



*Department of Electrical and Computer Engineering
ENCS4370 | Computer Architecture
The 2nd Project: Multicycle Processor Build and Design*

Prepared By:

Lana Batnij __ 1200308

Ansam Rihan __ 1200749

Ansam Osama __ 1201127

Instructor: Dr. Ayman Hroub and Dr. Aziz Qaruish

Section: 3&1

Date: 2nd/July/2023

- ***Abstract:***

This project aims to use Verilog to develop and verify a small RISC CPU. The CPU has a 32-bit instruction size, 32 general-purpose registers, a program counter (PC), a return address control stack, and a stack pointer (SP). Four instruction types are supported (R-type, I-type, J-type, and S-type), and an ALU with a "zero" signal signifies a zero result. There are separate data and instruction memory. A Datapath and a control path are included in the RTL design, and three design alternatives are available: single-cycle, multi-cycle, and 5-stage pipelined processors, and the team decided to build a multi-processor. Verification is carried out using a testbench and code sequences executed in the designed processor.

Table of Contents

• Abstract:	II
• Table of Figures:	IV
• List Of Tables:.....	V
• Project Discussion:.....	VI
○ Type of Processor:.....	VI
○ Design and implementation:	VII
○ Components:	VII
1) PC:	VII
2) Extenders:	VIII
3) Register File (decode Stage):.....	X
4) Control Unit:.....	XII
5) ALU:.....	XIII
6) Data Memory:.....	XIV
• Testing and results:	XVI
➤ Control Unit	XVI
➤ Data Memory	XVII
➤ Memory Instruction	XVIII
➤ Register File	XIX
➤ Extender	XXI
• Teamwork:	XXIV
• Conclusion:	XXV
• Appendix:.....	XXVI

• ***Table of Figures:***

Figure 1: Data-Path	VII
Figure 2: PC	VIII
Figure 3: Extenders(A)	IX
Figure 4; Extenders(B).....	IX
Figure 5: R-Type Instruction Format	X
Figure 6: I-type Instruction Format.....	X
Figure 7: J-Type Instruction Format	X
Figure 8: S-Type Instruction Format.....	X
Figure 9: Register File	XII
Figure 10: Data Memory.....	XV
Figure 11: Control Unit testbench.....	XVII
Figure 12: Control Unit testing output.....	XVII
Figure 13: Data Memory testbench.....	XVIII
Figure 14: Data Memory testing output.....	XVIII
Figure 15: Memory Instruction testbench.....	XIX
Figure 16: Memory Instruction testing output.....	XIX
Figure 17: Register File testbench	XX
Figure 18: Register File testing output.....	XX
Figure 19:Extender 5-32 testbench	XXI
Figure 20: Extender 5-32 testing output	XXI
Figure 21: Extender 24-32 testbench	XXII
Figure 22: Extender 24-32 testing output	XXII
Figure 23: Extender 14-32 testbench	XXIII
Figure 24: Extender 14-32 testing output	XXIII

- ***List Of Tables:***

Table 1: Control signals Truth Table.....XII

Table 2: Table of ALU operations XIV

- ***Project Discussion:***

- ***Type of Processor:***

The Team chose to Implement and design a multicycle Processor, it is been chosen since it is easier than pipelined in implementation. In a multicycle processor, the execution of an instruction is divided into multiple stages, each taking one clock cycle.

The following are the major concepts and characteristics of a multi-cycle processor:

- Instruction Fetch (IF): The CPU obtains the next instruction from memory at this stage. The program counter (PC) keeps track of the next instruction's memory address, and the instruction is loaded into an instruction register.
- Instruction Decode (ID): The taken instruction is decoded at this stage. The CPU determines the instruction type and extracts the required operands and control signals.
- Execution (EX): This stage is responsible for carrying out the instruction. It can include mathematical and logical operations like addition, subtraction, and comparison and memory operations like loading and storing data.
- Memory Access (MEM): Memory-related actions are handled by this stage. If the instruction involves memory access, this step is in charge of reading from or writing to memory.
- Write Back (WB): At this stage, the execution results are written back to the proper registers. It writes the computed values to the register file or other internal storage components.
- Control Signals: The control unit of the multi-cycle processor generates control signals that coordinate the different stages and enable the flow of instructions through the pipeline. These control signals dictate the operation of various components, such as multiplexers, ALUs, and memory units.
- Each level of the multi-cycle processor normally requires one or more clock cycles to complete. When the clock signal prompts the next cycle, the processor moves to the next level. A multi-cycle processor can make better use of hardware resources and enhance overall performance by breaking instruction processing into stages.

-

- It's worth noting that multi-cycle processor designs vary, and some processors may have additional stages or alternative names for the phases. Furthermore, more powerful processors may use pipelining or superscalar techniques to improve speed.

