1. Introduction

이번 과제의 목표는 RISC-V ISA의 Multi cycle CPU를 구현하는 것이다. 지난 과제에서 디자인한 Single cycle CPU와 어떻게 작동 원리가 달라지는지 알아볼 것이다.

A. Difference between single-cycle CPU and multi-cycle CPU

Single cycle CPU 와 Multi cycle CPU의 가장 중요한 차이점은 Instruction 하나를 실행하는 데 걸리는 클럭 사이클 횟수이다. Single cycle CPU는 instruction을 Fetch 하고(IF stage) Decode 하여(ID stage) 실행해야 하는 동작을 정하고, ALU로 연산을 수행(EX stage)한 뒤 MEM read/right 후(MEM stage) register에 Wright Back까지(WB stage) 일련의 과정을 전부 한 사이클 내에 처리하는 반면, Multi cycle CPU는 나눌 수 있는 부분을 쪼개서 독립적인 클럭 사이클로 나누어, 한 Instruction을 여러 사이클에 걸쳐 실행한다.

B. Why multi-cycle CPU is better?

이런 Multi cycle CPU는 Single cycle CPU에 비해 크게 두 가지 이점을 지니는데, 퍼포먼스 향상과, 리소스 절약이 그것이다.

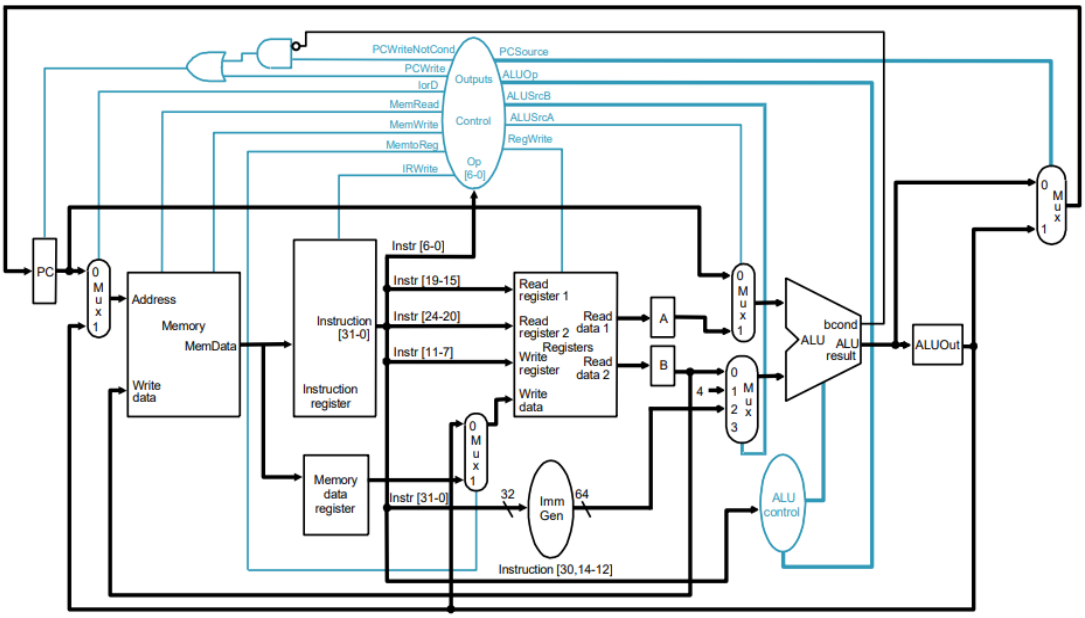
1. 퍼포먼스 향상: Single cycle CPU는 클럭 주기를 가장 오래 걸리는 instruction에 맞추어 놓았기에, 그보다 짧게 걸리는 instruction을 수행할 때 그만큼 시간이 낭비가 된다. 반면, Multi cycle CPU는 각 단계별로 사이클로 나누어 놓았기에, 짧게 걸리는 instruction의 경우 불필요한 stage를 생략할 수 있어서 그만큼 시간이 절약되어 최종 성능이 향상되게 된다.

2. 리소스 절약: Single cycle CPU의 경우, instruction이 전부 한 사이클 내에 진행되므로, main ALU는 계속 operand를 수행하느라 다른 작업을 할 수가 없다. 따라서 PC + 4 작업을 해주는 adder 하나 더, 마찬가지로 PC + Immediate 작업을 하는 adder 하나 더 추가하여 총 adder 가 3개 필요 해진다. 그러나, Multi cycle CPU는 main ALU가 operand를 실제로 연산해야 하는 건 EX stage 한 싸이클 뿐이므로, ALU 작업이 겹치지 않게 잘 설계한다면 instruction 내의 나머지 사이클 동안 PC 연산도 동일 ALU가 전부 처리하게 할 수 있다. 따라서 추가적인 adder 가 불필요하므로 리소스가 절약된다.

