|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **컴퓨터구조 결 과 보 고 서 (Result report)** | | | | |
| **Major** | **Student ID** | **Grade** | **Name** | **Experiment date** |
| 융합전자공학부 | 2015036580 | 3 | 김수영 | 2021-06-24 |
| Title | Advanced Pipeline 과제 | | | |

1. 전체 코드 첨부

(디버깅을 위한 output은 주석 처리되어 있음)

`timescale 1ns / 1ps

*//////////////////////////////////////////////////////////////////////////////////*

*// Company:*

*// Engineer:*

*//*

*// Create Date:    12:48:03 06/22/2021*

*// Design Name:*

*// Module Name:    advanced\_pipeline*

*// Project Name:*

*// Target Devices:*

*// Tool versions:*

*// Description:*

*//*

*// Dependencies:*

*//*

*// Revision:*

*// Revision 0.01 - File Created*

*// Additional Comments:*

*//*

*//////////////////////////////////////////////////////////////////////////////////*

module advanced\_pipeline (clk, reset, result, PC\_now,

    d\_Branch, d\_Jump, d\_RS\_RT\_Equal, d\_Branch\_Taken);

*// PC\_now, Instruction\_now,*

*// Rs\_now, Rt\_now, ALU\_input\_1, ALU\_input\_2, immi\_Shifted, PC\_4,*

*// load\_data, Beq\_address, J\_address, Write\_Register,*

*// debug\_flag\_2, Debug\_RegWrite, Debug\_Write\_Data,*

*// Debug\_PCWrite, Debug\_IF\_ID\_Write, Debug\_IF\_Flush,*

*// Debug\_ForWardA, Debug\_ForWardB);*

    input clk, reset;

    output [31:0] result, PC\_now; *// ALU result*

    input d\_Branch, d\_Jump, d\_RS\_RT\_Equal, d\_Branch\_Taken;

*// output [31:0] load\_data; // Data\_memory\_read*

*// output [31:0] PC\_now; // current PC value Debugging,*

*// output [31:0] Instruction\_now; // Fetch stage instruction*

*// output [31:0] Rs\_now, Rt\_now; // output from Register File*

*// output [31:0] ALU\_input\_1, ALU\_input\_2; // ALU inputs*

*// output [31:0] PC\_4, Beq\_address, immi\_Shifted; // for Branch Address Debugging*

*// output [31:0] J\_address; // Jump Address*

*// output [4:0] Write\_Register; // Rd value*

*// output [31:0] debug\_flag\_2, Debug\_Write\_Data; // ALU\_result in MEM stage, Write\_Data for Register File*

*// output Debug\_RegWrite; // RegWrite Op Flag*

*// output Debug\_PCWrite, Debug\_IF\_ID\_Write, Debug\_IF\_Flush;*

*// output [1:0] Debug\_ForWardA, Debug\_ForWardB;*

*// IF stage*

    wire [31:0] PC\_in;

    wire [31:0] PC\_out;

    wire [31:0] PC\_plus4;

    wire [31:0] IF\_instruction;

    wire [31:0] Branch\_MUX\_output;

    wire [31:0] Jump\_MUX\_output;

    wire PCWrite;

    wire IF\_ID\_Write;

    wire Branch\_taken;

*// ID stage*

    wire [31:0] ID\_PC\_plus4;

    wire [31:0] ID\_instruction;

    wire [31:0] Read\_data\_1, Read\_data\_2;

    wire [31:0] sign\_extended\_immi;

    wire [31:0] Jump\_Address;

    wire [31:0] ID\_Shifted\_immi;

    wire [31:0] ID\_Branch\_addr;

    wire ALUSrc, RegWrite;

    wire RegDst, Jump, Branch;

    wire MemRead, MemtoReg, MemWrite;

    wire RegDst\_t, Jump\_t, Branch\_t;

    wire MemRead\_t, MemtoReg\_t, MemWrite\_t;

    wire IF\_Flush, ID\_Flush, EX\_Flush;

    wire Rs\_Rt\_equal;

    wire [1:0] ALUOp;

    wire Hazard, Control\_Reset;

*// EX stage*

    wire [31:0] EX\_PC\_plus4;

    wire [31:0] EX\_instruction;

    wire [31:0] EX\_Read\_data\_1, EX\_Read\_data\_2;

    wire [31:0] EX\_sign\_extended\_immi;

    wire EX\_ALUSrc, EX\_RegWrite;

    wire EX\_RegDst, EX\_Jump, EX\_Branch;

    wire EX\_MemRead, EX\_MemtoReg, EX\_MemWrite;

    wire [1:0] EX\_ALUOp;

    wire [4:0] EX\_Write\_Register;

    wire [3:0] operation\_code;

    wire [31:0] Branch\_addr;

    wire [31:0] Shifted\_immi;

    wire [31:0] ALU\_input\_B;

    wire [31:0] ALU\_result;

    wire ALU\_zero;

    wire [1:0] ForwardA, ForwardB;

    wire [31:0] muxA\_out, muxB\_out;

*// MEM stage*

    wire [31:0] MEM\_ALU\_result, MEM\_Read\_data\_2;

    wire [4:0] MEM\_Write\_Register;

    wire [31:0] Data\_memory\_read;

    wire [31:0] MEM\_Branch\_addr;

    wire MEM\_RegWrite, MEM\_MemtoReg, MEM\_Branch;

    wire MEM\_MemRead, MEM\_MemWrite, MEM\_Jump;

    wire MEM\_ALU\_zero;

    wire PCSrc;

*// WB stage*

    wire [31:0] WB\_Data\_memory\_read, WB\_ALU\_result;

    wire [4:0] WB\_Write\_Register;

    wire [31:0] WB\_Write\_Data;

    wire WB\_RegWrite, WB\_MemtoReg;

*//////////////////////////////////*

*//// Instruction Fetch stage /////*

*//////////////////////////////////*

    Program\_Counter program\_counter (.clk(clk), .reset(reset),

     .PC\_in(PC\_in), .PC\_out(PC\_out), .PCWrite(PCWrite));

    Instruction\_Memory instruction\_memory (.address(PC\_out),

     .instruction(IF\_instruction), .reset(reset));

    ALU\_add\_only pc\_add\_4 (.input1(PC\_out), .input2(32'b0100), .add\_out(PC\_plus4));

*// assign Debug\_PCWrite = PCWrite;*

*// assign Debug\_IF\_Flush = IF\_Flush;*

    assign Branch\_taken = Branch & Rs\_Rt\_equal;

*// and (Branch\_taken, Branch, Rs\_Rt\_equal);*

    N\_bit\_MUX #(32) branch\_mux (.input0(PC\_plus4), .input1(ID\_Branch\_addr),

     .mux\_out(Branch\_MUX\_output), .control(Branch\_taken));

    N\_bit\_MUX #(32) jump\_mux (.input0(Branch\_MUX\_output), .input1(Jump\_Address),

     .mux\_out(PC\_in), .control(Jump));

    assign d\_Jump = Jump;

    assign d\_Branch = Branch;

    assign d\_RS\_RT\_Equal = Rs\_Rt\_equal;

    assign d\_Branch\_Taken = Branch\_taken;

    IF\_ID\_Stage\_Reg IF\_ID\_Stage\_Unit (.clk(clk), .reset(reset),

     .IF\_ID\_Write(IF\_ID\_Write), .IF\_Flush(IF\_Flush),

     .PC\_plus4\_in(PC\_plus4), .PC\_plus4\_out(ID\_PC\_plus4),

     .instruction\_in(IF\_instruction), .instruction\_out(ID\_instruction));

    assign PC\_now = PC\_out;

