컴퓨터구조 (CSED311)

Lab 3 Report

20200220 오상윤

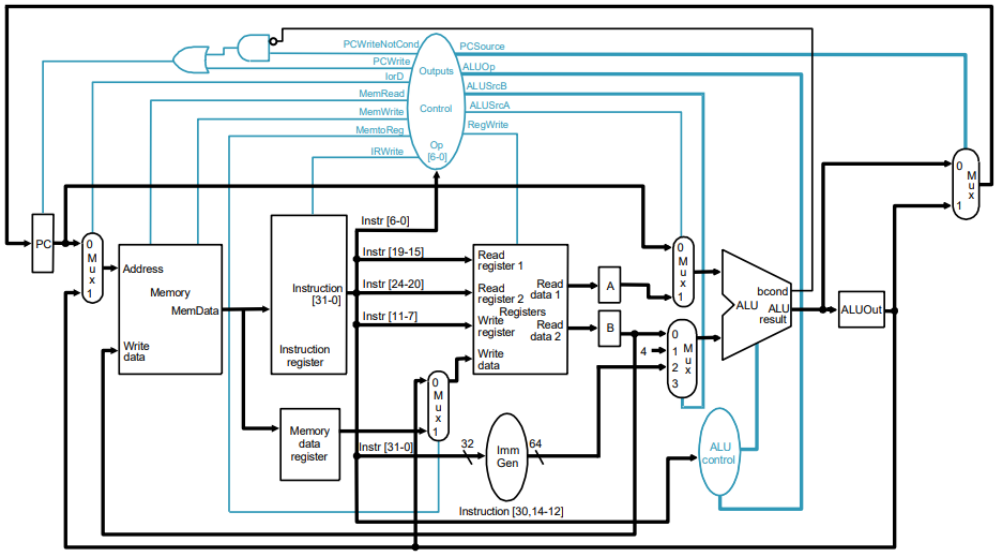
(1) Introduction

베릴로그를 이용하여 multi cycle RISC-V cpu를 구현하는 lab이었다. multi cycle cpu는 모든 instruction 실행 시 동일하게 한 cycle을 사용했던 single cycle cpu와 달리 높은 clock frequency를 사용하며 각 instruction type에 따라 필요한 만큼만 다른 수의 cycle 동안 작동한다. single cycle cpu는 모든 instruction이 가장 느린 instruction에 맞춰 실행되는 반면, multi cycle cpu는 각 instruction이 필요한 만큼의 시간만 소요하기 때문에 더 효율적이다.

주어진 cpu module은 지난 lab2의 single cycle cpu와 마찬가지로 input으로 reset과 clk를, output으로 machine의 중지를 나타내는 is_halted를 갖는다. clk의 positive edge가 한 cycle을 나타내며, cpu module은 ECALL instruction을 만났을 때 x17 레지스터의 값이 10인 경우 is_halted의 값을 1로 출력하여 machine의 중지를 알린다. 처리 가능한 instruction은 R-type, I-type, S-type, B-type, J-type이며, LOAD instruction은 LW, STORE instruction은 SW만을 처리 가능하다. 파일 구성은 cpu module을 담고 있는 cpu.v, RegisterFile module을 담고 있는 RegisterFile.v, Memory module을 담고 있는 Memory.v, 나머지 PC module을 담고 있는 PC.v, ControlUnit module, MicroStateMachine module을 담고 있는 ControlUnit.v, ImmediateGenerator module을 담고 있는 ImmediateGenerator.v, PC module을 담고 있는 ALU.v, ControlUnit module, ALU module을 담고 있는 ALU.v, constant들의 define을 담고 있는 opcodes.v로 구성되어 있다.

(2) Design

수업 자료에 있는 Multi-Cycle Datapath Design을 토대로 하여 전체적인 module을 Design 하였다.



1. PC module

   매 cycle마다 pc 값을 변경하는 module이다. 초기의 default pc값은 0x0으로 설정하며, 이후 PCwrite signal을 받을 때마다 logic을 통해 새롭게 계산된 새로운 pc 값 next_pc 를 current_pc 값으로 넣어준다. cpu는 current_pc를 받아 Memory module에 input으로 전달하여 해당 line의 instruction을 얻어낸다.

