1. Introduction

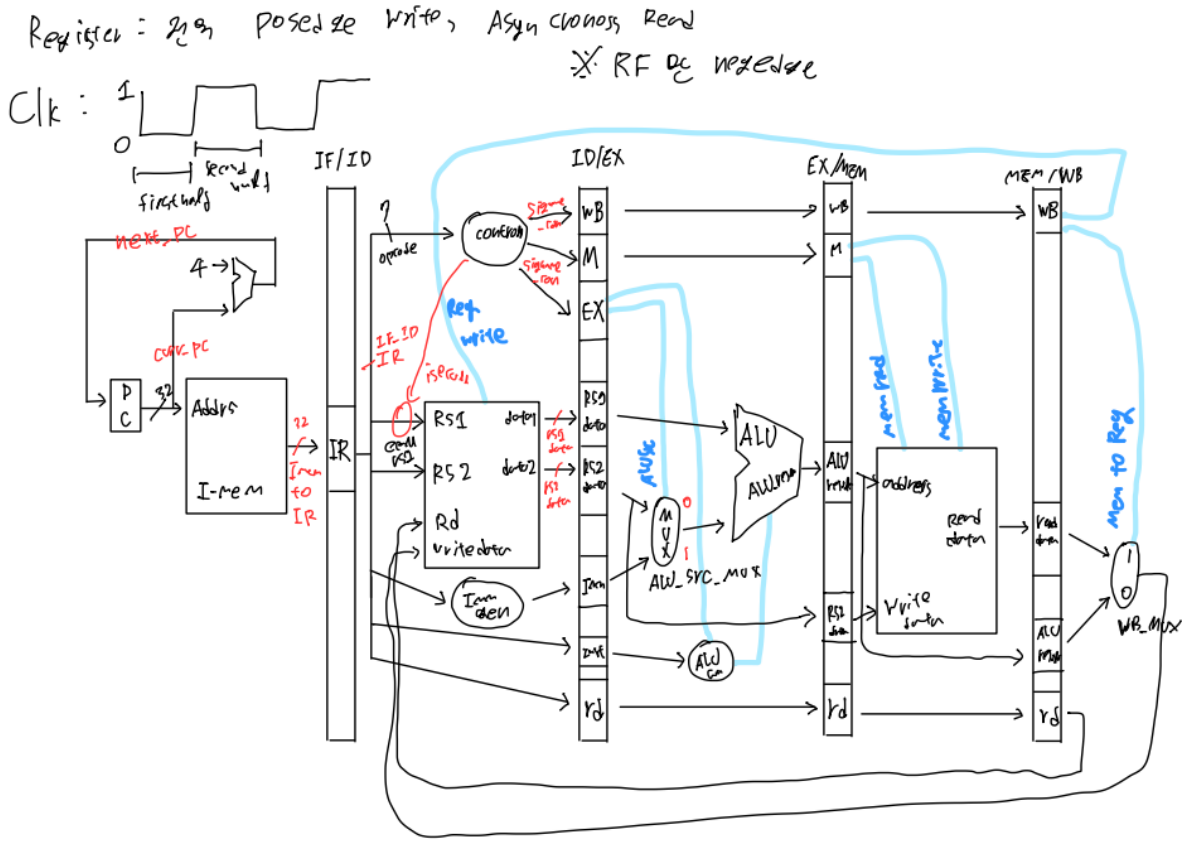
이번 과제의 목표는 RISC-V Five stage Pipelined CPU를 구현하는 것이다.

Pipelined CPU는 CPU내의 리소스들이 전부 동시에 실행될 필요는 없다는 점에 착안하여, Instruction 실행 과정을 여러 stage로 나누어, **각 stage마다 다른 instruction 이 동시에 실행**하도록 만듦으로써, CPU의 속도를 크게 향상시킬 수 있다.

이번 프로젝트에서 구현하는 Five stage Pipeline은 instruction의 실행과정을 IF, ID, EX, MEM, WB 다섯 stage로 나누어 설계한다.

2. Design

Five stage pipeline design



<수업시간에 배운 Pipeline 구조를 기반으로 한 Pipeline 디자인 설계도>

A. How does your pipelined CPU work?

우리가 디자인한 Pipeline CPU가 Instruction을 실행하는 과정은 다섯 단계로 나뉜다:

IF stage:

Instruction을 Fetch 하는 stage이다.

Program Counter값에 해당하는 Instruction을 I-mem으로부터 Fetch하고 IF/ID stage Pipeline Register에 저장한다.

