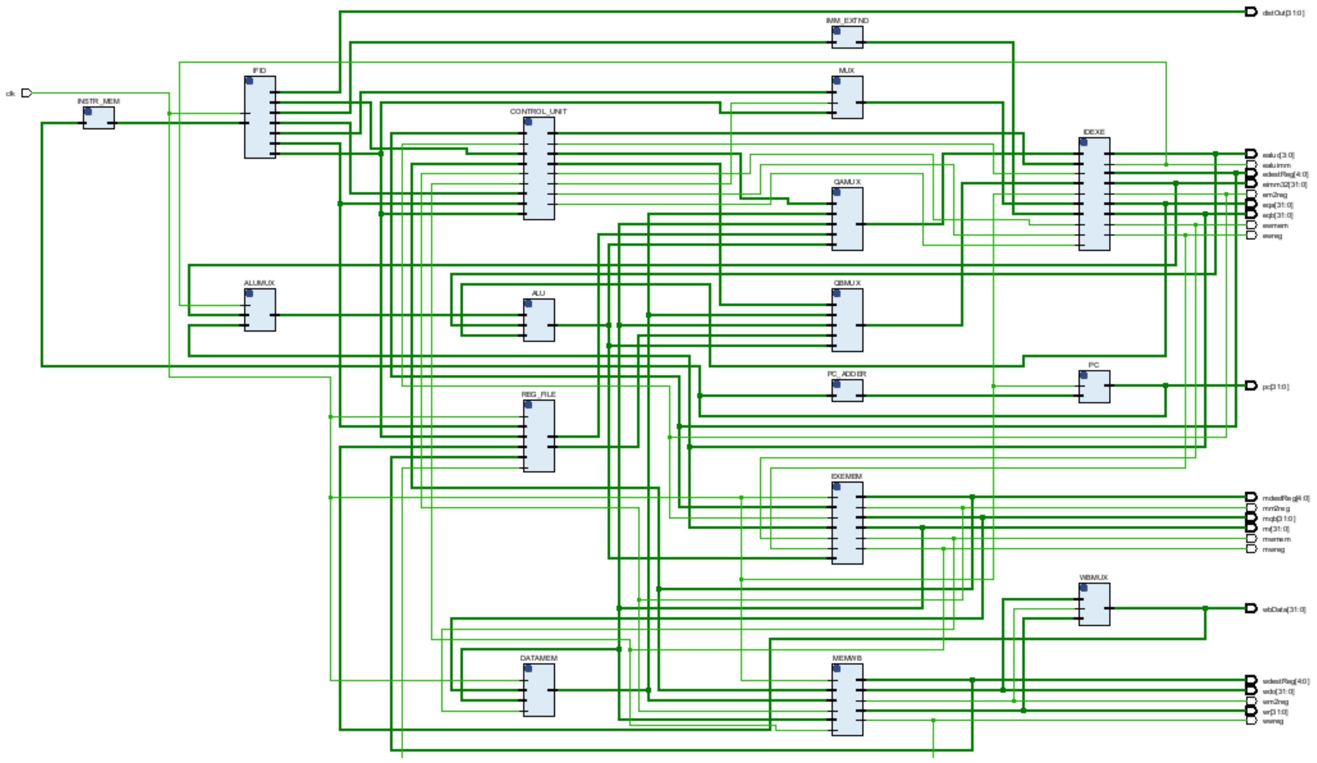
Verilog MIPS CPU Implementation



Verilog MIPS CPU Implementation

1. Abstract

This project presents the design and implementation of a 5-stage MIPS processor using Verilog. The MIPS architecture is a widely used RISC (Reduced Instruction Set Computing) architecture, known for its simplicity and efficiency. The processor design consists of five pipeline stages: Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), and Write Back (WB), adhering to the classic MIPS pipeline structure.

One crucial aspect addressed in this implementation is data forwarding, a technique employed to mitigate data hazards in pipelined processors. Data hazards arise when instructions depend on the results of previous instructions that have not yet completed execution. Forwarding allows the processor to transmit data directly from the output of one pipeline stage to the input of another, bypassing the need to wait for data to be written back to the register file. The source of the data is chosen by a mux, so forwarding can be correctly implemented and controlled. By implementing forwarding logic, this processor effectively resolves data hazards, enhancing its performance and throughput.

