CSED311: Lab4-2 Pipelined CPU w/ Control flow instructions

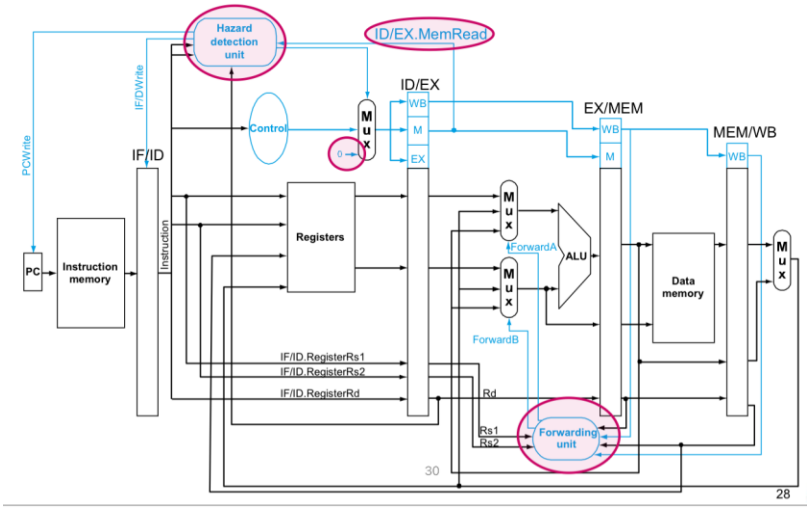
컴퓨터공학과 20220302 김지현

컴퓨터공학과 20220455 윤수인

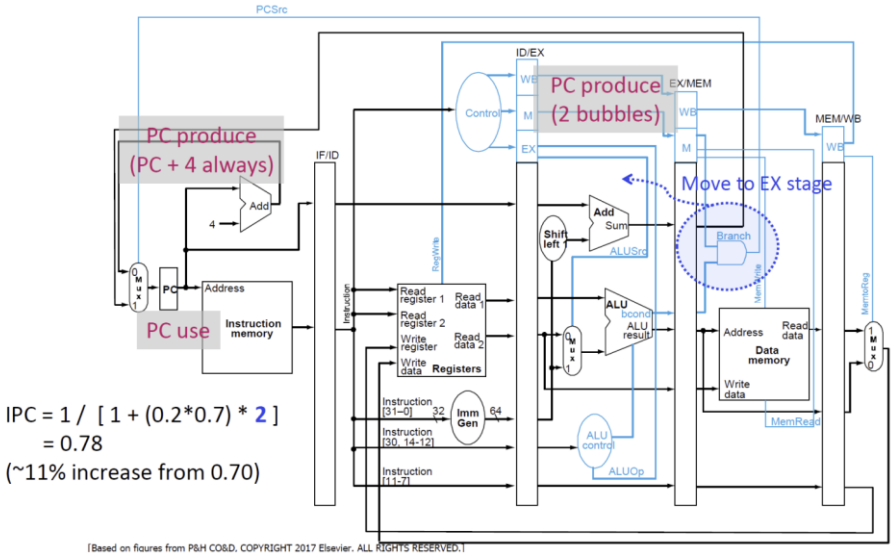
1. Introduction

- 본 과제는 Pipelined CPU 의 구조를 이해하고, Verilog 를 사용하여 control flows 를 포함한 Pipelined CPU 를 설계하고 구현하는 것을 목적으로 한다.
- Verilator 환경에서 실행하며, 레지스터 값과 사이클 수를 관찰한다.
- 이전 Lab 에서 설명했다시피, 하나의 instruction 을 수행하는 데에는 5 가지 stage(IF, ID, EX, MEM, WB)를 필요로 하며, Pipelined CPU 에서는 각 stage 별로 한 사이클이 걸리게 하고, 명령어 수행 과정을 여러 단계로 나누어 병렬로 처리한다. 각 단계는 독립적으로 수행되며, 다음 명령어가 이전 명령어의 결과를 기다리지 않고 실행될 수 있다.
- Control flow 를 고려하지 않았던 Lab 4-1 과 달리, 본 과제에서는 JAL, JALR 등 control flow instructions 를 고려하며, Branch prediction 을 사용하여 pipeline 이 stall 되는 경우를 최소화한다.

2. Design



<그림 1: Pipelined CPU (with data forwarding)>



<그림 2: Pipelined CPU (Branch is moved to EX stage>

Lab4-1 에서는 <그림 1>의 회로도를 참고하여, 한 stage 에서 다음 stage 로 넘겨줘야 하는 값들을 pipeline register 에 저장하여 다음 명령어가 이전 명령어의 결과를 기다리지 않고 실행될 수 있도록 Pipelined CPU 의 기본 구조를 구현하였다.

추가로, <그림 2>에서와 같이 Branch 가 EX stage 에서 resolve 되도록 하여 branch misprediction 이 2 개의 bubble 만 발생시키도록 하였다.

구현과 관련하여 Lab 4-1 과 달라진 점은 Implementation 파트에서 설명한다.

3. Implementation

- **pc.v**

```
1   module pc(input reset,
2   input clk,
3   input PCWrite,
4   input [31:0] next_pc,
5   output reg [31:0] current_pc);
6
7       always @(posedge clk) begin
8           if(reset) begin
9               current_pc <= 0;
10          end
11          else begin
12              if(PCWrite) begin
13                  current_pc <= next_pc;
14              end
15          end
16      end
17
18  endmodule
```

<그림 3 : pc.v 모듈의 구성>

Lab 4-1 과 동일하다. pc.v 모듈에서는 reset 이 1 인 경우 current_pc 의 값을 0 으로 업데이트하고, reset 이 1 이 아닌 경우에는 PCWrite 의 값에 따라 current_pc 값이 달라진다. PCWrite 이 1 인 경우, 다음 instruction 을 읽어와야 하므로 current_pc 의 값을 next_pc 의 값으로 업데이트한다. PCWrite 이 0 인 경우, hazard 등의 상황에 의해 다음 instruction 을 읽어오지 말아야 할 상황이므로, current_pc 값을 업데이트하지 않는다. 이 과정은 clock synchronous 하게 이루어진다.

