Architecture système

Pierron Théo

ENS Ker Lann

Table des matières

Chapitre 1

Logique combinatoire

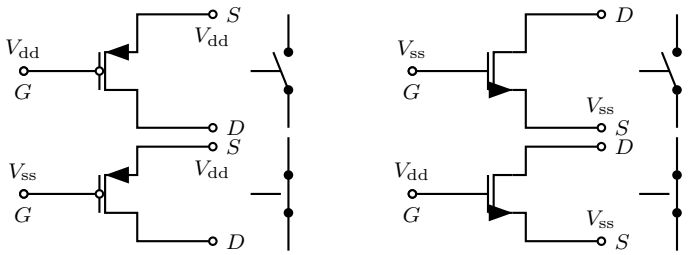
1.1 Transistors et table de vérité

Un microprocesseur est un circuit intégré fait d'un assemblage complexe de centaines de millions (voire de milliards) de transistors qui calculent essentiellement des fonctions logiques sur des signaux numériques. Conceptuellement, la complexité est maitrisée par une forte hiérarchisation des objets qui constituent le circuit final. L'assemblage de plusieurs transistors forme des portes élémentaires qui réalisent des fonctions très simples. À partir de ces portes élémentaires, des composants plus complexes sont créés.

Aujourd'hui, la technologie la plus largement répandue pour concevoir des circuits intégrés est la technologie CMOS (*Complementary Metal Oxide Semi-conductor*). Elle fait appel à un seul composant électronique de base, le transistor à effet de champ (MOSFET : *Metal Oxide Semi-conductor Field Effect Transistor*) qui se décline en deux versions : le transistor MOFSET de type N (nMOS) et le transistor MOFSET de type P (pMOS). La combinaison de ces deux types de transistors procure tous les éléments nécessaires à la réalisation de portes logiques élémentaires. La complexité d'un circuit se mesure en nombre de transistors, qu'ils soient de type N ou de type P. Le circuit le plus dense, réalisé en 2011 (circuit Stratix-5 d'Altera) compte 3,9 milliards de transistors. Le Quad-Core Itanium Tukwila d'Intel (2008) en intègre, quant à lui, 2 milliards.

Le but, ici, n'est pas de décrire le fonctionnement électronique d'un transistor. Nous n'en donnons qu'une idée intuitive, mais suffisante pour comprendre le fonctionnement des portes logiques élémentaires. Un transistors à effet de champ possède 3 « pattes » : une source S, un drain D et une grille G. La tension appliquée sur la grille module le courant qui traverse le transistor. Dans le monde du numérique, la modulation est en tout ou

rien. Ainsi, sur la grille seront appliquées des tensions telles que le transistor sera passant ou non-passant, mais pas dans un état intermédiaire. On peut abstraire ce fonctionnement en considérant les transistors comme des interrupteurs commandés par leur grille.



(a) pMOS (b) nMOS

FIGURE 1.1 – Transistors

Les tensions appliquées surr les transistors sont référencées pas V_{dd} (tension positive) et V_{ss} (tension négative). Ce sont les tensions d'alimentation sur lesquelles on connecte les sources des transistors en fonction de leur type (V_{dd} pour les MOFSET de type P et V_{ss} pour ceux de type N). Lorsque V_{GS} est nulle, le transistor est bloqué. En général, la tension V_{ss} est nulle (borne reliée à la masse) et V_{dd} est reliée à l'alimentation générale du circuit.

Pour simplifier et basculer vers le monde numérique, nous ne considérerons que ces deux valeurs en les abstrayant par une valeurs logique 0 ou 1. Arbitrairement, on peut choisir que 0 représente V_{ss} et 1 représente V_{dd}. La valeur des signaux circulant entre les transistors ne pourra donc prendre que l'une de ces deux valeurs. D'un point de vue électronique, c'est une simplification importante qui ne reflète pas toujours la réalité, mais qui permet d'appréhendre facilement le fonctionnement des circuits digitaux.

Un circuit est donc un ensemble hiérarchique d'objets au travers desquels circulent des signaux logiques. Ces objets peuvent être assimilés à des fonctions logiques dont le comportement peut être traduit par le biais d'une table de vérité. Par exemple considérons un objet numérique à 3 entrées (E_0, E_1 et E_2) qui produit la valeur logique 1 sur sa sortie (S) quand au moins deux de ces entrées sont à 1. Sa table de vérité sera la suivante :

E_0	E_1	E_2	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 1.2 – Table de vérité

Si la fonction logique possède N entrées, il fait reporter le 2^N combinaisons possibles dans la table de vérité.

Ce chapitre aborde essentiellement les portes logiques élémentaires et quelques composants de base qui se retrouvent au sein de tous les circuits numériques. Les circuits arithmétiques, qui sont également des circuits purement combinatoires, ne sont cependant pas abordés ici. Il feront l'ojet d'un chapitre à part entière.

1.2 Portes élémentaires

Dans cette section, nous présentons les trois portes élémentaires de base qui servent à la construction électronique des circuits. Il s'agit des portes NOT, NAND et NOR. En combinant ces portes entre elles, nous pouvons donc élaborer n'importe quelle fonction logique.

1.2.1 Porte NOT

Le porte NOT (ou porte NON, ou plus communément l'inverseur) fait la négation (ou l'inversion) de son signal d'entrée. Elle ne possède donc qu'une entrée et qu'une sortie. Son symbole est le suivant :

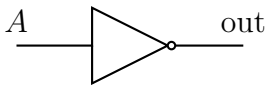


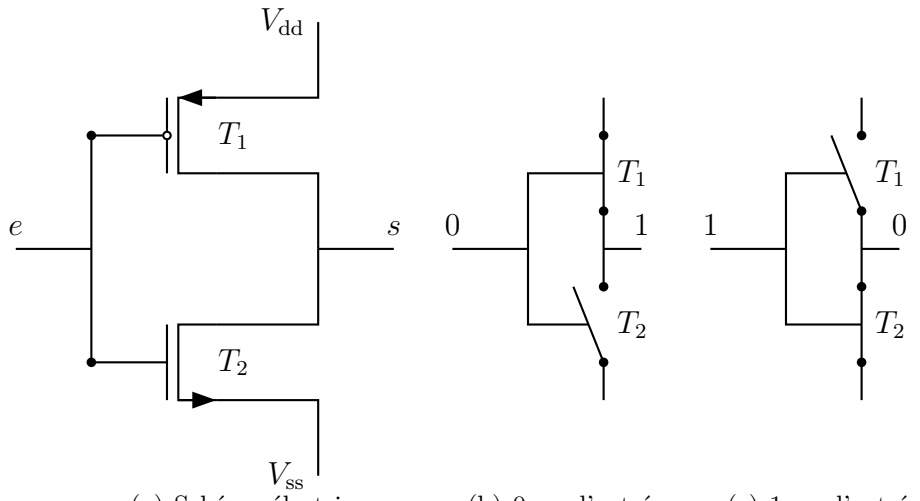
FIGURE 1.3 – Porte NOT – Symbole

Sa table de vérité est :

A	out
0	1
1	0

FIGURE 1.4 – Porte NOT – Table de vérité

D'un point de vue électronique l'inverseur est constitué par l'assemblage de deux transistors CMOS comme indiqué sur la figure 1.5(a). Si on substitue les transistors par des interrupteurs, on peut intuitivement expliquer le fonctionnement de cette porte de la manière suivante : lorsqu'un niveau 0 est présent sur l'entrée e (figure 1.5(b)), le transistor T_1 (pMOS) est passant (interrupteur fermé) et le transistor T_2 (nMOS) est bloqué (interrupteur ouvert). La sortie S est donc connectée à V_{dd} (niveau logique 1). Inversement, lorsqu'un niveau 1 est présent sur l'entrée e (figure 1.5(c)), le transistor T_1 est bloqué (interrupteur ouvert) et le transistor T_2 est passant (interrupteur fermé), mettant la sortie S au niveau V_{ss} (niveau logique 0).



(a) Schéma électrique (b) 0 sur l'entrée (c) 1 sur l'entrée

FIGURE 1.5 – Porte NOT – Fonctionnement

Dans la suite du document, pour ne pas surcharger les schémas, V_{dd} sera représenté par le signe + et V_{ss} par une flèche vers le bas. De plus, la négation peut être juste notifié par un petit cercle, noit en entrée, soit en sortie des portes.

1.2.2 Porte NAND

La porte NAND (ou porte NON ET) la plus simple possède deux entrées et une sortie. Elle réalise la négation d'un ET logique. Son symbole est le

suivant :

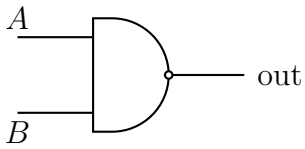


FIGURE 1.6 – Porte NAND – Symbole

Sa table de vérité est :

A	B	out
0	0	1
0	1	1
1	0	1
1	1	0

FIGURE 1.7 – Porte NAND – Table de vérité

Elle est réalisée au moyen de quatre transistors :

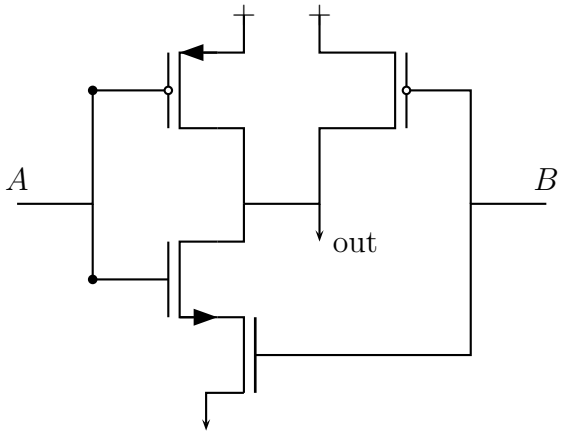


FIGURE 1.8 – Porte NAND – Réalisation pratique

Lorsqu'une des deux entrées A ou B est au niveau logique 0, un des deux transistors pMOS est passant et un des deux transistors nMOS est bloqu“. La sortie out est alors reliée à V_{dd} et est donc au niveau logique 1. Si les deux entrées sont au niveau logique 0, les deux transistors pMOS sont passants et les deux transistors nMOS sont bloqués. La situations est donc identique : la sortie out est au niveau logique 1. Par contre, si les entrées A et B sont au niveau logique 1, les deux transistors pMOS sont bloqués et les

deux transistors nMOS sont passants. La sortie est alors reliée à V_{ss}, donc au niveau logique 0.

Le nombre d'entrée d'une porte NAND peut être supérieur à 2. Si N est le nombre d'entrées, il faudra $2N$ transistors pour la réaliser : N pMOS connectés en parallèle et N nMOS connectés en série.

1.2.3 Porte NOR

La porte NOR (ou porte NON OU) la plus simple possède deux entrées et une sortie. Elle réalise la négation d'un OU logique. Son symbole est le suivant :

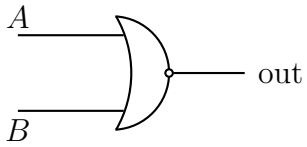


FIGURE 1.9 – Porte NOR – Symbole

Sa table de vérité est :

A	B	out
0	0	1
0	1	0
1	0	0
1	1	0

FIGURE 1.10 – Porte NOR – Table de vérité

Elle est également réalisée au moyen de 4 transistors (NAND inversée) :

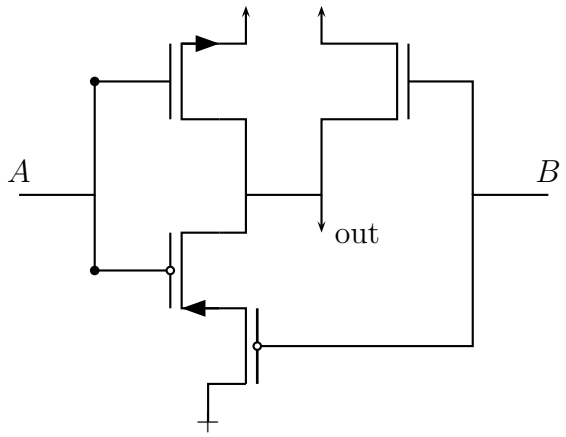


FIGURE 1.11 – Porte NOR – Réalisation pratique

Lorsqu'une des deux entrées A ou B est au niveau logique 1, un des deux transistos pMOS est passant et un des deux transistors nMOS est bloqué. La sortie out est alors reliée à V_{ss} et est donc au niveau logique 0. Si les deux entrées sont au niveau logique 1, les deux transistors pMOS sont bloqués et les deux transistors nMOS sont passants. La situation est donc identique : la sortie out est au niveau logique 0. Par contre, si les entrées A et B sont au niveau logique 0, les deux transistors pMOS sont passants et les deux transistors nMOS sont bloqués. La sortie est alors reliée à V_{dd}, donc au niveau logique 1.

Comme la porte NAND, le nombre d'entrées d'une porte NOR peut être supérieur à 2. Si N est le nombre d'entrées, il faudra $2N$ transistors pour la réaliser : N transistors pMOS en série et N transistors nMOS en parallèle.

1.2.4 Délai associé aux portes

Les trois portes NOT, NAND et NOR sont optimales en nombre de transistors et donc très rapides lorsqu'elles changent d'état car les signaux électrques traversent un nombre minimal de transistors. Notez cependant que le temps entre l'instant où on applique des niveaux logiques sur les entrées et le temps où on récupère le signal de sortie correspondant n'est pas nul : il existe un délai nommé Δ. Par la suite, ce sera le délai correspondant aux trois portes élémentaires. Pour simplifier, on considérera que le délai d'un inverseur, d'une porte NAND à eux entrées ou d'une porte NOR à deux entrées est identique. Pour des portes à N entrées, on considère un délai égale à $\lceil \log_2 N \rceil \Delta$. Le délai dépend bien sûr de la technologie et se mesure en centaines de picosecondes dans les dernières technologies.

1.3 Autres portes élémentaires

À partir des trois portes élémentaires présentées précédemment, nous pouvons augmenter le nombre de portes pour représenter les circuits. Les quatre autres portes de base les plus couramment utilisées sont les pourtes AND, OR, XOR et NXOR.

1.3.1 Porte AND

La porte AND (ou porte ET) la plus simple possède deux entrées et une sortie. Elle réalise un ET logique entre ses entrées. Son symbole est le suivant :

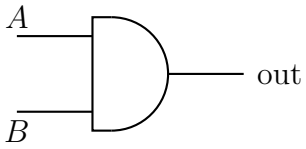


FIGURE 1.12 – Porte AND – Symbole

Sa table de vérité est :

A	B	out
0	0	0
0	1	0
1	0	0
1	1	1

FIGURE 1.13 – Porte AND – Table de vérité

Sa réalisation en termes de transistors consiste à implémenter une porte NAND suivie d'un inverseur. Elle est donc constituée de 6 transistors. Comme la porte NAND, son nombre d'entrées peut être quelconque.

1.3.2 Porte OR

La porte OR (ou porte OU) la plus simple possède deux entrées et une sortie. Elle réalise un OU logique entre ses entrées. Son symbole est le suivant :

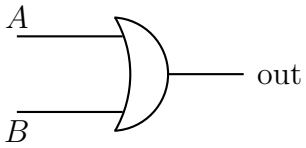


FIGURE 1.14 – Porte OR – Symbole

Sa table de vérité est :

A	B	out
0	0	0
0	1	1
1	0	1
1	1	1

FIGURE 1.15 – Porte OR – Table de vérité

Sa réalisation en termes de transistors consiste à implémenter une porte NOR suivie d'un inverseur. Elle est donc constituée de 6 transistors. Comme la porte NOR, son nombre d'entrées peut être quelconque.

1.3.3 Porte XOR

La porte XOR (ou porte OU exclusif) la plus simple possède deux entrées et une sortie. Elle réalise un OU exclusif entre ses entrées. Son symbole est le suivant :

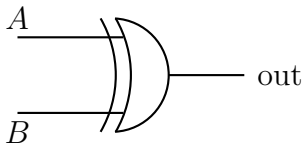


FIGURE 1.16 – Porte XOR – Symbole

Sa table de vérité est :

A	B	out
0	0	0
0	1	1
1	0	1
1	1	0

FIGURE 1.17 – Porte XOR – Table de vérité

Sa réalisation avec des transistors est plus complexe. Une porte XOR peut également avoir un nombre d'entrées supérieur à deux.

