

Project

Tyler Korz - CMPEN 331 Sec 002



1. Abstract:

This project explores the comprehensive design and implementation of a computer system using Verilog within the Vivado environment. The primary components, including the data path, control path, memory system, and I/O system, are individually crafted and meticulously integrated to form a cohesive architecture capable of executing programs.

The data path, responsible for data manipulation and computation, is designed to efficiently process instructions in conjunction with the control path. The control path dictates the flow of operations and ensures synchronized execution, orchestrated by the fetch and execute cycles. The memory system, a critical element, is tailored to efficiently store and retrieve data, while the I/O system facilitates communication with external devices.

The connection between instruction set design, addressing modes, and fetch-and-execute operations is thoroughly examined. The instruction set design shapes the capabilities of the processor, dictating the types of operations it can perform. Addressing modes, on the other hand, determine how operands are accessed and manipulated. Fetch and execute operations are tightly coordinated, with the control path orchestrating the retrieval of instructions and their subsequent execution.

The design process involves careful consideration of the relationships between these components, ensuring optimal performance and adherence to the desired architecture. The impact of instruction set design and addressing modes on the overall efficiency and versatility of the system is explored. Additionally, the seamless interaction between the data path, control path, memory system, and I/O system is pivotal in achieving a functional and reliable computer system.

The project showcases how Verilog, implemented on the Vivado platform, serves as a powerful tool for the creation of each major component. Through a systematic and iterative design process, the resulting computer system demonstrates the successful integration of diverse elements, offering insights into the complexities of modern processor architecture and its ability to execute programs with precision and efficiency.

1. Code:

**//MODULE**

`timescale 1ns / 1ps

module PC(

input [31:0] nextPc,

input clk,

output reg [31:0] pc

);

initial pc = 32'd100;

always @(posedge clk)

begin

pc = nextPc;

end

endmodule

module InstructionMemory(

input [31:0] pc,

output reg [31:0] instOut

);

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

initial begin

memory[25] = {6'b100011, 5'b00001, 5'b00010, 5'b00000, 5'b00000, 6'b000000};

memory[26] = {6'b100011, 5'b00001, 5'b00011, 5'b00000, 5'b00000, 6'b000100};

memory[27] = {6'b100011, 5'b00001, 5'b00100, 5'b00000, 5'b00000, 6'b001000};

memory[28] = {6'b100011, 5'b00001, 5'b00101, 5'b00000, 5'b00000, 6'b001100};

memory[29] = {6'b000000, 5'b00010, 5'b01010, 5'b00110, 5'b00000, 6'b100000};

end

always @(\*)

instOut = memory[pc[7:2]];

endmodule

module PCAdder(

input [31:0] pc,

output reg [31:0] nextPc

);

always @(\*)

nextPc <= pc + 4;

endmodule

module IFIDPipelineReg(

input [31:0]instOut,

input clk,

output reg [31:0] dinstOut

);

always @(posedge clk)

dinstOut <= instOut;

endmodule

module ControlUnit(

input [5:0] op, func,

input [4:0] rs, rt, mdestReg, edestReg,

input mm2reg, mwreg, em2reg, ewreg,

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

output reg [3:0] aluc,

output reg [1:0] fwa, fwb,

output reg [4:0] ors, ort

);

always @(\*)

begin

ors = rs;

ort = rt;

case(op)

'b000000:

begin

wreg = 'b1;

m2reg = 'b0;

wmem = 'b0;

aluimm = 'b0;

regrt = 'b0;

case(func)

'b100000: //+

aluc = 'b0010;

'b100010: //-

aluc = 'b0110;

'b100100: //&

aluc = 'b1000;

'b100101: // |

aluc = 'b1001;

'b100110: //xor

aluc = 'b1010;

endcase

end

'b100011: //Load Word

begin

wreg = 'b1;

m2reg = 'b1;

wmem = 'b0;

aluc = 'b0010;

aluimm = 'b1;

regrt = 'b1;

end

endcase

fwa = 'b00;

fwb = 'b00;

if (edestReg == rs)

fwa = 'b01;

if (edestReg == rt)

fwb = 'b01;

if (mdestReg == rs)

begin

if (mm2reg == 1)

fwa = 'b11;

else

fwa = 'b10;

end

if (mdestReg == rt)

begin

if (mm2reg == 0)

fwb = 'b11;

else

fwb = 'b10;

end

end

endmodule

module RegrtMux(

input [4:0] rt, rd,

input regrt,

output reg [4:0] destReg

);

always @(\*)

begin

if (regrt == 0)

destReg = rd;

else

destReg = rt;

