

MATTHIEU LOUMAIGNE

ÉLECTRONIQUE NUMÉRIQUE

M1 PHYSIQUE - UNIVERSITÉ D'ANGERS - JANVIER 2022



Table des matières

I	7
1 Logique Booléenne	9
1.1 Quelques définitions	9
1.1.1 Variable binaire	9
1.1.2 Fonction binaire	9
1.1.3 Table de vérité	9
1.2 Expressions booléennes	9
1.3 Représentation canonique	10
1.4 Deux théorèmes de l'algèbre booléenne	11
1.4.1 Les théorèmes de Boole	11
1.4.2 Les théorèmes de De Morgan	11
1.5 Les portes logiques	12
1.5.1 Porte NON (NOT)	12
1.5.2 Porte OU (OR)	12
1.5.3 Porte ET (AND)	12
1.5.4 Porte OU-EXCLUSIF (XOR)	12
1.5.5 Porte NON-ET (NAND)	12
1.5.6 Porte NI (NOR)	13
1.5.7 Multiplexeur (MUX)	13
1.5.8 Demultiplexeur (DMUX)	13
1.6 Mise en pratique.	14
1.7 TP / TD	14
1.7.1 Quelques exercices autour de l'algèbre booléenne	14
1.7.2 De l'universalité de la porte NAND	15
2 Arithmétique Booléenne	17
2.1 Les nombres binaires	17
2.2 Addition binaire	17
2.3 Nombre binaire algébrique	18
2.4 Représentation d'un nombre en virgule fixe	19
2.4.1 La représentation en virgule flottante	19
2.4.2 La représentation en virgule fixe	19
2.4.3 La multiplication	21
2.4.4 La division par 2	21
2.5 Les additionneurs	21
2.5.1 Le demi additionneur	22
2.5.2 L'additionneur complet	22

2.5.3	L'additionneur	22
2.5.4	Incrémenteur	24
2.6	L'unité Arithmétique et Logique (ALU)	24
2.7	TD 2	26
2.7.1	Complément à 2	26
2.7.2	Additionneur	27
2.7.3	ALU	27
3	Logique séquentielle	29
3.1	Quelques généralités	29
3.1.1	Horloge	29
3.1.2	Bascules	30
3.1.3	Compteurs	30
3.2	Implémentation Bascules à partir de portes logiques	30
3.2.1	Boucler des portes logiques	30
3.2.2	Bascule RS	31
3.2.3	Bascule D	31
3.3	Implémentation de la mémoire à partir de bascule	32
3.3.1	Registre	32
3.3.2	Random Access Memory	33
3.3.3	Compteur	33
3.4	TD 3	33
3.4.1	bascule D	33
3.4.2	Registre	33
3.4.3	RAM	35
3.4.4	Compteur	35
4	Langage machine	37
4.1	Architecture de l'ordinateur	37
4.1.1	Mémoire	37
4.1.2	Central Processing Unit (CPU)	38
4.1.3	Notre architecture	39
4.2	Langage machine	41
4.2.1	Notre ordinateur	41
4.2.2	Instruction adresse	42
4.2.3	Instruction calcul	42
4.2.4	Instruction vidéo	44
4.3	Programmation	45
4.3.1	Addition de deux nombres	45
4.3.2	Multiplication	48
4.4	TD 4	51
4.4.1	Réalisation du circuit de condition de jump	51
4.4.2	Le programme counter	52
4.4.3	CPU	52
4.4.4	L'ordinateur	53
4.4.5	Factorielle	53

II	Micro-contrôleur	57
5	Quelques généralités sur les microcontrôleurs	59
5.1	Arduino	60
5.1.1	Présentation de la plateforme	60
5.1.2	Getting started	60
5.2	Interruption	61
6	TP cours sur la plateforme Arduino	63
6.1	Sortie numérique	63
6.1.1	Lire la doc (RTFM)	63
6.1.2	Faire clignoter la LED placée sur la carte arduino (pin 13)	63
6.1.3	Faire clignoter une LED externe	64
6.1.4	Emettre SOS en Morse	64
6.1.5	Jouer une mélodie	64
6.1.6	Piloter un interrupteur avec l'arduino	65
6.2	Communication par la voie série.	65
6.2.1	Lire la doc (RTFM)	66
6.2.2	Communication de l'arduino vers l'ordinateur	66
6.2.3	Communication de l'ordinateur vers l'arduino	66
6.3	Sorties PWM	67
6.3.1	Lire la doc (RTFM)	67
6.3.2	Etude du signal à l'oscilloscope	67
6.3.3	Modulation de l'intensité d'une LED	68
6.4	Entrée numérique - digital read	68
6.4.1	Lire la doc (RTFM)	68
6.4.2	Les résistances de pull-up	69
6.5	ADC Analog to Digital Conversion	71
6.5.1	Lire la doc (RTFM)	71
6.5.2	Utilisation d'un potentiomètre	71
6.6	DAC Digital to Analog Conversion	71
6.6.1	PWM	71
6.7	Interruption	71
6.8	Timer.	71
6.9	Utiliser des commandes plus bas niveau.	71
6.10	Mini Projet	71
	Bibliographie	73
A	Quelques éléments de corrections des TD	75

Première partie

1 Logique Booléenne

1.1 Quelques définitions

1.1.1 Variable binaire

L'algèbre booléenne manie des grandeurs binaires¹ dite booléennes et qui sont typiquement appelées vrai/faux, 1/0, oui/non, on/off, etc. . .

1. On travaille avec des valeurs binaires (0 et 1) et non décimale avant tout pour des raisons technologiques, il est beaucoup plus facile de créer un interrupteur avec l'état ouvert ou fermé qu'un système permettant de contrôler et de mesurer de manière robuste des valeurs intermédiaires.

1.1.2 Fonction binaire

Une fonction booléenne est une fonction qui opère avec des grandeurs booléennes en entrée et retourne des valeurs booléennes.

1.1.3 Table de vérité

Une table de vérité énumère toutes les cas possibles d'une fonction booléenne. Plus précisément, la table de vérité répertorie de façon exhaustive toutes les combinaisons possibles de valeurs d'entrée et en donne les valeurs de sortie correspondantes (voir par exemple la table 1.1).

x	y	z	f(x,y,z)
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

TABLE 1.1: Un exemple de table de vérité pour la fonction booléenne $f(x,y,z)$

1.2 Expressions booléennes

Une fonction booléenne peut aussi être décrite en appliquant des opérations booléennes sur les variables d'entrées. Listons les trois principales opérations :

Fonction	x	0	0	1	1
	y	0	1	0	1
Constante 0	0	0	0	0	0
And	$x \cdot y$	0	0	0	1
x And Not y	$x \cdot \bar{y}$	0	0	1	0
x	x	0	0	1	1
Not x And y	$\bar{x} \cdot y$	0	1	0	0
y	y	0	1	0	1
Xor	$x \cdot \bar{y} + \bar{x} \cdot y$	0	1	1	0
Or	$x + y$	0	1	1	1
Nor	$\overline{x + y}$	1	0	0	0
Equivalence	$x \cdot y + \bar{x} \cdot \bar{y}$	1	0	0	1
Not y	\bar{y}	1	0	1	0
Si y alors x	$x + \bar{y}$	1	0	1	1
Not x	\bar{x}	1	1	0	0
Si x alors y	$\bar{x} \cdot y$	1	1	0	1
Nand	$\overline{x \cdot y}$	1	1	1	0
Constant 1	1	1	1	1	1

TABLE 1.2: Liste de toutes les fonctions booléennes de 2 variables. Il en existe $2^4 = 16$.

"And" (et) "x And y" vaut 1 exactement quand x et y valent 1. Cette opération est notée :

$$x \text{ and } y \equiv x \cdot y$$

"Or" (ou) "x Or y" vaut exactement 1 quand x ou y vaut 1. Cette opération est notée :

$$x \text{ or } y \equiv x + y$$

"Not" (non) "Not x" vaut exactement 1 quand x vaut 0. Cette opération se note :

$$\text{Not } x \equiv \bar{x}$$

La liste de toutes les fonctions booléennes possibles est donnée dans le tableau 1.2.

1.3 Représentation canonique

N'importe quelle fonction booléenne, aussi compliquée soit-elle peut s'exprimer en n'utilisant que trois opérateurs booléens : *et*, *ou* et *non*.

De plus, a minima, toute fonction booléenne peut toujours être écrite sous au moins une forme qui est appelée la *représentation canonique*.

On peut facilement obtenir cette représentation canonique en partant de la table de vérité (voir par exemple la table 1.1 pour fixer les idées). La méthode systématique est alors la suivante :

1. On ne regarde que les lignes de la table de vérité où la fonction booléenne (ici $f(x, y, z)$) vaut 1.

2. Pour chacune de ces lignes, on construit un terme en utilisant uniquement l'opérateur *et* (\cdot). Plus précisément, on applique l'opérateur *et* en utilisant x lorsque x égale 1 et en utilisant \bar{x} lorsque x égale 0.

Par exemple pour la troisième ligne, nous obtenons le terme $x \cdot \bar{y} \cdot z$.

3. Nous appliquons l'opérateur *ou* ($+$) à chacun de ces termes.

Dans le cas de la table 1.1 nous obtenons donc :

$$f(x, y, z) = x \cdot \bar{y} \cdot z + \bar{x} \cdot y \cdot z + \bar{x} \cdot \bar{y} \cdot z$$

Ainsi, il existe toujours une façon d'exprimer une fonction booléenne avec seulement les trois opérateurs booléens : *et*, *ou* et *non*. Ce n'est évidemment généralement pas la seule expression possible² et, généralement, la représentation canonique n'est pas la représentation la plus compacte d'une fonction booléenne. Ces écriture plus compact peuvent par exemple être obtenues via les théorèmes de Boole et de De Morgan.

2. En terme plus mathématiques, il n'y a pas unicité de la représentation algébrique d'une fonction booléenne

1.4 Deux théorèmes de l'algèbre booléenne

1.4.1 Les théorèmes de Boole

Les théorèmes de Boole sont donnés dans le tableau 1.3. Pour la plupart, il s'agit presque d'évidence.

Complémentarité	$a + \bar{a} = 1$, $a \cdot \bar{a} = 0$, $\bar{\bar{a}} = a$
Idempotence	$a + a + a + \dots = a$, $a \cdot a \cdot a \cdot \dots = a$
Éléments neutres	$a + 0 = a$, $a \cdot 1 = a$
Éléments absorbants	$a + 1 = 1$, $a \cdot 0 = 0$
Commutativité	$a + b = b + a$, $a \cdot b = b \cdot a$
Associativité	$(a + b) + c = a + (b + c) = a + b + c$, $(a \cdot b) \cdot c = a \cdot (b \cdot c) = a \cdot b \cdot c$
Distributivité	$(a + b) \cdot c = (a \cdot c) + (b \cdot c)$, $(a \cdot b) + c = (a + c) \cdot (b + c)$
Théorème d'absorption (1)	$a + (a \cdot b) = a$, $a \cdot (a + b) = a$
Théorème d'absorption (2)	$a \cdot \bar{b} + b = a + b$, $(a + \bar{b}) \cdot b = a \cdot b$
Théorème d'adjacence	$(a + \bar{b}) \cdot (a + b) = a$, $a \cdot \bar{b} + a \cdot b = a$

TABLE 1.3: Théorèmes de Boole

1.4.2 Les théorèmes de De Morgan

Ces deux théorèmes sont plus subtiles :

Théorème 1 :

$$\overline{(x + y)} = \bar{x} \cdot \bar{y}$$

Théorème 2 :

$$\overline{(x \cdot y)} = \bar{x} + \bar{y}$$

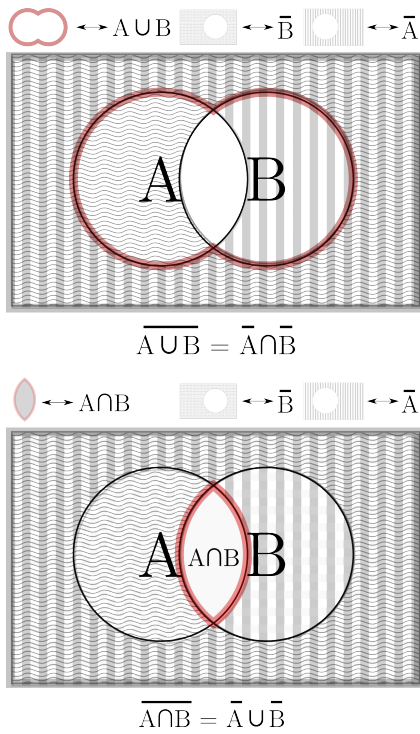


FIGURE 1.1: Illustration des théorèmes de De Morgan

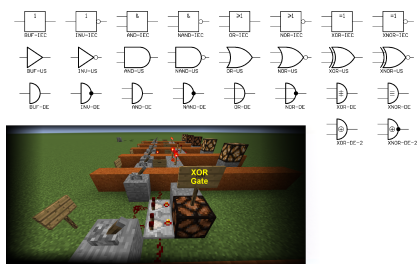


FIGURE 1.2: Ensemble de porte logique fabriquée à partir des règles de construction du jeu Minecraft

On peut noter la symétrie entre les deux théorèmes dans le rôle joué par l'opérateur et (\cdot) et l'opérateur ou ($+$).

Une illustration du théorème est présentée sur la figure 1.1.

1.5 Les portes logiques

Une porte est un dispositif physique qui implémente une fonction booléenne.

Il est important de noter que l'effet physique mise en jeu pour créer la porte logique n'a pas d'importance. Dès qu'une technologie permet de transporter l'information et d'avoir à disposition des interrupteurs, on peut l'utiliser pour créer des portes logiques. On utilise évidemment principalement les flux d'électrons (i.e. le courant électrique), mais on peut très bien aussi utilisé d'autre technologie comme par exemple des circuits optiques, magnétiques, pneumatique, hydrauliques, biologiques, etc (voir figure 1.2)

1.5.1 Porte NON (NOT)

La sortie d'une porte non prend le niveau logique opposé à celui de l'entrée.

Du fait que l'opérateur *non* est unaire, la porte non n'a qu'une entrée (et une sortie).

Le symbole de cette porte ainsi que la table de vérité sont donnés sur la figure 1.3

1.5.2 Porte OU (OR)

La sortie d'une porte OU est dans l'état 1 si *au moins une* de ses entrées est dans l'état 1.

1.5.3 Porte ET (AND)

La sortie d'une porte ET est dans l'état 1 si et seulement si *toutes* ses entrées sont dans l'état 1

1.5.4 Porte OU-EXCLUSIF (XOR)

La sortie d'une porte XOR est dans l'état 1 si *exactement une seule* de ses entrées est dans l'état 1.

1.5.5 Porte NON-ET (NAND)

La sortie d'une porte NON-ET (NAND) est au niveau 0 seulement si les deux entrées sont au niveau 1.







Nom	Symbole	Table de vérité	Nom	Symbole	Table de vérité																														
OR (ou)		<table border="1"> <thead> <tr> <th>x</th> <th>y</th> <th>S=x+y</th> </tr> </thead> <tbody> <tr><td>0</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>1</td><td>1</td></tr> <tr><td>1</td><td>0</td><td>1</td></tr> <tr><td>1</td><td>1</td><td>1</td></tr> </tbody> </table>	x	y	S=x+y	0	0	0	0	1	1	1	0	1	1	1	1	XOR		<table border="1"> <thead> <tr> <th>x</th> <th>y</th> <th>S=x^y</th> </tr> </thead> <tbody> <tr><td>0</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>1</td><td>1</td></tr> <tr><td>1</td><td>0</td><td>1</td></tr> <tr><td>1</td><td>1</td><td>0</td></tr> </tbody> </table>	x	y	S=x^y	0	0	0	0	1	1	1	0	1	1	1	0
x	y	S=x+y																																	
0	0	0																																	
0	1	1																																	
1	0	1																																	
1	1	1																																	
x	y	S=x^y																																	
0	0	0																																	
0	1	1																																	
1	0	1																																	
1	1	0																																	
AND (et)		<table border="1"> <thead> <tr> <th>x</th> <th>y</th> <th>S=x.y</th> </tr> </thead> <tbody> <tr><td>0</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>1</td><td>0</td></tr> <tr><td>1</td><td>0</td><td>0</td></tr> <tr><td>1</td><td>1</td><td>1</td></tr> </tbody> </table>	x	y	S=x.y	0	0	0	0	1	0	1	0	0	1	1	1	NAND		<table border="1"> <thead> <tr> <th>x</th> <th>y</th> <th>S=¬x.y</th> </tr> </thead> <tbody> <tr><td>0</td><td>0</td><td>1</td></tr> <tr><td>0</td><td>1</td><td>1</td></tr> <tr><td>1</td><td>0</td><td>1</td></tr> <tr><td>1</td><td>1</td><td>0</td></tr> </tbody> </table>	x	y	S=¬x.y	0	0	1	0	1	1	1	0	1	1	1	0
x	y	S=x.y																																	
0	0	0																																	
0	1	0																																	
1	0	0																																	
1	1	1																																	
x	y	S=¬x.y																																	
0	0	1																																	
0	1	1																																	
1	0	1																																	
1	1	0																																	
NOT (non)		<table border="1"> <thead> <tr> <th>x</th> <th>y</th> <th>S=¬x</th> </tr> </thead> <tbody> <tr><td>0</td><td>na</td><td>1</td></tr> <tr><td>na</td><td>na</td><td>na</td></tr> <tr><td>1</td><td>na</td><td>0</td></tr> <tr><td>na</td><td>na</td><td>na</td></tr> </tbody> </table>	x	y	S=¬x	0	na	1	na	na	na	1	na	0	na	na	na	NOR		<table border="1"> <thead> <tr> <th>x</th> <th>y</th> <th>S=¬x+y</th> </tr> </thead> <tbody> <tr><td>0</td><td>0</td><td>1</td></tr> <tr><td>0</td><td>1</td><td>0</td></tr> <tr><td>1</td><td>0</td><td>0</td></tr> <tr><td>1</td><td>1</td><td>0</td></tr> </tbody> </table>	x	y	S=¬x+y	0	0	1	0	1	0	1	0	0	1	1	0
x	y	S=¬x																																	
0	na	1																																	
na	na	na																																	
1	na	0																																	
na	na	na																																	
x	y	S=¬x+y																																	
0	0	1																																	
0	1	0																																	
1	0	0																																	
1	1	0																																	

FIGURE 1.3: Liste des principales portes logiques

La porte NAND est particulièrement prisée en recherche fondamentale et elle est parfois qualifiée de graal des portes logiques. En effet, avec uniquement des portes³ NAND, il est possible de réaliser toutes les autres portes logiques. Ainsi, la plupart des laboratoires en recherche fondamentale portent leur effort sur la mise au point, souvent à base de système nanoscopique, d'une porte NAND.

Un exemple d'une réalisation possible de porte NAND à partir de transistor est donnée sur la figure 1.4.

1.5.6 Porte NI (NOR)

La sortie d'une porte NI (NOR) est au niveau 1 seulement si les deux entrées sont au niveau 0.

1.5.7 Multiplexeur (MUX)

Un multiplexeur est une porte logique comportant trois entrées sel, a et b. Une de ces entrées, sel, est appelée le "sélecteur"⁴.

