컴퓨터 구조 Lab4

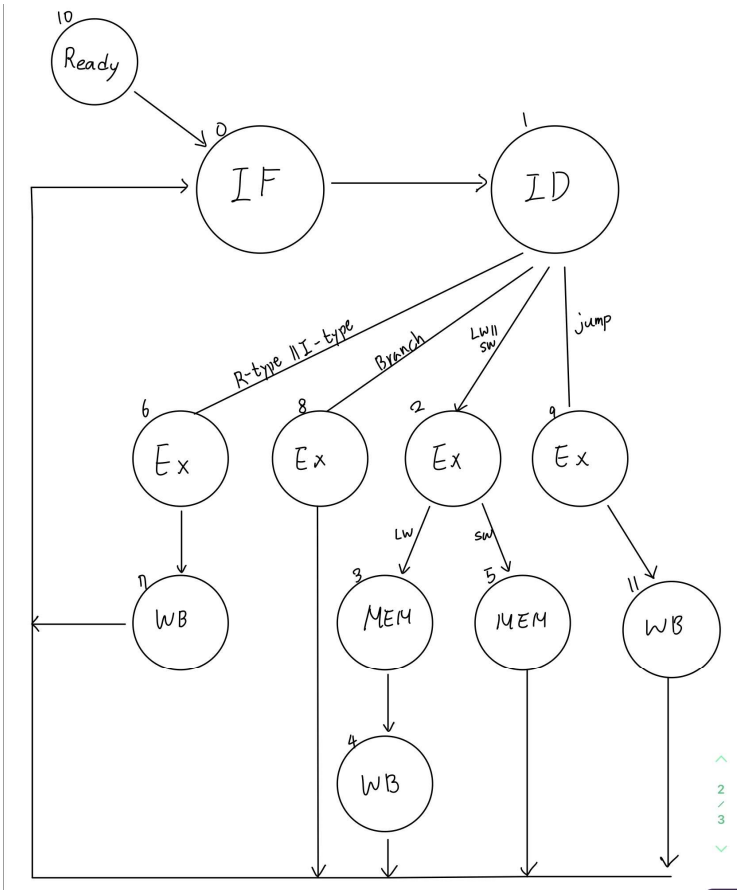
20190782 컴퓨터공학과 최서영
20190439 컴퓨터공학과 오승훈

Introduction

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

Design

이번 lab에서는 조교님께서 주신 alu, alu_control_unit, control_unit, cpu, register_fiel, util을 제외하고 datapath의 모듈들을 묶어주는 datapath_unit 모듈을 추가하여 mulit cycle CPU를 구현하였다. 다음은 state를 어떤 식으로 구성하였는지를 보여주는 그림이다.



제일 처음 ready state에서 기다리다가 reset_n이 1로 바뀌게 되면 IF state로 가게 된다. 이런 state change는 control_unit에서 하게 된다.

IF state에서는 read_m을 1로 바꾸어서 memory에서 해당 pc address에 있는 data 값을 읽어오게 된다. (여기서는 instruction) 이것을 instruction을 저장하는 instruction register에 저장한다. 동시에 ALU를 이용하여 현재 pc address에 1을 더해주고 이를 pc값에 업데이트하는 연산을 하게 된다. 그리고 다음 state인 ID state(1 state)로 가게 된다.

ID state에서는 저장된 instruction을 이용하여 해당 instruction이 어떤 type인지를 결정한다. Instruction을 크게 R, I-type, Branch type, LW || SW, jump로 나누어 그림에서 나타난 것과 같이 다음 state가 결정이 난다. 이때, 같은 state로 가더라도 control 값은 instruction마다 다를 수 있다. R-type과 I-type 같은 경우는 같이 ex state로 가지만 ALU_src_B 값이 서로 다르며, WWD, HALT 같은 경우는 wwd, halt signal를 set 해주게 된다. 또한 jump에서도 jmp와 jpr은 WB을 하지 않기 때문에 reg_write을 set하지 않지만 jal과 jrl은 reg_wirte과 write data로 들어가는 값을 결정해주는 mux의 signal을 set 해준다.

그 다음은 ID에서 결정 난 state로 이동한 다음 set된 signal들을 이용하여 연산을 진행하게 된다.

