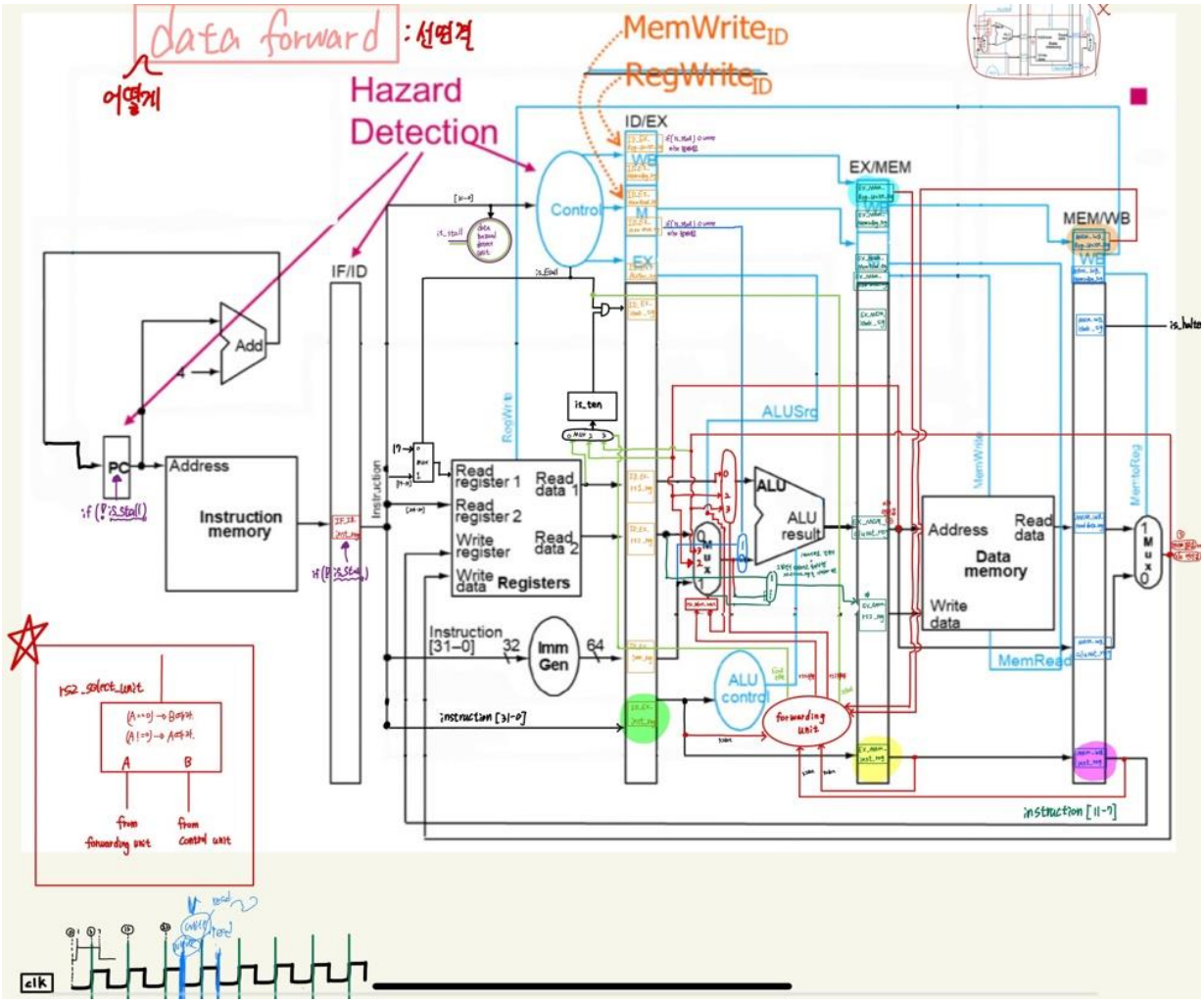
Pipelined CPU without control flow instructions

20210115 이세규, 20210706 이지원

1. introduction

이번 과제의 목표는 Vivado를 통해 pipelined CPU without control를 구현하며 pelined CPU와 그로 인해 발생하는 data hazard를 stall과 data forwarding으로 통제하는 방식에 대해 정확히 이해하는 것이다. 주어진 스켈레톤 코드와 LAB2의 single-cycle CPU의 구조도를 기반으로 Modularization을 중심으로 구현하였다.

