

Logic Design Final Project

공과대학 컴퓨터공학부
2021-16988 박재완

1 Structure

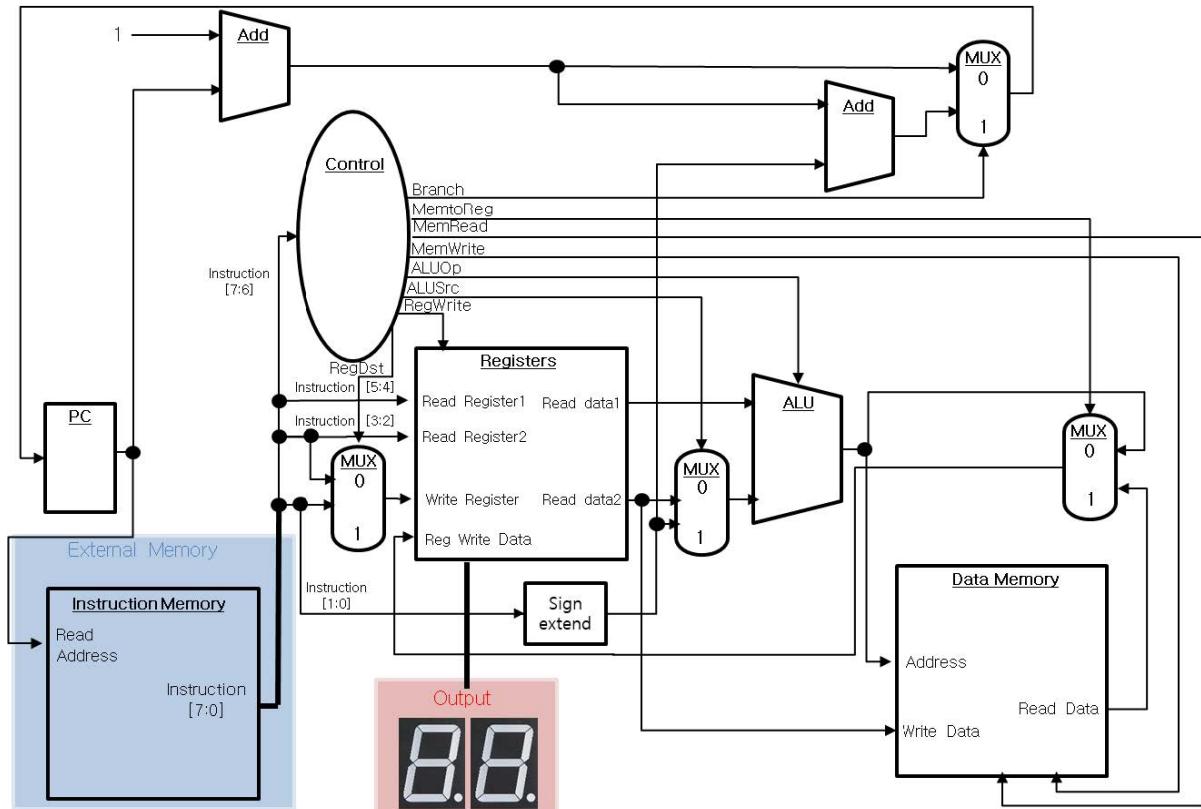


Figure 1: 구현한 microprocessor의 구조

위와 같은 구조를 갖는 microprocessor를 Verilog와 FPGA 보드를 이용하여 구현하였다. 전체를 구성하는 각 단위(Adder, MUX, Control Unit, Register, ALU, Data Memory, Program Counter)를 Verilog 모듈로 하나씩 정의하였고, 하나의 파일에서 이들을 전부 wire로 연결하여 정해진 작업을 수행할 수 있도록 하였다. Microprocessor가 하는 역할은, 외부로부터 받은 8bit instruction을 해석하여 명령에 따라 register 및 data memory를 변화시키고, 지정된 output을 7 segment display에 출력하는 것이었다. Instruction의 해석은 주어진 과제 스펙을 따랐다.

