Computer Architecture pa4 - Design Document

경영학과 2017-15108
박지상

1. What does the overall pipeline architecture look like?

뒷면 별도 첨부 (Datapath 중심으로 작성)

2. (About Part 1) When do structural / data hazards occur and how do you deal with them?

1) Structural Hazard: M1-M2 hazard

SNURISC6에서는 메모리 접근을 시작하는 M1단계와 메모리 접근을 완료하는 M2단계가 있다. 따라서 연속된 두 instruction이 data memory access를 필요로 하는 load 또는 store인 경우, 앞선 instruction의 처리가 완료될 때까지 뒤의 instruction은 1 cycle 동안 대기해야한다.

Hazard는 모두 ID 단계에서 탐지되어야하기 때문에, ID 단계의 instruction과 EX 단계의 instruction의 데이터 메모리 접근 관련 signal (c_dmem_en)을 AND 연산으로 체크하여 탐지한다. 만약 M1-M2 hazard가 탐지되었다면, IF와 ID 단계의 instruction을 1 cycle 동안 stall한다. 그리고 다음 cycle의 EX 단계에는 BUBBLE을 삽입하여 별도의 instruction이 수행되지 않도록한다.

```
# Check for M1-M2 hazard
M1_M2_hazard = self.c_dmem_en and Pipe.EX.c_dmem_en
```

2) Data Hazard: Read-After-Write Hazard

SNURISC6에서는 ALU의 operand로 들어가는 값이 레지스터의 값이고, 앞선 instruction들 중에서 해당 레지스터의 값을 덮어씌우는 instruction이 있는 경우에 오래된 값이 연산에 사용되므로 의도하지 않은 결과가 발생하게 된다. 따라서 앞선 instruction의 수행 결과가 WB 단계에서 레지스터에 값이 쓰여지기 전이더라도, 해당 값을 전달받아 레지스터에서 읽은 값을 대체해야한다.

어느 단계로부터 값을 전달받아야 하는지는 control signal(self.c_fwd_op1, self.c_fwd_op2, self.c_fwd_rs2)의 값으로 판별하며, EX, M1, M2, WB 단계 모두 dependency가 발생할 수 있다. (WB 단계에서 값이 레지스터에 쓰여지는 것이 ID단계에서 EXE pipeline register에 쓰여지는 것보다 늦기 때문에 WB 단계의 instruction과도 dependency가 발생한다.)

ID단계의 instruction과 A단계의 instruction과의 dependency를 판별하는 경우는 다음과 같다.
1) A단계의 instruction이 레지스터에 값을 쓰는 instruction인지 (c-rf-wen signal로 판별)
2) A단계의 instruction의 register destination이 0이 아닌경우 (x0의 값은 항상 0이므로)
3) A단계의 instruction의 rd 값과 ID단계의 instruction의 rs값과 같은 경우
위의 3조건을 AND 연산으로 체크하여 해당 operand가 사용되는 경우에 (self.c_rs1_oen, self.c_op2_sel == OP2_RS2, self.c_rs2_oen) forwarding signal을 할당한다. 또한, multiple writer의 상황을 고려하여 ID단계로부터 가까운 순서부터 판별하여 forwarding을 설정한다.

```
self.c_fwd_op1        =   FWD_EX      if (EX.reg_rd == Pipe.ID.rs1) and self.c_rs1_oen and   \
                                        (EX.reg_rd != 0) and EX.reg_c_rf_wen else    \
                          FWD_M1      if (M1.reg_rd == Pipe.ID.rs1) and self.c_rs1_oen and   \
                                        (M1.reg_rd != 0) and Pipe.M1.c_rf_wen else   \
                          FWD_M2      if (M2.reg_rd == Pipe.ID.rs1) and self.c_rs1_oen and   \
                                        (M2.reg_rd != 0) and Pipe.M2.c_rf_wen else   \
                          FWD_WB      if (WB.reg_rd == Pipe.ID.rs1) and self.c_rs1_oen and   \
                                        (WB.reg_rd != 0) and WB.reg_c_rf_wen else    \
                          FWD_NONE
```

코드로 표현된 hazard detection logic은 위와 같으며, dependency의 우선순위는 if-else의 로직에 따라 자동으로 구현되었다. 그리고, ID단계에서 forwarding signal에 따라서 값을 전달받아 operand의 값을 다음과 같이 설정한다.

```
self.op1_data = self.pc          if self.c_op1_sel == OP1_PC       else \
                Pipe.EX.alu_out if self.c_fwd_op1 == FWD_EX       else \
                Pipe.M1.alu_out if self.c_fwd_op1 == FWD_M1       else \
                Pipe.M2.wbdata  if self.c_fwd_op1 == FWD_M2       else \
                Pipe.WB.wbdata  if self.c_fwd_op1 == FWD_WB       else \
                rf_rs1_data
```

