Computer Architecture (CSED 311), Spring 2025
Lab 4: Pipelined CPU w/o control flow instructions

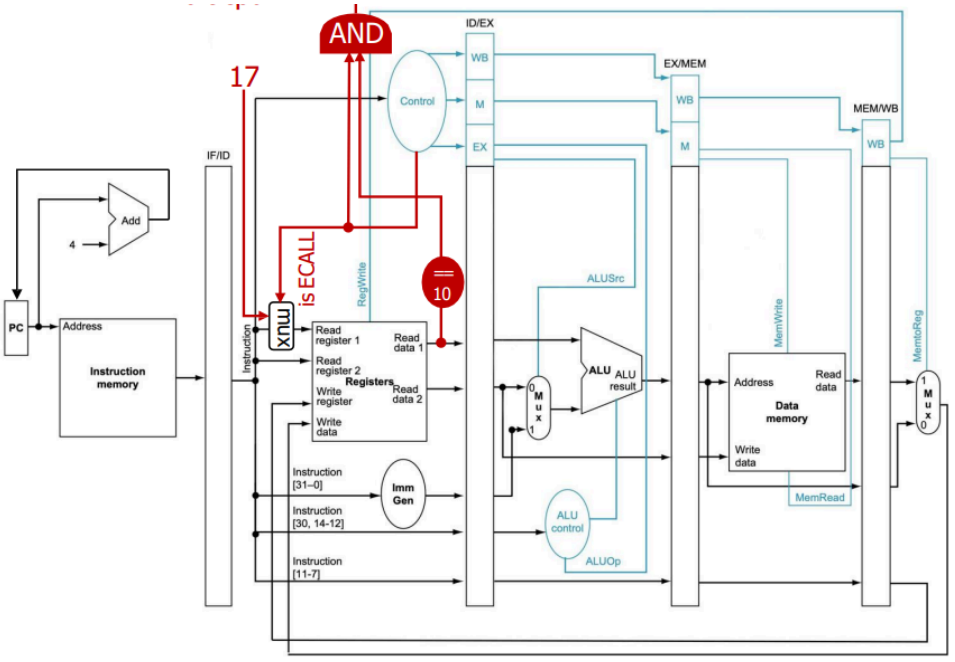
구현웅 (20210940), 김민서 (20220826)
4/28, 2025

1. Introduction

본 랩에서는 RV32I 명령어 집합을 구현하는 pipelined CPU를 설계한다. Single-cycle CPU를 바탕으로 pipelined의 개념을 설계 및 구현하고, 이로인해 발생하는 data hazard를 분석하여 이를 제어하기 위한 모듈을 제작한다. 최종적으로는, hazard로 인해 생기는 stall를 forwarding으로 최적화해나간다.

2. Design

본 랩의 CPU는 single-cycle CPU의 구조를 기반으로 하고 있으며, 각각의 stage 사이에 발생한 데이터를 다음 stage에서 사용할 수 있게 저장하는 register가 추가된다. 예를 들어, ID 단계에서는 pc에 값에 따라 instruction을 내보낸다. 다음 cycle에서는 새로운 pc에 의해 새로운 instruction이 출력되며, ID 단계에서는 이전 cycle에 생성된, IF/ID register에 저장된 instruction을 이용해 작업을 수행한다.



