BIRZEIT UNIVERSITY

FACULTY OF ENGINEERING AND TECHNOLOGY

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

ENCS4370 - Computer Architecture

Project 2 - Multi-Cycle RISC Processor

Prepared By

Doha Hmeid 1190120

Sara Rummaneh 1190005

Instructor

Dr. Aziz Qaroush

Section 2

29/1/2024

I. Abstract

This project aims to develop a RISC multi-cycle processor using the Verilog hardware description language and the active-HDL software tool. The project implements all the theoretical and practical aspects involved in creating a processor's architecture, featuring five stages: fetch, decode, ALU (Arithmetic Logic Unit), memory access, and write back. And supporting a predefined set of instructions and includes specific information and properties.

The report offers a comprehensive overview of the developmental stages involved in the design of the Central Processing Unit (CPU). This encompasses instruction analysis, writing instruction Register-Transfer Level (RTL) descriptions, transforming RTL into functional components, RICS machine design, and the implementation of Datapath and control signals.

Furthermore, the report explores the validation of the CPU's functionality through various testing methodologies, such as testbenches, and the execution of programs to ensure both accuracy and performance of the designed CPU.

II. Table of contents

III. Table of Figures

IV. Table of Tables

1. Introduction

1.1. RISC Processors

RISC processors, or Reduced Instruction Set Computer processors, are characterized by their support for a limited yet comprehensive set of instructions. These instructions are of fixed size, enabling the CPU to execute them within a single machine cycle. Consequently, the CPU's performance is enhanced as all the stages of an instruction can occur within a single clock cycle [1].

Due to the simplicity of the RISC CPUs' Instruction Set Architecture (ISA), they feature a large number of general-purpose registers for storing data and addresses. Data transfer between these registers and memory is exclusively facilitated by special load/store instructions, thereby reducing memory accesses and associated delays [1].

Furthermore, RISC processors support instruction pipelining to fasten CPU processes and maximize throughput. This involves breaking down instructions into multiple stages, allowing them to run in parallel with other instructions' stages on smaller clock cycles [1].

1.2. RISC Processors Architecture

The architecture of RISC processors is characterized by the division of processor instructions into five stages: instruction fetch, instruction decode, execute, memory access, and write back. Initially, the control unit transmits the necessary control signal, while the program counter holds the address of the instruction to be fetched. This gives the processor the ability to fetch the next instruction from memory while the current instruction is being decoded. This parallel behavior is enabled because each instruction has a fixed size [1].

Upon completion of the current instruction's execution, the program counter is incremented by 1 to indicate the address of the next instruction, and this process continues until a JUMP/BRANCH instruction is encountered, at which point the program counter is either incremented or decremented based on the offset [1].

The content of the program counter is used as input to the instruction memory to fetch the next instruction, and the instruction register (IR) is responsible for storing and decoding the instruction.

While the arithmetic and logical unit (ALU) utilizes the opcode to execute the desired operation, with the control signal provided by the control unit [1].

The registers array, which contains the general-purpose registers, features two read ports and one write port, enabling data to be written into or read from a specified address based on signals transmitted from the main control unit. The registers provide the necessary operands for any operation, and the result of the execution is stored in the destination register through the multiplexer [1].

While this represents the general architecture of RISC processors, specific variations may exist to accommodate the requirements of individual processors. In our project, we have made several modifications to this fundamental architecture to align with the requirements of our processor's ISA.

1.3. Multi-Cycle RISC Processors

Single-cycle processors have limited throughput and speed performance due to the requirement for control and data signals to propagate through the processor within a single cycle. This results in longer cycle times and creates some time periods of inactivity for many hardware components. Whereas, multi-cycle processors divide instructions into their fundamental function units and execute each part in separate shorter clock cycles. This behavior allows faster cycle times as signals have less distance to travel [2].

In a multi-cycle processor, each instruction takes only as many cycles as it needs, ensuring that no instruction takes extra time or uses more functional units than necessary. Therefore, the primary advantage of a multi-cycle design is the ability to share hardware elements, such as the ALU, among various tasks [2].

2. Design and Implementation

2.1. Processor Specifications

The processor we're going to implement have a certain specification which include:

- Instruction size is 32bits.
- Word size is also 32bits.
- It contains 16 general purpose register, each has 32bits length.
- It contains 2 special purpose register, each has 32bits length. Which is the program counter (PC) register which holds the address of next instruction. And the stack pointer (SP) register which points to top empty element of the stack.