- **adder.v**

```
module adder (
    input[31:0] in1,
    input [31:0] in2,
    output reg [31:0] out);

    assign out = in1 + in2;

endmodule
```

<그림 4: adder.v 모듈의 구성>

Lab 4-1 과 동일하다. 두 개의 32 bit input 레지스터 값을 합하여 out 에 할당한다. 이번 과제에서는 non-control flow 뿐만 아니라 control flow 를 고려하므로, PC + 4 및 PC + immediate 등의 연산을 처리할 때만 사용된다. 이 과정은 clock asynchronous 하게 이루어진다.

- **InstMemory.v**

```
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 = {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)
        // DO NOT TOUCH COMMENT BELOW
        /* verilator lint_off BLKSEQ */
        mem[i] = 32'b0;
        /* verilator lint_on BLKSEQ */
        // DO NOT TOUCH COMMENT ABOVE
      // Provide path of the file including instructions with binary format
      $readmemh("./student_tb/non-controlflow_mem.txt", mem);
    end
  end

endmodule
```

<그림 5: InstMemory.v 모듈의 구성>

Lab 4-1 과 동일하다. InstMemory.v 모듈은 reset 이 1 인 경우 clock synchronous 하게 instruction memory 를 초기화한다. Reset 이 1 이 아닌 경우, asynchronous 하게 instruction 주소를 받아와 메모리에 저장된 instruction 을 dout 에 저장한다.

- **DataMemory.v**

```
module DataMemory #(parameter MEM_DEPTH = 16384) (input reset,
                                                   input clk,
                                                   input [31:0] addr,    // address of the data memory
                                                   input [31:0] din,     // data to be written
                                                   input mem_read,       // is read signal driven?
                                                   input mem_write,      // is write signal driven?
                                                   output [31:0] dout);  // output of the data memory at addr
  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 = {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 (reset) begin
      for (i = 0; i < MEM_DEPTH; i = i + 1)
        // DO NOT TOUCH COMMENT BELOW
        /* verilator lint_off BLKSEQ */
        mem[i] = 32'b0;
        /* verilator lint_on BLKSEQ */
        // DO NOT TOUCH COMMENT ABOVE
    end

    else begin
      if (mem_write)
        mem[dmem_addr] <= din;
    end
  end
endmodule
```

<그림 6: DataMemory.v 모듈의 구성>

Lab 4-1 과 동일하다. DataMemory.v 모듈은 reset 이 1 인 경우 clock asynchronous 하게 data memory 를 초기화한다. Reset 이 1 이 아닌 경우, mem_read 가 1 일 때는 asynchronous 하게 dmem_addr 위치에 있는 데이터 값을 읽어 dout 에 저장한다. mem_write 이 1 일 때는 synchronous 하게 din 을 dmem_addr 위치에 저장한다

- **RegisterFile.v**

```
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 [31:0] print_reg[0:31]);  // output of rs 2
  integer i;
  // Register file
  reg [31:0] rf[0:31];
  assign print_reg = rf;
  // Asynchronously read register file
  assign rs1_dout = rf[rs1];
  assign rs2_dout = rf[rs2];

  always @(clk) begin
    if (clk==0) begin // negative edge
      if (write_enable & (rd != 0))
        rf[rd] <= rd_din;
    end
    else begin // positive edge
      if (reset) begin
        rf[0] <= 32'b0;
        rf[1] <= 32'b0;
        rf[2] <= 32'h2ffc; // stack pointer
        for (i = 3; i < 32; i = i + 1)
          rf[i] <= 32'b0;
      end
    end
  end
endmodule
```

<그림 7: RegisterFile.v 모듈의 구성>

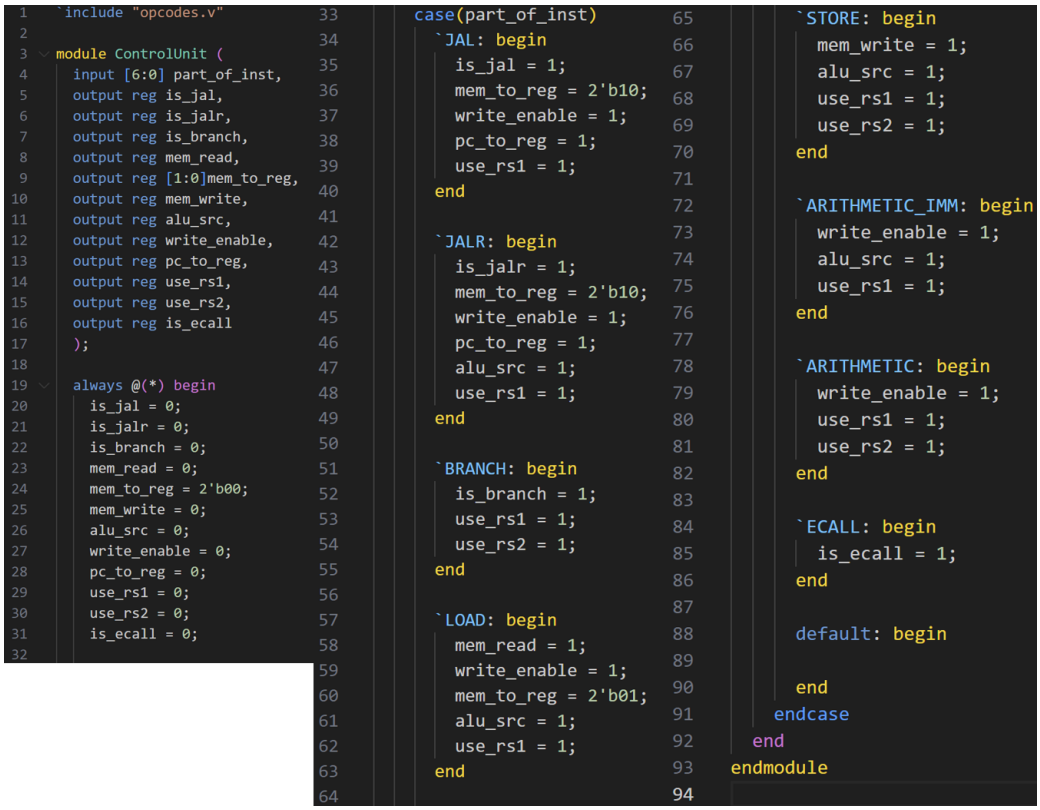
Lab 4-1 과 동일하다. RegisterFile.v 모듈은 register 의 위치를 받아와 해당 register 의 값을 읽거나 쓰는 역할을 한다. Register 값을 읽는 과정은 clock asynchronous 하고, 5-bit input 인 rs1 과 rs2 각각에 해당하는 register 의 값을 32-bit output 인 rs1_dout, rs2_dout 에 각각 저장한다.

