**Final Project Extra Credit – 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.

**Everything put together:**A diagram of a machine

Description automatically generated

For the extra credit, additions were made to further add to the capabilities of our MIPS CPU. Now, our CPU can handle branching, jumping, load word, store word, load upper immediate, jump and link, and almost all r-type instructions. Starting in the instruction fetch stage, a PCMux was implemented to choose which value of pc will be passed back into the pipeline as nextPc. A 0 value of pcsrc into this mux would indicate an r-type instruction, 1 would indicate a branch instruction, 2 would indicate a jump instruction, and 3 would represent a jump instruction. This value of pcsrc will be generated in the control unit where each instruction generates a pc source signal. Moving onto the instruction decode stage, a few additions were made. Two left-shift modules were implemented for the purpose of allowing jump instructions and branch instructions. Both modules left shift by 2. The address is the input of this module which is the least 26 significant bits of dinstOut. In the other left shift module for branches, imm which is the least significant 16 bits of dinstOut is passed into this module. The address left shift module takes in 26 bits and passes out 26 while the imm left shift module takes in 16 bits and passes out 32 bits. The upper 16 bits are concatenated with 0’s in the imm left shift output. The immediate extender takes in a new input, sext (sign extend). This value is responsible for determining the output of the immediate extender. The next module is the adder of the program counter for any branch operations. This takes in the delayed pc4 (dpc4) and the 32-bit immOut value from the imm left shift module. The final new addition to this module is a check to see if rs and rt are equal. This module is used specifically for branching instructions, using qa and qb as inputs and a signal to say if rs or rt is of equal value. In the execution stage, 3 new modules are added. First, we have another program counter that takes in the delayed dpc4 (epc4) and adds a binary 4 to it. This pc is used based on if the instruction is a jal instruction. Another mux was added before the ALU, this mux allows for shifting operations to choose whether ea or sa (bits 10 to 6 of dimm32) will be picked as the input for a. Finally, we have an f module. This module is used for jump and link instructions, taking in edestReg0 and output edestReg. If the instruction is not a jump and link (jal) then edestReg will be set equal to edestReg0. If it is a jal instruction, then edestReg will be set to a binary 31. Moving onto the memory stages and writeback stages, we see no changes or module additions.

**Verilog Code:  
 Testbench:**

`timescale 1ns / 1ps

module TestBench;

reg clk;

reg clrn;

wire [31:0] pc;

wire [31:0] dinstOut;

wire [31:0] ealu;

wire [31:0] malu;

wire [31:0] wbData;

initial begin

clk <= 1'b0;

clrn <=1'b0;

end

Datapath Datapath(.clrn(clrn), .clk(clk), .pc(pc), .dinstOut(dinstOut), .ealu(ealu), .malu(malu), .wbData(wbData));

// Clock generation

always begin

#5;

clk = ~clk;

end

always begin

#2; clrn = 1'b1;

end

endmodule

**Other Modules:**

`timescale 1ns / 1ps

module Datapath(

input clk,

input clrn,

output wire [31:0] pc,

output wire [31:0] dinstOut,

output wire [31:0] ealu,

output wire [31:0] malu,

output wire [31:0] wbData

);

//wire [31:0] nextPc, pc4, bpc, da, njpc, dpc4, dinstOut, qa, qb,

// jpc, mdo, db, dimm32, immOut, eimm32, sa, mwreg, wreg, m2reg,

// jal, alu, r, mb, walu, wdo, ea, eb, epc4, epc8, a, b;

//wire [25:0] addr;

//wire [15:0] imm;

//wire [5:0] op, func;

//wire [4:0] rs, rt, rd, wrn, ern, mrn, drn, ern0;

//wire [3:0] aluc, ealuc;

//wire [1:0] pcsrc, fwda, fwdb;

