4190.308**Computer Architecture**Spring 2020

Pipeline Lab Report

05/25 ~ 06/08

Name: Kim Gideok

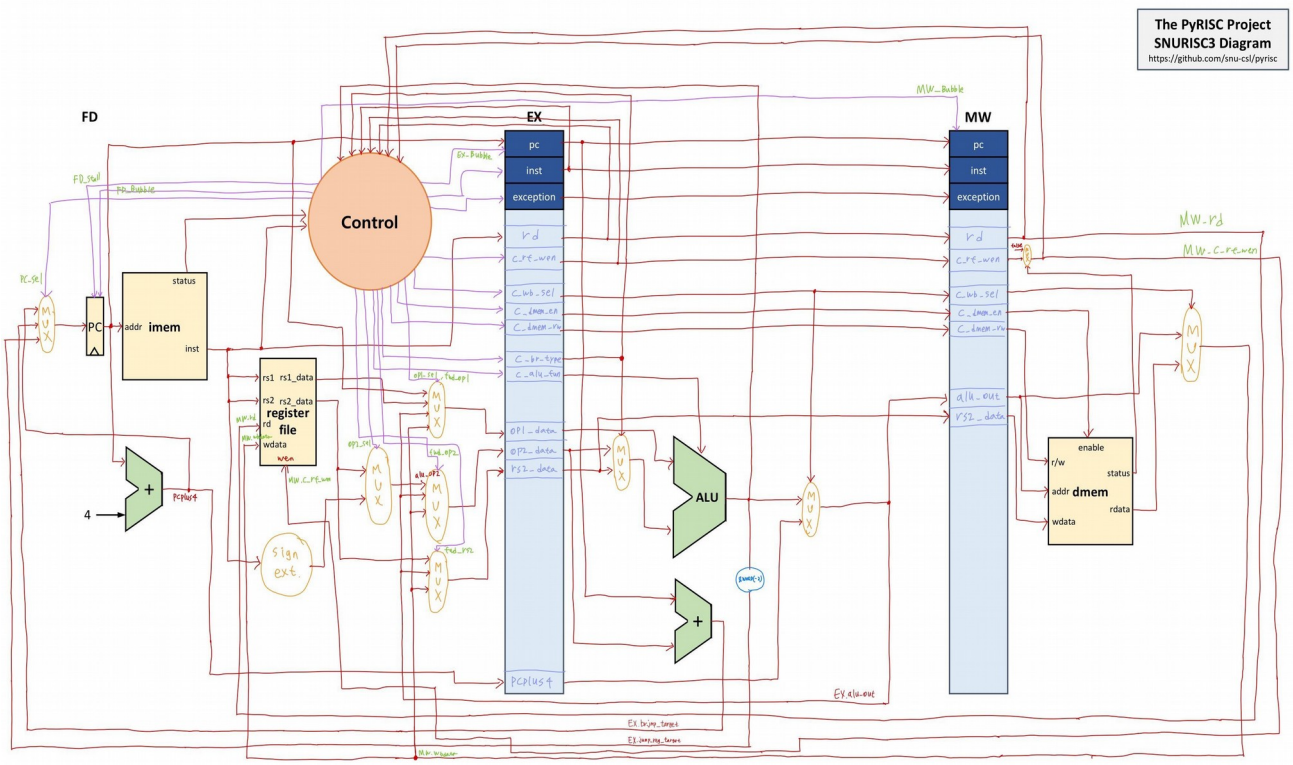
Student ID: 2018-13627

Introduction

Pipeline lab 은 instruction 이 속도 향상을 위해 pipeline 의 기법으로 진행이 되는 것을 직접 코딩해보는 lab 이다. 단순하게 instruction 을 진행시키는 Pipeline 기법이 아무런 문제 없이 잘 돌아가면 좋겠지만, 아무런 처리 없이 그냥 pipelining 을 하면, 여러가지 hazard 가 발생한다. Hazard 에는 여러 종류가 있지만 모든 hazard 의 근본적인 발생 원인은 이전 것이 완전히 처리되기 전에 현재의 것을 진행시키기 때문이다. 이러한 hazard 에는 크게 두 가지 종류가 있는데, 하나는 data 를 읽고 쓰는데 시간적인 오류로 인해 hazard 가 발생하는 경우, branch instruction 에서 다음 instruction 예측에 실패해서 잘못된 instruction 을 실행한 경우가 있다. 전자를 data hazard, 후자를 control hazard 라고 한다. 이번 lab 에서는 이 두가지 hazard 에 대한 처리가 아직 안 되어있는 snurisc3 를 완성시키는 것이다.

