Lab 5: Pipeline

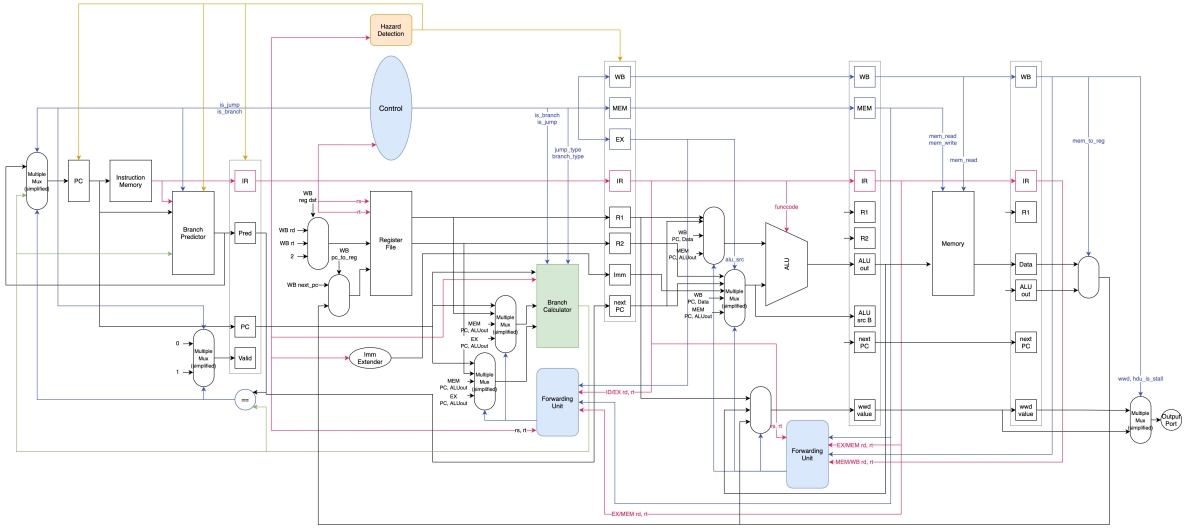
20190084 권민재, 20190335 양승원 CSED311

Introduction

이번 랩 과제에서는 Pipeline CPU를 구현하는 것이 목표이다. 앞선 랩 과제에서는 싱글 사이클 CPU와 멀티 사이클 CPU를 구현해보았다. 그러나 이들은 모두 한 번에 하나의 명령어만 수행할 수 있다. 명령어 실행 과정을 다섯 단계로 쪼개어 각각 다른 명령어를 하나씩 수행하는 것이 파이프라인 CPU의 아이디어이다.

이를 위해 IF/ID/EX/MEM/WB 사이에 진행상황을 저장하는 파이프라인 레지스터가 필요하다. data dependency를 해결하기 위한 data forwarding이나, branch 직후 stall 없이 다음 명령어를 수행하기 위한 branch predict, branch 예측 실패나 jump 수행을 했을 때 한 명령어를 쉬어가기 위한 flush나 stall 등이 추가적으로 필요하다.

Design



기존 single cycle CPU에 비해 Pipleline CPU에서 달라진 점을 중점적으로 기술하였다.

Pipeline Registers

파이프라인 레지스터들은 각 스테이지에서의 결과를 Latch 하기 위해 필요한 레지스터들이다. 각 스테이지마다 서로 다른 명령어들이 실행될 것이기 때문에, 각 스테이지는 각각의 상태를 저장해 둘 레지스터가 필요하기 때문에 파이프라인 레지스터가 필요하다. 기본적으로는 각 스테이지에서 생성된 결과를 다음 파이프라인 레지스터에 저장하거나 이전 파이프라인 레지스터의 값을 다음 파이프라인 레지스터로 전달하도록 디자인하였다. 다만, stall이나 flush 등의 이유로 latch 시키지 않아야할 때에도 있기 때문에 이를 고려하여 구현해야할 것이다.

Hazard Detection Unit

Hazard가 발생했을 경우에는 stall을 진행해야하기 때문에 이를 검출하는 유닛이 필요하다. 이 유닛은 ID에 위치하여, ID 스테이지의 source 레지스터가 이전 스테이지의 destination 레지스터와 같고, 이전 스테이지가 Load Instruction이라면 hazard가 발생했음을 알릴 수 있도록 설계했다.

Branch Predictor

Always-not-taken 전략을 취하는 분기 예측기로, IF 스테이지에 위치하도록 설계하였다. 이 유닛은 현재 PC를 입력받아 언제나 다음 PC (`PC + 1`)을 반환할 것이다.

Branch Calculator

Branch Calculator는 분기 채택 여부와 그 주소를 결정하는 계산기로, ID에 자리하고 있다. 이는 이 CPU 구현에서 별도의 분기 예측을 하지 않는 대신 클럭을 최대한 아끼기 위함이다. Branch calculator는 실제로 분기하거나 jump 해야할 주소를 계산하며, 이가 Branch Predictor가 예측한 주소와 다를 경우에는 CPU에서 IF 스테이지 flush를 진행하도록 설계하였다. 분기 예측 및 연산 과정에서 jump 또한 자연스럽게 처리할 수 있도록 각 유닛에서 jump도 처리하도록 디자인하였다.