다음은 각 모듈에 대한 대략적인 설명을 해보겠다. control_unit은 input으로 받은 instruction에 따라 다음 state가 무엇인지를 결정해주고, 해당 state에 대한 signal을 set 해주는 모듈이다. 더불어 pc_write과 (pc_write_cond && bcond)을 보고 input으로 받음 pc_nxt를 pc값에 update 할지 말지를 결정 후, update한다. 이때, pc update와 state change는 모두 clock synchronized 하게 동작한다.

MUX2_to_1와 MUX4_to_1은 기존의 mux의 기능이 같다. 이러한 mux들은 memory를 읽을 때 어떤 address를 사용할지, WB state에서 register에 update할 data는 무엇일지, ALU에 들어갈 input을 결정할 때, pc값을 update할 때 어떤 값으로 update할지를 결정할 때 사용된다. 이러한 mux에서 값을 고르는 signal은 control_unit에서 오게 된다.

Datapath_unit은 control_unit으로부터 signal을 받아 아래에서 소개할 모듈들을 연결하고, signal을 전달해주는 역할을 한다.

Alu_control_unit은 alu에 넣어주는 funcCode, opcode, branchType을 결정해주는 module이다. 이때, input으로 받는 ALUOP가 현재 어떤 상태인지를 알려주는데, ALUOp가 2이면 pc+1 연산을 해주므로 add하라는 신호를 주고, 00이면 각 instruction에 맞는 funcCode를 set하고, 1이면 branch이므로 branchTyp과 funcCode를 set해준다.

Register_file은 register 값들을 읽고, 쓸 수 있도록 구현한 모듈이다. 값을 읽는 input으로 들어온 register id에 맞는 register 값을 output으로 내보내고 값을 쓰는 경우 rd의 id에 해당하는 register에 값을 대입한다. 단, clock posedge이고 reg_write 혹은 pc_to_reg signal이 1일 때만 값을 수정할 수 있다.

Sign_extend는 immediate 값의 bit수를 맞춰주는 위한 모듈이다. input으로 받은 imm값을 잘

조정한 후 sign_extend한 값을 output으로 내놓는다.

ALU는 기존의 ALU 연산과 더불어 bcond를 판단하는 기능을 수행한다. ALU는 combinational logic이고, non-blocking으로 계산된다. 이때, 들어오는 input들은 mux들에서 두개의 register가 결정되고, funcCode, branchType은 Alu_control_unit에서 계산되어 들어온다. 들어온 register를 이용하여, 들어온 funcCode, branchType에 따라 알맞은 연산을 하여 output을 결정한다.

Implementation

먼저 cpu top module은 다음과 같다.

```
control_unit control_path(reset_n, pc_next, bcond, instruction_reg[15:12], instruction_reg[5:0], clk, pc_write_cond, pc_write, i_or_d, read_m, mem_to_reg, write_m, ir_write,
pc_to_reg, pc_src, is_halted, wwd, new_inst, reg_write, alu_src_A, alu_src_B, alu_op, pc, num_inst);

datapath_top_module data_path(reset_n, clk, pc, data, i_or_d, mem_read, mem_to_reg, mem_write,
ir_write, pc_to_reg, pc_src, halt, wwd, new_inst, reg_write, alu_src_A, alu_src_B, alu_op, pc_next, bcond, memory_address, memory_data, instruction_reg);

wire [15:0] top_mux_out;

mux2_1 top(i_or_d, pc, memory_address, top_mux_out);

assign data = write_m ? memory_data : 16'bz;
assign address = top_mux_out;
always@(posedge clk) begin
        if(wwd) begin
                output_port <= memory_address;
        end
        else begin
                output_port <= output_port;
        end
end
```

Control unit과 datapath를 각각 구현을 해두고, memory read를 해야하기 때문에 pc 값과 memory_address 중 골라주는 mux가 존재하고, data는 write_m이 1인 경우에는 memory_data를 읽어오며, 아닌 경우에는 input으로 data를 읽어 올 수 있도록 한다. 그리고 또한 wwd가 1인 경우에는 현재 memory_address라는 wire를 통해서 나오는 alu output이 들어갈 수 있도록 한다.

a. Datapath

```
always @(posedge clk) begin
        if(ir_write) begin
                instruction_reg <= mem_in_data;
        end
        else begin
                instruction_reg <= instruction_reg;
        end
end
```

Datapath 내부에서는 다음과 같이 instruction을 읽어온 경우에만 mem_in_data라는 instruction을 IF가 끝날 때 instruction에 저장을 하고, 그 후 다른 mem_in_data가 존재하더라도 instruction_reg 값은 바뀌지 않도록 한다.

