

# 컴퓨터 구조 (CSED311)

## <Lab4-1 - Pipelined CPU>

Team ID : 33

Student 1 : 곽민성 (Gwak Minseong, 20230840)

Student 2 : 김재환 (Kim Jaehwan, 20230499)

### 1. Introduction

이번 랩의 목표는 Pipelined CPU를 Verilog를 통해 구현하는 것이다. Pipelined CPU는 명령어 하나를 실행하는데 여러 stage를 사용하며, 각 stage는 (hazard를 고려하지 않으면) 서로 연관을 주지 않기 때문에 병렬적으로 수행할 수 있고, 이를 통해 컴퓨팅 자원을 최대한 집중적으로 활용하고 더 빠르게 명령어를 수행할 수 있게 된다. 이번 랩 4-1에서는 Control Flow가 없는 Pipelined CPU를 만들고, 4-2에서는 Control Flow까지 포함한 완전한 Pipelined CPU를 제작하게 된다. 이 랩을 성공적으로 진행함으로써 Pipelined CPU의 세부 구조와 동작을 정확하게 이해할 수 있었다.

### 2. Design

우리는 아래 강의 노트의 CPU 구조를 사용하여 Pipelined CPU를 구현하였다.

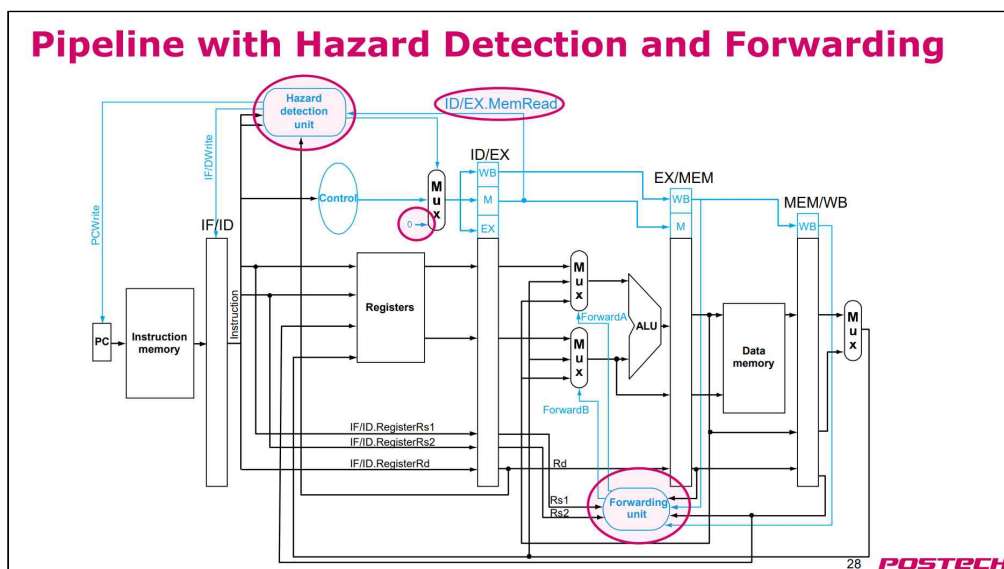


그림 1 Pipelined CPU Structure, with Hazard detection unit and Forwarding unit

#### Forwarding Logic

```
if (rs1_ex != x0) && (rs1_ex == rd_mem) && RegWrite_mem then
    forward operand from MEM stage // dist=1
Else if (rs1_ex != x0) && (rs1_ex == rd_wb) && RegWrite_wb then
    forward operand from WB stage // dist=2
else
    use the operand from register file // dist=3
```

Can this be done always?

- ◆ Do the same for rs2
- ◆ Ordering matters!! Must check the youngest match first! (MEM vs. WB)
- ◆ Why doesn't use\_rs1(IR<sub>ID</sub>) appear in the forwarding logic?
  - Recall: use\_rs1(I) returns true if I uses rs1 && rs1 != x0

그림 2 Forwarding Logic

#### Data Hazard Analysis (with Forwarding)

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

■ Even with data-forwarding, a hazard occurs for RAW dependence on an immediately preceding LW instruction

■ Stall = { [(rs1<sub>ID</sub> == rd<sub>EX</sub>) && use\_rs1(IR<sub>ID</sub>)] || [(rs2<sub>ID</sub> == rd<sub>EX</sub>) && use\_rs2(IR<sub>ID</sub>)] && MemRead<sub>EX</sub> }

i.e., op<sub>ID</sub> = LW/LH/...

