Faculty of Engineering and Technology

Electrical and Computer Engineering Department

ENCS4370

Computer Architecture

Project #2

Name	**Number**
Mazen Batrawi	1190102
Shahd AbuDaghash	1191448

Instructor: Dr. Aziz Qaroush

Section: 2

Date: 15-02-2023

Abstract

The objective of this project is to design and test a Single-Cycle RISC processor using Verilog HDL. The project is built using Active-HDL – Student Edition. The specifications of the processor were defined in the Instruction Set Architecture (ISA) provided. These specifications were analyzed thoroughly before starting with the data path design and implementation. Finally, the resulting data path was tested by writing a sample code to test the instructions.

Table of Contents

Table of Contents

Table of Contents

1. Design and Implementation

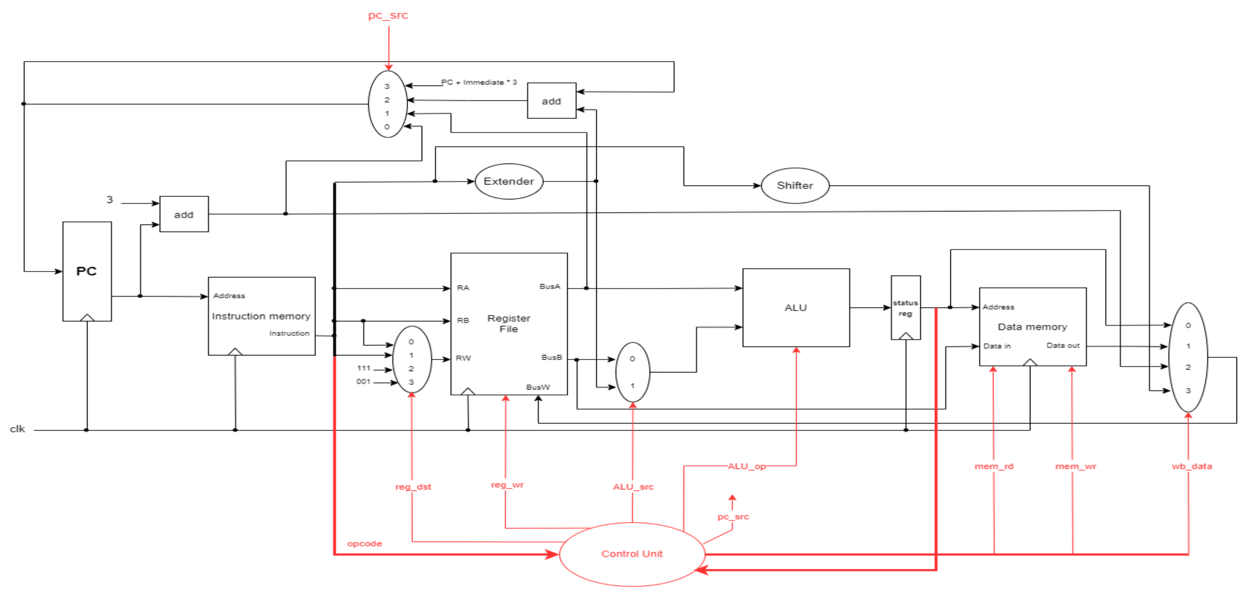


Figure 1: Data path

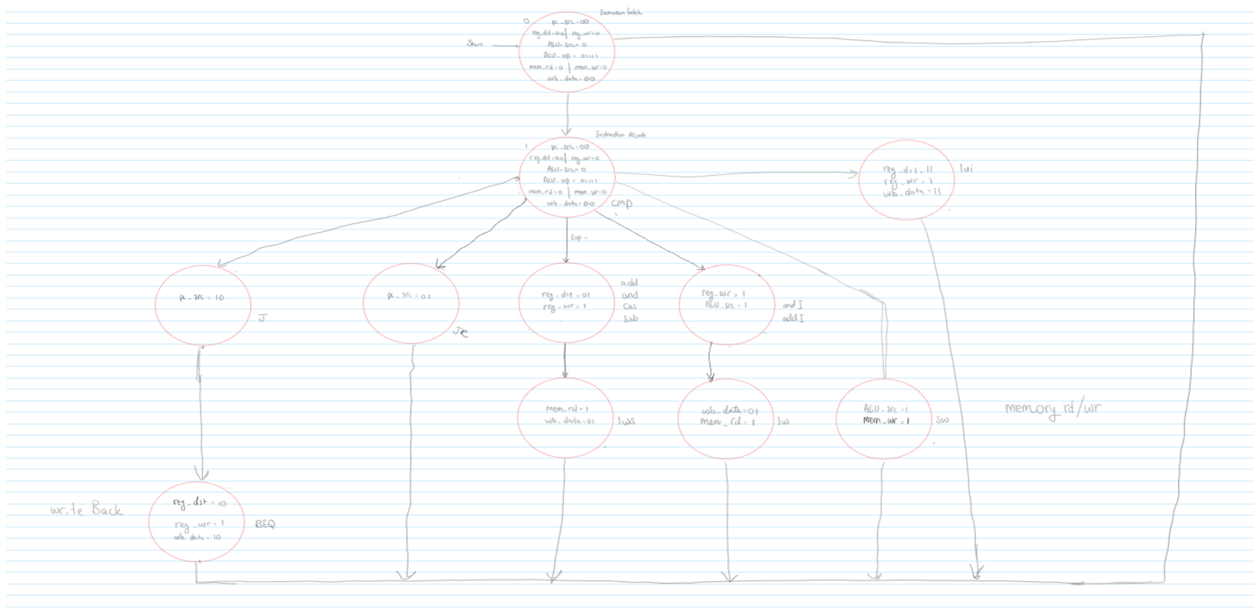


Figure 2: Finite state machine

This is the finite state machine of the single cycle CPU we designed.

A single cycle data path CPU is a type of central processing unit (CPU) design where each instruction is executed in a single clock cycle. In this design, all of the instructions pass through the same set of hardware components, making it a simple and easy-to-understand architecture. Here is a detailed description of the data path, its components, and control:

- Instruction Fetch: The instruction fetch unit is responsible for fetching instructions from memory. The program counter (PC) holds the address of the next instruction to be fetched. The PC value is sent to the memory address register, and the instruction is fetched from memory and stored in the instruction register.
- Instruction Decode: The instruction decoder is responsible for decoding the instruction fetched from memory. It identifies the opcode and determines the instruction's operands.
- Register File: The register file stores the data that the CPU is currently working with. It contains a set of registers, each of which holds a 24-bit data word. The register file is responsible for reading and writing data to and from the registers.
