**CSED-311 LAB4-2 REPORT**

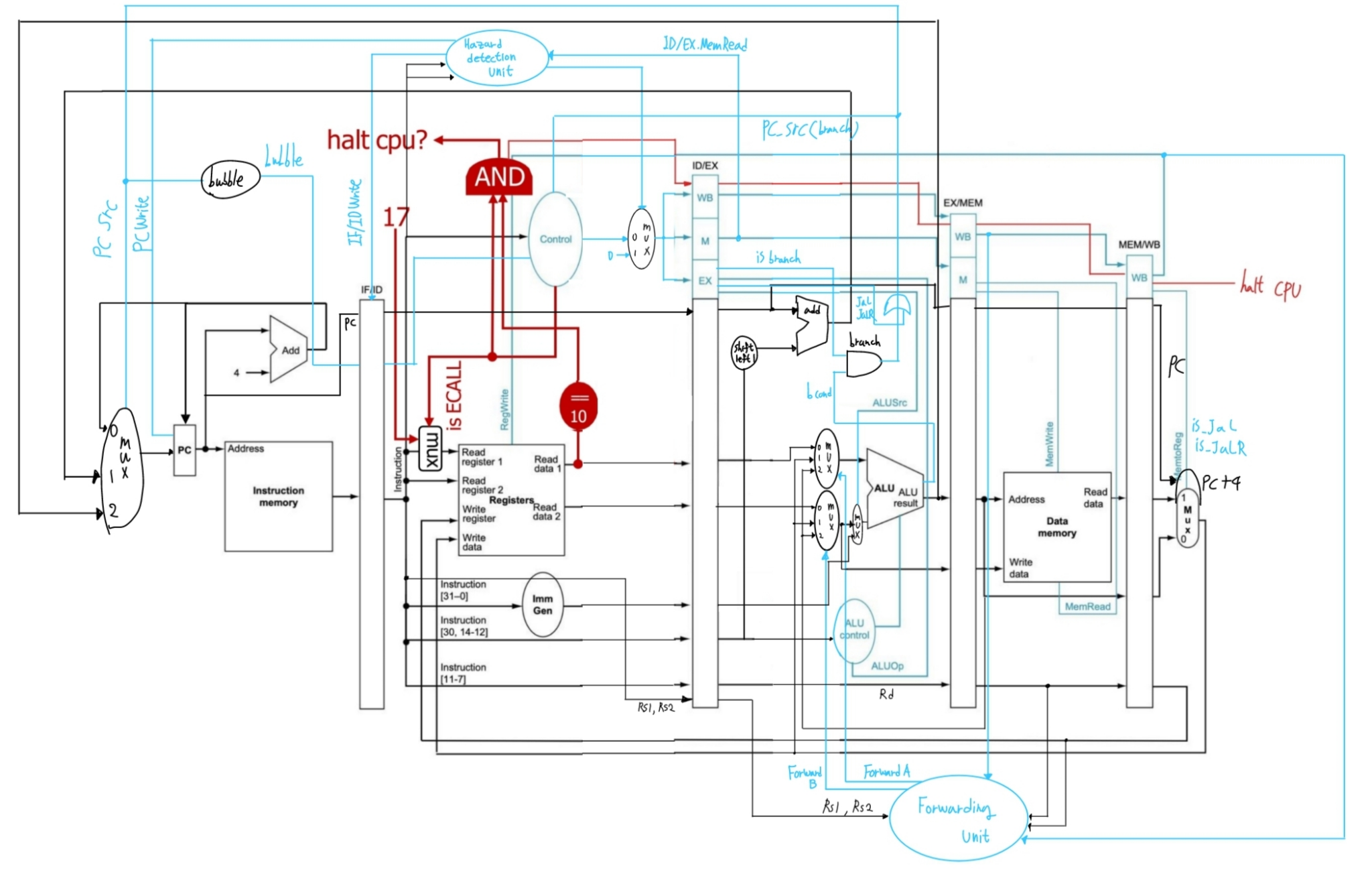
김모세

표승현

**Introduction**

지난 실습 시간에는 controlflow가 없는 pipelined CPU를 구현하였다. 이번 실습에서는 branch signal과 모듈을 이용해 controlflow를 처리할 수 있는 CPU 구현을 목표로 한다.