*// assign Instruction\_now = IF\_instruction;*

*// assign Debug\_IF\_ID\_Write = IF\_ID\_Write;*

*//////////////////////////////////*

*//// Instruction Decode stage ////*

*//////////////////////////////////*

*// assign Debug\_RegWrite = WB\_RegWrite;*

*// assign Debug\_Write\_Data = WB\_Write\_Data;*

    Register\_File regfile\_Unit (.clk(clk), .reset(reset),

     .Read\_Register\_1(ID\_instruction[25:21]),

     .Read\_Register\_2(ID\_instruction[20:16]),

     .Write\_Register(WB\_Write\_Register), .Write\_Data(WB\_Write\_Data),

     .Read\_Data\_1(Read\_data\_1), .Read\_Data\_2(Read\_data\_2),

     .RegWrite(WB\_RegWrite));

    assign Rs\_Rt\_equal = (Read\_data\_1 == Read\_data\_2)?1:0;

    assign Jump\_Address = {ID\_PC\_plus4[31:28], ID\_instruction[25:0], 2'b00};

    Sign\_Extension immi\_sign\_extension (.input\_16(ID\_instruction[15:0]), .output\_32(sign\_extended\_immi));

    assign ID\_Shifted\_immi = { sign\_extended\_immi[29:0], 2'b00 };

    ALU\_add\_only alu\_add\_only\_unit (.input1(ID\_PC\_plus4),

    .input2(ID\_Shifted\_immi), .add\_out(ID\_Branch\_addr));

*// assign immi\_Shifted = ID\_Shifted\_immi;*

*// assign Beq\_address = ID\_Branch\_addr;*

*// assign J\_address = Jump\_Address;*

    Hazard\_Detection\_Unit hazard\_detection\_unit (.ID\_EX\_RegisterRt(EX\_instruction[20:16]), .ID\_EX\_MemRead(EX\_MemRead),

     .IF\_ID\_RegisterRs(ID\_instruction[25:21]), .IF\_ID\_RegisterRt(ID\_instruction[20:16]),

     .PCWrite(PCWrite), .IF\_ID\_Write(IF\_ID\_Write), .Hazard(Hazard));

    Control control\_unit (.OpCode(ID\_instruction[31:26]),

     .RegDst(RegDst), .Jump(Jump), .Branch(Branch),

     .MemRead(MemRead), .MemtoReg(MemtoReg), .ALUOp(ALUOp),

     .MemWrite(MemWrite), .ALUSrc(ALUSrc), .RegWrite(RegWrite),

     .IF\_Flush(IF\_Flush), .ID\_Flush(ID\_Flush), .EX\_Flush(EX\_Flush));

    or (Control\_Reset, ID\_Flush, Hazard);

    Reset\_Control reset\_control\_unit(

     .RegDst\_in(RegDst), .ALUSrc\_in(ALUSrc), .Branch\_in(Branch),

     .MemRead\_in(MemRead), .MemtoReg\_in(MemtoReg),

     .MemWrite\_in(MemWrite), .RegWrite\_in(RegWrite),

     .RegDst\_out(RegDst\_t), .ALUSrc\_out(ALUSrc\_t), .Branch\_out(Branch\_t),

     .MemRead\_out(MemRead\_t), .MemtoReg\_out(MemtoReg\_t),

     .MemWrite\_out(MemWrite\_t), .RegWrite\_out(RegWrite\_t),

     .Control\_Reset(Control\_Reset));

    ID\_EX\_Stage\_Reg ID\_EX\_Stage\_Unit (.clk(clk), .reset(reset),

     .RegWrite\_in(RegWrite\_t), .RegWrite\_out(EX\_RegWrite),

     .MemtoReg\_in(MemtoReg\_t), .MemtoReg\_out(EX\_MemtoReg),

     .Branch\_in(Branch\_t), .Branch\_out(EX\_Branch),

     .MemRead\_in(MemRead\_t), .MemRead\_out(EX\_MemRead),

     .MemWrite\_in(MemWrite\_t), .MemWrite\_out(EX\_MemWrite),

     .Jump\_in(Jump), .Jump\_out(EX\_Jump),

     .RegDst\_in(RegDst\_t), .RegDst\_out(EX\_RegDst),

     .ALUSrc\_in(ALUSrc\_t), .ALUSrc\_out(EX\_ALUSrc),

     .ALUOp\_in(ALUOp), .ALUOp\_out(EX\_ALUOp),

     .PC\_plus4\_in(ID\_PC\_plus4), .PC\_plus4\_out(EX\_PC\_plus4),

     .read\_data\_1\_in(Read\_data\_1), .read\_data\_1\_out(EX\_Read\_data\_1),

     .read\_data\_2\_in(Read\_data\_2), .read\_data\_2\_out(EX\_Read\_data\_2),

     .sign\_extended\_immi\_in(sign\_extended\_immi), .sign\_extended\_immi\_out(EX\_sign\_extended\_immi),

     .instruction\_in(ID\_instruction), .instruction\_out(EX\_instruction));

*// assign Rs\_now = Read\_data\_1;*

*// assign Rt\_now = Read\_data\_2;*

*//////////////////////////////////*

*///////   Execute  stage  ////////*

*//////////////////////////////////*

    N\_bit\_MUX #(5) write\_reg\_mux (.input0(EX\_instruction[20:16]), .input1(EX\_instruction[15:11]),

     .mux\_out(EX\_Write\_Register), .control(EX\_RegDst));

    ALU\_Control alu\_control\_unit (.ALUOp(EX\_ALUOp), .f\_code(EX\_instruction[5:0]), .operation\_code(operation\_code));

    Forwarding\_Unit forwarding\_unit (.EX\_RegisterRs(EX\_instruction[25:21]), .EX\_RegisterRt(EX\_instruction[20:16]),

    .MEM\_RegisterRd(MEM\_Write\_Register), .WB\_RegisterRd(WB\_Write\_Register),

    .EX\_MEM\_RegWrite(MEM\_RegWrite), .MEM\_WB\_RegWrite(WB\_RegWrite),

    .ForwardA(ForwardA), .ForwardB(ForwardB));

    Mux\_32bit\_3to1 ALU\_MuxA (.in00(EX\_Read\_data\_1), .in01(WB\_Write\_Data), .in10(MEM\_ALU\_result),

        .mux\_out(muxA\_out), .control(ForwardA));

    Mux\_32bit\_3to1 ALU\_MuxB (.in00(EX\_Read\_data\_2), .in01(WB\_Write\_Data), .in10(MEM\_ALU\_result),

        .mux\_out(muxB\_out), .control(ForwardB));

*// assign Debug\_ForWardA = ForwardA;*

*// assign Debug\_ForWardB = ForwardB;*

    N\_bit\_MUX #(32) alu\_input\_mux (.input0(muxB\_out), .input1(EX\_sign\_extended\_immi),

     .mux\_out(ALU\_input\_B), .control(EX\_ALUSrc));

    ALU alu\_unit (.input1(muxA\_out), .input2(ALU\_input\_B),

     .alu\_out(ALU\_result), .zero(ALU\_zero), .control(operation\_code));

*// assign Shifted\_immi = { EX\_sign\_extended\_immi[29:0], 2'b00 };*

*// ALU\_add\_only alu\_add\_only\_unit (.input1(EX\_PC\_plus4),*

*// .input2(Shifted\_immi), .add\_out(Branch\_addr));*

    EX\_MEM\_Stage\_Reg EX\_MEM\_Stage\_Unit ( .clk(clk), .reset(reset),

    .RegWrite\_in(EX\_RegWrite), .RegWrite\_out(MEM\_RegWrite),

    .MemtoReg\_in(EX\_MemtoReg), .MemtoReg\_out(MEM\_MemtoReg),

    .Branch\_in(EX\_Branch), .Branch\_out(MEM\_Branch),

    .MemRead\_in(EX\_MemRead), .MemRead\_out(MEM\_MemRead),

    .MemWrite\_in(EX\_MemWrite),.MemWrite\_out(MEM\_MemWrite),

    .Jump\_in(EX\_Jump), .Jump\_out(MEM\_Jump),

*// .Branch\_addr\_in(Branch\_addr), .Branch\_addr\_out(MEM\_Branch\_addr),*

    .ALU\_zero\_in(ALU\_zero), .ALU\_zero\_out(MEM\_ALU\_zero),

    .ALU\_result\_in(ALU\_result), .ALU\_result\_out(MEM\_ALU\_result),

    .Read\_data\_2\_in(EX\_Read\_data\_2), .Read\_data\_2\_out(MEM\_Read\_data\_2),

    .RegisterRd\_in(EX\_Write\_Register), .RegisterRd\_out(MEM\_Write\_Register));

