Lab 1
Warm-up : discovering the target machine, DIGMIPS

Credits

This sequence of compilation labs has been designed by C. Alias and G. Iooss in 2013/14. Only minor cosmetic changes have been done since.

Objective

- Be familiar with the digmips instruction set.
- Understand how it executes on the digmips processor.
- Write simple programs, assemble, execute.
- Use digeval to simulate the processor execution.

EXERCISE #1 ▸ **Configuration**
Copy the archive for Lab1 and also the DIGMIPS files:

```
cp /home/lgonnord/M1-Compilation/archive_tp1.tgz <destinationdir>
cp -r /home/calias/M1-Compilation/digmips/ <destinationdir>
```

Also add the following variables to your `$PATH` system environment variable, by adding the following lines to your `~/.bahsrc`:

```
export PATH=$PATH:/home/calias/M1-Compilation/asm/bin/
export PATH=$PATH:/home/lgonnord/M1-Compilation/digeval/bin/
export PATH=$PATH:/home/lgonnord/M1-Compilation/asm/bin/
```

Open a new terminal to test the new environment variable (`echo $PATH$`).

1.1 The DIGMIPS processor

Your advisor will show you a demo of the DIGMIPS processor, implemented in the DIGLOG software. Appendix A contains screenshots of the processor. You will also find the instructions to replay the demo by yourself. [1]

Our processor works with a **8KB data memory** (addressable on 13 bits) and has **8 8-bit registers labeled from r0 to r7**. It operates using 16-bit instructions (see appendix). They will be detailed later on. Programs are stored inside the instruction memory, which is split in two 8 bit SRAMs:

- The memory on the right contains the **8 least significant bits** of each instruction
- The memory on the left contains the **8 most significant bits** of each instruction

DIGLOG allows **loading data into memory** with information stored in **a file**. The data inside the file must be in hexadecimal form for it to be loaded into the available RAMs. Because of this, **two files** are necessary: one file for the RAM on the left containing the 8 most significant bits of each instruction and another one for the RAM on the right containing the 8 least significant bits of each instruction.

The **hello_hi.ram** and **hello_lo.ram** files of the archive contain the encoding of a program which prints "Hello" on the screen :

[1] This activity may take time, if you do not manage to do it, skip, and go to the next section.

```
0000:A060A060A060A060A060A060A060A060
0010:A060A060A060A060A060E0
```

and

```
0000:483F653F6C3F6C3F6F3F203F773F6F3F
0010:723F6C3F643F213F0D3F1A
```

EXERCISE #2 ▸ **Demo**
Observe the procedure to execute the program on DIGLOG.

1.2 Programming with 1s and 0s

The different instructions of our processor are described in Section A.4. The encoding on 16 bits is specified in the Appendix A.

EXERCISE #3 ▸ **TD, manual assembling**
On paper, write (in DIGMIPS assembly language) a program which initializes the r0 register to 1 and increments it until it becomes equal to 10.

1. Using the appendix (Figure A.8), give the encoding of your program in binary and in hexadecimal form. (you can use a binary to hex web translator if you want). You can use labels.
2. Construct the corresponding `.ram` files and load your program inside DIGMIPS. Execute step by step. Be careful with labels and relative addresses.

OR

2bis Compare the solution with the one effectively executed by your instructor.

1.3 Assembling

We have written a tool that automatically generates the `.ram` files starting from a program written in machine language. Such a tool is called an *assembler*. Here is a an example of program accepted by our assembler:

'star.asm'

```
    ldi r1,1
    ldi r2,1            // counter = 1
    ldi r3,10           // max = 10

loop:
    ble r2,r3,loop_stmt
    j end_loop          // counter > 10? => stop
loop_stmt:

    ldi r4,'*'
    st r4,[r0+63]       // print a star

    add r2,r2,r1        // counter = counter + 1
    j loop              // next iteration

end_loop:
    j end_loop
```

`loop` and `end_loop` are *labels*. They help identify execution points inside the program to where we want to perform jumps. The assembler is in charge of computing the correct offset for ble and the absolute address for j.

EXERCISE #4 ▶ **Assembling a simple digmips assembly file**
Find `affiche10.asm` in the archive. Call the assembler tool [2] using `asm affiche10.asm`. The assembler outputs:

- The corresponding hexadecimal code on the standard output.
- Two **affiche10_hi.ram** and **affiche10_lo.ram** which can be loaded into memory inside DIGMIPS.

Then :

- Your instructor will show you the execution on the DIGMIPS processor.
- Run DIGEVAL on `affiche10.asm`. From now, we will use this evaluator instead of DIGLOG.
- Do the same with `star.asm`.

