Logic Design 003-05 Lab-FINAL
Seoyeon Park 2022-16050

Purpose

The final lab of the semester involves creating an 8-bit microprocessor with 4 instructions: add, store, load, and jump. This implementation will be coded in Verilog and applied to a SNU Logic Design Board which will then be tested connecting it to a TA board. The objective of this lab is to mimic a real computer by designing and integrating fundamental microprocessor components.

Background Theory

Combinational vs Sequential Logic
Combinational logic is a digital logic whose outputs are based on the current inputs. Sequential logic is a digital logic whose outputs are based on the current inputs and past states which are saved in memory.

Clocking and Synchronization
Key features of sequential logic circuits are clocking and synchronization. Clocking refers to the use of a clock signal to determine when data is captured and transferred. Synchronization ensures that all parts of the circuit update their states at the same time. This can be set to be based on the rising or falling edge of the clock.

Microprocessor Basics

Fetch-Decode-Execute Cycle
The basic operation cycle of a processor involves in the repetition of Fetch, Decode, and Execute. During fetch, the next instruction is retrieved from memory. During decode, the instruction is interpreted to determine the operation. During execute the operation is performed.

Data path
Data path refers to the hardware components like registers, or ALUs that perform data processing and movement.

Control Unit
Control unit generates control signals to execute operations and guide data paths according to the instructions.

Instruction Set Architecture (ISA) and Control Logic

Instruction Set Architecture (ISA)

The 8-bit microprocessor that is implemented in the lab supports three instruction formats: R-type, I-type and J-type. R-type instructions are used for arithmetic operations like add, I-type instructions are used for memory operations such as lw (load) and sw (store), J-type instructions are used for control flows like jump. Each instruction is 8 bits wide and is composed of a combination of the 2-bit instruction fields: op, rd, rs, rt, and imm. op stands for operation code, meaning the operation that the microprocessor must execute. rd stands for destination register, rs stands for source register (1), and rt stands for source register 2. Finally, imm represents a constant, this value is signed, and its range is from -2 ~ 1. (This is expressed using 2's complement on 2 bits).

The register files where the data will be stored include four 8-bit registers: $s0, $s1, $s2, $s3. Additionally, a Program Counter (PC) is implemented for instruction sequencing.

Instruction Encoding

The instructions are encoded based on their type. This is shown in *Figure 1*.

Figure 1: Instruction Encoding

Type	Example							
	7	6	5	4	3	2	1	0
R	op		rs		rt		rd	
I	op		rs		rt		imm	
J	op						imm	

Utilizing this encoding we can encode instructions such as $s3 = $s1 + $s2 as 0001 1011, where op = 00 (add), rs = 01 ($s1), rt = 10 ($s2), and rd = 11($s3). (As the add operation utilizes R-type format, the respective encoding was used.

Control Logic

In the microprocessor a control unit will interpret the opcode and generate the appropriate signals to route data and execute the proper operation. The key control signals propagated by the control unit are: RegDst, RegWrite, ALUSrc, Branch, MemRead, MemWrite, MemtoReg, and ALUOP. *Figure 2* summarizes the purposes of each signal.

Figure 2: Instruction Encoding

Signal	Purpose
RegDst	Selects destination register (rd for R-type instruction, rt for I-type instruction)
RegWrite	Enables writing result to rd, rt
ALUSrc	Selects where the second operand for the ALU comes from
Branch	Controls whether the PC is updated with a jump target address instead of normal increment of 1
MemRead	Enables reading data from Data memory
MemWrite	Enables writing data into Data memory
MemtoReg	Selects the data source that will be written back to a register
ALUOP	Selects the specific operation the ALU should perform

For every instruction, a different combination of signals is propagated to ensure that the instruction is executed correctly. *Figure 3* displays which signals are enabled based on the instruction.

Figure 3: Control Signal Table

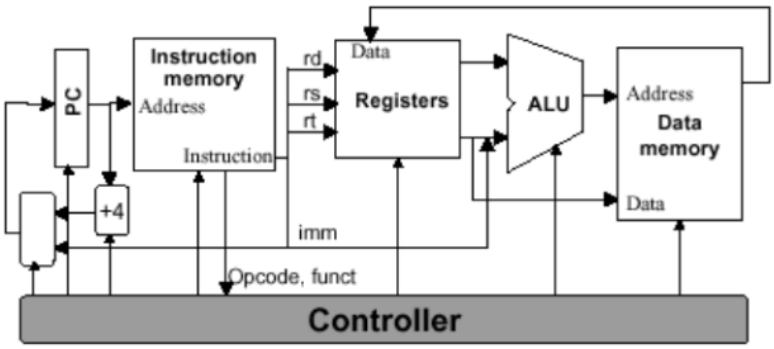
Instruction	RegDst	RegWrite	ALUSrc	Branch	MemRead	MemWrite	MemtoReg	ALUOP
R-format	1	1	0	0	0	0	0	1
lw	0	1	1	0	1	0	1	0
sw	x	0	1	0	0	1	x	0
j	x	0	0	1	0	0	x	0

Microprocessor Design

High Level Architecture

The design of the 8-bit microprocessor consists of five functional units all controlled by a central controller. *Figure 4* shows a high-level structure diagram of the microprocessor.

Figure 4: Computer Architecture (w/o pipeline)



The five functional units are Program Counter(PC), Instruction Memory, Registers, ALU, and Data Memory.

