Logic Design Final Project Report

2019-18499 김준혁

▶ Microprocessor의 하위 모듈 Verilog Code

```
21  module Freq_Div(
22      input CLK_IN,
23      input RST,
24      output reg CLK,
25      output reg [1:0] State
26      );
27
28      //50Mhz oscillator to 1hz Clock
29
30      reg [31:0] cnt;
31
32      initial begin
33          State <= 2'b00;
34      end
35
36      always @(posedge CLK_IN) begin
37          if(RST) begin
38              cnt <= 32'd0;
39              CLK <= 1'b0;
40              State <= 2'b00;
41          end else begin
42              if(cnt == 32'd25000000) begin
43                  cnt <= 32'd0;
44                  if(CLK == 1'b0) begin
45                      State <= 2'b00;
46                  end
47                  CLK <= ~CLK;
48              end else begin
49                  cnt <= cnt + 1;
50                  if(CLK == 1'b1) begin
51                      if(cnt == 32'd500) begin
52                          State <= 2'b01;
53                      end else if(cnt == 32'd1000) begin
54                          State <= 2'b10;
55                      end else if(cnt == 32'd1500) begin
56                          State <= 2'b11;
57                      end
58                  end
59              end
60          end
61      end
62
63  endmodule
64
```

(Frequency Divider의 Verilog Code - Freq_Div.v)

입력 : CLK_IN (50Mhz oscillator 입력), RST (리셋)

출력 : CLK (1hz Clock으로 변환된 출력), State (이에 따라 특정 모듈 활성화)

<간단한 모듈 설명>

FPGA 보드에 달려있는 50Mhz oscillator의 입력을 1hz로 변환시키며, 특정 시간이 지남에

따라 State를 변화시키는 모듈이다. 리셋 버튼을 누를 시에는 CLK이 초기화된다. oscillator가 2500만 번 진동 시에 CLK의 값을 바꿈으로써 1hz CLK을 생성한다.

기본 Frequency Divider와 다른 점은, 이 모듈 내에서 State를 시간에 따라 바꾼다는 것인데, 이는 Read/ Write와 관련된, Register와 Data Memory의 특정 기능 활성화의 딜레이로 인한 오류를 방지하기 위함이다. CLK가 0일 때 State는 00으로, CLK가 1이 되는 순간 이후 500회 진동마다 State를 01, 10, 11로 변화시켰다. State가 00 또는 11일 때는 특별한 모듈의 활성화가 없으며, 01, 10일 때 활성화가 일어나게 하였다. 자세한 내용은 관련 모듈 설명에서 추가로 설명할 것이다.

```
21  module State_Control(
22      input [1:0] State,
23      output State_Reg_Write,
24      output State_Memory
25      );
26
27      assign State_Reg_Write = State[1];
28      assign State_Memory = State[0];
29
30  endmodule
31
```

(State Control Unit의 Verilog Code - State_Control.v)
입력 : State
출력 : State_Reg_Write (레지스터 쓰기 활성화), State_Memory (메모리 읽기, 쓰기 활성화)

<간단한 모듈 설명>

Freq_Div에서 변화한 State를 이용해 어떠한 모듈을 활성화할지 컨트롤하는 유닛이다. State_Reg_Write은 레지스터에 쓰기를 활성화하는 것이며, State가 10일 때 1이 된다. State_Memory는 메모리의 읽기, 쓰기를 활성화하는 것이며, State가 01일 때 1이 된다.

```
21  module Control(
22      input [1:0] Opcode,
23      output [7:0] CtrlSign
24      );
25
26      ////////////////////////////////
27      //  CtrlSign Info             //
28      //  index   |   Meaning       //
29      //  7       |   RegDst        //
30      //  6       |   RegWrite      //
31      //  5       |   ALUSrc        //
32      //  4       |   Branch        //
33      //  3       |   MemRead       //
34      //  2       |   MemWrite      //
35      //  1       |   MemtoReg      //
36      //  0       |   ALUOP         //
37      ////////////////////////////////
38
39      assign CtrlSign = (Opcode == 2'b00) ? 8'b11000001 :
40                        (Opcode == 2'b01) ? 8'b01101010 :
41                        (Opcode == 2'b10) ? 8'b00100100 :
42                        8'b00010000;
43
44  endmodule
45
```

(Control Unit의 Verilog Code – Control.v)
입력 : Opcode (Instruction[7:6])
출력 : CtrlSign (Control Sign, 총 8가지)

