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

1. 전체 코드 첨부

(디버깅을 위한 output은 주석 처리되어 있음)

`timescale 1ns / 1ps

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

*// Company: Hanyang Univ.*

*// Engineer: Kim Soo Young*

*//*

*// Create Date:    13:14:17 06/02/2021*

*// Design Name:*

*// Module Name:    single\_cycle\_main*

*// Project Name:*

*// Target Devices:*

*// Tool versions:*

*// Description:*

*//*

*// Dependencies:*

*//*

*// Revision:*

*// Revision 0.01 - File Created*

*// Additional Comments:*

*//*

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

module basic\_pipeline (clk, reset, result

*// PC\_now, Instruction\_now,*

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

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

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

    );

    input clk, reset;

    output [31:0] result; *// ALU result*

*// Debugging Outputs*

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

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

*// 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*

*// IF stage*

    wire [31:0] PC\_in;

    wire [31:0] PC\_out;

    wire [31:0] PC\_plus4;

    wire [31:0] IF\_instruction;

*// 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 ALUSrc, RegWrite;

    wire RegDst, Jump, Branch;

    wire MemRead, MemtoReg, MemWrite;

    wire [1:0] ALUOp;

*// 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;

*// 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 /////*

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

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

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

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

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

    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));

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

     .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;*

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

*//// 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));

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

    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));

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

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

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

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

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

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

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

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

     .ALUSrc\_in(ALUSrc), .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));

    N\_bit\_MUX #(32) alu\_input\_mux (.input0(EX\_Read\_data\_2), .input1(EX\_sign\_extended\_immi),

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

    ALU alu\_unit (.input1(EX\_Read\_data\_1), .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 immi\_Shifted = Shifted\_immi;*

*// assign ALU\_input\_1 = EX\_Read\_data\_1;*

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

*// assign Beq\_address = Branch\_addr;*

    assign result = ALU\_result;

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

*///////   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);

    input clk, reset;

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

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

    always @(posedge clk or negedge reset) begin

        if (!reset) begin

            PC\_plus4\_out <= 32'b0;

            instruction\_out <= 32'b0;

        end

        else 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 negedge 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 negedge 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 negedge 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);

    input clk, reset;

    input [31:0] PC\_in;

    output reg [31:0] PC\_out;

    always @ (posedge clk or negedge reset) begin

        if (!reset)

            PC\_out <= 0;

        else

            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 @(negedge reset) begin

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

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

        end

        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 [7:0];

    integer k;

    always @(posedge clk or negedge reset) begin

        if (!reset) begin

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

                mem[k] = 32'b0;

            end

            mem[3] = 32'b0011;

            mem[4] = 32'b0011;

            mem[6] = 32'h0000\_0040;

        end

        else if (RegWrite)

            mem[Write\_Register] = Write\_Data;

    end

    always @(negedge clk) 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, Jump, Branch, MemRead, MemtoReg, ALUOp, MemWrite, ALUSrc, RegWrite);

    input [5:0] OpCode;

    output [1:0] ALUOp;

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

*// 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]));

*// 000100 (beq)*

    assign Branch=(~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]));

*// 000010 (j)*

    assign Jump=(~OpCode[5])&(~OpCode[4])&(~OpCode[3])&(~OpCode[2])&(OpCode[1])&(~OpCode[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 [63:0];

    integer k;

    always @(\*) begin

        if (!reset) begin

            for (k = 0; k < 64; k = k + 1) begin

                mem[k] = 32'b0;

            end

            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

2. 실행 결과

텍스트이(가) 표시된 사진

자동 생성된 설명

위 코드는 상단 조건에 따라 다음과 같은 작업을 반복한다.

