Faculty of Engineering and Technology
Electrical and Computer Engineering Department

ENCS4370-Computer Architecture

Project#2

Prepared by:

Aws Shaheen	**1212585**	**sec. 2**
Alhasan Manasrah	**1211705**	**sec. 1**
Mohammed Khdour	**1212517**	**sec. 3**

Instructors: Ayman Hroub & Aziz Qaroush

Deadline: 22-06-2024

Abstract

This project aims to generate a multi cycle RISC processor that contain control logic, instruction memory, data memory, and functional units like ALU and register files,it breaks down instruction execution into multiple steps, such as instruction fetch, instruction decode, execution memory, and the write back stage.

Table of Contents

Table of Figures

Table of Tables

1. Theory

1.1 Instruction Memory

In computer architecture the instruction memory is specialized type of memory used within a computer system to store the set of instructions that the processor executes. It fetches the instruction to decode and execute them in multiple cycles.

1.2 Register File

The Register file is an essential component of the CPU that contains set of registers which are small, fast storage locations directly accessible by the processor. These registers are used to store data temporarily during the execution of instructions.

1.3 ALU" Arithmetic Logic Unit"

The Arithmetic Logic Unit is responsible for performing arithmetic and logical operations, it's the main part in the execution stage, it decides which operation it's going to perform depending on the opcode, it performs several instructions like Add, Sub as arithmetic operations, or And OR as logical operations.

1.4 Data Memory

The Data Memory stores data used and produced by the CPU during its operations, information that the instructions manipulate, it's responsible of the Load, Store instructions, it either load the data from the registers to specific memory address or store the data saved in a register to specific memory address.

1.5 Multiplexers

A multiplexer is a combinational circuit that has many data inputs and a single output, depending on control or select inputs. For N input lines, log2(N) selection lines are required, or equivalently, for 2^n input lines, n selection lines are needed. Multiplexers are also known as "N-to-1 selectors," parallel-to-serial converters, many-to-one circuits, and universal logic circuits. They are mainly used to increase the amount of data that can be sent over a network within a certain amount of time and bandwidth, [1] while generating the RISC processor we used the three 2x1 Multiplexers, two 4x1 Multiplexers one of them was not fully used.

1.6 Sign Extenders

The sign extenders used are very simple they are responsible for two jobs only, first one is to extend the unsigned immediate to 16 Bits by adding zeros to the left, second one is to extend the signed immediate to 16 Bits by adding ones to the left for the negative sign or zeros to the positive sign

1.7 PC Control

The main control is essential unit in the CPU, it is responsible of set of control signals used in the Datapath especially the control signals that are responsible for deciding which PC to use for the next instruction to be fetched, also it takes zero and negative flag pits and produces the ALUop which in an input for the ALU to decide which operation will be performed.

opcode	Mode	zero	negative	PCSrc
0000	X	X	X	0
0001	X	X	X	0
0010	X	X	X	0
0011	X	X	X	0
0100	X	X	X	0
0101	X	X	X	0
0110	0	X	X	0
0110	1	X	X	0
0111	X	X	X	0
1000	0	0	0	2
1000	1	0	X	2
1001	0	X	1	2
1001	1	X	1	2
1010	0	1	X	2
1010	1	0	X	2
1011	0	0	X	2
1011	1	0	X	2
1100	X	X	X	1
1101	X	X	X	1
1110	X	X	X	3
1111	X	X	X	0

Table 1: PC Control signals Truth Table

1.8 Main Control

The main control is essential unit in the CPU, it is responsible of set of control signals used in the Datapath that helps to reduce the number of the components used to generate the RISC CPU, it generates the following control signals depending on the instruction and what are the control signals needed to perform the operation correctly:

Op	RASrc	RBSrc	RegDst	ExOP	RegWr	ALUSrc	MemRd	MemWR	Sv_Imm	ExtOpMem	MemOut	WBdata
Rtype	0	0	0	X	1	0	0	0	X	X	X	0
ADDI	0	X	0	1	1	1	0	0	X	X	X	0
ANDI	0	X	0	0	1	1	0	0	X	X	X	0
LW	0	X	0	1	1	1	1	0	X	X	0	1
LBu	0	X	0	1	1	1	1	0	X	0	1	1
LBs	0	X	0	1	1	1	1	0	X	1	1	1
SW	0	X	X	1	0	1	0	1	0	X	X	X
B	0	1	X	X	0	0	0	0	X	X	X	X
BZ	2	1	X	X	0	0	0	0	X	X	X	X
Jump	X	X	X	X	0	X	0	0	X	X	X	X
Call	X	X	1	X	1	X	0	0	X	X	X	2
Set	1	X	X	X	0	X	0	0	X	X	X	X
SV	0	X	X	1	0	X	0	1	1	X	X	X

Table 2: Main Control Signals Truth Table

2. Procedure

2.1 RTL Design

In this part the RTL design helped us when building the data path, and made it easier, it explains each instruction pass through which stages.