2. Design

A. Multi-cycle CPU design and implementation & state design

Multi cycle CPU의 대략적인 Instruction 수행 과정은 다음과 같다.



Multi cycle CPU의 구조도

IF stage cycle 동안에 MEM[PC]로부터 Instruction을 읽어와서 Instruction Register에 저장한다.

ID stage cycle 동안에는 RF로부터 rs1, rs2 값을 읽어와서 A, B 라는 레지스터에 저장한다.

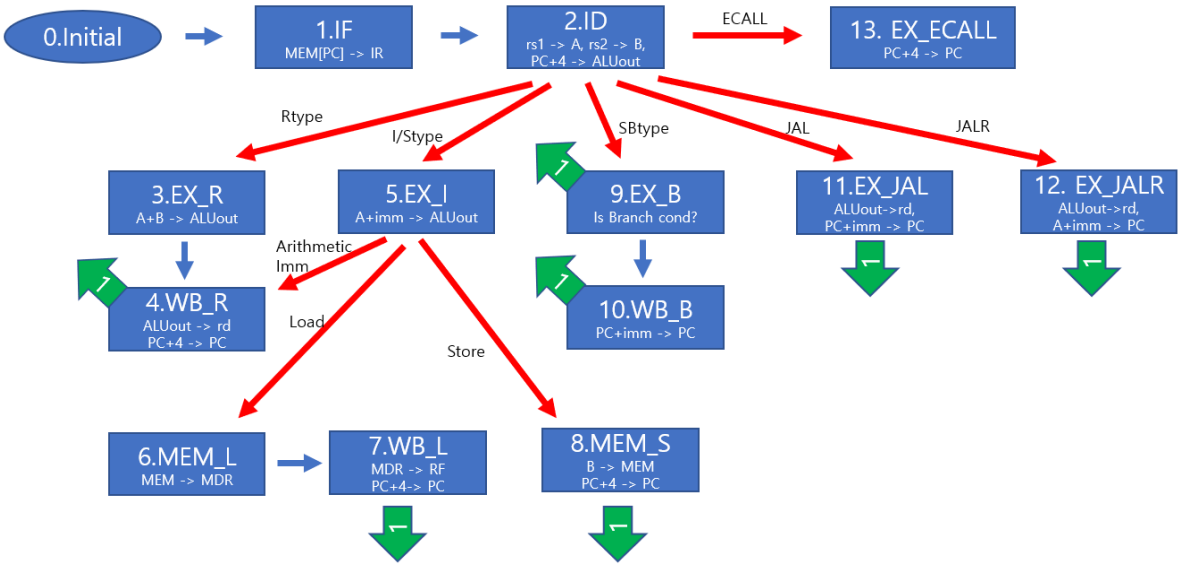
EX stage cycle 동안에는 ALU가 Instruction operand를 수행하는데, rs1/rs2를 쓸지, Immediate를 쓸지는 MUX 가 결정한다. ALU연산 결과는 ALU out 라는 레지스터에 저장되는데 Jump Instruction의 경우 이 단계에서 Register 값이 아닌, 바로 PC값을 연산하기도 한다.

MEM stage cycle 동안에는 필요에 따라 MEM[ALUOut]로부터 데이터를 읽어오거나 저장한다.

WB stage cycle 동안에는 ALUOut값 등을 필요에 따라 Write back 하고, 동시에 ALU 양쪽 source의 MUX 가 PC 와 4를 선택하여, PC+4값을 연산하여 바로 PC에 저장하게 된다.

모든 Instruction 이 위의 과정을 거치는 것은 아니며, 필요에 따라 각 stage 세부 내용이 바뀌거나 불필요한 stage가 빠지기도 한다.