The MIPS architecture's simplicity, coupled with the efficient pipelined implementation, makes it a good choice for chip architecture. The streamlined instruction set and pipelined structure contribute to reduced hardware complexity, enabling efficient use of resources and facilitating higher clock frequencies. Moreover, the inclusion of data forwarding mechanisms enhances the processor's efficiency and reduces the likelihood of pipeline stalls, further improving its performance characteristics.

Overall, this project demonstrates the successful implementation of a 5-stage MIPS processor using Verilog, showcasing the importance of data forwarding in pipelined architectures and the suitability of the design for chip implementation.

2. Extra Credit Explanation

The extra credit implements a lot of functionality to the implementation. The biggest one is stalling. Stalling, also known as pipeline stall or pipeline bubble, refers to the halting of the processor's pipeline due to various hazards. When a stall occurs, the pipeline stages stop progressing, causing a delay in the execution of subsequent instructions.

Stalling is a crucial concept in processor design because it directly impacts the overall performance and efficiency of the processor. Understanding and effectively managing stalls are essential for optimizing processor throughput and achieving high performance. For example, during a beq or a bne instruction, The conditions are checked in the ID stage so there is only one stall as opposed to multiple.

Stalling was implemented used a PC Reg Pipeline, so the value of PC could be paused for a stall. This also allowed for controlling PC based off of various instructions (such as j or jr). By letting the PC have multiple sources, the implementation is much more efficient and can handle many more vital instructions seen in MIPs.

Other features were also added in order to facilitate j-type instructions and others such as lui and andi. The entire code can been seen below.

3. Test Bench Desgin Code

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company: PSU CMPEN 331 SP24 Section 02
// Engineer: Jenicy Strong
//
// Create Date: 03/11/2024 06:25:19 PM
// Design Name: cpu testbench
// Project Name: Lab 4
//
//////////////////////////////////////////////////////////////////////////////////


module testbench();
    reg clk;
    wire [31:0] pc, eqa, eqb, eimm32, distOut, wr, wdo, mr, mqb;
    wire [4:0] edestReg, wdestReg, mdestReg;
    wire [3:0] ealuc;
    wire ewreg, em2reg, ealuimm, ewmem, wwreg, wm2reg, mwreg, mm2reg, mwmem;

    parameter PERIOD = 100;

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

    initial begin
        clk = 0;
    end

    always
        #(PERIOD / 2) clk = ~clk;

endmodule
```

4. Project Design Code

4.1. PC Mux

```
module pc_mux(pc4, bpc, afwrd, jpc, pcsrc, npc);
    input wire [31:0] pc4, bpc, afwrd, jpc;
    input wire [1:0] pcsrc;
    output reg [31:0] npc;

    always @(*) begin
        case(pcsrc)
            2'b00:
                 npc = pc4;
            2'b01:
                 npc = bpc;
            2'b10:
                 npc = afwrd;
            2'b11:
                 npc = jpc;

        endcase
    end
endmodule
```

4.2. PC Register

```
module pc_register(wpcir, npc, clk, pc);
    input wire wpcir, clk;
    input wire [31:0] npc;
    output reg [31:0] pc;

    initial begin
        pc = 32'd100; //initalize pc to 100dec
    end


    always @(posedge clk) begin
        if(wpcir == 1)
            pc = npc;
    end

endmodule
```

4.3. Instruction Memeory

```
module instruction_memory(pc, instOut);
    input wire [31:0] pc;
    output reg[31:0] instOut;

    reg [31:0] memory [0:63];
    initial begin
    /*
        memory[25] = {6'b100011, 5'b00000, 5'b00010, 16'h0000}; //lw $2, 00($1)
        memory[26] = {6'b100011, 5'b00000, 5'b00011, 16'h0004}; //lw $3, 04($1)
        memory[27] = {6'b100011, 5'b00000, 5'b00100, 16'h0008}; //lw $4, 08($1)
        memory[28] = {6'b100011, 5'b00000, 5'b00101, 16'h000c}; //lw $5, 12($1)
        memory[29] = {6'b100011, 5'b00000, 5'b01001, 16'h0010}; //lw $9, 16($1)

