컴퓨터 구조 Lab2 Single-Cycle CPU

20210207 이지현
20210794 정유진

1. Introduction

1.1. Lab 1은 Vivado를 이용하여 single-cycle RISC-V CPU를 구현하는 것을 목표로 한다.

1.2. CPU의 모든 implementation은 한 사이클에 한 instruction이 프로세스되도록 한다.

1.3. 주어진 스켈레톤 코드는 top.v, Memory.v, RegisterFile.v, cpu.v, ALU.v, ALUControlUit.v, ImmediateGenerator.v, ControlUnit.v, PC.v, opcodes.v이 있고 추가로 만든 V 파일은 add.v, mux.v가 있다.

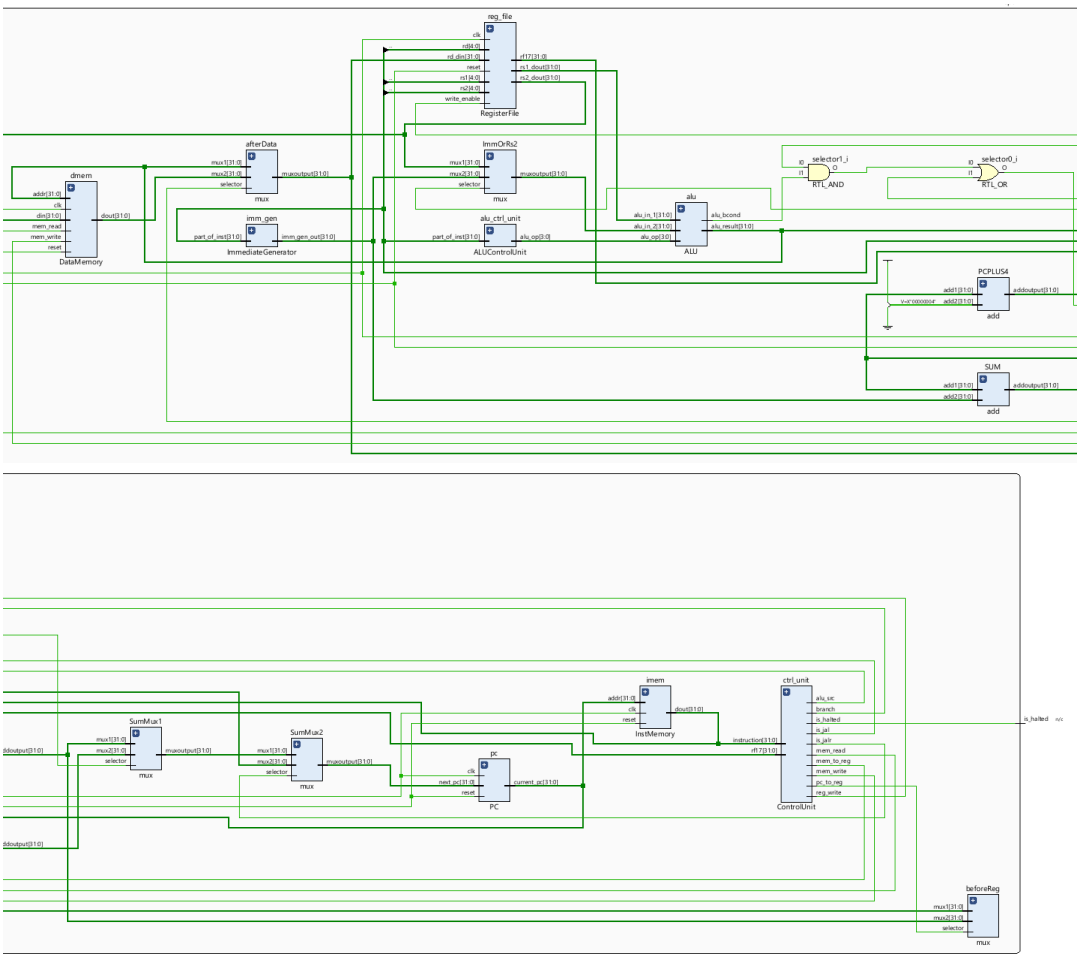
1.4. 테스트 코드는 총 3가지로 1. basic_ripes.asm, basic_mem.txt, control flow test 2.loop_ripes.asm, loop_mem.txt, 3. non-controlflow_ripes.asm, non-controlflow_mem.txt 가 있으며, Ripes를 통해 얻은 레지스터 값과 베릴로그 상에서 얻은 레지스터 값과 비교하며 정상적인 동작을 확인할 수 있다.

