**clLab # 5 - MIPS Datapath for R-type and I- Type Instructions**

CECS 341 – Computer Architecture Organization

Shawn Joseph,  ID# 025671644

Ty Tanaka, ID# 018208955

Yiang Shen,ID# 025794325

Professor: Jose Aceves

Logo

Description automatically generated

California State University, Long Beach

College of Engineering

1250 Bellflower Blvd, Long Beach, CA 90840

Lab Due Date 11/17/2020

**Goal/Objective**

The goal for this lab was to learn more about modifying the existing MIPS datapath we had from lab 4 to pass both R-type instructions and I-type through the module to perform certain calculations. It also was to learn how to interpret R-type instructions and I-type by taking the 32-bit inputs and seeing what each group of bits represented, such as the operation, input registers, and output register, and parsing the input to generate the output.

**Technical Description/Steps**

First, we had to import our previous ALU module from the previous lab 4 to add I-type instruction. There are modules we borrow from the last lab. Such as the control, program counter register, instruction memory, register file, and the overarching datapath. The instruction memory reads the input data and sends it to the control and register file. The register file sends in 2 32-bit inputs to the ALU for the a and b inputs in order to give it the data it needs to do the calculations on. The program counter adder also takes in the initial input and keeps track of the calculations at intervals of 4 by adding 4 to the din value and assigning it to the dout output, then passes it back into the program counter to perform the next operation.

[Control Modification]:

We had to modify the control unit to send more signals to allow for I-type instructions. In addition, we had to remove the PCADD module, replacing it with a sign extender, Shift Left modifier, 2 add circuits, and some other circuits to allow for branching operations to be performed. In addition, we had to modify the datapath to add several new wires and connections.

The control interprets the instruction so that the module knows what calculation to perform, sending information back to the register file and telling the ALU what instruction to perform. So other than the existing function cases for R-type instructions. We add cases for Op which are used for I-type instructions. These cases run from 6h08 to 6h0B, which total has 10 cases. In each case, we assign the RegDst always to be 0 since I-type instructions write back address is always from bits 20-16. We assign RegWrite to be 1 when the instruction is written back to the register file(rt). We assign Branch to 1 when the instruction performs a branch. We assign MemtoReg to 0 when we perform the operation that needs to write back to our register file in order to pass the value of ALU result back to register.

[Design of Datapath]:

To design the datapath, we used the module as a basis and modified any components with extra ports not in the original datapath so they had the correct amount of outputs. Afterwards, we added the new components to the datapath such as the muxes and the sign extension unit. Following that, we made each of the modules function by changing the code for each module so it would work properly.

They are we, br, dit, RegDst, MemRead ,MemtoReg ,MemWrite ,ALUSrc. We also add ALUin, SEval, rd, wd, rt, and shl2. For objects, we add sll(ShiftLeftTwo) and adder(Mux32x1). We keep the Instruction memory and regfile32 as the same. We add an object m(Mux4x1). We update our control to connect to new wires we added.

[Design of MUX]:

We have two 5 bits input which we named them *zero* and one. We have another 1 bit input named it *ctrl*. And we have a 5 bits output named *out*. Then we assign them when *ctrl* is 0, we set *out* to have the same value as *zero*.

if the *ctrl* equal to 1, then *out* is assign to have the same value as *one*. In addition, we have a second MUX module which takes in 2 32 bit inputs internally named zero and one and a control bit, and returns one if the control bit is 1 and zero if the control bit is zero.

Waveform diagram:

Graphical user interface

Description automatically generated

Datapath schematic:

Diagram

Description automatically generated

Mux4x1 schematic:

Chart, diagram

Description automatically generated

SignExtend schematic:

A picture containing diagram

Description automatically generated

Mux32x1 schematic:

Diagram

Description automatically generated

ShiftLeft2 schematic:

Diagram

Description automatically generated

Control unit schematic:

Diagram, schematic

Description automatically generatedDiagram, schematic

Description automatically generated

Program counter:

Diagram

Description automatically generated

ALU:

Diagram, schematic

Description automatically generated

Output:

Initial Memory values:

