**CSED-311 LAB4-1 REPORT**

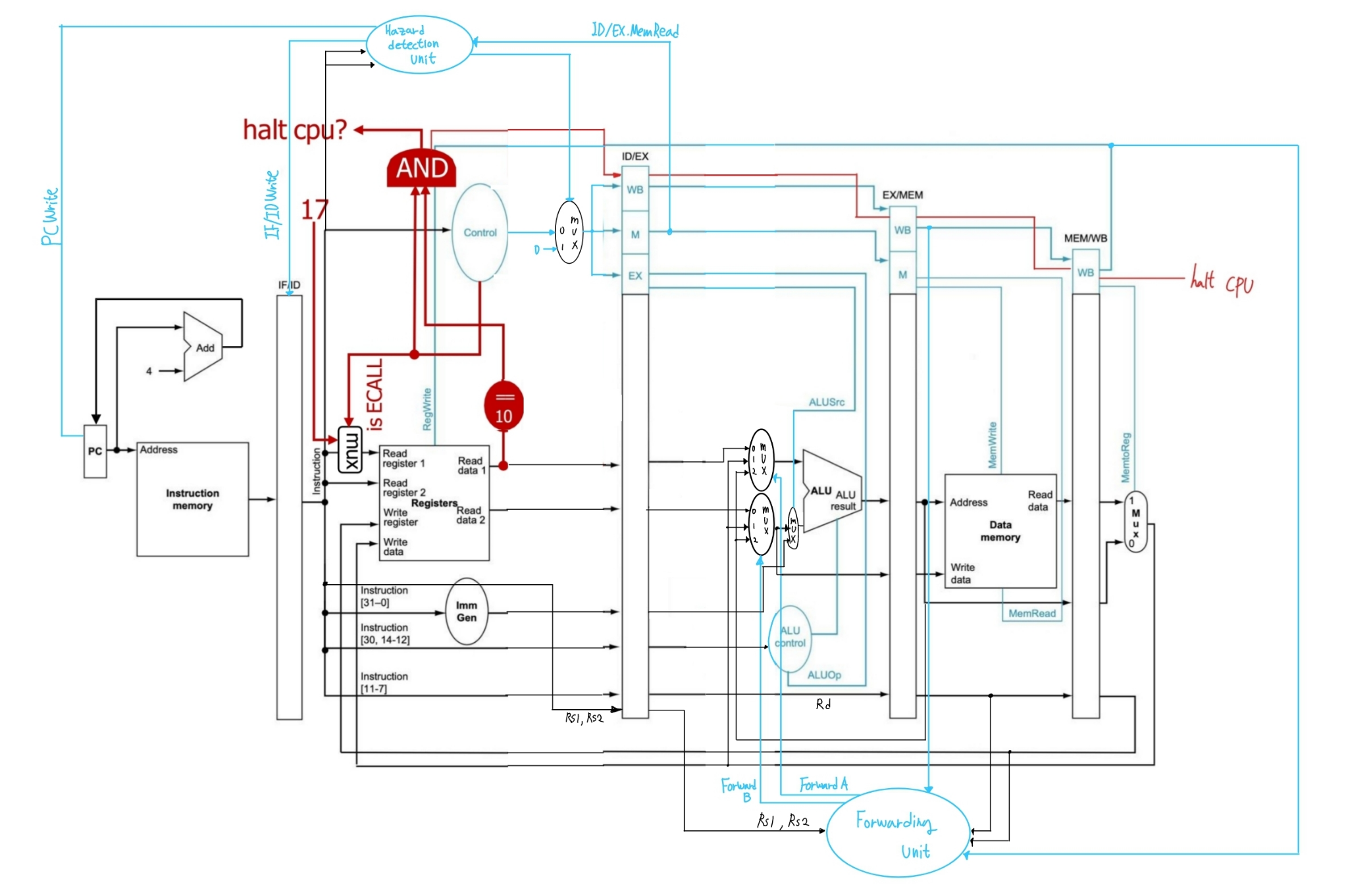
김모세

표승현

**Introduction**

Verilog를 이용해 pipelined CPU를 구현하는 것을 목표로 한다. Pipelined CPU는 multi cycle CPU와 같이 여러 사이클에 걸쳐 instruction을 수행한다. 하지만 각 stage가 각 사이클마다 다른 instruction을 수행할 수 있게 하여 IPC를 증가시켰다는 차이점이 있다.