1.5. Opcode.v의 insturction들을 구현하였으며, 아래 표를 참고하여 진행하였다.

imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND
000000000000		00000	000	00000	1110011	ECALL

1.6. Single cycle CPU의 design과 implementation을 설명할 때 각각의 모듈이 synchronous인지 asynchronous인지 확인하며 각각의 스테이지를 설명하고자 한다.

2. Design



CPU 동작이 이루어지는 과정에 대한 순차적인 설명은 다음과 같다.

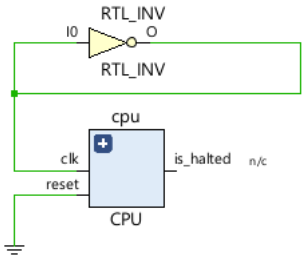
- Instruction processing은 보통 5가지 절차로 이루어진다.
- IF : instruction fetch로, 인스트럭션 실행을 위해선 메모리로부터 읽어와야한다.
  - Instruction memory, PC 가 주로 수행한다.
- ID : instruction decode로, 레지스터 파일에서 레지스터 값을 읽어오며 (operand fetch) instruction에 따른 control signal, immedgate value를 발생시킨다.
  - ControlUnit, immediateGenerator, RegisterFile 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.

다음은 각각의 모듈에 대한 synchronous/asynchronous 디자인의 포괄적인 설명이다.

Cpu	Top module로서 sub module들의 인스턴스 선언과 wire 변수들의 연결이 이루어지는 곳이다.
Add	두 input 값을 더해 output으로 내보내주는 역할을 하며, pc+4값의 계산이나 pc+imm 등 Next pc 연산을 주로 담당한다.
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	Asynchronous하게 Instruction을 읽어와, 이에 해당하는 CPU의 동작을 확인하여, 각 module에 내보내준다.
Memory	Instruction memory와 data memory로 나뉘며, Instruction memory에서는 asynchronous하게 instruction을 읽어오며 CPU Reset시 synchronous하게 초기화가 이루어진다. Data memory에서는 asynchronous하게 memory에서 data를 읽어오며 synchronous하게 data를 write한다. 마찬가지로 CPU Reset시 synchronous하게 초기화가 이루어진다.
Mux	selector가 0이면 output으로 mux1을 assign하고, 아니면 output으로 mux2를 assign한다. Dataflow modeling이다.
Opcodes	연산들의 funct3과 opcode를 선언해 놓았다.
PC	Synchronous하게 매 clk마다 current_pc를 next_pc의 값으로 업데이트 해준다.
RegisterFile	CPU 종료를 위해 rf17값을 내보내 주며, register file의 rs1 rs2 값을 asynchronous하게 읽어오고 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.

3. Implementation

3.1. <cpu.v> = top module



Input : clk, reset

Output : is_halted

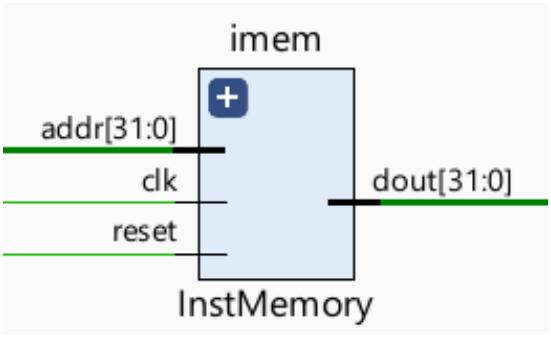
사용한 wire 변수들의 설명은 다음과 같다.