0: 00

1: 00

2: 00

3: 00

4: ff

5: ff

6: ff

7: ff

8: 12

9: 34

10: 56

11: 78

12: 00

13: 00

14: 00

15: 08

16: 00

17: 00

18: 00

19: 01

20: 00

21: 00

22: 00

23: 03

24: xx

25: xx

26: xx

27: xx

28: xx

29: xx

30: xx

31: xx

32: xx

33: xx

34: xx

35: xx

36: xx

37: xx

38: xx

39: xx

40: xx

41: xx

42: xx

43: xx

44: xx

Time: 30000, Output: ffffffff

Time: 50000, Output: 12345678

Time: 70000, Output: 00000008

Time: 90000, Output: 00000001

Time: 110000, Output: 00000003

Time: 130000, Output: 00000001

Time: 150000, Output: 00000000

Time: 170000, Output: 1234567f

Time: 190000, Output: 00000000

Time: 210000, Output: ffffffff

Time: 230000, Output: 00000002

Time: 250000, Output: 00000002

Time: 270000, Output: 00000002

Time: 290000, Output: 00000002

Time: 310000, Output: 00000000

Time: 330000, Output: 1234567f

Time: 350000, Output: 00000000

Time: 370000, Output: ffffffff

Time: 390000, Output: 00000000

Time: 410000, Output: 00000000

Time: 430000, Output: 00000018

Time: 450000, Output: 0000001c

Time: 470000, Output: 00000020

Time: 490000, Output: 00000024

Time: 510000, Output: 00000028

Time: 530000, Output: 0000002c

Register 0: 00000000

Register 1: 00000000

Register 2: 00000000

Register 3: 00000000

Register 4: 00000000

Register 5: 00000000

Register 6: 00000000

Register 7: 00000000

Register 8: 00000002

Register 9: 00000000

Register 10: 1234567f

Register 11: 00000000

Register 12: ffffffff

Register 13: 00000000

Register 14: 0000000f

Register 15: 00000010

Register 16: 00000011

Register 17: 00000012

Register 18: 00000013

Register 19: 00000014

Register 20: 00000015

Register 21: 00000016

Register 22: 00000017

Register 23: 00000018

Register 24: 00000019

Register 25: 0000001a

Register 26: 00000000

Register 27: 00000000

Register 28: 00000000

Register 29: 00000000

Register 30: 00000000

Register 31: 00000000

Final Memory values:

0: 00

1: 00

2: 00

3: 00

4: ff

5: ff

6: ff

7: ff

8: 12

9: 34

10: 56

11: 78

12: 00

13: 00

14: 00

15: 08

16: 00

17: 00

18: 00

19: 01

20: 00

21: 00

22: 00

23: 03

24: 00

25: 00

26: 00

27: 02

28: 00

29: 00

30: 00

31: 00

32: 12

33: 34

34: 56

35: 7f

36: 00

37: 00

38: 00

39: 00

40: ff

41: ff

42: ff

43: ff

44: xx

HDL code:

module ProgramCounter(

input clk,

input reset,

input [31:0] Din,

output reg[31:0] Dout

);

always @(posedge clk) begin

if(reset)

Dout = 0;

else

Dout = Din;

end

endmodule

module ShiftLeftTwo(

input [31:0] in,

output reg [31:0] out

);

always @(in) begin

out <= in << 2;

end

endmodule

module Mux32x1(

input [31:0] zero,

input [31:0] one,

input ctrl,

output reg [31:0] out

);

always @(zero or one or ctrl) begin

case(ctrl)

1'b0: out <= zero;

1'b1: out <= one;

//default: $display("Error in Mux32x1");

endcase

end

endmodule

module Mux4x1(

input [4:0] zero,

input [4:0] one,

input ctrl,

output reg [4:0] out

);

always @(zero or one or ctrl) begin

case(ctrl)

1'b0: out <= zero;

1'b1: out <= one;

//default: $display("Error in Mux4x1");

endcase

end

endmodule

module SignExtend(

input [15:0] in,

output reg [31:0] out

);

always @(in) begin

out <= {{16{in[15]}}, in[15:0]};

end

endmodule

module Control(

input [5:0] Op,

input [5:0] Func,

output reg RegDst, MemRead, MemToReg, MemWrite, ALUSrc, RegWrite,

output reg [1:0] Branch,

output reg [3:0] ALUCntl

);

always @(\*) begin

case(Op)

6'b0:begin

RegWrite <= 1'b1;

RegDst <= 1'b1;

Branch <= 2'b0;

