INTRODUCTION A L'ARCHITECTURE DES ORDINATEURS

Jean-Christophe BUISSON

octobre 2020

Chapitre I. Principes généraux

I.1. Organisation générale d'un ordinateur

I.1.1. Le modèle de Von Neumann

La plupart des ordinateurs ont l'organisation générale de la Figure I-1.

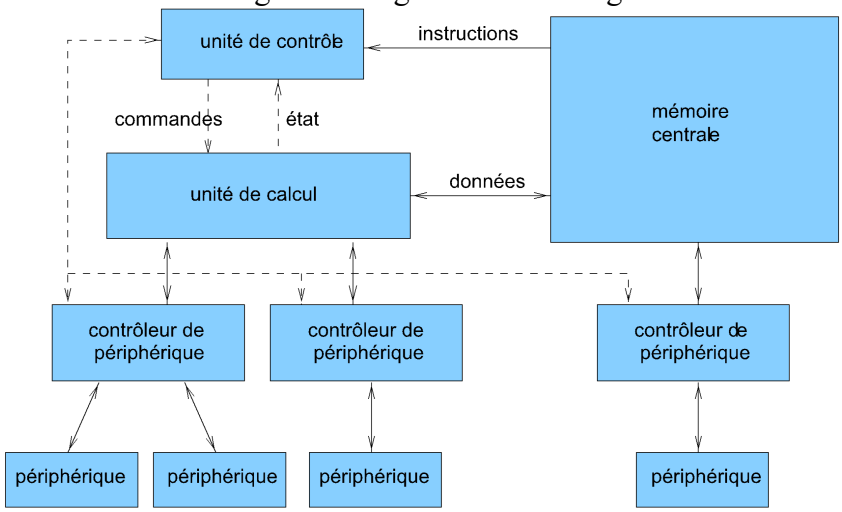


Figure I-1. Organisation générale d'un ordinateur; les flèches sont des chemins de données.

Cette organisation est dite de 'Von-Neumann', car elle ressemble beaucoup à la machine conçue par ***John Von Neumann*** sur la machine IAS peu après la deuxième guerre mondiale. Elle suppose et implique un certain nombre de propriétés caractéristiques des ordinateurs actuels :

La mémoire est un composant central ; c'est un tableau de mots de taille fixe

On voit sur le schéma que la ***mémoire centrale*** est au départ et à l'arrivée de l'exécution d'un programme. C'est elle qui contient le programme à exécuter et les données initiales ; c'est dans elle que seront stockés les résultats intermédiaires ou finaux.
C'est un tableau de mots binaires de taille fixe. Chaque mot est repéré par un nombre appelé adresse, et les programmes feront référence aux données et aux instructions en mémoire par leurs adresses.

Le programme exécuté est dans la mémoire centrale

Même si le programme qu'exécute un ordinateur est stocké initialement sur un disque dur, ce n'est pas avant d'avoir été copié en mémoire centrale, que son exécution peut commencer.
Un programme qu'exécute un ordinateur est composé ***d'instructions*** sous forme de mots mémoire. Le langage utilisé pour coder les instructions en mots binaires est appelé ***langage machine***, c'est le seul que comprenne le processeur. La taille des instructions varie beaucoup selon le type des processeurs. Parfois toutes les instructions ont la même taille, égale à un ou plusieurs mots mémoire; parfois elles ont des tailles variables selon la complexité de l'instruction.

Le processeur est le coordonnateur de l'exécution

Le ***processeur*** contrôle l'exécution des programmes, en organisant le chargement des instructions et le détail de leur exécution ; c'est aussi lui qui effectue tous les calculs,

notamment sur les nombres entiers. Ces deux fonctions sont associées dans la même unité car le contrôle de l'exécution nécessite souvent des calculs entiers, lorsque par exemple il faut faire une somme pour déterminer l'adresse de l'instruction suivante.
Le processeur a besoin de garder la trace d'un certain nombre d'informations pendant l'exécution des instructions : l'adresse de l'instruction en cours par exemple, ou le résultat de la dernière opération arithmétique. Il utilise pour cela des ***registres***, qui sont des mots mémoire fixes, à accès très rapide.

Le processeur ne dialogue pas directement avec les périphériques

On voit sur la figure que le processeur dialogue avec des contrôleurs de périphériques et non avec les périphériques eux-mêmes. Par exemple un processeur n'a aucune conscience de l'existence d'un disque dur ; il dialogue avec un contrôleur de disque, à l'aide d'instructions génériques d'entrées/sorties.

I.1.2. Organisation en bus

Les flèches entre les composants de la Figure I-1 étaient à prendre dans un sens fonctionnel; si elles devaient représenter effectivement les chemins par lesquels transitent les données, le fonctionnement serait très inefficace. En pratique, les données circulent entre les différents sous-systèmes d'un ordinateur sur des ***bus***, sortes d'autoroutes, et des circuits spécialisés jouent le rôle d'échangeurs aux carrefours de ces bus, de façon à permettre des transferts simultanés dans certaines circonstances. La Figure I-2 montre une organisation typique d'un système de type compatible PC.

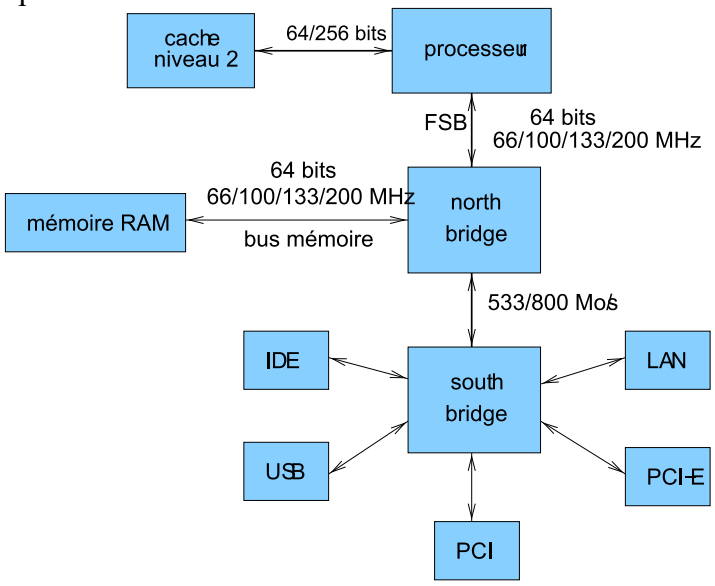


Figure I-2. Architecture générale des bus d'un ordinateur compatible PC.

Un circuit appelé ***north-bridge*** règle les échanges à très grande vitesse, notamment entre le processeur et la mémoire centrale. Il exploite de façon synchrone le bus FSB (***front side bus***), encore appelé ***bus système***, dont la vitesse est très corrélée à la puissance générale de l'ensemble. Un second circuit appelé ***south-bridge*** effectue quant à lui des échanges moins rapides avec des périphériques SATA, réseau, etc.
La Figure I-3 montre une photo de ***carte mère*** typique, avec l'emplacement physique des différents bus. La mémoire et le north bridge sont placés très près du processeur, car ils sont sur les bus les plus rapides. Le north-bridge est équipé d'un radiateur à cause de l'intensité du travail de transfert qu'il réalise.

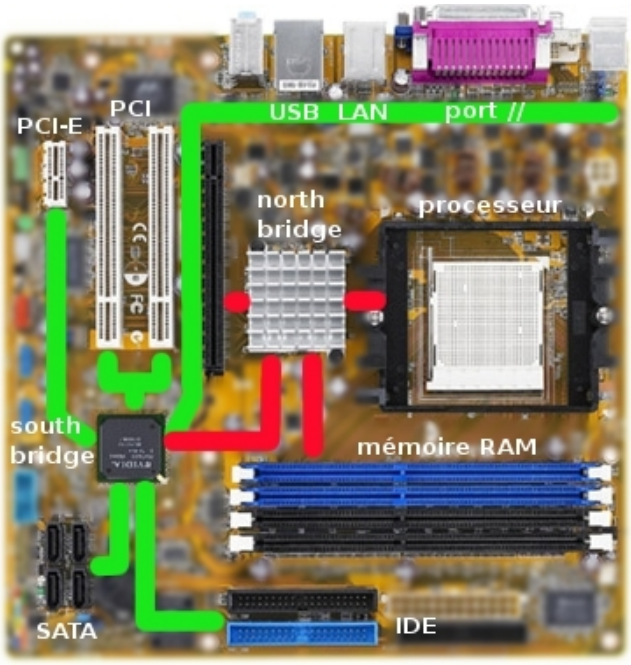


Figure I-3. Carte mère typique. Les bus rouges (FSB et mémoire), très rapides, sont gérés par le north bridge ; les bus verts (PCI, SATA, etc.), plus lents, sont gérés par le south bridge.

I.2. Bits et signaux électriques

L'information est stockée et véhiculée dans un ordinateur, logiquement sous forme de chiffres binaires ou ***bits*** (binary digit), et physiquement sous forme de signaux électriques.
La logique binaire est utilisée parce qu'il est plus facile de construire des systèmes électriques possédant deux états stables, et parce qu'on a maintenant beaucoup de résultats mathématiques en ***algèbre de Boole***, mais des implémentations sur des systèmes possédant plus de deux états stables sont envisageables.
Ces deux états sont notés '0' et '1' ; ils correspondent le plus souvent aux deux tensions électriques 0v et +Vcc respectivement, Vcc valant exactement +5v pour des circuits utilisant la technologie TTL, et une valeur comprise entre 2v et 15v environ en technologie CMOS (des valeurs de Vcc=3.3v, 2.5v et 1.8v étant utilisée de plus en plus souvent). Pour tenir compte du fait que ces signaux électriques vont être déformés lors des transmissions, des plages de valeurs plus larges ont été affectées à ces états. Par exemple, en technologie TTL, elles sont représentées Figure I-4. Il faut noter que ces assignations sont légèrement différentes selon qu'une tension est considérée en entrée ou en sortie d'un circuit.

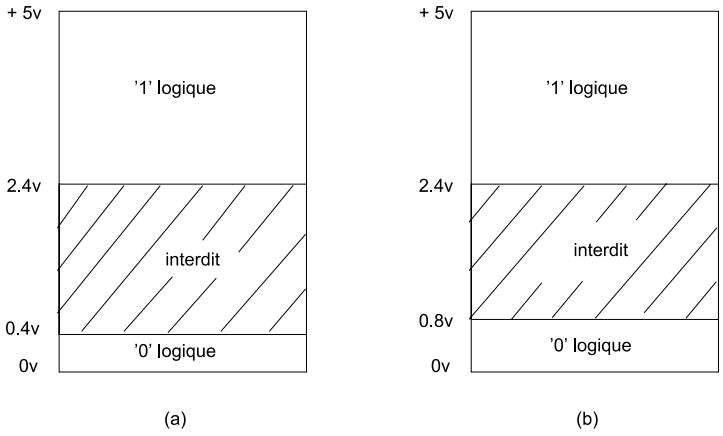


Figure I-4. Assignation des tensions électriques aux valeurs logiques (a) en entrée de circuit, (b) en sortie de circuit (technologie TTL).

Les circuits logiques qui sont utilisés pour combiner et transmettre les signaux électriques logiques ont des caractéristiques non linéaires qui ont entre autres pour fonction de remettre en forme un signal déformé par le bruit. On voit Figure I-5 une porte 'non' qui inverse le bit d'entrée, tout en produisant un signal de sortie de meilleure qualité que le signal d'entrée.

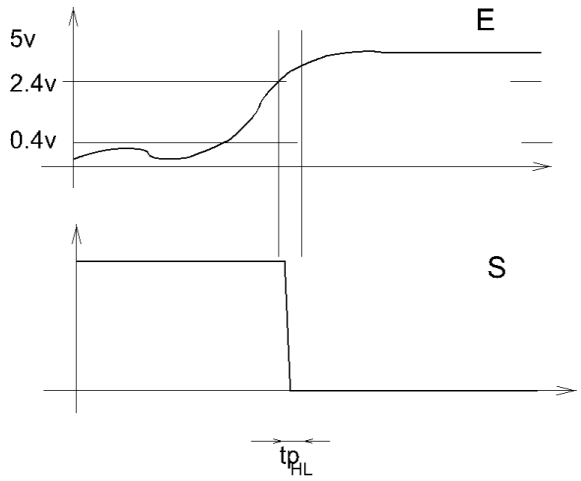


Figure I-5. Remise en forme d'un signal après traversée d'une porte 'non'.

Les signaux pourront donc traverser un nombre quelconque d'étages de circuits logiques sans que les déformations initiales ne soient amplifiées.
On notera sur la Figure I-5 la présence d'un ***temps de propagation*** tp_{HL} (HL indiquant : de haut vers bas) qui correspond à la durée que met le circuit pour calculer sa sortie après que ses entrées se soient modifiées. Ces temps de propagation se cumulent lors de la traversée de plusieurs étages, et c'est finalement eux qui limiteront la vitesse de fonctionnement maximale d'un circuit.

I.3. Technologies de fabrication des circuits intégrés

Les premiers ordinateurs ont été construits durant la deuxième guerre mondiale avec des tubes cathodiques (des 'lampes') comme dispositif de commutation. En 1947, les laboratoires Bell inventent le premier transistor, qui va très vite supplanter la lampe et apporter un facteur de réduction de taille important. Mais la véritable révolution n'interviendra qu'avec l'invention des circuits intégrés VLSI dans les années 1970, qui amèneront à la réalisation des premiers microprocesseurs.
Un grand nombre de technologies de fabrication de circuits intégrés existent, qui se distinguent par des qualités différentes en termes de vitesse de propagation, densité d'intégration, consommation et dissipation thermique. On va ébaucher rapidement dans cette section les trois familles les plus connues, en terminant par celle qui est actuellement la plus répandue et la plus significative, la famille CMOS.

Technologie TTL

La technologie TTL était la technologie standard utilisée à partir des années 1960 pour la réalisation de tous les types d'ordinateurs. Elle s'alimente typiquement en +5V, a une consommation modérée, est robuste et a des règles d'interface simples. Une très grande famille de circuits spécialisés, la famille 7400, est disponible sous forme de circuits en boîtier DIL, et fournit des modules tout prêts : portes, bascules, compteurs, etc. Les capacités d'intégration de circuits TTL sont faibles, et on ne peut guère mettre plus de quelques milliers de portes TTL sur une puce de circuit intégré. On l'utilise encore aujourd'hui pour des petites réalisations, notamment dans l'interfaçage avec d'autres circuits.

Technologie ECL

La technologie ECL est caractérisée par sa très grande rapidité, mais aussi par une consommation de courant élevée. Elle est de plus assez difficile à utiliser, notamment en raison de l'emploi de tensions d'alimentation négatives. Elle est utilisée dans des petites réalisations où la vitesse est critique, mais certains experts prévoient un développement de la technologie ECL avec l'utilisation de nouveaux types de semi-conducteurs.

Technologie CMOS

C'est la technologie reine actuelle. La technologie CMOS a d'abord été développée et vendue comme une alternative au TTL, plus lente mais avec une consommation réduite. Comme par ailleurs elle peut utiliser des tensions d'alimentation faibles (jusqu'à 1V), elle a immédiatement été très utilisée par les fabricants de montres digitales, pour qui la vitesse de traitement importait peu par rapport à la consommation. Mais on a progressivement réalisé que sa plus grande qualité était son taux d'intégration très élevé ; de plus, des tailles de transistors de plus en plus petites ont amenées avec elles des temps de commutation de plus en plus faibles. C'est la technologie CMOS qui est utilisée actuellement pour la fabrication de tous les processeurs et microcontrôleurs du marché, ainsi que pour toutes les mémoires statiques et les mémoires flash.
Par ailleurs, un circuit CMOS ne consomme significativement du courant que lors des commutations des transistors. Cette propriété a été à l'origine de circuits qui consomment moins de courant que leur équivalent TTL, qui eux en consomment au repos et lors des commutations. Néanmoins, au fur et à mesure qu'on diminue leur tension d'alimentation, les courants de fuite des transistors CMOS deviennent de plus en plus proches des courants de commutation, et par conséquent la consommation de ces circuits au repos n'est plus aussi négligeable qu'auparavant.

Chapitre II. Éléments de logique combinatoire

II.1. Circuits combinatoires

En 1854, Georges Boole publia son ouvrage séminal sur une algèbre manipulant des informations factuelles vraies ou fausses. Son travail a été redécouvert et développé sous la forme que nous connaissons maintenant par Shannon. Le nom de Boole est entré dans le vocabulaire courant, avec l'adjectif booléen qui désigne ce qui ne peut prendre que deux valeurs distinctes 'vrai'ou 'faux'.

Circuits combinatoires : définition

Un circuit combinatoire est un module tel que l'état des sorties ne dépend que de l'état des entrées. L'état des sorties doit être évalué une fois que les entrées sont stables, et après avoir attendu un temps suffisant à la propagation des signaux. On comprend intuitivement que les circuits combinatoires correspondent à des circuits sans état interne : face à la même situation (les mêmes entrées), ils produisent toujours les mêmes résultats (les mêmes sorties).
Un module d'addition en est un bon exemple : il a en entrées les signaux A et B à additionner, ainsi qu'une retenue CIN d'un étage précédent ; il fournit en sortie la somme S et une retenue COUT pour un étage suivant :

Figure II-1. Un additionneur est un circuit combinatoire

Si un circuit combinatoire est composé lui-même de sous-circuits combinatoires, on démontre que la définition de base est équivalente au fait que ces sous-circuits sont connectés entre eux sans rebouclages. Plus précisément, si on range les signaux de gauche à droite, en mettant tout a fait à gauche les entrées, puis dans chaque colonne les sous-circuits qui ont des entrées parmi les signaux produits dans la colonne de gauche, ou de colonnes plus à gauche, alors le circuit est combinatoire si et seulement s'il n'y a aucune liaison qui reboucle 'en arrière', en allant d'un étage à un étage plus à gauche (Figure II-2).
L'implication dans le sens 'sans rebouclage' → 'combinatoire' est facile à démontrer : si les entrées à gauche sont stables, alors par construction toutes les sorties des sous-circuits de la première colonne sont stables, et ne dépendent que de ces entrées. Donc les sorties des sous-circuits de la colonne 2 sont stables, et ne dépendent que des entrées. On démontre ainsi par récurrence que les sorties des circuits de chaque colonne sont stables et ne dépendent que des valeurs des entrées, jusqu'aux sorties finales. Le circuit est donc combinatoire.

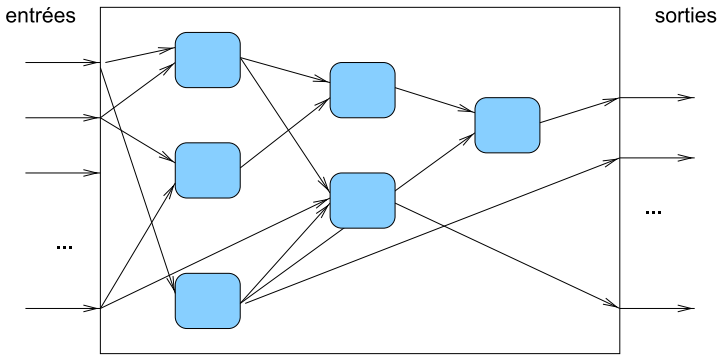


Figure II-2. Circuit combinatoire : les signaux internes et les sorties ne rebouclent pas en arrière.

Les figures II.3 et II.4 montrent des exemples de circuits combinatoires et non combinatoires formés à partir de portes logiques, dont on verra dans les sous-sections suivantes qu'elles sont elles-mêmes combinatoires.

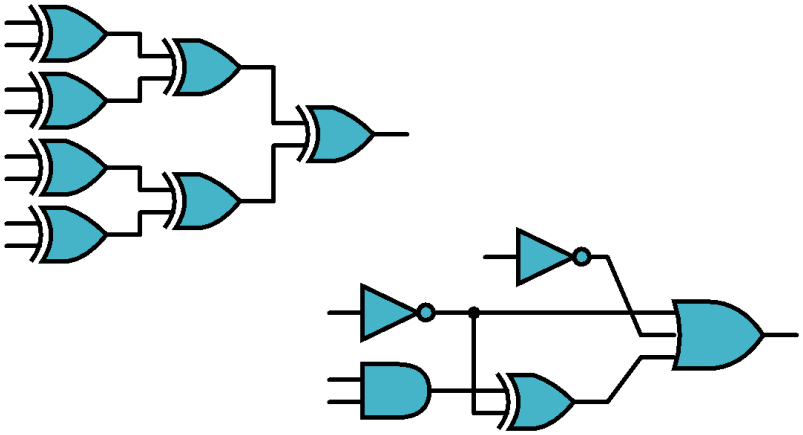


Figure II-3. Exemples de circuits combinatoires : les éléments internes sont eux-mêmes combinatoires, et il n'y a pas de rebouclage interne.

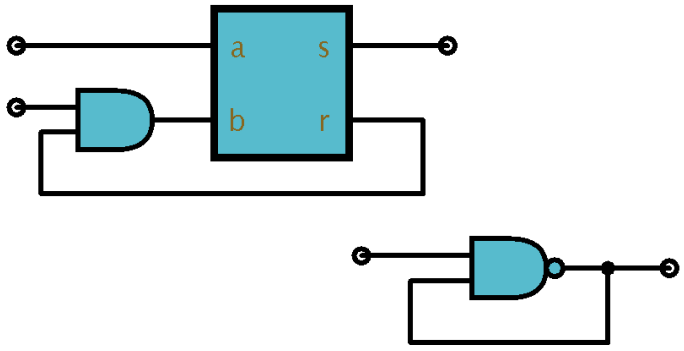
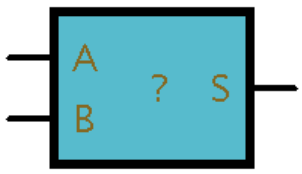


Figure II-4. Exemples de circuits non combinatoires : il y a des rebouclages internes.

II.2. Tables de vérité

Une des contributions essentielles de Boole est la table de vérité, qui permet de capturer les relations logiques entre les entrées et les sorties d'un circuit combinatoire sous une forme tabulaire.
Considérons par exemple un circuit inconnu possédant 2 entrées A et B et une sortie S.
Nous pouvons analyser exhaustivement ce circuit en lui présentant les 22 = 4 jeux d'entrées différents, et en mesurant à chaque fois l'état de la sortie. Le tout est consigné dans un tableau qu'on appelle table de vérité du circuit (figure II.5).



A	B	S
0	0	0
0	1	0
1	0	0
1	1	1

Figure II-5. Un circuit à deux entrées et une sortie, et sa table de vérité.

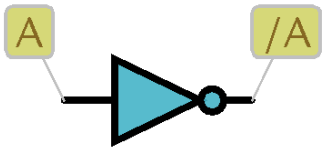
Ici, on reconnaît l'opération logique ET : S vaut 1 si et seulement si A et B valent 1.
La construction d'une table de vérité est bien sûr généralisable à un nombre quelconque n d'entrées et un nombre m de sorties. Elle possédera alors 2n lignes, ce qui n'est praticable que pour une valeur de n assez petite.

II.3. Algèbre et opérateurs de base

Plutôt que de décrire en extension la relation entre les entrées et une sortie, on peut la décrire en intention à l'aide d'une formule de calcul utilisant seulement trois opérateurs booléens : NON, ET, OU.

NON

NON (NOT en anglais) est un opérateur unaire, qui inverse son argument. On notera généralement $NOT(A) = \bar{A}$; et sa table de vérité est :



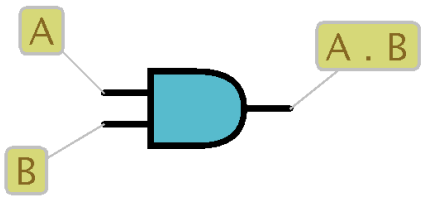
A	$\bar{A}$
0	1
1	0

Figure II-6. Table de vérité du NON,et dessin de la porte correspondante.

On a bien sûr : $\overline{\bar{A}} = A$.

ET

ET (AND en anglais) est un opérateur à 2 entrées ou plus, dont la sortie vaut 1 si et seulement si toutes ses entrées valent 1. On le note algébriquement comme un produit, c'est à dire $S = A \times B$ ou $S = AB$. La table de vérité d'un ET à deux entrées, et le dessin usuel de la porte correspondante sont représentés figure II.7.



A	B	$A.B$
0	0	0
0	1	0
1	0	0
1	1	1

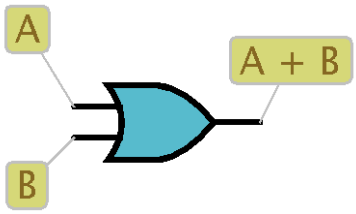
Figure II-7. Table de vérité du ET, et symbole de la porte correspondante.

De par sa définition, le ET est associatif : $A \,.\, B \,.\, C = (A \,.\, B) \,.\, C = A \,.\, (B \,.\, C)$. Il est également idempotent: $A \,.\, A = A$.

OU

OU (OR en anglais) est un opérateur à 2 entrées ou plus, dont la sortie vaut 1 si et seulement si une de ses entrées vaut 1. On le note algébriquement comme une somme, c'est à dire S = A

+ B. La table de vérité d'un OU à deux entrées, et le dessin usuel de la porte correspondante sont représentés figure II.8.



A	B	$A + B$
0	0	0
0	1	1
1	0	1
1	1	1

Figure II-8. Table de vérité du OU, et symbole de la porte correspondante.

De par sa définition, le OU est associatif : $A + B + C = (A + B) + C = A + (B + C)$. Il est également idempotent : $A + A = A$.

De la table de vérité à la formule algébrique

On peut automatiquement écrire la formule algébrique d'une fonction booléenne combinatoire dès lors qu'on a sa table de vérité. Il suffit de considérer seulement les lignes qui donnent une sortie égale à 1, d'écrire le minterm correspondant, qui code sous forme d'un ET la combinaison de ces entrées. Par exemple, le minterm associé au vecteur (A, B, C) = (1, 0, 1) est : $A\overline{B}C$. La formule algébrique qui décrit toute la fonction est le OU de tous ces minterms. Considérons par exemple la table de vérité de la fonction majorité à 3 entrées, qui vaut 1 lorsqu'une majorité de ses entrées (2 ou 3) vaut 1 (figure II.9).

A	B	C	S
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Figure II-9. Table de vérité de la fonction majorité à 3 entrées.

La table de vérité contient 4 lignes avec une sortie S à 1, ce qui donne :

$S = \overline{A}BC + A\overline{B}C + AB\overline{C} + ABC$

Cette formule est simplifiable, comme on le verra dans une section ultérieure.

Est-on sûr que la formule obtenue par cette méthode est correcte ? Pour une combinaison des entrées qui doit donner une sortie à 1, la formule possède le minterm correspondant, et donne aussi une valeur de 1. Pour toutes les autres combinaisons d'entrées, la table donne une sortie à 0, et la formule aussi, puisqu'aucun minterm n'est présent qui corresponde à ces combinaisons. La formule donne donc toujours le même résultat que la table.

Corollairement, cela démontre le résultat fondamental suivant :

> N'importe quelle fonction combinatoire s'exprime sous forme d'une somme de minterms.

Théorèmes de l'algèbre de Boole

Le tableau de la figure II.10 résume les principaux théorèmes et propriétés de **l'algèbre de Boole**. On peut les démontrer de façon algébrique, ou en utilisant des tables de vérité.
Les **théorèmes de De Morgan** permettent de transformer les ET en OU et vice-versa.
Le couple (ET, NON) ou le couple (OU, NON) suffisent donc à exprimer n'importe quelle formule algébrique combinatoire.
Le théorème d'absorption $A + AB = A$ montre qu'un terme plus spécifique disparaît au profit d'un terme moins spécifique. Par exemple dans l'équation $\ldots + \overline{A}B + \overline{A}B\overline{C} + \ldots$, le terme $\overline{A}B\overline{C}$ s'efface au profit de $\overline{A}B$, moins spécifique et qui donc l'inclut.

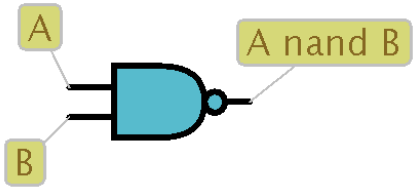
relation	relation duale	Propriété
$AB = BA$	$A + B = B + A$	Commutativité
$A.(B + C) = A.B + A.C$	$A + B.C = (A + B).(A + C)$	Distributivité
$1.A = A$	$0 + A = A$	Identité
$A.\overline{A} = 0$	$A + \overline{A} = 1$	Complément
$0.A = 0$	$1 + A = 1$	Zéro et un
$A.A = A$	$A + A = A$	Idempotence
$A.(B.C) = (A.B).C$	$A + (B + C) = (A + B) + C$	Associativité
$\overline{\overline{A}} = A$		Involution
$\overline{AB} = \overline{A} + \overline{B}$	$\overline{A + B} = \overline{A}.\overline{B}$	Théorèmes de De Morgan
$A.(A + B) = A$	$A + A.B = A$	Théorèmes d'absorption
$A + \overline{A}.B = A + B$	$A.(\overline{A} + B) = A.B$	Théorèmes d'absorption

Figure II-10. Propriétés et théorèmes de l'algèbre de Boole.

II.4. Autres portes logiques

NAND

NAND (= NOT AND) est un opérateur à 2 entrées ou plus, dont la sortie vaut 0 si et seulement si toutes ses entrées valent 1. On le note $\uparrow$, et on a donc à deux entrées : $A \uparrow B = \overline{A.B}$
La table de vérité d'un NAND à deux entrées, et le dessin usuel de la porte correspondante sont représentés figure II.11.



A	B	$A \uparrow B$
0	0	1
0	1	1
1	0	1
1	1	0

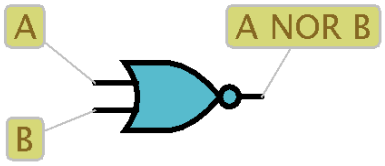
Figure II-11. Table de vérité du NAND, et symbole de la porte correspondante.

Le NAND est un opérateur dit **complet**, c'est à dire qu'il permet à lui seul d'exprimer n'importe quelle fonction combinatoire. Il permet en effet de former un NOT, en reliant ses

deux entrées : $\overline{A} = A \uparrow B$ Les théorèmes de De Morgan assurent donc qu'il peut exprimer un ET et un OU, et donc n'importe quelle expression combinatoire.

NOR

NOR (= NOT OR) est un opérateur à 2 entrées ou plus, dont la sortie vaut 0 si et seulement au moins une de ses entrées 1. On le note $\downarrow$, et on a donc à deux entrées : $A \downarrow B = \overline{A + B}$. La table de vérité d'un NOR à deux entrées, et le dessin usuel de la porte correspondante sont représentés figure II.12.



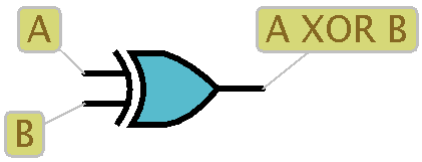
A	B	$A \downarrow B$
0	0	1
0	1	0
1	0	0
1	1	0

Figure II-12. Table de vérité du NOR, et symbole de la porte correspondante.

Comme le NAND, le NOR est également un opérateur **complet**. Il permet en effet de former un NOT, en reliant ses deux entrées : : $\overline{A} = A \downarrow B$. Les théorèmes de De Morgan assurent donc qu'il peut exprimer un ET et un OU, et donc n'importe quelle expression combinatoire.

XOR

XOR (= EXCLUSIVE OR) est un opérateur à 2 entrées ou plus. On peut le définir de plusieurs façons ; la définition qui permet le plus directement de démontrer ses propriétés est celle qui consiste à en faire un détecteur d'imparité : sa sortie vaut 1 si et seulement si un nombre impair de ses entrées est à 1. On le note $\oplus$; la table de vérité d'un XOR à deux entrées, et le dessin usuel de la porte correspondante sont représentés figure II.13.



A	B	$A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

Figure II-13. Table de vérité du XOR, et symbole de la porte correspondante.

On voit à partir de la table que $A \oplus B = \overline{A}B + A\overline{B}$. Lorsqu'il a deux entrées, il mérite son nom de OU exclusif puisque sa sortie ne vaut 1 que si l'une de ses entrées vaut 1, mais seulement une. Il possède plusieurs autres propriétés intéressantes :

- lorsqu'on inverse une entrée quelconque d'un XOR, sa sortie s'inverse. C'est évident puisque cela incrémente ou décrémente le nombre d'entrées à 1, et donc inverse la parité.
- le XOR à deux entrées est un inverseur commandé. En effet, en examinant la table de vérité du XOR (figure II.13), on constate que lorsque A vaut 0, la sortie S reproduit l'entrée B, et lorsque A vaut 1, la sortie S reproduit l'inverse de l'entrée B. L'entrée A peut être alors vue comme une commande d'inversion dans le passage de B vers S. Cette fonction sera très utile lorsqu'on souhaitera corriger une erreur détectée sur une ligne, par exemple.
- le XOR est un opérateur d'addition arithmétique. Cette propriété est valable quel que soit le nombre d'entrées du XOR. Bien sûr, il n'est question ici que du bit de poids faible du résultat (ajouter n termes de 1 bit produit un résultat sur log2(n) bits). Et en effet, si on regarde la table de vérité du XOR à deux entrées, on constate que la sortie est bien la somme arithmétique des deux entrées. Bien sûr à la dernière ligne, le résultat à une retenue que le XOR à lui seul ne peut pas produire.

On peut démontrer que ce résultat est général par une simple récurrence. Si en effet on suppose que $XOR(e_1, e_2, \dots, e_n)$ fournit le bit de poids faible de la somme $s_n = e_1 + e_2 + \cdots + e_n$ alors :

$XOR(e_1, e_2, \dots, e_n, e_{n+1}) = s_n$ si $e_{n+1} = 0$; $\overline{s_n}$ si $e_{n+1} = 1$

$= s_n$ avec inversion commandée par e_{n+1}

$= s_n \oplus e_{n+1}$

- XOR est un opérateur associatif. Sa définition même, en tant qu'indicateur d'imparité, assure cette propriété. Quelle que soit la façon de grouper les termes, le calcul donnera le même résultat. Ce résultat permet de construire avec un minimum de circuits des XORs à 3 entrées ou plus, en cascadant des XORs à 2 entrées.

Formule algébrique d'un XOR à plus de 2 entrées

Considérons le calcul du XOR de trois variables A, B et C. On sait que son expression algébrique peut se mettre sous forme d'une somme de minterms ; on va donc chercher parmi tous les minterms possibles ceux qu'il faut inclure dans la formule, et ceux qu'il faut rejeter. Avec ces trois variables A, B et C, il y a 8 minterms possibles : $\bar{A}\bar{B}\bar{C}$, $\bar{A}\bar{B}C$, $\bar{A}B\bar{C}$,…, $\bar{A}\bar{B}\bar{C}$ ne doit pas être inclus dans le résultat, puisqu'il donne 1 pour le vecteur (0,0,0) qui ne comporte pas un nombre impair de 1. $\bar{A}\bar{B}C$ doit l'être, car il donne 1 pour le vecteur (0,0,1) qui possède un nombre impair (1) de 1. La règle est simple : on doit inclure seulement les minterms qui ont un nombre impair de variables non barrées. Pour le XOR à trois entrées, on peut le faire de façon organisée en mettant d'abord les 3 minterms ayant une variable non barrée, puis le minterm ayant les 3 variables non barrées :

$$XOR(A, B, C) = A\bar{B}\bar{C} + \bar{A}B\bar{C} + \bar{A}\bar{B}C + ABC$$

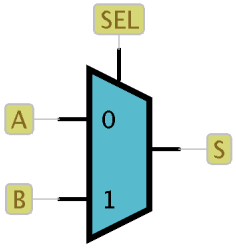
On peut montrer que cette formule n'est pas simplifiable en tant que somme de termes.
Pour un XOR à 4 entrées, on inclut les 4 minterms ayant 1 variable non barrée, puis les 4 minterms ayant 3 variables non barrées :

$$XOR(A, B, C, D) = A\bar{B}\bar{C}\bar{D} + \bar{A}B\bar{C}\bar{D} + \bar{A}\bar{B}C\bar{D} + \bar{A}\bar{B}\bar{C}D + ABC\bar{D} + AB\bar{C}D + A\bar{B}CD + \bar{A}BCD$$

Pour un nombre quelconque d'entrées, on peut montrer que le résultat comporte toujours la moitié de tous les minterms possibles, et que cette somme n'est jamais simplifiable en tant que somme de termes.

Le multiplexeur

Le multiplexeur est une porte à trois entrées, qui joue un rôle d'aiguillage. Son dessin indique bien cette fonction (figure II.14).



SEL	*S*
0	*A*
1	*B*

Figure II-14. Table de vérité condensée du multiplexeur, et dessin de la porte correspondante.

Lorsque la commande de sélection vaut 0, l'aiguillage est du côté de A, et S prend la valeur de A ; lorsqu'elle vaut 1, S prend la valeur de B. Algébriquement, on a donc :

$$S = \overline{SEL} \cdot A + SEL \cdot B$$

À partir de la compréhension du rôle d'aiguillage du multiplexeur, il est clair que si on inverse les deux entrées, la sortie sera inversée également :

$$\bar{S} = \overline{SEL} \cdot \bar{A} + SEL \cdot \bar{B}$$

Or ce n'est pas si facile à établir algébriquement :

$$\bar{S} = \overline{\overline{SEL} \cdot A + SEL \cdot B} = \overline{\overline{SEL} \cdot A} \cdot \overline{SEL \cdot B}$$

$$\bar{S} = (SEL + \bar{A}) \cdot (\overline{SEL} + \bar{B}) = \bar{A} \cdot \overline{SEL} + SEL \cdot \bar{B} + \bar{A} \cdot \bar{B}$$

On a un terme en trop, $\bar{A} \cdot \bar{B}$! En fait, on peut le faire disparaître en écrivant $\bar{A} \cdot \bar{B} = \bar{A} \cdot \bar{B} \cdot SEL + \bar{A} \cdot \bar{B} \cdot \overline{SEL}$:

$$\bar{S} = \bar{A} \cdot \overline{SEL} + SEL \cdot \bar{B} + \bar{A} \cdot \bar{B} \cdot SEL + \bar{A} \cdot \bar{B} \cdot \overline{SEL}$$

$$\bar{S} = \bar{A} \cdot \overline{SEL}(1 + \bar{B}) + SEL \cdot \bar{B}(1 + \bar{A}) = \overline{SEL} \cdot \bar{A} + SEL \cdot \bar{B}$$

On a ici un exemple où la compréhension d'une fonction a permis de se dispenser de calculs algébriques ou de méthodes de simplification.

II.5. Méthodes de simplification des fonctions combinatoires

II.5.1. Tables de Karnaugh

Une ***table de Karnaugh*** est utilisée pour trouver visuellement les simplifications à faire sur une somme de termes. Elle est très facile à utiliser, pour peu qu'il n'y ait pas plus de 4 variables en jeu. Au delà, il vaut mieux employer la méthode de Quine-Mc Cluskey décrite plus bas.

Considérons par exemple la fonction ayant la table de vérité de la figure II.17.

A	B	C	D	S
0	0	0	0	0
0	0	0	1	**1**
0	0	1	0	**1**
0	0	1	1	**1**
0	1	0	0	0
0	1	0	1	**1**
0	1	1	0	0
0	1	1	1	**1**
1	0	0	0	0
1	0	0	1	**1**
1	0	1	0	0
1	0	1	1	1
1	1	0	0	**1**
1	1	0	1	**1**
1	1	1	0	**1**
1	1	1	1	**1**

Figure II-15. Table de vérité d'une fonction à simplifier

On va représenter les valeurs de S dans un tableau 4x4, chaque case correspondant à un des 16 minterms possibles avec 4 variables A, B, C et D (figure II.18).

On remarquera d'abord les libellés des lignes et des colonnes, qui forment un ***code de Gray*** : d'une ligne à l'autre ou d'une colonne à l'autre, il n'y a qu'un bit d'entrée qui change.

La table a une forme de cylindre dans les deux directions, c'est à dire que les colonnes '00' et '10' doivent être considérées comme côte à côte, tout comme les lignes '00' et '10'.

CD \ AB	0,0	0,1	1,1	1,0
0,0			1	
0,1	1	1	1	1
1,1	1	1	1	1
1,0	1		1	

Figure II-16. Table de Karnaugh représentant la table de vérité.

Les 11 sorties '1' de la table de vérité ont été reportées dans la table de Karnaugh. En effectuant des groupes de '1' connexes de 2, 4 ou 8, on réalise une simplification (figure II-17).

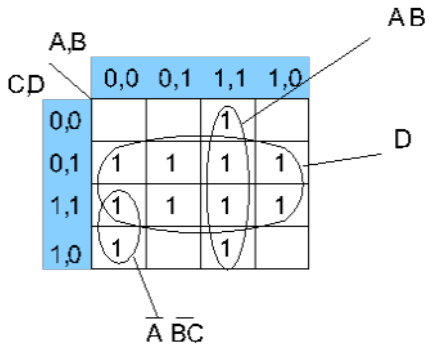


Figure II-17. Table de Karnaugh avec les regroupements.

Par exemple le regroupement $\bar{A}\bar{B}C$ correspond à la simplification $\bar{A}\bar{B}CD + \bar{A}\bar{B}C\bar{D} = \bar{A}\bar{B}C(D + \bar{D}) = \bar{A}\bar{B}C$. De même, le regroupement AB correspond à la simplification des 4 termes : $AB\bar{C}\bar{D} + AB\bar{C}D + ABC\bar{D} + ABCD = AB(\bar{C}\bar{D} + \bar{C}D + C\bar{D} + CD) = AB$. On repère facilement le résultat en regardant les libellés des lignes et des colonnes impliqués dans la simplification. Pour le terme simplifié D par exemple, il concerne les lignes CD=01 ;11 et les colonnes AB=**; seul D = 1 est constant, et toutes les autres variables disparaissent.
Finalement on a :

$$S = D + AB + \bar{A}\bar{B}C$$

Utilisation des combinaisons non spécifiées

Il y a des situations dans lesquelles certaines combinaisons des entrées ne se produisent jamais, qu'on peut appeler combinaisons interdites ou combinaisons non spécifiées.
Considérons par exemple un décodeur 7 segments, qui convertit un nombre binaire sur 4 bits en un vecteur de 7 bits qui commande un afficheur à Leds de type 'calculette' (Figure II-18).

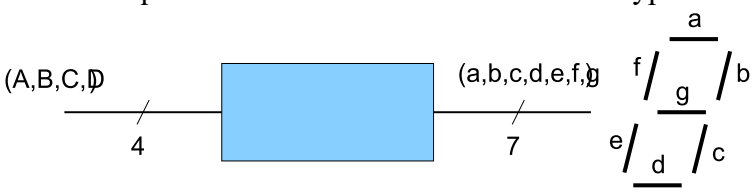


Figure II-18. Convertisseur 7 segments pour afficheur de calculette.

La spécification de ce convertisseur nous précise qu'il ne faut convertir que les chiffres de 0 à 9, c'est à dire les valeurs du vecteur (A, B, C, D) de $(0,0,0,0)$ à $(1,0,0,1)$. Ainsi le fonctionnement du circuit n'est pas spécifié pour les valeurs de $(1,0,1,0)$ à $(1,1,1,1)$, et on va exploiter cette latitude pour simplifier les équations.
Si on considère par exemple la table de vérité du segment a, elle comporte ainsi 6 lignes pour lesquelles la valeur de la sortie n'est pas spécifiée, qu'on note '*' (Figure II-19).

A	B	C	D	a
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	*
1	0	1	1	*
1	1	0	0	*
1	1	0	1	*
1	1	1	0	*
1	1	1	1	*

Figure II-19. Table de vérité du segment a.

Dans la table de Karnaugh correspondante, les '*' peuvent être incluses dans les regroupements (Figure II-20) ce qui conduit à leur donner la valeur '1'. Bien sûr on ne doit pas chercher à inclure toutes les '*' dans les regroupements : on cherche à inclure tous les '1', mais les '*'permettent de trouver des regroupements plus grands. Ici par exemple, on trouve le résultat très simple :

$$a = C + A + BD + \bar{B}\bar{D}$$

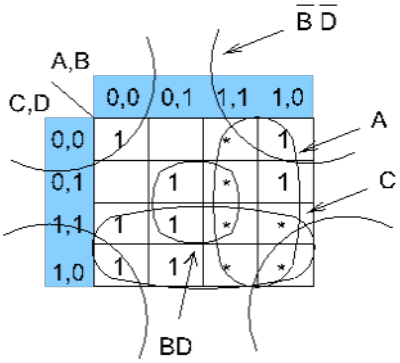


Figure II-20. Table de Karnaugh du segment a, avec les combinaisons non spécifiées marquées par des '*'. On peut inclure les '*' pour former des regroupements plus grands.

Détection et correction des glitches

Un autre usage des tables de Karnaugh est la détection des glitchs, c'est-à-dire des changements très brefs des signaux de sortie d'une fonction combinatoire lors de changements des entrées. Considérons par exemple la fonction suivante :

$$S = \bar{A}\bar{B}\bar{C}\bar{D} + \bar{A}B\bar{C}\bar{D} + AB\bar{C}\bar{D} + AB\bar{C}D + ABC\bar{D} + ABCD$$

On construit directement la table de Karnaugh associée à cette fonction, et on trouve immédiatement deux regroupements (Figure II-21).

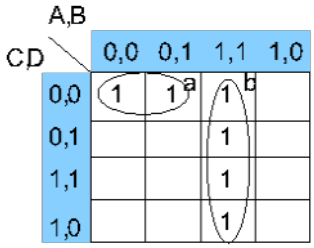


Figure II-21. Table de Karnaugh avec les regroupements. Un glitch est possible dans le passage de a à b, car les deux regroupements sont tangents à cet endroit.

L'équation simplifiée est alors :

$$S = AB + \bar{A}\bar{C}\bar{D}$$

Cette équation est plus simple que la première, mais elle va se comporter légèrement différemment lors de certains changements de valeurs des entrées. Considérons par exemple le changement du vecteur d'entrées (A, B, C, D) de $(0,1,0,0)$ à $(1,1,0,0)$, qui correspond au passage de a à b sur la Figure II-21. Dans ce cas, seul A a changé, et la sortie S devrait rester à 1, aussi bien pour l'équation initiale que pour l'équation simplifiée. Or pour l'équation simplifiée, le terme AB va passer de 0 à 1 tandis que le terme $\bar{A}\bar{C}\bar{D}$ va passer de 1 à 0 ; si ce changement se fait exactement en même temps, S restera à 1, mais si le premier terme passe à 0 légèrement avant que le deuxième ne passe à 1, un léger glitch sur S apparaîtra (Figure II-22).

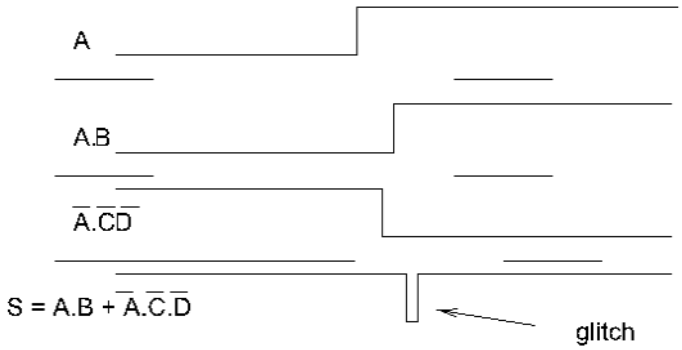


Figure II-22. Glitch sur l'équation $S = AB + \bar{A}\bar{C}\bar{D}$, lors du passage de (A, B, C, D) de $(0, 1, 0, 0)$ à $(1, 1, 0, 0)$.

Ce glitch était prévisible ***visuellement*** sur la table de Karnaugh, par le fait des deux regroupements qui sont adjacents sans se recouvrir. On constate également que c'est le seul changement d'une seule variable qui peut conduire à un tel problème. La solution est de rajouter un regroupement qui va faire la liaison entre ces deux groupes, en valant 1 pendant toute la transition (Figure II-23).

Finalement, on obtient l'expression simplifiée sans glitch :

$$S = AB + \bar{A}\bar{C}\bar{D} + B\bar{C}\bar{D}$$

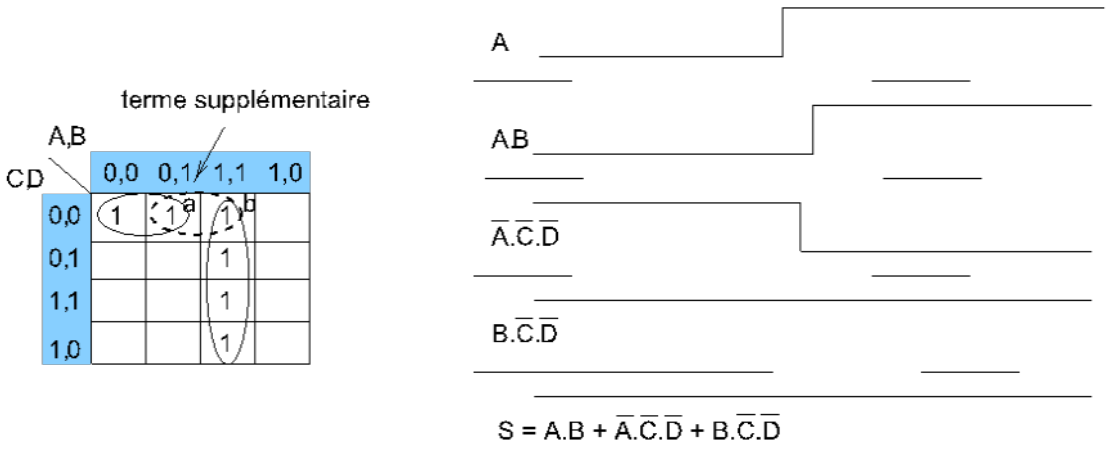


Figure II-23. Ajout du terme $B\bar{C}\bar{D}$ pour maintenir S à 1 lors de la transition de a vers b. Le glitch disparaît sur le signal de sortie S.

II.5.2. Méthode de Quine-Mc Cluskey

On va montrer son usage en simplifiant la fonction à 5 variables suivante :

$$f\ (A,B,C,D,E) = \bar{A}\bar{B}\bar{C}\bar{D}\bar{E} + \bar{A}\bar{B}\bar{C}D\bar{E} + \bar{A}B\bar{C}\bar{D}\bar{E} + \bar{A}B\bar{C}D\bar{E} + \bar{A}BC\bar{D}\bar{E} + \bar{A}BC\bar{D}E + AB\bar{C}D\bar{E} + ABC\bar{D}E + ABCD\bar{E}$$

Une table de Karnaugh serait délicate à utiliser dans ce cas, car elle aurait une taille de 4x8, et certains regroupements possibles pourraient ne pas être connexes.
Par économie d'écriture, on commence par représenter chaque minterm par le nombre associé à sa représentation binaire :

$$f\ (A,B,C,D,E) = \sum(0,2,8,10,12,13,26,29,30)$$

On place ensuite ces minterms dans un tableau, en les groupant selon le nombre de 1 qu'ils possèdent (Figure II-24).

nombre de 1	minterm	valeur binaire
0	0	00000
1	2	00010
	8	01000
2	10	01010
	12	01100
3	13	01101
	26	11010
4	29	11101
	30	11110

Figure II-24. On classe les minterms selon le nombre de 1 de leur valeur binaire.

Les simplifications ne peuvent se produire qu'entre groupes adjacents. La simplification entre les minterms 1 et 2 se notera par exemple : 000-0, en mettant un tiret à l'endroit où la variable a disparu. Lorsqu'on a effectué toutes les simplifications possibles, on recommence à partir des nouveaux groupes formés, jusqu'à ce qu'aucune simplification ne soit possible (Figure II-25).

étape 1			étape 2			étape 3	
0	00000	✓	0-2	000-0	✓	0-2-8-10	0-0-0
2	00010	✓	0-8	0-000	✓		
8	01000	✓	2-10	0-010	✓		
10	01010	✓	8-10	010-0	✓		
12	01100	✓	8-12	01-00			
13	01101	✓	10-26	-1010			
26	11010	✓	12-13	0110-			
29	11101	✓	13-29	-1101			
30	11110	✓	26-30	11-10			

Figure II-25. On simplifie entre groupes adjacents, étape par étape. Les termes qui ont été utilisés dans une simplification sont cochés.

Maintenant, il suffit de choisir dans ce tableau les termes qui vont inclure tous les minterms de départ, en essayant de trouver ceux qui correspondent à l'expression la plus simple. On peut le faire de façon purement intuitive, en commençant par les termes les plus à droite : par exemple les groupes 0-2-8-10, 8-12, 13-29, 26-30 vont inclure tous les minterms.
On trouve donc :

$$S = \bar{A}\bar{C}\bar{E} + \bar{A}B\bar{D}\bar{E} + BC\bar{D}E + AB\bar{D}\bar{E}$$

Dans les cas complexes, ou si on veut programmer cet algorithme, on énumère la liste des *implicants premiers*, qui sont les termes du tableau précédent qui n'ont été utilisés dans aucune simplification. Dans le tableau précédent, on a coché avec une '3' les termes qui ont été utilisés dans une simplification, donc la liste des implicants premiers est : 8-12, 10-26, 12-13, 13-29, 26-30, 0-2-8-10. Il est clair que le résultat simplifié ne comportera que des termes de cette liste, et le but est de déterminer parmi eux ceux qu'il est indispensable de placer dans le résultat,appelés *implicants premiers essentiels*. On construit pour cela la table des implicants premiers (Figure II-26).

			0	2	8	10	12	13	26	29	30
8-12	01-00				✓		✓				
10-26	-1010					✓			✓		
12-13	0110-						✓	✓			
13-29	-1101	*						✓		✓	
26-30	11-10	*							✓		✓
0-2-8-10	0-0-0	*	✓	✓	✓	✓					

Figure II-26. Table des implicants premiers ; les implicants premiers sont en lignes et les minterms de départ sont en colonnes. Un implicant premier est dit essentiel s'il est le seul à couvrir un des minterms en colonne ; on le repère avec une marque '*'.

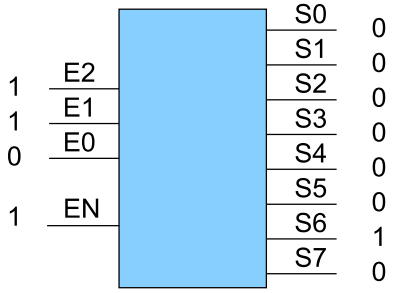
Parmi les 5 minterms placés en ligne dans le tableau, 3 sont dits essentiels, parce qu'ils sont les seuls à couvrir un des minterms placés en colonnes : l'implicant premier 13-29 est le seul à couvrir le minterm 29, 26-30 est le seul à couvrir 26 et 30, et 0-2-8-10 est le seul à couvrir les minterms 0 et 2. 8-12 et 12-13 ne sont pas essentiels, car 8, 12 et 13 sont couverts par d'autres implicants premiers.

Les implicants premiers essentiels sont nécessaires, mais pas forcement suffisants. Ici par exemple, si on ne prenait qu'eux, le minterm 12 ne serait pas couvert. Il reste donc une phase heuristique de choix parmi les implicants premiers non essentiels pour couvrir tous les minterms non encore couverts. Dans notre exemple, on peut ajouter, soit 8-12, soit 12-13 pour couvrir le 12 manquant, ce qui donne les deux possibilités suivantes :

- version avec 8-12 : $S = \bar{A}C\bar{E} + \bar{A}B\bar{D}\bar{E} + BC\bar{D}E + AB\bar{D}\bar{E}$
- version avec 12-13 : $S = A\bar{C}\bar{E} + \bar{A}BC\bar{D} + BC\bar{D}E + AB\bar{D}\bar{E}$

II.6. Décodeurs

Un décodeur est un circuit possédant n entrées et 2^n sorties numérotées (Figure II-27).
À tout moment, une et une seule sortie est active : celle dont le numéro correspond à la valeur binaire présente sur les n entrées. Le décodeur traduit donc la valeur d'entrée en une information de position spatiale.




```
module decodeur(e[2..0], en :
s[7..0])
    s[7] = en*e[2]*e[1]*e[0]
    s[6] = en*e[2]*e[1]*/e[0]
    s[5] = en*e[2]*/e[1]*e[0]
    s[4] = en*e[2]*/e[1]*/e[0]
    s[3] = en*/e[2]*e[1]*e[0]
    s[2] = en*/e[2]*e[1]*/e[0]
    s[1] = en*/e[2]*/e[1]*e[0]
    s[1] = en*/e[2]*/e[1]*/e[0]
end module
```

Figure II-27. Décodeur 3 vers 8.Avec la valeur 6 en entrée (110 en binaire), la sortie numéro 6 est activée.

Un décodeur peut être utilisé pour n'activer qu'au plus un composant 3-états à la fois, puisqu'au plus une seule de ses sorties est active à la fois.
Les décodeurs peuvent également être utilisés pour implémenter des fonctions logiques booléennes. Puisque chaque sortie représente un des minterms possibles, et que toute fonction combinatoire booléenne s'exprime sous forme d'une somme de minterms, on reliera avec un OU les sorties correspondants aux minterms désirés. On peut voir Figure II-28 la réalisation d'un OU exclusif (XOR) à trois entrées avec un décodeur et un OU à 4 entrées.