The memory in this machine also has specific requirements which include:

- The memory is word addressable.
- There is only two physical memory which are instruction memory and data memory.
- The instruction memory contains the instructions to be executed and it represents the code segments.
- The data memory stores both data segments and stack segments. Such that this memory unit is split to store both types of segments.

2.2. Instruction Set Architecture (ISA)

The instruction set (ISA) supports four instruction types (R-type, I-type, J-type, and S-type). And each type has a specific format as illustrated below.

1. Register Type (R-Type)

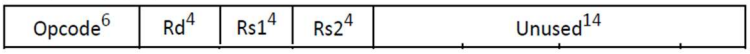

Figure 2-1: R-Type Instruction Format

2. Immediate Type (I-Type)

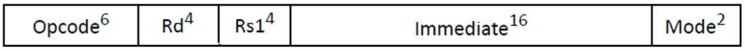

Figure 2-2: I-Type Instruction Format

3. Jump Type (J-Type)

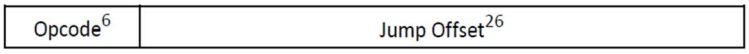

Figure 2-3: J-Type Instruction Format 1

Figure 2-4: J-Type Instruction Format 2

4. Stack (S-Type)

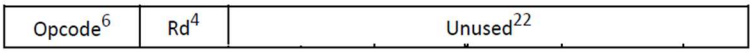

Figure 2-5: S-Type Instruction Format

The fields in the instruction types formats are explained in Table 2-1 below.

Field	Size	Instruction formats	Meaning
Opcode	6bits	Common among all instruction formats	used to distinguish the instruction form others
Rd	4bits	R-Type, I-Type, S-Type	The destination register
Rs1	4bits	R-Type, I-Type	The first source register
Rs2	4bits	R-Type	The second source register
Immediate	16bits	I-Type	Holds constant data and it's unsigned for logic instructions, and signed otherwise.
Mode	2bits	I-Type	Used to increment the Rs1 if mode=01
Jump Offset	26bits	J-Type	Used to calculate the target address to jump to
Unused	Multiple sizes depending on the instruction type	R-Type, J-Type, S-Type	Those bits are extra and have no meaning or use

Table 2-1: Instruction Format Fields

The processor's supported instructions from the ISA are illustrated in Table2-2 below.

No.	Instr.	Meaning	Opcode Value
R-Type Instructions			
1	AND	Reg(Rd) = Reg(Rs1) & Reg(Rs2)	000000
2	ADD	Reg(Rd) = Reg(Rs1) + Reg(Rs2)	000001
3	SUB	Reg(Rd) = Reg(Rs1) - Reg(Rs2)	000010
I-Type Instructions			
4	ANDI	Reg(Rd) = Reg(Rs1) & Imm16	000011
5	ADDI	Reg(Rd) = Reg(Rs1) + Imm16	000100
6	LW	Reg(Rd) = Mem(Reg(Rs1) + Imm16)	000101
7	LW.POI	Reg(Rd) = Mem(Reg(Rs1) + Imm16) Reg[Rs1] = Reg[Rs1] + 1	000110
8	SW	Mem(Reg(Rs1) + Imm16) = Reg(Rd)	000111
9	BGT	if (Reg(Rd) > Reg(Rs1)) Next PC = PC + sign_extended (Imm16) else PC = PC + 1	001000
10	BLT	if (Reg(Rd) < Reg(Rs1)) Next PC = PC + sign_extended (Imm16) else PC = PC + 1	001001
11	BEQ	if (Reg(Rd) == Reg(Rs1)) Next PC = PC + sign_extended (Imm16) else PC = PC + 1	001010
12	BNE	if (Reg(Rd) != Reg(Rs1)) Next PC = PC + sign_extended (Imm16) else PC = PC + 1	001011
J-Type Instructions			
13	JMP	Next PC = {PC[31:26], Immediate26 }	001100
14	CALL	Next PC = {PC[31:26], Immediate26 } PC + 1 is pushed on the stack	001101
15	RET	Next PC = top of the stack	001110
S-Type Instructions			
16	PUSH	Rd is pushed on the top of the stack	001111
17	POP	The top element of the stack is popped, and it is stored in the Rd register	010000

Table 2-2: Instructions' Encoding

2.3. Data Path

2.3.1. CPU

The overall design of the CPU containing all stages and supporting all instructions is shown in Figure2-6 below. In order to come with the multi-cycle CPU design, we divided the CPU into stages and implemented each stage after that we combined them together. In order to synchronize the stages behavior together, we added buffers between stages.

