컴퓨터구조 PA3
2015-11923 최한결

Problem1.

```
if(opcode == R_FORMAT)
begin
        alu_src <= 1'b0;
        mem_to_reg <= 1'b0;
        reg_write <= 1'b1;
        mem_read <= 1'b0;
        mem_write <= 1'b0;
        halt <= 1'b0;
        branch <= 1'b0;
        alu_operation <= (funct3 == 3'b000) ? (funct7 == 7'h20 ? alu.SUB_64 :(funct7 == 7'h0 ? alu.ADD_64 : alu.ALU_NOP)) :
                                              ((funct7 != 7'h0 ? alu.ALU_NOP :
                                                      (funct3 == 3'b111 ? alu.BIT_AND :
                                                      (funct3 == 3'b110 ? alu.BIT_OR :
                                                      (funct3 == 3'b100 ? alu.BIT_XOR : alu.ALU_NOP)))));
end
else if(opcode == I_FORMAT)
begin
        alu_src <= 1'b1;
        mem_to_reg <= 1'b0;
        reg_write <= 1'b1;
        mem_read <= 1'b0;
        mem_write <= 1'b0;
        halt <= 1'b0;
        branch <= 1'b0;
        alu_operation <= (funct3 != 3'b000) ? (funct3 == 3'b111 ? alu.BIT_AND : (funct3 == 3'b110 ? alu.BIT_OR : (funct3 == 3'b100 ? alu.BIT_XOR : alu.ALU_NOP))) : alu.ADD_64;
end
```

AND, OR, XOR, ANDi, Ori, XORi 명령어는 inst_decoder에서 디코딩 된 후 포맷에 맞게 각 신호가 레지스터에 저장되므로 구현할 사항은 ALU의 operation만 구현하면 되었다. 따라서 control.v의 alu_operation을 opcode가 R format일때, I format일때 각각을 위와 같이 변경하였다. 이를 이해하기 쉽게 써보면 각각 다음과 같다.

```
if(funct3 == 0b000){
  if(funct7 == 0x20)
    alu_operation = alu.SUB_64;
  else{
    if(funct7 == 0x0)
      alu_operation = alu.ADD_64;
    else
      alu_operation = alu.ALU_NOP;
  }
}
else{
  if(funct7 != 0x0)
    alu_operation = alu.ALU_NOP;
  else if(funct3 == 0b111
    alu_operation = alu.BIT_AND;
  else if(fucnt3 == 0b110)
    alu_operation = alu.BIT_OR;
  else if(funct3 == 0b100)
    alu_operation = alu.BIT_XOR;
  else
    alu_operation = alu.ALU_NOP;
}
```

```
if(funct3 != 0b000){
  if(funct3 == 0b111
    alu_operation = alu.BIT_AND;
  else if(fucnt3 == 0b110)
    alu_operation = alu.BIT_OR;
  else if(funct3 == 0b100)
    alu_operation = alu.BIT_XOR;
  else
    alu_operation = alu.ALU_NOP;
  }
}
else
  alu_operation = alu.ADD_64;
```

funct3과 funct7의 값에 따라 RISC-V ISA의 규칙에 맞추어 다음과 같이 alu_operation만 수정해주면 AND, OR, XOR, ANDi, Ori, XORi가 정상 작동한다.

Problem2.

```
assign hazard = EX_branch | MEM_branch | (EX_mem_read & (EX_rd == ID_rs1 | EX_rd == ID_rs2));
```

hazard detection을 다음과 같이 구현할 수 있다. problem4까지 마쳤을 때 구현된 pipeline은 branch prediction을 하지 않고 forwarding을 하므로 hazard는 branch연산이 일어날 때와 load-use hazard 두 경우이다. 따라서 hazard가 일어나는 경우는 위와 같다. (beq명령이 들어올 때 branch가 1이 되어 각 stage마다 전달된다.)

```
always @(posedge clk)
begin
        IF_pc <= reset ? 64'h0 : ((MEM_branch & MEM_zero) ? MEM_branch_address : (hazard ? IF_pc : IF_pc + 4));
        ID_inst <= reset ? 32'h0 : ((MEM_branch & MEM_zero) ? IF_inst : (hazard ? ID_inst : IF_inst));
end
//여기서 hazard detection시에 pc를 유지, ID_inst를 nop으로 설정
```

