컴퓨터 구조 Lab3 Multi-Cycle CPU

20210207 이지현
20210794 정유진

1. Introduction

1.1. Lab 3는 Vivado를 이용하여 multi-cycle RISC-V CPU를 구현하는 것을 목표로 한다.

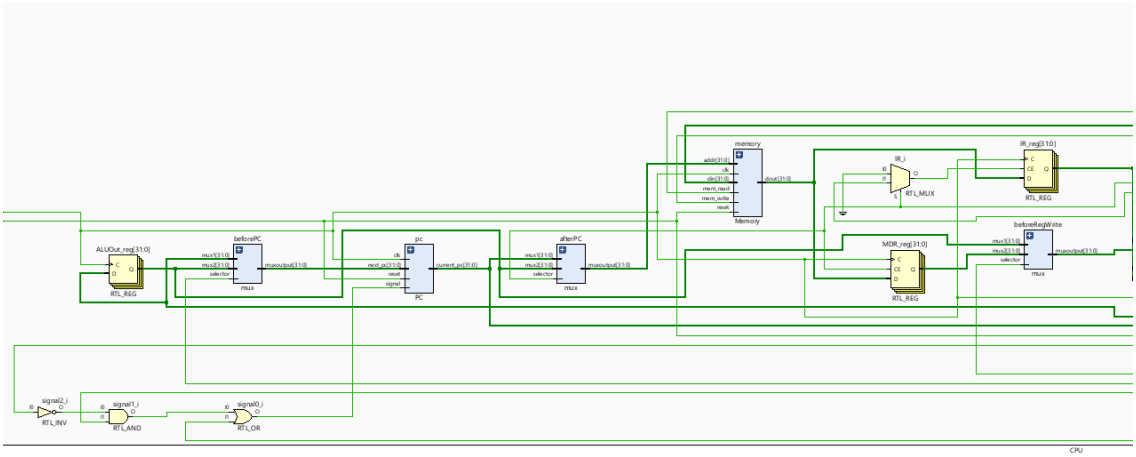
1.2. CPU의 모든 implementation은 한 사이클에 한 **1 stage (ALU, memory 등이 전부 1 cycle에 처리되도록) 가 프로세스되도록 한다.**

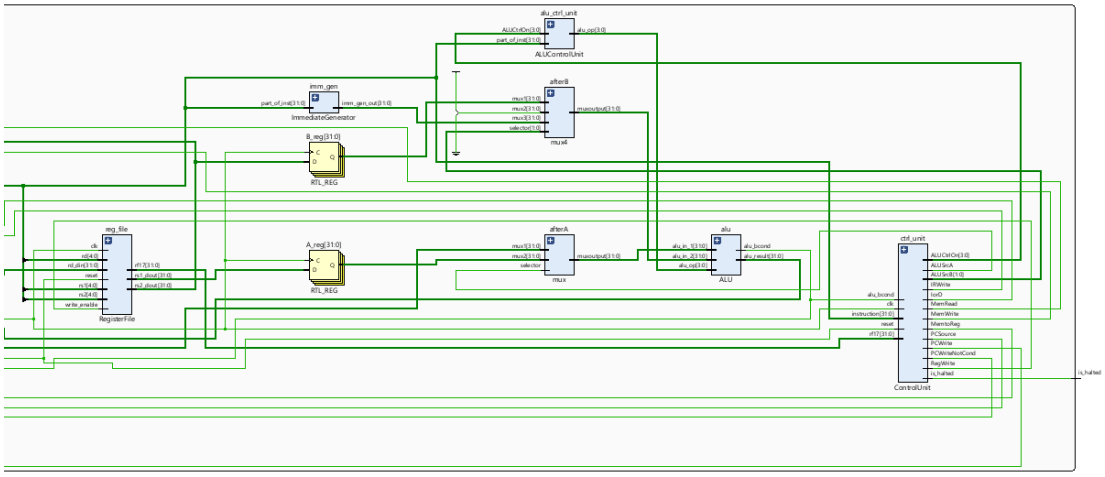
1.3. 주어진 스켈레톤 코드는 top.v, Memory.v, RegisterFile.v, cpu.v이 있고 추가로 만든 V 파일은 ALU.v, ALUControlUnit.v, ImmediateGenerator.v, ControlUnit.v, PC.v, opcodes.v, mux.v, mux4.v가 있다. (이 때 ALU, ImmediateGenerator, opcodes, mux는 single cycle cpu에서 사용한 파일을 그대로 재사용하였다)

1.4. 테스트 코드는 총 3가지로 1. basic 2. If-else 3. Loop 4. Non-controlflow 5. recursive가 있으며, Ripes를 통해 얻은 레지스터 값과 베릴로그 상에서 얻은 레지스터 값과 비교하며 정상적인 동작을 확인할 수 있다.

1.5. Multi cycle CPU의 design과 implementation을 설명할 때 각각의 모듈이 synchronous인지 asynchronous인지 확인하며 각각의 스테이지를 설명하고자 한다.

2. Design



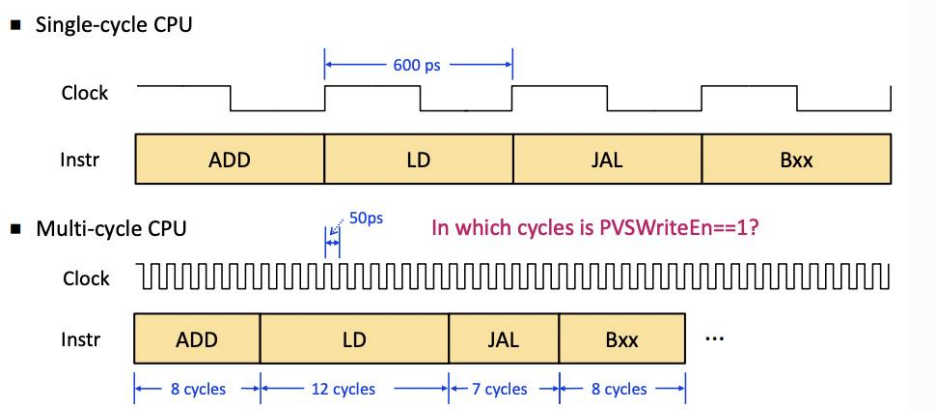


2.1. CPU 동작이 이루어지는 과정에 대한 순차적인 설명은 다음과 같다.

- Instruction processing은 보통 5가지 절차로 이루어진다.
- 이 과정은 Multicylce cpu에서는 필요한 stage만 선택적으로 이루어진다.
- IF : instruction fetch로, 인스트럭션 실행을 위해선 메모리로부터 읽어와야한다.
  - Instruction memory, PC 가 주로 수행한다.
- ID : instruction decode로, 레지스터 파일에서 레지스터 값을 읽어오며 (operand fetch) instruction에 따른 control signal, immedgate value를 발생시킨다.
  - ControlUnit, immediateGenerator, RegisterFile 가 주로 수행한다.
- EX : execution으로, ALU 등을 통한 연산이 이루어진다.
  - ALU, ALUControlUnit 가 주로 수행한다.
- MEM : data memory access로, instruction (load/store)에서 접근하고자 하는 메모리를 읽어온다.
  - Data memory 가 주로 수행한다.
- WB : write back으로, 레지스터 파일을 업데이트 하고 메모리에서 읽은 값을 rd에 저장하는 등의 과정이 이루어진다.
  - RegisterFile 가 주로 수행한다.

