EE312 <Lab 5: Pipeline CPU Lab>

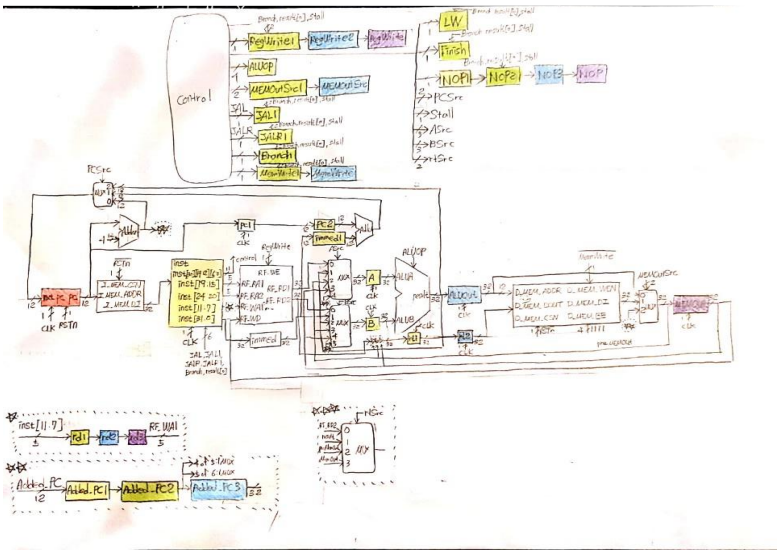
20170616 정희진

1. Instruction

Lab5는 pipelined CPU를 직접 코드를 작성해보는 랩이다. pipeline이 적용된 CPU에서는 data hazard와 control hazard가 일어날 수 있기 때문에 이를 잘 알고 다뤄야 한다. 이번 랩에서는 두 가지 hazard를 모두 고려해 코드를 작성하였다. data hazard의 경우 forwarding 방법을 이용하였다(단 Load instruction과 hazard가 함께 일어날 수 있는 instruction의 distance가 1일때는 stall의 방법을 사용하였다). 그리고 control hazard의 경우 'always-not-taken' branch predictor을 이용하였다.

2. Design

pipelined CPU design은 아래의 그림과 같다.



그림에서 색칠된 변수들은 모두 pipeline register이며, clock이 rising할때마다 update가 된다.

빨간색	WB단계와 IF단계 사이의 register
노란색	IF단계와 ID단계 사이의 register
연두색	ID단계와 EX단계 사이의 register
파란색	EX단계와 MEM단계 사이의 register
보라색	MEM단계와 WB단계 사이의 register

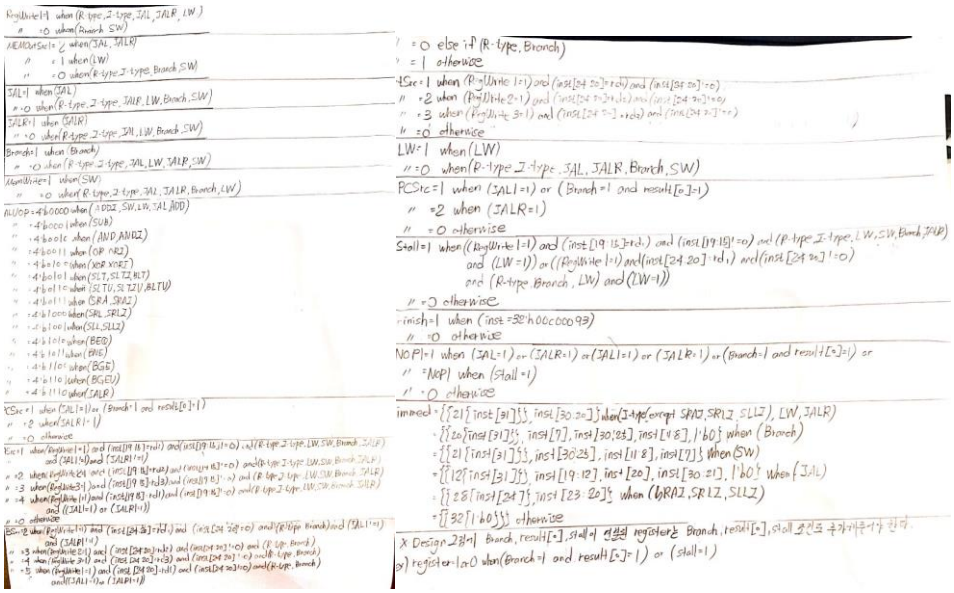
그림에서 나타난 변수들 중 이해하기 어려운 몇가지 것들의 의미는 다음과 같다.

