Pipelined CPU Project Report

Omar Rady

December 1st, 2023

CMPEN 331 Final Project with Extra Credit Report

A Pipelined Control Processing Unit with Forwarding

Dr. Mohamed Almekkawy

Device: xczu7eg-ffvf1517-1LV-i

Final Project Extra Credit Report

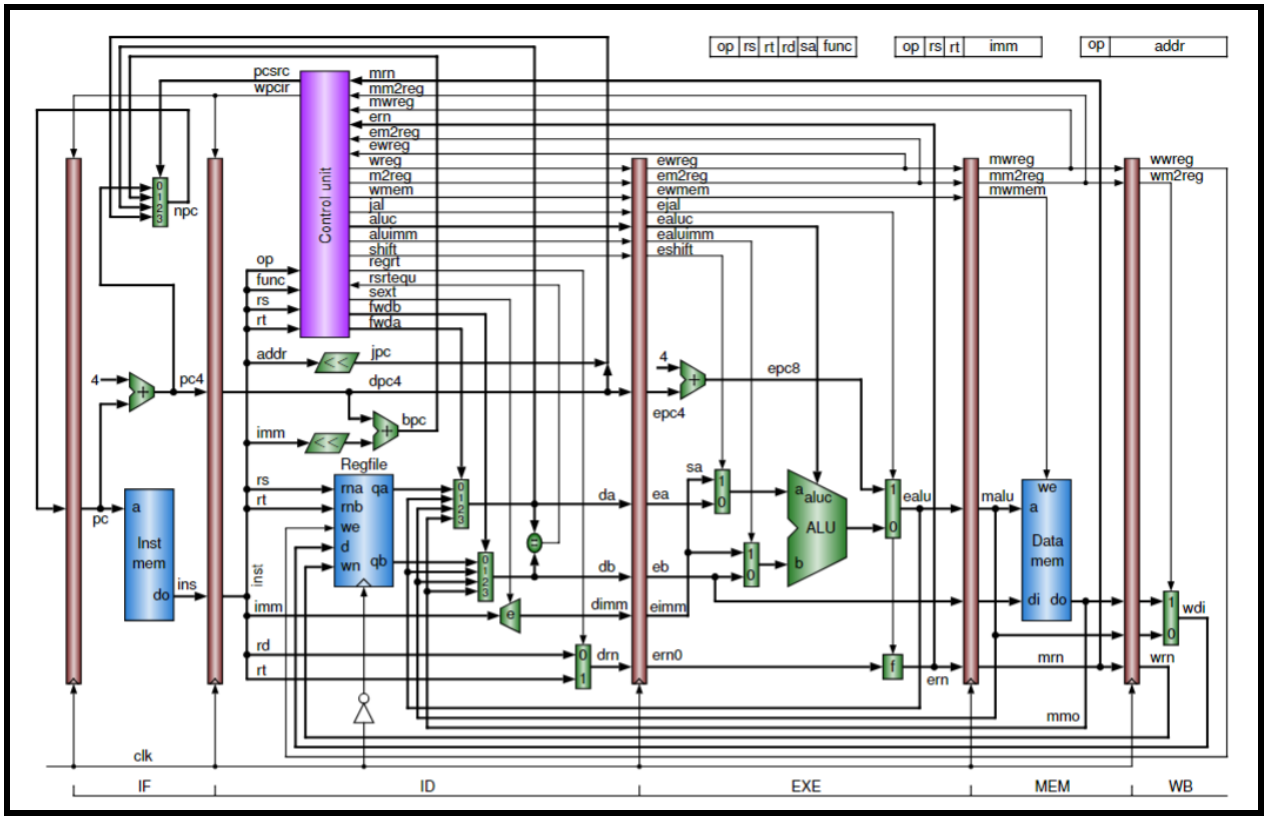
Table of Contents:

Abstract:

This report details the development of an advanced Control Processing Unit (CPU) as part of the Final Project Extra Credit, designed for the XC7Z010-CLG400-1 device. The project expands upon traditional CPU architecture by implementing enhanced features including delayed branching, jumping, and comprehensive support for load/store (LW/SW) operations, I-format, and all R-format instructions, including specific shifting commands like sll. The integration of these features allows for a more versatile and efficient processing unit, capable of handling a broader range of operations with improved performance and flexibility. This extended functionality not only demonstrates a deep understanding of CPU architecture but also aligns with contemporary needs in computer organization and architecture.

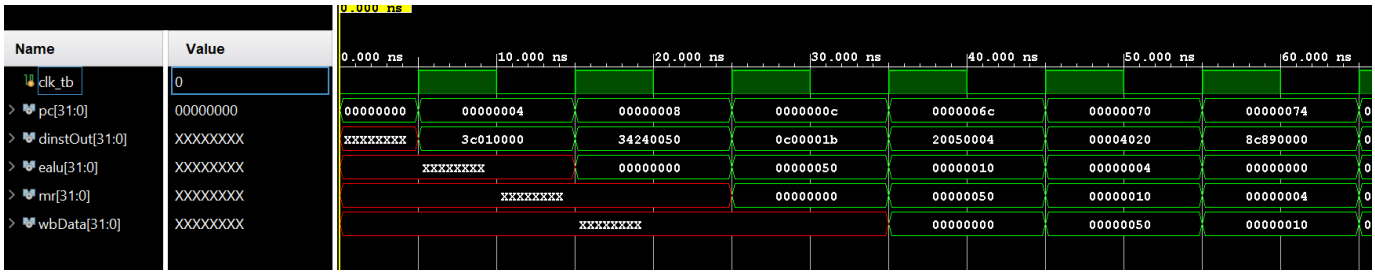
Introduction:

In the rapidly evolving field of computer organization, the need for versatile and efficient CPU architectures is paramount. The project presented here introduces a sophisticated CPU design. The CPU's architecture is designed to support delayed branching and jumping, crucial for optimizing instruction pipeline flow and reducing cycle penalties typically associated with branch instructions. The heart of the CPU is its pipelined architecture, which divides the processing into distinct stages, each responsible for a part of the instruction execution process. This pipelining not only increases the throughput of the CPU but also ensures that each instruction is efficiently processed through the system. The addition of forwarding mechanisms and stall control logic addresses the challenges of data hazards and pipeline interruptions, thereby maintaining smooth operation and maximizing performance. In conclusion, the CPU design presented in this report exemplifies a comprehensive approach to modern CPU architecture. Below is a detailed description of what I did to achieve this architecture.