*// assign PC\_4 = EX\_PC\_plus4;*

*// assign ALU\_input\_1 = muxA\_out;*

*// assign ALU\_input\_2 = ALU\_input\_B;*

    assign result = { 27'b0, ALU\_result[4:0]};

*//////////////////////////////////*

*///////   Memory  stage  /////////*

*//////////////////////////////////*

    Data\_Memory data\_memory\_unit (.clk(clk), .reset(reset), .MemAddr(MEM\_ALU\_result[7:0]),

     .Write\_Data(MEM\_Read\_data\_2), .Read\_Data(Data\_memory\_read),

     .MemRead(MEM\_MemRead), .MemWrite(MEM\_MemWrite));

    and (PCSrc, MEM\_Branch, MEM\_ALU\_zero);

*// assign debug\_flag\_2 = MEM\_ALU\_result;*

*// assign load\_data = Data\_memory\_read;*

    MEM\_WB\_Stage\_Reg MEM\_WB\_Stage\_Unit (.clk(clk), .reset(reset),

    .RegWrite\_in(MEM\_RegWrite), .RegWrite\_out(WB\_RegWrite),

    .MemtoReg\_in(MEM\_MemtoReg), .MemtoReg\_out(WB\_MemtoReg),

    .Data\_memory\_read\_in(Data\_memory\_read), .Data\_memory\_read\_out(WB\_Data\_memory\_read),

    .ALU\_result\_in(MEM\_ALU\_result), .ALU\_result\_out(WB\_ALU\_result),

    .Write\_Register\_in(MEM\_Write\_Register), .Write\_Register\_out(WB\_Write\_Register));

*//////////////////////////////////*

*///   Write Back stage      //////*

*//////////////////////////////////*

    N\_bit\_MUX #(32) write\_data\_mux (.input0(WB\_ALU\_result), .input1(WB\_Data\_memory\_read),

     .mux\_out(WB\_Write\_Data), .control(WB\_MemtoReg));

*// assign Write\_Register = WB\_Write\_Register;*

endmodule

*// IF/ID stage register*

module IF\_ID\_Stage\_Reg (clk, reset,

    PC\_plus4\_in, PC\_plus4\_out,

    instruction\_in, instruction\_out,

    IF\_ID\_Write, IF\_Flush);

    input clk, reset;

    input [31:0] PC\_plus4\_in, instruction\_in;

    output reg [31:0] PC\_plus4\_out, instruction\_out;

*// Hazard*

    input IF\_Flush, IF\_ID\_Write;

    always @(posedge clk or posedge reset) begin

        if (reset) begin

            PC\_plus4\_out <= 32'b0;

            instruction\_out <= 32'b0;

        end

        else if (IF\_Flush) begin

*// TODO : Check condition*

*// PC\_plus4\_out <= 32'b0;*

*// PC\_plus4\_out <= PC\_plus4\_in;*

            instruction\_out <= 32'b0;

        end

        else if (IF\_ID\_Write) begin

            PC\_plus4\_out <= PC\_plus4\_in;

            instruction\_out <= instruction\_in;

        end

    end

endmodule

*// ID/EX stage register*

module ID\_EX\_Stage\_Reg (clk, reset, RegWrite\_in, RegWrite\_out, MemtoReg\_in, MemtoReg\_out,

    Branch\_in, Branch\_out, MemRead\_in, MemRead\_out, MemWrite\_in, MemWrite\_out,

    Jump\_in, Jump\_out, RegDst\_in, RegDst\_out, ALUSrc\_in, ALUSrc\_out, ALUOp\_in, ALUOp\_out,

    PC\_plus4\_in, PC\_plus4\_out, read\_data\_1\_in, read\_data\_1\_out,

    read\_data\_2\_in, read\_data\_2\_out,

    sign\_extended\_immi\_in, sign\_extended\_immi\_out,

    instruction\_in, instruction\_out);

*// WB control signal*

    input RegWrite\_in, MemtoReg\_in;

    output reg RegWrite\_out, MemtoReg\_out;

*// MEM control signal*

    input Branch\_in, MemRead\_in, MemWrite\_in, Jump\_in;

    output reg Branch\_out, MemRead\_out, MemWrite\_out, Jump\_out;

*// EX control signal*

    input RegDst\_in, ALUSrc\_in;

    output reg RegDst\_out, ALUSrc\_out;

    input [1:0] ALUOp\_in;

    output reg [1:0] ALUOp\_out;

*// addr content*

    input [31:0] PC\_plus4\_in;

    output reg [31:0] PC\_plus4\_out;

*// data content*

    input [31:0] read\_data\_1\_in, read\_data\_2\_in, sign\_extended\_immi\_in;

    output reg [31:0] read\_data\_1\_out, read\_data\_2\_out, sign\_extended\_immi\_out;

*// reg content*

    input [31:0] instruction\_in;

    output reg [31:0] instruction\_out;

*// general signal*

    input clk, reset;

    always @(posedge clk or posedge reset) begin

        if (reset) begin

            RegWrite\_out <= 1'b0; MemtoReg\_out <= 1'b0;

            Branch\_out <= 1'b0; MemRead\_out <= 1'b0;

            MemWrite\_out <= 1'b0; Jump\_out <= 1'b0;

            RegDst\_out <= 1'b0; ALUSrc\_out <= 1'b0;

            ALUOp\_out <= 2'b0;

            PC\_plus4\_out <= 32'b0;

            read\_data\_1\_out <= 32'b0; read\_data\_2\_out <= 32'b0;

            sign\_extended\_immi\_out <= 32'b0;

            instruction\_out <= 32'b0;

        end

        else begin

            RegWrite\_out <= RegWrite\_in; MemtoReg\_out <= MemtoReg\_in;

            Branch\_out <= Branch\_in; MemRead\_out <= MemRead\_in;

            MemWrite\_out <= MemWrite\_in; Jump\_out <= Jump\_in;

            RegDst\_out <= RegDst\_in; ALUSrc\_out <= ALUSrc\_in;

            ALUOp\_out <= ALUOp\_in; PC\_plus4\_out <= PC\_plus4\_in;

            sign\_extended\_immi\_out <= sign\_extended\_immi\_in;

            read\_data\_1\_out <= read\_data\_1\_in;

            read\_data\_2\_out <= read\_data\_2\_in;

            instruction\_out <= instruction\_in;

        end

    end

endmodule

*// EX/MEM stage register*

module EX\_MEM\_Stage\_Reg (clk, reset,

    RegWrite\_in, RegWrite\_out, MemtoReg\_in, MemtoReg\_out,

    Branch\_in, Branch\_out, MemRead\_in, MemRead\_out,

    MemWrite\_in, MemWrite\_out, Jump\_in, Jump\_out,

*// Branch\_addr\_in, Branch\_addr\_out,*

    ALU\_zero\_in, ALU\_zero\_out,

    ALU\_result\_in, ALU\_result\_out, Read\_data\_2\_in, Read\_data\_2\_out,

    RegisterRd\_in, RegisterRd\_out);

*// WB control signal*

    input RegWrite\_in, MemtoReg\_in;

    output reg RegWrite\_out, MemtoReg\_out;

*// MEM control signal*

    input Branch\_in, MemRead\_in, MemWrite\_in, Jump\_in;

    output reg Branch\_out, MemRead\_out, MemWrite\_out, Jump\_out;

*// addr content*

*// input [31:0] Branch\_addr\_in;*

*// output reg [31:0] Branch\_addr\_out;*

*// data content*

    input ALU\_zero\_in;

    output reg ALU\_zero\_out;

*// results*

    input [31:0] ALU\_result\_in, Read\_data\_2\_in;

    output reg [31:0] ALU\_result\_out, Read\_data\_2\_out;

*// registers*

    input [4:0] RegisterRd\_in;

    output reg [4:0] RegisterRd\_out;

