

CSED311: Lab4 Pipelined CPU

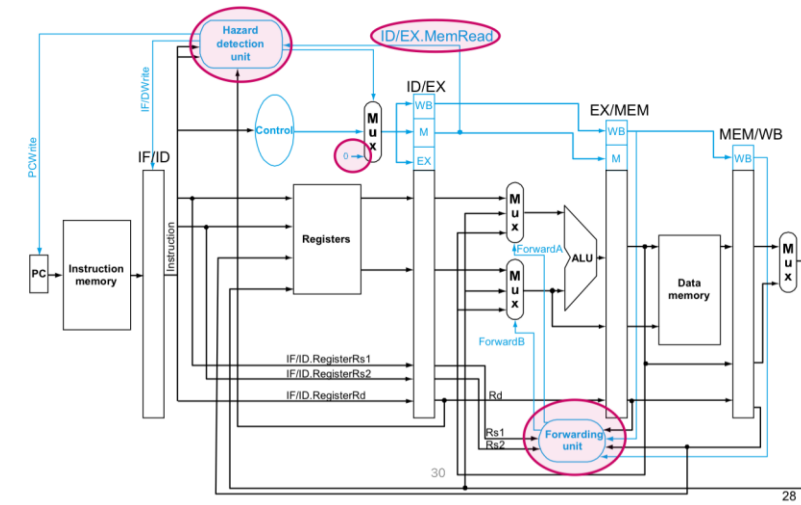
컴퓨터공학과 20220302 김지현

컴퓨터공학과 20220455 윤수인

1. Introduction

- 본 과제는 Pipelined CPU 의 구조를 이해하고, Verilog 를 사용하여 control flow 를 고려하지 않은 Pipelined CPU 를 설계하고 구현하는 것을 목적으로 한다.
- Verilator 환경에서 실행하며, 레지스터 값과 사이클 수를 관찰한다.
- 이전 Lab 에서 설명했듯이, 하나의 instruction 을 수행하는 데에는 5 가지 stage(IF, ID, EX, MEM, WB)를 필요로 하고, 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.

2. Design



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

위의 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.

Hazard detection 과 Forwarding unit 의 자세한 구현은 Implementation 파트에서 진행한다.

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 2 : pc.v 모듈의 구성>

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

```
1  module adder (  
2      input[31:0] in1,  
3      input [31:0] in2,  
4      output reg [31:0] out);  
5  
6      assign out = in1 + in2;  
7  
8  endmodule
```

<그림 3: adder.v 모듈의 구성>

두 개의 32 bit input 레지스터 값을 합하여 out 에 할당한다. 이번 과제에서는 control flow 를 고려하지 않으므로 PC + 4 연산을 처리할 때만 사용된다. 이 과정은 clock asynchronous 하게 이루어진다.

- InstMemory.v

```
1  module InstMemory #(parameter MEM_DEPTH = 1024) (input reset,  
2      input clk,  
3      input [31:0] addr, // address of the instruction memory  
4      output [31:0] dout); // instruction at addr  
5  
6  integer i;  
7  // Instruction memory  
8  reg [31:0] mem[0:MEM_DEPTH - 1];  
9  // Do not touch imem_addr  
10 wire [31:0] imem_addr;  
11 assign imem_addr = {addr >> 2};  
12  
13 // Asynchronously read instruction from the memory  
14 assign dout = mem[imem_addr];  
15  
16 // Initialize instruction memory (do not touch except path)  
17 always @(posedge clk) begin  
18     if (reset) begin  
19         for (i = 0; i < MEM_DEPTH; i = i + 1)  
20             // DO NOT TOUCH COMMENT BELOW  
21             /* verilator lint_off BLKSEQ */  
22             mem[i] = 32'b0;  
23             /* verilator lint_on BLKSEQ */  
24             // DO NOT TOUCH COMMENT ABOVE  
25             // Provide path of the file including instructions with binary format  
26             $readmemh("./student_tb/non-controlflow_mem.txt", mem);  
27     end  
28 end  
29 endmodule
```

