CSED311 Lab03

-Multi Cycle CPU-

Team ID: 2

20180740 염재후

1. Introduction

이번 랩의 목표는 multi cycle CPU를 구현하는 것이다. Instruction 하나에 cycle하나를 할당하는 single cycle cpu와 다르게, multi cycle CPU는 한 cycle에 functional unit을 사용하는 것을 목표로 한다. 따라서 하나의 instruction이 처리되는 동안, 각 clock cycle마다 state가 옮겨가는 Finite State Machine이라고 말할 수 있다. Multi cycle CPU에 대해서 더 자세한 내용은 Discussion에서 다루도록 한다.

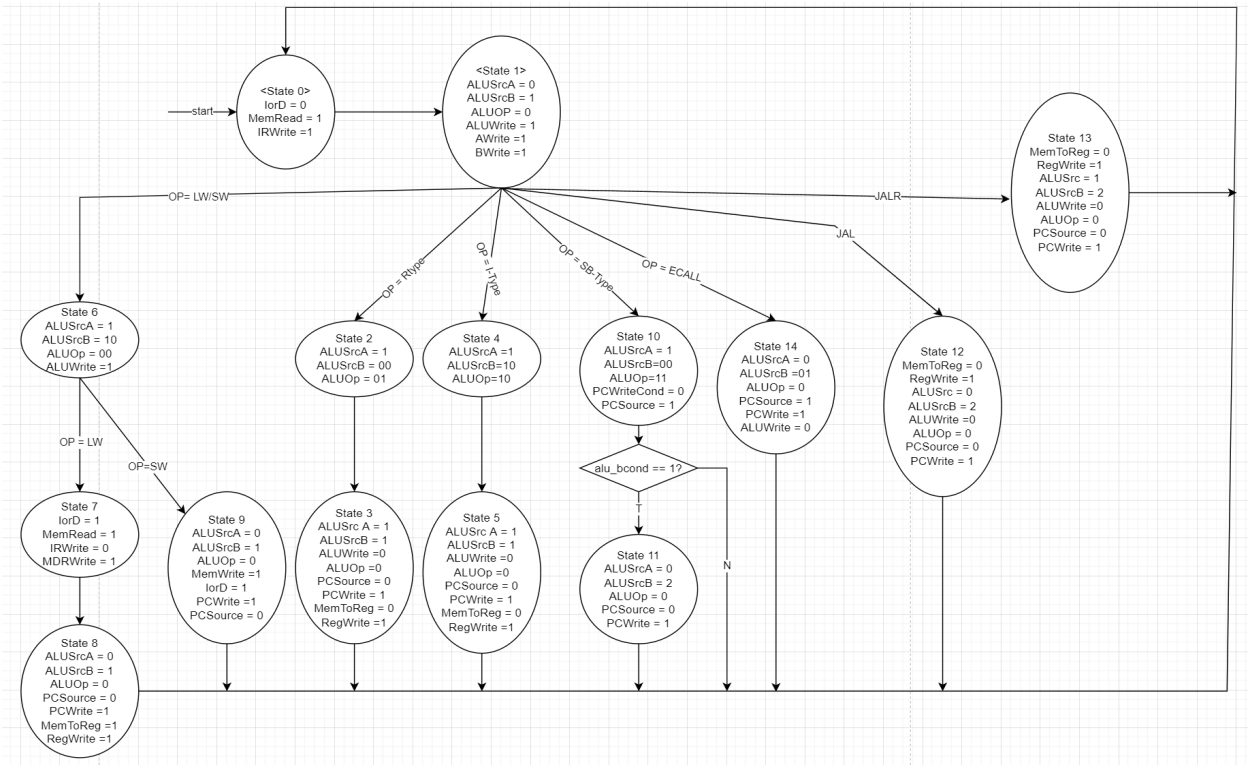
2. Design

1. Register Transfer Sequence Table

   다음은 Lecture note의 RT sequencing table을 기반으로 하여 ecall을 추가적으로 처리할 수 있게 정리한 RT sequence table이다. 이를 기반으로 FSM의 state를 정할 수 있으며, 동시에, 필요한 signal들을 정리할 수 있다.

