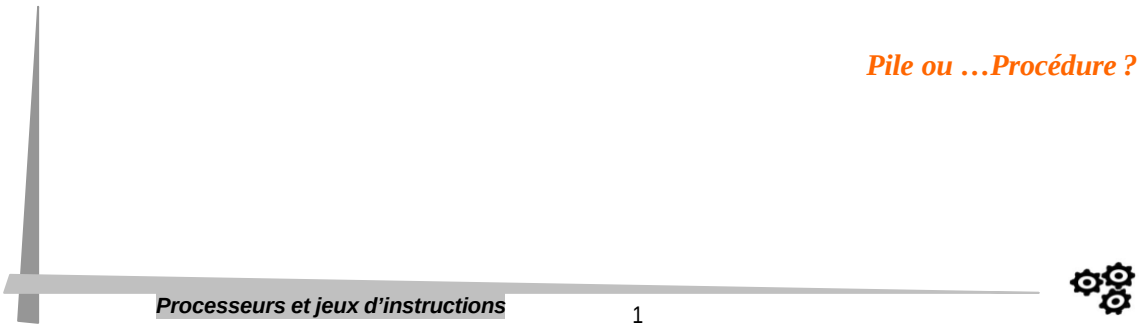
4 CHAPITRE 4

Processeurs
et
Jeux d'Instructions

De l'instruction

Au programme

Pile ou …Procédure ?



L'objectif de ce chapitre est de réaliser le passage du matériel au logiciel, permettant ainsi de faire évoluer le petit automate qu'est le processeur vers un ordinateur d'usage général. Ce passage s'appuie sur le modèle de programmation du processeur qui en est la vue offerte au programmeur : les registres, le jeu d'instructions et le modèle mémoire.

Le modèle de programmation est l'ossature pour la définition du premier niveau de langage de programmation d'un processeur, c'est-à-dire le langage assembleur.

Nous avons montré, dans les chapitres précédents, comment il est possible de construire un processeur à partir de briques élémentaires, voyons-en maintenant le fonctionnement avant d'aborder son utilisation au travers du logiciel et en premier lieu avec le langage assembleur. Quel processeur choisir pour en expliquer les principes de fonctionnement ?

Notre premier exemple est le Z80 (de Zilog), processeur 8 bits, compatible ascendant du 8080 d'Intel. Même si ce processeur est de conception ancienne, il est toutefois toujours en service et il se prête bien à un premier exercice pédagogique pour décrire l'exécution matérielle d'une instruction.

La deuxième partie du chapitre est une introduction au jeu d'instructions d'un processeur et à un langage de programmation assembleur. Le processeur 680x0 de Motorola est pris comme exemple pour son bon compromis entre simplicité et lisibilité. La programmation fait rapidement apparaître le besoin de structuration et de réutilisation que tous les langages modernes de haut niveau intègrent. Ce besoin est concrétisé par la notion de procédure.

La dernière partie est dédiée à la procédure et en particulier à la relation entre une procédure (ou fonction) dans un langage de haut niveau (ici le langage C) et son implémentation au niveau d'un langage assembleur. Seront alors illustrées toutes les implications de l'utilisation d'une procédure aussi bien au niveau de la gestion d'une pile que des techniques de passages de paramètres. Les illustrations porteront sur des processeurs d'architectures bien différentes, aussi bien CISC (Complex Instruction Set Computer), comme le 680x0 et le Pentium, que RISC (Reduced Instruction Set Computer) comme le MIPS, le SPARC et le PowerPc.

4.1 Le modèle de programmation d'un processeur.

Le but fixé est d'abandonner progressivement les aspects physiques (architecture interne, circuits logiques) d'un processeur au profit des caractéristiques d'une machine plus abstraite destinée à supporter l'exécution des programmes de l'utilisateur. Cet utilisateur lancera une application –traitement de texte, courrier électronique, navigateur…–, en sélectionnant la bonne icône sur une interface graphique. L'application résout le problème réel de l'utilisateur, par exemple récupérer son courrier, en chargeant et en faisant exécuter un programme écrit généralement dans un langage dit évolué ou de haut niveau. Ainsi, Java, Fortran, C, C++, ou autres, sont des langages conçus pour faire abstraction du processeur physique de la machine qui fait tourner l'application. Le programmeur écrit des programmes avec des instructions virtuelles (celles du langage et pas celles du processeur) et sur des variables abstraites au lieu des registres du processeur ou des cases mémoires. Mais en fin de compte, toute application, tout programme de haut niveau est traduit en programme en *langage assembleur* qui est exécuté en s'appuyant directement sur les caractéristiques matérielles du processeur : le jeu d'instruction et les registres. Le langage assembleur est un langage de bas niveau, c'est-à-dire le plus proche des instructions du processeur. Dans la suite, quand nous parlerons de programmeur, nous nous le mettrons dans le contexte d'une programmation en langage assembleur.

Le ***modèle de programmation*** est l'interface matériel/logiciel qui permet cette transition. Aussi appelé ***Instruction Set Architecture (ISA)***, il repose sur trois éléments principaux :

- les **registres** : le processeur a besoin de mémoires internes pour stocker les données sur lesquelles il va travailler. Ces mémoire internes ou registres sont les éléments que le programmeur doit connaître et manipuler pour programmer l'exécution d'un algorithme. TOUTES les données d'un programme passeront par les quelques registres du processeur. Les registres peuvent être de deux natures : généraux et dédiés. Le Z80 a des registres dédiés, c'est-à-dire réservé à des fonctions spécifiques, tandis que le MIPS possède des registres généraux banalisés. Le 680x0 est dans une situation intermédiaire : il a des registres généraux de données et des registres généraux d'adresses.
- le **jeu d'instructions** : le processeur est accessible à la programmation par l'intermédiaire de son jeu d'instructions c'est-à-dire l'ensemble des instructions qu'il est capable d'exécuter. Celles-ci correspondent en général à des opérations vraiment élémentaires comme nous avons pu en présenter lors de la description d'une unité arithmétique et logique. Développer un programme consiste à traduire un algorithme avec un langage de programmation (ici l'assembleur) dont les verbes sont pris dans le jeu d'instruction et dont les substantifs sont les variables ou constantes mappés en mémoire ou dans les registres. Une instruction comporte souvent deux parties : le code opératoire qui définit le type

d'opération à réaliser et les paramètres (optionnels) qui définissent les opérandes sur lesquels porte l'opération.

- le **modèle mémoire**. Il s'agit ici de la relation entre le contenu d'un registre et le contenu en mémoire à une adresse donnée. L'adressage de la mémoire se fait en général sur la base de l'octet. Dans le cas d'un processeur 8 bits, les registres de données sont de 8 bits. Pour faire un chargement de la mémoire vers un de ces registres, il suffit de donner l'adresse de l'octet en mémoire. Par contre si l'on travaille sur une données occupant plus d'un octet en mémoire, par exemple un entier codé sur 16 bits ou une adresse, la variable occupe deux octets en mémoire centrale et deux possibilités se présentent. Dans l'une, l'adresse en mémoire est l'adresse de l'octet de poids faible (convention *little endian* comme le Z80 ou les processeurs x86 d'Intel), dans l'autre, l'adresse en mémoire est l'adresse de l'octet de poids fort (convention *big endian*, comme par exemple le processeur 680x0).

Avant d'aborder les détails d'un modèle de programmation, il est utile de décrire, dans un cas simple, le fonctionnement d'un processeur. Le processeur est vu comme un automate exécutant un programme sous la forme d'un enchaînement d'instructions. Chaque instruction est elle-même décomposée en phases plus élémentaires impliquant le cas échéant des cycles mémoires. Prenons l'exemple du Z80.

4.2 Le fonctionnement du Z80.

La figure 4-1 donne l'architecture générale et simplifiée du Z80 : il y a des registres organisés autour du bus interne de données et du bus interne d'adresses, deux unités arithmétiques et logiques (une pour les opérations sur les données, l'autre pour du calcul d'adresse), un décodeur d'instruction et un séquenceur (appelé aussi unité de contrôle ou unité de commande).

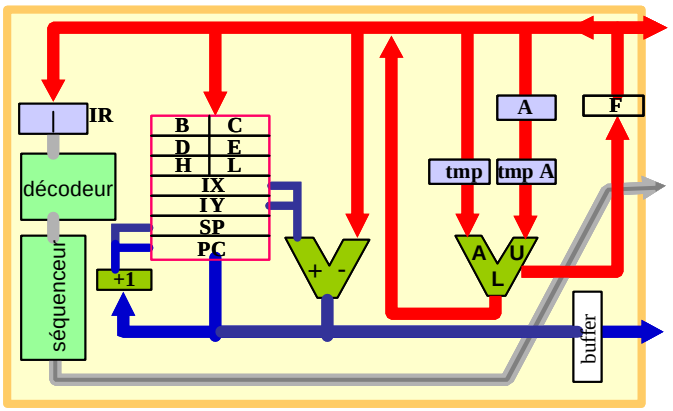


Figure 4–1 Architecture générale du processeur Z80.

4.2.1 Déroulement d'une instruction.

Le bus de données a une largeur de 8 bits et est bidirectionnel : c'est le même chemin qui est utilisé pour les transferts en lecture ou en écriture. Les deux opérations ne s'effectuant jamais en même temps, il n'y a pas de conflits sur le bus. Le bus d'adresse a une largeur de 16 bits ce qui confère au processeur une capacité maximale d'adressage de mémoire centrale de 64 Ko (65536 octets, 1K = 2^{10} = 1024). Le bus d'adresse est unidirectionnel : seul le processeur peut déposer une adresse sur ce bus et le tampon (*buffer*) en sortie de processeur permet de maintenir une adresse vers l'extérieur tout en déterminant une nouvelle adresse en interne.

L'architecture présentée est simplifiée, c'est-à-dire réduite aux éléments minimaux nécessaires à la compréhension du fonctionnement du processeur. Cela concerne en particulier les registres. Le Z80 possède 8 registres 8 bits : A, F, B, C, D, E, H et L. Le registre A (Accumulateur) est le registre privilégié pour les opérations arithmétiques et logiques : toutes les opérations se font par 'accumulation' sur ce registres. F est le registre des indicateurs d'opérations (*flags* ou *drapeaux*) permettant de mémoriser de manière synthétique (1 seul bit) un résultat positif, négatif, nul, débordement, … et ainsi servir de condition pour les instructions de saut conditionnel. L'ensemble de ces registres est complété par des registres dits alternés A' à L' permettant de permuter très rapidement, si nécessaire, les alternés avec les primaires A à L.

Les registres (B, C), (D, E) et (H, L) peuvent être appariés pour réaliser quelques opérations (addition, soustraction, incrémentation, décrémentation) sur les entiers de 16 bits, ils sont alors dénommés BC, DE et HL. La paire HL joue un rôle particulier comme pointeur (adresse) de données en mémoire centrale.

Il y a quatre registres strictement 16 bits et qui servent exclusivement à l'adressage. IX et IY sont des registres d'index qui correspondent aux indices de données structurées (tableaux) dans les langages de haut niveau. Ils servent de déplacement

La naissance du Z80 est étroitement liée à celle de son aîné, le 8080 d'Intel. Federico Faggin, après son doctorat de physique à Padoue en Italie, est embauché en 1968 par Fairchild pour développer les premiers circuits en technologie MOS. Il participe ensuite, avec Tedd Hoff chez Intel, à la conception du 4004, et devient l'architecte du 8080, sorti en 1974.

En 1975, il fonde, avec Masatoshi Shima (ancien de Busicom, la société japonaise ayant commandé les circuits à l'origine du 4004), la société Zilog avec l'idée de réaliser une version améliorée du 8080, ce qu'il n'avait pu faire chez Intel. Le Z80 doit être compatible ascendant en exécutant toutes les 78 instructions du 8080 de la même manière que ce dernier. L'amélioration portera sur un jeu d'instructions plus étendu (120 instructions de plus), plus de registres, une gestion intégrée de la mémoire (rafraîchissement dynamique), une amorce de pipeline et fonctionne à une fréquence de 2,5 MHz. Il est mis sur le marché en juillet 1976 : une vraie réussite technique et commerciale, il est toujours sur le marché et coûte ... 1 €.

(*offset*) par rapport à un pointeur (HL) et en accélèrent l'accès. SP, *Stack Pointer*, est le *pointeur de pile* particulièrement utile pour la gestion des appels de procédures. SP permet, avec des deux instructions spécifiques *Push* et *Pop*, –*empiler* et *dépiler*–, de gérer une zone de mémoire centrale comme *pile*, c'est-à-dire une file d'attente de type LIFO, (Last In-First Out, dernier arrivé premier servi). Le registre le plus important, pour l'explication du fonctionnement du processeur, donc le déroulement d'un programme quel qu'il soit, est PC (*Program Counter*), le compteur de programme.

4.2.1.1 Le déroulement de l'exécution d'une instruction.

Le Z80 suit l'architecture Von Neumann : le programme (code) et les données sur lequel ce programme travaille se trouvent dans la même mémoire et sont donc accessibles via le même chemin, le bus.

Programmes et données se trouvent donc initialement en mémoire centrale. Physiquement, une instruction est une suite d'octets (comprenant le code opératoire et les éventuels opérandes) qui pour être exécutée par le processeur doit être repérée par une adresse en mémoire. Avant l'exécution d'une l'instruction, le processeur doit donc connaître cette adresse : c'est le registre PC qui la contient. D'un point de vue automate, le contenu du registre PC définit les conditions initiales pour l'ordonnancement de l'exécution de l'instruction par le processeur.

Pour le Z80, le déroulement d'une instruction se fait en deux étapes cadencées par l'horloge du processeur.

La première étape appelée « **fetch** » consiste à '*aller chercher*' en mémoire centrale le code opératoire (1 octet) de l'instruction à l'adresse spécifiée par le registre PC, puis le stocker dans un registre particulier appelé *registre d'instruction*. Nous n'avons pas présenté ce registre avec les autres, car il ne fait pas partie du modèle de programmation du Z80 dans la mesure où il n'est pas accessible au programmeur. Pour toutes les instructions, la phase fetch se déroule sur trois coups d'horloge et correspond à un cycle de lecture en mémoire (cf. chapitre 3, section 3.6.2.2).

La seconde étape est celle de l'exécution qui s'écoule sur un nombre variable de coups d'horloge en fonction principalement des opérandes à lire ou à écrire en mémoire. L'instruction est décodée, puis opérée.

Pour illustrer le déroulement d'une instruction, nous prenons l'exemple de l'instruction ADD A, [HL] dont l'objectif est d'ajouter au contenu du registre A (qui vaut b5h) une variable en mémoire à l'adresse 2323h dont la valeur est 08h. L'instruction elle-même est à l'adresse 5000h.

La notation de l'instruction est une notation orientée langage de programmation, c'est-à-dire avec une syntaxe de l'instruction basée sur des mnémoniques. L'instruction a deux paramètres (A et HL) de description des opérandes, mais comme il s'agit de registres, il faut peu de bits pour les coder et ceux-ci peuvent être intégrés directement au code opératoire. L'instruction tient de la sorte sur un octet et son code est 86h.

Avant l'exécution de l'instruction, soit à la fin de l'instruction précédente, le compteur ordinal PC contient la valeur 5000h, adresse à laquelle il faut aller chercher le code opératoire de notre instruction.

