**Final Project – CMPEN 331**

**Tanner Kleintop**

**Section 002**

**Dr. Almekkawy**

**Abstract:**

**Stage 1: Instruction Fetch (IF)**

**A diagram of a computer

Description automatically generated**

Here, in the first stage of the pipeline, this deals with generating the program counter (PC) from the clock signal generated in the upper module (testbench). This stage deals with the 4 modules not grayed out, everything grayed out will be discussed further for its respective section. The first module is our program counter. This module is responsible for controlling when the program counter updates to the next program counter, this occurs *only* on the positive edge of the clock signal also known as the rising edge. The program counter is started at an initial value of a decimal 100. Next, we have the program counter adder. This module is responsible for adding 4 to the program counter at any signal as the pace of the program counter being updated is in the program counter module as discussed before. Thirdly, we have the Instruction Memory (Inst mem in diagram). This module is responsible for storing the instructions that the cpu will be processing. Instructions for the final project start in memory slot 25 all the way to 29. For the final piece of this module, it is finished of with the IFID Pipeline. This is the “divider between the instruction fetch and instruction decode stage.” This module is responsible for passing the instruction memory out on the positive clock edge. Note, that all 5 pipelines will operate *only* on the positive clock edge.

**Stage 2: Instruction Decode (ID)**

**A diagram of a machine

Description automatically generated**

In the second stage of the pipeline, this stage is primarily responsible for decoding the instructions passed through the first stage. When the first instruction is passed into this stage (assuming no stalls), the second instruction will be fetched in the instruction fetch stage. In a higher-level module, the instruction will be broken into its respective pieces for each stage (rs, rt, rd, op, func, imm, etc.). Though the wire sizes used for each module match the size necessary for each of those instruction pieces. First, we have our immediate extender which simply takes the imm portion of the instruction out, these bits are the last 16 bits of any instruction. With those bits, they are passed into this module and extended to a 32-bit number where the most significant 16 bits will depend on the 16th bit of the least significant bits. If it’s a 1, the most significant 16 will be a 1 and if it’s a 0, the most significant will be a 0. Next up, we have the register file (blue Regfile block) which is responsible for taking rs and rt and store them in their proper register addresses as defined by the op code being passed through. Those values are passed out as 32-bit addresses, qa for rs and qb for rt. Thirdly, we have the regrtMux (green mux on diagram). This Mux is responsible for passing rd or rt out depending on the value regrt (if 1 rt is passed through, if 0 rs is passed through). Fourthly, we move onto the “brain” of the CPU, the control unit. This module is responsible for setting all of the output signals for each type of instruction. Here we only cover R-type instructions, but load word was used in previous assignments. For our final project, we covered add, or, and, xor, sub, instructions. Based on the op code and function, this would determine what values the outputs would be set to. In my module I do this using case statements as it is the cleanest and easiest to work with in my opinion. Finally, this module finishes up with the IDEXE pipeline. This is responsible for updating its outputs based on the inputs only on the positive edge of the clock.

**Stage 3: Execution (EXE)**

**A diagram of a machine

Description automatically generated**

To start this stage, we begin with the IDEXE pipeline which was discussed in the previous stage. Second, in this stage we have the mux right before the alu. This mux is responsible for choosing qb or the immediate extended value based off if ealuimm is a 1 or a 0. If it is a 1, the immediate extended value is chosen, if it is a 0, qb is chosen. This choice depends on which type of instruction is being used. Currently no mux is needed for qa and it is just passed right into the mux but for the extra credit one will be later implemented. Note that the ALU takes in 2 32-bit numbers to perform arithmetic operations on. Next up, we have the alu. This simply performs arithmetic operations on a and b based on the 4 bit aluc value being passed into the ALU. This value is determined by which function is being passed into this stage of the pipeline. The aluc gets is value from the signal generated in the control unit where each arithmetic operation has a unique aluc value. The output r (result) of the alu is passed out of this module. To end this stage we finish with the EXEMEM pipeline. This will update the values passed into on the positive edge of the clock, like previous pipelines.

**Stage 4: Memory (MEM)**

**A diagram of a machine

Description automatically generated**

In our fourth stage, we have the memory module. This module is responsible for storing the results of the alu into memory if necessary. We start with the EXEMEM pipeline which was discussed in further detail in the previous stage, but it passes values only on the positive clock edge. Next, we have the data memory module (blue Data mem block) which is responsible for storing to memory and writing to register if needed. To end this module, We have the MEMWB pipeline which updates the outputs of this stage on the positive clock edge.

