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: `http://perso.crans.org/nicomarg/teaching/ArchiSys/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.hdl` : 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 `.hdl`.

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. One 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.)

$$out(t) = in(t-1)$$

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)

6 Computer

In this part, we will reuse the RAM built in the previous part as well as our ALU to build a *real* computer. Our computer will use two external peripherals: a keyboard and a screen (which are built-in), and whose specification is given to you respectively in Fig. 8 and Fig. 9. Note that the Hardware Simulator manages the inputs /outputs for you. In our case, we will not take care of the entries. In the Hardware Simulator, under the option **View** you can select **Screen** to see what some tests produce (like `ComputerRect-external.tst`). Our architecture has two distinct memories blocks, one will store our data (RAM) and the other will store the program we are running(ROM). For the ROM, we will use a *built-in* chip that will simulate this component see 10.

In every cycle the CPU preforms one of the possible instructions. The instructions of the Hack CPU are coded in 16 bits. We now go over those instructions are and how they are interpreted by the CPU. The CPU uses two registers : `A`, and `D`, and the memory in the address accessed in the previous cycle which is denoted by `M(A)`. All the instructions are of two types "A-instruction" or "C-instruction".

A-instruction(Fig. 3) tells the CPU from which address to fetch `M(A)` from. This instruction is recognized by the 15th bit of the instruction being set to 0. The rest of the bits refer to the address in the memory we want to fetch for the next cycle. So for example the instruction 0101000000000101 tells the CPU to fetch the memory in the address 101000000000101 for the following cycle.

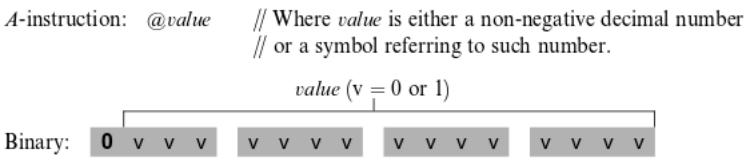

Figure 3: A-instruction

C-instruction(Fig. 4) this instruction is treated as a sequence of control bits that determine which function the ALU should compute, and in which registers the computed value should be stored. This instruction is recognized by the 15 – 13 bits of the instruction being set to 111. Next we break the instruction to three parts comp, dest, and jump, see 4.
The flags $c1, c2, \ldots, c6$ are going to the flags of the ALU, where :

$$c1 = zx; c2 = nx; c3 = zy; c4 = ny; c5 = f; c6 = no;$$

The flag a specifies which data should enter the input y of the ALU. If $a = 0$ then its the register `A` otherwise its `M(A)`. Note that, the input x of the ALU is always connected to `D`.
Next we have the flags $d1, d2, d3$ which specify where should the result of the ALU be stored. If the flag $d1$ is set to one then the the output is stored in `A`, if $d2$ is set to one then to `D`, and if if $d3$ is set to one then to `M(A)`. For example $(d1, d2, d3) = (1, 0, 1)$ tells the CPU to save the result in `A` and `M(A)`, but not in `D`.
The last set of flags are $j1, j2, j3$, which indicate whether we need to replace the value of the PC(program counter) by the value of `A`(jump to the instruction in address `A`). The jump happens according to the output of the ALU, for the different cases see Fig. 5.

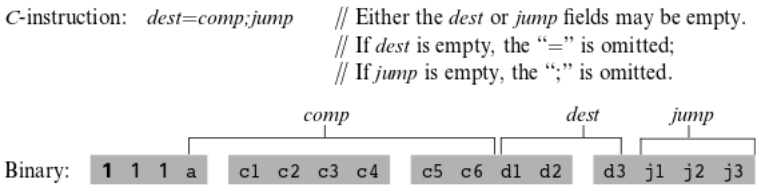

Figure 4: C-instruction

Now, to work: implement the chips in the following order:

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

The design of the CPU is also illustrated in, 6, (note that it misses some details).

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 5: The jump field of the C-instruction. Out refers to the ALU output (resulting from the instruction's comp part), and jump implies "continue execution with the instruction addressed by the A register."

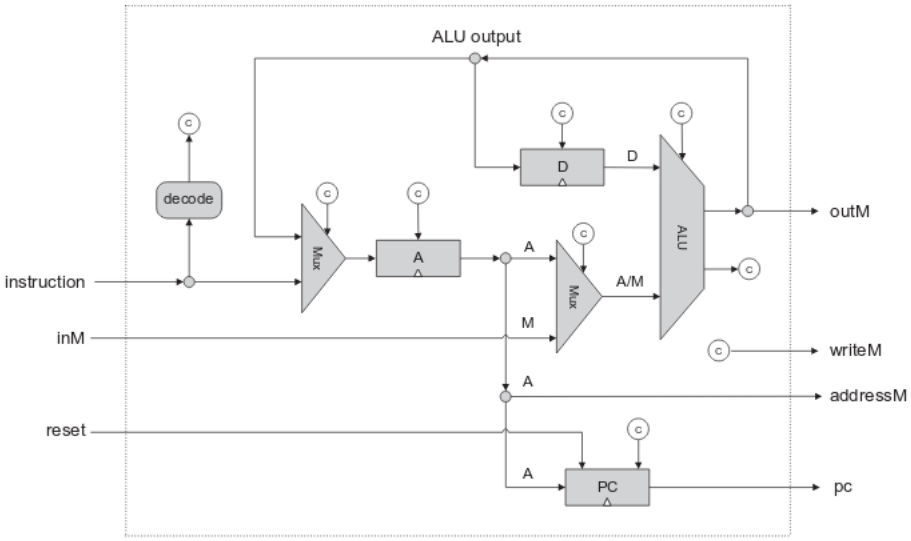

Figure 6: CPU

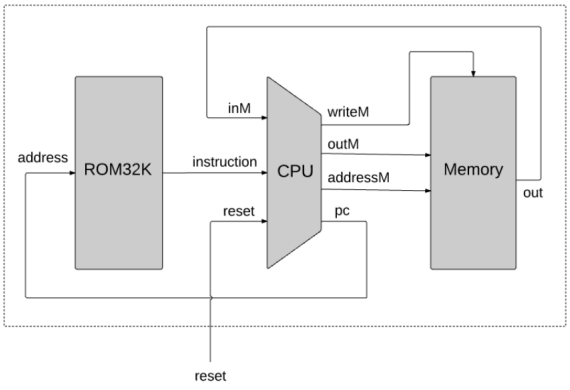

Figure 7: Computer

```
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 8: Keyboard

```
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 9: 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 10: ROM