Architecture des ordinateurs

15 juillet 2002

Table des matières

Table des figures

Chapitre 1

Introduction à l'architecture

1.1 Qu'appelle t-on architecture des ordinateurs ?

Définition schématique : l'**architecture** est l'étude et la description du fonctionnement des composants internes d'un ordinateur. Elle traite :

- du type des informations manipulées et de leur codage,
- du dialogue entre composants,
- du fonctionnement logique (pas électronique) interne des composants.

1.2 Vers l'ordinateur

1.2.1 Les inventions

A partir du 17^{e} siècle, certains savants ont commencé à penser que certaines tâches, comme le calcul arithmétique simple (addition, soustraction, ...), **étaient suffisamment mécaniques pour être automatisables**. Ne restait plus qu'à inventer les machines correspondantes.

- Première réalisation, la machine à calculer de PASCAL (1623-1662) était destinée à effectuer mécaniquement des additions et des soustractions. L'automatisation était réalisée à l'aide de roues dentées.
- LEIBNIZ (1646-1716) va plus loin ; il envisage qu'une machine puisse raisonner, c'est-à-dire qu'elle puisse enchaîner des propositions élémentaires pour effectuer des déductions. Pour LEIBNIZ le lien entre W et Z dans la suite « si W alors X ; si X alors Y ; si Y alors Z » s'impose avec un tel degré d'évidence qu'une machine devrait pouvoir le retrouver. Cela a amené LEIBNIZ à imaginer une machine à raisonner qui se calquerait sur une machine à calculer.
  Il a rajouté la multiplication et la division à la machine de PASCAL.
- CHARLES BABBAGE (1792-1871) a construit en 1833 une machine à calculer, *la machine à différences*. Le mécanisme de cette machine reprenait les principes du métier à tisser (FALCON, 1728) commandé par des cartons troués articulés. Ceux-ci, en entrant dans la machine, sont traversés par un système de tringlerie. La machine était non seulement capable d'exécuter isolément les quatre opérations arithmétiques élémentaires mais aussi d'effectuer des séquences de ces opérations, mais dans un but unique : calculer les tables numériques nécessaires à la navigation en mer. Les résultats étaient gravés sur un plateau de cuivre.
  Il a ensuite imaginé *une machine analytique* comportant quatre parties : le magasin (la mémoire), le moulin (l'unité de calcul), l'entrée (le lecteur de cartes perforées) et la sortie (perforation ou impression). Le moulin prenait les opérandes provenant du magasin, en faisait l'addition, la soustraction, la multiplication ou la division et renvoyait le résultat vers le magasin (le tout de manière entièrement mécanique). Il pouvait tester si un nombre était positif, ou effectuer un branchement conditionnel. La machine lisait ses instructions sur cartes perforées. La machine était donc programmable, au moyen d'un langage d'assemblage très simple et ADA AUGUSTA LOVELACE, collaboratrice de BABBAGE et fille de LORD BYRON, a écrit le premier programme informatique. Fautes de crédits et vu la technologie de l'époque, BABBAGE ne put mettre au point sa machine (une reconstitution pesant 3 tonnes existe).

1.2.2 Les théories

- En 1854, un mathématicien anglais, GEORGE BOOLE (1815-1864), publie un essai intitulé *Une étude des lois de la pensée*, et présente une algèbre pour simuler les raisonnements logiques. Son étude reprend celle de LEIBNIZ qui voulait déterminer de manière automatique si un raisonnement est correct. L'algèbre booléenne définit un ensemble à deux éléments contenant *vrai* et *faux* ainsi que des opérations pouvant toutes se ramener à deux opérations de base.
- Les travaux de BOOLE auraient pu rester dans les placards comme le sont tant d'autres, mais dans les années 30, CLAUDE SHANNON (1916-2001) fit le rapprochement entre les nombres binaires, l'algèbre de Boole et les circuits électriques. Il montra que le système binaire, manipulé par les opérations logiques de BOOLE (en prenant 1=*vrai* et 0=*faux*), permettait de réaliser toutes les opérations logiques et arithmétiques.

1.3 Naissance de l'ordinateur

Des machines électromécaniques (utilisant des cartes perforées) réalisant les idées de Babbage virent le jour et, en 1944, un calculateur (le Mark1 d'IBM) pouvait multiplier 2 nombres de 23 chiffres en 6 secondes. Mais l'ère de l'électronique qui suivit immédiatement allait permettre de réduire par 1000 les temps de réponse. En effet, les opérations booléennes peuvent être facilement réalisées dans des circuits électriques en plaçant judicieusement quelques interrupteurs (eux-mêmes contrôlés par des courants électriques). Ces interrupteurs ont d'abord été des tubes à vides puis des transistors. (anecdote tube-bug).

1945 : ENIAC (*Electronic Numerical Integrator And Calculator*) 18 000 tubes, 30 tonnes, multiplie 2 nombres de 10 chiffres en 3 millisecondes. Inconvénients : les données sont sur des cartes perforées mais les programmes sont câblés en mémoire et donc, pour passer d'un programme à un autre, il faut débrancher et rebrancher des centaines de câbles (ce n'est pas vraiment le premier ordinateur au sens actuel).

Fin 1945 : JOHN VON NEUMANN, associé à l'ENIAC, propose un modèle d'ordinateur qui fait abstraction du programme et se lance dans la construction d'un EDVAC (*Electronic Discrete Variable Automatic Computer*) : l'IAS. L'ordinateur est né, c'est la machine de VON NEUMANN. Celui-ci décrit les cinq composants essentiels de se qu'on appellera l'architecture de VON NEUMANN (cf. figure 1.1) :

- l'unité arithmétique et logique (UAL) ;
- l'unité de commande ;
- la mémoire centrale ;
- l'unité d'entrée ;
- l'unité de sortie.

Caractéristiques :

- une machine universelle contrôlée par programme (universelle : contrairement aux autres pas spécialement dédiées aux opérations arithmétiques) ;
- instructions et données, sous format binaire, stockées en mémoire (important : instructions et données dans la même mémoire) ;
- le programme peut modifier ses propres instructions ;
- ruptures de séquence.

1.4 Chronologie

Première génération (1945-1955) : tubes à vide (pannes fréquentes, difficiles à déceler, ordinateurs demandant beaucoup de place), et tores de ferrite pour les mémoires.

Deuxième génération (1955-1965) : remplacement des tubes par des transistors ; organisation de la machine autour d'un bus ; stockage sur bande magnétique ; écrans ; etc.

Troisième génération (1965-1980) : apparition des circuits intégrés (puces) et des processeurs $\Rightarrow$ éloignement du schéma de VON NEUMANN car processeurs d'E/S pour traitement direct avec la mémoire ; miniaturisation (ordinateurs plus petits, plus rapides et moins chers) ; gammes de machines (même langage d'assemblage pour des machines différentes, d'où réutilisabilité des programmes) ; multiprogrammation (plusieurs programmes en

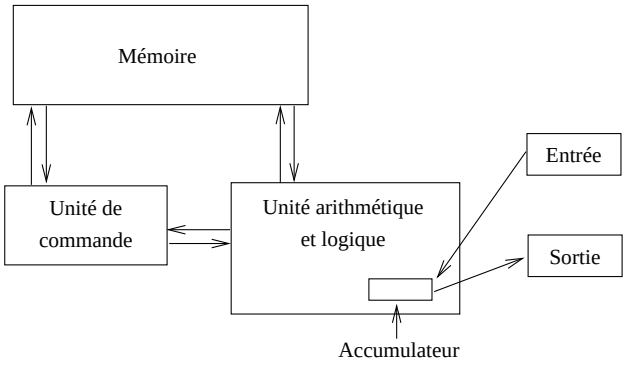


FIG. 1.1 – Schéma de la machine de VON NEUMANN.

mémoire, lorsqu'un programme entre en phase d'entrées-sorties, l'UC passe à l'exécution d'une partie d'un autre programme).

Quatrième génération (1980- ??) : ordinateurs personnels et VLSI (*Very Large Scale Integration*) : augmentation du nombres de transistors ; réseau ; terminaux.

1.5 Structure générale d'un ordinateur mono-processeur actuel

Voir la figure 1.2.

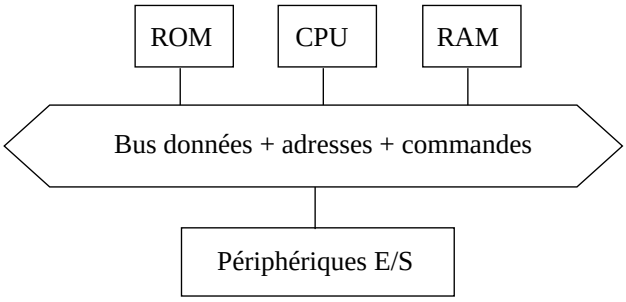


FIG. 1.2 – Structure générale d'un ordinateur mono-processeur actuel.

- CPU : *Central Processing Unit* : élément de l'ordinateur qui interprète et exécute les instructions d'un programme. CPU *n* bits ⇒ les informations sont codées sur *n* bits.
- ROM : *Read Only Memory* : mémoire « morte » que l'on ne peut que lire.
- RAM : *Random Access Memory* : mémoire vive (lecture et écriture) et volatile (se vide quand il n'y a plus de courant) ; c'est (généralement) une suite de cases contenant 8 bits (soit un **octet**), chaque case ayant sa propre adresse.
- Périphériques : disque dur (E/S), lecteur disquette (E/S), clavier (E), souris (E), écran (S), imprimante (S), lecteur cdrom (E), graveur (S), accès réseau (E/S).

CPU

Exemple d'évolution de processeurs : Intel, voir la figure 1.3.

Année	Processeur	Fréquence d'horloge (M Hz)	Puissance (MIPS)	Largeur registre (bits)	Largeur bus de données	Espace d'adres--sage	Nombre de transistors	Remarques
1971	4004	0,108	0,06	4	4	1 K	2 300	1er microproc. (puissance de l'ENIAC)
1972	8008	0,200		8	8	16 K	3 500	
1978	8086	4,77–10	0,75	16	16	1 M	29 000	
1978	8088	4,77–8	0,75	16	8	1 M	29 000	*in* IBM-PC ⇒ standard
1982	80286	6–12	1,5	16	16	16 M	134 000	
1986	80386	16–33	5	32	32	4 G	275 000	1er mode protégé Intel (permet le multitâche)
1989	80486	16–66	27	32	32	4 G	1 200 000	
1993	pentium	60–200	> 100	32	64	64 G	3 300 000	
1997	pentium II	233–450		32	64	64 G	7 500 000	
1999	pentium III	450–600		32	64	64 G	9 500 000	
2001	pentium 4	1300–2200		32	64	64 G	42 000 000	
2001	itanium	800– ??		64	64			

FIG. 1.3 – Évolution des processeurs Intel.

1.6 Langage

- L'activité d'une machine est rythmée par le CPU qui exécute le cycle suivant : chercher une instruction en mémoire (RAM), l'exécuter, chercher l'instruction suivante en mémoire, etc.
- Une instruction est une suite de bits contenant le code de l'opération (addition, soustraction, recherche d'un mot en mémoire, etc.) et ses opérandes (arguments de l'opération).
- Pour éviter d'avoir à connaître ce langage difficilement manipulable par l'homme, on définit des langages de plus haut niveau d'où une architecture en couches de machines virtuelles (cf. figure 1.4).

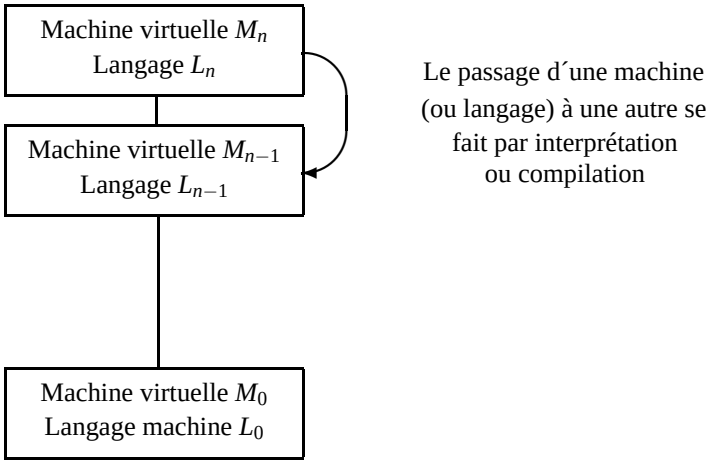


FIG. 1.4 – Architecture d'une machine multi-niveaux.

Interprétation : chaque instruction du langage L_n est traduite en la séquence adéquate du langage L_{n-1} exécutable, donc, par la machine virtuelle M_{n-1}. À chaque exécution il y a ré-interprétation.

Compilation : un programme de la machine M_n, en entier est traduit une fois pour toute en un programme en langage L_{n-1} exécutable directement par la machine virtuelle M_{n-1}.

Les machines actuelles présentes généralement l'architecture en couche de la figure 1.5.

Niveau 5	**Langages d'application**		Pascal, C, Cobol, Basic, Prolog, Caml, Lisp, Java
	↓ ↓	interprétation ou compilation	
Niveau 4	**Langage d'assemblage**		Instructions assembleur, notion de variables,
	↓	traduction assembleur	de piles d'exécution, de routines
Niveau 3	**Système d'exploitation**		Routines programmées
	↓	interprétation	
Niveau 2	**Couche machine**		Langage machine codé en binaire
	↓ ↓	interprétation (microprogramme)	
Niveau 1	**Couche microprogrammée**		Langage interne du microprocesseur
	↓ ↓	exécution directe par le matériel	
Niveau 0	**Couche physique**		

FIG. 1.5 – Les couches de la plupart des ordinateurs actuels.

Chapitre 2

Codage des informations

2.1 Numération en base *b*

2.1.1 Représentation

- La numération arabe utilise une représentation positionnelle contrairement à la numération romaine : le rang de chaque chiffre indique son poids.
  Numération romaine : `MCMXCIX` où `M` vaut toujours 1000, `C` vaut toujours 100, etc.
  Numération positionnelle : 1999 où le « 9 » le plus à droite vaut « 9 », celui immédiatement à sa gauche vaut « 90 », etc. La valeur d'un chiffre dépend de sa position.
- XXX_b indique que le nombre XXX est écrit en base *b*.
  Exemples :

$$\begin{array}{cccc} 10^3 & 10^2 & 10^1 & 10^0 & \\ 5 & 9 & 3 & 1_{10} & = 5\times 1000 + 9\times 100 + 3\times 10 + 1\times 1 \end{array}$$

$$\begin{array}{cccc} 2^3 & 2^2 & 2^1 & 2^0 & \\ 1 & 0 & 0 & 1_2 & = 1\times 8 + 0\times 4 + 0\times 2 + 1\times 1 = 9_{10} \end{array}$$

$$\begin{array}{ccccc} & 5^2 & 5^1 & 5^0 & \\ & 1 & 4 & 3_5 & = 1\times 25 + 4\times 5 + 3 = 48_{10} \end{array}$$

- Bases usuelles : base 10 et base 60.
- Représentation des nombres dans une base *b* :
  1. Si $b <= 10$, on utilise simplement les chiffres de 0 à $b-1$ (exemple : base 2).
  2. Si $b > 10$, on a deux possibilités :
     (a) Soit on utilise les chiffres de 0 à 9 et des lettres.
     Ainsi, en base 16 (numération hexadécimale) on utilise les chiffres de 0 à 9 et les lettres de A (=10) à F (=15).
     $$\texttt{E2A1}_{16} = 14\times 16^3 + 2\times 16^2 + 10\times 16 + 1 = 58017_{10}$$
     Exemple : l'adresse mémoire « Ox0090602f ».
     (b) Soit on se contente d'utiliser les chiffres, et on introduit des symboles séparateurs.
     Ainsi, la longitude de Paris est : $2°20'14''E$ (système sexagésimal).

2.1.2 Opérations arithmétiques

Les principes sont les mêmes que ceux utilisés en base 10 (cf. figure 2.1).
Exemples de calculs en base 2 (172 + 101 = 273 et $5 \times 5 = 25$) :

+	0	1
0	0	1
1	1	10

×	0	1
0	0	0
1	0	1

FIG. 2.1 – Tables d'addition et de multiplication en base 2.

	1	0	1	0	1	1	0	0
+		1	1	0	0	1	0	1
1	0	0	0	1	0	0	0	1

		1	0	1
×		1	0	1
		1	0	1
	0	0	0	
1	0	1		
1	1	0	0	1

2.1.3 Conversion

Binaire → décimal : trivial voir ci-dessus
Décimal → binaire : on procède par division entière successive par 2. Exemple : 29_{10} :

$$\begin{array}{rcll} 29 : 2 = & 14 & \text{reste } 1 & \\ 14 : 2 = & 7 & \text{reste } 0 & \\ 7 : 2 = & 3 & \text{reste } 1 & \Rightarrow \quad 29_{10} = 11101_2. \\ 3 : 2 = & 1 & \text{reste } 1 & \\ 1 : 2 = & 0 & \text{reste } 1 & \end{array}$$

Principe général de la conversion de nombres entiers : soit $X_{b_1} = Y_{b_2}$

$$X = x_n...x_0 = x_n \times b_1^n + ... + x_0 b_1^0 \text{ avec } x_n,...,x_0 < b_1,$$

$$Y = y_m...y_0 = y_m \times b_2^m + ... + y_0 b_2^0 \text{ avec } y_m,...,y_0 < b_2.$$

On a X_{b_1} et on cherche Y_{b_2}, sa représentation en base b_2.
On a $X = y_0 + b_2 \times (y_1 + y_2 b_2 + ... + y_n b_2^{m-1}) = y_0 + b_2 \times Q_0$
– y_0 = reste de la division entière de X par b_2. Le calcul est effectué dans la base b_1 et le reste est donc obtenu dans la base b_1 et doit être converti dans la base b_2 : on tabule les b_1 valeurs nécessaires (de 0 à $b_1 - 1$). Q_0 est le quotient de cette division. $Q_0 = y_1 + b_2 \times (y_2 + ... + y_n b_2^{m-2})$.
– y_1 = reste de la division entière base de Q_0 par b_2, etc.
– On réitère le procédé jusqu'à ce que le quotient soit nul.
Exemple : $14_5 = 9_{10} = 12_7$:
– Conversion de la base 5 vers la base 7 : division entière par $7 = 12_5$ en base 5.
$14_5 : 12_5 = 1$ reste 2
$1_5 : 12_5 = 0$ reste 1
Le codage de 14_5 est donc 12_7.
– Conversion de la base 7 vers la base 5 : division entière par $5 = 5_7$ en base 7.

