ENSEIGNEMENT DE PROMOTION SOCIALE

Cours de

STRUCTURE DES ORDINATEURS

- Le processeur -

Version provisoire

H. Schyns

Février 2013

Sommaire

1. Position du problème

Revenons un instant au schéma de principe d'un ordinateur :

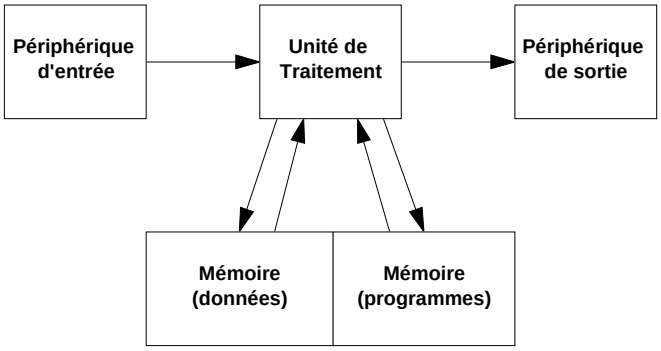


fig. 1.1 Schéma de principe d'un ordinateur

L'un des rôles de l'ordinateur est de traiter les données :

- les données à traiter sont soit introduites au moyen d'un périphérique d'entrée, soit récupérées dans les systèmes de mémoire de l'ordinateur.
- les commandes et procédures de traitement suivent le même chemin : soit depuis le périphérique d'entrée, soit depuis la mémoire.
- le processeur (*ang.: **C**entral **P**rocessing **U**nit*) constitue l'unité de traitement. Il réalise la suite des opérations logiques et arithmétiques décrites dans un programme; il ordonne le déplacement des données d'une partie de l'ordinateur vers une autre; il prend des décisions pour sauter à telle ou telle autre suite d'instructions.

Le processeur consiste en un circuit électronique extrêmement complexe qui peut accomplir des **opérations élémentaires** appelées **instructions**. Chaque modèle de processeur est construit autour d'un lexique plus ou moins vaste d'instructions.

Quand on examine le fonctionnement du processeur, il est important de garder à l'esprit qu'il est toujours en train de traiter un programme. Le premier programme est chargé par le BIOS au moment de la mise sous tension. Les programmes se chargent, s'exécutent, de déchargent et suivent sans interruption jusqu'au moment où l'utilisateur arrête la machine. Le système d'exploitation est, par excellence le programme (ou la bibliothèque de programmes) qui reste "actif" en permanence pour surveiller le clavier, la souris, les ports, etc. ([1]).

Techniquement parlant, le processeur idéal doit :

- travailler rapidement,
- être fiable,
- traiter des données volumineuses,
- traiter des instructions complexes,
- anticiper les demandes,
- être bon marché,
- être économe en énergie.

Ainsi que nous le verrons, ces différents aspects sont au centre de l'amélioration continue des performances des processeurs.

1 Qu'un programme soit actif ne signifie pas qu'il occupe 100% des ressources du processeur.

2. Les instructions

2.1. Le modèle des couches d'abstraction

Puisque le rôle du processeur est d'exécuter des instructions, il est nécessaire de définir ce que l'on entend par instruction et la manière dont celles-ci sont codées dans la machine

Revenons un instant sur le **modèle en couches** superposées vu dans le chapitre d'introduction. Chaque couche possède son langage particulier, c'est à dire un jeu d'instructions qui lui est propre.

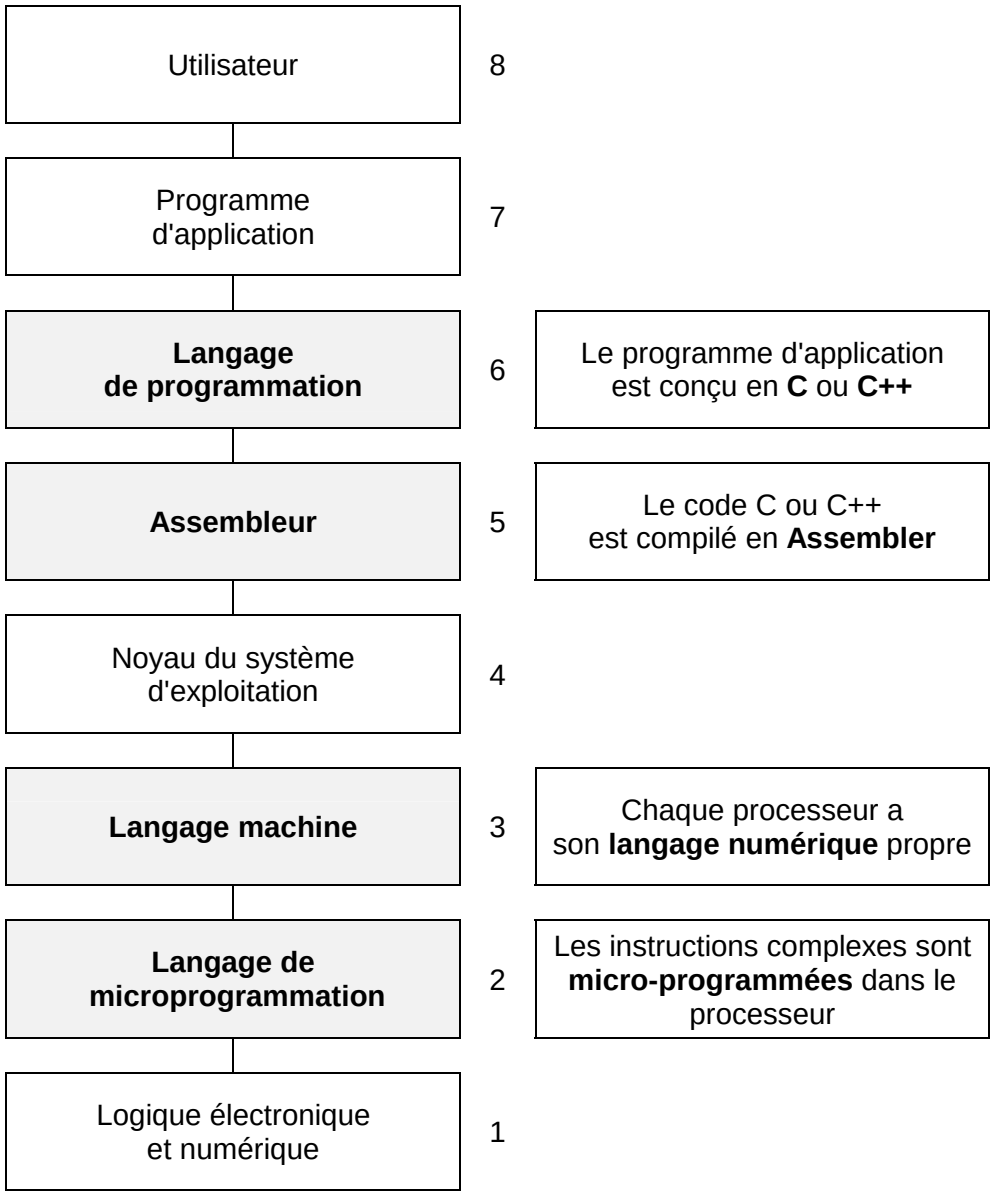


fig. 2.1 Modèle en couches d'abstraction

Intéressons-nous plus particulièrement aux couches de programmation proprement dites. C'est dans ces couches que l'on **écrit** les instructions que la machine doit exécuter ([1]).

1 Lors de l'écriture du programme, les instructions sont simplement sauvées sur le disque dur. Elles seront chargées en mémoire RAM lors de l'exécution.

2.2. Principe de compilation

2.2.1. Le langage de programmation

Partons d'un petit programme écrit en **langage C** (niveau 6) ([1]) :

```
main {
int a,b,c,d; /* définir les variables mémoire a,b,c,d      */
a = 5;       /* stocker 5 dans la variable mémoire a       */
b = 3;       /* stocker 3 dans la variable mémoire b       */
c = 8;       /* stocker 8 dans la variable mémoire c       */
d = a + b*c; /* multiplier le contenu des variables b et c */
}            /* y additionner le contenu de la variable a  */
             /* stocker le tout dans la variable d         */
```

Dans ce programme, a, b, c, d sont des ***variables symboliques***. Elles représentent des emplacements dans lesquels le programmeur stockera des "int" (integer), c'est-à-dire des nombres entiers signés codés sur 2 bytes.

Le programmeur n'a ***aucune*** idée des adresses RAM ***réelles*** qui seront effectivement attribuées à ces variables lors de l'exécution du programme. Ce choix sera fait par le système d'exploitation en fonction des disponibilités et il peut changer d'une exécution à l'autre. Tout ce que le programmeur sait, c'est que les adresses de a, b, c, d ne changeront pas ***pendant*** l'exécution du programme.

2.2.2. L'assembler

Malheureusement, le processeur est incapable de comprendre le langage C. Il faut donc le traduire vers un langage plus simple : l'ASSEMBLER. En **langage ASSEMBLER** (niveau 5), les instructions sont beaucoup plus élémentaires et le nombre d'instructions disponibles est beaucoup plus petit ([2]). On peut comparer ce langage avec les opérations à faire sur une calculatrice. Une instruction telle que

```
d = a + b*c;
```

devient quelque chose comme

```
entrer b
entrer c
multiplier ces deux valeurs
entrer a
additionner ce nombre au résultat précédent
stocker le résultat dans la mémoire d
```

Un programme peut être écrit directement en ASSEMBLER par un programmeur chevronné. Voici le même programme que ci-dessus :

1 Ceux qui pensent que ce programme est stupide ont absolument raison mais ils sont néanmoins priés de marquer un peu plus de respect pour l'auteur.

2 Il existe deux philosophies ASSEMBLER nommées RISC et CISC, l'une proposant peu d'instructions simples, l'autre proposant de nombreuses instructions complexes. Nous y reviendrons.

```
.MODEL small
.STACK 100h
.DATA
VAR_A dw          ; déclarer les variables
VAR_B dw
VAR_C dw
VAR_D dw
.CODE
mov ax, @data     ; adresse des variables
mov ds, ax
mov VAR_A, 5      ; mettre 5 dans VAR_A
mov VAR_B, 3      ; mettre 3 dans VAR_B
mov VAR_C, 8      ; mettre 8 dans VAR_C
mov ax, VAR_B     ; met le contenu de VAR_B dans le registre ax
imul VAR_C        ; multiplier le contenu du registre ax
                  ; avec le contenu de VAR_C
                  ; (le résultat est remis dans le registre ax)
add ax, VAR_A     ; additionner le contenu du registre ax
                  ; avec le contenu de VAR_A
                  ; (le résultat est remis dans le registre ax)
mov VAR_D, ax     ; mettre le contenu du registre ax dans VAR_D
mov ah, 4Ch
int 21h
END
```

En général, le programme est écrit dans un langage tel que C, C++, Pascal (niveau 6) et le travail de traduction et de décomposition est réalisé par un **compilateur** (*ang.: compiler*).

A cause de la décomposition des instructions, on note que le programme ASSEMBLER est beaucoup plus long que le programme C.

Il y a aussi une deuxième raison. Un processeur n'effectue une opération arithmétique que si les données ont été transférées dans *ses* mémoires internes, nommées **registres**. Ces registres sont des mémoires SRAM très rapides. Elles ont des noms symboliques standards définis par les constructeurs de processeurs. Dans l'exemple ci-dessus, ax, ds, ah sont les registres du processeur tandis que VAR_A, VAR_B, VAR_C, VAR_D seront des emplacements définis dans la mémoire RAM. Il faut donc chaque fois transférer l'information de la RAM vers le registre avant de faire une opération. C'est ce que fait une instruction telle que :

```
mov ax, VAR_B     ; met le contenu de VAR_B dans le registre ax
```

2.2.3. Intervention du système d'exploitation

Toutefois, dans le programme ASSEMBLER, VAR_A, VAR_B, VAR_C, VAR_D sont encore des variables symboliques. Avant d'obtenir un programme exécutable, Il faut modifier les instructions de manière à définir les emplacements de mémoire de manière relative. Il faut aussi ajouter une foule d'autres instructions qui préparent le terrain de notre petit programme.

Cette partie dépend du **système d'exploitation** (niveau 4). Dos, Windows et Linux, par exemple, ne gèrent pas le chargement des programmes en mémoire ni l'allocation de mémoire aux variables de la même manière. Cette étape très compliquée est heureusement automatisée grâce à un programme nommé **éditeur de liens** (*ang.: linker ou link editor*).

A ce stade, notre programme - exécutable sous Windows - passe de 20 à environ 5000 (cinq mille !) instructions ASSEMBLER. Voici le paragraphe qui nous intéresse, isolé de son contexte :

```
:
BYTE  4 DUP(0)
enter 0008, 00
mov word ptr [bp-02], 0005
mov word ptr [bp-04], 0003
mov word ptr [bp-06], 0008
mov ax, [bp-04]
imul word ptr [bp-06]
add ax, [bp-02]
mov [bp-08], ax
leave
ret
:
```

On note que VAR_A est devenu [bp-02]; VAR_B, [bp-04] et ainsi de suite (bp est le registre qui contient l'adresse du sommet de la "stack" c'est-à-dire la zone de RAM dans laquelle le programme peut stocker ses données).

2.2.4. Le langage machine

Même à ce stade, le processeur est incapable de comprendre le programme. Pourquoi ? Parce que le processeur ne traite que des 0 et des 1, c'est-à-dire du langage chiffré.

Chaque mot-clé ASSEMBLER correspond à un code d'opération numérique ou **OpCode** (les deux premiers digits hexa de chaque ligne ci-dessous); chaque variable est remplacée par son adresse relative numérique. Le couple compilateur-éditeur de liens fournit ainsi une version du programme en langage **MACHINE** (niveau 3) composée exclusivement de chiffres. C'est le programme exécutable (.EXE) :

```
:
00000000          ; BYTE  4 DUP(0)
C8080000          ; enter 0008, 00
C746FE0500        ; mov word ptr [bp-02], 0005
C746FC0300        ; mov word ptr [bp-04], 0003
C746FA0800        ; mov word ptr [bp-06], 0008
8B46FC            ; mov ax, [bp-04]
F76EFA            ; imul word ptr [bp-06]
0346FE            ; add ax, [bp-02]
8946F8            ; mov [bp-08], ax
C9                ; leave
C3                ; ret
:
```

Chaque famille de processeurs a son propre langage machine. C'est un langage natif, codé en "dur" dans chaque processeur, le seul qu'il comprenne; ce qui rend incompatibles des processeurs de familles différentes : un programme compilé et "linké" pour un processeur Intel devra être recompilé et "relinké" avec un autre outil pour fonctionner sur un processeur Motorola.

En fait, la version "chiffrée" du programme est illisible pour un programmeur. La suite de chiffres n'est pas structurée en ligne comme ci-dessus, mais les codes se suivent sans interruption :

```
:
803F0026C607FF06740726FF5F0207EBC726FF570207EBC0C3000000000C808
0000C746FE0500C746FC0300C746FA08008B46FCF76EFA0346FE8946F8C9C3
558BEC56571E078B7E048B76068B4E08D1E9FCF3A57301A48B46045F5E5DC3
:
```

Si, par malheur, on décale la lecture d'un seul byte, on arrive à un tout autre résultat ! Pour ne pas perdre le fil, le processeur utilisera un **pointeur d'instructions** (*ang.: instruction pointer*) qui, tel le doigt d'un lecteur débutant, suivra le déroulement du programme pas à pas.

Bien que le code machine soit illisible pour le commun des mortels, il existe des programmes, appelés **décompilateurs** (*ang.: decompilers*), qui permettent de repasser assez aisément des codes chiffrés au langage ASSEMBLER ([1]). Par contre, le retour au code source initial en langage C est pratiquement impossible.

2.2.5. La micro-programmation

Une instruction telle que ci-dessous est comprise par le processeur mais elle n'est pas directement exécutable :

```
F76EFA          ; imul word ptr [bp-06]
```

En effet, cette instruction effectue la multiplication de deux nombres entiers (imul) dont l'un est dans le registre A (par définition de imul) et l'autre en mémoire RAM où il occupe deux bytes (word ptr) et son adresse doit faire l'objet d'un calcul ([bp-06]).

L'instruction doit donc être décomposée c'est-à-dire remplacée par une suite d'actions plus élémentaires :

```
Transférer [bp] dans un registre de calcul
Soustraire 6
Transférer les deux bytes de cette adresse vers un registre
Effectuer la multiplication
```

Cette décomposition-remplacement se fait au sein du processeur. A chaque instruction complexe correspond une suite d'instructions élémentaires ([2]) appelées micro-opérations ou µops (Intel). La bibliothèque de micro-programmes est codée "en dur" dans l'une des zones ROM du processeur (niveau 2). Un programmeur même expérimenté n'y a pas accès. C'est notamment en améliorant l'efficacité de leur micro-code que les concepteurs améliorent les performances de leurs processeurs.

2.3. Philosophies CISC et RISC

2.3.1. Types d'instructions

Les instructions se rassemblent en **six groupes** que l'on retrouve sur toutes les machines :

- Les **transferts de données** (MOV, POP, PUSH,...)

  Elles provoquent le transfert de 1, 2, 4 ou 8 bytes de la mémoire RAM vers un registre interne, d'un registre à un autre, d'un registre vers la RAM, etc.
- Les **opérations arithmétiques** (ADD, SUB, IMUL, IDIV, ...)

  Tous les processeurs proposent les quatre opérations de base sur des opérandes de type entier signé ou non, entier court ou long. L'implémentation des opérations sur les réels, au sein d'un co-processeur arithmétique *intégré*, est relativement récente (486DX).

1 C'est un tel programme, W32DASM, qui a été utilisé pour illustrer les exemples de ce chapitre.

2 Plus exactement une suite de codes d'instruction

- Les **opérations logiques** (AND, OR, NOT, XOR, ...)

  Ces opérations comparent deux variables bit à bit.
- Les **contrôles de séquence** (JMP, CALL, RET, JNZ, JLE, ...)

  Le contrôles de séquence permettent de poursuivre le traitement dans une autre partie du programme. Ils peuvent être impératifs (JMP, CALL, RET) comme dans le cas des appels et retours de routines ([1]) ou conditionnels (JNZ, JLE) comme dans le cas des tests et des boucles ([2]).
- Les **entrées/sorties** (IN, OUT, ...)

  Les entrées/sorties directes se font au travers des ports qui conduisent clavier, à l'écran, au disque. Pour simplifier la tâche, le BIOS offre des services plus spécialisés qui sont invoqués au moyen d'interruptions logicielles (*ang.: interrupts*).
- Les **commandes** et **manipulations diverses** (NOP, WAIT, INT,...)

  Ces commandes diverses gèrent l'état du processeur (WAIT), provoquent le rafraîchissement des caches (WBINVD), convertissent les données d'un format vers un autre, etc.

A ce stade, deux conceptions s'affrontent :

- l'architecture CISC (*ang.: Complex Instruction Set Computer*),
- l'architecture RISC (*ang.: Reduced Instruction Set Computer*).

2.3.2. L'architecture CISC

L'architecture CISC ([3]) est utilisée par tous les processeurs de type x86 (Intel, AMD, Cyrix, ...).

L'objectif est de rendre le processeur capable de décoder et d'exécuter des instructions très complexes. Celles-ci sont soit micro-programmées dans une bibliothèque incluse dans le processeur, soit directement câblées (gravées) sur la puce de silicium de manière à accélérer leur traitement.

Puisque le processeur comprend des instructions complexes, le travail du programmeur (et du compilateur) est simplifié, ce qui est un avantage certain.

Par contre, du fait que les fonctions complexes sont implémentées sur le silicium, les coûts de conception et de production du processeur sont particulièrement élevés.