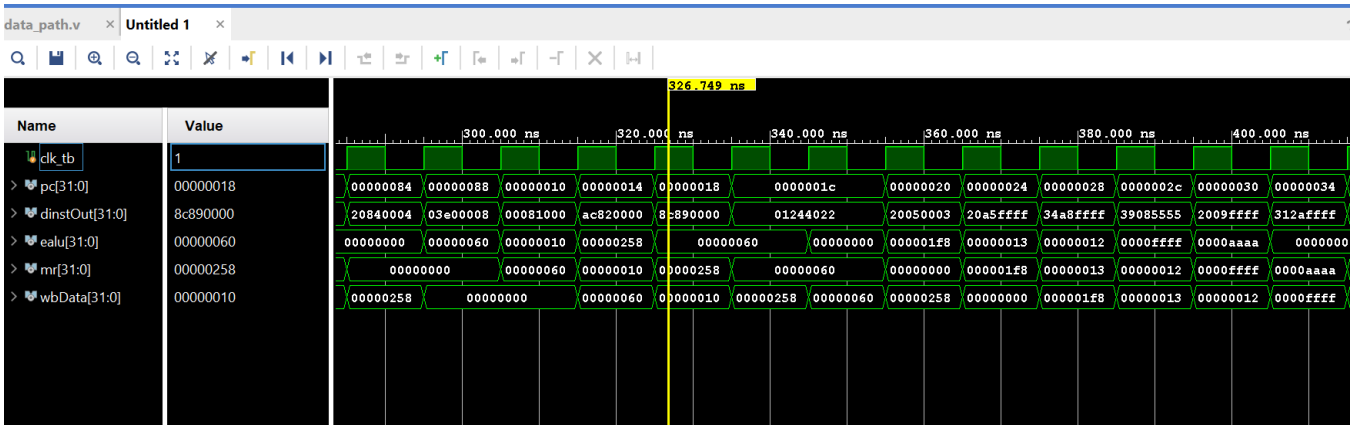


In order to achieve the pipeline architecture with forwarding and delayed branches I had to add some modules on top of the existing architecture of a cpu. For forwarding, I added 2 new muxes that check for forwarding conditions and if they hold true it forwards the value. For delayed branching, shifting and jumping I had to add 8 more modules. The first is a mux that decides what the next program counter value is, which of course changes than the normal +4 when dealing with branches and jumping. Additionally, we needed an immediate extender with concatenation for the jump functionality. When it comes to branching, it also needed it's own adder module and in addition a comparator module in the ID stage to limit the stalls needed. We also added jal muxes to deal with the Jal operation's special needs. Moreover, since our CPU now supports a variety of commands, including shifting, we needed another alu mux to pass in the correct variable to the ALU unit. Finally, we expanded the CPU and ALU operations to allow for the calculation of our delayed branching, jumping, shifting, I-formats, and all R-formats.

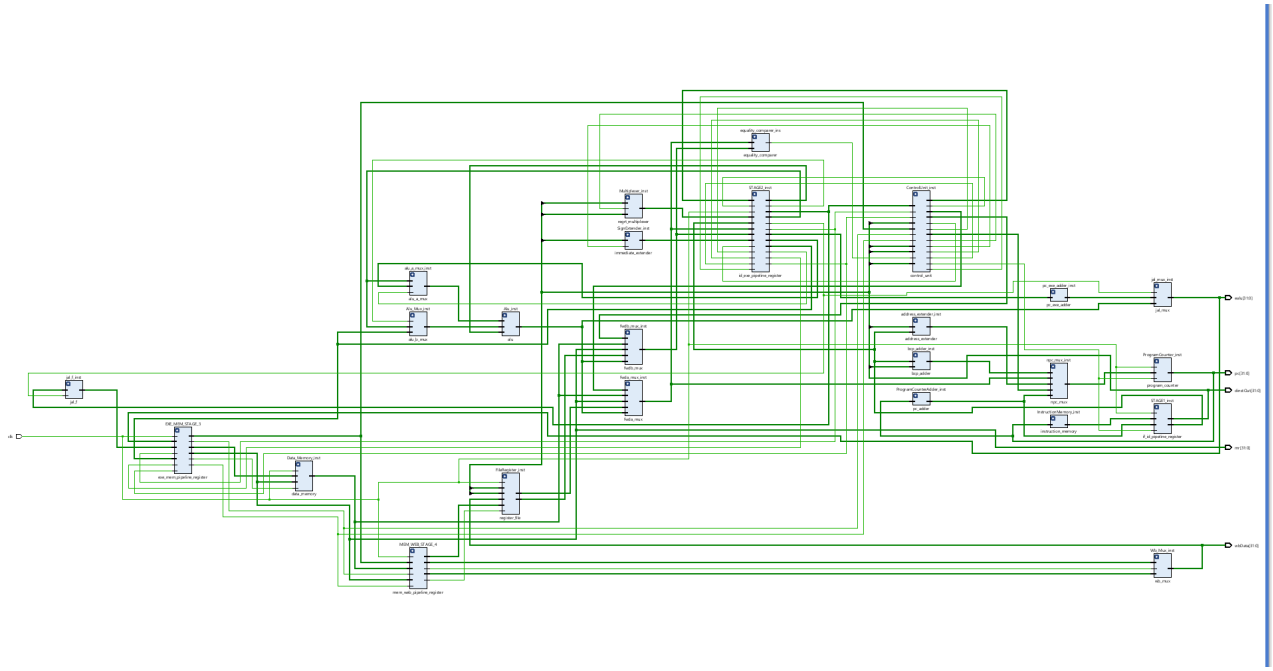
Waveform 1:



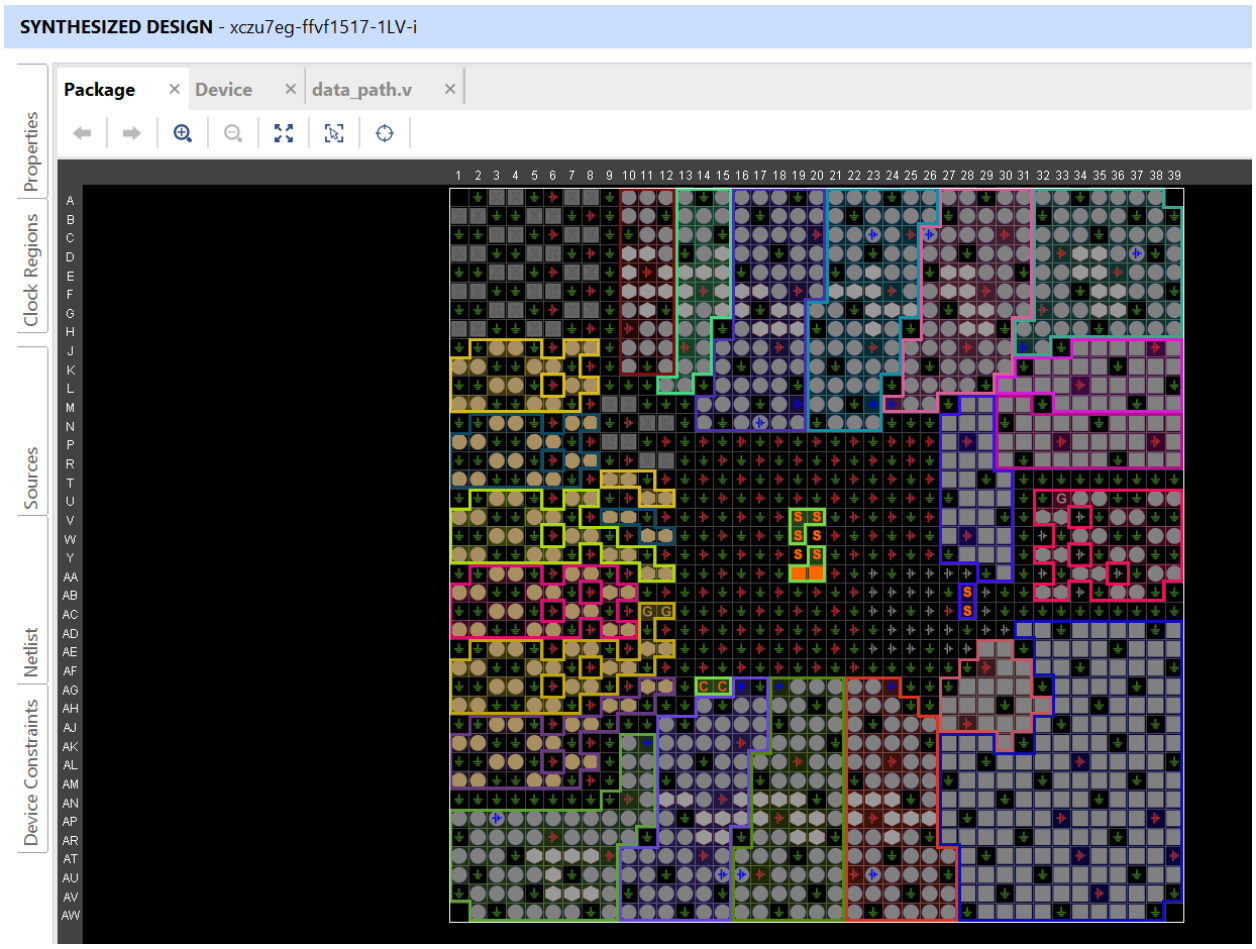
Waveform 2: (ram[5'h18] should be 0x00000258, the sum stored by sw instruction, which is instruction 10)



