Organisation de système
Chapitre 3

Même pour écrire un programme des plus modestes sur microprocesseurs de la famille 80x86, il faut avoir une familiarité considérable avec ces microprocesseurs. Ecrire un *bon* programme requiert une forte connaissance des composants matériels sous-jacents. Malheureusement, ces derniers ne sont pas homogènes. Des techniques cruciales sur le 8088 pourraient ne pas servir sur des systèmes 80486. De la même façon, des programmes optimaux sur 80486 pourraient ne produire que de piètres performances sur 80286. Heureusement, certaines techniques de programmation restent valables malgré les différences hardware. Ce chapitre traitera les effets que le matériel peut avoir sur les performances logicielles.

3.0 Vue d'ensemble du chapitre

Ce chapitre décrit les composants de base qui constituent un système informatique : le CPU, la mémoire, les entrées/sorties et les bus qui connectent toutes ces choses ensemble. Certes, on peut écrire des programmes en ignorant ces concepts, cependant des programmes à hautes performances requièrent une complète compréhension de ces derniers.

Ce chapitre commence sur l'organisation des bus et de la mémoire. Ces deux composants auront probablement un grand impact sur la performance, encore plus que la vitesse du CPU elle-même. Le fait de comprendre comment le système des bus est organisé vous aidera à concevoir des structures de données parfaitement performantes. De même, connaître les caractéristiques des performances de la mémoire, la manière dont les données sont localisées et l'opération de la mémoire *cache*, pourra vous aider à concevoir des logiciels qui s'exécutent aussi rapidement que possible. Si vous n'êtes pas intéressé à écrire un code ultra rapide vous pouvez sauter cette section ; cependant, la plupart des programmeurs finissent toujours par se préoccuper de la vitesse, donc, cette lecture ne sera pas inutile.

Malheureusement, la famille des microprocesseurs 80x86 est assez complexe et déboussole facilement les étudiants débutants. C'est pourquoi, ce chapitre se consacre à décrire dans le détail quatre membres hypothétiques de la famille 80x86 : les microprocesseurs 886, 8286, 8486 et 8686. Ces derniers représentent des versions simplifiées des puces (chips en anglais) 80x86 et permettent de traiter, sans vous enliser, les diverses caractéristiques architecturales de l'énorme ensemble d'instructions CISC. Dans ce chapitre on fera usage de processeurs hypothétiques, tels que les x86, pour décrire les concepts de l'encodage des instructions, des modes d'adressage, des instructions séquentielles, des queues de préchargement (*prefetched queues*), du pipelining et des opérations superscalaires. Encore une fois, il s'agit de concepts que vous n'avez pas besoin d'apprendre si vous voulez juste écrire du code correctement. Mais si vous voulez écrire des logiciels rapides, surtout en parlant des processeurs avancés comme le 80486, Pentium et au-dessus, vous devez impérativement acquérir ces notions.

Ce chapitre peut paraître comme allant trop loin du côté architectural, il vous donnera peut-être la sensation que de telles notions devraient se trouver dans des d'ouvrages spécifiques, plutôt que dans un livre de programmation. C'est bien loin de la vérité ! Ecrire de *bons* programmes en assembleur demande une maîtrise étendue des concepts de l'architecture matérielle. D'où l'importance qu'on lui donne dans ce chapitre.

3.1 Les composants de base du système

On nomme *architecture* la conception opérationnelle de base d'un ordinateur. C'est dans l'architecture de John von Neumann, un pionnier dans ce domaine, qui se basent la plupart des machines actuelles. Par exemple, la famille 80x86 utilise l'architecture von Neumann (VNA). Un système ainsi conçu comporte trois majeurs composants : l'*unité de traitement central* (CPU), l'*unité de mémoire* et l'*unité des entrées/sorties* (abrégé en E/S ou I/O en anglais). La façon dont un concepteur combine ces composants détermine l'impact sur les performances du système (voir fig. 3.1).

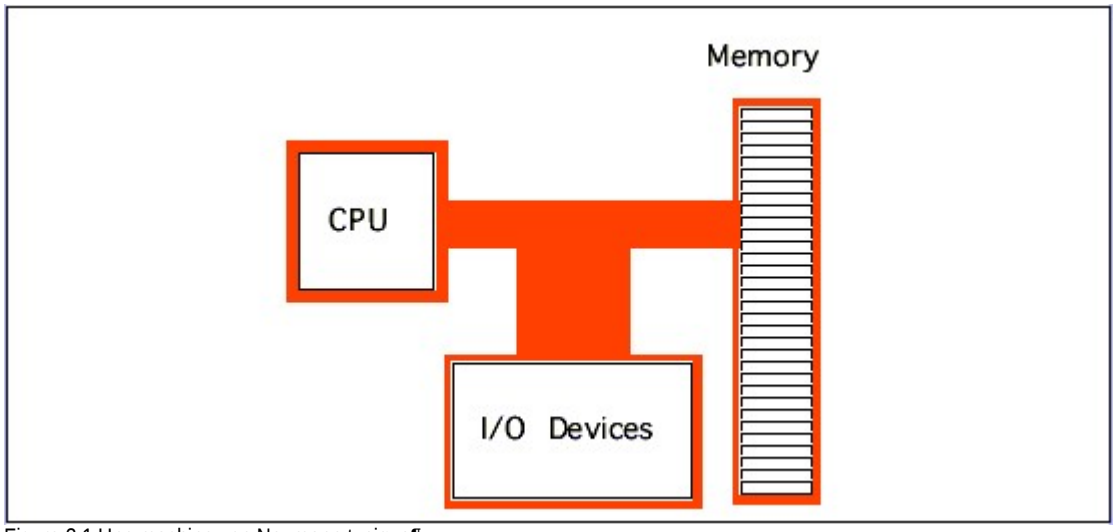


Figure 3.1 Une machine von Neumann typique¶

Dans des machines VNA, comme la catégorie 80x86, toute l'action se déroule dans le CPU. C'est là où ont lieu les opérations majeures. Les données et les instructions du CPU résident en mémoire avant que le microprocesseur ne les réclame. Du point de vue du CPU, la plupart des périphériques d'entrée/sortie ressemblent à la mémoire, car le CPU peut placer des données sur un périphérique de sortie et en lire sur un périphérique d'entrée. La différence principale entre la mémoire et les ports d'entrées/sorties est le fait que ces derniers sont associés à des périphériques du monde extérieur.

3.1.1 Les bus système

Les *bus système* connectent les diverses composantes d'une machine VNA. La famille 80x86 a trois bus principaux : le *bus des adresses*, le *bus de données* et le *bus de contrôle*. Un bus est une collection de fils dans lesquels des signaux électriques passent entre les composants du système. La nature de ces bus varie selon le processeur. Cependant, chaque type de bus transporte des informations de manière semblable sur tous les microprocesseurs ; par exemple, chaque bus de données peut avoir une implémentation différente selon qu'il s'agisse du 80386 ou du 8088, mais les deux font passer les données entre le CPU, les entrées/sorties et la mémoire.

Une composante ordinaire d'un système 80x86 utilise ce qu'on nomme le *standard TTL logic level*. Ceci veut dire que chaque fil dans un bus utilise un voltage standard pour représenter la valeur 0 ou la valeur 1[1]. On spécifiera toujours l'une de ces deux valeurs au lieu du voltage qu'elles représentent en vertu du fait que ces niveaux varient d'un ordinateur à l'autre (spécialement si on parle d'ordinateurs portables).

3.1.1.1 Le bus de données

Les processeurs 80x86 utilisent le *bus de données* pour transporter les données entre les différentes parties d'un ordinateur. La taille de ce bus varie largement au sein de la famille 80x86. En effet, elle définit *de facto* la «taille du processeur».

La "taille" du processeur

Il y a beaucoup de discussions dans le milieu des ingénieurs de matériel et logiciels à propos de la taille d'un processeur comme le 8088. D'une perspective de concepteur matériel, le 8088 est purement un processeur de 8 bits - il a seulement 8 lignes de données et il est compatible avec la mémoire et les E/S conçues pour des processeurs de 8

[1]La logique TTL représente la valeur zéro avec un voltage dans la plage 0,0 – 0,8 volts. Et elle représente la valeur 1 avec un voltage allant de 2,4 à 5 v. Si le signal d'une ligne de bus est entre 0,8 et 2,4 volts, sa valeur est indéterminée. Une telle condition doit exister seulement quand le bus se trouve entre un état et l'autre.

bits. Par contre, les ingénieurs de logiciels ont soutenu que le 8088 est un processeur de 16 bits. De leur perspective, il n'y a pas de distinction entre le 8088 (avec un bus de données de 8 bits) et le 8086 (avec un bus de données de 16 bits). En effet, la seule différence est la vitesse avec laquelle les deux processeurs travaillent ; le 8086 avec un bus de 16 bits est plus rapide. Finalement, les concepteurs de matériel ont eu raison. Malgré le fait que les ingénieurs de logiciels ont du mal à faire cette distinction dans leurs programmes, on appelle le 8088 un processeur de huit bits et le 8086 un processeur de 16 bits. De même, le 80386SX (qui a un bus de données de 16 bits) est un processeur 16 bits, alors que le 80386dx (ayant un bus de données de 32 bits) est un microprocesseur de 32 bits.

Dans un système 80x86, le bus de données contient huit, 16, 32 ou 64 lignes. Les microprocesseurs de type 8088 et 80188 ont un bus de données de huit bits (huit lignes de données). Alors que les microprocesseurs 8086, 80186, 80286 et 80386SX ont des bus de données de 16 bits. Les processeurs 80386DX, 80486 et Pentium Overdrive ont des bus de 32 bits. Enfin, les processeurs Pentium[tm] et Pentium Pro ont des bus de 64 bits. Des versions futures de la puce (80686/80786 ?) pourront avoir un bus encore plus large.

Le fait d'avoir un bus de données de huit bits ne limite pas le processeur à des types de données de huit bits. Il veut simplement dire que le processeur peut accéder seulement à un octet de données par cycle de mémoire (voir « Le processeur » au paragraphe 3.1.2). Donc les 8 bits d'un 8088 peuvent uniquement transmettre la moitié de l'information par unité de temps (cycle mémoire) par rapport aux 16 bits du 8086. Par conséquent, des processeurs avec des bus 16 bits sont évidemment plus rapides. Et les processeurs avec un bus de 32 bits le sont encore plus. La taille d'un bus de données affecte donc la performance du système plus que tout autre bus.

Vous entendrez souvent parler de processeurs de 8, 16, 32 ou 64 bits. Malgré qu'il y ait une légère controverse à propos de la taille du processeur, la plupart des gens agréent maintenant que cette taille soit déterminée par le nombre de lignes de données. Etant donné que les bus de la famille 80x86 mesurent 8, 16, 32 ou 64 bits il en résulte que l'accès aux données correspond aussi à un adressage de 8, 16, 32 ou 64 bits. Quoiqu'il soit même possible de traiter des données de 12 bits dans un 8088, la plupart des programmeurs travaillent avec l'adressage de 16 bits, car le processeur chargera et manipulera des blocs de 16 bits dans tous les cas. Ceci parce que le microprocesseur adresse toujours par blocs de 8 bits. Alors que récupérer 12 bits de la mémoire requiert deux opérations de mémoire. Etant donné que le processeur obtiendra à la fin 16 bits au lieu de 12, la plupart des programmeurs utilise les 16 bits au complet. En général, manipuler des données avec une longueur de 8, 16, 32 ou 64 bits est plus efficace.

Si les membres de la famille 80x86 avec 16, 32 et 64 bits peuvent traiter des données qui dépassent la taille du bus, ils peuvent également accéder aussi à de plus petites portions de mémoire, par exemple de 8, 16 ou 32 bits. Cependant, tout ce qu'on peut faire avec un petit bus de données peut également être fait avec de plus larges bus ; avec la différence que ces derniers peuvent avoir accès à la mémoire de manière plus rapide et peuvent transporter de plus gros blocs de données en une seule opération de mémoire. Sur la nature exacte de ces types d'accès, vous lirez plus tard (voir "La mémoire" au paragraphe 3.1.2).

Tableau 17 : taille du bus de données sur les processeurs 80x86

processeur	Taille du bus de données
8088	8
80188	8
8086	16
80186	16
80286	16
80386sx	16
80386dx	32
80486	32
classe 80586 - Pentium (Pro)	64

3.1.1.2 Le bus des adresses

On a vu que le bus de données, dans une architecture 80x86, transmet des informations entre un emplacement de mémoire particulier (ou un périphérique d'E/S) et le CPU. La seule question est, *« Quel emplacement de mémoire ou d'adresse E/S ? »* Le bus des adresses répond à la question. Pour différencier entre emplacements de mémoire et les ports des périphériques d'E/S, le concepteur du système assigne une

valeur unique à chaque emplacement de la mémoire et d'élément d'E/S. Quand le programme veut accéder à un emplacement de mémoire particulier ou à un périphérique d'E/S, il place l'adresse correspondante dans le bus des adresses. Les circuits associés à la mémoire ou à des périphériques d'E/S reconnaissent l'emplacement et informent la mémoire ou le périphérique d'E/S de lire ou de placer la donnée sur le bus de données. Dans les deux cas, tout autre emplacement de mémoire ignore la demande. Seulement le périphérique dont l'adresse correspond à la valeur dans le bus des adresses répond.

Avec une simple ligne d'adresse unique, un processeur peut créer exactement deux adresses : zéro et un. Avec *n* lignes d'adresses, le processeur peut fournir 2^n adresses uniques (car il y a 2^n valeurs uniques dans un nombre binaire de *n* bits). Donc, le nombre de bits du bus des adresses déterminera le nombre *maximal* de la mémoire adressable et des emplacements d'E/S. Par exemple, les bus des adresses des processeurs 8088 et 8086 sont des bus de 20 bits. Par conséquent, ils peuvent adresser jusqu'à 1 048 576 (ou 2^{20}) emplacements de mémoire. De plus larges bus peuvent adresser plus d'emplacements. Les deux derniers processeurs cités, par exemple, souffrent d'un espace d'adressage très réduit[2] - leur bus des adresses est trop petit. Les microprocesseurs plus récents ont des bus plus larges :

Tableau 18 : le bus des adresses de la famille 80x86

processeur	Taille du bus des adresses	Maximum de mémoire adressable	En français !
8088	20	1 048 576	Un méga-octet
8086	20	1 048 576	Un méga-octet
80188	20	1 048 576	Un méga-octet
80186	20	1 048 576	Un méga-octet
80286	24	16 777 216	Seize méga-octets
80386SX	24	16 777 216	Seize méga-octets
80386DX	32	4 294 976 296	Quatre gigas-octets
80486	32	4 294 976 296	Quatre gigas-octets
80586 / Pentium (Pro)	32	4 294 976 296	Quatre gigas-octets

De futurs processeurs supporteront probablement des bus des adresses de 48 bits. Arrivera probablement l'époque où les programmeurs considéreront 4Go de mémoire RAM insuffisants, un peu comme, de nos jours, on considère qu'un méga-octet constitue un bien pauvre espace (mais il y a eu une époque où un Mo de ram était considéré bien plus que ce dont on aurait jamais besoin !). Heureusement, l'architecture des puces 80386, 80486 et ultérieures permet une facile expansion du bus des adresses à 48 bits, grâce à la *segmentation*.

3.1.1.3 Le bus de contrôle

Le bus de contrôle est une collection éclectique de signaux qui contrôlent la manière dont le processeur communique avec le reste du système. Considérez un moment le bus des données. Le CPU l'utilise pour envoyer des données à la mémoire et en recevoir. Ceci amène à la question : « est-il en train de recevoir ou d'envoyer ? ». Dans le bus de contrôle il y a deux lignes, *read* et *write* qui spécifient la direction du flux des données. D'autres signaux incluent les horloges système, les lignes d'interruption et ainsi de suite. Le comportement exact du bus de contrôle varie selon les processeurs de la famille 80x86. Cependant, certaines lignes sont communes à toutes les machines et méritent une brève mention.

Les lignes de contrôle *read* et *write* gèrent la direction des données dans le bus. Quand les deux contiennent un 1 logique, le CPU et la mémoire-entrées/sorties *ne sont pas en train de communiquer entre eux*. Si la ligne de lecture est dans un état bas (0 logique), le CPU est en train de lire des données de la mémoire (c'est-à-dire, le système est en train de transmettre les données de la mémoire au CPU). Si la ligne d'écriture est dans un état bas, alors le système sera en train de transmettre des données du CPU à la mémoire.

Les *lignes de validation des octets* (*byte enable lines*) forment un autre ensemble de lignes de contrôle importantes. Ces dernières permettent aux processeurs de 16, 32 ou 64 bits de distribuer de plus petites portions de données. De plus amples détails apparaîtront dans la prochaine section.

La famille 80x86, contrairement à d'autres catégories de processeurs, fournit deux espaces d'adressage distincts : l'un pour la mémoire et l'autre pour les entrées/sorties. Si la taille des adresses mémoire dans les bus des adresses des différents processeurs 80x86 peut varier, la taille des emplacements pour les E/S est, dans le

[2] L'*espace adressable* est l'ensemble de tous les emplacements de mémoire adressables.

bus des adresses, toujours de 16 bits. Ceci permet au processeur de disposer jusqu'à 65 536 emplacements différents pour les entrées/sorties. Naturellement, la plupart des périphériques (comme les claviers, les imprimantes, les lecteurs, etc.), requièrent plus d'une adresse E/S. Néanmoins, 65 536 emplacements E/S sont plus que suffisants pour la plupart des applications. La conception de l'IBM PC original permettait seulement l'utilisation de 1 024 de ces adresses.

Quoique la famille 80x86 supporte deux espaces d'adressage, elle n'a pas deux bus des adresses (pour les E/S et la mémoire). Le système partage le bus pour les deux types. Des contrôles supplémentaires décident si une adresse indique une zone de la mémoire ou des entrées/sorties. Quand de tels signaux sont actifs, les périphériques d'E/S utilisent l'adresse du mot le moins significatif du bus des adresses. Quand ils sont inactifs, les périphériques d'E/S ignorent ces signaux (la mémoire prend alors le relais).

3.1.2 La mémoire

Un processeur ordinaire 80x86 peut adresser jusqu'à un maximum de 2^n emplacements de mémoire différents, où *n* est la taille en bits du bus des adresses[3]. Comme on a vu, les processeurs 80x86 peuvent avoir des bus des adresses 20, 24 et 32 bits (avec des bus 48 bits à venir).

Sans doute, la première question qui vous saute aux yeux serait : « Qu'est-ce que c'est exactement un emplacement de mémoire ? ». La famille 80x86 fonctionne avec une *mémoire adressable par octets*. Donc l'unité de base de la mémoire est l'octet. Alors, avec 20, 24 et 32 lignes d'adresses, les processeurs 80x86 peuvent adresser respectivement un méga-octet, seize méga-octets et quatre gigas-octets de mémoire.

Considérez la mémoire comme un tableau linéaire d'octets. L'adresse du premier octet est 0 et l'adresse du dernier est 2^n - 1. Pour un processeur 8088 avec un bus des adresses de 20 bits, la déclaration du tableau suivant écrite en pseudo Pascal constitue une bonne représentation :

```
Memory: array[0..1048575] of bytes;
```

Pour exécuter l'équivalent de l'instruction Pascal "Memory[125] := 0;" le CPU place la valeur zéro dans le bus de données, l'adresse 125 dans le bus d'adresse et teste la ligne d'écriture (étant donné que le CPU est en train d'écrire en mémoire), voir Figure 3.2.

Pour exécuter l'équivalent de "CPU := Memoire[125];", le CPU place la valeur 125 dans le bus des adresses et il test la ligne d'écriture (car il est en train d'écrire la mémoire), puis, il lit la donnée résultante depuis le bus de données (voir fig. 3.3).

L'explication qui précède s'applique *seulement* pour le traitement d'un seul octet. Mais, que se passe-t-il quand le processeur accède un mot ou un double-mot ? Puisque la mémoire consiste en un tableau d'octets, comment peut-on se débrouiller avec valeurs plus grandes que 8 bits ?

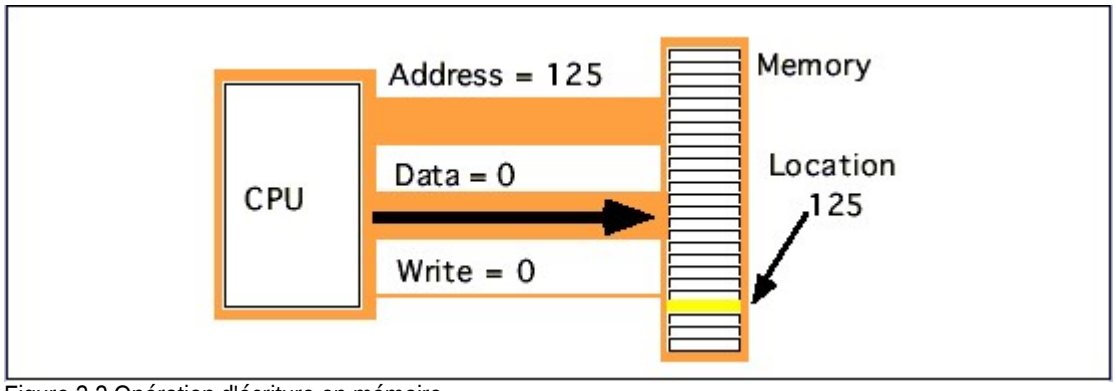


Figure 3.2 Opération d'écriture en mémoire

[3]Ceci est le maximum. La plupart des systèmes de la famille 80x86 ne sont pas pourvus d'une quantité de mémoire correspondante à cette borne.

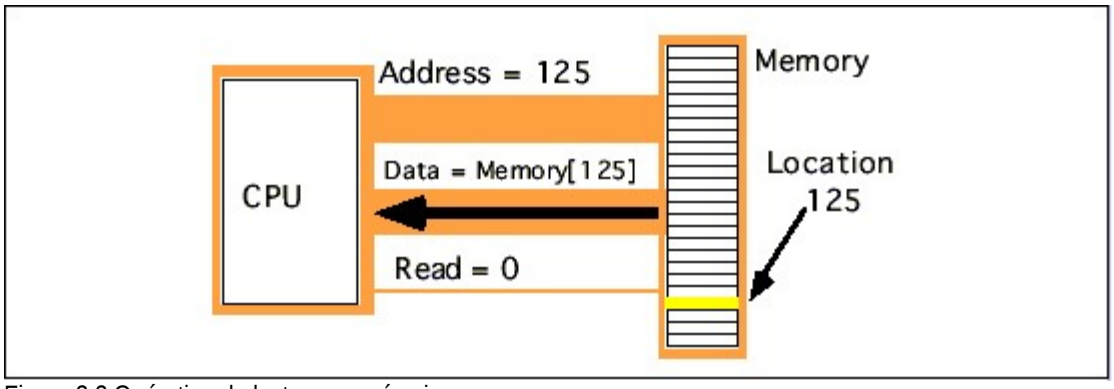


Figure 3.3 Opération de lecture en mémoire

Différents systèmes offrent différentes solutions à ce problème. La famille 80x86 le contourne en stockant l'octet le moins significatif du mot dans l'adresse spécifiée et l'octet le plus significatif dans l'emplacement suivant. Donc, un mot occupe deux adresses de mémoire consécutives (étant un mot composé de deux octets). De la même façon, un double-mot occupe quatre emplacements consécutifs. L'adresse du double-mot est l'adresse de l'octet le moins significatif. Les trois octets restants suivent l'octet le moins significatif avec l'octet le plus significatif correspondant à l'adresse du double mot plus trois (voir figure 3.4). Octets, mots et doubles-mots peuvent commencer à toute adresse valide en mémoire. Cependant, nous verrons bientôt que placer de gros objets à une adresse arbitraire n'est pas une bonne idée.

Notez que, dans la mémoire, des octets, des mots et des doubles-mots peuvent parfaitement se chevaucher. Par exemple, dans la figure 3.4, on peut avoir une variable d'un mot commençant à l'adresse 193, une variable d'un octet commençant à l'adresse 194, et une variable d'un double-mot commençant à l'adresse 192. Ces variables se chevauchent toutes.

Les microprocesseurs 8088 et 80188 ont un bus de données de 8 bits. Ceci signifie que le CPU ne peut transférer que huit bits à la fois. Etant donné que chaque adresse de mémoire correspond à un octet, cela résulte être le plus pratique des arrangements (d'un point de vue matériel), voir figure 3.5.

Le terme « tableau d'octets de mémoire adressable » signifie que le CPU peut adresser la mémoire en morceaux de 8 bits. Cela veut dire aussi que cette quantité est la plus petite unité de mémoire accessible d'un seul coup par le processeur. Autrement dit, si le processeur veut accéder à une valeur de 4 bits, il doit lire tout l'octet et ignorer les bits superflus. Veuillez considérer également que la capacité d'adresser par octets ne veut pas dire que le CPU peut adresser n'importe quel emplacement arbitraire de 8 bits. Quand vous spécifiez l'adresse 125 en mémoire, vous obtenez tout l'octet correspondant à cette adresse, rien de plus, rien de moins. Les adresses sont des entiers ; vous ne pouvez pas, par exemple, spécifier l'adresse 125,5 pour retrouver une quantité de bits inférieure à 8.

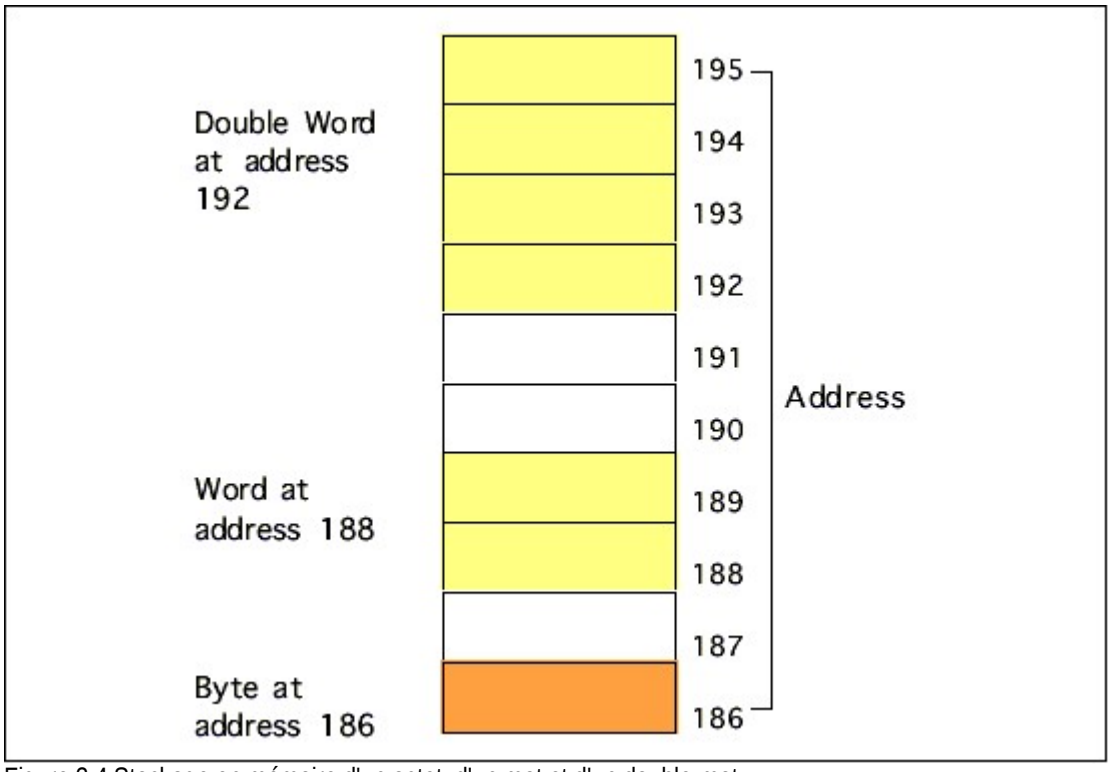


Figure 3.4 Stockage en mémoire d'un octet, d'un mot et d'un double-mot

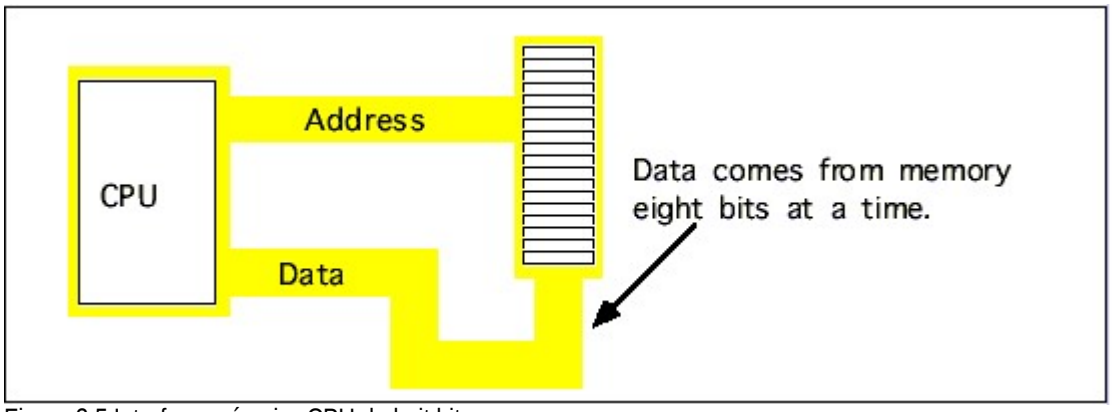


Figure 3.5 Interface mémoire-CPU de huit bits

Les processeurs 8088 et 80188 peuvent manipuler des valeurs d'un mot ou d'un double-mot, même avec leurs bus de huit bits. Cependant, ceci requiert plusieurs opérations de mémoire, car ces processeurs peuvent manipuler seulement un octet à la fois. Charger un mot demande deux opérations ; charger un double-mot en demande quatre.

Les processeurs 8086, 80186, 80286 et 30386SX ont des bus de données de 16 bits. Ceci leur permet d'accéder au double de mémoire dans le même laps de temps qu'avec les bus de 8 bits. Ces processeurs organisent la mémoire en deux banques paires et impaires (voir figure 3.6). La figure 3.7 illustre la connexion au CPU (D0-D7 se réfère à l'octet le moins significatif et D8-D15 à l'octet le plus significatif du bus de données).

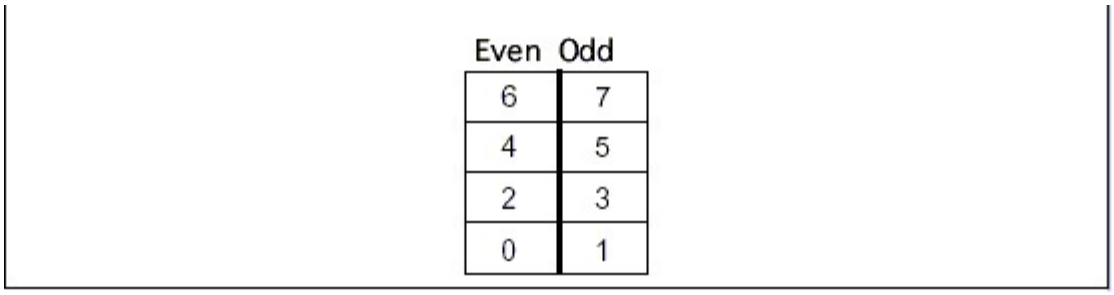


Figure 3.6 Adresses des octets dans une mémoire organisée par mots.

