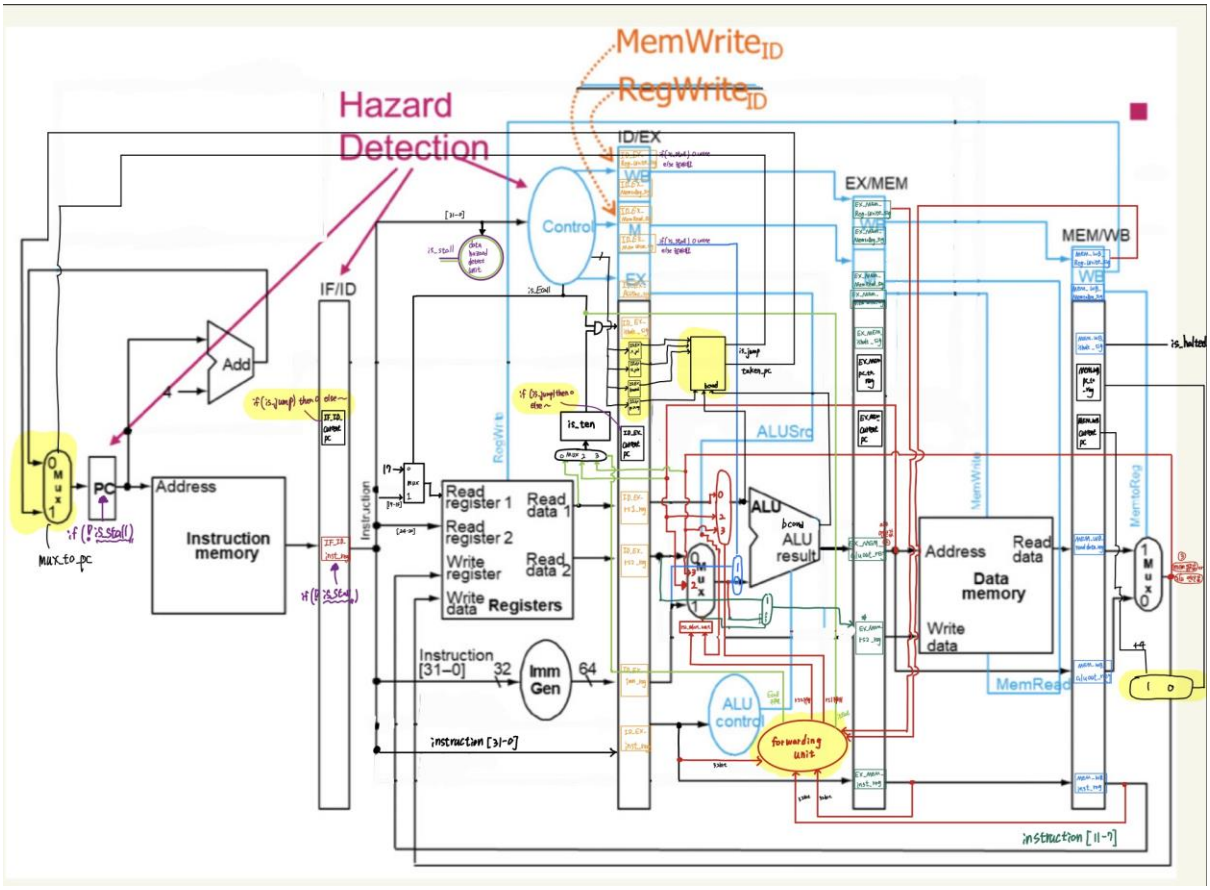
Pipelined CPU with control flow instructions

20210115 이세규 20210706 이지원

1. introduction

이번 과제의 목표는 이전에 구현하였던 control flow 연산을 할 수 없었던 pipelined cpu에 control flow 연산을 지원하는 기능을 추가하는 것이다. 이를 위해 필수적으로 구현해야 하는 사항에는 next pc를 잘 예측하였는지를 판단하는 기능, next pc를 옳지 않게 예측한 경우 flush를 해주는 기능, pc값을 write back할 수 있게 하는 기능이 있다. 우리 팀은 always not taken으로 branch를 predict하기 때문에 BTB가 필요 없다.

2. design



lab 4-1과 비교하였을 때 달라진 부분을 노란색으로 표시하였다.