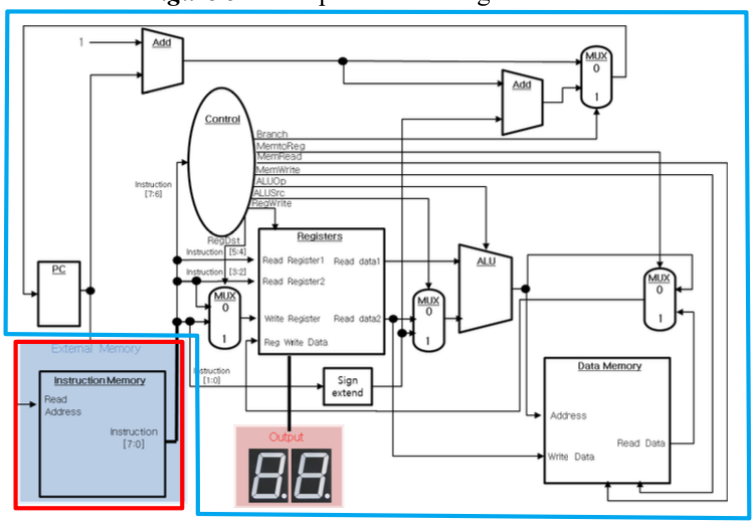
1. Program Counter is a register that holds the memory address of the instruction that is currently being executed or the next instruction to be fetched. It controls the instruction sequence.
2. Instruction Memory stores the machine code instructions that the CPU will execute. This unit receives the memory address of the instruction from the PC and the instruction from the controller. It outputs the Opcode to the controller and the signals rd, rs, and rt towards the registers.
3. Registers consist of four 8-bit registers that store data based on the instructions given by the controller.
4. ALU performs arithmetic operations, in our 8-bit microprocessor its primary function is addition.
5. The Data Memory stores reading and writing data given by the registers at the address calculated by the ALU.

All of this is controlled by a main controller that decodes each instruction and generates the needed control signals to direct the flow of data and execute the correct operation.

Data Path Design

The data path diagram of the microprocessor shows the specific connections inside of the microprocessor. *Figure 5* shows this diagram.

Figure 5: Microprocessor Design Data Path



Instruction execution can be explained by this diagram. This process can be divided into several phases: Instruction Fetch Phase, Instruction Decode and Register Fetch Phase, Execute Phase, Memory Access Phase, and Write Back Phase.

1. Instruction Fetch Phase
    a. PC, which holds the address of the current instruction, inputs Instruction Address towards Instruction Memory which then output an 8-bit instruction to the main data path.
    b. Utilizing a MUX controlled by the signal Branch, the PC's next value is selected.

        i. If Branch is 0, the PC's value is incremented by 1.
        ii. If Branch is 1, the PC's value is updated by (1+imm) simulating a jump in address.
    c. The PC's next value is then inputted into back into the PC.
2. Instruction Decode and Register Fetch Phase
    a. The instruction outputted by Instruction memory is divided into four: Opcode [7:6], rs [5:4], rt [3:2], and rd/imm [1:0].
    b. Opcode is sent to the Control unit to determine the necessary control signals for execution or instructions.
    c. rs is sent as a Read Register1 address, and rt is sent as a Read Register 2 address.
    d. The Register file outputs the content of the inputted registers as Read data1 and Read data2.
    e. Rd/imm is sent to a MUX and towards a Sign Extend Unit.
3. Execute Phase
    a. The output from the Registers, Read Data 1, is fed as the first operand to the ALU, and the second operand is selected by a MUX controlled by the ALUSrc signal.
        i. If ALUSrc is 0, Read data 2 is fed. (add)
        ii. If ALUSrc is 1, the imm field extended to 8 bits using Sign Extend is fed. (load, store)
    b. The ALU performs its operation and produces an ALU result signal.
4. Memory Access Phase
    a. The ALU result signal is used as the Address for the Data memory unit.
    b. The MemRead control signal enables reading data from Data Memory to the Read Data output.
    c. The MemWrite control signal enables writing data, coming from Read data 2, to Data Memory
5. Write Back Phase
    a. Based on a MUX controlled by the control signal MemtoReg, the data source for Reg Write Data is decided.
        i. If MemtoReg is 0, the ALU result is selected. (add)
        ii. If MemtoReg is 1, the Read Data from Data Memory unit is selected. (load)
    b. The destination register address for this write is selected by a Mux based on the control signal RegDst.
        i. If RegDst is 0, the rt field is used. (load)
        ii. If RegDst is 1, the rd field is used. (add)
    c. The write operation is controlled by the RegWrite control signal.

Component Level Design and Implementation

Several Verilog modules were created and utilized together to build the 8-bit microprocessor. An analysis of each file created will be given below.

1. Register.v

This module implements the four 8-bit registers needed in the implementation. This module can read two registers simultaneously and one register to be written to a specific address (W) if the RW signal is active. It handles asynchronous RESET, and it is composed of 7 inputs and 3 outputs. Their meaning/usage is explained in *Figure 6*.

Figure 6: Input/Output of Register.v

Input/Output	Meaning
R1 [1:0] (Input)	2-bit address of the first register to read
R2 [1:0] (Input)	2-bit address of the second register to read
W [1:0] (Input)	2-bit address of the register to write to
WD [7:0] (Input)	8-bit data to be written into the register at W [1:0] (Write Data)
RW (Input)	1-bit RegWrite signal, enables the write operation
CLK (Input)	1-bit clock signal
RESET (Input)	1-bit asynchronous reset signal
RD1 [7:0] (Output)	8-bit data read from the register with address given by R1.
RD2 [7:0] (Output)	8-bit data read from the register with address given by R2.
Y [7:0] (Output)	8-bit output of Write Data input.

Logic Explanation

The logic for this module is as follows.

1. Create array named mem containing 4 elements to represent the four 8-bit registers needed.
2. At the rising edge of CLK or RESET check whether RESET or RW signal is enabled.
    a. If RESET is 1, all mem locations are asynchronously cleared to 8b'00000000.
    b. If RESET is 0 and RW is 1, data WD is written into the register at address W.
3. Assign mem[R1] and mem[R2] to RD1 and RD2 respectively.
4. Assign WD to output Y.
    a. This output will be converted to a 7-segment output and displayed on the FPGA.