- ALU: The Arithmetic Logic Unit (ALU) performs arithmetic and logical operations on data. It takes two inputs, performs an operation, and outputs the result. The ALU is responsible for performing operations such as addition, subtraction, AND, SUB, and ADD.
- Memory Access: The memory access unit is responsible for reading or writing data to or from memory. The memory address is calculated by adding the contents of a register to an immediate value or the contents of another register. The memory data is stored in the data register.
- Write Back: The write back unit is responsible for writing data back to the register file. The result of an ALU operation or data read from memory is written to a register.
- Control Unit: The control unit is responsible for generating the control signals that coordinate the data path components. It receives the instruction opcode from the instruction decoder and generates the signals needed to control the operation of the ALU, register file, and memory access units.

In a single cycle data path CPU, all of the above components work together to execute a single instruction in one clock cycle. The control unit generates the necessary control signals to direct the flow of data through the data path components. The execution of each instruction is completed before the next instruction is fetched and executed. This design is simple and efficient but may have limitations in terms of performance and complexity for modern computing needs.

1.1.PC

The value of the Program Counter (PC), which indicates the memory address of the next instruction to be executed, is determined by this unit. The unit takes into account certain generated operations, such as BEQ, J, JAL, and LUI.

- The next address for normal operations is PC + 3.
- The next address for J (jump) operation PC + imm^{17}.
- The next address for BEQ (Branch if equal) operation is PC + 3 * imm^{10}.
- The next address for JAL (Jump and link) operation is PC + imm^{17}, and the return address is stored in R7, which is PC + 3.
- The next address for JR (Jump register) operation is PC = Reg(rs).

A multiplexer was used with it's control signal (pc_src) to determine the next address of the PC.