2. design



전체적인 구조는 싱글 사이클 CPU와 유사하고, 레지스터를 이용하여 값을 저장하고 있는 방식은 멀티 사이클 CPU에서 사용한 방식을 활용했다. IF, ID, EX, MEM, WB stage로 나뉘어 매 사이클마다 n번째 연산은 다음 연산의 단계로 진행되고 동시에 n-1번째 연산은 n번째 연산이 있었던 stage로 들어온다. 이때 모든 cpu의 IF, ID, EX, MEM, WB stage가 각각의 인스트럭션에 대해 연산을 수행하므로 cpu자원이 낭비되지 않고 높은 정도로 활용되어 기존의 싱글 사이클 CPU보다 훨

씬 효율적이다. 수업시간의 표현을 빌리자면 이러한 구현은 하드웨어의 활용도를 높여 throughput을 향상시킨 것으로 해석할 수 있다. 중간과정을 저장할 수 있는 레지스터를 배치하여 각 단계에서 출력된 값을 저장하였기 때문에 특정 시점에 여러 개의 연산이 CPU의 서로 다른 부분에서 동시에 이루어질 수 있다. 이때 레지스터는 Lab3의 멀티 사이클 CPU와 달리 각 stage 사이마다 존재하여 확실히 구분되어 있다는 것이 큰 특징이다. pipelined cpu는 여러 개의 인스트럭션들이 동시에 진행된다는 특징을 가지는 것이 효율면에서는 장점이다. 하지만 이러한 장점을 만들기 위해서 넘어야 할 벽이 하나 있다. 기존에는 프로그램의 각 연산이 겹치지 않고 순차적으로 실행되는 것이 보장되어 있었다. 이 말인 즉, 각 연산이 동일한 시점에 cpu에 머무르는 일이 없다는 것이다. 따라서 n번째 연산은 n-1번째 이하의 연산의 결과를 문제없이 사용할 수 있다. 하지만 pipelined cpu에서는 n번째 연산과 n-1번째 연산이 동일한 시점에 다른 단계로 진행될 수 있다. 따라서 n번째 연산에서 필요한 값이 n-1번째 연산에서 처리 중이어서 정상적으로 접근할 수 없는 경우가 생긴다는 점이 문제로 떠오른다. 이러한 문제가 data hazard이고 강의 시간에 배운 data forwarding을 활용하여 CPU의 어느 단계에 값이 존재하기만 하면 해당 값이 필요한 단계로 값을 미리 가지고 오는 식으로 문제를 해결하지만, 연산의 특징을 미루어 봤을 때 data forwarding 방법으로도 해결할 수 없는 경우가 생긴다. 이 때, stall을 하게 되며 상위 연산은 진행되도록 하고 younger 연산은 멈추도록 하여 older 연산이 올바른 값을 만들어 낼 시간을 준다. 이후 값이 준비가 되었다고 하면 즉각적으로 forwarding 하여 이후의 연산을 진행한다. load와 ecall의 경우에서는 forwarding만으론 해결될 수 없는 data hazard가 생기는데 이는 implementation에서 자세히 알아보겠다.

우리의 pipelined cpu가 어떤 식으로 동작하는지 첨언하자면, 위 그림에서 볼 수 있듯이 IF, ID, EX, MEM, WB 단계마다 사이에 레지스터를 배치하여 하나의 연산을 기준으로 했을 때 매 사이클별로 다음 단계로 진행될 수 있게 하였고 여러 연산이 동시에 진행된다는 관점에서 봤을 때 레지스터를 배치함으로써 서로 간섭하지 않고 독립적으로 작동할 수 있게 하였다. 모든 pipelined register는 posedge에 업데이트 되고 특정 단계에서 만들어낸 유용한 정보를 가지고 있는다. 이런 방식으로 pipeline의 기본 구조를 만들었고, stall 신호를 만드는 unit, data forwading을 위한 선 연결이 추가적으로 구현되어 있어 해결 가능한 data hazard를 다루었다.

3. implementation

이번 Lab4에서는 jal, jalr, branch를 구현하지 않으므로 다음 PC의 값으로 PC+4만을 갖는다는 것이 차이점이고, 이외에는 IF stage에서 기존의 하는 일과 거의 다르지 않다. IF stage는 PC의 instruction을 레지스터에 넘기는 것으로 정리되기에 매우 간단히 구현하였다.