따라서 CPU디자인을 위해 각 Instruction별로 어떤 stage를 진행하며, 각 stage에서 CPU가 어떤 동작을 해야 하는지 간략한 설명과 함께, state flow chart 로 표현해보았다:



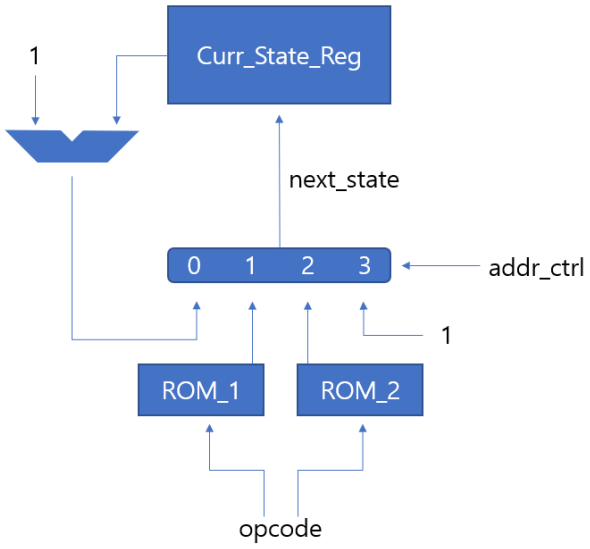
우리 조가 구현한 multi cycle CPU의 state flow chart

위 flow chart에서 파란색 화살표들은 무조건 해당하는 다음 state 로 가는(sequential) flow이다. 이렇게 무조건 연쇄적으로 실행되는 state넘버링을 순차적으로 배치하여, +1 adder 만으로 옮겨갈 수 있도록 설계했다.

빨간색 화살표는 opcode에 따라 달라질 수 있는 state flow이고, 따라서 opcode에 따라 다음 state를 출력해주는 ROM 회로에 의해 다음 state가결정된다. 이런 식의 ROM에 의한 state 변경 중 2state에서 다른 state로 변하는 것을 '1단계 점프', 5state에서 다른 state로 변하는 것을 '2단계 점프'로 지칭하도록 하겠다.

마지막으로 초록색 화살표는 현재 instruction 이 끝나서 다음 IF 스테이지 (1번 state)로 돌아간다는 것을 의미하며, next state를 업데이트 하는 MUX에 input 중 하나를 1로 넣고, 이것을 선택하게 만드는 식으로 구현한다.

B. Microcode controller state design



위 state design logic을 Microcode controller로 구현하기 위해서는 왼쪽 그림과 같은 구조가 필요하다.

위의 state flow chart logic에서 파란색 화살표(sequential)을 수행해야 하면, MUX에서 +1 adder로부터 오는 input을 고르고,

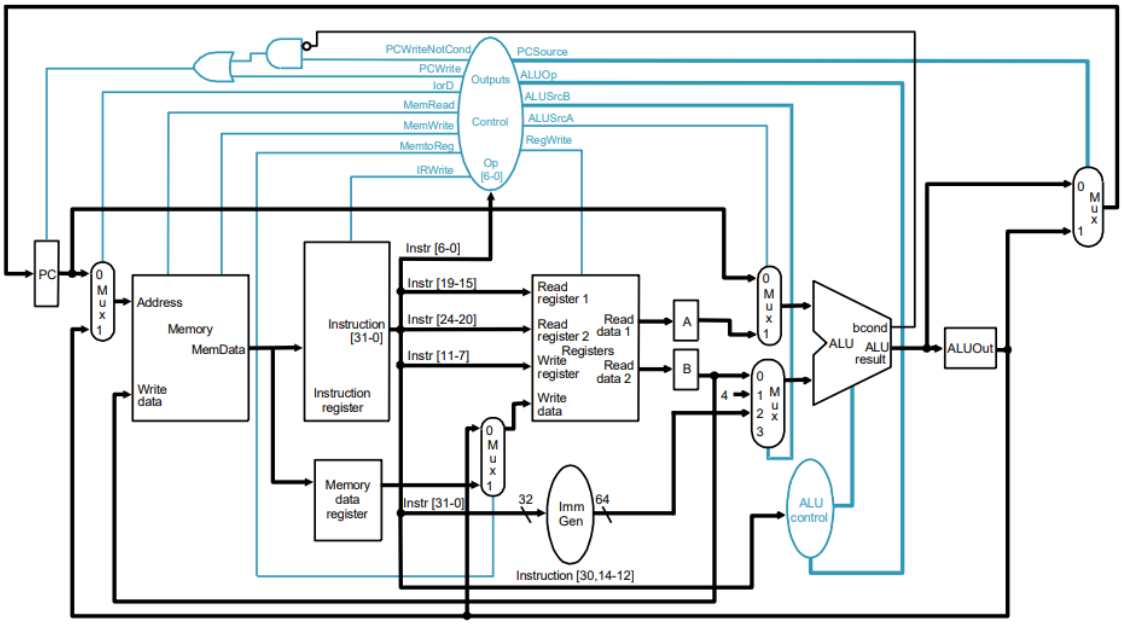
빨간색 화살표 중 1단계 점프는 ROM_1으로부터의 input을, 2단계 점프는 ROM_2으로부터의 input 2을 MUX에서 선택하며,

마지막으로 초록색 화살표(return to 1)은 MUX에서 3번 input을 골라서 다음 state를 1로 정하는 식으로 Micro program counter에 다음 state를 설정한다.