3) Data Hazard: Load-Use Hazard

2)의 Read-After-Write Hazard의 경우에서 앞선 instruction이 load인 경우에는 2)의 로직으로 hazard detection이 이루어졌더라도 곧바로 값을 전달받을 수 없다. 이는 load instruction의 경우에는 레지스터에 쓰여지는 값이 EX 단계에 결정되는 ALU의 결과값이 아니라, 메모리에서 읽어온 값이기 때문이다. 따라서 아직 값을 읽어오지 못한 EX와 M1 단계에서는 메모리에서 값을 읽어올 수 있을 때까지 delay를 발생시켜야한다. Load-use hazard의 판별은 다음과 같이 이루어진다.

```
EX_load_inst = EX.reg_c_dmem_en and EX.reg_c_dmem_rw == M_XRD
M1_load_inst = M1.reg_c_dmem_en and M1.reg_c_dmem_rw == M_XRD
load_use_hazard     = ((EX_load_inst and EX.reg_rd != 0) and              \
                      ((EX.reg_rd == Pipe.ID.rs1 and self.c_rs1_oen) or          \
                       (EX.reg_rd == Pipe.ID.rs2 and self.c_rs2_oen))) or \
                      ((M1_load_inst and M1.reg_rd != 0) and \
                       ((M1.reg_rd == Pipe.ID.rs1 and self.c_rs1_oen) or \
                        (M1.reg_rd == Pipe.ID.rs2 and self.c_rs2_oen)))
```

Load-use hazard가 탐지된 경우에, IF와 ID 단계의 instruction을 1 cycle 동안 stall한다.
그리고 다음 cycle의 EX 단계에는 BUBBLE을 삽입하여 별도의 instruction이 수행되지 않도록한다.

3. (About Part 2) When do control hazards occur and how do you deal with them with the BTFNT branch prediction scheme?

Control Hazard는 pc값에 변화를 줄 수도 있는 branch, jump instruction들에 대해서 발생한다. 이는 fetch해야하는 다음 instruction이 해당 instruction의 결과에 따라 달려있는데, 그 결과는 보통 바로 다음 instruction의 pc값을 결정하는 IF단계가 아닌 뒤에 있는 경우가 많기 때문이다.
따라서 일반적으로는 사전에 정해진 규칙에 따라 다음 pc값을 설정하는 예측(prediction) 절차와, 추후 단계에서 실제 결과가 나올 때 예측이 맞았다면 그대로 진행하고 틀렸다면 그에 따른 처리를 해주는 검증(verification) 절차로 이루어진다.

본 SNURISC6에서는 "Backward branch as Taken, Forward branch as Not Taken"에 따른 scheme을 택하며, JAL에 대해서는 "Always Taken", JALR에 대해서는 "Always Not Taken"을 scheme을 택한다.
Branch prediction은 IF 단계에서 이루어지는데, 먼저 instruction을 보고 branch instruction에 해당하는 경우에, 다음과 같이 분기를 나눠 각 상황에 맞게 다음 pc를 설정한다.

```
# Use Pipe.cpu.ras for the return address stack
if c_br_type == BR_J:
    # Jal Instruction - Always Taken
    self.pc_next = Pipe.cpu.adder_if.op(self.pc, imm)
elif c_br_type == BR_JR:
    # Jalr Instruction - Always Not Taken
    self.pc_next = self.pcplus4
```

```
else:
    if imm < 0:
        # Backward branch - Taken
        self.pc_next = Pipe.cpu.adder_if.op(self.pc, imm)
    else:
        # Forward branch - Not Taken
        self.pc_next = self.pcplus4
```

Branch Verification은 EX 단계에서 이루어지는데, JAL instruction은 misprediction이 발생하는 경우가 없으므로 고려하지 않고, JALR instruction은 매번 misprediction이 발생하므로 다음 pc(self.brjmp_target)을 ALU의 결과에 하위 1bit을 0으로 처리한 값으로 설정한다. 기타 branch operation의 경우에는 backward/forward 여부에 따라 misprediction 조건에 따라 처리한다.

```
if SWORD(self.op2_data) < 0:
    # Backward branch - Taken
    if (self.c_br_type == BR_EQ and self.alu_out != 1) or \
            (self.c_br_type == BR_NE and self.alu_out != 0) or \
            (self.c_br_type == BR_LT and self.alu_out != 1) or \
            (self.c_br_type == BR_GE and self.alu_out != 0) or \
            (self.c_br_type == BR_LTU and self.alu_out != 1) or \
            (self.c_br_type == BR_GEU and self.alu_out != 0):
        self.brjmp_miss = True
        self.brjmp_target = Pipe.cpu.adder_pcplus4.op(self.pc)
```

```
else:
    # Forward branch - Not Taken
    if (self.c_br_type == BR_EQ and self.alu_out == 1) or \
            (self.c_br_type == BR_NE and self.alu_out == 0) or \
            (self.c_br_type == BR_LT and self.alu_out == 1) or \
            (self.c_br_type == BR_GE and self.alu_out == 0) or \
            (self.c_br_type == BR_LTU and self.alu_out == 1) or \
            (self.c_br_type == BR_GEU and self.alu_out == 0):
        self.brjmp_miss = True
        self.brjmp_target = Pipe.cpu.adder_if.op(self.pc, self.op2_data)
```