Une instruction CISC comprend de 1 à 9 zones d'informations et sa taille varie de 1 à 16 bytes en fonction de leur complexité (cf exemple 2.2.4) :

Instruction prefix	Address size prefix	Operand size prefix	Segment override	**OpCode**	Mode r/M	SIB	Displacement	Immediate
0 or 1	0 or 1	0 or 1	0 or 1	**1 or 2**	0 or 1	0 or 1	0, 1, 2 or 4	0, 1, 2 or 4
Number of bytes								

L'instruction commence *éventuellement* par quelques bytes (0 à 4) relatifs à l'organisation du segment de programme auquel l'instruction se réfère et à la taille des variables.

Vient le code d'opération (OpCode) qui, suivant les cas occupe 1 ou 2 bytes.

1 Un programme est construit en une suite de paragraphe qui s'enchaînent mais tous les paragraphes ne se trouvent pas toujours dans la même zone de mémoire.

2 Certaines fonctions sont activées SI l'utilisateur clique ou TANT QUE les données ne sont pas épuisées...

3 CISC : ordinateur à jeu d'instructions complexes; RISC : ordinateur à jeu d'instructions limité

Suivent ensuite d'autres codes qui précisent si l'instruction travaille sur des informations stockées dans les registres ou dans la mémoire RAM.

Viennent ensuite les adresses ou les valeurs éventuelles, lesquelles peuvent occuper chacune 1, 2 ou 4 bytes.

Dans ce système, l'instruction la plus courte compte 1 byte et la plus longue est une phrase de 16 bytes. Selon les cas, le code d'instruction (OpCode) peut aussi bien se trouver dans le premier byte que dans le cinquième (ou n'importe où entre les deux).

Il est clair que le temps de lecture, de décodage et d'exécution d'une instruction CISC variera largement en fonction de l'instruction à traiter ([1]). La gestion d'un tel processeur particulièrement complexe. Néanmoins, rappelons que la plupart des PC sont équipés de processeur de ce type.

2.3.3. L'architecture RISC

Contrairement au processeur CISC, le processeur qui utilise la technologie RISC ne dispose que de très peu d'instructions codées en dur. Par exemple, aucun circuit ne permet le calcul d'une fonction trigonométrique (sinus, tangente) et souvent même la multiplication et la division de deux nombres réels doivent être émulées ([2]).

L'avantage d'une telle architecture est bien évidemment un coût de conception et de fabrication extrêmement réduits. De plus, les instructions simples ont toutes la même taille (typiquement 2 bytes). Elles peuvent être exécutées en un cycle d'horloge. De plus, comme nous le verrons plus loin, de tels processeurs sont facilement capables de traiter simultanément plusieurs instructions en parallèle. L'exécution des programmes RISC est donc plus rapide que ceux basés sur une architecture CISC.

Par contre, puisque le processeur ne comprend que des instructions simples, la rédaction d'un programme doit être beaucoup plus détaillée. La charge est donc reportée sur le programmeur, qui doit porter plus d'attention à l'organisation de son code, et sur le compilateur, qui doit être plus puissant. Le programmes RISC sont plus longs que les programmes CISC et ils nécessitent plus de mémoire pour stocker les résultats intermédiaires.

2.3.4. Finalement RISC ou CISC ?

On constate aujourd'hui une sorte de convergence des deux philosophies :

- l'architecture CISC, grâce à une micro-programmation interne, tend à remplacer les instructions complexes qui entrent dans le processeur par une série d'instructions plus simples, lesquelles seront exécutées par le "cœur" du processeur,
- l'architecture RISC, intègre de plus en plus souvent des circuits spécialisés pour réaliser les fonctions mathématiques évoluées.

1 On peut comparer la complexité du décodage des instructions à l'analyse grammaticale de phrases telles que : "Alain mange" ou "Ce matin, Alain demanda à Brigitte de porter un déjeuner à Christian".

2 L'émulation consiste à remplacer une instruction prévue pour un langage ou une machine en une suite d'instructions compréhensibles par un autre langage ou une autre machine. Emuler la multiplication, c'est décrire la suite d'opérations qu'il faudrait faire sur une calculatrice qui ne disposerait que des fonctions d'addition, de soustraction et de multiplication par 2.

3. Un processeur élémentaire

3.1. Aspect physique : la puce

La puce (*ang.: chip*) ou circuit intégré (*ang.: integrated circuit*) est une pièce de silicium de quelques millimètres carrés sur laquelle sont gravés des centaines de milliers de transistor. Ces transistors forment des portes logiques et sont assemblés en circuits.

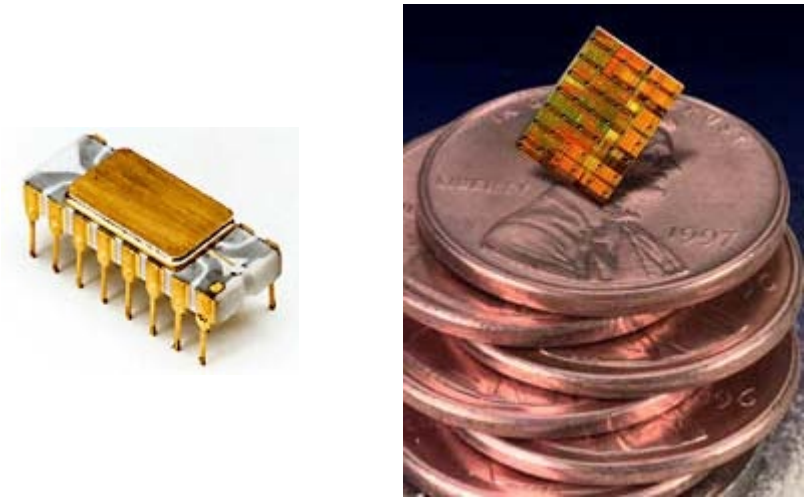


fig. 3.1 Processeur 4004 (g) et Puce CMOS 7S "Copper chip" sur une pile de petite monnaie (dr)

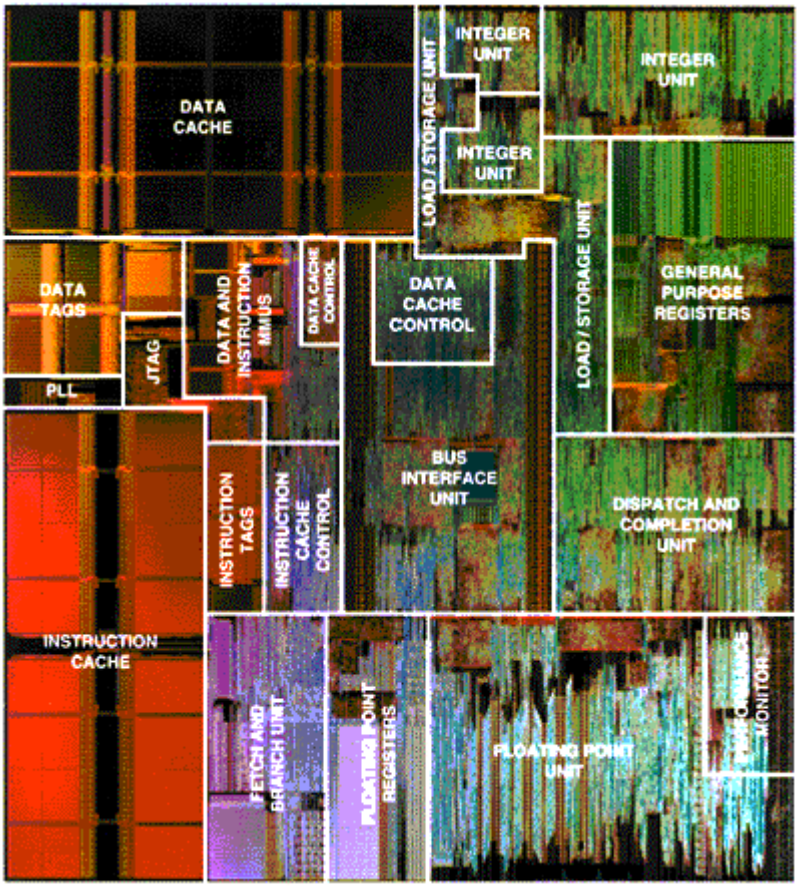


fig. 3.2 Micrographie de la puce PowerPC604e de Motorola (source: Morris)

La fig. 3.2 montre une micrographie de la surface d'un chip de silicium d'une taille réelle de quelques millimètres carrés. La densité et l'organisation des circuits sont différentes selon la tâche allouée à chaque zone. Dans une zone de mémoire, tous les bytes ont la même fonction, ce qui se traduit par une coloration régulière (data cache, instruction cache, instruction tags,...). Par contre, dans une zone de calcul,

tous les micro-circuits sont différents, ce qui donne une coloration plus irrégulière (floating point unit, integer unit, bus interface unit...)

3.2. Principe d'organisation

L'architecture générale du processeur ressemble à l'architecture de l'ensemble de l'ordinateur, à un facteur d'échelle près.

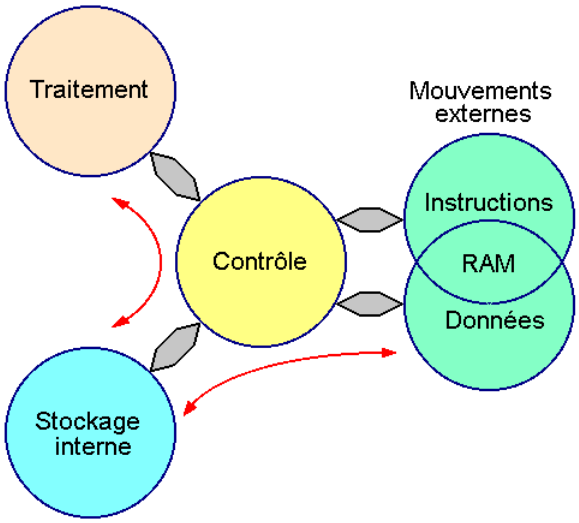


fig. 3.3 Schéma de principe d'un processeur

On y retrouve

- une unité de traitement arithmétique et logique (ALU),
- une unité de stockage interne (registres)
  et des unités de stockage temporaire (tampons),
- des unités externes de stockage des instructions et des données (RAM)
- des transferts qui se font via des bus locaux (adresse, données),
- une unité de contrôle (bus de contrôle) qui assure la supervision.

En fait, un processeur traite que les données et les instructions qui sont dans ses mémoires internes, nommées registres. Il doit donc disposer d'une série d'assistants qui assurent son alimentation régulière et qui évacuent les résultats.

Les registres sont de petites mémoires de type SRAM, très rapides. Au fil de l'évolution des processeurs le nombre de registres est passé de une dizaine à plusieurs centaines et leur capacité est passé de 1 à 8 bytes.

Les plus importants sont, en suivant l'ordre d'exécution d'une instruction :

- le compteur ordinal, qui contient l'adresse de la prochaine instruction à traiter
- le registre tampon, qui stocke temporairement une donnée ou une instruction provenant de la mémoire
- le registre d'instruction, qui contient l'instruction en cours de traitement
- le registre accumulateur, qui stocke les résultats des opérations arithmétiques et logiques,
- le registre d'état, qui stocker les indicateurs (*ang.: flags*)

Nous y reviendrons lors de l'explication du fonctionnement.

3.3. Principe de fonctionnement

Le premier microprocesseur (Intel 4004) a été inventé en 1971. Depuis, la puissance des processeurs a été considérablement augmentée mais le principe de base reste toujours valable.

Le processeur comprend une **horloge interne**. Celle-ci émet des impulsions électroniques régulières qui rythment toutes les actions de l'unité centrale. Entre deux tops d'horloge le processeur effectue une action élémentaire. Plus la fréquence de l'horloge est élevée, plus le processeur peut effectuer d'actions élémentaires en une seconde. Le standard actuel ([1]) dépasse largement 1 GHz, ce qui signifie que le processeur effectue plus d'un milliard (10^9) d'actions par seconde et que chaque action dure moins d'une nanoseconde.

Un cycle machine comprend quatre phases. Chaque phase est accomplie successivement par le processeur.

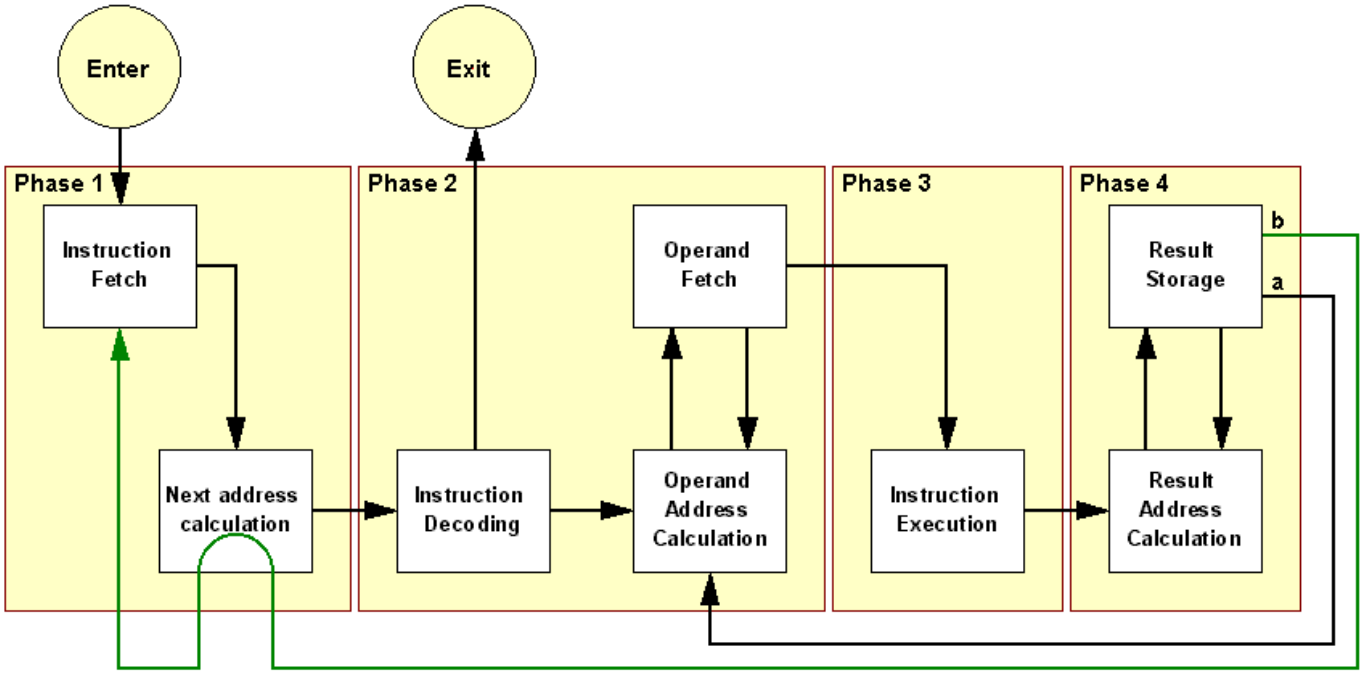


fig. 3.4 Schéma bloc d'exécution d'une instruction
(rang sup : accès mémoire et registres, rang inf : tâches processeur)

Passons-les rapidement en revue :

- **Fetch**

  Pendant la première phase, le processeur va chercher (*ang.: fetch*) l'instruction qui se trouve en mémoire RAM. L'adresse du début de cette instruction est contenue dans le registre **compteur ordinal** (CO) du processeur.

  Dès que les bytes qui composent l'instruction sont arrivés dans le tampon d'entrée, le compteur ordinal est mis à jour de manière à pointer sur l'instruction suivante.

- **Decode**

  L'**unité de contrôle** (UC) prend l'instruction en charge. Un décodeur d'instruction (*ang.: instruction decoder*) isole le ou les bytes qui contiennent le code opération (*OpCode*). Il isole et calcule les diverses adresses RAM des variables impliquées dans l'instruction (*ang.: operand address calculation*) et va chercher les valeurs correspondantes (*ang.: operand fetch*).

1 En 2004.

- **Execute**

  L'**Unité arithmétique et logique** (*ang.: **A**rithmetic **L**ogic **U**nit*) exécute (*ang.: execute*) l'opération arithmétique ou logique demandée en fonction de l'OpCode isolé à l'étape précédente. Elle renvoie le résultat dans le registre accumulateur.

- **Store** (ou Write Back)

  L'unité d'interface décode et assemble l'adresse à laquelle le résultat doit être stocké (*ang.: result address calculation*). Elle lit le résultat du registre accumulateur et le transfère vers la mémoire externe ou vers un autre registre interne.

Dans un processeur CISC, si l'instruction traite une chaîne de caractères ou un vecteur (dont les emplacements de mémoire sont contigus), le processus va simplement chercher l'adresse de l'élément suivant (retour "a"). Par contre, s'il s'agit d'une opération scalaire, le processus reprend au début, avec une lecture de l'adresse de l'instruction suivante (retour "b"). Dans un cas comme dans l'autre; le processeur vérifie d'abord qu'aucune interruption (*ang.: interrupt*) n'a été lancée par le programme ou par un périphérique.

Si l'instruction décodée à la phase 2 est "Fin de programme", le processus s'arrête. En fait, comme nous l'avons vu par ailleurs, les programmes s'empilent à la manière de poupées russes. Quand un programme se termine, le processeur reprend l'exécution du programme de base :

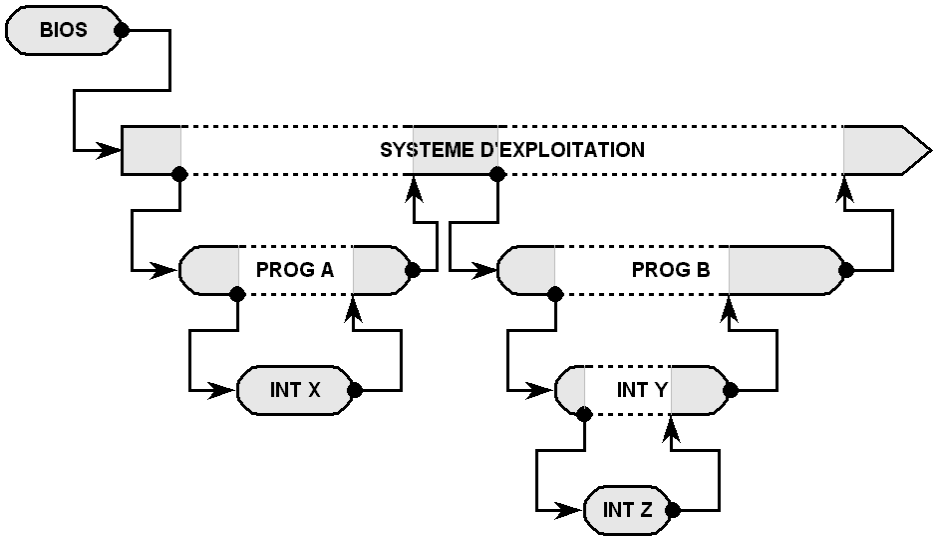


fig. 3.5 Empilage des programmes en cours d'exécution

3.4. Exécution dans le processeur

Reprenons le processus d'exécution mais cette fois en mettant en évidence l'organisation interne du processeur.

Pour simplifier l'approche, nous partons d'un processeur très rudimentaire (p.ex.: Intel 4004) pour lequel les instructions ont toujours une longueur de 2 bytes : un OpCode de 1 byte suivi d'une opérande de 1 byte ([1]).