Design

Pipeline Architecture



1. FD stage

Control 에서 받아오는 pc_sel signal 에 따라 다음 pc 를 선택하고 그에 따른 instruction 을 받아온다. Instruction 에 따른 operand 를 읽어오고, Control 에서 보내는 hazard 관련 signal 들에 따라 hazard 가 발생할 경우 그에 따른 적절한 forwarding 값을 operand 로 설정한다. 또한 다음 PC 값을 현재 PC+4 의 값으로 기본적으로 해 둔다. 그리고 현재 instruction 에 대한 정보들과 결과값들(operand 등)을 EX.reg 에 보내준다.

2. EX stage

EX.reg 를 통해 FD stage 에서 받아온 operand 들과 instruction 정보를 이용해 결과값을 계산한다. Branch instruction 의 경우에는 여기서 next pc 값이 계산된다. 현재 instruction 의 branch type 정보, memory load 를 하는지에 대한 정보 등을 Control 로 전달한다. 또한, 현재 instruction 에 대한 정보들과 결과값들(alu_out)을 MW.reg 에 보내준다.

3. MW stage

MW.reg 를 통해 EX stage 에서 받아온 결과값들과 instruction 을 토대로 메모리에 접근해서 원하는 값을 저장하거나 destination register 에 불러온다. 또한 여기서 발생할 수 있는 MEM hazard, Load-Use hazard 에 대한 처리를 위한 rd, c_rf_wen 값을 Control 로 보내준다.

4. Control
각 stage 에서 받아온 변수들을 이용해 hazard 를 판단해서 control signal 들을 설정해준다.

Data Hazard

1. EX hazard
EX hazard 는 현재 instruction 의 FD phase 에서 사용하려는 register 가 이전 instruction 에서 write back 되는 경우에 발생할 수 있다. 이 hazard 가 발생하는 이유는 현재 instruction 의 FD phase 가 이전 instruction 의 MW phase 보다 먼저 일어나기 때문이다. EX hazard 의 경우는 실제로 write back 할 값은 EX phase 에서 이미 계산이 완료된다. 따라서 이 값을 바로 현재 instruction 의 FD phase 에서 가져오면 이 hazard 는 해결될 수 있다. Snurisc3 에서는 Control 에서 FD 가 계산에 사용할 register 와 EX 의 rd register 를 비교한 값이 유효하게 같으면 FD 에서 op data 를 선택하는 MUX 에 signal 을 주어 EX 에서 계산된 data 를 op data 로 사용할 수 있게 하였다.

2. MEM hazard
MEM hazard 는 현재 instruction 의 FD phase 에서 사용하려는 register 가 전전 instruction 에서 write back 되는 경우에 발생할 수 있다. 이 hazard 가 발생하는 이유는 현재 instruction 의 FD phase 가 이전 instruction 의 MW phase 보다 먼저 일어나기 때문이다. MEM hazard 의 경우, 실제로 write back 할 값은 MW phase 에서 load 가 완료된다. 따라서 이 값을 바로 현재 instruction 의 FD phase 에서 가져오면 이 hazard 는 해결될 수 있다. Snurisc3 에서는 Control 에서 FD 가 계산에 사용할 register 와 MW 의 rd register 를 비교한 값이 같고 해당 register 를 현재 instruction 에서 op1 으로 사용할 경우, FD 에서 op data 를 선택하는 MUX 에 signal 을 주어 MW 에서 load 된 data 를 op data 로 사용할 수 있게 하였다.(단, EX hazard 와 겹치면 EX hazard 를 우선으로 하도록 하였다.)

```
self.fwd_op1    =   FWD_EX if(Pipe.EX.reg_rd == Pipe.FD.rs1) and rs1_oen and (Pipe.EX.reg_rd != 0) and
Pipe.EX.reg_c_rf_wen else \
            FWD_MW if(Pipe.MW.reg_rd == Pipe.FD.rs1) and rs1_oen and (Pipe.MW.reg_rd!=0) and
Pipe.MW.c_rf_wen else \
            FWD_NONE
```