1.3.4 Porte NXOR

La porte NXOR (ou porte NON OU exclusif) la plus simple possède deux entrées et une sortie. Elle réalise le complément d'un OU exclusif entre ses entrées. Son symbole est le suivant :

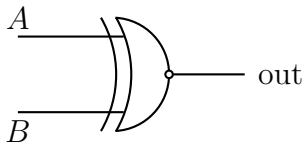


FIGURE 1.18 – Porte NXOR – Symbole

Sa table de vérité est :

A	B	out
0	0	1
0	1	0
1	0	0
1	1	1

FIGURE 1.19 – Porte NXOR – Table de vérité

Cette porte est également appelée coïncidence : elle se positionne au niveau logique 1 lorsque ses deux entrées sont égales.

1.3.5 Délai

Les portes précédentes sont les plus complexes (en termes de nombre de transistors) et sont donc plus lentes que les trois portes NOT, NAND et NOR.

1.4 Mise en oeuvre directe de fonctions logiques

L'implémentation directe d'une fonction logique à l'aide de portes élémentaires peut se déduire directement de sa table de vérité. Si on reprend l'exemple précédent, on a directement le schéma suivant :

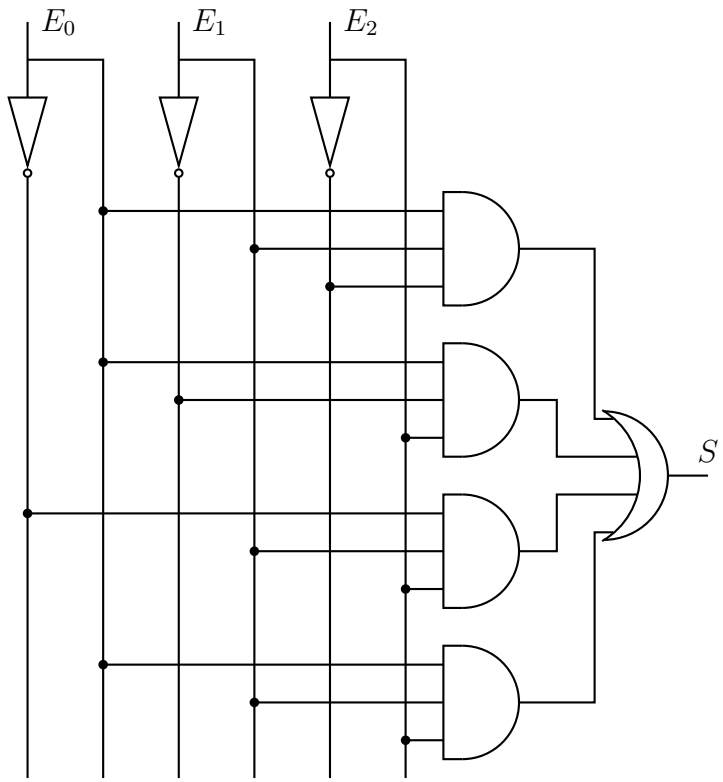


FIGURE 1.20 – Schéma logique naïf

La fonction prend la valeur 1 dans quatres cas. Pour chaque cas, on détexte quelle doit être la configuration d'entrée :

- $E_0 = E_1 = 1$ et $E_2 = 0$
- $E_0 = E_2 = 1$ et $E_1 = 0$
- $E_1 = E_2 = 1$ et $E_0 = 0$
- $E_0 = E_1 = E_2 = 1$

Cette mise en œuvre directe n'est cependant pas optimale. En général, elle conduit à des circuits volumineux, donc gourmands en transistors et par conséquent relativement lents. Dans l'exemple, le temps de propagation est l'addition des temps de traversée d'un inverseur, d'une porte ET à trois entrées et d'une porte OU à 4 entrées, soit un délai total de :

$$\Delta_{\text{inv}} + \Delta_{\text{AND3}} + \Delta_{\text{OR4}} = \Delta + 3\Delta + 3\Delta = 7\Delta$$

Le temps de traversée du ET à trois entrées (resp. OU à quatre entrées) est donné par le temps de traversée d'une porte NAND à trois entres (resp. porte NOR à quatre entrées) plus le temps de traversée d'un inverseur.

1.5 Optimisation

Les portes élémentaires décrites précédemment ont leur équivalent dans l'algèbre de Boole. Dès lors que nous devons concevoir des portes logiques plus complexes, se pose le problème de l'optimisation des circuits correspondants. En général, on cherchera à minimiser le nombre de transistors (surface réduite, circuit rapide) et donc à se ramener à l'agencement de portes élémentaires rapides. Cette étape d'optimisation peut se faire via le formalisme de l'algèbre de Boole que nous supposons connu du lecteur, mais dont nous rappelons l'essentiel.

L'algèbre de Boole considère l'ensemble $\{0, 1\}$ muni des lois :

- OU (noté $+$)
- ET (noté $\times$ ou $\cdot$)
- NON (noté $\neg$ ou $\bar{\cdot}$)

L'opération OU correspond à la porte OR, l'opération ET à la porte AND et l'opération NON à l'inverseur.

Proposition 1.1

- $+$ et $\cdot$ sont associatives et commutatives.
- $+$ est distributive sur $\cdot$ et $\cdot$ est distributive sur $+$
- Pour tout $n \in \mathbb{N}^*$, $\sum_{i=1}^{n} a = a$ et $\prod_{i=1}^{n} a = a$
- 0 est neutre pour $+$ et 1 est neutre pour $\cdot$: $a + 0 = a$ et $a \cdot 1 = a$.
- $0 \cdot a = 0$ et $1 + a = 1$.
- Absorption : $a + ab = a$ et $a(a + b) = a$
- Simplification : $a + \overline{a}b = a + b$ et $a(\overline{a} + b) = ab$
- Redondance : $ab + \overline{a}c = ab + \overline{a}c + bc$
- $\overline{\overline{a}} = a$, $a + \overline{a} = 1$ et $a\overline{a} = 0$

Théorème 1.1 de De Morgan *$\overline{a + b} = \overline{a}\overline{b}$ et $\overline{ab} = \overline{a} + \overline{b}$.*

Remarque 1.1 Ce théorème permet de transsformer une expression avec des opérateurs $+$ en une expression avec des opérateurs $\cdot$ et réciproquement. Il est particulièrement intéressant lorsqu'on cherche à exprimer une expression à l'aide de portes NOR ou de portes NAND.

Exemple 1.1 En reprenant l'exemple de l'introduction, la sortie S peut maintenant s'exprimer de la manière suivante :

$$S = E_0E_1\overline{E_2} + E_0\overline{E_1}E_2 + \overline{E_0}E_1E_2 + E_0E_1E_2$$

par une suite de transformations tirées des propriétés de l'algèbre de Boole, on simplifie cette équation et on cherche à la mettre sous une forme convenable pour une implémentation à l'aide des portes logiques élémentaires.

Par idempotence,

$$S = E_0E_1\overline{E_2} + E_0E_1E_2 + E_0\overline{E_1}E_2 + E_0E_1E_2 + \overline{E_0}E_1E_2 + E_0E_1E_2$$

Par distributivité,

$$S = E_0E_1(E_2 + \overline{E_2}) + E_0E_2(\overline{E_1} + E_1) + E_1E_2(\overline{E_0} + E_0)$$

Par complémentarité,

$$S = E_0E_1 + E_1E_2 + E_0E_2$$

En utilisant le théorème de De Morgan, on obtient une formule plus facile à implémenter :

$$S = \overline{\overline{E_0E_1} \cdot \overline{E_0E_2} \cdot \overline{E_1E_2}}$$

puisqu'elle ne fait appel qu'à des portes NAND, ce qui nous donne le schéma suivant :

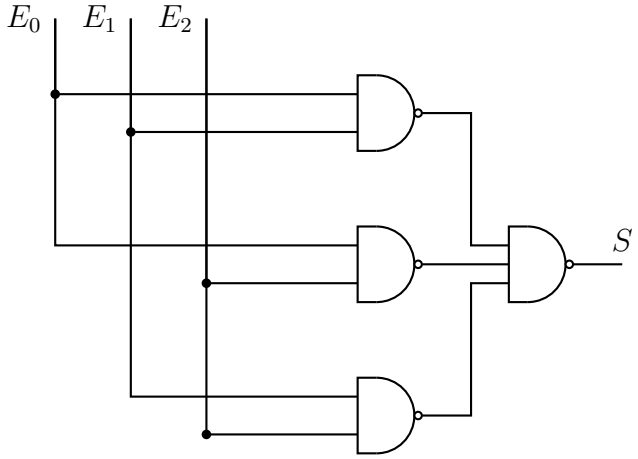


FIGURE 1.21 – Simplification

Par rapport à l'implémentation directe, le nombre de portes est largement réduit, ainsi que le temps de propagation qui n'est plus que de 3Δ.

1.6 Tableaux de Karnaugh

Les tableaux de Karnaugh proposent une autre technique our simplifier les fonctions logiques. Le principe repose sur une inspection visuelle d'une table 2D où les cases adjacentes ne diffèrent que par l'état d'une seule variable. Lorsque 2^p cases adjacentes sont dans le même état, on peut les regrouper

ensemble. Ce regroupement permet d'exprimer de manière plus concise une expression ou une sous-expression logique. Cette méthode utilise également le code de Gray qui a comme propriété de ne faire varier qu'un seul bit entre deux mots successifs.

Si N est le nombre de variables (entrées de la fonction logique), le tableau comprend 2^N cases. Chaque case représente l'état d'une sortie en fonction d'une unique configuration d'entrée. L'attribution des coordonnées d'une case est donnée par le code de Gray.

Dans le cas d'une fonction à trois entrées, par exemple, on génère le tableau de huit cases suivant :

	0 0	0 1	1 1	1 0
0	0	0	1	0
1	0	1	1	1

FIGURE 1.22 – Table de Karnaugh d'une fonction à trois entrées

Les coordonnées horizontales représentent les valeurs des entrées E_0 et E_1 positionnées suivant un code de Gray. Les coordonnées verticales représentent les valeurs de E_2. C'est, en quelque sorte, une représentation 2D de la table de vérité. Par exemple, la case en haut à gauche correspond à la valeur logique de la fonction pour $\overline{E_0} \cdot \overline{E_1} \cdot \overline{E_2}$. Ainsi, la fonction logique S déjà étudiée initialise le tableau avec des 1 dans les cases qui correspondent à au moins deux entrées à 1.

$$S = E_0 E_1 \overline{E_2} + E_0 \overline{E_1} E_2 + \overline{E_0} E_1 E_2 + E_0 E_1 E_2$$

La procédure consiste ensuite à regrouper les cases adjacentes qui sont remplies avec des 1. Le regroupement ne peut s'effectuer que par groupe de 2^p cases. Dans ce cas, on peut regrouper les coefficients $(2, 2)$ et $(2, 3)$, ou $(2, 3)$ et $(2, 4)$ ou encore $(1, 3)$ et $(2, 3)$.

Le regroupement vertical correspond au terme $E_0 E_1$. En effet, ce regroupement correspond à la somme des termes $E_0 E_1 \overline{E_2} + E_0 E_1 E_2 = E_0 E_1$. Quelle que soit la valeur de E_2 (coordonnées verticales), la valeur est toujours à 1. De même, le regroupement le plus à gauche correspond à $E_1 E_2$: le regroupement est indépendant de la valeur de E_0.

Pour faire des regroupements, il faut en fait considérer un tableau de Karnaugh comme un tore. Supposons le tableau de Karnaugh suivant :

	0 0	0 1	1 1	1 0
0 0	1	0	1	1
0 1	0	0	0	0
1 1	0	0	0	0
1 0	1	0	1	1

FIGURE 1.23 – Exemple de tableau de Karnaugh

Deux regroupements peuvent être effectués :

	0 0	0 1	1 1	1 0
0 0	1	0	1	1
0 1	0	0	0	0
1 1	0	0	0	0
1 0	1	0	1	1

FIGURE 1.24 – Regroupements possibles

Le premier regroupe les quatre coins et correspond à $\overline{E_1} \cdot \overline{E_3}$. Le second regroupe deux cases verticales et correspond à $E_0 E_1 \overline{E_3}$.

En général, plusieurs regroupements sont possibles. Il faut alors chercher à trouver les regroupements les plus grands, même s'il y a des chevauchements. Il est ainsi possible de mieux regrouper l'exemple précédent en faisant deux groupes de quatre.

On en déduit une fonction $F = \overline{E_1} \cdot \overline{E_3} + E_0 \overline{E_3} = \overline{E_3}(\overline{E_1} + E_0)$. La factorisation provient du chevauchement.

Il est quelquefois plus simple de calculer le complément de la fonction logique en regroupant les 0. Dans le tableau précédent, on obtiendrait : $\overline{F} = E_3 + \overline{E_0} E_1$ donc $F = \overline{E_3 + \overline{E_0} E_1} = \overline{E_3}(\overline{E_1} + E_0)$.

À partir de cinq variables, il n'est plus possible d'obtenir sur une surface plane un tableau dans lequel une case soit adjacente à cinq autres cases.

1.7 Temps de propagation

Nous l'avons déjà mentionné, le changement d'état d'un transistor n'est pas immédiat. L'onterconnexion de plusieurs transistors pour constituer une porte logique conduit donc fatalement à un temps de réponse de quelques centaines de picosecondes, voire quelques nanosecondes, entre l'instant où des valeurs logiques sont présentées sur ses entrées et l'instant où on récupère la valeur logique en sortie.

De la même manière, une fonction logique complexe est composée d'un agencement de portes élémentaires. Son temps de réponse (ou temps de propagation) est donc fonction du temps de traversée de chacune des portes élémentaires. Plus précisément, le temps de propagation d'un circuit logique est le temps de traversée du plus long chemin entre les entrées et la sortie.

Considérons, par exemple, le schéma du circuit logique suivant :

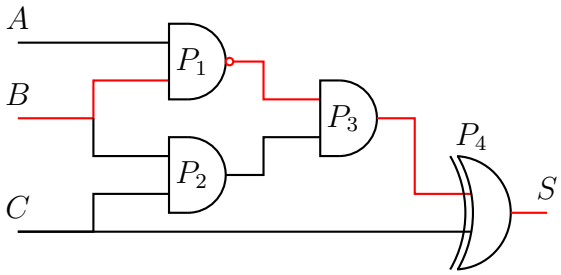


FIGURE 1.25 – Plus long chemin d'un circuit logique

Le plus long chemin correspond au chemin de l'entrée B vers la sortie S. En considérant que le temps de traversée des portes P_2 et P_3 est de 2Δ et que celui de la porte P_4 est de 3Δ, le temps de propagation total est de 7Δ. C'est le temps minimal pour que la sortie S soit stable. En effet, la variation des signaux d'entrée peut entraîner momentanément des variations temporaires (ou transitoires) sur la sortie. Imaginons par exemple que les trois entrées A, B et C soient au niveau 0 depuis un temps très supérieur à Δ. D'après la table de vérité du circuit, la sortie S est à 0 pour cette configuration des entrées.

A	B	C	P_1	P_2	P_3	P_4
0	0	0	1	0	0	0
0	0	1	1	0	0	0
0	1	0	1	0	0	0
0	1	1	0	0	0	0
1	0	0	1	0	0	1
1	0	1	1	0	0	1
1	1	0	1	1	1	0
1	1	1	0	1	1	1

FIGURE 1.26 – Valeurs de sortie de chacune des portes

Les colonnes P_1, P_2, P_3 et P_4 correspondent aux sorties des portes éponymes (ie $P_4 = S$).

Imaginons maintenant la configuration suivante en entrée : $A = 0$ et $B = C = 1$. Le chronogramme suivant donne en fonction du temps (échantilloné suivant Δ) l'état de chaque porte.

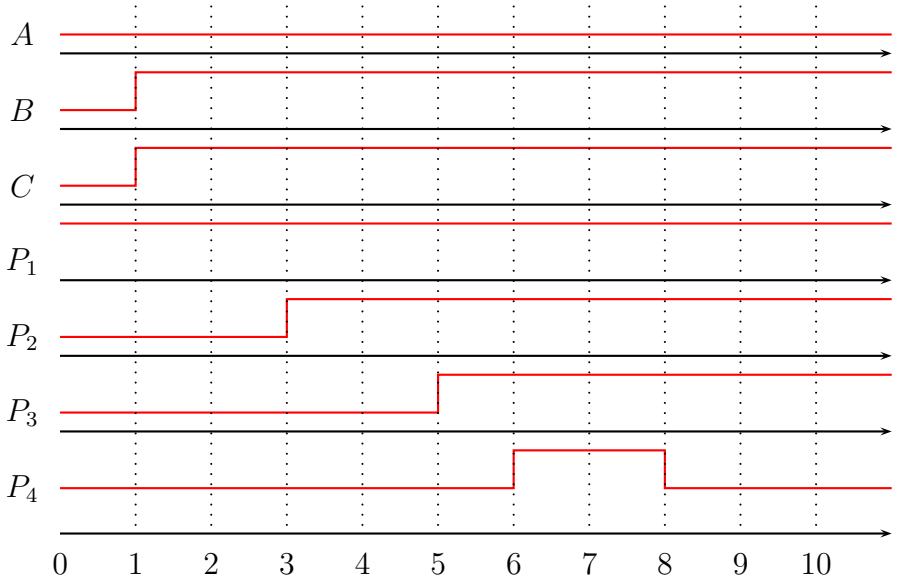


FIGURE 1.27 – Chronogramme

On remarque que pendant les instants $t = 3$ et $t = 4$, les sorties de P_2 et P_3 sont dans des états différents, ce qui induit un basculement transitoire (3Δ plus tard) de la porte P_4. Après 7Δ, le circuit retrouve un état stable. Logiquement, la transition de la configuration d'entrée ($A = B = C = 0$) vers la configuration $A = 0$ et $B = C = 1$ ne provoque pas de changement d'état. D'un point de vue électronique, la réalité est bien différente et on doit en tenir compte dans le processus de conception d'architectures plus complexes.