이번 과제에서는 control flow가 없으므로, Program Counter는 무조건 4씩 증가되어, 다음 사이클에 다음 Instruction을 받아오도록 한다.

ID stage:

IF/ID stage Pipeline Register로부터 Instruction을 받아와 Decode 하는 stage이다.

Control unit은 Instruction을 기반으로 각종 control signal을 생성하고, ID/EX stage Pipeline Register에 저장한다.

Register File은 Instruction의 rs1/rs2 field값에 해당하는 레지스터의 값을 읽어 ID/EX stage Pipeline Register에 저장한다.

Immediate generator는 Instruction의 내용에 따라 Immediate를 generate 하고, ID/EX stage Pipeline Register에 저장한다.

이후 stage들에서 필요한 rd, func7, func3 field 값도 ID/EX stage Pipeline Register에 저장한다.

이때, 다음 stage들에 있는 older instruction의 rd값과 현재 IDstage의 rs값을 비교하여 만약 Data Hazard가 감지되면, Data hazard가 해소될 때까지 ID stage의 Instruction이 진행하지 못하도록 멈추는 stall이 발생하게 된다.

EX stage:

ID/EX stage Pipeline Register로부터 rs1, rs2, immediate 등의 값을 받아와 ALU가 필요한 연산을 하는 stage이다.

ALU가 수행하는 연산의 종류와 MUX의 동작은 ID stage에서 ID/EX stage Pipeline Register에 저장해둔 control 신호와 func3, func7등의 값을 바탕으로 결정된다.

ALU의 결과 값과, 이후 stage들에서 필요한 rd값과 control 신호들이 EX/MEM stage Pipeline Register에 저장된다.

이때, Data hazard 해소를 위해, older instruction의 rd값과 현재 EXstage의 rs값이 일치한다면, older instruction이 진행중인 stage로부터 데이터를 끌어오는 data forwarding이 가능하다.

MEM stage:

EX/MEM stage Pipeline Register에 저장된 ALU output값을 바탕으로, Store Instruction이라면 D-mem으로부터 데이터를 가져오고, Load instruction이라면 데이터를 읽어온다.

Store Instruction이라면 ID stage에서 생성되어 지금까지 전달된 mem_write신호가 1일 것이고, Load instruction이라면 ID stage에서 생성되어 지금까지 전달된 mem_read신호가 1일 것이다.

ALU output, Mem read data, rd값, control signal 등이 MEM/WB stage Pipeline Register에 저장된다.

WB stage:

ID에서 생성되어 지금까지 전달된 rd값에 해당하는 레지스터에, 필요한 정보를 Write Back하는 stage이다.

ALU output와 Mem read data중 어느 데이터를 사용할지는 ID stage에서 생성되어 지금까지 전달된 신호를 바탕으로 mux가 정하며, 만약 Write Back이 필요 없는 Instruction이라면, 마찬가지로 지금까지 전달된 Reg_write신호가 0일 것이다.

B. How to implement hazard detection?

앞서 간단하게만 언급한 ID stage의 Data Hazard detection에 대해 자세하게 설명하겠다.

이번 과제 CPU에서 Data hazard는 Read After Write dependency에 의해 발생한다.

Older Instruction의 Destination Register와 그 뒤에 오는 Younger Instruction의 Source Register가 일치하는데, 둘의 거리가 너무 가깝다면, Older Instruction이 아직 Pipeline내에서 실행 중이어서 Destination Register에 값을 쓰기도 전에 Younger Instruction이 해당 Register를 읽어버린다면 잘못된 값을 읽는 것이다.

	R/I-Type	LW	SW	Bxx	JAL	JALR
IF						
ID	read RF	read RF	read RF	read RF		read RF
EX						
MEM						
WB	write RF	write RF			write RF	write RF

<각 Instruction type마다 Register를 읽고 쓰는 stage를 나타낸 표>

1.Hazard detection without forwarding

즉, 두 instruction중 앞 instruction의 rd와 뒤 instruction의 rs값이 일치한다면, **위 표에서 앞 instruction의 write RF stage보다 뒤 instruction의 read RF stage가 먼저 오면 안 되며,**

이 '안전거리'를 유지하지 못하여 data hazard가 발생하면, 뒤 instruction을 stall시켜서 data hazard가 해소될 때까지 기다려야 한다. 위 표를 보면 이 '안전거리'가 3임을 알 수 있으며, 따라서 **ID stage의 rs값과 EX, MEM, WB stage의 rd값과 비교하여** hazard를 detect해야 한다.

