Systèmes numériques

Milla Valnet

S1-L3-2019

Table des matières

Introduction

Ce polycopié a pour objectif de réorganiser les connaissances exigibles du cours de Systèmes Numériques, dont l'objectif est de comprendre comment se conçoit et s'utilise un circuit électronique (besoins, conception, simulation, vérification). Il s'appuie donc sur les cours dispensés à l'ENS mais aussi sur quelques ressources disponibles en ligne (voir bibliographie).

Ce cours reste un brouillon, il est donc très possible qu'il manque des choses ou que certaines soient fausse/imprécises/peu claires. Cherchant à l'améliorer, je vous invite donc à me faire toutes les remarques/corrections que vous jugeriez nécessaires. Donc même si j'espère qu'il pourra vous aider à réviser, gardez un regard critique par rapport à ce que j'ai écrit dedans !

1 Circuit : principes physiques

1.1 Transistors : définition et fonctionnement

Définitions

Un **transistor** est un composant électronique utilisé dans la plupart des circuits électroniques. C'est un semi-conducteur (conductivité entre celles des métaux et des isolants) qui permet de contrôler un courant ou une tension sur l'électrode de sortie grâce à une électrode d'entrée. Il effectue donc ces deux fonctions :

— fonction amplificatrice : $I_{\text{sortie}} = \beta I_{\text{entrée}}$

— fonction de commutation : passant ou bloquant

D'une certaine manière, un transistor est un morceau de conducteur dont la conductivité est commandé par la troisième broche. L'ordre de grandeur sur la taille d'un transistor se situe autour de $1\,\mu\text{m} \times 65\,\text{nm}$. Il existe différents types de transistors. Nous nous intéresserons ici aux transistors bipolaires et aux transistors à effet de champ.

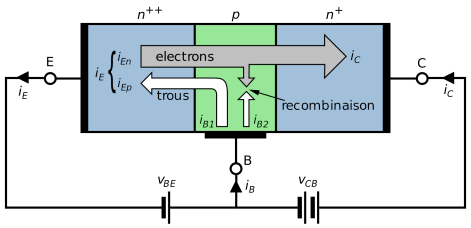
Loi de Moore

La **loi de Moore**, formulée dans les années 60 par Gordon Moore, cofondateur de la société Intel, est une conjecture affirmant que le nombre transistors par circuit de même taille allait doubler, à prix constants, tous les dix-huit mois. Moore en déduisait que la puissance des ordinateurs allait croître de manière exponentielle, et ce pour des années.

Il fit remarquer plus tard que sa conjecture se heurterait à une limite physique : la taille d'un atome.

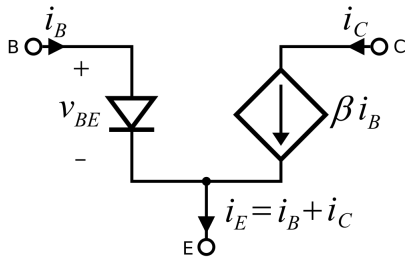
Transistor bipolaire

L'image ci-dessous représente un **transistor bipolaire** de type NPN (le type PNP inverse les charges des trois parties).



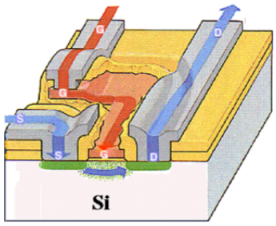
Sur ce type de transistor, les tensions V_{BE} et V_{CE}, ainsi que le courant entrant i_B sont positifs.Le E signifie ici émetteur, B base et C collecteur. L'émetteur, relié à la première zone N, est polarisé à une tension inférieur à celle de la base, reliée quant à elle à la zone P. Quant au collecteur, son potentiel est bien supérieur à celui de la base, ce qui implique que les électrons sont pour la plupart recueillis par le collecteur.

Cela se traduit par le comportement suivant. La jonction base-émetteur se comporte comme une diode. La jonction collecteur-émetteur se comporte de cette manière : lorsque $V_{BE} = 0$, elle peut être assimilée à une interrupteur ouvert. Sinon, elle laisse passer un courant (grossièrement) proportionnel au courant i_B mais bien plus élevé, d'où son utilisation possible en amplificateur ou interrupteur. Ceci donne l'illusion que i_B contrôle i_C En fonctionnement linéaire (amplificateur), on peut donc modéliser le transistor comme ceci :



Transistor à effet de champ

Le transistor **à effet de champ** utilise un champ électrique pour contrôler la conductivité d'un canal semi-conducteur possédant un excédent d'électrons (dopage de type N) ou de trous (dopage de type P), généralement un substrat de silicium. Il s'agit d'un composant à trois broches (grille, drain et source).



