컴퓨터 구조 Lab3

20190782 컴퓨터공학과 최서영
20190439 컴퓨터공학과 오승훈

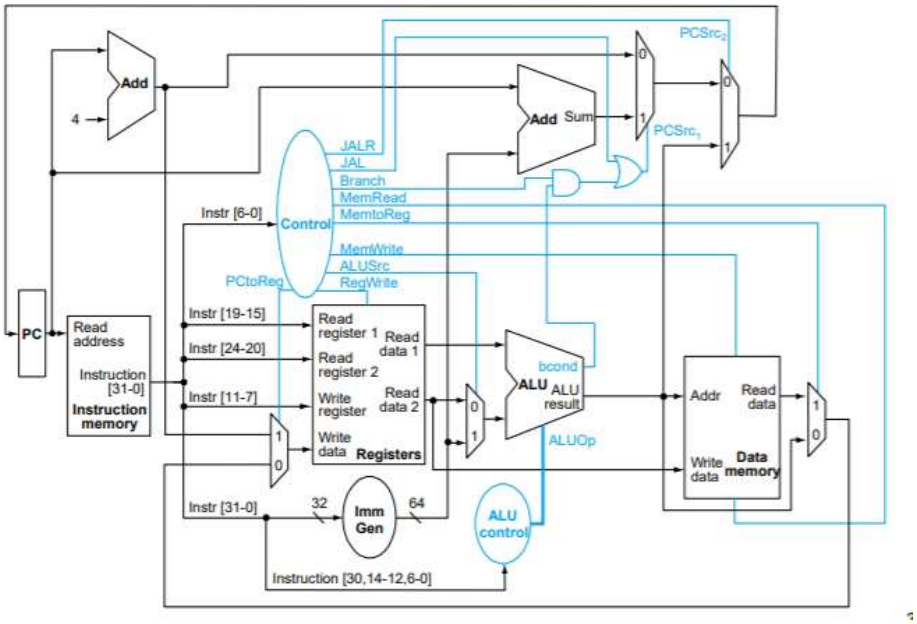
Introduction

이번 lab에서는 single cycle cpu를 구현해보는 것이 목표이다. Cpu를 크게 datapath와 control unit으로 나누어서 구현해야 하며, datapath 내부에는 register와 ALU module이 존재해야 한다. Single cycle Cpu가 어떤 식으로 동작하는지 잘 이해하여 modelsim에서 Verilog로 구현하는 것이 최종 목표이다.

Design

이번 lab에서는 조교님께서 처음 제공해주신 ALU, CPU, Control_unit, register_file 외에도 구현에 필요하다고 판단되어 Adder, datapath_real, mux_2_1, sign_extend 등의 모듈을 추가적으로 만들어 구현하였다. 먼저 control_unit에서 나온 다음 address를 받아 address를 다음 address로 변경하고, clock의 negative edge신호를 받아 ReadM을 1로 바꾸어 준다. ReadM이 1로 바뀌면 test bench에서 instruction을 data로 보내주면서, inputReady 값을 1로 바꿔준다. Instruction값이 바뀌고, inputReady의 positive edge 신호를 받으면 control_unit과 datapath에서 해당 instruction에 연산을 진행한다. Control_unit에서는 해당 instruction에 대한 mem_read, mem_write 등과 같은 datapath에서 작업을 수행할 수 있게 하는 signal 값들을 설정해준다. 그후, 해당 signal들을 datapath로 보내준다. datapath에서는 각각의 instruction 값에 맞게 opcode에 따라 register_file에 들어갈 값들을 정해준다. 그후, control_unit에서 보내준 signal들에 맞게 세부 모듈들을 작동시킨다. 이때, alu module이 주로 사용된다. ALU module은 inputReady의 negative edge에서 동작이 수행되는데 이는 타이밍을 Instruction decoding과 다른 시간에 진행하기 위해서이다. ALU module에서는 insturction의 opcode와 세부 function code를 이용하여 각 명령어에 맞는 연산을 진행하게 된다. 이때 만약 branch 연산이라면 branch_bit에 해당 조건이 맞는지를 계산한다. 그후, branch_bit와 연산 결과를 반환해준다. 만약, memory에 대한 접근(store, load)가 필요한 연산이라면 clock의 positive edge 신호를 받아 WriteM, 혹은 ReadM을 1로 바꾸어 주어 memory data에 접근한다. 이때, 접근에 필요한 memory address, 혹은 data는 이전 clock이 negative일 때 모든 계산을 해 놓았다. 이외의 작업이 끝나면 PC_Src1과 PC_Src2에 따라서 다음 address가 연산이 된다.

