Circuits et architecture des ordinateurs

Année 2018/2019

Université Paris Diderot

Olivier Carton

Version du 19 févr. 2019

1 Circuits et architecture des ordinateurs en M1

Cours référencé sur le site de l'Université Numérique Ingénierie et Technologie (Unit)

- Cours le vendredi en amphi 9E (HaF) de 10h45 à 12h45
- Bibliographie
- Travaux dirigés (logisim)
  - Mercredi en salle 2031 de 13h à 15h (Matthieu Picantin)
  - Mercredi en salle 2031 de 8h30 à 10h30 (Mihaela Sighireanu)
- Ce support de cours en PDF
- Références
- Années 2006/2007, 2007/2008, 2008/2009, 2009/2010, 2010/2011, 2011/2012, 2012/2013, 2013/2014, 2014/2015, 2015/2016 et 2016/2017

- Présentation du cours
- Cours n° 1 : historique et représentation des données
  - galeries de photos
  - fabrication d'un processeur
  - entiers
  - entiers signés
- Cours n° 2 : représentation des données (suite), transistors, portes
  - réels (norme IEEE 754)
  - caractères (ASCII et Unicode)
  - logique de Boole
  - table de vérité
  - tableaux de Karnaugh
  - transistors
  - portes logiques (inverseur, nand, nor)
- Cours n° 3 : additionneurs
  - circuits élémentaires
  - additionneurs
    - semi-additionneur
    - additionneur complet
    - additionneur par propagation de retenue (ripple-carry adder)
    - calcul des indicateurs
    - soustraction
- Cours n° 4 : additionneurs (suite)
  - additionneur par anticipation de retenue (carry-lookahead adder)
  - additionneur récursif
  - additionneur hybride
  - additionneur par sélection de retenue
- Cours n° 5 : mémoire
  - mémoire statique/mémoire dynamique
  - bascule RS

  - bascule D
  - mémoire 4 × 3 bits
- Cours n° 6 : circuits séquentiels et architecture générale d'un micro-processeur
  - principe des circuits séquentiels
  - construction d'une guirlande
  - cas d'un automate fini
  - modèle de von Neumann
    - unité de contrôle
    - unité de traitement
    - mémoire
- Cours n° 7 : description du LC-3
  - registres
  - organisation de la mémoire
  - jeu d'instructions du processeur LC-3
  - chemins de données du LC-3
- Cours n° 8 : programmation en assembleur du LC-3
  - longueur d'une chaîne
  - mutiplication non signée, signée et logarithmique
  - addition 32 bits
- Cours n° 9 : appels de sous-programmes, pile
  - appels de sous-programme
  - pile
  - sauvegarde des registres
  - tours de Hanoï
- Cours n° 10 : appels systèmes et interruptions
  - entrées/sorties
  - appels systèmes
  - interruptions
- Cours n° 11 : autres architectures
  - processeurs 80x86
  - comparaison CISC/RISC
  - architecture IA-64
- Cours n° 12 : pipeline
  - principe
  - étages
  - réalisation
  - aléas
- Cours n° 13 : gestion de la mémoire
  - mémoires associatives
  - mémoire virtuelle
  - mémoires cache
- Examen : vendredi 11 janvier de 8h30 à 11h30 en amphi 13E Le seul document autorisé est une feuille de memento.

2 Historique

2.1 Historique général

Quelques dates clés

500 av JC
apparition des bouliers et abaques
1632
invention de la règle à calcul
1642
Pascal invente la *Pascaline*
1833
machine de Babbage
1854
publication par Boole d'un ouvrage sur la logique
1904
invention du tube à vide
1937
article d'Alan Turing sur la calculabilité : machines de Turing
1943
création du ASCC Mark 1 (Harvard - IBM) : Automatic Sequence-Controlled Calculator
1946
construction de l'ENIAC
1947
invention du transistor (Bell)
1955
premier ordinateur à Transistors : TRADIC (Bell)
1958
premier circuit intégré (Texas Instrument)
1964
langage de programmation BASIC
1965
G. Moore énonce la loi qui porte son nom : loi de Moore
1969
système d'exploitation MULTICS puis UNIX (Bell)
1971
premier microprocesseur : 4004 d'Intel (4 bits, 108 KHz, 2300 transistors)
1972
microprocesseur 8008 d'Intel (8 bits, 200 KHz, 3500 transistors)
1973
langage C pour le développement d'UNIX
1974
premier microprocesseur Motorola : 6800 (8 bits)
1974
microprocesseur 8080 d'Intel

2.2 Historique des micro-processeurs

2.2.1 Références

- Histoire des micro-processeurs de 1971 à 1996
- Collection de micro-processeurs
- Autre collection de micro-processeurs
- Histoire des premiers micro-processeurs
- Base de données des micro-processeurs
- Base de données des micro-processeurs
- Site d'information sur les micro-processeurs
- Musée en français de la micro-informatique

2.2.2 Principaux micro-processeurs

Intel 4004
: 1971, 4 bits, 108 KHz, 2300 transistors

Intel 8008
: 1972, 8 bits, 200 KHz, 3500 transistors

Intel 8080
: 1974, 8 bits, 6000 transitors

Motorola 6800 (Photos avec boitier plastique et boitier céramique)
: 1974, 8 bits, 2Mhz, 7000 transistors

MOS Technology 6502
: 1975, 1 MHz, 8 bits

Zilog Z80
: 1976, 2Mhz, 8 bits

Intel 8086
: 1978, 16 bits

Motorola 6809
: 1978, 8 bits

Intel 8088
: 1979, 16 bits

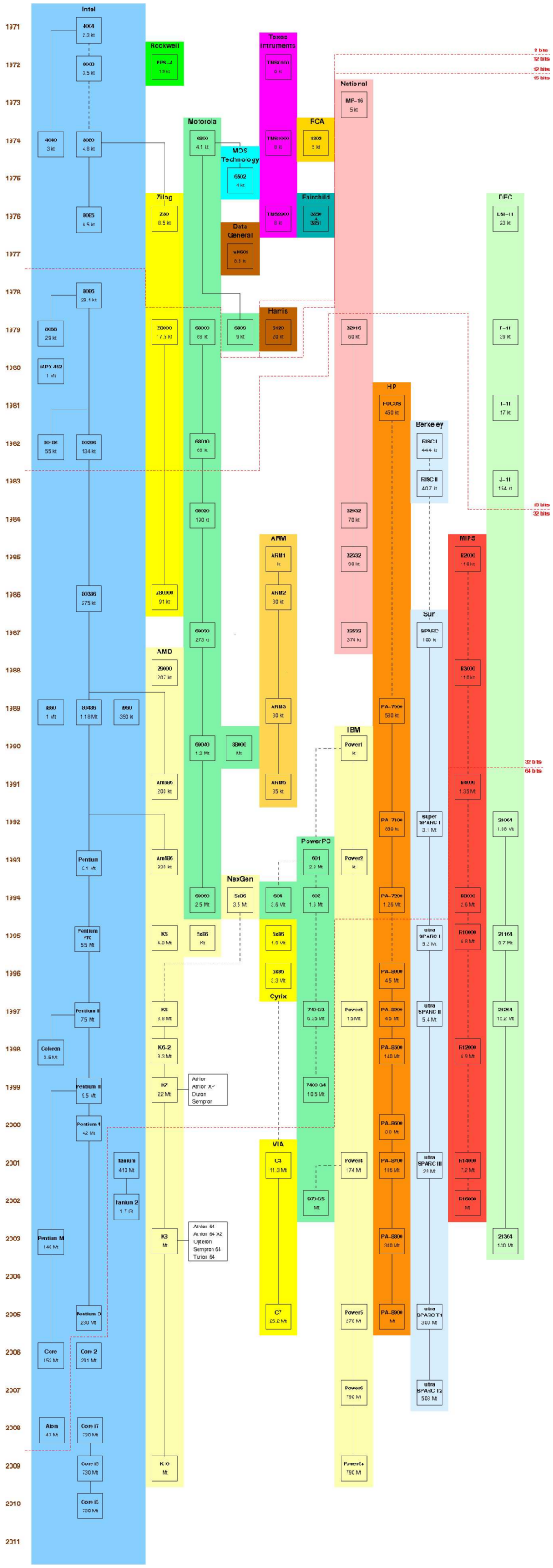
Motorola 68000
: 1979, 16 bits, 68000 transistors

Pentium
: 1993, CISC

PowerPC
: 1993, RISC

Sparc
: 1995, RISC

2.2.3 Généalogie des micro-processeurs


Intel
Rockwell
Texas Instruments
National
Motorola
RCA
MOS Technology
Fairchild
Zilog
Data General
Harris
DEC
HP
Berkeley
ARM
MIPS
Sun
AMD
IBM
PowerPC
NexGen
Cyrix
VIA
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011

Généalogie des micro-processeurs

2.2.3.1 Sources

- Intel
  - 4004
  - iAPX 432
  - Intel i860
  - Intel i960
  - Itanium
  - Core
  - Core 2
- Motorola
  - 68000
- F8
- IMP-16
- DEC (T-11)
- ARM
- PA-RISC (wikipedia)
- PA-RISC (openpa)
- RISC
- MIPS
- PowerPC
- DEC alpha
- SPARC
- National 320xx
- Comparatif de quelques architectures

3 Représentation des données

Dans un ordinateur, toute l'information est sous forme de bits qui sont regroupés en octets. Il faut donc qu'il y ait un codage de cette information. Ce codage dépend bien sûr du type des données. Cette partie décrit les codages les plus utilisés pour les types de base, c'est-à-dire les entiers, les nombres flottants et les caractères.

3.1 Entiers

3.1.1 Entiers positifs

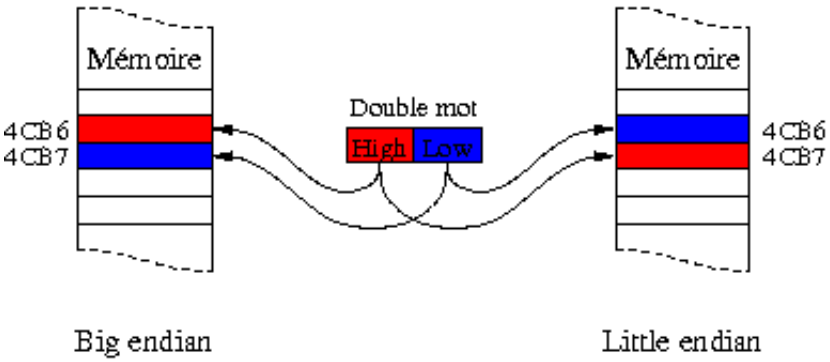
Les entiers positifs sont toujours codés en base 2. Une suite $b_{k-1},\ldots,b_0$ de k bits représente l'entier n donné par la formule suivante.

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

Avec k bits, on peut donc représenter tous les entiers de l'intervalle 0 … 2^k-1. Le nombre de bits utilisés pour coder les entiers dépend de la machine. C'est encore très souvent 32 bits mais l'apparition des micro-processeurs 64 bits rend le codage 64 bits de plus en plus fréquent.

3.1.2 Big endian/Little endian

Cette caractéristique décrit dans quelle ordre sont placés les octets qui représentent un entier. Dans le mode *big endian* les octets de poids fort sont placés en tête et occupent donc des emplacements mémoire avec des adresses plus petites. Dans le mode *little endian*, les octets de poids faibles sont au contraire placés en tête. Dans le cas d'entiers de 32 bits, il existe encore des modes mixtes. Cette terminologie provient du livre *Les voyages de Gulliver* de J. Swift.



Big et little endian

Le mode *big endian* accélère les opérations qui nécessitent de regarder en premier les bits de poids forts comme la recherche du signe, la comparaison de deux entiers et la division. Au contraire le mode *little endian* favorise les opérations qui commencent par les bits de poids faible comme l'addition et la multiplication.

Big endian	Mixed endian	Little endian
Motorola 68xx, 680x0 IBM Hewlett-Packard SPARC	Motorola PowerPC Silicon Graphics MIPS	Intel DEC VAX

3.1.3 Codage BCD (Binary Coded Decimal)

Il s'agit d'une représentation surtout utilisée dans les premiers temps de l'informatique. La représentation BCD d'un entier n est obtenue de la manière suivante. L'entier n est écrit en décimal puis chaque chiffre décimal entre 0 et 9 est ensuite codé sur 4 bits. Il faut donc (k+1)/2 octets pour coder un nombre ayant k chiffres décimaux. Le seul intérêt de cette représentation est de faciliter l'affichage en base 10 de l'entier. Certains processeurs possédaient quelques instructions spécifiques pour traiter, en particulier pour l'addition, les entiers codés en BCD.

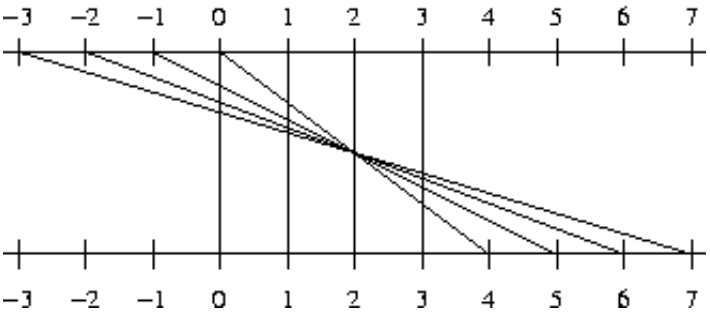
3.1.4 Entiers signés

Il existe plusieurs façons de coder les entiers signés. La représentation *avec un bit de signe* est la plus simple mais elle a trop d'inconvénients pour être utilisée en pratique. La représentation *biaisée* est uniquement utilisée pour le codage des exposants des flottants. La représentation *en complément à 2* est utilisée par tous les ordinateurs car elle facilite de nombreuses opérations.

3.1.4.1 Représentation avec un bit de signe

Dans cette représentation, le bit de poids fort indique le signe du nombre avec la convention 0 pour positif et 1 pour négatif. Les bits restants sont utilisés pour coder la valeur absolue du nombre comme un nombre positif. Cette représentation semble la plus naturelle mais elle n'est pas utilisée en pratique car elle souffre de nombreux défauts dont le principal est de compliquer les additions et soustractions. En effet, pour additionner deux nombres codés de cette façon, il est nécessaire de faire une addition ou une soustraction suivant que ces deux nombres sont du même signe ou de signes différents.

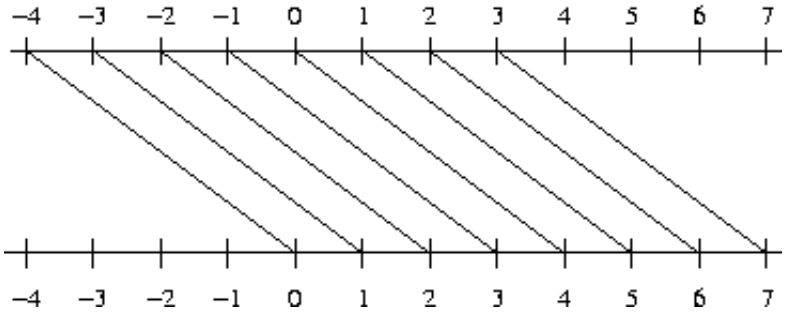
Avec k bits, cette représentation permet de représenter tous les entiers de de l'intervalle $-2^{k-1}+1$ … $2^{k-1}-1$. L'entier 0 a alors deux codages.



Représentation avec bit de signe sur 3 bits

3.1.4.2 Représentation biaisée

Cette représentation est utilisée pour le codage des exposants des nombres flottants. Avec k bits, cette représentation permet de représenter tous les entiers de l'intervalle -2^{k-1} … $2^{k-1}-1$. Chaque entier n de cet intervalle est codé par le codage de $n + 2^{k-1}$ comme un entier positif.



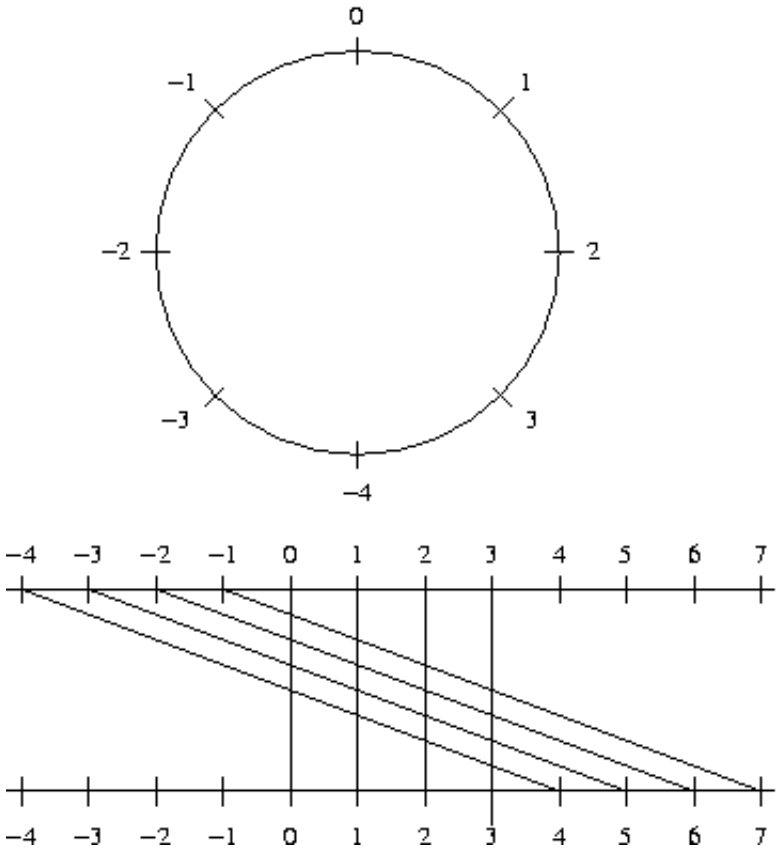
Représentation biaisée (avec un biais de 4) sur 3 bits

3.1.4.3 Représentation en complément à 2

Cette représentation est la plus importante car c'est elle qui est utilisée dans les ordinateurs. Malgré son apparente complexité, elle simplifie de nombreuses opérations sur les entiers.

Avec k bits, cette représentation permet de représenter tous les entiers de de l'intervalle -2^{k-1} … $2^{k-1}-1$. Les entiers de 0 à $2^{k-1}-1$ sont codés comme les nombres positifs. Le bit de poids fort de leur représentation est donc toujours 0. Un nombre négatif n de l'intervalle -2^{k-1} … -1 est représenté par le codage de $n+2^k$ (qui appartient à l'intervalle 2^{k-1} …2^k-1) sur k bits.

Dans la représentation en complément à 2, les représentations dont le bit de poids fort est 0 sont utilisées par les nombres positifs. Au contraire, les représentations dont le bit de poids fort est 1 sont utilisées par les nombres négatifs. Le bit de poids fort se comporte donc comme un bit de signe. Ceci facilite le test de signe d'un entier signé.



Représentation en complément à 2 sur 3 bits

3.1.4.3.1 Calcul de l'opposé

Étant donné un entier n positif ou négatif représenté en complément à 2, l'algorithme suivant permet de calculer la représentation en complément à 2 de son opposé -n. Ceci est bien sûr possible pour toutes les valeurs de l'intervalle -2^{k-1} … $2^{k-1}-1$ sauf pour -2^{k-1} dont l'opposé n'est plus dans l'intervalle.

L'algorithme est le suivant. Soit n' la représentation en complément à 2 de n. L'algorithme comporte les deux étapes suivantes qui utilisent des opérations présentes dans tous les micro-processeurs et facilement réalisables avec des portes logiques.

1. Calculer le complément bit à bit de n'.
2. Ajouter 1 au résultat de l'étape précédente.

Pour expliquer cette algorithme, on remarque que le complément bit à bit de m donne la différence entre le nombre écrit avec que des 1 et n', c'est-à-dire la valeur 2^k-n'-1. Le résultat de l'algorithme est donc 2^k-n'. Le tableau ci-dessous permet de vérifier que quel que soit le signe de n, l'algorithme donne le bon résultat. Soit n un entier positif ou négatif et soit n' son codage en complément à 2.

Nombre n	Codage n' de n	Résultat	Nombre codé
$0 \le n \le 2^{k-1}-1$	n' = n	2^k-n' = -n	-n
$-2^{k-1}+1 \le n \le -1$	n' = n+2^k	2^k-n'	-n
n = -2^{k-1}	n' = 2^{k-1}	n' = 2^{k-1}	n

3.1.4.3.2 Calcul de la somme

Un des grands avantages de la représentation en complément à 2 est de faciliter les additions et soustractions. En effet, l'addition de deux nombres m et n se fait simplement en additionnant leur représentations m' et n'.

Le tableau ci-dessous récapitule les différents cas de l'addition suivant les signes des opérandes. Soient m et n deux entiers signés représentés sur k bits. On distingue donc les trois cas : le cas m et n positifs, le cas m positif et n négatif et le cas m et n négatifs. Le quatrième cas est omis du tableau car il est symétrique du cas m positif et n négatif. Pour chacun des cas, on donne les codages respectifs m' et n' de m et n, puis on distingue à nouveau deux sous-cas suivant la valeur de la somme m'+n'. Le résultat de l'addition de m' et n' n'est pas nécessairement m'+n' puisque m'+n' peut être supérieur à 2^k et ne pas s'écrire sur k bits. Le résultat de cette addition est m'+n' si $0 \leq m'+n' \leq 2^k-1$ et m'+n'-2^k si $2^k \leq m'+n' \leq 2^{k+1}-1$. Les deux quantités C_k et C_{k-1} données dans une des colonnes sont les deux retenues (carry) calculées par un additionneur pour les k et k-1 bits des deux nombres.

Nombre m	Codage m'	Nombre n	Codage n'	Cas	Résultat	Nombre codé	C_k C_{k-1}	Commentaire
m positif $0 \leq m \leq 2^{k-1}-1$	m' = m	n positif $0 \leq n \leq 2^{k-1}-1$	n' = n	$0 \leq m'+n' \leq 2^{k-1}-1$	m'+n'	m+n	0 0	Résultat positif
				$2^{k-1} \leq m'+n' \leq 2^k-2$	m'+n'	m+n-2^k	0 1	Débordement car $m+n \geq 2^{k-1}$
m positif $0 \leq m \leq 2^{k-1}-1$	m' = m	n négatif $-2^{k-1} \leq n \leq -1$	n' = n + 2^k	$2^{k-1} \leq m'+n' \leq 2^k-1$	m'+n'	m+n	0 0	Résultat négatif car $\|m\| < \|n\|$
				$2^k \leq m'+n' \leq 2^k+2^{k-1}-1$	m'+n'-2^k	m+n	1 1	Résultat positif car $\|m\| > \|n\|$
m négatif $-2^{k-1} \leq m \leq -1$	m' = m + 2^k	n négatif $-2^{k-1} \leq n \leq -1$	n' = n + 2^k	$2^k \leq m'+n' \leq 2^k+2^{k-1}-1$	m'+n'-2^k	m+n+2^k	1 0	Débordement car $m+n < -2^{k-1}$
				$2^k+2^{k-1} \leq m'+n' \leq 2^{k+1}-1$	m'+n'-2^k	m+n	1 1	Résultat négatif

3.1.4.3.3 Exemple de calculs de somme

On se place avec k = 3 bits. Les nombres représentables en compléments à deux sont donc les entiers de -4 à 3. La table ci-dessous donne pour chaque paire de représentation, la somme et sa valeur. Les valeurs des deux retenues C_2 et C_3 sont codées par la couleur du fond. Les couleurs jaune et rouge correspondent aux débordements, c'est-à-dire à $C_2 \oplus C_3 = 1$.

$C_2 = 0$		$C_2 = 1$	
$C_3 = 0$	$C_3 = 1$	$C_3 = 0$	$C_3 = 1$

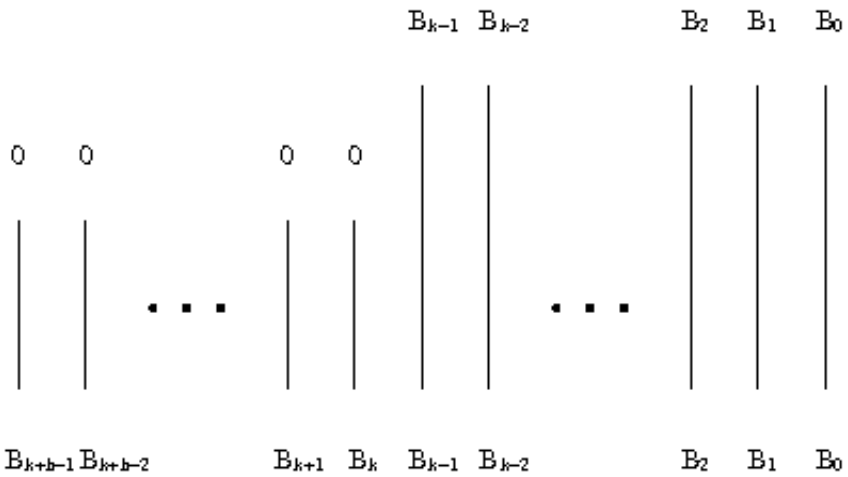
+	**100 -4**	**101 -3**	**110 -2**	**111 -1**	**000 0**	**001 1**	**010 2**	**011 3**
100 -4	000 0	001 1	010 2	011 3	100 -4	101 -3	110 -2	111 -1
101 -3	001 1	010 2	011 3	100 -4	101 -3	110 -2	111 -1	000 0
110 -2	010 2	011 3	100 -4	101 -3	110 -2	111 -1	000 0	001 1
111 -1	011 3	100 -4	101 -3	110 -2	111 -1	000 0	001 1	010 2
000 0	100 -4	101 -3	110 -2	111 -1	000 0	001 1	010 2	011 3
001 1	101 -3	110 -2	111 -1	000 0	001 1	010 2	011 3	100 -4
010 2	110 -2	111 -1	000 0	001 1	010 2	011 3	100 -4	101 -3
011 3	111 -1	000 0	001 1	010 2	011 3	100 -4	101 -3	110 -2

3.1.4.4 Extension signée et non signée

Avec toutes les représentations étudiées, un nombre signés ou non qui peut être représenté sur k bit peut encore être représenté avec k+h bits pour tout h ≥ 0. On étudie ici comme calculer cette représentation sur k+h bits pour la représentation des nombres non signés et pour la représentation en complément à 2 des nombres signés.

3.1.4.4.1 Extension non signée

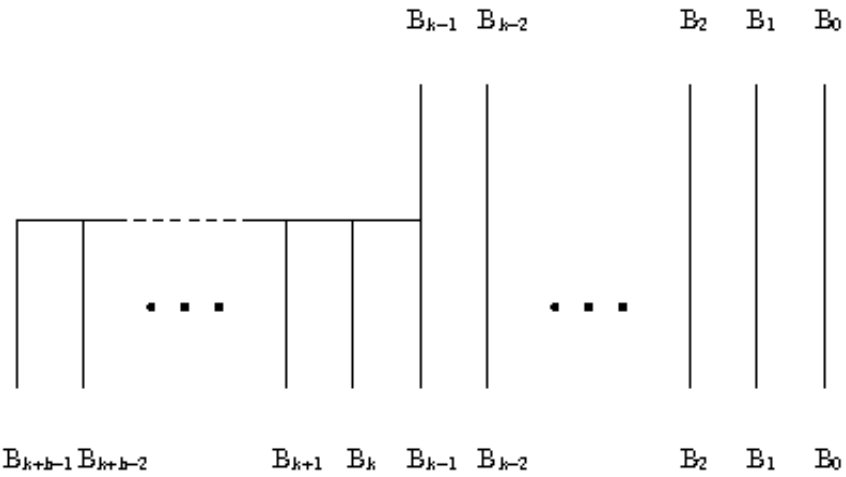
Soit un nombre n positif ayant $b_{k-1}\dots b_0$ comme représentation non signée sur k bits. La représentation non signée de n sur k+h bits est simplement la représentation $0\dots0b_{k-1}\dots b_0$ obtenue en ajoutant h chiffres 0 devant la représentation à k bits.



Extension non signée de k bits à k+h bits d'un entier

3.1.4.4.2 Extension signée

Soit un nombre n positif ou négatif ayant $b_{k-1}\dots b_0$ comme représentation en complément à 2 sur k bits. Si n est positif, la représentation en complément à 2 de n sur k+h bits est la représentation $0\dots0b_{k-1}\dots b_0$ obtenue en ajoutant h chiffres 0 devant la représentation à k bits. Si n est négatif, la représentation en complément à 2 de n sur k+h bits est la représentation $1\dots1b_{k-1}\dots b_0$ obtenue en ajoutant h chiffres 1 devant la représentation à k bits. Dans les deux cas, la représentation de n sur k+h est $b_{k-1}b_{k-1}\dots b_{k-1}b_{k-2}\dots b_1b_0$ obtenue en ajoutant h copies du bit de poids fort b_{k-1} devant la représentation sur k bits.



Extension signée de k bits à k+h bits d'un entier

3.1.4.5 Comparaison des représentations

Représentation	Avantages	Inconvénients
avec bit de signe	représentation naturelle intervalle symétrique changement de signe facile	2 représentations pour 0 comparaison difficile addition difficile
biaisée	comparaison facile véritable différence	représentation de 0 intervalle non symétrique addition difficile
en complément à 2	représentation de 0 bit de signe comparaison facile addition et soustraction semblables	intervalle non symétrique

3.2 Nombres en virgule fixe

3.3 Nombres en virgule flottante

La norme IEEE 754 définit des codages des nombres en virgule flottante sur un format de 32 bits appelé *simple précision* (déclaré par `float` en C), un format de 64 bits appelé *double précision* (déclaré `double` en C) et un format de 80 bits appelé *précision étendue* (déclaré `long double` en C). Elle définit aussi les opérations arithmétiques usuelles (+,-,×,/,√) et les arrondis à effectuer pour

ces opérations. Par contre, elle ne normalise pas les fonctions mathématiques comme exp, log, sin, cos, …. L'intérêt principal de cette norme est d'avoir des comportements identiques des programmes sur des machines différentes.

Le codage d'un nombre est inspiré de la notation scientifique comme $-1.5 \times 10^{+3}$. Chaque nombre est décomposé en trois parties : *signe*, *exposant* et *mantisse*. Le signe est codé par le bit de poids fort. Ensuite un certain nombre de bits sont consacrés à l'exposant et le reste à la mantisse. La table ci-dessous donne le nombre de bits de chacun des éléments dans les différents formats.

	Encodage	**Signe s**	**Exposant e**	**Mantisse m**	**Valeur d'un nombre**
Simple précision	32 bits	1 bit	8 bits $1 \le e \le 254$	23 bits	$(-1)^s \times 1.m \times 2^{e-127}$
Double précision	64 bits	1 bit	11 bits $1 \le e \le 2046$	52 bits	$(-1)^s \times 1.m \times 2^{e-1023}$
Précision étendue	80 bits	1 bit	15 bits $1 \le e \le 32766$	64 bits	$(-1)^s \times 1.m \times 2^{e-16383}$

Dans la table ci-dessus, la formule 1.m doit être interprétée de la façon suivante. Si les bits de la mantisse sont $b_1 b_2 \ldots b_n$, on a

$$1.m = 1 + b_1/2 + b_2/2^2 + b_3/2^3 + \ldots + b_n/2^n$$

Soit l'exemple en simple précision 101111110101100…0 (0xbf580000 en hexadécimal). Il se décompose en le signe s = 1, l'exposant $e = 01111110 = (126)_{10}$ et la mantisse m = 1010100…0. La valeur de 1.m = 1+1/2+1/8+1/16 = 1,6875. La valeur du nombre est donc $-1{,}6875 \times 2^{-1} = -0{,}84375$.

Les fonctions C suivantes permettent de considérer un flottant comme un entier et inversement afin par exemple d'afficher le codage hexadécimal d'un flottant. Il n'est pas possible d'utiliser les *cast* directement sur les valeurs car cela effectuerait une conversion. On utilise au contraire un *cast* sur les pointeurs.

```
// On suppose sizeof(int) == sizeof(float)
int float2int(float f) {
  int* p = (int*) &f;
  return *p;
}

float int2float(int n) {
  float* p = (float*) &n;
  return *p;
}

int main(void) {
  // Float --> hexa
  printf("%08x\n", float2int(-0.84375));

  // Hexa --> float
  printf("%g\n", int2float(0xbf580000));
}
```

Si le nombre de bits consacrés à l'exposant est k, la valeur de l'exposant e vérifie $0 < e < 2^k-1$. Les valeurs 0 et 2^k-1 sont réservées pour des valeurs spéciales.

La norme prévoit quatre types d'arrondi : vers 0, vers +∞, vers -∞ ou au plus près. En théorie, l'utilisateur a le choix de l'arrondi mais aucun des langages de programmation actuels ne permet de le choisir effectivement.

Les valeurs spéciales qui peuvent être représentées sont données par la table ci-dessous dans le cas de la simple précision.

Signe	Exposant	Mantisse	Valeur	Commentaire
0	0	0	0	unique représentation de 0
s	0	$m \neq 0$	$(-1)^s \times 0.m \times 2^{-126}$	nombres dénormalisés
0	255	0	$+\infty$	résultat de 1/0
1	255	0	$-\infty$	résultat de -1/0
0	255	$m \neq 0$	NaN	Not a Number : résultat de 0/0 ou $\sqrt{-1}$

Dans le cas de la simple précision, les valeurs maximales sont $\pm (2-2^{-23}) \times 2^{127} \cong 2^{128}$. La plus petite valeur positive représentable est $2^{-23} \times 2^{-126} = 2^{-149}$ (nombre dénormalisé).

3.4 Caractères

3.4.1 Codage ASCII

Le codage ASCII affecte un code sur 7 bits aux principaux caractères mais pas aux caractères accentués. Il existe des extensions plus ou moins standards pour coder les caractères accentués sur un octet comme le codage ISO Latin 1.

3.4.2 Codage UNICODE

Le codage Unicode permet de manipuler tous les caractères possibles. Il affecte en effet un code 32 bits à tous les caractères de toutes les langues. Il faut cependant faire attention au fait que les codes ne sont pas manipulés explicitement pour éviter que chaque caractère occupe 4 octets. On utilise un format de codage dont le plus répandu est le format UTF-8.

4 Transistors et portes logiques

On explique dans cette partie le fonctionnement schématique d'un transistor ainsi que la façon de réaliser des portes logiques avec des transistors.

4.1 Semi-conducteurs

Les trois semi-conducteurs utilisés dans la fabrication des composants électroniques sont les suivants :

- le silicium **Si** utilisé majoritairement,
- le germanium **Ge**,
- l'arséniure de gallium **AsGa**.

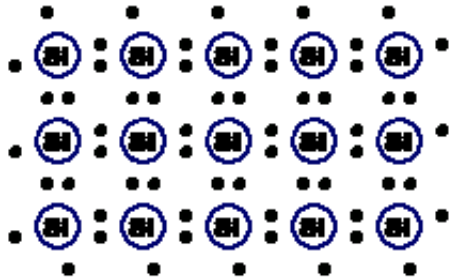
En plaçant ces différents éléments dans la table des éléments chimiques (cf. figure ci-dessous), on constate que ces éléments sont proches. Ces semi-conducteurs sont formés d'éléments ayant 4 électrons de valence sur la dernière couche comme le silicium ou le germanium ou d'un mélange d'éléments ayant 3 et 5 électrons de valence comme le gallium et l'arsenic.

B 3 5 Bore	**C** 4 6 Carbone	**N** 5 7 Azote
Al 3 13 Aluminium	**Si** 4 14 Silicium	**P** 5 15 Phosphore
Ga 3 31 Gallium	**Ge** 4 32 Germanium	**As** 5 33 Arsenic

Une partie de la table des éléments

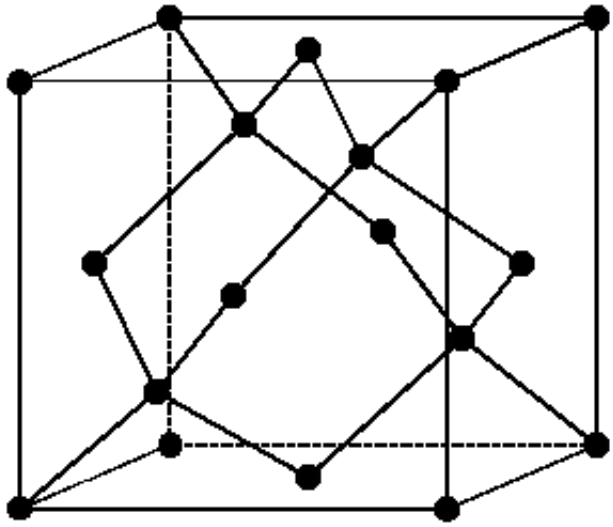
4.1.1 Semi-conducteur intrinsèque

Dans un cristal de silicium pur, les atomes forment 4 liaisons de covalence avec 4 voisins. Il remplissent ainsi la dernière couche en partageant les électrons.



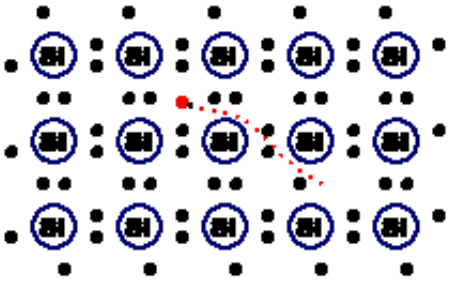
Cristal de silicium

Dans la figure ci-dessus, le cristal est représenté en plan mais il faut bien sûr imaginer ce cristal dans l'espace. La maille élémentaire du réseau est représentée à la figure ci-dessous.



Réseau dans l'espace

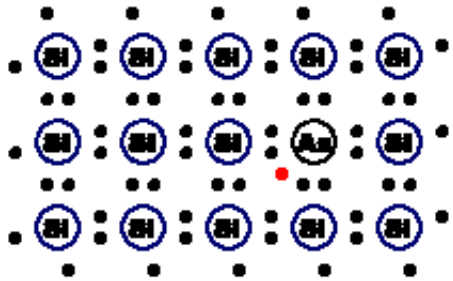
Au zéro absolu, c'est-à-dire à la température 0°K, la structure du cristal est stable et le silicium n'est pas conducteur du courant électrique. Lorsque la température augmente, les électrons possèdent une énergie supplémentaire qui provoque la rupture de certaines liaisons de covalence. Certains électrons deviennent libres (cf. figure ci-dessous) et le silicium est alors peu conducteur, d'où le nom de *semi-conducteur*.



Électron libre

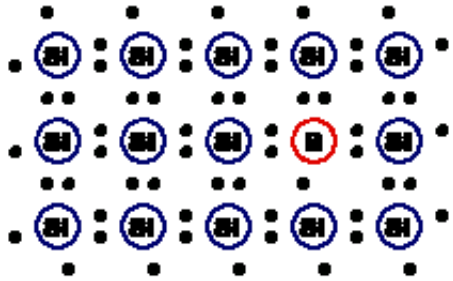
4.1.2 Semi-conducteurs dopés

Le silicium est *dopé* en introduisant des impuretés dans le cristal. On distingue deux types de dopage suivant la nature des éléments ajoutés au silicium. On parle de semi-conducteur *de type n* (pour négatif) lorsque le dopage est réalisé avec des éléments ayant 5 électrons de covalence comme le phosphore, l'arsenic et l'antimoine. L'atome avec 5 électrons de covalence forme 4 liaisons de covalence et garde un électron qui est alors relativement libre (cf. figure ci-dessous).



Silicium dopé n avec un électron libre

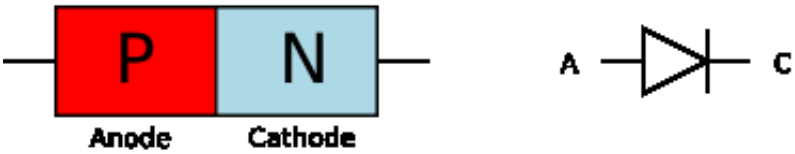
On parle de semi-conducteur *de type p* (pour positif) lorsque le dopage est réalisé avec des éléments ayant 3 électrons de covalence comme le bore, l'aluminium et le gallium. L'atome avec 3 électrons de covalence ne peut former que 3 liaisons de covalence. Il y en quelque sorte un trou d'électron (cf. figure ci-dessous).



Silicium dopé p avec un trou

4.2 Diode

Une jonction entre un semi-conducteur de type n et un semi-conducteur de type p est appelée une *diode*. La partie de type n est appelée *cathode* et la partie de type p est appelée *anode*. Le courant électrique ne peut passer à travers une diode que dans un seul sens comme l'évoque son symbole en forme d'entonnoir.



Diode

Le principe de fonctionnement de la diode est le suivant. Les électrons libres du semi-conducteur de type n ont tendance à aller boucher les trous du semi-conducteur de type p. Il en découle une diffusion des électrons de la région dopée n vers la région dopée p. Chaque électron qui se déplace laisse un ion positif dans la région n. Il s'ensuit donc un champ électrique de rappel vers la région n qui conduit à un équilibre. Dans cet équilibre, il y a une zone, appelée *zone de charge d'espace* qui ressemble à du silicium non dopé et où il y a en outre un champ électrique.

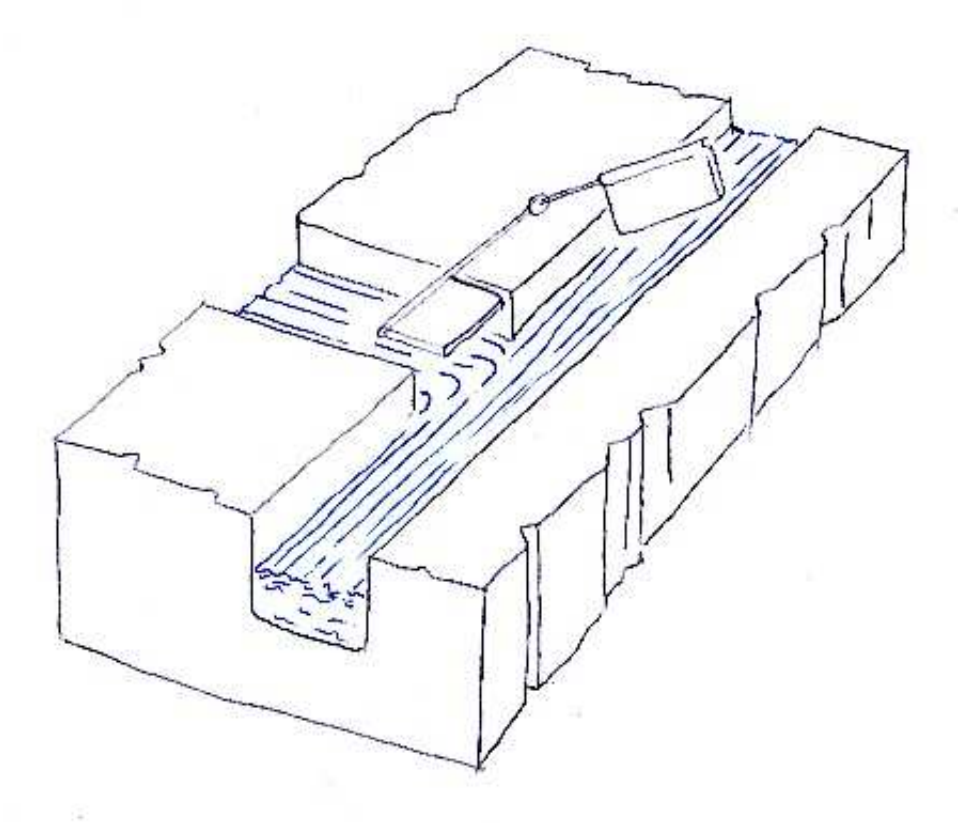
Si on applique une tension positive à la cathode et négative à l'anode, les électrons sont attirés vers le bord de la cathode et les trous vers le bord de l'anode. La zone de charge d'espace s'étend et la diode n'est pas conductrice. Si on contraire, on applique une tension positive à l'anode et négative à la cathode qui est supérieure au champ à l'équilibre, les électrons peuvent circuler de la cathode vers l'anode et la diode est conductrice.

4.3 Transistors

Le transistor est la *brique* avec laquelle sont construits les circuits électroniques tels les micro-processeurs. La technologie actuellement utilisée pour fabriquer les micro-processeurs est la technologie MOS (Metal-Oxide-Semiconductor). Il existe deux types de transistors MOS : les transistors de type *n* et les transistors de type *p*. Les micro-processeurs actuels utilisent des transistors des deux types. On parle alors de technologie CMOS (Complementary Metal-Oxide-Semiconductor).

4.3.1 Fonctionnement

Un transistor possède trois broches appelées drain, grille et source. Pour d'autres types de transistors, elle sont aussi appelées collecteur, base et émetteur. Dans le cas des transistors MOS, le drain et la source jouent des rôles (presque) symétriques et sont (pratiquement) interchangeables. Un transistor se comporte comme un interrupteur électrique entre la source et le drain qui serait commandé par la grille. Le dessin ci-dessous illustre de manière imagée la façon dont fonctionne un transistor.

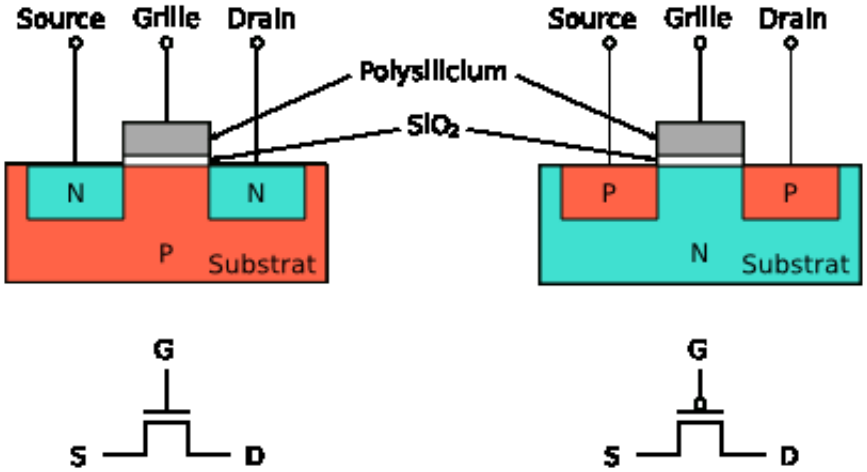


Fonctionnement imagé d'un transistor

Le comportement d'un transistor n-MOS est le suivant. Si la grille est mise à une tension de 2.9V, la source et le drain sont connectés. Si au contraire, la grille est mise à une tension de 0V, le circuit entre la source et le drain est ouvert. Le fonctionnement d'un transistor p-MOS est l'inverse. Le drain et la source sont connectés lorsque la tension appliquée à la grille est 0V.

4.3.2 Composition

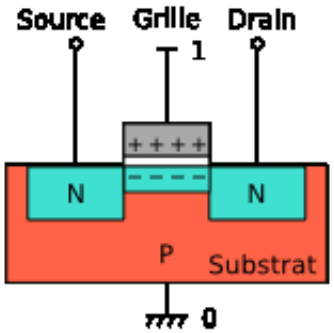
Un transistor est formé de deux jonctions np obtenues en intercalant une couche p (respectivement n) entre deux couches n (respectivement p). Le transistor de type n correspond aux trois couches *n-p-n* et le transistor de type p aux couches *p-n-p*. Il y a en outre une grille en métal au début de la technique MOS mais maintenant faite en *polysilicium*. Cette grille est séparée de la couche intermédiaire par une fine couche d'oxyde de silicium SiO_2 isolant. Le potentiel appliqué à la grille permet de modifier l'état de la couche intermédiaire.