**Design**



<Design of non controlflow pipelined CPU>

이번 실습에서는 controlflow를 고려하지 않은 pipelined CPU를 구현하였다. 구조는 교안의 회로도를 기반으로 하였다. 추가적으로 Data forwarding을 구현하기 위한 forwarding unit과 mux를 만들었다. 더불어 Halt 시그널을 정상적으로 인식하고 수행하기 위해 mux와 게이트를 통해 적절한 순간에 레지스터 값을 비교할 수 있도록 했다. 또한 halt 시그널을 WB/MEM 단계까지 전달할 수 있도록 설계하여 아직 완료되지 않은 instruction이 모두 수행된 후에 halt되도록 하였다.

**How does our pipelined CPU work?**

pipelined CPU는 한 사이클에 모든 stage가 각각의 instruction을 처리할 수 있게 하여 같은 사이클 동안 더 많은 명령을 수행할 수 있게 설계된 CPU이다. 이를 위해 각 stage에는 이전 stage의 정보와 시그널을 저장하는 register를 갖고 있으며 이로부터 전달 받은 값을 이용해 instruction을 이어서 수행한다. 예를 들어 rd 정보의 경우 WB 단계에서 어떤 register에 값을 저장할지 판단할 때 사용되므로 마지막 단계까지 rd값을 전달해야 한다. 때문에 ID stage부터 WB stage까지 wire를 통해 rd를 전달하는 모습을 확인할 수 있다.

다만 일부 data hazard의 경우 pipelined CPU를 stall 해야한다. Instruction에서 사용하는 register가 이전 instruction에서 memory에서 값을 읽어야 하는 경우 Data memory에서 값을 read할 수 있도록 stall하는 기능이 필요하다. 이를 Hazard detection unit으로 구현하여 ID\_EX\_mem\_read가 활성화 되어 있고, rd가 rs1 또는 rs2와 같을 때 PC 및 IF\_ID register를 업데이트 하지 않도록 하고 모든 시그널을 0으로 해준다.

**Implementation**

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. 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