전체적인 구상도는 다음과 같다.



Implementation

이번 랩에서는 single cycle cpu를 구현해보는 것이 목표였다. 이를 위해 총 2가지 모듈로 나누고, 그 후 내부에는 ALU, mux, adder와 같은 기본적인 모듈을 이용하여 구현을 하였다. 먼저 datapath는 실질적인 data들이 in, out이 되는 모듈이다. 이 모듈에서는 instruction이 들어오고, 어떤 register에서 값을 가져오는지, 어떤 값이 immediate value인지 등과 같은 것들을 정해주게 되고, memory read, write를 하게 되면 어떤 data를 어떤 memory address에 사용하는지, register write를 하게 된다면 어느 register에 write를 하는지 등과 같은 동작을 하는 모듈이다. 그리고 control unit이 존재하는데 control unit의 경우 instruction에 따라 어떤 값을 사용해야하는지를 ON, OFF 스위치처럼 mux와 alu 등의 작동을 control 해주는 동작과, pc 값이 어떤 값으로 update 되야하는지 등을 담당하는 module이 된다.

a. CPU_top module

```
always@(posedge inputReady) begin
        if(!clk) begin
                reg_instruction <= data;
        end
        else begin
                if(inputReady) begin
                        reg_instruction <= data;
                end
        end
end
```

Cpu top module에서는 negedge clock을 이용하여 instruction을 read를 해오는 것을 목표로 진행하였기 때문에 inputReady로 값이 들어오고, instruction이 들어온 것이 확인이 되

면 reg_instruction register에 값을 받아온 뒤 각 모듈로 값을 보내줬다. 그리고 positive clock에서는 inputReady 값이 1이 되는 경우는 memory read를 했을 경우밖에 없으므로 instruction을 넣어줬다.

```
control_unit control(reg_instruction, alu_src, reg_write, mem_read, mem_to_reg, mem_write, jalr, jal, branch, PctoReg, PCsrc1, PCsrc2,
pc_value, clk, immediate, alu_output, b_cond, reset_n, ackOutput, inputReady, ReadM, WriteM);

datapath_top_module datapath( alu_src, reg_write, mem_read, mem_to_reg, mem_write, clk, reg_instruction,
datapath_address_out, datapath_data_out, inputReady, ackOutput, reset_n, branch, pc_value, jalr, jal, PctoReg, PCsrc1, PCsrc2, immediate, alu_output, b_cond);
```

그 후 instruction을 datapath와 control unit으로 wire 연결을 해서 넣어주면 된다.

b. Datapath

```
initial begin
        register_read_1_index = 0;
        register_read_2_index = 0;
        register_write_reg_index = 0;
end

always @(instruction) begin
        if(!clk) begin
                if(instruction[15:12] == 15) begin
                        if(instruction[5:0] == 26) begin
                                register_write_reg_index <= 2;
                        end
                        else begin
                                register_read_1_index <= instruction[11:10];
                                register_read_2_index <= instruction[9:8];
                                register_write_reg_index <= instruction[7:6];
                        end
                end
                else if(instruction[15:12] == 10 ) begin
                        register_write_reg_index <= 2;
                end
                else begin
                        register_read_1_index <= instruction[11:10];
                        register_read_2_index <= instruction[9:8];
                        register_write_reg_index <= instruction[9:8];
                end
        end

end
```