2.Hazard detection with forwarding

그러나 실제로는 **Register File을 거치지 않아도 필요한 데이터 자체는 CPU Pipeline내에 존재하고 있다.** 예를 들자면, R type instruction은 **EX stage만 끝나면** 이미 ALU output에 Write Back 할 데이터는 들어 있는 것이다.

마찬가지로, R type instruction이 rs데이터가 필요한건 ID stage가 아니라 **EX stage 시작할 때**이다.

이런 식으로, 필요한 데이터를 CPU Pipeline 내부에서 앞으로 끌어오면 Data hazard를 크게 완화시킬 수 있다.

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

<RF이 아닌, data produce와 use를 기준으로 나타낸 표>

보면 위의 표 보다 hazard distance가 훨씬 줄어든 것을 볼 수 있다. 이를 이용하면, EX stage에서 Produce된 data를, 설령 바로 다음 Instruction에서 필요로 하더라도, MEM stage에서 EX stage로 해당 데이터를 forwarding 하면 되므로, 더 이상 data hazard를 발생시키지 않는다!

따라서 위 표를 보면, 모든 instruction이 EX stage에서 rs data를 필요로 하는데, 앞 instruction이 EX에서 data를 produce했다면 forwarding하면 되므로 더 이상 data hazard를 발생시키지 않지만, MEM 단계에서 data가 produce되는 **Load instruction만 data hazard를 발생**시킴을 알 수 있다.

3. Hazard detection actual implementation

위의 두 가지 hazard detection방식 중, 우리 조는 CPU에 data forwarding을 구현하였으므로, 2번의 **Hazard detection with forwarding** 방식을 적용하였다. 이 경우 **load instruction의 Destination Register와 바로 그 뒤에 오는 Younger Instruction의 Source Register가 일치**할 때에만 data hazard가 발생한다. 이때, younger instruction 은 ID stage에 있을 것이고, 바로 이전 Load instruction은 EX stage에 있을 것이다. 이를 알고리즘으로 표현하면 다음과 같다.

if (ID_EX_mem_read && //EX stage에 있는 instruction이 Load 인지 확인

((ID_EX_rd_field == IF_ID_rs1_field && IF_ID_rs1_field != 0) //EX_rd와 ID_rs1일치 확인

|| (ID_EX_rd_field == IF_ID_rs2_field && IF_ID_rs2_field != 0))) //EX_rd와 ID_rs2일치 확인

begin <stall> end

이때, rs가 x0인지 확인하는 이유는, x0는 언제나 0으로 고정이므로, 영향을 받지 않기 때문이다.

4. Hazard detection implementation – ecall

그러나, 위 상황에는 예외가 존재한다. 바로 ecall instruction이다. ecall instruction이 들어오면 무조건 ID stage에서 RF로부터 x17값을 읽어야 하기 때문이다. 따라서 이 경우 forwarding이 적용될 수 없기에, 앞의 명령어들이 rd가 x17레지스터라면, data hazard해소를 위해 Write Back될 때까지 기다려야 한다. 따라서 이 경우에는 hazard detection 조건이 달라진다. 그 조건은 다음과 같다:

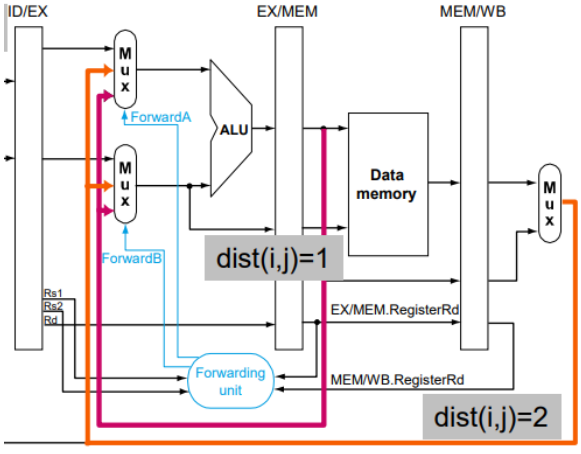
if(is_ecall && (ID_EX_rd_field == 'd17 || EX_MEM_rd_field == 'd17))

begin <stall> end

앞서 Write Back될 때까지 기다려야 한다고 했는데 위 알고리즘에서 EX 와 MEM stage만 확인하는 이유는, Write Back가 먼저 진행되고, 남은 half clk cycle동안 Reg read가 일어나기에, 둘이 같은 사이클에 일어나도 data hazard가 발생하지 않기 때문이다. (Internal forwarding)

C. How to implement data forwarding?

이번엔 앞서 간단하게만 언급한 EX stage의 data forwarding에 대해 자세하게 설명하겠다.



