**CSED-311 LAB3 REPORT**

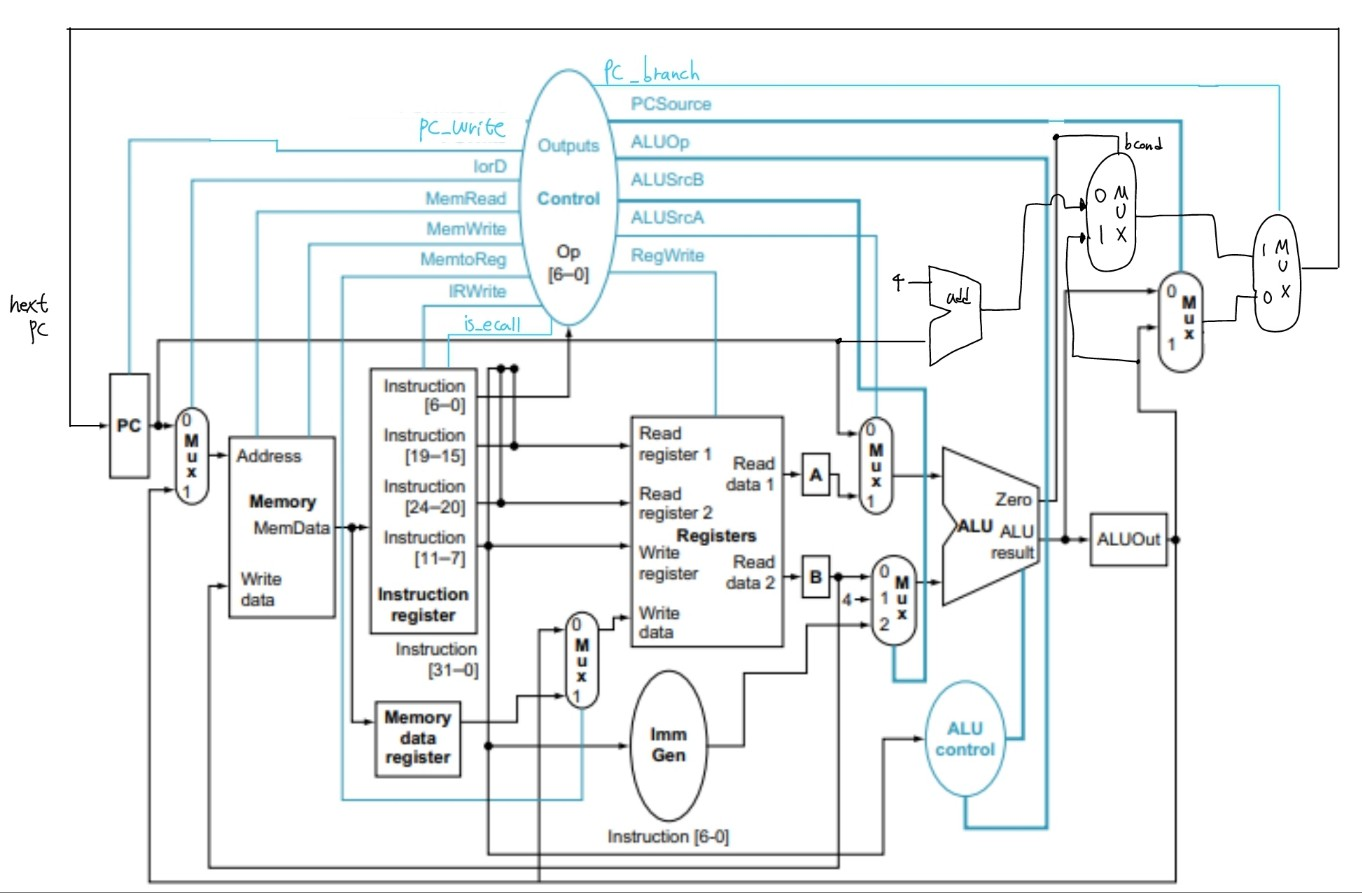
김모세

표승현

**Introduction**

RISC-V instruction을 기반으로 하는 Multi cycle CPU를 verilog를 이용해 구현한다. multie cycle CPU는 각 명령어를 실행할 때 여러 개의 작게 나뉜 클럭 사이클을 사용한다. 따라서 instruction은 여러 사이클에 걸쳐 수행된다. single cycle CPU에 비해 각 instruction에서 낭비되는 시간을 줄여 성능 면에서 보다 효율적이다.

**Design**

****

**<design of multi cycle cpu>**

기본적으로 수업 시간에 공부한 multi cycle CPU의 회로 구조를 토대로 구현하였다. 다만 branch instruction을 실행할 때 보다 안정적인 PC 업데이트를 위해 약간의 수정을 거쳤다. Instruction register로부터 instruction을 받아 branch인 경우 pc\_branch를 활성화한다. 그럴 경우 bcond값에 따라 pc+4 또는 pc+immediate value가 next pc가 된다. control unit에서 is\_ecall 시그널을 활성화하면, register file에서 신호를 받아 X17의 값을 확인한 후 작동을 종료하는 기능은 single cycle에서 구현한 방식과 같다.