×	0_7	1_7	2_7	3_7	4_7	5_7	6_7	10_7
5_7	0_7	5_7	13_7	21_7	26_7	34_7	42_7	50_7

TAB. 2.1 – Table de multiplication par 5 en base 7.

$12_7 : 5_7 = 1$ reste 4
$1_7 : 5_7 = 0$ reste 1
Le codage de 12_7 est donc 14_5.

– Exemple plus compliqué : conversion de 1452_7 en base 5...

$$\begin{array}{rcccc|l} & 1 & 4 & 5 & 2_7 & 5_7 \\ - & 1 & 3 & & & \hline \\ \cline{2-4} & & 1 & 5 & & \\ & - & 1 & 3 & & \\ \cline{3-5} & & & 2 & 2 & \\ & & - & 2 & 1 & \\ \cline{4-5} & & & & 1 & \end{array} \qquad \begin{array}{rcccc|l} & 2 & 2 & 3_7 & & 5_7 \\ - & 2 & 1 & & & \hline \\ \cline{2-4} & & 1 & 3 & & \\ & - & 1 & 3 & & \\ \cline{3-5} & & & 0 & & \end{array} \qquad \begin{array}{rcc|l} & 3 & 2_7 & 5_7 \\ - & 2 & 6 & \hline \\ \cline{2-3} & & 3 & \end{array}$$

D'où $1452_7 = 4301_5$. Les sceptiques peuvent vérifier que $1452_7 = 1\times7^3+4\times7^2+5\times7+2 = 576$ et $4301_5 = 4\times5^3+3\times5^2+0\times5+1 = 576$...

En pratique, si $b_1 \neq 10$, on évite d'effectuer la division en base b_1 en convertissant X en base 10 puis en procédant par division sur le nombre obtenu pour effectuer la conversion de la base 10 vers la base b_2.

Certaines conversions sont très faciles à réaliser comme la conversion binaire vers octal (base 8) ou binaire vers hexadécimal (base 16).

Exemple : 1010011101_2

– base 8 découpage par blocs de 3 chiffres :

$$\begin{array}{cccc} 001 & 010 & 011 & 101 \\ 1 & 2 & 3 & 5 \end{array} = 1235_8$$

– base 16 découpage par blocs de 4 chiffres :

$$\begin{array}{ccc} 0010 & 1001 & 1101 \\ 2 & 9 & \texttt{D} \end{array} = 29\texttt{D}_{16}$$

De manière générale, les conversions sont faciles lorsque b_2 est une puissance de b_1. Les conversions en base 8 ou 16 sont très fréquentes pour l'affichage des nombres binaires qui, par leurs longueurs, sont rapidement illisibles. Exemple : octal pour les codes de caractères, hexadécimal pour les adresses mémoires.

2.2 Représentation des entiers

2.2.1 Codage machine

– Codage en binaire, ici un chiffre est appelé un bit (***b**inary dig**it*** : chiffre binaire).
– Les nombres sont codés sur n octets, généralement 2 (*short* en C) ou 4 (*int* ou *long* en C).
– m bits $\Rightarrow 2^m$ nombres différents, si $n = 2 \Rightarrow 2^{16} = 65\,536$ nombres, si $n = 4 \Rightarrow 2^{32} = 4294\,967\,296$ nombres.
– Problème : capacité limitée. Si le résultat de l'opération est supérieur au nombre maximum représentable $\Rightarrow$ *overflow* (dépassement de capacité).
Exemple sur 4 bits : 9 + 7 = 16 qui ne peut être stocké sur 4 bits...

$$\begin{array}{rcccc} & 1 & 0 & 0 & 1 \\ + & & 1 & 1 & 1 \\ \hline & 0 & 0 & 0 & 0 \end{array}$$

Les jeux d'instructions processeurs proposent souvent deux types d'opérations : celles ignorant l'*overflow* et celles dont un *overflow* provoque l'exécution d'une routine particulière. Dans un langage comme le C, l'*overflow* est ignoré, en revanche le langage ADA fait appel aux instructions qui en tiennent compte.
– Problème : comment représenter les entiers négatifs ? On peut introduire un bit de signe : 0 pour « + » et 1 pour « - ». Ainsi sur 4 bits : $0110 = 6$ et $1110 = -6$.
  – Sur n bits, on code les entiers de l'intervalle $[-(2^{n-1}-1), 2^{n-1}-1]$: on utilise 1 bit pour le signe et $n-1$ bits pour la valeur, or avec $n-1$ bits on peut stocker 2^{n-1} valeurs.
  – Il existe deux zéros : 00..00 et 10..00.
  – Le traitement de l'addition est compliqué (alors que c'est l'opération la plus fréquente après le chargement d'un mot de la mémoire vers le processeur) car elle nécessite l'extraction du signe, puis un test pour orienter les entiers soit vers un circuit d'addition soit de soustraction et, finalement, il faut calculer le nouveau signe.

Pour simplifier l'addition, une solution consiste à représenter les entiers binaires en compléments à deux. Pour mieux appréhender cette technique, examinons son principe sur les nombres en base 10.

2.2.2 Complément décimal

Soit A un nombre décimal. Le complément à 9 de $A = \sum_{i=0}^{n-1} a_i 10^i$ est le nombre obtenu en soustrayant de 9 chaque chiffre de A : $C_9(A) = \sum_{i=0}^{n-1}(9-a_i)10^i$ et le complément à 10 de A est le complément à 9 de A auquel on ajoute 1 : $C_{10}(A) = 1 + C_9(A) = 1 + \sum_{i=0}^{n-1}(9-a_i)10^i$. Exemple :

	123	9672
Complément à 9	876	0327
Complément à 10	877	0328

On remarque que le complément du complément de A est égal à A :

- $C_9(C_9(A)) = \sum_{i=0}^{n-1}(9-(9-a_i))10^i = \sum_{i=0}^{n-1} a_i 10^i = A$.
- $C_{10}(C_{10}(A)) = \sum_{i=1}^{n-1}(9-(9-a_i))10^i + 1 + (9-(1+9-a_0)) = \sum_{i=0}^{n-1} a_i 10^i = A$.
- Notons $C_{10}(A) = \sum_{i=0}^{n-1} b_i 10^i$. $C_{10}(C_{10}(A)) = \sum_{i=0}^{n-1}(9-b_i)10^i + 1 = 1 + \sum_{i=0}^{n-1} 9.10^i - \sum_{i=0}^{n-1} b_i 10^i = 1 + \sum_{i=0}^{n-1} 9.10^i - \left(1 + \sum_{i=0}^{n-1}(9-a_i)10^i\right) = 1 + \sum_{i=0}^{n-1} 9.10^i - 1 - \sum_{i=0}^{n-1} 9.10^i + \sum_{i=0}^{n-1} a_i 10^i = \sum_{i=0}^{n-1} a_i 10^i$.

Lien avec la soustraction

Soient A et B deux nombres de 2 chiffres. Nous voulons calculer $C = B - A$. C peut être écrit sous la forme : $C = B - A + (99 + 1 - 100) = B + [(99 - A) + 1] - 100 = B + C_{10}(A) - 100$.

Ainsi, calculer C revient à additionner B et le complément de A à 10, puis à supprimer 100. Deux cas sont possibles :

- $B > A$. Exemple : $B = 23$ et $A = 12$. $C_{10}(A) = 88$.

$$\begin{array}{cccc} & & 2 & 3 \\ + & & & \\ & & 8 & 8 \\ = & & & \\ & 1 & 1 & 1 \end{array}$$

  Comme $B > A, 99 + (B - A) + 1 > 100$, ainsi en éliminant simplement le 1 en tête, on trouve $B - A$. Notons qu'avec une représentation machine sur deux bits, le 1 de tête est perdu automatiquement.
- $A < B$. Exemple : $B = 12$ et $A = 23$. $C_{10}(A) = 77$.

$$\begin{array}{cccc} & & 1 & 2 \\ + & & & \\ & & 7 & 7 \\ = & & & \\ & 0 & 8 & 9 \end{array}$$

  Comme $B < A, 99 + (B - A) + 1 < 100$, si on fait la soustraction par 100, on retrouve le résultat $(89 - 100 = -11)$, mais le résultat tel quel est le complément à 10 de la valeur absolue du résultat (complément de 11 ; $C_{10}(11) = 89$).

Conclusion : en codant les entiers avec un nombre fixe de chiffres, coder les entiers négatifs en complément à 10 permet de ramener les soustractions en addition.

Il faut toutefois faire attention à séparer les nombres négatifs des autres nombres. Ainsi, dans l'exemple précédent, les nombres [0 ; 49] sont positifs et [50 ; 99] négatifs.

2.2.3 Complément binaire et codage des entiers négatifs

- En binaire, le principe du complément est le même, le complément à 1 revient simplement à inverser les bits d'un nombre et le complément à 2 ajoute 1 au complément à 1.

Nous considérons des nombres binaires stockées sur n bits. $A = \sum_{i=0}^{n-1} a_i 2^i$.

$$C_1(A) = \sum_{i=0}^{n-1} (1 - a_i) 2^i.$$

$$C_2(A) = 1 + \sum_{i=0}^{n-1} (1 - a_i) 2^i = 1 + \sum_{i=0}^{n-1} 2^i - \sum_{i=0}^{n-1} a_i = 1 + (2^n - 1) - A = 2^n - A$$

D'où :

$$B + C_2(A) = B - A + 2^n$$

- Le premier bit d'un nombre *indique* son signe, s'il est négatif son complément permet de retrouver sa valeur absolue.
  Exemple pour des entiers codés sur 3 bits :

000 : positif, donc 0		
	100 : complément à 1 : 011 donc	-3
001 : positif, donc 1	complément à 2 : 100 donc	-4
	101 : complément à 1 : 010 donc	-2
010 : positif, donc 2	complément à 2 : 011 donc	-3
	110 : complément à 1 : 001 donc	-1
011 : positif, donc 3	complément à 2 : 010 donc	-2
	111 : complément à 1 : 000 donc	0
	complément à 2 : 001 donc	-1

- En machine les nombres négatifs sont directement stockés en complément (généralement en complément à 2). Une soustraction demande simplement de complémenter le deuxième opérande et d'effectuer une addition.
- L'utilisation du complément permet :
  - de n'utiliser pour l'addition et la soustraction que le seul circuit réalisant l'addition ;
  - de ne pas avoir à considérer le signe des nombres.
- Complément à 1 : complémentation simple, ajout d'un 1 au résultat de l'addition, deux zéros, tout nombre à son opposé.
  Complément à 2 : complémentation avec ajout de 1, un seul zéro, le plus petit nombre n'a pas d'opposé.
- Si le bit d'*overflow* est différent du bit de signe $\Rightarrow$ dépassement de capacité. Exemple sur 3 bits avec 3 + 2 et en complément à 2 avec -3 -2.
- Endianisme : manière de ranger les octets en mémoire. Prenons le nombre $26\,548_{10} = 67\texttt{B}4_{16}$. On distingue les architextures *little-endian* (octet de poids faible en tête) B467 et les architectures *big-endian* (octet de poids fort en tête) 67B4. (Intel : *little endian* ; Motorola, Sun : *big endian*).

2.3 Représentation des nombres fractionnaires

En machine, tout nombre est codé avec un nombre fini de chiffres, ainsi, il n'est pas possible de représenter tous les rationnels, et *a fortiori* tous les réels. Pour manipuler des nombres non entiers, on utilise une représentation prenant en compte la virgule dans les nombres ce qui permet de considérer certains nombres fractionnaires.

2.3.1 Changement de base

- binaire $\rightarrow$ décimal : $0.011_2 = 0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3} = 0.375_{10}$.
- décimal $\rightarrow$ binaire :
  - $0.5625 \times 2 = 1.125 = 1 + 0.125$
  - $0.125 \times 2 = 0.250 = 0 + 0.250$
  - $0.250 \times 2 = 0.5 = 0 + 0.5$
  - $0.5 \times 2 = 1.0 = 1 + 0 \Rightarrow 0.1001_2$

2.3.2 Virgule flottante

- Les premières machines utilisaient une représentation en virgule fixe où chaque nombre était séparé en deux parties contenant les chiffres avant et après la virgule. Depuis le début des années 60, la représentation en virgule flottante, plus souple, s'est imposée.
- La représentation en virgule flottante consiste à représenter les nombres sous la forme suivante : $M \times B^E$ où M est la mantisse, B une base (2, 8, 10, 16, ...) et E l'exposant.
  Exemples : $123 \times 10^3 = 123\ 000$, $-0.02 \times 10^{-1} = -0.002$, $1.011 \times 2^3{}_2 = 1011_2$.
- Un flottant est stocké de la manière suivante (norme IEEE 754) :

SM	Eb	M

  - SM : signe de la mantisse : 1 bit ;
  - Eb : exposant biaisé : 8 bits en simple précision et 11 en double précision ;
  - M : mantisse : 23 bits en simple précision et 52 en double.
- L'exposant est placé avant la mantisse pour simplifier le tri des nombres en virgule flottante (comparaison). Pour cela, l'exposant ne doit pas être représenté en complément à deux sinon 2^{-1} (111..1) serait supérieur à 2 (0..010).
- L'exposant est sans signe mais biaisé de 127. Ainsi :
  - Eb = 0 $\Rightarrow$ E = 0-127 = -127
  - Eb = 255 $\Rightarrow$ E = 255-127 = 128
  - Eb = 127 $\Rightarrow$ E = 0.
- L'exposant maximum est 127 et non 128 qui est réservé pour des configurations spéciales :
  0 11111111 00000...000 = +INF
  1 11111111 00000...000 = -INF
  Si 1/0 par exemple ou encore lors d'un *overflow*.
  Si la mantisse est différente de 0, alors le *float* est NaN (*Not a Number*). Cela se traduit lors d'opérations sans signification : racine avec un négatif, $+\text{INF} + -\text{INF}$, division invalide (0/0), multiplication invalide ($0 \times$ INF).
- La mantisse est *normalisée* au sens où elle est toujours de la forme 0.1.... Une mantisse est dite normalisée si son chiffre le plus à gauche (de poids fort) est un 1 (en codage binaire). Exemple : 1.001 est normalisée. Contre-exemple : 0.001 n'est pas normalisée.
  Lorsque l'exposant est -127, la mantisse n'est pas normalisé.
- Mettre le bit de signe en tête permet d'utiliser les mêmes circuits que pour les entiers pour tester le signe.

2.3.3 Opérations sur les nombres à virgule flottante

- Pour l'addition et la soustraction, il faut que les exposants aient la même valeur.
  Exemple d'addition : $0.3 \times 10^4 + 0.998 \times 10^6 =$
  1. Dénormaliser : $0.3 \times 10^4 = 0.003 \times 10^6$.
  2. Additionner les mantisses : $0.003 + 0.998 = 1.001$.
  3. Normaliser : $1.001 \times 10^6 = 0.1001 \times 10^7$.
- Pour la multiplication (resp. la division), on additionne (resp. soustrait) les exposants et on multiplie (resp. divise) les mantisses.
  Exemple : $(0.2 \times 10^{-3}) \times (0.3 \times 10^7)$
  1. Addition des exposants : $-3 + 7 = 4$.
  2. Multiplication des mantisses : $0.2 \times 0.3 = 0.06$.
  3. Normalisation du résultat : $0.06 \times 10^4 = 0.6 \times 10^3$.

2.3.4 Problème d'arrondis

Considérons la base 10 avec une représentation de la forme S MMM s ee, avec une mantisse normalisée entre 0.1 et 1 exclusivement. Ainsi, on peut représenter certains nombres de l'intervalle $[-0.999 \times 10^{-99}, 0.999 \times 10^{99}]$.

- La mantisse prend les valeurs entre 100 et 999 $\Rightarrow$ 900 valeurs, l'exposant prend les valeurs de -99 à +99 $\Rightarrow$ 199 valeurs, on ajoute le zéro et le fait que la mantisse soit signé $\Rightarrow 900 \times 199 \times 2 + 1 = 358\ 201$ valeurs possibles.
- Si une opération fournit un résultat dans la zone (cf. figure 2.2) :

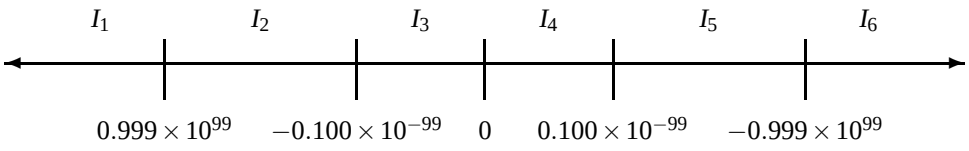


FIG. 2.2 – Représentation des réels et problèmes d'arrondis.

  - I1 ou I6 : il y a dépassement de capacité (*overflow*) ;
  - I2 ou I5 : il sera représentable mais avec éventuellement une perte de précision ;
  - I3 ou I4 : il ne pourra être représenté et sera approximé par zéro (*underflow*).
- En augmentant le nombre de chiffres de la mantisse, on rend les intervalles plus denses. En augmentant le nombre de chiffres de l'exposant, on élargit les zones I2 et I5 et on diminue les autres.
- La distance entre deux nombres représentables adjacents n'est pas toujours la même et augmente pour les grands nombres. Exemple :
  - le nombre représentable en machine le plus proche de 0.100×10^{-99} est 0.101×10^{-99}, distant de 1×10^{-102}.
  - le nombre représentable en machine le plus proche de 0.999×10^{99} est 0.998×10^{99}, distant de 1×10^{96}.

Remarque. Au niveau logiciel on trouve des librairies qui permettent le calcul (sur les entiers, les rationnels et les réels) en précision infinie, comme la librairie `gmp` de GNU (*GNU Multi Precision*).

2.4 Représentation des caractères

De nombreux codes ont été utilisés pour représenter les caractères (codes BAUDOT sur 5 bits : insuffisants pour coder chiffres et lettres, EBCDIC sur 8 bits (IBM)). Un code normalisé, le code ASCII (*American Standard, Code for Information Interchange*) s'est peu à peu imposé en raison de son utilisation dans les micro-ordinateurs.