Verilog Code

Figure 7: Register.v

```
module Register(
    input [1:0] R1,
    input [1:0] R2,
    input [1:0] W,
     input [7:0] WD,
     input RW,
    input CLK,
     input RESET,
    output [7:0] RD1,
    output [7:0] RD2,
    output [7:0] Y
    );

    integer i;
    reg [7:0] mem [0:3];

    always @(posedge CLK or posedge RESET) begin
        if(RESET == 1) begin
            for(i = 0; i < 4; i = i + 1) mem[i] <= 8'b00000000;
        end
        else if(RW == 1) begin
            mem[W] <= WD;
        end
    end
    assign RD1 = mem[R1];
    assign RD2 = mem[R2];
    assign Y = WD;

endmodule
```

2. PC.v

This module implements the Program Counter, a register that is responsible for holding the memory address of the instruction that is being executed or the next instruction to be fetched. This module handles asynchronous input RESET and is composed of 3 inputs and 1 output . The meaning of these is explained in *Figure 8.*

Figure 8: Input/Output of PC.v

Input/Output	Meaning
A [7:0] (Input)	8-bit address of the next address the PC should hold
CLK (Input)	1-bit clock signal
RESET (Input)	1-bit asynchronous reset signal
Y [7:0] (Output)	8-bit output representing the current address stored in the PC.

Logic Explanation

The logic for this module is as follows.

1. At rising edge of CLK or RESET check whether RESET signal is enabled.
    a. If RESET is 1, output Y is assigned to 8'b00000000, ensuring processor starts execution from memory address 0.
    b. If RESET is 0, output Y is assigned to A, moving the PC to the next instruction's address.

Verilog Code

Figure 9: PC.v

```
module PC(
    input [7:0] A,
     input RESET,
     input CLK,
    output reg [7:0] Y
    );

    always @(posedge CLK or posedge RESET) begin
        if(RESET == 1) Y <= 8'b00000000;
        else Y <= A;
    end


endmodule
```

3. MUX8.v and MUX2.v

Both modules implement a 2 to 1 Multiplexor. MUX8 is for 8-bit input signals and MUX2 is for 2-bit input signals. The output of this module is based on the state of the 1-bit select signal. They are composed of 3 inputs and 1 output . The meaning of these is explained in *Figure 10.*

Figure 10: Input/Output of MUX2.v, MUX8.v

MUX8	MUX2	Meaning
A [7:0] (Input)	A [1:0] (Input)	8-bit/ 2-bit input, this input is chosen when S is 0
B [7:0] (Input)	B [1:0] (Input)	8-bit/ 2-bit input, this input is chosen when S is 1
S (Input)	S (Input)	1-bit select signal
Y [7:0] (Output)	Y [1:0] (Output)	8-bit output representing selecting input.

Logic Explanation

The logic for this module is as follows.

1. If S is 0, Y is assigned the value of A.
2. If S is 1, Y is assigned the value of B.

Verilog Code

Figure 11: MUX8.v, MUX2.v

```
module MUX8(
    input [7:0] A,
    input [7:0] B,
    input S,
    output [7:0] Y
    );

    assign Y = (S == 0) ? A : B;

endmodule
```

```
module MUX2(
    input [1:0] A,
    input [1:0] B,
    input S,
    output [1:0] Y
    );

    assign Y = (S == 0) ? A : B;

endmodule
```

4. MEM.v

This module implements the Data Memory component. This module provides access to read and write to a block of 32 8-bit memory locations. This module handles asynchronous input RESET and is composed of 6 inputs and 1 output . The meaning of these is explained in *Figure 12.*

Figure 12: Input/Output of MEM.v

Input/Output	Meaning
AD [7:0] (Input)	8-bit address of the memory access for read and write.
WD [7:0] (Input)	8-bit write data
MW (Input)	1-bit MemWrite control signal
MR (Input)	1-bit MemRead control signal
CLK (Input)	1-bit clock signal
RESET (Input)	1-bit asynchronous reset signal
RD [7:0] (Output)	8-bit data read from memory

Logic Explanation

The logic for this module is as follows.

1. Create array named mem containing 32 elements to represent the 32 8-bit registers needed.
2. At the rising edge of CLK or RESET check whether RESET or MW signal is enabled.
    a. If RESET is 1, utilizing a for loop the mem array is set to the following.
        i. For addresses 0 – 15 mem[i] is set to i. (Ex: mem[4] = 4)
        ii. For addresses 16- 31 mem[i] is set to 16-i, or the 2's complement of i. (Ex: mem[31] = -15)
    b. If RESET is 0 and MW is 1, data WD is written into the memory at address AD.
3. Assign RD based on the MR signal.
    a. If MR is 0, output RD is assigned to 8'b00000000.
    b. If MR is 1, output RD is assigned to the content of the memory location mem[AD].

Verilog Code

Figure 13: MEM.v

```
module MEM(
    input [7:0] AD,
    input [7:0] WD,
     input MW,
     input MR,
     input RESET,
     input CLK,
    output [7:0] RD
    );

    reg [7:0] mem [0:31];
    integer i;

    always @(posedge CLK or posedge RESET) begin
        if(RESET == 1) begin
            for(i = 0; i < 32; i = i + 1) begin
                if(i < 16) mem[i] <= i;
                else mem[i] <= 16 - i;
            end
        end
        else if(MW == 1) begin
            mem[AD] <= WD;
        end
    end

    assign RD = (MR == 0) ? 8'b00000000 : mem[AD];


endmodule
```

5. freq_div.v

This module implements a Clock frequency divider. This module takes the 50MHz FPGA clock and generates a 1Hz output clock. This allows for the microprocessor operations to be viewed at a speed observable by humans. This module handles asynchronous input RESET and is composed of 2 inputs and 1 output . The meaning of these is explained in *Figure 14.*

Figure 14: Input/Output of freq_div.v

Input/Output	Meaning
sCLK (Input)	1-bit high frequency system clock signal
RESET (Input)	1-bit asynchronous reset signal
CLK (Output)	1-bit slower output clock

Logic Explanation

The logic for this module is as follows.

1. Create 32 bit register cnt, to count until 25,000,000
2. At the rising edge of the sCLK check for RESET signal
    a. If RESET is 1, set cnt and output CLK to 0
    b. If cnt = 32d'25000000 or 25,000,000 clock cycles, reset cnt to 0 and CLK is toggled to ~CLK.
        i. One full period of CLK will consist of 50,000,000 cycles at 50MHz, thus CLK is toggled every 25,000,000 cycles.
    c. Else increase counter by 1.