//wire wwreg, sext, regrt, mm2reg, em2reg,

// ewreg, aluimm, shift, rsrtequ,

// ewmem, ejal, ealuimm, eshift, wm2reg, mwmem, wmem;

wire [31:0] nextPc;

wire [31:0] pc4;

wire [31:0] bpc;

wire [31:0] da;

wire [31:0] njpc;

wire [1:0] pcsrc;

wire wpcir;

wire [31:0] dpc4;

wire [31:0] instOut;

wire [5:0] op;

wire [5:0] func;

wire [4:0] rs;

wire [4:0] rt;

wire [4:0] rd;

wire [15:0] imm;

wire [25:0] addr;

wire [31:0] qa;

wire [31:0] qb;

wire [31:0] immOut;

wire [25:0] jpc;

wire wwreg;

wire [4:0] wrn;

wire [31:0] mdo;

wire [31:0] db;

wire sext;

wire [31:0] dimm32;

wire [31:0] eimm32;

wire regrt;

wire [4:0] drn;

wire [4:0] mrn;

wire mm2reg;

wire mwreg;

wire [4:0] ern;

wire em2reg;

wire ewreg;

wire wreg;

wire m2reg;

wire wmem;

wire jal;

wire [3:0] aluc;

wire aluimm;

wire shift;

wire rsrtequ;

wire [1:0] fwda;

wire [1:0] fwdb;

wire ewmem;

wire ejal;

wire [3:0] ealuc;

wire ealuimm;

wire eshift;

wire [31:0] sa;

wire [31:0] ea;

wire [31:0] eb;

wire [31:0] epc4;

wire [31:0] epc8;

wire [31:0] a;

wire [31:0] b;

wire [31:0] r;

wire [4:0] ern0;

wire mwmem;

wire [31:0] mb;

wire wm2reg;

wire [31:0] walu;

wire [31:0] wdo;

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

pcAdder IF\_PCAdder(.pc(pc), .pc4(pc4));

IF\_ProgramCounterMux IF\_PCMux(.pc4(pc4), .bpc(bpc), .da(da), .njpc(njpc), .pcsrc(pcsrc), .nextPc(nextPc));

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

IFIDpipelineReg IFIDpipeline(.wpcir(wpcir), .pc4(pc4), .instOut(instOut), .clk(clk), .dinstOut(dinstOut), .dpc4(dpc4));

ID\_addrLS ID\_addrLS(.addr(addr), .jpc(jpc));

ID\_immLS ID\_immLS(.imm(imm), .immOut(immOut));

ID\_BranchProgramCounter ID\_BPC(.dpc4(dpc4), .immOut(immOut), .bpc(bpc));

RegisterFile ID\_RegFile(.rs(rs), .rt(rt), .wwreg(wwreg), .clrn(clrn), .clk(clk), .wbData(wbData), .wrn(wrn), .qa(qa), .qb(qb));

Fwd\_FwdMuxA ID\_FwdMuxA(.qa(qa), .malu(malu), .ealu(ealu), .mdo(mdo), .fwda(fwda), .da(da));

Fwd\_FwdMuxB ID\_FwdMuxB(.qb(qb), .malu(malu), .ealu(ealu), .mdo(mdo), .fwdb(fwdb), .db(db));

ImmediateExtender ID\_IE(.imm(imm), .dimm32(dimm32), .sext(sext));

RegrtMultiplexer ID\_RegrtMux(.rd(rd), .rt(rt), .regrt(regrt), .drn(drn));

ID\_rsrtequCheck ID\_rsrtequCheck(.da(da), .db(db), .rsrtequ(rsrtequ));

ID\_JumpPCconcatination ID\_JPCconcat(.jpc(jpc), .dpc4(dpc4), .njpc(njpc));

ControlUnit ID\_ControlUnit(.op(op), .rs(rs), .rt(rt), .func(func), .pcsrc(pcsrc), .wpcir(wpcir),

.mrn(mrn), .mm2reg(mm2reg), .mwreg(mwreg), .ern(ern), .em2reg(em2reg), .ewreg(ewreg), .wreg(wreg),

.m2reg(m2reg), .wmem(wmem), .jal(jal), .aluc(aluc), .aluimm(aluimm), .shift(shift), .regrt(regrt),

.rsrtequ(rsrtequ), .sext(sext), .fwda(fwda), .fwdb(fwdb));

IDEXEpipeline IDEXEpipeline(.wreg(wreg), .m2reg(m2reg), .wmem(wmem), .jal(jal), .aluc(aluc), .aluimm(aluimm),

.shift(shift), .dpc4(dpc4), .da(da), .db(db), .dimm32(dimm32), .drn(drn), .clk(clk), .ern0(ern0), .eimm32(eimm32),

.eb(eb), .ea(ea), .epc4(epc4), .eshift(eshift), .ealuimm(ealuimm), .ealuc(ealuc), .ejal(ejal), .ewmem(ewmem),