3. Load-Use hazard
Load-Use hazard 는 현재 instruction 의 FD phase 에서 사용하려는 register 가 이전 instruction 에서 write back 되는 경우에 발생할 수 있다. 이 hazard 가 발생하는 이유는 현재 instruction 의 FD phase 가 이전 instruction 의 MW phase 보다 먼저 일어나기 때문이다. Load-Use hazard 의 경우, 실제로 write back 할 값은 MW phase 에서 load 가 완료된다. 그런데 이전 instruction 의 MW phase 마저도 현재 instruction 의 FD phase 보다 나중에 나온다. 따라서 이때는 instruction 의 진행을 한 번 멈춰주는 stall 과 bubble 이 필요하다. Snurisc3 에서는 Control 에서 FD 가 계산에 사용할 register 와 EX 의 rd register 를 비교한 값이 유효하게 같고, EX instruction 의 MEM_EN, MEM_FCN 값을 통해 이 instruction 이 메모리 값을 읽어서 쓸 것인지를 판단하고 만약 그렇다면 stall 과 bubble 신호를 보낸다.

```
Pipe.EX_load_inst = Pipe.EX_cs[CS_MEM_EN] and Pipe.EX_cs[CS_MEM_FCN] == M_XRD
load_use_hazard=(Pipe.EX_load_inst and Pipe.EX.reg_rd != 0) and ((Pipe.EX.reg_rd == Pipe.FD.rs1 and
rs1_oen) or (Pipe.EX.reg_rd == Pipe.FD.rs2 and rs2_oen))
```

Control Hazard

Snurisc3 는 Branch instruction(Jump 포함)이 발생하면 우선 branch 조건이 만족하지 않는다고 가정하고 평소에 하던 것처럼 pc_next 를 pc+4 로 설정하고 진행을 한다. 그러다가 EX phase 에서 branch 검사 결과를 Control 에 넘기고 Control 에서는 이 결과를 바탕으로 branch 를 택할 것인지 그대로 진행할 것인지를 판단한다. 그리고 그 결과에 따라 만약 branch 를 수용하기로 판단이 되면 pc_next 값을 JALR instruction 의 경우는 EX.jump_reg_target 으로, 그 외의 branch operation 의 경우는 EX.brjmp_target 으로 설정하였다. 물론, branch 를 선택하지 않는다고 판단이 되면 그대로 pc+4 를 택하였다. branch 를 판단하는 방법은 EX.reg_c_br_type 에 따라 EX.alu_out 이 올바른지를 확인하는 것이다.

```
 if((Pipe.EX.reg_c_br_type == BR_NE and (not Pipe.EX.alu_out)) or (Pipe.EX.reg_c_br_type == BR_EQ and
Pipe.EX.alu_out) or (Pipe.EX.reg_c_br_type == BR_LT and Pipe.EX.alu_out) or (Pipe.EX.reg_c_br_type ==
```

```
BR_GE and (not Pipe.EX.alu_out)) or (Pipe.EX.reg_c_br_type == BR_LTU and Pipe.EX.alu_out) or
(Pipe.EX.reg_c_br_type == BR_GEU and (not Pipe.EX.alu_out)) or (Pipe.EX.reg_c_br_type == BR_J)):
            self.pc_sel=PC_BRJMP
        elif (Pipe.EX.reg_c_br_type == BR_JR):
            self.pc_sel=PC_JALR
        else:
            self.pc_sel=PC_4
self.pc_next    = self.pcplus4  if Pipe.CTL.pc_sel == PC_4       else \
                  Pipe.EX.brjmp_target if Pipe.CTL.pc_sel == PC_BRJMP else \
                  Pipe.EX.jump_reg_target if Pipe.CTL.pc_sel == PC_JALR else \
                  WORD(0)
```

Conclusion

이번 lab 에서는 pipeline 이 적용된 cpu 에서 data hazard 와 control hazard 를 해결하는 방법을 배웠다. 솔직히 말해서 처음에는 엄청나게 막막했다. 처음에 진행이 힘들었던 가장 큰 이유는 hazard 의 해결을 위해서 새로운 변수를 선언해야 하기 때문이었다. 구체적으로 변수를 들어서 pipeline 의 hazard 를 접근하여 본 적이 없기 때문에 난관이었다. 하지만 snurisc3, 5 의 skeleton 및 architecture 를 참고하여 천천히 접근하다보니 어떤 변수들이 필요한지 알게 되었고 마침내 snurisc3 의 완전한 구현에 성공할 수 있었다. Pipeline 의 핵심은 pipeline register 라는 것을 알게 되었고 이것들을 어떻게 이동시키는지 알게 되었다. 그리고 Control 에서 어떻게 현재 어떤 hazard 가 발생했고 어떻게 상황을 처리하는지 궁금했는데, 각 상태의 register 및 결과값들의 조건문들을 통해 control signal 을 내려주고 각 stage 에서 이런 signal 에 맞게 상황에 맞게 선택한다는 것을 알게 되었다. 전체적으로 지금까지 가장 어려웠던 lab 이었다는 생각이 들고, 그래도 하다보니 점차적으로 이해가 잘 되는 것 같아서 점차 나아가는 맛이 있었다.