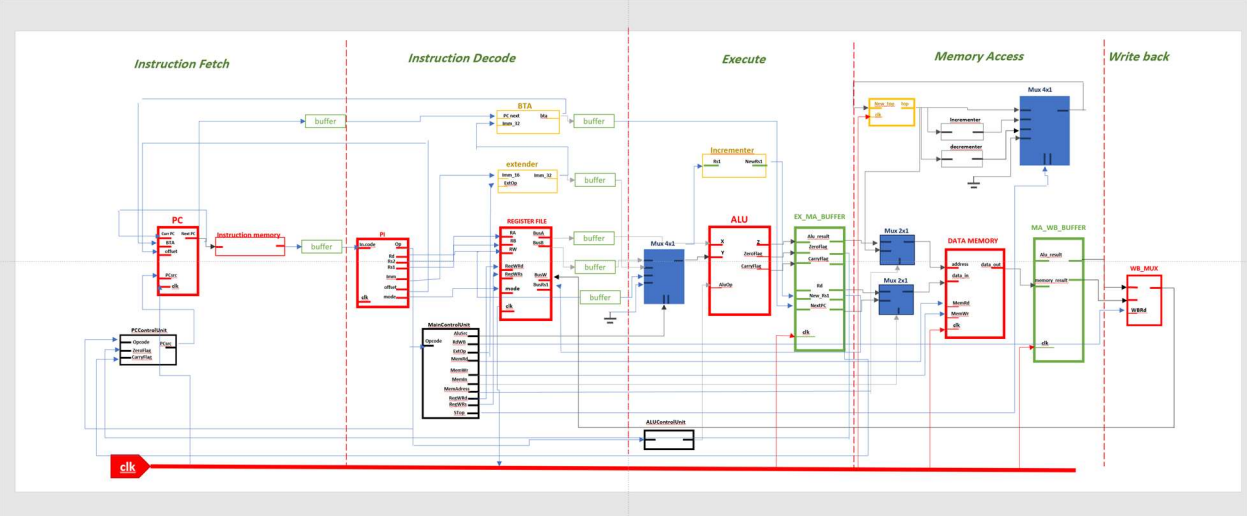

Figure 2-6: CPU Data Path

The flow of the processor's behavior:

1. Instruction fetch (Stage1)

In the first stage, the instruction is fetched from the instruction memory based on the address saved in the program counter. After that, the next program counter is calculated and it may have different values depending on the instruction type. The PC control unit is responsible for generating the signal 'PCscr' which is used to calculate the next PC address.

2. Instruction Decode (Stage2)

In the second stage, the instruction enters and then the values are extracted from it and are saved in the register file. Then, the value for the immediate is extended to 32bit. And the main control

unit takes the OpCode value and calculates the signals values that are needed to carry the flow of the CPU's behavior. Also, the decoded values are saved to the register file to use them in later stages.

3. Execute (Stage3)

The main functional unit in the execute stage is the ALU which stands for the arithmetic and logical unit. The ALU takes a signal called AluOp from the ALU control unit and based on it decides the operation to execute.

The ALU takes Rs1 as the first operand always. But the second operand has multiple sources depending on the instruction itself such that:

- The alu takes Rs2 as second operand for R-type instructions and outputs the arithmetic/logical operation result.
- The alu takes the extended immediate as second operand for arithmetic and logical I-type instructions and outputs the arithmetic/logical operation result.
- The alu takes the extended immediate as second operand for load/store I-type instructions and outputs the memory address as a result.
- The alu takes Rd as second operand for I-type jump instructions and calculates the subtraction result and based on it, it computes the flag values which are (ZeroFlag,CarryFlag).
- Other instructions don't use the ALU like jump and call.

In this stage, if the instruction was LW.POI, the value of Rs1 is incremented and saved in the stage buffer. Also, all the resulting values from this stage are buffered in the stage buffer which is called EX_MA_Buffer.

4. Memory Access (Stage4)

The main functional unit in this stage is the memory, and it's divided to hold both the data segments and stack segments. There are 2 multiplexers that control the behavior of the memory and each one takes a signal from the main control unit such that:

- The memory address equals the ALU result for the I-type load/store instructions, here the memory address is calculated from the ALU.
- The memory address equals the stack pointer for the S-type instructions, where in push Rd is pushed to the stack address and in pop the stack top is poped.
- The memory input equals Rd for the S-type PUSH instruction, since Rd needs to be written on the stack. And for I-type store instruction.
- The memory input equals the next PC for J-Type since the next PC is pushed to the stack.
- Other instructions don't use the memory.