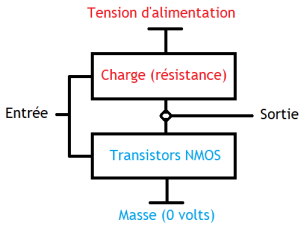
En l'absence de champ électrique, la grille est au même potentiel que le substrat et la source est donc isolé du drain. L'arrivée des charges sur la grille crée un champ magnétique local au-delà d'un certain seuil de la tension de grille. Les charges opposées à celle du canal sont donc attirées à la surface du substrat et le rendent conducteur. La source et le drain sont alors connectés par le transistor, lequel devient passant et peut alors être modélisé par une résistance dont la valeur dépend de la tension de grille.

1.2 Transistors PMOS et NMOS

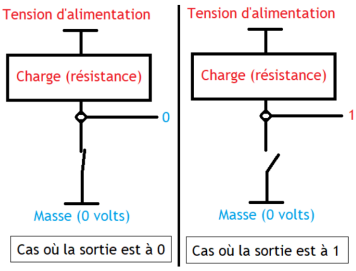
Tous les transistors présents dans les ordinateurs à ce jour sont de type **MOS** (Métal, Oxyde, Semi-conducteur). Ces transistors sont à effet de champ. Nous allons d'abord présenter le fonctionnement d'un transistor NMOS, puis par analogie, celui d'un transistor PMOS.

Transistor NMOS

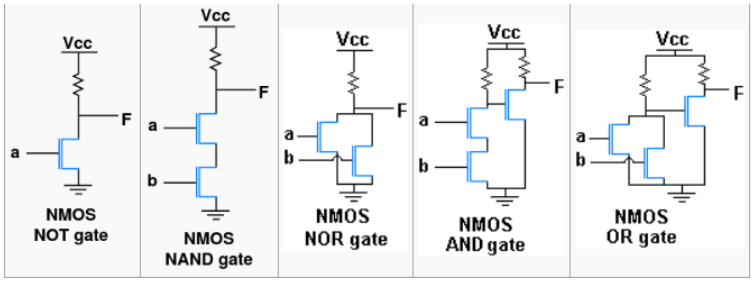
Un transistor **NMOS** est fabriqué en intercalant des transistors MOS avec une résistance selon le schéma suivant :



Lorsque la sortie doit être à 1, tous les transistors sont ouverts et il n'existe pas de chemin reliant la sortie à la masse, qui est alors relié à l'alimentation. Sa valeur est donc de 1 ; la résistance a pour but que l'intensité ne soit pas trop forte dans cette branche du circuit. Á l'inverse, lorsque la sortie doit être à 0, les transistors sont fermés et la sortie est reliée à la masse. Elle est donc mise à 0, ainsi que le résume le schéma suivant :



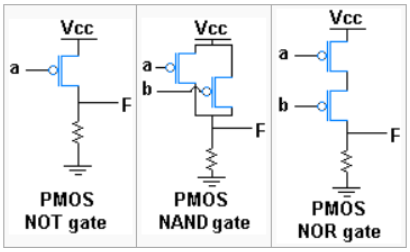
Les portes logiques classiques correspondent donc aux circuits NMOS suivants (pour plus de détails sur la construction de ces portes, consulter `https://fr.wikibooks.org/wiki/%C3%89lectronique/Les_familles_MOS_:_PMOS,_NOMS_et_CMOS`) :



On remarque que les portes OR et AND sont réalisées en ajoutant un NOT devant la sortie des portes NOR et NAND.

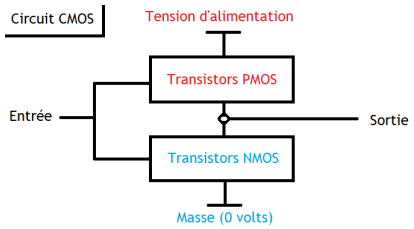
Transistor PMOS

Les portes **PMOS** sont construites de la même manière, à ceci près qu'elles sont inversées : la résistance prend la place de l'association de transistors et vice versa. Ainsi, les portes logiques classiques sont construites de cette manière :

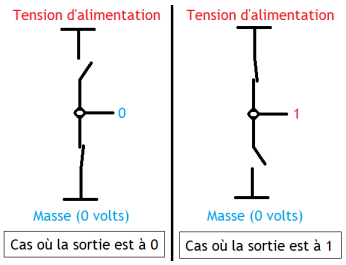


Logique CMOS

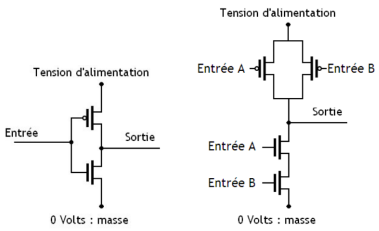
La logique **CMOS** associe les technologies PMOS et NMOS. Tout circuit CMOS se divise en deux parties, une intégralement constituée de transistors PMOS et une autre de transistors NMOS.



