Computer Architecture (001)
Programming Assignment 4
Design Document

자유전공학부
2021-15738 양석훈

December 2022

1 Part 1: When do the new data hazards occur due to the push and pop instructions and how do you deal with them?

먼저, PUSH, POP instruction의 cssignal을 아래와 같이 정하였다. 기존 lw, sw 그리고 sp 계산을 위한 add instruction 들과 비슷하게 정해서 기존에 구성된 logic을 최대한 활용하고자 하였다.

```
PUSH   : [ Y, BR_N  , OP1_RS1, OP2_IMI, OEN_1, OEN_1, ALU_SUB  , WB_MEM, REN_1, MEN_1, M_XWR, MT_W, ],
POP    : [ Y, BR_N  , OP1_RS1, OP2_IMI, OEN_1, OEN_0, ALU_ADD  , WB_MEM, REN_1, MEN_1, M_XRD, MT_W, ],
```

Figure 1: PUSH, POP cssignal

1.1 Show all the possible cases when data hazards can occur and your solutions to them

가능한 data hazards로는 load-use hazard가 존재한다. 특히, PUSH, POP instruction의 경우 sp register의 값을 사용하기에, 앞선 instruction 들에서 sp가 rd에 위치한다면, 이를 포워딩해와서 사용해야 한다. 또, data hazard는 아니지만 PUSH, POP은 모두 ALU를 사용하여 sp +- 4를 계산하기에 이 값 또한 주어져야 한다. 이를 위해 아래와 같이 PUSH, POP의 사용하지 않는 rs1을 sp로 설정하였고, rd 또한 sp로 설정하였다. 이때, POP instruction의 경우, 주어진 rd 값은 MM 단계에서 사용하기에 ID, EX stage에 pipeline register를 만들어, EX stage의 update()에서 원래의 rd로 돌려놓았다. 또한, aluop2 역시도 4로 설정하였다.

```
# CHANGE: for POP, PUSH instruction, rs1 shoud be sp(=2)
if ((self.reg_inst & PUSH_MASK) == PUSH):
    self.rs1 = Pipe.ID.rs1 = 2
    self.rd = Pipe.ID.rd = 2
if ((self.reg_inst & POP_MASK) == POP):
    self.rs1 = Pipe.ID.rs1 = 2
    self.saving_pop_rd = self.rd
    self.rd = 2
```

Figure 2: Change rs1, rd

```
# CHANGE: want to make 4
tmp = alu_op2
alu_op2 =   4       if (self.reg_inst & PUSH_MASK) == PUSH  else \
            4       if (self.reg_inst & POP_MASK) == POP    else \
            tmp
```

Figure 3: Change aluop2

이렇게 설정할 경우, 상당히 구현에 있어 편리함을 느낄 수 있었다. 이후, 발생하는 여러 문제들을 해결하기 위해 코딩하였는데, 아래의 설명들에서 발생한 문제들과 이를 해결한 방법들을 확인할 수

있다.

1.2 Explain the changes in the datapath

먼저, 위에서 말한 새로운 Pipeline register를 만든 변화가 있다. 이외에도 rs1, rs2, aluout 등의 결과를 MM, WM 등의 stage까지 확장하기 위해서 추가적인 pipeline register를 만들고 연결하여 사용하였다. 예시로, POP instruction의 경우 rd, sp의 두 레지스터에 값을 작성하여야 한다. 이를 위해, WB stage에서 reg-wbdata2를 만들어 self.aluout의 값을 넘겨주어서 sp를 WB stage에서 추가적으로 작성할 수 있도록 하였다. 이처럼, 편의를 위해서 pipeline register 들을 추가하였다.
또한, load-use hazard로 인한 forwarding 문제를 해결하기 위해서도 datapath를 일부 변화시켰다. 문제의 발단은 POP instruction은 MM stage에서 부터는 입력인 rd 값을 가지고 있어서 sp를 포워딩해올 방법이 없는 것이었다. 이를 해결하기 위하여 Pipeline register를 이용해 직접 비교하게 되었다. 예시로, 어느 순간에 push t0; add x0, x0, x0; pop a1과 같은 instruction set이 있다고 하자. 이때, 먼저 기존 forwarding signal에 대한 결과를 스켈레톤과 동일하게 계산한다. 이후, 추가적으로 확인하는 것이다. MM stage에 POP instruction이 있고, EX stage에는 bubble, 혹은 rd가 sp가 아닌 instruction이 있다면, ID stage에서는 POP instruction의 alu 결과를 포워딩해올 수 있다. 이를 위해서 아래 코드와 같은 조건들을 추가하여, 만족한다면 추가적으로 포워딩해오게 되었다. 자세한 조건은 코드에서 확인할 수 있을 것이다.

```
# CHANGE: load-use for POP
if ((Pipe.EX.reg_inst == BUBBLE) or (Pipe.EX.reg_rd != 2)) and ((Pipe.MM.reg_inst == BUBBLE) or \
    (Pipe.MM.reg_rd != 2)) and (Pipe.WB.reg_inst & POP_MASK == POP):
    if (self.rs1 == 2) and Pipe.CTL.rs1_oen:
        self.op1_data = Pipe.WB.alu_out
    if (self.rs2 == 2) and Pipe.CTL.rs2_oen:
        self.rs2_data = Pipe.WB.alu_out
if ((Pipe.EX.reg_inst == BUBBLE) or (Pipe.EX.reg_rd != 2)) and (Pipe.MM.reg_inst & POP_MASK == POP):
    if (self.rs1 == 2) and Pipe.CTL.rs1_oen:
        self.op1_data = Pipe.MM.alu_out
    if (self.rs2 == 2) and Pipe.CTL.rs2_oen:
        self.rs2_data = Pipe.MM.alu_out
```