1.4 Input/Output

DIGMIPS also has an extra file (io.lgf, tab 5) which allows one to input data using a keyboard and output data on a screen. Normally, we would use two dedicated instructions for input/output operations. Because we can only have 8 instructions, the decision was made to recycle the ld and st instructions in the following manner:

- **To output a character on the screen.** Place the ASCII code of the character to be printed inside r and execute st r,[whatever_register_name + 63]. The base register is not important, what matters is that the immediate value be 63. For example:

output

```
ldi  r0,'a'       // Places the ASCII code of 'a' inside r0
st   r0,[r7+63] // Outputs the content of r0.
```

- **To read a character from the keyboard.** In a similar fashion, to read a character, it suffices to execute: ld r,[whatever_register_name + 63]. For example:

read

```
ld  r,[r7+63] // Reads the state of the keyboard.
```

The characters read from the keyboard are stored in size 4 buffer. It is possible that no character is available to be read. In this case, the value 0 is read. As such, to read from the keyboard we must loop until a character becomes available.

EXERCISE #5 ▶ **I/O**
Write a program `reading.asm` which outputs "nb?", passes to the next line, reads an integer from 0 to 9 from the keyboard and outputs it on the next line. The program ends by printing the character 10 on a new line. Observe the demo on DIGLOG and use DIGEVAL to evaluate.

1.5 More advanced assembly code!

EXERCISE #6 ▶ **Algo in DIGMIPS assembly**
Write and execute the following programs in assembly :
- Draw squares and triangles of stars (character '*') of size n, n being given by the user.
- Count the number of non-nul bits of a given integer.

[2] the program `asm` should be in your PATH

Appendix A
Diglog and Digmips

A.1 Installing Diglog and getting started

You'll find DIGLOG and DIGMIPS sources and binaries on the following webpage :

`http://perso.ens-lyon.fr/christophe.alias/archi_lyon1.html`

On the ENS machines, simply add :

`/home/calias/M1-Compilation/diglog/bin`

to your `$PATH` variable and copy all files in

`/home/calias/M1-Compilation/digmips/`

on your account.

A.2 Howto to load a program into Digmips

DIGLOG is a piece of software which is challenging to use, so ben kind with it!. In particular, the `VerrNum` button must be off. Use arrows to navigate inside a tab, and `Shift+number` to swich tabs.

To load a file inside a SRAM, you have to enter **CNFG** mode (bottom right of the window) and click on the SRAM. Inside the *newcrt* window use the cursor to **descend** to *file name to load* by clicking on the bottom arrow and entering the name of the file to load. To finish, press space to validate your choice and go back to the *mylib* window.

Try it yourself.

- **Load the "Hello" program**. **Place the reset switch on 1** (en under the clock) to reset the instruction counter to zero. To launch the application, **place the reset switch to 0**. Use tab 5 (input/output) to **observe the result**.
- **Set reset to 1**. Use a switch instead of the clock. **Set reset to 0** and click on the switch to execute the program **step-by-step**.

A.3 Screenshots of the Digmips processor

The processor is composed of several files:

- **base.lgf** (tab 2) contains all the base components (multiplexers, decoders, etc (Fig A.1)
- **alu.lgf** (tab 1) contains the 8-bit ALU (Fig A.2)
- **register.lgf** (tab 6) contains 8 8-bit registers (Fig A.3)
- **datapath.lgf** (tab 4) contains the processor *datapath*. (Fig A.4, Fig A.5)
- **control.lgf** (tab 3) contains the *control circuit* for the processor (Fig A.6)
- **io.lgf** (tab 5) contains a screen and a small keyboard used for I/O operations. (Fig A.7)