L'entrée sel de sélection permet de choisir si la sortie est fixée par la valeur a ou la valeur b.

La table de vérité du multiplexeur est donnée sur la figure 1.5.

1.5.8 Demultiplexeur (DMUX)

Un demultiplexeur est une porte logique possédant deux entrées, in et sel, et deux sorties, a et b. Elle effectue la fonction inverse d'un

3. On peut arriver au même résultats en n'utilisant que des portes NOR

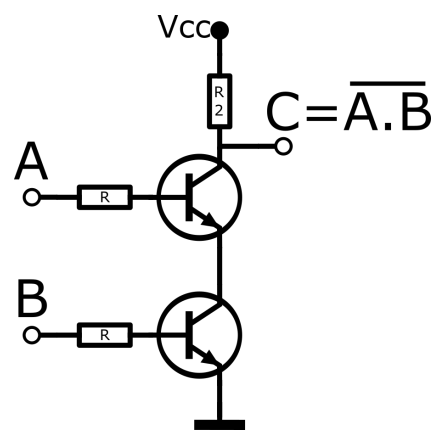


FIGURE 1.4: Schéma d'une porte NAND en utilisant des transistors. La tension en C vaut Vcc (i.e. la valeur booléenne 1) sauf si le transistor T1 et le transistor T2 sont passants (c'est à dire A=B=1) car alors C est relié à la masse.

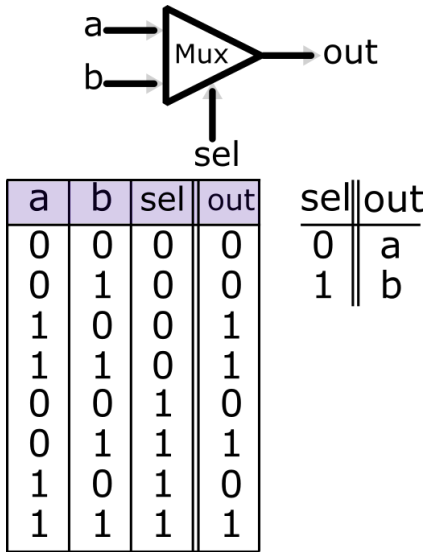


FIGURE 1.5: Schéma d'un multiplexeur, sa table de vérité et sa table de vérité simplifiée.

4. Un meilleur nom pour cette porte aurait d'ailleurs pu être un "selecteur". Le nom multiplexeur provient des télécommunications où il faut souvent faire passer successivement plusieurs signaux provenant de différentes sources sur un seul câble.

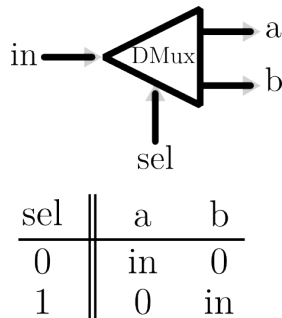


FIGURE 1.6: Schéma d'un demultiplexeur et sa table de vérité simplifiée.

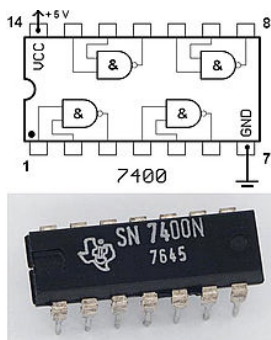


FIGURE 1.7: Image et schéma d'une puce 7400 qui contient 4 portes NAND.

5. Ainsi que des FPGA (Field-Programmable Gate Array)

multiplexeur. L'entrée sel de sélection permet d'aiguiller l'entrée vers la sortie a ou la sortie b.

La table de vérité du demultiplexeur est donnée sur la figure 1.6.

1.6 Mise en pratique

Nous traitons ici de l'utilisation pratique en électronique, mais rappelons l'universalité de la logique booléenne. Tout système permettant d'établir est apte à être utilisé pour effectuer de la logique booléenne.

Sauf cas extrêmement particulier on ne construit pas soi-même les portes logiques à partir de transistor. On peut par exemple directement utiliser des circuits intégrés comme par exemple ceux de la famille "74" (cf figure 1.7). On parle de *logique câblée*. Mais cette approche tend à disparaître pour être remplacé, comme nous le verrons dans la deuxième partie de ce cours, par l'utilisation de micro-contrôleur⁵. En d'autres termes, la logique câblée est remplacée par la logique programmée.

La logique programmée permet de se passer de nombreux câbles qui sont souvent sources d'erreur, peu fiables et surtout très peu flexibles (ou sens de non reprogrammable).

1.7 TP / TD

1.7.1 Quelques exercices autour de l'algèbre booléenne

Exercice 1

1. Simplifier l'expression :

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

2. Simplifier l'expression :

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

Exercice 2 Simplifiez les expressions suivantes en utilisant les théorèmes de De Morgan :

1.

$$S = \overline{\bar{A} \cdot B \cdot \bar{C}}$$

2.

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

3.

$$S = \overline{\bar{A} \cdot B \cdot \bar{C} \cdot D}$$

4.

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

5.

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

6.

$$S = \overline{\overline{A \cdot B \cdot CD}}$$

Exercice 3

1. Donner l'expression booléenne de la sortie S du schéma présenté sur la figure 1.8
2. Établir la table de vérité
3. Proposez un schéma simplifié.

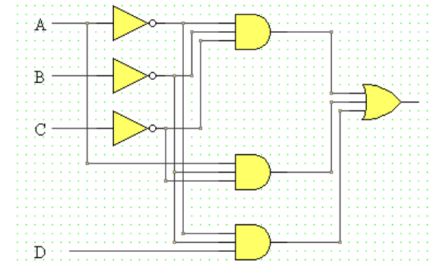


FIGURE 1.8: Exercice 3

Exercice 4 Soient 3 entrées ABC. La sortie S est vraie si la majorité des entrées est à 1.

1. Etablir la table de vérité de S
2. Etablir l'expression booléenne de S
3. Simplifier l'expression précédente
4. Réaliser le circuit sous logisim et vérifier son comportement.

XOR à 3 entrées La fonction logique OU exclusif (XOR) à 3 entrées n'est pas disponible dans les circuits intégrés disponibles sur le marché. On se propose de réaliser un circuit équivalent. Un OU exclusif (XOR) à 3 entrées n'a sa sortie S au niveau HAUT seulement si UNE SEULE des entrées A, B et C est au niveau HAUT, sinon S = 0.

1. Donner sa table de vérité
2. Donner l'expression logique et tenter de la simplifier
3. Donner le schéma lié à cette fonction.
4. Réaliser le montage correspondant avec des circuits de la famille 7400.

1.7.2 De l'universalité de la porte NAND

Avec comme seul composant des portes NAND créer un circuit (via logisim ou sur feuille) qui se comporte comme :

1. Une porte NON
2. Une porte ET
3. Une porte OU/XOR
4. Un multiplexeur/demultiplexeur.

2 Arithmétique Booléenne

Après avoir vu l'universalité de la logique booléenne dans le chapitre précédent, nous allons voir maintenant voir comment représenter en binaire des nombres algébriques (via la méthode du complément à 2) et fractionnaire (via l'écriture en virgule fixe).

Nous verrons ensuite comment additionner et multiplier deux nombres binaires. Nous implémenterons alors en porte logique un additionneur. Il s'agit d'une opération primordiale d'un ordinateur car d'un certain point de vue la plupart des opérations effectuées par un ordinateur peuvent être ramenées à l'addition de nombre binaire.

En suivant la démarche proposée par le livre "The element of computing system" Nisan and Schocken [2008], nous allons construire une Unité arithmétique et logique (ALU) rudimentaire mais relativement puissante.

2.1 Les nombres binaires

Soit $x = (x_n x_{n-1} \dots x_0)$ une suite de chiffres¹. La valeur de x en base b , qui s'écrit $(x)_b$ vaut :

$$(x_n x_{n-1} \dots x_0)_b = \sum_{i=0}^n x_i \cdot b^i$$

ce qui donne par exemple en base 2 :

$$(x_n x_{n-1} \dots x_0)_2 = \sum_{i=0}^n x_i \cdot 2^i \quad (2.1)$$

Ainsi le nombre $(0101)_2$ vaut :

$$(0101)_2 = 1 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2 + 0 \cdot 2^3 = 5$$

2.2 Addition binaire

L'addition binaire suit les mêmes règles que l'addition en base décimale.

1. Un chiffre est un caractère auquel on attribue une valeur entière positive. En base 2 on travaille typiquement avec les caractères 0 et 1, en base 10, 0,1,2,3,4,5,6,7,8, et 9 auquel on ajoute traditionnellement A,B,C,D,E et F en base 16 par exemple.

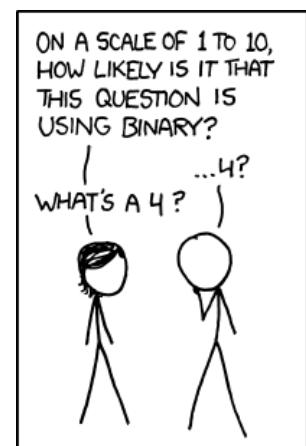


FIGURE 2.1: Xkcd - 1 to 10 - <https://xkcd.com/953/> - Randall Munroe

(1)	(1)	(1)		
	0	1	1	0
+	1	0	1	1
	1	0	0	1

TABLE 2.1: Exemple de l'addition de 6 + 11 = 17

2. Il existe bien d'autre méthode, on pourrait par exemple penser à dédier le bit de poids fort au signe et laisser les autres bits intacts. Ainsi par exemple 0101 serait 5 et 1101 serait -5. Cette méthode est simple à intégrer mais par contre elle ne conserve pas l'opération d'addition binaire alors que cela est le cas pour le complément à 2

2.3 Nombre binaire algébrique

Nous allons utiliser la méthode qui est de nos jours la plus commune² : le complément à 2.

Le complément à 2 est mathématiquement décrit par :

$$\bar{x} = \begin{cases} 2^n - x & \text{si } x \neq 0 \\ 0 & \text{sinon} \end{cases} \quad (2.2)$$

Le tableau 2.2 donne un comptage binaire avec 4 bits en prenant en compte le complément à 2.

TABLE 2.2: Représentation des nombres algébriques binaires à 4 bits en prenant en compte le complément à 2

Nbre positifs	Nbre négatif
0	0000
1	0001 1111 -1
2	0010 1110 -2
3	0011 1101 -3
4	0100 1100 -4
5	0101 1011 -5
6	0110 1010 -6
7	0111 1001 -7
	1000 -8

- On peut coder 2^n nombres allant de -2^{n-1} à $2^{n-1} - 1$
- Le code des nombres positif commence par un 0
- Le code des nombres négatifs commence par un 1
- On obtient le code de $-x$ à partir de celui de x à partir de la méthode suivante :
 1. En partant des bits de poids faible (ceux à droites), passer tous les 0 jusqu'à arriver au premier 1.
 2. Inverser alors tous les bits suivants jusqu'au bit de poids de fort.

3. Vous pouvez en effet vérifier avec le tableau 2.2, que lorsque l'on inverse les bits d'un nombre binaire ($x \leftrightarrow \bar{x}$, on obtient, dans le cadre du complément à 2 la valeur $-x - 1$. Écrit de façon plus compact :

$$\bar{x} \leftrightarrow -x - 1$$

On peut aussi, et cela est plus facile à implémenter avec des portes logiques, inverser tous les bits³ de x et ajouter 1.

Un des grand avantage du complément à 2 et qu'il conserve l'addition binaire sans avoir à faire attention si les nombres sont positifs ou négatifs. Ainsi, d'un point de vue matériel, il n'y a pas à rajouter de porte logique pour prendre en compte les nombres négatifs.

2.4 Représentation d'un nombre en virgule fixe

Il existe principalement⁴ deux méthodes pour écrire en binaire des nombres réels :

- La représentation en virgule flottante.
- La représentation en virgule fixe.

4. Citons aussi la représentation "Décimal codé binaire" (BCD) qui est facile à lire mais qui n'est pas la plus compacte en terme d'espace mémoire.

2.4.1 La représentation en virgule flottante

La représentation en virgule flottante est, par exemple, celle qui est utilisée pour stocker un "float" en langage C. Nous n'allons pas spécialement rentrer dans les détails de cette représentation car nous n'allons pas l'utiliser par la suite. Néanmoins, en quelques mots, il s'agit de transcrire en binaire la notation scientifique. On répartit les bits (par exemple les 32 bits pour une variable de type float) de la manière suivante :

- Le premier bit code pour le signe de l'exposant
- Les 8 bits suivants codent pour l'exposant de la puissance de dix
- et les 23 bits restants (mantisse ou significande) code, en binaire complété à deux, la partie décimale

Additionner ou multiplier de tels nombres demande, d'une part, de séparer l'exposant et la mantisse et ensuite d'effectuer plusieurs manipulations de ces quantités, et enfin de recoder le nombre obtenu en représentation en virgule flottante. Cela demande beaucoup⁵ d'instructions au processeur, même pour une simple addition, si bien qu'il faut souvent une puce (ou une partie de puce) dédiée à la gestion de ces calculs, on parle d'unité de calcul en virgule flottante.

5. Par exemple, pour un micro-contrôleur de type ARM cortex M3, une addition de deux nombres en virgule flottante (float ou double) demande 60 fois de temps de calcul qu'une addition de deux nombres entiers (int)

Le processeur que nous allons mettre au point dans la suite de ce cours ne gèrera pas la représentation en virgule flottante.

2.4.2 La représentation en virgule fixe

Comme son nom l'indique, le principe est de fixer un nombre donné de bits après la virgule. Par exemple, pour un nombre codé sur 16 bits, on peut considérer que 8 bits (ceux de poids faible), sont alloués à la représentation de la partie décimale.

6. Rappelons que l'endroit où on place la virgule est *arbitraire*. Par contre, évidemment, une fois ce choix fait, il faut s'y tenir.

Ainsi, les 8 bits de poids fort⁶ codent pour la partie *entière* alors que les 8 bits de poids faible codent pour la partie *décimale* (après la virgule). Soit un nombre n codé en binaire sous 16 bits :

$$a_1a_2a_3a_4 \quad a_5a_6a_7a_8 \quad b_1b_2b_3b_4 \quad b_5b_6b_7b_8$$

7. Nous utilisons ici, implicitement, un facteur d'échelle (scaling factor) de 2, on peut parfois aussi utiliser un facteur d'échelle de 10.

alors sa représentation en décimale sera ici⁷ :

$$\begin{aligned}
 n &= a_1 \times 2^7 + a_2 \times 2^6 + a_3 \times 2^5 + a_4 \times 2^4 \\
 &\quad + a_5 2^3 + a_6 \times 2^2 + a_7 \times 2^1 + a_8 \times 2^0 \\
 &+ \\
 &\quad b_1 \times 2^{-1} + b_2 \times 2^{-2} + b_3 \times 2^{-3} + b_4 \times 2^{-4} \\
 &\quad + b_5 2^{-5} + b_6 \times 2^{-6} + b_7 \times 2^{-7} + b_8 \times 2^{-8}
 \end{aligned}
 \tag{2.3}$$

Par exemple le nombre :

0000 1001 1001 1110

vaut :

$$2^3 + 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 0 \times 2^{-3} + 1 \times 2^{-4} + 2^{-5} + 2^{-6} + 2^{-7} = 9.617$$

Mathématiquement, pour un nombre X de N bits dont chaque bit est noté a_i et dont n_e bits codent la partie entière et n_f la partie fractionnaire, la valeur de nombre X est donnée par :

$$X = \sum_{i=0}^{N-1} a_i 2^{i-n_f} \tag{2.4}$$

Ce qui peut se réécrire :

$$X = 2^{-n_f} \sum_{i=0}^{N-1} a_i 2^i = 2^{-n_f} X_{\text{entier}} \tag{2.5}$$

où X_{entier} est la valeur du nombre en base 2 lorsque l'on ne tient pas compte d'une virgule fixe (cf equation 2.1). Il s'agit ainsi d'un moyen rapide d'obtenir rapidement la valeur d'un nombre en virgule fixe ou inversement son écriture. Reprenons l'exemple du nombre :

0000 1001 1001 1110

La valeur X_{entier} est⁸ de 2462. En plaçant comme auparavant la virgule sur le 8ème bit de poids fort, le nombre de bit n_f codant la partie fractionnaire vaut $n_f = 8$. La valeur décimale associée, en vertu de l'équation 2.5, est donc :

$$0000 1001 1001 1110 \rightarrow 2^{-n_f} X_{\text{entier}} = 2^{-8} \times 2462 = 9.617875$$

Inversement, si on veut coder le nombre décimale $\sqrt{2} \approx 1.41421$ avec la représentation en virgule fixe précédente, on divise ce nombre par 2^{-n_f} , en l'occurrence 2^{-8} , et on écrit en binaire la partie entière du nombre obtenu. Cela donne ici :

$$1.41421 \rightarrow 1.41421 \times 2^8 = 362 \rightarrow 0000 0001 0110 1010$$

On peut alors vérifier le résultat, en effet la partie décimale étant les 8 bits de poids faible soit 0110 1010 ce qui donne comme valeur (cf eq 2.3) :

$$2^{-2} + 2^{-3} + 2^{-5} + 2^{-7} = 0.25 + 0.125 + 0.03125 + 0.007812 = 0.414063$$

ce qui est bien la représentation la plus exacte sur 8bits de la partie fractionnaire du nombre $\sqrt{2} \approx 1.41421$

8. Vous pouvez trouver en ligne ou sur la calculatrice livrée avec windows en mode "programmeur" des outils pour passer rapidement de la notation binaire (et hexadécimale) à la notation décimale

Avantage et inconvénient de cette représentation Une telle représentation permet de conserver les règles du complément à deux de l'addition binaire. Cependant, elle présente deux principaux écueils :

- Il y a une perte importante de précision sur la partie décimale.
- Il faut faire plus attention au risque de débordement (overflow) de la partie entière. Par exemple, pour le nombre codé en 16 bits dont 8 bits sont dédiés à la partie décimale, il ne reste plus que 8 bits pour la partie entière soit une valeur maximale de 255 on un intervalle $[-128, 127]$ si on prend en compte un complément à deux.

2.4.3 La multiplication

La multiplication suit les mêmes règles qu'en notation décimale.

$$\begin{array}{r}
 \\
 \\
 \\
 \\
 \\
 \\
 \\
 \hline
 \\
 \\
 \\
 \\
 \\
 \\
 \\
 \hline
 0 1 0 0 0 0 1 0
 \end{array}$$

TABLE 2.3: Exemple de la multiplication de $6 \times 11 = 66$

La multiplication de deux nombres de n bits conduit à un résultat sur $2n$ bits.

Lorsque le nombre est codé avec une virgule fixe, avec la virgule placée sur le bits n_f , la virgule après multiplication est déplacée vers la gauche de $2n_f$ bits. Par exemple si on multiplie 2 nombres codés sur 8 bits dont la virgule est placée sur le 4ème bits, le résultats de la multiplication sera du 16 bits et la virgule sur le bits $2 \times 4 = 8$. Autre exemple, si multiplie deux nombres codés sur 16bits, dont la virgule est placée sur le 15ème bits⁹, le résultat est sur 32bits avec la virgule placée sur le bit numéro $2 \times 15 = 30$.