Transistors n-MOS et p-MOS

4.3.3 Principe de fonctionnement

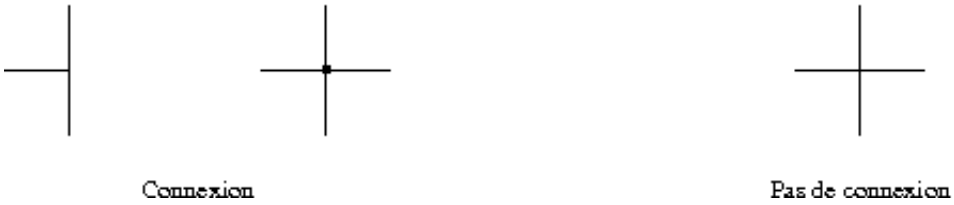
On considère un transistor de type n. Si aucune tension n'est appliquée à la grille, les deux jonctions *np* et *pn* se comporte comme des diodes en opposition et aucun courant ne peut passer entre la source et le drain. Si au contraire une tension positive est appliquée à la grille, les électrons chargés négativement s'accumulent dans le substrat dans la zone près de la grille. La région du substrat près de la grille va alors se comporter comme un semi-conducteur dopé n (cf. figure ci-dessous). Le courant peut alors passer entre la source et le drain.



Formation d'un tunnel n dans le substrat

4.4 Conventions dans les schémas

Dans tous les schémas du cours, on adopte la convention qui est illustrée à la figure suivante. Si un fil s'arrête a une intersection avec un autre fil, les deux fils sont connectés. Si par contre, deux fils se croisent, il n'y a pas connexion des deux fils sauf si l'intersection est matérialisée par un gros point.



Convention des connexions aux intersections

4.5 Portes *not*, *nand* et *nor*

La porte la plus simple est la porte *not* de la négation. Elle prend en entrée une valeur x qui vaut 0 ou 1 et elle sort la valeur 1-x. La table de vérité de cette porte est donnée ci-dessous.

Entrée	**Sortie**
I	**¬I**
0	1
1	0

La porte *not* peut être réalisée en logique CMOS par un circuit constitué de deux transistors, un de type n et un de type p. Ce circuit est appelé *inverseur*. L'inverseur ainsi que son symbole sont représentés à la figure ci-dessous.

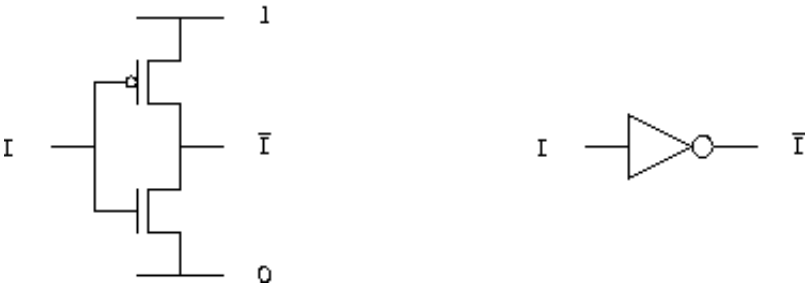


Schéma et symbole de l'inverseur

La porte *nand* prend en entrée deux valeurs 0 ou 1. La sortie vaut 0 si les deux entrées valent 1 et elle vaut 1 si au moins une des deux entrées vaut 0. La table de vérité est donnée ci-dessous.

Entrées		Sortie
A	**B**	**¬(A ∧ B)**
0	0	1
0	1	1
1	0	1
1	1	0

Un circuit pour réaliser la porte *nand* en logique CMOS est donné ci-dessous. Il est constitué de quatre transistors dont deux n-MOS et deux p-MOS.

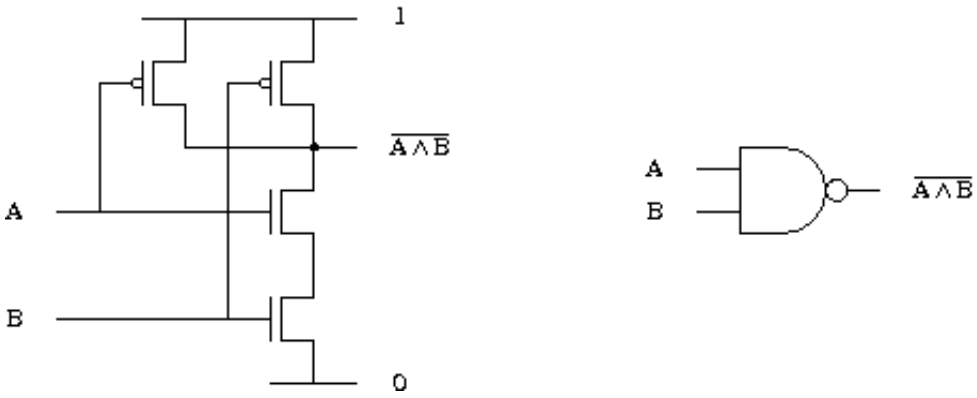


Schéma et symbole de la porte *nand*

La porte *nor* prend en entrée deux valeurs 0 ou 1. La sortie vaut 0 si au moins une des entrées vaut 1 et elle vaut 1 si les deux entrées valent 0. La table de vérité est donnée ci-dessous.

Entrées		Sortie
A	**B**	**¬(A ∨ B)**
0	0	1
0	1	0
1	0	0
1	1	0

Un circuit pour réaliser la porte *nor* en logique CMOS est donné ci-dessous. Il est constitué de quatre transistors dont deux n-MOS et deux p-MOS. C'est le circuit dual du circuit de la porte *nand*. Les deux transistors p-MOS qui était en parallèle dans le circuit de la porte *nand* sont en série dans le circuit de la porte *nor*. Au contraire, les deux transistors n-MOS qui était en série dans le circuit de la porte *nand* sont en parallèle dans le circuit de la porte *nor*.

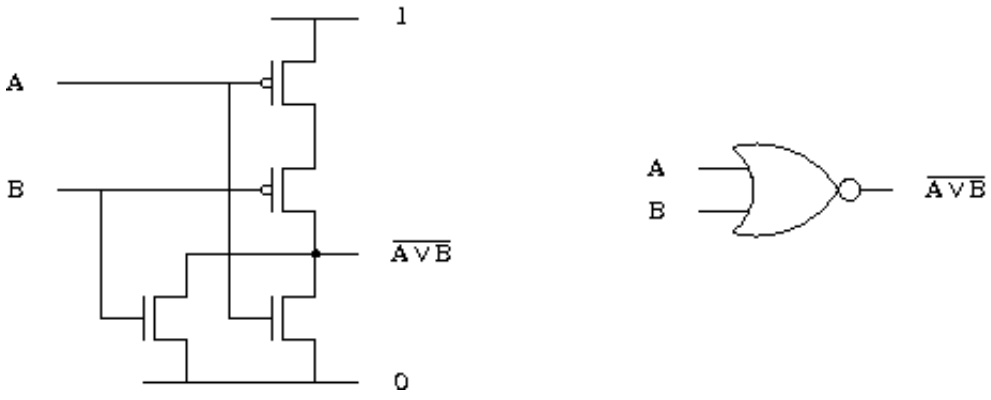


Schéma et symbole de la porte *nor*

4.6 Portes *or* et *and*

Si dans le schéma de la porte *nor*, chaque transistor n-MOS est remplacé par un transistor p-MOS et inversement, on obtient un schéma qui donne théoriquement une porte *and*. Pourtant, le circuit de la porte *and* n'est pas réalisé de cette manière. Cela provient du fait que la source et le drain des transistors ne jouent pas des rôles complètement symétriques. Pour des raisons de consommation, les connexions avec le 0 sont toujours commandées par des transistors de type n et les connexions avec le 1 par des transistors de type p.

Les circuits des portes *and* et or sont respectivement obtenus en combinant un circuit de la porte *nand* et *nor* avec un inverseur.

La porte *and* prend en entrée deux valeurs 0 ou 1. La sortie vaut 1 si les deux entrées valent 1 et elle vaut 0 sinon. La table de vérité est donnée ci-dessous.

Entrées		**Sortie**
A	**B**	**A ∧ B**
0	0	0
0	1	0
1	0	0
1	1	1

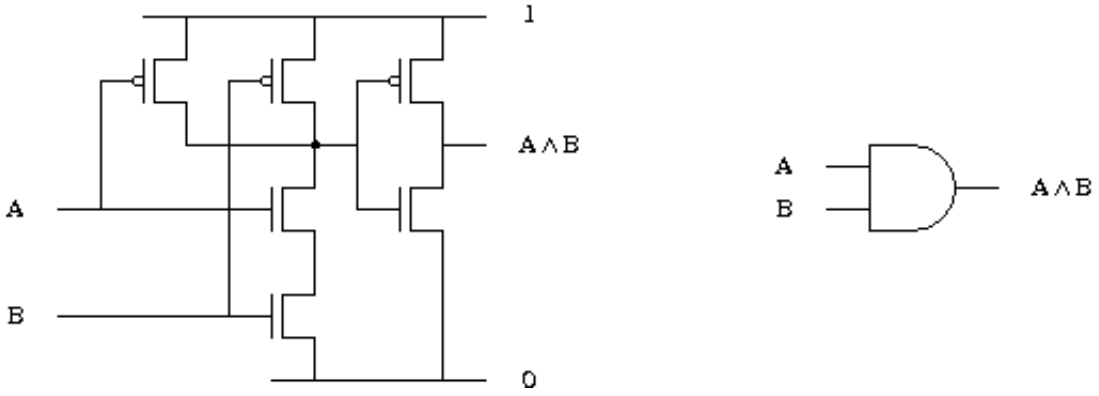
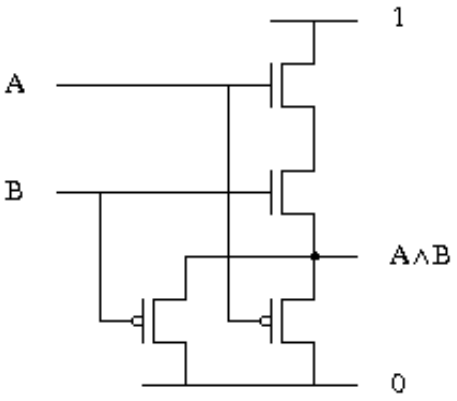


Schéma et symbole de la porte *and*

La porte *and* pourrait théoriquement être aussi réalisée par le schéma ci-dessous qui comporte moins de transistors. Ce schéma n'est cependant pas utilisé en pratique. Dans ce schéma, la sortie est reliée au 1 par des transistors n et au 0 par des transistors p. Ceci induit une consommation excessive par rapport au schéma précédent.



Mauvais schéma d'une porte *and*

La porte *or* prend en entrée deux valeurs 0 ou 1. La sortie vaut 0 si les deux entrées valent 0 et elle vaut 1 sinon. La table de vérité est donnée ci-dessous.

Entrées		**Sortie**
A	**B**	**A ∨ B**
0	0	0
0	1	1
1	0	1
1	1	1

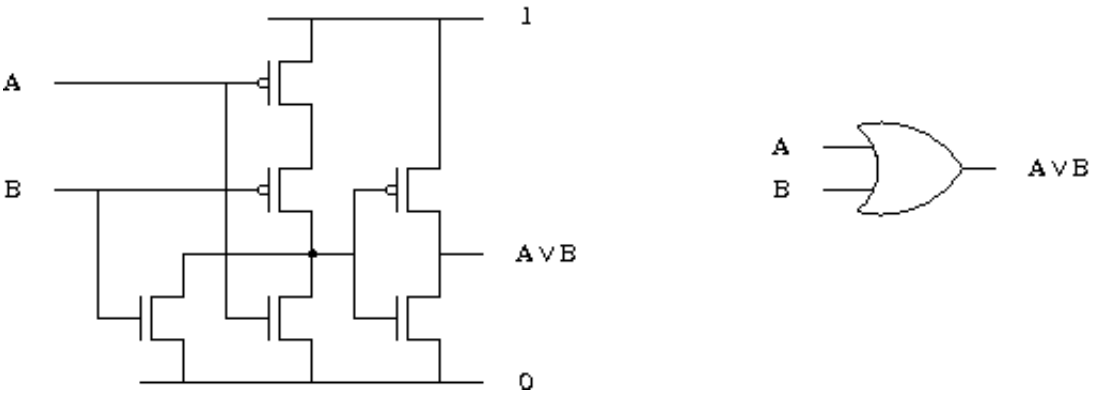


Schéma et symbole de la porte *or*

4.7 Portes *nand* et *nor* à trois entrées

On peut bien sûr réaliser une porte *nand* à trois entrées en combinant deux portes *nand* à deux entrées. Il est plus économique en nombre de transistors de réaliser directement cette porte. C'est la même chose pour la porte *nor* à trois entrées.

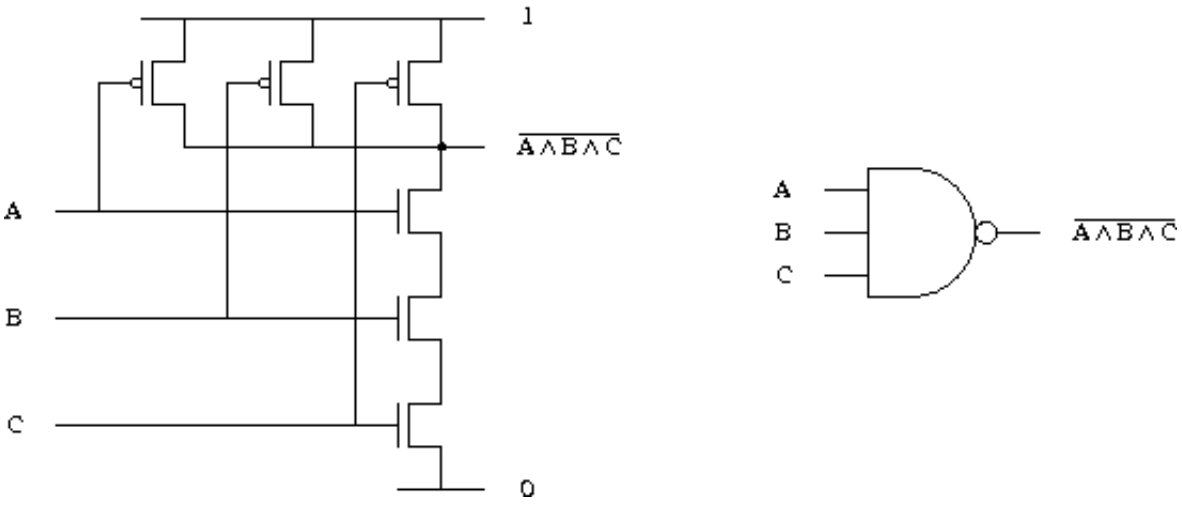


Schéma et symbole de la porte *nand* à 3 entrées

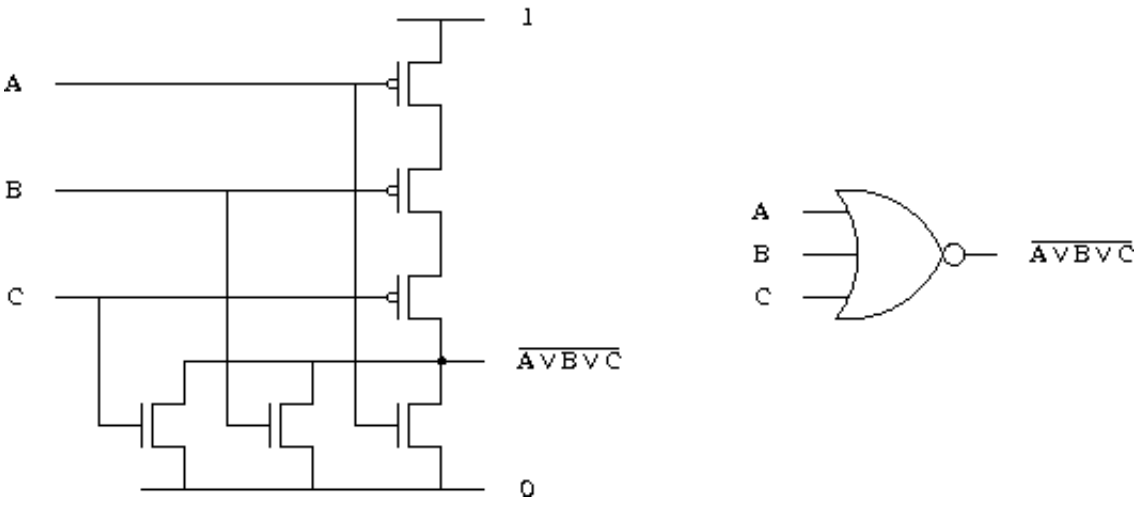


Schéma et symbole de la porte *nor* à 3 entrées

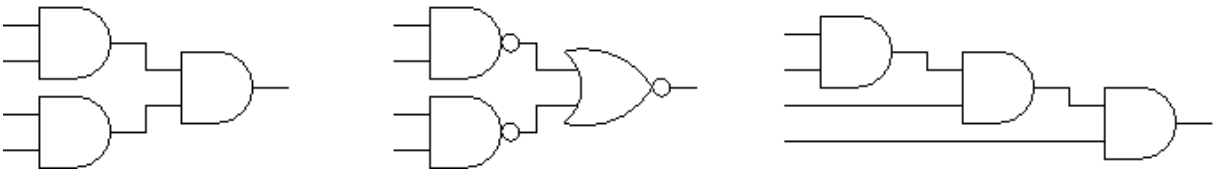
4.8 Portes *and* et *or* à entrées multiples

Les schémas des portes *nand* et *nor* à trois entrées peuvent être généralisés pour obtenir des schémas pour des portes *nand* et *nor* à un nombre quelconque d'entrées. Pour des raisons technologiques, ces portes ne sont pas réalisées de cette façon. Il est préférable de construire des portes à un grand nombre d'entrées en combinant plusieurs portes à 2 ou 3 entrées.

Des circuits pour les portes *and* ou *or* à entrées multiples peuvent être construits en utilisant un arbre constitué de portes *and* ou *or* respectivement. La construction d'un circuit pour une porte à k entrées peut être faite de manière récursive. Si k vaut 1 ou 2, le circuit est évident. Si k est supérieur à 2, on construit d'abord deux circuits ayant respectivement $\lfloor k/2 \rfloor$ et $\lceil k/2 \rceil$ entrées. Les sorties des ces deux circuits sont ensuite envoyées sur une dernière porte *and* ou *or*. On vérifie par récurrence que le nombre de portes d'un tel circuit est exactement k-1 et que sa profondeur (nombre maximal de portes mises en cascade) est $\lceil \log_2(k) \rceil$.

Pour k = 4, on obtient le premier circuit représenté à la figure ci-dessous. Pour économiser les transistors, chaque bloc de trois portes *and* ou *or* peut être remplacé par deux portes *nand* et une porte *nor* ou l'inverse. On obtient ainsi le second circuit représenté à la figure ci-dessous. Il existe d'autres

circuits comme le troisième représenté à la figure ci-dessous qui réalisent une porte *and* à 4 entrées. Ce circuit est cependant à éviter car il augmente le temps de latence en raison de sa plus grande profondeur.



Schémas de portes *and* à 4 entrées

En appliquant le procédé ci-dessus pour k = 16, puis en remplaçant chaque bloc de trois portes *and* par deux portes *nand* et une porte *nor*, on obtient le circuit ci-dessous qui a 10 portes *nand* et 5 portes *nor*. Ce circuit est particulièrement régulier car k est une puissance de 2.

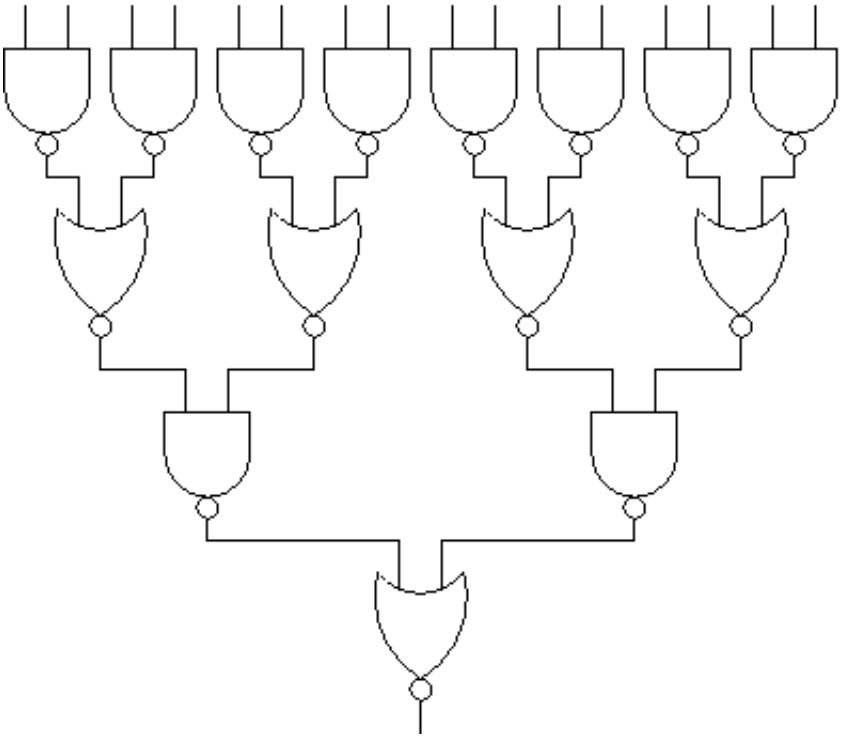
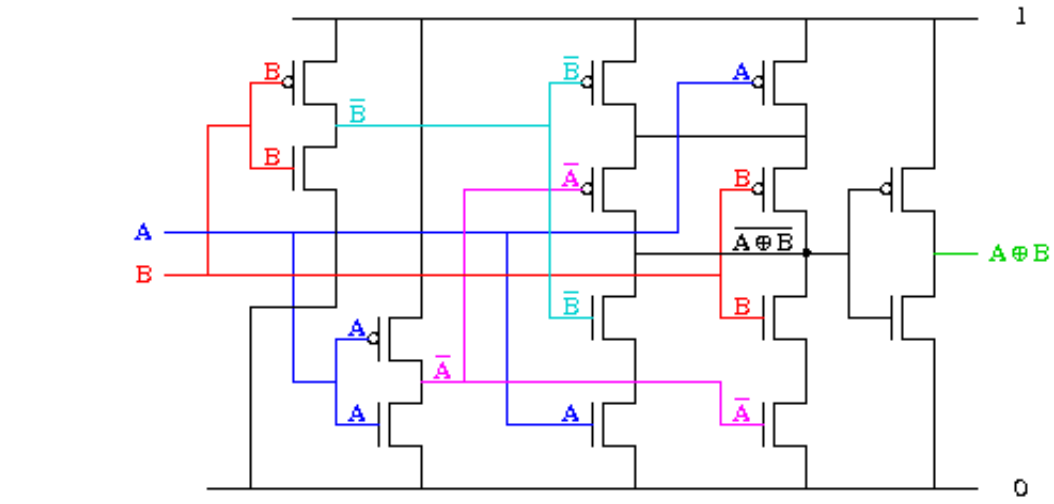


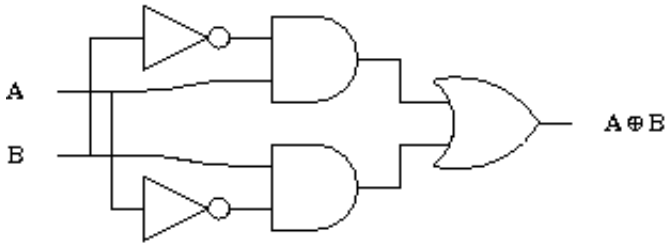
Schéma d'une porte *and* à 16 entrées

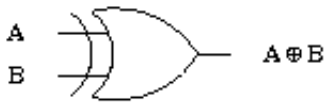
4.9 Porte *xor*

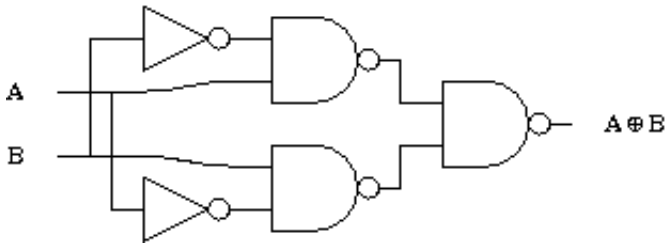
La porte *xor* permet de réaliser la fonction *ou exclusif* qui s'avère très utile pour construire les additionneurs. Comme toutes les autres fonctions booléennes, elle se réalise avec les portes *and* et *or* ou bien encore *nand* et *nor*. Il faut la considérer comme une abréviation pour un petit circuit très utile.

Entrées		Sortie
A	**B**	**A ⊕ B**
0	0	0
0	1	1
1	0	1
1	1	0









Schémas et symbole de la porte *xor*

5 Circuits élémentaires

Les décodeurs et multiplexeurs sont des circuits relativement élémentaires mais très souvent utilisés. Il s'agit de deux briques de base pour la construction de circuits plus élaborés. Ils sont en particulier présents dans chaque circuit mémoire. Un décodeur y décode l'adresse et active la ligne correspondante. Un multiplexeur permet d'y sélectionner la bonne sortie lors d'une lecture.

5.1 Décodeurs

Un décodeur k bits possède k entrées et 2^k sorties. La sortie dont le numéro est donné par les entrées est active (valeur 1) alors que toutes les autres sorties sont inactives (valeur 0).

Le décodeur 1 bit a donc une seule entrée A_0 et deux sorties S_0 et S_1. La table de vérité des sorties S_0 et S_1 est la suivante.

Entrée	**Sorties**	
$\mathbf{A_0}$	$\mathbf{S_0}$	$\mathbf{S_1}$
0	1	0
1	0	1

On remarque que la sortie S_0 est la négation de l'entrée A_0 alors que la sortie S_1 est égale à A_0. On a donc le circuit suivant avec un seul inverseur.

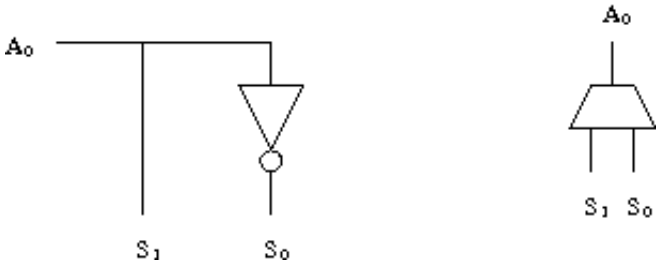


Schéma et symbole d'un décodeur 1 bit

Le décodeur 2 bits a deux entrées A_0 et A_1 ainsi que quatre sorties S_0, S_1, S_2 et S_3. La table de vérité des quatre sorties en fonction des deux entrées est la suivante.

Entrées		**Sorties**			
$\mathbf{A_1}$	$\mathbf{A_0}$	$\mathbf{S_0}$	$\mathbf{S_1}$	$\mathbf{S_2}$	$\mathbf{S_3}$
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

On remarque que l'on a les quatre formules suivantes pour les sorties.

$S_0 = \neg A_1 \wedge \neg A_0$
$S_1 = \neg A_1 \wedge A_0$
$S_2 = A_1 \wedge \neg A_0$
$S_3 = A_1 \wedge A_0$

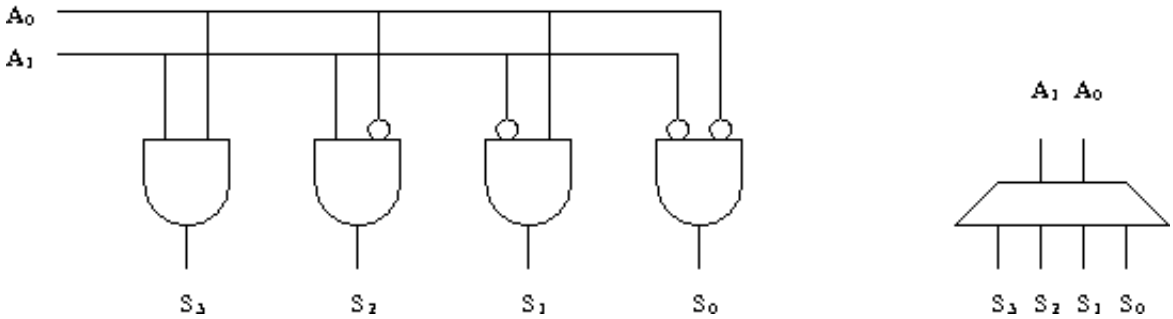


Schéma et symbole d'un décodeur 2 bits

Le décodeur 3 bits a 3 entrées et 8 sorties. Chaque sortie est obtenue comme le *et logique* des entrées ou de leurs négations. Les entrées devant être inversées correspondent aux zéros dans l'écriture en binaire du numéro de la sortie. Ainsi la sortie S_5 est égal à $A_2 \wedge \neg A_1 \wedge A_0$ car 5 s'écrit 101 en binaire.

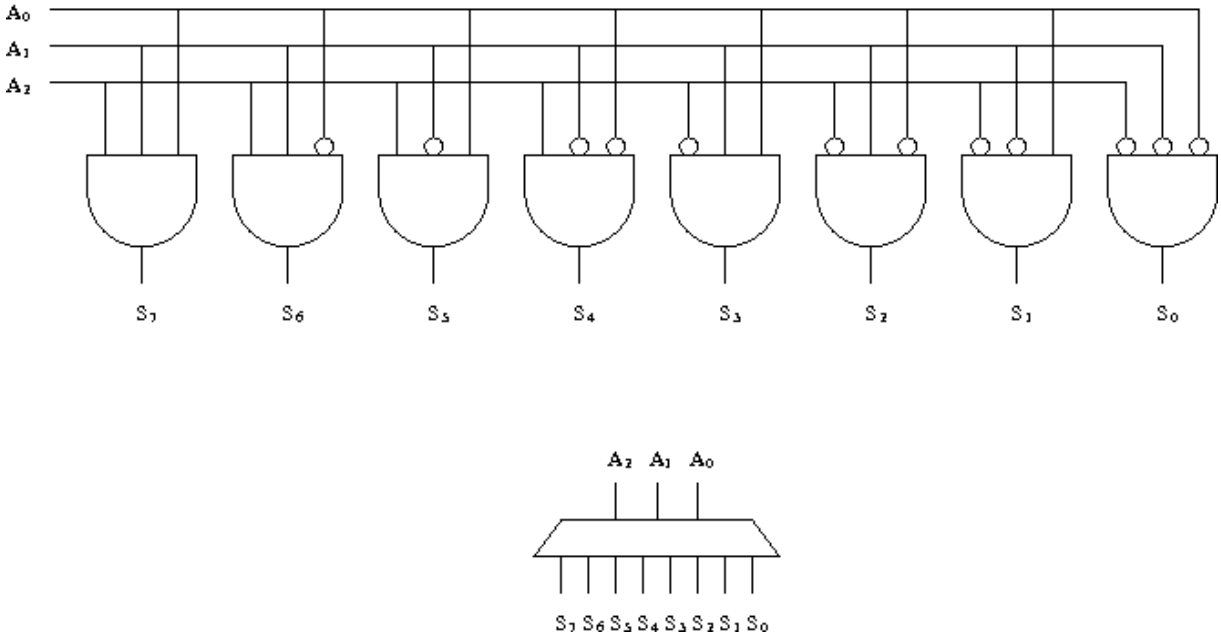


Schéma et symbole d'un décodeur 3 bits

Les circuits donnés pour k égal à 1, 2 et 3 se généralisent facilement à un entier quelconque. Le circuit est alors constitué d'inverseurs et de 2^k portes *and* ayant chacune k entrées. Le nombre d'inverseurs nécessaires est k puisqu'il en faut un pour chaque entrée A_i afin de disposer de sa négation $\neg A_i$. Sur les schémas pour k = 1, 2, 3, certains inverseurs peuvent être évités en réutilisant une valeur calculée par un autre. Par contre, les schémas deviennent beaucoup moins clair. Comme chaque porte *and* à k entrées peut être réalisée avec k-1 portes *and* à 2 entrées (cf. la section portes logiques), on obtient un total de $(k-1)2^k$ portes *and*. La profondeur de ce circuit est alors $\lceil \log_2(k) \rceil$.

5.1.1 Décodeur récursif

Il est possible de construire de manière récursive un décodeur ayant n entrée. Il faut alors introduire un circuit un peu plus général avec une nouvelle entrée appelée CS pour *Chip Select*. Un décodeur k bits avec une entrée CS possèdent donc k+1 entrées CS, A_0, …, A_{k-1} et 2^k sorties S_0, …, S_{2^k-1}. Ce circuit se comporte de la façon suivante. Si l'entrée CS vaut 0, toutes les sorties valent 0 et si CS vaut 1, il se comporte comme un simple décodeur. L'entrée dont le numéro s'écrit en binaire A_{k-1} … A_0 vaut 1 et toutes les autres sorties valent 0.

Pour k = 1, la table de vérité du décodeur 1 bit avec une entrée CS est donc la suivante.

Entrée		**Sorties**	
CS	**A_0**	**S_0**	**S_1**
0	0	0	0
0	1	0	0
1	0	1	0
1	1	0	1

On en déduit facilement le circuit ci-dessous.

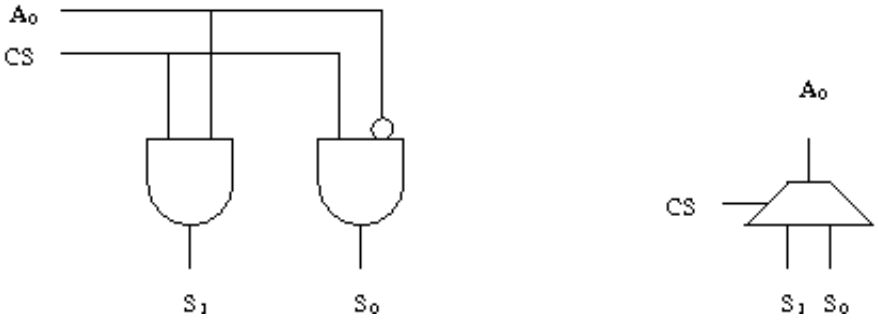
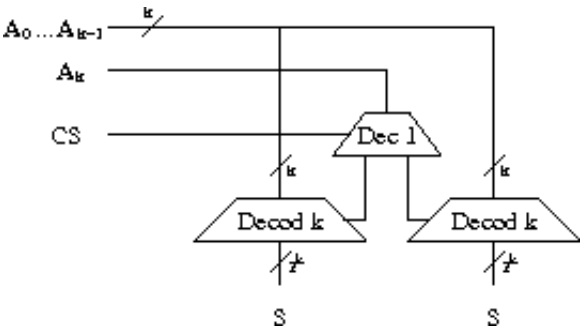


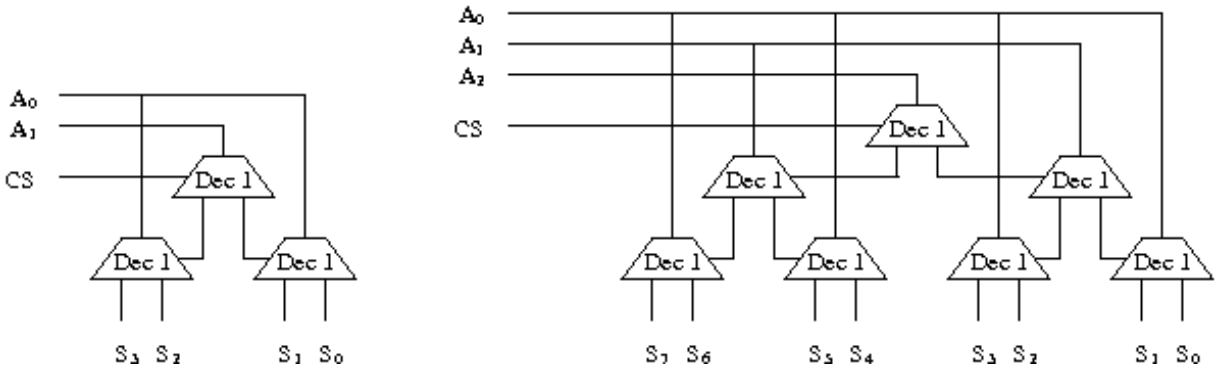
Schéma et symbole d'un décodeur 1 bit avec entrée CS

Pour k > 1, on peut construire un décodeur à k+1 entrées en utilisant deux décodeurs à k entrées et un décodeur à 1 entrée et en les combinant comme sur le schéma ci-dessous. Les deux décodeurs à k entrées reçoivent les entrées A_0, …, A_{k-1} et leurs deux entrées CS sont commandées par les deux sorties du décodeur 1 bit. Ce dernier reçoit en entrée A_k et CS.



Construction récursive d'un décodeur à k+1 entrées

Pour k égal à 2 et 3, la construction donne les deux décodeurs suivants.



Schémas des décodeurs récursifs 2 et 3 bits

On montre facilement que le nombre de décodeurs 1 bit utilisés dans le décodeur k bits est 2^k-1 et que le nombre de portes *and* utilisées est donc 2^{k+1}-2. La profondeur du circuit est k. Par rapport au décodeur construit de manière directe, le circuit construit récursivement utilise moins de portes logiques (2^{k+1} au lieu de $k2^k$) mais a une profondeur supérieure (k au lieu de $\lceil \log_2(k) \rceil$).

5.2 Multiplexeurs

Un multiplexeur est en quelque sorte l'inverse d'un décodeur. Un multiplexeurs k bits permet de sélectionner une entrée parmi 2^k disponibles. Un multiplexeur k bits a $k + 2^k$ entrées et une seule sortie. Les k premières entrées $A_0,\ldots,A_{k-1}$ sont appelées *bits d'adresses* car elles donnent le numéro de l'entrée à sélectionner parmi les entrées $B_0,\ldots,B_{2^k-1}$. La sortie S est alors égale à cette entrée sélectionnée.

Le multiplexeur 1 bit a donc 3 entrées A_0, B_0 et B_1 et une seule sortie S. La formule donnant la sortie S en fonction des entrées est la suivante.

$S = (A_0 \wedge B_1) \vee (\neg A_0 \wedge B_0)$

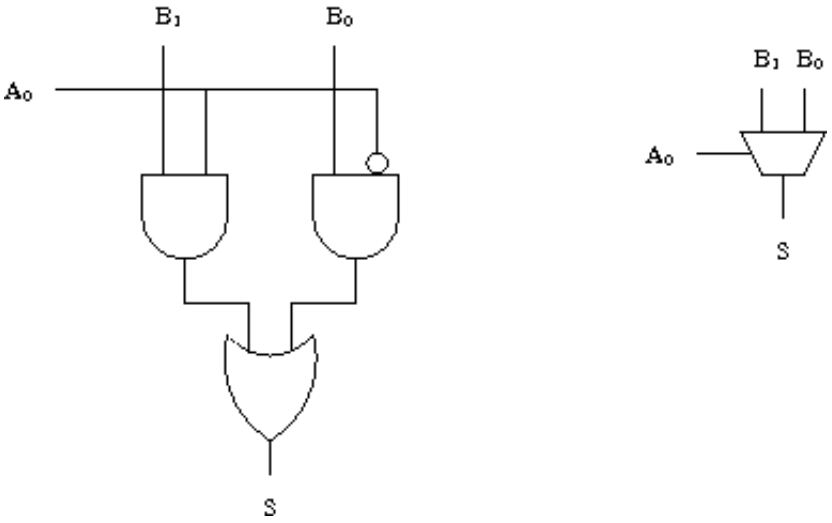


Schéma et symbole d'un multiplexeur 1 bit

Le multiplexeur 2 bit a donc 6 entrées A_0, A_1, B_0, B_1, B_2 et B_3 et une seule sortie S. La formule donnant la sortie S en fonction des entrées est la suivante.

$S = (A_1 \wedge A_0 \wedge B_3) \vee (A_1 \wedge \neg A_0 \wedge B_2) \vee (\neg A_1 \wedge A_0 \wedge B_1) \vee (\neg A_1 \wedge \neg A_0 \wedge B_0)$

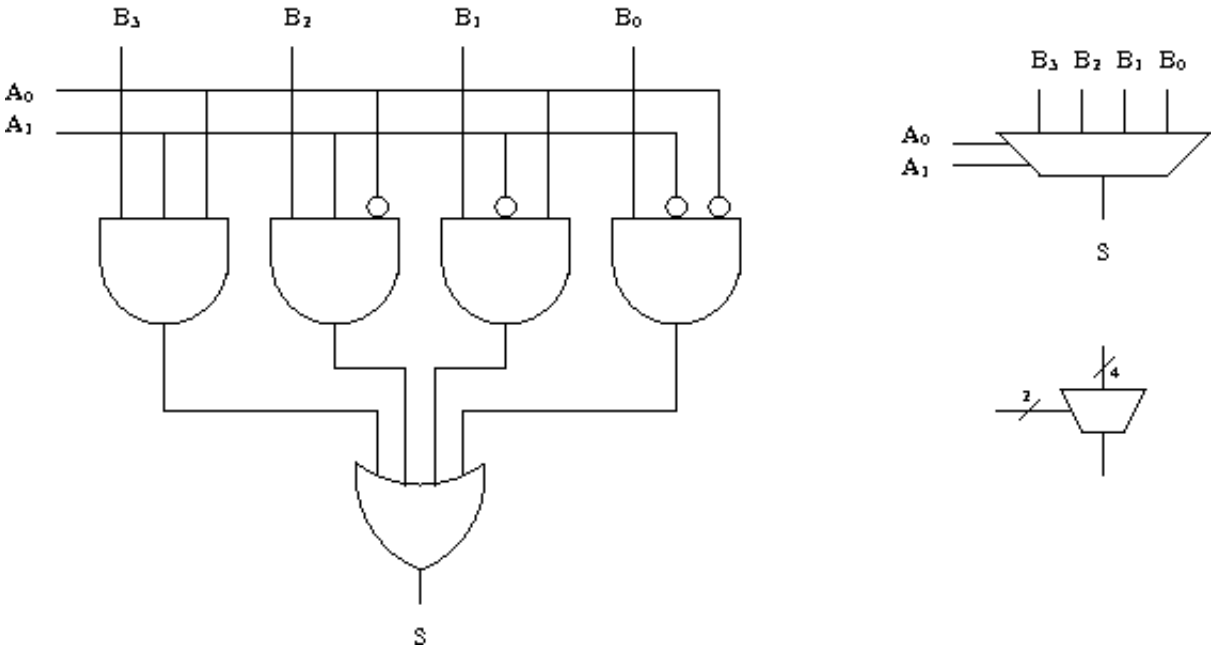
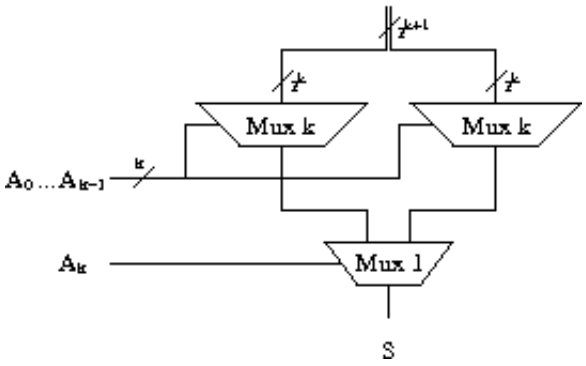


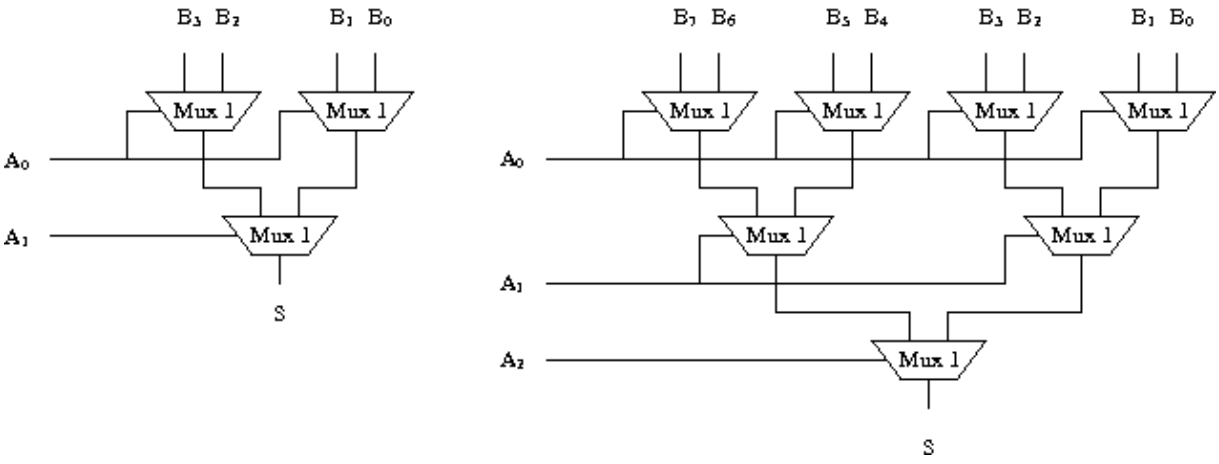
Schéma et symbole d'un multiplexeur 2 bits

Le multiplexeur k bits peut être construit récursivement en utilisant le schéma ci-dessous qui donne la construction d'un multiplexeur k+1 bits avec deux multiplexeurs k bits et un multiplexeur 1 bit.



Multiplexeur k+1 bits construit avec deux multiplexeurs k bits

Pour k égal à 2 et 3, la construction donne les deux multiplexeurs suivants.



Schémas des multiplexeurs récursifs 2 et 3 bits

Comme pour les décodeurs récursifs, on montre facilement que le nombre de multiplexeurs 1 bit utilisés dans le multiplexeur k bits est 2^k-1 et que le nombre de portes logiques *and* et *or* utilisées est donc $3(2^k-1)$. La profondeur du circuit est k. Par rapport au multiplexeur construit de manière directe, le circuit construit récursivement utilise moins de portes logiques ($3\ 2^k$ au lieu de $k2^k$) mais a une profondeur supérieure (k au lieu de $\lceil \log_2(k) \rceil$).

5.3 Construction de circuits

Dans cette partie, nous expliquons comment construire le circuit d'une fonction booléenne. Nous allons le faire sur l'exemple de la fonction *hidden bit*.

5.3.1 Définition

Nous commençons par donner la définition de la fonction *hidden bit*. Cette fonction prend en entrée k valeurs booléennes et retourne une valeur booléenne. Soient k entrées binaires $A_1,\ldots, A_k$ et soit s la somme de ces entrées, c'est-à-dire le nombre d'entrées égales à 1 : $s = \#\{ i \mid A_i = 1 \}$. Cette somme est comprise entre 0 et k. La sortie S est alors égale à 0 si s = 0 et elle est égale à A_s si $1 \leq s \leq k$.