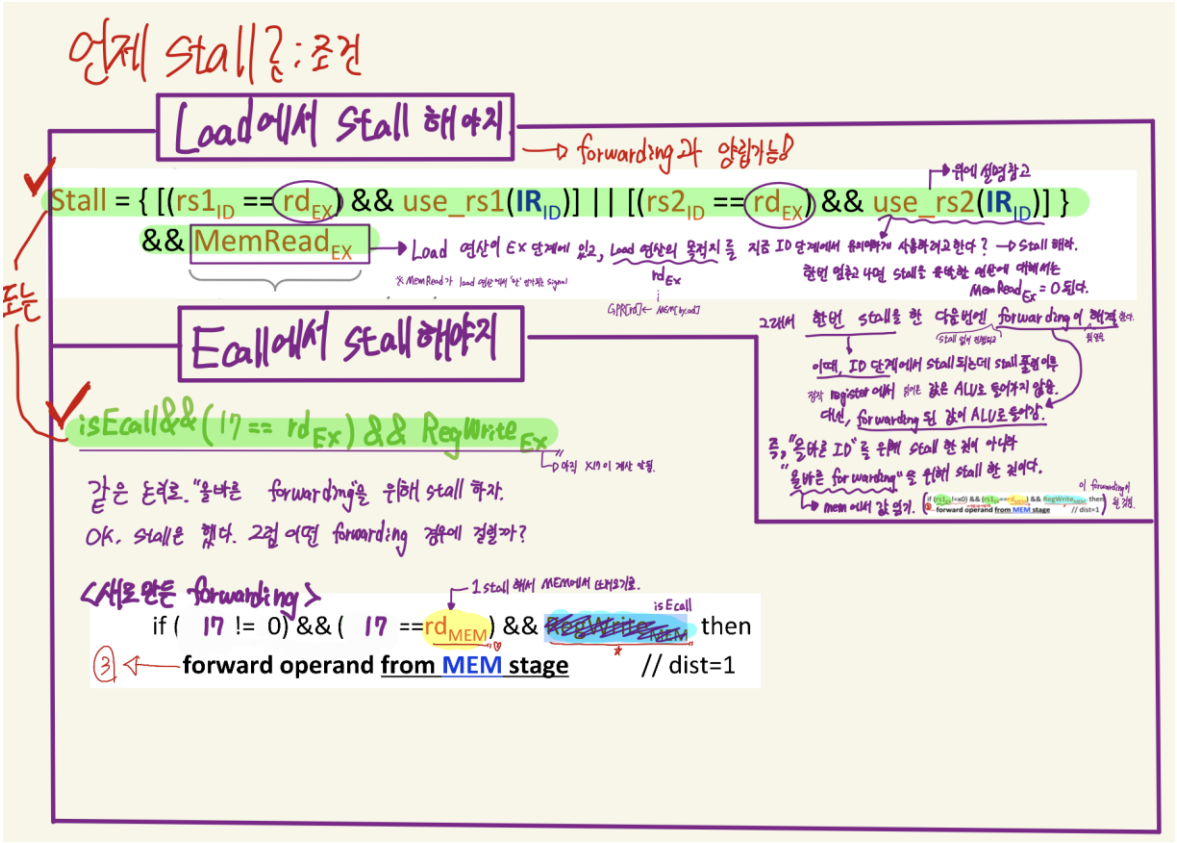
ID stage에서는 IF 단계에서 가져온 연산을 분석하여 레지스터 값 가지고 오기, immediate값 만들기, control 신호 만들기, data hazard detect unit 출력 만들기를 한다. 이렇게 만들어진 값은 해당 연산의 다음 단계인 EX단계를 위해, 그리고 현재 IF단계에 있는 연산이 뒤따라 처리되어도 문제가 생기는 것을 막기 위해 레지스터에 저장된다. Ecall에 대한 부분은 이전 Lab에서의 구현과 동

일하게 Ecall의 신호가 오면 rs1으로 17 레지스터를 읽어 10인지 확인하였고, 앞서 진행되고 있던 명령어가 잘 마치고 halt되도록 WB stage가 될 때까지 값을 레지스터로 넘기는 형식으로 구현했다. Control Unit에서의 연산도 기존의 것들과 다른 것이 거의 없고, JAL, JALR, BRANCH에서만 사용하는 시그널은 구현하지 않았다는 점이 차이이다.

- hazard detection과 stall

Data hazard detect unit은 여러 stage의 값들을 받아와 현재 ID stage에서 진행중인 명령어가 data hazard에 의해 stall해야 하는지 여부를 결정한다. Stall이 될 경우, pc값이 업데이트 되지 않고, IF/ID pipelined register가 업데이트 되지 않으며, ID/EX pipelined register에 있는 regwrite signal과 memwrite signal이 0이 된다. 즉, ID단계 이후의 연산은 진행되고 ID단계와 IF단계에 있는 연산이 각각 다음 단계로 가지 않는 것으로 생각할 수 있다. ID단계에서부터 비롯된 정보를 통해 실제로는 연산은 되겠지만 그 값을 singal을 조작하여 반영하지 않으므로 stall된 것과 동일한 효과를 낸다.