<table>
<tr><th>RT Sequencing</th><th>R-Type</th><th>I-Type</th><th>LD</th><th>SD</th><th>SB-Type</th><th>JAL</th><th>JALR</th><th>EALL</th></tr>
<tr><td>IF Stage</td><td colspan="8">IR <- MEM[PC]</td></tr>
<tr><td>ID Stage</td><td colspan="7">A <- RF[rs1(IR)]
B <- RF[rs2(IR)]
ALUOut <- PC+4
ecall</td><td>A <- RF[x17]
B <- 10
ALUOut <- PC+4</td></tr>
<tr><td>EX Stage</td><td>ALUOut <- A OP B</td><td>ALUOut <- A OP Imm</td><td colspan="2">ALUOut <- A+Imm(IR)</td><td>cond(A,B)?
Next state : PC<-PC+4</td><td>RF[rd(IR)] <- ALUOut
PC <- PC+Imm(IR)</td><td>RF[rd(IR)] <- ALUOut
PC <- A+Imm(IR)</td><td>PC <- ALUOut</td></tr>
<tr><td>MEM Stage</td><td></td><td></td><td>MDR <- MEM[ALUOut]</td><td>MEM[ALUOut] <- B
PC <- PC+4</td><td>PC <- PC + Imm(IR)</td><td></td><td></td><td></td></tr>
<tr><td>WB Stage</td><td>RF[rd(IR)] <- ALUOut
PC <- PC + 4</td><td>RF[rd(IR)] <- ALUOut
PC <- PC + 4</td><td>RF[rd(IR)] <- MDR
PC <- PC+4</td><td></td><td></td><td></td><td></td><td></td></tr>
</table>

2. Finite State Machine Design Regarding RT Sequence Table



다음은 RT Sequence Table을 기반으로 각 state와 각 state마다 필요한 signal, 그리고 transition에 필요한 조건을 정리한 table이다. 특이점으로는 is ecall signal의 경우는 is halted signal에 continusous assignment로 처리하기 위해서 continuous하게 처리하여, 이 finite state machine design에는 들어가지 않았다.

3. Submodules

A. ALU

ALU는 combinational logic만 존재하는 asynchronous한 module로, 들어온 signal과 input값에 맞게 값을 처리해서 내보내주는 역할을 수행한다.

B. ALU Control Unit

ALU Control Unit의 경우,

00	ADD
01	R type
10	I Type
11	SB type

다음과 같은 ALUOp signal을 받아서, ALU가 수행해야할 연산을 4 bit ALU control input으로 특정한다. 따라서 asynchronous하게 주어진 값들을 내

보내준다. I type과 R type을 분리한 이유는, funct7 code에 있는 second MSB에 따라서, R type의 연산의 종류가 달라지기 때문이다.

C. Immediate Generator

Immediate Generator는 asynchronous한 모듈로, instruction type에 따라서, Imm field를 모아서, 32bit Immediate로 내보낸다.

D. PC

PC는 대표적인 PVS중 하나로, 그 값이 synchronous하게 바뀌어야 한다. 단, PC가 바뀌지 않아야 할 경우에는 바뀌지 않아야 하기 때문에 이를 nextPCWrite signal을 통해 처리해준다.

E. Register File

Register File의 경우, 이번 Lab에서는 주어진 모듈이다. Register 역시 synchronous한 모듈이다. Register data가 안전해야 하기 때문이며, 또한 RegWrite signal이 synchronous하게 생성되기 때문이다.

F. Memory

Memory 역시, clock synchronous한 모듈이다. MemRead 그리고 MemWrite signal이 synchronous하게 생성되며, 동시에, 그 데이터의 안정성을 보장해야 하기 때문이다.