**Stage 5: Writeback (WB)**

**A diagram of a machine

Description automatically generated**

In our 5th and final stage, we have the writeback stage. This stage contains 2 modules, the MEMWB pipeline and the writeback mux. The MEMWB pipeline passes values outh of the 4th stage on the positive edge of the clock while the writeback mux determines which value will be passed back into the register file module back in stage 2. If wm2reg is a 0, the delayed result wr is passed through the mux back to the register file and if it is a 1 wdo which is the delayed output of the data memory module back in stage 4 is passed through. Wwreg is passed back into the register file which will determine if writing to a register is necessary. It is important to note that writing is the only this in this CPU that will occur on the negative edge of the clock. This again, is determined by wwreg.

**Forwarding:**

**A diagram of a computer

Description automatically generated**

To finish our final project, some extra modules and changes are added to each of the pipeline stages. The goal of this final project is to further develop our CPU to have proper hazard detection and use forwarding where necessary. Wpcir is displayed in this module but is not necessary as it determines where a stall is needed, which isn’t covered in the final project but rather the extra credit. Two modules are added to the functionality of our CPU, those being ForwardMuxA and ForwardMuxB. These muxes are responsible to determine wether we are performing ALU forwarding, Memory Forwarding, Data Memory Forwarding, or no forwarding at all. One mux deals with qa while the other deals with qb. To determine if forwarding is necessary, we added a hazard detection section to the Control Unit which is why 8 new inputs are necessary. This set of if and else statements checks for any possible hazard and sets the values of fwda and fwdb based off of which forwarding is necessary. Those values are then passed into the forwarding muxes which then pick the value to be passed back into the CPU based on the forwarding necessary, if necessary. This finishes off every necessary portion of the final project for the standards necessary given in the rubric.

Below the code for this project along with all necessary screenshots are given with a detailed description as to what is going on in the waveform output of the CPU.

**Verilog Code:**

**Testbench:**

module TestBench;

// Inputs

reg clk;

// Add other input signals here

// Outputs

wire [31:0] pc;

wire [31:0] dinstOut;

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;

//lab 4 - EXEMEM

wire mwreg;

wire mm2reg;

wire mwmem;

wire [4:0] mdestReg;

wire [31:0] mr;

wire [31:0] mqb;

//lab 4 - MEMWB

wire wwreg;

wire wm2reg;

wire [4:0] wdestReg;

wire [31:0] wr;

wire [31:0] wdo;

initial begin

clk <= 1'b0;

end

// Instantiate the Datapath module

// Instantiate the Datapath module

Datapath datapath(

.clk(clk),

.pc(pc),

.dinstOut(dinstOut),

.ewreg(ewreg),

.em2reg(em2reg),

.ewmem(ewmem),

.ealuc(ealuc),

.ealuimm(ealuimm),

.edestReg(edestReg),

.eqa(eqa),

.eqb(eqb),

.eimm32(eimm32),

.mwreg(mwreg),

.mm2reg(mm2reg),

.mwmem(mwmem),

.mdestReg(mdestReg),

.mr(mr),

.mqb(mqb),

.wwreg(wwreg),

.wm2reg(wm2reg),

.wdestReg(wdestReg),

.wr(wr),

.wdo(wdo)

);

// Clock generation

always begin

#10;

clk = ~clk;

end

endmodule

**Verilog Code:**

**CPU:**

module Datapath(

// Lab 3 - IDEXEpipeline

input clk, // Clock signal

output wire [31:0] pc, // Program Counter

output wire [31:0] dinstOut, // Data from the Instruction Memory

output wire ewreg, // Control signal for write enable in the EX stage

output wire em2reg, // Control signal for write enable in the Memory stage

output wire ewmem, // Control signal for write enable in the MEM stage

output wire [3:0] ealuc, // ALU control signal in the EX stage

output wire ealuimm, // ALU immediate in the EX stage

output wire [4:0] edestReg, // Destination register address in the EX stage

output wire [31:0] eqa, // Data from source register A in the EX stage

output wire [31:0] eqb, // Data from source register B in the EX stage

output wire [31:0] eimm32, // 32-bit immediate in the EX stage

// Lab 4 - EXEMEM

output wire mwreg, // Control signal for write enable in the MEM stage

output wire mm2reg, // Control signal for write enable in the M2 stage (MEM-WB)

output wire mwmem, // Control signal for memory write enable in the MEM stage

output wire [4:0] mdestReg, // Destination register address in the MEM stage

output wire [31:0] mr, // Data read from memory in the MEM stage

output wire [31:0] mqb, // Data from source register B in the MEM stage

// Lab 4 - MEMWB

output wire wwreg, // Control signal for write enable in the WB stage

output wire wm2reg, // Control signal for write enable in the M2 stage (WB)

output wire [4:0] wdestReg, // Destination register address in the WB stage

output wire [31:0] wr, // Result to be written to the register file in the WB stage

output wire [31:0] wdo // Data read from memory or ALU result to be written to the register file

