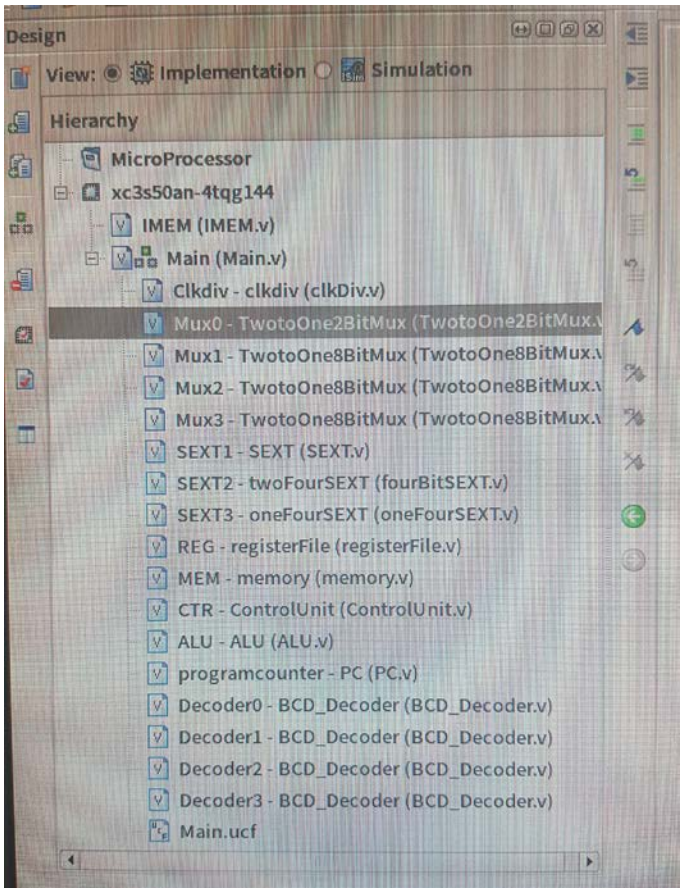
<논리설계실습 기말보고서>

2014-17831 김재원

#1 개요

본 실습을 통해 주어진 명령에 따라 8-bit 수를 더하고, 받아오고, 저장하는 기능을 하는 microprocessor을 디자인해보았습니다. 전체적인 구조는 다음과 같은 서브모듈들을 통해 구성되었고, 개별적인 기능은 각각에 대한 코드를 통해 설명하도록 하겠습니다.



#2-1 모듈 코드 설명 – Control Unit

```
--------------------------
module ControlUnit(
   input [1:0] op,
   output Branch,
   output MemToReg,
   output MemRead,
   output MemWrite,
   output ALUop,
   output ALUsrc,
   output RegWrite,
   output RegDst
```

```
    );

        assign Branch = (op == 2'b11) ? 1'b1 : 1'b0;
        assign MemToReg = (op == 2'b01) ? 1'b1 : 1'b0;
        assign MemRead = (op == 2'b01) ? 1'b1 : 1'b0;
        assign MemWrite = (op == 2'b10) ? 1'b1 : 1'b0;
        assign ALUop = (op == 2'b00) ? 1'b1: 1'b0;
        assign ALUsrc = (op == 2'b01) ? 1'b1 : (op == 2'b10) ? 1'b1: 1'b0;
        assign RegWrite = (op == 2'b00) ? 1'b1 : (op == 2'b01) ? 1'b1 :
1'b0;
        assign RegDst = (op == 2'b00) ? 1'b1 : 1'b0;

endmodule
```

먼저 control unit에는 2-bit의 input이 들어가고 그 결과로 8개의 1-bit output이 나옵니다. 이 때, input은 주어진 명령에서 일곱 번째와 여섯 번째 bit로, add/load/store/jump 중 어떤 명령을 수행할지 제시하는 역할을 합니다. Output은 각각 어떤 데이터를 가지고 어떤 기능을 수행할지 지시를 내려주는 역할을 합니다. 이를테면, input이 00일 경우 add 기능을 수행하게 되고, 이 때 필요한 해당 정보들, 즉 ALUop (어떤 연산을 수행할 것인가), RegWrite (연산 결과를 레지스터에 저장), RegDst (타겟 레지스터)을 control unit이 다른 모듈들에 전해줍니다.

