CSED311: Lab3 Multi-Cycle CPU

컴퓨터공학과 20220302 김지현

컴퓨터공학과 20220455 윤수인

1. Introduction

- 본 과제는 Multi-cycle CPU 가 Single-cycle CPU 보다 나은 이유를 이해하고, Verilog 를 사용하여 자체적인 datapath 와 control unit 을 갖는 Multi-cycle RISC-V CPU (RV32I)를 설계하고 구현하는 것을 목적으로 한다.
- Verilator 환경에서 실행하며, Ripes simulator 를 사용하여 각 instruction 에 따른 레지스터 값의 변화를 관찰한다.
- Single-cycle CPU 는 ALU, Memory, Register file 등 resource 의 underutilization 문제가 존재하였다. 이를 해결하기 위해 더 높은 clock frequency 를 사용하고, 각 instruction type 에 대해 다른 개수의 cycle 을 할당하는 방식으로 문제를 해결한다. 이러한 방식이 Multi-cycle CPU 에 해당한다.

- **Difference between single-cycle CPU and multi-cycle CPU**
  - **Single-cycle CPU**

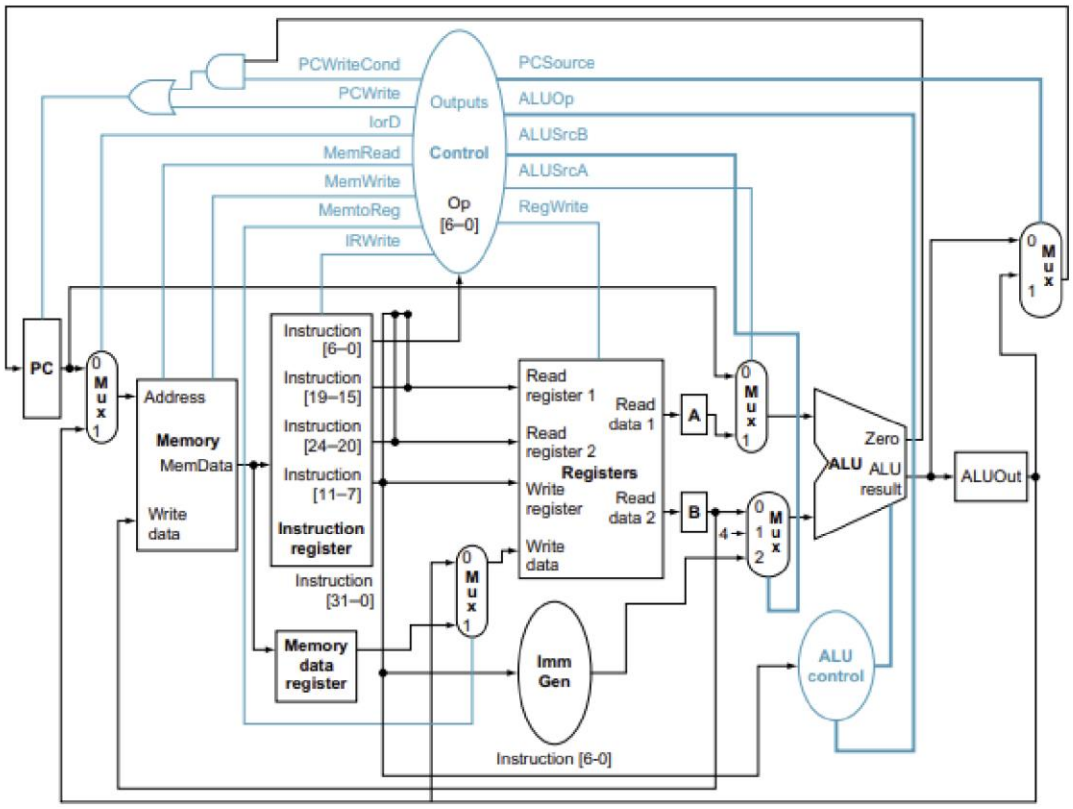
    모든 instruction 들을 하나의 clock cycle 내에서 시작, 완료한다. 이는 모든 instruction 처리에 동일한 시간이 소요되며, 이론적으로 간단한 구조로 설계할 수 있다. 하지만 모든 instruction 처리가 가장 긴 실행 시간을 가진 instruction 에 맞춰져야 하므로 전체 시스템의 clock frequency 가 제한된다. 또한 이와 관련하여 각 instruction 처리에 필요한 시간이 다름에도 불구하고 모든 instruction 을 동일한 cycle 내에서 처리해야 하기 때문에 효율성 역시 떨어질 수 있다.

  - **Multi-cycle CPU**

    Instruction 을 여러 단계에 걸쳐 처리한다. 각 단계는 별도의 clock cycle 에서 수행되며, 이를 통해 이 방식은 각 instruction 의 실행에 필요한 시간을 최적화하고, 하드웨어 자원을 재사용함으로써 효율성을 높인다. 이를 통해 결과적으로 전체적인 clock frequency 를 높일 수 있으며, resource reuse 를 통해 효율성을 극대화할 수 있다.

- **Why Multi-cycle CPU is better?**
- 앞서 서술한 바와 같이, Multi-cycle CPU 는 instruction 의 처리를 여러 단계로 나누어 각 단계에서 필요한 특정 하드웨어 resource 만을 사용한다. 이를 통해 동일한 resource 를 여러 instruction 처리에 재사용할 수 있게 함으로써 Resource utilization 을 극대화한다. 또한, 각 명령어를 처리하는 데 필요한 최소한의 클럭 사이클만을 사용하여, 전체적인 clock frequency 를 높일 수 있다. Single-cycle CPU 와 달리 Multi-cycle CPU 에서는 각 명령어의 복잡성에 따라 필요한 clock cycle 수가 다르므로, 더 높은 clock frequency 로 시스템을 운영할 수 있고, 이는 성능 향상으로 이어진다. Design 과 Implementation 에 관련된 다른 사항은 이후 부분에서 다루도록 한다.

2. Design



<그림 1: Multi-Cycle CPU (Datapath with Resource Reuse)>

위의 회로도를 참고하여, 기존 Lab2 에서 구현한 Single-cycle CPU 에서 다음과 같은 점을 수정하여 Multi-cycle CPU 를 구현하였다. Instruction / Data memory 를 하나로 합쳐 Memory 관련 부분을 총괄적으로 다루었다. Control unit 에서 다음 시그널들을 추가적으로 관리한다. ALUSrcA, ALUSrcB, IorD, IRWrite, PCSource, PCWrite, PCWriteNotCond 시그널들이 새로 추가되었다. 그리고, Resource 를 이용한 연산과 PC value 를 업데이트하는 로직을 병합하는 방식으로 구현하였기 때문에, 두 결과 값을 따로 저장하기 위해 ALUOut 레지스터를 추가적으로 활용한다.

자세한 구현과 Microcode 등에 대한 자세한 설명은 Implementation 파트에서 진행한다.

3. Implementation

- **pc.v**

```
module pc (
    input reset,
    input clk,
    input [31:0] next_pc,
    output reg [31:0] current_pc);

    always @(posedge clk) begin
        if (reset) begin
            current_pc <= 32'b0;
        end
        else begin
            current_pc <= next_pc;
        end
    end

endmodule
```

<그림 2 : pc.v 모듈의 구성>

pc.v 모듈에서는 reset 이 1 인 경우 current_pc 의 값을 0 으로 업데이트하고, reset 이 1 이 아닌 경우에는 next_pc 의 값으로 업데이트한다. 이 과정은 clock synchronous 하게 이루어진다.