5.3.2 Cas k = 1

Entrée	**Sorties**	
$\mathbf{A_1}$	**s**	**S**
0	0	0
1	1	1

On a alors l'égalité $S = A_1$. Le circuit est dans ce cas trivial puisqu'il se contente de mettre l'entrée A_1 en sortie.

5.3.3 Cas k = 2

Entrées		Sorties	
A_1	A_2	s	S
0	0	0	0
1	0	1	1
0	1	1	0
1	1	2	1

On a alors l'égalité $S = A_1$. Le circuit est encore une fois immédiat puisqu'il se contente de mettre l'entrée A_1 en sortie.

5.3.4 Cas k = 3

Le cas k = 3 est le premier cas intéressant. On commence par calculer la table qui donne pour chaque valeur des entrées les valeurs des deux sorties s et S.

Entrées			Sorties	
A_1	A_2	A_3	s	S
0	0	0	0	0
1	0	0	1	1
0	1	0	1	0
1	1	0	2	1
0	0	1	1	0
1	0	1	2	0
0	1	1	2	1
1	1	1	3	1

À Partir de cette table de vérité de la fonction, il est facile de trouver une expression de la sortie S en fonction des trois entrées A_1, A_2 et A_3.

$$S = A_1\ A_2\ A_3 + A_1\ A_2\ A_3 + A_1\ A_2\ A_3 + A_1\ A_2\ A_3$$

Les quatre monômes correspondent aux quatre lignes de la table où la sortie prend la valeur 1. Le premier monôme $A_1A_2A_3$ correspond par exemple à la seconde ligne de la table. La sortie S vaut 1 lorsque $A_1 = 1$, $A_2 = 0$ et $A_3 = 0$.

Il est possible de construire un circuit à partir de la formule ci-dessus. Les valeurs des monômes peuvent être calculées par quatre portes *et* à trois entrées et la valeur finale de S peut être calculée par trois portes *ou* à deux entrées. Afin de construire un circuit le plus petit possible, il est nécessaire de simplifier au maximum la formule.

La formule ci-dessus peut être simplifiée. La somme des deux premiers monômes peut être réduite à un seul monôme plus simple de la façon suivante.

$A_1\ A_2\ A_3 + A_1\ A_2\ A_3 = A_1\ A_3\ (A_2 + A_2) = A_1\ A_3$

Nous présentons maintenant une méthode graphique élémentaire permettant de trouver une expression simple pour une formule. Cette méthode dite des *tables de Karnaugh* (cf. aussi wikibook) ne donne pas toujours la meilleure expression mais elle fonctionne relativement bien tout en étant facile à mettre en œuvre.

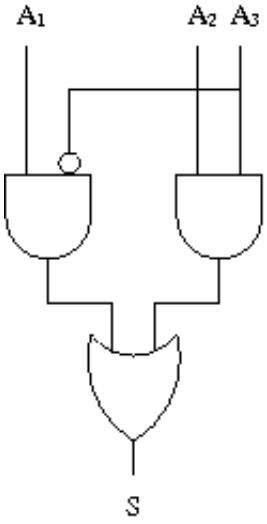
Nous expliquons la méthode sur notre exemple. La méthode consiste à disposer les valeurs de l'expression dans une table rectangulaire comme ci-dessous.

	$\mathbf{A_1}$		$\mathbf{A_1}$	
	$\mathbf{A_2}$	$\mathbf{A_2}$	$\mathbf{A_2}$	$\mathbf{A_2}$
$\mathbf{A_3}$	1	0	0	1
$\mathbf{A_3}$	1	1	0	0

Il faut essayer alors de regrouper les valeurs 1 en blocs rectangulaires. Ainsi, les deux 1 rouges correspondent à $A_1 = 1$ et $A_3 = 0$. Ils donnent donc le monôme $A_1 A_3$ que nous avions obtenu en regroupant les deux premiers monômes de notre formule initiale. Les blocs de 1 peuvent ne pas être contigus. Les deux 1 bleus correspondent à $A_2 = 1$ et $A_3 = 1$ et ils donnent le monôme $A_2 A_3$. Les blocs de 1 peuvent éventuellement se chevaucher. On a donc finalement trouvé la formule suivante pour S.

$S = A_2\ A_3 + A_1\ A_3$

Le circuit construit à partir de la formule précédente ne contient plus que deux portes *et* et une porte *ou* à deux entrées. On reconnaît le circuit d'un multiplexeur 1 bit où A_3 est le bit d'adresse (ou de commande) et A_1 et A_2 sont les entrées.



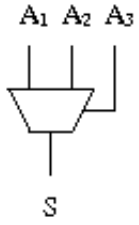


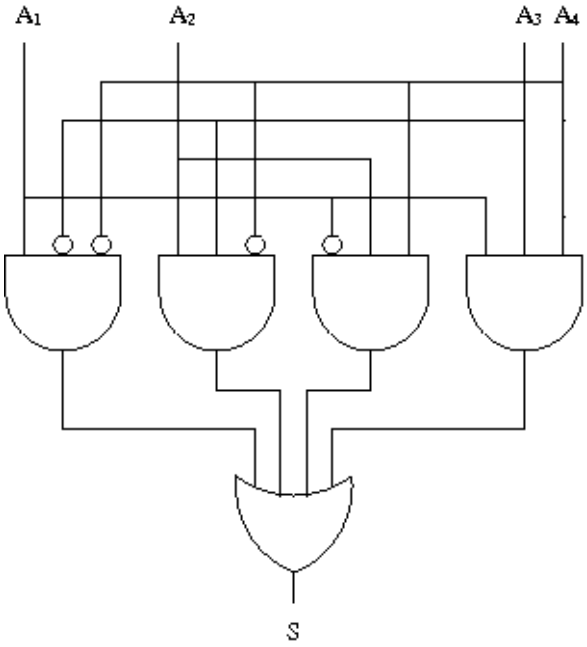
Schéma pour la fonction hidden bit à 3 entrées

5.3.5 Cas k = 4

Entrées				Sorties	
A_1	A_2	A_3	A_4	s	S
0	0	0	0	0	0
1	0	0	0	1	1
0	1	0	0	1	0
1	1	0	0	2	1
0	0	1	0	1	0
1	0	1	0	2	0
0	1	1	0	2	1
1	1	1	0	3	1
0	0	0	1	1	0
1	0	0	1	2	0
0	1	0	1	2	1
1	1	0	1	3	0
0	0	1	1	2	0
1	0	1	1	3	1
0	1	1	1	3	1
1	1	1	1	4	1

		A_1		A_1	
		A_2	A_2	A_2	A_2
A_3	A_4	1	1	0	1
	A_4	1	0	0	1
A_3	A_4	1	1	0	0
	A_4	0	0	0	1

$$S = A_1\ A_3\ A_4 + A_2\ A_3\ A_4 + A_1\ A_2\ A_4 + A_1\ A_3\ A_4$$



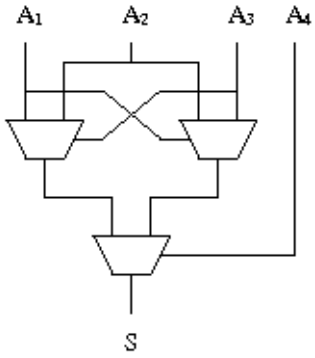


Schéma pour la fonction hidden bit à 4 entrées

5.3.6 Cas k = 5

$$S = A_1\ A_3\ A_4\ A_5 + A_2\ A_3\ A_4\ A_5 + A_1\ A_2\ A_4\ A_5 + A_1\ A_3\ A_4\ A_5 + A_1\ A_2\ A_4\ A_5 + A_1\ A_3\ A_4\ A_5 + A_1\ A_2\ A_3\ A_5 + A_1\ A_2\ A_4\ A_5$$

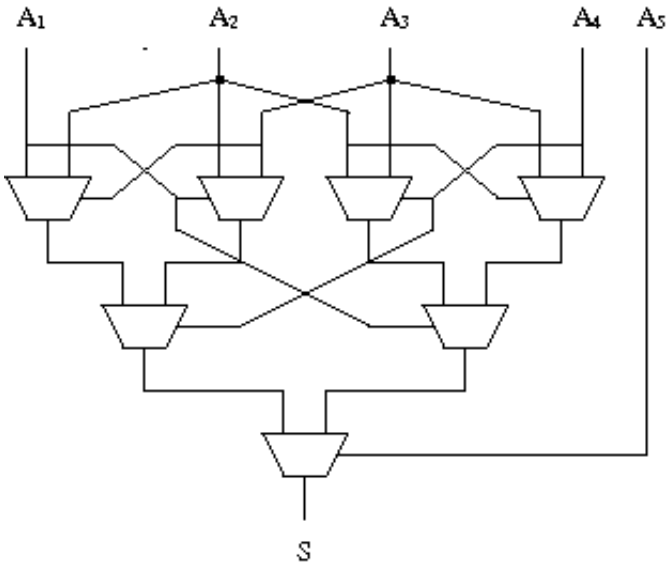


Schéma pour la fonction hidden bit à 5 entrées

6 Additionneurs

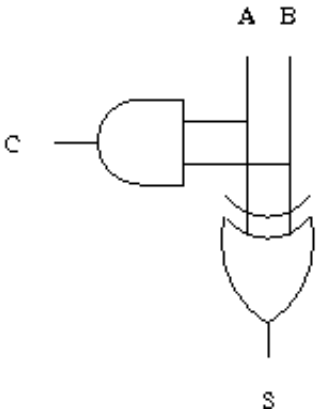
L'addition est une opération très courante dans un microprocesseur. Outre dans l'unité arithmétique, elle sert pour incrémenter le compteur de programme et pour les calculs d'adresses. Il est donc important qu'elle soit optimisée pour être rapide. Malgré la simplicité apparente du problème, il existe de multiples façons de construire des additionneurs efficaces en temps et en nombre de portes logiques utilisées.

6.1 Semi-additionneur

Ce premier circuit est la brique de base. Il prend en entrée deux bits A et B et calcule la somme S et la retenue C (pour Carry en anglais). Les bits C et S peuvent aussi être vus comme les bits de poids fort et de poids faible de l'écriture sur deux bits de la somme A + B.

Entrées		**Sorties**	
A	**B**	**C**	**S**
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

On remarque sur la table de vérité que S est le *ou exclusif* des deux entrées A et B, *i.e.* $S = A \oplus B$ et que C vaut 1 lorsque les deux entrées valent 1, c'est-à-dire $C = A \wedge B$.



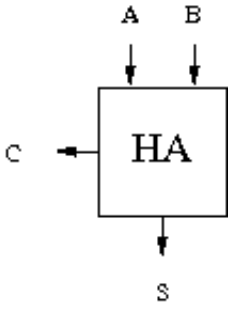


Schéma et symbole d'un semi-additionneur (HA)

6.2 Additionneur complet 1 bit

Pour construire un additionneur sur plusieurs bits, plusieurs additionneurs 1 bit sont mis en cascade. Chacun de ces addionneurs prend en entrée deux bits A et B ainsi que la retenue précédente C_0. Il calcule la somme S de ces trois valeurs binaires ainsi que la retenue C_1. Comme pour le semi-additionneur, ces bits C_1 et S peuvent aussi être vus comme les bits de poids fort et de poids faible de l'écriture sur deux bits de la somme $A + B + C_0$. Cette somme s'écrit justement sur deux bits car elle est comprise entre 0 et 3.

Entrées			Sorties	
A	**B**	**C_0**	**C_1**	**S**
0	0	0	0	0
0	1	0	0	1
1	0	0	0	1
1	1	0	1	0
0	0	1	0	1
0	1	1	1	0
1	0	1	1	0
1	1	1	1	1

On peut remarquer sur la table de vérité que S est le *ou exclusif* des trois entrées A, B et C_0, *i.e.* $S = A \oplus B \oplus C_0$ et que la retenue C_1 vaut 1 dès que deux des trois entrées valent 1, c'est-à-dire $C_1 = AB \vee AC_0 \vee BC_0$.

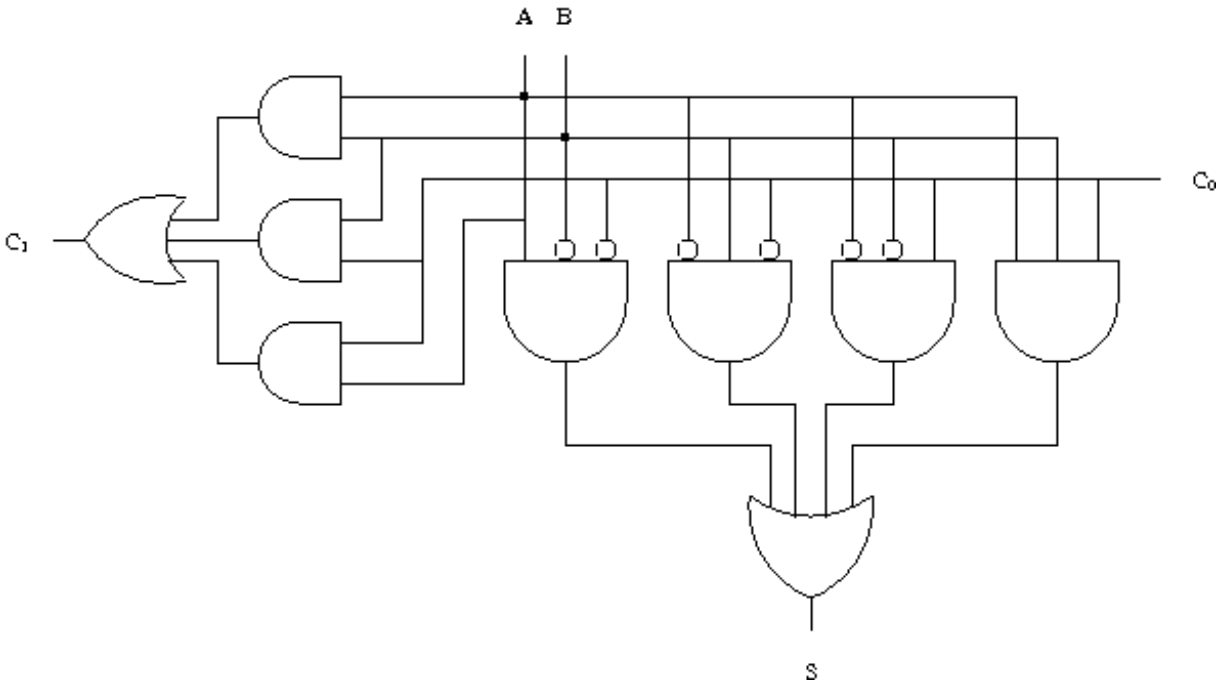


Schéma à partir des formules d'un additionneur complet

Ce circuit peut aussi être construit en assemblant deux semi-additionneurs en cascade. Le premier semi-additionneur calcule d'abord la somme de A et B puis le second calcule la somme du premier résultat et de C_0. La retenue C_1 vaut 1 s'il y a au moins une retenue à une des deux sommes effectuées.

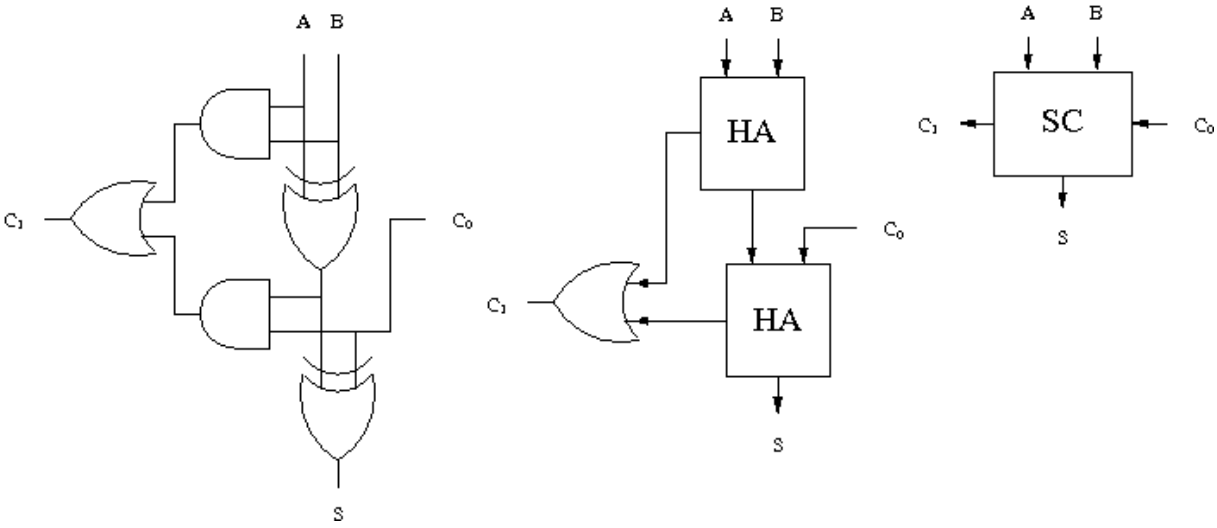


Schéma et symbole d'un additionneur complet (SC)

6.3 Additionneur par propagation de retenue

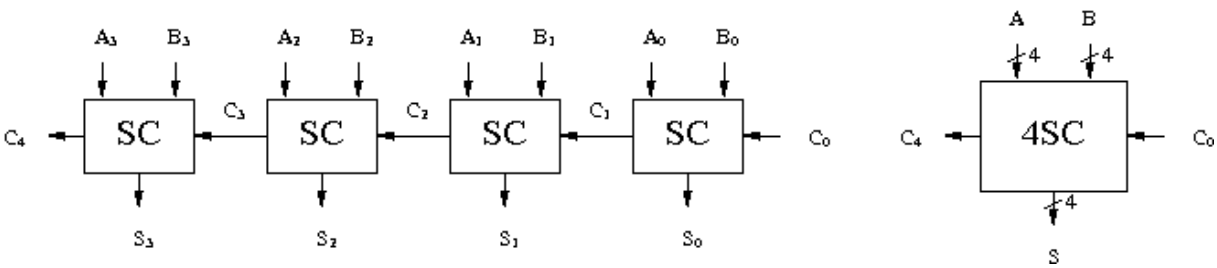
L'additionneur par propagation de retenue est le plus simple. Il est calqué sur l'algorithme manuel pour effectuer l'addition. Les paires de bits sont additionnés colonne par colonne et les retenues sont propagées vers la gauche.

En mettant en cascade k additionneurs complets 1 bits, on construit un additionneur k bits appelé *additionneur par propagation de retenue* car la retenue se propage d'additionneur en additionneur.

L'additionneur est formé de k additionneurs 1 bit numérotés de 0 à k-1 de la droite vers la gauche. L'additionneur i reçoit en entrées les bits A_i et B_i des entrées A et B ainsi que la retenue C_i engendrée par l'additionneur i-1. Il calcule le bit S_i de la somme et la retenue C_{i+1} qui est transmise à l'additionneur i+1.

L'additionneur 0 reçoit une retenue C_0 qui est normalement à 0 pour effectuer une addition. L'utilité de cette entrée est double. Elle permet d'une part de cascader les additionneurs pour former des additionneurs sur plus de bits. Elle permet d'autre part d'effectuer la soustraction.

Le nombre de portes logiques utilisées par un additionneur par propagation de retenue k bits est proportionnel au nombre k. Le temps de propagation de la retenue est également proportionnel au nombre k. Pour cette raison, cet additionneur est impraticable pour des nombres élevés de bits comme 16 ou 64. Il est seulement utilisé pour des petits nombres de bits, comme 4. Par contre, il peut servir de brique de base à des additionneurs plus sophistiqués comme l'additionneur hybride.



Additionneur 4 bits

6.4 Calcul des indicateurs

Dans tout micro-processeur, il existe des indicateurs ou flags qui sont des registres 1 bit pouvant prendre les valeurs 0 ou 1. Ils sont mis à jour dès qu'un chargement ou une opération logique ou arithmétique est effectuée. Ces indicateurs peuvent ensuite être testés par les branchement conditionnels.

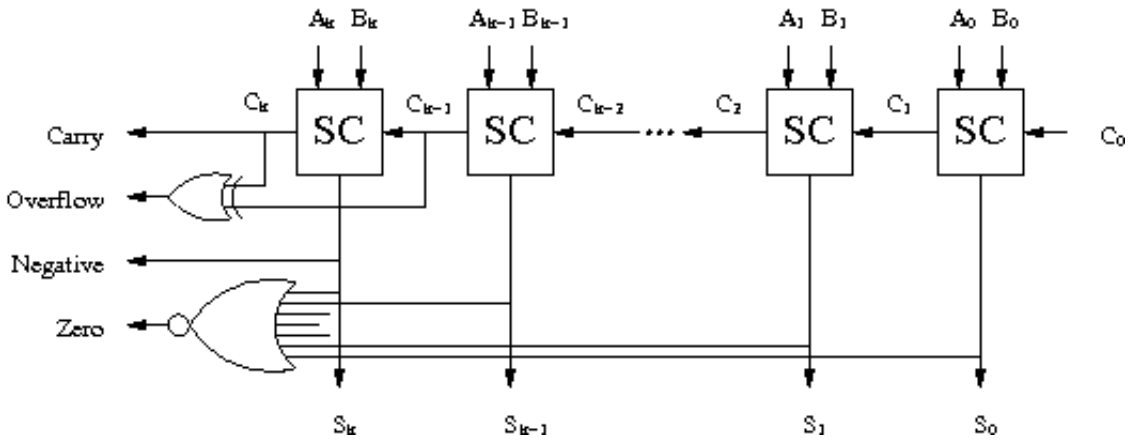
Les principaux indicateurs utilisés sont les indicateurs N, Z, C et O décrits ci-dessous. Chacun de ces indicateurs concerne le résultat de la dernière opération effectuée (chargement, logique ou arithmétique).

Indicateur N (pour Négatif)
: Il indique si le résultat est négatif. Comme les entiers sont représentés en compléments à 2, il est égal au bit de poids fort du résultat.

Indicateur Z (pour Zéro)
: Il indique si le résultat est égal 0. Il est donc égal au complémentaire du *ou logique* (c'est-à-dire un *nor*) de tous les bits du résultat.

Indicateur C (pour Carry)
: Il indique si l'opération a provoqué une retenue. Il est peut donc uniquement être positionné par les opérations arithmétiques. Il est mis à 1 lorsqu'il y a une retenue. Ceci correspond à un débordement pour une addition de nombres non signés.

Indicateur O (pour Overflow)
: Il indique un débordement lors d'une addition de nombres signés. Ce débordement intervient lorsque la somme de deux nombres positifs est supérieure à 2^{k-1} ou lorsque la somme de deux nombres négatifs est inférieure à $-2^{k-1}-1$. Cet indicateur est égal au *ou exclusif* $C_k \oplus C_{k-1}$ des deux dernières retenues.



Calcul des indicateurs

6.5 Additionneur par anticipation de retenue

La lenteur de l'additionneur par propagation de retenue impose d'utiliser d'autres techniques pour des additionneurs ayant un nombre important de bits. Comme cette lenteur est due au temps nécessaire à la propagation de la retenue, toutes les techniques ont pour but d'accélérer le calcul des retenues. La première technique appelée *anticipation de retenue* consiste à faire calculer les retenues par un circuit extérieur.

Afin de faciliter le calcul des retenues, on introduit deux quantités appelées G (pour Generate en anglais) et P (pour Propagate en anglais). Pour deux quantités binaires A et B, les quantités G et P sont définies de la façon suivante.

$G = AB$ et $P = A + B$

Soient $A = A_{n-1} \dots A_0$ et $B = B_{n-1} \dots B_0$ deux entrées de n bits. On note C_i la retenue de l'addition des i bits de poids faible de A et B. Pour accélérer le calcul des C_i, on introduit les deux quantités G_i et P_i associées aux entrées A_i et B_i par les formules suivantes.

$G_i = A_i B_i$ et $P_i = A_i + B_i$

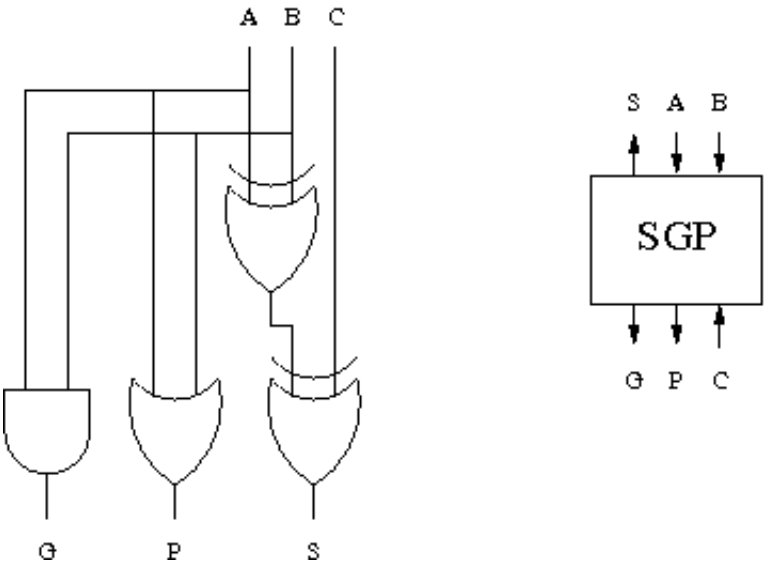
La valeur G_i est la retenue engendrée par l'addition des deux bits A_i et B_i et la valeur de P_i détermine si la retenue de C_i se propage. On a donc la formule suivante qui exprime simplement que la retenue C_{i+1} provient soit directement de l'addition des bits A_i et B_i soit de la propagation de la retenue C_i.

$C_{i+1} = G_i + P_i C_i$

En utilisant plusieurs fois cette formule, on peut obtenir les formules suivantes qui expriment C_{i+1} en fonction d'une retenue précédente et des valeurs G_j et P_j intermédiaires.

$C_{i+1} = G_i + P_i G_{i-1} + P_i P_{i-1} C_{i-1}$
$C_{i+1} = G_i + P_i G_{i-1} + P_i P_{i-1} G_{i-2} + P_i P_{i-1} P_{i-2} C_{i-2}$

Le circuit suivant permet de calculer la somme ainsi que les deux quantités G_i et P_i.



Cellule de calcul de S, G et P.

6.5.1 Calcul anticipé des retenues

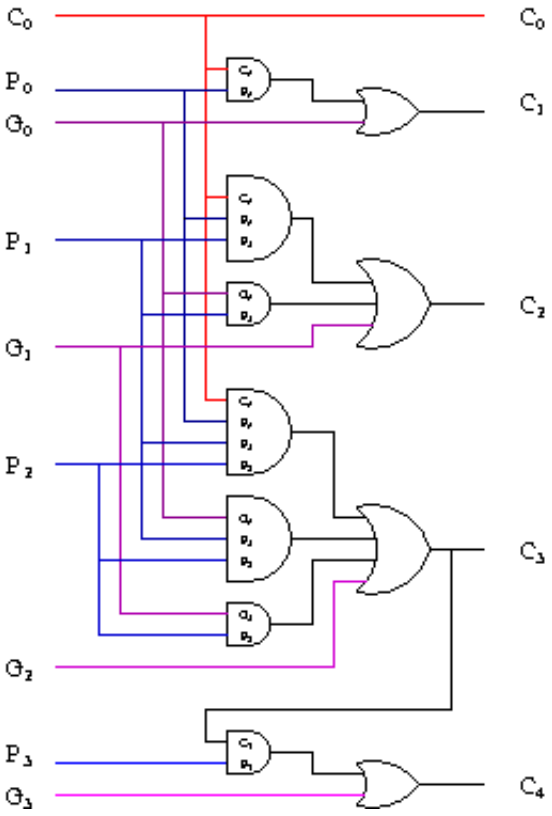
En utilisant les formules ci-dessus, on obtient les formules suivantes pour les retenues C_1, C_2, C_3, et C_4 à partir de C_0 et des signaux G_0, P_0, G_1, P_1, G_2, P_2 G_3 et P_3.

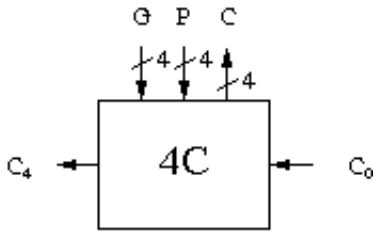
$C_1 = G_0 + P_0C_0$
$C_2 = G_1 + P_1G_0 + P_1P_0C_0$
$C_3 = G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0$
$C_4 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_0$

Comme la retenue C_4 n'est pas nécessaire pour calculer les sommes S_0, S_1, S_2 et S_3, celle-ci peut-être calculée en utilisant la formule suivante.

$C_4 = G_3 + P_3C_3$

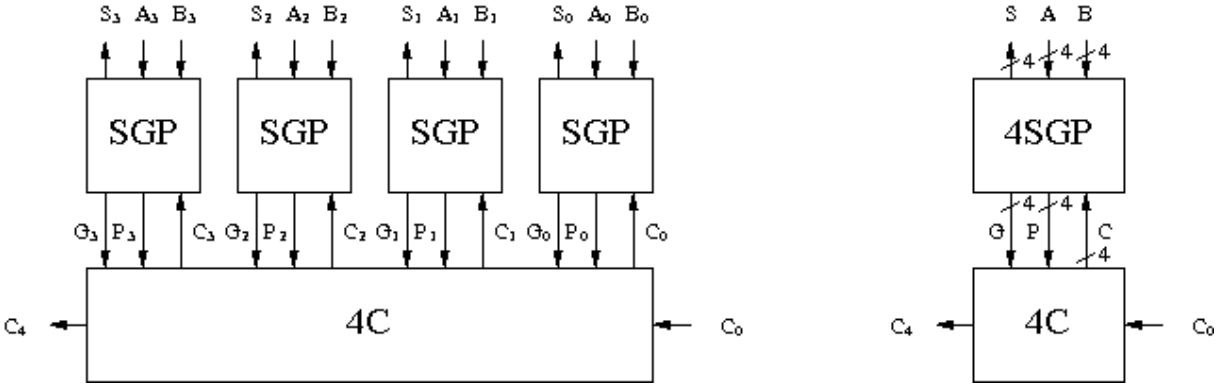
L'utilisation de cette dernière formule ne retarde pas les calculs des sommes et économise des portes.





Circuit de calcul des retenues

6.5.2 Additionneur 4 bits avec calcul anticipé des retenues

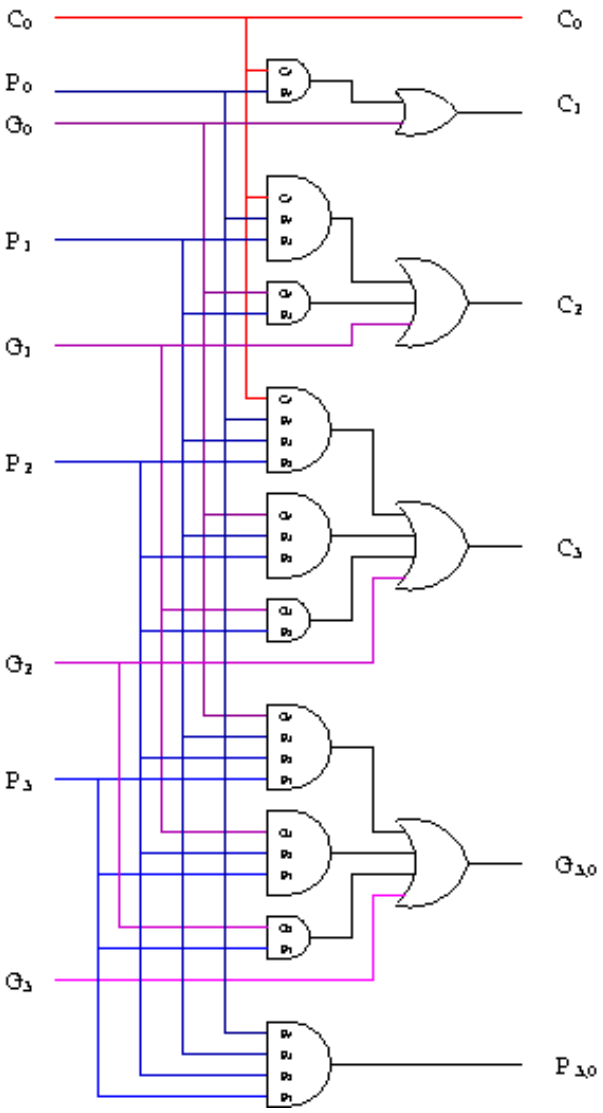


Additionneur 4 bits avec calcul anticipé des retenues

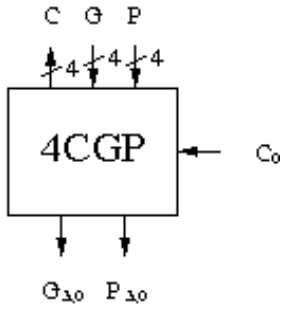
6.5.3 Additionneur 16 bits avec calcul anticipé des retenues

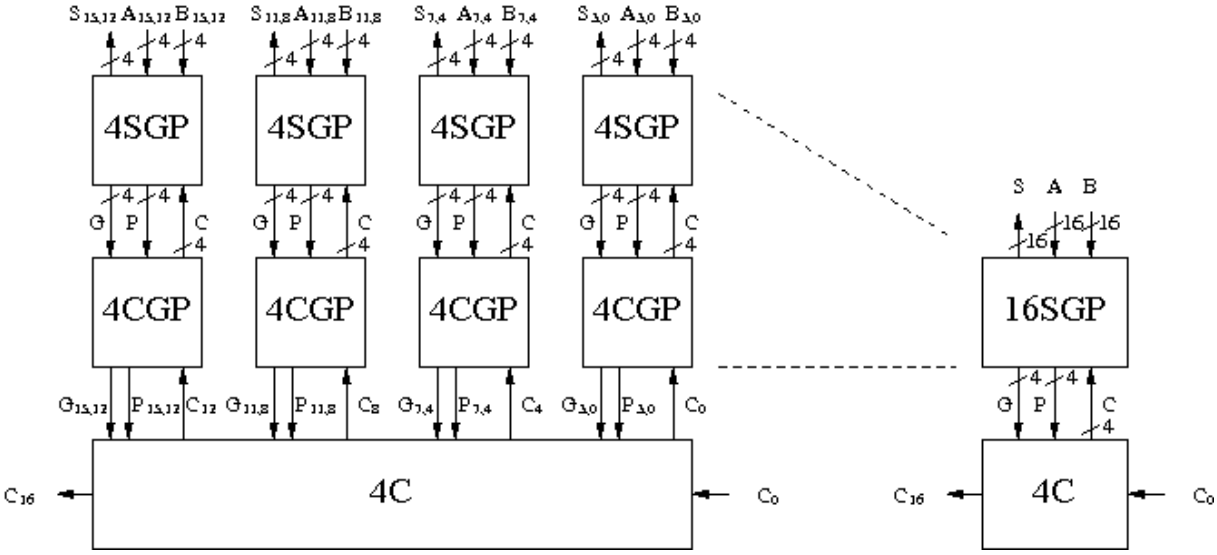
Plus généralement, on note pour deux indices i et j, $G_{j,i}$ et $P_{j,i}$ la retenue engendrée par l'addition des bits de i à j et la possibilité qu'une retenue se propage à travers les bits de i à j. On a donc la formule suivante qui exprime la retenue C_{j+1} en fonction de $G_{j,i}$, $P_{j,i}$ et C_i.

$$C_{j+1} = G_{j,i} + P_{j,i} C_i$$



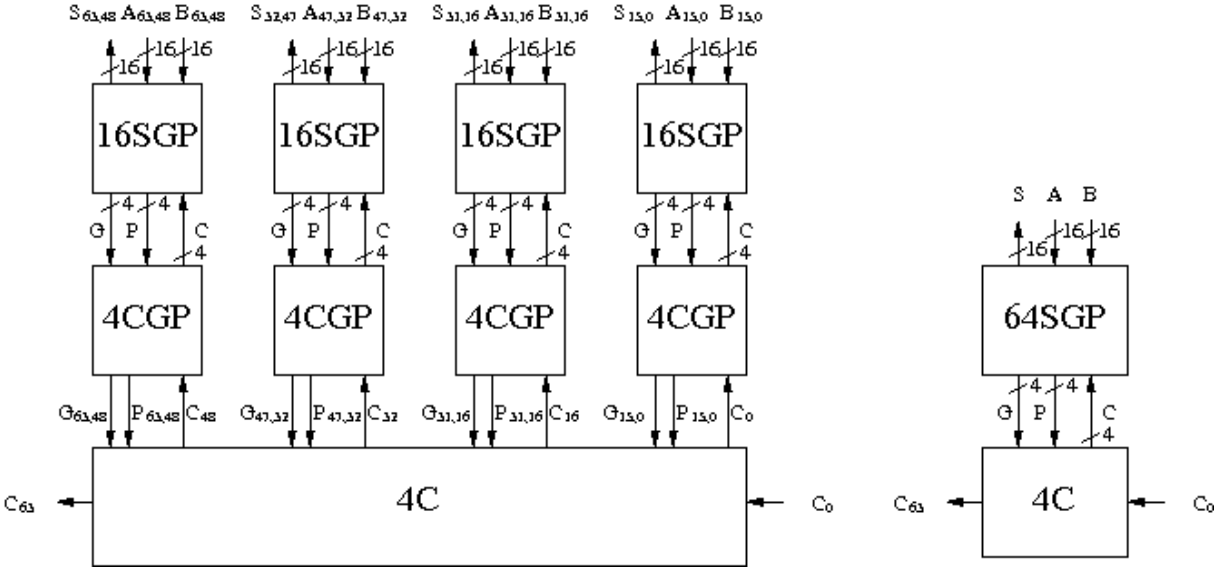
Circuit de calcul des retenues et de G et P





Additionneur 16 bits avec calcul anticipé des retenues

6.5.4 Additionneur 64 bits avec calcul anticipé des retenues



Additionneur 64 bits avec calcul anticipé des retenues

6.6 Additionneur récursif

On commence par définir une opération binaire sur les paires (G,P). Cette opération notée $*$ prend en paramètre deux paires (G, P) et (G', P') de valeurs binaires et retourne une nouvelle paire (G'', P'') = (G, P) $*$ (G', P') définie par les formules suivantes.

G'' = G + PG' et P'' = PP'

Cette opération n'est pas commutative mais elle est associative. Les deux calculs de ((G, P) $*$ (G', P')) $*$ (G'', P'') ou de (G, P) $*$ ((G', P') $*$ (G'', P'')) donnent en effet la paire suivante. Il s'agit pour les connaisseurs d'un produit semi-direct.

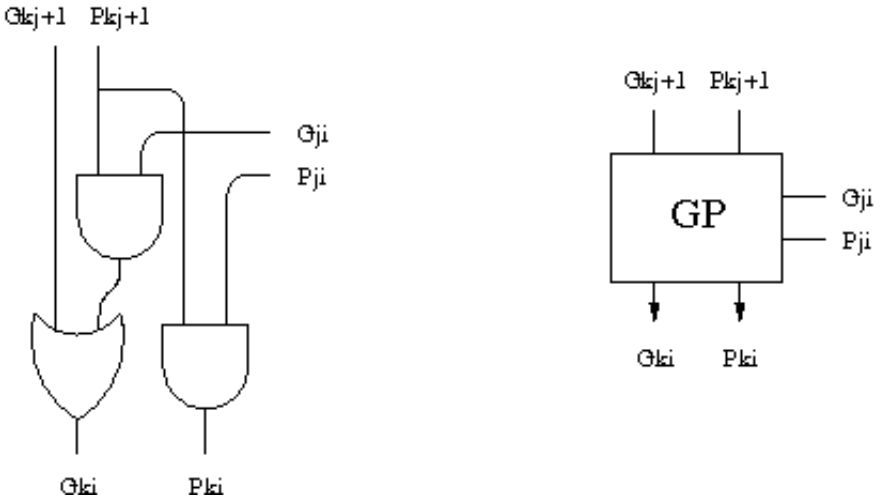
(G+PG'+PP'G'', PP'P'')

Cette opération $*$ permet des calculs rapides de toutes les valeurs G et P en utilisant les formules ci-dessous.

Pour i = k, on a $G_{i,i} = G_i$ et $P_{i,i} = P_i$. Pour $i \le j \le k$, on a la formule suivante qui permet de calculer récursivement toutes les valeurs $G_{k,i}$ et $P_{k,i}$.

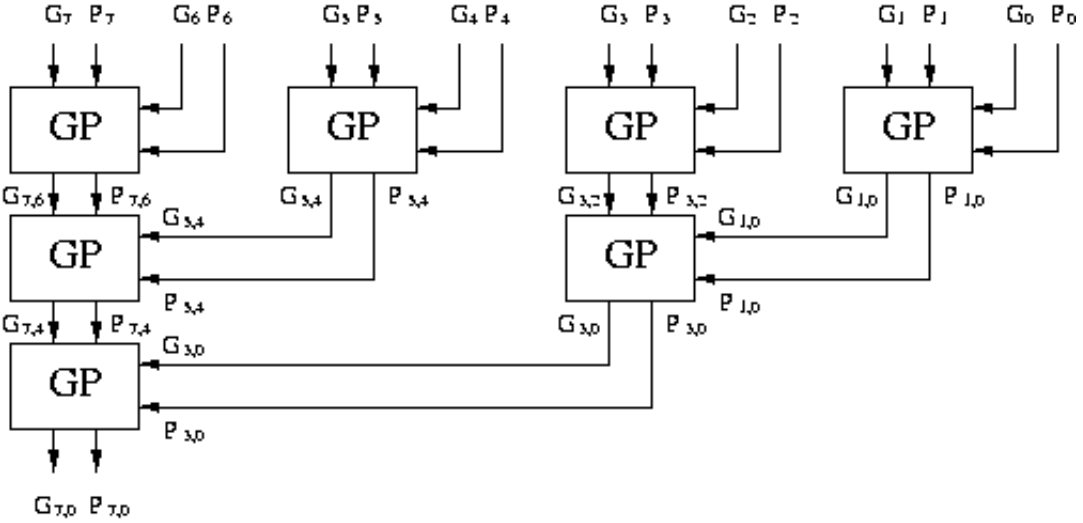
$(G_{k,i}, P_{k,i}) = (G_{k,j+1}, P_{k,j+1}) * (G_{j,i}, P_{j,i})$.

Le circuit GP ci-dessous calcule l'opération $*$. Il prend en entrée deux paires $(G_{k,j+1}, P_{k,j+1})$ et $(G_{j,i}, P_{j,i})$ et calcule la paire $(G_{k,i}, P_{k,i})$ en utilisant la formule ci-dessus.



Cellule de calcul des $G_{k,i}$ et $P_{k,i}$

Le circuit GP peut être utilisé pour former un arbre de calcul des valeurs $G_{k,i}$ et $P_{k,i}$. Cet arbre ne calcule pas toutes les valeurs $G_{k,i}$ et $P_{k,i}$ mais seulement celles nécessaires au calcul des retenues.

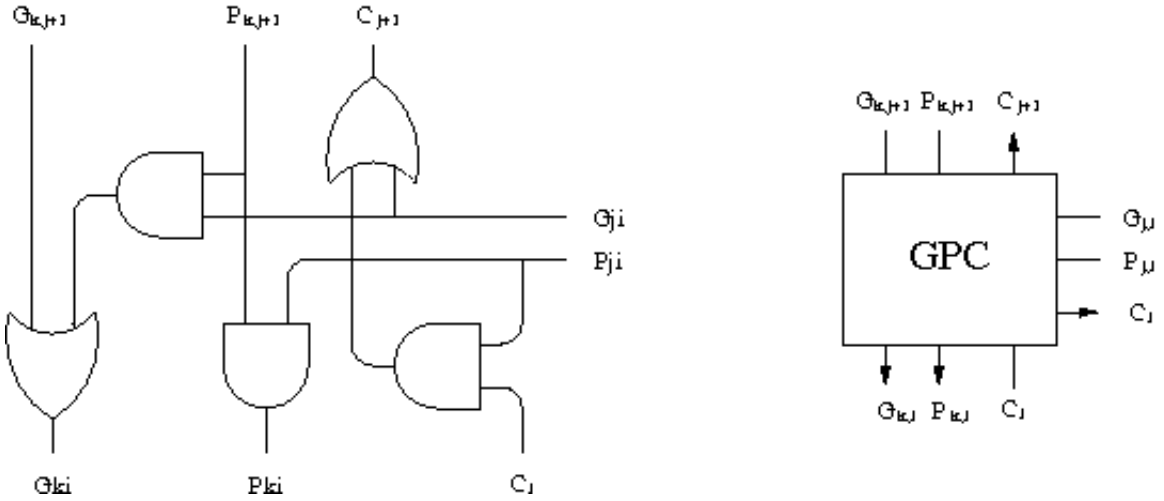


Arbre de calcul des $G_{k,i}$ et $P_{k,i}$

Le circuit GP peut être étendu en un circuit GPC qui effectue en outre le calcul de la retenue en utilisant la formule suivante déjà donnée ci-dessus.

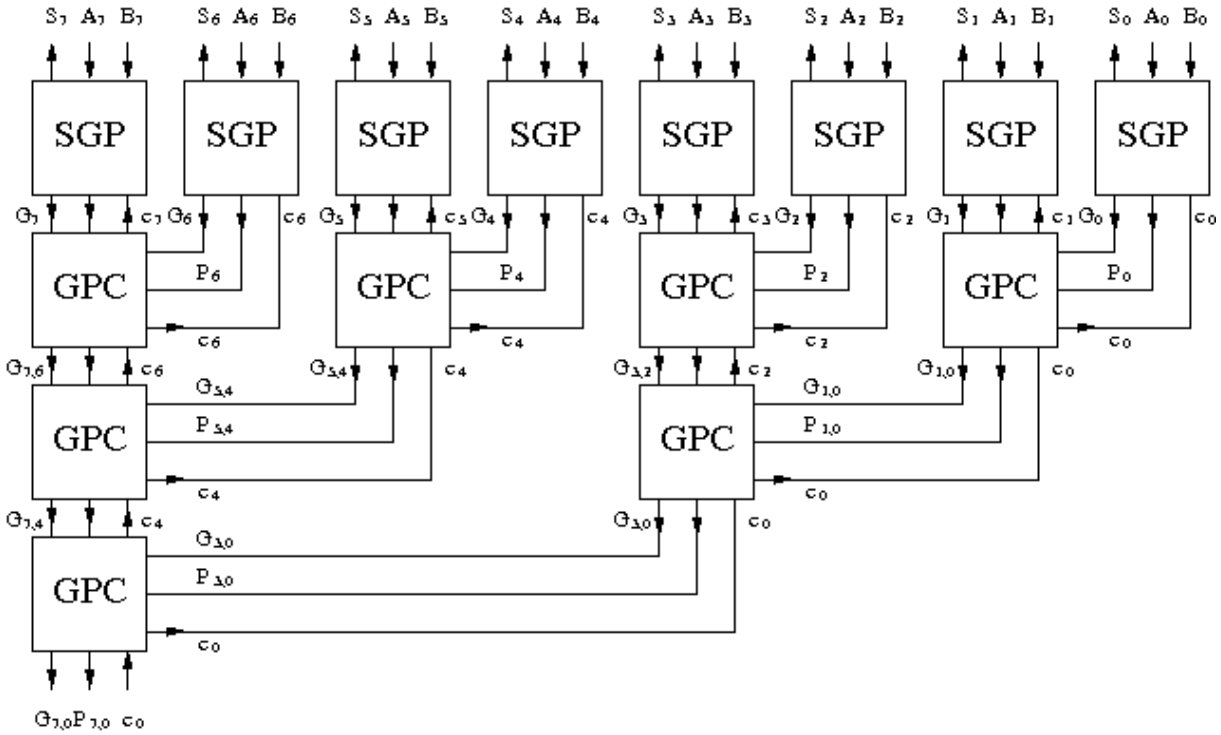
$C_{j+1} = G_{j,i} + P_{j,i} C_i$.