Control Unit의 output에 해당하는 여덟 개 데이터는 삼항 연산자를 통해 구현하였습니다. 예를 들어, Branch output은 input이 11일 때, 즉 jump 기능을 수행할 때만 1이 됩니다. 따라서 op, 즉 input이 11일 때만 1-bit 1이 되고 나머지 경우에 1-bit 0이 되는 형태의 삼항 연산자가 적합합니다.

#2-2 모듈 코드 설명 - MUX

```
---------------------------
module TwotoOne8BitMux(
    input I,
    input [7:0] R0,
    input [7:0] R1,
    output [7:0] O
    );

        assign O = (I) ? R1 : R0;

endmodule


module TwotoOne2BitMux(
    input I,
    input [1:0] R0,
    input [1:0] R1,
    output [1:0] O
    );
```

```
	assign O = (I) ? R1 : R0;
endmodule
```

MUX의 경우 총 두 종류, 즉 2-bit 크기와 8-bit 크기 2-to-1 MUX가 있습니다. 두 개 중 하나를 선택하는 간단한 형태의 MUX이기 때문에 삼항연산자를 사용하여 combinational logic으로 간단히 구현하였습니다.

#2-3 모듈 코드 설명 - ALU

```
module ALU(
   input op,
   input [7:0] a,
   input [7:0] b,
   output [7:0] c,
   output OverFlow
   );

	reg [8:0] sum;

	assign OverFlow = sum[8];

	always @(*) begin
		sum = a+b;
	end

	assign c = sum[7:0];
endmodule
```

ALU 모듈의 경우 8-bit 크기의 adder 역할을 수행합니다. 이 microprocessor에서 사용하는 ALU의 기능은 더하는 기능 밖에 없고, 덧셈 연산은 combinational logic으로 간단히 구현했습니다. 이 때 오버플로우 탐지를 위해 연산 결과를 9-bit 레지스터에 우선 저장하였습니다. 이는 MSB가 1일 경우 overflow가 일어났음을 알 수 있기 때문이고, 따라서 해당 MSB를 OverFlow를 나타내는 output에 저장하였습니다. 연산의 결과값에는 나머지 8-bit만 전달합니다.

#2-4 모듈 코드 설명 - DM

```
---------------------------
module memory(
   input [7:0] Address,
   input Read,
   input Write,
   input [7:0] WriteData,
   input Reset,
   input clock,
```

```
   output [7:0] ReadData
   );

        reg[7:0] memory[31:0]; //in order to reset, we need 32 Memory
Registers
        //Reset & Write

        always @(posedge clock or posedge Reset) begin

              if(Reset)
              begin
                    memory[0] = 8'b00000000;
                    memory[1] = 8'b00000001;
                    memory[2] = 8'b00000010;
                    memory[3] = 8'b00000011;
                    memory[4] = 8'b00000100;
                    memory[5] = 8'b00000101;
                    memory[6] = 8'b00000110;
                    memory[7] = 8'b00000111;
                    memory[8] = 8'b00001000;
                    memory[9] = 8'b00001001;
                    memory[10] = 8'b00001010;
                    memory[11] = 8'b00001011;
                    memory[12] = 8'b00001100;
                    memory[13] = 8'b00001101;
                    memory[14] = 8'b00001110;
                    memory[15] = 8'b00001111;

                    memory[16] = 8'b00000000;
                    memory[17] = 8'b11111111;
                    memory[18] = 8'b11111110;
                    memory[19] = 8'b11111101;
                    memory[20] = 8'b11111100;
                    memory[21] = 8'b11111011;
                    memory[22] = 8'b11111010;
                    memory[23] = 8'b11111001;
                    memory[24] = 8'b11111000;
                    memory[25] = 8'b11110111;
                    memory[26] = 8'b11110110;
                    memory[27] = 8'b11110101;
                    memory[28] = 8'b11110100;
                    memory[29] = 8'b11110011;
                    memory[30] = 8'b11110010;
                    memory[31] = 8'b11110001;
              end

              else if(Write)
              begin
                    memory[Address] = WriteData;
              end
        end

        //Read
       assign ReadData = (Read == 1'b1) ? memory[Address] : 0;

endmodule


-------------------------
```