end

endmodule

module RegFile(

input clk, wwreg,

input [4:0] rs, rt,

input [4:0] wdestReg,

input [31:0] wbData,

output reg [31:0] qa, qb

);

reg [31:0] registers[0:31];

integer i;

initial begin

registers[0] = 32'h00000000;

registers[1] = 32'ha00000aa;

registers[2] = 32'h10000011;

registers[3] = 32'h20000022;

registers[4] = 32'h30000033;

registers[5] = 32'h40000044;

registers[6] = 32'h50000055;

registers[7] = 32'h60000066;

registers[8] = 32'h70000077;

registers[9] = 32'h80000088;

registers[10] = 32'h90000099;

for (i = 11; i < 32; i = i+1)

registers[i] = 32'b0;

end

always @(\*)

begin

qa = registers[rs];

qb = registers[rt];

end

always @(negedge clk)

begin

if (wwreg == 1)

registers[wdestReg] = wbData;

end

endmodule

module ImmExtender(

input [15:0] imm,

output reg [31:0] imm32

);

always @(\*)

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

endmodule

module IDEXEPipelineReg(

input clk, wreg, m2reg, wmem, aluimm,

input [3:0] aluc,

input [4:0] destReg,

input [31:0] qa, qb, imm32,

output reg ewreg, em2reg, ewmem, ealuimm,

output reg [3:0] ealuc,

output reg [4:0] edestReg,

output reg [31:0] eqa, eqb, eimm32

);

always @(posedge clk)

begin

ewreg = wreg;

em2reg = m2reg;

ewmem = wmem;

ealuc = aluc;

ealuimm = aluimm;

edestReg = destReg;

eqa = qa;

eqb = qb;

eimm32 = imm32;

end

endmodule

module ALUMux(

input [31:0] eqb, eimm32,

input ealuimm,

output reg [31:0] b

);

always @(\*)

begin

if (ealuimm == 0)

b <= eqb;

else

b <= eimm32;

end

endmodule

module ALU(

input [31:0] eqa, b,

input [3:0] ealuc,

output reg [31:0] r

);

always @(\*)begin

case(ealuc)

4'b0000:

begin

//AND operation

r = eqa & b;

end

4'b0001:

begin

//OR operatiion

r = eqa | b;

end

4'b0010:

begin

//add operation

r = eqa + b;

end

4'b0110:

begin

//subtract operation

r = eqa - b;

end

4'b1100:

begin

//NOR operation

r = ~(eqa | b);

end

4'b1111:

begin

//XOR operation

r = eqa ^ b;

end

endcase

end

endmodule

module EXEMEMPipelineReg(

input clk, ewreg, em2reg, ewmem,

input [4:0] edestReg,

input [31:0] r, eqb,

output reg mwreg, mm2reg, mwmem,

output reg [4:0] mdestReg,

output reg [31:0] mr, mqb

);

always @(posedge clk)

begin

mwreg <= ewreg;

mm2reg <= em2reg;

mwmem <= ewmem;

mdestReg <= edestReg;

mr <= r;

mqb <= eqb;

end

endmodule

module DataMemory(

input clk, mwmem,

input [31:0] mr, mqb,

output reg [31:0] mdo

);

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

initial begin

memory[0] = 'hA00000AA;

memory[1] = 'h10000011;

memory[2] = 'h20000022;

memory[3] = 'h30000033;

memory[4] = 'h40000044;

memory[5] = 'h50000055;

memory[6] = 'h60000066;

memory[7] = 'h70000077;

memory[8] = 'h80000088;

memory[9] = 'h90000099;

end

//Read Memory

always @(\*)

mdo <= memory[mr[7:2]];

//Write Memory

always @(negedge clk)

begin

if (mwmem == 1)

begin

memory[mr[7:2]] <= mqb;

end

end

endmodule

module MEMWBPipelineReg (

input clk, mwreg, mm2reg,

input [4:0] mdestReg,

input [31:0] mr, mdo,

output reg wwreg, wm2reg,

output reg [4:0] wdestReg,

output reg [31:0] wr, wdo

);

always @(posedge clk)

begin

wwreg <= mwreg;

wm2reg <= mm2reg;

wdestReg <= mdestReg;

wr <= mr;

wdo <= mdo;

end

endmodule

module WbMux (

input [31:0] wr, wdo,

input wm2reg,

output reg [31:0] wbData

);

always @(\*)

begin

if (wm2reg == 1)

wbData = wdo;

else

wbData = wr;

end

endmodule

module FwMux (

input [31:0] q, r, mr, mdo,

input [1:0] fwd,

output reg [31:0] fq

);

always @(\*)

begin

case(fwd)

