20180188 서승연 20180373 김현지

Lab5. Pipelined CPU

Introduction

이번 과제의 목표는 Pipelined CPU를 구현하는 것이다. Pipeline은 동시에 여러 instruction을 수행함으로써 cpu 속도를 빠르게 만들어 준다. 하나의 instruction이 모든 과정을 수행할 때까지 기다리는 것이 아니라 수행과정을 단계로 나누어 각 단계를 수행하는 명령어가 하나가 되도록 만드는 것이다. 이 방법을 이용하면 resource를 최대한 활용할 수 있어서 효율적이다.

Pipelined CPU는 data, control hazard를 발생시킨다. 이번 과제에서는 이러한 hazard를 해결한 Pipelined cpu를 만드는 것이 목표다.

Data hazard는 어떤 instruction이 필요로 하는 값이 register에 write되지 않은(준비되지 않은) 상황에서 일어난다. 이는 stall과 forwarding으로 해결할 수 있다. stall은 데이터가 register file에 write될 때까지 명령어 수행을 멈추고 기다리는 것이다. 이 방식은 cpu clk을 낭비하기 때문에 우리는 forwarding을 사용한다. forwarding은 data가 write 될 때까지 기다리지 않고 이전 명령어에 의해 값이 produce된 순간 (계산된 순간) 뒤의 instruction에 전달하여 곧바로 사용할 수 있게 하는 방법이다.

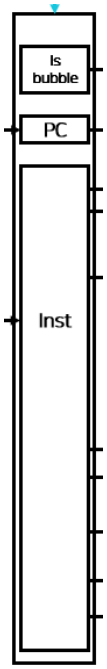
branch / jump instruction은 PC값이 특정한 주소로 설정된다. 이 명령어들은 PC 값에 대한 hazard를 발생시킨다. 이를 control hazard라고 한다. PC값이 정해질 때까지 기다리는 것은 clk을 낭비하는 일이므로 우리는 어떤 명령어를 반드시 실행하고 그것이 잘못된 실행인 경우 flush하는 방식을 선택해야 한다. 어떤 명령어를 실행할 것인지 결정하는 것이 branch prediction이다. 이번 lab 과제에서는 always not taken 즉, 항상 PC+1로 예측하여 구현하였다.

Design

1. Pipeline Register

IF/ID, ID/EX, EX/MEM, MEM/WB 각 stage사이 마다 어떠한 Pipeline register를 저장해야 할지 결정하였다.

a. IF/ID Pipeline Register

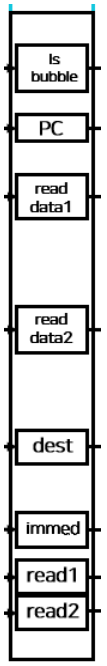


is_bubble : 해당 instruction이 bubble인지 여부를 저장한다.

PC : PC+1값을 저장한다.

Inst : Instruction 값을 저장한다.

b. ID/EX Pipeline Register



is_bubble : 해당 instruction이 bubble인지 여부를 저장하며, IF/ID pipeline register로부터 값을 받는다.

PC : IF/ID PC 값을 통해 저장하며, pc값을 register 2에 write back할때 필요하다.

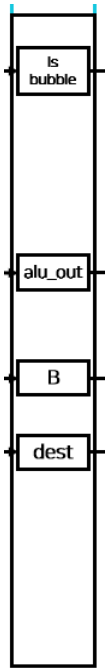
read data1, 2 : register file로 부터 읽은 값을 저장한다.

dest : write back할때 필요한 write destination register 값을 넘겨준다.

immed : immediate 값을 저장한다.

read1,2 : rs, rt값을 저장하며 forwarding에 필요하다.

c. EX/MEM Pipeline Register



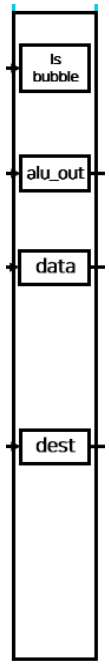
is bubble : 위의 pipeline register 설명과 같다.

alu_out : ALU에서 나온 alu result를 저장한다.

B : ALU계산에 사용한 두번째 input값을 저장하며, data memory의 write data로 사용된다.

dest : ID/EX의 dest 값을 넘겨받으며, WB단계에서 필요하다.

d. MEM/WB Pipeline Register



is bubble : 위의 pipeline register 설명과 같다.

alu_out : EX/MEM pipeline register의 alu_out를 넘겨 받으며, WB할 data로써 필요하다.

data : data memory에서 read한 data를 저장한다.

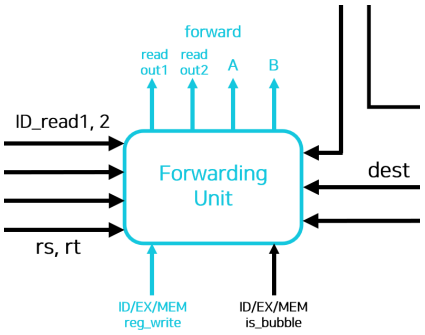
dest : EX/MEM의 dest 값을 넘겨받으며, WB단계에서 필요하다.

