

컴퓨터 구조 (CSED311)

<Lab3 - Multi-cycle CPU>

Team ID : 33

Student 1 : 곽민성 (Gwak Minseong, 20230840)

Student 2 : 김재환 (Kim Jaehwan, 20230499)

1. Introduction

이번 랩의 목표는 Multi-cycle CPU를 Verilog를 통해 구현하는 것이다. Multi-cycle CPU는 Single Cycle CPU의, 서로 다른 시간이 걸리는 두 명령어를 항상 한 사이클에 실행함으로써 생기는 단점 해결하기 위하여, 명령어마다 사이클을 세분화하여 하나의 명령이 정확히 필요한 시간 동안만 동작하도록 한다. 이 랩을 성공적으로 진행함으로써 Multi-cycle CPU의 세부 구조와 동작을 정확하게 이해할 수 있었다.

2. Design

우리는 아래 강의 노트의 CPU 구조를 사용하여 Multi-cycle CPU를 구현하였다.

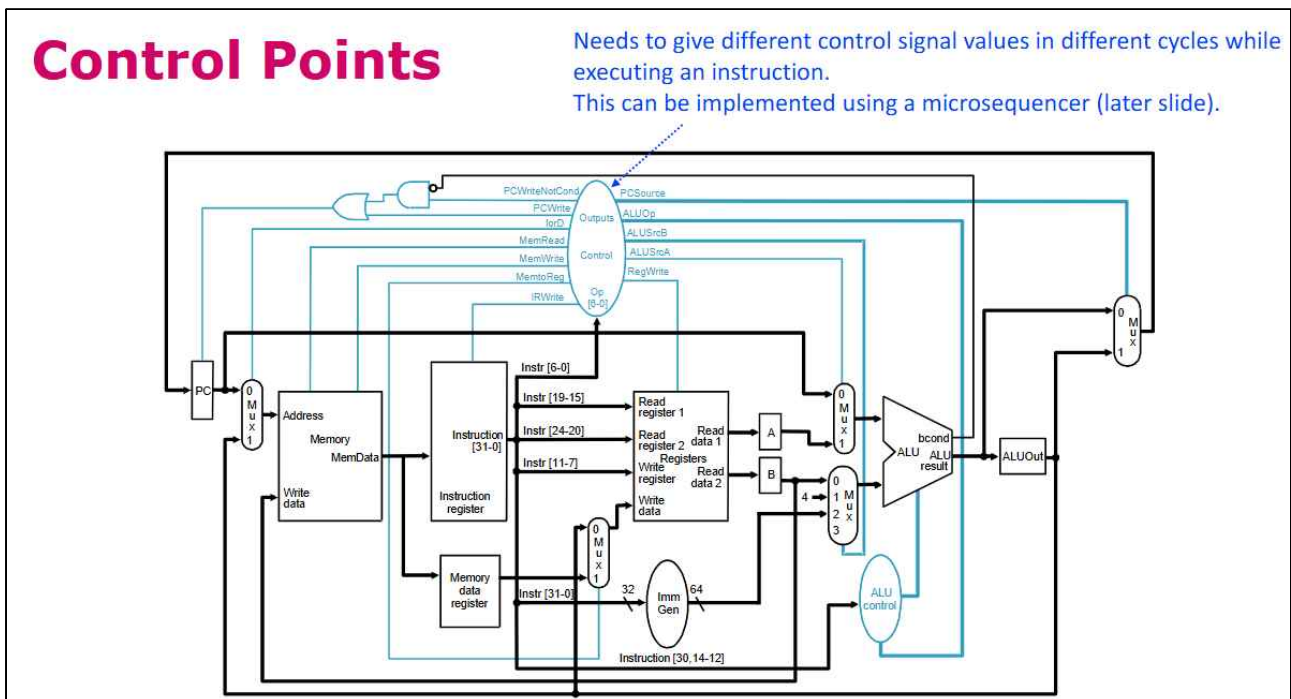


그림 1 Structure of Multi-cycle CPU

Combined RT Sequencing

	R-Type	LD	SD	Bxx	JAL	JALR
common steps	start: IR ← MEM[PC]					
	A ← RF[rs1(IR)]					
	B ← RF[rs2(IR)]					
	ALUOut ← PC + 4					
opcode dependent steps	ALUOut ← A+B	ALUOut ← A+imm(IR)	ALUOut ← A+imm(IR)	cond? (A, B) if (!cond) { PC ← ALUOut goto start (if cond) goto next	RF[rd(IR)] ← ALUOut PC ← PC+imm(IR) goto start	RF[rd(IR)] ← ALUOut PC ← A+imm(IR) goto start
	RF[rd(IR)] ← ALUOut PC ← PC+4 goto start	MDR ← MEM[ALUOut] goto start	MEM[ALUOut] ← B PC ← PC+4 goto start	PC ← PC+imm(IR) goto start		
		RF[rd(IR)] ← MDR PC ← PC+4 goto start				

RTs in each state corresponds to some setting of the control signals

그림 2 RT Sequencing Table

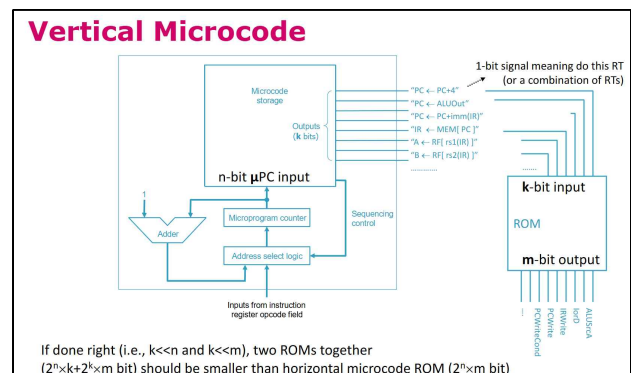


그림 3 Vertical Microcode