En voici le principe de fonctionnement. Le circuit PMOS du haut relie la sortie d'alimentation à la sortie si et seulement la sortie doit être à 1. De même, le circuit NMOS du bas relie la masse à la sortie si et seulement si la sortie doit être à 0 :



Les portes classiques sont donc construites de cette manière : on place le circuit CMOS correspondant à cette porte dans la partie CMOS et le circuit NMOS correspondant dans la partie NMOS, comme pour ces deux portes classiques NOT et NAND :



2 Logique

2.1 Représentation des nombres

Représentation en base b

Pour tout entier $n \in \mathbb{N}$, il existe un unique uplet $(a_0, ..., a_{k-1})$ avec $a_{k-1} \neq 0$ et $\forall i \in [|0, n-1|], a_i \in [|0, b-1|]$ tel que :

$$n = \sum_{i=0}^{k-1} a_i b^i$$

Pour $b = 10$, on parle de représentation décimale, $b = 16$ de représentation hexadécimale, $b = 8$ de représentation octale. Pour $b = 2$, il s'agit de la représentation binaire, que nous allons utiliser ici (les a_i sont les bits).

Représentation en complément à deux

Pour coder un entier relatif, on peut utiliser la représentation suivante :

$$n = -a_{n-1}2^{n-1} + \sum_{i=0}^{k-2} a_i b^i$$

Il faut simplement être vigilant si l'on veut passer d'un stockage sur $n-1$ bits à un stockage sur n bits :

$$n = -a_{n-1}2^{n-1} + \sum_{i=0}^{k-2} a_i b^i = -a_{n-1}2^{n-1} * (2-1) + \sum_{i=0}^{k-2} a_i b^i = -a_{n-1}2^{n} + \sum_{i=0}^{k-2} a_i b^i$$

Le n-ième bit a alors la même valeur que le bit de signe. C'est ce qu'on appelle une **extension de signe**.

Représentation en virgule fixe

Un nombre décimal d en base 2 peut également être approximé avec un uplet $(a_{k-1}, ..., a_0, a_{-1}, ..., a_{-m})$ (on ajoute une partie fractionnaire à la partie entière) :

$$n = \sum_{i=-m}^{k-1} a_i b^i$$

2.2 Logique combinatoire

On parle de **logique combinatoire**, car en traversant une porte combinatoire, la sortie est une combinaison des entrées.

Une **variable logique** est un élément de 0,1. On interprète 0 comme Faux et 1 comme Vrai. On peut coder une variable logique sur 1 bit (binary digit). En effet, le binaire peut être interprété comme deux niveaux de tension différents : un simple interrupteur suffit à coder ceci.

Les **fonctions combinatoires**, à la différence des fonctions séquentielles, ne dépendent que de l'état actuel de leurs entrées et non de leurs passés. Ce sont elles que l'on cherche à mettre sous forme de circuit en logique combinatoire. Il existe plusieurs manières de les représenter.

Une **table de vérité** est un tableau donnant la valeur de chaque sortie pour chaque entrée. Elle se contente donc de décrire le comportement du circuit sans décrire le circuit en lui-même.

On peut également le représenter sous la forme d'une équation, avec les notations suivantes :

Opérateur	Notation
NON a	$\overline{a}$
a ET b	a.b
a OU b	a+b
a XOR b	$a \oplus b$

Pour plus d'informations sur ces représentations., consulter `https://fr.wikibooks.org/wiki/Fonctionnement_d%27un_ordinateur/Les_circuits_combinatoires`.

La **distance de Hamming** entre deux entiers de n bits est le nombre de caractères différents entre ces deux mots (10 et 01 ont une distance de Hamming de deux).

Les deux prochaines sous-parties détaillent deux méthodes permettant de transformer une table de vérité en équation logique simple. Pour plus de détails, consulter : `https://fr.wikibooks.org/wiki/Fonctionnement_d%27un_ordinateur/Les_circuits_combinatoires`

Formes normales disjonctive et conjonctive

Mettre une équation logique sous **forme normale disjonctive** consiste à l'écrire sous la forme $\sum \Pi(e_i)$ (le symbole Π représente un AND entre plusieurs expressions et le symbole $\sum$ un OR entre plusieurs expressions), où les e_i sont des variables logiques ou leurs compléments. Si chacun des produits contient toutes les variables d'entrée, on parle de **forme canonique disjonctive** et ses produits sont appelés **mintermes**.

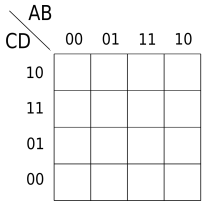
Mettre une équation logique sous **forme normale conjonctive** consiste à l'écrire sous la forme $\Pi \sum(e_i)$. Si chacun des produits contient toutes les variables d'entrée, on parle de **forme canonique conjonctive** et ses produits sont appelés **maxtermes**.

