**Final Project Report**

**CMPEN 331**

Section 2

Austin Schaefer

December 20th 2019

**Abstract**

For our final project, we were asked to implement forwarding in our CPU to avoid data hazards. Throughout the semester we were assigned to add a part of the CPU for every lab we were assigned.

For this final lab, I added a four-input multiplexer as a new module and new inputs and outputs into the control unit. I followed **figure 4** from the lab handout, to be specific.

I connected my register file to my four-input multiplexer. This either passes forward a register from the register file, or it uses forwarding to pass forward an output from the ALU.

What determines the forwarding is the control unit. The control unit takes in the rs, rt, and rd registers, and compares rs and rt to rd. If they match, then the control unit utilizes forwarding, by inputting 0, 1, 2, or 3 to the multiplexer.

In the multiplexer, 0 corresponds with qa and qb, while 1 and 2 correspond to sending forward the ALU output. 3 corresponds with the output from data memory. This multiplexer forwards the desired output, depending on the control unit’s input.

Of course, I also updated my ALU to support the operations required. I also added a tcl file to allow my project to generate bitstream.

There were a few errors I had to fix from previous labs to get this lab to work. My pipelining was not perfect; my Execute and Memory stages were running in the same clock cycle. I fixed this so that my Memory stage comes 1 clock cycle after the Execute stage. This fixed any issues I had with forwarding and writing back to the register file. At the end of the project, I learned to effectively use pipelining to prevent any data hazards by forwarding.

**Introduction**

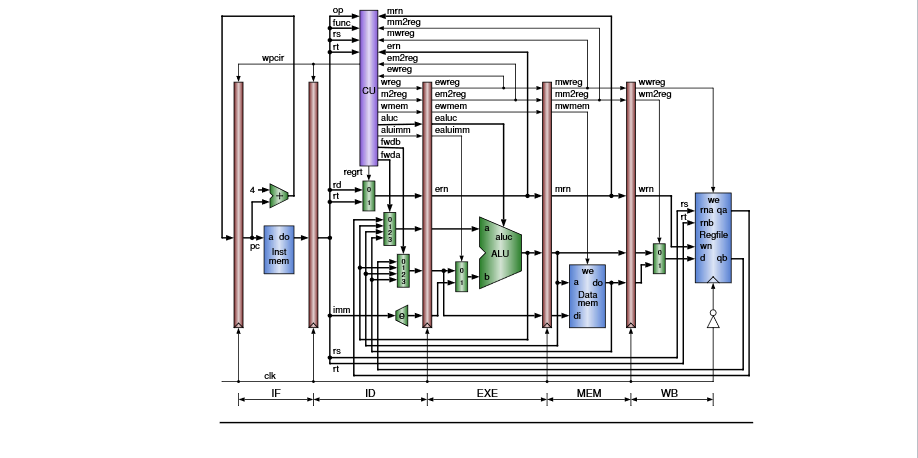
Here is an overview of the project, which we worked on throughout the semester:

The first register, the PC register, is used to output the next program counter. This is initialized in the adder module, in which I add 4 to get the next instruction. The current instruction goes into instruction memory, where it finds the specific instruction at that address.

This instruction is sent to the IF/ID register, where it will just output the instruction, which is then used to check the register files using rs and rt. The register file will output the contents of rs and rt. The control unit then decides if those outputs are correct, or if it will use forwarding. After making sure the rs and rt registers are correct, it passes those onto the ID/EXE register. It also passes forward a sign extension.

In the EXE stage, the ALU executes any necessary arithmetic operations. The output of the ALU is passed on to the EXE/MEM register. In the MEM stage, it will access data memory if necessary (for example, a load word instruction accesses memory now). Any data accessed from memory will be passed forward, along with the output of the ALU.

In the WB stage, any data from memory and ALU will be sent back to the register file

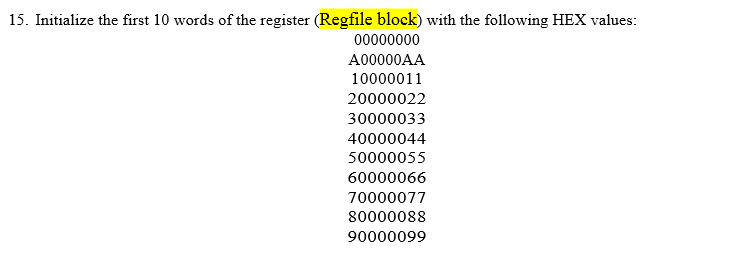
This was the diagram I followed to implement forwarding. This is figure 4 from the final project lab handout. This reflects what I have accomplished this semester.