G. Control Unit

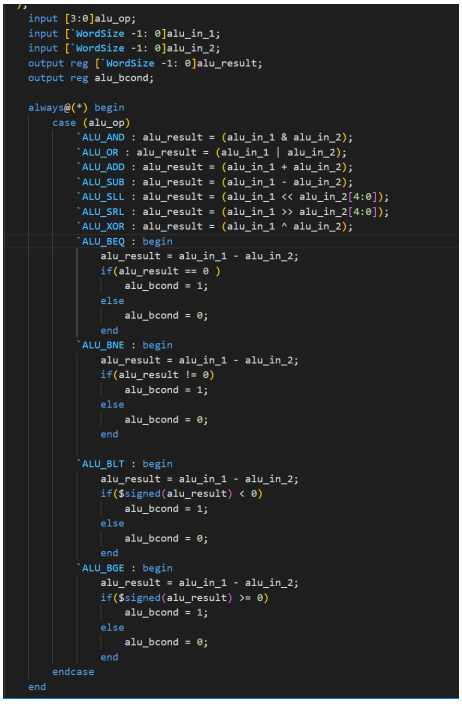
Control Unit은 각 State에 맞는 Control Signal을 생성해서 내보내는 역할을 하는 모듈이다. 따라서, Control Unit은 Synchronous한 모듈이어야 하는데, Signal을 생성하는 것은 asynchronous한 작업이지만, state transition을 적용하는 것은 clock synchronous하게 적용되어야 하기 때문이다.

H. MicroSeqeuncer

Micro sequencer 모듈의 경우, next state를 만들어주는 asynchronous한 모듈이다. Control unit에 기록된 현재 state와 그 state로 만들어진 AddrCtl signal에 기반하여, 다음 state를 정해준다. AddrCtl signal은 주어진 Appendix C를 기반으로 정해졌다.

3. Implementation

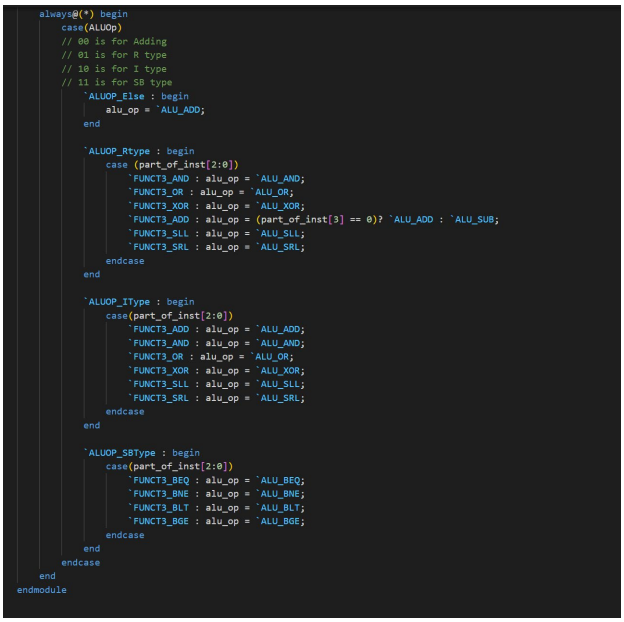
A. ALU



```
input [3:0]alu_op;
input [`WordSize -1: 0]alu_in_1;
input [`WordSize -1: 0]alu_in_2;
output reg [`WordSize -1: 0]alu_result;
output reg alu_bcond;

always@(*) begin
    case (alu_op)
        `ALU_AND : alu_result = (alu_in_1 & alu_in_2);
        `ALU_OR : alu_result = (alu_in_1 | alu_in_2);
        `ALU_ADD : alu_result = (alu_in_1 + alu_in_2);
        `ALU_SUB : alu_result = (alu_in_1 - alu_in_2);
        `ALU_SLL : alu_result = (alu_in_1 << alu_in_2[4:0]);
        `ALU_SRL : alu_result = (alu_in_1 >> alu_in_2[4:0]);
        `ALU_XOR : alu_result = (alu_in_1 ^ alu_in_2);
        `ALU_BEQ : begin
            alu_result = alu_in_1 - alu_in_2;
            if(alu_result == 0 )
                alu_bcond = 1;
            else
                alu_bcond = 0;
            end
        `ALU_BNE : begin
            alu_result = alu_in_1 - alu_in_2;
            if(alu_result != 0)
                alu_bcond = 1;
            else
                alu_bcond = 0;
            end

        `ALU_BLT : begin
            alu_result = alu_in_1 - alu_in_2;
            if($signed(alu_result) < 0)
                alu_bcond = 1;
            else
                alu_bcond = 0;
            end
        `ALU_BGE : begin
            alu_result = alu_in_1 - alu_in_2;
            if($signed(alu_result) >= 0)
                alu_bcond = 1;
            else
                alu_bcond = 0;
            end
    endcase
end
```

ALU의 디자인은 single cycle cpu와 다르지 않다. 주어진 alu_op signal에 맞게 알맞은 연산을 고른 후, 들어온 input값에 연산을 수행해서, 그 결과를 내보내 준다.