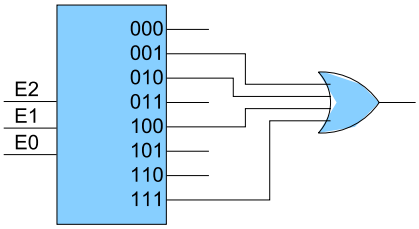
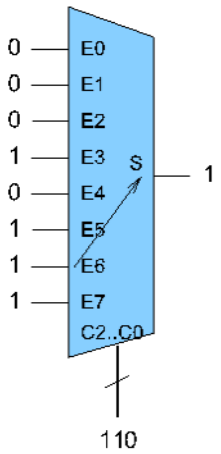


Figure II-28. Un décodeur 3 vers 8 utilisé pour réaliser un OU exclusif à 3 entrées.

II.7. Multiplexeurs

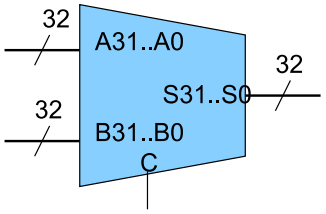
Ces sont des circuits d'aiguillage pour les signaux logiques. Un multiplexeur possède 2^n entrées de données, n entrées de commandes, et une seule sortie. On indique sur la commande le numéro (en binaire) de l'entrée de donnée qui va être aiguillée en sortie. On a en Figure II-29 un exemple de multiplexeur 8 vers 1 sur la commande duquel est écrit $110_2 = 6_{10}$; la sortie reflète alors l'état de la sixième entrée : E6. On donne aussi Figure II-29 l'écriture SHDL de ce multiplexeur, qui est celle d'un circuit combinatoire simple.




```
module mux8to1(e[7..0], c[2..0] : s)
    s = c[2]*c[1]*c[0]*e[7] +
        c[2]*c[1]*/c[0]*e[6] +
        c[2]*/c[1]*c[0]*e[5] +
        c[2]*/c[1]*/c[0]*e[4] +
        /c[2]*c[1]*c[0]*e[3] +
        /c[2]*c[1]*/c[0]*e[2] +
        /c[2]*/c[1]*c[0]*e[1] +
        /c[2]*/c[1]*/c[0]*e[0]
end module
```

Figure II-29. Multiplexeur 8 vers 1.L'entrée numéro 6 est aiguillée vers la sortie.

Bien sûr, les multiplexeurs peuvent être mis en parallèle pour aiguiller des bus entiers. On mettra alors en commun les lignes de commande, et en parallèle les lignes de données. On peut voir Figure II-30 un multiplexeur 2 vers 1 aiguillant des bus de 32 bits, avec l'écriture SHDL correspondante.




```
module mux32to1(a[31..0], b[31..0], c : s[31..0])
    s[31..0] = /c*a[31..0] + c*b[31..0]
end module
```

Figure II-30. Multiplexeur 2 vers 1, aiguillant des bus de 32 bits.

Un multiplexeur 2 vers 1 implémente matériellement une situation de type si-alors-sinon ; plus précisément : si C alors S=A sinon S=B (Figure II-31). C'est une remarque qui peut sembler banale, mais qui en fait va permettre d'organiser le découpage en modules d'un système complexe, dès lors qu'un tel genre de situation aura été identifié.

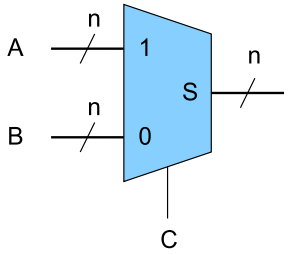


Figure II-31. Un multiplexeur 2 vers 1 implémente une situation 'si-alors-sinon' : si C alors S =A sinon S =B.

Un multiplexeur 2^n vers 1 peut également être utilisé directement pour implémenter des fonctions combinatoires à n entrées. Il suffit de relier les entrées de la fonction à réaliser aux entrées de commande du multiplexeur : chaque combinaison va conduire à un aiguillage spécifique, et il n'y a plus qu'à mettre la valeur 0 ou 1 attendue sur l'entrée de donnée correspondante du multiplexeur. Il n'y a cette fois aucun composant supplémentaire à rajouter, et on peut voir Figure II-32 l'implémentation d'une fonction XOR à 3 entrées avec un décodeur 8 vers 1. Bien sûr, cette solution n'est pas optimale en nombre de transistors utilisés, puisque toutes les entrées forcées à 0 sont reliées à des portes ET inutilisées.

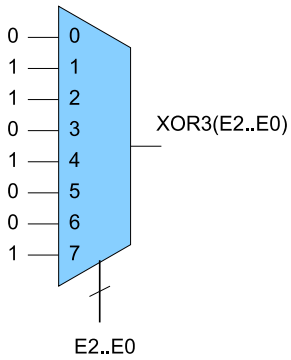
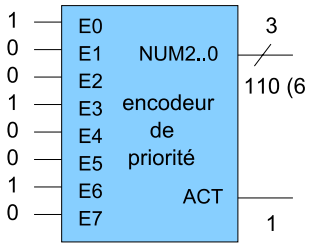


Figure II-32. Un multiplexeur 8 vers 1 utilisé pour réaliser un OU exclusif à 3 entrées.

II.8. Encodeurs de priorité

Un encodeur de priorité possède 2^n entrées et n sorties. Les entrées sont numérotées, et correspondent à des événements de priorité croissante. On verra en particulier comment utiliser les encodeurs de priorité pour gérer l'arrivée d'interruptions simultanées dans un processeur, telles que les événements réseau, les événements disque, les événements USB ou clavier ou souris, etc.
La sortie NUM contient le numéro de l'entrée activée la plus prioritaire, c'est à dire de numéro le plus élevé. Une autre sortie (ACT sur la figure) peut aussi indiquer s'il y a au moins une entrée active. Le schéma de la Figure II-33 montre un tel encodeur pour 23 entrées avec les entrées 0, 3 et 6 activées, et la valeur binaire 6 placée sur les sorties.




```
module encodeur(e[7..0]: num[2..0], act)
  num[2]=e[7]+e[6]+e[5]+e[4]
  num[1]=e[7]+e[6]+/e[5]*/e[4]*e[3]+
    /e[5]*/e[4]*e[2]
  num[0]=e[7]+/e[6]*e[5]+/e[6]*/e[4]*e[3]+
    /e[6]*/e[4]*/e[2]*e[1]
  act = e[7]+e[6]+e[5]+e[4]+e[3]+e[2]+e[1]+e[0]
end module
```

Figure II-33. Encodeur de priorités à 8 entrées. L'entrée active #6 est la plus prioritaire.

La table de vérité d'un tel circuit à 8 entrées possède 256 lignes ; on peut néanmoins la représenter de façon condensée (Figure II-34).
L'entrée #0 est souvent non câblée : l'absence d'entrée active est alors détectée lorsque NUM = 0, évitant ainsi d'avoir la sortie supplémentaire ACT.

e[7]	e[6]	e[5]	e[4]	e[3]	e[2]	e[1]	e[0]	num[2..0]	act
1	*	*	*	*	*	*	*	1 1 1	1
0	1	*	*	*	*	*	*	1 1 0	1
0	0	1	*	*	*	*	*	1 0 1	1
0	0	0	1	*	*	*	*	1 0 0	1
0	0	0	0	1	*	*	*	0 1 1	1
0	0	0	0	0	1	*	*	0 1 0	1
0	0	0	0	0	0	1	*	0 0 1	1
0	0	0	0	0	0	0	1	0 0 0	1
0	0	0	0	0	0	0	0	0 0 0	0

Figure II-34. Table de vérité condensée d'un encodeur de priorités à 8 entrées.

On a donc :

$$num[2] = e[7] + \overline{e[7]}.e[6] + \overline{e[7]}.\overline{e[6]}.e[5] + +\overline{e[7]}.\overline{e[6]}.\overline{e[5]}.e[4]$$
$$num[1] = e[7] + \overline{e[7]}.e[6] + \overline{e[7]}.\overline{e[6]}.\overline{e[5]}.\overline{e[4]}.e[3] + \overline{e[7]}.\overline{e[6]}.\overline{e[5]}.\overline{e[4]}.\overline{e[3]}.e[2]$$
$$num[0] = e[7] + \overline{e[7]}.e[6].e[5] + \overline{e[7]}.\overline{e[6]}.\overline{e[5]}.\overline{e[4]}.e[3] + \overline{e[7]}.\overline{e[6]}.\overline{e[5]}.\overline{e[4]}.\overline{e[3]}.\overline{e[2]}.e[1]$$

Le théorème d'absorption s'applique plusieurs fois, et on obtient finalement :

$$num[2] = e[7] + e[6] + e[5] + e[4]$$
$$num[1] = e[7] + e[6] + \overline{e[5]}.\overline{e[4]}.e[3] + \overline{e[5]}.\overline{e[4]}.e[2]$$
$$num[0] = e[7] + \overline{e[6]}.e[5] + \overline{e[6]}.\overline{e[4]}.e[3] + \overline{e[6]}.\overline{e[4]}.\overline{e[2]}.e[1]$$

L'écriture SHDL d'un encodeur à 8 entrées est donnée Figure II-33.

II.9. Circuit d'extension de signe

Un circuit d'extension de signe de p bits vers n bits, p < n est un circuit combinatoire qui transforme un nombre signé codé en complément à 2 sur p bits en ce même nombre codé en complément à 2 sur un nombre de bits n plus grand.
Considérons par exemple une extension de signe de 4 bits vers 8 bits, et quelques cas particuliers :

- 0011_2 (= 3) sera converti en 00000011_2
- 1111_2 (= -1) sera converti en 11111111_2
- 1000_2 (= -8) sera converti en 11111000_2

On voit donc que le mot de n bits est formé de la recopie du mot de p bits, complété à gauche de n − p bits tous égaux à 0 ou à 1 selon le signe du nombre. On recopie en fait n − p fois le bit de signe du mot de départ, c'est à dire le bit de rang p − 1 (Figure II-35).

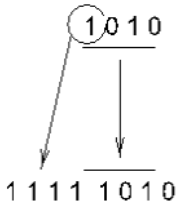


Figure II-35. Opération d'extension de signe. On duplique les p bits d'entrée et on ajoute à gauche n-p bits de signe.

La Figure II-36 montre à titre d'exemple le code SHDL correspondant à un circuit d'extension de signe de 13 bits vers 32 bits.

```
module signext13(e[12..0]: s[31..0])
   s[12..0] = e[12..0];
   s[31..13] =
e[12]*0b1111111111111111111
end module
```

Figure II-36. Écriture SHDL d'un circuit d'extension de signe 13 bits vers 32 bits. Les 13 bits de poids faibles sont recopiés et on ajoute 19 bits de poids forts, tous égaux au bit de signe e[12] du nombre de départ.

II.10. Décaleur à barillet

Le décaleur à barillet (barrel shifter) décale un mot binaire de n bits vers la gauche ou vers la droite, d'un nombre variable de bits. Le sens du décalage est défini par une commande R (pour right) et le nombre de bits du décalage est donné dans une commande NB sur p bits (Figure II-37). C'est un circuit directement employé à l'exécution des instructions de décalage et de rotation des processeurs, telles que les instructions sll et slr de CRAPS que nous verrons au chapitre suivant. Généralement, n = p 2 : un décaleur à barillet sur 8 bits aura une valeur de décalage codée sur 3 bits ; un décaleur sur 32 bits aura une valeur de décalage sur 5 bits, etc.

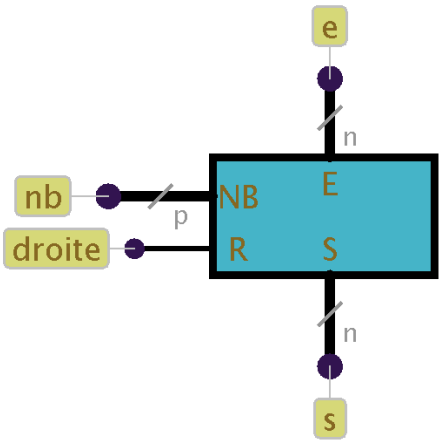


Figure II-37. Interface général d'un décaleur à barillet. Le mot de n bits est décalé à droite (resp.gauche) si R =1 (resp. 0), d'un nombre de bits NB. Les bits qui sortent sont perdus, et les bits entrants sont des 0.

Sur 8 bits par exemple, la Figure II-38 présente la table de vérité condensée d'un tel circuit, avec une valeur décalage sur 3 bits qui permet tous les décalages de 0 à 7 dans le sens droit (R = 1) comme dans le sens gauche (R = 0).

r	nb	entrées								sorties							
*	0 0 0	e_7	e_6	e_5	e_4	e_3	e_2	e_1	e_0	e_7	e_6	e_5	e_4	e_3	e_2	e_1	e_0
0	0 0 1	e_7	e_6	e_5	e_4	e_3	e_2	e_1	e_0	e_6	e_5	e_4	e_3	e_2	e_1	e_0	0
0	0 1 0	e_7	e_6	e_5	e_4	e_3	e_2	e_1	e_0	e_5	e_4	e_3	e_2	e_1	e_0	0	0
0	0 1 1	e_7	e_6	e_5	e_4	e_3	e_2	e_1	e_0	e_4	e_3	e_2	e_1	e_0	0	0	0
0	1 0 0	e_7	e_6	e_5	e_4	e_3	e_2	e_1	e_0	e_3	e_2	e_1	e_0	0	0	0	0
0	1 0 1	e_7	e_6	e_5	e_4	e_3	e_2	e_1	e_0	e_2	e_1	e_0	0	0	0	0	0
0	1 1 0	e_7	e_6	e_5	e_4	e_3	e_2	e_1	e_0	e_1	e_0	0	0	0	0	0	0
0	1 1 1	e_7	e_6	e_5	e_4	e_3	e_2	e_1	e_0	e_0	0	0	0	0	0	0	0
1	0 0 1	e_7	e_6	e_5	e_4	e_3	e_2	e_1	e_0	0	e_7	e_6	e_5	e_4	e_3	e_2	e_1
1	0 1 0	e_7	e_6	e_5	e_4	e_3	e_2	e_1	e_0	0	0	e_7	e_6	e_5	e_4	e_3	e_2
1	0 1 1	e_7	e_6	e_5	e_4	e_3	e_2	e_1	e_0	0	0	0	e_7	e_6	e_5	e_4	e_3
1	1 0 0	e_7	e_6	e_5	e_4	e_3	e_2	e_1	e_0	0	0	0	0	e_7	e_6	e_5	e_4
1	1 0 1	e_7	e_6	e_5	e_4	e_3	e_2	e_1	e_0	0	0	0	0	0	e_7	e_6	e_5
1	1 1 0	e_7	e_6	e_5	e_4	e_3	e_2	e_1	e_0	0	0	0	0	0	0	e_7	e_6
1	1 1 1	e_7	e_6	e_5	e_4	e_3	e_2	e_1	e_0	0	0	0	0	0	0	0	e_7

Figure II-38. Table de vérité condensée d'un décaleur à barillet sur 8 bits.

Un tel circuit est difficile à concevoir directement ; par ailleurs, si on voulait le réaliser sous forme d'un seul étage de propagation - ce qui est possible en théorie - les équations seraient complexes, et ce d'autant plus que les valeurs de n et p seraient grandes.
À l'inverse, une conception modulaire et récursive est possible de façon très simple, si on procède par étage, avec des décalages par puissances de 2 successives. Cette organisation générale est dite de décalage à barillet, et elle est montrée Figure II-39 pour un décaleur 8 bits. Tous les décaleurs ont leurs lignes R reliées ensemble, et donc opèrent dans le même sens. Ils sont tous équipés d'une ligne CS, et un décalage n'est effectué que si CS = 1. Ils effectuent chacun un nombre de décalages fixe, 4, 2 ou 1. Leur organisation en barillet permet d'effectuer toutes les valeurs de décalage entre 0 et 7. Le premier niveau effectue un décalage de 4 bits ou non, selon que nb[2] = 1 ou non. Le second niveau effectue ou non un décalage de 2 bits, selon la valeur de nb[1]. Le dernier niveau effectue un décalage de 1 bit, selon la valeur de nb[0].

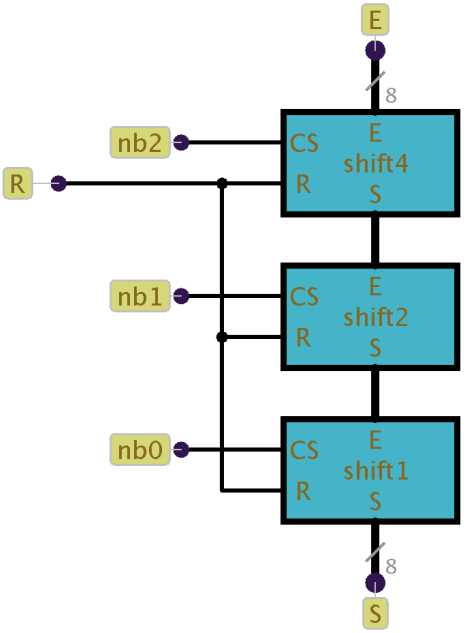


Figure II-39. Organisation en 3 niveaux (barillets) d'un décaleur 8 bits. Chaque niveau décale dans un sens ou un autre d'un nombre de positions égal à une puissance de deux, si sa ligne CS est à 1. Leurs combinaisons permettent toutes les valeurs de décalage.

Par exemple si nb = 101 et si R = 0, le premier niveau va décaler de 4 bits vers la gauche (sa ligne CS est à 1), transformant E = (e_7, e_6, e_5, e_4, e_3, e_2, e_1, e_0) en (e_3, e_2, e_1, e_0, 0, 0, 0, 0). Le second niveau va laisser passer ce vecteur sans le transformer. Le niveau du bas va effectuer un décalage de 1, transformant le vecteur en (e_2, e_1, e_0, 0, 0, 0, 0, 0). Au total, c'est donc bien 4 + 1 = 5 décalages à gauche qui ont été effectués.
Bien sûr, une telle organisation est généralisable à un nombre quelconque de bits. Un décaleur 32 bits serait constitué de 5 étages, formés de décaleurs fixes de 16, 8, 4, 2 et 1 bits. L'accroissement en complexité et en temps de propagation augmente donc en log(n), ce qui est très satisfaisant.

II.11. Sorties haute-impédance et buffers 3-états

Certains circuits possèdent des sorties dites trois états (tri-state), c'est à dire qu'en plus de pouvoir être dans l'état '0'ou l'état '1' (c'est à dire dans un état électrique de 'basse impédance'), elles peuvent être dans un troisième état dit de 'haute-impédance', souvent noté High-Z. Lorsqu'une sortie est en haute-impédance, tout se passe comme si elle n'était plus

connectée, car elle ne produit plus ni ne consomme plus aucun courant. Cette propriété permettra de relier directement entre-elles plusieurs sorties de ce type, sous réserve de garantir qu'au plus une seule de ces sorties produise du courant à un moment donné (sous peine de court-circuit !).
Les circuits ayant des sorties trois états possèdent en interne des composants appelés *buffer trois-états*, qui se représentent tels que sur la Figure II-40.

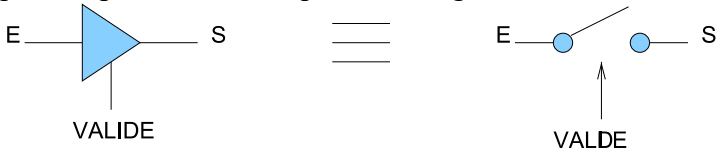


Figure II-40. Buffer 3-états - La sortie est en haute-impédance tant que VALIDE =0.

La ligne VALIDE qui arrive sur le côté du triangle, souvent notée OE (pour 'Output Enable'), commande l'état de la sortie. Tant que cette ligne est inactive, la sortie reste dans l'état High-Z.
Du point de vue de l'écriture SHDL, cette commande Output Enable sur un signal tel que S s'indique par l'ajout après le nom du signal d'une commande telle que : OE
Si on considère cet état High-Z comme un troisième état logique (ce qui n'est pas très exact sur le plan mathématique), on a la table de vérité de la Figure II-41.

A	B	$A + B$
0	0	0
0	1	1
1	0	1
1	1	1

Figure II-41. Table de vérité d'un buffer 3 états

On peut illustrer l'utilisation de ces buffers lors de la réalisation d'un multiplexeur (Figure II-42).

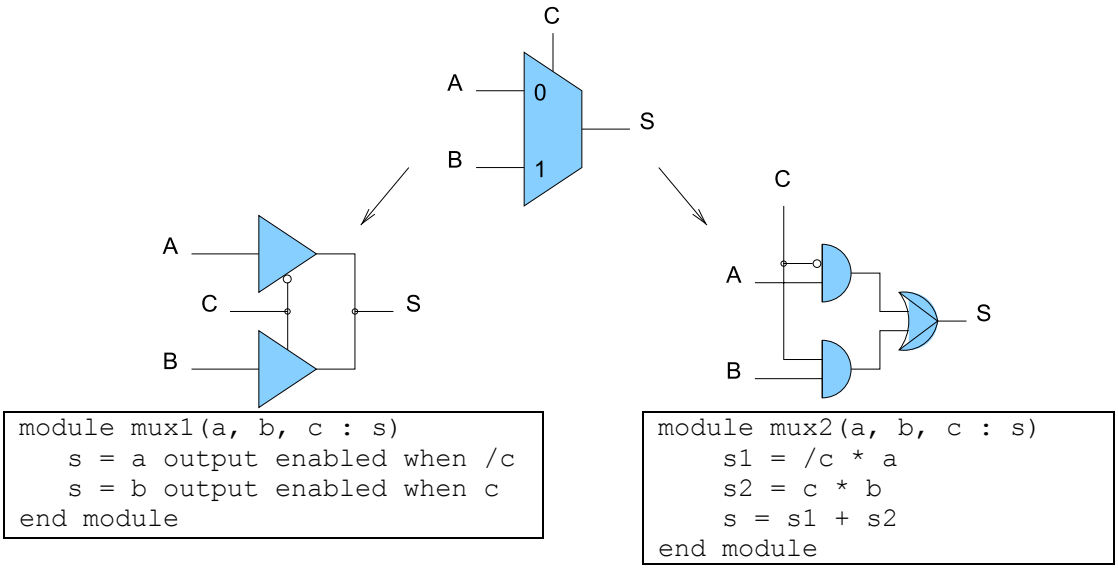


Figure II-42. Multiplexeur réalisé à l'aide de buffers 3-états, comparé à sa réalisation en logique combinatoire classique. On économise le 'ou' final.

Les sorties des deux buffers 3-états sont reliées entre-elles, car leurs lignes OE s'activent de façon mutuellement exclusive. L'idée de ce schéma est que, soit on doit laisser passer le courant qui vient de A (avec l'interrupteur commandé que forme le buffer trois états), soit on doit le laisser passer depuis B. Comme ces deux conditions sont mutuellement exclusives, on peut relier les sorties des deux buffers en toute sécurité. On économise ainsi le circuit OU du schéma classique de droite, qui n'opérait jamais avec ses deux entrées à 1 simultanément.
On notera l'écriture SHDL telle que : S = A:/C; L'équipotentielle s'appelle S, mais deux sources de courant peuvent y imposer leur potentiel ; on les note alors A:/C et B:C.
On voit sur la Figure II-43 une configuration électrique particulière de ce circuit : la commande C étant à 1, seul le buffer 3-états du bas va être dans un état de basse impédance, et va imposer son état électrique à la sortie S. Le buffer trois états du haut a sa sortie en haute impédance et n'entre pas en conflit avec l'autre pour l'établissement du potentiel de S ; tout se passe comme s'il était déconnecté du circuit, ce qui a été figuré par les deux lignes brisées.

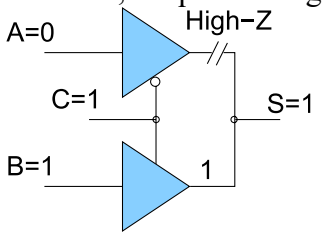


Figure II-43. Fonctionnement du multiplexeur. Dans cette configuration, c'est le buffer du bas qui impose son potentiel.

Si on relie ensemble deux sorties de buffers trois états, et qu'à un moment donné leurs lignes OE sont à 1en même temps, le court circuit est réel, et les circuits se mettent à chauffer dangereusement, entraînant parfois la destruction ou l'endommagement du plus faible.
Cela peut se produire dans des réalisations composées de plusieurs circuits intégrés, dont les sorties trois états sont reliées entre elles. Cela peut se produire également à l'intérieur d'un seul circuit comme un FPGA, dans lequel des lignes internes à trois états peuvent être reliées entre elles lors de la programmation - configuration. On évite parfois le pire avec un bon odorat, en détectant l'odeur de cuisson de la poussière présente à la surface des circuits à des températures avoisinant parfois les 100 degrés Celsius, et en débranchant en hâte l'alimentation !

C’est en fait la seule situation dangereuse pour les composants lorsque l’on fait de la logique digitale. Tant qu’on n’utilise pas de composants avec des sorties trois états, rien ne peut être endommagé si on prend soin de ne relier chaque sortie qu’à des entrées. Avec les composants ayant des sorties trois états, il faudra impérativement connecter leurs lignes OE à des circuits tels que des décodeurs (voir section 3) qui vont garantir l’exclusion mutuelle entre les sorties reliées entre elles.

II.12. Exercices corrigés

Exercice 1 : utilisation des tables de Karnaugh

Concevoir un circuit qui détecte la présence d’un chiffre décimal (entre 0 et 9) sur ses entrées A, B, C, D.

Solution

La table de vérité de ce détecteur donne ’1’ pour les valeurs de (0,0,0,0) à (1,0,0,1), et ’0’ pour les valeurs suivantes (Figure II-44).

A	B	C	D	S
0	0	0	0	1
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	1
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

Figure II-44. Table de vérité d’un détecteur de chiffre décimal.

On peut maintenant dessiner la table de Karnaugh correspondante, et chercher le plus petit nombre de regroupements qui recouvrent tous les ’1’ (Figure II-45).

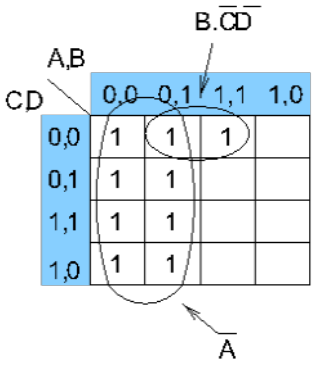


Figure II-45. Table de Karnaugh avec les regroupements pour le détecteur de chiffres décimaux.

Il y a deux regroupements, ce qui donne l'expression simplifiée :

$$S = \bar{A} + B\bar{C}\bar{D}$$

On peut trouver le même résultat de façon algébrique :

$$S = ((A,B,C,D) \leq 8) + ((A,B,C,D) = 9)) = \bar{A} + AB\bar{C}\bar{D}$$

Par absorption de A :

$$S = \bar{A} + B\bar{C}\bar{D}$$

Exercice 2 : analyse inductive des tables de vérité

Concevoir les transcodeurs suivants :

1. Binaire pur vers Gray réfléchi.
2. Gray réfléchi vers binaire pur.

Solution

La figure II.61 montre la table de vérité qui relie une valeur binaire (X,Y,Z,T) à un code de Gray (A,B,C,D). Le code de Gray a été construit selon la méthode récursive décrite à la section 5.4.

On pourrait dessiner des tables de Karnaugh, mais elles ne donneraient rien ici. Par ailleurs on va voir que les codes de Gray sont très étroitement associés à l'opérateur XOR, et on va essayer de deviner les relations directement.

X	Y	Z	T	A	B	C	D
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	1
0	0	1	1	0	0	1	0
0	1	0	0	0	1	0	0
0	1	0	1	0	1	0	1
0	1	1	0	0	1	1	1
0	1	1	1	0	1	1	0
1	0	0	0	1	0	0	0
1	0	0	1	1	0	0	1
1	0	1	0	1	0	1	1
1	0	1	1	1	0	1	0
1	1	0	0	1	1	0	0
1	1	0	1	1	1	0	1
1	1	1	0	1	1	1	1
1	1	1	1	1	1	1	0

Figure II-46. Binaire <-> code de Gray.

1. Binaire pur vers Gray réfléchi.

On voit tout de suite que A = X. B est égal à Y pendant la première moitié de la table, puis il est égal à $\bar{Y}$. 'Être dans la première moitié de la table' est gouverné par X ; en résumé B est égal à Y avec une inversion commandée par X. C'est XOR l'opérateur d'inversion commandée, donc : $B = X \oplus Y$

De façon analogue, on remarque que C est égal à Z lorsque Y vaut 0, et est égal à $\bar{Z}$ lorsque Y vaut 1, donc : $C = Y \oplus Z$. On trouve de même $D = Z \oplus T$.

2. Gray réfléchi vers binaire pur.

On a bien sûr X = A, et on voit aussi que $Y = A \oplus B$. Par contre pour Z, il est bien égal à C ou $\bar{C}$, mais l'inversion n'est pas simplement commandée par B. Il y a inversion durant le deuxième et le troisième quart de la table : cela évoque un XOR. Il y a en fait inversion lorsque $A \oplus B = 1$, donc : $Z = A \oplus B \oplus C$. Par induction on devine, et on vérifie ensuite, que $T = A \oplus B \oplus C \oplus D$

Chapitre III. Arithmétique binaire

III.1. Mots binaires

Les signaux binaires sont souvent regroupés pour former des mots. La largeur d'un mot binaire est le nombre des signaux qui sont regroupés. On appelle par exemple octet un mot de 8 bits (un mot de largeur 8).Un mot de n bits permet de coder 2^n valeurs différentes ; 256 valeurs par exemple pour un octet. On peut utiliser un octet pour coder un caractère alphanumérique à la norme ISO-8859 ; un mot de 32 bits pour représenter un nombre entier relatif ; un mot de 16 à 96 bits pour représenter des nombres réels en notation scientifique flottante selon la norme IEEE 754. Des structures de données plus complexes (tableaux, listes, ensembles, dictionnaires, etc.) nécessitent une agrégation de ces types simples.

Codage en binaire pur

On écrira les nombres binaires avec le chiffre 2 en indice. Par exemple : 01101101_2.
Sur n bits, l'intervalle des valeurs représentées est $[0, 2^n-1]$. Comme en décimal, le total est une somme pondérée, plus précisément le produit de chaque chiffre par la puissance de 2 de même rang:

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

où b_i est le bit de rang i du mot.
Par exemple, on a représenté sur la figure suivante les poids respectifs des bits du mot de 8 bits $10011010_2 = 154_{10}$:

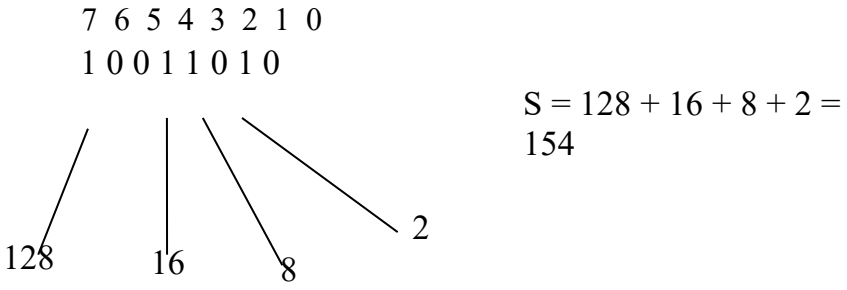


Figure III-1. Poids des chiffres en binaire pur

Poids forts, poids faibles

Par référence à cette notion de pondération des chiffres, on emploie couramment des expressions telles que 'les n bits de poids les plus forts', ou 'les n bits de poids les plus faibles' pour désigner les n bits les plus à gauche (respectivement les plus à droite) dans un mot binaire. Dans les documentations en anglais, on trouvera fréquemment des libellés tels que MSB (most significant bits) ou LSB (least significant bits) avec les même sens.

1 Kilo n'est pas 1000 (en informatique)

Lorsque vous achetez un kilowatt-heure d'électricité, il est bien entendu avec votre fournisseur qu'il s'agit de 1000 watts-heure. Ce n'est pas la même chose en informatique ! Pour continuer à énumérer en utilisant la base 2, les informaticiens ont décidé d'utiliser l'unité $1K = 2^{10} = 1024$, donc un peu plus de 1000. Le suffixe 'méga' est : $1M = 1K \times 1K = 2^{20}$; un 'giga' est : $1G = 1K \times 1M = 2^{30}$; un 'Tera' est : $1T = 1K \times 1G = 240$. Un grand nombre de valeurs usuelles s'expriment alors directement en multiples de 1K, 1M, 1G ou 1T, par exemple la taille des mémoires RAM, car elle est nécessairement une puissance de 2 à cause de leur organisation matricielle.

1K, 1M et 1G valent environ mille, un million et un milliard, mais la petite différence entre 1000 et 1K augmente lorsqu'on passe à 1M et 1G : 1M = 1024 * 1024 = 1 048 576, et 1G = 1024 * 1048576 = 1 073 741 824, soit presque 1,1 milliard. Les vendeurs de disques durs exploitent cette ambiguïté : le plus souvent si vous achetez un disque étiqueté '100 giga-octets', il s'agit d'un disque de 100 milliards d'octets, et non de 107 374 182 400 octets, soit une différence de 7%.

64 bits, la taille idéale pour les données ?

Les microprocesseurs manipulent des mots binaires de taille fixe, les plus petits de 8 bits, d'autres de 16 bits, 32 bits et maintenant 64 bits. Un processeur 8 bits, s'il veut ajouter 1000 et 1000, doit effectuer deux additions et non une seule, puisque 1000 ne 'rentre' pas dans un codage 8 bits. Un processeur 16 bits code directement des nombres entiers de l'intervalle [-32768, +32767], ce qui ne permet pas de coder des nombres un tant soit peu grands. On semble définitivement à l'aise avec 32 bits, mais ce n'est pas le cas : le plus grand entier qui peut être codé est 2^{32}, soit environ 4 milliards, ce qui ne suffit pas à représenter le montant en euros d'une journée de transactions à la bourse de Paris.
Avec 64 bits, on est définitivement sauvés côté entiers, à moins de vouloir compter les grains de sable sur la plage. Pour le codage des nombres réels, 64 bits est la taille de codage en double précision la plus usuelle, qui donne une précision d'une dizaine de chiffres après la virgule, suffisante dans la majorité des applications. Comme on doit aussi souvent stocker des adresses de mémoire dans des mots, 64 bits est aussi une taille suffisante, alors que 32 bits ne permettent pas de dépasser 4 giga-octets.
Une taille de mot de 64 bits a donc des qualités intrinsèques que n'avaient pas jusque là 16 bits ou 32 bits. Utiliser une taille plus grande de 128 bits dans un processeur conduirait à beaucoup de gaspillage, puisque la plupart des mots manipulés seraient très peu remplis. Ma prédiction personnelle est que, dans le futur, cette taille du mot mémoire ne va pas augmenter indéfiniment, mais va se stabiliser à 64 bits du fait des qualités qui viennent d'être décrites.

Nombres binaires à virgule

Comme en décimal, on peut manipuler des nombres à virgule en binaire. Les chiffres placés après la virgule ont alors des poids qui sont les puissances négatives décroissantes de 2. Ainsi, $0.1_2 = 2^{-1} = 0.5$; $0.01_2 = 2^{-2} = 0.25$, etc.
Si on cherche la représentation binaire approchée du nombre π, on trouvera π = 11.001001… (Figure III-2).

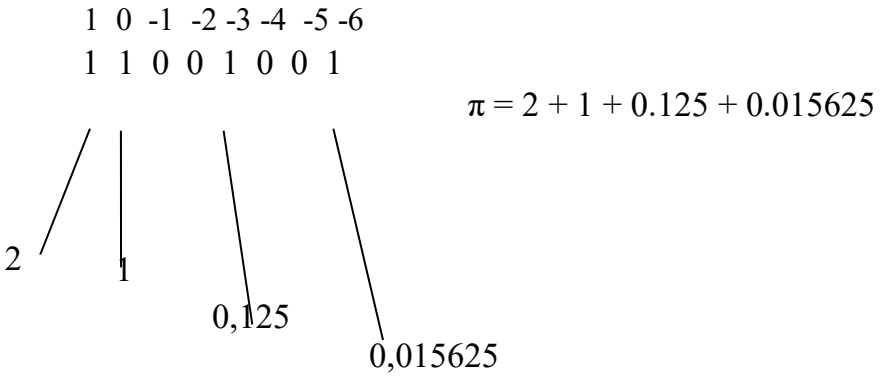


Figure III-2. Écriture binaire à virgule (approchée) du nombre π.

Pour chaque position, on met 1 ou 0 de façon à ce que la valeur courante ne dépasse pas la valeur à représenter, tout en s'en approchant le plus possible. Par exemple après avoir choisi 11.001 (total courant = $2^1 + 2^0 + 2^{-3} = 2 + 1 + 0.125 = 3.125$), on ne peut pas mettre de 1 en

position -4 ou -5, car cela dépasserait π. On doit attendre la position -6, et on a une valeur approchée de $2^1 + 2^0 + 2^{-3} + 2^{-6} = 3.140625$.

Incrémenter et décrémenter en binaire

Une opération fréquente effectuée sur des nombres binaires est le comptage. Par exemple sur 3 bits, passer de 000 à 001, puis 010, etc. Selon quel algorithme une valeur se transforme-t-elle en la suivante ?

On peut essayer de s'inspirer du comptage en décimal, qu'on sait faire depuis notre petite enfance, et pour lequel on ne nous a pourtant jamais donné l'algorithmique. Pour passer de 124 à 125, seul le chiffre des unités s'est incrémenté. En fait ce chiffre s'incrémente toujours, avec un passage de 9 à 0. Le chiffre des dizaines sera recopié avec incrémentation, si le chiffre des unités vaut 9, comme dans le passage de 129 à 130, sinon il sera recopié sans changement, comme dans le passage de 124 à 125. On tient notre algorithme : pour passer d'un nombre décimal à son suivant, on considère chaque chiffre un par un, et on le recopie en l'incrémentant si tous les chiffres qui sont à sa droite valent 9, ou sinon on le recopie sans changement. On remarquera que l'ordre de traitement sur les chiffres est indifférent, et qu'on peut effectuer cet algorithme en parallèle sur tous les chiffres. La Figure III-3 montre un exemple.

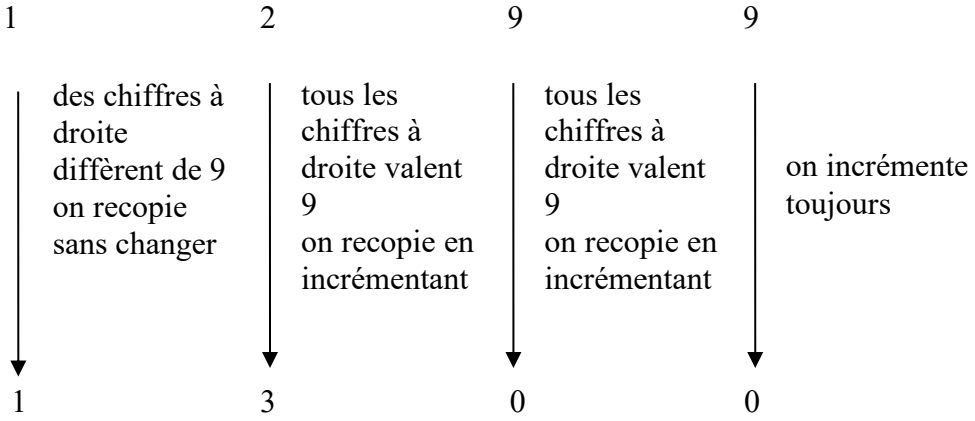


Figure III-3. Incrémentation d'un nombre décimal. Un chiffre est recopié avec incrémentation si tous les chiffres à sa droite valent 9 ; sinon il est recopié sans changement.

Le même algorithme s'applique en binaire, mais cette fois l'opération d'incrémentation d'un chiffre est encore plus simple, car elle se résume à une inversion du bit. Pour incrémenter un nombre binaire, on considère chaque bit du mot initial, et on le recopie avec inversion si tous les bits qui sont à sa droite valent 1, ou on le recopie sans changement sinon.

Le bit le plus à droite est toujours inversé. La Figure III-4 montre un exemple.

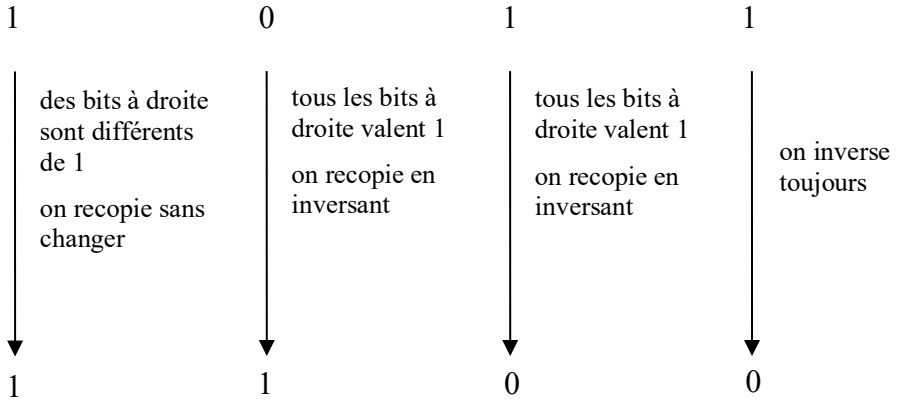


Figure III-4. Incrémentation d'un nombre binaire. Un bit est inversé si tous les bits à sa droite valent 1 ; sinon il est recopié sans changement.

Là encore l'algorithme peut s'effectuer en parallèle sur tous les bits, donc peut être réalisé directement par un circuit combinatoire, dont c'est le mode de fonctionnement naturel. Attention, il faut noter qu'on ne peut pas incrémenter la valeur 'sur place': le mot binaire qui contient le nombre de départ et le mot binaire qui contient le résultat doivent être distincts, et ne peuvent pas être confondus.
Un algorithme analogue existe pour le décomptage : on recopie un bit avec inversion si tous les bits à sa droite valent 0, sinon on le recopie sans changement. Par vacuité, le bit le plus à droite est toujours inversé. Par exemple, on passe de 110 à 101 en inversant les deux bits de poids faible, car tous les bits qui sont à leur droite valent 0.

Hexadécimal

Les mots manipulés par un ordinateur ont souvent une largeur supérieure à 16 ou 32 bits, et sont donc difficiles à transcrire en binaire. Pour obtenir une écriture concise, on utilise souvent la base 16, appelée hexadécimal, dont les 16 chiffres sont notés : 0,1,2,3,4,5,6,7,9,A,B,C,D,E,F. Chaque chiffre hexadécimal permet de coder 4 bits ($2^4 = 16$). Pour passer d'une notation binaire à une notation hexadécimale, il suffit de grouper les bits par paquets de 4, de droite à gauche à partir des poids faibles. Par exemple :

$0110.1101_2 = 6D_{16} = 109_{10}$
$0110.1000.1010.1100_2 = 68AC_{16}$

Débordement lors d'une addition non signée

Lorsqu'on additionne des nombres entiers naturels codés en binaire pur, une retenue finale indique un débordement.

III.2. Codage des nombres entiers naturels

On utilise le binaire pur. Avec n bits, on peut coder les nombres entiers positifs de l'intervalle $[0, 2^n-1]$.
L'addition et la soustraction s'effectuent de la manière habituelle ; il y a débordement de l'addition lorsqu'une retenue doit être propagée au delà du dernier bit de rang n.

Lecteur et imprimeur décimal

Si un programme utilise des nombres entiers naturels codés en binaire pur, la représentation externe de ce nombre à l'utilisateur du programme est généralement faite dans une autre base, le plus souvent la base 10. Un sous-programme appelé ***imprimeur*** est chargé de convertir une valeur en binaire pur en son écriture décimale, et un autre appelé ***lecteur*** fait la traduction inverse. On peut examiner rapidement les algorithmes qu'ils utilisent. Ils seraient bien sûr facilement transposables à d'autres bases.

Traduction du binaire vers le décimal

Considérons par exemple un mot de 8 bits $N = 10110110_2$; on cherche la suite des caractères qui représente ce nombre en écriture décimale. C'est un programme d'ordinateur qui réalise cette conversion, et qui dispose donc d'opérateurs arithmétiques (binaires) tels que addition et multiplications, qui sait faire des comparaisons, etc. Cette traduction ne comportera pas plus de 3 chiffres décimaux, notons les : XYZ. On choisit pour X le chiffre le plus grand tel que $100X \leq N$. Ici $X = 1$ convient car $1\times100 \leq N$ et $2\times100 > N$. Pour faire ce choix de X, on peut effectivement réaliser des multiplications, ou utiliser une table préconstruite. Il reste à traduire $N1 = N - X\times100 = 1010010_2$. Pour cette valeur, on cherche la plus grande des valeurs de Y

telle que 10×Y ≤ N1 ; on trouve Y = 8, et il reste à convertir N2 = N1 − Y×10 = 00000010_2 ; on trouve alors Z = 2. Finalement, les trois chiffres décimaux cherchés sont XYZ = 182.

Traduction du décimal vers le binaire

On cherche donc à convertir l'écriture décimale d'un nombre en un mot binaire. Le problème n'est pas le symétrique du précédent, car l'ordinateur sait faire toutes les opérations sur le codage binaire, alors qu'il ne sait en faire aucune directement en base 10.
Pour un nombre décimal tel que 'XYZ' (qui conduit à une valeur binaire sur 8 bits), il suffit en fait de faire le calcul n = 100x + 10y + z où x, y et z sont les numéros d'ordre associés aux chiffres 'X', 'Y' et 'Z' respectivement.

III.3. Codage des nombres entiers relatifs

Aux débuts de l'informatique, on codait généralement les nombres entiers signés en mettant dans le premier bit le signe ('1' signifiant 'moins') et dans les suivants la valeur absolue. -3 se codait par exemple 10000011 sur 8 bits. L'encodage et le décodage d'un tel nombre étaient simple (bien qu'on notera que 0 peut être codé +0 ou -0), mais on ne pouvait plus réaliser une simple addition comme avec des nombres naturels. Si par exemple on tente de faire l'addition bit à bit de 10000011 (-3) et de 00000100 (+4), on obtient : 10000111, c'est à dire -7, ce qui est absurde.

Codage en complément à 2

Un codage appelé complément à deux s'est rapidement imposé dans les années 1970.
Les différents bits d'un nombre codé en complément à 2 ont pour poids les puissances successives de 2, comme pour le binaire pur, sauf pour le bit de poids fort, qui a un poids de -2^{n-1}au lieu d'avoir un poids de 2^{n-1} :

$$s = -2^{n-1}b_{n-1} + \sum_{i=0}^{n-2} 2^i b_i$$

où b_i est le bit de rang i du mot.
On peut prouver que ce codage a les propriétés remarquables suivantes :

- c'est un codage équilibré: sur n bits, on représente les nombres relatifs de l'intervalle [-2^{n-1}, $2^{n-1}-1$] soit autant de nombre positifs ou nuls que de nombres strictement négatifs
- 0 se code sous forme d'un champ uniforme de 0 ; il n'a qu'un seul codage
- le bit de poids fort du codage représente le signe du nombre
- les opérations d'addition ou de soustraction en binaire pur s'appliquent également avec des opérandes codés en complément à deux, avec la différence que la retenue n'a pas de sens et doit être négligée lors d'une opération en complément à deux

Pour illustrer le fait que le même opérateur d'addition est utilisé pour le binaire pur et le complément à 2, considérons l'addition de la

interprétation non signée		interprétation signée
233	11101001	-23
+ 66	+ 01000010	+ +66
299	1 00101011	+43
(débordement)		(pas de débordement)

Figure III-5 :

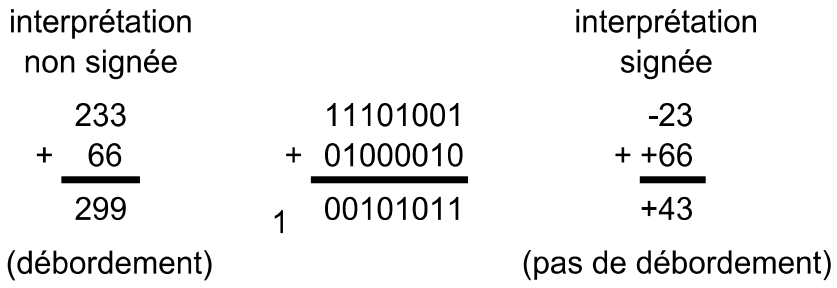


Figure III-5. Une même opération 'mécanique' d'addition s'interprète de deux façons différentes.

La même opération effectuée mécaniquement sur des codages (colonne centrale) s'interprète de deux façons différentes selon qu'on considère les nombres comme étant signés ou non.

Débordement lors de l'addition de nombres signés

Quand y a-t-il débordement lors d'une addition de deux nombres signés ? Il est clair qu'il ne peut pas se produire lorsque les opérandes sont de signes opposés, puisque la valeur absolue du résultat est alors inférieure à la plus grande des valeurs absolues des deux opérandes. Il ne peut donc se produire que s'ils sont de même signe, tous les deux positifs ou tous les deux négatifs. On démontre qu'alors il y a débordement si et seulement si l'addition des deux donne un résultat de signe opposé au signe commun des deux opérandes.

Comparaison de nombres en complément à 2

On se pose ici le problème de déterminer les positions respectives de deux nombres entiers A et B codés en complément à 2 : sont ils égaux, ou l'un est-il plus grand que l'autre ? Il faut d'abord examiner leurs signes, c'est à dire leurs bits de poids forts.
Si les signes sont différents, la comparaison est immédiate. Si les signes sont identiques, il suffit de comparer les n − 1 autres bits dans l'ordre binaire naturel, que les nombres soient positifs ou négatifs.

Calcul de l'opposé

Avec le codage en complément à 2, on démontre que le codage de l'opposé d'un nombre s'obtient en appliquant la procédure suivante:

- on inverse tous ses bits (= « complément à 1 »)
- on ajoute 1, sans tenir compte de l'éventuelle retenue (= « complément à 2 »)

Par exemple, +3 est codé 00000011 sur 8 bits. Pour obtenir le codage de -3, on inverse tous les bits du codage précédent, soit 11111100 et on ajoute 1, soit 11111101 qui est bien le codage de -3. Si on réapplique la procédure au codage de -3, on retrouvera le codage de +3.

Calcul de l'opposé (bis)

On démontre également que l'opposé peut se calculer à partir de l'original de la façon suivante : le bit de poids faible est recopié tel quel ; les autres bits sont inversés seulement s'il existe un bit à 1 sur leur droite dans l'original.

III.4. Problématique de l'addition

Lorsqu'on effectue une addition en binaire, la méthode la plus simple consiste à l'effectuer bit à bit, en commençant par le bit de poids faible (figure II.29).

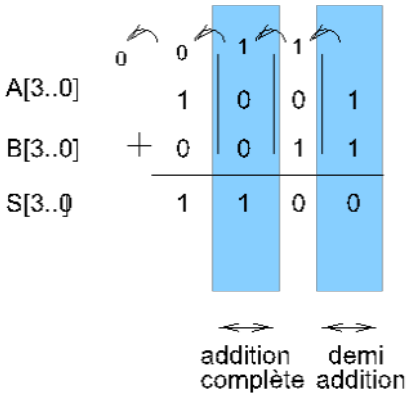
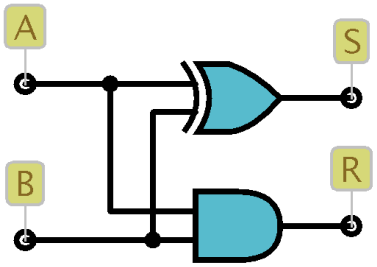


Figure III-6. Addition en binaire bit à bit.

A chaque rang i de l'addition, on fait la somme des bits A[i] et B[i], ainsi que de l'éventuelle retenue qui viendrait du rang précédent i − 1. Ce calcul produit le bit S[i] du résultat, ainsi qu'une retenue qui sera passée au rang suivant. Lorsqu'on a effectué toutes ces additions depuis le bit de poids le plus faible (rang 0) jusqu'au bit de poids le plus fort, la retenue finale sera la retenue qui sort du dernier rang d'addition.
On appelle ***demi-additionneur*** le circuit qui fait le calcul du rang 0, car il n'a pas à prendre en compte de retenue qui viendrait d'un étage précédent. On appelle ***additionneur complet*** le circuit qui prend en compte le calcul d'un bit de rang quelconque différent de 0.

III.4.1. Demi-additionneur

Un demi-additionneur est le circuit qui réalise une addition entre deux bits A et B, et qui produit la somme sur un bit S avec l'éventuelle retenue R. La table de vérité et le module correspondant sont représentés figure II.30.
Il est clair en effet que la somme est le XOR entre A et B, et la retenue ne peut intervenir que lorsque A et B valent 1 tous les deux.



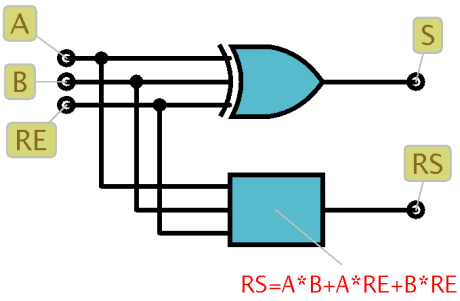
A	B	S	R
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Figure III-7. Table de vérité et schéma du demi-additionneur.

III.4.2. Additionneur complet

Un additionneur complet est un demi-additionneur qui comporte en plus en entrée une retenue entrante qui viendrait d'un autre étage d'addition. Les entrées sont donc les deux bits A et B et la retenue entrante RE ; les sorties sont le bit résultat S et la retenue sortante RS. Il est facile de déterminer S et RS au vu de leur fonction, sans avoir besoin de leur table de vérité :

- S est la somme des trois bits A, B et RE, et on sait depuis la section 3 que c'est un XOR à trois entrées : $S = A \oplus B \oplus RE$
- RS est la retenue de cette addition, et vaut 1lorsqu'il y a au moins deux valeurs à 1 dans le triplet (A, B, RE) : c'est donc la fonction majorité déjà vue en section 3, $RS = A.B + A \cdot RE + B \cdot RE$




```
module fulladder(A, B, RE: S, RS)
    S = /A*/B*RE + /A*B*/RE + A*/B*/RE + A*B*RE
    RS = A*B + A*RE + B*RE
end module
```

Figure III-8. Additionneur complet : la somme est un XOR, et il y a retenue lorsqu'une majorité des entrées vaut 1.

III.4.3. Additionneur ripple-carry

Un additionneur ***ripple carry*** réalise littéralement la méthode d'addition colonne par colonne, en commençant par une demi-addition des poids faibles et en enchaînant ensuite l'addition complète des bits suivants. Sur n bits, il nécessitera un demi-additionneur, et n − 1 additionneurs complets. Sur 4 bits par exemple, il prend la forme de la figure II.32.
La description en langage SHDL est très simple : il suffit de combiner les écritures d'un demi-additionneur et de 3 additionneurs complets (figure II.33).
Ce type d'additionneur est simple, mais lent : il faut attendre que toutes les retenues se soient propagées depuis le bit de poids le plus faible jusqu'au bit de poids le plus fort pour que le calcul soit terminé. Or un calcul d'addition est un calcul combinatoire, donc qui peut être effectué en théorie en une étape de propagation, sous forme d'une somme de minterms, et sans tous ces termes intermédiaires que représentent les retenues. Mais les équations à écrire deviennent très complexes dès le rang 3, et sont donc impraticables. Entre ces deux extrêmes, il est nécessaire de concevoir des additionneurs de complexité raisonnable pour un nombre de bits supérieur à 8, et avec des temps de propagation qui ne croissent pas linéairement avec ce nombre de bits. On va voir en section 8.5 comment les additionneurs ***carry lookahead*** parviennent à ce compromis.

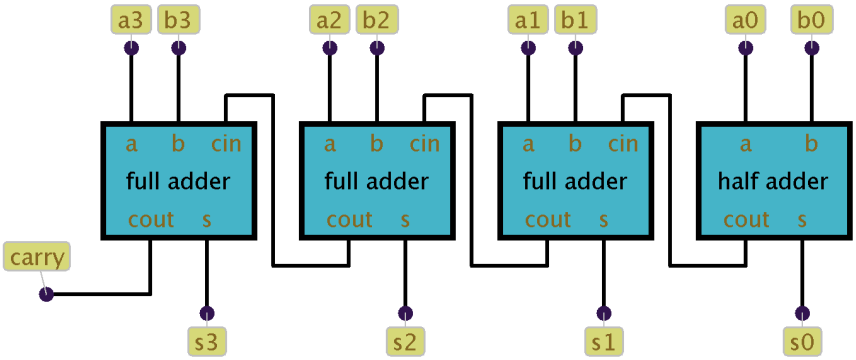


Figure III-9. Addition ripple carry 4 bits.Le rang 0 est calculé avec un demi-additionneur, et les autres rangs avec des additionneurs complets.

```
module ripplecarry4(a3,b2,a1,a0,b3,b2,b1,b0: s3,s2,s1,s0,carry)
    halfadder(a0,b0:s0,c0)
    fulladder(a1,b1,c0:s1,c1)
    fulladder(a2,b2,c1:s2,c2)
    fulladder(a3,b3,c2:s3,carry)
end module

module halfadder(a,b: s,cout)
    s = /a*b+a*/b
    cout = a*b
end module

module fulladder(a,b,cin: s,cout)
    s = /a*/b*cin+/a*b*/cin+a*/b*/cin+a*b*cin
    cout = a*b + a*cin + b*cin
end module
```

Figure III-10. Écriture SHDL d'un additionneur ripple-carry 4 bits. On combine un module halfadder et 3 modules fulladder.