이전 과제였던 multi-cycle-cpu와 single-cycle-cpu를 구현할 때 모두 짚고 넘어갔던 점들이지만 pipelining을 한다는 점 때문에 추가적으로 고려해야 할 것들이 있었다. 가장 크게 달라진 점은 fetch해온 연산이 틀릴 수 있다는 것이다. 이전의 구현에서는 fetch된 연산이 틀릴 일은 없었다.

하지만 pipelining의 특성상 stall이 일어나는 경우가 아니라면 한 사이클마다 연산이 fetch된다. n번째 연산에서 n+1번째 연산이 일어나는 pc값을 fetch단계에서 알 수는 없기에 next pc값을 예측해야한다. 이는 말 그대로 예측이기 때문에 틀릴 수 있고, 틀린 예측을 교정하기 위해서 필요한 것이 **flush** 기능이다. flush는 무시하고 싶은 연산의 단계에 있는 pvs를 변경시킬 수 있는 signal을 0으로 만들어주는 방식으로 구현할 수 있다. ex단계에서 next pc값의 예측이 맞았는지 판단하기 때문에 예측이 틀렸을 경우에는 판단한 시점에서 id단계와 if단계에 있는 연산을 무시해야한다. 따라서 if단계에 있는 연산을 무시하기 위해 연산비트를 저장하는 if_id 단계의 파이프라인 레지스터에 0이 입력되게 하여 없는 연산 취급을 하고, id단계에 있는 연산이 ex단계에 올 때 id_ex 단계의 모든 파이프라인 레지스터 값을 0으로 하여 없는 연산 취급을 하는 식으로 flush를 구현한다. 구체적인 예시를 통해 더 자세히 알아보자.

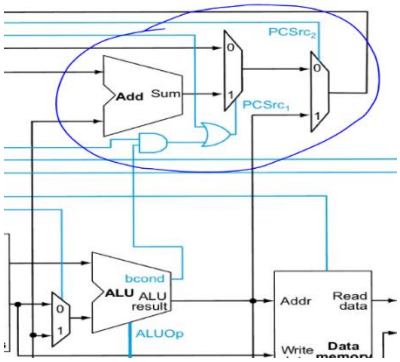
a, b, c, … ,d 연산이 있다고 하자. a가 jal연산이라고 하면, a는 pc값을 기준으로 특정 범위만큼 점프하여 a 다음에는 d연산을 해야 한다. 하지만 always not taken으로 구현하였기 때문에 b와 c 연산이 도합 두 사이클 동안 fetch 될 것이다. c가 fetch 되는 순간, a연산은 ex단계에 들어서고 이때 next pc는 a 연산이 있는 주소에서 +4를 한 곳이 아니라 d라는 것을 알게 된다. 즉, b와 c 연산을 무시해야 한다는 판단을 하는 것이다. 이제 위에서 설명한 대로 pipeline 레지스터 값을 변경하여 b, c 연산을 flush할 것이다.

그렇다면 **예측한 pc가 올바른 예측이었는지 틀린 예측이었는지는 어떤 식으로 판단**할까? id 단계에서 생기는 control signal의 is_jal, is_jalr, branch 신호와 cpu연산에서부터 나오는 bcond 신호를 통해 알 수 있다. 우선 is_jal과 is_jalr이 1인 경우, pc+4를 target으로 jump하지는 않으니 예측은 틀렸을 것이다. 따라서 is_jump라는 신호를 1로 하여 이전에 다루었던 flush를 진행한다. 또한 branch 신호가 1 일 경우 분기 조건 검사의 결과인 bcond에 따라 이 값이 1 인경우 is_jump에 신호를 인가하여 flush를 해야함을 알린다. 이와 동시에 올바른 next_pc값을 계산하여 다음 사이클부터 해당 값으로 instruction fetch를 진행한다.

추가적으로 lab 4-1에서 더해주어야 하는 부분은 register로 write back되는 값이 pc값일 수도 있기 때문에 pc+4값을 write back하는지를 결정하는 pc_to_reg 신호와 해당 연산의 pc값을 저장하는 current_pc를 추가한다. 이를 위해 wb 단계에 기존의 wb 값과 pc+4값을 선택하게 하는 mux가 하나 필요하며, instruction memory 왼편에 pc+4와 target 중 하나의 값으로 next_pc값을 결정하기 위한 mux도 추가해야한다.