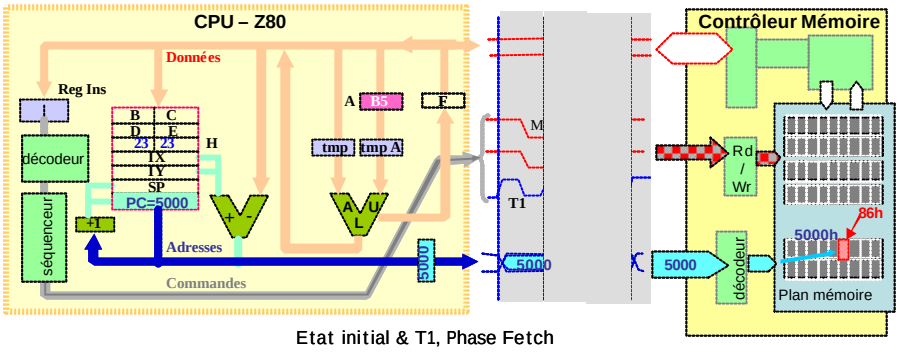


Figure 4–2 Etat initial et T1 phase fetch.

Au front montant de la période d'horloge T1, le processeur dépose le contenu 5000h sur le bus d'adresse, valeur qui est mémorisée temporairement dans un tampon. La valeur de l'adresse se propage sur le bus d'adresse jusqu'au décodeur du contrôleur de mémoire. Au front descendant de la demie période les valeurs sont stabilisées sur le bus et le processeur active l'ordre de lecture vers le contrôleur de mémoire en positionnant au niveau bas le signal RD* et le signal de requête mémoire MemReq*.

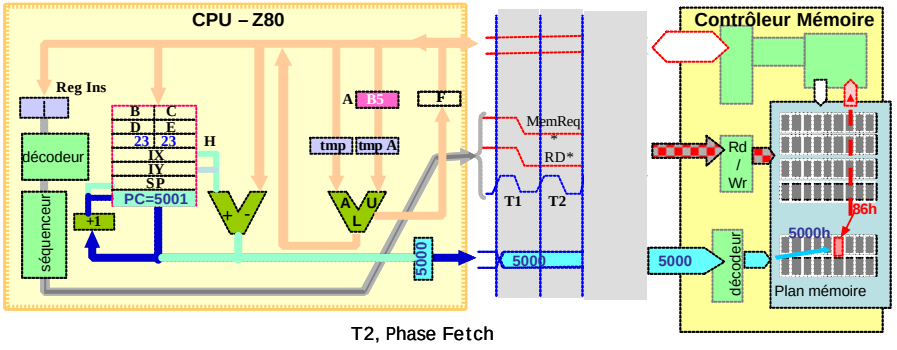


Figure 4–3 T2 phase fetch.

Le contrôleur de mémoire peut maintenant activer le processus de lecture et il a toute la période T2 pour récupérer la donnée (ici le code d'une instruction).

En fin de période, le contrôleur doit être en mesure de produire la donnée lue à l'adresse 5000h. On notera le point intéressant au niveau du processeur : pendant cette période d'inactivité pour lui, il prépare l'avancement du programme en incrément le compteur de programme qui pointe maintenant et jusqu'à la fin de l'instruction, sur l'adresse de la prochaine instruction à exécuter qui est donc 5001h. Le tampon de sortie isole momentanément le bus interne du bus externe.

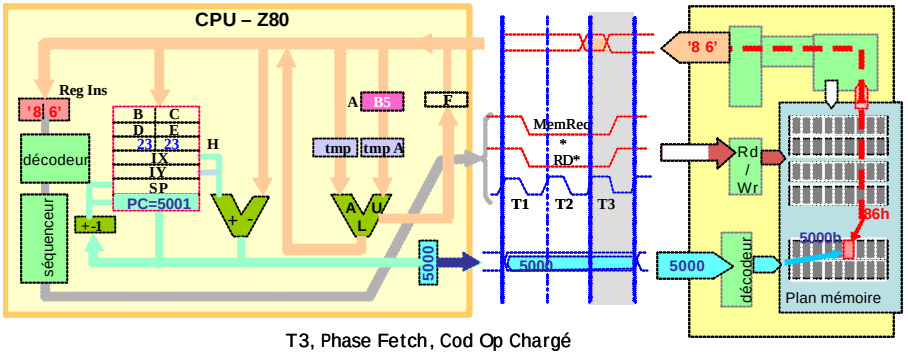


Figure 4–5 T3 phase fetch, Code Opératoire chargé

Le cycle de lecture de la phase fetch est légèrement différent de celui concernant les données ou les opérandes vu dans le chapitre 3 section 3.6.2.2. Les données sont échantillonnées au tout début de la période T3, pour utiliser une partie de T3 et la totalité de T4 pour le décodage du code opératoire et procéder au rafraîchissement de la DRAM qui est, dans le cas du Z80, gérée par le CPU.

La valeur lue (grâce à l'échantillonnage) comme code opératoire est transférée dans le registre d'instruction en vue de son décodage.

La suite du déroulement de l'instruction concerne maintenant la préparation de

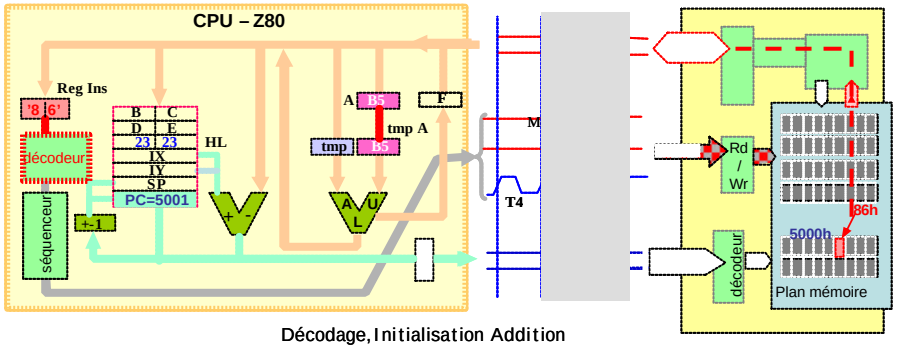


Figure 4–4 Décodage et Initialisation de l'addition.

l'addition et l'exécution de celle-ci.

A l'issue du décodage de l'instruction, le processeur sait qu'il doit additionner une valeur au contenu du registre A. Il fait alors une copie temporaire de A (B5h) dans le registre *tmp_A*, puisqu'au cours de l'exécution de l'instruction le contenu de A sera écrasé par le résultat de l'addition.

La suite concerne l'utilisation du registre HL contenant l'adresse 2323h de la variable en mémoire centrale. Le travail du processeur est d'aller rechercher en mémoire le contenu de cette adresse.

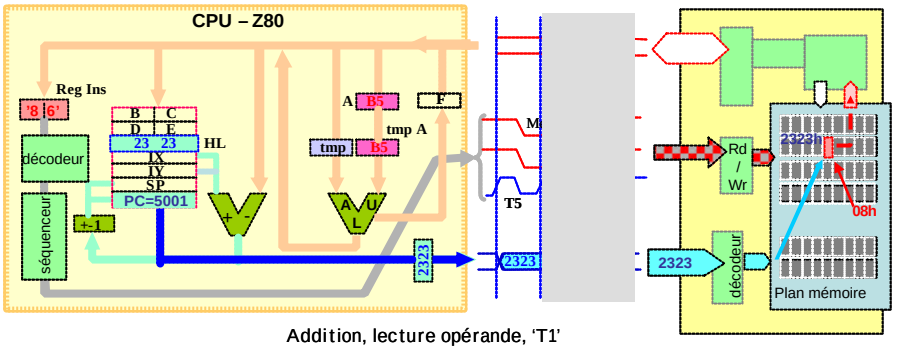


Figure 4–6 Addition, lecture de l'opérande, T1.

Le processeur lance un nouveau cycle mémoire, cette fois-ci pour récupérer un opérande. Au cours de la première période (T5) de ce nouveau cycle de lecture, le processeur fait basculer sur le bus d'adresse le contenu du registre HL, c'est-à-dire l'adresse 2323h. Celle est transmise vers le décodeur d'adresse du contrôleur de mémoire.

Le processus de lecture est ensuite quasi identique (à la position temporelle de lecture sur le bus de données près) au cycle de lecture précédent. Le contenu de

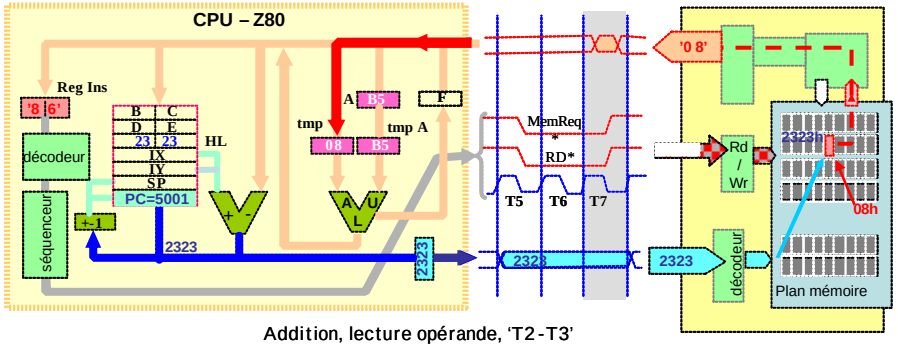


Figure 4–7 Addition, lecture de l'opérande T2-T3.

l'adresse 2323h (la valeur 08) est déposé sur le bus de données. Par contre le cheminement sur le bus est différent : cette fois la donnée échantillonnée sur le bus de données est déposée dans le registre *tmp*.

Tout est maintenant près pour faire réaliser effectivement l'addition par l'UAL. Le résultat est rangé dans le registre A, et l'on peut noter que le registre F des drapeaux de résultats d'opérations arithmétiques et logiques est mis à jour.

L'état final du processeur vis-à-vis de l'exécution du programme est donné dans la figure 4-8 avec l'ensemble des valeurs des registres. Le résultat de l'addition est dans le registre A et PC pointe sur la nouvelle instruction à exécuter.

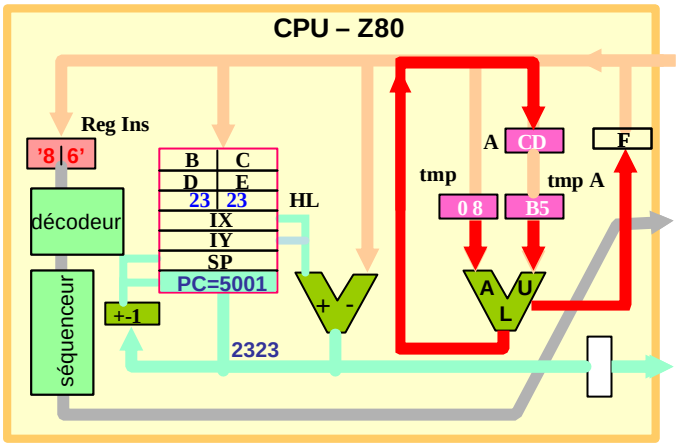


Figure 4–8 Addition et résultat.

Un premier bilan s'impose à ce niveau : le processeur est un automate qui enchaîne les instructions sans jamais s'arrêter : le programme (suite 'cohérente' d'instructions) doit donc, en permanence,l' alimenteravec des instructions. L'exécution d'un programme est totalement déterministe, il est toujours exécuté de la même manière.
On note aussi que, au cours de l'exécution d'une instruction vue dans son intégralité, le processeur passe l'essentiel du temps à gérer les échanges avec la mémoire. Le temps de l'opération, à proprement parlé quand il s'agit d'une fonction de calcul, va de 10 à 30% du temps d'exécution total de l'instruction. Globalement, un processeur passe beaucoup plus de temps à transférer les données d'un endroit à un autre qu'à faire des traitements.

4.2.1.2 Structuration, Organisation générale des instructions.

Les instructions du Z80 ont une longueur variable allant de 1 à 4 octets suivant leur complexité. Les instructions utilisées les plus fréquemment sont optimisées pour tenir sur 1 octet ce qui est un facteur déterminant dans la vitesse d'exécution.

Cette longueur d'instruction variable est une caractéristique des ***processeurs CISC***.

Les instructions d'un processeur sont classées en plusieurs catégories.

- Les instructions **arithmétiques et logiques**. Dans le cas du Z80, ces instructions ne sont pas très puissantes : elles concernent les opérations de base sur 8 bits et quelques opérations sur 16 bits. La multiplication et la division d'entiers 16 bits, et a fortiori 32 bits, sont forcément réalisées en logiciel. Il en est bien sûr de même pour les opérations sur les réels. Avec des processeurs plus récents, les opérations sur les entiers 32 bits et les réels sont directement effectuées au niveau interne par des unités de calcul spécialisées.
- Les instructions de **chargement et mémorisation**. Ce sont les instructions qui permettent d'effectuer les mouvements de données entre registres et entre registres et mémoire centrale, ou de bloc de mémoire à bloc de mémoire. D'un processeur à un autre, elles sont d'une grande diversité, principalement par la variété des modes d'adressage mis en œuvre. Elles s'appellent Load, Store, Move, Mov suivants les processeurs, LD sur le Z80.
- Les instructions de **contrôle de flux** ou instructions de **branchement**. Le but de ces instructions est de permettre de rompre la stricte linéarité du déroulement d'une suite d'instructions due au passage automatique du processeur à l'instruction suivante en mémoire. Les instructions de branchement sont celles qui permettent de modifier le contenu du compteur de programme PC et ainsi de changer le parcours linéaire des instructions d'un programme. Ce sont des instructions de type '*GoTo*', '*Aller_à*' à une adresse de programme particulière. Suivant les processeurs, elles s'appellent, JMP (*jump*), JP, Bra (*Branch*), …. Une classe particulière des instructions de branchement est constituée des instructions de branchement conditionnel : la modification du compteur de programme n'est faite que si une condition est vraie, sinon c'est l'instruction

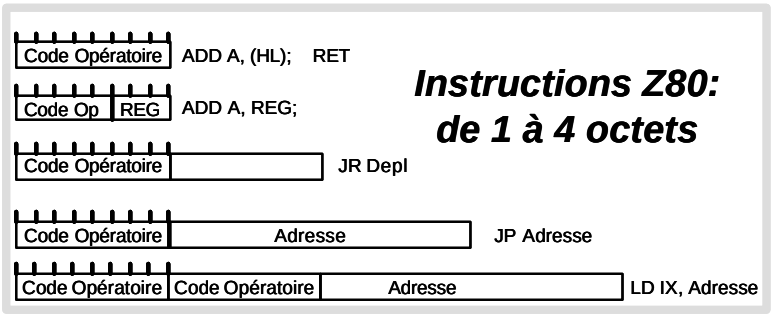


Figure 4–9 Addition et résultat.

normalement prévue en séquence qui est exécutée. Dans le cas du Z80, ce sont les drapeaux, c'est-à-dire les bits du registre F, qui indiquent si la condition est vraie ou fausse. L'association de certains drapeaux du registre F et d'une instruction de branchement conditionnel permet de réaliser les constructions algorithmiques de la forme « SI condition ALORS ….SINON ….. ». Le principe est le suivant : la condition résulte en général d'une opération de comparaison faite à l'aide des instructions arithmétiques et logiques. L'exécution de celle-ci (comparaison, soustraction, …) positionne un drapeau dans le registre F, par exemple le drapeau Z si le résultat de l'opération est nul. Une instruction de type « JMP Z Adresse1 » exploite la valeur du drapeau Z : s'il est à vrai, le processeur met la valeur *Adresse1* dans le registre PC et la prochaine instruction exécutée sera celle qui se situe à cette adresse. Si Z est à faux, alors c'est l'instruction normalement prévue et immédiatement derrière qui est exécutée.