        memory[30] = {6'b0, 5'b00010, 5'b01010, 5'b00110, 11'b00000100000}; //add $6,
$2, $10
        */
         //code provided via pdf
        memory[25] = 32'h3c010000;
        memory[26] = 32'h34240050;
        memory[27] = 32'h0c00001b;
        memory[28] = 32'h20050004;
        memory[29] = 32'hac820000;
        memory[30] = 32'h8c890000;
        memory[31] = 32'h01244022;
        memory[32] = 32'h20050003;
        memory[33] = 32'h20a5ffff;
        memory[34] = 32'h34a8ffff;
        memory[35] = 32'h39085555;
        memory[36] = 32'h2009ffff;
        memory[37] = 32'h312affff;
        memory[38] = 32'h01493025;
        memory[39] = 32'h01494026;
        memory[40] = 32'h01463824;
        memory[41] = 32'h10a00003;
        memory[42] = 32'h00000000;
        memory[43] = 32'h08000008;
        memory[44] = 32'h00000000;
        memory[45] = 32'h2005ffff;
        memory[46] = 32'h000543c0;
        memory[47] = 32'h00084400;
        memory[48] = 32'h00084403;
        memory[49] = 32'h000843c2;
        memory[50] = 32'h08000019;
        memory[51] = 32'h00000000;
        memory[52] = 32'h00004020;
        memory[53] = 32'h8c890000;
        memory[54] = 32'h01094020;
        memory[55] = 32'h20a5ffff;
        memory[56] = 32'h14a0fffc;
        memory[57] = 32'h20840004;
        memory[58] = 32'h03e00008;
        memory[59] = 32'h00081000;
   end
```

```
    always @(*) begin
        instOut = memory[pc[7:2]];
    end

endmodule
```

4.4. PC Adder

```
module pc_adder(pc, pc4);
    input wire [31:0] pc;
    output reg [31:0] pc4;

    always @(*) begin
        pc4 = pc + 4;
    end

 endmodule
```

4.5. IFID Pipeline Register

```
module ifid_pipeline_register(instOut, clk, pc4, wpcir, op, func, rd, rt, rs, imm,
distOut, dpc4, addr);
    input wire [31:0] instOut, pc4;
    input wire clk, wpcir;
    output reg [5:0] op, func;
    output reg [4:0] rs, rd, rt;
    output reg [15:0] imm;
    output reg [31:0] distOut, dpc4;
    output reg [25:0] addr;

    always @(posedge clk) begin
        op = instOut[31:26];
        rs = instOut[25:21];
        rt = instOut[20:16];
        rd = instOut[15:11];
        imm = instOut[15:0];
        func = instOut[5:0];
        addr = instOut[25:0] << 2;
        distOut = instOut;

        if(wpcir==1)
            dpc4 = pc4;
    end

endmodule
```

4.6. PC4 Adder

```
module adder_dpc4imm(dpc4, imm, bpc);
    input wire [31:0] dpc4;
    input wire [15:0] imm;
    output reg [31:0] bpc;

    always @(*) begin
        bpc = (imm << 2) + dpc4;
    end

endmodule
```

4.7. Add JPC

```
module add_jpc(addr, dpc4, jpc);
    input wire [25:0] addr;
    input wire [31:0] dpc4;
    output reg [31:0] jpc;

    always @(*) begin
        jpc = addr + dpc4;
    end

endmodule
```

4.8. Control Unit

```
module control_unit(op, func, rs, rt, mdestReg, mm2reg, mwreg, edestReg, em2reg,
ewreg, wreg, m2reg,
                        wmem, aluc, aluimm, regrt, fwdb, fwda, pcsrc, wpcir, sext,
shift, jal);
    input wire [5:0] op, func;
    input wire [4:0]  edestReg, mdestReg, rs, rt;
    input wire mm2reg, mwreg, ewreg, em2reg;
    output reg wreg, m2reg, wmem, aluimm, regrt, wpcir, sext, jal;
    output reg [3:0] aluc;
    output reg [1:0] fwdb, fwda, pcsrc, shift;

    reg i_rs, i_rt, rsrtequ;

    initial begin
        fwdb = 2'b00;
        fwda = 2'b00;
        pcsrc = 2'b00;
    end


    always @(*) begin
        if(rs==rt) begin
            rsrtequ=1;
        end
        else begin
            rsrtequ = 0;
        end
```

```
fwdb = 2'b00;
fwda = 2'b00;
i_rs = 1;
i_rt = 1;
jal = 0;
shift = 0;
pcsrc = 2'b00;

if (rt == edestReg) begin
    fwdb = 2'b01;
end
else if (rt == mdestReg) begin
    fwdb = 2'b10;
end

if (rs == edestReg) begin
    fwda = 2'b01;
end
else if (rs == mdestReg) begin
    fwda = 2'b10;
    if (op == 6'b100011) begin //LW
        fwda = 2'b11;
    end
end



 case(op)
    6'b000000: //r-type operation
    begin
        wreg = 1; //all r-types write to reg except jr
        m2reg = 0; //only 1 when lw
        wmem = 0; //only 1 when sw
        aluimm = 0; //only i-types
        regrt = 0; //only i-types
        sext = 1;
        i_rt = 1; //all r type but jr
        case(func)
            6'b100000: //adding
                aluc = 4'b0010;
            6'b100010: //subtracting
                aluc = 4'b0110;
            6'b100100: //and
                aluc = 4'b0000;
            6'b100101: // or
                aluc = 4'b0001;
            6'b100110: //xor
                aluc = 4'b0011;
            6'b000000: begin //sll
```

