Faculty of Engineering and Technology

Department of Electrical and Computer Engineering

COMPUTER ARCHITECTURE-ENCS4370

Final Project Report

Prepared by:

Ali Maree 1190502

Al-Ayham Maree 1191408

Sara Ammar 1191052

Instructor: Dr. Aziz Qaroush

Section: 2

Date: 15/2/2023

Introduction

This project involves designing and testing a simple Multi-Cycle RISC processor using Verilog HDL language and implementing an Instruction Set Architecture. The objectives of this project include designing a processor with an instruction size of 24 bits, conditionally executed instructions, eight general-purpose registers, a program counter, a status register with a zero (Z) flag bit, and three instruction types (R-type, I-type, and J-type). The project also requires implementing a multi-cycle Datapath with five stages, constructing a control unit using a state machine approach, and using five addressing modes similar to the MIPS32 ISA. The project aims to provide hands-on experience in designing and testing a processor using HDL language, implementing an instruction set architecture, and designing a multi-cycle Datapath with its control logic.

The multi-cycle Datapath in this project consists of five stages, which are similar to the ones presented in the class lectures. The five stages are Instruction Fetch (IF), Instruction Decode (ID), Execution (EX), Memory Access (MEM), and Write Back (WB). The IF stage is responsible for fetching instructions from memory, while the ID stage decodes the instruction and reads the necessary registers. The EX stage performs arithmetic and logical operations, the MEM stage accesses memory if necessary, and the WB stage writes the result back to the register file.

To control the multi-cycle Datapath, the project requires designing a control unit using a state machine approach. The control unit generates control signals that guide the operation of the Datapath in each stage, including signals that control the multiplexers, ALU, and memory interface. The control signals generated by the control unit depend on the instruction being executed and the current state of the Datapath.

Overall, this project provides a comprehensive understanding of processor design, instruction set architecture, and control logic. It also offers an opportunity to apply knowledge gained from class lectures and other related materials to design and test a simple multi-cycle RISC processor.

Table of Contents

Table of figures

Design Specification

1. Motivation

In this project, a simple multi-cycle RISC processor is designed according to the following specifications:

- Instruction size: 24 bits (word size 24-bit).
- Conditionally executed instructions.
- Eight 24-bit general-purpose registers: R0 to R7.
- R0 is hardwired to zero and cannot be written.
- Program Counter (PC) is a 24-bit special-purpose register with a 24-bit memory address space.
- 8-bit status register with the least significant bit as the zero (Z) flag bit.
- Suffix SF is appended to the instruction mnemonic to determine if the ALU instruction should update the flag bits or not.
- Three instruction types (R-type, I-type, and J-type).
- Five addressing modes like in MIPS32 ISA.

2. Instruction formats

The ISA for the processor includes three instruction formats: R-type, I-type, and J-type. These formats have two common fields:

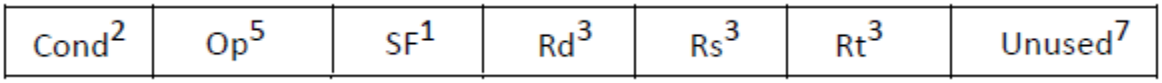
- **Opcode field** (5-bit) that specifies the operation to be performed.
- **Condition field** that enables conditional execution of each instruction. The condition field has two bits, and this project considers two conditions: equal and not equal. The Condition field in the processor ISA has three values:

    - 00: The instruction is always executed.
    - 01: The instruction is executed if the previous ALU instruction resulted in a zero result (zero flag bit is set), and it can be a NOP otherwise. This can be indicated in assembly language by appending the suffix EQ to the instruction mnemonic.
    - 10: The instruction is executed if the previous ALU instruction resulted in a non-zero result (zero flag bit is cleared), and it can be a NOP otherwise. This can be

indicated in assembly language by appending the suffix NE to the instruction mnemonic.

- 11: The value is unused.

2.1 R-type

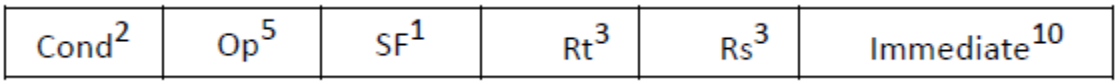


The R-type format has the following fields:

- 2-bit condition field (Cond).
- 5-bit opcode field (Op).
- 1-bit flag update field (SF), used only with subtraction instructions (SUBSF and SUBISF).
- 3-bit destination register field (Rd).
- 3-bit source 1 register field (Rs).
- 3-bit source 2 register field (Rt).
- 7-bit unused field for future use.

The SF field is always 0 for all other instructions, except for the two subtraction instructions mentioned above. An example of using the SUBSF instruction is provided, where the instruction subtracts the contents of two registers, updates the value in a destination register, and sets the Zero-Flag based on the result.

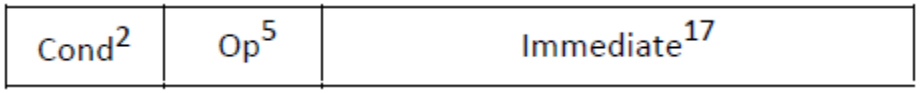
2.2 I-type



In addition to the common fields with R-type, it has:

- 3-bit destination register (Rt).
- 3-bit source 1 register (Rs).
- 10-bit signed immediate value in two's complement representation (2nd operand).

2.3 J-type



The J-type instruction in the processor's ISA includes the following fields:

- 2-bit condition field (Cond)
- 5-bit opcode field (Op)
- 17-bit signed immediate constant in two's complement representation, which represents the jump offset.

The J-type instruction format is different from the R-type and I-type instruction formats in that it includes a jump offset instead of register fields.

3. Instruction set

The following table shows the different instructions implemented. It shows their type, format, and their meaning in RTN. For each instruction listed below, there are two additional instruction mnemonics (EQ and NE). Moreover, only for SUB and SUBI operations, there is another instruction mnemonic (SF).