우리는 data forwarding을 구현했으므로 load와 ecall의 특정 경우에서만 hazard가 발생하고, stall이 필요하다. hazard발생 시 stall을 통해 해소하므로, stall조건이 결국 hazard detect조건임을 짚고 설명을 하겠다. 이에 대한 stall의 조건은 강의자료를 기반으로 확장하였으며 다음과 같다.



data forwarding을 구현하였기 때문에 stall을 하는 경우는 Load와 Ecall에서 두 가지로 나뉜다. data forwarding이 없는 구현에서와 data forwarding이 있는 구현에서 stall은 그 의미가 다르다는

점을 알아야 한다. 우선 data forwarding이 없는 구현에서는 stall을 하는 이유는 '레지스터 파일에 올바른 값을 저장하는 것을 기다리기 위해서' 이다. data forwarding이 없기 때문에 모든 data는 register file에서부터 가지고 오기 때문이다. 하지만 data forwarding이 있는 구현에서 stall을 하는 이유는 'cpu의 어느 단계이든 좋으니 값이 생기는 것을 기다리기 위해서'이다. data forwarding을 위한 선 연결과 신호는 위의 논리에 따라 구성하는 것이고 이 논리를 뒷받침하는 근간을 짚은 것이다. 이제 조건에 대해 풀어서 설명해보겠다. 현재 **Load연산이 EX단계에 있고**(MemRead EX: 해당 신호는 Load연산일때만 1이 인가되기에 Load연산임을 판단하는 신호로 활용), **Load연산의 목적지**(rdEX)**를 지금 ID단계에서 유의미하게 사용하려고 한다**(rs1 ID와 use_rs1(IR ID))**면 Stall 할 것.** 으로 해석할 수 있다. rs2에서도 rs1과 동일한 논리를 적용하면 rs1이 rs2로만 바뀐 구조가 같은 식이 나오고 이 둘을 '또는' 연산으로 묶어준다.

이와 유사한 논리가 Ecall 에서도 사용된다. 올바른 forwarding을 위해, Ecall연산일때만 1이 인가되는 isEcall, 필요한 값인 17번째 레지스터가 rd EX와 같고, EX단계에서 연산된 값이 이후에 레지스터로 쓰여지는 경우 stall을 한다. 즉, Ecall은 17번째 레지스터가 바뀔 것인데, 아직 어떤 값으로 바뀔지 결정되지 않은 경우 stall을 한다. 이 외의 경우에는 data forwarding을 위해 만들어 둔 경로에 의해 올바른 값이 해석된다.

마지막으로 Load와 Ecall에서 만 stall이 발생하는 이유를 알아보자. 우선 I type(load제외), R type 등의 연산의 경우 ex단계에서 값을 생성하며, 이 때 값이 필요하다. 이러한 특성 아래에서 한 사이클에 하나씩 연산이 처리되기 때문에 연산에 필요한 값은 cpu에 존재하긴 한다. 따라서 forwarding을 통해 data hazard를 해결할 수 있는 것이다. 반면에 load는 mem단계에서 값이 생성되고 ecall에서는 id단계에서 값을 필요로 한다. 이러한 차이가 load와 ecall에서 stall이 생기는 경우가 발생하는 이유이다.

