Architecture et Système Project 2021-2022
Département Informatique, ENS Paris Saclay

Project I: Computer Architecture

1 Project Guidelines

For the project, you need to send the completed files in the zipped folder available at `https://www.amritasuresh.github.io/teaching/project-logic-gates.tar.gz`. The assignments for Sections 2 and 4 require the `.hdl` files to be sent, along with the `.asm files` for Section 3. The deadline for the submission is **14h00 CET 22nd November 2021**.

2 Recap : Introduction to Logic Gates

2.1 Introduction

Using the chips from the TPs 2 and 3, we will now build some other components in a similar fashion. The `.hdl` along with the test cases are available in the `Project Logic Gates` folder of the zipped file. For highest points, try to implement the chips using the number of instructions specified. If you cannot find the optimal implementation, you will still be awarded partial points for a correct implementation.

2.2 Task

For this section, program the chips in the following order (assuming you have the chips from the TPs) :

1. `XNor gate` (2 instructions)
2. `Or16Way` (16 instructions)
3. `Subtractor` (4 instructions)
4. `Carry Lookahead Adder` (7 instructions)
5. `ALU single input` (4 instructions)
6. `ALU` (10 instructions)
7. `Priority encoder` (11 instructions)
8. `SR Flip Flop` (10 instructions)
9. `JK Flip Flop` (10 instructions)

3 Machine Language Programming

3.1 Overview

In computer programming, machine code is any low-level programming language, consisting of machine language instructions, which is used to control a computer's central processing unit (CPU). Each instruction causes the CPU to perform a very specific task, such as a load, a store, a jump, or an arithmetic logic unit (ALU) operation on one or more units of data in the CPU's registers or memory. Each hardware platform is designed to execute a certain machine language, expressed using agreed upon binary codes. Writing programs directly in binary code is a possible, but it is extremely tedious. Instead, we can write such programs in a low-level symbolic language, called assembly, and have them translated into binary code by a program called an assembler. In this project, you will write some low-level assembly programs, and will be forever thankful for high-level languages like C and Python. (Actually, assembly programming can be a lot of fun, if you are in the right mood ; it's an excellent brain teaser, and it allows you to control the underlying machine directly and completely.)

3.2 Introduction

To get a taste of low-level programming in machine language, and to get acquainted with the computer platform which we have been constructing for the past two TPs. In the process of working on this project, you will become familiar with the assembly process — translating from symbolic language to machine-language — and you will appreciate visually how native binary code executes on the target hardware platform. Moreover, you will get acquainted with how the instructions are written and interpreted by the machine. These lessons will be learned in the context of writing and testing the two low-level programs described below.

3.3 Task

Write and test the following program described below. When executed on the CPU Emulator, your program should generate the results mandated by the specified tests.

Description :

`mult.asm` : Multiplication : In the computer that we build, the top 16 RAM words (RAM[0]...RAM[15]) are also referred to as R0...R15.

With this terminology in mind, this program computes the value R0*R1 and stores the result in R2.

The program assumes that R0$>= 0$, R1$>= 0$, and R0*R1< 32768. Your program need not test these conditions, but rather assume that they hold.

Guidelines

— Use a plain text editor to write your `mult.asm` program using the assembly language specified in Appendix A.

— Use the supplied Assembler to translate your `mult.asm` program, producing a file containing binary instructions.

— Next, load the supplied `mult.tst` script into the CPU Emulator. This script loads the Mult program, and executes it.

— Run the script. If you get any errors, debug and edit your `mult.asm` program. Then assemble the program, re-run the `mult.tst` script, etc.

3.4 Resources

The assembly language is described in detail in Appendix A. You will need two tools : the supplied Assembler — a program that translates programs written in the assembly language into binary code, and the supplied CPU Emulator — a program that runs binary code on a simulated platform. Two other related and useful resources are the supplied Assembler Tutorial and the CPU Emulator Tutorial. I recommend going through these tutorials before starting to work on this project. The project files are available in a ZIP archive file available on the course web site.

4 Computer

In this part, we will finish our TP assignment. We will reuse the RAM built in the previous TP 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. 3 and Fig. 4. 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. 1, a diagram which summarizes the internal behavior of the CPU, and also the computer.

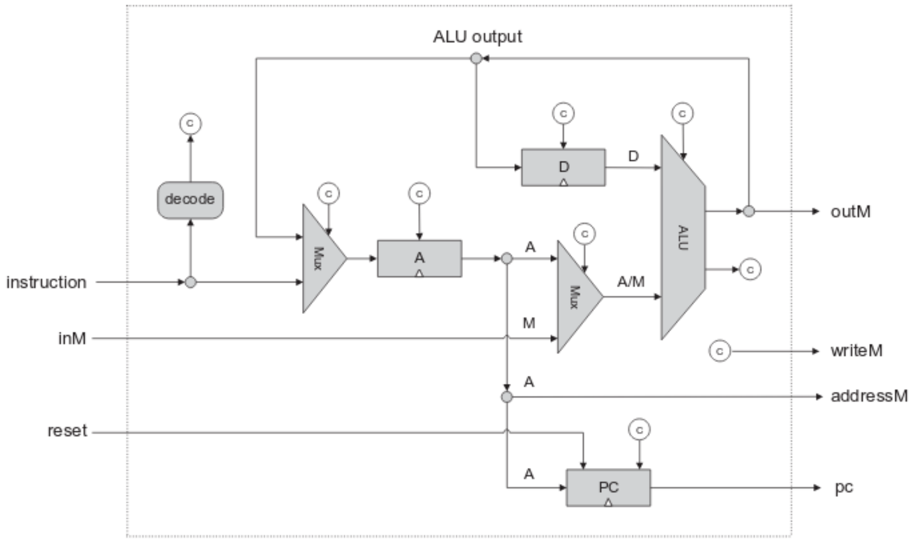

FIGURE 1 – CPU

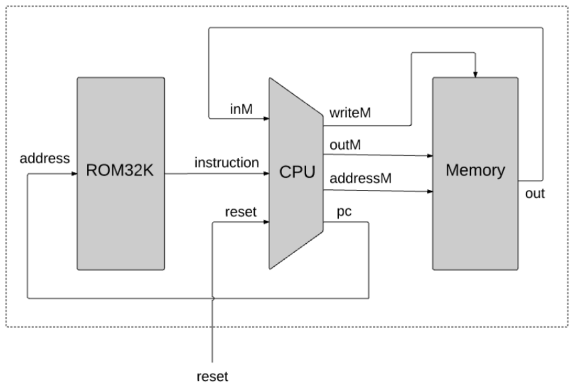

FIGURE 2 – Computer

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. 5.

```
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 3 – 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 4 – 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 5 – 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. 1 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.

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 6 – 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 7 – 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."