2.1.1 R-Type

Fetch instruction: instruction← MEM[PC]

Fetch operand: data1←REG[Rs1], data2 ← Reg [Rs2]

Execute operation: ALU_Result ← op (data1, data2)

Write ALU result: REG[Rd] ←ALU_result

Next pc: PC←PC+2

2.1.2 I-Type

Fetch instruction: instruction← MEM[PC]

Fetch operand: data1←REG[Rs1], data2 ← EXT (imm^5)

Execute operation: ALU_Result ← op (data1, data2)

Write ALU result: REG[Rd] ←ALU_result

Next pc: PC←PC+2

2.1.3 LW

Fetch instruction: Instruction ←MEM[PC]

Fetch base register: base ←Reg(rs1)

Calculate address: address ←base + sign_extend(imm5)

Read memory: data ←MEM [address]

Write register Rd: Reg (Rd) ←data

Next PC address: PC ←PC + 2

2.1.4 LBs

Fetch instruction: Instruction ←MEM[PC]

Fetch base register: base ←Reg(rs1)

Calculate address: address ←base + sign_extend(imm5)

Read memory: data ←MEM [address]

Extend byte: byte ← sign_ext(data [7:0])

Write register Rd: Reg (Rd) ←byte

Next PC address: PC ←PC + 2

2.1.5 LBu

Fetch instruction: Instruction ←MEM[PC]

Fetch base register: base ←Reg(rs1)

Calculate address: address ←base + sign_extend(imm5)

Read memory: data ←MEM [address]

Extend byte: byte ← zero_ext(data [7:0])

Write register Rd: Reg (Rd) ←byte

Next PC address: PC ←PC + 2

2.1.6 SW

Fetch instruction:	Instruction ← MEM[PC]
Fetch registers:	base ← Reg(rs1), data ← Reg (Rs2)
Calculate address:	address ← base + sign_extend(imm5)
Write memory:	MEM [address] ← data
Next PC address:	PC ← PC + 2

2.1.7 BGT

Fetch instruction:	instruction← MEM[PC]
Fetch operand:	data1←REG[Rs1], data2 ← Reg [Rs2]
Greater than:	zero, negative ← subtract (data1, data2)
Branch:	if (~zero && ~negative)
	PC ←PC + 2 + 2x sign_ext(offset^5)
	else
	PC←PC+2

2.1.8 BGTZ

Fetch instruction:	instruction← MEM[PC]
Fetch operand:	data1←REG[R0], data2 ← Reg [Rs2]
Greater than:	zero, negative ← subtract (data1, data2)
Branch:	if (~zero && ~negative)
	PC ←PC + 2 + 2x sign_ext(offset^5)
	else
	PC←PC+2

2.1.9 BLT

Fetch instruction:	instruction← MEM[PC]
Fetch operand:	data1←REG[Rs1], data2 ← Reg [Rs2]
Greater than:	zero, negative ← subtract (data1, data2)
Branch:	if (~zero && negative)
	PC ←PC + 2 + 2x sign_ext(offset^5)
	else
	PC←PC+2

2.1.10 BLTZ

Fetch instruction: instruction← MEM[PC]

Fetch operand: data1←REG[R0], data2 ← Reg [Rs2]

Greater than: zero, negative ← subtract (data1, data2)

Branch: if (~zero && negative)

PC ←PC + 2 + 2x sign_ext($offset^5$)

else

PC←PC+2

2.1.11 BEQ

Fetch instruction: instruction← MEM[PC]

Fetch operand: data1←REG[Rs1], data2 ← Reg [Rs2]

Greater than: zero, negative ← subtract (data1, data2)

Branch: if (zero)

PC ←PC + 2 + 2x sign_ext($offset^5$)

else

PC←PC+2

2.1.12 BEQZ

Fetch instruction: instruction← MEM[PC]

Fetch operand: data1←REG[Rs1], data2 ← Reg [Rs2]