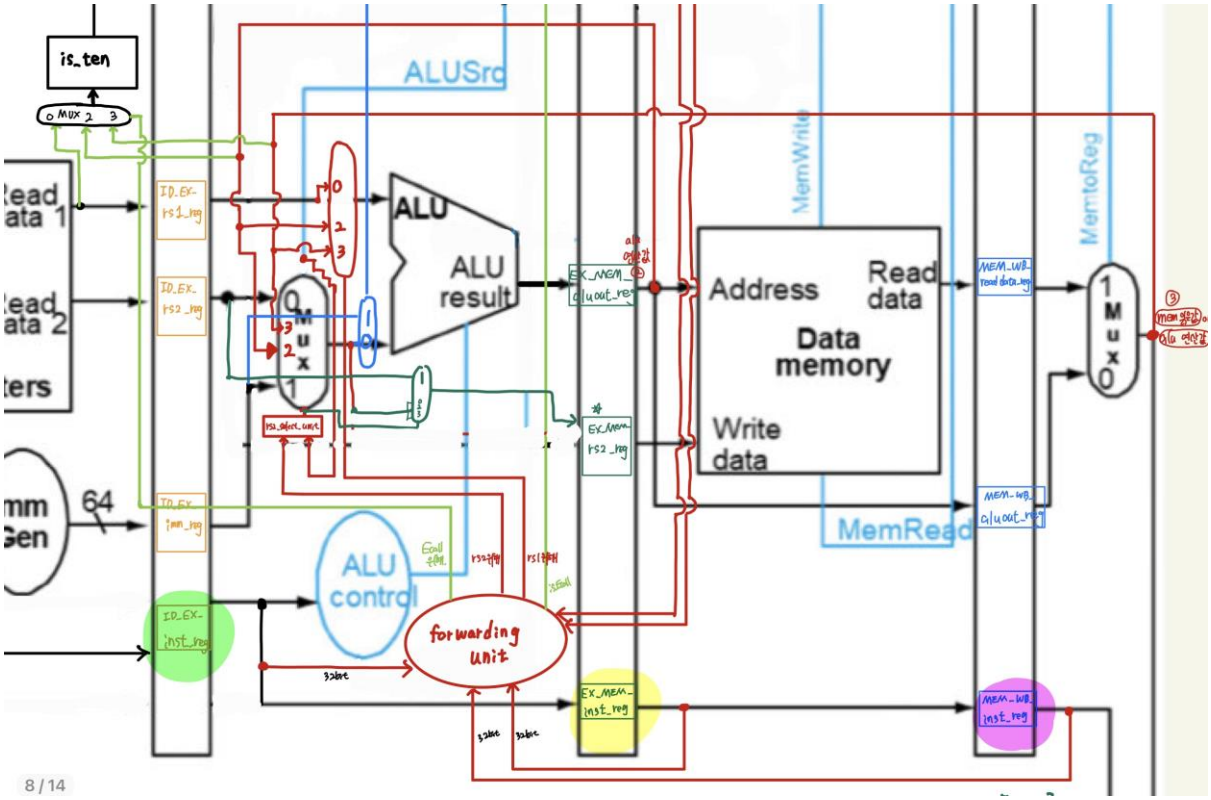
mem단계에서 값이 생기는 load의 경우 load연산보다 더 나중에 온 연산을 살펴 stall을 결정하고 id단계에서 값을 필요로 하는 ecall의 경우 이전 연산을 살펴 stall을 결정한다. 즉, load가 원인이 되어 이후의 연산에 지장이 되기 때문에 stall을 하는 것이고, ecall은 ecall연산을 정확하게 하기 위해서 stall을 하는 것이다. 이런 특징은 load 연산이 ex단계에 있는 시점에서 id단계에 있는 연산을 살피는 것과 isecall 연산이 id단계에 있는 시점에서 ex단계에 있는 연산을 살피는 stall조건에서도 드러나기도 한다.이를 직접 구현한 코드를 보자.

```
  always @(*) begin                                    //rs1은 JAL에서만 안씀. JAL이 아니면 rs1을 사용하는 것
임.                 //rs2는 R S B 에서 씀.
      if(
           (
           ( (part_of_inst[19:15]==ID_EX_inst[11:7]) && (part_of_inst[6:0]!=7'b1101111) )
           ||( (part_of_inst[24:20]==ID_EX_inst[11:7]) && ( (part_of_inst[6:0]==7'b0110011) ||
(part_of_inst[6:0]==7'b0100011) || (part_of_inst[6:0]==7'b1100011)  ) )
           )
           && ID_EX_mem_read
         )//if
          is_stall <=1;
       else if(
                is_Ecall&&(ID_EX_inst[11:7]==5'd17)&&ID_EX_reg_write
                )
           is_stall<=1;
       else
          is_stall <=0;
```

use_rs1, use_rs2는 각각 rs1과 rs2가 실제로 사용되었는지를 묻는 함수로, 0 레지스터에 접근하였는지와 해당 인스트럭션이 rs1이나 rs2를 사용했는지를 체크해야 한다. rs1은 JAL에서만 사용하지 않으며, rs2는 R type, S type, B type에서만 사용한다. 이를 opcode를 통해 확인하여 체크했다. stall 조건이 만족되지 않는 경우에만 1을 인가하여 stall되도록 한다.