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

그 외에는 instruction reg 값이 바뀌지 않으므로 수행할 register 값을 register index로 넣어주는 과정을 거치게 되고, 그 후 나머지는 수업 시간에 배운 multicycle cpu의 구조와 동일한 구조를 가진다.

```
mux4_1 data(mem_to_reg, ALU_OUT, mem_in_data, pc_value, 16'h0000, out_mux_data);

wire [15:0] register_out1, register_out2;

register_file register(register_out1, register_out2, register_read_1_index, register_read_2_index, register_write_reg_index, out_mux_data, reg_write, pc_to_reg, clk);

wire [15:0] out_sign;

sign_extend sign_extend(instruction_reg, out_sign);

wire [15:0] out_mux_A, out_mux_B;

mux2_1 mux_A(alu_src_A[0], pc_value, register_out1 , out_mux_A);

mux4_1 mux_B(alu_src_B, register_out2, 16'h0001, out_sign, 16'h0000, out_mux_B);

wire [3:0] funcCode;
wire [1:0] branchType;

alu_control_unit alu_control_unit(instruction_reg[5:0], instruction_reg[15:12], alu_op, clk, funcCode, branchType);

wire [15:0] C;
wire overflow_flag, bcond;

alu ALU(out_mux_A, out_mux_B, funcCode, branchType, C, overflow_flag, bcond, clk);
```

그렇지만 하나 다른 점이 존재한다면 수업 시간에 배운 multicycle cpu 에서는 register file에 write_back을 위한 data를 골라 주는 것이 ALU_OUT 과 memory data register 에서 값을 골라서 결정을 하는 것이었다면 우리의 multicycle cpu 에서는 JAL, JRL 에서 register 2번에 넣어주는 pc+1 값을 넣어주는 과정을 진행해주기 위해서 write data를 고르는 것을 4개의 mux를 이용하며, mem_in_data라는 memory read를 통해서 구현이 되는 load때 사용이 되는 wire와 pc + 1 값을 저장하고 있고 control unit에 존재하는 pc_value, ALU의 output 값을 가지고 있는 ALU_OUT 중 하나를 output으로 내보내는 MUX가 존재하게 된다.

```
always @(posedge clk) begin
        if(alu_op != 2'b01 || alu_op != 2'b11) begin
                ALU_OUT <= C;
        end
        else begin
                ALU_OUT <= ALU_OUT;
        end
end
```

마지막으로 ALU_OUT 값은 alu 에서 output이 되는 C 값을 항상 저장하게 되는데, alu_op 값이 01, 11인 경우를 제외하고 즉 jump를 수행하는 기능들과, branch를 제외하고는 항상 ALU output 값을 ALU_OUT register에 저장을 해놓고 이 값을 다시 이용하여 사용하고, 아닌 경우에는 ALU에서 나오는 값을 저장하지 않도록 제어를 해 뒀다.

b. Control_unit

Control unit에서는 design part에서 언급 되었던 state 별로 signal을 부여를 해 뒀으며 현재 clock synchronous하게 구현이 되어 있으므로 non_blocking으로 모든 것이 구현이 되어 있다.

```
always @(posedge clk) begin
        if(state == 0) begin
                alu_src_A <= 2'b00;
                alu_src_B <= 2'b10;

                ir_write <= 1'b0;
                pc_write <= 1'b0;
                mem_read <= 1'b0;
                if(opcode == 10 || (opcode == 15 && func_code == 26)) begin
                        alu_op <= 2'b11;
                end
                else begin
                        alu_op <= 2'b00;
                end

                state <= 4'b0001;
        end
```