PC	nextPC : 다음 clock의 PC currentPC : 현재 PC instruction : instruction 32비트 currentPCplus4 : 현재 PC + 4값
ALU	alu_op : alu control unit에서 나온 alu operation alu_in_1 : alu의 첫번째 input alu_in_2 : alu의 두번째 input alu_bcond : branch가 선택되었는지 여부 alu_result : alu의 output
register	writeRegister : RD real_RegWriteData : Register Write Data RegReadData1 : RS1 RegReadData2 : RS2 rf17 : Regster file [17]
Data memory	real_DataOutput : mux를 거치고 난 data output DataReadData : mux를 거치기 전 data output
Immediate generator & mux step	Imm_gen_out : 해독된 immediate들 real_RegReadData2 : mux를 거쳐서 ALU의 input으로 사용되는 immediate 혹은 rs2 값
Control unit	isJal : Jal instruction인지 isJalr : Jalr instruction인지 isBranch : Branch instruction인지 regWrite : register에 write되는지 memRead : 메모리를 읽는지 memToReg : 메모리 값을 레지스터에 넣는지 memWrite : 메모리에 쓰는지

	ALUSrc : ALU에 immediate를 넣는지 PCToReg : PC가 레지스터에 들어가는지 IsEcall : Ecall연산인지 is_halted : 정상적으로 종료되었는지
Sum	SumOut : Sum 거치고 난 뒤 output SumMux1Out : SumMux1 거치고 난 뒤 Output SumMux2Out : SumMux2 거치고 난 뒤 output

3.2. <Memory.v>

- Instruction memory (InstMemory)

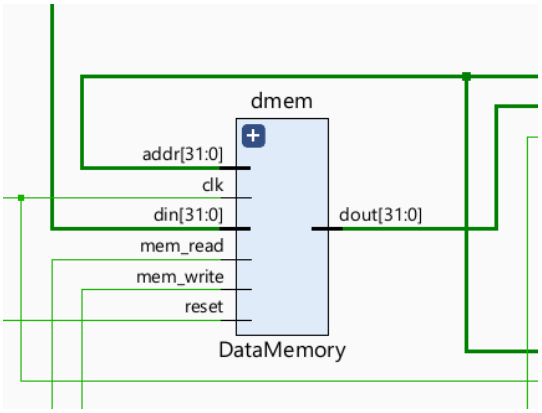


Input : addr, clk, reset

Output : dout

  - Asynchronous logic : imm_addr를 이용해 memory에 access해 dout에 담는 과정을 addr가 변할 때마다 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.

- Data memory

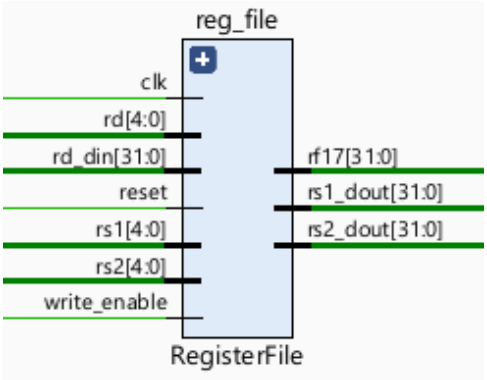


Input : addr, clk, din, mem_read, mem_write, reset

Output : dout

- Asynchronous logic : mem_read signal이 1이면 dout에 mem[dmem_addr]를 대입하는 과정은 assign의 오른쪽 값이 바뀌면 대입되는 방식을 이용했으므로 asynchronous한하게 진행된다고 볼 수 있다.
- (Positive clk) Synchronous logic : positive clk일 때 mem_write signal이 1인지 확인하고 din값을 memory에 write하는 과정과 reset이 1일때 data memory를 초기화하는 과정은 synchronous하게 진행된다.