<간단한 모듈 설명>
입력받는 Instruction[7:6]은 Opcode를 의미하며, 이 Opcode에 따라 활성화할 Control Signal을 출력하며, 이는 pdf에 있는 내용을 참고했다. 출력하는 CtrlSign은 7:0의 배열로 정의했으며, 7번부터 0번까지 각각 RegDst, RegWrite, ALUSrc, Branch, MemRead, MemWrite, MemtoReg, ALUOP를 의미한다. 각 Opcode에 따른 출력할 CtrlSign은 위의 코드와 같다. SW(Opcode = 10), J(Opcode = 11)의 경우에는 RegDst와 MemtoReg를 X(상관없음)로 pdf에 설명되어 있지만, 혼란을 방지하고자 우리 코드에서는 0으로 만들기로 하였다.

```
21  module MUX(
22      input [7:0] NumA,
23      input [7:0] NumB,
24      input Select,
25      output [7:0] Result
26      );
27
28      assign Result = (Select == 1'b0) ? NumA : NumB;
29
30  endmodule
31
```

(Multiplexer의 Verilog Code – MUX.v)
입력 : NumA (8bits), NumB (8bits), Select
출력 : Result (8bits)

<간단한 모듈 설명>
8비트로 이루어진 두 값을 입력을 받고, Select가 지정하는 위치의 값을 출력하는 모듈이다. Select가 0일 때 Result로 출력되는 것은 NumA, Select가 1일 때 NumB가 출력된다.

```
21  module PC(
22      input CLK,
23      input RST,
24      input [7:0] NewPC,
25      output reg [7:0] PCout
26      );
27
28      reg ready;
29
30      initial begin
31          PCout <= 0;
32          ready <= 0;
33      end
34
35      always @(posedge CLK or posedge RST) begin
36          if(RST) begin
37              PCout <= 0;
38              ready <= 0;
39          end else begin
40              if(ready == 1'b0) begin
41                  ready <= 1;
42              end else begin
43                  PCout <= NewPC;
44              end
45          end
46      end
47
48  endmodule
49
```

(PC 모듈의 Verilog Code - PC.v)
입력 : CLK (1hz Clock), RST (리셋), NewPC (다음 PC)
출력 : PCout (PC 출력)

<간단한 모듈 설명>
다음 PC를 입력받아 이를 CLK가 0에서 1이 될 때 기존 PC를 이로 교체하는 모듈이다. RST이 1일 때는 PC가 초기화되며,

```
21  module Reg_Data_Select_Converter(
22      input [3:0] IN,
23      output reg [1:0] OUT
24      );
25
26      always begin
27          if(IN[0] == 1) begin
28              OUT = 2'b00;
29          end else if(IN[1] == 1) begin
30              OUT = 2'b01;
31          end else if(IN[2] == 1) begin
32              OUT = 2'b10;
33          end else if(IN[3] == 1) begin
34              OUT = 2'b11;
35          end
36      end
37
38  endmodule
39
```

(Register Data Select Converter의 Verilog Code - Reg_Data_Select_Converter.v)
입력 : IN (4bits)
출력 : OUT (2bits)

<간단한 모듈 설명>
보고 싶은 레지스터를 선택하는 입력(버튼 4개로 이루어짐)을 2비트의 레지스터 주소값으로 변환시키는 모듈이다. 사실상 인코더라 봐도 무방하다. 4비트의 버튼입력을 2비트로 바꾸어준다.

```
21  module Registers(
22      input CLK,
23      input RST,
24      input State_Reg_Write,
25      input [1:0] Read_Reg1,
26      input [1:0] Read_Reg2,
27      input [1:0] Write_Reg,
28      input [7:0] Reg_Write_Data,
29      input RegWrite,
30      input [1:0] Reg_Data_Select,
31      output [7:0] Reg_Data,
32      output [7:0] Read_Data1,
33      output [7:0] Read_Data2,
34      output reg [7:0] Reg_Write_Data_Save
35      );
36
37      reg [7:0] register[3:0];
38
39      // Initialiize
40      initial begin
41          register[0] = 0;
42          register[1] = 0;
43          register[2] = 0;
44          register[3] = 0;
45      end
46
47      assign Read_Data1 = (Read_Reg1 == 2'b00) ? register[0] :
48                          (Read_Reg1 == 2'b01) ? register[1] :
49                          (Read_Reg1 == 2'b10) ? register[2] :
50                          register[3];
51
52      assign Read_Data2 = (Read_Reg2 == 2'b00) ? register[0] :
53                          (Read_Reg2 == 2'b01) ? register[1] :
54                          (Read_Reg2 == 2'b10) ? register[2] :
55                          register[3];
56
57      assign Reg_Data = (Reg_Data_Select == 2'b00) ? register[0] :
58                        (Reg_Data_Select == 2'b01) ? register[1] :
59                        (Reg_Data_Select == 2'b10) ? register[2] :
60                        (Reg_Data_Select == 2'b11) ? register[3] :
61                        8'b00000000;
62
63      always @(posedge RST or posedge State_Reg_Write) begin
64          if(RST) begin
65              register[0] = 0;
66              register[1] = 0;
67              register[2] = 0;
68              register[3] = 0;
69          end else begin
70              Reg_Write_Data_Save = Reg_Write_Data;
71              if(RegWrite == 1'b1) begin
72                  register[Write_Reg] = Reg_Write_Data;
73              end
74          end
75      end
76
77  endmodule
```

