

**Computer Architecture (ENCS4370)**

**Project #2**

**Multi-cycle MIPS RISC Processor in Verilog**

**Prepared by:**

**Christina Saba – 1201255**

**Joud Hijaz – 1200342**

**Pierre Backleh – 1201296**

**Supervised by:**

**Dr. Ayman Hroub**

**January 2024**

# Abstract

# Table of Content

# 1. Design & Implementation

Our method for this project was to create a multi-cycle CPU using the MIPS Reduced Instruction Set Computer (RISC) architecture. The data route is divided into separate phases in the processor's architecture, each of which is completed in a single clock cycle. FETCH, DECODE, EXECUTE, MEMORY, and WRITEBACK are the five essential phases.

## 1.1 Processor Architecture

A section of the MIPS instruction set, which is known for its effectiveness and simplicity, is used in this project. Every instruction follows the same format and has a length of 32 bits. The following categories are included in the instruction set:

1. ***R- Type Instructions:***

These instructions define the operation and register operands in a specified syntax and involve register operations.

1. ***J- Type Instructions:***

Jump instructions are used to change the program's control flow. They consist of calls to functions and unconditional jumps.

1. ***I- Type Instructions:***

Instructions for tasks like loading, saving, and branching that need immediate values.

1. ***S- Type Instructions:***

Guidelines for adding and removing values from the stack that are relevant to stacks.

## 1.2 Multi-cycle

## 

An instruction is executed in several separate stages, each taking one clock cycle to finish, in a multi-cycle processor design. These stages make it easier to execute instructions in a more organized and modular method.

1. ***Fetch Instruction:***

Using the Program Counter, the CPU retrieves the instruction from memory and stores it in an instruction memory.

1. ***Decode Instruction:***

To identify the kind of instruction and the operands involved, the fetched instruction is decoded at the instruction decode stage. The type and purpose of the instruction determine which control signals are generated in order to carry out the instruction.

1. ***Execution Stage:***

During the execution phase, the instruction's stated computation or action is carried out. The relevant arithmetic or logical operation is carried out once the operands needed for the operation are retrieved from memory or the register file.

1. ***Memory Access Stage:***

If the instruction calls for accessing or changing data in memory, memory operations are carried out during the memory access stage. Instructions for load (LW) and store (SW) are mostly employed at this step. When a load instruction is used, information is taken from memory and put into a register. Data from a register is written into memory at a designated memory location in order to execute a store instruction.

1. ***Write-back Stage:***

The process of executing an instruction ends with the write-back stage. Depending on the kind of instruction, it involves writing the computation's results back to memory or the destination register.

## 1.3 Instruction Types

### 1.3.1 Register Type



* 6-bit function, to determine the specific operation of the instruction
* 4-bit Rd: destination register
* 4-bit Rs1: first source register
* 4-bit Rs2: second source register
* 14-bit unused

Instruction Decoding for R-Type Register:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Function | RTL | | | | |
| **FETCH** | **DECODE** | **EXECUTE** | **MEMORY** | **WRITEBACK** |
| AND(000000) | PC ← PC + 1 | A = Rs1 B = Rs2 | Rd ← Rs1 AND | - | Rd = ALU Out |
| ADD(00001) | PC ← PC + 1 | A = Rs1 B = Rs2 | Rd ← Rs1 + Rs2 | - | Rd = ALU Out |
| SUB(000010) | PC ← PC + 1 | A = Rs1 B = Rs2 | Rd ← Rs1 - Rs2 | - | Rd = ALU Out |

### 1.3.2 Instruction Decoding for I-Type Register



* 4-bit Rd: destination register
* 4-bit Rs1: first source register
* 16-bit immediate: unsigned for logic instructions, and signed otherwise.
* 2-bit mode: this is used with load/store instructions only, such that:

00: no increment, 01: post increment the base register 10-11: unused

Instruction Decoding for I-Type Register:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Function | RTL | | | | |
| **FETCH** | **DECODE** | **EXECUTE** | **MEMORY** | **WRITEBACK** |
| ANDI(000011) | PC ← PC + 1 | A = Rs1  B = Imm | Rd ← Rs1 AND Rs2 | - | ALU  output |
| ADDI(000100) | PC ← PC + 1 | A = Rs1  B = Imm | Rd ← Rs1 + Imm | - | ALU output |
| LW(000101) | PC ← PC + 1 | A = Rs1  B = Imm | Rd ← Rs1 + Imm | Rd ← M[Rs1 + Imm] | ALU output |
| LW.POI(000110) | PC ← PC + 1 | A = Rs1  B = Imm | Rd ← Rs1 + Imm | Rd ← M[Rs1 + Imm]  Rs1 ← Rs1 + 1 | ALU output |
| SW(000111) | PC ← PC + 1 | A = Rs1  B = Imm | Rd ← Rs1 + Imm | M[Rs1 + Imm] → Rd | - |
| BGT(001000) | PC ← PC + Imm | A = Rs1  B = Rd | If Rs1 < Rd | - | - |
| BLT(001001) | PC ← PC + Imm | A = Rs1  B = Rd | If Rs1 > Rd | - | - |
| BEQ(001010) | PC ← PC + Imm | A = Rs1  B = Rd | If Rs1 == Rd | - | - |
| BNE(001011) | PC ← PC + Imm | A = Rs1  B = Rd | If Rs1 != Rd | - | - |

### 1.3.3 Instruction Decoding for J-Type Register

1. Jump (jmp) & call:

* 6-bit function, to determine the specific operation of the instruction
* 26-bit Offset: signed immediate

2. Return (ret)

* 6-bit function, to determine the specific operation of the instruction
* 26-bit: unused

Instruction Decoding for J-Type Register:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Function | RTL | | | | |
| **FETCH** | **DECODE** | **EXECUTE** | **MEMORY** | **WRITEBACK** |
| JMP(001100) | PC ← PC + Immediate26 | PC + Imm26 | - | - | - |
| CALL(001101) | PC ← PC + Immediate26 | PC + Imm26 |  |  |  |
| RET(001110) | PC ← pop(stack)  Take top of the stack | PC = pop(stack) | - | - | - |

**1.3.4 Instruction Decoding for S-Type Register**

* 4-bit Rd: destination register
* 22-bit: unused

Instruction Decoding for S-Type Register:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Function | RTL | | | | |
| **FETCH** | **DECODE** | **EXECUTE** | **MEMORY** | **WRITEBACK** |
| PUSH (001111) | PC ← PC + 1 | PUSH(Rd)  Rd is pushed on the top of the stack | PUSH (Rd) | - | TOP of the Stack = ALU  Out |
| POP(010000) | PC ← PC + 1 | Rd = POP()  The top element of the stack is popped, and it is stored in the Rd register | Rd = Fetch(top of the stack) | - | - |

## 

## 1.4 Data path

## 1.5 Components used to build the processor

1. ***Program Counter (PC):***

* Responsible for monitoring the address of the subsequent command to be retrieved.

1. ***Instruction Memory:***

* Keeps program instructions in memory, which the PC retrieves when needed.

1. ***Register File:***

* A group of registers used to store data temporarily while computation is being done.

1. ***ALU (Arithmetic Logic Unit):***

* Manipulating data using logical and arithmetic processes.

1. **Data Memory**

* Data Memory contains Operations to load and save data by storing and retrieving it.

1. **Multiplexers**

* Used to choose an input from a range depending on control signals.

1. **Sign Extender**

* Extends the immediate value for instructions with immediate operands.

1. **Control Unit**

* Creates control signals to synchronize different components' actions.

1. **Buffer**

* The buffer facilitates efficient data flow and synchronization between pipeline stages by acting as a temporary storage unit in between them.

## 1.6 *Control* Unit and Control Signals

### 1.6.1 Main Control Unit

Depending on the input op-code and instruction type, the control unit generates several types of control signals. These control signals dictate how the processor operates at various phases of the execution of an instruction.