1 Si une instruction n'utilise pas d'opérande (p.ex. pop) alors le deuxième byte contient 00h.

3.4.1. Phase 1 : Fetch

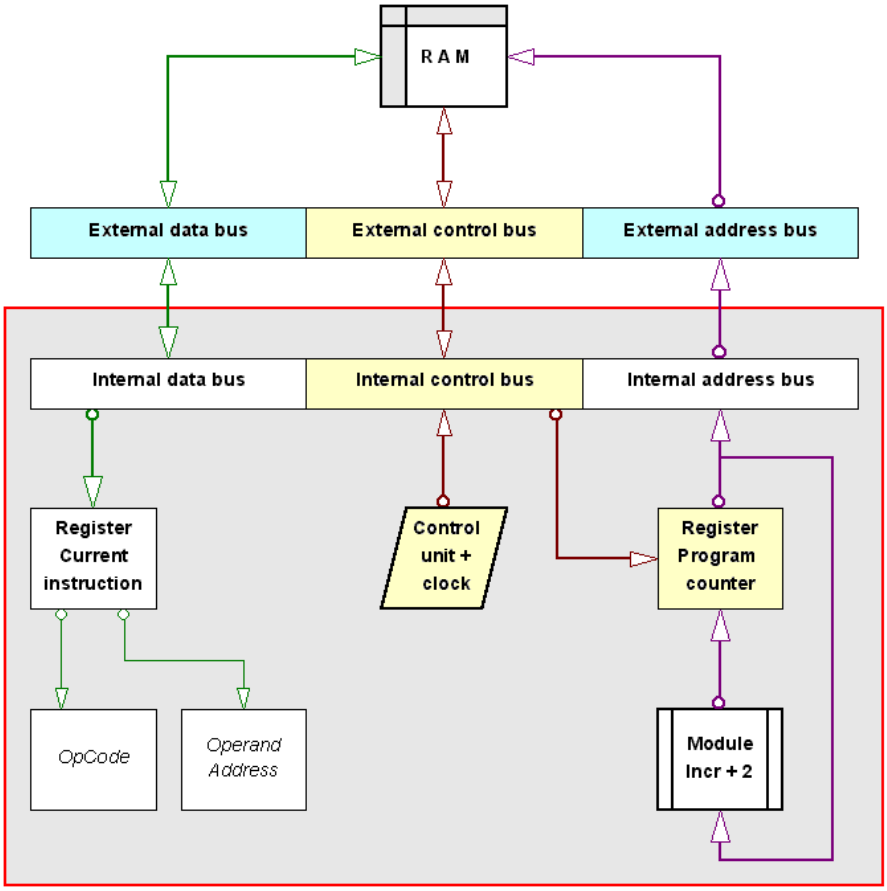


fig. 3.6 Activation de l'envoi de l'adresse de l'instruction

Ainsi qu'il a été dit, le processeur dispose de sa propre unité de contrôle (*ang.: control unit*), d'un bus interne d'adresses, d'un bus interne de données et d'un bus interne de contrôle. Ces bus internes sont en communication avec les bus externes correspondants. Le processeur contient une horloge dont la fréquence est un multiple entier ou demi-entier de celle du bus externe. Cependant, n'oublions pas que dans les premiers ordinateurs, le processeur et tous les bus fonctionnaient à la même fréquence.

Les instructions d'un programme en langage machine (numérique) sont généralement stockées l'extérieur du CPU, dans une mémoire RAM (ou ROM).

L'élément important est le **registre du compteur d'instruction** ou compteur ordinal (*ang.: Program counter register*). Il contient l'adresse RAM qui contient l'instruction qui doit être exécutée au prochain cycle ([1]).

Lors d'un "tick" d'horloge, le contrôleur ouvre les portes Tri-State et provoque le transfert de cette adresse vers le bus interne d'adresse. En même temps, il envoie les signaux de contrôle adéquats vers la RAM (Read).

1 On est en droit de se demander comment l'adresse de la toute première instruction est arrivée dans ce registre. Simple : lors d'un démarrage ou d'un reset de la machine, tous les registres sont effacés. La réapparition d'une tension aux bornes du processeur provoque l'inscription de l'adresse FFFF0h dans le conpteur ordinal. Cette adresse n'est autre que l'adresse de la première instruction de "BOOT" stockée dans le BIOS. Le reste suit...

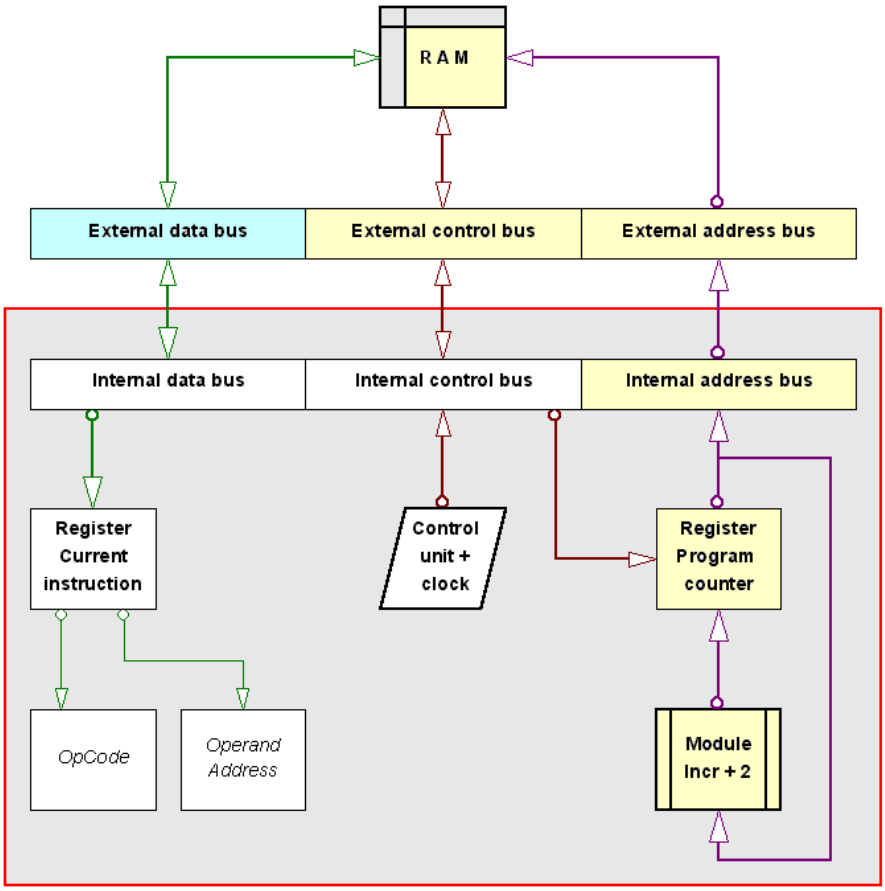


fig. 3.7 Transfert de l'adresse et incrémentation

L'adresse parcourt le bus externe d'adresse et arrive dans la RAM. La valeur de l'adresse est aussi redirigée vers le module incrémenteur qui déplace le pointeur de deux unités. Le compteur ordinal pointe ainsi vers l'instruction suivante.

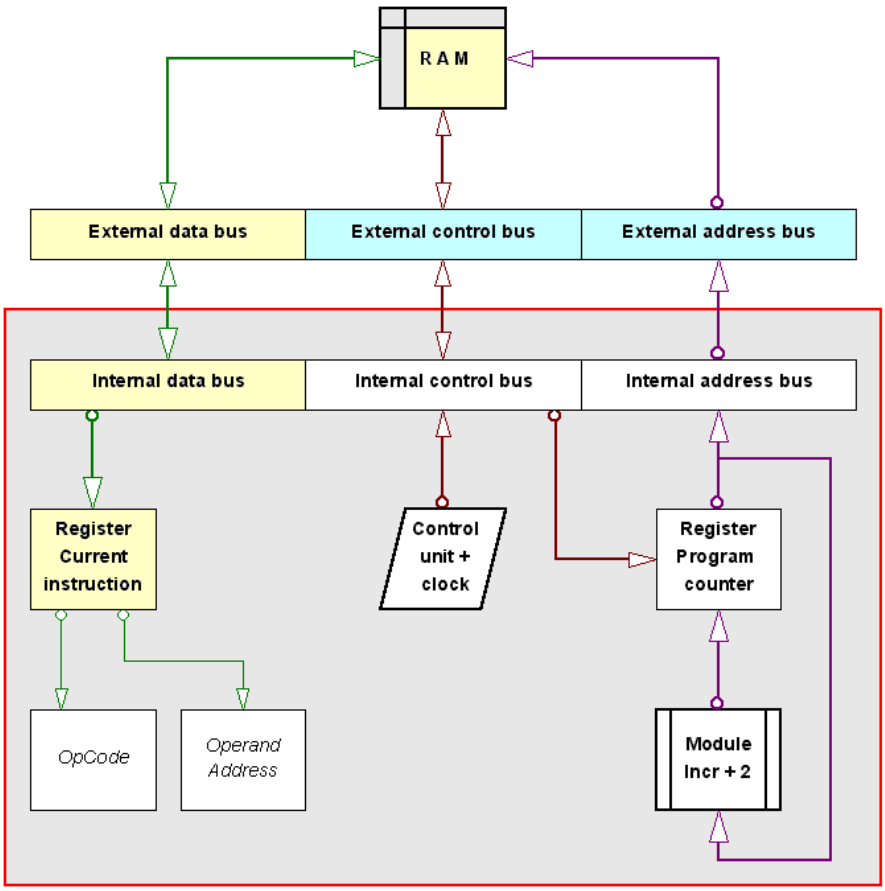


fig. 3.8 Réception de l'instruction sur le bus de données

La RAM renvoie sur le bus de données les valeurs qui étaient stockées aux emplacements dont on a fourni l'adresse ([1]). Ces valeurs, qui constituent en fait une instruction codée. Le processeur qui a fait la demande le sait; il fait donc transiter ces bytes par le bus interne de données les stocke dans le **registre d'instruction courante** (*ang.: current instruction register*).

3.4.2. Phase 2 : Decode

L'**unité de contrôle** (UC) prend l'instruction en charge. Le **décodeur** extraît l'OpCode d'une part et l'opérande éventuelle d'autre part.

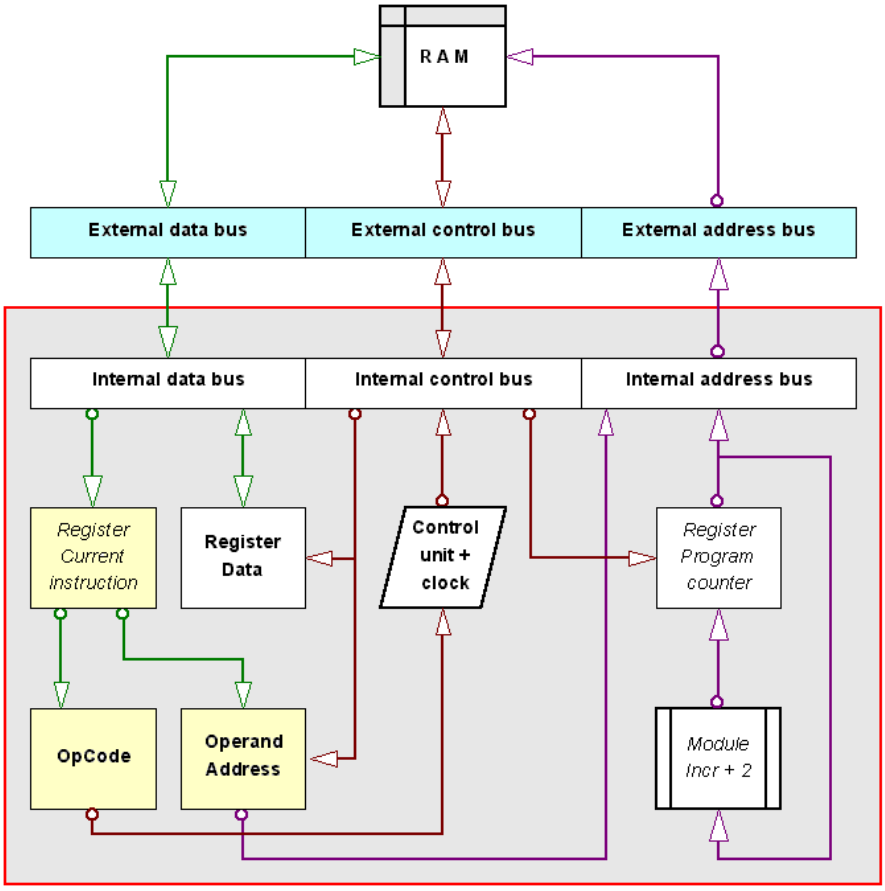


fig. 3.9 Décomposition de l'instruction en OpCode et opérande

L'OpCode est stocké en attendant que toutes les opérandes (une seule dans le cas présent) soient résolues. La recherche de la valeur de l'opérande peut se faire selon cinq modes :

- une valeur immédiate, c'est-à-dire une valeur numérique,
- l'adresse du registre qui contient la valeur
- l'adresse en mémoire RAM qui contient la valeur,
- une adresse indirecte à calculer à partir d'une adresse stockée en registre,
- une adresse indirecte à calculer à partir d'une adresse stockée en mémoire.

Le choix du mode est fixé par la valeur de l'OpCode ([2]). Par définition, notre processeur élémentaire est incapable de gérer les deux derniers cas qui impliquent un calcul intermédiaire.

---

1 Avec une RAM primitive qui ne renvoie qu'un seul byte à la fois, la lecture d'une instruction élémentaire exige deux envois d'adresse. Il suffit de modifier un peu le circuit incrémenteur. On a vu par ailleurs que les RAM plus récentes renvoient systématiquement une rafale de 4 bytes, ce qui simplifie le travail.

2 Par exemple, si l'OpCode est codé sur 8 bits, on peut décider que les 5 bits de poids fort définissent ./.

Le cas illustré ci-après est celui de la recherche en mémoire RAM. Grâce à l'analyse de l'OpCode, l'unité de contrôle reconnaît un accès à une opérande dont la valeur est stockée en mémoire RAM.

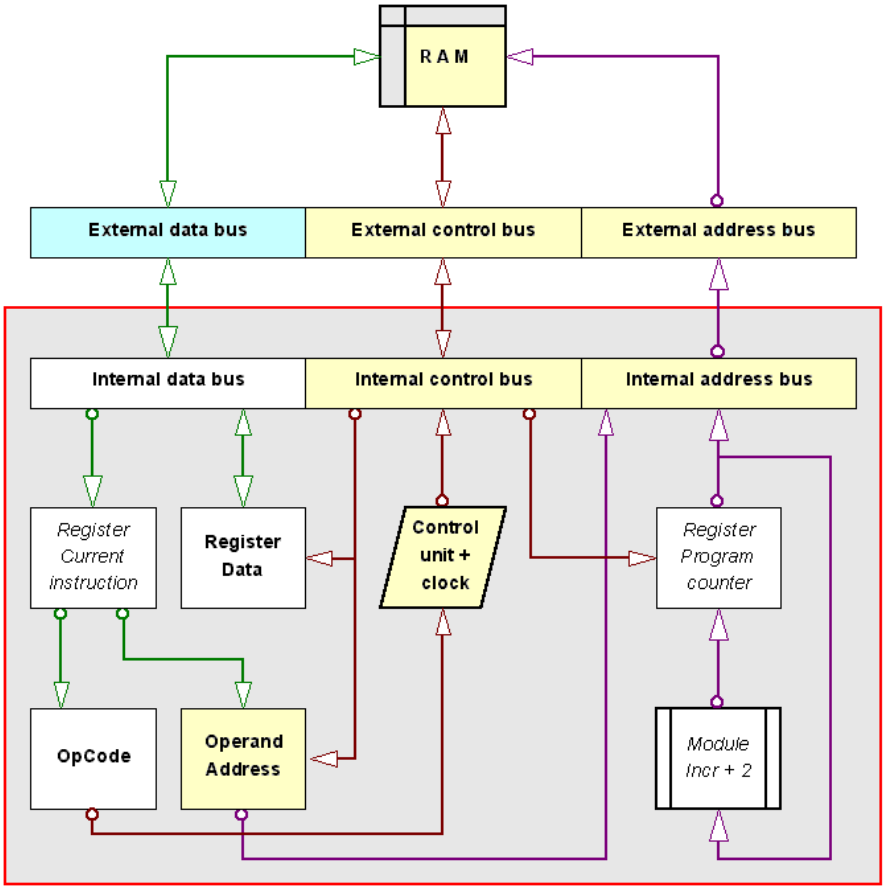


fig. 3.10 Transfert de l'adresse de l'opérande sur le bus d'adresse

L'adresse de l'opérande est transmise sur le bus d'adresse tandis que l'unité de contrôle envoie les signaux adéquats à la RAM.

A nouveau, la RAM renvoie les informations sur le bus de données externe. Cette fois, le processeur sait qu'il a demandé une donnée; il fait transiter les bytes renvoyés par la RAM par bus de données internes et les dirige vers le registre des données (*ang.: Data register*).

l'opération (32 opérations possibles) et que les 3 bits de poids faible définissent le mode d'accès à l'opérande (8 modes possibles).

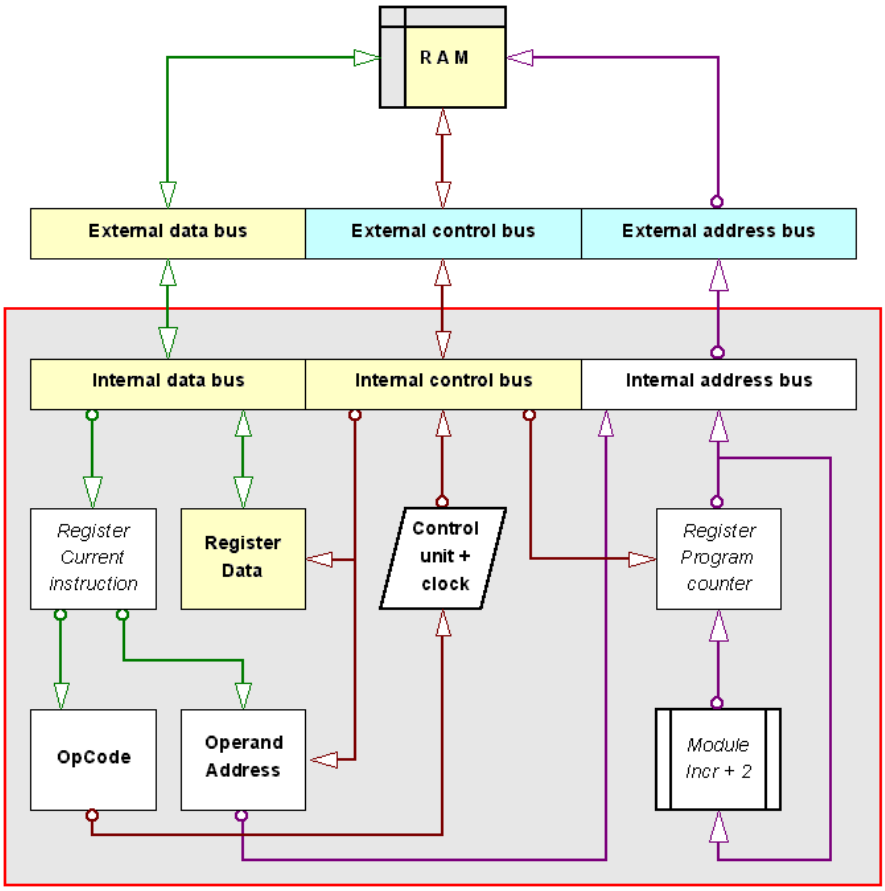


fig. 3.11 Réception des valeurs dans le registre des données

Dès que les données sont arrivées, l'unité de contrôle génère les signaux nécessaires sur le bus de contrôle afin de provoquer l'exécution de l'instruction :

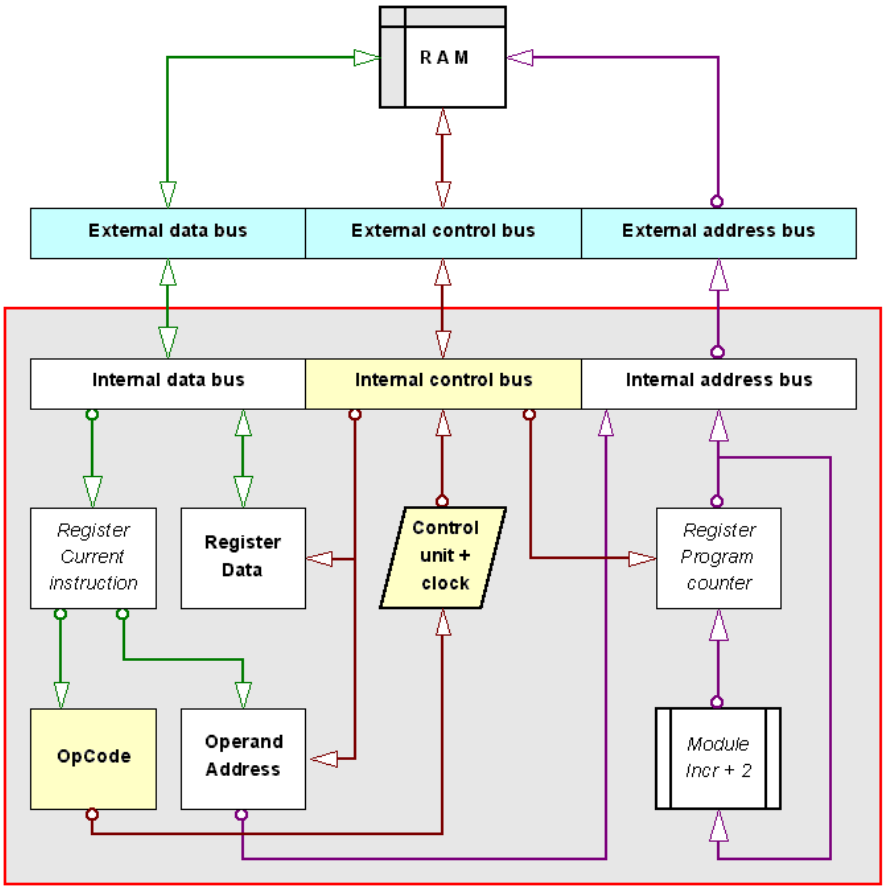


fig. 3.12 Génération des signaux d'exécution

3.4.3. Phase 3 : Execute

L'Unité arithmétique et logique (*ang.: **A**rithmetic **L**ogic **U**nit*) exécute l'opération arithmétique ou logique demandée par l'unité de contrôle (sur base de l'OpCode) et renvoie le résultat.