<그림 4: InstMemory.v 모듈의 구성>

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

```
1 module DataMemory #(parameter MEM_DEPTH = 16384) (input reset,
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
    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};

    // Asynchronously 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 5: DataMemory.v 모듈의 구성>

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

```
1 module RegisterFile(input reset,
2                     input clk,
3                     input [4:0] rs1,      // source register 1
4                     input [4:0] rs2,      // source register 2
5                     input [4:0] rd,        // destination register
6                     input [31:0] rd_din,   // input data for rd
7                     input write_enable,    // RegWrite signal
8                     output [31:0] rs1_dout, // output of rs 1
9                     output [31:0] rs2_dout, // output of rs 2
10                    output [31:0] print_reg[0:31]); // output of rs 2
11
12 integer i;
13 // Register file
14 reg [31:0] rf[0:31];
15 assign print_reg = rf;
16 // Asynchronously read register file
17 assign rs1_dout = rf[rs1];
18 assign rs2_dout = rf[rs2];
19
20 always @(clk) begin
21     if (clk==0) begin // negative edge
22         if (write_enable & (rd != 0))
23             rf[rd] <= rd_din;
24         end
25     else begin // positive edge
26         if (reset) begin
27             rf[0] <= 32'b0;
28             rf[1] <= 32'b0;
29             rf[2] <= 32'h2ffc; // stack pointer
30             for (i = 3; i < 32; i = i + 1)
31                 rf[i] <= 32'b0;
32             end
33         end
34     end
35 end
36 endmodule
```

<그림 6: RegisterFile.v 모듈의 구성>

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 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.

Clock 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

```

1  `include "opcodes.v"
2
3  module ControlUnit (
4      input [6:0] part_of_inst,
5      output reg mem_read,
6      output reg mem_to_reg,
7      output reg mem_write,
8      output reg alu_src,
9      output reg write_enable,
10     output reg pc_to_reg,
11     output reg use_rs1,
12     output reg use_rs2,
13     output reg is_ecall);
14
15     always @(*) begin
16         mem_read = 0;
17         mem_to_reg = 0;
18         mem_write = 0;
19         alu_src = 0;
20         write_enable = 0;
21         pc_to_reg = 0;
22         use_rs1 = 0;
23         use_rs2 = 0;
24         is_ecall = 0;
25
26         case(part_of_inst)
27             // `JAL: begin
28             // is_jal = 1;
29             // write_enable = 1;
30             // pc_to_reg = 1;
31             // end
32
33             // `JALR: begin
34             // is_jalr = 1;
35             // write_enable = 1;
36             // pc_to_reg = 1;
37             // alu_src = 1;
38             // end
44         `LOAD: begin
45             mem_read = 1;
46             write_enable = 1;
47             mem_to_reg = 1;
48             alu_src = 1;
49             use_rs1 = 1;
50         end
51
52         `STORE: begin
53             mem_write = 1;
54             alu_src = 1;
55             use_rs1 = 1;
56             use_rs2 = 1;
57         end
58
59         `ARITHMETIC_IMM: begin
60             write_enable = 1;
61             alu_src = 1;
62             use_rs1 = 1;
63         end
64
65         `ARITHMETIC: begin
66             write_enable = 1;
67             use_rs1 = 1;
68             use_rs2 = 1;
69         end
70
71         `ECALL: begin
72             is_ecall = 1;
73         end
74
75         default: begin
76             end
77         endcase
78     end
79 endmodule

```

<그림 7: 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 의 경우 non-controlflow 환경이기 때문에 JAL, JALR 연산을 주석 처리하였다.

이 과정은 **clock asynchronous** 하게 진행된다.

- ImmediateGenerator.v