- **Address_select_logic.v**

```
`include "opcodes.v"
`include "Stages.v"

module Address_select_logic(
    input [6:0] opcode,
    input [6:0] data,
    input [2:0] add_state,
    input [1:0] AddrCtrl,
    input IRWrite,
    output [2:0] next_state
    );

    reg [2:0] ROM1;
    reg [2:0] ROM2;
    wire [2:0] ROM1_wire;
    wire [2:0] ROM2_wire;

    assign ROM1_wire[2:0] = ROM1;
    assign ROM2_wire[2:0] = ROM2;

    initial begin
        ROM1 = 0;
        ROM2 = 0;
    end

    always @(*) begin
        if(IRWrite && data == `JAL) begin // J
            ROM1 = `EX;
            ROM2 = `IF;
        end
        else if(opcode == `ARITHMETIC) begin /
            ROM1 = `EX;
            ROM2 = `IF;
        end
        else if(opcode == `BRANCH) begin
            ROM1 = `B;
            ROM2 = `IF;
        end
        else if(opcode == `STORE) begin
            ROM1 = `MEM;
            ROM2 = `IF;
        end
        else if(opcode == `LOAD) begin
            ROM1 = `MEM;
            ROM2 = `WB;
        end
        else begin
            ROM1 = `IF;
            ROM2 = `IF;
        end
    end

    MUX4x1_2bit next_state_MUX(
        .in_bit(AddrCtrl),
        .in3(0),
        .in2(ROM1_wire),     // ROM1, ROM2 순서
        .in1(ROM2_wire),
        .in0(add_state),
        .out(next_state)
    );

endmodule
```

<그림 3: Address_select_logic.v 모듈의 구성>

Address_select_logic.v 모듈은 Appendix C 의 예시를 참고하여 구현하였다.

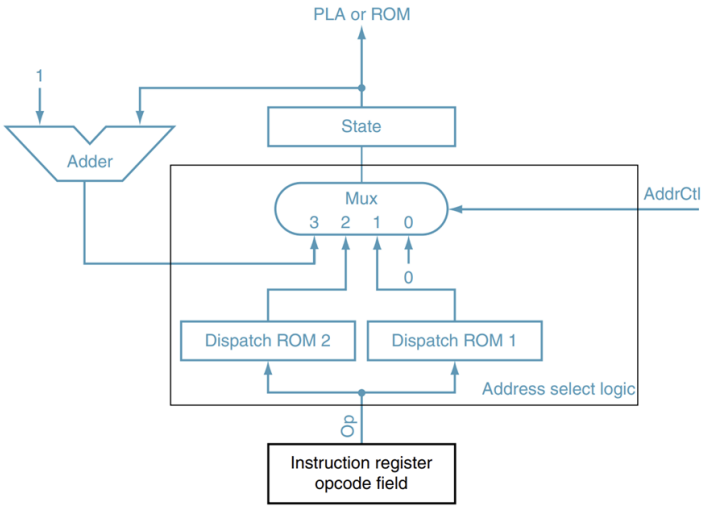


FIGURE C.4.2 This is the address select logic for the control unit of Figure C.4.1.

<그림 4: Address select logic 의 대략적인 형태 - Appendix C>

AddrCtl value	Action
0	Set state to 0
1	Dispatch with ROM 1
2	Dispatch with ROM 2
3	Use the incremented state

Dispatch ROM 1		
Op	Opcode name	Value
000000	R-format	0110
000100	beq	1000
100011	lw	0010
101011	sw	0010

Dispatch ROM 2		
Op	Opcode name	Value
100011	lw	0011
101011	sw	0101

FIGURE C.4.3 The dispatch ROMs each have $2^6 = 64$ entries that are 4 bits wide, since that is the number of bits in the state encoding. This figure only shows the entries in the ROM that are of interest for this subset. The first column in each table indicates the value of Op, which is the address used to access the dispatch ROM. The second column shows the symbolic name of the opcode. The third column indicates the value at that address in the ROM.

<그림 5: AddrCtrl 값에 따른 Action 과, 각 ROM value 설정 - Appendix C>

Address_select_logic.v 모듈에서는 opcode 와 IRWrite 시그널 값, data 를 고려하여 ROM1 과 ROM2 의 값을 설정한다. ROM1 과 ROM2 는 다음 state 를 결정하기 위한 임시 값을 저장하는 데 사용된다. Always @(*) 블록 내에서, 다음과 같은 조건에 따라 ROM1, ROM2 의 값을 설정한다.

- JAL instruction 처리: IRWrite 가 1 이고, JAL 연산을 수행하는 경우 ROM1 과 ROM2 는 각각 EX 와 IF state 로 설정된다. 이는 JAL 명령어가 EX state 를 거친 후 다음 instruction 을 받아오기 위해 IF state 로 진행되어야 함을 의미한다.
- ARITHMETIC, BRANCH, STORE, LOAD 처리: 각 instruction 타입에 따라 ROM1 과 ROM2 의 값을 설정한다. 예를 들어, LOAD 는 MEM state 이후 WB state 가 필요하고, BRANCH 의 경우 B state 이후 IF state 로 진행됨을 의미한다.
- 그 외 경우, ROM1, ROM2 를 모두 IF 로 설정한다.

최종적으로, MUX4x1_2bit 모듈을 사용하여 AddrCtrl 신호에 따라 다음 state 를 결정한다. 이 mux 는 AddrCtrl 값에 따라 add_state, ROM2_wire, ROM1_wire, 또는 0 중 하나를 선택하여 next_state 로 출력한다. (AddrCtrl 신호의 값은 ControlUnit 모듈에서 결정된다.)

이 과정은 clock asynchronous 하게 이루어진다.

- **Adder.v**

```
module Adder(
    input [2:0] A,
    input [2:0] B,
    output [2:0] result
    );
    assign result = A + B;
endmodule
```

<그림 6: Adder.v 모듈의 구성>

Adder.v 모듈은 microcode controller state design 에서 사용되며 input 으로는 state_wire 와 1 이 들어가고 output 으로는 add_state 이 할당되어 있다. 이 과정은 clock asynchronous 하게 이루어진다.

- **Memory.v**

```
module Memory #(parameter MEM_DEPTH = 16384) (input reset,
                                              input clk,
                                              input [31:0] addr,    // address of the memory
                                              input [31:0] din,     // data to be written
                                              input mem_read,       // is read signal driven?
                                              input mem_write,      // is write signal driven?
                                              output [31:0] dout);  // output of the data memory at addr
  integer i;
  // Memory
  reg [31:0] mem[0: MEM_DEPTH - 1];
  // Do not touch mem_addr
  wire [31:0] mem_addr;
  assign mem_addr = addr >> 2;

  // Asynchrnously read data from the memory
  assign dout = (mem_read) ? mem[mem_addr] : 32'b0;

  always @(posedge clk) begin
    // Initialize data memory (do not touch)
    if (reset) begin
      for (i = 0; i < MEM_DEPTH; i = i + 1)
        // DO NOT TOUCH COMMENT BELOW
        /* verilator lint_off BLKSEQ */
        mem[i] = 32'b0;
        /* verilator lint_on BLKSEQ */
        // DO NOT TOUCH COMMENT ABOVE
      // Provide path of the file including instructions with binary format
      //$readmemh("C:/Users/maril/Desktop/junior1/ComputerArchitecture/Lab3/lab3_4_12/student_tb/basic_mem.txt", mem);
      //$readmemh("C:/Users/maril/Desktop/junior1/ComputerArchitecture/Lab3/lab3_4_12/student_tb/ifelse_mem.txt", mem);
      //$readmemh("C:/Users/maril/Desktop/junior1/ComputerArchitecture/Lab3/lab3_4_12/student_tb/loop_mem.txt", mem);
      //$readmemh("C:/Users/maril/Desktop/junior1/ComputerArchitecture/Lab3/lab3_4_12/student_tb/non-controlflow_mem.txt", mem);
      $readmemh("C:/Users/maril/Desktop/junior1/ComputerArchitecture/Lab3/lab3_4_12/student_tb/recursive_mem.txt", mem);
    end

    // Synchronously write data to the memory
    else begin
      if (mem_write)
        mem[mem_addr] <= din;
    end
  end
endmodule
```

<그림 7: memory.v 모듈의 구성>

Memory.v 모듈은 reset 이 1 인 경우 clock synchronous 하게 memory 를 초기화한다. Reset 이 1 이 아닌 경우, mem_read control signal 이 1 일 경우 해당 주소의 데이터를 dout 에

asynchronous 하게 저장하고 mem_write control signal 이 1 일 경우 해당 주소에 din 을 synchronous 하게 저장한다.

