CMPEN 331

Final Project

Nandan Hirpara

**Abstract**  
  
The project presents the design and implementation of a simple 32-bit RISC processor using Verilog HDL. The processor follows the Harvard architecture and comprises essential components such as a program counter (PC), instruction memory, register file, arithmetic logic unit (ALU), control unit, and data memory. The processor supports basic R-type and load/store instructions.

The processor's pipeline consists of stages such as Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), and Write Back (WB). The IF stage fetches instructions from memory, the ID stage decodes instructions and generates control signals, the EX stage performs arithmetic or logic operations, the MEM stage handles data memory access, and the WB stage writes back results to registers.

Each module in the design, including the ALU, register file, and memory components, is modeled as a separate Verilog module to enhance modularity and readability. Control signals are generated by the Control Unit module based on the opcode and function code of the instructions.

This project code includes detailed modules for each functional block, such as the ALU, register file, memory components, and the main Datapath. A testbench is also provided to simulate and verify the functionality of the processor.

This architecture offers benefits in terms of modularity, simplicity, and ease of understanding. The pipeline structure enhances instruction throughput, enabling multiple instructions to be processed simultaneously. The project serves as a foundation for understanding processor architecture and can be extended for more complex instruction sets and features in future work.

**Introduction**

In the realm of computer architecture, the efficiency and speed of data processing within a Central Processing Unit (CPU) stand as pivotal factors. Our project focuses on the design and implementation of a CPU datapath, a crucial component responsible for executing instructions and managing data flow. The architecture employs a pipeline structure with distinct stages: Instruction Fetch (IF), Instruction Decode (ID), Execute (EXE), Memory (MEM), and Write Back (WB). Each stage plays a unique role in processing instructions, and their seamless coordination enhances the overall performance. The datapath integrates key components such as forwarding logic, stall management, and control units to ensure data integrity and mitigate hazards. Our design, implemented in Verilog, leverages a pipeline approach to optimize instruction execution, and the project report presents a comprehensive analysis of the design code, test bench, waveforms, Xilinx synthesis schematics, I/O planning, and floor planning. This architecture not only exemplifies effective pipelining but also showcases its applicability and benefits in the broader field of computer organization.  
  
The forwarding logic is as follows:

* **Forwarding Detection Logic (FWDA and FWDB):** a. FWDA and FWDB are strategically positioned to identify the necessity for forwarding. Detection is contingent upon the following conditions: i. Verification that the destination register to be written to (edestReg or mdestReg) is non-zero. ii. Comparison of these destination registers with the source registers (rs and rt) of the current instruction. iii. Determination of whether the forwarding data originates from the ALU, Memory, or Write Back stage.
* **Forwarding Multiplexer:** a. The FWDA and FWDB multiplexers dynamically operate based on signals from the forwarding logic. These multiplexers make decisions between the original operand data (qa or qb), the result from the Execution stage (r), the data from the Memory stage (mr), or the data from the Write Back stage (do), contingent upon the situation detected by the forwarding logic.

The stall logic, an integral component of the CPU, plays a crucial role in preserving data integrity within the pipeline, especially in handling read-after-write hazards. This logic is implemented within the Control Unit, which subsequently signals the PC counter and IFID register to stall when necessary.

**DEVICE**: - XC7K160TFBG676-3

**Desing code:**

`timescale 1ns / 1ps

module Pc\_adder(pc, nextPc);

input [31:0] pc;

output reg [31:0] nextPc;

always @(\*)

begin

nextPc = pc + 4;

end

endmodule

module PrgCounter(nextPc, clk, pc);

input [31:0] nextPc;

input clk;

output reg [31:0] pc;

initial begin

pc = 32'd100;

end

always @(posedge clk) begin

pc = nextPc;

end

endmodule

module InstructionMemory(pc, instOut);

input [31:0] pc;

output reg [31:0] instOut;

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

initial begin

memory[25] = 32'h00221820;

memory[26] = 32'h01232022;

memory[27] = 32'h00692825;

memory[28] = 32'h00693026;

memory[29] = 32'h00693824;

end

always @(\*) begin

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

end

endmodule

module IFIDPipelineRegister(instOut, clk, dinstOut);

input [31:0] instOut;

input clk;

output reg [31:0] dinstOut;

always @(posedge clk) begin

dinstOut = instOut;

end

endmodule

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

input [5:0] op;

input [5:0] func;

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

output reg [3:0] aluc;

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

input mm2reg, mwreg, em2reg, ewreg;

output reg [1:0] fwa, fwb;

output reg [4:0] ors, ort;

always @(\*) begin

case(op)