2.2. Single Cycle Cpu와 Multi cycle cpu의 다른점에 대해 설명하면 다음과 같다.

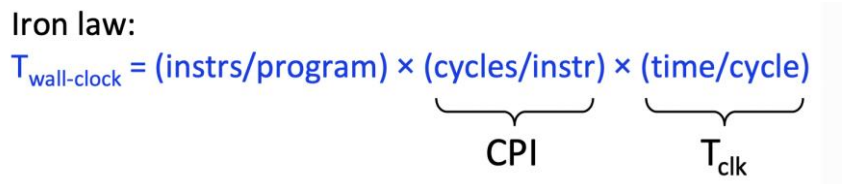
- Single cycle cpu 디자인에서는 clock이 600ps 였는데, multi cycle cpu는 50ps(더 적은 사이클) clock을 가진다고 하면 각각의 instruction은 multiple cycle을 필요로 한다. architecture & Programmer visible state update는 오직 last cycle에서 실행된다.
- instruction의 last cycle임을 알려주는 signal = "PVSWriteEn" (pvs write enable) 신호가 0일 때에는 architecture state가 변화하지 않는다



- single cycle cpu에서는 모든 inustrction에서 600ps(=1clk)가 걸렸지만 Multi cycle cpu에서는 50clk을 가지면서 각각의 instruction이 다른 clk이 걸린다
- 따라서 더 짧은 시간이 걸리고 총 Runtime도 빨라진다.

2.3. 왜 Multi-cycle CPU가 더 좋은지에 대해 설명하면 다음과 같다.

- 위에서 설명했듯이 Runtime이 더 빨라지고 성능이 더 좋아지는데, Iron Law로 구체화할 수 있다.



-
- Instruction/cycle이 고정되어있다고 가정했을 때, CPI * Tclk만 보면 되고 그 역인 MIPS을 통해 성능을 따질 수 있다.
- single cycle 에서는 교과서 기준으로 IPC = 1, f_clk = 1/600ps 이었다.
  - MIPS = 1 * 1667MHz = 1667 MIPs (in Single cycle CPU)
- multi cycle 에서는 교과서 기준으로 f_clk = 1/50ps (=20000MHz) 이므로
  - MIPS = IPC * 20000MHz 로 계산할 수 있다
  - 각각의 instruction 비율을 가정했을 때 (25% LD,10% SD,45% ALU,15% Bxx, 3% JAL ,2% JALR) 총 CPI_avg = Weighted arithmetic mea of CPI = 0.25x12 + ... + 0.02x8 = 9.12 이다.
- 위의 계산만 놓고 보면 2193/1667 = 31.6% 스피드 업 했음을 알 수 있다.
- CPU마다 다르겠지만, 보편적으로 이런 방식으로 multi-cycle cpu 성능이 single-cycle cpu보다 좋음을 알 수 있다.
- **즉, 각각의 instruction마다 필요로 하는 cycle만 소비하고 기다리지 않아도 되므로 latency가 짧아지고 성능은 높아질 것이다.**

2.4. 다음은 각각의 모듈에 대한 **synchronous/asynchronous 디자인의** 포괄적인 설명이다.

Cpu	Top module로서 sub module들의 인스턴스 선언과 wire 변수들의 연결이 이루어지는 곳이다.

ALU	ALU 연산을 한다. ALU operation의 종류와, ALU 연산을 할 input 2개를 asynchronous하게 읽어와서, 연산을 할 result와, branch operation일 경우에는 branch condition에 해당하는 지를 리턴한다.
ALUControlUnit	Instruction를 해독하여 ALU operation을 내보내주는 역할을 한다. Instruction을 asynchronous하게 읽어온다.
ControlUnit	Synchronous하게 매 clk마다 microPC의 값을 업데이트하여, 다음에 올 state를 전달한다. microPC가 바뀜에 따라, Asynchronous하게 control output와 다음 microPC를 어떻게 업데이트 할 지의 정보를(AddrCtl 레지스터에 저장)
Memory	Instruction memory와 data memory가 하나의 메모리 모듈에서 모두 이루어지며, fetch instruction과 access메모리 모두 하나의 모듈이 reuse된다. Mem_read signal이 1이면 data memory output으로 메모리값 asnychronous하게을 내보내고, mem_write가 1이라면 synchronous하게 data를 memory에 write한다. CPU Reset시 synchronous하게 초기화가 이루어진다.
Mux	selector가 0이면 output으로 mux1을 assign하고, 아니면 output으로 mux2를 assign한다. Dataflow modeling이다.
Mux4	Selector가 00이면 output으로 mux1을 assign하고, 01이면 mux2를 assign하고, 10이면 mux3를 assign한다. Dataflow modeling이다.
Opcodes	연산들의 funct3과 opcode를 선언해 놓았다.
PC	Synchronous하게 매 clk마다 current_pc를 next_pc의 값으로 업데이트 해준다.
RegisterFile	register file의 rs1 rs2 값을 asynchronous하게 읽어오고 write_enable = 1이고 rd가 0이 아니라면 synchronous하게 rd에 write한다. CPU Reset시 synchronous하게 초기화가 이루어진다.
Top	CPU instance를 선언하고, 처음에 reset을 한 뒤, clk를 특정 시간 간격마다 -clk로 바꾸어주어 synchronous한 동작에 도움을 준다. 프로그램이 끝나면, total_cycle과 register의 상황을 출력해 준다.

2.5. 같은 Resource를 reuse하는 부분에 대한 디자인 설명은 다음과 같다.

- ALU

- SIngle cycle cpu에서 ALU가 있음에도 여러개의 Adder를 지닌 이유는, 그들이 병렬적으로 작동했었기 때문이다
- 하지만 지금은 single instruction에 multiple cycle이 지나가기에 같은 ALU를 다른 행동에 모두 사용 가능하다.
- PC+4에 사용하던 adder와 PCSrc앞의 adder를 없애고 하나의 ALU만으로 reuse할

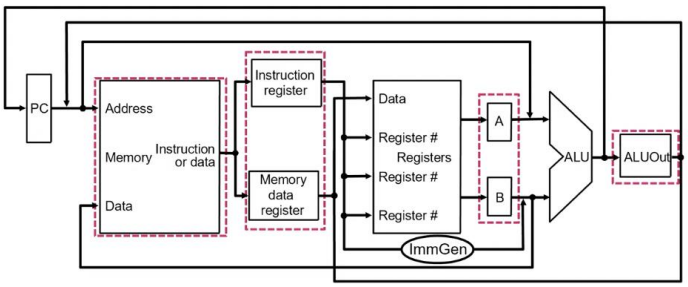
수 있다.

- 이들의 사용 시간의 conflict를 막고 ALU를 유동적으로 사용하기 위해서 IR (instruction register), MDR (memory data register), A, B, ALUout 레지스터를 사용한다.

- Memory

- Instruction memory와 data memory가 같은 Memory module을 reuse한다.
- 같은 memory를 사용하기 위해 instruction Register를 만들어준다.
- memory fetch 이후에 instruction data bit 를 instruction register 에 저장해 놓고, 그 다음 같은 memory 를 data excess 를 위해 사용한다.