(Registers의 Verilog Code – Registers.v)

입력 : CLK (1hz CLK), RST (리셋), State_Reg_Write (레지스터 쓰기 활성화 신호) Read_Reg1 (Register1(=RS)의 주소), Read_Reg2 (Register2(=RT)의 주소), Write_Reg (Write Register(=RD)의 주소), Reg_Write_Data (Write Register에 쓸 값), RegWrite (Register에 값을 쓸 것인지에 대한 CtrlSign), Reg_Data_Select (보고 싶은 레지스터 주소)

출력 : Reg_Data (보고 싶은 레지스터의 값), Read_Data1 (Register1(=RS)에 저장되어 있던

값), Read_Data2 (Register2(=RT)에 저장되어 있던 값, Reg_Write_Data_Save (레지스터에 쓴 값을 저장한 것)

<간단한 모듈 설명>

레지스터 모듈이다. 처음에 모두 0이라 초기화된다. Read_Data1은 Read_Reg1이 의미하는 주소의 레지스터값을 출력하며, Read_Data2는 Read_Reg2 주소의 레지스터값을 출력한다. Reg_Data는 버튼을 눌러 들어온 신호를 2비트로 인코드한 값에 해당하는 주소의 레지스터 값을 출력한다.

RST가 1로 입력되면, 모든 레지스터가 0으로 초기화되며, State_Reg_Write가 0에서 1이 되는 시점에서 RegWrite 신호가 1이라면, 들어온 Reg_Write_Data 값을 Write_Reg가 의미하는 주소의 레지스터에 저장하며, Reg_Write_Data 값을 출력을 위한 Reg_Write_Data_Save에 또한 저장한다(이는 RegWrite 신호와 관계없다).

```
21  module ALU(
22      input [7:0] NumA,
23      input [7:0] NumB,
24      input ALUOP,
25      output [7:0] Result
26      );
27
28      assign Result = NumA + NumB;
29
30  endmodule
31
```

(ALU의 Verilog Code - ALU.v)
입력 : NumA (8bits), NumB (8bits), ALUOP
출력 : Result (8bits)

<간단한 모듈 설명>

본래는 ALUOP 신호에 따라 8비트로 이루어진 두 입력을 더해서 출력하는 모듈이지만, 우리 코드에서는 ALUOP에 관련없이 항상 두 값을 더해서 출력한다.

```
21  module v8_to_5_Compressor(
22      input [7:0] Address8bits,
23      output [4:0] Address5bits
24      );
25
26      assign Address5bits = Address8bits[4:0];
27
28  endmodule
29
```

(Compressor (8bits->5bits) 의 Verilog Code - v8_to_5_Compressor.v)
입력 : Address8bits (메모리 주소 8비트 입력)
출력 : Address5bits (메모리 주소 5비트 출력)

