CSED311 Lab04

-Pipelined CPU, non-control flow-

Team ID: 2

20180740 염재후

1. Introduction

이번 Lab의 목표는 pipeline CPU를 구현하는 것이다. Pipelined CPU는 latency를 개선하지는 못하지만, CPU에 있는 unit들을 최대한 이용하여, Throughput을 증가시키는 CPU이다. unit들을 최대한 이용하기 위해 instruction을 다섯 단계, Instruction Fetch, Instruction Decode, Execution, Memory Access, Write Back으로 나누어서 처리한다. Pipeline CPU에는 필연적으로, Data Hazard와, Control Hazard가 나타나는데, 이번 Lab에서는 Data Hazard만을 고려한다.

Data Hazard는 Read After Write 상황에서 일어나게 되는데, Instruction이 동시에 처리되기 때문에, register value가 갱신되기 전에, outdated value가 사용되는 것을 막기 위해, Stall 혹은 Data Forwarding을 하게 된다. Data Forwarding의 경우, EX/MEM, MEM/WB단계에서 계산된 값을 다시 ID or EX Stage에 넣어줌으로써, register value는 갱신되지 않았지만, 갱신된 값을 사용할 수 있다.

Data Forwarding을 하게 되면, 대부분의 Data Hazard는 해결된다. 하지만, Load instruction의 경우, 값이 MEM Stage 이후에 접근할 수 있기 때문에, Load instruction 이후에, register를 사용하는 R type, I type 등의 instruction이 나오게 되면, 1 cycle동안 Stall을 해야 한다.

2. Design

Introduction에서 소개했듯, pipelined cpu는 IF, ID, EX, MEM, WB 총 다섯 단계를 거친다. Instruction Fetch, IF Stage에서는 PC address를 가지고, instruction을 가지고 오는 단계이다. ID, Instruction Decode Stage에서는 IF Stage에서 가지고 온 instruction의 각 field에서 register value와 control signal을 생성한다. EX, Execution Stage에서는 ID에서 생성한 control signal에 따라서, 알맞은 연산을 ALU에서 진행한다. MEM, Memory Access Stage에서는 Load instruction일 경우, EX Stage에서 계산한 주소를 기반으로 데

이터를 읽고, Store Instruction인 경우, 데이터를 저장한다. WB, Write Back Stage에서는 Data Memory에서 가지고 온 Data, 혹은 ALU에서 계산한 값을 register에 쓰는 단계이다. Pipelined cpu는 각 단계를 나눴기 때문에, 각 단계에 있는 instruction에 대한 정보를 저장하기 위해 register가 필요하고, 이 register들은 각 clock cycle마다 갱신된다.

즉, pipelined cpu는 PC, memory, register file과 같이 synchronous하게 write해야 하는 module을 제외하면, 나머지는 asynchronous한 module이다. asynchronous하게 output을 생산하면, synchronous하게 그 output을 다음 단계에 쓰기 위한 register로 써준다.

여기에 더해, 이번에 설계한 pipelined CPU는 data forwarding기능이 있기 때문에, Forwarding unit과, forwarding 상태에서의 hazard detection unit이 존재한다. 이에 대한 자세한 디자인은 forwarding unit과 hazard detection unit에서 다루도록 한다.

a. Forwarding Unit

Forwarding은 크게 EX/MEM to EX, MEM/WB to EX, MEM/WB to ID, instruction distance에 따라 세 종류로 나눌 수 있다. Forwarding Unit은 EX/MEM to EX, MEM/WB to EX, MEM/WB to ID, 이 두 가지를 처리해주는 module이다.

Forwarding이 필요한 경우는, 각 MEM, WB 단계에서 처리하는 데이터가 씌여질 rd가 EX에서 사용하는 경우이다. Forwarding이 없다면, 갱신될 때까지 Stall되어야 하기 때문이다. 따라서 forwarding이 필요한 조건은 다음과 같다.

