# CEG 3156: Computer Systems Design (Winter 2021)
# Prof. Rami Abielmona
# Laboratory #2: Single-Cycle RISC Processor

February 18, 2021

## 1 Objective

The objective of this laboratory is to design and build a single-cycle RISC-type processor in VHDL.

Upon completion, the student must be able to:

- Design, realize and test a single-cycle RISC processor;
- Demonstrate a complete understanding for RISC processors and instruction set architectures.

## 2 Pre-Lab

Modify the datapath shown for the single-cycle processor in the text book and included in this lab (figure 9), to support a new instruction: the branch if not equal (bne) instruction. Show the modifications of your datapath, as well as any control signal changes that need to be done to support the new instruction. Present and explain your work to the TA.

## 3 Introduction to Single-Cycle RISC Processors

In today's world, we have a small selection of design styles as applied to processors. We have studied, in class, one specific design style, called the **Reduced Instruction Set Computer (RISC)**. One specific example that was studied in detail is the MIPS processor, which was developed in the 1980s. The MIPS instruction set is actually used by NEC, Nintendo, Silicon Graphics, Sony and several other computer manufacturers.

We will be looking at an implementation of the MIPS processor, which includes the following functionalities:

- Memory-reference instructions (lw and sw)
- Arithmetic logic instructions (add, sub, and, or and slt)
- Control flow instructions (beq and j)

Our implementation will not include all integer operations supported by the MIPS ISA (e.g. multiply and divide), nor will it support floating-point operations, also supported by the MIPS processor. However, the design will be modular and expandable in order to support additional operations if the need arises.

The instructions that are supported, perform two generic steps: use the *program counter (PC)* to supply the instruction address and read one (in case of a lw instruction) or two (in all other cases) registers. These will guarantee that both the instruction and the required operands are fetched into the processor and are ready for execution. The *arithmetic logic unit (ALU)* is then utilized to perform some calculations. In the case of a memory-reference instruction, the ALU is used to calculate the address, in the data memory, of the word we are loading from or storing into a register. In the case of an arithmetic or logic instruction, the ALU is used to perform the actual operation. Finally, in the case of a control flow instruction, the ALU is used to perform a comparison, the result of which is used in the control flow decision process.

After the ALU access, each instruction differs in terms of the path taken for completion of the instruction execution. In the case of a memory-reference instruction, a memory access is done to write data (sw) or read data (lw). In the case of an arithmetic or logic instruction, a write back is performed into a register in order to save the result of the ALU operation. Finally, in the case of a control flow instruction, the next instruction address may need to be changed depending on the result of the comparison.

Let us now take a look at a high-level view of our processor (refer to figure 1). It is interesting to note that the datapath is shown horizontally, and no control logic is shown yet. Also note that different memories for instruction and data are utilized. The former is used to store the read-only accessible instructions of the program to be executed, while the latter is used to store the read-write accessible data utilized throughout the execution of the program.

## 3.1 MIPS Instruction Formats

The MIPS instruction set defines three main types of instructions: R-type, I-type and J-type instructions. The R-type instructions are mainly arithmetic and logical in nature (e.g. add, sub, and), with a typical instruction being `add $t1, $t2, $t3`, which directs the processor to add the contents of registers t1 and t2, and store the result in register t3.