3.4.3.1 Structure d'une ALU

Une ALU élémentaire qui devrait réaliser une addition aurait simplement besoin d'un circuit additionneur ([1]) alimenté par deux mémoires ou registres. Le résultat est renvoyé sur le bus de données interne.

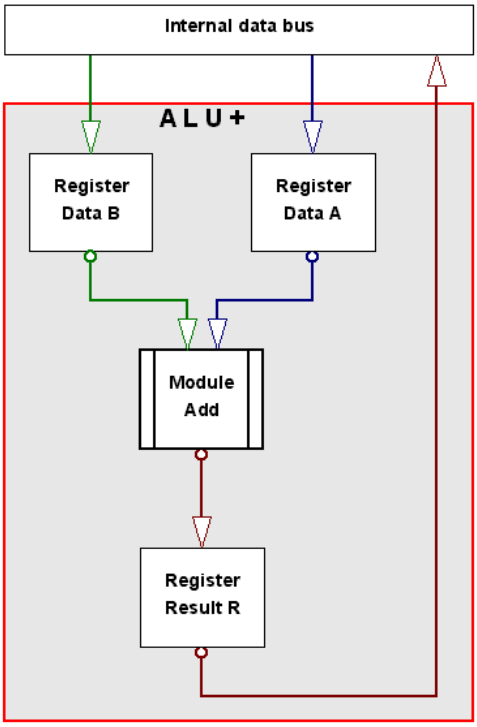


fig. 3.13 Un additionneur simple

3.4.3.2 Les flags

Il est utile cependant de récupérer des informations supplémentaires quant au déroulement de l'addition : y a-t-il eu overflow, faut-il traiter un dernier report, le résultat est-il nul, y a-t-il eu une erreur, etc. La réponse à chacune de ces questions ne nécessite qu'un seul bit nommé **flag**. L'ensemble des flags forme un ou deux bytes qui sont stockés dans un registre spécial (ang.: flag register).

1 Le circuit additionneur est expliqué au chapitre logique booléenne

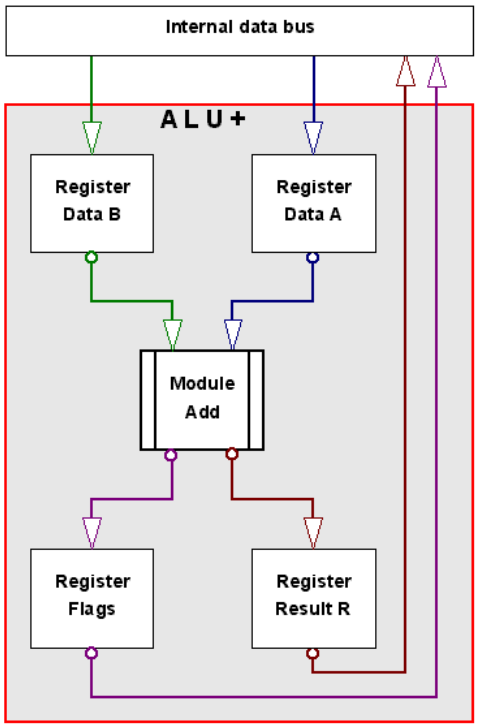


fig. 3.14 Un additionneur avec renvoi d'information supplémentaire (flags)

3.4.3.3 L'accumulateur

En général, dans un programme, les opérations arithmétiques se succèdent. Très souvent, un résultat intermédiaire est immédiatement réutilisé dans la suite du calcul. Il n'est pas intéressant de renvoyer ce résultat intermédiaire en mémoire RAM si on doit immédiatement le rappeler. C'est pourquoi les processeurs utilisent un registre spécial, destiné aux stockagex intermédiaires, nommé **accumulateur** (*ang.: accumulator*).

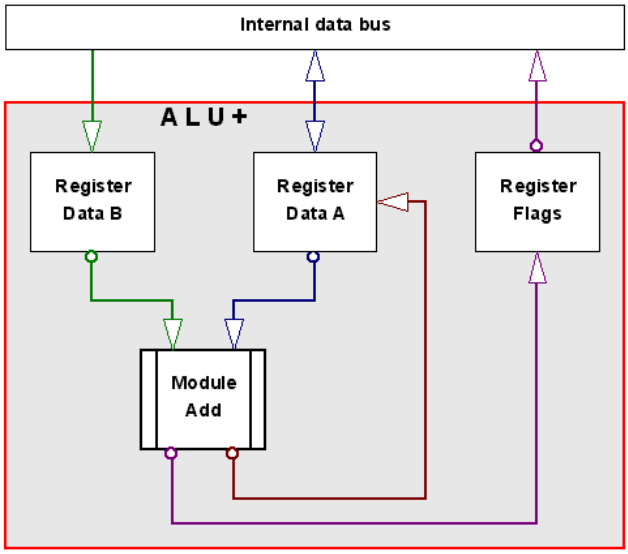


fig. 3.15 Le résultat est renvoyé dans le registre initial

Avant l'opération arithmétique, l'accumulateur (Data A) contient la valeur d'une des opérandes (l'autre est dans un autre registre Data B). Après l'opération arithmétique, l'accumulateur contient le résultat. Ceci permet d'enchaîner rapidement les opérations ([1]).

En fin de calcul, on renverra le résultat de l'accumulateur vers le bus de données.

1 L'accumulateur peut être comparé à l'affichage d'une calculatrice. Quand on enchaîne les additions, chaque donnée introduite fait place au total dès que l'on appuie sur la touche "+" ou "=".

3.4.3.4 Combiner les modules

Evidemment, une ALU ne se contente pas de faire des additions. Les plus simples permettent au moins de réaliser aussi des soustractions, des opérations logiques bit à bit AND, OR, XOR, etc. Une première idée consiste à placer plusieurs modules en parallèle.

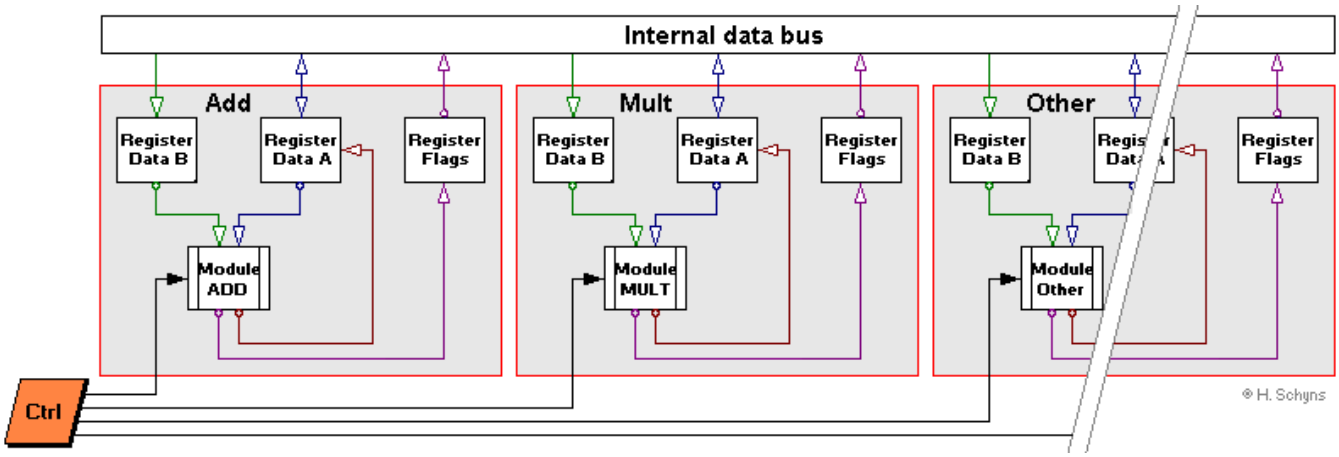


fig. 3.16 Sélection du module adéquat en fonction de l'OpCode (1ere version)

Une meilleure solution consiste à alimenter tous les modules arithmétiques à partir des deux mêmes registres :

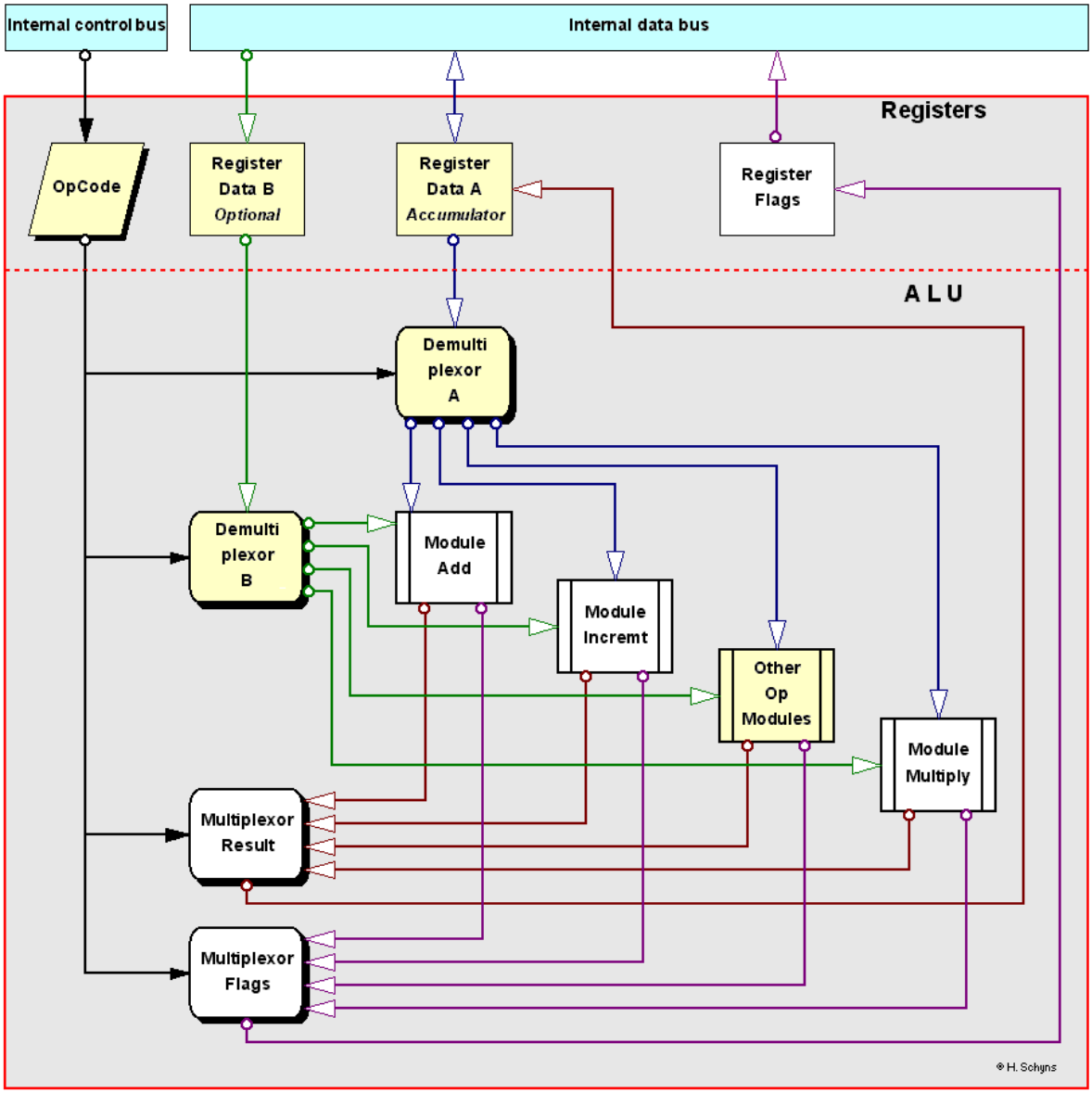


fig. *3.17 Démultiplexage des opérateurs en fonction de l'OpCode*

En effet, lors des décodages successifs, les "bonnes" données ont déjà été placées dans les registres A et B. Lors du déclenchement de l'exécution, L'OpCode (qui a été interprété par le module de contrôle) agit comme une clé de démultiplexage qui envoie simultanément les valeurs vers le module adéquat.

L'OpCode agit également en tant que clé de multiplexage pour récupérer le résultat ainsi que les flags en provenance du module choisi. Le résultat est redirigé vers le registre d'accumulation (*ang.: accumulator*) où il est prêt à être utilisé, éventuellement dans un autre module.

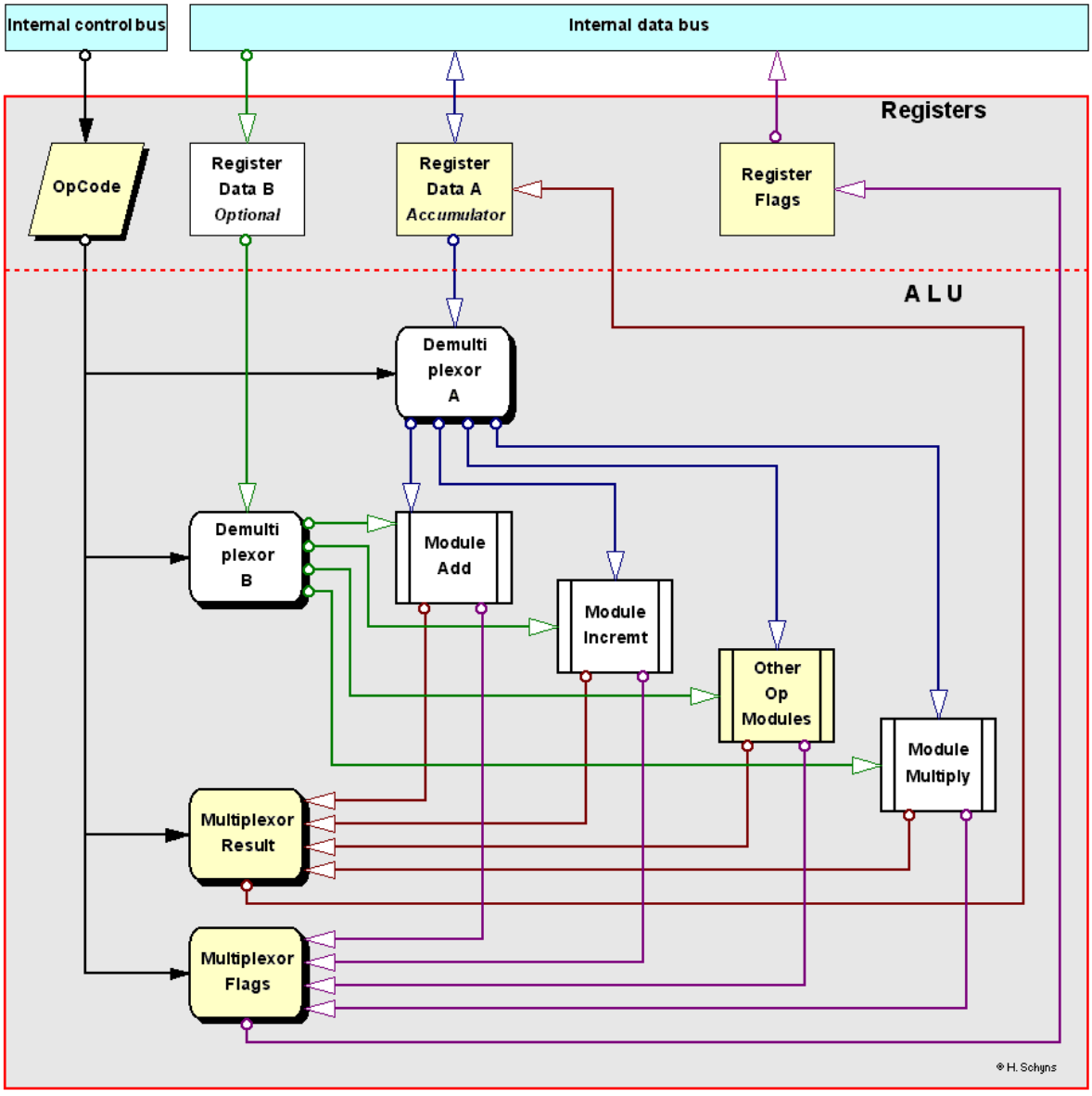


fig. 3.18 Multiplexage du résultat et des flags en fonction de l'OpCode

Notons que dans le cas de notre processeur élémentaire, une nouvelle opération arithmétique fait obligatoirement l'objet d'une nouvelle instruction.

3.4.4. Phase 4 : Store

A la fin de l'opération arithmétique, le résultat se trouve dans le registre accumulateur ([1]). Il faut le renvoyer vers la mémoire. Ce renvoi se fait :

1 Dans le cas de notre processeur élémentaire ou d'un processeur RISC, il faudrait une nouvelle instruction ./.

- en renvoyant l'adresse de l'opérande sur le bus d'adresse interne et, de là, vers le bus d'adresse externe et la mémoire RAM,
- en renvoyant simultanément le contenu du registre accumulateur vers le bus interne de données (transfert inverse) et, de là, vers le bus de données externe et la mémoire RAM,
- en envoyant simultanément sur le bus de contrôle les signaux d'écriture en RAM (WRITE ENABLE).