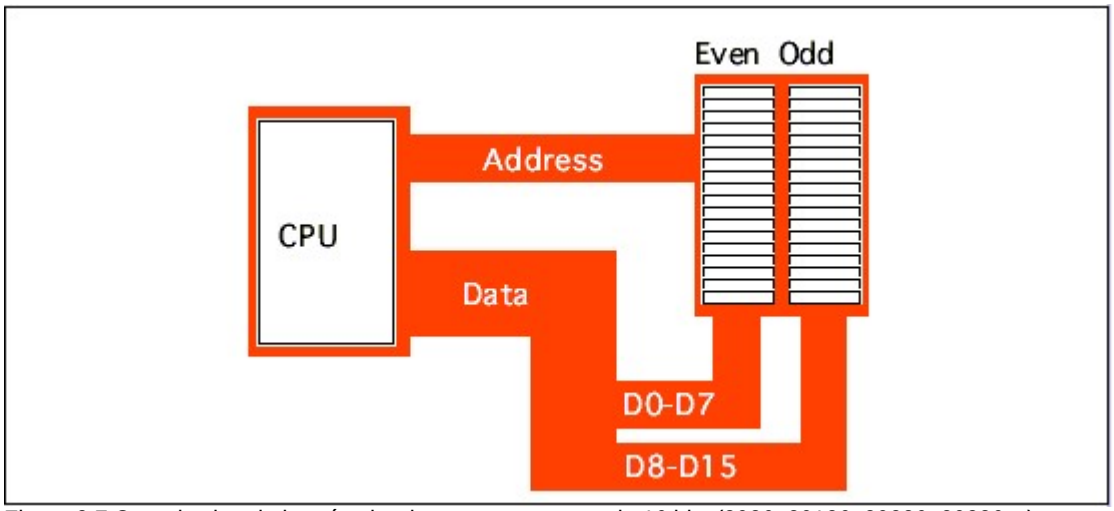


Figure 3.7 Organisation de la mémoire dans un processeur de 16 bits (8086, 80186, 80286, 80386sx)

Les membres 16 bits de la famille 80x86 peuvent charger un mot depuis n'importe quelle adresse. Comme mentionné plus tôt, lle processeur adresse l'octet le moins significatif de la valeur depuis l'adresse spécifiée à l'octet le plus significatif de la prochaine adresse consécutive. Ceci donne lieu à un problème subtil si vous regardez plus attentivement le diagramme ci-dessus. Qu'arrive-t-il si vous accédez à un mot dans une adresse impaire ? Bon, l'octet le moins significatif du mot se situe à l'adresse 125 et l'octet le plus significatif à l'adresse 126. Qu'est-ce qu'il y a de spécial ? Deux problèmes résultent de cette approche.

D'abord, observez encore la figure 3.7 : les lignes du bus allant de 8 à 15 (l'octet le plus significatif), sont connectées à la banque impaire, alors que les lignes allant de 0 à 7 (l'octet le moins significatif), sont connectées à la branche paire. Un accès à l'emplacement 125 aura pour effet de transférer les données au CPU par l'octet le plus significatif du bus de données, alors qu'on s'attend que cette donnée soit adressée dans l'octet le moins significatif ! Heureusement, le CPU reconnaît la situation et place automatiquement cette portion de mémoire sur l'octet le moins significatif.

Le second problème est encore plus obscur. Quand on accède à des mots, en réalité on est en train d'accéder à deux octets séparés, chacun desquels avec sa propre adresse. Donc, la question suivante surgit « Laquelle de ces adresses figurera dans le bus des adresses ? ». Les CPU 80x86 de 16 bits placent toujours des adresses paires dans le bus. Les octets pairs apparaissent toujours dans les lignes D0-D7 et les octets impairs dans les lignes D8-D15. Si vous accédez à un mot dans une adresse paire, le CPU peut transporter tout le bloc de 16 bits en une seule opération de mémoire. Au contraire, si vous accédez à un seul octet, le CPU active la banque appropriée (en utilisant une ligne de contrôle d'activation d'octets (byte enable)). Si l'octet se trouvait dans une adresse impaire, le CPU le transférerait de l'octet le plus significatif à l'octet le moins significatif du bus.

Donc, qu'arrive-t-il quand le CPU accède à un *mot* qui se trouve dans une adresse impaire, comme dans l'exemple qu'on a vu ci-dessus ? Le CPU *ne sera pas en mesure de placer 125 dans le bus d'adresse et lire la donnée de 16 bits d'un seul coup*. Il n'y a pas d'adresses impaires venant d'un CPU 80x86 de 16 bits. Les adresses traitées sont toujours paires. Donc, si vous cherchez de placer 125 dans le bus des adresses, vous y trouverez la valeur 124. Là où vous comptiez d'avoir une valeur de 16 bits à partir de l'adresse 125 vous l'obtiendrez avec l'octet le moins significatif à l'adresse 124 et l'octet le plus significatif à l'adresse 125 - ce qui ne

correspond pas du tout à ce que vous vous attendiez. Accéder un mot se trouvant dans une adresse impaire demande deux opérations de mémoire. D'abord, le CPU doit lire l'octet à l'adresse 125 et ensuite, il doit le faire à l'adresse 126. Finalement, il a besoin de permuter les positions internes de ces octets parce que les deux ont atteint l'unité centrale à partir de la mauvaise position.

Heureusement, les CPU 80x86 de 16 bits masquent ces détails au programmeur. Votre programme peut parfaitement se servir des adresses qui mieux lui chantent et le CPU interprétera comme il faut (et il permutera finalement) les données en mémoire. Cependant, ceci demande deux opérations de mémoire (situation correspondant exactement à ce qui arrivait normalement sur les processeurs 8088 et 80188). Par conséquent, l'accès à des mots par le biais d'adresses impaires est plus lent que l'accès à des adresses paires. **Faites donc attention à la façon dont vous utilisez la mémoire, car de ceci dépend la vitesse de votre programme.**

Accéder des blocs de mémoire de 32 bits demande toujours deux opérations de mémoire sur les processeurs de 16 bits. Et si vous le faites à des adresses impaires, les opérations requises passeront à trois.

Les processeurs de 32 bits de la famille 80x86 (les 80386, 80486 et Pentium Overdrive), utilisent quatre banques de mémoire associées au bus de données de 32 bits (voir la figure 3.8). L'adresse placée dans le bus des adresses est toujours un multiple de quatre. En utilisant diverses "lignes d'activation d'octets" (*byte enable lines*), le CPU peut sélectionner lequel des quatre octets dans le bus est demandé par un programme. Et, comme dans le cas des processeurs de 16 bits, l'unité centrale fera toujours de façon que ces octets soient arrangés convenablement.

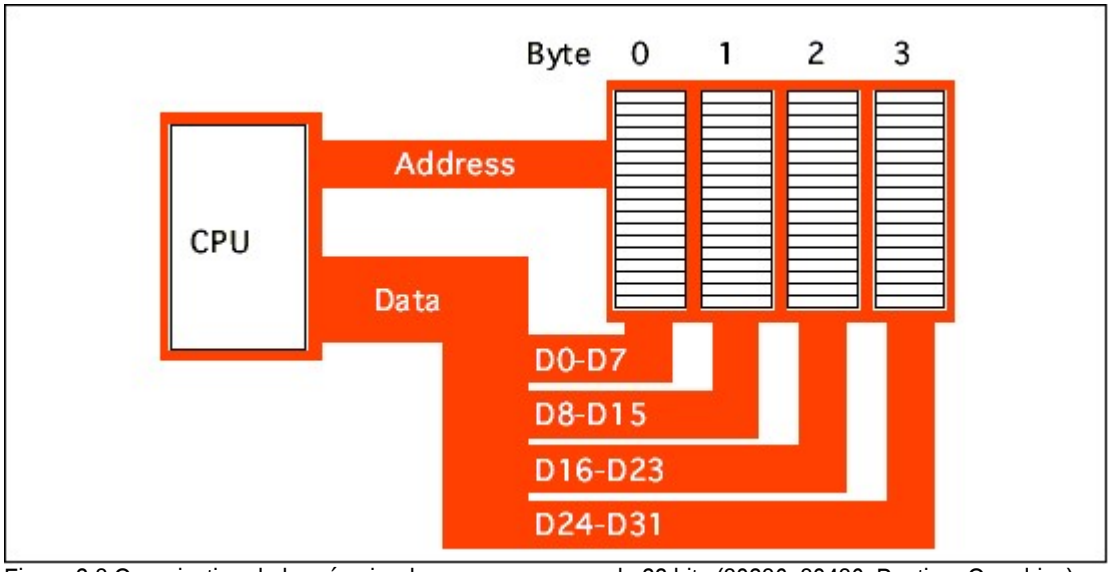


Figure 3.8 Organisation de la mémoire dans un processeur de 32 bits (80386, 80486, Pentium Overdrive)

Avec une interface de mémoire de 32 bits, les CPU 80x86 peuvent accéder à tout mot avec une seule opération de mémoire. Si l'opération (adresse MOD 4) n'est pas égale à trois, le CPU de 32 bits accède au mot de l'adresse spécifiée en une seule opération de mémoire. Cependant, si le reste est trois, il faudra trois opérations de mémoire pour avoir accès au mot (voir figure 3.9). C'est d'ailleurs le même problème rencontré avec les processeurs de 16 bits, sauf qu'il se produit la moitié des fois.

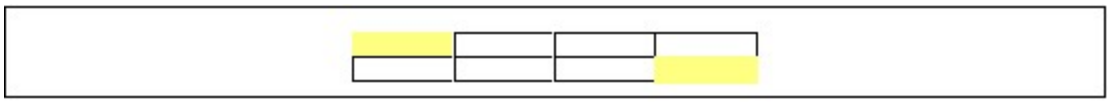


Figure 3.9 Accès à un mot à (adresse **mod** 4) = 3

Un CPU de 32 bits peut accéder à un double-mot en une seule opération de mémoire, *si toutefois* l'adresse de cette donnée est un multiple de quatre. Sinon, le CPU emploiera deux opérations.

Encore une fois, le CPU traite toutes ces situations automatiquement. A condition de charger des données correctes, le CPU fera n'importe quoi pour vous. Cependant, les performances tirent une grande partie de leur profit d'une bonne disposition des données. Comme règle générale, vous devriez toujours placer des mots à des

adresses paires et des doubles-mots à des adresses qui sont des multiples de quatre. Ceci augmentera la vitesse de vos programmes.

3.1.3 Les entrées/sorties

A côté des 20, 24 ou 32 lignes d'adresse, la famille 80x86 fournit également un bus des adresses de 16 bits pour les E/S. Ceci fait en tout deux espaces d'adressage. Les lignes du bus de contrôle différencient les adresses de mémoire des adresses des E/S. Mais, à part les fait d'être différenciées et d'avoir un espace bus plus petit, les adresses E/S ont les mêmes caractéristiques des adresses de mémoire. Les périphériques de mémoire et des entrées/sorties partagent le même bus de données et en particulier les 16 lignes de poids le moins significatif de ce dernier.

Il y a trois limitations pour le système d'E/S dans un IBM PC : d'abord, les CPU 80x86 requièrent des instructions spéciales pour accéder aux périphériques d'E/S ; en second lieu, les concepteurs des IBM PC utilisent les "meilleurs" emplacements d'E/S pour leurs propres besoins en limitant les autres développeurs à un moindre choix d'emplacements ; troisièmement, les systèmes 80x86 ne peuvent adresser plus de 65 536 (2^{16}) adresses d'E/S. En considérant qu'une carte d'affichage VGA normale peut demander jusqu'à 128 000 emplacements différents, vous noterez un problème de capacité.

Heureusement, les concepteurs peuvent adresser leurs périphériques d'E/S dans des adresses de mémoire aussi facilement qu'ils peuvent le faire avec l'espace d'adressage E/S. Donc, en utilisant des circuits appropriés, ils peuvent donner à leurs composants d'E/S l'apparence de la mémoire. Par exemple, c'est le cas des adapteurs d'affichage (display adapters) qui, dans un IBM PC, fonctionnent de cette façon.

L'accès aux périphériques est un sujet sur lequel nous retournerons dans des prochains chapitres. Pour le moment, présumez que les accès à la mémoire et aux E/S fonctionnent de la même façon.

3.2 Le « timing » du système

Bien que les ordinateurs modernes sont très rapides et ils le deviennent toujours plus, ils prennent encore un laps fini de temps pour mener à bout une opération, même étant la plus simple des tâches. Dans les machines Von Neumann, comme dans les 80x86, la plupart des opérations sont *sérialisées*. Ce qui veut dire que l'ordinateur exécute des commandes dans un ordre prédéterminé. Il n'exécutera pas, par exemple, l'instruction I:= I * 5 + 2; avant I:=J; dans la séquence :

```
I := J;
I := I * 5 + 2;
```

Naturellement, on a besoin de quelque moyen pour déterminer quelle instruction s'exécute en premier.

Sans doute, au sein des systèmes réels, les opérations ne s'effectuent pas de façon instantanée. Déplacer une copie de J dans I prend un certain temps. De la même façon, multiplier I par cinq et puis lui ajouter deux et stocker le tout dans I, c'est une opération qui prend du temps. Comme vous pouvez vérifier, la seconde instruction de l'exemple en Pascal est un peu plus lente que la première. Ceux qui sont intéressés à écrire des programmes rapides, se poseront la question : « Comment les processeurs exécutent les instructions et comment ils mesurent leur temps d'exécution ? »

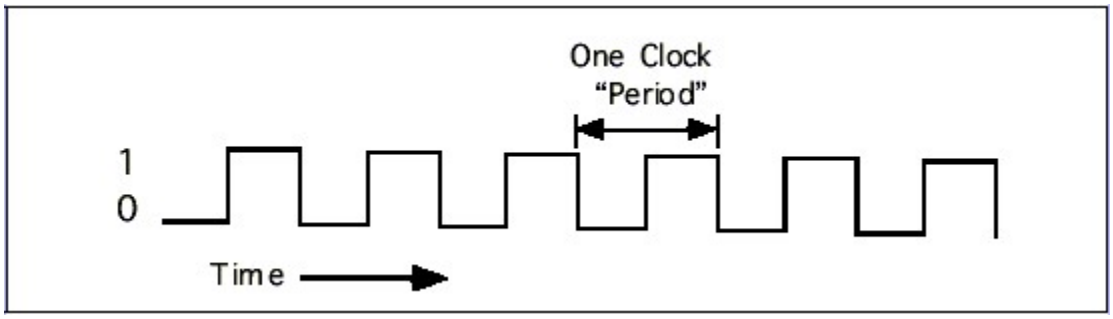
Le CPU est un ensemble de circuits très complexe. Sans se perdre en détails, il suffira de dire que les opérations à l'intérieur d'un CPU doivent être soigneusement coordonnées ou le processeur produira des résultats incorrects. Pour assurer que toute opération aura lieu au bon moment, les CPU 80x86 utilisent un signal alterné, appelé *horloge système* (system clock).

3.2.1 L'horloge système

Au niveau le plus bas, l'*horloge système* traite toutes les synchronisations d'un ordinateur. Il s'agit d'un signal électronique dans le bus de contrôle qui alterne entre zéro et un avec une fréquence périodique (voir figure 3.10). Les CPU sont un bon exemple d'un complexe système logique synchronisé (voir le chapitre précédent).

L'horloge système relie beaucoup des portions logiques qui constituent le CPU leur permettant d'opérer de manière synchronisée.

Figure 3.10 L'horloge système



La fréquence avec laquelle l'horloge système alterne entre zéro et un est nommé la *fréquence de l'horloge*. Le temps employé par l'horloge pour passer de 0 à 1 et revenir à 0 est la *période d'horloge*. Une période complète est appelée un *cycle d'horloge.* Dans les systèmes modernes il peut y avoir plusieurs de milliers de cycles par seconde. La fréquence d'horloge est donc le nombre de cycles qui ont lieu chaque seconde. Une puce 80486 a une vitesse de 66 millions de cycles par seconde. "Hertz" (Hz) est le terme technique pour désigner un cycle par seconde. Par conséquent, la puce 80486 tourne à 66 millions de hertz, ossia, 66 mégahertz (MHz). La famille 80x86 a une échelle de fréquences allant de 5 MHz à 200 MHz et encore plus. Notez qu'une période d'horloge (la quantité de temps écoulée pour chaque cycle) est la réciproque de la fréquence d'horloge. Par exemple, 1 MHz aura une période d'horloge d'une microseconde (1 / 1 000 000 de seconde). De la même façon, une horloge de 10 MHz aura une période de 100 nanosecondes (100 milliardièmes de seconde). Un CPU ayant une vitesse de 50 MHz aura donc une période de 20 nanosecondes. Notez qu'il est d'usage de se référer aux périodes d'horloge en millionièmes de seconde.

Pour assurer la synchronisation, la plupart des CPU commence une opération ou bien sur le *rebord tombant* (falling edge) (c'est-à-dire quand l'horloge est sur le point de tomber de 1 à 0), ou bien sur le *rebord montant* (rising edge) (c'est-à-dire quand l'horloge est sur le point de remonter de 0 à 1). L'horloge système emploie la plupart de son temps dans un de ces deux états ; il n'est entre les deux que pendant de très courtes périodes de temps. Par conséquent les *rebords* sont véritablement le parfait point de synchronisation.

Puisque toutes les opérations de l'unité centrale sont synchronisées autour de l'horloge, le CPU ne peut pas effectuer des tâches plus rapidement que celui-ci[4]. Cependant, le fait que le CPU tourne à une certaine fréquence ne veut pas dire qu'il y aura exécution d'autant d'opérations par seconde. Certaines opérations peuvent prendre plusieurs cycles pour être complétées, donc le CPU effectue souvent ses opérations à une fréquence moindre.

3.2.2 L'accès mémoire et l'horloge système

L'accès à la mémoire est probablement l'activité la plus commune du CPU. Un tel accès est définitivement une opération synchronisée autour de l'horloge système. C'est-à-dire, lire une valeur de la mémoire ou écrire une valeur dans la mémoire ne se produit pas plus souvent qu'une fois par cycle[5]. Par contre, dans certains processeurs 80x86, cela peut prendre plusieurs cycles d'horloge pour accéder à un emplacement de mémoire. Le *temps d'accès à la mémoire* est mesuré par le nombre des cycles d'horloge dont le système a besoin pour atteindre une adresse donnée ; ceci est une valeur importante puisque des temps d'accès plus longs se traduisent en moindres performances.

[4]Certaines des dernières versions du 80486 utilisent des circuits spéciaux de doublage d'horloge, pour aller deux fois plus rapidement que la fréquence ordinaire. Par exemple, avec une fréquence de 25 MHz la puce peut tourner à la vitesse effective de 50 MHz. Cependant la fréquence d'horloge interne est de 50 MHz. Evidemment, le CPU ne pourra pas dépasser cette seconde limite.

[5]Ceci est vrai même pour les horloges dont la vitesse a été doublée.

Différents processeurs 80x86 ont différentes durées d'accès à la mémoire et vont d'un à quatre cycles d'horloge. Par exemple, les CPU 8088 et 8086 demandent *quatre* cycles pour accéder à la mémoire ; alors que le processeur 80486 requiert seulement un cycle. Par conséquent, le 80486 exécute les programmes plus rapidement que le 8086, même en ayant à la même fréquence d'horloge.

Le temps d'accès à la mémoire est la quantité de temps entre une requête d'opération de mémoire (lecture ou écriture) et le temps requis pour que l'opération soit complétée. Dans un 8088/8086 à 5 MHz, le temps d'accès à la mémoire est approximativement de 800 ns (nanosecondes). Sur un CPU 80486 de 50 MHz, le temps d'accès à la mémoire est légèrement inférieur à 20 ns. Notez que le temps d'accès sur un 80486 est 40 fois supérieur au temps requis par les processeurs 8088/8086. Ceci parce que la fréquence d'horloge du 80486 est dix fois plus rapide et utilise ¼ de cycle pour accéder à la mémoire.

À l'heure de lire la mémoire, le temps d'accès se considère comme le temps écoulé à partir du moment où le CPU a placé une adresse dans le bus des adresses et le moment où le CPU récupère la donnée du bus. Dans un CPU 80486 avec un temps d'accès mémoire d'un cycle, une lecture ressemble à quelque chose comme dans la figure 3.11. Écrire la mémoire comporte un processus similaire (voir figure 3.12).

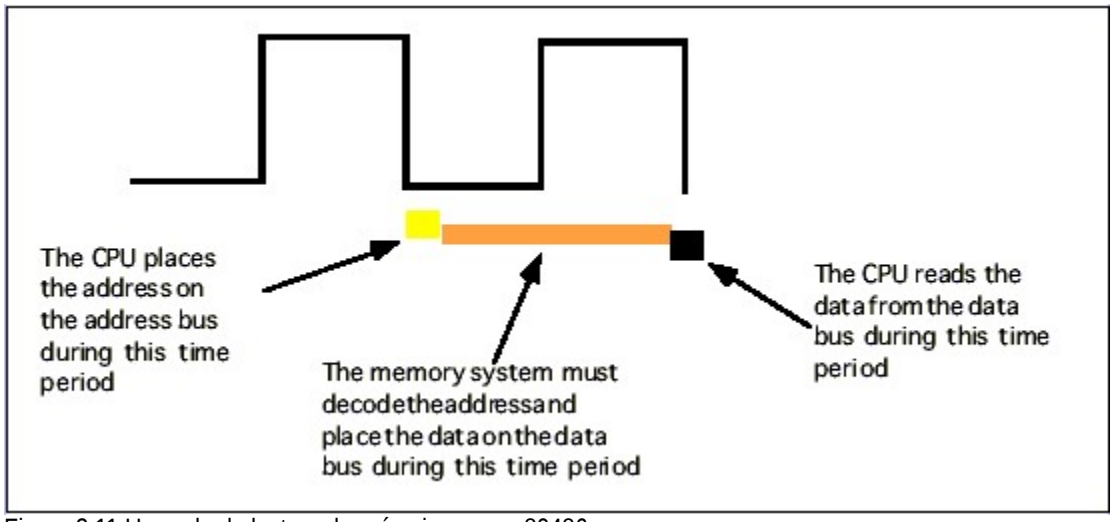


Figure 3.11 Un cycle de lecture de mémoire sur un 80486

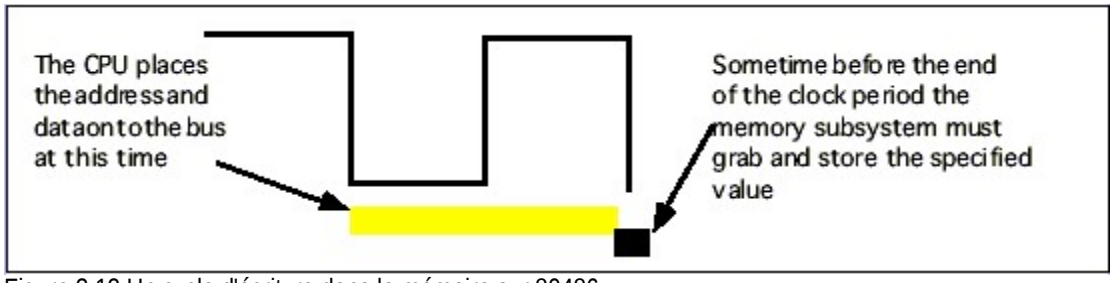


Figure 3.12 Un cycle d'écriture dans la mémoire sur 80486

Notez que le CPU n'attend pas pour la mémoire. Le temps d'accès est spécifié par la fréquence d'horloge. Si la mémoire n'est pas assez rapide, le CPU lira des données incohérentes dans une opération de lecture et ne placera pas des données correctes à la prochaine opération d'écriture. Ceci fera sûrement planter le système.

Les périphériques de mémoire ont plusieurs caractéristiques, mais les deux principales sont la capacité et la vitesse d'accès. Normalement les mémoires vives dynamiques (Random Access Memory) ont une capacité de quatre ou plus Mo et une vitesse d'accès moyenne allant entre 50 et 100 ns. Vous pouvez acheter des composants plus gros et plus rapides, mais ils sont beaucoup plus chers. Un système 80486 à 33 MHz utilise des mémoires de 70 ns.

Attends ! A une vitesse de 33 MHz une période d'horloge dure approximativement 33 ns. Comment un concepteur de système peut se débrouiller avec des mémoires de 70 ns ? La réponse demeure dans les états d'attente (*wait states*).

3.2.3 Les états d'attente (wait states)

Un état d'attente n'est autre chose qu'un cycle d'horloge supplémentaire servant à donner à divers périphériques le temps de compléter une opération. Par exemple un 80486 à 50 MHz a une période d'horloge de 20 ns. Ceci implique une mémoire allant à une vitesse d'accès de 20 ns. Mais, en fait, la situation est encore pire. Dans beaucoup de systèmes il y a des circuits additionnels entre le CPU et la mémoire : la logique de décodage et de mise en tampon. Ces circuits supplémentaires ajoutent des délais additionnels (voir figure 3.13). Dans ce diagramme, le système perd 10 ns pour le buffering et le décodage. Donc, si le CPU a besoin d'un retour des données en 20 ns, la mémoire devrait répondre en moins de 10 ns.

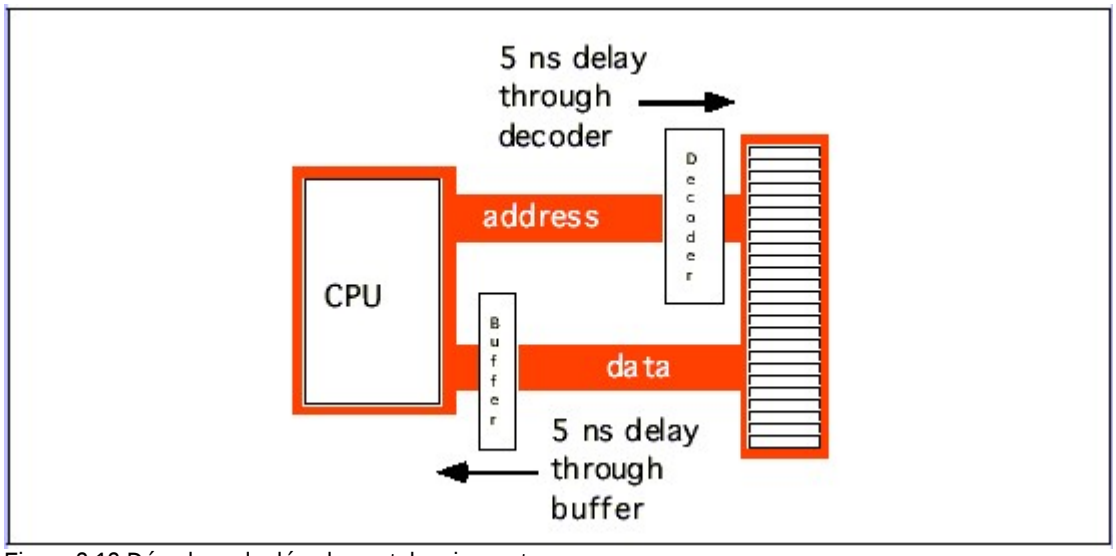


Figure 3.13 Décodage de décodage et de mise en tampon

Il est désormais possible d'acheter des mémoires travaillant à 10 ns. Cependant, elles sont extrêmement onéreuses ; elles sont aussi très volumineuses, consommant beaucoup de tension et générant trop de chaleur. Ce sont de mauvais attributs. Les superordinateurs utilisent ce type de mémoire. Mais ils coûtent aussi des millions de dollars, occupent des pièces entières et requièrent des refroidissements spéciaux et d'énormes alimentateurs. Pas le genre de chose que vous voulez placer dans votre bureau.

Si les mémoires avec des coûts raisonnables ne marchent pas avec des processeurs rapides, comment les compagnies se débrouillent-elles pour vendre des PC rapides ? Une partie de cette réponse peut être fournie par les états d'attente. Par exemple, si vous avez un processeur de 20 MHz avec une mémoire ayant une durée de cycle de 50 ns et vous perdez 10 ns pour le buffering et le décodage, vous disposez de 40 ns de vitesse mémoire. Et si vous pouviez vous permettre uniquement une mémoire de 80 ns sur un système travaillant à 20 MHz ? Ajouter un état d'attente qui porte le cycle de mémoire à 100 ns (deux cycles) résout le problème. En soustrayant 10 ns pour le décodage et le tamponnage, on a 90 ns libres. Par conséquent, une vitesse de 80 ns pourrait répondre convenablement avant que le CPU réclame les données.

Presque tous les CPU ordinaires existants fournissent un signal dans le bus de contrôle pour permettre l'insertion d'états d'attente. Généralement, les circuits de décodage testent cette ligne pour ajouter une période d'horloge supplémentaire, si nécessaire. Ceci donne à la mémoire suffisamment de temps d'accès et permet au système de fonctionner correctement (voir figure 3.14).

Parfois, un seul état d'attente n'est pas suffisant. Considérez un 80486 allant à 50 MHz. Une durée normale de cycle de mémoire devrait être inférieure à 20 ns. Par conséquent, moins de 10 ns sont requis après avoir enlevé le temps requis par le buffering et le décodage. Si vous êtes en train d'utiliser une mémoire à 60 ns dans le système, ajouter un seul état d'attente ne réglera pas la question. Chaque état d'attente vous donne 20 ns de gain, donc avec un seul de ces états vous aurez besoin d'autres 30 ns. Pour travailler avec une mémoire de 60 ns, vous devrez ajouter trois états d'attente (aucun état = 10 ns, un état = 30 ns, deux états = 50 ns, et trois états = 70 ns).

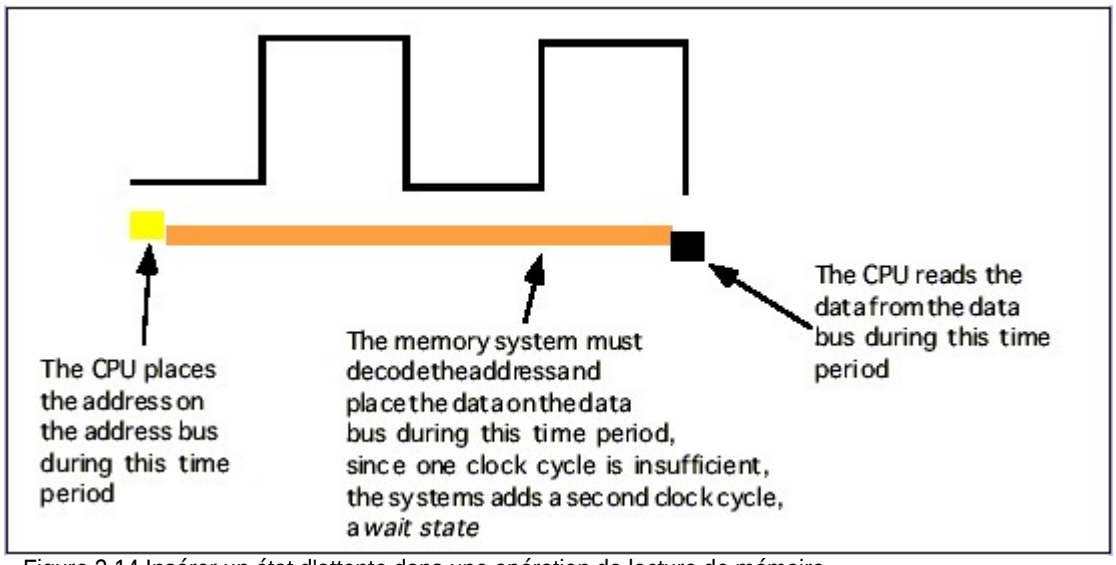


Figure 3.14 Insérer un état d'attente dans une opération de lecture de mémoire

Inutile de dire que pour la performance générale du système, les états d'attente ne sont pas une bonne idée, car, pendant que le CPU attend que la mémoire lui fournisse des données, il ne peut rien faire avec ces dernières.

Ajouter un état d'attente pour un cycle de mémoire dans un CPU 80486 *redouble* le temps requis pour accéder aux données. Ceci *divise par deux* la vitesse de l'accès mémoire. Exécuter un état d'attente dans chaque accès de mémoire c'est presque comme couper en deux la fréquence d'horloge. Vous obtiendrez beaucoup moins de travail terminé dans le même laps de temps.

Vous avez probablement vu une annonce comme : « 80386dx, 33MHz, 8 Mo de RAM et 0 états d'attente… seulement pour 1000 $! ». Si vous lisez plus attentivement le slogan, vous remarquerez que le fabricant est en train d'utiliser une mémoire dont la vitesse est de 80ns. Comment peuvent-ils mettre en œuvre des systèmes qui tournent à 33 MHz sans aucun état d'attente ? C'est facile : ils mentent.

Cependant, il est toujours possible de concevoir un système de mémoire que, *sous certaines circonstances spéciales* réussit à fonctionner sans états d'attente. Beaucoup de modèles de marketing prévoient que si leur système arrive à fonctionner sans états d'attentes au moins parfois, alors ils peuvent se vanter qu'il le fait toujours. En réalité les gens du marketing n'ont aucune idée de ce qu'un état d'attente est, sauf le fait qu'il est une mauvaise chose et son absence est une raison de se vanter.

Cependant, le fait qu'un système comporte des états d'attente ne veut pas du tout dire que le système est lent. Les concepteurs matériels peuvent faire recours à plusieurs procédés pour atteindre un état d'attente zéro la plupart du temps. Le plus commun de ces procédés est de se servir d'une mémoire *cache* (prononcé « *cash* » en anglais).