Forwarding Unit

Forwarding Unit은 forwarding이 필요한 유닛에 대해서 어떤 데이터를 쓰게 할 것인지 결정하는 유닛이다. 이 CPU 구현에는 ID 스테이지, EX 스테이지에 Forwarding Unit이 한 개씩 존재하도록 디자인하였다. ***ID 스테이지의 Forwarding Unit***은 Branch Calculator가 사용할 데이터를 결정한다. Branch Calculator는 기본적으로 레지스터에서 읽은 값이나 현재 스테이지의 PC를 이용하나, data hazard에 따라서 EX나 MEM 스테이지의 PC나 ALU 결과 값의 이용할 수 있도록 설계하였다. ***EX 스테이지의 Forwarding Unit***은 ALU가 사용할 데이터를 결정한다. ALU는 기본적으로 이전 스테이지에서 레지스터로부터 읽은 값을 이용하지만, data hazard에 따라서 MEM이나 WB 스테이지의 PC나 데이터를 이용할 수 있도록 설계하였다.

Implementation

num_inst

각 파이프라인 레지스터는 자신이 valid한 instruction을 담고 있는지 여부를 valid reg에 저장한다. 이 valid reg는 처음에는 1로 설정되었다가, 해당 instruction이 jump의 다음 instruction이어서 무시되어야하거나, flush되는 경우 0으로 설정된다.

valid instruction이 MEM 단계에 도달하면 num_inst가 1 증가한다.

Pipeline Registers

cpu.v 에 reg 로 선언한다.

한 클럭 사이클이 지날 때마다, posedge clk 시퀀셜 로직에서 파이프라인 레지스터의 값들을 업데이트 해준다. 앞의 파이프라인 레지스터에 있던 값인 경우 그대로 받아오고, IF/ID/EX/MEM/WB 각 단계에서 나오는 wire 들은 직후의 파이프라인 레지스터에 새로 저장한다.

stall되는 경우 IF_ID 파이프라인 레지스터가 업데이트하지 않으면 ID단계의 instruction은 그대로 보존된다.

Hazard Detection Unit

ID 단계에서 hazard를 분석하는데, ID/EX 파이프라인 레지스터에 load instruction이 있는지 id-ex의 is_store를 이용해 알아낸다. 그리고 id-ex의 rd가 현재 if-id 파이프라인 레지스터에 있는 rs나 rt와 같은 경우 hazard가 발생했음을 알 수 있다.

Branch Predict

Always Not Taken으로 구현하였다. Branch Predictor는 IF단계에서 항상 pc+1(`w__pred_pc` -> `r__if_id__pred_pc`)을 계산하고, Branch Calculator는 ID단계에서 실제 PC(`w__branch_address`)를 계산한다.

When it Fails

과거 예측했던 값과 현재 연산한 값이 다른 경우(`r__if_id__pred_pc != w__branch_address`), 즉 Branch Predict가 실패하는 경우, 현재 instruction fetch를 통해 IF_ID 파이프라인 레지스터에 있는 값은 잘못된 instruction의 값이다. 따라서 이것을 NOP로 비워준다. NOP에 관한 자세한 설명은 아래 Discussion에 존재한다.

When it's Successful

Branch Predict가 성공하는 경우, fetch되어 IF_ID 파이프라인 레지스터에 있는 instruction은 실제로 수행해야하는 instruction이 맞다. 따라서 별도의 처리가 필요하지 않다.

Jump

jump의 목적지가 되는 주소는 JMP, JAL instruction의 경우 immediate 값을 사용하고, JPR, JRL instruction의 경우 레지스터 값을 사용한다. 따라서 IF단계에서 다음 instruction의 주소를 특정할 수 없는 Control Hazard가 발생한다.

때문에 branch와 마찬가지로 IF_ID 파이프라인 레지스터에 있는 instruction을 NOP으로 flush하고, 드디어 계산된 점프 주소를 사용해 다음 instruction을 수행한다. 점프 주소는 branch와 같이 관리하기 위해 branch_calculator에서 나오는 `w__branch_address`를 사용한다.

Forwarding Unit

포워딩의 대상이 되는 instruction은 ID_EX 파이프라인 레지스터에 있고, 하나 전의 instruction은 EX_MEM 파이프라인 레지스터에, 둘 전의 instruction은 MEM_WB 파이프라인 레지스터에 존재한다. EX_MEM이나 MEM_WB 파이프라인 레지스터에 있는 RegWriteDest와 RegWrite를 통해 어떤 레지스터에 write가 진행되었는지 파악하고, 이것이 ID_EX 파이프라인 레지스터의 RS나 RT와 같다면 해당 레지스터에 포워딩을 해주게 된다.