2. Submodules

Submodule은 기존에 구현했던 ALU, Control Unit, Register file, ImmGen과 추가로 구현한 Forwarding Unit, Hazard Detection Unit까지 총 6개가 있으며, 9개의 Mux들이 있다.

a. Forwarding Unit

Forwarding Unit은 write back할 destination register와 read할 register가 같을 경우 write될 결과를 forwarding해주는 역할을 수행한다. 보통 ALU input에 해당하는 A, B의 forwarding에 대해서만 다루지만, PC계산에 사용되는 register 혹은 bcond의 경우 ID stage에서 일어나기 때문에 ID의 readout에 대한 forwarding도 계산하였다.



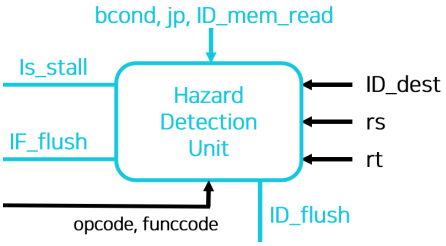
Input : rs,rt, ID_read1/2, ID/EX/MEM dest, ID/EX/MEM is_bubble
Signal : ID/EX/MEM reg_write (Forwarding 여부를 결정)
Output Signal : forward read out1/2 (ID 단계의 forwarding Mux의 signal)
forward A/B (EX단계의 forwarding Mux의 signal)

*** Mux Control Signal**

Forwarding이 일어나야하는 부분은 ALU계산을 위한 input값 A, B와 bcond 및 PC 계산을 위한 input값 read out1, 2이고, 이는 어떤 stage로 부터 forwarding이 일어났는지에 따라 한 input port에 다른 값이 들어간다. 이를 제어하기 위한 Mux가 총 4개 필요하고, 이 MUX 제어하는 signal 또한 필요하다. 이를 forwarding unit에서 output한다.

b. Hazard Detection Unit

Hazard Detection Unit은 load, jump, branch (taken)의 경우 flush 및 stall가 일어나도록 signal을 보내는 역할을 수행한다.



Input : rs, rt, ID_dest (Flush와 Stall 여부를 결정)
opcode, funccode (helper function use_read1,2의 결과를 결정)
Signal : bcond, jp, ID_mem_read (Flush와 Stall 여부를 결정)
Output Signal : is_stall, IF_flush, ID_flush

c. Control Unit

Control Unit에서는 각 instruction에 따라 필요한 control signal이 출력되도록 디자인하였다. 또한 Pipeline을 따라 각 단계에서 필요한 Control Signal들을 넘겨주어야 하기때문에, 이 단계에 대한 Signal 사이의 구분이 필요하다.

ID : pc_src, reg_dst, branch, branch_type, jp
EX : alu_op, alu_src_A/B
MEM : mem_read, mem_write
WB : mem_to_reg, reg_write, wwd, hlt,

*** Mux Control Signal**

먼저 PC, ALU Input1,2, Register의 Write data, Write Register는 한 input port에 instruction에 따라 다른 값이 들어간다. 이를 제어하기 위한 Mux가 총 5개 필요하고, 이 MUX 제어하는 signal 또한 필요하다.

- PC - pc_src
- ALU Input 1 - alu_src_A
- ALU Input 2 - alu_src_B
- Write Data - mem_to_reg
- Write Dest - reg_dst

Control Unit의 구현 자체는 기존의 Control Unit과 거의 유사하며, Pipeline 단계에 따른 구분 및 signal 이동은 Top level 에서 구현하도록 한다. 기존의 Control unit output의 한 가지 차이점은 PC 계산이 일찍 이루어짐에 따라, bcond을 통해 pc_src의 결과를 Control Unit에서 조정한다.

d. ALU, ImmGen, Register File

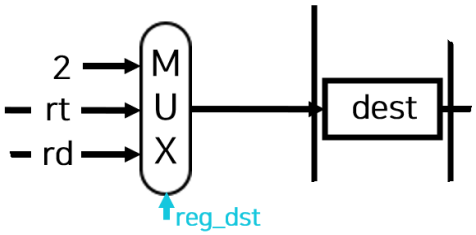
ALU의 경우 이전의 Lab에서 제작하였던 방식과 거의 유사하며, bcond의 결정이 ID단계에서 이루어져야하므로 이에 대한 기존의 기능을 제거하였다.

ImmGen의 경우 이전의 Lab에서 제작하였던 방식과 같은 방식으로 구현한다.

Register File의 경우 이전의 Lab에서 제작하였던 방식을 사용하되, register read를 하는 동시에 register write를 할때 write reg와 read reg가 같을 경우, 저장되어있던 register값이 아닌 write data값을 읽도록 한다.