- Un autre ensemble particulier d'instructions de branchement est constitué par les instructions d'**appel et de retour de procédure** (***Call***, ***Ret*** pour le Z80). Nous décrirons dans un paragraphe suivant les implications de leur mise en oeuvre.
- Les **instructions diverses**. Comme dans toute classification, il reste quelques instructions plus ou moins inclassables. Ce sont les instructions concernant les entrées/sorties, les interruptions ou autres instructions diverses dont certaines feront l'objet d'un zoom particulier dans d'autres chapitres.

4.3 La procédure : un élément de structuration.

Les premiers développements logiciels ont rapidement fait apparaître une nécessité de structuration dans le développement des programmes, spécialement pour la réutilisation de morceaux de codes déjà écrits. La création de bibliothèques de fonctions écrites et testées une fois pour toutes, s'avère utile pour la fiabilité et la rapidité du développement de grosses applications.

Même pour le développement d'un seul programme, il est intéressant de bien isoler les parties de code réutilisable. Une première forme de réutilisation de code déjà écrit est la 'recopie' de lignes de codes (on dirait maintenant faire du copier-coller) dans les macro-assembleurs. Le procédé consiste à répliquer autant de fois que nécessaire un morceau de code, le programme augmente en taille avec le nombre de réplications. Ce mode comporte aussi tous les dangers du 'copier-coller', c'est-à-dire l'extraction de quelque chose hors de son contexte. La forme la plus pratiquée et la plus sûre de réutilisation est l'écriture de procédure.

Une **procédure** est une suite d'instructions correspondant à une fonction bien précise. Un des premiers (au sens historique) intérêts de la procédure est que son code, dans le programme, n'est présent qu'une seule fois en mémoire procurant ainsi un gain de place non négligeable en occupation mémoire.

La procédure est un morceau de code qui se trouve en mémoire centrale à une adresse déterminée. Son utilisation est a priori : il suffirait d'une instruction de branchement inconditionnel de type « *GoTo Adresse_début_de_procédure* » pour que le processeur passe à son exécution. Le compteur de programme PC voit son contenu écrasé et remplacé par l'adresse du début de la procédure. La procédure passe dans sa phase d'exécution. Le problème se pose à la fin de cette exécution : il est nécessaire de définir cette fin et surtout il faudra pouvoir revenir à l'endroit du programme d'où l'on est venu pour exécuter la procédure. Une procédure est *appelée* et il faut 'marquer' la fin du code par une instruction spécifique de terminaison RET (*return*, *retour*) qui génère le retour au programme appelant. L'instruction qui fait l'appel de procédure doit en conséquence mémoriser, dans un registre par exemple, l'adresse de retour, adresse qui sera prise en compte par l'instruction RET.

Cependant, le problème se complique à la généralisation des procédures : il serait normal de pouvoir appeler une procédure à l'intérieur d'une procédure et ainsi de suite. La complication tient au niveau de la mémorisation de l'adresse de retour, ou plutôt, cette fois-ci, des adresses de retour. Il est donc impossible de mémoriser simplement l'adresse de retour dans un registre, car avec un nouvel appel à l'intérieur de la procédure, la première adresse de retour sera écrasée, perdue. A chaque appel de procédure, il faut maintenir une sorte de fil d'Ariane par la mémorisation successivement de toutes les adresses de retour. Le retour à l'appelant initial se fera en employant toutes ces adresses dans l'ordre inverse de leur stockage.

Concrètement ce stockage correspond à une file d'attente d'adresses de retour gérée en LIFO (*Last In First Out*) : la dernière adresse entrée dans la file est la première qui sera retirée. Cette structure de données en file d'attente résidente en mémoire centrale et gérée en LIFO est appelée une **pile**.

Initialement, la pile est vide et est seulement définie par son adresse de base BP (*Base Pointer*) qui marque le début de la zone en mémoire centrale qui lui est réservée. La pile va 'monter ou descendre' (grandir/diminuer) au fur et à mesure des empilements ou des dépilements : elle est alors définie par son *pointeur de pile* (SP, *Stack Pointer*) qui donne l'adresse du sommet de pile. À l'initialisation SP est égal à BP (pile vide). La pile est une structure de données à taille dynamique.

La 'main tenant le fil d'Ariane' du mécanisme de gestion (appel et retour) d'une procédure est le pointeur de pile SP. Au moment de l'appel de procédure, le processeur doit avoir à sa disposition le pointeur de pile pour mémoriser dans la pile l'adresse de retour contenue dans le registre PC. Dans le cas du Z80, c'est le registre SP, dont nous avons parlé lors de la présentation des registres de ce processeur, qui contient ce pointeur.

Voyons maintenant plus en détail le fonctionnement d'un appel de procédure.

Appel de procédure (Z80), instructions CALL et RET

La description d'un appel de procédure est faite sous l'angle du modèle de programmation, en abandonnant totalement la description physique. La vue est purement logicielle. Nous partons des conditions initiales suivantes : le processeur exécute un programme et à l'achèvement de l'exécution d'une instruction, le pointeur de pile SP vaut 3002 et le compteur de programme PC vaut 1A47h. Cette valeur pointe sur la prochaine instruction à exécuter, soit l'instruction **Call 2135h** (de code opératoire 'CD$_h$').

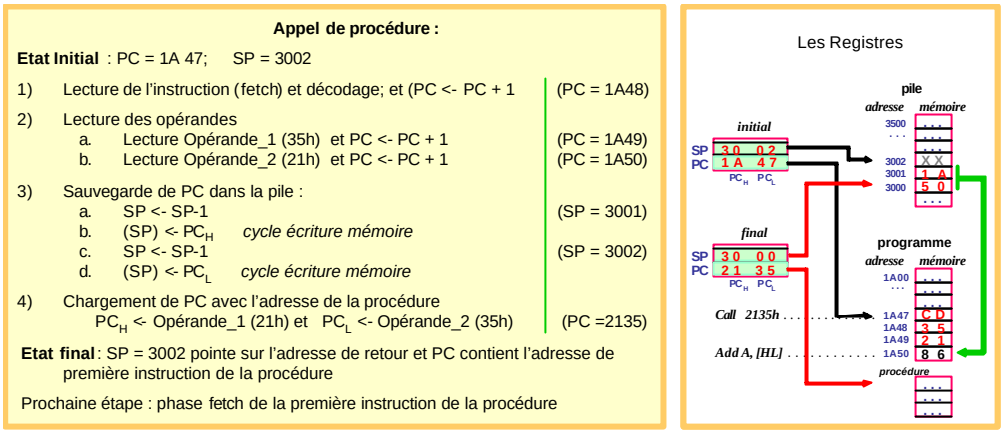


Figure 4–10 Appel de procédure.

Les différentes phases de cet appel sont décrites dans la figure 410. La partie droite visualise les registres PC et SP ainsi que les deux zones mémoires impliquées par l'appel : une zone de mémoire pour la pile et la zone mémoire de code contenant les instructions.

La zone mémoire pour la pile est réservée avant l'exécution d'un programme. La convention d'utilisation d'une pile est telle qu'elle augmente (monte) avec les adresses qui diminuent. Notre adresse de base est 3500h et l'adresse contenue dans le pointeur de pile est décrémentée au fur et à mesure du stockage dans la pile. Chaque opération d'empilement décrémente le pointeur de pile et range la valeur à la nouvelle adresse obtenue. C'est une convention universellement adoptée. Une opération de dépilement effectue la lecture de la valeur puis incrémente le pointeur de pile.

Au cours de la phase fetch (récupération du code opératoire 'CD') de l'instruction *Call 2135h* le processeur incrémente PC qui pointe maintenant sur l'opérande de l'instruction. Cet opérande est l'adresse de début de la procédure (2135h). On remarquera que, si on regarde de près, l'adresse de l'opérande pointe sur l'octet de poids faible (valeur 35h) de cette adresse. Cette convention d'organisation de la mémoire vis-à-vis des variables de plus d'un octet est appelée 'little endian' ou

'petit boutiste' : l'adresse de la variable donne l'adresse de l'octet de poids faible. C'est la convention adoptée sur le Z80.

Les deux octets de l'opérande sont lus, mis dans un registre temporaire et le compteur de programme PC est incrémenté. À la fin des deux lectures, PC vaut 1A50h et pointe sur la prochaine instruction (de code 86 – ADD A, [HL]) à exécuter lors du retour de l'appel de procédure.

Cette adresse est mise dans un registre temporaire, en attendant que le compteur de programme soit sauvegardé dans la pile. Cette sauvegarde se fait octet par octet et le pointeur de pile est décrémenté en conséquence. Le sommet de pile vaut 3002 à la fin de la sauvegarde. Le compteur de programme peut maintenant accueillir l'adresse de la procédure stockée momentanément dans le registre temporaire.

Le bilan du déroulement de l'instruction d'appel de procédure est en conséquence un simple changement de la valeur de PC avec une sauvegarde de l'adresse de retour. L'ensemble nécessite cependant, simplement pour faire l'appel, 17 périodes d'horloge.

RET retour de procédure :

Etat Initial : PC = 1A 47; SP = 3002

1) Lecture de l'instruction (fetch) et décodage; et PC ← PC + 1
   (valeur de PC indifférente, ne sera pas utilisée)
2) Restitution de PC à partir de la pile :
   a. PC_L ← (SP) *cycle lecture mémoire (pile)*
      SP ← SP+1 (SP = 3001)
   b. PC_H ← (SP) *cycle lecture mémoire (pile)*
      SP ← SP+1 (SP = 3000)

Etat final : SP = 3000 pointe sur le sommet de la pile d'avant l'appel et PC contient l'adresse de retour (1A50)

Prochaine étape : phase *fetch* de l'instruction suivant l'appel de procédure.

Figure 4–11 Retour de procédure.

La procédure peut maintenant s'exécuter. À la fin de cette exécution, la procédure comporte l'instruction RET sans paramètres. Plus précisément son paramètre est implicite : c'est le sommet de pile dont il faudra prendre le contenu pour le mettre dans PC. Le pointeur de pile est incrémenté et le programme reprend alors normalement son cours.

L'instruction RET demande 10 périodes d'horloge : l'ensemble des frais généraux (*overheads*) générés par l'appel de procédure (sans pour l'instant s'occuper du passage de paramètres) est de 27 périodes d'horloge, ce qui n'est tout de même pas négligeable.

4.4 Organisation des données (*Endianess - Boutisme*)

Nous venons de rencontrer pour la deuxième fois la notion de représentation d'un registre vers un autre support. La première fois, il s'est agit **de la transmission série des données** sur un support de transmission (**chapitre 3, section xx**) avec deux choix possibles : envoyer d'abord le bit de poids fort ou d'abord celui de poids faible. La question se repose, mais cette fois-ci pour 'transférer' les données des registres vers la mémoire (et bien sûr pour le transfert inverse) et avec la granularité de l'octet au lieu de celle du bit. L'organisation basique de la mémoire est une suite contiguë d'octets où un octet particulier est référencé par son adresse. Pour cet octet il n'y a aucune ambiguïté par rapport à un registre de 8 bits : l'ordre des 8 bits d'un registre est le même que celui des 8 bits de l'octet en mémoire.

Par contre, lorsque le processeur travaille sur un registre de 16 bits et si celui-ci est à transférer dans une variable en mémoire, il faut définir à quel octet du registre correspond l'adresse de la variable. Soit OF l'octet de poids fort (bits 8 à 15) du registre et Of l'octet de poids faible.

Convention 'Little Endian' ou 'petit boutiste'.

Dans la convention appelée 'Little Endian' ou 'petit boutiste' utilisée dans le Z80 (et dans les processeurs de la famille Intel x86 y compris le Pentium) l'adresse en mémoire d'une variable 16 bits est celle de l'octet Of : c'est l'octet Of qui est rangé en premier. L'octet OF le suit à l'adresse +1. L'intérêt de cette convention est d'éviter toute opération réelle dans une conversion sur des entiers de taille différentes (8, ou 16 bits) : la conversion devient implicite car l'adresse de la variable est l'adresse de l'octet de poids faible.

Par contre, cette convention ne facilite pas la lisibilité humaine : alors que la visualisation de la mémoire est consécutive, il faut permuter l'ordre des octets pour assurer une lisibilité par rapport à nos conventions classiques.

Cette convention est peu à peu abandonnée dans les processeurs modernes.

Convention 'Big Endian' ou 'grand boutiste'.

C'est la convention dominante maintenant : les octets sont rangés dans l'ordre décroissants des poids. L'octet de poids fort est le premier octet à être rangé, les autres le suivent dans l'ordre 'naturel' de lisibilité.

Dans la suite des exemples de programmes en assembleur, nous referons régulièrement mention à ces conventions.

La figure 4-12 reprend le codage de l'appel de procédure du Z80 où l'adresse de la procédure est donnée avec la convention Little Endian et le met en comparaison avec l'instruction équivalente BSR (*Branch Subroutine*) d'un processeur comparable, le 6809 de Motorola , utilisant la convention Big Endian.

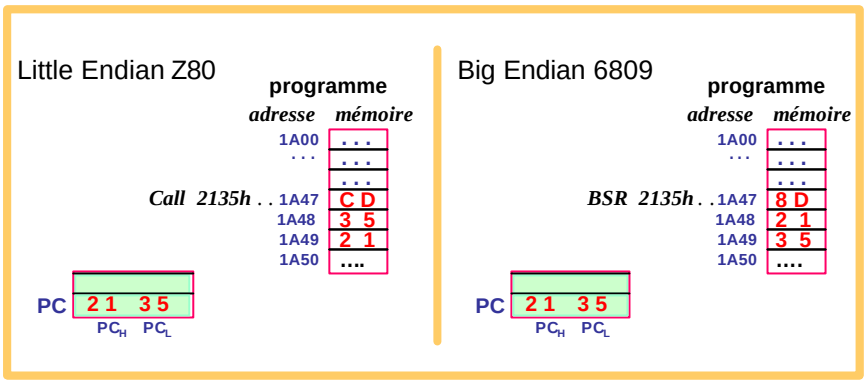


Figure 4–12 Little et Big Endian.

Le problème de lisibilité du Big Endian se complique encore lorsqu'on passe à une représentation d'une valeur sur 32 bits : il faut intervertir les mots de 16 bits de poids fort et 16 bits de poids faible et refaire la même intervention des octets de poids forts et faible à l'intérieur de chaque mot de 16 bits. Par exemple, si l'on regarde l'implantation en mémoire du nombre décimal 439 041 101 codé en complément à 2 et dont la valeur est $1a2a3a4d_h$, alors on obtient les représentations suivantes

$1a2b\ 3c4d_h$ avec un 68020
$4d3c\ 2b1a_h$ avec un386 (ou Pentium)

4.5 Gestion de Pile

… les derniers seront les premiers et les premiers seront les derniers...