Removing Redundancies



We can also use single memory by creating **timing differences** with proper controls
(It is also possible to even reduce to a single register read-write port)

2.6. Microcode controller state design 구현과 디자인에 대한 설명은 다음과 같다.

- (microcode controller, control unit code 스크린샷)

```
`define IF 4'b0000 // 0
`define ID 4'b0001 // 1
`define MEMADDRCOMPUTE 4'b0010 // 2
`define EXER 4'b0101 // 5
`define EXEI 4'b0110 // 6
`define BRANCHCOMPUTE 4'b1001 // 9
`define BRANCHSELECT 4'b1010 // 10
`define MEMREAD 4'b0011 // 3
`define MEMWRITE 4'b1000 // 8
`define WBALU 4'b0111 // 7
`define WB 4'b0100 // 4
`define WBJAL 4'b1011 // 11
`define WBJALR 4'b1100 // 12
`define ECALLCOMPUTE 4'b1101 // 13

`include "opcodes.v"
```

우선, 각 STATE를 미리 선언한다.

```
input [31:0] instruction;  // input
input [31:0] rf17;
input reset;
input clk;

output reg RegWrite; // 레지스터 쓰기 가능
output reg MemRead;      // output 메모리 읽기 가능
output reg MemWrite;    // output register 읽기 가능
output reg MemtoReg;     // output 메모리 쓰기 가능

output reg is_ecall;
output reg is_halted;

output reg ALUSrcA;
output reg IorD;
output reg IRWrite;
output reg PCSource;
output reg PCWrite;
output reg PCWriteNotCond;
output reg [1:0] ALUSrcB;

output reg [3:0] ALUCtrlOn;

input alu_bcond;
```

input은 instruction, rf17, reset, clk이고, 컨트롤 시그널들을 output으로 한다.

```
reg [6:0] opcode;
reg [3:0] microPC;
reg [3:0] AddrCtl;

reg [3:0] DispatchROM2[0:7];
reg [3:0] DispatchROM3[0:2];
reg [3:0] DispatchROM4;
reg [3:0] DispatchROM5;
```

opcode는 DispatchROM 테이블에서 state를 판단할 때 쓰이고, microPC는 state를 나타낸다. AddrCtl은 microPC가 어떤 테이블에서 결정될 지를 나타낸다.

```
always @(posedge clk) begin
    opcode = instruction[6:0];
    if(reset) begin
        microPC = `IF;
        DispatchROM2[0] = `MEMADDRCOMPUTE;
        DispatchROM2[1] = `EXER;
        DispatchROM2[2] = `EXEI;
        DispatchROM2[3] = `BRANCHCOMPUTE;
        DispatchROM2[4] = `WBJAL;
        DispatchROM2[5] = `WBJALR;
        DispatchROM2[6] = `ECALLCOMPUTE;

        DispatchROM3[0] = `MEMREAD;
        DispatchROM3[1] = `MEMWRITE;

        DispatchROM4 = `BRANCHSELECT;
        DispatchROM5 = `WBALU;
    end
```

우선, Synchronous Part에서, reset이 되었을 때, state를 IF로 두고, DispatchROM 테이블들을 선언해준다.

```
else begin
    case(AddrCtl)
        0: begin
            microPC = `IF;
        end
        1 : begin
            microPC = microPC + 1;
        end
```

만약, AddrCtl이 0이면, state는 IF가 된다. 1이면, state는 원래 값에서 1이 더해진다.

```
        2 : begin
            case(opcode)
                `LOAD: begin
                    microPC = DispatchROM2[0];
                end
                `STORE : begin
                    microPC = DispatchROM2[0];
                end
                `ARITHMETIC : begin
                    microPC = DispatchROM2[1];
                end
                `ARITHMETIC_IMM : begin
                    microPC = DispatchROM2[2];
                end
                `BRANCH : begin
                    microPC = DispatchROM2[3];
                end
                `JAL : begin
                    microPC = DispatchROM2[4];
                end
                `JALR : begin
                    microPC = DispatchROM2[5];
                end
                `ECALL : begin
                    microPC = DispatchROM2[6];
                end
            endcase
```

AddrCtl이 2인 경우는, ID 연산 직후에 AddrCtl이 2가 된다. ID 연산 이후에, opcode에 따라 다음 microPC(state)가 무엇이 될 지를 매핑하고 있다.

```
                end
                3 : begin
                    if(opcode == `LOAD)
                        microPC = DIspatchROM3[0];
                    if(opcode == `STORE)
                        microPC = DIspatchROM3[1];
                end
                4 : begin
                   microPC = DispatchROM4;
                end
                5 : begin
                    microPC = DispatchROM5;
                end
            endcase
        end
    end
```

AddrCtl이 3인 경우는, Memory Address Computation 이후에 opcode에 따라 다음 microPC가 MEM 부분의 load일지, write일지를 매핑하고 있다.

AddrCtl이 4인 경우는 Branch가 select 되었을 때의 이동으로, 나머지 AddrCtl로 커버할 수 없는 부분이라 새 테이블을 만들었다고 보면 된다.

AddrCtl이 5일 경우는 ALU(I-type과 R-type)을 한 뒤, write back 할 때의 WB stage로, 역시 나머지 AddrCtl로 커버할 수 없어 새 테이블을 만들었다.

다음은 Asynchronous한 부분에 대한 설명이다. microPC가 앞 synchronous part에 의해 변했을 때, 실행된다.

```
always @(*) begin
    RegWrite = 0;
    MemRead = 0;
    MemWrite = 0;
    MemtoReg = 0;
    is_ecall = 0;
    is_halted = 0;
    ALUSrcA = 0;
    IRWrite = 0;
    PCSource = 0;
    PCWrite = 0;
    PCWriteNotCond = 0;
    ALUSrcB = 0;
   ALUCtrlOn = 0;

    case(microPC)
```

우선, 컨트롤 시그널을 초기화해준다.

```
`IF : begin
    IorD = 0;
    MemRead = 1;
    IRWrite = 1;
    AddrCtl = 1;
end
`ID : begin
    ALUSrcA = 0; // PC를 더해줘서 ALUOut에 저장해 놓는다
    ALUSrcB = 1; // 4이다.
    ALUCtrlOn = 3'b001; // ALUCtrlOn이 1이면 PC+4 연산을 하는 것이다.
    AddrCtl = 2;
end
```

IF stage와 ID stage는 모든 instruction 공통으로, IF stage에서는 instruction fetch를 위해 IorD = 0(메모리에서 Instruction을 읽어옴), MemRead = 1(메모리를 읽어옴), IRWrite = 1(IR에 instruction을 씀)이고, 다음 state는 0000에서 0001로 넘어가므로 AddrCtl = 1이다.

ID stage는 PC = PC + 4를 하기 위해, ALUSrcA = 0(PC taken), ALUSrcB = 1(4 taken), ALUCtrlOn = 1(Adder)를 한다. PC = PC + 4를 하는 이유는, 다음에 올 수도 있는 Branch Not taken, JAL, JALR instruction에 필요하기 때문이다. 그 뒤, AddrCtl = 2로 하여, opcode에 따라 DispatchROM2 테이블을 확인한다.

```
`ECALLCOMPUTE : begin
    is_ecall = 1;
    if(rf17 == 10) begin
        is_halted = 1;
    end
    ALUSrcA = 0;
    ALUSrcB = 1;
    ALUCtrlOn = 3'b001;
    PCSource = 0;
    PCWrite = 1; // PC = PC + 4
    AddrCtl = 0; // go to IF stage
end
```

ID stage 직후 microPC이다. ECALL instruction일 경우, is_ecall = 1로 하고, 만약 rf17이 10이면 is_halted = 1이다. 아니라면, PC = PC + 4를 하고, ALU의 결과값을 바로 PCSource로 쓴다. AddrCtl = 0으로 하여 다시 IF stage로 돌아간다.

```
`MEMADDRCOMPUTE: begin
    ALUSrcA =1;
    ALUSrcB = 2;
    ALUCtrlOn = 1; // 주소 구하기
    AddrCtl = 3;
end
```

ID stage 직후 microPC이다. Load나 Store instruction일 경우, microPC가 MEMADDRCOMPUTE가 된다. 이때, 메모리의 주소는 rs1 + immediate이므로, 그 연산을 해 주기 위한 ALUSrcA, B, ALUCtrOn 신호를 만들어준다. AddrCtl = 3으로 하여, DispatchROM3 테이블을 확인하여 다음 microPC를 확인하게 된다.

```
`EXER : begin
    ALUSrcA = 1;
    ALUSrcB = 0;
    ALUCtrlOn = 2;
    AddrCtl = 5;
end
`EXEI : begin
    ALUSrcA = 1;
    ALUSrcB = 2;
    ALUCtrlOn = 2;
    AddrCtl = 5;
end
```