Il est aisé d'obtenir la forme canonique conjonctive d'une fonction logique dont on possède le tableau de vérité. A chaque ligne du tableau correspond une équation logique. On fait un OR entre toutes les lignes du tableau dont la sortie vaut 1 et on obtient une forme canonique conjonctive pour la fonction. On peut ensuite la simplifier par des manipulations logiques.

On peut aussi obtenir la forme canonique disjonctive, mais la méthode est moins intuitive.

Tableaux de Karnaugh

Pour obtenir une équation simple pour une fonction logique, on peut également utiliser la méthode des **tableaux de Karnaugh**. Cette méthode est utile pour les circuits simples (mais trop longue lorsque le nombre d'entrées dépasse 5 ou 6). Pour ce faire, on construit une sorte de table de vérité. Si l'on a (n variables en entrée $(a_1, ..., a_n)$, on les partitionnent en deux ensembles de tailles égales et on place sur la ligne du haut tous les mots $a_1...a_{\frac{n}{2}}$ possibles et sur la colonne de droite tous les mots $a_{\frac{n}{2}+1}...a_n$, comme ci-dessous pour 4 variables A, B, C, D :



Ensuite, dans la case de coordonnées $(b_{\frac{n}{2}+1}...b_n, b_1..b_{\frac{n}{2}})$, on inscrit la valeur de sortie pour les entrées $a_i = b_i$. On fait ensuite des regroupements par paquets de 2^n des 1 présents dans le tableau (ces regroupements doivent former des rectangles et peuvent se recouvrir - c'est même mieux.

Enfin, on convertit chaque regroupement en une équation logique : on trouve la variable qui ne varie pas dans les lignes et colonnes du regroupement, l'inverser si elle vaut toujours 0, faire un ET entre les variables qui ne varient pas puis faire des OU entre les différents regroupement.

De cette manière, on obtient une équation logique relativement simple pour la fonction logique fournie.

2.3 Représentation des circuits

Symboles

Pour simplifier la représentation des circuits, on utilise différents symboles, les portes logiques (pour plus de détail, consulter la page `https://fr.wikibooks.org/wiki/Fonctionnement_d%27un_ordinateur/Les_portes_logiques`).

FIGURE 1 – Portes NOT, AND, NAND, OR, XOR

Pour simplifier les notations, lorsqu'on place la porte AND (ou une autre) en série avec la porte NOT, on ajoute simplement un petit rond à la sortie de la porte. On peut obtenir toutes les fonctions logiques en combinant les portes précédentes (on peut n'utiliser que les portes NOT, AND et OR - en fait, une porte NOR ou NAND suffit).

Notion de temps de calcul

On a vu dans la partie précédente qu'une association de transistors permettaient de créer ces portes. Ainsi, les portes logiques respectent les lois de la physique : un changement d'état demande un temps que l'on nomme **temps de propagation**. Le temps de propagation à travers une porte est pré-caractérisé : il faut 1 ns pour parcourir une porte AND ou OR et 2 ns pour une porte XOR.

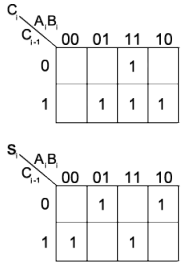
La connaissance du temps de propagation à travers les portes de bases nous permet, en sommant simplement les temps individuels de propagation à travers chaque porte, de connaître le temps de propagation à travers notre circuit entier. On définit pour cela le **chemin critique**, qui est le chemin à travers le circuit demandant le plus de temps pour être traversé. C'est lui qui limite et donc détermine la fréquence de fonctionnement d'un circuit.

2.4 Opérateurs

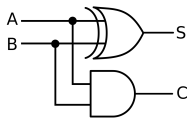
Dans cette partie, nous allons nous demander comment combiner les portes logiques permet d'effectuer les opérations arithmétiques élémentaires qu'on exige généralement d'un ordinateur (addition, soustraction, multiplication...). Je n'ai pas fait de partie sur la division parce qu'elle a peu (quasiment pas en fait) été abordée en cours, mais si quelqu'un veut s'en charger je lui envoie le fichier.tex.