○ *Design and implementation:*

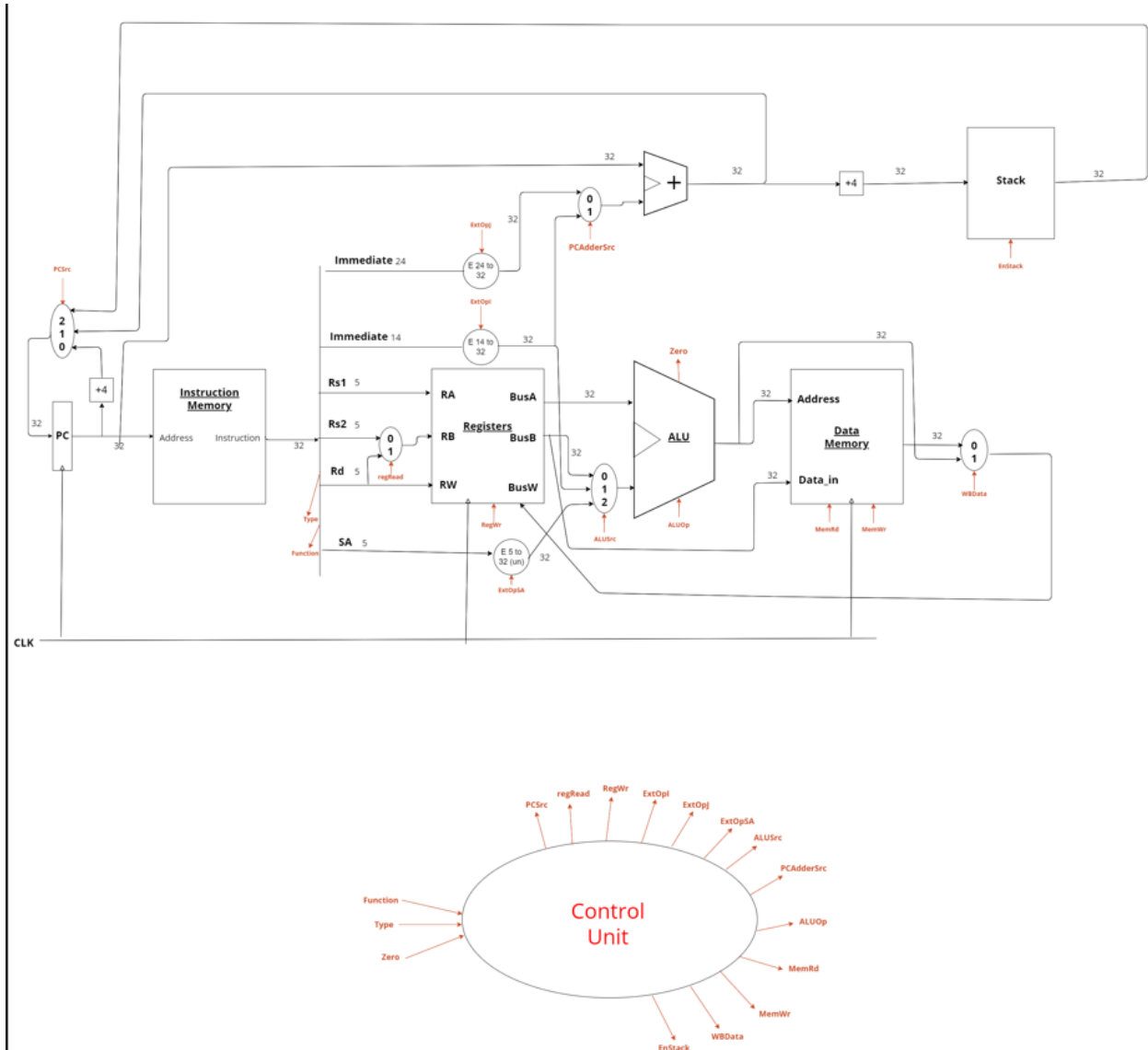


Figure 1: Data-Path

○ *Components:*

1) *PC:*

The PC is a register that stores the memory address of the following instruction to be fetched and executed. It is used to control the flow of the program by indicating the address of the

instruction to be fetched from the instruction memory. The instruction fetch stage increments the PC to point to the next sequential instruction in memory, unless a branch or jump instruction affects its value.

and its size is not specifically stated in the information provided, but because the instruction size is 32 bits, it may be deduced that the PC size would also be 32 bits to handle the memory locations.

For the R-type, the I-Type, and the S-type, the $PC = PC + 4$, and that's normal, while when the instruction is BEQ The PC is $PC + 4 * \text{immediate}$ (14-bit), and For J instruction It is $PC + \text{immediate}$ (24 bits).

A multiplexer is used here with the control signal PCSrc to decide what is the next address of the PC.

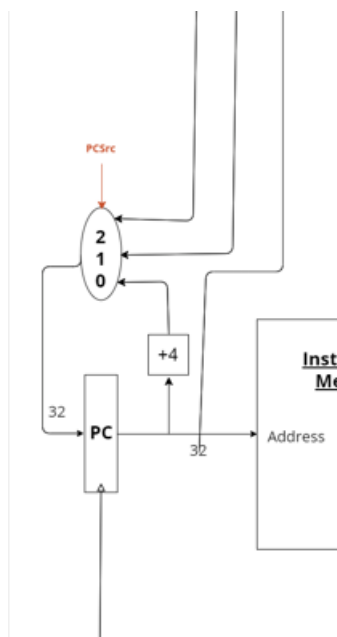


Figure 2: PC

2) Extenders:

In our Design, we have three extenders, but extenders in processor architecture are hardware components or units that are responsible for extending or altering the length or structure of data in the processor. These units are frequently employed to efficiently handle a variety of instructions or data formats. Extenders can conduct operations such as sign extension, zero extension, and format conversion to ensure data representation compatibility and consistency.

