2025 Spring

CSED311

컴퓨터구조

Lab 3 report

Team ID: 15

팀원 1: 20230345 이성재

팀원 2: 20230355 정지성

목차

I. 서론

Single-cycle CPU와 multi-cycle CPU 디자인의 큰 차이점들은 다음과 같다.

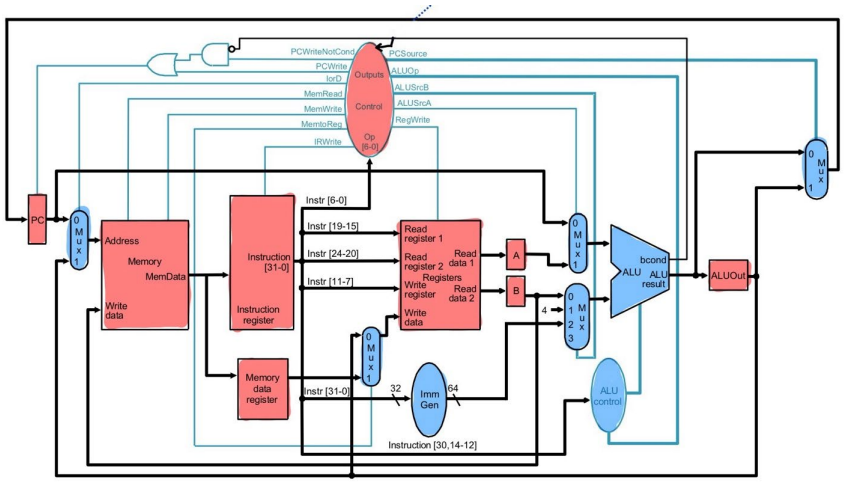
- Single-cycle CPU의 cycle는 가장 느린 instruction을 기준으로 맞추며, 한 instruction을 한 cycle에 처리한다. 그에 반해 multi-cycle CPU는 cycle의 시간을 감소하는 대신 하나의 instruction을 여러 cycle에 걸쳐 처리한다.
    - 이렇게 하면 빨리 끝나는 instruction에 대해서는 더 적은 cycle을, 늦게 끝나는 instruction에 대해서는 더 많은 cycle을 배정하여, 더 이상 소요 시간을 제일 느린 instruction에 맞추지 않아도 된다. 결론적으로 multi-cycle CPU의 instruction 당 평균 소요 시간이 더 작다.
- 한 instruction을 여러 cycle에 걸쳐 처리하므로, 하나의 모듈 (ALU, memory 등)을 한 instruction 안에서 여러 번 접근할 수 있다.
    - 즉 resource reuse가 가능해져, datapath의 길이를 줄일 수 있으며 공간 활용도도 높일 수 있다.
- Multi-cycle CPU는 하나의 instruction 안에서도 cycle에 따라 다른 행위를 해야 한다. 이 차이를 주기 위해서 control unit을 FSM을 이용하여 구현한다. 이는 control unit이 combinational logic으로만 구현되는 single-cycle CPU와 대조된다.

1번과 2번에서 비교한 장점들에 의해, multi-cycle CPU가 single-cycle CPU보다 더 좋다고 볼 수 있다. 이번 lab에서는 multi-cycle CPU를 구현하였다.

II. 디자인

A. CPU의 디자인

CPU 디자인은 Ch6, 14p에 나와있는 수업자료를 참고했다. 아래 디자인에서 언급할 만한 부분을 설명하겠다.