No.	Instr	Meaning	Encoding			
R-Type Instruction						
0	AND	Reg(Rd) = Reg(Rs) & Reg(Rt)	Op = 00000	Rs	Rt	Rd
1	CAS	Reg(Rd) = Max[Reg(Rs) , Reg(Rt)]	Op = 00001	Rs	Rt	Rd
2	Lws	Reg(Rd) = Mem[Reg(Rs) + Reg(Rt)]	Op = 00010	Rs	Rt	Rd
3	ADD	Reg(Rd) = Reg(Rs) + Reg(Rt)	Op = 00011	Rs	Rt	Rd
4	SUB	Reg(Rd) = Reg(Rs) – Reg(Rt)	Op = 00100	Rs	Rt	Rd
5	CMP	zero-flag = Reg(Rs) < Reg(Rt)	Op = 00101	Rs	Rt	0000
6	JR	PC = Reg(Rs)	Op = 00110	Rs	0000	0000
I-Type Instruction						
7	ANDI	Reg(Rt) = Reg(Rs) & Immediate10	Op = 00111	Rs	Rt	Immediate10
8	ADDI	Reg(Rt) = Reg(Rs) + Immediate10	Op = 01000	Rs	Rt	Immediate10
15	SUBI	Reg(Rt) = Reg(Rs) - Immediate10	Op = 01111	Rs	Rt	Immediate10
9	Lw	Reg(Rt) = Mem(Reg(Rs) + Imm10)	Op = 01001	Rs	Rt	Immediate10
10	Sw	Mem(Reg(Rs) + Imm10) = Reg(Rt)	Op = 01010	Rs	Rt	Immediate10
11	BEQ	Branch if (Reg(Rs) == Reg(Rt))	Op = 01011	Rs	Rt	Immediate10
J-Type Instruction						
12	J	PC = PC + Immediate17	Op = 01100	Immediate17		
13	JAL	R7 = PC + 3, PC = PC+Immediate17	Op = 01101	Immediate17		
14	LUI	R1 = Immediate17 << 4	Op = 01110	Immediate17		

Components

1. Memory

In this project, a memory component is used to store instructions and data separately. The instruction memory and data memory are separated in order to reduce conflicts and streamline the fetching and execution of instructions.

1.2. Instruction memory

The instruction memory is responsible for storing the program instructions that the processor will execute. In this implementation, the instruction memory is a read-only memory (ROM) that holds a fixed number of instructions. The input to the instruction memory is a 24-bit program counter (PC) that specifies the address of the instruction to be fetched from memory. The output from the instruction memory is a 24-bit instruction that is sent to the instruction decoder. Since the memory is read-only, it cannot be written to during the execution of the program.

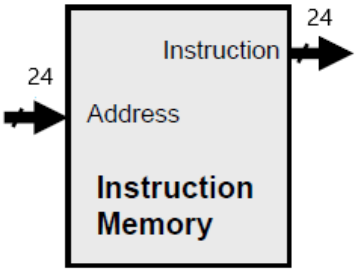


Figure 1: Instruction Memory

1.3. Data memory

The data memory module in this implementation is used to store and retrieve data required by the processor during the execution of instructions. It has an input port for a 24-bit address, 24-bit data input, a 1-bit memory read signal (MemRead), a 1-bit memory write signal (MemWrite), and a clock input. The MemRead signal is used to retrieve data from memory, while the MemWrite signal is used to write data to memory. When MemRead is asserted, the data memory module reads the data at the specified memory address and outputs the 24-bit data through the data output port. Similarly, when MemWrite is asserted, the data memory module writes the 24-bit input data to the specified memory address. The clock input signal synchronizes the read and write operations with the processor clock, ensuring that data is read and written at the appropriate time. The data memory module in this implementation has a size of 256 words, which translates to 16MB of addressable memory space.

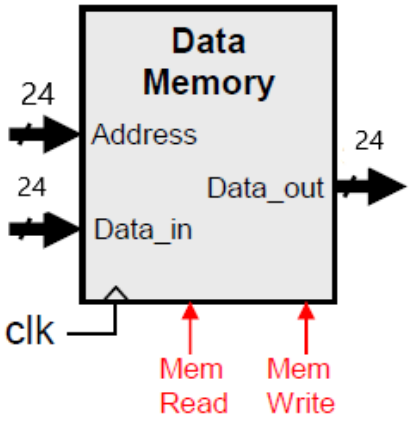
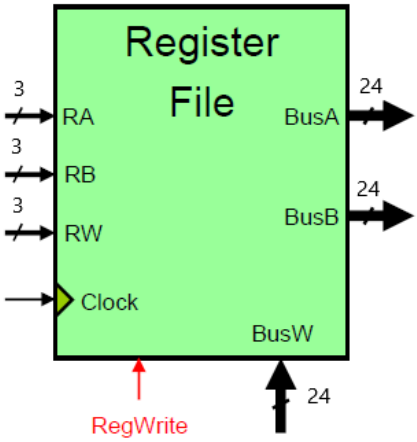


Figure 2: Data Memory

2. Register file

The register file is a key component of the processor that stores eight 24-bit general-purpose registers, named R0 through R7. It has three input ports for the source register addresses: RA and RB, each consisting of 3 bits, and RW consisting of 3 bits for the destination register address. The RegWrite signal, a 1-bit input, enables writing to the register file. The BusA and BusB, both 24-bit outputs, allow the read operation of the specified registers. When the RegWrite signal is asserted, the register file writes the 24-bit data on the Write bus to the register specified by the RW address. The register file is an essential component for the execution of instructions that require register operations. The ability to read and write to these registers efficiently is essential for the smooth and effective execution of the processor.



3. Arithmetic Logic Unit (ALU)

The Arithmetic Logic Unit (ALU) is a key component of the processor, responsible for performing arithmetic and logical operations on input data. The ALU module in this implementation has two 24-bit inputs, and a 3-bit ALUOp input which determines the type of operation to be performed. The module produces a 24-bit output, representing the result of the operation, and a 1-bit zero output which is set to 1 when the output of the ALU is 0, and set to 0 otherwise. The module supports five different operations, which are add, and, max, CMP, and shifter.

In this implementation of the ALU module, the subtraction process is carried out using the same addition circuit, but with the second input negated by taking its two's complement. This is achieved by inverting all of the bits of the second input and adding 1 to the result. The result of this operation is then fed into the ALU circuit as the second input, allowing for the subtraction operation to be performed.