**Single cycle CPU와 multi cycle CPU의 차이**

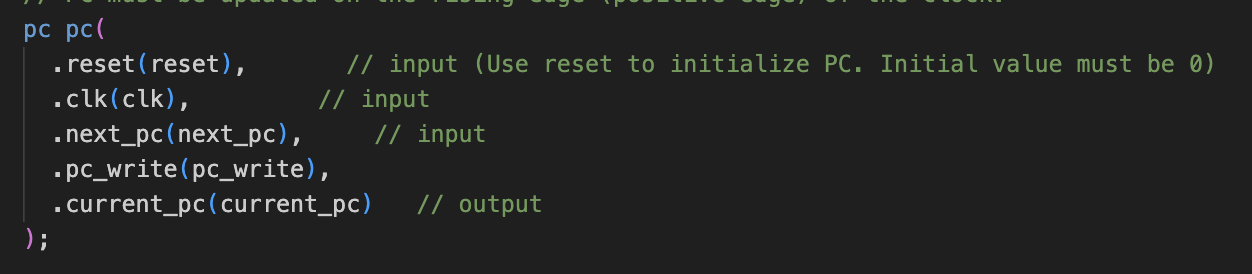
single cycle CPU는 한 사이클당 하나의 instruction을 수행하는 반면, multi cycle CPU는 여러 사이클에 걸쳐 하나의 instruction을 수행한다. Multi cycle CPU는 여러 사이클에 걸쳐 instruction을 수행하기 때문에 각 이전 사이클의 정보를 다음 사이클에서 사용할 수 있도록 저장하는 register를 필요로 한다. Register에 다음 사이클에 필요한 정보를 저장하여 다음 state에서 이를 사용할 수 있도록 한다.

**Multi cycle CPU가 더 나은 이유**

single cycle CPU는 고정된 하나의 사이클 시간동안 반드시 하나의 instruction을 수행할 수 있어야 한다. 때문에 가장 긴 latency를 가진 instruction에 맞춰 사이클 타임이 정해져 나머지 instruction 수행에 있어서는 시간의 낭비가 있다. 하지만 multi cycle CPU는 각각의 instruction 수행에 필요한 시간을 작게 나눈 cycle로 handling해 각 instruction마다 필요한 만큼만 시간을 소요할 수 있게끔 했다. 덕분에 낭비되는 시간이 줄어들어 전체적인 성능이 증가한다.

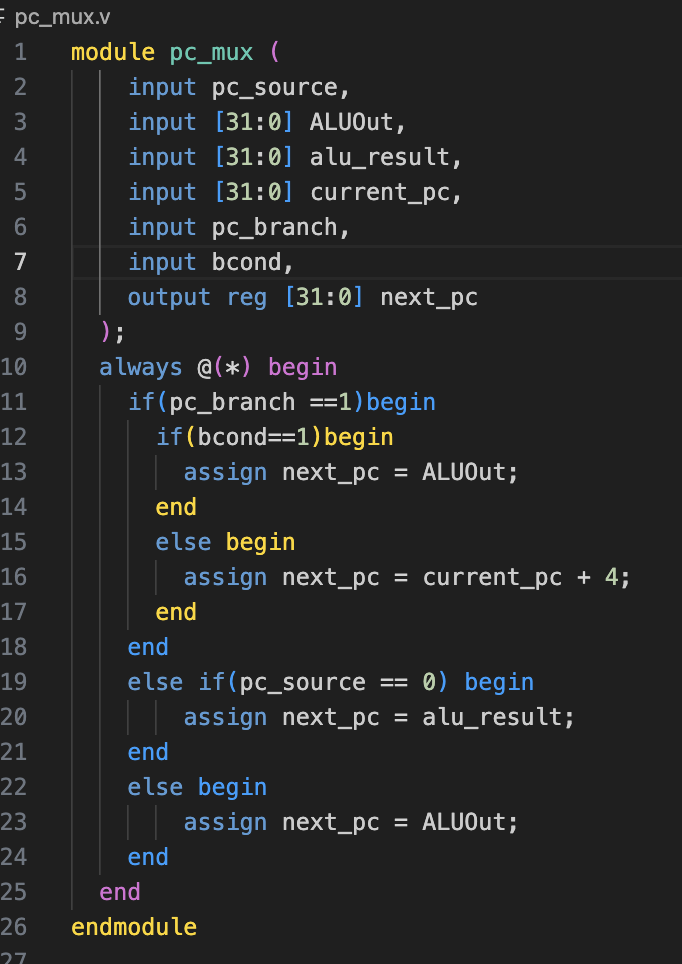
**Implementation**

1. pc 모듈



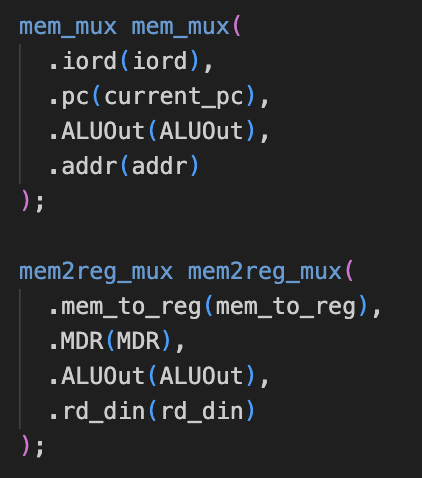
pc모듈에선 synchronous하게 current\_pc를 next\_pc로 업데이트한다. 또한 asynchronous하게 pc\_write 컨트롤 시그널이 켜졌을떄 pc(cpu에 정의한 register)를 바꿀 수 있도록 pc\_enable을 1로 해준다.