Memory 모듈은 32개 주소의 8-bit memory에 값을 저장하고 불러오는 역할을 합니다. 우선 clock의 rising edge에서 Reset input이 들어오면 각각의 값은 -15에서부터 15까지의 값으로 초기화가 됩니다. 또한, Control Unit으로부터 MemWrite 신호가 보내져 Write input으로 들어오게 되면 메모리 주소를 input으로 받아들여서 해당되는 메모리 주소의 값을 내보냅니다. 메모리에서 데이터를 읽는 기능 (ReadData)는 combinational logic으로 구현해서 clock과 상관 없이 값을 항상 대기하도록 하였습니다.

#2-5 모듈 코드 설명 – Register

```
-------------------------
module registerFile(
   input clock,
   input Reset,
   input [1:0] ReadReg1,
   input [1:0] ReadReg2,
   input Write,
   input [1:0] WriteReg,
   input [7:0] WriteData,
   output [7:0] ReadData1,
   output [7:0] ReadData2
   );

       reg[7:0] register [3:0];

       //Reset & Write Data
       always @(posedge clock or posedge Reset) begin

              if(Reset) //if Reset is asserted, initialize all Registers.
              begin
                     register[0] = 0;
                     register[1] = 0;
                     register[2] = 0;
                     register[3] = 0;
              end
              else if(Write)
              begin
                     register[WriteReg] = WriteData;
              end
       end

       //Read Data (this logic is combinational, not sequential)
       assign ReadData1 = register[ReadReg1];
       assign ReadData2 = register[ReadReg2];


endmodule


-------------------------
```

registerFile 모듈은 우선 Reset 신호가 들어왔을 때 네 개 레지스터 각각을 0으로 초기화해줍니다. 이 모듈은 주어진 명령 (add, load, store, jump)에 따라 레지스터의 값을 읽고 쓰는 역할을 합니다.

레지스터를 읽는 기능은 clock에 상관 없이 combinational logic으로 항상 실행이 되고, 적는 기능은 Reset느이 아닐 때 실행이 됩니다. 이 때, Memory로부터 들어온 WriteData 값을 WriteReg의 주소에 적어주게 됩니다.

#2-6 모듈 코드 설명 - Sign Extend

```
-------------------------
module SEXT(
   input [1:0] In,
   output reg [7:0] Out
   );

       always @(In) begin
             case(In)
                    2'b00: Out = 8'b00000000;
                    2'b01: Out = 8'b00000001;
                    2'b10: Out = 8'b11111110;
                    2'b11: Out = 8'b11111111;
             endcase
       end

endmodule

module twoFourSEXT(
   input [1:0] In,
   output reg [3:0] Out
   );

       always @(In) begin
             case(In)
                    2'b00: Out = 4'b0000;
                    2'b01: Out = 4'b0001;
                    2'b10: Out = 4'b1110;
                    2'b11: Out = 4'b1111;
             endcase
       end

endmodule

module oneFourSEXT(
   input In,
   output reg [3:0] Out
   );

       always @(In) begin
             case(In)
                    1'b0: Out = 4'b0000;
                    1'b1: Out = 4'b0001;
             endcase
       end

endmodule


-------------------------
```

본 마이크로프로세서에는 총 세 종류의 sign extend 모듈이 있습니다. 우선 instruction의 주소를 정해주기 위해 2-bit를 8-bit로 늘려주는 역할을 하는 SEXT 모듈이 있습니다. 이 모듈은 instruction에서 마지막 두 bit를 받아 8-bit bit string으로 늘려줍니다. 나머지 두 개 sign extend 모듈은 추가 기능 구현을 위한 것으로, 각각 ALU에서 오버플로우가 발생했을 때 전달되는 1-bit bit string과 add/load/store/jump 중 어떤 기능을 수행할 것인지 전달하는 2-bit bit string를 4-bit로 늘려주는 역할을 합니다. 이렇게 변환된 bit string은 4:7 decoder로 전달이 되기 때문에 SEXT 모듈과 달리 output이 4-bit입니다.