Register 값을 쓰는 과정은 clock synchronous 하다. 이번 과제에서는 Lab 4-1 과 마찬가지로 clk negative edge 에서 업데이트가 먼저 이루어지기 때문에, 동일한 레지스터를 같은 주기로 읽고 쓰더라도 internal forwarding 이 필요하지 않다.

따라서, clk 가 0 이고 write_enable 시그널이 1 이고 destination register(rd)가 존재할 때 32-bit input data 인 rd_din 을 rd 위치에 해당하는 register 에 저장한다.

Clk 가 0 이 아니고 Reset signal 이 1 일 때는 clock synchronous 하게 모든 register 를 0 으로 저장한 뒤 stack pointer 인 rf[2]를 2ffc 로 할당한다.

- **ControlUnit.v**



```
`include "opcodes.v"

module ControlUnit (
  input [6:0] part_of_inst,
  output reg is_jal,
  output reg is_jalr,
  output reg is_branch,
  output reg mem_read,
  output reg [1:0]mem_to_reg,
  output reg mem_write,
  output reg alu_src,
  output reg write_enable,
  output reg pc_to_reg,
  output reg use_rs1,
  output reg use_rs2,
  output reg is_ecall
  );

  always @(*) begin
    is_jal = 0;
    is_jalr = 0;
    is_branch = 0;
    mem_read = 0;
    mem_to_reg = 2'b00;
    mem_write = 0;
    alu_src = 0;
    write_enable = 0;
    pc_to_reg = 0;
    use_rs1 = 0;
    use_rs2 = 0;
    is_ecall = 0;

    case(part_of_inst)
      `JAL: begin
        is_jal = 1;
        mem_to_reg = 2'b10;
        write_enable = 1;
        pc_to_reg = 1;
        use_rs1 = 1;
      end

      `JALR: begin
        is_jalr = 1;
        mem_to_reg = 2'b10;
        write_enable = 1;
        pc_to_reg = 1;
        alu_src = 1;
        use_rs1 = 1;
      end

      `BRANCH: begin
        is_branch = 1;
        use_rs1 = 1;
        use_rs2 = 1;
      end

      `LOAD: begin
        mem_read = 1;
        write_enable = 1;
        mem_to_reg = 2'b01;
        alu_src = 1;
        use_rs1 = 1;
      end

      `STORE: begin
        mem_write = 1;
        alu_src = 1;
        use_rs1 = 1;
        use_rs2 = 1;
      end

      `ARITHMETIC_IMM: begin
        write_enable = 1;
        alu_src = 1;
        use_rs1 = 1;
      end

      `ARITHMETIC: begin
        write_enable = 1;
        use_rs1 = 1;
        use_rs2 = 1;
      end

      `ECALL: begin
        is_ecall = 1;
      end

      default: begin

      end
    endcase
  end
endmodule

```

<그림 8: ControlUnit.v 모듈의 구성>

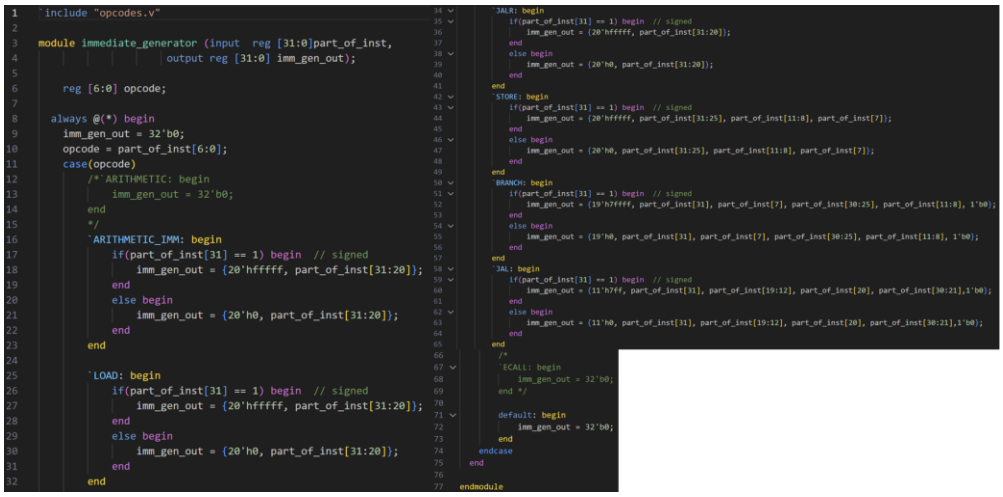
ControlUnit.v 모듈은 기본적으로 Single-cycle CPU 에서와 동일하게 part_of_inst 값에 따라 시그널들의 값을 활성화하거나 비활성화하는 구조로 동작한다. Lab 4-2 의 경우 control flow 도 고려하므로, Lab4-1 에서 주석 처리하였던 JAL, JALR, BRANCH 관련 부분을 다시 사용하였다.

Jal, jalr, branch 인스트럭션의 경우 각각 is_jal, is_jalr, 그리고 is_branch 시그널을 1 로 설정하였다. 이 값은 이후 EX stage 에서 해당 pipeline register 에 값을 넘겨준 후, cpu 에서 2-bit reg 인 pc_src 값을 결정하는 데에 사용된다. Pc_src 값을 정하는 과정은 추후 자세히 설명한다.

또한, hazard 처리를 위해 각 연산에서의 rs1, rs2 사용 여부(use_rs1, use_rs2)를 연산에 따라 활성화/비활성화하였다.

이 과정은 **clock asynchronous** 하게 진행된다.