2)pc\_mux 모듈



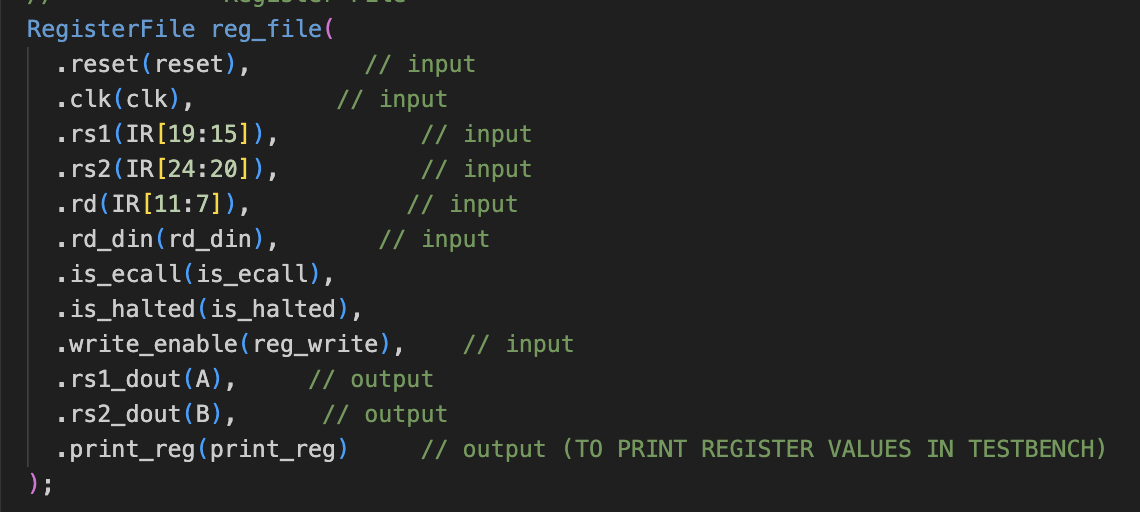
pc\_mux 모듈에선 asynchronous하게 next\_pc를 업데이트한다. 우선 branch가 켜지고 bcond이 1이면 next\_pc에 current\_pc + immediate 값을 넣고, bocnd이 0이라면 current\_pc+4값을 넣는다. pc\_source 시그널이 0이면 alu연산 결과를 바로 next\_pc에 넣고 1일 경우 ALUOut 레지스터 값을 넣는다.

3) mem\_mux, mem2reg\_mux 모듈



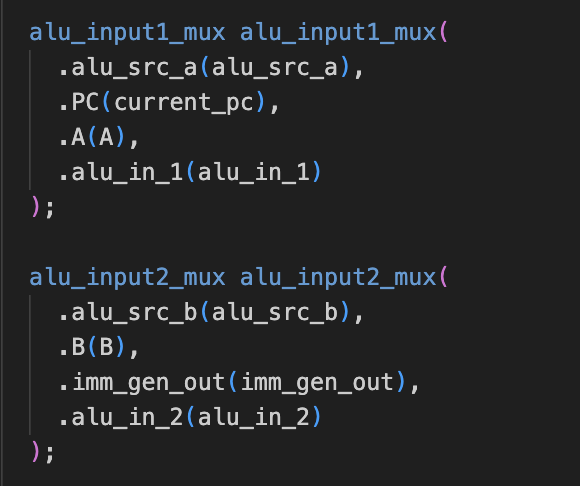
mem\_mux와 mem2reg\_mux 모듈에선 각 iord, mem\_to\_reg 시그널에 맞춰 알맞은 값을 메모리와 레지스터 파일의 쓰기 데이터 입력으로 asynchronous하게 넣어준다.

4) RegisterFile 모듈



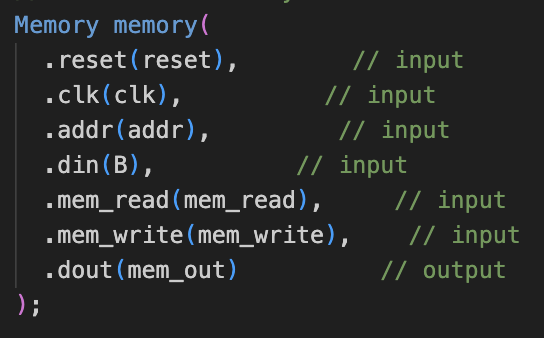
RegisterFile모듈에선 asynchronous하게 rs1\_dout과 rs2\_dout에 레지스터 파일에서 읽은 데이터를 넣어주고, write\_enable이 켜졌을 경우 synchronous하게 레지스터 파일에 데이터를 입력한다.

5)alu\_input1\_mux, alu\_input2\_mux



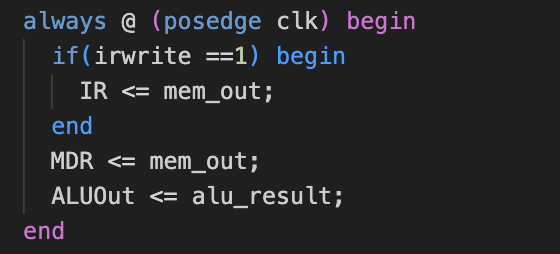
alu\_input1\_mux와 alu\_input2\_mux에선 각 alu\_src 시그널에 맞게 알맞은 값을 alu의 input으로 asynchronous하게 넣어준다.