Added_pc	JAL, JALR instruction의 경우 WB stage에서 pc+4를 저장해야한다. 이를 위한 pipeline regster이다.

Finish	프로그램이 종료되기 위해서는 0x00c00093(addi x1, x0, 0xc)와 0x00008067(jalr x0, x1, 0)가 연달아 fetch되어야 한다. 이때 0x00c00093을 나타내기 위한 register이다.
NOP	1 clock이 지날 때마다 1 instruction이 종료되면 좋지만 그렇지 못할 때가 있다. 1) JAL, JALR instruction은 다음 pc가 EX stage에서 결정되므로 2 clock이 버려진다. 2) Branch instruction은 taken되면 다음 pc가 EX stage에서 결정되므로 2 clock이 버려진다. 3) LW instruction은 register에 쓰여질 값이 MEM stage에서 결정되기 때문에 data hazard가 일어날 때 두 instrunction 간격이 1이면 forwarding 방법으로 ALU operands로 이동할 수 없게 된다. 이 때는 특별히 stall 방법을 이용하였고 이 경우 1 clock이 버려진다. NOP register은 위의 3가지 경우를 다룬 register이다. 이 register을 이용하여 NUM_INST가 알맞게 증가하도록 코드를 작성하였다.
Stall	위에서 말한 3가지 경우 중 3번째 경우를 다룬 register이다.

3. Implementation

위 디자인의 각각의 control들은 다음 그림과 같이 설정해주었다.



pipelined CPU 디자인의 알고리즘은 랩 3, 4와 많이 비슷하므로 data와 pc의 흐름이 어떻게 되는지에 대한 설명은 생략하는 것이 좋을 것 같다. 랩 3, 4와 비교했을 때 가장 큰 차이점은 CPU가 5가지 단계로 구분이 되고, 특별한 경우가 아닌 이상 1 clock(한 stage)마다 1 instruction을 다룰 수 있도록 해준다(특별한 경우는 'Design-NOP register 설명'에 나타나 있다). 이 랩에서는 5가지 단계를 다음과 같이 나누었다.

pc	IF	inst	ID	A, B, rt (ALU전 register)	EX	ALUOut	MEM	MEMOut	WB	pc

또한 pipelined CPU의 경우 data hazard와 control hazard를 미리 알고 다루어야 한다. 이 랩에서는 다음과 같이 hazard를 다루었다.

1) data hazard using forwarding

Design 그림에서 IF stage부분을 보면 A, B, rt register 전에 MUX가 존재한다. MUX와 연결된 data는 General register에서 fetch된 것도 있지만, result(EX), preMEMOut(MEM), MEMOut(WB)도 있다. 이는 ASrc, BSrc, rtSrc를 통해 두 instruction의 간격에 따라 다르게 control하였다.

2) data hazard using stall (special case)

Load instruction이 나타난 후, 연이어 Load instruction에서 쓰여질 register을 사용하는 instruction이 나타날 때는 stall 방법도 함께 이용하였다. Design 그림을 참고하면서 보면 Load instruction은 MEM stage에서 register에 들어갈 값을 받을 수 있다. 하지만 그 값을 사용할 instruction의 경우 ALU전에 값을 받아야하는데, 두 instruction의 거리가 1인 경우 forwarding을 할 수 없으므로 1 clock stall을 해주었다. 거리가 2가 된 두 instruction 사이의 단계는 NOP instruction처럼 신호가 나오도록 control하였다. 거리가 2가 된 후에는 1)과 같이 forwarding을 이용해 해결하였다.