2 Components

2.1 Register

```
1 'timescale 1ns / 1ps
2
3 module register_module(
4     input [1:0] read_register_one,
5     input [1:0] read_register_two,
6     input [1:0] write_register,
7     input [7:0] write_data,
```

```

8   input RegWrite,
9   input CLK,
10  input reset,
11  output [7:0] read_data_one,
12  output [7:0] read_data_two
13 );
14
15 reg [7:0] registers [0:3];
16
17 integer i;
18 initial begin
19   for (i = 0; i < 4; i = i + 1)
20     registers[i] <= 0;
21 end
22
23 assign read_data_one = registers[read_register_one];
24 assign read_data_two = registers[read_register_two];
25
26 always @(posedge CLK or posedge reset) begin
27   if (reset) begin
28     for (i = 0; i < 4; i = i + 1)
29       registers[i] <= 0;
30   end else begin
31     if (RegWrite)
32       registers[write_register] <= write_data;
33   end
34 end
35
36 endmodule
37

```

Listing 1: register_module.v

7bit 데이터를 저장하고, 읽거나 수정이 가능한 register 모듈을 만들었다. 과제 스펙에 따라 총 4개의 레지스터를 안에 두었으며, 전부 0으로 초기화한 뒤 입력 명령에 따라 동작을 수행하도록 하였다. ‘read_register_one’과 ‘read_register_two’로 읽어올 레지스터 인덱스를 받으면 각각에 저장된 값을 ‘read_data_one’, ‘read_data_two’로 출력하였다. 또한 RegWrite 신호가 들어올 경우 ‘write_register’ 인덱스의 레지스터에 ‘write_data’ 값을 저장하였다. 이때 레지스터 값 변경은 입력되는 clock 신호에 positive edge triggered 되도록 always 문으로 설정해주었다. 또한 reset이 입력된 경우는 clock과 무관하게 모든 레지스터의 값을 0으로 초기화해주었다.

2.2 Control Unit

```

1 `timescale 1ns / 1ps
2
3 module control_unit(
4   input [1:0] op,
5   output RegDst,
6   output RegWrite,
7   output ALUSrc,
8   output Branch,
9   output MemRead,
10  output MemWrite,
11  output MemtoReg,
12  output ALUOp
13 );
14
15 reg [7:0] out;
16 assign {RegDst, RegWrite, ALUSrc, Branch, MemRead, MemWrite, MemtoReg, ALUOp} = out;

```

```

17
18     always @(op) begin
19         case (op)
20             0: out = 8'b11000001;
21             1: out = 8'b01101010;
22             2: out = 8'b00100100;
23             3: out = 8'b00010000;
24         endcase
25     end
26
27 endmodule
28

```

Listing 2: control_unit.v

Instruction의 최상위 두 비트에는 microprocessor가 수행할 작업의 종류가 나타나 있는데, control unit은 이를 받아 다른 모듈들에 어떤 동작을 수행할지 알려주는 신호를 출력한다. 과제 스펙에 맞게 4가지 입력에 따라 RegDst, RegWrite, ALUSrc, Branch, MemRead, MemWrite, MemtoReg, ALUOp의 값을 정해주었다.

2.3 ALU

```

1 'timescale 1ns / 1ps
2
3 module ALU(
4     input signed [7:0] A,
5     input signed [7:0] B,
6     input ALUOp,
7     output signed [7:0] O
8 );
9
10    assign O = ALUOp ? A + B : A - B;
11
12 endmodule
13

```

Listing 3: ALU.v

‘ALUOp’ 신호에 따라 두 가지 동작을 수행하는 ALU를 구현하였다. 다만 본 과제에서는 이 신호와 무관하게 ALU가 언제나 몇 셀만을 구현하여야 했기 때문에, ‘ALUOp’ 값과 무관하게 결과를 출력하였다. 또한 본 과제에서는 overflow가 발생하는 상황은 배제하였기 때문에 carry-out은 신경쓰지 않았다.