III.4.4. Additionneur carry-lookahead

On a vu en section 8.1 que c'était la propagation de la retenue qui ralentissait le calcul d'une somme dans un additionneur ripple carry. Si on calcule a[n..0] + b[n..0], on a pour chaque additionneur complet :

$$s_i = a_i \oplus b_i \oplus c_i$$
$$c_i = a_i \,.\, b_i + a_i \,.\, c_i + b_i \,.\, c_i$$

Calculer s_3 par exemple nécessite de calculer c_3, donc s_2, donc c_2, etc. La méthode ***carry lookahead*** part du constat qu'on peut facilement déterminer à chaque étage s'il y aura une retenue ou non, en raisonnant en termes de retenue propagée et de retenue générée. Considérons l'addition des deux termes suivants :

P	P	P	G	G	P		P
1	0	1	1	1	0	0	1
0	1	0	1	1	1	0	0

Un 'G' est placé au dessus d'une colonne de rang *i* lorsque les termes à ajouter a_i et b_i valent tous deux '1' : on est alors sûr qu'une retenue sera 'Générée' pour l'étage suivant. Sinon, un 'P' est placé lorsqu'un des deux termes vaut '1' : si une retenue arrive depuis l'étage précédent, alors cet étage va la 'Propager', puisqu'on sera alors sûr qu'il y a au moins deux '1' parmi les trois bits à additionner à ce rang. En raisonnant uniquement sur les 'P' et les 'G', on trouve facilement quels étages vont émettre une retenue. Ainsi sur l'exemple, l'étage 0 peut propager une retenue, mais aucune n'a encore été générée. Les étages 3 et 4 génèrent une retenue, et les étages suivants 5, 6 et 7 la propagent.

Les termes G_i et P_i sont très faciles à calculer :

$$G_i = a_i \,.\, b_i$$
$$P_i = a_i + b_i$$

Un module autonome peut prendre en entrées les valeurs des G_i et P_i de tous les rangs et les utiliser pour calculer les retenues entrantes de chaque additionneur complet, selon le schéma général de la Figure III-11.

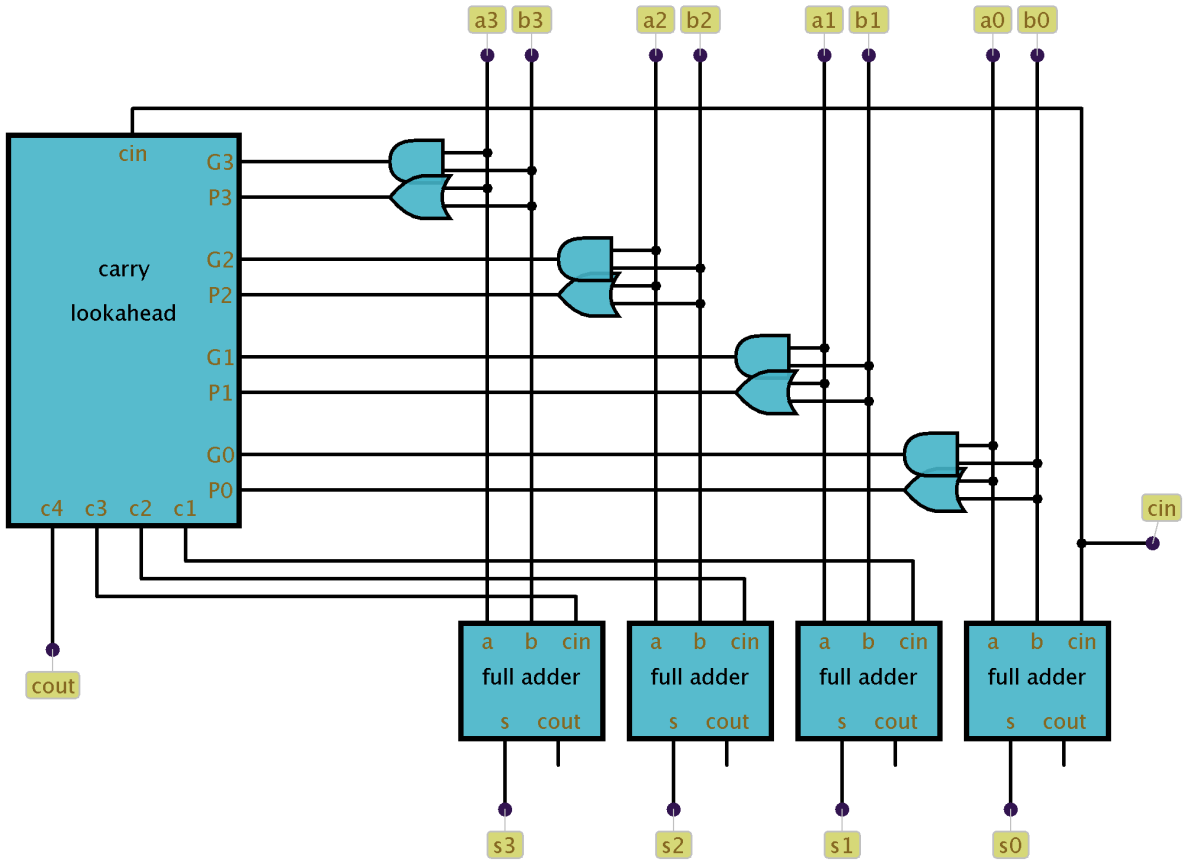


Figure III-11. Schéma général d'un additionneur carry-lookahead. On calcule pour chaque rang i deux bits Gi et Pi qui indiquent si cet étage va générer une retenue, ou en propager une de l'étage précédent.Un module spécifique carry_lookahead calcule rapidement à partir des Gi et Pi les retenues ci pour tous les rangs.

On notera que les retenues sortantes *cout* des additionneurs complets ne sont pas exploitées, puisque c'est le module *carry_lookahead* qui s'occupe du calcul de toutes les retenues.
Il faut maintenant réaliser un module *carry_lookahead* qui effectue ce calcul dans le temps le plus court. L'idée générale est qu'il y a une retenue au rang *i* si G_i = 1 ou si P_i= 1 et si une retenue existe au rang $i - 1$:

$$c_{i+1} = G_i + c_i \cdot P_i$$

On trouve :

$$c_1 = G_0 + c_{in} \cdot P_0$$
$$c_2 = G_1 + c_1 \cdot P_1 = G_1 + G_0.P_1 + c_{in} \cdot P_0.P_1$$
$$c_3 = G_2 + c_2 \cdot P_2 = G_2 + G_1.P_2 + G_0.P_1.P_2 + c_{in} \cdot P_0.P_1.P_2$$
$$c_4 = G_3 + c_3 \cdot P_3 = G_3 + G_2.P_3 + G_1.P_2.P_3 + G_0.P_1.P_2.P_3 + c_{in} \cdot P_0.P_1.P_2.P_3$$

Il ne faut pas se leurrer : il y a là aussi une cascade dans les calculs, mais elle porte sur des termes plus simples que si on avait opéré directement sur les termes *ai* et *bi*. Au-delà de 7 à 8 bits, les équations du module carry_lookahead deviennent trop complexes pour être implémentées sous forme de sommes de termes, et des signaux intermédiaires doivent être introduits. Si on considère la somme de produits comme unité de calcul et de propagation (hypothèse d'implémentation dans un CPLD ou un FPGA), le calcul complet d'une somme nécessite 1temps pour le calcul des Gi et Pi, 1temps pour le calcul des retenues, et 1 temps pour l'additionneur complet, soit un total de 3 temps de propagation, contre 4 pour l'additionneur ripple carry. Le gain est faible, mais il augmente à mesure que le nombre de bits grandit.
L'écriture complète d'un tel additionneur 4 bits en langage SHDL est donnée Figure III-12

```
end module

module carry_lookahead(G[3..0],P[3..0],c0:c4,c3,c2,c1)
    c1=G[0]+P[0]*c0
    c2=G[1]+P[1]*G[0]+P[1]*P[0]*c0
    c3=G[2]+P[2]*G[1]+P[2]*P[1]*G[0]+P[2]*P[1]*P[0]*c0
    c4=G[3]+P[3]*G[2]+P[3]*P[2]*G[1]+P[3]*P[2]*P[1]*G[0]+P[3]*P[2]*P[1]*P[0]*c0
end module

module cla4(a[3..0],cin,b[3..0]:s[3..0],cout)
    G[3..0] = a[3..0] * b[3..0]
    P[3..0] = a[3..0] + b[3..0]
    carry_lookahead(G[3..0],P[3..0],cin:cout,c3,c2,c1)
    xor3(a[3],b[3],c3:s[3])
    xor3(a[2],b[2],c2:s[2])
    xor3(a[1],b[1],c1:s[1])
    xor3(a[0],b[0],cin:s[0])
end module
```

Figure III-12. Écriture SHDL d'un additionneur 4 bits carry-lookahead. On notera l'écriture vectorielle pour les équations des termes G_i et P_i.

Association d'additionneurs carry-lookahead

La technique de l'additionneur carry-lookahead ne peut pas être utilisée au delà de 7 à 8 bits, mais on souhaite continuer à l'exploiter sur des additionneurs de plus grande taille. On peut subdiviser les mots à additionner en groupes de 4 bits par exemple, et effectuer dans chaque groupe une addition carry-lookahead. Se pose alors le problème de l'association de ces groupes : va-t-on se contenter de les chaîner en ripple carry ?
On peut en fait appliquer à nouveau la technique carry-lookahead à l'échelle des groupes. Chaque additionneur carry-lookahead sur 4 bits CLA_i va générer deux signaux GG_i(group generate) et GP_i (group propagate). GG_iindiquera que le groupe génèrera nécessairement une retenue et GP_i indiquera que si une retenue arrive dans ce groupe (par son entrée cin), elle sera propagée à travers lui au groupe suivant. La même méthode est ainsi appliquée à l'échelle des groupes, chaque groupe jouant ici le même rôle que jouait un étage de 1 bit dans l'additionneur carry-lookahead ordinaire. Le même module carry_lookahead est d'ailleurs employé pour calculer rapidement les retenues c4 , c8, c12 et c16 qui sont envoyées aux différents groupes. La Figure III-13 montre un exemple d'addition, et la Figure III-14 présente l'organisation générale d'un tel groupement.

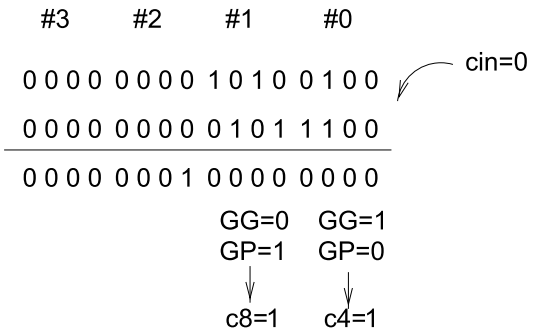


Figure III-13. Exemple d'addition 16 bits avec un additionneur group carry-lookahead. Le groupe #0 génère une retenue, qui est propagée au travers du groupe #1, et dont l'effet vient se terminer dans le groupe #2.

Il nous reste seulement à trouver les équations de *GG* et *GP*. *GG* indique qu'une retenue est générée par le groupe, c'est à dire générée an niveau du bit 3, ou générée au niveau du bit 2 et propagée au rang 3, etc. :

$$GG = G3 + G2 * P3 + G1 * P2 * P3 + G0 * P1 * P2 * P3$$

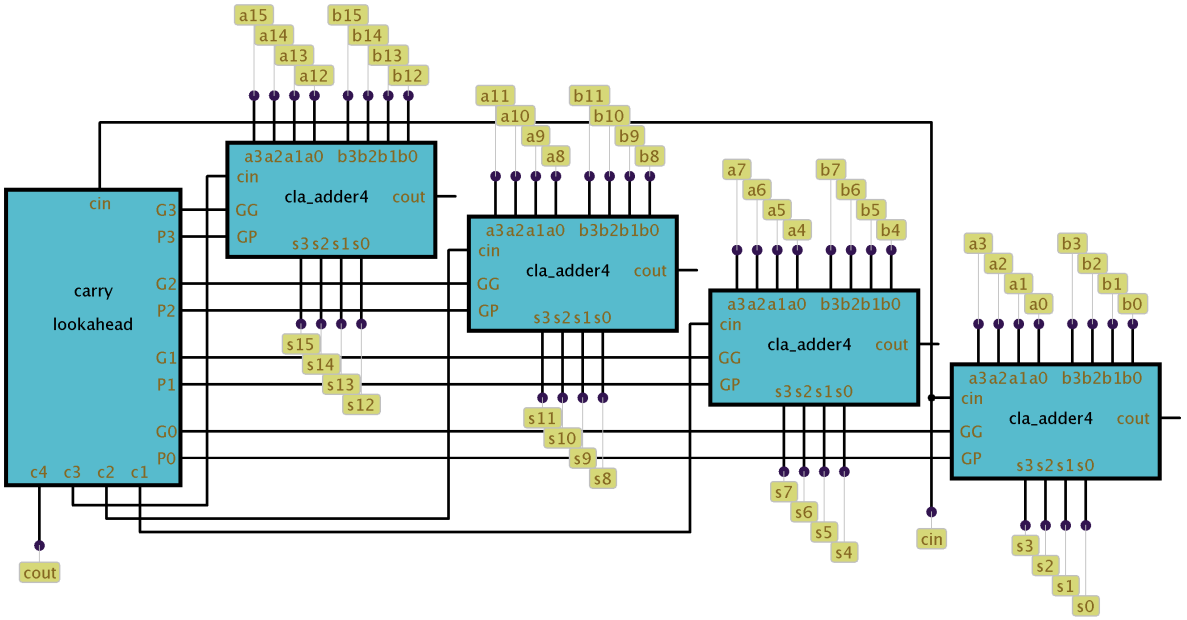


Figure III-14. Schéma général d'un groupement de 4 additionneurs 4 bits carry-lookahead. Chaque additionneur fournit deux signaux GG et GP qui indiquent s'il génère ou propage une retenue pour le groupe suivant. Un module carry_lookahead effectue le calcul rapide des retenues entrantes de chaque groupe à partir de ces signaux.

GP indique qu'une retenue arrivant par cin sera propagée tout au long du groupe, c'est à dire qu'à chaque rang de bit *i* il y a au moins un '1'sur *ai* ou *bi*, c'est à dire encore qu'à chaque rang *i* on a $Pi = 1$

$$GP = P0 * P1 * P2 * P3$$

La figure suivante donne l'ensemble complet des équations SHDL de cet additionneur. Les modules `carry_lookahead` et `fulladder` sont les même qu'à la figure II.35, `cla4` a été légèrement modifié puisqu'il produit maintenant les signaux GG et GP.

```
module cla16(a15..a0,b15..b0,cin:s15..s0,cout)
    cla4(a3..a0,b3..b0,cin:s3..s0,c4_,GG0,GP0)
    cla4(a7..a4,b7..b4,c4:s7..s4,c8_,GG1,GP1)
    cla4(a11..a8,b11..b8,c8:s11..s8,c12_,GG2,GP2)
    cla4(a15..a12,b15..b12,c12:s15..s12,c16_,GG3,GP3)
    carry_lookahead(GG3,GP3,GG2,GP2,GG1,GP1,GG0,GP0,cin:cout,c12,c8,c4)
end module

module cla4(a3,a2,a1,a0,b3,b2,b1,b0,cin:s3,s2,s1,s0,cout,GG,GP)
    G3..G0 = a3..a0 * b3..b0
    P3..P0 = a3..a0 + b3..b0
    carry_lookahead(G3,P3,G2,P2,G1,P1,G0,P0:cout,c2,c1,c0)
    full_adder(a3,b3,c2:s3,cout3)
    full_adder(a2,b2,c1:s2,cout2)
    full_adder(a1,b1,c0:s1,cout1)
    full_adder(a0,b0,cin:s0,cout0)
    GG = G3+G2*P3+G1*P2*P3+G0*P1*P2*P3
    GP = P0*P1*P2*P3
end module
```

Figure III-15. Écriture SHDL d'un additionneur 16 bits carry-lookahead. Les modules cla4, carry_lookahead et fulladder sont ceux utilisés dans la version 4 bits, cla4 ayant été légèrement modifié pour produire les signaux GG et GP.

III.4.5. Soustraction

Problématique de la soustraction

Comme l'addition, la soustraction peut être effectuée bit à bit, en commençant par le bit de poids faible.

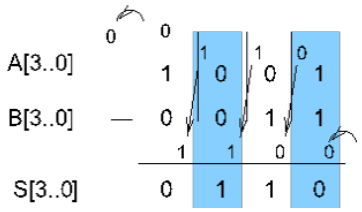


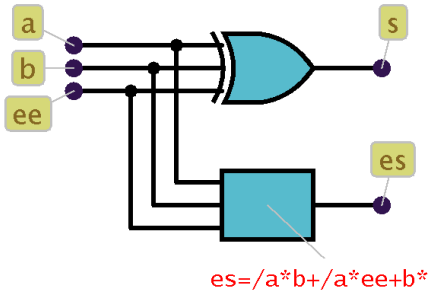
Figure III-16. Soustraction en binaire bit à bit. Un bit d'emprunt est propagé des bits de poids faibles vers les bits de poids forts.

Le bit qui est propagé de rang en rang est un bit d'emprunt : il vaut 1 lorsque ai est plus petit que *bi*, ou plus exactement lorsque *ai* est plus petit que *bi* plus le bit d'emprunt de l'étage précédent. Chaque rang de soustraction est un circuit combinatoire à 3 entrées et 2 sorties appelé ***soustracteur complet*** dont la table de vérité est donnéeFigure III-17. On a noté *EE* l'emprunt entrant, qui vient du rang précédent et *ES* l'emprunt sortant qui est passé au rang suivant.

A	B	*EE*	*S*	*ES*
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

Figure III-17. Table de vérité d'un soustracteur complet. On soustrait B à A ; EE est l'emprunt entrant et ES l'emprunt sortant.

On voit immédiatement que S est le XOR des trois entrées A,B et EE. L'emprunt sortant ES vaut 1 lorsque A vaut 0 et qu'un des deux B ou EE vaut 1, ou lorsque A vaut 1 et que B et EE valent tous les deux 1. On trouve donc : $ES = \bar{A} \cdot B + \bar{A} \cdot EE + EE \cdot B$
Le schéma du soustracteur complet et son écriture SHDL sont donnés Figure III-18.




```
module fullsub(A,B,EE: S,ES)
  S = /A*/B*EE+/A*B*/EE+A*/B*/EE+A*B*EE
  ES = /A*B + /A*EE + B*EE
end module
```

Figure III-18. Soustracteur complet : la différence est un XOR, et il y a retenue lorsqu'une majorité des signaux (/A,B,EE) vaut 1.

Soustracteur ripple-borrow

Comme un additionneur ripple-carry, un soustracteur ripple-borrow est formé par chaînage de plusieurs étages de soustracteurs complets. Comme lui, il est très simple de conception, mais présente aussi les mêmes inconvénients de lenteur dus à la propagation des signaux d'emprunt. Les figures suivantes montrent un additionneur 4 bits ripple-borrow et l'écriture SHDL associée.

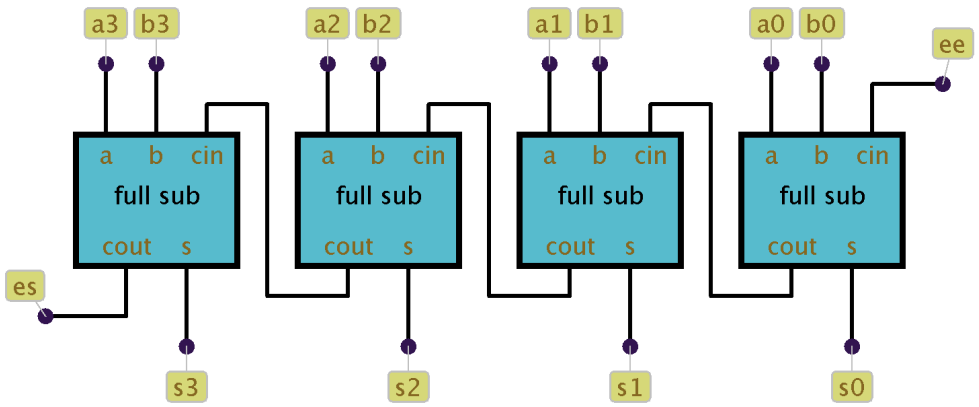


Figure III-19. Schéma général d'un soustracteur 4 bits ripple-borrow.

Transformation d'un additionneur en soustracteur

On peut faire la soustraction A − B en effectuant l'addition A + (− B). − B est obtenu en prenant le complément à 2 de B, c'est à dire le complément à 1 de B auquel on ajoute 1. L'ajout +1 peut se faire en exploitant un signal de retenue entrante, comme on peut le voir sur la figure suivante :

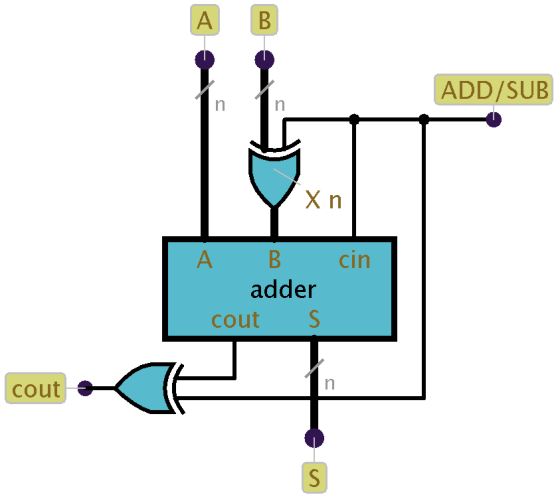


Figure III-20. Additionneur/soustracteur réalisé à partir d'un additionneur. Lorsque l'entrée ADD/ SUB vaut 0 l'addition A + B est réalisée ; lorsqu'elle vaut 1, les XORs calculent le complément à 1 de B et le forçage à 1 de la retenue entrante cin créé le complément à 2, d'où la soustraction.

Cette figure montre un additionneur/soustracteur : si ADD/SUB vaut 0 il effectue une addition ; s'il vaut 1 les n XORs effectuent le complément à 1 du vecteur B, et le forçage à 1 de la retenue entrante va conduire à ajouter 1, donc à construire le complément à 2 de B, donc à réaliser une soustraction. On perd malheureusement le signal de retenue entrante ; la retenue sortante cout par contre a bien le sens d'une retenue pour l'addition, et son inverse le sens d'un emprunt, d'où l'inversion finale commandée par le signal ADD/SUB.
(démonstration que Borrow = /Carry : si A>=B, Borrow=0, or /B+1 = 2^n - B, donc A+/B+1 = A -B + 2^n >= 2^n donc Carry=1 et même raisonnement pour A<B qui donne Borrow=1 et C=0)
La Figure III-21 donne le code SHDL associé pour un additionneur/soustracteur 16 bits ; elle reprend le module additionneur carry-lookahead 16 bits cla16 étudié à la section précédente.

```
module addsub16(a[15..0],b[15..0],addsub: s[15..0],cout)
   bb[15..0] = /addsub*b[15..0] + addsub*/b[15..0]
   cla16(a[15..0],bb[15..0],addsub:s[15..0],co)
   cout=/addsub*co+addsub*/co
end module
```

Figure III-21. Équations SHDL d'un additionneur/soustracteur 16 bits. On réutilise l'additionneur carry-lookahead 16 bits étudié auparavant.

III.4.6. Multiplicateur systolique

Problématique de la multiplication

On souhaite multiplier deux nombres de n bits non signés. Le résultat est sur 2n bits, puisque la multiplication des deux plus grands nombres sur n bits donne :

$$(2^n - 1)(2^n - 1) = 2^{2n} - 2 \cdot 2^n + 1 < 2^{2n} - 1$$

La problématique est la même que pour l'addition : puisqu'il s'agit d'un calcul combinatoire, il peut se faire en théorie sous forme d'une somme de minterms, en une étape de propagation (en considérant une fois encore que l'unité de propagation est la somme de produit). Mais bien sûr en pratique, les équations à implémenter seraient beaucoup trop complexes. En binaire, la méthode la plus directe pour effectuer une multiplication consiste à la poser comme à l'école :

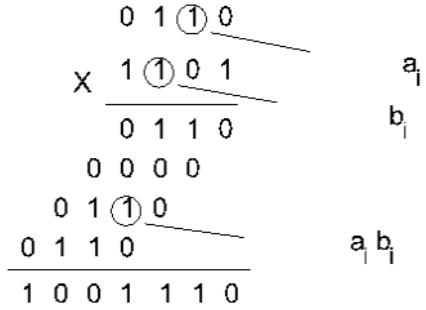


Figure III-22. Exemple de multiplication de nombres non signés de 4 bits.

Cette opération est plus facile à faire en binaire qu'en décimal, car il n'y a pas à connaître ses tables de multiplication ! Plus précisément, lorsqu'on multiplie un chiffre a_i avec un chiffre b_i, on produit (c'est le cas de le dire) $a_i \,.\, b_i$ (Figure III-23). Il reste ensuite à faire la somme des produits partiels ainsi obtenus.

Quand on fait cette somme à la main, on la fait colonne par colonne, en commençant par la colonne la plus à droite, et on additionne à la fois tous les chiffres qui apparaissent dans une colonne et la ou les retenues qui proviennent de la colonne précédente. Il peut en effet y avoir une retenue supérieure à 1 lors de la sommation des chiffres d'une colonne, contrairement au cas de l'addition.

L'idée du ***multiplicateur systolique*** (matriciel) consiste à réaliser cette somme des produits partiels dans une matrice de cellules qui a la même forme que les rangées à ajouter (figure II.47).

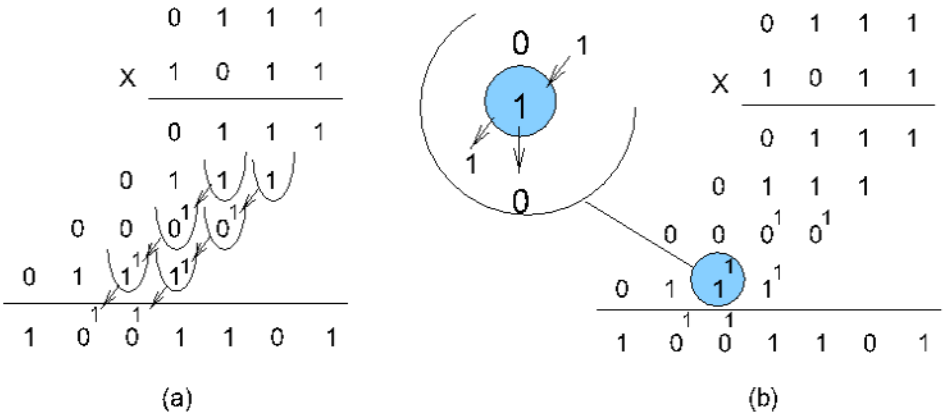


Figure III-23. (a) Les retenues peuvent être propagées à un niveau local, et non à l'échelle de toute la colonne. (b) entrées et sorties au niveau d'une cellule.

Chaque fois qu'une cellule produit une retenue, elle est passée directement à la colonne immédiatement à gauche, un rang plus bas (situation (a)). Cette technique permet de ne pas laisser les retenues s'accumuler à chaque colonne, en les traitants immédiatement à un niveau local. Chaque cellule doit faire l'addition de trois termes : le produit partiel $a_i \,.\, b_i$, la somme du niveau précédent, et la retenue qui provient de la cellule en haut à droite (situation (b)). Un additionneur complet permet d'effectuer cette somme, qui est passée à la cellule immédiatement en bas, et de calculer une retenue à passer à la cellule en bas à gauche (Figure III-24).

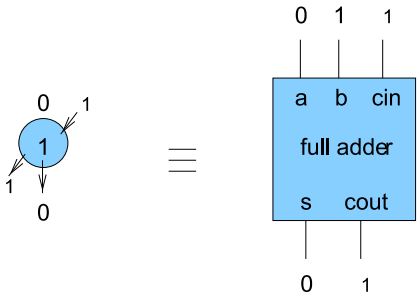


Figure III-24. Multiplicateur systolique : chaque cellule est un additionneur complet.

Le schéma général d'un tel multiplicateur est donné Figure III-25. Les cellules ont la même disposition que la somme des termes partiels, avec un recadrage à droite. On remarquera la dernière rangée, qui fournit les bits de poids forts du résultat final $s_7 . . s_4$. Les bits de poids faibles $s_3 . . s_0$ proviennent quant à eux des cellules de la colonne de droite.

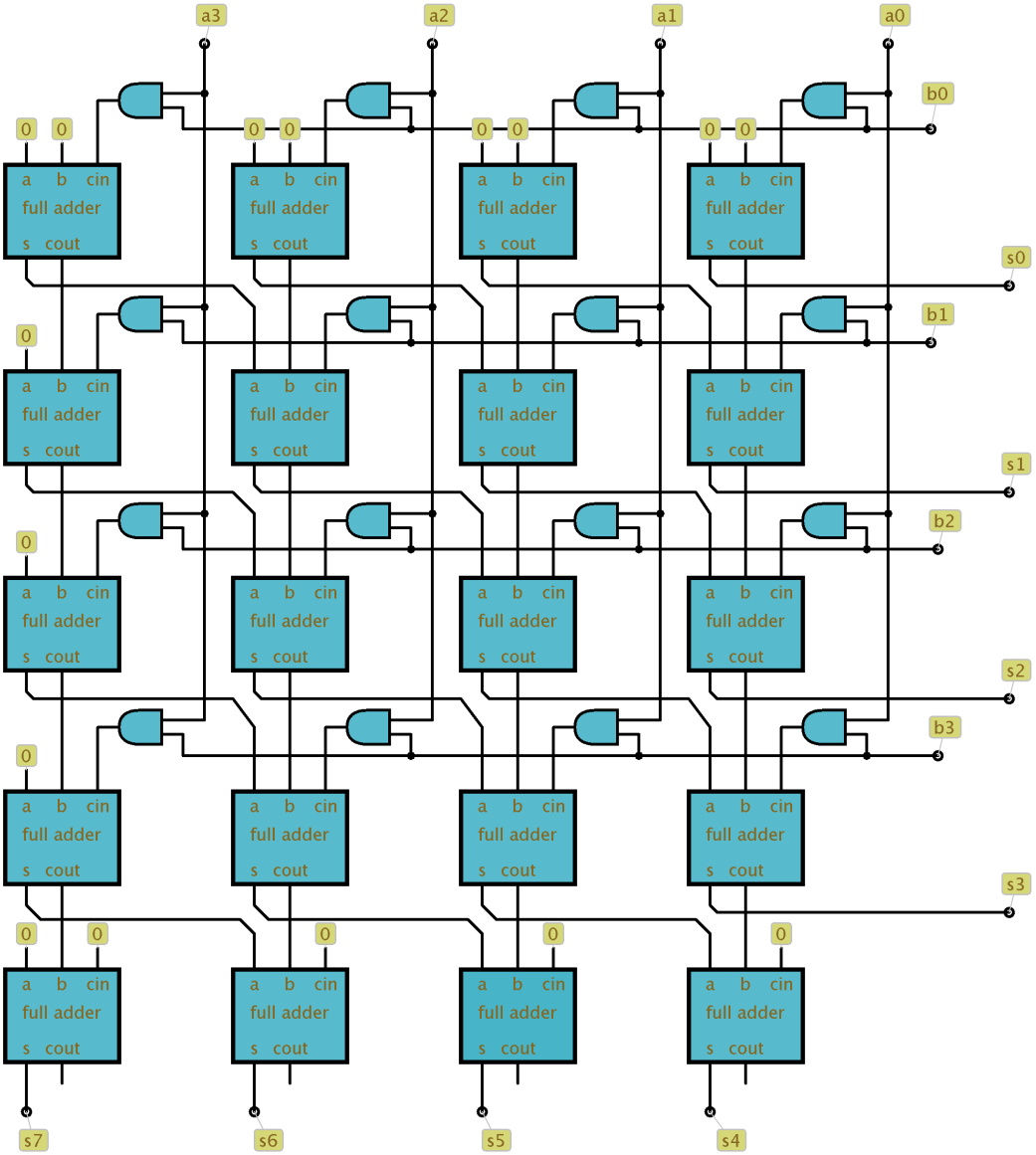


Figure III-25. Schéma général d'un multiplicateur systolique 4 bits x 4 bits vers 8 bits.

Le texte SHDL correspondant à ce schéma est le suivant :

```
module multsyst(a[3..0], b[3..0]: s[7..0])

    // première rangée ; produit s0
    p30=a[3]*b[0]; p20=a[2]*b[0]; p10=a[1]*b[0]; p00=a[0]*b[0]
    fulladder(0,0,p30:s30,c30)
    fulladder(0,0,p20:s20,c20)
    fulladder(0,0,p10:s10,c10)
    fulladder(0,0,p00:s[0],c00)

    // deuxième rangée ; produit s1
    p31=a[3]*b[1]; p21=a[2]*b[1]; p11=a[1]*b[1]; p01=a[0]*b[1]
    fulladder(c30,0,p31:s31,c31)
    fulladder(c20,s30,p21:s21,c21)
    fulladder(c10,s20,p11:s11,c11)
    fulladder(c00,s10,p01:s[1],c01)

    // troisième rangée ; produit s2
    p32=a[3]*b[2]; p22=a[2]*b[2]; p12=a[1]*b[2]; p02=a[0]*b[2]
    fulladder(c31,0,p32:s32,c32)
    fulladder(c21,s31,p22:s22,c22)
    fulladder(c11,s21,p12:s12,c12)
    fulladder(c01,s11,p02:s[2],c02)

    // quatrième rangée ; produit s3
    p33=a[3]*b[3]; p23=a[2]*b[3]; p13=a[1]*b[3]; p03=a[0]*b[3]
    fulladder(c32,0,p33:s33,c33)
    fulladder(c22,s32,p23:s23,c23)
    fulladder(c12,s22,p13:s13,c13)
    fulladder(c02,s12,p03:s[3],c03)

    // rangée du total final pour s7..s4
    fulladder(0,c33,0:s[7],c3x)
    fulladder(s33,c23,0:s[6],c2x)
    fulladder(s23,c13,0:s[5],c1x)
    fulladder(s13,c03,0:s[4],c0x)
end module
```

Figure III-26. Écriture SHDL d'un multiplicateur systolique 4 bits x 4 bits.

On voit facilement que ce multiplicateur 4x4 nécessite 6 temps de propagation (unité : somme de termes). De façon plus générale, un multiplicateur systolique n bits x n bits nécessite n+2 temps de propagation. C'est donc une méthode assez efficace, qui est facile à implémenter dans des circuits matriciels tels que CPLDs et FPGAs.

III.4.7. Division entière

Problématique de la division

Comme pour la multiplication, on va essayer de concevoir un réseau de cellules qui réalise une division non signée en utilisant la méthode euclidienne. Prenons un exemple :

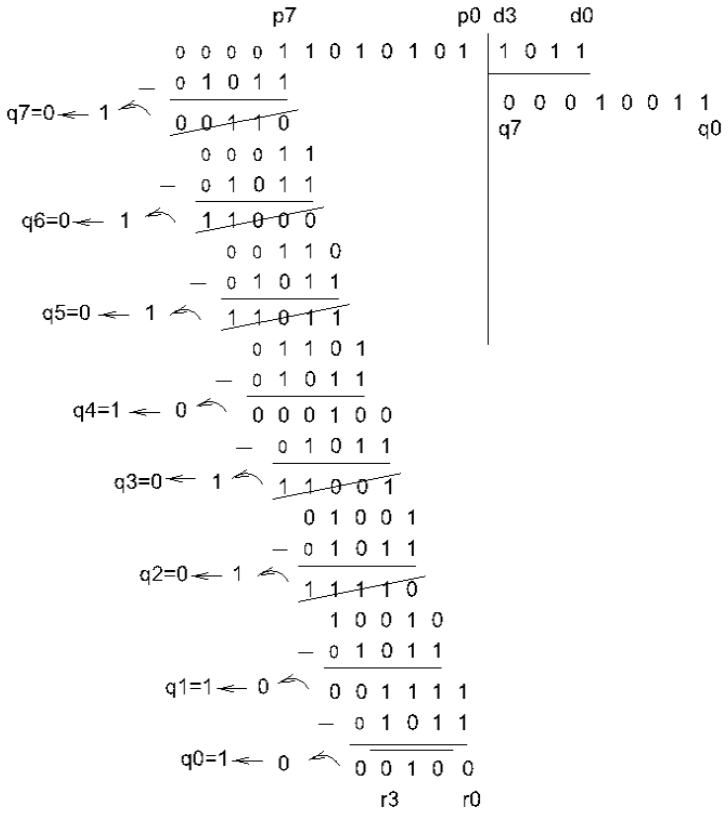


Figure III-27. Exemple de division euclidienne en binaire.

On a réalisé ici une division non signée 8 bits / 4 bits -> 8 bits. Le dividende est $p_7..p_0$ = 1101010101_2 = 213 et le diviseur est $d_3..d_0$ = 1011_2 = 11; on trouve un quotient $q_7..q_0$ = 00010011_2 = 19 et un reste $r_3..r_0$ = 0100_2 = 4.
On voit que le nombre d'étages de cette division est égal à la largeur du dividende, 8 dans l'exemple. À chaque étape, on effectue une soustraction entre un terme courant et le diviseur. Au départ, ce terme courant est le poids fort du dividende complété de 4 zéros à gauche. Le diviseur est également complété d'un zéro à gauche et c'est une soustraction sur 5 bits qui est réalisée. Il y a alors deux cas :

- la soustraction donne un résultat négatif, signalé par le bit d'emprunt à 1; on ne doit alors pas prendre son résultat pour le terme courant, mais le laisser tel quel. On lui rajoutera à droite un bit supplémentaire du dividende, à l'étage suivant. On ajoute un zéro à droite au quotient.
- la soustraction donne un résultat positif, signalé par le bit d'emprunt à 0; on remplace le terme courant par le résultat de cette soustraction, et on lui ajoutera à droite un bit supplémentaire du dividende, à l'étage suivant. On ajoute un 1 à droite au quotient.

Le reste $r_3..r_0$ est la valeur du terme courant après la dernière étape.

Structure du diviseur

On souhaite ainsi réaliser un diviseur 2n bits / n bits -> 2n bits; on va prendre ici comme dans l'exemple n = 4, mais la technique est bien sûr généralisable à toute valeur de n. La figure suivante montre la structure nécessaire du diviseur.

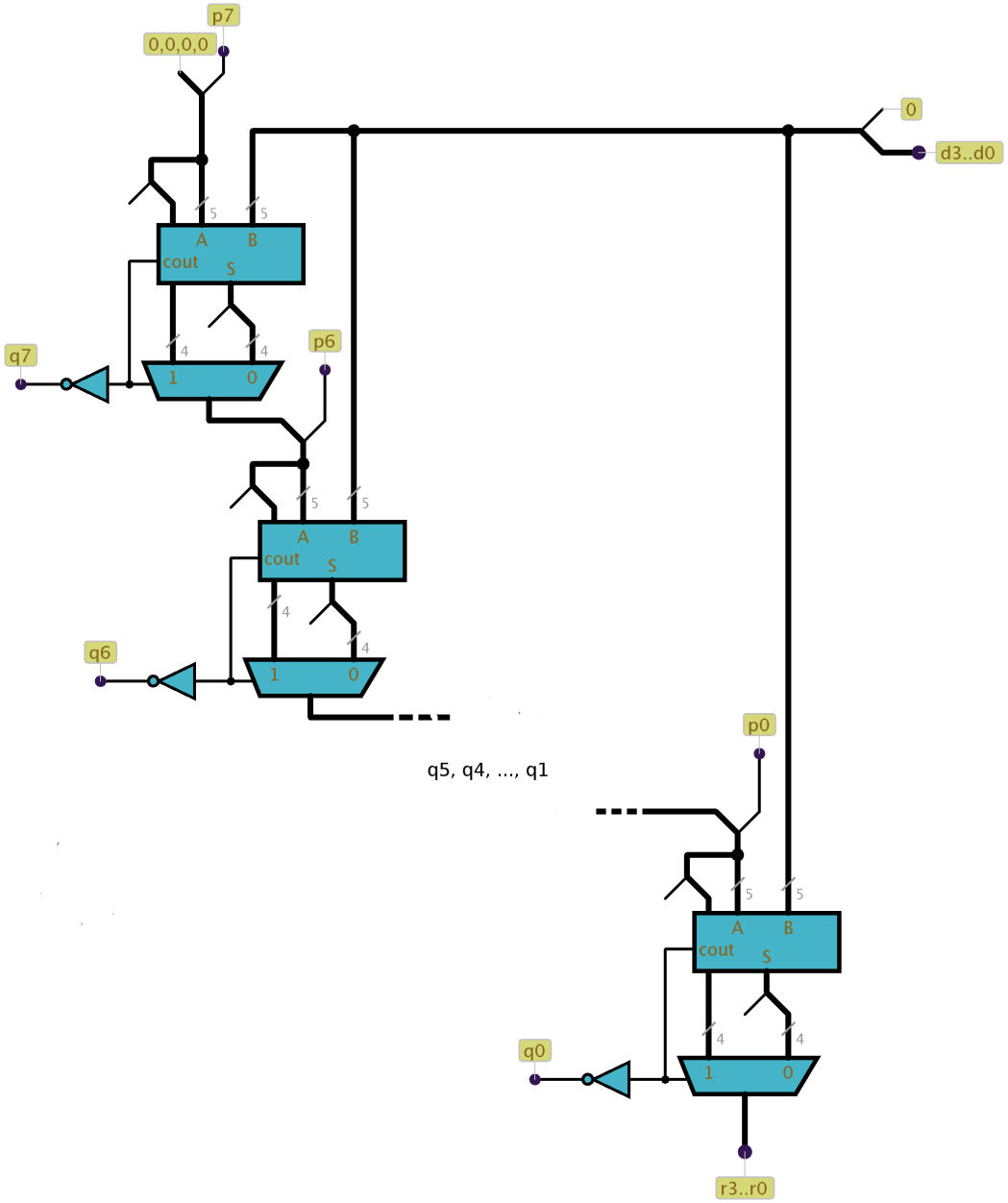


Figure III-28. Structure du diviseur non signé 8 bits / 4 bits -> 8 bits. À chaque étage on teste si on peut soustraire le diviseur au terme courant. Les bits du quotient sont les inverses des bits d'emprunt des soustractions; le reste est la valeur du terme courant après la dernière étape.

À chaque étage on réalise la soustraction entre le terme issu de l'étage précédent et $d_3..d_0$. On utilise pour cela des soustracteurs 5 bits sans retenue entrante, et avec une retenue sortante; on a vu aux sections précédentes comment les réaliser.
À chaque étage il faut aussi construire le terme courant, en fonction des deux cas examinés dans la discussion précédente et déterminés par la valeur du bit d'emprunt de la soustraction. On devra pour cela utiliser des multiplexeurs dont la commande sera le bit d'emprunt.
La figure suivante donne le code SHDL complet associé à ce diviseur.

```
module div8(p[7..0],d[3..0]: q[7..0],r[3..0])
    // soustracteur
    sub5(0,0,0,0,p[7],0,d[3],d[2],d[1],d[0]:x74,x73,x72,x71,x70,nq7)
    q[7]=/nq7
    // multiplexeur
    s74=nq7*z+/nq7*x73
    s73=nq7*z+/nq7*x73
    s72=nq7*z+/nq7*x72
    s71=nq7*z+/nq7*x71
    s70=nq7*p7+/nq7*x70
    sub5(s73,s72,s71,s70,p[6],0, d[3],d[2],d[1],d[0]:x64,x63,x62,x61,x60,nq6)
    q[6]=/nq6
    s64=nq6*s73+/nq6*x64
    s63=nq6*s72+/nq6*x63
    s62=nq6*s71+/nq6*x62
    s61=nq6*s70+/nq6*x61
    s60=nq6*p6+/nq6*x60
    ...
    // code analogue pour q4,...,q1
    ...
    sub5(s13,s12,s11,s10,p0,0, d[3],d[2],d[1],d[0]:x04,x03,x02,x01,x00,nq0)
    q[0]=/nq0
    s04=nq0*s13+/nq0*x04
    r[3]=nq0*s12+/nq0*x03
    r[2]=nq0*s11+/nq0*x02
    r[1]=nq0*s10+/nq0*x01
    r[0]=nq0*p0+/nq0*x00
end module
```

Figure III-29. Écriture SHDL d'un diviseur non signé 8 bits / 4 bits -> 4 bits.

III.4.8. Comparateurs

L'opération de ***comparaison*** entre nombres entiers est fondamentale dans un ordinateur. Si on souhaite comparer deux nombres de n bits, l'interface générale qu'on peut utiliser est celle de la Figure III-30.

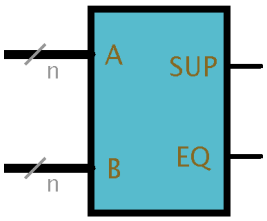


Figure III-30. Interface d'un comparateur sur n bits. Une sortie indique que A = B, une autre indique que A > B.

Une sortie SUP indique que A > B, l'autre EQ indique que A = B. Avoir ces deux sorties permet de tout savoir sur les positions respectives de A et B :

- $A > B$: SUP = 1.
- $A \geq B$: SUP = 1 ou EQ = 1.
- $A = B$: EQ = 1
- $A < B$: SUP = 0 et EQ = 0.
- $A \leq B$: SUP = 0.

Il faut maintenant savoir si les nombres que l'on compare sont des nombres entiers naturels, ou des nombres relatifs codés en complément à 2. En effet, si A = 1111 et B = 0101, une comparaison non signée va donner A > B (car A = 15 et B = 5), alors qu'une comparaison signée donnera A < B (car A = −1 et B = + 5).

Comparateur non signé

Supposons que A et B soient codés en binaire pur, par exemple sur 4 bits. On commence d'abord par comparer leurs poids forts A_3 et B_3 ; si celui de A est plus grand que celui de B, on peut conclure SUP = 1 ; sinon SUP ne peut valoir 1 que si $A_3 = B_3$ et que si $A_2..A_0 > B_2..B_0$. On a ainsi une définition récursive de SUP :

$$SUP = (A3 > B3) + (A3 = B3) \cdot (A2..A0 > B2..B0)$$

$$SUP = (A3 > B3) + (A3 = B3) \cdot ((A2 > B2) + (A2 = B2) \cdot (A1..A0 > B1..B0))$$

$$SUP = (A3 > B3) + (A3 = B3) \cdot ((A2 > B2) + (A2 = B2) \cdot ((A1 > B1) + (A1 = B1) \cdot (A0 > B0)))$$

$A_i > B_i$ correspond en fait à l'unique situation $A_i = 1$ et $B_i = 0$ et s'exprime donc $A_i - B_i$. Pour $A_i = B_i$, on peut utiliser le fait qu'un XOR est un opérateur de différence, et donc que son inverse est un opérateur de coïncidence : $(A_i = Bi) = \overline{A_i} \oplus B_i = A_i \cdot B_i + \overline{A_i} \cdot \overline{B_i}$

$$SUP = A_3\overline{B_3} + \overline{A_3 \oplus B_3} \cdot (A_2\overline{B_2} + A_2 \oplus B_2 \cdot (A_1\overline{B_1} + \overline{A_1} \oplus B_1 \cdot A_0\overline{B_0}))$$

Le code SHDL correspondant est :

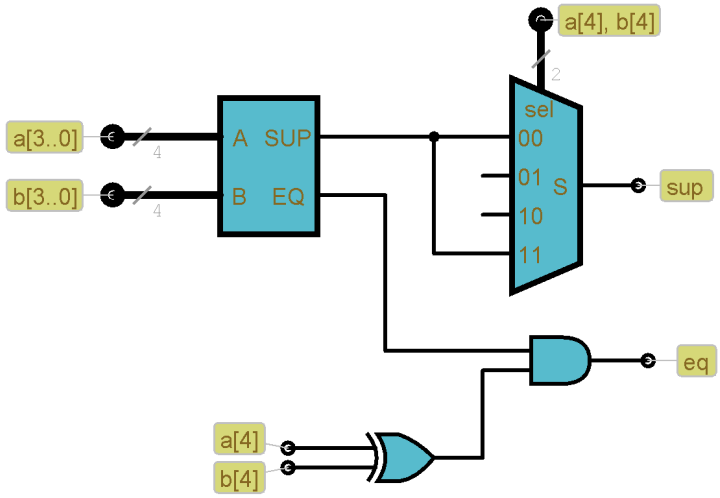
```
module cmpu4(a[3..0], b[3..0]: sup, eq)
  sup=a[3]*/b[3]+eq3*sup2
  eq3=a[3]*b[3]+/a[3]*/b[3]
  sup2=a[2]*/b[2]+eq2*sup1
  eq2=a[2]*b[2]+/a[2]*/b[2]
  sup1=a[1]*/b[1]+eq1*a[0]*/b[0]
  eq1=a[1]*b[1]+/a[1]*/b[1]
  eq0=a[0]*b[0]+/a[0]*/b[0]
  eq=eq3*eq2*eq1*eq0
end module
```

Comparateur signé

Une comparaison signée est un peu plus délicate qu'une comparaison non signée ; l'essentiel de la comparaison se joue en examinant les signes a_{n-1} et b_{n-1} des termes $A = a_{n-1}..a_0$ et $B = b_{n-1}..b_0$ à comparer :

- si A < 0 ($a_{n-1} = 1$) et B > 0 ($b_{n-1} = 0$) alors la comparaison est immédiate : SUP = 0 et EQ = 0.
- si A > 0 ($a_{n-1} = 0$) et B < 0 ($b_{n-1} = 1$) alors la comparaison est immédiate : SUP = 1 et EQ = 0.
- si A > 0 ($a_{n-1} = 0$) et B > 0 ($b_{n-1} = 0$) alors il suffit de comparer les n − 1 bits de poids faibles.
- si A < 0 ($a_{n-1} = 1$) et B < 0 ($b_{n-1} = 1$) alors il suffit également de comparer les n − 1bits de poids faibles. En effet, les nombres de l'intervalle $[2_{n-1},0]$ s'écrivent en complément à 2 : [1000..000, 1000..001, 1111..111] et on voit que les n − 1 bits de poids faible évoluent de façon croissante.

On a représenté sur la Figure III-31 un comparateur signé sur 5 bits utilisant un compteur non signé de 4 bits. Un multiplexeur 4 vers 1 sépare les 4 situations décrites.

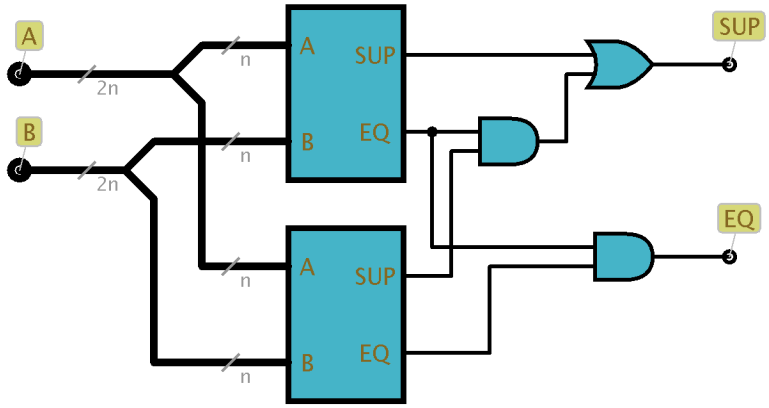



```
module cmps5(a[4..0],b[4..0]: sup, eq)
  cmpu4(a[3..0], b[3..0]: sup1, eq1)
  mux4(sup1, 1, 0, sup1, a[4], b[4]: sup)
  eq= a[4]*b[4]*eq1+/a[4]*/b[4]*eq1
  eq=eq4*eq1
end module
```

Figure III-31. Comparateur signé sur 5 bits. Si les signes a4 et b4 sont différents, la comparaison est immédiate ; s'ils sont égaux on compare en non signé les 4 bits de poids faibles.

Assemblage de comparateurs non signés

Il est facile de chaîner 2 comparateurs non signés de n et m bits pour former un comparateur non signé de n + m bits (figure II.57).




```
module cmpu(a[9..0], b[9..0]: sup, eq)
  cmpu5(a[9..5], b[9..5]: sup1, eq1)
  cmpu5(a[4..0], b[4..0]: sup2,eq2)
  eq=eq1*eq2
  sup=sup1+eq1*sup2
end module
```

Figure III-32. Association de deux comparateurs n bits non signés. Le résultat forme un comparateur 2n bits avec le même interface.

On compare d'abord les n bits de poids forts ; si ceux de A sont plus grands que ceux de B, le SUP final est asserté au travers du OU. Si les poids forts de A et ceux de B sont égaux, on compare les m bits de poids faibles et on conclue. La sortie finale EQ est quant à elle assertée

lorsque les deux sorties EQ sont assertées. On a représenté en figure II.58 le code SHDL d'un comparateur non signé sur 10 bits, formé par l'assemblage de deux comparateurs non signés de 5 bits.
La sortie de module EQ a été essentielle ici pour permettre d'associer les circuits. Par ailleurs, le nouveau module formé expose le même interface que ses parties : A et B en entrées, et SUP et EQ en sorties, ce qui permet à nouveau et récursivement d'associer ces circuits. Comme dans les disciplines logicielles de l'informatique, il est intéressant de trouver la bonne interface à un module, qui va permettre une bonne réutilisation.

III.5. Exercices corrigés

Exercice 1 : conversions

Écrire 54321 en binaire, puis en hexadécimal.

Solution

On peut se reporter à la table donnée en annexe A pour obtenir la liste des premières puissances de 2. La plus grande qui soit inférieure à 54321 est $2^{15} = 32768$. On peut écrire alors :

$54321 = 2^{15} + 21553$.

On recommence avec 21553 : il contient $2^{14} = 16384$, et il reste 21553 - 16384 = 5169, donc :

$54321 = 2^{15} + 2^{14} + 5169$.

5169 contient $2^{12} = 4096$, reste 1073 :

$54321 = 2^{15} + 2^{14} + 2^{12} + 1073$.

1073 contient $2^{10} = 1024$, reste 49 :

$54321 = 2^{15} + 2^{14} + 2^{12} + 2^{10} + 49$.

En continuant ce processus, on trouve :

$54321 = 2^{15} + 2^{14} + 2^{12} + 2^{10} + 2^5 + 2^4 + 2^0$.

L'écriture binaire est alors :

$54321 = 1101010000110001_2$

Pour trouver l'écriture hexadécimale, on regroupe les bits par paquets de 4 :

$54321 = 1101.0100.0011.0001_2$

On convertit ensuite chaque groupe en un chiffre hexadécimal équivalent :

$54321 = D431_{16}$

Exercice 2 : addition

1. Donner l'intervalle des valeurs possibles pour des nombres non signés codés en binaire pur sur 11 bits.
1. Donner l'intervalle des valeurs possibles pour des nombres signés codés en complément à 2 sur 11 bits.
2. Effectuer la somme en binaire : 11012 + 01112 et interpréter le résultat en arithmétique signée et non signée.

Solution

1. Intervalle des valeurs possibles en binaire pur sur 11 bits.

Avec 11 bits, le nombre de combinaisons possibles est $2^{11} = 2048$. L'intervalle des valeurs est donc : [0, 2047].

2. Intervalle des valeurs possibles en complément à 2 sur 11 bits.

On a vu dans le chapitre précédent que le codage en complément à 2 était équilibré, c'est à dire qu'il code autant de nombres strictement négatifs que de nombres positifs ou nuls. Pour respecter cet équilibre avec 2048 codes, il nous faut nécessairement 1024 codes positifs ou nuls et 1024 codes strictement négatifs, soit l'intervalle : [-1024, +1023].

3. Somme binaire $1101_2 + 0111_2$.

En effectuant la somme bit à bit, on trouve 0100_2 avec 1 de retenue sortante.
En arithmétique non signée, la retenue doit être prise en compte, et elle signale un débordement ; on peut tout de même intégrer ce bit de retenue comme bit de poids fort du résultat. On a donc fait la somme 13 + 7 et on trouve 10100_2, soit la valeur attendue 20.
En arithmétique en complément à 2, la retenue est ignorée. Les nombres ajoutés sont -3 et +7 et le résultat est +4, ce qui est également correct. Un débordement était impossible ici, car on ajoutait un nombre positif à un nombre négatif.