1. Rs1이 0이 아니여야 한다.
2. Rs1이 MEM 혹은 WB의 rd와 동일해야 한다.
3. MEM 혹은 WB 단계에서 register write signal이 1이어야 한다.

여기에 더해, MEM와 WB이 동시에 데이터를 처리하고 있다면, WB보다 MEM에 존재하는 데이터가 가장 최신 데이터이기 때문에, WB가 아닌 MEM의 데이터를 가져와야 한다.

다음과 같은 조건들을 rs2에도 동일하게 적용하여 forwarding을 결정하면 된다.

b. Internal Forwarding Unit

Internal Forwarding의 경우, MEM/WB register에서 나오지만, EX Stage가 아닌 ID Stage로 가기 때문에, Instruction Distance가 3만큼 멀다. 이 경우에도, 위와 비슷한 조건을 고려하여 forwarding을 진행하게 된다. 차이가 있다면, EX/MEM,

MEM/WB, 두 종류를 고려하는 것이 아니라 MEM/WB 하나만 고려하면 되는 것이다. 따라서 이를 고려하여 forwarding을 결정하면 된다.

추가적으로, Ecall instruction with Data forwarding이 무조건 한 cycle을 stall하는 조건으로 바뀌게 되면서, addi x17, x0, 10, Bubble, Ecall의 경우가 새로 생기게 되었다. 이 경우는, EX/MEM의 data가 ID Stage로 옮겨와야 하기 때문에, ID/EX에 Ecall signal이 있는지, 또한 EX/MEM의 rd가 17인지를 확인한 후에 forwarding해주어야 한다.

c. Hazard Detection Unit

Forwarding이 가능한 상황에서, Hazard가 일어나는 경우는 Load Instruction 바로 다음에 Load instruction에서 가져올 데이터를 사용하는 instruction이 오는 경우이다. 그 이유는, EX Stage가 끝나면, 사용 가능해지는 forwarding에 비해 Load instruction은 MEM stage가 끝나야 데이터를 사용할 수 있기 때문에 forwarding을 하더라도, stall을 피할 수 없다. 따라서 명시적인 조건은 다음과 같다.

1. ID Stage의 rs1(or rs2)과 EX의 rd가 동일하다.
2. Rs1(or rs2)이 0이 아니다.
3. ID Stage의 Instruction이 rs1(or rs2)를 사용한다.
4. EX Stage의 Memory Read Signal이 1이다.

Rs1을 사용하는 instruction의 경우는, JAL, ECALL을 제외한 나머지 instruction이 rs1을 사용하며, rs2를 사용하는 경우는 R-type, Store, Branch instruction의 경우에 rs2를 사용하게 된다.

따라서 ID stage의 instruction과 EX stage의 control signal(memory read)를 고려하여, stall condition을 결정해준다.

이 때, stall을 하게 된다면, pc가 갱신되지 않고, IF/ID의 register들이 갱신되지 않으며, control signal들이 0이 되어서 PVS에 어떤 변화도 일으키지 않게 된다.

추가적으로 Ecalldl 무조건 한 cycle을 stall해야 하는 조건이 추가되면서, Ecall과 관련된 조건들이 추가적으로 생겼다. 이에 대해 Ecall Hazard detect될 경우, Stall을 해준다.

d. Ecall Hazard Detection Unit

Forwarding이 있는 경우, 다른 instruction으로 x17 register를 바꾸는 경우, Ecall instruction은 무조건 한 cycle을 쉬어야 하며, load instruction으로 x17 register value를 바꾸는 경우, 두 cycle을 쉬어야 하는 조건이 새로 생기게 되어서 만들어진 모듈이다.

