
        ADD R3, R0, R2         ; condition fausse, R3 ← R2
        BA suite
vrai :  ADD R3, R0, R1         ; condition vraie, R3 ← R1
suite :                        Suite du programme
```

(b)

```
        evaluation cond1
        Bcond faux             ; si cond1 fausse
        evaluation cond2       ; cond1 vraie
        Bcond faux             ; si cond2 fausse
        ADD R3, R0, R1         ; condition vraie, R3 ← R1
        BA suite
faux :  ADD R3, R0, R2         ; condition fausse, R3 ← R2
suite :                        Suite du programme
```

Figure 4.12: Compilation de la conditionnelle complexe *Si (cond1) et cond2) alors R3 = R1, sinon R3 = R2* (a) avec branchements sur registres (b) avec branchements sur RCC.

Les deux techniques utilisent néammoins des branchements conditionnels pour implémenter l'instruction conditionnelle. Les branchements conditionnels peuvent ralentir l'exécution des instructions dans les pipelines qu'utilisent les processeurs modernes. Certaines architectures logicielles offrent donc des instructions de *transfert conditionnel*, ou *instructions prédiquées*, qui permettent d'éviter l'utilisation de branchement dans certains cas, et notamment pour l'exemple ci-dessus. Soit le transfert conditionnel

Mcond Rk, Ri, Rj ;

défini de la manière suivante: si Rk contient un booléen vrai, alors Rj est transféré dans Ri, sinon instruction neutre. La conditionnelle *Si R1 > R2, alors R3 = R1 sinon R3 = R2* se compile alors comme dans la fig 4.13. Il n'y a pas de rupture de séquence : l'instruction MGT est équivalente à un NOP si la condition est fausse.

```
CMP R4,R1,R2        ; Positionne les codes conditions dans R4
ADD R3,R0,R2        ; Transfere R2 dans R3
MGT R4,R3,R1        ; Si condition vraie (R1>R2), transfere R1 dans R3
```

Figure 4.13: Utilisation du transfert conditionnel

Les avantages potentiels des instructions prédiquées sont considérables. Toutes les conditionnelles peuvent se compiler sans rupture de séquence, à condition que toutes les instructions, qui peuvent potentiellement intervenir dans le bloc d'instructions conditionnées, puissent être prédiquées, et pas seulement une instruction de transfert. Ces avantages, en relation avec des techniques de compilation optimisantes, ont été étudiés depuis plusieurs années, et l'architecture IA-64 d'Intel propose un jeu d'instruction complètement prédiqué, aec 64 registres 1 bit pour abriter le prédicat. Le registre prédicat constitue alors un opérande supplémentaire

Les instruction prédiquées sont particulièrement utiles pour déplacer des instructions. Touours pour l'exemple du calcul d'un maximum, avec un jeu d'instruction à RCC, on peut remplacer les codes de la fig. 4.12 par celui de la fig. 4.14

```
    ADD R3, R0, R1      ; 1 transfere le contenu de R1 dans R3
    CMP R1, R2          ; 2 effectue R1 - R2 et positionne RCC
    BGT suite           ; 3 Saute les deux instructions suivantes si R1>R2
    ADD R3, R0, R2      ; 4 transfere le contenu de R2 dans R3
suite :                 ; 5 Suite du programme
```

Figure 4.14: Code de la conditionnelle *Si R1 > R2, alors R3 = R1 sinon R3 = R2* sur une architecture à RCC

L'intérêt de la transformation est d'une part la diminution de la taille statique du code, ce qui est rarement important, et la suppression d'un des deux branchements. Mais si l'instruction déplacée est susceptible de produire une exception - arithmétique flottante, accès non aligné -, ou de modifier l'état du processeur, par exemple en positionnant un drapeau, ce mouvement est impossible : la sémantique de la conditionnelle interdit que la branche non prise puisse avoir un efft quelconque. Or c'est par exemple le cas pour le code

si (R1 < R2) alors x = y sinon x = z

si y et z ne sont pas déja en registre : la lecture anticipée de y ou de z peut provoquer une exception d'accès non aligné ou de protection. En revanche, une instruction prédiquée peut être déplacée sans contrainte, puisqu'elle n'est pas exécutée si le prédicat est faux. En fait, une partie du problème n'est que déplacée : le compilateur a une tâche plus facile, mais le concepteur de microprocesseur doit implémenter cette sémantique d'annulation, ce qui n'est pas très simple, comme on le verra par la suite.

Boucles

Les instructions de boucle peuvent toujours être implantées à l'aide de l'instruction conditionnelle. Il y a deux type de boucles : tant que (incluant faire n fois) et répéter. Leur schéma de compilation est décrit fig. 4.15 : Dans le cas d'une boucle faire, il est nécessaire

Tant que		*Tant que optimisé*		*Répéter*	
deb :	test cond		BA test	deb :	code boucle
	Bcond fausse suite	deb :	code boucle		test cond
	code boucle	test	test cond		Bcond vraie deb
	BA deb		Bcond vraie deb		
suite :	...				

Figure 4.15: Schémas de compilation des boucles

de gérer explicitement un compteur de boucles (nombre d'itérations exécutées) et de tester à chaque itération si le compteur a atteint la valeur correspondant au nombre d'itérations à effectuer.

Certaines architectures logicielles, comme par exemple le Power PC, offrent une instruction de gestion de boucle. Essentiellement, il s'agit d'une version enrichie du branchement conditionnel. Un registre non général (registre compte-tour) est initialement chargé avec le nombre d'itérations à effectuer. L'instruction

BCNT Ri, deplacement

effectue les actions suivantes:

Ri := Ri - 1
Si Ri $\neq$ 0, CP := CP + deplacement sinon continuer en sequence.

Cette instruction est utile pour les boucles for, plus présentes dans le calcul numérique que dans les autres types de code. L'architecture x86 offre une instruction analogue, LOOP.

4.8 Sous-programmes

Fonctionnalités

Les sous-programmes sont des unités de code indépendantes, fonctions et procédures.