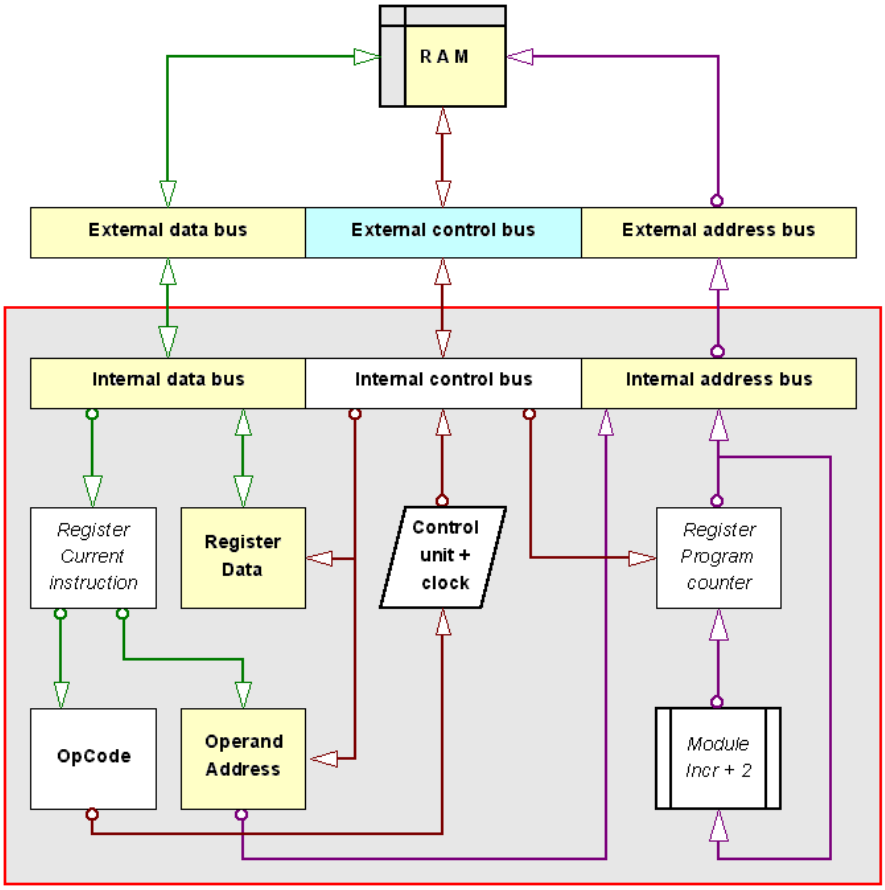


fig. 3.19 Renvoi du résultat sur le data bus et de l'adresse sur l'address bus

3.4.5. Conclusion

Chaque instruction exige un ou deux "tours" en RAM :

- le premier pour aller chercher l'instruction,
- le deuxième pour aller chercher l'opérande éventuelle.

Dans le cas d'une instruction de stockage on a également deux tours, mais le

- le premier pour aller chercher l'instruction,
- le deuxième pour renvoyer le résultat vers la RAM. Dans ce cas, le bus de données est parcouru en sens inverse.

Le processeur accomplit successivement chacune des quatre phases. Une nouvelle instruction ne peut pas être entamée tant que la précédente n'est pas totalement terminée.

pour transférer ce résultat vers une adresse de mémoire RAM.

4. Le processeur Intel 8088 / 8086

4.1. Position du problème

Les performances du premier processeur analysé au chapitre précédent sont très limitées comparées aux performance actuelles.

En particulier, dans ce premier processeur, les bus d'adresse et de données tant internes et externes n'ont que 8 bits de large. Il en va de même pour le registre d'accumulation. Ceci limite singulièrement la mémoire accessible et complique le traitement des grands nombres entiers et des nombres réels.

D'autre part, le CPU ne dispose que d'un registre d'opérande et d'un registre d'accumulation. Avec plusieurs registres, il serait possible de réaliser en une étape des opérations plus complexes.

Enfin, le processeur accomplit successivement chacune des quatre phases. Une nouvelle instruction ne peut pas être entamée tant que la précédente n'était pas totalement terminée.

Toutefois, ce processeur élémentaire a mis en évidence deux sous-structures :

- une structure chargée d'aller chercher ou de stocker des informations à l'extérieur du processeur,
- une structure chargée des opérations arithmétiques sur ces informations.

Cette organisation en deux blocs a été exploitée dans la conception de la génération suivante des processeurs tels que Intel 8088 :

- Le **Bus Interface Unit** (BUI) qui interface les bus d'adresse, de données et de contrôle et qui se chargera de l'étape Fetch
- L'**Execution Unit** (EU) qui contient les blocs chargés des étapes Decode, Execute et qui contient en outre plusieurs registres et des caches.

4.2. Caractéristiques techniques

Les processeurs 8088 et 8086 sont basés sur le même modèle général (fig. 4.1) :

- des bus d'**adresses interne et externe** de 20 bits de large, permettant d'accéder à 1 MB de mémoire centrale,
- un **bus de données externe** de 8 bits de large
- un **bus de données interne** de 16 bits de large. Il faudra donc deux ordres de lecture ou d'écriture pour passer du bus externe au bus interne et réciproquement.
- un **bus de contrôle** (17 bits)
- Fréquence interne de 5 ou 8 MHz (turbo mode)
- Architecture CISC capable d'effectuer les opérations arithmétiques signées ou non signées sur des nombres binaire de 8 ou 16 bits (y compris la multiplication et la division)
- Les opérations arithmétiques sur les nombres réels (float) ne sont pas implémentées et doivent être émulées lors de la compilation. Cependant, le 8086 peut être couplé à un co-processeur mathématique 8087 qui, lui, dispose des circuits nécessaires aux calculs sur des réels

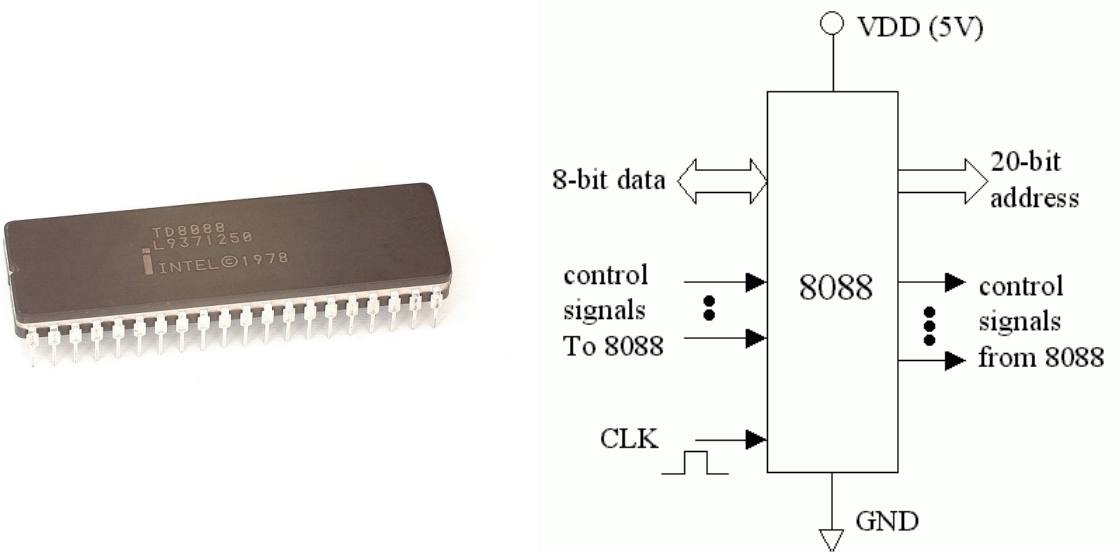


fig. 4.1 Le processeur 8088

Il est intéressant de noter que, pour éviter la prolifération de pins de connexion (40 pour le 8088), les 8 bits de données et les 8 premier bits des adresses étaient multiplexés sur les mêmes connecteurs.

4.3. La segmentation de la mémoire

4.3.1. Chargement d'un programme

Jusqu'à présent, nous avons admis que le programme écrit en langage machine se trouvait en mémoire centrale RAM. Les choses ne sont pas aussi simples.

Lors de la création du programme, le compilateur effectue la traduction des instructions (écrites par exemple en langage C) en langage machine et sauve le résultat sur le disque dur, sous la forme d'un fichier qui porté généralement l'extension **.exe**.

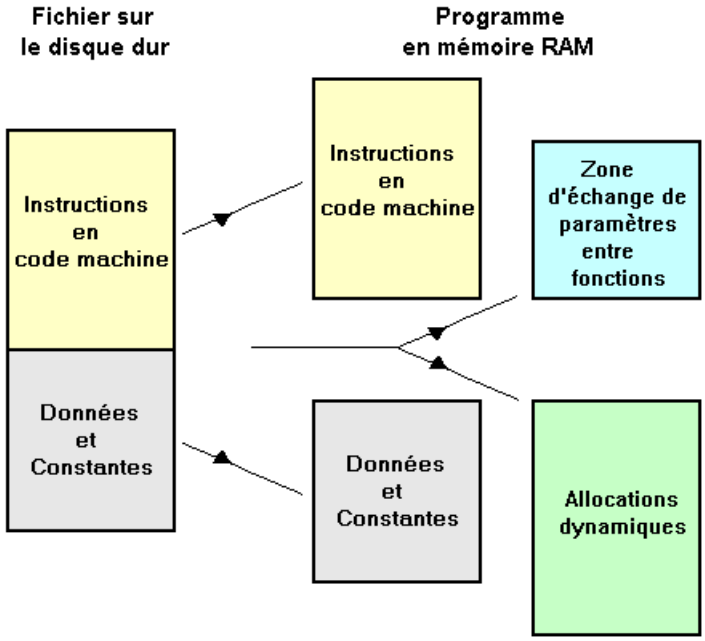


fig. 4.2 Segmentation d'un programme

Ce fichier est structuré en deux parties : l'une contient les instructions (code), l'autre les valeurs et les différentes constantes utilisées par les instructions (data). Il faut aussi prévoir une zone d'échange de données entre les différentes routines qui constituent le programme. De plus, dans la plupart des cas les variables organisées

en tableaux ne pourront être dimensionnées que lors de l'exécution du programme (fig. 4.2).

Il est important de se souvenir qu'un programme est structuré de manière relative (voir 2.2.3). Lors de la compilation, toutes les variables symboliques ont été transformées en adresses relatives. Il n'est pas nécessaire de connaître la position absolue d'une instruction ou d'une variable (35ème byte ou 318ème byte du texte) mais bien leur position relative (35 bytes *après* le byte de début de programme, 318 bytes *après* le byte de début de la zone de données).

Pour que le programme puisse être exécuté, le système d'exploitation doit d'abord le copie en mémoire centrale. Mais il existe déjà plein de choses en mémoire centrale, à commencer par les instructions propres au système d'exploitation lui-même. Il n'est donc pas possible de stocker le programme à partir d'une adresse précise car rien ne garantit qu'elle est libre.

Comment le pauvre pointeur d'instructions du processeur va-t-il savoir où il doit commencer à lire le programme et comment il pourra suivre son exécution ?

4.3.2. Les types de segments

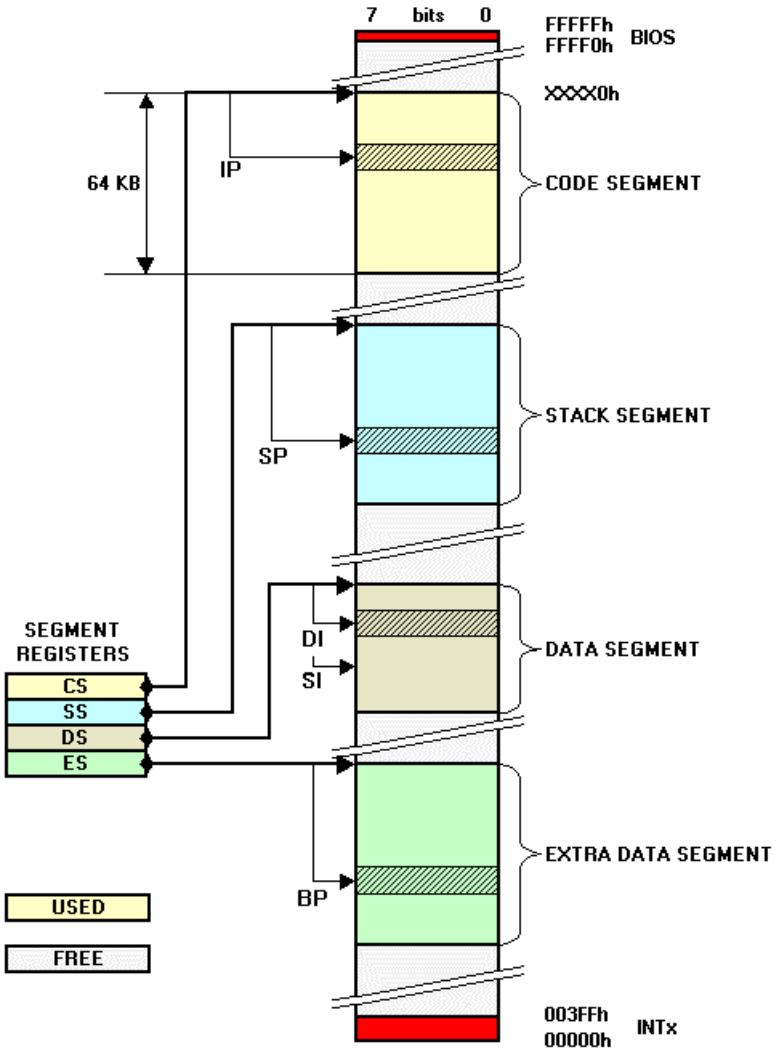


fig. 4.3 Segmentation de la mémoire RAM pour l'exécution d'un programme

La solution adoptée consiste à découper la mémoire RAM en tranches appelées **segments**. Un segment est un bloc de 64 kB qui commence à une adresse multiple

de 16 (soit une adresse du type $XXXX0_h$). Tous ces segments sont repris dans une table maintenue par le système d'exploitation ([1]).

Lorsqu'iil doit charger un programme, le système d'exploitation explore sa table afin de trouver l'espace nécessaire pour caser les quatre segments du programme :

- le **Code Segment** (**CS**), qui héberge les instructions. Un programme peut occuper plusieurs "code segments" en fonction de sa taille mais il n'y a qu'un seul segment actif à la fois ([2]);
- le **Stack Segment** (**SS**), qui héberge les valeurs qui sont passées d'un programme à un sous-programe et réciproquement;
- le **Data Segment** (**DS**), où sont stockées toutes les variables définies de manière statiques par le programme;
- l'**Extra Segment** (**ES**), qui gère les chaînes de caractères et l'espace alloué dynamiquement aux données du programme en cours d'exécution, en fonction de ses besoins.

En d'autre mots, le système d'exploitation copie la suite de bytes formant le bloc d'instructions du fichier **`.exe`** à l'endroit nommé Code Segment. Il copie ensuite la suite de bytes formant le bloc de données de ce même fichier **`.exe`** à l'endroit nommé Data Segment. Ensuite, il réserve une zone pour les variables qui servent au transfert des paramètres lors des appels de fonctions (Stack Segment) et une autre pour les variables qui seront allouées dynamiquement (Extra Segment).

Chacune de ces zones peut commencer à (presque) n'importe quelle adresse. D'une exécution à la suivante, rien ne garantit que le programme utiisera toujours la même zone de mémoire. Il est même plutôt certain que le programme ***n'utilisera pas*** deux fois de suite la même série de segments.

Les adresses de tous ces segments sont fournies par l'OS et stockées dans le processeur, dans la zone du BIU, dans des registres spécialement prévus à cet effet, nommés **segment registers** CS, SS, DS, ES. Dans le processeur 8088, ces registres ont une taille de 16 bits.

Pour aller chercher une instruction qui se trouve à une adresse physique précise en mémoire centrale, le processeur additionnera la valeur du pointeur d'instruction "relatif" (IP) à la valeur "absolue" de l'adresse du code segment (CS).

De même, pour ecrire ou lire une donnée, il utilisera le positionnement relatif donné par le programme (Offset SP) et l'additionnera à l'adresse "absolue" du début du stack segment (SS).

Tous ces pointeurs "relatifs" occupent eux aussi une série de registres spécialisés (*ang.: Pointers and index registers*) IP, SP, SI (source) et DI (destination), BP, etc. Ces registres ont aussi une taille de 16 bits ([3]).

1 La description de ce mécanisme fait partie du cours de Système d'Exploitation

2 De plus, parmi tous les programmes chargés simultanément en mémoire centrale, il n'y en a jamais qu'un seul qui soit actif à un moment donné. Le processeur passe successivement de l'un à l'autre en consacrant à chacun une petite tranche de son temps. On peut donc dire qu'a tout moment, il n'y a qu'un seul Code Segment actif.

3 Toujours dans le cas du processeur 8088 étudié ici.

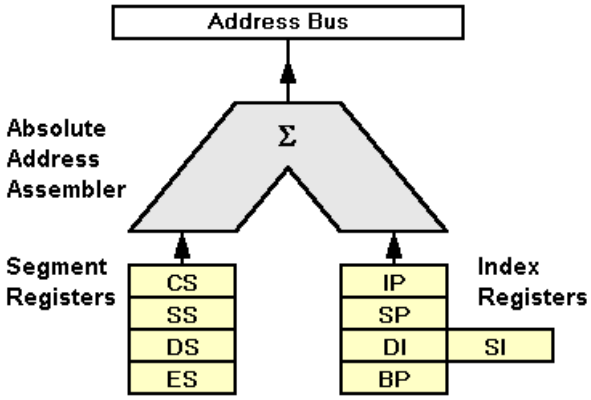


fig. 4.4 Calcul relatif des adresses

Tout accès à la mémoire, que ce soit pour lire une instruction ou pour écrire une donnée, passe obligatoirement par un calcul du type "base + offset" dont le résultat est envoyé sur le bus d'adresse.

4.3.3. Le calcul d'adresse sur le 8088

Il a été dit que le processeur 8088 disposait d'un bus d'adresse de 20 bits de large, ce qui lui permet d'adresser 1 MB de mémoire. Comment peut-il créer des adresses sur 20 bits alors que les registres n'ont que 16 bits de large ?

L'astuce consiste à décaler l'adresse du pointeur de base de 4 bits vers la gauche et à remplir les quatre bits à droite avec des zéros. Ce faisant, le nombre a été multiplié par 16 et il est passé de 16 à 20 bits. C'est donc maintenant un multiple de 16 et, écrit en hexadécimal, il occupe 5 nibbles dont le dernier à droite est forcément 0 ([1]).

Il suffit ensuite d'ajouter la valeur de l'offset (sans la décaler) et d'envoyer le tout sur le bus d'adresse.

Code segment (CS)	**3**	**4**	**8**	**A**	0
+ Instruction pointer (IP)		**4**	**2**	**1**	**4**
= Instruction address	**3**	**8**	**A**	**B**	**4**

Répétons que le même principe s'applique à *tous* les calculs d'adresse sur le 8088. La BIU, qui est chargée de l'interface avec la mémoire, disposera donc d'une unité arithmétique sommaire spécialisée pour ce type de calcul.

Cette astuce a souvent été critiquée car il est possible d'atteindre une même adresse avec plusieurs couples CS+IP. Ceci ne pose aucun problème en Assembler, mais une certaine confusion peut apparaître dans les langages qui, tels le C, utilisent intensivement les pointeurs. D'autre part, cette astuce est spécifique à l'adressage jusqu'à 1 MB. Au-delà, une autre technique d'adressage devra être utilisée.

Dans les processeurs plus récents (80286 et suivants), la décomposition des adresses en segment et offset est toujours de mise. Toutefois, ces processeurs disposent non seulement de bus d'adresse beaucoup plus larges (32 bits pour adresser 4 GB de RAM) mais, de plus, la taille des registres correspond au moins à la taille du bus. Dès lors, l'astuce "shift and add" n'est plus nécessaire et le calcul se ramène à une simple addition ([2]). Cependant, pour des raisons de compatibilité

1 Voilà pourquoi les segments doivent obligatoirement commencer à une adresse multiple de 16, du type XXXX0h.

2 Notons au passage que transfert d'informations d'un bus 16 bits vers un bus 32 bits équivaut à un ./.

"arrière" (*ang.: backward compatibility*), tous les processeurs de la famille Intel sont encore capables de travailler selon l'ancienne recette (*ang.: Real mode addressing*).

4.4. La pile d'instructions

Pour le processeur élémentaire analysé au chapitre précédent, nous avons admis que toutes les instructions avaient une taille de 2 bytes.

Les processeurs 8086 et 8088 admettent des instructions qui sont codées sur 1, 2, 3 ou 4 bytes selon leur complexité.