ID stage 직후 microPC이다. R-Type이나, I-Type ALU에 맞는 신호를 만들어준다. ALUCtrlOn = 2인 이유는, 단순 Adder가 아니라 복잡한 연산을 필요로 하기 때문이다. AddrCtl = 5로 하여, DispatchROM5 테이블(next microPC는 WBALU가 된다)을 가리킨다.

```
`BRANCHCOMPUTE: begin
            //branch 명령어일 경우 다음 PC는 BEQ rs1, rs2, imm13
            ALUSrcA = 1; // rs1
            ALUSrcB = 0; // rs2
            ALUCtrlOn = 2; // branch는 복잡한 연산
            //bcond가 나왔다. bcond가 0이면 PC = PC+4로 업데이트 해야 한다.
            PCSource = 1; // PC+4가 들어가 있는 위치 : ALUout
            PCWriteNotCond = 1; // bcond가 1이고 이것도 1이어야 PC가 브랜치 되는 것이허용된다.
            if(alu_bcond)
                AddrCtl = 4;
            else
                AddrCtl = 0;
 end
```

ID stage 직후 microPC이다. rs1, rs2를 비교하는 ALU를 하게 되면, alu_bcond가 나오게 된다. alu_bcond가 0일 경우에, 이 stage에서 PC = PC+4로 업데이트 되기 위하여 PCSource = 1이 된다. (PC가 ALUOut 레지스터의 값으로 된다는 의미인데, 이전 스테이지에서 PC = PC+4를 계산했으므로 ALUOut에 PC+4가 저장되어 있기 때문이 이를 쓰면 된다)

alu_bcond가 1이면, next microPC는 BRANCHSELECT가 되게 하여 그 스테이지에서 이동할 PC를 계산하게 되고, 0이면 IF stage로 돌아간다.

```
`BRANCHSELECT : begin
    ALUSrcA = 0; // 이건 bcond가 1일 경우이다.
    ALUSrcB = 2;
    ALUCtrlOn = 1;
    PCSource = 0;
    PCWrite = 1;
    AddrCtl = 0;
end
```

BRANCHCOMPUTE microPC 직후이다. PC + immediate연산을 하는 신호를 만들어 ALU 쪽으로 보내고, PCSource = 0으로 하여 ALU에서 나온 값을 바로 PC로 활용하게끔 한다. AddrCtl = 0이 되어 다시 IF stage로 가게 된다.

```
`MEMREAD: begin
    MemRead = 1;
    IorD = 1; // ALUOut가 Data address를 alu_bcond
    AddrCtl = 1;
 end
```

MEMADDRCOMPUTE microPC 직후이다. MemRead = 1로 하여 메모리를 읽어오고, IorD = 1로 하여 메모리 중 데이터를 읽어오게 된다. `define 선언상으로 MEMREAD = 3이고, WB = 4이므로 AddrCtl = 1로 하여 microPC가 WB가 되게끔 한다.

```
`MEMWRITE: begin
    MemWrite = 1;
    IorD = 1;
    ALUSrcA = 0;
    ALUSrcB = 1;
    ALUCtrlOn = 1;
    PCSource = 0;
    PCWrite = 1;
    AddrCtl = 0;
end
```

MEMADDRCOMPUTE microPC 직후이다. MemWrite = 1, IorD = 1로 하여 Data memory에 쓰게 된다. ALU 쪽으로 PC = PC + 4를 하는 신호를 보내주고, 그 값을 바로 PC에 쓰게끔 PCSource = 0으로 한다. AddrCtl = 0으로 하여 다시 IF Stage로 가게 된다.

```
`WBALU : begin
    RegWrite = 1;
    ALUSrcA = 0;
    ALUSrcB = 1;
    ALUCtrlOn = 1;
    PCSource = 0;
    PCWrite = 1;
    AddrCtl = 0;
end
```

EXER, EXEI microPC 직후이다. 레지스터에 값을 쓰기 위해 RegWrite = 1로 하고, PC = PC+4 연산을 하여 PC에 쓴다. AddrCtl = 0으로 하여 IF stage로 가게 된다.

```
`WB : begin
    RegWrite = 1;
    MemtoReg = 1;
    ALUSrcA = 0;
    ALUSrcB = 1;
    ALUCtrlOn = 1;
    PCSource = 0;
    PCWrite = 1;
    AddrCtl = 0;
end
```

MEMREAD microPC 직후이다. 레지스터에 값을 쓰고, Memory의 값이므로, MemtoReg = 1로 둔다. WBALU와 마찬가지로, PC = PC + 4를 하여 바로 PC에 쓰고, 다시 IF stage로 가게 된다.

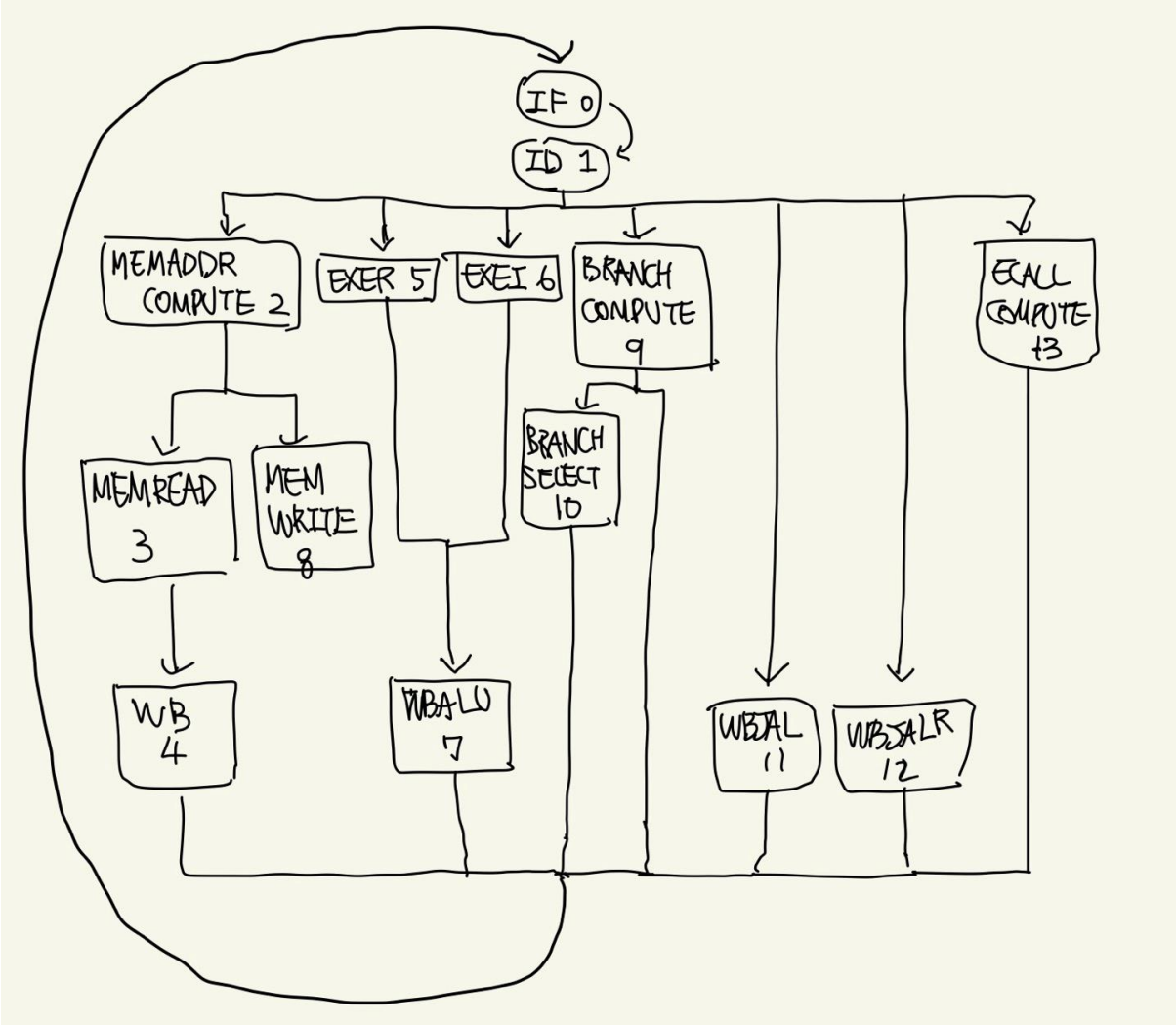
```
`WBJAL: begin
    RegWrite = 1;
    ALUSrcA = 0;
    ALUSrcB = 2;
    ALUCtrlOn = 1;
    PCSource = 0;
    PCWrite = 1;
    AddrCtl = 0;
end
```