2. Memory module

   메모리와 동일하게 동작하도록 설계되어 있으며 cpu로부터 addr을 받아 해당 addr의 data를 읽어 dout으로 내보낸다. single cycle cpu 구현 때와 달리, instruction과 data 를 하나의 memory module에서 처리하여 IF 단계에서는 memory에서 instruction 을 읽고, MEM 단계에서는 momery에서 data를 읽거나 쓰도록 하였다. 초기에 지정된 txt 파일로부터 instruction을 읽어와 메모리와 같이 동작하는 배열에 저장해 놓는다. Control unit으로부터 IorD signal을 받아 PC값을 addr로 받거나 ALU를 통해 계산된 메모리 주소값을 addr로 받는다. cpu는 dout으로 나온 data를 받아 memory data register에 저장하며, IRwrite signal이 켜져 있는 경우 Instruction register에 해당 data(현재 pc가 가리키는 instruction)을 저장한다.

3. ControlUnit module

   cpu로부터 instruction에 따른 opcode를 받아 opcode에 따라 control signal을 적절하게 설정한다. 내부적으로 MicroStateMachine을 갖고 있어, 각 instruction type에 따라 다른 state들을 거치게 되는데 각 state마다 cpu내의 각 module 들에게 적절한 signal을 전달하여 instruction type에 따라 다른 수의 cycle 동안 multi-cycle cpu가 동

작하도록 한다. cycle 마다 MicroStateMachine을 통해 계산된 다음 state 값으로 state가 변경된다. state는 0부터 13까지 총 14개의 state를 설계하였으며, 자세한 signal 구현은 implementation에서 설명하도록 하겠다.

4. MicroStateMachine

   현재 state와 instruction type을 바탕으로 다음 state를 계산하여 ControlUnit에게 전달한다. 총 14개의 state를 가지고 있으며, IF와 ID 단계에 해당하는 0, 1 state는 instruction type에 상관없이 모두 거치게 되어있고, 이후 state 부터는 instruction type에 따라 다른 state를 거치게 된다.

5. RegisterFile module

   register들의 값을 담고 있는 module로, source register 1, 2의 번호를 받아 각각의 register의 값을 output으로 내보낸다. write_enable이 1인 경우 destination register의 번호에 해당하는 register에 접근하여 rd_din을 통해 입력으로 들어온 값을 register에 기록한다. x0의 register에 기록을 시도하는 경우에는 기록되지 않는다.

6. ImmediateGenerator module

   instruction에서 opcode를 뽑아내 각 opcode에 따라 적절하게 immediate value를 생성하는 module이다. I-type, S-type, B-type, J-type에 따라 immediate value의 생성 방법이 다르며, 구체적인 생성은 아래 implemetation에서 설명할 것이다. cpu는 생성된 immediate value를 ALU의 입력 값 목록 중 하나로 설정한다.

7. ALUControlUnit module

   ControlUnit으로부터 ALUOp를 input으로 받고, 현재 instruction에서 opcode, func3, func7을 뽑아내 ALU가 수행할 operator를 적절하게 지정한다. ALUOp가 00인 경우는 ADD를, 01인 경우는 SUB를 지정하고, 이외의 경우(10인 경우)에는 func3, func7의 값을 통해 operator을 지정한다. ALUControlUnit module이 alu_op를 통해 operator를 지정하면, ALU module이 이를 input으로 받아 해당하는 연산을 수행한다.

8. ALU module

   ALUControlUnit으로부터 alu_op를 받아 input들에 대해 적절한 연산을 수행한 다음, 연산 결과를 내보낸다. single cycle cpu 구현 때와 달리, pc 값을 계산할 때도 별도의 module을 사용하는 대신 ControlUnit으로부터 signal을 받아 input들을 적절히 조정하여 ALU module을 reuse한다. 이때 B-type의 instruction인 경우 bcond를 지정해줘야 하므로, sub 연산인 경우 func3의 값을 참고하여 연산의 결과에 따라 bcond를 적절하게 지정해주도록 하였다. bcond 값은 ControlUnit으로 전달되어 BRANCH instruction인 경우 다음 pc값을 계산하기 위한 signal이 적절하게 생성될 수 있도록

한다.