둘 전의 instruction보다 하나 전의 instruction을 더 우선하여 포워딩해야하는데, 더 최근의 것이기 때문이다.

Discussion

- cpu.v의 posedge clk 시퀀셜 로직에서, 파이프라인 레지스터의 값을 뒤쪽부터 업데이트함으로써 오류를 막았다.
- instruction read나 data read가 memory의 posedge에서 발생하여, cpu의 posedge에서 이들 값을 받아올 수 없었다. 따라서 그 둘은 따로 마련된 combinational logic에서 대입해주었다.
- 비어있어 아무 동작도 하지 않는 instruction(`NOP`)을 표현할 방법을 고민하다가, `WWD $0` 을 사용하기로 하였다. 그 이유는 Branch Predict에 BTB를 사용하더라도 이에 영향을 주지 않을 뿐더러, 항상 jump나 branch 뒤에만 삽입되기 때문에 data dependency로 인해 stall될 걱정도 없기 때문이다. 기존의 정상적인 `WWD $0`와 구별을 위해, `0xf01c` 대신 `0xf11c`를 사용하였다.
- pc가 레지스터에 적히는 경우에는 pc를 포워딩할 수 있도록 파이프라인 레지스터와 mux를 구성했다.

- alu control unit이 다시 사라졌는데, branch predict를 위해 bcond 연산이 ID 단계로 이동함에 따라 존재 의의가 다시 사라졌기 때문이다.
- 레지스터 유닛을 위한 forwarding signal을 생성하는 forwarding unit은 EX단계에 있는데, 마찬가지로 data forwarding을 요구하는 branch calculator는 ID단계에 있기 때문에 BC를 위한 forwarding unit을 추가로 만들었다.
- jump는 immediate 값을 요구하는 JMP,JAL instruction과 레지스터 값을 요구하는 JPR,JRL instruction으로 나뉜다. 두 경우 모두 IF단계에서는 점프 도착 주소를 확정할 수 없다. 따라서 ID단계에서 주소를 연산한 후 IF단계에 존재하는 instruction을 flush한다.(NOP으로 설정한다.) immediate 값은 같은 ID 단계에 존재하는 imm_ext unit으로부터 가져오고, 레지스터 값은 포워딩이 필요한 경우 `Forwarding_BC_Unit` 을 거쳐 가져온다.
- branch의 경우에도 비슷한데, 다른 점은 branch_predictor가 정해준 다음 pc(`pc+1`)가 정말로 다음 pc였던 경우, 즉 branch not taken인 경우에는 flush없이 진행할 수 있다는 점이다. branch taken이었던 경우에는 IF 단계의 instruction을 NOP으로 만들고, 실제 branch taken 주소를 다시 이용하여 다음 instruction을 fetch하게 된다.
- 다른 instruction은 stall 없이 data forwarding만으로 바로바로 처리할 수 있으나, load instruction이 예외가 된다. load는 MEM단계에서야 레지스터에 어떤 값이 써질지 알 수 있다. 따라서 load 바로 다음 instruction이 load하는 바로 그 레지스터를 참조하는 경우, 한 instruction stall이 필요하다. 이것은 ID_EX 파이프라인 레지스터에 존재하는 instruction을 무력화해야하는데, reg_write와 mem_write를 끔으로써 수행한다. 그리고나서 PC와 IF_ID 파이프라인 레지스터를 업데이트하지 않으면, stall이 수행된 것이다.

Conclusion

- Multi-Cycle CPU - 5031 Clock Cycles

```
VSIM 3> run -all
# Clock # 5031
# The testbench is finished. Summarizing...
# All Pass!
# ** Note: $finish    : D:/Downloads/CSED311/lab/git/CSED311/Lab/Lab04/core/cpu_TB.v(151)
#    Time: 503250 ns  Iteration: 2  Instance: /cpu_TB
# 1
# Break in Module cpu_TB at D:/Downloads/CSED311/lab/git/CSED311/Lab/Lab04/core/cpu_TB.v line 151
```

- Pipelined CPU - 1293 Clock Cycles

```
VSIM 63> run -all
# GetModuleFileName: ÁöÁ¤µÈ ¸ðµâÀ» Ã£À» ¼ö ¾ø½À´Ï´Ù.
#
#
# Clock # 1293
# The testbench is finished. Summarizing...
# All Pass!
# ** Note: $finish    : D:/Downloads/CSED311/lab/git/CSED311/Lab/Lab05/core/cpu_TB.v(153)
#    Time: 129450 ns  Iteration: 2  Instance: /cpu_TB
# 1
# Break in Module cpu_TB at D:/Downloads/CSED311/lab/git/CSED311/Lab/Lab05/core/cpu_TB.v line 153
```

Pipelined CPU가 Multi-Cycle CPU보다 약 $\frac{5031-1293}{1293} \times 100 \simeq 289.0(\%)$ 향상되었다. 파이프라인을 이용해 여러 insturction을 동시에 실행하는 것이 대역폭을 향상시켜서 CPU의 성능 향상에 도움을 주는 것을 알 수 있다.