Another unit that operates in this stage is the stack pointer which holds the address of the empty topmost cell from the stack segments in the memory. The stack pointer needs to be changes after popping/pushing to the memory stack segments. Such that:

- After pushing to the stack segments, the stack pointer (SP) needs to be incremented to point on the new top empty cell.
- After popping from the memory stack segments, the stack pointer needs to be decremented to the currently new top empty cell.
- Otherwise, in cases where we only read the value for the SP, no change happens on it.

The memory takes two control signals in order to control reading/writing on the memory and the signals are MemRd and MemWr and they are generated from the main control unit. Also, the stack pointer depends on a signal from the control unit to implement the previous behavior. Both the SP and the memory need a clock to be synchronized with the other CPU's stages. The output of this stage is buffered in the MA_WB_buffer.

5. Write Back (Stage5)

This is the last stage and not all the instructions reach it. The instructions that use this stage to write back a value on the Rd register from the register file. The new value may have two sources, either from the alu result directly in cases of arithmetic and logical operations. Or from the data memory in cases of load operation. A control signals controls this selection and its RdWB and it's generated from the main control unit.

2.3.2. Instruction Memory

The instruction memory is a read-only memory and it's used to take the instructions from the program's file and save them to use them in the CPU. Its unit takes a 32bit address for the instruction and outputs the 32bit instruction itself.

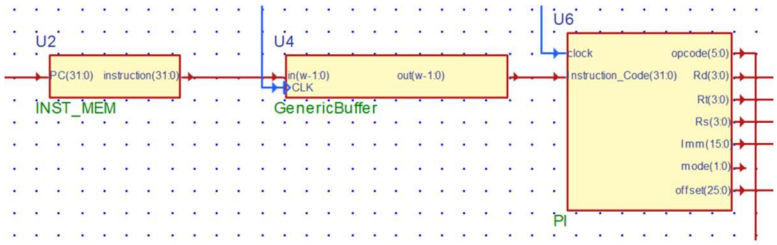

Figure 2-7: instruction memory block diagram

2.3.3. PC

The PC is a special purpose register holding the value for the program counter. It's used to fetch new instructions and control the flow of executing the program in cases of branching and jumping which change the sequential execution of programs. The register is synchronized with the CPU clock since it's value must be changed frequently, and there is a specific control unit for calculating the value for the next PC.

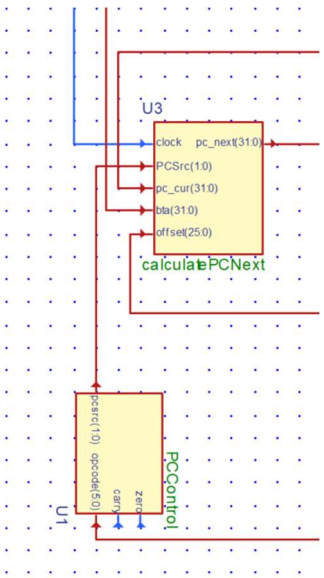

Figure 2-8: PC block diagram

2.3.4. Register File

The Register File maps the programs register addresses to the actual data inside them. It allows writing on the registers Rd and Rs1 such that it contains 2 read ports and 2 write ports. And each write port has a special signal that indicates the mode (disable/enable writing on the register). The following figure shows the block diagram of the register file.

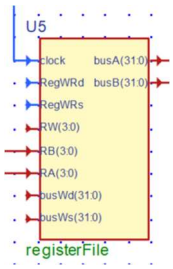

Figure 2-9: Register file block diagram

2.3.5. Extender

The extender unit takes the 16bit immediate and assigns 16 additional bits to extend it to a 32bit immediate. The extension is unsigned for logic instructions (where the ExtOp=0) and in this case the 16 additional bits equal zeros. Otherwise, the extension is signed (where the ExtOp=1) and in this case the 16 additional bits equal the most-high bit (sign bit) of the immediate. The following figure shows the block diagram of the extender

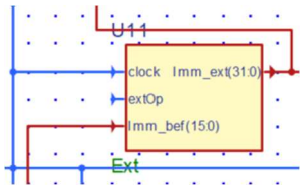

Figure 2-10: Extender block diagram

2.3.6. ALU

The ALU takes two inputs X and Y, each one has 32bit size. It also takes a control signal from the main control unit. And based on the selected AluOp, it does a certain operation on the inputs and calculates the output which is Z and it's also 32bit size. It generates two flags, the zero flag indicates if the results Z is all zeros or not. While the carry flag is high when the result Z is positive and negative otherwise. The following figure shows the block diagram of the ALU.