```
                aluc = 4'b1000;
                i_rs = 0;
                shift = 1;
            end
            6'b000010: begin //srl
                aluc = 4'b1001;
                i_rs = 0;
                shift = 1;
            end
            6'b000011: begin //sra
                aluc = 4'b1011;
                i_rs = 0;
                shift = 1;
            end
            6'b001000: begin //jr
                wreg = 0;
                pcsrc = 2'b10;
                i_rt = 0;
            end
        endcase
    end

    6'b100011: //LW
    begin
        wreg = 1;
        m2reg = 1;
        wmem = 0;
        aluc = 4'b0010;
        aluimm = 1;
        regrt = 1;
        sext = 1;
        i_rt = 1;
    end
    6'b101011: //SW
    begin
        wreg = 0;
        m2reg = 0;
        wmem = 1;
        aluc = 4'b0010;
        aluimm = 1;
        regrt = 1;
        sext = 1;
        i_rt = 1;
    end
    6'b001111: begin //lui
        wreg = 1; //all r-types write to reg except jr
        m2reg = 0; //only 1 when lw
        wmem = 0; //only 1 when sw
        aluc = 4'b0010; //adding (0+immediate)
        aluimm = 1; //only i-types
        regrt = 1; //only i-types
        sext = 0;
    end
    6'b001101: begin // ori
        wreg = 1;
        aluc = 4'b0001;//or
        m2reg = 0; //notlw
        wmem = 0; //not sw
        aluimm = 1; //not r-type
        regrt = 1; //not r-type
        i_rt = 1;
        sext = 0;
    end
    6'b000011: begin // jal
        wreg = 0;
        m2reg = 0; //notlw
        wmem = 0; //not sw
        aluimm = 1; //not r-type
        regrt = 1; //not r-type
        i_rt = 1;
        sext = 0;
        pcsrc = 2'b11;
        i_rs = 0;
        i_rt = 0;
        jal = 1;
    end
    6'b001000: begin // addi
        wreg = 1;
        m2reg = 0; //notlw
        aluc = 4'b0010; //adding
        wmem = 0; //not sw
        aluimm = 1; //not r-type
        regrt = 1; //not r-type;
        sext = 0;
    end
     6'b001110: begin //xori
        wreg = 1;
        aluc = 4'b0011; // xor
        m2reg = 0; //notlw
        wmem = 0; //not sw
        aluimm = 1; //not r-type
        regrt = 1; //not r-type
        i_rt = 1;
        sext = 0;
        shift = 1;
    end

    6'b001100: begin //andi
```

```
        wreg = 1;
        m2reg = 0; //notlw
        aluc = 4'b0000; //amding
        wmem = 0; //not sw
        aluimm = 1; //not r-type
        regrt = 1; //not r-type
        sext = 0;
        shift = 0;
    end
    6'b000100: begin // beq
        wreg = 0;
        m2reg = 0; //notlw
        wmem = 0; //not sw
        aluimm = 1; //not r-type
        regrt = 1; //not r-type
        i_rt = 1;
        sext = 1;
        shift = 0;
        //aluc = 4'b0110;
        if(rsrtequ == 1) begin
            pcsrc = 2'b01;
        end
    end
    6'b000101: begin // bne
        wreg = 0;
        m2reg = 0; //notlw
        wmem = 0; //not sw
        aluimm = 1; //not r-type
        regrt = 1; //not r-type
        sext = 1;
        shift = 0;
        //aluc = 4'b0110;
        if(rsrtequ == 0) begin
            pcsrc = 2'b01;
        end
    end

    6'b000010: begin //j
        wreg = 0;
        m2reg = 0; //notlw
        wmem = 0; //not sw
        aluimm = 1; //not r-type
        regrt = 1; //not r-type
        sext = 0;
        shift = 0;
        pcsrc = 2'b11;
        i_rt = 0;
        i_rs = 0;
    end
```