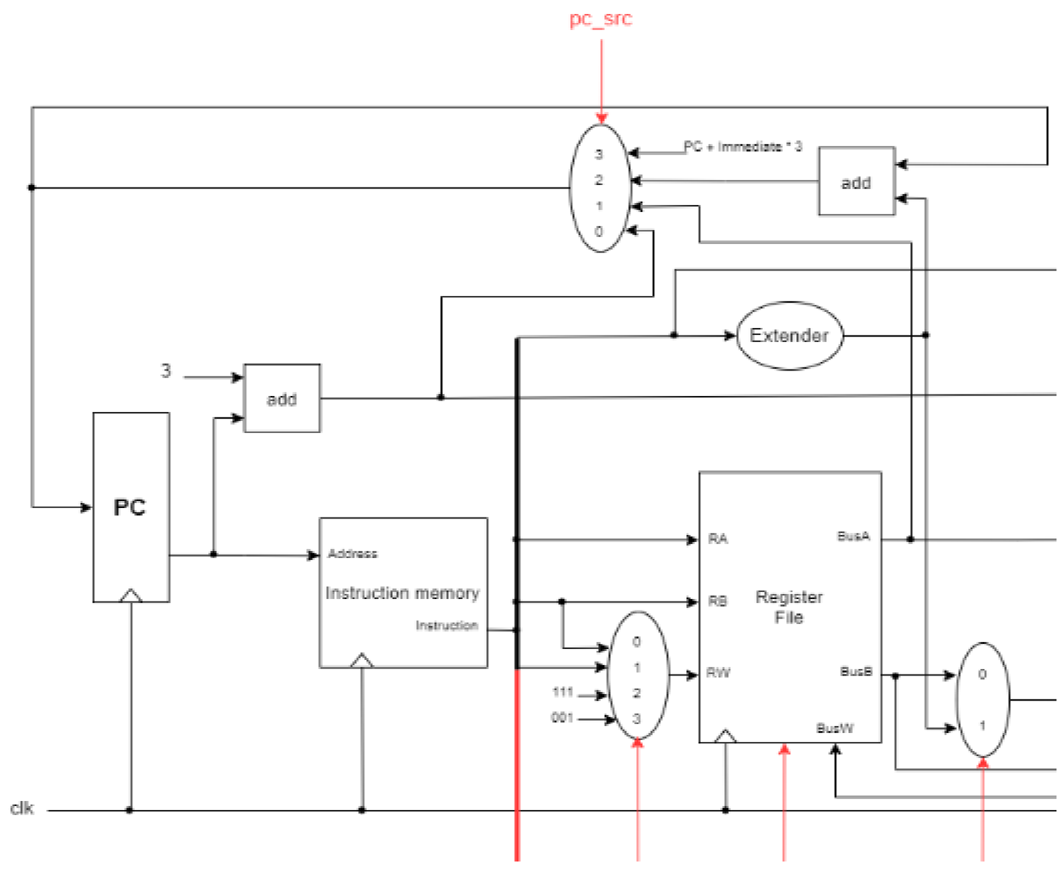


Figure 3: PC

1.2. 10-bit and 17-bit extender

An extender is a hardware component in a computer system that extends the number of bits in a binary value. In particular, a 10-bit or 17-bit extender takes an opcode and a 10-bit or 17-bit input and converts it to a 24-bit value depending on the opcode.

The purpose of an extender is to expand the range of values that can be represented by a binary value. For example, a 10-bit value can represent numbers between 0 and 1023, while a 17-bit value can represent numbers between 0 and 131071. By using an extender, these ranges can be extended to 24 bits, allowing for larger values to be represented.

The extender works by taking the input value and checking the opcode to determine the sign of the value. If the opcode indicates that the value is negative, the extender will sign-extend the value by filling the upper bits with 1's. If the value is positive, the extender will simply fill the upper bits with 0's.

Once the value has been sign-extended, the extender combines it with the opcode to create a 24-bit value. The resulting 24-bit value can then be used by other components in the computer system to perform calculations or other operations.

1.3. Decode Stage

The decode stage and instruction memory are critical components of a computer system that work together to interpret and execute instructions.

The decode stage is responsible for decoding the instruction that has been fetched from memory in the previous stage of the fetch-execute cycle. The decode stage receives the instruction from the instruction memory, which is a hardware component that holds the instruction currently being executed. The instruction is then decoded into its constituent parts, such as the opcode and any operands, which are the values on which the operation specified by the opcode will act.

Cond^2	Op^5	SF^1	Rd^3	Rs^3	Rt^3	Unused^7

Figure 4: R-type Instruction

Cond^2	Op^5	SF^1	Rt^3	Rs^3	Immediate^{10}

Figure 5: I-type Instruction

Cond^2	Op^5	Immediate^{17}

Figure 6: J-type Instruction

Figure 4 shows the instruction of R-type. The upper 2 bits hold the condition, the next 5 bits hold the opcode for the ALU operation, the next bit holds the SF value, for updating the zero flag. The next 9 bits contain the 3 registers rd, rs, and rt, each one of them is 3 bits. The last 7 bits are unused.

Figure 5 shows the instruction of I-type. The upper 2 bits hold the condition, the next 5 bits hold the opcode for the ALU operation, the next bit holds the SF value, for updating the zero flag. The next 6 bits contain the 3 registers rt and rs, each one of them is 3 bits. The last 10 bits are for the immediate value.

