Logic Design Final Lab

Designing a Microprocessor

2024-13755 | 김연준

Class 3, Team 12

Dept. of Computer Science and Engineering

Submission files

1	LDLAB_250617_class003_team12_김연준_2024-13755.pdf	Final Lab report file
2	Proc.v	Code
3	proctest.v	Testbench Code
4	Alu.v	Code
5	DMem.v	Code
6	freqDivider.v	Code
7	HexTo7Seg.v	Code
8	Imem.v	Code
9	Opcode2Seg.v	Code
10	RegFile.v	Code
11	RegWriteTo7Seg.v	Code
12	Proc.ucf	User constraint file

1. Overall Design and Structure

The main module of our microprocessor is implemented in `Proc.v`.

```
module Proc(
    input [7:0] inst,
    output [7:0] readAddr,
    output [6:0] segOut16,
    output [6:0] segOut1,
    output [6:0] segPc16,
    output [6:0] segPc1,
    output [6:0] segOpcode,
    output negLED,
    input CLK_osc,
    input RST,
    input HALT
    );
```

The input and output are as shown above. It receives the oscillator's clock, and instruction as input, and outputs 7 segment signals for instruction result. There are also additional ports for my team's additional features.

1.1. Frequency divider

```
31      // *** Clock (Frequency Divider) ***
32      wire CLK;
33      wire haltCLK; // for additional feature
34      freqDivider fDiv (
35          .clr(RST),
36          .HALT(HALT),
37          .CLK(CLK_osc),
38          .CLKout(CLK),
39          .haltCLKout(haltCLK)
40      );
41
```

```
1   `timescale 1ns / 1ps
2
3   module freqDivider(
4       input clr,
5       input HALT,
6       input CLK,
7       output reg CLKout,
8       output reg haltCLKout
9       );
10
11      reg[31:0] cnt;
12      reg[31:0] haltCnt;
13
14      always @(posedge CLK) begin
15          if (clr) begin
16              cnt <= 32'd0;
17              haltCnt <= 32'd0;
18
19              CLKout <= 1'b1;
20              haltCLKout <= 1'b1;
21          end
22          else if(HALT == 0) begin // ordinary clock
23              if (cnt == 32'd0) begin // for simulation test
24              //if (cnt >= 32'd25000000) begin
25                  cnt <= 32'd0;
26                  CLKout <= ~CLKout;
27              end
28              else begin
29                  cnt <= cnt + 1;
30              end
31          end
```

We implemented the frequency divider in `freqDivider.v`. It converts a 50MHz oscillator input into a 1Hz clock signal. 'haltCLK' is something for one of our additional features. (see section 3 for details)

1.2. Decoding instructions and assigning control signals

```
67      assign iOpcode = inst[7:6];
68      assign iRs = inst[5:4];
69      assign iRt = inst[3:2];
70      assign iRd = inst[1:0];
71      assign iImm = {{6{inst[1]}}, inst[1:0]}; // imm = signExtend(inst[1:0])
72
73      // (2) Control Signals
74      wire [7:0] control;
75      wire cRegDst;
76      wire cRegWrite;
77      wire cALUSrc;
78      wire cBranch;
79      wire cMemRead;
80      wire cMemWrite;
81      wire cMemtoReg;
82      wire cALUOp;
83      assign {cRegDst, cRegWrite, cALUSrc, cBranch,
84              cMemRead, cMemWrite, cMemtoReg, cALUOp} = control;
85      assign control =
86         (iOpcode == 2'b00) ? 8'b11000001 :     // add
87         (iOpcode == 2'b01) ? 8'b01101010 :     // lw
88         (iOpcode == 2'b10) ? 8'b00100100 :     // sw
89         (iOpcode == 2'b11) ? 8'b00010000 : 0;  // j
```

This part of `Proc.v` decodes the instruction input into multiple variables and assigns control signals (variables with ‘c’ prefixes) according to the opcode.