);

// Lab 3 wires

wire wreg; // Control signal for write enable

wire m2reg; // Control signal for write enable (Memory stage)

wire wmem; // Control signal for memory write enable

wire aluimm; // ALU immediate value

wire regrt; // Control signal for selecting a register

wire [4:0] destReg; // Destination register address

wire [31:0] qa; // Data from source register A

wire [31:0] qb; // Data from source register B

wire [31:0] imm32; // 32-bit immediate value

wire [31:0] nextPc; // Next program counter value

wire [31:0] instOut; // Data from the Instruction Memory

wire [3:0] aluc; // ALU control signal

wire [15:0] imm; // Immediate value

wire [4:0] rs; // Source register address

wire [4:0] rt; // Source register address

wire [4:0] rd; // Destination register address

wire [5:0] op; // Operation code

wire [5:0] func; // Function code

// Lab 4 wires

wire [31:0] b; // Data input to the ALU

wire [31:0] r; // Result from the ALU

wire [31:0] mdo; // Data read from memory or result from ALU

//Lab 5 wires

wire [31:0]wbData;

//final project wires

wire wpcir;

wire [1:0] fwda;

wire [1:0] fwdb;

wire [31:0] fwdAout;

wire [31:0] fwdBout;

// Instantiate various components and connect them

ProgramCounter IF\_ProgramCounter\_dp(.clk(clk), .nextPc(nextPc), .wpcir(wpcir), .pc(pc));

pcAdder IF\_pcAdder\_dp(.pc(pc), .nextPc(nextPc));

InstructionMemory IF\_InstructionMemory\_dp(.pc(pc), .instOut(instOut));

IFIDpipelineReg IFIDpipelineReg\_dp(.clk(clk), .instOut(instOut), .wpcir(wpcir), .dinstOut(dinstOut));

ControlUnit ID\_controlUnit\_dp(

.op(op),

.func(func),

.rs(rs),

.rt(rt),

.mdestReg(mdestReg),

.mm2reg(mm2reg),

.mwreg(mwreg),

.edestReg(edestReg),

.em2reg(em2reg),

.ewreg(ewreg),

.wreg(wreg),

.m2reg(m2reg),

.wmem(wmem),

.aluc(aluc),

.aluimm(aluimm),

.regrt(regrt),

.fwda(fwda),

.fwdb(fwdb),

.wpcir(wpcir)

);

RegrtMultiplexer ID\_RegrtMultiplexer\_dp(.rt(rt), .rd(rd), .regrt(regrt), .destReg(destReg));

RegisterFile ID\_RegisterFile\_dp(.rs(rs), .rt(rt), .wdestReg(wdestReg), .wbData(wbData), .wwreg(wwreg), .clk(clk), .qa(qa), .qb(qb));

ImmediateExtender ID\_ImmediateExtender\_dp(.imm(imm), .imm32(imm32));

IDEXEpipeline IDEXEpipeline\_dp(

.wreg(wreg),

.m2reg(m2reg),

.wmem(wmem),

.aluc(aluc),

.aluimm(aluimm),

.destReg(destReg),

.fwdAout(fwdAout),

.fwdBout(fwdBout),

.imm32(imm32),

.clk(clk),

.ewreg(ewreg),

.em2reg(em2reg),

.ewmem(ewmem),

.ealuc(ealuc),

.ealuimm(ealuimm),

.edestReg(edestReg),

.eqa(eqa),

.eqb(eqb),

.eimm32(eimm32)

);

ALU EXE\_ALU\_dp(.eqa(eqa), .b(b), .ealuc(ealuc), .r(r));

ALUMux EXE\_ALUMux\_dp(.eqb(eqb), .eimm32(eimm32), .ealuimm(ealuimm), .b(b));

EXEMEMpipeline EXEMempipeline\_dp(

.ewreg(ewreg),

.em2reg(em2reg),

.ewmem(ewmem),

.edestReg(edestReg),

.r(r),

.eqb(eqb),

.clk(clk),

.mwreg(mwreg),

.mm2reg(mm2reg),

.mwmem(mwmem),

.mdestReg(mdestReg),

.mr(mr),

.mqb(mqb)

);

DataMemory MEM\_DataMemory\_dp(.mr(mr), .mqb(mqb), .mwmem(mwmem), .clk(clk), .mdo(mdo));

MEMWBpipeline MEMWBpipeline\_dp(

.mwreg(mwreg),

.mm2reg(mm2reg),

.mdestReg(mdestReg),

.mr(mr),

.mdo(mdo),

.clk(clk),

.wwreg(wwreg),

.wm2reg(wm2reg),

.wdestReg(wdestReg),

.wr(wr),

.wdo(wdo)

);