따라서, Ecall signal이 들어오며, 동시에 Ecall instruction 앞의 instruction의 rd가 17이며, register의 값을 변경하는 signal이 asserted된 경우, Stall condition을 asserted해준다. 하지만, Stall되면서 Ecall instruction이 계속 ID에 머물 경우, ecall signal을 계속해서 보내주기 때문에, 이를 방지하기 위해서, ID/EX로 ecall signal이 갈 경우에는 다시 deasserted해주지만, ecall instruction 앞에 있던 명령어가 load instruction인 경우에는 그대로 asserted를 유지해준다.

e. ALU

ALU는 single cycle, multi cycle과 동일하게, given input에 대해 계산한 결과를 내보내는 combinational logic unit이다.

f. ALU Control Unit

ALU Control Unit은 ID/EX에 저장된 control signal output을 기반으로 ALU에서 어떤 연산을 수행할지 combinational logic에 따라 결정해주는 alu control input signal을 만드는 unit이다.

g. Control Unit

Control Unit은 IF/ID에 저장된 instruction을 기반으로, 해당 instruction을 수행하기 위해 필요한 control signal들을 combinational logic을 통해 만들어주는 unit이다.

h. PC

PC는 synchronous하게 PC address value를 갱신해주는 모듈이다. 각 clock cycle마다 next PC value로 update를 해주는데, 만약 hazard detect unit에서 stall을 위해 control signal을 asserted한다면, PC 갱신을 멈춘다.

i. Register File

Register file은 register를 synchronous하게 write하고 asynchronous하게 read하

는 모듈이다. Register를 write할 때, register number가 0일 경우, write를 무시한다.

j. Memory

Memory는 이 lab에서 instruction memory와 data memory로 나뉘는데, instruction memory는 rom이고, data memory는 writable하다. Instruction memory는 asynchronous하게 instruction을 읽어오며, data memory는 asynchronous하게 read하며 동시에 synchronous하게 data를 write한다.

3. Implementation

a. Forwarding Unit

```
module Forwarding_unit(
    input EX_MEM_reg_write,
    input MEM_WB_reg_write,
    input [`regnum-1: 0] EX_MEM_rd,
    input [`regnum-1: 0] MEM_WB_rd,
    input [`regnum-1: 0] rs1,
    input [`regnum-1: 0] rs2,
    output reg [1:0] ForwardA,
    output reg [1:0] ForwardB
);

always@(*)begin
    if(rs1 != 0 && rs1 == EX_MEM_rd && EX_MEM_reg_write) begin
        ForwardA = 2'b10;
    end
    else if(rs1 != 0 && rs1 == MEM_WB_rd && MEM_WB_reg_write) begin
        ForwardA = 2'b01;
    end
    else
        ForwardA = 2'b00;

    if(rs2 != 0 && rs2 == EX_MEM_rd && EX_MEM_reg_write) begin
        ForwardB = 2'b10;
    end
    else if(rs2 != 0 && rs2 == MEM_WB_rd && MEM_WB_reg_write) begin
        ForwardB = 2'b01;
    end
    else
        ForwardB = 2'b00;
end
endmodule
```

Forwarding unit은 design part에서 서술했던 logic을 동일하게 비교하여서, 각각 EX/MEM, MEM/WB, input from ID/EX를 고르게 해주는 signal을 생성해준다.

b. Internal Forwarding Unit

```
module Internal_Forwarding_unit(
    input [`regnum -1: 0] WB_rd,
    input [`regnum -1: 0] rs1_in,
    input [`regnum -1: 0] rs2_in,
    input WB_reg_write,
    output reg internal_forwardA,
    output reg internal_forwardB);

    always@(*)begin
        if(rs1_in != 0 && rs1_in == WB_rd && WB_reg_write == 1) begin
            internal_forwardA = 1;
        end
        else
            internal_forwardA = 0;

        if(rs2_in != 0 && rs2_in == WB_rd && WB_reg_write == 1) begin
            internal_forwardB = 1;
        end
        else
            internal_forwardB = 0;
    end
endmodule
```

```
assign rs1_dout_fwd = (internal_forwardA == 1) ? rd_din : (is_ecall && EX_MEM_rd == 17 && EX_MEM_reg_write) ? EX_MEM_alu_out : rs1_dout;
assign rs2_dout_fwd = (internal_forwardB == 1) ? rd_din : rs2_dout;
```

Internal forwarding unit은 input from RF, MEM/WB를 고려하여서, 어떤 값을 input으로 사용하는지 선택하는 signal을 생성해준다. 이 때 signal을 생성하는 logic은 Design에 서술된 바와 동일하다.