2.4 Data Memory

```

1 'timescale 1ns / 1ps
2
3 module data_memory(
4     input [7:0] address,
5     input [7:0] write_data,
6     input MemWrite, MemRead,
7     input CLK,
8     input reset,
9     output reg [7:0] read_data
10 );
11
12    reg [7:0] MEMORY [0:31];
13
14    integer i;
15    initial begin

```

```

16     for (i=0; i<32; i=i+1)
17         MEMORY[i] = i < 16 ? i : 16 - i;
18     end
19
20     always @(posedge CLK or posedge reset) begin
21         if (reset) begin
22             for (i=0; i<32; i=i+1)
23                 MEMORY[i] <= i < 16 ? i : 16 - i;
24         end else begin
25             if (MemWrite == 1) begin
26                 MEMORY[address] <= write_data;
27             end
28         end
29     end
30
31     always @(address or MemRead) begin
32         if (MemRead == 1)
33             read_data <= MEMORY[address];
34     end
35
36 endmodule
37

```

Listing 4: data_memory.v

7bit 데이터를 저장하는 메모리를 구현하였다. 총 32개의 데이터를 저장하였고, 순서대로 ‘0, 1, …, 15, 0, -1, …, -15’로 초기화해주었다. 또한 이 역시 특정 위치의 데이터를 읽어 출력하거나, 데이터를 수정하는 동작을 할 수 있도록 하였다. ‘MemRead’ 신호가 들어온 경우 ‘address’의 값을 ‘read_data’에 출력하였고, ‘MemWrite’ 신호가 들어온 경우 ‘address’ 위치의 데이터를 ‘write_data’로 바꿔주었다. 또한 값의 수정은 clock 신호에 positive edge triggered 되도록 always 문으로 설정해주었다. 나머지 reset이나 값 출력 동작은 clock과 무관하게 동작하도록 하였다.

2.5 Program Counter

```

1 'timescale 1ns / 1ps
2
3 module PC(
4     input CLK,
5     input reset,
6     input [7:0] D,
7     output reg [7:0] Q
8 );
9
10 initial Q = 0;
11
12 always @(posedge CLK or posedge reset) begin
13     if (reset)
14         Q <= 0;
15     else
16         Q <= D;
17 end
18
19 endmodule
20

```

Listing 5: PC.v

외부로부터 받아오는 instruction의 address를 바꾸어주는 역할을 한다. 기본적으로 ‘D’에 들어온 값으로 ‘Q’를 바꾸어주고, core logic에 있는 combinational circuit을 거쳐 ‘D’ 값이 결정되어 들어오면 다음 address가 결정된다.

2.6 MUX

```

1 'timescale 1ns / 1ps
2
3 module MUX_two(
4     input [1:0] IO,
5     input [1:0] I1,
6     input S0,
7     output [1:0] Z
8 );
9
10    assign Z = S0 ? I1 : IO;
11
12 endmodule
13

```

Listing 6: MUX_two.v

```

1 'timescale 1ns / 1ps
2
3 module MUX_eight(
4     input [7:0] IO,
5     input [7:0] I1,
6     input S0,
7     output [7:0] Z
8 );
9
10    assign Z = S0 ? I1 : IO;
11
12 endmodule
13

```

Listing 7: MUX_eight.v

입력된 control input ('S0')에 따라 두 값 중 하나를 선택해주는 2 to 1 MUX를 구현하였다. 실제 logic에서는 2bit, 8bit의 두 가지로 사용해야했기에 두 개의 모듈을 만들어주었다.

2.7 Frequency Divider

```

1 'timescale 1ns / 1ps
2
3 module frequency_divider(
4     input clkin,
5     input clr,
6     output reg clkout
7 );
8
9     reg [31:0] cnt;
10    initial begin
11        cnt = 0;
12        clkout = 0;
13    end
14
15    always @(posedge clkin or posedge clr) begin
16        if(clr) begin
17            cnt <= 32'd0;
18            clkout <= 1'b0;
19        end else if(cnt == 32'd25000000) begin
20            cnt <= 32'd0;
21            clkout <= ~clkout;
22        end else begin
23            cnt <= cnt + 1;
24        end
25    end
26
27 endmodule
28

```

Listing 8: frequency_divider.v

사용할 보드의 clock은 50MHz 신호를 주었는데, 과제 스펙에서는 1초에 한번 결과가 바뀌기를 요구했기에 진동수를 나누어주는 모듈을 제작하였다. Clock이 25000000번의 주기동안 움직일때마다 출력 clock을 toggle 시켜 결과적으로 50MHz 신호를 1Hz 신호로 변환할 수 있도록 하였다.

