# TP 03 : Construire un vrai ordinateur ?

## 1 Introduction

Dans ce TP, on va essayer de construire un petit ordinateur à partir d'une porte `NAND` seulement. Pour ce faire on va construire des unités logiques de plus en plus compliquées jusqu'à avoir un ordinateur capable de faire tourner des programmes. Pour ce TP, on va utiliser un Hardware Simulator. Son rôle sera de vérifier que vos unités logiques (*chips*) sont bien implémentées. De plus, il vous aidera aussi pour vos déboguer vos chips.

Le langage dans lequel vous allez écrire vos chips est un DSL (Domain Specific Language). C'est à dire que c'est un langage spécifiquement dédié à l'écriture de vos chips. Dans la prochaine section, je vais brièvement décrire ce langage afin que vous puissiez écrire rapidement vos premiers chips. N'hésitez-pas à me poser des questions sur le langage, notamment lorsque vous écrirez des chips utilisant des bus de données.

Chaque chip vient en trois fichiers :
— `XXX.hdl` : c'est dans ce fichier que vous écrirez le code de vos chips
— `XXX.tst` : ce fichier est déjà écrit pour vous et contient des tests.
— `XXX.cmp` : ce fichier est aussi déjà écrit et contient la sortie des tests.

Les deux derniers fichiers sont utilisés par le Hardware Simulator pour vérifier la correction de vos chips. A priori, vous éditerez seulement les fichiers dont l'extension est `.hdl`.

### Utilisation du Hardware Simulator

Une fois le Hardware Simulator lancé, pour vérifier votre chip vous devez faire deux étapes :
— Cliquer sur le bouton **load chip** pour charger le fichier `.hdl`
— Cliquer sur le bouton **load script** pour charger le fichier `.tst`
— Enfin, cliquer sur le bouton **run** (double flèche bleue) pour lancer les tests

Pour gagner du temps, je vous invite à sélectionner l'option **no animation** dans le menu **animation**.

### Plan

Ce TP est découpé en $4$ sections :

1. Dans la section 3, vous implémenterez les chips de base : `And`, `Or`, ...
2. Dans la section 4, on s'intéressera à la conception d'un ALU (Arithmetic Logic Unit)
3. Dans la section 5, on s'intéressera à des chips non-séquentiels afin de créer la mémoire (RAM) de notre ordinateur
4. Dans la section 7, on réunira les différents CHIPS écrits dans les sections précédentes pour fabriquer l'ordinateur.

L'archive contenant le projet est trouvable ici : `https://www.lsv.fr/~fthire/teaching/2018-2019/archisys/TP/3/bootstrap.tar.gz`.

## 2 Hardware Language

Le *Hardware Language* est un langage qui permet de programmer les chips. Tous les chips que vous allez écrire respectent le format suivant (exemple de la porte `AND`) :

```
/**
 * And gate:
 * out = 1 if (a == 1 and b == 1)
 *       0 otherwise
 */

CHIP And {
    IN a, b;
    OUT out;

    PARTS:
    //TODO
}
```

Chaque chip commence par un commentaire résumant le comportement attendu du chip. Ensuite, le code d'un chip se découpe en trois parties :

— *IN* : cette ligne donne des noms aux pins d'entrée. Ici, la porte `AND` à deux pins en entrée nommés `a` et `b`

— *OUT* : cette ligne donne des noms aux pins de sortie. Ici, la porte `AND` à un seul pin de sortie nommé `out`

— *PART* : Cette partie contient le programme que vous allez écrire pour implémenter le chips. Cette partie est composées d'une suite d'`instruction` (décrit ci-dessous) séparés par des points-virgules.

Le format d'une instruction est le suivant :

```
XXX(ipin1=var1, ipin2=var2, ..., opin1=var3, opin2=var4, ...)
```

La sémantique de cette instruction est qu'elle appelle le chip `XXX` :

— elle branche sur le pin d'entrée `ipin1` le pin `var1`, sur le pin d'entrée `ipin2` le pin `var2`.

— `var3` est le nouveau nom du pin de sortie `opin1`, et `var4` est le nouveau nom du pin de sortie `opin2`

Pour les deux premières sections, il est attendus que `var3` et `var4` soient des nouveaux noms (de nouvelles variables sont créees), tandis que `var1` et `var2` sont des noms qui doit déjà exister.