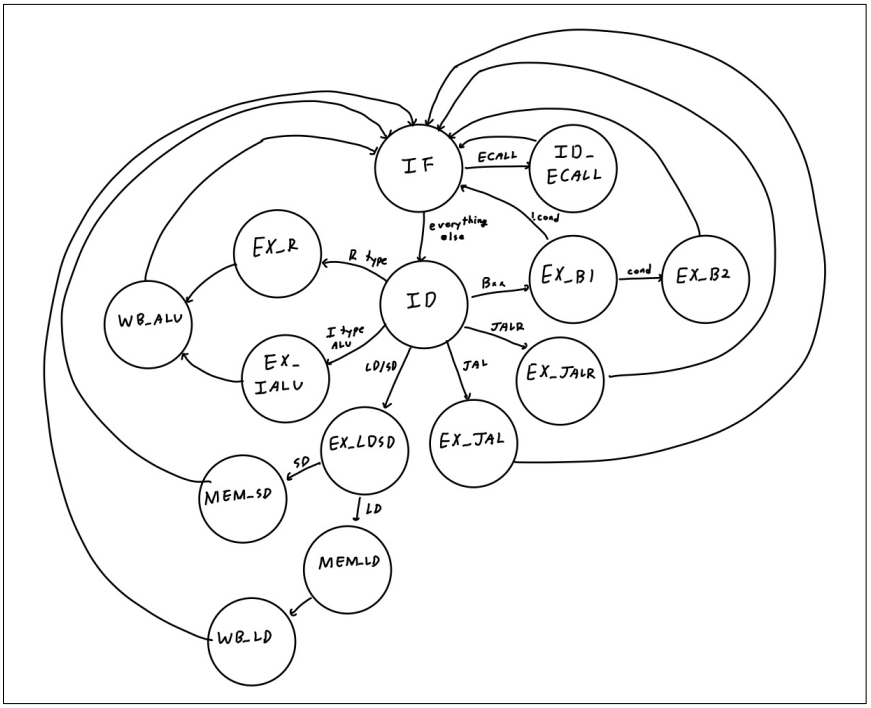
- 위 그림에서 clock synchronous한 module은 빨간색, asynchronous한 module은 파란색으로 표시하였다.
    - Memory와 register file은 데이터 저장 기능이 있으므로 synchronous해야 한다.
    - IR, MDR, A, B, ALUOut은 multi-cycle 구현을 위해 새로 도입한 latch이다.
    - IR레지스터는 control에 의해 enable이 조절되기 때문에, memory에서 출력되는 memdata값이 달라져도 IR값이 유지될 수 있다. 구현해야 하는 instruction들이 더 복잡해지면 MDR, A, B, ALUOut 등에도 control에 의한 enable을 주어야겠으나, 구현하는 instruction들이 그렇게 복잡하지 않아서 나머지 latch들은 "현재 사이클의 input 값을 다음 사이클로 넘겨주어 유지시켜주는" 형태 정도로 쓰이고 있다.
    - Control unit은 synchronous하도록 디자인되어 있다. 앞서 말했듯 multi-cycle CPU는 한 instruction이더라도 cycle에 따라 다른 행위를 해야 하므로 control unit은 clock synchronous해야 한다.
    - ALU control은 control unit이 주는 ALUOp 신호를 통해 cycle마다 다른 control을 출력하는 형태이며, ALU control 자체는 asynchronous하게 디자인하였다.
    - 나머지 모듈들은 그냥 주어진 input들을 combinational하게 처리하는 asynchronous 디자인으로 구현하였다.
- Resource reuse: Single-cycle CPU에 있던 세 개의 ALU를 하나로 합쳤으며, instruction memory와 data memory도 한 모듈로 합쳤다.
    - 이것이 가능한 이유는 한 instruction을 여러 cycle에 나누어 처리하기 때문이다.
    - 어느 cycle에 어떤 연산을 해야 하는지는 control unit이 결정해줄 것이다.
    - Single-cycle CPU 기준으로 어느 memory나 ALU를 '선택'하는지는 memory와 ALU 앞의 mux들을 통해 결정해 준다. (PC vs ALUOut), (PC vs A), (B vs 4 vs immediate)

B. Microcode controller의 디자인

Control unit은 자그마한 FSM으로 구현하였다. 구현의 편의를 위해 Moore machine으로 구현했다. Instruction들에 맞추어 state들을 잘 정의하고, 각 state마다 CPU가 해야 하는 계산이나 처리를 정의해 주면 된다. 여기서 주의해야 할 점은, 우리의 구현에서 resource reuse를 하고 있으므로 한 state에서 같은 resource를 두 번 이상 쓰지 않도록 주의해야 한다는 점이다.