3.2.4 La mémoire cache

Si vous analysez un programme ordinaire (tout comme certaines recherches ont fait), vous remarquerez sa tendance à accéder de façon répétitive aux mêmes emplacements de mémoire. De plus, vous remarquerez aussi qu'un programme accède souvent à des emplacements adjacents. Le nom technique donné à ce phénomène est « localisation temporelle des références » (*temporal locality of reference*) et « *localisation spatiale des références* » (spatial locality of reference). Quand un programme emploie la localisation spatiale, il accède à des adresses de mémoire contiguës. Et quand il emploie la localisation temporelle des références, il accède en fait, de façon répétitive, et pendant une courte période, aux mêmes adresses. Les deux formes de localisation se produisent dans le programme Pascal suivant :

```
for i := 0 to 10 do
        A [i] := 0;
```

Dans cette boucle, il y a deux occurrences chacune ayant une localisation spatiale et temporelle des références. Considérons la première, qui est la plus évidente.

Dans notre code Pascal, le programme référence la variable i plusieurs fois. La boucle for, compare i à 10 comme condition d'arrêt. Elle incrémente également i à la fin de chaque tour. L'instruction d'affectation utilise également i comme indice de tableau. Ceci montre l'action de la localisation temporelle des références puisque le CPU accède à i en trois endroit du programme dans un court laps de temps.

Ce programme montre également la localisation spatiale des références. La boucle initialise à zéro les éléments du tableau A, en plaçant un 0 dans le premier emplacement de A, un 0 dans le suivant, et ainsi de suite. En supposant que Pascal stocke les éléments de A dans des emplacements de mémoire consécutifs[6], chaque itération de la boucle accède à des adresses de mémoire adjacentes.

Dans le programme ci-dessus, il y a un exemple supplémentaire de la localisation spatiale et temporelle des références, quoiqu'il ne soit pas aussi évident. Les instructions de l'ordinateur qui demandent au système d'effectuer une tâche spécifique apparaissent aussi dans la mémoire. Elles apparaissent séquentiellement (localisation spatiale). L'ordinateur exécute également ces instructions de façon répétitive, une fois pour chaque itération (localisation temporaire).

Si vous jetez un coup d'oeil au *profil d'exécution* d'un programme, vous découvrirez qu'il n'exécute normalement que moins de la moitié des instructions. Généralement, un programme ordinaire peut n'utiliser que 10% ou 20% de la mémoire qui lui est allouée. À un moment donné, un programme dont la taille est d'un Mo peut n'accéder qu'à 8 ou 10 ko de la mémoire destinée aux données et au code. Donc, si vous avez payé une outrageuse somme d'argent pour des méga RAM sans états d'attente, vous n'en profiterez que rarement ! Ne serait-ce sympa si vous pouviez acheter une mémoire plus modeste mais réattribuant dynamiquement ses adresses lors de l'exécution du programme ?

C'est exactement ce que vous permet la mémoire cache. Elle est située entre le CPU et la mémoire centrale. C'est une petite portion de mémoire très rapide (zéro état d'attente). Contrairement à la RAM ordinaire, les octets apparaissant sur la mémoire cache n'ont pas d'adresse fixe ; le système peut donc réassigner l'adresse d'un objet de données. Ceci lui permet de garder les valeurs qui ont dernièrement été accédées dans le cache. Les adresses inutilisées par le CPU ou qui n'ont pas été accédées dernièrement, restent dans la (lente) mémoire centrale. Puisque la plupart des accès de mémoire sont des variables récemment utilisées (ou des adresses qui se trouvent au voisinage des adresses dernièrement utilisées), les données apparaissent généralement dans la mémoire cache.

La mémoire cache n'est pas parfaite. Bien qu'un programme puisse employer un temps considérable pour exécuter du code à un endroit donné, il appellera finalement une procédure ou ira se balader dans des sections de code à l'extérieur du cache. Dans de telles circonstances, le CPU devra s'adresser à la mémoire centrale pour charger des données. Puisque cette mémoire est lente, l'insertion des états d'attente sera requise.

Un "cache hit" ("réussite de cache") se produit chaque fois que le CPU y accède et y trouve des données. Dans ces cas, le processeur aura accès aux données sans états d'attente. Un "cache miss" ("échec de cache" se produit chaque fois que le chache n'a pas de données ou que le CPU accède à la mémoire centrale. Le CPU devra alors lire les données dans la mémoire conventionnelle, résultant en une perte de performances. Pour prendre avantage sur la localisation des références, le CPU copie les données dans le cache toutes les fois qu'il accède à une adresse que le cache ne contient pas. Puisque ces mêmes adresses sont susceptibles d'être accédées tôt, le CPU économisera du temps en les ayant déjà sur le cache.

Comme décrit ci-dessus, la mémoire cache traite l'aspect temporaire de l'accès mémoire, mais non l'aspect spatial. Quand on accède à des emplacements de mémoire sur le cache, la vitesse du programme ne sera pas meilleure si on accède constamment à des emplacements consécutifs (localisation spatiale des références). Pour résoudre ce problème, beaucoup de systèmes de mémoire cache lisent plusieurs octets consécutifs en mémoire conventionnelle quand une adresse n'est pas trouvée[7]. Le 80486, par exemple, lit 16 octets après chaque échec. Si vous lisez 16 octets, pourquoi les lire en bloc et non selon les besoins ? Il s'avère que beaucoup de puces de mémoire disponibles aujourd'hui ont des modes spéciaux permettant d'accéder rapidement à plusieurs emplacements consécutifs dans la puce. Le cache exploite cette fonctionnalité pour réduire le nombre moyen d'états d'attente requis pour accéder à la mémoire.

[6]Et il le fait, cf. début du chapitre 4.
[7]Les engeigneurs appelent ce bloc de données une *ligne de cache* (cache line).

Si vous écrivez un programme qui accède aléatoirement à la mémoire, utiliser un cache pourra en fait vous ralentir. Lire 16 bits après chaque échec de mémoire cache est plutôt cher si vous avez juste besoin de lire quelques octets dans la ligne de cache correspondante. Néanmoins, les systèmes de mémoire cache font un bon travail.

Il ne devrait pas surprendre que le rapport entre les *hits* et les *miss* croît avec la taille (en octets) de la mémoire cache. Par exemple, la puce 80486 a 8 192 octets dans le chip de sa mémoire cache. Intel se vante d'atteindre des probabilités de succès jusqu'à 80 ou 95% avec ce cache (c'est-à-dire que le CPU y trouverait des données avec 80-95% de probabilités). Ceci paraît très impressionnant. Cependant, si vous jouez un peu avec les chiffres, vous découvrirez que ce n'est pas aussi surprenant. Prenons en considération le cas 80%. Alors une adresse de mémoire sur cinq ne se trouvera pas dans le cache. Si vous avez un processeur de 50 MHz et une vitesse d'accès à la mémoire de 90 ns, sur cinq accès mémoire, seulement quatre se feront en un seul cycle d'horloge (car les adresses correspondantes se trouveront sur le cache), et le cinquième demandera au moins dix états d'attente[8]. Le système aura besoin de 15 cycles d'horloge pour accéder à cinq emplacements de mémoire, ce qui fait, dans la moyenne, trois cycles d'horloge par accès. C'est l'équivalent de deux états d'attente pour chaque accès à la mémoire. Maintenant, croyez-vous encore que votre machine tourne à zéro états d'attente ?

Il y a cependant deux solutions pour améliorer la situation. En premier, vous pouvez ajouter plus de mémoire cache. Ceci améliore son rapport de succès en réduisant le nombre d'états d'attente. Par exemple, accroître les chances de 80 à 90% vous permet d'accéder à 10 emplacements de mémoire en 20 cycles. Cela réduit le nombre moyen d'états d'attente par accès mémoire à 1, un progrès substantiel. Hélas, vous ne pouvez pas démonter la puce 80486 et souder plus de cache dans la microplaquette. Pourtant, le CPU 80586/Pentium a une mémoire cache bien plus grande et il fonctionne avec moins d'états d'attente.

Une autre moyen d'améliorer les performances est de construire un système cache à deux niveaux. Beaucoup de systèmes 80486 fonctionnent de cette façon. Le premier niveau est le cache à 8 192 octets dans la puce (intégrée). Le niveau suivant est entre le cache intégré et la mémoire centrale, c'est une mémoire cache secondaire montée sur la carte mère (voir fig. 3.15).

Une mémoire cache secondaire contient de 32 768 octets à un Mo de mémoire. Des tailles communes de cache sur les PC sont 65 536 et 262 144 octets.

Vous pouvez vous demander : « Pourquoi s'embêter avec deux niveaux de cache ? Pourquoi ne pas utiliser un seul gros cache de 262 144 octets ? » Eh bien, la mémoire cache secondaire n'opère pas avez zéro états d'attente. Un circuit supportant 262 144 octets de mémoire de 10 ns (20 ns pour une durée d'accès totale), serait cher. Donc, beaucoup de concepteurs de systèmes utilisent comme caches secondaires des mémoires plus lentes qui requièrent un ou deux états d'attente. C'est encore *beaucoup* plus rapide que la mémoire centrale. Sa combinaison avec le cache primaire intégré permet une meilleure performance du système.

Considérez l'exemple précédent avec 80% de probabilités de succès. Si le cache secondaire demande deux cycles pour chaque accès de mémoire et trois cycles pour le premier accès, alors un échec de cache comportera seulement six cycles d'horloge. La moyenne de la performance système serait de deux cycles pour chaque accès mémoire. Plus vite que les trois cycles demandés par un système sans cache secondaire. De plus, ce dernier peut mettre à jour ses données en parallèle avec le CPU. Donc, le nombre de *miss* de cache (qui affectent la performance du CPU), baisse considérablement.

Vous pensez peut-être : « Ceci a l'air intéressant, mais est-ce qu'il a un rapport avec la programmation ? ». Tout à fait. Si vous écrivez vos programmes soigneusement, de façon à tirer avantage du fonctionnement du cache, vous pouvez améliorer les performances. En mettant des variables que vous utilisez couramment dans la même ligne de cache, vous pouvez forcer cette mémoire à charger ces variables en groupe, en économisant des état d'attente supplémentaires pour chaque accès.

[8]Les dix états d'attente se produisent comme suit : cinq cycles d'horloge pour lire les premiers quatre octets (10+20+20+20+20)=90. Cependant le cache lit toujours 16 octets consécutifs. La plupart des mémoires permettent la lecture consecutive d'adresses en environ 40 ns, après avoir accédé au premier emplacement. Par conséquent le 80486 aura besoin de six cycles d'horloge additionnels pour lire les trois doubles-mots restants. Le total est 11 cycles ou 10 états d'attente.

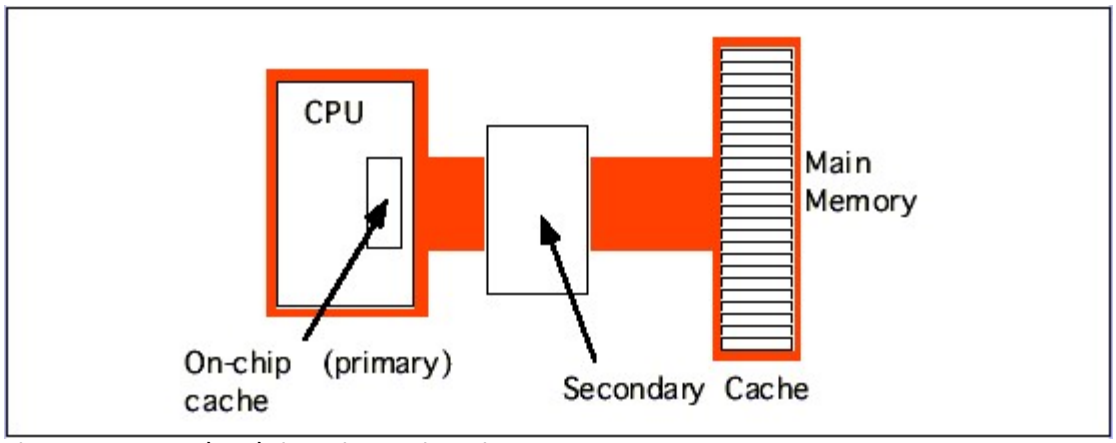


Figure 3.15 Un système à deux niveaux de cache

Si vous organisez votre programme de façon qu'il tend à exécuter la même séquence d'instructions de façon répétitive, il y aura un haut degré de localisation temporaire de références, et il exécutera plus rapidement.

3.3 Les processeurs « hypothétiques » 886, 8286, 8486 et 8686

Pour comprendre comment améliorer les performances système, il est temps d'explorer les opérations internes du CPU. Malheureusement, les processeurs de la famille 80x86 sont très complexes. En discuter l'organisation interne créerait plus de confusion que d'éclaircissements. Par conséquent, nous utiliserons d'abord les processeurs 886, 8286, 8486 et 8686 (les processeurs "x86"). Ces processeurs « hypothétiques » sont des simplifications extrêmes des divers membres de la famille. Ils mettent en évidence les caractéristiques architecturales importantes de la famille 80x86.

À part leur manière d'exécuter les instructions, les processeurs 886, 8286, 8486 et 8686 sont identiques. Ils ont tous le même ensemble de registres et ils exécutent tous le même ensemble d'instructions. Ceci introduit d'autres concepts, mais analysons-les un à la fois.

3.3.1 Les registres du CPU

Les registres du CPU sont des emplacements de mémoire très spéciaux et sont construits avec des flip-flops. Ils ne font pas partie de la mémoire centrale, le CPU les implémente sur puce. Les membres de la famille 80x86 ont différentes tailles de registres. Les CPU 886, 8286, 8486 et 8686 (les x86 d'or en avant) ont exactement quatre registres, tous de 16 bits. Toutes les opérations arithmétiques et de localisation se font dans les registres du CPU.

Puisque le processeur x86 a si peu de registres, nous donnerons à chaque registre son propre nom et nous nous référerons à ce nom plutôt qu'à son adresse. Les noms des registres x86 sont :

AX	- Le registre accumulateur
BX	- Le registre de base (adresse de base)
CX	- Le registre compteur
DX	- Le registre de données

A côté de ces registres, qui sont visibles par le programmeur, les processeurs x86 ont aussi un registre nommé *pointeur d'instruction* (instruction pointer) qui contient l'adresse de l'instruction suivante à exécuter. Il y a aussi un registre *flag* (qu'en français on nomme parfois *drapeaux*). Ce registre peut se souvenir si une valeur était plus petite, égale ou plus grande qu'une autre.

Puisque les registres font partie de la puce (on dit, dans le jargon, qu'ils sont *on-chip*) et sont traités directement par le CPU, ils sont beaucoup plus rapides que la mémoire. *Accéder à un emplacement de mémoire demande un ou plusieurs cycles d'horloge*. Accèder à une donnée dans le registres ne requiert ordinairement aucun cycle. Par conséquent, vous devriez essayer de garder les variables dans les registres. Certes, l'éventail

des registres disponibles est très restreint et beaucoup de registres ont des buts spéciaux qui limitent leur utilisation comme variables, mais ils restent un excellent endroit pour stocker des données temporaires.

3.3.2 L'unité logique et arithmétique

L'unité logique et arithmétique (ULA, ou ALU en anglais) et l'endroit dans le CPU où se produit le plus d'action. Par exemple, si vous voulez ajouter la valeur 5 au registre ax, le CPU doit :

- Copier la valeur dans l'ULA depuis le registre ax
- Envoyer la valeur 5 dans l'ULA
- Informer l'ULA d'ajouter ces deux valeurs ensemble
- Retourner le résultat au registre ax

3.3.3 L'unité d'Interface des bus

L'unité d'interface des bus (UIB, ou BIU en anglais) est responsable du contrôle du bus des adresses et du bus des données quand ils accèdent à la mémoire centrale. Si un cache est présent dans la puce du CPU, alors le UIB est également responsable de l'accès aux données dans le cache.

3.3.4 L'unité de contrôle et le jeu d'instructions

Une bonne question à poser à ce point serait « Comment un CPU effectue précisément les tâches qui lui sont allouées ? » Ceci est réalisé en lui donnant un ensemble fixe de commandes, ou *instructions* permettant son utilisation. Gardez à l'esprit que les concepteurs de CPU construisent ces processeurs à l'aide d'un ensemble de portes logiques ; ce sont elles qui permettent de réaliser ces instructions. Pour garder le nombre de portes logiques dans un ensemble raisonnablement petit (en termes de centaines de milliers), les concepteurs des CPU doivent nécessairement restreindre le nombre et la complexité des commandes que les CPU reconnaissent. Ce petit ensemble de commandes est *le jeu d'instructions du CPU*.

Dans une époque antérieure à celle de Von Neumann, les programmes des systèmes informatiques étaient souvent *câblés électriquement dans les circuits*. C'est-à-dire, *c'était l'implémentation matérielle des ordinateurs qui déterminait le type de problèmes que l'ordinateur en question devait résoudre*. Pour modifier un programme, il fallait donc refaire ce "câblage électrique". Une tâche bien ardue. L'étape successive fut d'implémenter les systèmes d'ordinateurs *programmables*, qui permettaient à un programmeur de changer l'implémentation matérielle d'un programme à l'aide de fiches et de douilles (sockets, plug wires, en anglais). Un programme d'ordinateur consistait donc en un ensemble matriciel de lignes trous (douilles), chaque rangée représentant une opération pendant l'exécution d'un programme. Le programmeur pouvait sélectionner une des diverses instructions en connectant une fiche dans une douille particulière de l'instruction désirée (voir figure 3.16). Certainement, la difficulté majeure avec ce schèma était que le nombre d'instructions possibles était sévèrement limité au nombre de douilles qu'on pouvait physiquement placer dans chaque ligne. Cependant, les concepteurs de CPU ont rapidement découvert qu'avec quelques circuits logiques supplémentaire, ils pouvaient réduire le nombre de douilles requises de n trous pour n instructions à $\log_2(n)$ trous pour n instructions. Ils réaliserent ceci en assignant un code numérique à chaque instruction, puis en encodant cette instruction selon la disposition d'un nombre binaire, à l'aide de $\log_2(n)$ trous (voir figure 3.17). Cette amélioration nécessitait huit fonctions logiques pour décoder les bits A, B et C du le panneau de raccordement, mais cela valait la peine, car il réduisait le nombre de douilles qu'il fallait répéter pour chaque instruction.

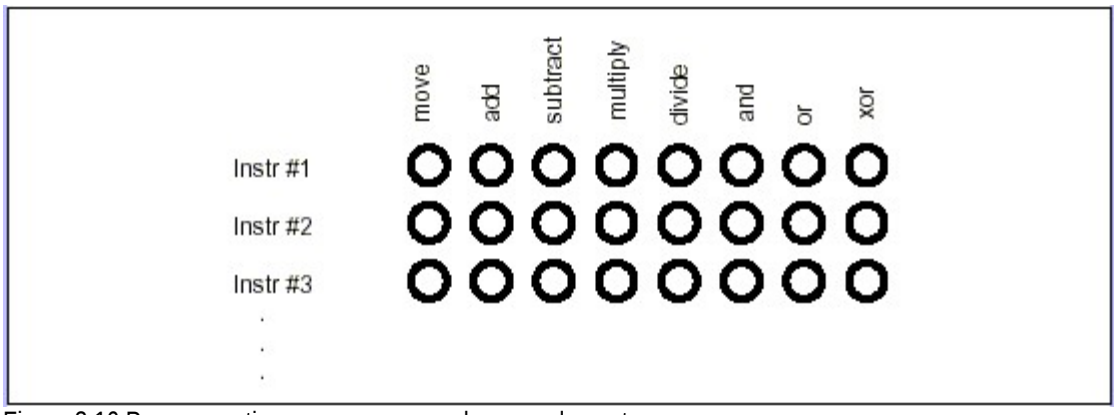


Figure 3.16 Programmation par un panneau de raccordement

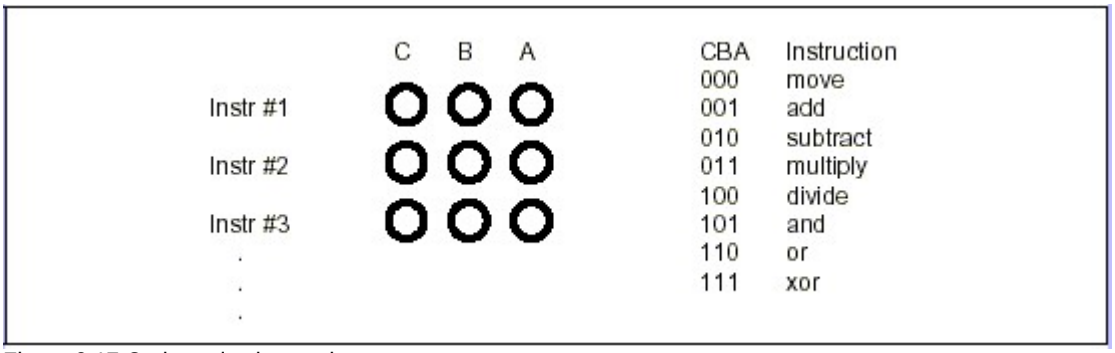


Figure 3.17 Codage des instructions

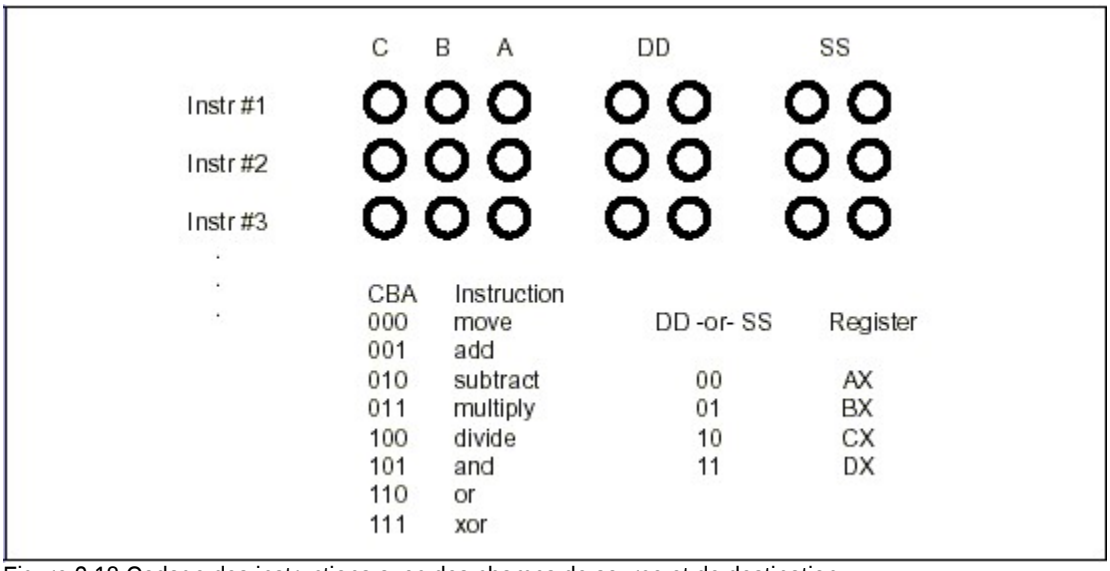


Figure 3.18 Codage des instructions avec des champs de source et de destination

Naturellement, certaines instructions du CPU ne se suffisent pas à elles-mêmes. Par exemple l'instruction *move* est une commande qui transporte des données d'un emplacement de l'ordinateur à un autre (par exemple, d'un registre à un autre). Par conséquent, l'instruction *move* demande deux opérandes : une opérande de *source* et une opérande de *destination*. Les concepteurs de CPU encodent normalement les opérandes de source et de destination comment faisant partie d'une instruction machine, certaines douilles correspondraient aux opérandes de source et certaines autres aux opérandes de destination. La figure 3.17 montre une combinaison possible pour traiter ceci. L'instruction *move* transporterait les données de registre de source au registre de destination, l'instruction *add* ajouterait la valeur du registre de source à la valeur du registre de destination, etc.

Un des principaux progrès fournis par les machines VNA dans la conception des ordinateurs fut le concept de *programme stocké*. Un gros problème avec la méthode des douilles était que le nombre des étapes d'un programme (instructions machine) était limité au nombre de lignes de trous disponibles dans la machine. John Von Naumann et d'autres ont identifié une relation entre les douilles du panneau électrique et les bits dans la mémoire ; ils ont imaginé pouvoir stocker les équivalents binaires d'un programme machine dans une mémoire principale et de pouvoir les charger depuis celle-ci dans un registre spécial de décodage (*decoding register*), connecté directement à l'instruction en train de décoder les circuits dans le CPU.

L'idée pouvait être réalisée en ajoutant encore un circuit supplémentaire au CPU. Ce circuit, l'*unité de contrôle* (UC, ou CU en anglais), charge depuis la mémoire les codes des instructions (également connus comme *codes d'opération* ou *opcodes*) et les transporte dans le registre qui les décode. L'unité de contrôle contient un registre spécial, le *pointeur d'instruction* qui contient l'adresse d'une instruction exécutable. L'unité de contrôle charge le code de l'instruction de la mémoire et le place dans le registre de décodage pour l'exécution. Après avoir exécuté l'instruction, l'unité de contrôle incrémente le pointeur d'instruction, il cherche dans la mémoire la prochaine instruction à exécuter et ainsi de suite.

En concevant un jeu d'instructions, les concepteurs du CPU choisissent généralement des opcodes qui ont une longueur multiple de *huit bits* de sorte que le CPU *puisse facilement retrouver des instructions complètes en mémoire*. Le but de ces concepteurs est d'assigner un nombre approprié de bits au champ d'une classe d'instructions (move, add, substract, etc.) et aux champs d'opérandes. Le fait de choisir plus de bits pour les champs d'instructions permet d'avoir plus de commandes, le fait de choisir des bits additionnels pour les champs d'opérandes permet d'avoir un plus grand nombre d'opérandes (par exemple, emplacement de mémoire ou registres). Il y a des complications supplémentaires. Certaines instructions n'ont qu'une opérande ou même, n'en ont aucune. Au lieu de gaspiller les bits associés à ces champs, les concepteurs des CPU les réutilisent souvent pour encoder des opcodes additionnels, encore une fois avec davantage de circuits. La famille Intel 80x86 profite totalement de ce fait en utilisant des instructions de un à dix octets de long. Etant donné qu'à ce stade, un tel degré de sophistication serait trop complexe, notre attention se basera d'abord sur les CPU x86 qui font usage d'un schéma d'encodage différent - et beaucoup plus simple.

3.3.5 Le jeu d'instructions x86

Le CPU x86 fournit vingt sortes (ou classes) d'instructions de base. Sept de ces fonctions ont deux opérandes, huit en ont une et cinq n'en ont aucune. Les instructions sont *mov* (deux versions), *add*, *sub*, *cmp*, *and*, *or*, *not*, *je*, *jne*, *jb*, *jbe*, *ja*, *jae*, *jmp*, *brk*, *iret*, *halt*, *get* et *put*. Le paragraphe suivant décrit le fonctionnement de chaque instruction.

L'instruction *mov* est constituée en fait de deux classes d'instructions fusionnées dans une seule. Les deux versions de l'instruction *mov* prennent les deux formes suivantes :

```
mov    reg, reg/mémoire/constante
mov    mémoire, reg
```

où *reg* est n'importe lequel des registres ax, bx, cx ou dx ; *constante* est une constante numérique (en notation hexadécimale), et *mémoire* est une opérande spécifiant une adresse de mémoire. La section sivante décrit les formes possibles qu'une opérande de mémoire peut prendre. L'opérande "reg/mémoire/constante" indique que cette opérande spécifique peut être un registre, une emplacement de mémoire ou une constante.

Les *instructions arithmétiques et logiques* prennent les formes suivantes :

```
add    reg, reg/mémoire/constante
sub    reg, reg/mémoire/constante
cmp    reg, reg/mémoire/constante
and    reg, reg/mémoire/constante
or     reg, reg/mémoire/constante
not    reg, reg/mémoire
```

L'instruction *add* ajoute la valeur de la seconde opérande à la première (qui doit être un registre) et laisse le résultat dans la première opérande. L'instruction *sub* soustrait la valeur de la seconde opérande de la première et laisse le résultat dans la première. L'instruction *cmp* compare les valeurs des deux opérandes et enregistre le résultat afin qu'il puisse être utilisé avec l'une des instructions de branchement (qui seront décrites sous peu). Les instructions *and* et *or* permettent d'effectuer des opérations logiques de type AND et OR entre les deux

opérandes et laissent le résultat dans la première opérande. L'instruction *not* permet d'inverser les bits d'une opérande de mémoire ou de registre.

Les instructions de *branchement (control transfer)* permettent d'interrompre l'exécution séquentielle des instructions dans la mémoire et transfèrent le contrôle dans une autre zone, soit inconditionnellement, soit après avoir testé le résultat d'une instruction de comparaison précédente. Ces instructions sont les suivantes :

```
ja     dest          -- Jump if Above                  [si >]
jae    dest          -- Jump if Above or Equal         [si ≥]
jb     dest          -- Jump if Below                  [si <]
jbe    dest          -- Jump if Below or Equal         [si ≤]
je     dest          -- Jump if Equal                  [si =]
jne    dest          -- Jump if Not Equal              [si ≠]
jmp    dest          -- Jump (saut inconditionnel)
iret                 -- Revenir d'une interruption
```

Les six premières instructions de cette classe permettent de vérifier le résultat d'une instruction cmp précédente pour *plus grand que*, *plus grand ou égal*, *plus petit que*, *plus petit ou égal*, *égal* ou *inégal*[9]. Par exemple, si vous comparez les registres ax et bx avec cmp et vous exécutez *ja*, le CPU x86 fera sauter le contrôle du programme dans la destination spécifiée si le contenu de ax est plus grand que le contenu de bx. Si ce n'est pas le cas, alors le contrôle tombera dans la prochaine instruction qui était prévue dans le programme. L'instruction *jmp* transfère inconditionnellement le contrôle à l'adresse spécifiée. L'instruction *iret* retourne le contrôle depuis une *routine de service d'interruption*, sujet qui sera traité bientôt.

Les instructions *get* et *put* permettent de lire et d'écrire des valeurs entières. *get*, interrompra l'exécution du programme et invitera l'utilisateur à entrer une valeur hexadécimale. Cela fait, la valeur sera stockée dans le registre ax. Alors que *put* affiche (toujours en hexadécimal) la valeur du registre ax.

Les instructions qui restent ne demandent pas d'opérande, elles sont *halt* et *brk*. *halt* provoque la terminaison du programme et *brk* interrompt l'exécution du programme en le laissant dans un état qui en permet la reprise.

Les processeurs x86 requièrent un opcode unique pour chaque instruction et non pour chaque classe d'instructions. Quoique "*mov* ax, bx" et "*mov* ax, cx" appartiennent à la même classe, elles ont différents opcodes, car le CPU les différencie. Cependant, avant d'observer tous les opcodes possibles, il serait peut-être utile d'approfondir toutes les opérandes possibles pour ces instructions.

3.3.6 Le mode d'adressage des x86

Les instructions x86 utilisent cinq différents types d'opérandes : registres, constantes et trois différents schémas d'adressage ; chacun de ces schémas est appelé *mode d'adressage*. Les processeurs x86 supportent le *mode d'adressage des registres*[10], le *mode d'adressage immédiat* et le *mode d'adressage direct*. Le paragraphe suivant explique chacun de ces modes.

Les opérandes registres sont les plus faciles à comprendre. Considérez la version suivante de l'instruction mov :

```
mov    ax, ax
mov    ax, bx
mov    ax, cx
mov    ax, dx
```

La première instruction ne fait absolument rien. Elle copie la valeur du registre ax dans le registre ax. Les trois instructions restantes copient le contenu des registres bx, cx et dx dans ax. Notez que les valeurs de bx, cx et dx restent les mêmes. La première opérande (la destination) n'est pas limitée au registre ax ; vous pouvez déplacer des valeurs dans n'importe quel registre.

Les constantes sont également faciles à traiter. Considérez les instructions suivantes :

```
mov    ax, 25
mov    bx, 195
```

[9]Les processeurs x86 effectuent uniquement des comparaisons non signées.
[10]Techniquement, les registres n'ont pas d'adresse ; néanmoins on va appliquer cette notation pour des questions de commodité.

```
mov    cx, 2056
mov    dx, 1000
```

Ces instructions sont toutes simples ; elles chargent leurs registres respectifs de la constante hexadécimale spécifiée[11].

Il y a trois modes d'adressages qui concernent l'accès des données en mémoire. Ces modes de chargement prennent les formes suivantes :

```
mov    ax, [1000]
mov    ax, [bx]
mov    ax, [1000 + bx]
```

La première instruction utilise le mode d'adressage direct pour charger dans ax la valeur de 16 bits stockée dans l'emplacement de mémoire qui commence à l'adresse 1000 (en hexadécimal).