1.8 Autres fonctions logiques usuelles

1.8.1 Décodeur

Un décodeur possède N entrées et 2^N sorties. À une configuration donnée des N entrées, correspond une sortie (écriture binaire). Lorsque cette configuration est présente sur les entrées, la sortie correspondante est mise à 1 et les autres à 0. Le symbole et la table de vérité d'un décodeur à trois entrées et huit sorties sont les suivants :

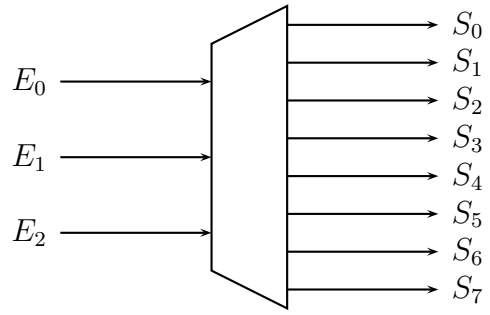


FIGURE 1.28 – Décodeur – Schéma

E_0	E_1	E_2	S_0	S_1	S_2	S_3	S_4	S_5	S_6	S_7
0	0	0	1	0	0	0	0	0	0	0
1	0	0	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
1	1	0	0	0	0	1	0	0	0	0
0	0	1	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

FIGURE 1.29 – Décodeur – Table de vérité

1.8.2 Multiplexeur

Un multiplexeur possède 2^N entrées, une commande de sélection de N bits et une sortie. Son rôle est d'aiguiller une des entrées vers la sortie en fonction de la configuration de sa commande de sélection. Le symbole d'un multiplexeur de 8 vers 1 est le suivant :

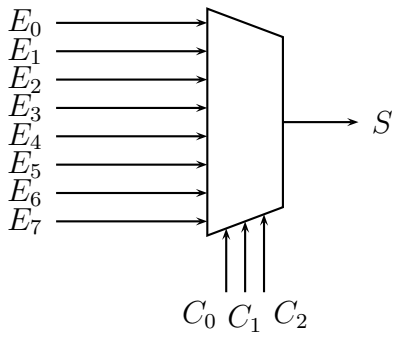


FIGURE 1.30 – Multiplexeur – Symbole

À chaque configuration (C_0, C_1, C_2) est associée une entrée qui est recopiée sur la sortie S. La table de vérité du multiplexeur est donnée par le tableau suivant.

C_0	C_1	C_2	S
0	0	0	E_0
1	0	0	E_1
0	1	0	E_2
1	1	0	E_3
0	0	1	E_4
1	0	1	E_5
0	1	1	E_6
1	1	1	E_7

FIGURE 1.31 – Multiplexeur – Table de vérité

1.9 Bus et portes 3 états

Un bus est un ensemble de fils permettant de véhiculer des informations en provenance de plusieurs sources vers une ou plusieurs destinations. Sa représentation est :

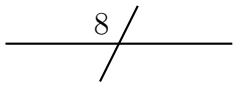


FIGURE 1.32 – Bus de taille 8

La barre diagonale complétée par un entier indique la taille (ou la largeur) du bus. C'est une manière compacte de représenter N fils en parallèle suir lesquels vont transiter des valeurs codées sur N bits.

Un système numérique quui possède par exemple les trois éléments E_1, E_2 et E_3 capables d'émettre une information vers un autre élément récepteur pourra être architecturé autour d'un bus de la manière suivante :

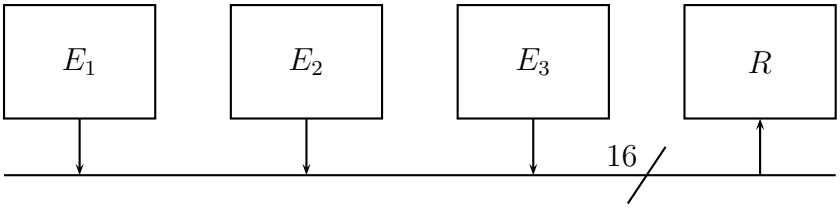


FIGURE 1.33 – Utilisation d'un bus

Pour que cela fonction, il faut bien sûr que les éléments E_1, E_2 et E_3 se synchronisent pour ne pas émettre en même temps des données sur le bus. D'un point de vue électrique, les émetteurs sont constitués de portes logiques élémentaires. Leurs sorties positionnent donc des valeurs logiques 0 ou 1 qui vont donc être en conflit si on les raccorde directement au bus. Pour simplifier, supposons que le dernier étage des émetteurs corresponde à des inverseurs. Une connexion directe est alors équivalente au montage suivant.

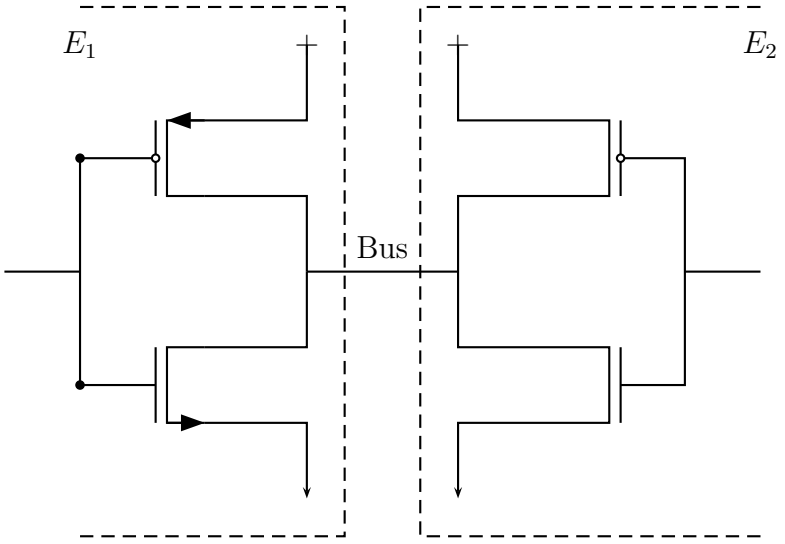


FIGURE 1.34 – Connexion directe de deux inverseurs

Si les deux sorties sont dans le même état, cela ne pose pas de problème. Par contre, si elles possèdent des états différents, il se produit un court-circuit, et donc une dégradation possible (voire certaine) des composants. Par exemple, si la sortie de E_1 est à 0, cela signifique que le transistor nMOS est passant. Si, en même temps, la sortie de E_2 est à 1, le transistor pMOS est passant. Nous avons donc deux transistors passants connectés en série entre V_{dd} et V_{ss}, donc l'équivalent d'un court-circuit (courant infini entre les deux transistors).

On peut généraliser cela à toutes les portes logiques et s'apercevoir qu'on ne peut absolument pas connecter directement deux sorties entre elles sans risque de déterioration.

Pour résoudre ce problème, on introduit les portes 3 états (ou TRI-STATE) qui ont la particularité de posséder un troisième état qui n'est ni 0 ni 1, mais dit « haute impédance ». Dans cet état la porte n'impose pas de tension particulière et ne perturbe pas les autres éléments qui peuvent lui être connectés. Le symbole d'une porte 3 étates est :

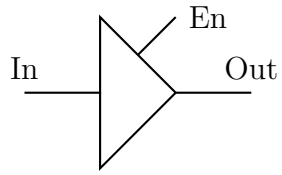


FIGURE 1.35 – Porte 3 états – Symbole

La commande En (Enable) permet soit de transmettre l'entrée In sur la sortie Out, soit de mettre la sortie Out en haute impédance. La table de vérité est :

In	En	Out
0	0	Z
1	0	Z
0	1	0
1	1	1

FIGURE 1.36 – Porte 3 états – Table de vérité

L'état Z représente le troisième état. Le schéma électrique équivalent est :

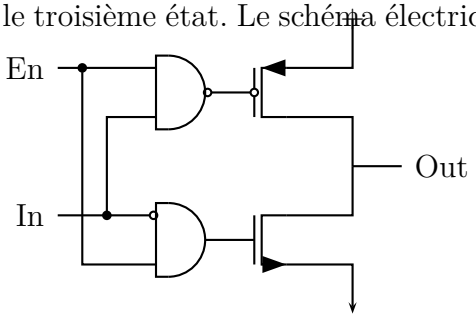


FIGURE 1.37 – Porte 3 états – Montage

Si la sortie En est à 0, les deux transistors nMOS et pMOS sont bloqués et la sortie out n'est donc reliée ni à V_{dd} ni à V_{ss}, mais comme déconnectée du reste du circuit. Si par contre En est à 1, le circuit se comporte comme un double inverseur.

Chapitre 2

Logique séquentielle

2.1 Introduction

Contrairement à la logique combinatoire, la logique séquentielle introduit explicitement la notion de temps. Un opérateur purement logique produira toujours la même sortie pour une configuration donnée de ses entrées. À l'opposé, un opérateur qui intègre une logique séquentielle mémorise un état relatif aux actions passées. Suivant cet état, une même configuration des entrées pourra produire des états de sorties différents.

Un exemple simple est le dispositif à 2 entrées ci-dessous pour allumer une lampe. Un niveau logique 1 sur l'entrée E_1 allume la lampe. Un niveau logique 1 sur l'entrée E_2 éteint la lampe. Ce dispositif est relié à deux boutons poussoirs.

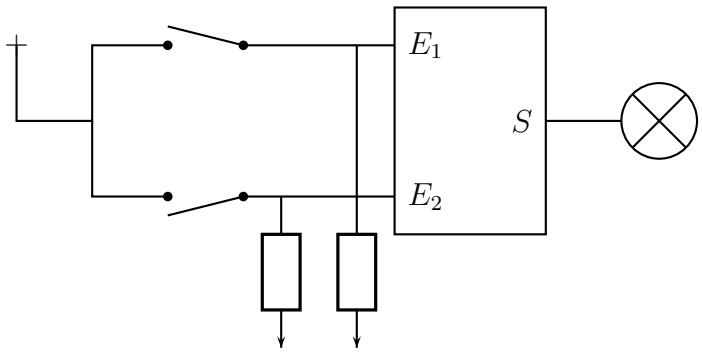


FIGURE 2.1 – Allumage de lampe

Supposons que la lampe soit éteinte. Au repose, $E_1 = E_2 = 0$ ne provoque aucune action et $S = 0$. Si on appuie sur le bouton du haut, la configuration $E_1 = 1$ et $E_2 = 0$ déclenche d'allumage de la lampe. Si on relâche le bouton, on a de nouveau la configuration $E_1 = E_2 = 0$ mais cette fois la sortie est à

1. Une pression sur le bouton du bas remetle système dans l'état de départ. Ce système simple met bien en évidence que la seule donnée des valeurs sur E_1 et E_2 ne suffit pas pour spécifier l'état de sortie S. Il faut tenir compte de l'histoire (ou du passé) du système.

La mémorisation de l'état d'un système se fait au moyen de dispositifs numériques élémentaires qu'on appelle des bascules. Comme pour la logique séquentielle, l'agencement de plusieurs bascules permet d'élaborer des opérateurs séquentiels plus complexes, notamment des composants électroniques essentiels dans la réalisation de systèmes numériques : les registres. On retrouve les registres dans pratiquement toutes les unités qui composent un microprocesseur. Leur fonction principale est la synchronisation des diverses unités à partir d'une horloge.

Dans un système numérique, l'horloge est un signal particulier qui a pour rôle de cadencer l'ensemble des opérations qui s'y déroulent. C'est un signal périodique sur lequel toutes les composantes d'un système se synchronisent afin de faciliter les échanges d'informations. Le composant « registre » capture à un instant donné l'état de ses entrées qui restent ensuite stables pendant toute une période de l'horloge. L'échantillonnage est réalisé sur le front montant (ou descendant) de l'horloge.

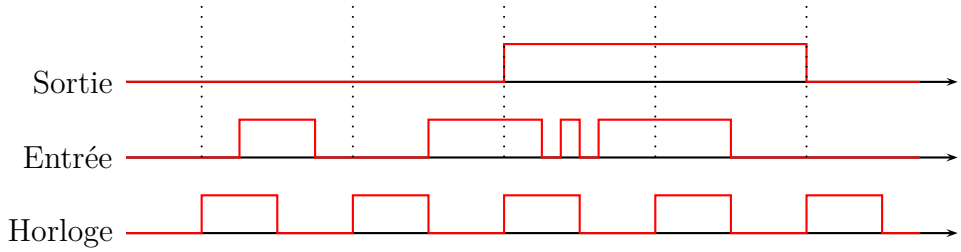


FIGURE 2.2 – Chronogramme d'un registre

Le chronogramme ci-dessus représente l'état de sortie d'un registre en fonction de son entrée et d'une horloge. À chaque front montant de l'horloge, le signal d'entrée est recopié en sortie. Ensuite, pendant une période entière, le signal de sortie reste inchangé, quelles que soient les variations du signal d'entrée.

2.2 Bascules

2.2.1 Bascule Flip-Flop

La bascule la plus simple est la bascule Flip-Flop, encore appelée bistable. Elle mémorise l'état d'un bit (une valeur binaire) et on la retrouve dans les

mémoires statiques. Son architecture est extrêmement simple et consiste en l'interconnexion de deux inverseurs :

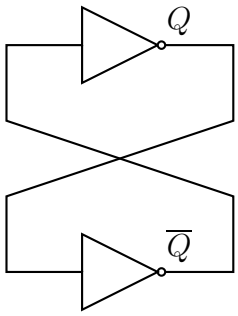


FIGURE 2.3 – Bascule Flip-Flop

On peut facilement vérifier la stabilité de ce dispositif : supposons que la sortie Q de l'inverseur du haut soit à la valeur logique 0. Cette valeur est propagée à l'entrée de l'inverseur du bas qui génère alors la valeur 1 sur sa sortie notée $\overline{Q}$. Une valeur 1 en entrée de l'inverseur du haut produit bien la valeur logique 0 sur la sortie Q. On peut faire exactement le même raisonnement en considérant la valeur logique 1 sur la sortie Q.

Ce dispositif est donc stable et s'auto-entretient. Cependant, le problème est d'initialiser ce dispositif dans un état donné, puis de le faire évoluer.

2.2.2 Bascule RS

La bascule RS est une évolution de la bascule Flip-Flop pour remédier au problème pointé précédemment.

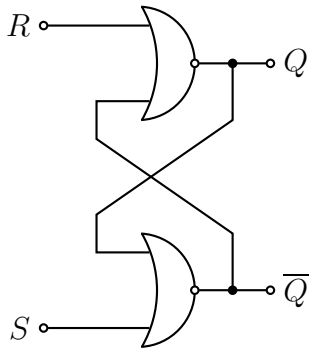


FIGURE 2.4 – Bascule RS – Montage

Deux entrées sont ajoutées : R (pour reset) et S (pour set). Un niveau logique 1 sur l'entrée R positionne la sortie Q à 0. En supposant que S est

également à 0, $\overline{Q}$ passe à 1 et maintient donc Q à la valeur 0 quand R repasse à 0. De la même façon, un niveau logique 1 sur l'entrée S positionne la sortie $\overline{Q}$ à 0 et donc la sortie Q à 1.

La table de vérité de la bascule RS est donc la suivante :

R	S	Q	$\overline{Q}$	
0	0	Q	$\overline{Q}$	Sorties inchangées
1	0	0	1	Reset
0	1	1	0	Set
1	1	0	0	Interdit

FIGURE 2.5 – Bascule RS – Table de vérité

Ce système permet donc d'initialiser la bascule dans un état voulu, puis de la faire évoluer en agissant sur les entrées R et S. On récupère l'état sur la sortie Q et son complément sur la sortie $\overline{Q}$. Il y a cependant une exception : lorsque les entrées R et S sont toutes les deux au niveau 1, les sorties Q et $\overline{Q}$ ne sont plus complémentées. On interdit donc, dans les spécifications sur l'usage de cette bascule, cette configuration sur le entrées.