그리고 또한 위와 같이 state가 0인 경우에 갖게 되는 위와 같은 signal은 state 0 일 때 쓰이는 signal이 아닌 바뀌는 state에서 사용이 되는 signal이므로 state 1 에서 쓰이는 signal이 위와 같게 set을 해둔 것이다. 그 외에 모든 state들이 설정이 되어 있으며 ready state가 존재하는데 이때는 아무런 signal이 동작이 없고, reset_n이 1이 되면 그 때부터 우리의 cpu가 사용되도록 하기 위해서 state를 하나 더 추가 해둔 것이다.

```
always @(posedge clk) begin
        if(pc_write || (pc_write_cond && bcond)) begin
                pc <= pc_nxt;
        end
        else begin
                pc <= pc;
        end
        if(!reset_n) begin
                state <= 4'b1010;
                pc <= 0;
                num_inst <= 16'h0000;

                wwd <= 1'b0;
                halt <= 1'b0;
                reg_write <= 1'b0;
                pc_write_cond <= 1'b0;

                mem_read <= 1'b0;
                alu_src_A <= 2'b00;
                alu_src_B <= 2'b00;
                i_or_d <= 1'b0;
                alu_op <= 2'b00;

                ir_write <= 1'b0; //store instruction
                pc_write<= 1'b0;
                pc_src <= 1'b0;
        end
end
```

그리고 위의 내용은 pc 값이 pc_write가 1이거나 pc_write_cond와 bcondition의 연산이 1인 경우에만 pc_nxt 라는 alu를 통해 계산이 되서 나오는 값을 이용하도록 동작 수행을 시켜뒀다. pc_nxt 값이 update 될 때는 총 3번의 case가 존재하는데 1번 case는 다음 pc 값을 가지고 있도록 만들게 하기 위해서 IF stage에서 pc + 1 값을 연산을 진행하여 ID stage에서는 pc + 1 값을 가지고 있도록 하는 경우, 2번 case는 branch instruction이 만족하여서 EX stage

이후 값이 적히도록 하게 하는 pc_write_cond와 bcond가 1인 경우, 3번 case는 jmp, jal, jpr 등과 같은 jump instruction을 수행 할 때 pc_write이 1이 되어 jump WB stage에서 pc 값이 update 되는 경우 총 3번이다.

Non-conditional instruction들은 한 번의 pc update를 거치고, branch와 jump의 경우 2번씩 pc 값을 update를 할 수 있도록 만들어 뒀다.

c. Alu control_unit

Alu control unit의 경우 ALUop에 따라서 값이 조금씩 달라지는데 ALUop는 현재 10인 경우에는 IF stage 때 pc = pc + 1 연산을 수행하기 위해서 사용을 하고 있으므로 funcCode를 + 연산을 진행해주는 funcCode를 적어줬고, 그 외의 ALUop 값이 10이 아닌 경우에는 opcode에 따라서 alu에 맞는 연산을 수행할 수 있게 하는 funcCode와 branchType을 보내줄 수 있도록 구현을 해뒀다.

```
always@(*) begin
        if(ALUOp == 2'b10) begin
                funcCode <= 4'd0;
        end
        else if(ALUOp == 2'b00) begin
                if(opcode == 4'b0000 || opcode == 4'b0001 || opcode == 4'b0010 || opcode == 4'b0011) begin
                        branchType <= opcode[1:0];
                        funcCode <= 4'd9; //branch funcCode == 9
                end
                else if(opcode == 4'd15) begin
                        if(funct == 6'd0) begin
                                funcCode <= 4'd0; // ADD
                        end
```

위의 설명에 해당하는 부분은 다음과 같다.