WbMux WB\_WbMux\_dp(.wr(wr), .wdo(wdo), .wm2reg(wm2reg), .wbData(wbData));

Fwd\_FwdMuxA Fwd\_FwdMuxA\_dp(.fwda(fwda), .qa(qa), .r(r), .mr(mr), .mdo(mdo), .fwdAout(fwdAout));

Fwd\_FwdMuxB Fwd\_FwdMuxB\_dp(.fwdb(fwdb), .qb(qb), .r(r), .mr(mr), .mdo(mdo), .fwdBout(fwdBout));

// Assign some control signals and data values

assign op = dinstOut[31:26];

assign func = dinstOut[5:0];

assign rs = dinstOut[25:21];

assign rt = dinstOut[20:16];

assign rd = dinstOut[15:11];

assign imm = dinstOut[15:0];

endmodule

module ProgramCounter(

input clk, // Clock input necessary as PC only updates on the positive edge of the clock.

input [31:0] nextPc, // Input from the PC adder looped back to update the next PC.

//final project input

input wpcir,

output reg [31:0] pc // Output of the PC module.

);

initial

begin

pc = 32'd100; // Initializing the PC value to start at 100 in decimal.

end

always @(posedge clk)

begin

if (wpcir == 1)

pc = nextPc; // Update PC to be nextPc only on the positive edge of the clock.

end

endmodule // End of the module

module pcAdder( //creation of the module used for the PC adder module in the cpu.

input [31:0] pc, //input of pc set to be 32 bits wide.

output reg [31:0] nextPc //output register of next pc that is also 32 bits wide.

);

always @(\*) begin //always block that changes ony any signal used to continually update nextPc.

nextPc <= pc + 32'b00000000000000000000000000000100; //setting nextPc equal to ithe input of pc plus a unsigned binary 32 bit 4.

end //end always block

endmodule //end of this module

module InstructionMemory( //instruction memory module within the cpu.

input [31:0] pc, //input of the instruction memory module.

output reg [31:0] instOut //instruction output of the memory module

);

reg [31:0] memory [0:63]; //32x64 array used to store instructions to memory.

initial begin

// //assign the instruction values in memory here (words 25 and 26)

// //lw $v0, 00($at)

// memory[25] = {6'b100011, 5'b00001, 5'b00010, 5'b00000, 5'b00000, 6'b000000};

// //lw $v1, 04($at)

// memory[26] = {6'b100011, 5'b00001, 5'b00011, 5'b00000, 5'b00000, 6'b000100};

// //lw $a0, 08($at)

// memory[27] = {6'b100011, 5'b00001, 5'b00100, 5'b00000, 5'b00000, 6'b001000};

// //lw $a1, 12($at)

// memory[28] = {6'b100011, 5'b00001, 5'b00101, 5'b00000, 5'b00000, 6'b001100};

// //add $a2, $v0, $t2

// memory[29] = {6'b000000, 5'b00010, 5'b01010, 5'b00110, 5'b00000, 6'b100000};

// end

//final project instructions

// add $3, $1, $2

memory[25] = {6'b000000, 5'b00001, 5'b00010, 5'b00011, 5'b00000, 6'b100000};

// sub $4, $9, $3

memory[26] = {6'b000000, 5'b01001, 5'b00011, 5'b00100, 5'b00000, 6'b100010};

// or $5, $3, $9

memory[27] = {6'b000000, 5'b00011, 5'b01001, 5'b00101, 5'b00000, 6'b100101};

// xor $6, $3, $9

memory[28] = {6'b000000, 5'b00011, 5'b01001, 5'b00110, 5'b00000, 6'b100110};

// and $7, $3, $9

memory[29] = {6'b000000, 5'b00011, 5'b01001, 5'b00111, 5'b00000, 6'b100100};

end

always @ (\*) //always block to update instruction out setting to the memory array of pc bits 7 to 2.

begin

instOut <= memory[pc[7:2]];

end

endmodule //end of module

module IFIDpipelineReg( //IFID pipeline

input clk, //clock input needed as dinstOut only updates on the positive edge of clock.

input [31:0] instOut, //input

//final project input

input wpcir,

output reg [31:0] dinstOut //output

);

always @ (posedge clk) //always block that will only update dinstOut on the positive edge of the clock. dinstOut is to the instOut input of this module.

begin

if (wpcir == 1)

dinstOut <= instOut;

end

endmodule //end module

module ControlUnit( //control unit module of the cpu

//inputs

input [5:0] op,

input [5:0] func,

//final project inputs

input [4:0] rs,

input [4:0] rt,

input [4:0] mdestReg,

input mm2reg,

input mwreg,

input [4:0] edestReg,

input em2reg,

input ewreg,

//outputs

output reg wreg,

output reg m2reg,

output reg wmem,

output reg [3:0] aluc,

output reg aluimm,

output reg regrt,

//final project outputs

output reg [1:0] fwda,

output reg [1:0] fwdb,

output reg wpcir

);

reg regUsage = 1'b1;

initial begin

wreg <= 0; //RegWrite

m2reg <= 0; //Mem2Reg

wmem <= 0; //Write Memory

aluimm <= 0; //ALU source

regrt <= 0; //Reg Destination

end

always @ (\*) begin //always block that will continually update

case (op) //case statement of the op code portion of dinstOut which is connected in the datapath module.