*// general signal*

    input clk, reset;

    always @(posedge clk or posedge reset) begin

        if (reset) begin

          RegWrite\_out <= 1'b0; MemtoReg\_out <= 1'b0;

          Branch\_out <= 1'b0; MemRead\_out <= 1'b0;

          MemWrite\_out <= 1'b0; Jump\_out <= 1'b0;

*//   Branch\_addr\_out <= 32'b0;*

          ALU\_zero\_out <= 1'b0;

          ALU\_result\_out <= 32'b0; Read\_data\_2\_out <= 32'b0;

          RegisterRd\_out <= 5'b0;

        end

        else begin

          RegWrite\_out <= RegWrite\_in; MemtoReg\_out <= MemtoReg\_in;

          Branch\_out <= Branch\_in; MemRead\_out <= MemRead\_in;

          MemWrite\_out <= MemWrite\_in; Jump\_out <= Jump\_in;

*//   Branch\_addr\_out <= Branch\_addr\_in;*

          ALU\_zero\_out <= ALU\_zero\_in;

          ALU\_result\_out <= ALU\_result\_in; Read\_data\_2\_out <= Read\_data\_2\_in;

          RegisterRd\_out <= RegisterRd\_in;

        end

    end

endmodule

*// MEM/WB stage register*

module MEM\_WB\_Stage\_Reg (RegWrite\_in, RegWrite\_out,

    MemtoReg\_in, MemtoReg\_out,

    Data\_memory\_read\_in, Data\_memory\_read\_out,

    ALU\_result\_in, ALU\_result\_out,

    Write\_Register\_in, Write\_Register\_out, clk, reset);

*// WB control signal*

    input RegWrite\_in, MemtoReg\_in;

    output reg RegWrite\_out, MemtoReg\_out;

*// data content*

    input [31:0] Data\_memory\_read\_in, ALU\_result\_in;

    output reg [31:0] Data\_memory\_read\_out, ALU\_result\_out;

    input [4:0] Write\_Register\_in;

    output reg [4:0] Write\_Register\_out;

*// general signal*

    input clk, reset;

    always @(posedge clk or posedge reset) begin

        if (reset) begin

            RegWrite\_out <= 1'b0; MemtoReg\_out <= 1'b0;

            Data\_memory\_read\_out <= 32'b0;  ALU\_result\_out <= 32'b0;

            Write\_Register\_out <= 5'b0;

        end

        else begin

            RegWrite\_out <= RegWrite\_in; MemtoReg\_out <= MemtoReg\_in;

            Data\_memory\_read\_out <= Data\_memory\_read\_in; ALU\_result\_out <= ALU\_result\_in;

            Write\_Register\_out <= Write\_Register\_in;

        end

    end

endmodule

*// PC*

module Program\_Counter (clk, reset, PC\_in, PC\_out, PCWrite);

    input PCWrite;

    input clk, reset;

    input [31:0] PC\_in;

    output reg [31:0] PC\_out;

    always @ (posedge clk or posedge reset) begin

        if (reset)

            PC\_out <= 0;

        else if (PCWrite)

            PC\_out <= PC\_in;

    end

endmodule

*// contains hard-code instructions*

module Instruction\_Memory (address, instruction, reset);

    input reset;

    input [31:0] address;

    output [31:0] instruction;

    reg [31:0] mem [7:0]; *// 8 instructions*

    integer k;

*// get instruction right away*

    assign instruction = mem[address[6:2]];

*// Initial setup at reset posedge*

    always @(posedge reset) begin

        for (k = 0; k < 8; k = k + 1) begin

            mem[k] = 32'b0; *// add $0 $0 $0*

        end

        mem[0] = 32'b100011\_00010\_00001\_0000000000000100; *// lw $1, 4($2)*

        mem[1] = 32'b000000\_00001\_00101\_00100\_00000\_100010; *// sub $4, $1, $5*

        mem[2] = 32'b000000\_00001\_00111\_00110\_00000\_100100; *// and $6, $1, $7*

        mem[3] = 32'b000000\_00001\_01001\_01000\_00000\_100101; *// or $8, $1, $9*

        mem[4] = 32'b000100\_00110\_00000\_1111111111111011; *// beq $6, $0, Label (-5)*

*// mem[0] = 32'b000000\_00011\_00100\_00010\_00000\_100000; // add $2, $3, $4*

*// mem[1] = 32'b000000\_00011\_00100\_00001\_00000\_100010; // sub $1, $3, $4*

*// mem[2] = 32'b100011\_00110\_00101\_0000000000000000; // lw $5, 0($6)*

*// mem[3] = 32'b000100\_00011\_00100\_1111111111111100; // beq $3, $4, Label (-4)*

    end

endmodule

*// 32-bit ALU for addition only*

module ALU\_add\_only (input1, input2, add\_out);

    input [31:0] input1, input2;

    output [31:0] add\_out;

    assign add\_out=input1+input2;

endmodule

*// N\_bit\_MUX for Usability*

module N\_bit\_MUX (input0, input1, mux\_out, control);

    parameter N = 32;

    input [N-1:0] input0, input1;

    input control;

    output [N-1:0] mux\_out;

    assign mux\_out = control ? input1 : input0;

endmodule

*// sync register file (write/read occupy half cycle each)*

*// write: on rising edge; data width 32 bit; address width 5 bit*

*// read: on falling edge; data width 32 bit; address width 5 bit*

module Register\_File (Read\_Register\_1, Read\_Register\_2,

    Write\_Register, Write\_Data, Read\_Data\_1, Read\_Data\_2,

    RegWrite, clk, reset);

    input [4:0] Read\_Register\_1, Read\_Register\_2, Write\_Register;

    input [31:0] Write\_Data;

    input clk, reset, RegWrite;

    output reg [31:0] Read\_Data\_1, Read\_Data\_2;

    reg [31:0] mem [15:0];

    integer k;

    always @(posedge clk or posedge reset) begin

        if (reset) begin

            for (k = 0; k < 16; k = k + 1) begin

                mem[k] <= 32'b0;

            end

            mem[1] <= 20;

            mem[2] <= 8;

            mem[5] <= 2;

            mem[6] <= 0;

            mem[7] <= 1;

            mem[9] <= 3;

*// mem[3] = 32'b0011;*

*// mem[4] = 32'b0011;*

*// mem[6] = 32'h0000\_0040;*

        end

        else if (RegWrite) begin

            mem[Write\_Register] <= Write\_Data;

        end

    end

    always @(\*) begin

        Read\_Data\_1 <= mem[Read\_Register\_1];

        Read\_Data\_2 <= mem[Read\_Register\_2];

    end

endmodule

*// sign-extend the 16-bit input to the 32\_bit output*

module Sign\_Extension (input\_16, output\_32);

    input [15:0] input\_16;

    output [31:0] output\_32;

    assign output\_32[15:0]  = input\_16[15:0];

    assign output\_32[31:16] = input\_16[15] ? 16'b1111\_1111\_1111\_1111: 16'b0;

endmodule

*// Control Path*

module Control (OpCode, RegDst,  MemRead, MemtoReg, ALUOp, MemWrite, ALUSrc, RegWrite,

    Jump, Branch, IF\_Flush, ID\_Flush, EX\_Flush);

    input [5:0] OpCode;

    output [1:0] ALUOp;

    output RegDst, MemRead, MemtoReg, MemWrite, ALUSrc, RegWrite;

    output Jump, Branch;

    output IF\_Flush, ID\_Flush, EX\_Flush;

*// 000000 : add, sub, and, or, slt*

*// 001000 : addi*

*// 100011 : lw*

*// 101011 : sw*

*// 000100 : beq*

*// 000010 : j*

*// 000000 (R-format)*

    assign RegDst=(~OpCode[5])&(~OpCode[4])&(~OpCode[3])&(~OpCode[2])&(~OpCode[1])&(~OpCode[0]);

*// 000000 (R-format)*

    assign ALUOp[1]=(~OpCode[5])&(~OpCode[4])&(~OpCode[3])&(~OpCode[2])&(~OpCode[1])&(~OpCode[0]);