Exercice 3 : simplifications algébriques

Simplifier algébriquement :

1. $F_1 = A\bar{B}C + \bar{A}C + A\bar{B} + \bar{A}\,B\,C$
2. $F2 = AB\bar{C}\bar{D} + ABD + AC + BC + \bar{A}C$
3. $F3 = A + \bar{B}C + AD\bar{E} + ABCDE + B\bar{C}D\bar{E} + \bar{A}CD$

Solution

1. 1. $F_1 = A\bar{B}C + \bar{A}C + A\bar{B} + \bar{A}BC$

Ici des termes disparaissent car ils sont plus spécifiques que d'autres. Par exemple $A\bar{B}C$ est plus spécifique que $A\bar{B}$ et le premier disparaît au profit du deuxième. De la même façon, $\bar{A}BC$ est plus spécifique que $\bar{A}C$ et disparaît. On a :
$F_1 = \bar{A}C + A\bar{B}$

2. $F2 = AB\bar{C}\bar{D} + ABD + AC + BC + \bar{A}C$

$F2 = AB(\bar{C}\bar{D} + D) + AC + BC + \bar{A}C$
Avec le théorème d'absorption, $\bar{C}\bar{D} + D = \bar{C} + D$, donc :
$F2 = ABD + AB\bar{C} + AC + BC + \bar{A}C$
Par ailleurs, $+\bar{A}C = C$, donc :
$F2 = ABD + AB\bar{C} + BC\,C$
$F2 = ABD + AB\bar{C} + C$
On peut encore utiliser le théorème d'absorption sur $AB\bar{C} + C$:
$F2 = ABD + AB + C$
$F2 = AB + C$

3. $F3 = A + \bar{B}C + AD\bar{E} + ABCDE + B\bar{C}D\bar{E} + \bar{A}CD$

A inclue $AD\bar{E}$ et , donc :
$F3 = A + \bar{B}C + B\bar{C}D\bar{E} + CD$ (absorption entre A et $\bar{A}CD$)
$F3 = A + \bar{B}C + D(B\bar{C}\bar{E} + C)$
$F3 = A + \bar{B}C + D(B\bar{E} + C)$ (absorption de $\bar{C}$)
$F3 = A + \bar{B}C + BD\bar{E} + CD$
Avec des tables de Kanaugh, des simplifications non adjacentes donneraient le même résultat.

Exercice 4 : circuit incrémenteur

Concevoir un circuit combinatoire prenant en entrée un nombre binaire $a_3 \ldots a_0$ et donnant en sortie $b_3 \ldots b_0$ la valeur d'entrée incrémentée de 1. Écrire le code SHDL correspondant.

Solution

L'algorithme de base de l'incrémentation a été donné en section 4.Le bit de poids faible b_0 est toujours inversé par rapport à a_0. Ensuite, b_n est l'inverse de a_n si et seulement si tous les bits qui sont à la droite de a_n valent 1, soit lorsque $a_{n-1} \ldots a_0 = 1$. On a donc une inversion commandée par cette valeur, et on a vu en section 4 que le XOR est la porte adaptée à cette opération :
$b_n = a_n \oplus (a_{n-1} \ldots a_0)$

Finalement, sur 4 bits :

$b_0 = a_0$

$b_1 = a_1 \oplus a_0$

$b_2 = a_1 \oplus (a_1 a_0)$

$b_3 = a_1 \oplus (a_2 a_1 a_0)$

Exprimé sous forme de sommes de termes :

$b_0 = a_0$

$b_1 = a_1\overline{a_0} + \overline{a_1}a_0$

$b_2 = \overline{a_2}a_1a_0 + a_2\overline{a_1}a_0 = \overline{a_2}a_1a_0 + a_2(\overline{a_1} + \overline{a_0}) = \overline{a_2}a_1a_0 + a_2\overline{a_1} + a_2\overline{a_0}$

$b_3 = \overline{a_3}a_2a_1a_0 + a_3\overline{a_2}a_1a_0 = \overline{a_3}a_2a_1a_0 + a_3\overline{a_2} + a_3\overline{a_1} + a_3\overline{a_0}$

Finalement, le module SHDL correspondant est :

```
module incr(a[3..0]: b[3..0])
  b[0]=/a[0]
  b[1]=/a[1]*a[0]+a[1]*/a[0]
  b[2]=/a[2]*a[1]*a[0]+a[2]*/a[1]+a[2]*/a[0]
  b[3]=/a[3]*a[2]*a[1]*a[0]+a[3]*/a[2]+a[3]*/a[1]+a[3]*/a[0]
end module
```

Chapitre IV. Elements de logique séquentielle

IV.1. Définition

Un circuit est dit ***séquentiel***, si les valeurs de ses sorties ne dépendent pas que des valeurs de ses entrées.
C'est donc le contraire d'un circuit combinatoire, dont les valeurs des sorties ne dépendaient que de celles de ses entrées, avec une propagation sans retour arrière des signaux des entrées vers les sorties.
À l'inverse, les sorties d'un circuit séquentiel dépendent non seulement des valeurs des entrées au moment présent, mais aussi d'un état interne qui dépend de l'historique des valeurs d'entrées précédentes. En vertu de ce que nous avions démontré pour les circuits combinatoires, cela implique un rebouclage vers l'arrière de certains signaux internes au circuit (**Erreur ! Source du renvoi introuvable.**).

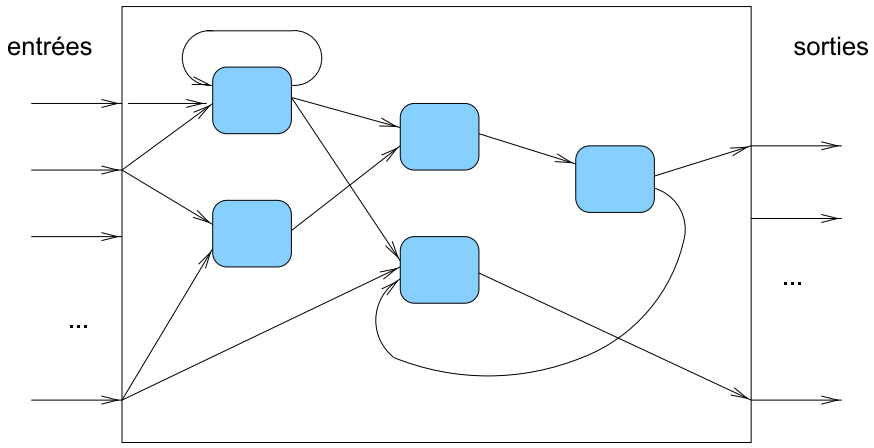
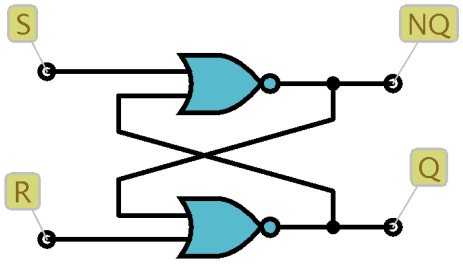


Figure III-33. Circuit séquentiel : certains signaux internes ou sorties rebouclent en arrière.

Ce type de circuit peut donner lieu à des fonctionnements très complexes ; il peut également être instable dans certaines configurations. Plus encore que pour les circuits combinatoires, des méthodes pour maîtriser leur complexité sont nécessaires. Une des voies pour la simplification de leur conception consiste à synchroniser les changements d'états sur les fronts d'une horloge unique et globale, et cela conduit à l'étude des circuits séquentiels dits synchrones purs, qu'on étudiera principalement dans ce livre.

IV.2. Latch RS

La notion de circuit séquentiel avec ses états internes incorpore la notion de mémoire (retour sur son passé).La cellule mémoire la plus simple est le latch RS (RS étant les initiales de Reset-Set), parfois appelé bistable (Figure III-34).




```
module latch_rs(s,r: q)
   q=/nq*/r
   nq=/q*/s
end module
```

Figure III-34. Latch RS.

Ce circuit a clairement des connexions qui reviennent en arrière. Il s'agit d'une cellule de mémorisation de 1 bit, dont le mode d'emploi est le suivant :

1. la cellule est en mode 'lecture' lorsque les entrées R et S sont toutes les deux à 0 ; la paire $Q, \bar{Q}$ est alors nécessairement dans un des deux états (0, 1) ou (1, 0), car on peut constater que ces deux états, et seulement eux, s'auto-entretiennent lorsque R et S valent 0.
2. pour mémoriser 0 (c'est à dire que les sorties soient dans l'état $Q, \bar{Q} = (0, 1)$ lorsque la cellule reviendra dans le mode 'lecture'), il faut appliquer un 1 sur R (reset), puis le remettre à 0. Quelqu'ait été son état antérieur, la cellule se retrouve alors dans l'état auto-stable $Q, \bar{Q} = (0, 1)$.
3. pour mémoriser 1 (c'est à dire que les sorties soient dans l'état $Q, \bar{Q} = (1, 0)$ lorsque la cellule reviendra dans le mode 'lecture'), il faut appliquer un 1 sur S (set), puis le remettre à 0. Quelqu'ait été son état antérieur, la cellule se retrouve alors dans l'état auto-stable $Q, \bar{Q} = (1, 0)$.

Ce circuit est bien séquentiel : ses sorties ne dépendent pas uniquement de ses entrées. La notion d'état interne est ici clairement présente sous la forme d'un bit mémorisé.

Graphe d'états

Les modes de fonctionnement décrits précédemment peuvent être résumés dans le graphe de laFigure III-35.

Un tel graphe est appelé ***graphe d'état*** du circuit ; il synthétise complètement son fonctionnement. Les deux nœuds correspondent aux deux états stables du circuit ; les arcs montrent les passages du circuit d'un état vers un autre lors d'un changement des valeurs des entrées. La valeur de la sortie Q est associée à chacun des deux états : 0 pour l'état a et 1 pour l'état b. On ne représente que les états stables du circuit, et non les configurations intermédiaires fugitives entre deux états.

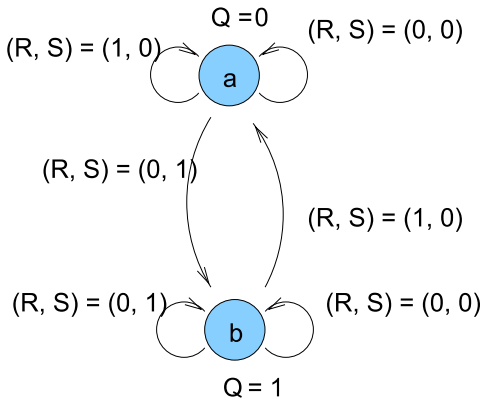


Figure III-35. Graphe d'états d'un latch RS. Les nœuds correspondent aux états stables du circuit, et les arcs représentent les transitions entre les états.

IV.3. Fronts et niveaux ; signaux d'horloge

Le niveau d'un signal, c'est sa valeur 0 ou 1. On parle de front lors d'un changement de niveau : front montant lorsque le signal passe de 0 à 1, front descendant lorsqu'il passe de 1 à 0. En pratique, ces transitions ne sont pas tout à fait instantanées, mais leur durée est très inférieure aux temps de propagation des portes (Figure III-36).

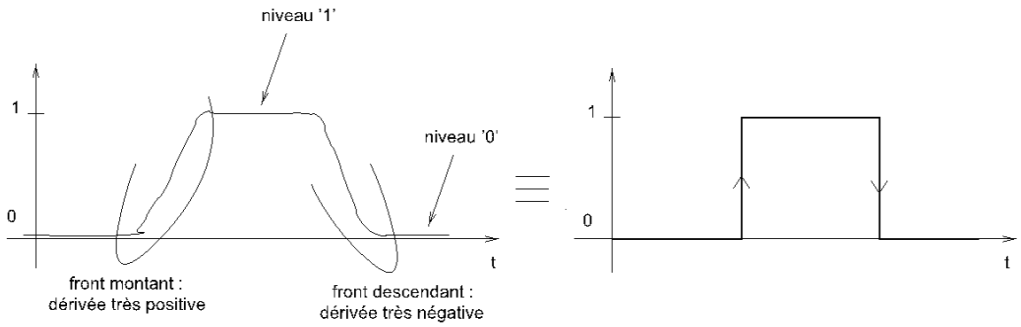


Figure III-36. Fronts et niveaux d'un signal réel, et leur équivalent logique simplifié.

Une horloge est un signal périodique, produit à partir d'un quartz ou d'un réseau RC. Il produit donc une succession périodique de fronts montants et/ou descendants.
Comme en physique ou en mathématiques, on emploie les termes de fréquence et de période pour qualifier la chronologie d'un signal d'horloge ; elle a généralement une forme carrée, mais ce n'est pas obligatoire.

IV.4. Circuits séquentiels synchrones et asynchrones : définitions

Circuits asynchrones purs

On appelle circuits séquentiels asynchrones purs des circuits séquentiels sans signaux d'horloge, et dans lesquels ce sont les changements de valeur des entrées qui sont à la source des changements d'états. Le latch RS est un exemple de circuit asynchrone pur : il n'est pas gouverné par une horloge, et ce sont les changements des entrées qui provoquent les changements d'état.

Circuits synchrones purs

On appelle circuit séquentiels synchrones purs des circuits séquentiels qui possèdent une horloge unique, et dont l'état interne se modifie précisément après chaque front (montant ou descendant) de l'horloge.
Les circuits séquentiels synchrones purs sont plus simples à étudier et à concevoir que les circuits séquentiels asynchrones purs, car le moment de la transition est défini de l'extérieur et sur un seul signal. Ils peuvent par contre être moins rapides qu'un équivalent asynchrone, et consommer plus d'énergie. En effet, on se rappelle (section 3) que les circuits CMOS consomment essentiellement du courant lors des changements d'états.
Un circuit séquentiel synchrone consommera ainsi du courant à chaque front d'horloge, contrairement à un circuit asynchrone pur. Cet effet est toutefois à relativiser avec certains circuits CMOS dont on a beaucoup abaissé la tension d'alimentation, et pour lesquels le courant de fuite (consommé au repos) est proche du courant de commutation (consommé lors des changements).

Autres circuits séquentiels

Les circuits séquentiels qui ne sont, ni synchrones purs, ni asynchrones purs, n'ont pas de dénomination particulière. Ils sont souvent l'union de plusieurs sous-ensembles qui sont chacun gouvernés de façon synchrone par une horloge, mais l'interface entre ces sous-domaines est souvent la cause de problèmes de fiabilité. On n'étudiera pas de tels circuits dans cet ouvrage.

IV.5. Graphes d'états

Considérons le fonctionnement du circuit séquentiel synchrone de laFigure III-37, détecteur de la séquence '1,0,1' :

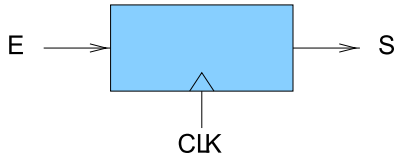


Figure III-37. Détecteur de séquence 1,0,1.

Une suite de chiffres binaires arrive sur E, et les instants d'échantillonnage (moments où la valeur de E est prise en compte) sont les fronts montants de CLK. On souhaite que la sortie S vaille 1 si et seulement si les deux dernières valeurs de E sont : 1,0,1.
Cette spécification est en fait imprécise, car elle ne dit pas exactement à quel moment par rapport à l'horloge CLK la sortie doit prendre sa valeur. Il y a deux possibilités :

1. la sortie ne dépend que de l'état interne du circuit, et ne peut changer que juste après chaque front d'horloge. Elle ne pourra changer ensuite qu'au prochain front d'horloge, même si les entrées changent entre temps. On appelle ***schéma de MOORE*** une telle conception, et elle conduit au ***graphe de MOORE*** de la Figure III-38.

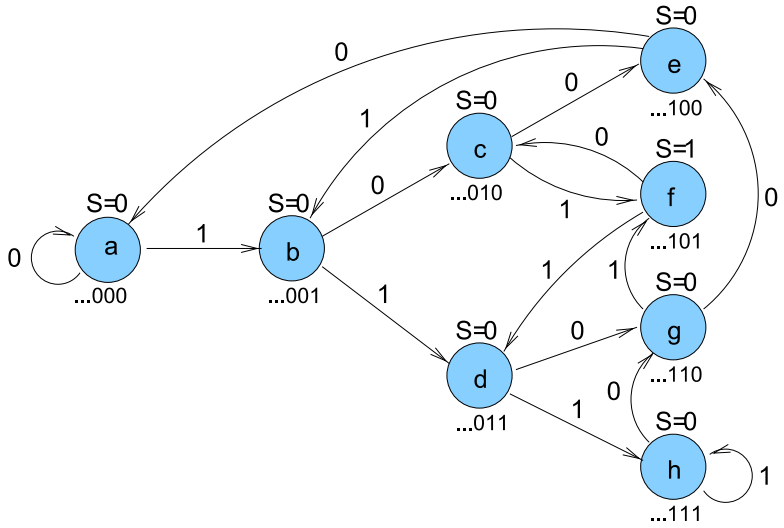


Figure III-38. Détecteur de séquence 1,0,1 de type Moore, non simplifié. Chaque état est associé à la réception d'une certaine séquence des 3 derniers bits. Seul l'état g provoque la sortie S =1.

La valeur de la sortie S est clairement associée aux états, et ne peut donc changer qu'avec eux. Les changements d'état se produisent à chaque front de l'horloge CLK, qui n'est pas représentée sur le graphe, mais dont la présence est implicite. Un arc libellé '0' par exemple indique un changement d'état lorsque E = 0.

2. la sortie dépend de l'état interne du circuit et de ses entrées, et on dit alors qu'il s'agit d'un ***schéma de MEALY***. Pour le détecteur de séquence par exemple, la sortie S pourrait valoir 1 dès que E passe à 1 pour la deuxième fois, avant même que sa valeur ne soit 'officiellement' échantillonnée au prochain front d'horloge. Cela conduit au ***graphe de MEALY*** de la Figure III-39.

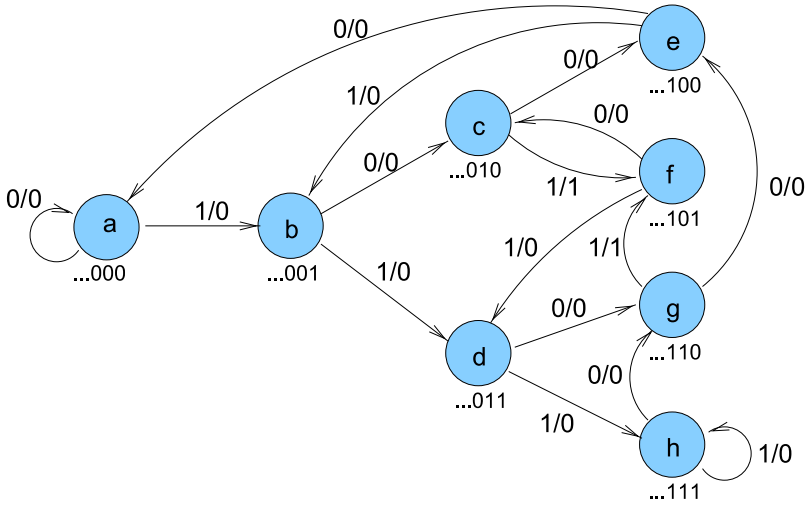


Figure III-39. Détecteur de séquence 1,0,1 de type Mealy,non simplifié.

L'arc libellé '1/0'de a vers b par exemple indique un changement d'état de a vers b lorsque E = 1, avec une valeur de la sortie S=0. Mais attention : le changement d'état ne se produira qu'au prochain front d'horloge, alors que le changement de sortie se produit immédiatement après le changement des entrées.

Différences entre les circuits de MOORE et de MEALY

La principale différence tient dans le fait que dans un circuit de MEALY, on obtient les sorties un emps d'horloge plus tôt. Par exemple dans la transition de c vers f pour E = 1, la sortie S = 1 sera échantillonnable (= mémorisée et utilisable) au front d'horloge suivant, en même temps que l'état courant passera à f. Dans le circuit de MOORE, la sortie passera à 1 ***après*** le front d'horloge suivant, et ne sera stable et échantillonnable qu'au front d'horloge encore ultérieur, c'est à dire un temps plus tard que dans le circuit de MEALY.
Pour certains types de circuits, il est plus facile de dessiner un graphe de MOORE, qui correspond à une vision plus 'statique'du problème. La bonne nouvelle est qu'il existe une méthode automatique de transformation d'un graphe de MOORE en graphe de MEALY, lorsque l'on veut profiter des avantages de ce dernier.

Transformation d'un graphe de Moore en graphe de Mealy

On voit sur la Figure III-40 comment un arc dans un graphe de MOORE se transforme en un arc dans un graphe de MEALY équivalent.

Figure III-40. Un arc d'un graphe de MOORE transformé en un arc équivalent d'un graphe de MEALY. Si avec l'entrée e on va en a associé à une sortie S, alors cette sortie peut être directement attachée à l'arc : e/S.

On a donc une méthode automatique pour transformer les graphes de MOORE en graphes de MEALY, très utile car les graphes de MOORE sont souvent plus faciles à concevoir. C'est elle qu'on a utilisée par exemple pour transformer le graphe de MOORE de la Figure III-38 en graphe de MEALY (Figure III-39).

Transformation d'un graphe de Mealy en graphe de Moore

Ce n'est pas toujours possible, et donc aucune méthode générale n'existe. Les circuits de MEALY sont donc intrinsèquement plus riches que ceux de MOORE.

IV.6. Tables de transitions

Les tables de transitions, encore appelées tables de Huffman ne sont rien d'autre qu'une version tabulaire des graphes de transitions. Par exemple, les tables associées aux graphes de MOORE et de MEALY du détecteur de séquence sont représentées Figure III-41 et Figure III-42.

avant		*après*
état	E	état
a	0	a
a	1	b
b	0	c
b	1	d
c	0	e
c	1	f
d	0	g
d	1	h
e	0	a
e	1	b
f	0	c
f	1	d
g	0	e
g	1	f
h	0	g
h	1	h

état	S
a	0
b	0
c	0
d	0
e	0
f	1
g	0
h	0

Figure III-41. Table de transitions du graphe de MOORE pour le détecteur de séquence 1, 0, 1. À chaque arc du graphe correspond une ligne dans la table. On notera la table complémentaire, qui donne les sorties associées à chaque état.

avant		*après*	
état	E	état	S
a	0	a	0
a	1	b	0
b	0	c	0
b	1	d	0
c	0	e	0
c	1	f	1
d	0	g	0
d	1	h	0
e	0	a	0
e	1	b	0
f	0	c	0
f	1	d	0
g	0	e	0
g	1	f	1
h	0	g	0
h	1	h	0

Figure III-42. Table de transitions du graphe de MEALY pour le détecteur de séquence 1, 0, 1. À chaque transition du graphe correspond une ligne de la table. La valeur de la sortie S est associée à chaque transition.

Simplification des tables de transition

Une table de transitions permet notamment de repérer facilement des états ***équivalents***. Un groupe G d'états sont dits équivalents si, pour chaque combinaison possible des entrées, ils conduisent à des états du groupe G avec les mêmes sorties. On peut alors diminuer le nombre d'états en remplaçant tous les états du groupe G par un seul, puis tenter de rappliquer cette procédure de réduction sur l'ensemble d'états réduit.
Par exemple, sur la table de transitions précédente associée au diagramme de MOORE, on constate que a et e sont équivalents, que c et g sont équivalents, que d et h sont équivalents. Attention : b et f ne sont pas équivalents car, bien qu'ils aient mêmes états suivants, ils sont associés à des sorties différentes. On peut réécrire la table de transitions en supprimant les lignes associées à e, g et h, et en remplaçant partout e par a, g par c, h par d (Figure III-43).

avant		*après*
état	E	état
a	0	a
a	1	b
b	0	c
b	1	d
c	0	a
c	1	f
d	0	c
d	1	d
f	0	c
f	1	d

état	S
a	0
b	0
c	0
d	0
f	1

Figure III-43. *Table de transitions du détecteur de séquence 1, 0, 1 après une première phase de simplification.*

Sur cette table, on constate que b et d sont équivalents, ce qui ne pouvait pas être déterminé à l'étape précédente. Le processus de simplification s'arrête ici. Finalement, 4 états suffisent pour ce détecteur de séquence. Ceux-ci ont perdu la signification qu'ils avaient initialement dans le graphe à 8 états. Le graphe simplifié est représenté Figure III-44.

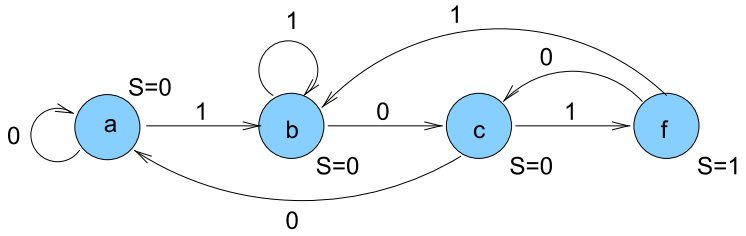


Figure III-44. Graphe de MOORE simplifié du détecteur de séquence 1,0,1

avant		*après*
état	E	état
a	0	a
a	1	b
b	0	c
b	1	b
c	0	a
c	1	f
f	0	c
f	1	b

état	S
a	0
b	0
c	0
f	1

Figure III-45. *Table de transitions simplifiée du détecteur de séquence 1, 0, 1.*

Cette procédure de simplification s'applique également aux graphes de MEALY, en cherchant des identités dans les couples état/sortie. Sur l'exemple précédent, on constate que a et e sont équivalents, que b et f sont équivalents, que c et g sont équivalents, et que d et h sont équivalents. En procédant aux suppressions et substitutions associées, on trouve la nouvelle table :

avant		*après*	
état	E	état	S
a	0	a	0
a	1	b	0
b	0	c	0
b	1	d	0
c	0	a	0
c	1	b	1
d	0	c	0
d	1	d	0

Cette fois, on voit que b et d sont équivalents :

avant		*après*	
état	E	état	S
a	0	a	0
a	1	b	0
b	0	c	0
b	1	b	0
c	0	a	0
c	1	b	1

Figure III-46. *Table de transitions de MOORE pour le détecteur de séquence 1,0,1, après une deuxième phase de simplification.*

Le processus s'arrête ici. Comme c'est souvent le cas, le graphe de MEALY est plus simple que le graphe de MOORE associé au même problème, ne comportant que 3 états (Figure III-47).

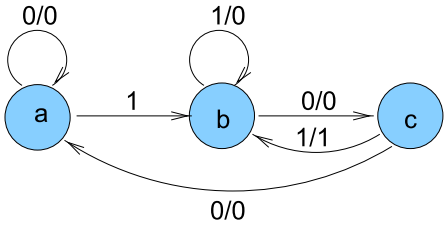


Figure III-47. Graphe de MEALY simplifié du détecteur de séquence 1,0,1

IV.7. Bascules synchrones

Notre but est maintenant de réaliser à l'aide de circuits logiques les machines séquentielles telles qu'elles peuvent être décrites par les graphes d'états. Pour faciliter cette synthèse, il a été imaginé de créer des composants appelés bascules, qui formeraient les briques de base à utiliser. Une bascule mémorise un seul bit et permet de représenter seulement deux états distincts, mais l'utilisation d'un vecteur de n bascules permet de coder jusqu'à 2^n états distincts. Par exemple, pour le détecteur de séquence analysé à la section précédente, le graphe de Moore simplifié comportait 4 états, et un vecteur de deux bascules va permettre de les encoder.
Par ailleurs, pour que tout un vecteur de bascules évolue de façon fiable d'état en état, il est souhaitable que chacune d'elles ait une entrée d'horloge reliée à une horloge commune (circuit synchrone), et ne puisse changer d'état *qu'au moment précis d'un front de cette horloge* (montant ou descendant), et jamais entre deux fronts, même si les autres entrées du circuit sont modifiées plusieurs fois entre temps. De tels circuits sont de conception difficile ; on les présente dans cette section.

IV.7.1. Latch RS actif sur un niveau d'horloge

Dans le but de rendre synchrone le latch RS vu précédemment, on peut essayer limiter la période d'écriture (Figure III-48).

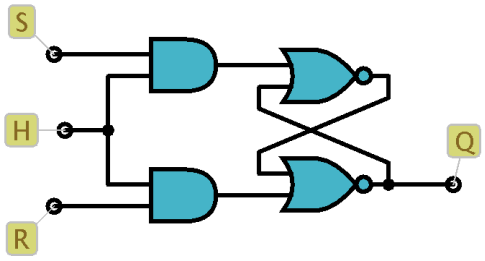
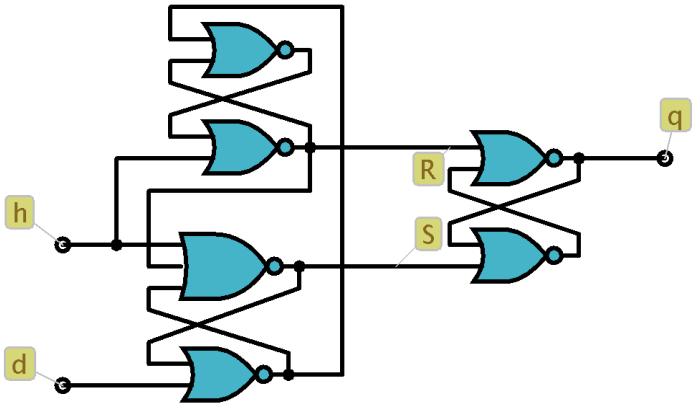


Figure III-48. Latch RS qui mémorise sur un niveau de H

L'état de ce circuit est modifiable lorsque H = 1, c'est à dire sur un niveau du signal H. Tant que H est au niveau 1, les modifications de S et de R vont avoir un effet sur l'état du bistable. Un tel latch peut avoir certaines applications, mais il n'est pas adapté à la réalisation de circuits séquentiels synchrones, où les changements d'états de leurs différentes parties doivent se produire de façon parfaitement synchronisée, au moment précis défini par un front d'une horloge commune. La section suivante va présenter un tel circuit.

IV.7.2. Bascule active sur front d'horloge

Nous avons donc besoin de composants séquentiels qui changent d'état sur un front d'horloge, et seulement à ce moment précis. Le latch RS vu à la section précédente ne remplissait pas cette condition, puisque le bistable pouvait changer de valeur tant que H était au niveau 1. Le circuit de laFigure III-49, qui réalise ce qu'on va appeler une bascule D (D pour 'delay'), ne change d'état quant à lui que sur le front descendant de l'horloge H.



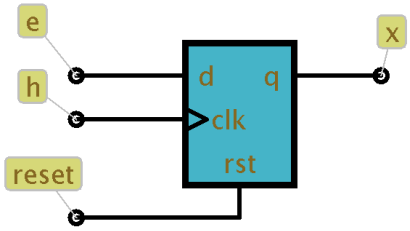

```
module basculeD(h,d: q)
  s=/h*/r*/ns
  ns=/s*/d
  r=/nr*/h
  nr=/ns*/r
  q=/r*/nq
  nq=/s*/q
end module
```

Figure III-49. Bascule D synchrone : l'état du bistable q ne change qu'au moment où h passe de 1 à 0.

La compréhension détaillée du fonctionnement de ce circuit est difficile, et l'utilisation d'un simulateur tel que celui de SHDL peut y aider. On peut déjà remarquer que lorsque h vaut 1, on a nécessairement R=0 et S=0, et donc le bistable final qui produit q est dans l'état de repos : tant que h vaut 1, l'état interne q ne peut pas changer, même si d change de valeur un nombre quelconque de fois. Lorsque h passe de 1 à 0, on peut montrer que le bistable du haut va stocker la valeur de $\overline{D}$ et la placer sur R, et que celui du bas va stocker la valeur de D et la placer sur S. On a donc au moment du front descendant un 'set' ou un 'reset' du bistable final, dans le sens de la mémorisation de la donnée d en entrée. Enfin, et c'est le point important, on peut montrer également que lorsque h est maintenu à 0, les valeurs de R et de S ne peuvent pas changer même si d change, et donc les changements de d n'affectent pas le bistable final. On a donc bien démontré la propriété essentielle de ce circuit, qui est que son état interne q est affecté durant la période très courte où l'horloge h passe de 1 à 0, c'est à dire durant un front descendant de celle ci, et que tous les changements sur d en dehors de cette période n'ont pas d'effet sur q.

IV.7.3. Bascule D

Une telle bascule D est représentée couramment par le schéma de la Figure III-50.




```
module test(e, h, reset: x)
  x = e on h, reset when reset
end module
```

Figure III-50. Représentation habituelle d'une bascule D synchrone.

On a indiqué également l'écriture SHDL associée à cette bascule. Le signal de remise à zéro RST n'était pas présent sur la Figure III-49. Il s'agit d'une remise à zéro asynchrone, qui provoque le forçage à 0 du contenu de la bascule, sans qu'il y ait besoin d'un front d'horloge. Tant que le signal RST est actif (sur niveau 1 sur la figure), la bascule est forcée à 0 et son contenu ne peut pas évoluer. Il s'agit du même signal reset que celui qui est présent sur votre micro-ordinateur, votre téléphone portable, etc. : en l'appliquant, vous remettez dans l'état 0 tous les composants séquentiels de votre machine.
Du point de vue graphique, le triangle devant l'entrée CLK indique un signal d'horloge. S'il était précédé d'un rond, cela indiquerait une horloge active sur front descendant ; en son

absence elle est active sur front montant. De la même façon, un rond devant l'entrée RST indique une activité sur niveau bas. La Figure III-51 illustre ces différentes situations.

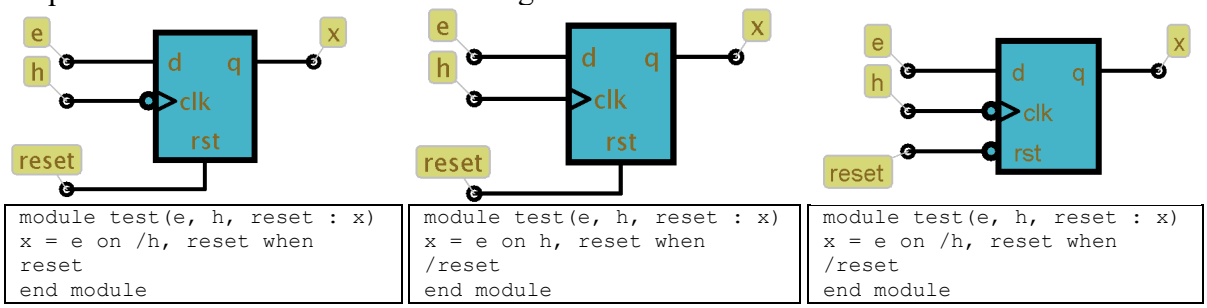


Figure III-51. Différentes bascules D. (a) horloge sur front descendant. (b) reset actif sur niveau bas. (c) horloge front descendant et reset actif sur niveau bas.

Bascules et langage SHDL

- `set when ..` permet un forçage à 1 asynchrone, c'est-à-dire indépendamment du front d'horloge. On s'en sert généralement pour initialiser le circuit.
- `.enabled when ..` fait que la bascule n'évolue pas tant que cette entrée n'est pas active. Ce modifieur est facultatif.

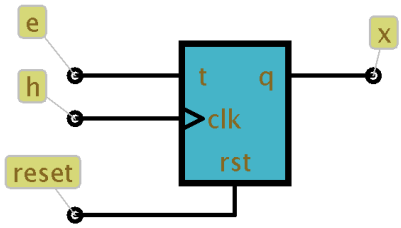
Équation d'évolution

L'équation d'évolution d'une bascule, c'est l'équation de la valeur que prendra l'état après le front d'horloge. Dans le cas de la bascule D, c'est tout simplement la valeur présente à son entrée :

$$Q := D$$

IV.7.4. Bascule T

Tout comme la bascule D, la bascule T (trigger) mémorise également un bit de façon synchrone, mais elle a des modalités différentes d'utilisation. Son schéma, une table de transitions synthétique et l'écriture SHDL associée sont donnés Figure III-52.



avant		*après*
T	Q	Q
0	x	x
1	x	$\bar{x}$

```
module test(e, h, reset : x)
  x = /e*x + e*/x on h,
reset when reset
end module
```

Figure III-52. Bascule T synchrone. Lorsque l'entrée T est à 0 elle ne change pas d'état ; lorsque l'entrée T vaut 1 son état s'inverse.

L'état mémorisé de la bascule T s'inverse (après le front d'horloge) si et seulement si son entrée T vaut 1. Lorsque son entrée T vaut 0, cet état reste inchangé. Elle est donc adaptée aux problématiques de changements d'une valeur et non à un simple stockage comme la bascule D. En dehors des variations sur les fronts d'horloge et la ligne de reset asynchrone, les deux seules versions possibles de la bascule T sont données **Erreur ! Source du renvoi introuvable.**.

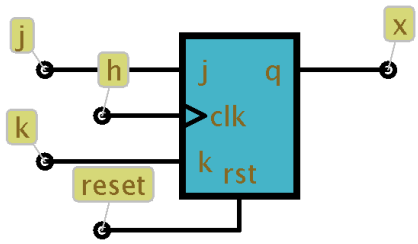
Equation d'évolution

$$Q := \bar{T}\,Q \;+\; T\bar{Q}$$

Le premier terme exprime bien que, si T est à 0, Q ne change pas, et que si T est à 1, Q s'inverse. On retrouve l'écriture SHDL de l'affectation séquentielle.

IV.7.5. Bascule JK

Comme les bascules D et T, la bascule JK (Jack-Kilby) mémorise un bit de façon synchrone. Son schéma, sa table de transitions simplifiée et l'écriture SHDL associée sont donnés Figure III-53.



avant			*après*
J	K	Q	Q
0	0	x	x
0	1	-	0
1	0	-	1
1	1	x	$\bar{x}$

```
module test(reset, h, j, k :
x)
  x = /k*x + j*/x on h, reset
when reset
end module
```

Figure III-53. Bascule JK synchrone, table de transitions simplifiée et écriture SHDL. Selon les valeurs de ses entrées J et K, elle permet le forçage de son état à 0 ou à 1, mais aussi la conservation ou l'inversion de son état.

Elle fonctionne de façon analogue au latch RS, J jouant le rôle de S et K de R. Lorsque J et K sont à 0, l'état de la bascule ne change pas au front d'horloge. Si on veut faire une mise à 1ou une mise à 0, on applique 1sur J (respectivement sur K) puis on applique un front d'horloge. De plus, mettre à la fois J et K à 1 est autorisé, et l'état de la bascule s'inverse au front d'horloge.

Équation d'évolution

$$Q := \bar{K}\,Q + J\bar{Q}$$

Elle est moins immédiatement évidente que celle des bascules D et T. On vérifie que dans tous les cas, on a le résultat attendu :

- si $J = 0$ et $K = 0$, $\bar{K}\,Q + J\bar{Q} = 0 + Q = Q$
- si $J = 0$ et $K = 1$, $\bar{K}\,Q + J\bar{Q} = 0 + 0 = 0$
- si $J = 1$ et $K = 0$, $\bar{K}\,Q + J\bar{Q} = Q + \bar{Q} = 1$
- si $J = 1$ et $K = 1$, $\bar{K}\,Q + J\bar{Q} = 0 + \bar{Q} = \bar{Q}$

La bascule JK existe également avec des entrées J et K inversées. La **Erreur ! Source du renvoi introuvable.** montre ces différentes formes, et l'écriture SHDL associée.

IV.7.6. Choix des bascules

La bascule D est clairement adaptée aux situations où on doit stocker un bit présent. La bascule T est adaptée aux situations qui se posent en termes de changement ou d'inversion. La bascule JK semble plus versatile, et elle est d'ailleurs souvent appelée bascule universelle. Par analogie avec les portes combinatoires, on pourrait se demander si cette bascule JK est 'plus puissante' que les bascules D ou T, c'est à dire s'il est possible de fabriquer avec elle des circuits que les autres ne pourraient pas faire. La réponse est non. Ces trois types de bascules ont une puissance d'expression équivalente : elles mémorisent un bit, et elles ne diffèrent que par les modalités de stockage de ce bit. Par exemple, on vérifie facilement qu'on peut fabriquer une bascule JK avec une bascule T et réciproquement (Figure III-54).

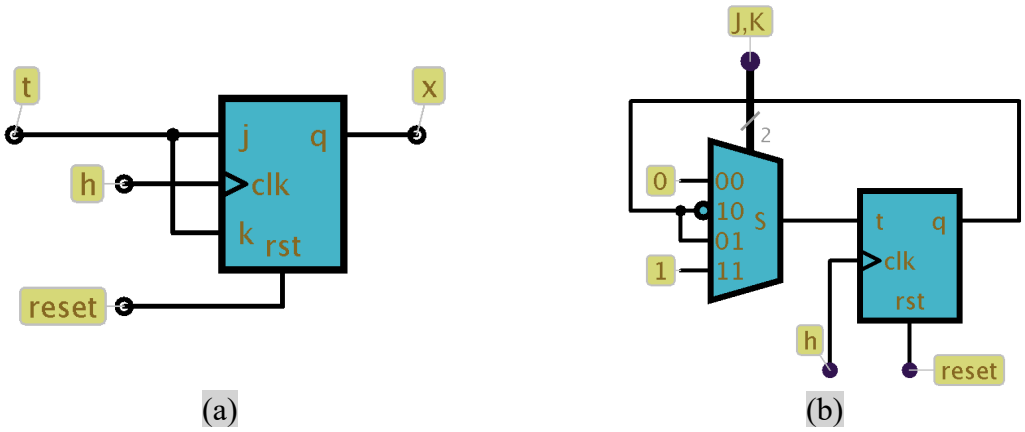


Figure III-54. (a) construction d'une bascule T avec une bascule JK,(b) construction d'une bascule JK avec une bascule T.

Dans le cas (a), il est clair qu'en reliant les deux entrées J et K, la bascule reste dans le même état si t vaut 0, et elle s'inverse si t vaut 1, ce qui correspond bien au fonctionnement d'une bascule T.
Dans le cas (b), on a figuré la solution avec un multiplexeur pour plus de clarté. La valeur mise en entrée de la bascule T dépend de la valeur du couple j, k, et on vérifie qu'on a le résultat souhaité dans tous les cas :

1. si jk = 00, on place 0 sur l'entrée de la bascule T et elle ne changera pas de valeur au front d'horloge.
2. Si jk = 10, on place sur l'entrée de la bascule T l'inverse de son état interne, et il est facile de voir que cela conduit à la mise à 1 de la bascule au prochain front d'horloge.
3. Si jk = 01, on place l'état courant de la bascule T sur son entrée, ce qui conduit à la forcer à 0 au prochain front.
4. Enfin si jk = 11, on place 1 en entrée de la bascule T ce qui va déclencher son inversion.

IV.7.7. Schéma général d'un circuit séquentiel synchrone de type MOORE

La Figure III-55 montre le schéma général d'un circuit séquentiel synchrone de type ***MOORE***. L'état interne est mémorisé dans un vecteur de n bits (2n état au plus) constitué de bascules de type quelconque. Pour que ce circuit soit bien de type synchrone pur, toutes les horloges des bascules ont bien été reliées ensemble directement à l'horloge générale, de façon à garantir un changement d'état de toutes les bascules exactement au même moment.

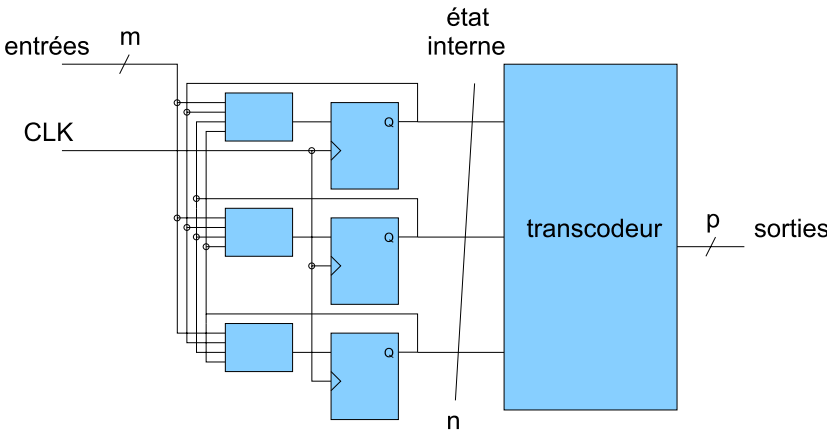


Figure III-55. Schéma général d'un circuit séquentiel synchrone de type MOORE

Les propriétés des bascules garantissent que cet état ne peut pas changer entre deux fronts d'horloge, même si les entrées du circuit changent plusieurs fois entre temps. Par ailleurs on voit que les sorties ne dépendent que de l'état interne, étant un simple transcodage combinatoire de celui-ci.

IV.7.8. Schéma général d'un circuit séquentiel synchrone de type MEALY

La Figure III-56 montre le schéma général d'un circuit séquentiel synchrone de type MEALY. Il s'agit également d'un circuit synchrone pur, dans lequel les horloges de toutes les bascules sont reliées entre elles à l'horloge générale. Avec un vecteur d'état composé de n bascules, on peut également coder au plus 2n états différents, et la mécanique de changement d'états est analogue à celle d'un circuit de type MOORE. La seule différence est dans le calcul des sorties : on voit qu'elles ne dépendent pas que de l'état interne, mais qu'elles dépendent aussi des entrées. C'est ce qui va leur permettre d'obtenir les mêmes sorties qu'un circuit de MOORE, un temps plus tôt.

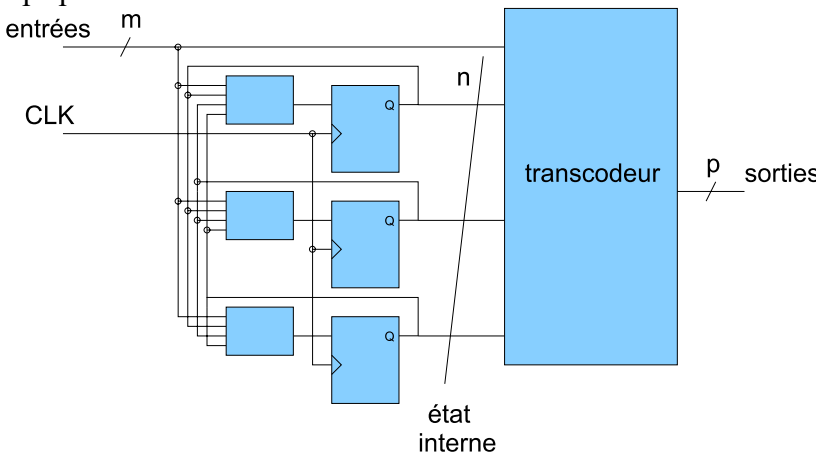


Figure III-56. Schéma général d'un circuit séquentiel synchrone de type MEALY.

IV.8. Synthèse d'un circuit séquentiel synchrone

IV.8.1. Étapes de la synthèse d'un circuit séquentiel synchrone

Lors de la synthèse d'un circuit séquentiel synchrone à l'aide de bascules, qu'il soit de type MOORE ou de type MEALY, on obtiendra le circuit le plus simple en suivant les étapes suivantes :

1. Dessin du graphe d'état : il permet une spécification claire du circuit

2. table de transitions : forme plus lisible du graphe, qui permet de vérifier qu'aucune transition n'est oubliée, et prépare la phase de simplification.
3. simplification de la table de transitions : on applique la méthode vue précédemment, parfois en plusieurs étapes.
4. détermination du nombre de bascules : le nombre d'états du graphe simplifié permet d'établir un nombre minimum *n* de bascules à employer. On ne choisit pas encore le type des bascules à employer, car la phase d'assignation fournira de nouvelles informations.
5. assignation des états : pour chaque état, un vecteur unique de *n* bits est assigné, appelé vecteur d'état. Des règles heuristiques d'assignation permettent de trouver celle qui conduira à des calculs simples.
6. table de transitions instanciée : on réécrit la table de transitions, en remplaçant chaque symbole d'état par le vecteur associé.
7. choix des bascules : en observant la table de transitions instanciée, certains types de bascules peuvent être préférés. On emploiera une bascule D lorsqu'un état suivant est la recopie d'une valeur de l'état précédent, une bascule T lorsque l'état suivant est l'inverse d'un état précédent ; dans les autres cas, la bascule JK peut être employée.
8. calcul des entrées des bascules et calcul des sorties du circuit.

IV.8.2. Synthèse du détecteur de séquence, version MOORE

A titre d'exemple, on va détailler la synthèse du circuit détecteur de la séquence 1,0,1 décrit précédemment, en version MOORE. On reprend donc toutes les étapes énumérées en introduction.

Graphe d'état, table de transitions, simplification

Ces trois étapes ont été réalisées en section 6, et ont abouties au graphe simplifié et à la table de transitions de la Figure III-57 et de la Figure III-58.

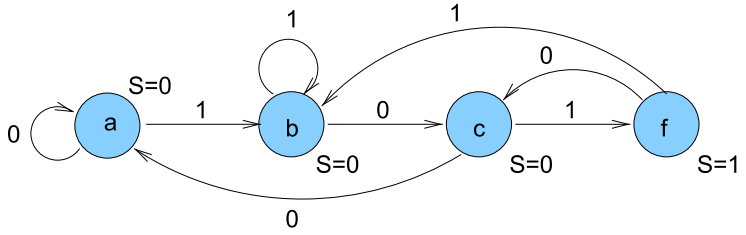


Figure III-57. Graphe de MOORE simplifié du détecteur de séquence 1,0,1.

avant		*après*
état	E	état
a	0	a
a	1	b
b	0	c
b	1	b
c	0	a
c	1	f
f	0	c
f	1	b

état	S
a	0
b	0
c	0
f	1

Figure III-58. *Table de transitions simplifiée du détecteur de séquence 1, 0, 1.*

Détermination du nombre de bascules

Il y a quatre états à coder, donc 2 bascules seront employées, dont on appellera les sorties X et Y.

Assignation des états

La phase d'assignation consiste donc à affecter un vecteur d'état à chacun des états du circuit ; ici un couple (X,Y) pour chacun des 4 états. N'importe quelle assignation peut être employée. Néanmoins, on peut toujours affecter à l'état initial le vecteur d'état (0,0,…), qui sera forcé lors d'un RESET asynchrone du circuit. Par ailleurs, certaines assignations donnent lieu à des calculs plus simples que d'autres. On les trouve souvent en appliquant les règles heuristiques suivantes, par ordre de priorité décroissante :

1. rendre adjacents les états de départ qui ont même état d'arrivée dans le graphe.
2. rendre adjacents les états d'arrivée qui ont même état de départ dans le graphe.
3. rendre adjacents les états qui ont mêmes sorties.

L'idée est de minimiser le nombre de bits qui changent (état ou sorties) lors des changements d'états.

Appliqué à notre circuit, la première règle préconise d'associer a et c, a et f, b et f ; la deuxième règle préconise d'associer a et b, b et c, a et f ; la troisième règle préconise d'associer a, b et c. La Figure III-59 propose une assignation qui respecte au mieux ces règles.

Y \ X	0	1
0	a	f
1	c	b

Figure III-59. Assignation des états : chaque état est associé à une configuration binaire des bascules

Table de transitions instanciée

On recopie la table de transitions précédente, en remplaçant chaque symbole d'état par son vecteur d'état (Figure III-60).

avant		*après*
XY	E	XY
00	0	00
00	1	11
11	0	01
11	1	11
01	0	00
01	1	10
10	0	01
10	1	11

XY	S
00	0
11	0
01	0
10	1

Figure III-60. *Table de transitions instanciée.*

Choix des bascules

En observant la table instanciée, on constate que X à l'état suivant reproduit l'entrée E : une bascule D s'impose pour X. Dans le cas de Y, il reproduit presque la valeur de X de l'état précédent, sauf pour la deuxième ligne. A titre d'exemple, on va choisir une bascule JK.

Calcul des entrées des bascules et de la sortie du circuit

Plusieurs méthodes sont possibles pour ces calculs. Celle qui sera employée ici est basée sur une réécriture de la table de transitions, à laquelle on rajoute les entrées des bascules, ici DX

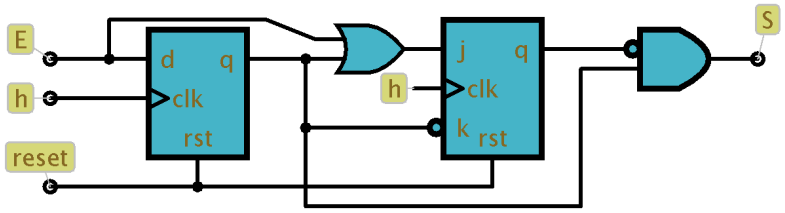
pour la bascule X et JY, KY pour la bascule Y (Figure III-61. *Table de transitions instanciée avec les entrées des bascules choisies.* .

avant					*après*
XY	E	DX	JY	KY	XY
00	0	0	0	*	00
00	1	1	1	*	11
11	0	0	*	0	01
11	1	1	*	0	11
01	0	0	*	1	00
01	1	1	*	1	10
10	0	0	1	*	01
10	1	1	1	*	11

XY	S
00	0
11	0
01	0
10	1

Figure III-61. *Table de transitions instanciée avec les entrées des bascules choisies.*

En fait on connaissait déjà le résultat pour X : $DX = E$, qui est la raison pour laquelle on avait choisie une bascule D. Pour Y, on trouve facilement que $JY = E + X$ et $= \bar{X}$. Par ailleurs, on a $S = X\bar{Y}$
La synthèse est terminée, et notre séquenceur se réduit au schéma de la Figure III-62. On notera l'écriture des affectations séquentielles de x et y : x := e est l'équation d'évolution de la bascule D, et y := x*y+jy*/y est une équation de la forme q := /k*q+j*/q, pour laquelle l'entrée K est inversée.




```
module seq101(rst,h,e:
s)
  x = e on h, reset
when reset
  y = x*y+jy*/y on h,
reset when reset
  jy = e+x
  s = x*/y
end module
```

Figure III-62. Schéma final synthétisé du détecteur de séquence 1,0,1 (type MOORE) et description SHDL associée

IV.8.3. Synthèse du détecteur de séquence, version MEALY

On effectue ici la synthèse du même détecteur de séquence 1,0,1, cette fois en version MEALY. Il permettra d'obtenir la sortie un front d'horloge avant la version MOORE, et on peut espérer un circuit encore plus simple puisque son graphe simplifié comporte un état de moins que celui de MOORE.

Graphe d'état, table de transitions, simplification

Ces trois étapes ont été réalisées en section 6, et ont abouties au graphe simplifié de la Figure III-63 et à la table de transitions de la

avant		*après*	
état	E	état	S
a	0	a	0
a	1	b	0
b	0	c	0
b	1	b	0
c	0	a	0
c	1	b	1

Figure III-64.

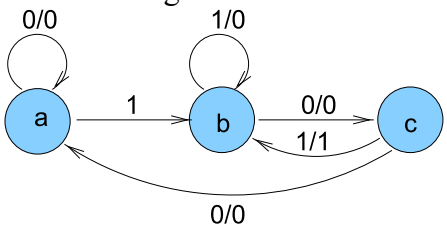


Figure III-63. Graphe de MEALY simplifié du détecteur de séquence 1,0,1

avant		*après*	
état	E	état	S
a	0	a	0
a	1	b	0
b	0	c	0
b	1	b	0
c	0	a	0
c	1	b	1

Figure III-64. *Table de transitions de MEALY simplifiée pour le détecteur de séquence 1,0,1.*

Assignation des états

La phase d'assignation consiste donc à affecter un vecteur d'état à chacun des états du circuit ; ici un couple (X,Y) pour chacun des 3 états. Afin d'obtenir les résultats les plus simples possibles, on essaie d'appliquer au mieux les heuristiques d'assignation décrites à la section précédente.

Appliquée à notre circuit, la première règle préconise d'associer a et c ; la deuxième règle préconise d'associer a et b. La Figure III-65 propose une assignation qui respecte au mieux ces règles.

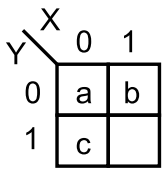


Figure III-65. Assignation des états : chaque état est associé à une configuration binaire des bascules

Table de transitions instanciée

On recopie la table de transitions précédente, en remplaçant chaque symbole d'état par son vecteur d'état (Figure III-66).

avant		*après*	
XY	E	XY	S
00	0	00	0
00	1	10	0
10	0	01	0
10	1	10	0
01	0	00	0
01	1	10	1

Figure III-66. Table de transitions instanciée.

Choix des bascules

En observant la table instanciée, on constate que X à l'état suivant reproduit l'entrée E : une bascule D s'impose pour X. Dans le cas de Y, il n'y a qu'une ligne où Y passe à 1 ; on peut choisir également une bascule D, dont on appellera l'entrée DY.