Greater than: zero, negative ← subtract (data1, data2)

Branch: if (zero)

PC ←PC + 2 + 2x sign_ext($offset^5$)

else

PC←PC+2

2.1.13 BNE

Fetch instruction: instruction← MEM[PC]

Fetch operand: data1←REG[Rs1], data2 ← Reg [Rs2]

Greater than: zero, negative ← subtract (data1, data2)

Branch: if (~zero)

PC ←PC + 2 + 2x sign_ext($offset^5$)

else

PC←PC+2

2.1.14 BNEZ

Fetch instruction:	instruction← MEM[PC]
Fetch operand:	data1←REG[R0], data2 ← Reg [Rs2]
Greater than:	zero, negative ← subtract (data1, data2)
Branch:	if (~zero)
	PC ←PC + 2 + 2x sign_ext(offset^5)
	else PC←PC+2

2.1.15 JMP

Fetch instruction:	instruction← MEM[PC]
Target PC address:	target ← PC [15:10] || address9 || '0'
JUMP:	PC ← target

2.1.16 Call

Fetch instruction:	instruction← MEM[PC]
Target PC address:	target ← PC [15:10] || address9 || '0'
Save address:	REG[R7] ← PC + 2
JUMP:	PC ← target

2.1.17 RET

Fetch instruction:	instruction← MEM[PC]
Fetch NextPC:	nextPC ← REG[R7]
Next PC:	PC ← nextPC

2.1.18 SV

Fetch instruction:	Instruction ← MEM[PC]
Fetch data:	address ← REG[Rs1], data ← EXT (imm8)
Write memory:	MEM [address] ← data
Next PC address:	PC ← PC + 2

2.2 State Diagram

In this part we have drown the state diagram that represents the stages that every instruction pass through, we have used Lucid.app[2] for this diagram.

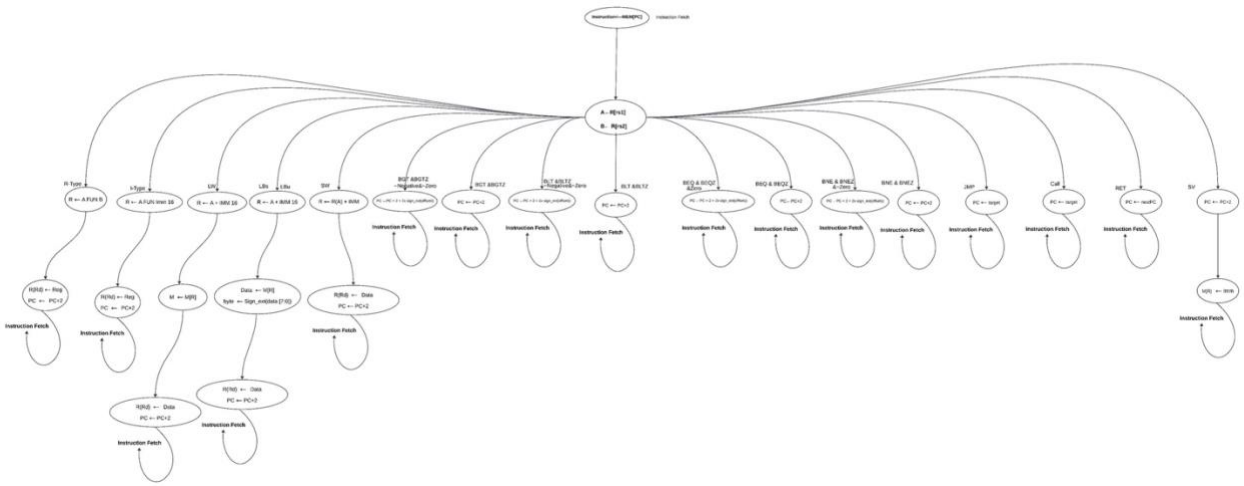

Figure 1: State Diagram

2.3 The Logic Equations

2.3.1 Main Control Logic Equations

RAsrc: OP(BZ) → RAsrc = 2
OP(Ret) → RAsrc = 1
Else → RAsrc = 0
RBsrc = Branch
RegDst = call
ExtOp = ~ANDI
RegWr = ~(sw+Branch+jump+ret+sv)
ALUsrc = ~(R_type+Branch)
MemRd = Lw+ LBU+LBS
MemWr = Sw +Sv
SvImm = Sv
ExtOpMemory = LBS
MemOut = ~LW
WB: LW+ LBU +LBS = 1 → WB = 1
Call = 1 → WB = 2
else → WB = 0