Une solution pourrait être de cacher la sortie complémentée et de décider que la configuration $R = S = 1$ exécute aussi une remize à zéro de la bascule. En réalité, cette configuration est dangereuse lorsqu'on considère la trnasition de $R = S = 1$ vers $R = S = 0$. En électronique, le temps de propagation des signaux est difficilement maîtrisable et on ne peut absolument pas affirmer que les valeurs sur les entrées R et S changeront d'état simultanément. Fatalement, une des deux entrées basculera vers le niveau 0 avant l'autre, mais on ne peut prédire laquelle. La bascule se retrouvera donc dans un des deux états possibles, sans qu'on puisse le déterminer à l'avance. Cette caractéristique de la bascule RS introduit donc un indéterminisme fâcheux qui motive l'interdiction de cette configuration $R = S = 1$.

Le symbole logique de la bascule RS est :

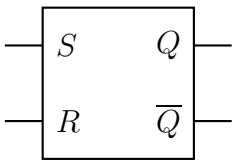


FIGURE 2.6 – Bascule RS – Symbole

On peut aussi réaliser une bascule RS avec des portes NAND, dans ce cas, la configuration interdite produit des niveaux logiques 1 sur les deux sorties Q et $\overline{Q}$.

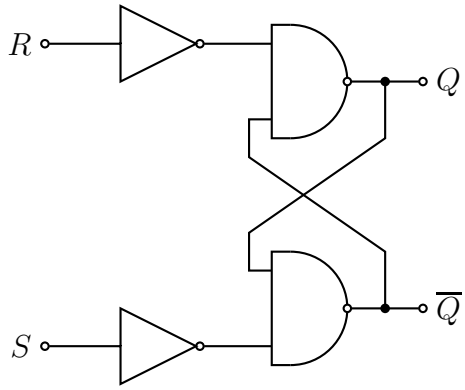


FIGURE 2.7 – Bascule RS – Montage avec des NAND

2.2.3 Bascule RS-T

Cette nouvelle bascule synchronise le changement d'état sur une horloge.

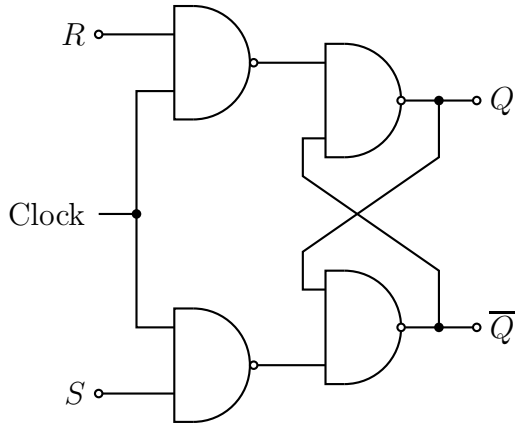


FIGURE 2.8 – Bascule RS-T – Montage

Lorsque le signal d'horloge Clock est au niveau logique 1, le comportement du système est équivalent au schéma précédent : les portes NAND en entrée peuvent être assimilées à des inverseurs et on se retrouve avec une bascule RS standard. Par contre, lorsque Clock est au niveau logique 0, les sorties des deux NAND d'entrées sont bloquées à 1 quel que soit le niveau logique de R et S. L'étage Flip-Flop est alors figé.

L'usage d'un tel dispositif peut d'avérer indispensable dès lors que les entrées R et S d'une bascule RS sont connectées à une logique combinatoire qui, comme nous l'avons déjà vu, peut engendre des états transitoires. Ces états transitoires peuvent alors provoquer des changements d'état non désirés

sur la bascule. L'idée est alors d'effectuer les calculs combinatoires pendant que l'horloge est au niveau logique 0. Lorsque les calculs sont terminées et que les sorties deviennent stables, les résultats peuvent être pris en compte en faisant coïncider le passage au niveau logique 1 de l'horloge.

Notez que ce dispositif ne résout en rien le problème de la configuration $R = S = 1$. Quand l'horloge transite du niveau bas vers le nuveau haut et que les deux entrées R et S sont à 1, les deux portes d'entrées NAND vont logiquement basculer « ensemble » vers un niveau logique bas, Mais, encore une fois, d'un point de vue électronique, la notion de simultanéité n'existe pas et on ne peut pas prédire quelle sera la porte la plus rapide. On aura un état transitoire imprédictible en entrée du Flip-Flop qui peut faire basculer dans un sens ou un autre.

Le symbole logique d'une bascule RS-T est :

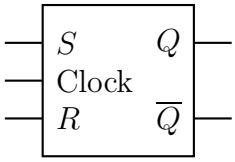


FIGURE 2.9 – Bascule RS-T – Symbole

2.2.4 Bascule JK

La bascule JK lève l'ambiguïté de la bascule RS sur la configuration $R = S = 1$.

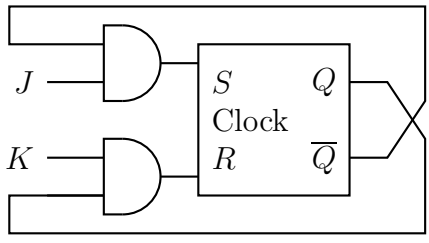


FIGURE 2.10 – Bascule JK – Montage

En entrée de la bascule RS-T sont ajoutées deux portes ET qui augmentent le niveau de rétroaction des sorties sur les entrées. La table de vérité d'un tel système est indiquée ci-dessous.

J	K	Q_n	S	R	Q_{n+1}
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	0	0
0	1	1	0	1	0
1	0	0	1	0	1
1	0	1	0	0	1
1	1	0	1	0	1
1	1	1	0	1	0

FIGURE 2.11 – Bascule JK – Table de vérité

Pour tout n, Q_n indique l'état à l'instant n. On remarque alors que la configuration $R = S = 1$ n'existe plus et qu'on peut simplifier la table de vérité.

J	K	Q_{n+1}	
0	0	Q_n	Sortie inchangée
0	1	0	Remise à zéro
1	0	1	Mise à 1
1	1	$\overline{Q_n}$	Inversion de la sortie

FIGURE 2.12 – Bascule JK – Table simplifiée

Ainsi, la configuration $E = S = 1$ provoque un changement d'état systématique de la bascule.

L'architecture en portes NAND d'une bascule JK est la suivante :

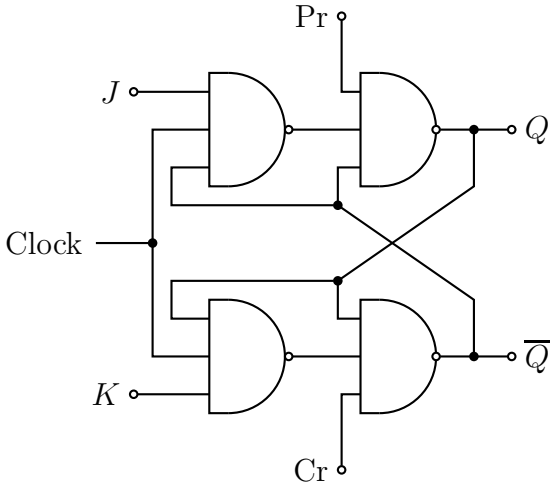


FIGURE 2.13 – Bascule JK – Montage avec des NAND

Deux commandes ont été rajoutées pour initialiser la bascule dans un état connu, les entrées Pr et Cr. Elles sont actives lorsqu'elles sont au niveau logique 0. Il est en effet extrêmement important qu'à la mise sous tension, ou sous l'action d'une procédure de réinitialisation, un système numérique soit dans un état bien défini et cohérent. Ces deux commandes sont donc maintenues à un niveau logique 1 prendant le fonctionnement normal d'un système et activées seulement dans les phases d'initialisation. Soit Pr sera maintenu temporairement à zéro pour initialiser la bascule à 1, soit Cr sera maintenu temporairement à zéro pour initialiser la bascule à 0. La configuration Pr = Cr = 0 est interdite et n'a pas de sens (la bascule ne peut être initialisée à 0 et 1 simultanément).

Le symbole logique d'une bascule JK est :

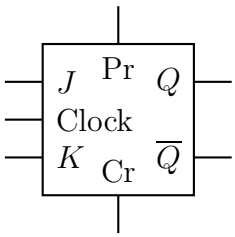


FIGURE 2.14 – Bascule JK – Symbole

Le comportement de la bascule JK n'est cependant pas encore exempt de tous défauts. Il existe une configuration qui engendre une instabilité se traduisant par une oscillation du système. Ce disfonctionnement se produit quand $J = K = 1$ et quand l'horloge transite du niveau 0 vers le niveau 1. En omettant les commandes Pr et Cr, on se retrouve avec un circuit équivalent à :

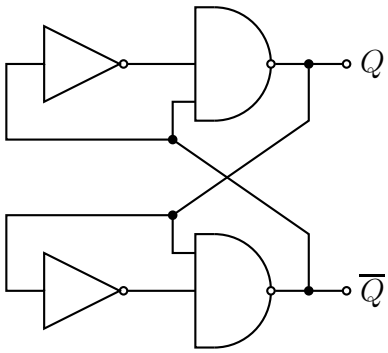


FIGURE 2.15 – Bascule JK – Montage équivalent quand $J = K = 1$

Dans ce cas, les portes NAND se comportent comme des inverseurs et on obtient alors sun circuit instable.

2.2.5 Bascule JK maître/esclave

La bascule JK maître/esclave pallie le défaut de la bascule JK en mettant en série deux bascules commandées sur des niveaux d'horloge différents. On évite ainsi que la rétroaction des sorties sur les entrées conduise à un comportement instable.

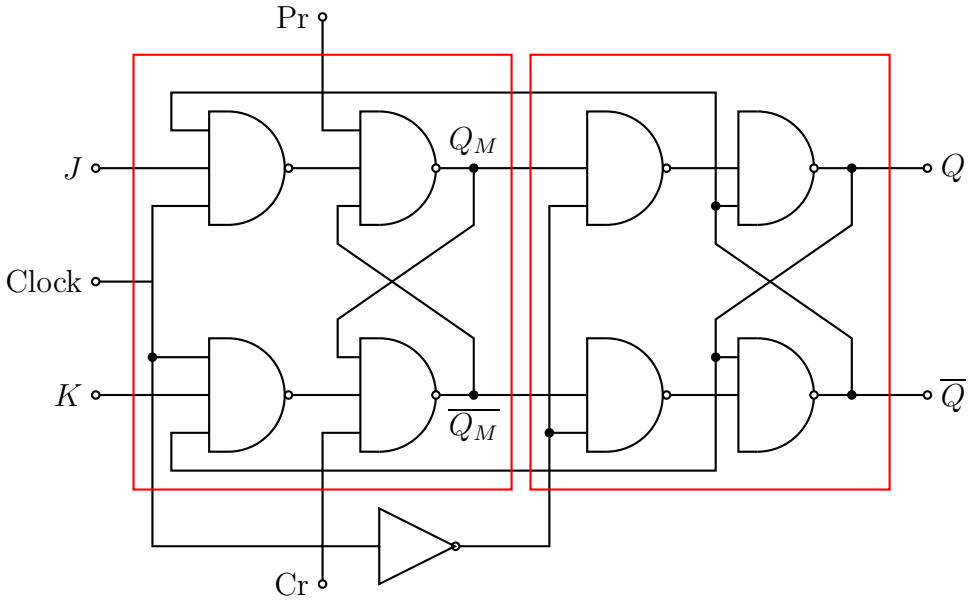


FIGURE 2.16 – Bascule JK maître/esclave

Lorsque l'horloge est au niveau logique 0, la première bascule est bloquée. Les sorties Q_M et $\overline{Q_M}$ sont donc dans un état stable est cohérent. Le second étage peut alors changer d'état sans être perturbé. Lorsque l'horloge est au niveau logique 1, le second étage est bloqué. Les sorties Q et $\overline{Q}$ ne peuvent donc pas changer et procurent des niveaux logiques stables sur l'étage d'entrée.

Lorsqu'on parle de bascule JK, c'est en général à cette dernière qu'on fait référence.

2.2.6 Bascule D

Une bascule D est une bascule JK sur laquelle les entrées J et K sont inversées.

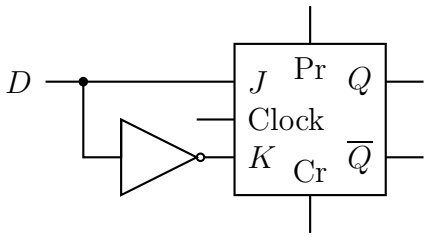


FIGURE 2.17 – Bascule D – Montage

Le comportement de la bascule D est immédiatement déduit de la table de vérité de la bascule JK :

- Si $D = 1$ alors $J = 1$ et $K = 0$ donc $Q_{n+1} = 1$
- Si $D = 0$ alors $J = 0$ et $K = 1$ donc $Q_{n+1} = 0$

La bascule D agit donc comme une unité de retard par rapport au signal d'horloge. Le symbole logique d'une bascule D est :

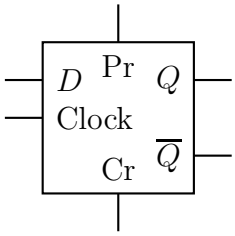


FIGURE 2.18 – Bascule D – Symbole

2.2.7 Bascule T

Une bascule T est une bascule JK dont les deux entrées sont connectées ensemble.

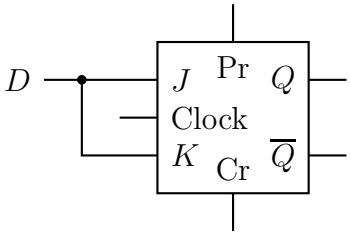


FIGURE 2.19 – Bascule T – Montage

Tout comme la bascule D, son comportement peut être déduit de la table de vérité de la bascule JK.

- Si $T = 1$ alors $J = K = 1$ donc $Q_{n+1} = \overline{Q_n}$
- Si $T = 0$ alors $J = K = 0$ donc $Q_{n+1} = Q_n$

Cette particularité est utilisée, comme on le verra par la suite, pour réaliser des compteurs asynchrones. Le symbole logique de la bascule T est :

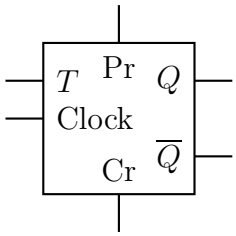


FIGURE 2.20 – Bascule T – Symbole

2.3 Registres

2.3.1 Registres 1 bit

Les registres sont des composants principalement réalisés à partir de bascules D. Au niveau de la conception d'un système numérique, ils apparaissent comme les entités élémentaires pour mémoriser des valeurs logiques. Le représentation simplifié d'un registre 1 bit est la suivante :

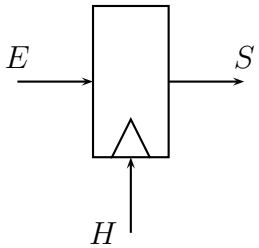


FIGURE 2.21 – Registre 1 bit

Le comportement est celui du chronogramme donné en introduction de ce chapitre (figure 2.1). La sortie S est synchronisée sur un des deux fronts de l'horloge. Quand rien n'est spécifié, on suppose que c'est le front montant. Le registre peut, éventuellement, être complété par les commandes Set et Reset pour le positionner dans un état connu lors de l'initialisation du système. À partir de cette cellule de base, des composants de mémorisation plus complexes sont élaborés.

2.3.2 Registres N bits

C'est une extension du registre 1 bit pour mémoriser simultanément plusieurs bits. N registres sont connectés en parallèle et commandés par la même horloge.

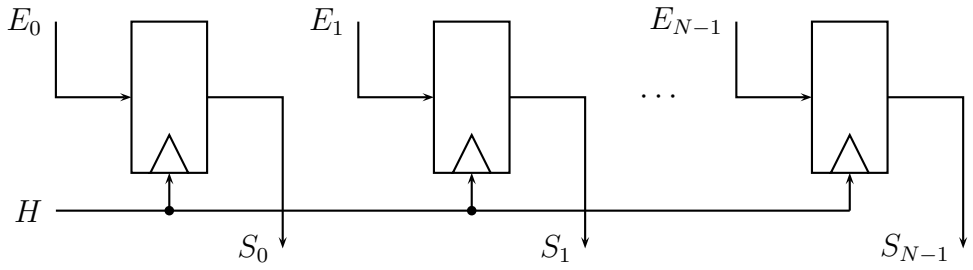


FIGURE 2.22 – Registre N bits – Montage

La représentation schématique est identique au registre 1 bit. Elle ajoute simplement l'information du nombre de bits sur les entrées et les sorties.