hazard가 일어났을 때 IF_stage에서 PC를 그대로 유지시키고 ID_inst도 그대로 유지시킨다.

```
//ID/EX pipeline registers
always @(posedge clk)
begin
        EX_alu_src      <= (reset) ? 1'h0 : (hazard ? 1'h0 : ID_alu_src);
        EX_mem_to_reg   <= (reset) ? 1'h0 : (hazard ? 1'h0 : ID_mem_to_reg);
        EX_reg_write    <= (reset) ? 1'h0 : (hazard ? 1'h0 : ID_reg_write);
        EX_mem_read     <= (reset) ? 1'h0 : (hazard ? 1'h0 : ID_mem_read);
        EX_mem_write    <= (reset) ? 1'h0 : (hazard ? 1'h0 :  ID_mem_write);
        EX_alu_op       <= (reset) ? 3'h0 : (hazard ? 3'h0 : ID_alu_op);
        EX_halt         <= (reset) ? 1'h0 : (hazard ? 1'h0 : ID_halt);
        EX_branch <= (reset) ? 1'h0 : (hazard ? 1'h0 : ID_branch);

        EX_rs1_data     <= (reset) ? 64'h0 : (hazard ? 64'h0 :
                                (MEM_mem_read & MEM_rd == ID_rs1) ? MEM_read_data :
                                (WB_mem_read & WB_rd == ID_rs1) ? WB_mem_rddata :
                                ((EX_reg_write & ~(EX_rd == 5'h0) & (EX_rd == ID_rs1)) ? EX_alu_result :
                                ((MEM_reg_write & ~(MEM_rd == 5'h0) & (MEM_rd == ID_rs1)) ? MEM_alu_result :
                                ((WB_reg_write & ~(WB_rd == 5'h0) & (WB_rd == ID_rs1)) ? WB_alu_result : ID_rf_data_rs1))));
        EX_rs2_data     <= (reset) ? 64'h0 : (hazard ? 64'h0 :
                                (MEM_mem_read & MEM_rd == ID_rs2) ? MEM_read_data :
                                (WB_mem_read & WB_rd == ID_rs2) ? WB_mem_rddata :
                                ((EX_reg_write & ~(EX_rd == 5'h0) & (EX_rd == ID_rs2)) ? EX_alu_result :
                                ((MEM_reg_write & ~(MEM_rd == 5'h0) & (MEM_rd == ID_rs2)) ? MEM_alu_result :
                                ((WB_reg_write & ~(WB_rd == 5'h0) & (WB_rd == ID_rs2)) ? WB_alu_result : ID_rf_data_rs2))));
        EX_imm64        <= (reset) ? 64'h0 : (hazard ? 64'h0 : ID_imm64);
        EX_rd           <= (reset) ? 5'h0 : (hazard ? 5'h0 : ID_rd);
end
```

hazard가 발생하면 ID/EX 레지스터의 모든 신호를 0으로 바꾸어 NOP가 명령어를 대체하게 한다.
다음과 같이 구현했을 경우 hazard가 일어나면 PC가 유지되어 ID stage까지는 계속 hazard가 일어난 다음 명령어를 읽어오고 EX stage이후엔 NOP가 명령어를 대체하게 된다. hazard가 끝나면 PC값이 다시 증가하기 시작하고 실행이 되지 않은 명령어부터 정상적으로 실행되기 시작한다.

Problem3.

```
assign sign_extended = inst[31] ? 52'hfffffffffffff : 52'h0;
assign imm64 =  ((opcode == 7'b0110011) ? 64'h0 :                                   //R type (IMM = DC)
                ((opcode[6:5] == 2'b00) ? {sign_extended, inst[31:20]} :            //I type
                ((opcode[6:5] == 2'b11) ? {sign_extended, inst[7], inst[30:25], inst[11:8], 1'b0} : //Beq
                {sign_extended, inst[31:25], inst[11:7]})));                        //SW: S type, I-imm
```

우선 beq를 구현하기 위해 beq의 format인 SB format의 decoder를 inst_decoder.v에 추가하였다. 각 위치의 값을 읽고 1bit 0을 imm[0]에 추가해준다.

```
else if(opcode == BEQ)
begin
        //추가
        alu_src <= 1'b0;
        mem_to_reg <= 1'b0;
        reg_write <= 1'b0;
        mem_read <= 1'b0;
        mem_write <= 1'b0;
        halt <= 1'b0;
        branch <= 1'b1;
        alu_operation <= alu.SUB_64;
end
```

BEQ의 control signal은 다음과 같다. 두 레지스터의 값을 빼고 그 값이 0이면 PC를 업데이트 하므로 기본적으로 SUB명령어의 control에 branch값만 다르다.

```
//EX/MEM pipeline registers
always @(posedge clk)
begin
        MEM_mem_to_reg  <= (reset) ? 1'h0 : EX_mem_to_reg;
        MEM_reg_write   <= (reset) ? 1'h0 : EX_reg_write;
        MEM_mem_read    <= (reset) ? 1'h0 : EX_mem_read;
        MEM_mem_write   <= (reset) ? 1'h0 : EX_mem_write;
        MEM_halt        <= (reset) ? 1'h0 : EX_halt;
        MEM_branch <= (reset) ? 1'h0 : EX_branch;

        MEM_rs2_data    <= (reset) ? 64'h0 : EX_rs2_data;
        MEM_alu_result  <= (reset) ? 64'h0 : EX_alu_result;
        MEM_zero        <= (reset) ? 1'h0 : alu_zero;
        MEM_sign        <= (reset) ? 1'h0 : alu_sign;
        MEM_rd          <= (reset) ? 5'h0 : EX_rd;
        MEM_branch_address <= (reset) ? 64'h0 : IF_pc + EX_imm64 - 8;

end
```