2.1. How to Pipelined CPU work

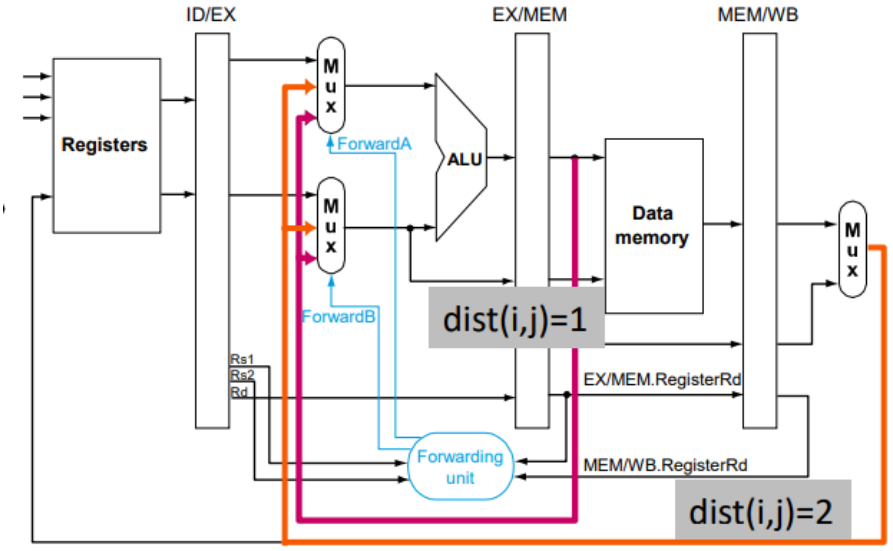
Pipeline의 핵심은 각각의 단계 사이에 register로 다음 단계에서 필요한 데이터를 저장하는데 있다. 매 cycle마다 새로운 instruction이 들어와 CPU 루틴을 돌게 되면서 각각의 유닛은 여러 사이클에 거친 일정한 출력을 제공해줄 수 없다. 이런 특징으로 사이클마다 생성된 출력을 stage 사이의 register 저장한다. Pipeline CPU에서 특정한 명령어는 각각의 stage를 순차적으로 진행한다고 해석할 수 있으며, control signal은 순차적으로 register에

저장되어 필요한 단계에 전달된다. 예를 들어, pc가 업데이트 되어 A라는 명령어 대한 instruction이 instuction memory의 출력으로서 IF/ID register 저장된다. 다음으로는 IF/ID의 저장된 A 명령어의 instruction으로 control signal를 생성하고 register unit에서 ALU의 입력을 생성한다. 다음 단계에 필요한 데이터들은 ID/EX register 저장된다. EX 단계에서도 마찬가지로 ID/EX register의 값을 이용해 데이터를 생성하고 이후 단계에 필요하다면 EX/MEM register 저장된다. MEM 단계에서는 ALU의 결과와 read, write signal로 동작을 수행하고 MEM/WB에 데이터를 저장할 것이다. 마지막으로 register를 계속 거쳐온 register unit의 신호와 데이터를 통해 write back를 수행한다.

2.2. Forwarding for data hazard

Pipeline의 큰 특징 중 하나는 특정 명령어가 끝나지 않더라도 계속해서 명령어를 CPU 루틴에 넣어준다는 것이다. 각각의 단계별로 특정한 명령어를 수행하고 있으며, 그 사이에 데이터 의존성이 존재한다면 문제가 발생하며, 이것이 hazard다.

Hazard는 여러 종류가 있지만 여기서는 data hazard만 다루며, register에 대해 서로 다른 두 instruction이 같은 위치에 대해 쓰기를 수행하고 읽는 경우에 hazard가 발생한다. 따라서 instruction 해석하는 단계에서 이미 들어간 cpu 내부에 동작 중인 instruction과 비교해 write back이 이루어지지 않은 register 공간에 접근한다고 하면, CPU에 무의미한 instruction를 보내는 것과 동일한 동작인 stall를 진행한다. 충분히 기다려 필요한 데이터가 write back이루어졌다면 stall은 해제되고 instruction은 ID stage에서 올바른 register 값을 읽고 EX 단계로 넘어갈 수 있다. Instruction에는 읽고 쓰기를 하는지에 대한 여부, 어디에 데이터를 참조하는지에 대한 정보를 알고 있으므로 단순한 논리 연산을 통해 stall 여부를 판단할 수 있다.



Stall를 통해 문제를 해결하는 건 CPU 성능에 굉장히 치명적이다. 실제로 write back 되어야할 데이터가 생성되는 시점은 EX 단계이거나 MEM 단계이며 이때 생성된 데이터를 바로 EX 단계의 입력으로 줄 수 있다면 stall를 최적화 할 수 있다. 단순하게 alu의 결과값을 다음 사이클에 alu의 입력으로 넣는 포트와 data memory의 결과를 바로 alu의 입력으로 넣는 포트를 설정하고 instruction를 비교해 이 포트를 사용할 지정해 주면 된다. 이러한 조건은 instruction의 비교를 통해서 구할 수 있다. 그리고 생성된 결과는 WB 단계를 거쳐서 forwarding과는 무관하게 register에 write back된다.