Les fonctionnalités nécessaires à l'appel de procédure sont (fig. 4.16)) :

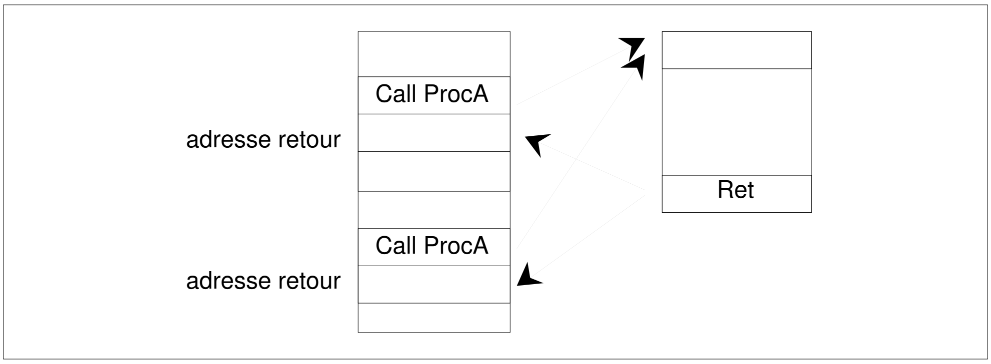

Figure 4.16: Appel et retour de sous-programme

- *Appel* : Branchement à une adresse.
- *Retour* : Branchement à l'instruction qui suit l'instruction d'appel. L'adresse de cette instruction est appelée *adresse de retour* dans la suite. C'est nécessairement un branchement absolu (PC $\leftarrow$ adresse), et non relatif (PC $\leftarrow$ PC + déplacement), car l'adresse de retour n'est pas connue avant l'exécution.
- *Passage des paramètres* : Un sous-programme doit pouvoir se voir passer des arguments et éventuellement retourner une valeur.
- *Sauvegarde et restauration du contexte* : annuler toute modification de l'état du processeur due à l'appel.

Il est important de préciser les contraintes pesant sur ces fonctionnalités.

- *Compilation séparée.* Les sous-programmes peuvent être compilés séparément, puis liés ensembles pour constituer un programme. L'architecture logicielle doit donc fournir un support pour une situation où la portée de vision du compilateur est limitée
- *Appels emboîtés.* Un sous-programme en appelle d'autres ; le niveau d'emboîtement n'est pas connu à la compilation (récursivité). Un sous programme qui n'en appelle pas d'autre est dit *terminal.*

Les fonctionnalités ci-dessus sont supportés partiellement en matériel, partiellement par des conventions logicielles. Toutes les architectures offrent un support spécialisé pour l'appel de sous-programme. Pour les trois autres points, la proportion de logiciel et de matériel est variable. Cette section présentera donc plus d'exemples que de principe généraux.

Il existe cependant une caractéristique commune des architectures RISC : elles privilégient le cas des sous-programmes terminaux qui passent peu de paramètres. La raison est

Architecture	Syntaxe	Signification
SPARC	CALL Dep_{30}	R15 ← PC ; PC ← PC + ES (Dep_{30}*4)
	JMPL Ra, Dep_{13}, Rd	Rd ← PC ; PC ← Ra + ES (Dep_{13}*4)
	JMPL Ra, Rb, Rd	Rd ← PC ; PC ← Ra + Rb
Power PC	BL Dep_{24}	LR ← PC ; PC ← PC + ES (Dep_{24}*4)
	BLA Dep_{24}	LR ← PC ; PC ← ES (Dep_{24}*4)
	BLR	PC ← LR
	BLRL	PC ↔ LR

Table 4.5: Instructions d'appel et retour de sous-programme

que ceux-ci sont nettement plus fréquents que les autres [2]. Privilégier signifie que l'architecture est organisée pour accéler l'exécution de type de sous-programmes, tout en offrant le support nécessaire à l'exécution des sous-programmes plus complexes.

Appel et retour de sous-programme

Fonctionnalités

L'appel et le retour de sous-programme signifient une modification du registre compteur de programme :

Appel : PC ← adresse sous-programme
Retour : PC ← adresse de retour

Les procédures sont appelées à partir de n'importe quelle partie d'un programme. Pour que le programme appelant puisse continuer son exécution après la fin d'exécution de la procédure, il est nécessaire que l'instruction d'appel, avant de modifier PC, sauvegarde l'adresse de retour. En outre, l'emplacement de cette sauvegarde doit être connu de l'appelée. On a donc deux *pseudo*-instructions fondamentales :

"CALL dep " : sauvegarde ← PC; PC ← dep
"RET "   PC ← sauvegarde

L'action "sauvegarde ← PC" ne correspond à aucune instruction déjà vue. D'autre part, elle n'a d'utilité que pour les appels de procédure. La pseudo-instruction CALL est donc implémentée dans tous les jeux d'instructions comme une instruction spécifique, différente des branchements. En revanche, le positionnement de PC est par définition un saut ou un branchement. La pseudo instruction RET peut donc être implémentée sans instruction nouvelle.

Architectures Chargement-Rangement

L'adresse de retour est toujours rangée dans un registre. C'est, soit dans un registre général (SPARC, MIPS, Alpha), ou bien un registre particulier, nommé LR comme Link Register (Power PC) (table. 4.5). Comme les instructions sont alignées (4 octets), les deux zéros finaux des adresses absolues (CALL, BLA) ou relatives (JMPL, BL) ne sont pas stockés, ce qui permet, à format constant, de multiplier par 4 la portée des appels.

Exemple du SPARC

L'architecture SPARC propose deux instructions d'appel, CALL et JMPL. Comme les instructions sont alignées (4 octets), l'instruction CALL permet d'atteindre toute la mémoire (sur 32 bits). CALL sauvegarde l'adresse de retour dans le registre R15. JMPL sauvegarde dans un registre quelconque, mais limite l'amplitude du branchement à 2^{13} instructions.

Rappelons que l'architecture SPARC ne propose que des branchements du type PC $\leftarrow$ PC + ES(dep_{22}). Les branchements ne sont donc pas utilisables pour le retour, car l'adresse de retour varie suivant l'appel ; elle ne peut pas être encodée dans l'instruction.

La séquence typique d'appel/retour devrait donc être

CALL myproc
JMPL R15, 0, R0 ; R0 cable a 0, donc le seul effet est PC $\leftarrow$ R15

(en fait, l'instruction est JMPL R15, 8, R0, pour des raisons qu'on verra dans le chapitre Pipeline)

L'instruction CALL myproc peut être remplacée par une instruction du type JMPL Ra, Rb ou dep, Rd, le retour étant alors JMPL Rd, 0, R0. Il faut alors une convention logicielle pour le choix de Rd.