6)Memory 모듈



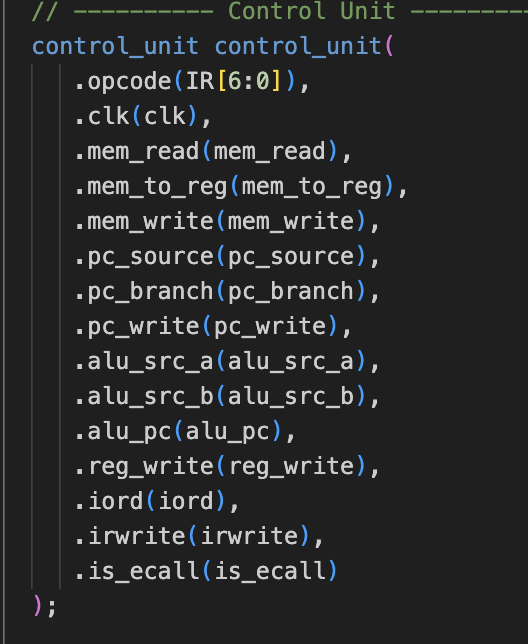
Memory 모듈에선 asynchronous하게 mem\_read가 켜졌을 경우 dout에 입력으로 받은 주소에 있는 값을 넣어준다. 또한 synchronous하게 mem\_write가 켜졌을 때 입력으로 받은 주소에 데이터를 쓴다.

7)IR, MDR, ALUOut 레지스터 업데이트



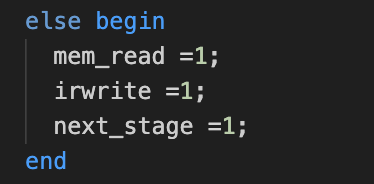
IR 레지스터의 경우 synchronous하게 irwrite시그널에 따라 memory에서 읽은 값을 입력해준다. MDR과 ALUOut 레지스터들 또한 매 clock cycle 마다 메모리에서 읽은 값과 alu연산 결과를 넣어준다. 이떄 MDR에는 memory에서 읽은 instruction이 들어갈 수도 있는데 이 경우 여전히 write\_enable은 0으로 되어 있기에 레지스터 파일에 쓰리기값이 쓰이는 것을 방지할 수 있다.