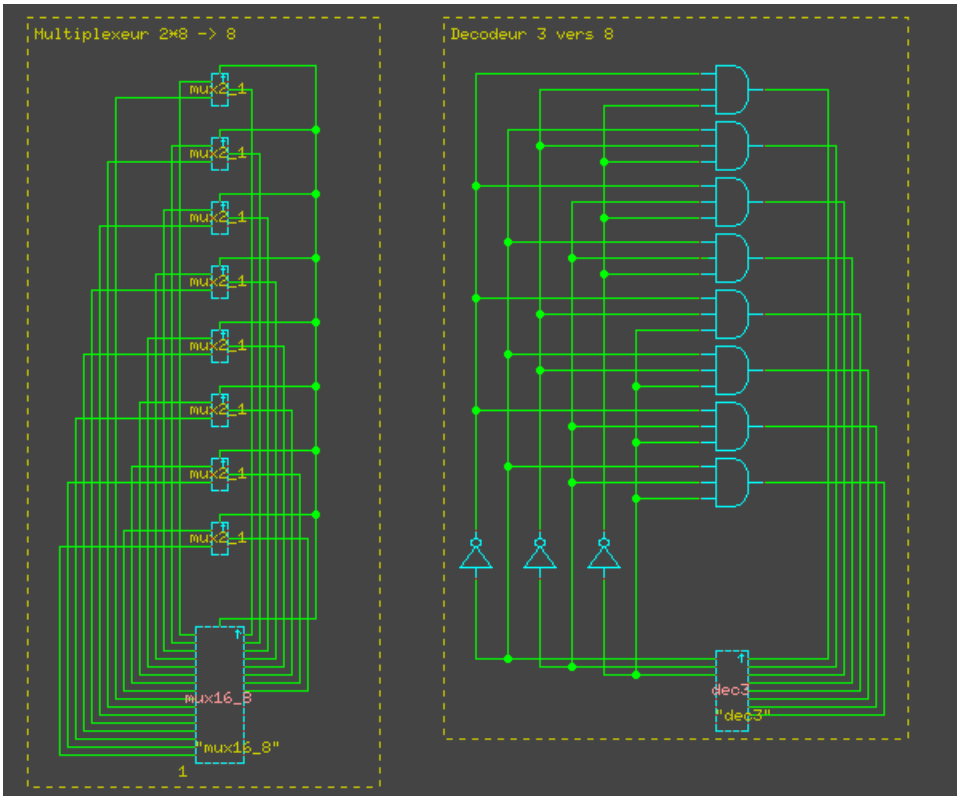

Figure A.1: mux and decoder

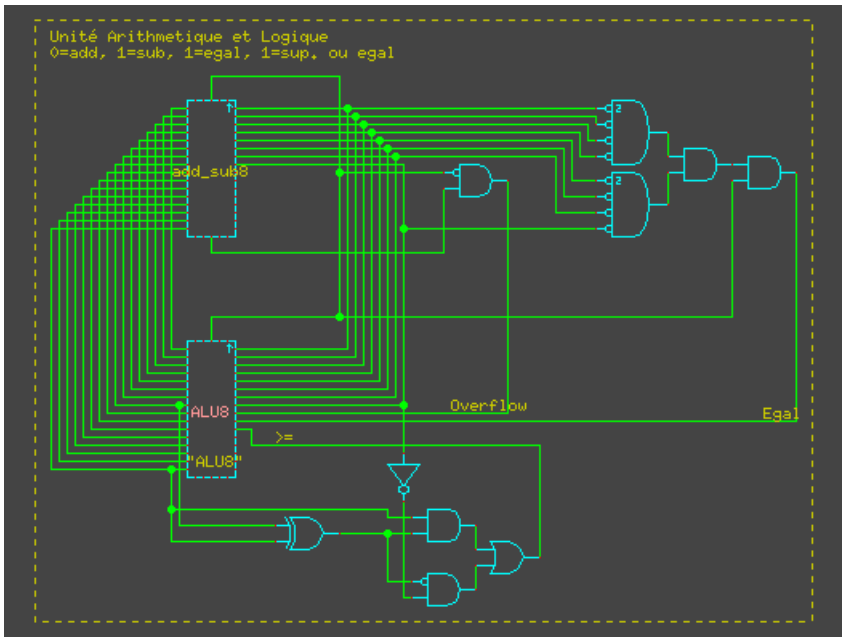

Figure A.2: ALU

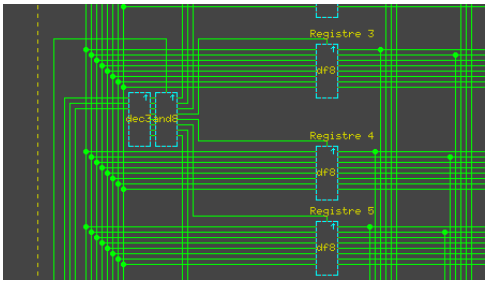

Figure A.3: Registers (part)