MemRead <= 1'b0;

MemToReg <= 1'b0;

MemWrite <= 1'b0;

ALUSrc <= 1'b0;

case(Func)

6'h20:begin //add

ALUCntl <= 4'b1010;

end

6'h21:begin //addu

ALUCntl <= 4'b0010;

end

6'h22:begin //sub

ALUCntl <= 4'b1110;

end

6'h23:begin //subu

ALUCntl <= 4'b0110;

end

6'h24:begin //and

ALUCntl <= 4'b0000;

end

6'h25:begin //or

ALUCntl <= 4'b0001;

end

6'h26:begin //xor

ALUCntl <= 4'b0011;

end

6'h27:begin //nor

ALUCntl <= 4'b1100;

end

6'h2A:begin //SLT

ALUCntl <= 4'b1111;

end

6'h2B:begin //SLTU

ALUCntl <= 4'b0100;

end

default:begin //invalid code

//$display("Error! Check control file");

end

endcase

end

6'h08:begin //addi

RegWrite <= 1'b1;

Branch <= 2'b0;

MemWrite <= 1'b0;

MemRead <= 1'b0;

MemToReg <= 1'b0;

RegDst <= 1'b0;

ALUCntl <= 4'b1010;

ALUSrc <= 1'b1;

end

6'h09:begin //addiu

Branch <= 2'b0;

RegWrite <= 1'b1;

MemWrite <= 1'b0;

MemRead <= 1'b0;

MemToReg <= 1'b0;

RegDst <= 1'b0;

ALUCntl <= 4'b0010;

ALUSrc <= 1'b1;

end

6'h0C:begin //And immediate

RegWrite <= 1'b1;

Branch <= 2'b0;

MemWrite <= 1'b0;

MemRead <= 1'b0;

MemToReg <= 1'b0;

RegDst <= 1'b0;

ALUCntl <= 4'b0000;

ALUSrc <= 1'b1;

end

6'h0D:begin //Ori

RegWrite <= 1'b1;

Branch <= 2'b0;

MemWrite <= 1'b0;

MemRead <= 1'b0;

MemToReg <= 1'b0;

RegDst <= 1'b0;

ALUSrc <= 1'b1;

MemWrite <= 1'b0;

ALUCntl <= 4'b0001;

end

6'h23:begin //lw

Branch <= 2'b0;

RegDst <= 1'b0;

RegWrite <= 1'b1;

ALUSrc <= 1'b1;

ALUCntl <= 4'b1010;

MemWrite <= 1'b0;

MemRead <= 1'b1;

MemToReg <= 1'b1;

end

6'h2B:begin //sw

Branch <= 2'b0;

RegDst <= 1'b0;

RegWrite <= 1'b0;

ALUSrc <= 1'b1;

ALUCntl <= 4'b1010;

MemWrite <= 1'b1;

MemRead <= 1'b0;

MemToReg <= 1'b0;

end

6'h04:begin //beq

RegDst <= 1'b0;

Branch <= 2'b01;

RegWrite <= 1'b0;

ALUCntl <= 4'b1110;

ALUSrc <= 1'b0;

MemWrite <= 1'b0;

MemRead <= 1'b0;

MemToReg <= 1'b0;

end

6'h05:begin //bne

RegDst <= 1'b0;

Branch <= 2'b10;

RegWrite <= 1'b0;

ALUCntl <= 4'b1110;

ALUSrc <= 1'b0;

MemWrite <= 1'b0;

MemRead <= 1'b0;

MemToReg <= 1'b0;

end

6'h0A:begin //slti

Branch <= 2'b0;

RegWrite <= 1'b1;

MemWrite <= 1'b0;

MemRead <= 1'b0;

MemToReg <= 1'b0;

RegDst <= 1'b0;

ALUCntl <= 4'b1111;

ALUSrc <= 1'b1;

end

6'h0B:begin //sltiu

Branch <= 2'b0;

RegWrite <= 1'b1;

MemWrite <= 1'b0;

MemRead <= 1'b0;

MemToReg <= 1'b0;

RegDst <= 1'b0;

ALUCntl <= 4'b0100;

