

TD 2 and 3 : Construction of a computer from NAND chips

1 Introduction

For this TP, you can find the zip folder containing all the requisite files here: <https://lmf.cnrs.fr/downloads/IsaVialard/bootstrap.zip>.

We will start from NAND chips and build increasingly complicated logical units until we have a computer capable of running programs.

For this lab, we will use a Hardware Simulator. Its role will be to check that your logical units (*chips*) are well implemented. In addition, it will also help you with your debugging your chips. (To launch it, use the command line `./tools/HardwareSimulator.sh` inside the `comparch` folder.)

The language in which you will write your chips is a DSL (Domain Specific Language), i.e. a language specifically dedicated to the writing of your chips. In the next section, this language will be briefly introduced so that you can start writing your first chips.

Each chip consists of three files:

- `XXX.hd1` : this is the file that you will write the code of your chips in.
- `XXX.tst` : this is a pre-written file which contains tests to verify if your code is right.
- `XXX.cmp` : this is also pre-written with the output of the test file.

The last two files are used by the Hardware Simulator to check the correctness of your chips. So for the most part, you will only edit files with the extension `.hd1`.

Using the Hardware Simulator

Once the Hardware Simulator is started, to check your chip you have to follow these steps:

- Click on the **load script** button to load the `.tst` file
- Click the **run** button (blue double-arrow) to run the tests

To save time, you can select the **no animation** button in the **animation** menu.

Plan

This TP is divided into 4 sections :

1. In Section 3, you have to implement the basic chipset : `And`, `Or`, ...
2. In Section 4, we are interested in the conception of an ALU (Arithmetic Logic Unit)
3. In Section 5, we are interested in non-sequential chips in order to create the memory (RAM) of our computer

2 Hardware Language

Hardware Language is a language that allows you to program chips. All the chips you are going to write respect the following format (example of the AND gate) :

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

CHIP And {
  IN a, b;
  OUT out;

  PARTS:
  //TODO
}
```

Each chip begins with a comment summarizing the expected behavior of the chip. Then, the code of a chip is divided into three parts:

- *IN* : this line gives names to the input pins. Here, the AND gate has two input pins named *a* and *b*
- *OUT* : this line gives names to the output pins. Here, the AND gate has a single output pin named *out*
- *PART* : This part contains the program you are going to write to implement the chips. This part is made up of a sequence of `texttt` instructions (described below) separated by semicolons.

The format of an instruction is as follows:

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

The semantics of this instruction is that it calls the chip *XXX* :

- it connects to the input pin *ipin1* the pin *var1*, to the input pin *ipin2* the pin *var2*.
- *var3* is the new name of the output pin *opin1*, and *var4* is the new name of the output pin *opin2*

For the first two sections, it is expected that *var3* and *var4* are new names (new variables are created), while *var1* and *var2* are names that must already exist.

For example, when you want to create a new chip and you want to use the AND chip, you can write the following instruction:

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

assuming that the variables *a1* and *a2* already exist.

3 Basic chips

We suppose in this section that the NAND chip is already constructed. Its interface is as follows:

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

*/

```
CHIP Nand {  
  IN a, b;  
  OUT out;  
  
  PARTS:  
  BUILTIN  
}
```

This is the only chip that you can use at the start. Once you have correctly implemented a chip, it can be used to implement later chips. The comments at the start of each chip are self-explanatory, so there are no additional comments required for the construction. However, the below order is a suggested order to efficiently implement the necessary chips:

The optimal number of components are written in the brackets in each of the components. Try to use *at most* those many components.

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

4 ALU

Adder

From the basic bricks that we built in the previous section, we will try to build a logical arithmetic unit (ALU). Our arithmetic unit will be able to add two binary integers coded on 16 bits, but not only. It could increment / decrement, do bitwise operations like $x \& y$ or $x | y$. For this, a first objective will be to recode a 16-bit adder. For this, you will have to implement the following chips:

1. HalfAdder (2 instructions)

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

ALU

Once the previous gates have been implemented, we are now interested in the ALU. The idea of this chip is to take two vectors encoded on 16 bits as inputs x and y and to calculate at output a vector of 16 bits $out(x, y)$. out is a function which is parameterized by 6 flags (as shown in Fig. 4).

```

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

```

Figure 1: The Arithmetic Logic Unit

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

Preliminaires : Remember that $-x = !x + 1$.

A few questions before embarking on the implementation (the order of the flags is the one presented above):

- What is the function $out(x, y)$ when we pass the flags as a vector (0, 1, 0, 1, 0, 1) ?
- What is the function $out(x, y)$ when we pass the flags as a vector (0, 0, 1, 1, 0, 1) ?
- What is the function $out(x, y)$ when we pass the flags as a vector (1, 1, 1, 1, 1, 1) ?
- Is it possible to calculate $x + 1$? If yes, what should be the value of the flags?
- Is it possible to calculate $x - 1$? If yes, what should be the value of the flags?
- Is it possible to calculate $x - y$? If yes, what should be the value of the flags?

Implementation : Now, implement the following chip:

5. ALU (15 instructions)

5 Memory

This part is interested in logic gates whose behavior depends on time. In other words, one of the output pins can be connected to one of the input pins. This implies that in practice, this style of chip uses a clock, which will regulate the electrical signal. However, in our case, and to simplify the design of our chips, we are not going to manipulate the clock directly. Instead, we'll assume that we have a *built-in* chip, like the `nand` gate given to us. This chip, called `DFF` copies its input to its output at each clock *tick* (see Fig. 5). Note that using a clock is the same as *discretizing* time.

(The Hardware Simulator is a Java program that isn't very smart about memory management. As a result, you should use the built-in versions of lower level RAM devices when constructing larger RAM devices (after you understand how to make the lower-level devices). Otherwise, the Hardware Simulator will recursively generate a ton of memory-resident software objects, each representing one of the parts that make up a typical RAM device. This may cause the simulator program to run slowly or to run out of memory. Hence, the folders have been divided into "a" and "b" to avoid this problem.)



$$\text{out}(t) = \text{in}(t-1)$$

Figure 2: DFF

For this section, program the chips in the following order:

1. `Bit` (2 instructions)
2. `Register` (16 instructions)
3. `PC` (5 instructions)
4. `RAM8` (10 instructions)
5. `RAM64` (10 instructions)
6. `RAM512` (10 instructions)
7. `RAM4K` (10 instructions)
8. `RAM16K` (6 instructions)