Princess Sumaya University for Technology

King Abdullah II Faculty of Engineering

Computer Engineering Department

COMPUTER ARCHITECTURE II
VERILOG PROJECT
GROUP NO.4

Authors:

Mohammed Waleed Abdulrahman 20160073

Osama 20160508

Supervisor:

Dr. A. Al Mousa

December 24, 2019

Abstract

The basic MIPS implementation does not include the hazard detection, forwarding unit, floating point and many other complex instructions. This project aims to overcome all these obstacles by upgrading the data flow, using more modules and modifying others. Thus, implementing previously mentioned modifications, the processor's throughput becomes more optimized as expected, however; implemintng the modules became much more complex and the processor became larger in size. Even though these disadvantages appear to be more of a drawback, the resultant advantages overweigh them.

TABLE OF CONTENTS

1 INTRODUCTION

This project aims to build MIPS pipelined processor using Verilog Hardware Description Language (VHDL) that performs a set of instructions. The designed processor is able to handle a several architecture issues such as hazard detection and forwarding. All modules inside the processor was built depending on basic MIPS architecture and then it was developed to support more instructions like floating point instructions.

1.1 OBJECTIVES

- To fully understand the data flow through the MIPS processor.
- To implement the processor using the VHDL.
- To test and understand different forwarding cases.

2 RESULTS AND DISCUSSIONS

The designed processor is composed of five main stages: Fetching, Decoding, Executing, Memory and Write Back. Each stage contains modules. In order to process instructions, the instructions will flow through the five stages.

The modules represent the hardware of the processor. Some of the instructions require an additional hardware (added MUXes) over the basic MIPS architecture (Shown in Figure 1) and an updated versions over the main modules (added inputs and outputs over the module). This part of the report discusses each stage and how the data flows through the modules in each stage:

1. **Instruction Fetching Stage**: This stage represents the first step in the instructions process. It contains the *Program Counter* (*PC*) module, *Instruction Memory*, pc_source_mux and pc_jump_mux.

   The *Program Counter* (*PC*) contains the address (location) of the instruction that will be executed. It is synchronized with the clock of the processor. The address will be passed to the *Instruction Memory* at the negative edge of the clock. Since the MIPS-architecture has a byte address memory and the MIPS-instruction is a 32-bit, therefore each instruction will take four locations in the Instruction Memory. After passing the current address, the program counter increases its values by four (using the 32-bit adder module), so the next address will be ready to pass at the next cycle, unless the Control Signal 'Flush' becomes Logic-High which means the *Control* Unit has detected a data hazard.

   The *Instruction Memory* takes the address of the fetched instruction from the *Program Counter* (PC) and fetches the corresponding instruction to the next stage. The stored bytes in the memory is in the Little-Endian format (The least significant value is stored first).

   There are two MUXs lies in this stage 'pc_source_mux' and 'pc_jump_mux'. The first MUX determines which address (PC+4 or the target address) will pass to the second MUX. The chosen

address will be determined by a select line derived from the *Control Unit* named 'Jmp'. The second MUX determines which address will be the source for the PC. The input values for the second MUX will be 'jump register address', 'branch address' and the passed address. The selected value will be determined using the select lines 'If_pcAddress' and 'ALUPcSrc'.

The 'If_pcAddress' (Least significant select line) is the OR combination between 'beq' and 'bne', means it will be Logic-High only in these two cases. The 'ALUPcSrc' (Most significant select line) will be Logic-High in the jump register instruction only. Snice the jump register instruction is an R-type, the *Control Unit* will not be able to distinguish this instruction, while the ALU control will distinguish the jump register depending on the value of the function.

When the 'Flush' signal is Logic-Low, the IF/ID register is responsible to pass the (PC+4) and the fetched instruction from the *Instruction Memory* at the positive edge clock, on the other hand when 'Flush' signal is Logic-High it will not update its values.

Figure 2 shows the Fetching stage and the mentioned modules.