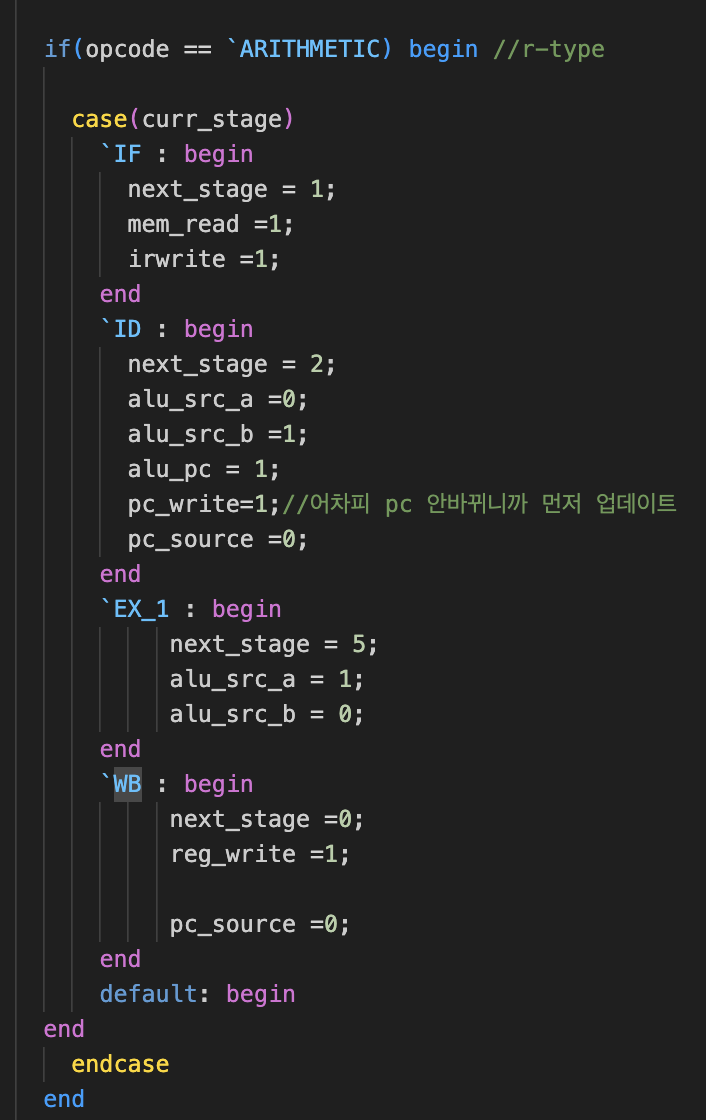
8) control\_unit 모듈



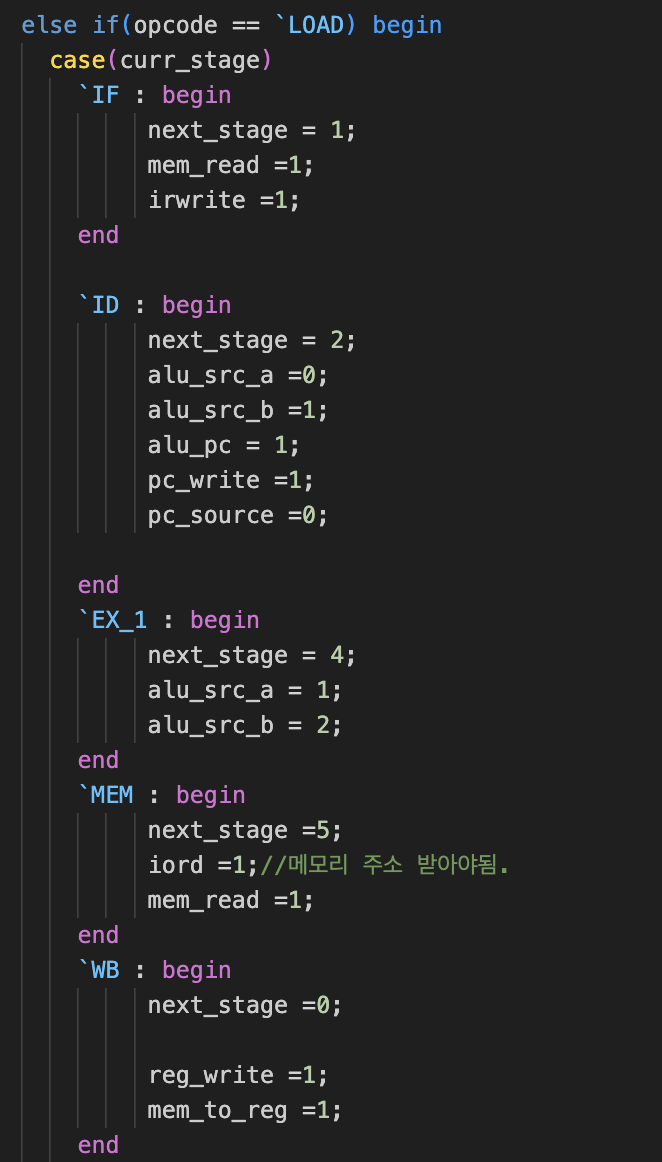
control\_unit 모듈에선 current\_stage에 컨트롤 시그널들을 세팅해주고 next\_stage를 지정한다. 또한 synchronous하게 current\_stage를 next\_stage로 업데이트한다. stage는 IF, ID, EX\_1, EX\_2, MEM, WB 6개가 있다. 가장 바깥에 if, else\_if, else문을 이용해 opcode에 따라 케이스를 나눴으며 각 조건문 안에서 case 문을 이용해 해당 opcode를 실행하는 상황에서 current\_stage에 따라 시그널들을 세팅해줬다. 이때 IF stage를 만들긴 했지만 IF 단계에선 현재 instruction의 opcode를 모르는 상황이기에 아래와 같이 else문(opcode가 정해지지 않은)으로 빼어 진행되도록 하였다.(앞으로 설명에서 IF stage의 설명은 생략하겠음) 또한 alu를 세 단계에서 이용하도록 설계했는데 ID 단계에서 먼저 pc+4를 계산해 pc 레지스터에 업데이트 해주도록 했고, EX\_1에선 arithmetic 명령어에 필요한 연산을 진행한다. EX\_2는 branch 나 Jump 명령어일때 분기주소 연산과 branch condition을 확인하는 두 부분으로 alu 사용을 나누기 위해 이용한다.

asynchronous하게 모든 시그널들을 0으로 초기화 한 뒤 필요한 시그널만 1로 바꾸는 식으로 구현했다.