	R/I-Type	LW	SW	Bxx	JAL	JALR
IF						
ID						
EX	use produce	use	use	use	produce	use produce
MEM		produce	(use)			
WB						

가능한 모든 forwarding 구현한다면, control flow는 본 랩에서 다루지 않으니, R/I-type은 EX단계에서 데이터가 형성된 다음 사이클에 다시 alu의 입력으로 넣어줄 수 있을 것이다. LW의 경우 MEM 단계를 거쳐야 데이터가 형성되므로 MEM 단계의 다음 사이클부터 출력을 alu의 입력으로 넣어줄 수 있다. SW의 경우, 데이터를 write back하지 않으므로 관련된 forwarding은 없다. 반대로 R/I-type, LW, SW 모두 데이터는 EX 단계에서 사용함을 알 수 있다. 따라서 R/I-Type 뒤에 다른 instruction이 오는 경우에는 forwarding을 통해 stall를 완전히 해결할 수 있으며, LW의 경우 한 사이클 만큼의 stall이 필요하다.

2.3. Comparing total cycles

Single-cycle CPU에서는 고정적으로 한 cycle에 하나의 instruction이 수행되었다. 하지만 pipeline CPU에서는 한 cycle 마다 instruction이 계속 CPU에 들어가고 처음에 모든 stage를 채울 때까지는 아무런 결과가 나오지 않지만, 다 채워진다면 cycle 마다 instruction이 수행된다. 또한, 경우에 따라 pipelined cpu는 stall이 발생해 시스템이 딜레이된다.

```
main:
        addi    sp,sp,-32
        sw      ra,28(sp)
        sw      s0,24(sp)
        addi    s0,sp,32
        addi    a5,zero,19
        sw      a5,-20(s0)
        addi    a5,zero,14
        sw      a5,-24(s0)
        lw      a4,-24(s0)
        lw      a5,-20(s0)
        add     a5,a4,a5
        addi    a0,zero,0
        not     a0,a0
        or      a0,zero,a0
        andi    a0,a0,12
        addi    a0,a0,28
        slli    a0,a0,2
        srli    a0,a0,4
        add     a2,a0,zero
        li      a7, 31
        ecall
```

```
        add     a0,a2,zero
        ori     a6,zero,14
        addi    a7,zero,4
        sll     a7,a6,a7
        xor     a7,a7,a0
        sub     a6,a6,a0
        srl     a7,a7,a6
        addi    a1,zero,63
        and     a3,a7,a1
        not     a2,a7
        sub     a6,a7,a6
        srli    a7,a7,3
        or      a3,a4,a5
        lw      ra,28(sp)
        lw      s0,24(sp)
        addi    sp,sp,32
        li      a7, 10
        ecall
```

본 랩에서 주어진 프로그램으로 분석해 본 다면, single-cycle CPU에서는 명령어의 수 만큼 총 39 cycles이 필요할 것이다. Forwarding이 제공되는 pipelined CPU의 경우에는 먼저 각각의 단계에 instruction 들어가는데 총 5 cycles 만큼 걸리며, 이후 부터는 cycle 마다 instruction이 수행된다.

```
        lw      a5,-20(s0)
        add     a5,a4,a5
```

따라서 위의 두 명령어는 a5에 대해 data memory의 데이터를 쓰고 바로 읽기를 시도하므로 1 cycle stall이 걸린다. 그외에는 stall이 없으므로, CPU가 비어있어 허비되는 6 cycles, 명령어의 수 39 cycle, stall의 1 cycle해서, 모든 instruction이 종료될 때까지 총 46 cycles이 필요하다. 추후의 implementation은 본 cycle 분석을 통해 올바른 forwarding을 구현할 것이다.