L'organisation d'un logiciel sur la base des procédures est un premier niveau de structuration en programmation. Elle concerne essentiellement le code. La structuration peut aussi être introduite au niveau des données : il s'agit de manipuler un ensemble de données homogènes comme un objet unique. La donnée se verra alors affecté un traitement comme élément appartenant à cet objet. Le calcul scientifique est friand de ce genre d'abstraction, les plus courantes étant les vecteurs et les tableaux. La donnée est alors référencée par sa position (son indice) dans l'objet vecteur ou l'objet tableau. La structure en langage C est une autre manière d'organiser les données : chaque donnée est un 'champ' de la structure. Dans ce cas les données peuvent être hétérogènes.

La pile dont nous avons besoin pour la gestion des appels de procédure est une liste d'attente, c'est-à-dire une liste de données mémorisées en vue d'un usage ultérieur. Les opérations qui se font sur une liste sont des opérations plus complexes qu'une simple lecture ou écriture : la mémorisation est une écriture qui augmente la taille de la liste alors que la 'lecture' est de fait une lecture avec consommation, c'est-à-

dire un retrait de la liste. Plutôt que de parler de lecture et écriture, on utilisera les termes d'insertion et d'extraction.

De telles listes sont appelées files d'attente auxquelles on doit associer un mode de gestion. L'un de ceux-ci, bien connu dans la vie quotidienne, est le mode FIFO (*First In First Out*, Premier Arrivé Premier Servi) qui est le modèle adopté et en général accepté, pour la gestion des files d'attente dans la plupart des services publics ou commerciaux (caisses, guichets, télésièges, …). La pile, quant à elle, est une liste d'attente gérée en mode LIFO (*Last In Fist Out*, Dernier Arrivé Premier Servi) : les humains ne l'aiment pas beaucoup se la voir appliquée, mais la pratiquent souvent quand il s'agit de dossiers dans une corbeille !

La bonne gestion d'une pile est déterminante pour le déroulement d'un programme et il faut absolument veillez à ce que le pointeur de pile soit cohérent tout au long du déroulement d'un programme. Les erreurs de manipulation de la pile sont fatales à la fois pour le programme (fausse adresse de retour, …), mais éventuellement aussi pour le système d'exploitation. Pour prévenir ce risque, la plupart des processeurs prévoient la gestion de deux piles différentes, une pour les applications des utilisateurs, une autre pour les programmes du système d'exploitation.

Motorola introduit ainsi, dès son processeur 8 bits 6809, 2 registres pointeurs SP_U et SP_S (Utilisateur et Système). On les retrouve dans le 68 000 que nous allons prendre comme cas d'illustration pour les techniques d'adressage.

4.6 Les techniques d'adressage, cas du 68000.

Nous avons introduit la mémoire centrale comme un espace à adressage linéaire continu La mémoire est une suite linéaire d'octets, chaque octet est repéré par son adresse et les octets se 'suivent' avec des adresses continues. Les adresses sont à valeurs entières et l'on passe d'un octet au suivant par l'incrémentation de l'adresse. Pourquoi parler de techniques d'adressage, puisque pour accéder à un octet en mémoire il faut et il suffit de donner son adresse. Une instruction du type '*lire source vers destination*' devrait suffire. Par exemple 'lire 5000_h vers A' est suffisante pour faire le transfert de l'octet en mémoire à l'adresse 5000 dans le registre A du processeur. Une telle instruction est effectivement suffisante, mais demande beaucoup de manipulation au programmeur pour déterminer l'adresse d'un élément dans des structures de données. Les constructeurs ont donc proposé un paramétrage de l'adressage destiné à faciliter la lisibilité du programme et surtout le travail du programmeur si le programme est développé en assembleur. Lorsque le programme est écrit dans un langage de haut niveau, les critères de facilité et de lisibilité n'interviennent plus puisque c'est le langage qui s'en charge et le compilateur qui fait la traduction en code machine. Par contre, le paramétrage a pour effet de compacter le code exécutable, ce qui était intéressant lorsque la mémoire était chère.

Ce paramétrage concerne les différentes manières d'accéder un élément en mémoire centrale. A chaque technique d'adressage correspond une manière de calculer une adresse en mémoire à partir des paramètres donnés dans l'instruction.

Nous allons décrire quelques unes de ces techniques avec le processeur 68 000 car les exemples sont assez simples à introduire.

4.6.1 Description succincte du 68000.

Le MC68000 de Motorola est un processeur 16 bits mais conçu dès l'origine avec des registres 32 bits de manière à assurer une compatibilité ascendante future. Il est à la base de la famille 680x0 (x= 2, 3, 4, 5, …) dont le 68020 est la première implémentation en vrai 32 bits (registres et bus d'adresse et de données sur 32bits).Les tailles des opérandes sont définies de la manière suivante : l'octet fait 8 bits, le mot 16, le mot long 32 et le double long 64.

Il possède nombreux modes d'adressage. Les modes d'adressage sont les différentes manières d'indiquer au processeur la façon d'accéder à une donnée en mémoire centrale vis-à-vis d'une structure de données d'un langage évolué (vecteurs, tableaux, structures, …). L'objectif des concepteurs a été que les instructions arithmétiques et logiques intègrent des schémas de description de lecture ou d'écriture des opérandes relativement complexes. Les instructions deviennent très puissantes, car en peu de lignes de code on peut réaliser des opérations complexes.

L'organisation des données dans les registres.

Les 8 registres de données peuvent prendre en charge des données de 1, 8,16, 32 et 64 bits, des adresses de 16 ou 32 bits, ainsi que des champs de bits de 1 à 32 bits. Les 7 registres d'adresses, A0 à A6, et le registre pointeur de pile, A7, manipulent des adresses sur 16 ou 32 bits.

Les registres de données : D0 à D7.

Chaque registre de données a une taille de 32 bits. Les opérandes de type *byte* occupent les 8 bits de poids faible, ceux de type *word* occupent les16 bits de poids faible et les *longs* occupent la totalité des 32 bits. Dans la programmation en assembleur, la différentiation de la taille des opérandes est faite en suffixant l'instruction par .b, .w, .l respectivement (Byte, Word, Long).

Le registre CCR (*Code Condition Register*) est associé aux registres de données : il mémorise les résultats

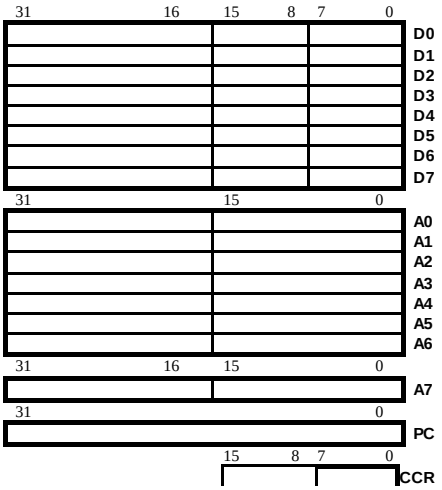


Figure 4–13 Les registres du 68K.

synthétiques des opérations logiques et arithmétiques pour être utilisés comme condition dans les instructions de branchement conditionnel (il est l'équivalent du registre F du Z80).

Les registres d'adresses A0 à A7.

Une adresse est donnée sur 32 bits ce qui procure un espace adressable de 4 Go. Le registre A7 joue un rôle particulier : c'est le pointeur de pile appelé aussi USP, *User Stack Pointer* (pointeur de pile utilisateur). Le processeur a deux modes de fonctionnement : dans le mode User, A7 est le pointeur de pile USP et dans le mode Superviseur (réservé à l'exécution du système d'exploitation), A7 est le pointeur de pile SSP.

Les instructions du 68000 et Notations assembleur.

Les instructions comportent un code opératoire sur 16 bits et un nombre variable d'opérandes. Les instructions ont des longueurs variables. Lorsqu'elles font apparaître des opérandes source et destination, la convention Motorola donne le premier opérande comme la source et le second pour la destination.

4.6.2 Les modes d'adressage du 68000.

Décrivons quelques uns des adressages les plus utilisés. Pour chacun des modes, nous prendrons un exemple d'instruction impliquant ce mode et qui sera visualisé avec les mouvements de données induits sur les registres, la mémoire où se trouve le code et celle où se trouve les données.

4.6.2.1 Adressage direct de registre de données

notation : Dn, taille : b,w,l (8,16, 32), L'opérande est le contenu de Dn

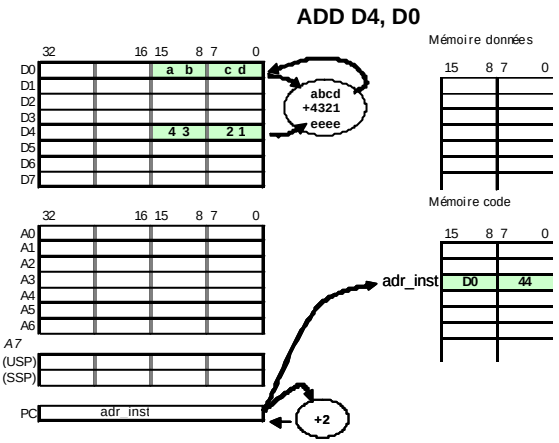
L'adressage direct de registre de données est un adressage qui concerne uniquement les registres.

L'instruction ADD D4, D0 effectue l'addition du contenu du registre D4 à celui de D0.

L'instruction demande un seul accès en mémoire pour la lecture du code opératoire (D0 40).

La zone mémoire de données n'est pas affectée par cette opération.

A la fin de l'exécution de l'instruction le compteur de programme est incrémenté de 2 pour pointer sur l'instruction suivante.



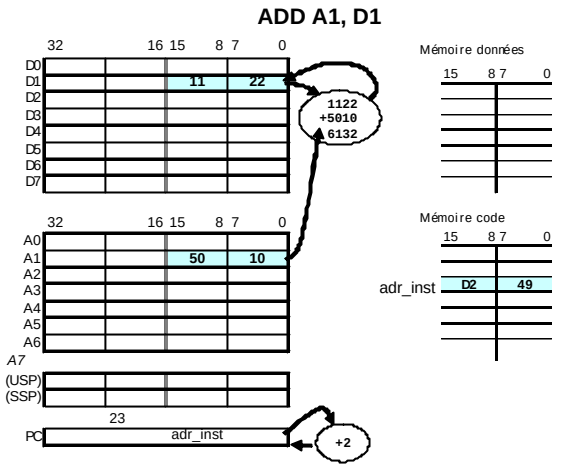
4.6.2.2 Adressage direct de registre d'adresse

notation : An, taille : w,l (16, 32), L'opérande est le contenu de An

Dans cette instruction l'opérande source est un registre d'adresse An et le registre destination un registre de données Dn.

Le contenu du registre A1 est additionné au contenu du registre D1.

L'instruction ne demande qu'un seul cycle mémoire (phase fetch) et à l'issue de l'exécution de l'instruction, le compteur de programme est incrémenté de 2 pour pointer sur l'instruction suivante.



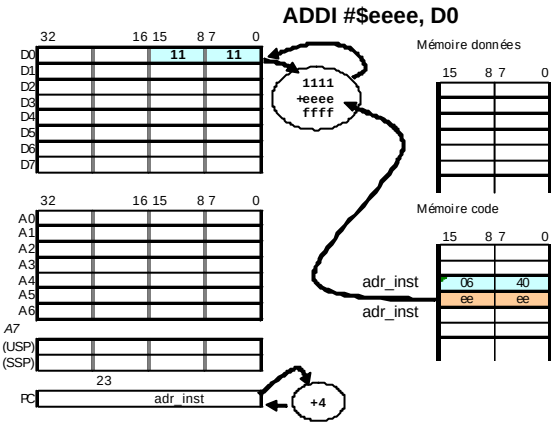
4.6.2.3 Adressage immédiat

notation :#valeur, taille : b,w,l (8,16, 32),

L'instruction fournit la valeur de l'opérande

L'instruction ADDI #$eeee, D0 ajoute la constante $eeee_h$ au contenu du registre D0.

L'adressage immédiat indique que l'opérande est placé juste derrière le code opératoire de l'instruction. Cet opérande est forcément une constante. L'adressage immédiat est spécifié par le symbole #.



Le nombre à ajouter peut être décrit en différentes bases. La base par défaut est 10 et le symbole $ de l'exemple indique un nombre en base 16.

L'instruction demande deux cycles de lecture mémoire (la phase fetch et la lecture de l'opérande). L'opérande étant sur 16 bits, le compteur de programme est incrémenté de 4 pour pointer sur l'instruction suivante.

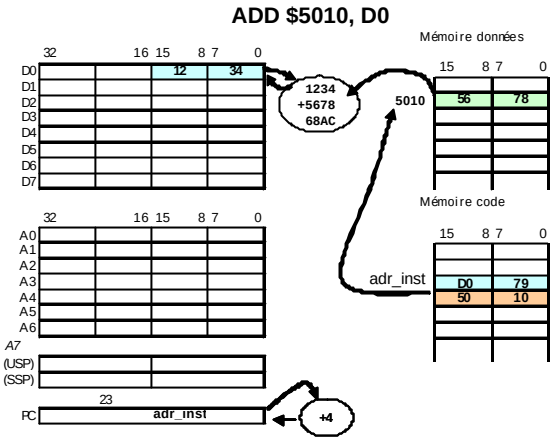
4.6.2.4 Adressage absolu

notation : valeur_adr, taille : w,l (16, 32),
L'instruction fournit l'adresse de l'opérande

Le premier opérande de cette instruction n'est pas une donnée (pas de signe #), mais une adresse décrite en valeur hexadécimale (signe $).

Le contenu de la variable à l'adresse 5010, soit 5678 est ajouté au contenu du registre D0.

L'instruction demande trois cycles de lecture mémoire (la phase fetch, la lecture de l'opérande adresse, puis celle de la donnée). L'opérande étant sur 16 bits, le compteur de programme est incrémenté au total de 4 pour pointer sur l'instruction suivante.

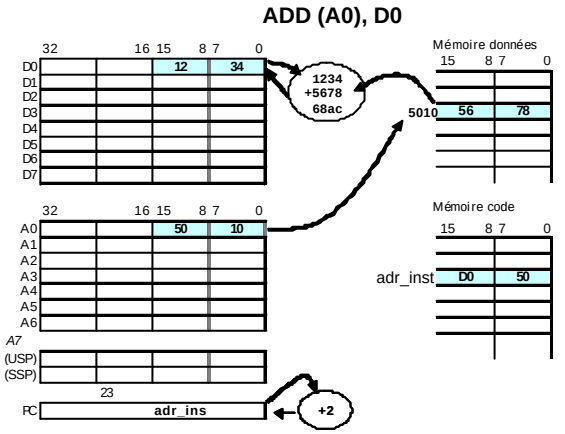


4.6.2.5 Adressage indirect sur registre d'adresse

notation : (An), taille : b,w,l (8,16, 32), An contient l'adresse de l'opérande : l'opérande est en mémoire à l'adresse pointée par An

L'adressage indirect sur registre d'adresse utilise le contenu d'un registre d'adresse, ici A0, comme adresse mémoire où se trouve la donnée à additionner au contenu du registre de données D0.

L'instruction demande deux cycles de lecture mémoire (la phase fetch et la lecture de la donnée). L'instruction n'ayant pas d'opérande explicite en mémoire, le compteur de programme est incrémenté de 2 pour pointer sur l'instruction suivante.



Variantes de ce type d'adressage : utilisation pour la gestion d'une pile de données

4.6.2.6 Adressage indirect sur registre d'adresse avec prédécrément : -(An) ; avec postincrément : (An)+

notation :-(An), taille : b,w,l (8,16, 32), On soustrait d'abord à An la taille de l'opérande,

l'opérande est en mémoire à la nouvelle adresse pointée par An

notation :(An)+, taille : b,w,l (8,16, 32), L'opérande est en mémoire à l'adresse pointée par An. On ajoute ensuite à An la taille de l'opérande.