1.3. The register file

```
91      // (3) Register File
92      wire [7:0] read1;
93      wire [7:0] read2;
94      wire [1:0] wIdx;
95      wire [7:0] wData;
96
97      // * Additional Feature *
98      //Connecting MSB of wData to negLED, c
99      assign negLED = wData[7];
100
101     RegFile rFile(
102        .rIdx1(iRs),
103        .rIdx2(iRt),
104        .write(cRegWrite),
105        .wIdx(wIdx),
106        .wData(wData),
107        .read1(read1),
108        .read2(read2),
109        .CLK(CLK),
110        .RST(RST)
111     );
112
```

```
1   `timescale 1ns / 1ps
2
3   module RegFile(
4       input [1:0] rIdx1,
5       input [1:0] rIdx2,
6        input write, // write enable
7       input [1:0] wIdx,
8       input [7:0] wData,
9       output [7:0] read1,
10      output [7:0] read2,
11       input CLK,
12       input RST
13      );
14
15       reg [7:0] regFile [3:0];
16
17       assign read1 = regFile[rIdx1];
18       assign read2 = regFile[rIdx2];
19
20       // Initialize Register File
21       always @ (posedge CLK or posedge RST) begin
22          if(RST == 1) begin
23              regFile[0] = 8'b0;
24              regFile[1] = 8'b0;
25              regFile[2] = 8'b0;
26              regFile[3] = 8'b0;
27          end
28          else begin
29              if(write) begin
30                  $display("wData = %d, wIdx = %d", wData, wIdx);
31                  regFile[wIdx] = wData;
32              end
33          end
34       end
35
36  endmodule
37
```

The register file module is defined at `RegFile.v`. The RegFile module has 4 data registers. The module is synchronized to the processor clock and handles data write and read.

1.4. The ALU

```
`timescale 1ns / 1ps

module Alu(
    input [7:0] val1,
    input [7:0] val2,
    input aluOp,
    output [7:0] result
    );

    wire [7:0] res;
    wire [7:0] v1, v2;

    assign v1 = val1;
    assign v2 = val2;
    assign result = res;

    assign res = v1 + v2;

    always@(res) begin
        //$display("%d+ %d = %d", v1, v2, res);
    end
endmodule

```

The part of `Proc.v` at the top and `ALU.v` at the bottom implements the ALU. ALU is a purely combinational module and was implemented with data flow style description.

1.5. The Data Memory

```
    // (5) Data Memory
    wire [7:0] memAddr;
    wire [7:0] memWData;
    wire [7:0] memRData;

    assign memWData = read2;
    assign memAddr = aluOut;

    DMem dMem(
        .Address(memAddr),
        .writeData(memWData),
        .readData(memRData),
        .MemRead(cMemRead),
        .MemWrite(cMemWrite),
        .CLK(CLK),
        .RST(RST)
    );
```

```
`timescale 1ns / 1ps

module DMem(
    input [7:0] Address,
    input [7:0] writeData,
    output [7:0] readData,
    input MemRead,
    input MemWrite,
    input CLK,
    input RST
    );

    reg [7:0] mem [0:31];
    wire [7:0] wD;
    wire [7:0] rD;
    wire [7:0] addr;

    assign addr = Address;
    assign wD = writeData;
    assign readData = rD;

    assign rD = (MemRead==1) ? mem[addr] : 0;

    integer i = 0;
    always @(posedge CLK or posedge RST) begin
        if (RST == 1) begin
            for(i = 0; i < 16; i = i+1) begin
                mem[i] = i;
                mem[16+i] = -i;
                //$display("%d", mem[16+i]);
            end
        end
        else begin
            if(MemWrite == 1) begin
                $display("wData: %d", wD);
                mem[addr] = wD;
            end
        end
    end


endmodule

```

The part of `Proc.v` at the top and `DMem.v` at the bottom implements the data memory. This part manages the reading/writing of data to the data memory. The DMem module is synchronized to the processor's clock.

1.6. Writing back to register file and output

```
148     // (6-1) Write Back
149     assign wIdx = (cMemtoReg == 0) ?
150             iRd :   // (op == add) rd
151             iRt;        // (op == lw) rt
152     assign wData =
153             (iOpcode == 2'b00) ? aluOut :    // (op == add) rs+rt
154             (iOpcode == 2'b01) ? memRData :  // (op == lw)  Mem[rs+imm]
155             8'b0;  // (op == sw) or (op == j)
156
157
158     // (6-2) 7Seg output converter
159     RegWriteTo7Seg to7segOut (
160        .reg_data(wData),
161        .seg_high(sOut16),
162        .seg_low(sOut1)
163     );
164
```