- **RegisterFile.v**

```
module RegisterFile(input reset,
                    input clk,
                    input [4:0] rs1,          // source register 1
                    input [4:0] rs2,          // source register 2
                    input [4:0] rd,           // destination register
                    input [31:0] rd_din,      // input data for rd
                    input write_enable,       // RegWrite signal
                    output [31:0] rs1_dout,   // output of rs 1
                    output [31:0] rs2_dout,
                    output [31:0] print_reg[0:31]);  // output of rs 2
  integer i;
  // Register file
  reg [31:0] rf[0:31];
  assign print_reg = rf;
  // Asynchronously read register file
  assign rs1_dout = rf[rs1];
  assign rs2_dout = rf[rs2];

  always @(posedge clk) begin
    // Initialize register file (do not touch)
    if (reset) begin
      for (i = 0; i < 32; i = i + 1)
        rf[i] <= 32'b0;
      rf[2] <= 32'h2ffc; // stack pointer
    end

    // Synchronously write data to the register
    else begin
      if (write_enable && rd != 0)
        rf[rd] <= rd_din;
    end
  end
endmodule
```

<그림 8: RegisterFile.v 모듈의 구성>

RegisterFile.v 모듈은 register 의 위치를 받아와 해당 register 의 값을 읽거나 쓰는 역할을 한다. Register 값을 읽는 과정은 clock asynchronous 하고, 5-bit input 인 rs1 과 rs2 각각에 해당하는 register 의 값을 32-bit output 인 rs1_dout, rs2_dout 에 각각 저장한다.

Register 값을 쓰는 과정은 clock synchronous 하고, write_enable 시그널이 1 이고 destination register(rd)가 존재할 때 32-bit input data 인 rd_din 을 rd 위치에 해당하는 register 에 저장한다.

Reset signal 이 1 일 때는 clock synchronous 하게 모든 register 를 0 으로 저장한 뒤 stack pointer 인 rf[2] 를 2ffc 로 할당한다.

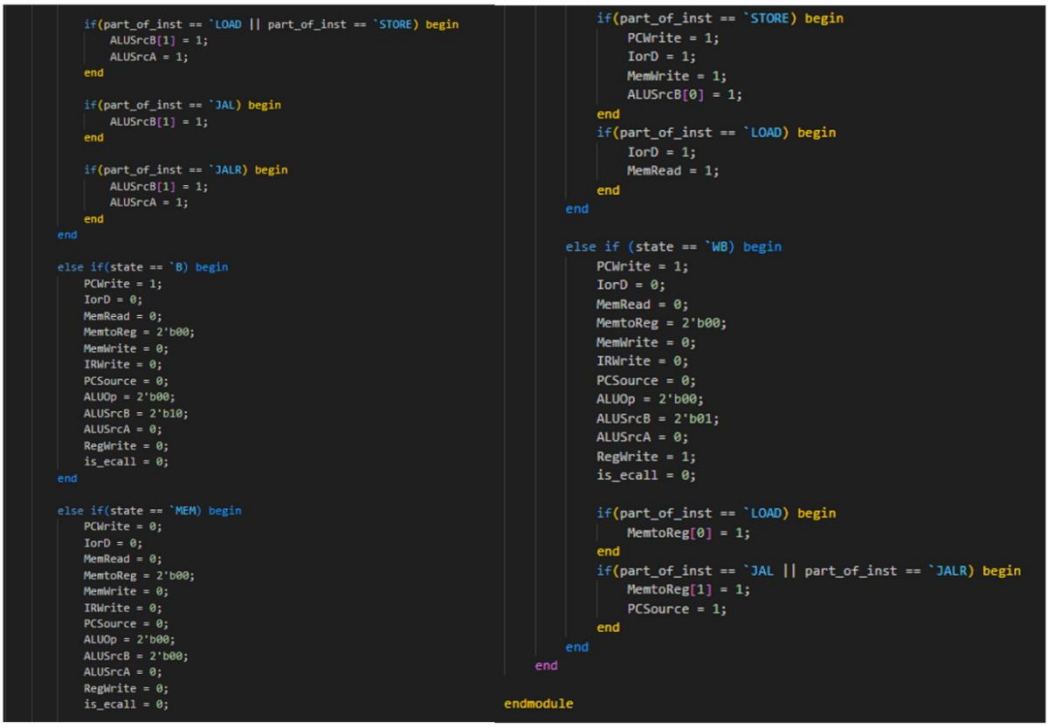
- **ControlUnit.v**

```v
`include "opcodes.v"
`include "Stages.v"


module ControlUnit(
    input reset,
    input clk,
    input [6:0] part_of_inst, // input
    input PCcond,
    input [6:0] data,
    output reg PCWrite,
    output reg IorD,
    output reg MemRead,
    output reg MemWrite,
    output reg [1:0] MemtoReg,
    output reg IRWrite,
    output reg PCSource,
    output reg [1:0] ALUOp,
    output reg [1:0] ALUSrcB,
    output reg ALUSrcA,
    output reg RegWrite,
    output reg is_ecall // output (ecall inst)
);

    reg [2:0] state;
    wire [2:0] state_wire;
    wire [2:0] next_state;
    wire [2:0] add_state;
    wire [1:0] AddrCtrl;
    wire IRWrite_wire;

    assign state_wire = state;
    assign IRWrite_wire = IRWrite;

    // AddrCtrl[1], [0] 순서 바꿈.
    assign AddrCtrl[1] = (IRWrite && (data == `JAL) && (state == `IF)) ||
                         ((part_of_inst == `STORE || part_of_inst == `LOAD || (PCcond && (part_of_inst == `BRANCH))) && (state == `EX)) ||
                         ((!(PCcond) && (part_of_inst == `BRANCH)) && (state == `EX)) ||
                         ((part_of_inst == `STORE) && (state == `MEM)) ||
                         (state == `WB) ||
                         (state == `B) ||
                         (part_of_inst == `ECALL);

    assign AddrCtrl[0] = ((part_of_inst == `LOAD) && (state == `MEM)) ||
                         ((part_of_inst == `STORE) && (state == `MEM)) ||
                         ((!(PCcond) && (part_of_inst == `BRANCH)) && (state == `EX)) ||
                         (state == `WB) ||
                         (state == `B) ||
                         (part_of_inst == `ECALL);
```

```v
initial begin
    PCWrite = 0;
    IorD = 0;
    MemRead = 0;
    MemtoReg = 2'b00;
    MemWrite = 0;
    IRWrite = 0;
    PCSource = 0;
    ALUOp = 2'b00;
    ALUSrcB = 2'b00;
    ALUSrcA = 0;
    RegWrite = 0;
    is_ecall = 0;
    state = `IF;
end

Adder Add1(
    .A(state_wire),
    .B(1),
    .result(add_state)
);

Address_select_logic Address_select_logic(
    .opcode(part_of_inst),
    .data(data),
    .add_state(add_state),
    .AddrCtrl(AddrCtrl),
    .IRWrite(IRWrite_wire),
    .next_state(next_state)
);

always @(posedge clk) begin
    if(reset) begin
        state <= `IF;
    end
    else begin
        state <= next_state;
    end
end
```

```v
always @(*) begin
    PCWrite = 0;
    IorD = 0;
    MemRead = 0;
    MemtoReg = 2'b00;
    MemWrite = 0;
    IRWrite = 0;
    PCSource = 0;
    ALUOp = 2'b00;
    ALUSrcB = 2'b00;
    ALUSrcA = 0;
    RegWrite = 0;
    is_ecall = 0;

    if(state == `IF) begin
        //PCWrite = 0;
        IorD = 0;
        MemRead = 1;
        MemtoReg = 2'b00;
        MemWrite = 0;
        IRWrite = 1;
        PCSource = 0;
        ALUOp = 2'b00;
        ALUSrcB = 2'b01;
        ALUSrcA = 0;
        RegWrite = 0;
        //is_ecall = 0;

        if(data == `ECALL) begin
            PCWrite = 1;
            is_ecall = 1;
        end
        else begin
            PCWrite = 0;
            is_ecall = 0;
        end
    end
```

```v
    else if(state == `ID) begin
        PCWrite = 0;
        IorD = 0;
        MemRead = 1;
        MemtoReg = 2'b00;
        MemWrite = 0;
        IRWrite = 1;
        PCSource = 0;
        ALUOp = 2'b00;
        ALUSrcB = 2'b01;
        ALUSrcA = 0;
        RegWrite = 0;
        is_ecall = 0;
    end

    else if(state == `EX) begin
        PCWrite = 0;
        IorD = 0;
        MemRead = 0;
        MemtoReg = 2'b00;
        MemWrite = 0;
        IRWrite = 0;
        PCSource = 0;
        ALUOp = 2'b00;
        ALUSrcB = 2'b00;
        ALUSrcA = 0;
        RegWrite = 0;
        is_ecall = 0;

        if(part_of_inst == `BRANCH) begin
            if(!PCcond) begin
                PCWrite = 1;
                PCSource = 1;
            end

            ALUOp[0] = 1;
            ALUSrcA = 1;
        end

        if(part_of_inst == `ARITHMETIC) begin
            ALUOp[1] = 1;
            ALUSrcA = 1;
        end

        if(part_of_inst == `ARITHMETIC_IMM) begin
            ALUOp[1] = 1;
            ALUSrcB[1] = 1;
            ALUSrcA = 1;
        end
```



```
        if(part_of_inst == `LOAD || part_of_inst == `STORE) begin
            ALUSrcB[1] = 1;
            ALUSrcA = 1;
        end

        if(part_of_inst == `JAL) begin
            ALUSrcB[1] = 1;
        end

        if(part_of_inst == `JALR) begin
            ALUSrcB[1] = 1;
            ALUSrcA = 1;
        end
    end

    else if(state == `B) begin
        PCWrite = 1;
        IorD = 0;
        MemRead = 0;
        MemtoReg = 2'b00;
        MemWrite = 0;
        IRWrite = 0;
        PCSource = 0;
        ALUOp = 2'b00;
        ALUSrcB = 2'b10;
        ALUSrcA = 0;
        RegWrite = 0;
        is_ecall = 0;
    end

    else if(state == `MEM) begin
        PCWrite = 0;
        IorD = 0;
        MemRead = 0;
        MemtoReg = 2'b00;
        MemWrite = 0;
        IRWrite = 0;
        PCSource = 0;
        ALUOp = 2'b00;
        ALUSrcB = 2'b00;
        ALUSrcA = 0;
        RegWrite = 0;
        is_ecall = 0;
```

```
        if(part_of_inst == `STORE) begin
            PCWrite = 1;
            IorD = 1;
            MemWrite = 1;
            ALUSrcB[0] = 1;
        end
        if(part_of_inst == `LOAD) begin
            IorD = 1;
            MemRead = 1;
        end
    end

    else if (state == `WB) begin
        PCWrite = 1;
        IorD = 0;
        MemRead = 0;
        MemtoReg = 2'b00;
        MemWrite = 0;
        IRWrite = 0;
        PCSource = 0;
        ALUOp = 2'b00;
        ALUSrcB = 2'b01;
        ALUSrcA = 0;
        RegWrite = 1;
        is_ecall = 0;

        if(part_of_inst == `LOAD) begin
            MemtoReg[0] = 1;
        end
        if(part_of_inst == `JAL || part_of_inst == `JALR) begin
            MemtoReg[1] = 1;
            PCSource = 1;
        end
    end
  end

endmodule
```