2.3.2 PC Control Logic Equation

PCSrc=2
if (opcode==1000∧mode==0∧zero==0∧negative==0) ∨
(opcode==1000∧mode==1∧zero==0)∨
(opcode==1000∧mode==1∧zero==0) ∨
(opcode==1001∧mode==0∧negative==1)∨
(opcode==1001∧mode==0∧negative==1)∨
(opcode==1001∧mode==1∧negative==1)∨
(opcode==1001∧mode==1∧negative==1) ∨
(opcode==1010∧mode==0∧zero==1)∨
(opcode==1010∧mode==0∧zero==1) ∨
(opcode==1010∧mode==1∧zero==0)∨
(opcode==1010∧mode==1∧zero==0) ∨
(opcode==1011∧mode==0∧zero==0)∨
(opcode==1011∧mode==0∧zero==0) ∨
(opcode==1011∧mode==1∧zero==0)∨
(opcode==1011∧mode==1∧zero==0)

PCSrc=1 if (opcode==1100)∨(opcode==1101)PCSrc=1 if (opcode==1100)∨ (opcode==1101)

PCSrc=3 if (opcode==1110)PCSrc=3 if (opcode==1110)

PCSrc=0 otherwisePCSrc=0 otherwise

2.4 Building the Data path

In this part we built the data path using Lucid.app [2], the data path contains the 5 stages that represents the RISC Processor.

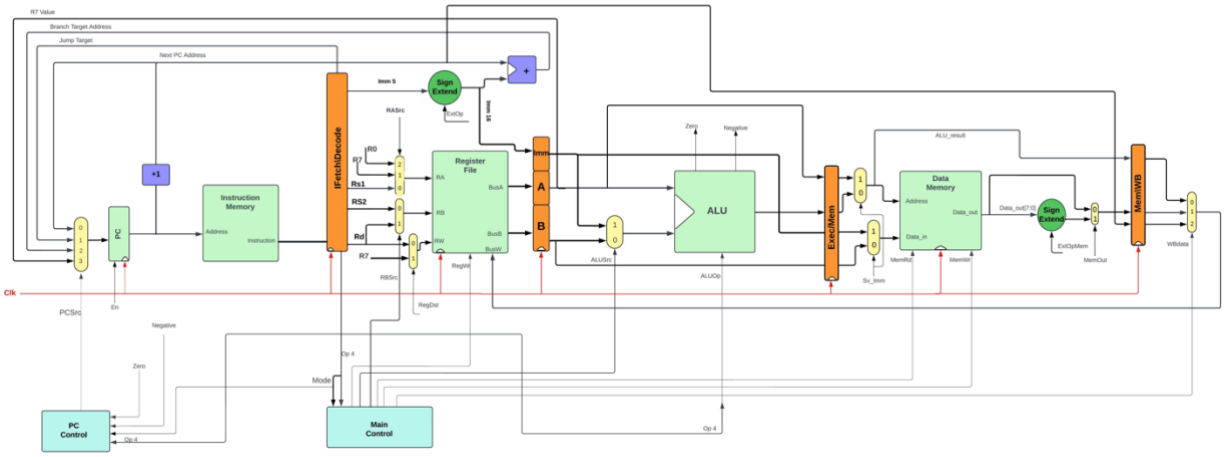

Figure 2 : Data Path

In figure 1 the data path contains at first the fetch stage, the fetch stage fetches the instruction from the instruction memory then the decode stage that split the instruction depending on each instruction type formula, like deciding what are the source and destination registers used and the op code in the R-type instructions like ADD,AND,SUB.

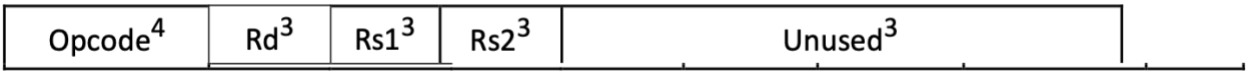

Figure 3: R-type

For the I-type instructions also in the decoding stage it decides the registers that are used in this instruction and the op code also with 1 bit mode for deciding if the operation is signed or unsigned like ADDI,ANDI.

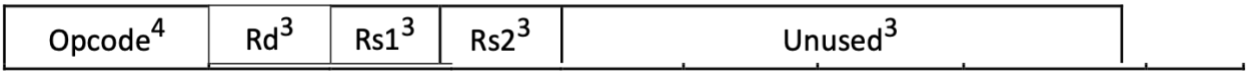

Figure 4: I-Type

The J type instructions are in two forms the first one is to decide the op code and the jump offset.