ID microPC 직후이다. 레지스터에 값을 쓰고(JAL 연산일 경우에 PC+4를 쓴다), PC = PC + immediate를 하여 바로 PC에 쓰고, 다시 IF stage로 가게 된다.

```
`WBJALR : begin
    RegWrite = 1;
    ALUSrcA = 1;
    ALUSrcB = 2;
    ALUCtrlOn = 1;
    PCSource = 0;
    PCWrite = 1;
    AddrCtl = 0; // go to IF
end
```

ID microPC 직후이다. 레지스터에 값을 쓰고, PC = rs1 + immediate를 하여 바로 PC에 쓰고, 다시 IF stage로 가게 된다.

microPC의 흐름에 관한 그림은 다음과 같다.



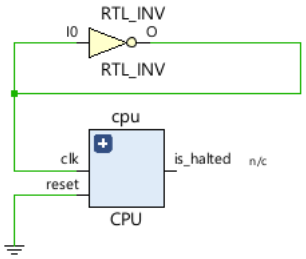
2.7. 각각의 test bench에서 걸린 총 cycle 수는 다음과 같다.

- Basic_mem = 117 cycles
- Ifelse_mem = 140 cycles

- Loop_mem = 978 cycles
- Non-controlflow = 158 cycles
- Recursive = 3687 cycles

3. Implementation

3.1. <cpu.v> = top module



Input : clk, reset

Output : is_halted

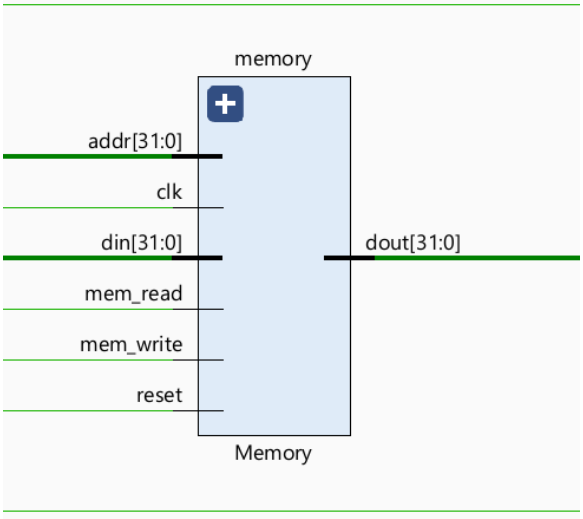
사용한 wire 변수들의 설명은 다음과 같다.

PC	nextPC : 다음 clock의 PC currentPC : 현재 PC real_currentPC : mux를 거치고 난 현재 PC currentPCplus4 : 현재 PC + 4값
ALU	alu_op : ALU operation의 종류로 "opcodes.v"에 선언되어 있다. alu_bcond : ALU를 통해 계산된 branch 여부 ALU_out : ALU output ALUCitrlOn : 1이면 ALU는 단순 Adder(PC 더하기 등)로 활용되고, 2이면 복잡한 연산에 쓰인다.
register	writeRegister : RD real_RegWriteData : Register Write Data RegReadData1 : RS1 RegReadData2 : RS2 MemData : memory data rf17 : Regster file [17] <- 레지스터파일 사용X MDR : memory data register IR : instruction register realA : mux 거친 뒤 A register realB : mux 거친 뒤 B register ALUOut : ALU output register
memory	real_DataOutput : mux를 거치고 난 data output DataReadData : mux를 거치기 전 data output
Immediate generator & mux step	Imm_gen_out : immediate generator를 통해 만들어진 immediate
Control unit	RegWrite : 레지스터에 데이터를 쓸 때의 컨트롤 시그널

	MemRead : 메모리를 읽을 때의 컨트롤 시그널 MemToReg : 메모리의 값을 레지스터로 보낼 때의 컨트롤 시그널 MemWrite : 메모리에 값을 쓸 때의 컨트롤 시그널 ALUSrcA : ALU의 첫번째 input으로 PC의 값이 들어갈 지, register의 값이 들어갈 지를 구분하는 컨트롤 시그널 [1:0] ALUSrcB : ALU의 두번째 input으로 register의 값, 4, immediate 중 어떤 값이 들어갈 지를 구분하는 컨트롤 시그널 PCSource : PC의 값이 ALU의 result에서 바로 가는지, ALUOut register에서 가는지를 구분하는 컨트롤 시그널 PCWrite : PC를 업데이트 해줄 때의 컨트롤 시그널 PCWriteNotCond : bcond가 0일 때, PC = PC+4를 해주기 위하여, PCWriteNotCond가 1이고, bcond가 0일 때만 branch instruction의 분기 안 함으로 판별하여 PC를 바꾸어 주기 위한 시그널 IRWrite : 인스트럭션 레지스터에 값을 쓸 때의 컨트롤 시그널 IorD : 메모리에서 Instruction을 받아오는지, Data를 받아오는지를 구분하는 컨트롤 시그널 Is_ecall : ecall instruction을 호출했을 때의 컨트롤 시그널 Is_halted : 종료 시 컨트롤 시그널

3.2. <Memory.v>

- Memory

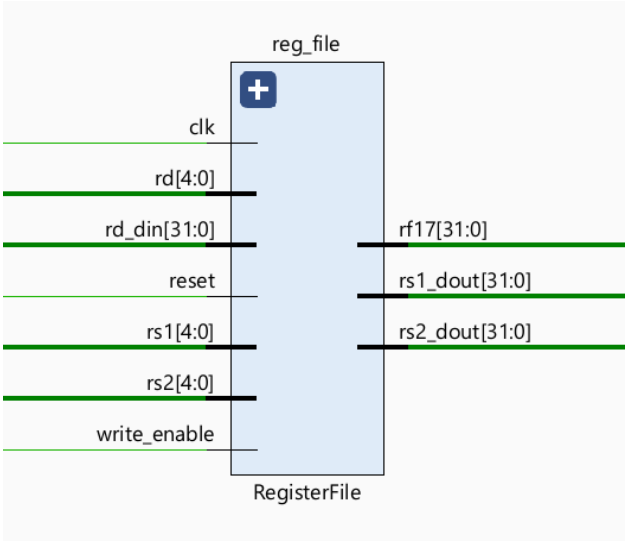


Input : reset, clk, addr, din, mem_read, mem_write

Output : dout

- Asynchronous logic : mem_addr를 이용해 memory에 access해 dout에 담는 과정을 mem_read가 1일 때 asynchronous하게 진행하였다.
- (Positive clk) Synchronous logic : reset버튼이 눌러졌을 때 instruction memory를 0으로 초기화하는 과정은 clk이 positive일 때마다 synchronous하게 진행하였으며, pc값이 돌아올때마다 각각의 테스트 코드에 있는 memory 경로의 값을 읽어오는 과정도 clk에 따라 진행되었다. Mem_write = 1일 때 memory에 din을 write하는 과정도 synchronous하게 진행되었다.