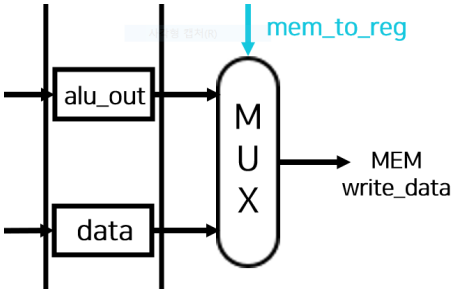
e. Mux

- **Write Dest Mux**



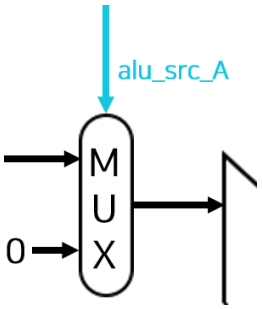
Mux Signal : reg_dst - 0 : rd
- 1 : rt
- 2 : 2

- **Write Data Mux**



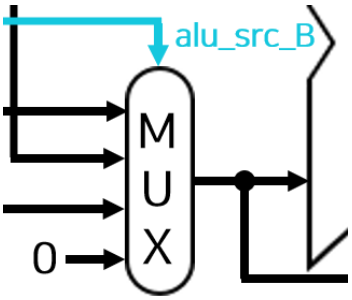
Mux Signal : mem_to_reg - 0 : alu_out
- 1 : data

- **ALUSrcA Mux**



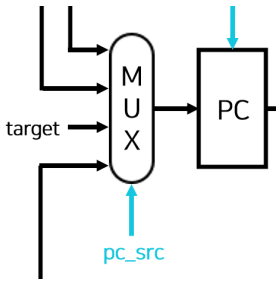
Mux Signal : reg_dst - 0 : forwardA
- 1 : 0

- **ALUSrcB Mux**



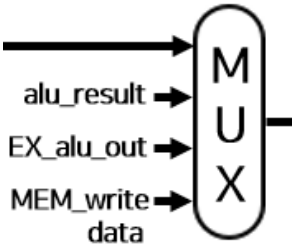
Mux Signal : alu_src_B - 0 : forwardB
- 1 : immed
- 2 : PC
- 3 : 0

■ **PCMux**



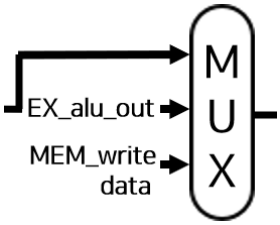
Mux Signal : pc_src - 0 : PC + 1
- 1 : IF_PC + immed
- 2 : IF_PC , target
- 3 : Readout1

■ **Forward Readout1,2 Mux**



Mux Signal : forward_read_out1 or 2
- 0 : read_out1 or 2
- 1 : alu_result
- 2 : EX_alu_out
- 3 : MEM_write_data (write data mux 결과)

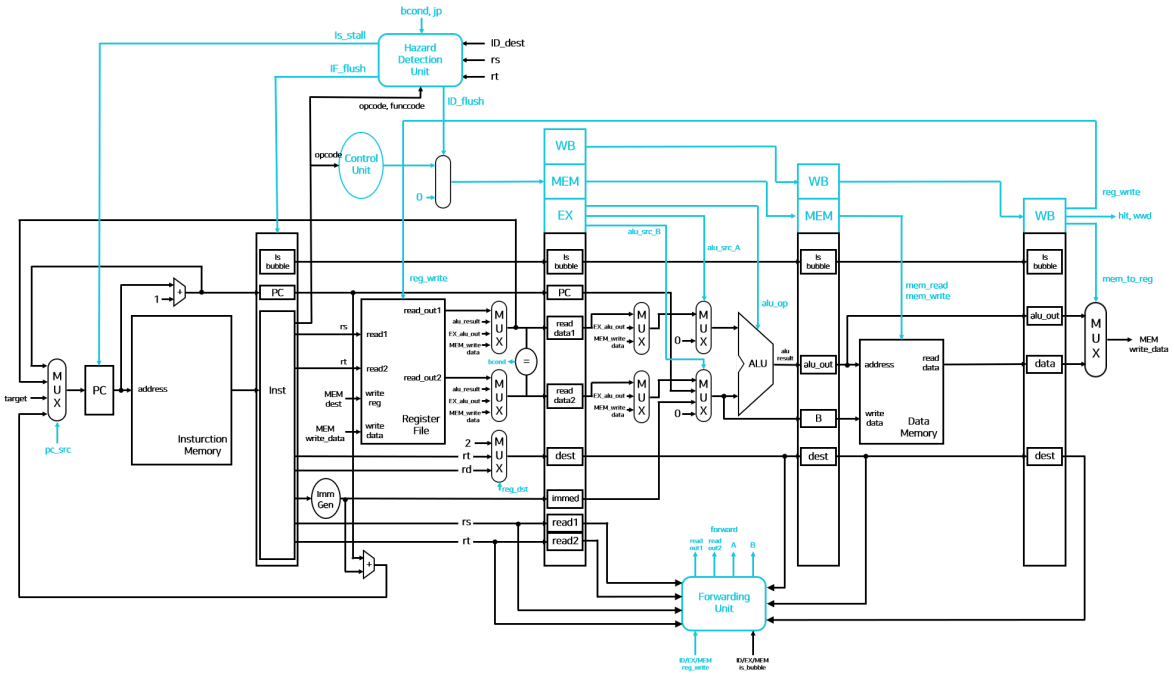
■ **Forward A, B Mux**



Mux Signal : forward_A or B
- 1 : ID_read_data1 or 2
- 2 : EX_alu_out
- 3 : MEM_write_data (write data mux 결과)