.em2reg(em2reg), .ewreg(ewreg));

EXE\_ProgramCounter EXE\_ProgramCounter(.epc4(epc4), .epc8(epc8));

EXE\_ALUMux2 EXE\_ALUMuxA(.sa(sa), .ea(ea), .eshift(eshift), .a(a));

ALUMux EXE\_ALUMuxB(.eimm32(eimm32), .eb(eb), .ealuimm(ealuimm), .b(b));

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

EXE\_ALUoutMux EXE\_ealuMux(.epc8(epc8), .r(r), .ejal(ejal), .ealu(ealu));

EXE\_f EXE\_f(.ern0(ern0), .ern(ern), .ejal(ejal));

EXEMEMpipeline EXEMEMpipeline(.ewreg(ewreg), .em2reg(em2reg), .ewmem(ewmem), .ealu(ealu), .eb(eb), .ern(ern),

.clk(clk), .mrn(mrn), .mb(mb), .malu(malu), .mwmem(mwmem), .mm2reg(mm2reg), .mwreg(mwreg));

DataMemory MEM\_DataMemory(.malu(malu), .mb(mb), .mwmem(mwmem), .mdo(mdo), .clk(clk));

MEMWBpipeline MEMWBpipeline(.mwreg(mwreg), .mm2reg(mm2reg), .mdo(mdo), .malu(malu), .mrn(mrn), .clk(clk),

.wrn(wrn), .walu(walu), .wdo(wdo), .wm2reg(wm2reg), .wwreg(wwreg));

WbMux WB\_wbMux(.wdo(wdo), .walu(walu), .wbData(wbData), .wm2reg(wm2reg));

// 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];

assign sa = {21'b0,eimm32[10:6],6'b0};

assign addr = dinstOut[25:0];

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'd0; // 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] pc4 //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.

pc4 <= 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 IF\_ProgramCounterMux(

input [1:0] pcsrc,

input [31:0] pc4,

input [31:0] bpc,

input [31:0] njpc,

input [31:0] da,

output reg [31:0] nextPc

);

always @ (\*)

begin

case(pcsrc)

2'b00: nextPc = pc4;

2'b01: nextPc = bpc;

2'b10: nextPc = da;

2'b11: nextPc = njpc;

endcase

end

endmodule

module InstructionMemory( // instruction memory, rom

input [31:0] pc, // rom address

output reg [31:0] instOut // rom content = rom[a]

);

wire [31:0] rom [0:63]; // rom cells: 64 words \* 32 bits

// 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[7:2]]; // use 6-bit word address to read rom

end

endmodule

module IFIDpipelineReg( //IFID pipeline

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

input [31:0] instOut, //input

//EC Input

input [31:0] pc4,

//final project input

input wpcir,

output reg [31:0] dinstOut, //output

//EC Output

output reg [31:0] dpc4

);

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;

dpc4 <= pc4;

end

endmodule //end module

module ID\_addrLS(

input [25:0] addr,

output reg [25:0] jpc

);

always @ (\*)

begin

jpc = addr << 2;

end

endmodule

module ID\_immLS(

input [15:0] imm,

output reg [31:0] immOut

);

always @ (\*)

begin

immOut = {16'b0, imm} << 2;

end

endmodule

module ID\_BranchProgramCounter(

input [31:0] dpc4,

input [31:0] immOut,

output reg [31:0] bpc

);

always @ (\*)

begin

bpc <= dpc4 + immOut;

end

endmodule

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] wrn,

input [31:0] wbData,

input wwreg,

input clk, clrn,

// 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 = 0; 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[wrn] = wbData;

end

endmodule

module Fwd\_FwdMuxA(

input [1:0] fwda,

input [31:0] qa,

input [31:0] ealu,

input [31:0] malu,

input [31:0] mdo,

output reg [31:0] da

);

always @ (\*) begin

case(fwda)

2'b00: da <= qa;

2'b01: da <= ealu;

2'b10: da <= malu;

2'b11: da <= mdo;

endcase

end

endmodule

module Fwd\_FwdMuxB(

input [1:0] fwdb,

input [31:0] qb,

input [31:0] ealu,

input [31:0] malu,

input [31:0] mdo,

output reg [31:0] db

);

always @ (\*) begin

case(fwdb)

2'b00: db <= qb;