그림 3 RAW on immediately preceding LW instruction, we still need STALL.

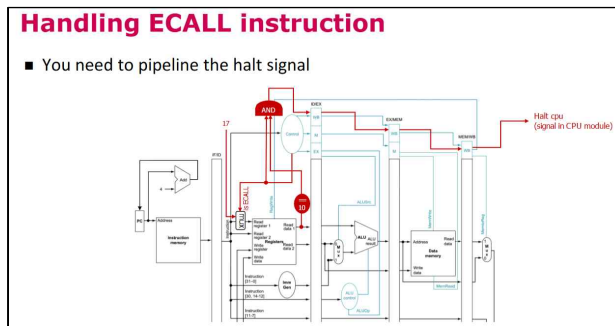


그림 4 ECALL instruction 처리

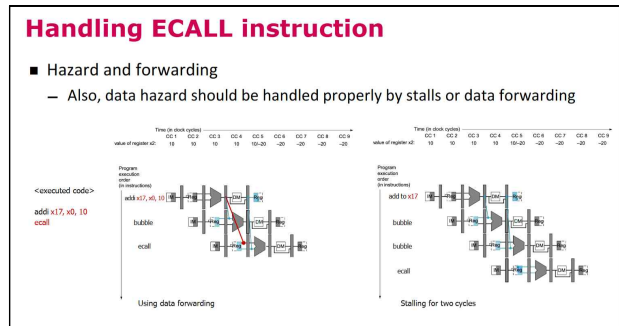


그림 5 We still need 1 Stall to handle ECALL data hazard

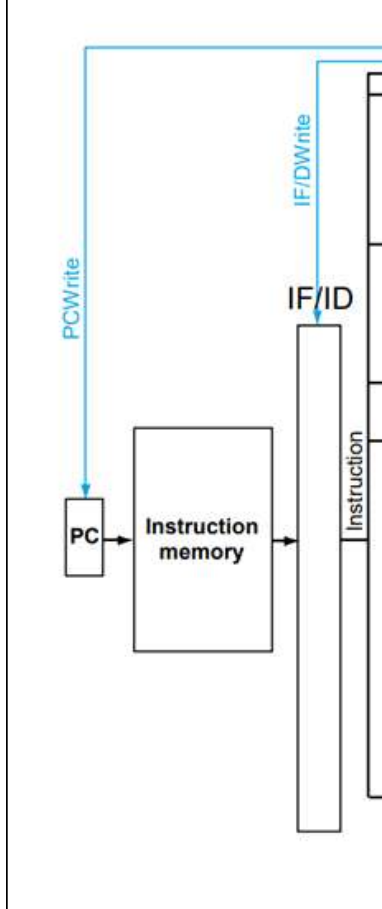
Pipelined CPU에서 모든 명령어는 IF, ID, EX, MEM, WB를 통과하는데 각 한 사이클씩 거치게 되며, 해당 단계를 수행하는데 필요한 전 단계의 명령어의 수행 결과와 Control Unit의 신호는 중간 단계의 레지스터에 저장되게 된다 (IF/ID, ID/EX, EX/MEM, MEM/WB). Pipelined CPU의 신호는 다른 사이클에 그 신호가 사용된다는 점을 빼면, 기본적으로 Single-cycle CPU와 동일하기 때문에 Lab2에서 구현했던 Single-cycle CPU의 Control Unit을 사용해서 구현하였다.