Le code ASCII représente les caractères sur 7 bits. Les 32 premiers (00-1F) représentent des caractères de contrôle utilisés soit dans des protocoles d'échange d'informations (ENQ, ACK, NAK, ...), soit pour le contrôle de certains terminaux (LF : Line Feed, CR : Carriage Return, BS : BackSpace, ...).

Les codes suivants (20-7F), sont utilisés pour coder les caractères alphabétiques (majuscules ou minuscules) et les caractères de ponctuation. $A = 41_{16} = 65_{10}$, $a = 61_{16} = 97_{10}$, or les caractères sont consécutifs dans le code $\Rightarrow$ pour passer des majuscules aux minuscules on ajoute 32 (on force le 5^{e} bit à 1).

En machine, on accède aux mots octets par octets $\Rightarrow$ un caractère utilise toujours 8 bits. Pour tirer partie des possibilités supplémentaires on a étendu le code ASCII à 8 bits ce qui permet de représenter certains caractères accentués de certains pays européens ou l'alphabet grec (d'où des problèmes de compatibilité potentiels et la nécessité d'expliciter le codage utilisé, comme le *iso-latin-1*).

Chapitre 3

Algèbre de Boole et circuits logiques

3.1 Introduction à la logique des propositions

– Une proposition est un énoncé vrai ou faux. Exemple : « il fait beau ». Contre-exemple : la phrase « mais qui êtes-vous ? » n'est pas une proposition.
– Les propositions peuvent être combinées par des **connecteurs** pour en former de nouvelles. Les connecteurs les plus courants sont : $\vee$ (*ou* logique), $\wedge$ (*et* logique) et $\neg$ (*non* logique). Les valeurs de ces propositions sont données par les tables de vérités de la table 3.1. On remarquera que le « ou » du langage courant à plutôt l'acception d'un « ou exclusif » alors qu'ici le « ou » logique correspond au « et/ou ».

p	$\neg p$
V	F
F	V

p	q	$p \vee q$
V	V	V
V	F	V
F	V	V
F	F	F

p	q	$p \wedge q$
V	V	V
V	F	F
F	V	F
F	F	F

TAB. 3.1 – Tables de vérités des connecteurs logiques $\neg$, $\vee$ et $\wedge$.

– La logique des propositions comporte d'autres connecteurs comme « si a alors b » et quelques lois comme $a \vee a = a$, $\neg(a \vee b) = \neg a \wedge \neg b$ (voir la démonstration sur la table 3.2). GEORGE BOOLE a exprimé la logique des propositions en termes algébriques.

a	b	$\neg a$	$\neg b$	$\neg \mathbf{a} \wedge \neg \mathbf{b}$	$a \vee b$	$\neg(\mathbf{a} \vee \mathbf{b})$
V	V	F	F	**F**	V	**F**
V	F	F	V	**F**	V	**F**
F	V	V	F	**F**	V	**F**
F	F	V	V	**V**	F	**V**

TAB. 3.2 – Loi élémentaire de logique : $\neg a \wedge \neg b = \neg(a \vee b)$.

3.2 Algèbre de Boole

– GEORGE BOOLE a structuré algébriquement la logique des propositions. CLAUDE SHANNON a proposé le codage : V = 0, F = 1 ; $a \vee b$ devient alors $a + b$, $a \wedge b$ devient $a \times b$ ou encore ab, et $\neg a$ devient $\overline{a} = 1 - a$.
– On définit donc une structure algébrique sur un ensemble fini, on obtient donc un nombre fini de fonctions à n variables.

– Tables de vérités des fonctions à une variable :

a	f_1	f_2	f_3	f_4
0	0	0	1	1
1	0	1	0	1

Seule une fonction est intéressante du point de vue logique, c'est la fonction f_3 qui est le *non* logique $f_3(a) = \overline{a}$ (les autres fonctions sont $f_1(a) = 0$, $f_2(a) = a$,et $f_4(a) = 1$).

– Tables de vérités des fonctions à deux variables : voir la table 3.3.

a	b	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}	f_{11}	f_{12}	f_{13}	f_{14}	f_{15}	f_{16}
0	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	0	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

TAB. 3.3 – Table des vérités des fonctions à deux variables.

Fonctions remarquables :

$$f_1(a,b) = 0$$
$$f_{16}(a,b) = 1$$
$$f_9(a,b) = ab : \textit{ et}$$
$$f_{15}(a,b) = a+b : \textit{ ou}$$
$$f_7(a,b) = a \oplus b : \textit{ ou exclusif} \text{ ou } \textit{xor}$$
$$f_2 : \textit{nor} \text{ (non ou)}$$
$$f_8 : \textit{nand} \text{ (non et)}$$

On peut montrer que toutes ces fonctions peuvent s'exprimer en fonction des trois opérations logiques *et*, *ou* et *non* :

$$f_1(a,b) = 0$$
$$f_2(a,b) = \overline{a+b}$$
$$f_3(a,b) = \overline{a}b$$
$$\ldots$$
$$f_7(a,b) = \overline{a}b + a\overline{b}$$

– Ainsi avec n variables, on construit 2^{2^n} fonctions (il y a 2^n configurations possibles des entrées, et pour chacune de ces configurations on a deux choix possibles).

– Une fonction à trois variables peut se décomposer en deux : $f(a,b,c) = f(0,b,c)$ si $a = 0$ (où $f(0,b,c)$ est une des fonctions du tableau 3.3) et $f(a,b,c) = f(1,b,c)$ si $a = 1$. Ainsi $f(a,b,c) = a.f(1,b,c) + \overline{a}.f(0,b,c)$. On montre ainsi par récurrence que toute fonction à n variables se ramène à une expression contenant des fonctions à deux variables.

– On appelle *groupe logique complet* un ensemble de fonctions logiques à partir desquelles il est possible de réaliser toutes les fonctions logiques. Ex : {*et*, *ou*, *non*}, {*et*, *non*}, {*ou*, *non*}, {*nand*}, {*nor*}, {*xor*, *et*}, etc.
Comme on peut ramener toute fonction à n variables à une expression formée exclusivement de fonctions à deux variables et des connecteurs *et*, *ou* et *non*, et vu que toutes les fonctions à deux variables peuvent être exprimées à l'aide de ces connecteurs, il nous suffit de montrer que l'on peut traduire ces trois connecteurs dans n'importe lequel des groupes logiques complets. Prenons par exemple le groupe réduit au connecteur *nand* :

non : $nand(a,b) = \overline{a \wedge b}$, d'où $nand(a,a) = \overline{a \wedge a} = \overline{a}$.

et : $nand(a,b) = \overline{a \wedge b}$, et il nous suffit de prendre la négation du résultat, en utilisant ce que nous venons juste de montrer. $nand\,(nand(a,b), nand(a,b)) = \overline{nand(a,b)} = \overline{(\overline{a \wedge b})} = a \wedge b$.

ou : $nand(c,d) = \overline{c \wedge d} = \overline{c} \vee \overline{d}$. D'où, en prenant $c = \overline{a} = nand(a,a)$ et $d = \overline{b} = nand(b,b)$, $nand\,(nand(a,a), nand(b,b)) = \overline{(\overline{a})} \vee \overline{(\overline{b})} = a \vee b$.

– L'algèbre de Boole se construit sur les booléens à partir de trois opérations internes : $+$, $\times$ et $^-$. Le tableau 3.4 présente les théorèmes et axiomes fondamentaux de l'algèbre de Boole.

Théorème des constantes	$a+0=a$	$a\times 0=0$
	$a+1=1$	$a\times 1=a$
Idempotence	$a+a=a$	$a\times a=a$
Complémentation	$a+\overline{a}=1$	$a\times\overline{a}=0$
Commutativité	$a+b=b+a$	$a\times b=b\times a$
Distributivité	$a+(bc)=(a+b)(a+c)$	
	$a(b+c)=(ab)+(ac)$	
Associativité	$a+(b+c)=(a+b)+c=a+b+c$	$a(bc)=(ab)c=abc$
Théorèmes de De Morgan	$\overline{ab}=\overline{a}+\overline{b}$	$\overline{a+b}=\overline{a}\times\overline{b}$

TAB. 3.4 – Théorèmes et axiomes fondamentaux de l'algèbre de Boole.

– De la table de vérité à l'expression booléenne :
  – développement en somme de produits logiques ;
  – développement en produit de sommes logiques : même principe en considérant les 0, en complémentant puis en appliquant De Morgan ⇒ produit de sommes.

  Un exemple est présenté figure 3.1.

a	b	c	f(a, b, c)
0	0	0	1
1	0	0	1
0	1	0	1
1	1	0	0
0	0	1	0
1	0	1	1
0	1	1	0
1	1	1	1

$f(a,b,c)=1$ si :
a = 0 et b = 0 et c = 0
a = 1 et b = 0 et c = 0 ou
a = 0 et b = 1 et c = 0 ou
a = 1 et b = 0 et c = 1 ou
a = 1 et b = 1 et c = 1 ou

D'où $f(a,b,c)=\overline{a}\overline{b}\overline{c}+a\overline{b}\overline{c}+\overline{a}b\overline{c}+a\overline{b}c+abc=\overline{b}\overline{c}+\overline{a}\overline{c}+ac=(\overline{a}+\overline{b})\overline{c}+ac$.

$f(a,b,c)=0$ si :
a = 1 et b = 1 et c = 0
a = 0 et b = 0 et c = 1 ou
a = 0 et b = 1 et c = 1

D'où $f(a,b,c)=\overline{ab\overline{c}}\times\overline{\overline{a}\overline{b}c}\times\overline{\overline{a}bc}=(\overline{a}+\overline{b}+c)\times(a+b+\overline{c})\times(a+\overline{b}+\overline{c})$.

FIG. 3.1 – Table de vérité d'une fonction et calcul de son expression booléenne.

3.3 Portes et circuits logiques

– Un ordinateur travaille en base 2. Électroniquement le *zéro* correspond(ait) à une tension de 0 à 0.8 V et le *un* à une tension comprise entre 2.8 et 5 V. On a vu que toute fonction binaire peut être représenté par une expression booléenne. Ainsi, tout circuit électrique ou électronique à deux valeurs de tension peut être représentée par une expression booléenne.
– Plutôt que de représenter les circuits par des expressions booléennes, on préfère les représenter en utilisant des symboles logiques (cf. figure 3.2).

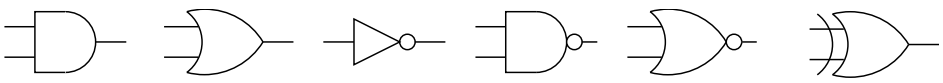


FIG. 3.2 – Portes logiques élémentaires.

  Concrètement chaque porte logique est réalisée électroniquement par un ou deux transistors.
– Plusieurs portes logiques forment un *circuit logique* (cf. figure 3.3 pour une exemple simple). On distingue deux types de circuits logiques :
  – les **circuits combinatoires** qui ne font que combiner selon une table de vérité les variables d'entrée ;

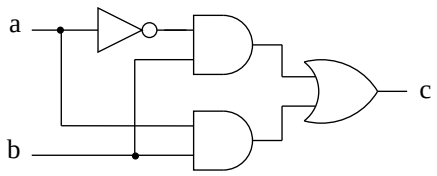


FIG. 3.3 – Exemple de circuit logique : $c = \overline{a}b + ab$.

- les **circuits séquentiels** construits à partir de circuits combinatoires et qui se caractérisent par une capacité de mémorisation.

– La réalisation d'un circuit passe d'abord par la recherche des expressions booléennes, ensuite par leur simplification basée sur l'algèbre de Boole (exemple : les diagrammes de Karnaugh, cf. TD).

3.4 Circuits combinatoires

3.4.1 Additionneur

– Addition de 2 bits x et y (demi-additionneur) :

x	y	S	R
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

$$\textit{table de vérité} \Rightarrow \begin{array}{ll} \text{somme} & = x \oplus y \\ & = x\overline{y} + \overline{x}y \\ & = (x+y)(\overline{x}+\overline{y}) \quad {}^{a} \\ & = (x+y)(\overline{xy}) \\ \text{retenue} & = xy \end{array}$$

[a] En utilisant deux fois la distributivité par l'addition.

La figure 3.4 présente deux réalisations possibles du demi-additionneur (*half-adder* en anglais), la première correspondant à la formule S = $x \oplus y$ et la deuxième à la formule S = $(x+y)(\overline{xy})$.

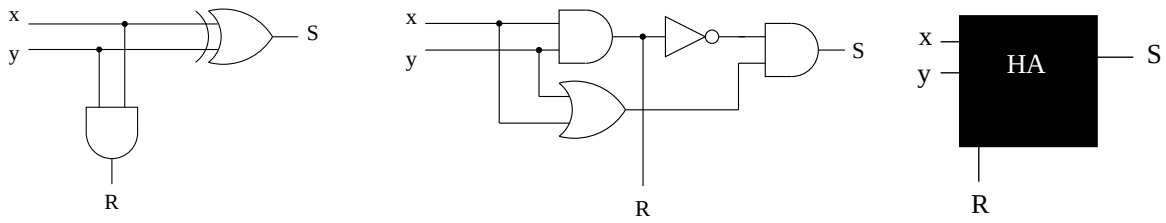


FIG. 3.4 – Deux réalisations du demi-additionneur dont x et y sont les entrées, S est le bit de somme et R celui de retenue ; et représentation schématique.

– Dans une addition de deux nombres, on additionne bit à bit en considérant la retenue du rang précédent ⇒ somme de 3 bits = additionneur complet. La table de vérité de l'additionneur et son schéma sont présentés figure 3.5.
Formules : la *somme s* vaut un si entre les bits x, y et la retenue d'entrée re le nombre de bits à un est impair ; la *retenue de sortie rs* vaut un si les bits x et y valent un, ou si l'un ou l'autre de ces bits vaut un alors que la retenue d'entrée re vaut un.

$$s = re \oplus x \oplus y \text{ et } rs = x \times y + x \times re + y \times re = x \times y + re \times (x \oplus y).$$

– Un additionneur se réalise à l'aide d'additionneur 3 bits mis bout à bout. La figure 3.10 présente un exemple d'un tel additionneur 4 bits.

x	y	re	s	rs
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

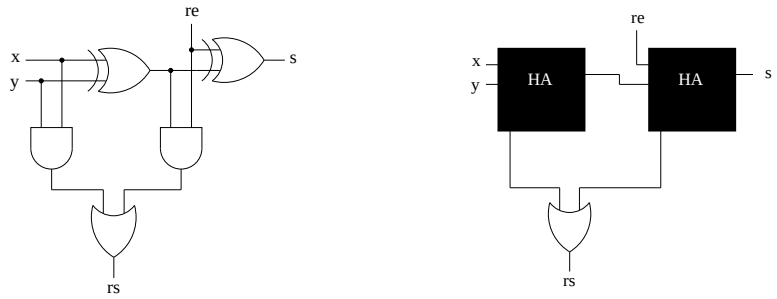


FIG. 3.5 – Additionneur complet : table de vérité, schéma détaillé et schéma synthétique.

– De la même manière on peut réaliser un soustracteur en utilisant une représentation en complément à 2. La multiplication peut alors également être réalisée à partir d'un tel circuit (pas efficacement mais simplement).

3.4.2 Incrémenteur

Retrancher ou ajouter 1 à un nombre est une opération fréquemment réalisée par un processeur, on pourrait utiliser l'addition mais on peut optimiser (le fait que les registres soient équipés d'incrémenteur ou décrémenteur permet en outre de libérer l'UAL $\Rightarrow$ exécution parallèle d'opérations).

La figure 3.6 présente un incrémenteur quatre bits.

Soient e_{n-1}, ..., e_0 les n bits de l'opérande et s_{n-1}, ..., s_0 ceux du résultat. Incrémenter revient à inverser le 1er bit (et les bits suivants s'il y a propagation de retenue).
D'où les formules :

$$s_0 = \overline{e_0}$$

$$s_i = \begin{cases} \overline{e_i} & \text{si } e_{i-1} = \ldots = e_0 = 1 \Leftrightarrow e_{i-1}\ldots e_0 = 1 \\ e_i & \text{sinon} \end{cases}$$

et donc :

$$s_i = e_i \oplus (e_{i-1} \times \ldots \times e_0).$$

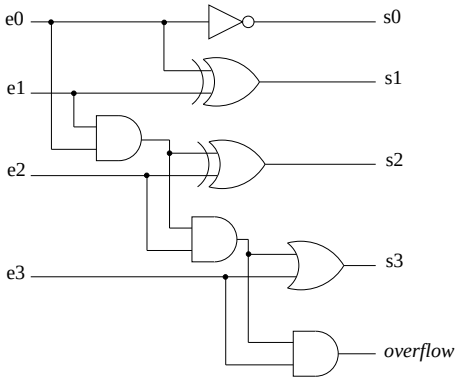


FIG. 3.6 – Incrémenteur 4 bits.

3.4.3 Comparateur

– Un comparateur deux bits a deux entrées x et y et trois sorties E (égalité), S ($x > y$), I ($x < y$).
Table de vérité :

x	y	E	S	I
0	0	1	0	0
0	1	0	0	1
1	0	0	1	0
1	1	1	0	0

D'où les formules : $E = \overline{x \oplus y}, S = x\overline{y}$ et $I = \overline{x}y$ et le circuit présenté figure 3.7.

– Pour un comparateur à plusieurs bits, prenons 4 bits, on s'inspire directement des définitions suivantes :

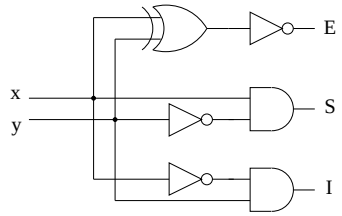


FIG. 3.7 – Comparateur.

1. $a = b$ si $a_0 = b_0$ et .... et $a_4 = b_4 \Rightarrow E = E_0.E_1.E_2.E_3$
2. $a > b$ si $a_3 > b_3$ ou ($a_3 = b_3$ et $a_2 > b_2$) ou .... $\Rightarrow S = S_3 + E_3.S_2 + E_3.E_2.S_1 + E_3.E_2.E_1.S_0$
3. $a < b$ si ni $a > b$ ni $a = b \Rightarrow I = \overline{E}.\overline{S} = EnorS$.