모든 instruction들의 PVS(PC, memory, register) 업데이트는 instruction의 마지막 사이클에서 이루어질 수 있도록 구현했다. 첫 번째 그림은 control unit의 FSM state transition diagram, 두 번째 사진은 각 state별로 CPU에서 해야 하는 계산/처리들을 정리했다.

(is_halted의 구현은 control unit의 밖에서 이루어져, ECALL의 control unit 내부 구현은 단순하다.)



	R type	I type ALU	Load	Store	JAL	JALR	Bxx	ECALL
IF	IR ← MEM[PC]							
ID	A ← RF[rs1(IR)] B ← RF[rs2(IR)] ALUOut ← PC+4							A ← RF[rs1(IR)] B ← RF[rs2(IR)] PC ← PC+4
EX1	ALUOut ← A op B	ALUOut ← A op imm	ALUOut ← A + imm		RF[rd(IR)]←ALUOut PC ← PC + imm	RF[rd(IR)]←ALUOut PC ← A + imm	cond? (A, B) if(!cond) PC ← ALUOut	-
EX2							PC ← PC + imm(IR)	
MEM	-	-	MDR ← MEM[ALUOut]	MEM[ALUOut] ← B PC ← PC + 4	-	-	-	-
WB	RF[rd(ir)] ← ALUOut PC ← PC + 4		RF[rd(ir)] ← MDR PC ← PC + 4	-	-	-	-	-

III. 구현

A. 전반적 CPU의 구현

- cpu.v 의 코드는 위의 "II-A. CPU 의 디자인"을 그대로 코드로 옮긴 것이다. 추가적인 로직엔 다음과 같은 것들이 있다.
  - **is_halted** 판별 로직: memory 에서 꺼내온 **`ecall_reg_cond`** (x17 == 10?)와 control unit 에서 꺼내온 **`is_ecall`** 을 and 로 결합하여 판별한다.
  - PCWriteNotCond 와 PCWrite control signal 을 pc 의 enable 에 연결하는 간단한 combinational logic 은 pc.v 에서 구현했다.
  - Control unit 의 input 으로 bcond 를 추가했다.
- Immediate generator, ALU, memory, register 은 single-cycle 의 것과 거의 동일하게 구현했다.
- mux 들은 ternary operator 을 이용해 구현했다.
- Latch 들(A, B, ALUOut, IR, MDR)은 추가적 모듈 없이 cpu.v 에서 reg 로 정의하고 always @(posedge clk)으로 업데이트했다. 이 중 IR 은 IRWrite == 1 인 경우에만 업데이트했다.

B. Control Unit과 Microcode Controller

우선 control unit의 코드를 차례대로 살펴보겠다.

```
module ControlUnit(
    input reset,
    input clk,
    input [6:0] opcode,
    input alu_bcond,
    output PCWriteNotCond,
    output PCWrite,
    output IorD,
    output MemRead,
    output MemWrite,
    output MemtoReg,
    output IRWrite,
    output PCSource,
    output [1:0] ALUOp,
    output [1:0] ALUSrcB,
    output ALUSrcA,
    output RegWrite,
    output is_ecall
);
```

- 모듈의 Input과 output이다. 교안의 것에서 **is_ecall**만 하나 추가해 주었다.

```
// We will implement the "MicroSequencer: ver 1.0" in the lecture notes
// Because each stage has just one cycle, not many redundant cells
// Also, we will design Moore machine for simplicity

reg [3:0] current_state;
wire [3:0] next_state;

assign is_ecall = (opcode == 7'b1110011);
```