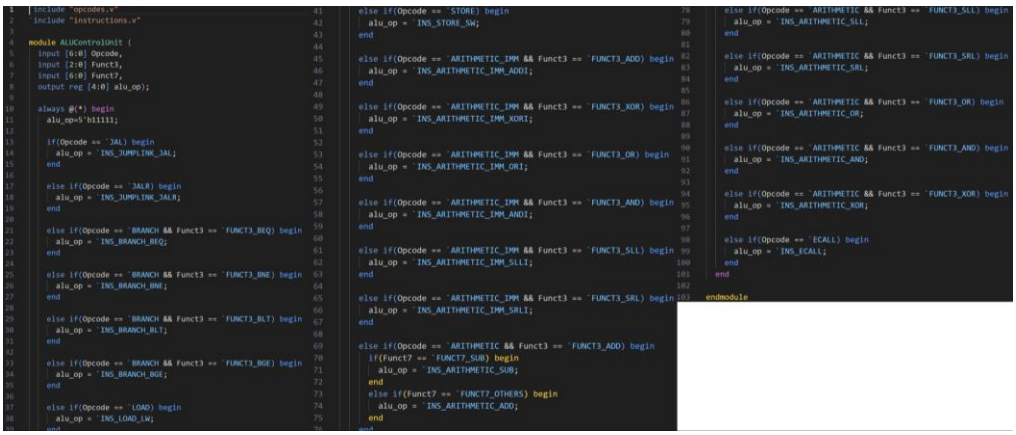
- **ImmediateGenerator.v**



<그림 9: ImmediateGenerator.v 모듈의 구성>

Lab4-1 과 동일하다. ImmediateGenerator.v 모듈은 32-bit 의 part_of_inst 를 input 으로 받아와 하위 7 bits 를 opcode 에 저장한다. Opcode 에 따라 적절한 immediate value 를 imm_gen_out 에 저장한다. Signed 연산 여부는 part_of_inst 의 상위 1 bit 로 판별한다. 전체 과정은 clock asynchronous 하게 이루어진다.

- **ALUControlUnit.v**



<그림 10: ALUControlUnit.v 모듈의 구성>

Lab 4-1 과 동일하다. ALUControlUnit.v 모듈은 opcode, funct3, funct7 에 따라 alu 가 수행해야 할 연산을 결정하는 모듈로, Single-cycle CPU 에서 작성한 코드를 사용하였다.

Opcode, funct3, funct7 은 각각 ID_EX_inst 의 [6:0], [14:12], [31:25] 부분에 해당하며, 각각의 값에 따라 alu_op 값을 지정한다.

이 과정은 clock asynchronous 하게 이루어진다.

- **alu.v**

```
`include "instructions.v"

module alu (
  input [4:0]  alu_op,      // input
  input [31:0] alu_in_1,    // input
  input [31:0] alu_in_2,    // input
  output reg [31:0] alu_result,  // output
  output reg alu_bcond);     // output

  always @(*) begin
    alu_result = 0;
    alu_bcond = 0;

    case(alu_op)
      `INS_JUMPLINK_JALR, `INS_LOAD_LW, `INS_STORE_SW, `INS_ARITHMETIC_IMM_ADDI, `INS_ARITHMETIC_ADD: begin
        alu_result = alu_in_1 + alu_in_2;
      end

      `INS_BRANCH_BEQ: begin
        alu_bcond = (alu_in_1 == alu_in_2) ? 1 : 0;
      end

      `INS_BRANCH_BNE: begin
        alu_bcond = (alu_in_1 != alu_in_2) ? 1 : 0;
      end

      `INS_BRANCH_BLT: begin
        alu_bcond = (alu_in_1 < alu_in_2) ? 1 : 0;
      end

      `INS_BRANCH_BGE: begin
        alu_bcond = (alu_in_1 >= alu_in_2) ? 1 : 0;
      end

      `INS_ARITHMETIC_IMM_XORI, `INS_ARITHMETIC_XOR: begin
        alu_result = alu_in_1 ^ alu_in_2;
      end
        `INS_ARITHMETIC_IMM_ORI, `INS_ARITHMETIC_OR: begin
          alu_result = alu_in_1 | alu_in_2;
        end

        `INS_ARITHMETIC_IMM_ANDI, `INS_ARITHMETIC_AND: begin
          alu_result = alu_in_1 & alu_in_2;
        end

        `INS_ARITHMETIC_IMM_SLLI, `INS_ARITHMETIC_SLL: begin
          alu_result = alu_in_1 << alu_in_2;
        end

        `INS_ARITHMETIC_IMM_SRLI, `INS_ARITHMETIC_SRL: begin
          alu_result = alu_in_1 >> alu_in_2;
        end

        `INS_ARITHMETIC_SUB: begin
          alu_result = alu_in_1 - alu_in_2;
        end


        default: begin

        end
      endcase
    end
  endmodule
```

<그림 11: alu.v 모듈의 구성>

Lab 4-1 과 동일하다. alu.v 모듈은 앞서 ALUControlUnit.v 모듈에서 지정한 alu_op 에 따라 알맞은 연산을 수행하여 연산 결과를 alu_result 에 저장하고, alu_bcond 를 활성화/비활성화한다. 이 과정은 clock asynchronous 하게 이루어진다.

- **ForwardingEcallUnit.v**

```
module ForwardingEcallUnit(
    input [4:0] rs1,
    input [4:0] ID_EX_rd,
    input [4:0] EX_MEM_rd,
    input [4:0] MEM_WB_rd,
    input ID_EX_reg_write,
    input EX_MEM_reg_write,
    input MEM_WB_reg_write,
    output reg [1:0] forwardEcall
    );

    always @(*) begin
        if((rs1 == ID_EX_rd) && ID_EX_reg_write) begin
            forwardEcall = 2'b01;
        end
        else if((rs1 == EX_MEM_rd) && EX_MEM_reg_write) begin
            forwardEcall = 2'b10;
        end
        else if((rs1 == MEM_WB_rd) && MEM_WB_reg_write) begin
            forwardEcall = 2'b11;
        end
        else begin
            forwardEcall = 2'b00;
        end
    end

endmodule

```

<그림 12: ForwardingEcallUnit.v 모듈의 구성>

Lab 4-1 과 동일하다. ForwardingEcallUnit.v 모듈은 EX, MEM, WB stage 에서 사용되는 rd 값이 17 이고 reg_write 이 1 인 경우, ecall_data 를 결정할 mux 의 input 으로 사용되는 forwardEcall 시그널 값을 결정한다. 이 과정은 clock asynchronous 하게 이루어진다.

- **MUX2x1.v**

```
module MUX2x1(input in_bit,
input [31:0] in1,
input [31:0] in0,
output [31:0] out);

    assign out = in_bit ? in1 : in0;

endmodule
```

<그림 13: MUX2x1.v 모듈의 구성>

Lab 4-1 과 동일하다. MUX2x1.v 모듈은 in_bit 의 값에 따라 in1 과 in2 중 하나의 값을 반환하는 역할로, 전부 cpu.v 에서 사용되었다. 필요에 따라 5 bit input, output 을 다루는 MUX2x1_5bit 모듈을 추가로 만들어 사용하였다. MUX2x1 이 사용된 부분은 다음과 같다.