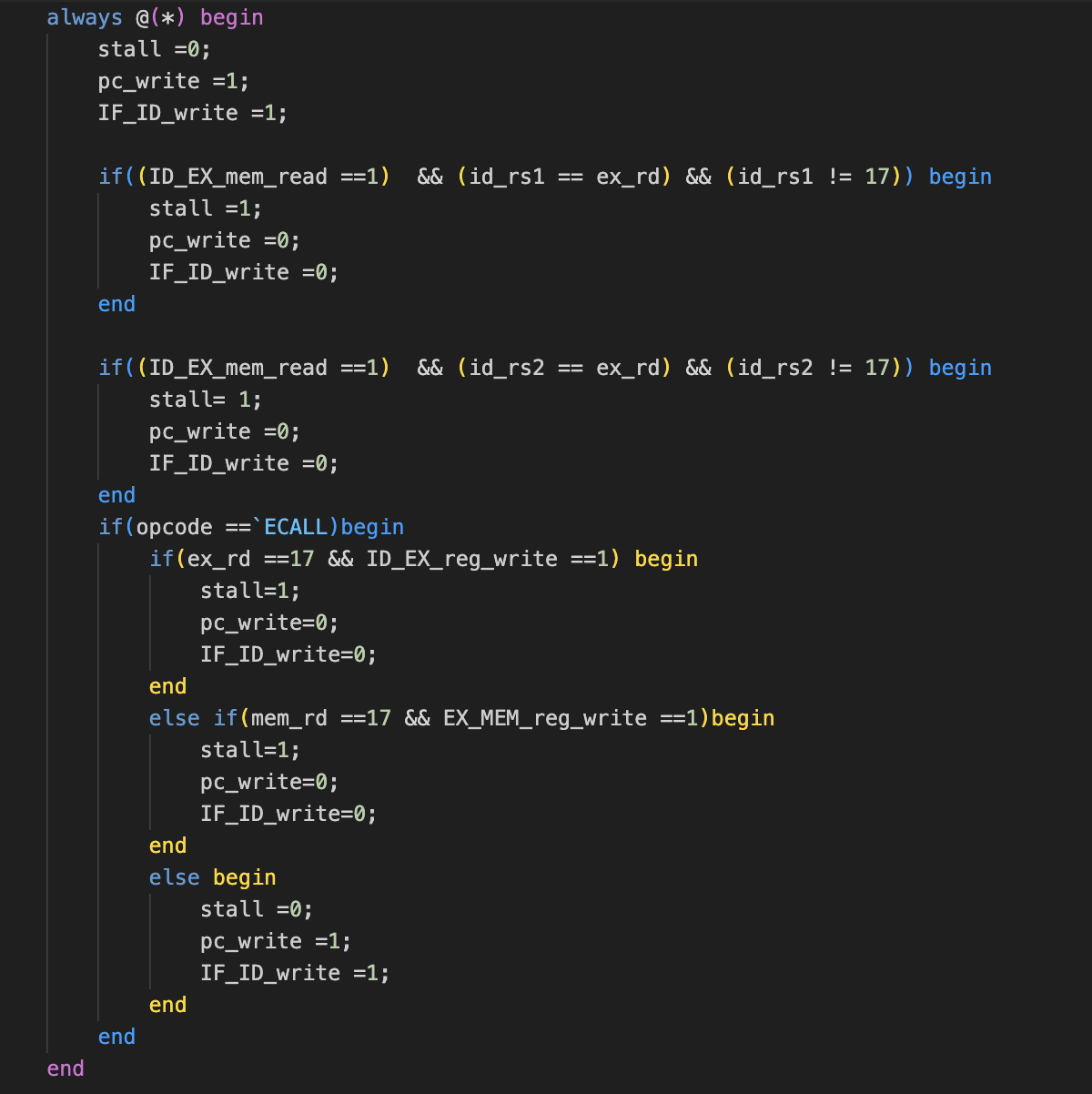
이 모듈에선 asynchronous하게 input으로 받은 레지스터 번호에 해당하는 레지스터를 읽어 내보내고, write\_enable이 1일 경우 synchronous하게 해당하는 레지스터에 값을 입력한다.

1. ControlUnit module

ControlUnit에선 instruction type에 맞게 signal들을 세팅한다. non-branch instruction만 다루므로 Branch, Jal ,Jalr instruction에 관한 부분들은 주석차리 하였으며 non-branch instruction에서 1로 켜주는 시그널들은 이전까지 lab과 동일하다. pipeline cpu 구현을 위해 추가된 부분은 stall에 관련된 부분인데, control unit의 input으로 들어오는 stall이 1로 되어있을 경우, 현재 ID단계에 있는 instruction은 레지스터, 메모리의 값을 변경해선 안되므로(bubble) output signal 중 mem\_write, reg\_write, mem\_read을 반드시 0으로 해준다.

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.