이는 ALU와 같이 중복된 모듈을 사이클(시간) 별로 분리하여 하나의 모듈만으로 처리할 수 있도록 개선된 구조이다. 이를 통해 구성된 모듈은 다음과 같다.

모듈 이름	설명	Synchronous / Asynchronous
ALU	실행해야 하는 Instruction(alu_op)와 두 수를 입력으로 받아 산술 연산의 결과를 반환한다. 만약 Branch Instruction의 경우에 Branch Condition의 결과 또한 반환한다.	Asynchronous
ALUControlUnit	Instruction을 입력으로 받아, ALU가 수행해야 하는 동작을 정의하고, 반환한다. 만약 현재 사이클에 $PC \leftarrow PC + 4$ 와 같은 다른 microinstruction을 실행해야 할 경우에는, 그 신호를 Control Unit으로부터 입력으로 받아 더하기 연산을 반환한다.	Asynchronous
ControlUnit	각 Instruction과 현재 Control Unit의 내부 state(IF, ID 등등)에 따라, 다른 모듈에서 필요한 Control 신호들을 생성한다.	Synchronous
ImmediateGenerator	Instruction을 읽어서, 그 Instruction에 있는 Immediate value를 파싱해서 반환한다.	Asynchronous
Memory	Instruction 혹은 Data의 주소를 입력으로 받아, 메모리상의 해당 주소에 위치한 값을 반환하거나(MemRead), 그 주소에 들어온 값을 쓴다(MemWrite).	Asynchronous (Read), Synchronous (Write)
Mux2 / Mux4	두 개의 입력과 1-bit의 sel 신호(Mux2), 4개의 입력과 2-bit의 sel 신호(Mux4)를 받아, sel 신호가 지시하는 입력을 그대로 반환한다.	Asynchronous
PC	next_pc를 입력으로 받아서, ControlUnit에서 나온 신호와 Bcond를 조합해서 만약 current_pc를 업데이트해야 하는 상황이라면 current_pc를 다음 값으로 업데이트한다.	Synchronous
RegisterFile	2개의 source 레지스터와 destination 레지스터의 번호, 입력값을 받아서 2개의 레지스터의 값을 반환하고, destination 레지스터에 값을 쓴다 (RegWrite).	Asynchronous (Read), Synchronous (Write)
ROM	ControlUnit의 내부 모듈로, Vertical Microcode를 구현하기 위해 사용되었다. 입력으로 여러 Microinstruction을 받아서 실제로 그 Microinstruction을 실행하기 위해 주어져야 하는 Control 신호를 생성한다.	Asynchronous