Ecall로 인한 EX/MEM to ID forwarding의 경우, is ecall condition과 EX_MEM_rd의 상태를 고려하여, MUX를 통해 골라주는 방식으로 구현하였다.

c. Hazard detect Unit

```
module Hazard_detection_unit(
    input [6:0] opcode,
    input ecall_hazard_detect,
    input ID_EX_mem_read,
    input [`regnum -1: 0]ID_EX_rd,
    input [`regnum -1: 0]ID_rs1,
    input [`regnum -1: 0]ID_rs2,
    output reg IF_ID_Write,
    output reg PCWrite,
    output reg hazard_detect);

    wire use_rs1, use_rs2;

    assign use_rs1 = !(opcode == `JAL || opcode == `ECALL);
    assign use_rs2 = (opcode == `ARITHMETIC || opcode == `STORE || opcode == `BRANCH);


    // IF_ID_Write = 1, IF/ID Reg can be written.
    // PCWrite = 1, PC can be written.
    // hazard_detect = 1, hazard detected -> control unit generates stall.
    always@(*) begin
        //Stall Condition
        if(((ID_rs1 == ID_EX_rd && ID_rs1 != 0 && use_rs1) ||
            (ID_rs2 == ID_EX_rd && ID_rs2 != 0 && use_rs2)) &&
            ID_EX_mem_read == 1) begin
                IF_ID_Write = 0;
                PCWrite = 0;
                hazard_detect = 1;
            end

        else if(ecall_hazard_detect == 1) begin
            IF_ID_Write = 0;
            PCWrite = 0;
            hazard_detect = 1;
        end

        // keep going
        else begin
            IF_ID_Write = 1;
            PCWrite =1;
            hazard_detect =0;
        end
    end
endmodule
```

Hazard detection unit은 load instruction 뒤에, load instruction의 rd register를 사용하는 경우, MEM Stage 이전 Stage들을 stall 시켜야 한다. 따라서, IF의 경우 PC 갱신을 중단하고, IF/ID register역시 바뀌지 않게 한다. 또한, EX stage에서는 control signal을 0로 만들어, PVS에 어떤 변화도 주지 않게 한다.

혹은 Ecall로 인한 Stall이 필요할 경우, Stall condition을 asserted 해준다.

d. Ecall Hazard Detection Unit

```
module Ecall_Hazard_detection_unit(
    input is_ecall,
    input [`regnum-1:0] ID_EX_rd,
    input ID_EX_reg_write,
    input ID_EX_is_ecall,
    input EX_MEM_mem_read,
    input [`regnum-1:0]EX_MEM_rd,
    output reg ecall_hazard_detect);

    always@(*) begin
        if(is_ecall && ID_EX_rd == 17 && ID_EX_reg_write == 1)
            ecall_hazard_detect = 1;
        if(ID_EX_is_ecall)
            ecall_hazard_detect = 0;
        if(is_ecall && EX_MEM_mem_read && EX_MEM_rd == 17 && ID_EX_is_ecall)
            ecall_hazard_detect = 1;
    end

endmodule
```

design에서 설명하였던, 경우를 토대로 hazard detection unit에 보낼 stall condition을 생성해준다. is ecall signal이 asserted된 경우 Ecall instruction 앞의 register의 rd번호가 17인지 확인하고, 그 값을 바꾸는지 확인해준다. 이를 확인해 output signal을 asserted해주되, 이미 한 번 stall 된 상태라면, deasserted로 바꿔주며, load인 경우, 다시 signal을 asserted해주는 방식으로 output signal을 관리한다.