Figure 6 shows the instruction if J-type. The upper 2 bits hold the condition, the next 5 bits hold the opcode for the ALU operation, the next 17 bits hold the jump address.

1.4. Register File

The register file is a key component of a computer system that stores data in registers and allows for fast access to this data during execution. The register file takes in three 3-bit inputs, which are used to select the registers to read from, as well as the busw, the clock, and the regwr signals.

The first two inputs, rs and rt, select the two source registers to read data from, while the third input is used to select the destination register to write data back to. The destination register can be either rt, rd, R7, or R1, depending on the instruction being executed.

The busw signal is used to write data back to the destination register. The regwr signal is used to enable writing to the register file. When the regwr signal is high, the register file is enabled to write data back to the destination register.

The register file has two outputs, busa and busb, which provide the values read from the selected source registers. These values can be used by other components in the computer system to perform calculations or other operations.

1.5. Control Unit

The responsibility of the control unit (CU) is to generate signals that instruct the processor components on how to respond to a given instruction. The signals are generated based on the opcode of the instruction.

To accomplish this, a binary decoder is used with the opcode as its input. A truth table is constructed to determine the equations for all of the required signals.

OPCODE	reg_dst	reg_wr	pc_src	ALU_src	ALU_op	mem_rd	mem_wr	WB_data	Operation
00000	01	1	00	0	00000	0	0	00	AND
00001	01	1	00	0	00001	0	0	00	CAS
00010	01	1	00	0	00010	1	0	01	Lws
00011	01	1	00	0	00011	0	0	00	ADD
00100	01	1	00	0	00100	0	0	00	SUB
00101	xx	0	00	0	00101	0	0	00	CMP
00110	xx	0	01	x	xxxxx	0	0	xx	JR
00111	00	1	00	1	00111	0	0	00	ANDI
01000	00	1	00	1	01000	0	0	00	ADDI
01001	00	1	00	1	01001	1	0	01	Lw
01010	xx	0	00	1	01010	0	0	xx	Sw
01011	xx	0	10	0	01011	0	0	xx	BEQ
01100	xx	0	10	x	xxxxx	0	0	xx	J
01101	10	1	10	x	01111	0	0	xx	JAL
01110	11	1	00	x	xxxxx	0	0	11	LUI

Table 1: Control signals truth table.

Each signal was set in a case depending on the opcode.

Note: 01111 in ALU_op is the default in the case.

1.6. ALU

The ALU takes two 24-bit inputs along with condition, SF, opcode, and the status register to produce the 24-bit output for the next stage. The following table shows the truth table for the ALU we’ve built.

OPCODE	Name	Condition	SF	Operation
00000	AND	00	0	AND
00000	ANDEQ	01	X	AND
00000	ANDNE	10	X	AND
00001	CAS	00	0	MAX
00001	CASEQ	01	X	MAX
00001	CASNE	10	X	MAX
00010	Lws	00	0	ADD
00010	LwsEQ	01	X	ADD
00010	LwsNE	10	X	ADD
00011	ADD	00	0	ADD
00011	SUBSF	00	1	SUB
00011	ADDEQ	01	X	ADD
00011	ADDNE	10	X	ADD
00100	SUB	00	0	SUB
00100	SUBEQ	01	X	SUB
00100	SUBNE	10	X	SUB
00101	CMP	X	0	CMP
00110	JR	00	0	-
00110	JREQ	01	X	-
00110	JRNE	10	X	-
00111	ANDI	00	0	AND
00111	ANDIEQ	00	X	AND
00111	ANDINE	10	X	AND
01000	ADDI	00	0	AND
01000	SUBISF	00	1	SUB
01000	ADDIEQ	01	X	ADD
01000	ADDINE	10	X	ADD
01001	Lw	00	0	ADD
01001	LwEQ	01	X	ADD
01001	LwNE	10	X	ADD
01010	Sw	00	0	ADD
01010	SwEQ	01	X	ADD
01010	SwNE	10	X	ADD
01011	BEQ	X	0	COM
01100	J	00	0	-
01100	JEQ	01	X	-
01100	JNE	10	X	-
01101	JAL	00	0	-
01101	JALEQ	01	X	-

01101	JALNE	10	X	-
01110	LUI	00	0	-
01110	LUIEQ	01	X	-
01110	LUINE	19	X	-

Table 2: Truth table for ALU

1.7. Data Memory

The data memory is a component of a computer system that stores data in memory locations. It takes as inputs a read enable signal (mem_rd), a write enable signal (mem_wr), a 24-bit memory address, and a 24-bit data input. It produces a 24-bit output, which is the data stored at the specified memory address.