Calcul des entrées des bascules et de la sortie du circuit

Ici encore on va réécrire la table de transitions, à laquelle on va rajouter les entrées des bascules, ici DX pour la bascule X et DY pour la bascule Y (Figure III-67).

avant				*après*	
XY	E	DX	DY	XY	S
00	0	0	0	00	0
00	1	1	0	10	0
10	0	0	1	01	0
10	1	1	0	10	0
01	0	0	0	00	0
01	1	1	0	10	1

Figure III-67. Table de transitions instanciée avec les entrées des bascules choisies

En fait on connaissait déjà le résultat pour X : DX = E, qui est la raison pour laquelle on avait choisi une bascule D. Pour Y, on peut faire une table de Karnaugh qui va nous permettre d'exploiter les combinaisons non spécifiées dues au fait que la table d'assignation n'est pas complètement remplie (Figure III-68).

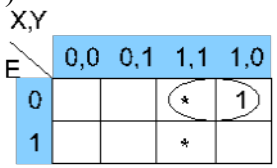
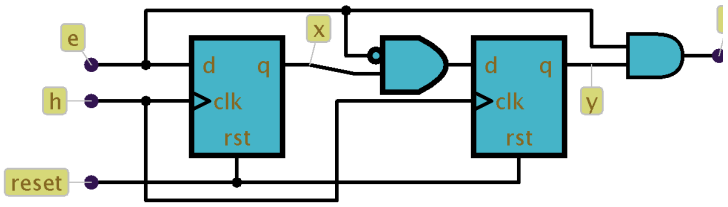


Figure III-68. Table de Karnaugh pour le calcul de DX. Le vecteur XY=11 est non spécifié

On trouve donc : $DX = \bar{E}\,X$. Par ailleurs, on a : $S = EY$.
La synthèse est terminée, et notre séquenceur se réduit au schéma de la Figure III-69. On voit que cette fois la sortie ne dépend plus seulement de l'état interne comme dans le circuit de MOORE ; elle dépend aussi de l'entrée E. Avec le simulateur SHDL, on pourra se convaincre que dans le circuit de MEALY on obtient bien la sortie S un front d'horloge avant le circuit de MOORE.




```
module seq101(rst, h, e : s)
  x = e on h, reset when
reset
  y = dy on h, reset when
reset
  dy = /e*x
  s = e*y
end module
```

Figure III-69. Schéma final synthétisé du détecteur de séquence 1,0,1 (type MEALY) et description SHDL associée. On voit que la sortie S dépend de l'état interne, mais aussi de l'entrée E

IV.8.4. Synthèse « one-hot encoding »

Une autre méthode de synthèse plus simple est fréquemment employée, le « one-hot encoding » : elle consiste à utiliser autant de bascules qu'il y a d'états, une seule étant à 1 à tout moment. Le '1' qui représente l'état courant circule alors de bascule en bascule, à la manière d'un jeton. À l'initialisation, la bascule représentant l'état initial doit être mise à 1 et les autre à 0.
Cette synthèse n'est pas optimale en nombre de bascules utilisées, mais elle permet une conception très lisible et maintenable. En utilisant des bascules D, l'équation de leurs entrées se dérivent directement des équations de transition.
Version MOORE :

```
module seq101HotMoore(rst, h, e : s)
   a := da on h, set when rst
   b := db on h, reset when rst
   c := dc on h, reset when rst
   f := df on h, reset when rst
   da = a*/e + c*/e
   db = a*e + b*e
   dc = b*/e + f*/e
   df = c*e
   s = f
end module
```

Version MEALY :

```
module seq101HotMealy(rst, h, e : s)
   a := da on h, set when rst
   b := db on h, reset when rst
   c := dc on h, reset when rst
   da = a*/e + c*/e
   db = a*e + b*e + c*e
   dc = b*/e
   s = c*e
end module
```

IV.8.5. Synthèse fonctionnelle

En raisonnant sur la fonction d'un circuit, on peut trouver les éléments nécessaires à sa réalisation, puis réfléchir à leur interconnexion.
Pour le détecteur de séquence '101' par exemple, on peut observer que la mémorisation des trois derniers bits reçus permettrait une solution simple.
Version MOORE :

```
module seq101Func(rst, h, e : s)
   a := e on h, reset when rst
   b := a on h, reset when rst
   c := b on h, reset when rst
   s = a*/b*c
end module
```

Version MEALY :

```
module seq101Func(rst, h, e : s)
   a := e on h, reset when rst
   b := a on h, reset when rst
   s = e*/a*b
end module
```

IV.9. Chronologie d'un circuit séquentiel synchrone

IV.9.1. Temps de setup, de hold et de propagation des bascules

Parmi les caractéristiques temporelles d'une bascule figurent les temps de *setup* et de *hold*. Le temps de setup est la durée ***avant le front d'horloge*** durant laquelle les entrées de la bascule doivent être stables pour que le basculement se passe correctement. Le temps de hold est la durée analogue ***après le front d'horloge***. L'union des deux définit une période autour du front d'horloge durant laquelle les entrées doivent être stables, faute de quoi le basculement ne se produirait pas comme prévu.
Le temps de propagation quant à lui est le temps après lequel on est garanti d'avoir la nouvelle valeur de la bascule, après la période d'instabilité de la transition.
Toutes ces durées sont illustrées sur la Figure III-70.

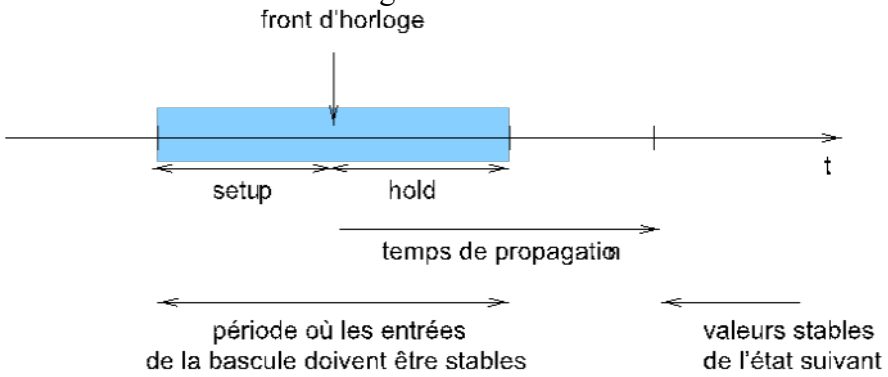


Figure III-70. Temps de setup, de hold et de propagation d'une bascule

IV.9.2. Chronologie détaillée d'une transition.

La lecture de cette section n'est pas nécessaire pour qui veut concevoir un circuit séquentiel. Il lui suffit d'admettre qu'après le front d'horloge, au moment où les bascules du circuit changent éventuellement d'état, on passe sans coup férir des valeurs de l'état précédent aux valeurs de l'état suivant.
On peut aussi chercher à comprendre en détail ce qui se passe lors d'une transition, et même craindre qu'elle ne se passe mal. Même si le front d'horloge est le même pour toutes les bascules, les valeurs des signaux de l'état précédent ne peuvent-elles pas interférer avec celles de l'état suivant ? C'est en effet possible dans certaines conditions. Considérons par exemple le schéma de la Figure III-71.
Si la bascule a a une commutation beaucoup plus rapide que la bascule b, la valeur de a va changer durant la période de hold de la bascule b, provoquant ainsi un fonctionnement incorrect.
Le plus souvent, c'est le temps de setup qui est violé, lorsqu'on augmente la vitesse d'horloge à un rythme tel qu'un front d'horloge arrive alors que les entrées des bascules, qui sont des fonctions combinatoires des entrées et de l'état courant, n'ont pas eu un temps suffisant pour se stabiliser.

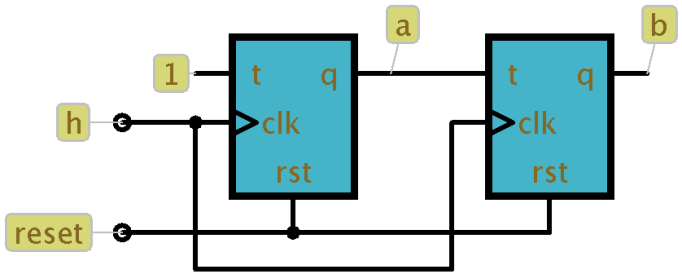


Figure III-71. Si la bascule a est trop rapide, le temps de hold de b sera violé.

En pratique, on emploie des bascules aux caractéristiques temporelles analogues.

IV.9.3. Bascules maître-esclave

Une façon de résoudre le problème de timing précédent consiste à séparer clairement le moment de capture (d'échantillonnage) des valeurs de l'état précédent du moment où on libère les valeurs pour l'état suivant, plutôt que de compter sur des délais de propagation trop serrés.
On utilise pour cela des bascules spéciales appelées bascules maître-esclave, câblées selon le modèle de la Figure III-72

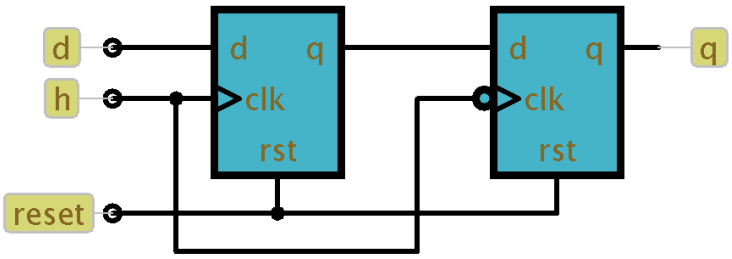


Figure III-72. Bascule maître-esclave.On capture les valeurs de l'état précédent au front montant de l'horloge, et on libère les valeurs pour l'état suivant au front descendant.

Sur le front montant de l'horloge h, les valeurs issues de l'état précédent sont prises en compte, sans être libérées encore vers les sorties, et donc sans risque de rétroaction néfaste. Plus tard, sur le front descendant de h, les nouvelles valeurs des bascules associées à l'état suivant sont libérées.
Les bascules maître-esclave peuvent être employées à peu près partout où sont employées des bascules ordinaires, mais il faut cette fois bien contrôler les deux fronts de l'horloge, et non un seul.

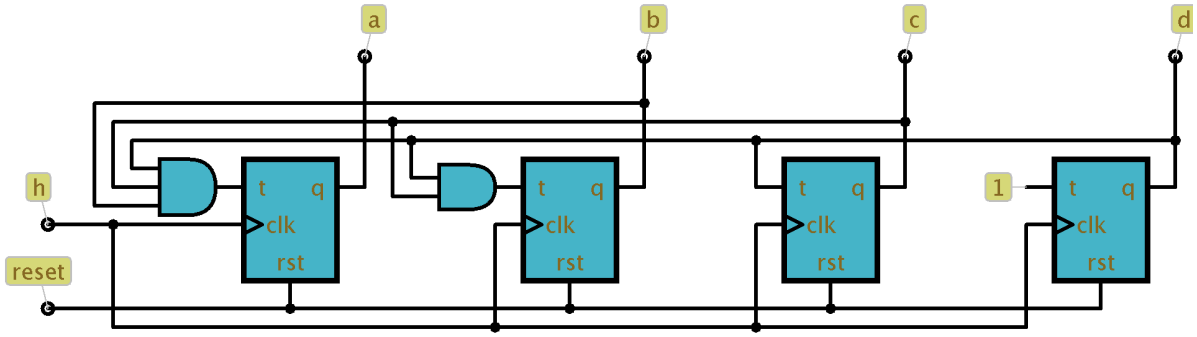
IV.10. Compteurs binaires

Compteur de base

Un compteur binaire sur n bits incrémente à chaque front d'horloge une valeur de n bits. On pourrait d'abord penser à créer un registre de n bits avec des bascules D, puis à relier leurs entrées à un circuit d'incrémentation, mais il y a une solution plus directe et efficace. On se rappelle en effet (section 4) l'algorithme de comptage : un bit est recopié avec inversion si tous les bits à sa droite valent 1, et le bit de poids faible est toujours inversé.
Cette description en termes d'inversion conduit à une conception à base de bascules T, dans laquelle l'entrée T d'une bascule est reliée au ET de tous les bits qui sont à sa droite. La

```
module cpt4(rst,h: a,b,c,d)
  d = /d on h,reset when rst
  c = /d*c + d*/c on h, reset when rst
  b = /tb*b + tb*/b on h, reset when rst
  tb = c*d
  a = /ta*a + ta*/a on h, reset when rst
  ta = b*c*d
end module
```

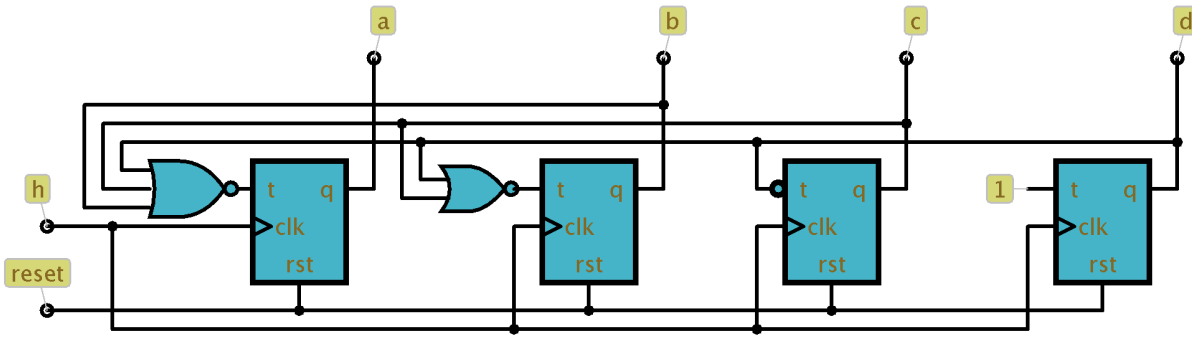
Figure III-73montre un exemple d'un tel compteur sur 4 bits.




```
module cpt4(rst,h: a,b,c,d)
  d = /d on h,reset when rst
  c = /d*c + d*/c on h, reset when rst
  b = /tb*b + tb*/b on h, reset when rst
  tb = c*d
  a = /ta*a + ta*/a on h, reset when rst
  ta = b*c*d
end module
```


Figure III-73. Compteur binaire 4 bits. Un bit est inversé lorsque tous les bits situés à sa droite valent 1.

Ce compteur compte bien sûr modulo 2^n : après la valeur '1111', suit la valeur '0000' sans qu'on puisse être prévenu de ce débordement.
On notera l'affectation séquentielle SHDL utilisée pour la bascule de poids faible d : d := /d; , qui se traduit par une bascule T avec un '1'sur l'entrée T. Pour le bit c, l'écriture c := d*c+/d*/c; se traduit également par une bascule T, mais avec une inversion avant son entrée. Les autres bascules T ont une écriture directement tirée de l'équation d'évolution standard d'une bascule T.
Dans la foulée, on peut également créer un circuit décompteur selon la même méthode : un bit est inversé lorsque tous les bits qui sont à sa droite valent 0 (Figure III-74).

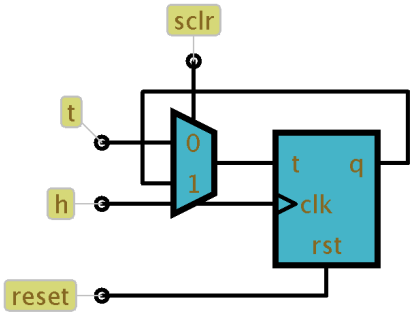



```
module dec4(rst, h : a, b, c, d)
  d = /d on h, reset when rst
  c = d*c + /d*/c on h, reset when rst
  b = /tb*b + tb*/b on h, reset when rst
  tb = /c*/d
  a = /ta*a + ta*/a on h, reset when rst
  ta = /b*/c*/d
end module
```

Figure III-74. Décompteur binaire 4 bits. Un bit est inversé lorsque tous les bits situés à sa droite valent 0

Ajout d'une remise à zéro synchrone

On souhaite maintenant ajouter une entrée sclr de remise à zéro synchrone, qui remet le compteur à zéro au prochain front d'horloge. Elle n'est bien sûr pas de même nature que l'entrée de reset asynchrone, qui n'est là que pour l'initialisation. Par ailleurs il ne faut pas essayer d'utiliser ce reset asynchrone pour implémenter notre remise à zéro synchrone.
La règle d'or des circuits séquentiels synchrones purs, c'est de relier entre-elles toutes les horloges et de relier entre eux tous les signaux de reset asynchrone, sans chercher à les modifier. Nous sommes heureux de pouvoir réinitialiser à tout coup notre ordinateur personnel en appuyant sur le bouton de reset, sans que cela ne dépende d'une condition plus ou moins bien pensée par un concepteur peu rigoureux.




```
module tclr(rst, h, sclr, t : q)
  q = /tq*q+tq*/q onh, reset when rst
  tq = /sclr*t+sclr*q
end module
```

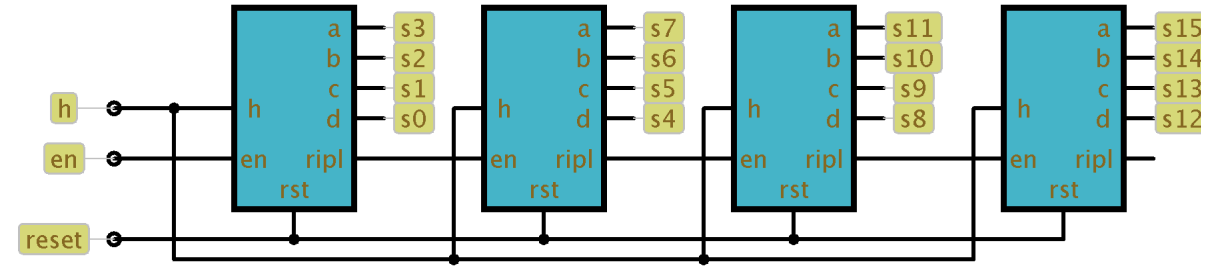
Figure III-75. Bascule T avec remise à zéro synchrone.

On notera l'écriture SHDL tq = /sclr*t+sclr*q; qui conduit à l'utilisation d'un multiplexeur 2 vers 1. Si sclr vaut 0, la bascule T reçoit son entrée t normalement ; si sclr vaut 1, la bascule T reçoit q en entrée, qui provoque la remise à 0 au prochain front d'horloge.
En effet, si la bascule avait l'état 0, en ayant 0 en entrée elle reste à l'état 0 ; si elle avait l'état 1, en recevant 1 en entrée, elle s'inverse et passe à 0 : dans tous les cas elle passe à zéro.

Chaînage de modules de comptage

Nous souhaitons maintenant chaîner plusieurs modules de comptage de n bits afin de former des compteurs de m×n bits. L'algorithmique de comptage par bloc est analogue à celle par bits : un bloc incrémente sa valeur si tous les blocs qui sont à sa droite sont à leur valeur maximum. Il nous manque donc pour chaque bloc un signal qui indique qu'il a atteint la valeur maximum. Pour économiser l'usage de portes ET, on va plutôt produire un signal ripl qui vaut 1 lorsque le compteur est à la valeur maximum et que en = 1. Pour chaîner les

modules, il suffit maintenant de relier le ripl de l'un au en de l'autre. La Figure III-76 montre le chaînage de 4 modules de 4 bits pour former un compteur 16 bits.



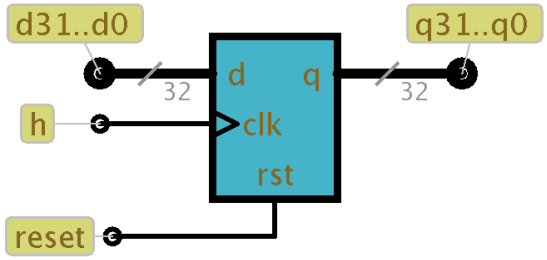

```
module cpt4(rst,h,en:a,b,c,d,ripl)
  d = /en*d+en*/d on h, reset when rst
  c = /tc*c+tc*/c on h, reset when rst
  tc = en*d
  b = /tb*b+tb*/b on h, reset when rst
  tb = en*c*d
  a = /ta*a+ta*/a on h, reset when rst
  ta = en*b*c*d
  ripl = en*a*b*c*d
end module
```

```
module cpt16(rst,h,en:s15..s0)
  cpt4(rst,h,en:s3,s2,s1,s0,ripl1)
  cpt4(rst,h,ripl1:s7,s6,s5,s4,ripl2)
  cpt4(rst,h,ripl2:s11,s10,s9,s8,ripl3)
  cpt4(rst,h,ripl3:s15,s14,s13,s12,ripl)
end module
```

Figure III-76. Compteur 16 bits formé avec 4 compteurs 4 bits. La sortie ripl d'un module indique qu'il est à la valeur maximum et que tous ceux qui sont à sa droite sont à leur maximum. Le module qui est à sa gauche a son entrée en reliée à ce signal, et ne peut s'incrémenter que lorsqu'il vaut 1.

IV.11. Registres

Un registre de n bits est un ensemble de n bascules D synchrones mises en parallèle et partageant la même horloge (Figure III-77).



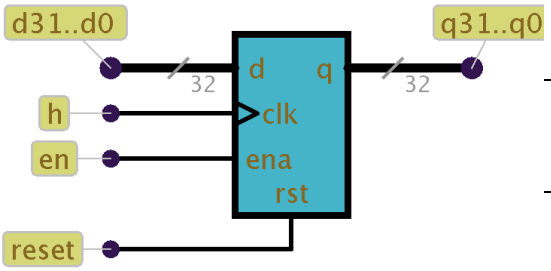

```
// Description SHDL d'un registre 32 bits
module reg32(d[31..0], h, reset: q[31..0])
  q[31..0] = d[31..0] on h, reset when reset
end module
```

Figure III-77. Registre de 32 bits avec son écriture SHDL.

On notera l'écriture vectorielle de type d[31..d0]. Elle est strictement équivalente à l'écriture d31..d0 qui a été utilisée jusqu'ici ; elle est plus courte lorsque le nom du signal est long, car il n'est mentionné qu'une fois.

Registre avec ligne de sélection

Un registre peut être équipé d'une ligne de sélection qui doit être activée si l'on veut faire une écriture ; ce type de signal a souvent pour nom en (enable), ou cs (chip select) ou encore ce (chip enable). On a vu à la section précédente que le langage SHDL disposait un modificateur optionnel pour prendre en compte directement un tel signal. La Figure III-78 met en œuvre cette écriture pour notre registre de 32 bits.




```
module reg32(d[31..0], h, ena, reset: q[31..0])
  q[31..0] = d[31..0] on h, reset when reset,
enabled when ena
end module
```

Figure III-78. Registre avec entrée de sélection, écriture SHDL directe.

IV.12. Circuit timer/PWM

Principe

Un circuit timer/PWM génère un signal cyclique rectangulaire dont les paramètres sont programmables. Ces paramètres sont :

- la période P, exprimée en nombre de cycles de l'horloge générale.
- la durée D à l'intérieur de cette période où la sortie vaut 0, exprimée en nombre de cycles.

La Figure III-79 illustre la forme du signal recherché.

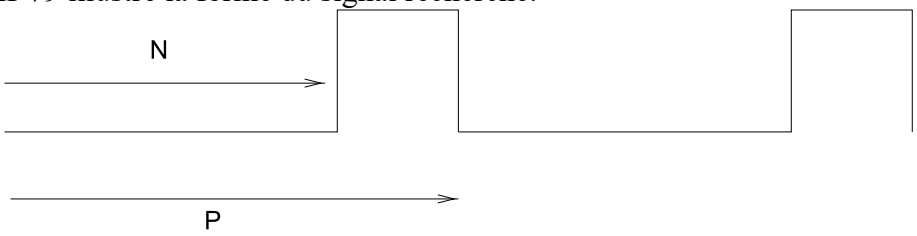


Figure III-79. Signal périodique généré par un circuit timer/PWM.

Le terme PWM (Pulse Width Modulation : modulation en largeur d'impulsion) fait référence à une technique utilisée fréquemment pour moduler l'énergie fournie à certains dispositifs tels que des lampes, des moteurs à courant continu, etc. Plutôt que de leur fournir une tension variable, difficile à produire avec précision, on leur fournit un signal périodique dont on fait varier le rapport cyclique entre les moments où la tension est maximale et ceux où la tension est nulle ($\frac{P-N}{P}$ sur la figure). La fréquence $\frac{1}{P}$ doit être assez élevée pour éviter les à-coups ou les clignotements, mais pas trop élevée pour éviter d'éventuels effets néfastes.
Par exemple, les LEDs à haute luminosité qui équipent les feux de stop et arrière des voitures ont leur luminosité commandée de cette manière.
Le circuit sera plutôt appelé 'timer' si on n'exploite que les fronts montants ou descendants du signal de sortie, pour générer une interruption par exemple.

Organisation

Pour réaliser un timer/PWM sur n bits, on utilise un compteur n bits et deux comparateurs n bits, qui comparent en permanence la valeur du compteur aux deux paramètres P et N (Figure III-80).
Le compteur dispose d'une remise à zéro synchrone (voir section 10.1), qui est activée dès que le compteur arrive à la valeur P. Ainsi il va compter 0, 1, …, P, c'est à dire avec une période de P+1 cycles d'horloge.

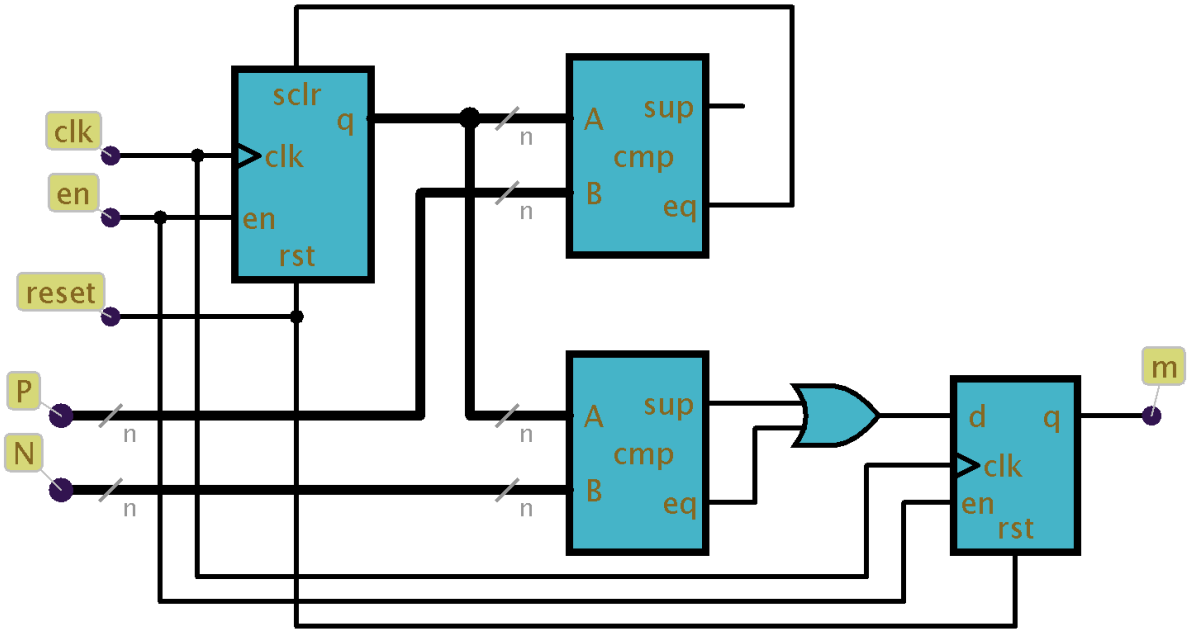


Figure III-80. Organisation d'un timer/PWM sur n bits. La valeur d'un compteur n bits est comparée aux valeurs de N et P ; quand il atteint P il est remis à 0 ; quand il dépasse N la valeur de la sortie m change.La bascule D finale prévient d'éventuels glitchs.

Durant cette période P+1, la valeur du compteur est comparée à la valeur de N, et la sortie m est le reflet de cette comparaison. La sortie m n'est pas directement le résultat combinatoire de la comparaison, mais on le fait passer au travers d'une bascule D, pour deux raisons :

- il pourrait être affecté de glitchs dans certaines situations ; en le faisant passer au travers d'une bascule D on est certain d'avoir un signal parfaitement propre.
- son timing est exactement aligné avec les fronts d'horloge, ce qui améliore la précision.

La Figure III-81 donne le code SHDL d'un tel circuit timer/PWM sur 4 bits. On réutilise les circuits de comptage et de comparaison déjà décrits en SHDL aux sections 10.1 et 8.9.

```
module pwm(rst, clk, en, p[15..0], n[15..0], m)

      count4Z(clk, rst, en, eqP : cnt[3..0], ripple)
      ucmp4(cnt[15..0], p[3..0] : supP, eqP)
      ucmp4(cnt[15..0], n[3..0] : supN, eqN)

      dm = supN + eqN
      m = dm on clk, reset when rst, enabled when en

end module
```

Figure III-81. Écriture SHDL d'un timer/PWM sur 4 bits.

La période p est égale à p3…p0 + 1 en nombre de cycles de l'horloge clk ; la valeur n3…n0 définit la valeur n. Le timing précis de ce module est défini par le chronogramme Figure III-82.

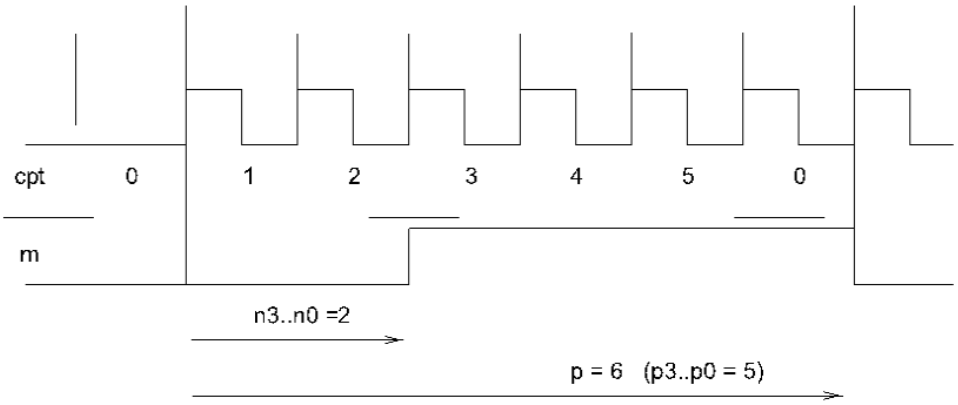
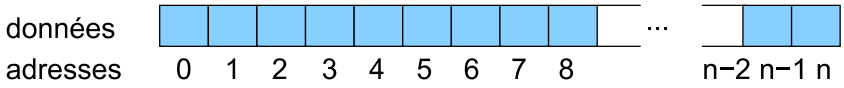


Figure III-82. Exemple de chronologie des signaux du timer/PWM sur 4 bits.

IV.13. RAMs et ROMs

IV.13.1. Introduction

Cette section est consacrée aux circuits utilisés dans la mémoire centrale des ordinateurs, qui contient les programmes et les données manipulés par les programmes. L'information y est stockée sous forme de mots de taille fixe, dont la largeur dépend de l'architecture utilisée. Ces mots mémoire sont ordonnés et possèdent chacun une adresse, sous forme d'un nombre. On peut donc voir une mémoire comme un ruban de cases numérotées :



…ou comme un tableau :

$$donnée = mem[adresse], adresse \in [0, n].$$

De telles mémoires se classent en deux grands types, selon qu'elles sont en lecture seule ou en lecture-écriture : les ***RAMs*** (Random Access Memory) peuvent être lues et écrites, et l'attribut random indique que ces lectures-écritures peuvent se faire à des suites d'adresses totalement quelconques, par opposition à des mémoires de type séquentiel (comme les disques durs) qui font des séries d'accès à des adresses consécutives. Les ***ROMs*** (Read Only Memory) sont fonctionnellement identiques aux RAMs, mais ne permettent que la lecture et pas l'écriture. Utilisées dans la mémoire centrale d'un ordinateur, les RAMs pourront stocker les programmes et les données, alors que les ROMs vont seulement stocker des programmes invariables, comme par exemple le programme exécuté au démarrage de la machine, et stocké dans la ROM dite de BIOS. Dans certains ordinateurs plus anciens, tout le système d'exploitation était stocké en ROM, ce qui permettait d'avoir un système incorruptible et au démarrage rapide.

Une RAM peut être ***statique*** ou ***dynamique***. Chaque bit mémoire d'une RAM statique (SRAM) est constitué d'une bascule, et conserve son état tant qu'elle est alimentée. A l'inverse, chaque bit d'une RAM dynamique (DRAM) est composé d'une capacité, qui doit être rafraîchie périodiquement par une électronique séparée. Les RAMs statiques ont un taux d'intégration plus faible que les RAM dynamiques, puisqu'un bit mémoire nécessite 6 transistors dans un cas, et une capacité plus un transistor dans l'autre.

Une RAM peut être ***synchrone*** ou ***asynchrone***, une RAM synchrone étant en fait une RAM asynchrone à laquelle on a ajouté une machine à états finis synchrone qui place les commandes de lecture et d'écriture dans un pipeline, afin de permettre d'accepter une nouvelle commande avant que la précédente n'ait été complétée.

Les barrettes de RAM de nos ordinateurs personnels sont des SDRAM, c'est-à-dire des RAM dynamiques synchrones, fonctionnant à des fréquences de 200MHz et plus. Elles sont souvent

de type DDR (double data rate), quand les fronts montants et descendants de l'horloge sont exploités pour les changements d'état.
Dans beaucoup d'autres appareils (assistants personnels, consoles de jeux, etc.), la RAM est de type statique asynchrone (SRAM), sous la forme de circuits intégrés. Les ROMs existent également dans un grand nombre de types différents, principalement selon la façon dont on peut programmer leur contenu (invariable, par définition). Il y a d'abord les ROMs programmées par masque à l'usine ; elles sont produites en grand nombre avec un faible coût à l'unité, mais leur contenu ne peut jamais être mis à jour ultérieurement.
Les ***PROMs*** (Programmable Rom) sont programmables par un appareil spécial, qui généralement détruit par un courant fort une liaison interne correspondant à un bit. Les EPROMs (Erasable PROM) fonctionnent de la même façon, mais possèdent une fenêtre transparente et peuvent être effacées par une exposition d'une vingtaine de minutes aux rayons ultraviolets. Elles sont maintenant souvent remplacées par des ***EEPROMs*** (Electrically EPROM), reprogrammables électriquement. Les ***mémoires Flash*** sont également une forme de mémoires effaçables électriquement, mais on réserve généralement le terme EEPROM aux mémoires capables d'effacer à l'échelle du mot, et le terme 'mémoires Flash' à celles qui effacent à l'échelle de blocs. Dans les deux cas, le temps d'effacement est long par rapport au temps de lecture, et elles ne peuvent pas être considérées comme une forme spéciale de RAM.
On trouve de la mémoire EEPROM et flash dans les assistants personnels, dans les sticks mémoire sur ports USB, pour le stockage du firmware de nombreux appareils (BIOS d'ordinateurs, lecteurs de DVD, tuners satellites, etc.)

IV.13.2. Mode d'emploi des mémoires statiques asynchrones

Les mémoires statiques ont généralement une organisation et un mode d'emploi spécifique. On a vu à la section précédente que chaque bit d'une telle mémoire était stocké dans une bascule, ce qui conduit pour les formes les plus simples à séparer les données à écrire des données lues, de la même façon que sont séparées l'entrée et la sortie d'une bascule.
La Figure III-83 montre l'interface d'un tel circuit.

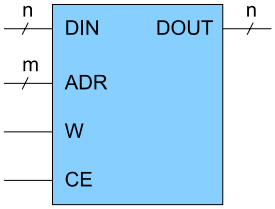


Figure III-83. Interface d'un boîtier de RAM statique asynchrone de 2^m mots de n bits, avec séparation des données lues et des données à écrire.

Le mode d'emploi de ce circuit est le suivant :

Lecture d'un mot mémoire

- activer la ligne CE (Chip Enable), placer l'adresse du mot sur les m lignes d'adresse ADR et garder inactive la ligne W (write).
- après un certain temps appelé temps d'accès en lecture, le mot mémoire (de largeur n) placé à cette adresse est disponible sur les lignes DOUT (Data Out).

Écriture d'un mot mémoire

- activer la ligne CE (Chip Enable), placer l'adresse du mot sur les m lignes d'adresse ADR, placer la donnée à écrire sur les n lignes DIN (Data In) et activer la ligne W (write).
- après un certain temps appelé temps d'écriture, le mot (de largeur n) est écrit en mémoire, et pourra être lu ultérieurement.

Le temps d'écriture est généralement voisin du temps d'accès en lecture ; ces temps varient d'une dizaine de nanosecondes pour les mémoires statiques les plus rapides à une centaine de nanosecondes.
En pratique, les boîtiers disponibles mettent en commun les lignes DIN et DOUT, de façon à minimiser le nombre de broches, et à faciliter l'interconnexion de plusieurs boîtiers. De tels circuits ont l'interface et la structure suivante :

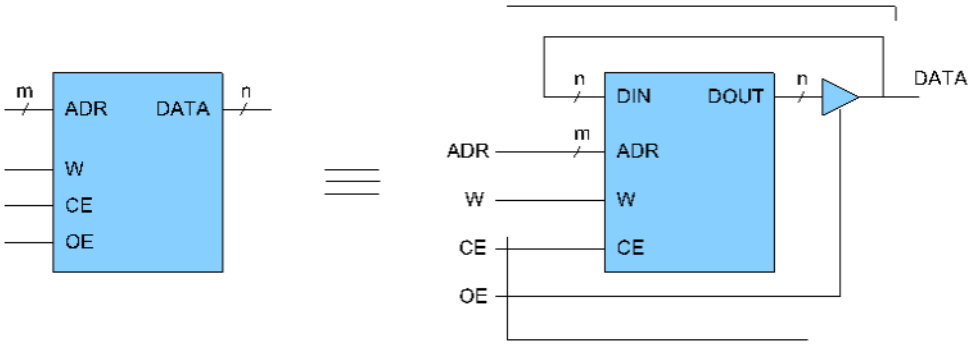


Figure III-84. Interface d'un boîtier de RAM statique asynchrone de 2^m mots de n bits, avec données lues et écrites communes.

Une entrée supplémentaire OE (Output Enable) est nécessaire, qui fonctionne de la façon décrite en début de chapitre. Lors de l'écriture d'un mot mémoire, la ligne OE doit être inactive, et la donnée à écrire peut être placée sur DATA sans risque de court-circuit. Lors d'une lecture en mémoire, la ligne OE doit être activée après le temps d'accès en lecture pour que la donnée lue soit disponible sur DATA. On verra plus loin comment interconnecter ces circuits ; les précautions déjà évoquées doivent être respectées ici, et en particulier le risque qu'il y a d'avoir à la fois OE active et une donnée présente en entrée de DATA, qui peut conduire à la destruction du circuit.

IV.13.3. Mode d'emploi des mémoires dynamiques asynchrones

Les mémoires dynamiques ont généralement une organisation légèrement différente des mémoires statiques, même si cette différence n'est pas à attribuer à leur nature statique ou dynamique. Leur interface est celui de la Figure III-85.

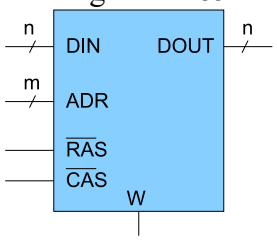


Figure III-85. Interface d'un boîtier de RAM dynamique asynchrone de 2^{2m} mots de n bits.

On remarque d'abord qu'il n'y a pas de ligne de validation (CE ou CS), et que deux nouvelles lignes RAS et CAS sont présentes. La légende de la figure parle de 2^{2m} mots de n bits, alors qu'on ne voit que m lignes d'adresses. On va voir qu'en fait, ces m lignes sont multiplexées, et qu'on y présente successivement un numéro de ligne sur m bits, puis un numéro de colonne sur m bits, pour accéder à une cellule mémoire dans une matrice de 2^m x 2^m = 2^{2m} cellules mémoire. La présentation des numéros de ligne et de colonne est synchronisée par les signaux RAS et CAS.
RAS signifie sélection de ligne d'adresse (Row Address Select); CAS signifie sélection de colonne d'adresse (Column Address Selection). Le boîtier est inactif tant que RAS et CAS sont inactifs. Les chronologies de lecture et d'écriture sont montrées Figure III-86 et Figure III-87.

Pour une lecture ou une écriture, on place d'abord un numéro de ligne sur ADR, qui est obtenu en prenant la moitié de l'adresse complète, généralement les poids forts et on active ensuite la ligne RAS (front descendant). On place ensuite sur ADR l'autre moitié de l'adresse complète, généralement les poids faibles, et on active ensuite la ligne CAS (front descendant). En cas de lecture (W = 1) la donnée sera disponible sur DOUT ; en cas d'écriture il faudra placer la donnée sur DIN.

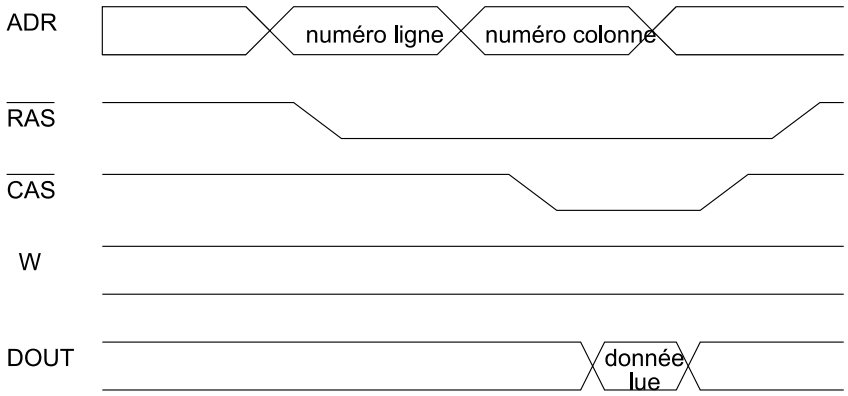


Figure III-86. Chronogramme de lecture dans une mémoire dynamique.

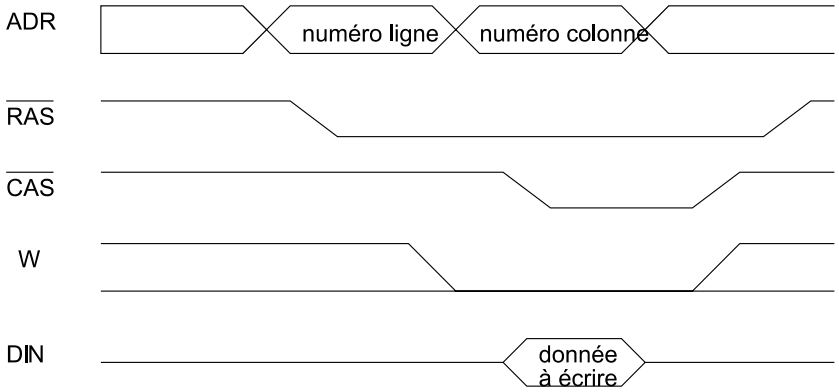


Figure III-87. Chronogramme d'écriture dans une mémoire dynamique.

Un cycle spécifique de rafraîchissement existe également, qui opère sur des lignes entières de cellules. Chaque ligne doit être rafraîchie typiquement toutes les 5ms, et cela n'est pas fait automatiquement par le circuit : c'est la logique de commande qui est autour qui doit s'en charger.

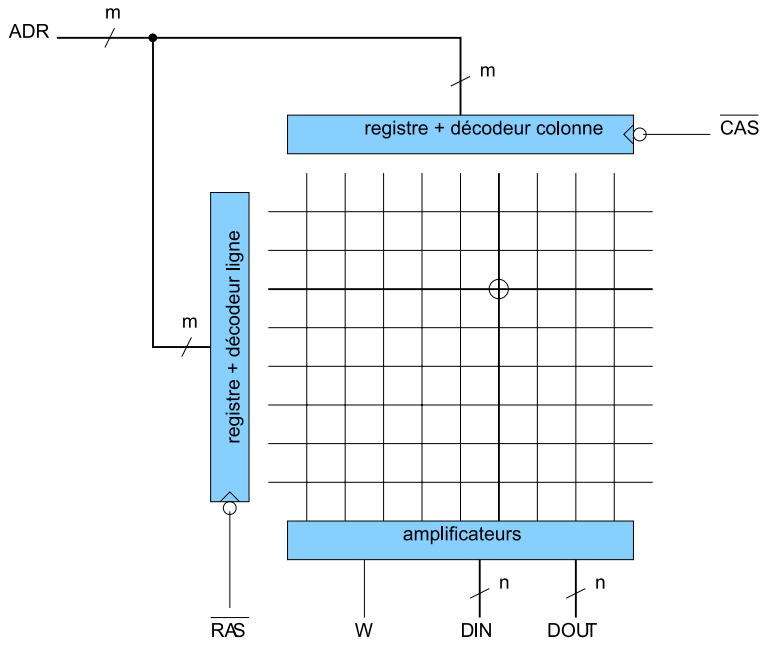


Figure III-88. Structure interne d'une mémoire dynamique de 2^{2m} mots de n bits.

Un tel mode de fonctionnement correspond à la structure interne de la Figure III-88. On voit que, lorsque le numéro de ligne est présenté sur ADR et qu'un front descendant est appliqué sur RAS, ce numéro de ligne est stocké dans le registre de ligne. Sa valeur est décodée et sélectionne une des m 2 lignes de la matrice. De la même façon, lorsque le numéro de colonne est présenté sur ADR et qu'un front descendant est appliqué sur CAS, ce numéro est stocké dans le registre de colonne, dont la valeur est également décodée et sélectionne une des colonnes. Une des cellules mémoire est ainsi sélectionnée. En cas de lecture (W = 1) la donnée sera disponible sur DOUT ; en cas d'écriture il faudra placer la donnée sur DIN. La mise en œuvre de ces circuits est plus complexe que celle des mémoires asynchrones, puisqu'on ne peut plus présenter l'adresse entière directement. On est obligé de la présenter en deux fois, généralement poids forts d'abord (numéro de ligne) et poids faibles ensuite (numéro de colonne), ce qui peut être fait à l'aide d'un multiplexeur.

IV.13.4.Association de mémoires statiques asynchrones

première association

Il faut d'abord remarquer que dans les deux cas, la mémoire obtenue a bien un nombre total de bits mémoire égal à la somme des bits mémoire des deux parties. Chaque boîtier initial possède m × 2n bits mémoire, et leur somme fait donc m × 2n+1. Dans la première association, on aura 2^n mots de 2m bits soit $2 \times m \times 2^n = m \times 2^{n+1}$; dans la deuxième on aura aussi les $m \times 2^{n+1}$ bits.
Pour former une mémoire de 2^n mots de 2m bits, il faut mettre essentiellement toutes les lignes des deux boîtiers en commun, à l'exception des lignes de données qu'il faut mettre en parallèle. Cela donne le circuit de la Figure III-89.

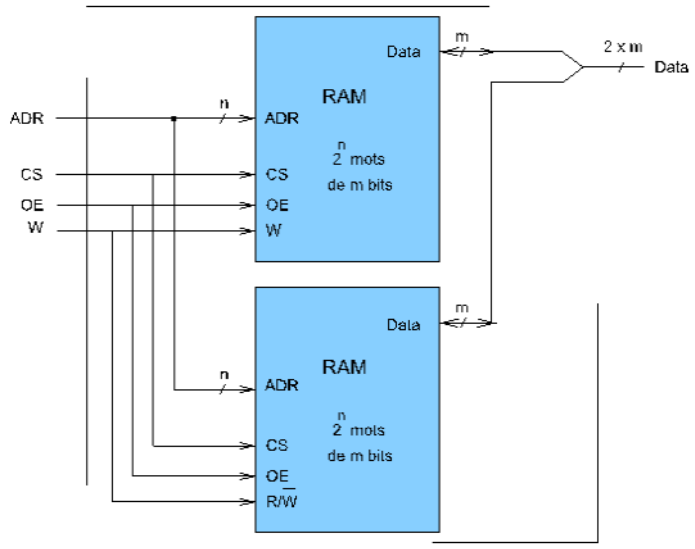


Figure III-89. Association de mémoires asynchrones, avec doublement de la largeur des données.

Lorsqu'un accès mémoire (lecture ou écriture) est effectué, les deux boîtiers sont sélectionnés en même temps à la même adresse, et fournissent ou acceptent simultanément les deux moitiés de largeur m de la donnée globale de largeur 2m.

deuxième association

Pour former une mémoire de 2^{n+1} mots de m bits, il va falloir mettre en commun les m lignes de données, et donc il ne va plus être possible de sélectionner les deux boîtiers en même temps, sous peine de court-circuit. Pour une moitié des adresses, il faudra sélectionner le premier boîtier, et l'autre pour l'autre moitié des adresses. La manière la plus simple est ici d'utiliser un des bits de l'adresse de n + 1 bits, par exemple adr[n], pour faire cette sélection (Figure III-90).

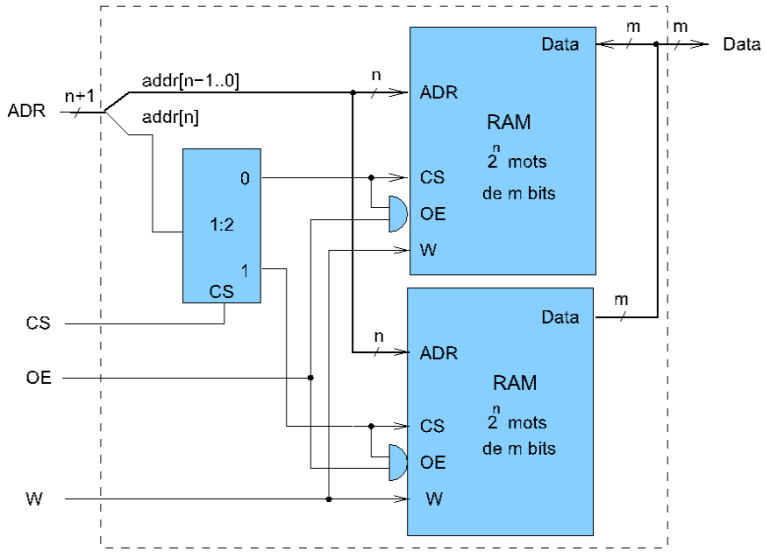


Figure III-90. Association de mémoires asynchrones, avec doublement du nombre d'adresses.

Ce bit est mis en entrée d'un décodeur 1 vers 2 : s'il vaut 0, c'est le boîtier mémoire du haut qui est sélectionné ; s'il vaut 1 c'est le boîtier du bas. Tout cela suppose que le signal CS global soit actif, car s'il ne l'est pas, le décodeur ne sélectionne aucun boîtier Le signal OE suit un traitement particulier : pour chaque boîtier, il ne doit être actif que si le OE global est actif, et si le boîtier est sélectionné (CS = 1). On est alors assuré qu'aucun court-circuit n'est possible, puisque les OE des deux boîtiers sont alors mutuellement exclusifs.

Dans cette configuration, l'espace des adresses de l'ensemble est [0 , $2^{n+1} - 1$] ; la première moitié [0, $2^n - 1$] correspond au premier boîtier ; la seconde [2^n, $2^{n+1} - 1$] correspond au second. Si on avait utilisé le bit de poids faible de l'adresse globale comme entrée du décodeur, la distribution des adresses entre les boîtiers aurait été différente : l'un aurait contenu les adresses paires, et l'autre les adresses impaires.

IV.14. Exercices corrigés

IV.14.1. Exercice 1 : système de sécurité

Énoncé

Réaliser un système de mise en marche de sécurité de machine dangereuse. Il est équipé de deux entrées A et B et la mise en marche est contrôlée par la sortie M. Il faut pour cela que la procédure suivante soit respectée :

1. A et B doivent être relâchés initialement ;
2. appuyer sur A,
3. appuyer sur B : la machine se met en marche.

Toute autre manipulation arrête ou laisse la machine arrêtée ; il faut ensuite reprendre la procédure au point 1 pour la mettre en marche.

Solution

On va adopter une solution de type MOORE. On fera également une conception synchrone, c'est à dire qu'on supposera présente une horloge suffisamment rapide pour ne rater aucun événement sur A et B. Le graphe d'état de ce problème est donné Figure III-91.

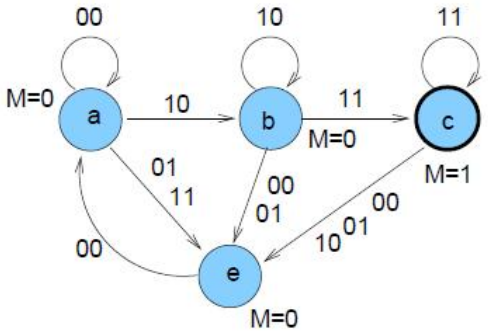


Figure III-91. Graphe d'état du système de sécurité, type MOORE

En partant de l'état a, on progresse jusqu'à l'état c pour lequel M=1 si on respecte la séquence de mise en marche. Sur ce chemin, on tombe dans l'état d'erreur e dès qu'on s'écarte de la procédure. Lorsqu'on est en e, on y reste tant que A et B ne sont pas relâchés tous les deux, conformément à la spécification de l'énoncé.
On réécrit ce graphe sous une forme tabulaire pour chercher s'il est simplifiable (Figure III-92).

avant			*après*
état	A	B	état
a	0	0	a
a	0	1	e
a	1	0	b
a	1	1	e
b	0	0	e
b	0	1	e
b	1	0	b
b	1	1	c
c	0	0	e
c	0	1	e
c	1	0	e
c	1	1	c
e	0	0	a
e	0	1	e
e	1	0	e
e	1	1	e

état	S
a	0
b	0
c	1
e	0

Figure III-92. Table de transitions du graphe de sécurité

Aucune simplification n'est possible ; le circuit possède donc un nombre minimal d'états de 4. Il faut donc 2 bascules pour coder cet état, dont on appellera les sorties X et Y.
Vient maintenant l'étape d'assignation, durant laquelle nous devons choisir une configuration des bascules X et Y pour chacun des états. On assignera à l'état a le vecteur XY=00, associer au reset des bascules. Pour les autres, il est souhaitable pour obtenir des calculs plus simples de respecter les heuristiques d'assignation, qui recommandent de placer côte à côte les états fortement reliés dans le graphe. On peut par exemple choisir l'assignation de la Figure III-93.

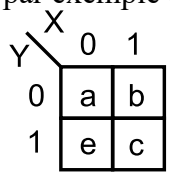


Figure III-93. Assignation des états du système de sécurité

On peut maintenant réécrire la table de transition, en remplaçant les états symboliques par leur vecteur associé (Figure III-94).

avant			*après*
état	A	B	état
00	0	0	00
00	0	1	01
00	1	0	10
00	1	1	01
10	0	0	01
10	0	1	01
10	1	0	10
10	1	1	11
11	0	0	01
11	0	1	01
11	1	0	01
11	1	1	11
01	0	0	00
01	0	1	01
01	1	0	01
01	1	1	01

état	S
00	0
10	0
11	1
01	0

Figure III-94. Table de transitions instanciée du graphe de sécurité

Aucune bascule ne semble particulièrement adaptée à ces changements d'état. À titre d'exemple, on va utiliser deux bascules T. On peut à nouveau écrire la table de transition instanciée, en rajoutant 2 colonnes TX et TY qui sont les entrées T des bascules X et Y (Figure III-95).

		avant			*après*
état	A	B	TX	TY	état
00	0	0	0	0	00
00	0	1	0	1	01
00	1	0	1	0	10
00	1	1	0	1	01
10	0	0	1	1	01
10	0	1	1	1	01
10	1	0	0	0	10
10	1	1	0	1	11
11	0	0	1	0	01
11	0	1	1	0	01
11	1	0	1	0	01
11	1	1	0	0	11
01	0	0	0	1	00
01	0	1	0	0	01
01	1	0	0	0	01
01	1	1	0	0	01

état	S
00	0
10	0
11	1
01	0

Figure III-95. Table de transitions instanciée du graphe de sécurité, avec les entrées des 2 bascules T.