Faire un exemple d'utilisation d'une pile utilisateur : les premiers seront les derniers…

8 7 6 5 4 3 2 1 ⇆ 1 2 3 4 5 6 7 8

pile

4.6.2.7 Adressage indirect sur registre avec déplacement

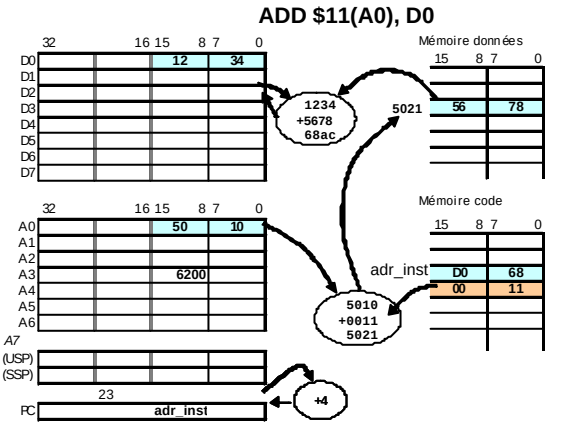
(adressage relatif / pointeur de base), notation : d16(An), taille : b,w,l (8,16, 32)

Adresse de l'opérande : contenu de An + déplacement 16 bits signé, fourni dans l'instruction

Le dernier mode d'adressage que nous allons décrire est l'adressage indirect sur registre avec un déplacement. Le registre d'adresse sert de pointeur de base à un élément, par exemple un tableau ou une structure, auquel on ajoute un déplacement.

L'instruction ADD $11(A0), D0 prend le contenu du registre A0 auquel est ajouté le déplacement de valeur 11_h pour obtenir l'adresse de la donnée à ajouter au contenu du registre D0.

L'instruction demande trois cycles de lecture mémoire (la phase fetch, la lecture de l'opérande index puis la lecture de la donnée). L'opérande étant sur 16 bits, le compteur de programme est incrémenté de 4 pour pointer sur l'instruction suivante.



Il est bien sûr possible d'aborder la programmation sur des principes de machine abstraite. On peut alors totalement masquer l'existence d'un langage assembleur.
Nous allons aborder une introduction à la programmation en assembleur dans le cadre de notre démarche ascendante. Le modèle de programmation est l'interface qui rend possible toutes les abstractions de niveau supérieur.
Cette introduction donne les principes de base de la programmation assembleur. Nous partirons de programmes très simples en langage C et nous regarderons le code assembleur généré par le compilateur.
La fin du chapitre détaille les mécanismes sous-jacents à l'appel de procédure et surtout au passage de paramètres.

4.7 Introduction au langage assembleur.

Pour cette introduction, nous allons dans un premier temps partir de petits programmes très simples en langage C. Nous demanderons au compilateur C de nous traduire le programme source C en programme assembleur pour le processeur 68000. Nous prendrons ensuite la même démarche et les mêmes exemples avec un processeur assez différent, le MIPS.

Les exemples sont illustrés par une présentation dans un même tableau du programme C, du programme correspondant en assembleur et finalement le code binaire exécutable généré qui met en œuvre le jeu d'instruction.

La partie gauche de la figure est un petit programme C ne comportant que des affectations simples. La colonne de droite est le programme assembleur 68000 généré automatiquement par le compilateur C.

La partie centrale est le code objet du programme (binaire), tel qu'il peut être chargé en mémoire centrale pour son exécution. La visualisation est faite en notation hexadécimale, 2 digits hexadécimaux décrivant un octet. La colonne de gauche marquée '*ad*' donne les adresses d'implémentation en mémoire. Ainsi, dans le premier exemple (figure 4-14), le code du programme commence à l'adresse 06 et se termine à l'adresse 25. La zone de données où sont stockées les variables commence à l'adresse 00 et se termine à l'adresse 05.

Le début du programme assembleur concerne la déclaration des variables pour lesquelles il faut réserver de la place en mémoire. Par convention le noms des variables en C gardent le même nom mais précédé du caractères '_'. Les '*int*' (entiers) en C deviennent pour ce compilateur des mots de 16 bits (suffixe .w) et sont déclarés sous forme de suite de 2 octets.

Dans la partie centrale, on peut constater que la variable '_a' a pour adresse 00 et la dernière adresse occupée par '_a' est l'adresse 01.

Le nom '*main*' en C est le nom d'une procédure particulière qui est souvent appelé *programme principal*. Ce nom devient un label ou étiquette en assembleur (*_main*). Un label est suivi du caractère '**:**' et prend pour valeur celle de l'adresse de l'instruction qui suit. Ainsi *_main* est égal à 06 qui est l'adresse de la première instruction du programme. Une étiquette est généralement utilisée comme le paramètre de destination dans une instruction de saut conditionnel ou inconditionnel.

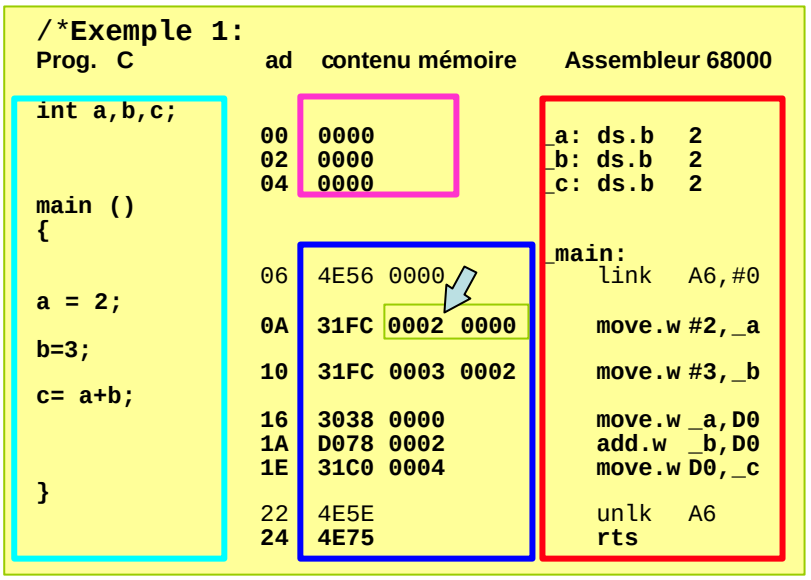


Figure 4–14 C et assembleur 68000, exemple 1.

La première instruction, *'link'*, est spécifique à la gestion d'une procédure et nous l'ignorerons dans un premier temps. Nous y reviendrons plus longuement par la suite. L'affectation 'a = 2 ;' en C devient en assembleur 'move.w #2, _a', dont la signification est : transférer un mot de 2 octets (suffixe w) dont la valeur est 2 vers la variable _a. L'instruction move.w utilise ici un adressage immédiat avec pour l'opérande source la constante 2. Le code binaire de l'instruction est 31fc 0002 0000.

Le code opératoire est 31fc et les 2 octets qui suivent (0002) contiennent le codage en complément à 2 (type int en C) de la constante 2. Les 2 octets suivants désignent l'adresse de la variable _a, soit 00. On note que la constante 2 a pour adresse 0c (première adresse après le code opératoire) et que cette adresse est l'adresse de l'octet de poids fort d la constante : la convention de représentation mémoire utilisée du 68000 est de type Big Endian. L'instruction occupe 6 octets lus par groupe de 2 (bus de données de 16 bits). Le déroulement de l'instruction demande 3 cycles de lectures en mémoire et un cycle d'écriture pour l'exécution (rangement à l'adresse de _a).

La deuxième affectation est identique dans sa forme à la valeur de la constante et de le la variable près. L'instruction 'c = a + b'[1] en C nécessite plusieurs instructions assembleur : l'addition ne peut se faire que sur un registre. Le compilateur a choisi le registre D0, mais il aurait pu choisir n'importe quel registre de données. L'addition est faite en trois temps : rangement de _a dans le registre D0, addition de la variable _b au contenu de D0, puis enfin le rangement de D0 à l'adresse de la variable _c.

Nous n'allons pas détailler chaque instruction, mais faisons cependant une exception pour l'instruction 'move.w _a, D0'. Son code est '3038 0000'. Alors que le mnémonique en langage assembleur de l'instruction est le même que le 'move' précédent, il n'a pas la même signification vis-à-vis de la technique d'adressage. Il s'agit toujours d'un transfert, mais les paramètres ne sont pas les mêmes : l'opérande source est une adresse mémoire alors que l'adresse destination est un registre. Autre caractéristique de cette instruction : elle est plus courte, seule l'adresse mémoire apparaît explicitement dans l'instruction. L'opérande destination étant un registre est codé directement dans le code opératoire.

```
/*Exemple 2:
Prog. C          ad   contenu mémoire     Assembleur 68000

main ()
{

int a,b,c;
  a = 2;

  b = 3;

  c = a + b;



}
                 00   4E56 FFFA           _main:
                                              link      A6,#-6
                 04   3D7C 0002 FFFE           move.w    #2,-2(A6)
                 0A   3D7C 0003 FFFC           move.w    #3,-4(A6)
                 10   302E FFFE                move.w    -2(A6),D0
                 14   D06E FFFC                add.w     -4(A6),D0
                 18   3D40 FFFA                move.w    D0,-6(A6)
                 1C   4E5E                     unlk      A6
                 1E   4E75                     rts
```

Figure 4–15 C et assembleur 68000, exemple 2.

[1] *Pour l'anecdote : a et b sont des constantes, donc la somme c est une constante. Un bon compilateur, faisant un minimum d'optimisation aurait du traduire*

les instructions	*a= 2 ;*	*en*	*move.w #2, _a*
	b=3 ;		*move.w #3, _b*
	c=a+b ;		***move.w #5, _c*** …

Variables dans la pile.

L'exemple 2 est une variante du programme précédent où la déclaration des variables est interne à la procédure *main*. Les variables deviennent locales à la procédure et sont allouées dynamiquement au moment de l'appel de la procédure main. La place sera rendue, libérée à la sortie de la procédure. L'instruction link A6, #-6 effectue cette réservation de mémoire. Si au moment de l'appel de la procédure, le pointeur de pile USP vaut 10b8, l'instruction va réserver la place nécessaire pour les 3 variables a, b et C dans la pile. La réservation se fait simplement en diminuant USP est diminué de 6 et vaut maintenant 10b2. USP est susceptible d'évoluer, en particulier si dans la procédure il y a un autre appel. USP étant par essence variable, les variables a, b et c ne peuvent être référencées par rapport à USP. Il faut donc, avant la réservation de la zone, mémoriser la valeur du pointeur de pile (ici 10b8) dans un registre. Les variables pourront ensuite être référencées sans problèmes par rapport à la valeur fixe de ce registre.

Dans l'exemple 2, l'instruction link se sert du registre d'adresse A6. L'adresse de la variable a est à -2 par rapport à A6, b à -4 et c à -6 (FFFE, FFFC, FFFA dans le code des instructions sont les valeurs -2, -4 et -6 en complément à 2). On peut noter que ce programme sera plus efficace en temps d'exécution car il implique moins d'accès à la mémoire.

L'exemple 3, figure 4-16, est le même que la première version, mais compilée pour un processeur 68020, c'est à dire la première version entièrement 32 bits du 68000.

Les différences principales tiennent dans les entiers représentés en 32 bits. Le compilateur a fait un choix différent du premier : il fait commencer le programme à l'adresse 00 et met la zone de données à la suite du code.

```
/*Exemple 3:  version 68020, 32 bits
Programme C          ad   contenu mémoire          Assembleur 68020

int a,b,c;           2e   0000 0000                _a:    dcb.l 1,0
                     32   0000 0000                _b:    dcb.l 1,0
                     36   0000 0000                _c:    dcb.l 1,0

main()               00   4e56 0000                _main:
{                    04   23fc 0000 0002 0000 002e link   a6, #-0
    a = 2;           0e   23fc 0000 0003 0000 0032 move.l #2,_a
    b = 3;           18   2039 0000 002e           move.l #3,_b
    c = a+b;         1e   d0b9 0000 0032           move.l _a,d0
                     24   23c0 0000 0036           add.l  _b,d0
 }                   2a   4e5e                     move.l d0,_c
                     2c   4e75                     unlk   a6
                                                   rts
```

Figure 4–16 C et assembleur 68000, exemple 21.

Le Z80 et le 68000 sont des processeurs qui relèvent du modèle CISC (Complex Instruction Set Computer), avec la caractéristique de posséder beaucoup de modes d'adressage, de pouvoir faire un adressage dans les instructions arithmétiques et logiques et d'avoir des instructions à longueur variable. Sans faire pour l'instant une discussion sur les raisons de la suprématie actuelle des processeurs RISC (Reduced Instruction Set Computer) au niveau de la performance par rapport aux CISC (chapitre 8), nous donnons maintenant un exemple de modèle de programmation d'un processeur RISC.

4.7.1 Modèle RISC 'Load and Store'.

La performance en vitesse d'exécution est actuellement basée sur la recherche de la simplicité et de la régularité dans le processeur. C'est une caractéristiques des processeurs RISC : il y a un grand nombre de registres généraux, très peu de modes d'adressage mémoire et les accès en mémoire se font indépendamment des opérations arithmétiques par des instructions de lecture et d'écriture appelées 'load' et 'store'. Les instructions ont toutes la même longueur.

A partir des années 80, plusieurs études ont montré que la complexité des CISC (avec la puissance des instructions correspondantes) devenait incompatible avec un bon niveau de performance. C'est la simplicité et la régularité, qui avec la diminution du coût des mémoires, deviennent les critères majeurs. La plupart des programmes cibles étant générés par des compilateurs (et non écrits directement en assembleur), ces études ont mis en évidence que, seules 20% des instructions d'un processeur de type CISC sont utilisées pendant plus de 80% du temps d'exécution d'un programme

La conception des processeurs est revue et les critères pris en compte sont alors :

- un jeu réduit d'instructions simples ayant toutes la même longueur et facilement décodables au niveau du processeur ;
- un nombre réduit de modes d'adressage ;
- les accès à la mémoire se font uniquement avec deux instructions : *load* (transfert de mémoire à registre) et *store* (transfert de registre à mémoire) ;
- un nombre élevé de registres pour diminuer la fréquence des échanges avec la mémoire ;
- la décomposition systématique des instructions en nombre fixe de phases élémentaires permettant une forme de parallélisme interne avec l'exécution de chacune des phases sur une unité particulière appelée étage de pipeline (chapitre 8).

En 1980, David Patterson, de l'université de Berkeley, définit une architecture de machine susceptible de répondre au cahier des charges précédent. Il lui attribue le nom de RISC, Reduced Instruction Set Computer, architecture à jeu réduit d'instructions. A partir d'études statistiques menées sur un grand nombre programmes contemporains et écrits en langage de haut niveau sur les processeurs de l'époque (appelés depuis CISC, Complex Instruction Set Computer), il prend en compte les faits suivants :

- l'instruction call est celle qui prend le plus de temps (en particulier sur un processeur VAX, un standard de l'époque et fabriqué par Digital Equipment Corp.) ;
- 80% des variables locales sont des scalaires ;
- 90% des structures de données complexes sont des variables globales ;
- la majorité des procédures ne possèdent moins de 7 paramètres (entrées et sorties) ;
- la profondeur maximale de niveau d'appels de procédure est généralement inférieure ou égale à 8.