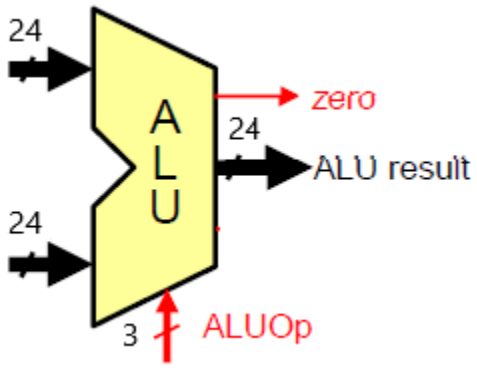


Figure 3: ALU

Table 1

ALUop	Operation
000	AND
001	ADD
010	MAX
011	CMP
100	Shifter

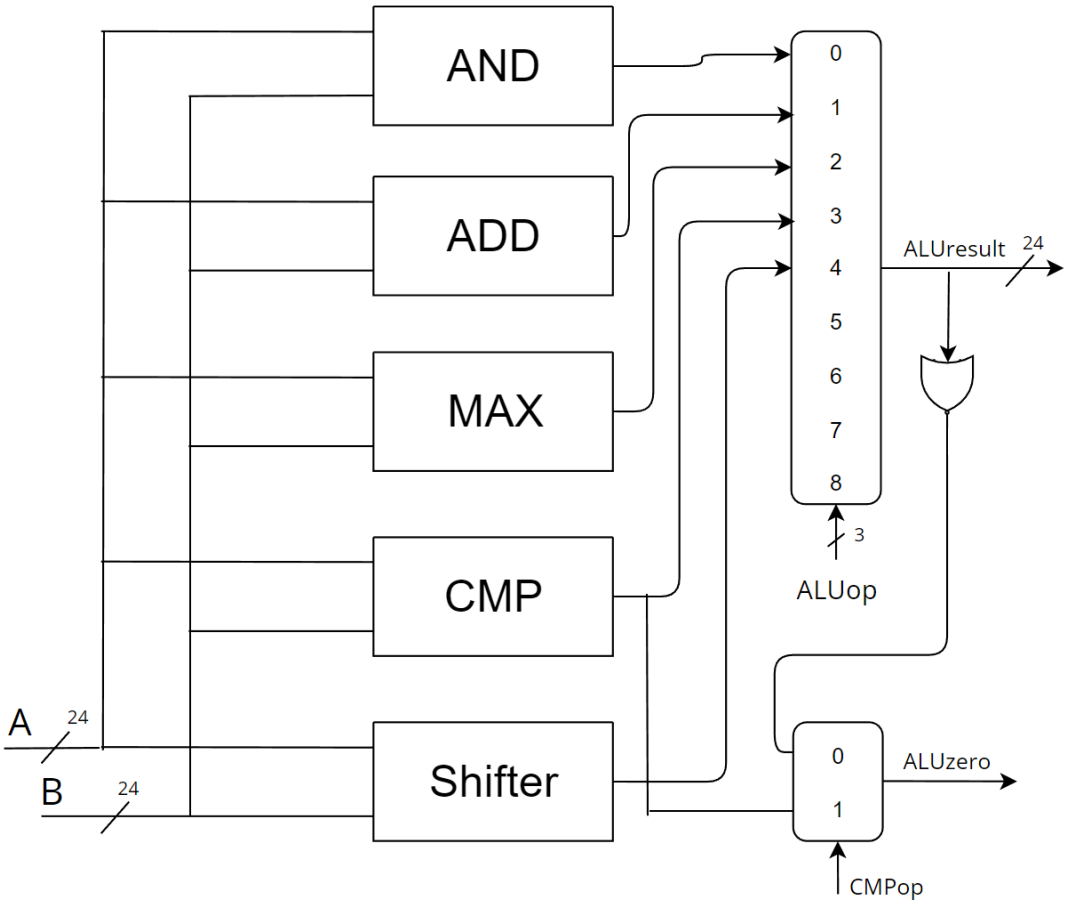


Figure 4: ALU implementation

4. Extender

The extender is a module that is used to extend the immediate values used in instructions from 10-bit or 17-bit to 24-bit. The extender takes as input the immediate value and an extender selection bit, which is a single bit that determines the size of the immediate value. If the extender selection bit is 0, the extender extends a 10-bit immediate value to 24 bits by sign-extending the immediate value to fill the remaining bits. If the extender selection bit is 1, the extender extends a 17-bit immediate value to 24 bits by sign-extending the immediate value to fill the remaining bits. The extended immediate value is then outputted from the extender as a 24-bit value. The extender is an important module in the processor as it allows the immediate values in instructions to be used as full 24-bit values, increasing the range of values that can be used in instructions.

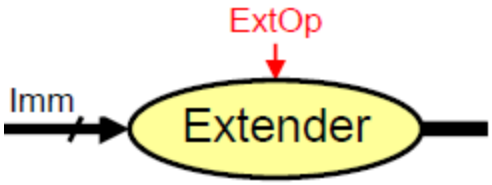


Figure 5: Extender

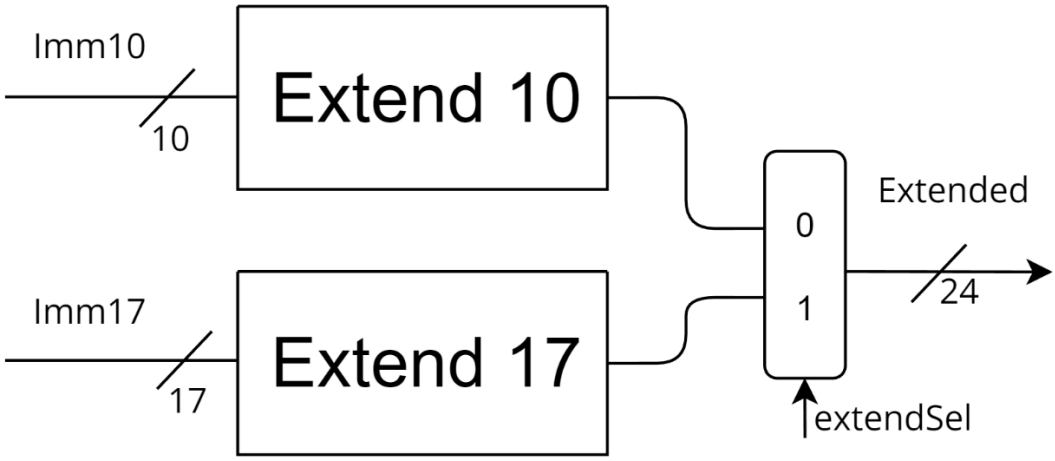


Figure 6: Extender implementation

5. Control Unit

The control unit is responsible for sequencing the execution of instructions. It receives the opcode of the current instruction and generates the necessary control signals to coordinate the fetch, decode, and execution phases of the instruction. The control unit controls the flow of data between the instruction memory, the register file, the ALU, and the data memory, and is responsible for ensuring that data is available when it is needed.