모듈 이름	설명	Synchronous / Asynchronous
ALU	두 수와 동작을 입력으로 받아, 산술 연산의 결과를 반환한다. Branch Instruction의 경우에, Branch Condition의 결과도 반환한다.	Asynchronous
ALUControlUnit	instruction을 입력으로 받아, ALU가 수행해야 하는 동작을 정의하고, 반환한다.	Asynchronous
ControlUnit	각 Instruction마다 다른 모듈에서 필요한 Control 신호들을 생성한다.	Asynchronous
DataForwardingUnit	Dataforwarding이 필요한 시점 (ID에서 현재 EX나 MEM 단계를 수행 중인 값이 필요할 때)에 그 값을 레지스터에서 가져오도록 Mux 신호를 생성한다.	Asynchronous
Datamemory	메모리의 주소와 값을 입력으로 받아, 메모리상의 주소의 위치의 값을 반환하거나(mem_read), 그 주소에 들어온 값을 쓴다 (mem_write).	Asynchronous (Read), Synchronous (Write)
HazardDetectionUnit	현재 RAW Hazard 상태이고, Stall을 해야 하는지 검사해서, PCWrite신호와 IF/ID 레지스터 Write 신호를 통해 Stall을 구현한다. Stall이 필요한 경우는 그림 3, 그림 5를 통해 확인할 수 있다. 각각 (1) LW 직후의 해당 레지스터 접근시, (2) x17 레지스터 접근 이후 ECALL 명령어 수행시 이다.	Asynchronous

ImmediateGenerator	Instruction을 읽어서, 그 Instruction에 있는 Immediate value를 파싱해서 반환한다.	Asynchronous
InstMemory	Instruction 주소를 입력으로 받아, Instruction 메모리상의 주소의 위치의 값을 반환한다 (즉, Instruction을 반환한다.)	Asynchronous (Read), Synchronous (Write)
Mux2 / Mux4	두 개의 입력과 1-bit의 sel 신호(Mux2), 4개의 입력과 2-bit의 sel 신호(Mux4)를 받아, sel 신호가 지시하는 입력을 그대로 반환한다.	Asynchronous
PC	다음 클럭의 pc를 입력으로 받아서, clk이 변화함에 맞추어 PCWrite 신호가 있을 때 pc(program counter)의 값을 다음 값으로 업데이트한다.	Synchronous
RegisterFile	2개의 source 레지스터와 destination 레지스터의 번호, 입력값을 받아서 2개의 레지스터의 값을 반환하고, destination 레지스터에 값을 쓴다 (RegWrite).	Asynchronous (Read), Synchronous (Write)

### 3. Implementation

우리는 위에서 구현된 모듈을 cpu에서 다음과 같이 wire를 이어서 구현하였다.

	Instruction Fetch	<pre> Mux2 PC_mux(     .sel(IF_PCsrc),          // input     .in0(IF_current_pc plus 4), // input     .in1(EX_MEM_branch_addr), // input     .out(IF_next_pc)        // output );  wire [31:0] IF_current_pc; // PC must be updated on the rising edge (positive edge) of the clock. PC pc(     .reset(reset),          // input (Use reset to initialize PC. Initial value must be 0)     .clk(clk),              // input     .next_pc(IF_next_pc),   // input     .PC_Write(PC_Write),    // input     .current_pc(IF_current_pc) // output );  Adder pc_adder(     .in0(IF_current_pc),    // input     .in1(4),                // input     .out(IF_current_pc_plus_4) // output );  wire [31:0] IF_instr; // ----- Instruction Memory ----- InstMemory imem(     .reset(reset),          // input     .clk(clk),              // input     .addr(IF_current_pc),   // input     .dout(IF_instr)         // output );         </pre>
IF	Reference Structure	