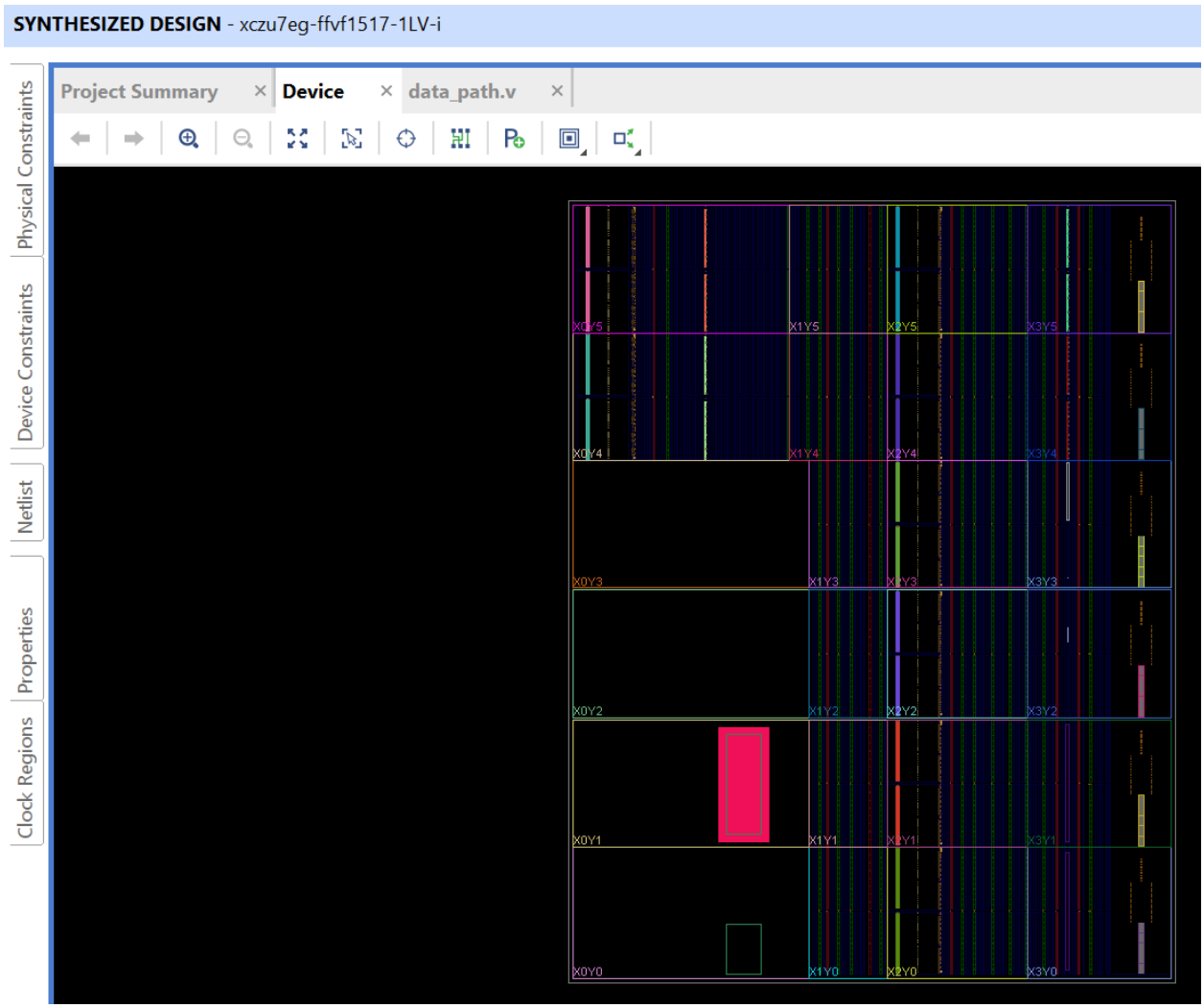
Design schematics from the RTL of the design:



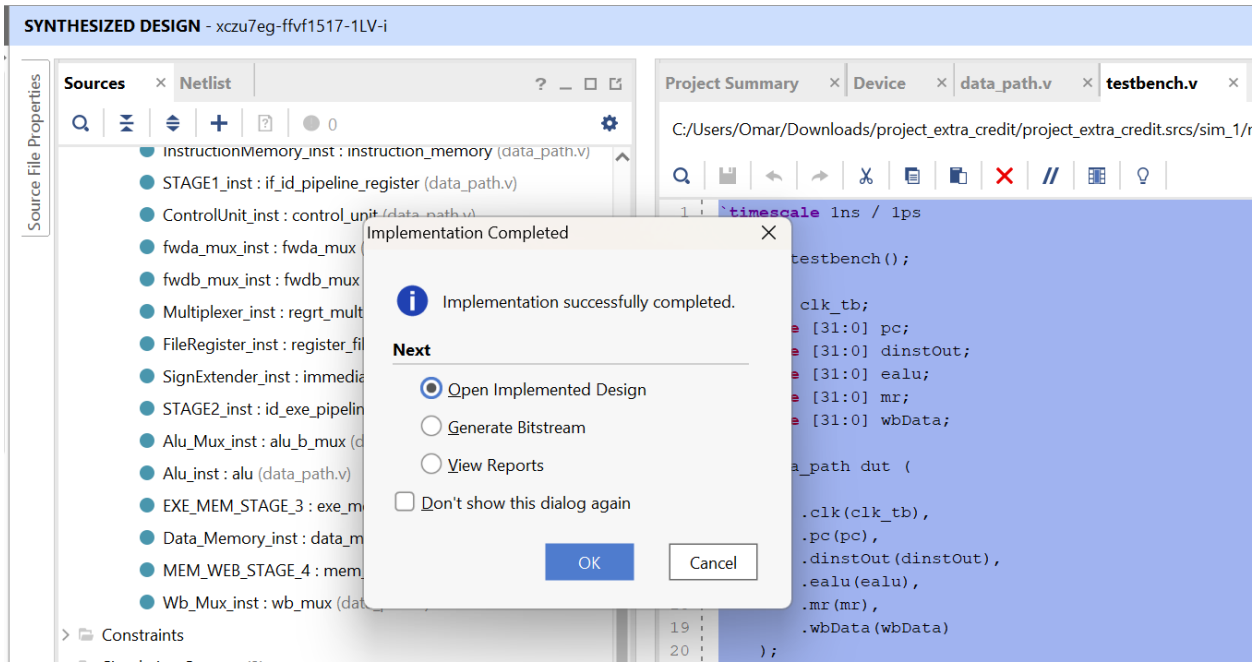
I/O planning: (large device with lots of I/O)