- 우선 state에 필요한 wire들을 정의하고, **is_ecall**을 빠르게 처리해 주었다. **is_halted**를 분별하는 로직은 메인 cpu 모듈에 구현되어 있어, control unit에서는 이 정도로만 해 주면 된다.

```
// combinationally get control values from state
StateToControl state_to_control(
    .current_state(current_state),          // input
    .PCWriteNotCond(PCWriteNotCond),        // output
    .PCWrite(PCWrite),                      // output
    .IorD(IorD),                            // output
    .MemRead(MemRead),                      // output
    .MemWrite(MemWrite),                    // output
    .MemtoReg(MemtoReg),                    // output
    .IRWrite(IRWrite),                      // output
    .PCSource(PCSource),                    // output
    .ALUOp(ALUOp),                          // output
    .ALUSrcB(ALUSrcB),                      // output
    .ALUSrcA(ALUSrcA),                      // output
    .RegWrite(RegWrite)                     // output
);

// combinationally get next state, based on current state and opcode
StateMachine state_machine(
    .opcode(opcode),                    // input
    .alu_bcond(alu_bcond),              // input
    .current_state(current_state),      // input
    .next_state(next_state)             // output
);

// sequentially update current state
always @(posedge clk) begin
    if (reset) begin
        current_state <= 4'b0000;
    end
    else begin
        current_state <= next_state;
    end
end
```

Control unit의 메인 기능이 담긴 코드이다. 이 모듈의 기능을 크게 세 가지로 나누면 다음과 같다.

- 현재 state에 따라 control 시그널 결정하기
- 현재 state와 opcode에 따라 다음 state 결정하기
- Latch를 이용해 state를 저장하고 업데이트하기

Latch를 제외한 기능들은 각각 모듈화해서 구현하였다. 이제 각각의 모듈 코드도 하나씩 살펴보겠다.

StateMachine 모듈은 microcode controller에 해당하는 부분이라고 볼 수 있다. 사실은 엄밀히 말하면 microcode라기보다는 그냥 FSM state transition module이다. 이렇게 구현한 이유는 **IV. 논의사항** 에서 다루도록 하겠다.

StateMachine 모듈의 코드는 다음과 같다. 디자인 파트에서 첨부한 state transition diagram을 그대로 코드로 옮긴 것임을 확인할 수 있다.

추가로, branch에 따라서 state이 달라져야 하는 경우가 있으므로 input에 **alu_bcond**이 포함된다.

```
`include "opcodes.v"
`include "states.v"

module StateMachine(
    input [6:0] opcode,
    input alu_bcond,
    input [3:0] current_state,
    output reg [3:0] next_state
);

    always @(*) begin
        case(current_state)
            `IF: if(opcode == `ECALL) begin
                next_state = `ID_ECALL;
            end
            else begin
                next_state = `ID;
            end

            `ID_ECALL: next_state = `IF;

            `ID: case(opcode)
                `ARITHMETIC: next_state = `EX_R;
                `ARITHMETIC_IMM: next_state = `EX_IALU;
                `LOAD: next_state = `EX_LDSD;
                `STORE: next_state = `EX_LDSD;
                `BRANCH: next_state = `EX_B1;
                `JAL: next_state = `EX_JAL;
                `JALR: next_state = `EX_JALR;
                default: next_state = `IF;
            endcase

            `EX_R: next_state = `WB_ALU;
            `EX_IALU: next_state = `WB_ALU;
            `WB_ALU: next_state = `IF;

            `EX_LDSD: if(opcode == `LOAD) begin
                next_state = `MEM_LD;
            end
            else begin
                next_state = `MEM_SD;
            end
            `MEM_LD: next_state = `WB_LD;
            `WB_LD: next_state = `IF;
            `MEM_SD: next_state = `IF;

            `EX_B1: if(alu_bcond == 0) begin
                next_state = `IF;
            end
            else begin
                next_state = `EX_B2;
            end
            `EX_B2: next_state = `IF;

            `EX_JAL: next_state = `IF;
            `EX_JALR: next_state = `IF;
            default: next_state = `IF;
        endcase
    end
endmodule
```

StateToControl 모듈은 microcode storage 에 해당하는 부분이라고 볼 수 있다. (여기서도 엄밀하게 말하면 microcode storage 는 아니지만 그와 동일한 기능을 하고 있다고 볼 수 있다.)