바로 위의 hazard detection 단락에서 설명하였듯이, Instruction이 실제로 data가 필요한 시점은 EX stage에 진입할 때이다.

그러나, ID stage에서 앞의 Instruction이 이미 Pipeline을 빠져나갔거나, WB stage를 진행중이라 Internal forwarding이 가능했다면 문제가 없으나,

data hazard를 발생시키는 instruction 들이 아직 EX, MEM 단계에 있었다면, 이를 forwarding 해서 가져와야 한다.

따라서, 각각 EX/MEM stage Pipeline Register에 저장 되어있는 ALU out과, MEM/WB stage Pipeline

Register에 저장 되어있는 Write Back Data로부터 데이터를 끌어오는 Forwarding path를 두 개 만들고, 이를 각각 ALU input양쪽에 MUX를 이용해 연결하면 된다.

ALU input 1 MUX가 각 input을 선택하는 logic은 다음과 같다:

```
if (EX_MEM_reg_write && (ID_EX_rs1_field == EX_MEM_rd_field) && ID_EX_rs1_field !=0)
<forward from MEM stage>

else: if (MEM_WB_reg_write && (ID_EX_rs1_field == MEM_WB_rd_field) && ID_EX_rs1_field !=0)

<forward from WB stage>

else:

default (rs1)
```

ALU input 2 MUX도 rs1이 rs2로 바뀌었을 뿐, logic은 같다.

즉, EX stage에 들어가는 중인 Instruction의 rs값과, MEM stage의 rd를 비교하여, 일치하면 EX/MEM stage Pipeline Register로부터 ALU out을 forwarding 하고, 불일치 시 EX stage에 들어가는 중인 Instruction의 rs값과, WB stage의 rd를 비교하여 Write Back 중인 데이터를 끌고 오는 것이다. 이 마저도 해당 사항이 없으면 data hazard가 애초에 없었다는 것이므로, 원래대로 ID/EX stage Pipeline Register에서 rs1 rs2 값을 받아오면 된다.

이때 MEM stage를 WB stage보다 먼저 비교하는 이유는, MEM stage의 데이터가 더 최신 데이터이기 때문이다.

위 logic에서 추가적으로 EX_MEM_reg_write를 확인하는 이유는 실제로 write back이 되는지 확인하는 용도이며, x0인지 확인하는 이유는 x0는 언제나 0으로 고정인데, data forwarding을 해 버리면 이상한 값을 불러와서 에러를 일으킬 수 있기 때문이다.

D. How to implement stall?

이번엔 앞서 간단하게만 언급한 stall의 실제 작동 방식을 설명하겠다. stall의 목적은, **IF, ID stage의 진행은 멈추고, EX, MEM, WB단계만 진행시켜서** ID stage에 들어 와있는 instruction의 data hazard를 해소하는 것이다.

따라서 PC업데이트를 막아서 IF stage를 멈추고, IF/ID stage Pipeline Register의 업데이트도 막아서 ID stage를 stall 시켜야 한다.

또한, 이때 EX로 넘어가는 ID/EX stage Pipeline Register에 저장되는 control signal중 Mem_write와 Reg_write를 0으로 만들어야 한다.

그 이유는, 두 신호가 꺼져 있으면, 해당 instruction이 결국 programmer visible state에 아무런 변화도 가할 수 없으므로, nop (bubble)화 되기 때문이다.

always @(posedge clk)

if(stall){

Next_PC = Curr_PC;

IF/ID_reg = IF/ID_reg;

ID/EX_Mem_write = 0;

ID/EX_Reg_write = 0;

}

stall동작의 pseudo code.

3.Implementation

동작과 설계 과정은 앞서 Design 내용에서 설명하였으므로 코드 위주로 간략하게 설명하겠다.

```
// ---------- Update program coun
// PC must be updated on the risir
PC pc(
  .reset(reset),       // input (Use r
  .clk(clk),         // input
  .next_pc(next_pc),     // input
  .PC_stall(PC_stall),
  .current_pc(curr_pc)   // output
);

//---------- PC+4 adder----------
PC_4_Adder pc_4_adder(
.current_pc(curr_pc),
.next_pc(next_pc)
);
```

IF stage이다.

control flow가 없으므로 PC+4 adder가 언제나 PC를 4씩 증가시킨다. 단, stall신호가 들어오면 PC Update가 1클럭동안 멈춰서 1cycle stall하게 된다.

```
always @(posedge clk) begin
  if (reset) begin
     IF_ID_IR <= 0;
  end
  else begin

  if (IF_ID_stall == 1)
     IF_ID_IR <= IF_ID_IR;
  else
     IF_ID_IR <= imem_to_IR;
  end
end
```

IF/ID stage Pipeline Register이다.