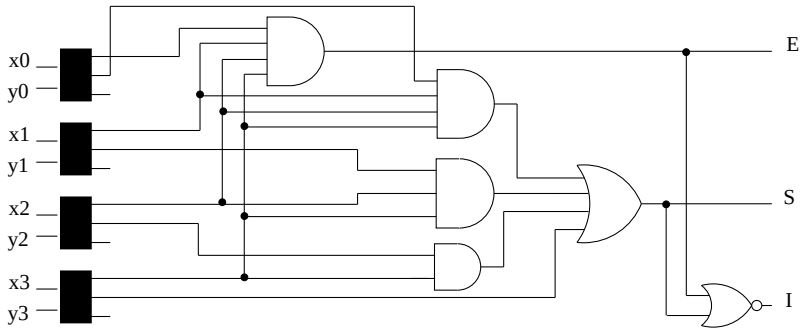


FIG. 3.8 – Comparateur quatre bits fabriqué à partir de comparateurs un bit (boîtes noires).

On a donc un circuit avec 4 comparateurs à deux bits plus quelques portes *et*, *ou* et une porte *nor* (ce circuit est trop délicat pour pouvoir le faire au tableau, mais il est représenté figure 3.8).

– Dans certains processeurs, pour éviter cette circuiterie, on utilise la soustraction, on teste l'indicateur de signe S et l'indicateur de zéro Z —réalisé par un *nor* comme le montre la figure 3.9.

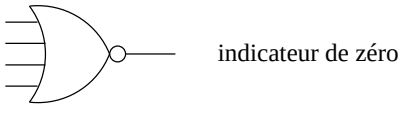


FIG. 3.9 – Indicateur de zéro.

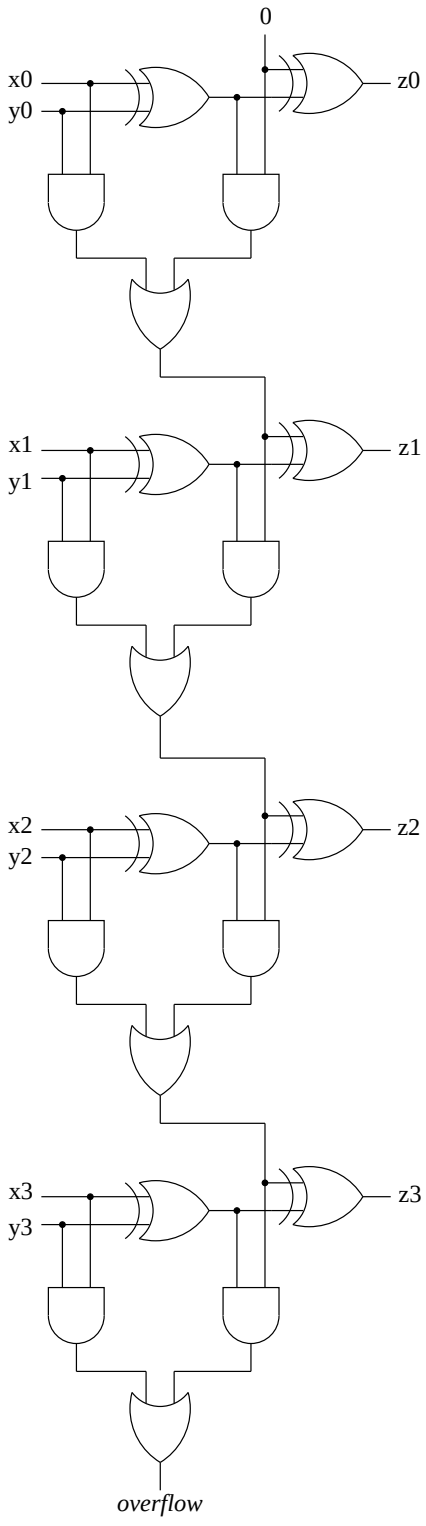


FIG. 3.10 – Additionneur 4 bits.

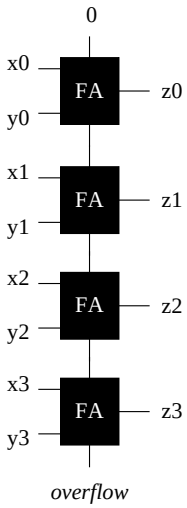


FIG. 3.11 – Représentation schématique de l'additionneur 4 bits.

Remarques :
- Plutôt que d'utiliser un additionneur pour les deux bits de poids faible, et de fixer la retenue entrante à la valeur zéro, on aurait pu n'utiliser qu'un demi-additionneur.
- On suppose dans les circuits combinatoires que la vitesse de propagation est instantanée ce qui n'est pas le cas : on a une propagation temporelle de la retenue, les bits de poids forts sont additionnés après les bits de poids faibles.
- L'additionneur parallèle est asynchrone : les sorties ne sont pas toutes stabilisées en même temps (cas défavorable : $n \times$ temps de passage dans un additionneur).
- Pour accélérer l'addition, il existe des circuits combinatoires à prédiction de retenue calculant directement la retenue de rang i en fonction des entrées de rang inférieur à i.
- Une autre possibilité est l'additionneur série utilisant un seul additionneur complet (plus économique, moins performant) dans lequel on injecte séquentiellement les bits de même rang. Les opérandes passent alors par un registre à décalage.

Pour comparer a et b, on fait $a - b$ puis :

$$\begin{array}{ll} Z = 1 : & a = b \\ Z = 0 : & a \neq b \\ Z = 0 \text{ et } S = 1 : & a < b \\ Z = 0 \text{ et } S = 0 : & a > b \\ Z = 1 \text{ ou } S = 1 : & a \leq b \\ Z = 1 \text{ ou } S = 0 : & a \geq b \end{array}$$

3.4.4 UAL

Une UAL est une Unité Arithmétique et Logique réalisant les opérations de base : opérations logiques (*et*, *ou* et *non*), opérations arithmétiques, comparaisons. Un code d'entrée détermine la partie du circuit qui va fournir le résultat. À la fin de l'opération des indicateurs sont regroupés dans un registre : dernière retenue, *overflow*, nullité du résultat, signe, supériorité, etc.

La figure 3.12 présente la représentation schématique d'une unité arithmétique et logique, et la figure 3.13 présente le schéma logique d'une telle unité 1 bit rudimentaire.

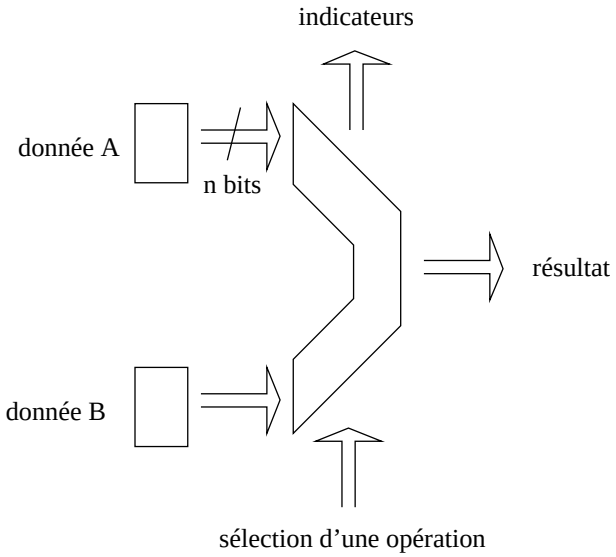


FIG. 3.12 – Représentation schématique d'une unité arithmétique et logique

3.5 Bascules des circuits séquentiels

3.5.1 Définitions

- Dans un *circuit combinatoire*, la valeur des sorties dépendent directement de la valeur des entrées. Un *circuit séquentiel*, quant à lui, a une faculté de mémorisation et la valeur des sorties à l'instant t dépend à la fois de la valeur des entrées et de la valeur des sorties à l'instant $t - 1$.
- Les circuits séquentiels de base, qui forment la brique élémentaire de tout circuit séquentiel, sont les bascules (*flip-flops*) qui ont la particularité d'avoir deux états stables (bistables) : ils sont capables de conserver l'état de leur sortie même si la combinaison des signaux d'entrée ayant provoqué cet état de sortie disparaît.

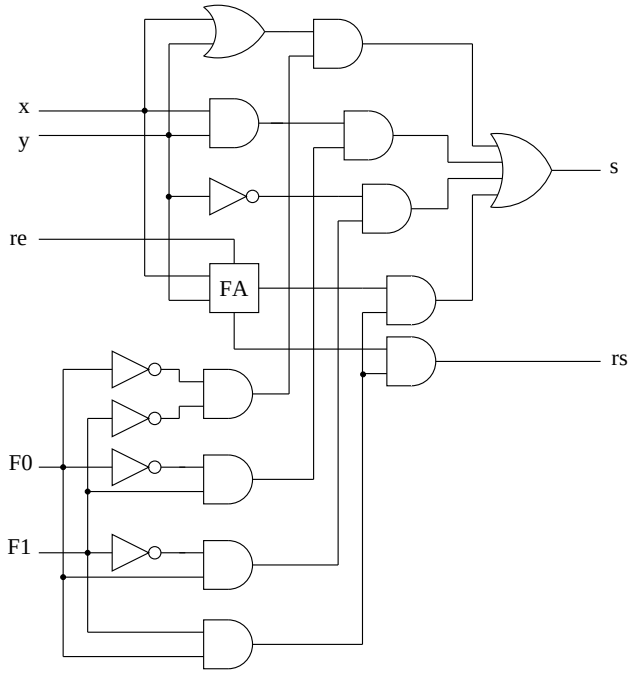


FIG. 3.13 – Circuit combinatoire d'une UAL rudimentaire à 1 bit possédant 4 opérations (**et**, **ou**, **non** et **addition**) de code d'opérations sur 2 bits F0 et F1 (00 : **ou**, 01 : **et**, 10 : **non**, 11 : **addition**)

– Une horloge est un composant passant indéfiniment et régulièrement d'un niveau haut à un niveau bas (succession de 1 et de 0), chaque transition s'appelle un *top*. On distingue les bascules synchrones, asservies à des impulsions d'horloge et donc insensibles aux bruits entre deux tops, et les bascules asynchrones, non asservies à une horloge et prenant en compte leurs entrées à tout moment.

3.5.2 Bascule RS

– Cette bascule est réalisée par le circuit combinatoire de la figure 3.14.

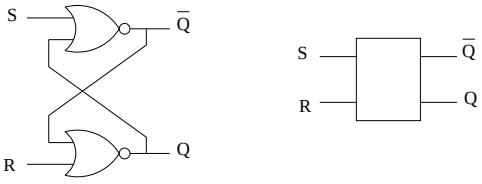


FIG. 3.14 – Bascule RS : circuit logique et schéma synthétique.

– Cette bascule présente deux états stables pour la configuration des entrées $R = S = 0$: le couple de sorties $Q = 0$ et $\overline{Q} = 1$ et le couple de sorties $Q = 1$ et $\overline{Q} = 0$.
– $R = 0$ et $S = 0 \Rightarrow Q_{t+1} = Q_t$
$R = 0$ et $S = 1 \Rightarrow Q_{t+1} = 1$ (mise à 1, S : *set*)
$R = 1$ et $S = 0 \Rightarrow Q_{t+1} = 0$ (effacement, R : *reset*)
$R = 1$ et $S = 1$: configurations interdites : tant que R et S restent à 1, Q et $\overline{Q}$ restent à 0 mais lorsque R et S changent on ne peut prévoir dans quel état on se trouve *(ex. : montrer ce qui se passe sur le circuit avec R et S*

passant à 0).

– Ce circuit donne lieu à la table de vérité suivante :

R	S	Q_t	Q_{t+1}	
0	0	0	0	$Q_{t+1} = Q_t$
0	0	1	1	$Q_{t+1} = Q_t$
0	1	0	1	Mise à 1
0	1	1	1	Mise à 1
1	0	0	0	Effacement
1	0	1	0	Effacement
1	1	0	$*$	Interdit
1	1	1	$*$	Interdit

En simplifiant par Karnaugh (voir la figure 3.15) en donnant la valeur 1 aux cas interdits, on obtient : $Q_{t+1} = S + \overline{R}Q_t$.

Q \ RS	RS	$\overline{\text{R}}$ S	$\overline{\text{RS}}$	R $\overline{\text{S}}$
Q	1	1	1	0
$\overline{\text{Q}}$	1	1	0	0

FIG. 3.15 – Table de Karnaugh pour la bascule RS.

– Les bascules RS sont fréquemment utilisées dans la construction de circuits anti-rebond. Propriétés utiles des bascules *RS*, si $R = 0$ et $S = 1 \Rightarrow Q = 1$ et si S change, Q reste à 1.

3.5.3 Bascule RSC

– La bascule RS est asynchrone au sens ou l'impulsion sur l'une des entrées est prise en compte à tout moment. Ceci pose deux problèmes :
  – sensibilité aux bruits parasites ;
  – synchronisation (quand peut-on prendre les données en entrées ?)

  Solution : R et S ne sont pris en compte qu'à des instants bien déterminés en utilisant une horloge. Le bon fonctionnement de la bascule suppose que la période de l'horloge est largement supérieur au temps de stabilisation du circuit. Le schéma de la bascule RSC est présenté figure 3.16.
  Quand $C = 0$, les valeurs de R et S sont ignorées et l'état précédent est conservé. Quand $C = 1$, la bascule passe sous le contrôle des entrées R et S et la bascule RSC a le même comportement qu'une bascule RS.

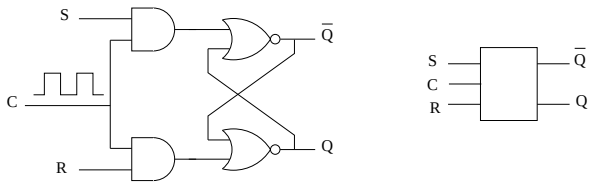


FIG. 3.16 – Bascule RSC : circuit logique et schéma synthétique.

– Le comportement de la bascule peut être résumé par la table de vérité suivante :

C	R	S	Q_{t+1}
1	0	0	Q_t
1	0	1	1
1	1	0	0
1	1	1	$*$
0	x	x	Q_t

3.5.4 Bascule D

– La bascule RSC ne résout pas le problème des conditions interdites. De plus ce que souhaite généralement le concepteur, c'est mémoriser l'état en entrée. D'où la bascule D (pour *data*) :

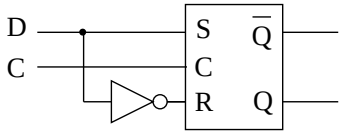


FIG. 3.17 – Circuit logique de la bascule D.

– Comme R et S ne peuvent pas être égaux, les cas interdits ne peuvent plus se produire. Si $C = 1 \Rightarrow Q_{t+1} = D$ et si $C = 0 \Rightarrow Q_{t+1} = Q_t$.
– Les bascules D sont généralement aménagées pour :
  – se déclencher sur un front montant (immunité maximale aux bruits) ;
  – disposer de deux entrées supplémentaires *preset* et *clear* indépendantes de l'horloge et permettant de mettre à 1 ou à 0 la bascule.

3.5.5 Bascules JK et T

Une autre manière de lever les cas interdits est la bascule JK (pas de signification particulière pour J et K) présenté figure 3.18. Cette bascule a le même comportement qu'une bascule RS sauf que si $J = K = 1 \Rightarrow Q_{t+1} = \overline{Q_t}$. La table de vérité ci-dessous définit son comportement quand $J = K = 1$:

Q_t	$\overline{Q_t}$	J	K	S	R	Q_{t+1}	$\overline{Q_{t+1}}$
0	1	1	1	1	0	1	0
1	0	1	1	0	1	0	1

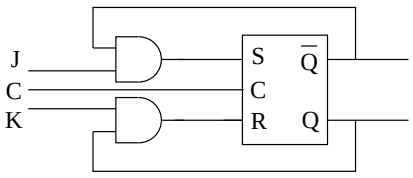


FIG. 3.18 – Bascule JK.

En simplifiant par la méthode Karnaugh (voir la table figure 3.19), on obtient l'équation caractéristique $Q_{t+1} = \overline{K}Q_t + J\overline{Q_t}$. Autrement dit, la table de vérité de la bascule JK est présenté figure 3.20.

En fixant $J = K = 1$, on obtient une bascule T (*trigger flip-flop*) dont le rôle est simplement d'inverser la sortie. La figure 3.21 présente cette bascule, et la figure 3.22 présente sa table de vérité.

Q \ JK	JK	$\bar{J}K$	$\overline{JK}$	$J\bar{K}$
Q	0	0	(1)	(1)
$\bar{Q}$	(1)	0	0	(1)

FIG. 3.19 – Table de Karnaugh pour la bascule JK.

J	K	Q_{t+1}
1	0	1
0	1	0
0	0	Q_t
1	1	Q_{t+1}

FIG. 3.20 – Table de vérité de la bascule JK ($C = 1$).

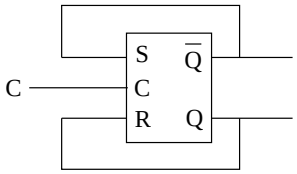


FIG. 3.21 – Bascule T.

C	Q_t	Q_{t+1}
0	0	0
0	1	1
1	0	1
1	1	0

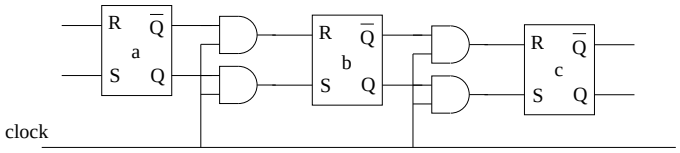
FIG. 3.22 – Table de vérité de la bascule T.

3.6 Application des bascules

Les bascules sont utilisées pour leurs effets mémoires et chacune permet de stocker un bit. Elles sont utilisées pour construire : registres, registres à décalage, compteurs, convertisseurs série-parallèle, etc.

3.6.1 Registres à décalage

La figure 3.23 présente le schéma d'un ensemble de trois registres à décalage et un tableau illustrant le comportement de ce système.



temps	a	b	c
0	0	1	1
1	0	0	1
2	0	0	0
3	1	0	0
4	0	1	0
5	0	0	1

FIG. 3.23 – Registres à décalage à droite et illustration de leur comportement.

3.6.2 Compteur binaire

Comme le montre la figure 3.24, nous réalisons un compteur binaire en utilisant des bascules *flip-flop* dont l'activation a lieu sur le front descendant du signal d'horloge. Notre compteur a trois bit et peut donc prendre 8 valeurs : de 0 à 7.