2'b01: db <= ealu;

2'b10: db <= malu;

2'b11: db <= mdo;

endcase

end

endmodule

module ImmediateExtender( //immediate extender module.

input [15:0] imm,

input sext,

output reg [31:0] dimm32

);

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

begin

if (sext == 1)

dimm32 = {{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.

else

dimm32 <= {16'b0, imm};

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] drn // Output register value (selected based on the control signal)

);

always @(\*)

begin

if (regrt == 0)

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

else

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

end

endmodule // End of the module

module ID\_rsrtequCheck(

input [31:0] da,

input [31:0] db,

output reg rsrtequ

);

always @ (\*)

begin

if (da == db)

rsrtequ <= 1;

else

rsrtequ <= 0;

end

endmodule

module ID\_JumpPCconcatination(

input [25:0] jpc,

input [31:0] dpc4,

output reg [31:0] njpc

);

always @ (\*)

begin

njpc <= {{dpc4[31:26]}, jpc};

end

endmodule

module ControlUnit( //control unit module of the cpu

//inputs

input [5:0] op, func,

//final project inputs

input [4:0] rs, rt, mrn, ern,

input mm2reg, mwreg, em2reg, ewreg,

//Extra Credit Input

input rsrtequ,

//outputs

output reg wreg, m2reg, wmem, aluimm, regrt,

output reg [3:0] aluc,

//final project outputs

output reg [1:0] fwda, fwdb,

output reg wpcir,

//Extra Credit Outputs

output reg sext, shift, jal,

output reg [1:0] pcsrc

);

reg regUsage = 1'b1;

initial begin

wreg <= 0; //RegWrite

m2reg <= 0; //Mem2Reg

wmem <= 0; //Write Memory

aluimm <= 0; //ALU source

regrt <= 0; //Reg Destination

pcsrc <= 2'b00;

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

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

6'b100000: begin

aluc = 4'b0010; //ADD Operation

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

sext = 1'bx;

jal = 1'b0;

shift = 1'b0;

pcsrc = 2'b00;

end

6'b100010: begin

aluc = 4'b0110; //SUB

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

sext = 1'bx;

jal = 1'b0;

shift = 1'b0;

pcsrc = 2'b00;

end

6'b100100: begin

aluc = 4'b0000; //AND Operation

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

sext = 1'bx;

jal = 1'b0;

shift = 1'b0;

pcsrc = 2'b00;

end

6'b100101: begin

aluc = 4'b0001; //OR

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

sext = 1'bx;

jal = 1'b0;

shift = 1'b0;

pcsrc = 2'b00;

end

6'b100110: begin

aluc = 4'b0011; //XOR Operation

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

sext = 1'bx;

jal = 1'b0;

shift = 1'b0;

pcsrc = 2'b00;

end

6'b000000: begin

aluc = 4'b0111; //sll

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

sext = 1'bx;

jal = 1'b0;

shift = 1'b1;

pcsrc = 2'b00;

end

6'b000010: begin

aluc = 4'b1110; //srl

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

sext = 1'bx;

jal = 1'b0;

shift = 1'b1;

pcsrc = 2'b00;

end

6'b000011: begin

aluc = 4'b1001; //sra

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

sext = 1'bx;

jal = 1'b0;

shift = 1'b1;

pcsrc = 2'b00;

end

6'b001000:

begin aluc = 4'bxxxx; //jr

wreg = 1'b0; // Write to the register file

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

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

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

regrt = 1'bx; // Destination register address

sext = 1'bx;

jal = 1'bx;

shift = 1'bx;

pcsrc = 2'b10;

end

endcase

end

6'b001000: //addi

begin

wreg <= 1'b1;

regrt <= 1'b1;

jal <= 1'b0;

m2reg <= 1'b0;

shift <= 1'b0;

aluimm <= 1'b1;

sext <= 1'b1;

aluc <= 4'b0010;

wmem <= 1'b0;

pcsrc <= 2'b00;

end

6'b00001100: //andi

begin

wreg <= 1'b1;

regrt <= 1'b1;

jal <= 1'b0;

m2reg <= 1'b0;

shift <= 1'b0;

aluimm <= 1'b1;

sext <= 1'b0;

aluc <= 4'b0000;

wmem <= 1'b0;

pcsrc <= 2'b00;

end

6'b001101: //ori

begin

wreg <= 1'b1;