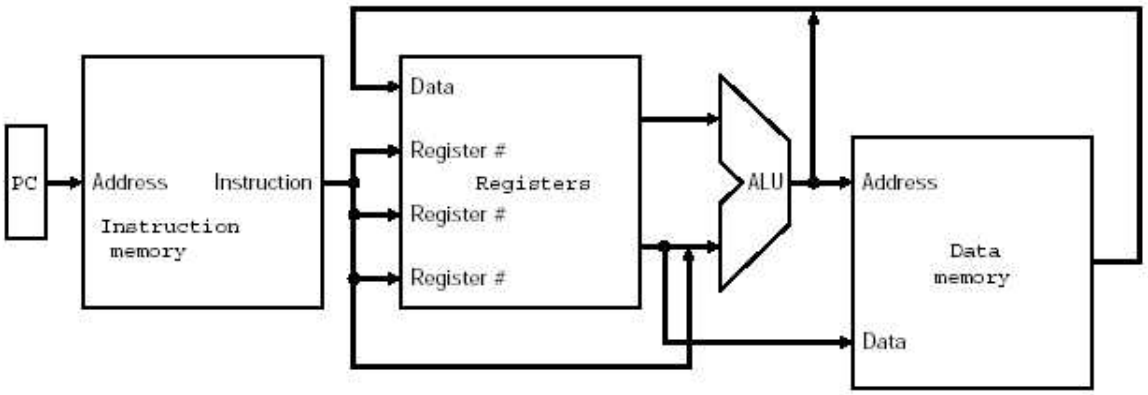



Figure 1: Typical MIPS processor datapath

The I-type instructions are mainly memory-reference in nature (e.g. load and store word), with a typical instruction being `lw $t1, offset_value($t2)`, which directs the processor to add the offset_value to the contents of register t2. The result will form the address, in the data memory, of the word that has to be accessed. In the case of a store, the contents of the register t1 are transferred to that word in memory, while in the case of a load, the contents of that word in memory are transferred to the register t1.

Finally, the J-type instructions are mainly control-flow in nature (e.g. branch and jump), with a typical instruction being `beq $t1, $t2, 25`, which directs the processor to compare the contents of registers t1 and t2, and if equal, perform a branch to a location in memory 25 instructions below the current PC. Note that a branch instruction can cause a jump above or below the current PC, while a jump instruction can only cause a branch below the current PC. A typical jump instruction is `j 2500`, which directs the processor to jump to the address 2500 (in word addressing).

Take a look at figures 2 and 3, for the internal representation of all of our instructions. Note the different opcodes for each type of instruction (0 for ALU operations, 35 for a load word operation, 43 for a store word operation, 4 for a branch if equal to operation and 2 for a jump operation).

Note that *rs* and *rt* are source registers in all supported instructions , while *rd* is a destination register, only used when an ALU instruction needs to write back the result into the register file. The *funct* field in the R-type instructions defines the ALU function to be performed as follows:

- Addition operation performed when the function field is 32
- Subtraction operation performed when the function field is 34

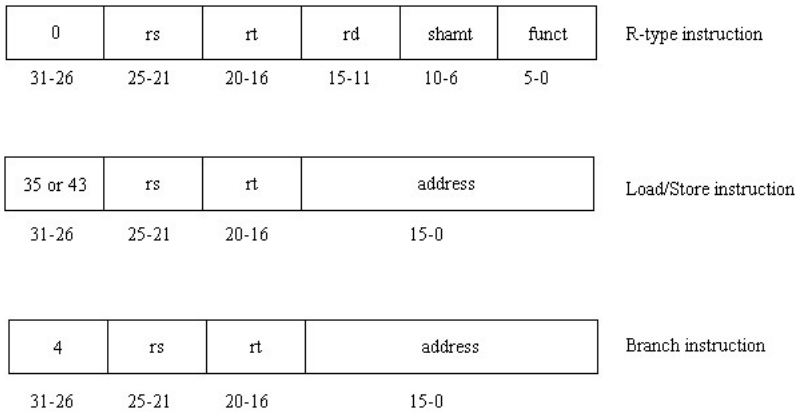



Figure 2: MIPS instruction format

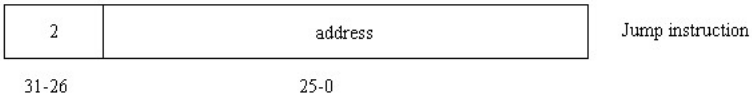



Figure 3: MIPS jump instruction format

- And operation performed when the function field is 36
- Or operation performed when the function field is 37
- Set if less than operation performed when the function field is 42

The *shamt* field in the R-type instructions is used for shifting operations which are not currently supported by our processor, and hence the field can be ignored.

The load and store instructions utilize *rs* as the source register, and *rt* as the source or destination register depending on whether the operation to be performed is a store or load, respectively. The branch instruction utilizes *rs* and *rt* as source registers, and the 16-bit address field as the offset value in the operation. Finally, the jump instruction utilizes the 26-bit address field in the calculation of the next instruction address.