Verilog Code

Figure 15: freq_div.v

```
module freq_div(
     input RESET,
    input sCLK,
    output reg CLK
    );

    reg [31:0] cnt;

    always @(posedge sCLK) begin
        if (RESET) begin
            cnt <= 32'd0;
            CLK <= 1'b0;
        end
        else if(cnt == 32'd25000000) begin
            cnt <= 32'd0;
            CLK <= ~CLK;
        end
        else begin
            cnt <= cnt + 1;
        end
    end

endmodule
```

6. CTRL.v

This module implements a Control Unit of the 8-bit microprocessor. This module takes opcode part of the instruction and generates all the necessary control signals to execute the operation. This module is composed of 1 input and 8 outputs. The meaning of these is explained in *Figure 16.*

Figure 16: Input/Output of CTRL.v

Input/Output	Meaning
A[1:0] (Input)	2-bit input representing opcode of instruction. (Instruction[7:6])
RegDst (Output)	Check *Figure 2* for meaning of each signal.
RegWrite (Output)	
ALUSrc (Output)	
Branch (Output)	
MemRead (Output)	
MemWrite (Output)	
MemtoReg (Output)	
ALUOP (Output)	

Logic Explanation

The logic for this module is as follows.

1. Create 8 bit register out to hold all control signals.
2. At every change of A
    a. If A is 4'b00 (add) assign 8'b11000001 to out.
    b. If A is 4'b01 (load) assign 8'b01101011 to out.
    c. If A is 4'b10 (store) assign 8'b00100101 to out.
    d. If A is 4'b11 (jump) assign 8'b00010000 to out.
3. Assign each individual bit of out register to signal outputs.
    a. Signal outputs should match *Figure 3*.

Contrary to the given signals in the LAB guide our implementation is giving the control signal ALUOP a 1 for operations load and store so the ALU can perform addition. If the ALUOP signal stays at 0, then the ALU would return a 0 output, which is not what we want.

Verilog Code

Figure 17: CTRL.v

```
module CTRL(
    input [1:0] A,
    output RegDst,
    output RegWrite,
    output ALUSrc,
    output Branch,
    output MemRead,
    output MemWrite,
    output MemtoReg,
    output ALUOP
    );

    reg [7:0] out;

    always @(A) begin
        case(A)
            4'b00 : out <= 8'b11000001;
            4'b01 : out <= 8'b01101011;
            4'b10 : out <= 8'b00100101;
            4'b11 : out <= 8'b00010000;
        endcase
    end

    assign RegDst = out[7];
    assign RegWrite = out[6];
   assign ALUSrc = out[5];
   assign Branch = out[4];
   assign MemRead = out[3];
   assign MemWrite = out[2];
   assign MemtoReg = out[1];
    assign ALUOP = out[0];

endmodule
```

7. BCD7.v

This module implements a Binary-Coded decimal to 7-Segment Display Decoder. It takes a 4-bit binary input, representing a hexadecimal digit from 0 to F, and converts it to a 7-bit output to be displayed in a 7-segment display. This module is composed of 1 input and 1 output. The meaning of these is explained in *Figure 18.*

Figure 18: Input/Output of BCD7.v

Input/Output	Meaning
A[3:0] (Input)	4-bit input representing hexadecimal digit (0~F)
Y[6:0] (Output)	7-bit output, representing mapping for 7 segment display

Logic Explanation

The logic for this module is as follows.

1. Assign Y based on A.
    a. Ex) If A is 4'b0000 (add) assign 7'b0111111 to Y.

Verilog Code

Figure 19: BCD7.v

```
module BCD7(
    input [3:0] A,
    output [6:0] Y
    );

    assign Y = (A == 4'b0000) ? 7'b0111111 :
                    (A == 4'b0001) ? 7'b0000110 :
                    (A == 4'b0010) ? 7'b1011011 :
                    (A == 4'b0011) ? 7'b1001111 :
                    (A == 4'b0100) ? 7'b1100110 :
                    (A == 4'b0101) ? 7'b1101101 :
                    (A == 4'b0110) ? 7'b1111101 :
                    (A == 4'b0111) ? 7'b0000111 :
                    (A == 4'b1000) ? 7'b1111111 :
                    (A == 4'b1001) ? 7'b1101111 :
                    (A == 4'b1010) ? 7'b1110111 :
                    (A == 4'b1011) ? 7'b1111100 :
                    (A == 4'b1100) ? 7'b0111001 :
                    (A == 4'b1101) ? 7'b1011110 :
                    (A == 4'b1110) ? 7'b1111001 : 7'b1110001;

endmodule
```

8. ALU.v

This module implements an Arithmetic Logic Unit, ALU, to perform an arithmetic operation on two 8-bit inputs. It was only implemented to perform the addition operation when ALUOP signal is 1. This module is composed of 3 inputs and 1 output. The meaning of these is explained in *Figure 20.*

Figure 20: Input/Output of ALU.v

Input/Output	Meaning
A[7:0] (Input)	8-bit first operand
B[7:0] (Input)	8-bit second operand
ALUOP (Input)	1 bit control signal
Y[7:0] (Output)	8-bit result of ALU operation

Logic Explanation

The logic for this module is as follows.

1. If ALUOP is 1, add A+B and assign it to Y.
2. IF ALUOP is 0, assign 0 to Y.

Verilog Code

Figure 21: ALU.v

```
module ALU(
    input [7:0] A,
    input [7:0] B,
    input ALUOP,
    output [7:0] Y
    );

    assign Y = (ALUOP == 1) ? A + B : 8'b00000000;

endmodule
```

9. ADD.v

This module implements an 8-bit adder. This module is composed of 2 inputs and 1 output. The meaning of these is explained in *Figure 22.*

Figure 22: Input/Output of ADD.v

Input/Output	Meaning
A[7:0] (Input)	8-bit first operand
B[7:0] (Input)	8-bit second operand
Y[7:0] (Output)	8-bit sum of A and B

Logic Explanation

The logic for this module is as follows.