## Instruction Decode

```
// ----- Control Unit -----
ControlUnit ctrl_unit (
  .Instr(IF_ID_inst[6:0]), // input
  .MemRead(ID_ctrl_mem_read), // output
  .MemToReg(ID_ctrl_mem_to_reg), // output
  .MemWrite(ID_ctrl_mem_write), // output
  .ALUSrc(ID_ctrl_alu_src), // output
  .RegWrite(ID_ctrl_write_enable), // output
  .PCtoReg(ID_ctrl_pc_to_reg), // output
  .Branch(ID_ctrl_branch), // output
  .is_ecall(ID_ctrl_is_ecall) // output (ecall inst)
);

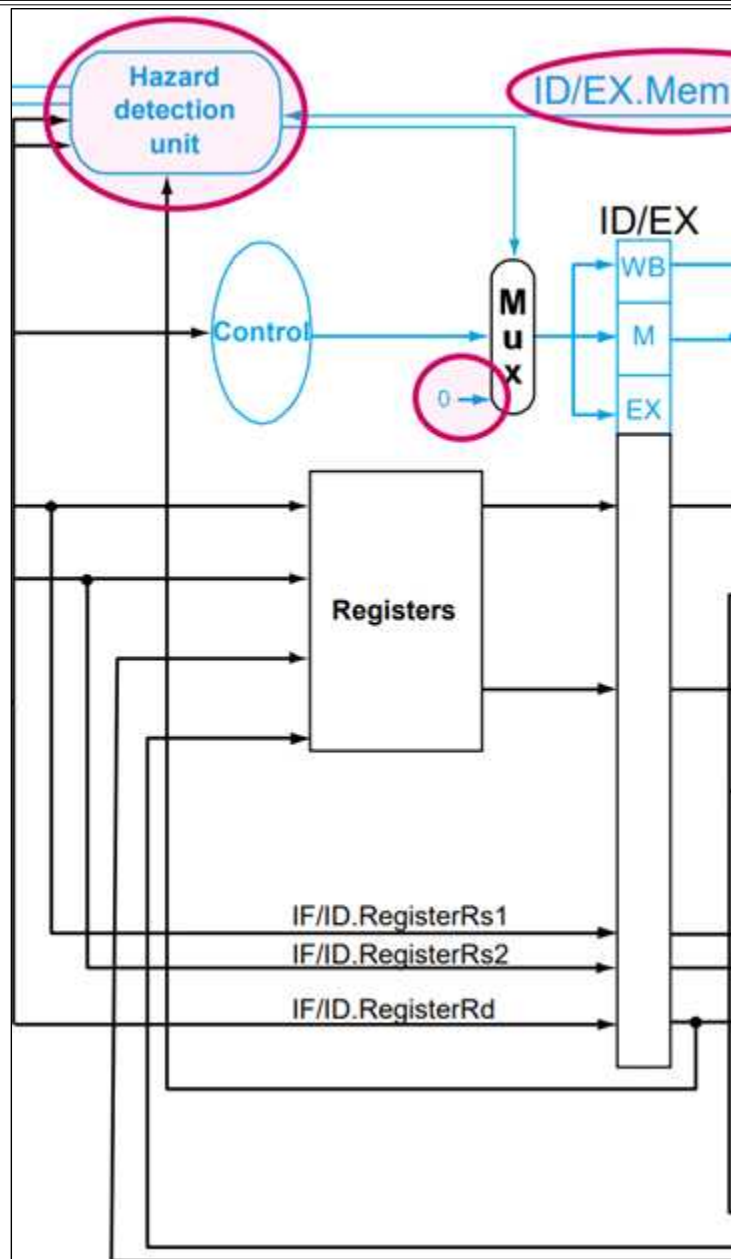
assign ID_is_halted = ID_ctrl_is_ecall && ((forward_ecall == 0 ? ID_rsl_dout : EX_MEM_alu_out) == 10);

// ----- Hazard Detection Unit -----
wire PC_Write, IF_ID_Write, ID_CtrlUnitMux_sel;
HazardDetectionUnit haz_detect_unit(
  .opcode(IF_ID_inst[6:0]),
  .ID_rsl(ID_rsl), // input
  .ID_rs2(ID_rs2), // input
  .ID_EX_rd(ID_EX_rd), // input // (TODO) EX_MEM_rd vs ID_EX_rd ??
  .ID_EX_mem_read(ID_EX_mem_read), // input
  .ID_ctrl_is_ecall(ID_ctrl_is_ecall),
  .PC_Write(PC_Write),
  .IF_ID_Write(IF_ID_Write),
  .ID_CtrlUnitMux_sel(ID_CtrlUnitMux_sel)
);

wire [31:0] ID_imm_out;
// ----- Immediate Generator -----
ImmediateGenerator imm_gen(
  .Instr(IF_ID_inst), // input
  .imm_gen_out(ID_imm_out) // output
);
```