Addition

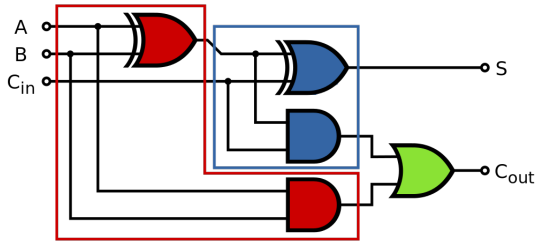
Notons a_i et b_i les nombres dont on doit calculer la somme s_i et la retenue sortante c_i en disposant de le retenue entrante c_{i-1}. Voici les tables de Karnaugh pour c_i et s_i, les cases vides correspondant à des 0 :



On remarque alors que $s_i = a_i$ xor b_i xor c_{i-1} et $c_i = (a_i$ and $b_i)$ or $(a_i$ and $r_i)$ or $(b_i$ and $r_i)$. C'est avec ce constat qu'on peut passer de calculs arithmétiques à calculs booléens.
Lorsqu'on réalise une somme de 2 bits sans retenue entrante (**half-adder**), on a donc le circuit suivant :

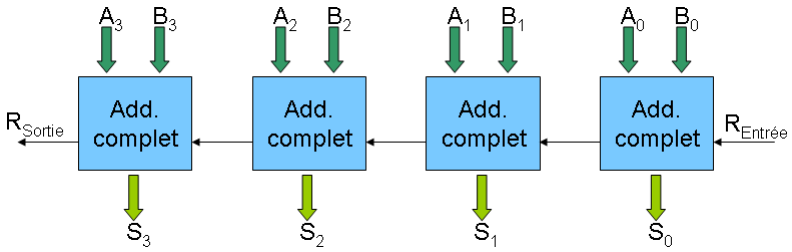


Lorsqu'on a également une retenue entrante (**full-adder**), on déduit ce circuit des calculs précédents :



Il y a 3 portes dans le chemin critique du full-adder, ont une porte xor. Le temps de calcul est de 4 ns.

On sait additionner 3 bits avec le full-adder. On cherche désormais à additionner 2 entiers n-bits (avec éventuellement une retenue entrante). L'idée est de réaliser le calcul "colonne par colonne", commme si l'on posait l'addition, en utilisant un additionneur à propagation de retenue (**carry-ripple adder**). Son fonctionnement est résumé par le schéma suivant :



Cependant, on multiplie alors par environ n la longueur du chemin critique donc le temps de propagation. On cherche à améliorer cette performance.

On peut optimiser le temps de calcul en utilisant un autre additionneur, le **carry-select adder**. L'idée consiste à séparer en blocs les nombres dont on doit calculer la somme : pour chaque bloc, on calcule leur addition en supposant que la retenue et 1 et en supposant que la retenue est 0. On utilise ensuite un multiplexeur pour sélectionner le résultat suivant la valeur de la retenue.
Cela permet de passer d'un facteur n à $sqrtn$.

Il existe également d'autres additionneurs (carry-save, carry-look aheand, carry-skip), qui donne des résultats proches du carry-select en temps mais nécessite moins de matériel. Pour se renseigner : `https://fr.wikibooks.org/wiki/Fonctionnement_d%27un_ordinateur/Les_circuits_de_calcul_entier`.

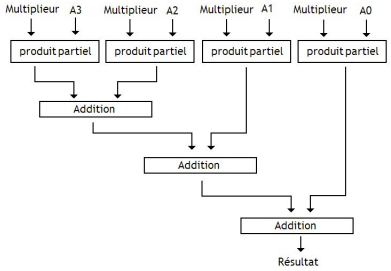
Je conclurai cette partie en faisant un point sur les **débordements d'entiers**. On veut en effet que l'ordinateur nous prévienne si le résultat du calcul effectué sort des bornes acceptées. (référence : diapo TD2 et TD2) Je devrais ajouter ça aujourdhui.

Soustraction

Il suffit de réaliser que en complément à deux, $-a = \overline{a} + 1$ (vrai pour un entier n-bit), donc $a - b = a + \overline{b} + 1$, ce qui permet d'utiliser les mêmes circuits que pour l'addition.

Multiplication

La multiplication de 2 bits correspond à une porte ET (ce qui est la raison pour laquelle on souvent Et multiplicativement). Pour multiplier deux entiers n-bits, on calcule alors en parallèle les produits partiels (comme lorsqu'on pose une multiplication), qu'on additionne ensuite. Ce schéma permet de se représenter le fonctionnement :



Calcul du maximum

Pour calculer le maximum entre deux nombres, on les compare bit à bit en commençant par le poids fort. Notons asb_i et bsa_i tels que si après la comparaison du i-ème bit :

— On sait que $a > b$: $asb_i = 1, bsa_i = 0$
— On sait que $a < b$: $asb_i = 0, bsa_i = 1$
— On ne sait pas encore : $asb_i = bsa_i = 0$

Ceci permet de déduire $asb_i = asb_{i+1} + \overline{bsa_{i+1}} a_i \overline{b_i}$ et $bsa_i = bsa_{i+1} + \overline{asb_{i+1}} \overline{a_i} b_i$

Le TD2 permet de se familiariser avec ces notions.