*// 000100 (beq)*

    assign ALUOp[0]=(~OpCode[5])&(~OpCode[4])&(~OpCode[3])&(OpCode[2])&(~OpCode[1])&(~OpCode[0]);

*// 100011 (lw), 101011 (sw)*

    assign ALUSrc=((OpCode[5])&(~OpCode[4])&(~OpCode[3])&(~OpCode[2])&(OpCode[1])&(OpCode[0]))  |

                      ((OpCode[5])&(~OpCode[4])&(OpCode[3])&(~OpCode[2])&(OpCode[1])&(OpCode[0]));

*// 100011 (lw)*

    assign MemRead=(OpCode[5])&(~OpCode[4])&(~OpCode[3])&(~OpCode[2])&(OpCode[1])&(OpCode[0]);

*// 101011 (sw)*

    assign MemWrite=(OpCode[5])&(~OpCode[4])&(OpCode[3])&(~OpCode[2])&(OpCode[1])&(OpCode[0]);

*// 100011 (lw)*

    assign MemtoReg=(OpCode[5])&(~OpCode[4])&(~OpCode[3])&(~OpCode[2])&(OpCode[1])&(OpCode[0]);

*// 000000 (R-format), 001000 (addi), 001100, 100011 (lw)*

    assign RegWrite=((~OpCode[5])&(~OpCode[4])&(~OpCode[3])&(~OpCode[2])&(~OpCode[1])&(~OpCode[0]))|

                    ((~OpCode[5])&(~OpCode[4])&(OpCode[3])&(~OpCode[2])&(~OpCode[1])&(~OpCode[0])) |

                         ((OpCode[5])&(~OpCode[4])&(~OpCode[3])&(~OpCode[2])&(OpCode[1])&(OpCode[0]));

*// 000100 (beq)*

    assign Branch=(~OpCode[5])&(~OpCode[4])&(~OpCode[3])&(OpCode[2])&(~OpCode[1])&(~OpCode[0]);

*// 000010 (j)*

    assign Jump=(~OpCode[5])&(~OpCode[4])&(~OpCode[3])&(~OpCode[2])&(OpCode[1])&(~OpCode[0]);

*// only jump*

    assign IF\_Flush=(~OpCode[5])&(~OpCode[4])&(~OpCode[3])&(~OpCode[2])&(OpCode[1])&(~OpCode[0]);

*// jump or branch*

    assign ID\_Flush=(~OpCode[5])&(~OpCode[4])&(~OpCode[3])&(~OpCode[2])&(OpCode[1])&(~OpCode[0]) |

                    (~OpCode[5])&(~OpCode[4])&(~OpCode[3])&(OpCode[2])&(~OpCode[1])&(~OpCode[0]);

*// Umm...*

    assign EX\_Flush=0;

endmodule

module ALU\_Control (ALUOp, f\_code, operation\_code);

    input [1:0] ALUOp;

    input [5:0] f\_code;

    output [3:0] operation\_code;

    assign operation\_code[3]=0;

*// 0  1   | x x x x x x branch => subtract*

*// 1  0   | x x 0 0 1 0 => R-type subtract*

*// 1  0   | x x 1 0 1 0 => R-type slt*

    assign operation\_code[2]=((~ALUOp[1])&(ALUOp[0])) |

                        ((ALUOp[1])&(~ALUOp[0])&(~f\_code[3])&(~f\_code[2])&(f\_code[1])&(~f\_code[0])) |

                        ((ALUOp[1])&(~ALUOp[0])&(f\_code[3])&(~f\_code[2])&(f\_code[1])&(~f\_code[0]));

*// 0  0   | x x x x x x lw or sw => add*

*// 0  1   | x x x x x x branch => subtract*

*// 1  0   | x x 0 0 0 0 => R-type add*

*// 1  0   | x x 0 0 1 0 => R-type subtract*

*// 1  0   | x x 1 0 1 0 => R-type slt*

    assign operation\_code[1]=((~ALUOp[1])&(~ALUOp[0])) |

                               ((~ALUOp[1])&(ALUOp[0]))  |

                                ((ALUOp[1])&(~ALUOp[0])&(~f\_code[3])&(~f\_code[2])&(~f\_code[1])&(~f\_code[0])) |

                                ((ALUOp[1])&(~ALUOp[0])&(~f\_code[3])&(~f\_code[2])&(f\_code[1])&(~f\_code[0]))  |

                                ((ALUOp[1])&(~ALUOp[0])&(f\_code[3])&(~f\_code[2])&(f\_code[1])&(~f\_code[0]));

*// ALU OP | f\_code field*

*// 1  0   | x x 0 1 0 1 => R-type Or*

*// 1  0   | x x 1 0 1 0 => R-type slt*

    assign operation\_code[0]=((ALUOp[1])&(~ALUOp[0])&(~f\_code[3])&(f\_code[2])&(~f\_code[1])&(f\_code[0])) |

                                ((ALUOp[1])&(~ALUOp[0])&(f\_code[3])&(~f\_code[2])&(f\_code[1])&(~f\_code[0]));

endmodule

*// 32-bit ALU*

module ALU (input1, input2, alu\_out, zero, control);

*// TODO : negative number handling*

    input [31:0] input1, input2;

    input [3:0] control;

    output reg [31:0] alu\_out;

    output reg zero;

    always @ (control or input1 or input2) begin

        case (control)

*// and*

            4'b0000: begin alu\_out<=input1&input2; zero<=0; end

*// or*

            4'b0001: begin alu\_out<=input1|input2; zero<=0; end

*// add*

            4'b0010: begin alu\_out<=input1+input2; zero<=0; end

*// subtract*

            4'b0110: begin

                if(input1 == input2)

                    zero <= 1;

                else

                    zero <= 0;

                    alu\_out <= input1 - input2;

                end

*// slt*

            4'b0111: begin

                zero <= 0;

                if(input1 - input2 >= 32'h8000\_0000)

                    alu\_out <= 32'b1;

                else

                    alu\_out <= 32'b0;

                end

        default: begin

            zero <= 0;

            alu\_out <= input1;

        end

        endcase

    end

endmodule

*// Referred Previous Memory Project*

module Data\_Memory (MemAddr, Write\_Data, Read\_Data, clk, reset, MemRead, MemWrite);

    input clk, reset;

    input MemRead, MemWrite;

    input [7:0] MemAddr;

    input [31:0] Write\_Data;

    output reg [31:0] Read\_Data;

    reg [31:0] mem [31:0];

    integer k;

    always @(\*) begin

        if (reset) begin

            for (k = 0; k < 32; k = k + 1) begin

                mem[k] = 32'b0;

            end

*// mem[0xC] = 30;*

            mem[3] = 30;

            mem[12] = 32'b0001\_1110;

            mem[16] = 30;

        end

        else

            if (MemRead && !MemWrite) begin

                Read\_Data = mem[MemAddr[7:2]];

            end

            else if (!MemRead && MemWrite) begin

                mem[MemAddr[7:2]] = Write\_Data;

            end

        else begin

            Read\_Data = 32'bx;

        end

    end

endmodule

module Forwarding\_Unit (EX\_RegisterRs, EX\_RegisterRt,

    MEM\_RegisterRd, WB\_RegisterRd,

    EX\_MEM\_RegWrite, MEM\_WB\_RegWrite,

*// IF\_ID\_RegisterRs, IF\_ID\_RegisterRt,*

    ForwardA, ForwardB);

*// ForwardC, ForwardD);*

    input [4:0] MEM\_RegisterRd, WB\_RegisterRd, EX\_RegisterRs, EX\_RegisterRt;

*// input [4:0] IF\_ID\_RegisterRs,IF\_ID\_RegisterRt;*

    input EX\_MEM\_RegWrite, MEM\_WB\_RegWrite;

*// output ForwardC,ForwardD;*

    output reg [1:0] ForwardA, ForwardB;