1. Assign Y to sum of A and B

Verilog Code

Figure 23: ADD.v

```
module ADD(
    input [7:0] A,
    input [7:0] B,
    output reg [7:0] Y
    );

    always @(*) begin
        Y <= A + B;
    end

endmodule
```

10. Output.v

This module takes the Write Data, output Y of Register, and Instruction opcode as inputs and outputs five seven segment displays. It displays the hexadecimal value of WD or the specific text message “STORE” and “JUMP”. This module is composed of 2 inputs and 5 outputs. The meaning of these is explained in *Figure 24.*

Figure 24: Input/Output of Output.v

Input/Output	Meaning
WD [7:0] (Input)	8-bit data that was written into a register (Register output Y)
Ins [1:0] (Input)	2-bit Instructions (opcode)
Seg1 [6:0] (Output)	7-bit output for 7 segment display (segment 1)
Seg2 [6:0] (Output)	7-bit output for 7 segment display (segment 2)
Seg3 [6:0] (Output)	7-bit output for 7 segment display (segment 3)

Seg4 [6:0] (Output)	7-bit output for 7 segment display (segment 4)
Seg5 [6:0] (Output)	7-bit output for 7 segment display (segment 5)

Logic Explanation

The logic for this module is as follows.

1. Create two instances of BCD7 module and divide WD signal into two. (WD[7:4], WD[3:0])
    a. This divides the Write Data into its most significant bit and least significant bit.
2. According to the instruction operation, the different segments are assigned their respective outputs.
    a. If Ins is 2'b10, the store operation, the 7-segment display will display the word "STORE" across Seg1 to Seg5.
    b. If Ins is 2'b11, the jump operation, the 7-segment display will display the word "JUMP" across Seg1 to Seg5. (M is divided into two segments)
    c. Else, add and load operation, the 7-segment display will display the upper 4 bits of WD into segment1, and the lower 4 bits of WD into segment2.
        i. All other segments are set to 0 to ensure only the result of add, or the loaded value for load is shown.

Verilog Code

Figure 23: Output.v

```
module Output(
    input [7:0] WD,
    input [1:0] Ins,
    output reg [6:0] Seg1,
    output reg [6:0] Seg2,
    output reg [6:0] Seg3,
    output reg [6:0] Seg4,
    output reg [6:0] Seg5
    );

    wire [6:0] tmp1, tmp2;

    BCD7 B1(WD[7:4], tmp1);
    BCD7 B2(WD[3:0], tmp2);


    always @(*) begin
        if(Ins == 2'b10) begin
            Seg1 <= 7'b1101101;
            Seg2 <= 7'b1111000;
            Seg3 <= 7'b0111111;
            Seg4 <= 7'b1110000;
            Seg5 <= 7'b1111001;
        end
        else if(Ins == 2'b11) begin
            Seg1 <= 7'b0001110;
            Seg2 <= 7'b0111110;
            Seg3 <= 7'b0110111;
            Seg4 <= 7'b0110111;
            Seg5 <= 7'b1110011;
        end
        else begin
            Seg1 <= tmp1;
            Seg2 <= tmp2;
            Seg3 <= 7'b0000000;
            Seg4 <= 7'b0000000;
            Seg5 <= 7'b0000000;
        end
    end

endmodule
```

11. SignEx.v

This module implements a Sign extender, which takes a 2-bit signed number and converts it into an 8-bit signed number. This module is composed of 1 input and 1 output. The meaning of these is explained in *Figure 22.*

Figure 22: Input/Output of SignEx.v

Input/Output	Meaning
A[1:0] (Input)	2-bit signed input
Y[7:0] (Input)	8-bit signed output

Logic Explanation

The logic for this module is as follows.

1. Assign to Y an 8-bit signed number by replicating the sign bit A[1] six times and concatenating it to A.
    a. Ex: 01 -> 00000001

Verilog Code

Figure 23: SignEx.v

```
module SignEx(
    input [1:0] A,
    output [7:0] Y
    );

    assign Y = {{6{A[1]}}, A};

endmodule
```

12. CPU.v

This module is the main module of the 8-bit microprocessor. It is designed to be implemented on the FPGA board. It combines all above modules to form a complete, operational CPU data path and outputs a 7-segment display according to the instructions inputted. This module is composed of 3 inputs and 6 outputs. The meaning of these is explained in *Figure 24.*

Figure 24: Input/Output of CPU.v

Input/Output	Meaning
Ins[7:0] (Input)	8-bit Instructions from TA board
RESET (Input)	1-bit asynchronous global reset signal
sCLK (Input)	System clock
SEG1 [6:0] (Output)	Output for 7-segment display (segment 1)
SEG2 [6:0] (Output)	Output for 7-segment display (segment 2)
SEG3 [6:0] (Output)	Output for 7-segment display (segment 3)
SEG4 [6:0] (Output)	Output for 7-segment display (segment 4)
SEG5 [6:0] (Output)	Output for 7-segment display (segment 5)
ADD [7:0] (Output)	8-bit output representing current PC address.

Logic Explanation

The logic for this module is as follows the data path design of the microprocessor. Most wires are named how they are described in Data Path and hold the same meaning. Wire PPC refers to Present PC, OPC refers to One PC (PC+1), JPC refers to Jump PC (OPC + SignExtend), and NPC refers to Next PC.

1. Clocking
    a. Utilizing the freq_div module a new CLK (1Hz) is created which will be used for all synchronous components.
2. PC and Instruction Fetch
    a. Utilizing PC module, PC is instantiated with NPC (Next PC) as input and PPC (Present PC) as output.
    b. PPC is assigned to ADD (current PC address).
3. Control Unit
    a. Utilizing the CTRL module, Control unit is instantiated with Ins[7:6] (opcode) as input and control signals (RegDst, RegWrite, ALUSrc, Branch, MemRead, MemWrite, MemtoReg, and ALUOP) as outputs.
4. Register File
    a. Utilizing the MUX2 module, A 2-bit multiplexor decides whether the Write Register address comes from rt (Ins[3:2]) or rd (Ins[1:0]).
    b. Utilizing the Register module, Registers are instantiated with Ins[5:4], Ins[3:2], WriteReg,

WriteData, RegWrite, CLK, and RESET as inputs and ReadData1, ReadData2, and Out as outputs.
    i. RegWrite enables write.
    ii. WriteData comes from an 8-bit MUX controlled by MemtoReg.
    iii. Out is sent to the Output module