B. ALU Control Unit



```
    always@(*) begin
        case(ALUOp)
        // 00 is for Adding
        // 01 is for R type
        // 10 is for I type
        // 11 is for SB type
            `ALUOP_Else : begin
                alu_op = `ALU_ADD;
            end

            `ALUOP_Rtype : begin
                case (part_of_inst[2:0])
                    `FUNCT3_AND : alu_op = `ALU_AND;
                    `FUNCT3_OR : alu_op = `ALU_OR;
                    `FUNCT3_XOR : alu_op = `ALU_XOR;
                    `FUNCT3_ADD : alu_op = (part_of_inst[3] == 0)? `ALU_ADD : `ALU_SUB;
                    `FUNCT3_SLL : alu_op = `ALU_SLL;
                    `FUNCT3_SRL : alu_op = `ALU_SRL;
                endcase
            end

            `ALUOP_IType : begin
                case(part_of_inst[2:0])
                    `FUNCT3_ADD : alu_op = `ALU_ADD;
                    `FUNCT3_AND : alu_op = `ALU_AND;
                    `FUNCT3_OR : alu_op = `ALU_OR;
                    `FUNCT3_XOR : alu_op = `ALU_XOR;
                    `FUNCT3_SLL : alu_op = `ALU_SLL;
                    `FUNCT3_SRL : alu_op = `ALU_SRL;
                endcase
            end

            `ALUOP_SBType : begin
                case(part_of_inst[2:0])
                    `FUNCT3_BEQ : alu_op = `ALU_BEQ;
                    `FUNCT3_BNE : alu_op = `ALU_BNE;
                    `FUNCT3_BLT : alu_op = `ALU_BLT;
                    `FUNCT3_BGE : alu_op = `ALU_BGE;
                endcase
            end
        endcase
    end
endmodule
```

ALU의 asynchronous logic design이다. 이는 single cycle과 매우 유사하지만, 조금 달라진 점이 생겼다. single cycle cpu에서는 instruction type에 따라서 먼저 case를 진행했지만, 이번에는 각 instruction type을 group으로 만들어 control unit에서 ALUOp signal을 주기 때문에, 주어진 ALUOp에 맞게 case문으로 특정한 연산을 정해준다.

C. Immediate Generator

```
module ImmediateGenerator(
    part_of_inst,
    imm_gen_out
);
    input [`WordSize -1: 0] part_of_inst;
    output reg[`WordSize -1: 0] imm_gen_out;

    always@(*) begin
        case(part_of_inst [6:0])
            `ARITHMETIC : imm_gen_out = 32'd0;
            `ARITHMETIC_IMM : imm_gen_out = $signed(part_of_inst[31:20]);
            `LOAD : imm_gen_out = $signed(part_of_inst[31:20]);
            `JALR : imm_gen_out = $signed(part_of_inst[31:20]);
            `STORE : imm_gen_out = $signed({part_of_inst[31:25], part_of_inst[11:7]});
            `BRANCH : imm_gen_out = $signed({part_of_inst[31], part_of_inst[7], part_of_inst[30:25], part_of_inst[11:8], 1'b0});
            `JAL : imm_gen_out = $signed({part_of_inst[31], part_of_inst[19:12], part_of_inst[20], part_of_inst[30:21], 1'b0});
        endcase
    end

endmodule
```

Immediate Generator는 Single Cycle CPU와 동일한 module을 사용하였다. 주어진 opcode에 맞게 immediate field를 모아서 32bit immediate를 생성해 내보낸다.