우리는 FSM 으로 구현했으므로 각 state 에 맞는 control output 을 세팅해주면 된다. 이것에 해당하는 코드를 모두 첨부하겠다.

```
`include "states.v"

module StateToControl(
    input [3:0] current_state,
    output reg PCWriteNotCond,
    output reg PCWrite,
    output reg IorD,
    output reg MemRead,
    output reg MemWrite,
    output reg MemtoReg,
    output reg IRWrite,
    output reg PCSource,
    output reg [1:0] ALUOp,
    output reg [1:0] ALUSrcB,
    output reg ALUSrcA,
    output reg RegWrite
    );

    // combinational logic to determine control based on state
    always @(*) begin
        PCWriteNotCond = 0;
        PCWrite = 0;
        IorD = 0;
        MemRead = 0;
        MemWrite = 0;
        MemtoReg = 0;
        IRWrite = 0;
        PCSource = 0;
        ALUOp = 2'b00;
        ALUSrcB = 2'b00;
        ALUSrcA = 0;
        RegWrite = 0;
```

```
        case(current_state)
            // IR <- MEM[PC]
            `IF: begin
                IorD = 0;
                MemRead = 1;
                IRWrite = 1;
            end

            // A <- RF[rs1(IR)]
            // B <- RF[rs2(IR)]
            // PC <- PC + 4
            `ID_ECALL: begin
                ALUSrcA = 0;
                ALUSrcB = 2'b01;
                ALUOp = 2'b00;
                PCSource = 0;
                PCWrite = 1;
            end

            // A <- RF[rs1(IR)]
            // B <- RF[rs2(IR)]
            // ALUOut <- PC + 4
            `ID: begin
                ALUSrcA = 0;
                ALUSrcB = 2'b01;
                ALUOp = 2'b00;
            end

            // ALUOut <- A op B
            `EX_R: begin
                ALUSrcA = 1;
                ALUSrcB = 2'b00;
                ALUOp = 2'b10;
            end
```

```
69          // ALUOut <- A op imm
70          `EX_IALU: begin
71              ALUSrcA = 1;
72              ALUSrcB = 2'b10;
73              ALUOp = 2'b10;
74          end
75
76          // RF[rd(IR)] <- ALUOut
77          // PC <- PC + 4
78          `WB_ALU: begin
79              MemtoReg = 0;
80              RegWrite = 1;
81
82              ALUSrcA = 0;
83              ALUSrcB = 2'b01;
84              ALUOp = 2'b00;
85              PCSource = 0;
86              PCWrite = 1;
87          end
88
89          // ALUOut <- A + imm
90          `EX_LDSD: begin
91              ALUSrcA = 1;
92              ALUSrcB = 2'b10;
93              ALUOp = 2'b00;
94          end
95
96          // MDR <- MEM[ALUOut]
97          `MEM_LD: begin
98              IorD = 1;
99              MemRead = 1;
100         end
```

```
102             // RF[rd(IR)] <- MDR
103             // PC <- PC + 4
104             `WB_LD: begin
105                 MemtoReg = 1;
106                 RegWrite = 1;
107
108                 ALUSrcA = 0;
109                 ALUSrcB = 2'b01;
110                 ALUOp = 2'b00;
111                 PCSource = 0;
112                 PCWrite = 1;
113             end
114
115             // MEM[ALUOut] <- B
116             // PC <- PC + 4
117             `MEM_SD: begin
118                 IorD = 1;
119                 MemWrite = 1;
120
121                 ALUSrcA = 0;
122                 ALUSrcB = 2'b01;
123                 ALUOp = 2'b00;
124                 PCSource = 0;
125                 PCWrite = 1;
126             end
```

```
128             // RF[rd(IR)] <- ALUOut
129             // PC <- PC + imm
130             `EX_JAL: begin
131                 MemtoReg = 0;
132                 RegWrite = 1;
133 
134                 ALUSrcA = 0;
135                 ALUSrcB = 2'b10;
136                 ALUOp = 2'b00;
137                 PCSource = 0;
138                 PCWrite = 1;
139             end
140 
141             // RF[rd(IR)] <- ALUOut
142             // PC <- A + imm
143             `EX_JALR: begin
144                 MemtoReg = 0;
145                 RegWrite = 1;
146 
147                 ALUSrcA = 1;
148                 ALUSrcB = 2'b10;
149                 ALUOp = 2'b00;
150                 PCSource = 0;
151                 PCWrite = 1;
152             end
153 
154             // cond? (A, B). if(!cond?) then PC <- ALUOut
155             `EX_B1: begin
156                 ALUSrcA = 1;
157                 ALUSrcB = 2'b00;
158                 ALUOp = 2'b01;
159 
160                 PCSource = 1;
161                 PCWriteNotCond = 1;
162             end
```

```
164             // PC <- PC + imm
165             `EX_B2: begin
166                 ALUSrcA = 0;
167                 ALUSrcB = 2'b10;
168                 ALUOp = 2'b00;
169 
170                 PCSource = 0;
171                 PCWrite = 1;
172             end
173 
174             default: ;
175         endcase
176     end
177 
178 endmodule
```