5. ALU
    a. Utilizing SignEx module, Ins[1:0] is sign extended and given to wire SignExtend.
    b. 8-bit MUX module controlled by ALUSrc selects between ReadData2 and SignExtend for the second operand of the ALU.
    c. Utilizing an ALU module, a ALU unit is instantiated taking the control signal ALUOP with the first operand being ReadData1, and the second operand being the output of the 8-bit MUX above.
6. Data Memory
    a. Utilizing MEM module, a Memory nit is instantiated with ALUResult, ReadData2, MemWrite, MemRead, RESET, and CLK as inputs and MemoryRead as output.
        i. The memory address of the Data Memory unit is given by the result of the ALU.
        ii. The write data of the Data Memory unit is given by ReadData2.
        iii. According to the control signals MemWrite and MemRead, Memory is written or read.
7. Write Back
    a. Utilizing an 8-bit MUX controlled by the MemtoReg signal selects between MemoryRead (Memory Read Data) or ALUResult to write back to the Register (Write Data).
8. PC Update Logic
    a. Utilizing ADD Module current PC address is added with 8'b00000001 and stored in OPC.
    b. Utilizing ADD Module OPC is added with SignExtend to calculate the potential jump target and stored in JPC.
    c. 8-bit MUX controlled by Branch selects between OPC and JPC for the NPC (Next PC).
    d. NPC signal is fed back to the PC module.
9. Output Display
    a. Utilizing Output module, the 7-segment displays are created with Out (Output from register), Ins[7:6] (opcode) as inputs.

Verilog Code

Figure 25: CPU.v

```
module CPU(
    input [7:0] Ins,
     input RESET,
     input sCLK,
    output [6:0] SEG1,
    output [6:0] SEG2,
    output [6:0] SEG3,
    output [6:0] SEG4,
    output [6:0] SEG5,
    output [7:0] ADD
    );

    wire CLK;
    wire RegDst, RegWrite, ALUSrc, Branch, MemRead, MemWrite, MemtoReg, ALUOP;
    wire [1:0] WriteReg;
    wire [7:0] ReadData1;
    wire [7:0] ReadData2;
    wire [7:0] WriteData;
    wire [7:0] Data3;
    wire [7:0] Out;
    wire [7:0] SignExtend;
    wire [7:0] ALUResult;
    wire [7:0] MemoryRead;
    wire [7:0] PPC, OPC, JPC, NPC;

    assign ADD = PPC;
    freq_div F(RESET, sCLK, CLK);

    CTRL C(Ins[7:6], RegDst, RegWrite, ALUSrc, Branch, MemRead, MemWrite, MemtoReg, ALUOP);
    MUX2 M1(Ins[3:2], Ins[1:0], RegDst, WriteReg);
    Register R(Ins[5:4], Ins[3:2], WriteReg, WriteData, RegWrite, CLK, RESET, ReadData1, ReadData2, Out);
    SignEx S(Ins[1:0], SignExtend);
    MUX8 M2(ReadData2, SignExtend, ALUSrc, Data3);
    ALU A1(ReadData1, Data3, ALUOP, ALUResult);
    MEM M3(ALUResult, ReadData2, MemWrite, MemRead, RESET, CLK, MemoryRead);
    MUX8 M4(ALUResult, MemoryRead, MemtoReg, WriteData);
    PC P(NPC, RESET, CLK, PPC);
    ADD A2(PPC, 8'b00000001, OPC);
    ADD A3(OPC, SignExtend, JPC);
    MUX8 M5(OPC, JPC, Branch, NPC);
    Output O(Out, Ins[7:6], SEG1, SEG2, SEG3, SEG4, SEG5);

endmodule
```

Testing Modules

To test the functionality of the microprocessor two new files were created: IMEM.v which stores the program's instructions and CPUT.v which integrates the IMEM module allowing simulations and verify the functionality of all the created modules.

1. IMEM.v

This module implements the Instruction Memory component of the microprocessor. It stores the program's instructions on different addresses and based on the address inputted it returns the instruction at that address. This module is composed of one input and one output.

Figure 26: Input/Output of IMEM.v

Input/Output	Meaning
ADD [7:0] (Input)	8-bit address of the instruction to be fetched
Ins [7:0] (Output)	8-bit output representing the instruction read from the memory location given by ADD

Logic Explanation

The logic for this module is as follows.

a. An array of 26 8-bit elements called mem is created.
b. For every memory address a specific 8-bit instruction is assigned
c. The instruction at memory address ADD is assigned to the output Ins.

Verilog Code

Figure 27: IMEM.v

```
module IMEM(
    input [7:0] ADD,
    output [7:0] Ins
    );
    wire [7:0] mem[25:0];

    assign mem[0] = 8'b01001000; // s2 = m[s0] = 0      / 0 0 0 0   / 00
    assign mem[1] = 8'b01001001; // s2 = m[s0 + 1] = 1  / 0 0 1 0   / 01
    assign mem[2] = 8'b01100101; // s1 = m[s2 + 1] = 2  / 0 2 1 0   / 02
    assign mem[3] = 8'b00011011; // s3 = s1 + s2 = 3    / 0 2 1 3   / 03
    assign mem[4] = 8'b11000001; // jump                    / 0 2 1 3   / JUMP
    assign mem[5] = 8'b11000001; // skip
    assign mem[6] = 8'b10101110; // m[s2 - 1] = s3      / 0 2 1 3   / STORE
    assign mem[7] = 8'b00011100; // s0 = s1 + s3        / 5 2 1 3   / 05
    assign mem[8] = 8'b00000000; // s0 = s0 + s0            / 10 2 1 3  / 0A
    assign mem[9] = 8'b01001001; // s2 = m[s0 + 1] = 11/ 10 2 11 3 / 0B
    assign mem[10] = 8'b00100001;// s1 = s0 + s2            / 10 21 11 3/ 15
    assign mem[11] = 8'b01011110;// s3 = m[s1 - 2] = -3/ 10 21 11 -3 / FD
    assign mem[12] = 8'b01011111;// s3 = m[s1 - 1] = -4/ 10 21 11 -4 / FC
    assign mem[13] = 8'b00101100;// s0 = s2 + s3        / 7 21 11 -4/ 07
    assign mem[14] = 8'b00000000;// s0 = s0 + s0        / 14 21 11 -4 / 0E
    assign mem[15] = 8'b10000110;// m[s0 - 2] = s1      / 14 21 11 -4 / STORE
    assign mem[16] = 8'b01100001;// s0 = m[s2 + 1]      / 21 21 11 -4 / 15
    assign mem[17] = 8'b00000100;// s0 = s0 + s1        / 42 21 11 -4 / 2A
    assign mem[18] = 8'b00000011;// s3 = s0 + s0            / 42 21 11 84 / 54
    assign mem[19] = 8'b00001110;// s2 = s0 + s3            / 42 21 126 84 / 7E
    assign mem[20] = 8'b11000001;// jump + 1 / JUMP
    assign mem[21] = 8'b11000001;// jump + 1 / JUMP
    assign mem[22] = 8'b11000010;// jump - 2 / JUMP
    assign mem[23] = 8'b00000110;// 3F
    assign mem[24] = 8'b11000011;// loop
    assign mem[25] = 8'b00000110;// dont come

    assign Ins = mem[ADD];

endmodule
```