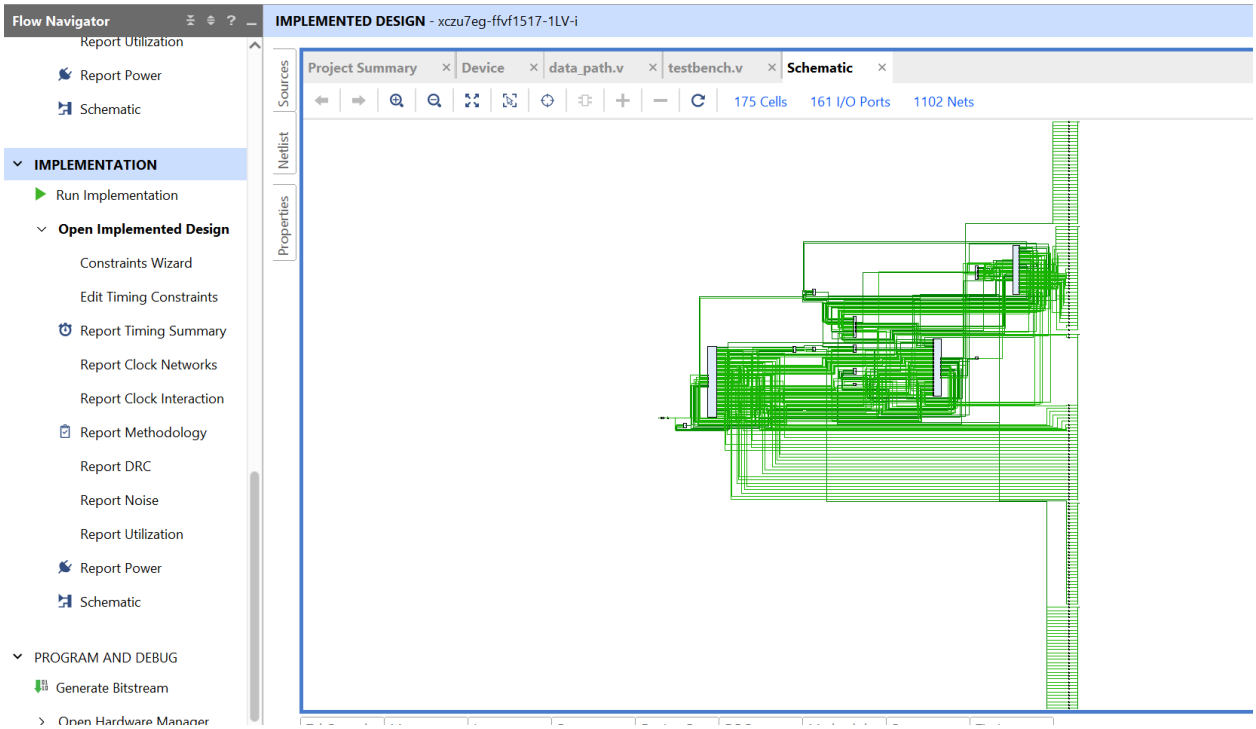


Floor Planning: (large device with lots of I/O)



Pass Implementation proof:

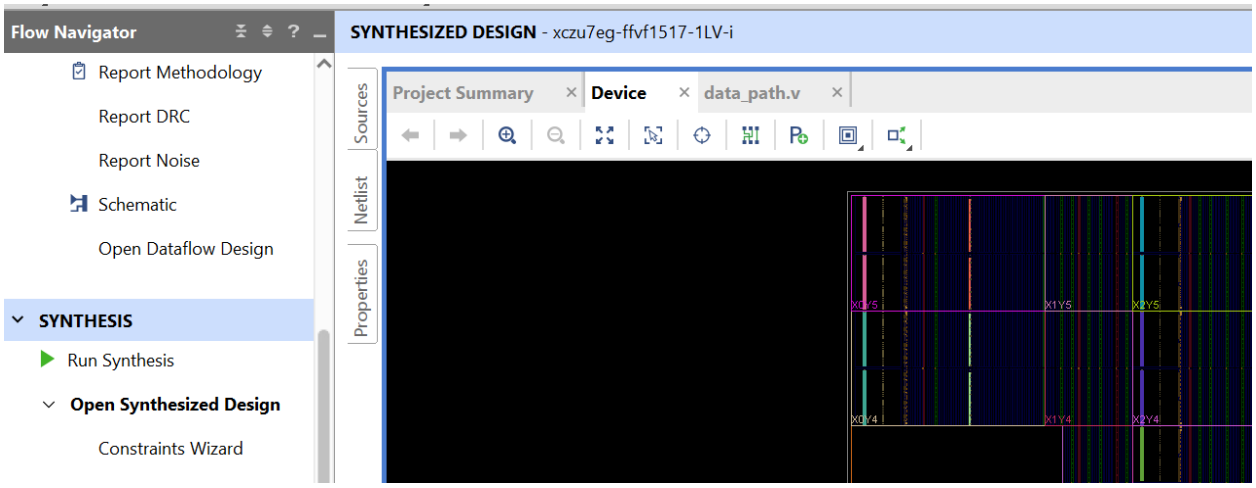




Pass compilation proof:

(proved with waveforms and synthesis and implementation working)

Pass synthesis proof:



<u>VERILOG SOURCE CODE:</u>

`timescale 1ns / 1ps

//

// Company: Penn State

// Engineer: Omar Rady

//

// Create Date: 11/10/2023 06:05:55 PM

// Module Name: cpu data path

// Project Name: Final Project with Extra Credit

// Target Devices: XC7Z010-CLG400-1

// Description: Lab5

//

// Dependencies:

```
//
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////

// MY DATA PATH MODULE WHERE modules are intiaited.
module data_path(
    input clk,
    output wire [31:0] pc,
    output wire [31:0] dinstOut,
    output wire [31:0] ealu,
    output wire [31:0] mr,
    output wire [31:0] wbData
 );

    // wires decoding the instruction
    wire [31:0] npc;
    wire [31:0] inst_out;
    wire [5:0] op;
    wire [4:0] rs;
    wire [4:0] rt;
    wire [4:0] rd;
```

```
wire [4:0] shamt;
wire [5:0] funct;
wire [15:0] imm;
wire [25:0] address;
assign op = dinstOut[31:26];
assign rs = dinstOut [25:21];
assign rt = dinstOut [20:16];
assign rd = dinstOut [15:11];
assign shamt = dinstOut [10:6];
assign funct = dinstOut [5:0];
assign imm = dinstOut [15:0];
assign address = dinstOut [25:0];

// Internal Wires within IF/ID
wire wreg;
wire wmem;
wire m2reg;
wire [3:0] aluc;
wire aluimm;
wire regrt;
wire [4:0] destReg;
wire [31:0] qa;
wire [31:0] qb;
```

```
wire [31:0] dqa;
wire [31:0] dqb;
wire [31:0] imm32;

// Internal Wires within ID/EXE
wire [31:0] b;
wire [31:0] a;
wire [1:0] fwda;
wire [1:0] fwdb;
wire ewreg;
wire em2reg;
wire ewmem;
wire [3:0] ealuc;
wire ealuimm;
wire [4:0] edestReg;
wire [31:0] eqa;
wire [31:0] eqb;
wire [31:0] eimm32;
wire [31:0] r;

// wires out of EXE/MEM
wire mwreg;
wire mm2reg;
```

```
wire mwmem;
wire [4:0] mdestReg;
wire [31:0] mqb;
wire [31:0] mdo;

// wires out of MEM/WB
wire wwreg;
wire wm2reg;
wire [4:0] wdestReg;
wire [31:0] wr;
wire [31:0] wdo;

// EXTRA CREDIT WIRES TO IMPLMENT SHIFT, JUMP, AND BRANCH
wire [1:0] pcsrc;
wire wpcir;
wire [31:0] bpc;
wire [31:0] jpc;
wire [31:0] pc4;
wire [31:0] dpc4;
wire [31:0] epc4;
wire [31:0] epc8;
wire rsrtequ;
wire sext;
```

```
    wire shift;
    wire jal;
    wire ejal;
    wire eshift;
    wire [4:0] edestReg2;

    // EXTRA CREDIT PROJECT MODULES FOR STALLS, DELAYED BRANCHES,
JUMPS, AND SHIFTS
    npc_mux npc_mux_inst(
        .pcsrc(pcsrc),
        .pc4(pc4),
        .bpc(bpc),
        .dqa(dqa),
        .jpc(jpc),
        .npc(npc)
    );
    address_extender address_extender_inst(
        .address(address),
        .dpc4(dpc4),
        .jpc(jpc)
    );

    bcp_adder bcp_adder_inst(
```

```
    .dpc4(dpc4),
    .imm(imm),
    .bpc(bpc)
);

equality_comparer equality_comparer_ins(
    .dqa(dqa),
    .dqb(dqb),
    .rsrtequ(rsrtequ)
);

pc_exe_adder pc_exe_adder_inst(
    .epc4(epc4),
    .epc8(epc8)
);

alu_a_mux alu_a_mux_inst(
    .eqa(eqa),
    .eimm32(eimm32),
    .eshift(eshift),
    .a(a)
);
```

```
jal_mux jal_mux_inst(
    .epc8(epc8),
    .r(r),
    .ejal(ejal),
    .ealu(ealu)
);

jal_f jal_f_inst(
    .edestReg(edestReg),
    .ejal(ejal),
    .edestReg2(edestReg2)
);

// ---------------------------------------------------------------------------

// ALL OTHER PREVIOUS OBJECT INSTANSIATIONS
program_counter ProgramCounter_inst (
    .clk(clk),
    .wpcir(wpcir),
    .npc(npc),
    .pc(pc)
);
```

```
pc_adder ProgramCounterAdder_inst (
    .pc(pc),
    .pc4(pc4)
);

instruction_memory InstructionMemory_inst (
    .pc(pc),
    .instOut(inst_out)
);

if_id_pipeline_register STAGE1_inst (
    .clk(clk),
    .wpcir(wpcir),
    .instOut(inst_out),
    .dInstOut(dinstOut),
    .pc4(pc4),
    .dpc4(dpc4)
);

control_unit ControlUnit_inst (
    .op(op),
    .func(funct),
    .wreg(wreg),
```

```
.m2reg(m2reg),
.wmem(wmem),
.aluc(aluc),
.aluimm(aluimm),
.regrt(regrt),
// PROJECT ADDTIONS
.rs(rs),
.rt(rt),
.mdestReg(mdestReg),
.mm2reg(mm2reg),
.mwreg(mwreg),
.edestReg(edestReg),
.em2reg(em2reg),
.ewreg(ewreg),
.fwdb(fwdb),
.fwda(fwda),
.wpcir(wpcir),
// Extra Credit
.pcsrc(pcsrc),
.rsrtequ(rsrtequ),
.sext(sext),
.shift(shift),
.jal(jal)
```

```
);

fwda_mux fwda_mux_inst(
    .fwda(fwda),
    .qa(qa),
    .r(r),
    .mr(mr),
    .mdo(mdo),
    .dqa(dqa)
);

fwdb_mux fwdb_mux_inst(
    .fwdb(fwdb),
    .qb(qb),
    .r(r),
    .mr(mr),
    .mdo(mdo),
    .dqb(dqb)
);

regrt_multiplexer Multiplexer_inst (
    .rt(rt),
    .rd(rd),
```

```
    .regrt(regrt),
    .destReg(destReg)
);

register_file FileRegister_inst (
    .rs(rs),
    .rt(rt),
    .qa(qa),
    .qb(qb),
    .wdestReg(wdestReg),
    .wbData(wbData),
    .wwreg(wwreg),
    .clk(clk)
);

immediate_extender SignExtender_inst (
    .imm(imm),
    .sext(sext),
    .imm32(imm32)
);

id_exe_pipeline_register STAGE2_inst (
    .clk(clk),
```

```
.wreg(wreg),
.ewreg(ewreg),
.m2reg(m2reg),
.em2reg(em2reg),
.wmem(wmem),
.ewmem(ewmem),
.aluc(aluc),
.ealuc(ealuc),
.aluimm(aluimm),
.ealuimm(ealuimm),
.destReg(destReg),
.edestReg(edestReg),
.dqa(dqa),
.eqa(eqa),
.dqb(dqb),
.eqb(eqb),
.imm32(imm32),
.eimm32(eimm32),
// EXTRA CREDIT VARIBLES
.dpc4(dpc4),
.epc4(epc4),
.jal(jal),
.ejal(ejal),
```

```
    .shift(shift),
    .eshift(eshift)
);

alu_b_mux Alu_Mux_inst (
    .eqb(eqb),
    .eimm32(eimm32),
    .ealuimm(ealuimm),
    .b(b)
);

alu Alu_inst (
    .a(a),
    .b(b),
    .ealuc(ealuc),
    .r(r)
);

exe_mem_pipeline_register EXE_MEM_STAGE_3(
    .clk(clk),
    .ewreg(ewreg),
    .em2reg(em2reg),
    .ewmem(ewmem),
```

```
    .edestReg2(edestReg2),
    .ealu(ealu),
    .eqb(eqb),
    .mwreg(mwreg),
    .mm2reg(mm2reg),
    .mwmem(mwmem),
    .mdestReg(mdestReg),
    .mr(mr),
    .mqb(mqb)
);

data_memory Data_Memory_inst(
    .clk(clk),
    .mr(mr),
    .mqb(mqb),
    .mwmem(mwmem),
    .mdo(mdo)
);

mem_web_pipeline_register MEM_WEB_STAGE_4(
    .clk(clk),
    .mwreg(mwreg),
    .mm2reg(mm2reg),
```

```
        .mdestReg(mdestReg),
        .mr(mr),
        .mdo(mdo),
        .wwreg(wwreg),
        .wm2reg(wm2reg),
        .wdestReg(wdestReg),
        .wr(wr),
        .wdo(wdo)
    );

    wb_mux Wb_Mux_inst(
        .wr(wr),
        .wdo(wdo),
        .wm2reg(wm2reg),
        .wbData(wbData)
    );