3. Top-level module (CPU)

Top level module인 cpu의 DataPath를 아래와 같이 디자인하였다.
(위의 각 module 및 mux, Pipeline Register를 상황에 맞게 연결한 것이다.)



대부분의 주요 기능들은 submodule을 통해 처리하고, Instruction decode, readout를 통한 bcond 계산, wwd와 hlt에 대한 output signal 처리, Pipeline register의 값을 clk synchronous하게 넘겨주는 등의 세부 구현을 cpu에서 처리하도록 한다.

Implementation

1. Forwarding Unit

forwarding unit에서는 IF단계의 read register rs, rt와 ID단계의 read register read1,2를 input으로 받아, ID/EX/MEM 단계의 각각의 write back destination register와 같은지, 그때 write back이 일어나는지, 해당 instruction이 bubble이 아닌지를 확인하고 그에 맞게 forwarding을 수행한다. Forward에 사용되는 macro는 4개가 있으며, 각 macro는 어디서 forwarding이 일어나는지를 의미한다.

```
always @(*) begin
    // Forwarding A
    if      (ID_read1 == EX_dest && EX_reg_write && !EX_is_bubble)    forward_A = `FORWARD_EX;
    else if (ID_read1 == MEM_dest && MEM_reg_write && !MEM_is_bubble) forward_A = `FORWARD_MEM;
    else                                                              forward_A = `FORWARD_ID;

    // Forwarding B
    if      (ID_read2 == EX_dest && EX_reg_write && !EX_is_bubble)    forward_B = `FORWARD_EX;
    else if (ID_read2 == MEM_dest && MEM_reg_write && !MEM_is_bubble) forward_B = `FORWARD_MEM;
    else                                                              forward_B = `FORWARD_ID;

    //Forwarding read_out1
    if      (IF_read1 == ID_dest && ID_reg_write && !ID_is_bubble)    forward_read_out1 = `FORWARD_ID;
    else if (IF_read1 == EX_dest && EX_reg_write && !EX_is_bubble)    forward_read_out1 = `FORWARD_EX;
    else if (IF_read1 == MEM_dest && MEM_reg_write && !MEM_is_bubble) forward_read_out1 = `FORWARD_MEM;
    else                                                              forward_read_out1 = `FORWARD_IF;

    //Forwarding read_out2
    if      (IF_read2 == ID_dest && ID_reg_write && !ID_is_bubble)    forward_read_out2 = `FORWARD_ID;
    else if (IF_read2 == EX_dest && EX_reg_write && !EX_is_bubble)    forward_read_out2 = `FORWARD_EX;
    else if (IF_read2 == MEM_dest && MEM_reg_write && !MEM_is_bubble) forward_read_out2 = `FORWARD_MEM;
    else                                                              forward_read_out2 = `FORWARD_IF;
end
```

2. Hazard Detection Unit

Hazard Detection Unit에서는 load instruction, jump instruction, taken된 branch instruction에 대해 stall과 flush를 수행하는 unit이다.

```
if (num_clock > 1) begin
    case (opcode)
        `ADI_OP : begin use_read1 = 1; use_read2 = 0; end
        `ORI_OP : begin use_read1 = 1; use_read2 = 0; end
        `LHI_OP : begin use_read1 = 0; use_read2 = 0; end
        `LWD_OP : begin use_read1 = 1; use_read2 = 0; end
        `SWD_OP : begin use_read1 = 1; use_read2 = 1; end

        `BNE_OP : begin use_read1 = 1; use_read2 = 1; end
        `BEQ_OP : begin use_read1 = 1; use_read2 = 1; end
        `BGZ_OP : begin use_read1 = 1; use_read2 = 0; end
        `BLZ_OP : begin use_read1 = 1; use_read2 = 0; end

        `JMP_OP : begin use_read1 = 0; use_read2 = 0; end
        `JAL_OP : begin use_read1 = 0; use_read2 = 0; end

        `ALU_OP : begin
            case (funccode)
                `INST_FUNC_JPR : begin use_read1 = 1; use_read2 = 0; end
                `INST_FUNC_JRL : begin use_read1 = 1; use_read2 = 0; end
                `INST_FUNC_WWD : begin use_read1 = 1; use_read2 = 0; end
                `INST_FUNC_HLT : begin use_read1 = 0; use_read2 = 0; end
                default        : begin use_read1 = 1; use_read2 = 1; end
            endcase
        end
    endcase
end
```