Figure 4: Load-use datapath

1.3 Explain the changes in the control signals

Control sigal은 loaduse hazard를 해결하기 위한 것과 forwarding을 위해서 변경하였다. 먼저, forwarding의 경우, 기존의 forwarding과 더불어, PUSH, POP instruction의 경우는 sp register의 값도 forwarding해야 한다. 이를 위해 위에서 말한 것처럼 datapath에서 PUSH, POP의 rs1, rd value를 sp register의 번호인 2로 설정해놓고 시작하였다. forwarding logic은 아래 사진과 같다. 간단하게 말하면, EX, MM, WB stage에 있는 instruction을 확인한 후, nearist를 forwarding 해오는 것이다. 자세한 조건은 아래의 코드와 같다.

다음으로는, load-use hazard를 해결하기 위한 것이다. POP instruction의 경우, ID stage에서 rd

```
# CHANGE: forwarding logic for PUSH, POP instruction
if opcode in [ PUSH, POP ]:
    self.fwd_op1    =   FWD_EX      if (EX.reg_rd == 2) and EX.reg_c_rf_wen and rs1_oen else       \
                        FWD_MM      if (MM.reg_rd == 2) and Pipe.MM.reg_c_rf_wen and rs1_oen else  \
                        FWD_WB      if (WB.reg_rd == 2) and WB.reg_c_rf_wen and rs1_oen else       \
                        FWD_NONE
```

Figure 5: Forwarding signal

를 2로 변경하였다. 이로 인해, 기존 조건으로는 pop t0; push t0와 같은 load-use hazard를 해결하지 못하기에 추가적인 조건식이 필요하였다. 아래 코드는 이 문제를 해결한 코드이다. 쉽게 말하면, rd와 비교하는 것과 추가적으로 POP instruction의 원래 rd 값을 임시로 저장해놓은 pipeline register를 통해서 load-use hazard를 해결할 수 있었다. 이 역시도, 자세한 조건은 아래의 코드와 같다. 이렇게 구현한 이후, load-use hazard의 해결은 lw에서 해결한 방법이 기존 스켈레톤에 구현되어 있었고, pop의 cssignal을 lw와 유사하게 구현하여 기존에 구현되어 있던 틀에 올라탈 수 있었다.

```
# CHANGE: load-use for POP instruction
if (not load_use_hazard):
    load_use_hazard     = (EX_load_inst and EX.reg_saving_pop_rd != 0) and          \
                          (EX.reg_inst & POP_MASK == POP) and                       \
                          ((EX.reg_saving_pop_rd == Pipe.ID.rs1 and rs1_oen) or     \
                           (EX.reg_saving_pop_rd == Pipe.ID.rs2 and rs2_oen))
```

