Computer organization LAB\_02

2017-15872 전기정보공학부 서정환

우선 forwarding의 경우 ex\_rs1, ex\_rs2, mem\_rd, wb\_rd, mem\_opcode, ex\_opcode, mem\_rs2를 input으로 받고, forwardA, forwardB, forwardS, forwardmem을 output으로 하였다. EX의 rs1이 mem의 rd와 같고, mem이 write하는 명령어이고 Load가 아니면 forwardA=01로 하여 ALU의 input을 MEM의 alu\_result로 가게 하였다. Load인 경우는 hazard 모듈에서 커버하여 여기서는 제외하였다. 이 경우가 아닌 경우에 비로소 WB에서 같은 방법으로 조사하였고, 이 때 WB는 Load라는 조건을 빼주고 검사하였다. 만약 해당하는 케이스이면 forwardA=10으로 주어 ALU\_in을 WB의 write\_data로 주었다. EX의 rs2의 경우도 같은 방법으로 조사하였고 rs2의 경우에는 rs1에 더해 추가되는 조건이 있었다. 우선 mem과 비교하는 경우 ex가 rs2를 ALU\_in으로 넣지 않고 sextimm을 넣는 경우일 수 있으므로, EX가 alu\_in으로 imm이 들어가는 경우를 제외해주는 조건을 추가하였다. 이 때 EX가 Store인 경우는 따로 생각해주었다. Store의 경우 GPR[rs2]가 alu\_in으로 사용되지 않고 Write\_data(Memory)로 사용되기 때문이다. 따라서 Ex\_write\_data(추후 MEM에서 메모리에 쓸 데이터가 됨)을 선택하는 3X1 mux를 두고 Store이면 GPR[rs2]를 MEM에서 가져오고, 아니면 그냥 EX의 rs2를 사용하였다. 반대로 Store이면 alu\_in으로는 EX의 sextimm을 사용하였고, 아니면 MEM에서 데이터를 가져왔다. MEM에서 조사했을 시 조건에 만족하지 않을 때 비로소 같은 방법으로 rs2에 대해 WB와 비교하였다. (순서대로 비교하여야 함.) 또한 추가적으로 mem의 rs\_2와 WB의 rd가 같고 MEM이 Store, WB가 Load인 경우 Load의 Writedata(WB)를 store의 MEM에서 메모리에 쓰는 데이터로 forward해주었다. 이는 cpu에서 mux로 구현하였고, 이런 경우는 hazard에서 굳이 stall하지 않도록 hazard에서 컨트롤해주었다. (Store에서 rs2를 쓰는 타이밍과 LW에서 rd가 나오는 타이밍(stage)가 같기 때문에 이 경우는 굳이 stall 필요가 없음.)

Hazard 모듈의 경우 크게 stall과 flush하는 경우를 다루었다. stall하는 경우는 EX의 rd와 ID의 rs가 같고, rs가 x0가 아니며, EX가 Load인 경우, 그리고 ID가 rs를 사용하는 명령어인 경우 stall이 이루어졌다. 이 부분 또한 rs1과 rs2의 구현이 약간의 차이가 있다. Rs1의 경우 앞서 설명한 조건이면 충분하다. Rs2의 경우 일단 rs1보다 사용하지 않는 명령어의 수가 더 많고 ID가 store인 경우에는 stall하지 않아도 된다. 이는 forwarding에서 나중에 처리해주기 때문이다.

Flush의 경우는 EX에서 branch가 taken되거나 jump인 경우 flush신호를 1로 만들었다. 다만 hazard 모듈에서 if\_pc+4를 받고 NEXT\_PC를 결정하는 logic이 있는데 branch가 taken이거나 JALR인 경우에는 target이 EX의 alu\_result이므로 NEXT\_PC를 alu\_result로 주었다. (input으로 EX\_alu\_result도 받음) JAL인 경우에는 EX에서 계산된 EX\_PC + EX\_sextimm값을 NEXT\_PC를 주었다. 해당 값도 hazard에서 input으로 받는다. 그 외의 경우에는 if\_pc\_plus+4를 NEXT\_PC로 주었다.

각각 forwarding과 hazard에서 나온 신호가 cpu에서 어떻게 적용되는지 살펴보자. Stall의 경우는 PC와 if/ID register를 한 cycle latching해주었고 ID/EX register는 flush해주었다. 즉 ID/EX register에서 나와 EX에서 처리될 명령이 사실상 아무런 영향도 끼치지 못하도록 모든 control signal을 0으로, forwarding에 영향을 미치지 않도록 rs1, rs2, rd를 모두 0으로 만들어주었다. LW는 정상적으로 EX/MEM register를 통해 이동한다.

Flush의 경우에는 IF/ID와 ID/EX register를 flush해주었고, IF에 있는 명령어는 해당 cycle에 바로 flush가 힘드므로, flush한다는 신호를 IF/ID register로 이동시켜 해당 cycle에 있던 IF 명령어가 ID에 가서 사실상 flush될 수 있도록 구현하였다. 정확히는 해당 명령어가 ID로 가서 ID/EX register를 통과할 때, ID/EX register에서 나오는 control signal이 0이 되도록, rs,rd도 0이 되도록 flush해주었다.

즉 stall 신호는 IF/ID, ID/EX register, PC에 들어가고, flush신호는 IF/ID, ID/EX register에 들어간다. 이에 따라 IF/ID ID/EX register와 PC를 위의 설명대로 동작하도록 구현해주었다. ID/EX의 경우 둘 중 어느 신호가 오더라도 flush하도록 구현하였다. EX에 BXX나 Jump, ID에 Load가 오는 경우 해당 cycle에서 flush신호가 발생하고, 다음 cycle에서 EX가 LW로 오면서 그 cycle에 있는 명령어와 상호작용하여 stall 명령이 발생할 수 있으나, 이미 flush명령으로 LW의 control signal과 rs, rd를 모두 0이 되도록 구현하여 해당 상황을 방지하였다. 혹시라도 발생하더라도 stall과 flush가 동시에 IF/ID에 들어올 시 flush명령어를 우선하도록 logic을 작성하였다.

Forwarding과 Hazard는 다음과 같이 작성하였고, CPU의 경우 IF/ID, ID/EX, EX/MEM, MEM/WB의 register를 두어 각 cycle마다 <=(nonblocking)으로 register의 값을 이동시켜 차례대로 각 명령어를 stage에서 처리하도록 구현하였다.