Il suffit maintenant de trouver les formules algébriques pour TX et TY, en fonction de X, Y, A et B. On peut pour cela utiliser des tables de Karnaugh (Figure III-96). On trouve :

1. $TX = A\bar{B}\bar{X}\bar{Y} + \bar{A}X + \bar{B}XY$
2. $TY = \bar{A}\bar{B}\bar{X}Y + B\bar{Y}X + \bar{A}X\bar{Y}$
3. $M = XY$

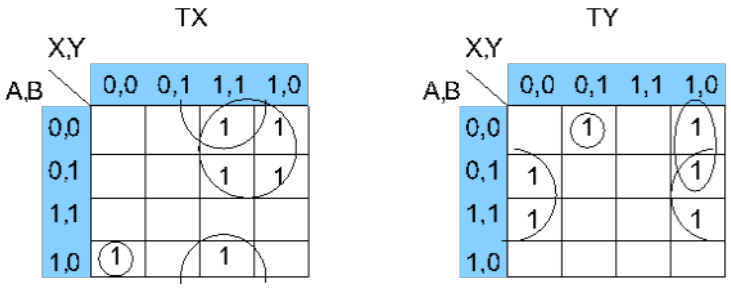


Figure III-96.Simplification des équations du système de sécurité

IV.14.2.Exercice 2 : compteur/décompteur

Énoncé

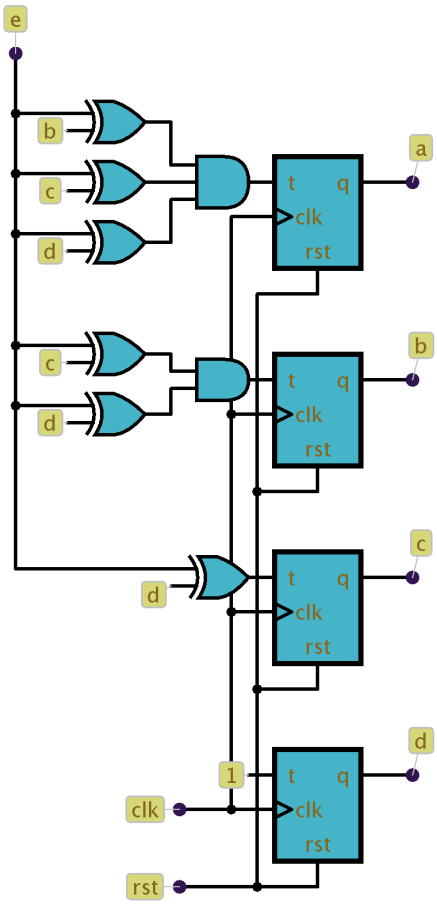
Concevoir un circuit séquentiel synchrone avec une entrée e et 4 sorties a,b,c,d, qui compte en binaire sur a,b,c,d lorsque e=0 et qui décompte lorsque e=1.

Solution

On pourrait concevoir ce circuit en construisant un graphe d'états, une table de transitions, etc. Mais cette méthode ne serait pas générique et avec 4 bits le graphe serait déjà de grande taille avec 16 états. Il est préférable ici de combiner deux circuits que l'on connaît déjà, le compteur et le décompteur.

On a déjà présenté les compteurs et les algorithmes de comptage et de décomptage en binaire (section 4). On réalise un compteur binaire 4 bits avec 4 bascules T ; la bascule de poids faible a pour entrée 1 car elle change d'état à chaque front d'horloge ; les autres bascules ont pour entrée le ET des bits qui sont à leur droite. Un décompteur se fait également avec 4 bascules T ; la bascule de poids faible change également à chaque front ; les autres bascules changent d'état lorsque tous les bits qui sont à leur droite sont des 0.

La synthèse des deux est donc simple : la bascule de poids faible est commune, car elle change d'état à chaque front d'horloge dans les deux cas. Pour les autres bits, il s'agit également de faire un ET dans les deux cas, mais des bits qui sont à droite pour le comptage et de l'inverse de ces mêmes bits pour le décomptage. Il suffit donc de mettre un inverseur commandé (un XOR, voir section 4) par le signal e devant chacun des bits à prendre en compte dans le ET. Cela conduit au schéma de la Figure III-97, avec un code SHDL immédiat.




```
module cptdec(rst, h, e : a, b, c, d)
  // bascule a
  a := /ta*a+ta*/a on clk, reset when rst
  ta = nb * nc * nd
  // bascule b
  b = /tb*b+tb*/b on clk, reset when rst
  tb = nc * nd
  // bascule c
  c = /tc*c+tc*/c on clk, reset when rst
  tc = nd
  // bascule d
  d = /d on clk, reset when rst
  // inversion de a,b,c,d commandée par e
  na = a*/e + /a*e
  nb = b*/e + /b*e
  nc = c*/e + /c*e
  nd = d*/e + /d*e
end module
```

Figure III-97. Compteur/décompteur : schéma et écriture SHDL

IV.14.3. Exercice 3 : utilisation des bascules

Énoncé

Construire une bascule D avec une bascule T, puis une bascule T avec une bascule D. On pourra utiliser des circuits combinatoires.

Solution

1. Bascule D avec une bascule T.

On peut se poser une question simple : comment mettre à 0 une bascule T ? Il suffit de mettre la valeur Q de la bascule sur son entrée T. Si elle valait 0 elle y reste et si elle valait 1 elle s'inverse et passe à 0.
Pour mettre à 1une bascule T, on voit aussi facilement qu'il suffit d'appliquer l'inverse de Q sur l'entrée T.
Dans les deux cas, on a appliqué sur l'entrée T la valeur Q de la bascule, avec une inversion commandée par la valeur D à mémoriser : XOR(D,Q).

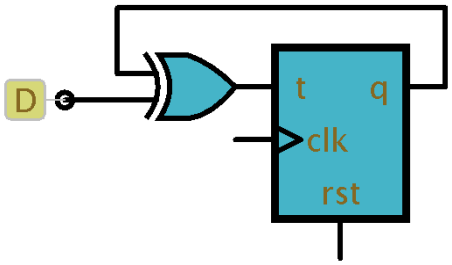


Figure III-98. Bascule D avec une bascule T

2. Bascule T avec une bascule D.

La solution est ici plus directe : il suffit d'appliquer sur D l'équation d'évolution de la bascule T, qui est $Q := \bar{T}\,Q + T\bar{Q}$

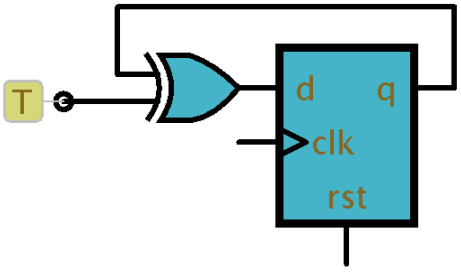


Figure III-99. Bascule T avec une bascule D

Chapitre V. Programmation du processeur 32 bits CRAPS/SPARC

Le ***langage machine*** (et sa forme équivalente symbolique dite ***assembleur***) est exécuté directement par le processeur, et il est à ce titre l'intermédiaire obligé de tous les composants logiciels d'un ordinateur. Même s'il est vrai que peu de gens programment directement en assembleur, son étude est éclairante pour la compréhension de la relation entre logiciel et matériel. Le but de ce présent chapitre est d'initier à la programmation en assembleur du processeur CRAPS au travers de programmes de différentes natures, de façon à ensuite mieux comprendre son architecture et son fonctionnement.

V.1. Le langage machine, à la frontière entre logiciel et matériel

Langage machine et ISA

Les instructions qu'exécute un processeur sont écrites dans un langage appelé ***langage machine***, et sont formées de codes écrits dans des mots mémoires consécutifs ; on appelle ***ISA*** (Instruction Set Architecture) ce langage et son organisation. L'informatique utilise un grand nombre de langages différents : des langages de programmation tels que C ou Java pour les programmeurs, des langages interprétés tels que les langages de macro des suites bureautiques, des langages de gestion de bases de données, etc. Mais la particularité du langage machine, c'est d'être **le** langage intermédiaire en lequel tous les autres langages vont être traduits, en une ou plusieurs étapes. Il est situé à la frontière entre le processeur et les logiciels, entre ***hardware*** et ***software***, et le processeur est ***l'interpréteur*** de ce langage (Figure III-100), celui qui met en action ses instructions.

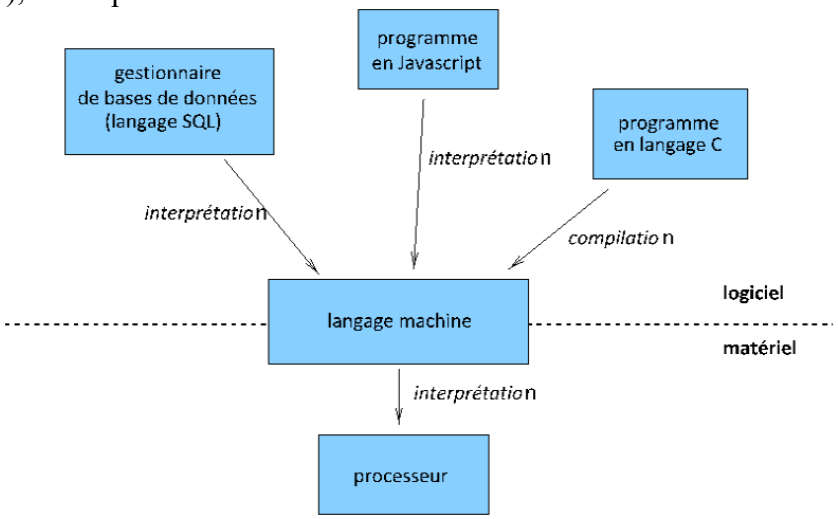


Figure III-100- Tous les programmes produisent du langage machine, qui est exécuté directement par le processeur.

ISA générale ou spécialisée ?

Bien sûr, il serait plus efficace d'avoir un processeur qui exécute directement un langage évolué, tout au moins pour les programmes écrits dans ce langage. De tels processeurs existent ou ont existé, notamment pour le langage Lisp, et plus récemment pour le langage intermédiaire (bytecode) utilisé en Java. Ces deux langages sont généralistes et peuvent être utilisés dans toutes les situations concrètes de l'informatique, et l'espoir des concepteurs de ces processeurs était d'élever au niveau de ces langages la frontière entre matériel et logiciel. Mais le bytecode Java exploite une machine à base de pile, peu adaptée à la compilation d'autres langages de programmation tels que C. Finalement, le compromis généralement

adopté lors de la conception d'un nouveau processeur est le développement d'une ISA adaptée à la compilation de la plupart des langages de programmation courants.

Évolution des ISA

Comme toute structure située à une frontière, l'ISA est un équilibre entre des forces opposées. Les concepteurs de compilateurs souhaitent une ISA régulière et possédant le plus de possibilités en termes d'adressage mémoire, d'opérations arithmétiques, etc., alors que les architectes des processeurs veulent une ISA simple à implémenter efficacement.
Une force importante qui limite l'apparition de nouvelles ISAs (de nouveaux langages machine) est la demande des clients d'une ***compatibilité ascendante*** d'un processeur d'une génération à l'autre, de façon à pouvoir faire fonctionner sans changement leurs anciens programmes. Cette force se manifeste de façon très puissante avec les processeurs de PC (dont l'ISA est souvent désignée par x86, parce que les noms des processeurs se terminent par 86 : 8086, 80286, etc.), pour laquelle tout changement conduisant à une non compatibilité est exclu. Ainsi, les processeurs x86 les plus récents continuent à manifester la bizarrerie d'écriture en mémoire inversée de type ***little-endian***, parce qu'elle était présente à l'origine sur le processeur 8 bits Intel 8080 pour des motifs de simplicité de conception, et qu'elle s'est transmise pour raison de compatibilité tout le long de la chaîne 8080, 8085, 8086, 80286, 80386, Pentium, Pentium 2, etc. jusqu'aux processeurs des PCs actuels.
D'autres processeurs moins connus ont également une lignée très longue et respectable. On peut citer le processeur ARM, un des premiers processeurs 32 bits commercialisés, qui fait évoluer son ISA de façon ascendante depuis les années 80 jusqu'à maintenant, et qui équipe un grand nombre de téléphones portables, les consoles de jeux portables Nintendo et un grand nombre d'appareils mobiles.

Processeurs CISC et RISC

Une nouvelle génération de processeurs est apparue dans les années 80, caractérisée par un jeu d'instruction limité à l'essentiel, d'où leur nom de processeurs ***RISC*** (Reduced Instruction Set Computer). Par ailleurs, leur ISA est généralement très régulière, contient beaucoup de registres utilisateurs, et les instructions sont codées sous forme d'un mot mémoire unique. Tout cela rend heureux les architectes du processeur, qui peuvent concevoir des architectures très efficaces pour cette ISA, et ne rend pas trop malheureux les concepteurs de compilateurs, qui ont un peu plus de difficulté à traduire les programmes, mais qui apprécient le gain en vitesse d'exécution. Par opposition, les anciens processeurs tels que les x86 ont été rebaptisés processeurs ***CISC*** (Complex Instruction Set Computer), et si ce n'était l'énorme investissement dû au succès du PC avec l'ISA x86, les processeurs RISC auraient connus un développement plus immédiat et plus large.

V.2. Ressources du programmeur CRAPS/SPARC

Le langage machine de CRAPS est un sous ensemble du langage machine du SPARC de SUN (au cas où vous ne l'auriez pas encore remarqué, 'CRAPS'= 'SPARC' à l'envers !), avec un certain nombre de limitations. C'est donc un langage de type RISC, qui sera partiellement compatible avec le SPARC au niveau binaire. Néanmoins il n'exploite pas la notion SPARC de fenêtre de registres et il n'a pas d'instruction d'arithmétique flottante.
Afin de rendre plus concrètes les descriptions qui sont suivre, on va progressivement programmer le petit algorithme suivant qui fait la somme des 10 éléments d'un tableau :

```
résultat <- 0 ;
pour i de 0 à 9 faire
      résultat <- résultat + TAB[i] ;
fin pour
```

V.2.1. Types de données de CRAPS

CRAPS est un processeur 32 bits : cela signifie que les mots mémoire qu'il manipule le plus directement ont 32 bits de large, et que tous ses registres sont de 32 bits.
Les entiers signés sont codés en complément à 2 et les opérations arithmétiques disponibles sont l'addition et la soustraction sur 32 bits, la multiplication 16 bits X 16 bits vers 32 bits et la division 32 bits / 16 bits vers 32 bits. Il ne dispose pas de registres pour les nombres flottants, ni d'instruction d'arithmétique flottante. Si des calculs sur des réels doivent être effectués, ils doivent être exécutés par des librairies logicielles spécialisées.
Aucun autre type de données n'est explicitement supporté. Les chaînes de caractères, les listes etc. devront être assemblées et manipulées en combinant les opérations sur les octets et les mots de 32 bits.

V.2.2. Les registres de CRAPS

Les registres banalisés

CRAPS possède 32 ***registres*** directement accessibles au programmeur (Figure III-101). On les note %r0, …%r31. Ils sont tous accessibles librement en lecture et en écriture, mais %r0 a un statut spécial : il vaut toujours 0 en lecture, quelles que soient les écritures qu'on fasse dedans. Cette bizarrerie est en fait très commode et on verra par la suite comment elle est exploitée.

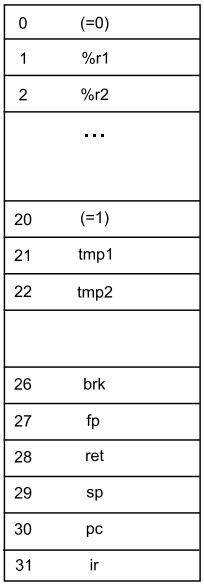


Figure III-101. Les registres du processeur CRAPS. Ils sont tous accessibles au programmeur, mais certains ont déjà des fonctions assignées par le processeur.

r31 sera utilisé comme registre instruction par le processeur et r30 comme compteur ordinal ; r29 sera généralement le pointeur de pile et r28 servira à stocker l'adresse de retour des instructions call (mais ces conventions ne sont pas imposées par le matériel).
Le programmeur dispose donc d'un nombre important de registres qu'il peut utiliser à sa guise dans ses opérations locales, ce qui est d'un grand confort et permet de limiter grandement le nombre d'accès à la mémoire centrale.
Notre algorithme se transforme de la façon suivante :

```
%r1 <- 0;
pour %r2 de 0 à 9 faire
   %r3 <- TAB[%r2] ;
   %r1 <- %r1 + %r3 ;
fin pour
```

Trois registres utilisateurs sont déjà mobilisés, et ce n'est qu'un début ! On voit l'intérêt qu'offre au programmeur une ISA fournissant beaucoup de registres utilisateurs.

V.2.3. Le modèle mémoire de CRAPS

Comme la plupart des processeurs actuels, l'ISA de CRAPS voit la mémoire sous forme d'un ruban ***d'octets*** situés à des adresses consécutives (Figure III-102).

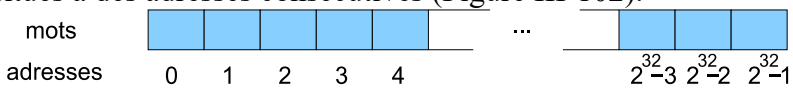


Figure III-102. La mémoire de CRAPS est un ruban de mots, numérotés par des adresses allant de 0 à $2^{32} - 1$.

Les adresses seront présentes dans CRAPS sous forme de mots de 32 bits, définissant ainsi une mémoire centrale pouvant compter jusqu'à 232 cases, soit 4 giga-octets, les adresses allant de 0 à $2^{32} - 1$. Le fait d'avoir 32 lignes d'adresses sera commode dans CRAPS pour manipuler les adresses à l'intérieur des registres. Ce choix n'était pas obligatoire, et on aurait pu imaginer un nombre de lignes d'adresses inférieur ou supérieur à la taille du mot mémoire du processeur, ici 32.
En pratique, l'ensemble de cet espace d'adressage n'est souvent pas entièrement occupé par de la mémoire physique. Un mécanisme de décodage des adresses que l'on étudiera au chapitre suivant permettra d'associer un bloc d'adresses particulier à un boîtier mémoire particulier ; par ailleurs la lecture et l'écriture à certaines adresses ne conduira pas à un simple stockage, mais déclenchera la commande de certains circuits, tels que le timer ou le circuit d'entrées-sorties, selon un mécanisme appelé ***entrées/sorties mappées en mémoire***, dont on étudiera l'implémentation au chapitre suivant, mais que l'on apprendra à utiliser en tant que programmeur un peu plus loin dans ce chapitre.
Reprenons notre petit algorithme de somme des nombres d'un tableau. On va désigner par mem[adr] l'octet en mémoire situé à l'adresse adr. Notre tableau TAB en mémoire centrale est formé de mots de 32 bits situés consécutivement.
Notre algorithme devient :

```
%r1 <- 0 ;
%r4 <- adresse de TAB ;
pour %r2 de 0 à 9 faire
   %r3 <- mem[%r4+%r2] ;
   %r1 <- %r1 + %r3 ;
fin pour
```

V.2.4. Les instructions de CRAPS

Toutes les instructions de CRAPS sont constituées d'un seul mot de 32 bits. Cette caractéristique des instructions d'avoir une taille fixe d'un mot est typique des processeurs RISC, et elle facilite grandement certains aspects de leur conception. Les instructions de

CRAPS sont formées de plusieurs groupes, chacun ayant une certaine structure de codage (Figure III-103).

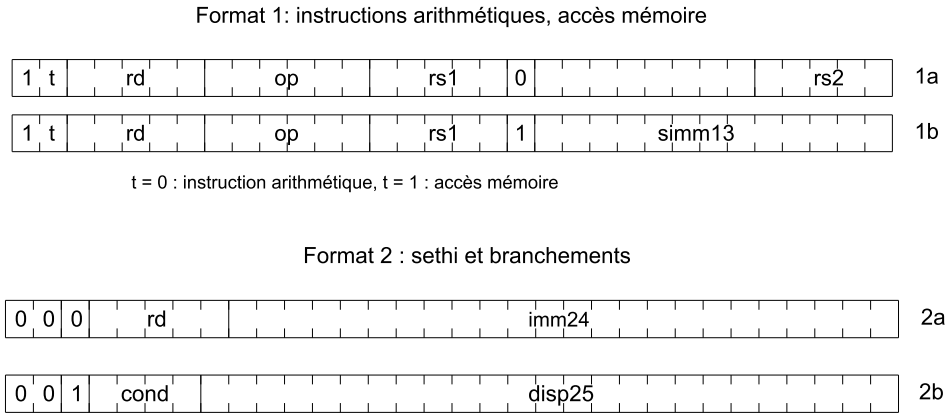


Figure III-103. Structure binaire des différents groupes d'instructions de CRAPS.

La valeur des deux bits de poids fort permet d'identifier immédiatement le groupe de l'instruction. Le tableau de la Figure III-104 donne la signification de leurs valeurs pour le processeur CRAPS.

op (t=0)	Instr.
000000	add
010000	addcc
000100	sub
010100	subcc
011010	umulcc
000001	and
010001	andcc
000010	or
010010	orcc
000011	xor
010011	xorcc
001101	slr
001110	sll

op (t=1)	Instr.
000000	ld
000100	st

cond	branchement
1000	ba
0001	be
1001	bne
0101	bcs
1101	bcc
1110	bpos
0110	bneg
0111	bvs
1111	bvc
1010	bg
0010	ble
1011	bge
0011	bl
1100	bgu
0100	bleu

Figure III-104. *Significations des valeurs d'opcodes.*

Les différentes combinaisons possibles sur les formats d'instructions conduisent au tableau d'instructions de la Figure III-105.

load/store	
ld [%ri+op], %rj st %ri, [%rj+op]	lecture mémoire 32 bits écriture mémoire 32 bits

contrôle	
ba addr bcc addr	branchement inconditionnel branchement conditionnel

calcul	
add %ri, op, %rj	addition %rj <- %ri + op
addcc %ri, op, %rj	addition %rj <- %ri + op (avec flags)
sub %ri, op, %rj	soustraction %rj <- %ri + op
subcc %ri, op, %rj	soustraction %rj <- %ri + op (avec flags)
umulcc %ri, op, %rj	multiplcation %rj <- %ri * op
and %ri, op, %rj	and %rj <- %ri and op
andcc %ri, op, %rj	and %rj <- %ri and op (avec flags)
or %ri, op, %rj	or %rj <- %ri and op
orcc %ri, op, %rj	or %rj <- %ri and op (avec flags)
xor %ri, op, %rj	xor %rj <- %ri and op
xorcc %ri, op, %rj	xor %rj <- %ri and op (avec flags)
sll %ri, op, %rj	décalage à gauche %rj <- %ri << %rj
slr %ri, op, %rj	décalage à droite %rj <- %ri >> %rj
sethi val22, %ri	%ri[21..0] <- val22

Figure III-105. ***Les groupes d'instructions du processeur CRAPS.***

Ce tableau est remarquablement réduit ; SPARC/CRAPS mérite véritablement son qualificatif de processeur RISC. Les versions suivantes du SPARC, notamment l'UltraSparc II sur 64 bits, vont ajouter à cet ensemble beaucoup de formats d'instructions supplémentaires, et perdre de cette élégante simplicité.

Langage assembleur

Dans ces tableaux Figure III-105, les instructions ont été écrites, non pas en langage machine qui aurait été peu compréhensible, mais dans une écriture symbolique équivalente dite ***écriture assembleur***. Pour chaque processeur, le constructeur fournit le descriptif complet du jeu d'instruction en langage machine, mais aussi une écriture assembleur conseillée équivalente. Les développeurs peuvent alors écrire des ***programmes assembleurs***, dits encore ***assembleurs***, dont le rôle est de traduire un texte source écrit en langage assembleur, en la suite des codes machine correspondants. Le terme ***assembleur*** désigne donc à la fois le langage du processeur (sous sa forme symbolique) et le programme qui traduit un texte source du langage assembleur vers le langage machine. La Figure III-106 montre une instruction écrite en assembleur, et son code machine équivalent.

```
addcc %r2, 10, %r3 ⇔ 10.00011.010000.00010.1.0000000001010
```

Figure III-106. Une instruction en assembleur CRAPS/SPARC et le code machine correspondant.

Instructions synthétiques

Certaines instructions semblent manquer. Par exemple, il n'y a pas d'instruction de copie d'un registre vers un autre, ou d'instruction qui compare 2 registres, ou encore d'instruction qui mette à zéro un registre. En fait, il n'est pas nécessaire que ces instructions existent, car elles sont des cas particuliers d'instructions plus générales. Par exemple la mise à 0 d'un registre %ri peut se faire par l'instruction orcc %r0, %r0, %ri. On exploite ici le fait que %r0 vaut toujours 0 ; le OR de 0 avec %r0 (qui vaut 0) est ainsi stocké dans %ri, ce qui est le résultat souhaité.

Dans la syntaxe du processeur CRAPS/SPARC figurent ainsi des instructions dites ***synthétiques***, qui sont en fait remplacées au moment de l'assemblage par la ou les

instructions équivalentes. La Figure III-107 montre l'ensemble des instructions synthétiques disponibles pour CRAPS.

Instruction	*Effet*	*implémentation*
`clr %ri`	met à zéro %ri	`orcc %r0, %r0, %ri`
`mov %ri,%rj`	copie %ri dans %rj	`orcc %ri, %r0, %rj`
`inccc %ri`	incrémente %ri	`addcc %ri, 1, %ri`
`deccc %ri`	décrémente %ri	`subcc %ri, 1, %ri`
`set val31..0, %ri`	copie *val* dans %ri	`sethi val31..8, %ri` `orcc %ri, val7..0, %ri`
`setq val12..0, %ri`	copie *val* dans %ri	`orcc %ri, val12..0, %ri`
`cmp %ri, %rj`	compare %ri et %rj	`subcc %ri, %rj, %r0`
`tst %ri`	teste nullité et signe de %ri	`orcc %ri, %r0, %r0`
`negcc %ri`	Calcule l'opposé de %ri	`subcc %r0, %ri, %ri`
`nop`	no operation	`sethi 0,%r0`
`call <label>`	Appel de sous-programme terminal	`or %r0, %r30, %r28` `ba <label>`

`rcall <label>`	Appel de sous-programme avec sauvegarde de l'adresse de retour	`push %r28`
		`call <label>`
		`pop %r28`

`ret`	retour de sous-programme	`add %r28, 1, %r30`
`push %ri`	empile %ri	`sub %r29, 1, %r29` `st %ri, [%r29]`
`pop %ri`	dépile %ri	`ld [%r29], %ri` `add %r29, 1, %r29`

Figure III-107. Les instructions synthétiques de CRAPS. Elles sont traduites en une écriture équivalente.

Initialisation d'un registre par une constante

L'instruction synthétique `set val,%ri` sert à initialiser un registre %ri avec une constante dite ***immédiate***, c'est à dire fournie directement dans le programme. Cette instruction synthétique est remplacée par deux instructions `sethi` $val_{31..8}$`,%ri` et `orcc` `%ri,`$val_{7..0}$`,%ri` qui vont effectivement bien faire cette affectation, en positionnant d'abord les 24 bits de poids fort avec sethi, puis les 8 bits de poids faibles avec `orcc`. Le fait d'avoir à exécuter deux instructions pour initialiser un registre à une valeur immédiate est typique des processeurs RISC. Ceux-ci ayant des instructions de la taille de leur mot mémoire, ils ne peuvent pas dans une seule instruction fournir une donnée immédiate de cette taille. On notera la présence de l'instruction synthétique `setq` (rajoutée par rapport au SPARC) qui permet de faire une telle initialisation en une seule instruction lorsque la constante est petite (13 bits signée, c'est à dire entre -4096 et +4095). Elle est équivalente à `orcc %r0,val,%ri` qui tire encore parti du fait que %r0 vaut 0.

On notera également la très classique instruction `mov %r1,%rj` qui en réalité est : `orcc %ri,%r0,%rj`, réalisant bien la copie du contenu de %ri dans %rj.
Nous pouvons maintenant mettre un certain nombre d'instructions CRAPS en face des lignes de notre petit algorithme :

```
clr     %r1              // %r1 <- 0 ;
set     TAB, %r4         // %r4 <- adresse de TAB ;
pour %r2 de 0 à 9 faire
  ld [%r4+%r2], %r3      // %r3 <- mem[%r4];
  addcc %r1, %r3, %r1    // %r1 <- %r1 + %r3 ;
fin pour
```

Dans ce programme, ne manquent que les instructions qui vont permettre d'implémenter la structure de contrôle `pour … fin pour`. Toutes les autres instructions sont de nature séquentielle, c'est à dire qu'elles s'exécutent dans l'ordre où elles sont présentes en mémoire, disposées à des adresses croissantes.

V.2.5. Ruptures de séquence en CRAPS

Bien sûr, si un processeur n'était doté que d'instructions séquentielles, il serait incapable d'exécuter la moindre tâche un peu complexe, puisqu'incapable d'implémenter des boucles ou des structures de contrôle de type 'tant que'. Il faut donc absolument qu'il soit doté d'instructions dites de ***rupture de séquence***, qui permettent, non seulement de 'sauter' à une instruction qui n'est pas la suivante, mais aussi de faire ce saut de façon conditionnelle, en fonction des résultats des opérations précédentes.
Un registre spécial de CRAPS/SPARC appelé %pc (*program counter*), non directement accessible au programmeur, contient en permanence l'adresse de l'instruction en cours d'exécution. Après l'exécution d'une instruction séquentielle, %pc est systématiquement incrémenté, de sorte qu'au cycle d'exécution suivant, le contrôle est passé à l'instruction suivante en mémoire.

Branchements inconditionnels

Pour effectuer un saut inconditionnel tel que `ba addr`, il suffit que cette instruction n'effectue pas l'incrémentation de %pc, mais au contraire 'force' la valeur de l'adresse `addr` dans `%pc`. Ainsi au cycle d'exécution suivant, le processeur chargera l'instruction située à cette nouvelle adresse, et le contrôle sera dont transféré à ce point dans le programme.

Branchements conditionnels

Ce saut peut également être fait de façon conditionnelle avec les instructions `bcc` telles que `bne, beq, bpos`, etc. A chacune de ces instructions est associé un test booléen ; si ce test est vérifié, le saut est effectué (%pc est alors affecté avec l'adresse de saut) ; sinon %pc est incrémenté, c'est à dire que le saut n'est pas fait et que le contrôle continue en séquence dans le programme, à l'instruction qui suit ce branchement.
Le test en question est fait sur la valeur de 4 bits N, Z, V, C appelés ***indicateurs*** ou ***flags***, qui sont positionnés après certaines instructions arithmétiques et logiques. Ils ont les significations suivantes :

- N indique que le résultat de l'opération est négatif, c'est à dire que son poids fort est à 1.
- Z indique que le résultat de l'opération est nul.
- V indique qu'il y a eu débordement d'une addition ou d'une soustraction signée.
- C indique qu'il y a eu une retenue lors d'une addition ou emprunt lors d'une soustraction.

Les instructions qui affectent les flags ont un nom qui se termine généralement par 'cc' : `addcc, xorcc`, etc. Pour être sûr des flags qui sont affectés ou non par une instruction, il faut se reporter à sa description détaillée, présente à la fin de l'ouvrage.
Les instructions de branchement conditionnel `bcc` effectuent le saut pour certaines combinaisons de valeurs des flags. Les tableaux des 3 figures suivantes les décrivent, le premier tableau présentant les branchements conditionnels les plus simples qui ne font intervenir qu'un seul flag, le second présentant les branchements associés à une arithmétique signée et le troisième les branchements associés à une arithmétique non signée.

Instruction	*Opération*	*Test*
ba	Branch always	1
beq (synonymes : be, bz)	Branch on equal	Z
bne (synonyme : bnz)	Branch on Not Equal	**not** Z
bneg (synonyme : bn)	Branch on Negative	N
bpos (synonyme : bnn)	Branch on Positive	**not** N
bcs (synonyme : blu)	Branch on Carry Set	C
bcc (synonyme : bgeu)	Branch on Carry Clear	**not** C
bvs	Branch on Overflow Set	V
bvc	Branch on Overflow Clear	**not** V

Figure III-108. *Branchements conditionnels associés à un seul flag.*

Instruction	*Opération*	*Test*
bg (synonyme : bgt)	Branch on equal	**not** (Z **or** (N **xor** V))
bge	Branch on Greater or Equal	**not (N xor V)**
bl (synonyme : blt)	Branch on Less	(N **xor** V)
ble	Branch on Less or Equal	**Z or (N xor V)**

Figure III-109. *Branchements conditionnels associés à une arithmétique signée.*

Instruction	*Opération*	*Test*
bgu	Branch on Greater Unsigned	**not (Z or C)**
bgeu (synonyme : bcc)	Branch on greater than, or equal, unsigned	**not C**
blu (synonyme : bcs)	Branch on less than, unsigned	C
bleu	Branch on Less or Equal Unsigned	**Z or C**

Figure III-110. Branchements conditionnels associés à une arithmétique non signée.

Les branchements du premier tableau qui ne font intervenir qu'un seul flag peuvent être utilisés dans la plupart des situations simples, et on peut en les combinant n'utiliser qu'eux dans des situations plus complexes.

Les branchements associés à l'arithmétique signée et non signée des deux tableaux suivants ne se comprennent que si on suppose qu'une comparaison entre deux valeurs vient d'être faite, c'est à dire dans un programme de la forme :

```
cmp %ri, reg/val
bxx saut
```

bxx représente une des instructions de branchement conditionnel, et reg/val représente un registre ou une constante. L'instruction synthétique `cmp %ri,op` est équivalente à `subcc %ri,op,%r0` c'est à dire qu'elle effectue la soustraction %ri – op (op pouvant être un registre ou une constante signée sur 13 bits) dans le but d'obtenir, non pas la valeur du résultat en tant que telle (qui est éliminée dans %r0), mais plutôt la valeur des flags après cette soustraction. Ceux-ci permettent en effet de tout savoir sur les positions respectives de %ri et de reg/val :

- $\%ri = op \Leftrightarrow \%ri - op = 0$, soit Z.
- $\%ri > op \Leftrightarrow \%ri - op > 0$, soit N = 0 et Z = 0, soit $\overline{N + Z}$.
- $\%ri \geq op \Leftrightarrow \%ri - op \geq 0$, soit $\overline{N}$.
- $\%ri < op \Leftrightarrow \%ri - op < 0$, soit N.
- $\%ri \leq op \Leftrightarrow \%ri - op \leq 0$, soit N = 1 ou Z = 1, soit Z + N.

En arithmétique signée, on doit remplacer N par (N + V) dans ces équations pour tenir compte des situations où on compare des grands nombres en valeur absolue, et où la soustraction provoque un débordement mais qui permet tout de même de conclure. Cela conduit alors aux équations du tableau Figure III-109.
En arithmétique non signée, on doit remplacer N par C, car C va avoir la même valeur que N lors de la soustraction de petites valeurs, mais va également permettre de conclure dans des situations où la soustraction a une grande amplitude et change le signe du résultat, telles que la comparaison de 0x00000001 et 0xFFFFFFFF : leur soustraction donne 0x00000002, avec N=0 et C=1. Seul C indique dans tous les cas qu'il a fallu emprunter pour soustraire les deux arguments. Cela conduit aux équations du tableau Figure III-110.
Maintenant, nous sommes prêts à terminer la programmation de notre algorithme :

```
///////   somme des éléments d'un tableau   //////
       clr    %r1             // %r1 <- 0 ;
       set    TAB, %r4        // %r4 <- adresse de TAB ;
       clr    %r2             // index i
loop:  ld     [%r4+%r2], %r3  // %r3 <- TAB[i];
       addcc  %r1, %r3, %r1   // %r1 <- %r1 + %r3 ;
       inccc  %r2
       cmp    %r2, 9
       bleu   loop           // fin pour
```

%r2 est initialisé à 0 au début de la boucle pour … fin pour. En fin de boucle, %r2 est incrémenté (`incc %r2`), puis comparé à 9. En toute rigueur il faut utiliser `bleu` puisqu'on fait un test ≤ non signé, mais `bpos` aurait fonctionné également car les valeurs des indices restent petites.

Branchements relatifs

Pour tous les branchements que nous venons de voir, `ba` et `bcc`, ce n'est pas l'adresse absolue du saut qui est codée, mais un déplacement ***relatif*** à cette adresse. Si on reprend le codage binaire de ces instructions (Figure III-103), on voit que le branchement est codé dans le champ *disp22* qui code ***un déplacement signé, en nombre de mots***. Le déplacement est signé, ce qui signifie qu'une valeur négative conduira à un saut en arrière dans le programme,

et un déplacement positif à un saut en avant. Ainsi, puisque l'instruction en cours d'exécution a son adresse stockée dans le registre %pc, le saut conduira à faire l'affectation :

%pc ← %pc + disp22

Par exemple, l'instruction de notre programme `bne loop` décrit un saut (conditionnel) de 4 instructions en arrière, soit :

```
00.0.1001.010.11111111111111111100          bne loop
```

Coder le saut sous forme d'un déplacement présente deux avantages :

- c'est plus économique en mémoire, puisque tout le saut est codé en une instruction d'un mot.
- l'instruction obtenue est ***translatable***, c'est à dire qu'un programme qui ne comprend que de telles instructions peut être placé n'importe où en mémoire. C'est une caractéristique importante des programmes lorsqu'ils doivent être chargés par le système d'exploitation à une place libre en mémoire.

Le saut effectué par ces instructions ne peut pas être fait d'une extrémité de la mémoire à l'autre. On peut voir sur le format binaire des instructions Figure III-103 que le déplacement est codé sur 22 bits, et donc on ne peut pas faire un saut de plus de 2^{21} instructions en avant ou en arrière. En pratique c'est une valeur très suffisante tant que le saut est fait à l'intérieur d'un même bloc de programme.

Branchements absolus

Dans certaines situations, on peut avoir le besoin de se brancher à une adresse absolue, et non une adresse locale et relative au bloc de code courant. Le système d'exploitation a ce besoin, lorsque son scheduler de tâches va enlever le contrôle à une tâche pour le donner à une autre par exemple, ou lorsqu'il va exécuter un sous-programme d'interruption. Nous en verrons des exemples plus loin dans ce chapitre.
Dans ce cas, on doit manipuler directement le compteur ordinal, c'est-à-dire le registre %r31, qui est directement accessible au programmeur.

V.2.6. Les modes d'adressage de CRAPS

Un ***mode d'adressage*** est un type d'accès aux données pour le processeur. Pour CRAPS, ils sont au nombre de 5 :

- *l'adressage registre :* la donnée est dans un registre, spécifié par son numéro. Par exemple :

```
addcc %r1, %r2, %r3
```

Ici les 3 données manipulées sont dans des registres.

- *l'adressage immédiat :* l'instruction elle-même, dans son codage en langage machine, contient la donnée. Par exemple :

```
addcc %r1, 2, %r3
```

Ici la deuxième donnée (2) est codée dans le champ *simm13* de l'instruction.

- *l'adressage direct* (ou absolu) : l'instruction contient l'adresse de la donnée. Par exemple :

```
ld [600], %r1
```

%r1 est chargé avec la donnée située en mémoire à l'adresse 600. Cette adresse 600 est codée dans l'instruction elle-même (un peu comme dans un adressage immédiat), dans son champ *simm13*. Comme cette valeur ne peut pas dépasser 4095, ce mode d'adressage est d'un usage très limité dans CRAPS.

- *l'adressage indirect :* l'instruction contient l'adresse de l'adresse de la donnée. Par exemple :

```
ld [%r1], %r2
```

%r2 est chargé avec la donnée située à l'adresse contenue dans %r1. « L'adresse de l'adresse » est ici le numéro du registre (1) qui contient l'adresse de la donnée. Ce mode d'adressage est indispensable pour accéder aux éléments d'un tableau de taille non connue a priori. C'est par exemple celui qu'on a utilisé dans notre programme de calcul de la somme des éléments d'un tableau, pour accéder aux nombres en mémoire.

- *l'adressage indirect avec déplacement*. Par exemple :

```
ld [%r1-2], %r2
```

Ce cas est similaire au précédent, mais on ajoute au contenu du registre une constante (ici -2) pour obtenir l'adresse de la donnée.

V.3. Directives d'un programme assembleur

Si on cherche à assembler effectivement, à l'aide de CRAPSEMU par exemple, le programme de somme des éléments d'un tableau étudié à la section précédente, le symbole TAB sera déclaré comme indéterminé. Il est en effet censé représenter l'adresse de début du tableau, mais rien n'a été spécifié pour l'initialiser. Une ***directive*** dans un programme assembleur est une ligne qui ne correspond pas à une instruction, mais qui spécifie :

- la localisation en mémoire d'un bloc d'instructions ou de données.
- la réservation et/ou l'initialisation d'un bloc de données.

Pour notre tableau TAB, nous avons besoin de spécifier à la fois son adresse, et son contenu initial, si on suppose qu'il est donné et non qu'il est le résultat d'un calcul antérieur. Le programme suivant contient les directives nécessaires :

```
;;;;;;;;;; zone de programme ;;;;;;;;;
      .org   0
      clr    %r1             // %r1 <- 0 ;
      set    TAB, %r4        // %r4 <- adresse de TAB ;
      clr    %r2             // pour i de 0 à 9
loop: ld     [%r4+%r2], %r3      // %r3 <- TAB[i];
      addcc  %r1, %r3, %r1   // %r1 <- %r1 + %r3 ;
      inccc  %r2
      cmp    %r2, 9
      bleu   loop            // fin pour
      ba *
;;;;;;;;;; zone de données ;;;;;;;;;
      .org   0x100
TAB   .word  12, -4, 5, 8, 3, 10, 4, -1, 1, 9
```

ADRESSE D'ASSEMBLAGE

La directive .org change l'adresse d'assemblage courante. Avec .org 0, le programme sera assemblé à partir de l'adresse 0. Le tableau TAB sera situé plus loin en mémoire, à partir de l'adresse 0x100, grâce à la directive .org 0x100.

ALLOCATION ET INITIALISATION DE MOTS EN MEMOIRE

La directive .word fournit une liste de nombres, écrits en décimal, hexadécimal ou binaire, qui seront placés en mémoire à partir de l'adresse d'assemblage courante. Dans notre programme exemple, cela nous permet de créer un tableau et de l'initialiser statiquement (c'est à dire avant le lancement du programme) avec 10 mots de 32 bits. Voici ce que donne le listing

d'assemblage complet, qui imprime toutes les adresses mémoire (colonne de gauche) et leur contenu :

```
00000000                    .org   0
00000000  82900000          clr    %r1
00000001  12000002          set    TAB, %r4
00000002  88912000
00000003  84900000          clr    %r2
00000004  C6010002   loop:  ld     [%r4+%r2], %r3
00000005  82804003          addcc  %r1, %r3, %r1
00000006 8480A001           inccc  %r2
00000007 80A0A009           cmp    %r2, 9
00000008 08BFFFFB           bleu   loop
00000009 10800000           ba     *
00000100                    .org   0x100
00000100 0000000C    TAB    .word 12, -4, 5, 8, 3, 10, 4, -1, 1, 9
00000101 FFFFFFFC
00000102 00000005
00000103 00000008
00000104 00000003
00000105 0000000A
00000106 00000004
00000107 FFFFFFFF
00000108 00000001
00000109 00000009
```

Par exemple à l'adresse 0x100 on trouve 0x0000000C, c'est à dire 12 ; à l'adresse 0x101 on trouve 0xFFFFFFFC, c'est à dire -4, etc.

UTILISATION DE SYMBOLES

On peut utiliser des ***symboles*** pour représenter des valeurs constantes de façon plus lisible dans un programme. On utilise pour cela une écriture du type :

```
N = 100
```

Dans cet exemple, partout où N apparaîtra, il sera remplacé par 100. Par exemple si on écrit : `set N, %r1`, tout se passe comme si on avait écrit `set 100, %r1`.
Par ailleurs, une arithmétique est utilisable sur de tels symboles. Si on voulait initialiser un tableau avec les 3 premiers multiples de 15, on écrirait :

```
N = 15
TAB  .word  N, 2*N, 3*N
```

De telles expressions arithmétiques peuvent comporter des signes '-', '+', '*', '/' et des parenthèses avec la syntaxe et la sémantique habituelle. Bien sûr, on aura compris que les calculs sur ces expressions sont effectués au moment de la phase d'assemblage et non au moment de l'exécution du programme. Si par exemple on écrit:

```
N = 15
set 2*N+4, %r1
```

, tout se passe comme si on avait écrit : `set 34, %r1`

V.4. Exemple de programme : crible d'Ératosthène

Afin d'illustrer de façon plus parlante les éléments de programmation en assembleur qui viennent d'être présentés, on va écrire un programme qui recherche tous les nombres premiers inférieurs à une certaine limite *N*, par la méthode du crible d'Eratosthène.

ALGORITHME

Cette méthode consiste à construire un tableau $TAB[i]$, $0 \leq i \leq n$ dans lequel $TAB[i] = 0$ signifie que i est premier, et $TAB[i] = 1$ signifie que i n'est pas premier. Pour cela on remplit d'abord tout le tableau de zéros. Ensuite, à partir de $i = 2$, on met à 1 $TAB[2 * i]$, $TAB[3* i]$, …jusqu'à la fin du tableau, puisque $2 * i$, $3 * i$, …ne sont pas premiers. On cherche ensuite, après 2, le premier indice i du tableau pour lequel $TAB[i] = 0$. En l'occurrence c'est 3, qui est nécessairement premier car sinon il aurait été coché précédemment. On met alors à 1 $TAB[2i]$, $TAB[3i]$, …jusqu'à la fin du tableau, puisque $2i$, $3i$, …ne sont pas premiers.
On recommence ensuite avec la prochaine valeur de i pour laquelle $TAB[i] = 0$, qui est nécessairement première, etc. On peut arrêter ce processus dès que i atteint $\sqrt{N}$, car on peut facilement montrer qu'alors tous les nombres non premiers ont déjà nécessairement été marqués à 1 dans le tableau.
On peut traduire cette méthode par l'algorithme suivant :

```
pour i de 0 à N-1
   TAB[i] <- 0 ;
fin pour
pour i de 2 à racine(N)
   si TAB[i] = 0 alors
      j <- 2*i ;
      tant que (j < N)
         TAB[j] <- 1 ;
         j <- j + i ;
      fin tq
   fin si
fin pour
```

PROGRAMMATION

D'après l'algorithme, les éléments du tableau TAB sont seulement les valeurs 0 ou 1. Un tableau de bits aurait fourni le codage le plus compact, mais aurait été difficile à manipuler. Puisque CRAPS n'a que des accès mémoire de type mot, on va se contenter ici d'un tableau de mots de 32 bits pour stocker ces valeurs ; ce tableau occupera donc N*4 octets.
Commençons d'abord par la mise à zéro des éléments du tableau :

```
N       =       300
TAB     =       0x150
        set     TAB, %r1          // %r1 = pointeur en début de tableau
        set     N, %r5            // %r5 = N durant tout le programme
        clr     %r2               // pour i de 0 à N-1
loop:   st      %r0, [%r1+%r2]    // TAB[i] <- 0
        inccc   %r2               // élément suivant
        cmp     %r2, %r5
        blu     loop ; fin pour
```

On cherche donc les nombres premiers jusqu'à 300, on suppose ici qu'il y a une zone de mémoire vive libre à l'adresse 0x150, où on va stocker les éléments du tableau TAB.
On a implémenté strictement l'algorithme, avec %r3 comme indice dans le tableau et %r1 comme pointeur sur le début du tableau. Pour écrire dans le tableau, on ne peut employer que

l'instruction `st` d'écriture en mémoire. La forme générale de cette instruction étant : `st %ri, [op+%rj]`, le registre à écrire sera nécessairement %r0 (qui vaut 0) et l'adresse du mot de 32 bits à écrire sera %r1+%r2. On fait ici le choix de garder un pointeur fixe %r1 en début de tableau, et de lui ajouter un déplacement variable %r2 pour accéder à un élément quelconque du tableau. Cela conduit à l'instruction :

```
      st   %r0,[%r1+%r2]     // TAB[i] <- 0
```

Les trois instructions suivantes sont relatives à la recherche de l'indice suivant. On commence d'abord par incrémenter l'indice :

```
      inccc %r2              // élément suivant
```

Il faut ensuite vérifier que l'indice est inférieur à la limite $N - 1$:

```
      cmp   %r3, %r5
```

Cette instruction synthétique `cmp %r3, %r5` équivaut à l'instruction `subcc %r3, %r5, %r0`. Après son exécution, les flags sont positionnés en fonction de la valeur de %*r*3 − N, c'est à dire de $i - N$. Puisque l'algorithme mentionne une relation d'inégalité, c'est la valeur du flag N qui nous intéresse. On a :

$$N = 1 \Leftrightarrow i - N < 0 \Leftrightarrow i < N \Leftrightarrow i \leq N - 1$$

On doit donc utiliser l'instruction bneg, qui effectue le branchement vers le début de la boucle si N = 1 :

```
      bneg loop     // fin pour
```

En fait, puisqu'il s'agit d'une comparaison non signée, il est même préférable d'utiliser le branchement blu (synonyme : bcs), qui fait une comparaison de type inférieur strict, non signée. Elle permettra en effet d'étendre l'algorithme à des valeurs de N très grandes, jusqu'au maximum possible sur 32 bits. On écrira :

```
      blu loop      // fin pour
```

On vient de traduire littéralement l'algorithme qui était donné, à la façon d'un compilateur, mais on peut rendre ce code plus compact et plus efficace.

Dans le reste de l'algorithme, la structure de contrôle de plus haut niveau est de la forme pour i de 2 à $\sqrt{N}$ ... fin pour. Cette fois l'indice i est utilisé de façon croissante. On va le traduire comme dans la première version de l'initialisation du tableau :

```
          setq 2,%r2      // pour i de 2 a racine(N)
loopi :    ...
          <corps de la boucle>
          ...
nexti:    inccc %r2
          cmp %r2, RN
          bleu loopi      // fin pour
```

On remarquera l'usage de l'instruction synthétique `setq` pour initialiser une petite constante, plus économique que set qui utilise deux instructions. RN est une constante égale à la racine de N ; on fait ici l'hypothèse qu'elle ne dépasse pas 13 bits afin de pouvoir employer directement `cmp %r2, RN`.

Dans le corps de la boucle, on doit ensuite lire l'élément *TAB*[*i*] du tableau, et choisir une autre valeur de i si cet élément de tableau vaut 1 :

```
          setq      2,%r2           // pour i de 2 a racine(N)
loopi:    ld        [%r1+%r2], %r6
          tst       %r6
          bne       nexti           // si TAB[i] = 0
          ...
```

L'élément de tableau est lu en mémoire, c'est donc une instruction `ld` qui doit être utilisée. Le tableau commence à l'adresse %r1 et on lit l'élément d'indice %r2 : il se trouve donc à l'adresse %r1+%r2. Il est lu dans %r6 et comme cette lecture ne modifie les flags (voir détail

de l'instruction ld en annexe), une instruction `tst %r6` est nécessaire. L'instruction `bne nexti` provoque la recherche d'un nouvel indice i si $Z = 0$, c'est à dire si %r6 est différent de 0, donc égal à 1 ici.
L'instruction j <- 2*i s'implémente très simplement sous forme d'une addition :

```
        addcc   %r2, %r2, %r3     // j <- 2*i
```

On a ensuite une structure de contrôle `tant que (j < N)`, qu'on va traduire :

```
loopj:      cmp     %r3, %r5    // comparaison entre %r3 et N
            bpos    nexti       // tant que (j < N)
            ...
            <corps de la boucle>
            ...
            ba      loopj       // fin tq
```

Elle est analogue à l'implémentation de la structure pour … fin pour, mais le test est fait en tête de boucle. %r4 contient depuis l'initialisation la constante N.
Le corps de la boucle se traduit ensuite directement :

```
check:      setq     1, %r4
            st       %r4, [%r1+%r3]     // TAB[j] <- 1
            addcc    %r3, %r2, %r3      // j <- j + i
```

Finalement, le programme complet est donné Figure III-111. On notera l'instruction d'arrêt sous forme de boucle infinie.

```
N         =         300               // taille tableau
RN        =         15                // sqrt(N)
TAB       =         0x100             // adresse du tableau
          set       TAB, %r1
          set       N, %r5
          clr       %r4
loop:     st        %r0, [%r1+%r4]    // TAB[i] <- 0
          inccc     %r4               // élément suivant
          cmp       %r4, %r5
          blu       loop              // fin pour
          setq      2, %r2            // pour i de 2 a rac(N)
loopi:    ld        [%r1+%r2], %r6
          tst       %r6
          bne       nexti             // si TAB[i] = 0
          addcc     %r2, %r2, %r3     // j <- 2*i
loopj:    cmp       %r3, %r5
          bpos      nexti             // tant que (j < N)
check:    setq      1, %r4
          st        %r4, [%r1+%r3]    // TAB[j] <- 1
          addcc     %r3, %r2, %r3     // j <- j + i
          ba        loopj
nexti:    inccc     %r2
          cmp       %r2, RN
          bleu      loopi             // fin pour
```

Figure III-111. Programme complet du crible d'Ératosthène en CRAPS/SPARC. Il construit un tableau d'octets tel qu'un élément d'indice i valant 0 indique que i est premier

V.5. Appels de sous-programmes terminaux

Pour le moment, nous n'avons écrit que des programmes monolithiques, utilisables une seule fois. Pourtant, il serait utile qu'un programme tel que la somme des éléments d'un tableau puisse être utilisé plusieurs fois sur différents tableaux durant l'exécution d'un programme, sans pour autant avoir à dupliquer son code.

CRAPS fournit un tel mécanisme sous la forme des deux instructions call et ret :

- call <adresse> sauvegarde l'adresse courante d'exécution dans le registre %r28, puis provoque un branchement à l'adresse spécifiée, où est situé le sous-programme qu'on souhaite appeler.
- ret provoque le branchement à l'adresse %r28+2. Si le contenu de %r28 n'a pas été modifié depuis l'appel call, l'exécution reprend à l'adresse qui ***suit*** celle de l'instruction call.

Tout se passe comme si call <adresse> était une instruction de somme de tableaux. On montre sur le programme suivant comment notre programme de somme des éléments d'un tableau a été transformé en ***sous-programme***, et invoqué sur deux tableaux différents :

```
/////////     programme principal    ////////
           set      TAB1, %r4       // arg. 1er appel
           call     tabsum          // 1er appel
           set      TAB2, %r4       // arg. 2eme appel
           call     tabsum          // 2eme appel
stop:      ba       stop            // arrêt
/////////    sous-programme    ////////
tabsum:    clr      %r1             // %r1 <- 0 ;
           clr      %r2             // pour i de 0 à 9
loop:      ld       [%r4], %r3      // %r3 <- mem[%r4,...,%r4+3];
           addcc    %r1, %r3, %r1   // %r1 <- %r1 + %r3 ;
           addcc    %r4, 4, %r4     // %r5 <- %r5 + 4 ;
           inccc    %r2
           cmp      %r2, 9
           bleu loop                // fin pour
           ret
///////      zone de données     ///////
          .org      0x1000
TAB1      .word     1, 2, 3, 4, 5, 6, 7, 8, 9, 10
TAB2      .word     2, 4, 6, 8, 10, 12, 14, 16, 18, 20
```

Ce mécanisme call … ret est appelé appel de sous-programmes ***terminaux***, c'est-à-dire de sous-programmes situés au dernier niveau d'appel. On l'aura sans doute déjà compris, il n'est pas possible avec ce mécanisme d'appeler un autre sous-programme à partir du corps d'un sous-programme appelé, puisque la première adresse de retour serait écrasée par la seconde dans %r31. Il nous manque maintenant un mécanisme qui permette de sauvegarder cette valeur ; il est présenté à la section suivante.

V.6. Utilisation de la pile

Principe

Une ***pile*** est une structure de données qui se manipule uniquement avec les deux opérations suivantes :

- PUSH *data* : sauvegarde (empile) *data* dans la pile.
- POP : enlève le sommet de la pile, et le renvoie.

On peut sauvegarder a priori autant de données qu'on le souhaite ; l'inconvénient est qu'on ne peut pas récupérer n'importe quand une donnée sauvegardée : on doit avant avoir récupéré toutes celles qui ont été empilées après elle.
Une telle pile pourrait être implémentée de plusieurs façons, à commencer par une pile matérielle à base d'un grand tableau de registres. La méthode la plus simple (mais aussi la

plus lente) est de stocker la pile sous forme de mots consécutifs dans la mémoire centrale, et d'utiliser un registre qui pointe sur la dernière valeur empilée, appelé ***pointeur de pile.***
Dans le cas de CRAPS, on utilisera le registre %r29, qu'on notera souvent %sp, et les opérations *push* et *pop* existent déjà sous forme d'instructions synthétiques :

```
push  %ri   :   sub   %r29, 1, %r29
                st    %ri, [%r29]
pop %ri     :   ld    [%r29], %ri
                add   %r29, 1, %r29
```

La Figure III-112 montre ce qu'on observe en mémoire, après quatre opérations successives d'empilement et de dépilement.