2. **Instruction Decoding Stage:** This stage represents understanding for the processor. During this stage, the *Control Unit* distinguishes the type of the instruction and sends the proper signals to the connected modules. This stage contains the *Control Unit*, the *Register File*, the *Floating Register File*, *Zero* and *Sign Extensions* and five MUXs.

   The *Control Unit* represents the brain of the processor, it recognizes the instruction type (R-type, I-type or J-type) and organizes the behavior of the modules through the whole pipeline. Since the *Control Unit* signals will be changed every cycle, the controlling signals will be passed to the ID/Ex register. So, each connected module with the *Control Unit* will receive its signals safely and it acts like a hazard detection unit, its functionality is to recognize the memory load data hazard which needs to stall one cycle prior to decoding the next instruction which means making the flush control signal Logic-High and forcing all the other control signals to become Logic-Low.

   The *Register File* receives the numbers of registers that will be used through the current instruction during this stage. The *Register File* contains 32x32-bit registers. It will pass registers that correspond to the received numbers. The basic MIPS architecture for the register file has only two output registers 'read data1' and 'read data 2'. They represent the source and target registers (rs, rt) respectively, however the register file in the designed processor has a third output register 'read data 3' represents the destination register (rd). This output register were added to the module to perform the instructions *Store Word New* (swn) and *Load Word New* (lwn). The two 32-bit registers *High* and *Low* lies in the in the *Register File*. The input signal '*float*' indicates the register file that the *High* and *Low* registers (one or both) will be used by one of the following instructions (*Multiply , Divide , Move From High* or *Move From Low*). The input 'RegWrite' will be Logic-High in all R-type instructions except for one instruction, this exception will be explained in the Write Back Stage.

The *Float Register File* implemented to deal with floating instructions. It contains a 32x32-bit registers namely (f0-f31). The *Floating Register File* works exactly the same as the basic MIPS *Register File*.

The *Zero Extension Unit* were added to perform the 'And Immediate' and 'OR Immediate' instructions. The *Zero Extension* fills the upper 16-bit with zeros while the *Sign Extension Unit* fills the upper 16-bit depending on the sign of the immediate value, (0 for positive and 1 for negative numbers).

There are five MUXs in this stage: rs_float1_mux, rt_float2_mux, zero_sign_extension_mux, write_register_mux and rd_float3_mux. The MUXs 'rs_float1_mux' and 'rt_float2_mux' determine which registers will be passed ('rs and rt' or 'fs and ft') and the *Control Unit* determines the values of the select lines. The zero_sign_extension_mux determines which value will pass to the next stage, the passed value will be selected by the *Control Unit* as well. The write_register_mux were added to determine which value will be written in the *Register File*.

Figure 3 shows the Decoding stage and its modules.

3. **Instruction Executing Stage:** The Executing stage is considered the most overwhelming stage, because it contains two of the main processor issues: *Arithmetic Calculations* and *Forwarding Unit*.

The *Arithmetic Logical Unit Control* (*ALU Control*) receives the Function and the ALU Opcode of the instruction from the ID/Ex register. It provides the *ALU* with the needed operation through the current instruction by the 4-bit wire 'ex_alu_signal'. The operation will be determined depending on the values of ALU Opcode and the Function. Since the *ALU Control* is able to distinguish between the R-type instructions, so it will determine with the *Control Unit* the proper *ALU* input operands. The MUXs paragraph will explain exactly the determination procedure.