Une convention logicielle est un ensemble de contraintes sur l'état du processeur, en particulier l'utilisation des registres, du type "la clé est sous le troisième pot de fleurs". Ici, la convention précisera que le registre destination est toujours, par exemple, R28. Ainsi, un sous-programme compilé séparément a la garantie que l'appelante a laissé l'adresse de retour (la clé) dans le registre R28 (le troisième pot de fleurs). Ces contraintes doivent être respectées par tous les codes susceptibles d'être intégrés avec d'autres, en particulier ceux crées par un compilateur. En revanche, la convention n'a pas à être repectée par un code destiné à être exécuté isolément (*stand-alone*), par exemple un programme assembleur exercice.

Exemple du POWER PC

La séquence d'appel-retour est simplement :

BL ou BLA myfunc
BLR

CISC

L'adresse de retour est rangée en mémoire, via une pile mémoire. Dans l'architecture x86, le pointeur de pile est un registre spécialisé appelé SP (stack pointer) ; on dispose des instructions CALL et RET:

CALL val ; Empiler (PC) ; PC $\leftarrow$ nouveau PC.

L'adresse peut être absolue ou relative. Le mode d'adressage peut être registre, immédiat ou indirect. Empiler PC signifie SP $\leftarrow$ SP - 4, puis Mem [SP] $\leftarrow$ PC.

RET imm_{16} ; dépiler (PC); SP $\leftarrow$ SP + imm16 octets.

Dépiler PC signifie PC $\leftarrow$ Mem[SP], puis SP $\leftarrow$ SP + 4.

L'appel/retour s'implémente donc par la paire CALL/RET. Chaque appel et chaque retour de sous-programme implique un accès mémoire.

Passage des paramètres

Le passage des paramètres de l'appelante vers l'appelée et le retour d'une valeur peut d'effectuer, soit par les registres du processeur, soit par la mémoire. Dans la plupart des cas, le nombre de paramètres à passer est réduit, et le passage peut se faire simplement par les registres : l'appelante écrit, par exemple le registre R5 ; l'appelée doit savoir que le paramètre se trouve dans R5. La compilation séparée requiert une convention logicielle qui décrit l'association paramètres-registre.

Ce n'est que pour les procédures avec un grand nombre de paramètres que les paramètres en excédent par rapport aux registres disponibles sont passés par la mémoire.

Conventions logicielles

Pour le MIPS, la convention est d'utiliser les registres R4 à R7, dans l'ordre des paramètres. Ceci autorise donc à passer par registre jusqu'à 4 paramètres.

Pour le Power PC, la convention est d'utiliser R3 à R10. Ceci autorise donc à passer par registre jusqu'à 8 paramètres.

Support matériel

L'architecture SPARC offre un support matériel supplémentaire pour le passage des paramètres : le multifenêtrage. Les 32 registres généraux sont spécialisés (fig. 4.17) en 8 registres globaux (notés g0-g7), 8 registres de sortie (notés o0-07), 8 registres locaux (l0-l7) et 8 registres d'entrée (i0-i7).

L'architecture SPARC définit une certain nombre (2 à 8) de fenêtres de 24 registres. Les différentes fenêtres sont recouvrantes, comme décrit dans la fig. 4.18. Une fenêtre n'a en propre que ses registres locaux. Ses registres d'entrée sont les registres de sortie de la fenêtre précédente, et ses registres de sortie sont les registres d'entrée de la fenêtre suivante. D'autre part, les registres globaux sont partagés par toutes les fenètres. Ainsi, chaque procédure voit 32 registres, mais le nombre de registres physiques est plus élevé. Enfin, l'ensemble de fenêtres est géré comme une pile. La commutation de fenètre s'effectue par deux instructions : SAVE et RESTORE. SAVE décrémente le pointeur de pile, RESTORE l'incrémente.

Numéro	Nom
r31	i7 = return ad
r30	i6 = fp
r29	i5
...	...
r24	i0
r23	l7
...	...
r16	l0
r15	07 = tmp
r14	06 = sp
r13	05
...	...
r8	O0
r7	g7
...	...
r1	g1
r0	0

Figure 4.17: Registres Sparc

Le passage de paramètres et la récupération des résultats se fait donc de manière simple, par le recouvrement entre sortie de fenêtre appelante et entrée de fenêtre appelée. Les paramètres sont écrits par l'appelante dans ses registres O0-07. L'appelée trouve ses dans les registres i0-i7, et une fonction retourne son résultat dans le registre O0 de l'appelante. Les paramètres sont écrits par l'appelante dans ses registres de sortie, et la valeur de retour est écrite par l'appelée dans i0.

Il faut bien noter que l'appel de procédure en lui même (CALL ou JMPL) n'effectue aucune commutation de fenètre. C'est donc à la procédure *appelée* de déclencher cette commutation au de but de la procédure. Une procédure non terminale commence par une instruction SAVE et se termine par une instruction RESTORE. Une procédure terminale n'est pas compilée en utilisant le mécanisme de fenêtrage, mais une convention logicielle analogue aux cas précédents.

Enfin, si la pile déborde, la fenètre à écraser est sauvegardée en mémoire avant la commutation, et l'évènement de débordement est mémorisé. Ainsi, lors du dépilement, la fenètre sauvegardée sera restaurée.

