Project I: Computer Architecture

1 Project Guidelines

For the project, you need to complete the files in the zipped folder available at `https://lmf.cnrs.fr/downloads/IsaVialard/comparch.zip`. The assignments for Sections 2 and 4 require the `.hdl` files to be sent, along with the `.asm` file for Section 3. The deadline for the submission is **18th November 2022**. Each question is marked with points. Don't worry about them too much, it is more a guideline for the effort required per question, as opposed to the grade.

To launch the Hardware Simulator, use the command line ./tools/HardwareSimulator.sh inside the comparch folder. The folder with the files you need to complete is `assignment/project`. The other folders in `assignment` are a correction of the chips of TP 2 and 3.

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. Note: All files related to this exercise are available on `comparch/assignment/project/logicgates`.

2.2 Task

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

1. `Or16Way` (3 instructions) `1 point`
2. `CarryLookahead4` `4 points`

   This is a 4-bit adder with the lookahead logic that you have seen in class. To learn more, refer to the circuit description available in the folder `logicgates`. *Note:* Any implementation will be awarded full points. Feel free to add other chips if it simplifies things for you.
3. `UnaryALU` (4 instructions) `2 points`
4. `ALU` (9 instructions) `4 points`

   With some chips constructed earlier in this task, we can build a more compact ALU.
5. `Subtract16` (1 instruction) `1 points`

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 analyzing the low-level programs described below.

3.3 Resources

Note: All files related to this exercise are available on `comparch/assignment/project/machinelanguage`. The assembly language is described in detail in the instruction manual in the `Machine Language` folder of the Project. I suggest taking a look at this tutorial before you start this section of the project.

To run your program, 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. These are included in the zipped file. To use the GUI of the CPU Emulator, run `tools/CPUEmulator.sh`. You can have a look at the CPU Emulator Tutorial in case you want to get an idea of how it works. If not, you can just learn by running the tool and experimenting.

3.4 Task

3.4.1 Warmup

We now move to the `Machine Language` folder of our Project. To first get a little acquainted with the machine language, let us look at some more examples. What do the following programs in machine language compute?

1.
```
@2
D=A
@3
D=D+A
@0
M=D
```

2 points

2.
```
@R0
D=M              // D = first number
@R1
D=D-M            // D = first number - second number
@OUTPUT_FIRST
D;JGT            // if D>0 goto output_first
@R1
D=M              // D = second number
@OUTPUT_D
0;JMP            // goto output_d
(OUTPUT_FIRST)
@R0
D=M              // D = first number
(OUTPUT_D)
@R2
M=D              // M[2] = D
(INFINITE_LOOP)
@INFINITE_LOOP
0;JMP            // infinite loop
```

3 points

3.4.2 Multiplying 2 Numbers

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. 5 points

Description:

`mult.asm` : 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.

```
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 1: Screen

4 Computer

4.1 Introduction

In previous projects we've built the computer's basic processing and storage devices (ALU and RAM, respectively). In this project we will put everything together, yielding the complete Hardware Platform. The result will be a general-purpose computer that can run any program that you fancy.

4.2 Task

Note: All files related to this exercise are available on `comparch/assignment/project/computer`.

Implement the chips in the following order:

1. `Memory` (6 instructions) `5 points`

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. 1 and Fig. 2. 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`).

```
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 2: Keyboard

2. `CPU` (19 instructions) `10 points`

Our objective is to come up with a logic gate architecture capable of (i) decoding an instruction (ii) executing this instruction, and (iii) 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 and a program counter (represented by the built-in chips `ARegister`, `DRegister` and `PC` (see Fig. 4)), 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. In order to help you, you will find in Fig. 3, a diagram which summarizes the internal behavior of the CPU.

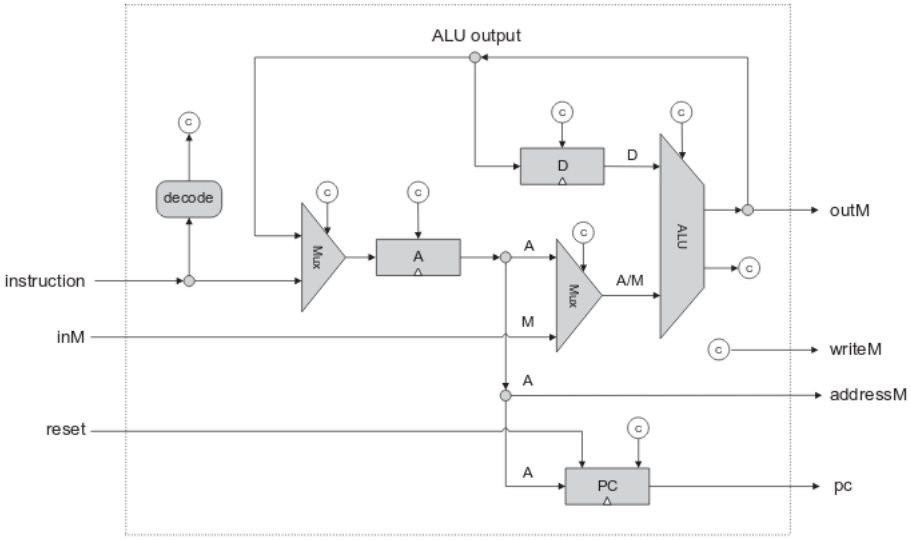

Figure 3: CPU

[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:

$$ixxac_1c_2c_3c_4c_5c_6d_1d_2d_3j_1j_2j_3\ .$$

```
Chip Name:
PC // 16-bit counter with load and reset controls.
Input:
in[16], load, inc, reset
Output:
out[16]
Function:
If reset(t-1) then out(t) = 0
   else if load(t-1) then out(t) = in(t-1)
         else if inc(t-1) then out(t) = out(t-1) + 1 (integer addition)
             else out(t) = out(t-1)
```

Figure 4: PC

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). See the instruction manual in the `Machine Language` folder to see how to decode the `comp`, `dest` and `jump` parts.

Now you are finally ready to put everything together ! The computer chip is straightforward, just follow the diagram in Fig. 5.

3. `Computer` (3 instructions) `3 points`

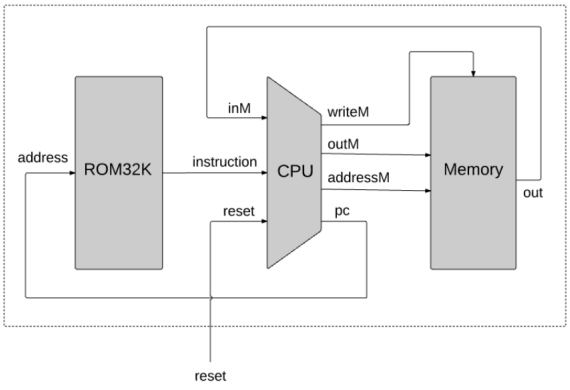

Figure 5: Computer

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. 6.

```
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 6: ROM