<그림 9, 10, 11: ControlUnit.v 모듈의 구성>

ControlUnit.v 모듈은 part_of_inst(opcode)와 현재 state 와 ecall 여부에 따라 PCWrite, IorD, MemRead, MemtoReg, MemWrite, IRWrite, PCSource, ALUOp, ALUSrcB, ALUSrcA, RegWrite, is_ecall 시그널의 값을 활성화/비활성화하여 CPU 의 동작을 결정한다.

초기에 각 시그널의 값을 0 또는 비활성화 상태로 설정하고, 상황에 따라 특정 시그널의 값을 변경한다.

- 현재 state 가 IF 인 경우
  - MemRead, IRWrite 시그널의 값을 1 로 설정하고, ALUSrcB 는 2'b01 로 설정한다.
  - Data == `ECALL 일 경우, 추가적으로 PCWrite 와 is_ecall 의 값을 1 로 설정한다.
- 현재 state 가 ID 인 경우
  - MemRead, IRWrite 시그널의 값을 1 로 설정하고, ALUSrcB 는 2'b01 로 설정한다.
- 현재 state 가 EX 인 경우
  - Part_of_inst 의 종류에 따라 시그널의 값을 다르게 설정한다.
  - Part_of_inst 가 `BRANCH 인 경우, ALUOp[0], ALUSrcA 의 값을 1 로 설정하며, PCCond 가 활성화되지 않은 경우 추가로 PCWrite 와 PCSource 값을 1 로 설정한다.
  - Part_of_inst 가 `ARITHMETIC 인 경우, ALUOp[1]과 ALUSrcA 의 값을 1 로 설정한다.

    - Part_of_inst 가 `ARITHMETIC_IMM 인 경우, ALUOp[1], ALUSrcB[1], ALUSrcA 의 값을 1 로 설정한다.
    - Part_of_inst 가 `LOAD 또는 `STORE 인 경우, ALUSrcB[1]과 ALUSrcA 의 값을 1 로 설정한다.
    - Part_of_inst 가 `JAL 인 경우, ALUSrcB[1]의 값을 1 로 설정한다.
    - Part_of_inst 가 `JALR 인 경우, ALUSrcB[1], ALUSrcA 의 값을 1 로 설정한다.
- 현재 state 가 B 인 경우
    - PCWrite, ALUSrcB[1]의 값을 1 로 설정한다.
- 현재 state 가 MEM 인 경우
    - Part_of_inst 가 `STORE 인 경우, PCWrite, IorD, MemWrite, ALUSrcB[0]의 값을 1 로 설정한다.
    - Part_of_inst 가 `LOAD 인 경우, IorD 와 MemRead 의 값을 1 로 설정한다.
- 현재 state 가 WB 인 경우
    - PCWrite, ALUSrcB[0], RegWrite 의 값을 1 로 설정한다.
    - Part_of_inst 가 `LOAD 인 경우, 추가로 MemtoReg[0]의 값을 1 로 설정한다.
    - Part_of_inst 가 `JAL 또는 `JALR 인 경우, 추가로 MemtoReg[1], PCSource 의 값을 1 로 설정한다.

이 과정은 **clock asynchronous** 하게 진행된다.

또한, Address_select_logic 모듈에서 활용하기 위한 AddrCtrl 의 각 인덱스의 값도 각각 설정한다. AddrCtrl[1], AddrCtrl[0]의 값은 각각 코드에 작성한 조건에 해당할 시 값을 1 로 설정한다.

Adder 모듈을 사용하여 현재 state 에 1 을 더한 값, 즉 별 다른 일이 없다면 이동하여야 할 state 인 add_state 를 얻는다. Address_select_logic 모듈에서 AddrCtrl 의 값을 확인하여 다음 state 를 결정할 때, 0, ROM1, ROM2, add_state 중 어떤 값을 반영할지 결정할 때 사용된다.

Next_state 가 결정되면, reset 이 활성화되지 않은 경우 state 를 next_state 로 변경한다. 만약 reset 이 활성화되었다면, state 를 IF 로 변경한다. 이 과정은 **clock synchronous** 하게 진행된다.