Il s'ensuit que, lors de la phase de "fetch", le pointeur d'instructions IP ne sait pas *a priori* combien de bytes il doit rapatrier de la mémoire RAM vers le registre d'instruction courante. Certes, le décodeur est capable d'extraire et d'analyser l'OpCode et, partant, de connaître la taille de l'instruction (à un OpCode donné correspond toujours la même taille d'instruction). Cependant, la phase de décodage intervient *après* la phase de "fetch" et, par conséquent, l'information sur la taille n'est pas encore disponible au moment où il faut chercher les instructions.

La solution consiste à créer un tampon ou pile (*ang.: buffer* ou *queue*) de quelques bytes ([1]). L'unité "fetch" du processeur veille à ce que la pile soit constamment alimentée. Comme la taille de la pile est supérieure à la taille de la plus grande instruction possible, la pile contient toujours au moins une instruction complète et le début de la suivante.

L'unite "decode" dispose ainsi de tous les bytes nécessaires à l'analyse d'une instruction. Elle prélève dans la pile les bytes au fur et à mesure de ses besoins. Les bytes analysés et utilisés sont retirés de la pile, ce qui provoque la descente des bytes résiduels et crée un trou en sommet de pile. L'unité "fetch" va alors chercher en mémoire le byte ou les bytes suivants pour les placer au sommet de la pile.

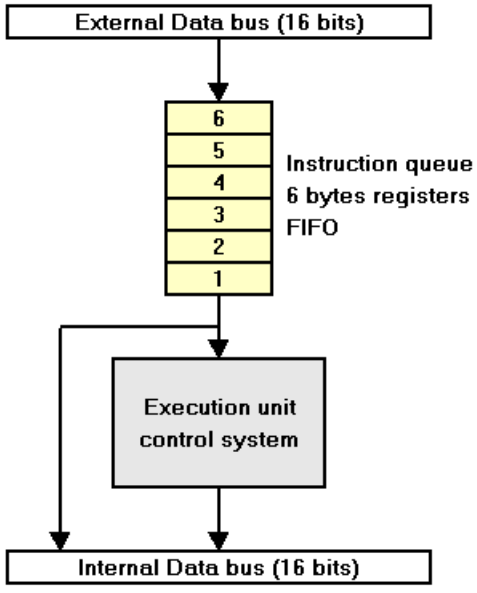


fig. 4.5 Queue FIFO des instructions

Cette technique est l'amorce du principe de "pipelining" qui sera développé dans un autre chapitre.

---

mécanisme "shift and add" dans lequel on décale le premier chiffre de 16 bits vers la gauche avant d'additionner le second.

1 6 bytes dans le 8086, 4 bytes dans le 8088, beaucoup plus dans les processeurs actuels

4.5. Les registres généraux

Le processeur élémentaire effectuait ses opérations depuis et vers un registre nommé accumulateur.

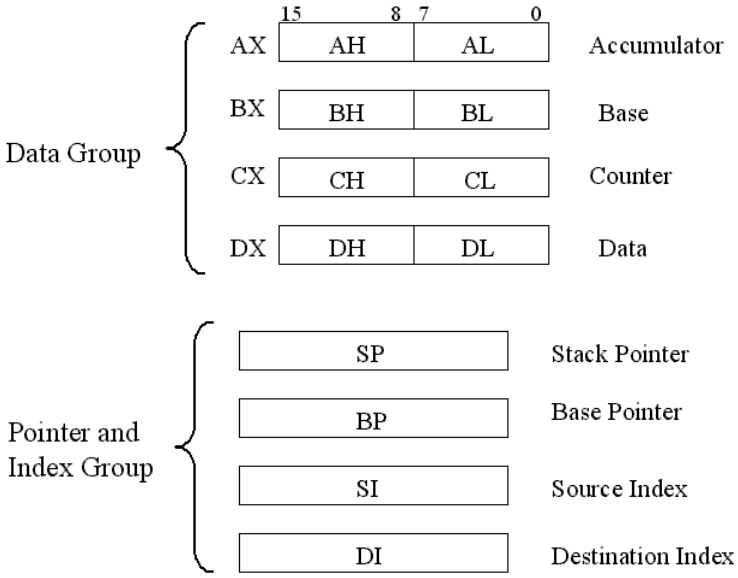


fig. 4.6 Les registres généraux

Le processeur 8088 dispose de quatre registres de calcul et de quatre registres d'index dont il a déjà été question plus haut. L'avantage des registres est de permettre le stockage local de données intermédiaires sans devoir recourir à des transferts vers la mémoire RAM.

Les registres de calcul ont une taille de **16 bits**. Ils ont été nommés A, B, C, D et sont suffixés par X, H ou L selon que l'on utilise les deux bytes, seulement de byte supérieur (*ang.: high*) ou seulement le byte inférieur (*ang.: low*). Ils sont absoluments équivalents mais l'usage veut qu'on réserve préférentiellement à chacun un rôle particulier :

- AX ou Accumulator (= AH + AL), utilisé dans les opérations arithmétiques;
- BX ou Base (= BH + BL), utilisé surtout pour l'adressage;
- CX ou Count (= CH + CL), utilisé principalement pour les comptages;
- DX ou Data (= DH + DL) utilisé surtout pour stocker des adresses et des pointeurs vers d'autres zones de la machine.

Les registres de pointeurs ont aussi une taille de 16 bits. A la différence des registres de calcul, ils ne sont pas accessibles par moitié. Leur rôle est aussi plus spécifique :

- SP ou Stack Pointer. La stack (*fr.: pile*) est une zone de mémoire dans laquelle on empile les informations qui doivent être passées d'une routine à l'autre. Elle est gérée en mode LIFO;
- BP ou Base Pointer contient l'adresse de la dernière donnée qui a été placée sur la stack;
- SI est utilisé pour l'adressage indirect
- DI est utilisé comme SI et pour l'adressage des chaînes de caractères.

4.6. Le registre des flags

Le registre des flags a une taille de 16 bits. Il contient de l'information relative au statut courant du microprocesseur (*ang.: status flag*) ainsi que certaines informations

de contrôle (*ang.: control flag*). Chaque bit du registre de flags a une signification particulière.

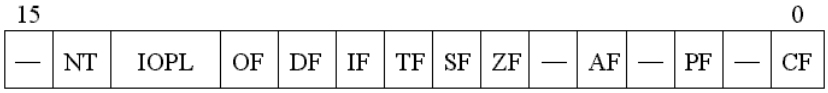


fig. 4.7 Le registre des flags

Les flags de contrôle sont :

- IF (*ang.: Interrupt enable flag*) autorise ou non les interruptions
- DF (*ang.: Direction flag*)
- TF (*ang.: Trap flag*) piège les erreurs éventuelles

Les flags de statut sont :

- CF (*ang.: Carry flag*) définit s'il y a un report
- PF (*ang.: Parity flag*) définit la parité du résultat
- AF (*ang.: Auxiliary carry flag*) autre définition d'un report
- ZF (*ang.: Zero flag*) définit si le résultat est nul
- SF (*ang.: Sign flag*) définit si le résultat est négatif
- OF (*ang.: Overflow flag*) définit si le calcul a provoqué un overflow
- NT (*ang.:Nested task flag*) définit s'il s'agit d'un résultat intermédiaire
- IOPL (*ang.: Input/output privilege level*)

4.7. Le schéma bloc global du 8088

Un schema global du processeur 8088 est présenté ci-dessous. Le bus et les lignes de contrôle ont été omises pour des raisons de clarté. On notera en particulier que l'information circule librement entre les registres grâce au bus interne de données.

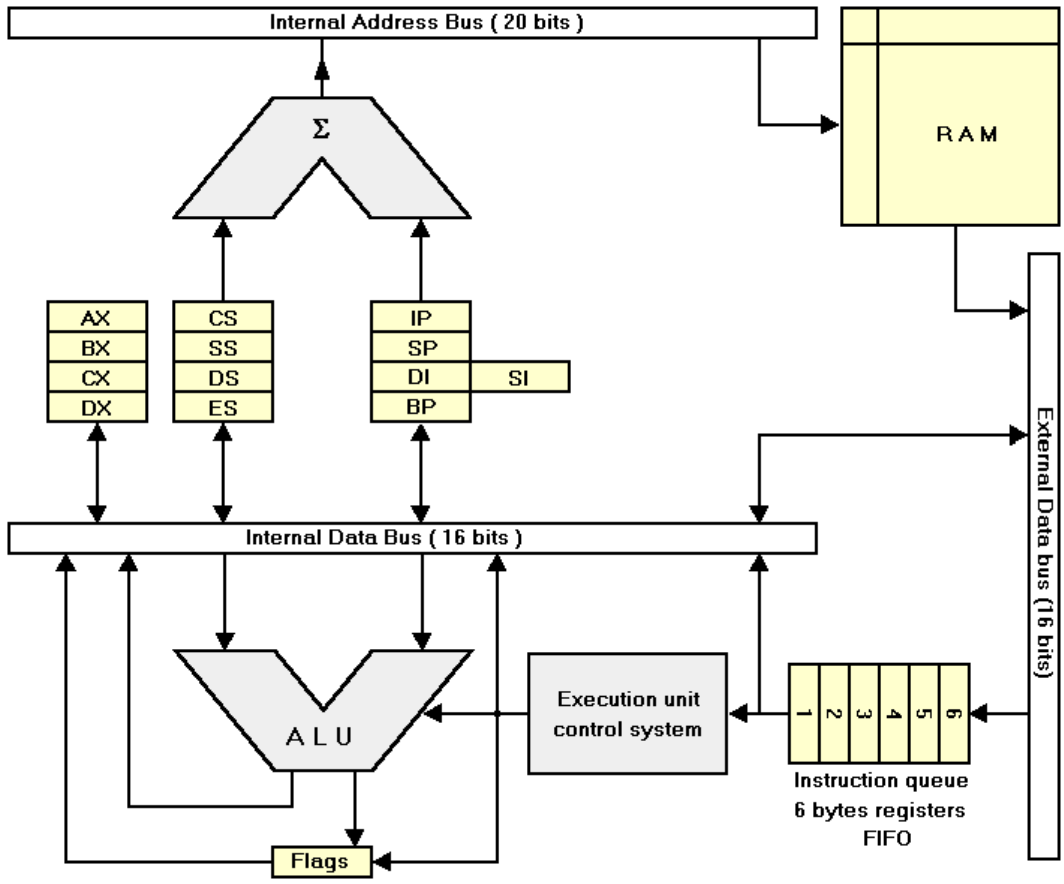


fig. 4.8 Diagramme bloc d'un processeur 8088

4.8. Un mot sur les adresses réservées

Dans l'ordinateur, certaines adresses sont réservées pour des applications spéciales. Des programmes ne peuvent pas y être chargés.

La zone d'adresses comprises entre $FFFF0_h$ et $FFFFF_h$, soit 16 bytes, est localisée dans la ROM du BIOS. C'est vers cette zone que pointe le compteur d'instructions lors du démarrage de la machine (registers CS=$FFFF_h$ et IP=0000_h). Cette zone est évidemment trop petite pour contenir un programme complet mais elle est suffisante pour contenir une instruction de branchement vers une autre adresse située dans la zone du BIOS (de $F0000_h$ à $FFFFF_h$)

La zone d'adresses comprises entre 00000_h et $003FF_h$ contient la table des pointeurs d'interruption.

Une interruption (*ang.: interrupt*) est un événement qui survient pendant que le processeur est en train d'exécuter un programme. Cet événement peut être déclenché par un périphérique (*ang.: hardware interrupt*) ou par une instruction du programme en cours (*ang.: software interrupt*). Les interruptions sont caractérisées par un code compris entre 00_h à FF_h. De ce fait, il y a 256 codes d'interruption possibles et la table des pointeurs d'interruption compte 256 entrées de 4 bytes chacune (soit 1 kB au total).

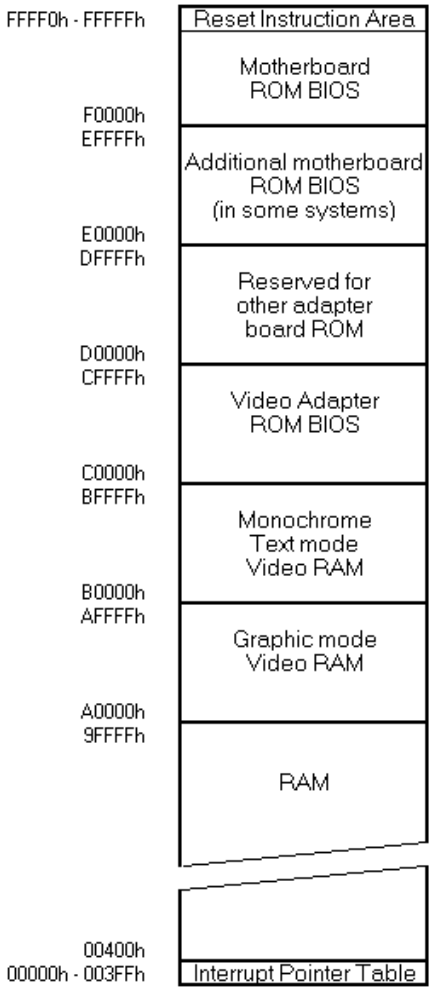


fig. 4.9 Zones d'adresses réservées

L'arrivée d'un interrupt (p.ex.: 21_h) suspend l'exécution du programme en cours. Le processeur lit de l'entrée de la table correspondante (p.ex.: située à l'adresse $21_h * 4 = 84_h$) qui n'est autre que l'adresse mémoire (16 bits CS + 16 bits IP) d'une routine à exécuter (*ang.: interrupt service routine*). Lorsque cette routine a été exécutée, le processeur reprend l'exécution du programme initial.

5. Les perfectionnements

5.1. Le pipelining

5.1.1. Principe

Le pipelining est au processeur ce que le travail à la chaîne est à l'industrie manufacturière.

Dans l'industrie, le montage à effectuer est découpé en une succession de phases, chacune de ces phases est confiée à un groupe d'ouvriers qui s'approvisionne auprès du groupe précédent et transmet sa réalisation au groupe suivant.

- si toutes les tâches prennent exactement le même temps, la chaîne de montage peut être cadencée comme un processus **synchrone**.
- si l'une au moins des tâches s'effectue en un temps différent des autres, alors la chaîne de montage est **asynchrone** et il faut prévoir des bacs tampons entre les différents ateliers.

Dans le cas du processeur, nous savons déjà que le traitement d'une instruction est décomposé en quatre phases : Fetch, Decode, Execute, Store. Nous avons vu que le processeur 8088 découple déjà la phase Fetch du reste du traitement. Il suffit de poursuivre le raisonnement est de confier la réalisation de chacune des phases à une unité spécialisée (fig. 5.1). Cette organisation porte le nom de **pipeline**.

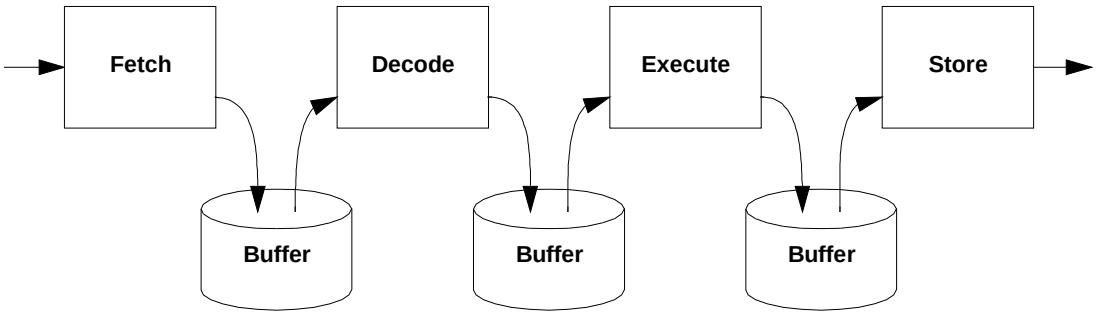


fig. 5.1Découpage du traitement

A l'origine [t_0] le processeur est inactif; tous les étages du pipeline sont inoccupés (tab. 5.1).

Temps	Fetch	Decode	Execute	Store
t_0	-	-	-	-
t_1	**i_1**	-	-	-
t_2	i_2	**i_1**	-	-
t_3	i_3	i_2	**i_1**	-
t_4	i_4	i_3	i_2	**i_1**
t_5	i_5	i_4	i_3	i_2

tab. 5.1 Fonctionnement du pipeline

A l'instant [t_1] l'étage **Fetch** va chercher une instruction [i_1] en mémoire. Tant que les bytes qui contiennent cette instruction ne sont pas arrivés, Decode et tous les étages suivants sont inactifs (*ang.: idle*).

A l'instant [t_2] Fetch a reçu l'instruction [i_1] et l'a déposée dans son buffer aval où **Decode** l'a récupérée pour commencer à l'interpréter. Fetch peut déjà aller chercher l'instruction suivante [i_2] en mémoire. Toutefois, Execute et Store ne disposent encore d'aucun élément pour commencer leur tâche.

A l'instant [t_3] Decode a fini le traitement de [i_1] et l'a transmise à **Execute** via son buffer aval. Entre-temps, Fetch, qui a récupéré l'instruction [i_2], l'a déposée dans son buffer aval afin que Decode puisse d'en occuper. Fetch peut maintenant aller chercher l'instruction [i_3] en mémoire. Toutefois, Store est toujours inactif.

A l'instant [t_4] l'instruction [i_1] arrive dans le buffer amont de **Store** qui peut enfin s'en occuper. Execute récupère et exécute [i_2] venant de Décode, au moment où Decode reçoit l'instruction [i_3] venant de Fetch. Fetch peut maintenant aller chercher l'instruction [i_4] en mémoire…

Le pipeline est amorcé. La traversée du pipeline par la première instruction, c'est-à-dire le temps nécessaire à son traitement complet, a demandé quatre intervalles de temps. Ce délai est appelé **temps de latence** (*ang.: latency time*).

Notons que le traitement d'**une** instruction par le processeur est **plus long** que sans pipeline car, *en moyenne*, chaque étage doit travailler au rythme de l'étage le plus lent, sinon les buffers intermédiaires se remplissent (ou s'épuisent) et il y a émission d'une alarme "**wait state**" ([1]). De plus, l'écriture des diverses productions dans ces buffers intermédiaires demande un temps de stabilisation et n'est donc pas instantanée.

Par contre, à partir du moment où le pipeline est amorcé, une instruction se termine à chaque intervalle de temps et le **débit global** du processeur (*ang.: throughput*) a été multiplié environ par quatre. A chaque instant, il y a autant d'instructions en cours de traitement que d'étages dans le pipeline.

5.1.2. Choix de la profondeur de pipeline

Le processeur Intel286, qui a connu un beau succès commercial, est l'un des premiers à implémenter la technique du pipeline. Le processeur Intel386SX poussera cette exploitation encore plus loin.

Nous avons illustré le principe d'un processeur RISC avec un pipeline à quatre étages mais, comme dans une chaîne de montage, plus on détaille les opérations, plus le nombre d'étages augmente. Les concepteurs de processeurs sont amenés à faire des compromis :

Peu d'étages	**Beaucoup d'étages**
- le temps de cycle doit être long car chaque étage est un circuit volumineux qui demande un temps de stabilisation long	- le temps de cycle peu être court car les étages sont des circuits petits et rapides
- peu de registres de pipeline pour assurer les tampons intermédiaires	- beaucoup de registres de pipeline pour assurer les tampons intermédiaires
- temps de traversée d'une seule instruction demande peu de cycles	- temps de traversée long mais débit élevé
- vidange de pipeline rapide	- vidange de pipeline pénalisant

1 Comme dans le cas des bus et des contrôleurs de la carte mère.

Un processeur Intel Pentium IV possède un pipeline avec une vingtaine d'étages.

5.2. Problèmes de gestion d'un pipeline

Un pipeline sera inévitablement perturbé si les instructions qu'il contient sont interdépendantes telles que :

- opération arithmétique longue
- opérations qui accèdent aux mêmes données
- branchement conditionnel
- saut inconditionnel

Toutes ces perturbations entraînent la purge puis le réamorçage d'un ou de plusieurs étages du pipeline, ce qui entraîne une chute des performances.