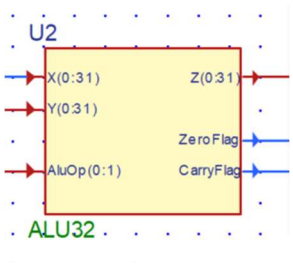

Figure 2-11: ALU block diagram

The figure below shows the simulation results for the ALU and it validates the ALU functionality.

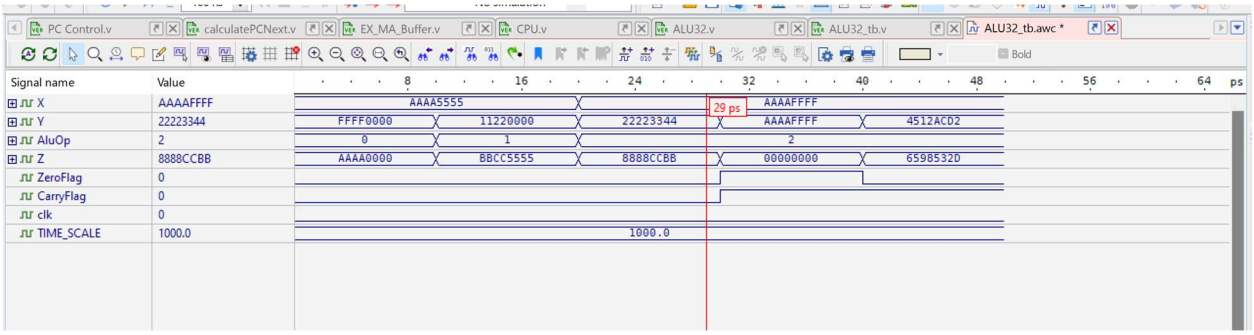

Figure 2-12: ALU test bench

2.3.7. SP

The SP is a special purpose register that is used to hold the value for the top empty cell from the stack segments in the data memory. And it depends on a clock, since it's value can be incremented (in case of pushing a new stack segment) or decremented (in case of popping a stack segment).

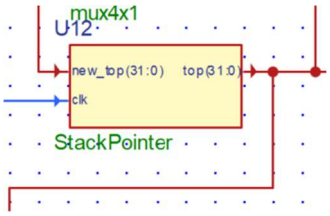

Figure 2-13: SP block diagram

2.3.8. Data Memory

The data memory holds two types of segments inside it which are the data segments and the stack segments. And the memory size is divided between them, the stack pointer initially points to the first empty segment from the stack segments in the data memory.

The data memory has 5 inputs which are:

- 32bit address port
- 32bit input
- MemWr - 1bit control signal for memory write
- MemRd - 1bit control signal for memory read
- Clk - 1bit clock signal

And it has one 32bit output which is the value read from the memory in loading instructions. To ensure CPU synchronization, the memory write instructions operate on the rising edge of the clock while the memory read instructions operate on the falling edge of the clock.

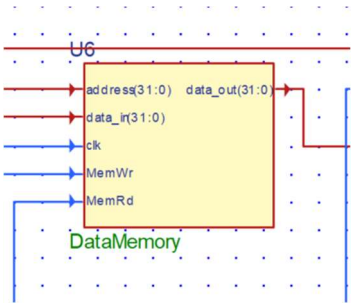

Figure 2-14: Data memory block diagram

2.4. Control Path

In order to control the behavior of the processor's data path, we came with the necessary control signals and then we built 3 control units which are: PC control unit, ALU control unit, and the Main control unit.

2.4.1. PC Control Unit

The PC control unit is responsible for generating the PCsrc signal which selects the source for the next PC address. It takes three inputs and based on them generates the PC mux4*1 selection line. The figure below shows the block diagram of the PC control unit with its inputs and outputs.

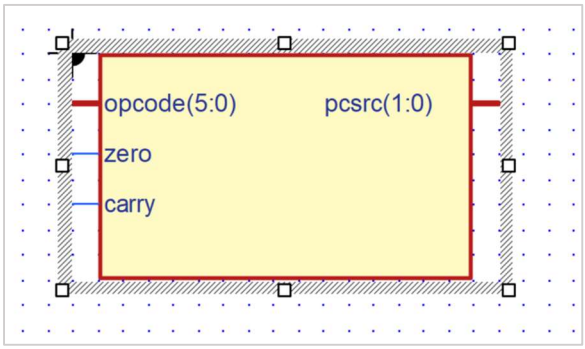

Figure 2-15: PC Control Unit Block Diagram

The table below shows the possible generated values for PCsrc and their meaning.