*// reg ForwardC,ForwardD;*

    wire EX\_HazardA, EX\_HazardB;

    wire WB\_HazardA, WB\_HazardB;

    assign EX\_HazardA = EX\_MEM\_RegWrite & (MEM\_RegisterRd!=0) & (MEM\_RegisterRd==EX\_RegisterRs);

    assign EX\_HazardB = EX\_MEM\_RegWrite & (MEM\_RegisterRd!=0) & (MEM\_RegisterRd==EX\_RegisterRt);

    assign WB\_HazardA = MEM\_WB\_RegWrite & (WB\_RegisterRd!=0) & (WB\_RegisterRd==EX\_RegisterRs);

    assign WB\_HazardB = MEM\_WB\_RegWrite & (WB\_RegisterRd!=0) & (WB\_RegisterRd==EX\_RegisterRt);

    always@ (EX\_MEM\_RegWrite or MEM\_WB\_RegWrite or EX\_HazardA or WB\_HazardA or EX\_HazardB or WB\_HazardB) begin

        if(EX\_HazardA)

            ForwardA<=2'b10;

        else if(!EX\_HazardA & WB\_HazardA)

            ForwardA<=2'b01;

        else

            ForwardA<=2'b00;

        if(EX\_HazardB)

            ForwardB<=2'b10;

        else if(!EX\_HazardB & WB\_HazardB)

            ForwardB<=2'b01;

        else

            ForwardB<=2'b00;

    end

endmodule

module Hazard\_Detection\_Unit (ID\_EX\_RegisterRt, ID\_EX\_MemRead,

    IF\_ID\_RegisterRs, IF\_ID\_RegisterRt,

    PCWrite, IF\_ID\_Write, Hazard);

    input [4:0] IF\_ID\_RegisterRs, IF\_ID\_RegisterRt;

    input [4:0] ID\_EX\_RegisterRt;

    input ID\_EX\_MemRead;

    output reg PCWrite, IF\_ID\_Write, Hazard;

    wire Rs\_Stall, Rt\_Stall;

    assign Rs\_Stall=(ID\_EX\_RegisterRt==IF\_ID\_RegisterRs);

    assign Rt\_Stall=(ID\_EX\_RegisterRt==IF\_ID\_RegisterRt);

    always@(\*) begin

        if(ID\_EX\_MemRead & (Rs\_Stall|Rt\_Stall)) begin

            PCWrite <= 0;

            IF\_ID\_Write <=0;

            Hazard <= 1;

        end

        else begin

            PCWrite <= 1;

            IF\_ID\_Write <= 1;

            Hazard <= 0;

        end

    end

endmodule

module Reset\_Control (RegDst\_in, ALUSrc\_in, Branch\_in, MemRead\_in,

    MemtoReg\_in, MemWrite\_in, RegWrite\_in, Control\_Reset,

    RegDst\_out, ALUSrc\_out, Branch\_out, MemRead\_out,

    MemtoReg\_out, MemWrite\_out, RegWrite\_out);

    input RegDst\_in, ALUSrc\_in, Branch\_in, MemRead\_in;

    input MemtoReg\_in, MemWrite\_in, RegWrite\_in;

    input Control\_Reset;

    output RegDst\_out, ALUSrc\_out, Branch\_out, MemRead\_out;

    output MemtoReg\_out, MemWrite\_out, RegWrite\_out;

    assign RegDst\_out = Control\_Reset ? 0 : RegDst\_in;

    assign ALUSrc\_out = Control\_Reset ? 0 : ALUSrc\_in;

    assign Branch\_out = Control\_Reset ? 0 : Branch\_in;

    assign MemRead\_out = Control\_Reset ? 0 : MemRead\_in;

    assign MemtoReg\_out = Control\_Reset ? 0 : MemtoReg\_in;

    assign MemWrite\_out = Control\_Reset ? 0 : MemWrite\_in;

    assign RegWrite\_out = Control\_Reset ? 0 : RegWrite\_in;

endmodule

module Mux\_32bit\_3to1 (in00, in01, in10, mux\_out, control);

    input [31:0] in00, in01, in10;

    output [31:0] mux\_out;

    input [1:0] control;

    reg [31:0] mux\_out;

    always @(in00 or in01 or in10 or control)

    begin

        case(control)

        2'b00:mux\_out<=in00;

        2'b01:mux\_out<=in01;

        2'b10:mux\_out<=in10;

        default: mux\_out<=in00;

        endcase

    end

endmodule

2. 실행 결과

텍스트이(가) 표시된 사진

자동 생성된 설명

위 코드는 상단 조건에 따라 다음과 같은 작업을 반복한다.

Beq에 의해 다시 Label로 되돌아가기 때문에, 같은 과정이 반복된다.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| PC Value | Fetch | Decode | Execute | Memory | Write Back |
| 00000 | lw |  |  |  |  |
| 00100 | Sub | lw |  |  |  |
| 01000 | And | Sub | lw |  |  |
| 01000 | And | Sub | **Stall** | lw |  |
| 01100 | Or | And | Sub**(Forwarding)** | Garbage | lw |
| 10000 | Beq | Or | And | Sub | Garbage |
| 10100 | **Flush** | Beq | Or | And | Sub |
| 00000(반복) | lw | NOP | Beq | Or | And |
| 00100 | Sub | Lw | NOP | Beq | Or |
| 01000 | Or | Sub | Lw | NOP | Beq |
| 01000 | Beq | Sub | **Stall** | lw | NOP |

Advanced Pipeline으로 변경되면서 논점이 되는 포인트들은 다음과 같다.

1. 01000에서, lw다음으로 sub이 오기 때문에, stall이 발생한다는 것을 알 수 있다.
2. 01100에서, sub의 rs인 $1의 값이 lw로부터 forwarding된다는 것을 알 수 있다.
3. 10100에서, Beq에 의한 Flush가 발생한다는 것을 알 수 있다.

디버깅을 위한 매개변수들을 추가하여 만든 waveform은 다음과 같다.

각 매개변수들에 대한 설명은 다음과 같다.

* Result : ALU 계산 결과 (후에 FPGA LED 값이 된다.)
* PC\_Now : 현 시점에서 Instruction Fetch에서 사용되는 PC를 나타냄
* Instruction\_Now : 현 시점에서 Instruction Decode에서 사용되는 Instruction을 나타냄
* Rs\_Now & Rt\_Now : Instruction\_Now를 통해 Register\_File에서 가져온 Rs,Rt 값이다. Falling Edge에서 최신화됨
* ALU\_Input\_1, ALU\_Input\_2 : Execute 단계에서 ALU의 input이 되는 두 피연산자
* Load\_data : Memory 단계에서 읽어온 데이터 (현재는 lw에서만 해당)
* Beq\_Address : Execute 단계에서 계산되는 branch address
* J\_address : Execute 단계에서 계산되는 Jump할 Address
* Immi\_Shifted : Execute 단계에서 계산되는 32비트 shift된 값 (이것이 PC와 합쳐져 Beq\_Address가 된다.)
* PC\_4 : Execute 단계에서의 PC, 큰 의미는 없다. 편의를 위해 추가
* Debug\_flag\_2 : 편의상 이것저것을 확인하기 위한 flag, 큰 의미는 없다.
* Debug\_Write\_Data, Write\_Register : Decode 단계의 Register\_File에 Write되는 Register Address와 Data
* Debug\_RegWrite, Debug\_PCWrite, Debug\_IF\_ID\_Write, Debug\_IF\_Flush : Hazard Detection과 stall 기능이 추가됨에 따른 Control 비트를 디버깅한다.
* Debug\_ForWardA, Debug\_ForWardB : Forwarding Unit이 추가됨에 따른 2비트의 Forwarding A, Forwarding B를 디버깅한다.

|  |  |
| --- | --- |
| 텍스트이(가) 표시된 사진  자동 생성된 설명lw | 텍스트이(가) 표시된 사진  자동 생성된 설명sub |
| 텍스트이(가) 표시된 사진  자동 생성된 설명And | 텍스트이(가) 표시된 사진  자동 생성된 설명Sub Stall |
| 텍스트이(가) 표시된 사진  자동 생성된 설명Or | Beq |
| 텍스트이(가) 표시된 사진  자동 생성된 설명NOP | 텍스트이(가) 표시된 사진  자동 생성된 설명lw (반복) |
| 텍스트이(가) 표시된 사진  자동 생성된 설명Sub (반복) |  |