When the mem_rd signal is asserted, the data memory outputs the contents of the memory location specified by the address input. When the mem_wr signal is asserted, the data memory writes the data input to the memory location specified by the address input. The write operation is performed on the rising edge of the clock signal.

The data memory typically uses an array of memory cells to store data. The memory cells are organized into rows and columns, and each cell can store a single bit of data. To store 24 bits of data in a single memory location, multiple cells are used, with each cell storing a portion of the data.

2. Testing and results

The testing for each functional component was done to make sure the outputs are correct. For each module, a testbench was written. Following are the parts carried on in the testing part:

2.1. Extender

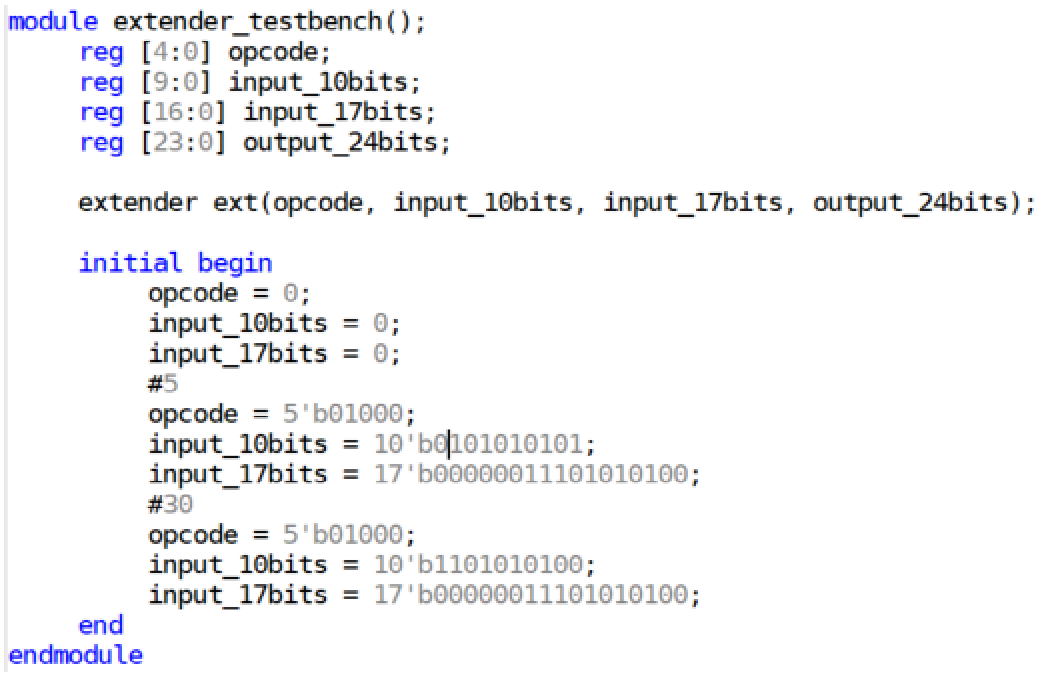
There are three inputs for this module: the opcode, the 10-bits immediate, and the 17-bits immediate. The opcode decides the type of extension. Since this circuit was tested on its own, the opcode didn't play a role in the testing.

The testbench for this circuit is shown in figure 7, and the resulting waveform is shown in figure 8.

An example from the waveform is when the 10-bits immediate input equals 10'b0101010101, the output in this case was 0000000000000000101010101. This is the expected result since extending the two's complement number copies the most significant bit (the sign) to the added bits, where in this case are zeros.

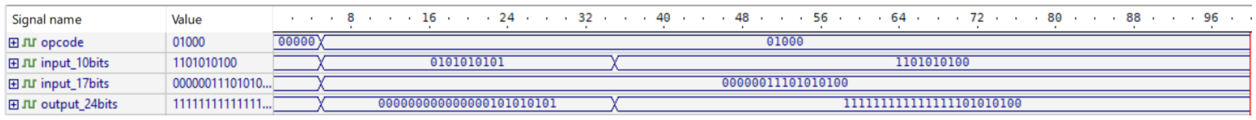
On the contrary, when the input was 1101010100, the output was 1111111111111111101010100. This is also the expected result following the same method of extension.

Similar results are done for the 17-bits input. When the 17-bits input is 00000011101010100, the output is 0000000000000011101010100. Flipping the most significant bit to 1 to have the input 10000011101010100 gives the result of 1111111110000011101010100. These results are shown in figure 9.