3.3. <RegisterFile.v >



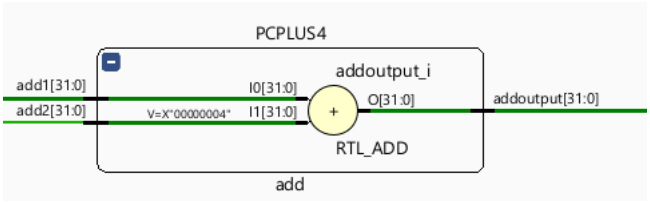
Input : clk, rd, rd_din, reset, rs1, rs2, write_enable
Output : rf17, rs1_dout, rs2_dout

- Asynchronous logic : rf[rs1], rf[rs2], rf[17]값이 변할 때마다 rs1_dout, rs2_dout, rf17에 assign문을 통해 대입해주는 과정은 asynchronous하다.
- (Positive clk) Synchronous logic : positive clk일 때마다 rf[0]의 값은 변하면 안되므로 0으로 고정시켜주고, write enable이 1인지를 확인해 rd_din을 레지스터 파일에 대입해주는 과정은 synchronous하다. 마찬가지로 reset이 1일 때 레지스터 파일을 0으로 초기화해주는 과정또한 synchrounous하다.

3.4. <add.v>

Add1과 add2를 input으로 받아서 더한다음 addoutput으로 출력해주는 모듈이다. Add module을 사용하여 instance를 만든 부분들은 다음과 같다.

- PCPlus4

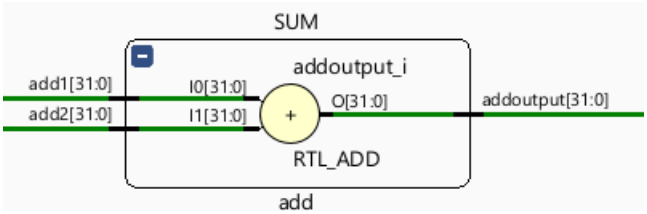


Input : add1 (currentPC), add2 (4)

Output : addoutput(currentPCplus4)

- Asynchronous logic : clk에 의존적인 Logic이나 intial을 이용하는 구간이 없고 assign문을 이용했으므로 asynchronous한 로직만 있다. Pc+4값을 구하기 위해서 currentpc와 4값을 더해서 currentPCplus4 wire에 전달해준다.

- Sum



Input : add1 (currentPC), add2 (imm_gen_out)

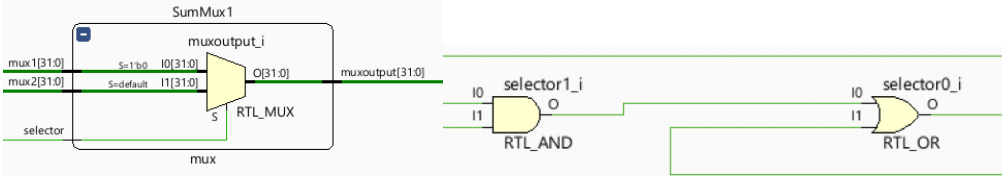
Output : addoutput(SumOut)

- Asynchronous logic : clk에 의존적인 Logic이나 intial을 이용하는 구간이 없고 assign문을 이용했으므로 asynchronous한 로직만 있다. PC + immediate 값을 구하기 위해서 currentpc와 imm_gen_out값을 더해서 SumOut wire에 전달해준다.

3.5. <mux.v>

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

- SumMux1

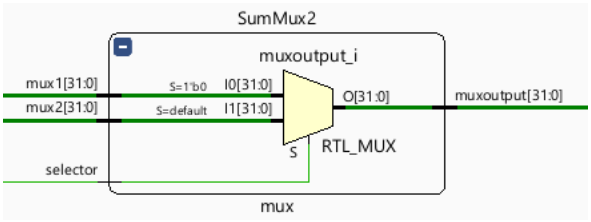


Input : mux1 (currentPCplus4), mux2 (SumOut), selector(isBranch && alu_bcond || isjal)

Output : muxoutput(SumMux1Out)

- Asynchronous logic : clk에 의존적인 Logic이나 intial을 이용하는 구간이 없고 assign문을 이용했으므로 asynchronous한 로직만 있다. PC+4 (다음 instruction)과 SumOut(PC+imm) 사이에서 src1 signal을 통해 한 값만 내보내주는 조건문 역할을 해준다.