The part of `Proc.v` shown above controls the register write index and data signals according to opcode and operation results. Also, the write data (wData) is displayed at two 7-segment cells. In the final lab announcement pdf file, it was not specified what we should display for 'sw' and 'j' instruction results. So, our processor just always displays '00' for 'sw' and 'j' instructions.

```
21  module RegWriteTo7Seg(
22      input [7:0] reg_data,
23      output [6:0] seg_high,
24      output [6:0] seg_low
25      );
26
27      wire [3:0] high = reg_data[7:4];
28      wire [3:0] low  = reg_data[3:0];
29
30      HexTo7Seg high_display (
31         .hex(high),
32         .seg(seg_high)
33      );
34
35      HexTo7Seg low_display (
36         .hex(low),
37         .seg(seg_low)
38      );
39
40  endmodule
41
```

```
21  module HexTo7Seg(
22      input [3:0] hex,           // 4-bit hex imput
23      output reg [6:0] seg    // 7-seg output (a~g)
24      );
25
26      always @(*) begin
27         case(hex)
28             4'h0: seg = 7'b111_1110; // abc_defg
29             4'h1: seg = 7'b011_0000;
30             4'h2: seg = 7'b110_1101;
31             4'h3: seg = 7'b111_1001;
32
33             4'h4: seg = 7'b011_0011;
34             4'h5: seg = 7'b101_1011;
35             4'h6: seg = 7'b101_1111;
36             4'h7: seg = 7'b111_0000;
37
38             4'h8: seg = 7'b111_1111;
39             4'h9: seg = 7'b111_1011;
40             4'hA: seg = 7'b111_0111;
41             4'hB: seg = 7'b001_1111;
42
43             4'hC: seg = 7'b100_1110;
44             4'hD: seg = 7'b011_1101;
45             4'hE: seg = 7'b100_1111;
46             4'hF: seg = 7'b100_0111;
47             default: seg = 7'b000_0000; // all segments off
48         endcase
49      end
50
51  endmodule
52
```

The conversion to binary data to hexadecimal 7-segment output is performed by the 'RegWriteTo7Seg' module. It is defined at `RegWriteTo7Seg.v` as shown above. it comprises two submodules 'HexTo7Seg', which is defined at `HexTo7Seg.v`.

1.7. Handling RST input and Updating PC

```
179     // *** Behavior ***
180
181     always @(posedge CLK or posedge RST) begin
182       if(RST == 1) begin
183       // Initialize / Reset processor
184           pc <= 8'b0;
185       end
186       else begin
187       // Update PC
188           $display("pc: %d", pc);
189           pc <= (cBranch==0) ?
190               (pc+1) :            // (op != j)
191               (pc+1 + iImm);  // (op == j)
192       end
193     end
194
195 endmodule
```

The part of `Proc.v` shown above describes the synchronous behavior of our processor. The next PC is determined regarding whether the instruction is 'j' or not. The RST (Reset) signal is given asynchronously and resets the processor immediately. The synchronous submodules of Proc (RegFile and DMem) also share the RST signal, and they are also reset as the signal is given.

2. Simulation and FPGA Implementation Results

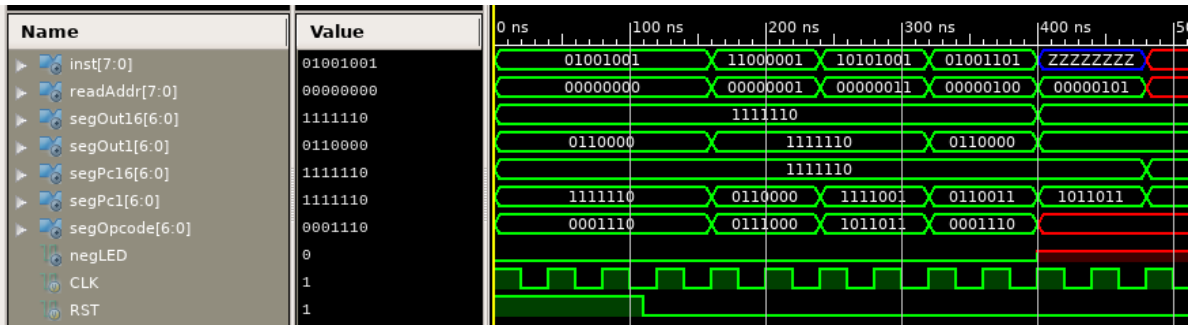
2.1. Simulation results

```
21  module imem(
22      output [7:0] instruction,
23      input [7:0] Read_Address
24      );
25
26      wire [7:0] MemByte[31:0];
27
28      // *** test program 1 ***
29
30      assign MemByte[0] = 8'b01001001;
31      assign MemByte[1] = 8'b11000001;
32      assign MemByte[2] = 8'b00011000;
33      assign MemByte[3] = 8'b10101001;
34      assign MemByte[4] = 8'b01001101;
35
```

```
70      assign instruction = MemByte[Read_Address];
71
72  endmodule
73
```