각 Pipeline 단계별로 결과를 구조화해보면 다음과 같다.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| PC\_Now | Fetch | Decode | Execute | Memory | Write Back |
| 00000 | Instruction\_Now = lw |  |  |  |  |
| 00100 | Instruction\_Now = Sub | Rs=$2 / Rt=$1 |  |  |  |
| 01000 | Instruction\_Now = And | Rs=$1 / Rt=$5 | Result=$2+4=1100 |  |  |
| 01000 | Instruction\_Now = And | Rs=$1 / Rt=$5 | **Stall (쓰레기 값)** | Mem[0xC] => 30 |  |
| 01100 | Instruction\_Now = Or | Rs=$1 / Rt=$7 | Result=$1-$5=28 | **Stall** | $1에 결과 30저장요청 |
| 10000 | Instruction\_Now = Beq | Rs=$1 / Rt=$9 | Result=$1 & $5=0  **WB Forwarding 발생** | sub이므로 없음 | **Stall** |
| 10100 | **Flush** | Rs=$6 / Rt=$0  Branch주소계산 | Result=30 | 3 =11111 | And이므로 없음 |  |
| 00000(반복 | Instruction\_Now = lw |  | Result=$6-$0=0 | Or이므로 없음 | $6에 결과 0 저장요청 |
| 00100 | Instruction\_Now = Sub | Rs=$2 / Rt=$1 |  | Beq이므로 없음 | $8에 결과 11111 저장 |
| 01000 | Instruction\_Now = And | Rs=$1 / Rt=$5 | Result=$2+4=1100 |  | Beq이므로 없음 |
| 01000 | Instruction\_Now = And | Rs=$1 / Rt=$5 | **Stall (쓰레기 값)** | Mem[0xC] => 30 |  |

3. 코드 설명

필요하다고 판단되는 Unit에 대한 코드 설명을 추가한다.

* Forwarding Unit

module Forwarding\_Unit (EX\_RegisterRs, EX\_RegisterRt,

    MEM\_RegisterRd, WB\_RegisterRd,

    EX\_MEM\_RegWrite, MEM\_WB\_RegWrite,

*// IF\_ID\_RegisterRs, IF\_ID\_RegisterRt,*

    ForwardA, ForwardB);

*// ForwardC, ForwardD);*

    input [4:0] MEM\_RegisterRd, WB\_RegisterRd, EX\_RegisterRs, EX\_RegisterRt;

*// input [4:0] IF\_ID\_RegisterRs,IF\_ID\_RegisterRt;*

    input EX\_MEM\_RegWrite, MEM\_WB\_RegWrite;

*// output ForwardC,ForwardD;*

    output reg [1:0] ForwardA, ForwardB;

*// reg ForwardC,ForwardD;*

    wire EX\_HazardA, EX\_HazardB;

    wire WB\_HazardA, WB\_HazardB;

    assign EX\_HazardA = EX\_MEM\_RegWrite & (MEM\_RegisterRd!=0) & (MEM\_RegisterRd==EX\_RegisterRs);

    assign EX\_HazardB = EX\_MEM\_RegWrite & (MEM\_RegisterRd!=0) & (MEM\_RegisterRd==EX\_RegisterRt);

    assign WB\_HazardA = MEM\_WB\_RegWrite & (WB\_RegisterRd!=0) & (WB\_RegisterRd==EX\_RegisterRs);

    assign WB\_HazardB = MEM\_WB\_RegWrite & (WB\_RegisterRd!=0) & (WB\_RegisterRd==EX\_RegisterRt);

    always@ (EX\_MEM\_RegWrite or MEM\_WB\_RegWrite or EX\_HazardA or WB\_HazardA or EX\_HazardB or WB\_HazardB) begin

        if(EX\_HazardA)

            ForwardA<=2'b10;

        else if(!EX\_HazardA & WB\_HazardA)

            ForwardA<=2'b01;

        else

            ForwardA<=2'b00;

        if(EX\_HazardB)

            ForwardB<=2'b10;

        else if(!EX\_HazardB & WB\_HazardB)

            ForwardB<=2'b01;

        else

            ForwardB<=2'b00;

    end

endmodule

00, 01, 10의 세가지 결과를 갖게 되며, 각 상황에 부합하는 조건은 교과서의 다음 로직을 따른다.

텍스트이(가) 표시된 사진

자동 생성된 설명

Forwarding Unit의 결과에 따라 ALU의 input이 달라지며, 이는 다음과 같이 구현 가능하다.

    Mux\_32bit\_3to1 ALU\_MuxA (.in00(EX\_Read\_data\_1), .in01(WB\_Write\_Data), .in10(MEM\_ALU\_result),

        .mux\_out(muxA\_out), .control(ForwardA));

    Mux\_32bit\_3to1 ALU\_MuxB (.in00(EX\_Read\_data\_2), .in01(WB\_Write\_Data), .in10(MEM\_ALU\_result),

        .mux\_out(muxB\_out), .control(ForwardB));

*// assign Debug\_ForWardA = ForwardA;*

*// assign Debug\_ForWardB = ForwardB;*

    N\_bit\_MUX #(32) alu\_input\_mux (.input0(muxB\_out), .input1(EX\_sign\_extended\_immi),

     .mux\_out(ALU\_input\_B), .control(EX\_ALUSrc));

* Hazard Detection Unit

module Hazard\_Detection\_Unit (ID\_EX\_RegisterRt, ID\_EX\_MemRead,

    IF\_ID\_RegisterRs, IF\_ID\_RegisterRt,

    PCWrite, IF\_ID\_Write, Hazard);

    input [4:0] IF\_ID\_RegisterRs, IF\_ID\_RegisterRt;

    input [4:0] ID\_EX\_RegisterRt;

    input ID\_EX\_MemRead;

    output reg PCWrite, IF\_ID\_Write, Hazard;

    wire Rs\_Stall, Rt\_Stall;

    assign Rs\_Stall=(ID\_EX\_RegisterRt==IF\_ID\_RegisterRs);

    assign Rt\_Stall=(ID\_EX\_RegisterRt==IF\_ID\_RegisterRt);

    always@(\*) begin

        if(ID\_EX\_MemRead & (Rs\_Stall|Rt\_Stall)) begin

            PCWrite <= 0;

            IF\_ID\_Write <=0;

            Hazard <= 1;

        end

        else begin

            PCWrite <= 1;

            IF\_ID\_Write <= 1;

            Hazard <= 0;

        end

    end

endmodule

lw의 결과를 바로 다음 연산에서 사용하고자 하는 경우 발생하며, 그 조건은 수업 중 살펴본 바와 같이 다음과 같다.

If (IF.ID.RS or IF.ID.RT == ID.EXE.RT) and (MemRead == 1)

IF.ID.Write를 0으로 만들고 PCWrite도 0으로 만든다. 그리고 Hazard라는 wire를 1로 하는데, 이것은 Control 신호를 제어하는데 사용한다.

    Hazard\_Detection\_Unit hazard\_detection\_unit (.ID\_EX\_RegisterRt(EX\_instruction[20:16]), .ID\_EX\_MemRead(EX\_MemRead),

     .IF\_ID\_RegisterRs(ID\_instruction[25:21]), .IF\_ID\_RegisterRt(ID\_instruction[20:16]),

     .PCWrite(PCWrite), .IF\_ID\_Write(IF\_ID\_Write), .Hazard(Hazard));

    Control control\_unit (.OpCode(ID\_instruction[31:26]),

     .RegDst(RegDst), .Jump(Jump), .Branch(Branch),

     .MemRead(MemRead), .MemtoReg(MemtoReg), .ALUOp(ALUOp),

     .MemWrite(MemWrite), .ALUSrc(ALUSrc), .RegWrite(RegWrite),

     .IF\_Flush(IF\_Flush), .ID\_Flush(ID\_Flush), .EX\_Flush(EX\_Flush));

    or (Control\_Reset, ID\_Flush, Hazard);

    Reset\_Control reset\_control\_unit(

     .RegDst\_in(RegDst), .ALUSrc\_in(ALUSrc), .Branch\_in(Branch),

     .MemRead\_in(MemRead), .MemtoReg\_in(MemtoReg),

     .MemWrite\_in(MemWrite), .RegWrite\_in(RegWrite),

     .RegDst\_out(RegDst\_t), .ALUSrc\_out(ALUSrc\_t), .Branch\_out(Branch\_t),

     .MemRead\_out(MemRead\_t), .MemtoReg\_out(MemtoReg\_t),

     .MemWrite\_out(MemWrite\_t), .RegWrite\_out(RegWrite\_t),

     .Control\_Reset(Control\_Reset));

위 코드에서 볼 수 있듯, Hazard와 Control로부터의 ID\_Flush값들 중 하나라도 1이 된다면 Control과 관련된 wire를 전부 0으로 만드는 Reset\_Control이 실행된다. 그래서, lw 다음의 Sub가 ALU연산을 할 지 모르지만, 메모리에는 어떠한 영향도 주지 않기 때문에, 사실상 쓰레기 값이 된다.