처음에는 register index 값들을 모두 초기화를 시켜준다. 그 후로는 datapath에서는 register라는 모듈 내부로 값을 넣어주기 위해서 instruction이 바뀌는 경우에만 그리고 또한 negedge clock에 instruction을 받아오도록 구현을 해뒀기 때문에 clk 값이 0인 경우에만 instruction이 들어온 경우이므로 instruction이 어떤 것이냐에 따라서 어느 레지스터를 사용하는지가 다르기 때문에 위와 같이 register 값이 type 또는 instruction에 맞게 들어가도록 선택을 해준다.

```
mux_2to_1_anytime write_data_selection(inputReady, PctoReg, memory_read_out, pc_value, write_data_selection_output_data);

register_file register( register_out_1, register_out_2, register_read_1_index, register_read_2_index, register_write_reg_index ,
write_data_selection_output_data, reg_write, clk, inputReady, ackOutput, PctoReg);

sign_extend sign_extend(instruction, clk, sign_extend_out_wire);

mux_2to_1_anytime ALU_src_selection(inputReady, alu_src, register_out_2, sign_extend_out_wire, alu_input_2);

alu alu_examine(register_out_1, alu_input_2, instruction, alu_output, clk, inputReady, bcond);

mux_2to_1_anytime momory_read_data_selection(mem_to_reg, mem_to_reg, alu_output, instruction, memory_read_out);
```

그 후에는 위와 같이 single cycle의 design 그림과 동일하도록 mux register file, alu, sign_extend 등의 모듈로 각각의 값을 wire 연결을 해준다.

c. Control unit

```
initial begin
   reg_write <= 1'b0;
   alu_src <= 1'b0;
   mem_read <= 1'b0;
   mem_write <= 1'b0;
   mem_to_reg <= 1'b0;
   jalr <= 1'b0;
   jal <= 1'b0;
   branch <= 1'b0;
   PctoReg <= 1'b0;
   PCsrc1 <= 1'b0;
   PCsrc2 <= 1'b0;
   pc_value <= 1'b0;
   ReadM <= 1'b0;
   WriteM <= 1'b0;
end
always @(posedge inputReady) begin
   ReadM <= 0;
end
always @(posedge ackOutput) begin
   WriteM <= 0;
end
always @(negedge clk) begin
   ReadM <= 1;
end
always @(posedge clk) begin
   if(mem_read) begin
     ReadM <= 1;
   end
   else if(mem_write) begin
     WriteM <= 1;
   end
end
```

처음에는 모든 control unit의 value들을 초기화를 시켜주고, negedge clk 마다 instruction을 읽어와야 하므로 ReadM 즉 memory에서 instruction을 읽어오는 값을 1로 만들어준다. 그 후 inputReady가 1이 되면 memory를 다 읽어온 것이므로 readM을 다시 1로 만들어준다. 그리고 posedge clk 마다 memory를 read 또는 write를 동작 수행을 할 것이기 때문에 mem_read가 1로 설정되어 있으면 memory read를 해야하는 동작을 수행해야 하므로 ReadM을 1로 설정을 해주고, mem_write이 1로 설정이 되면 WriteM을 1로 설정을 해줘서 memory read 또는 write이 가능하도록 한다. 그 후 ackOutput이 1로 설정이 되면 Write가 끝난 것이므로, WriteM이 0으로 설정을 한다.

```
always @(negedge clk) begin
   if(reset_n) begin
     pc_value <= pc_value_out;
   end
   else begin
     pc_value <= pc_value;
   end
end
```

그리고 이것은 negedge clk 마다 pc 값을 update를 해주면 되는 것이고, pc_value_out이라는 값은 single cycle design이 있다면 pc update를 하는 값들 중 맨 마지막 pc_value의 mux에서 나온 값이다.

```
always @(instr) begin
  if(!clk) begin
   reg_write = 1'b0; //RegWrite
   alu_src = 1'b0;   //ALUsrc
   mem_read = 1'b0;  //MemRead
   mem_write = 1'b0; //MemWrite
   mem_to_reg = 1'b0;//MemtoReg
   jalr = 1'b0;
   jal = 1'b0;
   branch = 1'b0;
   PctoReg = 1'b0;
   PCsrc1 = 1'b0;
   PCsrc2 = 1'b0;
   opcode = instr[15:12];

   case(opcode)
   `ALU_OP:begin
      reg_write = 1'b1;
   end
   `ADI_OP:begin
      reg_write = 1'b1;
      alu_src = 1'b1;
```

(밑에 case가 더 있는데 너무 길어서 제외합니다!)