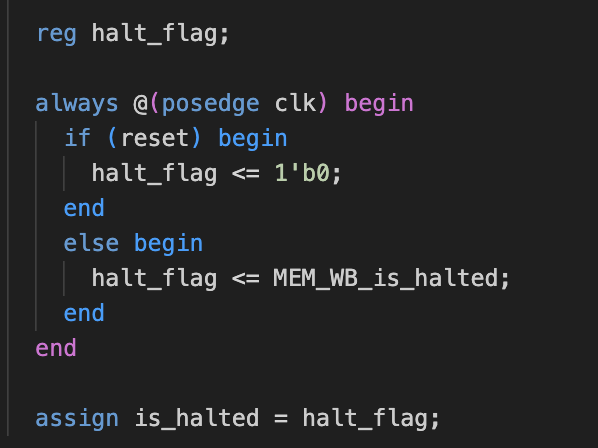


처음 두 if문은 load instruction의 hazard를 검사하는데 ID\_EX\_mem\_read가 1이면 이전 instruction이 load라는 것을 의미하며 id\_rs1과 id\_rs2가 ex\_rd와 같은지 확인하는 부분은 load에서 값을 입력하는 register와 이어지는 instruction이 읽는 register가 같은지 보는것이다. x17 register가 아닌지 확인하는 부분은 ecall instruction인지 아닌지 확인하기 위해서이다.

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)