```
         endcase
         wpcir = ~(ewreg & em2reg & (edestReg!=0) & (i_rs & (edestReg== rs) | i_rt &
(edestReg == rt)));
     end
endmodule
```

4.9. Qa Mux

```
module qa_mux(fwda, qa, r, mr, mdo, afwrd);
    input wire [1:0] fwda;
    input wire [31:0] qa, r, mr, mdo;
    output reg [31:0] afwrd;


    always @(*) begin
        case(fwda)
        2'b00: begin
            afwrd = qa;
        end
        2'b01: begin
            afwrd = r;
        end
        2'b10: begin
            afwrd = mr;
        end
        2'b11: begin
            afwrd = mdo;
        end
       endcase

    end
endmodule
```

4.10. Qb Mux

```
module qb_mux(fwdb, qb, r, mr, mdo, bfwrd);
    input wire [1:0] fwdb;
    input wire [31:0] qb, r, mr, mdo;
    output reg [31:0] bfwrd;


    always @(*) begin
        case(fwdb)
        2'b00: begin
            bfwrd = qb;
        end
        2'b01: begin
            bfwrd = r;
        end
        2'b10: begin
            bfwrd = mr;
        end
        2'b11: begin
            bfwrd = mdo;
        end
       endcase

    end

endmodule
```

4.11. Regrt Mux

```
module regrt_mux(rt, rd, regrt, destReg);
    input wire [4:0] rt, rd;
    input wire regrt;
    output reg [4:0] destReg;

    always @(*) begin
        if(regrt == 0)
            destReg = rd;
        else
            destReg = rt;
    end

endmodule
```

4.12. Register File

```
module register_file(rs, rt, wdestReg, wbData, wwreg, clk, qa, qb);
    input wire [4:0] rs, rt, wdestReg;
    input wire [31:0] wbData;
    input wire wwreg, clk;
    output reg [31:0] qa, qb;

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

    initial begin
        registers[0] = 32'h0;
        registers[1] = 32'h0;
        registers[2] = 32'h0;
        registers[3] = 32'h0;
        registers[4] = 32'h0;
        registers[5] = 32'h0;
        registers[6] = 32'h0;
        registers[7] = 32'h0;
        registers[8] = 32'h0;
        registers[9] = 32'h0;
        registers[10] = 32'h0;
        registers[11] = 32'h0;
        registers[12] = 32'h0;
        registers[13] = 32'h0;
        registers[14] = 32'h0;
        registers[15] = 32'h0;
        registers[16] = 32'h0;
        registers[17] = 32'h0;
        registers[18] = 32'h0;
        registers[19] = 32'h0;
        registers[20] = 32'h0;
        registers[21] = 32'h0;
        registers[22] = 32'h0;
        registers[23] = 32'h0;
        registers[24] = 32'h0;
        registers[25] = 32'h0;
        registers[26] = 32'h0;
        registers[27] = 32'h0;
        registers[28] = 32'h0;
        registers[29] = 32'h0;
        registers[30] = 32'h0;
        registers[31] = 32'h0;
    end

    always @(*) begin
        qa = registers[rs];
        qb = registers[rt];
    end

    always @(negedge clk) begin
        if (wwreg == 1)
            registers[wdestReg] = wbData;
    end

endmodule
```

4.13. Imm Extnder

```
module imm_extnd(imm, sext, imm32);
    input wire [15:0] imm;
    input wire sext;
    output reg [31:0] imm32;

    always @(*) begin
        if (sext == 1)
            imm32 = {{16{imm[15]}}, imm[15:0]};
        else
            imm32 = 32'b0 + imm;
    end
endmodule
```

4.14. IDEXE Pipeline Reg

```
module idexe_pipeline_reg(wreg, m2reg, wmem, aluc, aluimm, destReg, afwrd, bfwrd,
imm32, dpc4, jal, shift, clk,
                            ewreg, em2reg, ewmem, ealuc, ealuimm, edestReg0, eqa, eqb,
eimm32, epc4, ejal, eshift);
    input wire wreg, m2reg, aluimm, wmem, jal, shift, clk;
    input wire [4:0] destReg;
    input wire [3:0] aluc;
    input wire [31:0] afwrd, bfwrd, imm32, dpc4;
    output reg ewreg, em2reg, ealuimm, ewmem, ejal, eshift;
    output reg [4:0] edestReg0;
    output reg [3:0] ealuc;
    output reg [31:0] eqa, eqb, eimm32, epc4;