Dans cette partie, on a étudié la logique combinatoire, qui vérifie ceci : si on fournit au circuit deux entrées identiques à deux instants différents, les deux résultats seront identiques. Ceci n'est pas forcément vrai en logique séquentielle : on distingue différents états du circuit, qui possède donc une mémoire. C'est l'introduction des points mémoires qui permet la transition de la logique combinatoire à la logique séquentielle.

2.5 Logique séquentielle

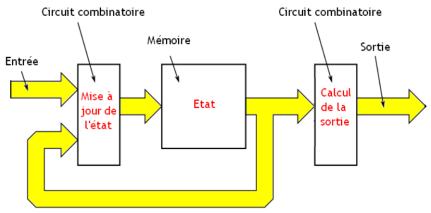
Les circuits combinatoires permettent de manipuler l'information. En revanche, ils ne permettent pas de la stocker. C'est la raison pour laquelle nous nous intéresserons à la **logique séquentielle**, qui dispose quant à eux d'une mémoire sur laquelle on peut réaliser

des actions de **lecture** (accès à la valeur stockée) et d'**écriture** (modification de la valeurs stockée).

L'**état** du circuit correspond à l'ensemble des données stockées dans la mémoire à un instant donné. Cet état peut dépendre de la donnée envoyée en entrée, mais aussi de l'état antérieur de la mémoire. On peut diviser un circuit séquentiel en deux parties, l'une consacrée à la mémoire (stockant l'état du circuit), et l'autre, un circuit combinatoire permettant la mise à jour de l'état et le calcul de la sortie. On peut classer les circuits séquentiels en deux catégories.

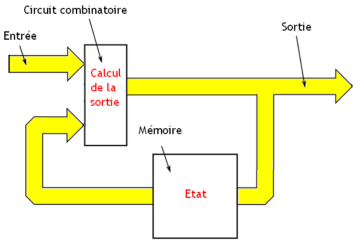
Automate de Moore

Dans un **automate de Moore**, la sortie ne dépend que de l'état mémorisé. Il peut être représenté par le schéma suivant.



Automate de Mealy

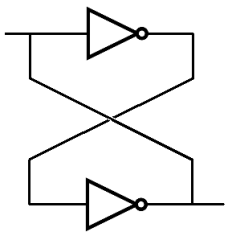
Dans un **automate de Mealy**, la sortie dépend de l'état du circuit, ainsi que de ses entrées.



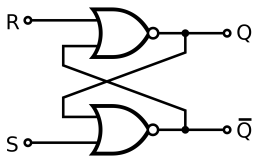
Point mémoire

On s'intéresse ici à la manière de créer un **point mémoire**. Tout d'abord, la notion de mémoire et donc celle de logique séquentielle est indissociable avec la notion de temps. On dispose donc d'un temps de référence, avec l'utilisation d'une horloge (en temps discret, alterne 0 et 1 avec une période constante, la période d'horloge).

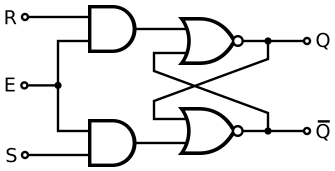
Pour créer un point mémoire, on peut partir de l'idée très simple de boucler deux portes NOT l'une sur l'autre, ce qui permet de conserver la donnée pour deux tours d'horloge.



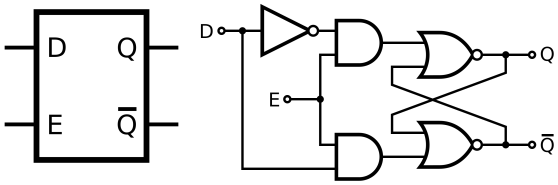
En réalité, on ne peut pas faire ça avec une porte not, car il faut faire pouvoir accéder à la donnée et la modifier. Les **bascules RS** possède donc une sortie pour récupérer le bit mémorisé et deux permettant de le modifier (une permettant de lui donner la valeur 1 et l'autre la valeur 0). C'est ce que fait la bascule RS à NOR suivante :



On peut lui ajouter une entrée E (pour enable) indiquant si l'on peut ou non écrire en méoire, ce qui donne le circuit suivant :



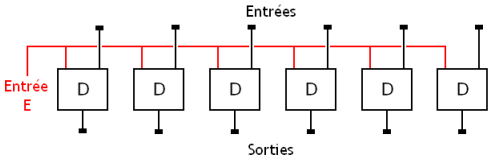
Les **bascules D** sont différentes en ce que deux entrées sont envoyées, E (Enable) et D (Data), qui contient directement la donnée à mémoriser. Une Bascule D à NOR ressemble donc à ceci :



Registres

Dans la partie précédente, on a vu comment stocker un bit. Mais on peut assembler les bascules afin de créer un circuit capable de stocker plusieurs bits, que l'on appelle un **registre**.