The operation and expected output are shown as comments.

2. CPUT.v

This module is nearly identical to the CPU.v module except some modifications to integrate IMEM. This test version does not utilize a freq_div module to slow down the time. Additionally, as the instructions are given by the IMEM module, Instructions was changed from an input to an output.

Logic Explanation

The logic for this module the same as the logic for CPU.v. The only difference is that now CLK is set as the sCLK, and the instructions are given by the IMEM module.

Verilog Code

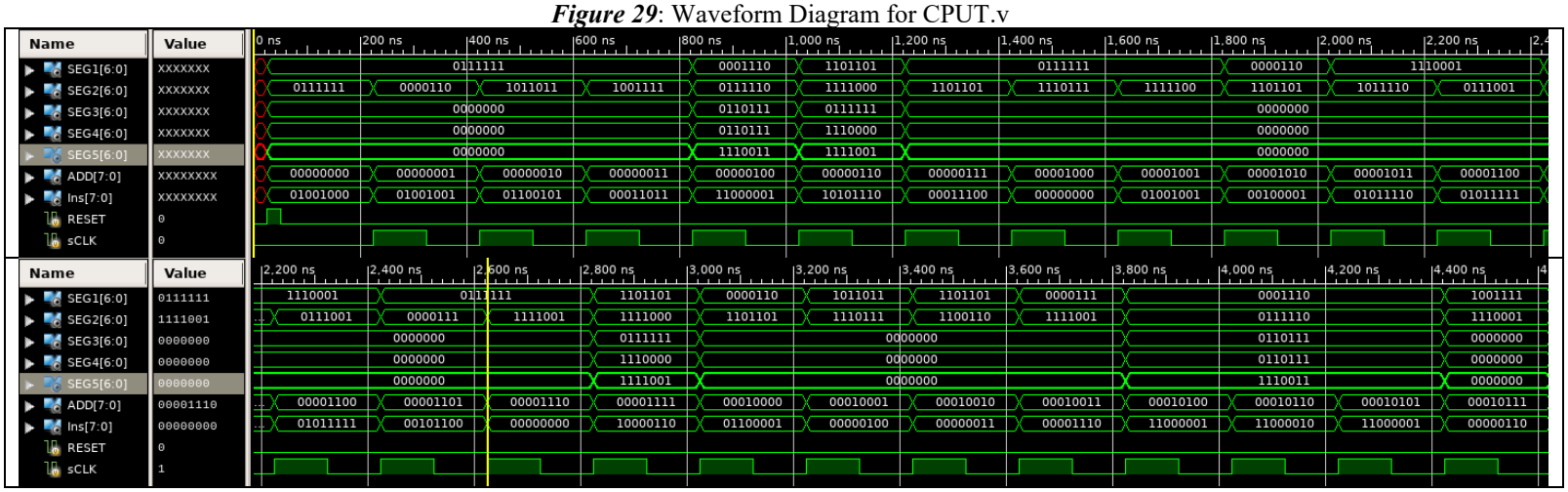
Figure 28: CPUT.v

```
module CPUT(
     input RESET,
     input sCLK,
    output [6:0] SEG1,
    output [6:0] SEG2,
    output [6:0] SEG3,
    output [6:0] SEG4,
    output [6:0] SEG5,
     output [7:0] ADD,
     output [7:0] Ins
    );

    wire CLK;
    wire RegDst, RegWrite, ALUSrc, Branch, MemRead, MemWrite, MemtoReg, ALUOP;
    wire [1:0] WriteReg;
    //wire [7:0] Ins;
    //wire [7:0] ADD;
    wire [7:0] ReadData1;
    wire [7:0] ReadData2;
    wire [7:0] WriteData;
    wire [7:0] Data3;
    wire [7:0] Out;
    wire [7:0] SignExtend;
    wire [7:0] ALUResult;
    wire [7:0] MemoryRead;
    wire [7:0] PPC, OPC, JPC, NPC;

    assign ADD = PPC;
    //freq_div F(RESET, sCLK, CLK);
    assign CLK = sCLK;

    CTRL C(Ins[7:6], RegDst, RegWrite, ALUSrc, Branch, MemRead, MemWrite, MemtoReg, ALUOP);
    MUX2 M1(Ins[3:2], Ins[1:0], RegDst, WriteReg);
    Register R(Ins[5:4], Ins[3:2], WriteReg, WriteData, RegWrite, CLK, RESET, ReadData1, ReadData2, Out);
    SignEx S(Ins[1:0], SignExtend);
    MUX8 M2(ReadData2, SignExtend, ALUSrc, Data3);
    ALU A1(ReadData1, Data3, ALUOP, ALUResult);
    MEM M3(ALUResult, ReadData2, MemWrite, MemRead, RESET, CLK, MemoryRead);
    MUX8 M4(ALUResult, MemoryRead, MemtoReg, WriteData);
    PC P(NPC, RESET, CLK, PPC);
    ADD A2(PPC, 8'b00000001, OPC);
    ADD A3(OPC, SignExtend, JPC);
    MUX8 M5(OPC, JPC, Branch, NPC);
    Output O(Out, Ins[7:6], SEG1, SEG2, SEG3, SEG4, SEG5);
    IMEM I(ADD, Ins);

endmodule
```

Simulation Results

Two types of simulations were done to test the functionality and accuracy of the microprocessor. The first type of simulation was creating a Verilog Test Bench and simulating the results. This gave us the following waveform diagrams.

