TD 3 : Construction of a real computer

1 Introduction

For this TP, you can find the UPDATED zip folder containing all the requisite files here : `https://www.amritasuresh.github.io/teaching/bootstrap.tar.gz`.

Continuing from last week's TP, we will finish our endeavour of building a small computer starting from (almost) scratch.

2 ALU

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. 2).

```
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

Preliminaires : 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 :

1. `ALU` (15 instructions)

3 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. 3). 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)

4 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. 5 and Fig. 6. 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`). Implement the chips in the following order :

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

In order to help you, you will find in Fig. 3, a diagram which summarizes the internal behavior of the CPU, and also the computer.

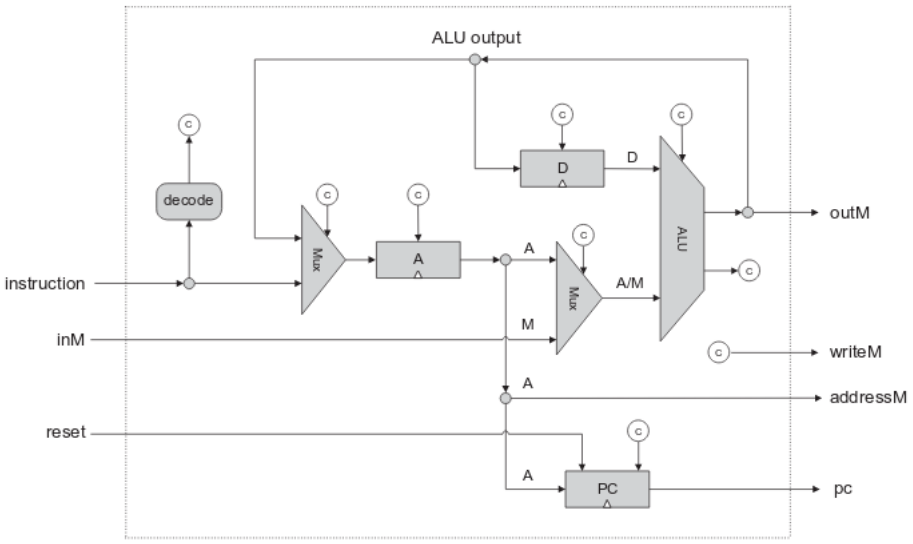

FIGURE 3 – CPU

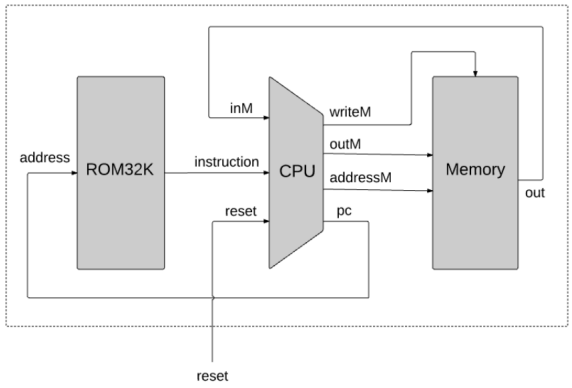

FIGURE 4 – Computer

In addition, in our architecture, we must load our program in memory which will be different from the `RAM`. Here it will be the `ROM`. For that, we will also use a *built-in* chip which will simulate this component see Fig. 7.

```
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

```
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

We now look more closely at the specifications of the machine language. Our objective is to come up with a logic gate architecture capable of (i) executing a given instruction, and (ii) determining which instruction should be fetched and executed next. In order to do so, the proposed CPU implementation includes an ALU chip capable of computing arithmetic/logical functions, a set of registers, a program counter, and some additional gates designed to help decode, execute, and fetch instructions. Since all these building blocks were already built in previous chapters, the key question that we face now is how to arrange and connect them in a way that effects the desired CPU operation.

The architecture shown in Fig. 3 is used to perform three classical CPU tasks : decoding the current instruction, executing the current instruction, and deciding which instruction to fetch and execute next. We now turn to describe these three tasks.

Instruction decoding

The 16-bit value of the CPU's instruction input represents either an `A-instruction` or a `C-instruction`. In order to figure out the semantics of this instruction, we can parse, or unpack it, into the following fields : `ixxaccccccdddjjj`. The `i`-bit (also known as opcode) codes the instruction type, which is either 0 for an `A-instruction` or 1 for a `C-instruction`. In case of an `A-instruction`, the entire instruction represent the 16-bit value of the constant that should be loaded into the A register. In case of a `C-instruction`, the `a`- and `c`-bits code the comp part of the instruction, while the `d`- and `j`-bits code the `dest` and `jump` parts of the instruction, respectively (the `x`-bits are not used, and can be ignored).

Instruction execution

The decoded fields of the instruction (`i`-, `a`-, `c`-, `d`-, and `j`-bits) are routed simultaneously to various parts of the CPU architecture, where they cause different chip-parts to do what they are supposed to do in order to execute either the A- or the C-instruction, as mandated by the machine language specification. In the case of a C-instruction, the single `a`-bit determines whether the ALU will operate on the A register input or on the M input, and the six `c`-bits determine which function the ALU will compute. The three `d`-bits are used to determine which registers should "accept" the ALU resulting output, and the three `j`-bits are used to for branching control, which can be seen in the following figures.

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 8 – The dest field of the C-instruction

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 9 – 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."