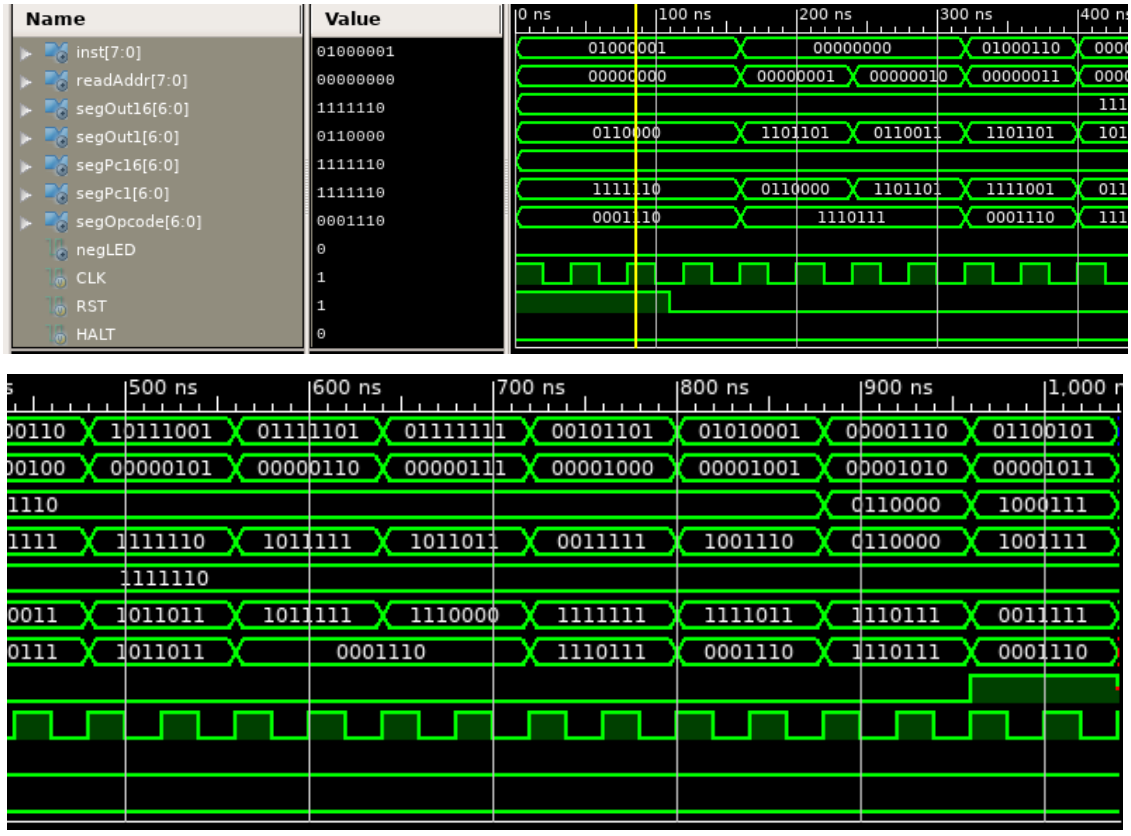
We tested out design with some programs we made according to the ISA. (The pseudocodes, machine codes, and the expected output of our testing programs are documented at **Appendix**.) The instruction memory module defined at `imem.v` shown above, replaces the role of the TA board used for actual testing. In all three programs we wrote, the output results (the values segOut1 and segOut16) were as anticipated. The simulation results – both waveform and console outputs - are shown below.

Simulation result for program 1.




```
pc:  0
wData =  1, wIdx = 2
pc:  1
pc:  3
wData:  1
pc:  4
wData =  1, wIdx = 3
pc:  5
pc:  x
pc:  x
```

Simulation result for program 2.