<간단한 모듈 설명>

ALU로부터 나온 결과가 데이터 메모리의 주소값으로 들어가게 되는데, 이는 8비트이기 때문에 32개의 데이터 메모리의 인덱스를 초기화하는 경우가 발생할 수 있다. 이를 방지하기 위해 8비트 주소값을 5비트로 압축하는 모듈이다.

```
21  module Data_Memory(
22      input CLK,
23      input RST,
24      input State_Memory,
25      input [4:0] Address,
26      input [7:0] Write_Data,
27      input MemRead,
28      input MemWrite,
29      output reg [7:0] Read_Data
30      );
31
32      reg [7:0] mem[31:0];
33
34      initial begin
35          mem[0] <= 0;
36          mem[1] <= 1;
37          mem[2] <= 2;
38          mem[3] <= 3;
39          mem[4] <= 4;
40          mem[5] <= 5;
41          mem[6] <= 6;
42          mem[7] <= 7;
43          mem[8] <= 8;
44          mem[9] <= 9;
45          mem[10] <= 10;
46          mem[11] <= 11;
47          mem[12] <= 12;
48          mem[13] <= 13;
49          mem[14] <= 14;
50          mem[15] <= 15;
51          mem[16] <= 0;
52          mem[17] <= -1;
53          mem[18] <= -2;
54          mem[19] <= -3;
55          mem[20] <= -4;
56          mem[21] <= -5;
57          mem[22] <= -6;
58          mem[23] <= -7;
59          mem[24] <= -8;
60          mem[25] <= -9;
61          mem[26] <= -10;
62          mem[27] <= -11;
63          mem[28] <= -12;
64          mem[29] <= -13;
65          mem[30] <= -14;
66          mem[31] <= -15;
67      end
```

```
68
69      always @(posedge RST or posedge State_Memory) begin
70          if(RST) begin
71              mem[0] <= 0;
72              mem[1] <= 1;
73              mem[2] <= 2;
74              mem[3] <= 3;
75              mem[4] <= 4;
76              mem[5] <= 5;
77              mem[6] <= 6;
78              mem[7] <= 7;
79              mem[8] <= 8;
80              mem[9] <= 9;
81              mem[10] <= 10;
82              mem[11] <= 11;
83              mem[12] <= 12;
84              mem[13] <= 13;
85              mem[14] <= 14;
86              mem[15] <= 15;
87              mem[16] <= 0;
88              mem[17] <= -1;
89              mem[18] <= -2;
90              mem[19] <= -3;
91              mem[20] <= -4;
92              mem[21] <= -5;
93              mem[22] <= -6;
94              mem[23] <= -7;
95              mem[24] <= -8;
96              mem[25] <= -9;
97              mem[26] <= -10;
98              mem[27] <= -11;
99              mem[28] <= -12;
100             mem[29] <= -13;
101             mem[30] <= -14;
102             mem[31] <= -15;
103         end else begin
104             if(MemWrite == 1'b1) begin
105                 mem[Address] <= Write_Data;
106             end
107             if(MemRead == 1'b1) begin
108                 Read_Data <= mem[Address];
109             end
110         end
111     end
112
113 endmodule
```

(Data Memory의 Verilog Code – Data_Memory.v)

입력 : CLK (1hz CLK), RST (리셋), State_Memory (데이터 메모리 읽기 및 쓰기 활성화 신호), Address (5bits), Write_Data (데이터 메모리에 쓸 값), MemRead (메모리 읽기 신호), MemWrite (메모리 쓰기 신호)

출력 : Read_Data (해당 주소의 데이터 메모리에 저장된 값),

<간단한 모듈 설명>

데이터 메모리 모듈이다. 시작할 때, RST가 1일 때 초기화되며, State_Memory가 0에서 1이 될 때, MemWrite가 1이면 Address가 가리키는 메모리에 들어온 데이터(Write_Data)를 저장하며, MemRead가 1이면 Address가 가리키는 메모리에 들어있는 값을 읽어 Read_Data로 내보낸다.

```
21  module Sign_Extend(
22      input [1:0] IN,
23      output [7:0] OUT
24      );
25
26      assign OUT = (IN[1] == 1) ? 8'b11111100 + IN : 8'b00000000 + IN;
27
28  endmodule
29
```

(Sign Extension의 Verilog Code - Sign_Extend.v)
입력 : IN (2bits)
출력 : OUT (8bits)

<간단한 모듈 설명>
2비트 값을 8비트로 Sign Extend 해주는 모듈이다. Instruction[1:0]을 Imm으로 만드는데 사용되었다.

```
21  module v7_Segment(
22      input [3:0] hex,
23      output reg [6:0] seg
24      );
25
26      always begin
27          case(hex)
28              4'b0000 : seg <= 7'b0111111;
29              4'b0001 : seg <= 7'b0000110;
30              4'b0010 : seg <= 7'b1011011;
31              4'b0011 : seg <= 7'b1001111;
32              4'b0100 : seg <= 7'b1100110;
33              4'b0101 : seg <= 7'b1101101;
34              4'b0110 : seg <= 7'b1111101;
35              4'b0111 : seg <= 7'b0000111;
36              4'b1000 : seg <= 7'b1111111;
37              4'b1001 : seg <= 7'b1101111;
38              4'b1010 : seg <= 7'b1110111;
39              4'b1011 : seg <= 7'b1111100;
40              4'b1100 : seg <= 7'b0111001;
41              4'b1101 : seg <= 7'b1011110;
42              4'b1110 : seg <= 7'b1111001;
43              4'b1111 : seg <= 7'b1110001;
44          endcase
45      end
46
47  endmodule
```

(7-Segment의 Verilog Code - v7_Segment.v)
입력 : hex (4bits)
출력 : seg (7bits)

<간단한 모듈 설명>
HEX(16진수)로 들어온 입력을 0~9까지는 숫자, 10~15까지는 A~F로, 7-Segment로 나타낼 수 있도록 변환해주는 모듈이다. 이때 B와 D는 소문자로 표현하여 다른 숫자와의 혼란을 방지하였다.