이와 같은 Logic으로 다음 state가 결정된다면, 현재 state를 기반으로 각종 control signal 과 다음 state를 결정하는 MUX 신호(addr_ctrl)을 generate 하는 Microcode memory를 만들 수 있다.

이 Microcode memory와 Micro program counter를 결합하여, CPU제어를 담당하는 Micro program controller unit이 완성된다.

C. Whether each module is clock synchronous or asynchronous.



Program Counter, Register File, Memory 와 같은 Programmer Visible State에 값을 쓰는 행위(write)는 clock synchronous 하게 일어난다. 또한, IR, MDR, A, B, ALUOut 같이 다음 cycle에서 쓰일 데이터를 임시로 저장하고 있는 레지스터들도 값이 업데이트 되는 건 clock synchronous, 레지스터로부터 값을 읽는 행위는 asynchronous 하게 일어난다. 또한 Control Unit의 Micro Program Counter 안에 들어있는 현재 state를 저장하는 curr_state_reg(current state register) 역시 값을 쓰는 행위는 clock synchronous, 값을 읽는 행위는 asynchronous 하게 일어난다. 이외에 다른 Unit들(e.g. ALU, Immediate Generator, ALU control) 등은 clock asynchronous하게 write와 read한다.

3.Implementation

A. Micro program controller unit

우선 각 CPU 구성 모듈로 control signal을 보내는 중추 역할을 하는 controller unit 구현부터 설명을 하겠다. 이번 CPU의 control unit은 이전 single cycle 때와 달리, 같은 Instruction 수행 중에도, 현재 state에 따라 다른 신호를 내보내야 한다.

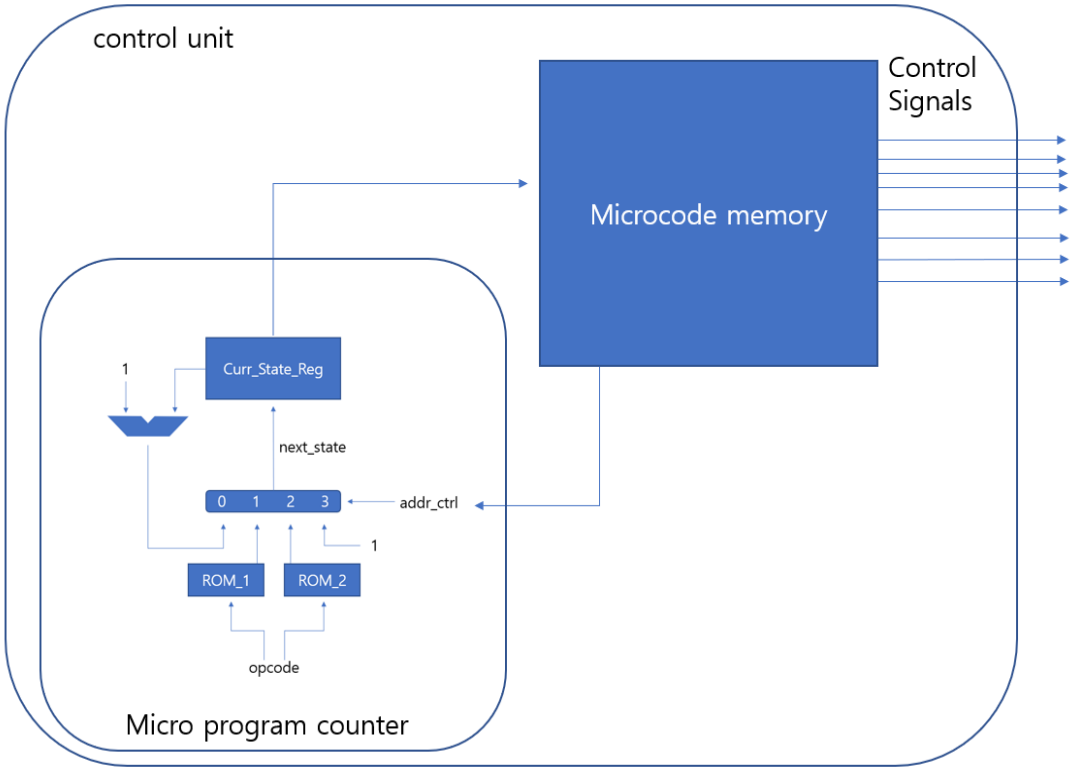
따라서 앞서 디자인 단계에서 설명한 Micro program controller unit 형태로 구현하였다.