C. ALUControl

ALUControl은 자체적으로 synchronous한 모듈로 구현하지 않으며, control unit을 통해 받은 정보를 이용해서 ALU에게 적절한 신호를 준다. Single-cycle cpu에서 구현한 ALU에서, control unit으로 ALUOp만 추가적으로 받아주면 된다. ALUOp에 따라 ALU가 해야 하는 행위는 다음과 같다.

- ALUOp == 2'b00: 덧셈
- ALUOp == 2'b01: branch
- ALUOp == 2'b10: funct3에 써있는 ALU operation

이를 반영한 코드를 아래에 첨부한다.

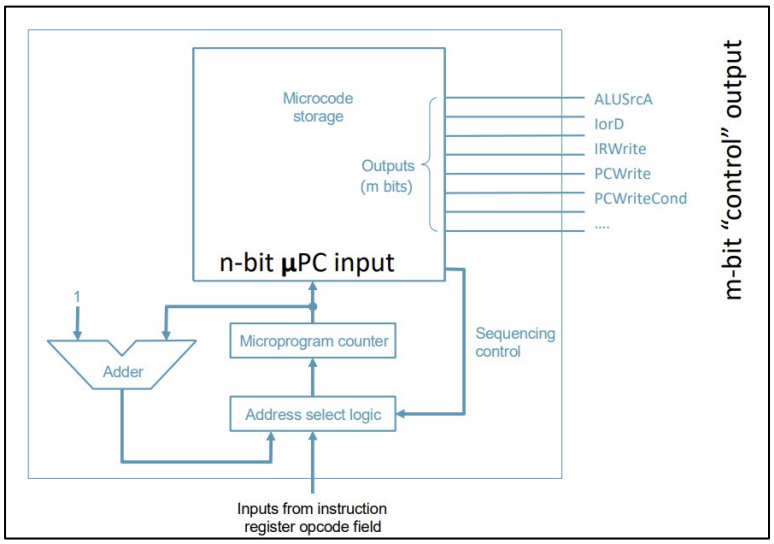
```
case (ALUOp)
    2'b00: ALUControl = `FUNC_ADD;
    2'b01: begin
        case(functs[2:0])
            `FUNCT3_BEQ: ALUControl = `FUNC_BEQ;
            `FUNCT3_BNE: ALUControl = `FUNC_BNE;
            `FUNCT3_BLT: ALUControl = `FUNC_BLT;
            `FUNCT3_BGE: ALUControl = `FUNC_BGE;
            default: ALUControl = 4'b0000;
        endcase
    end
```

```
2'b10: begin
    case(functs[2:0])
        `FUNCT3_ADD: begin
            if (opcode == `ARITHMETIC_IMM || functs[3] == 0)
                ALUControl = `FUNC_ADD;
            else
                ALUControl = `FUNC_SUB;
        end
        `FUNCT3_SLL: ALUControl = `FUNC_LLS;
        `FUNCT3_SRL: begin
            if (functs[3]) // funct7 bit
                ALUControl = `FUNC_ARS;
            else
                ALUControl = `FUNC_LRS;
        end
        `FUNCT3_XOR: ALUControl = `FUNC_XOR;
        `FUNCT3_OR:  ALUControl = `FUNC_OR;
        `FUNCT3_AND: ALUControl = `FUNC_AND;
        default: ;
    endcase
end
```