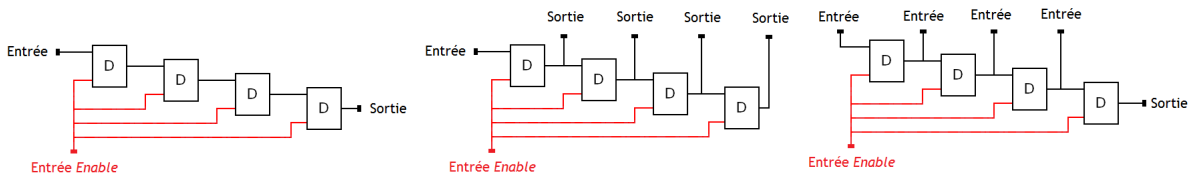
La manière la plus simple de procéder est sans doute la suivante :



E représente toujours la variable Enable et les bascules sont toutes mises à jour en même temps.

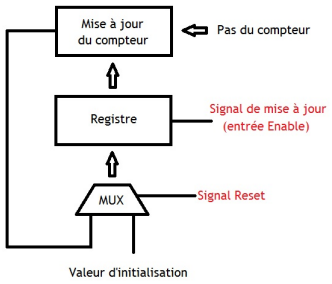
Les **registres à décalage** fonctionnent différemment. Leur contenu peut être décalé d'un cran sur la droite ou sur la gauche.

Les registres à **entrées et sorties série** permettent de mettre en attente des bits en conservantl leur ordre. Les registres **entrées séries et sorties parallèles** permettent de reconstituer bit par bit un nombre envoyer bit par bit sur un fil. Les registres **entrées parallèles et sortie série** permettent quant à eux d'envoyer un nombre donné bit par bit sur un fil.



Compteurs

En logique séquentielle, il peut être utile d'avoir des **compteurs**, qui mémorisent un nombre et le mettent à jour sur commande. Un compteur peut être schématisé de cette manière.



Tous les compteurs comptent modulo une certaine valeur : ce peut être la valeur maximale que peut contenir leur registre ou une valeur plus basse dont le dépassement est détecté par un circuit combinatoire. On peut même ajouter une entrée RESET permettant de réinitialiser le compteur sur commande.

On distingue compteurs synchrone et asynchrone. Dans le premier pas, Enable, qui permet d'incrémenter (ou de décrémenter) le compteur est en fait le signal de l'horloge.

Machines à états

Un circuit séquentiel peut être vu comme une machine à états (graphe) dont la transition entre deux états peut être conditionnée par la valeur de certaines entrées. Ces graphes permettent de comprendre le fonctionnement du circuit.

Optimisation : pipeline

Un **pipeline** (ou chaine de traitement) est lélément d'un processeur dans lequel l'exécution des instructions est découpée par étapes. L'objectif est de commencer une nouvelle instruction sans attendre que la précédente soit terminée. Le schéma ci-dessous met en parallèle l'exécution d'instructions sans et avec pipeline :

Instr. No.	Pipeline Stage						
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX
Clock Cycle	1	2	3	4	5	6	7

On nomme **étage** d'un pipeline chaque circuit dédié à une étape.

La technique la plus utilisée est le pipeline synchrone : les différents étages sont séparés par des registres tampons reliés au signal d'horloge. Ils deviennent accessibles en écriture à chaque période, ce qui permet aux données d'accéder à l'étape suivante.

Au final, le processeur a besoin de la même durée pour exécuter une unique instruction donnée car il doit parcourir tous les étages, mais en revanche, il augmente sa fréquence de traitement des informations (en la multipliant par le nombre d'étages) car chacun de ces étages à un temps d'exécution bien plus court.

2.6 Logique asynchrone

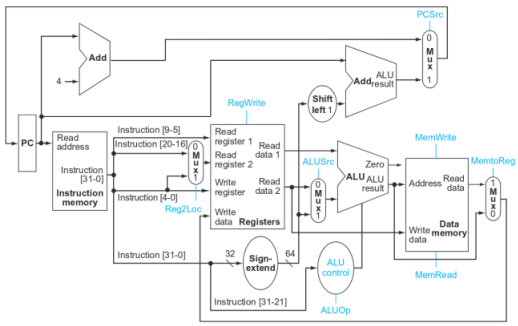
La logique asynchrone correspond en quelque sorte à une logique séquentielle sans horloge. On possède des portes de rendez-vous ou porte C. En logique synchrone, c'est le composant le plus lent qui rythme l'intégralité du circuit. La logique asynchrone peut plus facilement s'adapter à la situation, mais nécessite plus de câbles et de transistor. On n'a pas besoin de garantir (contrairement en logique synchrone) que tout arrive en même temps.