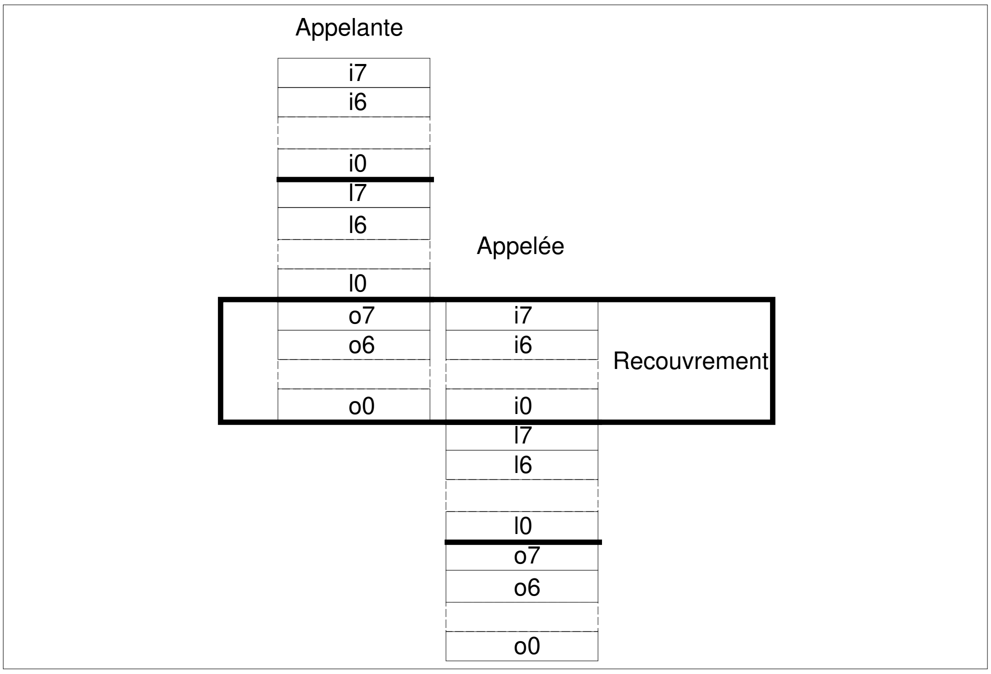

Figure 4.18: Recouvrement des fenetres SPARC

Pile d'exécution

Principe

L'appel de procédure est un mécanisme dynamique, c'est à dire non prévisible à la compilation : lors de la compilation séparée d'une procédure, le compilateur ne peut pas savoir dans quel contexte cette procédure sera appelée. Il est donc très souhaitable que les variables locales de la procédure puissent être adressées de façon indépendante du contexte d'exécution. C'est ce que réalise le mécanisme appelé *pile d'exécution*.

La fig. 4.19 montre le principe du mécanisme de pile d'exécution. Chaque sous-programme dispose d'une zone mémoire correspondant à ses variables locales. Le sommet de la pile est toujours repéré par une registe appelé SP (stack pointer). Le début de la zone des données de la procédure courante est souvent repérée par un registre appelé frame pointer (FP). Les variables sont donc adressées en base + déplacement : SP + délacement positif, ou FP + déplacement négatif.

Le calcul de la taille de cette zone est réalisé à la compilation à partir des déclarations de variables. Le compilateur, au début d'une la procédure, alloue la mémoire nécessaire, en décrémentant SP du nombre d'octets adéquat Dans les machines RISC, SP est un registre général : par exemple, pour l'architecture SPARC, SP = O6.

Le mécanisme de commutation de fenêtre du SPARC offre également un support matériel à la pile d'exécution. D'une part, l'ancien registre SP (O6) devient automa-

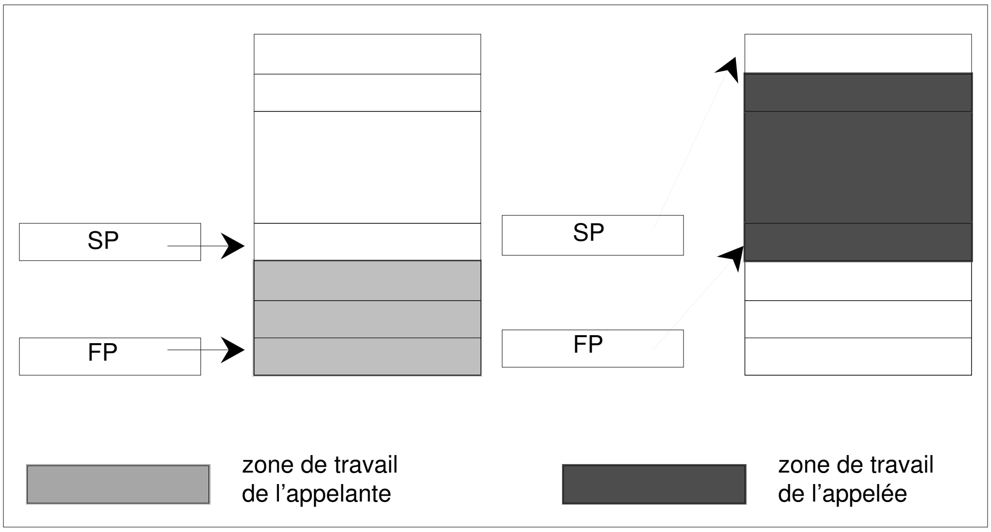

Figure 4.19: Principe de la pile d'exécution

tiquement le nouveau registre FP (i6). D'autre part, les instructions SAVE et RESTORE ont des fonctionnalités plus riches que la simple commutation de fenètres ; leur définition complète est :

SAVE Ra, Rb/Imm_{13}, Rd
    tmp $\leftarrow$Ra + (Rb ou ES (Imm_{13})
    CWP $\leftarrow$ CWP -1
    Rd $\leftarrow$ tmp

RESTORE Ra, Rb/Imm_{13}, Rd
    tmp $\leftarrow$Ra + (Rb ou ES (Imm_{13})
    CWP $\leftarrow$ CWP +1
    Rd $\leftarrow$ tmp

Il faut noter que l'addition se faisant avant la commutation de contexte, ce sont les valeurs des registres de l'ancienne fenêtre qui sont lus. L'écriture du résultat se faisant après la commutation de fenêtre, le registre affecté est celui du nouveau contexte.

Dans l'architecture x86, SP est un registre spécialisé: l'instruction PUSH opérande empile l'opérande et décrémente SP, l'instruction POP destination dépile et incrémente SP.

Sauvegarde du contexte

Le contexte d'exécution d'une procédure décrit l'état du processeur et de la mémoire. Il comprend en particulier :

- les registres généraux ;
- l'adresse de retour dans l'appelante ;
- l'état de ses variables.

Les variables sont sauvegardées par le mécanisme de pile d'exécution.

Sauvegarde des registres généraux

Sauf à être vide, un sous-programme utilise les registres généraux. Ces registres pouvaient contenir à l'appel une valeur qui restera utile apprès le retour du sous-programme. Donc, si celui-ci modifie le registre, il doit le sauvegarder à l'entrée et le restaurer avant de retourner.

Pour limiter le nombre de sauvegardes, il existe en général une convention logicielle, qui partage la sauvegarde entre appelante et appelée. Par exemple, dans le MIPS, les registres R8-R15, R24 et R25 sont en principe réservés à des valeurs temporaires, dont la durée de vie ne s'étend pas au delà des appels de procédure, alors que les registres R16-R23 sont rervés à des valeurs longue durée. La sauvegarde éventuelle des regsitres destinés aux valeurs temporaires est à la charge de l'appelante, celle des registres destinés aux valeurs longue durée est à la charge de l'appelée. On a une convention analogue sur le Power PC.