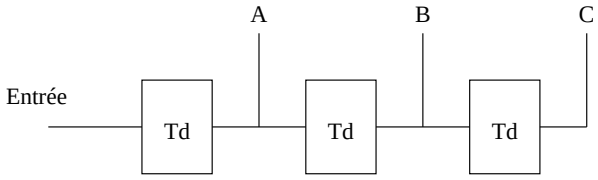
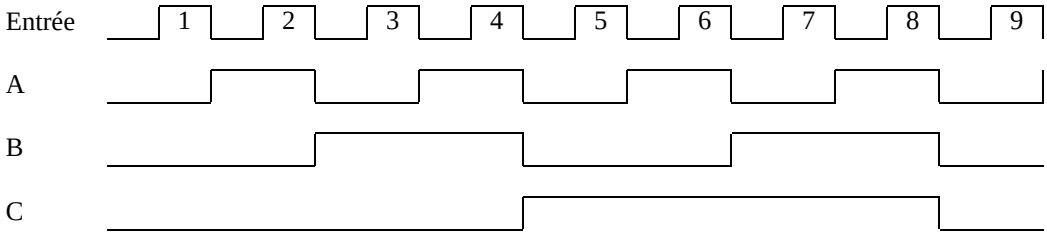


FIG. 3.24 – Schéma d'un compteur trois bits.



3.6.3 Mémoire de registres

Le schéma de la figure 3.25 correspond à une SRAM (*static RAM*) 4x3 bits. Un bit = une bascule = 2 portes NOR = 4 transistors (2 par porte) ⇒ mémoire très rapide. À l'opposé, la DRAM (*Dynamic RAM*) est simplement constituée d'un transistor couplé à un condensateur. Celui-ci se décharge progressivement et doit être périodiquement rechargé ⇒ rafraîchissement de la mémoire (d'où le vocable *dynamic*), ce mécanisme de rafraîchissement est intégré au boîtier. Ce temps de rafraîchissement, durant lequel aucun accès ne peut se faire, rend la DRAM moins rapide que la SRAM. Avantage : fabrication plus simple, moins coûteuse, intégration importante. SRAM : registres du processeur, mémoire cache ; DRAM : mémoire centrale.

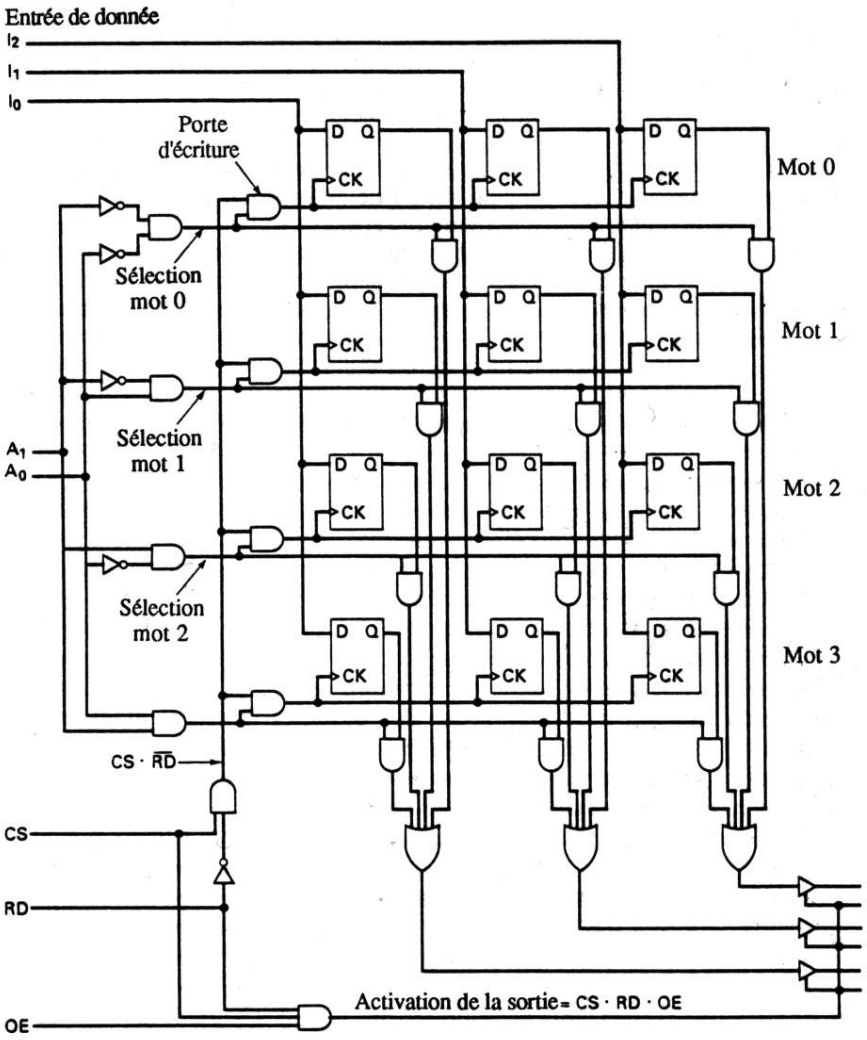


FIG. 3.25 – Schéma logique d'une mémoire 4×3 bits.

Chapitre 4

Mémoires

4.1 Hiérarchie de mémoire

- Un ordinateur utilise plusieurs types de mémoires (mémoire cache, mémoire centrale, disques durs, CD-ROM, etc.) aux caractéristiques variées (volatilité, rapidité d'accès, etc.)
- L'organisation de l'ensemble de ces mémoires tient compte de deux principes :

1. Le principe de localité : on constate que l'exécution d'un programme passe 90% du temps dans 10% du code.
   - Localité temporelle : une donnée ou une instruction utilisée récemment sera probablement réutilisée prochainement ;
   - Localité spatiale : une donnée (resp. une instruction) placée près d'une autre donnée (resp. d'une autre instruction) référencée récemment sera probablement accédée prochainement.

2. Plus une mémoire est rapide d'accès, plus elle est petite. En effet, le coût et l'encombrement des mémoires sont proportionnels à leur rapidité ce qui impose de faire un (bon) compromis rapidité/quantité.

   La mémoire centrale est volatile et parfois de taille insuffisante $\Rightarrow$ disques durs. Lecture/écriture en mémoire centrale : opérations lentes par rapport à la vitesse du processeur $\Rightarrow$ mémoire intermédiaire entre registres et mémoire centrale = mémoire cache. Il se dessine donc une hiérarchisation de l'ensemble des mémoires.

   Ordre décroissant selon : rapidité, prix, proximité du processeur.

Type	Capacité
Registre	< 400
Cache interne (L1) SRAM (bascules)	32 à 256 ko
Cache externe (L2) SRAM	128 Ko à 1 Mo
Mémoire centrale DRAM (condensateurs)	32 Mo à 1 Go
Disques durs	Dizaines de Go
Archivage (bandes, DVD, etc.)	Quelques Go

   Registres : les registres sont utilisés pour assurer le stockage temporaire d'informations nécessaires à l'exécution de l'instruction en cours de traitement. Les registres se situent à l'intérieur même de l'unité centrale. Un registre mémorise un certain nombre de bits, souvent un mot mémoire.

4.2 Organisation des informations

4.2.1 Format

Bit : unité mémoire de base : 0, 1.

Octet (*byte* en anglais) : 8 bits

Mot : unité d'information adressable (8, 16, 32, 24, 64, etc. bits). Les valeurs 32 et 64 tendent à se généraliser.

4.2.2 Caractéristiques des mémoires

Capacité ou taille de la mémoire : nombre d'octets.

Temps d'accès : temps qui s'écoule entre une demande de lecture ou d'écriture et son accomplissement.

Cycle mémoire : temps minimum entre deux accès consécutifs (supérieur au temps d'accès à cause des opérations de synchronisation, de rafraîchissement, etc.)

Débit : nombre d'octets ou bits pouvant être lus ou écrits par secondes.

Volatilité : perte ou non du contenu (RAM : volatile, si l'on coupe le courant tout est aussitôt perdu ; bande magnétique : perte après 5 ans).

4.2.3 Types d'accès

Accès séquentiel : pour accéder à une information, il faut lire toutes celles qui précèdent (bandes).

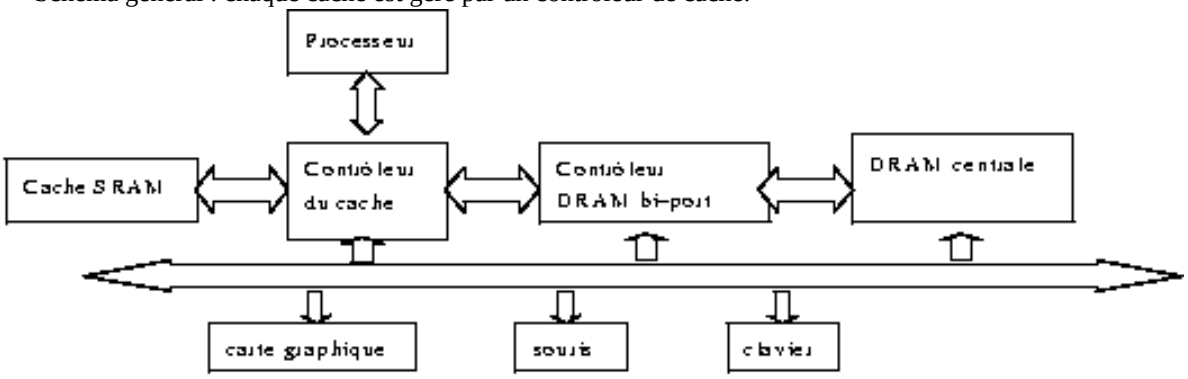
Accès direct : par adresse

Accès par le contenu (mémoire associative) : la mémoire n'est pas constituée d'une suite de données mais d'une suite de couples (adresse, donnée). La recherche se fait sur l'adresse pour accéder ensuite directement à la donnée.

4.3 Mémoires caches

4.3.1 Principes généraux

- But : éviter de rechercher en mémoire centrale des données déjà recherchées précédemment en les conservant près du processeur dans une mémoire à accès rapide (SRAM : 8 à 16 fois plus rapide que DRAM mais 4 à 8 fois plus volumineuse). Basée sur le principe de proximité (ex. : instructions des boucles).
- On distingue :
  - Le cache interne ou de niveau 1 ou L1 : dans le processeur, unifié = contient instructions et données (ex. : Intel 486), actuellement au moins 2 caches : 1 cache de données et 1 caches d'instructions (ex. Pentium : 2 caches L1 de 8 ko, Pentium III : 2 chaches L1 de 16 kO, actuellement cache L1 : 128 ko). Avantage des caches séparés : les opérations mémoires sur des instructions et données indépendantes peuvent être simultanées.
  - Cache externe ou de niveau 2 ou L2 (unifié) : à côté du processeur, généralement de 256 ko.
- Schéma général : chaque cache est géré par un contrôleur de cache.



– Exemple de fonctionnement :

1. Le processeur demande une instruction présente dans une boucle d'un programme en plaçant l'adresse de cette instruction sur le bus d'adresse.
2. Le contrôleur de cache cherche si l'information est dans le cache en fonction de l'adresse → échec
3. Le contrôleur de cache entame un cycle de lecture en mémoire centrale.
4. La mémoire centrale envoie l'information au processeur et le contrôleur de cache la copie au passage dans le cache.

⇒ Au prochain accès à cette même instruction, le contrôleur de cache enverra directement la donnée au processeur sans démarrer un cycle de lecture en mémoire centrale.

4.3.2 Mode d'accès des caches

– Une mémoire cache est structurée en lignes appelées *voies*. Chaque voie contient des données dont les adresses sont consécutives en mémoire centrale. La partie de l'adresse commune à ces données est appelée *étiquette*. Une voie est soit totalement inoccupée soit totalement pleine.
Remarque : une voie contient toujours des octets consécutifs, quand une donnée est requise, le contrôleur de cache demande les 32 octets (si la ligne contient 32 octets).
Inconvénient : toutes les données dont les adresses ont la même étiquette doivent prendre place dans le cache.
Motivation : comme lire une donnée en mémoire est très coûteux en temps, à chaque fois qu'une donnée est lue, les données « voisines » — c'est-à-dire qui seront stockées dans la même voie du cache sont aussi lues. On espère que cette lecture groupée sera profitable. Cette espérance est basée sur le *principe de localité*.
– À chaque voie est associée un *repère* placé dans une table des repères. Un repère contient l'étiquette de la voie ainsi que des bits d'information dont un bit indiquant si la ligne est occupée ou non.
– Dans une mémoire cache, l'emplacement d'une donnée est calculée en fonction de son adresse dans la mémoire centrale. Il existe 3 modes de placements : associatif, direct et par ensemble.
– **cache associatif** (*fully associative*) : une donnée peut être inscrite n'importe où dans le cache (cf. figure 4.1).

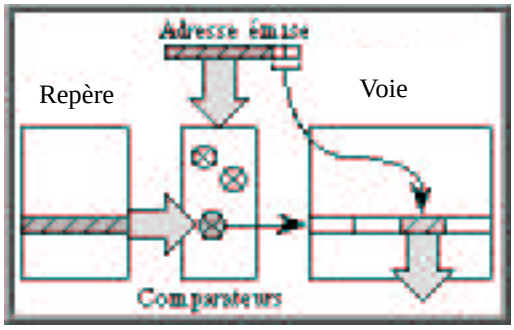


FIG. 4.1 – Cache associatif.

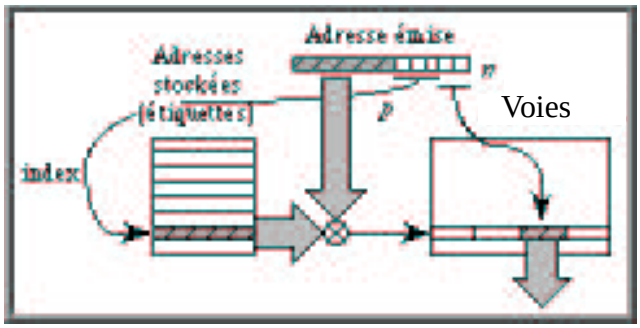


FIG. 4.2 – Cache direct.

– **cache direct** (*direct mapped*) : une donnée ne peut être placée que dans une voie précise. Ceci permet d'éviter les N comparateurs. Si le cache dispose de 2^p voies, p bits de l'adresse servent à désigner la voie concernée, l'entier positif représenté par les p bits est appelé *index* (cf. figure 4.2).
Prenons un cache direct de 4 Ko possédant 128 voies de 32 octets chacune. Les adresses sont sur 32 bits. Ainsi le découpage de l'adresse se réalise comme suit :
– bit 0-4 (5 *bits*) : n° d'octet dans la voie (car $32 = 2^5$)
– bit 5-11 (7 *bits*) : n° de voie (car $128 = 2^7$)
– bit 12-31 (20 *bits*) : étiquette
Inconvénient : toutes les données dont les adresses ont la même étiquette doivent prendre place dans la même voie du cache.

– solution adoptée : **cache associatif par *n*-ensembles** (*n-way set associative*). Il combine les deux techniques (c'est le plus couramment utilisé). On utilise ici *n* caches directs. L'adresse fournit un index et la donnée peut prendre place dans n'importe quel cache direct (cf. figure 4.3).

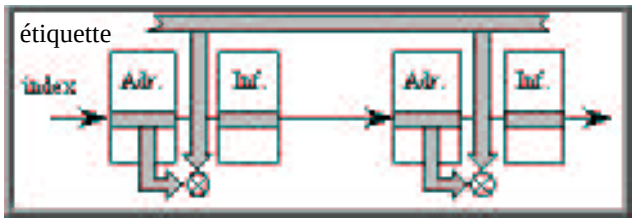


FIG. 4.3 – Cache associatif par *n* ensembles.

Un cache direct peut être vu comme un cache associatif par 1-ensemble et un cache associatif à *m* voies, comme un cache associatif à *m*-ensembles de 1 voie chacun.

4.3.3 Remplacement des données dans le cache

– Lorsqu'une donnée doit être écrite dans le cache et que les voies qu'elle peut occuper sont déjà prises, le contrôleur de cache doit choisir quelle voie remplacer.
– Dans un cache à accès direct le problème ne se pose évidemment pas. En revanche dans les caches associatifs, ou associatifs par ensemble, une stratégie doit être mise en œuvre dans de tel cas.
Stratégies possibles :

Random : pour changer uniformément les voies, celles-ci peuvent être remplacées de manière aléatoire.

Avantage : implémentation hard peu coûteuse.

Inconvénient : ignore le principe de localité.

Least-Recently Used (LRU) : pour réduire la probabilité de se débarrasser d'une information qui sera réutilisée prochainement, les données de la voie inutilisée depuis le plus longtemps sont remplacées.

Avantage : tient compte du principe de localité.

Inconvénient : quand le nombre de voies à gérer augmente, l'algorithme LRU devient de plus en plus coûteux (difficile à implémenter, lent et souvent approximatif).

Autres stratégies :
– **First-In First-Out (FIFO)**
– **Most-Recently Used (MRU)**
– **Least-Frequently Used (LFU)**
– **Most-Frequently Used (MFU)**

4.3.4 Interaction avec la mémoire centrale

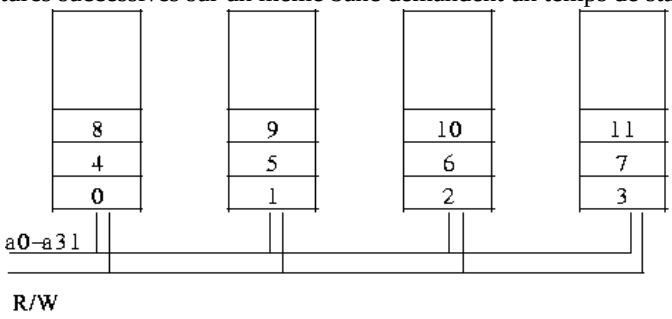
– La lecture est l'opération la plus courante dans les caches. Toutes les instructions sont lues et la plupart d'entre elles ne provoquent pas d'écriture. Les politiques de lecture lors d'un échec dans le cache sont :
  – **Read Through** : la lecture se fait directement de la mémoire centrale vers le CPU.
  – **No Read Through** : la lecture se fait de la mémoire centrale vers le cache puis du cache vers le CPU.
– Deux politiques d'écriture sont employées pour traiter le cas d'un succès dans le cache :
  – **Write Through** : l'information est écrite dans le cache et dans la mémoire centrale.
  Avantages :
    – un remplacement de l'information modifiée ne nécessite pas d'écriture en mémoire centrale et la lecture ayant provoqué le remplacement reste rapide ;
    – facile à réaliser ;
    – la mémoire centrale contient toujours des informations cohérentes avec le cache.
  Inconvénients :

  - l'écriture est lente ;
  - chaque écriture nécessite l'accès à la mémoire centrale (au détriment des périphériques).
- **Write Back** : l'information est écrite uniquement dans le cache. Elle est écrite dans la mémoire centrale seulement lors d'un remplacement. Un bit, appelé **dirty bit**, indique pour chaque voie s'il est nécessaire de mettre à jour la voie en mémoire centrale.
  Avantage :
  - l'écriture se fait à la vitesse d'accès de la mémoire cache ;
  - plusieurs écritures sur une voie ne demandent qu'une écriture en mémoire centrale ;
  - la mémoire centrale est d'avantage disponible pour les périphériques.