ID

## Reference Structure



EX

ALU/Excute

```
wire [31:0] EX_branch_addr;
Adder branch_adder(
    .in0(ID_EX_PC),          // input
    .in1(ID_EX_imm),         // input
    .out(EX_branch_addr)     // output
);

wire [7:0] EX_alu_op;
// ----- ALU Control Unit -----
ALUControlUnit alu_ctrl_unit (
    .instr(ID_EX_ALU_ctrl_unit_input), // input
    .alu_op(EX_alu_op)                 // output
);

// ----- Data Forwarding Unit -----
wire [1:0] forward_a;
wire [1:0] forward_b;
wire forward_ecall;
DataForwardingUnit data_fw_unit (
    .ID_EX_rs1(ID_EX_rs1),
    .ID_EX_rs2(ID_EX_rs2),
    .EX_MEM_rd(EX_MEM_rd),
    .MEM_WB_rd(MEM_WB_rd),
    .EX_MEM_reg_write(EX_MEM_reg_write),
    .MEM_WB_reg_write(MEM_WB_reg_write),
    .ID_ctrl_is_ecall(ID_ctrl_is_ecall),
    .forward_a(forward_a),
    .forward_b(forward_b),
    .forward_ecall(forward_ecall)
);

wire [31:0] EX_alu_in1, EX_alu_in2;
Mux4 DataforwardA (
    .sel(forward_a),
    .in0(ID_EX_rs1_data),
    .in1(WB_ID_rd_din),
    .in2(EX_MEM_alu_out),
    .in3(0),
    .out(EX_alu_in1)
);

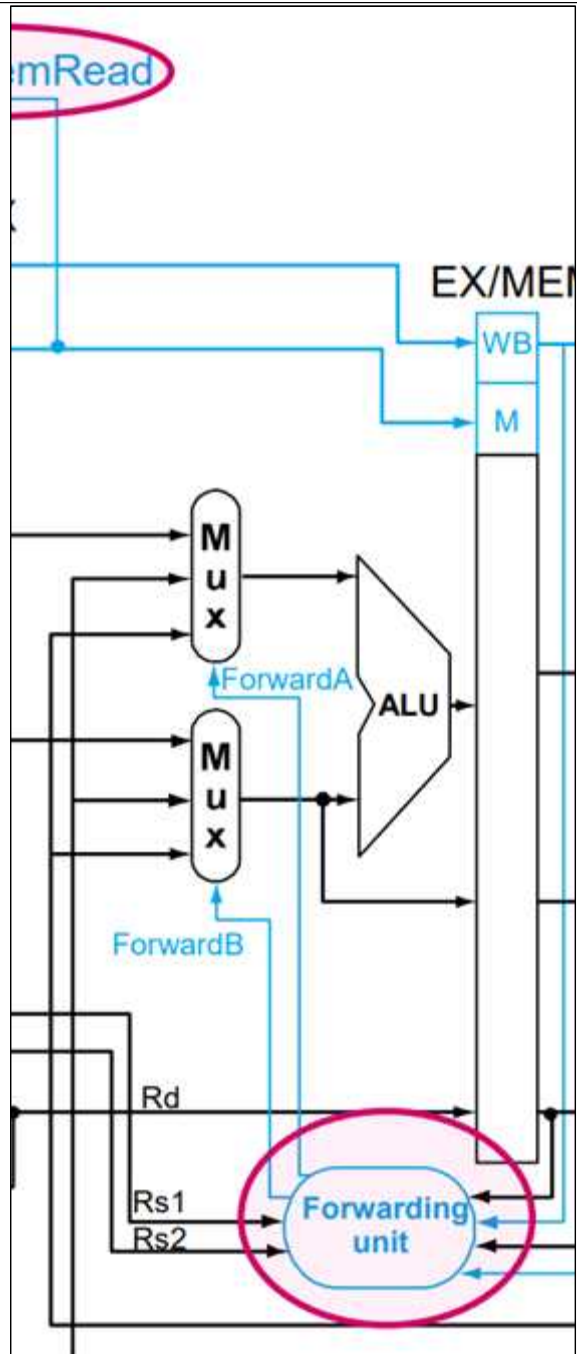
wire [31:0] EX_alu_src2;
Mux4 DataforwardB (
    .sel(forward_b),
    .in0(ID_EX_rs2_data),
    .in1(WB_ID_rd_din),
    .in2(EX_MEM_alu_out),
    .in3(0),
    .out(EX_alu_src2)
);

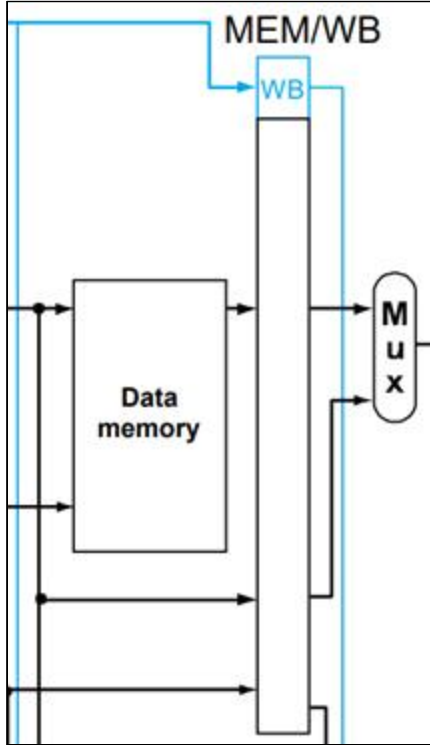
Mux2 ALU_in2_mux(
    .in0(EX_alu_src2),
    .in1(ID_EX_imm),
    .sel(ID_EX_alu_src),
    .out(EX_alu_in2)
);

wire [31:0] EX_alu_result;
wire EX_alu_bcond;
// ----- ALU -----
ALU alu (
    .alu_op(EX_alu_op),          // input
    .alu_in_1(EX_alu_in1),      // input
    .alu_in_2(EX_alu_in2),      // input
    .alu_result(EX_alu_result), // output
    .alu_bcond(EX_alu_bcond)    // output
);
```