5.2.1. Opération arithmétique longue

La plupart des opérations élémentaires (affectation, addition d'entiers, multiplication d'entiers) prennent exactement un cycle de calcul et ne posent aucun problème dans le pipeline.

Les divisions et les opérations arithmétiques sur les réels prennent beaucoup plus de temps d'exécution et génèrent des délais en aval.

Considérons le pipeline amorcé du tableau ci-dessous (tab. 5.2) dans lequel, au temps [t_0] :

- l'exécution de l'instruction [i_1] est terminée et le résultat est en cours de stockage,
- l'instruction [i_2] est une addition d'entiers en cours d'exécution (p.ex.: ia+ib),
- l'instruction [i_3] est une division de deux réels en cours de décodage (p.ex.: fa/fb).

Temps	Fetch	Decode	Execute	Store
t_0	i_4	**i_3**	i_2	i_1
t_1	i_5	i_4	**i_3**	i_2
t_2	i_5	i_4	**i_3**	-
t_3	i_5	i_4	**i_3**	-
t_5	i_6	i_5	i_4	**i_3**

tab. 5.2 Bulle d'exécution dans le pipeline

En [t_1], [i_2] est transféré à l'étage Store et [i_3] prend sa place dans l'étage Execute.

Tout se passe bien pendant le premier cycle du processeur… Malheureusement, la division est une opération complexe qui monopolise Execute pendant plusieurs cycles. Pendant ce temps Store est inemployé tandis que Fetch et Decode doivent suspendre leur activité.

Au bout d'un certain temps [t_5], l'exécution de [i_3] est terminée et le pipeline peut reprendre son fonctionnement normal.

5.2.2. Remède

Un remède à ce problème consiste à insérer un tampon (*ang.: buffer*) entre les étages Decode et Execute comme indiqué sur la fig. 5.1 :

Temps	Fetch	Decode	*Buffer*	Execute	Store
t_0	i_4	**i_3**	-	i_2	i_1
t_1	i_5	i_4	-	**i_3**	i_2
t_2	i_6	i_5	i_4	**i_3**	-
t_3	i_7	i_6	i_5 ,i_4	**i_3**	-
t_5	i_8	i_7	i_6 ,i_5	i_4	**i_3**

tab. 5.3 Résorption du blocage amont par insertion d'un buffer

Toutefois, cette solution ne résout pas totalement le problème :

- le buffer ayant une taille limitée, il sera quand même vite saturé si quelques opérations complexes se succèdent,
- le retard accumulé dans le buffer ne pourra être résorbé par Execute que si la cadence de travail augmente (improbable) ou si celle de Decode diminue (possible).

5.2.3. Délai d'accès aux données

Nous savons que la mémoire RAM travaille à une cadence inférieure à celle du processeur. Un bulle apparaîtra aussi quand Decode, qui doit aller chercher les opérandes, subit le délai de livraison de la RAM.

Dans l'exemple ci-dessous, supposons que ce soit l'instruction [i_3] qui demande une opérande stockée en mémoire.

Temps	Fetch	Decode	Execute	Store
t_0	i_4	**i_3**	i_2	i_1
t_1	i_4	**i_3**	-	i_2
t_2	i_4	**i_3**	-	-
t_3	i_5	i_4	**i_3**	-
t_5	i_6	i_5	i_4	**i_3**

tab. 5.4 Bulle due au décodage dans le pipeline

Toutefois, si le processeur intègre un buffer intermédiaire comme indiqué au tab. 5.3, Execute sera en mesure de poursuivre son activité et même de résorber son retard.

Une situation similaire se présente quand l'opérande à laquelle Decode doit accéder dépend d'un calcul en cours d'exécution, par exemple lors du calcul d'index d'un tableau :

```
⋮
x = v[ 3*i ] + v[ 3*i+1 ];
j++;
⋮
```

Par exemple, si nous admettons que la décomposition de ces instructions se fait de la manière suivante :

```
i1 à 3*i
i2 à 3*i+1
i3 à v[ 3*i ] + v[ 3*i+1 ]
i4 à x = v[ 3*i ] + v[ 3*i+1 ]
i5 à j++
```

Lorsque Decode rencontre l'instruction [i_3], il est incapable d'aller chercher les opérandes tant que le calcul des deux indices n'a pas été réalisé par Execute et que le résultat n'a pas été écrit quelque part par Store :

Temps	Fetch	Decode	Execute	Store
t_0	**i_3**	i_2	i_1	. . .
t_1	i_4	**i_3**	i_2	i_1
t_2	i_4	**i_3**	-	i_2
t_3	i_4	**i_3**	-	-
t_5	i_5	i_4	**i_3**	-

tab. 5.5 Bulle due à l'attente d'un résultat

5.2.4. Remède

Lors de la décomposition de l'instruction en étapes élémentaires, un compilateur intelligent remarquera que l'instruction [i_5] n'utilise pas les variables mentionnées dans les instructions précédentes. Elle pourrait donc remonter dans la pile d'instructions :

```
i1 à 3*i
i2 à 3*i+1
i5 à j++
i3 à v[ 3*i ] + v[ 3*i+1 ]
i4 à x = v[ 3*i ] + v[ 3*i+1 ]
```

Actuellement, cette opération est effectuée, au moment de l'exécution, par un module du processeur qui examine les instructions placées dans la file d'attente (buffer) situé entre Fetch et Decode et les replace dans un ordre optimal.

Cette **exécution dans le désordre** (*ang.: out of order execution*) permet de combler tout ou partie des bulles. Le pipeline devient (tab. 5.6)

Temps	Fetch	Decode	Execute	Store
t_0	**i_5**	i_2	i_1	. . .
t_1	**i_3**	**i_5**	i_2	i_1
t_2	i_4	**i_3**	**i_5**	i_2
t_3	i_4	**i_3**	-	**i_5**
t_5	i_6	i_4	**i_3**	-

tab. 5.6 Bulle partiellement comblée

5.2.5. Branchements conditionnels

Les branchements conditionnels sont générés par les instructions de test (if…then…else) que l'on retrouve aussi implicitement dans le contrôle de boucles (for, do…while, while). Ils portent aussi le nom de rupture de séquence.

Dans le pseudo-code ci-dessous, lorsque Fetch va chercher le test `if`, il en ignore le résultat. Il est donc incapable de déterminer si l'instruction qui devra être exécutée ensuite fera partie du bloc `then` ou du bloc `else`.

```
⋮
if (x > 0)
then y = v[ 3*i ] + v[ 3*i+1 ]
else y = w[ 3*i ] + w[ 3*i+1 ]
endif
⋮
```

Si Fetch poursuit son travail en chargeant les instructions du bloc `then` alors que Execute détermine plus tard qu'il fallait prendre celle du bloc `else`, il faudra vider le pipeline et tout le travail réalisé entre-temps l'aura été en vain.

Le coût de cette erreur peut être dramatique, surtout si les instructions qui suivent le test font elles-mêmes appel à du calcul d'indice. Or ces branchements sont très fréquents dans un programme typique : un test pour dix instructions !

5.2.6. Remède

Les processeurs intègrent un module de prédiction de branche (*ang.: branch prediction module*) qui tente d'anticiper le résultat du test avant son exécution.

Ce module utilise une méthode heuristique ou probabiliste qui se base sur le fait que certaines instructions (Opcodes) sont plus susceptibles de provoquer une rupture de séquence que d'autres. Par exemple :

Instruction	Issue la plus probable	Justification
`if (`**`x=0`**`)`	faux (else)	la probabilité de tirer 0 est beaucoup plus faible que celle de tirer un autre nombre
`if (`**`x<0`**`)`	faux (else)	on travaille plus souvent avec des nombres positifs que négatifs
`for(I=0;` **`I<n`**`; i++)`	vrai (boucle)	dans une boucle `for` il est plus probable de recommencer la boucle que de la quitter.

Cette technique atteint jusqu'à 75% de taux de succès (contre 50% si on prend une branche au hasard).

Les algorithmes actuels prennent aussi leurs décisions sur base de l'histoire récente. Ils sont ainsi capables de reconnaître une séquence déjà rencontrée et en déduire qu'ils sont dans une boucle. Dès lors, la probabilité de recommencer encore une fois la boucle est plus grande que celle d'en sortir.

Aujourd'hui le taux de succès de ces modules dépasse 90%.

5.3. Les architectures superscalaires

5.3.1. Principe

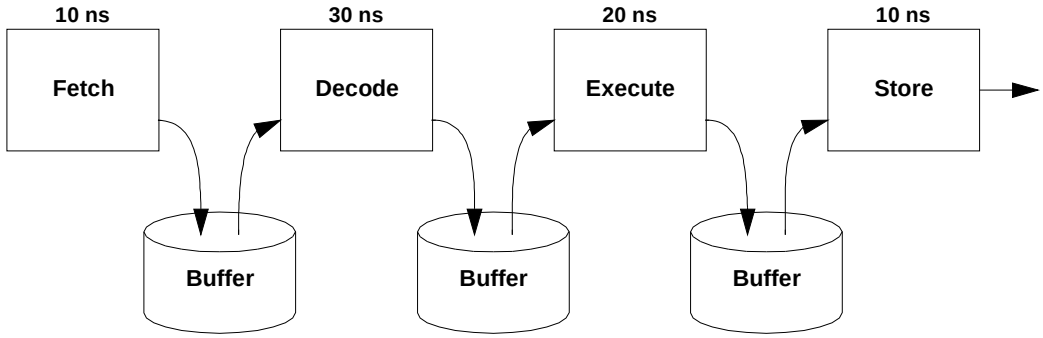


fig. 5.2 Durée inégale des traitements

Reprenons la chaîne de traitements illustrée à la fig. 5.1 mais imaginons cette fois que les différents traitements exigent des délais de réalisation différents. Il est clair que l'étape la plus lente (ici Decode) va imposer sa cadence à toutes les autres, ce qui nuit à la productivité.

La solution appliquée dans l'industrie consiste à dédoubler l'atelier le plus lent ou le plus sollicité : le traitement d'une pièce prend toujours le même temps mais en faisant travailler deux ateliers en parallèle, on double la production. Tout se passe comme si le temps de traitement était divisé par deux.

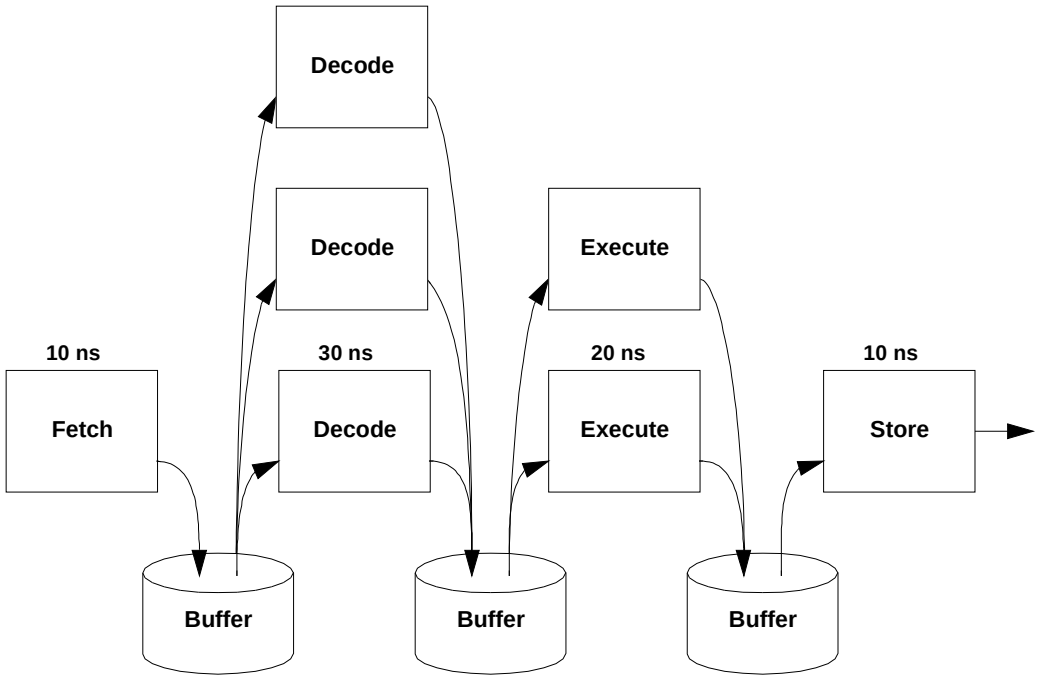


fig. 5.3 Solution : dédoubler les ateliers les plus lents

Ce principe est appliqué dans les processeurs modernes : en les dotant de plusieurs pipelines et tampons, on leur permet de traiter des instructions en parallèle, ce qui augmente le nombre d'instructions exécutées par cycle (fig. 5.3). On parle alors d'architecture superscalaire (*ang.: superscalar architecture*).

5.3.2. Implémentation

Dans une architecture superscalaire, les pipelines sont souvent spécialisés. On distingue trois grands types de pipelines :

- **traitement des entiers** (UAL)
  opérations sur des entiers (IADD, ISUB, IMUL, IDIV),
  décalages de bits ;
- **traitement des réels** (UVF)
  opérations sur réels (FADD, FSUB, FMUL, FDIV),
  fonctions trigonométriques, logarithmiques, racine carrée.
  Ce module dispose d'une petite mémoire ROM qui contient les valeurs des constantes usuelles (π, e, $\log_2(e)$, $\log_2(10)$, $\log_e(2)$) ainsi qu'une table de valeurs trigonométriques ;
- **traitement des adresses (GA)**
  calcul d'adresse pour les opérations `load` et `store`.

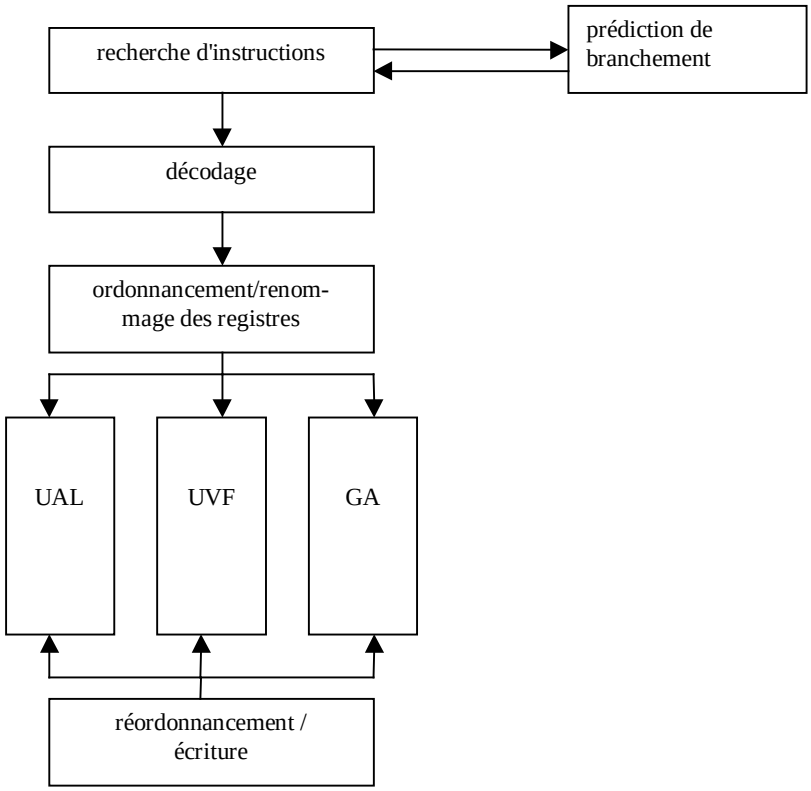


fig. 5.4 Organisation d'une architecture superscalaire

Comme une application contient généralement beaucoup plus de traitement d'entiers (indices de tableaux, indices de boucles, flags) que de traitement de réels, le processeur dédoublera les lignes de calcul sur entier.

La tendance actuelle consiste à ajouter des unités de traitement d'instructions spéciales telles que celles destinées aux multimédia (traitement numérique du son et de l'image MMX).

5.3.3. Problèmes liés au superscalaire

Dans une architecture superscalaire, une instruction de haut niveau est découpée en segments qui seront traités sur des modules différents.

```
float x, v[10];
int i;
⋮
x = v[ 3*i ] + v[ 3*i+1 ];
⋮
```

Ainsi, dans l'instruction ci-dessus, le calcul des indices (de type `int`) sera confié à au pipeline des entiers (ou à celui des adresses) tandis que le calcul proprement dit, qui concerne des variables de type `float` , sera confié au pipeline des réels.

On comprend que ce découpage nécessite une communication entre les différents processus et un sérieux travail de gestion des résultats… qui sera confié à des modules de synchronisation (*ang.: scheduler*).

5.4. Les caches

Les instructions récupérées par Fetch sont stockées dans un cache interne (L1). Si Fetch doit aller rechercher en RAM une instruction dont il a déjà rencontré l'adresse, il va plutôt la chercher dans son cache. Gain de temps.

Les plus vieilles instructions du cache sont progressivement replacées par les plus récentes.

Les instructions périmées ne sont pas perdues mais envoyées dans une cache de seconde ligne : la cache L2

Decode fait de même avec les instructions qu'il décode. Elles sont stockées dans un cache et rappelées si elles doivent être réexécutées.