The control unit has a control path that manages the state of the processor during each phase of instruction execution. It includes a finite-state machine that governs the control signals and the flow of data during each cycle of the processor. The finite-state machine is implemented using logic gates that determine which phase of instruction execution the processor is in and generate the appropriate control signals.

During the fetch phase, the control unit generates the necessary control signals to read the instruction from memory and update the program counter. During the decode phase, the control unit decodes the opcode of the instruction and generates the necessary control signals to read the operands from the register file or memory. Finally, during the execute phase, the control unit generates the necessary

control signals to perform the specified operation and write the result back to the register file or memory.

5.1 Truth Tables

After building the Datapath, the control signals were derived as shown in Tables below:

Table 2

INPUT				OUTPUT											
INST	Cond	OP	SF	PCsrc	RSsrc	RBsrc	RWsrc	RegWR	ALUsrc	ALUop	MRD	MWR	WB	extd	subReg
AND	00	00000	0	00	0	0	0	1	1	AND	0	0	10	X	0
CAS	00	00001	0	00	0	0	0	1	1	MAX	0	0	10	X	0
Lws	00	00010	0	00	0	0	0	1	1	ADD	1	0	11	X	0
ADD	00	00011	0	00	0	0	0	1	1	ADD	0	0	10	X	0
SUB	00	00100	0	00	0	0	0	1	1	SUB	0	0	10	X	1
CMP	00	00101	0	00	0	0	0	0	1	CMP	0	0	X	X	0
JR	00	00110	0	10	0	X	X	0	X	X	0	0	X	X	X
ANDI	00	00111	0	00	0	X	0	1	0	AND	0	0	10	0	0
ADDI	00	01000	0	00	0	X	0	1	0	ADD	0	0	10	0	0
Lw	00	01001	0	00	0	X	0	1	0	ADD	1	0	11	0	0
Sw	00	01010	0	00	0	1	X	0	0	ADD	0	1	X	0	0
BEQ	00	01011	0	11	0	1	X	0	1	SUB	0	0	X	0	1
J	00	01100	0	01	X	X	X	0	X	X	0	0	X	1	X
JAL	00	01101	0	01	X	X	10	1	X	X	0	0	01	1	X
LUI	00	01110	0	00	1	X	01	1	X	X	0	0	10	1	X
SUBI	00	01111	0	00	0	X	0	1	0	SUB	0	0	10	0	0

The table above show the control signals when the instruction is executed, when the instruction is not executed the truth table is the same but the signals RegW, MRD and MWR are set to zeros.

Table 3

Opcode					ALUOp			Operation
Opcode [4]	Opcode [3]	Opcode [2]	Opcode [1]	Opcode [0]	[2]	[1]	[0]	
0	0	0	0	0	0	0	0	AND
0	0	0	0	1	0	1	0	MAX
0	0	0	1	0	0	0	1	ADD
0	0	0	1	1	0	0	1	ADD
0	0	1	0	0	0	0	1	ADD[SUB]
0	0	1	0	1	0	1	1	CMP
0	0	1	1	0	X	X	X	X
0	0	1	1	1	0	0	0	AND
0	1	0	0	0	0	0	1	ADD
0	1	0	0	1	0	0	1	ADD
0	1	0	1	0	0	0	1	ADD
0	1	0	1	1	0	0	1	ADD[SUB]
0	1	1	0	0	X	X	X	X
0	1	1	0	1	X	X	X	X

0	1	1	1	0	1	0	0	shifter
0	1	1	1	1	0	0	1	ADD[SUB]

5.2 Boolean Expression

By using Table 2 & 3 and handling conditions and zero flag the following boolean expressions were derived:

- **WB[1]** = (~opcode[4] & ~opcode[3] & ~opcode[1]) | (~opcode[4] & opcode[1] & opcode[0]) | (~opcode[4] & opcode[3] & ~opcode[0]).
- **WB[0]** = (~opcode[4] & opcode[2] & ~opcode[1] & opcode[0]).
- **RWsrc[1]** = (~opcode[4] & opcode[3] & opcode[2] & ~opcode[1]).
- **RWsrc[0]** = (~opcode[4] & opcode[2] & opcode[1] & ~opcode[0]).
- **ALUop[0]** = (~opcode[4] & ~opcode[2] & opcode[1]) | (~opcode[4] & opcode[2] & ~opcode[1]) | (~opcode[4] & opcode[3] & opcode[0]) | (~opcode[4] & opcode[3] & ~opcode[2]).
- **ALUop[1]** = (~opcode[4] & ~opcode[3] & ~opcode[1] & opcode[0]).
- **ALUop[2]** = (~opcode[4] & opcode[2] & opcode[1] & ~opcode[0]).
- **extendSel** = (~opcode[4] & opcode[2] & ~opcode[1]) | (~opcode[4] & opcode[2] & ~opcode[0]);
- **RSsrc** = (~opcode[4] & opcode[3] & opcode[2] & ~opcode[0]).
- **ALUsrc** = (~opcode[4] & ~opcode[2] & opcode[1] & opcode[0]) | (~opcode[4] & ~opcode[3] & ~opcode[2]) | (~opcode[4] & ~opcode[3] & ~opcode[1]).
- **MWR** = (~condition[1] & ~opcode[4] & opcode[3] & ~opcode[2] & opcode[1] & ~opcode[0] & Zflag) | (~condition[0] & ~opcode[4] & opcode[3] & ~opcode[2] & opcode[1] & ~opcode[0] & ~Zflag).
- **RBsrc** = (~opcode[4] & opcode[3]).
- **MRD** = (~condition[1] & ~opcode[4] & ~opcode[3] & ~opcode[2] & opcode[1] & ~opcode[0] & Zflag) | (~condition[1] & ~opcode[4] & opcode[3] & ~opcode[2] & ~opcode[1] & opcode[0] & Zflag) | (~condition[0] & ~opcode[4] & ~opcode[3] & ~opcode[2] & opcode[1] & ~opcode[0] & ~Zflag) | (~condition[0] & ~opcode[4] & opcode[3] & ~opcode[2] & ~opcode[1] & opcode[0] & ~Zflag).
- **subFlag** = (~opcode[4] & opcode[3] & opcode[1] & opcode[0]) | (~opcode[4] & ~opcode[3] & ~opcode[1]).
- **RegWR** = (~condition[1] & ~opcode[4] & ~opcode[3] & ~opcode[2] & Zflag) | (~condition[1] & ~opcode[4] & ~opcode[3] & ~opcode[1] & ~opcode[0] & Zflag) | (~condition[1] & ~opcode[4] & ~opcode[2] & ~opcode[1] & Zflag) | (~condition[1] & ~opcode[4] & opcode[3] & opcode[2] & opcode[1] & Zflag) | (~condition[0] & ~opcode[4] & ~opcode[3] & ~opcode[2] & ~Zflag) | (~condition[0] & ~opcode[4] & ~opcode[3] & ~opcode[1] & ~opcode[0] & ~Zflag) |