d. Register file

레지스터 파일의 경우 저번 lab과 동일하지만 clock synchronous 할 수 있게 변경을 한 것 제외하고는 변경한 부분이 없다.

```
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 if(pc_to_reg) begin
        register[write_reg] <= write_data;
        end
        else begin
        register[write_reg] <= register[write_reg];
        end
end
```

e. Sign extend

Sign extend 역시 바뀐 부분이 없지만 저번에는 clock 값에도 영향을 받게끔 구현을 해뒀는

데 이번에는 그러한 clock값에 대한 제어를 모두 빼고, combinational logic으로 구현을 했다.

```
always@(*) begin
        if(instruction[15:12] == 4'b1001 || instruction[15:12] == 4'b1010) begin
                out_sign = instruction[11] ? {8'b1111, instruction[11:0]}:{8'b0000, instruction[11:0]};
        end
        else begin
                out_sign = instruction[7] ? {8'b11111111, instruction[7:0]}:{8'b00000000, instruction[7:0]};
        end
end
```

f. ALU

Alu 역시 바뀐 부분이 없지만 저번 lab 에서는 ALU_control unit을 사용하지 않았기 때문에 control signal을 받아서 사용하는 것과 overflow_flag를 사용하는 것을 제외하고는 변경된 것이 존재하지 않는다.

```
always@(*) begin
        if(func_code == 4'd0) begin
                C = A + B;
                overflow_flag = ((A[15] & B[15] & (~C[15])) | ((~A[15]) & (~B[15]) & C[15]));
        end
        else if(func_code == 4'd1) begin
                C = A - B;
                overflow_flag = ((A[15] & (~B[15]) & (~C[15])) | ((~A[15]) & B[15] & C[15]));
        end
        else if(func_code == 4'd2) begin
                C = A & B;
        end
```

g. Util

Util file에서 살짝 변경한 부분이 있는데 원래 조교님이 제공해주신 코드에서는 mux2_1 에서 selection bit이 2 bit로 구현이 되어 있었지만 selection bit는 1bit이면 충분하기 때문에 1bit으로 바꾸고 구현을 진행하였다.(원래는 input [1:0] sel 이었음)

```
module mux2_1 (sel, i1, i2, o);
    input sel;
    input [15:0] i1, i2;
    output reg [15:0] o;

    always @ (*) begin
       case (sel)
          0: o = i1;
          1: o = i2;
       endcase
    end

endmodule
```

Discussion

이번 랩에서는 single cycle 까지 blocking과 non blocking을 혼용해서 쓰던 우리 조의 경우에는 매우 힘든 lab이었던 것 같다. Non_blocking을 사용하게 되면 posedge 값을 기준으로 그 전의 변경되기 전 값이 들어가는 것과 timing을 생각해주는 것이 state를 그리는 것에서부터 혼동이 왔던 것 같다. 그리고 다른 것보다 처음 보는 instruction들이 등장 한 것이 매우 낯설게 느껴지는 것 같았고, 가장 걱정 되었던 것은 저번 lab에서 demo를 하면서 알게 된 것이지만 Model Sim을 사용해서 구현을 하고, 파일을 옮기는 과정에서 제일 최신 것을 기준으로 옮겨지는 것이 아니라 그 전 것들이 옮겨지면서 살짝의 오류가 난 것이 존재한다. 그러한 것을 방지하는 것을 어떻게 해야

할까 라는 고민도 많이 하게 된 lab이었던 것 같다.

또한 이상적인 multicycle cpu 였다면 사실 jump의 경우 IF EX WB의 3단계만 거치면 되지만 우리의 multicycle에서는 IF ID EX WB 이 4단계를 거쳐야한다는 것이 아쉬움으로 작용하는 것 같고, clock synchronous 해야 한다는 것이 어디까지 clock synchronous 해야하는 것인지에 대해서 혼동이 있어서 더 이상 code를 고치지는 못했던 것 같다.

마지막으로 overflow를 만드는 것과 halting을 보면서 pipelining에서 exception을 구현을 하도록 하겠구나 라는 걱정이 들면서 이번 lab의 code를 마무리 하였던 것 같다.

Conclusion

결과적으로 이번 multicycle cpu의 기능을 모두 구현을 하였고, 테스트 벤치를 통하여 이 multi cycle의 모든 instruction이 돌아감을 확인을 하였다. 그리고 이번 lab을 통하여 non-blocking과 blocking의 차이점과 timing 등을 확실히 이해할 수 있는 시간이 되었던 것 같고, 다음 pipeline cpu에서는 이 blocking과 non blocking을 더 잘 구분을 하여 code를 구현할 수 있을 것 같다는 생각이 들게 해준 lab이었다.