These were the instructions I used to test my code. I got this from the honors section of Lab 5. As you can see, forwarding is required for the “sub” and “or” instructions to calculate the correct values.

A close up of text on a white background

Description automatically generated

I used this data to initialize my registers in the register file. As you can see, I also got this data from the lab handout:



**Design Code**

//PC register for pipelining

module PC\_register(input clk, input [31:0] pc\_in, output reg [31:0] pc\_out

);

always @(negedge clk)

begin

pc\_out = pc\_in;

end

endmodule

// Increment pc by 4, fetch next instruction

module adder(a, pc);

input [31:0] a; // old program counter

output reg [31:0] pc; // new program counter

initial begin

pc = 100;

end

// increment pc by 4 to get next instruction

always @(a)

begin

pc = a + 4;

end

endmodule

//Put instructions in instruction memory, retrieve instruction and output

module Instruction\_memory(a, do);

input [31:0] a; // input program counter

output reg [31:0] do; // output instruction binary

reg [31:0] IM [0:116]; // 2 dimensional 32x105 array

// Put the two instructions we want to read, put in memory

initial begin

IM[100] = 32'b00000000001000100001100000100000;

IM[104] = 32'b00000001001000110010000000100010;

IM[108] = 32'b00000000011010010010100000100101;

IM[112] = 32'b00000000011010010011000000100110;

IM[116] = 32'b00000000011010010011100000100100;

//IM[6'd00] = 32'h34040050; // (00) main: ori $4, $zero, 0x50 no stall $4 = 0x0 | 0x50 = 0x50

//IM[6'd04] = 32'h8c880000; // (04) lw $8, 0($4) no stall $8 = ram[$4] = ram[0x50] = 0xa3

//IM[6'd08] = 32'h20840004; // (08) addi $4, $4, 4 no stall $4 = $4 + 4 = 0x54

//IM[6'd12] = 32'h8c890000; // (0c) lw $9, 0($4) no stall $9 = ram[$4] = ram[0x54] = 0x27

//IM[6'd16] = 32'h01094020; // (10) add $8, $8, $9 stall $8 = $8 + $9 = 0xa3 + 0x27 = 0xca

//IM[6'd20] = 32'h20840004; // (14) addi $4, $4, 4 no stall $4 = $4 + 4 = 0x58

//IM[6'd24] = 32'hac880000; // (18) sw $8, 0($4) no stall ram[$4] = $8 -> ram[0x58] = 0xca

end

// output the binary instruction at address [a]

always @(\*)

begin

do = IM[a];

end

endmodule

//IF/ID register for pipelining

module IF\_ID(

input clk,

input [31:0] do\_in,

output reg [31:0] do\_out

);

always @(negedge clk)

begin

do\_out = do\_in;

end

endmodule

// Output control based on OP field

module controlunit(

input [31:0] do\_in,

input [4:0] mrnCU\_in,

input mm2regCU\_in,

input mwregCU\_in,

input [4:0] ernCU\_in,

input em2regCU\_in,

input ewregCU\_in,

output reg regWrite,

output reg memToReg,

output reg memWrite,

output reg [3:0] aluOP, // aluc

output reg aluSrc, // aluimm

output reg [1:0] fwdb,

output reg [1:0] fwda,

output reg regrt

);

// wires for opcode and func

wire [5:0] op = do\_in[31:26];

wire [5:0] func = do\_in[5:0];

wire [4:0] rs = do\_in[25:21];

wire [4:0] rt = do\_in[20:16];

always @(\*) begin

if (op == 6'b100011) // lw

begin

regWrite <= 1;

memToReg <= 1;

memWrite <= 0;

aluOP <= 4'b0000;

aluSrc <= 1;

regrt <= 1;

end

else if (op == 6'b000000) // r-type

begin

regWrite = 1;

memToReg <= 0;

memWrite <= 0;

aluSrc <= 0;

regrt <= 0;

if (func == 6'b100000) // add

aluOP = 4'b0010;

else if (func == 6'b100010) // sub

aluOP = 4'b0110;

else if (func == 6'b100101) // or

aluOP = 4'b0001;

else if (func == 6'b100110) // xor

aluOP = 4'b1101;

else if (func == 6'b100100) // and

aluOP = 4'b0000;

end

if (rs == ernCU\_in) begin

fwda = 1;