(~condition[0] & ~opcode[4] & ~opcode[2] & ~opcode[1] & ~Zflag) | (~condition[0] & ~opcode[4] & opcode[3] & opcode[2] & opcode[1] & ~Zflag) | (~condition[0] & ~opcode[4] & ~opcode[3] & opcode[1] & opcode[0] & ~Zflag) | (~condition[1] & ~opcode[4] & ~opcode[3] & opcode[1] & opcode[0] & Zflag) | (~condition[0] & ~opcode[4] & opcode[3] & ~opcode[1] & opcode[0] & ~Zflag) | (~condition[1] & ~opcode[4] & opcode[3] & ~opcode[1] & opcode[0] & Zflag).

- **PCsrc[1]** = (~condition[1] & ~opcode[4] & ~opcode[3] & opcode[2] & opcode[1] & ~opcode[0] & zF) | (~condition[1] & ~opcode[4] & opcode[3] & ~opcode[2] & opcode[1] & opcode[0] & zF & BEQf) | (~condition[0] & ~opcode[4] & ~opcode[3] & opcode[2] & opcode[1] & ~opcode[0] & ~zF) | (~condition[0] & ~opcode[4] & opcode[3] & ~opcode[2] & opcode[1] & opcode[0] & ~zF & BEQf).
- **PCsrc[0]** = (~condition[1] & ~opcode[4] & opcode[3] & ~opcode[2] & opcode[1] & opcode[0] & zF & BEQf) | (~condition[1] & ~opcode[4] & opcode[3] & opcode[2] & ~opcode[1] & zF) | (~condition[0] & ~opcode[4] & opcode[3] & ~opcode[2] & opcode[1] & opcode[0] & ~zF & BEQf) | (~condition[0] & ~opcode[4] & opcode[3] & opcode[2] & ~opcode[1] & ~zF).

5.3 State Machine

A state machine is a fundamental component of a multicycle processor, and it governs the flow of instructions through the processor's various stages. The state machine operates by transitioning from one state to another based on a set of predefined conditions, such as the instruction type, the values of certain control signals, or the contents of specific registers. Each state represents a distinct phase of the instruction execution process, such as instruction fetch, decode, and execution.

During each clock cycle, the state machine advances to the next state based on the conditions specified in the current state. For example, the state machine may transition from the instruction fetch state to the instruction decode state once the instruction has been fetched from memory. Similarly, the state machine may transition from the instruction execute state to the write-back state once the ALU has completed the necessary computations.

The table below shows the stages that each instruction passes through and the number of cycles it needs:

Table 4

INSTRUCTION	Stages	CPI
AND	IF, ID, EXEC, WB	4
CAS	IF, ID, EXEC, WB	4
Lws	IF, ID, EXEC, MEM, WB	5
ADD	IF, ID, EXEC, WB	4
SUB	IF, ID, EXEC, WB	4
CMP	IF, ID, EXEC	3
JR	IF, ID	2
ANDI	IF, ID, EXEC, WB	4
ADDI	IF, ID, EXEC, WB	4
Lw	IF, ID, EXEC, MEM, WB	5
Sw	IF, ID, EXEC, MEM	4
BEQ	IF, ID, EXEC	3
J	IF, ID	2
JAL	IF, ID, WB	3
LUI	IF, ID, EXEC, WB	4
SUBI	IF, ID, EXEC, WB	4