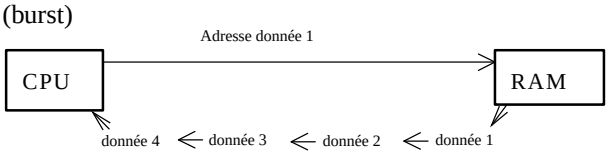
  Inconvénients :
  - implémentation hard compliquée ;
  - la mémoire centrale n'est pas toujours cohérente avec le cache ;
  - une lecture provoquant un remplacement peut conduire à une écriture en mémoire centrale.
- Il y a également deux politiques lors d'une écriture sur une information non présente dans le cache :
  - **Write Allocate** : l'information est d'abord chargée dans le cache puis modifiée.
  - **No Write Allocate** : l'information est directement modifiée dans la mémoire centrale et n'est pas chargée dans le cache.

4.4 Mémoire centrale

- Accès direct.
- Entrelacement en structurant la mémoire en bancs accessibles indépendamment pour accélérer les accès (car deux lectures ou écritures successives sur un même banc demandent un temps de stabilisation des signaux).



- Lecture en mode rafale (burst)



Chapitre 5

Le langage de la machine

5.1 Instructions machine

5.1.1 Le processeur

- Le processeur (CPU : *Central Processing Unit*) est un gigantesque circuit séquentiel dont les signaux d'entrées et de sorties sont principalement pris et écrits en mémoire RAM.
- Un programme est une suite d'instructions placées dans la mémoire. Une instruction est faite de 0 et 1 qui correspondent à des signaux pour le CPU. Deux instructions consécutives dans le programme sont consécutives dans la mémoire (consécutive dans l'espace virtuel, cf. adressage de la mémoire). Il existe dans tout processeur un registre spécial nommé PC (*Program Counter*) contenant l'adresse de l'instruction courante.
- Un compilateur traduit un programme écrit dans un langage conventionnel en instructions machine $\Rightarrow$ pour un langage, un compilateur par type de processeur.
- Rôle du processeur : chercher une instruction, l'exécuter, écrire son résultat, chercher une instruction, etc.
- *Schéma d'un processeur élémentaire avec : registres généraux, registres spéciaux (PC, SP, etc.), unité de décodage, UAL, unité de bus.*
  - Registres spéciaux :
    - PC : adresse en RAM de l'instruction courante, incrémenté automatiquement et modifié lors des sauts.
    - IR (*Instruction Register*) : contient l'instruction courante.
    - Flags : *overflow* (débordement), *carry* (retenue), zéro, etc.
    - SP (*Stack Pointer*) : pointeur de pile pour le passage des paramètres de fonctions.
  - Registres généraux : registres contenant les opérandes et les résultats des calculs, chaque registre a un numéro.
  - UAL : unité arithmétique et logique.
  - Unité de décodage : dirige les signaux depuis et vers l'UAL et les autres registres.
  - Unité de bus : dialogue avec la RAM pour les échanges entre la RAM et les registres.
- Un concepteur de processeur commence par écrire les instructions que doit savoir exécuter le CPU, le circuit du CPU est réalisé ensuite.
- Fabricants de processeurs : Intel, AMD, Digital (Ultra), Motorola (68xxx), Sun (Sparc).

5.1.2 Types d'instructions

Charger en provenance de la mémoire, sauver en mémoire, additionner, etc., sont des instructions pour le processeur, l'ensemble des instructions compréhensibles forme le *jeu d'instructions*. Les instructions que l'on trouve dans le jeu de chaque processeur peuvent être classées en 6 groupes :

transfert de données (*load*, *move*, *store*) : pour chargerde la mémoire, sauver en mémoire, effectuer des transferts de registre à registre, de mémoire à mémoire, etc. ;

opérations arithmétiques (*add*, *sub*, *mul*, *div*) : les quatre opérations de base en entier, signé ou non, court ou long, ou en réel en virgule flottante, simple ou double précision ;

opérations logique (*and*, *or*, *not*, *xor*, etc.) ;

contrôle de séquence : branchement impératifs et conditionnels, boucles, appels de procédures, etc. ;

entrées/sorties (*read*, *print*, etc.) ;

manipulation diverses : décalages, conversions de format, incrémentation de registres, etc.

Jeu d'instructions = 50 à 250 instructions.

RISC (*Reduced Instruction Set Computer*) : petit nombre d'instruction, format fixe, câblé.

CISC (*Complex Instruction Set Computer*) : jeu riche (racine carrée, multiplication en double précision) ⇒ opérations microprogrammées.

5.1.3 Format d'instruction

Chaque instruction est représentée en mémoire par une suite d'octets. Processeur Intel : 1 à 6 octets ; Mips : 4 octets. Un code, en tête des deux premiers octets permet de connaître le type de l'instruction : sa longueur et sa structure.

Exemple : processeur Mips 32 registres numérotés de 0 à 31 avec des instructions de taille fixe (32 bits).

Ex 1 :

Donne	0	17	18	8	0	32
Nombre de bits	6	5	5	5	5	6

- 0 : format de l'instruction ;
- 17, 18, 8 : numéros de registre (sur 5 bits car 32 registres) ;
- 0 : pas utilisé dans l'addition ;
- 32 : indique l'addition et signifie l'addition du contenu des registres 17 et 18 et le placement du résultat dans le registre 8.

Le premier et le dernier champ (0 : UAL et 32 : +) forment le code de l'opération (*opcode*).

Ex 2 :

Donne	4	8	7	9
Nombre de bits	6	5	5	16

- 4 : indique l'addition avec un opérande immédiat ;
- 8, 7 : numéros de registres ;
- 9 : opérande.

Ici la valeur d'un des opérandes est directement codée dans l'instruction. Ce code signifie l'addition du registre 7 et de la valeur « 9 » et le stockage du résultat dans le registre numéro 8.

Ex 3 :

Donne	5	8	21	1140
Nombre de bits	6	5	5	16

- 5 : branchement si non égal ;
- 8, 21 : uméros des registres à comparer ;
- 1140 : adresse où il faut se rendre s'il n'y a pas égalité.

Il est possible d'écrire dans le langage du processeur ⇒ il faut connaître le nombre de ses registres généraux.

Dans la pratique :

- code mnémonique pour les opérandes ;
- chaque registre à un nom ($1, ..., $31) ;
- les adresses ne sont données que par des étiquettes.

Exemple : les 3 instructions précédentes s'écrivent :

Étiquette	**Instruction**	**Signification**
Loop	add $8, $17, $18	($8 = $17 + $18)
	addi $8, $7, 9	($8 = $7 + 9)
	bne $8, $21, Loop	aller à Loop si $8 $\neq$ $21

Mode d'adressage	**Exemple d'instruction**	**Signification**	**Cas d'utilisation**
Registre	`Add R4, R3`	`R4` ← `R4 + R3`	La valeur est dans un registre
Immédiat ou littéral	`Add R4, #3`	`R4` ← `R4 + 3`	Pour les constantes. Dans certaines machines les modes littéral et immédiat sont deux modes distincts.
Déplacement ou basé	`Add R4, 100(R1)`	`R4` ← `R4 + M[100+R1]`	Accès aux variables locales
Indirect par registre	`Add R4, (R1)`	`R4` ← `R4 + M[R1]`	Accès en utilisant un pointeur ou une adresse calculée.
Indexé	`Add R3, (R1+R2)`	`R3` ← `R3 + M[R1+R2]`	Parfois utile dans l'adressage des tableaux (`R1` = base du tableau ; `R2` = valeur d'index).
Direct ou absolu	`Add R1, (1001)`	`R1` ← `R1 + M[1001]`	Parfois utile pour l'accès aux variables statiques ; les valeurs d'adresse doivent parfois être grandes.
Indirect via la mémoire	`Add R1, @(R3)`	`R1` ← `R1 + M[M[R3]]`	Si `R3` est l'adresse d'un pointeur p, ce mode donne $*p$.
Auto-incrémenté	`Add R1, (R2)+`	`R1` ← `R1 + M[R2]` `R2` ← `R2 +` d	Utile pour sauter à travers des tableaux dans une boucle. `R2` pointe sur le début du tableau ; chaque référence incrémente `R2` de la taille d'un élément : d.
Auto-décrémenté	`Add R1, -(R2)`	`R2` ← `R2 -` d `R1` ← `R1 + M[R2]`	Même utilisation que l'auto-incrément. L'auto-incrément/décrément peut aussi être utilisé pour implanter empilement et dépilement.
Indexé étendu	`Add R1, 100(R2)[R3]`	`R1` ← `R1 + M[100+R2+R3*`d`]`	Utilisé pour indexer des tableaux. Dans certaines machines, peut être appliqué à tout mode d'adressage basé.

FIG. 5.1 – Modes d'adressage dans l'architecture VAX.

5.1.4 Modes d'adressage

5.2 Appel de fonctions

Considérons le programme suivant :

```
adr.    pgm                                  adr.    pgm
instruction                                  instruction
1000    int somme(int n)                     1500    main()
        {                                            {
        int r ;                                      int x, s ;
        if (n == 0) return 0 ;                       x = 2 ;
        r = somme(n-1) ;                             s = somme(x) ; // call 1000
        return n+r ;                         1700    .
        }                                            .
                                                     }
```

- Deux problèmes se posent :
  - comment passer les paramètres à la fonction ? dans *main*, on peut mettre x dans un registre que *somme* récupère, mais *somme* est récursive aurons-nous assez de registres ?
  - il faut pouvoir se souvenir de l'instruction sur laquelle revenir dans A.
- Solution : utiliser une pile = zone de la RAM réservée dans laquelle on place toutes les variables locales aux fonctions ainsi que les adresses de retour.

Les figures de 5.2 à 5.8 montrent l'état de la pile avant et après les différents appels de fonctions qui ont lieu lors de l'exécution du programme ci-dessus. Dans ces figures, `SP` désigne le pointeur de pile (*Stack pointer*) et `FP`, le pointeur de cadre, qui est un pointeur local de pile pointant sur une position fixe à l'intérieur du bloc mémoire de la procédure

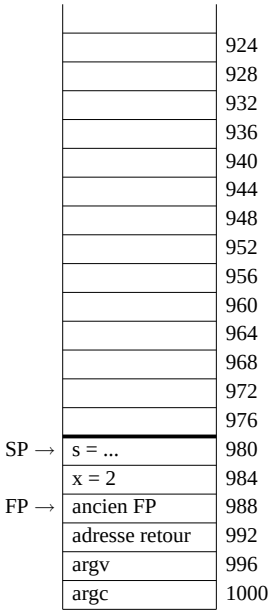


FIG. 5.2 – Avant l'appel à `somme(2)`.

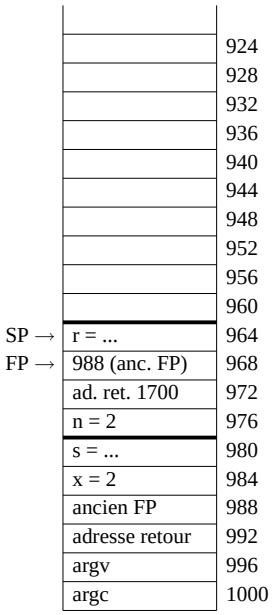


FIG. 5.3 – Avant l'appel à `somme(1)`.

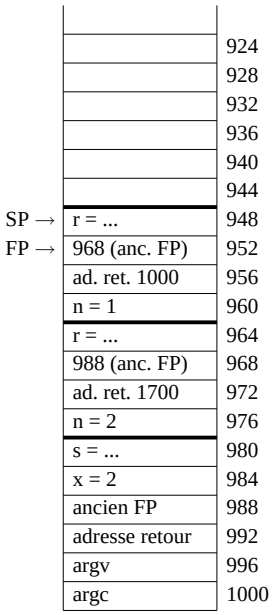


FIG. 5.4 – Avant l'appel à `somme(0)`.

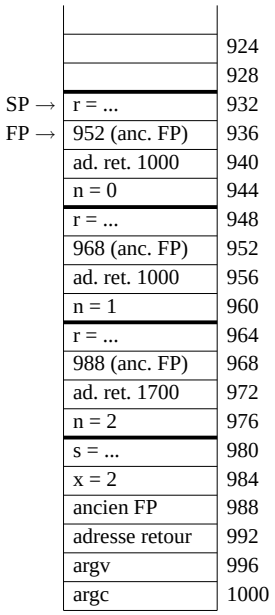


FIG. 5.5 – Avant le retour de l'appel à `somme(0)`.

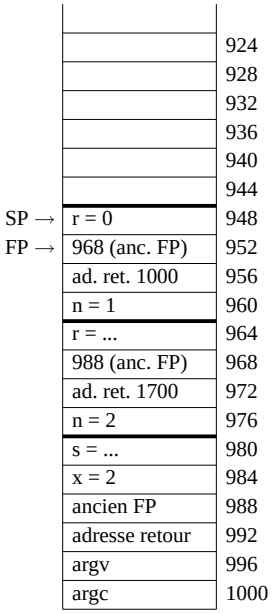


FIG. 5.6 – Avant le retour de l'appel à `somme(1)`.

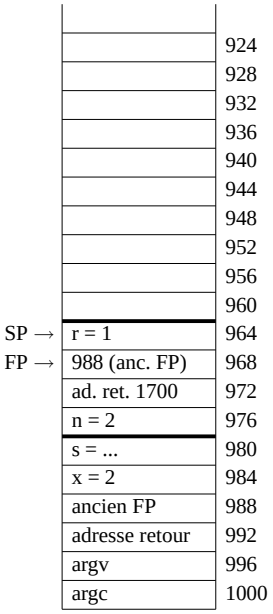


FIG. 5.7 – Avant le retour de l'appel à `somme(2)`.

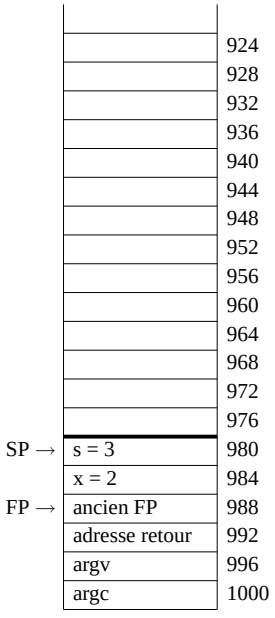


FIG. 5.8 – Après le retour de l'appel à `somme(2)`.

(*Frame pointer*).

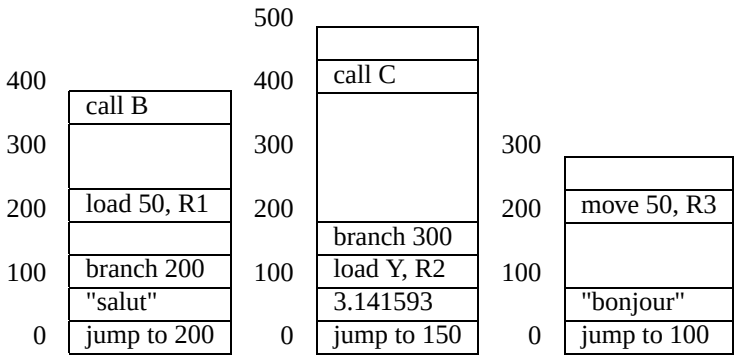
5.3 Exemples d'assembleur (montrés sur transparents)

Cf. figure 5.9.

On remarque que l'Intel empile d'abord le dernier paramètre pour finir par le premier, contrairement au mips et sparc.

5.4 Du programme C vers le processus

- Programme C —(compilation)⟶ langage d'assemblage —(assembleur)⟶ fichier objet —(édition de liens)⟶ fichier exécutable —(chargement)⟶ processus.
- Problème de l'édition de liens : trois programmes :



Soit `load` le chargement depuis la mémoire centrale et `move` le chargement d'une adresse interne (chaîne de caractères).

Il nous faut produire un exécutable à partir des trois modules. Problèmes :

- certaines adresses dans les instructions sont relatives : par exemple dans le programme C, l'adresse 50 est relative, d'où un problème de **relocalisation**.
- A veut exécuter l'appel `call B`, or l'adresse de B n'est pas connue : problème de **références externes**.

La figure 5.10 illustre ces problèmes.

- Travail de l'éditeur de liens :
  1. Construire une table contenant les longueurs de chaque module.
  2. Assigner une adresse de départ à chaque module.
  3. Retrouver toutes les instructions pour ajouter l'adresse de départ à chaque adresse relative.
  4. Retrouver toutes les instructions qui référencent des procédures et insérer les adresses à la place.

  Ex :

Module	**Longueur**	**Adresse de départ**
A	400	0
B	500	400
C	300	900

  Éditeur à 2 passes : 1re passe : étapes 1, 2 et 3 ; 2e passe : étape 4.
- Fichier objet = représentation binaire du programme comprenant six parties :

Entête	Instructions	Données	Info. relocalisation	Tables des symboles	Informations de mise au point

  - Entête : taille du fichier + position des 5 autres parties.
  - Instructions : non exécutable car références relatives.
  - Données : représentation binaire des chaînes de caractères, références aux variables externes.