2.8 7 Segment Hexadecimal Decoder

```

1 'timescale 1ns / 1ps
2
3 module hex_to_7(
4     input [3:0] hex,
5     output reg [6:0] seg
6 );
7
8     always @(hex) begin
9         case (hex)
10             0: seg <= 7'b0111111;
11             1: seg <= 7'b0000110;
12             2: seg <= 7'b1011011;
13             3: seg <= 7'b1001111;
14             4: seg <= 7'b1100110;
15             5: seg <= 7'b1101101;
16             6: seg <= 7'b1111101;
17             7: seg <= 7'b0000111;
18             8: seg <= 7'b1111111;
19             9: seg <= 7'b1101111;
20             10:seg <= 7'b1110111;
21             11:seg <= 7'b1111100;
22             12:seg <= 7'b0111001;
23             13:seg <= 7'b1011110;
24             14:seg <= 7'b1111001;
25             15:seg <= 7'b1110001;
26         endcase
27     end
28
29 endmodule
30

```

Listing 9: hex_to_7.v

4bit binary input을 16진수로 변환하여 0 - F를 7 segment display의 7bit pattern으로 출력해주는 모듈을 만들었다.

3 Core Logic

```

1 'timescale 1ns / 1ps
2
3 module microprocessor(
4     input CLKIn,
5     input reset,
6     input [7:0] instruction,
7     output [7:0] read_address,
8     output [6:0] seg_h,
9     output [6:0] seg_l
10 );
11
12     wire [1:0] op, rs, rt, rd;
13     wire [7:0] sign_extended_rd;

```

```

14  wire [1:0] write_register;
15  wire [7:0] reg_write_data, read_data_one, read_data_two;
16  wire [7:0] ALU_second_input, ALU_output;
17  wire [7:0] read_data_memory;
18  wire [7:0] PC_in;
19  wire [7:0] PC_out;
20  wire [7:0] MUX_0, MUX_1;
21  wire CLK;
22  wire RegDst, RegWrite, ALUSrc, Branch, MemRead, MemWrite, MemtoReg, ALUOp;
23
24  assign op = instruction[7:6];
25  assign rs = instruction[5:4];
26  assign rt = instruction[3:2];
27  assign rd = instruction[1:0];
28  assign sign_extended_rd[1:0] = rd;
29  assign sign_extended_rd[7:2] = {6{rd[1]}};
30
31  assign MUX_0 = read_address + 1;
32  assign MUX_1 = MUX_0 + sign_extended_rd;
33
34  frequency_divider F1(.clkin(CLKin), .clr(reset), .clkout(CLK));
35  control_unit C1(.op(op), .RegDst(RegDst), .RegWrite(RegWrite), .ALUSrc(ALUSrc), .Branch(Branch),
36           .MemRead(MemRead), .MemWrite(MemWrite), .MementoReg(MemtoReg), .ALUOp(ALUOp));
37  MUX_two M1(.I0(rt), .I1(rd), .S0(RegDst), .Z(write_register));
38  register_module R1(.read_register_one(rs), .read_register_two(rt), .write_register(write_register),
39           .write_data(reg_write_data), .RegWrite(RegWrite), .CLK(CLK), .reset(reset),
40           .read_data_one(read_data_one), .read_data_two(read_data_two));
41  MUX_eight M2(.I0(read_data_two), .I1(sign_extended_rd), .S0(ALUSrc), .Z(ALU_second_input));
42  ALU A1(.A(read_data_one), .B(ALU_second_input), .ALUOp(ALUOp), .O(ALU_output));
43  data_memory D1(.address(ALU_output), .write_data(read_data_two), .MemWrite(MemWrite),
44           .MemRead(MemRead), .CLK(CLK), .reset(reset), .read_data(read_data_memory));
45  MUX_eight M3(.I0(ALU_output), .I1(read_data_memory), .S0(MemtoReg), .Z(reg_write_data));
46  MUX_eight M4(.I0(MUX_0), .I1(MUX_1), .S0(Branch), .Z(PC_in));
47  PC P1(.D(PC_in), .CLK(CLK), .reset(reset), .Q(read_address));
48
49  hex_to_7 H1(.hex(reg_write_data[7:4]), .seg(seg_h));
50  hex_to_7 H2(.hex(reg_write_data[3:0]), .seg(seg_l));
51
52 endmodule
53