Par exemple, lorsque vous voulez créer un nouveau chip et vous souhaitez utiliser le chip `AND`, vous pouvez écrire l'instruction suivante :

```
...
AND(a=a1,b=a2, out=outand)
```

en supposant que les variables `a1` et `a2` existent déjà.

## 3 Les chips de base

On suppose dans cette partie que le chip `NAND` est déjà construit. Son interface est la suivante :

```
/**
 * Nand gate:
 * out = 1 if (a == 0 and b == 0)
 *       0 otherwise
 */

CHIP Nand {
```

```
    IN a, b;
    OUT out;

    PARTS:
    BUILTIN
}
```

C'est le seul chip que vous pourrez utiliser au début. Une fois que aurez implémenté correctement un chip, il sera utilisable pour l'implémentation des futurs chips bien entendu. Les commentaires au début de chaque fichier sont assez explicites et donc je ne rajouterai pas de commentaire. Cependant, je vous conseille d'implémenter vos chips dans l'ordre suivant :

1. `Not` (1 instruction)
2. `And` (2 instructions)
3. `Or` (3 instructions)
4. `Or8Way` (7 instructions)
5. `Xor` (5 instructions)
6. `Not16` (16 instructions)
7. `And16`
8. `Or16`
9. `Mux` (4 instructions)
10. `Mux16` (16 instructions)
11. `Mux4Way16` (3 instructions)
12. `Mux8Way16` (3 instructions)
13. `DMux` (3 instructions)
14. `DMux4Way` (3 instructions)
15. `DMux8Way` (5 instructions)

# 4 ALU

## Additionneur

À partir des briques de base que l'on a construites à la section précédente, on va essayer de construire une unité d'arithmétique logique (ALU). Notre unité d'arithmétique va être capable d'additionner deux entiers binaires codés sur 16 bits, mais pas seulement. Il pourrait incrémenter/décrémenter, faire des opérations bit à bit comme `x & y` ou `x | y`. Pour cela, un premier objectif va être de recoder une additionneur sur 16 bits. Pour cela, vous allez devoir implémenter les trois chips suivants :

1. `HalfAdder` (2 instructions)
2. `FullAdder` (3 instructions)
3. `Add16` (16 instructions)
4. (bonus) `Inc16` (1 instruction)

## ALU

Une fois ces portes implémentées, on va s'intéresser à l'ALU. L'idée de ce chip est de prendre deux vecteurs codés sur 16 bits en entrées $x$ et $y$ et de calculer en sortie un vecteur de 16 bits $out(x, y)$. $out$ étant une fonction qui est paramétré par $6$ flags :

— $zx$, met le vecteur $x$ à $0$


out(t) = in(t–1)

FIGURE 1 – DFF

— $nx$, inverse bit à bit le vecteur $x$
— $zy$, met le vecteur $y$ à $0$
— $ny$, inverse bit à bit le vecteur $y$
— $f$, si à $1$ calcule la somme des deux entrées, sinon fait un $ET$ logique bit à bit
— $no$, inverse bit à bit l'output

**Préliminaires :** quelques questions avant de vous lancer dans l'implémentation (l'ordre des flags est celui présenté ci-dessus) :

— Quelle est la fonction $out(x, y)$ quand on lui passe comme flag le vecteur $(0, 1, 0, 1, 0, 1)$ ?
— Quelle est la fonction $out(x, y)$ quand on lui passe comme flag le vecteur $(0, 0, 1, 1, 0, 1)$ ?
— Quelle est la fonction $out(x, y)$ quand on lui passe comme flag le vecteur $(1, 1, 1, 1, 1, 1)$ ?
— Est-il possible de calculer $x + 1$ ?
— Est-il possible de calculer $x - 1$ ?
— Est-il possible de calculer $x - y$ ?

**Output flag :** de plus, il vous est demandé de produire deux autres sorties :

— $zr$, est à $1$ si $out(x, y) = 0$, $0$ sinon
— $ng$, est à $1$ si $out(x, y) < 0$, $0$ sinon

**Implémentation :** maintenant, implémentez le chip suivant :

1. `ALU` (13 instructions)

# 5 Memory

Cette partie s'intéresse à des portes logiques dont le comportement dépend du temps. Autrement dit, un des pins de sortie peut se brancher sur un des pins d'entrées. Cela implique qu'en pratique, ce style de chip utilise une horloge, qui va réguler le signal électrique. Cependant, dans notre cas, et pour simplifier le design de nos chips, on ne va pas manipuler l'horloge directement. Au lieu de ça, on va considérer qu'on a un chip *built-in*, comme la porte `nand` qui nous est donné. Ce chip, appelé `DFF` recopie son entrée sur sa sortie à chaque *tick* d'horloge (voir Fig. 5). À noter qu'utiliser une horloge revient à *discrétiser* le temps.