A noter : une multiplication par 2 correspond à un décalage des bits vers la gauche.

2.4.4 La division par 2

Une division par 2 correspond à un décalage des bits vers la droite.

2.5 Les additionneurs

On cherche à réaliser l'opération d'addition binaire avec des portes logiques¹⁰.

Nous allons décomposés l'addition en 3 sous opérations :

- Un demi additionneur (*half-adder*) qui additionne deux bits

9. Il s'agit d'un choix assez pertinent lorsque l'on sait que les nombres à coder sont inférieurs à 1

10. C'est à dire, i fine, uniquement avec des portes NAND, puisque nous avons vu dans le chapitre précédent que n'importe quelle porte peut être réalisée avec uniquement des portes NAND.

- Un additionneur complet (*full-adder*) qui additionne trois bits
- L'additionneur qui additionne deux nombres de n bits.

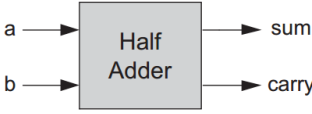
2.5.1 Le demi additionneur

Cette puce, dont le cahier des charges est donné sur la figure 2.2, réalise l'addition binaire des 2 bits d'entrée a et b . La puce possède deux sorties sum (la somme) et $carry$ (retenue en français). Bien qu'il y ait besoin de 4 combinaisons pour lister les couples de valeurs d'entrée a et b , le résultat lui ne peut prendre que trois valeurs : 0, 1 et 2. Le résultat de l'addition est géré par les deux bits de sortie :

- sum s'occupe des cas 0 et 1
- si les deux bits a et b sont égaux à 1, le résultat (10) ne peut pas être stocké sur 1 seul bits. On utilise alors le bits de sortie $carry$ pour transmettre ce dépassement.

Dit autrement, le bit de sortie sum représente le bit de poids faible de l'addition et le bit de sortie $carry$, le bits de poids fort de l'addition.

Inputs		Outputs	
a	b	carry	sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0



```

Chip name: HalfAdder
Inputs:    a, b
Outputs:  sum, carry
Function: sum  = LSB of a + b
              carry = MSB of a + b
    
```

FIGURE 2.2: Cahier des charges du demi additionneur

2.5.2 L'additionneur complet

Cette puce, dont le cahier des charges est donné sur la figure 2.3, réalise l'addition de trois nombres binaires. Ces 3 nombres seront, dans la suite, d'une part deux nombres binaires à additionner et d'autre part la carry d'un autre additionneur complet.

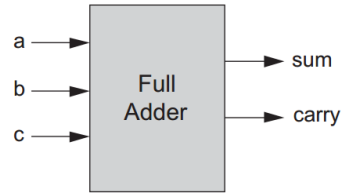
Bien que cela ne soit pas indispensable, nous implémenterons cette puce à partir du demi additionneur.

2.5.3 L'additionneur

Le cahier des charges de cette puce est donné sur la figure 2.5.3.

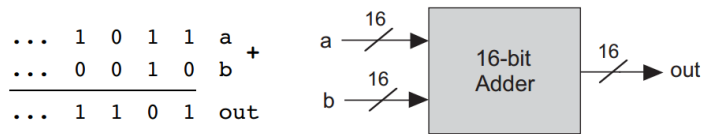
Pour fixer les idées, on va construire un additionneur 8bits. En d'autre termes, deux nombres binaires, A et B , codés sur 8bits, sont addi-

a	b	c	carry	sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



Chip name: FullAdder
Inputs: a, b, c
Outputs: sum, carry
Function: sum = LSB of a + b + c
 carry = MSB of a + b + c

FIGURE 2.3: Cahier des charges de l'additionneur complet



Chip name: Add16
Inputs: a[16], b[16]
Outputs: out[16]
Function: out = a + b
Comment: Integer 2's complement addition.
 Overflow is neither detected nor handled.

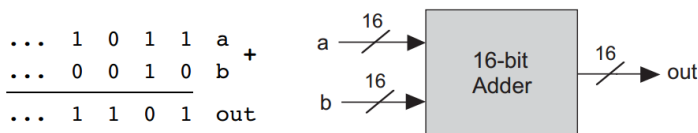
11. si $A + B > 255$ sans complément à deux ou $A + B < -128$ ou $A + B > 127$ avec un complément à deux

tionnés pour donner un nombre binaire C codé sur 8bits lui aussi. Un éventuel dépassement¹¹ n'est ni détecté, ni géré par la puce.

Nous implémenterons cette puce via l'utilisation en cascade de plusieurs additionneur complets.

2.5.4 Incrémenteur

Cette puce est un cas particulier d'additionneur, son cahier des charges est donné sur la figure 2.4.



```

Chip name: Add16
Inputs:    a[16], b[16]
Outputs:  out[16]
Function: out = a + b
Comment:  Integer 2's complement addition.
               Overflow is neither detected nor handled.
    
```

FIGURE 2.4: Cahier des charges de l'incrémenteur

12. On peut a priori coder $2^6 = 64$ fonctions pour l'ALU. L'implémentation prévu dans la référence Nisan and Schocken [2008] se restreint à documenter 18 fonctions réalisant un bon compromis entre polyvalence et simplicité d'implémentation et d'utilisation. Dit autrement, l'ALU, sans aucune autre modification est capable de réaliser d'autre opérations non listés ici.

2.6 L'unité Arithmétique et Logique (ALU)

Toutes les puces que nous avons réalisées jusqu'à maintenant sont génériques et se retrouve dans n'importe qu'elle processeur. L'unité Arithmétique et Logique (ALU), qui va être la puce centrale du processeur est elle spécifique à chaque architecture. Nous suivons ici l'architecture proposé par NISAN et SCHOCKEN dans leur livre "The Elements of Computing Systems : Building a Modern Computer from First Principles" Nisan and Schocken [2008]. Cette architecture, a visée pédagogique, a pour but d'offrir le maximum de fonctionnalités avec un nombre minimum de porte logique.

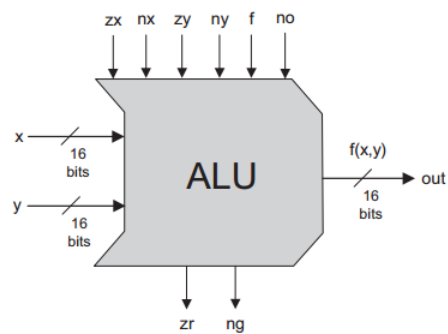
Le cahier des charges de la puce est donnée sur la figure 2.5.

La puce à 8 entrées :

- Deux entrée 16 bits, x et y sur lesquelles sont réalisés les opérations de l'ALU
- Six entrée de contrôle chacune sur 1 bit. Ces entrées indiquent quelle opération doit être réaliser par l'ALU parmi les 18 prévus¹².

La puce à 3 sorties :

- $f(x, y)$ codé sur 16 qui est le résultat du calcul de l'ALU avec les entrées x et y
- Deux sorties de contrôle chacune sur 1 bit.



```

Chip name: ALU
Inputs:  x[16], y[16],    // Two 16-bit data inputs
            zx,             // Zero the x input
            nx,             // Negate the x input
            zy,             // Zero the y input
            ny,             // Negate the y input
            f,              // Function code: 1 for Add, 0 for And
            no              // Negate the out output
Outputs: out[16],      // 16-bit output
            zr,             // True iff out=0
            ng              // True iff out<0
Function: if zx then x = 0      // 16-bit zero constant
            if nx then x = !x     // Bit-wise negation
            if zy then y = 0     // 16-bit zero constant
            if ny then y = !y    // Bit-wise negation
            if f then out = x + y // Integer 2's complement addition
                else out = x & y // Bit-wise And
            if no then out = !out // Bit-wise negation
            if out=0 then zr = 1 else zr = 0 // 16-bit eq. comparison
            if out<0 then ng = 1 else ng = 0 // 16-bit neg. comparison
Comment: Overflow is neither detected nor handled.

```

FIGURE 2.5: Cahier des charges de l'ALU

14. La soustraction $x - y$ est réalisé via l'opération :

$$x - y \leftrightarrow !(!(x) + y) \leftrightarrow \overline{\overline{x} + y}$$

En effet, rappelons comme mentionnée dans une note précédente que, dans le cadre du complément à 2 :

$$\overline{\overline{x}} (=!x) \leftrightarrow -x - 1$$

L'opération $\overline{\overline{x} + y}$ conduit donc à $-x - 1 + y$ et prendre l'inverse (opérateur !) de ce résultat :

La liste des fonctions¹³ réalisées par l'ALU est donnée dans la figure 2.6. Parmi ces instruction se trouve l'addition, la soustraction¹⁴, la négation, le "et" et le "ou" logique, l'incrémenter mais pas la multiplication ou la division. Les opérations manquantes¹⁵ seront implémentés à partir des 18 fonctions de base de l'ALU dans le chapitre 4. En d'autre termes, il n'y pas d'implémentation hardware (i.e physique) de la multiplication mais une implémentation software (via la programmation d'une séquence d'opérations).

	These bits instruct how to preset the x input		These bits instruct how to preset the y input		This bit selects between + / And	This bit inst. how to postset out	Resulting ALU output
	zx	nx	zy	ny	f	no	out=
15	if zx then x=0	if nx then x=!x	if zy then y=0	if ny then y=!y	if f then out=x+y else out=x&y	if no then out=!out	f(x,y)=
	1	0	1	0	1	0	0
	1	1	1	1	1	1	1
	1	1	1	0	1	0	-1
	0	0	1	1	0	0	x
	1	1	0	0	0	0	y
	0	0	1	1	0	1	!x
	1	1	0	0	0	1	!y
	0	0	1	1	1	1	-x
	1	1	0	0	1	1	-y
	0	1	1	1	1	1	x+1
	1	1	0	1	1	1	y+1
	0	0	1	1	1	0	x-1
	1	1	0	0	1	0	y-1
	0	0	0	0	1	0	x+y
	0	1	0	0	1	1	x-y
	0	0	0	1	1	1	y-x
	0	0	0	0	0	0	x&y
	0	1	0	1	0	1	x y

FIGURE 2.6: Table de vérité l'ALU

2.7 TD 2

2.7.1 Complément à 2

1. Quelle est la valeur maximale et la valeur minimale que l'on peut atteindre avec 5 chiffres binaires tout en prenant en compte le complément à 2.
2. Dresser le tableau de correspondance de ces 32 nombres binaires avec leur équivalent en base décimale

3. Vérifier que :

(a) $8 + (-11) = -3$

(b) $-5 + (13) = 8$

Représentation d'un nombre en virgule fixe

1. Donner la plus exacte approximation du nombre π sur 8 bits.
2. Effecteur, sur 8 bits et sans complément à deux, le calcul $8.31 + 2.42$.

2.7.2 Additionneur

En suivant les indications données dans les sections 2.5.1, 2.5.2, 2.5.3 et 2.5.4, construire avec logisim les puces réalisant : le demi-additionneur, l'additionneur complet, l'additionneur 8 bits et l'incrémenteur.

2.7.3 ALU

En suivant les indications données dans les sections 2.6 construire avec logisim l'ALU que nous allons utiliser par la suite.

NB : La transcription d'un "if" (si logique) est, en terme de porte logique un multiplexeur.

3 Logique séquentielle

Dans ce chapitre, nous cherchons à construire avec des portes logiques des circuits capables de conserver en mémoire des variables booléennes.

Les circuits vus dans les deux derniers chapitres présentent des sorties qui ne dépendent pas de la valeur des entrées. Le "passé", c'est-à-dire l'état antérieur n'est pas pris en compte. Il est donc impossible d'envisager la fonction mémoire.

Les circuits logiques séquentiels, à l'inverse des circuits combinatoires, sont des circuits dont la réaction ou l'état dépend des entrées mais aussi des événements précédemment apparus dans le temps.

3.1 Quelques généralités

Le fait de retenir une information (en l'occurrence ici un état "0" ou un état "1") est intrinsèquement lié à la notion de *temps*. Il faut pouvoir définir un *maintenant* où on veut avoir retenu l'information qui était présente *avant*.

3.1.1 Horloge

Dans la plupart des ordinateurs le passage du temps est matérialisé par une horloge qui délivre à intervalle régulier des impulsions électriques. Ces horloges sont souvent construites autour de quartz¹ taillé de telle sorte qu'il vibre à une fréquence bien définie². Une fois inséré dans un circuit électrique (par exemple celui de la figure 3.2), l'ensemble délivre des impulsions électriques. Ces impulsions sont envoyées à tous sous-circuits du processeur qui ont besoin d'une référence temporelle (typiquement les mémoires).

Entre les impulsions de l'horloge l'état de l'ordinateur est indéfini. Plus précisément, les électrons sont encore en mouvement au sein de toutes les portes logiques inter-connectées. Durant ce laps de temps, l'ALU que nous avons mis au point au chapitre précédent donne n'importe quoi comme résultat.

La fréquence maximale que peut prendre l'horloge doit donc être telle que l'intervalle entre deux tics laisse le temps, d'une part au signal d'horloge d'arriver aux différentes portes³ et d'autre part aux élec-

1. Le quartz est un milieu piezzo-electrique. D'un point de vue cristallographique le barycentre des charges + et des charges - d'une maille élémentaire peuvent ne pas coïncider. Ainsi, lorsque l'on comprime un tel cristal, une tension apparaît à ses bornes. A contrario, lorsqu'on le soumet à une impulsion électrique, le cristal piezzo-electrique se comprime puis se met à osciller après la fin de l'oscillation. Plus exactement, suite à cette excitation impulsionnelle, il vibre à sa pulsation de résonance avec un excellent facteur de qualité Q (typiquement $Q \approx 100\,000$)
2. Vous pouvez voir par exemple sur un arduino un cristal (qui prend l'aspect d'un boîtier métallique oblong) où il est inscrit "T16.000" pour indiquer sa fréquence de vibration à 16 MHz



FIGURE 3.1: Photo d'un boîtier contenant un quartz taillé de telle sorte à vibrer à une fréquence extrêmement précise.

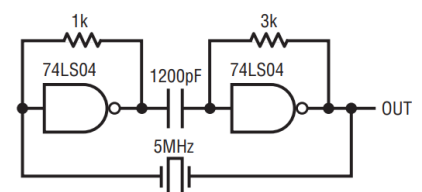


FIGURE 3.2: Exemple d'un circuit rudimentaire mettant en jeu un oscillateur à quartz et des portes logiques pour produire un signal d'horloge

3. Étonnamment, c'est ce point qui limite la montée en fréquence des processeurs (mais pas pour autant en puissance de calculs) depuis ces quinze dernières années. En effet, les liaisons métalliques au sein du processeur ont nécessairement une résistance et un effet capacitif entre elles ce qui crée un temps de retard $\tau = RC$. L'utilisation de photon à la place d'électron (un processeur optique) permettrait de ne plus avoir ce temps de retard et d'augmenter considérablement la fréquence des processeurs (et aussi diminuer leur consommation). Il s'agit d'un projet de recherche toujours actuel mais encore au stade du laboratoire.

trons au sein des portes d’atteindre leur position d’équilibre.

En 2021, la fréquence maximale des processeurs grand public est de l’ordre de 5 GHz. Donc, tous les 200 ps, les électrons s’agitent au sein des portes, trouvent leur position d’équilibre, puis on collecte les résultats données par les circuits (comme l’ALU par exemple) et on repart pour un autre cycle d’horloge . . . et tout cela 5 milliards de fois par seconde !

3.1.2 Bascules

Les *bascules* (ou *flip-flop* en anglais) sont les briques de bases des composants liés à la mémoire. Leur mission première est de pouvoir donner à un temps t en signal de sortie, la valeur qui était en entrée à temps $t_0 < t$. Dit autrement, son rôle consiste à enregistrer une information non permanente puis à conserver cet état lorsque l’information disparaît. Nous allons voir par la suite leur fonctionnement et comment les implémenter avec des portes logiques.

Ces bascules peuvent être synchrones ou asynchrones. Une bascule asynchrone n’est reliée à aucun système capable de mesurer le défilement du temps. Ces bascules peuvent donc changer d’état à tout moment. A contrario, une bascule synchrone est reliée au signal d’horloge. La bascule ne peut changer d’état que lors de l’arrivée d’une impulsion d’horloge. Ce dernier est alors utilisé pour synchroniser l’ensemble des bascules reliés à l’horloge. Concrètement, une bascule synchrone possédera une entrée pour recevoir le signal de l’horloge là où cela ne sera pas le cas pour une bascule asynchrone.

3.1.3 Compteurs

Un compteur est une puce séquentielle qui garde en mémoire un entier et qui s’incrémente à chaque nouvelle pulse d’horloge. Ces puces jouent un rôle important dans l’architecture d’un ordinateur⁴ par exemple pour savoir quelle instruction exécuter au tick d’horloge suivant.

4. Mais aussi d’un microprocesseur. Un arduino uno possède 3 compteurs aussi appelés timer.)

3.2 Implémentation Bascules à partir de portes logiques

3.2.1 Boucler des portes logiques

La bascule doit réaliser la fonction $out(t) = in(t_0)$ avec $t > t_0$. Elle doit donc, d’une façon où d’une autre relié le signal à son entrée au signal en sortie. On parle alors de *boucle* (ou *feedback* en anglais).

On pourrait alors naïvement essayer de boucler la sortie d’une porte logique vers son entrée. Cela conduit à un état indéfini du système (voir figure 3.3).

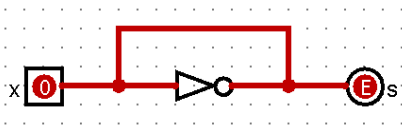


FIGURE 3.3: Bouclage d’une porte NOT. La sortie ne sait pas si elle doit prendre la valeur en sortie de porte, \bar{x} , où la valeur en entrée x .

Il faut utiliser au moins deux portes pour mettre en place le bouclage des portes logiques permettant de conserver un état logique.

3.2.2 Bascule RS

La bascule RS (pour Reset and Set) est présentée sur la figure 3.4. Il s'agit d'une bascule asynchrone car elle fait pas intervenir de signal d'horloge. Étudions son fonctionnement.

La bascule utilise deux portes NAND⁵ présentant un bouclage : une des deux entrées d'une porte NAND est reliée à la sortie de l'autre porte NAND. Elle possède deux entrées R (Reset) et S (Set) et deux sorties Q et \bar{Q} . Les signaux Q et \bar{Q} de sortie dépendent des deux entrées R et S *mais aussi* de l'état antérieur de Q.

La table de vérité de la bascule est donnée sur la figure 3.5.

On distingue quatre cas de figure :

- Il existe pour la sortie Q un état "interdit" qui est obtenu lorsque $R = S = 0$. En effet, comme cela est montré sur la figure 3.4, la sortie \bar{Q} de la bascule n'est pas égale à la valeur opposée de Q.
- L'état de "mémoire" est obtenu pour $R = S = 1$. La sortie reste égale à la valeur Q qui était déjà en mémoire.
- La mise à zéro ("reset") de la sortie Q (et donc $\bar{Q} = 1$) pour $R = 1$ et $S = 0$
- La mise à 1 ("set") de la sortie Q pour $R = 0$ et $S = 1$.