위의 코드와 같이 각 instruction에 대해 help function인 use read의 결과를 계산해준다. 각각의 instruction이 rs, rt를 계산에 사용하는지를 생각하면 이해하기 편하다.

```
if (ID_mem_read && ((ID_dest == read1 && use_read1) || (ID_dest == read2 && use_read2))) begin
    IF_flush = 0;
    ID_flush = (num_clock > 2) ? 1 : 0;
    is_stall = (num_clock > 2) ? !is_stall : 0;
end
else if (bcond | jp) begin
    IF_flush = 1;
    ID_flush = (num_clock > 2) ? 1 : 0;
    is_stall = 0;
end
else begin IF_flush = 0; ID_flush = 0; is_stall = 0; end
```

첫번째 조건문의 경우 load instruction이후 해당 register를 계산에 사용하는 instruction이 올 경우에 대해 stall과 IF flush를 해주는 코드이다.

두번째 조건문의 경우 branch taken이 일어나거나 jump의 경우 PC+1로 즉 not taken으로 예측한 instruction에 대해 flush를 해주는 코드이다.

그 외의 상황에 대해서는 어떤 시그널도 보내지 않는다.

3. Control Unit

control unit에서는 각 instruction의 기능에 맞는 control signal을 설정한다.

```
always @(*) begin
    case (opcode)
        `ADI_OP : begin alu_op = `FUNC_ADD; alu_src_A = `ALUSrcA_A; alu_src_B = `ALUSrcB_IMM; mem_read = 0; mem_write = 0; mem_to_reg = 0; reg_dst = `RegDst_RT;
        reg_write = 1; pc_src = `PC_DEF; hlt = 0; wwd = 0; branch_type = `BRANCH_NOT; branch = 0; jp = 0; end
        `ORI_OP : begin alu_op = `FUNC_ORR; alu_src_A = `ALUSrcA_A; alu_src_B = `ALUSrcB_IMM; mem_read = 0; mem_write = 0; mem_to_reg = 0; reg_dst = `RegDst_RT;
        reg_write = 1; pc_src = `PC_DEF; hlt = 0; wwd = 0; branch_type = `BRANCH_NOT; branch = 0; jp = 0; end
        `LHI_OP : begin alu_op = `FUNC_ADD; alu_src_A = `ALUSrcA_0; alu_src_B = `ALUSrcB_IMM; mem_read = 0; mem_write = 0; mem_to_reg = 0; reg_dst = `RegDst_RT;
        reg_write = 1; pc_src = `PC_DEF; hlt = 0; wwd = 0; branch_type = `BRANCH_NOT; branch = 0; jp = 0; end

        `LWD_OP : begin alu_op = `FUNC_ADD; alu_src_A = `ALUSrcA_A; alu_src_B = `ALUSrcB_IMM; mem_read = 1; mem_write = 0; mem_to_reg = 1; reg_dst = `RegDst_RT;
        reg_write = 1; pc_src = `PC_DEF; hlt = 0; wwd = 0; branch_type = `BRANCH_NOT; branch = 0; jp = 0; end
        `SWD_OP : begin alu_op = `FUNC_ADD; alu_src_A = `ALUSrcA_A; alu_src_B = `ALUSrcB_IMM; mem_read = 0; mem_write = 1; mem_to_reg = 0; reg_dst = `RegDst_RT;
        reg_write = 0; pc_src = `PC_DEF; hlt = 0; wwd = 0; branch_type = `BRANCH_NOT; branch = 0; jp = 0; end

        `BNE_OP : begin alu_op = `FUNC_SUB; alu_src_A = `ALUSrcA_A; alu_src_B = `ALUSrcB_B; mem_read = 0; mem_write = 0; mem_to_reg = 0; reg_write = 0;
        pc_src = (bcond) ? `PC_IMM : `PC_DEF; hlt = 0; wwd = 0; branch_type = `BRANCH_NE; branch = 1; jp = 0; end
        `BEQ_OP : begin alu_op = `FUNC_SUB; alu_src_A = `ALUSrcA_A; alu_src_B = `ALUSrcB_B; mem_read = 0; mem_write = 0; mem_to_reg = 0; reg_write = 0;
        pc_src = (bcond) ? `PC_IMM : `PC_DEF; hlt = 0; wwd = 0; branch_type = `BRANCH_EQ; branch = 1; jp = 0; end
        `BGZ_OP : begin alu_op = `FUNC_SUB; alu_src_A = `ALUSrcA_A; alu_src_B = `ALUSrcB_0; mem_read = 0; mem_write = 0; mem_to_reg = 0; reg_write = 0;
        pc_src = (bcond) ? `PC_IMM : `PC_DEF; hlt = 0; wwd = 0; branch_type = `BRANCH_GZ; branch = 1; jp = 0; end
        `BLZ_OP : begin alu_op = `FUNC_SUB; alu_src_A = `ALUSrcA_A; alu_src_B = `ALUSrcB_0; mem_read = 0; mem_write = 0; mem_to_reg = 0; reg_write = 0;
        pc_src = (bcond) ? `PC_IMM : `PC_DEF; hlt = 0; wwd = 0; branch_type = `BRANCH_LZ; branch = 1; jp = 0; end

        `JMP_OP : begin                                                    mem_read = 0; mem_write = 0; mem_to_reg = 0;
        reg_write = 0; pc_src = `PC_TAR; hlt = 0; wwd = 0; branch_type = `BRANCH_NOT; branch = 0; jp = 1; end
        `JAL_OP : begin alu_op = `FUNC_ADD; alu_src_A = `ALUSrcA_0; alu_src_B = `ALUSrcB_PC; mem_read = 0; mem_write = 0; mem_to_reg = 0; reg_dst = `RegDst_2;
        reg_write = 1; pc_src = `PC_TAR; hlt = 0; wwd = 0; branch_type = `BRANCH_NOT; branch = 0; jp = 1; end

        `ALU_OP : begin
            case (funccode)
                `INST_FUNC_JPR : begin                                                  mem_read = 0; mem_write = 0; mem_to_reg = 0;
                                 reg_write = 0; pc_src = `PC_REG; hlt = 0; wwd = 0; branch_type = `BRANCH_NOT; branch = 0; jp = 1; end
                `INST_FUNC_JRL : begin alu_op = `FUNC_ADD; alu_src_A = `ALUSrcA_0; alu_src_B = `ALUSrcB_PC; mem_read = 0; mem_write = 0; mem_to_reg = 0;
                 reg_dst = `RegDst_2; reg_write = 1; pc_src = `PC_REG; hlt = 0; wwd = 0; branch_type = `BRANCH_NOT; branch = 0; jp = 1; end
                `INST_FUNC_WWD : begin alu_op = `FUNC_ADD; alu_src_A = `ALUSrcA_A; alu_src_B = `ALUSrcB_0;  mem_read = 0; mem_write = 0; mem_to_reg = 0;
                                 reg_write = 0; pc_src = `PC_DEF; hlt = 0; wwd = 1; branch_type = `BRANCH_NOT; branch = 0; jp = 0; end
                `INST_FUNC_HLT : begin alu_op = `FUNC_ADD; alu_src_A = `ALUSrcA_A; alu_src_B = `ALUSrcB_B;  mem_read = 0; mem_write = 0; mem_to_reg = 0;
                                 reg_write = 0; pc_src = `PC_DEF; hlt = 1; wwd = 0; branch_type = `BRANCH_NOT; branch = 0; jp = 0; end

                default        : begin alu_op = funccode[2:0]; alu_src_A = `ALUSrcA_A; alu_src_B = `ALUSrcB_B; mem_read = 0; mem_write = 0; mem_to_reg = 0;
                 reg_dst = `RegDst_RD; reg_write = 1; pc_src = `PC_DEF; hlt = 0; wwd = 0; branch_type = `BRANCH_NOT; jp = 0; end
            endcase
        end
        default : begin alu_op = `FUNC_ADD; alu_src_A = `ALUSrcA_0; alu_src_B = `ALUSrcB_0; mem_read = 0; mem_write = 0; mem_to_reg = 0; reg_dst = 0; reg_write = 0;
        pc_src = `PC_DEF; hlt = 0; wwd = 0; branch_type = `BRANCH_NOT; branch = 0; jp = 0; end
    endcase
end
```