L'instruction mov ax, [bx] charge le registre ax de l'emplacement de mémoire indiqué dans le contenu du registre bx. C'est le mode d'adressage indirect. Au lieu d'utiliser la valeur dans le registre bx, l'instruction accède à l'emplacement de mémoire de l'adresse qui apparaît dans bx. Notez que les deux instructions suivantes :

```
mov    bx, 1000
mov    ax, [bx]
```

sont équivalentes à :

```
mov    ax, [1000]
```

Sans doute la seconde séquence est préférable. Cependant il y a certains cas où l'usage de l'indirection est plus rapide, plus court et meilleur. On verra certains exemples plus loin, en analysant les processeurs x86 individuellement.

Le dernier mode d'adressage est le mode indexé. Un exemple de ce type d'accès est donné par :

```
mov    ax, [1000 + bx]
```

Cette instruction additionne l'*offset* 1000 au contenu de bx pour produire une adresse de mémoire à charger. Cette instruction est utile pour accéder à des éléments de tableaux (array), à des enregistrements et d'autres structures de données.

3.3.7 encodage des instructions des x86

Bien qu'on peut arbitrairement assigner des opcodes à chaque instruction x86, gardez à l'esprit qu'un CPU réel utilise des circuits logiques pour décoder les opcodes et les traiter de façon appropriée. Un code d'opération normal du CPU utilise un certain nombre de bits pour identifier la classe d'instruction (comme mov, add, sub), et un certain nombre de bits pour encoder chaque opérande. Certains systèmes (par exemple CISC ou *Complex Instruction Set Computers*), encodent les champs de façon vraiment complexe en produisant des instructions très compactes. D'autres systèmes (par exemple RISC ou *Reduced Instruction Set Computers*) encodent les opcodes de façon très simple même si ceci représente un gaspillage de quelques bits dans l'opcode ou même si ceci limite le nombre des opérations. La famille Intel 80x86 est définitivement CISC et elle dispose d'un des plus complexes décodages d'opcodes jamais conçu. Le but principal des processeurs hypothétiques x86 est de présenter le concept d'encodage d'instructions sans l'encombrante complexité des processeurs 80x86 tout en montrant le style d'encodage CISC.

Une instruction x86 classique prend la forme représentée dans la figure 3.19. Une instruction de base peut être longue un ou trois octets. L'opcode de l'instruction consiste en un seul octet qui contient trois champs. Le premier champ, les trois bits les plus significatifs, définit la classe de l'instruction. Ceci fournit huit combinaisons. Comme vous vous souvenez, il y a vingt classes d'instructions ; on ne peut pas encoder 20 classes dans trois bits, donc on doit inventer un truc pour traiter les autres classes. Comme vous pouvez voir dans le figure 3.19, l'opcode de base encode les instructions *mov* (deux classes : une où le champ rr indique la destination et mmm indique la source, une où le champ rr indique la source et mmm indique la destination), *add*, *sub*, *cmp* et *or*. Il y a une

[11]Toutes les constantes numériques dans les x86 sont hexadécimales. Le suffixe "h" n'est pas nécessaire.

classe supplémentaire : *special*. Cette classe fournit un mécanisme permettant d'élargir le nombre des classes disponibles ; on y reviendra sous peu.

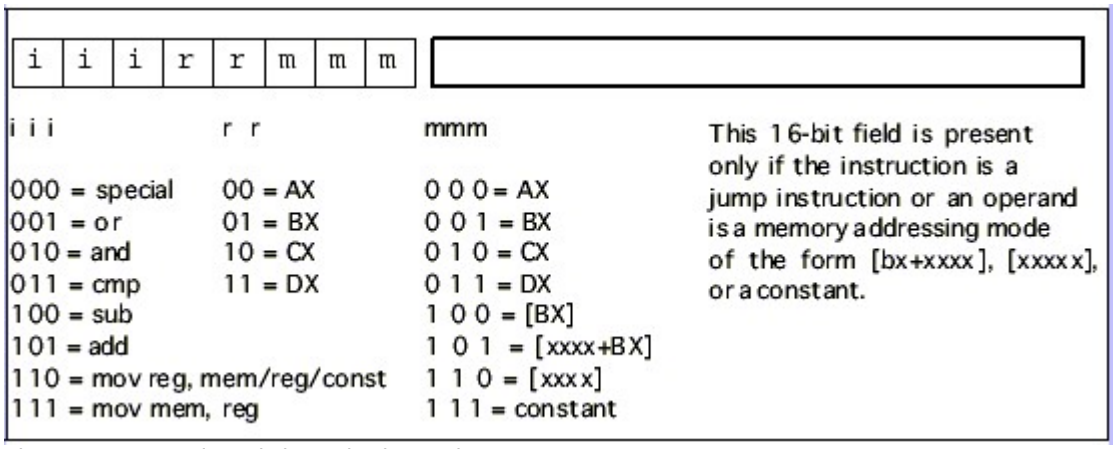


Figure 3.19 L'encodage de base des instructions x86

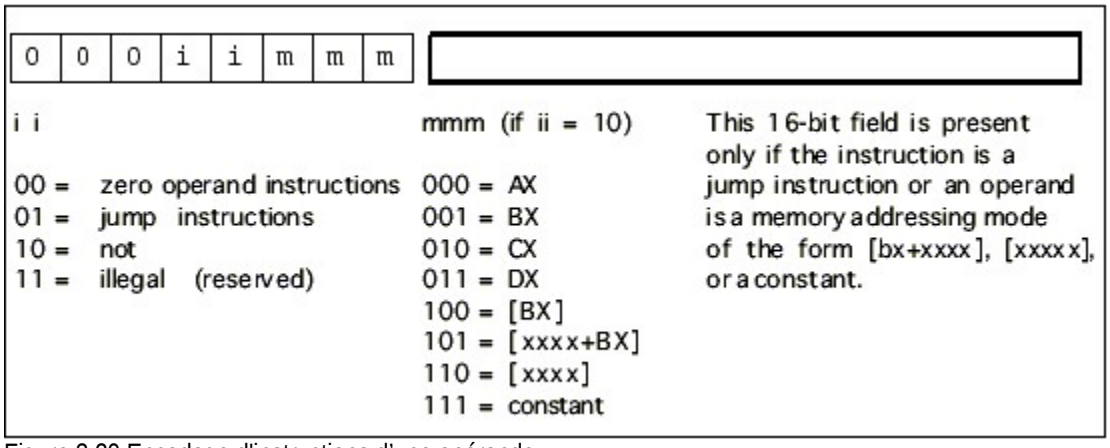


Figure 3.20 Encodage d'instructions d'une opérande

Pour déterminer un opcode d'instruction particulier, vous avez simplement besoin de sélectionner les bits appropriés des champs iii, rr et mmm. Par exemple, pour encoder *mov ax, bx* vous sélectionnerez iii = 110 (mov reg, reg), rr = 00 (ax) et mmm = 001 (bx). Ceci produit l'instruction d'un octet 11000001 ou 0C0h.

Certaines instructions x86 requièrent plus d'un octet. Par exemple, l'instruction *mov ax, [1000]* charge le registre ax du contenu de l'emplacement 1000. L'encodage pour l'opcode est 11000110 ou 0C6h. Cependant l'encodage pour le opcode de *mov ax, [2000]* est également 0C6h. Evidemment, ces deux instructions font des choses différentes, la première charge ax à partir du contenu de l'adresse 1000 et la seconde charge ax du contenu de l'adresse mémoire 2000. Pour encoder une adresse selon les modes d'adressage [xxxx] ou [xxxx + bx] ou pour encoder la constante pour le mode d'adressage immédiat, les opcodes doivent être suivis par une adresse ou une constante de 16 bits, avec l'octet moins significatif juste après l'opcode en mémoire et l'octet le plus significatif à la suite. Alors, les trois octets encodant *mov ax, [1000]* seraient 0C6h, 00h et 10h[12] et les trois octets encodant *mov ax, [2000]* seraient 0C6h, 00h et 20h.

L'opcode *special* permet au CPU x86 d'élargir l'ensemble des instructions disponibles. Celui-ci traite divers types d'instructions ayant une ou aucune opérande, comme montré dans les figures 3.20 et 3.21.

[12]Souvenez-vous que toutes les constantes numériques sont hexadécimales.

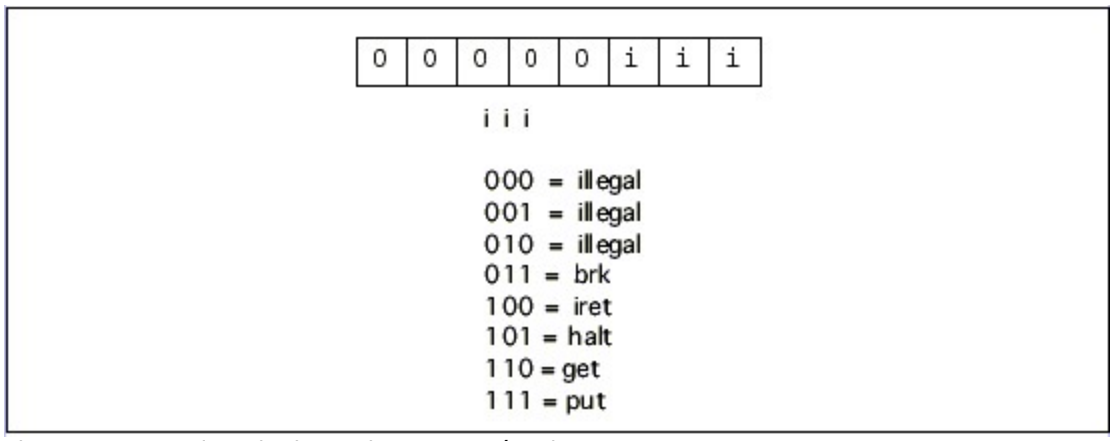


Figure 3.21 L'encodage des instructions sans opérandes

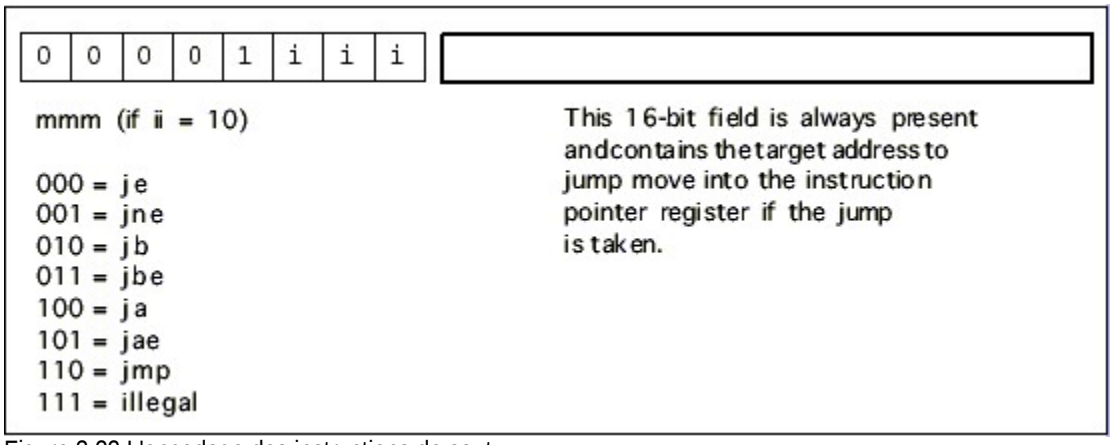


Figure 3.22 L'encodage des instructions de saut

Il y a quatre classes d'instructions prenant une seule opérande. Le premier opcode (*ii* = 00) admet un ensemble d'instructions sans opérandes (figure 3.21). La seconde possibilité (avec *ii* = 01) est également un opcode d'expansion qui fournit toutes les instructions de saut (voir fig. 3.22). La troisième combinaison (*rr* = 10) est l'instruction *not*. Ceci est l'opération logique NOT qui invertit tous les bits dans le registre de destination ou dans l'opérande de mémoire. Le quatrième opcode (ii = 11) est inutilisé. Toute tentative de l'exécuter arrêtera le processeur avec une erreur d'instruction non permise. Les concepteurs des CPU se réservent souvent des opcodes de ce genre pour pouvoir ensuite élargir l'ensemble des instructions (comme Intel a fait en passant du 80286 au 80386).

Il y a sept instructions de saut dans l'ensemble des instructions x86. Toutes prennent la forme suivante :

```
jxx    adresse
```

L'instruction jmp copie la valeur immédiate de 16 bits (l'adresse) qui suit l'opcode dans le registre IP. Par conséquent, le CPU chargera la prochaine instruction à l'adresse visée. Les six instructions restantes sont des sauts conditionnels. Elles testent certaines conditions et passent le contrôle à l'endroit spécifié, si les conditions sont vraies ; elles passent à l'instruction suivante si la condition est fausse. Ces six instructions *ja*, *jae*, *jb*, *jbe*, *je* et *jne* permettent de tester *plus grand*, *plus grand ou égal*, *plus petit*, *plus petit ou égal*, *égal* et *inégal*. Normalement, on les exécute immédiatement après une instruction *cmp*, puisqu'elle ajuste les flags *moindre que* et *égal* que l'instruction de saut conditionnel teste. Noter que les opcodes possibles pour les instructions de saut sont en réalité huit, mais le x86 n'en utilise que sept. Le huitième est un autre opcode réservé.

Le dernier groupe d'instructions, les instructions sans opérandes, apparaissent dans la figure 3.21. Trois de ces instructions sont des opcodes illégaux. L'instruction *brk* (break) arrête le CPU tant que l'utilisateur ne le redémarre manuellement. Ceci est utile pour mettre un programme en pause pendant l'exécution pour en observer les résultats. L'instruction iret (interrupt return) permet au contrôle de revenir d'une *routine de service d'interruption* (RSI ou ISR en anglais). On discutera de ces routines plus tard. L'instruction *halt* termine

l'exécution d'un programme. L'instruction *get* lit une valeur hexadécimale entrée par l'utilisateur et la stocke dans le registre ax ; l'instruction *put* envoie en sortie la valeur dans ax.

3.3.8 L'exécution des instructions étape par étape

Un CPU x86 *ne termine pas* l'exécution d'une instruction en un cycle d'horloge. Pour chaque instruction, il passe par diverses étapes. Par exemple, pour exécuter l'instruction *mov reg, reg/mem/const*, l'unité de contrôle effectue les commandes suivantes :

- Charger l'octet de l'instruction dans la mémoire.
- Mettre à jour le registre IP pour pointer sur l'octet suivant.
- Décoder l'instruction pour voir qu'est-ce qu'elle fait.
- Si requis, charger l'opérande de 16 bits de l'instruction dans la mémoire.
- Si requis, mettre à jour IP pour qu'il pointe au-delà de l'opérande.
- Calculer l'adresse de l'opérande, si requis (c'est-à-dire bx + xxxx).
- Charger l'opérande.
- Stocker la valeur trouvée dans le registre de destination.

Une description étape par étape peut aider à rendre clair ce que le CPU est en train de faire. Dans la première étape, il recherche l'octet de l'instruction dans la mémoire. Pour ce faire, il copie la valeur du registre IP dans le bus des adresses et y lit l'octet à cette adresse. Ceci prend un cycle d'horloge[13].

Après avoir chargé l'octet de l'instruction en cours, le CPU met à jour le registre IP pour qu'il pointe sur l'octet suivant dans le flux d'instructions. Si l'instruction courante compte plusieurs octets, IP est en train de pointer sur la première opérande de celle-ci. Si au contraire elle est une instruction d'un seul octet, alors il pointera sur la prochaine instruction. Ceci prend un autre cycle d'horloge.

La prochaine étape est de décoder l'instruction pour voir ce qu'elle fait. Ceci indiquera au CPU, parmi d'autres choses, s'il faut charger des octets d'opérandes additionnels dans la mémoire. Ce qui prend encore un cycle.

Pendant le décodage, le CPU détermine les types d'opérandes que l'instruction demande. Si l'instruction requiert une opérande constante de 16 bits (c'est-à-dire, si le champ *mmm* est 101, 110 ou 111), alors le CPU chargera la constante dans la mémoire. Cette étape peut demander zéro, un ou deux cycles. Elle demande zéro cycles s'il n'y a pas d'opérandes de 16 bits à charger ; un cycle si l'opérande de 16 bits fait bien partie du mot (autrement dit, si elle commence à une adresse paire) ; deux cycles si la dernière condition n'est pas satisfaite (elle commence à une adresse impaire).

Si le CPU trouve l'opérande de 16 bits, il doit incrémenter IP de deux pour qu'il pointe sur l'octet suivant l'opérande. Cette opération prend zéro cycles ou un cycle d'horloge. Zéro cycles s'il n'y a pas d'opérande et un cycle si l'opérande est présente.

Ensuite, le CPU calcule l'adresse de l'opérande si celle-ci correspond à une adresse de mémoire. Cette étape est requise seulement quand le champ *mmm* de l'octet de l'instruction est 101 ou 100. Si le champ *mmm* contient 100, alors le CPU calcule la somme entre le registre bx et la constante de 16 bits ; ceci demande deux cycles, un pour charger la valeur de bx et l'autre pour effectuer la somme entre bx et xxxx. Si le champ *mmm* contient 100, alors le CPU charge la valeur de bx pour obtenir l'adresse mémoire. Ceci demande un cycle. Si *mmm* ne contient ni l"un, ni l'autre, alors cette étape est sautée et aucun cycle n’est consommé.

Trouver l'opérande peut prendre de zéro à trois cycles, selon la nature de l'opérande. Si elle est une constante (*mmm* = 111), alors cette étape ne demande pas de cycles parce qu'on a déjà trouvé cette constante dans la mémoire dans une étape précédente. Si l'opérande est un registre (*mmm* = 000, 001, 010 ou 011), alors un cycle d'horloge sera consommé. Si c'est une opérande en mémoire alignée sur un mot (donc à une adresse paire et alors *mmm* = 100, 101 ou 110), alors il se produira deux cycles d'horloge. Si c'est une opérande non alignée, trois cycles seront requis.

La dernière étape de l'instruction *mov* est de stocker la valeur dans l'emplacement de destination. Puisque la destination de l'instruction de chargement est toujours un registre[14], alors cette opération demande un cycle d'horloge.

[13]Ici, on considérera un cycle d'horloge et un cycle de mémoire comme s'ils étaient d'une durée équivalente.
[14]Dans cette version de mov, naturellement (n.d.t.)

Dans son ensemble, l'instruction *mov* prend en total entre cinq et onze cycles, selon ses opérandes et leurs adresses de début (paires ou impaires).

Le CPU fait ce qui suit pour une instruction de type *mov mem, reg* :

- Charger l'octet de l'instruction dans la mémoire (un cycle).
- Mettre à jour IP pour qu'il pointe sur l'instruction suivante (un cycle).
- Décoder l'instruction pour l'interpréter (un cycle).
- Si requis, charger l'opérande dans la mémoire (zéro cycles si le mode d'adressage est [bx], un cycle s'il s'agit du mode [xxxx], [xxxx + bx] ou xxxx et que la valeur xxxx qui suit l'opcode commence à une adresse paire, et deux cycles si la valeur de xxxx commence à une adresse impaire).
- Si requis, mettre à jour IP pour qu'il pointe après l'opérande (zéro cycles s'il n'y avait pas cette opérande, un cycle si elle était présente).
- Calculer l'adresse de l'opérande (zéro cycles si le mode d'adressage n'est pas [bx] ou [xxxx + bx], un cycle si le mode est [bx] ou deux cycles si le mode d'adressage est [xxxx + bx]).
- Obtenir du registre la valeur à stocker dans la mémoire (un cycle).
- Stocker la valeur chargée dans l'emplacement de destination (un cycle si c'est un registre, deux cycles si c'est un mot commençant à la bonne adresse ou trois cycles si l'adresse est impaire).

Le timing pour les deux dernières étapes est différent par rapport à l'autre version de *mov*, parce que, dans cette instruction-là, la donnée était lue de la mémoire ; cette version de *mov* "charge" une valeur à partir d'un registre. Et elle demande aussi de cinq à onze cycles pour s'exécuter.

Les instructions *and*, *sub*, *cmp*, *and* et *or* font ce qui suit :

- Charger l'octet de l'instruction de la mémoire (un cycle).
- Mettre à jour IP pour qu'il pointe sur la prochaine instruction (un cycle).
- Décoder l'instruction (un cycle).
- Si requis, charger une opérande constante (zéro cycles si le mode d'adressage est [bx], un cycle s'il s'agit du mode [xxxx], [xxxx + bx] ou xxxx et la valeur xxxx qui suit le opcode commence à une adresse paire, et deux cycles si cette valeur commence à une adresse impaire).
- Si requis, mettre à jour IP pour qu'il pointe après l'opérande constante (zéro cycles ou un cycle).
- Calculer l'adresse de l'opérande (zéro cycles si le mode d'adressage n'est pas [bx] ou [xxxx + bx], un cycle si le mode est [bx] ou deux cycles s'il est [xxxx + bx]).
- Obtenir la valeur de l'opérande et l'envoyer dans l'unité logique et arithmétique (ULA) (zéro cycles si elle est une constante, un cycle si elle est un registre, deux cycles si elle commence à une adresse paire et trois, dans le cas contraire).
- Charger la valeur de la première opérande (un registre) et l'envoyer à l'ULA (un cycle).
- Demander à l'ULA d'additionner, de soustraire, de comparer ou d'effectuer un AND ou un OR (un cycle).
- Garder la valeur chargée dans le registre de la première opérande (un cycle).

Ces instructions demandent entre huit et dix-sept cycles d'horloge pour exécuter.

L'instruction *not* est semblable, mais elle fonctionne un peu plus rapidement parce qu'elle a une seule opérande :

- Charger l'octet de l'instruction dans la mémoire (un cycle).
- Mettre à jour IP pour qu'il pointe sur l'octet suivant (un cycle).
- Décoder l'instruction (un cycle).
- Si requis, charger une opérande constante dans la mémoire (zéro cycles si le mode d'adressage est [bx], un cycle s'il s'agit du mode [xxxx], [xxxx + bx] ou xxxx et la valeur xxxx qui suit le opcode commence à une adresse paire, et deux cycles si la valeur de xxxx commence à une adresse impaire).
- Si requis, mettre à jour IP pour qu'il pointe après l'opérande constante (zéro cycle ou un cycle).
- Calculer l'adresse de l'opérande (zéro cycles si le mode d'adressage n'est pas [bx] ou [xxxx + bx], un cycle si le mode est [bx] ou deux cycles si le mode d'adressage est [xxxx + bx]).

- Obtenir la valeur de l'opérande et l'envoyer dans l'unité logique et arithmétique (un cycle si c'est un registre, deux cycles si elle commence à une adresse paire et trois cycles si elle commence à une adresse impaire).
- Demander à l'ULA d'effectuer un NOT (un cycle).
- Stocker la valeur chargée dans l'opérande (un cycle si c'est un registre, deux cycles si c'est un mot commençant à la bonne adresse ou trois cycles sinon).

L'instruction *not* peut prendre de six à quinze cycles pour exécuter.

Les sauts conditionnels fonctionnent comme suit :

- Charger l'octet de l'instruction dans la mémoire (un cycle).
- Mettre à jour IP pour qu'il pointe sur l'octet suivant (un cycle).
- Décoder l'instruction (un cycle).
- Charger l'adresse cible depuis la mémoire (un cycle si xxxx est une adresse paire et deux si xxxx est une adresse impaire).
- Mettre à jour IP pour qu'il pointe après l'adresse (un cycle).
- Tester les drapeaux (flags) *plus petit que* et *égalité* du CPU (un cycle).
- Si la valeur du drapeau est appropriée pour le saut conditionnel, le CPU copie la constante de 16 bits dans le registre IP (zéro cycles s'il n'y a pas de branchement, un cycle dans le cas contraire).

Le saut inconditionnel est identique en termes d'opérations à l'instruction *mov reg, xxxx*, sauf que le registre de destination est le registre IP x86 au lieu que ax, bx, cx, ou dx.

Les instructions *brk*, *iret*, *halt*, *put* et *get* sont sans intérêt ici. Elles figurent dans le jeu d'instructions principalement pour des programmes et des preuves. On ne peut pas vraiment donner un comptage des cycles puisqu'elles peuvent prendre une quantité de temps indéfinie pour compléter leur tâche.

3.3.9 Les différences entre les processeurs x86

Tous les processeurs x86 partagent le même jeu d'instructions, les mêmes modes d'adressage et exécutent leurs instructions à l'aide de la même séquence d'étapes. Alors, quelle est la différence ? Pourquoi ne pas inventer un seul processeur au lieu de quatre ?

La principale raison de soulever cette question est d'expliquer les différences de performance reliées aux quatre caractéristiques matérielles : les *queues de préchargement*, les *caches*, les *pipelines* et la *conception superscalaire*. Le processeur 886 est un "périphérique" bon marché qui n'implémente aucune de ces caractéristiques raffinées. Le processeur 8286 implémente la queue de préchargement. Le 8486 a une queue de préchargement, un cache et un pipeline. Le 8086, à part d'avoir toutes les caractéristiques précédentes est muni aussi de l'opération superscalaire. En étudiant chacun ce ces processeurs vous pouvez voir les bénéfices apportés par chacune de ces technologies.

3.3.10 Le processeur 886

Celui-ci est le membre le plus lent de la famille x86. La durée de chaque instruction a été traitée dans la section précédente. L'instruction *mov*, par exemple, demande entre cinq et douze cycles pour exécuter, selon les opérandes. Le tableau suivant fournit le timing des diverses formes des instructions du le processeur 886.

Tableau 19 : temps d'exécution pour les instructions 886

Instruction ⇨ Mode d'adressage ⇩	mov (les deux formes)	add, sub, cmp, and, or	not	jmp	jxx
reg, reg	5	7			

reg, xxxx	6-7	8-9			
reg, [bx]	7-8	9-10			
reg, [xxxx]	8-10	10-12			
reg, [xxxx+bx]	10-12	12-14			
[bx], reg	7-8				
[xxxx], reg	8-10				
[xxxx+bx], reg	10-12				
reg			6		
[bx]			9-11		
[xxxx]			10-13		
[xxxx+bx]			12-15		
xxxx				6-7	6-8

Il y a trois choses importantes à noter ici. En premier, les instructions longues prennent plus de temps pour s'exécuter. En second lieu, les instructions qui font référence à la mémoire, au lieu qu'aux registres, ont une exécution généralement plus lente ; ceci est particulièrement vrai s'il y a des états d'attente associés à l'accès mémoire (le tableau ci-dessus suppose zéro états d'attente). Finalement, les instructionsns utilisant des modes d'adressages complexes sont plus lentes. Celles qui utilisent des registres comme opérandes sont plus courtes, n'accédent pas à la mémoire et n'utilisent pas de mode d'adressage complexe. *C'est la raison pour laquelle vous devriez garder vos variables dans les registres chaque fois que vous pouvez*.

3.3.11 Le processeur 8286

La clé pour améliorer la vitesse d'un processeur est d'effectuer des opérations en parallèle. Si, dans le timing donné pour le 886, l'on pouvait effectuer deux opérations pour chaque cycle d'horloge, le CPU pourrait exécuter les instructions deux fois plus rapidement avec la même vitesse. Cependant, décider simplement d'exécuter deux opérations par cycle n'est pas aussi facile. Plusieurs étapes dans l'exécution d'une instruction partagent des *unités fonctionnelles* dans le CPU (les unités fonctionnelles sont des groupes de composantes logiques qui effectuent une opération commune, comme l'unité de contrôle et l'unité logique et arithmétique). Une unité fonctionnelle peut exécuter seulement une opération à la fois. Par conséquent, vous ne pouvez pas faire deux opérations en même temps qui utilisent la même unité fonctionnelle (par exemple, incrémenter le registre IP et faire la somme de deux valeurs). Une autre difficulté à remarquer est qu'une opération peut dépendre du résultat d'une autre. Par exemple, les deux dernières étapes de l'instruction *add* impliquent l'addition des deux valeurs et la garde du résultat. Vous ne pouvez pas stocker un résultat avant d'avoir effectué une somme. Il y a aussi d'autres ressources que le CPU ne peut pas partager entre les étapes d'une instruction. Par exemple, il y a seulement un bus de données ; le CPU ne peut pas charger un opcode en même temps qu'il écrit une information dans la mémoire. L'astuce pour concevoir un CPU qui exécute plusieurs étapes en parallèle est d'arranger ces étapes de façon à réduire les conflits ou bien d'ajouter des composants logiques de façon que deux opérations (ou plus) peuvent se produire simultanément en s'exécutant sur différentes unités fonctionnelles.

Considérez encore les étapes que *mov reg, mem/reg/const* nécessite :

- Charger l'octet de l'instruction de la mémoire.
- Mettre à jour le registre IP pour pointer sur le prochain octet.
- Décoder l'instruction pour voir ce qu'elle fait.
- Si requis, charger de la mémoire l'opérande de 16 bits de l'instruction.
- Si requis, mettre à jour IP pour qu'il pointe au-delà de l'opérande.
- Calculer l'adresse de l'opérande, si requis (c'est-à-dire bx + xxxx).
- Charger l'opérande.
- Garder la valeur trouvée dans le registre de destination.

La première opération utilise la valeur du registre IP (donc on ne peut pas exécuter en parallèle l'incrément) et le bus pour charger l'opcode de l'instruction de la mémoire. Toute étape qui suit celle-ci dépend de l'opcode chargé, donc c'est improbable qu'on peut mettre en parallèle cette étape avec toute autre.

La seconde et la troisième opération ne partagent aucune unité fonctionnelle et ne décodent pas non plus un opcode qui dépend de la valeur de IP. Donc, on peut facilement modifier l'unité de contrôle pour qu'elle incrémente IP en même temps qu'elle décode l'instruction. Ceci permet de gagner un cycle dans l'exécution de l'instruction *mov*.

La troisième et la quatrième opération ci-dessus (décoder et, de façon optionnelle, charger l'opérande de 16 bits) n'ont pas l'air de coller ensemble en parallèle, car, pour savoir s'il est nécessaire de charger une opérande de 16 bits en mémoire, le CPU a besoin de décoder d'abord l'instruction. Cependant, on peut projeter un CPU qui anticipe et charge la valeur de 16 bits pour qu'elle soit disponible en tout cas, au besoin. Mais, il y a un problème : il faut avoir l'adresse de l'opérande à charger (donc la valeur du registre IP) et on doit attendre l'incrément de IP avant d'aller charger cette opérande. Si on incrémente IP en même temps qu'on décode l'instruction, il faudra attendre le prochain cycle avant de décoder l'instruction en tant que telle.

Puisque les trois étapes suivantes sont facultatives, il y a plusieurs séquences d'instructions possibles à ce point :

#1 (étape 4, étape 5, étape 6 et étape 7) - p.e., *mov ax, [1000+bx]*
#2 (étape 4, étape 5 et étape 7) - p.e., *mov ax, [1000]*
#3 (étapes 6 et 7) - p.e., *mov ax, [bx]*
#4 (étape 7) - p.e., *mov ax, bx*

Dans la séquence ci-dessus, l'étape 7 dépend toujours des étapes qui précèdent ; par conséquent, celle-ci ne peut pas s'exécuter en parallèle avec une autre. L'étape 6 est également reliée à l'étape 4, puisque cette dernière utilise la valeur du registre IP, cependant l'étape 5 peut exécuter en parallèle avec n'importe quelle autre. On peut donc économiser un cycle dans les deux premières séquences avec le résultat que voici :

#1 (étape 4, étapes 5/6 et étape 7)
#2 (étape 4, étapes 5/7)
#3 (étapes 6 et 7)
#4 (étape 7)

Certainement, il n'y a pas de moyen de dédoubler les étapes 7 et 8 dans l'instruction *mov*, car elle doit d'abord récupérer la valeur avant de la stocker quelque part. En combinant ces étapes comme ci-dessus, on obtient ce qui suit :

- Charger l'octet de l'instruction depuis la mémoire.
- Décoder l'instruction et mettre à jour IP.
- Si requis, charger l'opérande de 16 bits de l'instruction dans la mémoire.
- Calculer l'adresse de l'opérande, si requis (c'est-à-dire bx + xxxx).
- Si besoin, charger l'opérande et mettre à jour IP pour qu'il pointe au-delà de l'opérande.
- Stocker la valeur chargée dans le registre de destination.

En ajoutant quelques circuits logiques supplémentaires dans le CPU, on a pu exécuter l'instruction *mov* avec un ou deux cycles d'horloge en moins. Cette simple optimisation fonctionne bien aussi avec beaucoup d'autres instructions.

