Computer Organization and Design (HY219)
Winter Semester 2021-2022

Laboratory 5

1. Implementation of MIPS (format R) of a machine cycle (5 units)

The objective of the workshop is to implement and verify the datapath unit and the control unit of the MIPS processor **of a machine cycle,** using the Verilog hardware description language, and the library
of ready-made units, part of which you implemented in the previous workshop. In the first part of the lab you will implement some of the MIPS commands that follow the R format, specifically the **add, sub, or, and** and **slt** commands . The theory for this workshop is covered in Chapters 4.3-4.4 of the book.

Figure 1 shows the architecture diagram that you will implement in the first phase of this lab, using the two ALU and Register units.
File of the previous workshop. PC is the 32-bit Program Counter and indicates the location in Instruction Memory from which the next instruction should be read. THE
PC should be properly reset as we explained in class at the value 32'h0 to be able to start the program. In addition, you are given the

implementation of a 1 KB memory which can be used to store 32-bit MIPS instructions. In the following paragraphs we describe the structure and function of the memory. The control unit drives each input signal of the data

unit

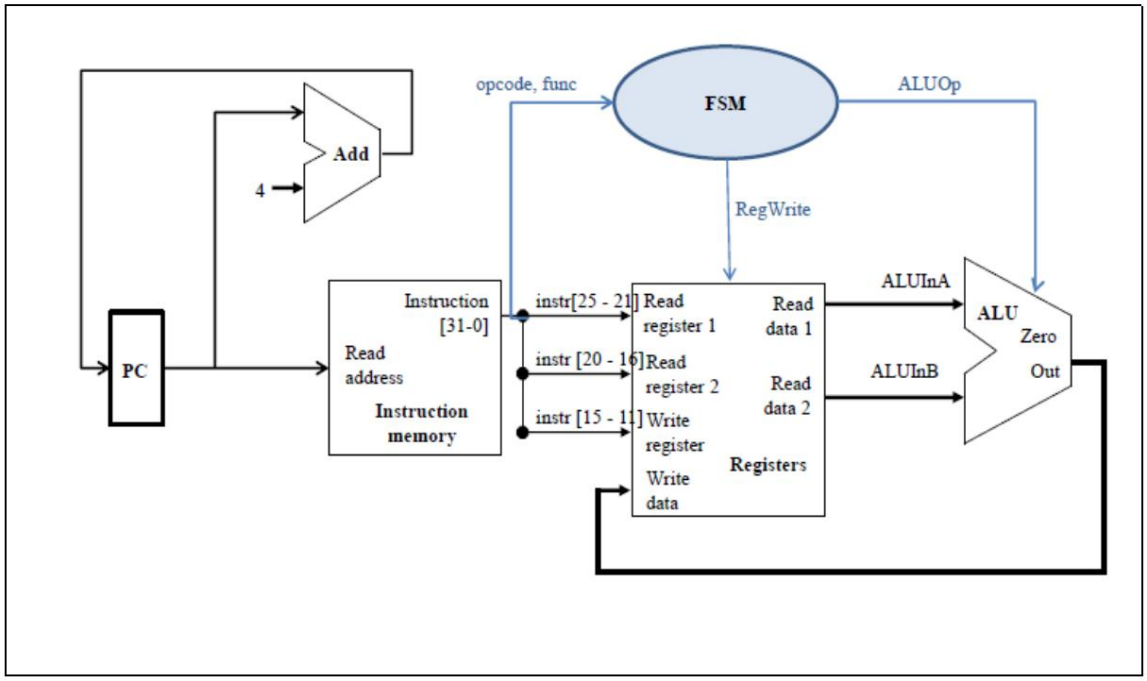

Figure 1. One-cycle MIPS architecture for R-format instructions.