3.3. <RegisterFile.v >



Input : reset, clk, rs1, rs2, rd, rd_din, write_enable
Output : rs1_dout, rs2_dout

- Asynchronous logic : rf[rs1], rf[rs2]값이 변할 때마다 rs1_dout, rs2_dout에 assign문을 통해 대입해주는 과정은 asynchronous하다.
- (Positive clk) Synchronous logic : positive clk일 때마다 rf[0]의 값은 변하면 안되므로 rd가 0이 아니고 write enable이 1인지를 확인해 rd_din을 레지스터 파일에 대입해주는 과정은 synchronous하다. 마찬가지로 reset이 1일 때 레지스터 파일을 0으로 초기화해주는 과정또한 synchrounous하다.

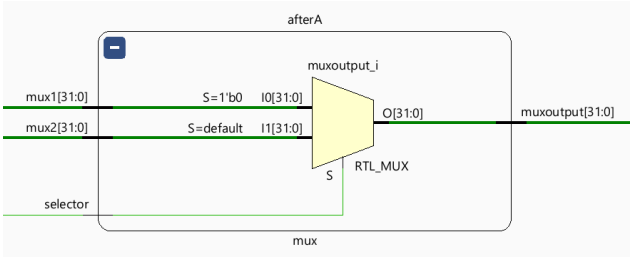
- 단, 이때 registerfile.v를 변경할 수 없으므로, rf17은 ECALL이 1일 때 전 instruction에 담긴 레지스터값을 읽어오는 것으로 해결할 수 있었다.

3.4. <mux.v>

mux1과 mux2와 selector를 input으로 받아서 selector가 0이면 mux1을, 1이면 mux2를 출력해주는 모듈이다. mux module을 사용하여 instance를 만든 부분들은

다음과 같다.

- afterA

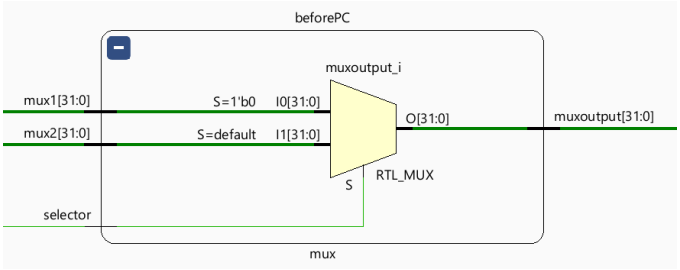


Input : mux1 (currentPC), mux2 (A), selector(ALUSrcA)

Output : muxoutput(realA)

  - Asynchronous logic : clk에 의존적인 Logic이나 intial을 이용하는 구간이 없고 assign문을 이용했으므로 asynchronous한 로직만 있다. currentPC와 A 사이에서 ALUSrcA signal을 통해 한 값만 내보내주는 조건문 역할을 해준다.

- beforePC

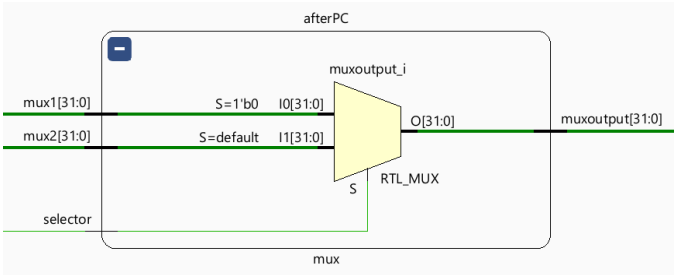


Input : mux1 (ALU_out), mux2 (ALUOut), selector(PCSource)

Output : muxoutput(nextPC)

  - Asynchronous logic : clk에 의존적인 Logic이나 intial을 이용하는 구간이 없고 assign문을 이용했으므로 asynchronous한 로직만 있다. ALU_out과 ALUOut 사이에서 PCSource signal을 통해 한 값만 내보내주는 조건문 역할을 해준다.

- afterPC



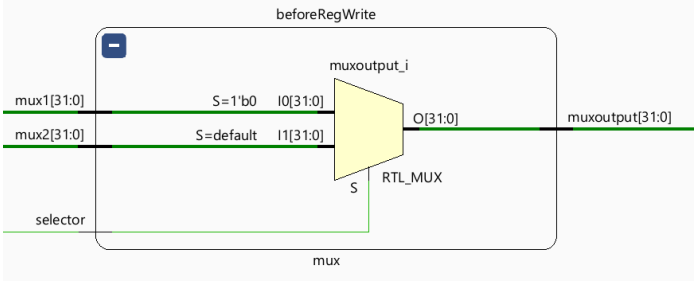
Input : mux1 (currentPC), mux2 (ALUOut), selector(IorD)

Output : muxoutput(real_currentPC)

  - Asynchronous logic : clk에 의존적인 Logic이나 intial을 이용하는 구간이 없고

assign문을 이용했으므로 asynchronous한 로직만 있다. currentPC과 ALUOut 사이에서 IorD selector를 통해 한 값만 내보내주는 조건문 역할을 한다.

- beforeRegWrite



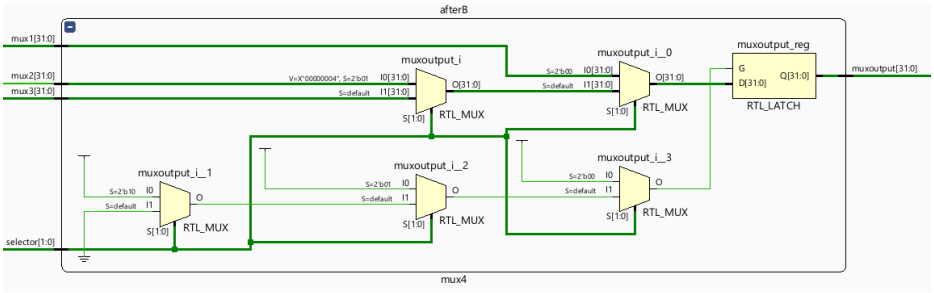
Input : mux1 (ALU_out), mux2 (MDR), selector(memToReg)
Output : muxoutput(real_RegWriteData)

- Asynchronous logic : clk에 의존적인 Logic이나 intial을 이용하는 구간이 없고 assign문을 이용했으므로 asynchronous한 로직만 있다. ALU_out과 MDR중에서 memToReg signal에 따라 한 값만 내보내주는 조건문 역할을 한다.

3.5. <mux4.v>

mux1과 mux2와 mux3과 selector를 input으로 받아서 selector가 00이면 mux1을, 01이면 mux2를 10이면 mux3를 출력해주는 모듈이다. mux module을 사용하여 instance를 만든 부분들은 다음과 같다.