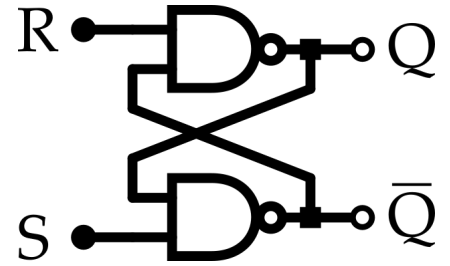


FIGURE 3.4: Bascule RS implémentée avec des portes NAND

R	S	Q	
0	0	X	interdit
0	1	1	mise à 1
1	0	0	mise à 0
1	1	Q	mémorisation

FIGURE 3.5: Table de vérité de la Bascule RS

5. On peut aussi l'implémenter avec deux portes NOR

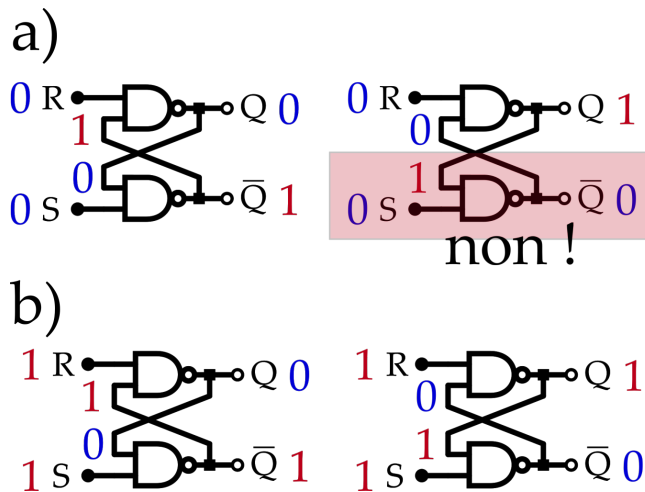


FIGURE 3.6: a) État interdit de la bascule RS. La porte NAND indiquée en rouge ne satisfait pas sa table de vérité. b) Mise en mémoire ($R = S = 1$) pour la bascule RS. Les deux cas de figure (selon les valeurs précédentes de la sortie Q) sont stables. Elle peut servir parfois à filtrer les rebonds d'interrupteur mécanique.

Nous n'utiliserons pas par la suite la bascule RS car, du fait qu'elle soit asynchrone, il n'est pas possible de faire fonctionner en parallèle et de façon synchronisées plusieurs de ces bascules⁶. Elle constitue néanmoins la brique de base de la bascule D qui est, elle, très utilisée pour les montage synchrone.

3.2.3 Bascule D

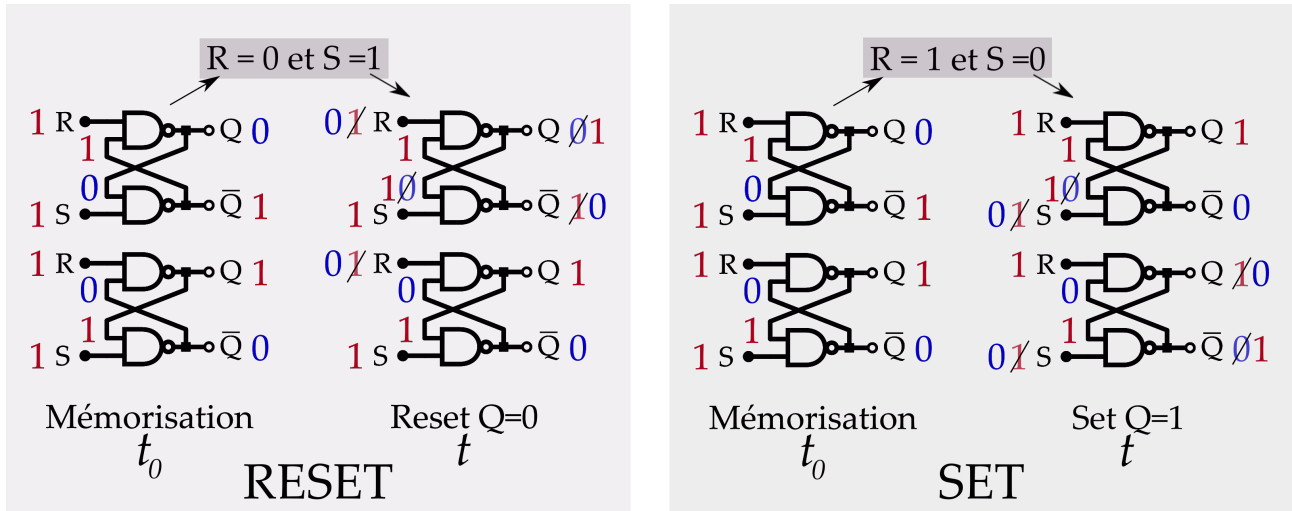


FIGURE 3.7: Etat "ReSet" et "Set" de la bascule RS. Quelque soit l'état précédent de la bascule (dans l'état de mémorisation), la configuration $R = 1, S = 0$ ("Reset") conduit à $Q = 0$ et la configuration $R = 0, S = 1$ ("Set") conduit à $Q = 1$.

CLK	D	Q
0	X	Q_0
1	0	0
1	1	1

inchangé

FIGURE 3.8: Table de vérité de la Bascule D

La bascule D est synchrone, son implémentation en porte NAND est donnée sur la figure 3.9 et sa table de vérité sur la figure 3.8.

Il s'agit en fait d'une bascule RS avec un amont un système permettant de synchroniser l'entrée D (comme "Data") avec le signal d'horloge :

- Lorsque D vaut 1 cela revient à faire "set" ($S = 0$ et $R = 1$) sur la bascule RS qui garde donc en mémoire 1.
- Lorsque D vaut 0 cela revient à faire "reset" ($R = 0$ et $S = 1$) sur la bascule RS qui garde donc en mémoire 1.

Il s'agit de la brique de base des mémoires que l'on va utiliser par la suite.

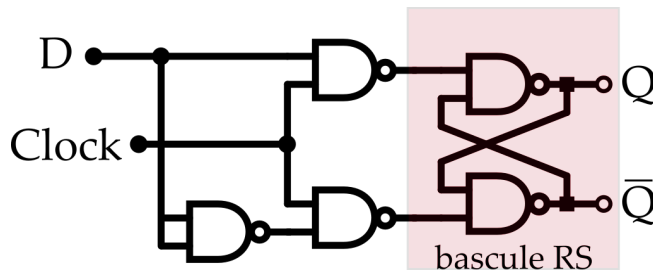


FIGURE 3.9: Bascule D implémentée avec des portes NAND. On peut identifier une bascule RS précédé d'un circuit permettant de synchroniser l'entrée D (comme "Data") avec le signal d'horloge (clock). Cette bascule mémorise la valeur de D lorsque l'horloge (clock) vaut 1 (il n'y pas ici de prise en compte de front montant ou descendant).

3.3 Implémentation de la mémoire à partir de bascule

3.3.1 Registre

Le cahier de charge d'un registre est donné sur la figure 3.10. Le registre a pour but de stocker un seul bit d'information (un seul "0" ou "1"). Il possède une entrée et une sortie qui contiennent des données codées sur 1bit. La sortie "out" émet l'état actuel du registre. Une

entrée de contrôle "load" indique si on doit écrire dans la mémoire l'état de l'entrée "in".

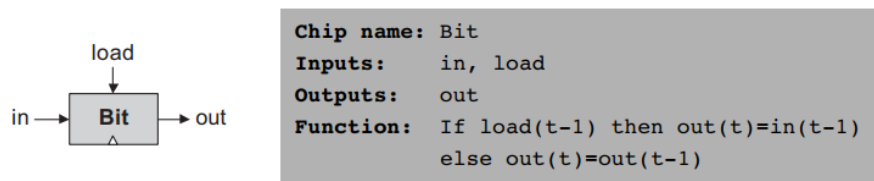


FIGURE 3.10: Cahier des charges d'un registre

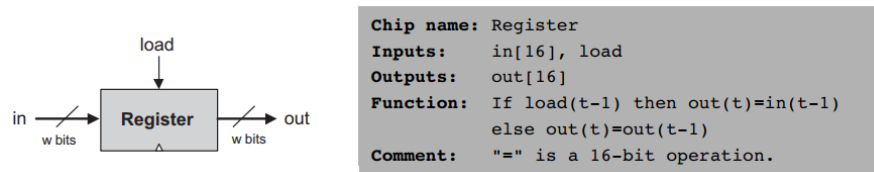


FIGURE 3.11: Cahier des charges d'un registre 16 bits

3.3.2 Random Access Memory

Une mémoire à accès aléatoire⁷ (RAM pour Random Access Memory) est une banque de registre munie d'un circuit d'aiguillage permettant d'accéder à un de ces registres pour ou lire ou écrire des données. Chaque registre possède un adresse unique. Muni de cette adresse le circuit d'aiguillage (d'adressage) permet de s'adresser au registre souhaité (que ce soit pour écrire une donnée dedans et/ou lire son contenu). Comme on peut le voir sur le cahier des charges (fig 3.13), la RAM possède deux entrées 16 bits, une première pour l'adresse permettant donc de choisir le registre et la deuxième entrée 16 bits pour donner une éventuelle valeur à insérer dans le registre indiqué par l'adresse. En sortie, on obtient la valeur présente dans le registre indiqué par l'adresse.

7. Dans une telle mémoire, contrairement à un disque dur ou un disque optique, le temps d'accès à un élément de mémoire est le même pour tous les éléments.

3.3.3 Compteur

Le cahier des charges d'un compteur est donné sur la figure 3.14.

3.4 TD 3

3.4.1 bascule D

Reproduire via logisim le schéma de la Bascule D (fig 3.9) et vérifier qu'il correspond bien à sa table de vérité.

3.4.2 Registre

En suivant son cahier des charges (fig 3.10) réalisé un registre un bit. Par soucis de performance, on utilisera la bascule D de logisim

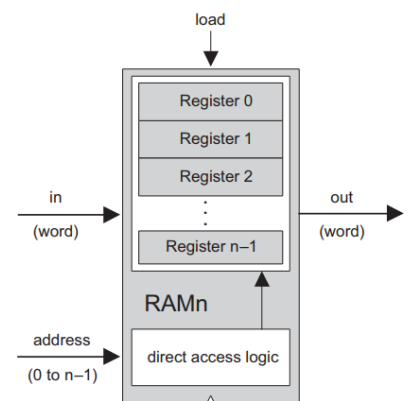
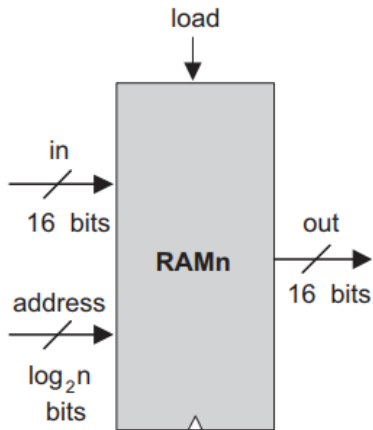


FIGURE 3.12: Schéma du fonctionnement de la Random Access Memory (RAM)



Chip name: RAMn // n and k are listed below

Inputs: in[16], address[k], load

Outputs: out[16]

Function: out(t)=RAM[address(t)](t)

If load(t-1) then

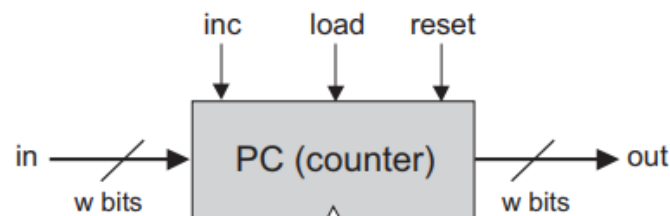
RAM[address(t-1)](t)=in(t-1)

Comment: "=" is a 16-bit operation.

The specific RAM chips needed for the Hack platform are:

Chip name	n	k
RAM8	8	3
RAM64	64	6
RAM512	512	9
RAM4K	4096	12
RAM16K	16384	14

FIGURE 3.13: Cahier des charges de la Random Access Memory (RAM)



Chip name: PC // 16-bit counter

Inputs: in[16], inc, load, reset

Outputs: out[16]

Function: If reset(t-1) then out(t)=0

else if load(t-1) then out(t)=in(t-1)

else if inc(t-1) then out(t)=out(t-1)+1

else out(t)=out(t-1)

Comment: "=" is 16-bit assignment.

"+" is 16-bit arithmetic addition.

FIGURE 3.14: Cahier des charges du compteur

plutôt que le montage réalisé précédemment.

On rappelle que pour réaliser un aiguillage entre deux parties d'un circuit en fonction d'une troisième variable (l'équivalent d'un "if" en code), on utilise un multiplexeur.

3.4.3 RAM

En suivant son cahier des charges (fig 3.13) réaliser, à partir d'une collection de registre et d'un circuit d'aiguillage gérant l'adresse, une mémoire RAM.

Pour minimiser le nombre de connexion à établir tout en gardant la logique de la puce, on réalisera une RAM contenant des données codées sous 4bits et avec un adressage des registres sous 2 bits.

3.4.4 Compteur

En suivant son cahier des charges (fig 3.14) réaliser un compteur.

4 *Langage machine*

A partir de porte NAND, nous avons réalisés dans le chapitre 2 une unité d'arithmétique logique (ALU) capable, entre autre de faire des additions et dans le chapitre 3 des mémoires composées d'ensemble de registre adressable (RAM).

Nous allons dans ce chapitre, toujours en suivant la démarche donnée par le livre Nisan and Schocken [2008] assembler ces deux éléments pour créer un processeur. Nous allons tout d'abord parler de l'*architecture* de l'ordinateur, c'est à dire comment agencer mémoire et ALU. Nous verrons ensuite comment coder les instructions que pourra effectuer le processeur ce qui nous conduira à la notion de *langage machine*.

Enfin, dans une troisième partie, nous verrons comment programmer cet ordinateur, c'est à dire écrire du code en langage machine afin, par exemple, de calculer une addition, une multiplication ou encore une factorielle.

4.1 *Architecture de l'ordinateur*

Flexibilité d'un ordinateur via le changement de programme.

Le code du programme est stockée et manipulée dans la mémoire de l'ordinateur exactement comme les données transitant au sein de l'ordinateur.

Harvard vs Von Neuman.

4.1.1 *Mémoire*

Mémoire pour les données High-level programs manipulate abstract artifacts like variables, arrays, and objects. When translated into machine language, these data abstractions become series of binary numbers, stored in the computer's data memory. Once an individual word has been selected from the data memory by specifying its address, it can be either read or written to. In the former case, we retrieve the word's value. In the latter case, we store a new value into the selected location, erasing the old value

Mémoire pour les instructions When translated into machine language, each high-level command becomes a series of binary words, representing machine language instructions. These instructions are stored in the computer's instruction memory. In each step of the computer's operation, the CPU fetches (i.e., reads) a word from the instruction memory, decodes it, executes the specified instruction, and figures out which instruction to execute next. Thus, changing the contents of the instruction memory has the effect of completely changing the computer's operation. The instructions that reside in the instruction memory are written in an agreed-upon formalism called machine language. In some computers, the specification of each operation and the codes representing its operands are represented in a singleword instruction. Other computers split this specification over several words

4.1.2 Central Processing Unit (CPU)

ALU Il s'agit de l'ALU construite au chapitre 2.

Registre When translated into machine language, each high-level command becomes a series of binary words, representing machine language instructions. These instructions are stored in the computer's instruction memory. In each step of the computer's operation, the CPU fetches (i.e., reads) a word from the instruction memory, decodes it, executes the specified instruction, and figures out which instruction to execute next. Thus, changing the contents of the instruction memory has the effect of completely changing the computer's operation. The instructions that reside in the instruction memory are written in an agreed-upon formalism called machine language. In some computers, the specification of each operation and the codes representing its operands are represented in a singleword instruction. Other computers split this specification over several words

Uniquement deux registres :

Le registre de données (data register)

Le registre d'adresses (addressing register)

Le registre d'instruction (Program counter register)

l'équivalent de notre mémoire à court-terme.

The CPU has to continuously access the memory in order to read data and write data. In every one of these operations, we must specify which individual memory word has to be accessed, namely, supply an address. In some cases this address appears as part of the current instruction, while in others it depends on the execution of a previous instruction. In the latter case, the address should be stored in a register whose contents can be later treated as a memory address—an addressing register.

These registers give the CPU short-term memory services. For example, when calculating the value of $(a - b)c$, we must first compute and re-

member the value of $(a - b)$. Although this result can be temporarily stored in some memory location, a better solution is to store it locally inside the CPU—in a data register.

Unité de contrôle voir section suivante

A computer instruction is represented as a binary code, typically 16, 32, or 64 bits wide. Before such an instruction can be executed, it must be decoded,

and the information embedded in it must be used to signal various hardware devices (ALU, registers, memory) how to execute the instruction. The instruction decoding is done by some control unit, which is also responsible for figuring out which instruction to fetch and execute next. The CPU operation can now be described as a repeated loop : fetch an instruction (word) from memory ; decode it ; execute it, fetch the next instruction, and so on. The instruction execution may involve one or more of the following micro tasks : have the ALU compute some value, manipulate internal registers, read a word from the memory, and write a word to the memory. In the process of executing these tasks, the CPU also figures out which instruction to fetch and execute next.

4.1.3 Notre architecture

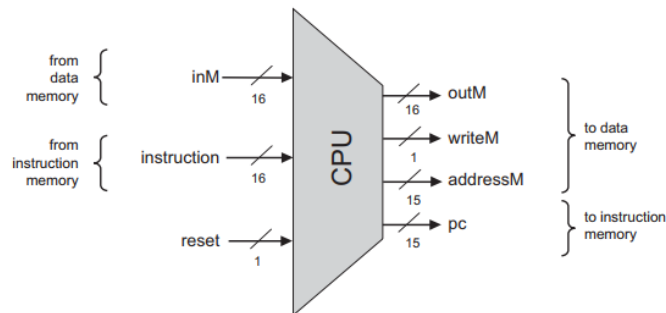
The Hack platform is a 16-bit von Neumann machine, consisting of a CPU, two separate memory modules serving as instruction memory and data memory

The Hack CPU consists of the ALU specified in chapter 2 and three registers called data register (D), address register (A), and program counter (PC). D and A are general-purpose 16-bit registers

While the D-register is used solely to store data values, the contents of the A-register can be interpreted in three different ways, depending on the instruction's context : as a data value, as a RAM address, or as a ROM address

CPU The CPU of the Hack platform is designed to execute 16-bit instructions according to the Hack machine language . It expects to be connected to two separate memory modules : an instruction memory, from which it fetches instructions for execution, and a data memory, from which it can read, and into which it can write, data values. Figure 5.2 gives the specification details

Instruction Memory The Hack instruction memory is implemented in a direct-access Read-Only Memory device, also called ROM. The Hack ROM consists of 32K addressable 16-bit registers, as shown in figure 5.3.



```

Chip Name: CPU // Central Processing Unit
Inputs: inM[16], // M value input (M = contents of RAM[A])
           instruction[16], // Instruction for execution
           reset // Signals whether to restart the current
                // program (reset=1) or continue executing
                // the current program (reset=0)
Outputs: outM[16], // M value output
            writeM, // Write to M?
            addressM[15], // Address of M in data memory
            pc[15] // Address of next instruction
Function: Executes the instruction according to the Hack machine language
              specification. The D and A in the language specification refer to
              CPU-resident registers, while M refers to the memory location
              addressed by A (inM holds the value of this location).

              If the instruction needs to write a value to M, the value is
              placed in outM, the address is placed in addressM, and the writeM
              bit is asserted. (When writeM=0, any value may appear in outM.)

              If reset=1, then the CPU jumps to address 0 (i.e., sets pc=0 in
              the next time unit) rather than to the address resulting from
              executing the current instruction.
  
