4190.308

Computer Architecture

Spring 2020

Pipeline Lab Report

05/25 ~ 06/08

Name: Yang Kichang

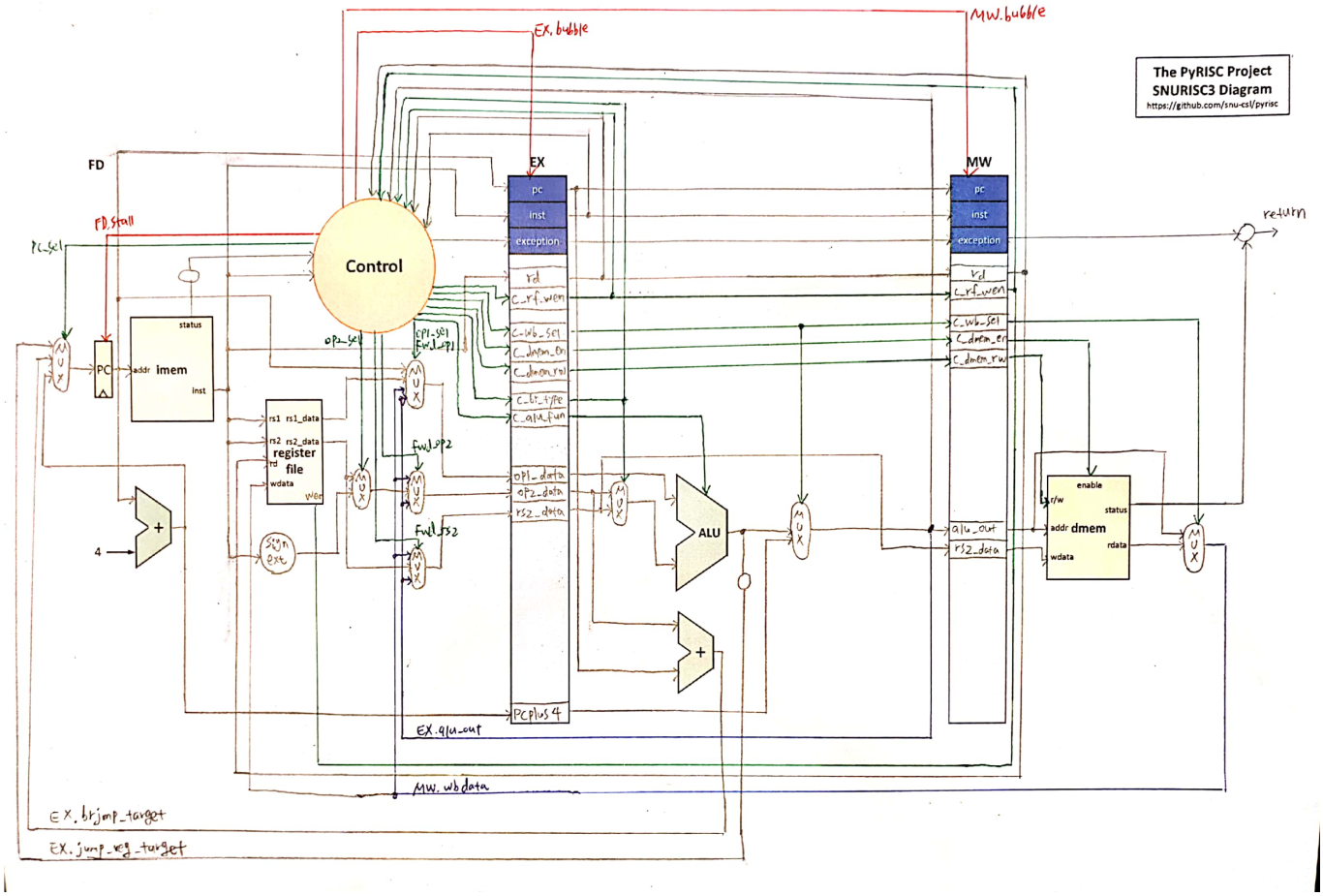
Student ID: 2015-13078

Introduction

본 프로젝트의 목적은 3 stage pipelined RISC-V processor 를 구현하는 것이었다. 3 개의 stage 는 각각 기존 5 stage 에서의 IF/ID stage 가 합쳐진 FD stage, EX stage, 기존 5 stage 의 MEM/WB stage 가 합쳐진 MW stage 이다. Processor 가 RV32I 명령어들을 올바르게 처리하기 위해 필요한 datapath 및 control signal 들을 만들고, 발생 가능한 data hazard, control hazard 를 처리하기 위한 forwarding / stall logic 을 구현하였다.