ALU 모듈은 single-cycle CPU의 구현과 동일하기 때문에 다루지 않겠다.

IV. 논의 사항

앞서 말했듯 우리의 구현은 엄밀한 의미의 microcode 대신 그냥 FSM transition으로 control unit을 구현하였다. 이 부분에서는 1) FSM적인 구현과 microcode적인 구현의 차이 및 장단점, 2) FSM으로 구현한 이유에 대해서 서술하고자 한다.



1) FSM vs microcode

Microcode 는 CPU 내부에서 명령어 해석 및 제어 신호 생성을 담당하는 하위 레벨 코드이다. 위의 그림처럼, opcode 를 받으면 내부적 로직을 이용해 microprogram counter 을 조종하고, 이 microprogram counter 이 가리키는 microcode storage 에서 control output 을 결정시킨다. 이렇게 하면 일단적으로 FSM 으로 구현했을 때보다 공간을 더 아낄 수 있고, 새로운 instruction 을 implement 하고 싶을 때의 확장성이 더 뛰어나다.

2) FSM 으로 구현한 이유

Microcoding: Ver 1.0

~2^(opcode field's bit width)

State label	Control flow	Conditional targets					
		R/I-type	LD	SD	Bxx	JALR	JAL
IF_1	next	-	-	-	-	-	-
IF_2	next	-	-	-	-	-	-
IF_3	next	-	-	-	-	-	-
IF_4	go to	ID	ID	ID	ID	ID	EX_1
ID	next	-	-	-	-	-	
EX_1	next	-	-	-	-	-	-
EX_2	go to	WB	MEM_1	MEM_1	IF_1	WB	WB
MEM_1	next		-	-			
MEM_2	next		-	-			
MEM_3	next		-	-			
MEM_4	go to		WB	IF_1			
WB	go to	IF_1	IF_1			IF_1	IF_1
CPI		8	12	11	7	8	7

~2^(state register's bit width)

Each cell corresponds to a row in the ROM. Note the redundant or unused cells.

우리의 구현에서 state 가 그렇게 많지 않고, state transition 이 opcode 에 dependent 한 경우가 그렇지 않은 경우보다 많다. 따라서 대부분의 state transition 이 microcode 에서는 branch 형태일 것이다. 구체적으로, 위의 그림에서 " - "에 해당하는 cell 이 우리 implementation 에서는 거의 없다. 따라서 그냥 FSM 으로 구현하는 것이 더 자연스럽다고 생각했다.