MEM_branch를 추가해 branch연산이 실행중인지 체크하고 hazard detection에 전달한다. 또 MEM stage에서 branch_address를 계산해서 PC로 넘겨준다. (값이 IF_pc + EX_imm64 – 8 인 이유는 hazard detection에서 hazard가 발생했을 때 hazard가 발생한 다음 명령어가 ID_inst 계속 저장되고 PC는 hazard가 발생한 다음 다음 명령어를 가르키고 있다. 따라서 EX stage에서 branch가 실행될 때 branch명령어의 주소는 IF_pc – 8이다.)

```
always @(posedge clk)
begin
        IF_pc <= reset ? 64'h0 : ((MEM_branch & MEM_zero) ? MEM_branch_address : (hazard ? IF_pc : IF_pc + 4));
        ID_inst <= reset ? 32'h0 : ((MEM_branch & MEM_zero) ? IF_inst : (hazard ? ID_inst : IF_inst));
end
//여기서 hazard detection시에 pc를 유지, ID_inst를 nop으로 설정
```

branch 명령어가 EX stage에서 연산이 끝나고 두 레지스터가 같을 때 (두 값을 뺐을 때 MEM_zero가 0일 때) IF_pc는 MEM_branch_address를 가르킨다.

Problem4.

```
//ID/EX pipeline registers
always @(posedge clk)
begin
        EX_alu_src      <= (reset) ? 1'h0 : (hazard ? 1'h0 : ID_alu_src);
        EX_mem_to_reg   <= (reset) ? 1'h0 : (hazard ? 1'h0 : ID_mem_to_reg);
        EX_reg_write    <= (reset) ? 1'h0 : (hazard ? 1'h0 : ID_reg_write);
        EX_mem_read     <= (reset) ? 1'h0 : (hazard ? 1'h0 : ID_mem_read);
        EX_mem_write    <= (reset) ? 1'h0 : (hazard ? 1'h0 :  ID_mem_write);
        EX_alu_op       <= (reset) ? 3'h0 : (hazard ? 3'h0 : ID_alu_op);
        EX_halt         <= (reset) ? 1'h0 : (hazard ? 1'h0 : ID_halt);
        EX_branch <= (reset) ? 1'h0 : (hazard ? 1'h0 : ID_branch);

        EX_rs1_data     <= (reset) ? 64'h0 : (hazard ? 64'h0 :
                                (MEM_mem_read & MEM_rd == ID_rs1) ? MEM_read_data :
                                (WB_mem_read & WB_rd == ID_rs1) ? WB_mem_rddata :
                                ((EX_reg_write & ~(EX_rd == 5'h0) & (EX_rd == ID_rs1)) ? EX_alu_result :
                                ((MEM_reg_write & ~(MEM_rd == 5'h0) & (MEM_rd == ID_rs1)) ? MEM_alu_result :
                                ((WB_reg_write & ~(WB_rd == 5'h0) & (WB_rd == ID_rs1)) ? WB_alu_result : ID_rf_data_rs1))));
        EX_rs2_data     <= (reset) ? 64'h0 : (hazard ? 64'h0 :
                                (MEM_mem_read & MEM_rd == ID_rs2) ? MEM_read_data :
                                (WB_mem_read & WB_rd == ID_rs2) ? WB_mem_rddata :
                                ((EX_reg_write & ~(EX_rd == 5'h0) & (EX_rd == ID_rs2)) ? EX_alu_result :
                                ((MEM_reg_write & ~(MEM_rd == 5'h0) & (MEM_rd == ID_rs2)) ? MEM_alu_result :
                                ((WB_reg_write & ~(WB_rd == 5'h0) & (WB_rd == ID_rs2)) ? WB_alu_result : ID_rf_data_rs2))));
        EX_imm64        <= (reset) ? 64'h0 : (hazard ? 64'h0 : ID_imm64);
        EX_rd           <= (reset) ? 5'h0 : (hazard ? 5'h0 : ID_rd);
end
```

ID/EX register에 rs1 data와 rs2 data를 업데이트 할 때 load명령어에서의 forwarding과 다른 연산 명령어로 인해 register가 업데이트 될 때의 forwarding을 나누어 구현했다. load명령어가 실행되고 있을 때, 즉 MEM_mem_read 혹은 WB_mem_read가 true이고 그 load명령의 rd와 현재 ID stage의 rs1 혹은 rs2가 같을 때 메모리에서 읽어온 값으로 ID/EX register의 rs1혹은 rs2의 값을 업데이트한다. (load 명령어가 들어온 후 한 사이클 stall이 일어나므로 EX_mem_read는 고려하지 않아도 된다.)
또 EX_reg_write,MEM_reg_write, WB_reg_write가 true이고 register값을 변경하는 그 연산 명령어와 현재 ID stage의 rs1혹은 rs2가 같을 때 또한 연산이 끝난 결과값으로 ID/EX register의 rs1 혹은 rs2의 값을 업데이트한다.