Ethan Fan

CMPEN331 Final Project

12/9/2023

**Abstract Report**

In this Vivado project, I created two forwarding multiplexers for our rs and rt inputs, as well as modified our control unit to support these multiplexers. This is so we can accelerate our datapath with pipelining and forwarding instead of stalling. This datapath consisted of five different stages, Instruction Fetch (IF), Instruction Decode (ID), ALU Computation (EXE), Memory Manage (MEM), and Write Back (WB).

Normally, we need to wait until the instruction is finished executing so we can grab the appropriate data at the written register. But with forwarding added, we can grab the information as soon as its ready, either at the EXE stage, MEM stage, or WB stage. This is the main feature that I implemented in this project.

**Introduction**

The 5 stages of our datapath have already been implemented from previous labs. Here is a brief overview as to how each stage is connected:

1. **Instruction Fetch**
   1. *Modules*: PC Adder, Program Counter, Instruction Memory, IF/ID Pipeline
   2. *Analysis*: The Program Counter has a certain value, which represents the address of a certain instruction inside the Instruction Memory. The Instruction Memory reads this value and outputs the 32-bit instruction to the IF/ID Pipeline. The PC Adder then increments the Program Counter by 4 to fetch the next instruction in line.
2. **Instruction Decode**
   1. *Modules:* IF/ID Pipeline, Control Unit, Register Mux, Forwarding Mux A, Forwarding Mux B, Immediate Sign Extender, Register File, ID/EXE Pipeline.
   2. *Analysis:* The IF/ID Pipeline returns the 32-bit instruction. This instruction is then broken down into a number of parts, depending on the type of instruction it is.
      1. The opcode, func, rs, and rt parts are fed into the Control Unit.
      2. The rd, and rt parts are fed into the Register Mux.
      3. The immediate is fed into the Sign Extender.
      4. The rs and rt parts are fed into the Register File.
   3. The Control Unit depends on eight inputs to determine which values the selector bits for the forwarding multiplexers will take. It will check whether rs or rt equals a certain register value at either the EXE stage, MEM stage, or WB stage (this is just one of many conditionals). The Control Unit also determines the value the Register Mux will output either rd or rt.
      1. The Control Unit also outputs many bits whose values are dependent on whatever opcode and func are. These outputs are called signal bits.
   4. The Immediate Sign Extender adds 16 extra bits at the start of the immediate input, which is initially 16 bits.
   5. All of these modules (except IF/ID Pipeline) will have outputs which will be fed into the ID/EXE Pipeline.
3. **ALU Computation**
   1. *Modules:* ID/EXE Pipeline, ALU Mux, ALU, EXE/MEM Pipeline
   2. *Analysis:* The output of the ID/EXE Pipeline consists of the signal bits which are passed into the EXE/MEM pipeline while some are also passed back to the control unit to determine which values the forwarding mux selector bits will be.
   3. Our main focus is the ALU module. Depending on the aluc 4 bit input, it will do a computation with the other 2 inputs and output the answer. This answer will also be forwarded back to the forwarding muxs, which could potentially be used again.
4. **Memory Manage**
   1. *Modules:* EXE/MEM Pipeline, Data Memory, MEM/WB Pipeline
   2. *Analysis:* Signal bits from the EXE/MEM Pipeline are fed to the MEM/WB Pipeline. The Data Memory module takes care of either storing outputs of the ALU into itself, or grabbing information from itself to use as an input to the forwarding muxs.
5. **Write Back**
   1. *Modules:* MEM/WB Pipeline, Write Back Mux, Register File.
   2. *Analysis:* The wDestReg is written into the Register File. The output do from Data Memory is inputted into the Pipeline, which is then outputted and inputted into d of the Reg file. The qa and qb outputs are finally redirected back to the forwarding muxs, which can be used again.

The benefits of using this architecture of the datapath is that it is capable of speeding up the workload by redirecting data in a smart way instead of just stalling. The usage of this datapath allows us to asynchronously run multiple instructions without worrying too much about stalling instructions and killing time (similar to the laundry analogy taught in class).