```

1 include "opcodes.v"
2 module immediate_generator (input reg [31:0] part_of_inst,
3                             output reg [31:0] imm_gen_out);
4
5     reg [6:0] opcode;
6
7     always @(*) begin
8         imm_gen_out = 32'b0;
9         opcode = part_of_inst[6:0];
10        case(opcode)
11            /* ARITHMETIC: begin
12                imm_gen_out = 32'b0;
13            end
14            */
15            /* ARITHMETIC_IMM: begin
16                if(part_of_inst[31] == 1) begin // signed
17                    imm_gen_out = {20'hffff, part_of_inst[31:20]};
18                end
19                else begin
20                    imm_gen_out = {20'h0, part_of_inst[31:20]};
21                end
22            end
23        end
24
25        /* LOAD: begin
26            if(part_of_inst[31] == 1) begin // signed
27                imm_gen_out = {20'hffff, part_of_inst[31:20]};
28            end
29            else begin
30                imm_gen_out = {20'h0, part_of_inst[31:20]};
31            end
32        end
33    end
34 endmodule
35
36 36R: begin
37     if(part_of_inst[31] == 1) begin // signed
38         imm_gen_out = {20'hffff, part_of_inst[31:20]};
39     end
40     else begin
41         imm_gen_out = {20'h0, part_of_inst[31:20]};
42     end
43 end
44
45 STORE: begin
46     if(part_of_inst[31] == 1) begin // signed
47         imm_gen_out = {20'hffff, part_of_inst[31:25], part_of_inst[11:8], part_of_inst[7]};
48     end
49     else begin
50         imm_gen_out = {20'h0, part_of_inst[31:25], part_of_inst[11:8], part_of_inst[7]};
51     end
52 end
53
54 BRANCH: begin
55     if(part_of_inst[31] == 1) begin // signed
56         imm_gen_out = {19'hffff, part_of_inst[11], part_of_inst[7], part_of_inst[30:25], part_of_inst[11:8], 1'h0};
57     end
58     else begin
59         imm_gen_out = {19'h0, part_of_inst[11], part_of_inst[7], part_of_inst[30:25], part_of_inst[11:8], 1'h0};
60     end
61 end
62
63 3AL: begin
64     if(part_of_inst[31] == 1) begin // signed
65         imm_gen_out = {11'hfff, part_of_inst[11], part_of_inst[19:12], part_of_inst[20], part_of_inst[30:21], 1'h0};
66     end
67     else begin
68         imm_gen_out = {11'h0, part_of_inst[11], part_of_inst[19:12], part_of_inst[20], part_of_inst[30:21], 1'h0};
69     end
70 end
71
72 /* ECALL: begin
73     imm_gen_out = 32'b0;
74 end */
75
76 default: begin
77     imm_gen_out = 32'b0;
78 end
79 endcase
80 end
81
82 endmodule

```

<그림 8: ImmediateGenerator.v 모듈의 구성>

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

