**Lab 3 – CMPEN 331**

**Tanner Kleintop**

**Section 002**

**Dr. Almekkawy**

**Verilog Code:**

module Datapath(

input clk,

output wire [31:0] pc,

output wire [31:0] dinstOut,

output wire ewreg,

output wire em2reg,

output wire ewmem,

output wire [3:0] ealuc,

output wire ealuimm,

output wire [4:0] edestReg,

output wire [31:0] eqa,

output wire [31:0] eqb,

output wire [31:0] eimm32

);

wire wreg;

wire m2reg;

wire wmem;

wire aluimm;

wire regrt;

wire[4:0] destReg;

wire[31:0] qa;

wire[31:0] qb;

wire[31:0] imm32;

wire [31:0] nextPc;

wire [31:0] instOut;

wire [3:0] aluc;

wire [15:0] imm;

wire [4:0] rs;

wire [4:0] rt;

wire [4:0] rd;

wire [5:0] op;

wire [5:0] func;

ProgramCounter ProgramCounter\_db(.clk(clk), .nextPc(nextPc), .pc(pc));

pcAdder pcAdder\_db(.pc(pc), .nextPc(nextPc));

InstructionMemory InstructionMemory\_db(.pc(pc), .instOut(instOut));

IFIDpipelineReg IFIDpipelineReg\_db(.clk(clk), .instOut(instOut), .dinstOut(dinstOut));

ControlUnit controlUnit\_db(.op(op), .func(func), .wreg(wreg), .m2reg(m2reg), .wmem(wmem), .aluc(aluc), .aluimm(aluimm), .regrt(regrt));

RegrtMultiplexer RegrtMultiplexer\_db(.rt(rt), .rd(rd), .regrt(regrt), .destReg(destReg));

RegisterFile RegisterFile\_db(.rs(rs), .rt(rt), .qa(qa), .qb(qb));

ImmediateExtender ImmediateExtender\_db(.imm(imm), .imm32(imm32));

IDEXEpipeline IDEXEpipeline\_db(

.wreg(wreg),

.m2reg(m2reg),

.wmem(wmem),

.aluc(aluc),

.aluimm(aluimm),

.destReg(destReg),

.qa(qa),

.qb(qb),

.imm32(imm32),

.clk(clk),

.ewreg(ewreg),

.em2reg(em2reg),

.ewmem(ewmem),

.ealuc(ealuc),

.ealuimm(ealuimm),

.edestReg(edestReg),

.eqa(eqa),

.eqb(eqb),

.eimm32(eimm32)

);

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( //module for the program counter (PC pipeline) module of the cpu.

input clk, //clk 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 next pc.

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

);

initial //initial block used to initalize the value of pc so it starts at a 100 in decimal.

begin

pc = 32'd100; //setting pc equal to a 100 in decimal.

end

always @(posedge clk) //update pc to be nextPc only on the positive edge of the clock.

begin

pc = nextPc;

end

endmodule //end of 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};

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

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

dinstOut = instOut;

end

endmodule //end module

module ControlUnit( //control unit module of the cpu

//inputs

input [5:0] op,

input [5:0] func,

//outputs

output reg wreg,

output reg m2reg,

output reg wmem,

output reg [3:0] aluc,

output reg aluimm,

output reg regrt

);

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

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

6'b100000: // ADD instruction

begin

// Set control signals for ADD instruction

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

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

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

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

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

regrt = 1'b1; // Destination register address

end

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'b0000; // ALU operation for addition

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

regrt = 1'b0; // Destination register address

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

endmodule

module RegrtMultiplexer( //multiplexer module for the cpu.

//inputs

input [4:0] rt,

input [4:0] rd,

input regrt,

//outputs

output reg [4:0] destReg

);

always @ (\*) //always block used to update the value of the destination register depending on the value of regrt.

begin

if (regrt == 0) //if regrt is a 0

destReg = rd; //set destination register equal to rd

else if (regrt == 1) //if regrt is a 1

destReg = rt; //set destReg to equal rt.

else

destReg = 6'b0; // Default assignment for other cases

end

endmodule

module RegisterFile( //register file module of the cpu.