IF staged에서 받아온 instruction이 저장된다.

단, stall신호가 들어오면 IR update가 1cycle 멈추고

IR stage를 stall 시킨다.

```
HazardDetectionUnit hazrard_detection_unit(
  .ID_EX_mem_read(ID_EX_mem_read),
  .ID_EX_rd_field(ID_EX_rd),
  .IF_ID_rs1_field(ecall_rs_1),
  .IF_ID_rs2_field(IF_ID_IR[24:20]),
  .EX_MEM_rd_field(EX_MEM_rd),
  .MEM_WB_rd_field(MEM_WB_rd),
  .is_ecall(is_ecall_con),
  .PC_stall(PC_stall),
  .IF_ID__stall(IF_ID_stall),
  .control_unit_stall(control_unit_stall)
);

// ---------- Register File ----------
  MUX_ECALL ecall_mux(
  .mux_in1(IF_ID_IR[19:15]),
  .mux_in2(ecall_x_17),
  .select_signal(is_ecall_con),
  .mux_out(ecall_rs_1)
);
```

여기 서부터는 ID stage이다.

Hazard Detection Unit이 Data Hazard 여부를 판별하여 stall신호를 내보낸다.

앞서 Design 단락에서 설명한 두 가지 stall logic중 평소에는 stall with forwarding logic을 사용하다가,

is_ecall 신호가 들어오면

ecall stall logic을 골라서 사용한다.

```
ControlUnit ctrl_unit (
  .part_of_inst(IF_ID_IR[6:0]),      // input
  .mem_read(mem_read_con),      // output
  .mem_to_reg(mem_to_reg_con),    // output
  .mem_write(mem_write_con),     // output
  .alu_src(alu_src_con),       // output
  .write_enable(write_enable_con),  // output
  .pc_to_reg(),     // output <= don't need it w/o cont flow so it's left empty
  .alu_op(alu_op_con),        // output 2bit
  .is_ecall(is_ecall_con)       // output (ecall inst)
);

// ---------- Immediate Generator ----------
ImmediateGenerator imm_gen(
  .part_of_inst(IF_ID_IR),          // input
  .imm_gen_out(imm_gen_out)    // output
);

assign id_ex_halt_signal = (is_ecall_con && (rs1_data == 'd10));
```

```
RegisterFile reg_file (
  .reset (reset),    // input
  .clk (clk),          // input
  .rs1 (ecall_rs_1),         // input
  .rs2 (IF_ID_IR[24:20]),         // input
  .rd (MEM_WB_rd),          // input
  .rd_din (WB_data),       // input
  .write_enable (MEM_WB_reg_write),    // input
  .rs1_dout (rs1_data),     // output
  .rs2_dout (rs2_data)      // output
);
```

Control Unit이 generate한 control signal들은 Pipeline Register를 따라 전달되며 앞으로 필요한 stage에서 쓰이게 된다.

ECALL MUX는 is_ecall신호가 들어오면 RF로 들어가는 rs1값을 x17로 변조하여, x17값이 얼마인지 확인할 수 있게 해 주는 역할을 한다. 이때 x17 == 10 이라면, halt signal이 generate 되어 pipeline을 따라 전해지게 된다.

```
  else begin
    ID_EX_mem_read <= mem_read_con;
    ID_EX_mem_to_reg <= mem_to_reg_con;

    ID_EX_rs1_data <= rs1_data;
    ID_EX_rs2_data <= rs2_data;

    if (control_unit_stall == 1) begin
      ID_EX_mem_write <= 0;
      ID_EX_reg_write <= 0;
    end
    else begin
      ID_EX_mem_write <= mem_write_con;
      ID_EX_reg_write <= write_enable_con;
    end

    ID_EX_alu_src <= alu_src_con;
    ID_EX_alu_op <= alu_op_con;
    ID_EX_alu_op <= alu_op_con;

    ID_EX_imm <= imm_gen_out;
    ID_EX_ALU_ctrl_unit_input <= {IF_ID_IR[30], IF_ID_IR[14:12]};

    ID_EX_rs1_field <= IF_ID_IR[19:15];
    ID_EX_rs2_field <= IF_ID_IR[24:20];
    ID_EX_rd <= IF_ID_IR[11:7];

    ID_EX_halted <= id_ex_halt_signal;
```

ID/EX stage Pipeline Register이다.