Figure 6: Load-use hazard signal

2 Part 2: How do you implement the branch prediction using the BTB?

2.1 Explain your implementation of the BTB

기본적으로, BTB는 파이썬의 list를 이용하여 구현하였다. 2차원 리스트로 구현하여 N*3의 크기를 가지도록 하였다. 여기서, N은총 BTB의 크기이고, 3은 valid, tag, target을 저장하기 위한 공간이다. lookup() 메소드는 주어진 pc에 대해 index와 tag를 구하여, 해당하는 공간을 찾아 valid bit이 1인 경우에 해당 btb의 target address를 리턴하고, 아니면 0을 리턴하도록 하였다. remove() 메소드는 간단하게 valid bit을 0으로 만들어 접근하지 못하도록 구현하였고, add() 메소드는 btb에서 index를 찾아 tag, target address를 덮어씌우고, valid bit을 1로 만들어주도록 구현하였다. BTB를 구현한 코드는 아래 사진과 같다. BTB를 이용하여 branch prediction을 하는 과정은 다음과 같다. branch instruction들은 jal, jalr, beq, bne, blt, bge, bltu, bgeu가 있다.
먼저, jalr의 경우 snurisc5의 구현과 동일하게 다음 instruction을 계속 실행하다가 jalr이 EX stage에서 점프할 address가 구해지면 기존 IF, ID에 있던 instruction을 flush하고, target address부터 이어서 계속 실행하도록 구현하였다. 따라서, IF stage에서는 jalr의 경우 무조건 다음 명령어가 fetch되도록 구현하였다.

```python
class BTB(object):

    def __init__(self, k):
        self.k = k
        # initialize your BTB here
        self.N = 1 << self.k
        self.table = [[0 for col in range(3)] for row in range(self.N)]

    # Lookup the entry corresponding to the pc
    # It will return the target address if there is a matching entry
    def lookup(self, pc):
        index = (pc >> 2) % self.N
        tag = pc >> (self.k + 2)
        if (self.table[index][0] == 1) and (self.table[index][1] == tag):
            return self.table[index][2]
        else:
            return 0

    # Add an entry
    def add(self, pc, target):
        index = (pc >> 2) % self.N
        tag = pc >> (self.k + 2)
        self.table[index][0] = 1
        self.table[index][1] = tag
        self.table[index][2] = target
        return

    # Make the corresponding entry invalid
    def remove(self, pc):
        index = (pc >> 2) % self.N
        self.table[index][0] = 0
        return
```

Figure 7: BTB implementation

다른 branch들의 경우, 위에서 구현한 BTB를 사용하였다. 해당 명령어가 처음 실행되는 경우, 즉, lookup() 메소드의 결과가 0인 경우는 다음 instruction (pc + 4) 이 실행되도록 하고, not taken 하였으므로 self.taken을 0으로 표시해주었다. 만약, 이미 lookup() 메소드를 통해 table을 확인한 결과 유효한 target address가 있다면, 그 address로 점프하고, self.taken을 1로 표시해주었다.
여기서, self.taken은 pipeline register를 이용해 IF stage에서 부터, EX stage까지 해당 값을 전달해주도록 하여 후에 misspredict를 확인하는데 사용할 수 있도록 구현하였다. 아래는 해당 부분을 구현한 코드이다.

2.2 Explain how you find out that the current branch is mispredicted and handle the mispredicted branch

BTB를 통해 branch prediction을 하던 도중, misprecict된 branch를 찾는 방법은 아래와 같다. 먼저, branch를 네 종류로 구분하였다. (beq, blt, bltu), (bne, bge, bltu), (jal), (jalr)의 네 종류로 구분해 놓은 것이다. 각 instruction에 대해 EX stage의 update() 메소드 가장 뒤에서 여부를 판정

```
# CHANGE: BTB
self.pc_next = self.pcplus4
opcode = RISCV.opcode(self.inst)
if opcode in [ JAL, BEQ, BNE, BLT, BGE, BLTU, BGEU ]:
    if (Pipe.EX.btb.lookup(self.pc) == 0):
        self.pc_next = self.pcplus4
        self.taken = 0
    else:
        self.pc_next = Pipe.EX.btb.lookup(self.pc)
        self.taken = 1
if opcode in [ JALR ]:
    self.pc_next = self.pcplus4
```

Figure 8: Branch prediction

하였다. 아래는 mispredict를 판정하고 handle하는 코드이다.

첫 번째로, jalr instruction의 경우이다. 스펙에서 처럼 jalr은 항상 not taken하도록 구현되었다

```
# CHANGE: BTB
opcode = RISCV.opcode(self.inst)
flush = False
if opcode in [ BEQ, BLT, BLTU ]:
    if (self.alu_out == 1) and (self.taken == 0):
        flush = True
        IF.reg_pc = self.brjmp_target
        self.btb.add(self.pc, self.brjmp_target)
    elif (self.alu_out == 0) and (self.taken == 1):
        flush = True
        IF.reg_pc = self.pcplus4
        self.btb.add(self.pc, self.pcplus4)
if opcode in [ BNE, BGE, BGEU ]:
    if (self.alu_out == 0) and (self.taken == 0):
        flush = True
        IF.reg_pc = self.brjmp_target
        self.btb.add(self.pc, self.brjmp_target)
    elif (self.alu_out == 1) and (self.taken == 1):
        flush = True
        IF.reg_pc = self.pcplus4
        self.btb.add(self.pc, self.pcplus4)
if opcode in [ JAL ]:
    if (self.taken == 0):
        flush = True
        IF.reg_pc = self.brjmp_target
        self.btb.add(self.pc, self.brjmp_target)
if opcode in [ JALR ]:
    flush = True
    IF.reg_pc = self.jump_reg_target
```