## 3.2 Fetch and Increment

The first order of business is to ensure that the PC is fetched appropriately, and incremented by 4, as our instruction memory is 32-bits wide. This is performed simultaneously as shown in figure 4. Note the use of an instruction memory to store the instructions of our program, as well as an adder in order to increment

the PC by 4. The PC is a register that holds the address of the next instruction to be executed, and is loadable on every clock cycle, hence the write control signal is omitted from the figure, and the PC is understood to be updated at every clock cycle.

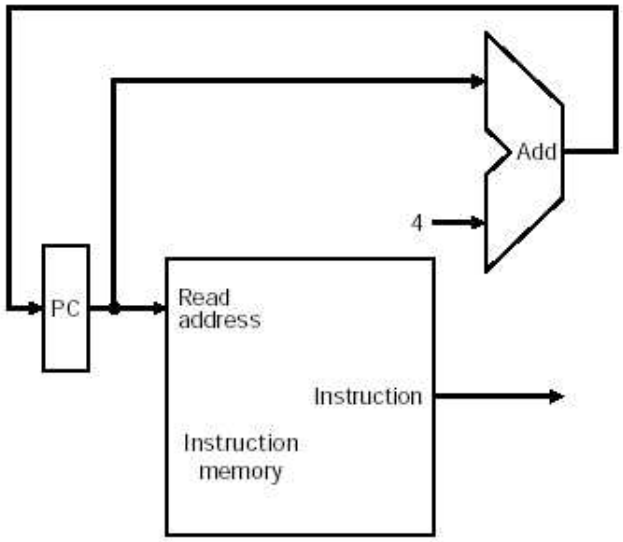



Figure 4: Fetch and increment datapath

### 3.3 R-type Instruction Datapath

Let us now take a look at our R-type instruction datapath. It is shown in figure 5, and utilizes two basic building blocks: a register file and an ALU. All data signals are 32-bits wide, and we have thirty-two 32-bit registers, hence a 5-bit register addressing is required for the register file. The latter contains two read ports and one write port, allowing three simultaneous accesses (two reads and one write) to the register file. The ALU, on the other hand, is very similar to the one designed in class, and consists of a 32-bit MIPS ALU, capable of addition, subtraction, AND/OR logical operations and a set less than (comparison) operation. It provides a *Zero* status signal indicating whether the ALU result is equal to zero (Zero = 1) or not (Zero = 0). Note that, as aforementioned, the ALU result has to be written back into the register file for later use. Finally, note that a control signal (*RegWrite*) is needed to control the writing mechanism of the register file.

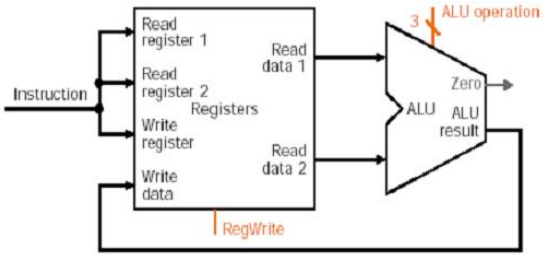



Figure 5: R-type instruction datapath

### 3.4 I-type Instruction Datapath

Let us now take a look at our I-type instruction datapath. It is shown in figure 6, and utilizes four basic building blocks: a register file, an ALU, a data memory and a sign-extension unit. All data signals are 32-bits wide, except for the 16-bit address field. The register file is used to decode the register's contents and add those to the sign-extended (to 32-bits) offset value. The latter is done to maintain the sign of the offset value (this is called **PC-relative addressing**). Once the address is calculated as the output of the ALU, it is sent to the data memory unit, which contains one read and one write port, allowing for a simultaneous read and write from and into the memory. In the case of a store instruction, the contents of the first operand (in the instruction) are latched onto the *Write Data* bus, while the ALU result is latched onto the *Address* bus of the data memory. The resulting action is the storage of the first operand's contents in the data memory. In the case of load instruction, the contents of the data memory word pointed to by the *Address* bus are latched onto the *Read Data* bus and written back (on the next clock cycle) into the register file, in the register indicated by the first operand of the instruction. The resulting action is the loading into the first operand, the contents of the data memory word. Note that the data memory is controlled by two different signals, one to allow it to read, and the other to allow it to write.