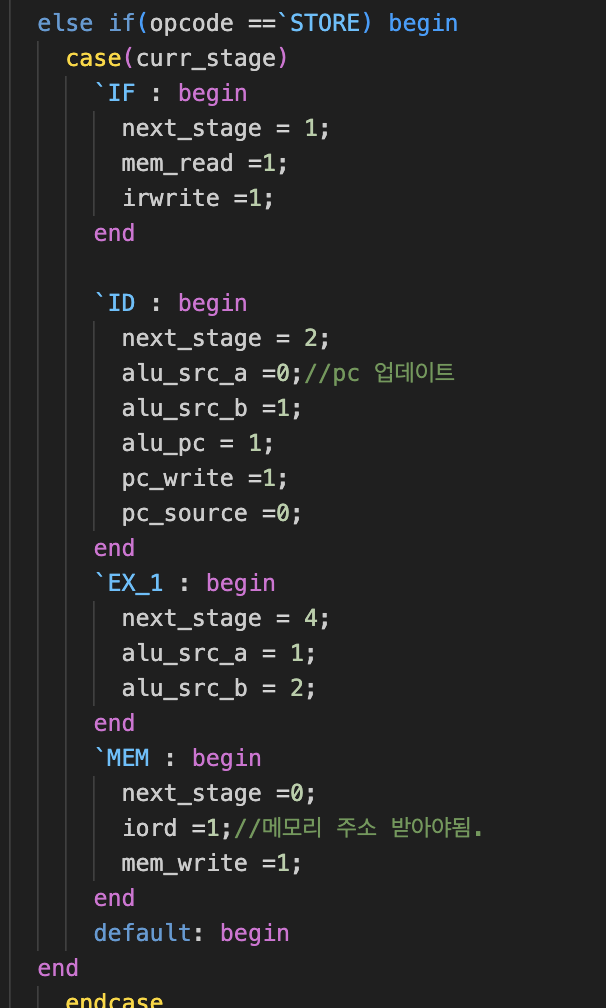




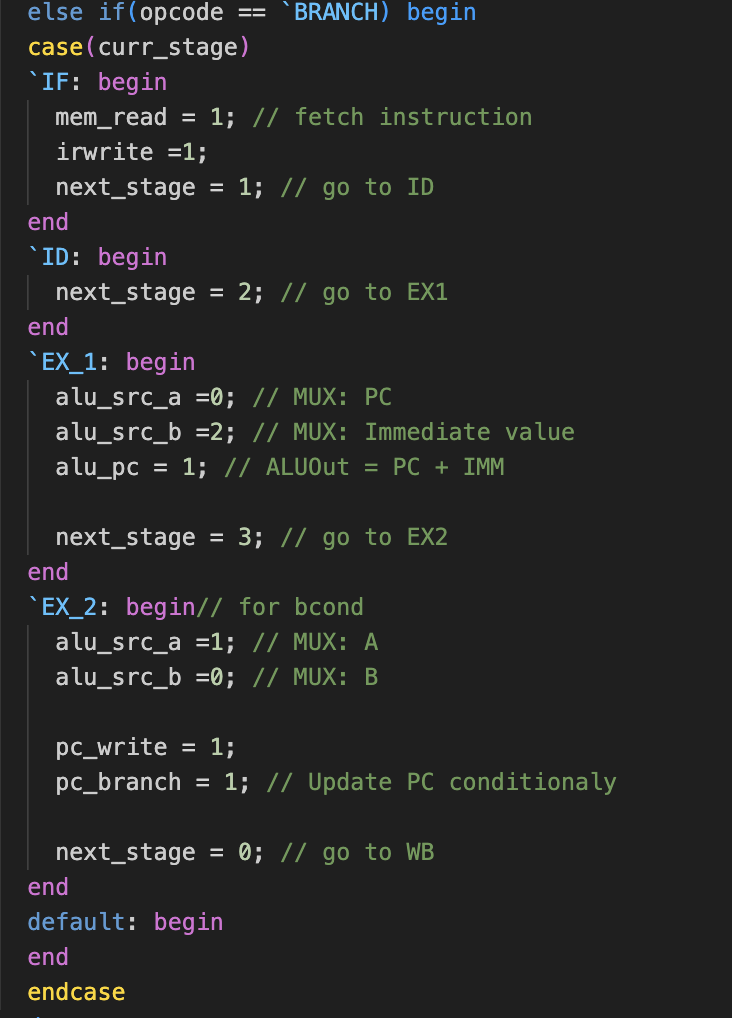
‘arithmetic 명령어의 경우 IF이후 ID단계에서 필요한 레지스터 값들을 읽고 pc+4를 미리 계산해 PC레지스터에 업데이트 해준다.(irwrite가 0이므로 현재 instr이 바뀌진 않는다.) EX\_1단계에선 ID에서 읽은 두 레지스터 값을 이용해 연산을 진행하고 WB단계에선 그 결과를 목적지 레지스터에 써준다. ‘arithmetic\_imm 명령어의 경우에도 위와 거의 같은데 EX\_1 단계에서 두번쨰 alu의 input으로 immediate를 고른것에 차이가 있다.



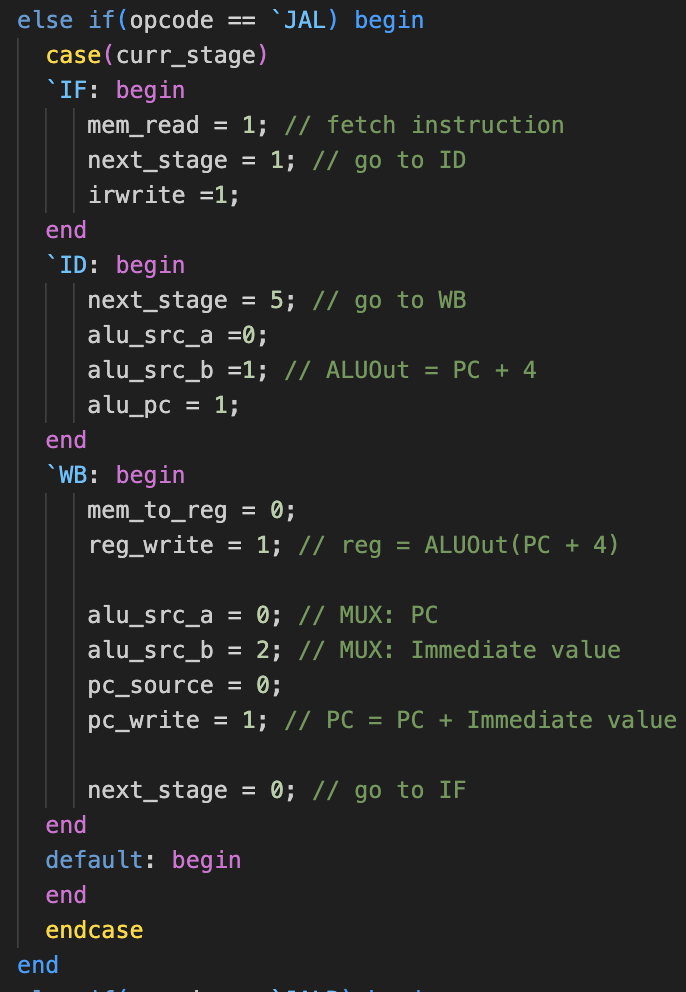
load 명령어의 경우 EX\_1단계까지는 ‘arithmetic\_imm과 같다. 하지만 EX\_1단계 이후 바로 MEM 단계로 넘어가며 MEM에서 원하는 데이터를 읽은 뒤 WB으로 넘어가 레지스터 파일에 입력하도록 한다.