6'b000000: // r-types

begin

wreg = 1'b1;

m2reg = 1'b0;

wmem = 1'b0;

aluimm = 1'b0;

regrt = 1'b0;

case(func)

6'b100000: // ADD instruction

begin //setting value of controls signals for ADD instruction

aluc = 4'b0010;

end

6'b100010: // SUB instruction

begin //setting value of controls signals for SUB instruction

aluc = 4'b0110;

end

6'b100100: // AND instruction

begin //setting value of controls signals for AND instruction

aluc = 4'b1000;

end

6'b100101: // OR instruction

begin //setting value of controls signals for OR instruction

aluc = 4'b1001;

end

6'b100110: // XOR instruction

begin //setting value of controls signals for XOR instruction

aluc = 4'b1010;

end

endcase

end

6'b100011: //LW

begin

//setting value of controls signals for LW instruction

wreg = 1'b1;

m2reg = 1'b1;

wmem = 1'b0;

aluc = 4'b0010;

aluimm = 1'b1;

regrt = 1'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 RegRTMuliplexer(rt, rd, regrt, destReg);

input [4:0] rt;

input [4:0] rd;

input regrt;

output reg [4:0] destReg;

always @(\*) begin

if(regrt == 0) begin

destReg = rd;

end

else if(regrt == 1) begin

destReg = rt;

end

end

endmodule

module RegisterFile(rs, rt, qa, qb, wdestReg, wbData, clk, wwreg);

input [4:0] rs;

input [4:0] rt;

output reg [31:0] qa;

output reg [31:0] qb;

input [4:0] wdestReg;

input [31:0] wbData;

input clk, wwreg;

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) begin

registers[i] = 0;

end

end

always @(\*) begin

qa = registers[rs];

qb = registers[rt];

end

always @(negedge clk)

begin

if (wwreg == 1)

registers[wdestReg] = wbData;

end

endmodule

module ImmediateExtender(imm, imm32);

input [15:0] imm;

output reg [31:0] imm32;

always @(\*) begin

imm32 <= {{16{imm[15]}},{imm[15:0]}};

end

endmodule

module IDEXEPipelineRegister(wreg, m2reg, wmem, aluc, aluimm, destReg, qa, qb, imm32, clk,

ewreg, em2reg, ewmem, ealuc, ealuimm, edestReg, eqa, eqb, eimm32);

input wreg;

input m2reg;

input wmem;

input [3:0] aluc;

input aluimm;

input [4:0] destReg;

input [31:0] qa;

input [31:0] qb;

input [31:0] imm32;

input clk;

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 clk) begin

ewreg = wreg;

em2reg = m2reg;

ewmem = wmem;

ealuc = aluc;

ealuimm = aluimm;

edestReg = destReg;

eqa = qa;

eqb = qb;

eimm32 = imm32;

end

endmodule

module ALU\_Mux(eqb, eimm32, ealuimm, b);

input [31:0] eqb;

input [31:0] eimm32;

input ealuimm;

output reg [31:0] b;

always @(\*)

begin

if (ealuimm == 0)

b <= eqb;

else

b <= eimm32;

end

endmodule

module ALU(eqa, b, ealuc, r);

input [31:0] eqa;

input [31:0] b;

input [3:0] ealuc;

output reg [31:0] r;

always @(\*)

begin

case(ealuc)

'b0010:

r <= eqa + b;

'b0110:

r <= eqa - b;

'b1000:

r <= eqa & b;

'b1001:

r <= eqa | b;

'b1010:

r <= eqa ^ b;

endcase

end

endmodule

module Exe\_mem(clk, ewreg, em2reg, ewmem, edestReg, r, eqb, mwreg, mm2reg, mwmem, mdestReg, mr, mqb);

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 Data\_mem(mr, mqb, mwmem, clk, mdo);

input [31:0] mr;

input [31:0] mqb;

input mwmem;

input clk;

output reg [31:0] mdo;

reg [31:0] memory[0:127];

initial begin

memory[0]=32'hA00000AA;

memory[1]=32'h10000011;

memory[2]=32'h20000022;

memory[3]=32'h30000033;

memory[4]=32'h40000044;

memory[5]=32'h50000055;

memory[6]=32'h60000066;

memory[7]=32'h70000077;

memory[8]=32'h80000088;

memory[9]=32'h90000099;

end

always @(\*) begin

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

end

always @(negedge clk)

begin

if (mwmem == 1)

begin

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

end

end

endmodule

module Mem\_wb(clk, mwreg, mm2reg, mdestReg, mr, mdo, wwreg, wm2reg, wdestReg, wr, wdo);

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 == 0)

wbData = wr;

else

wbData = wdo;