6'b000000: // R-type instructions

begin

wreg = 1'b1; // Write to the register file

m2reg = 1'b0; // Do not write to memory

wmem = 1'b0; // Do not write to memory

aluimm = 1'b0; // ALU source from registers

regrt = 1'b0; // Destination register address

case (func) //case statement to check which operation is performed.

6'b100000: aluc = 4'b0010; //ADD Operation

6'b100010: aluc = 4'b0110; //SUB Operation

6'b100100: aluc = 4'b0000; //AND Operation

6'b100101: aluc = 4'b0001; //OR Operation

6'b100110: aluc = 4'b0011; //XOR Operation

default: // Default behavior for unspecified func values

begin

// Set default control signals here

// You can assign default values or behavior

// for cases where func is unspecified

// For example, you can set all signals to 0.

wreg = 1'b0;

m2reg = 1'b0;

wmem = 1'b0;

aluc = 4'b0000;

aluimm = 1'b0;

regrt = 1'b0;

end

endcase

end

6'b100011: // LW instruction

begin

// Set control signals for LW instruction

wreg = 1'b1; // Write to the register file

m2reg = 1'b1; // Write to memory

wmem = 1'b0; // Do not write to memory

aluc = 4'b0010; // ALU operation for addition

aluimm = 1'b1; // ALU source from registers

regrt = 1'b1; // Destination register address

end

6'b101011: //SW instruction

begin

wreg = 1'b0;

m2reg = 1'b0;

wmem = 1'b1;

aluc = 4'b0010;

aluimm = 1'b1;

regrt = 1'b1;

end

default: // Default behavior for unspecified op values

begin

// Set default control signals here for unspecified op values

// You can assign default values or behavior for cases where op is unspecified.

// For example, you can set all signals to 0.

wreg = 1'b0;

m2reg = 1'b0;

wmem = 1'b0;

aluc = 4'b0000;

aluimm = 1'b0;

regrt = 1'b0;

end

endcase

end

//stall

always @ (\*)

begin

if ((ewreg == 1'b1) && (em2reg == 1'b1) && (edestReg != 5'b0) && (regUsage == 1'b1)

&& ((edestReg == rs) || (edestReg == rt))) begin

wreg = 1'b0;

wmem = 1'b0;

wpcir = 1'b0;

end

else begin

wpcir = 1'b1;

end

// forwarding

if ((ewreg == 1'b1) && (edestReg != 5'b0) && (edestReg == rs) && (em2reg == 1'b0)) begin

fwda = 2'b01;

end

else if ((mwreg == 1'b1) && (mdestReg != 5'b0) && (mdestReg == rs) && (mm2reg == 1'b0)) begin

fwda = 2'b10;

end

else if ((mwreg == 1'b1) && (mdestReg != 5'b0) && (mdestReg == rs) && (mm2reg == 1'b1)) begin

fwda = 2'b11;

end

else begin

fwda = 2'b00;

end

if ((ewreg == 1'b1) && (edestReg != 5'b0) && (edestReg == rt) && (em2reg == 1'b0)) begin

fwdb = 2'b01;

end

else if ((mwreg == 1'b1) && (mdestReg != 5'b0) && (mdestReg == rt) && (mm2reg == 1'b0)) begin

fwdb = 2'b10;

end

else if ((mwreg == 1'b1) && (mdestReg != 5'b0) && (mdestReg == rt) && (mm2reg == 1'b1)) begin

fwdb = 2'b11;

end

else begin

fwdb = 2'b00;

end

end

endmodule

module RegrtMultiplexer(

// Inputs

input [4:0] rt, // Input register value rt

input [4:0] rd, // Input register value rd

input regrt, // Control signal to select the output (0 for rd, 1 for rt)

// Output

output reg [4:0] destReg // Output register value (selected based on the control signal)

);

always @(\*)

begin

if (regrt == 0)

destReg = rd; // If regrt is 0, select rd as the output.

else

destReg = rt; // If regrt is 1, select rt as the output.

end

endmodule // End of the module

module RegisterFile(

// Inputs

input [4:0] rs, // Input for the source register (rs)

input [4:0] rt, // Input for the target register (rt)