**Design**



<Design of pipelined CPU with controlflow>

이번 실습에서는 지난 시간 구현한 pipelined CPU에 controlflow를 추가로 구현한다. 구조는 lab4-1의 회로도를 기반으로 하였고, branch와 jal, jalr instruction을 처리하기 위한 몇 가지 모듈이 추가되었다. Branch prediction은 always not taken 기법으로 구현했다. 따라서 분기되는 모든 순간을 misprediction으로 간주하여 bubble 시그널을 만들어 내도록 했다.

**Implementatio**

1. cpu module

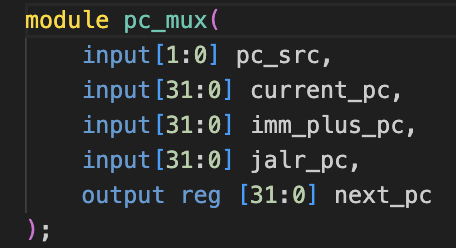
cpu module에선 우리의 cpu design에 필요한 wire와 register가 정의되어 있으며 세부 모듈들이 불리고 있다. 또한 매 posedge clk마다 파이프라인 register들을 다음 파이프라인으로 넘긴다.

1. PC module

이번 랩에서는 non-branch instruction만 다루므로 매 사이클마다

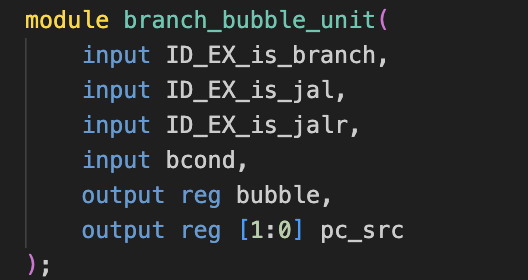
pc를 4씩 더해준다. data\_forwarding을 구현해도 해결할 수 없는 hazard의 경우 pc\_write singal이 0이 되는데 이때는 pc를 증가시키지 않도록 한다.

1. pc\_mux module



asynchronous하게 pc\_source값에 따라 next\_pc를 결정한다. pc\_source가 0이면 branch가 taken되지 않은 상황이므로 pc+4를 next\_pc로 하고, 1이면 현재 pc에 변위만큼 더한것을 next\_pc로, 2면 rs1 레지스터 값과 변위를 더한것을 next\_pc로 한다. 분기가 일어났을때 bubble 두개를 넣어줘야 하는데 이는 branch\_bubble\_unit module에서 다룬다.

1. branch\_bubble\_unit module



branch\_bubble\_unit은 EX 단계에서 branch를 resolve하는 역할을 한다. ID\_EX 단계 파이프라인 레지스터에서 세개의 control\_flow instruction 정보를 받고 alu에서 bcond를 받아 pc\_source와 bubble을 설정해준다. ID\_EX\_is\_branch && bcond 가 1인 경우와 ID\_EX\_jal인 경우 pc\_source를 1로 설정해 pc+imm이 pc\_mux module에서 next\_pc가 되도록 해주고, ID\_EX\_is\_jalr이 1인 경우 pc\_source는 2, bubble은 1로 해줘 next\_pc에 올바른 분기주소가 들어가도록 한다. 위 두경우가 아니면 pc\_source와 bubble모두 0으로 하여 파이프라인에 인스트럭션들이 정상적으로 흘러가도록 한다. bubble이 1로 되면 그 시점에 IF, ID단계에 있던 두 인스트럭션의 컨트롤 시그널들을 변경해 레지스터와 메모리값을 변경하지 못하도록 한다. (asynchronous하게 변경함)

1. InstMemory module

이 모듈에선 asynchronous하게 주소에 맞는 instruction을 내보낸다. (이렇게 읽은 instruction은 cpu module에서 posedge clk마다 IF/ID 파이프라인 레지스터의 IF\_ID\_inst에 저장된다.)