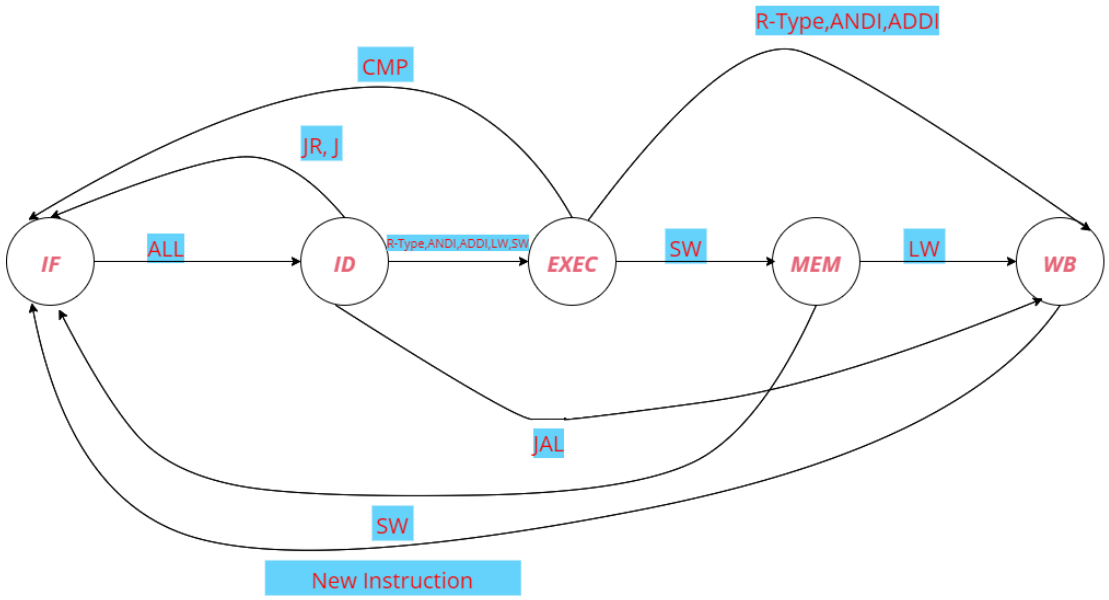


Figure 7:The state diagram

The Boolean expression to specify the next state based on the current state, instruction type, zeroflag and conditions:

- nextState[2] = (~curr[2] & curr[0] & ~opcode[4] & opcode[3] & opcode[2] & ~opcode[1] & opcode[0] & ~cond[1] & zF) | (~curr[2] & curr[0] & ~opcode[4] & opcode[3] & opcode[2] & ~opcode[1] & opcode[0] & ~cond[0] & ~zF) | (~curr[2] & curr[1] & ~opcode[4] & ~opcode[3] & ~opcode[1] & ~opcode[0] & ~cond[1]) | (~curr[2] & curr[1] & ~opcode[4] & ~opcode[3] & ~opcode[1] & ~opcode[0] & ~cond[0]) | (~curr[2] & curr[1] & ~opcode[4] & ~opcode[2] & ~opcode[1] & ~opcode[0] & ~cond[1]) | (~curr[2] & curr[1] & ~opcode[4] & ~opcode[2] & ~opcode[1] & ~opcode[0] & ~cond[0]) | (~curr[2] & curr[1] & ~opcode[4] & opcode[3] & opcode[2] & opcode[0] & ~cond[1]) | (~curr[2] & curr[1] & ~opcode[4] & opcode[3] & opcode[2] & opcode[0] & ~cond[0]) | (~curr[2] & curr[1] & ~opcode[4] & opcode[3] & opcode[2] & opcode[1] & ~cond[1]) | (~curr[2] & curr[1] & ~opcode[4] & opcode[3] & opcode[2] & opcode[1] & ~cond[0]) | (~curr[2] & curr[1] & curr[0] & ~opcode[4] & ~opcode[3] & ~cond[1]) | (~curr[2] & curr[1] & curr[0] & ~opcode[4] & ~opcode[3] & ~cond[0]) | (~curr[2] & curr[1] & ~opcode[4] & ~opcode[3] & ~opcode[2] & ~opcode[1] & ~cond[1]) | (~curr[2] & curr[1] & ~opcode[4] & ~opcode[3] & ~opcode[2] & ~opcode[1] & ~cond[0]) | (~curr[2] & curr[1] & ~opcode[4] & ~opcode[3] & opcode[1] & opcode[0] & ~cond[0]) | (~curr[2] & curr[1] & ~opcode[4] & ~opcode[3] & opcode[1] & opcode[0] & ~cond[1]) | (~curr[2] & curr[1] & curr[0] & ~opcode[4] & ~opcode[1] & ~cond[1]) | (~curr[2] & curr[1] & curr[0] & ~opcode[4] & ~opcode[1] & ~cond[0]).

- nextState[1] = (~curr[2] & curr[0] & ~opcode[4] & ~opcode[3] & ~opcode[1] & ~cond[1] & zF) | (~curr[2] & curr[0] & ~opcode[4] & ~opcode[3] & ~opcode[1] & ~cond[0] & ~zF) | (~curr[2] & ~curr[1] & curr[0] & ~opcode[4] & ~opcode[2] & ~cond[1] & zF) | (~curr[2] & ~curr[1] & curr[0] & ~opcode[4] & ~opcode[2] & ~cond[0] & ~zF) | (~curr[2] & curr[1] & ~curr[0] & ~opcode[4] & ~opcode[2] & opcode[1] & ~opcode[0] & ~cond[1]) | (~curr[2] & curr[1] & ~curr[0] & ~opcode[4] & ~opcode[2] & opcode[1] & ~opcode[0] & ~cond[0]) | (~curr[2] & curr[1] & ~curr[0] & ~opcode[4] & opcode[3] & ~opcode[2] & ~opcode[1] & opcode[0] & ~cond[1]) | (~curr[2] & curr[1] & ~curr[0] & ~opcode[4] & opcode[3] & ~opcode[2] & ~opcode[1] & opcode[0] & ~cond[0]) | (~curr[2] & ~curr[1] & curr[0] & ~opcode[4] & ~opcode[3] & ~opcode[0] & ~cond[0] & ~zF) | (~curr[2] & ~curr[1] & curr[0] & ~opcode[4] & ~opcode[3] & ~opcode[0] & ~cond[1] & zF) |

(~curr[2] & ~curr[1] & curr[0] & ~opcode[4] & opcode[3] & opcode[1] & ~cond[0] & ~zF) | (~curr[2] & ~curr[1] & curr[0] & ~opcode[4] & opcode[3] & opcode[1] & ~cond[1] & zF).

- nextState[0] = (~curr[2] & ~curr[1] & ~curr[0] & ~opcode[4] & ~cond[1]) | (~curr[2] & ~curr[1] & ~curr[0] & ~opcode[4] & ~cond[0]) | (~curr[2] & ~curr[0] & ~opcode[4] & ~opcode[2] & opcode[1] & ~opcode[0] & ~cond[1]) | (~curr[2] & ~curr[0] & ~opcode[4] & ~opcode[2] & opcode[1] & ~opcode[0] & ~cond[0]) | (~curr[2] & ~curr[0] & ~opcode[4] & opcode[3] & ~opcode[2] & ~opcode[1] & opcode[0] & ~cond[1]) | (~curr[2] & ~curr[0] & ~opcode[4] & opcode[3] & ~opcode[2] & ~opcode[1] & opcode[0] & ~cond[0]).

