Compilation

TP 0.0 : The target architecture: Digmips

C. Alias & S. Filip

The goal of these TPs is to construct a C compiler for Digmips, a small MIPS processor implemented using the Diglog logic simulation software. The purpose of this TP is to start working with Diglog and Digmips, by constructing some programs in assembly language.

Exercise 1. Diglog *warm up*

Add the directory `/home/calias/M1-Compilation/diglog/bin` to your `PATH` environment variable. Open a terminal and enter `diglog`.

- Diglog is comprised of two windows called *mylib* and *mycrt*. *mylib* lets you construct a circuit, while *mycrt* is an auxiliary window used for interaction (load/save, etc.). Diglog allows for an intuitive way for drawing circuits with the help of a GUI.
- At the bottom of the *mylib* window, there is an area containing some logic gates and connectors. This bar allows you easy access to the logic gates you use most frequently. To place a logic gate inside this bar, simply click on it and drag it to the desired place. If you place a gate on an already existing gate, the new one will take its place.
- On the left and on the right there are menus, but most of the commands can be launched from the keyboard, which is usually faster.
- For **placing a logic gate**, click on the gate inside the bottom bar and drag it to the desired location inside the window.
- For **adding a wire**, use the mouse like in a usual drawing application. By click on the left mouse button you start drawing a wire, mark an angle or specify the end of the wire. When you are done tracing the wire, you must click on the right mouse button.
- To **set the boolean value transmitted through a wire**, we can use a *generator*, meaning the component situated on the left of the bottom bar. By clicking in the center of the generator, we can toggle its value between 0 and 1.
- To **visualize the value transmitted through a wire**, we can use a *probe*. It's the square component situated on the right of the generator. We can just as easily pass to *glow mode* by using the g key. To exit glow mode, press g again.
- To **use other logic gates**, you can use the *catalogue*. It is available by clicking on CAT (located bottom left) and afterwards on the chosen component. If this is not enough, you can search inside the component library by clicking on LIBR. A list of components will be displayed inside the *newcrt* window. You can navigate through the different component categories by using the +, - keys. To select a component, it suffices to click under it. To quit, press the space key.
- Diglog is composed of **tabs**, accessible through the numeric keys 1, 2, 3, etc.
- To **save the current tab inside a file**, use Shift+S. Enter the name of the file inside the *newcrt* window.
- To **load a file inside the current tab**, use Shift+L. Enter the name of the file inside the *newcrt* window.
- To **quit** Diglog, enter Shift+Q. A confirmation box will appear inside the *newcrt* window.

Try it yourself.

- **Construct a 4-bit AND circuit**. Add the emitters. Experiment using *glow-mode*.

- **Insert a clock**. Set the frequency at one cycle per second. To do this, **go in CNFG mode** (bottom right) and click on the clock. The clock parameters will be displayed inside the *newcrt* window. Modify the values using the arrow keys. Validate your choice using the space key and return to the main window.

- **Save** your work inside a file called `tp1.lgf`.

Exercise 2. DIGMIPS *warm up*

Copy the content of the directory `/home/calias/M1-Compilation/digmips/` inside your account. Go to your `digmips/` directory, and use the command `diglog *.lgf` to load the MIPS processor.

The processor is composed of several files:

- **base.lgf** (tab 2) contains all the base components (multiplexers, decoders, etc.)

- **alu.lgf** (tab 1) contains the 8-bit ALU.

- **register.lgf** (tab 6) contains 8 8-bit registers.

- **datapath.lgf** (tab 4) contains the processor *datapath*.

- **control.lgf** (tab 3) contains the *control circuit* for the processor.

- **io.lgf** (tab 5) contains a screen and a small keyboard used for I/O operations.

Our processor can work 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 contain the encoding of a program which prints "Hello" on the screen.

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**.

Exercise 3. *Programming with 1s and 0s*

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 $\mathbf{r_{dest}}, [\mathbf{r_{base}} + \mathbf{imm7}]$**: loads into r_{dest} the data located in memory at the address $r_{base} + \text{imm7}$, where imm7 is a 7-bit integer. We also talk about an *immediate value* since the integer is directly available inside the instruction.
- **st $\mathbf{r_1}, [\mathbf{r_{base}} + \mathbf{imm7}]$**: stores the value located in the r_1 in memory at the address $r_{base} + \text{imm7}$.
- **ble $\mathbf{r_1}, \mathbf{r_2}, \mathbf{imm7}$**: if $r_1 \leq r_2$, jumps to the instruction located at the address $\text{imm7} + 1$. (if not, continue to the next instruction located at the current address plus 1). This instruction allows us to implement **for** and **while** loops, as well as **if**.
- **ldi $\mathbf{r_{dest}}, \mathbf{imm8}$**: 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_2 for the 8 least significant bits and by r_1 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.

Each instruction is encoded on 16 bits as specified in the appendix.

Try it yourself.

- **On paper, write a program** which initializes the r0 register to 1 and increments it until it becomes equal to 10.
- Using the appendix, **give the encoding of your program** in binary and in hexadecimal form.
- **Construct the corresponding .ram files** and **load** your program inside Digmips. **Execute step by step**.

Exercise 4. *Programming in assembly*

There is a tool which 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 program accepted by our assembler:

```
      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` et `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.

Try it yourself.

- Add the directory `/home/calias/M1-Compilation/asm/bin/` to your `$PATH` environment variable.
- Write the previous program inside a file and save it to `affiche10.asm`.
- Call the assembler 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.
- Test the resulting assembler files inside Digmips.

Exercise 5. *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 doing 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,[does_not_matter_what_register + 63]. The base register is not important, what matters is that the immediate value be 63. For example:

  ```
  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, it suffices to execute: ld r,[does_not_matter_what_register + 63]. For example:

  ```
  ld r,[r7+63] // Read 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.

Try it yourself.

- 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 13 on a new line.

Instruction format

The instructions are encoded using 16 bits in the following manner. The '-' character specifies unused bits.

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