1. eq\_mux module

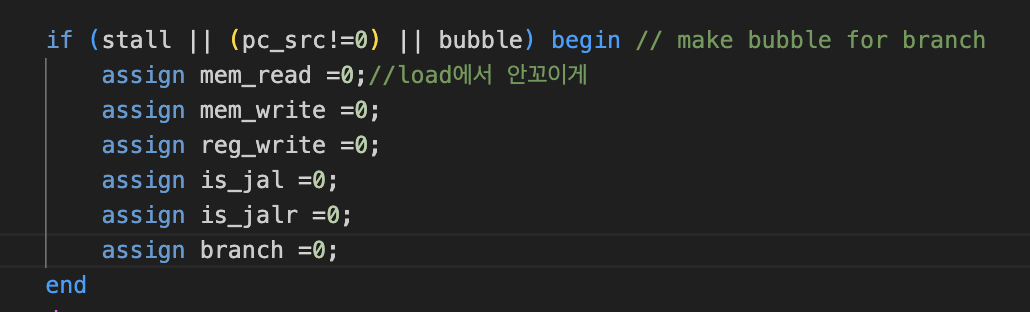
이 모듈에선 ecall instruction일때 rs\_1의 input으로 17을 넣어준다(ecall 아니면 instruction에서 읽은 레지스터 번호가 들어가도록 함)

1. RegisterFile module

이 모듈에선 asynchronous하게 input으로 받은 레지스터 번호에 해당하는 레지스터를 읽어 내보내고, write\_enable이 1일 경우 synchronous하게 해당하는 레지스터에 값을 입력한다.

1. ControlUnit module

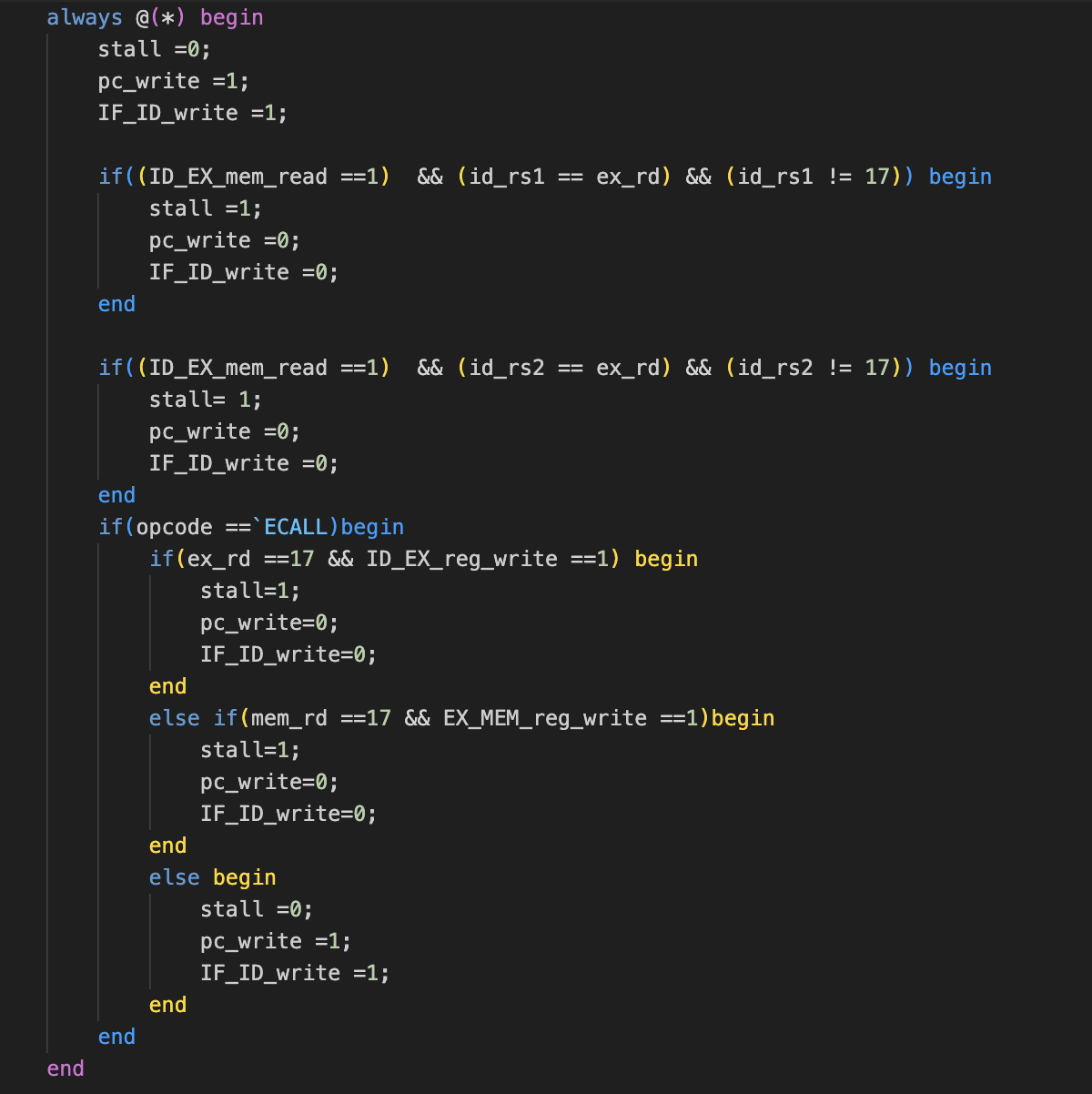
ControlUnit에선 ID단계의 현재 인스트럭션에 맞는 시그널들을 설정해준다. 설정해주는 값들은 이전까지의 랩들과 같다. 추가된 부분은 아래와 같다