5.4 Update flags

When the instruction is SUBSF, CMP, BEQ, etc. the value of the flags is changed. The change of the value sometimes take place and sometimes it's neglected according to the SF flag, and the instruction itself. The value of these flags {zero, beq} is used in other control signals, so we must control the change of the variables by designing this component. The flags are updated at the first stage [IF] since the update takes the values of the previous instructions.

The boolean expressions in the update flag module:

zeroFlag = (~opcode[4] & ~opcode[2] & prevZ) | (~opcode[4] & ~opcode[3] & opcode[2] & ~opcode[1] & ~opcode[0] & SF & ALUflag) | (~opcode[4] & prevZ & ~SF) | (~opcode[4] & opcode[3] & opcode[2] & opcode[1] & opcode[0] & SF & ALUflag) | (~opcode[4] & ~opcode[3] & opcode[0] & prevZ) | (~opcode[4] & opcode[1] & ~opcode[0] & prevZ) | (~opcode[4] & opcode[3] & ~opcode[1] & prevZ).

BeqFlag = (~opcode[4] & ~opcode[3] & prevBEQ) | (~opcode[4] & ~opcode[1] & prevBEQ) | (~opcode[4] & ~opcode[0] & prevBEQ) | (~opcode[4] & opcode[3] & ~opcode[2] & opcode[1] & opcode[0] & ALUflag) | (~opcode[4] & ~prevZ & prevBEQ & SF) | (~opcode[4] & opcode[2] & prevBEQ).

6. Full Datapath

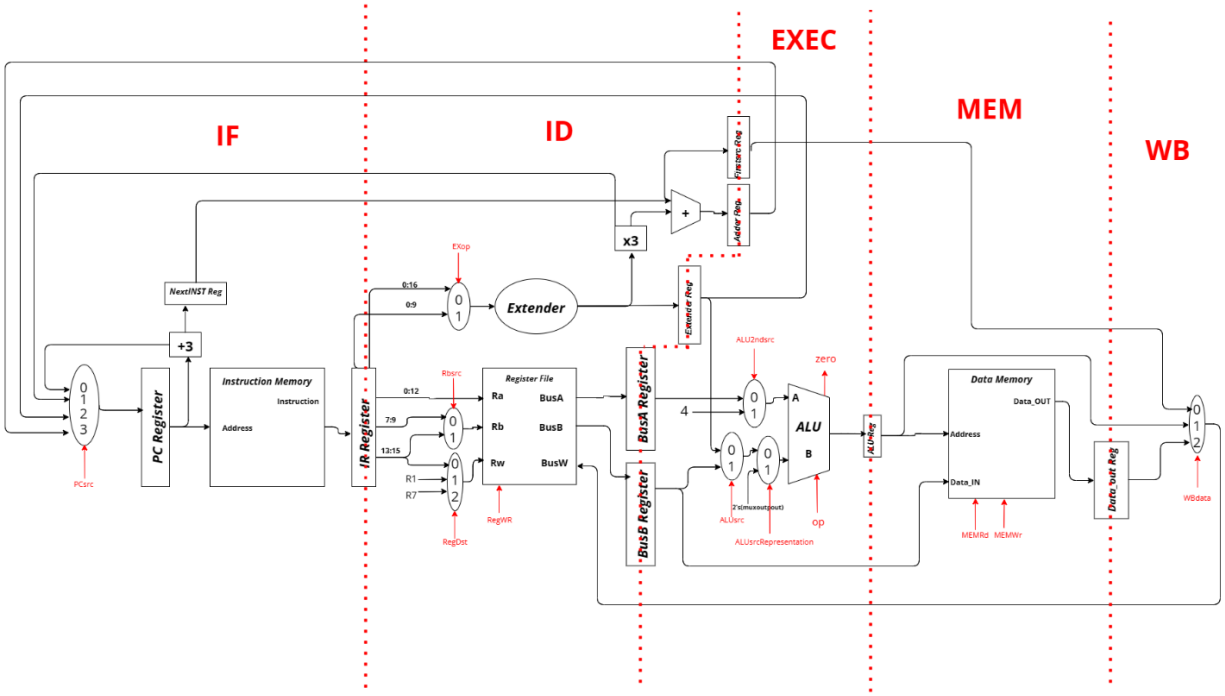


Figure 8: Full Datapath

Simulation and Testing

All test cases in the drive link below:

Testing - Google Drive

Conclusion and features work

In conclusion, the Multi-Cycle Processor architecture is a powerful and flexible processor design that can execute complex instructions with greater efficiency and flexibility by dividing them into multiple cycles. This allows for more efficient use of hardware resources and faster execution of instructions. In addition, the Multi-Cycle Processor can employ pipelining, which involves breaking down the instruction execution into stages that can be executed in parallel, leading to even greater efficiency gains. However, pipelining can introduce hazards, such as data hazards, control hazards, and structural hazards, which must be addressed using techniques such as forwarding, branch prediction, and hardware interlocks. Overall, the Multi-Cycle Processor is a robust design that can be used to efficiently execute complex instructions, but it is important to take into account the potential hazards introduced by pipelining and use appropriate techniques to mitigate them.