Micro program controller는 두 파트로 구성된다. 현재 state를 저장 및 업데이트 해서 전달해주는 Micro program counter 파트와, Micro program counter로부터 전달받은 현재 state값을 바탕으로 각종 시그널과 다음 state를 위한 Address select logic을 도출하는 Microcode memory 파트이다.

1) Micro program counter

```
module MicroCounter(
   input reset,
   input clk,
   input [6:0] part_of_inst,
   input [1:0] addr_ctrl,
   output [3:0] curr_state
   );

    reg [3:0] internal_curr_state;
```

internal_curr_state 레지스터가 현재 CPU의 state를 표시하는 레지스터이다.

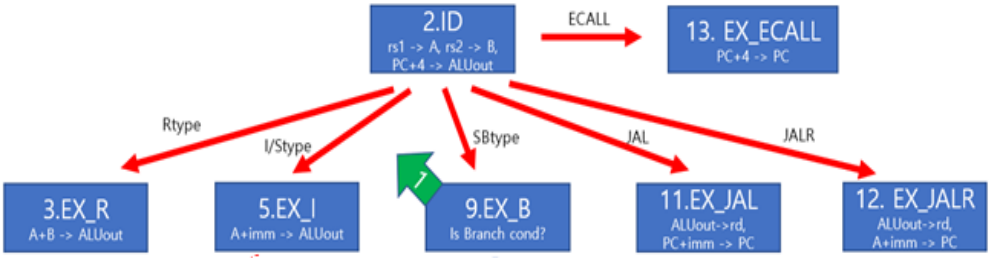


2개의 파트로 구성된 Micro program controller의 개념도

```
wire [3:0] state_to_adder;
wire [3:0] adder_to_mux;
wire [3:0] jump_state_1;
wire [3:0] jump_state_2;
wire [3:0] next_state;
wire [3:0] always1;

assign always1 = 1;
assign curr_state = internal_curr_state;
assign state_to_adder = internal_curr_state;

Adder adder1(.A(always1), .B(state_to_adder), .result(adder_to_mux));

ROM1 rom1(.opcode(part_of_inst), .next_state(jump_state_1));
ROM2 rom2(.opcode(part_of_inst), .next_state(jump_state_2));

Mux4_1_4bits mc_mux(.mux_in1(adder_to_mux), .mux_in2(jump_state_1), .mux_in3(jump_st

always @(posedge clk) begin
    if(reset) begin
       internal_curr_state <= 4'b0000;
       instruction_counter <= 0;
    end
    else begin
        internal_curr_state <= next_state;

        if (next_state == 1) instruction_counter <= instruction_counter + 1;
    end
end
```

```
.mux_in3(jump_state_2), .mux_in4(always1), .select_signal(addr_ctrl), .mux_out(next_state));
```

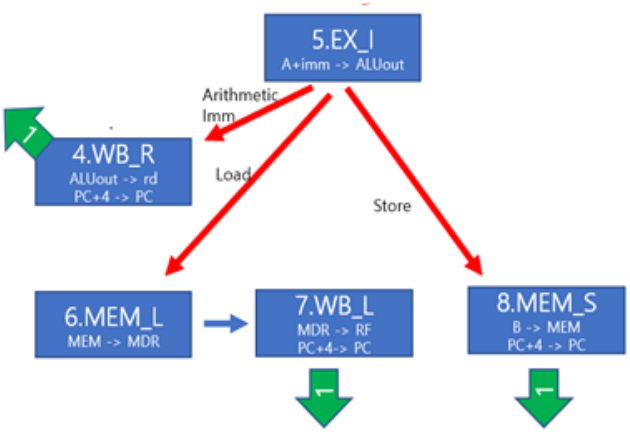
앞서 Design 문단에서 설명한 것과 동일하게 internal_curr_state를 다음 state로 update할 때, opcode를 바탕으로 다음 state를 산출하는 ROM 두 개, 그리고 +1 adder, 그리고 1 값을 각각 4:1 MUX에 넣고, address select logic 신호에 따라 그중 하나를 골라서 update 하도록 디자인했다. adder는 단순히 curr_state에 1을 더하며, Design 문단의 flow chart에서 파란색 화살표 역할을 하게 된다. MUX가 always1 input을 선택한다면, 현재 instruction 이 끝나서 다음 IF 스테이지 (1번 state)로 돌아간다는 것을 의미하며, Design 문단 flow chart의 초록색 화살표에 대응한다