Imm, rs1,rs2 외에도 이후 stage에서 쓰일 signal등의 값들이 매 클럭마다 업데이트 된다.

```
ForwardingUnit forwarding_unit(
  .ID_EX_rs1_field(ID_EX_rs1_field),
  .ID_EX_rs2_field(ID_EX_rs2_field),
  .EX_MEM_rd_field(EX_MEM_rd),
  .MEM_WB_rd_field(MEM_WB_rd),
  .EX_MEM_reg_write(EX_MEM_reg_write),
  .MEM_WB_reg_write(MEM_WB_reg_write),
  .forwarding_unit_to_mux_A(forwarding_unit_to_mux_A),
  .forwarding_unit_to_mux_B(forwarding_unit_to_mux_B)
);
```

여기서부터는 EX stage이다.

```
Mux4_1 alu_forward_mux_A(
  .mux_in1(ID_EX_rs1_data),
  .mux_in2(EX_MEM_alu_out),
  .mux_in3(WB_data),
  .mux_in4(),
  .select_signal(forwarding_unit_to_mux_A),
  .mux_out(mux_A_to_ALU)
);

Mux4_1 alu_forward_mux_B(
  .mux_in1(ID_EX_rs2_data),
  .mux_in2(EX_MEM_alu_out),
  .mux_in3(WB_data),
  .mux_in4(),
  .select_signal(forwarding_unit_to_mux_B),
  .mux_out(mux_B_to_alu_src_mux)
);
```

Forwarding unit은 앞서 Design 단락에서 설명한 Forwarding logic에 따라,

EX stage에서 사용중인 rs1,rs2와 MEM,WB stage의 rd 값을 비교하고, 실사용 여부 등을 고려하여 forwarding 여부를 결정하고,

그에 맞도록 두 개의 forwarding mux를 작동시킨다.

```
//ALU source MUX
MUX_2_1 ALU_src(
.mux_in1(mux_B_to_alu_src_mux),
.mux_in2(ID_EX_imm),
.select_signal(ID_EX_alu_src),
.mux_out(srcmux_to_alu)
);


// ---------- ALU ----------
ALU alu (
  .alu_op(alu_op),      // input
  .alu_in_1(mux_A_to_ALU),    // input
  .alu_in_2(srcmux_to_alu),    // input
  .alu_result(alu_result),  // output
  .alu_bcond()     // output don't need w/o con flow
);
```

ALU와 ALU 2번 source의 mux가 하는 역할은 과거 과제들과 사실상 동일하다.

차이점이라면, Control unit으로부터 직접 signal을 받는 게 아닌, ID/EX stage Pipeline Register로부터 필요한 signal 들을 가져와서 사용한다.

```
 else begin
      MEM_WB_mem_to_reg <= EX_MEM_mem_to_reg;
      MEM_WB_reg_write <= EX_MEM_reg_write;

      MEM_WB_alu_out <= EX_MEM_alu_out;
      MEM_WB_read_data <= mem_read_data;
      MEM_WB_rd <= EX_MEM_rd;

      MEM_WB_halted <= EX_MEM_halted;
 end
```

EX/MEM stage Pipeline Register 이다.

MEM stage에서 필요한 값들이 업데이트 된다.

```
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_read_data)         // output
);
```

MEM stage에서는 EX/MEM stage Pipeline Register로부터 mem_read및 mem_write signal을 받아와서 필요한 Load/Store동작을 수행한다.

```
else begin
     MEM_WB_mem_to_reg <= EX_MEM_mem_to_reg;
     MEM_WB_reg_write <= EX_MEM_reg_write;

     MEM_WB_alu_out <= EX_MEM_alu_out;
     MEM_WB_read_data <= mem_read_data;
     MEM_WB_rd <= EX_MEM_rd;

     MEM_WB_halted <= EX_MEM_halted;
end
```

MEM/WB stage Pipeline Register 이다.

WB stage에서 필요한 데이터가 업데이트 된다.

```
RegisterFile reg_file (
  .reset (reset),     // input
  .clk (clk),         // input
  .rs1 (ecall_rs_1),          // input
  .rs2 (IF_ID_IR[24:20]),          // input
  .rd (MEM_WB_rd),            // input
  .rd_din (WB_data),        // input
  .write_enable (MEM_WB_reg_write),     // input
  .rs1_dout (rs1_data),      // output
  .rs2_dout (rs2_data)       // output
);
```