Dans le cas du SPARC, le mécanisme de fenêtrage limite les sauvegardes nécessaires, puisque chaque procédure dispose de 8 registres privés, les registre l0-l7. En revanche, les registres I0-I7 de l'appelée doivent être sauvegardés s'ils sont utilisés, puisqu'ils sont en fait les registres O0-O7 de l'appelante. La seule exception est le registre I0 lorsque l'appelée est une fonction : la valeur de retour y est placée.

Adresse de retour

L'utilisation d'un registre particulier pour ranger l'adresse de retour n'est suffisant que pour les procédures terminales : les appels successifs écrasent le contenu de l'unique registre de sauvegarde.

Pour pouvoir exécuter correctement des procédures non terminales, il existe deux solutions : soit disposer de plusieurs registres différents pour ranger les adresses de retour ; soit sauvegarder l'adresse de retour en mémoire avant d'appeler une nouvelle procédure et la restituer lors du retour. Quel que soit le nombre de registres disponibles pour ranger les adresses de retour, il est évident que la seconde solution est toujours indispensable lorsque le nombre d'appels successifs de procédures imbriquées dépasse le nombre de registres disponibles.

```
int myfunc (int k) {
return (k+1);
}

void init (int tab [], int i , int j) {
int k ;
for (k = i ; k < j ; k++)
    tab [k] = myfunc (k);
}
void main(void)
{
int tab [100];
init (tab, 2, 10);
printf ("%d", tab [4]);
}
```

Figure 4.20: Exemple d'appels de procédure en C

L'architecture SPARC rend ce passage par la mémoire inutile : l'adresse de retour est sauvegardée dans R15, mais celui-ci est identique à O7 (cf fig. 4.17). Donc, la commutation de fenêtre offre un registre R15 frais pour la sauvegarde, et l'adresse de retour apparaît dans i7 !

Etude de cas

Pour illustrer ce qui précède, on étudie en détail la compilation d'un petit programme (fig. 4.8), qui contient deux appels de sous-programme, l'un non terminal, et l'autre terminal.

Power PC

Le désassemblage du code résultant de la compilation séparée des trois routines C est décrit fig. 4.21

Appel et retour : init et myfunc sont appelés par bl (instructions main_7 et init_13). Tous les retours (main_15 , init_25, myfunc_2) se font par blr.

Un décodage précis des instructions bl montre que le déplacement est 0 : comme les routines sont compilées séparément, l'adresse de branchement n'est pas encore calculée (cf partie Edition de liens).

Passage des paramètres Les instructions 4, 5 et 6 de main chargent respectivement dans R3, R4 et R5 les paramètres de l'appel de init (&tab[0], 2 et 10). Dans init, avant l'appel de myfunc, R3 est positionné avec la valeur courante de k contenue dans R31 (init_12).

myfunc retourne son résultat dans R3, qui est utilisé comme argument du rangement (init_15).

Le paramètre tab est passé par référence : la routine init ne connaît que R3, qui est @tab[0]. Les paramètres entiers sont passés par valeur.

Pile d'exécution Les instruction STWU SP, -n(SP) assurent en même temps l'allocation mémoire de la zone de travail de la procédure, et la sauvegarde du pointeur de pile. En effet, STWU est un rangement en mode d'adressage post-incrémenté. La variable tab du main est adressée en mode base + déplacement (main_4). Comme tab est passé par référence, la procédure init modifie l'état de la pile d'exécution de l'appelant (init_15). Les instructions main_13 et init_20 désallouent la zone de la procédure qui se termine en modifiant SP. La fonction myfunc, qui est terminale, ne déclare pas de variable, et n'a rien à sauvegarder, n'alloue pas de zone de travail.

Sauvegarde et restauration du contexte L'adresse de retour est sauvegardée en deux étapes : mflr (Move From Link Register) transfère le contenu de LR vers R0, qui n'est pas câblé à 0. Ensuite, une instruction de rangement sauvegarde l'adresse de retour en mémoire, d'ailleurs dans la zone de travail de l'appelante. La restauration est en miroir. Le transfert entre LR et mémoire doit se faire par l'intermédiaire d'un registre général, car il n'existe pas d'instruction de transfert entre LR et la mémoire. Une procédure terminale se dispense de cette sauvegarde, qui lui serait inutile.

Le pointeur de pile SP est sauvegardé en même temps que la zone de travail est allouée.

Le procédure init a besoin des registres de travail R29, 30 et 31. Elle les sauvegarde donc avant de les modifier, puis les restaure à la fin. la procédure main, qui n'effectue aucun calcul, ne sauvegarde aucun regsitre de travail.

SPARC

Le désassemblage du code résultant de la compilation séparée des trois routines C est décrit fig. 4.22

L'instruction jump adresse est une instruction synthétique (ie plus lisible) pour l'instruction jmpl adresse R0. L'instruction synthétique ret équivaut à JMPL R31, 8,R0. De même, l'instruction synthétique RESTORE équivaut à RESTORE R0, R0, R0, c'est à dire à une commutation retour de fenêtre.

Le code SPARC présente une particularité, qui sera expliquée dans le chapitre pipeline : les branchements retardés. Toute séquence

Instruction de branchement
Instruction A
Instruction B

a la sémantique séquentielle :

Instruction A
Instruction de branchement
Instruction B

L'instruction qui suit syntaxiquement le branchement est exécutée avant que le branchement prenne effet.
Appel et retour Tous les appels utilisent l'instruction CALL. Comme pour le code Power PC, les adresses ne sont pas résolues.

Pour le retour, la fonction myinit est terminale ; il n'y a pas de commutation de fenêtre, donc myinit partage la fenêtre de l'appelante, et trouve l'adresse de retour dans R15 (o7). Les autres fonctions commencent par une commutation, et trouvent donc l'adresse de retour dans R31.
Passage de paramètres La procédure main positionne o0, o1 et o2, que la procédure init trouve dans i0, i1 et i2. En revanche, myfunc, au lieu de chercher son paramètre d'entrée et retourner son résultat dans i0, utilise o0. Ainsi, pour l'appelante, le fait que myfunc soit terminale ou non na pas besoin d'être connu, ce qui est indispensable pour la compilation séparée : init passe son paramètre et retrouve son résultat dans o0.