    always @(posedge clk) begin
        ewreg = wreg;
        em2reg = m2reg;
        ewmem = wmem;
        ealuc = aluc;
        ealuimm = aluimm;
        edestReg0 = destReg;
        eqa = afwrd;
        eqb = bfwrd;
        eimm32 = imm32;
        ejal = jal;
        eshift = shift;
        epc4 = dpc4;
    end

endmodule
```

4.15. Pc4 Adder

```
module pc4_addr(epc4, epc8);
    input wire [31:0] epc4;
    output reg [31:0] epc8;

    always @(*) begin
        epc8 = epc4 + 4;
    end

endmodule
```

4.16. Shift Mux

```
module shift_mux(eimm32, eqa, eshift, aout);
    input wire [31:0] eimm32, eqa;
    input wire eshift;
    output reg [31:0] aout;

    always @(*) begin
        if (eshift == 1)
            aout = eimm32;
        else
```

```
            aout = eqa;
    end

endmodule
```

4.17. ALU Mux

```
module alu_mux(eqb, eimm32, ealuimm, b);
    input wire [31:0] eqb, eimm32;
    input wire ealuimm;
    output reg [31:0] b;

    always @* begin
        if (ealuimm == 1)
            b = eimm32;
        else
            b = eqb;
        end

endmodule
```

4.18. ALU

```
module alu(eqa, b , ealuc, r);
    input wire [31:0] eqa, b;
    input wire [3:0] ealuc;
    output reg [31:0] r;

    always @* begin
        case (ealuc)
            4'b0010: //add
            begin
                r = eqa+b;
            end
            4'b0110: //subtract
            begin
                r = eqa-b;

            end
            4'b0001: //or
            begin
                r = eqa | b;
            end
            4'b0000: //and
            begin
                r = eqa & b;
            end
            4'b0011: //xor
            begin
                r = eqa ^ b;
            end
            4'b1000: //sll
            begin
                r = b<<(eqa);
            end
            4'b1001: //srl
            begin
                r = b>>(eqa);
            end
            4'b1011: //sra
            begin
                r = b>>>(eqa);
            end
        endcase
    end

endmodule
```

4.19. JAL Mux

```
module jal_mux(ejal, r, epc8, ealu);
    input wire [31:0] r, epc8;
    input wire ejal;
    output reg [31:0] ealu;

    always @(*) begin
        if (ejal== 1)
            ealu = epc8;
        else
            ealu = r;
    end

endmodule
```

4.20. F

```
module f(edestReg0, ejal, edestReg);
    input wire[4:0] edestReg0;
    input wire ejal;
    output reg [4:0] edestReg;

    always @(*) begin
        if (ejal ==1)
            edestReg = 4'd31;
        else
            edestReg = edestReg0;
    end

endmodule
```

4.21. EXEMEM Pipeline Reg

```
module exemem_pipeline_reg(ewreg, em2reg, ewmem, edestReg, ealu, eqb, clk,
                             mwreg, mm2reg, mwmem, mdestReg, mr, mqb);
    input wire [31:0] ealu, eqb;
    input wire [4:0] edestReg;
    input wire ewreg, em2reg, ewmem, clk;
    output reg [31:0] mr, mqb;
    output reg [4:0] mdestReg;
    output reg mwreg, mm2reg, mwmem;


    always @(posedge clk) begin
        mr = ealu;
        mqb = eqb;
        mdestReg = edestReg;
        mwreg = ewreg;
        mm2reg =em2reg;
        mwmem = ewmem;
    end

endmodule
```

4.22. Data Memory

```
module data_mem(mr, mqb, mwmem, clk, mdo);
    input wire [31:0] mr, mqb;
    input wire mwmem, clk;
    output reg [31:0] mdo;

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

    initial begin
    /*
        datMemory[0] = 32'hA00000AA;
        datMemory[1] = 32'h10000011;
        datMemory[2] = 32'h20000022;
        datMemory[3] = 32'h30000033;
        datMemory[4] = 32'h40000044;
        datMemory[5] = 32'h50000055;
        datMemory[6] = 32'h60000066;
        datMemory[7] = 32'h70000077;
        datMemory[8] = 32'h80000088;
        datMemory[9] = 32'h90000099;
        */
        datMemory[5'h14] = 32'h000000a3;
        datMemory[5'h15] = 32'h00000027;
        datMemory[5'h16] = 32'h00000079;
        datMemory[5'h17] = 32'h00000115;
   end
```