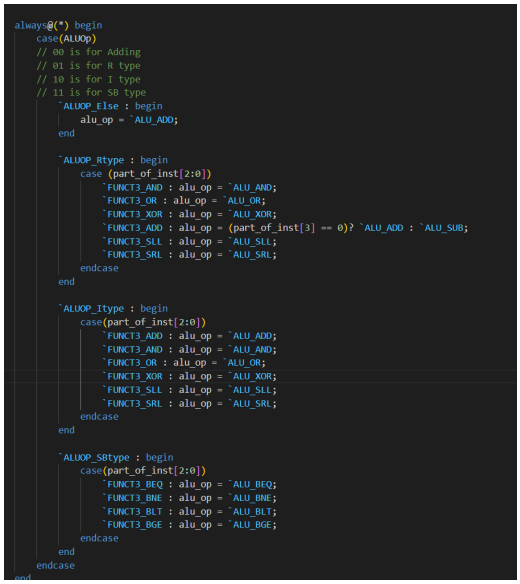
e. ALU

```
always@(*) begin
    case (alu_op)
        `ALU_AND : alu_result = (alu_in_1 & alu_in_2);
        `ALU_OR : alu_result = (alu_in_1 | alu_in_2);
        `ALU_ADD : alu_result = (alu_in_1 + alu_in_2);
        `ALU_SUB : alu_result = (alu_in_1 - alu_in_2);
        `ALU_SLL : alu_result = (alu_in_1 << alu_in_2[4:0]);
        `ALU_SRL : alu_result = (alu_in_1 >> alu_in_2[4:0]);
        `ALU_XOR : alu_result = (alu_in_1 ^ alu_in_2);
        `ALU_BEQ : begin
            alu_result = alu_in_1 - alu_in_2;
            if(alu_result == 0 )
                alu_zero = 1;
            else
                alu_zero = 0;
            end
        `ALU_BNE : begin
            alu_result = alu_in_1 - alu_in_2;
            if(alu_result != 0)
                alu_zero = 1;
            else
                alu_zero = 0;
            end

        `ALU_BLT : begin
            alu_result = alu_in_1 - alu_in_2;
            if($signed(alu_result) < 0)
                alu_zero = 1;
            else
                alu_zero = 0;
            end
        `ALU_BGE : begin
            alu_result = alu_in_1 - alu_in_2;
            if($signed(alu_result) >= 0)
                alu_zero = 1;
            else
                alu_zero = 0;
            end
    endcase
```

ALU의 경우, 주어진 alu control input에 맞게 input 값을 asynchronous하게 처

리하여 output을 내보내는 module이다.