### 3.5 J-type Instruction Datapath

Let us now take a look at our J-type instruction datapath. It is shown in figure 7, and utilizes three basic building blocks: a register file, an ALU and a sign-extension unit. All data signals are 32-bits wide, except for the 16-bit address offset value. The register file outputs the contents of the two operands, in the case of a branch instruction, whilst the ALU proceeds to compare the two numbers. If the latter are found to be equal, then the *Zero* status flag is asserted. The main control unit then decides whether to take the branch or not. If the branch is to be taken, the branch target is written to the PC. The branch

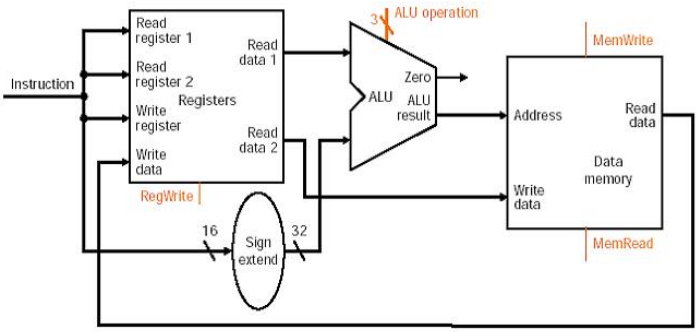



Figure 6: I-type instruction datapath

target consists a base address (PC + 4) and the sign-extended offset, shifted by 2 to the left, in order to make the jump a word offset. In the case of a jump instruction, shown later on, we calculate the new address simply through a shift left by 2 of the address field in the instruction.

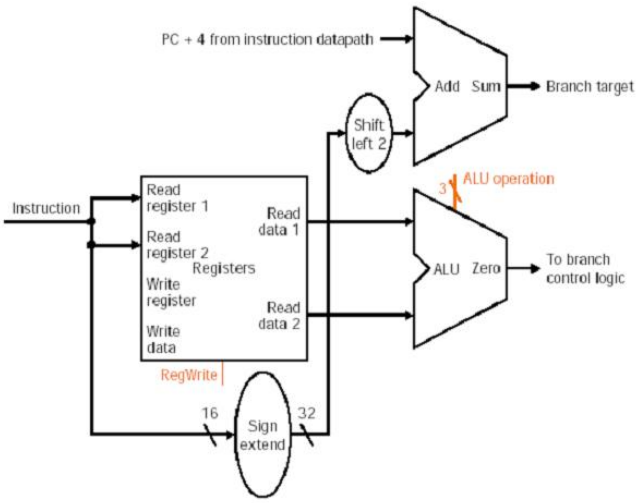



Figure 7: J-type instruction datapath

# 4 The Completed Datapath

Let us now group the datapaths that we've build in the previous section, and add all associated multiplexers and control signals, as well as the ALU control

unit and the main control unit, to construct our single-cycle RISC processor. The resultant is shown in figure 9.

### 4.1 Notes About Single-Cycle Design

There are a few issues worth noting about our single-cycle processor. They are:

- All instructions must begin and conclude in one clock cycle;
- All values must reach a stable state;
- The cycle time is determined by the longest path delay;
- The control logic is not systematically defined.

That said, there are some inefficiencies involved with the design of a single-cycle processor, which include the difficulty in supporting more complex (e.g. floating-point) instructions, as well as a lower clock frequency. These could be alleviated through different solutions, one of which is discussed below.

## 5 Laboratory

In this lab, you will proceed to implement a single-cycle processor as shown in figure 9. The input/output specification of the processor is shown in table 1. We will be working with **8-bit datapaths** and **32-bit instruction widths**, while the control lines obviously remains constant. Also note that the 8-bit datapaths, will only allow for 8-bit number representations and operations. We will be implementing a register file of **eight 8-bit registers**. Instruction memory will remain at 32-bits, and will contain a maximum of 256 instructions. The data memory, however, will be 8-bits, and will contain a maximum of 256 words.