▶ Microprocessor의 Verilog Code

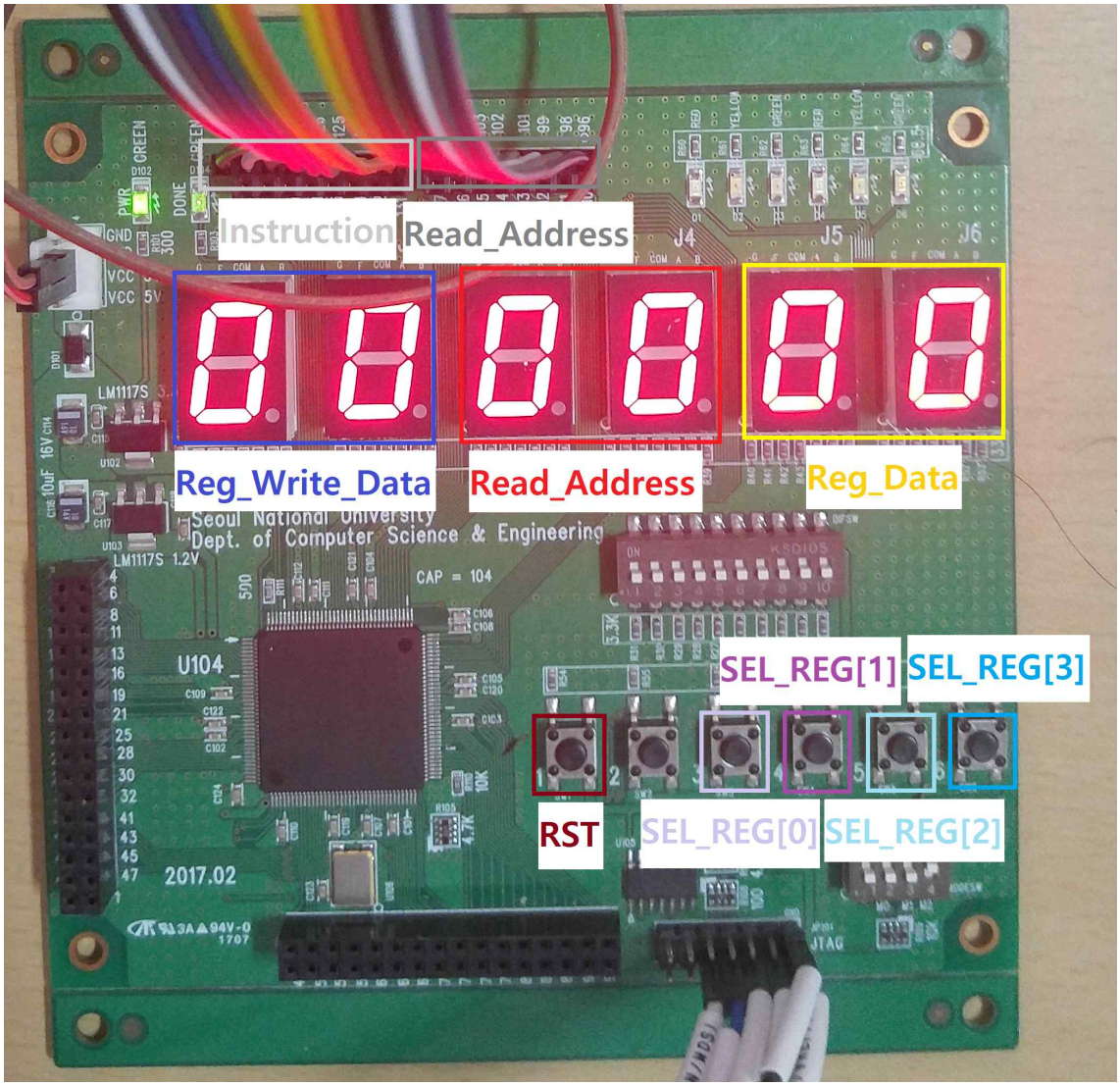
```
module Microprocessor(
    input [7:0] Instruction,
    input CLK_IN,
    input RST,
    input [3:0] SEL_REG,
    output [7:0] Read_Address,  //same as PC
    output [6:0] SEG1,  // bigger digit
    output [6:0] SEG2,  // smaller digit
    output [6:0] SEG3,  // PC bigger digit
    output [6:0] SEG4,  // PC smaller digit
    output [6:0] SEG5,  // test bigger digit
    output [6:0] SEG6   // test smaller digit
    );

    ////////////////////////////////////////
    // Instruction Info                   //
    //  index(Type) |   Meaning           //
    //  7:6 (all)   |   Opcode(op)        //
    //  5:4 (R, I)  |   SourceReg1(rs)    //
    //  3:2 (R, I)  |   SourceReg2(rt)    //
    //  1:0 (R)     |   DestinationReg(rd)//
    //  1:0 (I, J)  |   immediate(imm)    //
    ////////////////////////////////////////

    ////////////////////////////
    //  CtrlSign Info         //
    //  index   |   Meaning   //
    //  7       |   RegDst    //
    //  6       |   RegWrite  //
    //  5       |   ALUSrc    //
    //  4       |   Branch    //
    //  3       |   MemRead   //
    //  2       |   MemWrite  //
    //  1       |   MemtoReg  //
    //  0       |   ALUOP     //
    ////////////////////////////

    wire CLK;               // Just a Clock
    wire [1:0] State;       // For Control the timing
    wire [7:0] CtrlSign;    // Control Sign
    wire [7:0] Imm;         // Ummediate (constant)
    wire [7:0] Read_Data1;      // Result of Read register1
    wire [7:0] Read_Data2;      // Result of Read register2
    wire [7:0] ALU_Result;  // Result of ALU
    wire [7:0] Read_Data;   // Result of Data memory
    wire [7:0] Reg_Write_Data_Save; // What to must show
    wire State_Reg_Write;   // Write on register when this is 1
    wire State_Memory;      // activate memory r/w when this is 1
    wire [7:0] NewPC;
    wire [1:0] Write_Reg;
    wire [7:0] Reg_Write_Data;  //
    wire [7:0] ALU_Input2;
    wire [4:0] ALU_Result_5;        // Compressed ALU_Result
    wire [1:0] Reg_Data_Select;     //what reg to watch
    wire [7:0] Reg_Data;

    //CLK generator
    Freq_Div FD1(.CLK_IN(CLK_IN), .RST(RST), .CLK(CLK), .State(State)); //Get 1hz CLK

    //State Control