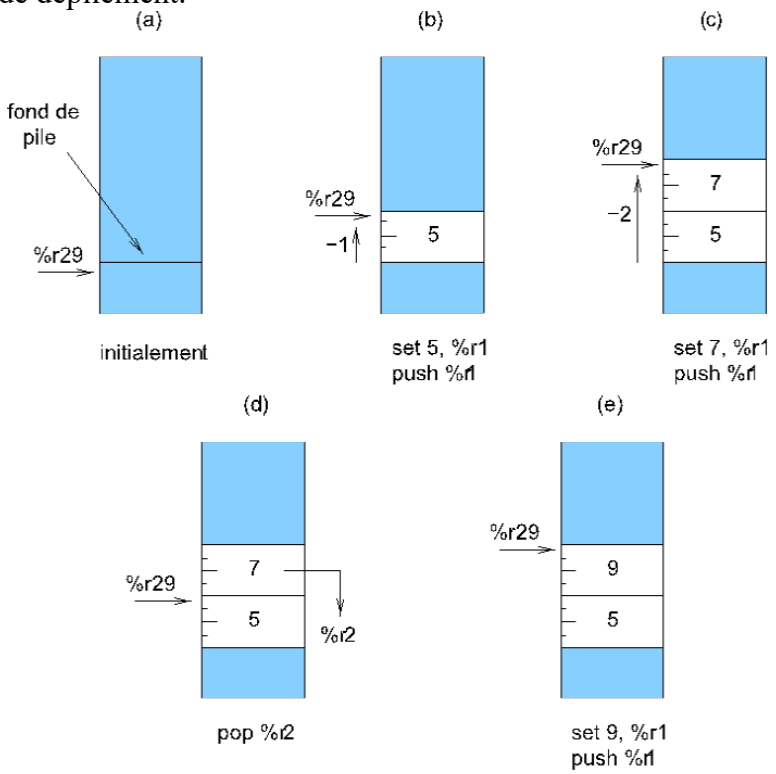


Figure III-112. *Opérations successives d'empilement et de dépilement.*

En disposant le ruban mémoire de haut en bas, on observe que les opérations d'empilement se traduisent visuellement par un empilement des valeurs les unes sur les autres : le 7 par dessus le 5, etc. Réciproquement un dépilement se traduit par la saisie de la valeur au sommet, c'est à dire celle qui est pointée par %r29. On notera cependant que la valeur dépilée n'est pas effacée de la mémoire, mais qu'elle est laissée telle quelle. Cela n'a aucune importance, car elle n'est plus accessible par des opération pop, et elle peut être écrasée par une opération push ultérieure. C'est ce qui se passe lors du push de la valeur 9 (situation (e)).
La pile de CRAPS, comme celle de la plupart des processeurs, a les deux propriétés suivantes :

- lors d'un empilement, le sommet de la pile 'remonte', c'est à dire que la valeur du pointeur de pile diminue.
- le pointeur de pile pointe toujours sur la dernière valeur empilée.

Ces indications sont utiles notamment lorsqu'on débogue un programme et qu'on souhaite savoir précisément ce que la pile contient.

Allocation de l'espace de pile

La pile étant en mémoire centrale, doit être dans une zone de RAM. Par ailleurs, il faut prévoir qu'avec les empilements successifs elle va 'grignoter' de la place en mémoire, en progressant vers les adresses basses (Figure III-113).

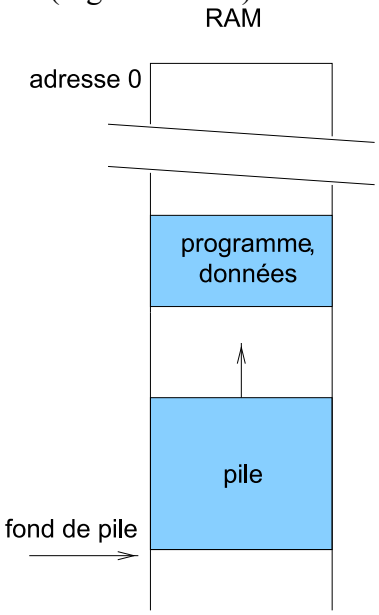


Figure III-113. La pile grandit en mémoire en progressant vers les adresses basses. Elle peut déborder sur des zones de programme ou de données.

Il y a donc un danger potentiel d'écrasement d'une zone de programmes ou de données par la pile, si celle-ci grandit trop. Lorsqu'on utilise une pile dans un programme, il faut donc initialiser le pointeur de pile en prenant en compte la position des autres données en mémoire, et la taille maximale que la pile peut prendre au cours de l'exécution des programmes.

SAUVEGARDE DE L'ADRESSE DE RETOUR DE SOUS-PROGRAMME

Un premier usage que l'on peut faire d'une pile est de résoudre notre problème de sauvegarde de l'adresse de retour lors d'un appel de sous-programme de type call … ret.
La solution est très simple :

- juste avant l'appel call, il faut empiler l'adresse de retour %r28
- juste après le retour du call, on dépile dans %r28 cette adresse

Ainsi, même si %r28 est modifié dans l'appel du sous-programme, on est sûr de retrouver la valeur de %r28 que l'on avait avant le call, c'est à dire l'adresse de retour (sous réserve que la pile soit au même niveau au retour qu'au départ).

APPELS DE FONCTIONS RECURSIVES

Cette méthode ouvre d'un seul coup la possibilité de réaliser des programmes ***récursifs*** et mutuellement récursifs, puisque les sauvegardes/restaurations des adresses de retour peuvent être faites en nombre quelconque dans la pile.
On va l'illustrer par un calcul de plus grand commun diviseur (PGCD), qui s'obtient par l'algorithme récursif suivant :

```
fonction int pgcd(int x, int y)
   si (x = y) alors
      val <- x ;
   sinon si (x > y) alors
      val <- pgcd(x-y, y) ;
   sinon
```

```
            val <- pgcd(x, y-x) ;
          return val ;
      fin fonction
```

Par exemple, PGCD(78,143) = PGCD(78,65) = PGCD(13,65) = PGCD(13,52) = PGCD(13,39) = PGCD(13,26) = PGCD(13,13) = 13.
Comme par ailleurs 78 = 2×3×13 et 143 = 11×13, on vérifie bien que 13 est leur plus grand commun diviseur.
Pour le programmer récursivement en CRAPS, il nous suffit à l'intérieur du sous-programme pgcd de rappeler par call le sous-programme pgcd, en prenant soin de sauvegarder dans la pile tout l'environnement local nécessaire au calcul, c'est à dire :

- les valeurs de registres que l'on souhaite conserver au travers de cet appel
- l'adresse de retour %r28

La Figure III-114 donne le listing complet du programme.

```
STACK     =         0x150
          .org      0
main:     set       STACK, %sp      // init pointeur pile
          set       78, %r1         // valeur de x
          set       143, %r2        // valeur de y
          call      pgcd            // appel pgcd
stop:     ba        stop            // arrêt

// calcule pgcd(x,y)
// in : x = %r1, y = %r2
// out : résultat dans %r3
pgcd:     push      %r28            // sauvegarde @ retour
          cmp       %r2, %r1
          bne       skip ; x=y ?
// x = y : return x
          mov       %r1, %r3        // val <- x
          ba retour                 // retour
skip:     bneg sup                  // x>y ?
// x < y : préparation de pgcd(x, y-x)
          subcc     %r2, %r1, %r2   // y <- y - x
          call      pgcd            // appel récursif
          ba        retour
sup: // x > y : préparation de pgcd(x-y, y)
          subcc     %r1, %r2, %r1   // x <- x - y
          call      pgcd            // appel récursif
          pop       %r28            // restaur. @ retour
retour:   ret
```

Figure III-114. Programme récursif de calcul du PGCD. Le programme est très lisible, au prix d'une exécution plus lente.

Le programme est la traduction littérale de l'algorithme récursif. Si le calcul nécessite *n* appels récursifs, la pile va grossir progressivement jusqu'à occuper *n* mots en mémoire, adresses de retour sauvegardées à chaque appel. Ensuite, *n* exécutions de l'instruction `ret` vont remettre la pile à son niveau initial.
L'utilisation de la récursivité dans le cas de ce programme est maladroite, car elle est assez inefficace à l'exécution, vu le grand nombre d'accès mémoire effectués. L'algorithme peut très simplement être traduit de façon itérative :

```
      fonction int pgcd(int x, int y)
```

```
        tant que (x <> y) faire
           si (x > y) alors
              x <- x - y ;
           sinon
              y <- y -x ;
           fin si
        fin tq ;
        return x ;
     fin fonction
```

Cet algorithme se traduit directement par le programme Figure III-115.

```
RAM         =         0x100
            .org      0
main:       set       78, %r1          // valeur de x
            set       143, %r2         // valeur de y
            call      pgcd             // appel pgcd
stop:       ba        stop             // arrêt

// calcule pgcd(x,y)
// in : x = %r1, y = %r2
// out : résultat dans %r3
pgcd:       cmp       %r2, %r1         // tant que x <> y
            bne       skip             // x=y ?
// x = y : return x
            mov       %r1, %r3         // val <- x
            ret       retour
skip:       bneg      sup              // x>y ?
// x < y
            subcc     %r2, %r1, %r2    // y <- y - x
            ba        pgcd
sup: // x > y
            subcc     %r1, %r2, %r1    // x <- x - y
            ba        pgcd
```

Figure III-115. Programme itératif de calcul du PGCD. Il est plus efficace que la version récursive car il n'y a plus d'accès en mémoire.

La leçon à tirer de cet exemple est qu'il ne faut pas abuser de la récursivité dans les fonctions de bas niveau, si un algorithme itératif est disponible. Elle est en effet gourmande en temps et en espace occupé en mémoire. La récursivité est par contre indispensable dans les fonctions de haut niveau, pour des raisons de clarté et de modularité dont la discussion sort du cadre de cet ouvrage.
Enfin on peut noter que lorsqu'on paramètre la compilation d'un programme pour qu'il produise un code plus efficace, il cherche entre autres techniques à dérécursiver les récursions terminales comme celles de notre fonction PGCD, afin de les rendre itératives.

V.7. Langages à structure de blocs et 'stack-frames'

On s'intéresse dans cette section aux problèmes que posent aux compilateurs les langages à structure de blocs tels que C, Java, etc. lors de la phase de ***génération de code***.
Un compilateur est en effet essentiellement un traducteur du langage source vers un langage cible qui est généralement le langage assembleur du processeur visé. On cherche dans cette section quelles méthodes ces compilateurs mettent en oeuvre de façon systématique pour implémenter la notion de bloc, ainsi que lors des appels de fonctions.

Implémentation des blocs sous forme de stack-frames

Les programmes écrits dans des langages tels que C et Java sont organisés en ***blocs***, encadrés par des symboles tels que {... } ou begin … end. À l'intérieur de chaque bloc on peut déclarer de nouvelles variables, dont la portée est limitée au bloc.
Généralement un bloc est associé à une structure de contrôle (boucle for ou while) ou au corps d'une fonction. On peut néanmoins créer un bloc à tout moment dans un programme, et y déclarer les variables locales de son choix : leur portée sera limitée à ce bloc. Ainsi en langage C, on peut écrire le petit programme (inutile) suivant:

```
int main(int argn,char** args) {
   int x = 1 ;
   int y = 10 ;
   printf("x=%d\n", x) ;
   {
      int x = 2 ;
      int z = x + y ;
      printf("x=%d\n", x) ;
   }
   printf("x=%d\n", x) ;
}
```

À l'exécution, il va produire l'affichage :

```
x=1
x=2
x=1
```

Il y a deux blocs dans ce programme : celui associé au corps de la procédure main, et un bloc interne situé au beau milieu, et qui n'est là que pour illustrer notre propos. La portée de la variable x située dans le bloc interne est limitée à ce bloc, et elle a temporairement pris la place de la variable x du bloc plus externe. La variable z n'est visible que dans le bloc interne, et la variable y est visible dans les deux blocs.
La pile est utilisée pour mettre en place cette visibilité temporaire des variables. Lors de l'entrée dans un bloc, le compilateur crée un ***stack-frame*** dans la pile, c'est à dire une zone de mémoire dans laquelle seront placées les valeurs des variables déclarées dans le bloc. Les stack-frames associés aux deux blocs de notre programme pourraient être implantés comme sur la Figure III-116.

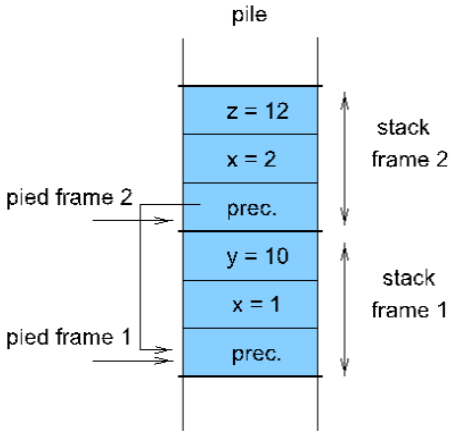


Figure III-116. Stack-frames associés aux deux blocs de code du programme. Chaque variable locale a sa place dans le stack-frame associé au bloc. On remarquera le pointeur vers le stack-frame précédent.

Un registre est généralement affecté au pointage au pied du stack-frame courant, et les accès aux différents éléments du stack-frame se font très simplement par adressage indirect avec

déplacement. Par exemple, si on utilise %r27 comme pointeur de stack-frame, on aura le mécanisme d'adressage de la Figure III-117.

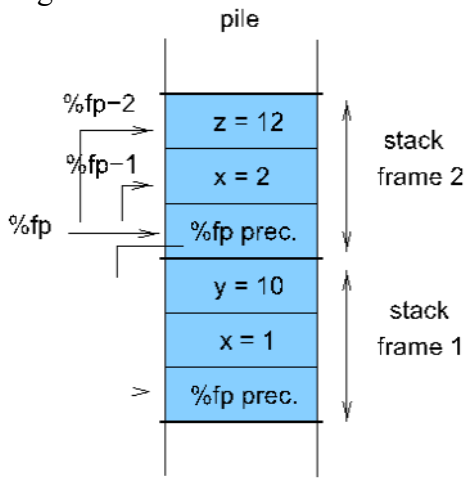


Figure III-117. Utilisation du registre %fp=%r27 comme pointeur à la base du stack-frame courant. Les accès aux variables locales se font par adressage indirect avec déplacement.

Le code assembleur traduisant ce programme pourrait être :

```
STACK       =           0x200
main:       // initialisation pointeur de pile
            set         STACK, %sp
            // création du stack-frame 1
            push        %fp             // empile val. prec.de %fp
            mov         %sp, %fp        // %fp reste fixe au pied du stack frame
            add         %sp, -2, %sp    // réserve 2 mots dans la pile
            setq        1, %r1
            st          %r1, [%fp-1]    // x <- 1
            setq        10, %r1
            st          %r1, [%fp-2]    // y <- 10
            // création du stack-frame 2
            push        %fp             // empile val. prec. de %fp
            mov         %sp, %fp        // %fp reste fixe au pied du stack frame
            add         %sp, -2, %sp    // réserve 2 mots dans la pile
            setq        2, %r1
            st          %r1, [%fp-1]    // x <- 2
            // recherche de la valeur de y par double indirection
            ld          [%fp], %r2      // %r2 <- "@stack-frame 1"
            ld          [%r2-2], %r1    // %r1 <- y
            ld          [%fp-1], %r3    // %r3 <- x
            addcc       %r1, %r3, %r4   // %r4 <- x + y
            st          %r4, [%fp-2]    // stockage de z
            // sortie stack frame 2
            mov         %fp, %sp        // sp <- fp
            pop         %fp             // dépilement dans %fp de la val prec.
            // sortie stack frame 1
            mov         %fp, %sp        // sp <- fp
            pop         %fp             // dépilement dans %fp de la val prec.
```

La création d'un stack frame consiste essentiellement à réserver la place nécessaire aux variables locales ; il s'agit d'une simple soustraction sur la valeur de %sp. À chaque création d'un stack-frame, la valeur courante de %fp=%r27 est empilée, et %fp est positionné au pied du stack-frame. L'accès à une variable locale à ce bloc se fait directement par référence à %fp

; par exemple l'accès à x dans le stack-frame 2 se fait par [%fp-1]. Plus délicat est l'accès à la variable y dans l'expression z = x + y ; en effet y n'appartient pas au bloc courant, mais au bloc de rang inférieur. La référence à la variable y est correcte, mais le compilateur doit dans ce cas accéder à y par une double indirection, en allant chercher d'abord l'adresse du stack-frame de rang inférieur, puis en faisant à partir de là un accès indirect avec déplacement. Selon les cas, plusieurs de ces indirections peuvent être nécessaires.

Appel de fonction

L'appel d'une fonction tel qu'il est typiquement implémenté par le compilateur d'un langage à structure de blocs passe par la création d'un stack-frame qui contiendra toutes les données manipulées par elle, à savoir :

- les paramètres d'appel,
- les variables locales au bloc principal de la fonction,
- la valeur que la fonction va renvoyer.

Ainsi, chaque nouvel appel aura son propre contexte de travail sous forme d'un stack-frame, et les appels successifs apparaitront sous forme d'un empilement de stack-frames qui ne mélangeront pas leurs variables locales et les paramètres d'appel, permettant notamment l'usage de la récursivité.
À la sortie d'un appel de fonction, la valeur renvoyée par la fonction est récupérée avant que l'espace du stack-frame ne soit désalloué de la pile (Figure III-118). On restaure également l'adresse de retour, (valeur de %r28 dans le cas de CRAPS) qui avait été empilée juste avant le branchement au code de la fonction.

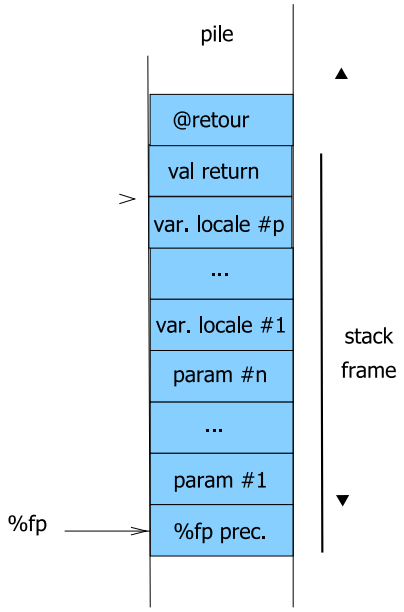


Figure III-118. Stack-frame créé lors d'un appel de fonction. On y trouve les paramètres d'appel, les variables locales, l'emplacement de la valeur à renvoyer. L'adresse de retour est empilée par-dessus juste avant le call, mais ne fait pas partie du stack-frame. Le registre %fp pointe au pied du stack frame courant ; le stack frame contient également la valeur de %fp du stack-frame précédent, ce qui permet de retrouver les informations des contextes antérieurs.

A titre d'exemple, on va écrire selon ce principe un programme de calcul de factorielle. Le stack-frame associé à chaque instance d'appel de la fonction contiendra (Figure III-119) :

- l'argument d'appel x
- une variable locale val qui joue en même temps le rôle de valeur à retourner

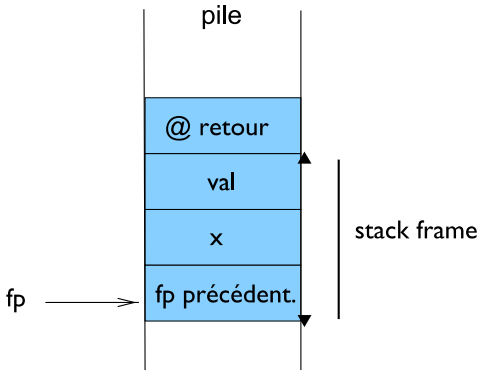


Figure III-119. Stack-frame créé lors d'un appel de la fonction fact ; on y trouve le paramètres d'appel x et une variable locale val, qui joue en même temps le rôle de valeur de return.

Le programme principal appelle fact(5) : pour cela, il créé un stack-frame dans lequel il initialise à 5 la valeur de x, dans la case mémoire d'adresse %fp-1. Juste avant l'appel par call, il empile l'adresse de retour qui se retrouve donc au sommet du stack-frame. Au retour, la valeur renvoyée peut être récupérée dans val, à l'adresse %fp-2. Une fois ce stack-frame d'appel défait, la pile se retrouve exactement dans l'état initial.
Avant chaque appel récursif de fact, un stack-frame est créé de la même manière, et on y dépose les arguments d'appel (ici x). Au retour de la fonction, la valeur retournée est récupérée dans le stack-frame, puis celui-ci est défait.

```
STACK       = 0x200
main:       // initialisation pointeur de pile
            set         STACK,      %sp
            // création du stack-frame d'appel
            push        %fp               // empile val. prec. de %fp
            mov         %sp, %fp    // %fp reste au pied du stack frame
            add         %sp, -3, %sp      // réserve 2 mots dans la pile
            setq        5, %r1
            st          %r1, [%fp-1]      // x <- 5
            // branchement à fact avec sauvegarde @retour
            push        %r28
            call        fact
            pop         %r28
            // récupération résultat dans %r2
            ld          [%fp-2], %r2
            // sortie du stack frame d'appel
            mov         %fp, %sp          // sp <- fp
            pop         %fp         // dépilement dans %fp de la val prec
            // le programme est terminé
            // %r2 contient le résultat et la pile est dans l'état initial
stop:       ba          stop

//////////  fonction fact /////////
fact:       ld          [%fp-1], %r1      // lecture de x
            cmp         %r1, 1            // x <= 1 ?
            bgu         sup
            st          %r1, [%fp-2]      // x <= 1: cas terminal, val = x
            ba          end_fact
sup:        // x > y : création du stack frame d'appel de fact(x-1)
            push        %fp               // empile val. prec. de %fp
            mov         %sp, %fp    // %fp reste au pied du stack frame
            add         %sp, -3, %sp      // réserve 2 mots dans la pile
            sub         %r1, 1, %r1
            st          %r1, [%fp-1]      // x <- x-1
            // branchement à fact avec sauvegarde @retour
            push        %r28
```

```
            call        fact
            pop         %r28
            // récupération résultat
            ld          [%fp-2], %r2
            // sortie du stack frame d'appel
            mov         %fp, %sp          // sp <- fp
            pop         %fp         // dépilement dans %fp de la val prec
            ld          [%fp-1], %r1            // lecture de x
            umulcc            %r1, %r2, %r2     // calcul de x*fact(x-1)
            st          %r2, [%fp-2]            // stockage dans val
end_fact:   // le résultat est dans [%fp-2]
            ret
```

V.8. Programmation des entrées/sorties

Jusqu'ici nous n'avons écrit que des programmes 'autistes' : ils n'ont effectué aucun échange avec l'extérieur. Les données sur lesquelles ils travaillaient leurs étaient fournies directement sous forme de zones de mémoire préremplies avec les valeurs du problème et ils produisaient des résultats sous forme de données écrites en RAM.
On va considérer que l'interface d'un ordinateur avec l'extérieur est fait au travers d'un ensemble de lignes digitales, chacune d'elle étant à un moment donné une entrée ou une sortie. Si c'est une entrée, le processeur doit être capable de lire son état 0 ou 1 ; si c'est une sortie il doit pouvoir forcer son état à 0 ou à 1. Des amplificateurs ou des atténuateurs en entrée ou en sortie peuvent permettre la lecture ou la commande de courants très forts ou très faibles ; même la commande ou l'acquisition de signaux analogiques rentre dans ce cadre, puisqu'on peut utiliser des convertisseurs numérique/analogique qui vont faire la conversion d'un monde vers l'autre.
Certaines de ces lignes sont des entrées ou des sorties sans pouvoir changer de nature : ce sera par exemple le cas des sorties des circuits appelés timer/PWM chargés de produire des signaux paramétrables de forme rectangulaire. Certaines lignes d'usage plus souple seront des lignes d'entrée/sortie générales, qui pourront être configurées individuellement et par programme en entrée ou en sortie.

Instructions spéciales d'entrées/sorties

Sur certains processeurs existent des instructions spéciales d'entrées/sorties. Le x86 par exemple possède une instruction IN AL,PORT qui lit un octet de PORT vers AL et une instruction OUT PORT,AL qui écrit un octet de AL vers PORT. PORT désigne un numéro de port, élément parmi un espace d'adressage spécial, propre aux entrées/sorties. Les écritures sur des numéros de port particuliers mettent en activité les périphériques associés tels que contrôleurs USB, contrôleurs de bus PCI Express, etc.

Entrées/sorties mappées en mémoire

Sur d'autres processeurs tels que CRAPS/SPARC, il n'existe pas de telles instructions spécialisées. Comment alors peuvent-ils faire pour interagir avec l'extérieur ? Ils utilisent en fait un mécanisme appelé entrées/sorties ***mappées en mémoire :*** c'est en lisant et en écrivant à certains emplacements mémoire particuliers, que le processeur va dialoguer avec les périphériques. L'avantage de cette méthode, c'est qu'il n'y a pas d'instructions spéciales à utiliser et que, en CRAPS par exemple, de simples ld et st suffiront. Son inconvénient, c'est de gaspiller certains emplacements de l'espace d'adressage mémoire, qui ne peuvent plus être utilisés pour de la véritable RAM ou ROM.
La machine CRAPS, telle qu'elle sera décrite en détail au chapitre suivant, et telle qu'elle est utilisable au travers du simulateur CrapsEmu et de son implantation dans un FPGA, est équipée d'un timer/PWM, de 8 lignes d'entrées reliées à 8 interrupteurs à glissière, de 8 lignes

de sorties reliées à 8 LED et de 4 afficheurs 7-segment. Ces périphériques sont donc ***mappés en mémoire*** et on va décrire maintenant la façon de les programmer.

V.8.1. Programmation des lignes d'entrées et de sorties

CRAPS est donc équipé de 8 lignes d'entrées reliées à des interrupteurs à glissière, et de 8 lignes de sorties reliées à des LED, dont un extrait du câblage est donné Figure III-120.

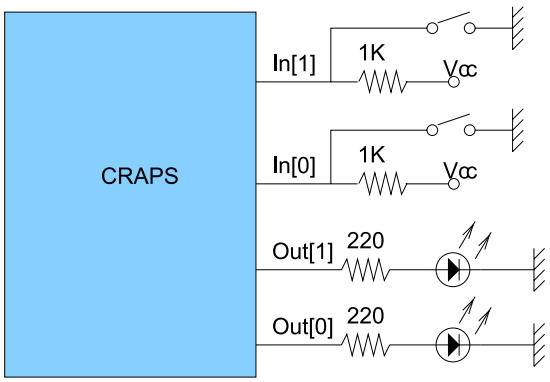


Figure III-120. Partie du câblage des lignes d'entrées et de sorties de CRAPS.

On voit que si une de lignes de sortie Out[0] ou Out[1] est à '0', la diode correspondante est éteinte ; si elle est à '1', un courant d'environ 20mA traverse la diode et l'allume. Inversement, lorsqu'un des interrupteurs est ouvert, la ligne correspondante lit '1'(car la résistance de 1K induit un courant entrant de 5mA si Vcc=5v) ; s'il est fermé elle lit '0'.

Adresses mémoire utilisées

Dans la configuration que nous allons décrire, les lignes d'entrées et de sorties de CRAPS sont ***mappées en mémoire*** selon le tableau de la Figure III-121.

Adresse	*read/write*	*Opération*
0x9000 0000	read	Lecture des 8 lignes d'entrées (switches)
0xB000 0000	write	Ecriture des 8 lignes de sorties (leds)

Figure III-121. Mapping des addresses d'entrées et de sorties de CRAPS

Les valeurs 0x900000, 0xB00000 ne dépendent pas du processeur CRAPS en lui même, mais de la façon dont il est interfacé avec ses périphériques. On détaillera complètement cet aspect de ce qu'on appelle le ***décodage des adresses*** au chapitre consacré à la conception de CRAPS.

Lecture/écriture sur les lignes

Pour lire l'état des entrées, il suffit de lire en 0x900000, et de tester dans le mot lu les bits des rangs 0 à 7 correspondants aux entrées. Pour positionner l'état des sorties, il suffit d'écrire en 0xB00000 un mot dont les bits 0 à 7 correspondent aux valeurs que l'on souhaite placer sur les lignes.

Par exemple sur le schéma de la Figure III-120, pour lire l'état de l'interrupteur situé sur la ligne In[1] et allumer la diode située sur Out[0] si cet interrupteur est ouvert (Out[0]=1) on écrira le code suivant :

```
INPUTS      =          0x90000000        // adresse de base des entrées
OUTPUTS     =          0xB0000000        // adresse de base des sorties
            set        INPUTS, %r1
```

```
            set         OUTPUTS, %r2
            ld          [%r1], %r3          // lecture des lignes d'entrée
            andcc       %r3, 0b10, %r3      // isolation du bit 1
            srl         %r3, 1, %r3         // déplacement de ce bit au rang 0
            st          %r3, [%r2]          // écriture sur les lignes de sortie
```

Exemple : programmation d'un jeu de 'pile ou face'

Le programme de la Figure III-122 exploite le câblage des entrées/sorties de la Figure III-120, et fait allumer en alternance l'une des deux LEDs situées sur IO[0] et IO[1]. Lorsque le joueur appuie sur l'interrupteur situé sur IO[0], l'état des LEDs se fige, faisant apparaître une des deux combinaisons.

```
INPUTS      =           0x90000000          // adresse de base des entrées
OUTPUTS     =           0xB0000000          // adresse de base des sorties
            set         INPUTS, %r1
            set         OUTPUTS, %r2
            setq        0b10, %r4           // état initial des LEDs
            st          %r4, [%r2]          // écriture état des LEDs
loop:       ld          [%r1], %r3          // lecture des lignes d'entrées
            andcc       %r3, 0b10, %r3      // isolation du bit 1
            bne         loop                // attend qu'il vaille 1
            // interrupteur ouvert : on inverse l'état des LEDs
            xor         %r4, 0b11, %r4      // inversion par xor
            st          %r4, [%r2]          // écriture état des LEDs
            ba          loop                // reboucle éternellement
```

Figure III-122. Programme d'un jeu de 'pile ou face'. Les LEDs #0 et #1 s'allument en alternance, et un appui sur l'interrupteur #2 fige la configuration.

Les commentaires du programme parlent d'eux-mêmes. On notera l'inversion par le xor des deux bits de poids faibles de l'état des LEDs dans %r4.

V.8.2. Programmation des afficheurs 7-segments

La périphérie de CRAPS comporte 4 afficheurs 7-segments qu'on peut allumer/éteindre globalement, et dont on peut affecter la valeur lorsqu'ils sont allumés. Ils sont capables d'afficher des nombres hexadécimaux de 0000 à FFFF.

Adresses mémoire utilisées

Ces afficheurs sont ***mappés en mémoire*** selon le tableau de la Figure III-123.

Adresse	*read/write*	*Opération*
0xA000 0000	write	Donnée à afficher (16 bits de poids faible)
0xA000 0001	write	Les 4 bits de poids faible de la donnée écrite commandent respectivement l'activation des 4 afficheurs

Figure III-123. Mapping des addresses des afficheurs 7-segments.

Par exemple, le code suivant allume les afficheurs, puis y affiche le nombre « ABCD » :

```
SSEGS   =       0xA0000000          // adresse de base afficheurs 7-segs
        set     SSEGS, %r1      // %r1 <- adresse de base
        setq    0b1111, %r2
        st      %r2, [%r1+1]      // activation des 4 afficheurs
```

```
set     0xABCD, %r2
st      %r2, [%r1]        // affichage de "ABCD"
```

V.8.3. Programmation du timer/PWM

Principe général

CRAPS possède un ***timer/PWM*** qui lui permet de générer sur une ligne de sortie dédiée, des signaux rectangulaires dont la période et le rapport cyclique sont programmables. La forme du signal est donnée Figure III-124.

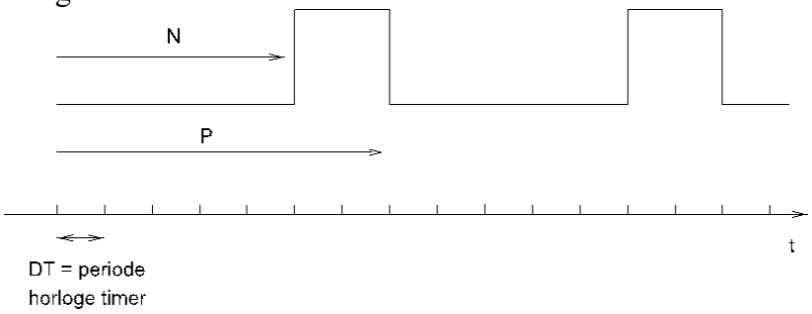


Figure III-124. *Forme et paramètres du signal de sortie du timer/PWM de CRAPS.*

Les usages d'un timer/PWM sont nombreux. Dans un système multi-tâches, la sortie d'un timer sert à déclencher une interruption qui effectue la commutation entre les différentes tâches. Il fonctionne à une fréquence d'environ 50 hertz, ce qui donne à l'utilisateur l'illusion que ses différents programmes fonctionnent en parallèle.
On s'en sert également pour contrôler avec précision l'énergie communiquée à un appareil de type lampe ou moteur à courant continu (après amplification), celle-ci étant proportionnelle au rapport cyclique $\frac{P-N}{P}$.

Registres de commande

Il s'agit d'un timer 16 bits, c'est à dire que les paramètres *P* et *N* sont codés sur 16 bits, et représentent un nombre de cycles de l'horloge du timer. Dans la version actuelle de CRAPS, l'horloge du timer est obtenue à partir de l'horloge générale, dont la fréquence a été prédivisée par 32. Avec une fréquence d'horloge de 50 MHz, la période DT de l'horloge du timer est ainsi de $\frac{1}{50\ 10^6} * 32 = 0{,}64\ \mu s$
Le timer possède en interne deux registres qui stockent les valeurs N et P nécessaires à son fonctionnement et qui sont également mappés en mémoire, aux adresses données Figure III-125.

Adresse	*read/write*	*Opération*
0xC000 0000	write	Ecriture de la valeur de P
0xC000 0001	write	Ecriture de la valeur de N

Figure III-125. Mapping des addresses associées au timer/PWM

Si par exemple on souhaite générer un signal carré de période 100 DT, on écrira le code suivant :

```
BASETIMER       =       0xC0000000          // adresse de base du timer
                set     BASETIMER, %r1      // %r1 = base timer
                setq    100, %r2            // P = 100
                st      %r2, [%r1]          // écriture de P
```

```
setq        50, %r2             // N = 50
st          %r2, [%r1+1]        // écriture de N
```

Exemple d'utilisation : commande d'un servo-moteur

Un ***servo-moteur*** est composé d'un moteur à courant continu avec un ensemble d'engrenages démultiplicateurs qui lui donnent un couple très élevé. Il est muni d'une électronique de commande qui permet de contrôler avec précision, non pas la vitesse de rotation, mais la position angulaire de son axe, qui n'a un débattement que de 180° (Figure III-126).

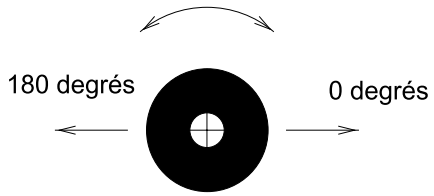


Figure III-126. Servo-moteur. La rotation de l'axe est très démultipliée et a un couple élevé. La commande ajuste la position angulaire de l'axe, qui peut varier de 0° à 180°.

La commande se fait sur un seul fil, à la norme TTL (0 ou 5v), et consiste en une impulsion périodique dont la largeur code la position angulaire de l'axe (Figure III-127).

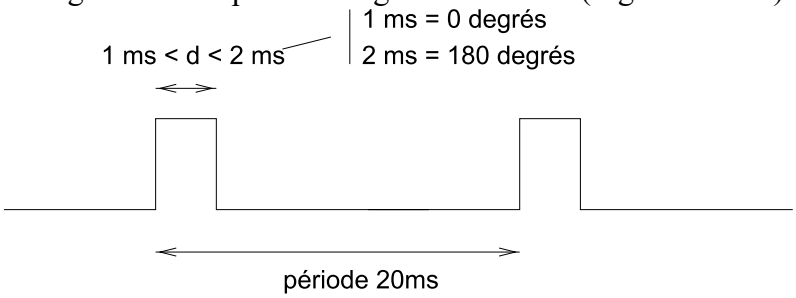


Figure III-127. Forme du signal de commande d'un servo-moteur. Il est périodique de période 20ms ; l'impulsion a une largeur comprise entre 1ms et 2ms, qui code la position angulaire de l'axe entre 0° et 180°.

Dans ce diagramme, P = 20ms et N varie entre 18ms et 19ms. On a vu que la période de base du timer est DT = 0,64 µs, ce qui correspond aux valeurs de registres P = 31250 (DT) et N variant entre 28125 (DT) et 29687 (DT).
La Figure III-128 donne un sous-programme de positionnement du servo-moteur, auquel on donne comme paramètre l'angle de positionnement dans %r1 sous forme d'une valeur entre 0 et 1562, 0 correspondant à un angle de 0° et 1562 à un angle de 180°. Le sous-programme ajoute 28125 à cette valeur et programme ensuite les registres P et N du timer.

```
BASETIMER       =           0xC0000000          // adresse de base du timer
main:           // exemple d'utilisation
                set         390, %r1
                call        servo
stop:           ba          stop

servo:          set         BASETIMER, %r2      // %r2 = base timer
                set         31250, %r3          // valeur de P
                st          %r3, [%r2]          // écriture de P
                set         28125, %r3
                add         %r1, %r3, %r1       // N = %r1 + 28125
                st          %r1, [%r2+1]        // écriture de N
                ret
```

Figure III-128. *Sous-programme de commande de servo-moteur.*

On comprend tout l'intérêt de commander un servo-moteur par un tel circuit timer/PWM : le signal périodique est généré de façon autonome et le processeur n'a besoin d'intervenir que lors d'un changement de position, qui se traduit par une simple écriture en mémoire.

V.9. Programmation des exceptions

V.9.1. Définitions

Une ***exception*** est un événement exceptionnel qui intervient au cours du fonctionnement du processeur. La prise en compte d'une exception se traduit par l'exécution d'un sous-programme qui lui est associé, appelé ***gestionnaire d'exception***. On pourrait dire qu'une exception est un appel de sous-programme inattendu.
On classe les exceptions en deux groupes, selon que leur cause est interne ou externe.

Les traps

Les ***traps*** sont des exceptions à cause interne, provoqués par l'exécution d'une instruction par le processeur. Une liste de traps courants fera comprendre leur nature :

- ***instruction illégale***. Le processeur tente d'exécuter une instruction dont le code machine ne correspond à aucune instruction connue.
- ***bus error***. Le processeur tente d'effectuer un accès mémoire à une adresse où aucune mémoire n'est implantée.
- ***division par 0***. Le processeur exécute une instruction de division et le diviseur est nul.
- ***erreur d'alignement***. Le processeur tente d'effectuer un accès mémoire à une adresse qui n'est pas un multiple de 4 si par exemple il s'agit d'un processeur 32 bits. La plupart des processeurs exigent en effet que la lecture ou l'écriture des mots respectent de telles contraintes d'alignement.
- ***instruction privilégiée***. Le processeur tente d'exécuter une instruction privilégiée (réservée aux programmes en mode superviseur) mais il n'est pas en mode superviseur.

On dit de telles exceptions qu'elles sont *synchrones*, car elles surviennent à un endroit prévisible du programme. Elles correspondent généralement à un fonctionnement incorrect de celui-ci et conduisent souvent à l'arrêt de la tâche associée.
Les traps sont ainsi l'équivalent matériel des exceptions logicielles, qui elles aussi se déclenchent en cas d'exécution incorrecte d'instructions, et qui ont un gestionnaire associé.

Les interruptions

Une interruption est une exception à cause externe, généralement liée à un événement d'entrée/sortie. Par exemple :

- la lecture de données par un disque est terminée et le contrôleur de disques envoie au processeur une interruption pour l'en informer.
- un paquet de réseau vient d'arriver.
- un événement clavier ou souris vient de se produire.
- un changement est intervenu sur une des lignes d'entrées/sorties.
- un événement est intervenu sur la chaîne USB.

Les interruptions sont dites *asynchrones*, car elles surviennent à des moments totalement imprévisibles du fonctionnement du processeur, et non en rapport avec des instructions précises. Le processeur est prévenu de l'occurrence de l'interruption par l'activation d'une ligne externe spécifique, souvent commune à plusieurs ou à toutes les interruptions. C'est seulement ensuite que le type de l'interruption sera identifié, et qu'un sous-programme

associé sera exécuté, et ce d'une façon qui devra rester transparente pour le programme principal qui était en cours d'exécution à ce moment. Par exemple lorsque vous appuyez sur une touche de votre clavier d'ordinateur, aucun programme n'attend spécifiquement et activement cette frappe. Lors de la frappe, une interruption est générée par le contrôleur de clavier vers le processeur, et celui-ci suspend le programme en cours d'exécution, et appelle un sous-programme gestionnaire de cette interruption, dont le rôle va être d'obtenir du contrôleur de clavier le code de la touche frappée et de le placer dans un buffer situé au niveau du système d'exploitation. L'exécution de ce sous-programme sera très rapide et on verra qu'elle sera faite de façon transparente pour le programme qui a été interrompu.

Une interruption = un coup de sonnette

Pour mieux faire comprendre les traps et les interruptions, on peut employer une analogie domestique. Supposons que nous soyons en train de faire un gâteau, en suivant les instructions d'une recette : ce sera l'équivalent du programme principal en cours d'exécution. La plupart des instructions de la recette sont sans problème : remuer la pâte, etc. Un petit nombre d'entre-elles peuvent mal se passer : si on doit casser un œuf mais qu'il n'est pas frais ; si on doit mettre une pincée de sel mais qu'on n'a plus de sel, etc. Ces problèmes spécifiques sont associés à des instructions spécifiques, ce sont l'équivalent des traps. Le plus souvent, leur prise en compte consistera à abandonner la tâche en cours.
Supposons maintenant que pendant que nous faisons cette recette nous attendions la visite du facteur, mais que notre sonnette soit en panne. Comme le moment précis de cette visite n'est pas défini, cela va nous obliger à interrompre sans cesse notre recette pour aller regarder par la fenêtre. Cela sera pénalisant pour la recette, mais aussi pas très efficace : le facteur peut très bien être venu et reparti en pensant qu'il n'y avait personne, si on laisse passer trop de temps entre deux coups d'œil. Avec une sonnette, qui joue le rôle d'une ligne d'interruption, on peut réaliser la recette sans penser sans cesse au facteur. Si un coup de sonnette survient, on terminera l'instruction en cours, (par exemple, il faut terminer de casser un œuf si on a déjà commencé) et on cochera sur la recette l'endroit où on s'est arrêté.
On réagira ensuite à l'interruption par une procédure adaptée : aller à la porte, ouvrir, saluer, récupérer éventuellement des données, etc. De retour de cette procédure, dont le traitement aura été court, on reprendra la recette à l'instruction qui suit celle qu'on avait cochée.
Cette analogie permet même de présenter le problème des niveaux de priorité entre exceptions. Comment doit-on faire si par exemple le téléphone sonne alors qu'on est en train de répondre au visiteur à la porte ? Si on juge que le téléphone est plus prioritaire que la porte d'entrée, alors on interrompra le dialogue en cours avec le visiteur pour effectuer la procédure associée au téléphone. Une fois qu'on aura fini de répondre, on reprendra le dialogue avec le visiteur là où on l'avait arrêté, pour finalement revenir au programme principal (la recette), dont l'exécution n'aura pas été perturbée.

Niveaux de priorité

Ainsi, à chaque exception (trap ou interruption) est associé un ***niveau de priorité***, codé généralement sous forme d'un nombre entier. Par ailleurs le processeur entretient dans un registre d'état une valeur de niveau courant d'exécution. Lorsque le processeur exécute un programme à un niveau p, il ne peut être interrompu que par une exception de niveau q supérieur à p. Lorsque cette interruption est prise en compte et que son gestionnaire est exécuté, le processeur élève son niveau d'exécution à q de façon à ne plus pouvoir être interrompu que par des exceptions de niveau supérieur à q. Lorsque le processeur sort du gestionnaire de cette exception, il reprend le niveau d'exécution qu'il avait avant l'appel. Le programme principal s'exécute au niveau 0, et il est par conséquent interruptible par toutes les exceptions.

La Figure III-129 montre un exemple de scénario d'exécution du programme principal et de deux gestionnaires d'exception de niveaux de priorité différents.

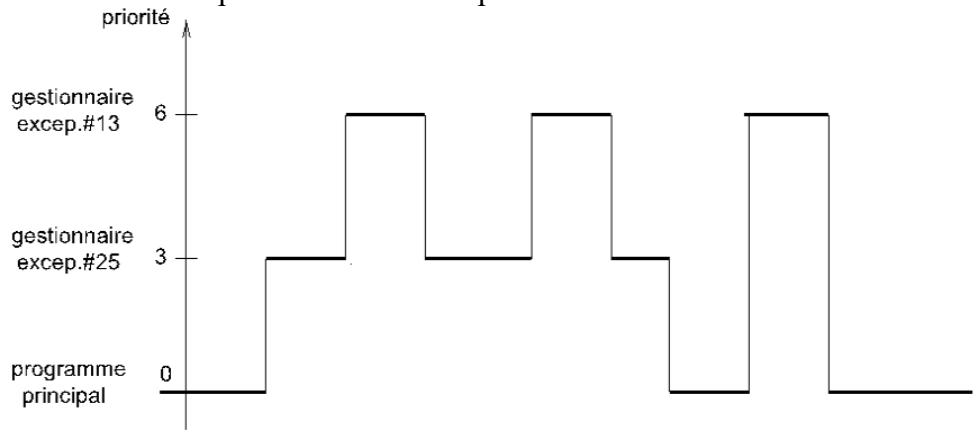


Figure III-129. Niveaux de priorités et gestionnaires d'exceptions. Le programme principal est interrompu par l'exception #25, qui lui même est interrompu par l'exception #13;chaque interruption est transparente pour celui qui est interrompu.

Numéro de l'exception

À chaque exception est également associé un numéro de type, unique par type d'exception, qui va servir à localiser l'adresse de son gestionnaire d'exception. Ce numéro n'a en principe pas de rapport direct avec le niveau de priorité de l'exception.

Gestionnaires d'exceptions

Lorsqu'une exception est prise en compte, un sous-programme appelé ***gestionnaire d'exception*** ou ***handler d'exception*** est appelé, associé au numéro de type *tt* de l'exception. Il existe généralement une ***table de vecteurs d'exception*** qui contient les adresses de tous les gestionnaires d'exception, classée par numéro de type *tt*. Cette table est initialisée lors du chargement du système d'exploitation, ou elle peut être en ROM. Souvent, un registre spécial du processeur pointe sur le début de cette table.

V.9.2. Les exceptions dans le processeur CRAPS

La gestion des exceptions est extrêmement simplifiée dans CRAPS. Il n'y a pas de traps (exceptions à cause interne) et il n'y a qu'une seule ligne d'interruption, qui peut être reliée à la sortie du timer ou tout autre signal. L'interruption correspond alors à un front montant de ce signal, qui est mémorisé dans une bascule visible en tant que bit 2 du registre d'état %psr = %r25. Le processeur ne possède pas plusieurs niveaux d'exécution.
Une fois l'interruption déclenchée, la procédure suivante, ininterruptible, est automatiquement exécutée par le processeur :

- l'instruction en cours d'exécution est terminée
- le bit 2 de %psr (le bit qui mémorise l'interruption) est remis à 0
- %pc est empilé, de façon à mémoriser l'adresse de retour dans le programme interrompu
- %psr est empilé, de façon à sauvegarder les flags
- %pc ← 1, ce qui va conduire au branchement dans le handler d'interruption

Le handler d'interruption est donc nécessairement à l'adresse 1. Ce doit être un sous-programme court, pour ne pas interrompre trop longtemps le programme principal. Il doit se terminer par l'instruction `reti`, qui provoque un retour dans le programme interrompu, à l'adresse suivant l'instruction qui a été interrompue.
Le handler doit prendre soin de ne pas changer les valeurs des registres, pour ne pas perturber le programme qui a été interrompu. Cela passe souvent par un empilement des valeurs des

registres que le handler manipule, puis un dépilement juste avant le `reti` pour redonner à ces registres les valeurs qu'ils avaient au moment de l'interruption. Il n'est pas nécessaire de sauvegarder les flags, qui sont automatiquement sauvegardés par la procédure de prise en compte de l'interruption.
Enfin, puisque le handler est à l'adresse 1, cela implique qu'en cas d'usage des interruptions, la première instruction du programme principal à l'adresse 0 doit être un branchement inconditionnel plus loin en mémoire vers la suite du programme, au-delà du handler d'interruption.
L'instruction `reti` fonctionne de la façon suivante :

- on dépile dans %psr (%r25) pour qu'il retrouve sa valeur initiale
- on dépile dans %pc pour que le contrôle retourne à l'adresse qui a été interrompue

Il s'agit bien d'une instruction et non d'une instruction synthétique, car elle devra s'exécuter de façon atomique et non interruptible.

Exemple : horloge temps réel

À titre d'exemple, on va mettre en place une horloge qui permettra à un programme d'obtenir une heure d'une précision parfaite, sous forme d'une fonction `getTimeMillis`.
À chaque appel de cette fonction, on obtiendra dans %r1 le nombre de millisecondes écoulées depuis la mise en route de la machine ; cette valeur s'incrémentera de façon exacte entre chaque appel, sans que le programme principal en cours d'exécution ne s'en occupe explicitement.
On va pour cela programmer le timer pour qu'il fournisse des impulsions toutes les millisecondes, et brancher sa sortie à la ligne d'interruption. Le handler de l'interruption incrémentera simplement un compteur de millisecondes, que la fonction `getTimeMillis` se contentera de renvoyer.

```
BASETIMER       =       0x400000        // adresse de base du timer
STACK           =       0x200           // adresse du sommet de pile
TIME            =       0x100           // mot mémoire contenant le temps en ms
                .org    0
                ba      main            // brancht vers le début du programme

                .org    1               // adresse du handler d'interruption = 1
handler:        push    %r1             // sauvegarde %r1 et %r2
                push    %r2
                set     TIME, %r1
                ld      [%r1], %r2      // lecture du temps
                add     %r2, 1, %r2     // incrémentation
                st      %r2, [%r1]      // écriture valeur incrémentée
                pop     %r2             // restauration %r1 et %r2
                pop     %r1
                reti

getTimeMillis:  set     TIME, %r1
                ld      [%r1], %r1
                ret

init_timer:     // configuration du timer à une période de 1ms
                set     BASETIMER, %r1  // %r1 = base timer
                setq    1562, %r2       // P = 1562 * DT = 1ms
                st      %r2, [%r1]      // écriture de P
```

```
                setq    781, %r2          // N = P/2
                st      %r2, [%r1+1]      // écriture de N
                ret

main:           set     STACK, %r30       // initialisation sommet de la pile
                call    init_timer        // initialisation timer
loop:           call    getTimeMillis
                ba      loop
```

V.10. Exercices corrigés

V.10.1. multiplication programmée

Énoncé

Écrire un sous-programme qui effectue la multiplication de deux nombres non signés de 16 bits placés dans les poids faibles de %r1 et %r2 respectivement, avec un résultat sur 32 bits dans %r3.

Solution

On suppose donc qu'on ne dispose pas de l'instruction `umulcc`. On pourrait mettre en place une boucle qu'on effectuerait %r1 fois, et dans laquelle on cumulerait %r2 dans %r3. Cette méthode serait simple, mais particulièrement inefficace pour de grandes valeurs de %r1.On va plutôt utiliser une méthode dans laquelle on prend les bits de %r2 un par un en commençant par le poids faible, et on cumule dans %r3 (initialisé à 0) la valeur de %r1, que l'on décale vers la gauche à chaque étape. Cela correspond à l'algorithme suivant :

```
; calcul du produit A x B
résultat <- 0 ;
tant que (B <> 0) faire
   b0 <- poids_faible(B) ;
   décaler B d'un bit vers la droite ;
   si b0 = 1 faire
      résultat <- résultat + A ;
   fin si
   décaler A d'un bit vers la gauche ;
fin tq
```

Le programme qui suit est une traduction littérale de l'algorithme. On notera le test `bne noadd` qui n'est pas fait immédiatement après l'instruction `andcc %r2, 1, %r0` qui positionne le flag Z ; cela est possible car l'instruction qui la suit `srl %r2, 1, %r2` ne modifie pas les flags.

```
            // programme principal de test
            set         17, %r1       // A <- 17
            set         14, %r2       // B <- 14
            call        mulu16        // calcul de A x B
stop:       ba          stop          // arrêt

mulu16:     clr         %r3           // résultat <- 0
loop:       tst         %r2           // tant que B <> 0
            be          fin
            andcc       %r2, 1, %r0   // b0 (Z) <- poids faible de B
            srl         %r2, 1, %r2   // décale B vers la droite
            bne         noadd          // si b0 = 1 faire
            addcc       %r1, %r3, %r3  // résultat <- résultat + A
```

```
noadd:      sll         %r1, 1, %r1     // décale A à gauche
            ba          loop            // fin tq
fin:        ret
```

V.10.2. Système de sécurité

Énoncé

On souhaite utiliser les lignes d'entrées/sorties de CRAPS pour réaliser un système de mise en marche de machine dangereuse. On supposera que les entrées In[0] et In[1] sont reliées à des boutons poussoirs appelés A et B respectivement, équipés d'un dispositif anti-rebond. La ligne Out[0] commandera la mise en marche de la machine. On demande d'écrire un programme qui fonctionne en permanence, et qui déclenche la mise en marche de la machine lorsque la procédure suivante est respectée :

- A et B doivent être relâchés initialement ;
- appuyer sur A,
- appuyer sur B : la machine se met en marche.

Toute autre manipulation arrête ou laisse la machine arrêtée ; il faut ensuite reprendre la procédure au point 1 pour la mettre en marche.

Solution

On a déjà réalisé une telle commande à l'aide de circuits séquentiels. On demande maintenant de le réaliser par programme, comme on pourrait le faire avec un microcontrôleur.
On rappelle le graphe d'états de ce système Figure III-130.

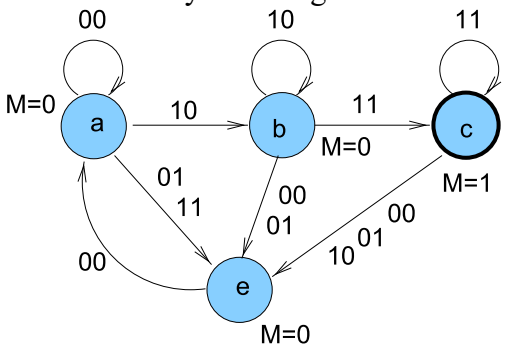


Figure III-130. *Graphe d'états du système de sécurité.*

À chaque état dans le graphe correspondra une position dans le programme. Cela conduit à l'algorithme suivant :

```
label a:
   écrire 0 sur M ;
   lire état de A et B ;
   cas (A,B)
      (0,0): aller en a ;
      (1,0): aller en b ;
      autre: aller en e ;
   fin cas
label b:
   lire état de A et B ;
   cas (A,B)
      (1,0): aller en b ;
      (1,1): aller en c ;
      autre: aller en e ;
   fin cas
```

```
label c:
   écrire 1 sur M ;
   lire état de A et B ;
   cas (A,B)
      (1,1): aller en c ;
      autre: aller en e ;
   fin cas
label e:
   écrire 0 sur M ;
   lire état de A et B ;
   cas (A,B)
      (0,0): aller en a ;
      autre: aller en e ;
   fin cas
```

Le programme qui suit est une traduction littérale de l'algorithme.

```
INPUTS      =          0x90000000        // adresse de base des entrées
OUTPUTS     =          0xB0000000        // adresse de base des sorties
            set        INPUTS, %r3
            set        OUTPUTS, %r4
a:          setq       0b0, %r2          // M <- 0
            st         %r0, [%r4]
            ld         [%r3], %r2        // lecture état de A et B
            andcc      %r2, 0b11, %r2    // isolation des bits 0 et 1
            subcc      %r2, 0b00, %r0
            be         a                 // (A,B) = (0,0) : aller en a
            subcc      %r2, 0b10, %r0    // (A,B) = (1,0) : aller en b
            bne        e                 // sinon aller en e ;
b:          setq       0b0, %r2
            st         %r2, [%r4]        // M <- 0
            ld         [%r3], %r2        // lecture état de A et B
            andcc      %r2, 0b11, %r2    // isolation des bits 0 et 1
            subcc      %r2, 0b10, %r0
            be         b                 // (A,B) = (1,0) : aller en b
            subcc      %r2, 0b11, %r0    // (A,B) = (1,1) : aller en c
            bne        e                 // sinon aller en e
c:          setq       0b1, %r2
            st         %r2, [%r4]        // M <- 1
            ld         [%r3], %r2        // lecture état de A et B
            andcc      %r2, 0b11, %r2    // isolation des bits 0 et 1
            subcc      %r2, 0b11, %r0    // (A,B) = (1,1) : aller en c
            be         c                 // sinon aller en e
e:          setq       0b0, %r2
            st         %r2, [%r4]        // M <- 0
            ld         [%r3], %r2        // lecture état de A et B
            andcc      %r2, 0b11, %r2    // isolation des bits 0 et 1
            subcc      %r2, 0b00, %r0    // (A,B) <> (0,0) : aller en e
            bne        e
            ba         a                  // sinon aller en a
```

Chapitre VI. Amélioration des performances

VI.1. Pipelines

Dans l'organisation initiale d'un processeur telle que celle de notre CRAPS, l'exécution d'une instruction se fait en une séquence de plusieurs étapes. Chaque étape ne fait intervenir qu'une partie de la micro-machine, pendant que les autres éléments restent inoccupés, ce qui laisse une marge d'efficacité inexploitée. L'idée d'une organisation en pipeline, c'est de réaliser cette succession d'étapes comme sur une chaine d'assemblage automobile, avec des postes spécialisés à chaque étape. On charge une instruction dans la première étape, puis au cycle d'horloge suivant, cette instruction passe à l'étape 2 pendant qu'une nouvelle instruction en chargée à l'étape 1, et ainsi de suite. Idéalement, si le pipeline est constitué de n étages, la première instruction sera totalement exécutée après n cycles d'horloges, puis une nouvelle instruction se terminera à chaque nouveau cycle d'horloge. Ainsi, même si le temps d'exécution d'une instruction est toujours le même, le débit de sortie des instructions est quant-à lui multiplié par n. En pratique, plusieurs facteurs empêcheront souvent le pipeline de fonctionner à plein rendement, mais l'efficacité n'en sera pas moins très augmentée.