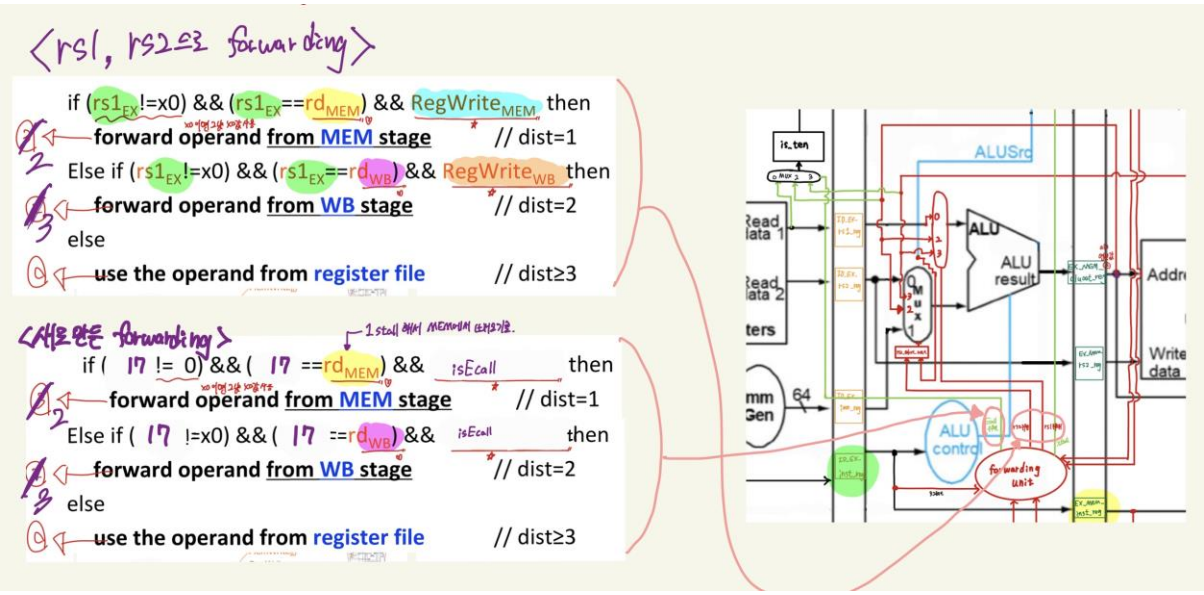
- data forwarding

EX stage에서는 data forwarding에 대한 부분이 들어간다는 점이 기존의 CPU들과 큰 차이점이다. data forwarding유닛 역시 다른 stage에서 받아온 값들로 forwarding의 여부를 결정한다. Forwarding의 여부를 결정하였다면 forwarding unit은 0,2,3 중에 값을 Mux의 select값으로 보내주는데, 0은 forwarding이 일어나지 않는, 즉 이전 인스트럭션이 준 값들로 정상적으로 진행하면 되는 경우를 의미하고 2는 MEM stage에 존재하는 값을 forwarding 하는 경우를 의미하며 3은 WB stage에 존재하는 값으로 forwarding이 이루어지는 경우를 의미한다. 이에 대한 조건은 강의 자료를 참고하여 확장하였으며 관련된 논의를 해보자.



우선, forwarding을 위해서는 data가 전달될 수 있는 경로가 필요하다. 이는 위 그림에서 빨간색 선으로 표현하였다. 요약하자면 ex_mem_aluout_reg의 값과 wb단계에서 mux의 결과를 alu의 입력으로 포워딩하는 것이 전부이다. 구현의 편의를 위하여 ex_mem_aluout_reg 값에 2라는 테그를, wb단계의 mux 결과에는 3이라는 테그를 달았다. 이는 mux의 select와 직접적으로 연관되어 보기에 쉽다. 이렇게 기본적인 data forward 경로를 만들었고, 이제 할 것은 '언제' 어떤 값을 선택할 지 결정하는 것이다. 이 역할을 forwarding unit이 해준다. forwarding unit의 논리를 아래의 그

림과 함께 살펴보자.



보고서의 이전 장에서 stall을 통해 값이 존재하는 경우라면 레지스터 파일에 wb하기 전에라도 언제든지 미리 가져올 수 있도록 하였고 앞에서 선 연결 또한 마쳤다. 이제 정확한 데이터를 선택해주면 된다.

EX단계에 있는 rs1이 MEM단계에 있는 rd와 같고 MEM단계에 있는 rd가 register file로 write back 될 예정이라면 rs1은 wb될 값 사용해야 한다. 따라서 이 경우 forwarding unit은 2의 출력을 주어 MEM단계에 있는 값으로 이후의 연산을 진행할 수 있게 한다. 이는 forwarding하는 거리가 1인 경우에 해당한다.