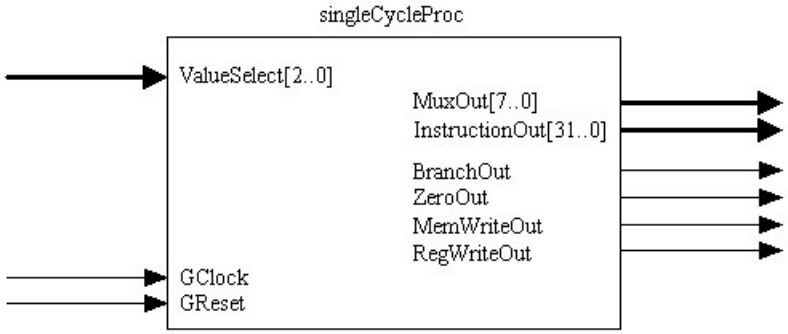



Figure 8: Single-cycle processor entity

The MuxOut[7..0] and ValueSelect[2..0] output and input are intertwined in the following manner:

| *Port Type* | *Name* | *Description* |
|---|---|---|
| **Input** | GClock | Global clock needed to synchronize the circuitry |
| **Input** | GReset | Global reset needed to bring the internals to known states |
| **Input** | ValueSelect[2..0] | Selector for MuxOut[7..0] |
| **Output** | MuxOut[7..0] | Multiplexer output controlled by ValueSelect[2..0] |
| **Output** | InstructionOut[31..0] | The current instruction being executed |
| **Output** | BranchOut | The branch control signal |
| **Output** | ZeroOut | The zero status signal |
| **Output** | MemWriteOut | The memory write control signal |
| **Output** | RegWriteOut | The register write control signal |

Table 1: Input/Output Specification

| *ValueSelect[2..0]* | *MuxOut[7..0]* | *Description* |
|---|---|---|
| **000** | PC[7..0] | The program counter value |
| **001** | ALUResult[7..0] | The result of the current ALU operation |
| **010** | ReadData1[7..0] | The read data 1 port of the register file |
| **011** | ReadData2[7..0] | The read data 2 port of the register file |
| **100** | WriteData[7..0] | The write data port of the register file |
| **Other** | ['0', RegDst, Jump, MemRead, MemtoReg, AluOp[1..0], AluSrc] | The remaining control information |

Table 2: Output Multiplexer Selection

## 5.1 Instruction and Data Memory

Since we are operating with separate instruction and data memories, you will be allowed to use two Altera-supplied LPM cores. The instruction memory can be an instantiation of the LPM_ROM function (256x32), while the data memory can be an instantiation of the LPM_RAM_DQ function (256x8). That said, if the group chooses to implement their own RAM and ROM modules, they are welcome to do so.

A sample instruction memory initialization file (MIF) is given below:

```
DEPTH = 256;
WIDTH = 32;

ADDRESS_RADIX = HEX;
DATA_RADIX = BIN;

CONTENT
BEGIN
```

```
-- Use no operations (nop) for default instructions
[00..FF]:   00000000; -- nop(add $t1, $t1, $t1)

-- Place MIPS instruction here
00: 8C020000;   --lw $2,0 memory(00)=55
04: 8C030001;   --lw $3,1 memory(01)=AA
08: 00430820;   --add $1,$2,$3
0C: AC010003;   --sw $1,3 memory(03)=FF
10: 1022FFFF;   --beq $1,$2,-4
14: 1021FFFA;   --beq $1,$1,-24

END;
```

A sample data memory initialization file (MIF) is given below:

```
DEPTH = 256;
WIDTH = 8;

ADDRESS_RADIX = HEX;
DATA_RADIX = HEX;

CONTENT
BEGIN

-- Default Memory Values
[00..FF]: 00;

-- Initial Values
00:   55;
01:   AA;

END;
```

## 5.2 Verification: Running a Benchmark Program

Once completed, the design has to be tested and verified. The testing process can be accomplished throughout the design, with unit testing, then module testing, and finally system testing. As soon as the processor seems to be behaving according to specifications, a verification step is to be performed by running the following program, stored in instruction memory (remember to initialize the data memory as shown in the comments):

```
lw $2, 0;          $t2 = memory(00) = 55
lw $3, 1;          $t3 = memory(01) = AA
sub $1, $2, $3;    $t1 = $t2 - $t3 = 55
```

```
or $4, $1, $3;    $t4 = $t1 or $t3 = FF
sw $4, 3;         memory(03) = $t4 = FF
add $1, $2, $3;   $t1 = $t2 + $t3 = FF
sw $1, 4;         memory(04) = $t1 = FF
lw $2, 3;         $t2 = memory(03) = FF
lw $3, 4;         $t3 = memory(04) = FF
j 11;             jump to address 44
beq $1, $1, -44;  loop back to beginning of program
beq $1, $2, -8;   test if $t1 = $t2 ?
```

This program will have to be demonstrated as running to your TA as proof of success for your design. As well, the calculation of the maximum clock frequency has to be shown and explained in your report.