//Lab 5 Inputs

input [4:0] wdestReg,

input [31:0] wbData,

input wwreg,

input clk,

// Outputs

output reg [31:0] qa, // Output for the value stored in the source register

output reg [31:0] qb // Output for the value stored in the target register

);

reg [31:0] register [0:31]; // 32x32 array for registers (register file)

// Initialize all registers to 0

integer r;

initial begin

for (r = 11; r <= 31; r = r + 1) begin

register[r] = 0; // Initialize each register to 0.

register[0] = 32'h00000000;

register[1] = 32'hA00000AA;

register[2] = 32'h10000011;

register[3] = 32'h20000022;

register[4] = 32'h30000033;

register[5] = 32'h40000044;

register[6] = 32'h50000055;

register[7] = 32'h60000066;

register[8] = 32'h70000077;

register[9] = 32'h80000088;

register[10] = 32'h90000099;

end

end

always @ (\*) // Always block to update qa and qb based on the input rs and rt values.

begin

qa = register[rs]; // Output qa is the value stored in the source register (rs).

qb = register[rt]; // Output qb is the value stored in the target register (rt).

end

always @ (negedge clk)

begin

if (wwreg == 1)

register[wdestReg] = wbData;

end

endmodule // End of the module

module ImmediateExtender( //immediate extender module.

input [15:0] imm,

output reg [31:0] imm32

);

always @ (\*) //always block to update the value of imm32.

begin

imm32 = {{16{imm[15]}}, imm}; //sets imm32 to be equal to imm. the last bit is concatinated to the other 16 bits based on if the sign bit is a zero or one.

end

endmodule

module IDEXEpipeline(

// Inputs

input wreg, // Control signal for writing to the register file

input m2reg, // Control signal for writing to the register file (M2 stage)

input wmem, // Control signal for writing to memory

input [3:0] aluc, // ALU control signal

input aluimm, // ALU immediate value

input [4:0] destReg, // Destination register address

input [31:0] fwdAout, // Value from source register A

input [31:0] fwdBout, // Value from source register B

input [31:0] imm32, // 32-bit immediate value

input clk, // Clock signal

// Outputs

output reg ewreg, // Output for write enable signal

output reg em2reg, // Output for write enable signal (M2 stage)

output reg ewmem, // Output for memory write enable signal

output reg [3:0] ealuc, // Output for ALU control signal

output reg ealuimm, // Output for ALU immediate value

output reg [4:0] edestReg, // Output for destination register address

output reg [31:0] eqa, // Output for source register A value

output reg [31:0] eqb, // Output for source register B value

output reg [31:0] eimm32 // Output for 32-bit immediate value

);

// On the positive edge of the clock, update the output signals with the input values.

always @ (posedge clk)

begin

ewreg = wreg;

em2reg = m2reg;

ewmem = wmem;

ealuc = aluc;

ealuimm = aluimm;

edestReg = destReg;

eqa = fwdAout;

eqb = fwdBout;

eimm32 = imm32;

end

endmodule // End of the module

module ALU(

input [31:0] eqa, // Input A for the ALU

input [31:0] b, // Input B for the ALU

input [3:0] ealuc, // ALU control signal

output reg [31:0] r // Output of the ALU

);

// ALU operation codes

// 0000 - AND

// 0001 - OR

// 0010 - ADD

// 0110 - SUBTRACT

// 0111 - SET LESS THAN

// 1100 - NOR

// 0011 - XOR

always @ (\*)

begin

case(ealuc)

4'b0000: r = eqa & b; // AND operation

4'b0001: r = eqa | b; // OR operation

4'b0010: r = eqa + b; // ADD operation

4'b0110: r = eqa - b; // SUBTRACT operation

4'b1100: r = ~(eqa | b); // NOR operation

4'b0011: r = eqa ^ b; // XOR operation

endcase

end

endmodule // End of the module

module ALUMux(

input [31:0] eqb, // Input B data from the ALU

input [31:0] eimm32, // Immediate value from the pipeline

input ealuimm, // Mux control signal

output reg [31:0] b // Output data selected by the Mux

);

always @(\*) begin

case(ealuimm)