- afterB

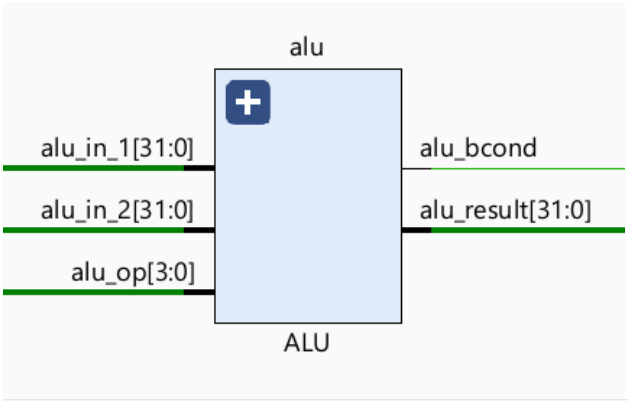


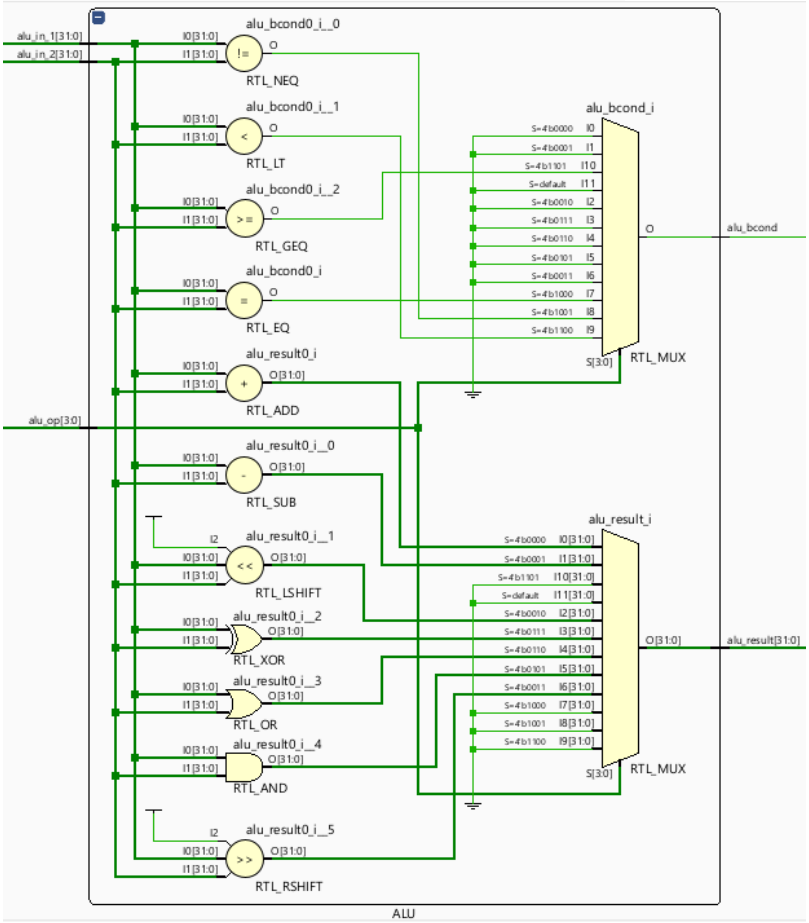
Input : mux1 (B), mux2 (4), mux3(realB), selector(ALUSrcB)
Output : muxoutput(realB)

- Asynchronous logic : clk에 의존적인 Logic이나 intial을 이용하는 구간이 없고 assign문을 이용했으므로 asynchronous한 로직만 있다. B와 4와 realB 중에서 ALUSrcB signal에 따라 한 값만 내보내주는 조건문 역할을 한다.

3.6. <ALU.v>

- ALU



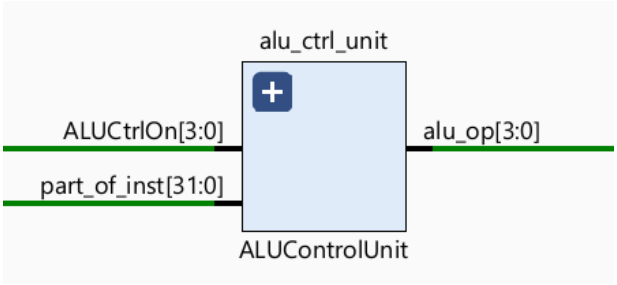


input : alu_in_1, alu_in_2, alu_op
output : alu_bcond, alu_result

- Asynchronous Logic : switch-case문을 이용해 alu_op 4비트에 따라 alu_result의 연산이 달라지도록 구현하였다. 특히 branch instruction의 경우 op 4비트 MSB가 1인 특징을 지니며, BEQ BNE BLT, BGE 의 조건에 따라 alu_bcond가 설정된다. 이들은 asynchronous하게 연산된다.

3.7. <ALUControlUnit.v>

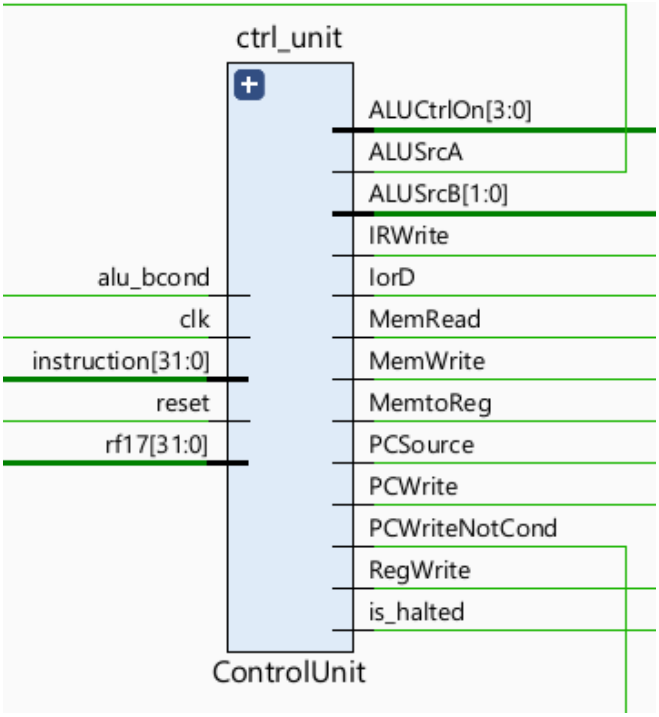
- alu_ctrl_unit



input : ALUCtrlOn, part_of_inst
output : alu_op

- Asynchronous Logic : input인 part_of_list가 바뀔 때마다 asynchronous하게 동작하여. 각 part_of_inst에 맞는 alu_op를 대입한다.
- Implementation : ALUCtrlOn이 1일 때는 ALU를 단순 Adder로 활용한다는 의미로 alu_op = add가 된다, 2일 때는 복잡한 alu_op를 확인하게 된다. part_of_inst를 가공해서 input으로 넣을 수도 있었겠지만, 가공하지 않고 32bit의 whole instruction을 input으로 넣었다.
  우선, opcode에 해당하는 part_of_inst[6:0]의 값에 따라 연산의 종류에 대한 case를 구분했다.
  ARITHMETIC 연산일 때는, ALU 연산에 해당하는 부분인 part_of_inst[14:12]와, part_of_inst[30]을 추가로 검토해 주었다. add와 sub는 funct3이 일치하고, sub 연산만 part_of_inst[30]이 1이기 때문에 따로 구분했다.
  ARITHMETIC_IMM 연산일 때도, ALU 연산 부분은 part_of_inst[14:12]를 사용하여 alu operation을 구분해 주었다.
  LOAD, JALR, STORE 연산일 때의 ALU 연산은 add이다. 왜냐하면 각각 ALU unit에서 immediate value와의 연산 부분이 있기 때문이다. (JAL은 별도의 Add unit을 활용하여 ALU 연산이 존재하지 않는다.)
  BRANCH 연산일 때는 part_of_inst[14:12]를 사용하여 별도로 alu_operation을 구분해 주어야 했다. 그 이유는, ALU unit에서 rs1과 rs2에 따른 branch condition도 계산해야 하기 때문이다.