Finally, in order to measure the efficiency of our single-cycle processor, we will need to calculate its CPU execution time as shown in the example on page 373 of our course book. Using that example, and the benchmark program above, calculate the CPU execution time of our processor. Assume, as in the example, that memory units have a 2 ns operation time, while the register file has a 1 ns operation time. **However**, calculate the worst path delay of your ALU and adders, assuming that a gate delay is 0.01 ns. Show all your work and reasoning.

### 5.3 Bonus: 32-bit Single-Cycle Processor Design

For an **extra 5 marks**, modify your processor design to make it a completely 32-bit data and 32-bit instruction single-cycle processor, and include the *branch if not equal to (bne)* instruction to your design (opcode = 5). Do not show all the components again, but instead, discuss the modifications performed on the components in order to run the 32-bit processor. If the datapath and control logic have changed, present and discuss the modifications. You have to demonstrate your new design to your TA.

### 5.4 Bonus: Multicycle Processor Design

For an **extra 15 marks**, implement the multicycle processor shown in class and in the textbook. Again, do not show all the components again, but instead, discuss the modifications performed on the components in order to realize the multicycle implementation. If new components were added, it goes without saying that their design and realization has to be discussed. After completing the design, run the benchmark program and compare the CPU execution time with the single-cycle processor implementation. You have to demonstrate your new design to your TA.

## 6 Design Restrictions

- Verilog implementations will not be accepted. Perform all implementations in VHDL code only

- Behavioral level of modeling will not be accepted. Design should be done at the structural level of modeling
- Register Transfer Logic (RTL) design and coding will is mandatory
- Use graphical design for the top-level entity, and use your judgement for any other sub-blocks. However, all atomic modules have to be implemented in VHDL (i.e. D flip-flop, 1-bit adder, 1-bit comparator and so on)
- No core instantiations are allowed (i.e. LPMs from Altera or free IP cores from the Internet), except for the ones mentioned in the lab instructions (LPM_ROM and LPM_RAM_DQ). All remaining building blocks have to be designed and realized by the group
- The top level entity is given in input/output specification format, but the internals are left up to the group. A sample schematic solution for the single-cycle processor has been given, but the group is free to design the system using another methodology
- The design has to be synchronous and globally reset-able. This means that global clock and reset signals are required in all functional blocks (the single-cycle and multicycle processors)
- Simulate all designs and check your simulation results with your theoretical ones (e.g. run the benchmark program)
- Download the design to the Cyclone chip on your DE-2 boards, and use the on-board DIP switches to input ValueSelect[2..0], and the sixteen LEDs to demonstrate correct output functionality (MuxOutput[7..0] and the control signals shown in figure 8)
- Only show the current instruction (InstructionOut[31..0]) in simulation demonstration. You do not have to show it during the live demonstration
- Each group must demonstrate a working version of the laboratory to the TA, before the due time of the report