```

1 include "opcodes.v"
2 include "instructions.v"
3 module ALUControlUnit (
4     input [6:0] Opcode,
5     input [2:0] Funct3,
6     input [6:0] Funct7,
7     output reg [31:0] alu_op);
8
9     always @(*) begin
10        alu_op = 32'b11111;
11
12        if(Opcode == 3AL) begin
13            alu_op = INS_JMPL_IMM_3AL;
14        end
15
16        else if(Opcode == 3ALR) begin
17            alu_op = INS_JMPL_IMM_3ALR;
18        end
19
20        else if(Opcode == BRANCH & Funct3 == FUNCT3_BEQ) begin
21            alu_op = INS_BRANCH_BEQ;
22        end
23
24        else if(Opcode == BRANCH & Funct3 == FUNCT3_BNE) begin
25            alu_op = INS_BRANCH_BNE;
26        end
27
28        else if(Opcode == BRANCH & Funct3 == FUNCT3_BLT) begin
29            alu_op = INS_BRANCH_BLT;
30        end
31
32        else if(Opcode == BRANCH & Funct3 == FUNCT3_BGE) begin
33            alu_op = INS_BRANCH_BGE;
34        end
35
36        else if(Opcode == LOAD) begin
37            alu_op = INS_LOAD_IMM;
38        end
39
40        else if(Opcode == STORE) begin
41            alu_op = INS_STORE_IMM;
42        end
43
44        else if(Opcode == ARITHMETIC_IMM & Funct3 == FUNCT3_ADD) begin
45            alu_op = INS_ARITHMETIC_IMM_ADD;
46        end
47
48        else if(Opcode == ARITHMETIC_IMM & Funct3 == FUNCT3_XOR) begin
49            alu_op = INS_ARITHMETIC_IMM_XOR;
50        end
51
52        else if(Opcode == ARITHMETIC_IMM & Funct3 == FUNCT3_OR) begin
53            alu_op = INS_ARITHMETIC_IMM_OR;
54        end
55
56        else if(Opcode == ARITHMETIC_IMM & Funct3 == FUNCT3_AND) begin
57            alu_op = INS_ARITHMETIC_IMM_AND;
58        end
59
60        else if(Opcode == ARITHMETIC_IMM & Funct3 == FUNCT3_SLL) begin
61            alu_op = INS_ARITHMETIC_IMM_SLL;
62        end
63
64        else if(Opcode == ARITHMETIC_IMM & Funct3 == FUNCT3_SRL) begin
65            alu_op = INS_ARITHMETIC_IMM_SRL;
66        end
67
68        else if(Opcode == ARITHMETIC & Funct3 == FUNCT3_ADD) begin
69            if(Funct7 == FUNCT7_SIM) begin
70                alu_op = INS_ARITHMETIC_SIM;
71            end
72            else if(Funct7 == FUNCT7_OTHERS) begin
73                alu_op = INS_ARITHMETIC_ADD;
74            end
75        end
76
77        else if(Opcode == ARITHMETIC & Funct3 == FUNCT3_SLL) begin
78            alu_op = INS_ARITHMETIC_SLL;
79        end
80
81        else if(Opcode == ARITHMETIC & Funct3 == FUNCT3_SRL) begin
82            alu_op = INS_ARITHMETIC_SRL;
83        end
84
85        else if(Opcode == ARITHMETIC & Funct3 == FUNCT3_OR) begin
86            alu_op = INS_ARITHMETIC_OR;
87        end
88
89        else if(Opcode == ARITHMETIC & Funct3 == FUNCT3_AND) begin
90            alu_op = INS_ARITHMETIC_AND;
91        end
92
93        else if(Opcode == ECALL) begin
94            alu_op = INS_ECALL;
95        end
96    end
97 endmodule

```

<그림 9: ALUControlUnit.v 모듈의 구성>

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

```
1 include "instructions.v"
2
3 module alu (
4     input [4:0] alu_op, // input
5     input [31:0] alu_in_1, // input
6     input [31:0] alu_in_2, // input
7     output reg [31:0] alu_result, // output
8     output reg alu_bcond; // output
9
10    always @(*) begin
11        alu_result = 0;
12        alu_bcond = 0;
13
14        case(alu_op)
15            'INS_JUMPLINK_JALR, 'INS_LOAD_LW, 'INS_STORE_SW, 'INS_ARITHMETIC_IMM_ADDI, 'INS_ARITHMETIC_ADD: begin
16                alu_result = alu_in_1 + alu_in_2;
17            end
18
19            'INS_BRANCH_BEQ: begin
20                alu_bcond = (alu_in_1 == alu_in_2) ? 1 : 0;
21            end
22
23            'INS_BRANCH_BNE: begin
24                alu_bcond = (alu_in_1 != alu_in_2) ? 1 : 0;
25            end
26
27            'INS_BRANCH_BLT: begin
28                alu_bcond = (alu_in_1 < alu_in_2) ? 1 : 0;
29            end
30
31            'INS_BRANCH_BGE: begin
32                alu_bcond = (alu_in_1 >= alu_in_2) ? 1 : 0;
33            end
34
35            'INS_ARITHMETIC_IMM_XORI, 'INS_ARITHMETIC_XOR: begin
36                alu_result = alu_in_1 ^ alu_in_2;
37            end
38
39            'INS_ARITHMETIC_IMM_ORI, 'INS_ARITHMETIC_OR: begin
40                alu_result = alu_in_1 | alu_in_2;
41            end
42
43            'INS_ARITHMETIC_IMM_ANDI, 'INS_ARITHMETIC_AND: begin
44                alu_result = alu_in_1 & alu_in_2;
45            end
46
47            'INS_ARITHMETIC_IMM_SLLI, 'INS_ARITHMETIC_SLL: begin
48                alu_result = alu_in_1 << alu_in_2;
49            end
50
51            'INS_ARITHMETIC_IMM_SRLI, 'INS_ARITHMETIC_SRL: begin
52                alu_result = alu_in_1 >> alu_in_2;
53            end
54
55            'INS_ARITHMETIC_SUB: begin
56                alu_result = alu_in_1 - alu_in_2;
57            end
58
59            default: begin
60
61            end
62        endcase
63    end
64 endmodule
```

<그림 10: alu.v 모듈의 구성>

alu.v 모듈은 앞서 ALUControlUnit.v 모듈에서 지정한 alu_op 에 따라 알맞은 연산을 수행하여 연산 결과를 alu_result 에 저장한다. 이 과정은 clock asynchronous 하게 이루어진다.

- ForwardingEcallUnit.v