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

module Datapath(clk, pc, dinstOut, ewreg, em2reg, ewmem, ealuc, ealuimm, edestReg, eqa, eqb, eimm32, mwreg, mm2reg, mwmem, mdestReg, mr, mqb, wwreg, wm2reg, wdestReg, wr, wdo);

input clk;

output [31:0] pc;

output [31:0] dinstOut;

output ewreg;

output em2reg;

output ewmem;

output [3:0] ealuc;

output ealuimm;

output [4:0] edestReg;

output [31:0] eqa;

output [31:0] eqb;

output [31:0] eimm32;

output mwreg, mm2reg, mwmem;

output [4:0] mdestReg;

output [31:0] mr, mqb;

output wwreg, wm2reg;

output [4:0] wdestReg;

output [31:0] wr, wdo;

wire [31:0] pc;

wire [31:0] nextPc;

wire [31:0] instOut;

wire [31:0] dinstOut;

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] imm32;

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, mm2reg, mwmem;

wire [4:0] mdestReg;

wire [31:0] mr, mqb;

wire wwreg, wm2reg;

wire [4:0] wdestReg;

wire [31:0] wr, wdo;

wire [31:0] b;

wire [31:0] r;

wire [31:0] mdo;

wire [31:0] wbData;

wire [4:0] ors, ort;

wire [1:0] fwa, fwb;

wire [31:0] fqa, fqb;

Pc\_adder pc\_adder (pc, nextPc);

PrgCounter prgCounter(nextPc, clk, pc);

InstructionMemory instructionMemory(pc, instOut);

IFIDPipelineRegister ifIdPipelineRegister(instOut, clk, dinstOut);

ControlUnit controlUnit(dinstOut[31:26], dinstOut[5:0], dinstOut[25:21], dinstOut[20:16], mdestReg, edestReg, mm2reg, mwreg,

em2reg, ewreg, wreg, m2reg, wmem, aluimm, regrt, aluc, fwa, fwb, ors, ort);

RegRTMuliplexer regRTMuliplexer(dinstOut[20:16], dinstOut[15:11], regrt, destReg);

RegisterFile registerFile(dinstOut[25:21], dinstOut[20:16], fqa, fqb, wdestReg, wbData, clk, wwreg);

ImmediateExtender immediateExtender(dinstOut[15:0], imm32);

IDEXEPipelineRegister idExePipelineRegister(wreg, m2reg, wmem, aluc, aluimm, destReg, qa, qb, imm32, clk,

ewreg, em2reg, ewmem, ealuc, ealuimm, edestReg, eqa, eqb, eimm32);

ALU\_Mux alu\_mux(eqb, eimm32, ealuimm, b);

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

Exe\_mem exe\_mem(clk, ewreg, em2reg, ewmem, edestReg, r, eqb, mwreg, mm2reg, mwmem, mdestReg, mr, mqb);

Data\_mem data\_mem(mr, mqb, mwmem, clk, mdo);

Mem\_wb mem\_wb(clk, mwreg, mm2reg, mdestReg, mr, mdo, wwreg, wm2reg, wdestReg, wr, wdo);

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

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

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

endmodule

**Testbench Code:**

`timescale 1ns / 1ps

//////////////////////////////////////////////////////////////////////////////////

// Company:

// Engineer:

//

// Create Date: 12/09/2023 02:18:37 PM

// Design Name:

// Module Name: testbench

// Project Name:

// Target Devices:

// Tool Versions:

// Description:

//

// Dependencies:

//

// Revision:

// Revision 0.01 - File Created

// Additional Comments:

//

//////////////////////////////////////////////////////////////////////////////////

module testbench();

reg clk;

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, mm2reg, mwmem;

wire [4:0] mdestReg;

wire [31:0] mr, mqb;

wire wwreg, wm2reg;

wire [4:0] wdestReg;

wire [31:0] wr, wdo;

initial begin

clk <= 1'b0;

end

Datapath datapath(clk, pc, dinstOut, ewreg, em2reg, ewmem, ealuc, ealuimm, edestReg, eqa, eqb, eimm32, mwreg, mm2reg, mwmem, mdestReg, mr, mqb, wwreg, wm2reg, wdestReg, wr, wdo);