Un autre problème avec l'exécution de l'instruction *mov* concerne l'alignement des opcodes. Considérez l'instruction *mov ax, [1000]* qui apparaît à l'adresse 100 en mémoire. Le CPU consomme un cyle pour charger son opcode et, après avoir décodé l'instruction et avoir déterminé si elle a une opérande de 16 bits, il emploie deux cycles additionnels pour charger cette opérande (car cette dernière apparaît à une adresse impaire - 101). Et ce qui est vraiment ironique est que les deux cycles additionnels employés pour charger ces deux octets sont superflus, car, d'après tout, si le CPU a trouvé l'octet le moins significatif de l'opérande quand il a saisi l'opcode de l'instruction (n'oubliez pas que les CPU x86 sont des processeurs de 16 bits et recherchent toujours des valeurs de 16 bits dans la mémoire), pourquoi ne pas enregistrer cet octet et utiliser seulement un cycle additionnel pour trouver l'autre ? Ceci ferait épargner un cycle quand l'instruction commence à une adresse paire (et l'opérande tombe dans une adresse impaire). Pour réaliser cela, il suffit d'un registre d'un octet et un peu plus de circuits logiques, et en vaut bien l'effort.

Pendant qu'on ajoute un registre pour servir de tampon aux octets des opérandes, on pourrait considérer quelques optimisations additionnelles qui peuvent utiliser la même logique. Par exemple, considérez ce qui arrive avec cette même instruction *mov* exécutée ci-dessus. Si on charge l'opcode et l'octet le moins significatif de

l'opérande dans le premier cycle et l'octet le plus significatif de l'opérande dans le second cycle, on est vraiment en train de lire quatre octets, et non trois. Le quatrième est l'opcode de l'instruction suivante. Si l'on pouvait enregistrer cet opcode jusqu'à l'exécution de l'instruction suivante, on pourrait économiser encore un cycle de son temps d'exécution, puisqu'il n'y aurait plus à charger l'octet de l'opcode. De plus, puisque le décodeur de l'instruction est inutilisé quand le CPU est en train d'exécuter l'instruction *mov*, on pourrait à ce moment décoder la prochaine exécution pendant que l'instruction courante est en train de s'exécuter, de façon à économiser un autre cycle de l'exécution de la prochaine instruction. En moyenne, il y aura recherche de l'octet supplémentaire toutes les deux instructions. Par conséquent, implémenter ce simple schème nous permettra d'éliminer deux cycles additionnels dans à peu près 50% des instructions exécutées.

Peut-on faire quelque chose à propos de l'autre 50% ? La réponse est oui. Notez que l'exécution de l'instruction *mov* ne comporte pas un accès à la mémoire pour chaque cycle d'horloge. Par exemple, quand on est en train de stocker une donnée dans le registre de destination, le bus est disponible. Pendant les périodes de temps où le bus est disponible, on peut précharger des opcodes et des opérandes et enregistrer ces valeurs pour exécuter la prochaine instruction.

L'amélioration majeure que le 8286 a eue par rapport au 886 a été la *queue de préchargement* (prefetched queue). Même si le CPU n'est pas en train d'utiliser l'unité d'interface des bus (UIB), celle-ci peut charger des octets additionnels dans le flux des instructions. Chaque fois que le CPU a besoin d'une instruction ou d'un octet d'opérande, il saisit le prochain octet disponible dans la queue de préchargement. Puisque le UIB saisit deux octets à la fois et le CPU consomme généralement moins de deux octets par cycle, tout octet que le CPU charge normalement dans le flux des instructions, sera déjà prêt dans la queue de préchargement.

Notez cependant que rien ne garantit que toutes les instructions et les opérandes seront en attente dans la queue au besoin. Par exemple, une instruction *jmp 1000*, rendrait invalide tout ce qui se trouve dans la queue. Si cette instruction apparaît aux emplacements 400, 401, et 402 en mémoire, la queue de préchargement contiendrait les octets des adresses 403, 404, 405, 406, 407, etc. Après avoir mis à jour IP à l'adresse 1000, les octets à partir de l'adresse 404 ne seront plus d'aucune utilité. Donc, le système devra s'interrompre un moment afin de charger le double-mot à l'adresse 1000 avant de continuer.

Une autre amélioration qu'on peut apporter c'est dédoubler le décodage de l'instruction avec la dernière étape de l'instruction précédente. Après que le CPU aura traité l'opérande, le prochain octet disponible dans la queue de préchargement est un opcode, et le CPU peut le décoder en prévision de son exécution. Sans doute, si l'instruction courante modifie le registre IP, tout le temps employé à décoder la prochaine instruction sera gaspillée, mais puisque ceci se produit en parallèle avec d'autres opérations, ceci ne constituera pas un grand inconvénient pour le système.

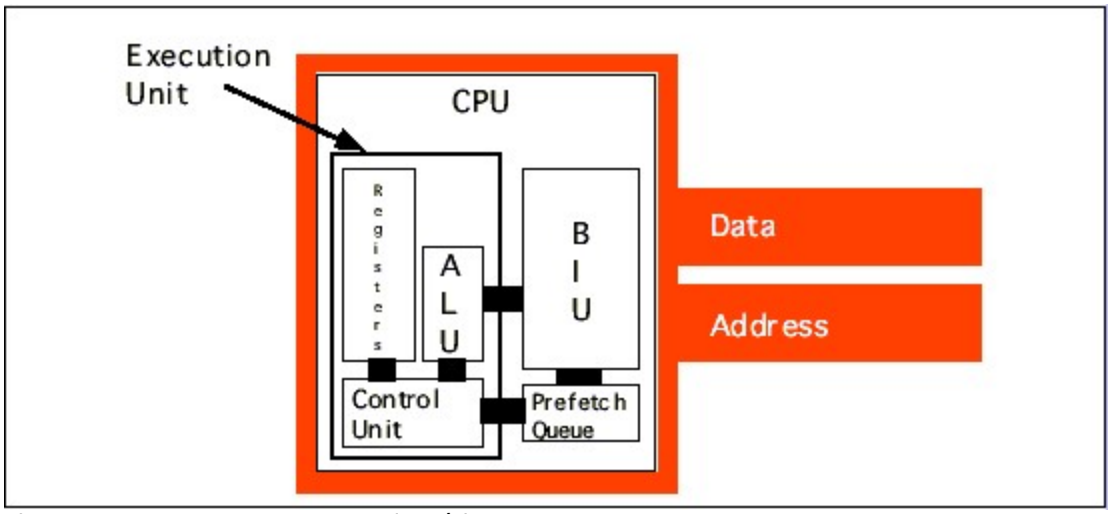


Figure 3.23 Un CPU avec une queue de préchargement

Cette séquence d'optimisations du système demande tout de même quelques modifications dans le matériel. Un diagramme du système dans son ensemble se voit dans la figure 3.23. La séquence d'exécution des instructions suppose maintenant que les événements suivants se produisent en tâche de fond :

- Si la queue de préchargement n'est pas pleine (généralement elle a une capacité allant de huit à trente-deux octets, selon le processeur) et le UIB est disponible pendant le cycle d'horloge courant, charger le prochain mot de la mémoire et à l'adresse dans IP au début du cycle[15].
- Si le décodeur de l'instruction est disponible et l'instruction courante ne demande pas une opérande, commencer à décoder l'opcode en tête de la queue de préchargement (si présent), sinon, commencer à décoder le troisième octet dans la queue de préchargement (si présent). Si l'octet désiré ne se trouve pas dans celle-ci, ne pas déclencher cet événement.

Le timing d'exécution de l'instruction fait quelques suppositions optimistes, à savoir, que tous les opcodes et toutes les opérandes d'instruction sont déjà présents dans la queue de préchargement et que l'opcode de l'instruction courante a déjà été décodé. Si cela se révèle faux, l'exécution d'une instruction 8286 sera retardée pendant que le système charge les données de la mémoire ou décode l'instruction. Ce qui suit, décrit les étapes pour chacune des instructions du 8286 :

mov reg, mem/reg/const

- Si nécessaire, calculer la somme de [xxxx+bx] (un cycle dans cette éventualité).
- Charger l'opérande source. Zéro cycles si c'est une constante (en supposant comme étant déjà présente dans la queue de préchargement), un cycle s'il s'agit d'un registre, deux cycles si c'est une adresse de mémoire paire, trois si impaire.
- Stocker le résultat dans le registre de destination. Un cycle.

mov mem, reg

- Si nécessaire, calculer la somme de [xxxx+bx] (un cycle dans cette éventualité).
- Charger l'opérande source (un registre). Un cycle.
- Stocker la source dans l'opérande de destination. Deux cycles si c'est une adresse de mémoire paire, et trois si c'est une adresse de mémoire impaire.

instr reg, mem/reg/const (où instr = add, sub, cmp, and ou *or)*

- Si nécessaire, calculer la somme de [xxxx+bx] (un cycle dans cette éventualité).
- Charger l'opérande source. Zéro cycles si c'est une constante (en supposant qu'elle était déjà présente dans la queue de préchargement), un cycle si c'est un registre, deux cycles s'il s'agit d'une adresse de mémoire paire et trois s'il s'agit d'une adresse impaire.
- Charger la valeur de la première opérande (un registre). Un cycle.
- Calculer la somme, la différence, etc. comme il se convient. Un cycle.
- Stocker le résultat dans le registre de destination. Un cycle.

not mem/reg

- Si nécessaire, calculer la somme de [xxxx+bx] (un cycle dans cette éventualité).
- Charger l'opérande source. Un cycle si c'est un registre, deux cycles s'il s'agit d'une adresse de mémoire paire et trois s'il s'agit d'une adresse impaire.
- Inverser la valeur. Un cycle.
- Stocker le résultat. Un cycle si c'est un registre, deux cycles si c'est une adresse de mémoire paire, trois, si c'est une adresse impaire.

jcc xxxx (saut conditionnel, cc = a, ae, b, be, e, ne)

- Tester le drapeau du code de condition courant (plus petit ou égal). Un cycle.
- Si les valeurs du drapeau sont appropriées pour le branchement correspondant, le CPU copie l'opérande d'instruction de 16 bits dans le registre IP. Un autre cycle.

jmp xxxx

- Le CPU copie l'opérande d'instruction de 16 bits dans le registre IP. Un cycle.

Comme nous avons vu pour le 886, nous n'allons pas considérer le temps d'exécution des autres instructions x86, car la plupart d'entre elles sont indéterminées.

Les instructions de branchements semblent s'exécuter très rapidement dans le 8286. Cependant, elles peuvent courir très lentement. N'oubliez pas que le fait de sauter d'un emplacement à un autre rend le contenu de la queue de préchargement invalide. Donc, même si les instructions de branchement ne demandent qu'un cycle

[15]Cette opération charge uniquement un octet si IP contient une valeur impaire.

pour s'exécuter, elles forcent le CPU a vider la queue de préchargement et, par conséquent, consomment plusieurs cycles pour charger l'instruction suivante, les opérandes suivantes et décoder cette instruction. Donc deux ou trois instructions après le saut sont nécessaires avant que la queue revient à un état habilitant le CPU d'exécuter des opérations en parallèle. Ceci entraîne une importante implication à l'heure de programmer : *évitez les sauts aussi souvent que possible si vous vouez que votre code soit efficace.*

Notez que les branchements conditionnels n'invalident la queue de préchargement que s'ils effectuent le saut. Si la condition est fausse, ils tombent dans la prochaine instruction qui était indiquée et continuent à utiliser les valeurs de la queue, ainsi que toute autre instruction normale. Donc, si, en écrivant vos programmes, vous pouvez déterminer quelle condition est la plus probable (par exemple, plus petite que ou ne pas plus petite que), vous pouvez arranger votre code de façon à faire échouer la condition dans le cas le plus commun, au lieu d'effectuer le saut.

La taille des instructions (en octets) peut aussi affecter la performance de la queue. Pour charger une instruction d'un octet, il ne faut jamais plus d'un cycle d'horloge, mais, pour charger une valeur de trois octets, cela demande toujours deux cycles. Par conséquent, si la cible d'une instruction de branchement constitue deux instructions d'un octet, le UIB peut charger les deux instructions en un cycle et commencer à décoder le second octet pendant que le premier est en cours d'exécution. Si, au contraire, ces instructions sont de trois octets, le CPU peut ne pas avoir assez de temps pour charger et décoder la seconde et la troisième instruction dans le laps de temps nécessaire pour qu'il termine avec la première. Vous devrez donc tenter d'utiliser des instructions courtes autant que possible, car elles améliorent la performance de la queue de préchargement.

La table suivante fournit les temps d'exécution (optimiste) des instructions sur 8286 :

Tableau 20 : temps d'exécution pour les instructions 8286

Instruction ⇨ Mode d'adressage ⇩	mov (les deux formes)	add, sub, cmp, and, or	not	jmp	jxx
reg, reg	2	4			
reg, xxxx	1	3			
reg, [bx]	3-4	5-6			
reg, [xxxx]	3-4	5-6			
reg, [xxxx+bx]	4-5	6-7			
[bx], reg	3-4	5-6			
[xxxx], reg	3-4	5-6			
[xxxx+bx], reg	4-5	6-7			
reg			3		
[bx]			5-7		
[xxxx]			5-7		
[xxxx+bx]			6-8		
xxxx				pqp[a]	2[b] 2 + pqp

a. Perte de la queue de préchargement
b. Si le saut échoue

Noter la vitesse de l'instruction *mov* sur le 8086 par rapport à celle de la version 886. La raison de cette amélioration est que la queue de préchargement permet au processeur de dédoubler l'exécution des instructions adjacentes. Cependant, cette table est plus qu'optimiste. Notez la condition : « supposer que le opcode dans la queue de préchargement est bien présent et qu'il a été bien décodé ». À ce sujet, considérez les trois séquences suivantes :

```
????:         jmp,   1000
1000:         jmp,   2000
2000:         mov,   cx, 3000[bx]
```

La seconde et la troisième instruction ne s'exécuteront pas aussi rapidement que dans l'estimation donnée dans le tableau. Chaque fois qu'on modifie la valeur du registre IP, le CPU vide la queue de préchargement et il ne peut charger et décoder la prochaine instruction, mais devra charger l'opcode, le décoder, etc., ce qui ralentit l'exécution de ces instructions. Dans cette éventualité, la seule amélioration possible a été de mettre à jour IP en parallèle avec une autre étape.

Normalement, inclure une queue de préchargement augmente les performances. C'est pourquoi Intel en fournit une pour chaque modèle 80x86, à partir du 8088. Dans ces processeurs le UIB charge constamment les données de la queue de préchargement chaque fois que le programme n'est pas en train de lire ou d'écrire.

Ces queues fonctionnent de façon optimale avec des larges bus de données. Le processeur 8286 tourne beaucoup plus rapidement que le 886 *parce qu'il peut garder pleine la queue de préchargement.* Cependant, considérez les instructions suivantes :

```
100:          mov,   ax, [1000]
105:          mov,   bx, [2000]
10A:          mov,   cx, [3000]
```

Puisque les registres ax, bx, et cx sont de 16 bits, voici ce qu'il arrive (en supposant que la première instruction a été trouvée dans la queue de préchargement et a déjà été décodée) :

- Charger l'octet de l'opcode de la queue de préchargement (zéro cycles).
- Décoder l'instruction (zéro cycles).
- Il y a une opérande pour cette instruction, donc l'obtenir de la queue de préchargement (zéro cycles).
- Obtenir la valeur de la seconde opérande (un cycle). Mettre à jour IP.
- Garder la valeur chargée dans le registre de destination (un cycle). Charger deux octets du flux de code. Décoder la prochaine instruction.

Fin de la première instruction. Il y a actuellement deux octets dans la queue de préchargement.

- Charger l'octet de l'opérande de la queue de préchargement (zéro cycles).
- Décoder l'instruction pour voir ce qu'elle fait (zéro cycles).
- S'il y a une opérande pour cette instruction, l'obtenir de la queue de préchargement (un cycle parce qu'il manque encore un octet).
- Obtenir la valeur de la seconde opérande (un cycle). Mettre à jour IP.
- Garder la valeur chargée dans le registre de destination (un cycle). Charger deux octets du flux de code. Décoder la prochaine instruction.

Fin de la seconde instruction. Il y a maintenant trois octets dans la queue de préchargement.

- Charger l'octet de l'opérande de la queue de préchargement (zéro cycles).
- Décoder l'instruction (zéro cycles).
- S'il y a une opérande pour cette instruction, l'obtenir de la queue de préchargement (zéro cycles).
- Obtenir la valeur de la seconde opérande (un cycle). Mettre à jour IP.
- Garder la valeur chargée dans le registre de destination (un cycle). Charger deux octets du flux de code. Décoder la prochaine instruction.

Comme vous pouvez le voir, la seconde instruction demande un cycle de plus que les deux autres. Ceci parce que le UIB ne peut pas remplir la queue aussi rapidement que le processeur exécute les instructions. Ce problème est amplifié quand on limite la taille de la queue de préchargement à un certain nombre d'octets. Ce problème n'existe pas dans le processeur 8286, mais il existe certainement dans les autres processeurs 80x86.

Vous verrez bientôt que les processeurs 80x86 *tendent très facilement à épuiser la queue de préchargement*. Naturellement, une fois que celle-ci est vide, le CPU attend le UIB pour charger de nouveaux opcodes de la mémoire, ce qui ralentit le programme. Exécuter de plus courtes instructions aide à garder pleine la queue. Par exemple, le 8286 peut charger deux instructions d'un octet avec un seul cycle de mémoire, mais il prend 1,5 cycles pour charger une instruction de trois octets. Normalement, il emploie plus de temps à exécuter quatre instructions d'un octet qu'une instruction de trois octets. Ceci donne à la queue du temps pour remplir et décoder de nouvelles instructions. Dans les systèmes munis d'une queue de préchargement, il est possible de trouver huit instructions de deux octets qui opèrent plus rapidement qu'un ensemble équivalent de quatre instructions de

quatre octets. La raison en est qu'avec des instructions courtes, la queue a le temps de se remplir de nouveau elle-même.

Morale de l'histoire : *en programmant un processeur avec queue de préchargement, utilisez toujours les instructions les plus courtes possibles.*

3.3.12 Le processeur 8486

L'exécution des instructions en parallèle à l'aide d'une interface bus et d'une unité d'exécution est un cas spécial de *pipelining*. Le processeur 8486 incorpore le pipelining pour améliorer les performances. À quelques exceptions près, on verra que ce moyen permet d'exécuter une instruction par cycle d'horloge.

L'avantage des queues de préchargement était que le CPU pouvait dédoubler le chargement et le décodage des instructions pendant qu'elles étaient exécutées. Autrement dit, pendant qu'une instruction était en état d'exécution l'unité d'interface des bus chargeait et décodait l'instruction suivante. Dans la mesure où vous acceptez d'ajouter des composants matériels, vous pourrez exécuter presque toutes les opérations en parallèle. C'est l'idée qui est derrière le pipeling.

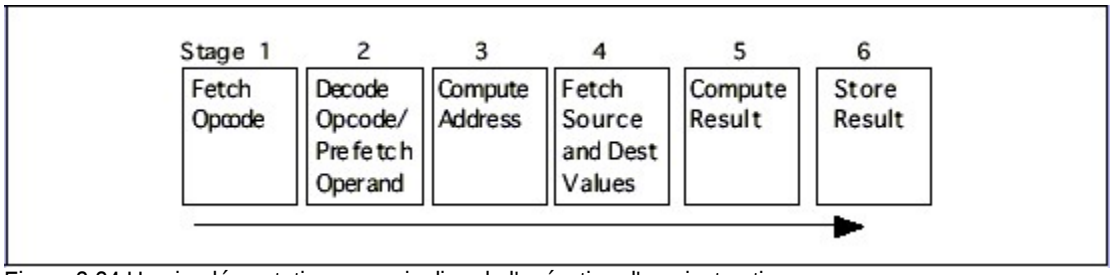


Figure 3.24 Une implémentation avec pipeline de l'exécution d'une instruction

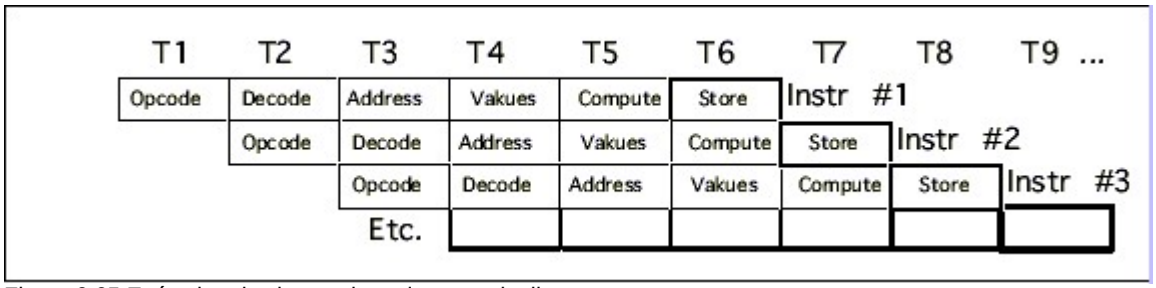


Figure 3.25 Exécution des instructions dans un pipeline

3.3.12.1 Le pipeline du 8486

Considérez les étapes nécessaires pour effectuer une opération générique :

- Charger l'opcode.
- Décoder l'opcode et (en parallèle) précharger une opérande éventuelle de 16 bits.
- Calculer le mode d'adressage complexe (par exemple [xxxx+bx]), si applicable.
- Charger la valeur source de la mémoire (s'il y a une opérande de mémoire) et la valeur du registre de destination (si approprié).
- Calculer le résultat.
- Stocker le résultat dans le registre de destination.

En supposant que vous voulez bien payer pour du silicium additionnel, vous pourriez construire un petit miniprocesseur pour traiter chacune des étapes ci-dessus. Ceci ressemblerait à quelque chose comme dans la figure 3.24.

Si vous concevez un composant matériel séparé pour chaque étape du pipeline ci-dessus, presque toutes ces étapes peuvent travailler en parallèle. Sans doute vous ne pouvez pas charger et décoder l'opcode pour une

instruction en même temps, mais vous pouvez charger un opcode pendant que vous êtes en train de décoder l'instruction précédente. Si vous avez un pipeline à *n* étapes, vous aurez normalement *n* instructions en train de s'exécuter simultanément. Le processeur 8486 a un pipeline de six étapes, il peut donc dédoubler l'exécution de six instructions séparées.

La figure 3.25 montre ce processus. T1, T2, T3, etc. représentent les "tic-tac" consécutifs de l'horloge système. A T = T1 le CPU charge l'octet du opcode pour la première instruction.

A T = T2, le CPU commence à décoder l'opcode pour la première instruction. En parallèle, il charge 16 bits de la queue de préchargement, dans l'éventualité que l'instruction a une opérande. Puisque la première instruction n'a plus besoin du circuit qui charge les opcodes, le CPU cherchera aussitôt l'opcode de l'instruction suivante en même temps qu'il décode la première instruction. Veuillez noter qu'il y a un conflit mineur ici. Le CPU est en train de charger le prochain octet de la queue de préchargement pour l'utiliser en tant qu'opérande, en même temps qu'il charge 16 bits de la queue de préchargement pour l'utiliser en tant qu'opcode. Comment peut-il faire les deux choses en même temps ? Vous verrez bientôt la solution.

A T = T3, le CPU calcule une adresse d'opérande pour la première instruction, s'il y a lieu. Le CPU ne fait plus rien sur la première instruction à moins d'utiliser le mode d'adressage [xxxx+bx]. Pendant T3, le CPU décode également l'opcode de la seconde instruction et charge toute opérande nécessaire. Finalement, le CPU charge également l'opcode de la troisième instruction. A chaque tic de l'horloge, une autre étape dans l'exécution de chaque instruction dans le pipeline est complétée et le CPU tire encore une autre instruction de la mémoire.

A T = T6, le CPU complète l'exécution de la première instruction, calcule le résultat de la seconde, etc., et, finalement, il charge l'opcode de la sixième instruction dans le pipeline. La chose importante à voir est que, après T = T5, le CPU termine une instruction à chaque cycle d'horloge. *Une fois que le CPU remplit le pipeline, il complète une instruction pour chaque cycle.* Notez que ceci reste vrai même si vous avez des modes d'adressage complexes à calculer, des opérations à charger de la mémoire ou d'autres opérations qui utilisent plusieurs cycles sur des processeurs sans pieline. Tout ce dont vous avez besoin est d'ajouter plus d'étapes pour le pipeline, et vous pourrez encore effectivement traiter chaque instruction en un cycle d'horloge.

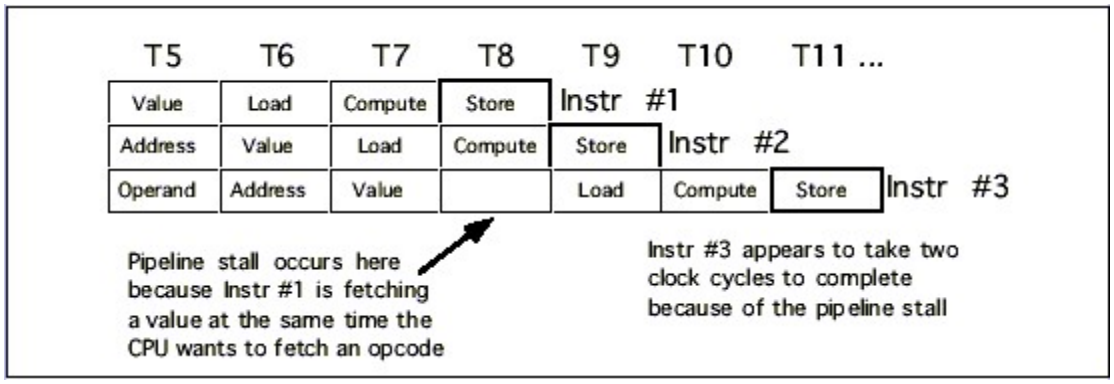
3.3.12.2 blocages du pipeline

Malheureusement, le scénario présenté dans la section précédente est un peu trop simpliste. Il y a deux inconvénients à ce pipeline : les conflits de bus et l'exécution non séquentielle du programme. Les deux problèmes peuvent augmenter le temps d'exécution moyen des instructions dans le pipeline.

Les conflits de bus ont lieu quand une instruction a besoin d'accéder à un objet en mémoire. Par exemple, si une instruction *mov mem, reg* a besoin de stocker des données en mémoire et une autre instruction *mov reg, mem* est en train d'y lire quelque chose, il peut se produire un conflit pour le bus d'adresse et le bus des données, puisque le CPU tentera simultanément de charger et d'écrire des données en mémoire.

Une manière simple de traiter les conflits de bus consiste à *bloquer le pipeline* (pipeline stall). Le CPU, devant faire face à un conflit pour le bus, donne la priorité à l'instruction la plus avancée dans le pipeline. Le CPU suspend le chargement des opcodes jusqu'à ce que l'instruction courante ne charge (ou écrive) son opérande. Ceci a pour effet que la nouvelle instruction dans le pipeline s'exécutera en deux cycles au lieu qu'en un (voir figure 3.26).

Figure 3.26 Un blocage du pipeline



Cet exemple n'est qu'un cas de conflit de bus. Il y en a d'autres. Par exemple, comme il a été précédemment noté, le fait de charger les opérandes d'une instruction demande l'accès à la queue de préchargement en même temps que le CPU doit charger l'opcode de l'instruction suivante. De plus, dans des processeurs un peu plus avancés que le 8486 (par exemple, le 80486), il y a d'autres sources de conflits de bus qui se manifestent. En considérant le simple schéma ci-dessus, il est improbable de pouvoir exécuter beaucoup d'instructions à un débit d'une par cycle d'horloge (on dit aussi, en anglais, *clock per Instruction CPI*).

Heureusement, une utilisation intelligente du système de cache peut éliminer beaucoup de blocages comme ceux dont on vient de parler. La prochaine section sur le caching décrira comment on réalise ceci. Cependant, il n'est pas toujours possible, même avec un cache, d'éviter de bloquer le pipeline. Ce que vous ne pouvez pas résoudre du côté matériel, vous pouvez l'améliorer du côté logiciel. Si vous évitez autant que possible d'utiliser la mémoire, vous pouvez réduire les conflits de bus et vos programmes s'exécuteront plus rapidement. D'autre part, utiliser des instructions courtes aide également à réduire ce problème et le blocage éventuel du pipeline.

Qu'est-ce qui se passe quand une instruction *modifie* le registre IP ? Le temps que l'instruction

```
jxx    adresse
```

complète son exécution, on a déjà commencé cinq autres instructions et on est seulement un cycle loin de l'achèvement de la première d'elles. Evidemment, le CPU ne doit pas exécuter ces instructions, sous peine de produire des résultats incorrects.

La seule solution raisonnable est de vider tout le pipeline et recommencer à charger des opcodes de nouveau. Cependant, faire cela provoque une sérieuse dégradation du temps d'exécution. Il faudra six cycles d'horloge (la longueur du pipeline du 8486), avant que l'instruction suivante puisse compléter son exécution. Bien évidemment vous devriez éviter d'utiliser des instructions qui interrompent le flux séquentiel de l'exécution du programme. Ceci montre également un autre problème - la longueur du pipeline. Plus un pipeline est long plus vous pourrez réaliser d'opérations par cycle dans le système. Cependant, allonger un pipeline peut ralentir le système si vous faites beaucoup de sauts/branchements. Malheureusement, vous ne pouvez pas contrôler le nombre d'étapes d'un pipeline. Vous pouvez, cependant, contrôler le nombre d'instructions de branchement qui apparaissent dans votre programme. Logiquement, dans un système à pipeline, vous devriez les réduire au minimum.

3.3.12.3 Le cache, la queue de préchargement et le 8486

Les concepteurs de systèmes peuvent résoudre divers problèmes concernant les conflits de bus via une utilisation intelligente de la queue de préchargement et de la mémoire cache. Ils peuvent concevoir la queue de préchargement à servir de tampon pour les données depuis le flux d'instructions et ils peuvent concevoir le cache avec des zones de données et de code séparées. Les deux techniques peuvent améliorer les performances du système et éliminer certains conflits dans le bus.

La queue de préchargement se comporte simplement comme un tampon entre le flux des instructions en mémoire et les circuits de chargement des opcodes. Maheureusement, sur le 8486 elle ne jouit pas de l'avantage qu'elle avait sur le 8286. Elle fonctionne bien sur des 8286 parce que le CPU n'est pas constamment en train d'accéder à la mémoire. Quand le CPU ne fait pas cela, le UIB peut charger des opcodes d'instruction supplémentaires pour la queue. Hélas, le CPU 8486 accède à la mémoire continuellement, puisqu'il charge un

octet d'opcode à chaque cycle d'horloge. Par conséquent, la queue de préchargement ne peut pas tirer avantage des cycles morts du bus pour charger des octets d'opcode additionnels - il n'y a pas de cycle mort. Cependant, la queue de préchargement est encore utile dans le 8486, pour une très simple raison : le UIB charge deux octets à chaque accès de mémoire et encore, certaines instructions comptent seulement un octet. Sans la queue de préchargement, le système devrait charger chaque opcode exprès, même si le UIB a déjà chargé "accidentalement" l'opcode avec les instructions précédentes. Avec la queue de préchargement, le système ne rechargera pas chaque opcode. Il les chargera une fois pour toutes et les enregistrera pour les utiliser avec l'unité de chargement des opcodes.

Par exemple, si l'on exécute deux instructions d'un octet, le UIB peut charger les deux opcodes en un cycle de mémoire, en laissant libre le bus pour d'autres opérations. Le CPU pourra utiliser ces cycles de bus disponibles pour charger des opcodes additionnels ou pour traiter avec d'autres accès à la mémoire.

Sans doute, toutes les instructions ne sont pas d'un octet. Le 8486 a deux tailles d'instruction : un octet et trois octets. Si vous exécutez divers chargements d'instructions de trois octets, ces exécutions seront plus lentes, par exemple,

```
mov    ax, 1000
mov    bx, 2000
mov    cx, 3000
add    ax, 5000
```

Chacune de ces instructions lit un octet de opcode et une opérande de 16 bits (la constante). Par conséquent, il emploie une moyenne de 1,5 cycles d'horloge pour lire chacune de ces instructions. Comme résultat, elles nécessiteront six cycles d'horloge pour exécuter, au lieu de quatre.

Encore une fois, on revient à la même règle : *les programmes les plus rapides sont ceux qui font usage des instructions les plus courtes*. Si vous pouvez utiliser de telles instructions pour accomplir certaines tâches, faites-le. La séquence suivante fournit un bon exemple :

```
mov    ax, 1000
mov    bx, 1000
mov    cx, 1000
add    ax, 1000
```

On peut réduire la taille de ce programme et augmenter sa vitesse d'exécution en le modifiant par :

```
mov    ax, 1000
mov    bx, ax
mov    cx, ax
add    ax, ax
```

Ce dernier code a seulement cinq octets de long par rapport à l'autre qui en a douze. Le code précédent aurait requis un minimum de cinq cycles d'horloge pour s'exécuter et encore plus si on ajoutait à cela des problèmes de conflits de bus. Alors que le dernier exemple en prend seulement quatre[16]. De plus, le second exemple laisse le bus libre pour trois ou quatre périodes d'horloge, donc le UIB peut charger des opcodes additionnels. Rappelez-vous : *plus court*, veut dire souvent *plus rapide*.