- mux_handle_ecall: MUX2x1_5bit 모듈을 사용하였으며, is_ecall 시그널의 값에 따라 rs1 의 값을 17 또는 기존 rs1 값으로 결정한다. Output 인 rs1_out 을 RegisterFile 의 input 으로 사용한다.
- mux_ALUSrc: EX stage 에서 alu 의 두 번째 input 을 결정하기 위한 mux 로, ID_EX_alu_src 의 값에 따라 data forwarding 된 rs2 데이터 값 또는 immediate 값을 alu 의 두 번째 input 으로 결정한다.

이 과정은 clock asynchronous 하게 이루어진다.

- **ForwardingUnit.v**

```
`include "opcodes.v"

`define MEM 2'b01
`define WB 2'b10

module ForwardingUnit(
    input [4:0] ID_EX_rs1,
    input [4:0] ID_EX_rs2,
    input [4:0] EX_MEM_rd,
    input EX_MEM_reg_write,
    input [4:0] MEM_WB_rd,
    input MEM_WB_reg_write,
    input [4:0] _x0,
    output reg [1:0] ForwardA,
    output reg [1:0] ForwardB);

    always @(*) begin
        if((ID_EX_rs1 != _x0) && (ID_EX_rs1 == EX_MEM_rd) && EX_MEM_reg_write) begin
            ForwardA = `MEM;
        end
        else if((ID_EX_rs1 != _x0) && (ID_EX_rs1 == MEM_WB_rd) && MEM_WB_reg_write) begin
            ForwardA = `WB;
        end
        else begin
            ForwardA = 2'b00;
        end

        if((ID_EX_rs2 != _x0) && (ID_EX_rs2 == EX_MEM_rd) && EX_MEM_reg_write) begin
            ForwardB = `MEM;
        end
        else if((ID_EX_rs2 != _x0) && (ID_EX_rs2 == MEM_WB_rd) && MEM_WB_reg_write) begin
            ForwardB = `WB;
        end
        else begin
            ForwardB = 2'b00;
        end
    end

endmodule
```

<그림 14: ForwardingUnit.v 모듈의 구성>

Lab 4-1 과 동일하다. ForwardingUnit.v 모듈은 rs1, rs2 가 어떤 stage 에서 forwarding 될지 결정하는 모듈이다.

1) EX stage 에서 사용될 rs1 값이 0 이 아니며, 해당 rs1 이 MEM stage 에서 사용될 rd 값과 같고, reg_write 이 1 일 경우: 두 instruction 사이의 거리가 1 이므로 MEM stage 에서 forwarding 이 이루어진다. 따라서 ForwardA 를 01 로 할당한다.
2) EX stage 에서 사용될 rs1 값이 0 이 아니며, 해당 rs1 이 WB stage 에서 사용될 rd 값과 같고, reg_write 이 1 일 경우: 두 instruction 사이의 거리가 2 이므로 WB stage 에서 forwarding 이 이루어진다. 따라서 ForwardA 를 10 으로 할당한다.
3) 그 외의 경우: 두 instruction 사이의 거리가 3 이상이므로 forwarding 을 할 필요가 없다. 따라서 ForwardA 를 00 으로 설정한다.

Rs2 의 경우도 동일하게 적용한다.

이 과정은 clock asynchronous 하게 이루어진다.

- **MUX4x1.v**

```
module MUX4x1(input [1:0] in_bit,
    input [31:0] in3,
    input [31:0] in2,
    input [31:0] in1,
    input [31:0] in0,
    output [31:0] out);

    assign out = in_bit[1] ? (in_bit[0] ? in3 : in2) : (in_bit[0] ? in1 : in0);
endmodule
```

<그림 15: MUX4x1.v 모듈의 구성>

Lab 4-1 과 동일하다. MUX4x1 모듈은 in_bit 값에 따라 4 가지 (in0~in3) input 중 하나의 값을 반환하는 역할이다. 이는 Clock asynchronous 하게 이루어진다.

MUX4x1 이 사용된 부분은 다음과 같다.

- MUX_ecall_data: forwarding ecall 과 관련하여, is_halted 값을 결정할 때 사용될 ecall_data 값을 정하는 mux 이다. Forwarding_ecall_unit 에서 결정된 forwardEcall 값에 따라 rd_din, EX_MEM_alu_out, alu_result, rs1_dout 중 어떤 값을 ecall_data 로 사용할지 결정한다. Ecall 여부 결정은 ID stage 에서만 이루어지기 때문에, 위 값들 중 하나를 data forwarding 하여 ecall 여부 결정에 사용한다.
- mux_forward_a: forwarding unit 에서 결정된 ForwardA 값에 따라 alu 의 첫 번째 input 값을 결정하는 mux 이다. ForwardA 값에 따라 0, rd_din, EX_MEM_alu_out, ID_EX_rs1_data 중 어떤 값을 사용할지 결정한다.
- mux_forward_b: forwarding unit 에서 결정된 ForwardB 값에 따라, mux_ALUSrc 의 input 으로 사용될 값을 결정하는 mux 이다. ForwardB 값에 따라 0, rd_din, EX_MEM_alu_out, ID_EX_rs2_data 중 어떤 값을 사용할지 결정한다.
- mux_MemtoReg: 2 bit reg 인 MEM_WB_mem_to_reg 값에 따라 RegisterFile 모듈과 mux_forward_a, mux_forward_b, MUX_ecall_data 의 input 으로 사용될 rd_din 값을 결정한다. MEM_WB_pc_plus_4, MEM_WB_mem_to_reg_src_1, MEM_WB_mem_to_reg_src_2 중 어떤 값을 사용할지 결정한다.
- mux_PCSrc: 2 bit reg 인 pc_src 의 값에 따라 pc_plus_4, pc_plus_imm, alu_result 중 어떤 값을 next_pc 로 사용할지 결정한다. 이때, pc_src 는 아래와 같은 로직으로 결정된다.

```
always @(*) begin
  if(ID_EX_is_jal|(ID_EX_is_branch & bcond)) begin
    pc_src=2'b01;
  end
  else if(ID_EX_is_jalr) begin
    pc_src=2'b10;
  end
  else begin
    pc_src=2'b00;
  end
end
```