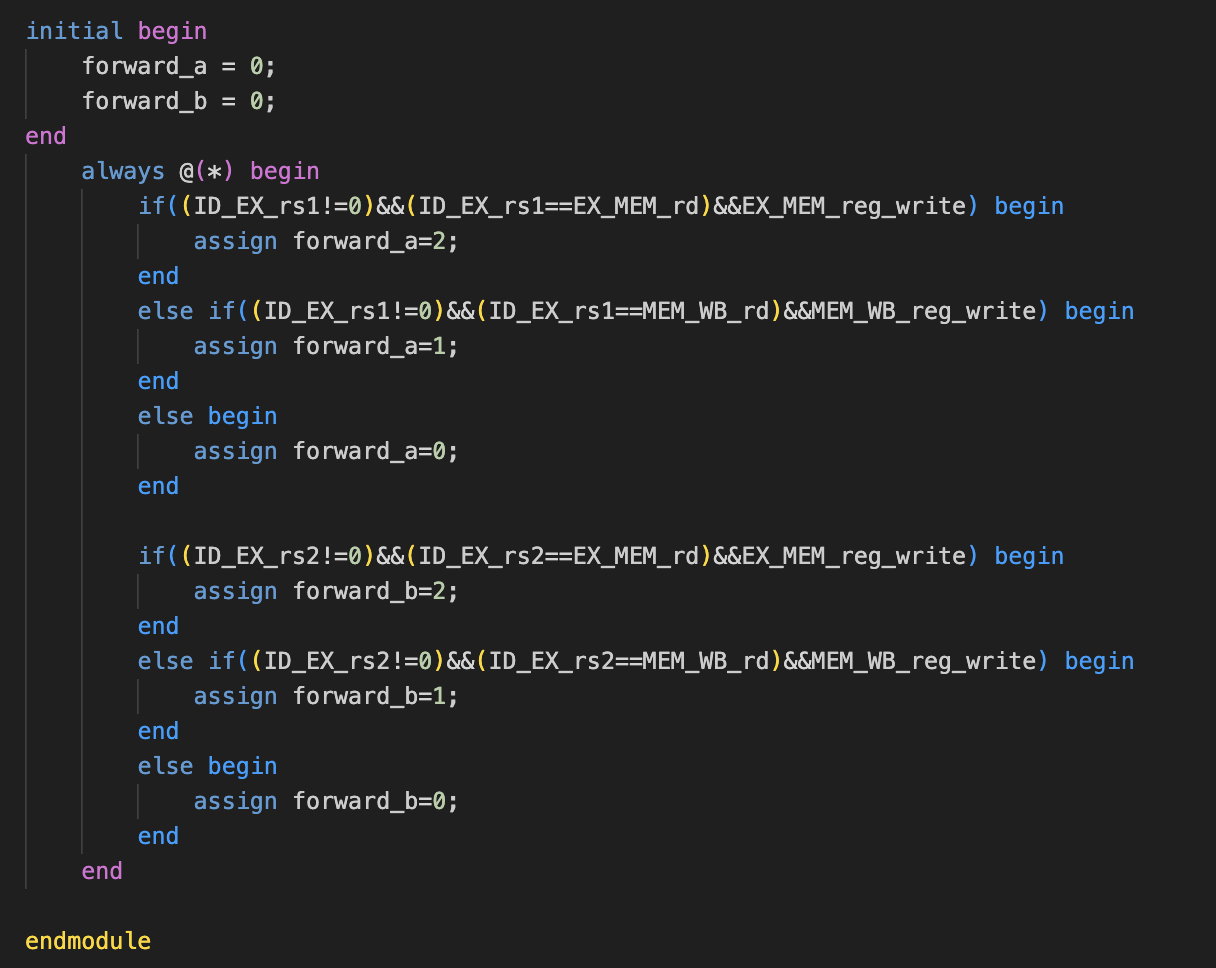
1. ImmediateGenerator module

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

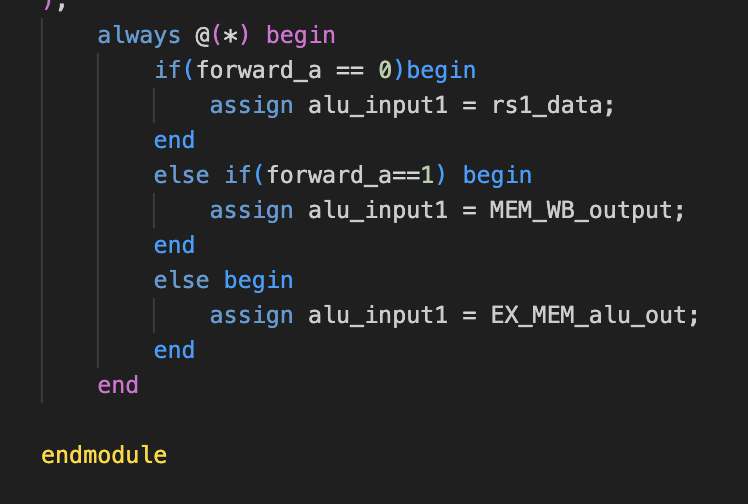


EX단계 instruction이 써야하는 첫번째 레지스터의 인덱스가 MEM단계 instruction이 새로운 값을 써주는 destination 레지스터의 인덱스와 같으면 EX/MEM 파이프라인 레지스터에서 바로 그 값을 EX단계 instruction에 넘겨줄 수 있도록 forward\_a를 2로 해준다(EX단계의 forward\_a mux의 signal로 쓰임). 또한 EX단계 instrucion의 rs1과 과 WB단계 instruction이 새로운 값을 써주는 rd를 비교해 두 인데스가 같을 경우 forward\_a를 1로 한다.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

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

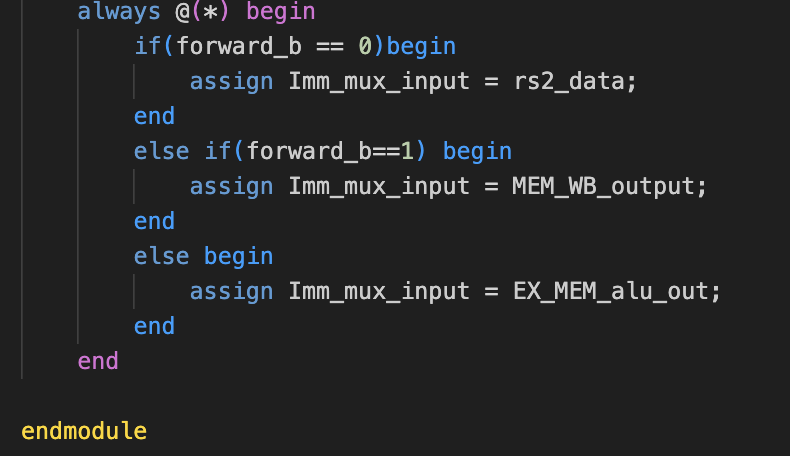
위 과정은 asynchronous하게 구현된다.

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.