Pour cette partie, je vous invite à programmer vos chips dans l'ordre suivant :

1. `Bit` (2 instructions)
2. `Register` (16 instructions)
3. `PC` (5 instructions)
4. `RAM8` (10 instructions)
5. `RAM64` (10 instructions)

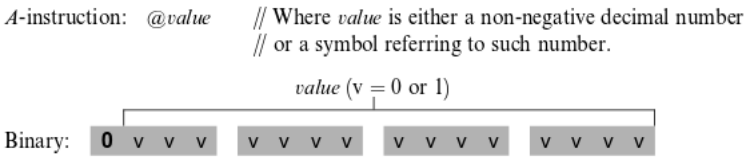


FIGURE 2 – A-instruction

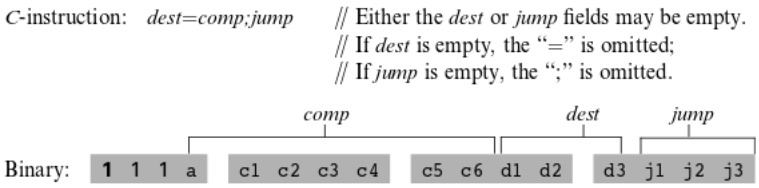


FIGURE 3 – A-instruction

6. `RAM512` (10 instructions)
7. `RAM4K` (10 instructions)
8. `RAM16K` (6 instructions)

# 6 Assembleur

Les instructions de notre ordinateur se codent sur 16 bits. Cette partie vous donne des informations sur le fonctionnement du CPU afin de savoir comment interpréter ces instructions. Le CPU utilise trois registres : `A`, `D` et `M`. Le registre `A` peut servir à référencer des endroits de la mémoire. Il a donc une instruction dédiée afin de lui passé des adresses de 15 bits présentée en Fig. 2. Le CPU gère un deuxième type d'instruction présentée en Fig. 3. Cette instruction va passer deux registres à l'ALU, détermine quelle opération de l'ALU est demandée et affecte le résultat à un ou plusieurs registres. De plus, cette instruction peut changer le *program counter* si on souhaite *jumper* vers une adresse du programme. Pratique pour coder un **if ... then ... else** ou bien un boucle **while**. Pour cette instruction, le registre `D` est toujours branché au bus $x$ de l'ALU. Le bit $a$ de l'instruction permet de dire si le bus $y$ sera branché au registre `A` ($a = 1$) ou bien au registre `M` sinon. Les bits $c1, \ldots, c6$ correspondent aux flags de l'ALU. Ils sont dans le même ordre que celui donné section 4. Les trois bits $d1, d2, d3$ indiquent quelles registres recevront le résultat de l'ALU : $d1$ pour le registre `A`, $d2$ pour le registre `D` et enfin $d3$ pour le registre `M`. Enfin, les trois $j1, j2, j3$ indiquent s'il faut modifier la valeur du *program counter* avec la valeur du registre `A`. Les différents cas sont résumés en Fig. 4

| 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 $out \geq 0$ jump |
| 1 | 0 | 0 | JLT | If $out < 0$ jump |
| 1 | 0 | 1 | JNE | If $out \neq 0$ jump |
| 1 | 1 | 0 | JLE | If $out \leq 0$ jump |
| 1 | 1 | 1 | JMP | Jump |

FIGURE 4 – jump

```
Chip Name:
Keyboard    // Memory map of the physical keyboard.
            // Outputs the code of the currently
            // pressed key.
Output:
out[16]     // The ASCII code of the pressed key, or
            // one of the special codes
Function:
Outputs the code of the key presently pressed on the
physical keyboard.
Comment:
This chip is continuously being refreshed from a
physical keyboard unit (simulators must simulate this
service).
```