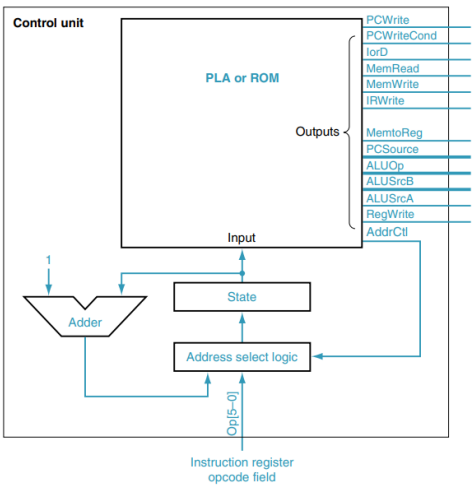
3.8. <ControlUnit.v>



input : alu_bcond, clk, instruction[31:0], reset, rf17[31:0]

output : ALUCtrlOn[3:0], ALUSrcA, ALUSrcB[1:0], IRWrite, IorD, MemRead, MemWrite, MemtoReg, PCSource, PCWrite, PCWriteNotCond, RegWrite, is_halted

- Synchronous logic : synchronous하게 매 clk마다 opcode와 AddrCtl의 값을 바탕으로 매 microPC를 업데이트 해준다.
- Asynchronous logic : Asycnronous하게 microPC가 바뀔 때마다, 해당 microPC에 맞게 output인 control singnal을 결정한다.
- input인 instruction이 바뀔 때마다 asynchronous하게 동작하며, 각 instruction에 맞는 control signal을 blocking assignment로 대입한다.
- implementation : 자세한 코드 설명은 2.6에 서술하였다.
  [구조에 대한 추가 설명]



Address select logic은, AddrCtl에 따라 state = `IF로 보내거나(0으로), state에 1을

더하거나, DispatchROM 테이블의 값에 따라 state를 설정해 준다. DispatchROM 테이블은 다음과 같다.

[DispatchROM2]

OPCODE	nextMicroPC
LOAD & STORE	`MEMADDRCOMPUTE
ARITHMETIC	`EXER
ARITHMETIC_IMM	`EXEI
BRANCH	`BRANCHCOMPUTE
JAL	`WBJAL
JALR	`WBJALR
ECALL	`ECALLCOMPUTE

[DispatchROM3]

OPCODE	nextMicroPC
LOAD	`MEMREAD
STORE	`MEMWRITE

[DispatchROM4]

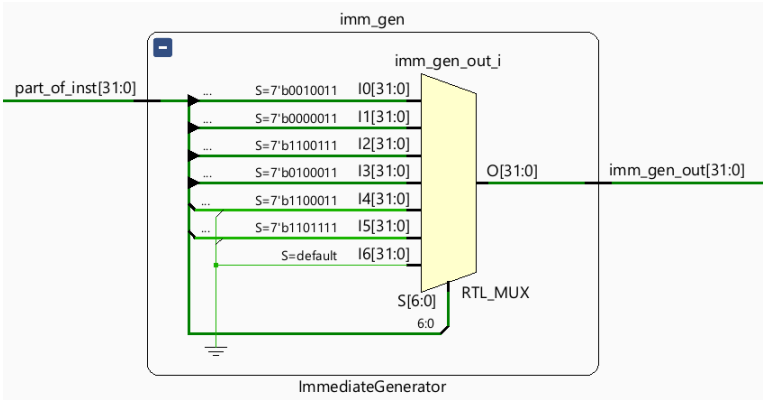
OPCODE	nextMicroPC
확인안함	`BRANCHSELECT

[DispatchROM5]

OPCODE	nextMicroPC
확인안함	`WBALU

3.9. <ImmediateGenerator.v>

- imm_gen



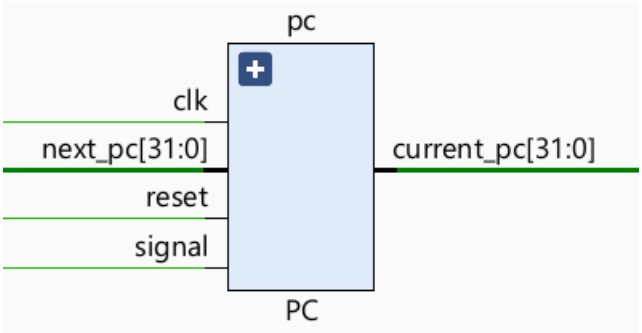
input : part_of_inst

output : imm_gen_out

- Asynchronous Logic : input인 part_of_inst이 바뀔 때마다 asynchronous하게 동작하며, 각 instruction을 해독하여 내부에 있는 immediate value를 찾아낸다.
- Implementation : 우선, opcode인 part_of_inst[6:0]을 통해 명령어의 종류를 구분했다. 각 명령어들에 대하여 RISC-V 규정에 따라 immediate value를 찾을 수 있다. 자세한 값은 Introduction에 있는 1.5의 표를 참조하였다. STORE와 BRANCH, JAL 명령의 경우에는 sign-extension을 해 주었다.

3.10. <PC.v>

- pc



input : clk. next_pc, reset, signal

output : current_pc

- (Positive clk) Synchronous Logic : positive clk일 때마다 우선 reset이 1일 경우 current_pc를 0으로 초기화해준다. 이는 reset이 언제 눌려도 clk이 positive가 될 때 current_pc가 바뀌는 결과를 야기한다. reset이 0일 경우에는 positive clk마다 한 클럭 동안 계산되었을 next_pc를 current_pc에 넣어 준다. 단, 이 과정은 signal (!alu_bcond && PCWriteNotCond || PCWrite)이 1일 때만 시행한다.

4. Discussion

4.1. Ex1, Ex2 스테이지 구분의 필요성

구현 요건에 따라 IF, ID, EXE, MEM, WB stage마다 한 클럭을 할당하려 했었다. (이는 수업 자료에서 스테이지를 IF1, IF2, IF3, IF4 등으로 구분한 것과는 다르다) 하지만, EXE stage에서 branch를 계산할 때, branch 여부를 계산하고, branch가 select 되었을 경우에는 branch 위치를 계산해야 하므로 두 번의 ALU가 필요하게 되었다. 따라서, branch instruction의 경우에는 EXE stage가 나뉘었다.

또한, microcode controller를 적용함으로서, microPC를 사용하게 되어 state 사이의 명확한 구분(이름을 사용, EXE, MEM처럼)을 하는 것이 아니라 microPC의 이름을 사용하여 구분하게 되었으므로 Ex1, Ex2 스테이지라기보다는 BRANCHCOMPUTE,

BRANCHSELECT microPC로 구분하게 된 것으로 보는 것이 더 나은 방법일 수도 있겠다

4.2. Microcode controller 적용

처음에는 Microcode controller를 사용하지 않고, Asynchronous 부분에서 nextStage를 하나하나 나누어 계산하여, Synchronous 부분에서는 clk에 따라 stage <= nextStage 연산만을 해 주었다. 하지만, DispatchROM 테이블을 이용한 Microcode controller를 추후 적용하게 되면서, 만약 다른 stage가 생기게 되어도 이를 테이블 부분에서 적용하기만 하면 되므로 더욱 확장성 좋은 multi_cycle CPU가 되었다.

4.3. Rf17값을 Register file 쓰지 않고 가져오기

이전 single cycle cpu 구현 과정에서는 Register file 자체에서 rf17을 asynchronous하게 가져와서 is_halted 설정에 사용하였다. 하지만 이번 과제에서는 registerFile.v를 수정하지 않고 rf17값을 통해 is_halted를 설정해야했다. IR[11:7] == 17이고 regwrite == 1 일 때의 write data값을 cpu의 Rf17레지스터로 따로 설정해 저장해 놓았고, isEcall일 때 이 값을 확인해 control unit에서 is_halted를 1로 설정할 수 있었다.

5. Conclusion

5.1. 수업시간에 배운 multi cycle CPU를 직접 구현하면서 CPU의 동작을 이해하고, 컴퓨터처럼 사고하는 능력을 배울 수 있었다.

5.2. RISC-V를 해독하고 신호를 보내는 부분과, 메모리, 레지스터를 다루는 부분으로 역할을 분담하였다.

5.3. Multi-cycle cpu가 Single cycle cpu 보다 어떤 점이 좋고, 어떻게 성능의 향상이 이루어지는지 직접 구현해보면서 이해할 수 있었다. 이를 위해 적절한 신호를 보내는 게 굉장히 추상적이고 모호했는데, Controlunit을 직접 구현해보며 심층적인 이해가 가능했다고 생각한다.