They are critical in supporting a wide range of instruction sets and data kinds, allowing the processor to successfully handle multiple operations and instructions with varying operand sizes and formats.

The first extender is used to convert the 14-bit unsigned Immediate to a 32-bit one which is used This immediate is going to be unsigned for the Logical operation but signed for the address calculation ones when the I-type instructions are in process and that to calculate the address or in arithmetical or logical operation, while the other extender is used to convert the 24-bit signed immediate into a 32-bit signed one that is used in the calculation of the address of the next instruction when we dealing with the jump instructions. The third Extender is to turn the 5-bit unsigned immediate into a 32-bit one so it can be used as a shift amount.

Control signals were used with the extender which is the ExtOpj, ExtOpi, and ExtSa all of them have a 1-bit to represent if the instruction needs to use this Extender or not.

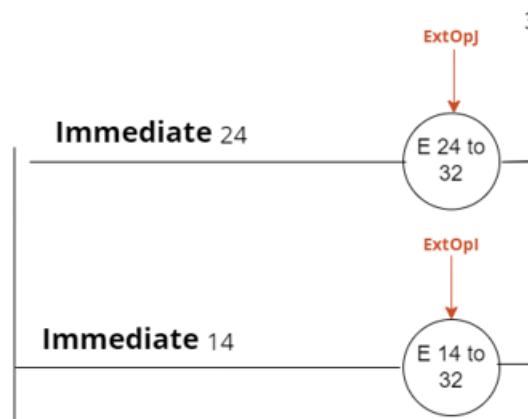


Figure 3: Extenders(A)

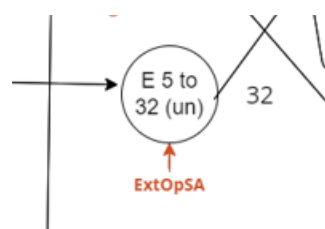


Figure 4; Extenders(B)

3) Register File (decode Stage):

The fetched instruction is decoded in a processor's decode stage to discover its type, operands, and the exact operation it represents. This step is in charge of extracting information from the instruction and generating the required control signals for the following stages.

The register file is an important component in the decoding stage. A register file is a hardware component that contains a collection of general-purpose registers (GPRs). These registers store data that instructions can access and manipulate during program execution. A register file is often made up of many storage parts. The register file is accessed at the decode stage to read the values from the specified source registers mentioned in the specification of the instructions. The instruction's register addresses or indices are utilized to choose the appropriate registers from the register file. The values received from the register file are then sent on to the next level for processing, such as arithmetic operations, memory accesses, or control flow choices. By the End of the Decoding Stage, the processor Can tell what type of Instructions that is going to deal with and it parameters that will be used in progress.

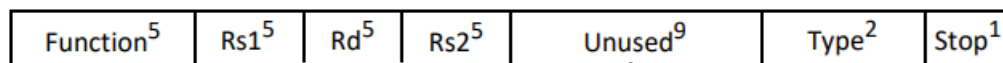


Figure 5: R-Type Instruction Format

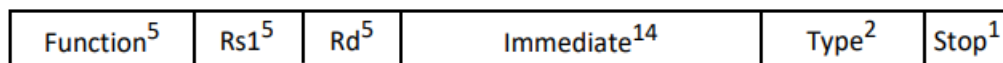


Figure 6: I-type Instruction Format

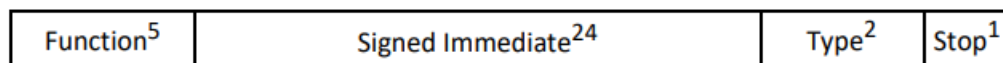


Figure 7: J-Type Instruction Format

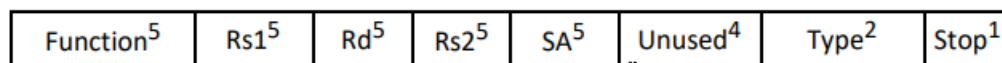


Figure 8: S-Type Instruction Format

In the Seen Format, there are some common fields which are:

The 2-bit instruction type (00: R-Type, 01: J-Type, 10: I-Type, 11: S-Type) and 5-bit function, which determines the precise operation of the instruction, and stop a bit, which is the least

significant bit of each instruction binary format and is used to identify the conclusion of a function code block. In other words, if the value of this stop bit is "1," it implies that this is the function's final instruction, and the execution control should return to the return address stored at the top of the control stack.