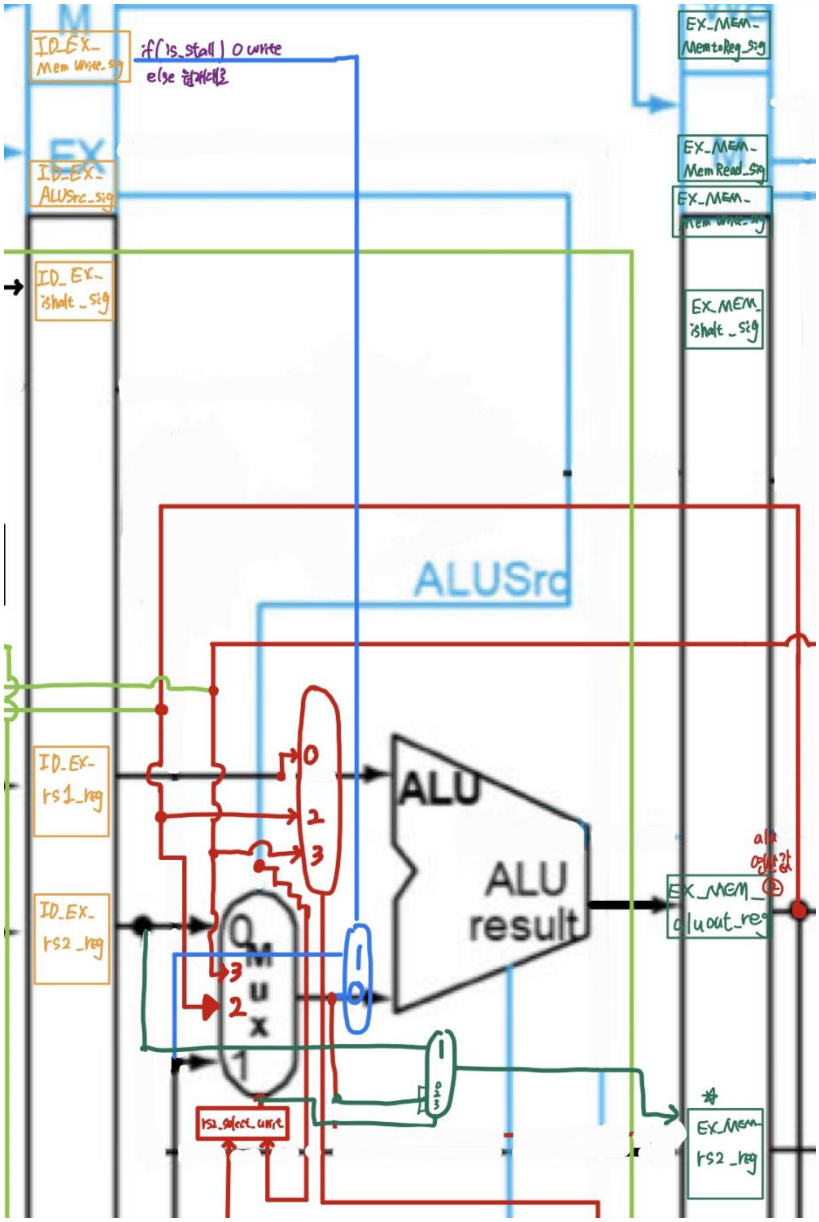
거리가 2인 경우를 살펴보면, EX단계에서 사용하려고 하는 rs1이 WB단계에 있는 rd와 같고 WB단계에 있는 rd가 register file로 write back될 예정이라면 rs1은 wb될 값을 사용해야 한다. 따라서 이 경우 forwarding unit은 3의 출력을 주어 WB단계에 있는 값으로 이후의 연산을 진행할 수 있게 한다.

이 밖의 경우는 연산 사이의 거리가 3이상인 경우인데 이는 internal forwarding이 구현되어 있다는 가정 아래에서 forwarding할 이유가 없다. 따라서 0의 출력을 주어 forwarding이 될 필요가 없음을 알린다. internal forwading은 동일한 사이클에 register file에 write한 값을 read할 수 있도록 구현한 것을 의미한다.

rs1에 적용한 논리를 rs2에도 적용해주고 alu의 2번째 입력 앞 mux의 select와 연결한다. 위의 두 경우 모두 EX단계에서 사용하는 레지스터가 0번째라면 forwarding할 필요가 없다. 그 이유는 0번째 레지스터는 항상 0이기 때문이다.

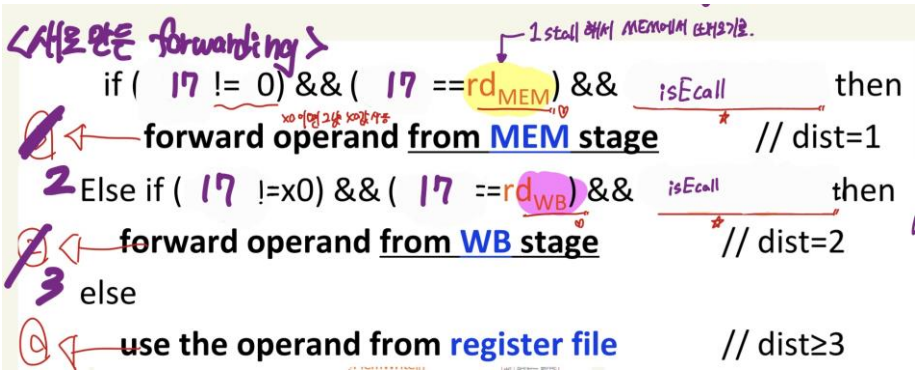
여기서 한 가지 중요한 점은 forwarding을 할 때 가장 최근의 연산으로부터 생긴 값을 사용해야 한다는 것이다. 따라서 if문의 순서를 위 그림처럼 해야 한다. 그렇지 않으면 올바르지 않은 값

이 forwarding되는 꼴이 된다.