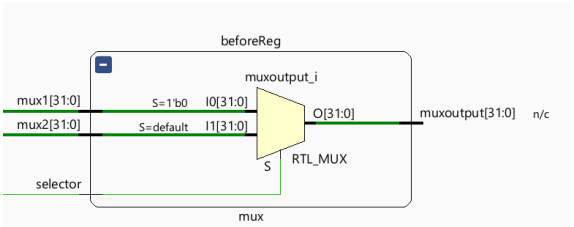
- SumMux2



Input : mux1 (SumMux1Out), mux2 (ALU_out), selector(isJalr)
Output : muxoutput(nextPC)

- Asynchronous logic : clk에 의존적인 Logic이나 intial을 이용하는 구간이 없고 assign문을 이용했으므로 asynchronous한 로직만 있다. SumMux1Out과 ALU_out 사이에서 isJalr signal을 통해 한 값만 내보내주는 조건문 역할을 해준다.

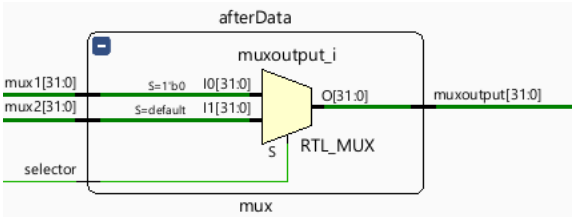
- beforeReg



Input : mux1 (real_DataOutput), mux2 (currentPCplus4), selector(PCToReg)
Output : muxoutput(real_RegWriteData)

- Asynchronous logic : clk에 의존적인 Logic이나 intial을 이용하는 구간이 없고 assign문을 이용했으므로 asynchronous한 로직만 있다. DataOutput과 PC+4 사이에서 PCtoReg selector를 통해 한 값만 내보내주는 조건문 역할을 한다.

- AfterData



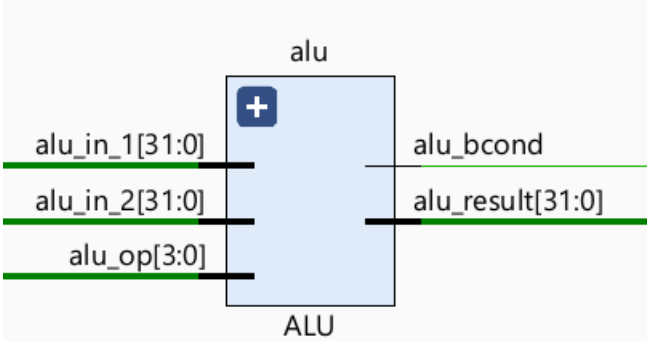
Input : mux1 (ALU_out), mux2 (DataReadData), selector(memToReg)

Output : muxoutput(real_DataOutput)

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

3.6. <ALU.v>

- alu



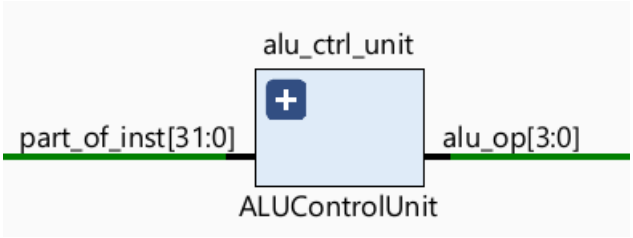
input : alu_in_1, alu_in_2, alu_op

output : alu_bcond, alu_result

- Asynchronous Logic :
- Implementation :