#2-7 모듈 코드 설명 – Clock Divider

```
-------------------------
module clkdiv(
   input clkin,
   input clr,
   output reg clkout
   );

        reg[31:0] cnt;
        always @(posedge clkin or posedge clr) begin
              if(clr) begin
                    cnt <= 32'd0;
                    clkout <= 1'b0;
              end
              else if(cnt == 32'd25000000) begin
                    cnt<= 32'b0;
                    clkout <= ~clkout;
              end
              else begin
                    cnt <= cnt+1;
              end
       end

endmodule


-------------------------
```

본 모듈는 아홉 번째 실습 시간에 주어진 코드를 사용하여 구현하였습니다. FPGA 보드 상의 oscillator가 보내는 신호를 사람이 볼 수 있도록 변환하는 역할을 합니다. 이 때 50Mhz의 clock을 1초에 한 번씩으로 조정해줍니다.

#2-8 모듈 코드 설명 – Program Counter

```
-------------------------
module PC(
   input [7:0] in,
   input clock,
   input Reset,
```

```
    output [7:0] out
    );

        reg[7:0] inn;
        assign out = inn;

        always @(posedge clock or posedge Reset) begin

               if(Reset)
               begin
                      inn = 8'd0;
               end

               else
               begin
                      inn = in;
               end

        end

endmodule


-------------------------
```

PC 모듈은 다음 clock에 보내야할 instruction의 주소를 전달하는 역할을 합니다. Reset이 들어올 경우 이 주소는 8-bit 0으로 초기화가 됩니다.

#2-9 모듈 코드 설명 – Seven_to_Hex Decoder

```
-------------------------
module BCD_Decoder(
    input [3:0] BCD,
    output reg [6:0] Light
    );

        always @(BCD) begin
               case(BCD)
                      4'd0: Light <= 7'b0111111;
                      4'd1: Light <= 7'b0000110;
                      4'd2: Light <= 7'b1011011;
                      4'd3: Light <= 7'b1001111;
                      4'd4: Light <= 7'b1100110;
                      4'd5: Light <= 7'b1101101;
                      4'd6: Light <= 7'b1111101;
                      4'd7: Light <= 7'b0000111;
                      4'd8: Light <= 7'b1111111;
                      4'd9: Light <= 7'b1101111;
                      4'd10: Light <= 7'b1110111;
                      4'd11: Light <= 7'b1111100;
                      4'd12: Light <= 7'b0111001;
                      4'd13: Light <= 7'b1011110;
                      4'd14: Light <= 7'b1111001;
                      4'd15: Light <= 7'b1110001;
               endcase
        end
```

```
endmodule
```

```
-------------------------
```

본 모듈은 4-bit 숫자를 input으로 받아 숫자를 display하기 위한 7-bit의 신호로 변환해주는 역할을 합니다. 0에서부터 16에 해당하는 F까지의 숫자를 seven-segment display로 나타내기 위한 bit string들을 저장해주는 역할을 합니다.