ROM1이 담당하는 '1단계 점프' flow chart

```
module ROM1(
    input [6:0] opcode,
    output reg [3:0] next_state
    );

    always @(*) begin
        case(opcode)
            `ARITHMETIC: begin next_state = 3; end

            `ARITHMETIC_IMM: begin next_state = 5; end

            `LOAD: begin next_state = 5; end

            `STORE: begin next_state = 5; end

            `BRANCH: begin next_state = 9; end

            `JAL: begin next_state = 11; end

            `JALR: begin next_state = 12; end

            `ECALL: begin next_state = 13; end
        endcase
    end
endmodule
```

실제 ROM1의 코드, 위 flow chart 와 동일한 역할을 하는 걸 볼 수 있다



ROM2가 담당하는 '2단계 점프' flow chart

```
module ROM2(
    input [6:0] opcode,
    output reg [3:0] next_state
);

    always @(*) begin
        case(opcode)
            `ARITHMETIC_IMM: begin next_state = 4; end

            `LOAD: begin next_state = 6; end

            `STORE: begin next_state = 8; end
        endcase
    end

endmodule
```

실제 ROM2의 코드, 위 flow chart 와 동일한 역할을 하는 걸 볼 수 있다

2) Microcode memory

```
module MicroCode(
   input alu_bcond,
   input [31:0] x17,
   input [3:0] curr_state,
   output pc_write_not_cond, //output
   output pc_write,     //output
   output i_or_d, //output
   output mem_read, //output
   output mem_write, //output
   output mem_to_reg, //output
   output ir_write, //output
   output pc_source, //output
   output alu_src_a, //output
   output [1:0] alu_src_b, //output
   output reg_write, //output
   output ALUOp, //output
   output halt, //output
   output [1:0] addr_ctrl
   );
```

Microcode memory는 Micro program counter로부터 전달받은 curr_state 값을 바탕으로 각종 control signal을 결정해야 한다.

```
//signal
   assign pc_write_not_cond = (curr_state == 9 && alu_bcond == 0) ? 1 : 0;
   assign pc_write = (curr_state == 11 || curr_state == 12 || curr_state == 10 || curr_state == 8 || curr_state == 7 || curr_state == 4 || curr_state == 13) ? 1 : 0;
   assign i_or_d = (curr_state == 6 || curr_state == 8) ? 1 : 0;
   assign mem_read = (curr_state == 1 || curr_state == 6) ? 1 : 0;
   assign mem_write = (curr_state == 8) ? 1 : 0;
   assign mem_to_reg  = (curr_state == 7) ? 1 : 0;
   assign ir_write = (curr_state == 1) ? 1 : 0;
   assign pc_source = (curr_state == 9) ? 1 : 0;
   assign alu_src_a = (curr_state == 3 || curr_state == 5 || curr_state == 9 || curr_state == 12) ? 1 : 0; //1: 3,4,5,7
   assign alu_src_b = (curr_state == 3 || curr_state == 9) ? 2'b00 : ((curr_state == 2 || curr_state == 8 || curr_state == 7 ||  curr_state == 4 || curr_state == 13) ?
   assign reg_write = (curr_state ==4 || curr_state == 7 || curr_state == 11 || curr_state == 12) ? 1 : 0; //12,11,6,7
   assign ALUOp = (curr_state == 2 || curr_state == 11 || curr_state == 12 || curr_state == 10 || curr_state == 8 || curr_state == 7 || curr_state == 4) ? 1 : 0;// 1 :
   assign halt = (curr_state == 13 && x17 == 10) ? 1 : 0; //output
```

```
) ? 2'b01 : ((curr_state == 5 || curr_state == 11 || curr_state == 12 || curr_state == 10) ? 2'b10 : 2'b11));//0: 3,5   1: 2,10,11,12    2: 4,6,7,8

1 : always add (pc+4, pc+imm) 0: depends on opcode  //1,12,11,10,6,7,8
```

curr_state 값을 바탕으로 각종 control signal을 결정하는 logic 코드 부분(소스코드 참고)