1'b0: // Select eqb as the output

begin

b <= eqb;

end

1'b1: // Select eimm32 as the output

begin

b <= eimm32;

end

endcase

end

endmodule // End of the module

module EXEMEMpipeline(

input ewreg, // Control signal for writing to the register file

input em2reg, // Control signal for writing to the register file (Memory stage)

input ewmem, // Control signal for writing to memory

input [4:0] edestReg, // Destination register address

input [31:0] r, // Result from the ALU

input [31:0] eqb, // Value from source register B

input clk, // Clock signal

output reg mwreg, // Output for write enable signal

output reg mm2reg, // Output for write enable signal (M2 stage)

output reg mwmem, // Output for memory write enable signal

output reg [4:0] mdestReg, // Output for destination register address

output reg [31:0] mr, // Output for result from the ALU

output reg [31:0] mqb // Output for value from source register B

);

always @ (posedge clk)

begin

mwreg = ewreg;

mm2reg = em2reg;

mwmem = ewmem;

mdestReg = edestReg;

mr = r;

mqb = eqb;

end

endmodule // End of the module

module DataMemory(

input [31:0] mr, // Memory read address

input [31:0] mqb, // Data to be written to memory

input mwmem, // Memory write control signal

input clk, // Clock signal

output reg [31:0] mdo // Data read from memory

);

// Define a data memory array with 64 words

reg [31:0] dataMemory [0:63];

// Initialize data memory with some values (words 0-9)

initial begin

dataMemory[0] = 32'hA00000AA;

dataMemory[1] = 32'h10000011;

dataMemory[2] = 32'h20000022;

dataMemory[3] = 32'h30000033;

dataMemory[4] = 32'h40000044;

dataMemory[5] = 32'h50000055;

dataMemory[6] = 32'h60000066;

dataMemory[7] = 32'h70000077;

dataMemory[8] = 32'h80000088;

dataMemory[9] = 32'h90000099;

end

always @(\*) begin

// Reading: Set mdo to the value at the memory read address (bits 7:2 of mr)

mdo = dataMemory[mr[7:2]];

end

always @(negedge clk) begin

// Writing: If mwmem is 1, write the value in mqb to the memory at the read address

if (mwmem == 1) begin

//dataMemory[mr[7:2]] <= mqb;

dataMemory[mr] <= mqb;

end

end

endmodule // End of the module

module MEMWBpipeline(

input mwreg, // Control signal for writing to the register file

input mm2reg, // Control signal for writing to the register file (Memory Stage)

input [4:0] mdestReg, // Destination register address

input [31:0] mr, // Result from the data memory

input [31:0] mdo, // Data read from the data memory

input clk, // Clock signal

output reg wwreg, // Output for write enable signal

output reg wm2reg, // Output for write enable signal (Memory stage)

output reg [4:0] wdestReg, // Output for destination register address

output reg [31:0] wr, // Output for result from the data memory

output reg [31:0] wdo // Output for data read from the data memory

);

always @ (posedge clk)

begin

wwreg = mwreg;

wm2reg = mm2reg;

wdestReg = mdestReg;

wr = mr;

wdo = mdo;

end

endmodule // End of the module

module WbMux(

input [31:0] wr,

input [31:0] wdo,

input wm2reg,

output reg [31:0] wbData

);

always @ (\*)

begin

if (wm2reg == 0)

wbData = wr;

else

wbData = wdo;

end

endmodule

module Fwd\_FwdMuxA(

input [1:0] fwda,

input [31:0] qa,

input [31:0] r,

input [31:0] mr,

input [31:0] mdo,

output reg [31:0] fwdAout

);

always @ (\*) begin

case(fwda)

2'b00: fwdAout <= qa;

2'b01: fwdAout <= r;

2'b10: fwdAout <= mr;

2'b11: fwdAout <= mdo;

endcase

end

endmodule

module Fwd\_FwdMuxB(

input [1:0] fwdb,

input [31:0] qb,

input [31:0] r,

input [31:0] mr,

input [31:0] mdo,

output reg [31:0] fwdBout

);

always @ (\*) begin

case(fwdb)

2'b00: fwdBout <= qb;

2'b01: fwdBout <= r;

2'b10: fwdBout <= mr;

2'b11: fwdBout <= mdo;

endcase

end

endmodule

**Waveform**

A screenshot of a computer

Description automatically generated

Here, this waveform uses different instructions than used in previous labs (lab 1 – lab 5). The instructions used here a picked specific in mind of needing forwarding as the instructions following the first instruction (add $3, $1, $2) have dependencies on the result of register 3. Because of this, forwarding and hazard detection will need to be implemented. ealuc displays 5 values for each of the 5 instructions. 2 is an add instruction, 6 is a subtraction instruction, 1 is an or instruction, 3 is an xor instruction, and 0 is an and instruction. Note the and instruction is there but it looks like its not because a default statement keeps ealuc as 0 when no instruction is passed through. edestReg points of the destination register in which each resulting value will be stored from the operation. eqa(representing register rs) and eqb (representing register rt) hold each value of the performed operation. So first register 3 stores the value of the add operation adding rs and rt, so you get a00000aa + 10000011 = b00000bb. This is now stored in register 3. The second instruction takes register 3 (now holding b00000bb) and register 9 (holding 80000088) and subtracts them. This gives you (80000088 – b00000bb) which results in (cfffffcd). After the subtraction instruction, rs and rt stay the exact same because the last 3 instructions all use register 3 and 9 in the same place. Since the last 3 instructions never change the values of rs and rt, the values will stay the same. Then, mdestreg represents delayed edgestreg in the pipeline. mr will be the delayed output of the alu (output of the exemempipeline). wr is the delayed output at the memwbpipeline. Looking at either mr or wr, both will properly show the result of the 5 instructions being passed through the pipeline. At the bottom, I also displayed the output of the forwarding mux the show which of the values is being picked by the mux.