//inputs

input [4:0] rs,

input [4:0] rt,

//outputs

output reg [31:0] qa,

output reg [31:0] qb

);

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

//initalize all register to a 0

integer r;

initial begin

for (r = 0; r <= 31; r = r + 1) begin

register[r] = 0;

end

end

always @ (\*) //always block to update qa and qb to be the array address for a specific register. this will update on any signal.

begin

qa = register[rs];

qb = register[rt];

end

endmodule

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( //idexe pipeline

//inputs

input wreg,

input m2reg,

input wmem,

input [3:0] aluc,

input aluimm,

input [4:0] destReg,

input [31:0] qa,

input [31:0] qb,

input [31:0] imm32,

input clk,

//outputs

output reg ewreg,

output reg em2reg,

output reg ewmem,

output reg [3:0] ealuc,

output reg ealuimm,

output reg [4:0] edestReg,

output reg [31:0] eqa,

output reg [31:0] eqb,

output reg [31:0] eimm32

);

//only the postive edge of the clock these values will be updated based on the clocks signal.

always @ (posedge clk)

begin

ewreg = wreg;

em2reg = m2reg;

ewmem = wmem;

ealuc = aluc;

ealuimm = aluimm;

edestReg = destReg;

eqa = qa;

eqb = qb;

eimm32 = imm32;

end

endmodule

**Test Bench:**

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;

initial begin

clk <= 1'b0;

end

// 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)

);

// Clock generation

always begin

#10;

clk = ~clk;

end

endmodule

**RTL Schematic Model:**

**A computer screen shot of a computer

Description automatically generated**

**Synthesis Schematic Model:**

**A diagram of a computer

Description automatically generated**

**A screenshot of a computer

Description automatically generated**

**A computer screen shot of a diagram

Description automatically generated**

**A screenshot of a computer

Description automatically generated**

**A screenshot of a computer

Description automatically generated**

**A computer screen shot of a diagram

Description automatically generated**

**A screenshot of a computer

Description automatically generated**

**A screenshot of a computer

Description automatically generated**

**A screenshot of a computer

Description automatically generated**

**A diagram of a computer

Description automatically generated**

**A screenshot of a computer

Description automatically generated**

**A screenshot of a computer

Description automatically generated**

**A white background with black text

Description automatically generated**

**I/O Planning**

**A screenshot of a computer game

Description automatically generated**

**Floor Planning**

**A screenshot of a computer

Description automatically generated**

**Waveforms:  
A screenshot of a computer

Description automatically generated**

**A screenshot of a computer

Description automatically generated**

**Waveform Explanation:**

Above are the two screenshots of the waveform. One is the full duration of the test benches clock and the other is a zoomed in shot of where all the data starts. So the first image shows the values beginning as x’s. This is because the data isn’t fully “fetched” until one full clock cycle is complete. Pc increments at a value of 4, starting at 100. So it goes 100,104,108,112, etc. until the end of the cycle which in my case is 300. Note that the values on the waveform are shown in hexadecimal as showing a good shot wasn’t possible with 32 bit long binary values. dinstOut represents the full 32 bit number of machine code being passed throughout the model. For our case dinstOut either represents an r-type function or an i-type function. Once dinstOut is updated, the 2 values that are not x’s are the 2 values being read from the instruction memory. Those being array 25 and array 26. Since those are the only 2 values, they are read and once read x’s wil be provided as there is no other values in the memory to be read. Ewreg, em2reg, ewmem, ealuc, ealuimm, edestreg, eqa, eqb, and eimm are all outputs of the control unit and at this time serve no purpose until connected to future modules of the lab. Though it is important to note that they start as a one and drop to a zero because of the added default statement for each case statement, without it the values would stay ones/zeros. This waveform properly demonstrates that the clock functions, and that pc properly increments only on the positive edge of the clock.