Le circuit GPC prend donc entrée deux paires $(G_{k,j+1}, P_{k,j+1})$ et $(G_{j,i}, P_{j,i})$ et une retenue C_i et calcule la paire $(G_{k,i}, P_{k,i})$ et la retenue C_{j+1}.



Cellule de calcul des $G_{k,i}$, $P_{k,i}$ et C_j

Si on remplace les circuit GP par des circuits GPC dans l'arbre de calcul ci-dessus, on obtient un circuit qui calcule en outre les retenues.



Additionneur récursif 8 bits

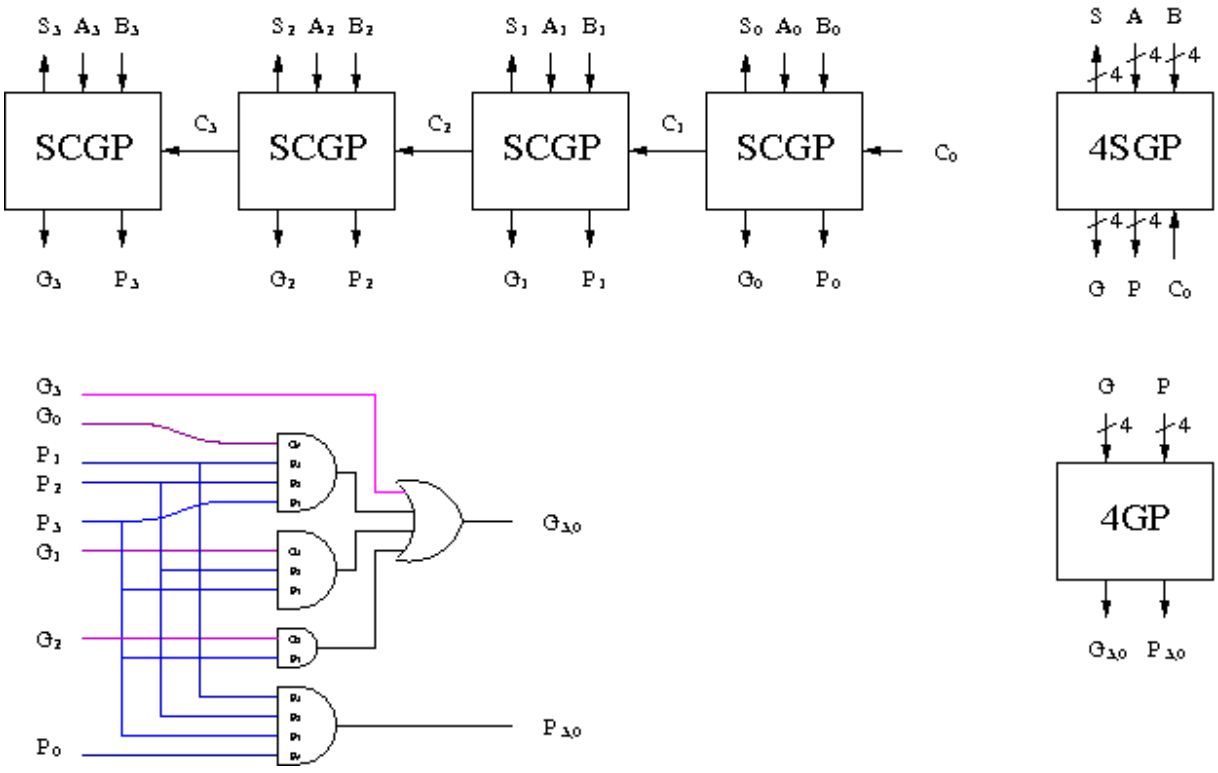
Pour un circuit travaillant sur des entrées de k bits, la profondeur de l'arbre est égale à log k. Le nombre de portes du circuit est donc de l'ordre de k. Le temps de calcul des retenues est proportionnel à log k. Les signaux G et P doivent descendre jusqu'à la racine de l'arbre puis les calculs de retenues doivent remonter jusqu'aux feuilles.

6.7 Additionneur hybride

L'idée générale d'un additionneur hybride est de combiner des techniques différentes de calcul de retenues pour construire un gros additionneur. Une première technique comme la propagation de la retenue peut être utilisée pour construire des petits additionneurs qui sont ensuite regroupés en utilisant une autre technique comme le calcul anticipé de la retenue. On construit ci-dessous un additionneur 16 bits en combinant 4 additionneurs 4 bits par propagation de retenue. Ces 4 additionneurs 4 bits sont assemblés autour d'un circuit d'anticipation de retenue qui calcule leurs 4 retenues d'entrée.

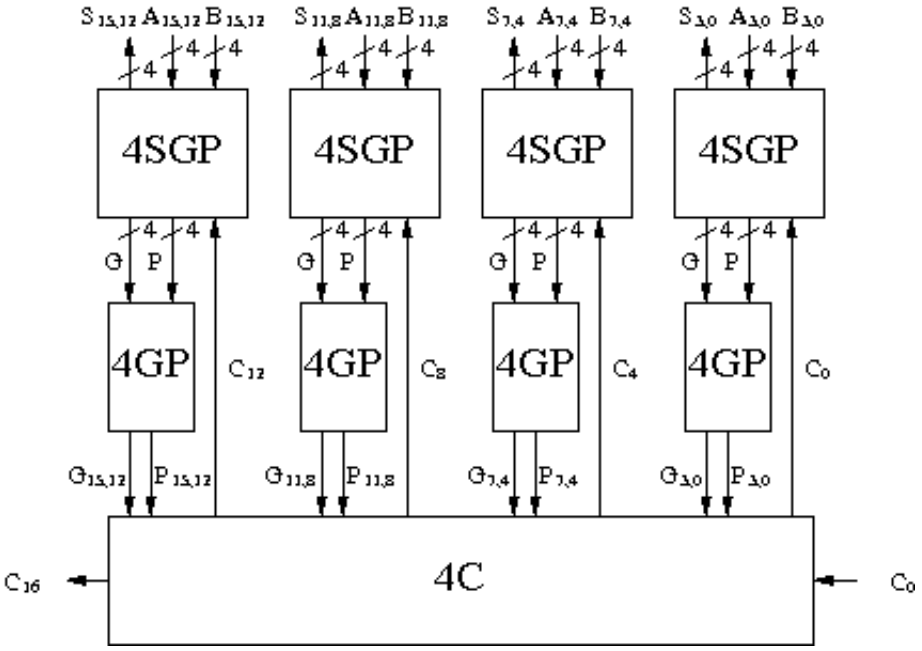
Afin de pouvoir utiliser un circuit d'anticipation de retenue, il faut disposer d'un circuit calculant les valeurs G et P associées à chacun des blocs de 4 bits. Ce circuit GP prend en entrée les 8 valeurs G_0, G_1, G_2, G_3, P_0, P_1, P_2, et P_3 associées aux 4 paires de bits et produit les valeurs $G_{3,0}$ et $P_{3,0}$ du bloc de 4 bits.

On construit 4 additionneurs par propagation de retenue qui calculent également les valeurs G_0, G_1, G_2, G_3, P_0, P_1, P_2, et P_3 associées aux 4 paires de bits.



Additionneur 4 bits et calcul de G et P

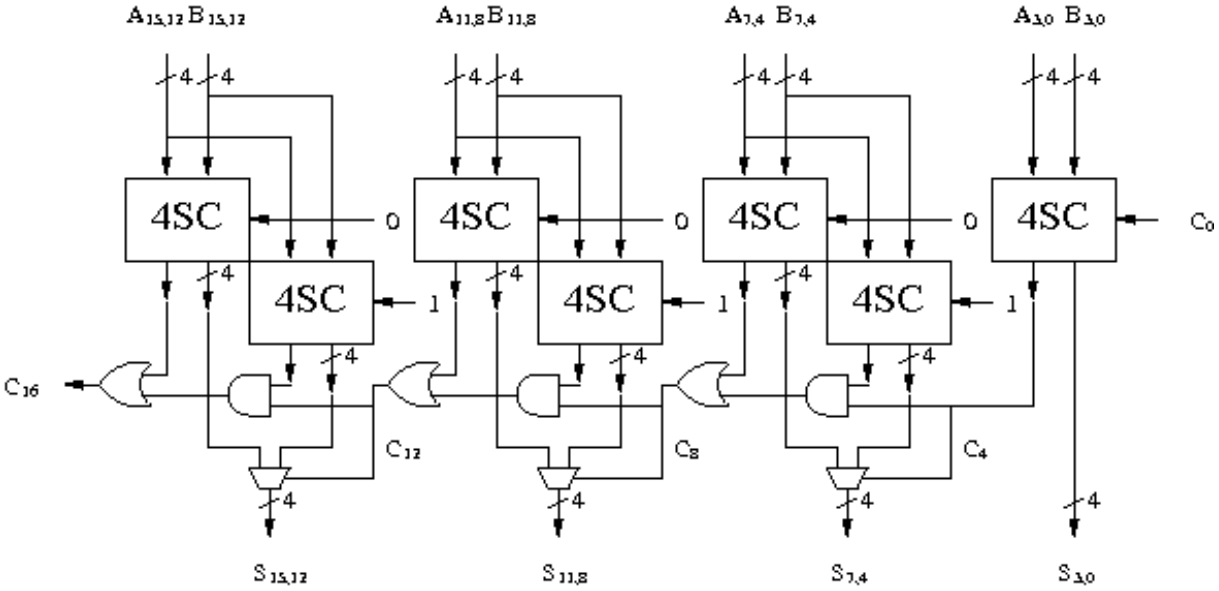
Les quatre additionneurs par propagation de retenue 4 bits 4SGP sont combinés au circuit d'anticipation de retenue 4C par l'intermédiaire des circuits GP de calcul des valeurs G et P des blocs. Le circuit global obtenu est un additionneur hybride 16 bits.



Additionneur hybride 16 bits

6.8 Additionneur par sélection de retenue

L'idée générale de l'additionneur par sélection de retenue est d'effectuer en parallèle le calcul avec une retenue de 0 et le calcul avec une retenue de 1 puis de sélectionner le bon résultat lorsque la retenue est enfin connue.



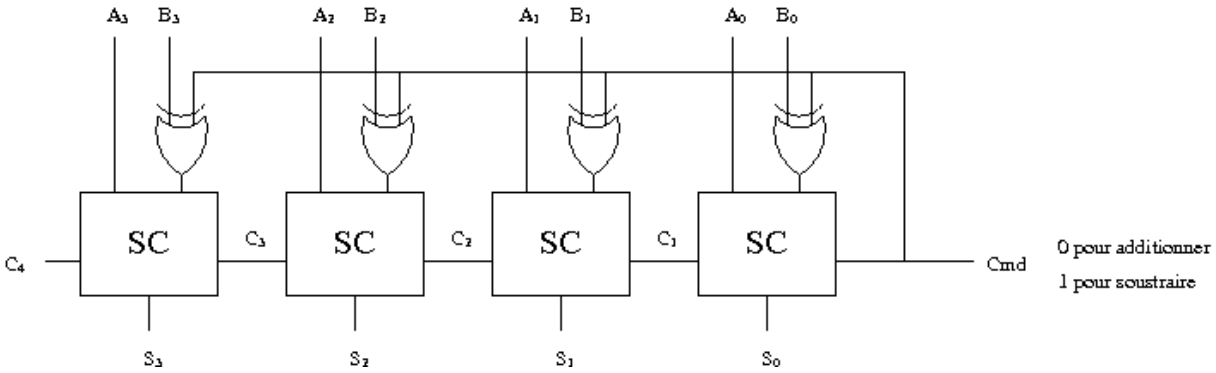
Additionneur par sélection de retenue 16 bits

6.9 Soustracteur

Pour calculer la différence A - B de deux nombre signés A et B, on utilise un circuit qui calcule d'abord l'opposé -B de B puis effectue la somme de A avec -B grâce à un additionneur. Le calcul de -B est réalisé en prenant la négation de B bit à bit puis en ajoutant 1 au résultat obtenu. Ce dernier 1 est en fait ajouté directement à la somme de A et -B en l'injectant comme retenue C_0 à l'additionneur.

Le circuit ci-dessous effectue une somme ou une différence suivant la valeur de la commande Cmd. Si Cmd vaut 0, le circuit calcule la somme A + B. Si, au contraire, Cmd vaut 1, le circuit calcule la différence A - B. En effet, chacune des portes *xor* effectue la négation ou non d'une entrée B_i suivant la valeur de Cmd.

La différence est ici calculée avec un simple additionneur par propagation de retenue mais tout autre additionneur plus sophistiqué aurait pu aussi être utilisé.



Soustracteur 4 bits

7 Mémoires

On appelle *mémoire* un composant électronique permettant de stocker une information sous forme binaire. On distingue les mémoires à lecture seule appelées ROM pour *Read Only Memory* et les mémoires à lecture/écriture appelées improprement RAM pour *Random Access Memory*. Le contenu des mémoires à lecture seule est fixé lors de la fabrication en usine et reste dans la mémoire lorsque celle-ci n'est pas alimentée. Au contraire, les mémoires à lecture/écriture sont *volatiles*. Le contenu est perdu dès qu'elles ne sont plus alimentées.

Il existe des mémoires intermédiaires entre les RAM et les ROM. Elles se comportent comme des ROM dans la mesure où elles ne perdent pas leur contenu sans alimentation mais ce contenu peut toutefois être modifié par un processus spécial. Le contenu d'une EPROM est effacé par une exposition aux ultra-violets. Un nouveau contenu peut alors être écrit par un appareil spécialisé. Les EEPROM peuvent être reprogrammées de manière électrique. Les mémoires *flash* sont des EEPROM dont la reprogrammation est rapide (d'où leur nom).

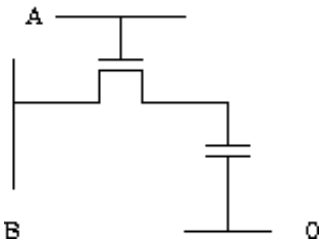
Les mémoires des tous premiers ordinateurs étaient magnétiques. Les mémoires sont maintenant des composants électroniques à base de transistors. Il existe deux types de mémoires qui se distinguent par leur technique de fabrication : les *mémoires dynamiques* et les *mémoires statiques*. Il s'agit dans les deux cas de mémoires volatiles qui nécessitent une alimentation pour conserver leur contenu. La mémoire dynamique est appelée DRAM pour *Dymanic RAM* par opposition à la mémoire statique appelée SRAM pour *Static RAM*.

Mémoire dynamique	**Mémoire statique**
Grande densité d'intégration	Petite densité d'intégration
Bon marché	Chère
Lente	Rapide
Mécanisme de rafraîchissent	

Comme son coût est moindre et que sa densité d'intégration est supérieure, la mémoire dynamique est utilisée pour la mémoire principale de l'ordinateur. Par contre, la mémoire statique est utilisée pour les caches en raison de sa plus grande vitesse.

7.1 Mémoire dynamique

Chaque élément de mémoire dynamique est formé d'un condensateur et d'un transistor de commande (cf. figure ci-dessous). La ligne A est appelée *ligne de commande* ou *ligne d'adresse*. La ligne B est la *ligne de donnée* sur laquelle est lu ou écrit le bit d'information. Le bit d'information est représenté par la charge du condensateur. Lorsque la ligne de commande est à 0, le condensateur est isolé de la ligne de donnée et la charge reste prisonnière du condensateur. Au contraire, lorsque la ligne de commande est à 1, on peut lire le bit en détectant la charge ou écrire un nouveau bit en forçant la ligne de donnée à une valeur.



Mémoire dynamique 1 bit

7.1.1 Types de mémoires dynamiques

Il existe deux types principaux de mémoire dynamique qui se distinguent par leur façon de communiquer avec le processeur. Les premières mémoires dynamiques étaient *asynchrones* alors que les mémoires actuelles sont *synchrones*. Ces dernières sont appelées SDRAM pour *Synchronous Dymanic RAM*.

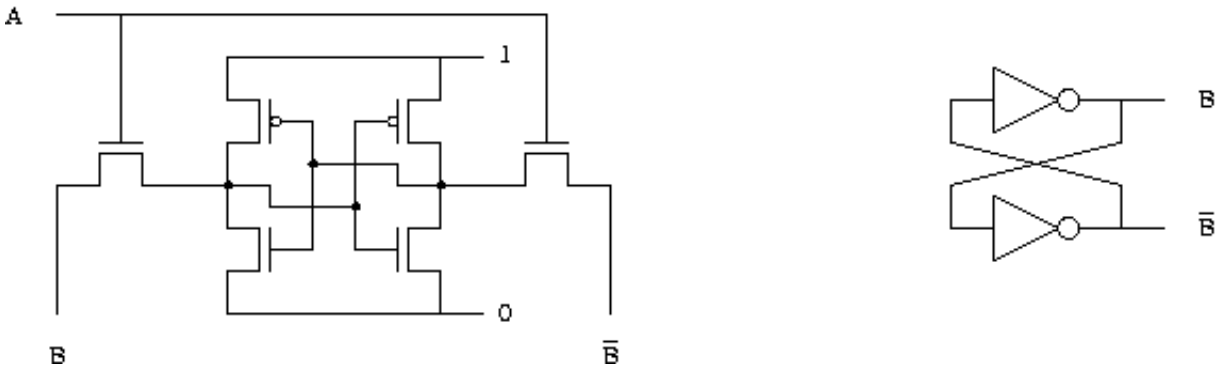
Lorsque le processeur lit une donnée dans une mémoire asynchrone, celui-ci lui envoie l'adresse puis attend que celle-ci lui retourne la donnée à cette adresse. Plusieurs cycles horloge peuvent s'écouler avant que la donnée ne parvienne au processeur. Après réception de la donnée, le processeur peut à nouveau demander une autre donnée à la mémoire. À chaque requête à la mémoire, le processeur reste inactif en attendant que la donnée n'arrive.

Dans le cas de mémoire synchrone, le processeur peut envoyer à la mémoire une nouvelle requête de lecture ou d'écriture avant que celle-ci n'ait fini de traiter la première requête. Les demandes successives sont alors exécutées séquentiellement par la mémoire. Chaque requête est reçue pendant un cycle d'horloge et les données sont délivrées quelques cycles d'horloge plus tard. Le nombre de cycles d'horloge entre la requête et la donnée est fixe. De cette manière, le processeur peut déterminer à quelle requête correspond chaque donnée. Le principe de fonctionnement d'une mémoire synchrone est identique au *pipeline* utilisé pour réaliser un processeur. La mémoire est organisée comme une chaîne où sont traitées les requêtes. La mémoire traite simultanément plusieurs requêtes qui se trouvent à des étapes différentes de la chaîne. Le traitement d'une seule requête prend plusieurs cycles d'horloge mais une requête est traitée à chaque cycle.

Parmi les mémoires synchrones, on distingue encore plusieurs variantes. Les mémoires SDR SDRAM pour *Single Data Rate SDRAM* reçoivent une requête à chaque cycle d'horloge. Les mémoires DDR SDRAM pour *Double Data Rate DRAM* permettent de doubler le débit de données entre le processeur et la mémoire. Une requête est encore traitée à chaque cycle d'horloge mais chaque requête concerne deux mots consécutifs en mémoire. Le premier mot est transmis sur le front montant du signal d'horloge alors que le second est transmis sur front descendant du signal d'horloge. Les mémoires DDR2 SDRAM doublent encore le débit de données en traitant quatre mots consécutifs en mémoire à chaque requête.

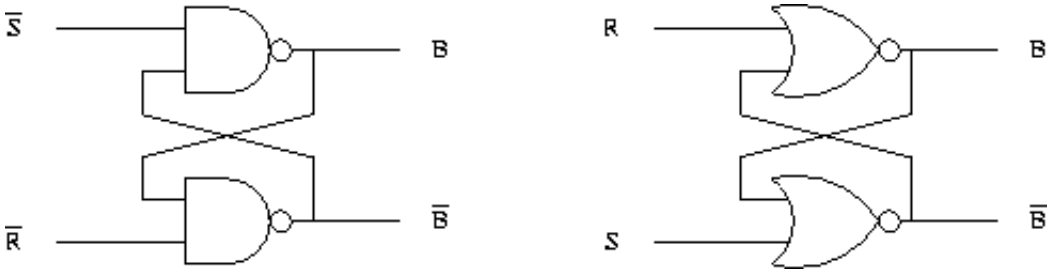
7.2 Mémoire statique

Chaque élément de mémoire statique est formé de six transistors. Quatre de ces six transistors constituent deux inverseurs mis tête-bêche et les deux derniers, commandés par la *ligne d'adresse* A, relient les inverseurs aux lignes de données



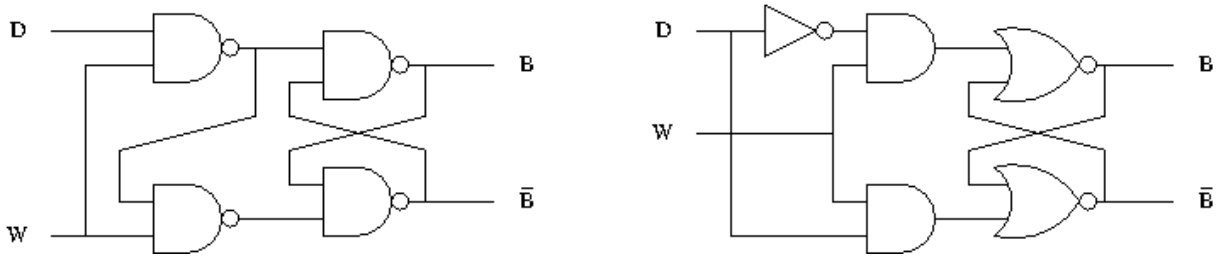
Mémoire statique 1 bit

Le *verrou SR* est un bit de mémoire muni de deux entrées S et R permettant de le positionner à 1 (Set) ou à 0 (Reset). Ce circuit peut être réalisé avec deux portes *nor* ou deux portes *nand*. Dans ce dernier cas, les commandes S et R sont inversées : elles sont actives pour la valeur 0 et inactive pour la valeur 1.



Verrou SR

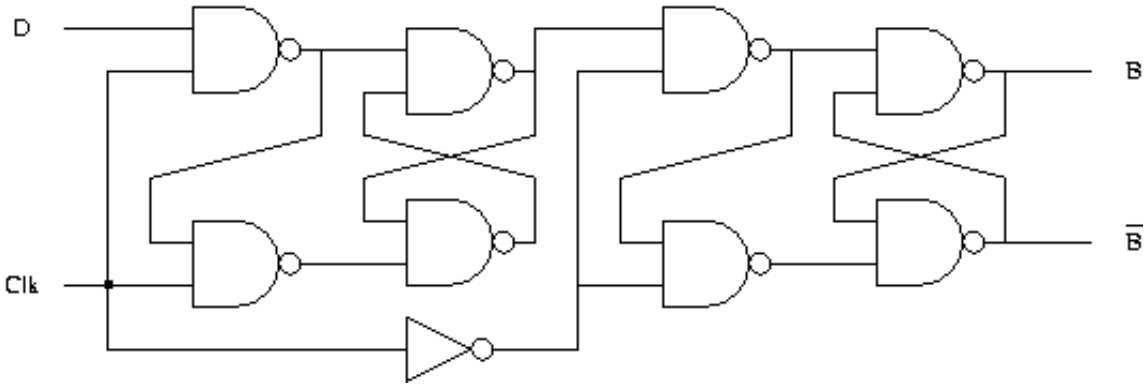
L'inconvénient principal du verrou SR est que son état n'est pas spécifié si les deux commandes S et R sont activées simultanément. Le *verrou D* résout ce problème. Ce verrou a deux entrées D (pour Data) et W (pour Write). La première entrée donne la valeur qui doit être écrite dans le bit de mémoire et la seconde entrée valide l'entrée D. Si W vaut 0, rien n'est écrit dans la mémoire et son état reste inchangé. Si au contraire W vaut 1, la valeur de D est écrite dans la mémoire.



Verrou D (mémoire 1 bit)

La *bascule D* est obtenue en mettant en série deux verrous D avec des entrées W en opposition (négation l'une de l'autre). L'entrée W du premier verrou D est l'entrée Clk de la bascule et sa sortie est l'entrée D du second verrou D. L'entrée W du second verrou est la négation ¬Clk de l'entrée Clk de la bascule. Quand l'entrée Clk de la bascule vaut 1, son entrée D est écrite dans le premier verrou pendant que le second verrou reste inchangé. Quand l'entrée Clk de la bascule vaut 0, le premier verrou reste inchangé et son état est écrit dans le second verrou. Il s'agit d'une sorte de *bufferisation* qui permet de lire et d'écrire simultanément dans une mémoire 1 bit. Pendant que Clk vaut 1, le premier verrou reçoit la nouvelle valeur pendant que l'ancienne valeur est lue dans le second verrou. Pendant que Clk vaut 0, la nouvelle valeur est transférée dans le second verrou.

Les bascules D sont utilisées dans les circuits séquentiels et elle reçoivent alors le signal d'horloge sur l'entrée Clk (pour Clock).

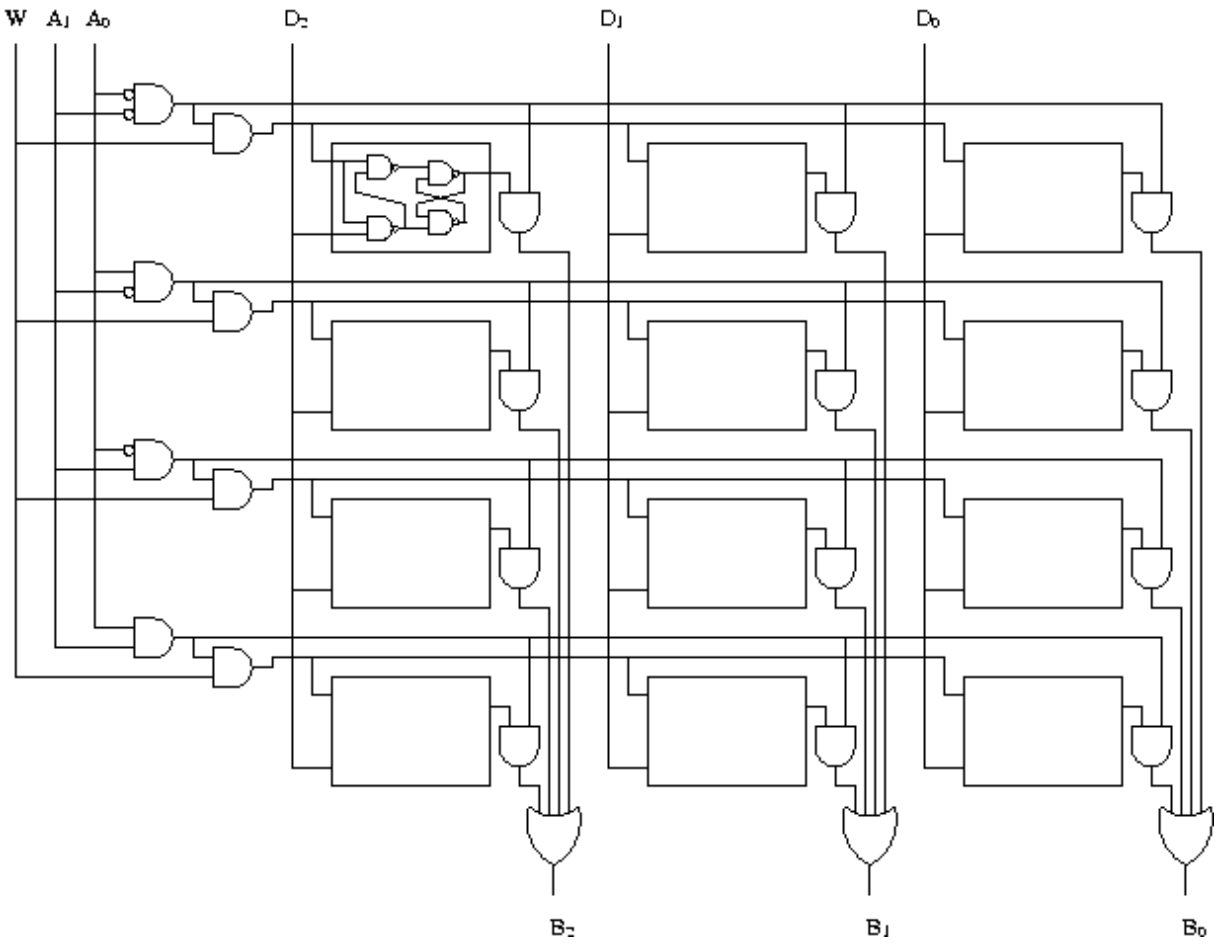


Bascule D (mémoire 1 bit avec bufferisation)

7.3 Organisation de la mémoire

7.3.1 Mémoires k × n bits

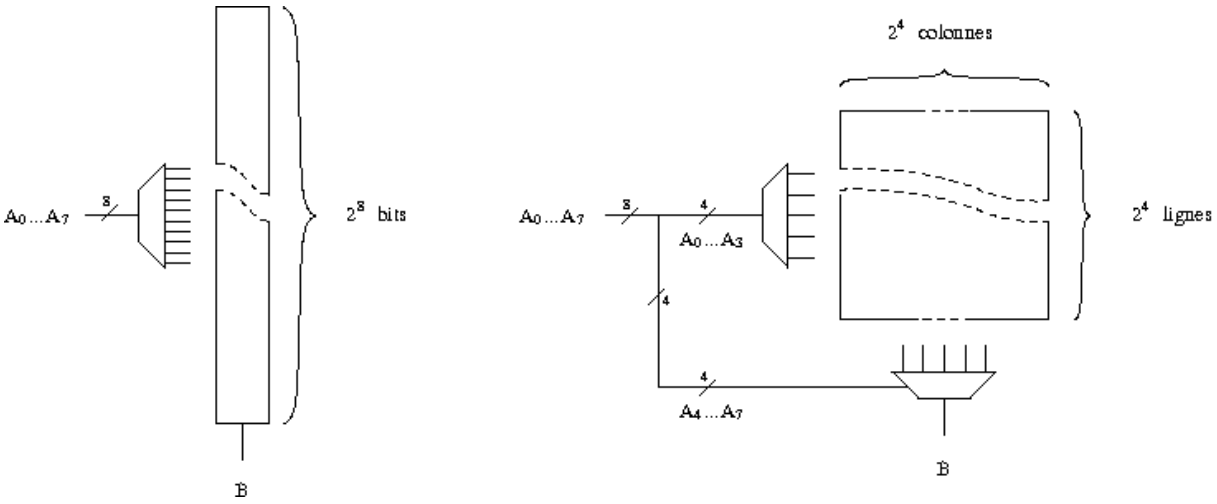
La mémoire peut être vue comme organisée en grille. Chaque ligne correspond à une adresse. Le nombre de lignes est donc une puissance de deux. Ce nombre est égal à 2^k où k est le nombre de lignes d'adresse. Chaque colonne de la grille correspond à un des bits de la donnée transmise. Le nombre de colonnes peut être quelconque même si dans la pratique, c'est aussi une puissance de deux, souvent 8, 16 ou 32.



Mémoire 4 × 3 bits

7.3.2 Mémoires 1 bit

Dans le cas d'une mémoire 1 bit, il y a un intérêt à ne pas l'organiser comme une seule colonne selon le principe précédent mais encore sous forme d'une grille. Il y a alors un gain sur le nombre de portes logiques utilisées pour le décodage des adresses.



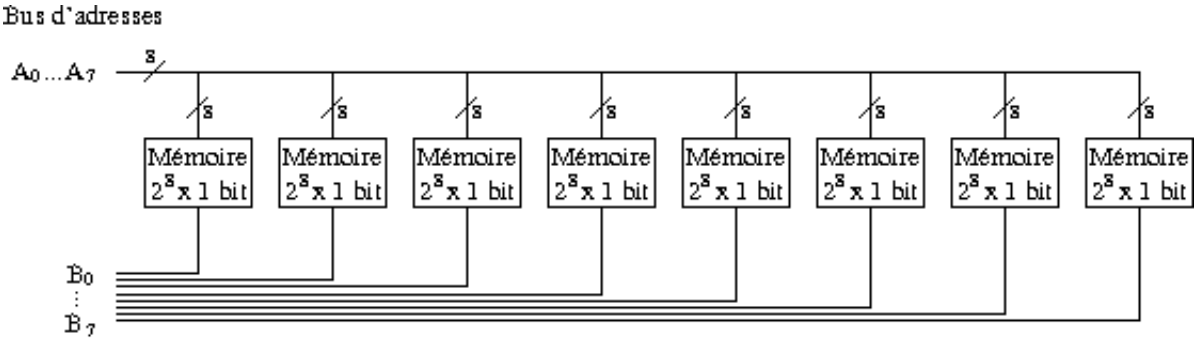
Organisations *colonne* et *grille* d'une mémoire $2^8 \times 1$ bits

Soit par exemple une mémoire $2^8 \times 1$ bits. Si elle est organisée selon une seule colonne, on a un décodeur à 8 entrées qui utilise donc $256 = 2^8$ portes *et logique* ayant chacune 8 entrées. Cette même mémoire peut être organisée selon une grille 16×16. Les quatre premiers bits $A_0 \ldots A_3$ sont envoyés sur un décodeur pour sélectionner la ligne parmi les $16 = 2^4$ lignes. Un multiplexeur commandé par les quatre derniers bits $A_4 \ldots A_7$ permet de sélectionner le bit parmi les 16 bits provenant des 16 colonnes. Le décodeur et le multiplexeur utilisent chacun 16 portes *et logique* ayant chacune 3 ou 4 entrées. Ceci fait un total de 32 portes dans l'organisation en grille et donc un gain de 224 portes portes par rapport à l'organisation en colonne. Dans le cas de mémoires ayant un nombre très important de bits, le gain est bien supérieur.

Dans le cas d'une grille, celle-ci n'a pas besoin d'être carrée comme dans l'exemple. Il est possible d'utiliser tous les intermédiaires entre une seule colonne et un carré. Par contre, ce sont les grilles carrées qui procurent le plus grand gain en nombre de portes logiques. Dans l'organisation colonne, le nombre de portes utilisées est proportionnel au nombre de bits de la mémoire. Chaque bit mémoire nécessite un seul transistor alors que chaque porte logique nécessite plusieurs transistors. Du coup, l'essentiel des transistors est finalement utilisé pour le décodage des adresses. Dans l'organisation en grille, le nombre de portes logiques utilisées pour le décodage est proportionnel à la racine carrée $\sqrt{n}$ de n où n est le nombre de bits de la mémoire.

7.3.3 Couplage de mémoire

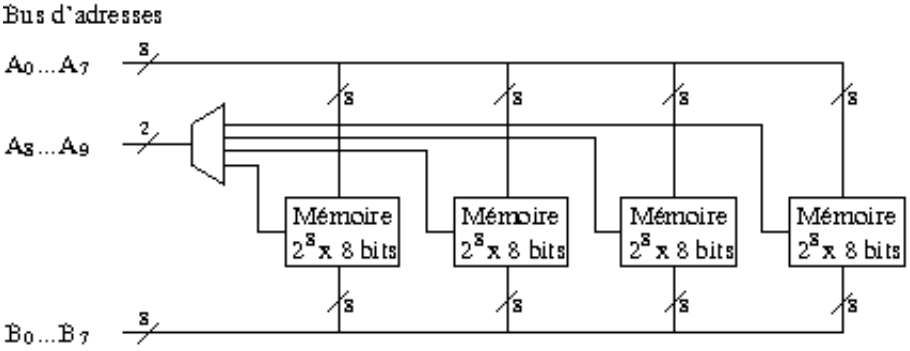
La plupart des mémoires utilisées dans les ordinateurs sont des mémoires 1 bit. Pour obtenir une mémoire dont chaque mot est un octet, il est nécessaire d'assembler huit boîtiers. Ceux-ci sont mis en parallèle dans la mesure où ils reçoivent tous la même adresse. Par contre, chacun des boîtiers reçoit une seule ligne du bus de donnée.



Mémoire $2^8 \times 8$ bits

7.3.4 Augmentation de la taille

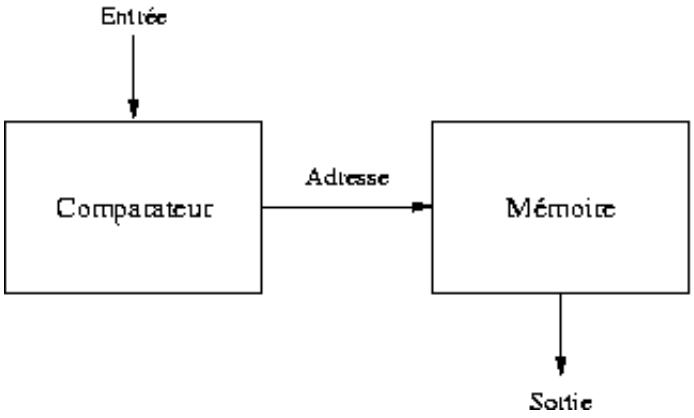
Pour construire une mémoire ayant $2^{k'}$ mots avec des boîtiers mémoire 2^k mots, il faut utiliser $2^{k'-k}$ boîtiers. Les k' bits d'adresses sont alors distribués de la manière suivante. Les k bits de poids faibles $A_0 \ldots A_{k-1}$ sont envoyés sur tous les boîtiers. Les k'-k bits de poids fort $A_k \ldots A_{k'-1}$ arrivent sur un décodeur permettant de sélectionner un seul des $2^{k'-k}$ boîtiers mémoire. Tous les bits du bus de données sont reliés à chacun des boîtiers. Ceci est possible aussi bien en lecture qu'en écriture car les entrée/sorties des boîtiers sont dans un état dit *haute impédance* lorsque le circuit n'est pas sélectionné.



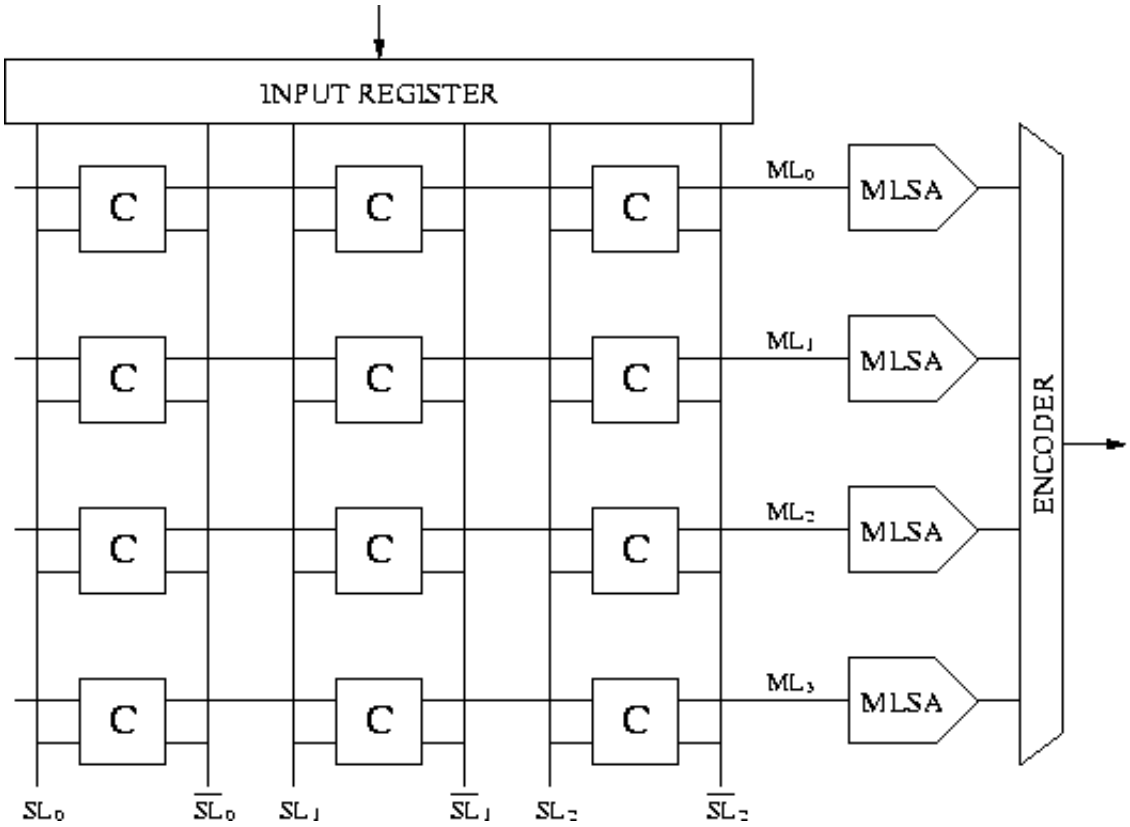
Mémoire $2^{12} \times 8$ bits

7.4 Mémoires associatives

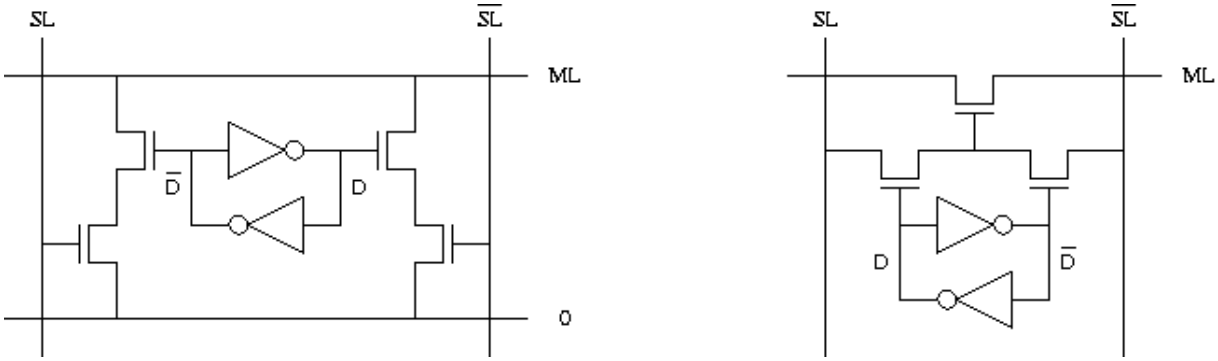
Les mémoires associatives permettent de stocker un ensemble de paires (clés,valeurs), de rechercher en temps constant si une valeur est associée à une clé. Un tel dispositif est utilisé dans les routeurs réseaux et surtout pour le *Translation Lookaside Buffer* de la mémoire virtuelle. La mémoire associative sert alors à traduire les adresses virtuelles en adresses physiques.



Principe de la mémoire associative



Organisation d'une mémoire associative

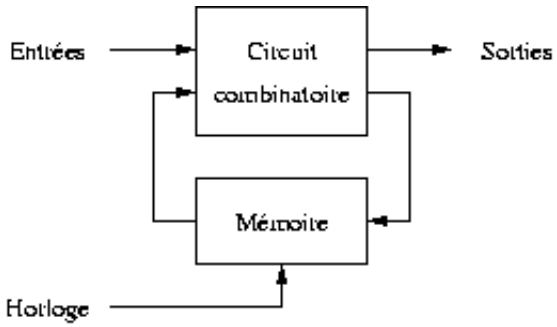


Cellule de mémoire associative

8 Circuits séquentiels

Il n'y a aucune notion de temps dans les circuits combinatoires. Un circuit combinatoire est simplement une fonction qui calcule des valeurs de sortie en fonction des valeurs d'entrée. On peut ajouter la notion de temps avec un *signal d'horloge*. Un signal d'horloge est un signal carré périodique qui peut être produit par un quartz.

8.1 Principe



Principe du circuit séquentiel

8.2 Horloge

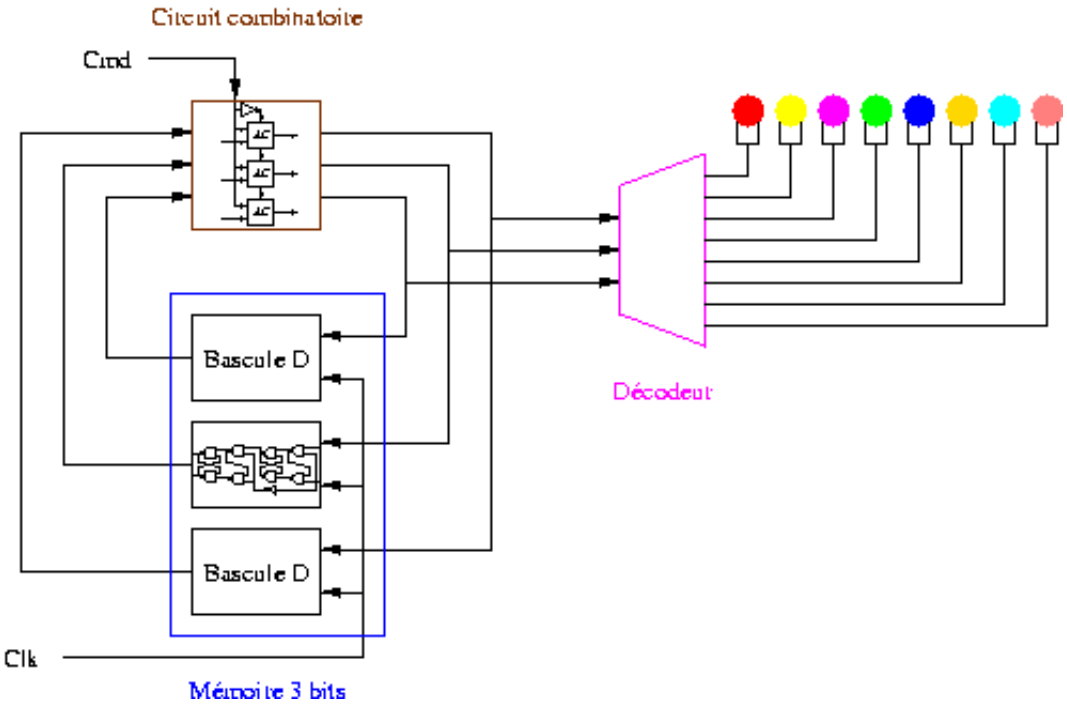
Signal d'horloge

8.3 Exemple

Le petit circuit ci-dessous permet d'incrémenter ou de décrémenter un entier codé sur trois bits. Soit n l'entier représenté en binaire par les trois bits d'entrée $A_2A_1A_0$. Si la valeur en entrée de `Cmd` est 0, l'entier représenté par les trois bits de sortie $S_2S_1S_0$ est n+1. Si au contraire la valeur en entrée de `Cmd` est 1, l'entier représenté par les trois bits de sortie $S_2S_1S_0$ est n-1. Les calculs de n+1 ou n-1 sont bien sûr effectués modulo 8.