이전 과제인 single cycle cpu의 control unit과 거의 유사하다.

control unit에서 설정된 signal들은 cpu의 pipeline register에 의해 다음 stage로 전달된다.

4. CPU

a. IF

```
if (!is_stall) begin
    PC <= PC_NXT;
    IF_PC <= PC + 1;
    if (!IF_flush) begin
        IF_inst <= data1;
        IF_is_bubble <= 0;
    end
    else begin
        IF_inst <= 0;
        IF_is_bubble <= 1;
    end
end else begin
    IF_is_bubble <= 1;
end
```

```
PC_NXT = PCMuxOut;
```

```
assign address1 = read_m1 ? PC : address1;
assign read_m1 = 1;
```

```
mux4_1 #(`WORD_SIZE) PCMux(pc_src, PC + `WORD_SIZE'd1, IF_PC + immediate, {IF_PC[15:12], target}, ForwardReadOut1MuxOut, PCMuxOut);
```

위의 가장 왼쪽의 사진은 posedge clk마다 실행되는 코드이다. address1에 PC 값을 넣어 메모리에서 명령어를 읽어온다. 읽어온 instruction은 IF_inst에 저장하여 ID 단계로 넘긴다. stall이 일어난 상황에서는 새로운 명령어를 읽지 않으며 IF flush 상황에서는 IF_inst를 0으로 초기화시켜 ID 과정이 일어나는 것을 막고 bubble을 넣는다. PC값은 pc_src signal에 의해 선택된다.