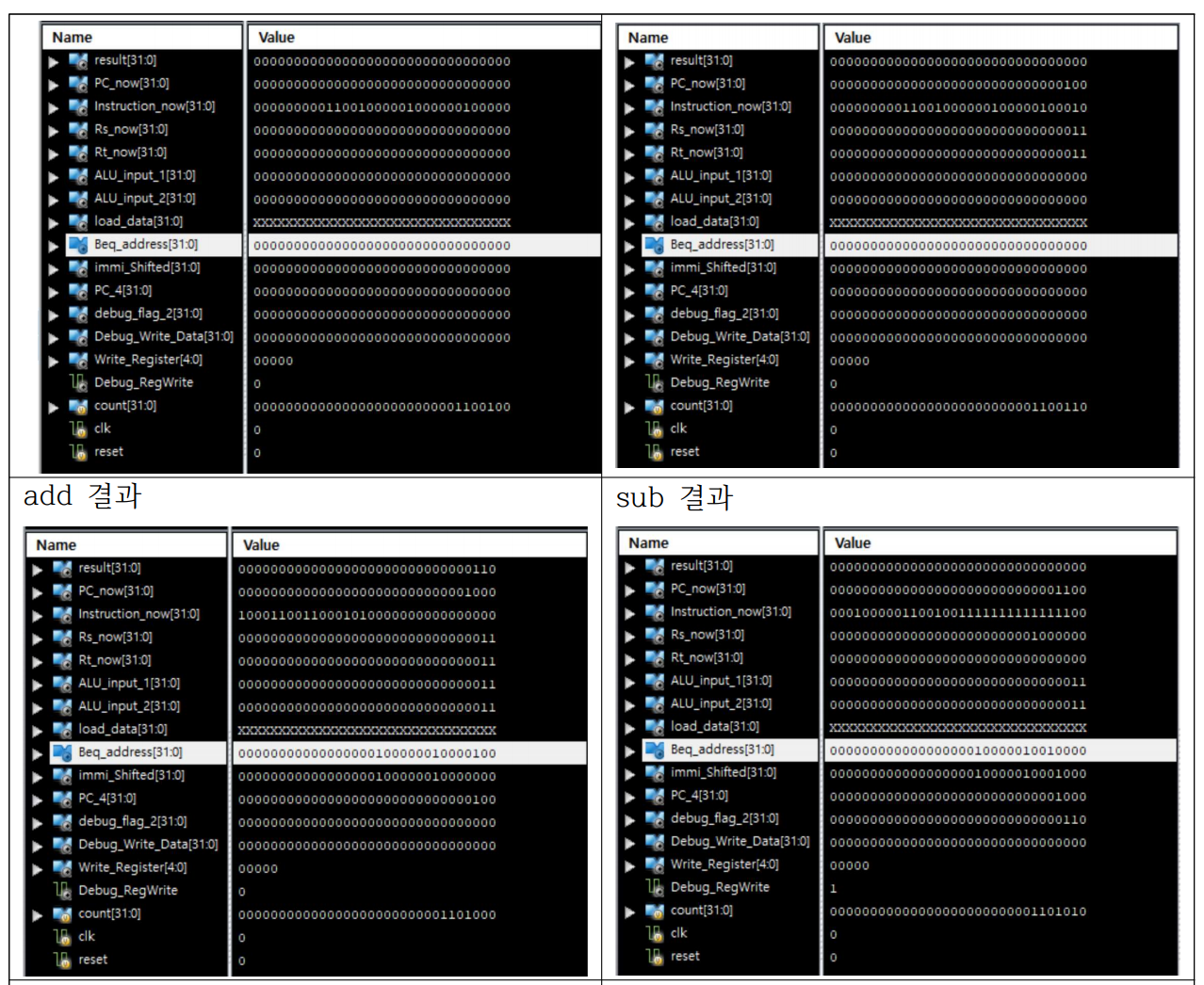
Beq에 의해 다시 Label로 되돌아가기 때문에, 같은 과정이 반복된다.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| PC Value | Fetch | Decode | Execute | Memory | Write Back |
| 00000 | Add |  |  |  |  |
| 00100 | Sub | Add |  |  |  |
| 01000 | lw | Sub | Add |  |  |
| 01100 | beq | lw | Sub | Add |  |
| 10000 | NOP | beq | lw | Sub | Add |
| 10100 | NOP | NOP | beq | lw | Sub |
| 11000 | NOP | NOP | NOP | beq | lw |
| 00000 | Add (반복) | NOP | NOP | NOP | beq |
| 00100 | Sub | Add (반복) | NOP | NOP | NOP |

각 매개변수들에 대한 설명은 다음과 같다.