또한 Micro program counter의 Address select logic도 결정해 주어야 한다:

```
//addr ctrl logic
   assign addr_ctrl = (curr_state == 0 || curr_state == 1 || curr_state == 3 || curr_state == 6 || (curr_state == 9 && alu_bcond)) ? 2'b00 :
      (curr_state == 2 ? 2'b01 :
      (curr_state == 4 || curr_state == 7 || curr_state == 8 || (curr_state == 9 && !alu_bcond) || curr_state == 10 || curr_state == 11 || curr_s
      2'b10));
```

```
rr_state == 12 || curr_state == 13 ? 2'b11 :
```

curr_state 값을 바탕으로 각종 Address control을 결정하는 logic 코드 부분

특이사항으로, 9번 state일 때는 branch condition이 충족되면 10번 state로 이동하여 정상적으로 branch 되도록 addr_ctrl신호가 0 (이때의 Address select logic은 +1 adder 가 된다) 이 되고, branch condition이 아니라면, 그대로 1번 state 로 돌아가도록 addr_ctrl신호가 3이 된다. 또한, state 13는 원래는 단순 CP+4 만하는 state이나, x17 레지스터 값이 10 이라면, halt 신호를 내보내어 CPU를 멈추게 된다.

B. Multi-cycle CPU Implementation

```
Mux IorD_mux(
  .mux_in1(pc_to_mux),
  .mux_in2(ALUOut),
  .select_signal(i_or_d),
  .mux_out(i_or_d_mux_to_mem)
);

  // ---------- Memory ----------
Memory memory(
  .reset(reset),        // input
  .clk(clk),            // input
  .addr(i_or_d_mux_to_mem),        // input
  .din(B),              // input
  .mem_read(mem_read),      // input
  .mem_write(mem_write),    // input
  .dout(mem_data)           // output
);
```

Multi-cycle(멀티 싸이클) CPU에서는 single-cycle(싱글 싸이클) CPU와 다르게 Memory resource reuse를 하여 instruction memory와 data memory를 하나로 합쳐 하나의 memory unit을 사용한다. 따라서 메모리의 address가 Instruction에 접근하는 건지, data에 접근하는 건지를 선별하는 mux가 필요하다. 이러한 기능을 하는 mux를 mux의 select signal의 이름을 따서 IorD mux라고 한다.

```
always @ (posedge clk) begin
  if (ir_write) begin
    IR <= mem_data;
  end
end

always @ (posedge clk) begin
  MDR <= mem_data;
end
```

멀티 싸이클 CPU에서는 싱글 싸이클 CPU와 달리 resource conflict의 상황을 고려해야 한다. Resource reuse로 인해 IM과 DM을 합쳐 하나의 메모리로 만들었다. 이렇다 보니 메모리에서 읽은 값이 instruction인 지, data인 지를 구분하지 않으면 conflict가 발생한다. 따라서 IR 레지스터와 MDR 레지스터를 각각 만들고 IR 레지스터에는 IRWrite이라는 control 신호를 인가하여 해당 신호가 켜진 상황에서만 IR 레지스터를 update한다. 이런 방식으로 두 개의 레지스터에 메모리에서 읽은 instruction과 data를 각각 저장(keep)할 수 있게 된다. 또한 MDR 레지스터는 load instruction에서 메모리에 접근할 주소가 들어있는 ALUOut에 PC + 4가 쓰여지는 conflict을 방지한다.

```
Mux mem_to_reg_mux(
  .mux_in1(ALUOut),
  .mux_in2(MDR),
  .select_signal(mem_to_reg),
  .mux_out(write_data_to_reg_file)
);
```

single-cycle CPU에서도 유사한 mux가 존재했다. Write register에 쓰는 값을 ALUOut에 들어있는 값으로 할 지, MDR에 들어있는 값으로 할 지를 선택한다.

```
always @ (posedge clk) begin
   A <= rs1_to_A;
   B <= rs2_to_B;

   if (IR[11:7] == 17 && reg_write == 1)
      x17 <= write_data_to_reg_file;
end
```

A, B 레지스터를 만들어 레지스터 파일에서 읽은 값(RF[rs1], RF[rs2])을 저장(keep)한다. 또한 x17 레지스터를 만들어서 rf[17]의 값에 10이 들어있는 지를 체크해야지만 halt 신호를 보낼 수 있다.