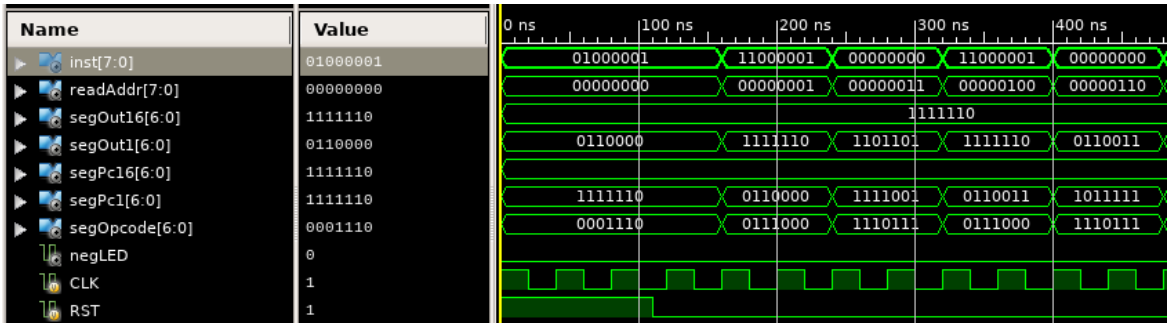
```
pc:  0
wData =  1, wIdx = 0
pc:  1
wData =  2, wIdx = 0
pc:  2
wData =  4, wIdx = 0
pc:  3
wData =  2, wIdx = 1
pc:  4
wData =  6, wIdx = 2
pc:  5
wData:  6
pc:  6
wData =  6, wIdx = 3
```

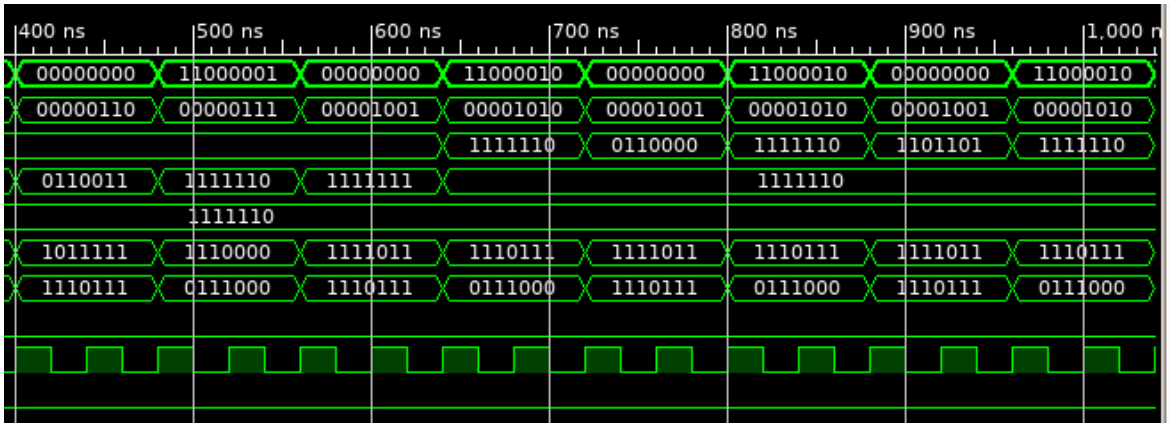
```
pc:  7
wData =  5, wIdx = 3
pc:  8
wData =  11, wIdx = 1
pc:  9
wData =  12, wIdx = 0
pc:  10
wData =  17, wIdx = 2

# run 1.00us
pc:  11
wData = 254, wIdx = 1
pc:  12
```

Console | Compilation Log

Simulation result for program 3.



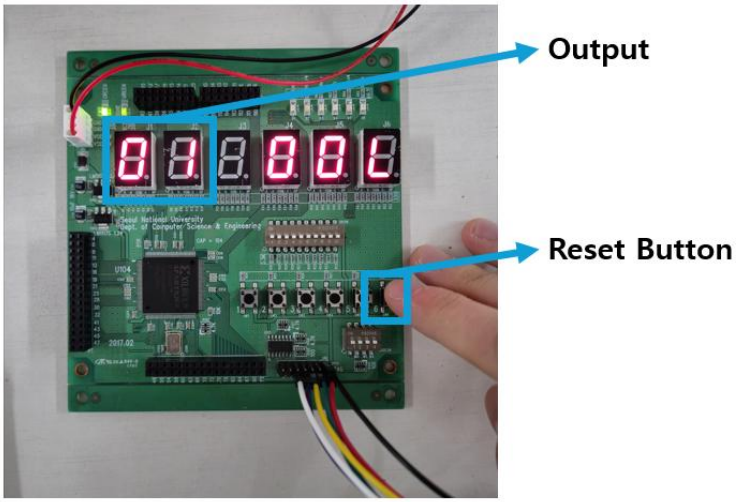



```
pc:  0
wData =  1, wIdx = 0
pc:  1
pc:  3
wData =  2, wIdx = 0
pc:  4
pc:  6
wData =  4, wIdx = 0
pc:  7
pc:  9
wData =  8, wIdx = 0
```

```
pc:  10
pc:  9
wData =  16, wIdx = 0
pc:  10
pc:  9
wData =  32, wIdx = 0

# run 1.00us
pc:  10
pc:  9
wData =  64, wIdx = 0
pc:  10
```