3. Implementation

본 랩에서 구현한 pipelined CPU의 기본 토대는 single-cycle CPU를 그대로 따라가고, 여기에 pipelining, hazard detection 및 stall, 그리고 data forwarding이 추가로 구현되었다. pipelining, hazard detection, forwarding unit, 그리고 ecall의 구현을 중심으로 살펴보겠다.

3.1. Pipelining

Pipelining의 핵심은 이전 stage에서 계산되어 다음 stage로 전파될 데이터를 두 stage 사이의 레지스터에 임시로 저장하는 것에 있다. 이는 다음과 같이 구현된다.

```
// Update IF/ID pipeline registers here
always @(posedge clk) begin
  if (reset) begin
    IF_ID_inst <= 0;
    IF_ID_PC   <= 0;
  end else if (!is_stall) begin
    IF_ID_inst <= instruction;
    IF_ID_PC   <= current_pc;
  end
  // On else, IF_ID_inst doesn't not change
end

// Update ID/EX pipeline registers here
always @(posedge clk) begin
  if (reset) begin
    /* used in ex stage */
    ID_EX_inst <= 0;
    ID_EX_alu_src <= 0;
    ID_EX_rs1_data <= 0;
    ID_EX_rs2_data <= 0;
    ID_EX_imm <= 0;
    ID_EX_PC <= 0;
    ID_EX_ALU_ctrl_unit_input <= 0;
    ID_EX_mem_write <= 0;
    ID_EX_mem_read <= 0;
    ID_EX_rd <= 0;
    ID_EX_reg_write <= 0;
    ID_EX_mem_to_reg <= 0;
    ID_EX_is_halted <= 0;

  end else begin
    /* used in ex stage */
    ID_EX_inst <= IF_ID_inst;
    ID_EX_alu_src <= alu_src;
    ID_EX_rs1_data <= rs1_dout;
    ID_EX_rs2_data <= rs2_dout;
    ID_EX_imm <= imm;
    ID_EX_PC <= IF_ID_PC;
    ID_EX_ALU_ctrl_unit_input <= {IF_ID_inst[30], IF_ID_inst[14:12],
IF_ID_inst[6:0]};

    /* used in mem stage */
    ID_EX_mem_write <= mem_write;
    ID_EX_mem_read <= mem_read;

    /* used in wb stage */
```

```
    ID_EX_rd <= IF_ID_inst[11:7];
    ID_EX_reg_write <= write_enable;
    ID_EX_mem_to_reg <= mem_to_reg;
    ID_EX_is_halted <= is_ecall && ((forward_ecall ? EX_MEM_alu_out :
rs1_dout) == 10);
  end
end

// Update EX/MEM pipeline registers here
always @(posedge clk) begin
  if (reset) begin
    EX_MEM_alu_out <= 0;
    EX_MEM_dmem_data <= 0;
    EX_MEM_mem_read <= 0;
    EX_MEM_mem_write <= 0;
    EX_MEM_rd <= 0;
    EX_MEM_reg_write <= 0;
    EX_MEM_pc_imm <= 0;
    EX_MEM_is_halted <= 0;
  end else begin
    /* used in mem stage */
    EX_MEM_alu_out <= alu_result;
    EX_MEM_dmem_data <= alu_rs2;

    EX_MEM_mem_read <= ID_EX_mem_read;
    EX_MEM_mem_write <= ID_EX_mem_write;

    EX_MEM_pc_imm <= pc_imm;  // jal, branch

    /* used in wb stage */
    // EX_MEM_alu_out is also used
    EX_MEM_rd <= ID_EX_rd;
    EX_MEM_reg_write <= ID_EX_reg_write;
    EX_MEM_mem_to_reg <= ID_EX_mem_to_reg;
    EX_MEM_is_halted <= ID_EX_is_halted;
  end
end

// Update MEM/WB pipeline registers here
always @(posedge clk) begin
  if (reset) begin
    MEM_WB_rd <= 0;
    MEM_WB_alu_out <= 0;
    MEM_WB_reg_write <= 0;
    MEM_WB_mem_dout <= 0;
    MEM_WB_is_halted <= 0;
```

```
  end else begin
    MEM_WB_rd <= EX_MEM_rd;
    MEM_WB_alu_out <= EX_MEM_alu_out;
    MEM_WB_reg_write <= EX_MEM_reg_write;
    MEM_WB_mem_to_reg <= EX_MEM_mem_to_reg;
    MEM_WB_mem_dout <= mem_dout;

    MEM_WB_is_halted <= EX_MEM_is_halted;
  end
end
```