**Design Code**

module datapath(input wire clock,

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,

output wire mwreg,

output wire mm2reg,

output wire mwmem,

output wire [4:0] mdestReg,

output wire [31:0] mr,

output wire [31:0] mqb,

output wire wwreg,

output wire wm2reg,

output wire [4:0] wdestReg,

output wire [31:0] wr,

output wire [31:0] wdo

);

wire [31:0] nextPc;

pc pCounter(.nextPc(nextPc), .clock(clock), .pc(pc));

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

wire [31:0] instOut;

inst\_memory instMemory(.pc(pc), .instOut(instOut));

ifid\_pipeline\_reg ifidPipelineReg(.instOut(instOut), .clock(clock), .dinstOut(dinstOut));

wire [5:0] op = dinstOut[31:26];

wire [4:0] rs = dinstOut[25:21];

wire [4:0] rt = dinstOut[20:16];

wire [4:0] rd = dinstOut[15:11];

wire [5:0] func = dinstOut[5:0];

wire wreg;

wire m2reg;

wire wmem;

wire [3:0] aluc;

wire aluimm;

wire regrt;

wire [4:0] destReg;

wire [31:0] qa;

wire [31:0] qb;

wire [31:0] wbData;

wire [15:0] imm = dinstOut[15:0];

wire [31:0] imm32;

wire [31:0] r;

wire [31:0] mdo;

wire [1:0] fwda;

wire [1:0] fwdb;

wire [31:0] fwd\_qa;

wire [31:0] fwd\_qb;

control\_unit controlUnit(.op(op),

.func(func),

.rs(rs),

.rt(rt),

.mdestReg(mdestReg),

.mm2reg(mm2reg),

.mwreg(mwreg),

.edestReg(edestReg),

.em2reg(em2reg),

.ewreg(ewreg),

.fwda(fwda),

.fwdb(fwdb),

.wreg(wreg),

.m2reg(m2reg),

.wmem(wmem),

.aluc(aluc),

.aluimm(aluimm),

.regrt(regrt)

);

reg\_multiplex regMultiplex(.rt(rt), .rd(rd), .regrt(regrt), .destReg(destReg));

forward\_a\_mux fwdA(.fwda(fwda), .qa(qa), .r(r), .mr(mr), .mdo(mdo), .fwd\_qa(fwd\_qa));

forward\_b\_mux fwdB(.fwdb(fwdb), .qb(qb), .r(r), .mr(mr), .mdo(mdo), .fwd\_qb(fwd\_qb));

reg\_file regFile(.rs(rs), .rt(rt), .wdestReg(wdestReg), .wbData(wbData), .wwreg(wwreg), .clock(clock), .qa(qa), .qb(qb));

immediate\_extender immediateExtender(.imm(imm), .imm32(imm32));

id\_exe\_pipeline\_reg idExePipelineReg(

.wreg(wreg),

.m2reg(m2reg),

.wmem(wmem),

.aluc(aluc),

.aluimm(aluimm),

.destReg(destReg),

.qa(fwd\_qa), //

.qb(fwd\_qb), //

.imm32(imm32),

.clock(clock),

.ewreg(ewreg),

.em2reg(em2reg),

.ewmem(ewmem),

.ealuc(ealuc),

.ealuimm(ealuimm),

.edestReg(edestReg),

.eqa(eqa),

.eqb(eqb),

.eimm32(eimm32)

);

wire [31:0] b;

alu\_multiplex aluMultiplex(.eqb(eqb), .eimm32(eimm32), .ealuimm(ealuimm), .b(b));

alu ALU(.eqa(eqa), .b(b), .ealuc(ealuc), .r(r));

exemem\_pipeline\_reg exeMemPipelineReg(

.ewreg(ewreg),

.em2reg(em2reg),

.ewmem(ewmem),

.edestReg(edestReg),

.r(r),

.eqb(eqb),

.clock(clock),

.mwreg(mwreg),

.mm2reg(mm2reg),

.mwmem(mwmem),

.mdestReg(mdestReg),

.mr(mr),

.mqb(mqb)