```

Listing 10: microprocessor.v

2절에서 소개한 8개의 모듈을 종합하여 **Figure 1**의 모습대로 microprocessor를 구현하였다. **Figure 1**에 연결되어있는 모든 wire를 Verilog에서도 wire로 표현해주었고, 각 wire의 값을 알맞은 모듈의 input 혹은 output에 연결해주었다. 다만 **Figure 1**에 있는 두 개의 adder은 따로 구현하지 않고 assign 문을 이용하여 구현하였다.

전체적인 input은 외부의 instruction memory에서 받아오는 ‘instruction’, 그리고 clock 및 reset 신호였다. Output은 instruction memory에서 받아올 다음 instruction의 위치에 해당하는 ‘read_address’, 그리고 두 개의 7 segment pattern이었다. 이때 7 segment로 출력하는 값은 register에 입력되는 ‘write_data’를 16 진수 두 자리수로 변환한 값이었다. 또한 받아올 다음 instruction의 address는 기본적으로 전보다 1씩 증가한 값으로 하되, 전 instruction이 ‘11’로 시작하는 경우 약속된 해당 명령을 따라 이동하도록 하였다.

4 Simulation Results

새로운 보드를 이용하여 instruction memory를 구현하고, 14개의 명령을 저장해두고 0번부터 실행되도록 하여 테스트를 해보았다.

```
1 'timescale 1ns / 1ps
2
3 module IMEM(
4     input [7:0] read_address,
5     output [7:0] instruction
6 );
7
8     wire [7:0] MemByte[31:0];
9
10    assign MemByte[0] = {2'b01, 2'b00, 2'b10, 2'b01}; // $s2=Mem[$s0+1] / 1 = 01 / $s2 <= 1
11    assign MemByte[1] = {2'b11, 2'b00, 2'b00, 2'b01}; // j+1           / 0 = 00 / PC <= PC + 1
12    assign MemByte[2] = {2'b00, 2'b01, 2'b10, 2'b00}; // $s0=$s1+$s2      / X
13    assign MemByte[3] = {2'b10, 2'b10, 2'b10, 2'b01}; // Mem[$s2+1]=$s2 / 2 = 02 / Mem[2] <= 1
14    assign MemByte[4] = {2'b01, 2'b00, 2'b11, 2'b01}; // $s3=Mem[$s0+1] / 1 = 01 / $s3 <= 1
15    assign MemByte[5] = {2'b01, 2'b10, 2'b00, 2'b01}; // $s0=Mem[$s2+1] / 1 = 01 / $s0 <= 1
16    assign MemByte[6] = {2'b10, 2'b00, 2'b01, 2'b01}; // Mem[$s0+1]=$s1 / 2 = 02 / Mem[2] <= 0
17    assign MemByte[7] = {2'b00, 2'b00, 2'b00, 2'b00}; // $s0=$s0+$s0      / 2 = 02 / $s0 <= 2
18    assign MemByte[8] = {2'b01, 2'b00, 2'b11, 2'b01}; // $s3=Mem[$s0+1] / 3 = 03 / $s3 <= 3
19    assign MemByte[9] = {2'b00, 2'b00, 2'b11, 2'b11}; // $s3=$s0+$s3      / 5 = 05 / $s3 <= 5
20    assign MemByte[10] = {2'b00, 2'b11, 2'b11, 2'b00}; // $s0=$s3+$s3      / 10 = 0A / $s0 <= 10
21    assign MemByte[11] = {2'b00, 2'b00, 2'b00, 2'b11}; // $s3=$s0+$s0      / 20 = 14 / $s3 <= 20
22    assign MemByte[12] = {2'b01, 2'b11, 2'b10, 2'b11}; // $s2=Mem[$s3-1] / -3 = FD / $s2 <= 11111101
23    assign MemByte[13] = {2'b11, 2'b00, 2'b00, 2'b11}; // j-1              / 20 = 14 / PC <= PC - 1
24
25    assign instruction = MemByte[read_address];
26
27 endmodule
28
```