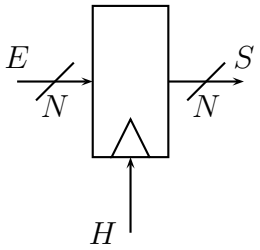


FIGURE 2.23 – Registre N bits – Symbole

Au front d'horloge montant, les N bits présent sur l'entrée E seront mémorisés dans le registre et disponibles en sortie S quelques instants plus tard, le temps que l'étage de sortie du Flip-Flop bascule.

2.3.3 Registre à décalage

Dans ce dispositif, les registres sont connectés en série de telle manière que la sortie i de l'étage i est connectée à l'entrée $i + 1$ de l'étage $i + 1$.

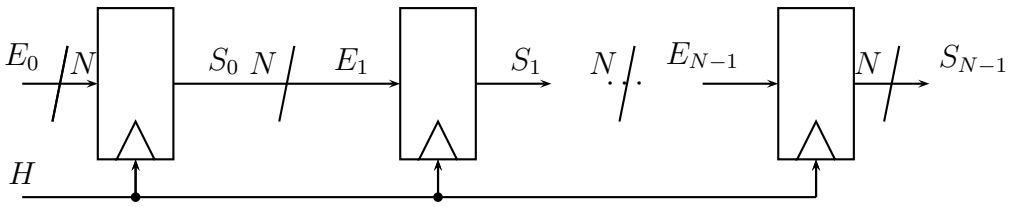


FIGURE 2.24 – Registre à décalage – Montage

Les valeurs présentes sur l'entrée E_0 vont progresser au rythme de l'horloge dans le réseau de registres. Si on suppose, par exemple, que les valeurs de la suite $(1, 2, 3, \ldots)$ se succèdent sur l'entrée E_0 à chaque période d'horloge, on obtient le chronogramme suivant avec un registre à décalage à 4 étages.

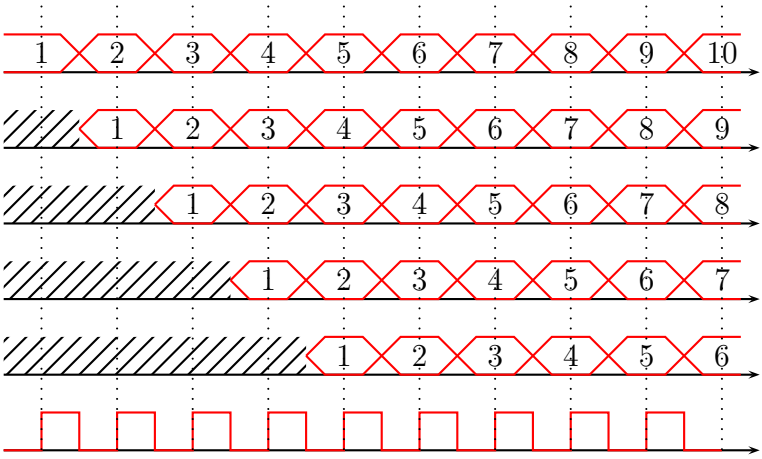


FIGURE 2.25 – Chronogramme d'un registre à décalage de 4 étages

Les valeurs hachurées sur les sorties S_0 à S_4 indiquent qu'on ne connaît pas encore leur contenu. Leur valeur est indéterminée. Dans cet exemple, il faut attendre 4 coups d'horloge pour que l'ensemble soit complètement initialisé. Remarquez qu'entre le front montant de l'horloge et l'établissement d'une valeur sur les sorties, il s'écoule un certain temps symbolisant le temps de réponse des registres.

2.4 Compteurs

Un compteur est un dispositif qui, au rythme d'une horloge, fait progresser une valeur sur ses sorties. La valeur est codée sur N bits en binaire. Par exemple, un codage binaire sur 3 bits donne :

Valeur décimale	Valeur binaire
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

FIGURE 2.26 – Codage binaire sur 3 bits

Un compteur sur 3 bits délivrera donc indéfiniment la suite (000, 001, 010, 011, 100, 101, 110, 111, 000, m...). Plus généralement, un compteur sur N bits génèrera la suite $(0, 1, \ldots, 2^{N-1}, 0, 1, \ldots)$.

2.4.1 Compteur asynchrone

Ce type de compteur est réalisé avec des bascules T dont la table de vérité est donnée par :

- Si $T = 1$ alors $Q_{n+1} = \overline{Q_n}$
- Si $T = 0$ alors $Q_{n+1} = Q_n$

Le compteur connecte des bascules T en série de la manière suivante :

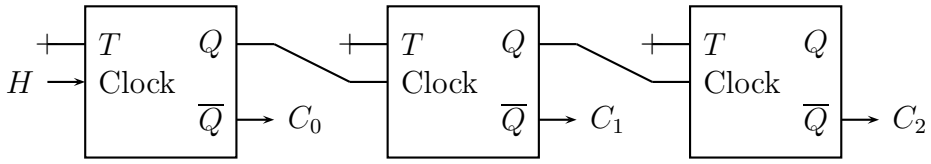


FIGURE 2.27 – Compteur asynchrone – Montage

L'horloge du système est connectée à l'entrée horloge de la première bascule. En suite, l'horloge s'une bascule i est connectée à la sortie de la bascule $i - 1$. La sortie du compteur est prise sur les sorties $\overline{Q}$ des bascules. Les entrées T des bascules sont toutes connectées au niveau logique 1 (inversion systématique de l'état à chaque top d'horloge).

Si on suppose, à un instant donné, que l'état du compteur est $C_0 = C_1 = C_2 = 0$, les états de sortie Q suivants seront ceux donnés par le chronogramme suivant :

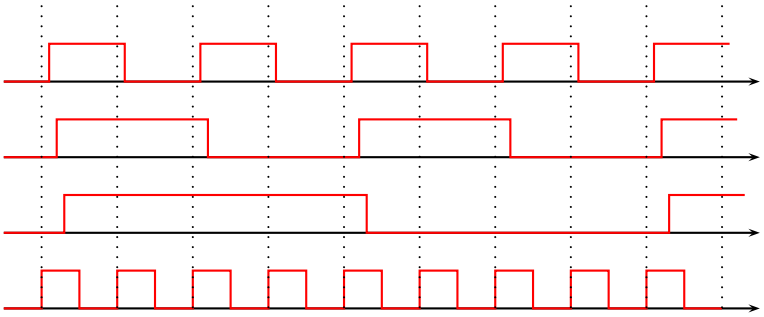


FIGURE 2.28 – Chronogramme d'un compteur asynchrone

La première bascule change d'état à chaque front montant de l'horloge, ce qui a pour effet de générer sur Q_0 un signal périodique de fréquence deux fois plus faible que l'horloge. Ce signal étant appliqué sur l'entrée horloge de la seconde bascule, icelle le divise également par deux et ainsi de suite. Les valeurs recueillies sur les sorties Q correspondent à la valeur complémentée de la sortie du compteur.

Ce compteur est simple à mettre en oeuvre mais présente l'inconvénient d'avoir un temps de réponse qui croît linéairement avec le nombre de bascule. En effet, le changement d'état de la dernière bascule est déterminé séquentiellement par les changements successifs de toutes les bascules précédentes. Le chronogramme ci-dessus montre bien les délais de plus en plus importants sur les bits de poids fort des sorties entre le front montant de l'horloge et l'établissement de la valeur finale.

2.4.2 Compteur synchrone

Le compteur synchrone corrige le problème du compteur précédent et produit une valeur de sortie synchrone avec l'horloge. L'idée est de détecter à l'avance les changements d'états. On peut remarquer que la valeur Q_i change d'état lorsque toutes les valeurs Q_j pour $j < i$ sont au niveau logique 1. Par exemple, pour la valeur Q_2, icelle s'inversera à chaque fois que $Q_0 = Q_1 = 1$.

Q_0	Q_1	Q_2	
0	0	0	
1	0	0	
0	1	0	
1	1	0	Changement Q_2
0	0	1	
1	0	1	
0	1	1	
1	1	1	Changement Q_2

FIGURE 2.29 – Changement d'état de Q_2

Ainsi, pour effectuer un changement d'état au temps t, il faut évaluer à l'instant $t-1$ les sorties des bascules T. La bascule d'ordre i changera d'état à l'instant t si les bascules d'ordre 0 à $i-1$ sont toutes au niveau logique 1 à l'instant $t-1$. D'un point de vue pratique, il faut réaliser un ET logique sur les sorties des bascules j avec $j < i$ et propager le résultat sur l'entrée T de la bascule i.

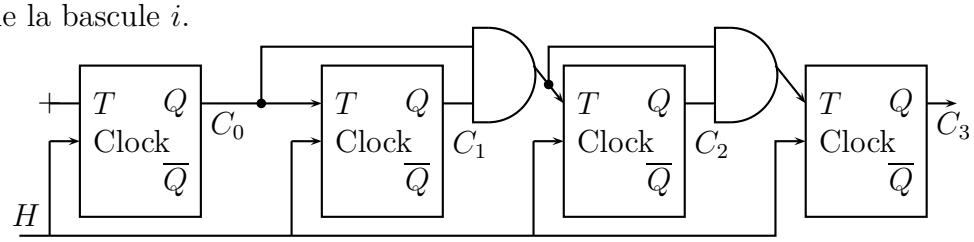


FIGURE 2.30 – Compteur synchrone – Montage

Le schéma ci-dessus décrit une implémentation d'un compteur synchrone. Icelui est complètement synchronisé sur l'horloge. À chaque front montant de ce signal, les sorties basculent toutes en même temps. Entre deux fronts d'horloge, les entrées T sont évaluées à travers la cascade de ET logiques.

Chapitre 3

Automates

3.1 Introduction

Ce chapitre aborde la mise en œuvre pratique d'automates dans les circuits numériques. Ces éléments interviennent dès qu'une suite d'opérations doit s'effectuer suivant un déroulement prédéterminé, mais conditionné par l'environnement. Un automate est un dispositif qui possède N entrées reliées aux évènements externes et P sorties qui, en fonction de son état, agissent sur l'environnement. L'ensemble est synchronisé par une horloge. Ainsi ; à chaque pas de temps, l'automate évolue (ou non) vers un nouvel état.

Pour illustrer notre propos, on va raisonner sur l'exemple d'un distributeur de chewing-gums simple qui délivre uniquement deux sortes de produits : des chewing-gums à la menthe ou à la fraise, les deux à un tarif unique de 1 euro. Ce distributeur est commandé par un contrôleur (un automate) qui va délivrer (ou non) les chewing-gums en fonction d'évènements tels que la réception d'une pièce de 1 euro, la pression du bouton fraise ou du bouton menthe et la disponibilité de chewing-gums dans la réserve du distributeur.

Dans ce dispositif, l'automate possède 6 entrées connectées à divers capteurs du distributeur :

- `piece` : vrai si une pièce a été introduite
- `1euro` : vrai si la pièce introduite est une pièce de 1 euro
- `fraise` : vrai si la sélection est fraise
- `menthe` : vrai si la sélection est menthe
- `vide_fraise` : vrai si a réserve de chewing-gum fraise est vide
- `vide_menthe` : vrai si a réserve de chewing-gum menthe est vide

L'automate positionne également des signaux de sortie pour agir sur l'environnement :

- `rendre_pièce` : pour rendre la pièce introduite

- `donne_fraise` : pour délivrer un chewing-gum à la fraise
- `donne_menthe` : pour délivrer un chewing-gum à la menthe

À partir de ces signaux d'entrées/sorties, le comportement du distributeur peut être spécifié par l'automate suivant :

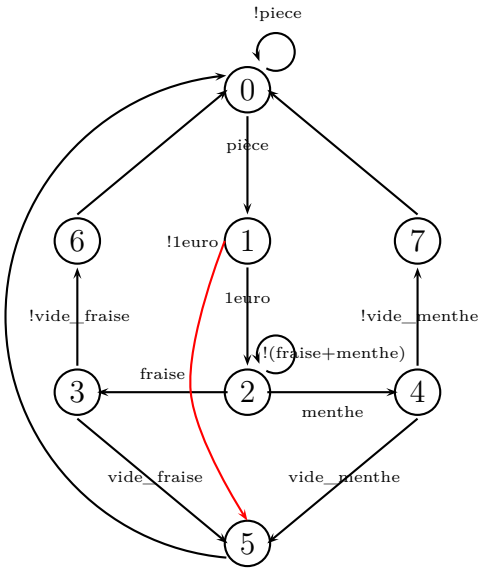


FIGURE 3.1 – Automate

Les états de l'automate sont représentés par des cercles et les transitions par les flèches étiquetées avec une condition. On associe une action à certains états :

États	Actions
0	
1	
2	
3	
4	
5	Rendre la pièce
6	Délivrer un chewing-gum Fraise
7	Délivrer un chewing-gum Menthe

L'état 0 correspond à un état de repos. L'automate attends qu'une pièce soit insérée dans le distributeur. Tant que cette condition n'est pas réalisée, l'automate reste dans cet état. L'insertion d'une pièce le fait passer à l'état 1 qui en accuse réception. Si la pièce correspond à 1 euro, l'automate transite vers l'état 2. Si ce n'est pas le cas, il bascule vers l'état 5. À partir de l'état 2, deux solutions sont possibles en fonction de la sélection effectuée : le choix

fraise conduit vers l'état 3 et le choix menthe vers l'état 4. Mais on reste dans l'état 2 tant qu'une sélection n'a pas été faite. Le distributeur vérifie ensuite que ses réserves lui permettent de délivrer les produits demandés. Si la réserve du chewing-gum sélectionnée est vide alors l'automate aiguille vers l'état 5 dont l'acction est de rendre la pièce. Dans le cas contraire, les états 6 et 7 s'occupent de délivrer un chewing-gum fraise ou menthe.

Plusieurs implémentations matérielles de cet automate sont possibles. La plus immédiate consiste à affecter un état à une registre qui prendra la valeur 0 si l'état est inactf et 1 dans le cas contraire. Dans l'exemple, il faudra 8 registres 1 bit si l'automate possède 8 états. Une autre possibilité est de coder les numéros des états sur un nombre minimal de bits. La taille du registre pour numéroter cette valeur sera plus petite. Dans l'exemple, un registre de 3 bit suffira pour coder les états. Entre ces deux extrêmes, tout est possible. Le choix fépendra de la taille et de la complexité de l'automate. Un automate qui recense mille états gagnera à être compacté sur dix bits (voire un peu plus) plutôt que d'être réalisé sur la base d'un bit par état.

Les deux sections suivantes proposent les deux mises en œuvre évoquées sur l'exemple du distributeur de chewing-gums. La première est appelée encodage linéaire et affecte un bit par état. La seconde code de manière minimale l'état de l'automate.

3.2 Encodage linéaire

Dans ce schéma, le codage des états de l'automate et leur mémorisation dans des registres est immédiat :

états	R_0	R_1	R_2	R_3	R_4	R_5	R_6	R_7
0	1	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0
2	0	0	1	0	0	0	0	0
3	0	0	0	1	0	0	0	0
4	0	0	0	0	1	0	0	0
5	0	0	0	0	0	1	0	0
6	0	0	0	0	0	0	1	0
7	0	0	0	0	0	0	0	1

FIGURE 3.2 – États des registres – première architecture

Il faut maintenant préciser les fonctions de transition pour passer d'un état à un autre. Transiter de l'état 0 vers l'état 1 implique que la condition

`pièce` soit vraie. Donc pour arriver dans l'état 1, il faut que deux conditions soient vraies : être dans l'état de 0 (donc que R_0 soit positionné à 1) et avoir reçu la pièce.

On en déduit que l'équation logique correspondant à la transition R_0 vers R_1 : $R_{1,t} = R_{0,t-1}\texttt{pièce}$. Plus exactement, cela signifie qu'au prochain coup d'horloge (au temps t), l'automate évoluera vers l'état 1 si et seulement si au temps $t-1$ la condition `pièce` est vraie et l'automate est dans l'état 0.

De la même manière, pour aller vers l'état 2, il faut être dans l'état 1 et il faut que la condition `1euro` soit vraie, ce qui correspond à $R_{2,t} = R_{1,t-1}\texttt{1euro}$. Mais il faut également rester dans cet état tant que la sélection n'a pas été effectuée. Il y a donc une autre alternative pour atteindre R_2 qui est $R_{2,t} = R_{2,t-1}\overline{\texttt{fraise} + \texttt{menthe}}$. L'équation logique des transitions pour atteindre R_2 sera $R_{2,t} = (R_{1,t-1}\texttt{1euro}) + (R_{2,t-1}\overline{\texttt{fraise} + \texttt{menthe}})$.