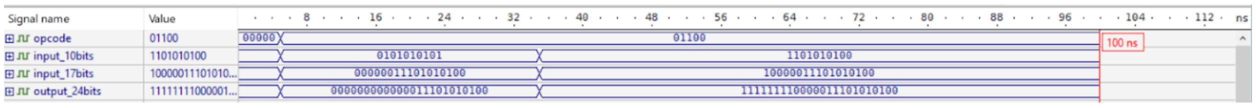
```verilog
module extender_testbench();
	reg [4:0] opcode;
	reg [9:0] input_10bits;
	reg [16:0] input_17bits;
	reg [23:0] output_24bits;

	extender ext(opcode, input_10bits, input_17bits, output_24bits);

	initial begin
		opcode = 0;
		input_10bits = 0;
		input_17bits = 0;
		#5
		opcode = 5'b01000;
		input_10bits = 10'b0101010101;
		input_17bits = 17'b00000011101010100;
		#30
		opcode = 5'b01000;
		input_10bits = 10'b1101010100;
		input_17bits = 17'b00000011101010100;
	end
endmodule
```

Figure 7: Extender testbench



Signal name	Value			
opcode	01000	00000	01000	
input_10bits	1101010100		0101010101	1101010100
input_17bits	00000011101010...		00000011101010100	
output_24bits	11111111111111...		000000000000000101010101	111111111111111101010100

Figure 8: First case output (Extender)



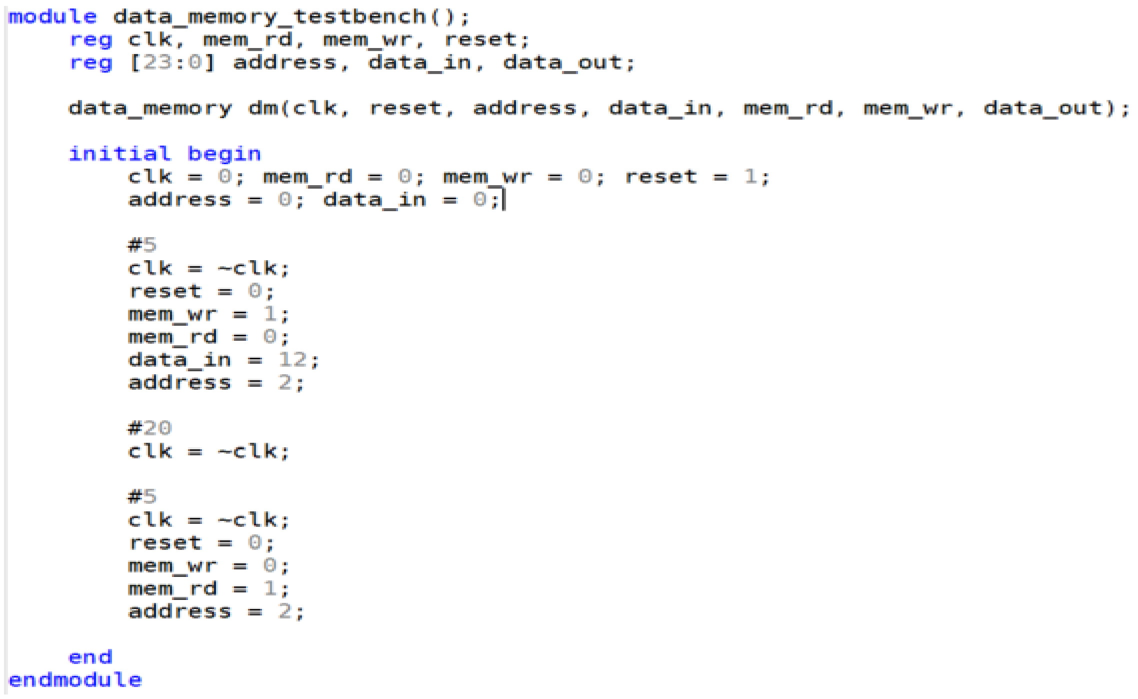
Signal name	Value			
opcode	01100	00000	01100	
input_10bits	1101010100		0101010101	1101010100
input_17bits	10000011101010...		00000011101010100	10000011101010100
output_24bits	11111111000001...		000000000000011101010100	111111110000011101010100

Figure 9: Second case output (Extender)

2.2. Data memory

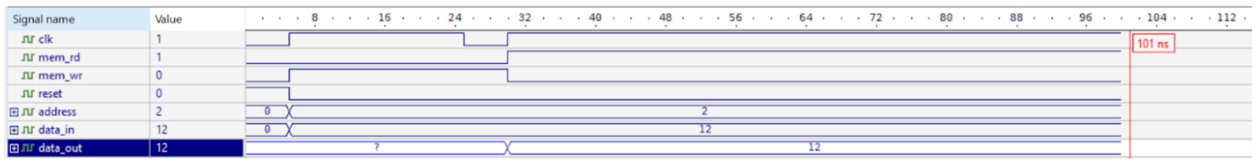
There are 2 operations associated with the data memory: reading and writing to it. These operations are controlled by two control signals mem_rd and mem_wr. The following testbench in figure 10 experiments both of the operations.