우리는 ControlUnit을 아래와 같이 구성하였다.

Control Unit의 내부 상태 : 초기에 state는 1로 초기화되며 각 posedge 클럭마다 1씩 증가한다. state가 1일 때 instruction fetch가 일어나며 받아온 명령어에 따라, 어느 클럭에서 state를 1로 초기화한다. 만약 state가 1일때 R-type 명령어를 받아왔다면, state가 4일때 다음 state를 1로 초기화하는 식으로 작동한다. 이 과정은 *RT-sequencing table*(그림 2)을 이용하였으며, 테이블에 나오지 않는 I-type arithmetic-imm은 4사이클로 작동하도록 설계하였다.

state: 1 (IF: R-type)	→ 2 (R)	→ 3 (R)	→ 4 (R)	→ 1 (IF: next)
state: 1 (IF: LD)	→ 2 (LD)	→ 3 (LD)	→ 4 (LD)	→ 5 (LD)→1 (IF: next)
state: 1 (IF: SD)	→ 2 (SD)	→ 3 (SD)	→ 4 (SD)	→ 1 (IF: next)
state: 1 (IF: Bxx)	→ 2 (Bxx)	→ 3 (Bxx (cond))	→ 4 (Bxx)	→ 1 (IF: next)
(state: 1 (IF: Bxx) → 2 (Bxx) → 3 (Bxx(not cond)) → 1 (IF: next))				
(BXX 명령어는 Branch가 taken 되는지, 되지 않는지에 따라 사이클 수가 다르다)				
state: 1 (IF: JAL)	→ 2 (JAL)	→ 3 (JAL)		→ 1 (IF: next)
state: 1 (IF: JALR)	→ 2 (JALR)	→ 3 (JALR)		→ 1 (IF: next)

또 그림 2를 보면 각 단계에서 어떤 microinstruction을 실행해야 하는지 확인할 수 있다. 이 부분을 그림 3에서 보이는 ROM 모듈을 따로 제작하여 microinstruction을 실행하기 위해 필요한 Control 신호를 출력한다. 그 구조는 다음과 같다.