L'ensemble des équations logiques correspondant aux fonctions de transitions s'écrivent :

- $R_{0,t} = (R_{0,t-1}\overline{\texttt{piece}}) + R_{5,t-1} + R_{6,t-1} + R_{7,t-1}$
- $R_{1,t} = (R_{0,t-1}\texttt{piece})$
- $R_{2,t} = (R_{1,t-1}\texttt{1euro}) + (R_{2,t-1}\overline{\texttt{fraise} + \texttt{menthe}})$
- $R_{3,t} = (R_{2,t-1}\texttt{fraise})$
- $R_{4,t} = (R_{2,t-1}\texttt{menthe})$
- $R_{5,t} = (R_{1,t-1}\overline{\texttt{1euro}}) + (R_{3,t-1}\texttt{vide\_fraise}) + (R_{4,t-1}\texttt{vide\_menthe})$
- $R_{6,t} = (R_{3,t-1}\overline{\texttt{vide\_fraise}})$
- $R_{7,t} = (R_{4,t-1}\overline{\texttt{vide\_menthe}})$

L'implémentation est alors immédiate. L'entrée d'un registre est alimentée par la fonction de transition correspondants, comme le montre le schéma suivant :

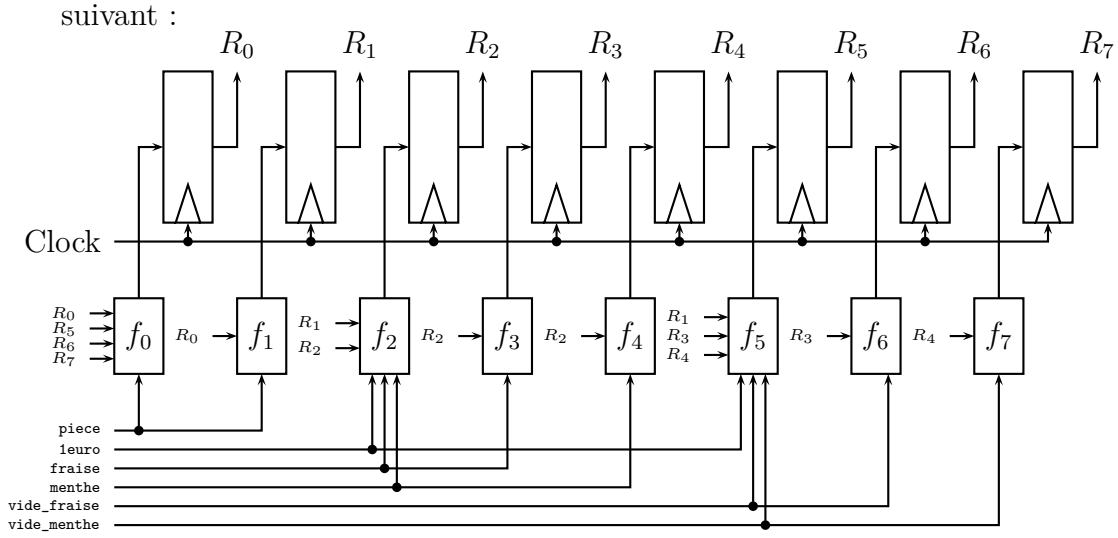


FIGURE 3.3 – Montage de l'automate

Les composants f_0 à f_7 encapsulent les fonctions de transition des états 0 à 7. Par exemple la fonction f_0 calcule la fonction logique $R_0\overline{\texttt{pièce}} + R_5 + R_6 + R_7$.

Les trois signaux `rendre_pièce`, `donner_fraise` et `donner_menthe` sont directement capturés sur les sorties des registres R_5, R_6 et R_7.

3.3 Encodage minimal

Nous choisissont maintenant de coder les états de l'automate sur 3 bits, mémorisés dans trois registres R_0, R_1 et R_2 selon le schéma suivant :

états	R_0	R_1	R_2
0	0	0	0
1	1	0	0
2	0	1	0
3	1	1	0
4	0	0	1
5	1	0	1
6	0	1	1
7	1	1	1

FIGURE 3.4 – États des registres – encodage minimal

Ce codage est complètement arbitraire. Tout autre codage est possible du moment que pour un état donnée, il existe une unique configuration (R_0, R_1, R_2). On peut maintenant construire la table de vérité qui prend en compte les signaux d'entrée et l'état de l'automate au temps $t-1$, et qui indique en sortie l'état de l'automate au temps t. Il y a 6 entrées et 3 bits pour coder l'état, soit 2^9 configurations possibles. Pour condenser la table de vérité, on ne peut reporter que les évènements intéressants, ie ceux qui font évoluer l'automate en fonction d'un état particulier de l'automate. Par exemple, le passage vers l'état 1 est seulement conditionné par l'acquisition d'une pièce. Les autres signaux d'entrée ne sont pas pris en considération. Ils sont alors notés x dans la table de vérité. Par exemple, la cinquième ligne de la table de vérité ci-dessous précise que le passage de l'état 0 (au temps $t-1$) à l'état 1 (au temps t) s'effectue quand le signal `pièce` est à 1 quelles que soient les valeurs des autres signaux.

pièce	1e	f	m	v_f	v_m	$R_{0,t-1}$	$R_{1,t-1}$	$R_{2,t-1}$	$R_{0,t}$	$R_{1,t}$	$R_{2,t}$
0	x	x	x	x	x	0	0	0	0	0	0
x	x	x	x	x	x	1	0	1	0	0	0
x	x	x	x	x	x	1	1	0	0	0	0
x	x	x	x	x	x	1	1	1	0	0	0
1	x	x	x	x	x	0	0	0	1	0	0
x	1	x	x	x	x	1	0	0	0	1	0
x	x	0	0	x	x	0	1	0	0	1	0
x	x	1	x	x	x	0	1	0	1	1	0
x	x	x	1	x	x	0	1	0	0	0	1
x	0	x	x	x	x	1	0	0	1	0	1
x	x	x	x	1	x	1	1	0	1	0	1
x	x	x	x	x	1	0	0	1	1	0	1
x	x	x	x	0	x	1	1	0	0	1	1
x	x	x	x	x	0	0	0	1	1	1	1

FIGURE 3.5 – Table de vérité – Encodage minimal

À partir de cette table de vérité, on déduit les valeurs de $R_{0,t}$, $R_{1,t}$ et $R_{2,t}$:

$$\begin{aligned}
R_{0,t} = {} & (\overline{R_0} \cdot \overline{R_1} \cdot \overline{R_2}\texttt{pièce}) + (\overline{R_0}R_1\overline{R_2}\texttt{fraise}) + (R_0\overline{R_1} \cdot \overline{R_2} \cdot \overline{\texttt{1euro}}) \\
& + (R_0R_1\overline{R_2}\texttt{vide_fraise}) + (\overline{R_0} \cdot \overline{R_1}R_2\texttt{vide_menthe}) \\
& + (\overline{R_0} \cdot \overline{R_1}R_2\overline{\texttt{vide_menthe}}) \\
R_{1,t} = {} & (R_0\overline{R_1} \cdot \overline{R_2}\texttt{1euro}) + (\overline{R_0}R_1\overline{R_2} \cdot \overline{\texttt{fraise}} \cdot \overline{\texttt{menthe}})) \\
& + (R_0R_1\overline{R_2} \cdot \overline{\texttt{vide_fraise}}) + (\overline{R_0} \cdot \overline{R_1}R_2\overline{\texttt{vide_menthe}}) \\
& + (\overline{R_0}R_1\overline{R_2}\texttt{fraise}) \\
R_{2,t} = {} & (\overline{R_0}R_1\overline{R_2}\texttt{menthe}) + (R_0\overline{R_1} \cdot \overline{R_2} \cdot \overline{\texttt{1euro}}) + (R_0R_1\overline{R_2}\texttt{vide_fraise}) \\
& + (\overline{R_0} \cdot \overline{R_1}R_2\texttt{vide_menthe}) + (R_0R_1\overline{R_2} \cdot \overline{\texttt{vide_fraise}}) \\
& + (\overline{R_0} \cdot \overline{R_1}R_2\overline{\texttt{vide_menthe}})
\end{aligned}$$

où on a noté R_i pour $R_{i,t-1}$.

Les valeurs de `rendre_pièce`, `donner_fraise` et `donner_menthe` valides respectivement aux états 5, 6 et 7 sont calculées à partir des sorties des registres $R_{0,t}$, $R_{1,t}$ et $R_{2,t}$:

$$\begin{aligned}
\texttt{rendre_pièce} &= R_{0,t}\overline{R_{1,t}}R_{2,t} \\
\texttt{donner_fraise} &= \overline{R_{0,t}}R_{1,t}R_{2,t} \\
\texttt{donner_menthe} &= R_{0,t}R_{1,t}R_{2,t}
\end{aligned}$$

Le schéma électrique correspondant est le suivant :

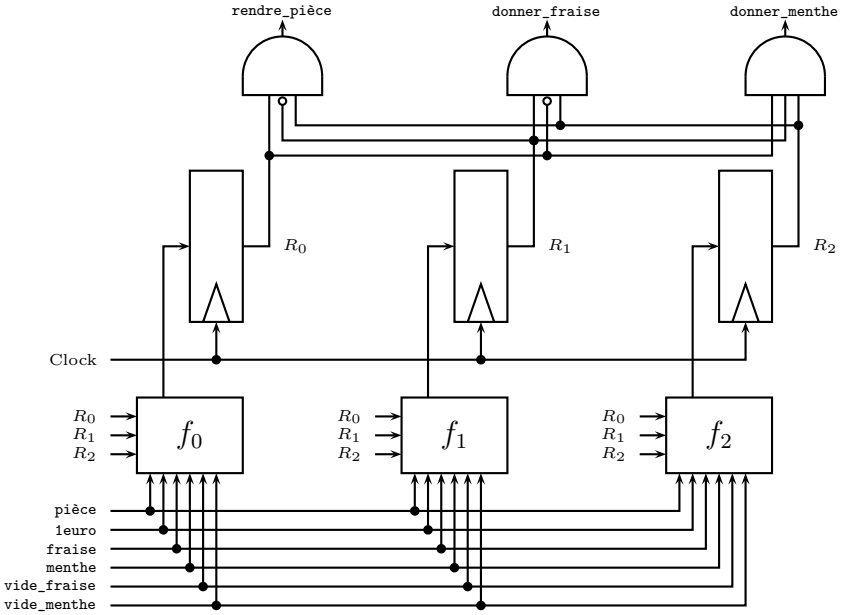


FIGURE 3.6 – Montage de l'automate – encodage minimal

Les modules f_0, f_1 et f_2 calculent les fonctions de transition de l'automate. Par rapport à un encodage linéaire, ces fonctions, même si elles sont moins nombreuses, sont plus complexes. En termes de nombre de portes logiques pour l'ensemble des fonctions, l'encodage linéaire est plus économe. Par contre, ici, le nombre de registres est réduit. Il n'est donc pas évident a priori d'estimer, pour cet exemple, quel est le meilleur codage.

3.4 Automates de More et de Mealy

Quel que soit leur encodage, les automates peuvent êtres classés en deux grandes catégories : les automates de Moer et les automates de Mealy.

3.4.1 Automate de More

Cet automate se compose :

- d'un registre (plusieurs bits) qui mémorise l'état de l'automate et qui est cadencé par une horloge
- d'une fonction de transition T qui calcule le prochain état en fonction des évènements extérieurs et de l'état de l'automate
- d'une fonction de génération G qui, en fonction de l'état de l'automate, produit les signaux de sortie

Les fonctions T et G sont purement combinatoires.

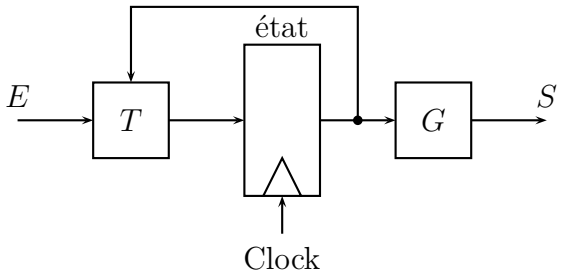


FIGURE 3.7 – Automate de More

Dans ce modèle, les sorties sont uniquement fonction de l'état de l'automate et donc synchrone avec l'horloge. L'automate du distributeur de chewing-gums réalisé précédemment appartient à cette catégorie.

3.4.2 Automate de Mealy

Cet automate se compose :

- d'un registre (plusieurs bits) qui mémorise l'état de l'automate et qui est cadencé par une horloge
- d'une fonction de transition T qui calcule le prochain état en fonction des évènements extérieurs et de l'état de l'automate
- d'une fonction de génération G qui, en fonction de l'état de l'automate **et des entrées**, produit les signaux de sortie

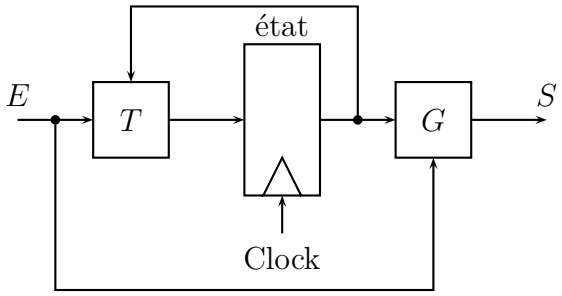


FIGURE 3.8 – Automate de More

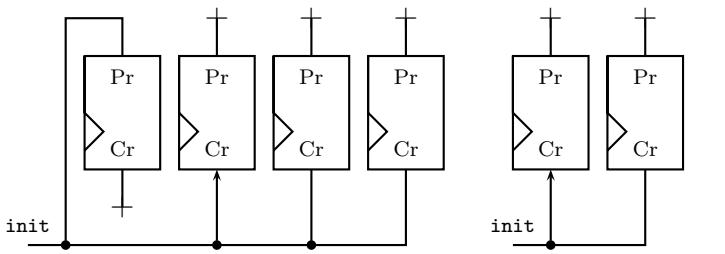
Dans ce modèle, les sorties ne sont plus synchronisées sur l'horloge. Dès qu'une entrée change d'état, les sorties peuvent être modifiées.

3.5 Initialisation

Lors de la mise sous tension d'un système numérique à base de logique séquentielle, les états des registres sont indéterminés. Plus exactement, ils

sont dans un état (0 ou 1) qu'on ne peut pas prédire à l'avance. Dans le cas de l'encodage linéaire (1 bit par registre), on peut alors avoir plusieurs états à 1, ce qui ne correspond à aucune spécification. Une autre possibilité est d'avoir tous les registres à 0, ce qui est également hors spécification. Dans le cas de l'encodage minimal, on peut également être dans un état inconnu. Par exemple, un automate à 9 états, codé au minimum sur 4 bits (soit 16 états possibles), peut se retrouver dans un état non prévu.

Il faut donc systématiquement ajouter une commande d'initialisation qui positionne l'automate dans un état bien déterminé à partir duquel il deviendra opérationnel. Si on dispose de registres avec des entrées Pr (preset) et Cr (clear), on peut agir directement sir l'initialisation du système comme le montre le schéma ci-dessous :



(a) Un bit par état (b) Encodage minimal

FIGURE 3.9 – Initialisation

Seuls les registres sont représentés. Les commandes Pr et Cr sont actives au niveau bas. Le schéma (a) représente un encodage linéaire pour un automate à 4 états. La valeur logique 0 sur l'entrée `init` positionne le premier registre à 1 et les trois autres à 0. Le schéma (b) représente un encodage minimal sur 2 bits. Dans ce cas une valeur logique 0 sur l'entrée `init` positionne les deux registres à 0 (on supposera ici que l'état initial correspond à cet état).