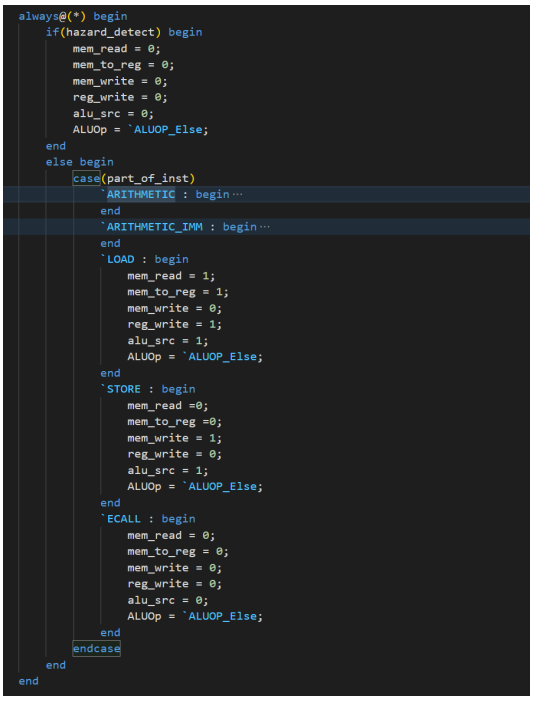
f. ALU Control Unit



```
always@(*) begin
    case(ALUOp)
    // 00 is for Adding
    // 01 is for R type
    // 10 is for I type
    // 11 is for SB type
        `ALUOP_Else : begin
            alu_op = `ALU_ADD;
        end

        `ALUOP_Rtype : begin
            case (part_of_inst[2:0])
                `FUNCT3_AND : alu_op = `ALU_AND;
                `FUNCT3_OR : alu_op = `ALU_OR;
                `FUNCT3_XOR : alu_op = `ALU_XOR;
                `FUNCT3_ADD : alu_op = (part_of_inst[3] == 0)? `ALU_ADD : `ALU_SUB;
                `FUNCT3_SLL : alu_op = `ALU_SLL;
                `FUNCT3_SRL : alu_op = `ALU_SRL;
            endcase
        end

        `ALUOP_Itype : begin
            case(part_of_inst[2:0])
                `FUNCT3_ADD : alu_op = `ALU_ADD;
                `FUNCT3_AND : alu_op = `ALU_AND;
                `FUNCT3_OR : alu_op = `ALU_OR;
                `FUNCT3_XOR : alu_op = `ALU_XOR;
                `FUNCT3_SLL : alu_op = `ALU_SLL;
                `FUNCT3_SRL : alu_op = `ALU_SRL;
            endcase
        end

        `ALUOP_SBtype : begin
            case(part_of_inst[2:0])
                `FUNCT3_BEQ : alu_op = `ALU_BEQ;
                `FUNCT3_BNE : alu_op = `ALU_BNE;
                `FUNCT3_BLT : alu_op = `ALU_BLT;
                `FUNCT3_BGE : alu_op = `ALU_BGE;
            endcase
        end
    endcase
end
```

ALU control input은 ALUOp from control unit과 instruction에서 추출한 funct code를 기반으로, ALU가 어떤 연산을 취해야 하는지 결정하는 signal을 만들어 주는 unit이다.

g. Control Unit



```
always@(*) begin
    if(hazard_detect) begin
        mem_read = 0;
        mem_to_reg = 0;
        mem_write = 0;
        reg_write = 0;
        alu_src = 0;
        ALUOp = `ALUOP_Else;
    end
    else begin
        case(part_of_inst)
            `ARITHMETIC : begin···
            end
            `ARITHMETIC_IMM : begin···
            end
            `LOAD : begin
                mem_read = 1;
                mem_to_reg = 1;
                mem_write = 0;
                reg_write = 1;
                alu_src = 1;
                ALUOp = `ALUOP_Else;
            end
            `STORE : begin
                mem_read =0;
                mem_to_reg =0;
                mem_write = 1;
                reg_write = 0;
                alu_src = 1;
                ALUOp = `ALUOP_Else;
            end
            `ECALL : begin
                mem_read = 0;
                mem_to_reg = 0;
                mem_write = 0;
                reg_write = 0;
                alu_src = 0;
                ALUOp = `ALUOP_Else;
            end
        endcase
    end
end
```

Control unit의 경우, 주어진 opcode에 맞게, 각 instruction을 처리하기 위한

control signal들을 asynchronous하게 만들어주는 unit이다. 따라서 각 instruction 마다 case로 나누어 control signal을 만들어준다.

h. PC

```
module PC(
    reset,
    clk,
    next_pc,
    nextPCWrite,
    current_pc
    );

    input reset;
    input clk;
    input nextPCWrite;
    input [`WordSize -1 : 0] next_pc;
    output reg [`WordSize -1 : 0] current_pc;

    always@(posedge clk)begin
        if(reset)
            current_pc <= 0 ;
        else if(nextPCWrite)
            current_pc <= next_pc;
        else
            current_pc <= current_pc;
    end
endmodule
```