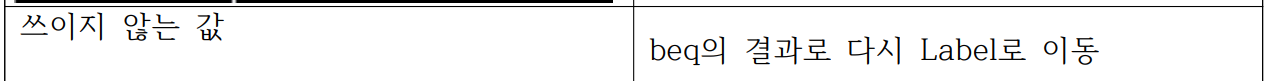
* Result : ALU 계산 결과 (후에 FPGA LED 값이 된다.)
* PC\_Now : 현 시점에서 Instruction Fetch에서 사용되는 PC를 나타냄
* Instruction\_Now : 현 시점에서 Instruction Decode에서 사용되는 Instruction을 나타냄
* Rs\_Now & Rt\_Now : Instruction\_Now를 통해 Register\_File에서 가져온 Rs,Rt 값이다. Falling Edge에서 최신화됨
* ALU\_Input\_1, ALU\_Input\_2 : Execute 단계에서 ALU의 input이 되는 두 피연산자
* Load\_data : Memory 단계에서 읽어온 데이터 (현재는 lw에서만 해당)
* Beq\_Address : Execute 단계에서 계산되는 branch address
* 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 : Write Back 단계에서 결정되는 RegWrite값

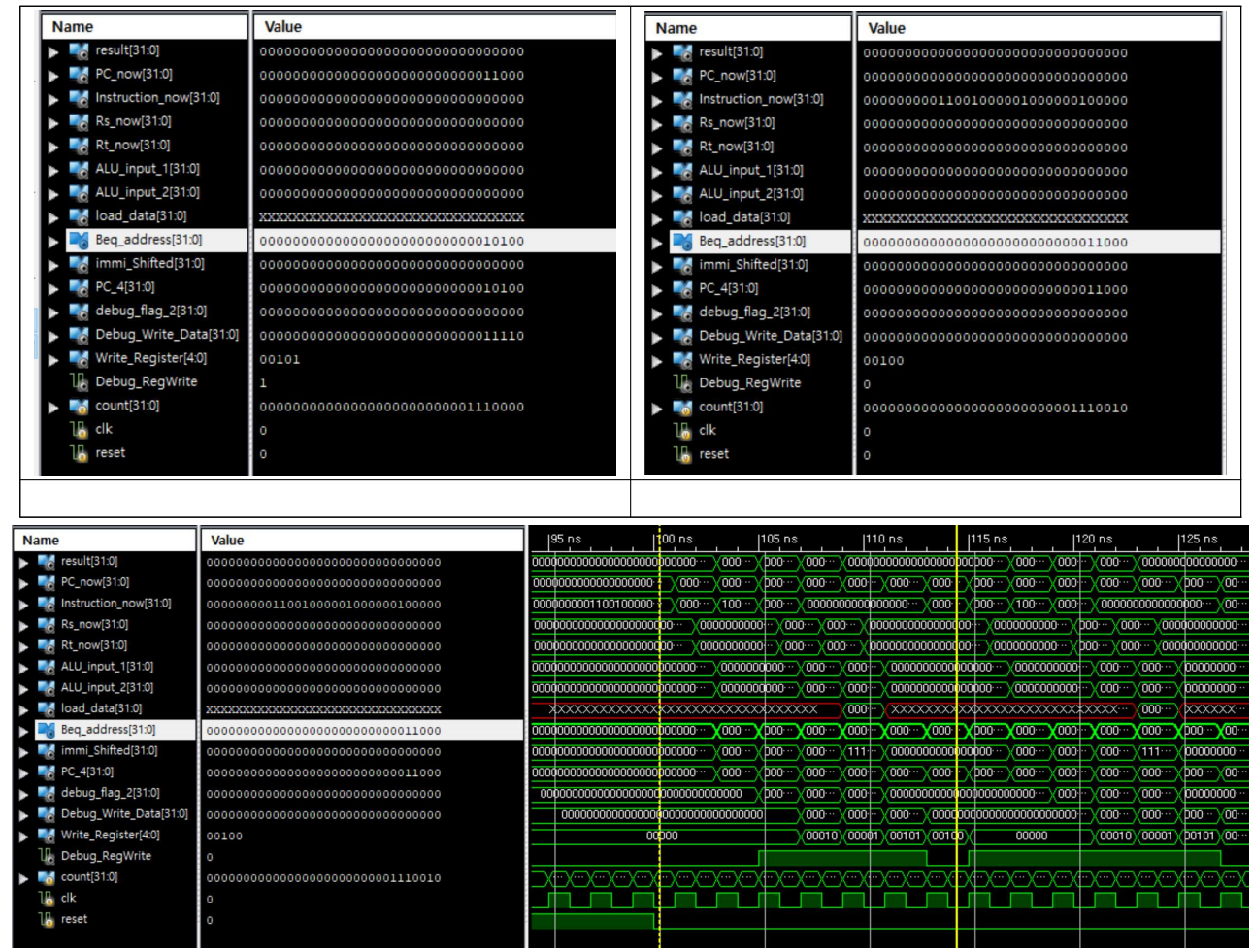
디버깅 Flag들을 포함하여 실행시킨 각 단계에서의 Waveform은 다음과 같다.