);

data\_memory dataMemory(

.mr(mr),

.mqb(mqb),

.mwmem(mwmem),

.clock(clock),

.mdo(mdo)

);

memwb\_pipeline\_reg memWbPipelineReg(

.mwreg(mwreg),

.mm2reg(mm2reg),

.mdestReg(mdestReg),

.mr(mr),

.mdo(mdo),

.clock(clock),

.wwreg(wwreg),

.wm2reg(wm2reg),

.wdestReg(wdestReg),

.wr(wr),

.wdo(wdo)

);

wb\_mux wbMux(

.wr(wr), .wdo(wdo), .wm2reg(wm2reg), .wbData(wbData)

);

endmodule

module pc(input wire [31:0] nextPc, input wire clock, output reg [31:0] pc);

initial begin

pc = 32'd100;

end

always @ (posedge clock) begin

pc <= nextPc;

end

endmodule

module pc\_adder(input wire [31:0] pc, output reg [31:0] nextPc);

always @(\*) begin

nextPc = pc + 32'd4;

end

endmodule

module inst\_memory(input wire [31:0] pc, output reg [31:0] instOut);

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

initial begin

memory[25] = {6'b000000, //add $3, $1, $2

5'b00001,

5'b00010,

5'b00011,

5'b00000,

6'b100000

};

memory[26] = {6'b000000, //sub $4, $9, $3

5'b01001,

5'b00011,

5'b00100,

5'b00000,

6'b100010

};

memory[27] = {6'b000000, //or $5, $3, $9

5'b00011,

5'b01001,

5'b00101,

5'b00000,

6'b100101

};

memory[28] = {6'b000000, //xor $6, $3, $9

5'b00011,

5'b01001,

5'b00110,

5'b00000,

6'b100110

};

memory[29] = {6'b000000, //and $7, $3, $9

5'b00011,

5'b01001,

5'b00111,

5'b00000,

6'b100100

};

end

always @(\*) begin

instOut <= memory[pc[7:2]]; //divide pc by 4 to access memory 25 and 26

end

endmodule

module ifid\_pipeline\_reg(input wire [31:0] instOut, input wire clock, output reg [31:0] dinstOut);

always @ (posedge clock) begin

dinstOut <= instOut;

end

endmodule

module control\_unit(

input wire [5:0] op,

input wire [5:0] func,

//project update

input wire [4:0] rs,

input wire [4:0] rt,

input wire [4:0] mdestReg,

input wire mm2reg,

input wire mwreg,

input wire [4:0] edestReg,

input wire em2reg,

input wire ewreg,

output reg [1:0] fwda, //rs

output reg [1:0] fwdb, //rt

//output reg wpcir, //enable write to PC

//

output reg wreg,

output reg m2reg,

output reg wmem,

output reg [3:0] aluc,

output reg aluimm,

output reg regrt

);

always @(\*) begin

case (op)

6'b000000: //memory[29]: r-type instructions

begin

case(func)

6'b100000: //add r-type

begin

aluc <= 4'b0010;

end

6'b100010: //sub

begin

aluc <= 4'b0110;

end

6'b100101: //or

begin

aluc <= 4'b0001;

end

6'b100110: //xor

begin

aluc <= 4'b1100;

end

6'b100100: //and

begin

aluc <= 4'b0000;

end

endcase

wreg <= 1'b1;

m2reg <= 1'b0;

wmem <= 1'b0;

aluimm <= 1'b0;

regrt <= 1'b0;

end

6'b100011: //load word

begin

wreg <= 1'b1;

m2reg <= 1'b1;

wmem <= 1'b0;

aluc <= 4'b0010;

aluimm <= 1'b1;

regrt <= 1'b1;

end

endcase

//forwarding updates

if (ewreg == 1'b1 && em2reg == 1'b0 && edestReg != 5'b00000 && edestReg == rs) begin

fwda <= 2'b01;