endmodule


// ---------- STAGE 1: INSTRUCTION FETCH ----------

module program_counter(
```

```
    input clk, input wire wpcir,
    input wire [31:0] npc, output reg [31:0] pc
);
    initial begin
        pc = 32'd0;
    end
    always @(posedge clk) begin
        if (wpcir) begin pc <= npc; end
    end
endmodule



module instruction_memory(input wire [31:0] pc, output reg [31:0] instOut); // CHANGE TO
LW INSTRUCTIONS
    wire [31:0] rom [0:63];
    // rom[word_addr] = instruction // (pc) label instruction
     assign rom[6'h00] = 32'h3c010000; // (00) main: lui $1, 0
     assign rom[6'h01] = 32'h34240050; // (04) ori $4, $1, 80
     assign rom[6'h02] = 32'h0c00001b; // (08) call: jal sum
     assign rom[6'h03] = 32'h20050004; // (0c) dslot1: addi $5, $0, 4
     assign rom[6'h04] = 32'hac820000; // (10) return: sw $2, 0($4)
     assign rom[6'h05] = 32'h8c890000; // (14) lw $9, 0($4)
     assign rom[6'h06] = 32'h01244022; // (18) sub $8, $9, $4
```

```
assign rom[6'h07] = 32'h20050003; // (1c) addi $5, $0, 3

assign rom[6'h08] = 32'h20a5ffff; // (20) loop2: addi $5, $5, -1

assign rom[6'h09] = 32'h34a8ffff; // (24) ori $8, $5, 0xffff

assign rom[6'h0a] = 32'h39085555; // (28) xori $8, $8, 0x5555

assign rom[6'h0b] = 32'h2009ffff; // (2c) addi $9, $0, -1

assign rom[6'h0c] = 32'h312affff; // (30) andi $10,$9,0xffff

assign rom[6'h0d] = 32'h01493025; // (34) or $6, $10, $9

assign rom[6'h0e] = 32'h01494026; // (38) xor $8, $10, $9

assign rom[6'h0f] = 32'h01463824; // (3c) and $7, $10, $6

assign rom[6'h10] = 32'h10a00003; // (40) beq $5, $0, shift

assign rom[6'h11] = 32'h00000000; // (44) dslot2: nop

assign rom[6'h12] = 32'h08000008; // (48) j loop2

assign rom[6'h13] = 32'h00000000; // (4c) dslot3: nop

assign rom[6'h14] = 32'h2005ffff; // (50) shift: addi $5, $0, -1

assign rom[6'h15] = 32'h000543c0; // (54) sll $8, $5, 15

assign rom[6'h16] = 32'h00084400; // (58) sll $8, $8, 16

assign rom[6'h17] = 32'h00084403; // (5c) sra $8, $8, 16

assign rom[6'h18] = 32'h000843c2; // (60) srl $8, $8, 15

assign rom[6'h19] = 32'h08000019; // (64) finish: j finish

assign rom[6'h1a] = 32'h00000000; // (68) dslot4: nop

assign rom[6'h1b] = 32'h00004020; // (6c) sum: add $8, $0, $0

assign rom[6'h1c] = 32'h8c890000; // (70) loop: lw $9, 0($4)

assign rom[6'h1d] = 32'h01094020; // (74) stall: add $8, $8, $9
```

```
    assign rom[6'h1e] = 32'h20a5ffff; // (78) addi $5, $5, -1
    assign rom[6'h1f] = 32'h14a0fffc; // (7c) bne $5, $0, loop
    assign rom[6'h20] = 32'h20840004; // (80) dslot5: addi $4, $4, 4
    assign rom[6'h21] = 32'h03e00008; // (84) jr $31
    assign rom[6'h22] = 32'h00081000; // (88) dslot6: sll $2, $8, 0
  always @(*) begin
    instOut = rom[pc / 4];
  end
endmodule


module pc_adder(input wire [31:0] pc, output reg [31:0] pc4);
  always @(*) begin
    pc4 = pc + 32'd4;
  end
endmodule


module if_id_pipeline_register(
  input clk,
  input wire wpcir,
  input wire [31:0] instOut,
  output reg [31:0] dInstOut,
```

```
    input wire [31:0] pc4,
    output reg [31:0] dpc4
);
    always @(posedge clk) begin
        if (wpcir) begin
            dInstOut <= instOut;
            dpc4 <= pc4;
        end
    end
endmodule



module npc_mux( // EXTRA CREDIT PROJECT MODULES FOR DELAYED BRANCHES
    input wire [1:0] pcsrc,
    input wire [31:0] pc4,
    input wire [31:0] bpc,
    input wire [31:0] dqa,
    input wire [31:0] jpc,
    output reg [31:0] npc
);
    always @(*) begin
        if(pcsrc==0) begin npc = pc4; end
        if(pcsrc==1) begin npc = bpc; end
```

```
        if(pcsrc==2) begin npc = dqa; end
        if(pcsrc==3) begin npc = jpc; end
        end
endmodule


// ---------------------- STAGE 2: INSTRUCTION DECODE ----------------------

module control_unit(
    input wire [5:0] op,
    input wire [5:0] func,
    output reg wreg,
    output reg m2reg,
    output reg wmem,
    output reg [3:0]aluc,
    output reg aluimm,
    output reg regrt,
    // PROJECT ADDTIONS
    input wire [4:0] rs,
    input wire [4:0] rt,
    input wire [4:0] mdestReg,
    input wire mm2reg,
    input wire mwreg,
```

```
    input wire [4:0] edestReg,
    input wire em2reg,
    input wire ewreg,
    output reg [1:0] fwdb,
    output reg [1:0] fwda,
    output reg wpcir,
    // EXTRA CREDIT VARIBLES
    output reg [1:0] pcsrc,
    input wire rsrtequ,
    output reg jal,
    output reg shift,
    output reg sext
);
    initial begin wpcir = 1'b1; pcsrc = 2'b00; end
     // STALL LOGIC
    reg stall = 1'b0;
    reg i_rs = 1'b0;
    reg i_rt = 1'b0;
    always @(*) begin
        wpcir = 1'b1;
        shift = 0;
        if(op == 6'b000000) begin i_rs = 1'b1; i_rt = 1'b1; end
        if(op == 6'b100011) begin i_rs = 1'b1; i_rt = 1'b0; end
```

```
stall = ewreg & em2reg & (edestReg!=0) & (i_rs & (edestReg == rs) | i_rt & (edestReg ==
rt));
if(stall) begin // NEED TO PREVENT WRITE
    wpcir = 1'b0;
    wreg = 0;
    m2reg = 0;
    wmem = 0;
    aluc = 4'b0000;
end
else begin
case(op)
    6'b000000: begin // R-TYPE Instructions
        wreg = 1;
        m2reg = 0;
        wmem = 0;
        aluimm = 0;
        regrt = 0;
        jal = 0;
        pcsrc = 2'b00;
        case(func)
            6'b100000: aluc = 4'b0010; // add
            6'b100010: aluc = 4'b0110; // sub
            6'b100100: aluc = 4'b0000; // and
```

```
        6'b100101: aluc = 4'b0001; // or
        6'b100110: aluc = 4'b0011; // xor
        6'b000000: begin aluc = 4'b1010; shift = 1; end // sll
        6'b000010: begin aluc = 4'b1100; shift = 1; end // srl
        6'b000011: begin aluc = 4'b1011; shift = 1; end // sra
        6'b001000: pcsrc = 2'b10; // jr
        // Additional R-TYPE instructions if needed
    endcase
end
// ------------  I-FORMATS ------------
6'b001000: // addi command
begin
    wreg = 1;
    regrt = 1;
    m2reg = 0;
    wmem = 0;
    aluc = 4'b0010; // Add
    aluimm = 1;
    sext = 1; // Sign extend the immediate
    jal = 0;
    pcsrc = 2'b00;
end
6'b001100: // andi command
```

```
begin
    wreg = 1;
    regrt = 1;
    wmem = 0;
    m2reg = 0;
    aluc = 4'b0000; // And