(à développer…)

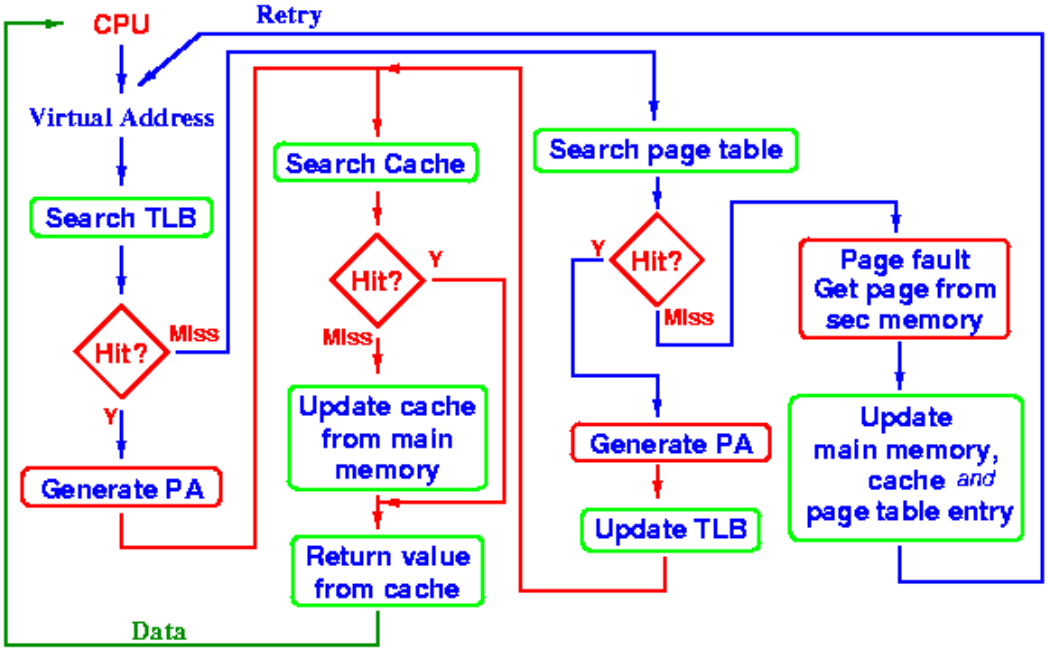


fig. 5.5 Principe de recherche dans un cache

6. Architectures optimisées

6.1. Intel 80386 SX

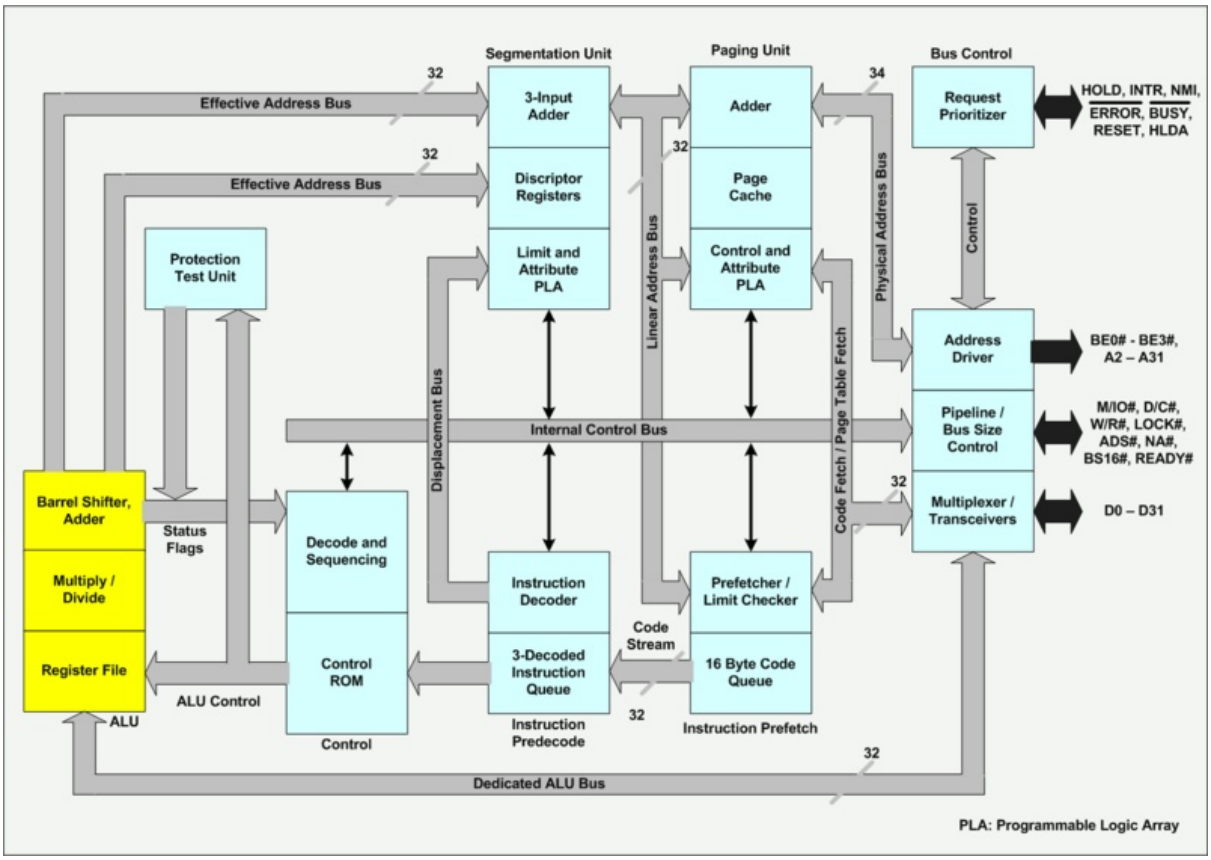


fig. 6.1 Organisation du 386SX de Intel

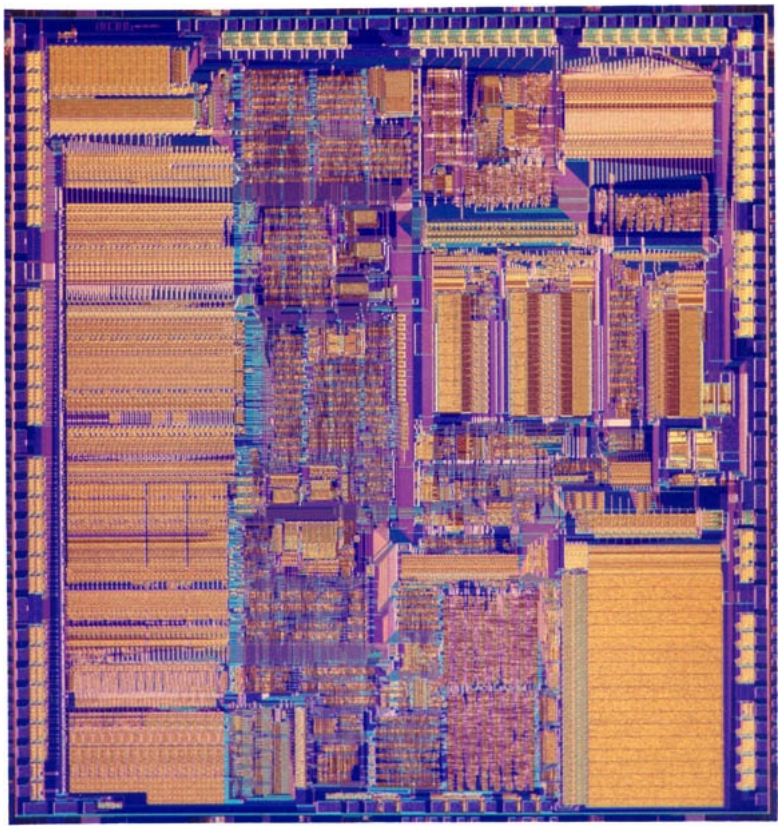


fig. 6.2 Vue interne du 386SX de Intel

6.2. Intel Pentium I

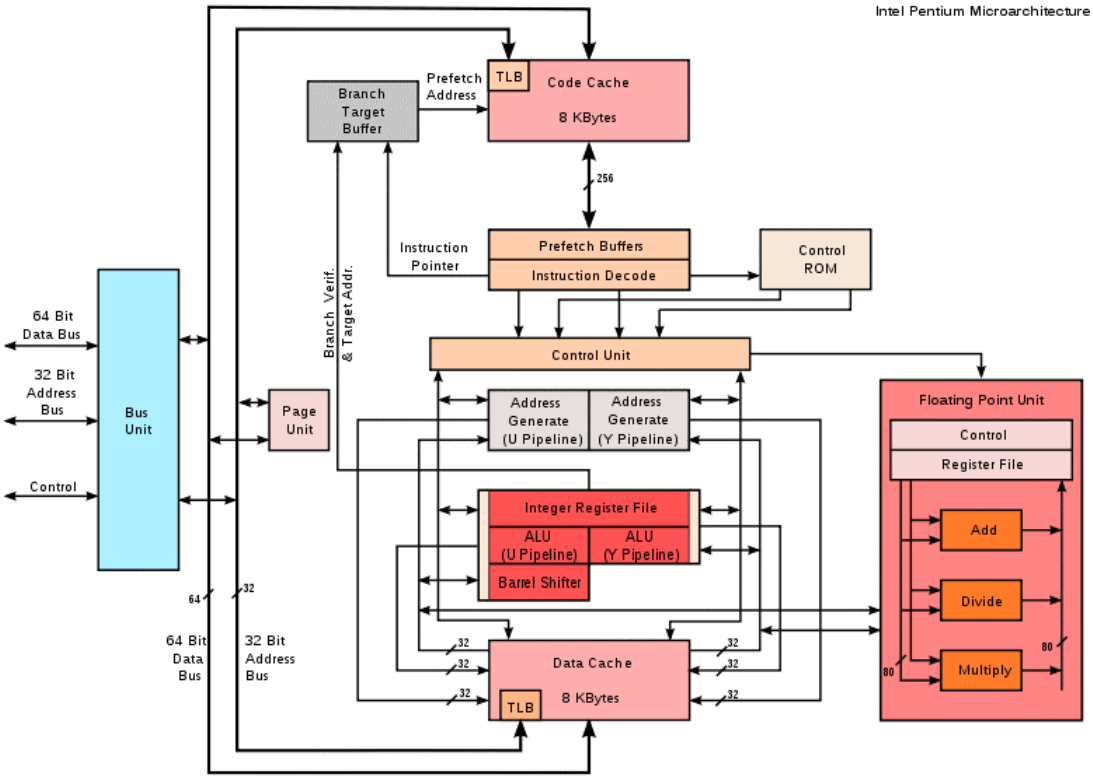


fig. 6.3 Organisation du Pentium I de Intel

- premier processeur à architecture x86
- deux pipelines "integer" (soit deux instructions sur integer par cycle CPU)
- Floating Point Unit améliorée (soit une instruction sur float par cycle CPU)
- bus de données interne à 64 bits.
- deux caches de 8 kB, un pour les instructions et un pour les données
- module de prédiction de branche

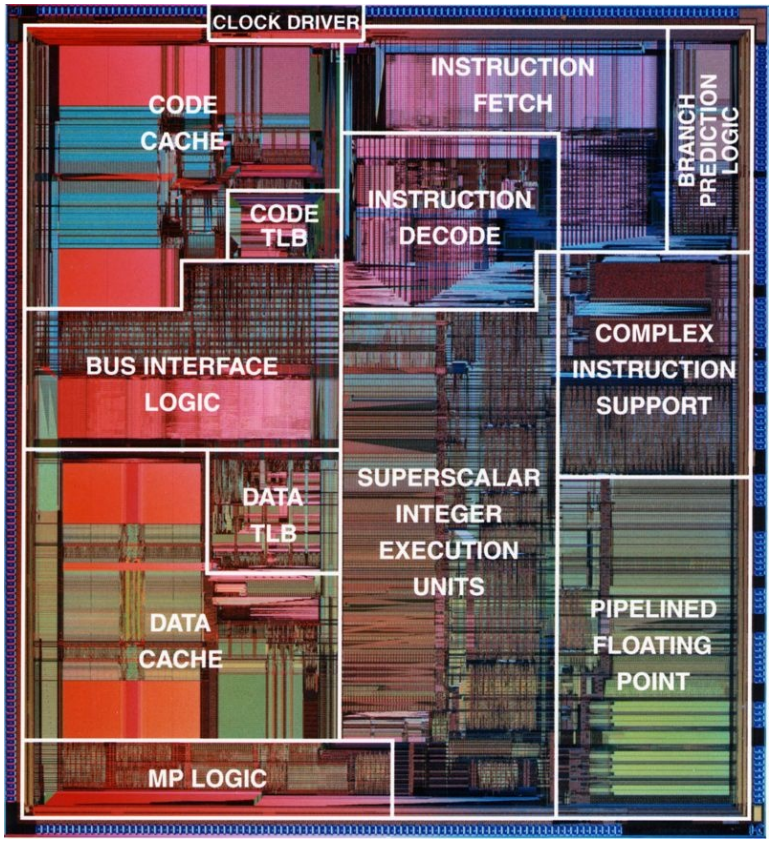


fig. 6.4 Vue interne du Pentium I de Intel

6.3. Intel Pentium III

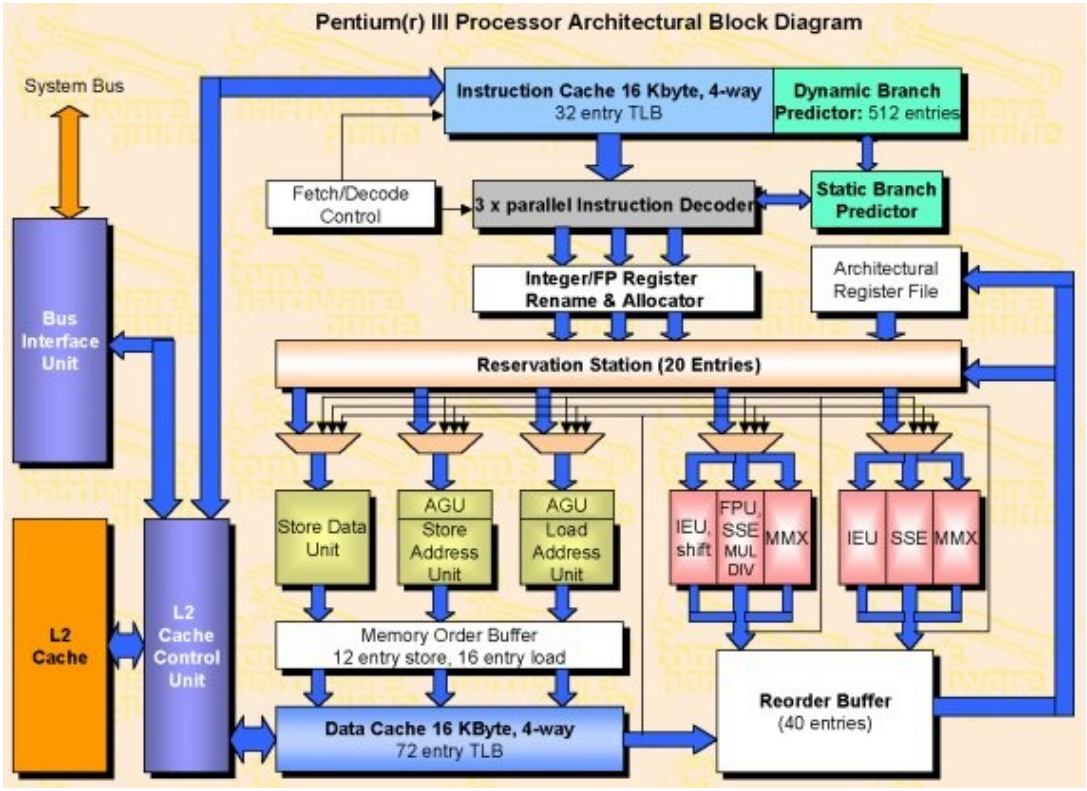


fig. 6.5 Organisation du Pentium III de Intel

- superscalaire out-of-order.
- formats d'instructions variables.
- 3 unités de décodage parallèles :
  1 microprogrammée (si instruction = plus de 4 microcodes), 2 câblées.
- jusqu'à 20 microcodes en attente.
- Plusieurs pipelines mais au mieux 5 instructions exécutées en même temps :
  2 calculs entiers, 1 flottant, 2 accès mémoire load/store.
- Pipeline sur entier : 12 à 17 étages,
  Pipeline sur float : environ 25 étages.
- cache L2 : 512 Ko.

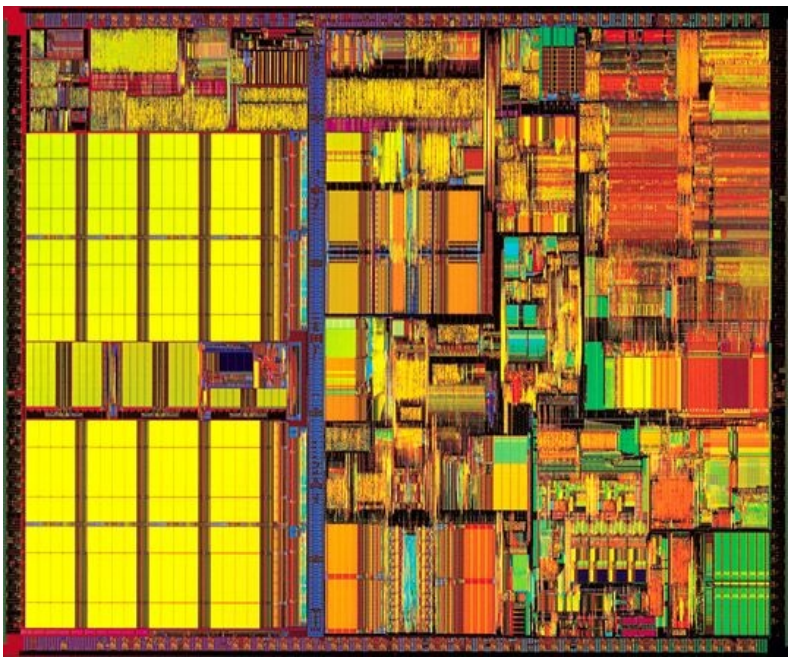
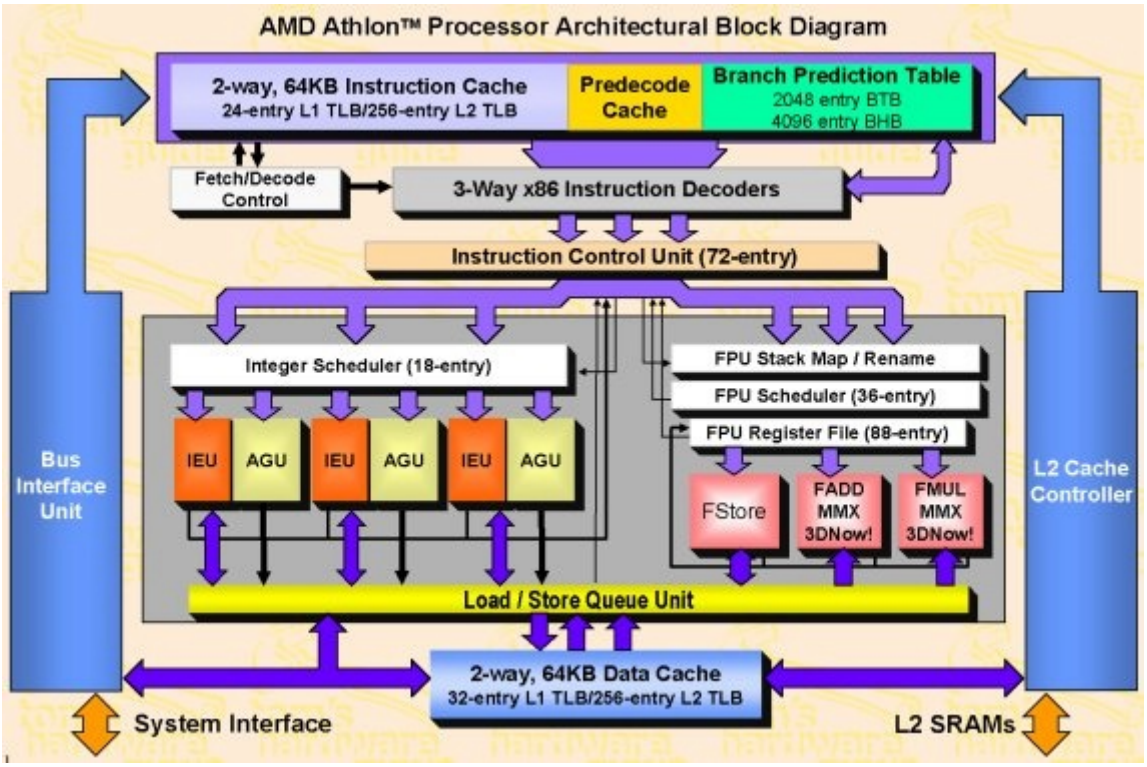


fig. 6.6 Vue interne du Pentium III de Intel

6.4. AMD Athlon K7



AGU : Adress Generation Unit
IEU : Integer Execution Unit
BTB : Branch Target Buffer
BHB : Branch History Buffer

- superscalaire out-of-order.
- format d'instructions variables.
- 6 unités de décodage parallèles (3 microprogrammées, 3 câblées)
  mais seules 3 peuvent fonctionner en même temps.
- jusqu'à 20 microcodes en attente.
- 9 pipelines (3 ALU, 3 FPU + SIMD (MMX), 3 AGU (address generation unit)).
- pipeline entier : 10 étages, pipeline float : 15 étages.
- cache L1 : 64Ko/64Ko, cache L2 : 512 Ko.

(à suivre...)

7. Sources

Intel
Notices techniques des processeurs
www.intel.com

S. Tisserant
Architecture et Technologie des Ordinateurs - 2003
ESIL
http://www.mines.u-nancy.fr/~tisseran/cours/architectures/machines_virtuelles.html

Dan Mepham
Pipelining, pushing the Clockspeed Envelope
http://www.hardwarecentral.com/hardwarecentral/tutorials/2427/2/

John Morris
Computer Architecture - The Anatomy of Modern Processors
http://ciips.ee.uwa.edu.au/~morris/CA406/CA_ToC.html

Eric Garcia
Architectures des ordinateurs Cours 3 – 2002
IUT GTR, Montbéliard

Haibo Wang
Intel 8088 (8086) Microprocessor Structure
ECE Department
Southern Illinois University

Scott Mueller
Upgrading and Repairing PC's
Q.U.E. (16th Edition)
ISBN 0-7897-3210-6
(une source remarquable. A conseiller)