store 명령어의 경우 load명령어와 EX\_1까지는 같지만 이후 MEM단계에서 메모리에 원하는 값을 입력한 뒤 명령어가 종료된다는 점에서 차이가 있다.(다음 명령어 fetch)



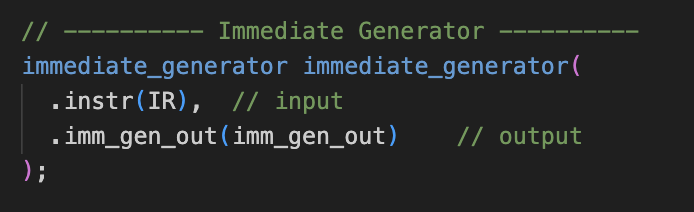
branch 명령어의 경우 ID단계에서 레지스터 값을 읽지 않고 immediate만 생성한다.(또한 ID에서 pc+4미리 계산 안함) 이후 EX\_1단계로 가 분기주소를 미리 연산한다. 다음 EX\_2 단계로 넘어가 branch condition을 만족하는지 확인하고 조건에 맞게 다음 pc를 결정하도록 시그널을 보내준다.



jal 명령어의 경우 ID에서 pc+4를 계산한 뒤 WB 단계로 바로 넘어가 pc+4를 레지스터 파일에 입력해주는 동시에 pc+immediate값을 다음 pc로 업데이트 해주도록 시그널들을 세팅한다.

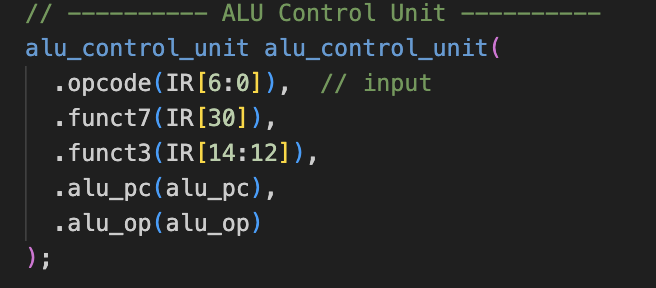
jalr의 경우 jal과 WB단계에서 첫번째 alu의 값을 레지스터 파일에서 읽은 값으로 하는 부분만 차이가 있다.

9) immediate\_generator 모듈



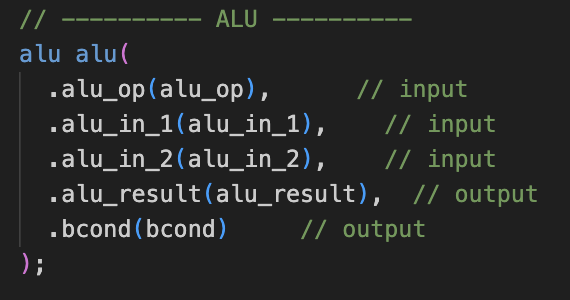
immediate\_generator모듈은 asynchronous하게 instruction을 받아 imm\_gen\_out을 구한다.

10)alu\_control\_unit 모듈



alu\_control\_unit 모듈은 asynchronous하게 opcode, funct7, funct3, alu\_pc(pc연산을 위해 alu가 addition을 실행하도록 하는 시그널)를 input으로 받아 alu\_op를 내보낸다.