맨마지막으로 instruction이 들어오고, clk가 0일때마다 다른 control value들 ex) reg_write, alu_src, mem_read etc... 들을 update를 해주는데 이것은 opcode의 case에 따라서 값을 모두 update 하도록 구현을 해뒀다.

d. Other module

d-1) register file

```
initial begin
        register[0] = 0;
        register[1] = 0;
        register[2] = 0;
        register[3] = 0;
end

assign read_out1 = register[read1];
assign read_out2 = register[read2];

always@(posedge clk) begin
        if(reg_write == 1) begin
        register[write_reg] = write_data;
        end
        else begin
        register[write_reg] = register[write_reg];
        end
end
always@(negedge inputReady) begin
        if(reg_write == 1 && clk) begin
        register[write_reg] = write_data;
        end
        else if(PctoReg) begin
        register[write_reg] = write_data;
        end
        else begin
        register[write_reg] = register[write_reg];
        end
end
```

이 모듈은 register 값들이 저장이 되어있고, 각 instruction이 들어오면 각 instruction에 맞는 value 값들을 datapath의 alu 연산으로 보내주는 역할을 한다.

그리고 또한 clk이 posedge일 때 reg_write bit이 1로 켜져있으면 register write_back을 진행을 해주고, inputReady 값이 설정이 되어서 negedge inputReady로 가게 될

때 clk가 1이라면 memory read가 완료가 된 상태이므로 이때의 data를 register에 write를 해준다. 또한 PctoReg 값이 설정이 되면 pc_value를 저장을 해야하기 때문에 이 때 역시 register write_back을 진행을 시켜주는 역할을 한다.

d-2) alu

```
always @(negedge inputReady) begin
        branch_bit = 0;
        if(instruction[15:12] == 4'b1111) begin
                case(instruction[5:0])
                0: alu_output <= alu_input_1 + alu_input_2;
                1: alu_output <= alu_input_1 - alu_input_2;
                2: alu_output <= alu_input_1 & alu_input_2;
                3: alu_output <= alu_input_1 | alu_input_2;
                4: alu_output <= ~(alu_input_1);
                5: alu_output <= ~(alu_input_1) + 1;
                6: alu_output <= alu_input_1 << 1;
                7: alu_output <= signed_input_1 >>> 1;
                25: alu_output <= alu_input_1;
                26: alu_output <= alu_input_1;
                endcase
        end
        else if(instruction[15:12] == 4'd4 || instruction[15:12] == 4'd5 || instruction[15:12] == 4'd6 ||
                instruction[15:12] == 4'd7 || instruction[15:12] == 4'd8 ) begin
                case(instruction[15:12])
                4: alu_output <= alu_input_1 + signed_input_2;
                5: alu_output <= alu_input_1 | signed_input_2;
                6: alu_output <= alu_input_2 << 8;
                7: alu_output <= alu_input_1 + signed_input_2;
                8: alu_output <= alu_input_1 + signed_input_2;
                endcase
        end
        else if(instruction[15:12] == 4'd0 || instruction[15:12] == 4'd1 || instruction[15:12] == 4'd2 || instruction[15:12] == 4'd3) begin
                case(instruction[15:12])
                0: branch_bit = (alu_input_1 != alu_input_2);
                1: branch_bit = (alu_input_1 == alu_input_2);
                2: branch_bit = (alu_input_1 > 0);
                3: branch_bit = (alu_input_1 < 0);
                endcase
        end
        else if(instruction[15:12] == 4'd9 || instruction[15:12] == 4'd10 ) begin
                case(instruction[15:12])
                9: alu_output = signed_input_2;
                10: alu_output = alu_input_2;
                endcase
        end
end
```