lab 4-1에서 load의 raw dependency 구현을 위해 stall 신호가 1로 켜지면 mem\_read, mem\_write, reg\_write를 0으로 해줬었는데, control flow 인스트럭션을 처리하기 위해 bubble과 pc\_surc가 0이 아닐때도 조건에 추가하고, is\_jal, is\_jalr, branch 신호도 0으로 해주었다. pc\_src!=0 조건은 EX단계에서 분기가 결정되었을때 ID단계에 있는 잘못된 인스트럭션이 register와 memory를 변경하지 못하게 하도록 추가되었고, bubble 조건은 IF단계의 instruction이 register와 memory를 변경하지 못하도록 한다. (cpu의 wire bubble이 1로 되면 이는 IF/ID\_bubble을 업데이트하고 IF/ID\_bubble은 control unit의 input인 bubble로 들어가 IF단계에 있던 인스트럭션이 ID단계로 왔을때 bubble로 바꿔준다.)

1. Hazard module

Hazard 모듈에선 asynchronous하게 load isntruction에 관한 RAW hazard와 Ecall instruction에 관한 hazard가 발견될 경우 파이파라인을 멈추기 위해 stall을 1로 해주고, pc\_write와 IF\_ID\_write를 0으로 해준다. 코드는 다음과 같다.

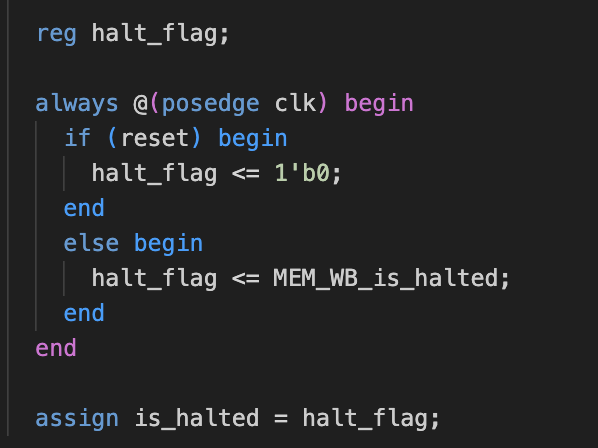


처음 두 if문은 load instruction의 hazard를 검사하는데 ID\_EX\_mem\_read가 1이면 이전 instruction이 load라는 것을 의미하며 id\_rs1과 id\_rs2가 ex\_rd와 같은지 확인하는 부분은 load에서 값을 입력하는 register와 이어지는 instruction이 읽는 register가 같은지 보는것이다. x17 register가 아닌지 확인하는 부분은 ecall instruction인지 아닌지 확인하기 위해서이다.