    aluimm = 1;
    sext = 0;
    jal = 0;
    pcsrc = 2'b00;
end
6'b001101: // ori command
begin
    wreg = 1;
    regrt = 1;
    wmem = 0;
    m2reg = 0;
    aluc = 4'b0001; // OR
    aluimm = 1;
    sext = 0;
    jal = 0;
    pcsrc = 2'b00;
end
```

```
6'b001110: // xori command
begin
    wreg = 1;
    regrt = 1;
    wmem = 0;
    m2reg = 0;
    aluc = 4'b0011; // XOR
    aluimm = 1;
    sext = 0;
    jal = 0;
    pcsrc = 2'b00;
end
6'b100011: // lw command
begin
    wreg = 1;
    regrt = 1;
    m2reg = 1;
    wmem = 0;
    aluc = 4'b0010;
    aluimm = 1;
    sext = 1;
    jal = 0;
    pcsrc = 2'b00;
```

```
end
6'b101011: // sw command
begin
    wreg = 0;
    regrt = 1;
    m2reg = 0;
    wmem = 1;
    aluc = 4'b0010;
    aluimm = 1;
    sext = 1;
    jal = 0;
    pcsrc = 2'b00;
end
6'b000100: // beq command
begin
    wreg = 0;
    m2reg = 0;
    wmem = 0;
    aluc = 4'b0110; // Subtract for comparison
    aluimm = 0;
    jal = 0;
    pcsrc = rsrtequ ? 2'b01 : 2'b00; // Branch if rs and rt are equal
end
```

```
6'b000101: // bne command
begin
    wreg = 0;
    m2reg = 0;
    wmem = 0;
    aluc = 4'b0110; // Subtract for comparison
    aluimm = 0;
    jal = 0;
    pcsrc = rsrtequ ? 2'b00 : 2'b01; // Branch if rs and rt are not equal
end
6'b001111: // lui command
begin
    pcsrc = 2'b00;
    wreg = 1;
    m2reg = 0;
    wmem = 0;
    aluc = 4'b0100; // Operation for LUI
    aluimm = 1;
    sext = 0;
    jal = 0;
end
// ------- j format --------
6'b000010: // j command
```

```
        begin
            wreg = 0;
            m2reg = 0;
            wmem = 0;
            aluimm = 0;
            aluc = 4'b0000;
            pcsrc = 2'b11;
            jal = 0;
        end
        6'b000011: // jal command
        begin
            m2reg = 0;
            wmem = 0;
            aluimm = 0;
            aluc = 4'b0000;
            pcsrc = 2'b11;
            wreg = 1;
            jal = 1;
        end
    endcase end
end
// FINAL PROJECT FORWARDING UNIT
always @(*) begin
```

```
// Default no forwarding
fwda = 2'b00;
fwdb = 2'b00;

// Forwarding logic for ALU operand A (rs)
if (ewreg && (edestReg == rs) && !em2reg) begin
    // Forward from ALU result in EX stage
    fwda = 2'b01;
end else if (mwreg && (mdestReg == rs) && !mm2reg) begin
    // Forward from ALU result in MEM stage
    fwda = 2'b10;
end else if (mwreg && (mdestReg == rs) && mm2reg) begin
    // Forward from data memory (load-use hazard)
    fwda = 2'b11;
end

// Forwarding logic for ALU operand B (rt)
if (ewreg && (edestReg == rt) && !em2reg) begin
    // Forward from ALU result in EX stage
    fwdb = 2'b01;
end else if (mwreg && (mdestReg == rt) && !mm2reg) begin
    // Forward from ALU result in MEM stage
    fwdb = 2'b10;
```

```
        end else if (mwreg && (mdestReg == rt) && mm2reg) begin
            // Forward from data memory (load-use hazard)
            fwdb = 2'b11;
        end
    end
endmodule

module regrt_multiplexer(
input wire [4:0] rt,
input wire [4:0] rd,
input wire regrt,
output reg [4:0] destReg
);
    always @(*) begin
        if(regrt==0)
            begin
                destReg = rd;
            end
        if(regrt==1)
            begin
                destReg = rt;
            end
    end
```

```
endmodule


module register_file(
input wire [4:0] rs,
input wire [4:0] rt,
output reg [31:0] qa,
output reg [31:0] qb,
// UPDATED REG GETS ADDTIONAL PARAMETERS:
input [4:0] wdestReg,
input [31:0] wbData,
input wwreg,
input clk
);
    reg [31:0] register [0:31];
    integer i;
    initial begin
        for (i = 0; i < 32; i = i + 1) begin
            register[i] = 32'h00000000;
        end
    end
    always @(*)
        begin
            qa = register[rs];
```

```
        qb = register[rt];
      end
   always @(negedge clk)
    begin
      if(wwreg == 1) begin
        register[wdestReg] = wbData;
      end
    end
endmodule

module immediate_extender(input wire [15:0] imm, input wire sext, output reg [31:0] imm32);
   always @(*)
      begin
        if (sext)
          imm32 = {{16{imm[15]}}, imm};
        else
          imm32 = {16'b0, imm};
       end
endmodule

module id_exe_pipeline_register(
input clk,
input  wreg,          output reg ewreg,
```

```
input  m2reg,         output reg em2reg,
input  wmem,          output reg ewmem,
input  [3:0] aluc,    output reg [3:0] ealuc,
input  aluimm,        output reg ealuimm,
input [4:0] destReg, output reg [4:0] edestReg,
input [31:0] dqa,     output reg [31:0] eqa,
input [31:0] dqb,     output reg [31:0] eqb,
input [31:0] imm32,  output reg [31:0] eimm32,
// EXTRA CREDIT VARIBLES
input [31:0] dpc4, output reg [31:0] epc4,
input jal, output reg ejal,
input shift, output reg eshift
);
   always @(posedge clk)
      begin
         ewreg <= wreg;
         em2reg <= m2reg;
         ewmem <= wmem;
         ealuc <= aluc;
         ealuimm <= aluimm;
         edestReg <= destReg;
         eqa <= dqa;
         eqb <= dqb;
```

```
        eimm32 <= imm32;
        epc4 <= dpc4;
        ejal <= jal;
        eshift <= shift;
    end
endmodule

// FINAL PROJECT 2 NEW Muxes
module fwda_mux(
input wire [1:0] fwda,
input wire [31:0] qa,
input wire [31:0] r,
input wire [31:0] mr,
input wire [31:0] mdo,
output reg [31:0] dqa
);
    always @(*) begin
        if(fwda==0) begin dqa = qa; end
        if(fwda==1) begin dqa = r; end
        if(fwda==2) begin dqa = mr; end
        if(fwda==3) begin dqa = mdo; end
    end
endmodule
```

```
module fwdb_mux(
input wire [1:0] fwdb,
input wire [31:0] qb,
input wire [31:0] r,
input wire [31:0] mr,
input wire [31:0] mdo,
output reg [31:0] dqb
);
    always @(*) begin
        if(fwdb==0) begin dqb = qb; end
        if(fwdb==1) begin dqb = r; end
        if(fwdb==2) begin dqb = mr; end
        if(fwdb==3) begin dqb = mdo; end
    end
endmodule

// EXTRA CREDIT MODULES
module address_extender(
    input wire [25:0] address,
    input wire [31:0] dpc4,
    output reg [31:0] jpc
);
```

```
    always @(*)
        begin
            jpc = {dpc4[31:28], address[25:0], 2'b00};
        end