PCsrc Value	Meaning
00 = 0	Next PC = PC + 1 For normal behavior and branch not taken cases
01 = 1	Next PC = { PC[31:28] | offset } for JUMP, CALL instructions
10 = 2	Next PC = BTA = PC + sign_extended (Imm16) For branch instructions when branch is taken BGT, BLT, BEQ, BNE
11 = 3	Next PC = SP (top of the stack) For RET instruction

Table 2-3: PC Control Unit Signals

The truth table below shows all the instructions with their OpCode and the generated signals by the PC control unit for each one of them. The instructions that don't have an effect on the PC or the program flow have don't care values (x).

Instruction	Opcode	ZeroFlag	CarryFlag	PCsrc
AND	000000	x	x	0 = 00
ADD	000001	x	x	0 = 00
SUB	000010	x	x	0 = 00
ANDI	000011	x	x	0 = 00
ADDI	000100	x	x	0 = 00
LWs	000101	x	x	0 = 00
LW.POI	000110	x	x	0 = 00
SW	000111	x	x	0 = 00
BGT	001000	1	0	2 = 10
BLT	001001	0	0	2 = 10
BEQ	001010	1	x	2 = 10
BNE	001011	0	x	2 = 10
JMP	001100	x	x	1 = 01
CALL	001101	x	x	1 = 01
RET	001110	x	x	3 = 11
PUSH	001111	x	x	0 = 00
POP	010000	x	x	0 = 00

Table 2-4: PC Control Unit Truth Table

2.4.2. Main Control Unit

This is the main control unit and it generates most of the signals that are needed for the functional units. The figure below shows the block diagram of the main control unit with its inputs and outputs.

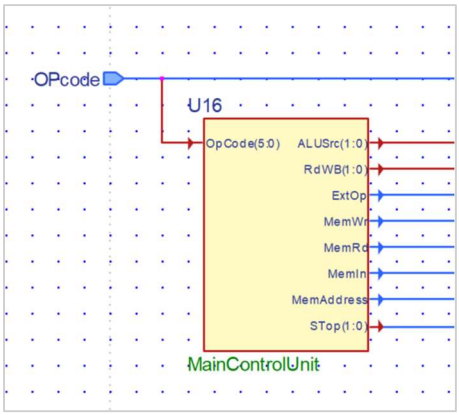

Figure 2-16: Main Control Unit Block Diagram

This control unit takes one input only which is the OpCode of the current instruction in order to calculate its signals. It generates multiple signals and the table below shows each one of them with their meaning and use.

Signal	Meaning	Value
ALUSrc	Used as the ALU MUX4*1 selection line To choose the second input/source for the ALU	00 Rs2 is the second input 01 The extended immediate is the second input 10 The Rd is the second input 11 nothing
RdWB	Used as the write back stage MUX2*1 selection line, to choose the value to write back on Rd	0 takes ALU result to write on Rd 1 takes memory output to write on Rd
ExtOp	Used as the extender signal, to extend the immediate with or without sign	0 unsigned extension for logical operations 1 signed extension
MemWr	To write on data memory	0 don't write on data memory 1 write on data memory
MemRd	To read from data memory	0 don't read from data memory 1 read from data memory
MemIn	Used as the MUX2*1 selection, to select the data memory input	0 take Rd as input to data memory 1 take next PC as input to data memory

MemAddress	Used as the MUX2*1 selection, to select the data memory address (data segment or stack segment)	0 take ALU result as memory address 1 take stack top pointer as memory address
RegWRd	Used to enable/disable writing on the register file Rd port	0 disable write on Rd 1 enable write on Rd
RegWRs	Used to enable/disable writing on the register file Rs1 port	0 disable write on Rs1 1 enable write on Rs1
STop	Used as the Stack MUX4*1 selection line To choose the value to update the stack top poinet	00 increment SP (stack pointer) 01 decrement SP (stack pointer) 10 don't change the value 11 nothing

Table 2-5: Main Control Unit Signals

The truth table below shows all the instructions with their OpCode and the generated signals by the main control unit for each one of them.