Control Unit
<pre>`include "opcodes" module ControlUnit(input reset, input clk, input [6:0] Instr, input ALUBcond, output PCWriteNotCond, output PCWrite, output IorD, output MemRead, output MemWrite, output MemtoReg, output IRWrite, output PCSource, output ALUOp, output [1:0] ALUSrcB, output ALUSrcA, output RegWrite, output is_ecall); reg [2:0] state; initial begin state = 1; end</pre>

```

always @(posedge clk) begin
    if(reset) begin
        state <= 1;
    end
    else begin
        case(Instr)
            `JAL, `JALR: begin
                if(state == 3) state <= 1;
                else state <= state + 1;
            end
            `ARITHMETIC, `ARITHMETIC_IMM, `STORE: begin
                if(state == 4) state <= 1;
                else state <= state + 1;
            end
            `LOAD: begin
                if(state == 5) state <= 1;
                else state <= state + 1;
            end
            `BRANCH: begin
                if(state == 4) state <= 1;
                else if(state == 3 && !ALUBcond) state <= 1;
                else state <= state + 1;
            end
            `ECALL: begin
                if(state == 3) state <= 1;
                else state <= state + 1;
            end
            default: state <= state+1;
        endcase
    end
end

wire to_IR_from_MEM_PC    = (state == 1);           // 1
wire to_A_from_RF_RS1     = (state == 2);           // 2
wire to_B_from_RF_RS2     = (state == 2);           // 2
wire to_ALUOut_from_PCp4  = (state == 2);           // 2
wire to_ALUOut_from_ApB    = (state == 3 && Instr == `ARITHMETIC); // R3
wire to_RF_rd_from_ALUOut = ((state == 4 && Instr == `ARITHMETIC)
    || (state == 3 && Instr == `JAL)
    || (state == 3 && Instr == `JALR)
    || (state == 4 && Instr == `ARITHMETIC_IMM)); // R4, JAL3, JALR3, I4
wire to_PC_from_PCp4      = ((state == 4 && Instr == `ARITHMETIC)
    || (state == 5 && Instr == `LOAD)
    || (state == 4 && Instr == `STORE)
    || (state == 4 && Instr == `ARITHMETIC_IMM)
    || (state == 3 && Instr == `ECALL)); // R4, LD5, SD4, I4
wire to_ALUOut_from_Apimm = ((state == 3 && Instr == `LOAD)
    || (state == 3 && Instr == `STORE)
    || (state == 3 && Instr == `ARITHMETIC_IMM)); // LD3, SD3, I3
wire to_MDR_from_MEM_ALUOut = (state == 4 && Instr == `LOAD); // LD4
wire to_RF_rd_from_MDR     = (state == 5 && Instr == `LOAD); // LD5
wire to_MEM_ALUOut_from_B  = (state == 4 && Instr == `STORE); // SD4
wire to_PC_from_ALUOut     = (state == 3 && Instr == `BRANCH); // B3
wire to_PC_from_PCpimm     = ((state == 4 && Instr == `BRANCH)

```

```

                                || (state == 3 && Instr == `JAL));          // B4, JAL3
wire to_PC_from_Apimm    = (state == 3 && Instr == `JALR);              // JALR3

assign is_ecall = (Instr == `ECALL);
assign ALUOp = !(state == 3 && (Instr == `ARITHMETIC || Instr == `ARITHMETIC_IMM || Instr ==
`BRANCH));

ROM ROM(
    .to_IR_from_MEM_PC(to_IR_from_MEM_PC),
    .to_A_from_RF_RS1(to_A_from_RF_RS1),
    .to_B_from_RF_RS2(to_B_from_RF_RS2),
    .to_ALUOut_from_PCp4(to_ALUOut_from_PCp4),
    .to_ALUOut_from_ApB(to_ALUOut_from_ApB),
    .to_RF_rd_from_ALUOut(to_RF_rd_from_ALUOut),
    .to_PC_from_PCp4(to_PC_from_PCp4),
    .to_ALUOut_from_Apimm(to_ALUOut_from_Apimm),
    .to_MDR_from_MEM_ALUOut(to_MDR_from_MEM_ALUOut),
    .to_RF_rd_from_MDR(to_RF_rd_from_MDR),
    .to_MEM_ALUOut_from_B(to_MEM_ALUOut_from_B),
    .to_PC_from_ALUOut(to_PC_from_ALUOut),
    .to_PC_from_PCpimm(to_PC_from_PCpimm),
    .to_PC_from_Apimm(to_PC_from_Apimm),
    .PCWriteNotCond(PCWriteNotCond),
    .PCWrite(PCWrite),
    .IorD(IorD),
    .MemRead(MemRead),
    .MemWrite(MemWrite),
    .MemtoReg(MemtoReg),
    .IRWrite(IRWrite),
    .PCSource(PCSource),
    .ALUSrcB(ALUSrcB),
    .ALUSrcA(ALUSrcA),
    .RegWrite(RegWrite)
);

endmodule