First, the mem_rd flag was set to 0 and the mem_wr flag was set to 1, indicating that the operation of writing on the memory will be executed. The data of 12 is written on the memory at address 2. No clear result will be shown in this case.
However, the result was demonstrated when the mem_rd and mem_wr were set to 1 and 0, respectively. The address was kept at value 2 to display the value that was just written. The simulation in figure 11 shows that the output saved in this memory address is 12, the same written value. This is a proof that the memory is working perfectly.



```
module data_memory_testbench();
    reg clk, mem_rd, mem_wr, reset;
    reg [23:0] address, data_in, data_out;

    data_memory dm(clk, reset, address, data_in, mem_rd, mem_wr, data_out);

    initial begin
        clk = 0; mem_rd = 0; mem_wr = 0; reset = 1;
        address = 0; data_in = 0;

        #5
        clk = ~clk;
        reset = 0;
        mem_wr = 1;
        mem_rd = 0;
        data_in = 12;
        address = 2;

        #20
        clk = ~clk;

        #5
        clk = ~clk;
        reset = 0;
        mem_wr = 0;
        mem_rd = 1;
        address = 2;

    end
endmodule
```

Figure 10: Data memory testbench

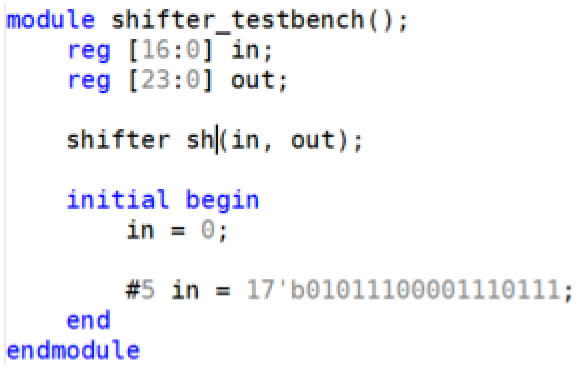


Signal name	Value
clk	1
mem_rd	1
mem_wr	0
reset	0
address	2
data_in	12
data_out	12

Figure 11: Data memory testing output

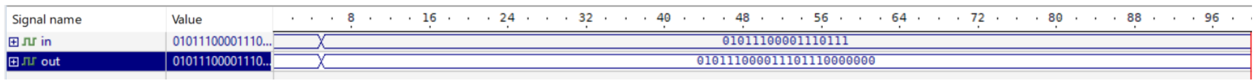
2.3. Shifter

To test this module, a random 17 bits value was inserted to the shifter. This value was 01011100001110111. Figure 12 shows the testbench code, and figure 13 shows that 7 zeros were added to the end of this number to give a 24-bits output of 010111000011101110000000. This is used in the LUI operation.



```
module shifter_testbench();
    reg [16:0] in;
    reg [23:0] out;

    shifter sh(in, out);

    initial begin
        in = 0;

        #5 in = 17'b01011100001110111;
    end
endmodule
```