Data forwarding과 연관되지 않은 경우에 한하여, ID stage의 모듈은 IF_ID_* 레지스터의 값을, EX stage의 모듈은 ID_EX_* 레지스터를, MEM stage의 모듈은 EX_MEM_* 레지스터를, 그리고 WB stage의 모듈은 MEM_WB_* 레지스터의 값을 사용한다.

3.2. Hazard Detection

HazardDetection 모듈은 data hazard 상황을 감지하여 stall 여부를 결정한다. Stall의 구현은 program counter의 조작을 막고 현재의 값으로 유지시키고, 메모리와 레지스터에 값을 쓰지 못하게 막음으로써 이루어진다. Stall signall이 is_stall로 주어질 때, 다음과 같이 stall이 구현된다.

```
// PC
module PC (
    input             reset,
    input             clk,
    input             is_stall,
    input      [31:0] next_pc,
    output reg [31:0] current_pc
);
  initial begin
    current_pc = 0;
  end

  always @(posedge clk) begin
    if (reset) current_pc <= 0;
    else if (!is_stall) current_pc <= next_pc;
    //On else, not change
  end
endmodule

// CPU
module cpu (
  // ...
```

```
);
  // ...

  // Update IF/ID pipeline registers here
  always @(posedge clk) begin
    if (reset) begin
      IF_ID_inst <= 0;
      IF_ID_PC   <= 0;
    end else if (!is_stall) begin
      IF_ID_inst <= instruction;
      IF_ID_PC   <= current_pc;
    end
    // On else, IF_ID_inst doesn't not change
  end

// Control Unit
module ControlUnit (
    // ...
);
  always @(*) begin
    // ...
    // assign mem_write and write_enable
    // ...

    // stall
    mem_write    &= !is_stall;
    write_enable &= !is_stall;
  end
endmodule
```

Stall 여부의 결정은 HazardDetection 모듈에서 이루어진다. Data forwarding이 이루어질 때, data hazard 판단 방법을 알아보자. 다음은 언제 레지스터의 값이 사용되고 생성되는지를 나타낸 것이다.

	R/I-Type	LW	SW	Bxx	JAL	JALR	ECALL
IF							
ID							use
EX	use produce	use	use	use	produce	use produce	
MEM		produce	(use)				
WB							

LW가 다른 명령어보다 앞에 올 때, LW가 EX stage에 있다고 하면 stall condition은 ID_EX_mem_read && ((IF_ID_rs1 == ID_EX_rd && use_rs1) || (IF_ID_rs2 == ID_EX_rd && use_rs2) || (is_ecall && IF_ID_rs1 == EX_MEM_rd))이다. 여기서 use_rs1, use_rs2, is_ecall은 각각 ID stage에서 명령어가 rs1, rs2를 사용하는지 여부와 명령어가 ECALL인지 여부를 나타낸다.