Fichier C

```
main()
{
int a, b, c ;

a = 12 ;
b = 24 ;
c = a + b ;

printf("La valeur de c est : %d\n", c) ;
}
```

Assembleur Mips

```
$LC0 :

.byte 0x4c,0x61,0x20,0x76,0x61,0x6c,0x65,0x75
.byte 0x72,0x20,0x64,0x65,0x20,0x63,0x20,0x65
.byte 0x73,0x74,0x20,0x3a,0x20,0x25,0x64,0xa
.byte 0x0

.
.
.
li $2,0x0000000c # 12
sw $2,24($fp)
li $2,0x00000018 # 24
sw $2,28($fp)
lw $2,24($fp)
lw $3,28($fp)
addu $2,$2,$3
sw $2,32($fp)
la $4,$LC0
lw $5,32($fp)
jal printf
.
.
```

Assembleur Intel

```
.LC0 :
.string "La valeur de c est : %d\n"
.
.
.
movl $12,-4(%ebp)
movl $24,-8(%ebp)
movl -4(%ebp),%edx
addl -8(%ebp),%edx
movl %edx,-12(%ebp)
movl -12(%ebp),%eax
pushl %eax
pushl $.LC0
call printf
addl $8,%esp
.
.
```

Assembleur Sparc

```
.LLC0 :
.asciz "La valeur de c est : %d\n"

.
.
mov 12,%o0
st %o0,[%fp-20]
mov 24,%o0
st %o0,[%fp-24]
ld [%fp-20],%o0
ld [%fp-24],%o1
add %o0,%o1,%o0
st %o0,[%fp-28]
sethi %hi(.LLC0),%o1
or %o1,%lo(.LLC0),%o0
ld [%fp-28],%o1
call printf,0
.
.
```

FIG. 5.9 – Exemples d'assembleur.

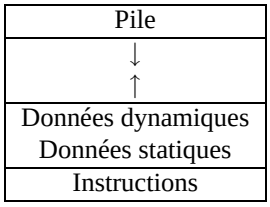
1200		
1100	move 50, R3	move 950, R3
1000		
	"bonjour"	"bonjour"
900	jump to 100	jump to 1000
800	call C	branch 900
700		
600		
	branch 300	branch 700
500	load Y, R2	load Y, R2
	3.141593	3.141593
400	jump to 150	jump to 550
	call B	branch 400
300		
200	load 50, R1	load 50, R1
100	branch 200	branch 200
	"salut"	"salut"
0	jump to 200	jump to 200

FIG. 5.10 – Ensembles des fragments de codes avant et après édition de liens. Ici on a arbitrairement supposé que la première instruction d'un programme se trouvait toujours à l'adresse 0.

  - Information de relocalisation : identifie les instructions et données qui dépendent d'adresses absolues (ex. : jump 1000).
  - Tables des symboles : table des étiquettes internes et externes (noms de fonctions).
  - Informations de mises au point : pour le débuggage (liens entre lignes sources et instructions, etc.).

- Éditeur de liens : permet d'assembler des fichiers interdépendants compilés séparément, il réalise 3 tâches :
  - recherche dans les bibliothèques de programmes pour retrouver les procédures utilisées par le programme ;
  - détermine les emplacements de chaque module et relocalise les instructions en ajustant les références absolues ;
  - résout les références entre fichiers.

  Évoquer pb : fonction non existante ou non dans la bibliothèque.
- Chargement = on passe de l'exécutable au processus : structure d'un processus en mémoire : 3 parties (pile, données, texte).



- Chargeur :
  - lit la taille du fichier et définit l'espace nécessaire ;
  - crée un espace d'adressage suffisamment grand pour contenir également la pile ;
  - copie les données et les instructions ;
  - copie sur la pile les arguments passés au programme ;
  - initialise les registres (tous à 0, sauf celui de la pile : premier emplacement libre) ;
  - effectue le saut vers la première instruction du *main*.

Chapitre 6

Pipeline

6.1 Étages d'un pipeline

- CPU = volumineux circuit séquentiel cadencé par une horloge.
- À un instant donné de l'exécution d'une instruction, l'instruction n'utilise que quelques circuits du CPU. L'idée est d'utiliser les autres circuits (UAL, chercheur d'instructions, décodeur, etc.) pour l'exécution d'autres instructions. On obtient ainsi un recouvrement partiel de l'exécution de plusieurs instructions, que l'on appelle *pipelining*. Pour ce faire le CPU est découpé en sous-circuits séquentiels, chacun pouvant à un instant donné traiter une partie d'une instruction. De cette manière, le CPU réalise un travail à la chaîne. Chaque maillon de la chaîne est alors un *étage* du pipeline.
- Décomposition de l'exécution d'une instruction en étapes élémentaires (pour une architecture simple où tous les opérandes sont dans les registres ou dans l'instruction). Bien évidemment, le traitement de toutes les instructions ne nécessite pas toutes ces étapes.

  1. Recherche de l'instruction (**IF** : *Instruction Fetch*) :
     - aller chercher la nouvelle instruction et la placer dans le registre d'instruction.
  2. Décodage de l'instruction et lecture des registres (**ID** : *Instruction Decode*) :
     - en fonction du code de l'instruction, générer les signaux pour l'UAL, la mémoire des registres, etc. ;
     - placer en entrée de la mémoire de registres l'adresse des registres concernés.
  3. Exécution et calcul de l'adresse effective(**EX** : *EXecution*) :
     - opération effectuée par l'UAL pour le calcul d'une valeur, d'une adresse (pour un saut).
  4. Accès à la mémoire et réalisation des branchements (**MEM** : *MEMory*) : chargement (*load)*, écriture (*store*), exécution des branchements et sauts.
  5. Mise à jour des registres (**WB** : *Write Back cycle*) : écriture des résultats dans les registres (+, -, ...) ou dépose d'une valeur dans un registre (*load*).

- Dans un CPU pipeliné, le CPU exécute les instructions telles qu'elles se suivent en mémoire. Après avoir cherché une instruction, le processeur va chercher tout de suite l'instruction suivante en mémoire pendant qu'il décode la première, etc..

Instr Num/cycle horloge	1	2	3	4	5	6	7	8	9
instr i	IF	ID	EX	MEM	WB				
instr i+1		IF	ID	EX	MEM	WB			
instr i+2			IF	ID	EX	MEM	WB		
instr i+3				IF	ID	EX	MEM	WB	
instr i+4					IF	ID	EX	MEM	WB

6.2 Registres de pipeline

– Pour faire du chemin de la figure 6.1 un pipeline, il est nécessaire d'ajouter des *registres pipelines* pour séparer les étages sinon, par exemple, la recherche de nouvelle instruction pourrait écraser le contenu du registre d'instruction avant que le décodage de l'instruction n'ai fini de se servir de sa valeur. La figure 6.2 présente le même pipeline après l'insertion de deux registres.

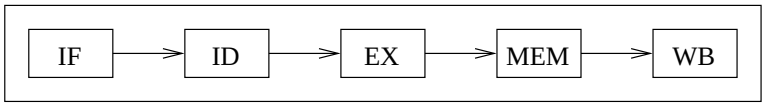


FIG. 6.1 – Étages du pipeline.

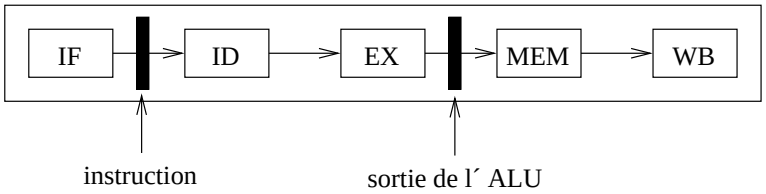
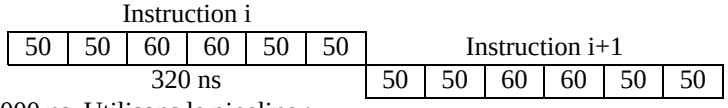


FIG. 6.2 – Étages du pipeline après insertion d'un registre pour contenir le code de l'instruction et un autre pour mémoriser les sorties de l'UAL.

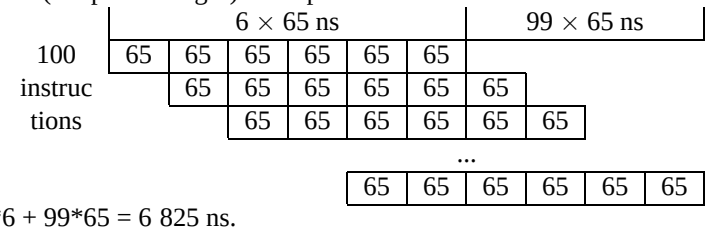
6.3 Performances d'un pipeline

– Considérons un pipeline à 6 étages de temps : 50 ns, 50 ns, 60 ns, 60 ns, 50 ns et 50 ns. Dans un premier temps, on n'utilise pas le pipeline. Combien de temps prennent l'exécution de 100 instructions ?



100 instructions : 32 000 ns. Utilisons le pipeline :
– le temps de chaque étage est alors le même (celui du plus lent car les autres étages doivent attendre) ;
– le temps de passage entre deux étages ne peut pas être instantané, il faut attendre une stabilisation des registres de pipeline, prenons ici 5 ns.

Temps d'un étage = MAX(temps des étages) + temps de stabilisation = 60 + 5 = 65 ns.



100 instructions = 65*6 + 99*65 = 6 825 ns.
Temps moyen traitement d'une instruction non pipeliné = 320 ns.
Temps moyen traitement d'une instruction pipeliné = 65 ns (dans notre exemple : 68,25 ns).
Accélération = 320/65 = 4,92.

– Un pipeline n'accélère pas le temps de traitement d'une instruction mais augmente le débit des instructions : dans un intervalle de temps on traite plus d'instructions.

6.4 Problèmes liés au pipeline

La mise en place d'un pipeline soulève plusieurs problèmes :

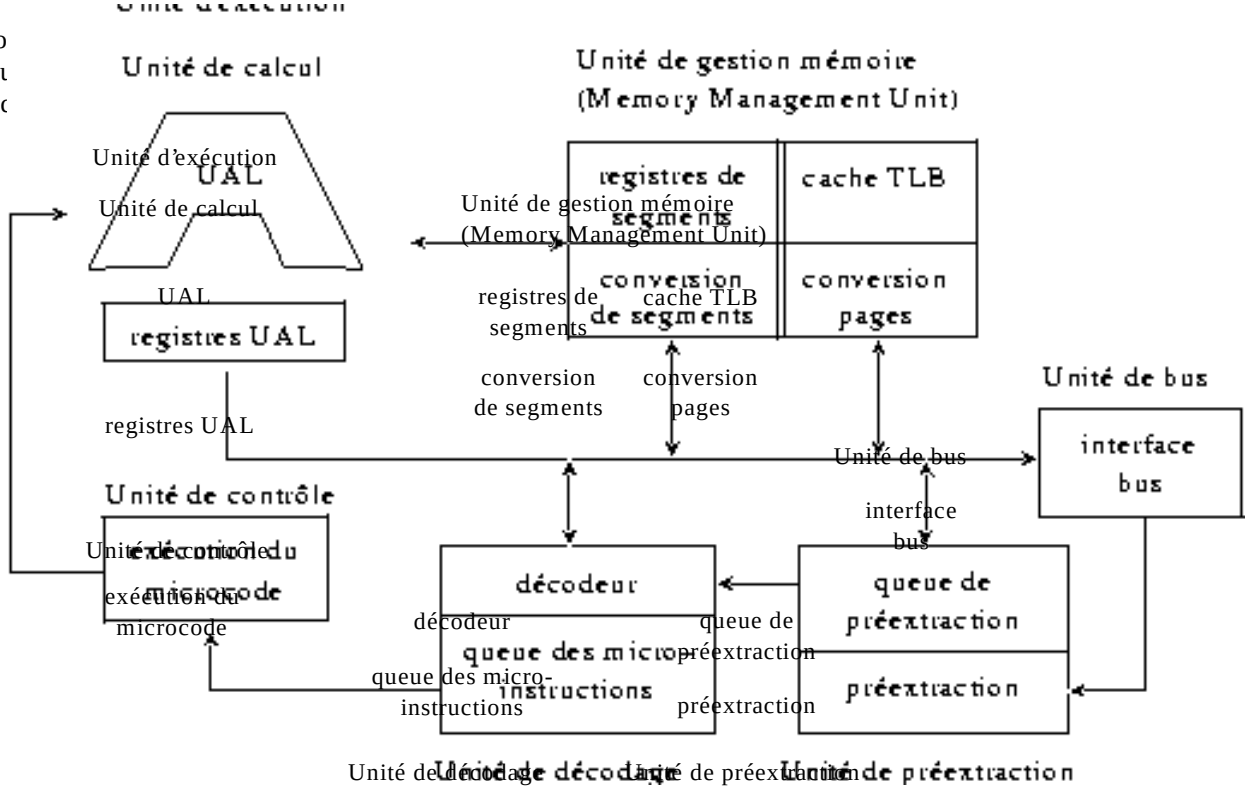
- le temps de traitement dans chaque unité doit être à peu près égal, sinon les unités rapides attendent les unités lentes ;
- aléas de branchement : s'il y a saut d'instruction, les instructions qui suivent le saut et qui sont en train d'être traitées dans les étages inférieurs le sont pour rien, il faudra vider le pipeline ; pour atténuer l'effet des branchements, on peut spécifier après le branchement des instructions qui seront *toujours* exécutées, comme c'est le cas dans l'assembleur SPARC ;
- conflits de dépendance, par exemple entre deux instructions consécutives travaillant sur la même zone mémoire (RAM ou registres) ; solution logicielle : le compilateur génère des instructions *nop* (ne fait rien) ; solution matérielle : détection des dépendances et mise en attente des instructions suivantes.

Chapitre 7

Unités d’un microprocesseur

7.1 Unités internes

Qu’il so
le processeu
complexes c



Unité de bus : fait l’interface entre le bus interne et le bus externe (qui ne sont pas forcément de même taille ni de même vitesse), il gère également la circulation interne entre unités. Il est sollicité d’une part par l’unité de contrôle pour la recherche des données en mémoire centrale, données placées ensuite dans les registres, et d’autre part par l’unité de préextraction pour la recherche d’instructions.

Unité de préextraction : dans un processeur pipeliné, dès que l’unité de bus ne travaille pas, elle demande la recherche de l’instruction suivante et la place dans la queue de préextraction, elle transmet l’instruction suivante à l’unité de décodage.

Unité de décodage : traduit chaque instruction assembleur en microcodes.

Unité de contrôle : exécution des microcodes en synchronisant les signaux envoyés à l’UAL ou à l’interface de bus pour la recherche de données.

Unité arithmétique et logique : contient des registres et réalise les opérations arithmétiques, logiques, les décalages et comparaisons.

Unité de gestion mémoire : sur demande de l'unité de contrôle, elle convertit une adresse virtuelle en adresse physique. Par ce mécanisme tout programme est découpé en « pages » : des adresses consécutives dans le programme ne sont pas forcément consécutives en mémoire centrale. De plus tout le programme n'est pas forcément chargé en mémoire mais seulement les pages utiles.

Revenons sur les unités de décodage et de gestion mémoire.

7.2 Unité de décodage

- Le processeur comprend un langage qui lui est propre. Il doit traduire chaque instruction assembleur en une suite de micro-instructions ou microcodes. Chaque microcode correspond à un ensemble de signaux à envoyer aux différentes unités. Ainsi, par exemple, les processeurs Pentium de la marque Intel et les processeurs K6 et K7 de la marque AMD comprennent le même langage d'assemblage mais traduisent chaque instruction de ce langage en microcodes qui leur sont propres et qui reflètent les architectures internes et différentes de ces processeurs.
- L'unité de décodage est une fonction booléenne qui prend en entrées les signaux correspondant aux bits du code de l'instruction assembleur et donne en sorties les microcodes, c'est-à-dire les signaux pour l'UAL, la MMU, etc.
- Cette fonction booléenne est réalisée soit par un circuit logique, soit par une ROM qui contient la table de vérité de la fonction : instruction du décodeur = adresse dans la ROM, donnée de chaque ligne mémoire = valeurs des signaux de sorties.

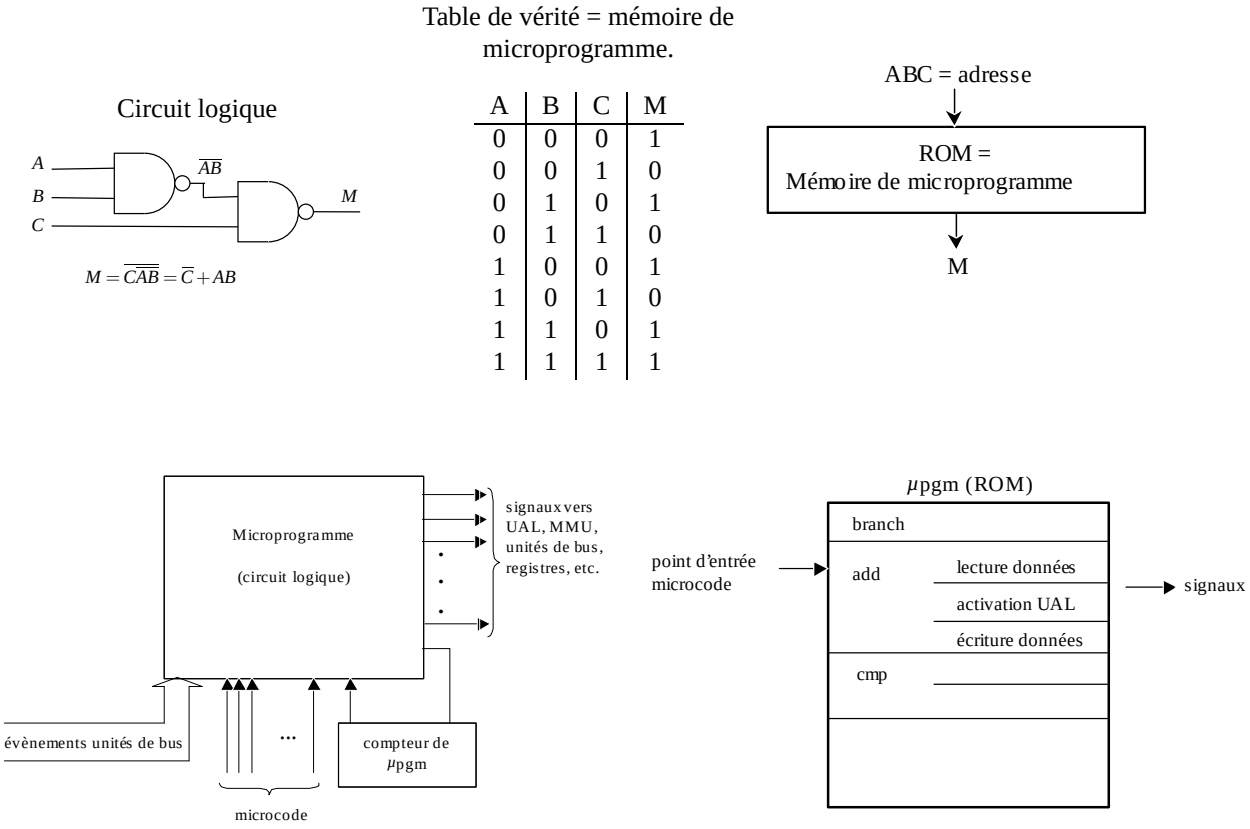


FIG. 7.1 – Exemple d'équivalence circuit logique/ROM.