11) alu 모듈

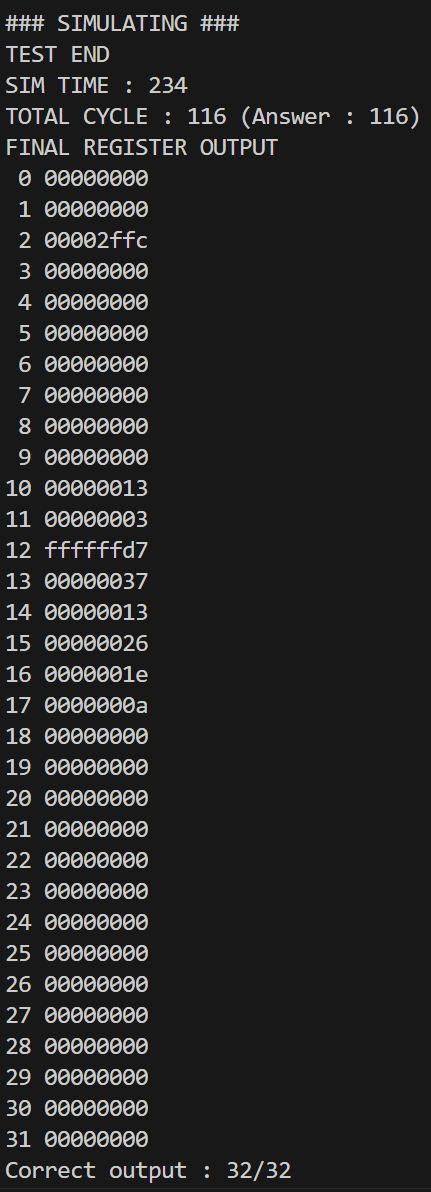
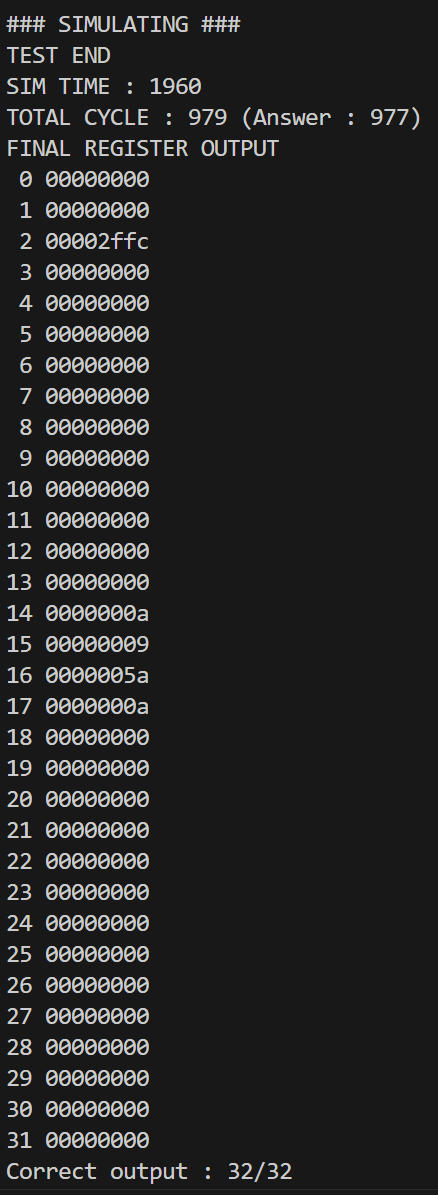


alu 모듈은 asynchronous하게 입력으로 받은 값들과 신호를 이용해 연산을 하고 그 값을 내보낸다.

**Discussion**

1. Branch에서의 next pc 연산
   1. 처음 시도에서는 교안에서 설명된 방식대로 구현하고자 하였다. pc+4를 미리 계산해서 alu out에 저장하고, 다음 사이클에서 bcond를 연산 후 연산 결과에 따라 alu out의 pc+4를 업데이트할지 alu로부터 pc+immediate value를 연산하여 업데이트할지 정하는 방식이었다.
   2. 첫번째 문제는 bcond signal을 handling하는 과정에서 발생했다. instruction이나 alu result같은 경우 각각의 값을 일시적으로 저장하는 register를 통해 다음 사이클로 정보를 전달할 수 있다. 하지만 signal은 그런 기능을 하는 register가 없어 bcond를 계산 후 pc+immediate를 계산하기 위해 다음 사이클로 넘어가는 순간 bcond의 값이 사라지는 문제가 생겼다. 이를 해결하기 위해 bcond 연산을 next pc 값이 모두 연산된 뒤로 순서를 미뤄 signal이 필요한 순간 사라지는 것을 해결했다.
   3. 연산 순서를 바꾼 뒤 추가적인 문제가 발생하였다. 바뀐 순서에 따라 pc+4를 앞선 state에서 미리 pc에 업데이트 한 후 bcond값에 따라 pc에 pc+immediate value로 덮어 씌우는 방식을 시도하였다. 하지만 pc+immediate value을 계산하는 시점에서 이미 pc가 업데이트되어 의도한 pc 값이 계산되지 않는 문제가 발생하였다. 이를 해결하기 위해 branch taken이라는 signal과 mux, pc adder를 추가로 만들고 branch taken이 활성화된 경우 bcond 값에 따라 pc+4 또는 pc+immediate value 값을 next pc로 갖게끔 수정하였다.
2. verilator read 오류
   1. verilator에서 read를 이용해 txt의 정보를 읽을 때, 맨 마지막 줄에 들여쓰기를 하지 않으면 마지막 줄이 누락되는 문제를 발견하였다. txt 마지막 줄에 공백을 추가하여 이를 해결하였다.

**Conclusion**

<basic\_ripe & loop\_ripe 수행 결과>

위 사진은 각각 lab3에서 작성한 코드를 통해 basic\_ripe과 loop\_ripe를 수행한 결과이다. basic에선 총 사이클은 116이 소요되었고, loop에서는 979만큼 소요되었다. multi cycle CPU에서는 하나의 instruction이 여러개의 사이클 즉, 여러 개의 state에 의해 수행되므로 state를 어떻게 정의하는냐에 따라 소요되는 cycle이 달라질 것이다. 본 실습에서는 IF, ID, EX\_1, EX\_2, MEM, WB으로 총 6개의 state를 정의하여 구현했다.

실습을 진행하며 ripes 프로그램과 gtkWave를 보다 잘 활용할 수 있게 되었다. 작성한 코드의 파형을 추적하여 register의 값이 ripes의 것과 달라지는 부분을 중점적으로 검토하니 더욱 수월하게 디버깅할 수 있었다. multi cycle CPU를 성공적으로 구현했고 register 값도 목표한 값과 동일하게 출력되는 것을 확인했다.