- **ImmediateGenerator.v**

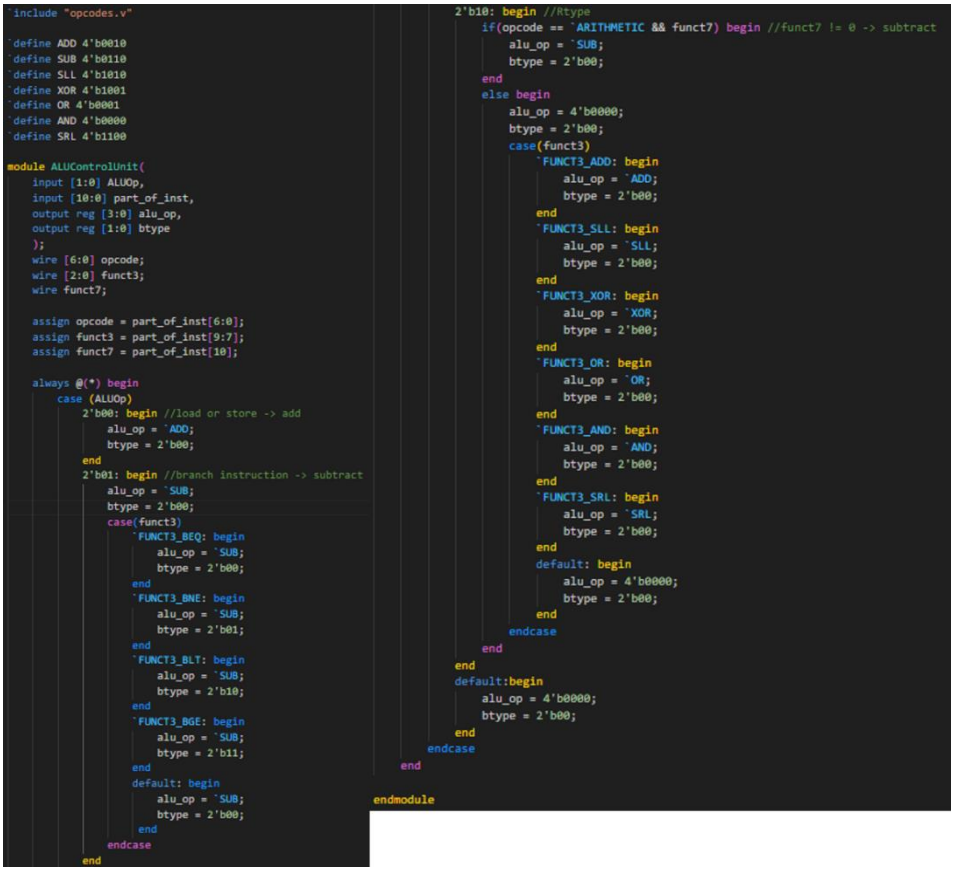
```
`include "opcodes.v"

module ImmediateGenerator(
    input [31:0] part_of_inst,  // input
    output reg [31:0] imm_gen_out  // output
  );

  reg [6:0] opcode;

  always @(*) begin
    imm_gen_out = 32'b0;
    opcode = part_of_inst[6:0];
    case(opcode)
        /*`ARITHMETIC: begin
            imm_gen_out = 32'b0;
        end
        */
        `ARITHMETIC_IMM: begin
            if(part_of_inst[31] == 1) begin  // signed
                imm_gen_out = {20'hfffff, part_of_inst[31:20]};
            end
            else begin
                imm_gen_out = {20'h0, part_of_inst[31:20]};
            end
        end

        `LOAD: begin
            if(part_of_inst[31] == 1) begin  // signed
                imm_gen_out = {20'hfffff, part_of_inst[31:20]};
            end
            else begin
                imm_gen_out = {20'h0, part_of_inst[31:20]};
            end
        end

        `JALR: begin
            if(part_of_inst[31] == 1) begin  // signed
                imm_gen_out = {20'hfffff, part_of_inst[31:20]};
            end
            else begin
                imm_gen_out = {20'h0, part_of_inst[31:20]};
            end
        end
```

```
        `STORE: begin
            if(part_of_inst[31] == 1) begin  // signed
                imm_gen_out = {20'hfffff, part_of_inst[31:25], part_of_inst[11:8], part_of_inst[7]};
            end
            else begin
                imm_gen_out = {20'h0, part_of_inst[31:25], part_of_inst[11:8], part_of_inst[7]};
            end
        end
        `BRANCH: begin
            if(part_of_inst[31] == 1) begin  // signed
                imm_gen_out = {19'h7ffff, part_of_inst[31], part_of_inst[7], part_of_inst[30:25], part_of_inst[11:8], 1'b0};
            end
            else begin
                imm_gen_out = {19'h0, part_of_inst[31], part_of_inst[7], part_of_inst[30:25], part_of_inst[11:8], 1'b0};
            end
        end
        `JAL: begin
            if(part_of_inst[31] == 1) begin  // signed
                imm_gen_out = {11'h7ff, part_of_inst[31], part_of_inst[19:12], part_of_inst[20], part_of_inst[30:21],1'b0};
            end
            else begin
                imm_gen_out = {11'h0, part_of_inst[31], part_of_inst[19:12], part_of_inst[20], part_of_inst[30:21],1'b0};
            end
        end
        /*
        `ECALL: begin
            imm_gen_out = 32'b0;
        end */

        default: begin
            imm_gen_out = 32'b0;
        end
    endcase
  end


endmodule
```

<그림 12, 13: ImmediateGenerator.v 모듈의 구성>

ImmediateGenerator.v 모듈은 32-bit 의 part_of_inst 를 input 으로 받아와 하위 7 bits 를 opcode 에 저장한다. Opcode 에 따라 적절한 immediate value 를 imm_gen_out 에 저장한다. Signed 연산 여부는 part_of_inst 의 상위 1 bit 로 판별한다. 전체 과정은 clock asynchronous 하게 이루어진다.

- **ALUControlUnit.v**



```
`include "opcodes.v"

`define ADD 4'b0010
`define SUB 4'b0110
`define SLL 4'b1010
`define XOR 4'b1001
`define OR 4'b0001
`define AND 4'b0000
`define SRL 4'b1100

module ALUControlUnit(
    input [1:0] ALUOp,
    input [10:0] part_of_inst,
    output reg [3:0] alu_op,
    output reg [1:0] btype
    );
    wire [6:0] opcode;
    wire [2:0] funct3;
    wire funct7;

    assign opcode = part_of_inst[6:0];
    assign funct3 = part_of_inst[9:7];
    assign funct7 = part_of_inst[10];

    always @(*) begin
        case (ALUOp)
            2'b00: begin //load or store -> add
                alu_op = `ADD;
                btype = 2'b00;
            end
            2'b01: begin //branch instruction -> subtract
                alu_op = `SUB;
                btype = 2'b00;
                case(funct3)
                    `FUNCT3_BEQ: begin
                        alu_op = `SUB;
                        btype = 2'b00;
                    end
                    `FUNCT3_BNE: begin
                        alu_op = `SUB;
                        btype = 2'b01;
                    end
                    `FUNCT3_BLT: begin
                        alu_op = `SUB;
                        btype = 2'b10;
                    end
                    `FUNCT3_BGE: begin
                        alu_op = `SUB;
                        btype = 2'b11;
                    end
                    default: begin
                        alu_op = `SUB;
                        btype = 2'b00;
                     end
                endcase
            end
            2'b10: begin //Rtype
                if(opcode == `ARITHMETIC && funct7) begin //funct7 != 0 -> subtract
                    alu_op = `SUB;
                    btype = 2'b00;
                end
                else begin
                    alu_op = 4'b0000;
                    btype = 2'b00;
                    case(funct3)
                        `FUNCT3_ADD: begin
                            alu_op = `ADD;
                            btype = 2'b00;
                        end
                        `FUNCT3_SLL: begin
                            alu_op = `SLL;
                            btype = 2'b00;
                        end
                        `FUNCT3_XOR: begin
                            alu_op = `XOR;
                            btype = 2'b00;
                        end
                        `FUNCT3_OR: begin
                            alu_op = `OR;
                            btype = 2'b00;
                        end
                        `FUNCT3_AND: begin
                            alu_op = `AND;
                            btype = 2'b00;
                        end
                        `FUNCT3_SRL: begin
                            alu_op = `SRL;
                            btype = 2'b00;
                        end
                        default: begin
                            alu_op = 4'b0000;
                            btype = 2'b00;
                        end
                    endcase
                end
            end
            default:begin
                alu_op = 4'b0000;
                btype = 2'b00;
            end
        endcase
    end

endmodule
```

<그림 14: ALUControlUnit.v 모듈의 구성>

ALUControlUnit.v 모듈은 opcode, funct3, funct7 에 따라 alu 가 수행해야 할 연산을 결정하는 모듈로, 경우에 따라 위 그림에서 지정된 instruction 번호를 alu_op 에 할당하고, Branch 여부와 관련하여 btype 의 값을 변경한다. Opcode, funct3, funct7 은 각각 part_of_inst 의 [6:0], [9:7], [10] 부분에 해당한다. 각 instruction 에 해당하는 번호는 ADD, SUB, SLL, XOR, OR, AND, SRL 을 각각 4'b0010, 4'b0110, 4'b1010, 4'b1001, 4'b0001, 4'b0000, 4'b1100 으로 설정하였다.

2-bit input 인 ALUOp(control unit 에서 결정되는 값으로, 실제 alu 가 행해야 할 연산의 종류에 해당하는 alu_op 와는 다르다)의 값에 따라 각기 다른 값으로 alu_op 와 btype 을 설정한다. 각 instruction 에 따른 ALUOp 값은 아래의 표를 참고하였다.