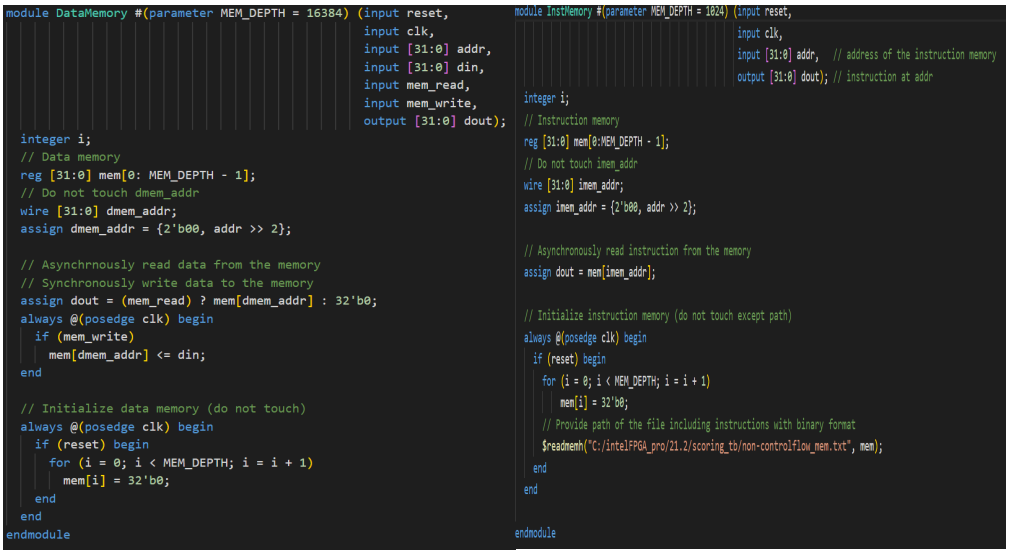
PC는 synchronous하게 PC address value를 바꿔주는 module이다. 이 때, hazard로 인한 stall이 있을 수 있기 때문에, control unit으로부터 오는 signal을 고려하여 PC address value를 갱신해주도록 한다.

i. Register File

```
module RegisterFile(input reset,
                    input clk,
                    input [4:0] rs1,          // source register 1
                    input [4:0] rs2,          // source register 2
                    input [4:0] rd,           // destination register
                    input [31:0] rd_din,      // input data for rd
                    input write_enable,          // RegWrite signal
                    output [31:0] rs1_dout,   // output of rs 1
                    output [31:0] rs2_dout);  // output of rs 2
  integer i;
  // Register file
  reg [31:0] rf[0:31];

  // Asynchronously read register file
  // Synchronously write data to the register file
  assign rs1_dout = rf[rs1];
  assign rs2_dout = rf[rs2];
  always @(posedge clk) begin
    if (write_enable & (rd != 0))
      rf[rd] <= rd_din;
  end

  // Initialize register file (do not touch)
  always @(posedge clk) begin
    // Reset register file
    if (reset) begin
      for (i = 0; i < 32; i = i + 1)
        rf[i] = 32'b0;
      rf[2] = 32'h2ffc; // stack pointer
    end
  end
endmodule
```

Register file은 given file이나, 분석을 하자면, synchronous하게 write해주고, asynchronous하게 read할 수 있는 module이다. Memory와 유사하지만, 이 때 register number가 0일 경우 unwritable하다는 것을 고려해주면 된다.