Malgré que la queue de préchargement libère des cycles de bus et élimine les conflits, certains problèmes existent encore. Supposez que la longueur moyenne d'une instruction dans une séquence est de 2,5 octets (obtenue en ayant trois instructions de trois octets et une instruction d'un octet). Dans un tel cas, le bus sera occupé à charger des opcodes et des opérandes. Il n'aura donc pas de temps libre pour accéder à la mémoire. En présumant aussi que certaines de ces instructions accèdent à la mémoire, le pipeline sera bloqué, ralentissant l'exécution.

Supposez que le CPU ait deux espaces de mémoire séparés, un pour les instructions et un pour les données, chacun avec son propre bus. Ceci s'appelle l'*architecture Harvard*, puisque la première machine de ce type a été construite à l'université de Harvard. Dans une machine Harvard, il n'y a pas de risque de conflit dans le bus. Le UIB peut continuer à charger des opcodes dans le bus d'instructions pendant qu'il accède à la mémoire dans le bus données/mémoire (voir figure 3.27).

[16]En realité les deux exemples seraient plus lents. Voir la section qui traite des effets de bord pour plus de détails.

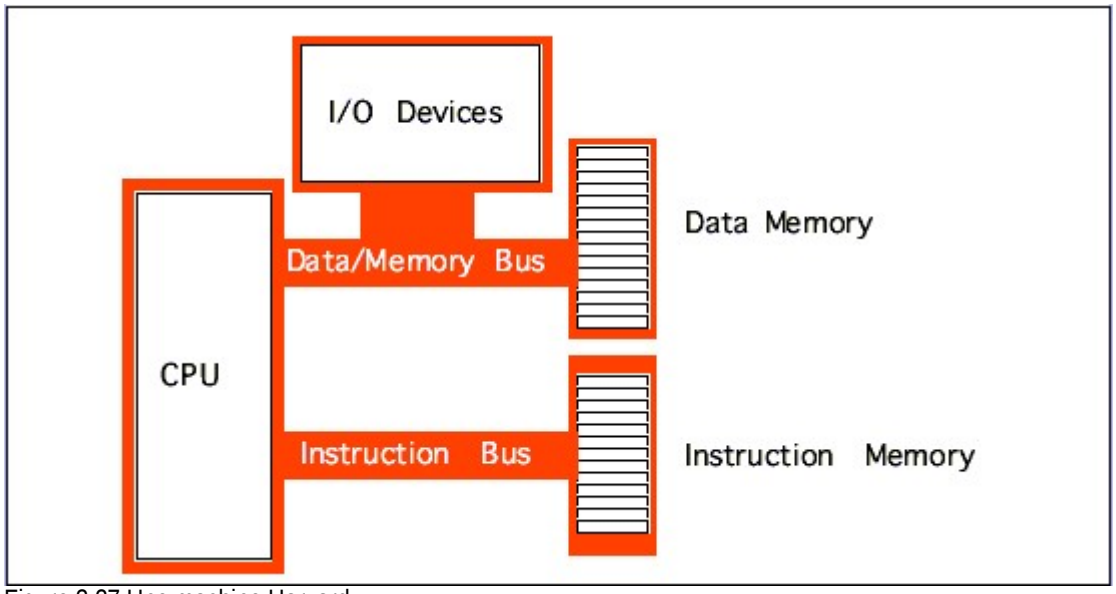


Figure 3.27 Une machine Harvard

En réalité, il y a très peu de machines Harvard. Les fiches supplémentaires requises dans le processeur pour supporter deux bus physiques séparés augmentent le coût du processeur et introduisent plusieurs autres problèmes d'ingénierie. Cependant, les concepteurs de microprocesseurs ont découvert qu'ils peuvent tirer profit des architectures Harvard, avec peu de désavantages en utilisant deux caches sur puce séparés pour les données et les instructions. Les CPU avancés se servent d'une architecture Harvard interne et d'une architecture Von Neumann externe. La figure 3.28 montre la structure du 8486 avec des caches d'instructions et de données séparées.

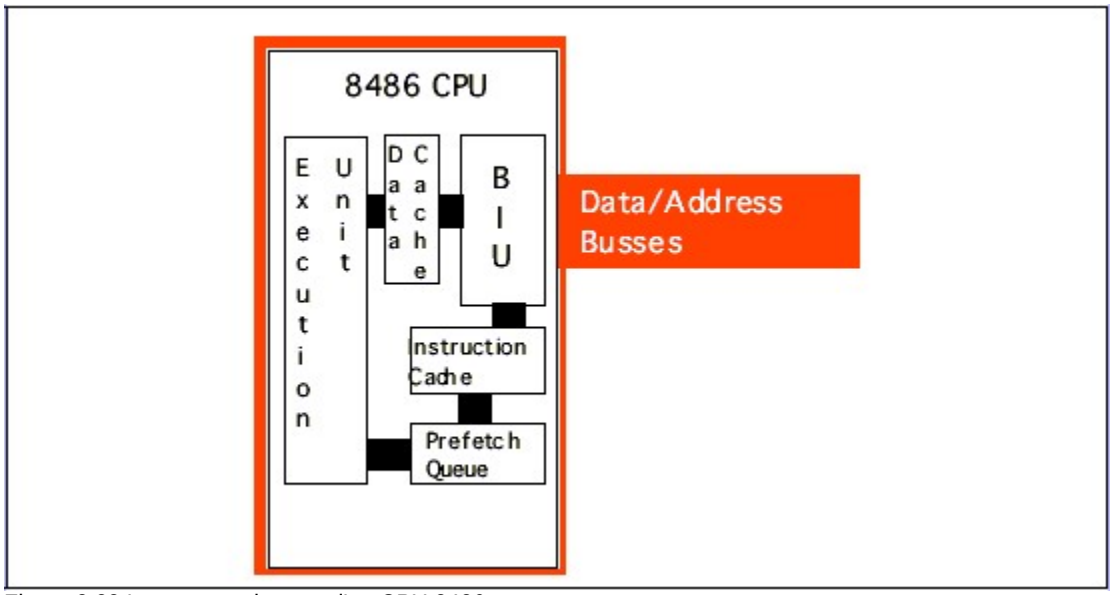


Figure 3.28 La structure interne d'un CPU 8486

Chaque chemin dans le CPU représente un bus indépendant. Les données peuvent circuler sur toutes les voies en même temps. Ceci veut dire que la queue de préchargement peut obtenir les opcodes des instructions du cache des instructions pendant que l'unité d'exécution écrit des données dans le cache des données. Maintenant, le UIB charge seulement les opcodes de la mémoire chaque fois qu'il ne peut pas en localiser dans le cache d'instructions. De même, le cache des données sert de tampon à la mémoire. Le CPU utilise le bus de données/adresses seulement quand il est en train de lire une valeur qui n'est pas sur cache ou quand il vide les données et les renvoie dans la mémoire.

Le 8486 traite le chargement des instructions des opérandes ou des opcodes d'une manière tortueuse. Grâce à l'ajout d'un circuit décodeur supplémentaire, il décode l'instruction au début de la queue de préchargement et il décode trois octets dans la queue de préchargement en parallèle. Puis, si l'instruction précédente n'a pas une opérande de 16 bits, le CPU utilise le résultat du premier décodeur ; si la dernière instruction utilise cette opérande, le CPU utilise le résultat du second décodeur.

Malgré que vous ne pouvez pas contrôler la présence, la taille ou le type de cache dans le CPU, comme programmeur en assembleur vous devez connaître la manière dont le cache fonctionne pour pouvoir écrire les meilleurs programmes. Les caches d'instructions internes sont généralement petits (8192 octets sur 80486, par exemple). Par conséquent, plus vos instructions seront courtes, plus elles pourront entrer dans le cache (vous entendez désormais parler d'instructions courtes jusqu'à l'ennui). Un conflit de bus se produira d'autant moins souvent que vous aurez plus d'instructions sur le cache. De même, le fait d'utiliser les registres pour garder les résultats temporaires permet de garder le cache des données moins occupé et de réduire donc les vidages du cache sur mémoire ou d'y aller recharger des donnés aussi souvent. *Utilisez les registres toutes les fois que vous pouvez !*

3.3.12.4 Les effets de bord sur 8486

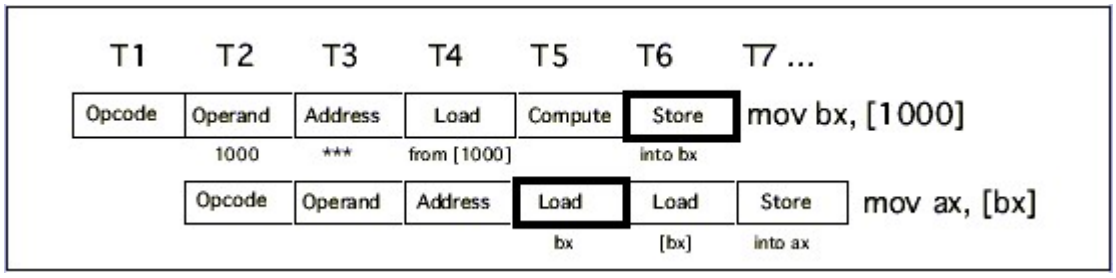
Il y a encore un autre problème dans l'utilisation du pipeline : les effets de bord sur les données (*hazards* en anglais). Regardez le profil d'exécution de la séquence d'instructions suivante :

```
mov    bx, 1000
mov    ax, [bx]
```

Quand ces deux instructions s'exécutent, le pipeline aura un aspect semblable à la figure 3.29.

Notez le problème majeur ici. Ces deux instructions chargent la valeur de 16 bits dont l'adresse est l'emplacement 1000 en mémoire. *Mais cette séquence d'instructions ne fonctionnera pas correctement !* Malheureusement, la seconde instruction utilise la valeur de bx avant que la première instruction ne charge le contenu de la mémoire à l'emplacement 1000 (T4 et T6 dans le diagramme ci-dessus).

Figure 3.29 Un effet de bord sur 8486



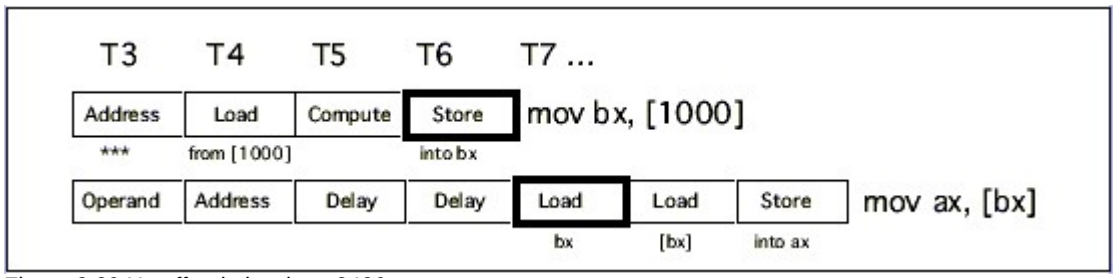


Figure 3.30 Un effet de bord sur 8486

Les processeurs CISC comme, ceux 80x86, traitent automatiquement les effets de bord[17]. Cependant, ils bloquent le pipeline pour synchroniser les deux instructions. L'exécution sur un 8486 serait similaire à la figure 3.30.

En retardant la seconde instruction de deux cycles d'horloge, le 8486 garantit que le chargement de l'instruction chargera ax à partir de la bonne adresse. Malheureusement, la seconde instruction de chargement exécutera en trois cycles au lieu d'un. Néanmoins, requérir deux cycles d'horloge supplémentaires est mieux qu'obtenir des résultats incorrects. Dans vos programmes, vous pouvez heureusement réduire l'impact des effets de bord sur la vitesse d'exécution.

Notez que les effets de bord sur les données se produisent quand l'opérande source d'une instruction était l'opérande de destination dans une instruction précédente. Il n'y a aucun mal à charger bx de l'adresse [1000] et puis charger ax de bx, *à moins que ces actions se produisent l'une après l'autre*. Supposez que la séquence du code était :

```
mov    cx, 2000
mov    bx, [1000]
mov    ax, [bx]
```

On pourrait réduire les effets de bord existant dans la séquence du code *en réarrangeant les instructions*. En le faisant, nous obtenons ceci :

```
mov    bx, [1000]
mov    cx, 2000
mov    ax, [bx]
```

Maintenant, l'instruction *mov ax* requiert seulement un cycle additionnel au lieu de deux. En insérant une autre instruction entre *mov bx* et *mov ax*, vous pouvez tout à fait éliminer les effets de bord[18].

Dans un processeur à pipeline, l'ordre des instructions dans un programme peut dramatiquement affecter la performance de celui-ci. Faites toujours attention à tous les effets de bord possibles dans vos séquences d'instructions. Eliminez-les autant que possible en réarrangeant les instructions.

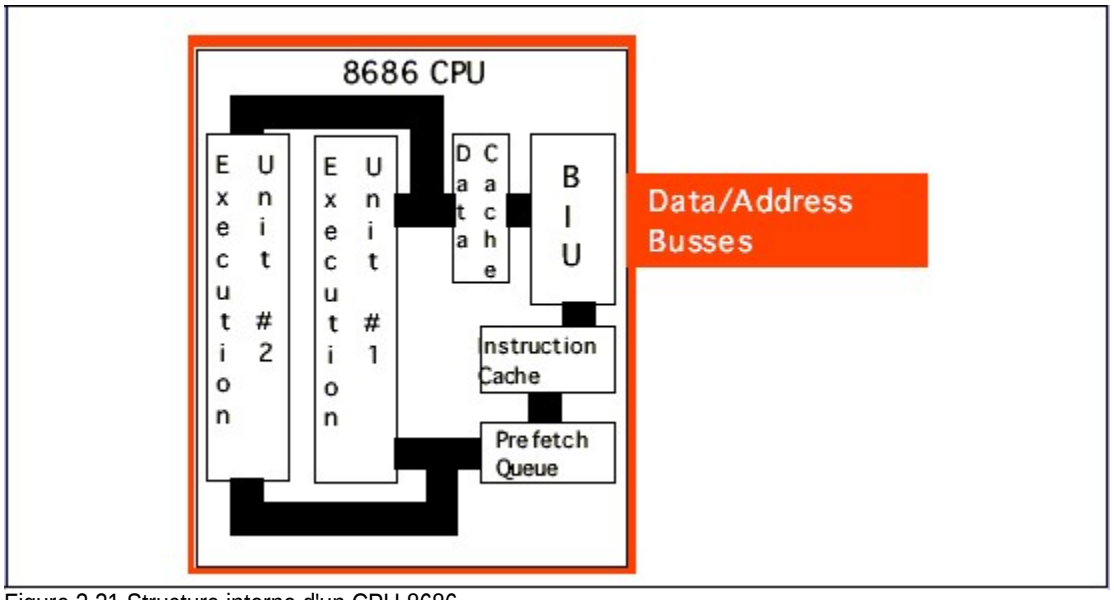


Figure 3.31 Structure interne d'un CPU 8686

3.3.13 Le processeur 8686

[17]Les puces RISC ne font pas cela. Si vous essayez cette séquence sur un processeur RISC vous n'obtiendrez pas un résultat correct.
[18]Sans doute, toute instruction qu'on insère à ce point *ne doit pas* modifier les valeurs dans les registres ax et bx.

Avec l'architecture en pipeline du 8486, on peut porter au plus les temps d'exécution à un CPI (un cycle par instruction). Peut-on exécuter des instructions encore plus rapidement ? De premier abord, on pourrait penser : « Certainement non, on ne peut pas effectuer plus d'une opération par cycle d'horloge ». Gardez à l'esprit, cependant, qu'une instruction *n'est pas* une opération simple. Dans les exemples présentés ici, chaque instruction a pris de six à huit cycles pour s'achever. En ajoutant sept ou huit unités séparées dans le CPU, on pourrait effectivement exécuter ces huit opérations en un seul cycle. Si l'on ajoute encore plus de matériel de façon à pouvoir exécuter, disons, seize opérations d'un coup, pourrait-on en venir avec un processeur à 0,5 CPI ? La réponse est «oui». Un CPU qui inclut ces composantes matérielles additionnelles est un CPU *superscalaire* et il peut exécuter plus d'une opération pendant un cycle d'horloge. Celle-ci est la fonctionnalité que le 8686 ajoute.

Un CPU superscalaire a, essentiellement, plusieurs unités d'exécution (voir figure 3.31). S'il trouve deux ou plus instructions dans le flux d'instructions (c'est-à-dire dans la queue de préchargement), qui peuvent s'exécuter indépendamment, il les exécutera.

Il y a au moins deux avantages du fait d'être superscalaire. Supposez avoir les instructions suivantes dans le flux des instructions :

```
mov    ax, 1000
mov    bx, 2000
```

S'il n'y a pas d'autres problèmes ou effets de bord dans le code environnant et tous les six octets pour ces deux instructions se trouvent actuellement dans la queue de préchargement, le CPU sera parfaitement capable de les exécuter en parallèle. Tout ceci ne requiert qu'une pièce additionnelle de silicium dans la puce du CPU.

En plus d'accélérer l'exécution d'instructions indépendantes, un CPU superscalaire peut accélérer les séquences du programme qui ont des effets de bord. Une limitation du CPU 8486 était qu'une fois que les effets de bord se produisaient, l'instruction responsable bloquait complètement le pipeline. Toute instruction suivante devait attendre la synchronisation des instructions de la part du CPU. Avec un CPU superscalaire, cependant, les instructions qui suivent l'effet de bord peuvent continuer leur exécution à travers du pipeline aussi longtemps qu'elles ne comportent pas des effets de bord elles-mêmes. Ce qui allège (mais n'élimine pas) la nécessité d'affiner l'arrangement des instructions.

Comme programmeur assembleur, *votre façon d'écrire des programmes pour un processeur superscalaire peut en affecter sérieusement les performances*. D'abord et en tout premier lieu, il y a la règle que vous connaissez déjà trop désormais : *utilisez des instructions courtes*. Plus vous le ferez, plus le CPU pourra les traiter en une seule opération et, par conséquent, plus de chances vous aurez qu'il courira plus rapidement qu'un CPI. La plupart des CPU superscalaires ne dupliquent pas complètement l'unité d'exécution. Il pourrait y avoir de multiples unités logiques et arithmétiques, de virgule flottante, etc. Ce qui veut dire que certaines séquences d'instructions s'exécuteront très rapidement et d'autres non. Vous devez étudier la composition exacte de votre CPU pour décider quelles séquences d'instructions produisent les meilleures performances.

3.4 Les entrées/sorties (E/S)

Il y a trois genres d'entrées et de sorties de base qu'un système peut utiliser : *les entrées/sorties mappées sur les entrées/sorties* (I/O mapped I/O), *les entrées/sorties mappées en mémoire* (memory-mapped I/O) et l'*accès direct à la mémoire* (Direct Memory Access ou DMA). Les E/S mappées sur E/S utilisent des instructions spéciales pour transférer les données entre l'ordinateur et le monde extérieur ; les E/S mappées en mémoire se servent, pour communiquer avec des périphériques externes, d'emplacements spéciaux de mémoire dans l'espace d'adressage du CPU ; et, finalement, l'accès DMA est une forme spéciale du second type d'accès où les périphériques lisent et écrivent dans la mémoire sans devoir passer par le CPU. Chaque mécanisme d'E/S dispose de son propre ensemble d'avantages et de désavantages et on en parlera dans cette section.

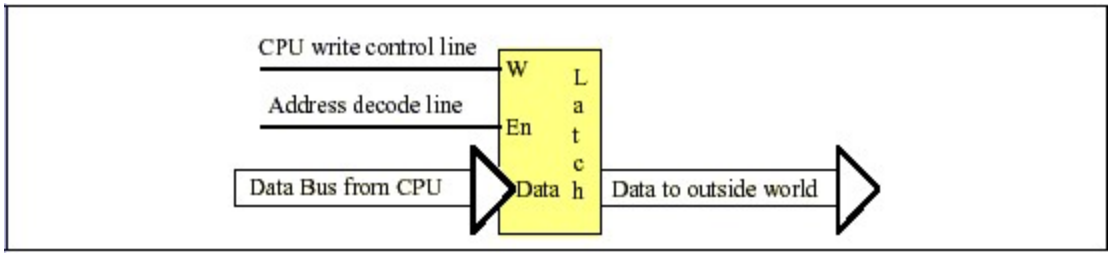
La première chose à savoir sur les entrées/sorties d'un système est que les E/S d'une architecture matérielle sont radicalement différentes par rapport aux E/S traitées dans un langage de haut niveau. Dans une véritable architecture vous ne trouverez que rarement des instructions machine se comportant comme des writeln, des printf ou même les instructions Put et Get des x86[19]. En fait, la plupart des instructions d'E/S se comportent exactement comme les instructions *mov*. Pour envoyer une donnée dans un périphérique de sortie, le CPU

[19]En effet, *get* et *put* sont similaires à writeln ou printf pour simplifier l'écriture des programmes x86.

déplace simplement la donnée dans un emplacement de mémoire spécial (dans l'espace d'adressage des E/S, si l'accès est de type E/S mappé sur E/S (voir "Les entrées/sorties" au paragraphe 3.1.3) ou dans l'espace d'adressage de la mémoire, si l'accès utilisé est de type E/S mappé en mémoire). Pour lire une donnée d'un périphérique d'entrée, le CPU déplace simplement la donnée de l'adresse (d'E/S ou de mémoire) de ce périphérique au CPU. En plus du fait qu'il y a plus d'états d'attente associés avec un périphérique donné qu'avec la mémoire, les opérations d'entrée ou de sortie sont très semblables aux opérations de lecture de d'écriture en mémoire (voir "L'accès mémoire et l'horloge système" au paragraphe 3.2.2).

Un port E/S est un périphérique qui ressemble à une cellule de mémoire de l'ordinateur, mais contient des connexions vers le monde extérieur. Un port E/S utilise un verrou (latch) au lieu d'un flip-flop pour implémenter une cellule de mémoire[20]. Quand le CPU écrit à l'adresse associée au verrou, le périphérique capture la donnée et la rend disponible sur un ensemble de fils externes au CPU et au système de mémoire (voir figure 3.32). Notez que les ports E/S peuvent être de lecture seule, d'écriture seule ou de lecture/écriture. Le port de la figure 3.32, par exemple, est un port d'écriture seulement. Puisque les sorties dans le verrou ne reviennent pas dans le bus des données du CPU, celui-ci ne peut pas lire ce que le verrou contient. Les deux adresses qui décodent et écrivent les lignes de contrôle doivent être actives pour que le verrou fonctionne ; en lisant dans l'adresse du verrou, la ligne de décodage (decode line) est active, mais non la ligne de contrôle d'écriture (write control line).

Figure 3.32 Un port de sortie créé avec un seul verrou



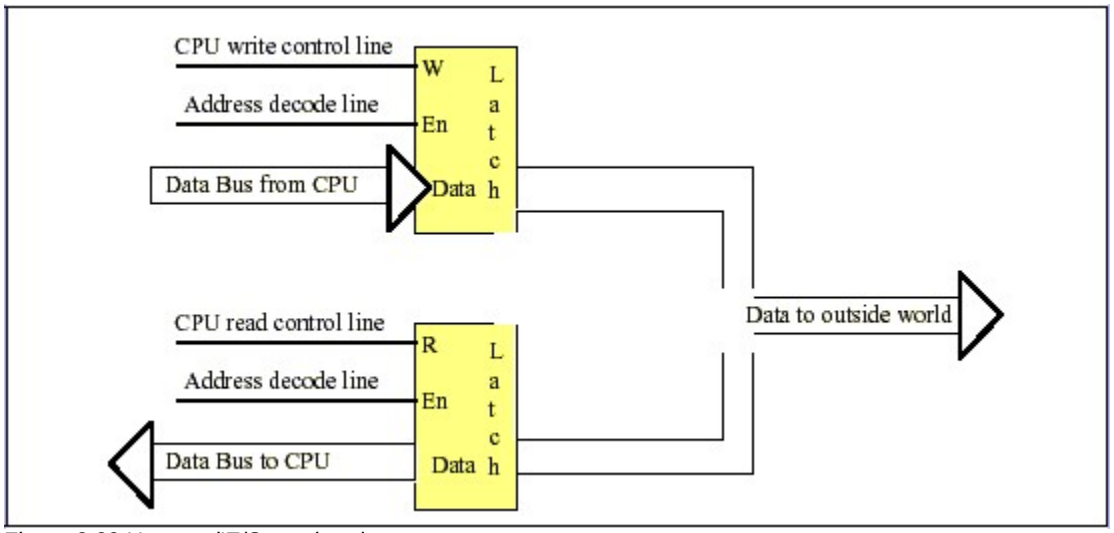


Figure 3.33 Un port d'E/S requiert deux verrous

La figure 3.33 montre comment créer un port de lecture/écriture (entrée/sortie). Les données écrites dans le port de sortie reviennent en arrière par un verrou transparent. Quand le CPU lit l'adresse décodée, les lignes de lecture et de décodage sont actives et ceci rend actif le verrou inférieur. Ce qui place les données précédemment écrites dans le port de sortie du bus des données du CPU, permettant au CPU de lire les données. Un port d'entrée en lecture seule est simplement la partie basse de la figure 3.33 ; le système ignore toute donnée écrite sur ce port.

[20]Un latch est un type de verrou qui doit être très rapidement libéré; d'ailleurs ils connaissent uniquement deux états : mis ou pas mis ! Habituellement, ils empêchent que plus d'un processus modifie la zone mémoire qu'ils protégent. Pour qu'un processus puisse altérer la zone mémoire protégée par un latch, l'autre processus qui détient ce latch doit le relâcher, n.d.t.

Un exemple parfait de port de sortie est le port parallèle de l'imprimante. Le CPU écrit généralement un caractère ASCII dans un port de sortie qui a la taille d'un octet et qui se relie au connecteur DB-25F à l'arrière du boîtier de l'ordinateur. Un câble transmet cette donnée à l'imprimante via un port d'entrée (entrée à l'imprimante) qui la reçoit. Un processeur à l'intérieur de l'imprimante convertit ce caractère ASCII en une séquence de points qui seront imprimés sur le papier.

Généralement un périphérique donné utilisera plus d'un port d'E/S. Une interface parallèle pour imprimante, par exemple, utilise trois ports : un port de lecture/écriture, un port d'entrée et un port de sortie. Le port lecture/écriture est le port de données (sa nature de lecture/écriture permet au CPU de lire le dernier caractère ASCII écrit dans le port de l'imprimante). Le port d'entrée, renvoie le signal de contrôle provenant de l'imprimante ; ces signaux indiquent quand l'imprimante est prête à accepter un autre caractère, si elle est en ligne ou encore s'il n'y a plus de papier, etc. Le port de sortie transmet le contrôle de l'information à l'imprimante pour lui indiquer, par exemple, quand des données sont disponibles pour l'impression.

Pour un programmeur, la différence entre les opérations E/S mappées sur E/S et celles mappées dans la mémoire se situe dans les instructions à utiliser. Pour des entrées/sorties de ce dernier type, toute instruction pouvant accéder à la mémoire, peut également accéder à un port d'E/S mappé dans celle-ci. Dans les x86, les instructions *mov*, *add*, *sub*, *cmp*, *and*, *or* et *not* peuvent lire la mémoire ; *mov* et *not* peuvent y écrire des données. L'accès des E/S mappées sur E/S utilise des instructions spéciales pour accéder aux ports d'E/S. Par exemple, un CPU x86 utilise les instructions *get* et *put*[21]. La famille 80x86 de Intel utilise les instructions `in` et `out`. Ces instructions fonctionnent tout comme une instruction *mov*, mais elles placent leurs adresses dans le bus des adresses des E/S au lieu que dans le bus des adresses de la mémoire (voir "Les entrées/sorties" au paragraphe 3.1.3).

Les styles E/S mappées en mémoire et E/S mappées sur E/S requièrent que le CPU déplace les données entre le périphérique externe et la mémoire principale. Par exemple, pour recevoir d'un port une séquence de dix octets d'entrée et pouvoir stocker ces octets dans la mémoire, le CPU doit lire chaque valeur depuis le périphérique et la y ranger dans la RAM. Pour des périphériques d'E/S vraiment rapides, le CPU peut être trop lent s'il débite ces données un octet à la fois. De tels périphériques contiennent généralement une interface permettant au bus du CPU de lire et écrire directement dans la mémoire. Ceci est connu comme un *accès direct à la mémoire*, car le périphérique peut directement accéder à celle-ci sans passer par le CPU. Ceci permet souvent aux opérations d'E/S de procéder en parallèle avec d'autres opérations du CPU, de façon à accroître la vitesse générale du système. Notez cependant que le CPU et le DMA ne peuvent pas en même temps utiliser le bus des adresses et de données. Par conséquent, des traitements simultanés ont lieu seulement si le CPU a un cache et si ce dernier exécute du code et accède aux données (donc le bus est libre). Néanmoins, même si le CPU doit s'interrompre et attendre les opérations DMA, les E/S sont encore très rapides, car beaucoup d'opérations de bus pendant les accès E/S mappées sur E/S ou en mémoire consistent à charger des instructions ou à accéder à des ports qui n'interfèrent pas pendant les opérations DMA.

3.5 Interruptions et sondages d'entrées/sorties

Beaucoup de périphériques E/S ne peuvent accepter des données avec un débit arbitraire. Par exemple, un PC basé sur la technologie Pentium est capable d'envoyer plusieurs millions de caractères par seconde à une imprimante, mais cette dernière est (probablement) incapable d'imprimer autant de caractères à la seconde. De même, un périphérique d'entrée, comme un clavier, ne peut pas fournir plusieurs millions de frappes par seconde (car il fonctionne à des vitesses humaines, non d'ordinateur). Le CPU nécessite quelque mécanisme pour coordonner le transfert de données entre l'ordinateur et ses périphériques.

Un moyen commun de coordonner ces transferts est de fournir certains *bits d'état* dans un port d'entrée secondaire. Par exemple, un simple bit contenant 1 dans un port d'E/S peut indiquer au CPU qu'une imprimante est prête pour accepter de nouvelles données, alors qu'un 0 peut indiquer qu'elle est occupée et que le CPU ne doit pas envoyer des données supplémentaires. De la même façon, un bit activé dans un port différent peut indiquer au CPU qu'une frappe de clavier est disponible dans le port des données de celui-ci, alors qu'un 0 dans le même bit signalerait le cas contraire. Le CPU peut tester ces bits avant de lire une touche provenant du clavier ou d'écrire un caractère sur l'imprimante.

[21] *get* et *put* sont un peu extravagantes par rapport aux véritables instructions de l'accès E/S mappées sur E/S, mais nous ignorerons la différence ici.

Considérez que le port de données de l'imprimante est mappé à l'adresse 0FFE0h et que le bit contenant l'état du port est le bit 0 de l'octet mappé à l'adresse 0FFE2h. Le code suivant attend que l'imprimante soit prête à accepter un octet de données, puis il l'écrit depuis ax dans l'octet le moins significatif du registre ax du port de l'imprimante :

```
0000: mov     bx, [FFE2]
0003: and     bx, 1
0006: cmp     bx, 0
0009: je      0000
000C: mov     [FFE0], ax
        .       .
        .       .
        .       .
```

La première instruction charge la donnée sur le port d'état d'entrée. La seconde instruction effectue un AND logique entre la valeur et 1 pour effacer les bits de 1 à 15 et fixer le bit zéro à l'état courant du port de l'imprimante. Notez que ceci produit la valeur 0 dans bx si l'imprimante est occupée et la valeur 1 dans le cas contraire. La troisième instruction vérifie bx pour voir si celui-ci contient zéro (imprimante occupée). Si c'est le cas, le programme retourne à l'emplacement zéro et répète le processus indéfiniment, jusqu'à ce que le bit d'état de l'imprimante passe à un[22].

Le code suivant fournit un exemple de lecture du clavier. Il suppose que l'état du bit du clavier soit à zéro à l'adresse 0FFE6h (zéro veut dire qu'aucune touche n'a été pressée) et que le code ASCII d'un caractère éventuel apparaîtra à l'adresse 0FFE4h quand le bit contenu dans 0FFE6h sera 1 :

```
0000: mov     bx, [FFE6]
0003: and     bx, 1
0006: cmp     bx, 0
0009: je      0000
000C: mov     ax, [FFE4]
        .       .
        .       .
        .       .
```

Le type de l'opération E/S, quand le CPU est constamment en train de tester un port pour voir si des données sont disponibles, s'appelle *interrogation* (polls), c'est-à-dire que le CPU interroge le port s'il contient des données disponibles ou bien s'il est capable d'en accepter. Mais cette technique est inéfficace. Considérez ce qui arrive dans le code précédent si l'utilisateur attend dix secondes avant de frapper une touche, le CPU tournera dans une boucle sans rien faire (à part de tester l'état du port du clavier), pendant tout ce temps.