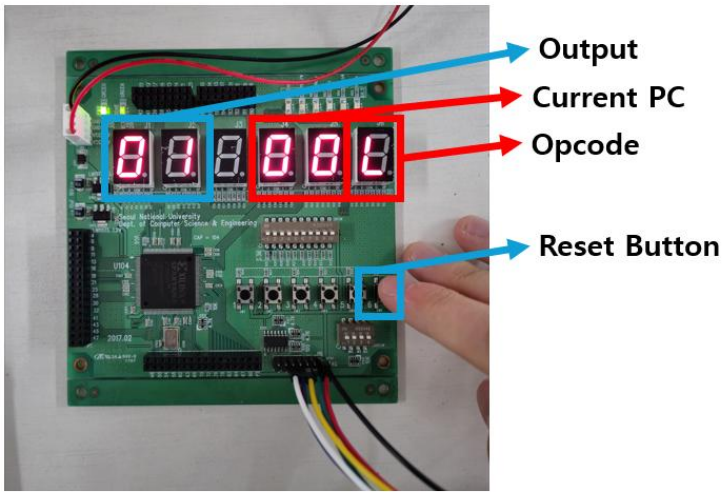
2.2. FPGA implementation results



The figure above shows the FPGA implementation result of our microprocessor. The output values are displayed at the first and second 7-segment. Also, the sixth tactile switch is the reset button. The test results for FPGA implementation were consistent with our simulation results.

3. Additional Features

3.1. PC and Opcode display

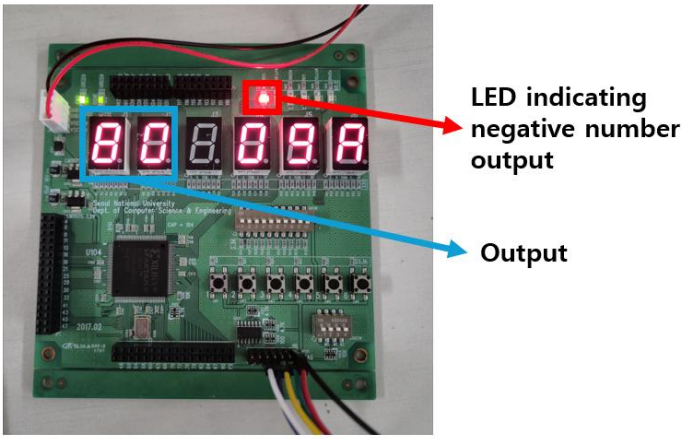



```
165     // (6-3) *Additional Feature* display PC to 7seg
166     RegWriteTo7Seg to7segPc (
167        .reg_data(pc),
168        .seg_high(sPc16),
169        .seg_low(sPc1)
170     );
171
172     // (6-3) *Additional Feature* output opcode to 7seg
173     Opcode2Seg opcode2seg (
174        .opcode(iOpcode),
175        .seg(op2seg)
176     );
```

```
23  // *Additional Feature* display opcode to 7seg
24
25  module Opcode2Seg(
26      input [1:0] opcode,
27      output [6:0] seg
28      );
29
30      assign seg =
31          (opcode == 2'b00) ? 7'b1110111 :    // 'A' (add)
32          (opcode == 2'b01) ? 7'b0001110 :    // 'L' (lw)
33          (opcode == 2'b10) ? 7'b1011011 :       // 'S' (sw)
34          (opcode == 2'b11) ? 7'b0111000 : 0; // 'J' (j)
35
36
37  endmodule
38
```

In addition to the output display, we put extra displays for the current PC and the opcode of current instruction. We showed the opcode as one of the symbolic characters (A, L, S, and J) through the Opcode2Seg module defined at `Opcode2Seg.v`. (Actually, this feature helped us debug the hardware).