EX

Reference Structure



MEM	Data Memory Access	<pre> wire [31:0] MEM_dout; // ----- Data Memory ----- DataMemory dmem(     .reset (reset),          // input     .clk (clk),              // input     .addr (EX_MEM_alu_out),  // input     .din (EX_MEM_dmem_data), // input     .mem_read (EX_MEM_mem_read), // input     .mem_write (EX_MEM_mem_write), // input     .dout (MEM_dout)         // output ); </pre>
	Reference Structure	 <p>The diagram illustrates the data path for the MEM stage. A vertical bus labeled 'MEM/WB' at the top is connected to a 'Data memory' block. The 'Data memory' block has multiple inputs and outputs connected to this bus. A 'Mux' (multiplexer) is shown on the right, with its output connected to the 'WB' (Write-Back) port of the 'MEM/WB' bus. The 'WB' port is also connected to the 'Data memory' block.</p>
WB	Write-Back	<p>위 동작에서, memory 모듈과 RegisterFile 모듈로 다시 들어가는 wire를 통해 Write-Back 된다.</p>



## 4. Discussion

A. How does your pipelined CPU work?

명령어 실행 과정을 IF/ID/EX/MEM/WB 다섯 단계로 나눠서, 각 단계마다 다른 명령어를 동시에 처리하여 높은 처리량을 보여준다. Hazard가 잘 처리되고 stall이 많이 작동하지 않다면 single cycle CPU와 비교하여 비슷한 사이클 수를 갖는 대신 클럭 주기를 더 짧게 설정하여 더 빠른 처리 속도를 가질 수 있다. 대신 구조가 더 복잡하며 hazard를 잘 처리하도록 구현해야한다.

B. Compare total cycles between the single cycle and pipelined CPU

Single cycle CPU의 경우 39 사이클, data forwarding을 구현한 pipelined CPU의 경우 46 사이클이 걸린다. 이는 첫 번째 명령어의 IF/ID/EX/MEM을 수행하는데 추가로 4사이클이 필요하고, 명령어 실행 과정에서 과정에서 RAW Hazard에 의한 3번의 STALL이 있기 때문이다.

C. How to implement hazard detection? When detected?

데이터 포워딩을 구현하지 않은 경우 - 현재 ID 단계에서 사용하고자 하는 소스 레지스터가 실제로 사용되면서, EX단계 또는 MEM단계에서 목적지 레지스터이면서 쓰고자 하는 경우 항상 stall이 필요하다.

### ■ Helper functions

– `use_rs1(l)` returns true if `l` uses `rs1` && `rs1!=x0`

Why do we need this?