On peut également considérer que l'entrée `init` est un signal comme un autre et l'intégrer comme condition dans l'automate. Les fonctions de transition doivent donc considérer la valeur de ce signal. Par exemple, les équations logiques de l'automate du distributeur correspondant aux fonctions de transitions doivent être modifiées de la manière suivante si on spécifie que la valeur logique 1 sur l'entrée init doit remettre le système à zéro :

- $R_{0,t} = (R_{0,t-1}\overline{\texttt{piece}}) + R_{5,t-1} + R_{6,t-1} + R_{7,t-1} + \texttt{init}$
- $R_{1,t} = (R_{0,t-1}\texttt{piece})\overline{\texttt{init}}$
- $R_{2,t} = ((R_{1,t-1}\texttt{1euro}) + (R_{2,t-1}\overline{\texttt{fraise} + \texttt{menthe}}))\overline{\texttt{init}}$

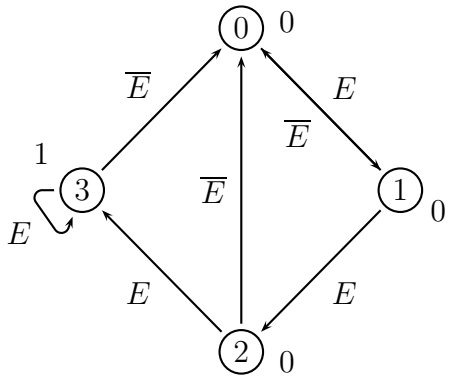
- $R_{3,t} = (R_{2,t-1}\texttt{fraise})\overline{\texttt{init}}$
- $R_{4,t} = (R_{2,t-1}\texttt{menthe})\overline{\texttt{init}}$
- $R_{5,t} = ((R_{1,t-1}\overline{\texttt{1euro}}) + (R_{3,t-1}\texttt{v\_f}) + (R_{4,t-1}\texttt{v\_m}))\overline{\texttt{init}}$
- $R_{6,t} = (R_{3,t-1}\overline{\texttt{vide\_fraise}})\overline{\texttt{init}}$
- $R_{7,t} = (R_{4,t-1}\overline{\texttt{vide\_menthe}})\overline{\texttt{init}}$

Dès lors que des registres avec des entrées Pr et Cr sont disponibles, la mise en œuvre de l'initialisation d'un automate est simplifiée. Cette solution est donc préférable.

3.6 Codage avec une look-up table

Une look-up table (ou en abrégé LUT) est une structure de données qui permet l'association de couples de valeurs. Son usage permet de simplifier (voire systématiser) la conception des automates, principalement au niveau du calcul des fonctions de transitions dans le cas d'un encodage minimal. En pratique, une LUT est une mémoire dont les adresses sont connectées à la fois aux signaux d'entrées et aux états stables de 'automate. La sortie de la mémoire délivre l'état suivant.

Pour illustrer notre propos, considérons un dispositif matériel à une entrée (1 bit) et une sortie (1 bit). L'entrée reçoit, au rythme de l'horloge, une série de valeurs booléennes. La sortie passe à 1 quand l'entrée reçoit trois uns consécutifs. L'automate qui spécifie le comportement de ce système est :



L'automate a quatre états, on peut donc les coder sur 2 bits (00 pour l'état 0, 01 pour 1, 10 pour 2 et 11 pour 3). On en déduit la table de vérité suivante :

$R_{1,t-1}$	$R_{0,t-1}$	E	$R_{1,t}$	$R_{0,t}$	S
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	0
0	1	1	1	0	0
1	0	0	0	0	0
1	0	1	1	1	1
1	1	0	0	0	0
1	1	1	1	1	1

FIGURE 3.10 – Table de vérité – Automate à LUT

Cette table correspond au contenu de la LUT adressée par $R_{1,t-1}$, $R_{0,t-1}$ et E. Cette LUT sera une mémoire de 8 mots de 3 bits. Le nombre de mots est dicté par le nombre d'entrées (ici 1) et le nombre de bits pour coder les états (ici 2). Il faut donc 2^{1+2} mots pour mémoriser l'ensemble des configurations. La taille d'un mot mémoire correspond au nombre de bits pour coder les états et le nombre de sorties.

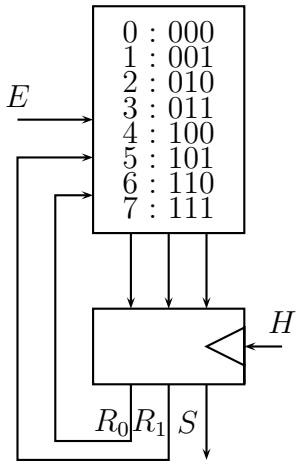


FIGURE 3.11 – Montage d'un automate à LUT

L'initialisation de ce système se fait simplement en agissant sue les commandes d'initialisation du registre.

La mise en œuvre d'un automate avec une LUT est très souple et permet facilement de modifier son comportement : il faut juste reconfigurer la LUT (la mémoire) avec de nouvelles valeurs.

Chapitre 4

Fonctionnement du microprocesseur

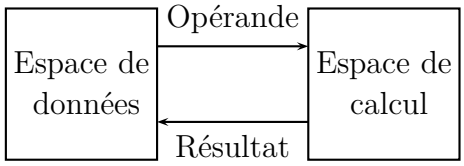


FIGURE 4.1 – Microprocesseur

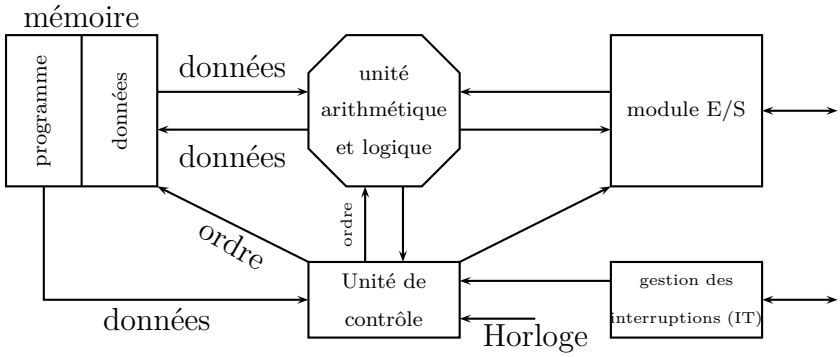


FIGURE 4.2 – Architecture Von Neumann

Deux types d'instructions :

- calcul : $1 + 1$ par exemple.
- contrôle : lance un test pour prendre une décision

– Entrée/Sortie : acquérir une information extérieure

4.1 Organisation de la mémoire

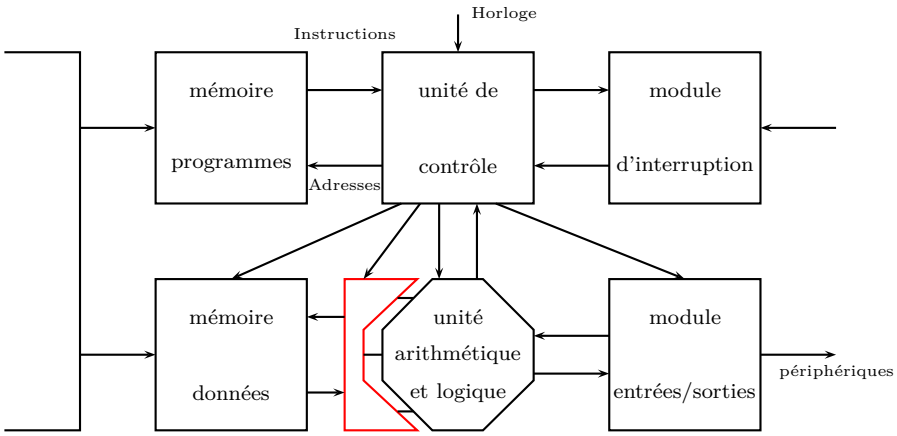


FIGURE 4.3 – Organisation de la mémoire

Le composant rouge est composé de quelques dizaines de registres, appelé le banc registre.

Plus la mémoire est grande, plus elle est lente. On crée donc une hiérarchie de la mémoire (ici deux niveaux : la mémoire et les bancs registres). La vitesse maximale de l'horloge est donc décidée par les bancs registres.

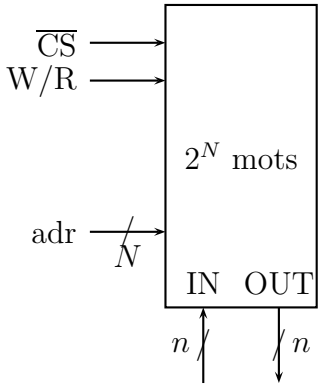


FIGURE 4.4 – Représentation de la mémoire

L'écriture en mémoire se fait en plusieurs étapes : on spécifie une adresse sur l'entrée d'adresse (@), on présente le mot à retenir sur l'entrée IN, on positionne l'entrée de sélection $\overline{CS}$ à 0 et on donne une impulsion d'écriture sur W/R.

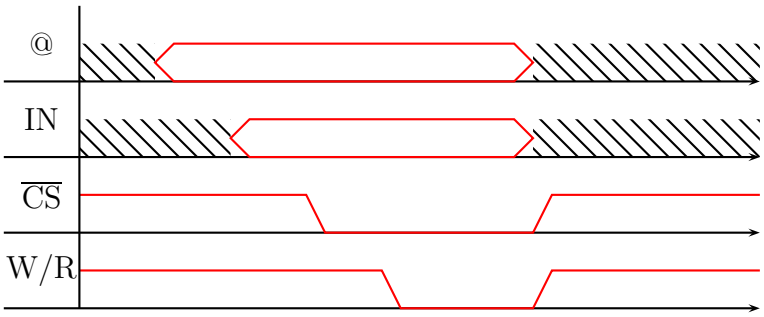


FIGURE 4.5 – Chronogramme écriture en mémoire

Pour lire une donnée, on spécifie l'adresse, on active $\overline{CS}$, on positionne W/R et on récupère la donnée.

Pour la mémoire des programmes, on n'a pas besoin de dire si on veut lire ou écrire en fonctionnement normal.

4.2 UAL

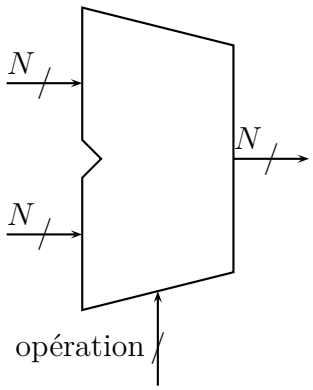


FIGURE 4.6 – Unité logique

4.2.1 Opérations

Il y a différents types d'opérations :

- Logiques : NOT, OR, AND (ex : 1000 OR 0001=1001)
- Décalage (droite/gauche) (ex : 00110000=00001100 (à droite))
- Arithmétique

4.2.2 Entiers

Les entiers sont représentés en complément à 2. Soit A un entier signé sur N bits. Soit A^+ son complément à 2. On a $A^+ = \overline{A} + 1$ avec $\overline{A}$ défini par $A + \overline{A} = 2^N - 1$, donc $A + A^+ = 0$.

4.2.3 Flottants – IEEE754

On décompose les 32 bits en :
- 1 bit de signe
- 8 bits pour l'exposant
- 23 bits pour la mantisse

Les instructions permettent la gestion des entrées/sorties ou le multi-tâches.

4.3 Jeu d'instructions

On a 4 types d'instructions : UAL/Mémoire/E-S/Contrôle.

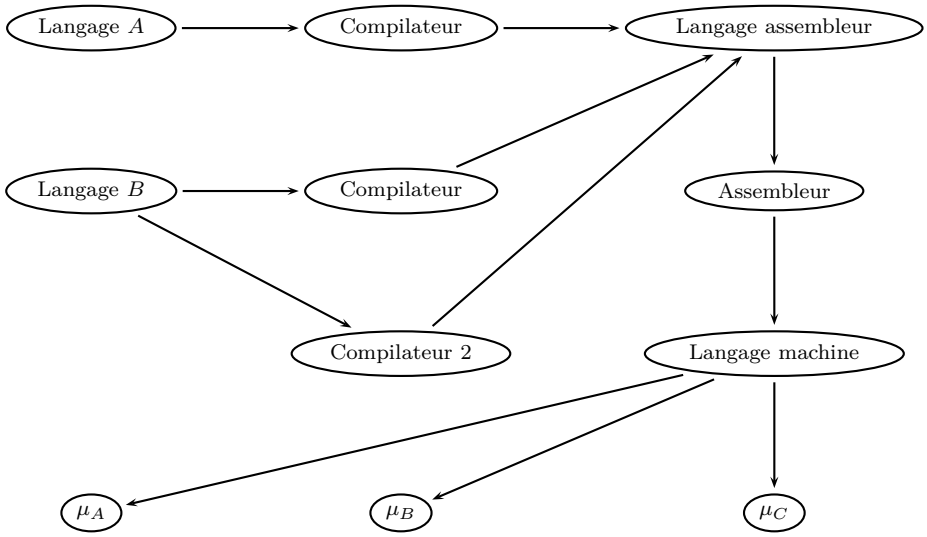


FIGURE 4.7 – Centralité du langage assembleur

Les instructions ont la forme <nom> <destination> <arguments>. Par exemple, les instructions UAL sont
- L'addition : ADD R_d R_1 R_2 ($R_d \leftarrow R_1 + R_2$)
- La soustraction : SUB R_d R_1 R_2 ($R_d \leftarrow R_1 - R_2$)
- Le décalage à gauche : SL R_d R_1 R_2 ($R_d = R_1 \ll R_2$, on décale R_1 de R_2 bits)

- Le décalage à droite : SR R_d R_1 R_2 ($R_d = R_1 \gg R_2$)

Les instructions logiques sont (opérations bit à bit) :

- AND R_d R_1 R_2
- OR R_d R_1 R_2
- XOR R_d R_1 R_2

Avec ces opérations, on peut en définir d'autres :

- NOT R_d R_1 = XOR R_d R_1 $1 \dots 1$
- INC R_d = ADD R_d R_d 1
- CLEAR R_d = XOR R_d R_d R_d

Fixons $R_1 = 1$. Définissons la fonction SET R_2 k qui assigne k à R_2. Par exemple,

XOR R_2 R_2 R_2	# $R_2 \leftarrow 0000$
ADD R_2 R_2 R_1	# $R_2 \leftarrow 0001$
SL R_2 R_2 R_1	# $R_2 \leftarrow 0010$
SL R_2 R_2 R_1	# $R_2 \leftarrow 0100$
ADD R_2 R_2 R_1	# $R_2 \leftarrow 0101$
SL R_2 R_2 R_1	# $R_2 \leftarrow 1010$

Une autre solution est de stocker toutes les constantes dans la mémoire de données. Il suffit alors d'éxécuter des instructions de la forme :

$$\text{OP.UAL}_i \underbrace{R_d}_{32 \text{ bits}} \underbrace{R_1}_{32 \text{ bits}} \underbrace{\text{Imm}}_{16 \text{ bits}}$$

avec i pour « immédiat » (ce qui signifie que le nombre Imm est codé en dur, ie que ce n'est pas le contenu d'un registre).

On a les instructions mémoires LD et STR (load et store) :

LD R_d R_a : $R_d \leftarrow$ la case mémoire à l'adresse R_a.
STR R_s R_a : stocke R_s à l'adresse R_a.

De même que précédemment, on a l'analogue LD_i R_d Imm avec Imm un entier positif codé sur 16 bits.

Des instructions E/S :

IN R_d R_p : $R_d \leftarrow$ la donnée sur le port indicé par R_p.
IN_i R_d Imm
OUT R_s R_p : envoie R_s sur le port indicé par R_p.

L'instruction de contrôle JMP R_s : on passe à l'instruction à l'adresse R_s dans la mémoire de programme.

Dans un assembleur, il y a deux parties :

- Une partie déclarative où on déclare les variables : VAR TOTO 100 crée le tableau TOTO de 100 cases
- Une partie de programmes déclarés par

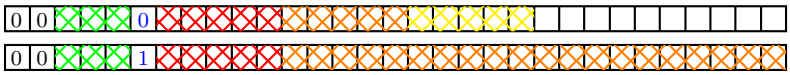
```
[label:] instruction commentaires
```

Par exemple

```
CPT :  XOR R0 R0 R0   # R0 ← 0
       XOR R3 R3 R3   # R3 ← 0
BCL :  OUT R3 8
       ADDi R3 R3 1   # R3++
       JEQU BCL R0
```

Une instruction est codée sur 32 bits. On a vu qu'il y avait quatre types d'instructions. Ce type est codé sur les deux premiers bits (31 et 30).

- 00 pour une instruction UAL
- 01 pour une instruction MEM
- 10 pour une instruction E/S
- 11 pour une instruction CTRL



- En vert l'instruction (dans l'ordre croissant des bits ADD, SUB, AND, OR, XOR, SL, SR)
- En bleu 0 ou 1 selon si on a une instruction immédiate.
- En rouge le registre de destination.
- En orange et jaune les registres sources.

Dans le cas immédiat, l'orange correspond à l'entier immédiat. Le codage binaire de l'instruction SUB R_{10} R_5 R_{17} sera :

$$00001001010001011000100000000000$$

Une manière concise est habituelle de noter le codage d'une intruction est de l'exprimer en hexadécimal. Pour l'instruction si-dessus, cela donne :

$$00001001010001011000100000000000 = 0x09458800$$

Une instruction est ainsi décrite sous la forme de 8 caractères hexadécimaux.

La valeur codée dans le champ immédiat est sur 16 bits. Les valeurs manipulées par le processeur sont, quant à elles, sur 32 bits. Cela pose a priori 2 problèmes :

- Opérations sur des arguments de type différent : 1 sur 32 bits, 1 sur 16 bits ;
- Acquisition de valeurs constantes limitées à 16 bits.