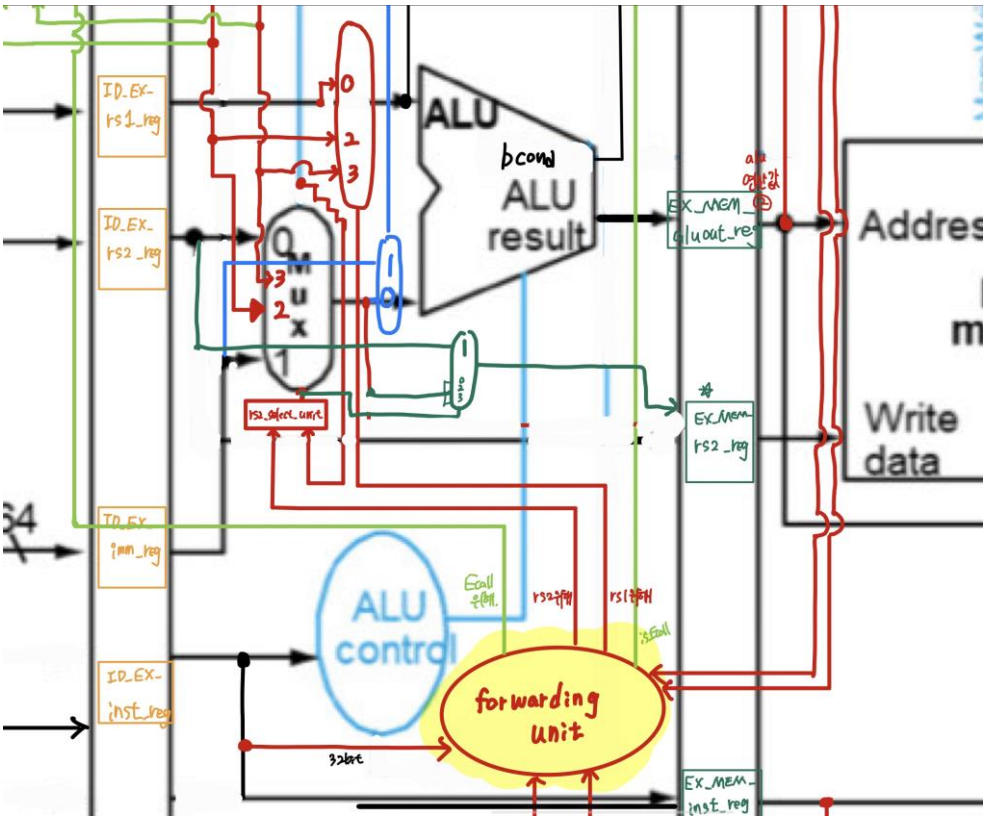
위 내용을 요약하자면, **항상 pc+4값을 next_pc로 predict 하여 always not taken으로 branch를 predict하며, 이 predict의 옳고 그름은 ex단계에서 판단하고 predict가 틀렸을 경우 flush하는 식으로 control flow를 다룬다.**

3. implementation



우리 팀이 구현한 방식에서는 jalr의 경우 target을 alu에서 계산하지 않는다. 대신 예측의 옳고 그름을 판단함과 동시에 target을 계산하는 모듈을 따로 만들었다. 이로써 single cycle cpu에는 있었던 jal의 target과 jalr의 target을 결정하는 mux를 cpu.v에서 없앨 수 있었다. 즉, 왼쪽 그림에서 파란색 동그라미 친 부분을 하나의 모듈로 구현한 것이다. 아래의 코드를 보면 해당 모듈에서 is_jump 신호와 taken_pc를 모두 구성함을 알 수 있다.

```
always @(*) begin
  if(ID_EX_is_jal||ID_EX_is_jalr||(ID_EX_branch&&bcond)) begin //점프 뛰는 경우 -> JAL, JALR, BXX의 taken
    is_jump <= 1;
    if(ID_EX_is_jalr) begin
      taken_pc <= (ALU1+ID_EX_imm)&32'hFFFFFFFE; //FORWARDING된거로.  ID_EX_imm
    end
    else begin
      taken_pc <= ID_EX_current_pc+ID_EX_imm;
    end
  end
  else begin //점프 안뛰는 경우 -> 나머지
    is_jump <= 0;
    taken_pc <= 0;
  end
end
```