Edition de liens et chargement d'un programme

On se limite dans cette section à un contexte UNIX.

Introduction

Le passage d'un programme écrit en LHN vers un programme qui s'exécute comprend trois étapes.

- Compilation : d'un fichier LHN vers un fichier relogeable. Par exemple, les fichiers main.o, init.o, myfunc.o sont produits par la commande

  gcc -O3 -c main.c init.c myfunc.c

- Edition de lien : d'un ensemble de fichiers objets vers un fichier exécutable (ld comme Link eDitor). Par exemple, l'exécutable prog est produit par

  ld -o prog main.o init.o myfunc.o

- Chargement :
  - Copie de l'exécutable en mémoire : les sections du fichier exécutable correspondant au code sont placées dans des pages mémoire avec droits de lecture et d'exécution, les sections correspondant aux données dans des pages avec droit de lecture/écriture.
  - Passage des arguments de la ligne de commande (= paramètres du main) si nécessaire.
  - Positionnement de PC et du pointeur de pile.

La différence qualitative entre un relogeable et un exécutable est le traitement des références absolues (variables non locales, adresse des instructions etc.) et des variable et procédures non visibles en compilation séparée. Dans un relogeable, seules les références relatives et visibles sont résolues : branchements relatifs, variables locales. Les références absolues ou inconnues sont laissées en attente. Dans un exécutable, instructions et données ont toutes une adresse et toutes les références sont résolues.

La fig. 4.23 montre le désassemblage de l'exécutable prog. Les adresses des instructions sont fixées (première colonne) ; les sous-programmes sont dans un même référentiel mémoire. A partir de cette information, l'éditeur de liens peut calculer les adresses de branchement pour les appels de sous-programme call : la première instruction call est 7f ff ff fA, soit un branchement à PC - (6 inst), c'est à dire myfunc ; la deuxième instruction call est 7f ff ff f0, soit un branchement à PC - (16 inst), c'est à dire init.

Format d'un fichier objet

Sous Unix, le format actuel est le format ELF (Executable and Linking Format), qui unifie les trois types d'objet [12]. Comme les fichiers objet participent à l'édition de lien (construction d'un programme) et à l'exécution d'un programme, ce format fournit deux vues parallèles du contenu d'un fichier (fig. 4.24).

L'en-tête (header), décrit l'organisation du fichier (emplacements des sections/segments) et les informations globales sur le type de fichier objet (relogeable, exécutable, shared, core).

Les sections sont contigues et disjointes, éventuellement vides. Chacune d'entre elles contient un certain type d'information sur le programme. La table 4.6 présente les sections les plus fréquentes ; le format en prévoit d'autre, ainsi que la possibilité d'en définir.

Nom	Description
.bss	Données non intialisées
.data	Données initialisées
.fini	Instructions exécutées après la fin du programme utilisateur
.init	Instructions exécutées avant l'entrée dans le programme utilisateur
.rel<name>	Informations de relocation pour la section name
.symtab	Table des symboles (voir ci-dessous)
.strtab	Table de chaînes de caractères
.text	Les instructions du programme utilisateur

Table 4.6: Quelques sections d'un programme objet

La table des symboles contient une entrée pour toutes les définitions et références mémoire symboliques. Chaque entrée contient cinq champs :

- st_name : un index dans la strtab ; l'entrée correspondante contient une chaine de caractère qui décrit le symbole.

- st_value : essentiellement, l'adresse qui contient la valeur du symbole. Dans un relogeable, c'est le déplacement par rapport au début de la section. Dans un exécutable ou un shared, c'est l'adresse mémoire.
- st_size : la taille du symbole en octets
- st_info : la portée du symbole, à l'édition de lien. Un symbole local n'est visible qu'à l'intérieur du fichier qui le définit ; des symboles locaux de même nom peuvent exister dans plusieurs fichiers sans conflit. Un symbole global est visible de tous les fichiers à l'édition de lien, et peut ainsi satifaire les références encore non définies d'un autre fichier. un symbole faible (weak) est intermédiaire : il se comporte comme un symbole global, mais un symbole global de même nom aura la précédence.

La relocation consiste à mettre en relation des références symboliques avec leurs définitions. Par exemple, la section de relocation de init.o contient les champs suivants :

Offset	Symndx	Type	Addend
0x10	myfunc	R_SPARC_WDISP30	0

qui signifient que le symbole myfunc, qui est référencé à l'adresse 0x10 (cf fig 4.22) est un déplacement destiné à un call pour l'architecture SPARC. Pour les fichiers relogeables, cette information est exploitée à l'édition de liens, pour transformer les références symboliques en adresses mémoire, pas nécessairement absolues. Par exemple, l'éditeurs de liens résoudra la référence myfunc en la mettant en relation avec sa définition dans le fichier myfunc.o, et la relogera ainsi. La relocation compre,d donc deux étapes :

- le parcours de la tables des symboles des fichiers liés permet de résoudre les symboles non définis (U) ;
- les références mémoires symboliques sont évaluées grâce à la section de relocation.

Librairies partagées

Outre les fichiers relogeables et exécutables, il existe un troisième type de fichier binaire décrivant un programme : les fichiers objets partagés. Ils contiennent un code et des données utilisables à la fois à l'édition de lien et à l'exécution

Les fichiers relogeables peuvent être regroupés sous forme de librairies, et liés statiquement à d'autres fichiers relogeables pour former un exécutable. Dans ce cas, une copie des fichiers qui satisfont les références non encore résolues est incorporée dans l'exécutable. Tout se passe comme si tous les fichiers avaient été compilés simultanément. La taille de l'exécutable est donc la somme de la taille des composantes, et l'exécutable doit être relinké si l'archive est modifiée.

Une solution intermédiaire entre l'exécutable et le relogeoble est la librairie partagée et la liaison dynamique (dynamic linking). Les fonctions et procedures utiles sont compilées et regroupées en un objet partagé aussi appelé librairie dynamique. L'objet partagé a les caractéristiques d'un exécutable, en particulier toutes les références internes sont résolues et les références résolues sont relogées. L'objet partagé est inclus dans l'espace mémoire du programme à l'exécution, et c'est cette opération qui est appelée liaison dynamique. C'est la fonction d'une édition de lien, mais à l'excéution : les références externes du programme qui font appel à la librairie partagée sont résolues et relogées à l'exécution.

Il y a deux avantages à la liaison dynamique.