Figure 9: Find mispredict branch

고, 위에서 말하였다. 즉, 항상 mispredict 되는 것이다. 이를 따라서, EX stage에서 점프할 target address가 나오면, 항상 IF, ID stage의 instruction을 flush하고 target address로 점프하도록 하였

다.
두 번째로, jal instruction이다. jal의 경우, taken하였는지의 여부를 확인하면 mispredict인지 알 수 있다. jal은 항상 taken 해야하는 것이므로 self.taken = 0라면 mispredict 되었다는 결론을 내릴 수 있다. 이 경우, add() 메소드를 통해서 btb를 새롭게 갱신하고 앞선 instruction들을 flush하며, target address로 점프하도록 구현하였다.
세 번째로는 (beq, blt, bltu) instruction이다. 이들의 특징으로는, alu의 연산 결과가 1, 즉 두 수를 비교한 결과가 참일때 taken해야 한다는 특성을 가진 것이다. 따라서, self.aluout과 self.taken을 비교하여 두 수가 다르다면 mispredict된 것이다. 이 경우, taken 하였다면 not-taken으로 btb를 업데이트하고, 앞선 instruction을 flush 하며, add() 메소드를 통해서 pc + 4로 target을 업데이트 하였다. not-taken에서 mispredict가 발생하였을 때는 위의 코드처럼 반대의 작업을 하면 된다.
마지막으로, (bne, bge, bltu) instruction이다. 이들은 alu의 연산 결과가 1, 즉 두 수를 비교한 결과가 참일때 not-taken해야 한다. 즉, 위의 (beq, blt, bltu) instruction과 비슷하게 mispredict를 확인하고 handle할 수 있다. 자세한 설명은 코드를 참고하면 될 것이다.
추가로, flush는 아래 코드와 같이 진행하였다. 각 pipeline register에 bubble에 해당하는 값을 대입해줌으로써 기존에 존재하던 instruction들을 flush해 bubble로 변화시켰다.

```
if flush:
    ID.reg_inst         = WORD(BUBBLE)
    ID.reg_exception    = WORD(EXC_NONE)
    ID.reg_pcplus4      = WORD(0)
    EX.reg_inst         = WORD(BUBBLE)
    EX.reg_exception    = WORD(EXC_NONE)
    EX.reg_c_br_type    = WORD(BR_N)
    EX.reg_c_rf_wen     = False
    EX.reg_c_dmem_en    = False
```

Figure 10: Handle mispredict branch

2.3 Explain the changes in the datapath and control signals

이 구현을 위해 datapath를 변경하였다. 기존에 EX stage에서 결과가 나오기 까지 기다리던 코드를 제거하고, lookup() 메소드로 바로 예측해서 진행하였다. misspredict를 확인하기 위해 taken 하였는지에 대한 여부가 필요했는데, 이를 구현하기 위해 ID, EX stage에 taken의 여부를 저장하는 pipeline register를 새로 만들어 IF stage에서 EX stage로 taken 여부를 전달해주었다. 위에서 설명한 self.taken에 대한 설명이다. 또, flush를 위해서 ID, EX stage의 pipeline register에 bubble의 instruction에 해당하는 값들을 대입하는 datapath를 추가하였다. 또, flush 된 이후, IF stage의 pc를 target으로 변경하는 과정 역시도 datapath를 추가해서 해결하였다.
control signal은 변경하지 않았다. 기존에 존재하던 ID, EX stage에 bubble을 만들어 주는 signal은 제거하고, 직접 pipeline register를 bubble로 바꾸도록 하였다. 이처럼, 위에서 설명한 작업들을 모두 datapath를 이용하여 진행하여서, control logic 상의 변화는 크게 없다고 할 수 있다. 변화는

기존에 branch calculation, flush 등을 위해 존재하던 control signal들을 이용하지 않은 것이라고 말할 수 있을 것이다.

3 Part 3: Conclusion

한 학기 동안 어렵기도 했지만, 덕분에 많은 것을 배워가고, 좋은 경험을 많이 해볼 수 있었던 것 같습니다. 수업을 들으면서 기존에 관심 없던 low level에도 새롭게 관심을 가지게 되어 앞으로 더욱 열심히 공부해 볼 생각입니다. 아마 서울대학교를 다니면서 이런 강의를 다시 들어볼 수 있을지 모르겠습니다ㅎㅎ. 특히, 늦은 밤이나 휴일에도 카톡방에서 바로바로 대답해주신 덕분에 공부가 더욱 잘 된 것 같습니다. 한 학기 동안 정말 감사했습니다!!!