@0 0110_4020 // HEX code for add $t0, $t0, $s0 @1 0124_4822 // HEX code for sub $t1, $t1, $a0 @2 ….

path), and vice versa, each output signal of the data segment drives the control segment. Thus, by combining the two sections, the complete implementation of the processor is obtained. The control part is relatively simple in this lab since it needs to generate very few control signals, taking as input the bits of the 32-bit opcode *and* func *instruction* needed to decode it. It would be better to put the control module in a different file than the

data section to facilitate implementation later. A good idea would be, for example, to have a *cpu.v* file at a higher level of abstraction that will place the data part module and the control part module and connect them.

Memory unit - Memory

module Memory (clock, reset, ren, wen, addr, din, dout);
Asynchronous instruction/data memory.

Ports:

clock: Input. Synchronization clock. All register writes should occur on the negative edge of the clock, just like the register file. The clock is generated in the testbench.

reset: Login. Asynchronous reset. Active low. Only when the reset is 1 does the memory work normally. The reset signal is not used to reset the memory elements. The reset is created in the testbench.

ren (Read Enable): Input (1 bit): read enable.

wen (Write Enable): Input (1 bit): write enable.

addr: Input (32 bits): access address. The address is in bytes. Since we only have words in our implementation, the two lowest bits of the address should be zero. For practical reasons, during simulation, only the first 1024 words of memory are implemented. Therefore, the 20 highest bits of the address are not used and are don't cares.

din (Data Input): Input (32 bits): data input to be written. **dout (Data Output):** Output (32 bits): read data output.

Memory Initialization

Memory can be initialized with its **$readmemh** or **$readmemb** commands Verilog which reads data from a file and initializes arrays with that data. Example of using **$readmemh** in the test context inside some **initial** structure : in the testbench is the

$readmemh("program.hex", cpu0.cpu_IMem.data);

Where cpu0 is the display name of the processor's cpu unit, cpu_IMem the name of the processor's memory display, and data is the name of the memory table in the library's memory unit. The associated memory file, program.hex
is a *text file* and must be in the same directory as the simulation run and have the following format:

```
@0 0110_4020 // HEX code for add $t0, $t0, $s0 @1 0124_4822 //
HEX code for sub $t1, $t1, $a0 @2 …
```

The first column, @X, corresponds to the memory address X, and the second column to the data that will be stored during initialization at address X. Note that the memory address must always be expressed in hexadecimal. The underscore is a Verilog delimiter character, to make the data easier to read. If the **$readmemb command is used,** the data must be in binary. For example:

```
@0 0000_0001_0001_0000_0100_0000_0010_0000 @1 …
```

Note: You can develop your code in MARS, assemble and use the *"Dump Memory to File"* icon to save the code to a text file in HEX or BIN format. This will help you to generate the *program.hex*
file automatically.

Simulation to Verify Correct Operation Along with the
structural description of the data section of the processor, a control box should be implemented, which will run a small program stored in memory, which will verify the correct execution mode for the different types of instructions, i.e. type R.

To verify the data part of the processor you provide a testbench skeleton template, which you should modify (testbench.v *file).* This template file contains instructions for displaying the processor unit, initializing the register, setting the clock, and applying the control signals per cycle. The initialization of the register file will be done by directly assigning the value to each register, while the initialization of the command memory will be done using the **$readmemh command.**

Verifying your circuit should be done by checking the register values on each machine cycle. According to what we have said, each instruction lasts exactly one machine cycle, and the only unit that can be changed by the program execution is the register file.

2. Implementation of MIPS (LW, SW, Branches) of a machine cycle (5 credits)

In the second part of the lab you will complete the MIPS implementation of a machine cycle, incorporating the **lw, sw, addi, beq,** and **bne** commands that follow format I. The theory for this lab is covered in Chapters 4.3-4.4 of the book .

Figure 2 shows the diagram of the architecture you should implement, enriching the datapath unit and the control unit of the previous lab's architecture. Particular attention should be paid to the correct connection of the subunits. For example

the data memory will be used by the **lw** and **sw commands,** the **ADD** unit to find the new address of the PC after a branch command, as well as the large number of multiplexers as shown in shape.