j. Memory



```
module DataMemory #(parameter MEM_DEPTH = 16384) (input reset,
                                                  input clk,
                                                  input [31:0] addr,
                                                  input [31:0] din,
                                                  input mem_read,
                                                  input mem_write,
                                                  output [31:0] dout);
  integer i;
  // Data memory
  reg [31:0] mem[0: MEM_DEPTH - 1];
  // Do not touch dmem_addr
  wire [31:0] dmem_addr;
  assign dmem_addr = {2'b00, addr >> 2};

  // Asynchrnously read data from the memory
  // Synchronously write data to the memory
  assign dout = (mem_read) ? mem[dmem_addr] : 32'b0;
  always @(posedge clk) begin
    if (mem_write)
      mem[dmem_addr] <= din;
  end

  // Initialize data memory (do not touch)
  always @(posedge clk) begin
    if (reset) begin
      for (i = 0; i < MEM_DEPTH; i = i + 1)
        mem[i] = 32'b0;
    end
  end
endmodule
```

```
module InstMemory #(parameter MEM_DEPTH = 1024) (input reset,
                                                 input clk,
                                                 input [31:0] addr,  // address of the instruction memory
                                                 output [31:0] dout); // instruction at addr
  integer i;
  // Instruction memory
  reg [31:0] mem[0:MEM_DEPTH - 1];
  // Do not touch imem_addr
  wire [31:0] imem_addr;
  assign imem_addr = {2'b00, addr >> 2};

  // Asynchronously read instruction from the memory
  assign dout = mem[imem_addr];

  // Initialize instruction memory (do not touch except path)
  always @(posedge clk) begin
    if (reset) begin
      for (i = 0; i < MEM_DEPTH; i = i + 1)
          mem[i] = 32'b0;
      // Provide path of the file including instructions with binary format
      $readmemh("C:/intelFPGA_pro/21.2/scoring_tb/non-controlflow_mem.txt", mem);
    end
  end

endmodule
```

이번 랩에서는 instruction memory와 data memory가 나뉘어져 있는데, instruction memory는 이번 lab에서는 read only memory이며, data memory는 writable한 memory이다. Write는 synchronous하며, read는 asynchronous하게 수행할 수 있도록 한다.

4. Discussion

Pipelined cpu의 total cycle은 59cycle

Single cycle cpu는 ripes 기준 51cycle이다.

이론적으로 single cycle cpu와 pipelined cpu 는 4 cycle 차이가 나므로, 55 cycle을 예측할 수 있다. 하지만, 구현한 pipelined cpu의 경우, 59cycle로 4cycle 차이가 난다. 이 cycle 차이는 먼저, load instruction의 data hazard로 인한 stall로 인한 1 cycle 차이가 발생한다. 그리고 ecall instruction이 3번 나타나는데, 각 ecall instruction의 previous instruction이 모두 rd를 17로 가지며, reg write signal이 asserted된 상태이기 때문에, 3 cycle만큼 더 stall하게 된다. 따라서 총 4cycle을 stall하기 때문에 이러한 차이가 발생한다고 생각한다.

Clock Cycle 수 자체는 single cycle과 크게 차이가 나지 않기 때문에 clock frequency를 충분히 활용하지 못하지 않을까? 라는 의문이 생길 수 있지만, pipelined cpu의 clock cycle은 각 IF, ID, EX, MEM, WB를 실행하기 위한 clock period이기 때문에 single cycle보다는 더 작다. Multi cycle과 비교한다면, multi cycle의 clock period보다는 더 길지만, pipelined cpu는 instruction을 겹쳐서 해결하기 때문에 throughput적

인 측면으로는 pipelined cpu가 더 월등하다고 할 수 있다.

5. Conclusion

Pipelined CPU의 동작 방식을 이해하고 구현할 수 있었다. Pipelined CPU에서 발생하는 Hazard에는 Data Hazard와 Control Hazard가 있는데, Control Hazard의 경우는 이번 랩의 주안점이 아니기 때문에, always PC+4로 가정하고 해결하였고, Data Hazard의 경우 Data Forwarding을 통해서 해결할 수 있었다.