Listing 11: IMEM.v

위와 같이 14개의 명령어를 저장해두었고, 실행이 시작되면 0부터 13까지 순차적으로 실행되며 중간에 2번은 건너뛰고, 13번에 도달해서는 무한히 13번을 실행하게 된다. 결과적으로 7 segment display에 출력되는 값은 순서대로 01, 00, 02, 01, 01, 02, 02, 03, 05, 0A, 14, FD, 14 이어야 하며, 실제로 아래와 같이 출력되었다.

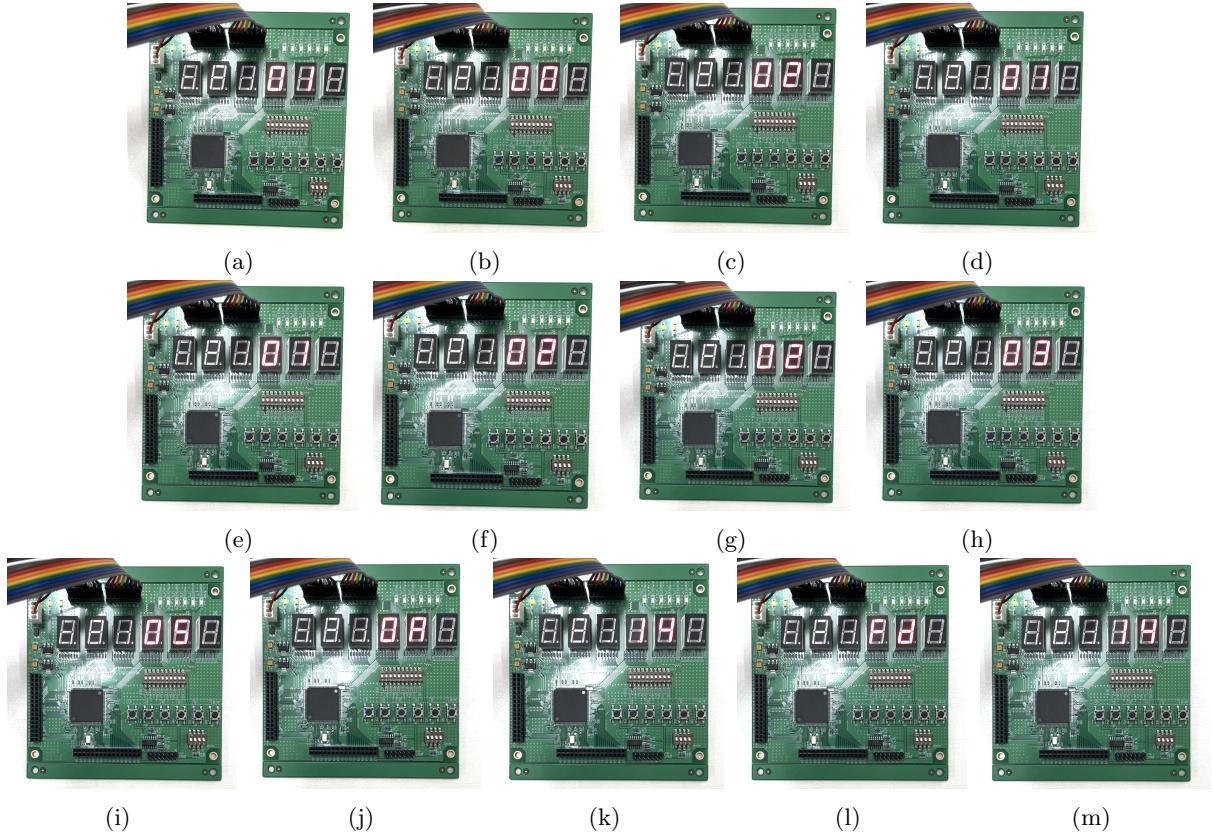