```


Data Memory Hack's data memory chip has the interface of a typical RAM device, like that built in chapter 3 (see, e.g., figure 3.3). To read the contents of register n , we put n in the memory's address input and probe the memory's out output. This is a combinational operation, independent of the clock. To write a value v into register n , we put v in the in input, n in the address input, and assert the memory's load bit. This is a sequential operation, and so register n will commit to the new value v in the next clock cycle. In addition to serving as the computer's general-purpose data store, the data memory also interfaces between the CPU and the computer's input/output devices, using memory maps.

4.2 Langage machine

Arbitraire (mais efficace).

A machine language is an agreed-upon formalism, designed to code low-level programs as series of machine instructions.

A machine language program is a series of coded instructions. For example, a typical instruction in a 16-bit computer may be 1010001100011001. In order to figure out what this instruction means, we have to know the rules of the game, namely, the instruction set of the underlying hardware platform.

x86 1000 instructions différentes (en 2016)

Langage machine, Assembleur, langage haut niveau comme le C et le rôle du compilateur.

Although most people will never write programs directly in machine language, the study of low-level programming is a prerequisite to a complete understanding of computer architectures.

De plus, il est assez fascinant de se rendre compte à quel point les logiciels les plus sophistiqués ne sont, en fait, qu'une longue succession d'instruction extrêmement élémentaires, chacune déclenchant une opération très simple (comme une addition) réalisée à partir de'un agencement de portes logiques.

4.2.1 Notre ordinateur

The Hack machine language is based on two 16-bit command types :

- Une instruction de type "adresse" ou "A-instruction"
- Une instruction du type "calcul" ou "C-instruction"

Although the Hack syntax is more accessible than that of most machine languages, it may still look obscure to readers who are not familiar with low-level programming. In particular, note that every operation involving a memory location requires two Hack commands : One for selecting the address on which we want to operate, and one for specifying the desired operation.

4.2.2 *Instruction adresse*

Une instruction de type A est utilisée pour mettre une valeur codée sur 15-bit dans le registre d'adresse A du CPU. Elle se code ainsi :

$$0 \underbrace{vvv \ vvvv \ vvvv \ vvvv}_{v=0 \text{ ou } 1 : \text{ valeur à mettre dans A}} \tag{4.1}$$

La valeur 0vvv vvvv vvvv vvvv codé en binaire est alors mise dans le registre A. Notons que le bit de poids fort de l'instruction est un 0, ce qui permet de la distinguer d'une instruction de type C (voir section 4.2.3) dont le bit de poids fort est un 1.

Par exemple l'instruction :

$$0000 \ 000 \ 0000 \ 1110$$

qui commence par zéro, est donc une instruction de type A et va mettre dans le registre A la valeur 14 (i.e 1110 en binaire).

4.2.3 *Instruction calcul*

Cette instruction est plus complexe que l'instruction A vu précédemment, c'est elle qui déclenche un calcul au sein de l'ALU.

Une instruction de type C à la forme :

$$1 \ 1 \ 1 \ \underbrace{a \ c_1c_2c_3c_4 \ c_5c_6}_{\text{calcul}} \ \underbrace{d_1d_2 \ d_3}_{\text{dest}} \ \underbrace{j_1j_2j_3}_{\text{jump}} \tag{4.2}$$

En partant du bits de poids fort :

- Le premier bit d'identification
- Les deux bits suivants qui ne sont pas utilisés
- La partie calcul contenant 7 bits ($a \ c_1c_2c_3c_4 \ c_5c_6$) permettant de coder le calcul à effectuer
- La partie destination contenant 3 bits pour coder où stocker le résultat du calcul de l'ALU

Partie Identification L'instruction de type C commence par un 1, ce qui la distingue d'une instruction de type A

Partie calcul La partie calcul ($a \ c_1c_2c_3c_4 \ c_5c_6$) composés de 7 bits (soit 128 possibilités) permet d'effectuer des calculs à partir de trois sources de données :

- La valeur D contenue dans le registre D
- La valeur A contenue dans le registre A
- La valeur $M[A]$ contenue dans la mémoire pointée par l'adresse contenue dans A.

(when a=0) <i>comp mnemonic</i>	c1	c2	c3	c4	c5	c6	(when a=1) <i>comp mnemonic</i>
0	1	0	1	0	1	0	
1	1	1	1	1	1	1	
-1	1	1	1	0	1	0	
D	0	0	1	1	0	0	
A	1	1	0	0	0	0	M
!D	0	0	1	1	0	1	
!A	1	1	0	0	0	1	!M
-D	0	0	1	1	1	1	
-A	1	1	0	0	1	1	-M
D+1	0	1	1	1	1	1	
A+1	1	1	0	1	1	1	M+1
D-1	0	0	1	1	1	0	
A-1	1	1	0	0	1	0	M-1
D+A	0	0	0	0	1	0	D+M
D-A	0	1	0	0	1	1	D-M
A-D	0	0	0	1	1	1	M-D
D&A	0	0	0	0	0	0	D&M
D A	0	1	0	1	0	1	D M

Parmi les 128 valeurs possibles des 7 bits de la partie calcul de l'instruction de type C, seules¹ 28 sont implémentés. Elles sont répertoriées sur la figure 4.1.

Par exemple, pour que l'ALU calcul $D - 1$, c'est à dire prendre la valeur dans le registre D et lui soustraire 1, il faut donner comme instruction :

$$1110 \quad \mathbf{0011} \quad \mathbf{10}d_1d_2 \quad d_3j_1j_2j_3$$

la partie "calcul" de l'instruction étant en gras.

Pour calculer la valeur² $D|M$:

$$1111 \quad \mathbf{0101} \quad \mathbf{01}d_1d_2 \quad d_3j_1j_2j_3$$

Partie destination La partie destination ($d_1d_2 \quad d_3$) qui indique où le résultat du calcul de l'ALU doit être stocké. Les différentes possibilités sont données sur la figure 4.2

Par exemple, l'instruction de type C :

$$1110 \quad \mathbf{0011} \quad \mathbf{1011} \quad 0j_1j_2j_3$$

qui calcule, comme on la vu précédemment $D - 1$ stocke le résultat dans le registre A et le registre D mais pas l'élément de mémoire $M[A]$ (i.e) la case de mémoire de la RAM dont l'adresse est A.

Partie jump La partie "jump", dit à l'ordinateur ce qui doit faire pour la prochaine exécution. Doit-il continuer à l'instruction suivante dans

FIGURE 4.1: Tableau présentant le champ *calcul* d'une instruction de type C. D et A sont les variables contenus dans les registres du même nom et $M[A]$ est la valeur dans emplacement mémoire pointé par l'adresse contenue dans le registre A. "+" et "-" signifient addition et soustraction (avec prise en compte du complément à 2), "!", "|", "&" représente les opérateurs booléens 16bits not, or et and respectivement.

NB1 : Noter la similitude des bits $c_1c_2c_3c_4c_5c_6$ avec les bits de contrôle de l'ALU

NB2 : Noter que lorsque le bit a vaut 1 on travaille avec les valeurs contenues dans la mémoire et lorsque a vaut 0 on travaille avec les valeurs contenues dans le registre A. Des valeurs contenues dans le registre A sont pointées par l'adresse contenue dans le registre A

d1	d2	d3	Mnemonic	Destination (where to store the computed value)
0	0	0	null	The value is not stored anywhere
0	0	1	M	Memory[A] (memory register addressed by A)
0	1	0	D	D register
0	1	1	MD	Memory[A] and D register
1	0	0	A	A register
1	0	1	AM	A register and Memory[A]
1	1	0	AD	A register and D register
1	1	1	AMD	A register, Memory[A], and D register

FIGURE 4.2: Tableau présentant le champ *destination* d'une instruction de type C.

la ROM (ce qui est le comportement par défaut) ou, au contraire, embrancher vers une instruction différente ?

Faire un "jump" dépend de l'état de sortie de l'ALU. Les différents cas de figure sont répertoriés sur la figure 4.3.

Lorsque les conditions de "jump" sont remplies, l'ordinateur doit effectuer au prochain cycle d'horloge l'instruction dans la ROM qui se trouve à l'adresse présente dans le registre A.

j1 (out < 0)	j2 (out = 0)	j3 (out > 0)	Mnemonic	Effect
0	0	0	null	No jump
0	0	1	JGT	If out > 0 jump
0	1	0	JEQ	If out = 0 jump
0	1	1	JGE	If out ≥ 0 jump
1	0	0	JLT	If out < 0 jump
1	0	1	JNE	If out ≠ 0 jump
1	1	0	JLE	If out ≤ 0 jump
1	1	1	JMP	Jump

FIGURE 4.3: Tableau présentant le champ *jump* d'une instruction de type C. "Out" représente la valeur de sortie de l'ALU (telle que spécifiée par la partie calcul de l'instruction C), et "jump" veut dire "continuer l'exécution du programme avec instruction contenue à l'adresse pointé par la valeur du registre A".

Par exemple, l'instruction de type C :

1110 0011 1011 0010

qui calcule, comme on la vu précédemment $D - 1$ et qui stocke le résultat dans le registre A et le registre D, indique aussi ce qu'il faut faire pour l'instruction suivante. Si le résultat de l'ALU en l'occurrence $D - 1$ vaut zéro, alors la prochaine instruction que fera le CPU n'est pas la suivante dans la rom, mais celle dont le numéro est contenu dans le registre A.

4.2.4 Instruction vidéo

NB : Ce type d'instruction n'est pas présente dans la référence Nisan and Schocken [2008].

Une instruction de type V à la forme :

$$100x \quad xxx \quad xxxx \quad \underbrace{r_1r_2r_3r_4}_{\text{adresse registre}} \quad (4.3)$$

En partant du bits de poids fort :

- Les premier bit d'identification
- Tous les bits marqués "x" n'ont pas d'importance.
- Les quatre bits de poids faibles indique le numéro du registre de la mémoire vidéo.

Lors d'une instruction vidéo, le contenu du registre D est copié vers le registre mémoire dont l'adresse est donnée par les 4 derniers bits de l'instruction V.

4.3 Programmation

4.3.1 Addition de deux nombres

On cherche à additionner deux nombres a et b . Pour fixer les idées nous allons réaliser l'opération $13 + 29 = 42$

En C cela donnerait :

```
int a=13;
int b=29;
int c = a + b;
```

Le code machine va commencer par initialiser les valeurs a et b et prévoir l'espace mémoire pour le résultat c dans la RAM. Nous allons choisir arbitrairement que l'adresse 0000 de la RAM contiendra la valeur de a , la 0001 celle de b et 0002 celle de c .

adresse Ram	Variable
0000	a
0001	b
0002	c

TABLE 4.1: Assignation *arbitraire* de l'espace mémoire en RAM pour effectuer une addition

Cela est plus compliqué que simplement écrire "int a = 13;". Il faut pour cela 4 étapes³ :

1. Mettre la valeur 13 dans le registre A avec une instruction de type A :

$$13 \rightarrow A$$

2. Copier le contenu de A vers D :

$$A \rightarrow D$$

3. Mettre dans la A la valeur de l'espace mémoire de la variable a (en l'occurrence 0000 cf tableau 4.1)

$$0 \rightarrow A$$

3. On pourrait réduire un peu le nombre d'instruction en ne mettant pas en mémoire soit a soit b et en l'utilisant comme variable temporaire dans le registre a .

4. Copier la valeur de D vers l'espace mémoire prévu pour a , i.e. $M[A] : D \rightarrow M[A]$

Plus précisément :

1. Il faut d'abord utiliser une instruction de type A pour insérer la valeur que l'on souhaite mettre en RAM. Ces instructions prennent la forme (cf eq 4.1) :

$$0 \underbrace{vvv \quad vvvv \quad vvvv \quad vvvv}_{v=0 \text{ ou } 1 : \text{ valeur à mettre dans A}}$$

Ici par exemple pour la valeur de $a = 13$:

$$0000 \quad 0000 \quad 0000 \quad \underbrace{1101}_{13 \text{ en déc., } D \text{ en héra}}$$

2. Copier le registre A vers D pour garder temporairement cette valeur en mémoire.

En langage machine cela s'écrit :

$$A \rightarrow D$$

$$111a \quad c_1c_2c_3c_4 \quad c_5c_6d_1d_2 \quad d_3j_1j_2j_3$$

$$\underbrace{1110}_E \quad \underbrace{1100}_C \quad \underbrace{0001}_1 \quad \underbrace{0000}_0$$

où on demande à l'ALU de sortir la valeur du registre A ($c_1c_2c_3c_4c_5c_6 = 110000$) que l'on copie vers D ($d_1d_2d_3 = 010$)

3. Mettre dans A le numéro de l'adresse mémoire où on veut copier la valeur, ici pour la variable a par exemple, c'est l'emplacement 0, ce qui correspond à l'instruction de type A suivante

$$0000 \quad 0000 \quad 0000 \quad 0000$$

4. Copier le contenu de D (où on avait gardé la valeur à copier) dans l'emplacement mémoire à l'adresse A , c'est à dire $M[A]$

En langage machine cela s'écrit :

$$D \rightarrow M[A]$$

$$111a \quad c_1c_2c_3c_4 \quad c_5c_6d_1d_2 \quad d_3j_1j_2j_3$$

$$\underbrace{1110}_E \quad \underbrace{0011}_3 \quad \underbrace{0000}_0 \quad \underbrace{1000}_8$$

où on demande à l'ALU de sortir la valeur du registre D ($c_1c_2c_3c_4c_5c_6 = 001100$) que l'on copie vers M ($d_1d_2d_3 = 001$) avec $a = 0$.

Les lignes de code machine correspondantes sont reportées dans le tableau 4.2.

On passe ensuite à la partie addition. A nouveau, il faut plus d'étape que simplement écrire $c = a + b$;

1. Mettre la valeur de l'adresse de a dans le registre A ($0 \rightarrow A$).
2. Copier la mémoire pointé par le registre A dans le registre D ($M[A] \rightarrow D$)

TABLE 4.2: Langage machine pour l'addition - Phase initialisation. Il n'est pas nécessaire d'initialiser l'espace mémoire pour le résultat $c = a + b$, on a déjà décidé (arbitrairement) qu'il s'agit de l'adresse 0002

num ROM	code C	"Assembleur"	Code machine
0000	int a= 13;	13 → A	000D
0001		A → D	EC10
0002		0000(&a) → A	0000
0003		D → M[A]	E308
0004	int b= 29;	0005 → A	001D
0005		A → D	EC10
0006		0001(&b) → A	0001
0007		D → M[A]	E308

3. Mettre la valeur de l'adresse de b dans le registre A ($1 \rightarrow A$)
4. Faire l'addition entre la valeur dans D et la valeur dans la mémoire pointée par A i.e. $M[A]$. On met le résultat dans le registre D ($M[A] + D \rightarrow D$)
5. On met l'adresse de c dans le registre A ($2 \rightarrow A$)
6. On copie la valeur de D dans la mémoire pointée par A ($D \rightarrow M[A]$)

Plus précisément :

- On met la valeur de l'adresse de a (i.e 0000) dans le registre A avec l'instruction de type A suivante :

0000 0000 0000 0000

- On copie la mémoire pointé par le registre A ($M[A]$) dans le registre D avec l'instruction de type C suivante :

$M[A] \rightarrow D$

$111a$	$c_1c_2c_3c_4$	$c_5c_6d_1d_2$	$d_3j_1j_2j_3$
$\underbrace{1111}_F$	$\underbrace{1100}_C$	$\underbrace{0001}_1$	$\underbrace{0000}_0$

- On mets la valeur de l'adresse de b (1) dans le registre A :

0000 0000 0000 0001

- On effectue l'addition entre la valeur dans D et la valeur dans la mémoire pointée par A i.e. $M[A]$. On met le résultat dans le registre D . Ce qui donne en langage machine :

$M[A] + D \rightarrow D$

$111a$	$c_1c_2c_3c_4$	$c_5c_6d_1d_2$	$d_3j_1j_2j_3$
$\underbrace{1111}_F$	$\underbrace{0000}_0$	$\underbrace{1001}_9$	$\underbrace{0000}_0$

- On met l'adresse de c dans le registre A :

0000 0000 0000 0002

- On copie la valeur de D dans la mémoire pointée par A :

$D \rightarrow M[A]$

$111a$	$c_1c_2c_3c_4$	$c_5c_6d_1d_2$	$d_3j_1j_2j_3$
$\underbrace{1110}_E$	$\underbrace{0011}_3$	$\underbrace{0000}_0$	$\underbrace{1000}_8$

Les lignes de code machine correspondantes sont reportées dans le tableau 4.3.

TABLE 4.3: Langage machine pour l'addition - Phase initialisation. Il n'est pas nécessaire d'initialiser l'espace mémoire pour le résultat $c = a + b$, on a déjà décidé (arbitrairement) qu'il s'agit de l'adresse 0002

num ROM	code C	"Assembleur"	Code machine
0008	$c = a + b;$	0000(&a) $\rightarrow A$	0000
0009		$M[A] \rightarrow D$	FC10
0010		0001(&b) $\rightarrow A$	0001
0011		$M[A] + D \rightarrow D$	F090
0012		0002(&c) $\rightarrow A$	0002
0013		$D \rightarrow M[A]$	E308

4.3.2 Multiplication

L'ALU ne possède pas les portes logiques permettant de réaliser une opération de multiplication. Nous allons donc implémenter cette opération essentielle au niveau du software. Nous cherchons donc à multiplier deux nombres a et b et n'utilisant que des additions.

La procédure est relativement simple, en C cela donnerait :

```
int a, b;
int r = 0;
int j = b;

while (j != 0)
{
    r += a;
    j--;
}
```

Nous voyons qu'en plus de a et b nous avons besoin d'utiliser deux autres emplacement mémoire dans la RAM pour les variables r (pour résultat) et j qui joue le rôle de compteur pour la boucle while.

Pour fixer les idées nous allons effectuer la multiplication 3×15 ou, dit autrement, $a = 3$ et $b = 5$.

TABLE 4.4: Assignation arbitraire de l'espace mémoire en RAM

adresse Ram	Variable
0000	a
0001	b
0002	r
0003	j

Le code machine va donc commencer par initialiser ces valeurs dans la RAM. Nous allons choisir arbitrairement que l'adresse 0000 de la RAM contiendra la valeur de a , la 0001 celle de b , 0002 celle de r et 0003 celle de j . Nous allons :

1. Utiliser une instruction de type A pour insérer la valeur que mettre en RAM. Ces instructions prennent la forme :
2. Copier le registre A vers D pour garder temporairement cette valeur en mémoire.