```
1 module ForwardingEcallUnit(  
2     input [4:0] rs1,  
3     input [4:0] ID_EX_rd,  
4     input [4:0] EX_MEM_rd,  
5     input [4:0] MEM_WB_rd,  
6     input ID_EX_reg_write,  
7     input EX_MEM_reg_write,  
8     input MEM_WB_reg_write,  
9     output reg [1:0] forwardEcall  
10 );  
11  
12 always @(*) begin  
13     if((rs1 == ID_EX_rd) && ID_EX_reg_write) begin  
14         forwardEcall = 2'b01;  
15     end  
16     else if((rs1 == EX_MEM_rd) && EX_MEM_reg_write) begin  
17         forwardEcall = 2'b10;  
18     end  
19     else if((rs1 == MEM_WB_rd) && MEM_WB_reg_write) begin  
20         forwardEcall = 2'b11;  
21     end  
22     else begin  
23         forwardEcall = 2'b00;  
24     end  
25 end  
26  
27 endmodule  
28
```

<그림 11: ForwardingEcallUnit.v 모듈의 구성>

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 12: MUX2x1.v 모듈의 구성>

MUX2x1.v 모듈은 in_bit의 값에 따라 in1과 in2 중 하나의 값을 반환하는 역할로, 전부 cpu.v에서 사용되었다. 필요에 따라 5 bit input, output을 다루는 MUX2x1_5bit 모듈을 추가로 만들어 사용하였다. MUX2x1이 사용된 부분은 다음과 같다.

- mux_handle_ecall: is_ecall 신호의 값에 따라 rs1의 값을 17 또는 기존 rs1 값으로 결정한다. Output인 rs1_out을 RegisterFile의 input으로 사용한다.
- mux_ALUSrc: EX stage에서 alu의 두 번째 input을 결정하기 위한 mux로, ID_EX_alu_src의 값에 따라 data forwarding된 rs2 데이터 값 또는 immediate 값을 alu의 두 번째 input으로 결정한다.
- mux_MemtoReg: 레지스터에 저장할 값을 결정하는 mux로, MEM_WB_mem_to_reg 신호의 값에 따라 alu의 결과 값 혹은 data memory의 결과 값을 rd_din에 저장한다.

이 과정은 clock asynchronous 하게 이루어진다.

- ForwardingUnit.v

```

1  `include "opcodes.v"
2
3  `define MEM 2'b01
4  `define WB 2'b10
5
6  module ForwardingUnit(
7      input [4:0] ID_EX_rs1,
8      input [4:0] ID_EX_rs2,
9      input [4:0] EX_MEM_rd,
10     input EX_MEM_reg_write,
11     input [4:0] MEM_WB_rd,
12     input MEM_WB_reg_write,
13     input [4:0] _x0,
14     output reg [1:0] ForwardA,
15     output reg [1:0] ForwardB);
16
17     always @(*) begin
18         if((ID_EX_rs1 != _x0) && (ID_EX_rs1 == EX_MEM_rd) && EX_MEM_reg_write) begin
19             ForwardA = `MEM;
20         end
21         else if((ID_EX_rs1 != _x0) && (ID_EX_rs1 == MEM_WB_rd) && MEM_WB_reg_write) begin
22             ForwardA = `WB;
23         end
24         else begin
25             ForwardA = 2'b00;
26         end
27
28         if((ID_EX_rs2 != _x0) && (ID_EX_rs2 == EX_MEM_rd) && EX_MEM_reg_write) begin
29             ForwardB = `MEM;
30         end
31         else if((ID_EX_rs2 != _x0) && (ID_EX_rs2 == MEM_WB_rd) && MEM_WB_reg_write) begin
32             ForwardB = `WB;
33         end
34         else begin
35             ForwardB = 2'b00;
36         end
37     end
38
39 endmodule

```

<그림 13: ForwardingUnit.v 모듈의 구성>

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 14: MUX4x1.v 모듈의 구성>

MUX4x1 모듈은 in_bit 값에 따라 4 가지 (in0~in3) input 중 하나의 값을 반환하는 역할이다. 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.

이는 Clock asynchronous 하게 이루어진다.

- HazardDetectUnit.v