Dans les anciens PC, (par exemple le Apple II), c'est exactement ainsi qu'un programme lit le clavier ; quand il veut lire une donnée depuis le clavier, il interroge l'état du port du clavier tant qu'une touche n'est disponible. De tels ordinateurs ne peuvent effectuer d'autres opérations pendant qu'ils attendent des frappes. Et, surtout, s'il passe trop de temps pendant la vérification de l'état du port du clavier, l'utilisateur peut presser une autre touche et le contenu de la première frappe serait perdu[23].

La solution à ce problème est de fournir un mécanisme d'interruption. Une interruption est un événement matériel externe (comme une frappe de clavier) qui cause l'interruption de la séquence de l'instruction courante du CPU et provoque l'appel d'une *routine de service d'interruption* (RSI ou IRS en anglais). Cette routine enregistre tous les registres et les drapeaux (donc elle ne dérange pas les calculs qu'elle interrompt), effectue n'importe quelle opération nécessaire pour traiter la source de l'interruption, restaure les registres et les drapeaux et finalement fait reprendre l'exécution du code interrompu. Dans beaucoup de systèmes (comme le PC), plusieurs périphériques génèrent une interruption chaque fois qu'ils ont des données disponibles ou ils peuvent en accepter du CPU. Une RSI traite rapidement ce qui a été demandé en *tâche de fond* (background), en permettant certaines autres opérations en *avant-plan* (foreground).

Les CPU qui supportent les interruptions doivent fournir certains mécanismes permettant au programmeur de spécifier les adresses de la RSI à exécuter quand une interruption a lieu. Normalement, un vecteur d'interruption

[22]Notez que ceci n'est qu'un cas hypothétique servant d'exemple. Le port parallèle de l'imprimante du PC *n'est pas* mappé aux adresses de mémoire 0FFE0h et 0FFE2h.
[23]Un port de données d'un clavier fournit généralement seulement le dernier caractère tapé, il n'offre pas un "tampon de clavier" au système.

est un emplacement de mémoire spécial contenant l'adresse de la routine à exécuter quand une interruption se produit. Les CPU x86, par exemple, contiennent deux vecteurs d'interruption : l'un pour une interruption générale et l'autre pour une interruption de *réinitialisation* (reset) (l'interruption de réinitialisation, dans beaucoup d'ordinateurs, correspond à l'action de presser le bouton *reset*). La famille Intel 80x86 supporte jusqu'à 256 vecteurs d'interruption différents.

Après qu'une RSI complète une opération, elle retourne généralement le contrôle à la tâche d'avant-plan avec une instruction spéciale de *retour d'interruption*. Dans les x86 l'instruction *iret* (interrupt return) traite ce cas. Une RSI devrait toujours terminer avec cette instruction de façon à pouvoir redonner le contrôle au programme interrompu.

Un système d'entrées géré par les interruptions utilise la RSI pour lire les données d'un port d'entrée et les mettre en tampon chaque fois qu'une donnée devient disponible. Le programme d'avant-plan peut lire ces données du tampon à son gré sans perte de données provenant du port. De la même façon, un système de sorties géré par les interruptions (générant une interruption chaque fois que le périphérique de sortie est prêt à l'accepter), peut supprimer les données d'un tampon toutes les fois où un périphérique peut accepter une nouvelle donnée.

3.6 Exercices de laboratoire

Dans ce laboratoire, vous utiliserez le programme "SIMX86.EXE" du sous-répertoire du chapitre 3. Ce programme contient un compilateur assembleur intégré, comprenant un débogueur et un interrupteur pour les CPU hypothétiques x86. Vous apprendrez comment écrire des programmes assembleur x86 de base, les assembler (les compiler), modifier le contenu de la mémoire et exécuter ces programmes. Vous expérimenterez aussi les modes d'adressage aux E/S mappées en mémoire, E/S mappées sur E/S, DMA, ainsi que les services de sondage des périphériques et d'interruption.

Dans cet ensemble d'exercices vous utiliserez le programme SIMx86.exe pour entrer, éditer, initialiser et émuler les programmes x86. Ce logiciel nécessite que vous installez deux fichiers dans votre répertoire WINDOWS\SYSTEM. Regardez à ce propos le fichier README.TXT dans le sous-répertoire CH3 pour avoir plus de détails.

3.6.1 Le logiciel SIMx86 - Quelques programmes x86 simples

Pour exécuter SIMx86, double-cliquez dans son icône ou choisissez "exécuter" (ou "run") dans le menu de démarrage de Windows et entrez le nom du chemin pour SIMx86. Le programme SIMx86 consiste en trois écrans principaux que vous pouvez sélectionner en cliquant sur les onglets appropriés, *Editor*, *Memory* ou *Emulator*. Par défaut, SIMx86 s'ouvre dans l'onglet Editor. A l'aide de cet éditeur, vous pouvez écrire et assembler les programmes x86 ; dans l'écran de la mémoire, vous pouvez voir et modifier le contenu de celle-ci ; alors que dans l'écran de l'émulateur, vous pouvez exécuter les programmes x86 et vous pouvez les voir dans la mémoire.

Le programme SIMx86 contient deux menus : File et Edit. Ils sont des menus Windows standard, donc il n'y a pas besoin de vraiment les décrire, sauf dans deux aspects. D'abord, les éléments du menu File, New, Open, Save et Save As concernent les données de la boîte de l'éditeur de texte, ils n'ont pas d'effet sur les autres sections du programme. Ensuite l'élément, Print du menu File permet d'imprimer le code source apparaissant dans l'éditeur, si l'écran de celui-ci est actif, ou bien il affiche la fenêtre entière si les écrans de la mémoire ou de l'émulateur sont actifs.

Pour voir comment le programme SIMx86 fonctionne, passez à l'écran de l'éditeur (si vous n'y êtes pas déjà). Choisissez "Open" du menu File et ouvrez "EX1.X86" dans le sous-répertoire du chapitre 3. Le contenu de ce fichier devrait ressembler à ceci :

```
mov    ax, [1000]
mov    bx, [1002]
add    ax, bx
sub    ax, 1
mov    bx, ax
add    bx, ax
```

```
add    ax, bx
halt
```

Cette courte séquence de code fait la somme des deux valeurs des emplacements 1000 et 1002, lui soustrait 1, et multiplie le résultat par trois (((ax + ax) + ax) = ax * 3). Le résultat final est laissé dans ax, puis le programme s'arrête.

Dans l'écran de l'éditeur, vous verrez trois objets : la fenêtre de l'éditeur, un champ de texte contenant l'adresse de début et un bouton ("Assemble"). La champ de texte "Starting Address" contient un nombre hexadécimal spécifiant où l'assembleur stockera le code machine pour le programme x86 que vous écrirez avec l'éditeur. Par défaut, cette adresse est zéro. La seule raison de le changer est quand vous écrivez des routines de services d'interruption, puisque l'adresse de défaut de la réinitialisation est zéro. Le bouton "Assemble" ordonne au programme SIMx86 de convertir votre code source en code machine de type x86 et de stocker le début du résultat à l'adresse de départ spécifiée en mémoire. Commencez tout de suite en pressant le bouton "Assemble", ce qui provoque l'assemblage de ce programme dans la mémoire.

Maintenant, pressez l'onglet "Memory" pour sélectionner l'écran de la mémoire. Dans cet écran, vous verrez un ensemble de 64 cases arrangées par huit rangées de huit emplacements. Du côté gauche de ces huit rangées vous pouvez voir un ensemble de huit adresses (hexadécimales) (par défaut, elles sont 0000, 0008, 0010, 0018, 0020, 0028, 0030 et 0038). Ceci vous indique que les premières huit cases en haut de l'écran correspondent aux emplacements de mémoire 0, 1, 2, 3, 4, 5, 6 et 7 ; la seconde rangée de huit cases correspond aux adresses 8, 9, A, B, C, D, E et F ; et ainsi de suite. A ce point vous devriez être capable de voir les codes machine pour le programme que vous venez juste d'assembler dans la plage 0000 - 000D. Le reste de la mémoire ne contient que des zéros.

L'écran de la mémoire vous permet de voir et éventuellement modifier 64 octets d'un total de 64 ko fournie dans les processeurs x86. Si vous voulez observer certains emplacements de mémoire autres que la plage 0000 - 003F, vous avez juste à modifier la première adresse (celle qui contient actuellement zéro). Vous allez maintenant changer l'adresse de départ à 1000, de sorte à pouvoir modifier les valeurs aux emplacements 1000 et 1002 (souvenez-vous, le programme les additionne ensemble). Tapez les nombres suivants dans les cellules correspondantes : entrez la valeur 34 à l'adresse 1000, la valeur 12 à l'adresse 1001, la valeur 01 à l'adresse 1002 et la valeur 02 à l'adresse 1003. Notez que si vous entrez une valeur hexadécimale incorrecte, le système colorera la cellule de rouge et émettra un signal sonore.

En tapant une adresse dans la cellule de l'adresse de départ, vous pouvez voir et modifier le contenu des emplacements pratiquement partout. Notez que si vous insérez une adresse qui n'est pas un nombre pair multiple de huit, le programme SIMx86 désactive jusqu'à sept cases dans la première rangée. Par exemple, si vous entrez l'adresse de départ 1002, le simulateur désactivera les premières deux cellules, car elles correspondent aux adresses 1000 et 1001. La première adresse active sera 1002. Notez aussi que SIMx86 réserve les emplacements de mémoire FFF0 à FFFF pour les entrées/sorties mappées. Par conséquent, il vous empêchera d'éditer ces adresses. La plage FFF0 - FFF7 correspond aux ports d'entrée en lecture seule (et vous pourrez voir les valeurs même si SIMx86 désactive ces emplacements). Les adresses FFF8 à FFFF correspondent aux ports de sortie en écriture seule ; si vous y jetez un œil, SIMx86 affiche des valeurs aléatoires à l'intérieur de ces dernières. Dans l'écran de la mémoire, à part la grille des adresses, vous trouverez autres deux cases éditables et un bouton. Ce bouton "Clear Memory", nettoie la mémoire en écrivant des zéros partout. Puisque le code objet et les valeurs initiales de votre programme se trouvent actuellement en mémoire, vous ne devriez pas presser ce bouton. Si vous le faites, vous aurez à assembler votre code de nouveau et à retaper les valeurs qui se trouvent de 1000 à 1003.

Les deux zones de texte dans l'écran de la mémoire vous permettent de modifier l'adresse du vecteur d'interruption et celle du vecteur des adresses (Reset Vector). Par défaut, ce dernier est à zéro. Ce qui veut dire que SIMx86 commence l'exécution du programme à l'adresse zéro chaque fois que vous réinitialisez l'émulateur. Puisque voutre programme commence à l'emplacement zéro, vous ne devriez pas changer la valeur par défaut du vecteur d'adresses.

La valeur du vecteur d'interruption est FFFF par défaut. FFFF est une valeur spéciale qui prévient SIMx86 qu'« il n'y a pas une routine de services d'interruption présente dans le système, donc ignorer toutes les interruptions ». Toute autre valeur doit être l'adresse de la RSI que SIMx86 appellera toutes les fois qu'une interruption se produira. Puisque le programme que vous venez d'assembler n'a pas de routine de service d'interruption, vous devriez laisser la case d'adresse du vecteur d'interruption à sa valeur par défaut.

Finalement, cliquez sur l'onglet "Emulator" pour donner un coup d'oeil à l'écran de celui-ci. Cet écran est beaucoup plus encombré que les deux autres. Dans le côté supérieur gauche de l'affichage, il y a un champ de texte nommé IP. Cette case garde la valeur courante du registre du pointeur d'instructions x86. Chaque fois que SIMx86 exécute un programme, il commence l'exécution avec l'instruction se trouvant à cette adresse. Quand vous pressez le bouton "Reset" (ou quand vous entrez dans SIMx86 pour la première fois), le registre IP contient la valeur trouvée dans le vecteur des adresses (Reset Vector). Si le registre ne contient pas la valeur zéro à ce moment, pressez le bouton Reset dans l'écran Emulator et le système sera reinitialisé.

Immédiatement au-dessous de la valeur IP, la page de l'émulateur *désassemble* l'instruction trouvée à l'adresse du registre IP. Celle-ci est l'instruction qui vient tout de suite après et que SIMx86 exécutera quand vous pressez les boutons "Run" ou "Step". Notez que SIMx86 n'obtient pas cette instruction de la fenêtre du code source de l'écrain Editor. Au contraire, il décode le opcode dans la mémoire (dans l'adresse trouvée dans IP) et génère la chaîne lui-même. Par conséquent, il peut y avoir des différences mineures entre l'instruction que vous venez d'écrire et l'instruction que SIMx86 affiche à cette page. Notez qu'une instruction désassemblée contient plusieurs valeurs numériques devant l'instruction concrète. La première valeur de quatre chiffres est l'adresse mémoire de l'instruction. Les deux chiffres suivants (ou les trois paires de chiffres suivantes) sont les opcodes et les opérandes d'instruction possibles. Par exemple, le code machine de l'instruction mov ax, [1000] est C6 00 10, car il s'agit des trois ensembles de chiffres figurant à ce point.

Au bas de l'instruction courante désassemblée, SIMx86 affiche les 15 instructions qu'il désassemble. L'adresse de départ de ces opérations n'est pas la valeur du registre IP ; c'est la valeur indiquée dans le champ de texte en bas de l'écran à droite qui indique l'adresse de départ. Les deux flèches à côté de cette valeur vous permettent d'incrémenter ou de décrémenter rapidement cette adresse. Supposez qu'elle est à zéro (donnez-lui cette valeur si elle ne l'a pas), pressez la fleche bas. Notez que ceci incrémente l'adresse de départ de 1. Maintenant regardez de nouveau dans la zone de texte. Comme vous pouvez voir, presser la flèche bas a produit un résultat intéressant. La première instruction (à l'adresse 0001) est "****". Les quatre astérisques indiquent que cet opcode particulier est un opcode d'instruction invalide. La seconde instruction à l'adresse 2002, est not ax. Étant donné que le programme que vous venez d'asembler ne contient pas un opcode invalide ou une instruction not ax, vous pourriez vous demander d'où ces instructions viennent. Cependant, notez l'adresse de départ à la première instruction : 0001. Ceci est le second octet de la première instruction dans votre programme. En effet, l'instruction illégale (opcode = 00) et l'instruction not ax (opcode = 10) sont en réalité un désassemblage de l'instruction mov ax, [1000] avec une opérande de deux octets. Ceci devrait bien illustrer un problème avec le désassemblage : on peut obtenir des instructions déphasées en spécifiant une adresse de départ qui est au milieu d'une instruction à plusieurs octets. Vous devrez considérer ceci à l'heure de désassembler du code.

Au milieu de l'écran de l'émulateur, il y a plusieurs boutons : Run, Step, Halt, Interrupt et Reset (la boîte "Running" est un indicateur, non un bouton). Presser le bouton Run causera l'exécution du programme (commençant par l'adresse du registre IP) en "pleine" vitesse. Presser le bouton Step permet de n'exécuter que l'instruction spécifiée dans IP et d'arrêter ensuite. Le bouton Halt, qui est actif seulement quand un programme est en train d'exécuter, a pour effet d'arrêter l'exécution. Presser le bouton Interrupt génère une interruption et presser Reset réinitialise le système (et arrête l'exécution si un programme est en cours d'exécution). Notez que presser le bouton Reset nettoie les registres x86 et charge IP de la valeur se trouvant dans le vecteur des adresses (Reset Vector).

Si l'indicateur "Running" est gris, SIMx86 n'est pas en train d'exécuter un programme. Il devient rouge pendant une exécution. Vous pouvez vous en servir comme un moyen facile de constater si un programme est en cours d'exécution, si le programme est occupé à calculer quelque chose (ou s'il est en train d'exécuter une boucle infinie) et il n'y a pas d'E/S pour indiquer l'exécution.

Les champs de texte avec les étiquettes AX, BX, CX et DX vous permettent de modifier les valeurs de ces registres pendant qu'un programme n'est pas en train de s'exécuter (ces champs sont inactifs pendant l'exécution). Ces cellules affichent également la valeur actuelle des registres quand un programme arrête son exécution ou entre les instructions quand vous arrêtez l'exécution. Notez que pendant l'exécution d'un programme, ces valeurs sont statiques et ne reflètent pas les valeurs actuelles[24].

Les boîtes à cocher "Less" et "Equal" reflètent les valeurs des drapeaux "less than" et "equal". L'instruction x86 *cmp* ajuste ces drapeaux selon le résultat d'une comparaison. Vous pouvez en voir les valeurs pendant qu'une

[24]Mais la valeur de la dernière instruction accomplie (n.d.t.).

exécution n'est pas en cours. Vous pouvez aussi les initialiser à *true* ou *false* en cliquant sur la boîte appropriée avec la souris (toujours quand SIMx86 n'est pas en train d'exécuter un programme).

Dans la section moyenne de l'écran de l'émulateur, il y a quatre indicateurs circulaires (LEDs) et quatre boutons interrupteurs. Sur chacun de ces objets, il y a une adresse hexadécimale indiquant l'adresse où ils sont mappés en mémoire. Si aux adresses correspondantes il y a des zéros, alors les indicateurs sont désactivés (ils sont blancs). Si à ces adresses il y a des 1, alors ils sont activés (ils sont rouges). Notez que ces indicateurs répondent seulement au bit zéro de leur adresse de port. Ces périphériques de sortie ignorent tout autre bit dans les valeurs écrites à ces adresses.

Les boutons interrupteurs fournissent quatre périphériques d'entrée mappés en mémoire. Pour chaque adresse se trouvant au-dessus de ces interrupteurs, SIMx86 retourne zéro si l'interrupteur est éteint (en position *off*) ; et il retourne 1 si le bouton est en position *on*. Vous pouvez allumer ou éteindre un de ces interrupteurs simplement en y cliquant dessus. Notez qu'un petit rectangle sur l'interrupteur devient rouge si l'interrupteur est en position *on*.

Les deux colonnes au côté droit de l'écran Emulator ("Input" et "Output") affichent les valeurs d'entrée lues avec l'instruction get et les valeurs entrées avec l'instruction put.

Pour ce premier exercice, vous utiliserez le bouton Step pour tracer à travers chacune des instructions du programme EX1.x86. D'abord, commencez par presser le bouton Reset[25]. Ensuite, pressez le bouton Step une fois. Notez que les valeurs des registres IP et AX changent. La valeur du registre IP change à 0003, puisqu'elle correspond à l'adresse de la prochaine instruction en mémoire, alors que la valeur du registre AX devient 1234 car c'est la valeur que vous avez placée à la locaton 1000 quand vous avez modifié l'écran de la mémoire. Continuez votre trace à travers les instructions qui restent (en cliquant Step à chaque fois), jusqu'à ce que vous obteniez la boîte de dialogue "Halt Encountered".

Pour votre journal de laboratoire : expliquez les résultats obtenus après l'exécution de chaque instruction. Notez que faire une trace le long d'un programme est une méthode excellente pour assurer que vous avez pleinement compris comment le programme fonctionne. Comme règle générale, vous devriez toujours faire la trace de tout programme que vous écrivez quand vous le testez.

3.6.2 Opérations simples de type mappées en E/S

Allez dans l'écran de l'éditeur et chargez le fichier EX2x86. Ce programme présente de nouveaux concepts, donc prenez votre temps pour étudier ce code :

```
            mov    bx,    1000
 a:         get                  ; la valeur sera gardée dans ax
            mov    [bx],  ax
            add    bx,    2
            cmp    ax,    0
            jne    a


            mov    cx,    bx
            mov    bx,    1000
            mov    ax,    0
b:          add    ax,    [bx]
            add    bx,    2
            cmp    bx,    cx
            jb     b

            put
            halt
```

La première chose à noter ce sont les deux chaînes "a:" et "b:" apparaissant dans la première colonne du listing. L'assembleur SIMx86 vous permet de spécifier jusqu'à 26 *étiquettes* en indiquant un caractère alphabétique suivi par le caractère deux points. Les étiquettes sont généralement, et dans un certain sens, des opérandes pour les instructions de saut. Par conséquent, le "jne a" veut dire : « si non égal, saute à la valeur

[25]C'est une bonne habitude de cliquer sur le bouton Reset avant d'exécuter tout programme ou de le tracer.

prise par l'instruction étiquetée avec 'a:'», au lieu de dire «si non égal, saute à l'emplacement de mémoire dix (0Ah) dans la mémoire».

Utiliser des étiquettes est beaucoup plus pratique que spécifier manuellement l'adresse mémoire de l'instruction cible, spécialement si cette dernière apparaît plus tard dans le code. L'assembleur SIMx86 calcule l'adresse de ces étiquettes et substitue l'adresse correcte pour les opérandes des instructions de saut. Notez que vous pouvez spécifier une adresse numérique dans le champ de l'opérande d'une instruction de saut. Cependant, toutes les adresses numériques doivent commencer par un chiffre décimal (même si elles sont des valeurs hexadécimales). Si votre adresse cible doit commencer par une valeur de la plage A-F, vous aurez simplement à mettre un zéro comme valeur initiale[26]. Par exemple, si "jne a" signifie « saute si non égal à l'empolacement 0Ah », vous écrirez l'instruction comme "jne 0A".

Ce programme contient deux boucles. Dans la première, il lit une séquence de valeurs de l'utilisateur tant qu'il n'entre pas la valeur zéro. Cette boucle garde chaque mot dans des adresses de mémoire successives à partir de l'adresse 1000h. Souvenez-vous que chaque mot lu par l'utilisateur nécessite deux octets ; c'est pourquoi la boucle ajoute 2 à bx dans chaque itération.

La seconde boucle scanne les valeurs entrées et calcule leur somme. A la fin de la boucle, le code affiche la somme des deux sorties en utilisant l'instruction put.

Pour votre rapport de laboratoire : faites la trace dans ce programme et décrivez comment chaque instruction fonctionne. Réinitialisez le x86 et exécutez ce programme à pleine vitesse. Entrez diverses valeurs et décrivez les résultats. Décrivez les instructions *get* et *put* et dites pourquoi elles exécutent des opérations de type E/S mappées sur E/S et non des opérations d'E/S mappées dans la mémoire.

3.6.3 Opérations de type E/S mappées dans la mémoire

Dans l'éditeur, chargez le programme EX3.x86. Ce programme prend la forme suivante (on ajoute des commentaires ici pour rendre plus claires les opérations) :

```
a:          mov    ax,    [FFF0]
            mov    bx,    [FFF2]

            mov    cx,    ax     ;Calcule BoutonInterr0 AND BoutonInterr1
            and    cx,    bx
            mov    [FFF8], cx

            mov    cx,    ax     ;Calcule BoutonInterr0 OR BoutonInterr1
            or     cx,    bx
            mov    [FFFA], cx

            mov    cx,    ax     ;Calcule BoutonInterr0 XOR BoutonInterr1
            mov    dx,    bx     ;Souvenez-vous que xor = AB' + A'B
            not    cx
            not    dx[27]
            and    cx,    bx
            and    dx,    ax
            or     cx,    dx
            mov    [FFFC], cx

            not    cx             ;Calcule BoutonInterr0 = BoutonInterr1
            mov    [FFFE], cx     ;Souvenez-vous que  égal = NOT XOR

            mov    ax,    [FFF4] ;Lit le troisième bouton interrupteur
            cmp    ax,    0      ;Regarde s'il est activé
            je     a             ;Répéter ce programme tant que off
            halt
```

[26]La raison de ceci est que, si l'adresse d'une opérande de l'instruction *jmp* commençait par une lettre, l'assembleur la prendrait pour une chaîne de caractères. On utilise donc cette notation pour permetre à l'assembleur de reconnaître de façon nette que l'opérande est un nombre (n.d.t.).

[27]Il y avait une erreur ici dans l'original. On voyait en effet "not bx". Mais, ceci n'a pas de sens, d'autant plus que, en ouvrant EX3.x86, on voit tout de suite que "not dx" est la bonne instruction, ndt.

Les emplacements 0FFF0h, 0FFF2h et 0FFF4h correspondent aux trois premiers boutons interrupteurs de l'écran Emulator. Ce sont des périphériques ayant des E/S mappées en mémoire qui mettent 0 ou 1 dans l'emplacement mémoire correspondant selon que l'état du bouton est activé ou éteint. Les adresses 0FFF8h, 0FFFAh, 0FFFCh et 0FFFEh correspondent aux quatre indicateurs circulaires (LEDs). Écrire un zéro dans cet emplacement équivaut à mettre les indicateurs dans l'état éteint, alors qu'écrire 1 les allume.

Ce programme effectue les fonctions logiques *and*, *or*, *xor* et *xnor* (non *xor*) pour les valeurs lues dans les premiers deux boutons interrupteurs. Il affiche le résultat dans les quatre indicateurs de sortie. Et il lit la valeur du troisième bouton d'interruption pour déterminer quand arrêter. Quand ce bouton est à la position *on*, le programme termine son exécution.

Pour votre journal : exécutez ce programme et expérimentez les quatre possibles combinaisons de éteint/allumé des premiers deux boutons interrupteurs. Placez vos résultats dans votre rapport de laboratoire.

3.6.4 Exercices DMA

Dans cet exercice vous commencerez l'exécution d'un programme (EX4.x86) qui examine et effectue des opérations sur les valeurs trouvées dans la mémoire. Puis vous passerez à l'écran Memory et vous en modifierez les valeurs (vous accéderez directement à la mémoire pendant l'exécution du programme) ; ceci simule un périphérique utilisant le DMA.

Le programme EX4.x86 commence son exécution en initialisant à zéro l'emplacement de mémoire 1000h. Puis, il boucle jusqu'à ce qu'une de deux conditions est satisfaite - soit que l'utilisateur change le bouton interrupteur FFF0 ou qu'il modifie la valeur à l'emplacement 1000. Changer l'état du bouton interrupteur à l'adresse FFF0 arrête l'exécution du programme. Changer la valeur de l'emplacement de mémoire 1000h transfère le contrôle à une section du programme qui fait la somme de *n* mots, où *n* est la nouvelle valeur de l'emplacement 1000h. Le programme additionne les mots apparaissant dans des adresses contiguës en mémoire à partir de l'adresse 1002h. Il ressemble à ce qui suit :

```
d:          mov   cx,   0      ;nettoie la location 1000h avant le test
            mov   [1000], cx

; La boucle suivante teste le changement de l'adresse 1000h ou bien si le
; bouton d'interruption à l'emplacement FFF0 se trouve à la position "on"

a:          mov   cx,   [1000] ;Contrôle du changement de 1000h.
            cmp   cx,   0      ;Si oui, sauter dans la section qui additionne
            jne   c            ;les valeurs

            mov   ax,   [fff0] ;Si l'adresse 1000h contient encore zéro,
            cmp   ax,   0      ;lire l'interrupteur fff0 et voir s'il est
            je    a            ;à l'état off. Si oui, boucler en arrière.
            halt               ;Sinon, quitter le programme

;Le code suivant fait la somme d'un nombre de "cx" mots contigus à partir de
;l'emplacement 1002h. Après avoir effectué les opérations, afficher le résultat.

c:          mov   bx,   1002   ;Initialisation de BX pour qu'il pointe sur un
                               ;tableau de données
            mov   ax,   0      ;Initialisation de la somme
b:          add   ax,   [bx]   ;Additionner ax avec l'élément suivant du
                               ;tableau
            add   bx,   2      ;Faire pointer bx sur l'élément suivant
            sub   cx,   1      ;Décrémenter le compteur d'éléments.
            cmp   cx,   0      ;Tester si toutes les valeurs ont été
                               ;additionnées.
            jne   b
            put                ;Afficher la somme et recommencer.
            jmp   a
```

Chargez ce programme dans SIMx86 et assemblez-le. Passez à l'écran Emulate, pressez le bouton Reset, assurez-vous que l'interrupteur FFF0 soit en position *off* et finalement exécutez le programme. Une fois qu'il est en exécution, passez à l'écran de la mémoire en cliquant sur l'onglet Memory. Modifiez l'adresse de départ pour qu'elle corresponde à l'adresse 1000. Faites passser l'emplacement 1000h à la valeur 5. Revenez à l'écran d''émulation. En supposant que les emplacements de mémoire 1002 à 100B contiennent tous des zéros, le programme devrait l'afficher dans la colonne des sorties.

Retournez à la page de la mémoire. Qu'est-ce que l'emplacement 1000h contient maintenant ? Changez les octets les moins significatifs des mots aux adresses 1002, 1004 et 1006 pour y mettre les valeurs 1, 2 et 3 respectivement. Faites passer la valeur de l'emplacement 1000h à 3. Passez de nouveau à la page de l'émulateur. Décrivez les sorties dans votre journal de laboratoire. Essayez d'entrer d'autres valeurs dans la mémoire. Pressez le bouton interrupteur de l'adresse FFF0 quand vous voulez quitter l'exécution du programme.

Pour votre rapport de laboratoire : expliquez comment ce programme utilise l'accès DMA pour fournir ses entrées. Exécutez différents tests avec des valeurs différentes à l'emplacement 1000h et différentes valeurs dans le tableau de données commençant à l'adresse 1002. Inscrivez les résultats dans votre rapport.

Pour aller plus loin : Placez la valeur 12 dans l'emplacement de mémoire 1000. Expliquez pourquoi le programme affiche alors deux valeurs, au lieu d'en afficher une.

3.6.5 Exercices sur les E/S pilotées par les interruptions

Dans cet exercice, vous chargerez deux programmes en mémoire : un programme principal et une routine de service d'interruptions. Cet exercice sert à montrer l'usage des interruptions et des routines associées.

Le programme principal (EX5a.x86) compare constamment les emplacements de mémoire 1000h et 1002h. Si elles ne sont pas égales, le programme affiche le contenu de l'emplacement 1000h et copie la valeur à l'emplacement 1002h, puis répète le processus. Le programme répète cette boucle jusqu'à ce que l'utilisateur mette le bouton interrupteur de l'emplacement FFF0 à la position *on*. Le code est le suivant :

```
a:          mov    ax,    [1000] ;Charge la donnée de l'emplacement 1000 et
            cmp    ax,    [1002] ;contrôle si le contenu est le même que
            je     b            ;celui de l'emplacement 1002h. Si oui, vérifier
            put                  ;ce qui contient l'adresse FFF0. Si les deux
            mov    [1002], ax    ;valeurs sont différentes, afficher la valeur
                                 ;contenue dans 1000h et les rendre égales.

b:          mov    ax,    [fff0] ;Tester l'interrupteur FFF0 pour voir s'il
            cmp    ax,    0      ;faut quitter le programme.
            je     a
            halt
```

La routine d'interruption (EX5b.x86) se trouve à l'adresse mémoire 100h. Quand une interruption a lieu, cette RSI incrémente simplement la valeur à l'adresse 1000h, en chargeant cette valeur dans ax, en y ajoutant 1 et en stockant de nouveau cette valeur à l'emplacement 1000h. Après ces instructions, la RSI retourne le contrôle au programme principal. La routine de services d'interruption contient le code suivant :

```
            mov    ax,    [1000] ;Incrémenter de 1 l'emplacement 1000h et
            add    ax,    1      ;retourner le contrôle au code interrompu
            mov    [1000], ax
            iret
```

Vous devez charger et assembler les deux fichiers avant d'essayer d'exécuter le programme principal. Commencez par charger ce dernier (EX5a.x86) dans la mémoire et assemblez-le à l'adresse zéro. Puis, chargez la routine (EX5b.x86) dans la mémoire, spécifiez 100 comme adresse initiale et assemblez le code. **Attention :** *si vous oubliez de charger l'adresse de départ vous effacerez le programme principal pendant l'assemblage de la routine. Si ceci arrive, vous devrez répéter toute la procédure dès le début.*

Après avoir assemblé le code, la prochaine étape est d'initialiser le vecteur d'interruption pour qu'il contienne l'adresse de la routine. Pour ce faire, passez à l'écran de la mémoire. La cellule du vecteur d'interruption contient normalement la valeur 0FFFFh (qui indique que l'interrupteur est désactivé). Changez le contenu avec la valeur 100 pour qu'il contienne l'adresse de la routine. Ceci active également le système d'interruption.

Finalement, passez à l'écran émulateur, assurez-vous que le bouton interrupteur de l'adresse FFF0 soit à la position *off*, pressez le bouton Reset et commencez l'exécution. Normalement, rien ne devrait arriver. Maintenant, pressez le bouton d'interruption et observez les résultats.