- La possibilité de faire évoluer le code sans relinker ; la plupart des librairies standard sont maintenant partagée, et la liaison dynamique est le comportement par défaut lorsqu'il existe une version statique et une version dynamique d'un librairie.
- L'économie d'encombrement disque, et la possibilité de partager le code, y compris en mémoire, d'où économie d'encombrement mémoire. La librairie partagée n'est pas copiée dans l'exécutable à l'édition de lien. Seules quelques informations de résolution sont enregistrée. Ainsi, le a.out de chacun ne contient pas une copie du code de printf ! En outre, l'objet partagé, s'il ne requiert pas de modification, peut être partagé entre plusieurs utilisateurs en mémoire.

Que se passe-t-il à l'édition de lien dynamique ? Essentiellement, le code partagé est inclus (mappé) dans l'espace mémoire de l'exécutable qui l'appelle. Les références externes de l'exécutable sont résolues et relogées. Bien évidemment, une librairie partagée peut en appeler une autre, l'image mémoire du programme se construit donc progressivement et non pas en une seule étape comme au chargement d'un exécutable construit par lien statique.

Un objet partagé est typiquement compilé pour que le code ne dépende pas de sa position. En particulier, il est souhaitable qu'aucune adresse absolue n'y figure, donc que les branchements soient relatifs. L'objet partagé peut alors apparaître à des adresses différentes vu de différents processus, tout en n'existant qu'à un exmplaire en mémoire. En revanche, si le code contient des adresses absolues, il peut être utilisé en édition de lien dynamique, mais perd l'avantage du partage. Chaque processus recoit alors une copie privée du code, où l'édition de lien dynamique aura relogé les adresses abolues à la volée, c'est à dire modifié le code d'une façon dépendante de chaque processus exécutant. Il en va de même si la librairie partagée contient des données adressées en absolu.

Quelques utilitaires

On peut visualiser de façon détaillée le code produit par un compilateur et/ou un éditeur de liens grâce à la commande `dis` :

`dis fichier` : désassemble la section .text

avec ad mémoire, code hexadécimal et assembleur.

On peut visualiser l'ensemble des sections d'un ELF avec la commande `dump`. Les options précisent les sections et le formattage, en particulier -r pour les informations de relocation.

On peut visualiser la table des symboles avec la commande `nm`

```
; myfunc
 1  00000000 :  38630001  addi  r3,r3,1         ; R3 ← k + 1
 2  00000004 :  4E800020  blr                   ; retour
init
 1  00000000 :  7C0802A6  mflr  r0              ; debut sauvegarde adresse retour
 2  00000004 :  93E1FFFC  stw   r31,-4(SP)      ; sauvegarde contexte registre
 3  00000008 :  93C1FFF8  stw   r30,-8(SP)
 4  0000000C :  93A1FFF4  stw   r29,-12(SP)
 5  00000010 :  90010008  stw   r0,8(SP)        ; fin sauvegarde adresse retour
 6  00000014 :  9421FFB0  stwu  SP,-80(SP)      ; allocation pile d'exécution
 7  00000018 :  7C7D1B78  mr    r29,r3          ; copie R3 dans registre de travail
 8  0000001C :  9081006C  stw   r4,108(SP)      ; sauvegarde deuxième paramètre
 9  00000020 :  7CBE2B78  mr    r30,r5          ; copie 3eme parametre dans R30
10  00000024 :  83E1006C  lwz   r31,108(SP)     ; copie 2eme paramètre dans R31
11  00000028 :  48000018  b     *+24            ; 00000040 ; sauter au test de boucle
12  0000002C :  7FE3FB78  mr    r3,r31          ; passage du paramètre k à myfunc
13  00000030 :  48000001  bl    .myfunc         ; appel myfunc
14  00000034 :  57E4103A  slwi  r4,r31,2        ; R4 ← R31*4 (shift left)
15  00000038 :  7C7D212E  stwx  r3,r29,r4       ; tab [k] ← R3
16  0000003C :  3BFF0001  addi  r31,r31,1       ; k ← k+1
17  00000040 :  7C1FF000  cmpw  r31,r30         ; test sortie boucle
18  00000044 :  4180FFE8  blt   *-24            ; branchement à inst 12 0000002C
19  00000048 :  80010058  lwz   r0,88(SP)       ; debut restauration adresse retour
20  0000004C :  38210050  addi  SP,SP,80        ; désallocation de la zone de travail
21  00000050 :  7C0803A6  mtlr  r0              ; fin restauration adresse retour
22  00000054 :  83E1FFFC  lwz   r31,-4(SP)      ; restauration du contexte registres
23  00000058 :  83C1FFF8  lwz   r30,-8(SP)
24  0000005C :  83A1FFF4  lwz   r29,-12(SP)
25  00000060 :  4E800020  blr                   ; retour
; main
 1  00000000 :  7C0802A6  mflr  r0
 2  00000004 :  90010008  stw   r0,8(SP)
 3  00000008 :  9421FE30  stwu  SP,-464(SP)
 4  0000000C :  38610038  addi  r3,SP,56        ; R3 ← @tab[0]
 5  00000010 :  38800002  li    r4,2            ; R4 ← premier param.
 6  00000014 :  38A0000A  li    r5,10           ; R5 ← premier param.
 7  00000018 :  48000001  bl    .init           appel init
 8  0000001C :  38620000  addi  r3,RTOC,@643
 9  00000020 :  80810048  lwz   r4,72(SP)
10  00000024 :  48000001  bl    .printf
11  00000028 :  0000000   nop
12  0000002C :  800101D8  lwz   r0,472(SP)
13  00000030 :  382101D0  addi  SP,SP,464
14  00000034 :  7C0803A6  mtlr  r0
15  00000038 :  4E800020  blr
```

Figure 4.21: Code PPC des sous-programmes exemples. La première colonne est le numéro de l'instruction ; la deuxième son déplacement par rapport au début du sous-programme ; la troisième le code hexadécimal de l'instruction ; la quatrième le code en langage d'assemblage.