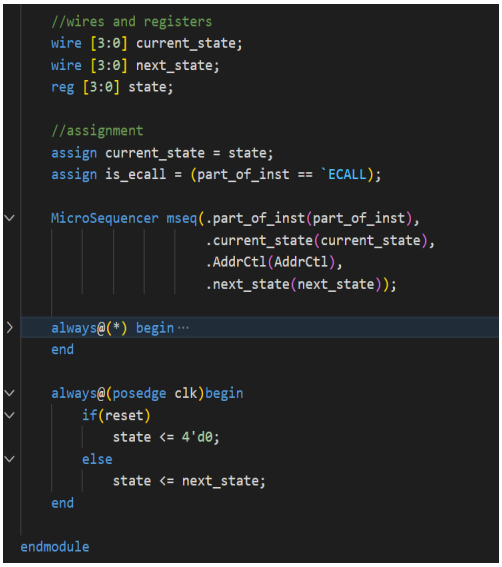
D. PC

```
module PC(
    reset,
    clk,
    next_pc,
    nextPCWrite,
    current_pc
    );

    input reset;
    input clk;
    input nextPCWrite;
    input [`WordSize -1 : 0] next_pc;
    output reg [`WordSize -1 : 0] current_pc;

    always@(posedge clk)begin
        if(reset)
            current_pc <= `WordSize'd0;
        else if(nextPCWrite)
            current_pc <= next_pc;
        else
            current_pc <= current_pc;
    end
endmodule
```

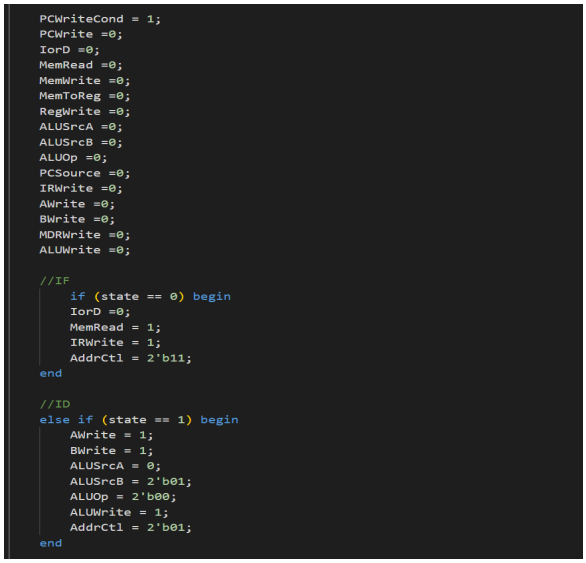
PC의 경우, clock cycle이 바뀔 때마다 새로운 PC로 갱신한다는 아이디어는 유사하지만, 이번에는 매 clock cycle마다 PC가 바뀌지는 않기 때문에, Signal을

고려하여, 바뀌어야만 할 상황에 PC가 갱신되도록 구현해주었다.

E. Control Unit



```
    //wires and registers
    wire [3:0] current_state;
    wire [3:0] next_state;
    reg [3:0] state;

    //assignment
    assign current_state = state;
    assign is_ecall = (part_of_inst == `ECALL);

    MicroSequencer mseq(.part_of_inst(part_of_inst),
                        .current_state(current_state),
                        .AddrCtl(AddrCtl),
                        .next_state(next_state));

    always@(*) begin ···
    end

    always@(posedge clk)begin
        if(reset)
            state <= 4'd0;
        else
            state <= next_state;
    end

endmodule
```



```
    PCWriteCond = 1;
    PCWrite =0;
    IorD =0;
    MemRead =0;
    MemWrite =0;
    MemToReg =0;
    RegWrite =0;
    ALUSrcA =0;
    ALUSrcB =0;
    ALUOp =0;
    PCSource =0;
    IRWrite =0;
    AWrite =0;
    BWrite =0;
    MDRWrite =0;
    ALUWrite =0;

    //IF
        if (state == 0) begin
        IorD =0;
        MemRead = 1;
        IRWrite = 1;
        AddrCtl = 2'b11;
    end

    //ID
    else if (state == 1) begin
        AWrite = 1;
        BWrite = 1;
        ALUSrcA = 0;
        ALUSrcB = 2'b01;
        ALUOp = 2'b00;
        ALUWrite = 1;
        AddrCtl = 2'b01;
    end
```

각 사진은 synchronous logic과 asynchronous logic의 일부이다. Synchronous logic의 경우는, 각 clock cycle마다 state를 업데이트 해준다.