x17 레지스터는 Instruction의 rd field가 17이고, reg_write가 켜진 상황에서 레지스터 파일에 들어오는 write data의 값을 keep한다.

```
Mux alu_src_a_mux(
  .mux_in1(pc_to_mux),
  .mux_in2(A),
  .select_signal(alu_src_a),
  .mux_out(alu_src_mux_a_to_alu)
);

Mux4_1 alu_src_b_mux(
  .mux_in1(B),
  .mux_in2(always4),
  .mux_in3(imm_gen_to_alu_src_mux_b),
  .mux_in4(always0),
  .select_signal(alu_src_b),
  .mux_out(alu_src_mux_b_to_alu)
);
```

ALU에 들어가는 값을 선별하는 2개의 mux이다. ALU reuse 때문에 PC adder가 하던 연산도 추가로 해야 하므로 이를 고려한 port들이 추가되었다.

```
always @ (posedge clk) begin
  ALUOut <= alu_to_alu_out;
end

Mux pc_src_mux(
  .mux_in1(alu_to_alu_out),
  .mux_in2(ALUOut),
  .select_signal(pc_source),
  .mux_out(pc_src_mux_to_pc)
);
```

ALU result를 keep하는 ALUOut 레지스터를 추가하였다. 해당 레지스터는 branch instruction에서 conflict를 방지해준다. 만약 branch not taken인 경우 PC + 4를, branch taken인 경우 PC + Imm(IR)을 next PC 값으로 설정해야 한다. 하지만 branch condition을 판단하느라 ALU를 사용하기 때문에 conflict가 발생할 수 있다. 이를 방지하기 위해서 ID stage(2번째 state)에서 PC + 4 값을 ALUOut 레지스터에 저장하고, Not taken일 때는 ALUOut 레지스터에 저장된 값을 next PC로 전달하고 Taken일 때는 ALUOut 레지스터에 keep한 값이 아닌 PC + Imm(IR)을 한 ALU result를 next PC로 전달한다.

C. Resource reuse design and implementation

Multi-Cycle CPU에서는 2가지의 Resource reuse를 진행하였다.

첫 번째는 ALU resource reuse이다. 기존의 싱글 싸이클 CPU에서는 PC값을 계산하는데 2개의 adder를 추가적으로 사용했다. 멀티 싸이클 CPU에서는 PC값을 계산하는 연산을 모두 ALU에서 진행한다. 위의 B 항목에서 서술 했다시피, 이를 구현하기 위해 ALU input 앞에 2개의 mux(alu_src_a_mux, alu_src_b_mux)가 존재한다. 또한 ALU에서 conflict이 나는 것을 방지하기 위해 A, B, ALUOut 레지스터를 사용했다.

두 번째는 Memory reuse이다. 기존의 싱글 싸이클 CPU에서는 Instruction Memory와 Data Memory 2개가 존재한 것을 이번 멀티 싸이클 CPU에서는 하나로 reuse하였다. 이러다 보니, 메모리에서 읽은 값이 instruction인지 data인지를 체크하지 않으면 conflict가 발생한다. 항목 B에서 설명했다시피, IR와 MDR 레지스터를 만들고 IRWrite control 신호를 IR에 인가하여 두 개의 레지스터에 메모리에서 읽은 instruction과 data를 각각 저장(keep)할 수 있게 된다.

메모리와 ALU의 내부 구현 자체는 기존의 싱글 싸이클 CPU와 동일하다. 주변 Unit들과 control flow의 변화가 생긴 것이다.

D. Number of cycles took it took to run basic_ripes, and loop_ripes examples

```
TOTAL CYCLE        118

TOTAL CYCLE        979
```

basic_ripes를 돌렸을 때는 총 118 cycles, loop_ripes를 돌렸을 때는 총 979 cycles가 소요되었다.

4. Discussion

처음에는 마이크로 카운터와 마이크로 메모리를 사용하지 않고 단순 FSM으로 Control Unit을 구현하였다. 그때의 state flow chart와 지금의 state flow chart를 비교해보니, 확실히 마이크로 코드 개념을 적용했을 때 state flow chart가 효율적으로 작성된 것을 확인할 수 있다. 물론 본 Lab에서는 그렇게 많은 state가 필요하지 않아 크게 체감이 되지는 않았지만 실제 현업에서 control signal을 생성할 때 이보다 훨씬 많은 state가 필요할 것이므로 그럴 때 마이크로 코드 개념을 적용하여 state flow chart를 만드는 것이 큰 이득이 될 것이라 생각했다.

5. Conclusion

본 Lab을 통해서 멀티 싸이클 CPU를 구현하였다. Resource reuse나 Microcode 같은 개념을 코드레벨로 구현할 수 있는 좋은 경험이었다.