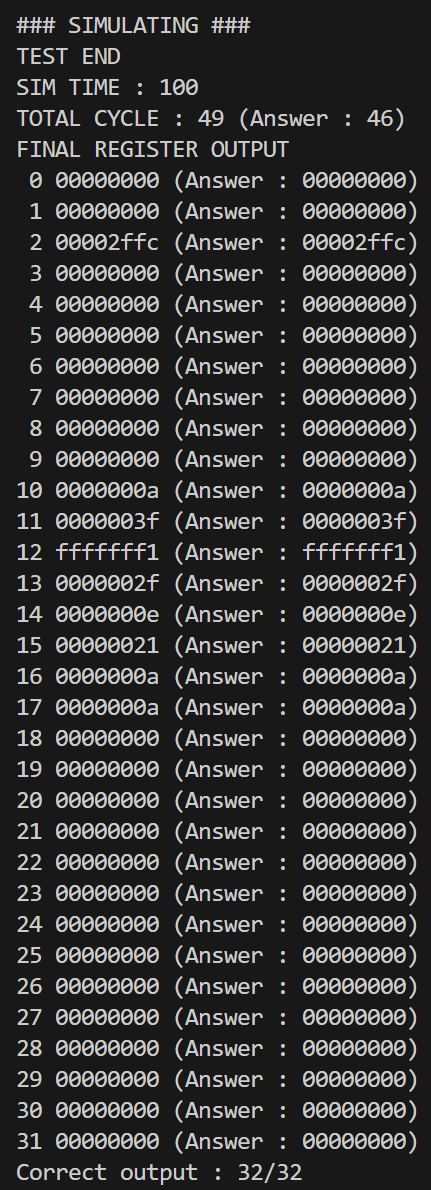
1. mem\_src\_mux module

mem\_src\_mux 모듈은 synchronous하게 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**

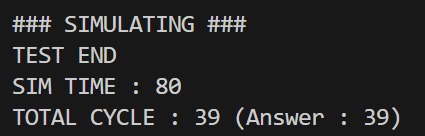
1. Forward B mux와 immediate mux의 순서
   1. Data forwarding을 구현하는 과정에서 EX stage의 mux 배치 순서에 관해 논의했다. 어떤 mux를 마지막에 두는지에 따라 작동 결과가 상이하기 때문에 이를 결정하는 것은 중요하다.
   2. Forwarding B mux가 마지막에 배치되는 경우 immediate value가 사용되는 instruction임에도 불구하고 forwarding 연산 결과가 ALU input을 들어갈 수 있다. 때문에 forwarding B mux를 먼저 통과하고 immediate mux를 그 다음에 통과하게 한다. Forwarding 조건이 성립해 forwarding B mux가 다른 값을 output으로 내보내더라도 immediate value가 필요한 경우 ALUsrc 시그널을 받아 immediate value를 ALU input으로 전달할 수 있기 때문이다.
2. is\_ecall의 stall 처리
   1. is\_ecall 시그널을 받았을 때 stall을 해줘야 한다. is\_ecall 시그널을 받고 halt 시그널을 활성화 시키기 전 x17에 10을 넣고 이를 확인하기 때문이다. halt 시그널이 활성화 되는 단계는 ID stage이기 때문에 data forwarding으로 이를 해결할 수 없다. 더불어 ECALL instruction은 rs 정보가 없기 때문에 다른 instruction과 같은 조건문을 통해 stall 시그널을 줄 수 없다. 따라서 Hazard detection unit에서 ecall을 예외적으로 처리하여 적절하게 stall 시그널을 줄 수 있도록 하였다.

**Conclusion**



모든 레지스터가 목표한 값에 도달한 모습을 확인할 수 있다.

**Compare total cycles between the single cycle and pipelined CPU**



single cycle CPU 환경에서 똑같은 instruction set를 수행한 결과이다. 총 39개의 instruction으로 single cycle은 총 39 사이클이 소요되었다. Pipelined CPU의 경우 best case일 때 한 사이클에 한 instruction을 수행할 수 있지만 data hazard가 발생하는 경우 stall이 불가피하다. 이로 인해 조금 더 많은 49 사이클이 소요 되었다. 하지만 각 사이클 시간이 pipelined CPU의 것이 훨씬 짧기 때문에 성능 면에서는 pipelined CPU가 더욱 우수하다.