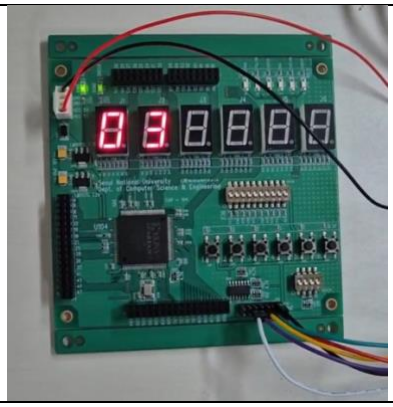
Figure 29: Waveform Diagram for CPUT.v

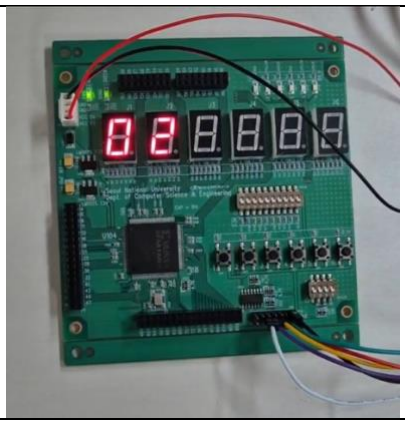
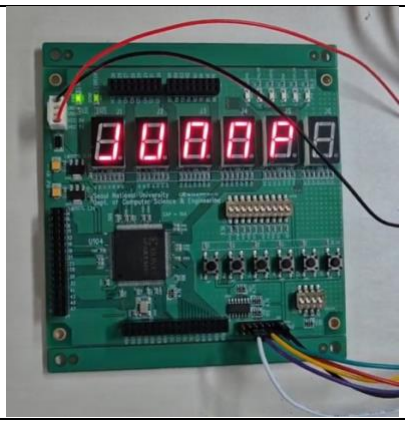


The waveform diagram displays the correct outputs of the instructions given by IMEM. After the RESET signal the addresses of the instructions increase sequentially unless the jump operation is executed. All 4 operations are shown to be working properly, and their outputs match the expected outputs shown in *Figure 27*. The add operation is shown at around 600ns where the instruction 2'b00011011 (s3 = s1 +s 2 = 3) is performed and the output in the segments is the 7 segment for 0 on segment 1 and the 7 segment for 3 in segment 2. The load operation is shown around 400ns where the instruction 2'b01100101 (s1 = m[s2+1] =2) is performed and the output in the segments is the 7 segment for 0 on segment 1 and the 7 segment for 2 on segment 2. The store operation is shown around 1000ns where the instruction 2'b10101110 (m[s2-1] = s3) is performed and the output displays "STORE" in 7 segments across the 5 segment outputs. The jump operation is shown around 800ns where the instruction 2'b11000001(jump) is performed and the output displays "JUMP". Additionally, we can see that the next address is shown to be 2'b00000110 instead of 2'b00000101 as this address was skipped.

By implementing CPUT.v, with some modifications so that it could be tested in an FPGA, we could observe the same results.

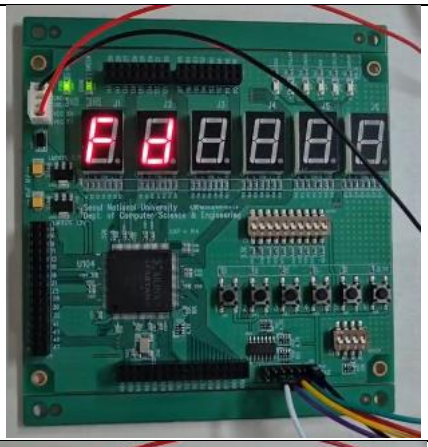
Figure 30: FPGA Results of CPUT.v

Add		Store	

Load		Jump	

Additional tests were also done such, as multiple jumps shown around 3,800ns, or to see if the added hexadecimal numbers(A,b,C,d,E) were expressed correctly. The results are shown below.

Figure 31: Expression of Hexadecimals.

A		d	
b		E	

C		F	

Extra Implementation

The extra implementation that was added in this microprocessor is the display of the words “JUMP” and “StOrE” when the jump operation and store operation are executed. To implement this feature, the module Output was created to initialize five 7-segment displays to display the words. *Figure 30* demonstrates how it is displayed in an FPGA. For the M in “JUMP” two 7 segment displays were used.

Conclusion

The 8-bit microprocessor was successfully created using Verilog HDl. This was accomplished by creating various core components such as Program Counter, Register, ALU, Control unit, and Data Memory and combining them based on the Data Path diagram that was given. Understanding how each component worked and how they were interconnected was crucial in completing the project. Not only understanding the diagrams provided but understanding our code was also important as in our implementation for the operations store and load we gave the ALUOP signal 1 instead of the proposed 0, as out ALU model only performs the addition operation when the ALUOP signal is 1. If the signal is 0 it would output 0. Additionally, creating separate modules such as IMEM and CPUT to simulate and debug the whole design were features that were not shown in the Lab guide thus the team had to think of ways to efficiently debug and test the microprocessor. The extra implementation also included the creation of the Output module which was something created for the sole reason to display jump and store when the operations were executing. This project reinforced everything we have learned during class and lab and provided valuable hands-on experience in creating a device that we are constantly surrounded with.

References

Katz, R. H., & Borriello, G. (2005). Contemporary Logic Design (2nd ed.). Prentice Hall.

David A. Patterson, John L. Hennessy. (2005). Computer Organization and Design (3rd ed., pp.306). Morgan Kaufmann.

"Different Instruction Cycles." *GeeksforGeeks*, www.geeksforgeeks.org/computer-organization-architecture/different-instruction-cycles/. Accessed 16 June 2025.