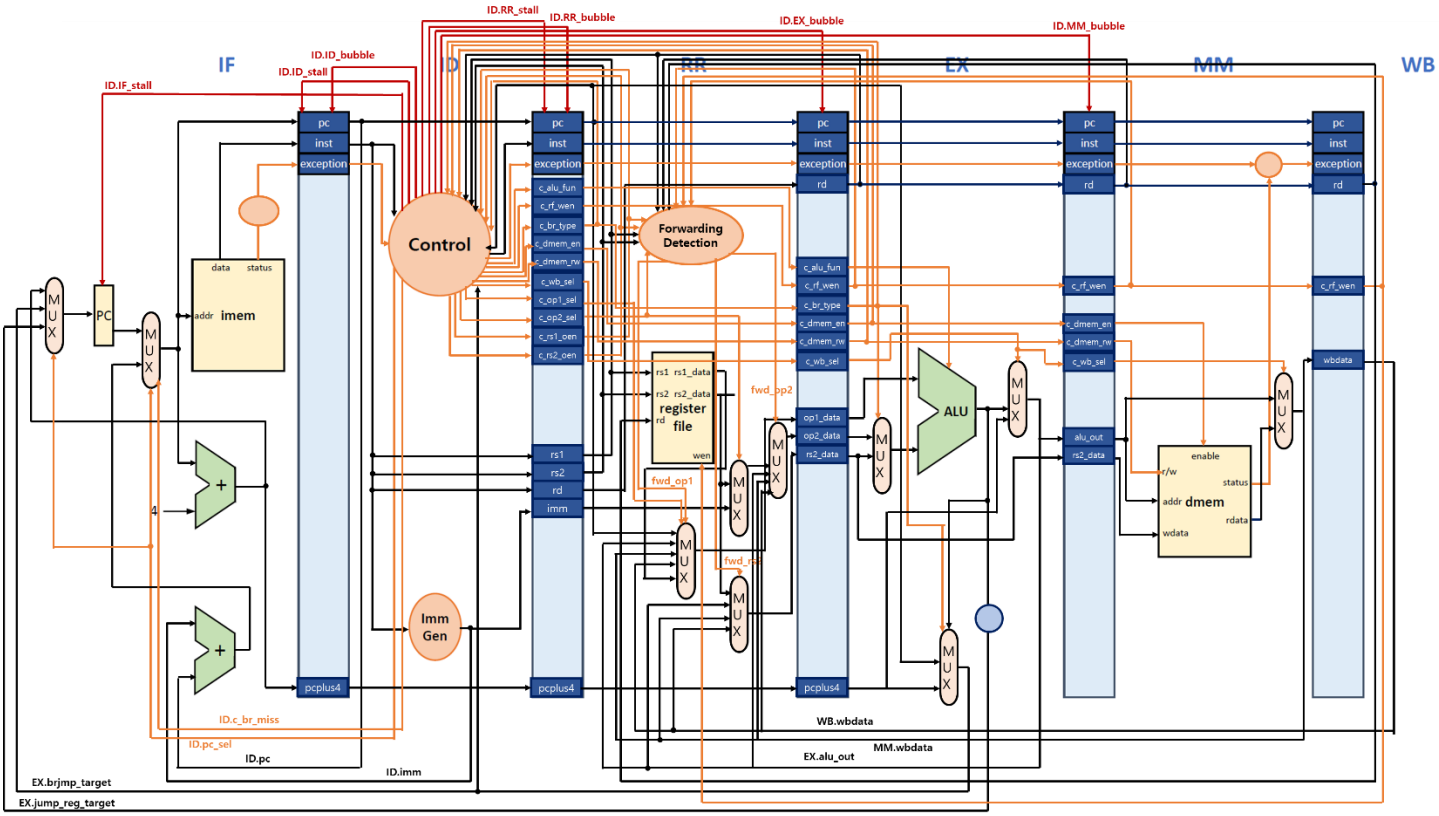
Computer Architecture – Project 4 - Report

2019-18499 Junhyeok Kim

1. Overall pipeline architecture



최대한 제가 작성한 코드에 맞게 회로를 그렸으며, 일부 다른 점이 존재할 수 있습니다(나름대로 검수를 했습니다). 코드 안의 주석은 수정을 거치면서 일부 코드와 완전히 대응되지 않는 부분이 있을 수 있어 약간의 혼란을 야기할 수 있는 점에 대해서는 죄송합니다. 또한 선이 조금 복잡해 충분한 확대를 이용하시면 선의 구분이 보일 것입니다.