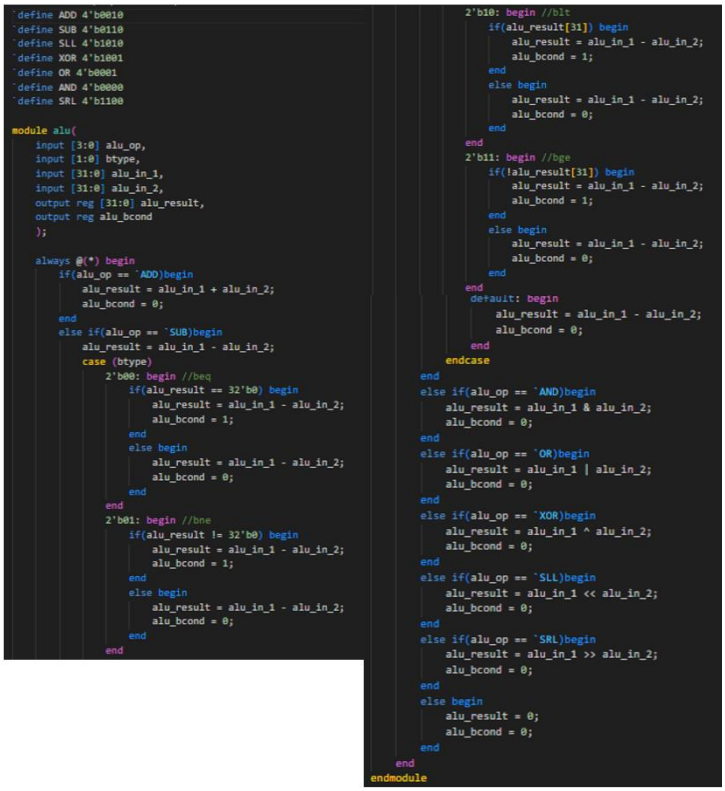
ALU Control	Instruction
0010	addu
0110	subu
0000	and
0001	or
1001	xor
1010	sll
1011	sra
1100	srl
0111	slt
1110	sltu
xxxx	jr

<그림 15: ALUControlUnit.v 모듈의 구성>

- ALUOp 가 2'b00 (LOAD 또는 STORE)인 경우
  - Alu 는 add 연산을 수행해야 하므로, alu_op 는 `ADD 로 설정한다. Branch 는 발생하지 않으므로 btype 은 2'b00 으로 설정한다.
- ALUOp 가 2'b01 (Branch instruction)인 경우
  - Alu 는 subtract 연산을 수행해야 하므로, alu_op 는 `SUB 로 설정한다.
  - Funct3 의 값에 따라, BEQ, BNE, BLT, BGE 의 각기 다른 branch instruction 을 수행하므로, btype 을 각각 2'b00, 2'b01, 2'b10, 2'b11 로 설정한다.
- ALUOp 가 2'b10 (Rtype)인 경우
  - Opcode 가 `ARITHMETIC 이고, funct7 이 0 이 아닌 경우, alu 는 subtract 연산을 수행한다. 따라서 alu_op 는 `SUB 로 설정한다.
  - 위에 해당하지 않는 경우, funct3 의 값에 따라 각각 ADD, SLL, XOR, OR, AND, SRL 로 구분하여 alu_op 의 값을 설정한다.
  - Branch 가 발생하지 않으므로 btype 은 2'b00 으로 설정한다.

이 과정은 clock asynchronous 하게 이루어진다.

- **alu.v**



```
`define ADD 4'b0010
`define SUB 4'b0110
`define SLL 4'b1010
`define XOR 4'b1001
`define OR 4'b0001
`define AND 4'b0000
`define SRL 4'b1100

module alu(
    input [3:0] alu_op,
    input [1:0] btype,
    input [31:0] alu_in_1,
    input [31:0] alu_in_2,
    output reg [31:0] alu_result,
    output reg alu_bcond
    );

    always @(*) begin
        if(alu_op == `ADD)begin
            alu_result = alu_in_1 + alu_in_2;
            alu_bcond = 0;
        end
        else if(alu_op == `SUB)begin
            alu_result = alu_in_1 - alu_in_2;
            case (btype)
                2'b00: begin //beq
                    if(alu_result == 32'b0) begin
                        alu_result = alu_in_1 - alu_in_2;
                        alu_bcond = 1;
                    end
                    else begin
                        alu_result = alu_in_1 - alu_in_2;
                        alu_bcond = 0;
                    end
                end
                2'b01: begin //bne
                    if(alu_result != 32'b0) begin
                        alu_result = alu_in_1 - alu_in_2;
                        alu_bcond = 1;
                    end
                    else begin
                        alu_result = alu_in_1 - alu_in_2;
                        alu_bcond = 0;
                    end
                end
                2'b10: begin //blt
                    if(alu_result[31]) begin
                        alu_result = alu_in_1 - alu_in_2;
                        alu_bcond = 1;
                    end
                    else begin
                        alu_result = alu_in_1 - alu_in_2;
                        alu_bcond = 0;
                    end
                end
                2'b11: begin //bge
                    if(!alu_result[31]) begin
                        alu_result = alu_in_1 - alu_in_2;
                        alu_bcond = 1;
                    end
                    else begin
                        alu_result = alu_in_1 - alu_in_2;
                        alu_bcond = 0;
                    end
                end
                default: begin
                    alu_result = alu_in_1 - alu_in_2;
                    alu_bcond = 0;
                end
            endcase
        end
        else if(alu_op == `AND)begin
            alu_result = alu_in_1 & alu_in_2;
            alu_bcond = 0;
        end
        else if(alu_op == `OR)begin
            alu_result = alu_in_1 | alu_in_2;
            alu_bcond = 0;
        end
        else if(alu_op == `XOR)begin
            alu_result = alu_in_1 ^ alu_in_2;
            alu_bcond = 0;
        end
        else if(alu_op == `SLL)begin
            alu_result = alu_in_1 << alu_in_2;
            alu_bcond = 0;
        end
        else if(alu_op == `SRL)begin
            alu_result = alu_in_1 >> alu_in_2;
            alu_bcond = 0;
        end
        else begin
            alu_result = 0;
            alu_bcond = 0;
        end
    end
endmodule
```

<그림 16: alu.v 모듈의 구성>

alu.v 모듈은 앞서 ALUControlUnit.v 모듈에서 지정한 alu_op 에 따라 알맞은 연산을 수행하여 연산 결과를 alu_result 와 alu_bcond 에 저장한다. 이 과정은 clock asynchronous 하게 이루어진다.

- **MUX2x1.v**

```
module MUX2x1(input in_bit,
input [31:0] in1,
input [31:0] in0,
output [31:0] out);

    assign out = in_bit ? in1 : in0;

endmodule
```

<그림 17: MUX2x1.v 모듈의 구성>

MUX2to1.v 모듈은 select 의 값에 따라 in1 과 in2 중 하나의 값을 반환하는 역할로, 전부 cpu.v 에서 사용되었다. MUX2to1 이 사용된 부분은 다음과 같다.

- addr_MUX: Instruction memory access 인지 data memory access 인지 구분하는 IorD control 시그널의 값에 따라 ALUOut_wire 과 current_pc 값 중 하나를 선택하여 addr 에 저장한다.
- ALU_in1_MUX: ALUSrcA signal 에 따라 A_wire 값, 즉 rs1_out 값 또는 current_pc 값을 ALU 의 첫 번째 input(ALU_in1) 으로 가진다.
- next_pc_MUX: PCSource signal 에 따라 ALUOut_wire 값 또는 ALU_result 값을 next pc value 로 가진다.
- rs1_out_MUX(2bit version): is_ecall 컨트롤이 1 이 되었을 때, 17 또는 rs1 을 rs1_out 에 저장한다. 이 경우 2 bit 로 따로 처리할 필요가 있어, MUX2x1 을 32 bit 에서 2 bit 버전으로 변경한 MUX2x1_2bit 모듈을 추가로 만들어 사용하였다.

- **MUX4x1.v**

```
module MUX4x1(input [1:0] in_bit,
    input [31:0] in3,
    input [31:0] in2,
    input [31:0] in1,
    input [31:0] in0,
    output [31:0] out);

    assign out = in_bit[1] ? (in_bit[0] ? in3 : in2) : (in_bit[0] ? in1 : in0);
endmodule
```

<그림 18: MUX4x1.v 모듈의 구성>

MUX4x1 모듈은 select 값에 따라 4 가지 (in0~in3) input 중 하나의 값을 반환하는 역할로, cpu.v 와 Address_select_logic 모듈에서 사용되었다. MUX4x1 이 사용된 부분은 다음과 같다.