3.2. Negative number output indicator

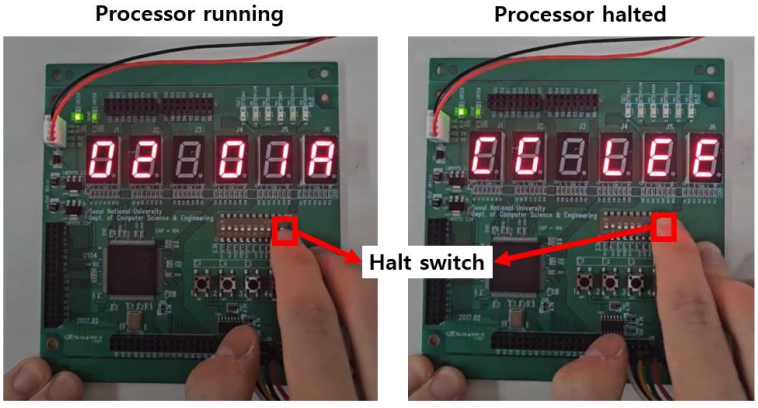



```
97      // * Additional Feature *
98      //Connecting MSB of wData to negLED, describing whether wData is negative.
99      assign negLED = wData[7];
100
```

We also added an LED display that indicates negative number outputs of an operation. The processor takes the MSB of the two's complement output and lights up the LED if the output is a negative number.

3.3. Processor interrupt switch

We added a switch that interrupts the processor and halts the execution of instructions. But we thought that just halting is quite bland. So, we designed it so that the text 'CG LEE' blinks on five of the 7-segment displays, during the halt state. We chose the text to express our respect and fondness for Professor Lee.



We needed a periodic clock signal to make the text blink. But the main clock is frozen, so we added a secondary clock 'haltCLK' which activates only at halt state. The code below shows the modified parts of `freqDivider.v` and `Proc.v`.

```v
module freqDivider(
    input clr,
    input HALT,
    input CLK,
    output reg CLKout,
    output reg haltCLKout
    );

    reg[31:0] cnt;
    reg[31:0] haltCnt;

    always @(posedge CLK) begin
        if (clr) begin
            cnt <= 32'd0;
            haltCnt <= 32'd0;

            CLKout <= 1'b1;
            haltCLKout <= 1'b1;
        end
        else if(HALT == 0) begin // ordinary clock
            if (cnt == 32'd0) begin // for simulation test
            //if (cnt >= 32'd25000000) begin
                cnt <= 32'd0;
                CLKout <= ~CLKout;
            end
            else begin
                cnt <= cnt + 1;
            end
        end
        else begin // clock for events when HALT is given
            if (haltCnt >= 32'd7500000) begin
                haltCnt <= 32'd0;
                haltCLKout <= ~haltCLKout;
            end
            else begin
                haltCnt <= haltCnt + 1;
            end
        end
    end
endmodule
```

```v
    assign segOut16 =  (HALT == 0) ? sOut16 : ((haltCLK) ? 7'b1001110 : 0); // C
    assign segOut1 =   (HALT == 0) ? sOut1     : ((haltCLK) ? 7'b1011110 : 0); // G
    assign segPc16 =   (HALT == 0) ? sPc16     : ((haltCLK) ? 7'b0001110 : 0); // L
    assign segPc1 =        (HALT == 0) ? sPc1  : ((haltCLK) ? 7'b1001111 : 0); // E
    assign segOpcode =     (HALT == 0) ? op2seg : ((haltCLK) ? 7'b1001111 : 0); // E
```

Appendix. Test case programs used to test our design

Test program #1 (from final lab pdf)

Address	Machine code	Meaning		Expected Output
0x00	0100 1001	S2 <- Mem[S0+1]	S2 <- 1	0x01
0x01	1100 0001	PC <- (PC+1) + 1	Goto 0x03	0x00
0x02	0001 1000	S0 <- S1 + S2	(Skipped)	
0x03	1010 1001	Mem[S2+1] <- S2	Mem[2] <- 1	0x00
0x04	0100 1101	S3 <- S0 + 1	S3 <- 1	0x01
0x05	-	End		-

Expected output sequence:

(Start) -> 0x01 -> 0x00 -> 0xxx -> 0x01 -> (End)

Test program #2 (store and load test)