VI.1.1. Un pipeline pour CRAPS

Comme tous les processeurs RISC, CRAPS a un jeu d'instructions qui se prête bien au découpage pour un pipeline. Par exemple, pour quasiment toutes les instructions, il y a un calcul et un seul à effectuer par l'UAL, même pour les accès mémoire pour lesquels l'adresse est le résultat d'une somme. On aura donc un étage dédié à ce calcul dans le pipeline. On peut par exemple utiliser le découpage suivant :

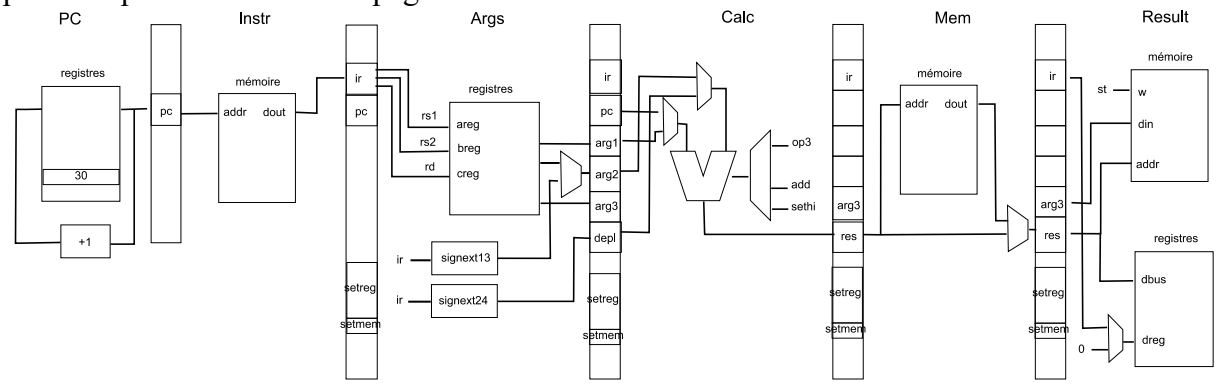


Figure III-131. Pipeline pour CRAPS.

Les composants sont séparés les uns des autres par des ***registres d'étage***, qui mémorisent les informations nécessaires aux composants de l'étage pour réaliser leur tâche spécifique. Les étages et leurs tâches associés sont :

- Etage *PC* : le registre pc est mémorisé dans le registre d'étage *PC*, puis il est incrémenté de 4. Après exécution du cycle, il est prêt à produire le pc de l'instruction suivante qui entrera dans le pipeline.
- Etage *Instr* : l'instruction située à l'adresse *PC* est lue en mémoire, et enregistrée dans le champ *ir* du registre d'étage suivant.
- Etage *Args* : une lecture parallèle des registres dont les numéros sont formés des champs *rs1, rs2, rs3* de l'instruction du registre d'étage est effectuée, en prévision des différentes situations possibles. De même, l'extraction des constantes immédiates et des déplacements

de branchement sont réalisés. Tous ces résultats sont mémorisés dans le registre d'étage suivant.

- Etage *Calc* : l'addition pour les branchements et les accès mémoire, ou l'opération associée à l'instruction de calcul, sont effectués. Le résultat est stocké dans le champ *res* du registre d'étage suivant.
- Etage *Mem* : pour une instruction ld, lecture en mémoire à l'adresse contenue dans le champ *res* du registre d'étage et stockage du résultat dans le champ *res* du registre d'étage suivant. Pour les autres instructions, le champ *res* est simplement transféré à l'étage suivant.
- Etage *Result* : le résultat est mémorisé dans un registre (pc si c'est un branchement), ou dans une case mémoire, selon le type de l'instruction.

Par exemple, une instruction telle que `ld [%r1-2], %r3` va s'exécuter de la façon suivante :

- *PC* : Mémorisation du *PC* dans le registre d'étage suivant, incrémentation de PC de 4
- *Instr* : Lecture de l'instruction en mémoire à l'adresse calculée à l'étage précédent
- *Args* : lecture simultanée de %r1, %r2 et %r3. La lecture de %r2, inutile, correspond à une extraction incongrue du champ *rs2* ; de même la lecture de %r3 sera inutile. Par contre, le champ *simm13* est étendu sur 32 bits puis mémorisé dans le champ *arg2* du registre d'étage suivant.
- *Calc* : %r1 (*arg1*) et -2 (*arg2*) sont additionnés, puis mémorisés dans le champ *res* du registre d'étage suivant.
- *Mem* : lecture en mémoire à l'adresse contenue dans *res*, puis stockage du résultat dans le champ *res* de l'étage suivant.
- *Result* : écriture de la valeur lue, située dans le champ *res* du registre d'étage, dans le registre de numéro *rd*, lu dans le champ *ir* du registre d'étage.

Chaque étage réalise une tâche simple et spécialisée. Cette simplicité permet d'abaisser les temps de cycle d'horloge, et donc d'augmenter le débit de sortie des instructions.

Aléas

Il existe trois types de situations, appelées ***aléas***, où le pipeline ne peut fonctionner à plein rendement, et où des étages restent inoccupés :

- ***Aléas structurels*** – Ils correspondent au cas où deux instructions utilisent la même ressource du processeur (registre, mémoire, UAL)
  Par exemple, il y a un conflit potentiel pour l'écriture dans le registre pc entre les étages *Pc* et *Result*, mais ce conflit ne sera effectif que s'il y a une instruction de branchement dans l'étage *Result*.
- ***Aléas de données*** – Il y a aléa de donnée lorsqu'une instruction produit un résultat, et qu'une instruction suivante a besoin de ce résultat avant que la précédente n'ait eu le temps de l'écrire en mémoire ou dans le registre destination. Par exemple, dans le morceau de programme suivant :

  ```
  add     %r1, %r2, %r3
  st      %r3, [%r4]
  ```

  Les instructions vont se suivre dans le pipeline, et lorsque le `add` sera dans l'étage *Args*, l'instruction `st` sera dans l'étage *Calc*. Mais `add` ne pourra pas continuer, car elle

a besoin de la valeur de %r3 qui sera modifiée par `st` lorsque celle-ci finira son exécution.

Autre exemple :

```
st      %r3, [%r4]
ld      [%r4],%r5
```

Ici, l'aléa de données porte sur la case mémoire dont l'adresse est dans %r4.

- ***Aléas de contrôle*** – Un aléa de contrôle se produit chaque fois qu'un branchement est effectué. Lorsqu'une instruction de branchement est chargée, il est nécessaire de connaître l'adresse de destination du branchement pour charger l'instruction suivante. Or, savoir qu'une instruction est un branchement n'est connu qu'à l'étage *Instr*, et l'adresse du branchement n'est connue qu'à l'étage *Result*. Plusieurs stratégies sont possibles pour palier ce problème.

Bulles dans le pipeline

Lorsqu'un aléa est détecté, les instructions présentes à chaque étage ne peuvent pas continuer à être exécutées telles quelles. Dans ce cas, le pipeline est partagé en deux : les étages de la partie droite du pipeline qui ne sont pas affectés par les aléas peuvent continuer à s'exécuter et à se décaler progressivement jusqu'à sortir ; les autres étages restent bloqués (aléas structurel, aléas de données) ou effacés (aléas de contrôle) tant que des aléas persistent. Les étages vacants entre les deux sont appelés des « bulles » du pipeline.
Il est important de noter que les conséquences de l'exécution d'une instruction (écriture dans un registre ou dans une case mémoire et affectation des flags) ne sont effectivement réalisées que dans le dernier étage du pipeline. Une instruction qui atteint ce stade est définitivement validée et le registre ou la case mémoire résultat sont définitivement affectés en une opération atomique ; en cas d'abandon d'une instruction qui est encore à un étage inférieur et donc partiellement exécutée, il n'y a rien besoin de défaire.

Détection des aléas

Pour détecter les aléas de données sur les registres, les registres d'étage mémorisent pour chaque instruction, à partir de l'étage de décodage *Instr*, un champ de bits '*setreg'*, qui indique pour chaque registre si l'instruction modifiera sa valeur dans l'étage *Result*. Une instruction située dans l'étage *Args* sera bloquée si elle a besoin de lire la valeur d'un registre pour lequel le bit correspondant de '*setreg'* est à 1 dans les étages plus haut *Calc*, *Mem* ou *Result*.

Un aléa de donnée sur une case mémoire est difficile à repérer finement ; on peut se contenter d'un unique bit 'setmem' dans les registres d'étage à partir de l'étage *Instr*, qui indique que l'instruction va modifier une case mémoire. Une instruction située dans l'étage *Mem* sera bloquée si le bit 'setmem' du registre d'étage *Result* est à 1. On notera qu'en principe, une instruction `st` provoque un aléa de donnée systématique, puisqu'elle est susceptible de modifier les codes des prochaines instructions qui n'ont pas encore été lus en mémoire. En pratique, le système d'exploitation, via la segmentation de la mémoire, interdit à un programme de modifier la zone de code d'un autre programme, et cet aléa n'existe donc pas.

On peut considérer que l'aléa de contrôle est une forme d'aléa de données sur le registre pc, qui peut être géré en utilisant le champ de bits 'setreg'. A partir de l'étage *Instr*, une instruction de branchement va mettre à 1 le bit de 'setreg' associé à pc, indiquant par là qu'elle est susceptible de modifier pc (même si ce n'est pas certain). Ensuite, les étages qui

n'ont pas un tel bit à 1 sur leur droite peuvent continuer à progresser dans le pipeline, y compris l'instruction de branchement elle-même.

Gestion des aléas de contrôle

On l'a vu, lorsqu'une instruction de branchement est chargée, il est nécessaire de connaître l'adresse de destination pour pouvoir continuer à charger le pipeline avec les instructions suivantes. Or cette adresse de destination ne sera connue avec certitude que quand l'instruction de branchement aura traversée le dernier étage du pipeline. Plusieurs stratégies sont alors possibles :

- Le processeur peut charger des bulles dans le pipeline jusqu'à ce que le branchement sorte du pipeline. C'est bien sûr la méthode la moins efficace
- le processeur peut faire le pari systématique que le branchement n'aura pas lieu (c'est le cas le plus fréquent, et le plus simple). Il peut alors continuer à charger dans le pipeline les instructions qui suivent le branchement, mais en se préparant à les effacer (les remplacer par des bulles) si le pari est perdu, c'est-à-dire si finalement il s'avère que le branchement est effectué, quand il est dans l'étage *Result*. Cette gestion peut être faite en utilisant le bit associé au registre pc dans le champ de bits *setreg*
- le processeur peut réaliser des prédictions de branchement. Des techniques sophistiquées existent, qui seront détaillées plus loin, et qui permettent de prédire avec un assez bon taux de réussite si un branchement sera pris ou non, à partir de l'histoire des événements récents.

Exemple d'exécution

Considérons le programme suivant :

```
          setq      10, %r1
          setq      1000, %r2
loop:          st        %r0, [%r2+%r1]
          deccc          %r1
          bne       loop
          i6
          i7
          …
```

L'entrée des instructions dans le pipeline est montrée Figure III-132. Après le chargement de `bne`, le processeur fait le pari que le branchement n'aura pas lieu et continue de charger les instructions suivantes.

cycle	PC	Instr	Args	Calc	Mem	Result
1	@setq	o	o	o	o	o
2	@setq	setq	o	o	o	o
3	@st	setq	setq	o	o	o
4	@deccc	st	setq	setq	o	o
5	@bne	deccc	st	setq	setq	o
6	@bne	deccc	st	o	setq	setq

7	@bne	deccc	st	o	o	setq
8	@bne	deccc	st	o	o	o
9	@i6	bne	deccc	st	o	o
10	@i7	i6	bne	deccc	st	o
11	@i8	i7	i6	bne	deccc	st
12	@i9	i8	i7	i6	bne	deccc
13	@i10	i9	i8	i7	i6	bne
14	o	o	o	o	o	o
15	@st	o	o	o	o	o
16	@deccc	st	o	o	o	o
17	@bne	deccc	st	o	o	o
18	@i6	bne	deccc	st	o	o
19	@i7	i6	bne	decc	st	o

Figure III-132. Entrée des instructions dans le pipeline.

Dans l'étage *PC*, les instructions sont précédées d'un signe '@' car c'est seulement leur adresse qui est dans le registre pc, mais le code instruction lui-même n'a pas encore été lu en mémoire ni décodé. Cette lecture et ce décodage sont réalisés à l'étage suivant *Instr*.
Au cycle 5, un aléa de données se produit pour les registres %r1 et %r2 entre l'instruction `st` qui est dans l'étage *Args* et qui a besoin de leur valeur, et les instructions `setq` qui modifient ces registres et qui n'ont pas fini de s'exécuter. Des bulles sont donc introduites, jusqu'au cycle 8. A ce moment, les instructions `st`, `deccc` et `bne` peuvent progresser dans le pipeline, et l'instruction `@i1` y entre également. Au cycle 9, un aléa de contrôle se produit à cause de la présence et du décodage de l'instruction `bne` à l'étage *Instr*. Le processeur continue à charger les instructions i7, i8, etc. mais au cycle 13, il constate que le branchement est pris et qu'il faut effacer toutes les instructions à gauche du branchement. Le même aléa se reproduit au cycle 17, etc.

Le problème des exceptions

Lorsqu'une exception est prise en compte, le système d'exploitation s'attend à ce que l'instruction en cours soit terminée, puis que le contrôle soit passé au handler de l'exception. Il s'attend aussi à ce que, lorsque l'exécution du handler sera terminée, le contrôle reprenne après l'instruction interrompue.
On comprend qu'avec une exécution basée sur un pipeline, la situation est plus complexe. Tout d'abord, qu'est-ce que « l'instruction en cours » ? En effet, jusqu'à 8 instructions sont simultanément en cours d'exécution dans le pipeline, à des degrés d'avancement divers.

SI l'exception est une interruption, c'est-à-dire à cause externe (timer, I/O, etc.), on peut considérer que « l'instruction en cours » est l'instruction située dans le dernier étage du pipeline, car c'est la seule qui est certaine d'être exécutée. Les instructions qui sont à sa gauche dans le pipeline sont celles qui suivent possiblement dans le programme (ce n'est pas certain) l'instruction interrompue. Le plus simple est de les remplacer par des bulles, car le processeur doit exécuter immédiatement le handler de l'interruption.
Si l'exception est un trap, c'est-à-dire à cause interne (division par 0, instruction illégale, problème d'accès mémoire), elle peut survenir alors que l'instruction n'est pas complètement exécutée. Une instruction illégale sera repérée comme telle dans l'étage *Instr* ; une division par zéro sera détectée dans l'étage *Calc*, une erreur de lecture mémoire dans l'étage *Mem* et une erreur d'écriture mémoire dans l'étage final *Result*. Le problème est qu'il n'est pas certain que cette instruction atteigne le dernier étage du pipeline, à cause d'un branchement ou même d'une interruption. La solution est donc de mettre à 1 un bit spécial du registre d'étage où s'est déclenché le trap, et de continuer à exécuter les instructions du pipeline jusqu'à ce que ce bit, qui est décalé progressivement vers la droite, atteigne le dernier étage, s'il l'atteint. Dans ce cas, on procède comme lors d'une interruption, c'est-à-dire qu'on remplace tous les autres étages par des bulles, et on lance l'exécution du handler de l'exception.

Prédiction de branchement

On a vu dans l'exemple Figure III-132 que les branchements pouvaient introduire de nombreuses bulles dans le pipeline, et ce d'autant plus que celui-ci possède un grand nombre d'étages. Comme il y a en moyenne un branchement toutes les 5 instructions, l'intérêt du pipeline sera fortement diminué s'il n'y a pas moyen de prédire avec une assez bonne certitude si un branchement va être pris ou non.

VI.2. Caches mémoire

VI.2.1. Problème des accès mémoire

L'écart de performance en vitesse entre le processeur et la mémoire a tendance à s'accroitre. Un accès mémoire nécessite parfois plusieurs dizaines de cycles d'horloge du processeur, sachant qu'il y a en moyenne un accès mémoire pour 3 instructions exécutées. Sur un système de type x86 par exemple, on a maintenant des CPU avec des horloges supérieures au GHz, alors que la mémoire fonctionne à des fréquences beaucoup plus basse (figure VI-1). Cette différence de performance aurait un impact très négatif sur la performance globale si on n'introduisait pas de mécanisme pour masquer cette latence.

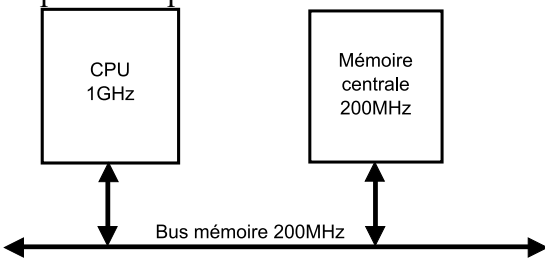


Figure III-133. Différences de vitesse entre le processeur et la mémoire centrale

Pourtant, on sait très bien faire de la mémoire qui fonctionnerait à la fréquence du CPU – qu'on pense par exemple au bistable composé d'une paire de NAND - mais elle ne pourrait être que sur la même puce, et d'une taille réduite.
D'un autre côté, on sait faire des mémoires (relativement) lentes et de grande taille.
Le problème est donc : est-il possible de concevoir une mémoire de grande taille avec des accès rapides, tout au moins dans la plupart des cas ? La réponse est : oui, si on utilise le mécanisme des mémoires cache.

VI.2.2. Mémoire cache

Un mémoire cache est une petite mémoire, très rapide, placée entre le CPU et la mémoire centrale plus lente (Figure III-134). Le processeur n'a aucunement conscience de sa présence et tout se passe pour lui comme s'il dialoguait directement avec la mémoire centrale.

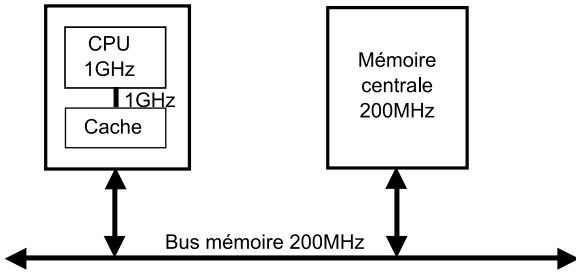


Figure III-134. Le cache est placé entre le CPU et la mémoire centrale

L'idée de base est de garder les mots les plus utilisés dans le cache. Lorsque le processeur fait un accès mémoire en lecture ou en écriture, si le mot recherché est dans le cache, la requête est exécutée immédiatement sans accès à la mémoire centrale et on parle de ***cache hit ;*** sinon on parle de ***cache miss*** et un accès en mémoire centrale est alors effectué. En pratique, on observe un taux de *cache hit* de l'ordre de 80% à 90%, ce qui conduit à une forte diminution de la latence moyenne de la mémoire.

VI.2.3. Localité des accès mémoire

Lorsqu'on fait un accès mémoire à l'adresse x, la probabilité est grande que les prochains accès mémoire (localité temporelle) seront à des adresses proches de x (localité spatiale): mots dans une pile, manipulation de chaînes, de tableaux et matrices, de structures.
Pour exploiter la localité spatiale, le cache contient des copies des mots mémoire par blocs, appelés ***cache lines***. En pratique, les *cache lines* ont une taille de 32 ou 64 octets.
Pour exploiter la localité temporelle, un choix judicieux des *cache lines* à retirer doit être fait lorsqu'il faut rajouter une *cache line* à un cache déjà plein
Par ailleurs, on fait souvent des caches séparés pour les instructions et pour les données (***split cache*** vs ***unified cache***).

VI.2.4. Hiérarchie des caches

- Fréquemment: 3 niveaux de cache
- Level 1 cache : sur la puce CPU elle-même, 128/256K
- Level 2 cache : dans le même package que le CPU, mais pas sur la même puce, 2/4Mo
- Level 3 cache : sur la carte mère

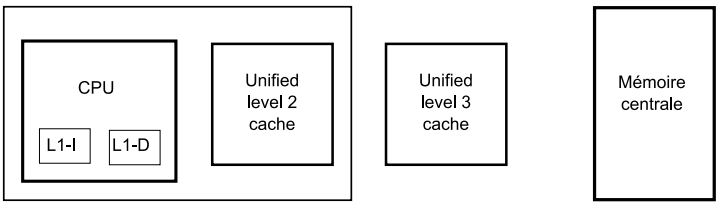


Figure III-135. Hiérarchie de caches à trois niveaux

VI.2.5. Direct-mapped cache

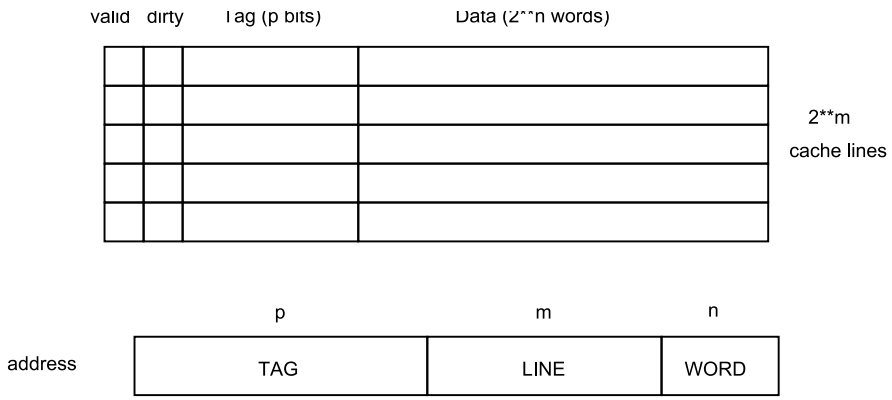


Figure III-136. Organisation de la table des cache lines

- Le bit *valid* indique que la *cache line* contient une copie, exacte ou non, de la mémoire. Initialement tous les bits *valid* sont à 0.
- Le bit *dirty* indique que le contenu de la cache line a été modifié depuis la lecture en mémoire
- Lors d'un accès mémoire, le champ LINE est extrait de l'adresse et permet d'accéder à une *cache line*.
- Si *valid=1* et si TAG égale le champ tag de la *cache line : cache hit.* Sinon: *cache miss*.

- *Cache hit* en lecture: le mot de numéro WORD est extrait de la *cache line*
- *Cache miss* en lecture: le bloc de données est lu en mémoire et placé dans la *cache line*; le champ *tag* de la *cache line* est mis à jour, *valid* ← 1, *dirty* ← 0
- *Cache hit* en écriture: le mot WORD de la *cache line* est mis à jour. Aucune écriture en mémoire centrale n'est faite. *dirty* ← 1.
- *Cache miss* en écriture: la *cache line* accédée est réécrite en mémoire centrale si son bit *dirty* est à 1. Une nouvelle *cache line* associée à TAG/LINE est lue depuis la mémoire, le mot WORD est mis à jour dans la *cache line, dirty* ← 1.

- Design très simple
- Bonnes performances sur des accès à des adresses aléatoires
- Si le processeur fait des accès mémoires à des adresses différentes mais qui ont le même LINE, situation pire que sans cache. Possible avec des accès modulo-N à des tableaux, matrices, etc.

VI.2.6. Set-associative cache

- n-way set-associative cache: chaque cache line est une mémoire associative de n blocs, généralement 2 ou 4
- Le surcoût de recherche dans la mémoire associative est compensé par l'amélioration des performances
- Stratégie de retrait quand une mémoire associative est pleine: généralement LRU (least recently used)

VI.2.7. Snooping caches pour multiprocesseurs à mémoire partagée

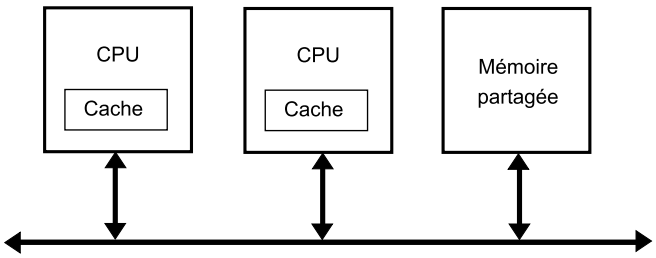


Figure III-137. Organisation d'un système multiprocesseur à mémoire partagée

- Problème: les caches locaux peuvent devenir inconsistants = problème de **cohérence de cache**

VI.2.8. Protocoles de cohérence de caches

- Empêchent différentes versions de la même *cache line* d'apparaître dans plusieurs caches
- Le contrôleur de chaque cache doit écouter les requêtes mémoires faites sur le bus = ***snooping***

VI.2.9. Protocole write-through

- Read miss → lit la donnée en mémoire
- Read hit → lit la donnée dans le cache
- Write miss → met à jour la donnée en mémoire
- Write hit → met à jour le cache et la mémoire, invalide les lignes correspondantes dans les caches des autres CPU. Variante: les autres CPU mettent à jour leur cache au lieu de l'invalider.
- La mémoire est à jour en permanence
- Pb: chaque écriture effectue une opération sur le bus

VI.2.10. Protocole write-back

- Utiliser un bit dirty pour chaque cache line
- Protocole MESI (Pentium, UltraSparc): chaque cache line peut être dans 4 états:
  - I. invalid: la cache line contient une donnée invalide
  - S. shared: plusieurs caches peuvent contenir la donnée, la mémoire est à jour
  - E. exclusive: aucun autre cache ne contient la même ligne, la mémoire est à jour
  - M. modified: la cache line est valide, la mémoire est invalide, aucune copie n'existe

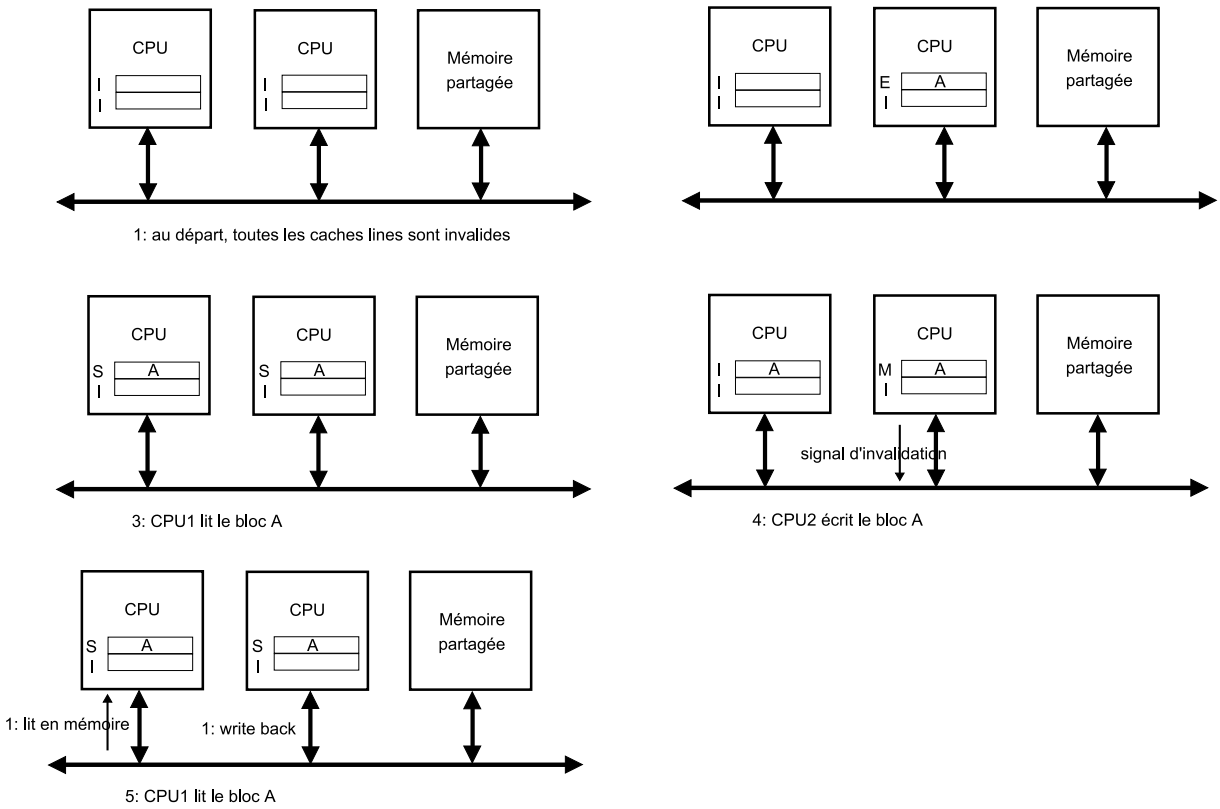


Figure III-138. Evolution des bits MESI lors d'une succession d'accès mémoire en lecture et en écriture

VI.3. Mémoire virtuelle

- Idée: utiliser la mémoire secondaire (disque) pour donner l'illusion d'une mémoire principale plus grande
- Séparation entre l'espace d'adressage réel et l'espace d'adressage virtuel.
- Le processeur travaille avec des adresses virtuelles; le mécanisme les traduit en adresses réelles, de façon transparente pour les programmes
- Nécessite un MMU (memory management unit) et la coopération du système d'exploitation

- La mémoire virtuelle permet:
  - D'augmenter le taux de multi-programmation
  - De mettre en place des mécanismes de protection de la mémoire

VI.3.1. Mémoire virtuelle paginée

- L'espace d'adressage virtuel est découpé en pages (généralement 4K à 64K). L'espace réel est découpé en portions de même taille, appelées ***page frames***.
- Le système maintient en RAM une **table des pages** qui contient notamment l'association numéro de page (virtuelle) → numéro de page frame (réelle)
- Le MMU contient une mémoire associative pour accélérer cette traduction: ***translation lookaside buffer*** (TLB)

VI.3.2. Pagination

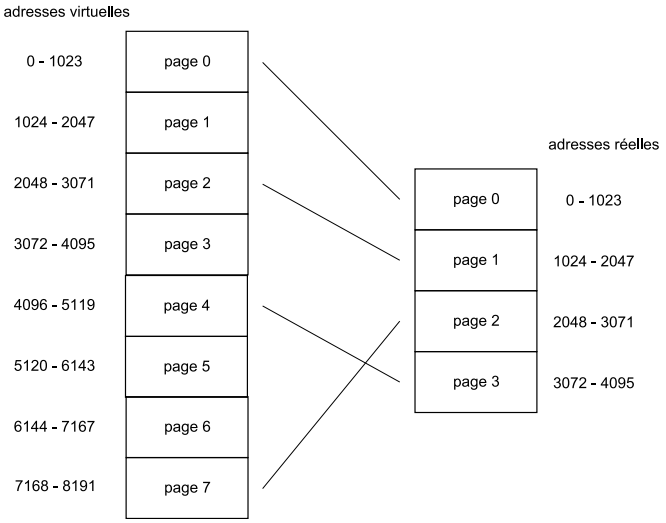


Figure III-139. Adresses virtuelles et adresses réelles

VI.3.3. Table des pages

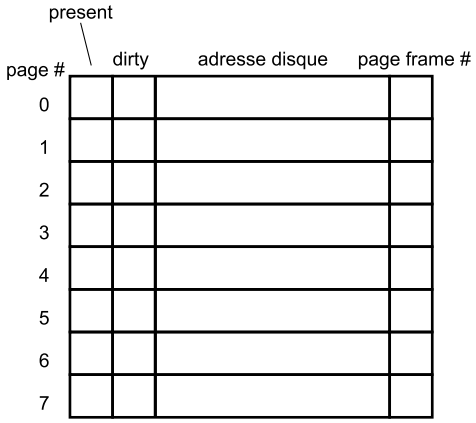


Figure III-140. Table des pages

VI.3.4. Traduction d'adresse

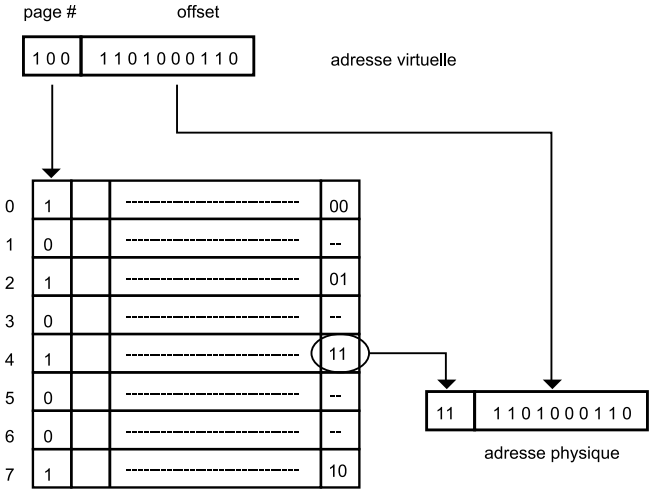


Figure III-141Traduction d'une adresse virtuelle en adresse physique

VI.3.5. Défaut de page

- **Défaut de page** lorsque le MMU indique que la page demandée n'est pas dans la mémoire→ interruption
- Handler de l'IT:
  - Un algorithme de remplacement (FIFO, LRU) choisit une page frame à écraser. Si la page est dirty, elle est réécrite sur le disque
  - La page à charger est lue sur le disque et copiée en mémoire à l'emplacement prévu par le numéro de page frame.
  - La table des pages est mise à jour, notamment pour l'association numéro de page → numéro de page frame

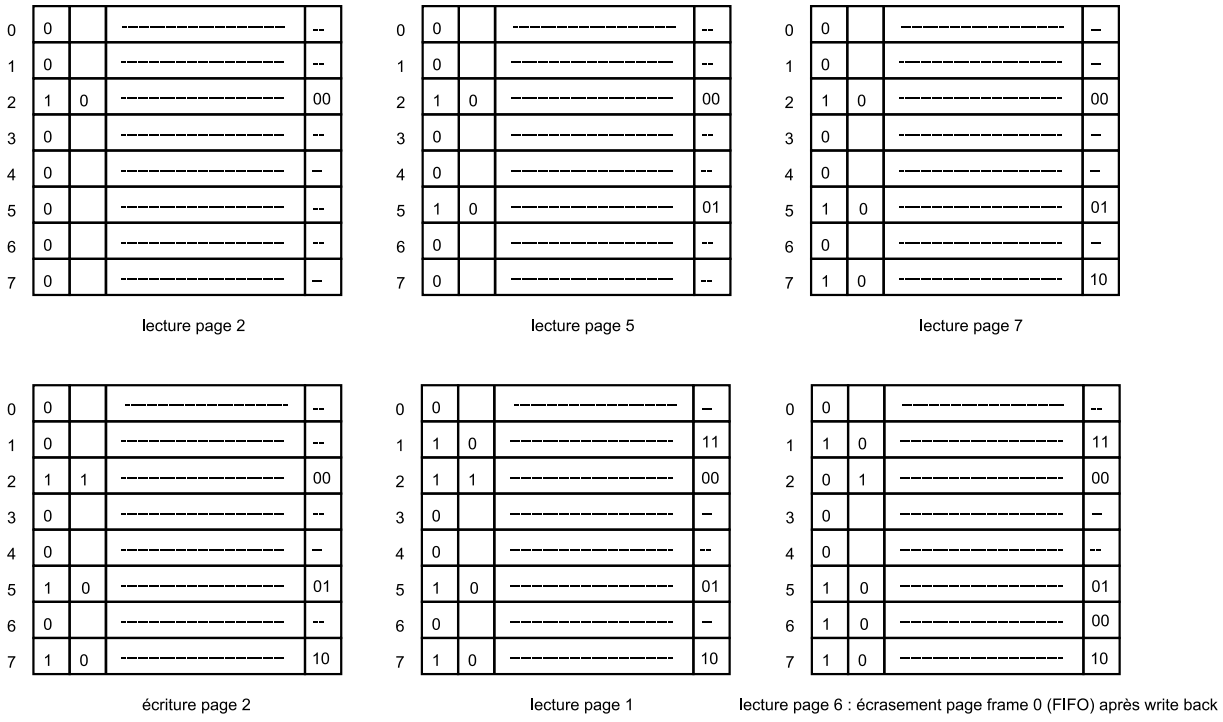


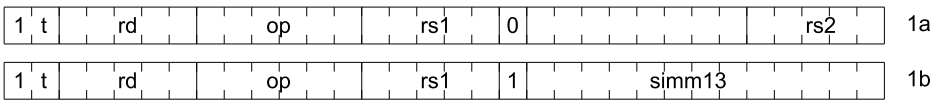
Figure III-142. Evolution de la table des pages lors d'une succession de lecture et écritures sur le disque.

Chapitre VII. CRAPS : guide du programmeur

Le langage assembleur du processeur CRAPS est largement inspiré de celui du SPARC version 8. Il n'implémente pas la notion de fenêtre de registres et les instructions qui suivent les branchements n'ont pas de statut spécial. Il ne possède pas d'unité de calcul flottant.

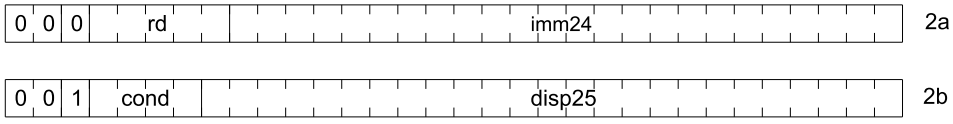
VII.1. Format binaire des instructions de CRAPS

Format 1: instructions arithmétiques, accès mémoire



t = 0 : instruction arithmétique, t = 1 : accès mémoire

Format 2 : sethi et branchements



op (t=0)	Instr.
000000	add
010000	addcc
000100	sub
010100	subcc
011010	umulcc
000001	and
010001	andcc
000010	or
010010	orcc
000011	xor
010011	xorcc
001101	slr
001110	sll

op (t=1)	Instr.
000000	ld
000100	st

cond	branchement
1000	ba
0001	be
1001	bne
0101	bcs
1101	bcc
1110	bpos
0110	bneg
0111	bvs
1111	bvc
1010	bg
0010	ble
1011	bge
0011	bl
1100	bgu
0100	bleu

VII.2. Instructions synthétiques

Les instructions du tableau suivant sont appelées *instructions synthétiques*; ce sont des cas particuliers d'instructions plus générales. Plusieurs d'entre-elles s'appuient sur le fait que %r0 vaut toujours 0.

Instruction	*Effet*	*implémentation*
`clr %ri`	met à zéro %ri	`orcc %r0, %r0, %ri`
`mov %ri,%rj`	copie %ri dans %rj	`orcc %ri, %r0, %rj`
`inccc %ri`	incrémente %ri	`addcc %ri, 1, %ri`
`deccc %ri`	décrémente %ri	`subcc %ri, 1, %ri`
`set val31..0, %ri`	copie *val* dans %ri	`sethi val31..8, %ri` `orcc %ri, val7..0, %ri`

`setq val12..0, %ri`	copie *val* dans %ri	`orcc %ri, val12..0, %ri`
`cmp %ri, %rj`	compare %ri et %rj	`subcc %ri, %rj, %r0`
`tst %ri`	teste nullité et signe de %ri	`orcc %ri, %r0, %r0`
`negcc %ri`	Calcule opposé de %ri	`subcc %r0, %ri, %ri`
`nop`	no operation	`sethi 0,%r0`
`call <label>`	Appel de sous-programme terminal	`or %r0, %r30, %r28` `ba <label>`

`rcall <label>`	Appel de sous-programme avec sauvegarde de l'adresse de retour	`push %r28`
		`call <label>`
		`pop %r28`

`ret`	retour de sous-programme terminal	`add %r28, 1, %r30`
`push %ri`	empile %ri	`sub %r29, 1, %r29` `st %ri, [%r29]`
`pop %ri`	dépile %ri	`ld [%r29], %ri` `add %r29, 1, %r29`

VII.1. Directives de l'assembleur CRAPS

Syntaxe	*Rôle*
`.org` *val32*	Force à *val32* la nouvelle adresse d'assemblage
`label =` *val32*	Associe *label* à *val32* dans la table des symboles
`.word` *val32* `[,`*val32*`]*`	Alloue et initialise des mots mémoires consécutifs
`.global` *sym*	Fait de *sym* un symbole global, visible par les autres modules

VII.1. Tables des branchements conditionnels

Instruction	*Opération*	*Test*
ba	Branch always	1
beq (synonymes : be, bz)	Branch on equal	Z
bne (synonyme : bnz)	Branch on Not Equal	**not** Z
bneg (synonyme : bn)	Branch on Negative	N
bpos (synonyme : bnn)	Branch on Positive	**not** N
bcs (synonyme : blu)	Branch on Carry Set	C
bcc (synonyme : bgeu)	Branch on Carry Clear	**not** C
bvs	Branch on Overflow Set	V
bvc	Branch on Overflow Clear	**not** V

Branchements conditionnels associés à un seul flag.

Instruction	*Opération*	*Test*
bg (synonyme : bgt)	Branch on Greater	**not** (Z **or** (N **xor** V))
bge	Branch on Greater or Equal	**not (N xor V)**
bl (synonyme : blt)	Branch on Less	(N **xor** V)
ble	Branch on Less or Equal	**Z or (N xor V)**

Branchements conditionnels associés à une arithmétique signée.

Instruction	*Opération*	*Test*
bgu	Branch on Greater Unsigned	**not (Z or C)**
bgeu (synonyme : bcc)	Branch on greater than, or equal, unsigned	**not C**
blu (synonyme : bcs)	Branch on less than, unsigned	C
bleu	Branch on Less or Equal Unsigned	**Z or C**

Branchements conditionnels associés à une arithmétique non signée.

VII.2. Jeu d'instructions du processeur CRAPS

Instruction : `add`

Description : idem `addcc`, mais aucun flag n'est affecté.

Instruction : `addcc`

Description : Effectue l'addition des deux opérandes, et place le résultat dans l'opérande résultat. La retenue C n'est pas ajoutée aux arguments. Les flags (condition codes, cc) sont positionnés conformément au résultat.

Flags affectés : N, Z, V, C

Exemple : `addcc %r1, 5, %r1`

Ajoute 5 au contenu de *%r1*, et positionne les flags.

Instruction : `and`

Description : idem `andcc`, mais aucun flag n'est affecté.

Instruction : `andcc`

Description : Effectue un ET logique bit à bit entre les opérandes sources, et place le résultat dans l'opérande résultat. Les flags (condition codes, cc) sont positionnés conformément au résultat.

Flags affectés : N, Z

Exemple : `andcc %r1, %r2, %r3`

Effectue le ET logique bit à bit entre les contenus de *%r1* et *%r2*, et place le résultat dans *%r3*.

Instruction : `b(`*`cc`*`)`

Description : Si la condition *cc* est vérifiée, se branche à l'adresse obtenue en ajoutant 4 x *disp25* à l'adresse de l'instruction courante. Si la condition n'est pas vérifiée, passe à l'instruction suivante en séquence. *disp25* peut être négatif, ce qui correspond à un branchement à un point antérieur du programme. On trouvera en annexe les tables complètes des conditions de test possibles.

Flags affectés : aucun

Exemple : `bcs label`

Se branche à *label* si *C* vaut 1. C'est le déplacement relatif entre label et l'adresse courante de l'instruction, en nombre de mots, qui est codé dans le champ *disp25* de l'instruction.

Instruction : `ld`

Description : Load word. Charge un registre à partir d'un mot de 32 bits de la mémoire centrale, soit 4 octets consécutifs. L'adresse est calculée en ajoutant le contenu du registre du champ *rs1* au contenu du champ *rs2* ou de la valeur immédiate contenue dans le champ *simm13*, selon le cas.

Flags affectés : aucun.

Exemple : `ld [%r2+%r3], %r1`

Copie dans le registre *%r1* le mot mémoire dont l'adresse est obtenue en additionnant les valeurs des registres *%r2* et *%r3*.

Instruction : `or`

Description : idem `orcc`, mais aucun flag n'est affecté.

Instruction : `orcc`

Description : Effectue un OU logique bit à bit entre les opérandes sources, et place le résultat dans l'opérande résultat. Les flags (condition codes, cc) sont positionnés conformément au résultat.

Flags affectés : N, Z

Exemple : `orcc %r1, 1, %r1`

Positionne à 1 le bit de poids le plus faible de *%r1* et laisse tous les autres inchangés.

Instruction : `reti`

Description : Défait les empilements de %pc et de %psr qui sont réalisés lors de la prise en compte d'une exception. %psr reprend donc sa valeur initiale (et en particulier les flags), et %pc reprend la valeur qu'il avait au moment du départ. Cette instruction doit être placée à la fin de chaque handler d'interruption.

Flags affectés : potentiellement tous, lors du dépilement de %psr

Exemple : `reti`

Instruction : `sethi`

Description : Positionne les 24 bits de poids forts d'un registre, et force à zéro les 8 bits de poids faibles.

Flags affectés : aucun

Exemple : `sethi 0xE2F1A0, %r1`

Positionne les 24 bits de poids forts de *%r1* à $E2F1A0_{16}$ et force à 0 les 8 bits de poids faibles.

Instruction : `sll`

Description : Décale le contenu d'un registre vers la gauche d'un nombre de positions désigné par le deuxième argument, compris entre 0 et 31. Si ce nombre est fourni sous forme d'une valeur de registre, seuls les 5 bits de poids faibles sont utilisés.

Flags affectés : aucun

Exemple : `sll %r1, 7, %r2`

Décale le contenu de *%r1* vers la gauche de 7 bits, avec insertion de 7 zéros par la droite, et stocke le résultat dans *%r2*. Aucun flag n'est affecté.

Instruction : `srl`

Description : Décale le contenu d'un registre vers la droite d'un nombre de positions désigné par le deuxième argument, compris entre 0 et 31. Si ce nombre est fourni sous forme d'une valeur de registre, seuls les 5 bits de poids faibles sont utilisés.

Flags affectés : aucun

Exemple : `srl %r1, 7, %r2`

Décale le contenu de *%r1* vers la droite de 7 bits, avec insertion de 7 zéros par la gauche, et stocke le résultat dans *%r2*. Aucun flag n'est affecté.

Instruction : `st`

Description : Stocke le contenu d'un registre en mémoire centrale. L'adresse est calculée en ajoutant le contenu du registre du champ *rs1* au contenu du champ *rs2* ou de la valeur contenue dans le champ *simm13*, selon le cas. Le champ *rd* est utilisé pour désigner le registre source.

Flags affectés : aucun

Exemple : `st %r1, [%r2]`

Copie le contenu du registre *%r1* dans la case mémoire dont l'adresse est la valeur de %r2. Dans ce cas particulier, *rs2* = 0, et l'instruction pourrait être écrite : st %r1, [%r2+%r0]

Instruction : `sub`

Description : idem `subcc`, mais aucun flag n'est affecté.

Instruction : `subcc`

Description : Effectue la soustraction des deux opérandes, et place le résultat dans l'opérande résultat. Les flags (condition codes, cc) sont positionnés conformément au résultat.

Flags affectés : N, Z, V, C

Exemple : `subcc %r1, 5, %r1`

Soustrait 5 au contenu de *%r1*, et positionne les flags.

Instruction : `umulcc`

Description : Unsigned multiply. Effectue une multiplication non signée 16 bits x 16 bits vers 32 bits. Seuls les 16 bits de poids faibles des deux opérandes source sont pris en compte dans le calcul.

Flags affectés : Z

Exemple : `umulcc %r7, %r1, %r6`

Copie dans %r6 le résultat de la multiplication non signée entre les 16 bits de poids faibles de %r7 et les 16 bits de poids faibles de %r1.

Instruction : `xor`

Description : idem `xorcc`, mais aucun flag n'est affecté.

Instruction : `xorcc`

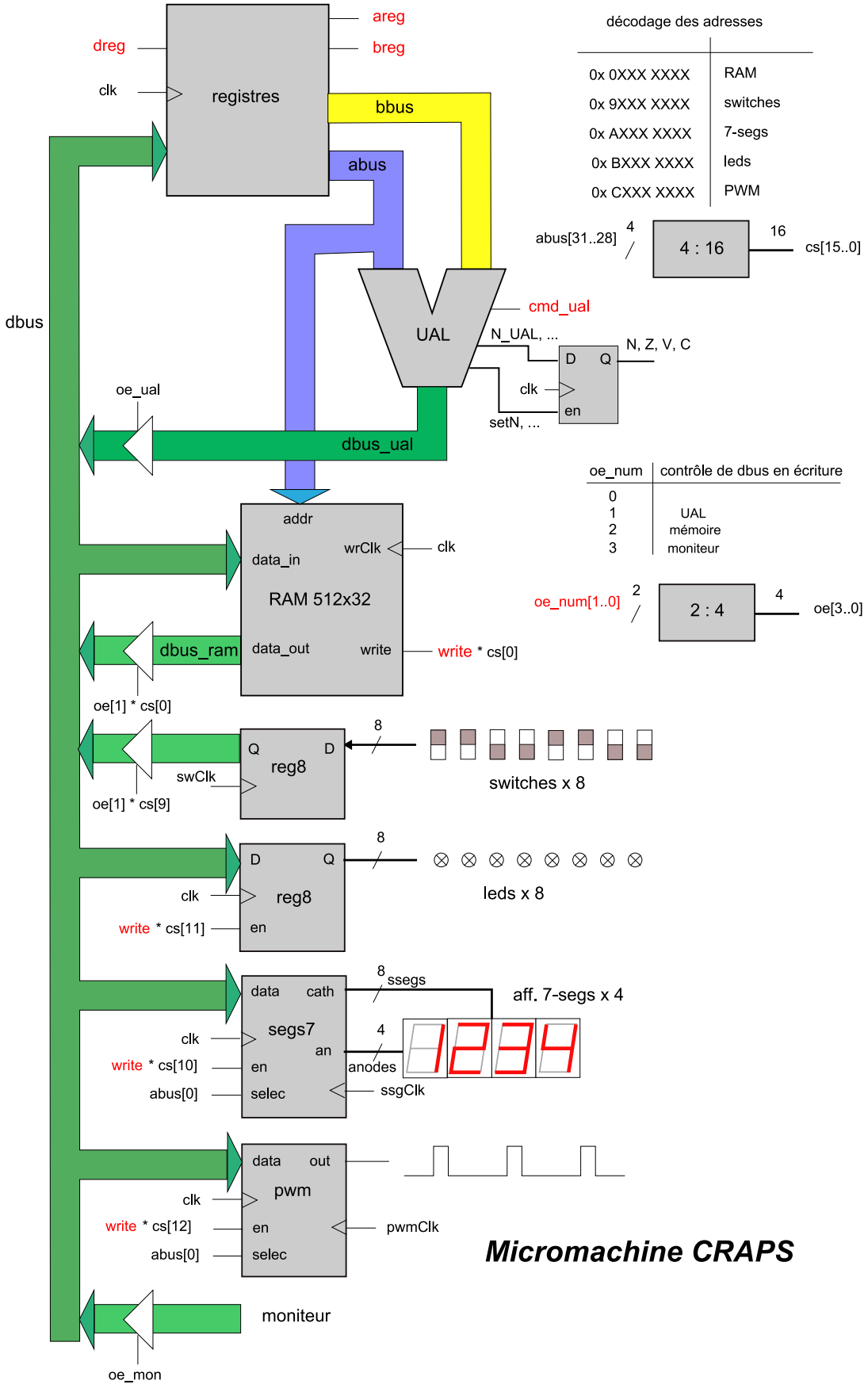
Description : Effectue un XOR logique bit à bit entre les opérandes sources et place le résultat dans l'opérande résultat. Les flags (condition codes, cc) sont positionnés conformément au résultat.

Flags affectés : N, Z

Exemple : `xorcc %r7, %r1, %r6`

Copie dans %r6 le xor calculé bit à bit entre %r7 et %r1, et positionne les flags N et Z en conséquence.

VII.3. La micromachine de CRAPS



SYNTAXE SHDL PAR L'EXEMPLE

MODULES

```
module xor3(a, b, c : s)          // module à 3 entrées a,b,c et une sortie s
  s = a*/b*/c + /a*b*/c + /a*/b*c; // équation combinatoire
end module

module xor6(a, b, c, d, e, f, s)        // le ':' n'est pas obligatoire
  xor3(a, b, c, s1);              // incorporation d'une instance du module xor3
  xor3(d, e, f, s2);              // incorporation d'une 2ème instance du module xor3
  s = s1*/s2 + /s1*s2;            // s = xor des deux sorties intermédiaires
end module

module xor6V(e[5..0] : s)         // même module avec un vecteur d'entrée
  xor3V(e[2..0], s1);
  xor3V(e[5..3], s2);
  s = s1*/s2 + /s1*s2;
end module

module descendant(rst, h, : down)       // circuit séquentiel
  down := e;                      // := affectation séquentielle (ici : bascule D)
  down.clk = h;                   // horloge, front montant
  down.rst = rst ;                // signal de reset asynchrone
end module
```

AFFECTATIONS COMBINATOIRES

```
x = y*z + u*v*w ;
x = 0 ;                                 // autorise car la valeur est 0 ou 1
x = 123 ;                               // INTERDIT (x scalaire, arité 1)
x[7..0] = 123 ;                         // pas de problème car 123 tient sur 8 bits
x[7..0] = 623 ;                         // INTERDIT (incompatibilité d'arités)
x[7..0] = -123 ;
x[7..0] = 0x7E ;
x[7..0] = 0b10101010 ;
x[7..0] = y[7..0]*z[7..0]*u[7..0] ;     // AND position par position
x[7..0] = y[7..0]*z[6..0] ;             // INTERDIT (incompatibilité d'arités)
x[7..0] = y[7..0]+z[7..0]+u[7..0] ;     // OR position par position
x[7..0] = y[7..0]+z[6..0] ;             // INTERDIT (incompatibilité d'arités)
x[7..0] = y*z[7..0] ;                   // factorisation du scalaire y
x[7..0] = y*z[7..0]*u[7..0]*v ;         // factorisation et AND par position
x[7..0] = y[7..0] * 0b1010 ;
x[7..0] = y[7..0] * 0b1010101010 ;      // INTERDIT (incompatibilité d'arités)
x[7..0] = y[7..0] + 0b1101 ;
```

TRI-STATE

```
x = y:oe;           // buffer 3-états de y vers x, commandé par oe
x = /y:/oe;         // idem avec des inversions sur l'entrée et la commande
x = y*z:oe;         // INTERDIT (tri-state uniquement pour les buffers 3 états)
x[7..0] = y[7..0]:/oe;          // vecteur de buffers 3-états avec mise en commun
de la commande
x[7..0] = 0b10101010:oe;        // entrée littérale, écrite en binaire
x[7..0] = y[7..0]:0b10101010;
x[7..0] = 0b11110000:0b10101010;
x[7..0] = 0b1010:0b10101010;
```

```
x[7..0] = 0b11110000:0b101010;   // INTERDIT (incompatibilité d'arités)
```

AFFECTATIONS SEQUENTIELLES

```
q := d ;                          // équation d'évolution de la bascule D
q := /d ;                         // bascule D avec entrée inversée
q := /t*q + t*/q ;                // bascule T
q := /q*t + /t*q ;                // bascule T avec entrée inversée
q := /k*q + j*/q ;                // bascule JK
q := a*q + b*/q ;                 // bascule JK : b=J, /a=K
q := a*q + b*/c ;                 // INTERDIT (ne correspond à aucune bascule)
q := a*q ;                        // INTERDIT (ne correspond à aucune bascule)
q.rst = /nrst ;                   // reset asynchrone sur niveau bas
q.rst = a * b ;                   // INTERDIT (seulement un terme à un seul signal)
q.ena = en ;                      // en = signal d'enable synchrone
q.set = set ;                     // set = signal de mise à 1 asynchrone
q.clk = /h ;                      // horloge de la bascule : front descendant de h
q[7..0] := 123 ;
q[7..0] := /d[7..0] ;             // vecteur de bascules D
q[7..0] := /t[7..0]*q[7..0] + t[7..0]*/q[7..0] ;     // vecteur de bascules T
q[7..0].ena = en ;                // mise en commun
q[7..0].ena = en[7..0] ;
q[7..0].ena = en[5..0] ;          // INTERDIT (incompatibilité d'arités)
```

SIGNAUX PREDEFINIS SUR NEXYS

```
mclk                // horloge 50MHz
btn[3..0]           // 4 boutons poussoir
sw[7..0]            // 8 switches à glissière
ld[7..0]            // 8 leds haute luminosité
ssg[7..0]           // cathodes (dp, g, f, e, d, c, b, a) des afficheurs 7 segments
an[3..0]            // anodes des afficheurs 7 segments
red, grn, blue      // port VGA : 3 signaux de couleur
hs, vs              // port VGA: signaux de synchro. horizontale et verticale
ja_out[7..0]        // connecteur d'extension ja, signaux de sortie
jb_out[7..0]        // connecteur d'extension jb, signaux de sortie
jc_out[7..0]        // connecteur d'extension jc, signaux de sortie
jd_out[7..0]        // connecteur d'extension jd, signaux de sortie
```