Add $3, $1, $2: a00000aa + 10000011 = b00000bb

Sub $4, $9, $3: 80000088 – b00000bb = cfffffcd

Or $5, $3, $9: b00000bb | 80000088 = b00000bb

Xor $6, $3, $9: b00000bb ^ 80000088 = 30000033

And $7, $3, $9: b00000bb & 80000088 = 80000088

**Schematic**  
 **RTL**

A diagram of a computer

Description automatically generated

**Synthesis**

**A drawing of a diagram

Description automatically generated**

**IO**

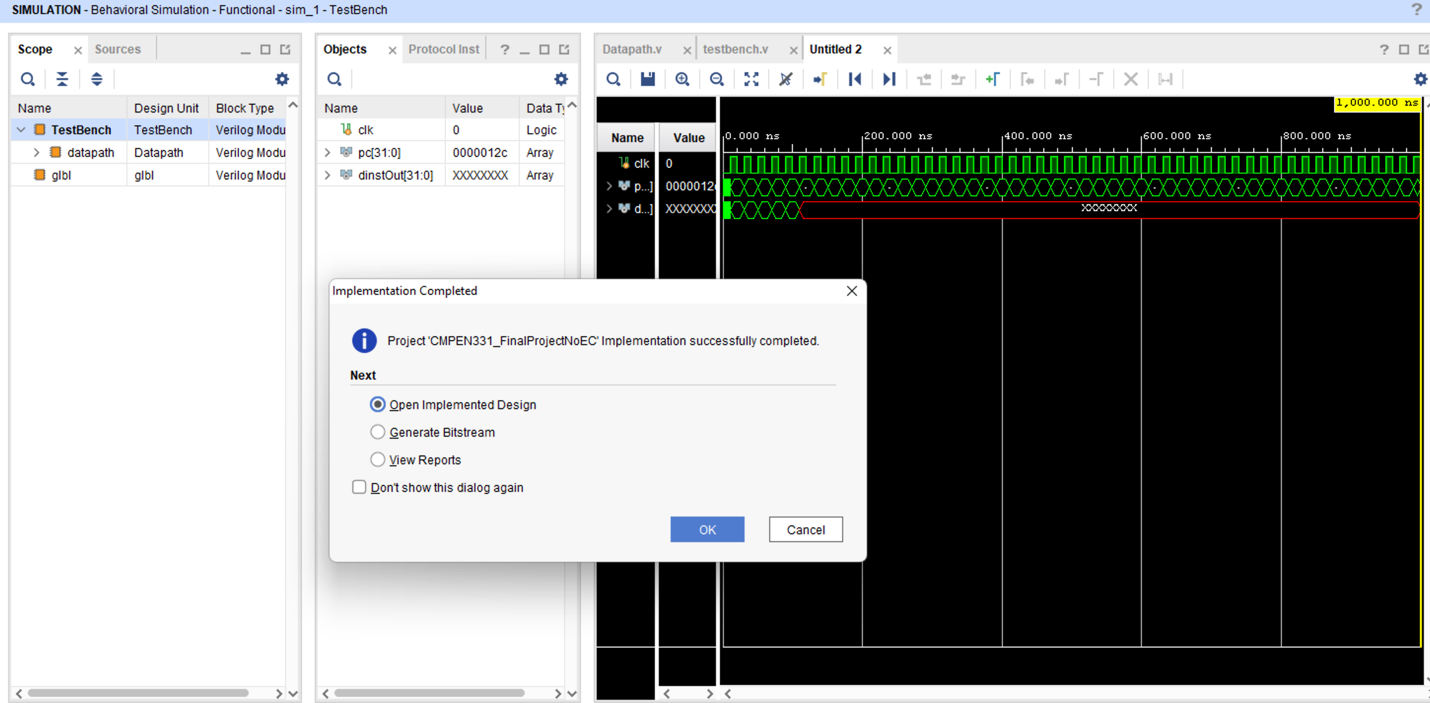
A screenshot of a game

Description automatically generated

**Floorplanning**

**A screen shot of a computer screen

Description automatically generated**

**Proof of Implementation Passing:  
**

**Proof of Bitstream Passing:**

**A screenshot of a computer

Description automatically generated**