Address	Machine code	Meaning		Expected Output
0x00	0100 0001	S0 <- Mem[S0+1]	S0 <- 1	0x01
0x01	0000 0000	S0 <- S0 + S0	S0 <- 2	0x02
0x02	0000 0000	S0 <- S0 + S0	S0 <- 4	0x04
0x03	0100 0110	S1 <- Mem[S0-2]	S1 <- 2	0x02
0x04	0000 0110	S2 <- S0 + S1	S2 <- 6	0x06
0x05	1011 1001	Mem[S3+1] <- S2	Mem[1] <- 6	0x00
0x06	0111 1101	S3 <- Mem[S3+1]	S3 <- 6	0x06
0x07	0111 1111	S3 <- Mem[S3-1]	S3 <- 5	0x05
0x08	0010 1101	S1 <- S2 + S3	S1 <- 11	0x0b
0x09	0101 0001	S0 <- Mem[S1+1]	S0 <- 12	0x0c
0x0a	0000 1110	S2 <- S0 + S3	S2 <- 17	0x0b
0x0b	0110 0101	S1 <- Mem[S2+1]	S1 <- (-2)	0xfe
0x0c	-	End		-

Expected output sequence:

(Start) -> 0x01 -> 0x02 -> 0x04 -> 0x02 -> 0x06 -> 0x00 -> 0x06 -> 0x05 -> 0x0b -> 0x0c -> 0x0b -> 0xfe -> (End)

Test program #3 (jump test)

Address	Machine code	Meaning		Expected Output
0x00	0100 0001	S0 <- Mem[S0+1]	S0 <- 1	0x01
0x01	1100 0001	PC <- (PC+1) + 1	Goto 0x03	0x00
0x02	1100 0001	PC <- (PC+1) + 1	(Skipped)	
0x03	0000 0000	S0 <- S0 + S0	S0 <- 2	0x02
0x04	1100 0001	PC <- (PC+1) + 1	Goto 0x06	0x00
0x05	1100 0001	PC <- (PC+1) + 1	(Skipped)	
0x06	0000 0000	S0 <- S0 + S0	S0 <- 4	0x04
0x07	1100 0001	PC <- (PC+1) + 1	Goto 0x09	0x00
0x08	1100 0001	PC <- (PC+1) + 1	(Skipped)	
0x09	0000 0000	S0 <- S0 + S0	S0 <- 8, 16, 32, ...	0x08, 0x10, 0x20, ...
0x0a	1100 0010	PC <- (PC+1) -2	Goto 0x09	0x00

Expected output sequence:

(Start) -> 0x01 -> 0x00 -> 0x02 -> 0x00 -> 0x04 -> 0x00 -> ***0x08 -> 0x00 -> 0x10 -> 0x00 -> 0x20 -> 0x00 -> 0x00 -> ... (Infinite loop)***

Test program #4 (jump and load test)

Address	Machine code	Meaning		Expected Output
0x00	0100 0001	S0 <- Mem[S0+1]	S0 <- 1	0x01
0x01	1100 0001	PC <- (PC+1) + 1	Goto 0x03	0x00
0x02	1100 0001	PC <- (PC+1) + 1	(Skipped)	
0x03	0101 1001	S2 <- Mem[S1+1]	S2 <- 1	0x01
0x04	0010 1010	S2 <- S2 + S2	S2 <- 2	0x02
0x05	0010 1010	S2 <- S2 + S2	S2 <- 4	0x04
0x06	0010 1010	S2 <- S2 + S2	S2 <- 8	0x08
0x07	0010 1010	S2 <- S2 + S2	S2 <- 16	0x10
0x08	0010 1110	S2 <- S2 + S3	S2 <- 17	0x11
0x09	0110 1101	S3 <- Mem[S2+1]	S3 <- -1	0xff
0x0a	0000 1100	S0 <- S0 + S3	S0 <- S0 - 1	0x00, 0xff, 0xfe, …
0x0b	1100 0010	PC <- (PC+1) - 2	Goto 0x0a	0x00

Expected output sequence:

(Start) -> 0x01 -> 0x00 -> 0x01 -> 0x02 -> 0x04 -> 0x08 -> 0x10 -> 0x11 -> 0xff -> 0x00 -> ***0x00 -> 0xff -> 0x00 -> 0xfe -> 0x00 -> 0xfd -> 0x00 -> 0xfc -> … (Infinite loop)***