Figure 12: Shifter testbench

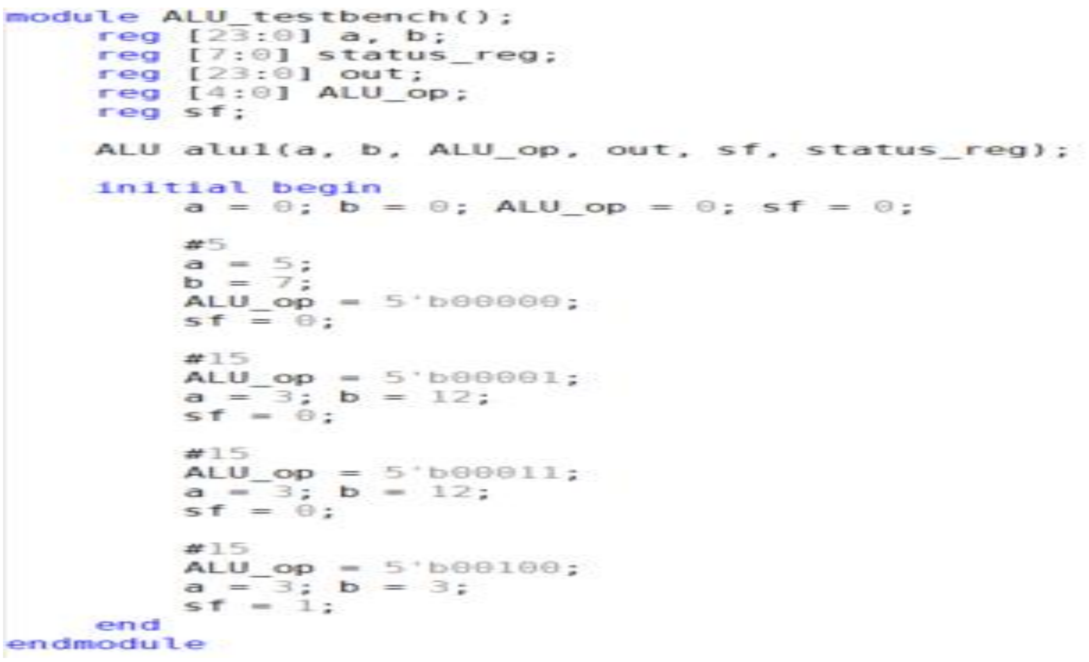


```
Signal name   Value             8   16   24   32   40   48   56   64   72   80   88   96
in            01011100001110...      01011100001110111
out           01011100001110...      010111000011101110000000
```

Figure 13: Shifter testing output

2.4. ALU

The ALU was tested with 4 different operations. The first one is with opcode 00000, the inputs had the values 5 and 7, the result is their ANDing. The second operation is 00001, the inputs had the values 3 and 12, the result is the maximum between them. The third operation is 00011, the inputs had the values 3 and 12, the result is their sum. The fourth operation is 00100, the inputs had the values 3 and 3, SF was set to 1, the result is their subtraction. Figure 14 shows the testbench of the ALU, and figure 15 shows the results.



```
module ALU_testbench();
    reg [23:0] a, b;
    reg [7:0] status_reg;
    reg [23:0] out;
    reg [4:0] ALU_op;
    reg sf;

    ALU alu1(a, b, ALU_op, out, sf, status_reg);

    initial begin
        a = 0; b = 0; ALU_op = 0; sf = 0;

        #5
        a = 5;
        b = 7;
        ALU_op = 5'b00000;
        sf = 0;

        #15
        ALU_op = 5'b00001;
        a = 3; b = 12;
        sf = 0;

        #15
        ALU_op = 5'b00011;
        a = 3; b = 12;
        sf = 0;

        #15
        ALU_op = 5'b00100;
        a = 3; b = 3;
        sf = 1;
    end
endmodule
```

Figure 14: ALU testbench

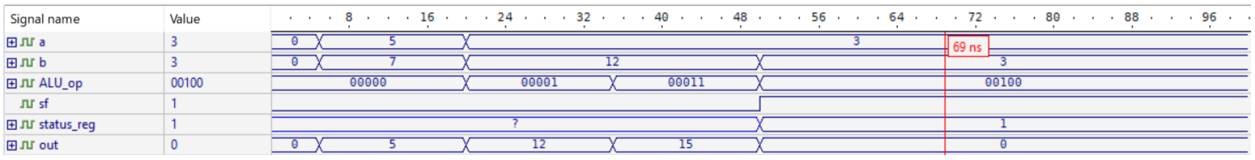


Figure 15: ALU testing output

Teamwork

Our team collaborated on a project by first brainstorming and refining ideas for each component's design. After agreeing on the design, we created a control unit signals table. Next, one team member built and tested the PC, IF/ID, and register file, while another team member built and tested the Decode Condition, sub-Condition, add Condition, and ALU control unit. The last team member built the ALU unit, Control unit, and extenders. Once all units were functioning correctly, we worked together to construct the data path and thoroughly tested it to ensure it met all necessary requirements. Then we wrote the code together and tested the units.

Conclusion

Designing a single cycle CPU using Verilog and Active-HDL tool was a challenging yet rewarding project that helped to gain an in-depth understanding of the principles of digital design. The project involved designing the data path, components, and finite state machine necessary for the CPU to execute instructions, including the control unit, register file, ALU, and data memory. Truth tables were used to obtain signal equations, and Verilog code was implemented for each component. Overall, the project was an excellent opportunity to gain practical experience in digital design and Verilog programming, culminating in a functional single cycle CPU design.

Appendix

The code is attached with the submission.