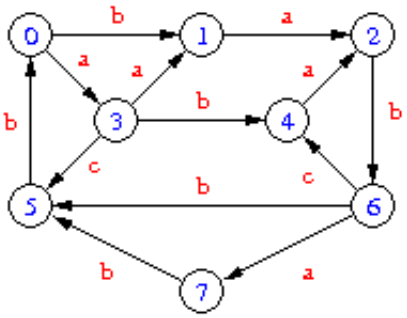
Incrémenteur/décrémenteur à commande

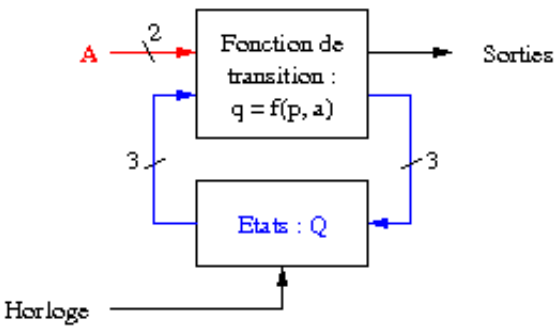
Le circuit ci-dessous réalise une *guirlande lumineuse*. Les lampes s'allument à tour de rôle de manière cyclique. La vitesse du processus est contrôlée par le signal d'horloge `Clk`. Le sens de rotation du cycle est déterminé par l'entrée `Cmd`.



Guirlande à commande

8.4 Automate





Implantation d'un automate

9 Architecture d'un micro-processeur

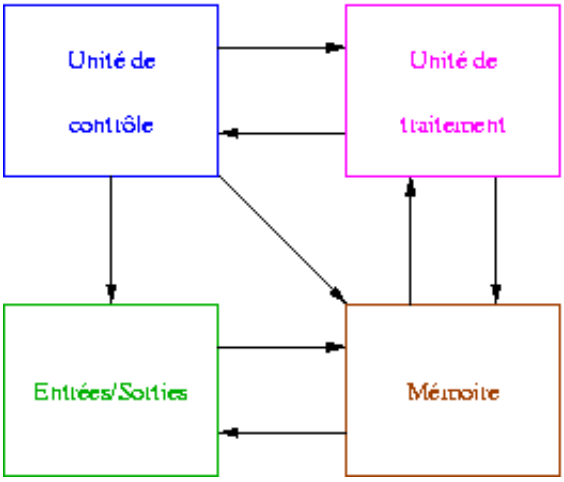
Avant d'étudier l'architecture détaillée d'un micro-processeur précis, on aborde ici l'architecture globale d'un micro-processeur. Tous les micro-processeurs conçus jusqu'à aujourd'hui s'organisent globalement de même façon. Ils peuvent être décomposés en quatre composants décrits par le modèle de von Neumann.

9.1 Modèle de von Neumann

Le modèle de von Neumann donne les quatre composants essentiels qui constituent un micro-processeur. Il décrit également les interactions entre ces différents composants.

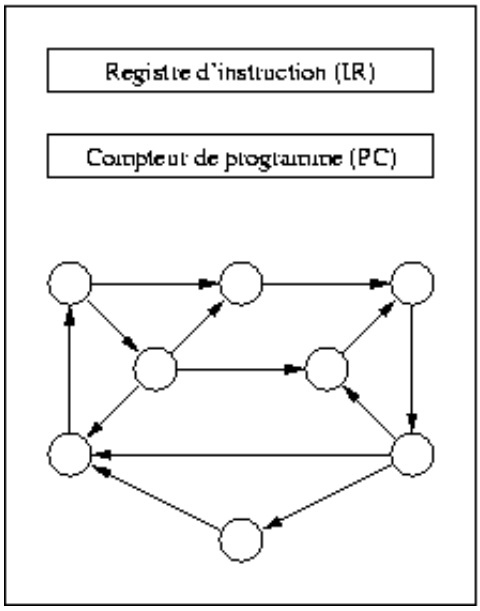
Les quatre composants du modèle de von Neumann sont les suivants.

1. unité de contrôle
2. unité de traitement
3. mémoire
4. unité d'entrées/sorties

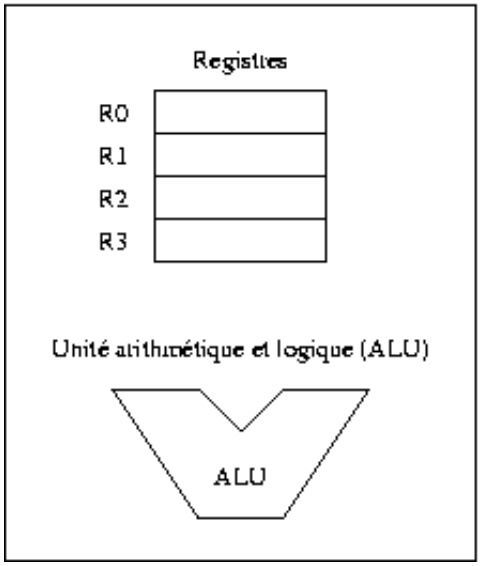


Modèle de von Neumann

9.2 Organisation interne des composants



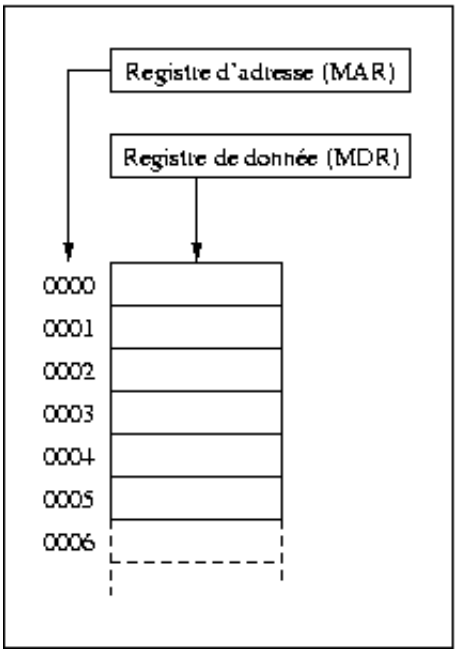
Unité de contrôle



Unité de traitement

9.2.1 Mémoire

Le *registre d'adresse* (*Memory Address Register*) est le registre où est stocké l'adresse de la case mémoire lue ou écrite lors d'un accès à la mémoire. La donnée transite par le *registre de donnée* (*Memory Data Register*). Lors d'une lecture, la donnée parvient au processeur dans ce registre. Lors d'une écriture, la donnée est placée dans ce registre par le processeur avant d'être écrite dans la case mémoire.



Mémoire

9.3 Instructions

9.4 Cycle d'exécution

1. chargement de l'instruction
2. décodage de l'instruction
3. calcul des adresses des opérandes
4. chargement des opérandes
5. exécution
6. mise en place du résultat

9.5 Registres PC et IR

Le *compteur de programme* (PC pour *Program counter*) contient en permanence l'adresse de la prochaine instruction à exécuter. À chaque début de cycle d'exécution, l'instruction à exécuter est chargée dans le registre IR à partir de l'adresse contenue dans le registre PC. Ensuite, le registre PC est incrémenté pour pointer sur l'instruction suivante.

Le *registre d'instruction* (IR) contient l'instruction en cours d'exécution. Ce registre est chargé au début du cycle d'exécution par l'instruction dont l'adresse est donnée par le compteur de programme PC.

9.6 Codage des instructions

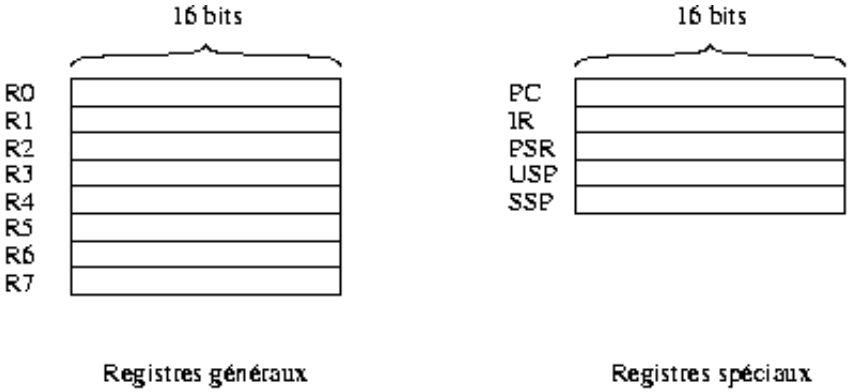
Les instructions exécutés par le processeur sont stockées en mémoire. Toutes les instructions possibles sont représentées par des codes. Le code d'une instruction est appelé *op-code*. Les *op-code* peuvent être de longueur fixe comme dans le LC-3 ou de longueur variable comme dans le Pentium.

10 Micro-processeur LC-3

Le micro-processeur LC-3 est à vocation pédagogique. Il n'existe pas de réalisation concrète de ce processeur mais il existe des simulateurs permettant d'exécuter des programmes. L'intérêt de ce micro-processeur est qu'il constitue un bon compromis de complexité. Il est suffisamment simple pour qu'il puisse être appréhendé dans son ensemble et que son schéma en portes logiques soit accessible. Il comprend cependant les principaux mécanismes des micro-processeurs (appels système, interruptions) et son jeu d'instructions est assez riche pour écrire des programmes intéressants.

10.1 Registres

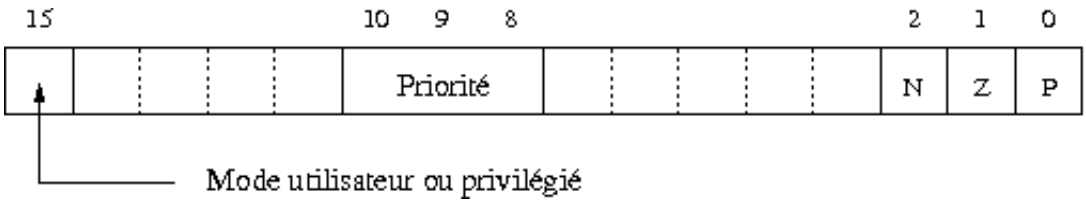
Le micro-processeur LC-3 dispose de 8 registres généraux 16 bits appelés R0,…,R7. Il possède aussi quelques registres spécifiques dont l'utilisation est abordée plus tard. Le registre PSR (Program Status Register) regroupe plusieurs indicateurs binaires dont l'indicateur de mode (mode utilisateur ou mode privilégié), les indicateurs n, z et p qui sont testés par les branchements conditionnels ainsi que le niveau de priorité des interruptions. Les registres USP (User Stack Pointer) et SSP (System Stack Pointer) permettent de sauvegarder le registre R6 suivant que le programme est en mode privilégié ou non. Comme tous les micro-processeurs, le LC-3 dispose d'un compteur de programme PC et d'un registre d'instruction IR qui sont tous les deux des registres 16 bits.



Registres du LC-3

10.1.1 Registre PSR

Le registre PSR regroupe plusieurs informations différentes. Le bit de numéro 15 indique si le processeur est en mode utilisateur (0) ou privilégié (1). Les trois bits de numéros 10 à 8 donnent la priorité d'exécution. Les trois bits de numéros 2 à 0 contiennent respectivement les indicateurs n, z et p.



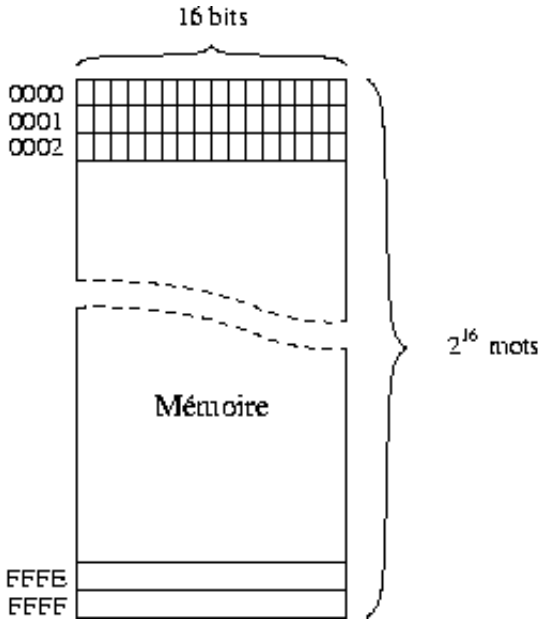
Registre PSR

10.2 Indicateurs N, Z et P

Les *indicateurs* sont des registres 1 bit. Les trois indicateurs n, z et p font, en fait, partie du registre spécial PSR. Ils sont positionnés dès qu'une nouvelle valeur est mise dans un des registres généraux R0,…,R7. Ceci a lieu lors de l'exécution d'une instruction logique (`NOT` et `AND`), arithmétique (`ADD`) ou d'une instruction de chargement (`LD`, `LDI`, `LDR` et `LEA`). Ils indiquent respectivement si cette nouvelle valeur est négative (n), nulle (z) et positive (p). Ces indicateurs sont utilisés par l'instruction de branchement conditionnel `BR` pour savoir si le branchement doit être pris ou non.

10.3 Mémoire

La mémoire du LC-3 est organisée par mots de 16 bits. L'adressage du LC-3 est également de 16 bits. La mémoire du LC-3 est donc formée de 2^{16} mots de 16 bits, c'est-à-dire 128 KiB avec des adresses de 0000 à FFFF en hexadécimal. Cette organisation de la mémoire est inhabituelle mais elle simplifie le câblage. La mémoire de la plupart des micro-processeurs est organisée par octets (mots de 8 bits). Par contre, ces micro-processeurs ont la possibilité de charger directement 2, 4 ou 8 octets. Pour certains, ce bloc doit être aligné sur une adresse multiple de 2, 4 ou 8 alors que ce n'est pas nécessaire pour d'autres.



Mémoire du LC-3

10.4 Instructions

Les instructions du LC-3 se répartissent en trois familles : les instructions arithmétiques et logiques, les instructions de chargement et rangement et les instructions de branchement (appelées aussi *instructions de saut* ou encore *instructions de contrôle*).

Les instructions arithmétiques et logiques du LC-3 sont au nombre de trois. Le LC-3 possède une instruction `ADD` pour l'addition, une instruction `AND` pour le *et logique* bit à bit et une instruction `NOT` pour la négation bit à bit. Le résultat de ces trois opérations est toujours placé dans un registre. Les deux opérandes peuvent être soit les contenus de deux registres soit le contenu d'un registre et une constante pour `ADD` et `AND`.

Les instructions de chargement et rangement comprennent des instructions (avec des noms commencent par LD) permettant de charger un registre avec une valeur et des instructions (avec des noms commencent par ST) permettant de ranger en mémoire le contenu d'un registre. Ces instructions se différencient par leurs *modes d'adressage* qui peut être immédiat, direct, indirect ou relatif. Les instructions de chargement sont LD, LDI, LDR, LEA et les instructions de rangement sont ST, STI et STR.

Les instructions de branchement comprennent les deux instructions de saut BR et JMP, les deux instructions d'appel de sous-routine JSR et JSRR, une instruction TRAP d'appel système, une instruction RET de retour de sous-routine et une instruction RTI de retour d'interruption.

10.4.1 Description des instructions

10.4.1.1 Instructions arithmétiques et logiques

Ces instructions du LC-3 n'utilisent que les registres aussi bien pour les sources que pour la destination. Ceci est une caractéristique des architectures RISC. Le nombre d'instructions arithmétiques et logiques du LC-3 est réduit au strict nécessaire. Les micro-processeurs réels possèdent aussi des instructions pour les autres opérations arithmétiques (soustraction, multiplication, division) et les autres opérations logiques (*ou logique, ou exclusif*). Ils possèdent aussi des instructions pour l'addition avec retenue et les décalages.

10.4.1.1.1 Instruction NOT

L'instruction NOT permet de faire le *non logique* bit à bit d'une valeur 16 bits. Sa syntaxe est NOT DR,SR où DR et SR sont les registres destination et source.

10.4.1.1.2 Instruction ADD

L'instruction ADD permet de faire l'addition de deux valeurs 16 bits. Elle convient pour les additions pour les nombres signés ou non puisque les nombres sont représentés en complément à 2. Elle a deux formes différentes. Dans la première forme, les deux valeurs sont les contenus de deux registres généraux. Dans la seconde forme, la première valeur est le contenu d'un registre et la seconde est une constante (adressage immédiat). Dans les deux formes, le résultat est rangé dans un registre.

La première forme a la syntaxe ADD DR,SR1,SR2 où DR est le *registre destination* où est rangé le résultat et SR1 et SR2 sont les *registres sources* d'où proviennent les deux valeurs. La seconde forme a la syntaxe ADD DR,SR1,Imm5 où DR et SR1 sont encore les registres destination et source et Imm5 est une constante codée sur 5 bits ($-16 \leq Imm5 \leq 15$). Avant d'effecteur l'opération, la constante Imm5 est étendue de façon signée sur 16 bits en recopiant le bit 4 sur les bits 5 à 15.

10.4.1.1.3 Instruction AND

L'instruction AND permet de faire le *et logique* bit à bit de deux valeurs 16 bits. Elle a deux formes similaires à celles de l'instruction ADD de syntaxes AND DR,SR1,SR2 et AND DR,SR1,Imm5.

10.4.1.2 Instructions de chargement et rangement

Les instructions de chargement permettent de charger un des registres généraux avec un mot en mémoire alors que les instructions de rangement permettent de ranger en mémoire le contenu d'un de ces registres. Ce sont les seules instructions faisant des accès à la mémoire. Ces différentes instructions se différencient par leur mode d'adressage. Un *mode d'adressage* spécifie la façon dont l'adresse mémoire est calculée. Le micro-processeur LC-3 possède les principaux modes d'adressage qui sont les modes d'adressage immédiat, direct, relatif et indirect. La terminologie pour les modes d'adressage n'est pas fixe car chaque constructeur a introduit ses propres termes. Le même mode peut avoir des noms différents chez deux constructeurs et le même nom utilisé par deux constructeurs peut recouvrir des modes différents.

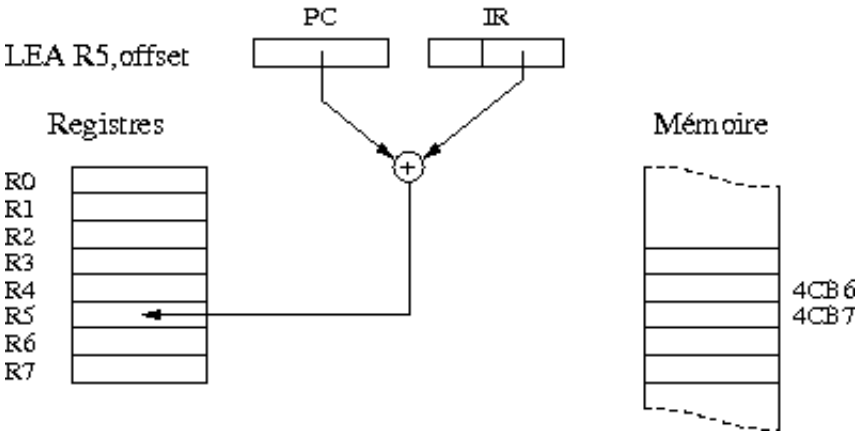
Il faut faire attention au fait que presque tous les modes d'adressage du LC-3 sont relatifs au compteur de programme. Cette particularité est là pour compenser la petite taille des offsets qui permettent un adressage à un espace réduit.

Exceptée l'instruction LEA, les instructions de chargement et rangement vont par paire. Pour chaque mode d'adressage, il y a une instruction de chargement dont le mnémonique commence par LD pour *LoaD* et une instruction de rangement dont le mnémonique commence par ST pour *STore*.

Toutes les instructions de chargement et rangement contiennent un offset. Cet offset n'est en général pas donné explicitement par le programmeur. Celui-ci utilise des étiquettes et l'assembleur se charge de calculer les offsets.

10.4.1.2.1 Instruction LEA

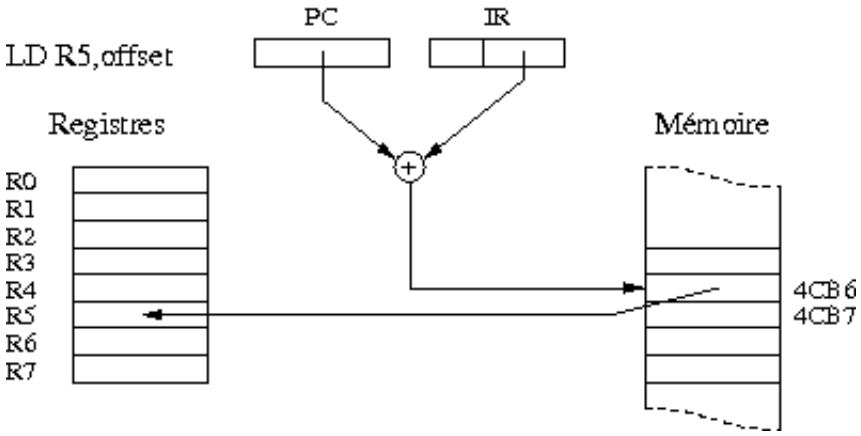
L'instruction LEA pour *Load Effective Address* charge dans un des registres généraux la somme du compteur de programme et d'un offset codé dans l'instruction. Ce mode adressage est généralement appelé adressage *absolu* ou *immédiat* car la valeur chargée dans le registre est directement contenue dans l'instruction. Aucun accès supplémentaire à la mémoire n'est nécessaire. Cette instruction est par exemple utilisée pour charger dans un registre l'adresse d'un tableau.



Adressage immédiat

10.4.1.2.2 Instructions LD et ST

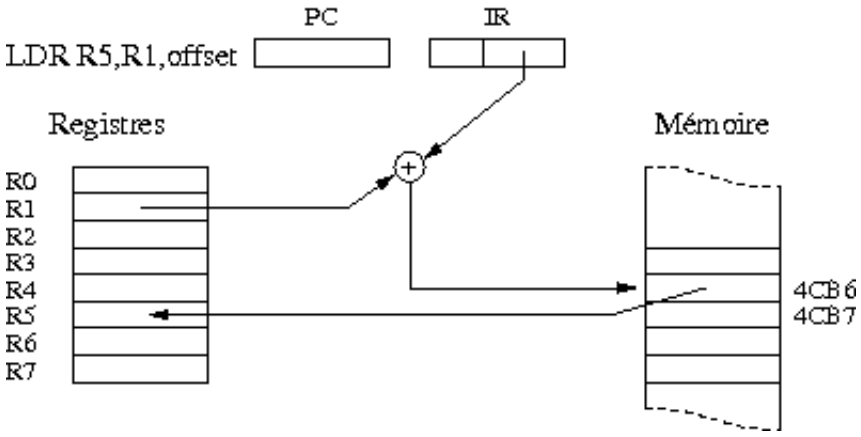
Les instructions LD et ST sont les instructions générales de chargement et rangement. L'instruction LD charge dans un des registres généraux le mot mémoire à l'adresse égale à la somme du compteur de programme et d'un offset codé dans l'instruction. L'instruction ST range le contenu du registre à cette même adresse. Ce mode adressage est généralement appelé adressage *direct*. Il nécessite un seul accès à la mémoire.



Adressage direct

10.4.1.2.3 Instructions LDR et STR

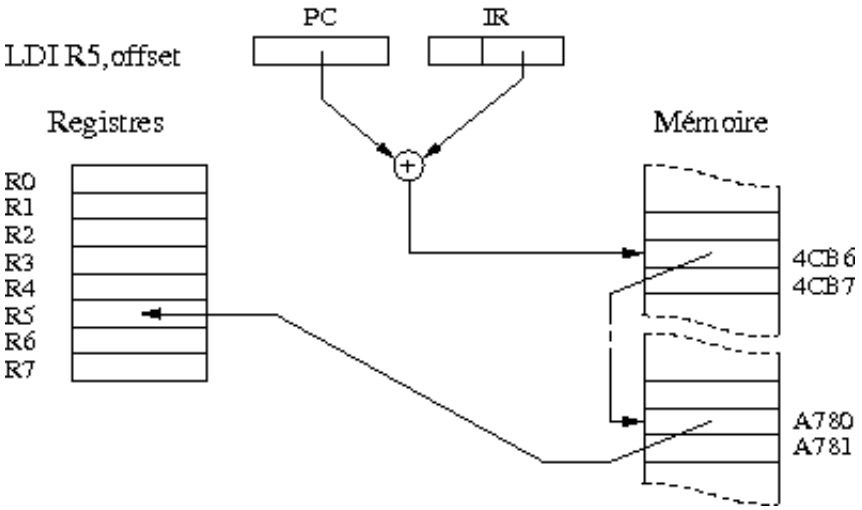
L'instruction LDR charge dans un des registres généraux le mot mémoire à l'adresse égale à la somme du registre de base et d'un offset codé dans l'instruction. L'instruction STR range le contenu du registre à cette même adresse. Ce mode adressage est généralement appelé adressage *relatif* ou *basé*. Il nécessite un seul accès à la mémoire. La première utilité de ces instructions est de manipuler les objets sur la pile comme par exemple les variables locales des fonctions. Elles servent aussi à accéder aux champs des structures de données. Le registre de base pointe sur le début de la structure et l'offset du champ dans la structure est codé dans l'instruction.



Adressage relatif

10.4.1.2.4 Instructions `LDI` et `STI`

Les instructions `LDI` et `STI` sont les instructions les plus complexes de chargement et rangement. Elle mettent en œuvre une double indirection. Elles calculent, comme les instructions `LD` et `SD`, la somme du compteur de programme et de l'offset. Elle chargent la valeur contenue à cette adresse puis utilisent cette valeur à nouveau comme adresse pour charger ou ranger le registre. Ce mode adressage est généralement appelé adressage *indirect*. Il nécessite deux accès à la mémoire.



Adressage indirect

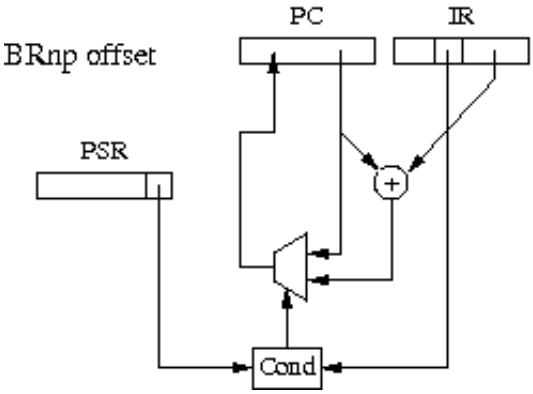
10.4.1.3 Instructions de branchements

10.4.1.3.1 Instruction `BR`

L'instruction générale de branchement est l'instruction `BR`. Elle modifie le déroulement normal des instructions en changeant la valeur contenue par le registre PC. Cette nouvelle valeur est obtenue en ajoutant à la valeur de PC un offset codé sur 9 bits dans l'instruction.

Le branchement peut être inconditionnel ou conditionnel. Le cas *conditionnel* signifie que le branchement est réellement exécuté seulement si une condition est vérifiée. Certains des trois indicateurs n, z, et p sont examinés. Si au moins un des indicateurs examinés vaut 1, le branchement est pris. Sinon, le branchement n'est pas pris et le déroulement du programme continue avec l'instruction suivante.

On rappelle que les trois indicateurs n, z et p sont mis à jour dès qu'une nouvelle valeur est chargée dans un des registres généraux. Ceci peut être réalisé par une instruction arithmétique ou logique ou par une instruction de chargement. Les indicateurs qui doivent être pris en compte par l'instruction `BR` sont ajoutés au mnémonique `BR` pour former un nouveau mnémonique. Ainsi l'instruction `BRnp` exécute le branchement si l'indicateur n ou l'indicateur p vaut 1, c'est-à-dire si l'indicateur z vaut 0. Le branchement est alors pris si la dernière valeur chargée dans un registre est non nulle.



Branchement conditionnel

Les programmes sont écrits en langage d'assembleur qui autorise l'utilisation d'étiquettes (*labels* en anglais) qui évitent au programmeur de spécifier explicitement les offsets des branchements. Le programmeur écrit simplement une instruction `BR label` et l'assembleur se charge de calculer l'offset, c'est-à-dire la différence entre l'adresse de l'instruction de branchement et l'adresse désignée par l'étiquette `label`.

Comme l'offset est codé sur 9 bits, l'instruction `BR` peut uniquement atteindre une adresse située de -255 à 256 mots mémoire. Le décalage d'une unité est dû au fait que le calcul de la nouvelle adresse est effectué après l'incrémentation du compteur de programme qui a lieu lors de la phase de chargement de l'instruction.

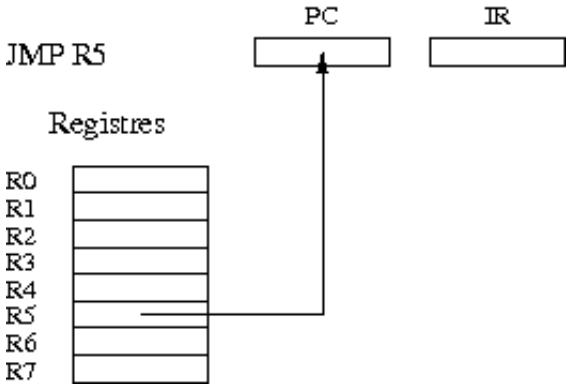
10.4.1.3.2 Instruction `JMP`

L'instruction `JMP` permet de charger le compteur de programme PC avec le contenu d'un des registres généraux. L'intérêt de cette instruction est multiple.

1. L'instruction `JMP` compense la faiblesse de l'instruction `BR` qui peut uniquement effectuer un branchement proche. L'instruction `JMP` permet de sauter à n'importe quelle adresse. Si l'étiquette `label` désigne une adresse trop éloignée, le branchement à cette adresse peut être effectuée par le code suivant.

```
        LD R7,labelp
        JMP R7
labelp: .FILL label
        ...
label:  ...
```

2. L'instruction `JMP` est indispensable pour tout les langages de programmation qui manipulent explicitement comme C et C++ ou implicitement des pointeurs sur des fonctions. Tous les langages de programmation objet manipulent des pointeurs sur des fonctions dans la mesure où chaque objet a une table de ses méthodes.
3. L'instruction `JMP` permet aussi les retours de sous-routines puisque l'instruction `RET` est en fait une abréviation pour l'instruction `JMP R7`.



Branchement par registre

10.4.2 Récapitulatif des Instructions

La table suivante donne une description concise de chacune des instructions. Cette description comprend la syntaxe, l'action ainsi que le codage de l'instruction. La colonne *nzp* indique par une étoile * si les indicateurs n, z et p sont affectés par l'instruction.

Syntaxe	Action	nzp	Codage															
			Op-code				Arguments											
			15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
NOT DR,SR	DR ← not SR	*	1	0	0	1	DR			SR			1	1	1	1	1	1
ADD DR,SR1,SR2	DR ← SR1 + SR2	*	0	0	0	1	DR			SR1			0	0	0	SR2		
ADD DR,SR1,Imm5	DR ← SR1 + SEXT(Imm5)	*	0	0	0	1	DR			SR1			1	Imm5				
AND DR,SR1,SR2	DR ← SR1 and SR2	*	0	1	0	1	DR			SR1			0	0	0	SR2		
AND DR,SR1,Imm5	DR ← SR1 and SEXT(Imm5)	*	0	1	0	1	DR			SR1			1	Imm5				
LEA DR,label	DR ← PC + SEXT(PCoffset9)	*	1	1	1	0	DR			PCoffset9								
LD DR,label	DR ← mem[PC + SEXT(PCoffset9)]	*	0	0	1	0	DR			PCoffset9								
ST SR,label	mem[PC + SEXT(PCoffset9)] ← SR		0	0	1	1	SR			PCoffset9								
LDR DR,BaseR,Offset6	DR ← mem[BaseR + SEXT(Offset6)]	*	0	1	1	0	DR			BaseR			Offset6					
STR SR,BaseR,Offset6	mem[BaseR + SEXT(Offset6)] ← SR		0	1	1	1	SR			BaseR			Offset6					
LDI DR,label	DR ← mem[mem[PC + SEXT(PCoffset9)]]	*	1	0	1	0	DR			PCoffset9								
STI SR,label	mem[mem[PC + SEXT(PCoffset9)]] ← SR		1	0	1	1	SR			PCoffset9								
BR[n][z][p] label	Si (cond) PC ← PC + SEXT(PCoffset9)		0	0	0	0	n		z	p	PCoffset9							
NOP	No Operation		0	0	0	0	0		0	0	0	0	0	0	0	0	0	0
JMP BaseR	PC ← BaseR		1	1	0	0	0	0	0	BaseR			0	0	0	0	0	0
RET (≡ JMP R7)	PC ← R7		1	1	0	0	0	0	0	1	1	1	0	0	0	0	0	0
JSR label	R7 ← PC; PC ← PC + SEXT(PCoffset11)		0	1	0	0	1	PCoffset11										
JSRR BaseR	R7 ← PC; PC ← BaseR		0	1	0	0	0	0	0	BaseR			0	0	0	0	0	0
RTI	cf. interruptions		1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
TRAP Trapvect8	R7 ← PC; PC ← mem[Trapvect8]		1	1	1	1	0	0	0	0	Trapvect8							
Réservé			1	1	0	1												

10.5 Codage des instructions

Chaque instruction est représentée en mémoire par un code. Il est important que le codage des instructions soit réfléchi et régulier pour simplifier les circuits de décodage.

Toutes les instructions du LC-3 sont codées sur un mot de 16 bits. Les quatre premiers bits contiennent l'*op-code* qui détermine l'instruction. Les 12 bits restant codent les paramètres de l'instruction qui peuvent être des numéros de registres ou des constantes.

À titre d'exemple, les codages des deux instructions ADD et BR sont détaillés ci-dessous.

10.5.1 Codage de ADD

L'instruction ADD peut prendre deux formes. Une première forme est ADD DR,SR1,SR2 où DR, SR1 et SR2 sont les trois registres destination, source 1 et source 2. Une seconde forme est ADD DR,SR1,Imm5 où DR et SR1 sont deux registres destination et source et Imm5 est une constante signée sur 5 bits.

Le codage de l'instruction ADD est le suivant. Les quatre premiers bits de numéros 15 à 12 contiennent l'op-code 0001 de l'instruction ADD. Les trois bits suivants de numéros 11 à 9 contiennent le numéro du registre destination DR. Les trois bits suivants de numéros 8 à 6 contiennent le numéro du registre source SR1. Les six derniers bits contiennent le codage du troisième paramètre SR2 ou Imm5. Le premier de ces six bits de numéro 5 permet de distinguer entre les deux formes de l'instruction. Il vaut 0 pour la première forme et 1 pour la seconde forme. Dans la première forme, les bits de numéros 2 à 0 contiennent le numéro du registre SR2 et les bits de numéros 4 et 3 sont inutilisés et forcés à 0. Dans la seconde forme, les cinq derniers bits de numéro 4 à 0 contiennent la constante Imm5.

Le tableau ci-dessous donne un exemple de codage pour chacune des formes de l'instruction ADD.

Instruction	**Codage**															**Code en hexa**
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
	Op-code				DR			SR1			0	0	0	SR2			
ADD R2,R7,R5	0	0	0	1	0	1	0	1	1	1	0	0	0	1	0	1	0x15C5
	Op-code				DR			SR1			1	Imm5					
ADD R6,R6,-1	0	0	0	1	1	1	0	1	1	0	1	1	1	1	1	1	0x1DBF

10.5.2 Codage de BR

Le codage de l'instruction BR est le suivant. Les quatre premiers bits de numéros 15 à 12 contiennent l'op-code 0000 de l'instruction BR. Les trois bits suivants de numéros 11 à 9 déterminent respectivement si l'indicateur n, z et p est pris en compte dans la condition. Si ces trois bits sont b_{11}, b_{10} et b_9, la condition *cond* est donnée par la formule suivante.

$$cond = (b_{11} \wedge n) \vee (b_{10} \wedge z) \vee (b_9 \wedge p)$$

Les 9 derniers bits de numéros 8 à 0 codent l'offset du branchement.

Si les trois bits b_{11}, b_{10} et b_9 valent 1, la condition est toujours vérifiée puisque n + z + p = 1. Il s'agit alors de l'instruction de branchement inconditionnel BR. Si au contraire les trois bits b_{11}, b_{10} et b_9 valent 0, la condition n'est jamais vérifiée et le branchement n'est jamais pris. Il s'agit alors de l'instruction NOP qui ne fait rien. En prenant un offset égal à 0, le code de l'instruction NOP est 0x0000.

Le tableau ci-dessous donne un exemple de codage pour trois formes typiques de l'instruction BR. Dans les exemples ci-dessous, on a donné des valeurs explicites aux offsets afin d'expliquer leur codage mais cela ne correspond pas à la façon dont l'instruction BR est utilisée dans les programmes.

Instruction	Codage																Code en hexa
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
	Op-code				n	z	p	PCoffset9									
BRnp -137	0	0	0	0	1	0	1	1	0	1	1	1	0	1	1	1	0x0B77
BR 137	0	0	0	0	1	1	1	0	1	0	0	0	1	0	0	1	0x0E89
NOP	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0x0000

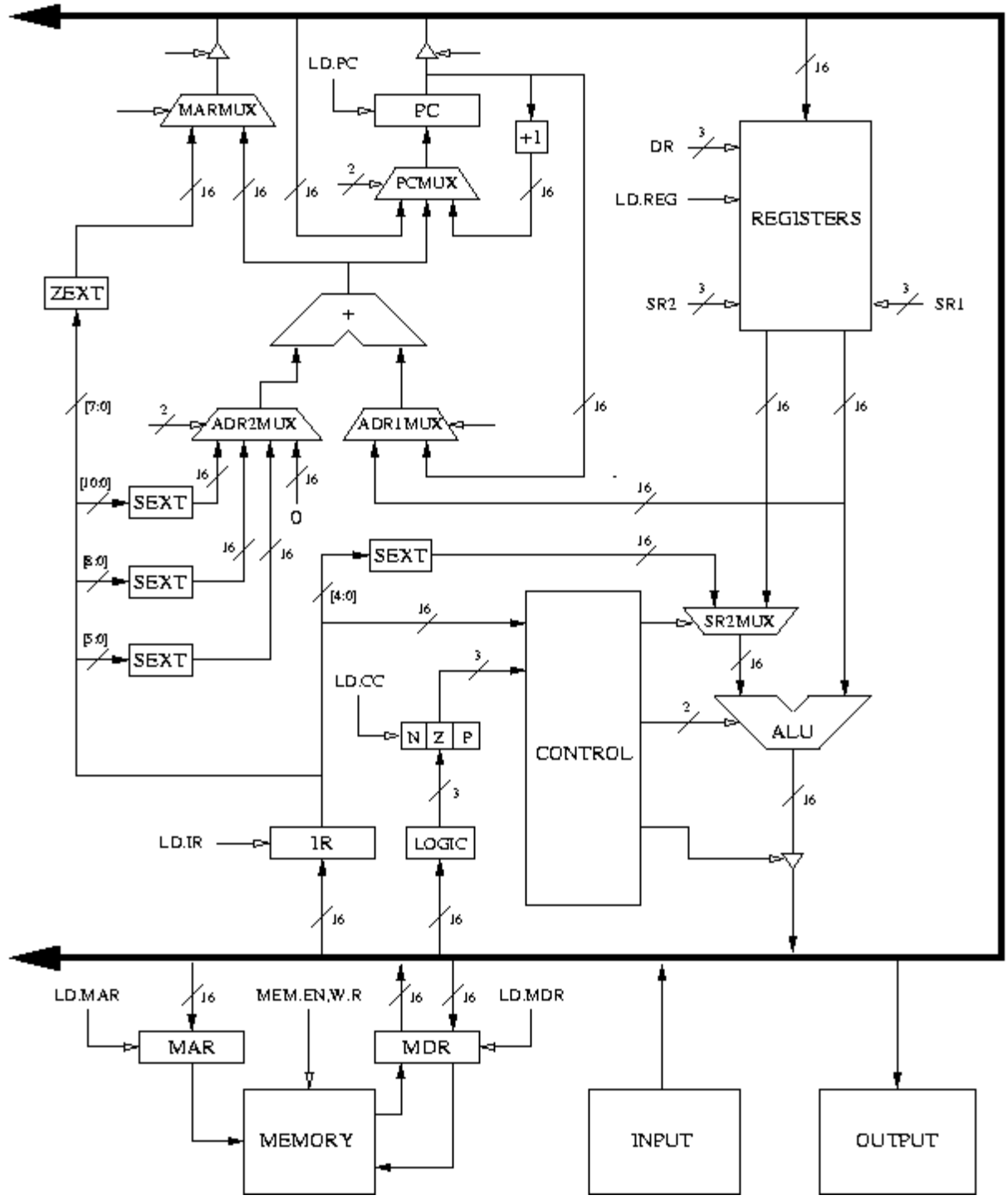
10.5.3 Répartition des op-code

On peut remarquer les *op-code* des instructions n'ont pas été affectés de manière aléatoire. On peut même observer une certaine régularité qui facilite le décodage. Les *op-code* des instructions de chargement et de rangement qui se correspondent (`LD` et `ST`, `LDR` et `STR`, …) ne différent que d'un seul bit.

MSB	LSB			
	0 0	0 1	1 0	1 1
0 0	BR	ADD	LD	ST
0 1	JSR(R)	AND	LDR	STR
1 0	RTI	NOT	LDI	STI
1 1	JMP		LEA	TRAP

10.6 Schéma interne du LC-3

Le schéma ci-dessous donne une réalisation du processeur LC-3 avec les différents éléments : registres, ALU, unité de contrôle, …. Les détails des entrées/sorties sont donnés dans un schéma ultérieur.



Chemins de données du LC-3

11 Programmation du LC-3

11.1 Programmation en assembleur

La programmation en langage machine est rapidement fastidieuse. Il est inhumain de se souvenir du codage de chacune des instructions d'un micro-processeur même simplifié à l'extrême comme le LC-3 et *a fortiori* encore plus pour un micro-processeur réel. Le *langage d'assemblage* ou *assembleur* par abus de langage rend la tâche beaucoup plus facile même si cette programmation reste longue et technique.

L'assembleur est un programme qui autorise le programmeur à écrire les instructions du micro-processeur sous forme symbolique. Il se charge de traduire cette écriture symbolique en codage hexadécimal ou binaire. Il est beaucoup plus aisé d'écrire `ADD R6,R1,R2` plutôt que 0x1C42 en hexadécimal ou encore 0001 1100 0100 0010 en binaire.

L'assembleur permet aussi l'utilisation d'*étiquettes symboliques* pour désigner certaines adresses. L'étiquette est associée à une adresse et peut alors être utilisée dans d'autres instructions. On peut par exemple écrire un morceau de code comme ci-dessous.

```
        ...
label:  ADD R6,R1,R2
        ...
        BRz label
        ...
```

L'étiquette *label* est alors associée à l'adresse ou se trouvera l'instruction `ADD R6,R1,R2` lors de l'exécution du programme. Cette adresse est alors utilisée pour calculer le codage de l'instruction `BRz label`. Comme le codage de l'instruction `BR` contient un offset, l'assembleur calcule la différence entre les adresses des deux instructions `ADD R6,R1,R2` et `BRz label` et place cette valeur dans les 9 bits de poids faible du codage de `BRz`.

11.1.1 Syntaxe

La syntaxe des programmes en assembleur est la même pour tous les assembleurs à quelques détails près.

Les instructions d'un programme en assembleur sont mises dans un fichier dont le nom se termine par l'extension `.asm`. Chaque ligne du fichier contient au plus une instruction du programme. Chaque ligne est en fait divisée en deux champs qui peuvent chacun ne rien contenir. Le premier champ contient une étiquette et le second champ une instruction. L'instruction est formée de son nom symbolique appelé *mnémonique* et d'éventuel paramètres.

Les espaces sont uniquement utiles comme séparateurs. Plusieurs espaces sont équivalents à un seul. Les programmes sont indentés de telle manière que les étiquettes se trouvent dans une première colonne et les instructions dans une seconde colonne.

L'instruction peut être remplacée dans une ligne par une directive d'assemblage. Les directives d'assemblage permettent de donner des ordres à l'assembleur. Elles sont l'équivalent des directives commençant par `'#'` comme `#include` ou `#if` du langage C.

11.1.1.1 Constantes

Il existe deux types de constantes : les constantes entières et les chaînes de caractères. Les chaînes de caractères peuvent uniquement apparaître après la directive `.STRINGZ`. Elles sont alors délimitées par deux caractères `'"'` comme dans `"Exemple de chaine"` et sont implicitement terminées par le caractère nul `'\0'`.

Les constantes entières peuvent apparaître comme paramètre des instructions du LC3 et des directives `.ORIG`, `.FILL` et `.BLKW`. Elles peuvent être données soit sous la forme d'un entier écrit dans la base 10 ou 16, soit sous la forme d'un caractère. Les constantes écrite en base 16 sont préfixées par le caractère `x`. Lorsque la constante est donnée sous forme d'un caractère, sa valeur est le code ASCII du caractère. Le caractère est alors entouré de caractères `'''` comme dans `'Z'`.

```
; Exemple de constantes
        .ORIG x3000     ; Constante entière en base 16
        AND R2,R1,2     ; Constante entière en base 10
        ADD R6,R5,-1    ; Constante entière négative en base 10
        .FILL 'Z'       ; Constante sous forme d'un caractère
        .STRINGZ "Chaine"  ; Constante chaîne de caractères
        .END
```

11.1.1.2 Commentaires

Les commentaires sont introduits par le caractère point-virgule `';'` et il s'étendent jusqu'à la fin de la ligne. Ils sont mis après les deux champs d'étiquette et d'instruction mais comme ces deux champs peuvent être vides, les commentaires peuvent commencer dès le début de la ligne ou après l'étiquette. Voici quelques exemples de commentaires

```
; Commentaire dès le début de la ligne
label:  ADD R6,R1,R2    ; Commentaire après l'instruction

loop:                   ; Commentaire après l'étiquette (avec indentation)
        BRz label
```

11.1.2 Directives d'assemblage

`.ORIG` ⟨`adresse`⟩
: Cette directive est suivi d'une adresse constante. Elle spécifie l'adresse à laquelle doit commencer le bloc d'instructions qui suit. Tout bloc d'instructions doit donc commencer par une directive `.ORIG`.

`.END`
: Cette directive termine un bloc d'instructions.

`.FILL` ⟨`valeur`⟩
: Cette directive réserve un mot de 16 bits et le remplit avec la valeur constante donnée en paramètre.

`.STRINGZ` ⟨`chaine`⟩
: Cette directive réserve un nombre de mots de 16 bits égal à la longueur de la chaîne de caractères terminée par un caractère nul et y place la chaîne. Chaque caractère de la chaîne occupe un mot de 16 bits.

`.BLKW` ⟨`nombre`⟩
: Cette directive réserve le nombre de mots de 16 bits passé en paramètre.

11.1.3 Étiquettes (labels)

Les étiquettes sont des identificateurs formés de caractères alphanumériques et commençant par une lettre. Elles désignent une adresse qui peut alors être utilisée dans les instructions.