Le premier problème est levé par une conversion automatique d'une valeur 16 bits en une valeur 32 bits. Cette conversion s'opère au sein du séquenceur. Le second problème se résout de manière algorithmique en procédant en plusieurs temps :

- Acquisition des 16 bits de poids fort de la constante
- Décalage à gauche sur 16 bits
- Addition avec les 16 bits de poids faible

Par exemple, l'acquisition de la valeur 65537 dans le registre R_1 demandera de dérouler les instructions suivantes :

$$\begin{array}{ll} \text{SUB } R_1\ R_1\ R_1 & \#\ R_1 \leftarrow 0 \\ \text{ADDi } R_1\ R_1\ 1 & \#\ 65536 = 1 \ll 16 \\ \text{SLi } R_1\ R_1\ 16 & \#\ R_1 = 1 \ll 16 = 65536 \\ \text{ADDi } R_1\ R_1\ 1 & \#\ R_1 \leftarrow R_1 + 1 = 65537 \end{array}$$

Instructions mémoire :

- En vert l'instruction (000 pour LD et 001 pour ST)
- En rouge le registre de destination.
- En orange le registre source.

Instructions E/S :

- En vert l'instruction (000 pour IN et 001 pour OUT)
- En rouge le registre de destination.
- En orange le registre source.

Instructions de contrôle :

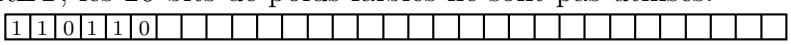
- JMP :

  Les 26 bits de poids faibles codent l'adresse de branchement (environ 64 millions d'adresses).
- CALL est similaire :

- Pour RET, les 26 bits de poids faibles ne sont pas utilisés.

- JEQU et JNEQ :

avec en vert 100 pour JEQU et 001 pour JSUP. En effet, le code de l'instruction JEQU est identique au code de l'instruction XOR. Ce codage permet à l'UAL d'effectuer l'opération logique XOR entre deux registres et de déterminer ainsi s'ils sont égaux via l'indicateur Z (résultat = zéro). De même, le code de l'instruction JSUP est identique au code de l'instruction SUB pour permettre à l'UAL d'effectuer une soustraction et de positionner l'indicateur P (résultat > 0).

L'adresse de saut est codée sur 11 bits. Sa valeur est comprise entre -2^{10} et $2^{10} - 1$. Un branchement relatif est donc limité à un saut à plus ou moins 1024 adresses à partir de la valeur courante du compteur ordinal (PC). Dans la majorité des cas, c'est amplement suffisant.

Pour des sauts conditionnels plus importants, on agrège 3 instructions de branchement : JEQU R_1 R_2 x devient alors

JEQU R_1 R_2 2
JEQU R_0 R_0 2
JMP x

4.4 Exemple de microprogrammation

On va traduire le code C suivant en assembleur.

```
1  int F1(int k){return (j+1);}
2
3  int main (int argc, char *argv[]){
4    int TAB1[100];
5    int TAB2[100];
6    int i,j;
7    i=0;
8    j=0;
9    while (i<100){
10     j=F2(j+TAB1[i]);
11     TAB2[i]=j;
12     i=F1(i);
13   }
14 }
```

4.4.1 Passage de paramètres

On met la valeur du paramètre dans la pile et on incrémente le Stack Pointer (SP). Pour renvoyer le résultat, on fait pareil. SP est stocké dans R_{30}.


```
        VAR TAB1 100          # Réserve en mémoire 100 cases pour TAB1
        VAR TAB2 100          # Réserve en mémoire 100 cases pour TAB2
MAIN :  XOR R10 R10 R10       # i = R10 = 0
        XOR R11 R11 R11       # j = R11 = 0
WHILE : SUB_i R0 R10 100      # R0 ← i - 100, soit -100 initialement.
        JEQU END R0           # Si R0 = 0, va à END sinon continue
        ADD_i R1 R10 TAB1     # R1 ← @TAB[i] (adresse de TAB[i])
        LD R2 R1              # R1 ← TAB[i]
        ADD R2 R2 R11         # R2 ← TAB[i] + j.
        ST R2 R30             # MEM[SP] ← R2
        ADD_i R30 R30 1       # SP ++
        CALL_i R31 F2
        SUB_i R30 R30 1       # SP - -
        LD R11 R30            # R11 = MEM[SP]
        ADD_i R4 R10 TAB2     # R4 ← @TAB2[i]
        ST R11 R4
        ST R10 R30
        ADD_i R30 R30 1
        CALL_i R31 F1
        SUB_i R30 R30 1
        LD R10 R30
        JNEQ WHILE R0
END :   JMP SYSTEM
```

4.4.2 Assembler le programme

On a besoin de deux tableaux :

Data	@	label	size
...	...	TAB1	100
...	...	TAB2	100
Prog	@	Mném.	dependency
MAIN	...	...	...
	...	...	...
F1	...	...	...
	...	...	...

Data	@	label	size
...	...	STRUCTF2	50
Prog	@	Mném.	Dép.
F2	...	...	...
	...	...	...
	...	JMP R31	...

4.4.3 Éditeur de liens

On a besoin de trois adresses : celle de début du programme (ex : 500), celle de début de stockage des données (3000) et pour le label SYSTEM (5000). On fusionne alors les deux tableaux, on obtient le tableau suivant :

...	3000	TAB1	100
	3100	TAB2	100
	3200	STRUCTF2	50
Program	address	instruction	dependency
MAIN	500	XOR R10 R10 R10	
	501	XOR R11 R11 R11	
BCL	502	SUBi R0 R10 R10	
	503	JEQU +18 R0	END
	504	ADDi R1 R10 TAB1	@TAB1
	505	LD R2 R1	
	506	ADD R2 R2 R11	
	507	ST R2 R30	
	508	ADDi R30 R30 1	
	509	CALLi R31 24	F2
	510	SUBi R30 R30 1	
	511	LD R11 R30	
	512	ADDi R4 R10 TAB2	@TAB2
	513	ST R11 R4	
	514	ST R10 R30	
	515	ADDi R30 R30 1	
	516	CALLi R31 +6	F1
	517	SUBi R30 R30 1	
	518	LD R10 R30	
	519	XOR R0 R0 R0	
	520	JEQU -18 R0	BCL
END	521	JMPi 5000	SYSTEM
F1	522	ST R1 R30	
	523	ADDi R30 R30 1	
	524	ST R2 R30	
	525	SUBi R30 2	
	526	LD R2 R1	
	527	ADDi R2 R2 1	
	528	ST R2 R1	
	529	LD R2 R30	
	530	SUBi R30 R30 1	
	531	LD R1 R30	
	532	JMP R31	
F2	533	ST R1 R30	
		contenu de F_2	
		JMP R31	

Chapitre 5

Mémoires

Le schéma général d'une mémoire est

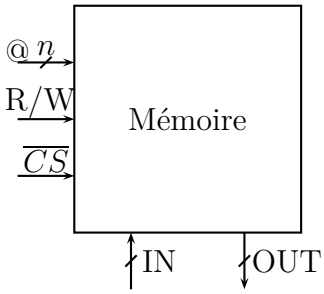


FIGURE 5.1 – Schéma général d'une mémoire

Les caractéristiques d'une mémoire sont sa capacité, son temps d'accès (en ns), son prix, sa taille, son débit (Mb/s), sa durée de vie, sa consommation et sa fiabilité.

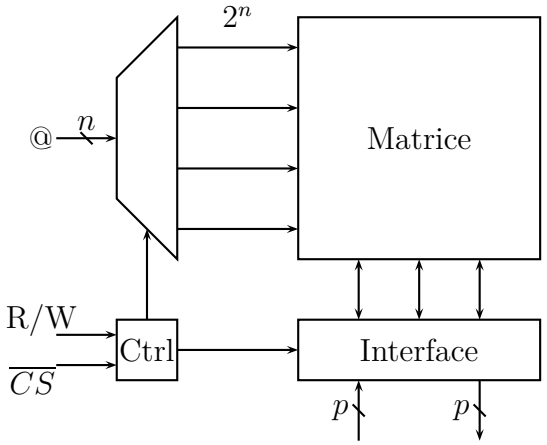


FIGURE 5.2 – Organisation globale de la mémoire

5.1 SRAM : Static Random Access Memory

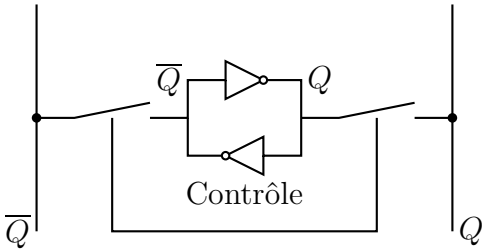


FIGURE 5.3 – Point mémoire

Pour lire, on ferme les interrupteurs et on lit les valeurs de Q et $\overline{Q}$. Pour écrire, on force une valeur sur Q et $\overline{Q}$ qui s'impose par rapport à la valeur sur la bascule car beaucoup plus forte. L'écriture consomme donc car elle crée un court-circuit dans la bascule. Quelques chiffres : $10 - 50$Mo et $1 - 10$ns d'accès.

5.2 DRAM : Synamic Random Access Memory

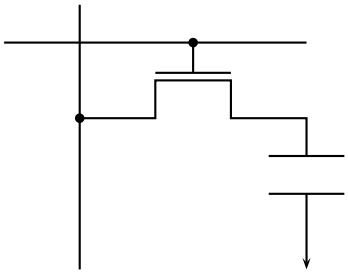


FIGURE 5.4 – DRAM

Si on veut stocker 1, on charge le condensateur. Sinon on la décharge. Mais le condensateur va se décharger en quelques millisecondes. On ajoute donc un mécanisme de rafraîchissement qui recharge les capacités toutes les 10-50ms.

Pour écrire, on vient charger ou décharger le condensateur. Pour lire, on précharfe la ligne de sortie à la valeur 1, ensuite on actionne la ligne horizontake. Si on avait stocké un 1, la valeur de sortie ne change pas. Si on avait 0, on peut détecter une variation. Après une lecture il faut réécrire la valeur. Quelques chiffres : 50-100ns et 1Go.

On peut diviser par deux le nombre de pins pour l'adresse en considérant une matrice : on adresse les lignes par les bits de poids fort et les colonnes par les bits de poids faible. Cette solution présente l'inconvénient d'augmenter le temps d'accès (registres en plus). On peut envoyer deux poids faibles de suite si on veut des mots côte à côte.

5.3 La famille des ROM

5.3.1 ROM : read only memory

On connecte (ou non) une diode entre la ligne i et la colonne j pour stocker un 1 (ou un 0) à la place j du mot mémoire numéro i. On ne peut donc pas changer ces données.

5.3.2 PROM : programmable read only memory

On met partout une diode et un fusible et on grille les fusibles là où on veut stocker des 0.

5.3.3 EPROM : erasable programmable read only memory

On met à la place de diode+fusible un transistor à deux grilles : une grille de contrôle et une grills flottante. En fonction du remplissage d'électrons, le transistor est soit ouvert soit fermé. On peut retirer ces électrons avec des UV.

5.3.4 EEPROM : Electricity erasable programmable read only memory

On peut maintenant effacer électriquement sans passer par les UV. Ceci donne ,aissance aux mémoires flash. Cette mémoire a un nombre d'écritures limité : environ un million.

Chapitre 6

Systèmes de fichiers

Pourquoi un système de fichiers ?

- Rémanence de l'information
- Partage de l'information
- Quantité d'information

Le disque dur est composé de plusieurs plateaux qui contiennent des pistes divisées en secteurs (128 octets et des informations pour le localiser). Le tout est géré par une tête de lecture.

Les performances du disque sont sa vitesse de rotation, sa stratégie d'allocation et le temps de tranfert vers la mémoire.

Le nom des fichiers dépend de l'OS. Par exemple Windows ne tient pas compte de la casse à la différence de Linux.

Il y a différents types de fichiers :

- les fichiers ordinaires (texte, binaire,...)
- les répertoires
- les tubes nommés
- les sockets
- les fichiers spéciaux (`/div/...` , imprimantes,...)
- les liens symboliques

Chaque fichier a des attributs ; sa taille, sa date de création/dernière modification, ses droits d'accès. On peut faire plusieurs opérations sur un fichier : CREATE, OPEN, CLOSE, WRITE, READ, DELETE, APPEND, SEEK.

Un système de fichiers est constitué d'un bloc de boot, d'un super bloc, d'une partie contenant l'organisation de l'arborescence et d'une zone libre.

Un bloc est composé de plusieurs secteurs, sa taille est comprise entre 1 et 8Ko. Un fichier est composé d'un nombre entier de blocs.

Chapitre 7

Hiérarchie mémoire – Cache

Plus la capacité d'une mémoire est grande, plus la vitesse diminue. Pour se donner une idée, les cycles d'un processeur à 3 GHz durent 0.33ns, alors que le temps d'accès à une DRAM est de 0.66ns. L'accès à d'autres mémoires représente 200 cycles !

On pourrait remplacer la DRAM par de la SRAM (dix fois plus rapide) mais la SRAM coûte 3000 euros le Go alors que la DRAM est à 50 euros le Go.

La hiérarchie mémoire est la suivante :

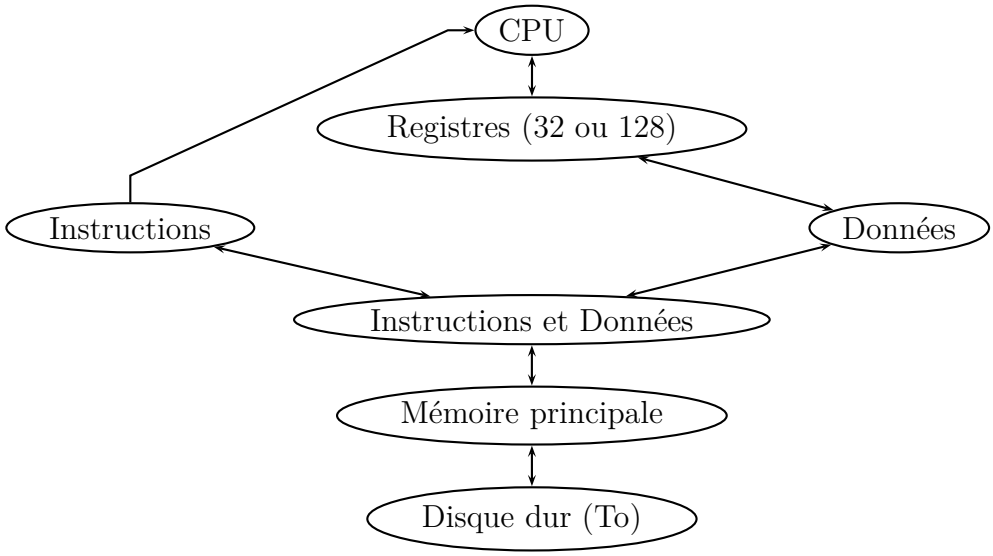


FIGURE 7.1 – Hiérarchie mémoire

Les blocs Instructions et Donnés en dessous des registres forment le cache L1, qui est constitué de 32Ko de SRAM.

En dessous, on trouve le cache L2 constitué de quelques Mo de SRAM et la mémoire principale, constituée de quelques Go de DRAM.

Pourquoi ça fonctionne ?

- Un objet déjà accédé a de grandes chances de l'être de nouveau.
- Des objets proches auront de grandes chances d'être accédés « ensemble ».

Mémoire cache « direct mapped »

Prenons l'exemple d'une mémoire de 32 mots (adresses de 00000 à 11111). On stocke au même endroits tout ce qui finit par 000

Ainsi, on a une plus petite mémoire qui contient les adresses de poids fort à poids faible fixé. Le cache est constitué d'un indicateur de présence (« flag », booléen), des deux bits de poids fort (« tag ») et des données. Il a une sortie HIT/MISS qui dit si le mot qu'on cherche est bien présent dans le cache.

Quand on fera LD_i R_1 10000, on se positionne dans le cache à l'adresse 000, on regarde s'il y a un mot (flag=true) et si oui, on vérifie que ce soit le bon (tag=10 ?).

Pour que le cache soit cohérent (ie que tout aille bien après un store en mémoire), on a plusieurs stratégies pour l'écriture en mémoire :

- « unité write throught » : on écrit dans le cache avant d'écrire dnas la mémoire qui est très pénalisante car le processeur est bloqué tant que l'écriture n'est pas réalisée
- « file d'attente » qui pose des problèmes quand la file est pleine
- « write back » : on écrit tout de suite dans la cache et on écrira dans la mémoire au moment où le bloc sera remplacé. C'est beaucoup plus efficace mais plus difficile à implémenter.