Pour votre journal : décrivez la sortie du programme chaque fois que vous pressez le bouton d'interruption. Expliquez chaque étape que vous devriez suivre si vous vouliez placer la routine à l'adresse 2000h au lieu de l'adresse 100h.

Pour aller plus loin : écrivez une routine de service d'interruption faisant quelque chose de simple. Exécutez le programme principal et pressez le bouton d'interruption pour tester votre code. Vérifiez le bon fonctionnement de votre routine.

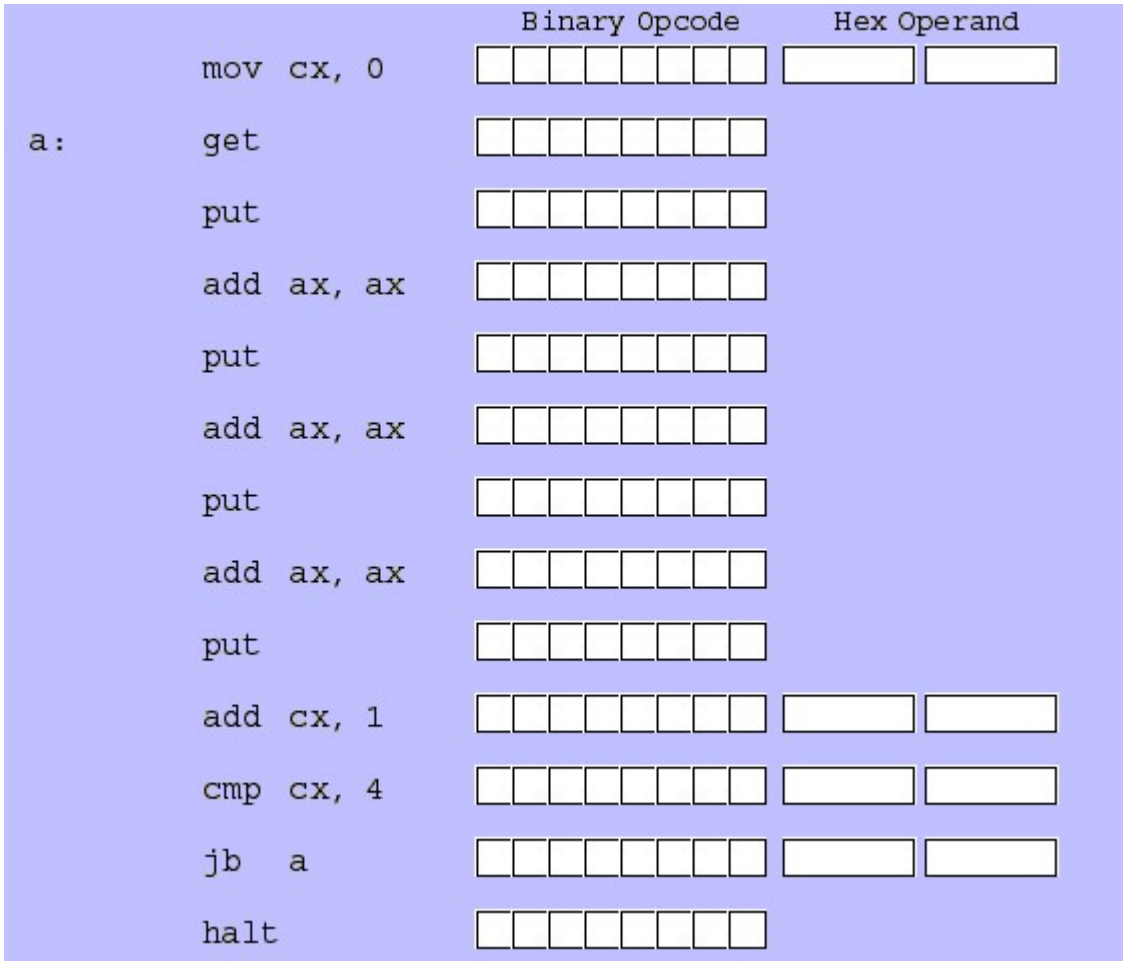
3.6.6 Exercices de programmation en langage machine et d'encodage des instructions

Jusqu'à présent vous avez créé des programmes en langage machine avec l'assembleur incorporé SIMx86. Un assembleur est un programme qui traduit un fichier source en format ASCII contenant des représentations textuelles en un programme en langage machine. Le programme assembleur vous permet d'économiser considérablement le travail consistant à traduire des instructions humainement lisibles en code machine. Même si cela est pénible, vous pouvez effectuer vous-mêmes cette traduction. Dans cet exercice vous allez créer quelques programmes très courts en langage machine en encodant les instructions et en entrant les opcodes hexadécimaux dans la mémoire via l'écran Memory.

En utilisant les techniques d'encodage trouvées dans les figures 3.19, 3.20, 3.21 et 3.22, écrivez les valeurs hexadécimales pour les opcodes à côté de chacune des instructions à la page suivante.

Vous pouvez supposer que le programme commence à l'adresse zéro et, par conséquent, l'étiquette "a" sera à l'adresse 0003, puisque l'instruction *mov cx, 0* a trois octets de long.

Pour votre rapport de laboratoire : entrez les opcodes hexadécimaux et les opérandes dans la mémoire à partir de l'adresse zéro, en vous servant de l'éditeur de mémoire. Imprimez ces valeurs et incluez-les dans votre rapport. Passez à l'écran d'émulation et désassemblez le code à partir de l'adresse zéro. Vérifiez que ce code soit le même que le code assembleur ci-dessous. Imprimez une copie du code désassemblé et incluez-la dans votre rapport. Exécutez ce programme et vérifiez si son fonctionnement est correct.



3.6.7 Exercices de Code Automodificateur

Dans l'exercice de laboratoire précédent, vous avez découvert que le système ne différencie pas réellement les données et les instructions en mémoire. Vous avez pu entrer des valeurs hexadécimales et le processeur x86 a su les traiter comme une séquence d'instructions exécutables. Il est également possible pour un programme de stocker les données directement dans la mémoire, puis de les exécuter. Un programme est automodificateur s'il crée ou modifie certaines instructions qu'il exécute.

Considérez le programme x86 suivant (EX6.x86) :

```
            sub    ax,    ax
            mov    [100], ax

a:          mov    ax,    [100]
            cmp    ax,    0
            je     b
            halt

b:          mov    ax,    00C6
            mov    [100], ax
            mov    ax,    0710
            mov    [102], ax
            mov    ax,    A6A0
            mov    [104], ax
            mov    ax,    1000
            mov    [106], ax
```

```
        mov    ax,    8007
        mov    [108], ax
        mov    ax,    00E6
        mov    [10A], ax
        mov    ax,    0E10
        mov    [10C], ax
        mov    ax,    4
        mov    [10E], ax
        jmp    100
```

Ce programme écrit le code suivant à l'emplacement 100 et puis l'exécute :

```
        mov    ax,    1000
        put
        add    ax,    ax
        add    ax,    [1000]
        put
        sub    ax,    ax
        mov    [1000], ax
        jmp    0004              ;0004 est l'adresse de l'étiquette a:
```

Pour votre rapport de laboratoire : exécutez le programme EX7.x86 et vérifiez s'il génère le code ci-dessus à partir de l'emplacement 100.

Bien que ce programme illustre le principe du code automodificateur, il ne fait pas grand chose d'utile. Comme règle générale, on n'utilise pas le code automodificateur dans la manière montrée ci-dessus, où un segment écrit une certaine séquence d'instructions et puis les exécute. Normalement, la plupart de ces programmes trouvent leur utilité dans la modification des instructions déjà existantes et souvent seulement les opérandes de ces instructions.

Les codes automodificateurs sont rares dans les programmes en assembleur modernes. Ces programmes sont difficiles à lire, à comprendre et à déboguer et ils sont souvent instables. Les programmeurs s'en servent quand le pouvoir d'une certaine architecture de CPU fait défaut pour résoudre un certain problème. Les derniers processeurs de la famille Intel 80x86 n'ont pas de lacunes à propos des instructions et de modes d'adressage, donc il est très rare de trouver des programmes de ce genre[28]. Cependant, l'ensemble des instructions des processeurs x86 est très faible, donc on y trouve certainement des exemples où les codes automodificateurs se révèlent utiles.

Un bon exemple d'un défaut architectural dans les processeurs x86 concerne les sous-routines. L'ensemble des instructions x86 ne fournit pas de moyen (direct) pour appeler une sous-routine ou revenir. Cependant, vous pouvez facilement simuler un appel et un retour en utilisant l'instruction *jmp* et du code automodificateur. Considérez la sous-routine suivante qui se situe à l'adresse 100 en mémoire :

```
        ; Convertisseur d'entiers en binaires.
        ; Il attend une valeur entière non signée dans AX.
        ; Il la convertit en une chaîne de 0 et 1 et la stocke dans la
        ; mémoire à partir de l'emplacement 1000h.

        mov    bx,    1000        ;Adresse de départ de la chaîne.
        mov    cx,    10          ;16 (10h) chiffres dans un mot.
a:      mov    dx,    0           ;Supposer que le bit courant est 0.
        cmp    ax,    8000        ;Voir si le bit le plus significatif de
                                  ;AX est 0 ou 1.
        jb     b                  ;Brancher si le bit le plus significatif
                                  ;de AX est à 0.
        mov    dx,    1           ;Le contenu de AX est 1, le signaler.
b:      mov    [bx],  dx          ;Stocker 0 ou 1 dans la prochaine
                                  ;adresse pour les chaînes.
        add    bx,    1           ;Faire pointer BX à l'emplacement
                                  ;suivant de la chaîne.
```

[28]Certains virus ou certaines techniques de protection de code des programmes font usage de codes automodificateurs pour rendre difficile leur détection ou leur dépassement.

```
add    ax,    ax          ;AX = AX * 2 (opération de décalage
                          ;à gauche).
sub    cx,    1           ;décompte des 16 bits.
cmp    cx,    0           ;Répéter 16 fois.
ja     a
                          ;Retourner à l'appelant via le code
jmp    0                  ;automodificateur.
```

La seule instruction qu'un programme modifie dans cette sous-routine est la toute dernière instruction de saut. Celle-ci doit transférer le contrôle à la première instruction au-delà du *jmp* dans le code appelant qui transfère le contrôle à cette sous-routine ; l'appeleur doit garder l'adresse de retour dans l'opérande de l'instruction *jmp* du code ci-dessus. Il s'avère que que l'instruction *jmp* est à l'adresse 120h (en supposant que le code commence à l'adresse 100h). Par conséquent, l'appeleur doit stocker l'adresse de retour à l'emplacement 121h (l'opérande de l'instruction *jmp*). Le "programme principal" suivant fait trois appels à la sous-routine ci-dessus :

```
mov    ax,    000c        ;Adresse de l'instruction BRK ci-
                          ;dessous.
mov    [121], ax          ;Stocker dans JMP comme adresse de
                          ;retour.
mov    ax,    1234        ;Convertir 1234h en binaire.
jmp    100                ;"Appeler" la sous-routine.
brk

mov    ax,    0019        ;Adresse de l'instruction BRK ci-
                          ;dessus.
mov    [121], ax          ;
mov    ax,    fdeb        ;Convertir 0FDEBh en binaire.
jmp    100                ;
brk

mov    ax,    26          ;Adresse de l'instruction halt ci-
                          ;dessous.
mov    [121], ax
mov    ax,    2345        ;Convertir 2345h en binaire.
jmp    100
halt
```

Chargez la sous-routine (EX7s.x86) dans SIMx86 et assemblez-la à partir de l'emplacement 100h. Ensuite, chargez le programme principal (EX7m.x86) dans la mémoire et assemblez-le à partir de l'adresse zéro. Passez à l'écran d'émulation et vérifiez que toutes les adresses de retour (0Ch, 19h et 26h) sont correctes. Également, vérifiez si l'adresse de retour doit être écrite à l'emplacement 121h. Puis, exécutez le programme. Il exécutera une instruction *brk* après chacun des premiers deux appels. Cette instruction met temporairement le programme en pause. A ce point, vous pouvez passer à l'écran de la mémoire et regarder aux emplacements 1000-100F. Ils devraient contenir la conversion pseudo binaire de la valeur passée à la sous-routine. Une fois vérifié que cette conversion est correcte, retournez à l'écran d'émulation et pressez le bouton Run pour continuer l'exécution du programme après le *brk*.

Pour votre rapport de laboratoire : décrivez comment les codes automodificateurs fonctionnent et expliquez en détail comment le dernier code l'utilise pour simuler des instructions d'appel et de retour. Expliquez les modifications nécessaires pour faire passer le programme principal à l'adresse 800h et la sous routine à l'adresse 900h.

Pour plus d'expériences : Modifiez le programme et la sous-routine de façon à les faire fonctionner correctement aux adresses fournies ci-dessus (800h et 900h).

3.7 Projets de programmation

Note : vous devez écrire ces programmes à l'aide du code du langage assembleur utilisé dans le programme SIMx86. Incluez un document de spécification, un plan de test, un listing et un échantillon de la sortie du programme.

1) Le jeu d'instructions du x86 n'inclut pas d'instruction de multiplication. Ecrivez un court programme qui lit deux valeurs de l'utilisateur et affiche leur produit (astuce : souvenez-vous qu'une multiplication n'est rien de plus qu'une addition répétée).

2) Créez une sous-routine appelable qui effectue la multiplication dans le problème ci-desus. Passez à la sous-routine les deux valeurs à multiplier via les registres AX et BX. Retournez le produit dans le registre CX. Utilisez la technique de code automodificateur décrite à la section 3.6.7.

3) Écrivez un programme lisant deux nombres de deux bits dans les interrupteurs FFF0/FFF2 et FFF4/FFF6. En traitant ces bits comme des valeurs logiques, votre code devra calculer la somme sur trois bits de ces deux valeurs (un résultat de deux bits plus retenue). Utilisez les équations logiques de l'additionneur complet (full adder) étudié dans le chapitre précédent. *N'effectuez pas une somme en utilisant simplement l'instruction add (x86)*. Affichez le résultat dans les indicateurs situés aux adresses FFF8, FFFA et FFFC.

4) Écrivez une sous-routine qui attend une adresse dans BX, un compteur dans CX et une valeur dans AX. Elle devrait écrire CX copies de AX dans les mots successifs de la mémoire à partir de l'adresse située dans BX. Écrivez un programme principal qui appelle cette sous-routine plusieurs fois avec des adresses différentes. Utilisez le mécanisme de retour et d'appel de sous-routines utilisé dans les exercices de laboratoire pour illustrer le principe du code automodificateur.

5) Écrivez la fonction logique générique du chapitre deux pour un processeur x86. Truc : *add ax, ax* fait un décalage à gauche de la valeur dans AX. Vous pouvez tester pour voir si le bit le plus significatif est activé pour vérifier si la valeur de AX est plus grande que 8000h.

6) Écrivez un programme qui lit un numéro de fonction générique à partir d'une fonction à quatre entrées fournie par l'utilisateur et qui écrit le résultat dans les indicateurs (LEDs) tout en continuant à lire de nouvelles valeurs depuis l'état des boutons d'interruption.

7) Écrivez un programme lisant un tableau de mots à partir de l'adresse 1000h et d'une longueur spécifiée dans la valeur contenue dans CX. Le programme doit localiser la valeur maximale dans ce tableau. Affichez la valeur après que celle-ci a été lue dans le tableau.

8) Écrivez un programme qui effectue une opération de complément à deux sur un tableau commençant à l'adresse 1000h. CX devrait contenir le nombre de valeurs dans ce tableau. Basez-vous sur le fait que chaque élément est un entier de deux octets.

9) Écrivez un programme qui produit un "light show" en faisant clignoter alternativement les indicateurs du simulateur SIMx86. Il devrait effectuer ceci en écrivant un ensemble de valeurs aux adresses de ces indicateurs, en produisant un délai d'une certaine durée (par l'entremise d'une boucle vide), puis en répétant l'opération indéfiniment. Les valeurs à écrire aux adresses des indicateurs devront être stockées dans un tableau en mémoire ; le programme charge un nouveau jeu de données à chaque itération de la boucle.

10) Écrivez un programme simple qui trie les mots des emplacements 1000 à 10FF en ordre croissant. Vous pouvez utiliser un simple algorithme de tri par insertion. Le code Pascal pour un tel tri est :

```
for i := 0 to n - 1 do
       for j := i + 1 to n do
              if(memory[i] > memory[j]) then
              begin
                     temp := memory[i];
                     memory[i] := memory[j];
                     memory[j] := temp;
              end;
```

3.8 Résumé

Ecrire de bons programmes en langage assembleur demande de fortes connaissances du matériel sous-jacent. Connaître simplement le jeu d'instructions est insuffisant. Pour produire les meilleurs programmes, vous devez comprendre comment le matériel les exécute et comment il accède aux données.

La plupart des systèmes modernes stockent les programmes et les données dans le même espace de mémoire (architecture Von Neumann). Comme la plupart des machines Von Neumann, un système 80x86 ordinaire se sert de trois composantes majeures : l'*unité centrale de traitement* (CPU), les *entrées/sorties* (E/S) et la *mémoire*. Voir :

· "Les composants de base du système", paragraphe 3.1

Les données voyagent entre le CPU, les périphériques d'E/S et la mémoire à l'aide des *bus système*. La famille 80x86 emploie trois bus majeurs : le *bus des dresses*, le *bus de données* et le *bus de contrôle*. Le bus des adresses porte un nombre binaire qui spécifie à quel emplacement de mémoire ou à quel port d'E/S le CPU souhaite accéder ; le bus des données transporte les données entre le CPU et la mémoire ou entre le CPU et les E/S ; le bus de contrôle transmet d'importants signaux qui déterminent si le CPU est en train de lire ou d'écrire des données de mémoire ou s'il est en train d'accéder à un port d'E/S. Voir :

· "Les bus système", paragraphe 3.1.1
· "Le bus de données", paragraphe 3.1.1.1
· "Le bus de adresses", paragraphe 3.1.1.2
· "Le bus de contrôle", paragraphe 3.1.1.3

Le nombre des lignes de données dans le bus de données détermine la *taille* du processeur. Quand on dit qu'un processeur est un processeur *de huit bits*, on veut dire que son bus des données contient huit lignes. Ceci ne veut pas dire que la taille effective du CPU est de huit bits. Voir :

· "Les bus de données" et "La 'Taille' du processeur" au paragraphe 3.1.1.1

Le bus des adresses transmet un nombre binaire du CPU à la mémoire ou aux entrées/sorties pour sélectionner un élément particulier de la mémoire ou d'un port d'E/S. Le nombre de lignes dans ce bus établit le nombre maximal d'emplacements de mémoire auxquels le CPU peut accéder. Des tailles typiques de bus des adresses dans les CPU 80x86 sont de 20, 24 et 32 bits. Voir :

· "Le bus des adresses" au paragraphe 3.1.1.2

Les CPU 80x86 ont aussi un bus de contrôle contenant divers signaux nécessaires à des opérations système correctes. L'horloge système, les signaux de contrôle lecture/écriture et les contrôles d'E/S ou de mémoire sont des exemples de plusieurs lignes apparaissant dans le bus de contrôle. Voir :

· "Le bus de contrôle" au paragraphe 3.1.1.3

La mémoire est la partie de la machine où le CPU stocke les instructions des programmes et les données. Dans les systèmes basés sur la technologie 80x86, la mémoire apparaît comme un tableau linéaire d'octets, chacun avec sa propre adresse unique. L'adresse du premier octet en mémoire est 0 et l'adresse du dernier octet disponible est 2^n-1, où *n* est le nombre de lignes dans le bus des adresses. Les processeurs 80x86 stockent des mots en deux emplacements consécutifs de mémoire. L'octet le moins significatif du mot est l'adresse la plus basse des deux ; l'octet le plus significatif suit immédiatement l'octet le moins significatif à l'emplacement suivant supérieur. Quoiqu'un mot consomme deux adresses de mémoire, quand on se sert des mots, on utilise simplement l'adresse de l'octet le moins significatif. Les doubles mots consomment quatre octets consécutifs en mémoire. L'octet le moins significatif apparaît à l'adresse la plus basse des quatre, alors que l'octet le plus significatif est l'adresse la plus haute. "L'adresse" d'un double-mot est aussi l'adresse de l'octet le moins significatif. Voir :

· "La mémoire" au paragraphe 3.1.2

Les CPU avec des bus de 16, 32 ou 64 bits organisent généralement la mémoire par banques. Une mémoire de 16 bits en utilise deux de huit bits chacune, une mémoire de 32 bits en utilise quatre de huit bits chacune et les systèmes de 64 bits en utilisent huit de huit bits chacune. Accéder à un mot ou à un double-mot à la même adresse dans toutes les banques est plus rapide qu'accéder à un objet parsemé entre deux adresses dans des banques différentes. Par conséquent, vous devriez aligner les mots pour qu'ils commencent à une adresse paire

et les double-mots pour qu'ils commencent à une adresse multiple de quatre. Alors que vous pouvez placer des octets individuels à n'importe quelle adresse. Voir :

· "La mémoire" au paragraphe 3.1.2

Les CPU 80x86 fournissent un espace de 16 bits séparé pour les adresses d'E/S permettant au CPU d'accéder à n'importe quel des 65 536 ports disponibles. Un périphérique typique connecté à un IBM PC utilise seulement dix de ces lignes d'adresses, en limitant le système à seulement 1024 ports différents. Le plus grand bénéfice à utiliser un espace d'adressage des E/S au lieu de mapper touts les périphériques d'E/S en mémoire est que ces dernières n'ont pas besoin d'empiéter sur l'espace de mémoire adressable. Pour différencier les E/S et les accès de mémoire, il y a des lignes de contrôle spéciales dans les bus de système. Voir :

· "Le bus de contrôle", paragraphe 3.1.1.3
· "Les entrées/sorties", paragraphe 3.1.3

L'horloge système contrôle la vitesse à laquelle le processeur effectue des opérations de base. La plupart des activités du CPU ont lieu dans le rebord montant ou tombant du tic de l'horloge. Les exemples incluent l'exécution des instructions système, les accès à la mémoire et le contrôle des états d'attente. Plus rapide est l'horloge système, plus rapide sera l'exécution des programmes ; cependant, la mémoire devrait être aussi rapide que l'horloge ou vous aurez besoin d'introduire des états d'attente, qui ralentissent dramatiquement le système. Voir :

· "Le « timing » du système", paragraphe 3.2
· "L'horloge système", paragraphe 3.2.1
· "L'accès mémoire et l'horloge système", paragraphe 3.2.2
· "Les états d'attente", paragraphe 3.2.3

La plupart des programmes montrent *une localisation des références*. Ou bien ils ont accès au même emplacement de mémoire répétitivement et pendant une courte période (*localisation temporelle*) ou bien ils accèdent à des emplacements de mémoire proches pendant une courte période (*localisation spatiale*). Un système de *mémoire cache* exploite ce phénomène pour réduire les états d'attente. Une petite mémoire cache peut réaliser des moyennes d'accès allant jusqu'à 85-95% de probabilités de succès. Les *systèmes de cache à deux niveaux* utilisent deux caches différents (normalement un dans la puce du CPU et l'autre dans un chip séparé sur la carte mère), pour atteindre des performances système encore meilleures. Voir :

· "La mémoire cache", paragraphe 3.2.4

Les CPU comme ceux de la famille 80x86 divisent l'exécution d'une instruction machine en diverses étapes séparées, chacune réquérant un cycle d'horloge. Ces étapes incluent le chargement d'un opcode d'une instruction, le décodage de cet opcode, le chargement des opérandes de l'instruction, le calcul des adresses de mémoire, l'accès à la mémoire, l'exécution de l'opération de base et le stockage du résultat. Dans un CPU très simpliste, une instruction simple peut prendre divers cycles d'horloge. La meilleure manière d'améliorer les performances d'un CPU est d'exécuter plusieurs opérations internes de façon simultanée. Un simple schème est de mettre une instruction dans la queue de préchargement du CPU. Ceci vous permettra de gagner du temps en dédoublant les opcodes et les opérations de chargement et décodage, ce qui permet souvent de diminuer de moitié le temps d'exécution. Une autre alternative est d'utiliser un pipeline d'instructions de façon à pouvoir exécuter plusieurs instructions en parallèle. Finalement, on peut également concevoir un CPU superscalaire, qui exécute deux ou plus instructions simultanément. Toutes ces techniques améliorent l'exécution des programmes. Voir :

· "Le processeur 886", paragraphe 3.3.10
· "Le processeur 8286", paragraphe 3.3.11
· "Le processeur 8486", paragraphe 3.3.12
· "Le processeur 8686", paragraphe 3.3.13

Bien que les processeurs superscalaires et munis de pipeline améliorent grandement les performances, obtenir les meilleurs résultats requiert un traitement soigneux de la part du programmeur. Dans les programmes médiocrement écrits, les blocages du pipeline et les effets de bord peuvent causer une perte majeure de performances. En étant donc attentif à l'organisation de la séquence des instructions dans vos programmes vous pouvez permettre une exécution deux ou trois fois plus rapide. Voir :

· "Le pipeline du 8486", paragraphe 3.3.12.1
· "Blocages du pipeline", paragraphe 3.3.12.2
· "Cache, queue de préchargement et le 8486", paragraphe 3.3.12.3
· "Les effets de bord sur 8486", paragraphe 3.3.12.4
· "Le processeur 8686", paragraphe 3.3.13

Le système d'entrées/sorties est le troisième composant majeur des machines Von Neumann (la mémoire et le CPU constituent les deux autres). Il y a trois manières primaires pour transporter les données entre le système et le monde extérieur : les E/S mappées sur E/S, le mappage des E/S en mémoire et le DMA. Pour plus d'informations, voir :

· "Les entrées/sorties (E/S)" au paragraphe 3.4

Pour améliorer les performances du système, la plupart des ordinateurs modernes utilisent les interruptions pour alerter le CPU quand une opération d'E/S est complète. Ceci permet au CPU de continuer avec d'autres traitements au lieu d'attendre qu'une opération d'E/S soit achevée (sondages (polls) du port d'E/S). Pour plus d'informations sur les interruptions et les sondages des opérations E/S voir :

· "Interruptions et sondages d'E/S" au paragraphe 3.5

3.9 Questions

1. Quels sont les trois composants qui constituent une machine Von Neumann ?
2. Quelle est le propos :

   a) Du bus système
   b) Du bus des adresses
   c) Du bus de données
   d) Du bus de contrôle
3. Quel bus définit la "taille" du processeur ?
4. Quel bus contrôle la quantité de mémoire à votre disposition ?
5. Est-ce que la taille du bus des données contrôle la valeur maximale que le CPU peut traiter ? Expliquez.
6. Quelles sont les tailles des bus de données des processeurs :

   a) 8088 b) 8086 c) 80286 d) 80386sx
   e) 80386 f) 80486 g) 80586/Pentium
7. Quelles sont les tailles des bus des adresses des processeurs ci-dessus ?
8. De combien de banques de mémoire disposent les processeurs ci-dessus ?
9. Expliquez comment adresser un mot dans une mémoire adressable par octets (autrement dit dans quelle adresse). Expliquez comment stocker un double-mot.
10. Combien d'opérations mémoire seront nécessaires pour lire un mot dans l'adresse indiquée pour les processeurs suivants ?

Tableau 21 : Cycles de mémoire pour adresser des mots

	100	101	102	103	104	105
8088						
80286						
80386						

11. Répétez l'exercice ci-dessus avec de doubles-mots :

Tableau 22 : Cycles de mémoire pour adresser des doubles-mots

	100	101	102	103	104	105
8088						
80286						
80386						

12. Expliquez quelles sont les meilleures adresses pour stocker des variables octet, mot et double-mot dans les processeurs 80286 et 80386.

13. Combien d'emplacements d'E/S différents pouvez-vous adresser dans les puces 80x86 ? Et combien sont normalement disponibles dans un PC ?

14. Quelle est la fonction de l'horloge système ?

15. Qu'est-ce qu'un cycle d'horloge ?

16. Quelle est la relation entre fréquence et période d'horloge ?

17. Combien de cycles d'horloge sont requis pour chacun des processeurs suivants pour lire un octet de la mémoire ?

    a) 8088 b) 8086 c) 80486

18. Qu'est-ce que cela signifique "temps d'accès à la mémoire" ?

19. Qu'est-ce que c'est un *état d'attente* ?

20. Si vous utilisez un 80486 aux vitesses d'horloge suivantes, combien d'états d'attente sont requis si vous disposez d'une mémoire RAM de 80ns (sans considérer d'autres délais) ?

    a) 20 Mhz b) 25 Mhz c) 33 Mhz d) 50 Mhz e) 100 Mhz

21. Si votre CPU tourne à 50 Mhz, une mémoire vive de 20ns ne sera probablement assez rapide pour produire zéro états d'attente. Expliquez pourquoi.

22. Puisqu'une mémoire qui s'exécute en moins de 10ns est disponible, pourquoi tous les systèmes ne fonctionnent pas à zéro états d'attente ?

23. Expliquez comment le cache opère pour économiser quelques états d'attente.

24. Quelle est la différence entre localisation de référence spatiale et localisation de référence temporelle ?

25. Expliquez où les localisations de référence spatiale et temporelle ont lieu dans le code Pascal suivant :

```
while i < 10 do begin
       x := x * i;
       i := i + 1;
end;
```

26. Comment la mémoire cache améliore-t-elle les performances d'une section de code qui faut usage de la localisation spatiale de référence ?

27. Sous quelles circonstances le cache ne vous sera d'aucune utilité pour économiser des états d'attente ?

28. Quel est le nombre effectif (en moyenne) d'états d'attente avec lesques les systèmes suivants opèrent ?

    a) 80% de fonctionnalité d'accès (hit ratio ou HT) au cache sans états d'attente, 10 états d'attente (WS) pour la mémoire, 0 WS pour le cache.
    b) 90% de fonctionnalité du cache ; 07 WS pour la mémoire ; 0 WS pour le cache.
    c) 95% de fonctionnalité du cache ; 10 WS pour la mémoire ; 1 WS pour le cache.
    d) 50% de fonctionnalité du cache ; 02 WS pour la mémoire ; 0 WS pour le cache.

29. Quelle est la finalité des systèmes de cache à deux niveaux ? Qu'économisent-ils ?

30. Quel est le nombre effectif d'états d'attente sur les systèmes suivants ?

    a) 80% de fonctionnalité (hit ratio ou HR) d'accès au cache primaire sans états d'attente ; 95% de fonctionnalité d'accès au cache secondaire avez 2 WS ; 10 WS pour l'accès à la mémoire principale.
    b) 50% de HR pour le cache primaire sans états d'attente ; 98% de HR pour le cache secondaire avec 1 WS ; 5 WS pour l'accès à la mémoire principale.
    c) 95% de HR pour le cache primaire avec 1 WS ; 98% de HR pour le cache secondaire avec 4 WS ; 10 WS pour l'accès à la mémoire principale.

31. Expliquez la fonction de *l'unité d'interface des bus*, de *l'unité d'exécution* et de *l'unité de contrôle*.

32. Pourquoi faut-il plus d'un cycle d'horloge pour exécuter une instruction ? Donnez certains exemples à l'aide des modèles x86.

33. Comment une queue de préchargement vous permet d'économiser du temps ? Donnez certains exemples.

34. Comment un pipeline vous permet (au moins par approximation) d'exécuter une instruction par cycle d'horloge ? Donnez un exemple.

35. Qu'est-ce qu'un effet de bord ?

36. Qu'est-ce qui arrive sur un 8486 quand un effet de bord a lieu ?

37. Comment pourriez-vous éliminer les effets de bord ?

38. De quelle manière une instruction de saut (JMP/Jcc) affecte

    a) la queue de préchargement ?
    b) le pipeline ?

39. Quel est un blocage du pipeline ?

40. À côté des bénéfices évidents produits par la réduction des états d'attente, comment un cache peut-il améliorer les performances d'un système à pipeline ?

41. Qu'est-ce que c'est une architecture Harvard ?

42. De quelle manière un CPU superscalaire est-il censé accélérer l'exécution ?

43. Quelles sont les deux techniques principales que vous devriez utiliser sur un CPU superscalaire pour assurer une exécution de code aussi rapide que possible ? (Note : il s'agit de détails mécaniques ; les "meilleurs algorithmes" ne comptent pas ici).

44. Qu'est-ce que c'est une interruption ? Comment améliore-t-elle les performances du système ?

45. Qu'est-ce qu'on entend par "sondage" des E/S ?

46. Quelle est la différence entre les E/S mappées en mémoire et les E/S mappées sur les E/S ?

47. Le DMA est un cas spécial d'E/S mappée en mémoire. Expliquez.