end

else if (mwreg == 1'b1 && mm2reg == 1'b0 && mdestReg != 5'b00000 && mdestReg == rs) begin

fwda <= 2'b10;

end

else if (mwreg == 1'b1 && mm2reg == 1'b1 && mdestReg != 5'b00000 && mdestReg == rs) begin

fwda <= 2'b11;

end

else begin

fwda <= 2'b00;

end

if (ewreg == 1'b1 && em2reg == 1'b0 && edestReg != 5'b00000 && edestReg == rt) begin

fwdb <= 2'b01;

end

else if (mwreg == 1'b1 && mm2reg == 1'b0 && mdestReg != 5'b00000 && mdestReg == rt) begin

fwdb <= 2'b10;

end

else if (mwreg == 1'b1 && mm2reg == 1'b1 && mdestReg != 5'b00000 && mdestReg == rt) begin

fwdb <= 2'b11;

end

else begin

fwdb <= 2'b00;

end

end

endmodule

module reg\_multiplex(

input wire [4:0] rt,

input wire [4:0] rd,

input wire regrt,

output reg [4:0] destReg

);

always @(\*) begin

if (regrt == 1'b0) begin

destReg <= rd;

end

else begin

destReg <= rt;

end

end

endmodule

module forward\_a\_mux(

input wire [1:0] fwda, //control\_unit OUTPUT

input wire [31:0] qa, //regfile OUTPUT

input wire [31:0] r, //alu OUTPUT

input wire [31:0] mr, //exemem\_pipeline OUTPUT

input wire [31:0] mdo, //datamemory OUTPUT

output reg [31:0] fwd\_qa //goes into idexePipeLineReg: replaces its qa

);

always @(\*) begin

case(fwda)

2'b00:

begin

fwd\_qa <= qa;

end

2'b01:

begin

fwd\_qa <= r;

end

2'b10:

begin

fwd\_qa <= mr;

end

2'b11:

begin

fwd\_qa <= mdo;

end

endcase

end

endmodule

module forward\_b\_mux(

input wire [1:0] fwdb, //control\_unit OUTPUT

input wire [31:0] qb, //regfile OUTPUT

input wire [31:0] r, //alu OUTPUT

input wire [31:0] mr, //exemem\_pipeline OUTPUT

input wire [31:0] mdo, //datamemory OUTPUT

output reg [31:0] fwd\_qb //goes into idexePipeLineReg: replaces its qa

);

always @(\*) begin

case(fwdb)

2'b00:

begin

fwd\_qb <= qb;

end

2'b01:

begin

fwd\_qb <= r;

end

2'b10:

begin

fwd\_qb <= mr;

end

2'b11:

begin

fwd\_qb <= mdo;

end

endcase

end

endmodule

module reg\_file(

input wire [4:0] rs,

input wire [4:0] rt,

input wire [4:0] wdestReg,

input wire [31:0] wbData,

input wire wwreg,

input wire clock,

output reg [31:0] qa,

output reg [31:0] qb

);

reg [31:0] register [0:31];

integer i;

initial begin

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

register[i] = 0;

end

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

always @(\*) begin

qa <= register[rs];

qb <= register[rt];

end

always @(negedge clock) begin

if (wwreg == 1'b1) begin

register[wdestReg] <= wbData;

end

end

endmodule

module immediate\_extender(input wire [15:0] imm, output reg [31:0] imm32);

always @(\*) begin

imm32 = {{16{imm[15]}},imm};

end

endmodule

module id\_exe\_pipeline\_reg(

input wire wreg,

input wire m2reg,

input wire wmem,

input wire [3:0] aluc,

input wire aluimm,

input wire [4:0] destReg,

input wire [31:0] qa,

input wire [31:0] qb,

input wire [31:0] imm32,

input wire clock,

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

);

always @(posedge clock) begin

ewreg <= wreg;

em2reg <= m2reg;

ewmem <= wmem;

ealuc <= aluc;

ealuimm <= aluimm;

edestReg <= destReg;

eqa <= qa;

eqb <= qb;

eimm32 <= imm32;

end

endmodule

module alu\_multiplex(

input wire [31:0] eqb,

input wire [31:0] eimm32,

input wire ealuimm,

output reg [31:0] b

);

always @(\*) begin

if (ealuimm == 1'b0) begin

b <= eqb;

end

else begin

b <= eimm32;

end

end

endmodule

module alu(

input wire [31:0] eqa,

input wire [31:0] b,

input wire [3:0] ealuc,

output reg [31:0] r

);

always @(\*) begin

case(ealuc)