만약 Verification 과정에서 misprediction 여부가 판별될 경우, signal EX.brjmp_miss = True로 설정이되며, EX.brjmp_target에 올바르게 산출된 pc값이 설정된다.
이후 ID단계의 Control Logic에서 branch misprediction이 탐지되었다면 ID와 IF 단계에 있는 instruction을 bubble로 만들기 위해 ID.ID_bubble값과 ID.EX_bubble값을 True로 설정한다.
해당 Signal이 Bubble로 설정된 경우, 해당 단계의 update()에서 다음 단계에 Bubble이 삽입된다.
이후 IF단계에서는 branch misprediction이 탐지되었다면 다음 pc의 값을 Pipe.EX.brjmp_target으로 설정하고, 이외의 처리를 진행하지 않는다. (self.pc_next = Pipe.EX.brjmp_target)

4. (About Part 3) How do you use RAS?

JALR은 branch prediction을 잘못할 가능성은 없지만, IF단계에서 레지스터의 ra 값을 읽어올 수가 없기에 (structural hazard) part2에서는 매번 not taken으로 처리하고 branch verification 단계에서 misprediction을 처리하는 방식으로 구현되었다.

Structural hazard에 의한 비효율을 줄이기 위해 RAS가 도입되었는데, 이는 ra값이 새롭게 추가되는 경우(즉, Jal과 Jalr로 function call이 발생하는 경우)에 스택에 ra값을 추가하고, jalr이 function return으로 사용되었을 때 스택의 값을 pop해서 사용하는 방식을 택하고 있다.

IF단계에서 branch prediction을 할 때, Jal과 Jalr이 function call로 쓰여졌다고 판단되는 상황인 rd 값이 1인 경우에는 pc+4값을 ras에 푸쉬한다. 또한 Jalr이 function return으로 쓰여졌다고 판단되는 상황인 rd = 0이고 rs1 = 1이며 imm = 0인 경우에는 ras의 값을 pop해서 사용한다. 이 때 ras가 빈 stack이었을 확률이 있기 때문에, pop()의 리턴 값인 status를 체크하여 False인 경우에는 ras값을 사용하지 않고 pc+4값을 활용한다.

```
if c_br_type == BR_J:
    # Jal Instruction - Always Taken
    self.pc_next = Pipe.cpu.adder_if.op(self.pc, imm)
    if RISCV.rd(self.inst) == 1:
        Pipe.cpu.ras.push(self.pcplus4)
```

```
elif c_br_type == BR_JR:
    # Jalr Instruction - Always Not Taken
    self.pc_next = self.pcplus4

    if RISCV.rd(self.inst) == 1:
        Pipe.cpu.ras.push(self.pcplus4)

    if RISCV.rd(self.inst) == 0 and \
        RISCV.rs1(self.inst) == 1 and \
        imm == 0:
        self.pc_next, status = Pipe.cpu.ras.pop()
        if not status:
            self.pc_next = self.pcplus4
```

다만 Jalr instruction에 대해서 ras에서 pop된 값으로 predict했다고 하더라도, 해당 값이 항상 옳다는 보장은 없다. 따라서 EX 단계에서 branch verification을 할 때 ras에서 받았던 address값 (ID 단계에 있는 instruction의 address값: Pipe.ID.reg_pc)과 ALU에서 실제로 계산된 원래 target address를 비교하여 검증한다. 이 때 만약 두 값이 다르다면, 계산된 target address를 brjmp_target으로 설정하고 branch misprediction이 발생했다는 signal인 self.brjmp_miss값을 True로 설정해주고, 이에 대한 처리하는 Part2에서와 동일하게 처리한다.

```
self.brjmp_miss = False
if self.c_br_type != BR_N:
    if self.c_br_type == BR_JR:
            # Jalr instruction - MISPREDICTED
        if RISCV.rd(self.inst) == 0 and \
                RISCV.rs1(self.inst) == 1 and \
                SWORD(self.op2_data) == 0:
            self.brjmp_target = self.alu_out & WORD(0xfffffffe)
            if self.brjmp_target != Pipe.ID.reg_pc:
                self.brjmp_miss = True
```