세번째 if문은 ecall instruction이 ID단계에 있을때 이전 instruction이나 전전 instruction이 x17을 destination register로 다루고 있는 동시에 reg\_write signal이 1로 켜져 있는지 확인한다. 그 이유는 ecall instruction이 불리면 x17 register에 들어있는 값이 10인지 확인해야 하는데 아직 완료되지 않은 instruction들이 x17에 10을 쓰는 과정에 있을 수 있기 때문이다. ecall isntruction이 다른 instruction들과 다르게 data forwarding을 이용할 수 없는 이유는 x17 == 10? 연산이 EX단계가 아닌 ID단계에서 이뤄지기 때문이다.

1. halt module

halt 모듈은 ID단계에서 asynchronous하게 ecall instruction이 불렸을 때 x17레지스터 값과 10이 같은지 비교해 둘이 같다면 is\_halted signal을 1로 켜준다. 이렇게 켜진 is\_halted signal은 cpu module에서 다음 posedge clk때 ID\_EX pipeline 레지스터의 ID\_EX\_is\_halted 레지스터에 전달되며 이후 매 posedge clk마다 다음 파이프라인 register로 전달된다. 최종적으로 MEM\_WB\_is\_halted register가 1이면 cpu module의 output이 is\_halted가 1로 켜져 process가 종료된다.

(cpu module에 구현된 halt)