end

else if (rs == mrnCU\_in && mwregCU\_in == 1 && mm2regCU\_in == 0) begin

fwda = 2;

end

else if (rs == mrnCU\_in && mwregCU\_in == 0 && mm2regCU\_in == 1) begin

fwda = 3;

end

else begin

fwda = 0;

end

if (rt == ernCU\_in) begin

fwdb = 1;

end

else if (rt == mrnCU\_in && mwregCU\_in == 1 && mm2regCU\_in == 0) begin

fwdb = 2;

end

else if (rt == mrnCU\_in && mwregCU\_in == 0 && mm2regCU\_in == 1) begin

fwdb = 3;

end

else begin

fwdb = 0;

end

end

endmodule

//Register file initialized to 0. Output rs and rt

module Reg\_file(clk, we, rna, rnb, wn, data, qa, qb);

input clk;

input we;

input [4:0] rna;

input [4:0] rnb;

input [4:0] wn;

input [31:0] data;

output reg [31:0] qa;

output reg [31:0] qb;

reg [31:0] regs [0:31]; // 32 bit array of 32 bit arrays

integer i;

// Initialize all regs to 0 with for loop

initial begin

for (i = 0; i < 32; i = i + 1)

begin

regs[i] = 0;

end

regs[0] = 32'h00000000;

regs[1] = 32'hA00000AA;

regs[2] = 32'h10000011;

regs[3] = 32'h20000022;

regs[4] = 32'h30000033;

regs[5] = 32'h40000044;

regs[6] = 32'h50000055;

regs[7] = 32'h60000066;

regs[8] = 32'h70000077;

regs[9] = 32'h80000088;

regs[10] = 32'h90000099;

end

always @(\*) begin

qa = regs[rna];

qb = regs[rnb];

end

always @(posedge clk) begin // Change to posedge so that it writes back to register before reading

if(we) begin regs[wn] = data; end

end

endmodule

module mux(do, regrt, out);

input [31:0] do; // 32 bit inst

input regrt; // from control unit

output wire [4:0] out; // output rt or rd

assign out = regrt ? do[20:16] : do[15:11];

endmodule

// Sign extend leading bit

module sign\_ext(do, out);

input [31:0] do;

output reg [31:0] out;

wire [15:0] in;

assign in = do[15:0];

always @ (\*)

begin

out[15:0] <= in;

if (in[15] == 1)

out[31:16] <= 16'b1111111111111111;

else

out[31:16] <= 16'b0000000000000000;

end

endmodule

module MuxFourWay(

input [1:0] fwda,

input [31:0] qa,

input [31:0] ealu,

input [31:0] malu,

input [31:0] datamem\_do,

output wire [31:0] forwarded\_a

);

assign forwarded\_a = fwda[1] ? (fwda[0] ? datamem\_do : malu) : (fwda[0] ? ealu : qa);

endmodule

//ID/EXE register for pipelining

module ID\_EXE(

input clk,

input wreg,

input m2reg,

input wmem,

input [3:0] aluOp,

input aluimm,

input [4:0] mux\_to\_id\_exe,

input [31:0] qa,

input [31:0] qb,

input [31:0] sign\_extend\_to\_id\_exe,

output reg ewreg,

output reg em2reg, ewmem,

output reg [3:0] ealuc,

output reg ealuimm,

output reg [4:0] mux\_out,

output reg [31:0] qa\_out,

output reg [31:0] qb\_out,

output reg [31:0] sign\_extend\_out

);

// outputs

always @(negedge clk)

begin

ewreg = wreg;

em2reg = m2reg;

ewmem = wmem;

ealuimm = aluimm;

ealuc = aluOp;

qa\_out = qa;

qb\_out = qb;

sign\_extend\_out = sign\_extend\_to\_id\_exe;

mux\_out = mux\_to\_id\_exe;

end

endmodule

module ALU(

input [31:0] qa\_in,

input [3:0] ealuc,

input [31:0] mux2in,

output reg [31:0] ALU\_out

);

always @(\*)

begin

if (ealuc == 4'b10) // add

ALU\_out = qa\_in + mux2in;

else if (ealuc == 4'b0110) // subtract

ALU\_out = qa\_in - mux2in;

else if (ealuc == 4'b0000) // AND

ALU\_out = qa\_in & mux2in;

else if (ealuc == 4'b0001) // OR

ALU\_out = qa\_in | mux2in;

else if (ealuc == 4'b1101) // XOR

ALU\_out = qa\_in ^ mux2in;