텍스트, 스크린샷, 검은색이(가) 표시된 사진

자동 생성된 설명





각 Pipeline 단계별로 결과를 구조화해보면 다음과 같다.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| PC\_Now | Fetch | Decode | Execute | Memory | Write Back |
| 00000 | Instruction\_Now = Add |  |  |  |  |
| 00100 | Instruction\_Now = Sub | Rs=$3 / Rt=$4 |  |  |  |
| 01000 | Instruction\_Now = lw | Rs=$3 / Rt=$4 | Result=$3+$4=6 |  |  |
| 01100 | Instruction\_Now = beq | Rs=$6 / Rt=$5 | Result=$3-$4=0 | Add이므로 Nop |  |
| 10000 | NOP | Rs=$3 / Rt=$4 | Result=$6+0=0x40 | Sub이므로 Nop | $2에 결과 6저장요청 |
| 10100 | NOP | NOP | Result=$3-$4 | Mem[0x40] Read | $1에 결과 0저장요청 |
| 11000 | NOP | NOP | NOP | Branch 주소 계산 (PCSrc 변경) | $5에 결과 30저장요청 |
| 00000 | Branch & Add (반복) | NOP | NOP | NOP | Beq이므로 Nop |
| 00100 | 계속 반복 | Rs=$3 / Rt=$4 | NOP | NOP | NOP |

3. 코드 설명

필요하다고 판단되는 Unit에 대한 코드 설명을 추가한다.

* Program Counter

*// PC*

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

    input clk, reset;

    input [31:0] PC\_in;

    output reg [31:0] PC\_out;

    always @ (posedge clk or negedge reset) begin

        if (!reset)

            PC\_out <= 0;

        else

            PC\_out <= PC\_in;

    end

endmodule

다음 PC로 업데이트, 혹은 reset에 의해 업데이트를 중지하는 기능을 갖는다.

다음 PC는 PC + 4 혹은, Jump/Branch Address가 될 수 있다.

* Instruction Memory

*// 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 @(negedge reset) begin

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

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

        end

        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

PC에 따라 새로운 Instruction을 불러온다.