Appendix

1. We have written Python code to convert the MIPS instructions into a Binary Representation:

```
import re

def decimal_to_3bit_binary(decimal):
    binary = ""
    for i in range(2, -1, -1):
        binary += str(decimal >> i & 1)
    return binary

def twos_complement_10(decimal):
    if decimal >= 0:
        return bin(decimal)[2:].zfill(10)
    else:
        return bin(2**10 + decimal)[2:]

def twos_complement_17(decimal):
    if decimal >= 0:
        return bin(decimal)[2:].zfill(17)
    else:
        return bin(2**17 + decimal)[2:].zfill(17)

opcode = {
    "AND": "00000",
    "CAS": "00001",
    "LWS": "00010",
    "ADD": "00011",
    "SUB": "00100",
    "CMP": "00101",
    "JR": "00110",
    "ANDI": "00111",
    "ADDI": "01000",
    "SUBI": "01111",
    "LW": "01001",
    "SW": "01010",
    "BEQ": "01011",
    "J": "01100",
    "JAL": "01101",
    "LUI": "01110"
}

instruction_type = dict.fromkeys(["AND", "CAS", "LWS", "ADD", "SUB", "CMP", "JR"], "R")
instruction_type.update(dict.fromkeys(["ANDI", "ADDI", "SUBI", "LW", "SW", "BEQ"], "I"))
instruction_type.update(dict.fromkeys(["J", "JAL", "LUI"], "J"))

instruction_CPI = dict.fromkeys(["AND", "CAS", "ADD", "SUB", "ANDI", "ADDI", "SUBI", "SW", "JAL", "LUI"], 4)
instruction_CPI.update(dict.fromkeys(["LWS", "LW"], 5))
instruction_CPI.update(dict.fromkeys(["J", "BEQ", "JR", "CMP"], 3))
```

```
memory = ""
num_of_instructions = 0
total_CPI = 0

with open("code.txt", "r") as file:
    lines = file.readlines()
    lines = [line.replace("\n", "").upper() for line in lines]
    for instruction in lines:
        instruction = instruction.upper()
        instruction = instruction.replace(",", "")
        comp = instruction.split()
        bin_instruct = ""
        tempCond = ""
        tempSF = ""
        if 'NE' in comp[0]:
            tempCond += "10"
        elif 'EQ' in comp[0] and comp[0] != 'BEQ':
            tempCond += "01"
        else:
            tempCond += "00"

        if "SF" in comp[0]:
            tempSF += "1"
        else:
            tempSF += "0"

        comp[0] = comp[0].replace("EQ" if comp[0] != 'BEQ' else "", "").replace("NE", "").replace("SF", "")
        if opcode.get(comp[0]) == None:
            continue
        total_CPI += instruction_CPI.get(comp[0])
        if instruction_type.get(comp[0]) == "R":
            bin_instruct += tempCond
            bin_instruct += opcode.get(comp[0])
            bin_instruct += tempSF
            if comp[0] == "CMP":
                bin_instruct += "0"*3
                for i in range(len(comp) - 1):
                    bin_instruct += decimal_to_3bit_binary(int(re.sub(r'\D', '', comp[i+1])))
                bin_instruct += "0"*7
            elif comp[0] == "JR":
                bin_instruct += "0" * 3
                bin_instruct += decimal_to_3bit_binary(int(re.sub(r'\D', '', comp[1])))
                bin_instruct += "0" * 10
            else:
                for i in range(len(comp) - 1):
                    bin_instruct += decimal_to_3bit_binary(int(re.sub(r'\D', '', comp[i+1])))
                bin_instruct += "0"*7

        elif instruction_type.get(comp[0]) == "I":
            bin_instruct += tempCond
```

```
            bin_instruct += opcode.get(comp[0])
            bin_instruct += tempSF

            for i in range(len(comp) - 2):
                bin_instruct += decimal_to_3bit_binary(int(re.sub(r'\D', '', comp[i + 1])))

            bin_instruct += twos_complement_10(int(comp[3]))

        elif instruction_type.get(comp[0]) == "J":
            bin_instruct += tempCond
            bin_instruct += opcode.get(comp[0])
            bin_instruct += twos_complement_17(int(comp[1])) if comp[0] == "LUI" else twos_complement_17(lines.index(comp[1]))
            #bin_instruct += twos_complement_17(lines.index(comp[1]))
        print("Instruction: " + instruction.replace("\n", ""))
        print("Binary representation: " + bin_instruct)
        print("Hexadecimal representation: " + hex(int(bin_instruct, 2)))
        print("Length: " + str(len(bin_instruct)) + " bits")
        print()
        #memory += hex(int(bin_instruct, 2)).upper().replace("0X", "24'h") + ", "
        tempStr = hex(int(bin_instruct, 2)).upper().replace("0X", "")
        tempStr = "0" * (6 - len(tempStr)) + tempStr
        arrBytes = re.findall('.{1,2}', tempStr)
        tempStr = [("8'h" + arrBytes[i] + ",") for i in range(len(arrBytes))]
        memory += " ".join(tempStr) + " "
        num_of_instructions += 3


memory += ("8'h00, ") * (1024 - num_of_instructions)
index = memory.rindex(',')
new_mem = memory[:index] + memory[index+1:]

instruction_memory_code = "reg [7:0] mem [0:1023] = '{" + new_mem + "};\n"

print("-" * 20)
print("Instruction memory Verilog code\n")
print(instruction_memory_code)
print("-" * 20)
print("Number of instructions: " + str(num_of_instructions))
print("Total CPI: " + str(total_CPI))
```

2. Because the State Machine's truth table was too long, we wrote Python code to extract the maxterms from the table to find the NextState equations:

```
import csv
def truth_table_to_decimal(truth_table):
    decimal_values = []
    for row in truth_table:
        decimal_value = 0
        for i, value in enumerate(row):
            if value == '1':
                decimal_value += 2 ** (len(row) - i - 1)
        decimal_values.append(decimal_value)
    print("Len: " + str(len(decimal_values)))
    return decimal_values

def read_truth_table_from_csv(file_path):
    N0 = []
    N0x = []
    N1 = []
    N1x = []
    N2 = []
    N2x = []
    with open(file_path, 'r') as csv_file:
        csv_reader = csv.reader(csv_file)
        for row in csv_reader:
            if row[11] == '1':
                N2.append(row[0:11])

            if row[11] == 'x':
                N2x.append(row[0:11])

            if row[12] == '1':
                N1.append(row[0:11])

            if row[12] == 'x':
                N1x.append(row[0:11])

            if row[13] == '1':
                N0.append(row[0:11])

            if row[13] == 'x':
                N0x.append(row[0:11])
    print("N2:")
    print(truth_table_to_decimal(N2))
    print("N2 don't care:")
    print(truth_table_to_decimal(N2x))
    print("\n")

    print("N1:")
    print(truth_table_to_decimal(N1))
```

```
    print("N1 don't care:")
    print(truth_table_to_decimal(N1x))
    print("\n")

    print("N0:")
    print(truth_table_to_decimal(N0))
    print("N0 don't care:")
    print(truth_table_to_decimal(N0x))
    print("\n")

read_truth_table_from_csv('truth_table_ready.csv')
```