En langage machine cela s'écrit :

$$\begin{array}{cccc}
 A \rightarrow D & & & \\
 111a & c_1c_2c_3c_4 & c_5c_6d_1d_2 & d_3j_1j_2j_3 \\
 \underbrace{1110}_E & \underbrace{1100}_C & \underbrace{0001}_1 & \underbrace{0000}_0
 \end{array}$$

où on demande à l'ALU de sortir la valeur du registre A ($c_1c_2c_3c_4c_5c_6 = 110000$) que l'on copie vers D ($d_1d_2d_3 = 010$)

3. Mettre dans A le numéro de l'adresse mémoire où on veut copier la valeur
4. Copier le contenu de D (où on avait gardé la valeur à copier) dans l'emplacement mémoire à l'adresse A , c'est à dire $M[A]$

En langage machine cela s'écrit :

$$\begin{array}{cccc}
 D \rightarrow M[A] & & & \\
 111a & c_1c_2c_3c_4 & c_5c_6d_1d_2 & d_3j_1j_2j_3 \\
 \underbrace{1110}_E & \underbrace{0011}_3 & \underbrace{0000}_0 & \underbrace{1000}_8
 \end{array}$$

où on demande à l'ALU de sortir la valeur du registre D ($c_1c_2c_3c_4c_5c_6 = 001100$) que l'on copie vers M ($d_1d_2d_3 = 001$) avec $a = 0$.

Les lignes de code machine correspondantes sont reportées dans le tableau 4.5.

num ROM	code C	"Assembleur"	Code machine
0000	int a= 3;	0003 → A	0003
0001		A → D	EC10
0002		0000(&a) → A	0000
0003		D → M[A]	E308
0004	int b= 5;	0005 → A	0005
0005		A → D	EC10
0006		0001(&b) → A	0001
0007		D → M[A]	E308
0008	int r= 0;	0000 → A	0000
0009		A → D	EC10
000A		0002(&r) → A	0002
000B		D → M[A]	E308
000C	int j= b;	0001(&b) → A	0001
000D		M[A] → D	FC10
000E		0003(&j) → A	0003
000F		D → M[A]	E308

TABLE 4.5: Langage machine pour la multiplication - Phase initialisation

Nous abordons ensuite la boucle while. Il faut tester si la valeur de j vaut 0. Si cela est le cas on *jump*⁴ vers l'adresse contenu dans le registre A .

Nous devons donc :

1. Mettre dans le registre D la valeur de J . Ce qui se décompose en :
 - (a) Mettre l'adresse de j (soit 0003) dans A .

4. Plus exactement le compteur pc qui indique la position de l'instruction à effectuer dans la ROM passe à la valeur contenue dans A

(b) Copier $M[A]$ dans D : En langage machine cela s'écrit :

$$\begin{array}{cccc}
 & M[A] \rightarrow D & & \\
 111a & c_1c_2c_3c_4 & c_5c_6d_1d_2 & d_3j_1j_2j_3 \\
 \underbrace{1111}_F & \underbrace{1100}_C & \underbrace{0001}_1 & \underbrace{0000}_0
 \end{array}$$

où on demande à l'ALU de sortir de la RAM à l'adresse A , i.e. $M[A]$ ($c_1c_2c_3c_4c_5c_6 = 110000$), avec $a = 1$, que l'on copie vers D ($d_1d_2d_3 = 010$).

2. Mettre dans A l'adresse où va s'effectuer le jump, c'est à dire ici la fin du programme, qui est prise arbitrairement à 0080 .
3. Tester si la valeur dans D (i.e. j) vaut 0 et si cela est le cas faire un jump.

En langage machine cela s'écrit :

$$\begin{array}{cccc}
 & D?0 & & \\
 111a & c_1c_2c_3c_4 & c_5c_6d_1d_2 & d_3j_1j_2j_3 \\
 \underbrace{1110}_E & \underbrace{0011}_3 & \underbrace{0000}_0 & \underbrace{0010}_2
 \end{array}$$

où on demande à l'ALU de sortir le registre D ($c_1c_2c_3c_4c_5c_6 = 001100$), avec $a = 0$, les bits de sorties $d_1d_2d_3$ n'ont ici pas d'importance, par contre les bits jumps doivent être mis à $j_1j_2j_3 = 010$, c'est à dire jump si la sortie de l'ALU est nulle.

TABLE 4.6: Langage machine pour la multiplication - Test boucle while

num ROM	code C	"Assembleur"	Code machine
0010	while (j!=0)	0003(&j) → A	0003
0011		$M[A] \rightarrow D$	FC10
0012		0080(&end) → A	0080
0013		D?o	E302

On entre ensuite dans la boucle while et devons effectuer l'instruction $r+ = a$. Nous pouvons pour ce faire :

1. Mettre la valeur de a dans le registre D avec les deux instructions 0000 puis FC10
2. Mettre la valeur de l'adresse de r dans le registre A avec l'instruction 0002
3. Effectuer l'opération $r = r + a$ en demandant à l'ALU $D + M[A]$ et en mettant le résultat dans $M[A]$

En langage machine cela s'écrit :

$$\begin{array}{cccc}
 & D + M[A] \rightarrow M[A] & & \\
 111a & c_1c_2c_3c_4 & c_5c_6d_1d_2 & d_3j_1j_2j_3 \\
 \underbrace{1111}_F & \underbrace{0000}_0 & \underbrace{1000}_8 & \underbrace{1000}_8
 \end{array}$$

où on demande à l'ALU de sortir le registre D ($c_1c_2c_3c_4c_5c_6 = 000010$), avec $a = 1$, que l'on copie vers $M[A]$ ($d_1d_2d_3 = 001$)

Il faut ensuite effectuer $j - -$, ce qui peut se faire en deux instructions :

1. Mettre l'adresse de j dans le registre A 0003
2. Demander à l'ALU de calculer $M[A] - 1$ et de mettre le résultat dans $M[A]$.

En langage machine cela s'écrit :

$$\begin{array}{cccc}
 & & M[A] - 1 \rightarrow M[A] & \\
 111a & c_1c_2c_3c_4 & c_5c_6d_1d_2 & d_3j_1j_2j_3 \\
 \underbrace{1111}_F & \underbrace{1100}_C & \underbrace{1000}_8 & \underbrace{1000}_8
 \end{array}$$

où on demande à l'ALU de sortir $M[A] - 1$ ($c_1c_2c_3c_4c_5c_6 = 110010$), avec $a = 1$, que l'on copie vers $M[A]$ ($d_1d_2d_3 = 001$)

Enfin, il faut s'occuper de la fin de la boucle while, le programme doit revenir au début de la boucle. Pour ce faire :

1. On met dans le registre A le numéro de l'instruction dans la ROM correspondant au début de la boucle while, en l'occurrence 0010 (voir le tableau 4.6)
2. On fait un jump inconditionnel. En langage machine cela s'écrit⁵ :

$$\begin{array}{cccc}
 & & JUMP & \\
 111a & c_1c_2c_3c_4 & c_5c_6d_1d_2 & d_3j_1j_2j_3 \\
 \underbrace{1110}_E & \underbrace{0000}_0 & \underbrace{0000}_0 & \underbrace{0111}_7
 \end{array}$$

où la sortie de l'ALU et sa destination n'ont pas d'importance, par contre il faut effectuer un jump inconditionnel avec $j_1j_2j_3 = 111$. Le programme va alors reboucler à l'instruction contenue dans le registre A en l'occurrence la 0010.

5. L'instruction doit être de type C. En effet, une instruction de type A 7, mettrait la valeur 7 dans le registre A sans prendre en compte le jump. La nature exacte de l'instruction C n'a pas d'importance, ici on met arbitrairement des 0.

num ROM	code C	"Assembleur"	Code machine
0014	r+=a	0000(&a) → A	0000
0015		M[A] → D	FC10
0016		0002(&r) → A	0002
0017		M[A] + D → M[A]	F088
0018	$j - -$	0003(&j) → A	0003
0019		M[A] - 1 → M[A]	FC88
001A	}	0010(&while) → A	0010
001B		JUMP	E007

TABLE 4.7: Langage machine pour la multiplication - Dans la boucle while

4.4 TD 4

4.4.1 Réalisation du circuit de condition de jump

Réaliser, à partir de portes logiques, un circuit avec en entrée :

- une entrée "control" sur 3 bits

- une entrée zero sur 1 bit
- une entrée neg sur 1 bit
- et
- une sortie "jump?" sur 1 bit

qui permet de dire (via la sortie "jump?"), à partir des sorties zero et neg de l'alu, des 3 bits de contrôle $j_1j_2j_3$ d'une instruction de type C et des règles de jump donnée sur la figure 4.3 p.44, si l'ordinateur doit effectuer un jump au prochain cycle d'horloge.

4.4.2 Le programme counter

6. A nouveau, nous pourrions utiliser celui que nous avons créé dans le chapitre 3

En utilisant⁶ un "counter" de logisim, créer un circuit qui a 4 entrées :

- Une sur 16 bits pour la valeur du registre A
- Une sur 1 bits qui recoit la valeur "jump" du circuit de condition de jump
- Une sur 1 bits appelée "reset"
- Une sur 1 bits pour l'horloge
- et une sortie :

- pc sur 16 bits qui indique le numéro de l'instruction suivante

construire un circuit à base de porte logique et du composant "counter" de logisim qui incremente la sortie "pc" de 1 à chaque tick d'horloge sauf si :

- si l'entrée jump est sur "1", la sortie "pc" doit alors valoir la valeur contenue dans le registre A
- L'entrée reset est sur 1, pc doit alors valoir 0. L'entrée reset a préséance sur l'entrée jump⁷.

7. Si Reset et jump sont sur 1 simultanément, la sortie pc doit valoir 0

4.4.3 CPU

A partir de :

- l'ALU réalisée au chapitre 2,
- Le circuit logique de gestion des jumps construit précédemment,
- le programme counter construit précédemment,
- Deux registres de logisim l'un appelé A l'autre D ,
- Deux entrées 16 bits, l'une appelée "inM" et l'autre "instruction",
- Trois sorties 16 bits appelées "outM", "adressM", et "pc"
- Une sortie 1 bit appelée "writeM".
- Deux entrées 1 bits, l'une pour l'horloge et l'autre pour un bouton reset.

construire le CPU tel qu'il a été défini dans la section 4.1.3 p.40.

La tâche principale est de décoder l'instruction. Plus précisément :

- L'instruction est-elle de type A ou non ?

- Si elle est de type A alors :
 - Il faut relier l'entrée instruction (qui contient alors la valeur à mettre dans le registre A) au registre A
- Si l'instruction est de type C il faut :
 - Décoder les bits de calcul ($a_1c_2c_3c_4c_5c_6$) pour les envoyer à l'ALU.
 - Via le bit a aiguiller soit le registre A soit l'entrée "inM" vers l'entrée de l'ALU
- Décoder les bits de destination ($d_1d_2d_3$) pour aiguiller où écrire la sortie de l'ALU.
- Décoder les bits de jump avec le circuit logique de gestion des jumps et informer du resultat le counter de programme.
- Brancher la sortie du programme counter vers la sortie "pc"

4.4.4 L'ordinateur

A partir de :

- Le CPU
- la mémoire ROM et RAM (sur 16 bits) intégrée⁸ à logisim
- Une horloge
- Une entrée reset sur 1 bit
- Un circuit Vram (fournit par l'enseignant)

Cabler ces éléments pour enfin obtenir l'ordinateur.

8. Les éléments de mémoire que nous avons construit au chapitre 3 fonctionnerait très bien mais finissent par ralentir l'exécution de logisim

4.4.5 Factorielle

Programmer la fonction factorielle. Vous pouvez implémenter la fonction par récursion ou alors par une série de multiplication.

```

int n=6;
int r = 1;

while (n !=0)
{
    r = multiplier(r,n);
}

int multiplier(int a, int b)
{
    int j = b;
    int rm = 0;
    while (j !=0)
    {
        rm += a;
        j--;
    }
}

```

```

return rm;
}

```

TABLE 4.8: Assignation *arbitraire* de l'espace mémoire en RAM

adresse Ram	Variable
0000	n
0001	r
0002	a
0003	b
0004	j
0005	rm

TABLE 4.9: Langage machine pour la factorielle - Phase initialisation

num ROM	code C	"Assembleur"	Code machine
0000	int n= 6;	0006 → A	0006
0001		A → D	EC10
0002		0000(&n) → A	0000
0003		D → M[A]	E308
0004	int r= 1;	0000 → A	0001
0005		A → D	EC10
0006		0001(&r) → A	0001
0007		D → M[A]	E308

TABLE 4.10: Langage machine pour la factorielle - Test boucle while

num ROM	code C	"Assembleur"	Code machine
0008	while (n != 0)	0000(&n) → A	0000
0009		M[A] → D	FC10
000A		F000(&end) → A	F000
000B		D ? 0	E302

TABLE 4.11: Langage machine pour la factorielle - Dans la boucle while

num ROM	code C	"Assembleur"	Code machine
000C	r=mult(r,n)	0000(&n) → A	0000
000D		M[A] → D	FC10
000E		0002(&a) → A	0002
000F		D → M[A]	E308
0010		0001(&r) → A	0001
0011		M[A] → D	FC10
0012		0003(&b) → A	0003
0013		D → M[A]	E308
0014		0100(&fct _{mult}) → A	0100
0015		JUMP	E007
0016	n --	0000(&n) → A	0000
0017		M[A] - 1 → M[A]	FC88
0018	}	0008(&while) → A	0008
0019		JUMP	E007

TABLE 4.12: Langage machine pour la factorielle - Fonction Multiplier

num ROM	code C	"Assembleur"	Code machine
0100	int j= b ;	0003(&b) → A	0003
0101		M[A] → D	FC10
0102		0004(&j) → A	0004
0103		D → M[A]	E308
0104	int rm= o ;	0000 → A	0000
0105		A → D	EC10
0106		0005(&rm) → A	0001
0107		D → M[A]	E308
0108	while (j !=o)	0004(&j) → A	0004
0109		M[A] → D	FC10
010A		01A4(&Fin - mult) → A	01A4
010B		D?o	E302
010C	rm+=a	0002(&a) → A	0002
010D		M[A] → D	FC10
010E		0005(&rm) → A	0005
010F		M[A] + D → M[A]	F008
01A0	j --	0004(&j) → A	0004
01A1		M[A] - 1 → M[A]	FC88
01A2	}	0108(&while) → A	0108
01A3		JUMP	E007
01A4	r = rm	0005(&rm) → A	0005
01A5		A → D	EC10
01A6		0001(&r) → A	0001
01A7		D → M[A]	E308
01A8		0016(&Aprs - mult) → A	0016
01A9		JUMP	E007

Deuxième partie

Micro-contrôleur

5 Quelques généralités sur les microcontrôleurs

Un micro-contrôleur est un agencement¹ complexe de micro composants électroniques (dont en grande majorité des transistors) dans le but d'effectuer des fonctions programmables.

Sa miniaturisation et son coût très faible lui permet de remplacer la logique câblée qui est maintenant devenue quasiment obsolète.

C'est un composant programmable, ce qui implique qu'il faut coder ; typiquement en assembleur ou en C.

Un micro-contrôleur possède quelques éléments clefs :

- Un CPU (central processing unit), ou le processeur capable d'effectuer sequentiellement des instructions machines (addition, accès mémoire, ...). Il est quelque part assez semblable à celui que nous avons créé dans la partie I du cours.
- De la mémoire :
 - Non volatile, pour stocker le programme à effectuer
 - Volatile, pour stocker les calculs intermédiaire.
- Des interfaces d'entrées-sorties comme par exemple :
 - Des entrées et sortie logiques capables d'émettre et de recevoir des tensions representant des niveaux logiques "0" ou "1".
 - Des entrées et sortie analogiques munies² de ADC (Analog to Digital Convertor) et de DAC (Digital to Analog Convertor)

Par rapport à un ordinateur un micro-contrôleur :

- a typiquement moins de puissance de calcul.
- n'a pas nécessairement d'interface³ type windows ou linux avec l'utilisateur ("operating system").
- a une consommation beaucoup plus faible (180 μ W pour arduino pro mini à comparer au TDP (Thermal Design Power) de plusieurs dizaines de watt des CPU x86 que l'on trouve dans les ordinateurs grand public.)
- sont beaucoup moins cher, on trouve des micro-contrôleurs performants pour moins de 1 euro.

1. Cette agencement est figé, contrairement au FPGA (Field-Programmable Gate Array) qui sont des composants dont l'architecture interne (les connections entre les transistors) peut être reprogrammé à la demande.

2. CAN et CNA en français.

3. On parle plus précisément de RTOS pour Real Time Operating system

5.1 Arduino

5.1.1 Présentation de la plateforme

Le succès de la plateforme arduino peut s'expliquer :

- Une des grandes forces de l'éco-système est de mettre à disposition un environnement de développement (IDE) qui simplifie *énormément* la programmation du microcontrôleur.
- Plaque de prototypage qui est d'ailleurs devenu un standard.
- Premier sur le marché

Pin layout.

Doc

Timer.

5.1.2 Getting started

Pas de cours de C.

Le but est de programmer le micro-contrôleur.

Paramétrage de la carte La carte arduino se branche via un port USB. Elle apparaît alors comme une connection série avec un nom⁴ commençant par COM. Il n'est pas possible de choisir quelle port COM sera choisi par la carte Arduino.

Il faut paramétrer deux points :

- Le type de carte :

Outils → Type de carte $\xrightarrow{\text{Dépend de la carte}}$ Arduino Guenuino/uno

Nous utiliserons ici principalement l'arduino *uno*.

- Le port COM de la carte :

Outils → Port $\xrightarrow{\text{Par exemple}}$ COM11

Structure du programme Voici la structure minimale d'un programme :

l'en tête qui sert typiquement à

- l'indication de l'utilisation de *bibliothèques*
- la déclaration des variables *globales* (i.e. communes à l'ensemble du programme)
- la déclaration de *constante* (exemple $pi = 3.14\dots$).

un setup Cette partie n'est lue qu'une seule fois au *démarrage* de l'arduino. Noter que l'arduino redémarre :

4. Il est fréquent que les ordinateurs ait déjà un port série qui sera appelé typiquement COM1. L'arduino aura donc un numéro de port série différent

- à chaque mise sous tension
- à chaque fois que l'on ouvre un terminal série dans l'IDE (voir ??)
- A chaque fois que l'on presse le bouton⁵ reset.

5. ou de façon équivalente en mettant un signal HIGH sur le pin Reset.

une loop (boucle) Cette partie du code est lue en boucle le plus rapidement possible. Lorsque le microcontrôleur arrive à la fin du code dans la boucle, il recommence au début. Cette boucle est effectuée jusqu'à ce que l'arduino ne soit plus alimenté électriquement ou qu'il soit reprogrammé.

Televersement

5.2 Interruption

Et ensuite? S'inspirer d'exemple.

Fichier → Exemple

Toujours consulter les fiches références des fonctions accessible via l'IDE

Aide → Référence

Chercher sur internet

Regarder la documentation du microprocesseur.

6 TP cours sur la plateforme Arduino

6.1 Sortie numérique

Les sorties numériques sont câblées sur les pins¹ 0 à 13 de l'arduino. Nous allons étudier dans cette section l'émission de signaux *logiques*, c'est à dire ne pouvant prendre que deux valeurs : HIGH ou LOW, ce qui se traduit pour l'arduino² en terme de signaux analogiques par une tension de 5 V ou 0 V.