The above figures Specify Each Instruction Format, the R-type Besides having a 5-bits For Function, 2-bits for type, and one for Stop it has a 5-bit for each RS1, RS2, and RD and those are the source registers and the destination one. While the I-type has 5 bits For RS1 which is the source register and another five for the RD which is the destination one, and it has a 14-bit for unsigned immediately for the logic instructions and signed for others. Meanwhile, the J-type has only 24-bits signed immediate beside the common fields, and lastly, the S-type has a 5-bit for each RS1, and RS2, which are the source registers, and five For RD which is the destination one, and there are another five bits that will be used for as a shift amount.

As we can note that there are 9 bits in R-type and 4 bits in S-type and this field is added to make all instructions have the same length which is 32-bit.

The register file outputs two values, busa, and busb, which are read from the designated source registers. Other components in the computer system can utilize these values to perform computations or other activities. To write data back to the destination register, the busw signal is used. The Regwr is a control signal that allows writing to the register file. The register file is enabled to write data back to the destination register when the Regwr signal is high.

Multiplexers are used with the Register file the First one is to decide what value should be on BusB, the value of RS2 or RD and it is used with a control signal regRead.

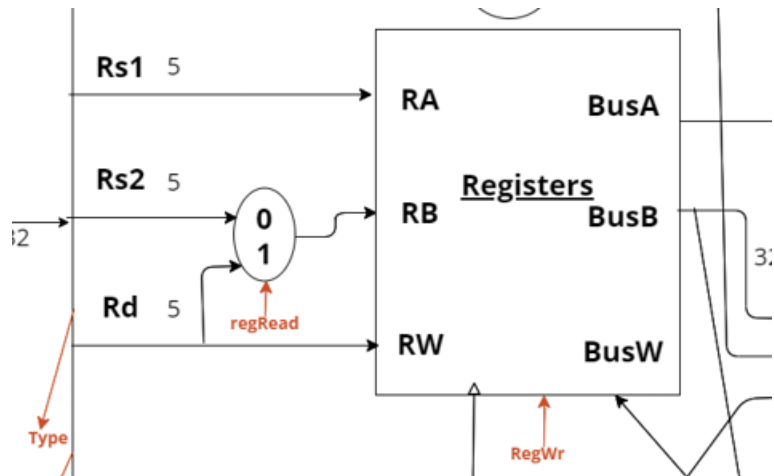


Figure 9: Register File

4) Control Unit:

The control unit (CU) is responsible for generating signals that advise the processor components on how to respond to a particular instruction. The signals are created based on the instruction's opcode. A binary decoder using the opcode as its input is utilized to do this. To determine the equations for all of the required signals, a truth table is developed.

Name	Type	Function	regRead	RegWr	ExtOpJ	ExtOpI	ExtOpSA	ALUSrc	PCAddrSrc	ALUOp	MemRd	MemWr	WBData	EnStack	PCSrc
R-Type Instructions															
AND	2'b00	5'b00000	0	1	x	x	x	0	x	AND (0)	0	0	1	0	0
ADD	2'b00	5'b00001	0	1	x	x	x	0	x	ADD (1)	0	0	1	0	0
SUB	2'b00	5'b00010	0	1	x	x	x	0	x	SUB (2)	0	0	1	0	0
CMP	2'b00	5'b00011	0	0	x	x	x	0	x	SUB (2)	0	0	1	0	0
I-Type Instructions															
ANDI	2'b10	5'b00000	x	1	x	0 (un)	x	1	x	AND (0)	0	0	1	0	0
ADDI	2'b10	5'b00001	x	1	x	1 (s)	x	1	x	ADD (1)	0	0	1	0	0
LW	2'b10	5'b00010	x	1	x	1 (s)	x	1	x	ADD (1)	1	0	0	0	0
SW	2'b10	5'b00011	1	0	x	1 (s)	x	1	x	ADD (1)	0	1	x	0	0
BEQ	2'b10	5'b00100	1	0	x	1 (s)	x	0	if zero flag -> 1 else -> x	SUB (2)	0	0	x	0	if zero flag -> 1 else -> 0
J-Type Instructions															
J	2'b01	5'b00000	x	0	1 (s)	x	x	x	0	x	0	0	x	0	1
Jal	2'b01	5'b00001	x	0	1 (s)	x	x	x	0	x	0	0	x	1	1
S-Type Instructions															
SLL	2'b11	5'b00000	x	1	x	x	0 (un)	2	x	SLL (3)	0	0	1	0	0
SLR	2'b11	5'b00001	x	1	x	x	0 (un)	2	x	SLR (4)	0	0	1	0	0
SLLV	2'b11	5'b00010	0	1	x	x	x	0	x	SLL (3)	0	0	1	0	0
SLRV	2'b11	5'b00011	0	1	x	x	x	0	x	SLR (4)	0	0	1	0	0
								2 - bit	3 - bit						
															2 - bit if stop then 2

Table 1: Control signals Truth Table

Here are the Boolean equations for the control signals based on our control signals:

$read = 1$ (for all instructions)

$RegWr = 0$ (for R-Type instructions),

$RegWr = 1$ (for I-Type, S-Type, and J-Type instructions)