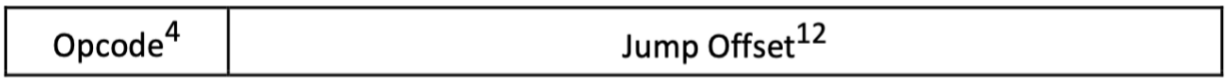

Figure 5: J-type first formula

The second just decides the op code and the next PC will be the value stored in R7, for ret instruction.

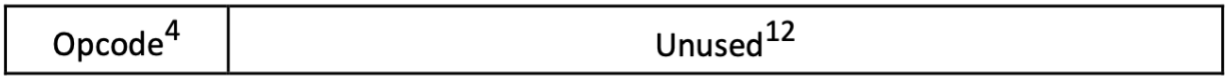

Figure 6: J-type second formula

At last, the S-type instructions, in the decode stage it specifies the source register, the op code and the immediate.

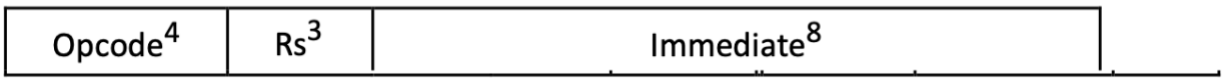

Figure 7: S-type

Some instructions like the J-type instructions finishes the execution in the decode stage, then after the decode stage the instructions that need more than the previous two stages the instruction go to the execution stage, in this the main functional unit is the ALU, its used for performing the calculations for the arithmetic operations or to perform the address calculations for the Load, Store instructions. Then the data memory stage which is only used by the load, store instructions its either store the data in specific address or load data from specific address, then in the end of the data path is the write back stage which writes the result of the ALU to the destination register, or the next PC, or the data loaded from the data memory and then write it back to the register, this happens by helping from the control units "The main control unit & The PC control unit" that generates the control signals, the control signals are passed to the multiplexers to decide what to pass through the mux, and enter the next stage correctly, the control units main job is to generate the control signal that helps in passing through the stages correctly, for example in the R-type instructions there Is a control signal named ALU source, to decide what to pass to the ALU, if its R-type instruction it pass the second source register, if its I-type instruction it pass the immediate with the first source register to the ALU "which is passed by default", the PC control generates control signals to decide on what is the PC that contains the next instruction, and generate the ALU op code.

2.5 Writing the Verilog Code

In this part we have written all modules needed to implement the RISC processor using EDA playground [3], and Active HDL [4], our code contains each component in the Datapath separated, and then we combined them in one Verilog file named Datapath contains also the test bench for testing our processor, and how it deals with the instructions given in the test bench.

2.6 Testing and Results

In this part we have tested our final project using different types of instructions, and looked through the wave form to see the results.

```
1  module instruction_Memory (pc,instruction);
2    input [15:0] pc;
3    output reg [15:0] instruction;
4    reg [7:0] Memory [0:255];
5
6    initial begin
7      Memory[0] = 8'b0101_0000;
8      Memory[1] = 8'b0001_0010;// add $R1,$R1,$R2
9
10     Memory[2] = 8'b0111_0000;
11     Memory[3] = 8'b1111_0010;// Sv
12
13     Memory[4] = 8'b0101_0000;
14     Memory[5] = 8'b0001_0010;// add
15
16     Memory[6] = 8'b0111_0000;
17     Memory[7] = 8'b1111_0010;// Sv
18
19 end
20
21   always @(*) begin
22     instruction[7:0] <= Memory [pc];
23     instruction [15:8] <= Memory [pc+1];
24   end
25 endmodule
```