else

ALU\_out = mux2in;

end

endmodule

module mux2(

input ealuimm,

input [31:0] qb\_in,

input [31:0] signextendin,

output reg [31:0] mux2out

);

always@(\*) begin

assign mux2out = ealuimm ? signextendin : qb\_in;

end

endmodule

module EXE\_MEM(

input clk,

input ewreg,

input em2reg,

input ewmem,

input [4:0] mux1\_in,

input [31:0] ALU\_in,

input [31:0] qb\_in,

output reg mwreg,

output reg mm2reg,

output reg mwmem,

output reg [4:0] mux1\_out,

output reg [31:0] ALU\_out,

output reg [31:0] qb\_out

);

always @(negedge clk)

begin

mwreg = ewreg;

mm2reg = em2reg;

mwmem = ewmem;

mux1\_out = mux1\_in;

ALU\_out = ALU\_in;

qb\_out = qb\_in;

end

endmodule

module Data\_memory(

input mwmem, // write enable

input [31:0] ALU\_in, // address / ALU output to data mem

input [31:0] qb\_in, // data in/ qb from regfile

output wire [31:0] datamem\_out

);

reg [31:0] Data [0:200]; // 32 \* 201 Data RAM

assign datamem\_out = Data[ALU\_in[6:2]]; // use 5-bit word address

always @ (mwmem) begin

if (mwmem) Data[ALU\_in[6:2]] = qb\_in; // write ram

end

integer i;

initial begin // ram initialization

for (i = 0; i < 32; i = i + 1)

Data[i] = 0;

Data[0] = 32'hA00000AA;

Data[4] = 32'h10000011;

Data[8] = 32'h20000022;

Data[12] = 32'h30000033;

Data[16] = 32'h40000044;

Data[20] = 32'h50000055;

Data[24] = 32'h60000066;

Data[28] = 32'h70000077;

Data[32] = 32'h80000088;

Data[36] = 32'h90000099;

end

endmodule

module MEM\_WB(

input clk,

input mwreg,

input mm2reg,

input [4:0] mux1\_in,

input [31:0] ALU\_in,

input [31:0] datamem\_in,

output reg wwreg,

output reg wm2reg,

output reg [4:0] mux1\_out,

output reg [31:0] ALU\_out,

output reg [31:0] datamem\_out

);

always @(negedge clk)

begin

wwreg = mwreg;

wm2reg = mm2reg;

mux1\_out = mux1\_in;

ALU\_out = ALU\_in;

datamem\_out = datamem\_in;

end

endmodule

**Test Bench**

`timescale 1ns / 1ps

module test\_bench;

// clock

wire clk;

// pc

wire [31:0] next\_pc;

wire [31:0] current\_pc;

// 32 bit Instruction

wire [31:0] do\_in;

wire [31:0] do\_out;

// Control unit inputs and outputs

wire regWrite;

wire memToreg;

wire memWrite;

wire [3:0] aluOp;

wire aluSrc;

wire [4:0] mrnCU\_in;

wire mm2regCU\_in;

wire mwregCU\_in;

wire [4:0] ernCU\_in;

wire em2regCU\_in;

wire ewregCU\_in;

wire eregWrite;

wire ememToreg;

wire ememWrite;

wire [3:0] ealuc;

wire ealuimm;

wire [1:0] fwdb;

wire [1:0] fwda;

wire regrt;

// 1st multiplexer

wire [4:0] mux;

wire [4:0] mux\_out;

// sign extend

wire [31:0] sign\_extend;

wire [31:0] sign\_extend\_out;

//regfile inputs

wire [4:0] rna;

wire [4:0] rnb;

wire [4:0] wn\_regfile;

wire [31:0] do\_regfile;

wire we;

// regfile and ID/EXE outputs

wire [31:0] qa;

wire [31:0] qb;

// 4 Way mux outputs

wire [31:0] forwarded\_qb\_rt;

wire [31:0] forwarded\_qa\_rs;

// ID/EXE outputs

wire [31:0] qa\_out;

wire [31:0] qb\_out;

// 2nd multiplexer

wire [31:0] mux2\_out;

// ALU

wire [31:0] ALU\_out;

// EXE/MEM outputs

wire mwreg;

wire mm2reg;

wire mwmem;

wire [4:0] mux1\_exemem\_out;

wire [31:0] r\_exemem\_out;

wire [31:0] qb\_exemem\_out;

//data memory

wire [31:0] datamem\_out;