always begin

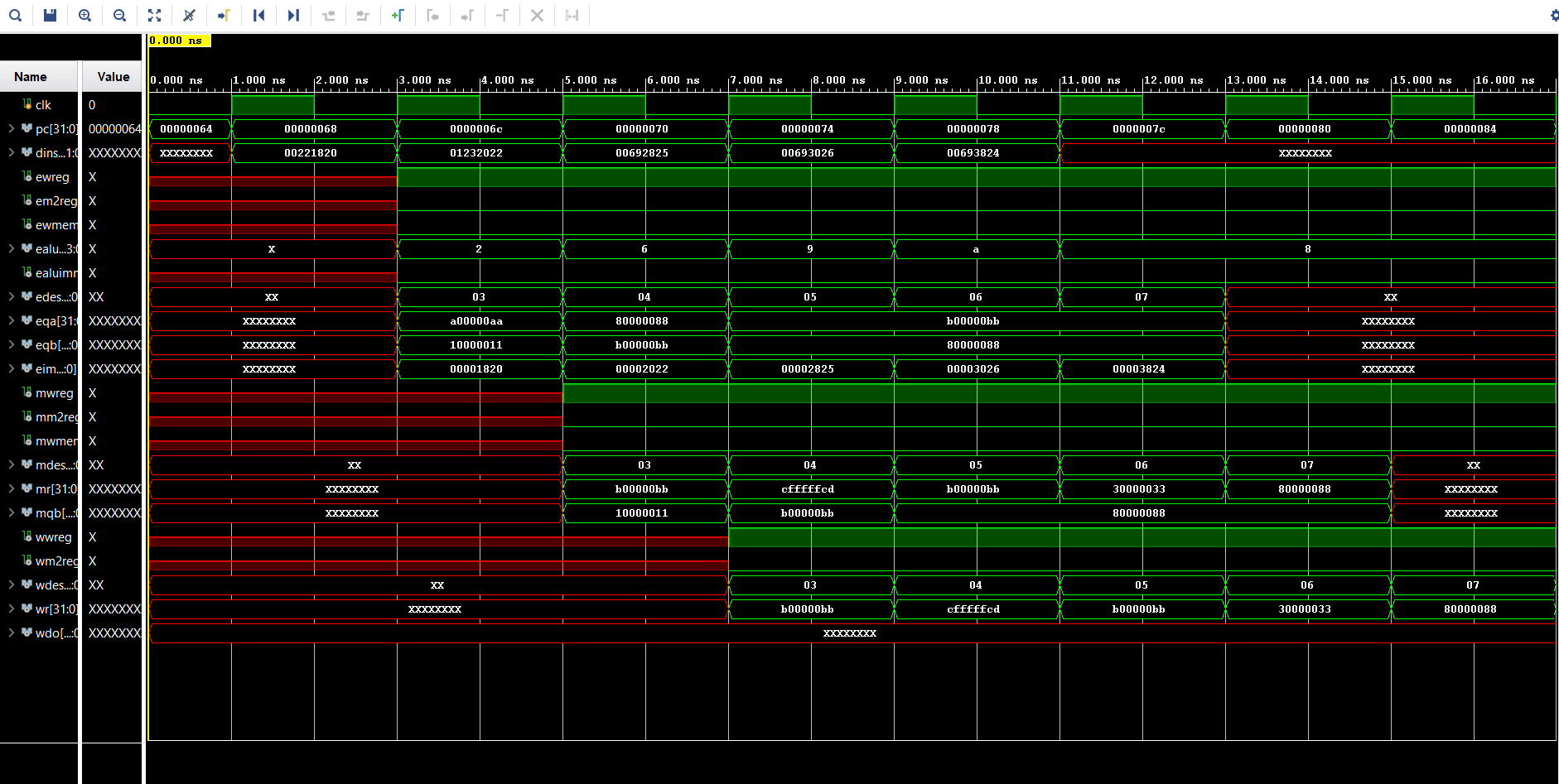
#1;

clk = ~clk;

end

endmodule

**Waveform**:



**Design Schematics:**

A diagram of a computer

Description automatically generated

**IO Planning:**

A screenshot of a game

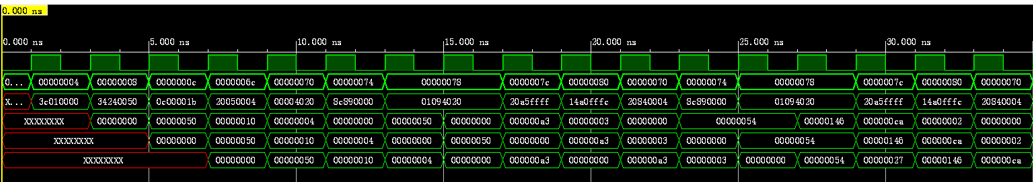
Description automatically generated

**Floor Planning:**

A screenshot of a computer screen

Description automatically generated

**Extra Credit:**   
  
Waveforms:

A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generatedA screenshot of a computer screen

Description automatically generated

Design Schematics:

A drawing of a building

Description automatically generated

A green and white drawing of a tower

Description automatically generated with medium confidence

IO Planning:

A screenshot of a game

Description automatically generated

Floor Planning:

A screen shot of a computer

Description automatically generated