Patterson développe ainsi le RISC I, puis le RISC II, projet universitaire qui débouche ensuite sur l'architecture SPARC (Scalable Processor ARChitecture) de Sun.

A la même époque, les chercheurs de l'université de Stanford, dont les travaux portent sur le parallélisme et les structures en pipeline, proposèrent la machine MIPS (Machine without Interlocked Pipeline Stages). L'idée maîtresse de John Hennessy, le père du MIPS, est de mettre en évidence dans le jeu d'instruction toutes les activités du processeur qui peuvent affecter les performances afin que les compilateurs puissent réellement optimiser le code. John Hennessy fonde ensuite la société MIPS. Ce processeur est utilisé dans les stations de travail SG (Silicon Graphix).

4.7.1.1 Le processeur MIPS (en version 32 bits).

Prenons le processeur MIPS pour expliciter le modèle de programmation dans une architecture RISC.

Son modèle est relativement simple. Il en comporte un ensemble de 32 registres généraux notés R0 à R31, mais tous ne sont pas indifféremment utilisables. Par exemple, comme dans la plupart des processeurs RISC, le registre R0, en lecture seule, est 'précâblé' à la valeur 0. L'initialisation à 0 d'une variable, opération courante en programmation, est ainsi très rapide. Le registre R1 est lui réservé à l'assembleur.

Les registres ont aussi des 'petits noms' pour une utilisation plus standardisée au niveau logiciel (a0, a7, …t0, t7).

Les instructions ont toutes la même longueur (32 bits). La distinction des instructions est faite suivant trois types de format. Le **format R** correspond aux instructions arithmétiques, le format s'appelle **R,** comme registre, car ces opérations ne se font que sur des registres.

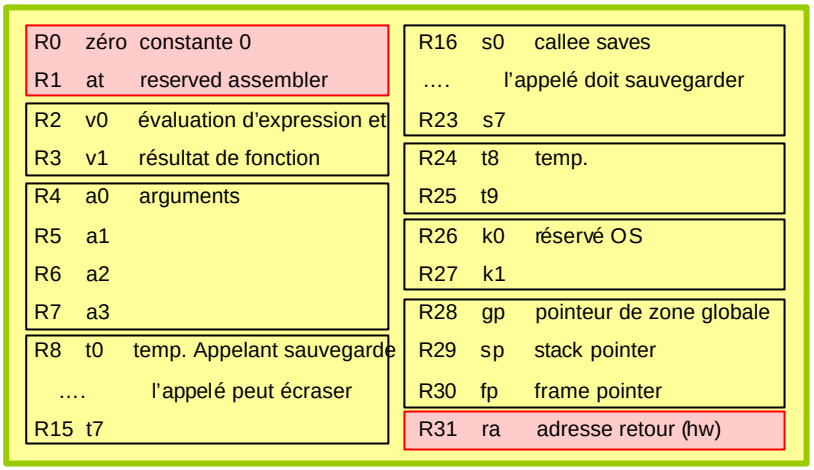


Figure 4–17 Registres généraux du processeur MIPS.

Le **format I** correspond aux instructions immédiates (opérande immédiat), de transfert et branchement et le **format J** correspond aux instructions de saut.

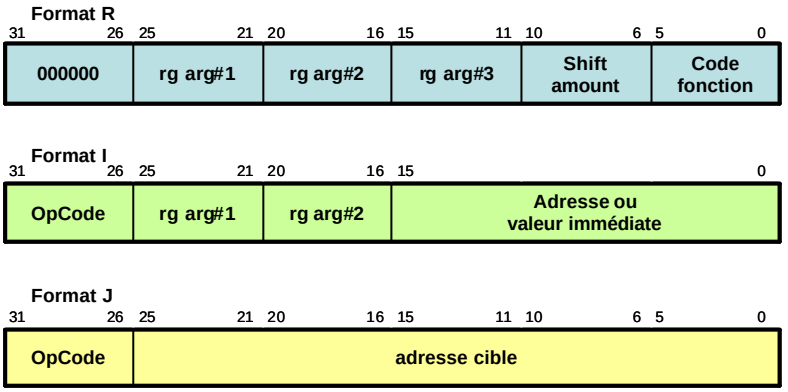


Figure 4–18 Formats d'instructions du processeur MIPS.

Les instructions de format R.

Les instructions arithmétiques n'ont que des registres en arguments (pas de variables en mémoire). Le premier argument est le registre destination et les deux suivants sont les opérandes sources. Par exemple, une addition sera décrite par :

add $s0, $s1, $s2 ce qui revient à faire l'opération $s0 = $s1 + $s2

($ est un caractère de spécification de registre).

Les instructions de format I.

L'interprétation des 3 arguments est dépendante du code opératoire. Dans le cas d'une addition nous aurons par exemple

addi $v0, $zero, 1 ce qui revient de mettre à 1 le registre $v0. $v0 est l'opérande destination, $zéro (R0) est un registre source et 1 est une valeur immédiate (une constante).

Dans le cas d'une instruction de branchement conditionnel, les deux registres font l'objet de la comparaison de décision pour le branchement et le dernier champ donne l'adresse de débranchement sous la forme d'un offset.. Par exemple, l'instruction :

bne $t3, $zero, label0 fait le branchement à l'étiquette label0 si le contenu de $t3 est différent de 0 (*bne* = *b*ranche if *n*ot *e*qual). On remarque qu'il n'y pas de registre de drapeaux mis à jour automatiquement par l'UAL : l'instruction fait la comparaison et le saut (à la différence du Z80 et du 68000 vus précédemment).

Le format I comporte également les deux seules instructions de transfert mémoire / registres. L'instruction **load**, notée lw, effectue un chargement depuis la mémoire vers un registre ; l'instruction **store**, notée sw, se charge du rangement d'un registre vers la mémoire. L'instruction :

lw $t0, 48($t1) est équivalente à t0 = mem[t1 + 48]. Le registre t0 est le registre de destination (chargement, load, lecture), t1 est un registre source auquel on ajoute la valeur immédiate 48 pour obtenir l'adresse mémoire d'où se fait le chargement.

sw $t0, 48($t1) est une instruction de rangement du registre t0 vers la mémoire.

Les instructions de format J.

L'instruction caractéristique du format J est l'instruction de saut (*'goto'*). Elle se présente sous la forme j étiquette, où étiquette est une adresse sur 26 bits.

Les modes d'adressage.

Il y a quatre modes d'adressage pour le processeur MIPS. Dans ce domaine également c'est la simplicité qui prime, d'où ce faible nombre de modes d'adressage.

L'**adressage par registre** : c'est l'adressage le plus simple et le plus efficace : l'opérande de l'instruction est contenu dans le registre désigné.

add $s0, $s1, $s2 est une instruction dont les paramètres sont des noms de registres, l'opérande est dans le registre désigné.

L'**adressage immédiat** : c'est un adressage où l'opérande est une constante directement donnée dans l'instruction.

addi $v0, $zero, 1 est une instruction à adressage immédiat dans le cas du dernier paramètre, la valeur du paramètre est l'opérande.

L'**adressage indexé** ou adressage avec déplacement : l'opérande se trouve à l'emplacement mémoire d'adresse égale au contenu d'un registre ajouté un déplacement contenu dans l'instruction elle-même.

lw $t0, 24($t1) correspond au chargement d'un élément en mémoire qui se trouve à l'adresse contenue dans le registre t1 à laquelle est ajoutée la constante 24. Le déplacement peut aussi être fait par rapport au compteur de programme PC.

---- revoir les notations valeurs, références, pointeurs ----

Les appels de procédures.

Il y a deux instructions pour gérer les appels de procédures. Pour les comprendre, il faut avoir à l'esprit que le paradigme RISC est basé sur une utilisation maximale des registres au détriment de la mémoire et en particulier de la pile. C'est donc au programmeur (ou plutôt au compilateur) de gérer le problème de l'écrasement de l'adresse de retour, dans un registre libre. Ce n'est que lorsqu'il n'y a plus de registres de libre que le mécanisme de la pile en mémoire est mis en œuvre. L'instruction ***jal proc1*** (*jump and link*) de fait le débranchement vers l'adresse proc1 de début de la procédure et mémorise l'adresse de retour dans le registre *ra* (*return address* ou R31). La procédure se termine par l'instruction jr (jump register) avec comme argument *ra* : ***jr $ra***.

```
procA :
   …
      jal   procB

procB
   …
    sub $sp, $sp, 4
    sw  $ra, 0($sp)
    jal procC
    lw  $ra, 0($sp)
    add $sp, $sp, 4
    …
    jr $ra
procC
    …
    …
    jr $ra
```

Figure 4–19 Procédures MIPS.

On peut penser que cette programmation est plus difficile pour un programmeur que dans le cas d'un processeur CISC où il suffit d'appeler les instructions 'call' et 'ret'. C'est effectivement le cas, les RISC sont conçus avec un jeu réduit d'instructions plus simples, la programmation est donc plus laborieuse. Il faut aussi tenir compte d'une autre évolution, celle des compilateurs. A la pleine époque du CISC, on pensait que la programmation la plus efficace en temps d'exécution était la programmation en assembleur : un compilateur de langage évolué apportant de son coté une surcharge en code au programme assembleur généré automatiquement.
Les processeurs RISC sont conçus avec l'hypothèse c'est le compilateur, qui avec des possibilités d'optimisation, qui est le mieux à même de générer un programme optimal vis-à-vis du processeur. Au programmeur est laissé le développement d'une application avec un langage de haut niveau, et on laisse au bon soin du compilateur de traduire le plus efficacement possible ce programme en langage assembleur. Finalement, la lisibilité n'est pas un critère fondamental. Patterson donne des exemples où un programme C est plus rapide qu'un programme directement écrit en assembleur. La plupart des compilateurs C, C++ actuels sont réellement performants du point de vue de l'efficacité du code cible.

L'exemple de la figure 4-20 illustre les appels de procédure MIPS. La procédure A fait appel à la procédure B. Supposerons qu'au moment de cet appel toutes les sauvegardes nécessaires sont faites. L'instruction jal procB fait la sauvegarde de l'adresse de retour dans le registre $ra. La procédure B, après un certain nombre d'instructions, appelle à la procédure C. Mais avant cet appel, il est nécessaire de sauvegarder le registre $ra, sinon l'adresse de retour de B est perdue. Dans cet exemple, la sauvegarde est faite dans la pile (elle aurait pu être faite dans un autre registre libre). La gestion de la pile n'est pas automatique, elle est de la responsabilité du programmeur. La première instruction décrémente le pointeur de pile pour accueillir une adresse sur 32 bits. Après l'ajustement du pointeur de pile, le registre $ra est rangé au sommet de la pile. L'appel à la procédure C peut alors être fait en toute sécurité pour le retour ultérieur à la procédure A (instruction jr $ra à la fin de la procédure B).

La procédure C ne faisant aucun appel de procédure, il n'y a pas de sauvegarde à faire et dans ce cas on gagne le temps d'une sauvegarde automatique inutile.

4.7.2 Procédures et passage de paramètres.

La procédure est, nous l'avons introduit ainsi, le premier niveau de structuration d'un programme et elle requiert pour son utilisation, une encapsulation d'un ensemble d'instructions correspondant à la résolution d'une fonction déterminée. D'une certaine manière la procédure est aussi 'isolée' ou protégée du reste du programme. Elle peut avoir des variables internes de travail non visibles à l'extérieur qui sont les variables *locales* aussi appelées *privées*. Par contre, il faut bien sûr, pour que la procédure serve à quelque chose, pouvoir lui transmettre des données et pouvoir récupérer les résultats de la fonction. On parle alors de **passage de paramètres**, avec les paramètres d'entrées et les paramètres de retour.

Pour effectuer ce passage, différentes techniques sont possibles. Nous allons en décrire quelques unes et les illustrer avec les deux modèles de programmation que nous venons de voir, celui du 68000 et celui du MIPS.

Une des techniques la plus simple consiste à transmettre les données de manière implicite : ce sont les **variables globales**. Les variables sont en mémoire centrale avec une adresse statique (fixe) et rendue visible à toutes les procédures. On utilise alors simplement les variables sans les déclarées à l'intérieur de la procédure. Cette technique est assez dangereuse car n'importe quelle procédure peut modifier la variable, sans parler du fait qu'une variable globale est potentiellement, dans un système multitâche, une ressource critique et doit être traitée en tant que telle à l'aide des mécanismes d'exclusion mutuelle. Cette technique est vivement déconseillée par tous les guides de programmation et doit être réservée à des situations exceptionnelles. Alors, promis et juré, nous le ferons qu'une seule fois, juste pour voir…

La technique classique est de ménager une ouverture autorisée, une sorte de guichet de dépôt ou de retrait. Dans la plupart des langages de programmation ce guichet est symbolisé par une liste de noms de paramètres mise entre deux parenthèses après le nom de la procédure : *somme (a, b, c)*. Dans certains cas la distinction est nette entre paramètres d'entrées et paramètres de sortie et lorsqu'il y a un paramètre de sortie considéré comme principal, la procédure renvoie une valeur. C'est une fonction utilisable dans une expression comme une variable ou un élément de tableau. Dans notre exemple, nous utiliserons la fonction *som* qui appliquée aux paramètres a et b renvoie la valeur c : *c = som (a,b)*.

La notation précédente avec les parenthèses existe dans les langages de haut niveau, mais pas au stade du modèle de programmation du processeur.

Pour l'assembleur, le nom de la procédure n'est ni plus ni moins qu'une simple étiquette, c'est-à-dire une adresse. Il faut passer des paramètres entre l'appelant et l'appelé et vice versa. Par où et comment passent alors les paramètres ? Par où ? Le paramètre est mis dans une unité de stockage : c'est en mémoire centrale, dans la pile ou dans les registres[2].

- **Passage par les registres** : c'est le passage le plus rapide, mais il est limité par le nombre de registres disponibles. Le paramètre est référencé par le nom du registre.
- **Passage par la pile** : le paramètre est référencé en dynamique par rapport au pointeur de pile, mais la donnée est en mémoire centrale. Le passage est moins rapide que par les registres, mais il n'y a pas de limitation en nombre de paramètres.
- **Passage par référence explicite de la variable** : c'est de fait une technique de variable globale.

Comment passer le paramètre ?

- Le paramètre est passé par valeur, dans ce cas on transmet une copie de la donnée et l'original reste intact dans le programme appelant. Le paramètre ne peut être que 'lu' dans la procédure appelée.
- Le paramètre est passé par référence, autrement par son adresse et dans ce cas le paramètre peut être 'lu' et 'modifié'. Tout paramètre de sortie doit être passé par référence. D'une certaine manière, le passage par référence revient à rendre partiellement globale la variable, d'où des dangers équivalents à ceux des variables globales.

Procédure 'Epargner' 1 entrée, pas de sortie !, 'objet' *tire-lire*

[2] Il existe une technique variante du passage par registre qui utilise le renommage de registre avec une gestion de fenêtre de registres (processeur SPARC de Sun et Itanium de Intel et HP). Nous la traiterons à part.

Variables locales et ré-entrance.

La ré-entrance est la propriété que doit présenter une procédure pour être appelable plusieurs fois sans que les exécutions précédentes soient terminées. C'est le cas des bibliothèques de procédures dans les systèmes multitâches où un même code est appelé à différents moments par plusieurs applications. Dans ce cas, chaque instance d'exécution de la procédure doit travailler sur une zone de données différente, gérée idéalement en mettant toutes les données locales dans la pile.