2'b00:

fq = q;

2'b01:

fq = r;

2'b10:

fq = mr;

2'b11:

fq = mdo;

endcase

end

endmodule

**//Datapath**

`timescale 1ns / 1ps

module DataPath(

input clk,

output wire [31:0] pc, instOut, dinstOut,

output wire ewreg, em2reg, ewmem, ealuimm,

output wire [3:0] ealuc,

output wire [4:0] edestReg,

output wire [31:0] eqa, eqb, eimm32,

output wire mwreg, mm2reg, mwmem,

output wire [4:0] mdestReg,

output wire [31:0] mr, mqb,

output wire [4:0] ors, ort,

output wire wwreg, wm2reg,

output wire [4:0] wdestReg,

output wire [31:0] wr, wdo, wbData,

output wire [1:0] fwa, fwb

);

wire [31:0] nextPc;

PC pc\_dp(nextPc, clk, pc);

PCAdder pcadder\_dp(pc, nextPc);

InstructionMemory im\_dp(pc, instOut);

IFIDPipelineReg ifidreg\_dp(instOut, clk, dinstOut);

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

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

wire wreg, m2reg, wmem, aluimm, regrt;

wire [3:0] aluc;

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

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

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

ControlUnit cu\_dp(op, func, rs, rt, mdestReg, edestReg, mm2reg, mwreg, em2reg, ewreg, wreg, m2reg, wmem, aluimm, regrt, aluc, fwa, fwb, ors, ort);

wire [31:0] qa;

wire [31:0] qb;

FwMux fwmuxa\_dp(fqa, r, mr, mdo, fwa, qa);

FwMux fwmuxb\_dp(fqb, r, mr, mdo, fwb, qb);

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

wire [31:0] imm32;

ImmExtender immex\_dp(imm, imm32);

IDEXEPipelineReg idexereg\_dp(clk, wreg, m2reg, wmem, aluimm, aluc, destReg, qa, qb, imm32, ewreg, em2reg, ewmem, ealuimm, ealuc, edestReg, eqa, eqb, eimm32);

wire [31:0] b;

ALUMux alumux\_dp(eqb, eimm32, ealuimm, b);

wire [31:0] r;

ALU alu\_dp(eqa, b, ealuc, r);

EXEMEMPipelineReg exememreg\_dp(clk, ewreg, em2reg, ewmem, edestReg, r, eqb, mwreg, mm2reg, mwmem, mdestReg, mr, mqb);

wire [31:0] mdo;

DataMemory datamem\_dp(clk, mwmem, mr, mqb, mdo);

MEMWBPipelineReg memwbreb\_dp(clk, mwreg, mm2reg, mdestReg, mr, mdo, wwreg, wm2reg, wdestReg, wr, wdo);

WbMux wbmux\_dp(wr, wdo, wm2reg, wbData);

wire [4:0] destReg;

RegrtMux regrtmux\_dp(rt, rd, regrt, destReg);

wire [31:0] fqa;

wire [31:0] fqb;

RegFile rf\_dp(clk, wwreg, rs, rt, wdestReg, wbData, fqa, fqb);

endmodule

**//TestBench**

`timescale 1ns / 1ps

module testbench();

reg clk\_tb;

initial clk\_tb = 1'b0;

wire [31:0] pc, instOut, dinstOut;

wire ewreg, em2reg, ewmem, ealuimm;

wire [3:0] ealuc;

wire [4:0] edestReg;

wire [31:0] eqa, eqb, eimm32;

wire mwreg, mm2reg, mwmem;

wire [4:0] mdestReg;

wire [31:0] mr, mqb;

wire [4:0] ors, ort;

wire wwreg, wm2reg;

wire [4:0] wdestReg;

wire [31:0] wr, wdo, wbData;

wire [1:0] fwa, fwb;

DataPath dp\_tb (

clk\_tb,

pc, instOut, dinstOut,

ewreg, em2reg, ewmem, ealuimm,

ealuc,

edestReg,

eqa, eqb, eimm32,

mwreg, mm2reg, mwmem,

mdestReg,

mr, mqb,

ors, ort,

wwreg, wm2reg,

wdestReg,

wr, wdo, wbData,

fwa, fwb

);

always

begin

#5

clk\_tb = ~clk\_tb;

end

endmodule

1. Pictures:

**Timing**

**A computer screen shot of a black screen

Description automatically generated**

**Schematic**

**A line with green lines

Description automatically generated with medium confidence**

**A diagram of a circuit board

Description automatically generated**

**I/O Planning**

**A screenshot of a video game

Description automatically generated**

**Floor Planning**

**A screen shot of a video game

Description automatically generated**