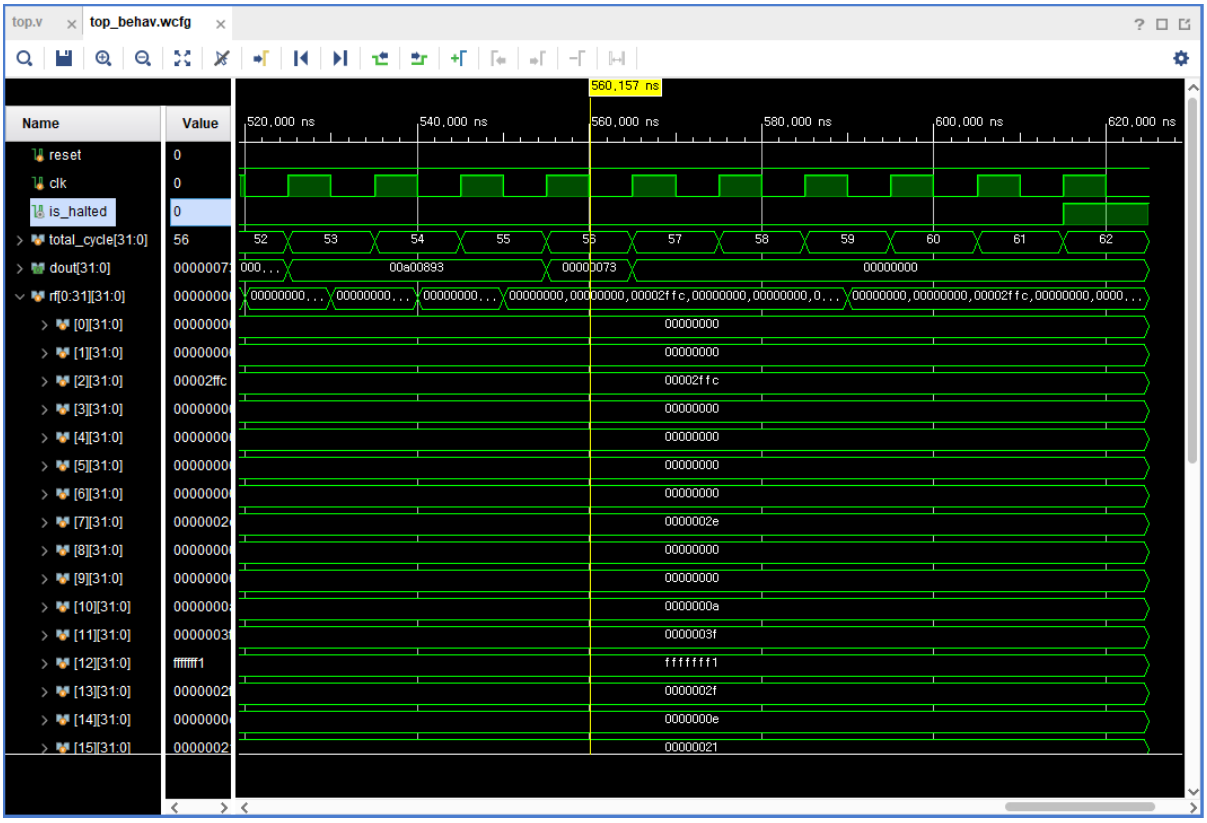
WB stage에서는 MEM/WB stage Pipeline Register에 업데이트된 signal을 바탕으로, 어떤 정보를 rd에 write back할 지 정한다.

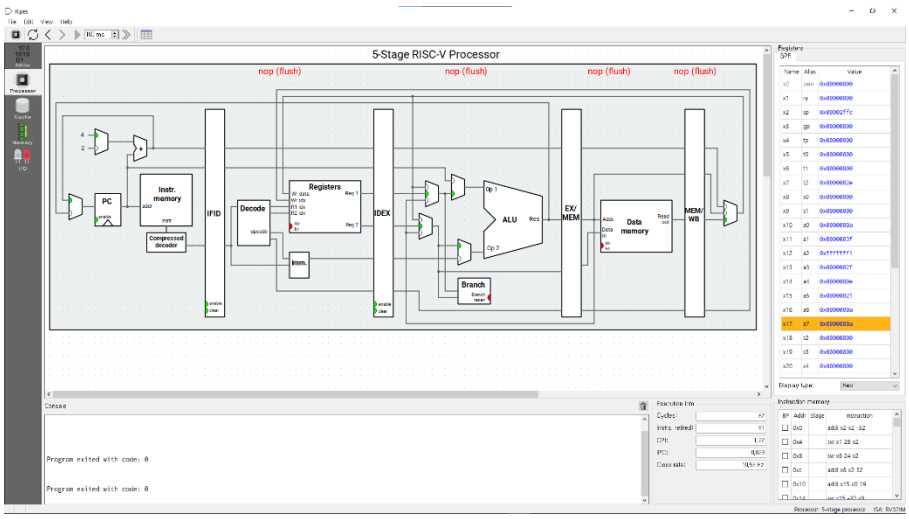
```
assign is_halted = MEM_WB_halted;
```

동시에 여기까지 halt신호가 전해졌다는 건 이전의 모든 instruction이 실행 종료 되었다는 뜻이므로, CPU가 동작을 종료한다.