Dans les exemples qui suivront, nous aurons une illustrations de ces différents points : variable globale, variable locale, différentes techniques de passage de paramètres.

Les procédures et le 68000 (variables locales, pile, instructions Link et Unlink).

Les concepteurs du processeur 68000 ont introduit l'instruction *link* pour faciliter la mise en œuvre de l'allocation dynamique de mémoire pour les variables locales à une procédure. L'instruction est relativement complexe et demande quelques explications.

La valeur du pointeur de pile est prise à l'entrée de la procédure. A partir de cette valeur on peut réserver une zone pour la déclaration des variables locales, place qui sera récupérable à la sortie de la procédure. Comme nous l'avons déjà indiqué (section 4.7, Exemple 2), le pointeur de pile ne peut pas servir de référence aux variables car il est amené à évoluer dans la suite de l'exécution de la procédure. La valeur originelle du pointeur de pile est mémorisée dans un registre d'adresse qui sert ensuite d'adresse de base pour les variables locales.

L'instruction Unlink, utilisée à la fin de la procédure, libère la zone en remettant simplement le pointeur de pile à sa valeur initiale d'avant la réservation.

L'exemple de la figure 4-20 donne une version des programmes précédents où l'addition est faite dans une fonction chargée de l'addition de deux entiers (fonction som en C et procédure _som en assembleur). L'exemple n'a évidemment qu'une valeur illustrative[3] du principe de passage de paramètres avec un cas simple.

Le début du programme reste identique pour l'affectation des constantes 2 et 3 aux variables a et b. Ces dernières, paramètres de la procédure *som*, sont passées par la pile. Rappelons que le registre A7 est le registre pointeur de pile, aussi appelé USP. L'instruction de préparation du passage 'move b, -(A7)' utilise l'adressage indirect sur registre d'adresse avec pré-décrémentation : le pointeur de pile est décrémenté

[3] Dans ce cas et d'un point de vue efficacité, il serait un peu abusif d'utiliser réellement une telle procédure puisque cela consisterait à remplacer les 3 instructions de l'addition originelle par les mêmes plus un surcoût de 8 instructions nécessaires pour la mise en œuvre de la procédure.

de la taille de *b* (2 octets), puis b est rangé à l'adresse mémoire pointée par A7. La même opération de transfert dans la pile est avec *a* : avant l'appel à la procédure (*jsr _som*) la variable a est en sommet de pile et *b* juste derrière. Les variables *a* et *b* sont donc des paramètres passés par valeur. Voyons maintenant le corps de la procédure. Dans la fonction C, nous avons déclaré, alors qu'elle n'est pas utile, une variable locale (*int k*) afin de voir l'utilisation conjointe de la pile pour le passage de paramètres et le stockage d'une variable locale. Lors de l'appel, l'instruction *jsr* sauvegarde en sommet de pile l'adresse de retour et le pointeur de pile est diminué de la taille de l'adresse. A l'entrée de la procédure, une réservation de 2 octets est faite en sommet de pile pour accueillir la variable k et la base de la zone mémoire de stockage dynamique est mémorisée dans le registre A6.

```
int a,b,c;
                _a:       ds.b  2
                _b:       ds.b  2
                _c:       ds.b  2
int som (a, b)
{
                _som:     link    A6,#-2
int k;
k= a+b;
                   move.w  8(A6),D0
                   add.w   10(A6),D0
                   move.w  D0,-2(A6)
return k;
                   move.w -2(A6),D0
}                  unlk     A6
                   rts
main ()
{               _main:
                   link     A6,#0
a = 2;
                   move.w  #2,_a
b=3;
                   move.w  #3,_b
c = som (a,b); ----------------------
                   move.w  _b,-(A7)
                   move.w      _a,-(A7)
                   jsr     _som
                   add.l   #4,A7
                   move.w  D0,_c
}                  unlk     A6
                rts
```

Figure 4–20 Som « 68000 ».

Dans le corps de la procédure, les variables et paramètres peuvent maintenant être référencées par rapport à cette base fixe dans la pile. Examinons maintenant la préparation de l'addition. L'instruction move.w 8(A6) met *a* dans le registre D0. A6 vaut 1008 et le déplacement de 8 fait remonter dans la pile à l'adresse 1016 qui est l'adresse où *a* a été empilée. L'instruction suivante fait l'addition à D0 du contenu de (A6 + 10) qui est l'adresse de la valeur de *b* dans la pile. Le résultat est rangé dans la variable locale (à -2 par rapport à A6). La procédure est construite comme une fonction et renvoie sa valeur

	$@_{10}$	contenu pile	
SP anc	1020	----------	
move$_w$ b, -(A7)	1019	valeur de b	
	1018	valeur de b	←@dépil. de b
move$_w$ a, -(A7)	1017	valeur de a	
	1016	valeur de a	← @dépil. de a
jsr _som	1015	@ retour	
	1014	@ retour	
	1013	@ retour	
	1012	@ retour	← @dépil. @ret
Link A6, #-2	1011	A6	Sauveg de A6
	1010	A6	
	1009	A6	
'@dépil. A6→'	1008	A6	et A6 ← 1008
	1007	local k	
SP nouv	1006	local k	← @ . de k

Figure 4–21 Som « 68000 », la pile

via le registre D0 (au retour de la procédure *som*, la procédure *main* transfère le contenu de D0 à l'adresse de la variable *c*.)

La dernière instruction, *Unlink*, libère la place réservée pour la variable locale : le pointeur de pile est remis à 1012, c'est-à-dire à la valeur qu'il avait avant cette réservation. L'instruction rts récupère l'adresse de retour dans la pile et remonte le pointeur de pile à sa valeur d'avant l'appel de procédure (1016).

Dernier point : quel est le rôle de l'instruction add.l #4, A7 ? On remarque qu'avant l'appel à la procédure, il a fallu réserver 4 octets dans la pile pour le passage des paramètres *a* et *b*. L'ajout de 4 à A7 revient à remonter le pointeur de pile de 4, ce qui libère la place correspondante. Cette libération est indispensable, sinon, à chaque appel de la procédure la pile croît d'autant jusqu'à son débordement (*stack overflow*) par rapport à la taille initialement réservée pour elle.

Version Pentium Intel.

Le modèle de programmation du Pentium est, à quelques différences mineures près, celui du processeur 386, premier 32 bits de la famille de processeurs x86 d'Intel. Ce modèle est relativement complexe[4] : les registres sont spécialisés pour tenir compte de la gestion par segmentation de la mémoire. À ce stade, seuls les registres et les instructions requis pour la compréhension de l'exemple sont décrits.

Les registres dits 'généraux' sont des héritages en longue filiation du 8080 : les registres 8 bits *A*, *B* et *C* sont devenus les registres 16 bits *AX*, *BX* et *CX* du 8086 (x comme extension) puis *EAX*, *EBX* et *ECX* (e comme extension) 32 bits du 386. Ces registres servent aux opérations classiques, mais ne sont pas tous indifférents vis-à-vis des instructions qui les utilisent. Le registre *ESP* est le pointeur de pile et le registre *EBP* (*base pointer*) est le pointeur de base dans la pile pour référencer les variables locales. *EIP* (extended instruction pointer) est le compteur de programme.

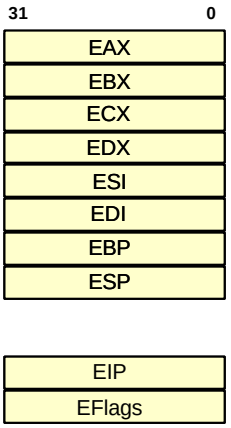


Figure 4–22 Registres du Pentium.

Ces processeurs ont des instructions spécifiques pour la manipulation de la pile : *push* est l'instruction empiler et *pop* celle qui dépile.

Par rapport au 68000, les paramètres sources et destinations sont inversés[5]. Une instruction *mov*

[4] Les processeurs de la famille x86, du 80386 au Pentium, font partie d'une architecture maintenant appelée IA32 par Intel, sont l'aboutissement (et probablement la fin) de l'architecture CISC. Le processeur va jusqu'à contenir des éléments du noyau d'un système d'exploitation.

[5] Les deux constructeurs Intel et Motorola ont souvent pris de conventions inverses l'une de l'autre : représentation little endian, modèle de mémoire segmenté, opérande destination

arg1_dest, arg2_src se lit comme 'transférer dans arg1 depuis arg2.

Dans la procédure *main*, le compilateur a réservé 12 emplacements pour les 3 variables *a*, *b* et *c* (3 entiers 32 bits). Les variables *a* et *b* sont mises à 2 et à 3 comme dans le programme du 68000. Le code binaire, rajouté comme un commentaire dans le programme, montre que la constante 2 est représentée en little endian, c'est-à-dire 02 00 00 00. Pour notre lecture classique des entiers, il faut intervertir les 2 mots de 16 bits et les 2 octets à l'intérieur du mot. Les paramètres a et b sont passés par valeur dans la pile : copie depuis la zone locale vers le registre *EAX* puis *EAX* est copié dans la pile.

```
_main   PROC NEAR
        push    ebp
        mov     ebp, esp
        sub     esp, 12

        mov     DWORD PTR -4[ebp], 2
;00006 c7 45 fc 02 00 00 00  ;
        mov     DWORD PTR -8[ebp], 3
        mov     eax, DWORD PTR -8[ebp]
        push    eax
        mov     ecx, DWORD PTR -4[ebp]
        push    ecx
        call    _som
        add     esp, 8
        mov     DWORD PTR -12[ebp], eax

        mov     esp, ebp
        pop     ebp
        ret     0

_som    PROC NEAR
        push    ebp
        mov     ebp, esp
        sub     esp, 4

        mov     eax, DWORD PTR 8[ebp]
        add     eax, DWORD PTR 12[ebp]
        mov     DWORD PTR -4[ebp], eax

        mov     eax, DWORD PTR -4[ebp]
        mov     esp, ebp
        pop     ebp
        ret     0
```

Figure4–23 som « pentium ».

Après l'appel de la procédure *som*, la place utilisée dans la pile pour les deux arguments est libérée en additionnant 8 au pointeur de pile. Le résultat de la fonction est passé par le registre *EAX* et ensuite transféré dans la variable *c*. La valeur initiale du pointeur de pile, mémorisée au début dans *EBP*, est restituée ; *EBP* est ensuite dépilé pour retrouver sa valeur initiale. La procédure *main* effectue ainsi son retour au programme appelant, c'est-à-dire le système d'exploitation.

Le même mécanisme de gestion de la pile est aussi utilisé dans le corps de la procédure *som*. Les paramètres sont passés par la pile : ils sont référencés en positif par rapport à *EBP* qui est la copie du pointeur de pile en entrée de procédure. La variable locale est dans la nouvelle zone allouée est et donc référencée en négatif (-4) par rapport à la base *EBP*. Le résultat de l'addition est retourné par le registre *EAX*.

Le principe général reste très voisin de celui vu, en détail, pour le 68000.

---

depuis source, ordre de priorité par nombre décroissant pour Intel ; big endian, modèle de mémoire linéaire, opérande source vers destination, ordre de priorité par nombre décroissant pour Motorola, …

La version MIPS.

Le processeur MIPS privilégie les registres par rapport à la pile (mémoire). Les registres sont visibles de la procédure appelante et de la procédure appelée. Autrement dit, les procédures partagent les mêmes registres qui jouent le rôle de variables globales. Cette situation, par ailleurs valable pour tous les processeurs, impose que la procédure 'appelée' sauvegarde les registres qu'elle modifie de manière à ce que l'appelante puisse récupérer les registres dans l'état où elle les avait laissés. Cette technique, si elle est sûre, peut amener à faire beaucoup de sauvegardes inutiles. Dans l'assembleur du MIPS, une convention d'utilisation des registres a été définie pour répartir le travail de sauvegarde entre l'appelante et l'appelée. Le respect de ces conventions permet de compiler séparément les procédures. Attention : cette convention n'est pas une propriété du processeur, mais une caractéristique d'un compilateur ou d'une famille de compilateurs.

- Le passage de paramètres : l'appelante passe les 4 premiers paramètres de la procédure appelée par les registres $a0 à $a3 (R4-R7). S'il y a plus de paramètres, il faut les passer par la pile. Les registres $v0 et $v1 (R2, R3) servent à retourner les valeurs des procédures fonctions.
- Les variables temporaires de l'appelée sont mises dans les registres $t0 à $t9. L'appelée utilise comme elle veut ces registres : si nécessaire, ils sont sauvegardés par l'appelante.
- Les registres $s0 à $s7 sont sauvegardés, le cas échéant, par l'appelée.
- Le registre $sp (R29) est le pointeur de pile et $bp (R30) le pointeur de cadre (ou de bloc) utilisé en général pour les variables locales.

```
som:
     subu    $sp,$sp,32
     sw      $fp,8($sp)
     move    $fp,$sp
     sw      $a0,16($fp)
     sw      $a1,20($fp)
     lw      $v1,16($fp)
     lw      $v0,20($fp)
     addu    $v0,$3,$2
     sw      $v0,0($fp)
     lw      $v0,0($fp)
     move    $sp,$fp
     lw      $fp,8($sp)
     addu    $sp,$sp,32
     j       $ra

main:
     subu    $sp,$sp,40
     sw      $ra,36($sp)
     sw      $fp,32($sp)
     move    $fp,$sp
     li      $v0,12
     sw      $v0,A
     li      $v0,13
     sw      $v0,B
     lw      $a0,A
     lw      $a1,B
     jal     som
     sw      $v0,C
     move    $sp,$fp
     lw      $ra,36($sp)
     lw      $fp,32($sp)
     addu    $sp,$sp,40
     j       $ra

     .comm   A,4
     .comm   B,4
     .comm   C,4
```

Figure 4–24 som « MIPS ».

Le programme.

Au premier abord, le programme est nettement plus long, au sens où il comporte bien davantage de lignes d'instruction que les versions CISC précédentes. Les procédures *main* et *som* ont des prologues et des épilogues pour la gestion de la pile en entrée et en sortie de procédure. La décrémentation du pointeur de pile n'est automatique : $sp est

décrémenté de 40 pour préparer les sauvegardes et la place pour les variables locales dans le bloc pointé par $fp. L'adresse de retour et le pointeur de cadre sont sauvegardés. On remarque que l'adresse de retour n'est pas sauvegardée dans le prologue de *som* : la procédure est une procédure dite « feuille », c'est-à-dire qu'elle ne contient pas d'appel de procédure et le registre $ra n'a pas besoin d'être sauvegardé.

Les corps des procédures *main* et *som* sont assez simples à lire. Les données sont transmises et récupérées par les registres $a0 et $a1 et le résultat de la somme est retourné par le registre $v0. On notera également, qu'à la fois dans *main* et dans *som*, il y a manifestement du code inutile. Ceci est du au fait que le compilateur a fait une traduction standard sans la moindre optimisation.

La version SPARC.

Le SPARC est un processeur RISC conçu, dans les grandes lignes, suivant des principes voisins à ceux du MIPS. Beaucoup d'instructions sont donc très semblables. L'originalité du SPARC réside dans la manière dont gérés les registres, en particulier pour les appels de procédures. Partant d'hypothèses déjà données sur le nombre moyen de paramètres en entrée et en sortie d'une procédure et sur le niveau de profondeur moyen d'appels de procédure, le SPARC offre à chaque procédure, lors de son appel, un ensemble de 32 registres 'logiques' une fenêtre (fenêtre glissante) sur un ensemble plus conséquent (par exemple 128) de registres.