Instruction	Opcode	ALUSrc	RdWB	ExtOp	MemWr	MemRd	MemIn	MemAddress	RegWRd	RegWRs	STop
AND	000000	00	0	x	0	0	x	x	1	0	xx
ADD	000001	00	0	x	0	0	x	x	1	0	xx
SUB	000010	00	0	x	0	0	x	x	1	0	xx
ANDI	000011	01	0	0	0	0	x	x	1	0	xx
ADDI	000100	01	0	1	0	0	x	x	1	0	xx
LWs	000101	01	1	1	0	1	x	0	1	0	xx
LW.POI	000110	01	1	1	0	1	x	0	1	1	xx
SW	000111	01	x	1	1	0	0	0	0	0	xx
BGT	001000	10	x	1	0	0	x	x	0	0	xx
BLT	001001	10	x	1	0	0	x	x	0	0	xx
BEQ	001010	10	x	1	0	0	x	x	0	0	xx
BNE	001011	10	x	1	0	0	x	x	0	0	xx
JMP	001100	xx	x	1	0	0	x	x	0	0	xx
CALL	001101	xx	x	x	1	0	1	1	0	0	00
RET	001110	xx	x	1	0	1	x	1	0	0	10
PUSH	001111	xx	x	x	1	0	0	1	0	0	00
POP	010000	xx	1	x	0	1	x	1	1	0	01

Table 2-6: Main Control Unit Truth Table

2.4.3. ALU Control Unit

This control unit is used to decide the operation to be performed from the ALU operations. Such that there are 3 operations done in the ALU which are: logical and, addition, and subtraction. So, the unit takes the opcode of the instruction and based on that it generates a signal indicating the ALU operation. Below is the block diagram of the ALU control unit, it takes the Opcode and generates the AluOp signal which selects the ALU operation to be performed.

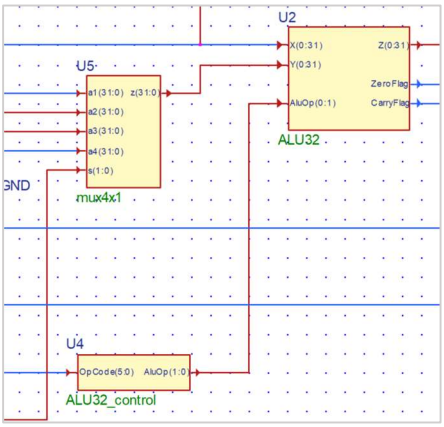

Figure 2-17: ALU Control Unit Block Diagram

The table below shows all the instructions with the AluOp signal generated for them by the ALU control unit. Some instructions don't need to use the Alu like the jumping instructions and the S-Type instructions. So, the values for them are don't care.

Instruction	Opcode	AluOp	ALU Operation
AND	000000	00	AND
ADD	000001	01	ADD
SUB	000010	10	SUB
ANDI	000011	00	AND
ADDI	000100	01	ADD
LWs	000101	01	ADD
LW.POI	000110	01	ADD
SW	000111	01	ADD
BGT	001000	10	SUB
BLT	001001	10	SUB

BEQ	001010	10	SUB
BNE	001011	10	SUB
JMP	001100	xx	x
CALL	001101	xx	x
RET	001110	xx	x
PUSH	001111	xx	x
POP	010000	xx	x

Table 2-7: Truth Table for ALU Control Unit

3. Simulation and Testing

To make sure that the induvial components work correctly, a test bench was written for each functional unit and their simulations were checked previously (note: the test benches are included in the project's folder).

To check that the CPU as a whole unit works correctly, we wrote a test bench for it and initialized the clock cycle's timing. Then, we wrote a simple program that contains multiple instructions from the ISA that this CPU supports. We translated these instructions to their binary representation and then to make them even simpler we wrote the hex-based values for them. After that, we gave the registers initial values to use them with the instructions' test.

3.1. The Test Program

The table below shows the instructions we wrote to test in our program. (note: the program can be found the project's folder as program.hex file).

Instruction	Binary value	Hex value
ADD R3, R1,R2	000001 0011 0001 0010 00000000000000	04c48000
ADDI R0,R3,8	000100 0000 0011 0000000000001000 00	100c0020
BGT R5,R1,2	001000 0101 0001 0000000000000010 00	21440008
AND,R0,R0,R0	000000 0000 0000 0000 00000000000000	00000000
JMP 8	001100 00000000000000000000001000	30000008
ADD,R0,R0,R0	000000 0000 0000 0000 00000000000000	00000000
AND R4,R0,R4	000000 0100 0000 0100 00000000000001	01010001
AND R4,R0,R8	000000 0100 0000 1000 10001000000000	01022200
LW R9,R2,0	000101 1001 0010 0000000000000000 00	16480000
SW R5,R1,3	000111 0101 0001 0000000000000011 00	1d44000c

Table 3-1: Test Progarm

3.2. The Register Values

The table below shows the initial values we set to the 16 general purpose register in the CPU, in order to test the instructions. (note: the values can be found the project's folder as data1.hex file).