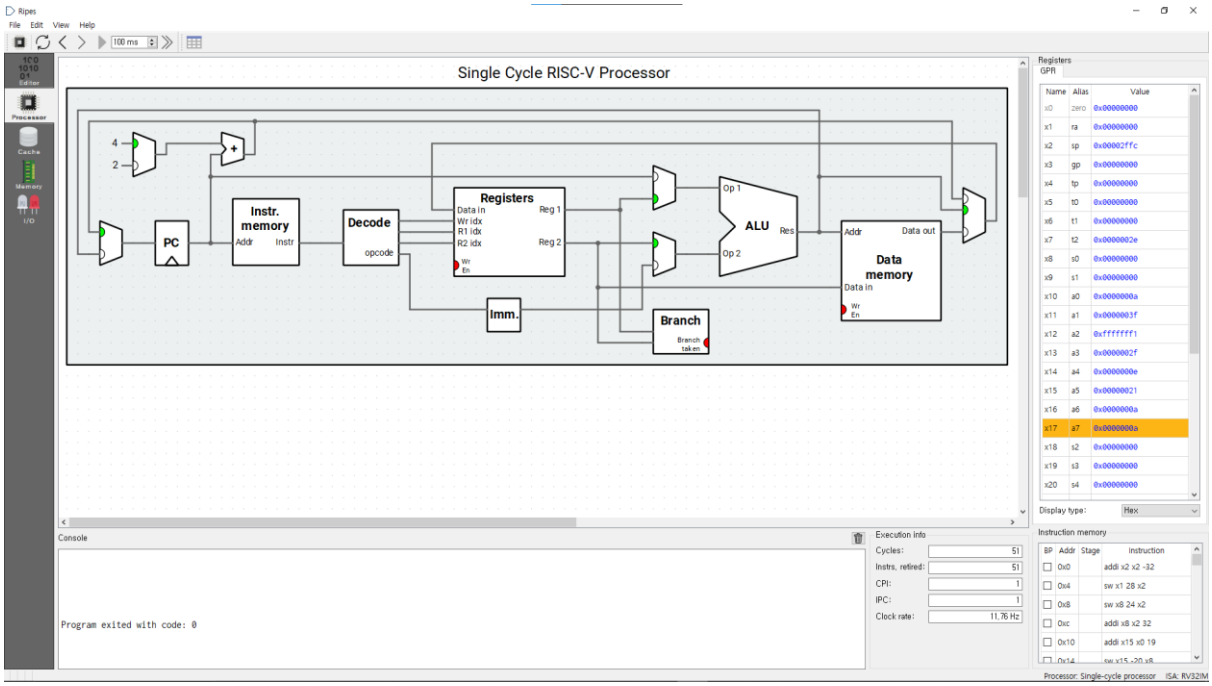
4. Discussion

Compare total cycles between the single cycle and pipelined CPU





우리가 설계한 RISC-V Five Stage Pipeline CPU는 “non_control_flow_mem.txt”를 실행하는 데에 62 사이클이 걸렸으며, 이는 Ripes 시뮬레이션 결과 와도 동일하다.



Single Cycle CPU로 동일 코드를 실행하면 52사이클이 걸린다.

10 사이클 차이가 나는 것은 다음과 같이 해석 가능하다:

11: lw x15 -20 x8

12: add x15 x14 x15 data hazard로 1stall

29: ecall 이 2stall발생시킴

50: ecall 이 2stall발생시킴

52: ecall 이 2stall발생시킴

halt signal flush 3cycle

= + 10 cycle

5. Conclusion

CPU코드를 처음 작성하고 디버깅을 하던 도중, forwarding이 정상 작동 중임에도 전혀 이상한 값이 레지스터에 저장되는 현상이 계속 발생했다.

원인을 찾아보니, Forwarding unit이 rs가 x0 임에도 불구하고 forwarding해 버려서 0이 아닌 전혀 다른 값이 ALU에 들어간 것이었다. 비로소 왜 강의자료에서 forwarding logic을 배울 때 use_rs1(ID)같은 helper function이 등장했는지 알 수 있었다.