Sur les 32 registres vus par une procédure, il y a :

- • 8 registres globaux, %g0 à %g7, qui sont vus par toutes les procédures. Ils peuvent contenir des variables globales. Ces registres globaux sont donc pour ainsi dire dans une fenêtre fixe. %g0 est câblé à la valeur 0.
- Les 24 autres registres constituent la fenêtre glissante et qui va changer d'un appel de procédure à l'autre. L'objectif recherché par ce mécanisme de fenêtrage est triple :
  isoler et protéger les variables locales propres à une procédure ;
  permettre une communication pour passer des paramètres entre les procédures et
  garder le lien dans les appels pour le retour à l'appelant.

Les 24 registres se répartissent en trois catégories, certains parmi eux jouent un rôle particulier.

- 8 registres d'entrées, %i0 à %i7, destinés à contenir les paramètres entrants de la procédure à laquelle est associée la fenêtre. Le registre %i0 est en général aussi utilisé pour retourner une valeur dans le cas d'une procédure appelée de type fonction.
- 8 registres locaux, %l0 à %l7, destinés à contenir les variables locales de la procédure à laquelle la fenêtre est associée.
- 8 registres de sortie, %o0 à %o7, destinés à contenir les paramètres sortants de la procédure. Le registre %o6 est le pointeur de pile, désigné également par

%sp, courant de la procédure appelante. Dans la procédure appelée, il s'appelle %i6 devient le pointeur de cadre %fp de la procédure appelée. fp est le pointeur de base servant de référence dans la pile pour la procédure courante (équivalent du registre intervenant dans l'instruction link du 68000).

La fenêtre glissante procède par recouvrement : 8 registres sont en recouvrement pour deux procédures consécutives de manière à ce que les paramètres sortants, %o0 à %o7, de la procédure appelante soient en correspondance avec les paramètres entrants, %i0 à %i7, de la procédure appelée.

Lorsqu'une fenêtre est en position pour une procédure donnée et définie dans le registre CWP (Current Window Pointer), l'appel d'une nouvelle procédure fait glisser la fenêtre de 16 registres, laissant 8 registres physiques en commun et en 'protégeant' les variables locales de la procédure appelante en les rendant invisibles. Le registre CWP est modifié en conséquence.

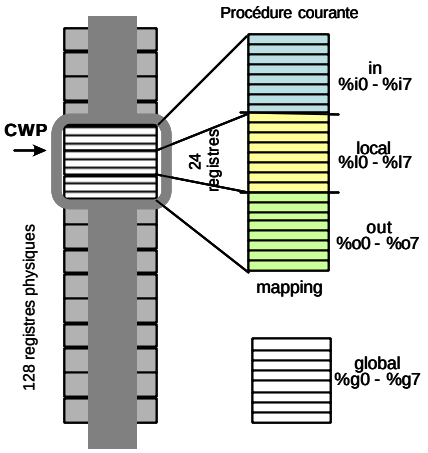


Figure 4–25 Fenêtre glissante.

Instructions d'appel de procédure.

D'une manière générale, les instructions de saut du SPARC sont 'retardées' : elles sont suivies d'une instruction vide *nop*. C'est le bon fonctionnement du pipeline du processeur qui impose cette règle de programmation. Ce point sera repris dans le chapitre 8.

Les instructions de gestion des procédures ont des particularités liées au mécanisme de fenêtre glissante. L'instruction *call label* réalise un saut inconditionnel à l'étiquette *label*, mais avant que le saut ne soit réellement fait, la valeur courante (celle de l'instruction *call*) du compteur de programme %pc est mémorisée, non pas dans la pile, mais dans le registre %o7 de la fenêtre en cours au moment de l'appel.

L'instruction *ret* n'existe pas (sauf sous la forme d'une macro) et le retour se fait par une instruction de saut *jmpl* (*JuMP and Link*) avec lien

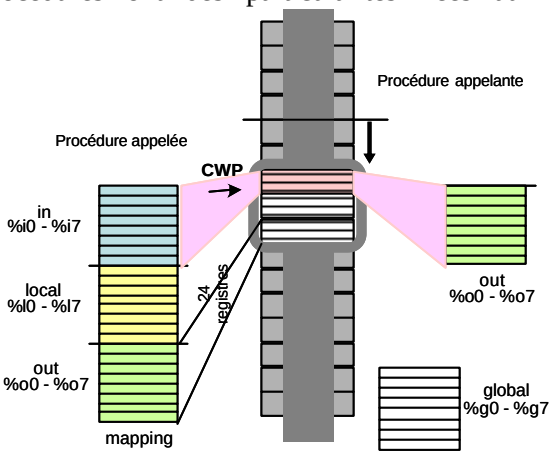


Figure 4–26 Fenêtre de registres SPARC.

réalise un saut à l'adresse indiquée en paramètre et recopie la valeur courante de %pc dans le registre de destination : *jmpl adresse, regdest*. La macro *ret* trouvé dans un programme en assembleur SPARC est équivalente à *jmpl [%i7 + 8], %g0* . Au moment de l'appel, la valeur courante du compteur de programme est copiée dans %o7 et au moment du retour cette valeur est accessible dans %i7. La valeur +8 provient du fait que la valeur de %pc sauvegardée est celle de l'instruction d'appel (et non l'instruction suivante comme dans les CISC) et qu'une instruction de saut, y compris un call, est suivie par une instruction nop. La prochaine instruction à exécutée au retour est donc à +8 par rapport à l'appel. *jmpl* mémorise l'adresse courante (dernière instruction de la procédure appelée) dans %g0, mais n'étant pas utilisée elle pourra être perdue.

Lors d'un appel de procédure, rien n'est fait automatiquement, en dehors du saut et la mémorisation du %pc courant. Il faut explicitement effectuer le glissement de la fenêtre et revenir à la situation initiale avant le retour à l'appelant.

L'instruction *save %sp, taille, %sp* est équivalente à une addition de la taille au registre %sp avec en parallèle une décrémentation du registre CWP de 1. Ce déplacement de la fenêtre revient à 'sauver' les registres de l'appelant. La modification de %sp décale d'autant le sommet de pile et correspondant à une réservation de place *'taille'* mots dans la pile pour d'éventuelles sauvegardes. Le premier paramètre %sp est le pointeur de pile de la procédure appelante. Au cours de l'exécution de l'instruction *save* le pointeur de pile change : le dernier paramètre %sp est de fait le pointeur de pile dans la fenêtre de la nouvelle procédure. L'instruction *restore* fait l'opération inverse de *save*. La position de cette instruction est dans le 'delay slot' c'est à dire juste après l'instruction de saut *ret* (imposé par le pipeline).

L'instruction sethi.

Toutes les instructions font 4 octets, cela pose donc des problèmes pour manipuler dans certains cas un adressage de 32 bits. L'adresse 32 bits est décomposée en deux parties : la partie haute de 22 bits et la partie basse de 10 bits. Les opérateurs *hi* (*high*) et *lo* (*low*) permettent d'extraire ces deux champs à partir d'une adresse complète. L'instruction *sethi* permet de positionner à une valeur donnée les 22 bits d'un registre destination.

Le programme.

Le programme *main* est une procédure qui se voit affectée une fenêtre de registre avec une réservation de place dans la pile pour d'éventuelles sauvegardes. Ce sont les trois instructions *sethi, mov,* st qui effectuent cette tâche. Les 22 bits de poids fort de l'adresse de *a* sont mis dans %o0 par l'instruction *sethi,* les 10 bits de poids faible sont à 0. La valeur 2 est rangée dans le registre dans le registre %o1 par l'instruction *mov*. L'instruction *st* range le contenu du registre %o1 (valeur 2) à l'adresse pointée par le registre %o0 auquel il faut ajouter les 10 bits de poids

faible de l'adresse de *a*. Le même travail est effectué pour l'affectation de la valeur 3 à *b*.

Les 4 instructions qui suivent font le passage de paramètres par les registres out (%o0 et %o1) de la fenêtre du *main*, registres qui deviendront les registres *in* (%i0 et %i1) de la procédure appelée *som*. Le passage de paramètre est fait par pointeur : ce sont les adresses des variables *a* et *b*. La technique de dépôt d'une adresse dans un registre est la même que pour les affectations précédente (*sethi*).

L'instruction *call* doit être suivie d'une instruction vide (*nop*). Au retour de l'appel, la procédure fonction renvoie la valeur du résultat dans le registre %o0, valeur que la procédure *som* aura mise dans son registre %i0. Le résultat de la somme est enfin rangé à l'adresse de la variable *c*.

```
som:
      save    %sp,-120,%sp
      st      %i0,     [%fp+68]
      st      %i1,     [%fp+72]
      ld      [%fp+68],%o0
      ld      [%fp+72],%o1
      add     %o0,%o1, %o0
      st      %o0,     [%fp-20]
      ld      [%fp-20],%o0
      mov     %o0,%i0
      ret
      restore

main:
      save    %sp,-112,%sp
      sethi   %hi(a),  %o0
      mov     2,       %o1
      st      %o1,     [%o0+%lo(a)]
      sethi   %hi(b),  %o0
      mov     3,       %o1
      st      %o1,     [%o0+%lo(b)]
      sethi   %hi(a),  %o0
      sethi   %hi(b),  %o1
      ld      [%o0+%lo(a)],%o0
      ld      [%o1+%lo(b)],%o1
      call    som
      nop
      sethi   %hi(c),  %o1
      st      %o0,     [%o1+%lo(c)]
      ret
      restore
```

Figure 4–27 som 'SPARC'.

Dans la procédure *som*, une nouvelle fenêtre de registre est affectée avec également une réservation de place dans la pile pour des variables locales ou d'éventuelles sauvegardes. La zone de variables locales est référencée par rapport au '*frame pointer*' %fp. Le compilateur se sert ainsi de deux emplacements dans la zone locale (%fp +68 et +72) pour récupérer les valeurs des variables à partir des adresses passées en paramètres et obtenues dans les registres in (%i0 et %i1). L'addition est faite, le résultat est mis dans %o0. Ce résultat est rangé dans la variable locale *k* dont l'adresse est référencée par rapport au pointeur de cadre *fp*, puis est mis dans le registre %i0 pour être récupérable par la procédure appelante après le *restore* et le *return* (l'interversion constatée des deux instructions est également imputable à la gestion interne du pipeline).

Exemple avec un compilateur qui optimise. Exemple Power PC G4

Pour terminer ce panorama de traduction de programma C vers un assembleur, nous illustrerons un travail d'optimisation de code fait par le compilateur. Tous les compilateurs modernes, quels que soient les processeurs peuvent réaliser des optimisations efficaces. Nous profitons de ce denier exemple pour présenter le bout de code obtenu pour le processeur PowerPC G4 (Motorola, IBM, base installée Mac Apple).

Comme pour la version MIPS, les registres sont largement utilisés. L'affectation des variables est faite comme dans les autres versions de programmes.

```
_main:
        mflr r3
        li r10,2
        bcl 20,31,L1$pb
L1$pb:
        li r0,3
        mflr r2
        li r4,5
        addis r11,r2,ha16(L_a$non_lazy_ptr-L1$pb)
        addis r7,r2,ha16(L_b$non_lazy_ptr-L1$pb)
        mtlr r3
        lwz r8,lo16(L_a$non_lazy_ptr-L1$pb)(r11)
        lwz r6,lo16(L_b$non_lazy_ptr-L1$pb)(r7)
        addis r5,r2,ha16(L_c$non_lazy_ptr-L1$pb)
        lwz r2,lo16(L_c$non_lazy_ptr-L1$pb)(r5)
        stw r10,0(r8)
        stw r0,0(r6)
        stw r4,0(r2)
        blr


_som:
        add r3,r3,r4
        blr
.comm _a,4
.comm _b,4
.comm _c,4
```

Figure 4–28 Registres généraux du processeur MIPS.

Par contre l'optimisation faite est assez intéressante : dans le *main* il n'y a plus d'appel à la procédure *som*. Le compilateur s'est aperçu d'une part que les deux valeurs passées à la procédure sont des constantes (2 et 3) et, d'autre part que la procédure *som* ne fait que l'addition de ces deux paramètres.

En conclusion, le compilateur affecte la valeur 5 directement, ou dit autrement : le programme ne sert à rien…

Little/Big Endian ***Boutien ou boutiste : on n'a pas fini d'en voir le bout.***

Les termes *Little* et *Big Endian* furent utilisées avec une consonance ironique dans un article resté célèbre « Les guerres saintes et un plaidoyer pour la paix » ('On Holy Wars and a plea for Peace') de Danny Cohen et publié en 1980 dans la revue IEEE Computer. D. Cohen traite du problème du positionnement d'un octet dans un mot lorsque celui-ci fait l'objet d'une transmission ou d'un lecture/écriture en mémoire : dans un octet quel bit est transmis en premier, dans un mot quel octet est transféré en premier.

Il y a priori deux choix possibles, choix qui, d'une manière ou d'une autre, sont plus ou moins équivalents. Pour mettre en évidence, à la fois la futilité et l'importance des conséquence d'un choix big endian et little endian, D. Cohen fait une référence aux 'Voyages de Gulliver' de Jonathan Swift (1726). Aux pays de Lilliput, les lilliputiens sont des gens de petite taille et ont des problèmes politiques assortis à cette taille. Pourtant une guerre civile féroce se déclenche à la suite d'une décision aux conséquences dramatiques. Les lilliputiens consomment des œufs cuits qu'ils ouvrent indifféremment du coté 'large' de l'œuf ou du 'petit coté'. L'empereur des lilliputiens proclame que dorénavant les oeufs doivent obligatoirement être ouverts du petit coté (little end). Réaction immédiate des farouches défenseurs du gros coté (les big endians) et la situation s'envenime jusqu'à engendrer 11000 morts. Les « big endians » durent se réfugier sur une île voisine du royaume de Blefuscu. Jonathan Swift fait une allusion à la situation politique de l'époque en Angleterre avec la guerre de religion entre l'église anglicane et l'église catholique ; Lilliput représente l'Angleterre et Blefescu est évidemment la France… La fable de Swift est peut être toujours d'actualité !

Pour 'simplifier' les choses, il y a deux versions de traduction française de endian : boutien et boutiste. Nous parlons donc de '*gros et petit boutien*' dans un camps et de '*gros et petit boutiste*' dans l'autre. Pour notre part, nous garderons les termes proposés par Danny Cohen... même s'ils sont en anglais.

Pour l'anecdote, notons encore une variante, probablement due à des difficultés d'audition et/ou de perception de l'anglais, qui parle de *indian* en lieu et place de endian en prétextant que les lilliputiens étaient des tribus indiennes.

Pour finir, nous pouvons remarquer que cette problématique n'est ni nouvelle ni spécifique à l'informatique et doit trouver son origine dans la transcription des nombres. Deux exemples. Dans la représentation textuelle des nombres, anglais et français sont big endian : 21 se dit 'vingt et un', 'twenty one' alors que les allemands sont little endian : 21 se dit 'ein und zwanzig'.

Au niveau des dates, les européens sont Little Endian avec le format jj/mm/aa, les japonais sont Big Endian (aa/mm/jj) et les américains sont … Middle Endian (mi-boutien ou mi-boutiste) avec le format mm/jj/aa. …

Table des matières