L'adresse est spécifiée en mettant l'étiquette suivie du caractère `':'` dans la première colonne d'une ligne du programme. L'étiquette est alors affectée de l'adresse à laquelle se trouve l'instruction. L'étiquette peut également être mise avant une directive d'assemblage comme .BLKW, `.FILL` ou `.STRINGZ`. Elle désigne alors l'adresse à laquelle sont placés les mots mémoires produits par la directive.

```
; Exemple d'étiquettes
label:  ADD R0,R1,R2    ; 'label' désigne l'adresse de l'instruction ADD
char:   .FILL 'Z'       ; 'char'  désigne l'adresse du caractère 'Z'
string: .STRINGZ "Chaine"  ; 'string' désigne l'adresse du premier caractère 'C'
```

11.2 Exemples de programmes

On étudie dans cette partie quelques exemples de programmes (très) simples afin d'illustrer quelques techniques classiques de programmation en langage machine. Les codes complets de ces programmes ainsi que d'autres sont disponibles ci-dessous

- Longueur d'une chaîne de caractères
- Tours de hanoï
- Multiplication non signée
- Multiplication signée
- Multiplication logarithmique
- Calcul du logarithme en base 2

Les programmes ci-dessous sont présentés comme des fragments de programmes puisque les routines n'ont pas encore été présentées. Il faudrait les écrire sous forme de routines pour une réelle utilisation.

11.2.1 Longueur d'une chaîne

On commence par rappeler le programme en C pour calculer la longueur d'une chaîne de caractères terminée par `'\0'`.

```
int strlen(char* p) {
  int c = 0;
  while (*p != '\0') {
    p++;
    c++;
  }
  return c;
}
```

Le programme ci-dessous calcule la longueur d'une chaîne de caractères terminée par le caractère nul `'\0'`. Le registre R0 est utilisé comme pointeur pour parcourir la chaîne alors que le registre R1 est utilisé comme compteur.

```
; @param R0 adresse de la chaîne
; @return R1 longueur de la chaîne
        .ORIG x3000
        LEA R0,chaine   ; Chargement dans RO de l'adresse de la chaîne
        AND R1,R1,0     ; Mise à 0 du compteur : c = 0
loop:   LDR R2,R0,0     ; Chargement dans R2 du caractère pointé par R0
        BRz fini        ; Test de fin de chaîne
        ADD R0,R0,1     ; Incrémentation du pointeur : p++
        ADD R1,R1,1     ; Incrémentation du compteur : c++
        BR loop
fini:   NOP
chaine: .STRINGZ "Hello World"
        .END
```

Le programme C pour calculer la longueur d'une chaîne de caractères peut être écrit de manière différente en utilisant l'arithmétique sur les pointeurs.

```
int strlen(char* p) {
  char* q = p;
  while (*p != '\0')
    p++;
  return p-q;
}
```

Ce programme C se transpose également en langage d'assembleur. Le registre R0 est encore utilisé comme pointeur pour parcourir la chaîne et le registre R1 sauvegarde la valeur initiale de R0. Il contient en fait l'opposé de la valeur initiale de R0 afin de calculer la différence. Ce programme a le léger avantage d'être plus rapide que le précédent car la boucle principale contient une instruction de moins.

```
; @param R0 adresse de la chaîne
; @return R0 longueur de la chaîne
        .ORIG x3000
        LEA R0,chaine   ; Chargement dans R0 de l'adresse de la chaîne
        NOT R1,R0       ; R1 = -R0
        ADD R1,R1,1
loop:   LDR R2,R0,0     ; Chargement dans R2 du caractère pointé par R0
        BRz fini        ; Test de fin de chaîne
        ADD R0,R0,1     ; Incrémentation du pointeur : p++
        BR loop
fini:   ADD R0,R0,R1    ; Calcul de la différence q-p
chaine: .STRINGZ "Hello World"
        .END
```

11.2.2 Nombre d'occurrences d'un caractère dans une chaîne

Le programme ci-dessous calcule le nombre d'occurrences d'un caractère donné dans une chaîne de caractères terminée par le caractère nul `'\0'`. Le registre R0 est utilisé comme pointeur pour parcourir la chaîne. Le registre R1 contient le caractère recherché et le registre R2 est utilisé comme compteur. La comparaison entre chaque caractère de la chaîne et le caractère recherché est effectuée en calculant la différence de leurs codes et en testant si celle-ci est nulle. Dans ce but, le programme commence par calculer l'opposé du code du caractère recherché afin de calculer chaque différence par une addition.

```
; @param R0 adresse de la chaîne
; @param R1 caractère
; @return R2 nombre d'occurrences du caractère
        .ORIG x3000
        LEA R0,chaine   ; Chargement dans RO de l'adresse de la chaîne
        LD R1,caract    ; Chargement dans R1 du code ASCII de 'l'
        AND R2,R2,0     ; Mise à 0 du compteur
        NOT R1,R1       ; Calcul de l'opposé de R1
        ADD R1,R1,1     ; R1 = -R1
loop:   LDR R3,R0,0     ; Chargement dans R3 du caractère pointé par R0
        BRz fini        ; Test de fin de chaîne
        ADD R3,R3,R1    ; Comparaison avec 'l'
        BRnp suite      ; Non égalité
        ADD R2,R2,1     ; Incrémentation du compteur
suite:  ADD R0,R0,1     ; Incrémentation du pointeur
        BR loop
fini:   NOP
chaine: .STRINGZ "Hello World"
caract: .FILL 'l'
        .END
```

11.2.3 Multiplication

Tous les micro-processeurs actuels possèdent une multiplication câblée pour les entiers et pour les nombres flottants. Comme les premiers micro-processeurs, le LC-3 n'a pas de telle instruction. La multiplication doit donc être réalisée par programme. Les programmes ci-dessous donnent quelques implémentation de la multiplication pour les entiers signés et non signés.

11.2.3.1 Multiplication naïve non signée

On commence par une implémentation très naïve de la multiplication qui est effectuée par additions successives. On commence par donner une implémentation sous forme de programme C puis on donne ensuite une traduction en langage d'assembleur.

```
// Programme sous forme d'une fonction en C
int mult(int x, int n) {
  int r = 0;            // Résultat
  while(n != 0) {
    r += x;
    n--;
  }
  return r;
}
```

Dans le programme ci-dessous, les registres R0 et R1 contiennent respectivement x et n. Le registre R2 contient les valeurs successives de r.

```
; @param R0 x
; @param R1 n
; @return R2 x × n
        .ORIG x3000
        AND R2,R2,0     ; r = 0
        AND R1,R1,R1    ; Mise à jour de l'indicateur z pour le test
        BRz fini
loop:   ADD R2,R2,R0    ; r += x
        ADD R1,R1,-1    ; n--
        BRnp loop       ; Boucle si n != 0
fini:   NOP
        .END
```

11.2.3.2 Multiplication naïve signée

Le programme précédent fonctionne encore si la valeur de x est négative. Par contre, il ne fonctionne plus si la valeur de n est négative. Les décrémentations successives de n ne conduisent pas à 0 en le nombre d'étapes voulues. Le programme suivant résout ce problème en inversant les signes de x et n si n est négatif. Ensuite, il procède de manière identique au programme précédent.

```
; @param R0 x
; @param R1 n
; @return R2 x × n
        .ORIG x3000
        AND R2,R2,0     ; r = 0
        AND R1,R1,R1    ; Mise à jour de l'indicateur z pour le test
        BRz fini
        BRp loop
        ; Changement de signe de x et n
        NOT R0,R0       ; x = -x
        ADD R0,R0,1
        NOT R1,R1       ; n = -n
        ADD R1,R1,1
loop:   ADD R2,R2,R0    ; r += x
        ADD R1,R1,-1    ; n--
        BRnp loop       ; Boucle si n != 0
fini:   NOP
        .END
```

11.2.3.3 Multiplication logarithmique non signée

Le problème majeur des deux programmes précédents est leur temps d'exécution. En effet, le nombre d'itérations de la boucle est égal à la valeur de R1 et le temps est donc proportionnel à cette valeur. Le programme ci-dessous donne une meilleure implémentation de la multiplication de nombres non signés. Ce programme pourrait être étendu aux nombres signés de la même façon que pour l'implémentation naïve. Son temps d'exécution est proportionnel au nombre de bits des registres.

On commence par donner un algorithme récursif en C pour calculer une multiplication qui est inspiré de l'exponentiation logarithmique.

```
// Programme sous forme d'une fonction récursive en C
// Calcul de x * n de façon logarithmique
float mult(float x, int n) {
  if (n == 0)
    return 0;
  // Calcul anticipé de la valeur pour n/2 afin
  // d'avoir un seul appel récursif.
  float y = mult(x, n/2);
```

```
  if (n%2 == 0)
    return y + y;
  else
    return y + y + x;
}
```

Pour éviter d'écrire un programme récursif, on exprime le problème d'une autre façon. On suppose que l'écriture binaire du contenu de R1 est $b_{k-1} \ldots b_0$ où k est le nombre de bits de chaque registre, c'est-à-dire 16 pour le LC-3. On a alors la formule suivante qui exprime le produit x × n avec uniquement des multiplications par 2 et des additions. Cette formule est très semblable au schéma de Horner pour évaluer un polynôme.

$$x \times n = ((\ldots((xb_{k-1}2 + xb_{k-2})2 + xb_{k-3})2 + \ldots)2 + xb_1)2 + xb_0.$$

On note $x_1, \ldots, x_k$ les résultats partiels obtenus en évaluant la formule ci-dessus de la gauche vers la droite. On pose $x_0 = 0$ et pour $1 \leq j \leq k$, le nombre x_j est donné par la formule suivante.

$$x_j = ((\ldots((xb_{k-1}2 + xb_{k-2})2 + xb_{k-3})2 + \ldots)2 + xb_{k-j+1})2 + xb_{k-j}.$$

Le nombre x_k est égal au produit x × n. Il est de plus facile de calculer x_j en connaissant x_{j-1} et b_{k-j}. On a en effet les relations suivantes.

$x_j = 2x_{j-1}$ si $b_{k-j} = 0$
$x_j = 2x_{j-1} + x$ si $b_{k-j} = 1$

Le programme suivant calcule le produit x × n en partant de $x_0 = 0$ puis en calculant de proche en proche les nombres x_j grâce aux formules ci-dessus. Les registres R0 et R1 contiennent respectivement x et n. Le registre R2 contient les valeurs successives $x_0, \ldots, x_k$ et le registre R3 est utilisé comme compteur. Pour accéder aux différents bits $b_{k-1}, \ldots, b_0$ de n, on utilise le procédé suivant. Le signe de n vu comme un entier signé donne la valeur de b_{k-1}. Ensuite, l'entier n est décalé vers la gauche d'une position à chaque itération. Le signe des contenus successifs de R2 donne les valeurs $b_{k-2}, \ldots, b_0$.

```
; @param R0 x
; @param R1 n
; @return R2 x × n
        .ORIG x3000
        AND R2,R2,0     ; Initialisation x0 = 0
        LD R3,cst16     ; Initialisation du compteur
        AND R1,R1,R1
        BRzp bit0
bit1:   ADD R2,R2,R0    ; Addition de x si bk-j = 1
bit0:   ADD R3,R3,-1    ; Décrémentation du compteur
        BRz fini
        ADD R2,R2,R2    ; Calcul de 2xj-1
        ADD R1,R1,R1    ; Décalage de n vers la gauche
        BRn bit1
        BR bit0
fini:   NOP
cst16:  .FILL 16
        .END
```

11.2.4 Addition 32 bits

Dans la majorité des micro-processeurs réels, il y a outre les flags n, z et p un flag c qui reçoit la retenue (Carry) lors d'une addition par une instruction ADD. Cela permet de récupérer facilement cette retenue en testant le flag c. Certains micro-processeurs possèdent aussi une instruction ADC qui ajoute au résultat de l'addition la valeur du flag c. Ceci permet d'enchaîner facilement des additions pour faire des calculs sur des entiers codés sur plusieurs mots. Le micro-processeur LC-3 n'a pas de telles facilités et il faut donc exploiter au mieux ses possibilités.

```
; @param R1,R0 poids fort et poids faible du premier argument
; @param R3,R2 poids fort et poids faible du second argument
; @return R5,R4 poids fort et poids faible du résultat
        .ORIG x3000
        ; Addition des poids faibles
        ADD R4,R2,R0
        ; Addition des poids forts
        ADD R5,R3,R2
        ; Test s'il y a une retenue sur les poids faibles
        AND R0,R0,R0
        BRn bit1
        ; Cas où le bit de poids fort de R0 est 0
        AND R2,R2,R2
        ; Si les deux bits de poids fort de R1 et R0 sont 0,
        ; il n'y a pas de retenue.
        BRzp fini
        ; Si un seul des deux bits de poids fort de R1 et R0 est 1,
        ; il faut tester le bit de poids fort de R4 pour savoir
        ; s'il y a eu une retenue.
testR4: AND R4,R4,R4
        BRzp carry
        BR fini
        ; Cas où le bit de poids fort de R0 est 1
bit1:   AND R2,R2,R2
        ; Si le bit de poids fort de R1 est 0, on est ramené au cas
        ; un seul des deux bits de poids fort de R1 et R0 est 1.
        BRzp testR4
        ; Si les deux bits de poids fort de R1 et R0 sont 1,
        ; il y a nécessairement une retenue.
        ; Correction de la retenue
carry:  ADD R5,R5,1
fini:   NOP
        .END
```

11.2.5 Conversion en hexa du contenu d'un registre

```
; @param R0 valeur à convertir
        ADD R1,R0,0             ; R1 <- R0
        ld R2,cst4              ; Nombre de caractères
loopo:  ld R3,cst4              ; Nombre de bits par caractère
        ; Rotation de 4 bits
loopi:  AND R1,R1,R1
        BRn $1
        ADD R1,R1,R1
        BR $2
$1:     ADD R1,R1,R1
        ADD R1,R1,1
$2:     ADD R3,R3,-1
        BRp loopi
        ; Recupération de 4 bits
```

```
        AND R0,R1,0xf
        ADD R0,R0,-0xa
        BRn $3
        ; Chiffres de 0 à 9
        ld R3,cst40             ; 0x40 = '0' + 0xa
        BR $4
        ; Chiffres de A à F
$3:     ld R3,cst51             ; 0x51 = 'A' + 0xa
$4:     ADD R0,R0,R3
        TRAP 0x21               ; Affichage
        ADD R2,R2,-1
        BRp loopo
        NOP
        TRAP 0x25               ; Arrêt
cst4:   .fill 4
cst40:  .fill 0x40
cst51:  .fill 0x51
```

12 Les sous-routines et la pile

12.1 Instruction JMP

L'instruction JMP permet de transférer l'exécution du programme à une adresse contenue dans un des 8 registres R0,…,R7. Ce genre d'instruction est nécessaire à tout programme qui gère dynamiquement les appels de fonction. C'est le cas de tout programme en C qui manipule des pointeurs sur des fonctions. C'est aussi le cas des langages objets comme C++ dont les objets contiennent une table des méthodes sous forme de pointeurs de fonctions. Dans le cas du LC-3, cette instruction permet de compenser la faiblesse de l'instruction BR qui peut uniquement sauter à une adresse éloignée d'au plus 256 mots (l'offset est codé sur 9 bits en complément à 2). L'instruction JMP est aussi indispensable pour réaliser les retours de sous-routine comme nous allons le voir.

La réalisation d'une boucle se fait par exemple de la manière suivante.

```
        LD R0,nbiter    ; Nombre d'itérations de la boucle
; Boucle
loop:   ...             ; Début de la boucle
        ...
        ADD R0,R0,-1    ; Décrémentation du compteur
        BRp loop        ; Retour au début de la boucle
; Suite du programme
        ...
```

12.2 Sous-routines

12.2.1 Appel

Lorsqu'un programme appelle une sous-routine, il faut d'une certaine manière mémoriser l'adresse à laquelle doit revenir s'exécuter le programme à la fin de la routine. Dans le micro-processeur LC-3, l'adresse de retour, c'est-à-dire l'adresse de l'instruction suivante est mise dans le registre R7 lors d'un appel de sous-routine. Ce registre a donc un rôle particulier.

Il existe deux instructions JSR et JSRR permettant d'appeler une sous-routine. L'instruction JSR est semblable à l'instruction BR de branchement inconditionnel. Il n'y a pas de variante conditionnelle de l'instruction JSR. Les trois bits utilisés pour la condition dans le codage l'instruction BR sont récupérés pour avoir un offset de 11 bits au lieu de 9 bits pour BR. Un bit est aussi nécessaire pour distinguer JSR et JSRR qui utilisent le même op-code (cf. codage des instructions du LC-3). L'adresse de la sous-routine est stockée dans le code l'instruction sous forme d'un offset de 11 bits. L'instruction JSRR est semblable à l'instruction JMP. Elle permet d'appeler une sous-routine dont l'adresse est contenue dans un des 8 registres. Cette deux instructions ont en commun de transférer la valeur du compteur de programme PC incrémenté (adresse de l'instruction suivante) dans le registre R7 avant de charger PC avec l'adresse de la sous-routine.

12.2.2 Retour

Lors d'un appel à une sous-routine, l'adresse de retour est mise dans le registre R7. La sous-routine se termine donc par un saut à cette adresse avec l'instruction JMP R7. Cette instruction peut aussi être désignée par le mnémonique RET.

```
; Appel de la sous-routine sub
        ...
        JSR sub         ; Adresse de l'instruction suivante dans R7
        ...

; Sous-routine sub
sub:    ...
        ...
        RET             ; JMP R7
```

12.2.3 Exemple

On reprend le fragment de programme pour calculer la longueur d'une chaîne sous la forme d'une sous-routine appelée par un programme principal.

```
        .ORIG x3000
; Programme principal
        ...
        ; Premier appel
        LEA R0,chaine   ; Chargement dans R0 de l'adresse de la chaîne
        JSR strlen      ; Appel de la sous-routine
        ...
        ; Autre appel
        JSR strlen
        ...

; Chaine constante
chaine: .STRINGZ "Hello World"
        .END

; Sous-routine pour calculer la longueur d'une chaîne terminée par '\0'
; @param R0 adresse de la chaîne
; @return R0 longueur de la chaîne
; L'adresse de retour est dans R7
strlen: AND R1,R1,0     ; Mise à 0 du compteur : c = 0
loop:   LDR R2,R0,0     ; Chargement dans R2 du caractère pointé par R0
        BRz fini        ; Test de fin de chaîne
        ADD R0,R0,1     ; Incrémentation du pointeur : p++
        ADD R1,R1,1     ; Incrémentation du compteur : c++
        BR loop
fini:   ADD R0,R1,0     ; R0 = R1
        RET             ; Retour par JMP R7
```

La routine `strlen` utilise le registre R1 pour effectuer ses calculs. Si le programme principal utilise également ce registre R1, le contenu de ce registre doit être sauvegardé pendant l'appel à `strlen`. Cette question est abordée ci-dessous.

12.3 Sauvegarde des registres

Les registres du micro-processeur LC-3 sont en nombre limité. Lorsque qu'une routine a besoin de registres pour faire des calculs intermédiaires, il est préférable de sauvegarder le contenu de ces registres. Cela évite de détruire le contenu du registre qui est peut-être utilisé par le programme appelant la routine. En particulier si une routine doit appeler une autre sous-routine, elle doit préserver le registre R7 qui contient l'adresse de retour.

12.3.1 Registres

Une routine `sub` qui appelle une autre routine `subsub` peut utiliser un registre, par exemple R5 pour sauvegarder le registre R7 qui contient l'adresse de retour de `sub`. Cette méthode donne un code efficace car il n'utilise que les registres. Par contre, on arrive rapidement à cours de registres. Il faut donc utiliser d'autres techniques.

```
sub:    ADD R5,R7,0     ; Sauvegarde de R7 dans R5
        ...
        JSR subsub
        ...
        ADD R7,R5,0     ; Restauration de R7
        RET

subsub: ...

        RET
```

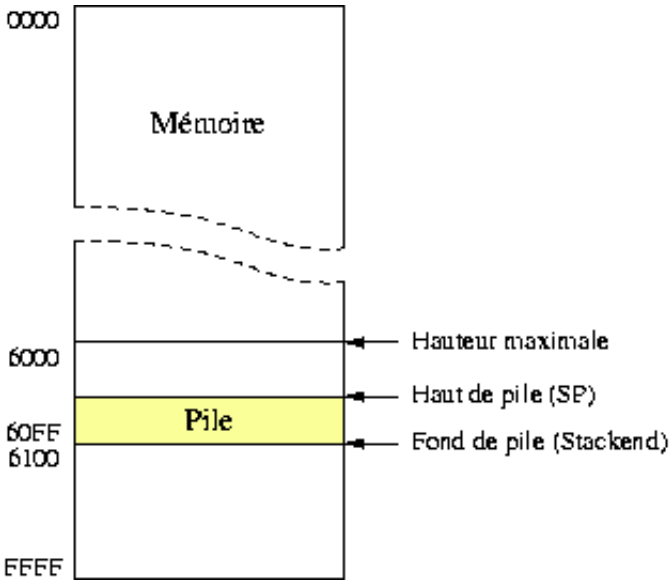
12.3.2 Emplacements mémoire réservés

Une autre méthode consiste à réserver des emplacements mémoire pour la sauvegarde des registres pendant l'exécution d'une routine. Les contenus des registres sont rangés dans ces emplacements au début de la routine et ils sont restaurés à la fin de celle-ci. Cette méthode a deux inconvénients majeurs. D'une part, elle nécessite de réserver de la mémoire pour chaque routine. Cela peut gaspiller beaucoup de mémoire si le programme est conséquent. De plus, l'espace réservé pour les routines non appelées est perdu. D'autre part, cette méthode conduit à du code qui n'est pas réentrant.

```
; Sous-routine sub sauvegardant R0 et R1
sub:    ST R0,saveR0    ; Sauvegarde de R0
        ST R1,saveR1    ; Sauvegarde de R1

        ...             ; Corps de la procédure

        LD R1,saveR1    ; Restauration de R1
        LD R0,saveR0    ; Restauration de R0
        RET
saveR0: .BLKW 1         ; Emplacement de sauvegarde de R0
saveR1: .BLKW 1         ; Emplacement de sauvegarde de R1

; Appel de la sous-routine
; Le registre R7 est détruit
        JSR sub
```

12.3.3 Utilisation d'une pile

La meilleure méthode est d'utiliser une pile. Cela s'apparente à la méthode des emplacements réservés mais dans le cas d'une pile, c'est un espace global qui est partagé par toutes les sous-routines. Par contre, cette méthode nécessite de réserver un des registres à la gestion de la pile. Dans le cas du micro-processeur LC-3, c'est le registre R6 qui est généralement utilisé.



Pile en mémoire

12.3.3.1 Utilisation du registre R6

Le registre R6 est utilisé comme pointeur de pile. Pendant toute l'exécution du programme, le registre R6 donne l'adresse de la dernière valeur mise sur la pile. La pile croît vers les adresses décroissantes. N'importe quel autre registre (hormis R7) aurait pu être utilisé mais il est préférable d'utiliser R6 en cas d'interruption.

12.3.3.2 Empilement d'un registre

Un empilement est réalisé en déplaçant vers les adresses basses le pointeur de pile pour le faire pointer sur le premier emplacement libre. Ensuite le contenu du registre y est rangé en employant un rangement avec adressage relatif avec offset de 0.

```
ADD R6,R6,-1    ; Déplacement du haut de pile
STR Ri,R6,0     ; Rangement de la valeur
```

12.3.3.3 Dépilement d'un registre

Le dépilement est bien sûr l'opération inverse. Le contenu du registre est récupéré avec un chargement avec adressage relatif. Ensuite le pointeur de pile est incrémenté pour le faire pointer sur le haut de la pile.

```
LDR Ri,R6,0     ; Récupération de la valeur
ADD R6,R6,1     ; Restauration du haut de pile
```

Aucune vérification n'est faite pour savoir si la pile déborde. Il appartient au programmeur de vérifier que dans chaque routine, il effectue autant de dépilements que d'empilements.

12.3.3.4 Empilement ou dépilement de plusieurs registres

Lors de l'empilement ou du dépilement de plusieurs registres, l'offset de l'adressage relatif permet de faire en une seule instruction les différentes incrémentations ou décrémentations du registre de pile. L'empilement des registres R0, R1 et R2 (dans cet ordre peut par exemple être réalisé de la manière suivante.

```
ADD R6,R6,-3    ; Déplacement du haut de pile
STR R0,R6,2     ; Empilement de R0
STR R1,R6,1     ; Empilement de R1
STR R2,R6,0     ; Empilement de R2
```

Le fait de placer la décrémentation du registre de pile R6 avant de placer les contenus des registres R0, R1 et R2 sur la pile n'est pas anodin. Il semble à première vue que l'instruction `ADD R6,R6,-3` pourrait être placée après les trois instructions `STR` en changeant bien sûr les offsets utilisés par ces trois dernières par les valeurs -1, -2 et -3. Ceci est en fait faux. Si une interruption survient entre le placement des contenus des registres sur la pile et la décrémentation de R6, le contenu de la pile peut être altéré par de valeurs que l'interruption mettrait sur la pile. Le fait de décrémenter d'abord R6 peut être considéré comme une façon de réserver sur la pile les emplacements pour les contenus de R0, R1 et R2. Dans le cas du LC-3, les programmes utilisateur sont en quelque sorte protégés de ce type de problèmes car les interruptions utilisent la pile système qui est distincte de la pile utilisateur. Par contre, le code exécuté en mode privilégié doit respecter cette contrainte.

Le dépilement des registres R2, R1 et R0 se fait la même manière. L'ordre des trois dépilements effectués par les instructions `LDR` n'est pas imposé. Le résultat serait le même en les mettant dans un ordre différent. Par contre, on a respecté l'ordre inverse des empilements pour bien montrer qu'il s'agit de l'utilisation d'une pile.

```
LDR R0,R6,2     ; Dépilement de R0
LDR R1,R6,1     ; Dépilement de R1
LDR R2,R6,0     ; Dépilement de R2
ADD R6,R6,3     ; Restauration du haut de pile
```

La raison pour laquelle l'incrémentation de R6 est placée après le chargement des registres avec les valeurs sur la pile est identique à la raison pour laquelle la décrémentation de R6 est placée avant le placement des contenus des registres sur la pile.

La pile permet en particulier d'échanger les contenus de deux registres sans utiliser de registre supplémentaire autre que le pointeur de pile R6. Le morceau de code suivant échange par exemple les contenus des registres R1 et R2.

```
ADD R6,R6,-2    ; Déplacement du haut de pile
STR R1,R6,1     ; Empilement de R1
STR R2,R6,0     ; Empilement de R2
LDR R2,R6,1     ; Dépilement du contenu de R1 dans R2
LDR R1,R6,0     ; Dépilement du contenu de R2 dans R1
ADD R6,R6,2     ; Restauration du haut de pile
```

Si un micro-processeur possède une opération de ou exclusif bit à bit appelée `XOR`, l'échange des contenus de deux registres peut aussi se faire de la manière compliquée suivante.

```
                ; R1 : x        R2 : y
XOR R1,R1,R2    ; R1 : x ^ y    R2 : y
XOR R2,R1,R2    ; R1 : x ^ y    R2 : x
XOR R1,R1,R2    ; R1 : y        R2 : x
```

12.3.3.5 Appels de sous-routine imbriqués

Dans le cas d'appels de sous-routine imbriqués, c'est-à-dire d'une sous-routine `sub` appelant une (autre) sous-routine `subsub`, il est nécessaire de sauvegarder sur la pile l'adresse de retour de `sub` contenue dans le registre R7. À l'appel de la seconde sous-routine `subsub`, le registre R7 reçoit l'adresse de retour de celle-ci et son contenu précédent est écrasé.

```
sub:    ADD R6,R6,-2    ; Sauvegarde sur la pile de
        STR R7,R6,1     ; - l'adresse de retour
        STR R0,R6,0     ; - registre R0
        ...
        JSR subsub
        ...
        LDR R0,R6,0     ; Restauration de la pile de
        LDR R7,R6,1     ; - registre R0
        ADD R6,R6,2     ; - l'adresse de retour
        RET             ; Retour par JMP R7

subsub: ...             ; R7 contient l'adresse de retour
                        ; c'est-à-dire l'adresse de l'instruction
                        ; suivant JSR subsub

        RET             ; Retour par JMP R7
```

12.3.3.6 Initialisation de la pile

L'initialisation de la pile se fait en mettant dans le registre R6 l'adresse du haut de la pile. Il s'agit en fait de l'adresse du premier emplacement mémoire après l'espace réservé à la pile. Comme tout utilisation de la pile commence par décrémenter le registre R6, cet emplacement n'est normalement jamais utilisé par la pile.

```
; Initialisation de la pile au début du programme
psp:   .FILL stackend
main:   LD R6,psp       ; Équivalent à LEA R6,stackend
        ...


; Réservation de l'espace pour la pile
        .ORIG 0x6000
        .BLKW 0x100     ; Taille de la pile : 256 octets
stackend:               ; Adresse de mot mémoire suivant
```

12.4 Programmation

Pour illustrer l'utilisation de la pile, un programme récursif (source) calculant la solution du problème des *tours de Hanoï* est donné ci-dessous.

```
; Tours de Hanoï
        .ORIG x3000
; Programme principal
; Le pointeur de pile R6 est initialisé par le simulateur
hanoim: LD R0,nbrdisk   ; Nombre de disques
        LD R1,startst   ; Piquet de depart
        LD R2,endst     ; Piquet d'arrivée
        JSR hanoi
        TRAP x25        ; HALT
; Constantes
nbrdisk:.FILL 3         ; Nombre de disques
startst:.FILL 1         ; Piquet de depart
endst:  .FILL 2         ; Piquet d'arrivée

; Calcul du piquet intermédiaire avant de lancer la procédure récursive
; @param R0 nombre de disques
; @param R1 piquet de départ
; @param R2 piquet d'arrivée
hanoi:
```

```
        ; Calcul du troisième piquet : R3 = 6 - R1 - R2
        ADD R3,R2,R1
        NOT R3,R3
        ADD R3,R3,7     ; 7 = 6 + 1

; Procédure récursive
; @param R0 nombre de disques
; @param R1 piquet de départ
; @param R2 piquet d'arrivée
; @param R3 piquet intermédiaire
hanoirec:
        ; Décrémentation du nombre de disques
        ADD R0,R0,-1
        ; Test si le nombre de disques est 0
        BRn hanoiend
        ; Empilement de l'adresse de retour
        ADD R6,R6,-1
        STR R7,R6,0
        JSR swapR2R3
        JSR hanoirec
        JSR swapR2R3
        JSR print
        JSR swapR1R3
        JSR hanoirec
        JSR swapR1R3
        ; Dépilement de l'adresse de retour
        LDR R7,R6,0
        ADD R6,R6,1
        ; Restauration du nombre de piquets
hanoiend:
        ADD R0,R0,1
        RET

; Échange des contenus de R1 et R3
swapR1R3:
        ADD R4,R3,0
        ADD R3,R1,0
        ADD R1,R4,0
        RET
; Échange des contenus de R2 et R3
swapR2R3:
        ADD R4,R3,0
        ADD R3,R2,0
        ADD R2,R4,0
        RET

; Affichage de R1 -> R2
print:  ; Empilement de R7 et R0
        ADD R6,R6,-2
        STR R7,R6,1
        STR R0,R6,0
        ; Affichage du caractère '0' + R1
        LD R0,char0
        ADD R0,R0,R1
        TRAP x21        ; Appel système putc
        ; Affichage de la chaîne " --> "
        LEA R0,arrow
        TRAP x22        ; Appel système puts
        ; Affichage du caractère '0' + R2
        LD R0,char0
        ADD R0,R0,R2
        TRAP x21        ; Appel système putc
```

```
        ; Retour à la ligne
        LD R0,charnl
        TRAP x21        ; Appel système putc
        ; Dépilement de R0 et R7
        LDR R0,R6,0
        LDR R7,R6,1
        ADD R6,R6,2
        RET
; Constantes pour l'affichage
char0:  .FILL '0'
charnl: .FILL '\n'
arrow:  .STRINGZ " --> "
        .END
```

12.5 Comparaison avec d'autres micro-processeurs

Beaucoup de micro-processeurs (surtout CISC comme le 80x86) possèdent un registre, généralement appelé SP, dédié à la gestion de la pile. Il y a alors des instructions spécifiques pour manipuler la pile. La pile est alors systématiquement utilisée pour les appels à des sous-routines. Certaines instructions permettent en effet les appels et les retours de sous-routines. D'autres instruction permettent d'empiler ou de dépiler un ou plusieurs registres afin de simplifier la sauvegarde des registres.

12.5.1 Instructions CALL et RET

Les instructions d'appels de sous-routine empilent alors l'adresse de retour plutôt que de la mettre dans un registre particulier. Comme l'adresse de retour d'une sous-routine se trouve toujours sur la pile, il existe une instruction spécifique, généralement appelée `RET` pour terminer les sous-routines. Cette instruction dépile l'adresse de retour puis la met dans le compteur de programme PC. Ceci explique pourquoi l'instruction `JMP R7` du LC-3 est aussi appelée `RET`.

12.5.2 Instructions PUSH et POP

Les micro-processeurs ayant un registre dédié à la pile possèdent des instructions, généralement appelées `PUSH` et `POP` permettant d'empiler et de dépiler un registre sur la pile. Ces instructions décrémentent et incrémentent automatiquement le registre de pile pour le mettre à jour. Certains micro-processeurs comme le 80x86 ont même des instructions permettant d'empiler et de dépiler tous les registres ou un certain nombre d'entre eux.

12.6 Appels système

L'instruction `TRAP` permet de faire un appel au système. Les mots mémoire des adresses 0x00 à 0xFF contiennent une table des appels système. Chaque emplacement mémoire contient l'adresse d'un appel système. Il y a donc 256 appels système possibles. L'instruction `TRAP` contient le numéro d'un appel système, c'est-à-dire l'indice d'une entrée de la table des appels systèmes.

Comme l'instruction `JSR`, l'instruction `TRAP` sauvegarde le compteur de programme PC dans le registre R7 puis charge PC avec l'entrée de la table des appels système dont le numéro est indiqué par l'instruction. Le retour d'un appel système se fait par l'instruction `RET`.

La table des appels système procure une indirection qui a l'intérêt de rendre les programmes utilisateurs indépendant du système. Dans la mesure où l'organisation (c'est-à-dire la correspondance entre les numéros de la table et les appels système) de la table reste identique, il n'est pas nécessaire de

changer (recompiler) les programmes utilisateurs.

Sur les micro-processeurs usuels, l'instruction `TRAP` fait passer le micro-processeur en mode privilégié (aussi appelé mode système). C'est d'ailleurs souvent la seule instruction permettant de passer en mode privilégié. Ceci garantit que seul le code du système d'exploitation soit exécuté en mode privilégié. Par contre, il faut une instruction capable de sortir du mode privilégié avant d'effectuer le retour au programme principal. Le processeur LC-3 ne possède pas de telle instruction.

12.7 Interruptions

Les interruptions sont un mécanisme permettant à un circuit extérieur au micro-processeur d'interrompre le programme en cours d'exécution afin de faire exécuter une routine spécifique. Les interruptions sont en particulier utilisées pour la gestion des entrées/sorties et pour la gestion des processus.

12.7.1 Initiation d'une interruption

Le micro-processeur possède une ou plusieurs broches permettant de recevoir des signaux provenant des circuits extérieurs susceptibles de provoquer des interruptions. Avant d'exécuter chaque instruction, le micro-processeur vérifie s'il y a un signal (de valeur 1) sur une de ces broches. Si aucun signal n'est présent, il continue le programme et il exécute l'instruction suivante. Si au contraire un signal est présent, il interrompt le programme en cours et il commence à exécuter une sous-routine spécifique appelée *sous-routine d'interruption*. Lorsque cette sous-routine se termine, le micro-processeur reprend l'exécution du programme en cours à l'instruction où il en était.

L'exécution d'une sous-routine d'interruption s'apparente à l'exécution d'une sous-routine du programme. Par contre elle n'est initiée par aucune instruction du programme. Elle est initiée par le signal d'interruption et elle peut intervenir à n'importe quel moment de l'exécution du programme interrompu. Pour cette raison, cette sous-routine d'interruption ne doit modifier aucun des registres R0,…,R6 et R7 du micro-processeur. De la même façon, elle ne doit pas modifier aucun des indicateurs n, z et p car l'interruption peut intervenir entre une instruction qui positionne ces indicateurs (comme `LD` et `ADD`) et une instruction de branchement conditionnel (comme `BRz`) qui les utilise.

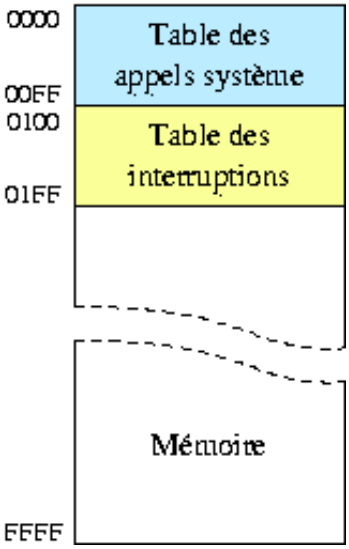
12.7.2 Retour d'une interruption

Afin de pouvoir revenir à l'instruction à exécuter, le micro-processeur doit sauvegarder le compteur de programme. Comme aucun registre ne peut être modifié, le compteur de programme ne peut être transféré dans le registre R7 comme les instructions `JSR`, `JSRR` et `TRAP` le font. Le compteur de programme est empilé sur la pile ainsi que le registre PSR qui contient les indicateurs n, z et p. Ensuite le compteur de programme est chargé avec l'adresse de la routine d'interruption. La routine d'interruption se charge elle-même d'empiler les registres dont elle a besoin afin de les restaurer lorsqu'elle se termine. La routine d'interruption se termine par une instruction spécifique `RTI` (pour *ReTurn Interrupt*). Il faut d'abord dépiler le registre PSR puis sauter à l'instruction dont l'adresse est sur la pile. Comme le registre R7 doit être préservé, l'instruction `RET` ne peut être utilisée. L'instruction `RTI` se charge de dépiler PSR et de dépiler l'adresse de retour pour la charger dans le registre PC.

12.7.3 Vecteurs d'interruption

Dans les micro-processeurs très simples, l'adresse de la routine d'interruption est fixe. S'il y a plusieurs circuits extérieurs susceptibles de provoquer une interruption, le premier travail de la routine d'interruption est de déterminer quel est le circuit qui a effectivement provoqué l'interruption. Ceci est fait en interrogeant les registres d'état (Status Register) de chacun de ces circuits.

S'il y a beaucoup de circuits pouvant provoquer une interruption, il peut être long de déterminer lequel a provoqué l'interruption. Comme le temps pour gérer chaque interruption est limité, les micro-processeurs possèdent plusieurs routines d'interruption. Chacune d'entre elles gère un des circuits extérieurs. Comme pour les appels systèmes, il existe une table des interruptions qui contient les adresses des différentes routines d'interruption. Pour le micro-processeur LC-3, celle-ci se trouve aux adresses de 0x100 à 0x1FF. Lorsqu'un circuit engendre une interruption, il transmet au micro-processeur via le bus de données un *vecteur d'interruption*. Ce vecteur est en fait l'indice d'une entrée de la table des interruptions. Pour le micro-processeur LC-3, il s'agit d'une valeur 8 bits auquel le micro-processeur ajoute 0x100 pour trouver l'entrée dans la table. Chaque circuit externe reçoit le vecteur d'interruption qui lui est attribué lors de son initialisation par le micro-processeur.



Tables des appels systèmes et interruptions

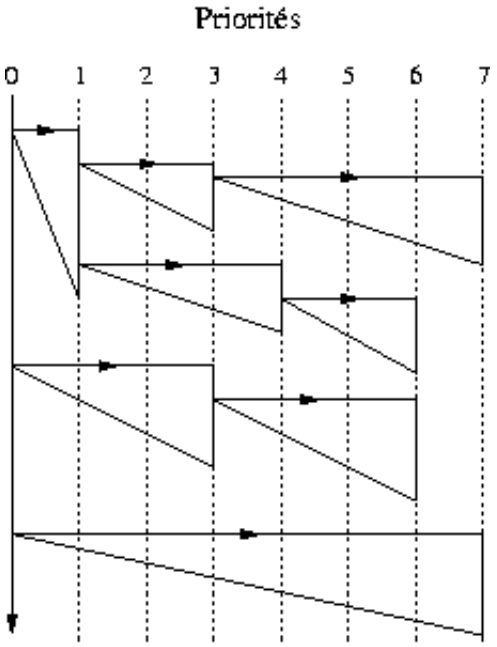
12.7.4 Priorités

Dans le micro-processeur LC-3, chaque programme est exécuté avec une certaine priorité allant de 0 à 7. Celle-ci est stockée dans 3 bits du registre PSR. Lorsqu'une interruption est demandée par un circuit extérieur, celle-ci a aussi une priorité. Elle est alors effectivement effectuée si sa priorité est supérieure à la priorité du programme en cours d'exécution. Sinon, elle est ignorée.

Lorsqu'une interruption est acceptée, la routine d'interruption est exécutée avec la priorité de l'interruption. Cette priorité est mise dans le registre PSR après la sauvegarde de celui-ci sur la pile. À la fin de la routine, l'instruction `RTI` restaure le registre PSR en le dépilant. Le programme interrompu retrouve ainsi sa priorité.

Il est possible qu'une demande d'interruption intervienne pendant l'exécution d'une autre routine d'interruption. Cette interruption est prise en compte si elle a une priorité supérieure à celle en cours. Ce mécanisme permet de gérer les urgences différentes des interruptions. Une interruption liée à un

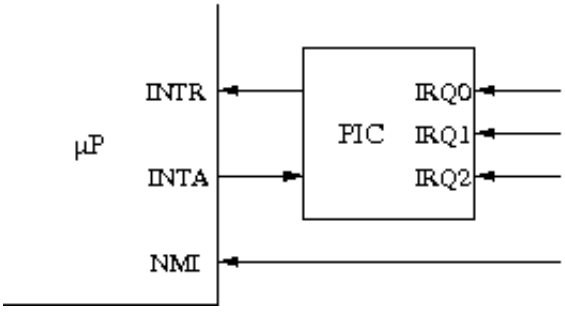
défaut de cache est nettement plus urgente qu'une autre liée à l'arrivée d'un caractère d'un clavier.



Imbrications des interruptions

12.7.5 Contrôleur d'interruption

Certains micro-processeurs comme le 8086 ne disposent que de deux interruptions. Chacune de ses deux interruptions correspond à une broche du circuit du micro-processeur. Une première interruption appelée NMI (Non Maskable Interrupt) est une interruption de priorité maximale puisqu'elle ne peut pas être ignorée par le micro-processeur. Celle-ci est généralement utilisée pour les problèmes graves comme un défaut de mémoire. Une seconde interruption appelée INTR est utilisée par tous les autres circuits extérieurs. Afin de gérer des priorités et une file d'attente des interruptions, un circuit spécialisé appelé PIC (Programmable Interrupt Controler) est adjoint au micro-processeur. Ce circuit reçoit les demandes d'interruption des circuits et les transmet au micro-processeur. Le micro-processeur dispose d'une sortie INTA (Interrupt Acknowledge) pour indiquer au PIC qu'une interruption est traitée et qu'il peut en envoyer une autre.



Contrôleur d'interruptions

12.7.6 Séparation des piles système et utilisateur

Pour plusieurs raisons, il est préférable que la pile utilisée par le système soit distincte de la pile de chacun des programmes utilisateurs. D'abord, si un programme gère mal sa pile et que celle-ci déborde, cela n'empêche pas le système de fonctionner correctement. D'autre part, si la pile est partagée, un programme peut en inspectant sa pile trouver des informations de sécurité qu'il ne devrait pas avoir.

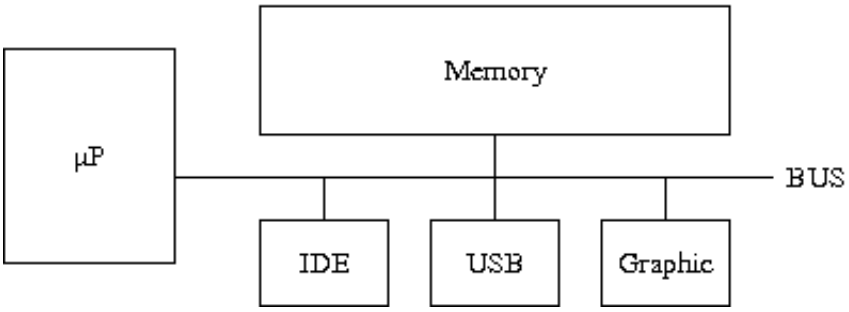
Pour ces raisons, le micro-processeur LC-3 permet d'utiliser deux piles distinctes. Ceci est mis en œuvre de la manière suivante. Le micro-processeur dispose de deux registres USP (User Stack Pointer) et SSP (System Stack Pointer) qui sauvegardent le registre R6 utilisé comme pointeur de pile. Lorsque le micro-processeur LC-3 passe en mode privilégié (lors d'une interruption), le registre R6 est sauvegardé dans USP et R6 est chargé avec le contenu de SSP. Lorsqu'il sort du mode privilégié, R6 est sauvegardé dans SSP et R6 est chargé avec le contenu de USP. Ce mécanisme impose bien sûr que le pointeur de pile soit le registre R6.

13 Entrées/Sorties

Un micro-processeur ne communique pas directement avec l'extérieur. Toutes les entrées/sorties se font par l'intermédiaire de circuits spécialisés qui déchargent ainsi le processeur de certaines tâches. Il existe des circuits spécialisés dans les différents types de connexions : série, parallèle, SCSI, IDE, USB, …. Un circuit pour une connexion série se charge par exemple de surveiller les signaux de la ligne et de le traduire en un octet disponible pour le processeur.

13.1 Adressage des circuits d'entrées/sorties

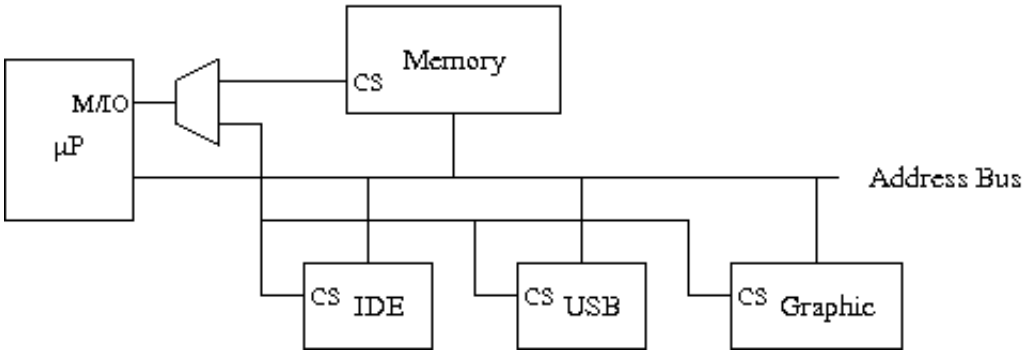
Ces différents circuits résident sur la carte mère et communiquent avec le processeur par l'intermédiaire du bus de donnée. La communication est en général à l'initiative du micro-processeur. Hormis les exceptions, ces circuits se contentent de répondre aux demandes du micro-processeur et d'exécuter ses instructions. Les circuits les plus simples ont généralement un *registre de contrôle* et un *registre de donnée* qui peuvent être lus ou chargés par le micro-processeur. Le registre de contrôle permet au micro-processeur de connaître l'état du circuit et éventuellement de le configurer. Le registre de donnée permet l'échange des données en entrée et/ou en sortie. Le micro-processeur lit dans ce registre les données en entrée et y écrit les données en sortie.