반면, asynchronous logic은 reg로 인해 기억된 현재 state에 기반하여 control signal을 만들어준다. 더 자세한 state와 control signal value는 FSM transition diagram에 기술되어 있다.

F. MicroSequencer

```
module MicroSequencer(
    input [6:0] part_of_inst,
    input [3:0] current_state,
    input [1:0]AddrCtl,
    output reg [3:0] next_state
);

    //reg [3:0] state;
    always@(*) begin
        case(AddrCtl)
            `AddrCtl_Reset : next_state = 0;
            `AddrCtl_Next : next_state = current_state + 1;
            `AddrCtl_Branch1 : begin
                    case(part_of_inst)
                        `ARITHMETIC : next_state = 2;
                        `ARITHMETIC_IMM : next_state = 4;
                        `LOAD : next_state = 6;
                        `STORE : next_state = 6;
                        `BRANCH : next_state = 10;
                        `JAL : next_state = 12;
                        `JALR : next_state = 13;
                        `ECALL : next_state = 14;
                    endcase
            end
            `AddrCtl_Branch2 : begin
                case(part_of_inst)
                    `LOAD : next_state = 7;
                    `STORE : next_state = 9;
                endcase
            end
        endcase
    end

endmodule
```

Microsequencer의 경우에는, control unit에 저장된 state과, 각 state에서 정해진 다음 state로 가기 위한 signal, AddrCtl을 input으로 받아서, 다음 state를 계산하여 내보낸다. 이 때, opcode를 통해, state transition에 있어서 branch가 존재할 수 있게 해준다.

Resource reuse implementation

Multi cycle cpu는 single cycle cpu와 달리 resource, 즉, ALU, memory를 reuse한다. 이에 대한 implementation으로는, state 1에서 일어나게 되는데, pipeline에서는 ID Stage에 해당하는 단계에서는 ALU를 사용하지 않기 때문에, 이 시점에 PC+4를 ALU를 사용하여 계산하여 준다. 결국 이 reuse는 control signal을 어떻게 할당하느냐에 따라 다르기 때문에 Finite State Machine Transition Diagram에 기술된 control signal들이 위의 Control Unit의 state마다 잘 할당됨으로써, 구현 가능함을 알 수 있다.

Number of Cycles of Each Example:

1. Non-controlflow_mem.txt

   in ripes: 39 cycles

   my implementation: 157 cycles

2. Ifelse_mem.txt

   In ripes: 34 cycles

   my implementation: 139 cycles

3. Recursive_mem.txt

   In ripes: 896 cycles

   my implementation: 3686 cycles

4. Discussion

Difference between single cycle cpu and multi cycle cpu, and following advantage

Single cycle cpu는 모든 instruction이 하나의 cycle에 완료되어야 하기 때문에, 가능한 clock frequency를 모두 다 사용하지 못할 수 있다. 특히, 가장 시간을 많이 소요하는 load instruction에 clock period가 맞춰져 있기 때문에, 시간이 낭비된다. 또한, resource 측면에서 볼 경우, single cycle cpu의 경우는 adder 2개와 ,

ALU, 그리고 instruction memory와 data memory가 나눠져 있다. 하지만 이 functional unit들은 항상 사용되는 것이 아니라, 특정 instruction의 특정 부분에만 사용되기 때문에, 낭비된다고 할 수 있다.

반면, multi cycle cpu는 각 instruction에서 필요한 functional unit만을 사용하기 때문에, 각 instruction이 필요한 시간만큼만 사용된다. 이는 single cycle cpu에 비하면, 굉장히 낭비가 없는 시스템이라고 생각할 수 있다. 또한, instruction을 fetch하거나 decode하는 과정과 같이, ALU를 사용하지 않는 경우, ALU를 사용하여 PC+4를 미리 계산할 수 있기 때문에, resource 적인 측면에서도 single cycle cpu에 비해서 더 경제적이라고 할 수 있다.

5. Conclusion

결과적으로, multi cycle cpu는 single cycle cpu와 비슷하지만, state를 나누고 각 state에 따라 control signal을 할당하는 것으로, resource reuse를 통해, 더 경제적인 cpu를 설계할 수 있고, 동시에, clock frequency를 더 잘 활용할 수 있음을 알 수 있다.