myfunc()				
0:	81 c3 e0 08	jmp	%o7 + 8	; retour
4:	90 02 20 01	add	%o0, 1, %o0	; calcul de k+1
init()				
0:	9d e3 bf 90	save	%sp, -112, %sp	; commutation fenêtre
4:	80 a6 40 1a	cmp	%i1, %i2	; test i $\geq j$ j
8:	16 80 00 09	bge	0x2c	;
c:	01 00 00 00	nop		;
10:	40 00 00 00	call	0x10	; appel myfunc
14:	90 10 00 19	mov	%i1, %o0	; passage de k à myfunc
18:	93 2e 60 02	sll	%i1, 2, %o1	; o1 $\leftarrow$ k*4
1c:	b2 06 60 01	add	%i1, 1,%i1	; k $\leftarrow$ k+1
20:	80 a6 40 1a	cmp	%i1, %i2	; test sortie de boucle
24:	06 bf ff fb	bl	0x10	; branchement début de boucle
28:	d0 26 00 09	st	%o0, [%i0 + %o1]	; tab[k] $\leftarrow$ o0
2c:	81 c7 e0 08	ret		; branchement retour
30:	81 e8 00 00	restore		; commutation fenêtre
main()				
0:	9d e3 be 00	save	%sp, -512, %sp	
4:	90 07 be 60	add	%fp, -416, %o0	
8:	92 10 20 02	mov	2,%o1	
c:	40 00 00 00	call	0xc	
10:	94 10 20 0a	mov	10, %o2	
14:	d2 07 be 70	ld	[%fp - 400], %o1	
18:	11 00 00 00	sethi	%hi(0x0), %o0	
1c:	40 00 00 00	call	0x1c	
20:	90 12 20 00	or	%o0, 0, %o0	
24:	81 c7 e0 08	ret		
28:	81 e8 00 00	restore		

Figure 4.22: Code SPARC des sous-programmes exemples. Compilé avec gcc, options -03 -S -c. La première colonne est le déplacement par rapport au début du sous-programme ; la deuxième le code hexadécimal de l'instruction ; la troisième le code en langage d'assemblage.

```
myfunc
10800 :  81 c3 e0 08   jmp      %o7 + 8
10804 :  90 02 20 01   add      %o0, 1, %o0
init
10808 :  9d e3 bf 90   save     %sp, -112, %sp
1080c :  80 a6 40 1a   cmp      %i1, %i2
10810 :  16 80 00 09   bge      0x10834
10814 :  01 00 00 00   nop
10818 :  7f ff ff fa   call     gcc2_compiled.
1081c :  90 10 00 19   mov      %i1, %o0
10820 :  93 2e 60 02   sll      %i1, 2, %o1
10824 :  b2 06 60 01   add      %i1, 1, %i1
10828 :  80 a6 40 1a   cmp      %i1, %i2
1082c :  06 bf ff fb   bl       0x10818
10830 :  d0 26 00 09   st       %o0, [%i0 + %o1]
10834 :  81 c7 e0 08   ret
10838 :  81 e8 00 00   restore
main
1083c :  9d e3 be 00   save     %sp, -512, %sp
10840 :  90 07 be 60   add      %fp, -416, %o0
10844 :  92 10 20 02   mov      2, %o1
10848 :  7f ff ff f0   call     gcc2_compiled.
1084c :  94 10 20 0a   mov      10, %o2
10850 :  d2 07 be 70   ld       [%fp - 400], %o1
10854 :  11 00 00 44   sethi    %hi(0x11000), %o0
10858 :  40 00 43 5b   call     printf@@SYSVABI_1.3
1085c :  90 12 23 a0   or       %o0, 928, %o0
10860 :  81 c7 e0 08   ret
10864 :  81 e8 00 00   restore
```

Figure 4.23: Code exécutable SPARC.

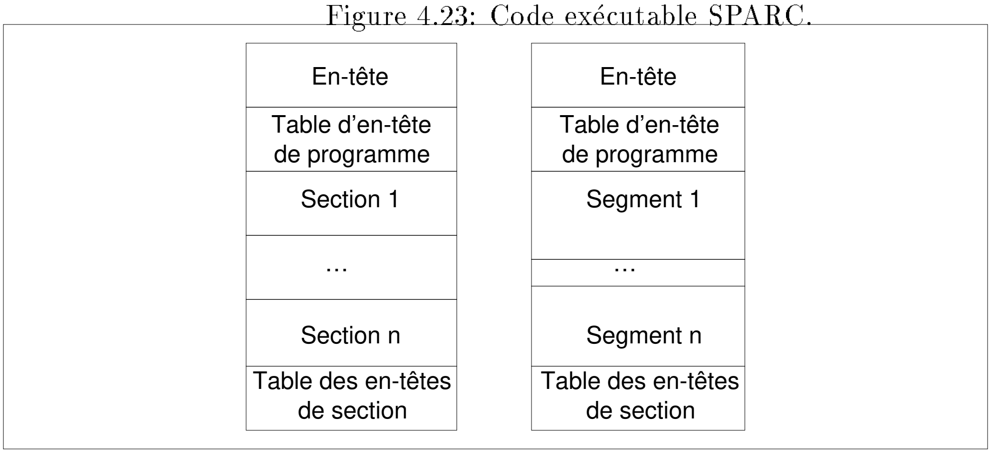

Figure 4.24: Format d'un fichier ELF

Bibliographie

[1] D. Patterson, J. Hennessy. *Organisation et conception des ordinateurs.* Dunod 94 Traduction de *Computer Organization and Design : The Harware/Software Interface*, Morgan-Kauffman 94.

[2] J. Hennessy, D. Patterson. *Architecture des ordinateurs : une approche quantitative, Deuxième édition.* Thompson Publishing France, 96. Traduction de *Computer Architecture. A Quantitative Approach.* McGrawHill 96

[3] D. Etiemble. *Architecture des microprocesseurs RISC.* Dunod, collection A2I, 92

[4] *IEEE Transactions on Computers.* Octobre 96.

[5] D.Patterson et al. A case for intelligent RAM : IRAM. *IEEE Micro.* Avril 97.

[6] V. Cuppu et al. A Performance comparison of contemporary DRAM architectures. *Proc. 26th ISCA.* Mai 99.

[7] http://www.asc-inc.com/ascii.html

[8] http://unicode.org

[9] *ARIANE 5 Flight 501 Failure, Report by the Inquiry Board.* http://www.esrin.esa.it/htdocs/tidc/Press/Press96/ariane5rep.html

[10] D. Etiemble. *Fondements du matériel.* Polycopié de Licence d'Informatique.

[11] D. Goldberg. What every computer scientist should know about floating point arithmetics. *ACM Computing Surveys*, 23:1, pp 5-48, Mars 91.

[12] Mary Lou Nohr. *Understanding ELF Objetcs Files and Debugging Tools.* Prentice Hall, 94.