```
1 module HazardDetectUnit(  
2     input [4:0] rs1,  
3     input [4:0] rs2,  
4     input [4:0] ID_EX_rd,  
5     input ID_EX_mem_read,  
6     input use_rs1,  
7     input use_rs2,  
8     output reg PCWrite,  
9     output reg IF_IDWrite,  
10    output reg hazard_detected);  
11  
12  
13    always @(*) begin  
14        if((((rs1 == ID_EX_rd) && use_rs1)) || (((rs2 == ID_EX_rd) && use_rs2)) && ID_EX_mem_read) begin  
15            hazard_detected = 1;  
16            IF_IDWrite = 0;  
17            PCWrite = 0;  
18        end  
19        else begin  
20            hazard_detected = 0;  
21            IF_IDWrite = 1;  
22            PCWrite = 1;  
23        end  
24    end  
25  
26 endmodule
```

<그림 15: HazardDetectUnit.v 모듈의 구성>

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.

4. Discussion

- Single cycle CPU 와 Pipelined CPU 의 사이클 수 비교(non-controlflow test 의 경우)
Single cycle CPU 에서는 수행해야 할 instruction 의 수가 39 개이고, 각 instruction 별로 하나의 사이클이 할당되어 있기 때문에, 총 사이클 수는 39 개가 되어야 한다.
Pipelined CPU 에서는 총 39 개의 사이클에 추가로 병렬적으로 instruction 을 처리하기 때문에, 4 개의 사이클이 추가적으로 필요하다. 11 번째 instruction 에서 lw 후 해당 레지스터의 값이 바로 다음 instruction 에 사용되기 때문에, out of order execution 을 사용하지 않는 한 한 번의 stall 이 발생한다. 따라서 총 $39 + 4 + 1 = 44$ 개의 사이클이 소모된다.
사이클 수만 관찰하였을 때는 Pipelined CPU 의 소요 시간이 Single cycle CPU 보다 큰 것처럼 보이나, Pipelined CPU 는 clock period 가 critical path 에 해당하는 시간으로 결정되므로, 각 사이클에 소모되는 시간이 Single cycle CPU 에 비해 더 짧다.
- Hazard Detection 구현
위의 HazardDetectUnit 에서 hazard 가 감지되었을 경우, hazard_detected 를 1 로 설정하고, stall 을 발생시켜야 한다. 따라서 ID_EX pipeline register 에 값을 넘겨줄 때 hazard_detected 가 1 인 경우, mem_write 과 reg_write 값을 0 으로 설정해 준다.
- Data Forwarding 구현
앞서 설명한 바와 같이, ForwardingUnit 에서 어떤 stage 의 결과 값을 forwarding 할지 결정하고, mux 를 사용하여 alu 의 첫 번째 input 값과, ALUSrc mux 의 input 으로 사용될 값을 결정한다.

5. Conclusion

주어진 testbench 를 활용하여 non-controlflow_mem 에 해당하는 테스트를 수행하였다. 그 결과, 아래 사진과 같이 올바른 register 값을 출력하는 것을 확인하였다. 사이클 수는 Discussion 에서 설명한 바와 같이 44 로 나타나는 것을 확인할 수 있었다.

```
### SIMULATING ###
TEST END
SIM TIME : 90
TOTAL CYCLE : 44 (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
```

<그림 16: Pipelined CPU 에서 non-controlflow test simulation 결과>