b. ID

```
register_file RegisterFile(read_out1, read_out2, read1, read2, MEM_dest, WriteDataMuxOut, MEM_reg_write, clk, reset_n);
```

```
// ID
opcode = IF_inst[15:12];
rs = IF_inst[11:10];
rt = IF_inst[9:8];
rd = IF_inst[7:6];
funccode = IF_inst[5:0];
target = IF_inst[11:0];

read1 = rs; read2 = rt;

if (branch) begin
    case (branch_type)
        `BRANCH_NE : bcond = ForwardReadOut1MuxOut != ForwardReadOut2MuxOut ? 1 : 0;
        `BRANCH_EQ : bcond = ForwardReadOut1MuxOut == ForwardReadOut2MuxOut ? 1 : 0;
        `BRANCH_GZ : bcond = ForwardReadOut1MuxOut > 0 ? 1 : 0;
        `BRANCH_LZ : bcond = ForwardReadOut1MuxOut < 0 ? 1 : 0;
    endcase
end
```

ID과정에서는 IF_inst에 저장된 instruction을 decode하여 저장한다. 이 값을 이용하여 register file에서 해당 number에 대한 value를 읽어온다. bcond 역시 ID에서 계산한다.

```
forwarding_unit ForwardingUnit(rs, rt, ID_read1, ID_read2, ID_dest, EX_dest, MEM_dest, ID_reg_write, EX_reg_write, MEM_reg_write, ID_is_bubble, EX_is_bubble, MEM_is_bubble,
                               forward_A, forward_B, forward_read_out1, forward_read_out2);
```

```
mux4_1 #(`WORD_SIZE) ForwardReadOut1Mux(forward_read_out1, read_out1, alu_result, read_m2? data2 :EX_alu_out, WriteDataMuxOut, ForwardReadOut1MuxOut);
mux4_1 #(`WORD_SIZE) ForwardReadOut2Mux(forward_read_out2, read_out2, alu_result, read_m2? data2 : EX_alu_out, WriteDataMuxOut, ForwardReadOut2MuxOut);
```

forwarding unit에 의해 설정된 signal을 이용하여 forwarding을 수행한다. register에서 읽은 값, alu 계산값, EX register의 alu 계산값, WB을 위한 값 중 하나를 가져와서 이용한다.

```
control_unit ControlUnit(opcode, funccode, clk, reset_n, bcond, alu_op, alu_src_A, alu_src_B,
                         mem_read, mem_write, mem_to_reg, reg_dst, reg_write,
                         pc_src, hlt, wwd, branch_type, branch, jp);
```

decode 결과를 이용하여 control value를 설정한다.

```
// ID
ID_PC <= IF_PC;
ID_readData1 <= read_out1;
ID_readData2 <= read_out2;
ID_dest <= WriteDestMuxOut;
ID_read1 <= rs;
ID_read2 <= rt;
ID_immediate <= immediate;

// ID/EX Control
ID_alu_op <= alu_op;
ID_alu_src_A <= alu_src_A;
ID_alu_src_B <= alu_src_B;
ID_branch_type <= branch_type;
ID_reg_write <= reg_write;
ID_wwd <= wwd;
ID_hlt <= hlt;
```

```
if(!ID_flush) begin
    ID_mem_read <= mem_read;
    ID_mem_write <= mem_write;
    ID_mem_to_reg <= mem_to_reg;
end else begin
    ID_mem_read <= 0;
    ID_mem_write <= 0;
    ID_mem_to_reg <= 0;
end
ID_is_bubble <= IF_is_bubble;
```

ID/EX register에 필요한 값(control value, data)을 저장한다. ID_flush인 경우 pipeline register를 reset하여 bubble을 삽입한다.

c. EXE

```
mux4_1 #(`WORD_SIZE) ForwardAMux(forward_A, `WORD_SIZE'd0, ID_readData1, EX_alu_out, WriteDataMuxOut, ForwardAMuxOut);
mux4_1 #(`WORD_SIZE) ForwardBMux(forward_B, `WORD_SIZE'd0, ID_readData2, EX_alu_out, WriteDataMuxOut, ForwardBMuxOut);
mux2_1 #(`WORD_SIZE) ALUSrcAMux(ID_alu_src_A, ForwardAMuxOut, `WORD_SIZE'd0, ALUSrcAMuxOut);
mux4_1 #(`WORD_SIZE) ALUSrcBMux(ID_alu_src_B, ForwardBMuxOut, ID_immediate, ID_PC, `WORD_SIZE'd0, ALUSrcBMuxOut);
```

```
alu ALU(ALUSrcAMuxOut, ALUSrcBMuxOut, ID_alu_op, alu_result, overflow_flag);
```

forwarding unit에 의해 설정된 signal에 따라 data를 forwarding해온다. ID로부터는 포워딩 하지 않는다. 각 stage에서 해당하는 값을 가져온다. EX(register에서 읽은 값-ID_readData), MEM(계산값-EX_aluout), WB(write할 값-WriteDataMuxOut)
forwarding 결과를 alu unit에 넣어주어 계산을 진행한다.

```
// EX
EX_B <= ForwardBMuxOut;
EX_alu_out <= alu_result;
EX_dest <= ID_dest;
EX_mem_read <= ID_mem_read;
EX_mem_write <= ID_mem_write;
EX_mem_to_reg <= ID_mem_to_reg;
EX_reg_write <= ID_reg_write;
EX_wwd <= ID_wwd;
EX_hlt <= ID_hlt;
EX_is_bubble <= ID_is_bubble;
```

pipeline register에 필요한 값을 저장하여 전달한다.