Register	Value
R0	00000000
R1	00000001
R2	00000002
R3	00000000
R4	00000000
R5	00000006
R6	00000000
R7	00000000
R8	00000000
R9	00000000
R10	00000000
R11	00000000
R12	00000000
R13	00000000
R14	00000000
R15	00000000

Table 3-2: General Purposr Register Values

3.3. The Simulation Results

We manually entered the hex program and the register's data to the processor project folder in active-HDL. Then we wrote a test bench for the CPU and synchronize it with a clock. After that, we ran the simulation and we got the following waveform. From observing the waveform results, we notice that the mutli-cycle processor behaves correctly and we got the expected results.

Figure 3-1: CPU Simulation Results1

The figure below shows the values in the memory after the CPU finished executing the program. From the highlighted data, we realize that the instructions were done successfully.

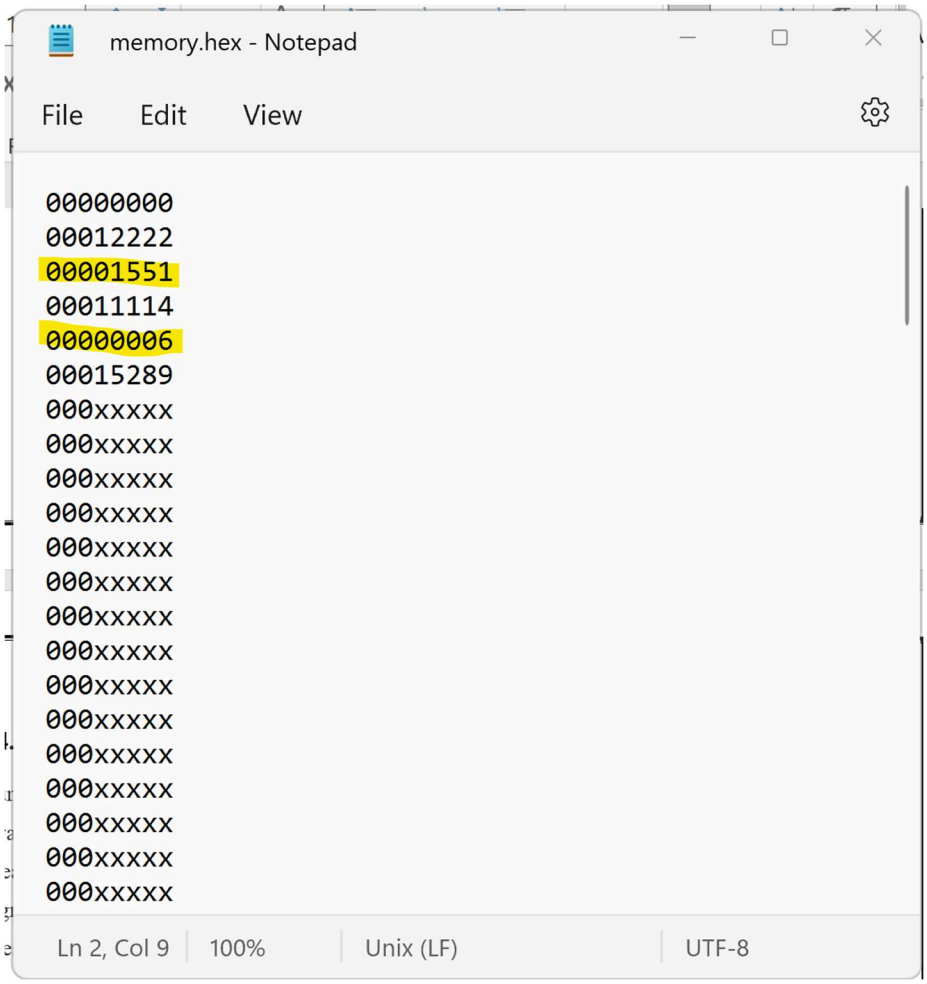

Figure 3-2: Memory Results

4. Conclusion

In summary, the project has successfully implemented a RISC multi-cycle processor, showcasing the various stages involved in its design, implementation, and testing. The collaborative efforts of the team were pivotal in creating and implementing a comprehensive processor design. Lastly, the designed processor shows promise for future enhancements, potentially evolving into a pipelined processor.

5. References

[1] [Online]. Available: https://electronicsdesk.com/risc-processor.html. [Accessed 28 1 2024].

[2] [Online]. Available: https://en.wikibooks.org/wiki/Microprocessor_Design/Multi_Cycle_Processors. [Accessed 28 1 2024].