(3) Implementation

1. PC module

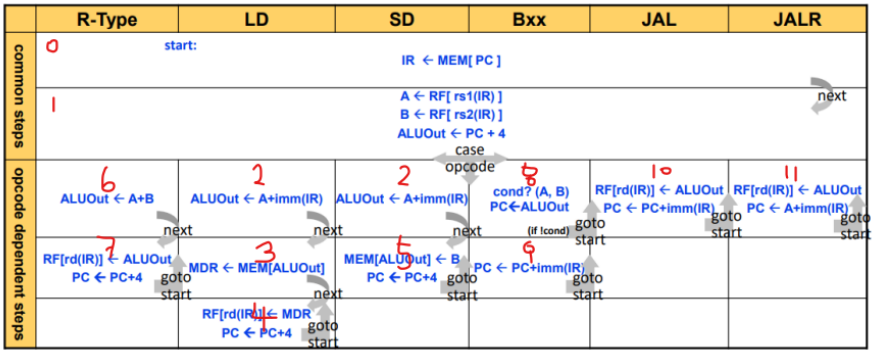
input으로 reset, clk, change_pc, next_pc 를 받으며, output으로 current_pc를 갖는다. change_pc는 pc_write가 1이거나 pc_write_cond와 alu의 !bcond값이 1인 경우 1의 값을 갖는 input이다. clock synchronous 하게 동작하며 clk의 positive edge마다 change_pc값이 1인 경우 current_pc의 값을 next_pc로 바꾸어 준다. default PC의 값이 0x0이므로 reset이 1인 경우, pc를 32'b0으로 초기화하도록 하였다.

2. Memory module

input으로 reset, clk, addr, din, mem_read, mem_write를 받으며, output으로 dout을 갖는다. mem이라는 reg 배열을 가지고 있어 실제 메모리처럼 동작하도록 구현되어 있으며, 초기에 (reset이 1일 때) readmemh를 통해 지정된 경로에 존재하는 txt 파일에 접근하여 txt 파일 속 binary format의 instruntion들을 mem 배열에 저장한다. 이후 asynchrounous하게 mem_read가 1인 경우, addr 값에 따라 mem 배열에 접근하여 해당하는 index의 원소 값을 dout으로 전달한다. mem_write가 1인 경우, addr 값에 따라 clock synchronous하게 clk가 positive edge일 때 mem 배열에 접근하여 해당하는 index의 원소에 din 값을 넣어준다.

3. ControlUnit module

input으로 opcode, clk, reset, alu_bcond를 받으며, output으로 pc_write_cond, pc_write, i_or_d, mem_read, mem_write, mem_to_reg, ir_write, pc_source, ALUOp, alu_src_B, alu_src_A, reg_write, is_ecall을 갖는다. cpu로부터 받는 opcode가 ECALL을 가리키는 경우 asynchronous하게 is_ecall을 1로 만든다. multi-cycle 구현을 위해 0부터 13까지 총 14개의 state를 설정하였으며, opcode를 통해 얻어낸 instruction type에 따라 각기 다른 state들을 거치며 다른 수의 cycle동안 instruction이 실행되도록 하였다.