La fonction à utiliser est ici *DigitalWrite*.

Quelques détails techniques : quand un pin logique de l'arduino est configurée en sortie (via la commande `pinMode(pin, OUTPUT)`), son impédance devient faible de l'ordre de 25Ω et la sortie peut délivrer³ jusqu'à 40 mA (ou accepter un courant négatif 40 mA)

1. Les pins 0 et 1 sont néanmoins plus ou moins réservés aux échanges RS232 (ports RX et TX)

2. A noter, beaucoup d'autre microcontrôleur utilise une tension de 3.3 V pour les signaux HIGH.

3. En anglais : "source" pour délivrer et "sink" pour absorber un courant.

6.1.1 Lire la doc (RTFM)

Regarder dans le menu référence (Aide → Référence) la (courte) documentation sur :

- la fonction *DigitalWrite*
- la fonction *pinMode*
- puis celle de la fonction *delay*.

6.1.2 Faire clignoter la LED placée sur la carte arduino (pin 13)

Le pin 13 est aussi connecté, via une résistance, à une LED, dite utilisateur, de la carte arduino.

Voici la démarche à suivre pour faire clignoter cette LED :

- Dans le setup, dire que le pin 13 est une sortie via "`pinMode(13, OUTPUT)`".
- Dans la loop (boucle), utiliser la fonction *DigitalWrite* pour mettre le pin 13 sur l'état HIGH.
- Faire attendre le micro-contrôleur avec la fonction *delay*.
- utiliser la fonction *DigitalWrite* pour mettre le pin 13 sur l'état LOW.

La loop va se répéter et la LED clignoter.

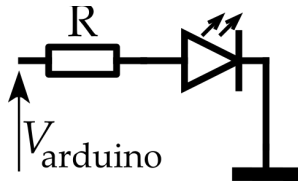


FIGURE 6.1: Circuit électrique pour alimenter une LED à partir d’une sortie logique de l’arduino. La résistance R sert à limiter le courant.

- 4. La tension seuil est la tension de gap entre la bande de valence et la bande de conduction c’est elle qui fixe la fréquence ν des photons émis par la relation $h\nu = eV_{seuil}$ où e est la charge de l’électron et h la constante de Planck. Cette relation traduit simplement la conservation de l’énergie.
- 5. Ce raisonnement n’est, en fait, pas tout à fait juste. En effet, les pin de sorties de l’arduino ont une impédance de l’ordre de 25Ω , dans le cas d’un court circuit (ou d’un branchement sur une diode passante comme cela est le cas ici), le courant maximum sera de l’ordre de $5V / 25\Omega = 200\text{ mA}$ et non 50 A
- 6. Au delà de 20 mA , les LED standards, c’est à dire celle de faible puissance, brillent plus fort, mais cela est au détriment de leur temps de vie.
- 7. La fonction ne renvoie pas de resultat (il n’y a pas de "return" à la fin de la fonction), "typeRetourFonction" est donc void (i.e. le mot clé en C pour une variable qui n’a pas de type)

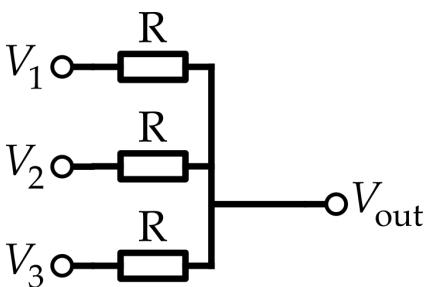


FIGURE 6.2: Circuit passif pour additionner 3 tensions. L’application du théorème de Millman donne $V_{out} = (V_1 + V_2 + V_3)/3$. On peut aussi facilement comprendre ce circuit en voyant que les resistance R transforme les tensions V_n en courant i_n et ces courants s’ajoute au niveau du noeud (cf loi des noeuds). Nous voyons qu’en plus d’être additionnées les tensions sont aussi atténuées. Il faudrait utiliser un montage actif (par exemple avec un AO) pour éviter cette atténuation. On prendra typiquement $R \approx 1\text{ k}\Omega$

6.1.3 Faire clignoter une LED externe

Réaliser le schéma de la figure 6.1. La résistance a pour but de limiter le courant fourni par l’arduino. En effet, une fois passé la tension seuil⁴ V_{seuil} de la LED, cette dernière se comporte quasiment comme un fil de resistance quasi-nulle. Mettons, pour fixer les idées⁵, que cette resistance est de 0.1Ω . Alimentée sous 5 V , un courant de 50 A est appelée dans le circuit! La LED et/ou la sortie de l’arduino vont cramer. Il faut donc limiter le courant.

Un arduino ne peut pas délivrer beaucoup plus que 40 mA , il faut donc absolument mettre la résistance entre la LED et la sortie de l’arduino qui délivre 5 V . Choisir la résistance R tel que le courant soit de l’ordre⁶ de 20 mA dans le circuit.

6.1.4 Emettre SOS en Morse

Reprendre la partie précédente et faire clignoter la LED pour émettre SOS en morse (... - - - ..., soit trois signaux courts, trois longs et trois courts).

6.1.5 Jouer une mélodie

1. Créer un signal "musical" à 440 Hz que vous enverrez sur une cellule piezzo qui va à son tour vibrer à 440 Hz . Il n’est ici pas nécessaire de mettre de résistance et vous pouvez directement brancher le piezzo entre la sortie digitale de l’arduino et la masse.
2. Dans un deuxième temps, créer une fonction "note" qui aura pour argument une fréquence et une durée. Programmer ensuite l’arduino pour qu’il joue une courte mélodie (au clair de la lune par exemple.)

On rappelle la syntaxe pour déclarer une fonction en C :

```
typeRetourFonction nomDeLaFonction(
typeArgument1 nomArg1,
typeArgument2 nomArg2)
{
// Ici le code de la fonction.
}
```

ce qui donnera⁷ ici :

```
void note(float freq, float length)
{
// Ici le code de la fonction.
}
```

3. Réaliser le montage présentée sur la figure 6.2 qui permet d’additionner passivement deux tensions. Utilisez le pour jouer des mélodies polyphoniques (plusieurs notes sont jouées en même temps).

6.1.6 Piloter un interrupteur avec l'arduino

Comme dit précédemment, un pin de l'arduino⁸ ne peut pas délivrer plus de 40 mA. Par conséquent, pour délivrer plus de courant à un circuit⁹, il faut passer par une alimentation extérieure contrôlée par l'arduino. Pour une commande en on/off et des faibles courants i de commutation ($i \lesssim 100$ mA) le plus simple est d'utiliser un transistor bipolaire¹⁰. Réaliser le montage de la figure 6.3 pour faire clignoter 5 LED mises en parallèles¹¹. Le courant total demandé au générateur sera donc de l'ordre de 5×100 mA et ne peut pas être fourni directement par l'Arduino.

On utilisera ici un transistor bipolaire NPN par exemple un 2N2222A dont le coefficient d'amplification β est de l'ordre de 100.

Tout se passe comme si un transistor bipolaire était un interrupteur contrôlé en courant. Or, les sorties de l'Arduino délivrent une tension, il faut donc utiliser une résistance R_B pour transformer cette tension en courant. Mais comment choisir la valeur de R_B ?

Nous voulons délivrer au niveau du collecteur un courant i_C de l'ordre de 100 mA, le transistor est alors saturé si le courant i_B est telle que $i_B > i_C/\beta$ soit environ 1 mA, avec une tension de $U_{\text{Arduino}} = 5$ V en sortie de l'arduino, il faut donc un résistance d'environ $R_B = U_{\text{Arduino}}/(i_C/\beta) = 5$ k Ω .

6.2 Communication par la voie série

Le port série¹² est utilisé pour communiquer entre la carte et l'arduino et un ordinateur (ou un autre appareil).

Le but est d'envoyer un signal numérique, qui est donc composé de 0 et de 1, d'un appareil vers l'autre. Par exemple, pour transmettre la lettre "a" en ASCII¹³, il faut transmettre les bits suivants : 0110 0001.

Dans le cas d'une liaison parallèle, il y a autant de câble que bits d'information à envoyer (huit câbles pour le cas du "a") et les bits sont envoyés simultanément.

Dans le cas d'une liaison série, comme cela est le cas ici¹⁴ pour l'arduino, il n'y a qu'un seul câble et on envoie successivement les bits du message sur ce câble.

La communication série de l'arduino possède donc deux câbles¹⁵ l'un pour *transmettre* (TX) et l'autre pour *recevoir* (RX). On peut accéder à ces deux câbles soit par la liaison USB¹⁶ de l'arduino (qui émule une liaison série au niveau de l'ordinateur) ou, mais c'est plus rarement le cas, directement à partir des pins 0 et 1 de l'arduino.

L'interface permet d'accéder aux ports séries sans rentrer spécifiquement dans ces détails techniques.

L'utilisation du port série de l'arduino nécessite l'appel de la commande¹⁷ `Serial.begin(baud)` où "baud"¹⁸ est de l'ordre de la vitesse

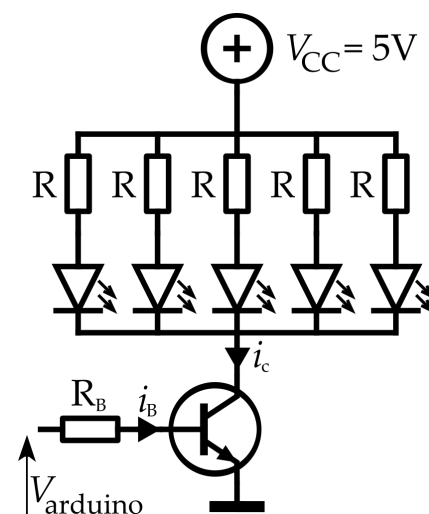


FIGURE 6.3: Circuit électrique pour que l'arduino commande un transistor bipolaire pour faire clignoter 5 LED mises en parallèles.

8. Et il ne faut pas dépasser 200 mA en sommant le courant délivré par l'ensemble des pins.
9. Cela est aussi le cas lorsque l'on veut commander un moteur ou un relai.
10. Pour des courants plus forts, on utilisera des transistor de type MOSFET. Ces derniers sont commandés en tension. Il est intéressant d'utiliser des MOSFET dit "niveaux logiques" qui commutent pour des tensions de 5 V, comme par exemple le IRF740 ou aussi le ULN2803A qui utilise plusieurs darlington.
11. Si les 5 LED sont mises en *série*, la tension nécessaire pour obtenir la mise en marche des 5 LED (i.e. plus grande que le gap du semi-conducteur multiplié par le nombre de LED en série) est de $\approx 5 \times 2 = 10$ V ce qui ne peut pas être directement obtenu avec une sortie logique d'un micro-contrôleur.
12. Aussi connu sous le nom d'UART (Universal Asynchronous Receiver Transmitter) ou même RS-232. Ce n'est pas exactement la même chose, mais nous n'avons pas ici besoin de ce niveau de détail.
13. American Standard Code for Information Interchange est une norme de codage des caractères. Elle utilise 7 bits et peut donc coder 128 caractères. On la retrouve très souvent dans les fichiers textes brutes (avec une extension .txt), elle est peu à peu remplacée par d'autres codages comme l'utf8 qui permet de gérer les caractères exotiques et les émojis.
14. La liaison série tend à devenir omniprésente en particulier à cause de l'usage massif des ports USB (Universal Serial Bus)

15. Il y a aussi 2 LED nommées RX et TX qui informe du trafic sur les câbles correspondant
16. Cette liaison fait appel à un deuxième micro-contrôleur auxiliaire : un ATmega16U2
17. On peut voir au passage que "Serial" est un fait un objet (au sens de la Programmation Orientée Objet comme cela est par exemple le cas en C++), l'opérateur "." permet d'accéder à la méthode begin de l'instance de l'objet Serial.
18. Le baud est une très vieille unité héritée des communications télégraphiques (!). Le baud est l'inverse de la durée en secondes du plus court élément du signal. Il ne faut pas le confondre avec le nombre de bit par seconde transmis le long du câble série.
En effet, le signal envoyé sur un port série ne peut pas être constitué uniquement des données à transmettre. Il faut par exemple indiqué quel est le début du signal (start bit) et la fin du signal (stop bit). On peut ensuite aussi ajoutée des bits de contrôle (parity bit), pour vérifier si le signal a été transmis sans être corrompu.
19. Selon la documentation du microcontrôleur, on peut monter jusqu'à 1MBd.
20. L'arduino embarque très peu de mémoire vive (2ko SRAM) et ne peut pas stocker une grande quantité d'information. Lorsque l'on fait de nombreuses mesures (plus d'une centaine), il faut transférer le contenu de la mémoire vers un ordinateur ou un autre moyen de stockage comme par exemple une carte SD.
21. Il existe d'autre méthode plus bas niveau comme par exemple "write".
22. La méthode "println" affiche la variable et ajoute un caractère de retour à la ligne "\n". On peut éviter cela en utilisant la méthode "print"

de succession de "0" et de "1" le long du câble série. Les valeurs typiques¹⁹ pour le baudrate sont 9600, 57600 ou 115200.

La taille du buffer (mémoire tampon) est, pour l'arduino uno, de 64 octets.

La communication de l'arduino vers l'ordinateur permet typiquement :

- Transmettre²⁰ le résultat d'une mesure.
- Debugger le programme, pour connaître par exemple la valeur d'une variable.

Notons enfin qu'utiliser le port série est lent et ralenti considérablement le programme.

6.2.1 Lire la doc (RTFM)

Regarder dans le menu référence (Aide → Référence) la (courte) documentation²¹ sur :

- la commande *Serial* puis *begin*
- la commande *Serial* puis *println*

6.2.2 Communication de l'arduino vers l'ordinateur

Une fois le port série initialisé avec, dans le setup, la commande *Serial.begin(57600)*, on affiche sur le port série une variable via la commande²² : *Serial.println(variable)*. Le type de la variable peut être une chaîne de caractère, un "int" ou encore même un "float". La fonction "println" les convertit par défaut pour les affichages les plus courants.

Écrire un programme qui additionne les 100 premiers entiers positifs ($\sum_{i=0}^{100} i$) et qui affiche les résultats intermédiaires.

Vérifier ensuite (toujours avec le port série), que vous obtenez bien le résultat attendu à savoir $N(N + 1)/2$, où N est l'entier où s'arrête la somme à savoir ici 100.

6.2.3 Communication de l'ordinateur vers l'arduino

Cela est un tout petit peu plus délicat que de dialoguer de l'arduino vers l'ordinateur. Il y a deux stratégies :

- Regarder en permanence si de nouvelles données ont été envoyées dans le port série. On parle de polling. Voici un exemple de code :

```
int incomingByte = 0; // pour les données du port serie

void setup() {
  Serial.begin(9600); // opens serial port, sets data rate to 9600 bps
}
```

```
void loop() {

  // Tester s'il y a des données en attente dans la mémoire tampon du serial
  if (Serial.available() > 0) {
    // Lire le contenu de la mémoire tampon ->un<- caractère à la fois.
    incomingByte = Serial.read();

    // Afficher la valeur qui vient d'être lue sous forme de caractère (char)
    Serial.print((char)incomingByte);
  }
}
```

— On peut utiliser une interruption (voir aussi la section ??), lorsqu'un nouveau élément arrive dans la mémoire tampon du port série, cela déclenche une interruption qui arrête la boucle principale.

Regarder dans le menu référence (Aide → Référence) la (courte) documentation sur :

- les commandes Serial read, Serial readString, Serial aviable.
- la commande SerialEvent

En utilisant une méthode de type polling, allumer une LED lorsque le mot "on" est envoyé sur le port serie et eteindre cette même LED lorsque le mot "off" est envoyé sur le port série.

6.3 Sorties PWM

Plusieurs sorties numériques²³ de l'arduino sont capables d'émettre un signal de type PWM (Pulse Width Modulation). Un tel signal est une succession de d'impulsion (comprise pour l'arduino uno entre 0 et 5 V) dont le rapport cyclique peut être changer (voir figure 6.4).

La fréquence du signal est par défaut²⁴ de 490 Hz.

La commande²⁵ a utilisé est analogWrite().

6.3.1 Lire la doc (RTFM)

Regarder dans le menu référence (Aide → Référence) la (courte) documentation sur :

- la commande analogWrite

6.3.2 Etude du signal à l'oscilloscope

Mesurer le signal PWM émit par l'arduino à l'oscilloscope. Changer la valeur envoyée à la fonction "analogWrite" pour voir l'effet sur le signal emis par l'arduino. Comparer à la figure 6.4.

- 23. Elle sont indiquées par la présence d'un tilde " ~ " devant le numéro du pin
- 24. Cette dernière peut être changée via des librairies tierces ou en modifiant les registres.

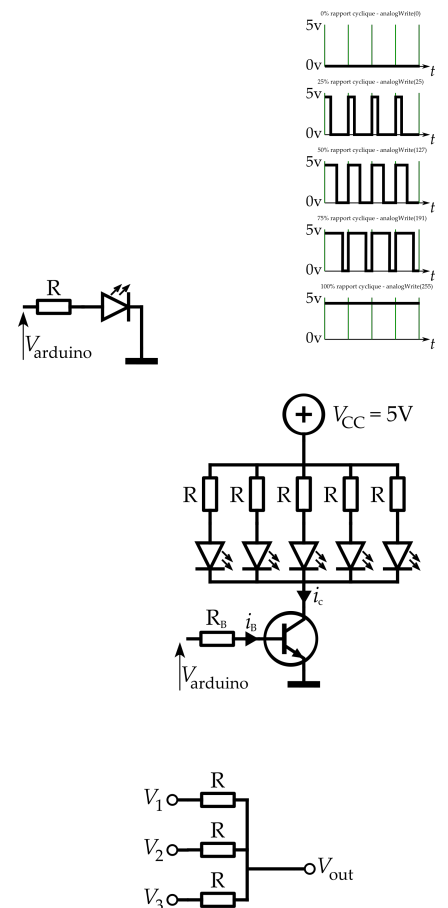


FIGURE 6.4: Signal PWM.

- 25. Noter que malgré le nom "analogWrite", le signal reste numérique (0 ou 1). Il est possible de transformer un signal PWM en vrai signal analogique en le filtrant. Cela fait l'objet de la section 6.6.1.

26. La fréquence du signal PWM (490 Hz est bien plus grande que la fréquence de réponse de l'oeil (≈ 25 Hz). Ainsi, via la persistance rétinienne, nous ne voyons pas le clignotement de la LED mais nous percevons un changement d'intensité moyenne. L'oeil effectue un filtre passe-bas qui donne la valeur moyenne du PWM (nous utilisons cette technique pour créer un DAC à la section 6.6.1)

27. L'impédance d'entrée est de l'ordre de $100\text{ M}\Omega$

28. Le temps de mesure pour un arduino UNO est de l'ordre de $5\ \mu\text{s}$ soit une fréquence max de 200 kHz

6.3.3 Modulation de l'intensité d'une LED

1. Alimenter une LED (avec une résistance pour limiter le courant) avec un signal PWM délivré par l'arduino. Vérifier qu'en modifiant le rapport cyclique du signal PWM vous modifier l'intensité de la LED perçue par votre oeil²⁶.
2. Modifier la valeur du rapport cyclique au sein de la loop pour créer un effet de "breathing led" (la LED augmente et diminue sa luminosité avec une période de l'ordre de la seconde.).
3. Avec 2 LED oranges et 1 LED rouge et en utilisant la fonction fonction random (voir dans le référence) de l'arduino. Recréer l'effet d'un éclairage par une bougie dont la flamme bouge légèrement.