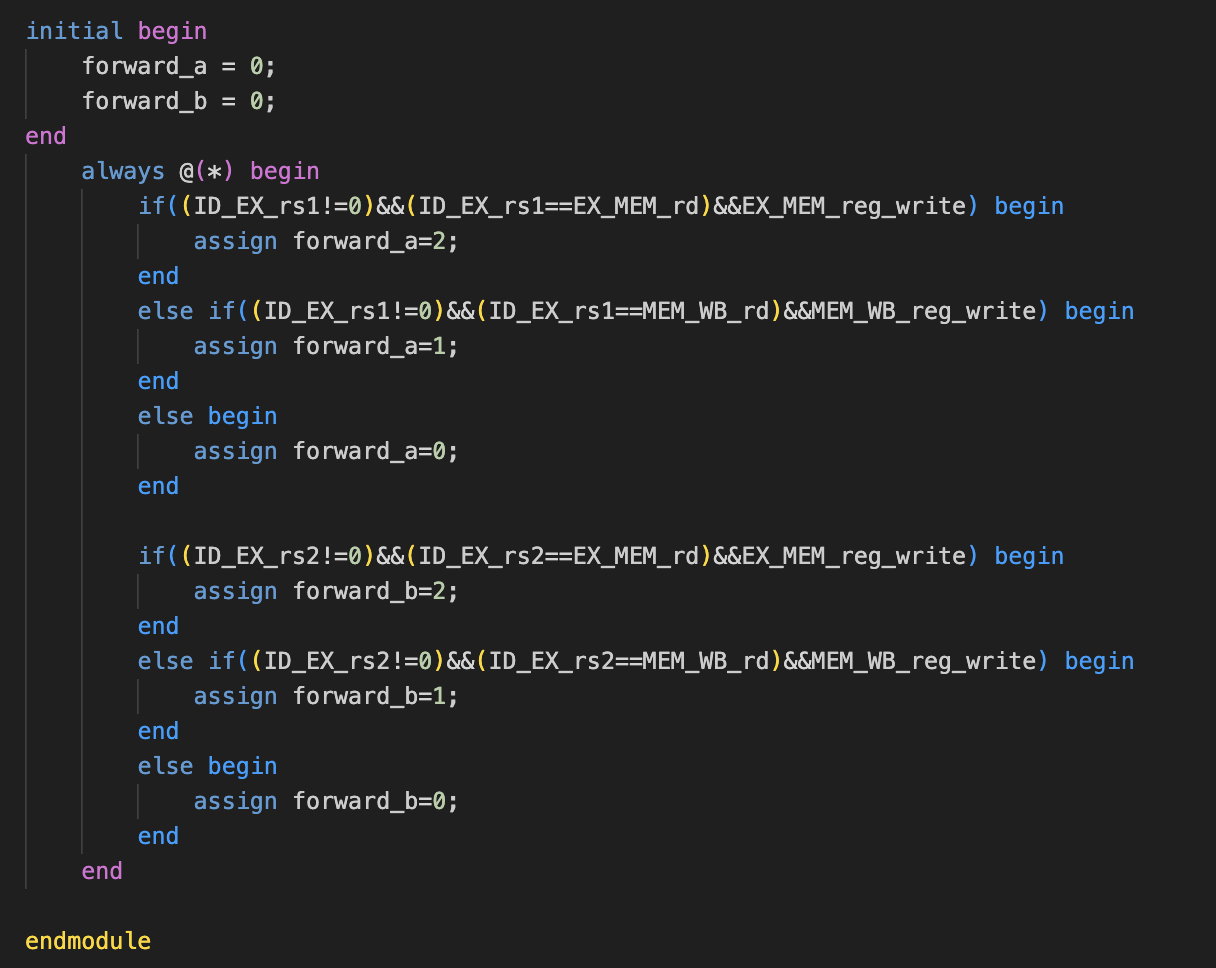
1. ImmediateGenerator module

Immediate를 이용하는 instruction의 경우 올바른 immediate를 생성한다.(asynchronous)

1. ALUControlUnit

single cycle cpu lab과 같다. opcode, funct7, funct3 값을 입력으로 받아 alu에서 수행해야 하는 연산을 나타내는 alu\_op를 ALU에 전달함.(asynchronous)

1. forward\_unit module

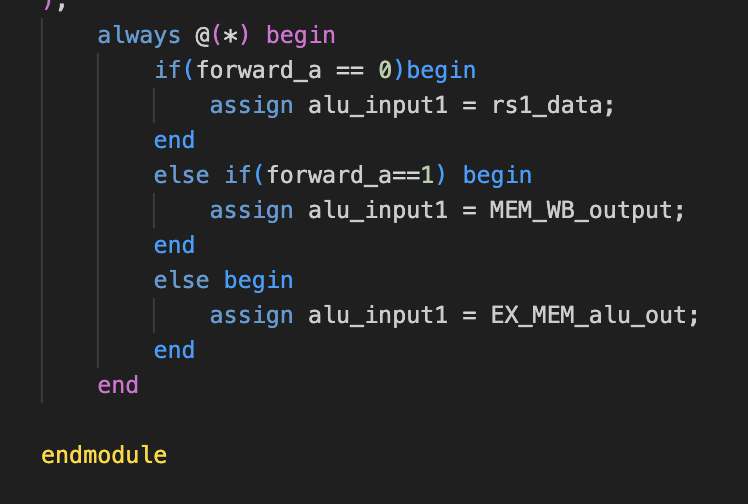


EX단계 instruction이 써야하는 첫번째 레지스터의 인덱스가 MEM단계 instruction이 새로운 값을 써주는 destination 레지스터의 인덱스와 같으면 EX/MEM 파이프라인 레지스터에서 바로 그 값을 EX단계 instruction에 넘겨줄 수 있도록 forward\_a를 2로 해준다(EX단계의 forward\_a mux의 signal로 쓰임). 또한 EX단계 instrucion의 rs1과 과 WB단계 instruction이 새로운 값을 써주는 rd를 비교해 두 인데스가 같을 경우 forward\_a를 1로 한다. 위 두 경우에 해당하지 않으면 forward\_a를 0으로 한다.

위 과정을 동일하게 rs2에도 적용한다. (forward\_b를 결정함)

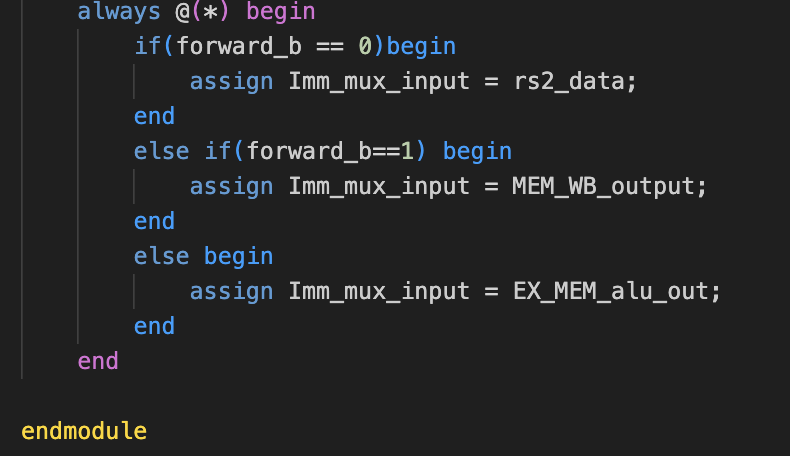
위 과정은 asynchronous하게 구현된다.