Alu의 경우 inputReady 값이 negedge가 될 때마다 값을 받아오도록 구현을 해뒀고, instruction이 어떤 type이냐에 따라서 연산을 달리해야해서 다음과 같은 코드를 가지게 된다. 여기서 alu_output은 alu 모듈에서 output 되는 data 또는 address 값이 되고, signed_intput_2는 sign_extend 된 값이 연산이 진행될 때 unsigned로 진행되는 것을 막기 위해서 따로 signed wire를 생성하여 값을 넣어둔 것이다. 또한 branch_bit는 현재 single cycle cpu design에서 존재하는 bcond과 동일한 역할을 수행한다. Branch 조건이 만족하는 상태인지 아닌지로 값을 설정해준다.

d-3) mux_2to_1

```
module mux_2to_1_anytime(inputReady, bit, first, second, output_data);
        input bit;
        input[15:0] first;
        input[15:0] second;
        input inputReady;
        output reg [15:0] output_data;

        initial begin
                output_data = 0;
        end

        always@(*) begin
                if(bit == 1 ) begin
                        output_data <= second;
                end
                else begin
                        output_data <= first;
                end
        end
endmodule
```

Mux 값은 따로 timing이 없고 항상 값의 변화가 생기면 즉각적으로 값이 나가도록 설정을 해뒀고, 1일 때 두 번째 input 값이, 0일 때 첫 번째 input 값이 output으로 나가도록 구현을 해뒀다.

d-4) sign_extend

```
always@(instruction) begin
        if(instruction[15:12] == 4'b1010 && clk == 0) begin
                out_sign = instruction[11] ? {8'b1111, instruction[11:0]}:{8'b0000, instruction[11:0]};
        end
        else if(clk == 0) begin
                out_sign = instruction[7] ? {8'b11111111, instruction[7:0]}:{8'b00000000, instruction[7:0]};
        end
        else begin
                out_sign = out_sign;
        end
end
```

이 모듈의 경우 instruction이 들어오고, clk이 0일때마다 16bit으로 sign_extend 된 값을 내보내도록 구현을 해뒀다.

d-5) adder

이 module은 input으로 받아온 두 개의 data를 더 해서 output으로 내보내는 역할을 수행한다.

Discussion

이번 lab의 경우 timing 문제가 가장 문제였던 것 같다. 언제 read를 해오고, 언제 write를 하고, 이런 것들을 결정하지 않고 코드부터 짜기 시작한 것이 많은 시간을 투자하게 만들었고, 또한 다른 모듈에서 같은 timing에 값을 불러올 때 어떤 이유에서인지 값이 제대로 안 들어가는 경우들 때문에 문제가 많았던 것 같다. 예로 들자면 mux와 alu 모두 posedge clk 에 동작을 하게 해뒀고, 이 모듈을 부르는 top module에서 이 두 개의 output 값을 & 연산을 진행한다면 하나의 module은 그 전 값이 들어가고, 다른 하나의 모듈은 현재 계산된 값이 들어가서 연산이 되는 경우들이 있었다. 정확한 이유는 아직도 모르겠지만 중요 module들에서 timing에 맞게만 값을 넣어주면 된다는 것을 이해를 하였고, 다른 세부적인 모듈들은 그 input되는 값이 바뀐 경우에만 값을 register에 넣어주면 된다는 것을 이해하게 되었다.

Conclusion

결과적으로 이번 single cycle cpu의 기능을 모두 구현하였고, 테스트 벤치를 통하여 이 single cycle이 모두 돌아간다는 것을 모두 확인을 하였다. 그리고 이 lab을 통하여 IF, ID, EX, MEM, WB 등과 같은 것을 진행할 때 각각의 timing이 모두 달라야 하며 그 timing을 설정을 잘 나눠 주는 것이 cpu를 구현 할 때 중요하다는 것을 깨닫게 해준 lab이라고 생각한다.