Design

Pipeline Architecture



FD stage: 주어진 PC 로부터 instruction 을 가져온 후 이를 decode 한다. Instruction 을 바탕으로 register file 에서 필요한 register 값을 읽어들이고 immediate 값을 생성하며, Control logic 은 각종 control signal 들을 생성한다. Forwarding/Stall logic 역시 FD stage 에 위치해있어, 현재 FD stage 에 위치한 instruction 과 앞선 instruction 들의 정보를 조합하여 hazard 를 탐지하고 각종 forwarding / stall signal 을 생성한다.

EX stage: ALU 를 이용한 실제 연산이 이루어진다. Branch instruction 의 경우 ALU 를 통해 branch 여부가 계산되며 별도의 adder 를 통해 target address 가 계산된다. Jal instruction 도 별도의 adder 를 통해, jalr instruction 은 ALU 를 통해 target address 가 계산된다.

MW stage: 메모리에 접근하고, register file 에 적절한 값을 저장한다.

Data Hazard

1. Register write 을 하는 instruction 후 해당 register 값을 이용하는 instruction 이 오는 경우
아직 register file 에 결과값이 저장되지 않았는데 다음/다다음 instruction 에서 해당 값을 이용해야 되는 경우, forwarding 을 통해 처리할 수 있다. 앞서 언급했듯이 forwarding 은 FD stage 에서 처리되며, 특정 조건이 만족될 때 EX / MW stage 에서의 값이 FD stage 로 forwarding 되는 방식이다.

1.1. 두 instruction 이 연속적인 경우
EX stage 의 ALU 연산 결과가 FD stage 에서 필요하다. 아래와 같은 조건이 만족되면 forwarding 을 진행하도록 한다.

- EX stage 의 instruction 이 register write 을 진행하는 instruction 이며, destination register 가 x0 이 아니다.
- FD stage 의 instruction 에서 register1/register2 값을 필요로 하며, 해당 register 번호가 EX stage 의 instruction 의 destination register 번호와 일치한다.

조건이 만족될 경우, forwarding signal 을 생성하여, pipeline register 에 기존 register 값이 아닌, forwarding 된 값이 저장되도록 한다.

1.2. 두 instruction 사이에 다른 instruction 이 하나 있는 경우
MW stage 의 write back 값이 FD stage 에서 필요하다. Register write 은 cycle 의 가장 마지막에 진행된다고 가정하므로 이와 같은 forwarding 이 필요하다. 아래와 같은 조건이 만족되면 forwarding 을 진행하도록 한다.

- MW stage 의 instruction 이 register write 을 진행하는 instruction 이며, destination register 가 x0 이 아니다.
- FD stage 의 instruction 에서 register1/register2 값을 필요로 하며, 해당 register 번호가 MW stage 의 instruction 의 destination register 번호와 일치한다.

조건이 만족될 경우, forwarding signal 을 생성하여, pipeline register 에 기존 register 값이 아닌, forwarding 된 값이 저장되도록 한다.

* 1.1 과 1.2 가 동시에 발생할 경우, 최근 결과값이 우선이 되어야 하므로 1.1 이 적용된다.
* 코드와 위의 diagram 을 보면 Forwarding 대상으로 op1, op2, rs2 이 있다. 여기서 op1 은 register1 값, rs2 는 register2 값, op2 는 immediate 값 이용시 immediate 값 / 이용하지 않을 시 register2 값이다. rs2 와 op2 가 모두 필요한 이유는 branch / store instruction 이 register2 값과 immediate 값을 모두 사용하기 때문이다. 따라서 이들은 op2 에 immediate 값이, op1/rs2 에 register 1/2 값이 담기며, op1/rs2 가 forwarding 대상이 된다.

2. Load instruction 을 통한 register write 후, 해당 register 값을 이용하는 instruction 이 오는 경우
Load instruction 은 register write 을 진행하는 다른 instruction 들과 다르게, 메모리에 접근하여 값을 얻어내야 한다. 이에 따라 MW stage 의 후반부에 값이 준비되고, forwarding 만으로는 이 값에 대한 dependency 를 해결할 수 없다. 따라서 1 cycle stall 이 불가피하다.
Load instruction 이 EX stage 에 도달하고, load 이후의 instruction 이 FD stage 에 있는 시점에서, 이러한 dependency 가 존재하는지 확인한 후 필요하다면 stall 을 진행한다. Dependency 존재는 다음과 같은 조건을 이용하여 확인한다.

- EX stage 의 instruction 이 Load instruction 이며, destination register 가 x0 이 아니다.
- FD stage 의 instruction 에서 register1/register2 값을 필요로 하며, 해당 register 번호가 EX stage 의 instruction 의 destination register 번호와 일치한다.