The *ALU* operand is designed to perform integer and floating instructions. Instructions such as Multiply (Mult) and Divide (Div) requires 64-bit in result. In order to do so, the output of the *ALU* was extended to be 64-bit. This solution is better in hardware size than declaring a special *ALU* operand to perform the floating instructions. The result for integer instructions will lie in the lower 32-bit of the result. The Load Upper Immediate (lui) instruction is handled inside this module, it takes the immediate value and concatenate it with a 16-bit 0 at the lower part. The *ALU* has an induvial output bit named 'ex_zero'. This bit represents the Zero Flag. The Zero Flag will be Logic-High only if the result of the *ALU* is ZERO. This case will happen with the Branch on Equal Instruction (Beq) according to the subtraction of the source and target registers. For Branch on not Equal instruction (Bne), the Zero Flag will be Logic-Low, so to solve this problem, an AND-gate with a bubble at one of the inputs were added to the pipeline, so the Logic-Low value will be inverted making the 'Bne' bit Logic-High in Branch on not Equal case. The target address of (Beq and Bne) instructions will be calculated by the 32-bit adder module.

The *Forwarding Unit* is a combinational circuit that detects the hazards and takes the appropriate actions in the forwarding operation. The source, target and destination registers in the Executing Stage are inputs for the *Forwarding Unit*. In order to detect a forwarding-operation is needed or not, the target and destination registers of the next stages should be known for the *Forwarding Unit.* It handles the cases: R-type followed by R-type, R-type followed by I-type, I-type followed by R-type and I-type followed by I-type. The forwardA term represents source dependency, so the forward will be at the first operand of the *ALU*. The ForwardB term represents target dependency, so the forward will be at the second operand of the *ALU*. The term 'forwardA' is used in the book therefore, the designed processor came up with a forwardC. The forwardC term is a memory-to-memory forwarding. It is special for an I-type instruction followed by Store. Taking the Load instruction followed by Store is considered a good example to clarify the forwardC.

There are seven MUXs in this stage: rs_jal1_mux, rs_extension_rd_mux, jal2_mux, forwardA_mux, forwardB_mux, reg_destination_mux and forwardC_mux. The rs_jal1_mux determines which value (rs or PC+4) will be passed to the forwardA_mux. The select line will be determined by the *Control Unit.*

The rs_extension_rd_mux passes the selected value to the jal2_mux. The rs_extension_rd_mux has three input values (rt, ex_signed_imd and rd) and two select lines: 'aluSrc0' and 'the XOR-combination of ex_alu_Src and shift'. The aluSrc0 is the most select line and the XOR-result is least one. The reason why the XOR-gate were added is that the least select line will always be Logic-Low except for the Shift instruction. Since the shift instruction is an R-type instruction, the *Control Unit* will not be able to distinguish which instruction the R-type is. However, the *ALU Control* will be able to do so by checking the value of the function. The shift value will be Logic-High only in the shift instruction making the value of the *Control Unit* got inverted. So the immediate value will pass through the forwardB_mux.

The values of forward A, forwardB and forwardC MUXs will be determined by the *Forwarding Unit.* The first two MUXs determines which values will pass to the *ALU* and forwardC value will pass to the next stage.

The reg_destination_mux determines where the final value will be saved. The MUX will pass the target register (rt) in the I-type instructions and it will pass the destination register (rd) in the R-type. However, Load Word New and Store World New instructions are considered as an exceptional cases, since both of them are an R-type and need to save their final values in (rt). The target register will be chosen in the same way as the shift value selected.

Figure 4 shows the Instruction Executing Stage.

4. **Memory Stage:** The Memory stage has a single module, this module is the *Data Memory*.
The *Data Memory* will be used only during the load and store instructions. The *Data Memory* looks like the *Instruction Memory*. It is a byte address and each instruction is a 32-bit hence, each instruction will take four locations in the *Data Memory*. The *Data Memory* receives the

address from the EX/Mem stage register and loads or stores the corresponding location at this stage. The ‘MemWrite’ and ‘MemRead’ will always be Logic-Low except for the Store Word New and The Load Word New. Through these two instructions the signals derived from the *Control Unit* will be inverted by the effect of the XOR-gate, so the MemWrite and MemRead will be Logic-High making the *Data Memory* works correctly.

The input ‘MemDataSize’ is a single bit indicates the *Data Memory* of how much size (Word or Byte) will be loaded or stored through the current instruction. If MemDataSize is Logic-Low the needed size will be a word and if MemDataSize is Logic-High the needed size will be a byte and the upper 24-bit will be zeros.