```

ROM

```

`include "opcodes"
module ROM(
    input to_IR_from_MEM_PC,      // 1
    input to_A_from_RF_RS1,      // 2
    input to_B_from_RF_RS2,      // 2
    input to_ALUOut_from_PCp4,    // 2
    input to_ALUOut_from_ApB,     // R3
    input to_RF_rd_from_ALUOut,   // R4, JAL3, JALR3
    input to_PC_from_PCp4,       // R4, LD5, SD4
    input to_ALUOut_from_Apimm,   // LD3, SD3
    input to_MDR_from_MEM_ALUOut, // LD4
    input to_RF_rd_from_MDR,     // LD5
    input to_MEM_ALUOut_from_B,   // SD4
    input to_PC_from_ALUOut,     // B3
    input to_PC_from_PCpimm,     // B4, JAL3
    input to_PC_from_Apimm,      // JALR3

```

```

output PCWriteNotCond,
output PCWrite,
output IorD,
output MemRead,
output MemWrite,
output MemtoReg,
output IRWrite,
output PCSource,
output [1:0] ALUSrcB,
output ALUSrcA,
output RegWrite
);

assign PCWriteNotCond = to_PC_from_ALUOut;
assign PCWrite       = to_PC_from_PCp4 || to_PC_from_PCpimm || to_PC_from_Apimm;
assign IorD          = !to_IR_from_MEM_PC;
assign MemRead       = to_IR_from_MEM_PC || to_MDR_from_MEM_ALUOut;
assign MemWrite      = to_MEM_ALUOut_from_B;
assign MemtoReg      = to_RF_rd_from_MDR;
assign IRWrite       = to_IR_from_MEM_PC;
assign PCSource      = to_PC_from_ALUOut;
assign ALUSrcB       = (to_ALUOut_from_ApB || to_PC_from_ALUOut ? 0 :
(to_ALUOut_from_PCp4 || to_PC_from_PCp4) ? 1 :
(to_ALUOut_from_Apimm || to_PC_from_PCpimm || to_PC_from_Apimm) ? 2 : 3);
assign ALUSrcA       = !(to_ALUOut_from_PCp4 || to_PC_from_PCp4 || to_PC_from_PCpimm);
assign RegWrite      = to_RF_rd_from_ALUOut || to_RF_rd_from_MDR;

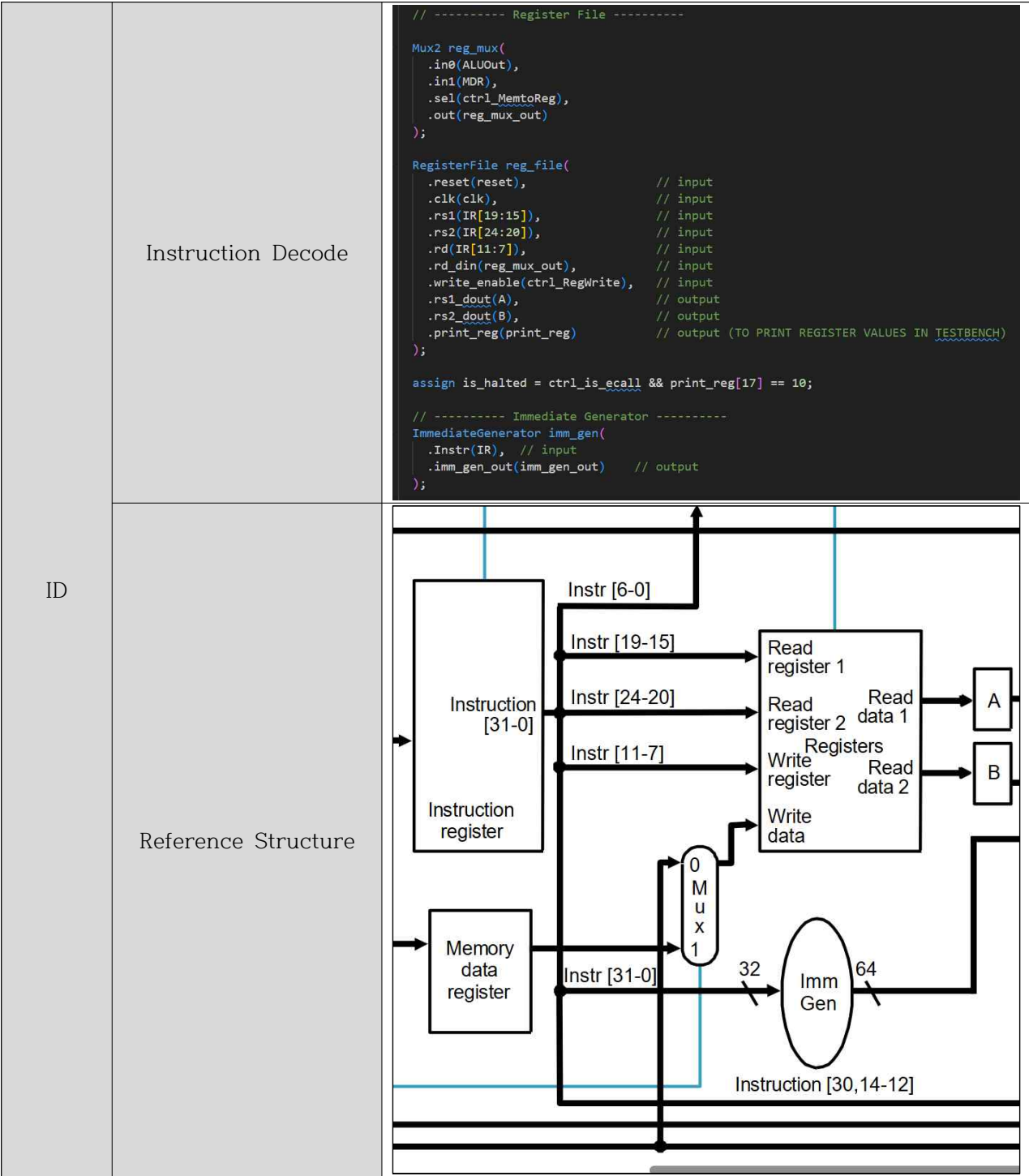
endmodule

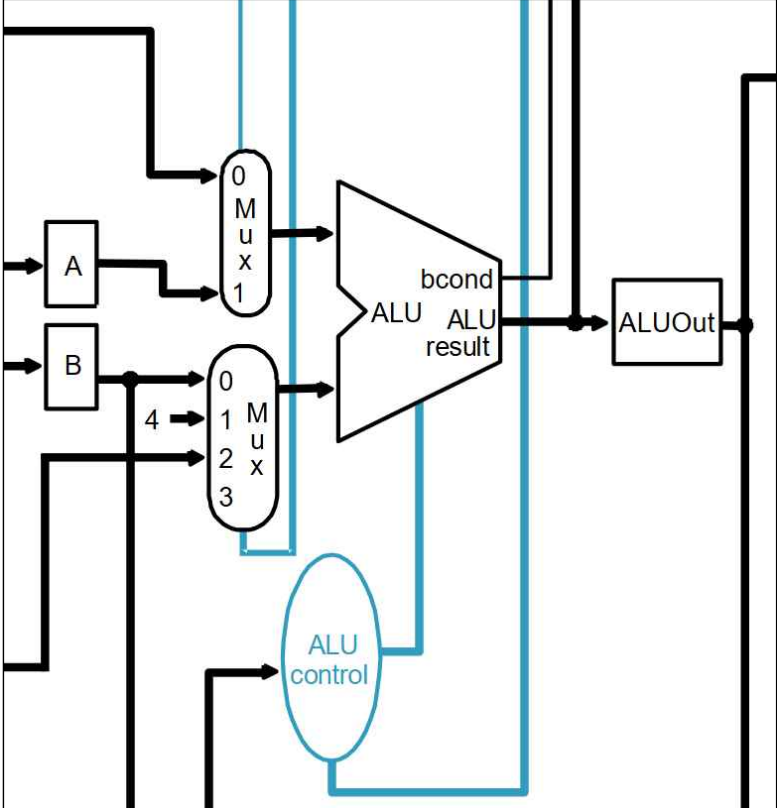
```