Figure 8: first test code

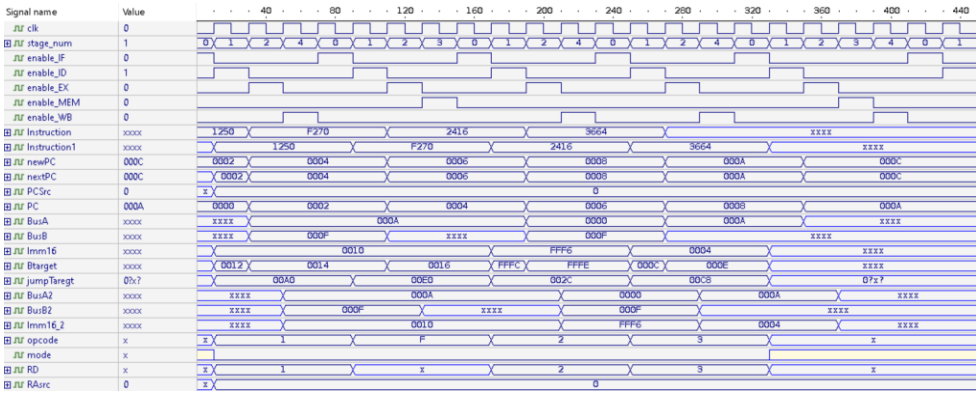

Figure 9: wave form for the first test

The precious figures show the instructions used to test and their results in the wave form, the first test contains 4 instruction 2 Add instructions and two SV instructions, the Add instruction ads R1 and R2, and store the result in R1, we used the instruction twice to check if the results are the same when testing same instruction.

```
module instruction_Memory (pc,instruction);
  input [15:0] pc;
  output reg [15:0] instruction;
  reg [7:0] Memory [0:255];

  initial begin

    // R-Type
    Memory[0] = 8'b0101_0000;// add(4) $R1(3),$R1(3),$R2(3)
    Memory[1] = 8'b0001_0010;// add(4) $R1(3),$R1(3),$R2(3)

    // S-Type
    Memory[2] = 8'b0111_0000;
    Memory[3] = 8'b1111_0010;

    // R-Type
    Memory[4] = 8'b0001_0110;
    Memory[5] = 8'b0010_0100;// sub(4) $R1(3),$R1(3),$R2(3)

    // I-Type
    Memory[6] = 8'b0110_0100;
    Memory[7] = 8'b0011_0110;// addi


end

  always @(*) begin
    instruction[7:0] <= Memory [pc];
    instruction [15:8] <= Memory [pc+1];
  end
endmodule
```

Figure 10: second test code

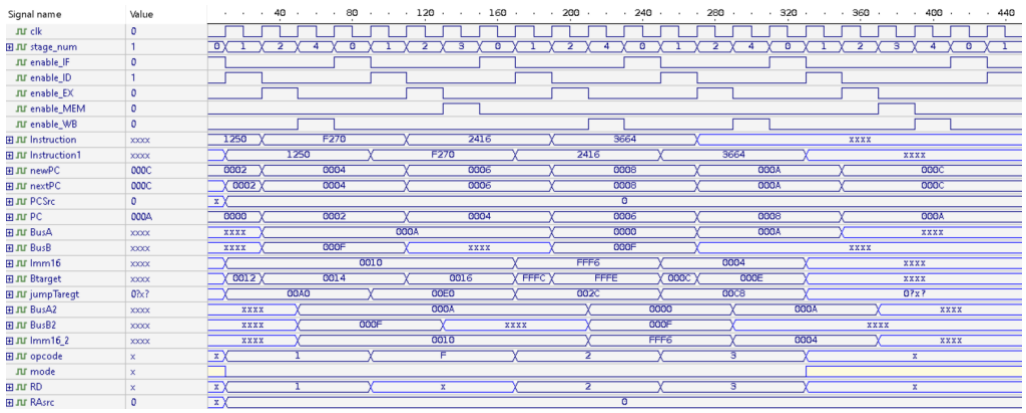

Figure 11: Test 2 wave Form

In the second test we tested the SUB and the ADDI, the results are shown in the previous wave form, the sub instruction substitutes the value stored in R2 from the value stored in R1, and store the result in R1, then the ADDI the destination register is R6, the source register is R6, the immediate is 4.

3. Conclusion

In conclusion, the design and implementation of the multi-cycle RISC processor demonstrate a structured approach to CPU architecture, emphasizing control logic, instruction memory, and data memory. The integration of various components, such as the ALU, register files, and multiplexers, ensures efficient instruction execution across multiple cycles. The project successfully highlights the importance of breaking down instruction execution into distinct stages, enhancing the processor's overall performance and functionality. This comprehensive exploration of a multi-cycle RISC processor underscores the significance of meticulous design and verification in advancing computer architecture.

4. References

[1]: https://www.geeksforgeeks.org/multiplexers-in-digital-logic/

[2]: https://www.lucidchart.com/pages/

[3]: https://www.edaplayground.com/

[4]: https://www.aldec.com/en/products/fpga_simulation/active_hdl_student