6.4 Entrée numérique - digital read

Une fois configuré les pins en entrée²⁷ (input), via la commande `pinMode(pin, INPUT)`, on peut connaître²⁸ l'état logique du pin en question avec la commande `digitalRead(pin)` qui renvoie HIGH ou LOW.

6.4.1 Lire la doc (RTFM)

Regarder dans le menu référence (Aide → Référence) la (courte) documentation sur :

- la commande `digitalRead`
- la commande `pinMode` (à nouveau).

Anyone who writes software for microcontrollers will have to configure and manage general purpose input/output (GPIO) pins. On their surface, GPIO configuration seems simple : pins are input or output, and they can be high or low.

However, inevitably you will come across a fancy processor with a plethora of configuration options, or an electrical engineer will request pin settings which you don't understand ("make this line Hi-Z").

This guide aims to help you understand different pin configuration options that are provided on modern microcontrollers.

Most modern GPIO lines are implemented as a tri-state buffer. This means that the GPIO line can effectively assume three values :

Logical 0 (connection to ground) Logical 1 (connection to VCC) High-impedance (also called "floating", "Hi-Z", "tri-stated")

A signal is said to be "floating" when its state is indeterminate, meaning that it is neither connected to VCC or to ground. The signal's voltage will "float" to match the residual voltage.

The term "floating" is often used interchangeably to describe a pin which is in the high-impedance state.

Pull-ups are resistors that connect a signal to VCC. Pull-ups are used to set a default state when the signal is floating.

Recall that when an input pin is in high-impedance mode and not driven by external sources, it is floating at a residual voltage level. Pull-up resistors prevent the pin from floating by forcing the signal to VCC when it is not being actively driven. When another source drives the signal low (connects to ground), the pull-up is overridden and the input pin will read a '0'. (Divisuer de tension avec un fil sans resistance à la masse donc $R = 0$).

Many microcontrollers supply internal pull-up configuration options. Sometimes, a specific pull-up resistor value is required which necessitates using an external pull-up instead of a chip's internal pull-up.

Pull-downs are resistors that connect an signal to ground. Pull-downs are used to set a default state when the signal is floating. When another source drives the signal high (connects to VCC), the pull-down is overridden and the input pin will read a '1'.

GPIO Input Modes When a GPIO is configured as an input, it can be used to read the state of the electrical signal. Configuring a GPIO as an input puts the pin into a high-impedance state.

In general, there GPIO inputs are primarily configured in one of three ways :

High-impedance (default – floats if not driven) Pull-up (internal resistor connected to VCC) Pull-down (internal resistor connected to Ground) Most GPIO input pins also feature internal hysteresis, which prevents spurious state changes on the pins. Usually hysteresis is a built-in feature, rather than a configurable setting.

Since there is no internal connection to VCC, the open drain output does not put out a voltage like an Arduino GPIO pin. You have to provide your own external pullup. While this may seem inconvenient, the open drain configuration allows for a lot of flexibility.

6.4.2 Les résistances de pull-up

1. Brancher un bouton entre le 5V de l'arduino et une des pins logique de l'arduino. Utiliser la fonction `digitalRead` pour detecter la pression du bouton ce qui déclenchera l'allumage d'une LED. Voir que le circuit a un comportement erratique.

Plus précisément, lorsque le bouton est pressé on obtient un état HIGH (5V), par contre lorsque l'interrupteur est ouvert, l'état logique est alors fluctuant. En effet, les entrées de l'arduino sont très sensible et l'interrupteur ouvert peut capter les signaux electromagnétique présent dans la pièce et avoir un tension qui fluctue fortement passant parfois au niveau logique²⁹ HIGH.

On peut résoudre ce problème de deux façons :

- Par une résistance pull-down (resistance de rappel en français)
On insère une résistance entre la masse et l'entrée de l'arduino

29. Selon la documentation une tension supérieur à 3V est considérée comme HIGH par le micro-contrôleur de l'arduino UNO.

30. La résistance d'entrée de l'Arduino étant de $100\text{M}\Omega$, le courant d'entrée sous 5V est négligeable (50nA)

(voir figure 6.5), cette résistance a pour rôle d'assurer que lorsque l'interrupteur est ouvert le pin d'entrée de l'Arduino est la masse. Lorsque le bouton est pressée, d'une part la résistance de Pull-down se trouve en parallèle de la résistance d'entrée de l'arduino, d'autre part un courant a priori non négligeable³⁰ circule dans la résistance $R_{\text{pull down}}$ de telle sorte qu'avec une tension de 5V à ses bornes, il faut au moins prendre une résistance de $5\text{k}\Omega$ pour limiter le courant à environ 1mA

— Par une résistance pull-up (résistance de tirage en français).

On relie le pin d'entrée au 5V via une résistance (dite pull-up) et l'interrupteur à la masse. Ce montage (voir figure 6.6) est peut-être moins intuitif que le précédent :

- Lorsque l'interrupteur est *ouvert*, le 5V est connecté au pin d'entrée de l'arduino via la résistance de pull-up qui *limite le courant*, l'état est donc *HIGH*.
- Lorsque l'interrupteur est *fermé*, le pin d'entrée de l'arduino est alors relié à la masse (malgré la présence de liaison du 5V avec la résistance de pull-up), l'état est donc *LOW*.

L'utilisation d'une résistance de pull-up inverse la logique, lorsque l'interrupteur est fermé est *LOW* et inversement. Il faut en tenir compte dans le code.

NB : la résistance de pull-up doit être au moins d'environ $5\text{k}\Omega$.

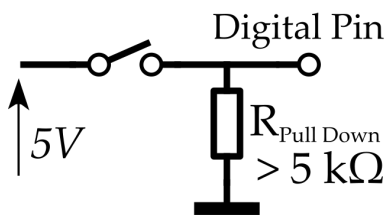


FIGURE 6.5: Insertion d'une résistance pull-down pour stabiliser l'état LOW des entrées digitales

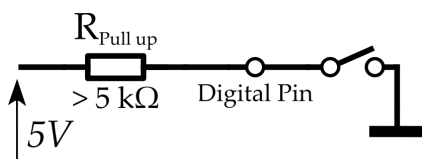


FIGURE 6.6: Insertion d'une résistance pull-up pour stabiliser l'état LOW des entrées digitales

31. En interne, demander le mode `INPUT_PULLUP`, connecte le pin d'entrée, via un multiplexeur, vers le 5V via la résistance de pull-up

2. Insérer une résistance de pull-down (cf fig 6.5) pour stabiliser l'état LOW de l'entrée digitale de l'arduino.
3. Oter la résistance de pull-down et mettre une résistance de pull-up (cf fig 6.6) pour stabiliser l'état LOW de l'entrée digitale de l'arduino. Vérifier que l'état logique du bouton est inverser et faire les modifications nécessaires dans le code pour revenir au fonctionnement initial.
4. Oter les résistances de pull-up ou pull-down externe et utiliser la fonction `pinMode` (relire la documentation si nécessaire) pour enclencher la résistance de pull-up interne de l'entrée numérique de l'arduino. Le mot clef est alors `INPUT_PULLUP`. La connexion, tout comme dans la figure 6.6 doit être faite entre le pin d'entrée et la *masse*³¹. Corriger, si nécessaire, dans le code l'inversion de logique associée à l'utilisation d'une résistance de pull-up.
5. A chaque pression d'un bouton on incrémente un compteur, ce nombre est affiché en binaire grace à 4 diodes. Un deuxième bouton permet de remettre le compteur à zero.
6. On branche 7 Led pour créer un dé lumineux. Lorsque l'on appuie sur un bouton un nombre est choisi au hasard entre 1 et 6, on l'affiche sur les LED (la septième servant au point du milieu pour le 5 par exemple), puis on fait clignoter le résultat.

6.5 ADC Analog to Digital Conversion

L'arduino UNO est muni de 6 entrées, numérotée de A0 à A5, capable d'effectuer une conversion analogique vers numérique (ADC en anglais pour Analog to Digital Conversion).

Le tension électrique zentre les bornes Ax et la masse de l'arduino est numérisé sur la plage 0-5V sur 10bits (avec donc une valeur allant de 0 à $10^{10} - 1 = 1023$). Cela donne donc une résolution de $5/1024 \approx 4.9$ mV. L'impédance d'entrée est donnée pour 100 M Ω . La fréquence d'échantillonnage maximum est de l'ordre³² de 10 kHz).

6.5.1 Lire la doc (RTFM)

Regarder dans le menu référence (Aide → Référence) la (courte) documentation sur :

- la commande *analogRead*

6.5.2 Utilisation d'un potentiomètre

- Brancher un potentiomètre en diviseur de tension entre 0 et 5V vers l'une des entrées analogiques de l'arduino. Avec la fonction *analogRead* obtenir la valeur
- Allumer une rangée de LED, 3 vertes, 1 orange et 1 rouge en fonction de la valeur detectée.

Intensité musique - peak metre et RMS metre. Photodiode avec grosse résistance. Testeur de pile.

6.6 DAC Digital to Analog Conversion

6.6.1 PWM

En PWM ac un RC

Avec un réseau de résistance

6.7 Interuption

6.8 Timer

6.9 Utiliser des commandes plus bas niveau

6.10 Mini Projet

32. Il faut faire attention lorsque l'on bascule rapidement d'une voie à une autre car il peut y avoir du cross-talk (diaphonie en français) entre les différentes voies. En effet, les 6 pins partage le même ADC, un multiplexeur aiguille le pin d'entrée demandé par l'utilisateur vers l'ADC.

Bibliographie

Noam Nisan and Shimon Schocken. *The Elements of Computing Systems : Building a Modern Computer from First Principles (History of Computing S.)*. The MIT Press, 2008. ISBN 9780262640688.

A Quelques éléments de corrections des TD

TD 1

Exercice 1

1.

$$S = (\bar{A} + B) \cdot (A + B) = A \cdot \bar{A} + A \cdot B + B \cdot \bar{A} + B \cdot B = 0 + A \cdot B + B \cdot \bar{A} + B = B \cdot (A + \bar{A} + 1) = B$$

NB : on peut directement reconnaître un des théorème de Boole.

2.

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

Exercice 2

1.

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

2.

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

3.

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

4.

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

5.

$$S = \overline{(A + \bar{B}) + (B + \bar{A})} = \bar{A} \cdot B \cdot \bar{B} \cdot A = 0$$

6.

$$S = \overline{\overline{\bar{A} \cdot B \cdot CD}} = (\bar{A} + \bar{B}) \cdot C + \bar{D}$$

Exercice 3

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

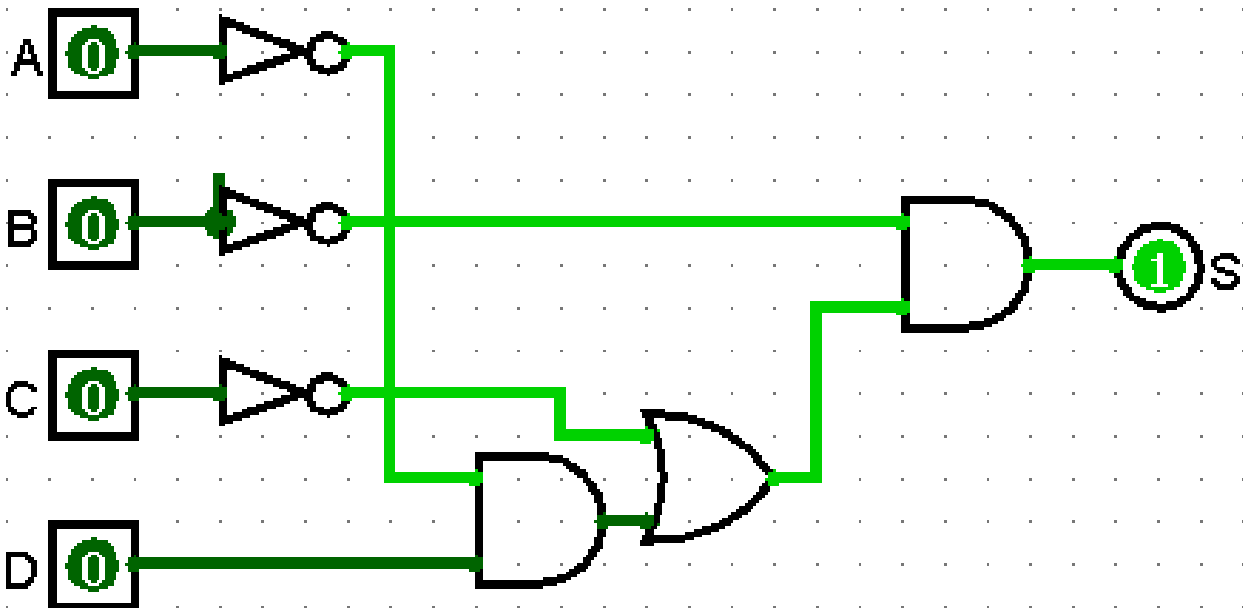


FIGURE A.1: Exercice 3. Schéma simplifié.

Exercice 4 Écriture canonique :

$$S = \bar{A} \cdot B \cdot C + A \cdot \bar{B} \cdot C + A \cdot B \cdot \bar{C} + A \cdot B \cdot C = (\bar{A} \cdot B + A \cdot \bar{B}) \cdot C + A \cdot B = (A \oplus B) \cdot C + A \cdot B$$

XOR 3 entrées

De l'universalité de la porte NAND Voir figure A.3.

1. Une porte NON :

$$\bar{x} \cdot \bar{x} = \bar{x} + \bar{x} = \bar{x}$$

2. Une porte ET On utilise le fait qu'une le NOT d'une porte NAND est une porte NAND.

3. Une porte OU :

$$\overline{\bar{x} \cdot \bar{y}} = x + y$$

4. Une porte XOR : On veut produit $x \cdot \bar{y} + \bar{x} \cdot y$. On peut utiliser les portes que l'on a mis au point avant.

On peut aussi voir que :

$$\overline{\overline{a \cdot (a \cdot b)} \cdot \overline{b \cdot (a \cdot b)}} = a \cdot (\bar{a} \cdot \bar{b}) + b \cdot (\bar{a} \cdot \bar{b}) = a \cdot (\bar{a} + \bar{b}) + b \cdot (\bar{a} + \bar{b}) = a \cdot \bar{b} + \bar{a} \cdot b$$

5. Un multiplexeur/demultiplexeur.

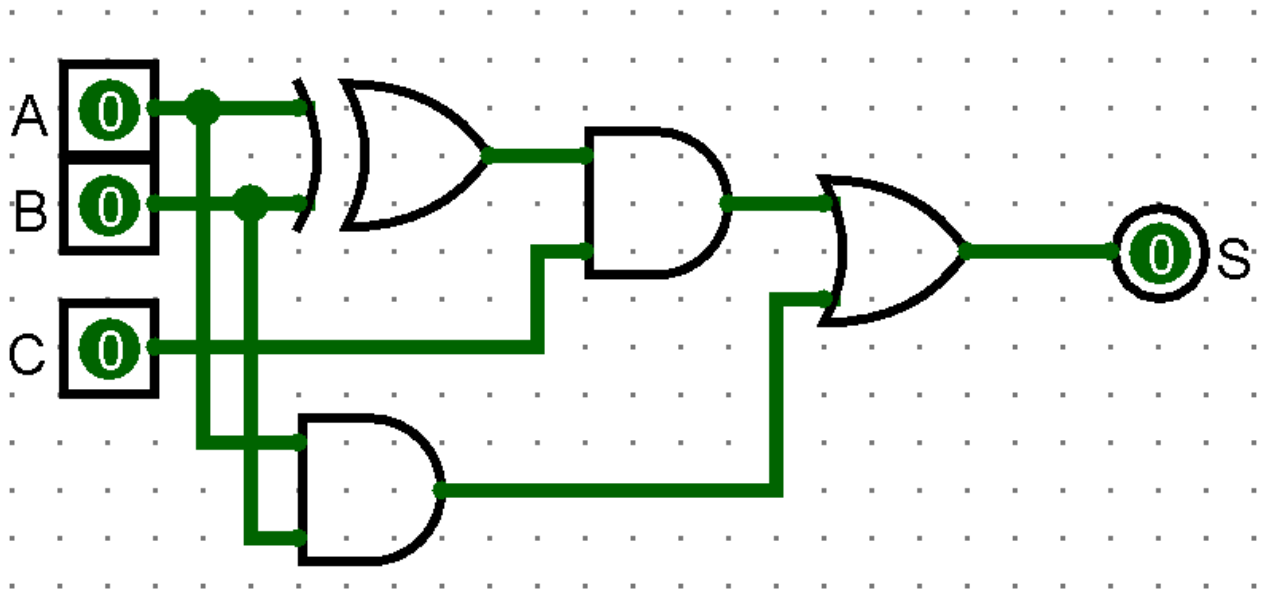


FIGURE A.2: Exercice 4. Schéma.

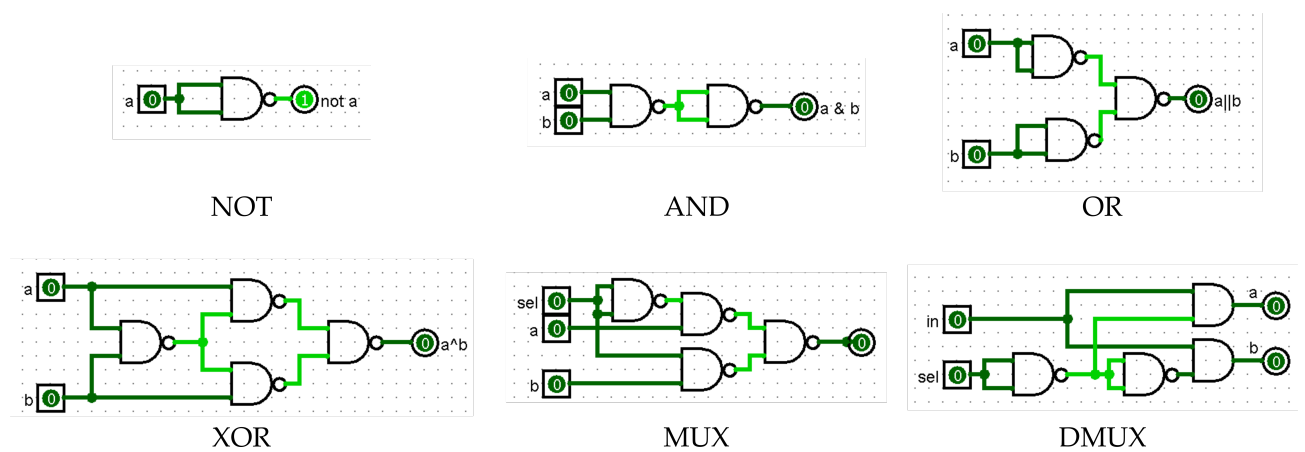


FIGURE A.3: Liste des principales portes logique réalisés à partir de la porte NAND.