```
    always @* begin
        mdo = datMemory[mr[7:2]];
    end

    always @(negedge clk) begin
        if (mwmem ==1)
            // mqb = mem array at pos mr
            datMemory[mr] = mqb;

    end

endmodule
```

4.23. MEMWB Pipeline Reg

```
module memwb_pipeline_reg(mwreg, mm2reg, mdestReg, mr, mdo, clk,
                             wwreg, wm2reg, wdestReg, wr, wdo);
    input wire [31:0] mr, mdo;
    input wire [4:0] mdestReg;
    input wire mwreg, mm2reg, clk;
    output reg [31:0] wr, wdo;
    output reg [4:0] wdestReg;
    output reg wwreg, wm2reg;


    always @(posedge clk) begin
        wwreg = mwreg;
        wm2reg = mm2reg;
        wdestReg = mdestReg;
        wr = mr;
        wdo = mdo;
    end
endmodule
```

4.24. WB Mux

```
module wb_mux(wr, wdo, wm2reg, wbData);
    input wire [31:0] wr, wdo;
    input wire wm2reg;
    output reg [31:0] wbData;

    always @* begin
        if(wm2reg == 1)
            wbData = wdo;
        else
            wbData = wr;
    end


endmodule
```

4.25. Datapath

```
module datapath(clk, pc, distOut, ewreg, em2reg, ewmem, ealuc, ealuimm, edestReg, eqa,
eqb, eimm32, wwreg, wm2reg, wdestReg, wr, wdo,
                    mwreg, mm2reg, mwmem, mdestReg, mr, mqb, wbData);
    input wire clk;
    output wire [31:0] pc, eqa, eqb, eimm32, distOut, wr, wdo, mr, mqb, wbData;
    output wire [4:0] edestReg, wdestReg, mdestReg;
    output wire [3:0] ealuc;
    output wire ewreg, em2reg, ealuimm, ewmem, wwreg, wm2reg, mwreg, mm2reg, mwmem;

    wire wreg, m2reg, wmem, aluimm, regrt, wpcir, eql, sext, ejal, jal, eshift;
    wire [4:0] destReg, rs, rt, rd, edestReg0;
    wire [5:0] op, func;
    wire [3:0] aluc;
    wire [31:0] qa, qb, imm32, nextPc_to_PC, instOut, b, r, mdo, afwrd, bfwrd, pc4,
bpc, jpc;
    wire [31:0] npc, dpc4, epc4, epc8, aout, ealu;
    wire [25:0] addr;
    wire [15:0] imm;
    wire [1:0] pcsrc, fwdb, fwda, shift;



    pc_mux pcMux(pc4, bpc, afwrd, jpc, pcsrc, npc);
    pc_register pcReg(wpcir, npc, clk, pc);
    instruction_memory instructMem(pc, instOut);
    pc_adder pcAdder(pc, pc4);
    ifid_pipeline_register IFID_Reg(instOut, clk, pc4, wpcir, op, func, rd, rt, rs,
imm, distOut, dpc4, addr);



    adder_dpc4imm pcAdder4Imm(dpc4, imm, bpc);
    dadb_eql daDbEql(afwrd, bfwrd, eql);
    add_jpc addJPC(addr, dpc4, jpc);
    control_unit controlUnit(op, func, rs, rt, mdestReg, mm2reg, mwreg, edestReg,
em2reg, ewreg, wreg, m2reg,
                        wmem, aluc, aluimm, regrt, fwdb, fwda, pcsrc, wpcir, sext,
shift, jal);
    qa_mux qaMux(fwda, qa, r, mr, mdo, afwrd);
    qb_mux qbMux(fwdb, qb, r, mr, mdo, bfwrd);
    regrt_mux regretMux(rt, rd, regrt, destReg);
    register_file regFile(rs, rt, wdestReg, wbData, wwreg, clk, qa, qb);
    imm_extnd immExtd(imm, sext, imm32);
    idexe_pipeline_reg IDEXE_Reg(wreg, m2reg, wmem, aluc, aluimm, destReg, afwrd,
bfwrd, imm32, dpc4, jal, shift, clk,
                            ewreg, em2reg, ewmem, ealuc, ealuimm, edestReg0, eqa, eqb,
eimm32, epc4, ejal, eshift);