### ■ Stall when

```
(rs1_ID == rd_EX) && use_rs1(IR_ID) && RegWrite_EX or  
(rs1_ID == rd_MEM) && use_rs1(IR_ID) && RegWrite_MEM or  
(rs2_ID == rd_EX) && use_rs2(IR_ID) && RegWrite_EX or  
(rs2_ID == rd_MEM) && use_rs2(IR_ID) && RegWrite_MEM
```

데이터 포워딩을 구현한 경우 - LW 명령을 제외한 나머지의 경우는 데이터 포워딩을 통해 결과를 미리 전달할 수 있으므로 stall이 발생하지 않는다. 단, LW 명령의 경우 메모리에서 불러오는데 시간이 필요하므로 이 경우만 따로 확인한다. 현재 ID 단계에서 사용하고자 하는 소스 레지스터가 EX단계에서의 목적지 레지스터이면서, 그 목적지 레지스터가 메모리의 값을 쓰고자 하는 경우.

```
■ Stall = { [(rs1_ID == rd_EX) && use_rs1(IR_ID)] || [(rs2_ID == rd_EX) && use_rs2(IR_ID)] }  
            && MemRead_EX  
            └──────────┘  
            i.e., op_EX=LW/LH/..
```

27

D. How to implement data forwarding? When forwarding?

현재 EX단계에서 사용하고자 하는 레지스터가 MEM단계에서 나온 값에 의해 쓰여질 예정이거나, WB 단계에서 쓰여지고 있는 경우에 Data forwarding이 필요하다. 이 경우가 아니라면 현재 저장되어 있는 레지스터 값이 최신값이므로 그대로 사용한다. (그림 2)

### E. ECALL 명령어

특수한 경우로, ECALL 명령어를 실행할 때 x17 레지스터가 필요하다. 이때는 다른 경우와 다르게 ID 단계 중에 x17 레지스터가 필요하기 때문에, 만약 전 instruction이 x17에 작성하는 명령이라면 1단계 stall이 필수적으로 필요하다. 이 경우를 Hazard Detection Unit과 Data Forwarding Unit을 통해 추가적으로 구현하였다.

## 5. Conclusion

위에서 설명한 구조대로 CPU를 구현하여 제공된 모든 테스트 벤치에 대해서 정답과 같은 시뮬레이션 결과를 얻었다. 사이클 수 및 모든 레지스터 결과가 주어진 정답과 동일함을 확인할 수 있다.

	Test Bench
non-controlflow	<pre>### SIMULATING ### TEST END SIM TIME : 94 TOTAL CYCLE : 46 (Answer : 46) FINAL REGISTER OUTPUT 0 00000000 (Answer : 00000000) 1 00000000 (Answer : 00000000) 2 00002ffc (Answer : 00002ffc) 3 00000000 (Answer : 00000000) 4 00000000 (Answer : 00000000) 5 00000000 (Answer : 00000000) 6 00000000 (Answer : 00000000) 7 00000000 (Answer : 00000000) 8 00000000 (Answer : 00000000) 9 00000000 (Answer : 00000000) 10 0000000a (Answer : 0000000a) 11 0000003f (Answer : 0000003f) 12 ffffffff1 (Answer : ffffffff1) 13 0000002f (Answer : 0000002f) 14 0000000e (Answer : 0000000e) 15 00000021 (Answer : 00000021) 16 0000000a (Answer : 0000000a) 17 0000000a (Answer : 0000000a) 18 00000000 (Answer : 00000000) 19 00000000 (Answer : 00000000) 20 00000000 (Answer : 00000000) 21 00000000 (Answer : 00000000) 22 00000000 (Answer : 00000000) 23 00000000 (Answer : 00000000) 24 00000000 (Answer : 00000000) 25 00000000 (Answer : 00000000) 26 00000000 (Answer : 00000000) 27 00000000 (Answer : 00000000) 28 00000000 (Answer : 00000000) 29 00000000 (Answer : 00000000) 30 00000000 (Answer : 00000000) 31 00000000 (Answer : 00000000) Correct output : 32/32</pre>

위와 같이 정상 동작하는 모습을 확인할 수 있다.

## 6. References

[1] Lecture notes.