- cpu.v 에서 사용된 부분
  - write_data_MUX: control 에 따라 ALU_result, MDR_wire, 그리고 ALUOut_wire 중 하나를 write_data 에 저장하게 된다.
  - ALU_in2_MUX: ALUSrcB signal 에 따라 immediate generator 를 통해 얻은 imm_gen_out, 4 (pc + 4 를 위한), 그리고 B_wire, 즉 rs2_out 값을 ALU 의 두 번째 input 값으로 가진다.
- Address_select_logic.v 에서 사용된 부분
  - next_state_MUX (2bit version): AddrCtrl signal 에 따라, ROM1wire/ROM2wire/add_state 을 next_state 의 값으로 가진다. 이 경우 2 bit 로 따로 처리할 필요가 있어, MUX4x1 을 32 bit 에서 2 bit 버전으로 변경한 MUX4x1_2bit 모듈을 추가로 만들어 사용하였다.

- 기타 define 들

```
// Need to implement total 22 instructions - can express in 5 bits
`define INS_JUMPLINK_JAL 5'b00000
`define INS_JUMPLINK_JALR 5'b00001
`define INS_BRANCH_BEQ 5'b00010
`define INS_BRANCH_BNE 5'b00011
`define INS_BRANCH_BLT 5'b00100
`define INS_BRANCH_BGE 5'b00101
`define INS_LOAD_LW 5'b00110
`define INS_STORE_SW 5'b00111
`define INS_ARITHMETIC_IMM_ADDI 5'b01000
`define INS_ARITHMETIC_IMM_XORI 5'b01001
`define INS_ARITHMETIC_IMM_ORI 5'b01010
`define INS_ARITHMETIC_IMM_ANDI 5'b01011
`define INS_ARITHMETIC_IMM_SLLI 5'b01100
`define INS_ARITHMETIC_IMM_SRLI 5'b01101
`define INS_ARITHMETIC_ADD 5'b01110
`define INS_ARITHMETIC_SUB 5'b01111
`define INS_ARITHMETIC_SLL 5'b10000
`define INS_ARITHMETIC_XOR 5'b10001
`define INS_ARITHMETIC_SRL  5'b10010
`define INS_ARITHMETIC_OR 5'b10011
`define INS_ARITHMETIC_AND 5'b10100
`define INS_ECALL 5'b10101

`define INS_ALU_ADDER 5'b10110
```

<그림 19: instructions.v 모듈>

```
// The constants below are from riscv spec
// Please do not change the values

// OPCODE
// R-type instruction opcodes
`define ARITHMETIC      7'b0110011
// I-type instruction opcodes
`define ARITHMETIC_IMM  7'b0010011
`define LOAD            7'b0000011
`define JALR            7'b1100111
// S-type instruction opcodes
`define STORE           7'b0100011
// B-type instruction opcodes
`define BRANCH          7'b1100011
// U-type instruction opcodes
//`define LUI             7'b0110111
//`define AUIPC           7'b0010111
// J-type instruction opcodes
`define JAL             7'b1101111


`define ECALL           7'b1110011

// FUNCT3
`define FUNCT3_BEQ      3'b000
`define FUNCT3_BNE      3'b001
`define FUNCT3_BLT      3'b100
`define FUNCT3_BGE      3'b101

`define FUNCT3_LW       3'b010
`define FUNCT3_SW       3'b010

`define FUNCT3_ADD      3'b000
`define FUNCT3_SUB      3'b000
`define FUNCT3_SLL      3'b001
`define FUNCT3_XOR      3'b100
`define FUNCT3_OR       3'b110
`define FUNCT3_AND      3'b111
`define FUNCT3_SRL      3'b101

// FUNCT7
`define FUNCT7_SUB      7'b0100000
`define FUNCT7_OTHERS   7'b0000000
```

<그림 20: opcodes.v 모듈>

```
//5 Stages :
`define IF 3'b000 //0
`define ID 3'b001 //1
`define EX 3'b010 //2
`define WB 3'b011 //3
`define MEM 3'b100 //4
`define B 3'b101 //5
```

<그림 21: Stages.v 모듈>

4. Discussion

Multi-cycle CPU 구현에서 중점을 두어야 할 부분 중 하나는 여러 연산을 처리하기 위해 설계된 모듈의 사용과 이 모듈들에서 요구되는 컨트롤 신호(signal)의 변화이다. 이러한 구현에서 각 연산은 여러 clock cycle 에 걸쳐 실행되며, 이는 다양한 stages(본 구현에서는 IF, ID, EX, WB, MEM, B)를 통하여 CPU 가 더 효율적으로 작동할 수 있도록 한다. 각 스테이지에서 컨트롤 신호가 어떻게 변화하는지 이해하자면 다음과 같다.

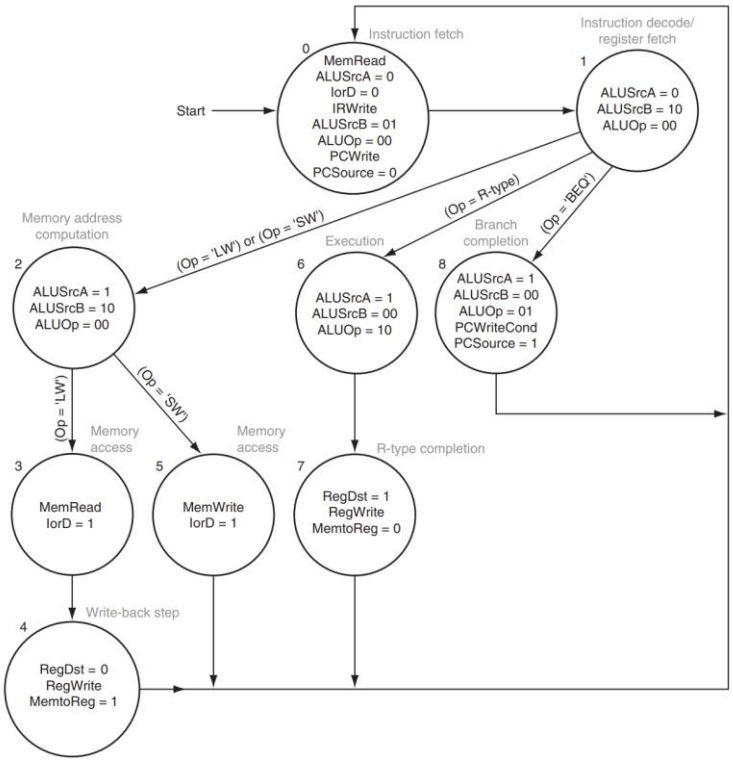
첫째로, Multi-cycle CPU 에서는 각 연산이 여러 단계를 거치게 되므로, 이에 따라 컨트롤 신호도 단계별로 변화해야 한다. 예를 들어, instruction 을 가져오는 IF stage 에서는 MemRead 시그널이 활성화되고, EX 단계에서는 수행해야 할 instruction 이 무엇인지에 따라(branch, arithmetic 등) 각기 다른 시그널을 활성화해야 한다.

둘째로, 각각의 컨트롤 신호는 해당 stage 에서만 유효해야 하며, 다음 stage 로 전환되면 새로운 컨트롤 신호가 발생해야 한다. 이는 CPU 가 각 stage 에서 수행해야 할 instruction 에 해당하는 동작을 정확하게 실행할 수 있도록 보장한다. 예를 들어, 데이터를 Memory 에서 읽어야 하는 stage 에서는 MemRead 시그널이 활성화되어야 하지만, 다음 stage 에서는 해당 신호가 비활성화되어야 한다. 이를 위해, always 문 내에서 각 시그널의 값을 모두 비활성화하고, 상황에 따라 활성화하는 방식으로 코드를 작성하였다.

셋째로, 컨트롤 신호의 정확한 타이밍과 순서는 CPU 의 성능과 효율성에 직접적인 영향을 미친다. 잘못된 컨트롤 신호나 타이밍은 연산 오류를 일으키거나 전반적인 성능을 저하시킬 수 있다. 따라서, 각 stage 에서 요구되는 컨트롤 신호를 정확히 설계하고, 이러한 신호들 사이의 관계를 정확히 고려하는 것이 중요하다.