$ExtOpJ = 0$ (for R-Type instructions), $ExtOpJ = 1$ (for I-Type and S-Type instructions), $ExtOpJ = 0$ or 1 (for J-Type instructions)

$ExtOpI = 0$ or 1 (depending on the instruction)

$ExtOpSA = 0$ (for R-Type and J-Type instructions), $ExtOpSA = 2$ (for S-Type instructions)

$ALUSrc = 0$ (for R-Type, I-Type, and S-Type instructions), $ALUSrc = 1$ (for J-Type instructions)

$PCAdderSrc = 0$ (for all instructions)

$ALUOp = 1$ (for R-Type instructions), $ALUOp = 0$ (for I-Type and S-Type instructions), $ALUOp = 2$ (for J-Type instructions)

$MemRd = 0$ (for R-Type and I-Type instructions), $MemRd = 1$ (for LW instruction)

$MemWr = 0$ (for all instructions), $MemWr = 1$ (for SW instruction)

$WBData = 0$ (for all instructions)

$EnStack = 0$ (for all instructions)

$PCSrc = 0$ (for J-Type instructions), $PCSrc = 1$ (for BEQ instruction)

5) ALU:

ALU (Arithmetic Logic Unit) is a processor component that performs arithmetic and logical operations on binary data. It is in charge of carrying out the many computations and comparisons required by instructions during program execution.

In this Design, the ALU can perform about 15 Operations that are divided into four types, so the following table shows the truth table for the ALU we've built:

No.	Instr	Meaning	Function Value
R-Type Instructions			
1	AND	$\text{Reg(Rd)} = \text{Reg(Rs1)} \& \text{Reg(Rs2)}$	00000
2	ADD	$\text{Reg(Rd)} = \text{Reg(Rs1)} + \text{Reg(Rs2)}$	00001
3	SUB	$\text{Reg(Rd)} = \text{Reg(Rs1)} - \text{Reg(Rs2)}$	00010
4	CMP	zero-signal = $\text{Reg(Rs)} < \text{Reg(Rs2)}$	00011
I-Type Instructions			
5	ANDI	$\text{Reg(Rd)} = \text{Reg(Rs1)} \& \text{Immediate}^{14}$	00000
6	ADDI	$\text{Reg(Rd)} = \text{Reg(Rs1)} + \text{Immediate}^{14}$	00001
7	LW	$\text{Reg(Rd)} = \text{Mem}(\text{Reg(Rs1)} + \text{Imm}^{14})$	00010
8	SW	$\text{Mem}(\text{Reg(Rs1)} + \text{Imm}^{14}) = \text{Reg(Rd)}$	00011
9	BEQ	Branch if ($\text{Reg(Rs1)} == \text{Reg(Rd)}$)	00100
J-Type Instructions			
10	J	$\text{PC} = \text{PC} + \text{Immediate}^{24}$	00000
11	JAL	$\text{PC} = \text{PC} + \text{Immediate}^{24}$ Stack.Push (PC + 4)	00001
S-Type Instructions			
12	SLL	$\text{Reg(Rd)} = \text{Reg(Rs1)} \ll \text{SA}^5$	00000
13	SLR	$\text{Reg(Rd)} = \text{Reg(Rs1)} \gg \text{SA}^5$	00001
14	SLLV	$\text{Reg(Rd)} = \text{Reg(Rs1)} \ll \text{Reg(Rs2)}$	00010
15	SLRV	$\text{Reg(Rd)} = \text{Reg(Rs1)} \gg \text{Reg(Rs2)}$	00011

Table 2: Table of ALU operations

6) Data Memory:

The processor interfaces with the data memory component during this step to perform memory operations. Load instructions read from memory and store them in registers, whereas store instructions read from registers and write them to memory.

The data memory stage is in charge of accessing memory locations and transferring data between the CPU and memory. It guarantees that the required data is read from or written to memory appropriately during the execution of load and store instructions.

The data memory outputs the contents of the memory location indicated by the address input when the MemRd signal is asserted. The data memory writes the data input to the memory location indicated by the address input when the MemWr signal is asserted. The write operation is carried out on the clock signal's rising edge.