ECALL의 경우는 바로 앞에 오는 명령어가 LW가 아니라고 하더라도, R/I-Type, JAL, JALR와 같이 EX stage에서 produce하여 ID_EX_reg_write가 1이어서 레지스터에 값을 쓰는 경우에도 stall이 필요하다. 이를 위하여 is_ecall && IF_ID_rs1 == ID_EX_rd && ID_EX_reg_write도 stall condition에 추가해야 한다.

```
module HazardDetection (
    input      [31:0] IF_ID_inst,
    input      [ 4:0] ID_EX_rd,
    input      [ 4:0] EX_MEM_rd,
    input             ID_EX_mem_read,
    input             ID_EX_reg_write,
    output reg        stall
);
  wire [4:0] IF_ID_rs1 = is_ecall ? 17 : IF_ID_inst[19:15];
  wire [4:0] IF_ID_rs2 = IF_ID_inst[24:20];
  wire [6:0] IF_ID_opcode = IF_ID_inst[6:0];
  reg        is_ecall;
  reg        use_rs1;
  reg        use_rs2;

  assign stall = ID_EX_mem_read && (
    (IF_ID_rs1 == ID_EX_rd && use_rs1) ||
    (IF_ID_rs2 == ID_EX_rd && use_rs2) ||
    (is_ecall && IF_ID_rs1 == EX_MEM_rd)) ||
    is_ecall && IF_ID_rs1 == ID_EX_rd && ID_EX_reg_write;

  always @(*) begin
    case (IF_ID_opcode)
      `ARITHMETIC:   {use_rs1, use_rs2, is_ecall} = 3'b110;
      `ARITHMETIC_IMM, `LOAD, `JALR:
                     {use_rs1, use_rs2, is_ecall} = 3'b100;
      `STORE:        {use_rs1, use_rs2, is_ecall} = 3'b110;
      `BRANCH:       {use_rs1, use_rs2, is_ecall} = 3'b110;
      `JAL:          {use_rs1, use_rs2, is_ecall} = 3'b000;
      `ECALL:        {use_rs1, use_rs2, is_ecall} = 3'b101;
      default:       {use_rs1, use_rs2, is_ecall} = 3'b000;
    endcase
    use_rs1 &= IF_ID_rs1 != 0;
    use_rs2 &= IF_ID_rs2 != 0;
  end
```

```
endmodule
```

3.3. Forwarding Unit

Data forwarding은 EX stage에서 rs1/rs2의 값을 직전 instruction의 MEM stage에서의 alu 출력값으로 할 것인지, 2 instruction 이전의 WB stage에서의 rd에 대한 입력 데이터 값으로 할 것인지, 또는 자기 자신의 앞선 ID stage에서 읽어온 값을 사용할지를 정한다. 이전 instruction의 stagef로부터 data forwarding을 할 경우는 더 앞선 stage로부터 forwarding 여부를 먼저 체크한다. 즉, 직전 insttruction의 MEM stage에서 forwarding 여부를 2 instruction 이전의 WB stage에서 forwarding 여부보다 먼저 확인해야 한다.

직전 instruction의 MEM stage에서 현재 instruction의 EX stage로 forwarding되는 경우는 MEM stage에서 reg_write가 1이고 EX stage에서 rs1이 x0이 아니며 EX_MEM_rs1 == ID_EX_rs1이어야 한다. 2 instruction 이전의 WB stage에서 forwarding되는 경우는 WB stage에서 reg_write가 1이고 EX stage에서 rs1이 x0이 아니며 MEM_WB_rs1 == ID_EX_rs1이어야 한다. 그 외의 경우는 data forwarding을 하지 않는다. rs2의 경우에도 동일하다.

```
// CPU module

mux32 alu_rs2_mux (
    .select(ID_EX_alu_src),
    .w0(alu_rs2),
    .w1(ID_EX_imm),
    .dout(alu_rs2_or_imm)
);

mux32_2 alu_rs1_forward_mux (
    .select(forward_A),
    .w0(ID_EX_rs1_data),
    .w1(rd_din),
    .w2(EX_MEM_alu_out),
    .dout(alu_rs1)
);

mux32_2 alu_rs2_forward_mux (
    .select(forward_B),
    .w0(ID_EX_rs2_data),
    .w1(rd_din),
    .w2(EX_MEM_alu_out),
    .dout(alu_rs2)
);

ALU alu (
    .alu_op    (alu_op),          // input
```

```
    .alu_in_1  (alu_rs1),         // input
    .alu_in_2  (alu_rs2_or_imm),  // input
    .alu_result(alu_result),      // output
    .alu_bcond (bcond)            // output
);
```

```
module ForwardingUnit (
    input      [31:0] ID_EX_inst,
    input      [ 4:0] EX_MEM_rd,
    input      [ 4:0] MEM_WB_rd,
    input             EX_MEM_reg_write,
    input             MEM_WB_reg_write,
    output reg [ 1:0] forward_A,
    output reg [ 1:0] forward_B,
    output reg [ 1:0] forward_ecall
);
  wire [4:0] ID_EX_rs1 = ID_EX_inst[19:15];
  wire [4:0] ID_EX_rs2 = ID_EX_inst[24:20];

  always @(*) begin
    if (EX_MEM_reg_write && ID_EX_rs1 != 0 && EX_MEM_rd == ID_EX_rs1)
begin
      forward_A = 2'b10;
    end else if (MEM_WB_reg_write && ID_EX_rs1 != 0 && MEM_WB_rd ==
ID_EX_rs1) begin
      forward_A = 2'b01;
    end else begin
      forward_A = 2'b00;
    end

    if (EX_MEM_reg_write && ID_EX_rs2 != 0 && EX_MEM_rd == ID_EX_rs2)
begin
      forward_B = 2'b10;
    end else if (MEM_WB_reg_write && ID_EX_rs2 != 0 && MEM_WB_rd ==
ID_EX_rs2) begin
      forward_B = 2'b01;
    end else begin
      forward_B = 2'b00;
    end

    if (EX_MEM_reg_write && EX_MEM_rd == 17) begin
      forward_ecall = 2'b10;
    end else if (MEM_WB_reg_write && MEM_WB_rd == 17) begin
```

```
      forward_ecall = 2'b01;
    end else begin
      forward_ecall = 2'b00;
    end
  end

endmodule
```