또한 lab 4-1구현과 비교하였을 때 한 가지 수정한 점에는 우리 팀이 구현한 forwarding logic에 맞게 forwarding unit을 조금 더 안정화한 것이 있다. 이를 설명하기 위해서는 1. forwarding은 alu의 입력 중 aluop를 제외한 두 입력에 대해 필요한데, 두 입력 중 기존에 rs1이 들어오던 입력 부분, rs2와 imm이 들어오던 입력 부분의 forwarding 양상이 다르다는 점을 인지하는 것과 2.우리 팀의 forwarding logic 구현 방식에 대한 이해가 필요하다. mux 아래에 있는 rs2_select_unit의 경우 forwarding unit의 신호가 0일때만 기존의 신호인 ALUSrc를 따르도록 구현되어 있다. 따라서 rs1의 forwarding logic과 rs2의 forwarding logic을 동일하게 할 경우 alu에서 imm값을 써야 할 때 imm값이 아닌 다른 값을 쓰게 될 수 있는 것이다. lab 4-1에서는 store 연산에서 이러한 경우가 발생하여 그림의 파란색, 초록색 mux를 추가하여 문제를 해결하였다. lab 4-2에서도 유사한 문제가 발생하였는데 이번에는 forwarding unit의 rs2관련 출력을 보다 엄격하게 통제하여 문제를 해결하였다.

```
begin
  if(id_ex_inst[6:0]==`ARITHMETIC_IMM) //arithmetic_imm 일때는 rs2에 forwarding이 필요없음
  begin
    forwarding_rs2 <=2'd0;
  end
else
  begin
    if((id_ex_inst[24:20]!=0)&&(id_ex_inst[24:20]==ex_mem_inst[11:7])&&ex_mem_reg_write) f
    else if((id_ex_inst[24:20]!=0)&&(id_ex_inst[24:20]==mem_wb_inst[11:7])&&mem_wb_reg_wri
    else forwarding_rs2<=2'd0;
  end
end
```

I type 연산의 경우 alu의 rs2, imm 입력 부분에 imm이 들어가므로 forwarding이 전혀 필요없음을 확인하여 위와 같은 구현을 추가하였다. 강의 시간에 배운 use_rs1이라는 보조 함수를 쓰는 것과 동일한 배경인데, '우연히' 비트가 일치하여 예외적인 상황을 만드는 것을 방지한 구현이라고 해석할 수 있다. (위 추가 구현이 없었더라면 I type 연산을 하는데 우연의 일치로 rs2에 forwarding이 일어나는 bit와 동일한 연산이 입력될 경우, imm이 아닌 다른 값이 alu 연산에 활용되는 오류가 생길 수 있다.)

4. discussion

시스템을 디자인하는 것에서 처음부터 완벽하게 디자인하는 것이 매우 어렵다는 깨달음을 얻었다. 수업시간에 배운 내용을 기반으로 디자인을 하였고 그에 따라 구현을 하였지만 실제 동작에서는 고려해야 하는 사항이 더 있었다. 이러한 점들은 매우 미묘하여 설계의 오류를 찾아내는 과정 또한 쉽지 않았다. 물론 처음 디자인 단계에서 모든 경우를 다 고려하고 완벽하게 설계를 하는 것이 가장 좋겠으나 현실적으로 불가능한 부분이 있기 때문에 익숙하지 않은 시스템을 만들땐 큰 흐름을 우선적으로 설계하고 예외적인 케이스를 발견하여 기존의 구현을 더 보완하는 식의

접근이 바람직하다는 생각을 하였다. 단, 특정한 예외만을 처리하는 임기응변식의 구현을 하는 것이 아니라 해당 예외를 발생시킨 상위의 오류 개념을 찾아내어 (lab 4-2에서 forwarding logic의 취약점을 파악한 것처럼) 안정적인 시스템을 만드는 식의 노력을 하는 것이 의미있다는 것 또한 알게 되었다.

5. conclusion

recursive_mem -> 1187사이클

non-controlflow_mem -> 46사이클

basic_mem -> 35 사이클

loop_mem -> 322 사이클

non_control_flow_mem -> 59 사이클

ifelse_mem -> 43사이클

위 결과는 우리 팀이 구현한 pipelined cpu with control flow의 각 테스트 케이스별 소요 사이클 수이다. 모든 branch 연산에 대해 not taken으로 예측하였기 때문에 다른 예측기를 사용하였을 경우에 기대할 수 있는 성능향상은 없다. jal, jalr, BXX 타입의 연산이 올 경우 항상 2 사이클의 flush가 일어나기 때문이다. 이 과제를 진행하며 pipelined cpu에 대한 더 깊은 이해와 시스템 디자인에 대한 이해를 높일 수 있었다.