각 state는 수업자료의 Combined RT Sequencing 슬라이드의 table을 참고하여 설정하였으며, 각 칸 마다 state 번호를 지정하여 state로 사용하였다. table에 등장하지 않은 I-type의 Arithmetic immediate type을 위한 state 12와 ECALL을 위한 state 13을 추가적으로 설정하였다. Control unit은 현재 state를 바탕으로 각 signal들을 asynchronous하게 설정하여 다른 module들이 적절하게 동작하도록 한다. 또한 MicroStateMachine을 통해 계산된 next_state를 clock synchronous하게 cur_state에 넣어준다. ControlUnit module의 code는 다음과 같다.

```
module ControlUnit(
    input [6:0] opcode,
    input clk,
    input reset,
    input alu_bcond,
    output reg pc_write_cond,
    output reg pc_write,
    output reg i_or_d,
    output reg mem_read,
    output reg mem_write,
    output reg mem_to_reg,
    output reg ir_write,
    output reg pc_source,
    output reg [1:0] ALUOp,
    output reg [1:0] alu_src_B,
    output reg alu_src_A,
    output reg reg_write,
    output is_ecall);

    reg [5:0] cur_state=0;
    wire [5:0] next_state;

    assign is_ecall=(opcode==`ECALL);

    always @(*) begin
        pc_write_cond=0;
        pc_write=0;
        i_or_d=0;
        mem_write=0;
        mem_read=0;
        mem_to_reg=0;
        ir_write=0;
        pc_source=0;
        ALUOp=0;
        alu_src_B=0;
        alu_src_A=0;
        reg_write=0;
        case(cur_state)
            0: begin
                mem_read=1;
                i_or_d=0;
                ir_write=1;
            end
            1: begin
                alu_src_A=0;
                alu_src_B=2'b01;
                ALUOp =2'b00;
            end
            2: begin
                alu_src_A=1;
                alu_src_B=2'b10;
                ALUOp=2'b00;
            end
            3: begin
                mem_read=1;
                i_or_d=1;
            end
            4: begin
                reg_write=1;
                mem_to_reg=1;
                //
                alu_src_A=0;
                alu_src_B=2'b01;
                ALUOp=2'b00;
                pc_write=1;
                pc_source=0;
            end
            5: begin
                mem_write=1;
                i_or_d=1;
                //
                alu_src_A=0;
                alu_src_B=2'b01;
                ALUOp=2'b00;
                pc_write=1;
                pc_source=0;
            end
            6: begin
                alu_src_A=1;
                alu_src_B=2'b00;
                ALUOp=2'b10;
            end
```

```
            7: begin
                reg_write=1;
                mem_to_reg=0;
                //
                alu_src_A=0;
                alu_src_B=2'b01;
                ALUOp=2'b00;
                pc_write=1;
                pc_source=0;
            end
            8: begin
                alu_src_A=1;
                alu_src_B=2'b00;
                ALUOp=2'b01; // branch 일때 ALUOp 01
                // pc_write_cond=1; // branch 일때

                pc_source=1; //pc+4 가 ALUOut에 저장되어 있으므로
                pc_write=!alu_bcond;
            end
            9: begin
                alu_src_A=0;
                alu_src_B=2'b10;
                ALUOp=2'b00;
                pc_write=1;
                pc_source=0;
            end
            10: begin
                mem_to_reg=0;
                reg_write=1;
                //
                alu_src_A=0;
                alu_src_B=2'b10;
                ALUOp=2'b00;
                pc_write=1;
                pc_source=0;
            end
            11: begin
                mem_to_reg=0;
                reg_write=1;
                //
                alu_src_A=1;
                alu_src_B=2'b10;
                ALUOp=2'b00;
                pc_write=1;
                pc_source=0;
            end
            12: begin
                alu_src_A=1;
                alu_src_B=2'b10;
                ALUOp=2'b10;
            end
            13: begin
                alu_src_A=0;
                alu_src_B=2'b01;
                ALUOp=2'b00;
                pc_write=1;
                pc_source=0;
            end
        endcase
    end
    always @(posedge clk) begin
        if (reset) begin
            cur_state<=0;
        end
        else begin
            cur_state<=next_state;
        end
    end
    MicroStateMachine msm(
        .opcode(opcode),
        .clk(clk),
        .reset(reset),
        .alu_bcond(alu_bcond),
        .cur_state(cur_state),
        .next_state(next_state)
    );
endmodule
```

4. MicroStateMachine

input으로 opcode, alu_bcond, cur_state를 받아 output으로 next_state를 내보낸다. asynchronous하게 cur_state와 opcode에 따라 다음 state를 계산한다. Branch instruction의 경우, alu_bcond 값에 따라 alu_bcond가 1이면 state 9로 이동하여 pc 값을 pc+imm으로 변경하도록 하고, alu_bcond가 0이면 state 0으로 이동한다. MicroStateMachine module의 코드는 아래와 같다.

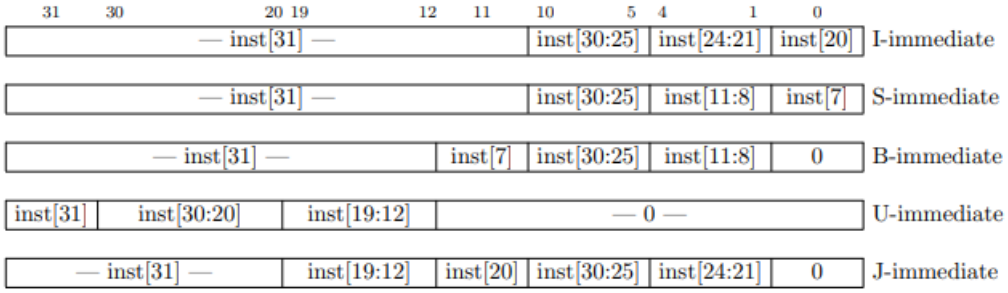
```
module MicroStateMachine (input [6:0] opcode,
                          input clk,
                          input reset,
                          input alu_bcond,
                          input [5:0] cur_state,
                          output reg [5:0] next_state);
    always @(*) begin
        case(cur_state)
            0: begin
                next_state=1;
            end
            1: begin
                case(opcode)
                    `ARITHMETIC: next_state=6;
                    `ARITHMETIC_IMM: next_state=12;
                    `LOAD: next_state=2;
                    `STORE: next_state=2;
                    `BRANCH: next_state=8;
                    `JAL: next_state=10;
                    `JALR: next_state=11;
                    `ECALL: next_state=13;
                endcase
            end
            2: begin
                case(opcode)
                    `LOAD: next_state=3;
                    `STORE: next_state=5;
                endcase
            end
            3: begin
                next_state=4;
            end
            4: begin
                next_state=0;
            end
            5: begin
                next_state=0;
            end
            6: begin
                next_state=7;
            end
            7: begin
                next_state=0;
            end
            8: begin
                if(alu_bcond) begin
                    next_state=9;
                end
                else begin
                    next_state=0;
                end
            end
            9: begin
                next_state=0;
            end
            10: begin
                next_state=0;
            end
            11: begin
                next_state=0;
            end
            12: begin
                next_state=7;
            end
            13: begin
                next_state=0;
            end
        endcase
    end

endmodule
```