<그림 16: pc_src 결정 로직>

```
MUX4x1 mux_PCSrc (
  .in_bit(pc_src),
  .in3(0),
  .in2(alu_result),
  .in1(pc_plus_imm),
  .in0(pc_plus_4),
  .out(next_pc)
);
```

<그림 17: mux_PCSrc>

Jal 과 branch(beq, blt 등) 연산인 경우 pc_src 를 2'b01 로 설정하여 PC + immediate 값을 다음 pc 로 설정하도록 한다. Jalr 연산인 경우 pc_src 를 2'b10 으로 설정하여 alu 연산 결과를 다음 pc 로 설정하도록 한다. 그 외의 경우, pc_src 를 2'b00 으로 설정하여 PC + 4 값을 다음 pc 로 설정하도록 한다. 이 과정은 clock asynchronous 하게 이루어진다.

- **HazardDetectUnit.v**

```
module HazardDetectUnit(
    input [4:0] rs1,
    input [4:0] rs2,
    input [4:0] ID_EX_rd,
    input ID_EX_mem_read,
    input use_rs1,
    input use_rs2,
    output reg PCWrite,
    output reg IF_IDWrite,
    output reg hazard_detected);


    always @(*) begin
        if((((rs1 == ID_EX_rd) && use_rs1)) || (((rs2 == ID_EX_rd) && use_rs2))) && ID_EX_mem_read) begin
            hazard_detected = 1;
            IF_IDWrite = 0;
            PCWrite = 0;
        end
        else begin
            hazard_detected = 0;
            IF_IDWrite = 1;
            PCWrite = 1;
        end
    end

endmodule
```

<그림 18: HazardDetectUnit.v 모듈의 구성>

Lab 4-1 과 동일하다. HazardDetectUnit.v 모듈은 data forwarding 을 하더라도 발생하는 RAW dependence 에 의한 hazard 를 감지하기 위한 유닛이다. 위 코드상의 조건인 경우 stall 이 발생하므로, hazard_detected 를 1 로 설정하고, IF/ID pipeline register 에서 fetch 된 instruction 을 decode 할지 여부를 결정하는 IF_IDWrite 과, PC 를 업데이트할지 결정하는 PCWrite 을 0 으로 설정한다.

이 과정은 clock asynchronous 하게 이루어진다.

- **Branch prediction 구현**

```
assign is_flush = (pc_src == 2'b10) | (pc_src == 2'b01);
```

```
// Update IF/ID pipeline registers here
always @(posedge clk) begin
  if (reset) begin
    IF_ID_inst <= 0;
    IF_ID_current_pc <= 0;
    IF_ID_pc_plus_4 <= 0;
    IF_ID_is_flush <= 0;
  end
  else begin
    if(!hazard_detected) begin
      IF_ID_inst <= inst_out;
      IF_ID_current_pc <= current_pc;
      IF_ID_pc_plus_4 <= pc_plus_4;
      IF_ID_is_flush <= is_flush;
      //$display("%h", IF_ID_inst);
    end
  end
end
```

```
// Update ID/EX pipeline registers here
always @(posedge clk) begin
  if (reset|hazard_detected|is_flush|IF_ID_is_flush) begin
    ID_EX_alu_op <= 0;          // will be used in EX stage
    ID_EX_alu_src <= 0;         // will be used in EX stage
    ID_EX_mem_write <= 0;       // will be used in MEM stage
    ID_EX_mem_read <= 0;        // will be used in MEM stage
    ID_EX_mem_to_reg <= 0;      // will be used in WB stage
    ID_EX_reg_write <= 0;       // will be used in WB stage
    ID_EX_is_halted <= 0;
    ID_EX_pc_plus_4 <= 0;
    //ID_EX_is_jump_needed <= 0;
    ID_EX_is_branch <= 0;
    ID_EX_is_jal <= 0;
    ID_EX_is_jalr <= 0;
    // From others
    ID_EX_rs1_data <= 0;
    ID_EX_rs2_data <= 0;
    ID_EX_imm <= 0;
    ID_EX_rd <= 0;
    ID_EX_current_pc <= 0;
    ID_EX_inst <= 0;
    ID_EX_rs1 <= 0;
    ID_EX_rs2 <= 0;
  end
  else begin
    // From the control unit
    ID_EX_mem_write <= mem_write;      // will be used in WB stage
    ID_EX_reg_write <= reg_write;      // will be used in WB stage
    ID_EX_alu_op <= alu_op;         // will be used in EX stage
    ID_EX_alu_src <= alu_src;        // will be used in EX stage

    ID_EX_mem_read <= mem_read;       // will be used in MEM stage
    ID_EX_mem_to_reg <= mem_to_reg;     // will be used in WB stage

    ID_EX_is_halted <= is_halted_temp;
    ID_EX_pc_plus_4 <= IF_ID_pc_plus_4;
    //ID_EX_is_jump_needed <= is_jump_needed;
    ID_EX_is_branch <= is_branch;
    ID_EX_is_jal <= is_jal;
    ID_EX_is_jalr <= is_jalr;
    // From others
    ID_EX_rs1_data <= rs1_dout;
    ID_EX_rs2_data <= rs2_dout;
    ID_EX_imm <= imm_gen_out;
    ID_EX_rd <= rd;  // IF_ID_inst[11:7]
    ID_EX_current_pc <= IF_ID_current_pc;
    ID_EX_inst <= IF_ID_inst;
    ID_EX_rs1 <= rs1;
    ID_EX_rs2 <= rs2;
  end
```

<그림 19: Branch prediction 과 관련하여 is_flush 의 사용>

Always-not-taken 방식으로 구현하였다. fetch 되어 있는 instruction 을 flush 해야하는지 여부를 나타내는 is_flush wire 를 선언하여 pc_src 가 2'b01 이거나 2'b10 인 경우 값이 1 이 되도록 하였다. 즉, branch, jal, jalr 연산의 경우 is_flush 가 1 이 되도록 하여, 파이프라인 레지스터의 내용을 초기화해야 함을 나타낸다. 이후 IF/ID pipeline registers 를 업데이트 하는 과정에서 IF_ID_is_flush 의 값을 0 또는 is_flush 로 업데이트한다.