4'b0010: begin

r <= eqa + b;

end

4'b0110: begin

r <= eqa - b;

end

4'b0001: begin

r <= eqa | b;

end

4'b1100: begin

r <= eqa ^ b;

end

4'b0000: begin

r <= eqa & b;

end

endcase

end

endmodule

module exemem\_pipeline\_reg(

input wire ewreg,

input wire em2reg,

input wire ewmem,

input wire [4:0] edestReg,

input wire [31:0] r,

input wire [31:0] eqb,

input wire clock,

output reg mwreg,

output reg mm2reg,

output reg mwmem,

output reg [4:0] mdestReg,

output reg [31:0] mr,

output reg [31:0] mqb

);

always @(posedge clock) begin

mwreg <= ewreg;

mm2reg <= em2reg;

mwmem <= ewmem;

mdestReg <= edestReg;

mr <= r;

mqb <= eqb;

end

endmodule

module data\_memory(

input wire [31:0] mr,

input wire [31:0] mqb,

input wire mwmem,

input wire clock,

output reg [31:0] mdo

);

reg [31:0] data\_memory [0:63];

initial begin

data\_memory[0] = 32'hA00000AA;

data\_memory[1] = 32'h10000011;

data\_memory[2] = 32'h20000022;

data\_memory[3] = 32'h30000033;

data\_memory[4] = 32'h40000044;

data\_memory[5] = 32'h50000055;

data\_memory[6] = 32'h60000066;

data\_memory[7] = 32'h70000077;

data\_memory[8] = 32'h80000088;

data\_memory[9] = 32'h90000099;

end

always @(\*) begin

mdo <= data\_memory[mr[31:2]];

end

always @(negedge clock) begin

if (mwmem == 1'b1) begin

data\_memory[mr[31:2]] <= mqb;

end

end

endmodule

module memwb\_pipeline\_reg(

input wire mwreg,

input wire mm2reg,

input wire [4:0] mdestReg,

input wire [31:0] mr,

input wire [31:0] mdo,

input wire clock,

output reg wwreg,

output reg wm2reg,

output reg [4:0] wdestReg,

output reg [31:0] wr,

output reg [31:0] wdo

);

always @(posedge clock) begin

wwreg <= mwreg;

wm2reg <= mm2reg;

wdestReg <= mdestReg;

wr <= mr;

wdo <= mdo;

end

endmodule

module wb\_mux(input wire [31:0] wr, input wire [31:0] wdo, input wire wm2reg, output reg [31:0] wbData);

always @(\*) begin

if (wm2reg == 1'b0) begin

wbData <= wr;

end

else if (wm2reg == 1'b1) begin

wbData <= wdo;

end

end

endmodule

**Testbench Code**

`timescale 1ns / 1ps

module testbench();

reg clock;

initial clock = 1'b0;

always begin

#1 clock = ~clock;

end

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;

wire mwreg;

wire mm2reg;

wire mwmem;

wire [4:0] mdestReg;

wire [31:0] mr;

wire [31:0] mqb;

wire wwreg;

wire wm2reg;

wire [4:0] wdestReg;

wire [31:0] wr;

wire [31:0] wdo;

datapath dataPath(

.clock(clock),

.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)

);

Endmodule

**Waveforms**

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

**Design Schematics**

A green lines on a white background

Description automatically generated

**I/O Planning**

A screenshot of a game

Description automatically generated

**Floor Planning**

A screenshot of a computer

Description automatically generated