Simulation to Verify Correct Operation Along with the structural

description of the data section of the processor, a control box should be implemented, which will run a small program stored in memory, which will verify the correct execution mode for the different types of instructions. The proposed schedule is shown below. You can use a different program for verification, as long as you verify all types of orders.

label: add $t1, $t0, $a0 sw $t1, 5($t3)
lw $s2, 5($t3), slt
$t1, $t0, $s3 addi $s0,
$s0, 1 beq $t1 , $at, label

If you have the desire and time, you could also implement additional commands that follow the I-format, such as subi , **lb, sb,** kok commands. To verify the data part of the processor you provide a testbench skeleton template, which you should modify (file *testbench.v).* This template file contains instructions for displaying the processor unit, the

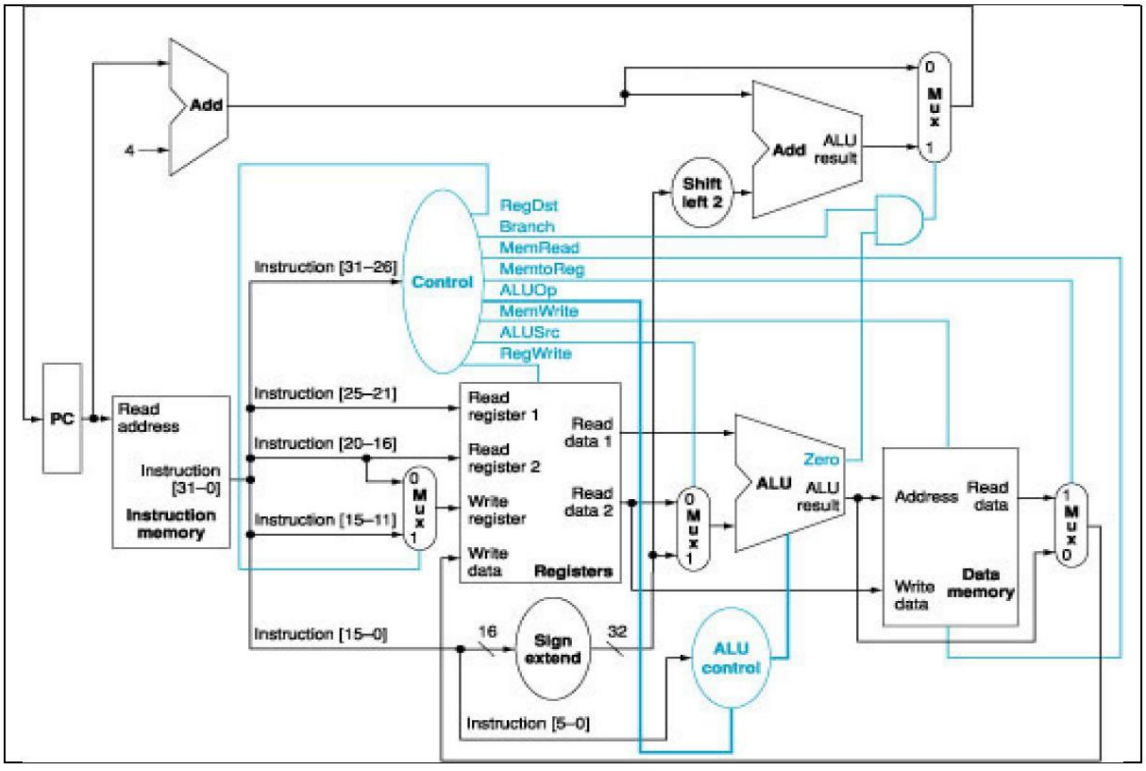

Figure 2. One-cycle MIPS architecture for R-format, load/store and branches.

initializing the register, setting the clock and applying the control signals per cycle. The initialization of the register file will be done by directly assigning the value to each register, while the initialization of the command memory will be done using the **$readmemh command.**

Verifying your circuit should be done by checking the register values on each machine cycle. According to what we have said, each instruction takes exactly one machine cycle.