d. MEM

```
assign read_m2 = EX_mem_read;
assign write_m2 = EX_mem_write;
assign address2 = (read_m2 | write_m2) ? EX_alu_out : address2;
assign data2 = write_m2 ? EX_B : `WORD_SIZE'bz;
```

EXE에서부터 전달받은 값을 이용하여 각 value들을 설정한다.

```
//MEM
MEM_data <= data2;
MEM_alu_out <= EX_alu_out;
MEM_dest <= EX_dest;

MEM_mem_to_reg <= EX_mem_to_reg;
MEM_reg_write <= EX_reg_write;
MEM_wwd <= EX_wwd;
MEM_hlt <= EX_hlt;
MEM_is_bubble <= EX_is_bubble;
```

pipeline register에 필요한 값을 저장하여 전달한다.

e. WB

```
mux4_1 #(2) WriteDestMux(reg_dst, rd, rt, 2'd2, 2'd0, WriteDestMuxOut);
mux2_1 #(`WORD_SIZE) WriteDataMux(MEM_mem_to_reg, MEM_alu_out, MEM_data, WriteDataMuxOut);
```

control unit에서 설정한 mem_to_reg 값을 이용하여 alu 결과값과 MEM에서 읽어온 data 값 중 어느 값을 쓸 지 결정한다.

```
register_file RegisterFile(read_out1, read_out2, read1, read2, MEM_dest, WriteDataMuxOut, MEM_reg_write, clk, reset_n);
```

writedatamux의 결과값과 ID단계에서 설정된 write를 진행할 register 번호를 register file의 input으로 넣어서 WB을 수행한다.

```
num_clock <= num_clock + 1;
num_inst <= (num_clock >= 5 && !MEM_is_bubble) ? num_inst + 1 : num_inst;
```

instruction의 수행이 끝났으므로 조건에 맞게 num_inst를 증가시킨다.

Discussion

test를 위한 num_inst를 설정하는 과정에 어려움이 있었다. 처음에는 instruction이 fetch될 때마다 값을 증가시켰다. 하지만 중간에 flush되는 경우는 해당 instruction이 실행되지 않은 것으로 처리해야 하므로 is_bubble 변수를 도입했다. WB 과정에서 실행 중인 instruction이 bubble이 아닌 경우에만 집계하는 방식으로 변경했다. 또 첫 4 cycle은 쓰레기값이 저장되어 있기 때문에 num_clock 역시 고려했다.

처음에는 forward unit에서 alu input 2개에 대한 forwarding signal만 만들어냈다. 하지만 bcond 계산에서도 forwarding이 필요함을 알게 되었다. 우리는 branch prediction이 실패했을 때를 소실되는 clk 수를 최소화 하기 위해서 ID 과정에서 bcond 계산을 수행하도록 cpu를 설계했다. 따라서 ID에서도 forwarding이 필요함을 알게 되었다. forwarding unit에서 forward_read_out1과 2를 함께 설정하여 bcond 계산의 data hazard를 없앴다.

Conclusion

pipelined CPU의 동작 방식을 이해하고 구현하였다. pipeline에 의해 발생하는 문제인 data hazard와 control hazard를 이해하고 이에 대한 해결책인 forwarding과 branch prediction을 구현했다. branch prediction의 경우 always not taken으로 구현했다.

아래 그림 중 위의 그림은 Lab5에서 TestBench에 주어진 경우를 모두 통과한 모습이다. 아래 그림은 Lab4의 결과를 출력한 모습이다.

```
VSIM 36> run -all
# Clock # 1213
# The testbench is finished. Summarizing...
# All Pass!
# ** Note: $finish    : C:/intelFPGA_pro/20.4/lab5/cpu_TB.v(153)
#    Time: 121450 ns  Iteration: 2  Instance: /cpu_TB
```

```
VSIM 2> run -all
# Clock # 4510
# The testbench is finished. Summarizing...
# All Pass!
# ** Note: $finish    : C:/intelFPGA_pro/20.4/lab4/cpu_TB.v(151)
#    Time: 451150 ns  Iteration: 2  Instance: /cpu_TB
```

4510번의 clk이 필요했던 multi cycle cpu에 비해 1213번으로 눈에 띄게 clk 수가 줄어든 것을 확인할 수 있다. 우리는 pipelined cpu가 multi-cycle cpu에 비해 3.71배 빠르다는 결과를 얻었다.