이때, PC는 word단위로 update되기에, 하위 2비트는 사용하지 않게 함으로 사용 메모리를 줄일 수 있다.

Instruction들을 Hard Coding 되어 있다.

* IF\_ID\_Stage

*// IF/ID stage register*

module IF\_ID\_Stage\_Reg (clk, reset, PC\_plus4\_in, PC\_plus4\_out,

    instruction\_in, instruction\_out);

    input clk, reset;

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

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

    always @(posedge clk or negedge reset) begin

        if (!reset) begin

            PC\_plus4\_out <= 32'b0;

            instruction\_out <= 32'b0;

        end

        else begin

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

            instruction\_out <= instruction\_in;

        end

    end

endmodule

Fetch => Decode로 넘어가면서, 필요한 데이터들을 넘겨주는 역할을 한다.

Reset시, 데이터는 모두 초기화된다.

Decode => Execute, Execute => Memory, Memory => Write Back 모두 비슷한 구조를 갖기에 여기에서의 설명과 동일하다.

위 Unit들을 통해 Instruction Fetch는 다음과 같이 구성된다.

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

*//// Instruction Fetch stage /////*

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

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

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

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

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

    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));

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

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

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

다른 Pipeline 단계들도 비슷하게 구성되며, 이는 주어진 Diagram에 따라 배치 및 연결된다.

* Register File

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 [7:0];

    integer k;

    always @(posedge clk or negedge reset) begin

        if (!reset) begin

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

                mem[k] = 32'b0;

            end

            mem[3] = 32'b0011;

            mem[4] = 32'b0011;

            mem[6] = 32'h0000\_0040;

        end

        else if (RegWrite)

            mem[Write\_Register] = Write\_Data;

    end

    always @(negedge clk) begin

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

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

    end

endmodule

레지스터의 초기화 및 Read/Write를 담당하는 Unit이다.

Posedge와 Negedge로 나뉘어 동작하며, Posedge시는 Register Write를, Negedge시는 Register Read를 수행한다.

이를 통해 Write로 Register를 업데이트함과 동시에 Read하여 사용할 수 있게 된다.

* Control

*// Control Path*

module Control (OpCode, RegDst, Jump, Branch, MemRead, MemtoReg, ALUOp, MemWrite, ALUSrc, RegWrite);

    input [5:0] OpCode;

    output [1:0] ALUOp;

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

*// 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]));

*// 000100 (beq)*

    assign Branch=(~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]));

*// 000010 (j)*

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

endmodule

Instruction의 Opcode를 읽고, 해당하는 control signal을 생성한다.

각 경우에 대해서는 다음 표와 같이 매칭된다.

테이블이(가) 표시된 사진

자동 생성된 설명

다음으로 ID\_EX\_Stage를 거쳐 Execute 단계에 진입한다.

    N\_bit\_MUX #(32) alu\_input\_mux (.input0(EX\_Read\_data\_2), .input1(EX\_sign\_extended\_immi),

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

ALU에 진입하기 전, 두번째 피연산자를 선택하기 위해 MUX가 추가된 모습을 볼 수 있다.

이는 R-type과 I-type이 차이를 가질 것이다.

* ALU\_Control

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

Opcode로 구분할 수 없는 연산자들을 다시 한 번 구분하기 위해 사용된다.

복잡해 보이지만 아래 표를 그대로 구현한 것일 뿐이다.

테이블이(가) 표시된 사진

자동 생성된 설명

* ALU

*// 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

ALU\_Control의 output을 받아 해당하는 연산을 수행한다.

And, Or, Add, Sub, Slt 를 구현하였다.

* Branch 주소 계산

    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));

I-type의 immediate 부분만을 추려 sign extension 한 뒤, shift 2 bit를 수행한다.

이후, 해당 Instruction Fetch 당시의 PC + 4와 Add 하여 Branch Address를 계산한다.