3 Le microprocesseur

Un processeur (aussi appelé CPU pour Central Processing Unity) est un composant qui exécute les instructions machines (opérations élémentaires) d'un programme informatique.

Le microprocesseur, quant à lui, est un processeur dont les composants électroniques sont suffisamment miniaturisés pour pouvoir tenir dans un seul circuit intégré. C'est l'apparition du transistor qui a permis une telle miniaturisation.

Une référence intéressante pour plus d'information est le livre "Computer, organization and design", de Patterson et Kennessy. Ci-dessous un schéma de principe faisant apparaître ses principaux composants.



Je ne détaille pas plus, le reste correspond au projet et le cours qu'on a eu à ce sujet était complet et clair.

Pour simuler un microprocesseur ou n'importe quel circuit, on peut utiliser un **langage de description matérielle** comme VHDL, Verilog ou Minijazz.

4 Cryptographie

Cette partie est très succinct, si quiconque veut y ajouter quelque chose, je peux lui transmettre le fichier .tex correspondant.

4.1 Introduction

La cryptographie vise à garantir ces trois aspects de la transmission d'un message :
— **authenticité** : garantir l'identité de l'émetteur
— **sûreté** : garantir la confidentialité du message
— **intégrité** : garantir qu'une donnée n'a pas été modifiée

4.2 Méthodes

Authenticité

Pour garantir l'identité de l'émetteur, l'émetteur et le récepteur peuvent disposer d'une clé privée. L'idée est que le nombre de clés soit suffisamment grand pour qu'une recherche exhaustive de la clé ne soit pas envisageable. En pratique, 96 bits suffisent.

On cherche à ne pas rendre accès aux paramètres privés avec les paramètres publics. Par exemple, une clé publique peut être $N = pq$ où p et q sont des nombres entiers très grands et p et q des donner privés. La factorisation de N est alors trop difficile pour remonter facilement à p et q.

Sûreté

Plutôt que de transmettre x, on peut transmettre $f(x)$ de telle sorte que la personne recevant le message, en connaissant la réciproque f^{-1}, peut en déduire x (il faut donc que f soit bijective). On peut par exemple modifier des bits (Substitute Bytes) toujours de la même manière, utiliser des multiplications/divisions par des polynômes... Un bon algorithme de chiffrement (encryption) mélange plusieurs étapes et doit être bijectif pour permettre le décodage (decryption).

Intégrité

Pour vérifier qu'une donnée n'a pas été modifiée, on peut allouer un bit de parité (contenant la parité du résultat) et vérifier si sa valeur à changer. En revanche, cela ne permet pas de corriger le résultat.

4.3 Attaques

Plusieurs moyens peuvent être mis en œuvre pour obtenir une clé.

Observer un calcul

Un calcul peut être observer de diverses manières. On nomme **Side-channel attack** le nombre d'observations du calcul nécessaire pour connaître la clé.

— On peut ajouter une résistance et on mesure le temps d'exécution du calcul ainsi que le courant nécessaire. Cependant, l'ajout d'une résistance peut entraîner des dysfonctionnements du circuit.
— De la même manière, on peut ajouter une inductance.
— On peut mesurer le rayonnement émis par le circuit. Cette méthode à l'avantage de ne pas ajouter de composant au circuit de base.

Pour garantir la sécurité vis à vis de ses attaques, on cherche à obtenir des algorithmes **équilibrés**, i.e. le nombre de multiplications, d'additions (qui n'utilise pas le courant de la même manière)... ne dépend pas des bits de la clé. De cette manière, repérer le motif ne donne pas d'information sur la clé. On peut insérer des valeurs dont le résultat ne dépend pas. C'est ce que l'on nomme un **masquage**.

Casser un calcul

De même, il existe différentes manières de casser un calcul.

— Avec un faisceau laser, on peut créer des paires d'électrons ou de trous dans les transistors, ce qui modifie leur comportement et donc le résultat.

— L'injection d'un champ intense induit du courant et peut générer des fautes. On parle de **EMI Disturbance System**.

— On peut induire des fautes en mémoire. On parle de **Row Hammer**.

— La méthode de l'**AES** consiste quant à elle à perturber un unique octet.

L'intérêt de casser un calcul est de regarder comment la propagation des perturbations peut modifier le résultat. Cela peut en effet permettre de trouver la clé.

Références

[1] Cours dispensés à l'ENS Ulm par Hadrien Barral, Sumanta Chaudhuri, Sylvain Guilley et Georges-Axel Jaloyan

[2] Polycopié de Jean Vuillemin disponible sur la page du cours ou avec le lien `https://perso.telecom-paristech.fr/guilley/ENS/ENS2015Vuillemin.pdf`

[3] `https://fr.wikibooks.org/wiki/Fonctionnement_d%27un_ordinateur`