FIGURE 5 – Keyboard

# 7 Computer

Dans cette partie, on va réutiliser la RAM construite à la partie précédente ainsi que notre ALU pour construire un *vrai* ordinateur. Notre ordinateur va utiliser deux périphériques externes : un clavier et un écran dont la spécification vous est donnée respectivement en Fig. 5 et Fig. 6. À noter que le Hardware Simulator gère les entrées/sorties pour vous. Dans notre cas, on ne s'occupera pas des entrées. Dans le Hardware Simulator, sous l'option **View** vous pouvez sélectionner **Screen** pour regarder ce que produisent certains tests (comme `ComputerRect-external.tst`). De plus, dans notre architecture, il faut charger notre programme en mémoire qui sera différent de la `RAM`. Ici, ce sera la `ROM`. Pour cela, on va aussi utiliser un chip *built-in* qui simulera ce composant voir Fig. 7. Je vous conseille d'implémenter les chips dans l'ordre suivant :

1. `Memory` (7 instructions)
2. `CPU` (20 instructions)
3. `Computer` (3 instructions)

Afin de vous aider, vous trouverez en Fig. 8, un schéma qui résume le comportement interne du CPU.

```
Chip Name:
Screen       // Memory map of the physical screen
Inputs:
in[16],      // What to write
load,        // Write-enable bit
address[13]  // Where to write
Output:
out[16]      // Screen value at the given address
Function:
Functions exactly like a 16-bit 8K RAM:
1. out(t)=Screen[address(t)](t)
2. If load(t-1) then Screen[address(t-1)](t)=in(t-1)
(t is the current time unit, or cycle)
Comment:
Has the side effect of continuously refreshing a 256
by 512 black-and-white screen (simulators must
simulate this device). Each row in the physical
screen is represented by 32 consecutive 16-bit words,
starting at the top left corner of the screen. Thus
the pixel at row r from the top and column c from the
left (0<=r<=255, 0<=c<=511) reflects the c%16 bit
(counting from LSB to MSB) of the word found at
Screen[r*32+c/16].
```

FIGURE 6 – Screen

```
Chip Name:
ROM32K            // 16-bit read-only 32K memory
Input:
address[15]       // Address in the ROM
Output:
out[16]           // Value of ROM[address]
Function:
out=ROM[address]  // 16-bit assignment
Comment:
The ROM is preloaded with a machine language program.
Hardware implementations can treat the ROM as a
built-in chip. Software simulators must supply a
mechanism for loading a program into the ROM.
```

FIGURE 7 – ROM

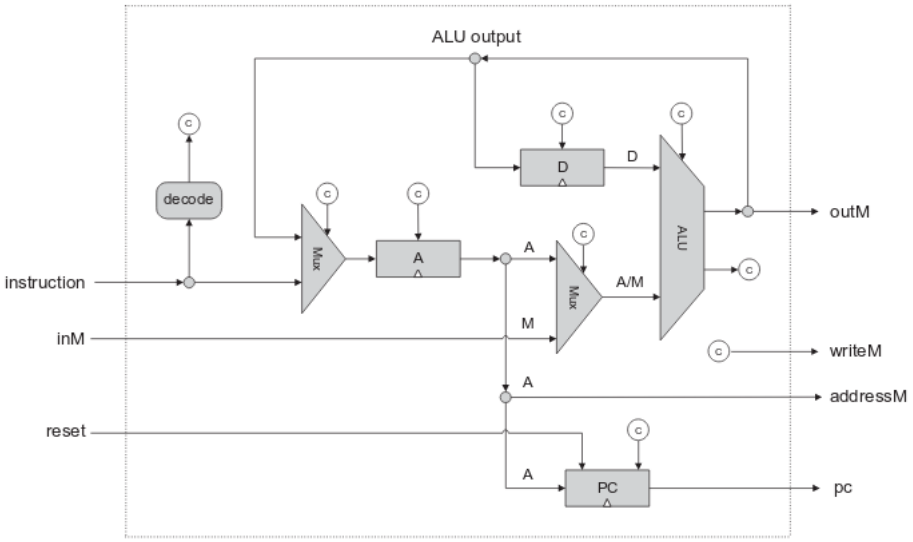
ALU output
decode
Mux
A
D
D
ALU
outM
instruction
A
Mux
A/M
inM
M
writeM
A
addressM
reset
A
PC
pc

FIGURE 8 – CPU