다음으로 Memory 단계를 살펴본다.

* Data Memory

*// 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 [63:0];

    integer k;

    always @(\*) begin

        if (!reset) begin

            for (k = 0; k < 64; k = k + 1) begin

                mem[k] = 32'b0;

            end

            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

내부 Memory에 저장되어 있는 Data의 Read/Write를 수행하며, 이전 과제와 큰 차이는 없다.

* PCSrc를 위한 and

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

Diagram과 같이 Control로부터의 Branch와 ALU의 뺄셈 결과인 zero를 and 하여 PCSrc를 업데이트한다.

다음으로 Write Back 단계를 살핀다.

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

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

Basic Pipeline의 Write Back 단계에서는 Mux만을 갖는다.

MemtoReg 값에 따라 다시 Register\_File로 저장할 데이터를 전송할 때, ALU의 결과를 저장할 지, 혹은 Data로 부터 가져온 데이터를 저장할 지 선택하게 된다.

추가적으로 언급하지는 않았지만 자주 사용되는 Unit들은 다음과 같다.

* N\_Bit\_MUX : 32비트, 5비트 등 다양한 output을 갖는 mux가 필요한데, 이를 따로 만들지 않고, parameter를 사용한 N\_Bit\_MUX를 구현하였다.
* ALU\_Add\_only : 오로지 덧셈만을 수행하는 ALU이다.
* Sign\_Extension : 최상위 비트에 따라 Sign\_Extension을 수행한다.

다음으로 FPGA 구동을 위한 Startup 코드를 보자.

* Startup

제공되는 예제와 차이나는 부분만을 첨부하였다.

    assign clk\_slow = clk\_operating;

    basic\_pipeline my\_pipeline(.clk(clk\_slow), .reset(reset), .result(ALU\_Result));

*//Counter*

    always @(posedge clk\_50MHz or negedge reset) begin

        if(!reset) begin

            counter<=0;

            clk\_operating <= 1'b0;

        end

        else begin

            counter<=counter+1;

            clk\_operating <= ((counter == 25'b1\_1111\_1111\_1111\_1111\_1111\_1111) ? ~clk\_operating: clk\_operating );

        end

    end

*// // Test Led*

*// always @(posedge counter[24] or negedge reset) begin*

*//  if (!reset) begin*

*//      led <= 16'b0;*

*//  end*

*//  else begin*

*//      // 7 9 10 ... 23*

*//      led<= ~test\_led;*

*//      test\_led <= ~test\_led;*

*//  end*

*// end*

*//Led*

*//always @(posedge counter[24] or negedge reset) begin*

    always @(posedge clk\_50MHz or negedge reset) begin

        if (!reset) begin

            led <= 16'b0;

        end

        else begin

*// 7 9 10 ... 23*

            led<= ALU\_Result[15:0];

        end

    end

느린 clock을 사용하기 위해, 2^25비트마다 반복되는 clk\_slow를 새로 선언하고, 해당 값을 pipeline에 연결한 모습을 볼 수 있다.

더불어, led에는 ALU결과의 하위 16비트를 매칭해 두었다.

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 | Result=$3+$4=6 |
| 01100 | Result=$3-$4=0 |
| 10000 | Result=$6+0=0x40 |
| 10100 | Result=$3-$4=0 |
| 11000 | All Zero |
| 00000 | All Zero |
| 00100 | All Zero |

마지막으로 첨부하는 TB 파일들은 다음과 같다.

tb\_basic\_pipeline : clk, reset, result만을 output으로 갖는 test bench

tb\_basic\_pipeline\_simple\_ver : 각종 디버깅 output들을 모두 output으로 갖는 testbench

tb\_data\_memory : Data Memory를 위한 testbench

tb\_control : Control을 위한 testbench

tb\_register\_file : Register File을 위한 testbench

tb\_instruction\_memory : instruction memory를 위한 testbench