2. When do data hazards occur and how do you deal with them?

Data hazard는 EX단계에서 Load가 실행되며, Load로 불러온 값을 저장하는 레지스터(rd)가 x0이 아니면서, 이 rd가 전 단계인 RR단계에서 사용되는 경우(rs1, rs2 모두)에 발생합니다. 이를 해결하기 위해 다음과 같은 코드를 짰습니다.

```
EX_load_inst = EX.reg_c_dmem_en and EX.reg_c_dmem_rw == M_XRD
load_use_hazard = (EX_load_inst and EX.reg_rd != 0) and \
                  ((EX.reg_rd == RR.reg_rs1 and RR.reg_c_rs1_oen) or \
                   (EX.reg_rd == RR.reg_rs2 and RR.reg_c_rs2_oen))
```

```
self.IF_stall = load_use_hazard
self.ID_stall = load_use_hazard
self.RR_stall = load_use_hazard
self.ID_bubble = EX_jalr or EX_brjmp
self.RR_bubble = EX_jalr or EX_brjmp
self.EX_bubble = load_use_hazard or EX_jalr or EX_brjmp
```

(ID단계의 compute(self)에 포함됨)

먼저, EX단계에서 Load하는 중인지, 즉 EX.reg_c_dmem_en과 EX.reg_C_dmem_rw 둘 다 M_XRD 값을 갖는지를 확인하며, 불러온 값을 저장하는 레지스터 rd가 0이 아닌지 EX.reg_rd != 0을 확인, 그리고 EX.reg_rd가 RR.reg_rs1 또는 RR.reg_rs2와 같으며, 각각의 레지스터(rs1, rs2)가 enable된 상태인지(RR.reg_c_rs1_oen, RR.reg_c_rs2_oen)를 동시에 확인하여 최종적으로 load_use_hazard가 true이면, Data hazard가 발생, False는 미발생으로 판단하여, 이는 IF_stall, ID_stall, RR_stall, EX_bubble을 true로 하여, hazard를 해결하였습니다.
이를 detect하기 위해 기존의 snurisc5와 비교하여, Control Unit이 약간 보강되었으며, RR단계 추가에 따라 Pipe.ID.RR_stall, Pipe.ID.RR_bubble이 추가되었습니다.

3. When do control hazards occur and how do youl deal with them with always-taken branch prediction scheme?

jalr을 제외한 jal, branch에 대해 always-taken branch prediction을 적용하였기 때문에, IF단계에서 predict했던 것이 EX단계에서 틀렸다는 것이 확인되었을 경우, 즉 EX단계에서 branch나 jal을 통해 jump를 하지 않는 경우, 또한 jalr을 통해 jump하게되는 경우에 Control hazard가 발생하게 됩니다. 이를 해결하기 위해 여러 단계에서 다음과 같은 코드를 추가 및 수정했습니다.

```
IF.reg_pc = Pipe.cpu.adder_brtarget.op(Pipe.ID.pc, Pipe.ID.imm) if Pipe.ID.pc_sel == PC_BRJMP else \
            IF.reg_pc
```

(IF단계 compute(self)에서 DO NOT TOUCH 부분 위쪽)

```
self.pc_next =  Pipe.EX.brjmp_target if Pipe.ID.c_br_miss else \
                self.pcplus4 if Pipe.ID.pc_sel == PC_4 else \
                self.pcplus4 if Pipe.ID.pc_sel == PC_BRJMP else \
                Pipe.EX.jump_reg_target if Pipe.ID.pc_sel == PC_JALR else \
                WORD(0)
```

(IF단계 compute(self)에서 DO NOT TOUCH 부분 아래쪽)

```
# Control signal to select the next PC
self.pc_sel = PC_BRJMP if self.c_br_type == BR_NE or \
                          self.c_br_type == BR_EQ or \
                          self.c_br_type == BR_GE or \
                          self.c_br_type == BR_GEU or \
                          self.c_br_type == BR_LT or \
                          self.c_br_type == BR_LTU or \
                          self.c_br_type == BR_J else \
              PC_JALR if EX.reg_c_br_type == BR_JR else \
              PC_4
```

(ID단계 compute(self) 내부 일부)

```
EX_brjmp = EX.reg_c_br_type in [BR_NE, BR_EQ, BR_GE, BR_GEU, BR_LT, BR_LTU, BR_J] and Pipe.EX.brjmp_target != RR.reg_pc

EX_jalr = self.pc_sel == PC_JALR

# For load-use hazard, ID and IF are stalled for one cycle (and EX bubbled)
# For mispredicted branches, instructions in ID and IF should be cancelled (become BUBBLE)
self.IF_stall = load_use_hazard
self.ID_stall = load_use_hazard
self.RR_stall = load_use_hazard
self.ID_bubble = EX_jalr or EX_brjmp
self.RR_bubble = EX_jalr or EX_brjmp
self.EX_bubble = load_use_hazard or EX_jalr or EX_brjmp
```