3) control hazard using 'always-not-taken' branch predictor

pc register은 특별한 경우를 제외하고 pc+4가 되도록 코드를 작성하였다. 특별한 경우란, JAL, JALR이 나타날 때, 또는 Branch instruction이 나타나고 taken될 때이다. 이 경우 EX stage에서 다음 pc를 결정하기 때문에 두 단계가 어쩔 수 없이 버려지게 된다. 이 instruction과 다음 pc에서 나타난 instruction 사이에서는 NOP instruction처럼 신호가 나오도록 control하였다.

4. Evaluation

모든 testcase를 통과하였으며 속도도 빨랐다.

1) TB_RISCV_inst.v

```
# Test #    1 has been passed
# Test #    2 has been passed
# Test #    3 has been passed
# Test #    4 has been passed
# Test #    5 has been passed
# Test #    6 has been passed
# Test #    7 has been passed
# Test #    8 has been passed
# Test #    9 has been passed
# Test #   10 has been passed
# Test #   11 has been passed
# Test #   12 has been passed
# Test #   13 has been passed
# Test #   14 has been passed
# Test #   15 has been passed
# Test #   16 has been passed
# Test #   17 has been passed
# Test #   18 has been passed
# Test #   19 has been passed
# Test #   20 has been passed
# Finish:         23 cycle
# Success.
# ** Note: $finish    : C:/Users/jhj/Desktop/code/release/testbench/TB_RISCV_inst.v(175)
#    Time: 345 ns  Iteration: 1  Instance: /TB_RISCV
```

2) TB_RISCV_forloop.v,

```
# Finish:        101 cycle
# Success.
# ** Note: $finish    : C:/Users/jhj/Desktop/code/release/testbench/TB_RISCV_forloop.v(166)
#    Time: 1125 ns  Iteration: 1  Instance: /TB_RISCV
```

3) TB_RISCV_sort.v

```
# Finish:      14728 cycle
# Success.
# ** Note: $finish    : C:/Users/jhj/Desktop/code/release/testbench/TB_RISCV_sort.v(192)
#    Time: 147395 ns  Iteration: 1  Instance: /TB_RISCV
```

5. Discussion

각 stage에서 변수들이 어떤 값을 가질지 고려하며 설계하는데 많은 시행착오가 있었다.

1) pipeline register의 경우 clock이 rising할 때마다 그전 register값으로 update가 되어야하는데 처음에는 동시에 update되는 등 신호가 이상하게 나왔다. 이는 '<='와 '=' 를 제대로 구분하지 않고 섞어서 썼기 때문이었다. clock이 rising할 때 프로그램은 '='가 있는 코드를 먼저 실행하고 그 다음으로 '<='가 있는 코드를 실행하였다. register가 제대로 update되기 위해서는 clock이 rising할 때마다 바뀌는 register들을 모두 '<='로 사용하여야 되었다.

2) HALT와 NUM_INST을 다루는 것이 어려웠다. 이를 다루기 위해서는 추가적인 register가 필요했고, 이 랩에서는 NOP register와 Finish register을 이용하였다.

3) instruction을 flush하는 것이 어려웠다. 간단히 inst register에 NOP instruction을 넣으면 되는 것도 있었지만, 여러가지 control pipeline register를 조절하면서 flush해야하는 경우도 있었다.

4) forwarding을 하기 위한 조건과 stall을 하기 위한 조건을 맞추기가 어려웠다. 신호를 보고 하나씩 대입하면서 빠진 조건을 채워나갔다.

5) TB에 나타난 instruction이 asm파일에서 어떤 instruction에 해당하는지 주소까지 주석을 달아주면 디버깅을 하기 편할 것 같다(sorting 파일은 instruction과 loop가 많이 있어 test된 instruction이 어떤 instruction인지 잘 알기 어려웠다).

6. Conclusion

모든 testcase를 잘 통과한 것으로 보아 의도한 대로 잘 설계된 것으로 보인다. 랩 코드를 작성하니 수업시간 때 배웠던 내용들을 좀더 잘 이해할 수 있게 되었다. 이번 랩에서 "always-taken" branch prediction with the BTB 와 "2-bit saturation counter" with the BTB를 함께 작성하지 않아서 아쉽지만 시간이 충분할 때 두 branch prediction을 이용해 작성을 해봐야겠다는 생각을 하였다.