V. 결론

5개의 example에 대한 testbench를 실행해본 결과는 아래와 같다. 순서대로 basic, non-controlflow, loop, ifelse, recursive example에 대해 testing을 해 본 결과이다.

```
### SIMULATING ###
TEST END
SIM TIME : 234
TOTAL CYCLE : 116 (Answer : 116)
FINAL REGISTER OUTPUT
 0 00000000
 1 00000000
 2 00002ffc
 3 00000000
 4 00000000
 5 00000000
 6 00000000
 7 00000000
 8 00000000
 9 00000000
10 00000013
11 00000003
12 ffffffd7
13 00000037
14 00000013
15 00000026
16 0000001e
17 0000000a
18 00000000
19 00000000
20 00000000
21 00000000
22 00000000
23 00000000
24 00000000
25 00000000
26 00000000
27 00000000
28 00000000
29 00000000
30 00000000
31 00000000
Correct output : 32/32
```

```
### SIMULATING ###
TEST END
SIM TIME : 314
TOTAL CYCLE : 156 (Answer : 157)
FINAL REGISTER OUTPUT
 0 00000000
 1 00000000
 2 00002ffc
 3 00000000
 4 00000000
 5 00000000
 6 00000000
 7 00000000
 8 00000000
 9 00000000
10 0000000a
11 0000003f
12 fffffff1
13 0000002f
14 0000000e
15 00000021
16 0000000a
17 0000000a
18 00000000
19 00000000
20 00000000
21 00000000
22 00000000
23 00000000
24 00000000
25 00000000
26 00000000
27 00000000
28 00000000
29 00000000
30 00000000
31 00000000
Correct output : 32/32
```

```
### SIMULATING ###
TEST END
SIM TIME : 1956
TOTAL CYCLE : 977 (Answer : 977)
FINAL REGISTER OUTPUT
 0 00000000
 1 00000000
 2 00002ffc
 3 00000000
 4 00000000
 5 00000000
 6 00000000
 7 00000000
 8 00000000
 9 00000000
10 00000000
11 00000000
12 00000000
13 00000000
14 0000000a
15 00000009
16 0000005a
17 0000000a
18 00000000
19 00000000
20 00000000
21 00000000
22 00000000
23 00000000
24 00000000
25 00000000
26 00000000
27 00000000
28 00000000
29 00000000
30 00000000
31 00000000
Correct output : 32/32
```

```
### SIMULATING ###
TEST END
SIM TIME : 280
TOTAL CYCLE : 139 (Answer : 139)
FINAL REGISTER OUTPUT
 0 00000000
 1 00000000
 2 00002ffc
 3 00000000
 4 00000000
 5 00000000
 6 00000000
 7 00000000
 8 00000000
 9 00000000
10 00000000
11 00000000
12 00000000
13 00000000
14 0000000a
15 00000028
16 00000000
17 0000000a
18 00000000
19 00000000
20 00000000
21 00000000
22 00000000
23 00000000
24 00000000
25 00000000
26 00000000
27 00000000
28 00000000
29 00000000
30 00000000
31 00000000
Correct output : 32/32
```

```
### SIMULATING ###
TEST END
SIM TIME : 7374
TOTAL CYCLE : 3686 (Answer : 3686)
FINAL REGISTER OUTPUT
 0 00000000
 1 00000000
 2 00002ffc
 3 00000000
 4 00000000
 5 00000000
 6 00000000
 7 00000000
 8 00000000
 9 00000000
10 0000000d
11 00000000
12 00000000
13 00000000
14 00000001
15 0000000d
16 00000015
17 0000000a
18 00000000
19 00000000
20 00000000
21 00000022
22 00000000
23 00000037
24 00000059
25 00000000
26 00000000
27 00000000
28 00000000
29 00000000
30 00000000
31 00000000
Correct output : 32/32
```

구체적으로, basic example에서 116 cycles, loop example에서 977 cycles을 사용하였다. Final register outputs도 answer의 것과 같은 것으로 보아 우리의 multi-cycle CPU 구현은 정확하다는 결론을 내릴 수 있다.

처음에는 multi-cycle CPU가 조금 낯설고 어렵게 느껴졌었다. 특히 control flow를 state들을 이용해서 구현하는 부분이 헷갈렸는데, 직접 구현해 보면서 익숙해지게 되어 좋았다. 이해가 더 잘 된 느낌이다.