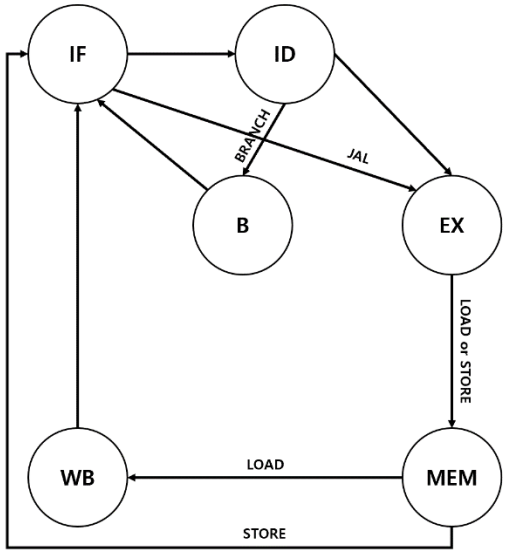
결론적으로, 멀티 사이클 CPU 구현에서는 여러 연산을 처리하는 모듈의 사용과 이 과정에서 각 stage 별로 변화하는 컨트롤 신호의 중요성을 고려해야 한다.

기본적으로 control unit 에서 다음 state 를 결정할 때는 이전에 서술한 바와 같이 address select logic 을 사용한다. 하지만, 초기 구상과 비교하였을 때 AddrCtrl 의 인덱스 순서가 바뀌어 Appendix C 의 구조와 좀 더 유사한 형태로 구현하였다. 다만 Appendix C 와 state 설계 자체가 다르기 때문에, 어느 정도 구현에 차이를 두었다. 아래의 그림은 Appendix C 에서 나와있는 state 설계이다.



<그림 22: Appendix C : State Diagram>

Appendix C 를 참고하여 본 과제에서 구현하고자 한 state 의 흐름을 나타낸 State Diagram(Mealy)을 아래의 그림에 첨부하였다. IF, ID, EX, B, MEM, WB 이다. 대부분 이전의 5 stages 로 구성된 것과 같지만, Branch instruction 과 관련된 수행 부분을 EX 가 아닌 B 로 따로 분리하고(실질적으로 EX_2 의 역할), 해당 stage 에 대한 시그널만 따로 활성화하여 branch compute 에 대한 경우만 개별적으로 수행할 수 있도록 하였다. 즉, 실제 기능적인 측면으로 구분하면, IF, ID, MEM, WB 에 각각 1 cycle, EX 에 2 cycle(EX 와 B)을 할당한 것으로 볼 수 있다.



<그림 23: State Diagram>

Resource reuse explanation

Multi-cycle CPU 구현은 기능 유닛이 서로 다른 clock cycle 에 사용되는 한 instruction 당 두 번 이상 사용될 수 있도록 한다. 이러한 재사용은 구현에 필요한 하드웨어의 양을 줄이는, 즉 비용을 절감하는 데 도움이 될 수 있다. 본 과제에서는 크게 세 가지 부분에서 Resource reuse 를 사용하였다.

1. Memory

   기존 Single-cycle CPU 에서는 Instruction memory 와 Data memory 를 따로 고려하여 unit 을 각각 만들어 준 반면, 본 과제에서 Multi-cycle CPU 를 구현할 때는 Memory 모듈 하나만을 사용하여 해당 모듈 내에서 asynchronous 하게 memory 에서 데이터를 read 하고, synchronous 하게 memory 에 data 를 write 하도록 Memory 유닛을 reuse 하였다.

2. ALU

   기존 Single-cycle CPU 에서는 다음 PC 의 값을 결정할 때 PC + 4 를 수행하는 adder 가 따로 존재하였다. 하지만 해당 덧셈 연산은 ALU 에서도 수행할 수 있다. 그렇기 때문에 ALU 유닛을 reuse 하고, 새로운 multiplexer 를 추가하여 원래의 Arithmetic 연산을 수행할지, PC 에 대한 연산을 수행할지 여부를 결정하도록 하였다. 이는 resource reuse 를 통해 하드웨어적인 성능 향상을 이루었다고 볼 수 있다.

3. 추가적인 레지스터

Multi-cycle CPU 구현 과정에서 ALU 유닛의 뒤에 ALUOut 레지스터를 추가하여 ALU 연산의 output 을 저장한다. 이는 해당 값이 다음 clock cycle 에서 사용될 때까지 ALU 유닛의 output 을 유지한다. 현재 clock cycle 이 끝나면 다음 clock cycle 에서 사용되는 모든 데이터를 상태 요소(레지스터 등)에 저장해야 한다. ALUOut 레지스터의 경우, ALU 에서 계산된 결과를 저장하므로, ALU 가 다음 연산을 위해 바로 재사용될 수 있도록 한다.

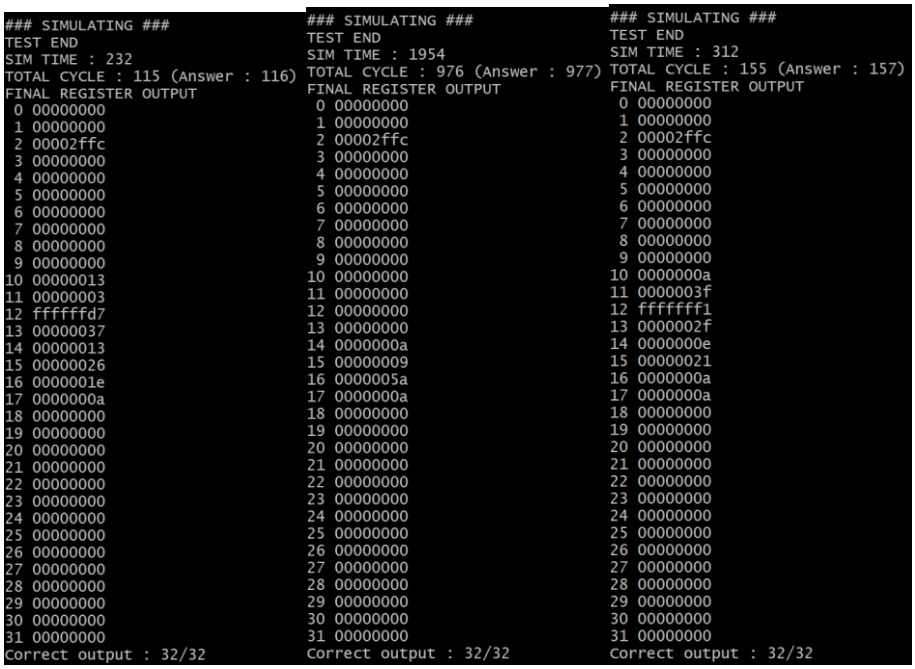
5. Conclusion

주어진 testbench 를 활용하여 basic_mem, loop_mem, non-controlflow_mem, ifelse_mem, recursive_mem 에 해당하는 5 개의 테스트를 수행하였다. 그 결과, 아래와 같이 올바른 register 값을 출력하는 것을 확인하였다.

그러나 cycle 수에는 정답과 차이가 존재하였다. 각 테스트케이스별로 사이클 수는 다음과 같다.

- basic_mem: 115
- loop_mem: 976
- non-controllflow_mem: 155
- ifelse_mem: 138
- recursive_mem: 3726

이러한 차이는 앞서 Stage 를 총 6 개 (IF, ID, EX, B, MEM, WB) 로 설정하였기 때문으로 판단된다. 실질적으로 B 는 Branch compute 에 대한 Execute stage 이나, 이를 따로 구분하여 새로운 stage 를 만든 것이기 때문에, 원래의 5 stage 로 가정하여 구성했을 때의 cycle 수와 차이가 존재한다. 때때로 answer cycle 수보다 값이 적게 나오는 경우는, 주어진 asm 코드에 따라 branch 연산을 수행하지 않는 경우도 있기 때문에, B stage 로 진입하지 않고 그냥 연산을 수행하는 경우도 있기 때문으로 보인다. 또는, Branch 연산을 수행하는 경우를 따로 분리함으로써, EX_1 과 EX_2 로 분리하여 2 개의 cycle 을 할당한 것에 비해, 상황에 따라 1 개의 cycle 을 수행할 수 있으므로 cycle 수가 줄어든 것으로 보인다.



<그림 24, 25, 26, 27, 28: Simulation 결과>