regrt <= 1'b1;

jal <= 1'b0;

m2reg <= 1'b0;

shift <= 1'b0;

aluimm <= 1'b1;

sext <= 1'b0;

aluc <= 4'b0001;

wmem <= 1'b0;

pcsrc <= 2'b00;

end

6'b001110: //xori

begin

wreg <= 1'b1;

regrt <= 1'b1;

jal <= 1'b0;

m2reg <= 1'b0;

shift <= 1'b0;

aluimm <= 1'b1;

sext <= 1'b0;

aluc <= 4'b0011;

wmem <= 1'b0;

pcsrc <= 2'b00;

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

jal = 1'b0;

shift = 1'b0;

pcsrc = 2'b00;

sext = 1'b1;

end

6'b101011: //SW instruction

begin

wreg = 1'b0;

m2reg = 1'b0;

wmem = 1'b1;

aluc = 4'b0010;

aluimm = 1'b1;

regrt = 1'bx;

jal = 1'b0;

shift = 1'b0;

pcsrc = 2'b00;

sext = 1'b1;

end

6'b000100: //BEQ instruction

begin

wreg = 1'b0;

//m2reg = 1'bx; /\*\*\*/

wmem = 1'b0;

aluc = 4'b0011;

aluimm = 1'b0;

regrt = 1'bx;

jal = 1'bx;

shift = 1'b0;

sext = 1'b1;

if (rsrtequ == 1)

pcsrc <= 2'b01;

else

pcsrc <= 2'b00;

end

6'b000101: //BNE instruction

begin

wreg = 1'b0;

// m2reg = 1'bx; /\*\*\*/

wmem = 1'b0;

aluc = 4'b0011;

aluimm = 1'b0;

regrt = 1'bx;

jal = 1'bx;

shift = 1'b0;

sext = 1'b1;

if (rsrtequ == 1)

pcsrc <= 2'b01;

else

pcsrc <= 2'b00;

end

6'b001111: //lui

begin

wreg <= 1'b1;

regrt <= 1'b1;

jal <= 1'b0;

m2reg <= 1'b0;

shift <= 1'bx;

aluimm <= 1'b1;

sext <= 1'bx;

aluc <= 4'b0100;

wmem <= 1'b0;

pcsrc <= 2'b00;

end

6'b000010: //j

begin

wreg <= 1'b0;

regrt <= 1'bx;

jal <= 1'bx;

m2reg <= 1'bx;

shift <= 1'bx;

aluimm <= 1'bx;

sext <= 1'bx;

aluc <= 4'bxxxx;

wmem <= 1'b0;

pcsrc <= 2'b11;

end

6'b000011: //jal

begin

wreg <= 1'b1;

regrt <= 1'bx;

jal <= 1'b1;

//m2reg <= 1'bx; /\*\*\*\*/

shift <= 1'bx;

aluimm <= 1'bx;

sext <= 1'bx;

aluc <= 4'bxxxx;

wmem <= 1'b0;

pcsrc <= 2'b11;

end

endcase

end

//stall

always @ (\*)

begin

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

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

wreg = 1'b0;

wmem = 1'b0;

wpcir = 1'b0;

end

else begin

wpcir = 1'b1;

end

// forwarding

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

fwda = 2'b01;

end

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

fwda = 2'b10;

end

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

fwda = 2'b11;

end

else begin

fwda = 2'b00;

end

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

fwdb = 2'b01;

end

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

fwdb = 2'b10;

end

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

fwdb = 2'b11;

end

else begin

fwdb = 2'b00;

end

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] drn, // Destination register address

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

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

input [31:0] dimm32, // 32-bit immediate value

input clk, // Clock signal

//EC Input

input [31:0] dpc4,

input shift, jal,

// 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] ern0, // Output for destination register address

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

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

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

//EC Output

output reg [31:0] epc4,

output reg ejal, eshift

);

// 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;

ern0 = drn;

ea = da;

eb = db;

eimm32 = dimm32;

epc4 = dpc4;

ejal = jal;

eshift = shift;

end

endmodule // End of the module

module EXE\_ProgramCounter(

input [31:0] epc4,

output reg [31:0] epc8

);

always @ (\*)

begin

epc8 <= epc4 + 32'b00000000000000000000000000000100;

end

endmodule

module EXE\_ALUMux2(

input [31:0] sa,

input [31:0] ea,

input eshift,

output reg [31:0] a

);

always @ (\*)

begin

case(eshift)