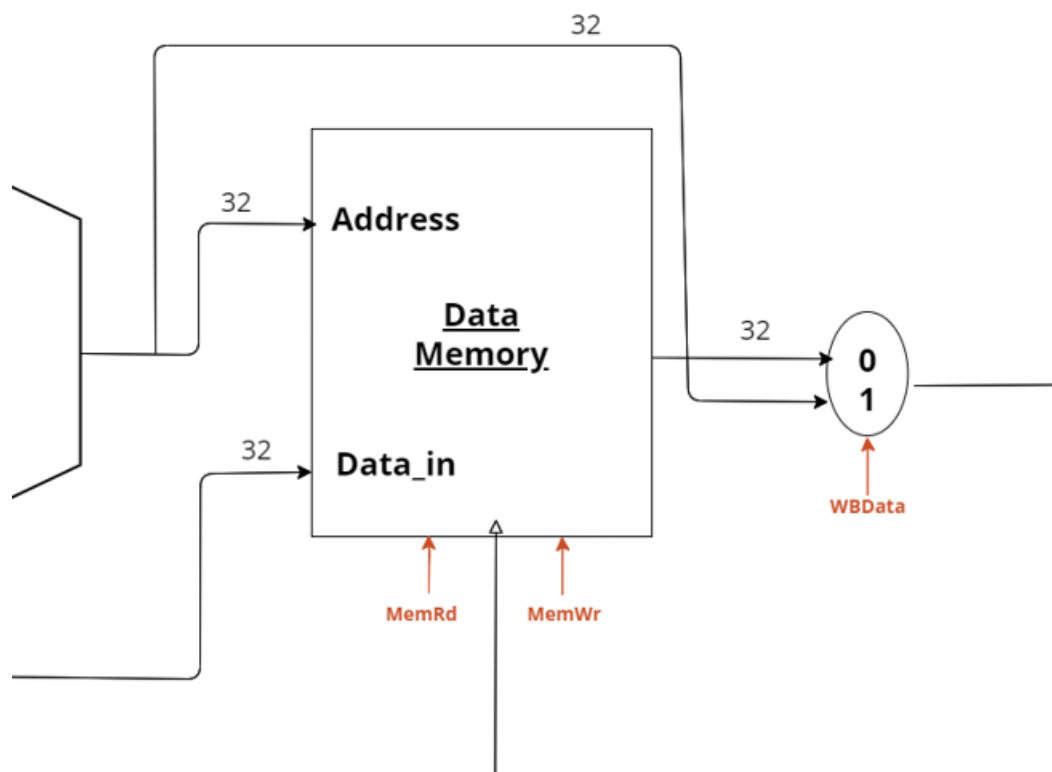


Figure 10: Data Memory

- **Testing and results:**

Each functional component underwent thorough testing to ensure the accuracy of its outputs. For each module, a testbench was written. The following are the parts carried on in the testing part:

➤ **Control Unit**

the test bench code for the control_unit module performs two test cases. In the first test case, the inputs remain constant for 10-time units, with stops set to 0, Function set to 00000, Type set to 00, and Zero set to 0. The outputs of the control_unit module are evaluated based on the specified conditions in the always block. The specific values of the outputs are not mentioned in the test bench code but are determined by the combination of the Type and Function inputs. In the second test case, after a delay of 30 time units, the inputs are updated. stop is set to 1, Function is set to 00001, Type is set to 01, and zero is set to 1. These new input values are maintained for 30-time units, and the outputs of the control_unit module are evaluated based on these updated inputs.

After the completion of the second test case, the simulation ends, and the results can be examined as shown in Fig [12].

```
module control_unit_tb;
    reg clk;
    reg stop;
    reg [4:0] Function;
    reg [1:0] Type;
    reg zero;
    wire regRead;
    wire regWr;
    wire ExtOpJ;
    wire ExtOpI;
    wire ExtOpSA;
    wire [1:0] ALUSrc;
    wire PCAdderSrc;
    wire [2:0] ALUOp;
    wire MemRd;
    wire MemWr;
    wire WBData;
    wire EnStack;
    wire [1:0] PCSrc;

    control_unit cu(clk, stop, Function, Type, zero, regRead, regWr, ExtOpJ, ExtOpI, ExtOpSA, ALUSrc, PCAdderSrc, ALUOp, MemRd, MemWr, WBData, EnStack, PCSrc);
```



```

initial begin
    clk = ~clk;
    stop = 0;
    Function = 5'b000000;
    Type = 2'b00;
    zero = 0;
    #10;
    clk = ~clk;
    stop = 1;
    Function = 5'b000001;
    Type = 2'b01;
    zero = 1;
    #30;
    clk = ~clk;
    $finish;
end
endmodule

```

Figure 11: Control Unit testbench

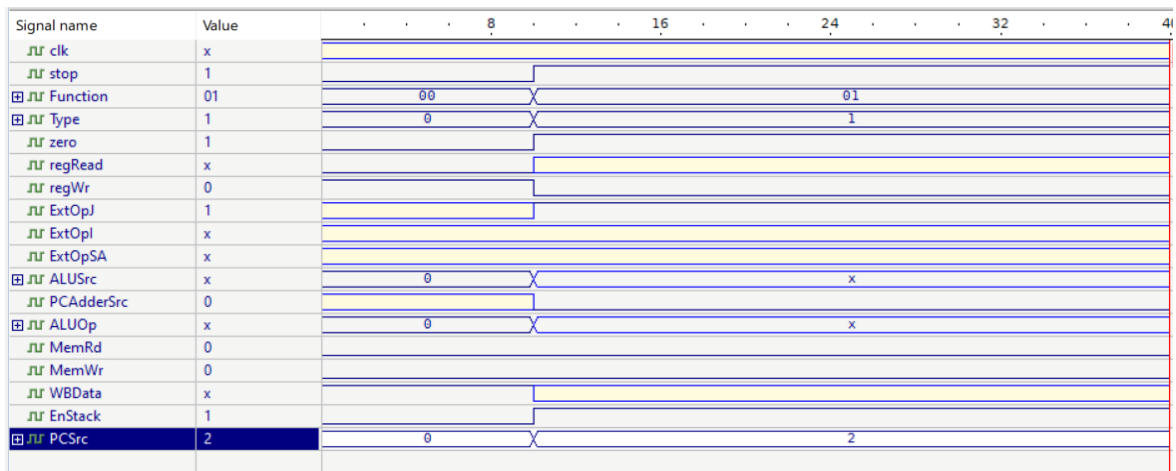


Figure 12: Control Unit testing output

➤ Data Memory

This testbench initializes the signals, performs a memory write operation by writing the value 12 to memory location 2, waits for some time, and then performs a memory read operation from memory location 2. The simulation then ends after a delay of #20.