3. Implementation

우리는 위에서 구현된 모듈을 cpu에서 다음과 같이 wire를 이어서 구현하였다.

IF	Instruction Fetch	<pre> // ----- Update program counter ----- // PC must be updated on the rising edge (positive edge) of the clock. PC pc(.reset(reset), // input (Use reset to initialize PC. Initial value must be 0) .clk(clk), // input .PCWrite(PCWrite), // input .next_pc(next_pc), // input .current_pc(current_pc) // output); Mux2 addr_mux(.in0(current_pc), .in1(ALUOut), .sel(ctrl_IorD), .out(addr_mux_out)); // ----- Memory ----- Memory memory(.reset(reset), // input .clk(clk), // input .addr(addr_mux_out), // input .din(B), // input .mem_read(ctrl_MemRead), // input .mem_write(ctrl_MemWrite), // input .dout(mem_out) // output); always @(posedge clk) begin if(reset) begin IR <= 0; MDR <= 0; end else begin if(ctrl_IRWrite) begin IR <= mem_out; MDR <= MDR; end else begin MDR <= mem_out; IR <= IR; end end end end </pre>
	Reference Structure	



EX	ALU/Excute	<pre> // ----- ALU Control Unit ----- ALUControlUnit alu_ctrl_unit(.Instr30(IR[30]), // input .funct3(IR[14:12]), // input .opcode(IR[6:0]), // input .NoInst(ctrl1_ALUOp), // input .alu_op(alu_op) // output); // ----- ALU ----- Mux2 ALUMux1(.in0(current_pc), .in1(A), .sel(ctrl1_ALUSrcA), .out(alu_in_1)); Mux4 ALUMux2(.in0(B), .in1(4), .in2(imm_gen_out), .in3(0), .sel(ctrl1_ALUSrcB), .out(alu_in_2)); ALU alu(.alu_op(alu_op), // input .alu_in_1(alu_in_1), // input .alu_in_2(alu_in_2), // input .alu_result(alu_result), // output .alu_bcond(ALUBcond) // output); always @(posedge clk) begin ALUOut <= alu_result; end Mux2 next_pc_mux(.in0(alu_result), .in1(ALUOut), .sel(ctrl1_PCSource), .out(next_pc)); </pre>
	Reference Structure	
MEM	Data Memory Access	IF 단계의 Memory로 들어가는 Control 신호(IorD)가 바뀌면서 Memory 모듈에서 Data Memory를 가져오게 된다.
WB	Write-Back	위 동작에서, memory 모듈과 RegisterFile 모듈로 다시 들어가는 wire를 통해 Write-Back 된다.

4. Discussion

A. Multi-cycle cpu와 Single-cycle cpu의 차이 및 왜 Multi-cycle cpu가 좋은가?

Single-cycle cpu는 한 사이클에 모든 명령을 수행해야 하며, 사이클의 길이가 가장 느린 명령어(LD)에 맞추어져 있기 때문에 느리다. 대신, Multi-cycle cpu는 명령어마다 다른 사이클 수를 사용하여 각 명령어마다 필요한 만큼의 시간만 사용하기 때문에 더 효율적이다.