* The control unit generates the following control signals:

1. ***PC\_src***: signals have the function of updating the PC in response to return, branch instructions, and other modifications in the control flow.
2. ***Ext\_src:*** control signal that, after processing I-Type instructions, begins the sign extension procedure. It gives the processor instructions to duplicate the immediate value's sign bit to fill the whole word.
3. ***RegW1:*** This signal enables or disables the reading of data from the register file.
4. ***RegW2:*** This signal enables or disables the writing of data back to the register file.
5. ***Read:*** This signal enables or disables the reading of data from the Data memory.
6. ***Write:*** This signal enables or disables the writing of data back to the register file.
7. ***Reg\_Des:*** It can be used to choose the value from a register or the immediate value as the source for Rs2.
8. ***ALU\_src***: It could be used to choose the value for Rs2 from a register or the immediate value.
9. ***Wb\_data***: Involves choosing between an input taken from the stack and the outcome of an operation calculated in the ALU.
10. ***J\_src***: For choosing between an outcome of the sack memory and the 26-bit immediate value.

* The op-code and instruction type are used to dynamically determine the control signals. In case of encountering an unsupported op-code or instruction type, the control signals are assigned "don't care" values represented by 'x'.

### 1.6.2 Truth Table for ALU Control Units

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Type | Function | ALUsrc | ALUop | Reg\_Des |
| R-type | AND | 0 | AND | 0 |
| R-type | ADD | 0 | ADD | 0 |
| R-type | SUB | 0 | SUB | 0 |
| I-type | ADDI | 1 | ADD | 1 |
| I-type | ANDI | 1 | SUB | 1 |
| I-type | LW | 1 | ADD | 1 |
| I-type | LW.POI | 1 | ADD | 1 |
| I-type | SW | 1 | ADD | 1 |
| I-type | BEQ | 0 | SUB | 1 |
| I-type | BNE | 0 | SUB | 1 |
| I-type | BGT | 0 | SUB | 1 |
| I-type | BLT | 0 | SUB | 0 |
| J-type | JMP | X | X | X |
| J-type | CALL | X | X | X |
| J-type | RET | X | X | X |
| S-type | PUSH | X | PUSH | 1 |
| S-type | POP | X | POP | 1 |

* This table is essential for processor design since it specifies the behavior the processor should have for every form of instruction. It helps in producing control signals that direct the ALU, register file, and memory units, among other data-path components.

### 1.6.3 Another Truth Tables for Control Signals

1)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Type | Function | Zero flag | Carry Flag | Negative Flag | PC\_src | J-src |
| R-type | X | X | X | X | 0 | X |
| I-Type | ANDI | X | X | X | 0 | X |
| I-type | ADDI | X | X | X | 0 | X |
| I-type | LW | X | X | X | 0 | X |
| I-type | LW.POI | X | X | X | 0 | X |
| I-type | SW | X | X | X | 0 | X |
| I-type | BEQ | 1 | X | X | 2 | X |
| I-type | BNE | 0 | X | X | 2 | X |
| I-type | BGT | X | 1 | X | 2 | X |
| I-type | BLT | X | X | 1 | 2 | X |
| J-type | JMP | X | X | X | 1 | 0 |
| J-type | CALL | X | X | X | 1 | 0 |
| J-type | RET | X | X | X | 1 | 1 |
| S-type | PUSH | X | X | X | 0 | X |
| S-type | POP | X | X | X | 0 | X |

2)

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Type | Function | Read | Write | ExtSrc | RegW1 | RegW2 | WB | Wb\_data |
| R-type | AND, ADD, SUB | 0 | 0 | X | X | 0 | 1 | 0 |
| I-type | ADDI, ANDI | 0 | 0 | 1 | 1 | 0 | 1 | 0 |
| I-type | LW | 1 | 0 | 1 | 1 | 0 | 1 | 1 |
| I-type | LW.POI | 1 | 0 | 1 | 1 | 1 | 1 | 1 |
| I-type | SW | 0 | 1 | 1 | 0 | 0 | 0 | X |
| I-type | BEQ, BNE, BLT, BGT | 0 | 0 | 1 | 0 | 0 | 0 | X |
| J-type | JMP, CALL, RET | 0 | 0 | X | X | X | 0 | X |
| S-type | PUSH | 0 | 0 | X | X | X | 0 | X |
| S-type | POP | 0 | 0 | X | 1 | 0 | 1 | 2 |