Figure 5 shows the Memory Stage.

5. **Write Back Stage:** The Write Back stage is last stage through the MIPS-pipeline. Number of the modules in this stage is very few. There is a single MUX in this stage named ‘write_data_mux’. The MUX determines which value will pass (‘the *ALU* result’ or ‘the *Data Memory* outcome’). In the R-type instructions, the passed value will always be *ALU* result except for the Load Word New (lwn).

   The XOR-gate were added to this stage especially for Store Word New (swn) instruction. Since the Store Word New is an R-type, the wb_regWriteFinal wire will always be Logic-High, so in this case after storing the value of the new word in the *Data Memory* the value of the target address will be overwritten by the memory target address. Using the XOR-gate will make the value of ‘wb_regWriteFinal’ will be Logic-Low, so the target register value R[rt] will not change.

   Figure 6 shows the Write Back Stage.

2.1 FIGURES

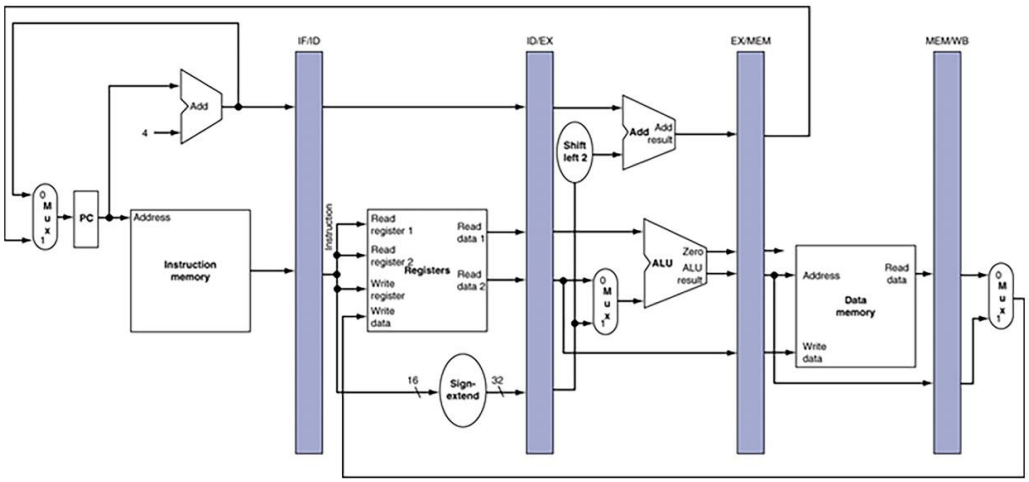

Figure 1: The Basic MIPS Architecture

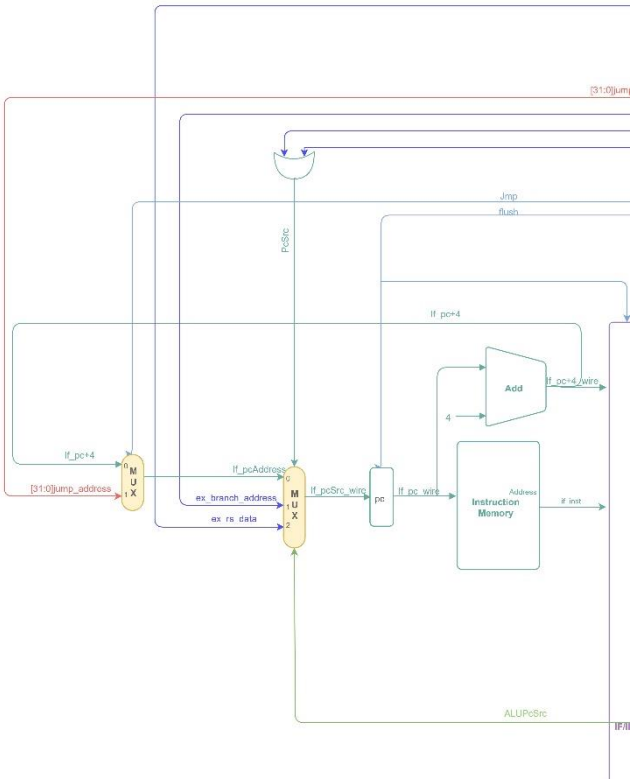

Figure 2: Instruction Fetching Stage

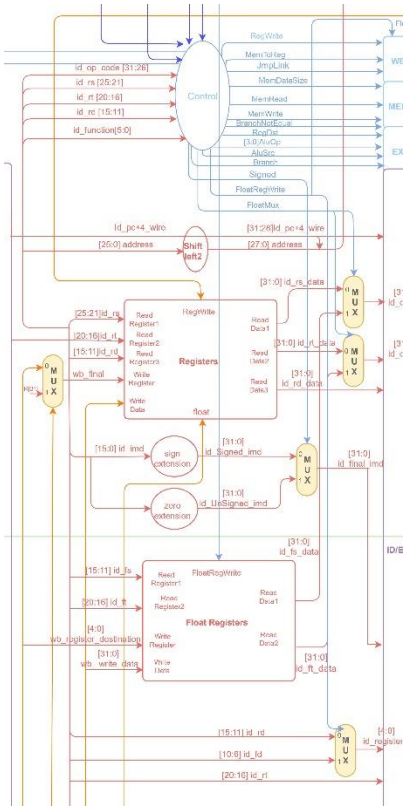

Figure 3: Instruction Decoding Stage

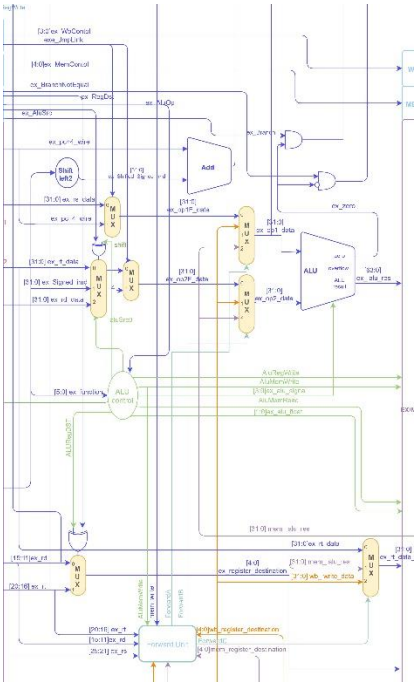

Figure 4: Instruction Executing Stage

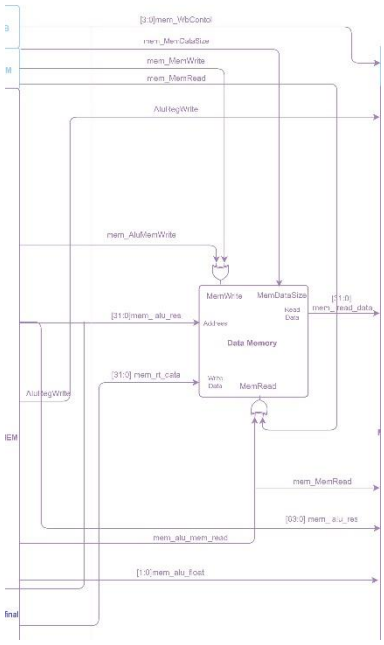

Figure 5: Memory Stage

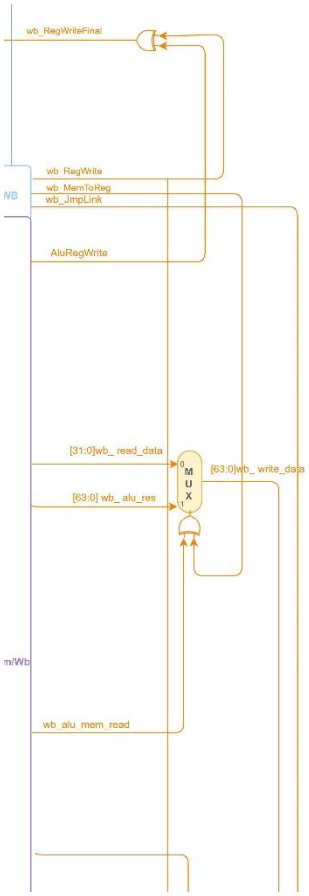