The results of the simulation can be observed by monitoring the data_out signal, which represents the output data read from the memory. In Fig [14], the output didn't observe because the memory is empty.

```

325
326 module data_memory_testbench();
327     reg clk, mem_rd, mem_wr;
328     reg [31:0] address, data_in, data_out;
329     data_memory dm(clk, address, data_in, mem_rd, mem_wr, data_out);
330     initial begin
331         clk = 0; mem_rd = 0; mem_wr = 0;
332         address = 0; data_in = 0;
333         #5
334         clk = ~clk;
335         mem_wr = 1;
336         mem_rd = 0;
337         data_in = 12;
338         address = 2;
339         #20
340         clk = ~clk;
341         #5
342         clk = ~clk;
343         mem_wr = 0;
344         mem_rd = 1;
345         address = 2;
346
347         #20
348
349         $finish;
350     end
351 endmodule
352

```

Figure 13: Data Memory testbench

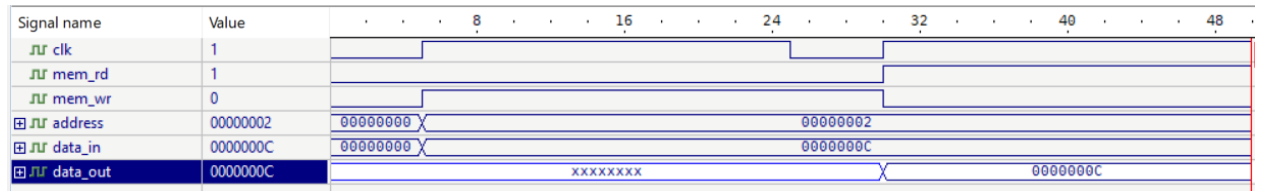


Figure 14: Data Memory testing output

➤ Memory Instruction

The test bench for the instruction_memory module which is shown in Fig [15], includes a clock signal (CLK), an address input (address), and an instruction output (instruction) to interface with the instruction_memory module. It instantiates the instruction_memory module and sets up a simulation using an initial block. In the test bench, the address is initially set to 0. After a delay of 5-time units (#5), the clock signal is toggled ($\text{clk} = \sim\text{clk}$), and the address is changed to 2. Another delay of 20-time units (#20) is added, and then the clock signal is toggled again.

The result of the test bench will depend on the contents of the memory array (mem) and the values assigned to the address input. To get the results for our code, the memory array should be populated with actual instructions before running the test bench.

```

module instruction_memory_tb;
    reg clk;
    reg [31:0] address;
    wire [31:0] instruction;

    instruction_memory im(clk, address, instruction);

    initial begin
        address = 0;

        #5
        clk = ~clk;
        address = 2;

        #20
        clk = ~clk;

    end
endmodule

```

Figure 15: Memory Instruction testbench




Signal name	Value	48121620																
 clk	x																	
 address	00000002	00000000								X	00000002							
 instruction	xxxxxxxx	xxxxxxxx																

Figure 16: Memory Instruction testing output

➤ **Register File**

The initial values for the inputs are set, including a toggling of the clock signal (CLK) and various values for the control and data inputs. After a delay of 10-time units (#10), the clock signal is toggled again, the register write control signal is activated (regWr = 1), and a new value is assigned to the write data bus (BusW). Another delay of 10-time units

is added, and then the clock signal is toggled for the final time. The simulation is then terminated using the \$finish system task.

The result of the test bench will depend on the behavior of the register_file module in response to the given input values. Specifically, it will show the values read from the register file (BusA and BusB) after the specified delays and modifications to the control and data inputs. Since the code does not include any output display or assertion statements, the exact values of BusA and BusB cannot be determined, and that is what happens in Fig [18].

```

200
201 module register_file_tb;
202     reg clk;
203     reg regWr;
204     reg [4:0] RA;
205     reg [4:0] RB;
206     reg [4:0] RW;
207     reg [31:0] BusW;
208     wire [31:0] BusA;
209     wire [31:0] BusB;
210
211     register_file rf(clk, regWr, RA, RB, RW, BusW, BusA, BusB);
212
213     initial begin
214         clk = ~clk;
215         regWr = 0;
216         RA = 2;
217         RB = 3;
218         RW = 4;
219         BusW = 32'hABCDEFFF;
220         #10;
221         clk = ~clk;
222         regWr = 1;
223         BusW = 32'h12345678;
224         #10;
225         clk = ~clk;
226         $finish;
227     end