Circuits d'entrées/sorties

Afin de répondre aux demandes du micro-processeur, les circuits doivent être sélectionnés par celui-ci. Il existe deux méthodes classiques pour faire cet adressage. La première méthode était plutôt utilisée par les processeurs anciens alors que la seconde est plus fréquente sur les micro-processeurs modernes.

La première méthode consiste à séparer l'adressage de la mémoire et l'adressage des entrées/sorties. En plus du bus de d'adresse, il y a une ligne supplémentaire où le processeur indique s'il s'adresse à la mémoire ou aux circuits d'entrées/sorties. Cette méthode a l'avantage d'augmenter d'une certaine façon l'espace d'adressage. C'est pourquoi les premiers micro-processeurs l'utilisaient car leur espace d'adressage était souvent réduit. Cette méthode a par contre l'inconvénient de nécessiter des instructions spécifiques pour accéder aux circuits d'entrées/sorties.



Adressage distinct des circuits d'entrées/sorties

La seconde méthode consiste à réserver une partie de l'adressage de la mémoire aux entrées/sorties. On parle d'*entrées/sorties en mémoire*. Cette méthode réduit donc l'espace d'adressage disponible pour la mémoire mais ce n'est pas gênant sur les micro-processeurs modernes qui disposent d'un (très) large espace d'adressage. Elle permet aussi l'utilisation des instructions de chargement et rangement pour accéder aux circuits d'entrées/sorties et de bénéficier ainsi des différents modes d'adressage. Cette méthode est toujours utilisée sur les micro-processeurs RISC car elle diminue le nombre d'instructions et simplifie le décodage de celles-ci.

13.2 Scrutation des circuits d'entrées/sorties

Lorsqu'un circuit d'entrée/sortie a reçu une donnée, le micro-processeur doit venir la lire pour la récupérer. Les buffers de ces circuits sont généralement assez petits. Ceci implique que la donnée doit être lue rapidement. Sinon, des données qui arrivent risquent d'être perdues.

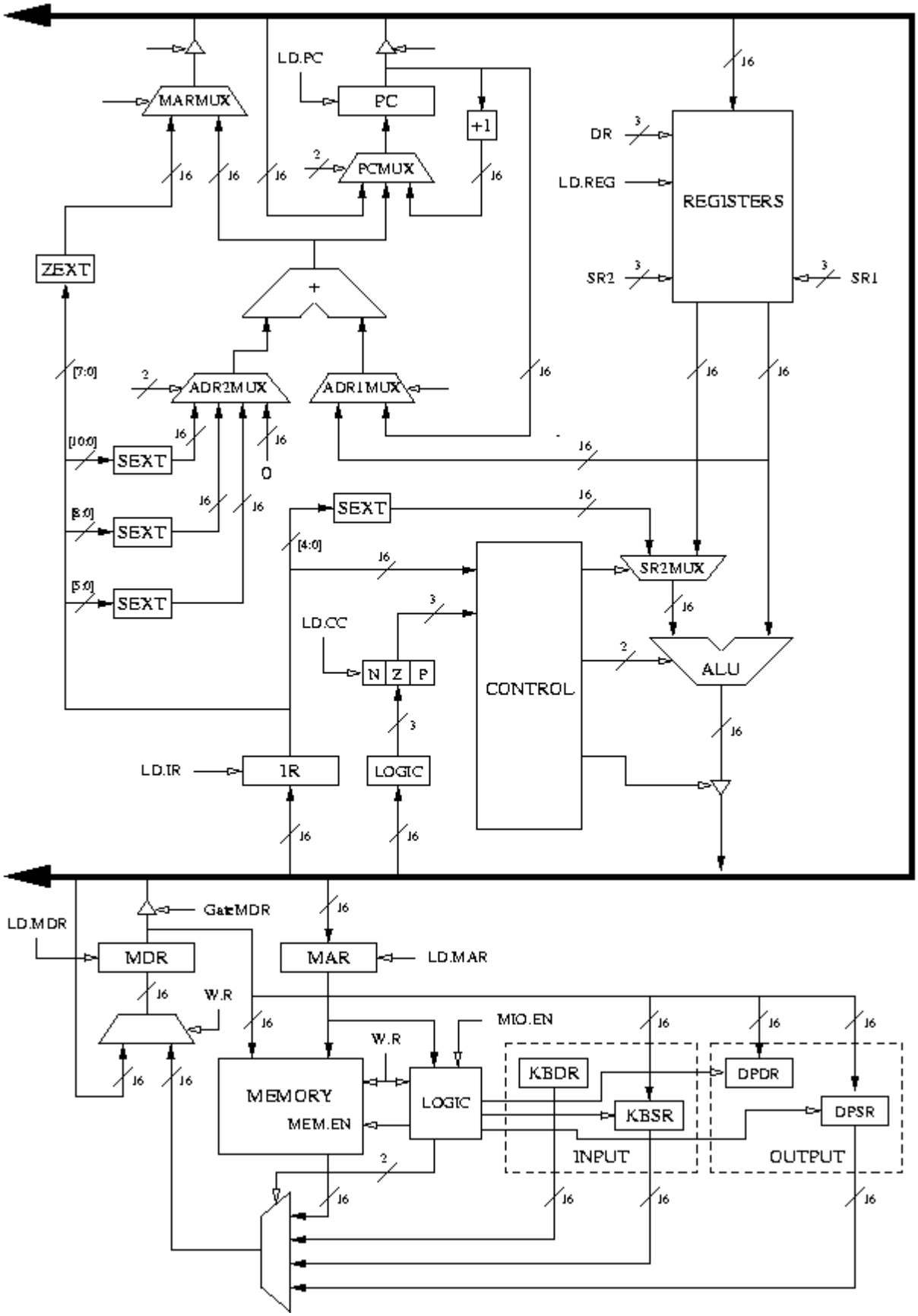
Il existe essentiellement deux méthodes pour éviter que des données soient perdues. La première consiste à faire en sorte que le micro-processeur interroge régulièrement les différents circuits d'entrée/sortie pour savoir si une donnée est disponible. Cette méthode est la plus simple. Par contre, elle utilise beaucoup le micro-processeur même quand il n'a aucune entrée/sortie. La seconde utilise les interruption qui permettent aux circuit d'entrée/sortie de prévenir le micro-processeur lorsqu'une donnée est disponible. Le micro-processeur interrompt alors la tâche en cours pour lire la donnée et la placer dans un buffer en mémoire où elle est mise en attente pour une tâche.

13.3 Entrées/Sorties du LC-3

Deux circuits d'entrée/sortie sont adjoints au micro-processeur. Le premier circuit permet de connecter un clavier et le second de le relier à un terminal pour afficher du texte. Chacun de ces circuits peut par exemple être un circuit pour une connexion série.

Le micro-processeur LC-3 utilise des entrées/sorties en mémoire. Deux adresses mémoire sont réservées pour chacun des deux circuits d'entrées/sorties. Une première adresse sert pour le registre de contrôle et une seconde sert pour le registre de donnée.

Un circuit extérieur marqué `Logic` sur la figure ci-dessous détermine si l'adresse présente dans le registre MAR (Memory Address Register) est une adresse en mémoire ou une adresse d'un circuit d'entrée/sortie. Dans le premier cas, il active la mémoire. Dans le second cas, il active le circuit correspondant à l'adresse.



Chemins de données du LC-3 avec les entrées/sorties

13.3.1 Lecture et écriture d'un caractère

13.3.1.1 Lecture

Le circuit pour le clavier utilise l'adresse 0xfe00 pour le registre de contrôle et l'adresse 0xfe02 pour le registre de donnée. Le registre de contrôle permet uniquement de savoir si un caractère est disponible. La valeur du bit de poids fort (numéro 15) de ce registre indique si un caractère est disponible. La lecture de ce caractère se fait alors dans le registre de donnée. La routine ci-dessous effectue une lecture bloquante d'un caractère du clavier.

```
DEFINE KBSR 4xfe00      ; Registre de contrôle (Status) du clavier
DEFINE KBDR 4xfe02      ; Registre de donnée du clavier

getc:
waitkb: LDI R2,pKBSR
        BRzp waitkb
        LDI R0,pKBDR
        RET


pKBSR: .FILL KBSR       ; Pointeur sur le registre de contrôle
pKBDR: .FILL KBDR       ; Pointeur sur le registre de donnée
```

13.3.1.2 Écriture

Le circuit pour l'écran utilise l'adresse 0xfe04 pour le registre de contrôle et l'adresse 0xfe06 pour le registre de donnée. Le registre de contrôle permet uniquement de savoir si le dernier caractère écrit a été envoyé et si le circuit est prêt à envoyer un nouveau caractère. L'envoi d'un caractère sur une ligne série prend un certain temps. Le micro-processeur doit attendre que l'émission d'un caractère soit terminée avant d'en écrire un autre dans le registre de donnée. La valeur du bit de poids fort (numéro 15) du registre de contrôle indique si le circuit est prêt à envoyer. L'émission du caractère est initiée en l'écrivant dans le registre de donnée du circuit. La routine ci-dessous effectue une écriture bloquante d'un caractère sur l'écran.

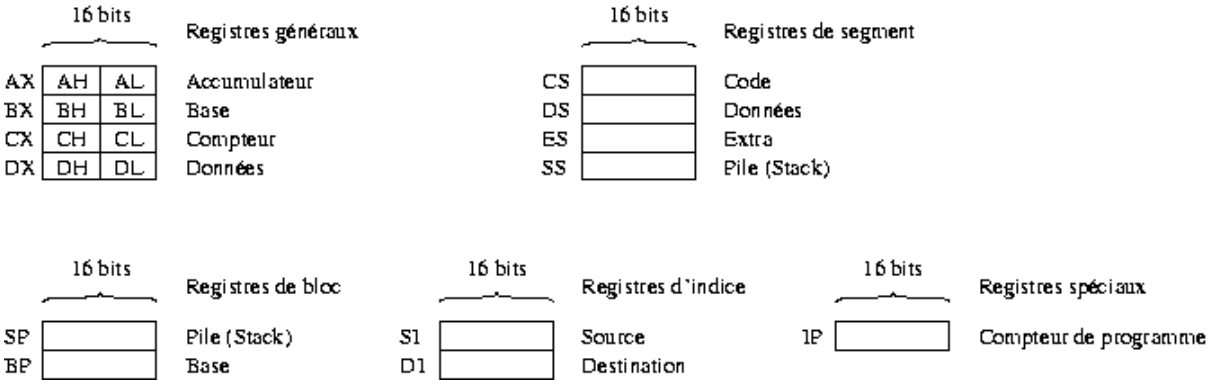
```
DEFINE DPSR 4xfe04      ; Registre de contrôle (Status) de l'écran
DEFINE DPDR 4xfe06      ; Registre de donnée de l'écran

putc:
waitdp:
        LDI R2,pDPSR
        BRzp waitdp
        STI R0,pDPDR
        RET

pDPSR: .FILL DPSR       ; Pointeur sur le registre de contrôle
pDPDR: .FILL DPDR       ; Pointeur sur le registre de donnée
```

14 Autres architectures

- processeurs 80x86
- comparaison CISC/RISC
- architecture IA-64

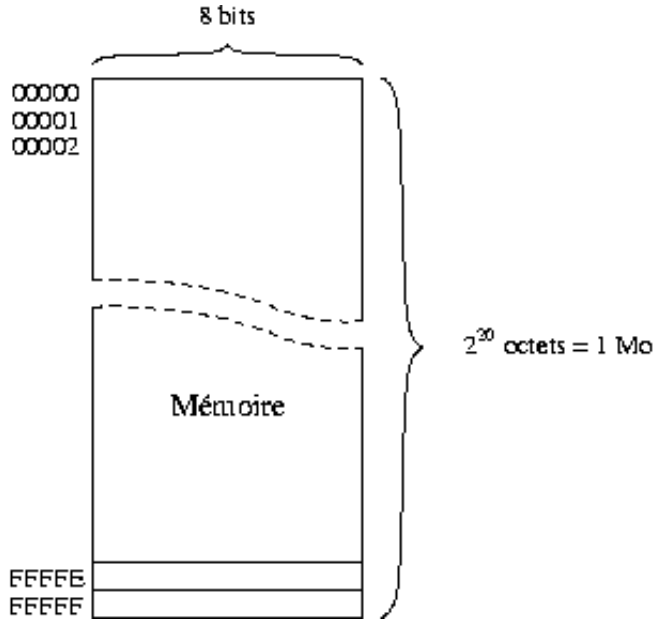
14.1 Processeurs 80x86

14.1.1 Registres



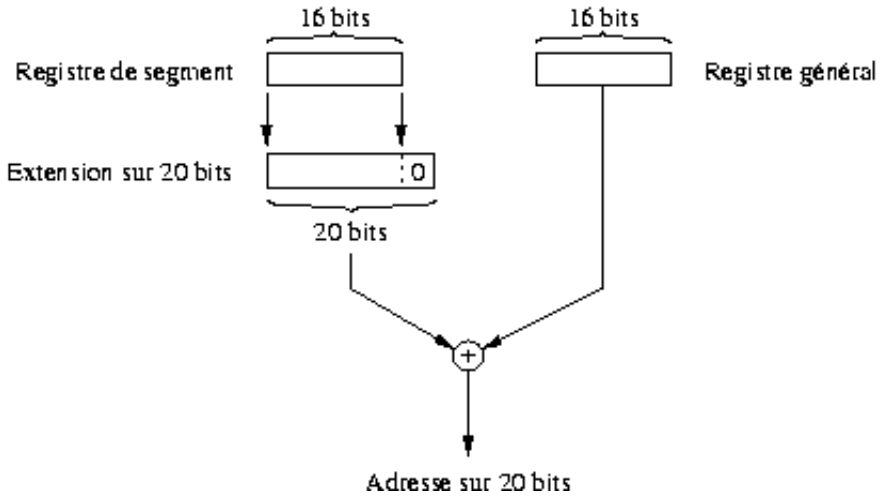
Registres du micro-processeur 8086

14.1.2 Mémoire



Mémoire du micro-processeur 8086

14.1.3 Adressage par segments



Adressage du micro-processeur 8086

14.1.4 Instructions

14.1.5 Instruction de chargement et rangement

- `MOV` chargement et rangement
- `PUSH` empilement
- `POP` dépilement

14.1.6 Instructions arithmétiques

Le micro-processeur possède un jeu d'instruction à deux opérandes. Ceci signifie que dans les opération arithmétiques et logiques, la destination est nécessairement une des deux operandes.

- `INC` incrémentation
- `DEC` décrémentation
- `ADD` addition
- `ADC` addition avec retenue
- `SUB` soustraction
- `CMP` comparaison (soustraction sans rangement)
- `NEG` opposé

14.1.7 Instructions logiques

- `NOT` négation
- `OR`, `XOR` *ou logique* et *ou exclusif*
- `AND`, `TEST` *et logique* et *et logique* sans rangement
- `SHL`, `SHR`, `SAL`, `SAR` décalages gauche et droite
- `ROL`, `ROR`, `RCL`, `RCR` rotations gauche et droite (avec injection de la retenue)

14.1.8 Branchements

- `JMP` branchement
- `JZ`, `JNZ` branchement si zéro ou non
- `JO`, `JNO` branchement si débordement (Overflow) ou non
- `JS`, `JNS` branchement si négatif ou non
- `JA`, `JAE`, `JG`, `JGE` branchement si plus grand (ou égal) ou non
- `JB`, `JBE`, `JL`, `JLE` branchement si plus petit (ou égal) ou non

14.1.9 Appel et retour de sous-routine

- `CALL` et `CALLF` appel et appel lointain
- `RET` et `RETF` retour et retour lointain
- `INT` appel système

14.1.10 Modes d'adressages

absolu

indirect
: avec les registres BX, SI ou DI

base + offset
: avec les registres BP, BX, BI ou SI avec offset 8 ou 16 bits

indexé
: l'addresse est la somme de deux registres. Les paires possibles sont BX+SI, BX+DI, BP+SI et BP+DI

base + index + offset
: identique au mode précédent avec en plus un offset 8 ou 16 bits

14.1.11 Instructions complexes

- `PUSHA`, `POPA` empilement et dépilement de tous les registres
- `ECHG` échange de deux valeurs
- `MOVS` déplacement de chaîne : mem[ES:DI] ← mem[DS:SI]; DI ← DI+1; SI ← SI+1. Cette instruction peut être précédée d'un préfixe comme `REP`, `REPZ` ou `REPNZ` qui donne le nombre de fois à la répéter.
- `LOOP label` décrémentation et saut : `DEC CX` et `JNZ label`

14.1.12 Programmation

Voici un petit exemple de programme qui calcule la longueur d'une chaîne de caractères

```
; Segment de données
data            SEGMENT
chaine          DB 'Hello world\0'
data            ENDS

; Segment de code
code            SEGMENT
                ASSUME DS:data, CS:code
```

```
main:           ; Initialisation du registre de segment de données
                MOV AX,data
                MOV DS,AX

                ; Chargement de l'adresse de la chaîne dans BX
                ; Il s'agit de l'adresse relative au début du segment
                MOV BX,offset chaine
                ; Initialisation compteur
                MOV CX,0

loop:           MOV AL,[BX]
                CMP AL,0        ; Test de fin de chaîne
                JZ fini
                INC BX          ; Incrémentation pointeur
                INC CX          ; Incrémentation compteur
                JMP loop

fini:           ; Retour au DOS
                MOV AH,4CH
                INT 21H
code            ENDS
                END main        ; Adresse de lancement
```

14.2 Comparaison CISC/RISC

Les processeurs généraux actuels se répartissent en deux grandes catégories appelées CISC pour *Complex Instruction Set Computer* et RISC pour *Reduced Instruction Set Computer*. Les processeurs de ces deux catégories se distinguent par la conception de leurs jeux d'instructions. Les processeurs CISC possèdent un jeu étendu d'instructions complexes. Chacune de ces instructions peut effectuer plusieurs opérations élémentaires comme charger une valeur en mémoire, faire une opération arithmétique et ranger le résultat en mémoire. Au contraire, les processeurs RISC possèdent un jeu d'instructions réduit où chaque instruction effectue une seule opération élémentaire. Le jeu d'instructions d'un processeur RISC est plus uniforme. Toutes les instructions sont codées sur la même taille et toutes s'exécute dans le même temps (un cycle d'horloge en général). L'organisation du jeu d'instructions est souvent appelé ISA pour *Instruction Set Architecture*

La répartition des principaux processeurs dans les deux catégories est la suivante.

CISC	RISC
S/360 (IBM) VAX (DEC) 68xx, 680x0 (Motorola) x86, Pentium (Intel)	Alpha (DEC) PowerPC (Motorola) MIPS PA-RISC (Hewlett-Packard) SPARC

14.2.1 Historique

Dans les premiers temps de l'informatique, les ordinateurs étaient programmés en langage machine ou en assembleur. Pour faciliter la programmation, les concepteurs de micro-processeurs dotèrent ceux-ci d'instructions de plus en plus complexes permettant aux programmeurs de coder de manière plus concise et plus efficace les programmes.

Lorsque les premiers langages de haut niveau remplacèrent l'assembleur, cette tendance s'accentua. Les concepteurs de micro-processeurs s'efforcèrent de combler le fossé entre le langage machine et les langages de haut niveau. Des instructions proches des constructions typiques des langages de haut niveau furent ajoutés aux micro-processeurs. L'idée était de faciliter la compilation des langages de haut niveau au détriment de la complexité des micro-processeurs. On ajouta par exemple des instructions spécifiques pour les appels/retours de fonctions ou des instructions spéciales pour les boucles pour décrémenter un registre et faire un saut si le résultat est non nul, tout ça en une seule instruction machine.

Un élément qui favorisa cette complexification des micro-processeurs était le manque (à cause du prix) de mémoire et la lenteur de celle-ci. L'absence de beaucoup de mémoire force les programmes à être le plus compacts possible. Pour réduire le nombre d'instructions nécessaires à un programme, il faut que chaque instruction fasse plusieurs opérations élémentaires. La lenteur relative de la mémoire pousse aussi à avoir des instructions complexes. Il n'y a alors moins de codes d'instructions à lire en mémoire et le programme en est accéléré.

Comme la densité d'intégration des transistors était encore faible, les micro-processeurs possédaient très peu de registres internes. De plus, un grand nombre de registres auraient nécessité plus de bits pour coder les instructions. Pour compenser cette lacune en registres, certaines instructions étaient capables, par exemple, de charger deux valeurs en mémoire, de faire la somme et de ranger le résultat

en mémoire. Il y avait de nombreux modes d'adressage et tous les modes d'adressages étaient possibles à toutes les instructions. On parle alors d'*orthogonalité* lorsque toutes les instructions peuvent utiliser tous les modes d'adressages. L'aboutissement de ce type de jeux d'instructions est celui du VAX.

La complexification des jeux d'instructions a pour effet de compliquer notablement la phase de décodage des instructions. On peux constater que sur certains micro-processeurs à jeu d'instructions complexe, la moitié des transistors sur la puce de silicium est consacrée au décodage des instructions et au contrôle de l'exécution de celles-ci.

Comme le décodage des instructions est rendu difficile par un jeu d'instructions complexes, l'exécution des instructions simples est ralenti par un décodage compliqué.

Lorsque le jeu d'instructions est complexe, la plupart des compilateurs n'utilisent pas tout le jeu d'instructions. Il se contentent souvent d'un nombre réduit d'instructions. Le résultat est que les instructions les plus puissantes sont très rarement utilisées. On arrive alors au paradoxe suivant. Les instructions complexes qui ne sont pratiquement pas utilisées ralentissent les instructions simples qui sont utilisées la plupart du temps.

Des études statistiques sur des programmes tels des systèmes d'exploitation ou des applications réelles ont montré les faits suivants.

- 80 % des programmes n'utilisent que 20 % du jeu d'instructions.
- Les instructions les plus utilisées sont :
  - les instructions de chargement et de rangement,
  - les appels de sous-routines.
- Les appels de fonctions sont très gourmands en temps : sauvegarde et restitution du contexte et passage des paramètres et de la valeur de retour.
- 80 % des variables locales sont des entiers.
- 90 % des structures complexes sont des variables globales.
- La profondeur maximale d'appels imbriqués et en moyenne de 8. Une profondeur plus importante ne se rencontre que dans 1 % des cas.

L'apparition de l'architecture RISC vint de la volonté de favoriser au maximum les instructions simples qui constituent la grande partie des programmes. L'idée est de supprimer les instructions complexes et les modes d'adressage élaborés afin d'augmenter la fréquence d'horloge et d'augmenter ainsi la vitesse d'exécution de toutes les instructions. La fréquence d'horloge d'un micro-processeur est souvent dictée par les instructions les plus complexes qui prennent plus de temps.

La philosophie essentielle des processeurs RISC est d'avoir un nombre important de registres. Des instructions de chargement et de rangement avec quelques modes d'adressage sont les seules à faire les échanges avec la mémoire. Toutes les autres instructions travaillent uniquement avec les registres.

L'apparition des micro-processeurs RISC est en partie due l'augmentation de la mémoire disponible sur les ordinateurs. Celle-ci n'est plus une limitation à la taille des programme. Un autre facteur important est le fossé qui s'est creusé entre la vitesse des processeurs et celle de la mémoire. Les processeurs ont une fréquence d'horloge élevée par rapport à la vitesse de la mémoire. Chaque accès à la mémoire les pénalise. Ceci accentue le rôle d'un nombre important de registres qui évitent les accès à la mémoire. La lenteur de la mémoire est en partie compensée par la présence de caches faits de mémoires rapides. Ceux-ci sont très efficaces pour lire les instructions des programmes puisque ces accès à la mémoire se font à des cases mémoire contiguës.

14.2.2 Caractéristiques

Les principales caractéristiques des processeurs RISC sont les suivantes.

Codage uniforme des instructions
: Toutes les instructions sont codées avec un même nombre de bits, généralement un mot machine. L'*op-code* se trouve à la même position pour toutes les instructions. Ceci facilite le décodage des instructions.

Registres indifférenciés et nombreux
: Tous les registres peuvent être utilisés dans tous les contextes. Il n'y a par exemple pas de registre spécifique pour la pile. Les processeurs séparent cependant les registres pour les valeurs flottantes des autres registres.

Limitation des accès mémoire
: Les seules instructions ayant accès à la mémoire sont les instructions de chargement et de rangement. Toutes les autres instructions opèrent sur les registres. Il en résulte une utilisation intensive des registres.

Nombre réduit de modes d'adressage
: Il n'y pas de mode d'adressage complexe. Les modes d'adressages possibles sont généralement immédiat, direct, indirect et relatifs. Un registre est souvent fixé à la valeur 0 afin d'obtenir certains modes d'adressages simples comme cas particulier d'autres modes d'adressage.

Nombre réduit de types de données
: Les seuls types de données supportés sont les entiers de différentes tailles (8, 16, 32 et 64 bits) et des nombres flottants en simple et double précision. Certains processeurs CISC comportent des instructions pour le traitement des chaînes de caractères, des polynômes ou des complexes

La simplification du jeu d'instructions a reporté une partie du travail sur le compilateur. Ce dernier joue un rôle essentiel dans l'optimisation du code engendré. Il doit en particulier gérer les points suivants.

- allocation optimale des registres,
- élimination des redondances,
- optimisation des boucles en ne conservant à l'intérieur que ce qui est modifié,
- optimisation du pipeline,
- optimisation du choix des instructions,

14.3 Architecture IA-64

L'architecture IA-64 (Intel Architecture 64 bits) est l'architecture des processeurs 64 bits Itanium et Itanium 2 développés conjointement par Intel et Hewlett-Packard. Elle utilise une technique appelée EPIC (Explicitly Parallel Instruction Computing).

L'architecture de ce micro-processeur se distingue des autres architectures car elle introduit plusieurs concepts novateurs. Bien que l'Itanium soit un (semi-)échec commercial, il est intéressant d'en étudier les caractéristiques essentielles.

Le principe de la technique EPIC est d'avoir une parallélisation explicite des instructions. Dans un programme d'un micro-processeur classique, les instructions sont disposées de manière séquentielle. Le micro-processeur peut essayer d'en exécuter certaines en parallèle mais il ne dispose d'aucune indication pour l'aider. Il doit donc à la volée gérer les dépendances des instructions et leur affecter une unité d'exécution. Les instructions du processeur Itanium sont au contraire disposées en paquets (appelés *bundles*), prêtes à être parallélisées. Chaque paquet dispose en plus d'indications sur les unités nécessaires à l'exécution de ses instructions. La parallélisation des instructions est donc préparée au moment de la compilation et de l'assemblage.

Voici quelques caractéristiques essentielles de l'architecture IA-64.

- Nombre important de registres pour faciliter la parallélisation des instructions.
- Certains registres sont statiques et d'autres sont gérés sous forme d'une pile pour faciliter les appels de fonctions.
- Instruction de chargement anticipé pour masquer la latence de la mémoire
- Regroupement des instructions en paquets (*bundles*) avec des indications explicites des unités nécessaires
- Exécution par prédicats des instructions.

14.3.1 Organisation

L'organisation de l'architecture IA-64 est basée sur un nombre important de registres et sur de multiples unités d'exécution. Ces deux caractéristiques permettent un degré élevé de parallélisme. Dans un micro-processeur classique, le nombre de registres réellement disponibles peut être supérieur au nombre de registres utilisés par l'assembleur. Le micro-processeur effectue à la volée un renommage des registres afin d'utiliser au mieux tous les registres. Dans l'architecture IA-64, tous les registres sont visibles de l'assembleur.

14.3.1.1 Registres

Les registres dont dispose l'architecture IA-64 sont les suivants.

- 128 registres généraux de 64 bits
- 128 registres applicatifs de 64 bits
- 128 registres flottants de 82 bits
- 64 bits de prédictions

14.3.1.2 Unités d'exécution

Le nombre d'unités d'exécution peut varier et dépend du nombre de transistors disponibles pour une implémentation donnée. Le micro-processeur s'arrange pour utiliser au mieux les unités dont il dispose. L'architecture distingue les quatre types suivants d'unités d'exécution.

Unité I
: unité arithmétique et logique sur les entiers (additions, soustractions, opérations logiques, décalage)

Unité M
: unité de chargement et de rangement entre les registres et la mémoire, plus quelques opérations sur les entiers

Unité B
: unité de branchement

Unité F
: unité d'opérations sur les nombres flottants

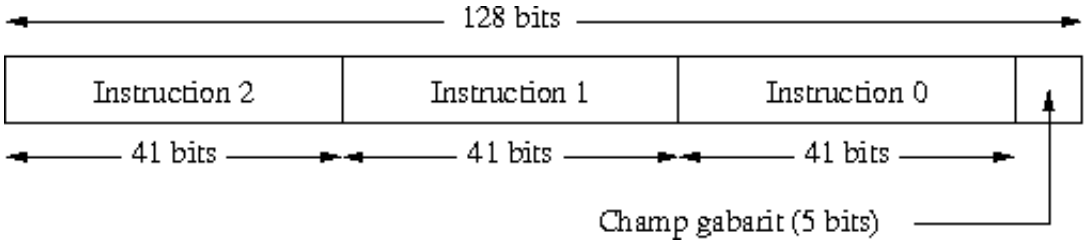
14.3.1.3 Type d'instructions

L'architecture distingue aussi des types d'instructions qui correspondent plus ou moins aux unités qui peuvent les exécuter.

Type d'instruction	Description	Unité d'exécution
A	UAL	Unité I ou unité M
I	Entier non UAL	Unité I
M	Accès mémoire	Unité M
F	Opération sur flottants	Unité F
B	Branchement	Unité B
L + X	Étendu	Unité I/Unité B

14.3.2 Format des Instructions

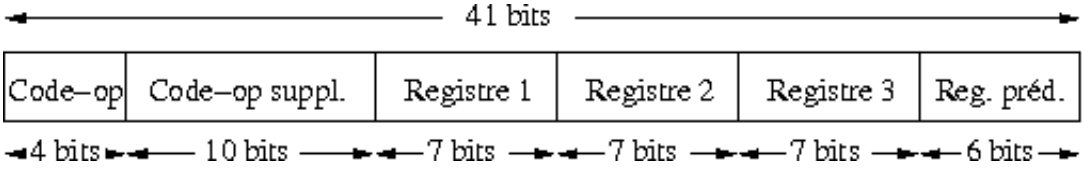
Les instructions sont regroupées en paquets (*bundles*) de 128 bits contenant chacun trois instructions et des indications de parallélisations appelées *champ gabarit*. Chaque instruction est codée sur 41 bits. Il reste donc 5 = 128 - 3×41 bits pour le champ gabarit. Le format d'un paquet est le suivant.



Format d'un paquet de 128 bits

Chaque instruction est codée sur 41 bits. Les 4 premiers bits donnent le code-op principal et les 6 derniers bits donnent le numéro du registre prédicatif qui conditionne l'exécution de l'instruction. L'utilisation des 31 = 41 - 4 - 6 bits restant dépend du code-op principal. Une utilisation typique est

cependant de coder un code-op supplémentaire sur les 10 premiers bits puis de coder 3 registres sur 7 bits chacun.



Format d'une instruction sur 41 bits

L'interprétation du code-op principal dépend du champ gabarit car celui-ci indique déjà l'unité d'exécution nécessaire à l'instruction. Pour que cette information ne soit pas redondante avec le code-op, c'est le code-op combiné avec le champ gabarit qui détermine l'instruction précise. Le gabarit indique d'abord une catégorie d'instructions, le code-op donne ensuite l'instruction dans la catégorie et le code-op supplémentaire peut encore préciser l'instruction. Il s'agit en fait d'une organisation hiérarchique des codes-op.

14.3.2.1 Champ gabarit

Le champ gabarit code deux informations distinctes. D'une part, il donne l'unité d'exécution nécessaire à chacune des trois instructions du paquet. D'autre part, il précise les positions d'éventuelles *limites*. Ces limites séparent des blocs d'instructions qui peuvent chevaucher plusieurs paquets. Le micro-processeur peut en effet charger plusieurs paquets simultanément pour essayer d'exécuter un maximum d'instructions en parallèle. Les blocs sont des suites d'instructions consécutives dans le flot d'instructions. Une limite indique qu'une instruction après elle peut avoir une dépendance avec une instruction avant elle. Autrement dit, toutes les instructions entre deux limites consécutives ne présentent pas de dépendance. Elles peuvent donc être exécutées en parallèle.

Ces deux informations sont regroupées dans un *modèle* dont le numéro est donné par le champ gabarit. Parmi les 32 codes possibles, seuls 24 sont assignés à des modèles. Les 8 codes restants sont réservés pour une utilisation ultérieure. La table suivante donne les modèles prévus. Les limites sont indiquées par une double barre verticale ||.

Modèle	Instruction 0	Limite	Instruction 1	Limite	Instruction 2	Limite
00	Unité M		Unité I		Unité I	
01	Unité M		Unité I		Unité I	||
02	Unité M		Unité I	||	Unité I	
03	Unité M		Unité I	||	Unité I	||
04	Unité M		Unité L		Unité X	
05	Unité M		Unité L		Unité X	||
08	Unité M		Unité M		Unité I	
09	Unité M		Unité M		Unité I	||
0A	Unité M	||	Unité M		Unité I	
0B	Unité M	||	Unité M		Unité I	||
0C	Unité M		Unité M		Unité I	
0D	Unité M		Unité F		Unité I	||
0E	Unité M		Unité M		Unité F	
0F	Unité M		Unité M		Unité F	||
10	Unité M		Unité I		Unité B	
11	Unité M		Unité I		Unité B	||
12	Unité M		Unité B		Unité B	
13	Unité M		Unité B		Unité B	||
16	Unité B		Unité B		Unité B	
17	Unité B		Unité B		Unité B	||
18	Unité M		Unité M		Unité B	
19	Unité M		Unité M		Unité B	||
1C	Unité M		Unité F		Unité B	
1D	Unité M		Unité F		Unité B	||

Il est à la charge du compilateur de regrouper au mieux les instructions en paquets puis en blocs afin d'optimiser l'utilisation des unités d'exécution du micro-processeur.

14.3.3 Exécution prédicative

Les registres prédicatifs sont des registres binaires qui sont positionnés par les instructions de comparaison. Ils s'apparentent donc aux indicateurs binaires comme n, z et p présents dans la plupart des micro-processeurs. Par contre, ils sont tous indifférenciés et chacun d'entre eux peut être positionné par une comparaison. En fait, chaque comparaison positionne deux registres prédicatifs avec des valeurs opposées.

L'exécution de chaque instruction est conditionnée par la valeur d'un des registres prédicatifs déterminé par les 6 derniers bits de l'instruction. Celle-ci est exécutée seulement si ce registre prédicatif vaut 1. Sinon, elle n'est pas exécutée. Le registre prédicatif p0 a toujours la valeur 1. Une instruction indique donc ce registre prédicatif si elle doit toujours être exécutée.

La compilation d'une structure de contrôle `if-then-else` d'un langage de haut niveau se traduit par des sauts en langage d'assembleur. La compilation du morceau de code ci-dessous

```
  if (cond) {
    bloc 1
  } else {
    bloc 2
  }
```

conduit au code d'assembleur LC-3 suivant.

```
        ; Calcul de cond
        ...
        BRz bloc2       ; Condition fausse
        ; Début du bloc 1
        ...
        BR finif        ; Fin du bloc 1
bloc2:  ; Début du bloc 2
        ...
finif:  ; Suite du programme
```

Si les blocs d'instructions 1 et 2 sont relativement courts, la présence des sauts pénalise l'exécution du programme.

Le principe d'utilisation des registres prédicatifs est le suivant. La calcul de la condition positionne deux registres prédicatifs p1 et p2 de manière opposée. Si la condition est vraie, p1 est positionné à vrai et p2 à faux et si la condition est fausse, p1 est positionné à faux et p2 à vrai. Ensuite, le programme exécute les instructions du bloc 1 conditionnées par p1 et les instructions du bloc 2 conditionnées par p2. Cela donne le code en assembleur suivant où les sauts ont disparu. La syntaxe `(p) instr` signifie que l'instruction `instr` est conditionnée par le registre prédicatif `p`. Le micro-processeur exécute plus d'instructions mais cela est largement compensé par la disparition des sauts si les blocs 1 et 2 sont assez courts.

```
        ; Calcul de cond
        ...
        ; Positionnement de p1 et p2
        ; Début du bloc 1
        (p1) instr1
        (p1) instr2
        ...
        ; Début du bloc 2
        (p2) instr1
        (p2) instr2
        ...
        ; Suite du programme
```

14.3.4 Chargement spéculatif

L'idée du *chargement spéculatif* est d'anticiper le chargement de valeurs à partir de la mémoire afin d'éviter que le calcul reste bloqué en attente d'une valeur. Le chargement d'une valeur est scindé en deux actions. La première est de demander le chargement sans attendre que celui-ci soit terminé. La seconde est de vérifier que le chargement a été effectivement accompli au moment où la valeur doit être utilisée. Le principe est de placer la *demande* bien avant l'utilisation de la valeur. Le chargement a alors largement le temps de s'effectuer. La *vérification* est ensuite faite juste avant d'utiliser la valeur. Si la valeur n'est pas encore chargée, le calcul est bloqué mais cet événement survient rarement.

15 Pipeline

15.1 Principe

Le pipeline est un mécanisme permettant d'accroître la vitesse d'exécution des instructions dans un micro-processeur. L'idée générale est d'appliquer le principe du *travail à la chaîne* à l'exécution des instructions. Dans un micro-processeur sans pipeline, les instructions sont exécutées les unes après les autres. Une nouvelle instruction n'est commencée que lorsque l'instruction précédente est complètement terminée. Avec un pipeline, le micro-processeur commence une nouvelle instruction avant d'avoir fini la précédente. Plusieurs instructions se trouvent donc simultanément en cours d'exécution au cœur du micro-processeur. Le temps d'exécution d'une seule instruction n'est pas réduit. Par contre, le débit du micro-processeur, c'est-à-dire le nombre d'instructions exécutées par unité de temps, est augmenté. Il est multiplié par le nombre d'instructions qui sont exécutées simultanément.

15.2 Étages du pipeline

Afin de mettre en œuvre un pipeline, la première tâche est de découper l'exécution des instructions en plusieurs *étapes*. Chaque étape est prise en charge par un *étage* du pipeline. Si le pipeline possède n étages, il y a n instructions en cours d'exécution simultanée, chacune dans une étape différente. Le facteur d'accélération est donc le nombre n d'étages. On verra que plusieurs problèmes réduisent ce facteur d'accélération. Sur les micro-processeurs actuels, le nombre d'étages du pipeline peut atteindre une douzaine ou même une vingtaine. La fréquence d'horloge est limitée par l'étape qui est la plus longue à réaliser. L'objectif des concepteurs de micro-processeurs est d'équilibrer au mieux les étapes afin d'optimiser la performance.

Pour le micro-processeur LC-3, on va découper l'exécution des instructions en les cinq étapes suivantes. On va donc construire un pipeline à cinq étages.

Lecture de l'instruction (Instruction Fetch)
: La prochaine instruction à exécuter est chargée à partir de la case mémoire pointée par le compteur de programme PC dans le registre d'instruction IR. Ensuite le compteur de programme est incrémenté pour pointer sur l'instruction suivante.

Décodage de l'instruction (Instruction Decode)
: Cette étape consiste à préparer les arguments de l'instruction pour l'étape suivante où ils seront utilisés. Ces arguments sont placés dans deux registres A et B. Si l'instruction utilise le contenu de un ou deux registres, ceux-ci sont lus et leurs contenus sont rangés en A et B. Si l'instruction contient une valeur immédiate, celle-ci est étendue (signée ou non signée) à 16 bits et placée dans le registre B. Pour les instructions de branchement avec offset, le contenu de PC est rangé en A et l'offset étendu dans B. Pour les instructions de branchement avec un registre, le contenu de ce registre est rangé en A et B est rempli avec 0. Les instructions de rangement `ST*` mettent le contenu du registre qui doit être transféré en mémoire dans le registre C.

Exécution de l'instruction (Instruction Execution)
: Cette étape utilise l'unité arithmétique et logique pour combiner les arguments. L'opération précise réalisée par cette étape dépend du type de l'instruction.

  Instruction arithmétique ou logique (`ADD`, `AND` et `NOT`)
  : Les deux arguments contenus dans les registres A et B sont fournis à l'unité arithmétique et logique pour calculer le résultat.

Instruction de chargement et rangement (`LD*` et `ST*`)
: Le calcul de l'adresse est effectué à partir de l'adresse provenant du registre A et de l'offset contenu dans le registre B.

Instruction de branchement (`BR*`, `JMP`, `JSR`, `JSRR` et `TRAP`)
: Pour les instructions contenant un offset, l'addition avec le contenu du PC est effectuée à cette étape. Pour les instructions utilisant un registre, le contenu du registre est juste transmis.

Accès à la mémoire (Memory Access)
: Cette étape est uniquement utile pour les instruction de chargement et de rangement. Pour les instructions arithmétiques et logiques ou les branchements, rien n'est effectué. L'adresse du mot mémoire est contenue dans le registre R. Dans le cas d'un rangement, la valeur à ranger provient du registre C. Dans le cas d'un chargement, la valeur lue en mémoire est mise dans le registre R pour l'étape suivante.

Rangement du résultat (Write Back)
: Le résultat des opérations arithmétiques et logiques est rangé dans le registre destination. La valeur lue en mémoire par les instructions de chargement est aussi rangée dans le registre destination. Les instructions de branchement rangent la nouvelle adresse dans PC.

Ce schéma ne s'applique pas aux deux instructions `LDI` et `STI` qui nécessitent deux accès à la mémoire. Les processeurs utilisant un pipeline ne possèdent pas d'instruction avec ce mode d'adressage indirect. Dans la suite, on ignore ces deux instructions.

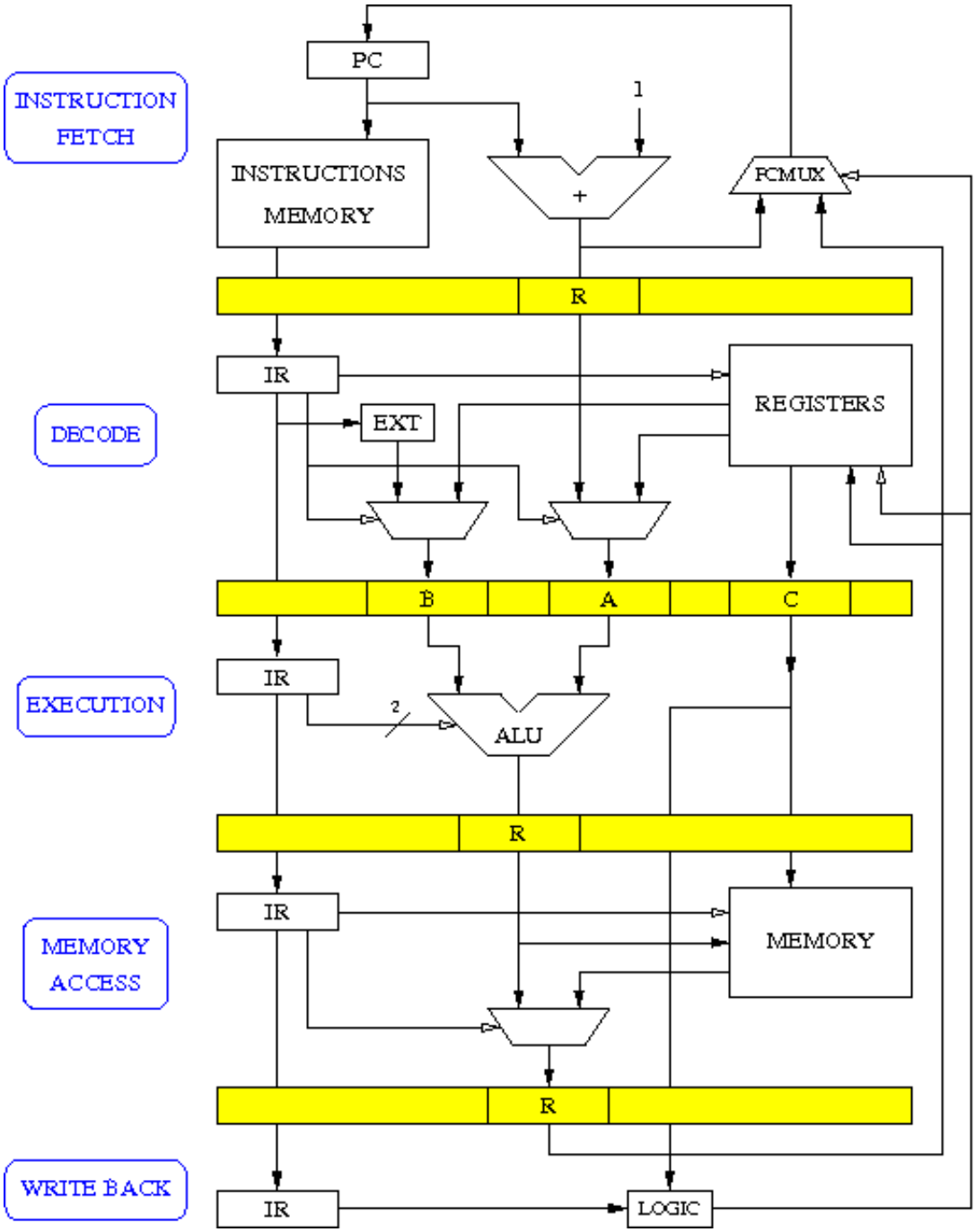
Le pipeline a réduit le nombre d'additionneurs puisque le même est utilisé pour les calculs arithmétiques et les calculs d'adresses. Par contre, certains registres ont dû être multipliés. Le registre d'instruction IR est présent à chaque étage du pipeline. En effet, l'opération à effectuer à chaque étage dépend de l'instruction en cours. Cette multiplication de certains éléments est le prix à payer pour l'exécution simultanée de plusieurs instructions.

Le tableau ci-dessous représente l'exécution d'un programme pendant quelques cycles d'horloge. Chacune des instructions commence son exécution un cycle après l'instruction précédente et passe par les cinq étapes d'exécution pendant cinq cycles consécutifs.

Programme	**Cycles d'horloge**									
	1	2	3	4	5	6	7	8	9	10
Inst. n° 1	IF	ID	IE	MA	WB					
Inst. n° 2		IF	ID	IE	MA	WB				
Inst. n° 3			IF	ID	IE	MA	WB			
Inst. n° 4				IF	ID	IE	MA	WB		
Inst. n° 5					IF	ID	IE	MA	WB	
Inst. n° 6						IF	ID	IE	MA	WB

15.3 Réalisation du pipeline

Le pipeline est réalisé comme à la figure ci-dessous.