ALUSrc <= 1'b1;

end

default:begin

//$display("Error! check control file");

end

endcase

end

endmodule

module ALU(

input [31:0] A,

input [31:0] B,

input [3:0] ALUCntl,

output reg [31:0] ALUout,

output reg C,

output reg N,

output Z,

output reg V

);

reg signed [31:0]A\_s,B\_s;

assign Z = (ALUout == 32'b0)?1'b1:1'b0;

always@(\*)begin

A\_s = A;

B\_s = B;

case(ALUCntl)

4'b1010:begin //add

{C,ALUout} = A\_s + B\_s;

if((A\_s[31] & B\_s[31] & ~ALUout[31])||(~A\_s[31] & ~B\_s[31] & ALUout[31]))//If a is pos,b is pos and result is neg, Or vice visa

V = 1'b1;

else

V = 1'b0;

N = ALUout[31];

end

4'b1110:begin //Sub

{C,ALUout} = A\_s - B\_s;

if((A\_s[31] & ~B\_s[31] & ~ALUout[31])||(~A\_s[31] & B\_s[31] & ALUout[31]))

V = 1'b1;

else

V = 1'b0;

N = ALUout[31];

end

4'b0010:begin //add Unsigned

{C,ALUout} = A + B;

V = C;

N = 1'b0;

end

4'b0110:begin //Sub Unsigned

{C,ALUout} = A - B;

V = C;

N = 1'b0;

end

4'b0000:begin //And

{V,C,ALUout} = {2'bx,(A&B)};

N = ALUout[31];

end

4'b0001:begin //Or

{V,C,ALUout} = {2'bx,(A|B)};

N = ALUout[31];

end

4'b0011:begin //XOr

{V,C,ALUout} = {2'bx,(A^B)};

N = ALUout[31];

end

4'b1100:begin //Nor

{V,C,ALUout} = {2'bx,~(A|B)};

N = ALUout[31];

end

4'b0111:begin //Not

{V,C,ALUout} = {2'bx,~(A)};

N = ALUout[31];

end

4'b1101:begin //Not

{C,ALUout} = {(A)<<1};

V = 1'bx;

N = ALUout[31];

end

4'b1111:begin //SLT

C = 0;

V = 0;

N = 0;

if(A[31] & ~B[31])

ALUout = 0;

else if(B[31] & ~A[31])

ALUout = 32'hFFFFFFFF;

else if(~B[31] & ~A[31])

begin

if(A > B)

ALUout = 32'hFFFFFFFF;

else

ALUout = 0;

end

else

begin

if(A < B)

ALUout = 32'hFFFFFFFF;

else

ALUout = 0;

end

end

4'b0100: begin //SLTU

C = 0;

V = 0;

N = 0;

if(A > B)

ALUout = 32'hFFFFFFFF;

else

ALUout = 0;

end

default: begin

C = 0;

N = 0;

V = 0;

ALUout = 0;

//$display("Error in ALU!");

end

endcase

end

endmodule

module DataMem(

input clk,

input mem\_wr,

input mem\_rd,

input [31:0] addr,

input [31:0] wr\_data,

output [31:0] rd\_data

);

reg [7:0] dmem [0:4095];

// write

always@(posedge clk) begin

if(mem\_wr) begin

dmem[addr[11:0] + 2'd3] <= wr\_data[7:0];

dmem[addr[11:0] + 2'd2] <= wr\_data[15:8];

dmem[addr[11:0] + 2'd1] <= wr\_data[23:16];

dmem[addr[11:0] + 2'd0] <= wr\_data[31:24];

end

end

//read

assign rd\_data = (mem\_rd) ? { dmem[addr[11:0] + 2'd0],

dmem[addr[11:0] + 2'd1],

dmem[addr[11:0] + 2'd2],

dmem[addr[11:0] + 2'd3] }

: 32'hz;

Endmodule

module Mux32x1(

input [31:0] zero,

input [31:0] one,

input ctrl,

output reg [31:0] out

);

always @(zero or one or ctrl) begin

case(ctrl)

1'b0: out <= zero;

1'b1: out <= one;

//default: $display("Error in Mux32x1");

endcase

end

endmodule

Conclusion:

We were successfully able to execute the program. At first, we were having trouble getting the module to successfully modify the data memory, but we were able to fix this in succeeding iterations of the program. During this process, using the waveform diagram by adding waves to it from the program was an extremely useful method of debugging modules by checking the inputs and outputs, and in the future, we plan on using this more extensively.