endmodule

module bcp_adder(
    input wire [31:0] dpc4,
    input wire [15:0] imm,
    output reg [31:0] bpc
);
    wire [31:0] imm_extended = {{16{imm[15]}}, imm};
    always @(*)
        begin
            bpc = dpc4 + (imm_extended << 2);
        end
endmodule

module equality_comparer(
    input wire [31:0] dqa,
    input wire [31:0] dqb,
    output reg rsrtequ
);
```

```
    always @(*)
        begin
            if(dqa == dqb) begin rsrtequ = 1; end
            else begin rsrtequ = 0; end
        end
endmodule

// ---------------------- STAGE 3: EXE Stage ----------------------

module alu_b_mux(
    input [31:0] eqb,
    input [31:0] eimm32,
    input ealuimm,
    output reg [31:0] b
);
   always @(*)
       begin
          if(ealuimm == 0)
            begin
                b = eqb;
            end
          if(ealuimm == 1)
            begin
```

```
            b = eimm32;
        end
    end
endmodule

module alu(
    input [31:0] a,
    input [31:0] b,
    input [3:0] ealuc,
    output reg [31:0] r
);
   always @(*)
    begin
      case (ealuc)
        4'b0010: r = a + b;                 // Add
        4'b0110: r = a - b;                 // Subtract
        4'b0000: r = a & b;                 // AND
        4'b0001: r = a | b;                 // OR
        4'b0011: r = a ^ b;                 // XOR
        4'b1010: r = b << a[4:0];           // Shift left logical (sll)
        4'b1100: r = b >> a[4:0];           // Shift right logical (srl)
        4'b1011: r = $signed(b) >>> a[4:0];   // Shift right arithmetic (sra)
        4'b0100: r = b << 16;               // Load Upper Immediate (lui)
```

```
        default: r = 32'b0;                  // Default output value
      endcase
    end
endmodule


module exe_mem_pipeline_register(
    input clk,
    // Inputs
    input ewreg,
    input em2reg,
    input ewmem,
    input [4:0] edestReg2,
    input [31:0] ealu,
    input [31:0] eqb,
    // Outputs
    output reg mwreg,
    output reg mm2reg,
    output reg mwmem,
    output reg [4:0] mdestReg,
    output reg [31:0] mr, // also called malu
    output reg [31:0] mqb
);
always @(posedge clk)
```

```
    begin
        mwreg <= ewreg;
        mm2reg <= em2reg;
        mwmem <= ewmem;
        mdestReg <= edestReg2;
        mr <= ealu;
        mqb <= eqb;
    end
endmodule

// EXTRA CREDIT MODULES
module pc_exe_adder(input wire [31:0] epc4, output reg [31:0] epc8);
    always @(*)
        begin
            epc8 = epc4 + 32'd4;
        end
endmodule

module alu_a_mux(
    input wire [31:0] eqa,
    input wire [31:0] eimm32,
    input wire eshift,
    output reg [31:0] a
```

```
);
    always @(*)
        begin
            if(eshift) begin a = {27'b0, eimm32[10:6]};  end
            else begin a = eqa; end
        end
endmodule

module jal_mux(
    input wire [31:0] epc8,
    input wire [31:0] r,
    input ejal,
    output reg [31:0] ealu
);
    always @(*)
        begin
            if(ejal) begin ealu = epc8; end
            else begin ealu = r; end
        end
endmodule

module jal_f(
    input wire [4:0] edestReg,
```

```
    input wire ejal,
    output reg [4:0] edestReg2
);
    always @(*)
        begin
            if(ejal) begin edestReg2 = 5'd31; end
            else begin edestReg2 = edestReg; end

        end
endmodule


// ---------------------- STAGE 4: MEMORY ----------------------

module data_memory(
    input clk,
    input [31:0] mr,
    input [31:0] mqb,
    input mwmem,
    output reg [31:0] mdo
);
    // MY MEMORY
    reg [31:0] ram [63:0];
```

```
    integer i;
    initial begin
        for (i = 0; i < 32; i = i + 1)
            ram[i] = 0;
        // ram[word_addr] = data // (byte_addr) item in data array
         ram[5'h14] = 32'h000000a3; // (50) data[0] 0 + a3 = a3
         ram[5'h15] = 32'h00000027; // (54) data[1] a3 + 27 = ca
         ram[5'h16] = 32'h00000079; // (58) data[2] ca + 79 = 143
         ram[5'h17] = 32'h00000115; // (5c) data[3] 143 + 115 = 258
         // ram[5'h18] should be 0x00000258, the sum stored by sw instruction
    end
    always @(*) begin
        mdo = ram[mr / 4];
    end
    always @(posedge clk) begin
        if(mwmem) begin
            ram[mr / 4] <= mqb;
        end
    end
endmodule

module mem_web_pipeline_register(
    input clk,       // Clock signal
```

```
    // Inputs
    input mwreg,       // Control signal for writing to the register
    input mm2reg,      // Control signal for choosing source for register write-back
    input [4:0] mdestReg, // Destination register address
    input [31:0] mr,   // Memory address for read
    input [31:0] mdo,  // Output data from memory
    // Outputs
    output reg wwreg,      // Control signal for writing to the register
    output reg wm2reg,     // Control signal for choosing source for register write-back
    output reg [4:0] wdestReg, // Destination register address for write-back
    output reg [31:0] wr,  // Register address for read-back
    output reg [31:0] wdo  // Data to write back to the register
);
    always @(posedge clk) begin
        wwreg    <= mwreg;
        wm2reg   <= mm2reg;
        wdestReg <= mdestReg;
        wr       <= mr;
        wdo      <= mdo;
    end
endmodule

// ---------------------- LAB 5: WB Modules ----------------------
```

```
module wb_mux(
    input [31:0] wr,
    input [31:0] wdo,
    input wm2reg,
    output reg [31:0] wbData
);
    always @(*) begin
        if(wm2reg == 1) begin
            wbData = wdo;
        end
        if(wm2reg == 0) begin
            wbData = wr;
        end
    end
endmodule
```

VERILOG TESTBENCH CODE:

```
`timescale 1ns / 1ps


module testbench();


    reg clk_tb;
    wire [31:0] pc;
```

```
    wire [31:0] dinstOut;
    wire [31:0] ealu;
    wire [31:0] mr;
    wire [31:0] wbData;

    data_path dut (

        .clk(clk_tb),
        .pc(pc),
        .dinstOut(dinstOut),
        .ealu(ealu),
        .mr(mr),
        .wbData(wbData)
    );

    initial begin
        clk_tb = 0;
    end
    always begin
        #5 clk_tb = !clk_tb;
    end
endmodule
```