Pipeline

15.4 Aléas

Le bon fonctionnement du pipeline peut être perturbé par plusieurs événements appelés *aléas* (*pipeline hazard* en anglais). Ces événements sont classés en trois catégories.

aléas structurels
: Ce type de problèmes survient lorsque deux instructions dans des étages différents du pipeline nécessitent la même ressource.

aléas de données
: Ce type de problèmes survient lorsqu'une instruction nécessite une donnée qui n'a pas encore été calculée par une instruction précédente. Ceci provient du fait que les instructions lisent leurs arguments dans les premiers étages du pipeline alors qu'elles produisent leur résultat dans les derniers étages.

aléas de contrôle
: Ce type de problèmes survient dès qu'une instruction de branchement est exécutée. Si le branchement est effectué, les instructions qui suivent dans le pipeline ne doivent pas être exécutée. Ceci provient du fait que la nouvelle adresse est calculée alors que les instructions qui suivent ont déjà été chargées

La solution générale pour résoudre un aléa est de bloquer l'instruction qui pose problème et toutes celles qui suivent dans le pipeline jusqu'à ce que le problème se résolve. On voit alors apparaître des *bulles* dans le pipeline. De manière pratique, la bulle correspond à l'exécution de l'instruction `NOP` qui ne fait rien.

Programme	**Cycles d'horloge**											
	1	2	3	4	5	6	7	8	9	10	11	12
Inst. n° 1	IF	ID	IE	MA	WB							
Inst. n° 2		IF	ID	IE	MA	WB						
Inst. n° 3			IF	ID	Bulle		IE	MA	WB			
Inst. n° 4				IF	Bulle		ID	IE	MA	WB		
Inst. n° 5							IF	ID	IE	MA	WB	
Inst. n° 6								IF	ID	IE	MA	WB

15.4.1 Aléas structurels

Considérons par exemple le morceau de code suivant.

```
LDR R7,R6,0
ADD R6,R6,1
ADD R0,R0,1
ADD R1,R1,1
```

Le déroulement de l'exécution des quatre premières instructions dans le pipeline devrait être le suivant.

Programme	**Cycles d'horloge**							
	1	2	3	4	5	6	7	8
`LDR R7,R6,0`	IF	ID	IE	MA	WB			
`ADD R6,R6,1`		IF	ID	IE	MA	WB		
`ADD R0,R0,1`			IF	ID	IE	MA	WB	
`ADD R1,R1,1`				IF	ID	IE	MA	WB

L'étape MA (accès mémoire) de l'instruction `LDR R7,R6,0` a lieu en même temps que l'étape IF (chargement de l'instruction) de l'instruction `ADD R1,R1,1`. Ces deux étapes nécessitent simultanément l'accès à la mémoire. Il s'agit d'un *aléa structurel*. Comme cela est impossible, l'instruction `ADD R1,R1,1` et celles qui suivent sont retardées d'un cycle.

Programme	**Cycles d'horloge**										
	1	2	3	4	5	6	7	8	9	10	11
`LDR R7,R6,0`	IF	ID	IE	MA	WB						
`ADD R6,R6,1`		IF	ID	IE	MA	WB					
`ADD R0,R0,1`			IF	ID	IE	MA	WB				
`ADD R1,R1,1`					IF	ID	IE	MA	WB		
Inst. n° 5						IF	ID	IE	MA	WB	
Inst. n° 6							IF	ID	IE	MA	WB

Le conflit d'accès à la mémoire se produit à chaque fois qu'une instruction de chargement ou de rangement est exécutée. Celle-ci rentre systématiquement en conflit avec le chargement d'une instruction qui a lieu à chaque cycle d'horloge. Ce problème est généralement résolu en séparant la mémoire où se trouvent les instructions de celle où se trouvent les données. Ceci est réalisé au niveau des caches de niveau 1. Le micro-processeur doit alors avoir deux bus distincts pour accéder simultanément aux deux caches.

Comme le micro-processeur LC-3 est très simple, les seuls aléas structurels possibles sont ceux dûs à des accès simultanés à la mémoire. Les registres sont utilisés aux étapes ID (décodage) et WB (rangement du résultat) mais les premiers accès sont en lecture et les seconds en écriture. Ceci ne provoque pas d'aléa.

15.4.2 Aléas de données

Considérons le morceau de code suivant.

```
ADD R1,R1,1
ADD R0,R1,R2
```

Le déroulement de l'exécution de ces deux instructions dans le pipeline devrait être le suivant.

Programme	**Cycles d'horloge**					
	1	2	3	4	5	6
`ADD R1,R1,1`	IF	ID	IE	MA	WB	
`ADD R2,R1,R0`		IF	ID	IE	MA	WB

Le problème est que le résultat de la première instruction est écrit dans le registre R1 après la lecture de ce même registre par la seconde instruction. La valeur utilisée par la seconde instruction est alors erronée.

Le résultat de la première instruction est disponible dès la fin de l'étape IE (exécution de l'instruction) de celle-ci. Il est seulement utilisé à l'étape IE de la seconde instruction. Il suffit alors de le fournir en entrée de l'additionneur à la place de la valeur lue dans R1 par la seconde instruction. Ceci est réalisé

en ajoutant un chemin de données.

Programme	Cycles d'horloge					
	1	2	3	4	5	6
`ADD R1,R1,1`	IF	ID	IE	MA	WB	
`ADD R2,R1,R0`		IF	ID	IE	MA	WB

Considérons un autre morceau de code assez semblable.

```
LDR R1,R6,0
ADD R0,R1,R2
```

Le déroulement de l'exécution de ces deux instructions dans le pipeline devrait être le suivant.

Programme	Cycles d'horloge					
	1	2	3	4	5	6
`LDR R1,R6,0`	IF	ID	IE	MA	WB	
`ADD R2,R1,R0`		IF	ID	IE	MA	WB

Dans cet exemple encore, le résultat de la première instruction est écrit dans le registre R1 après la lecture de ce même registre par la seconde instruction. Par contre, le résultat n'est pas disponible avant l'étape MA (accès mémoire) de la première instruction. Comme cette étape a lieu après l'étape IE de la seconde instruction, il ne suffit pas d'ajouter un chemin de données pour faire disparaître l'aléa. Il faut en outre retarder la seconde instruction. Ceci introduit une bulle dans le pipeline.

Programme	Cycles d'horloge								
	1	2	3	4	5	6	7	8	9
`LDR R1,R6,0`	IF	ID	IE	MA	WB				
`ADD R2,R1,R0`		IF	ID		IE	MA	WB		
Inst. n° 3			IF		ID	IE	MA	WB	
Inst. n° 4					IF	ID	IE	MA	WB

15.4.2.1 Optimisation du code par le compilateur

Considérons le morceau de programme suivant écrit dans un langage comme C.

```
x = x1 + x2;
y = y1 + y2;
```

La compilation de ce morceau de code pourrait produire les instructions suivantes où `x`, `x1`, …, `y2` désignent alors les emplacements mémoire réservés à ces variables.

```
LD R1,x1
LD R2,x2
ADD R0,R1,R2
ST R0,x
LD R1,y1
LD R2,y2
ADD R0,R1,R2
ST R0,y
```

L'exécution de ces instructions provoque des aléas de données entre les instructions de chargement et les instructions d'addition. Ces aléas introduisent des bulles dans le pipeline. Ces bulles peuvent être évitées si le compilateur ordonne judicieusement les instructions comme dans le code ci-dessous. Plus de registres sont alors nécessaires.

```
LD R1,x1
LD R2,x2
LD R3,y1
ADD R0,R1,R2
LD R4,y2
ST R0,x
ADD R0,R3,R4
ST R0,y
```

15.4.3 Aléas de branchement

Lors de l'exécution d'une instruction de branchement conditionnel, on dit que le branchement est *pris* si la condition est vérifiée et que le programme se poursuit effectivement à la nouvelle adresse. Un branchement sans condition est toujours pris.

Lorsqu'un branchement est pris, l'adresse de celui-ci est calculée à l'étape IE (exécution de l'instruction) et rangée dans le registre PC à l'étape WB (rangement du résultat). Toutes les instructions qui suivent l'instruction de branchement ne doivent pas être exécutées. Au niveau du pipeline, on obtient le diagramme d'exécution suivant.

Programme	**Cycles d'horloge**											
	1	2	3	4	5	6	7	8	9	10	11	12
Branchement	IF	ID	IE	MA	WB							
Inst. suivante n° 1		IF	ID	IE								
Inst. suivante n° 2			IF	ID								
Inst. suivante n° 3				IF								
Inst. cible n° 1						IF	ID	IE	MA	WB		
Inst. cible n° 2							IF	ID	IE	MA	WB	
Inst. cible n° 3								IF	ID	IE	MA	WB

On constate que l'exécution d'un branchement pris dégrade notablement la performance du pipeline puisque quatre cycles sont perdus. Comme les branchements constituent en général 20 à 30% des instructions exécutées par un programme, il est primordial d'améliorer leur exécution.

Dans le cas du micro-processeur LC-3, les instructions de branchement sont relativement simples. Une façon simple d'optimiser les branchements est de ne pas leur faire suivre toutes les étapes du pipeline afin que la nouvelle adresse soit écrite le plus tôt possible dans le registre PC.

Pour les branchements conditionnels, la condition ne dépend que des indicateurs n, z et p et du code de l'instruction. Cette condition peut donc être calculée à l'étape ID (décodage de l'instruction). De même, l'adresse du branchement est soit le contenu d'un registre soit la somme de PC et d'un offset. Dans les deux cas, cette valeur peut être rendue disponible à la fin de l'étape ID. Le prix à payer est l'ajout d'un nouvel additionneur dédié à ce calcul. La nouvelle adresse est alors écrite dans le registre PC à la fin de l'étape ID.

Le diagramme précédent devient alors le diagramme ci-dessous qui montre qu'il ne reste plus qu'un seul cycle d'horloge de perdu.

Programme	**Cycles d'horloge**								
	1	2	3	4	5	6	7	8	9
Branchement	IF	ID							
Inst. suivante n° 1		IF							
Inst. cible n° 1			IF	ID	IE	MA	WB		
Inst. cible n° 2				IF	ID	IE	MA	WB	
Inst. cible n° 3					IF	ID	IE	MA	WB

16 Mémoire virtuelle

16.1 Principe

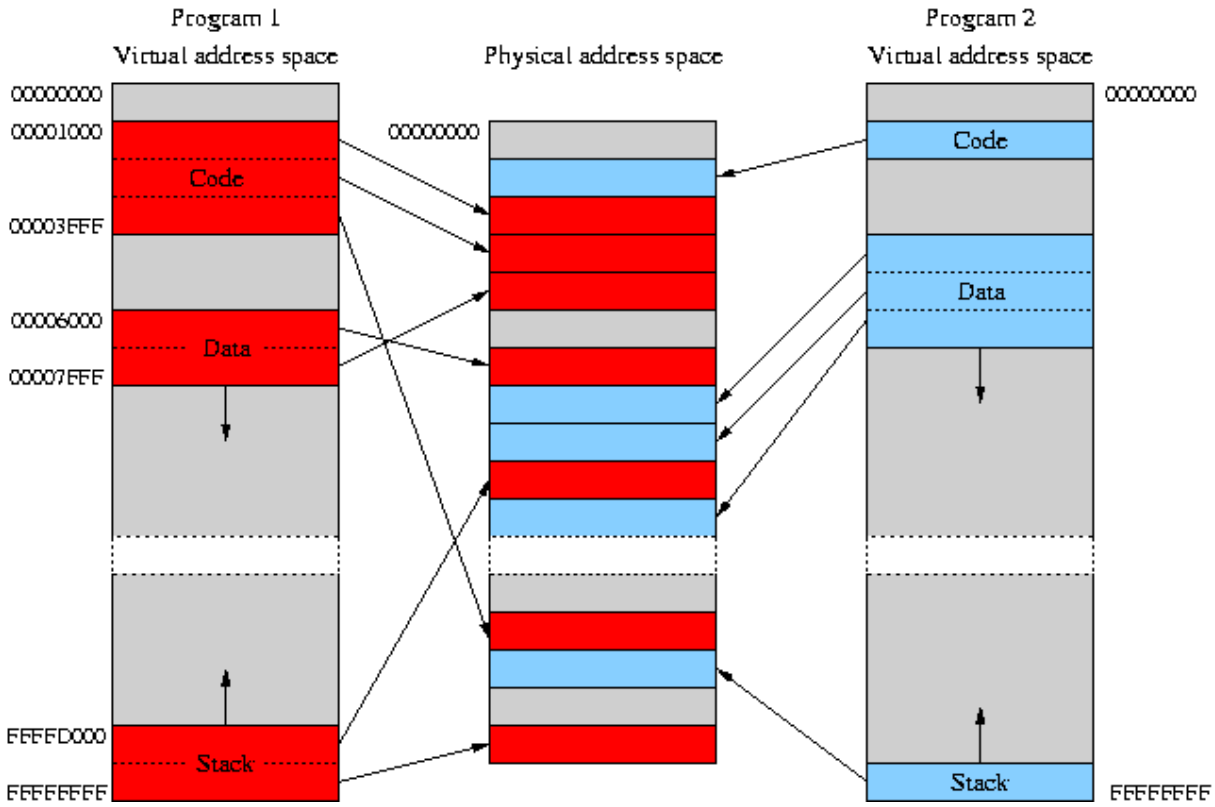
Lors de la compilation d'un programme écrit dans un langage de haut niveau, toutes les références à des variables locales ou globales sont transformées en des références à des emplacements mémoire alloués à ces variables. Les variables globales sont généralement allouées dans la zone des données alors que les variables locales sont plutôt allouées sur la pile. Les pointeurs de certains langages comme C ou C++ ou les références de Java sont aussi des adresses qui sont manipulées de manière explicites.

Dans un ordinateur sans mémoire virtuelle, les adresses manipulées implicitement ou explicitement par les programmes correspondent aux adresses réelles des données en mémoire. Le principe de *mémoire virtuelle* est de séparer les adresses manipulées par les programmes et les adresses réelles des données en mémoire. Les adresses manipulées par les programmes sont appelées *adresses virtuelles* et les adresses des données en mémoire sont appelées *adresses physiques*.

Chaque programme dispose d'un *espace d'adressage virtuel* constitué de toutes les adresses virtuelles. Pour que les programmes fonctionnent correctement, il doit y avoir une correspondance entre les adresses virtuelles et les adresses physiques. Cette correspondance est assurée conjointement d'une part au niveau matériel par un circuit adjoint au processeur appelé MMU (Memory Management Unit) et d'autre part au niveau logiciel par le système d'exploitation. Dans les micro-processeurs récents, ce circuit MMU est souvent intégré au processeur.

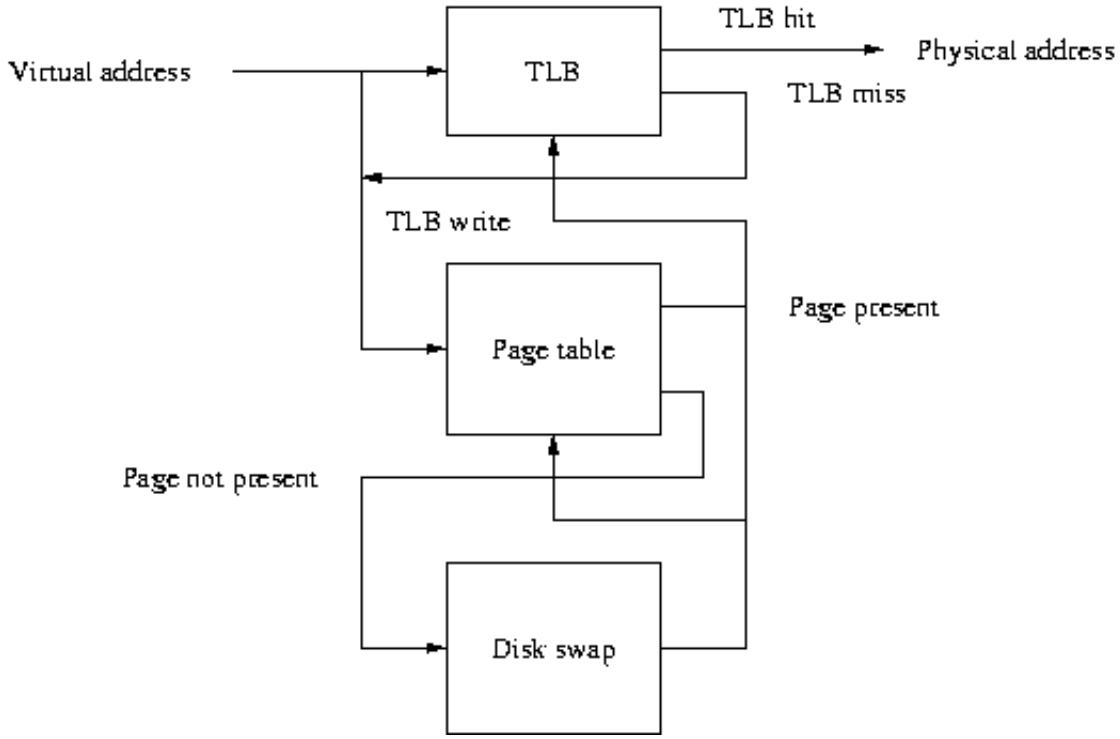
16.2 Fonctionnement

L'espace d'adressage virtuel de chaque programme ainsi que l'espace mémoire sont divisés en blocs appelés *pages* de même taille. Cette taille est très souvent de l'ordre 4 Ko mais peut aussi être de 8 ou 16 Ko sur des micro-processeurs 64 bits. Un bloc de l'espace d'adressage virtuel est appelé *page virtuelle* et un bloc de l'espace mémoire est appelé *page physique*. L'article wikipedia utilise les termes *page* et *frame*. Pour chaque page virtuelle utilisée par un programme correspond une page physique. Le système maintient pour chaque programme une table de correspondance entre les pages virtuelles et les pages physiques. Cette table est appelée *table des pages*. Elle est bien sûr propre à chaque programme. Deux programmes fonctionnant simultanément peuvent utiliser la même adresse virtuelle, par exemple 0x1234, pour des données différentes qui doivent donc se trouver à des adresses physiques différentes.



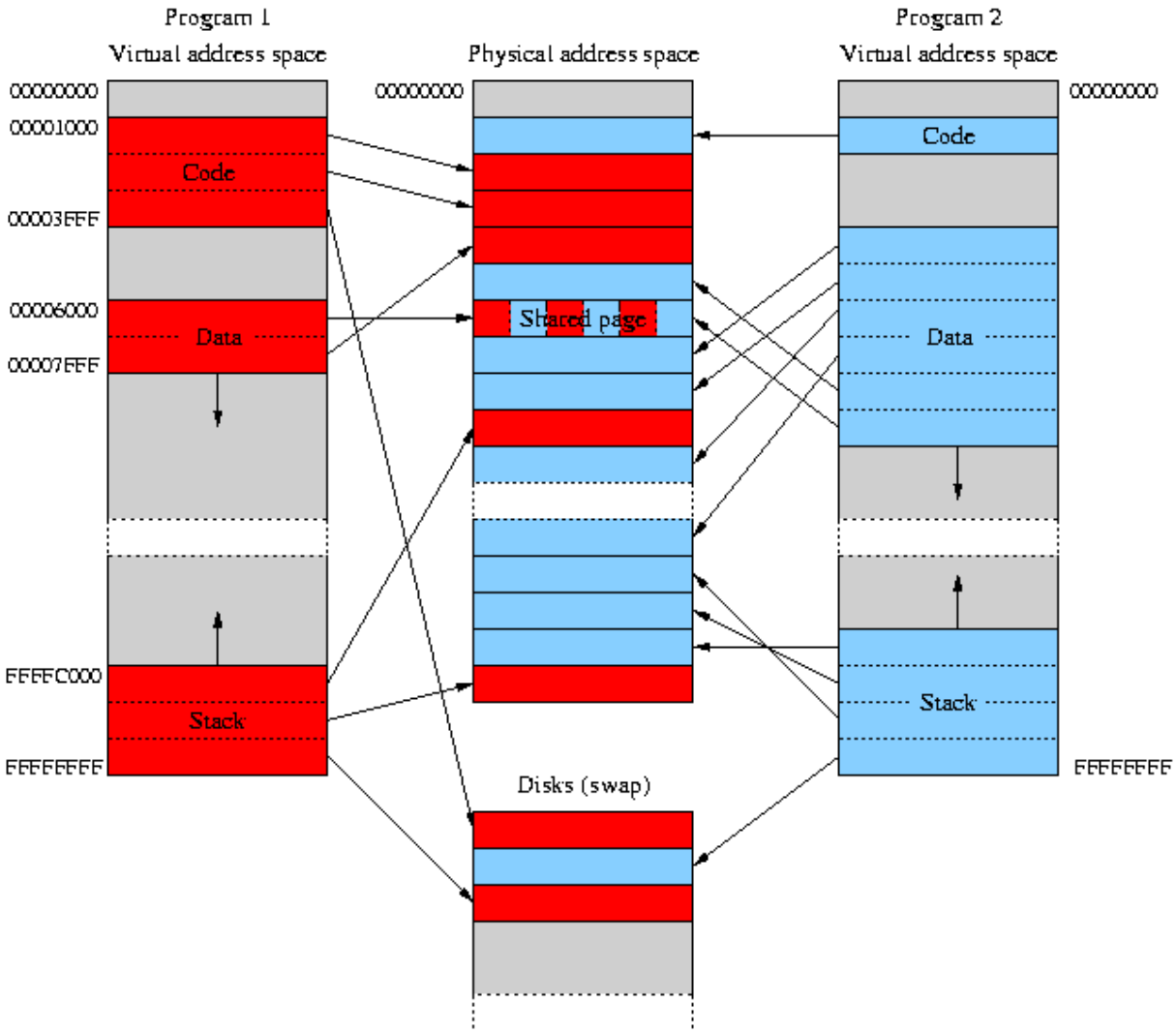
Principe de la mémoire virtuelle

16.3 Translation Lookaside Buffer



Fonctionnement du Translation Lookaside Buffer

16.4 Swap



Principe de la mémoire virtuelle avec swap

16.5 Table des pages hiérarchique

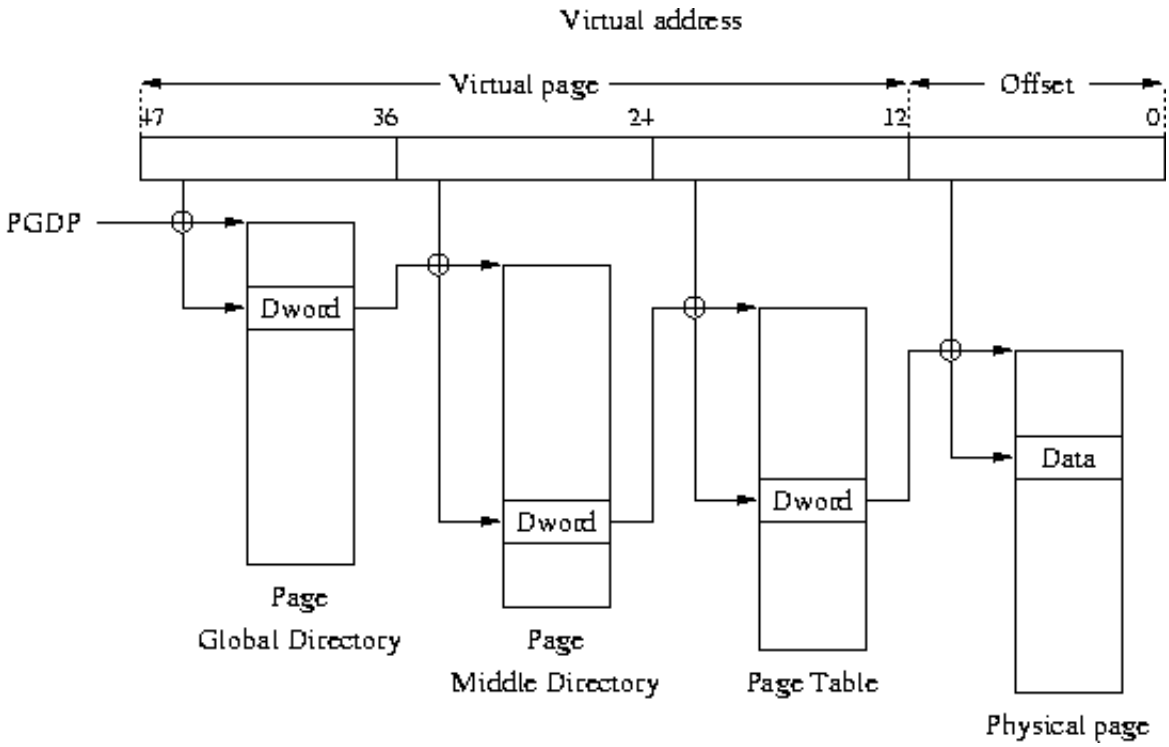
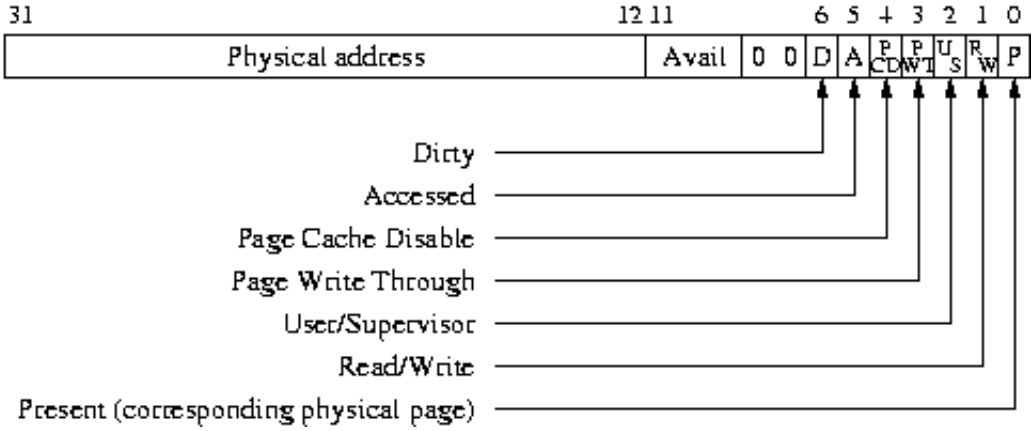


Table des pages hiérarchique



Structure des entrées (Dword) de la table des pages (80x86)

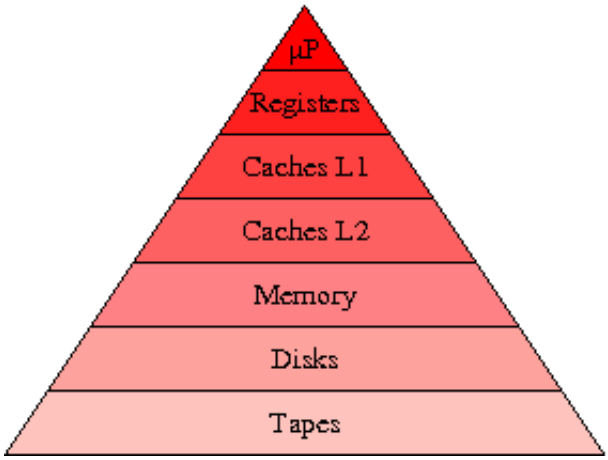
16.6 Linux

Cet article développe entre autre l'organisation propre à linux pour la mémoire virtuelle.

17 Caches mémoire

17.1 Hiérarchie de mémoire

Un ordinateur dispose de différents types de mémoires qui se distinguent par leurs vitesses d'accès. Plus la mémoire est rapide plus elle coûte cher et plus la quantité disponible sur l'ordinateur est réduite. Les registres internes entiers ou flottants du micro-processeur constituent la mémoire à laquelle celui-ci accède le plus rapidement mais ils sont en nombre très limité. Il y a ensuite la mémoire vive, les disques durs puis les bandes qui sont très lentes mais qui permettent de stocker des quantités très importantes de données. Les caches sont intermédiaires entre les registres internes du micro-processeur et la mémoires vive. Ils sont faits de mémoire rapide mais ils sont de taille réduite par rapport à la mémoire centrale.

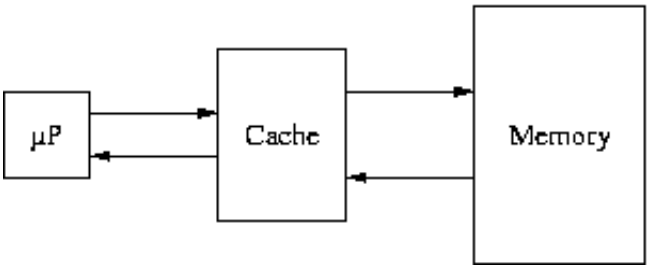


Hiérarchie de mémoire

17.2 Principe

Quand il est question de vitesse d'accès à la mémoire, il faut distinguer la *latence* et le *débit*. La latence est le temps qui s'écoule entre la demande des données et l'arrivée de la première donnée. Le débit mesure ensuite le flux de données transmises pendant le régime stable c'est-à-dire une fois la latence écoulée. Pour un disque dur, le temps de latence est relativement long car la tête de lecture doit être positionnée mécaniquement puis il faut encore attendre que le bon secteur se trouve sous la tête.

Depuis quelques temps, il s'est creusé un fossé entre la vitesse des micro-processeurs et la vitesse des mémoires dynamiques qui sont utilisées comme mémoire centrale des ordinateurs. Les mémoires SDRAM augmentent le débit mais réduisent peu la latence. Pour éviter que le micro-processeur perde du temps à attendre les données de la mémoire, des *caches mémoire* formés de mémoires statiques plus rapides sont intercalés entre le micro-processeur et la mémoire centrale. Le but est semblable à celui des caches disques permettant d'accélérer les accès aux disques durs.



Principe du cache

Le bon fonctionnement des caches est basé sur le *principe de localité* qui dit que le code et les données des programmes ne sont pas utilisées de manière uniforme. On constate souvent que 10% du code d'un programme contribue à 90% des instructions exécutées. On distingue deux types de localité. La *localité temporelle* indique que des éléments auxquels on a eu accès récemment seront probablement utilisés dans un futur proche. La *localité spatiale* indique que des éléments proches ont tendances à être référencés à des instants proches.

17.3 Fonctionnement

Le cache contient des copies de données qui sont en mémoire centrale. Avant tout accès à la mémoire, le processeur vérifie si les données ne sont pas présentes dans le cache. Auquel cas, le processeur utilise les données contenues dans le cache et n'accède pas à la mémoire. Sinon, il est nécessaire d'aller chercher les données en mémoire centrale.

17.3.1 Organisation

Le cache est organisé par lignes. Chaque ligne contient une portion de 8 à 512 octets des données en mémoires et une *étiquette* (*tag* en anglais) qui est l'adresse de ces données en mémoire. Lorsque le micro-processeur veut accéder à la mémoire, il compare l'adresse avec les étiquettes des lignes du cache. S'il trouve l'adresse parmi les étiquettes, le micro-processeur utilise directement les données du cache. On parle alors de *succès de cache*. Sinon on parle de *défaut de cache* ou d'*échec de cache* (*cache miss*).

Index	Tag	Dirty	Data
0	362F	0	4F ...
1	F3E0	0	00 ...
2	980A	1	F1 ...
3	8127	0	00 ...
4	A40B	1	56 ...
5	FFBC	1	90 ...
6	51DE	0	BC ...
7	0013	0	1A ...

Organisation du cache

17.3.2 Défauts de cache

La réponse à un défaut de cache dépend de la nature de l'accès aux données. Dans le cas d'une lecture, les données sont chargées de la mémoire centrale dans le cache puis envoyées au micro-processeur. Ce chargement nécessite de libérer au préalable une ligne du cache pour y placer les nouvelles données. Le choix de la ligne à libérer est contrôlé par la *politique de remplacement*. Le chargement dans le cache des données à partir de la mémoire centrale implique un délai puisque cette dernière est beaucoup plus lente que le cache. Ce délai est parfois augmenté par le fait qu'il faille copier en mémoire le contenu de la ligne libérée.

Dans le cas d'un défaut de cache lors d'une écriture, deux techniques sont utilisées. La première technique est de faire comme pour une lecture en chargeant les données dans le cache puis de laisser le processeur écrire dans le cache. La seconde technique est d'écrire directement les données en mémoire. L'idée est que lors d'une écriture, le processeur n'a pas besoin d'attendre que celle-ci soit terminée pour continuer à travailler. Dans ce cas, les écritures sont mises en attentes dans un buffer afin de permettre à ces écritures de s'effectuer sans ralentir le processeur même en cas de plusieurs écritures consécutives.

17.3.3 Politiques d'écriture

Il existe plusieurs façons appelées *politiques d'écriture* de gérer les écritures dans les caches. La politique appelée *write-through* consiste à répercuter en mémoire centrale chaque écriture dans le cache. Chaque écriture dans le cache provoque alors une écriture en mémoire centrale. À l'opposé, la politique *write-back* retarde au maximum les écritures en mémoire centrale. Les données qui ont été écrites dans le cache sont écrites en mémoire centrale au moment où la ligne qui contient ces données est libérée. Pour savoir si cette écriture est nécessaire, chaque ligne contient un bit appelé *dirty bit* qui indique s'il y a eu au moins une écriture dans cette ligne.

Il existe aussi des politiques intermédiaires. Dans le cas de la politique *write-through*, les écritures à faire peuvent être mises en attente temporairement dans une file. Plusieurs écritures consécutives à la même adresse peuvent ainsi être répercutées par une seule écriture en mémoire. Ce mécanisme réduit les échanges avec la mémoire centrale. Dans le cas d'une politique *write-back*, le cache peut anticiper l'écriture en mémoire de certaines lignes modifiées du cache. Il profite de périodes sans échange avec la mémoire pour écrire certaines lignes dont le *dirty bit* est positionné. Cette technique permet d'éviter l'écriture de la ligne au moment celle-ci est libérée.

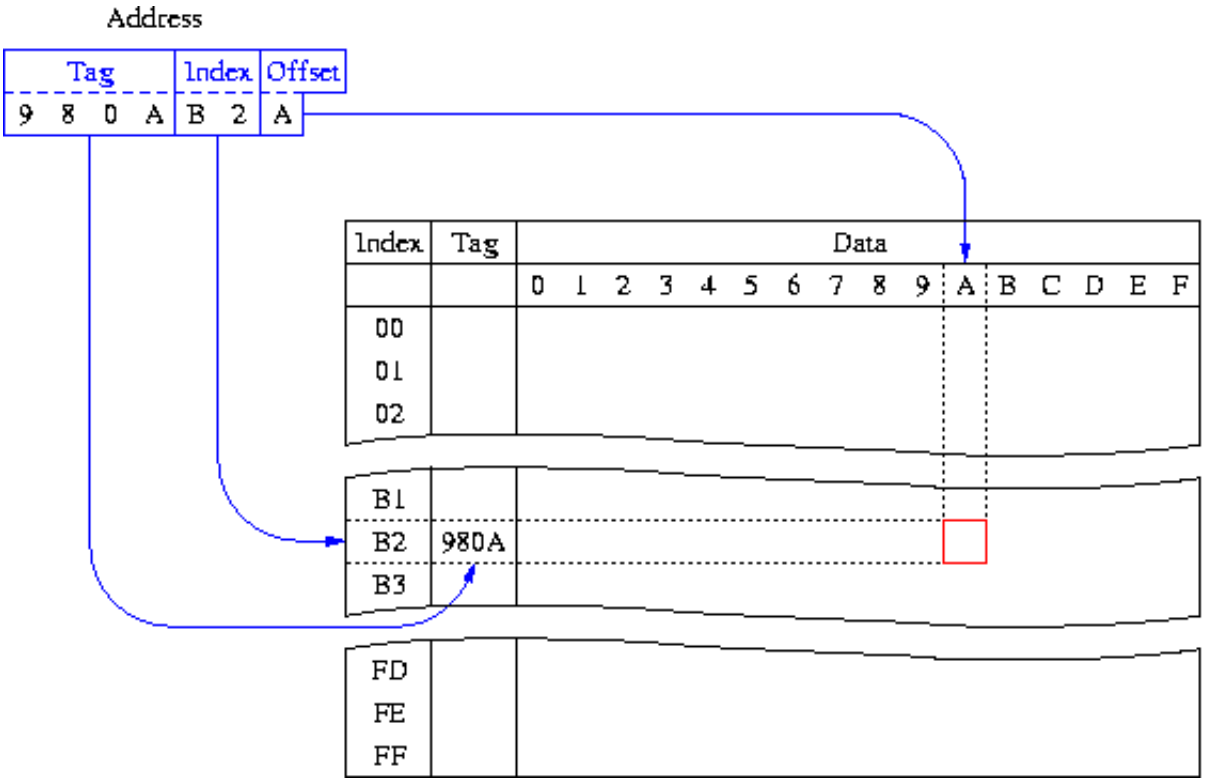
17.4 Associativité

Un élément crucial de l'efficacité du cache est de retrouver rapidement si des données à une adresse mémoire sont déjà dans le cache. Afin d'accélérer cette recherche, le nombre de lignes où peuvent être mises les données à une adresse mémoire fixée est souvent réduit. Ce nombre ne dépend pas de l'adresse et il est appelé l'*associativité* du cache. Lorsque ce nombre est réduit à 1, c'est-à-dire que les données de chaque adresse peuvent être mises dans une seule ligne du cache, on parle de *cache direct*. Si au contraire l'associativité est égale au nombre de ligne du cache, c'est-à-dire que chaque donnée peut être mise dans n'importe quelle ligne du cache, le cache est dit *complètement associatif*. Si l'associativité est un entier n, on parle de *cache n-associatif* (*n-way* en anglais). Les caches sont très souvent directs, 2-, 3- ou 4-associatifs mais rarement plus. Le cache de niveau 1 de l'Athlon est par exemple 2-associatif.

17.4.1 Caches directs

Dans le cas des caches directs, la ligne où sont placées les données à une adresse sont généralement déterminés par des bits de poids faible de l'adresse. Cette approche a plusieurs avantages. D'une part le numéro de la ligne est très facile à déterminer à partir de l'adresse. D'autre part, les bits utilisés pour déterminer la lignes n'ont pas à être stocker dans l'étiquette, ce qui offre un gains de quelques bits.

Soit un cache direct composé de 256 lignes contenant chacune 16 octets. Comme chaque ligne contient 16 octets, les 4 bits de poids faible de chaque adresse servent uniquement à donner la position (offset) des données dans la ligne. Comme le cache a 256 lignes, les 8 bits suivants déterminent la ligne où les données doivent être placées.



Cache direct

L'avantage des caches directs est de simplifier au maximum la recherche des données dans le cache. Il suffit en effet de comparer l'étiquette de la ligne correspondante avec une partie de l'adresse. Comme la ligne est unique, il est même possible de commencer la lecture du cache pendant la comparaison de l'étiquette avec l'adresse. Si cette comparaison révèle un défaut de cache, cette lecture anticipée du cache est annulée.

Les caches directs souffrent par contre d'un problème. Si un programme utilise simultanément deux parties de la mémoire qui doivent aller dans la même ligne de cache, il peut se produire de nombreux défauts de cache.

17.4.2 Caches n-associatifs

Dans le cas des caches n-associatifs, l'organisation du cache est similaires aux caches directs. À chaque adresse correspond n lignes consécutives du cache au lieu d'une. L'indice de la première ligne est encore déterminé par des bits de poids faible de l'adresse. On appelle généralement *ensemble* les blocs de mémoire ayant même indice.

Pour n égal à 2 ou 4, un cache n-associatif se révèle presqu'aussi performant qu'un cache direct ayant 2 ou 4 fois plus de mémoire. Par contre, au delà de 8, le cache est pénalisé par le temps pris par la recherche des données dans le cache puisqu'il faut comparer une partie de l'adresse avec n étiquettes. On remarque en outre que les caches complètement associatifs ont des performances comparables aux caches 8-associatifs.

17.5 Politiques de remplacement

Lorsque le cache n'est pas direct et que survient un défaut de cache, les données doivent être placées dans une des lignes du cache déterminées par l'adresse. Il reste alors à choisir quelle ligne libérer pour les nouvelles données. Il existe plusieurs façons de procéder à ce choix qui sont appelées *politiques de remplacement*. L'objectif de ces politiques est de minimiser le nombre de défauts de cache, en essayant de prévoir au mieux les données qui seront utilisées par le micro-processeur.

Ces différentes politiques de remplacement sont bien sûr un compromis entre leur performance et le surcoût de calculs qu'elles impliquent. Comme ces calculs doivent être réalisés au niveau des circuits du cache, ils sont nécessairement simples. Les politiques favorisent la libération d'une ligne où il n'y a eu aucune écriture. Ce choix économise l'écriture en mémoire centrale des données contenues dans la ligne libérée.

Les deux politiques les plus couramment utilisées sont le tirage aléatoire et la politique dite du *moins récemment utilisé* (Least Recently Used). Le tirage aléatoire consiste à choisir au hasard une des lignes possibles. L'avantage de cette politique est de minimiser le surcoût de calcul tout en donnant des performances raisonnables. La seconde politique choisit la ligne à laquelle le dernier accès est le plus ancien. L'idée sous-jacente est que cette ligne a une probabilité plus faible d'être utilisée à l'avenir. En particulier, les données qui ne sont plus utilisées par le micro-processeur disparaissent du cache. En fait, ce sont le plus souvent des approximations de cette politique qui sont utilisées en pratique.

17.6 Défauts de cache

Lorsqu'il y a trop de défauts de cache, les performances s'effondrent. Le cache fonctionne alors alors à la vitesse de la mémoire centrale ou même plus lentement en raison des surcoûts de traitement. Beaucoup de recherche a été consacré à la réduction du nombre de défauts de cache. Les défauts de caches sont généralement classés en trois catégories.

défauts de première référence (compulsory misses)
: À la première référence à un bloc, celui-ci doit être chargé dans le cache. Ces défaut sont en quelque sorte inévitables.

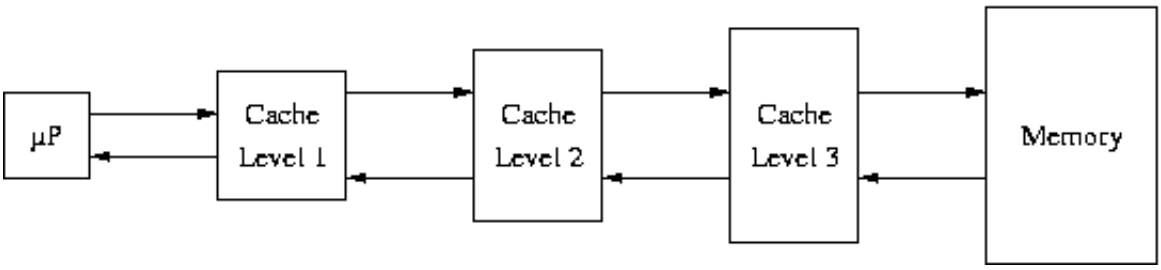
défauts de capacité (capacity misses)
: Ces défauts sont dû au fait que le cache ne peut pas contenir tous les blocs référencés pendant l'exécution du programme. Le nombre de ces défauts peut être réduit en augmentant la taille du cache.

défauts de conflit (conflict misses)

Ces défauts interviennent en plus des deux précédents types. Un bloc a pu être chargé puis enlevé du cache car d'autres blocs avec le même indice ont été chargés. Le nombre de ces défauts peut être réduit en augmentant l'associativité du cache.

17.7 Hiérarchie de cache

Pour un meilleur compromis entre le prix de la mémoire cache et son efficacité, on utilise plusieurs niveaux de cache. On parle alors de *hiérarchie de cache*. La plupart des ordinateurs performants utilisent 3 à 4 niveaux de cache. Le cache de niveau 1 est le plus près du micro-processeur. Il est petit mais fait de mémoire très rapide. Le cache de niveau 2 est de taille plus grande mais fait de mémoire moins rapide. Le cache de niveau 3 est encore plus grand mais fait de mémoire encore mois rapide. Le cache de niveau 1 est directement intégré au processeur. Le cache de niveau 2 est souvent dans le même boîtier que le micro-processeur ce qui permet un large bus entre les deux. Le bus de niveau 3 est en général sur la carte mère du processeur.

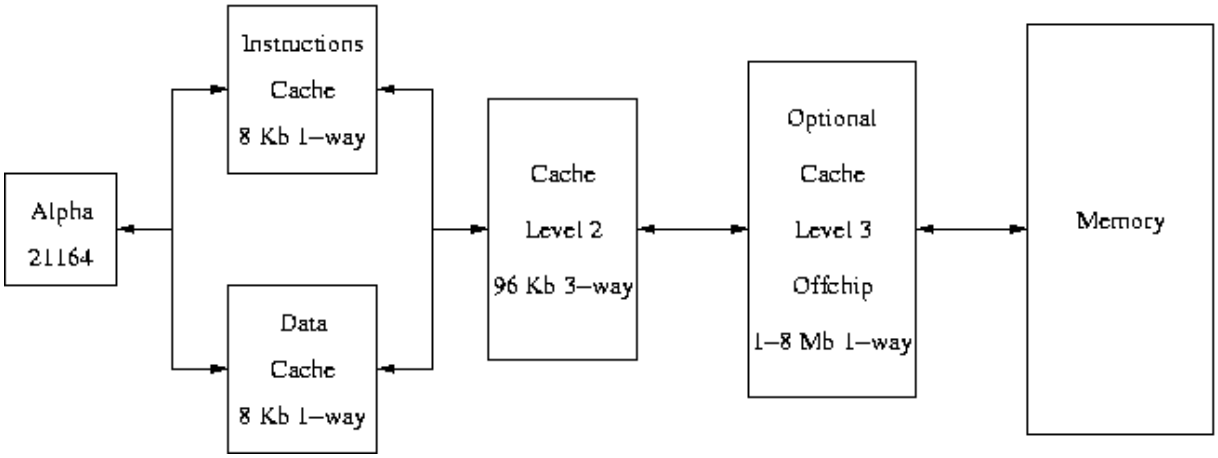


Hiérarchie de caches

17.8 Séparation des caches

Le cache de niveau 1 est souvent organisé en deux caches distincts, un pour les instructions et un autre pour les données. Deux bus permettent alors au micro-processeur d'accéder simultanément aux deux caches. Dans le cas d'une architecture avec un pipeline, cette organisation est indispensable pour éviter certains aléas structurels. En outre, les accès aux instructions ou aux données se comportent de manière assez différente. La séparation des caches autorise des optimisations différentes pour les deux caches.

La figure ci-dessous présente l'architecture à trois niveaux de caches du micro-processeur Alpha 21164.



Hiérarchie de caches du micro-processeur Alpha 21164

18 Bibliographie

1. J. L. Hennessy and D. A. Patterson, *Architecture des machines*. Thomson Publishing, 1996, ISBN 2-84180-022-9.
2. W. Stallings, *Organisation et architectures des machines*. Pearson Education, 2003, ISBN 2-7440-7007-6.
3. Y. N. Patt and S. J. Patel, *Introduction to computing systems: from bits and gates to C and beyond*. McGraw Hill, 2004, ISBN 0-07-121503-4.
4. A. Tanenbaum, *Architecture de l'ordinateur*. Pearson Education, 2005, ISBN 2-7440-7122-6.
5. B. Goossens, *Architecture et micro-architecture des processeurs*. Springer, 2002, ISBN 2-287-59761-1.
6. D. Harris and S. Harris, *Digital design and computer architecture*. Morgan Kaufmann Publishers, 2007, ISBN 0-123704979.

Table des matières