#3-1 전체 구조 설명 – Main

```
-------------------------
module Main(
   input [7:0] instruction,
   output [7:0] pc,
   input Reset,
   input clock,
   output [6:0] Reg1,
   output [6:0] Reg2,
   output [6:0] OverFlowSeg,
   output [6:0] OpSeg,
   output clockLED
   );

        // STEP 1 : Declare All The Wires

        //Instruction Set
        wire [1:0] inst1_0, inst3_2, inst5_4, inst7_6;
        wire [7:0] instSEXT1_0;
        wire [3:0] instSEXT7_6;
        wire [3:0] overflowSEXT;
        //Clock
        wire CLK;
        //Multiplexer
        wire [1:0] WriteReg_Mux;
        wire [7:0] WriteData_Mux;
        wire [7:0] PCin_Mux;
        wire [7:0] ALUb_Mux;
        //Register
        wire [7:0] ReadData1, ReadData2;
        //Memory
        wire [7:0] ReadData;
        //추가된내용
        //wire [7:0] Address;
        //ControlUnit
        wire Branch, MemToReg, MemRead, MemWrite, ALUop, ALUsrc, RegWrite,
RegDst;
        //Arithmetic Logic Unit
        wire [7:0] c;
        wire OverFlow;
        //PC
        //wire [7:0] out;
        //assign PC = out;

        //STEP 2: Assign
```

```
        //Instruction Set
        assign inst1_0 = instruction[1:0];
        assign inst3_2 = instruction[3:2];
        assign inst5_4 = instruction[5:4];
        assign inst7_6 = instruction[7:6];

        //clkdiv
        clkdiv Clkdiv(.clkin(clock), .clr(Reset), .clkout(CLK));
        assign clockLED = CLK;

        //Multiplexer
        TwotoOne2BitMux
Mux0(.I(RegDst), .R0(inst3_2), .R1(inst1_0), .O(WriteReg_Mux));
        TwotoOne8BitMux
Mux1(.I(MemToReg), .R0(c), .R1(ReadData), .O(WriteData_Mux));
        TwotoOne8BitMux
Mux2(.I(ALUsrc), .R0(ReadData2), .R1(instSEXT1_0), .O(ALUb_Mux));
        TwotoOne8BitMux
Mux3(.I(Branch), .R0(pc+8'b00000001), .R1(pc+8'b00000001+instSEXT1_0), .O(
PCin_Mux));

        //SEXT
        SEXT SEXT1(.In(inst1_0), .Out(instSEXT1_0));
        twoFourSEXT SEXT2 (.In(inst7_6), .Out(instSEXT7_6));
        oneFourSEXT SEXT3 (.In(OverFlow), .Out(overflowSEXT));

        //Register
        registerFile
REG(.ReadReg1(inst5_4), .ReadReg2(inst3_2), .Write(RegWrite),

       .WriteReg(WriteReg_Mux), .WriteData(WriteData_Mux), .ReadData1(ReadD
ata1),

       .ReadData2(ReadData2), .clock(CLK), .Reset(Reset));
        //Memory
        memory
MEM(.Write(MemWrite), .Read(MemRead), .WriteData(ReadData2), .Address(c),
.ReadData(ReadData),
                                  .clock(CLK), .Reset(Reset));

        //ControlUnit
        ControlUnit
CTR(.op(inst7_6), .Branch(Branch), .MemToReg(MemToReg), .MemRead(MemRead),

         .MemWrite(MemWrite), .ALUop(ALUop), .ALUsrc(ALUsrc), .RegWrite(Reg
Write),
                                           .RegDst(RegDst));

        //Arithmetic Logic Unit
        ALU
ALU(.op(ALUop), .a(ReadData1), .b(ALUb_Mux), .c(c), .OverFlow(OverFlow));

        //PC
        PC
programcounter(.in(PCin_Mux), .clock(CLK), .Reset(Reset), .out(pc));

        //7-Segment Display Decoder
        BCD_Decoder Decoder0(.BCD(WriteData_Mux[7:4]), .Light(Reg1));
```

```
        BCD_Decoder Decoder1(.BCD(WriteData_Mux[3:0]), .Light(Reg2));
        BCD_Decoder Decoder2(.BCD(overflowSEXT), .Light(OverFlowSeg));
        BCD_Decoder Decoder3(.BCD(instSEXT7_6), .Light(OpSeg));

endmodule
```

Main 모듈에서는 서브 모듈을 연결하여 microprocessor가 작동하도록 합니다. 코드가 쓰여진 순서대로 세부 구성을 설명하도록 하겠습니다. 우선 clock과 Reset, 그리고 instruction을 input으로 받아들입니다. 그 다음, 8-bit instruction을 2-bit씩 총 네 개로 끊어서 저장합니다. Instruction이 add/load/store/jump의 operation, 레지스터1의 값, 레지스터 2의 값, 그리고 값을 적는 위치의 총 네 개 정보를 담은 2-bit bit string으로 구성되어 있기 때문입니다. 그 다음, clock에 따라 LED가 깜빡이도록 하기 위해 clkDiv로 변환한 것을 clockLED로 보내줍니다.