5. RegisterFile module

intput으로 reset, clk, rs1, rs2, rd, rd_din, write_enable을 받으며, output으로 rs1_dout, rs2_dout을 갖는다. rf라는 reg배열을 가지고 있어 이 배열이 register들의 값들을 담고 있다. 초기에 (reset이 1일 때) rf 배열을 모두 0으로 초기화하고, x2 register는 그 값을 0x2ffc로 초기화한다. rs1, rs2, rd는 각각 source register 1, 2, destination register의 number를 나타낸다. asynchronous하게 rs1과 rs2값에 따라 해당 register의 값을 rs1_dout과 rs2_dout에게 넘겨준다. 또한 clock synchronous하게 clk가 positive edge일 때 write_enable의 값이 1인 경우 rd에 해당하는 register에 rd_din으로 들어온 값을 기록한다. 만약 x0에 값을 기록하고자 하는 경우에는 x0는 항상 0의 값을 가지므로 값을 기록하지 않는다.

6. ImmediateGenerator module

input으로 part_of_inst, output으로 imm_gen_out을 갖는다. cpu로부터 받는 part_of_inst는 32 bit의 instruction이며 하위 비트 7개로 opcode를 뽑아내 opcode에 따라 asynchronous하게 instruction으로부터 immediate value를 적절하게 생성하여 imm_gen_out으로 내보낸다. opcode를 통해 type을 얻어낸 다음 각 type에 따라 immediate value의 생성방법은 아래의 riscv-spec 자료의 그림을 따른다.



7. ALUControlUnit module

input으로 part_of_inst, ALUOp, output으로 alu_op를 갖는다. cpu로부터 받는 part_of_inst는 32 bit의 instruction이며, instruction에서 opcode, func3, func7을 뽑아내 ALUOp값과 func7, func3를 고려하여 asynchronous하게 alu에게 전달할 operator인 alu_op를 적절하게 지정하여 내보낸다.

8. ALU module

input으로 alu_op, alu_in_1, alu_in_2, funct3를 받으며, output으로 alu_result, alu_bcond를 갖는다. asynchronous하게 동작하는 combinational logic이며, alu_op에 따라 operator에 맞게 alu_in_1과 alu_in_2간의 연산을 수행하여 alu_result로 내보낸다. 이때 sub 연산을 수행한 경우, funct3의 값에 따른 Branch 유형에 따라 bcond를 적절하게 지정하도록 하였다. sub 연산 이외의 연산의 경우 연산의 결과와 상관없이 bcond를 0으로 지정한다.

(4) Discussion

lab2 에서 주어졌던 basic_ripes 와 loop_ripes examples 들을 multi-cycle cpu 에 입력하여 돌려 보았다. Total clock cycles 수는 basic_mem 을 돌렸을 때는 116 cycles, loop_mem 을 돌렸을 때는 977 cycles 가 나왔다. 같은 examples 에 대해 single-cycle cpu 는 basic_mem 을 돌렸을 때는 28 cycles, loop_mem 을 돌렸을 때는 222 cycles 가 나왔다. multi-cycle cpu 는 한 cycle 마다 한 state 를 돈 것이므로, single-cycle cpu 의 결과에 load instruction 의 state 수인 5 를 곱해준 값 140, 1110 과 multi-cycle cpu 와 성능 비교가

가능하다. 결국 multi-cycle cpu 가 basic_mem 에 대해서는 24 cycles 가, loop_mem 에 대해서는 133 cycles 만큼 적으므로, 성능 향상이 이루어진 것을 확인할 수 있다.

(5) Conclusion

주어진 skeleton code 에서 적절하게 code 를 디자인하여 Multi-Cycle CPU 를 구현해보면서 Single-Cycle CPU 와의 차이점을 익히고, 직접 주어진 testbench 를 통해 성능을 비교해보면서 성능의 개선을 확인해볼 수 있었다.