Figure 2: Microprocessor 테스트 결과. (a)부터 (m)까지 순서대로 진행한 모습이다.

또한 중간 각 wire의 값까지 포함하여 출력하도록 하여 Xilinx로 simulation을 해보았다. 이때 편의상 clock frequency는 2.5MHz로 하였고, 2000ns 시점에서 reset을 입력하였다. 전체적인 결과는 보드에서 실행한 것과 같은 것을 볼 수 있고, 초기화 이후에는 가장 처음과 같이 실행이 이루어지는 모습을 볼 수 있다.

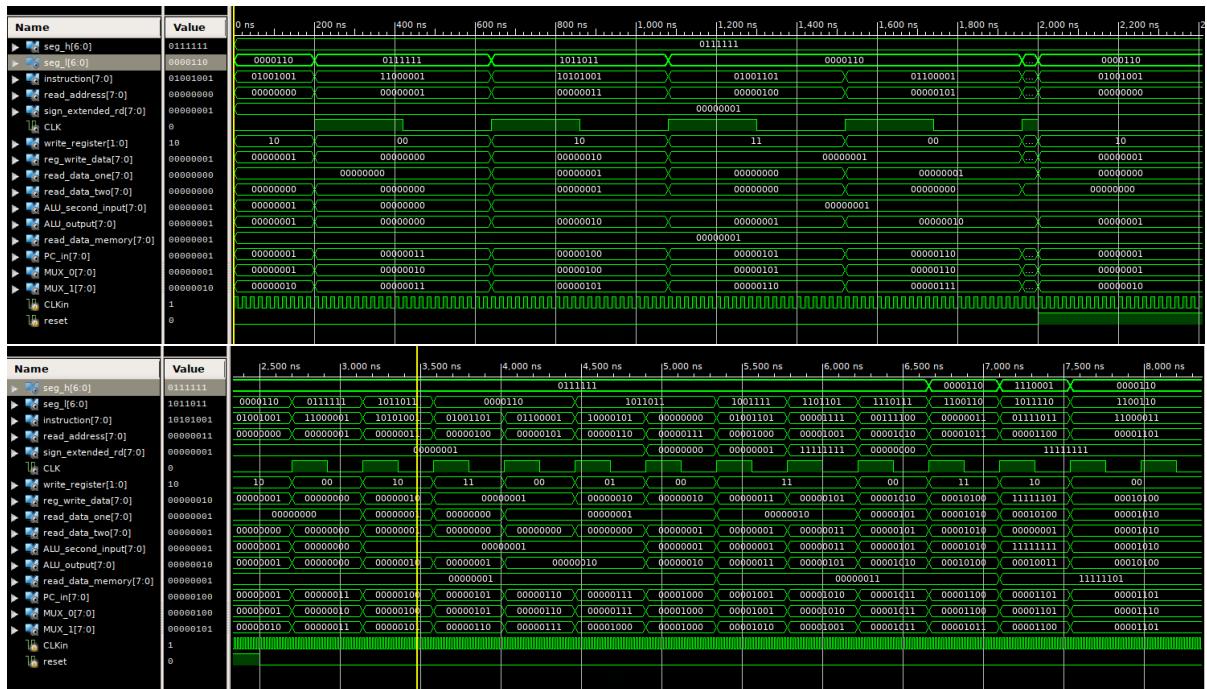


Figure 3: Microprocessor 시뮬레이션 결과