나머지 코드는 주어진 스펙의 구조 다이어그램을 통해 그 input과 output을 쉽게 파악할 수 있습니다. 주어진 스펙에 맞게 총 네 개의 MUX와 Control Unit, Register file, memory, ALU, decoder, pc, 그리고 sign extend가 사용되고 있습니다. 다만 추가적인 기능, 즉 overflow detection과 instruction display로 인해 약간의 변형이 있습니다. 우선 Overflow를 ALU에서 감지하면 이 값이 decoder로 전달되어 Overflow가 일어날 때마다 1을 보여줍니다. 또한, instruction에서 operation에 해당되는 부분을 또 하나의 display로 보여주도록 추가 기능을 구현하였습니다. 예를 들어, add 기능을 수행할 경우 0이, load를 수행할 경우 1이 보여지는 형태입니다. 이들 기능을 위해 sign extend (oneFourSEXT, twoFourSEXT)와 decoder가 각각 두 개씩 추가적으로 사용되었습니다.

#3-2 전체 구조 설명 - Test

```
--------------------------
module finalTest_1;

	// Inputs
	reg [7:0] instruction;
	reg Reset;
	reg clock;

	// Outputs
	wire [7:0] pc;
	wire [6:0] Reg1;
	wire [6:0] Reg2;
	wire [6:0] OverFlowSeg;
	wire [6:0] OpSeg;
	wire clockLED;

	// Instantiate the Unit Under Test (UUT)
	Main uut (
		.instruction(instruction),
		.pc(pc),
```

```
		.Reset(Reset),
		.clock(clock),
		.Reg1(Reg1),
		.Reg2(Reg2),
		.OverFlowSeg(OverFlowSeg),
		.OpSeg(OpSeg),
		.clockLED(clockLED)
	);
always #10 clock=~clock;
	initial begin
		// Initialize Inputs
		instruction = 0;
		Reset = 1;
		clock = 0;

		#100000
		Reset = 0;

		// Wait 100 ns for global reset to finish
		#50000;
		instruction = 8'b01000100; #100000;
		instruction  = 8'b01001001; #100000;
		instruction  = 8'b00011001; #100000
		instruction  = 8'b10000100; #100000;

		instruction = 8'b01000100; #100000;
		instruction = 8'b01001001; #100000;
		instruction = 8'b00011001; #160;
		instruction  = 8'b10000100; #160;

		instruction = 8'b01000100; #160;
		instruction = 8'b01001001; #160;
		instruction  = 8'b00011001; #160;
		instruction = 8'b10000100; #160;

		instruction  = 8'b01000100; #160;
		instruction = 8'b01001001; #160;
		instruction  = 8'b00011001; #160;
		instruction = 8'b10000100; #160;

		instruction  = 8'b01000100; #160;
		instruction = 8'b01001001; #160;
		instruction = 8'b00011001; #160;
		instruction = 8'b10000100; #160;

		instruction  = 8'b11000011; #160;
		// Add stimulus here

	end

endmodule
```

구현한 마이크로프로세서가 제대로 작동하는지 확인하기 위해 작성한 테스트 모듈입니다. 주어진 IMEM의 주소를 가져와서 사용하였습니다.

#4 시뮬레이션 결과

위 테스트 모듈을 사용하여 시뮬레이션을 진행한 결과 다음과 같은 결과를 얻을 수 있었습니다. 16개 instruction이 실행되는 동안 00→01→01→XX→01→01→02…의 값이 보여져야하는데, 대체적으로 잘 작동하는 것을 확인할 수 있었습니다. 이를 테면, 첫 번째 instruction인 8'b01000100이 들어왔을 때, 00번 메모리의 값인 0을 01번 레지스터로 load해야합니다. 따라서 display의 두 개값을 보여주는 Reg1과 Reg2 모두에 0이 보여져야 하는데, 8'11111110으로 0에 해당하는 값이 잘 보여지고 있음을 알 수 있습니다. 다만, add instruction이 실행될 때마다 실제 계산한 값보다 1이 증가된 값이 보여지고 있었습니다. WriteData나 ReadDate, 그리고 레지스터의 값이 모두 올바르게 전달되고 있음을 확인하였기 때문에 이것이 timing 문제로 인해 발생한 것임을 유추할 수 있었습니다. ReadData가 combinational logic으로 구현되어 시뮬레이션 실행 시 계산되어 저장된 값을 너무 빠르게 불러온 탓인데, FPGA에 구현할 시에는 해당 문제가 발생하지 않습니다.