(ID단계 compute(self) 내부 일부)

```
self.ID.c_br_miss = EX_brjmp
```

(ID단계 compute(self) 내부 일부)

```
self.brjmp_target = RR.reg_pc if (self.c_br_type == BR_NE and (not self.alu_out)) or \
                                 (self.c_br_type == BR_EQ and self.alu_out) or \
                                 (self.c_br_type == BR_GE and (not self.alu_out)) or \
                                 (self.c_br_type == BR_GEU and (not self.alu_out)) or \
                                 (self.c_br_type == BR_LT and self.alu_out) or \
                                 (self.c_br_type == BR_LTU and self.alu_out) or \
                                 (self.c_br_type == BR_J) else \
                    self.pcplus4
```

(EX단계 compute(self) 내부 일부)

본래 snurisc5에서 EX단계에 존재했던 adder_brtarget을 IF단계로 옮기고, ID단계를 통해 계산된 Pipe.ID.pc_sel에 따라 IF.reg_pc를 유지하거나, Pipe.ID.pc와 Pipe.ID.imm의 합으로 바꿉니다. 본래 바로 직접 Pipe.IF.pc를 바꾸고 싶었으나, DO NOT TOUCH되어 있어, 이를 위해 Pipe.IF.reg_pc를 변경함으로써 이를 간접적으로 구현하였습니다.

IF단계의 pc_next 또한 약간 수정하였는데, 본래 Pipe.ID.pc_sel이 PC_BRJMP와 같을 때, Pipe.EX.brjmp_target의 값으로 pc_next가 바뀌는 대신에, 이때는 PC_4의 경우와 같이 self.pcplus4가 되도록 하였으며, ID 단계에서 추가한 Pipe.ID.c_br_miss를 이용하여, 이 값이 true일 때 Pipe.EX.brjmp_target의 값, 즉 후술할 것이지만 branch가 not-taken되었을 때 가야할 pc 값으로 pc_next가 바뀌게 됩니다.

ID단계에서 Pipe.ID.pc_sel은 기존에 always-not-taken 방식이던 것을 always-taken으로 바꿔주기 위해, Pipe.ID.c_br_type에 대해서 branch, jal일 때 Pipe.ID.pc_sel이 PC_BRJMP값을 갖도록 하였습니다. jalr에 대해서는 기존과 동일하게 두어 always-not-taken 방식을 유지하였습니다.

ID단계에서 내부에 기존의 always-not-taken 방식으로 동작했던 EX_brjmp를 jalr에서만 작동시키도록 EX_jalr로 교체, 이는 Pipe.ID.pc_sel이 PC_JALR 값을 가질 때 항상 true가 되도록 하였습니다. 실질적으로 새롭다고 할 수 있는 수정 후 EX_brjmp는 EX.reg_c_br_type이 branch와 jal인지, 그리고 EX단계에서 나온 branch의 결과와 prediction의 결과가 일치하는지 확인하기 위해 Pipe.EX.brjmp_target != RR.reg_pc를 확인하여 이 결과값이 EX_brjmp에 들어가도록 했습니다. 이 내부값을 외부로 이용하기 위해 Pipe.ID.c_br_miss에 이 값을 그대로 이용하였습니다(아예 EX.brjmp를 외부에서 이용할 수 있게 할 수 있지만, 단순히 수정을 안 한 것입니다). 그리고, 이 misprediction에 대해 ID, RR, EX에서 bubble이 되도록 하여 Control hazard를 해결했습니다.

EX단계에서는 앞서 말했던 올바른 Pipe.EX.brjmp_target을 산출하기 위해, 위와 같이 Pipe.EX.c_br_type의 상태(현 단계에서 branch나 jal이 실행되었는지)와 Pipe.EX.alu_out(branch 조건을 만족하는지)를 확인, 만족할 경우 jump 하여 이동하게 된(predict 성공하여 RR에서 실행되는 pc값과 같다) RR.reg_pc값을 가지며, 아닌 경우, mispredict되었다고 판단, 해당 instruction의 pc+4인 Pipe.EX.pcplus4를 갖게 하여, 앞서 말한 IF와 ID단계에서 비교 및 control hazard detect 및 resolve가 이루어지도록 하였습니다.

최종적으로 이 Control hazard를 해결하면서 hardware difference로, adder_brtarget이 EX단계에서 IF단계로 이동하였으며, ID단계에서 내부적으로 EX_jalr와 Pipe.ID.c_br_miss가 추가, 이외 IF, ID, EX단계에서 다수의 MUX의 이용에 변화가 발생하였습니다.