ID/EX pipeline registers 를 업데이트할 때, reset, hazard_detected, is_flush, IF_ID_is_flush 조건 중 하나라도 참(true)일 경우 해당 레지스터들을 초기화한다. 이는 분기 또는 점프가 발생했거나, hazard 가 감지되었을 때, 잘못된 데이터가 다음 파이프라인 스테이지로 전달되는 것을 방지한다. 결론적으로, is_flush 와 IF_ID_is_flush 신호는 분기(또는 점프)가 발생했을 때 잘못된 명령어들을 파이프라인에서 제거하는 데 중요한 역할을 한다. 이러한 방식으로, 파이프라인은 분기 명령어(branch, jal, jalr)가 평가되고 나서 새로운 실행 경로로 정확하게 전환할 수 있다.

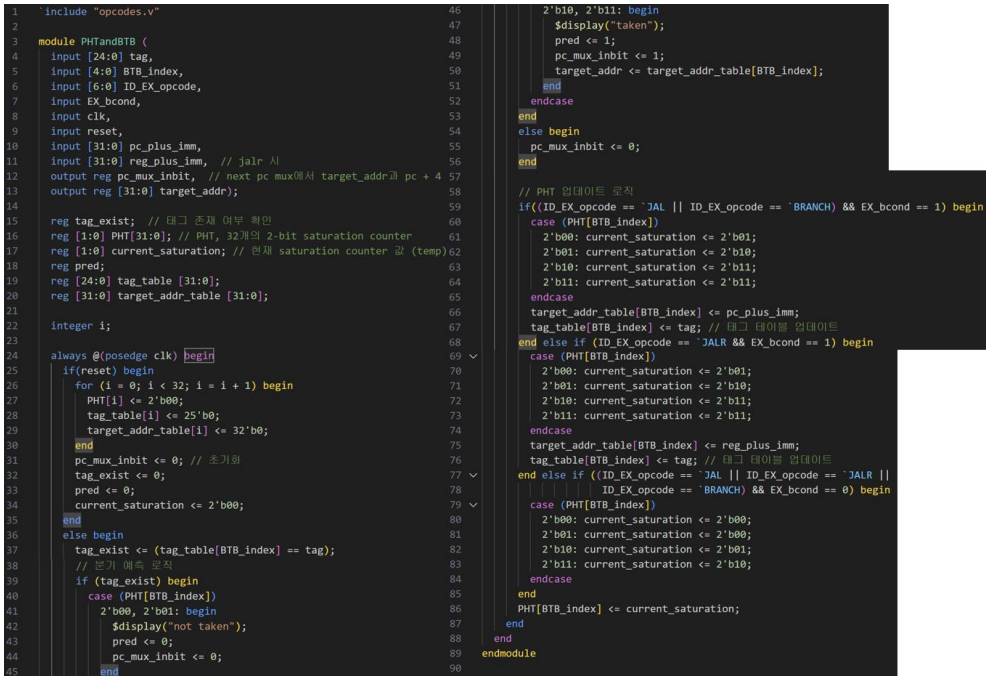
4. Discussion

- 구현 과정에서 Control flow instructions 처리에서의 use_rs1, use_rs2 값을 설정하지 않아 문제가 발생하였다. Hazard detection unit 에서 use_rs1, use_rs2 값을 input 으로 받아 hazard 여부를 판정하는데, branch, jal, jalr 의 경우에 대한 use_rs1, use_rs2 값을 설정하지 않아 control hazard 를 detect 할 수 없었고, 그 결과 loop 시뮬레이션의 결과가 비정상적으로 나타났다. 아래와 같이 use_rs1, use_rs2 값을 적절히 설정하여 문제를 해결할 수 있었다.

```
`JAL: begin
  is_jal = 1;
  mem_to_reg = 2'b10;
  write_enable = 1;
  pc_to_reg = 1;
  use_rs1 = 1;
end

`JALR: begin
  is_jalr = 1;
  mem_to_reg = 2'b10;
  write_enable = 1;
  pc_to_reg = 1;
  alu_src = 1;
  use_rs1 = 1;
end

`BRANCH: begin
  is_branch = 1;
  use_rs1 = 1;
  use_rs2 = 1;
end
```

<그림 20: ControlUnit.v – JAL, JALR, BRANCH 에 대한 처리>

- 초기 구상은 PHT(Pattern History Table)과 BTB(Branch Target Buffer)를 사용한 Branch Prediction 로직을 구현하고자 하였다. 아래와 같이 PHTandBTB 모듈을 만들어 구현하고자 하였으나, 모듈을 적용하는 데에 어려움이 있어 시간 관계상 always not taken 방식을 사용하였다.