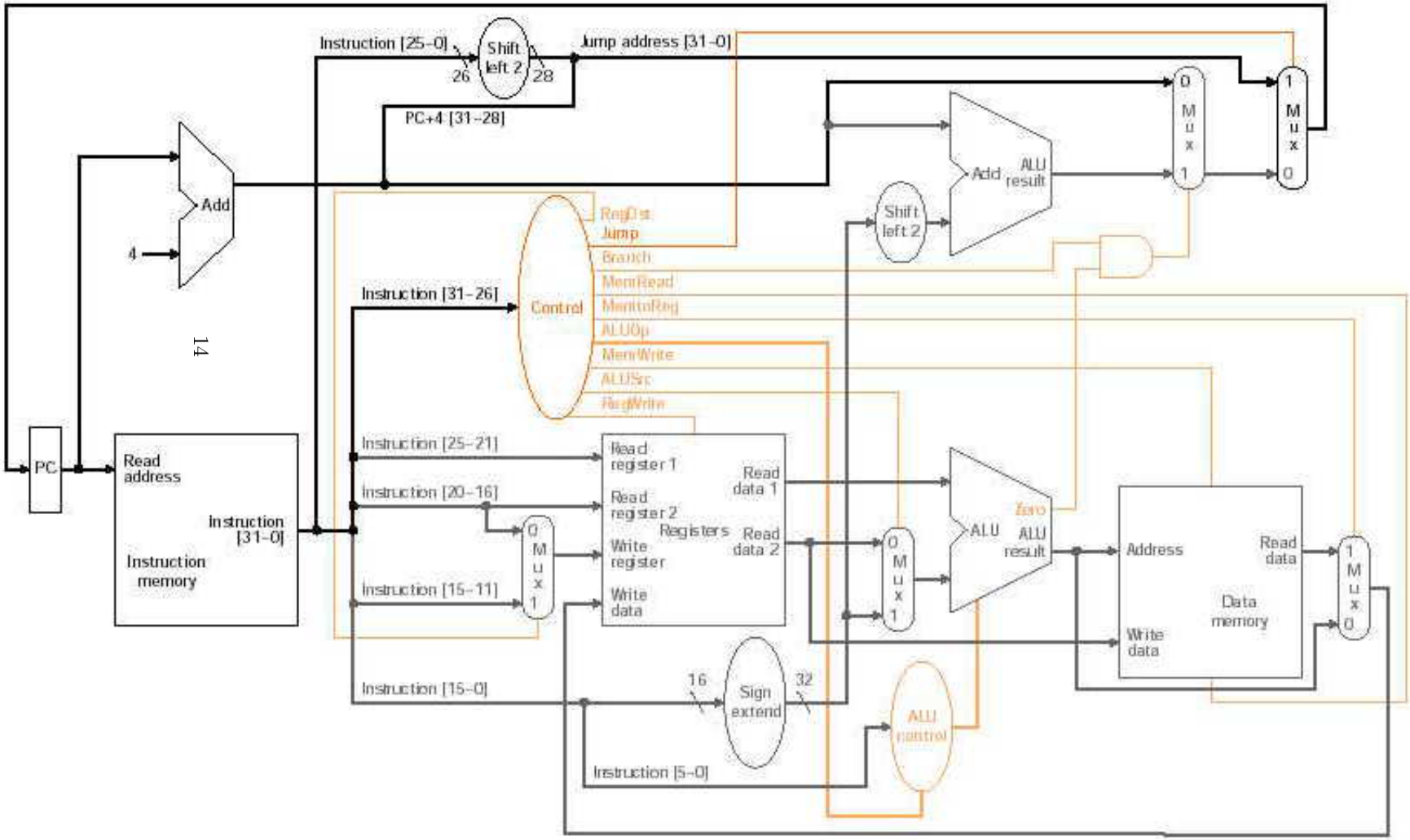
# 7 Report Reminders

- Include timing simulations with explanation for all VHDL source files
- Describe and comment all your VHDL source files
- Include a flowchart representation of your solution to the problem
- Include a block diagram of your solution to the problem
- If using ASM design, include all appropriate charts and paths (control and data)

- If using FSM design, include the state diagram with all appropriate inputs and outputs
- Describe, in your own words, your solution to the problem
- Describe your design obstacles and how they were overcome
- Append all VHDL source code and graphical design files to your report
- Submit a soft copy of all VHDL and graphical design files with your report

# 8 Acknowledgements

All figures (except 2, 3 and 8) are taken out of Chapter 4 of our course textbook. For more detailed explanations and examples, please refer to *Computer Organization and Design, The Hardware/Software Interface* by David A. Patterson and John L. Hennessy.


Instruction [25–0]
Shift left 2
26
28
Jump address [31–0]
PC+4 [31–28]
Add
4
M u x
0
1
Add
ALU result
Shift left 2
RegDst
Jump
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Control
Instruction [31–26]
PC
Read address
Instruction [31–0]
Instruction memory
Instruction [25–21]
Instruction [20–16]
Instruction [15–11]
Instruction [15–0]
Instruction [5–0]
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
16
Sign extend
32
ALU
Zero
ALU result
ALU control
Address
Read data
Data memory
Write data