ecall의 경우는 데이터를 EX stage가 아닌 ID stage로 forward한다는 점을 제외하면 유사하다.

```
// CPU module

// Update ID/EX pipeline registers here
always @(posedge clk) begin
  // ...
  /* used in wb stage */
  ID_EX_is_halted <= is_ecall && (ecall_rs1_forwarded == 10);
end

mux32_2 ecall_forward_mux (
  .select(forward_ecall),
  .w0(rs1_dout),
  .w1(rd_din),
  .w2(EX_MEM_alu_out),
  .dout(ecall_rs1_forwarded)
);
```

4. Discussion

`non-controlflow_mem.txt`를 이전에 구현한 single cycle CPU와 이번 pipelined CPU에서 실행하였을 때 cycle 수는 각각 39 / 46 cycle이었다. single cycle CPU보다 pipelined CPU에서 cycle 수가 더 많은 것은 각 cycle에 하나의 stage만을 실행하므로 latency가 각각 5 / 1 cycle로 4 cycle 차이가 나며, 이 외에 `make waves`로 GTKWave를 실행하여 waveform을 확인해본 결과 pipelined cpu에서 stall이 3번 일어나기에 총 7 cycle 차이가 남을 설명할 수 있다.

ecall instruction의 구현 방법에 따라 전체 cycle 수가 달라짐을 확인할 수 있었다. x17 register의 값을 ID stage에서 비교하도록 하는 제한이 없다면, 비교를 EX stage에서 하도록 하여 정답보다 stall을 1 cycle 줄일 수 있었다. 구현 명세에 따라 ID stage에서 비교하도록 하였을 때는 data forwarding을 구현하였을 때 정답 사이클이 나오고, 구현하지 않았을 때는 정답보다 stall이 1 cycle 더 나타났다.

5. Conclusion

본 랩에서는 RV32I 명령어 집합을 구현한 pipelined CPU를 설계하였다. Single-cycle CPU의 구조에서 단계 사이사이에 데이터를 임시로 저장할 register를 추가해 구현하였으며, 이 과정에서 발생하는 새로운 문제인 hazard를 분석하고 stall로 해결하는 방법을 배웠다. 더 나아가 CPU 내부의 데이터 구조에 대해 이해하고 forwarding을 통해 hazard를 해소하는 법을 익혔다. Pipelining과 forwarding은 CPU 내에서만 적용되는 개념이 아닌 범용적으로 자원의 사용을 최적하는 기법이다. 본 랩에서의 경험은 CPU에 대한 이해 뿐 아니라, 컴퓨터 분야 전반에 대해 유용한 기술이 될 것이다.