    State_Control SC1(.State(State), .State_Reg_Write(State_Reg_Write), .State_Memory(State_Memory));   //transform State into Specific Control

    //Control Unit
    Control C1(.Opcode(Instruction[7:6]), .CtrlSign(CtrlSign)); //Send CtrlUnit Opcode

    //NewPC Generator
    //NewPC_Generator NPG1(.Read_Address(Read_Address), .Branch(CtrlSign[4]), .Imm(Imm), .NewPC(NewPC));

     //PC
     PC P1(.CLK(CLK), .RST(RST), .NewPC(CtrlSign[4] ? (Read_Address + 8'b00000001 + Imm) : (Read_Address + 8'b00000001)), .PCout(Read_Address));

     assign Write_Reg = (CtrlSign[7] == 1) ? Instruction[1:0] : Instruction[3:2];

     MUX M2(.NumA(ALU_Result), .NumB(Read_Data), .Select(CtrlSign[1]), .Result(Reg_Write_Data));

     //Reg_Data_Select GEN
     Reg_Data_Select_Converter RDSC1(.IN(SEL_REG), .OUT(Reg_Data_Select));

     //Registers
     Registers R1(.CLK(CLK), .RST(RST), .State_Reg_Write(State_Reg_Write), .Read_Reg1(Instruction[5:4]),
                 .Read_Reg2(Instruction[3:2]), .Write_Reg(Write_Reg), .Reg_Write_Data(Reg_Write_Data),
                 .RegWrite(CtrlSign[6]), .Read_Data1(Read_Data1), .Read_Data2(Read_Data2),
                 .Reg_Write_Data_Save(Reg_Write_Data_Save), .Reg_Data_Select(Reg_Data_Select),
                 .Reg_Data(Reg_Data));

     //Make ALU_Input2
     MUX M3(.NumA(Read_Data2), .NumB(Imm), .Select(CtrlSign[5]), .Result(ALU_Input2));

     //ALU
     ALU A1(.NumA(Read_Data1), .NumB(ALU_Input2), .ALUOP(CtrlSign[0]), .Result(ALU_Result));

     //Data Memory Address 8bits to 5bits
     v8_to_5_Compressor COMP1(.Address8bits(ALU_Result), .Address5bits(ALU_Result_5));

     //Data Memory
     Data_Memory D1(.CLK(CLK), .RST(RST), .State_Memory(State_Memory), .Address(ALU_Result_5),
                 .Write_Data(Read_Data2), .MemRead(CtrlSign[3]), .MemWrite(CtrlSign[2]), .Read_Data(Read_Data));

     //Sign Extension
     Sign_Extend SE1(.IN(Instruction[1:0]), .OUT(Imm));

     v7_Segment V1(.hex(Reg_Write_Data_Save[7:4]), .seg(SEG1));
     v7_Segment V2(.hex(Reg_Write_Data_Save[3:0]), .seg(SEG2));
     v7_Segment V3(.hex(Read_Address[7:4]), .seg(SEG3));
     v7_Segment V4(.hex(Read_Address[3:0]), .seg(SEG4));
     v7_Segment V5(.hex(Reg_Data[7:4]), .seg(SEG5));
     v7_Segment V6(.hex(Reg_Data[3:0]), .seg(SEG6));

endmodule
```