다음으로 Branch에 따른 Instruction Fetch 단계에서의 변화가 있다.

    and (Branch\_taken, Branch, Rs\_Rt\_equal);

    N\_bit\_MUX #(32) branch\_mux (.input0(PC\_plus4), .input1(ID\_Branch\_addr),

     .mux\_out(Branch\_MUX\_output), .control(Branch\_taken));

    N\_bit\_MUX #(32) jump\_mux (.input0(Branch\_MUX\_output), .input1(Jump\_Address),

     .mux\_out(PC\_in), .control(Jump));

과제 그림에서 볼 수 있듯이, RS와 RT가 같고, Control output 중 Branch가 1이라면, 추가 Step을 밟지 않고 바로 Branch 및 Flush를 진행한다. 위 그림은 Jump에 대한 경우도 보이고 있는데, 이것은 후에 설명하겠다.

*// IF/ID stage register*

module IF\_ID\_Stage\_Reg (clk, reset,

    PC\_plus4\_in, PC\_plus4\_out,

    instruction\_in, instruction\_out,

    IF\_ID\_Write, IF\_Flush);

    input clk, reset;

    input [31:0] PC\_plus4\_in, instruction\_in;

    output reg [31:0] PC\_plus4\_out, instruction\_out;

*// Hazard*

    input IF\_Flush, IF\_ID\_Write;

    always @(posedge clk or negedge reset) begin

        if (!reset) begin

            PC\_plus4\_out <= 32'b0;

            instruction\_out <= 32'b0;

        end

        else if (IF\_Flush) begin

*// TODO : Check condition*

*// PC\_plus4\_out <= 32'b0;*

*// PC\_plus4\_out <= PC\_plus4\_in;*

            instruction\_out <= 32'b0;

        end

        else if (IF\_ID\_Write) begin

            PC\_plus4\_out <= PC\_plus4\_in;

            instruction\_out <= instruction\_in;

        end

    end

endmodule

* IF\_ID\_Stage\_Reg

IF.ID.Write와 IF.Flush가 추가됨에 따라 이에 대한 추가 처리가 이루어진다.

IF.Flush는 instruction을 전부 0으로 만들고, IF.ID.Write는 0이 될 시 PC update가 이루어지지 않는다.

다음으로 FPGA 구동을 위한 Startup 코드를 보자.

* Startup

`timescale 1ns / 1ps

*//////////////////////////////////////////////////////////////////////////////////*

*// Company:*

*// Engineer:*

*//*

*// Create Date:    23:36:42 06/21/2021*

*// Design Name:*

*// Module Name:    startup*

*// Project Name:*

*// Target Devices:*

*// Tool versions:*

*// Description:*

*//*

*// Dependencies:*

*//*

*// Revision:*

*// Revision 0.01 - File Created*

*// Additional Comments:*

*//*

*//////////////////////////////////////////////////////////////////////////////////*

module Startup(

    input   clk\_50MHz,

    input reset,

    output [15:0] ALU\_Result,

    output reg [3:0] digit,

    output reg [7:0] fnd,

    output Branch,

    output Jump,

    output RS\_RT\_Equal,

    output Branch\_Taken

    );

    reg [24:0] counter;

    reg [3:0] number;

    reg [15:0] test\_led = 0;

    wire clk\_slow;

    reg digit\_sel;

    reg clk\_operating;

    assign clk\_slow = clk\_operating;

    wire slow\_clock;

    wire [31:0] PC;

    clock\_divider my\_divider(.clk(clk\_50MHz), .rst(reset), .clk\_operating(slow\_clock), .my\_clk(clk));

*//assign debug\_slow\_clock = slow\_clock;*

    advanced\_pipeline my\_pipeline(.clk(slow\_clock), .reset(reset), .result(ALU\_Result), .PC\_now(PC),

        .d\_Branch(Branch), .d\_Jump(Jump), .d\_RS\_RT\_Equal(RS\_RT\_Equal), .d\_Branch\_Taken(Branch\_Taken));

*//assign Debug\_ALU = ALU\_Result;*

*//Fnd number*

*// always @(posedge clk or negedge reset)*

    always @(posedge clk or posedge reset)

        begin

            if (reset)

                begin

                    digit\_sel<=0;

                    digit<=4'b0000;

                    number<=0;

                end

            else

                begin

                    digit\_sel <= digit\_sel + 1'b1;

                        case(digit\_sel)

                            0: begin digit<=4'b1110; number<= PC % 32'd10; end

                            1: begin digit<=4'b1101; number<= PC / 32'd10; end

                        endcase

                end

            end

*//Fnd number*

    always @(\*)

        case (number)

        4'h0 : fnd=8'b00000011;

        4'h1 : fnd=8'b10011111;

        4'h2 : fnd=8'b00100101;

        4'h3 : fnd=8'b00001101;

        4'h4 : fnd=8'b10011001;

        4'h5 : fnd=8'b01001001;

        4'h6 : fnd=8'b01000001;

        4'h7 : fnd=8'b00011011;

        4'h8 : fnd=8'b00000001;

        4'h9 : fnd=8'b00011001;

        4'ha : fnd=8'b10001001;

        4'hb : fnd=8'b10000011;

        4'hc : fnd=8'b01100011;

        4'hd : fnd=8'b10000101;

        4'he : fnd=8'b01100001;

        4'hf : fnd=8'b01110001;

        default : fnd=8'b00000000;

            endcase

endmodule

진행중인 PC값의 변화를 살피기 위해 fnd 에 이를 표시하는 코드가 추가되었다.

**이는 동기인 이주영 학우로부터 도움을 받았다.**

    clock\_divider my\_divider(.clk(clk\_50MHz), .rst(reset), .clk\_operating(slow\_clock), .my\_clk(clk));

    advanced\_pipeline my\_pipeline(.clk(slow\_clock), .reset(reset), .result(ALU\_Result), .PC\_now(PC),

        .d\_Branch(Branch), .d\_Jump(Jump), .d\_RS\_RT\_Equal(RS\_RT\_Equal), .d\_Branch\_Taken(Branch\_Taken));

느린 counter를 만들기 위해, 수업 중 제시되었던 clock\_divider를 사용하고, 이것을 pipeline의 input으로 입력하였다.

구현 중 각 Stage를 넘어가면서 non-blocking으로 구현을 하지 않아 많은 시간을 디버깅에 할애하였다.

LED의 결과, 즉 ALU의 결과는 다음 값들을 반복한다.

상단의 표 중에서 Execute단만을 가져왔으며, 결과는 2진수로 LED 위에 표현될 것이다.

|  |  |
| --- | --- |
| PC\_Now | Execute |
| 00000 | All Zero |
| 00100 | All Zero |
| 01000 | 1100 |
| 01000 | 1010 |
| 01100 | 11100 |
| 10000 | 00000 |
| 10100 | 11111 |
| 00000 | 00000 (반복) |
| 00100 | 00000 |

첨부하는 추가 TB 파일들은 다음과 같다.

tb\_forwarding\_unit : forwarding unit을 위한 test bench

tb\_hazard\_control : hazard detection unit을 위한 test bench