3.7. <ALUControlUnit.v>

- alu_ctrl_unit



input : part_of_inst

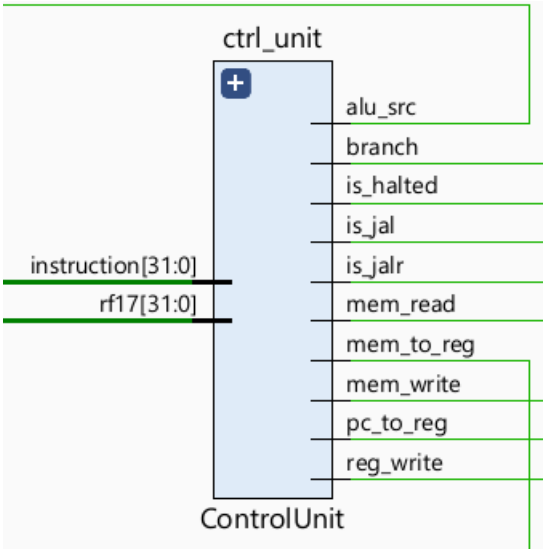
output : alu_op

- Asynchronous Logic : input인 part_of_list가 바뀔 때마다 asynchronous하게

동작하여. 각 part_of_inst에 맞는 alu_op를 대입한다.

- Implementation : 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 : instruction, rf17
output : is_jal, is_jalr, branch, reg_write, mem_read, mem_to_reg, mem_write, alu_src, pc_to_reg, is_ecall, is_halted

- Asynchronous logic : input인 instruction이 바뀔 때마다 asynchronous하게 동작하며, 각 instruction에 맞는 control signal을 blocking assignment로 대입한다.
- implementation : 우선, 해당되지 않는 control singal을 구분하기 위해 모든

output을 0으로 초기화한다. 그리고, instruction의 종류를 구분하는 opcode인 instruction[6:0]의 값에 따라 case를 구분한다.

ARITHMETIC 연산일 경우, reg_write = 1이다. Write register에 값을 쓰기 때문이다. ARITHMETIC_IMM 연산일 경우 reg_write, alu_src = 1이다. register에 값을 쓰고, immediate value를 사용하기 때문이다.

LOAD 연산일 경우 reg_write, alu_src, mem_read, mem_to_reg = 1이다. register에 값을 쓰고, immediate value를 사용해 주소를 계산하고, 메모리에서 값을 읽고, 메모리의 값을 레지스터에 쓰기 때문이다. mem_read 신호는 Data memory. 유닛에 전달된다. mem_to_reg 신호는 ALU의 결과와 데이터에서 읽어온 값 중, 데이터에서 읽어온 값을 Write_data에 넣어주기 위해 가져오게 된다.

STORE 연산일 경우 alu_src, mem_write = 1이다. immediate value를 사용해 주소를 계산하고, 메모리에 값을 쓰기 때문이다.

JALR 연산일 경우 is_jalr, reg_write, alu_src, pc_to_reg = 1이다. register에 값을 쓰고, immediate value를 사용해 주소를 계산하기 때문이다. PC + 4를 Registers의 Write data에 써 주어야 하기 때문에, pc_to_reg 신호를 설정해 준다. is_jalr 신호는 점프를 할 위치를 정할 때 사용된다. jal과 branch와는 다르게 ALU에서 계산된 값으로 점프하게 되므로 구분해 준다.

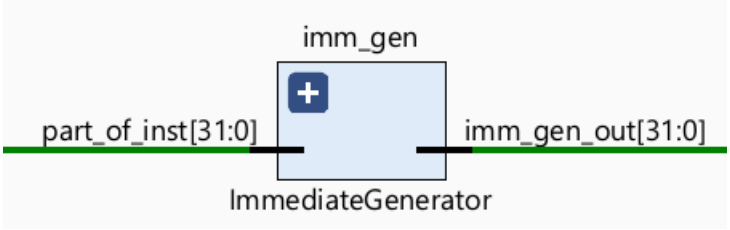
JAL 연산일 경우 is_jal, reg_write, alu_src, pc_to_reg = 1이다. is_jal 신호는 PC에 immediate 값을 바로 더해서 점프를 해야 하므로 이를 jalr과 구분하기 위해 사용해 준다.