(Microprocessor의 Verilog Code – Microprocessor.v)

입력 : Instruction (8-bit Inst), CLK_IN (50Mhz oscillator), RST (Reset Button), SEL_REG (Select Register Button)

출력 : Read_Address (PC), SEG1 (Reg_Write_Data[7:4]), SEG2(Reg_Write_Data[3:0]), SEG3(Read_Address[7:4]), SEG4(Read_Address[3:0]), SEG5(Reg_Data[7:4]), SEG6(Reg_Data[3:0])

<간단한 모듈 설명>



8-bit Instruction 입력을 받고, 50Mhz 진동을 1hz로 변환한 것을 이용해, 1초마다 하나씩 Instruction의 명령을 수행하는 마이크로프로세서다. 리셋 버튼을 누르면 초기화된다. 7-Segment 출력은 맨 왼쪽 2개는 Reg_Write_Data(정확히는 그 값을 저장한 Reg_Write_Data_Save), 가운데 2개는 Read_Address(현재 PC), 맨 오른쪽 2개는 보려고 한 레지스터에 들어있는 값을 보여준다. 이 레지스터 선택은 오른쪽 버튼 4개를 이용해 선택할 수 있다.

이 마이크로프로세서는 기본적으로 pdf에 나오는 형태대로 wire들을 연결해주었다. 대부분

의 모듈과 들어간 변수들은 하위 모듈에서 대부분 설명했기 때문에 생략한다. 이에 설명되지 않은 부분만 설명하려고 한다.

원래 NewPC_Generator라는 모듈을 이용하려 했지만, 찾지 못한 오류 때문에 주석 처리하여 비활성화 시켰으며, 이에 대한 내용은 아래에 추가하였다. 대신 PC 모듈에 들어갈 NewPC에 본래 들어갈 모듈의 결과를 직접 계산하여 들어가게 하였다.

Write_Reg는 MUX를 이용해 구할 수 있지만, 직접 만든 MUX.v는 8비트 입력을 받기 때문에, 이를 변환해주는 과정을 생략하기 위해 assign을 통해 직접 계산하여 정의했다.

이름이 M3인 MUX는 ALU에 들어갈 두 번째 값(NumB)을 Read_Data2와 Imm 중 선택하는 역할을 한다.

▶ 찾지 못한 오류로 제외된 Verilog 코드

```
21  module ADD(
22      input [7:0] NumA,
23      input [7:0] NumB,
24      output [7:0] Result
25      );
26
27      assign Result = NumA + NumB;
28
29  endmodule
30
```

(Adder의 Verilog Code - ADD.v)
입력 : NumA (8bits), NumB (8bits)
출력 : Result (8bits)

<간단한 모듈 설명>

8비트로 이루어진 두 값을 입력을 받고, 두 값을 더한 결과를 내보내는 Adder 모듈이다. 아래의 NewPC_Generator가 오류로 인해 제외되면서, ADD 모듈이 쓰이는 곳이 없어 제외되었다.

```
21  module NewPC_Generator(
22      input [7:0] Read_Address,
23      input Branch,
24      input [7:0] Imm,
25      output [7:0] NewPC
26      );
27
28      wire [7:0] Result_A1;
29      wire [7:0] Result_A2;
30
31      ADD A1(.NumA(Read_Address), .NumB(8'b00000001), .Result(Result_A1));    //Read_Address + 1
32      ADD A2(.NumA(Result_A1), .NumB(Imm), .Result(Result_A2));               //Read_Address + 1 + Imm
33      MUX M1(.NumA(Result_A1), .NumB(Result_A2), .Select(Branch), .Result(NewPC));    //Select between up two
34
35  endmodule
36
```

(New PC Generator의 Verilog Code - NewPC_Generator.v)
입력 : Read_Address (현재 PC), Branch, Imm
출력 : NewPC (다음 PC)

<간단한 모듈 설명>

다음 PC를 생성하는 모듈로, 2개의 ADD와 1개의 MUX로 이루어져 있다. ADD를 통해 현재 PC+1과 현재 PC+1+Imm을 생성하여, MUX에서 Branch 신호에 따라 둘 중 하나의 값을 다음 PC로 선택하는 모듈이었다. 하지만, J 명령에서 알 수 없는 오류를 일으켜 제외되었다.