– **ROM** : moins rapide que le circuit (temps d'accès mémoire), plus volumineux, plus facile à réaliser. De plus, l'ajout d'une instruction demande simplement d'étendre la ROM et d'ajouter les nouvelles lignes à la table de vérité. Ce système est utilisé dans les processeurs CISC car leur jeu d'instructions est énorme. On parle alors de jeu d'instructions microprogrammé.
Circuit : plus rapide, peu volumineux (moins de portes que la ROM). Ce système est utilisé dans les processeurs processeurs RISC. On parle alors de jeu d'instructions câblé.
Actuellement, la différence entre RISC et CISC s'estompe avec des processeurs utilisant les deux techniques : les microcodes les plus fréquents sont câblés et les autres microprogrammés. Pour un exemple d'équivalence circuit logique/ROM, voir la figure 7.1.

7.3 MMU (*Memory Management Unit*)

La MMU est la réponse à deux problèmes :
– **Problème 1** : une instruction peut faire référence à une adresse interdite (lecture dans la zone d'un autre programme ou écriture dans la zone des instructions du même programme). Comment protéger ces zones ?
Solution : découper le programme en zones (appelés segments) $\Rightarrow$ segments des instructions, des données, de la pile. Chaque segment possède des droits d'accès (lecture simple pour le segment des instructions et lecture/écriture dans les segments de données et pile). Une table (table des segments) contient les adresses minimum et maximum de chaque segment. Chaque adresse fournie à l'unité de bus est préalablement testée par la MMU et le programme est interrompu si l'adresse ne correspond pas à celle d'un segment ou si l'opération associée est interdite.
– **Problème 2** : considérons une machine ayant un bus d'adresses de 32 bits et une mémoire RAM de 32 M octets (32 Mo=2^{25} octets, d'où 25 bits significatifs par adresse). En RAM les adresses physiques vont de 0 à $2^{25}-1$. Sur le bus d'adresses peuvent circuler des adresses allant de 0 à $2^{32}-1$ soit un espace d'adressage de 4 Go.
Si les adresses dans les programmes tiennent compte de l'adressage possible en RAM :
  – ajout/diminution de RAM difficile (les programmes doivent être recompilés) ;
  – gaspillage de 7 signaux du bus d'adresses et moins de programmes peuvent être activés en même temps (car l'espace d'adressage est limité à la capacité de la RAM).

  Solution : travailler avec un espace d'adressage de 4 Go appelé espace d'adressage virtuel. Cet espace est découpé en pages de taille fixe (souvent 4 ko). Certaines pages se trouvent en RAM, d'autres se situent en mémoire auxiliaire (disque dur), chaque fois qu'une adresse virtuelle est générée, la MMU agit de la manière suivante :
  – soit l'information est en mémoire centrale auquel cas l'adresse virtuelle est traduite en adresse physique par l'intermédiaire de la TLB (**Table Lookaside Buffer**) sorte de table de conversion ;
  – soit l'adresse ne s'y trouve pas auquel cas, le programme est interrompu et le processeur exécute une fonction du système d'exploitation, celle-ci est chargée de placer la page en mémoire centrale, la MMU reprend ensuite la traduction d'adresse interrompue.

7.4 Accroître les performances

Pour rendre plus efficace les processeurs, des unités ont été ajoutées :
– **Caches de données** et d'instructions séparés pour permettre des accès simultanés.
– **Unité FPU** (*Floating Point Unit*) qui réalise les calculs flottants. Cette unité renferme parfois une petite ROM contenant des constantes : 0 (pour arrondir à 0), 1, π, $\log_2(e)$, $\log_2(10)$, $\log_e(2)$; ou encore des circuits réalisant des fonctions classiques (fonctions trigonométriques, logarithmiques, racine carrée).
– **Unité d'anticipation de branchement** qui, pour limiter les purges du pipeline lors des branchements, tente de prédire, en fonction des exécutions passées, si un branchement aura lieu ou non. Cette unité contient un cache, le BTB (*Branch Target Buffer*) contenant les instructions de branchement déjà rencontrées et un bit ou plusieurs bits indiquant pour chaque instruction si le branchement a eu lieu.
– Unités spécialisées pour le chargement, rangement, UAL à entrées multiples, etc.
– De plus, les processeurs actuels contiennent **plusieurs pipelines** qui peuvent alors exécuter des instructions en parallèle. Ces processeurs sont dits **superscalaires**. Toutefois, la phase de décodage doit déterminer si les ins-

tructions courantes peuvent s'appareiller et dans le cas contraire mettre en attente certaines instructions conflictuelles en bloquant un ou plusieurs pipelines. Notons que les pipelines sont généralement dédiés à des instructions particulières, ainsi le Pentium II possède 2 pipelines entiers et 1 pipeline flottant. Pour tirer pleinement parti des pipelines, les caches L2 permettent plusieurs accès simultanés.

– Pour finir, les processeurs les plus récents évitent au maximum les mises en attente dues au conflits en exécutant les microcodes dans le désordre : ***Out-of-Order Execution*** opposé à *In-Order Execution*. Ainsi, une instruction en attente peut laisser la place à une instruction suivante et n'être exécutée qu'ensuite.

7.5 Exemples d'architecture

7.5.1 Architecture R10.000 (MIPS)

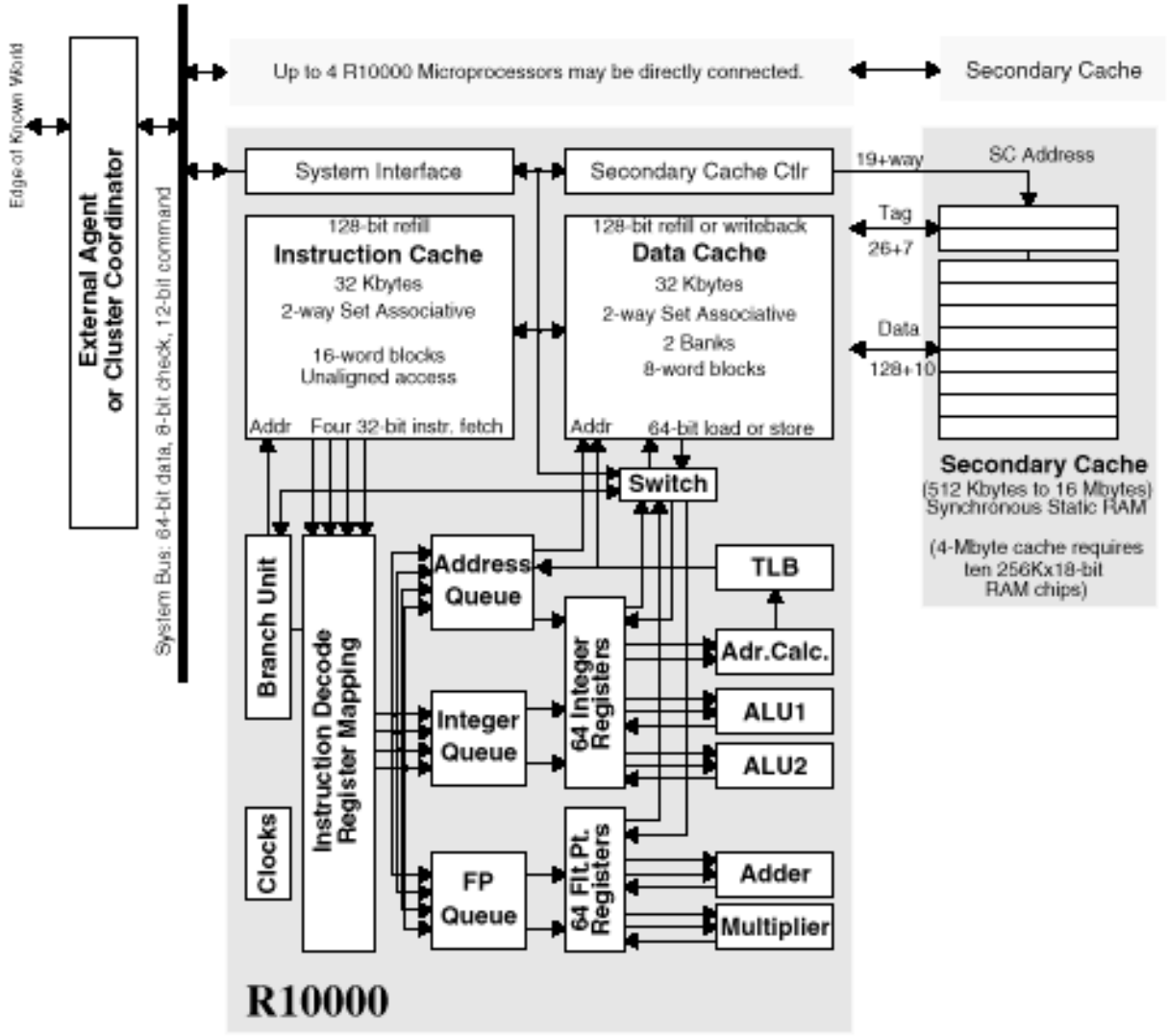
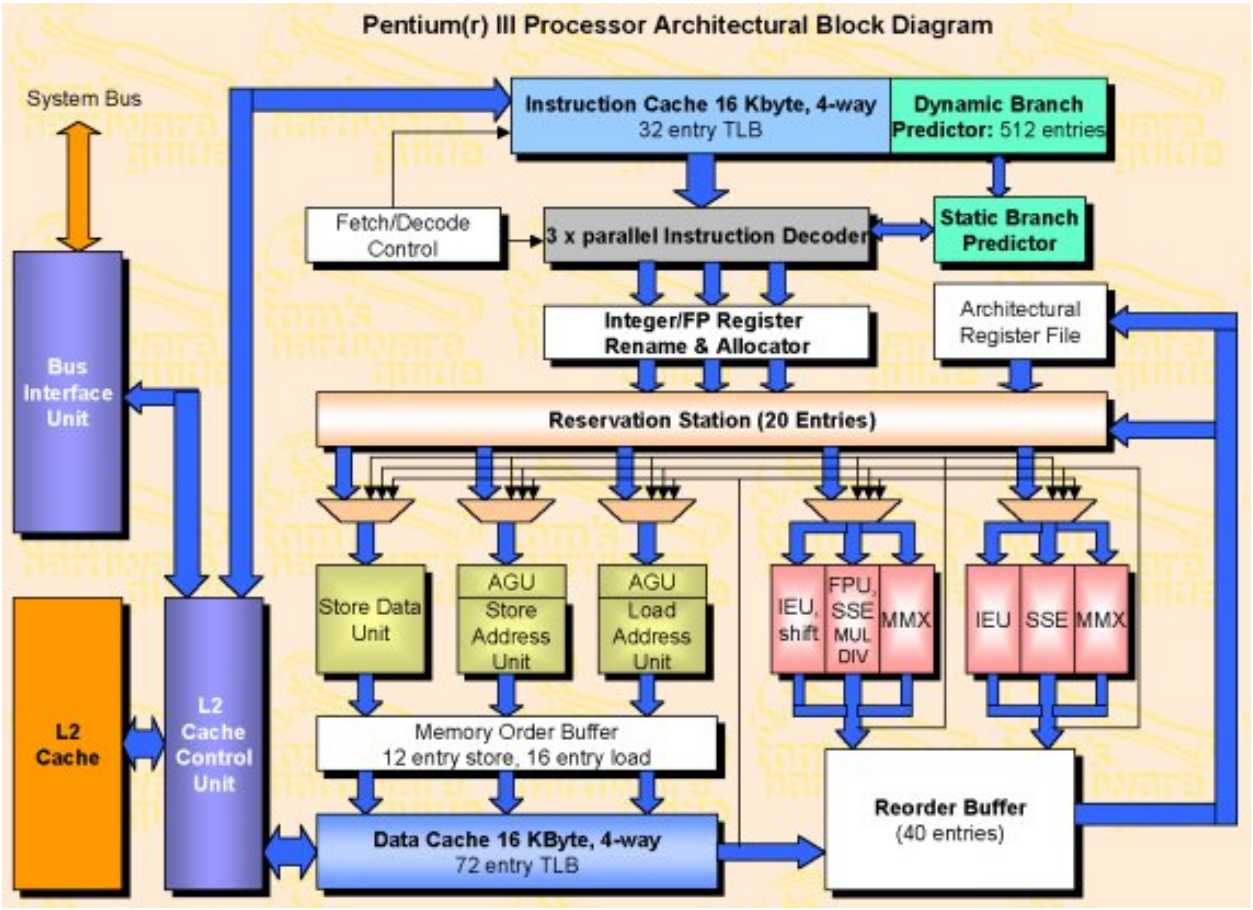


Figure 1-5 Block Diagram of the R10000 Processor

– Superscalaire *out-of-order*, RISC.
– Format d'instructions fixe sur 64 bits.

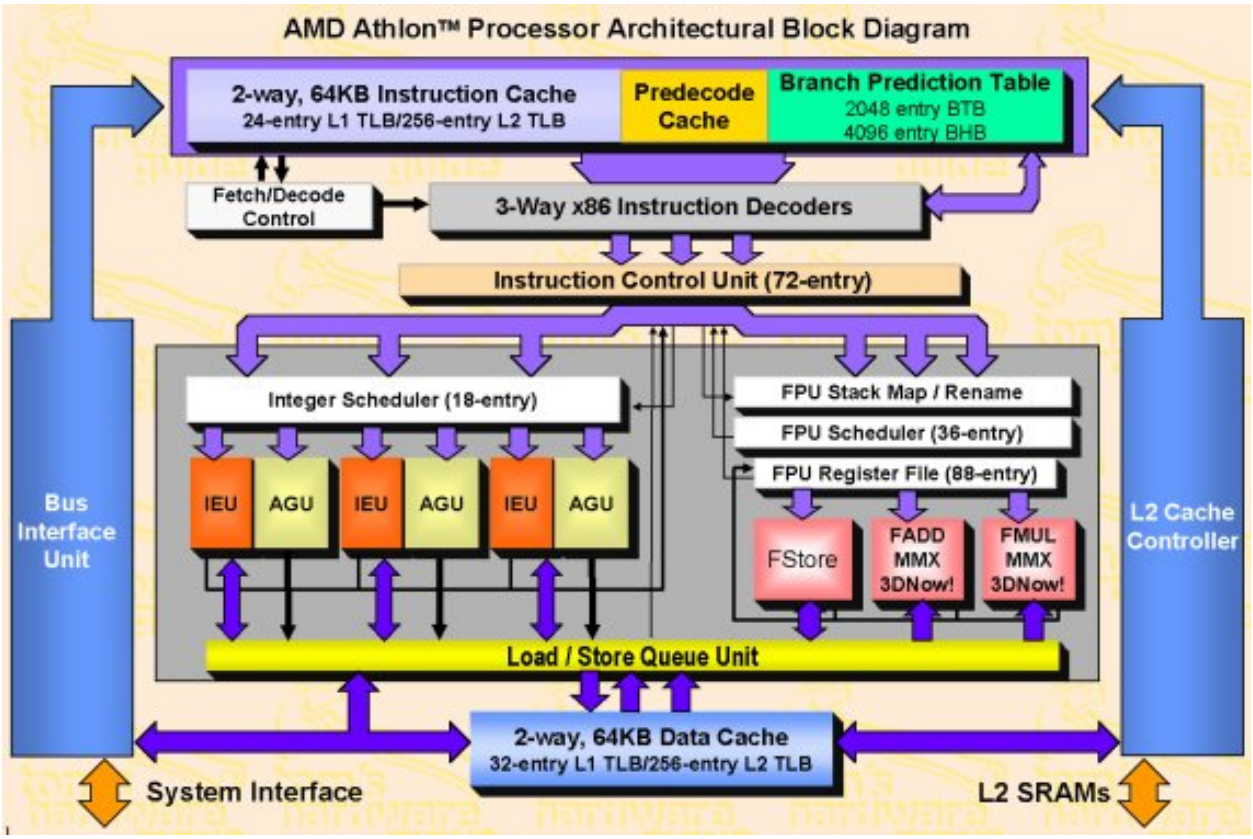
– 5 pipelines : 2 entiers (ALU1, ALU2), 2 flottants (1 addition, 1 multiplication), 1 pour les accès mémoire load/store.
– 4 instructions décodées à chaque cycle horloge.
– TLB : cache associatif par blocs de 64 entrées (1 entrée = 2 pages), adresses physiques sur 40 bits ⇒ 1 Teraoctets.
– 2 caches associatifs par blocs de 32 Ko.

7.5.2 Le pentium III (Intel)



– Superscalaire *out-of-order*.
– Format 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 entier : 12 à 17 étages ; pipeline flottant : environ 25 étages.
– Cache L2 : 512 Ko.
– AGU : *Adress Generation Unit*
– IEU : *Integer Execution Unit*
– BTB : *Branch Target Buffer*
– BHB : *Branch History Buffer*

7.5.3 L'Athlon (AMD)



- Superscalaire *out-of-order*.
- Format d'instructions variable.
- 6 unités de décodage parallèles (3 microprogrammées, 3 câblées) mais seules 3 peuvent fonctionner en même temps.
- Jusqu'à 72 microcodes en attente.
- Plusieurs pipelines mais au mieux 9 instructions exécutées en même temps.
- Pipeline entier : 10 étages ; pipeline flottant : 15 étages.
- Cache L2 : 512 Ko.