이러한 Load-use hazard 가 탐지되면, PC 를 stall 시키고, 현재 FD stage 의 instruction 을 bubble 로 만들어서 1 cycle 을 지연시킨다. 그러면 load instruction 이 MW stage 에 도달하여 메모리에서 원하는 값을 얻게 되고, 이를 기존의 forwarding logic 에 따라 FD stage 로 forwarding 하게 된다.

* Load instruction 직후 등장하는 store instruction 의 rs2 가 load 값에 dependent 한 경우, stall 을 피할 수 있다. Load instruction 이 MW stage 에서 wbdata 를 얻었을 때, 이를 EX stage 로 forwarding 하여 MW pipeline register 의 rs2_data 에 forwarding 된 값이 저장되도록 하면 된다.
이것은 구현 후 주석으로 처리해놓았다. 구현한 부분은 다음과 같다.

- Pipeline register 에 rs2 추가
- Store instruction 의 rs2 가 직전의 load instruction 에 dependent 한 경우, stall 하지 않도록 stall logic 추가
- EX stage 에 forwarding logic 추가

Load 직후 store 가 등장하는 코드를 이용하여 실험한 결과 CPI 가 낮아지는 것을 확인할 수 있었다. 예를 들어 loadstore.s 를 실행하면, 기존에는 14cycle 이 소요되지만 해당 기능을 추가하면 13cycle 안에 완료된다.

Control Hazard

1. Branch instruction
본 프로젝트의 구현에서는 branch instruction 에 대해 “always not taken”으로 예측한다. 따라서 branch 가 수행되어야 하는 상황이 생기면 문제가 발생한다. Branch 여부 및 branch address 는 EX stage 에서 계산된다. 따라서 만약 branch 를 수행해야 한다면 FD stage 에 있는 instruction 을 bubble 로 만들고, control signal 을 통해 PC 에 기존 PC+4 값이 아니라 계산된 branch address 값이 저장되도록 한다.
Branch 여부를 판단하기 위해서는 EX stage 의 c_br_type 과 alu_out 값을 이용한다. 예를 들어, beq instruction 은 ALU operation 으로 “set if equal”을 사용하므로, alu_out 값이 1 이면 branch 한다. Branch 를 해야 하는 경우 1 cycle penalty 가 발생하며, branch 하지 않는 경우 penalty 는 없다.

	beq	bne	bge	bgeu	blt	bltu
ALU operation	seq	seq	slt	sltu	slt	sltu
Branch 하기 위한 ALU 결과값	1	0	0	0	1	1

* seq: set if equal slt: set if less than sltu: set if less than unsigned

2. Jal / Jalr instruction
Jal 과 Jalr 도 비슷하게 EX stage 에서 target address 가 계산되므로 FD stage 에 있는 instruction 을 bubble 로 만들고, control signal 을 통해 PC 에 기존 PC+4 값이 아니라 새로운 target address 값이 저장되도록 한다.
다만, branch 와 다르게 항상 jump 해야 하므로, 항상 1 cycle penalty 가 발생하게 된다.

Conclusion

단순히 눈으로 datapath 그림들을 보던 것에 비해, 이번 프로젝트를 통해 이를 직접 구현해보면서 더 구체적이고 명확한 이해를 할 수 있었다. 특히 COD 책을 읽을 때는 pipeline, data hazard, control hazard 등의 내용들이 순차적으로 나오기 때문에 지식이 파편화되어 머리 속에 저장되는 느낌이 들었는데, 이번에 그 모든 것들을 한 번에 구현하며 더욱 완성도 있는 지식이 된 느낌이다.
또한, 책에는 나오지 않았던 구체적인 구현 방법들을 배울 수 있었다. 예를 들어 branch / store instruction 을 위해 op2_data 와 rs2_data 를 따로 두는 것, jal / jalr 을 처리하는 방법 등이 있다.
어떤 내용을 깊이 있게 습득하기 위해서는 역시 직접 구현해보는 것이 가장 좋다는 것을 느꼈다.

가장 힘들었던 것은 처음 뼈대 코드를 보고 이를 이해하는 일이었다. 코드가 상당히 복잡하여 각 부분이 무엇을 담당하는 코드인지 알기 어려웠고, 각종 상수들이 많아서 이들의 의미를 파악하고 기억하는 것이 힘들었다. 하지만 코드를 반복해서 읽으면서 점차 익숙해졌다. 특히, 5 stage pipeline 을 구현해놓은 코드를 보는 것이 많은 도움이 되었다.

뼈대 코드를 제대로 이해한 후, 나머지 부분을 구현하는 과정은 크게 어렵지 않았다. 로직이 크게 복잡하지 않았고, 3 stage 는 5 stage 보다 약간 더 단순하기 때문이다.