// MEM/WB

wire wwreg;

wire wm2reg;

wire[4:0] mux1\_memwb\_out;

wire[31:0] r\_memwb\_out;

wire[31:0] do\_memwb\_out;

// WB multiplexer

wire [31:0] wbmux\_to\_regfile;

test\_clk test\_clk(clk);

// Tick the clock

adder adder(current\_pc, next\_pc);

// adder gives PC register new pc

PC\_register PC\_register(clk, next\_pc, current\_pc);

// PC register outputs old pc while taking in new pc

Instruction\_memory Instruction\_memory(current\_pc, do\_in);

// IM takes in current inst address, outputs instruction at that address

IF\_ID IF\_ID(clk, do\_in, do\_out);

// Takes in the 32 bit instruction, outputs 32 bit instruction

assign mrnCU\_in = mux1\_exemem\_out;

assign mm2regCU\_in = mm2reg;

assign mwregCU\_in = mwreg;

assign ernCU\_in = mux\_out;

assign em2regCU\_in = ememToreg;

assign ewregCU\_in = eregWrite;

controlunit controlunit(do\_out, mrnCU\_in, mm2regCU\_in, mwregCU\_in, ernCU\_in, em2regCU\_in, ewregCU\_in, regWrite, memToreg, memWrite, aluOp, aluSrc, fwdb, fwda, regrt);

// Takes in instruction, outputs hard-coded values

mux muxmodule(do\_out, regrt, mux);

// Get rd and rt from do\_out. If 0, output rd, if 1, output rt

assign rna = do\_out[25:21]; //rs

assign rnb = do\_out[20:16]; //rt

assign wn\_regfile = mux1\_memwb\_out;

assign do\_regfile = wbmux\_to\_regfile;

assign we = wwreg;

Reg\_file Reg\_file(clk, we, rna, rnb, wn\_regfile, do\_regfile, qa, qb);

// Get rs and rt from instruction, put into qa and qb

MuxFourWay mux4qb\_rt(fwdb, qb, ALU\_out, r\_exemem\_out, datamem\_out, forwarded\_qb\_rt);

MuxFourWay mux4qa\_rs(fwda, qa, ALU\_out, r\_exemem\_out, datamem\_out, forwarded\_qa\_rs);

sign\_ext sign\_ext\_module(do\_out, sign\_extend);

// Sign extend the instruction

ID\_EXE ID\_EXE(clk, regWrite, memToreg, memWrite, aluOp, aluSrc, mux,

forwarded\_qa\_rs, forwarded\_qb\_rt, sign\_extend, eregWrite, ememToreg, ememWrite, ealuc, ealuimm,

mux\_out, qa\_out, qb\_out, sign\_extend\_out);

// Many inputs and outputs for ID/EXE

mux2 mux2(ealuimm, qb\_out, sign\_extend\_out, mux2\_out);

ALU ALU(qa\_out, ealuc, mux2\_out, ALU\_out);

EXE\_MEM EXE\_MEM(clk, eregWrite, ememToreg, ememWrite, mux\_out, ALU\_out, qb\_out,

mwreg, mm2reg, mwmem, mux1\_exemem\_out, r\_exemem\_out, qb\_exemem\_out);

Data\_memory Data\_memory(mwmem, r\_exemem\_out, qb\_exemem\_out, datamem\_out);

MEM\_WB MEM\_WB(clk, mwreg, mm2reg, mux1\_exemem\_out, r\_exemem\_out, datamem\_out,

wwreg, wm2reg, mux1\_memwb\_out, r\_memwb\_out, do\_memwb\_out);

mux2 wbmux(wm2reg, r\_memwb\_out, do\_memwb\_out, wbmux\_to\_regfile);

endmodule

`timescale 1ns / 1ps

module test\_clk(

output reg clk

);

initial begin

clk = 0;

end

always

#5 clk = !clk; // invert clock

endmodule

**Waveforms**

**A screen shot of a monitor

Description automatically generatedA picture containing window

Description automatically generated**

A flat screen tv

Description automatically generated

**Floor Planning**

**After running synthesis**

**A screenshot of a computer screen

Description automatically generated**

**After running implementation**

**A screen shot of a computer

Description automatically generated**

**I/O Planning**

A screenshot of a computer screen

Description automatically generated

**Schematic**

**A screenshot of a social media post

Description automatically generated**

**Bit Stream**

A screenshot of a social media post

Description automatically generatedA screenshot of a social media post

Description automatically generated