```

Figure 17: Register File testbench

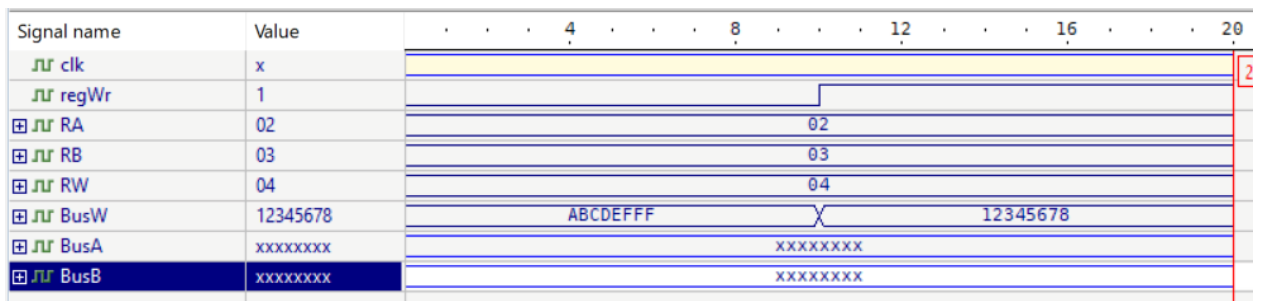


Figure 18: Register File testing output

➤ Extender

The `extender14to32_tb`, `extender24to32_tb`, and `extender5to32_tb` modules are testbench modules that instantiate the respective extender modules and provide test cases for simulation. The test benches set the input signals of the modules and monitor the output signals during simulation. Test cases are provided to verify the functionality of each module.

```

159 module extender5to32_tb();
160     reg ExtOpSA ,clk;
161     reg [4:0] input_5bits;
162     wire [31:0] output_32bits;
163
164     extender5to32 extender(clk,ExtOpSA, input_5bits, output_32bits);
165
166     initial begin
167         $monitor("Input: %b, Output: %b", input_5bits, output_32bits);
168
169         // Test case 1: ExtOpSA = 0, input_5bits = 5'b01101
170         ExtOpSA = 0;
171         input_5bits = 5'b01101;
172         clk = ~clk;
173         #5;
174
175         // Test case 2: ExtOpSA = 1, input_5bits = 5'b10100
176         ExtOpSA = 1;
177         input_5bits = 5'b10100;
178         clk = ~clk;
179         #10;
180
181         $finish;
182     end
183 endmodule
184

```

Figure 19:Extender 5-32 testbench


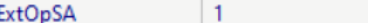

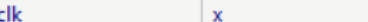




Signal name	Value	4	8	12
 ExtOpSA	1			
 clk	x			
  input_5bits	14	0D	X	14
  output_32bits	FFFFFFF4	0000000D	X	FFFFFFF4

Figure 20: Extender 5-32 testing output

```

module extender24to32_tb();
    reg ExtOpJ , clk;
    reg [23:0] input_24bits;
    wire [31:0] output_32bits;

    extender24to32 extender(clk, ExtOpJ, input_24bits, output_32bits);

    initial begin
        $monitor("Input: %b, Output: %b", input_24bits, output_32bits);

        // Test case 1: ExtOpJ = 0, input_24bits = 24'b00110011001100110011
        ExtOpJ = 0;
        input_24bits = 24'b00110011001100110011;
        clk = ~clk;
        #10;

        // Test case 2: ExtOpJ = 1, input_24bits = 24'b11001100110011001100
        clk = ~clk;
        ExtOpJ = 1;
        input_24bits = 24'b11001100110011001100;
        #10;

        $finish;
    end
endmodule

```

Figure 21: Extender 24-32 testbench

Signal name	Value	4	8	12	16	20	24
ExtOpJ	1						
clk	x						
input_24bits	CCCCCC	333333					
output_32bits	FFCCCC	00333333					

Figure 22: Extender 24-32 testing output

DA

0 1

- ***Teamwork:***

The members of our group engaged in a project by first brainstorming and developing ideas for the design of each component. We prepared a control unit signals table after we agreed on the design. The PC, IF/ID, and register file were then created and tested by one team member, while the Decode Condition, sub-Condition, add Condition, and ALU control unit were built and tested by another. The final team member constructed the ALU unit, Control unit, and extenders. Once all units were operational, we collaborated to build the data flow and fully test it to verify it satisfied all criteria. Then we collaborated on writing the code and testing the unit.

- ***Conclusion:***

Developing a Multi-cycle CPU with Verilog and the Active-HDL tool was a tough but gratifying project that helped me obtain a thorough understanding of digital design principles. The project entailed creating the data pipeline, components, and finite state machine required for the CPU to execute instructions, such as the control unit, register file, ALU, and data memory. To obtain signal equations, truth tables were employed, and Verilog code was written for each component. Overall, the project provided a great opportunity to obtain hands-on experience in digital design and Verilog programming, ending in a workable one-cycle CPU design.

- *Appendix:*

Our Code File will submit with the report.