이렇게 forwarding을 구현할 때, ALU의 2번째 input은 매우 섬세하게 다루어져야 하는데, 그 이유는 alu의 2번째 입력으로 rs2값을 사용하지 않고 immediate 값을 사용하는 경우 때문이다. immediate값은 forwarding되는 값이 아니라는 점을 고려하였을 때 왼쪽 그림의 파란선으로 구현한 논리가 없다면 mem_write 신호가 1일 때 즉, SW와 같은 연산을 할 때 저장을 할 주소 계산을 immediate값이 아닌 다른 값으로 잘못할 수 있게 된다. 따라서 주소 계산을 할 때는 확정적으로 immediate 값을 사용할 수 있도록 해준다. 또한 초록색 선으로 구현한 논리는 SW와 같은 연산을 할 때 저장할 값을 올바르게 해주는 역할을 한다. 즉, Store의 경우 rs2값이 ALU에서 사용되지 않지만 그 값이 MEM stage에서 write data로써 쓰여야 하는데 이를 고려해주기 위함이다. 이때 write 되는 data는 forwarding이 될 수 있음을 고려하여 기존의 forwarding 결과를 이용한다.

요약하면, 파란색으로 구현한 논리는 store을 진행할 때 write data에서 forwarding이 일어난다면 immediate를 사용하지 못하는 것을 막기 위해 존해가고, 초록색으로 구현한 논리는 store을 진행할 때 write data에서 forwarding을 고려한 올바른 값 저장을 하기 위해서 존재한다. 즉, Store의 경우를 다루기 위한 구조이다.



한편, isEcall로도 forwarding이 필요하다. 이는 rs1과 rs2에서 forwarding하는 원리와 동일하고 한 가지 다른 점은 isEcall 신호가 인가되었을 때만 forwarding을 한다는 것이다. 이 신호는 별다른 register를 거치지 않고 곧바로 forwarding unit으로 전달된다는 점 또한 design의 그림을 보면 알 수 있다.

MEM stage와 WB stage에서 일어나는 일은 기존 싱글 사이클 CPU와 대동소이 하며 구현 또한 기존의 것을 그대로 따랐다. 한 가지 다른 점이 있다면 앞서 언급한 is_halted 신호가 wb단계에서 1일 때 비로소 cpu가 종료된다는 점이다.

4. discussion

Store는 rs2가 ALU 연산에서는 쓰이지 않으나 그 값이 memory의 write data에 쓰인다. I type, R type과 다른 양상 때문에 forwarding을 처리하기 위한 또 다른 구조를 설계해야만 했다. 그렇지 않으면 forwarding이 일어날 때, 주소 연산은 물론 저장되는 값 또한 잘못된 값이 되어버린다. 설계 단계에서 이를 고려하였다면 더욱 좋았겠지만 설계의 오류를 탐지할 수 있는 실력을 기를 수 있는 좋은 기회였던 것 같다.

pipelined CPU는 앞서 구현했던 싱글 사이클 CPU와 멀티 사이클 CPU보다 훨씬 우월했는데, 그 이유는 바로 쉬고 있는 부분이 거의 존재하지 않으며 한 번에 5개의 인스트럭션들을 수행하기 때문이다.

5. conclusion

"non_control_flow.txt"파일에 대해 비교를 진행했다. 싱글 사이클 CPU의 경우 총 51사이클이 걸리고 Lab4의 pipelined CPU의 경우에는 59사이클이 걸렸다. 싱글 사이클 CPU와의 비교를 통해 stall은 총 59-51-4=4번 일어났다는 것을 알 수 있다. 하나의 사이클에 하나의 단계를 수행하는 pipelined CPU의 클락 주기는 하나의 사이클에 모든 단계를 수행해야 하는 싱글 사이클 CPU의 클락 주기보다 대폭 줄어들었을 것이므로 사이클 수는 조금 늘었지만 전체적인 성능을 보았을 때 이는 엄청난 개선이라고 말할 수 있다.

우리 팀은 pipelined CPU를 구현하면서 data hazard 문제를 해결하기 위해서는 어떻게 stall을 해야 하고, 효율을 높이기 위해서 어떻게 data forwarding을 진행해야 할 지를 알 수 있었다.