BRANCH 연산일 경우 branch = 1이다. branch 신호는, branch와 b_cond가 모두 1일 때, branch가 taken되어 PC의 값이 변하므로 이를 jal, jalr과 구분하기 위해 사용해 준다.

ECALL 연산일 경우 is_ecall = 1이고, rf17 레지스터가 10일 경우 is_halted = 1이다. rf17 레지스터가 10일 경우 정상적으로 종료됨을 의미하여 is_halted가 1로 변한다.

3.9. <ImmediateGenerator.v>

- imm_gen



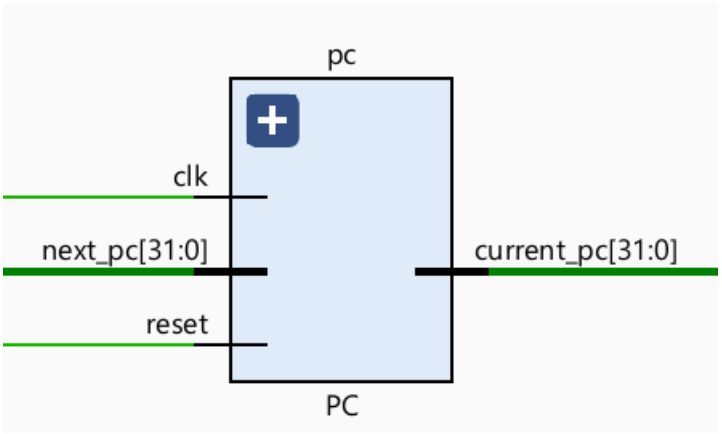
input : part_of_inst

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

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.

4. Discussion

4.1. 모듈의 instance 사용

모듈의 구조를 만들고 이를 마치 c++에서의 객체처럼 생성하여 사용하는 방식은 이번 랩에서 특히 mux와 adder 구현에 유용하게 사용되었다. mux와 adder는 같은 구조가 다른 input으로 여러 번 반복되기 때문에 모듈의 instance 사용의 유용성을 잘 경험할 수 있었다.

4.2. Register x0의 불변하는 0

loop-mem test case가 원하는 대로 잘 작동하지 않았다. Instruction을 하나하나 확인해 본 결과, x0는 0으로 불변해야 하는데, x0에 다른 값이 들어가도 이것이 처리가 되지 않는 현상을 발견하였다. 따라서 x0는 0으로 고정시키는 식을 추가하였다.

4.3. Switch-case문에서의 vivado 문법

정해진 instruction을 해독하는 것을 구현하기 위해서는 if문보다 switch문이 훨씬 유용하다. vivado에서는

```
case(비교할 수) :
        num 1: begin
        end
        num2 : begin
        end
        ...
endcase
```

문법을 사용한다.

4.4. Immediate Generater에서의 Sign-Extension

Sign-Extension을 하기 위해서 처음에는 ($signed)를 잘 사용하여 계산해 보려고 했지만, 자꾸 오류가 발생하였다. 그래서, 가장 직관적인 방법으로

```
if(part_of_inst[31] == 1)
        imm_gen_out[31:12] = 20'b11111111111111111111;
else
        imm_gen_out[31:12] = 20'b00000000000000000000;
```

과 같은 직접 대입 방식을 사용하였다.

5. Conclusion

5.1. 수업시간에 배운 single 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. 해독하고 신호를 보내는 부분은 Asynchronous하게 동작하였지만, 메모리, 레지스터, PC를 다루는 부분은 Synchronous하게 동작하였다. 그 이유는 결과가 나오지 않으면 그 뒤의 작업을 진행할 수 없게끔 posedge 사이의 시간 간격을 두어야 하기 때문이고, 프로그램의 진행 단계에 맞게 동기화 시키기 위함이다.