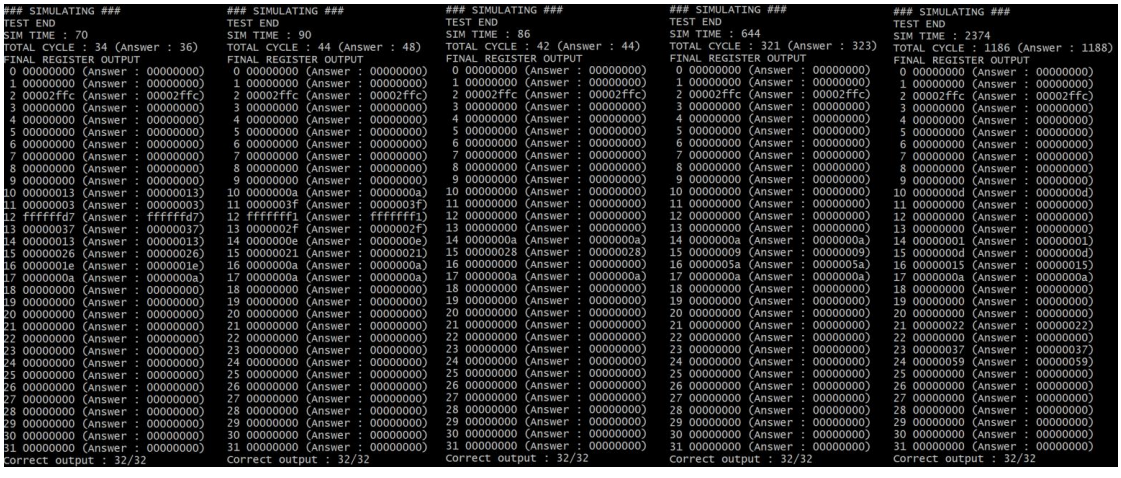
```
`include "opcodes.v"

module PHTandBTB (
  input [24:0] tag,
  input [4:0] BTB_index,
  input [6:0] ID_EX_opcode,
  input EX_bcond,
  input clk,
  input reset,
  input [31:0] pc_plus_imm,
  input [31:0] reg_plus_imm,  // jalr 시
  output reg pc_mux_inbit,  // next pc mux에서 target_addr과 pc + 4
  output reg [31:0] target_addr);

  reg tag_exist;  // 태그 존재 여부 확인
  reg [1:0] PHT[31:0]; // PHT, 32개의 2-bit saturation counter
  reg [1:0] current_saturation; // 현재 saturation counter 값 (temp)
  reg pred;
  reg [24:0] tag_table [31:0];
  reg [31:0] target_addr_table [31:0];

  integer i;

  always @(posedge clk) begin
    if(reset) begin
      for (i = 0; i < 32; i = i + 1) begin
        PHT[i] <= 2'b00;
        tag_table[i] <= 25'b0;
        target_addr_table[i] <= 32'b0;
      end
      pc_mux_inbit <= 0; // 초기화
      tag_exist <= 0;
      pred <= 0;
      current_saturation <= 2'b00;
    end
    else begin
      tag_exist <= (tag_table[BTB_index] == tag);
      // 분기 예측 로직
      if (tag_exist) begin
        case (PHT[BTB_index])
          2'b00, 2'b01: begin
            $display("not taken");
            pred <= 0;
            pc_mux_inbit <= 0;
          end
          2'b10, 2'b11: begin
            $display("taken");
            pred <= 1;
            pc_mux_inbit <= 1;
            target_addr <= target_addr_table[BTB_index];
          end
        endcase
      end
      else begin
        pc_mux_inbit <= 0;
      end

      // PHT 업데이트 로직
      if((ID_EX_opcode == `JAL || ID_EX_opcode == `BRANCH) && EX_bcond == 1) begin
        case (PHT[BTB_index])
          2'b00: current_saturation <= 2'b01;
          2'b01: current_saturation <= 2'b10;
          2'b10: current_saturation <= 2'b11;
          2'b11: current_saturation <= 2'b11;
        endcase
        target_addr_table[BTB_index] <= pc_plus_imm;
        tag_table[BTB_index] <= tag; // 태그 테이블 업데이트
      end else if (ID_EX_opcode == `JALR && EX_bcond == 1) begin
        case (PHT[BTB_index])
          2'b00: current_saturation <= 2'b01;
          2'b01: current_saturation <= 2'b10;
          2'b10: current_saturation <= 2'b11;
          2'b11: current_saturation <= 2'b11;
        endcase
        target_addr_table[BTB_index] <= reg_plus_imm;
        tag_table[BTB_index] <= tag; // 태그 테이블 업데이트
      end else if ((ID_EX_opcode == `JAL || ID_EX_opcode == `JALR ||
                    ID_EX_opcode == `BRANCH) && EX_bcond == 0) begin
        case (PHT[BTB_index])
          2'b00: current_saturation <= 2'b00;
          2'b01: current_saturation <= 2'b00;
          2'b10: current_saturation <= 2'b01;
          2'b11: current_saturation <= 2'b10;
        endcase
      end
      PHT[BTB_index] <= current_saturation;
    end
  end
endmodule
```

<그림 21: PHTandBTB.v 모듈의 구성>

해당 모듈을 적용한 후 분기 예측 실패 시의 로직을 추가하여, misprediction 수를 줄일 수 있을 것이며, 더 나은 형태의 Branch prediction 을 구현할 수 있을 것이다

5. Conclusion

주어진 testbench 를 활용하여 basic, non-controlflow, ifelse, loop, recursive 에 해당하는 테스트를 수행하였다. 그 결과, 아래 사진과 같이 올바른 register 값을 출력하는 것을 확인하였다.



```
### SIMULATING ###
TEST END
SIM TIME : 70
TOTAL CYCLE : 34 (Answer : 36)
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 00000013 (Answer : 00000013)
11 00000003 (Answer : 00000003)
12 fffffffd7 (Answer : fffffffd7)
13 00000037 (Answer : 00000037)
14 00000013 (Answer : 00000013)
15 00000026 (Answer : 00000026)
16 0000001e (Answer : 0000001e)
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
```

```
### SIMULATING ###
TEST END
SIM TIME : 90
TOTAL CYCLE : 44 (Answer : 48)
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 ffffffd1 (Answer : ffffffd1)
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
```

```
### SIMULATING ###
TEST END
SIM TIME : 86
TOTAL CYCLE : 42 (Answer : 44)
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 00000000 (Answer : 00000000)
11 00000000 (Answer : 00000000)
12 00000000 (Answer : 00000000)
13 00000000 (Answer : 00000000)
14 0000000a (Answer : 0000000a)
15 00000028 (Answer : 00000028)
16 00000000 (Answer : 00000000)
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
```

```
### SIMULATING ###
TEST END
SIM TIME : 644
TOTAL CYCLE : 321 (Answer : 323)
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 00000000 (Answer : 00000000)
11 00000000 (Answer : 00000000)
12 00000000 (Answer : 00000000)
13 00000000 (Answer : 00000000)
14 0000000a (Answer : 0000000a)
15 00000009 (Answer : 00000009)
16 0000005a (Answer : 0000005a)
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
```

```
### SIMULATING ###
TEST END
SIM TIME : 2374
TOTAL CYCLE : 1186 (Answer : 1188)
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 0000000d (Answer : 0000000d)
11 00000000 (Answer : 00000000)
12 00000000 (Answer : 00000000)
13 00000000 (Answer : 00000000)
14 00000001 (Answer : 00000001)
15 0000000d (Answer : 0000000d)
16 00000015 (Answer : 00000015)
17 0000000a (Answer : 0000000a)
18 00000000 (Answer : 00000000)
19 00000000 (Answer : 00000000)
20 00000000 (Answer : 00000000)
21 00000022 (Answer : 00000022)
22 00000000 (Answer : 00000000)
23 00000037 (Answer : 00000037)
24 00000059 (Answer : 00000059)
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
```

<그림 22: Pipelined CPU w/ controlflow - test simulation 결과>