B. 명령어 별 사이클 수 (ecall 포함)

강의 노트에서 제공된 RT(Register Transfer) Sequencing table을 사용하여 사이클 수를 결정하였다. R-type, Store, Branch는 4 사이클, Load는 5 사이클, JAL과 JALR은 3사이클을 사용한다. 테이블에서 나와있지 않은 I-type (ARITHMETIC-IMM)은 4사이클, ECALL (nop)은 3 사이클, ECALL (exit)은 1사이클을 사용하도록 구현하였다.

C. ALUControlUnit 역할?

우리가 제작한 ALUControlUnit은 두가지 역할을 갖는다

- ControlUnit으로부터 제어신호를 받아, 필요하다면 더하기 연산을 수행.
- 제어신호가 꺼져있다면, Instruction이 지정하는 연산을 수행.

D. Control unit 구조 (vertical microcode)

vertical microcode 구조를 차용하였다. ControlUnit 모듈은 명령어를 입력받아 상태에 맞는 각 제어신호를 출력해야한다. 이를 state 레지스터를 이용해 상태를 구성하였다.

- state와 현재 명령어에 따라 RT-sequencing 표에 나와있는 신호를 제어한다.
- RT-sequencing 신호를 이용하여 실제 제어신호를 만들어 출력한다.

E. basic rides, loop rides에서 사이클 수 비교

실제 asm 파일이 실행되었을 때 사이클 수와 우리가 제작한 CPU의 사이클 수, 정답으로 제시한 사이클 수를 맞추어 보기 위해 다음과 같이 예측해 보았고, 모두 일치함을 확인할 수 있었다.

basic	리셋 1 사이클 + (4사이클 명령어 * 21개, 5사이클 명령어 6개) + ECALL(exit) 1 사이클 = 116 사이클
loop	리셋 1사이클 첫 번째 반복문 전 (main, *L2+0, *L2+4): 4사이클 명령어 7개, 5사이클 명령어 2개, 3사이클 명령어 1개(JAL) $4*7 + 5*2 + 3 = 41$ 사이클 첫 번째 반복문(10회) (*L2+8(BLT), L3, *L2+0, *L2+4) Bxx(cond) 4사이클, 5사이클 명령어 5개, 4사이클 명령어 4개 $(4 + 5*5 + 4*4) * 10 = 450$ 사이클 첫 번째 반복문과 두 번째 반복문 중간 (*L2+8(BLT) ~ *L2+16, *L4+0, *L4+4) Bxx(not cond) 3사이클, 4사이클 명령어 2개, 5사이클 명령어 1개, JAL 3사이클 $3 + 4*2 + 5 + 3 = 19$ 사이클

두 번째 반복문(10회) (*L4+8, L5, *L4+0, *L4+4)

Bxx(cond) 4사이클, 5사이클 명령어 4개, 4사이클 명령어 5개

$(4 + 5*4 + 4*5) * 10 = 440$ 사이클

두 번째 반복문 후

Bxx(not cond) 3사이클, 5사이클 2개, 4사이클 3개, ECALL(exit) 1사이클

$3 + 5*2 + 4*3 + 1 = 26$ 사이클

총합 : $1 + 41 + 450 + 19 + 440 + 26 = 977$ 사이클

위 두 테스트 케이스에 대해서 계산한 사이클은 직접 구현한 CPU, 제공된 정답과 모두 일치했다.

5. Conclusion

위에서 설명한 구조대로 CPU를 구현하여 제공된 모든 테스트 벤치에 대해서 정답과 같은 시뮬레이션 결과를 얻었다. 사이클 수 및 모든 레지스터 결과가 주어진 정답과 동일함을 확인할 수 있다.

	Test Bench
basic	### 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 ffffffff d7 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

ifelse

```
### 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
```

loop

```
### 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
```

non-controlflow

```
### SIMULATING ###  
TEST END  
SIM TIME : 316  
TOTAL CYCLE : 157 (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 ffffffff1  
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
```

recursive

```
### 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
```

위와 같이 정상 동작하는 모습을 확인할 수 있다.

6. References

[1] Lecture notes.