2'b0: a = ea;

2'b1: a = sa;

endcase

end

endmodule

module ALUMux(

input [31:0] eb, // 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: b <= eb;

1'b1: b <= eimm32;

endcase

end

endmodule // End of the module

module ALU(

input [31:0] a, // 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 = a & b; // AND operation

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

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

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

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

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

4'b0111: r = b << a;//sll

4'b1110: r = b >> a; //srl

4'b1001: r = $signed(b) >>> a; //sra

4'b0100: r = b << 16; //lui

endcase

end

endmodule // End of the module

module EXE\_ALUoutMux(

input [31:0] epc8,

input [31:0] r,

input ejal,

output reg [31:0] ealu

);

always @ (\*)

begin

case(ejal)

1'b0: ealu = r;

1'b1: ealu = epc8;

endcase

end

endmodule

module EXE\_f(

input [4:0] ern0,

input ejal,

output reg [4:0] ern

);

always @ (\*)

begin

case(ejal)

1'b0: ern = ern0;

1'b1: ern = 5'b11111;

endcase

end

endmodule

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] ern, // Destination register address

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

input [31:0] eb, // 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] mrn, // Output for destination register address

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

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

);

always @ (posedge clk)

begin

mwreg = ewreg;

mm2reg = em2reg;

mwmem = ewmem;

mrn = ern;

malu = ealu;

mb = eb;

end

endmodule // End of the module

module DataMemory( // data memory, ram

input clk, // clock

input [31:0] malu, // ram address (addr)

input [31:0] mb, // data in (to memory) (datain)

input mwmem, // write enable (we)

output [31:0] mdo // data out (from memory) (dataout)

);

reg [31:0] ram [0:31]; // ram cells: 32 words \* 32 bits

assign mdo = ram[malu[6:2]]; // use 5-bit word address

always @ (posedge clk) begin

if (mwmem) ram[malu[6:2]] = mb; // write ram

end

integer i;

initial begin // ram initialization

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

endmodule

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] mrn, // Destination register address

input [31:0] malu, // 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] wrn, // Output for destination register address

output reg [31:0] walu, // 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;

wrn = mrn;

walu = malu;

wdo = mdo;

end

endmodule // End of the module

module WbMux(

input [31:0] walu,

input [31:0] wdo,

input wm2reg,

output reg [31:0] wbData

);

always @ (\*)

begin

if (wm2reg == 0)

wbData = walu;

else

wbData = wdo;

end

endmodule

**Schematic:  
 RTL:**

**A green and white diagram

Description automatically generated with medium confidence**

**Synthesis:**

**A green and white computer generated image

Description automatically generated with medium confidence**

**I/O:**

**A screenshot of a game

Description automatically generated**

**Floorplanning:**

A screenshot of a video game

Description automatically generated

**Waveform:**

A screen shot of a computer

Description automatically generated

Note the waveforms below are zoomed is pieces piece by piece of the full waveform. This waveform displays the same behavior as Figure 7 on the project doc. The red in the beginning is the same as the 0’s on the doc, they are red because they are uninitialized.

A black screen with green and red lines

Description automatically generated

A black screen with green lines

Description automatically generated

A black screen with green lines

Description automatically generated

A black screen with green lines

Description automatically generated

A black screen with green lines

Description automatically generated

A black screen with green lines

Description automatically generated

A black screen with green lines

Description automatically generated

A black screen with green lines

Description automatically generated

A black screen with green lines

Description automatically generated

Note that the waveform continues until 1,000ns but continues to produce no results as the only instructions passed through dinstOut are nops (00000000) and the finish instruction (80000019).

The purpose of this waveform is to iterate through a list of instructions. It is aimed to check the 20 instructions. The main part of the test program is a subroutine in which four 32-bit memory words are summed by a for loop. After returning from the subroutine, the sum is stored in the data memory by a sw instruction. A code pattern that causes pipeline stall is also prepared within the loop. Word address is used to assign the content of each word (a 32-bit instruction). The parenthesized hexadecimal number in the center of each line is the byte address (PC). Looking through the dinstOut instructions, you can properly see the stalls and nops properly occur where needed, iterations of the for loop, and proper display of the instruction set completing.