1. forward\_a\_mux module



forward\_unit에서 세팅한 forward\_a 시그널을 입력으로 받아 알맞은 값을 alu의 첫번째 input으로 넣어준다. (aynchronous)

1. forward\_b\_mux module



forward\_unit에서 세팅한 forward\_b 시그널을 입력으로 받아 알맞은 값을 imm\_mux의 인풋으로 넣는다. (asynchronous)

1. Imm\_mux module

Imm\_mux 모듈에선 alu\_src시그널을 받아 alu의 두번째 input으로 레지스터에서 읽은 값(forwarding 포함) 과 Immediate 중 무엇을 연결할지 결정한다.

1. ALU module

ALU 모듈은 aynchronous하게 alu control 모듈에서 받은 시그널과 forwarding과정을 거친 두 개의 input을 받아 연산을 수행한 뒤 alu\_result에 값을 할당한다.

1. DataMemory module

DataMemory 모듈은 mem\_read가 1일 경우 asynchronous하게 메모리에서 읽은 값을 dout에 할당하고, mem\_write가 1일 경우 synchronous하게 메모리에 적절한 값을 써준다.

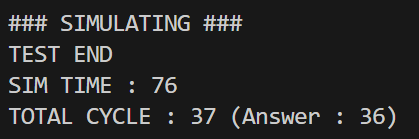
1. mem\_src\_mux module

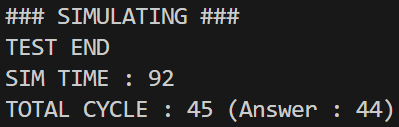
mem\_src\_mux 모듈은 synchronous하게 jump and link 명령어이면 pc+4를 rd 레지스터에 입력하도록 하고, mem2reg 시그널이 1일 경우(for load instruction)메모리에서 읽은 값을 MEM\_output에 연결해 레지스터에 값이 쓰일 수 있도록 하고, mem2reg가 0이면 현재 instruction의 alu\_result를 MEM\_output에 할당한다.

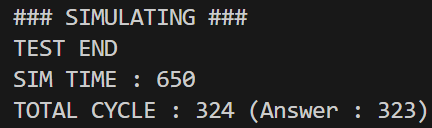
**Discussion**

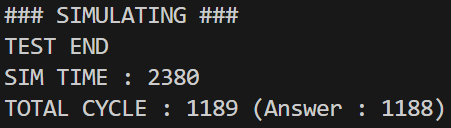
1. stall과 bubble의 겹침 문제
   1. test case를 실행하며 signal 값을 추적하는 과정에서 data haward의 발생과 misprediction으로 인한 bubble이 연달아 일어나는 순간에 의도치 않은 동작을 하게 되는 모습을 확인했다.
   2. 원인은 control unit에 있었다. lab 4-1의 control unit을 그대로 사용했는데, controlflow 기능을 구현하는 과정에서 is\_jal, is\_jalr, branch signal이 추가되었고 stall 또는 bubble이 활성화 될 때 앞서 말한 signal도 0으로 처리해주지 않아 문제가 발생하였다. 이를 0으로 처리해주자 올바르게 동작하였다.

**Conclusion**

 basic

 ifelse

 loop

 recursive

각 test case를 수행한 결과이다. cycle은 정답과 같게 도출되었다. 본 실습에서는 가장 기본적인 branch prediction인 always not taken을 이용하였다. 하지만 2-bit global prediction과 같이 보다 복잡한 prediction 기법을 이용한다면 bubble로 인한 cycle 손실을 줄여 나은 결과를 도출할 수 있을 것으로 예상한다. 특히 loop case처럼 같은 분기가 반복적으로 일어나는 경우에 더욱 유리할 것이다.