Figure 6: Write Back Stage

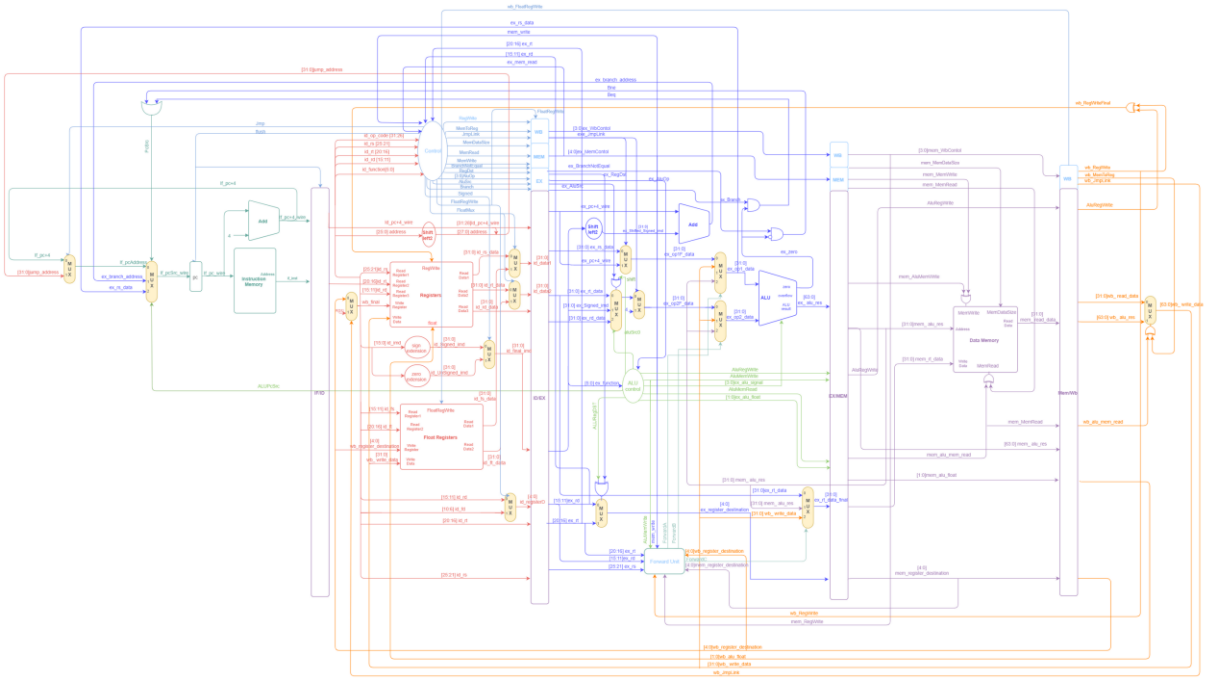

Figure 7: The Full Hardware Design

3 CONCLUSION

The designed processor performs all the given instructions in the custom MIPS sheet and covers most of the data hazard's cases. Building a processor has many ways to be implemented; we preferred to develop our design using our knowledgement of MIPS that was covered in class. The results satisfied the expectations that were put beforehand, such as forwarding operands from *ALU*-to-*ALU*, *Memory*-to-*ALU* and *Memory-to-Memory* and hazard detection. We implemented test benches that support our results.

4 REFERENCES

[1] Computer Organization and Design: The Hardware and Software Interface.