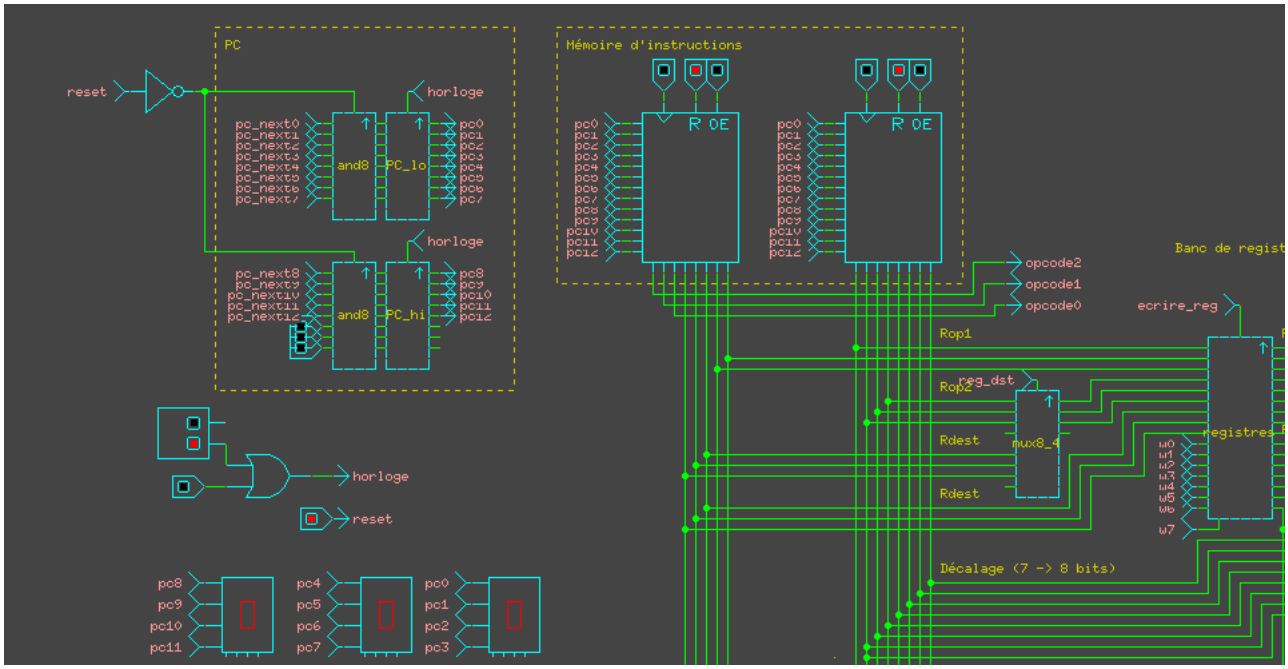

Figure A.4: Datapath 1 : PC + instruction memory

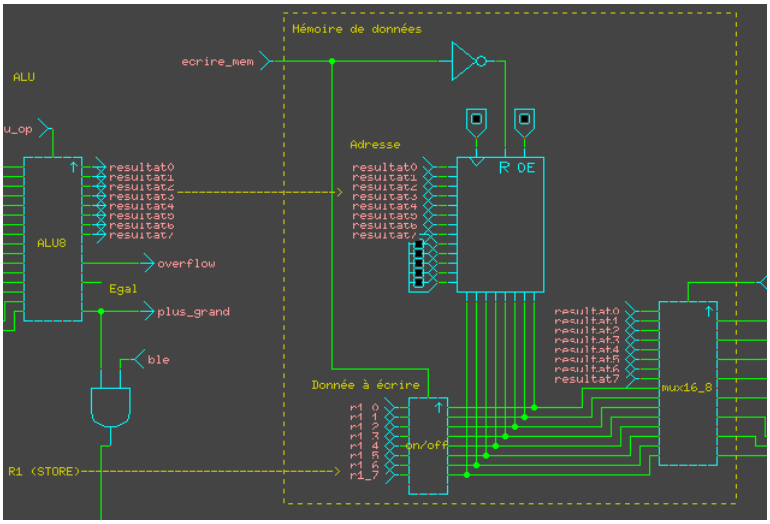

Figure A.5: Datapath 2 : ALU + data memory

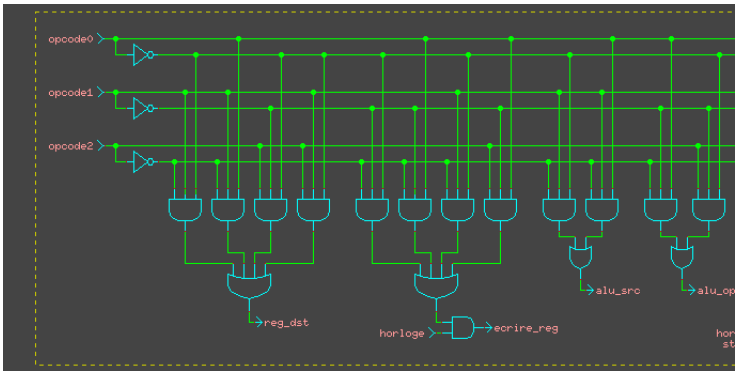

Figure A.6: Control Unit

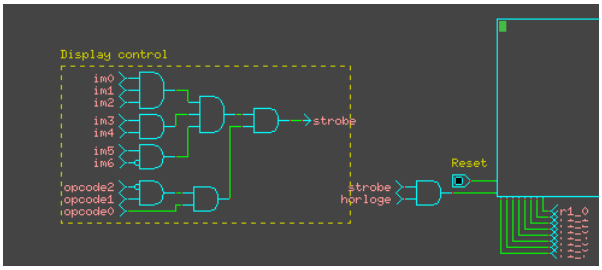

Figure A.7: Screen

A.4 ISA for Digmips

Our processor has the following instructions:

- **add $\mathbf{r_{dest}}, \mathbf{r_1}, \mathbf{r_2}$**: adds the content of registers r_1 and r_2, and stores the result inside the r_{dest} register.
- **sub $\mathbf{r_{dest}}, \mathbf{r_1}, \mathbf{r_2}$**: computes $r_1 - r_2$ and places the result inside the register r_{dest}.
- **ld $\mathbf{r_{dest}}$, [$\mathbf{r_{base}}$ + **imm7**]**: loads into r_{dest} the data located in memory at the address r_{base} + imm7, where imm7 is a 7-bit integer. We also talk about an *immediate value* since the integer is directly available inside the instruction.
- **st $\mathbf{r_1}$, [$\mathbf{r_{base}}$ + **imm7**]**: stores the value located in the r_1 in memory at the address r_{base} + imm7.
- **ble $\mathbf{r_1}, \mathbf{r_2}$, imm7**: if $r_1 \leq r_2$, jumps to the instruction located at the current address plus imm7 + 1. (if not, continue to the next instruction located at the current address plus 1). You can use *labels* to replace imm7, the adequate value will be computed during the assembling phase. This instruction allows us to implement **for** and **while** loops, as well as **if**.
- **ldi $\mathbf{r_{dest}}$, imm8**: writes the 8 bit integer imm8 inside the r_{dest} register.
- **ja $\mathbf{r_1}, \mathbf{r_2}$**: jumps to the 13 bit address defined by r_1 for the 8 least significant bits and by *r b* for the 5 (most significant) remaining bits.
- **j imm13**: jumps to the 13 bit address imm13, where imm13 is a 13-bit integer. This instruction, along with `ja`, allows for the implementation of function calls.

Instruction encoding The instructions are encoded using 16 bits, using the correspondance table in Figure A.8. The '-' character specifies unused bits.

Instruction	Encoding
`add` r_{dest}, r_1, r_2	15 14 13 | 12 11 10 | 9 8 7 | 6 5 4 | 3 2 1 0 0 0 0 | | | | | | - - - - opcode | r_{dest} | r_1 | r_2 |
`sub` r_{dest}, r_1, r_2	15 14 13 | 12 11 10 | 9 8 7 | 6 5 4 | 3 2 1 0 0 0 1 | | | | | | - - - - opcode | r_{dest} | r_1 | r_2 |
`ld` $r_{dest}, [r_{base} + imm7]$	15 14 13 | 12 11 10 | 9 8 7 | 6 5 4 3 2 1 0 0 1 0 | | | | | | | | | | | | | | | | opcode | r_{dest} | r_{base} | imm7
`st` $r_1, [r_{base} + imm7]$	15 14 13 | 12 11 10 | 9 8 7 | 6 5 4 3 2 1 0 0 1 1 | | | | | | | | | | | | | | | | opcode | r_{dest} | r_{base} | imm7
`ble` r_1, r_2, imm7	15 14 13 | 12 11 10 | 9 8 7 | 6 5 4 3 2 1 0 1 0 0 | | | | | | | | | | | | | | | | opcode | r_1 | r_2 | imm7
`ldi` r_{dest}, imm8	15 14 13 | 12 11 10 | 9 8 | 7 6 5 4 3 2 1 0 1 0 1 | | | - - | | | | | | | | | opcode | r_{dest} | | imm8
`ja` r_1, r_2	15 14 13 | 12 11 10 | 9 8 7 | 6 5 4 3 2 1 0 1 1 0 | | | | | | - - - - - - - opcode | r_1 | r_2 |
`j` imm13	15 14 13 | 12 11 10 9 8 7 6 5 4 3 2 1 0 1 1 1 | | | | | | | | | | | | | | | | opcode | imm13

Figure A.8: Instruction Set Architecture for DIGMIPS

Remarks on data Some assembler come with assembly directive like `.END` or `.FILL` that enable to stop the execution of a given program or fit the main memory with data such as strings, for instance. Here our toy assembly doesn't provide such issues, so data must be computed inside the program.