▶ 시뮬레이션을 통한 Microprocessor 확인

```
21  module InstructionMem10(
22      input [7:0] Read_Address,
23      output [7:0] Instruction
24      );
25
26       wire [7:0] MemByte [31:0];
27
28       assign MemByte[0]=8'b01000100;
29       assign MemByte[1]=8'b01001001;
30       assign MemByte[2]=8'b00011001;
31       assign MemByte[3]=8'b10000100;
32
33       assign MemByte[4]=8'b01000100;
34       assign MemByte[5]=8'b01001001;
35       assign MemByte[6]=8'b00011001;
36       assign MemByte[7]=8'b10000100;
37
38       assign MemByte[8]=8'b01000100;
39       assign MemByte[9]=8'b01001001;
40       assign MemByte[10]=8'b00011001;
41       assign MemByte[11]=8'b10000100;
42
43       assign MemByte[12]=8'b01000100;
44       assign MemByte[13]=8'b01001001;
45       assign MemByte[14]=8'b00011001;
46       assign MemByte[15]=8'b10000100;
47
48       assign MemByte[16]=8'b01000100;
49       assign MemByte[17]=8'b01001001;
50       assign MemByte[18]=8'b00011001;
51       assign MemByte[19]=8'b10000100;
52
53       assign MemByte[20]=8'b11000011;
54
55
56       assign Instruction = MemByte[Read_Address];
57
58  endmodule
```

(Microprocessor를 시뮬레이션한 코드)

Instruction	해석	PC	Reg[0]	Reg[1]	Reg[2]	Reg[3]	RWD	M[0]	M[1]
01 00 01 00	lw s1, 0(s0)	0	0	0	0	0	0	0	1
01 00 10 01	lw s2, 1(s0)	1	0	0	1	0	1	0	1
00 01 10 01	add s1, s1, s2	2	0	1	1	0	1	0	1
10 00 01 00	sw s1, 0(s0)	3	0	1	1	0	x	1	1
	lw s1, 0(s0)	4	0	1	1	0	1	1	1
	lw s2, 1(s0)	5	0	1	1	0	1	1	1
	add s1, s1, s2	6	0	2	1	0	2	1	1
	sw s1, 0(s0)	7	0	2	1	0	x	2	1
	lw s1, 0(s0)	8		2	1		2	2	
	lw s2, 1(s0)	9		2	1		1	2	
	add s1, s1, s2	10		3	1		3	2	
	sw s1, 0(s0)	11		3	1		x	3	
	lw s1, 0(s0)	12		3	1		3	3	
	lw s2, 1(s0)	13		3	1		1	3	
	add s1, s1, s2	14		4	1		4	3	
	sw s1, 0(s0)	15		4	1		x	4	
	lw s1, 0(s0)	16		4	1		4	4	
	lw s2, 1(s0)	17		4	1		1	4	
	add s1, s1, s2	18		5	1		5	4	
	sw s1, 0(s0)	19		5	1		x	5	
	jump -1	20		5	1		x	5	

(위의 코드를 정리하고, 어떤 값이 나와야 하는지를 나타낸 것)

일단 본 시뮬레이션 코드를 돌려본 결과는 첨부한 동영상을 통해 볼 수 있다. Reg 값을 버튼을 눌러서 체크하는 것 또한 확인할 수 있다.

영상의 마지막 jump 부분에서 PC가 FF가 되는 문제가 있다. 이는 처음에 별거 아니라 생각하고 넘어간 불찰이 있었다. 6월 19일 실습실에서 재테스트를 보기 전에 영상을 찍었으며, 해당 재테스트에서 점프 부분에 문제가 생긴다는 것을 알고, 수정하였기 때문에, 해당 영상에서는 이러한 부분이 해결되기 전이라 완벽한 결과를 보여주지는 못했다. 더군다나 재테스트 이후 바로 FPGA 보드를 제출하여, 영상을 다시 찍지도 못하였다. 이러한 부분을 수정할 수는 없다는 아쉬운 점이 남았다.

다만 점프를 제외한 나머지 부분에서 명령의 실행과 읽기와 쓰기가 잘 되었다는 것을 확인할 수 있었다.