    pc4_addr pc4Addr(epc4, epc8);
    shift_mux shiftMux(eimm32, eqa, eshift, aout);
    alu_mux aluMux(eqb, eimm32, ealuimm, b);
    alu ALU(eqa, b , ealuc, r);
    jal_mux jalMux(ejal, r, epc8, ealu);
    f F(edestReg0, ejal, edestReg);
    exemem_pipeline_reg EXEMEM_Reg(ewreg, em2reg, ewmem, edestReg, ealu, eqb, clk,
                             mwreg, mm2reg, mwmem, mdestReg, mr, mqb);


    data_mem dataMem(mr, mqb, mwmem, clk, mdo);
    memwb_pipeline_reg MEMWB_Reg(mwreg, mm2reg, mdestReg, mr, mdo, clk,
                             wwreg, wm2reg, wdestReg, wr, wdo);
    wb_mux wbMux(wr, wdo, wm2reg, wbData);

endmodule
```

5. Schematics

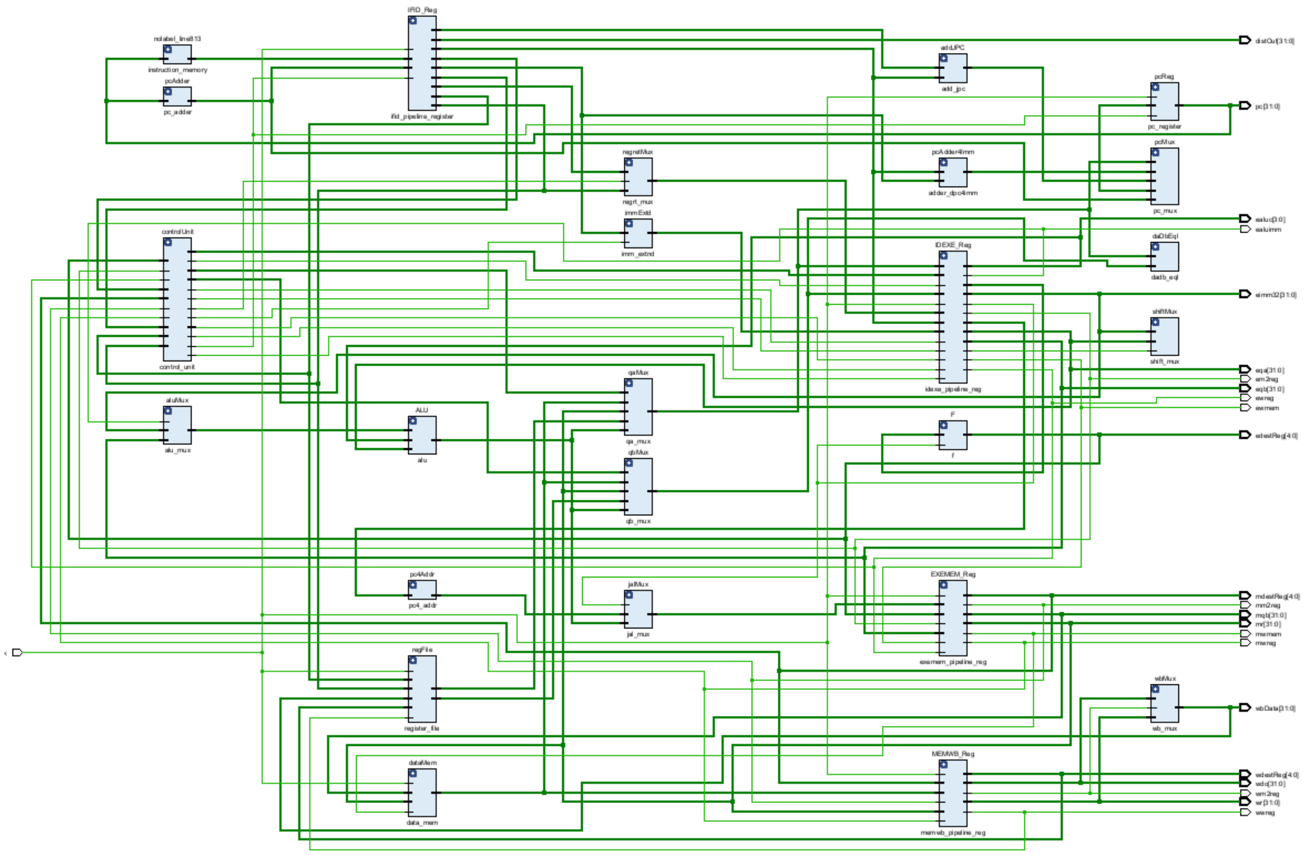


Figure 1: Schematic generated from Xilinx synthesis of the total design and the extra credit.

6. Wave Form

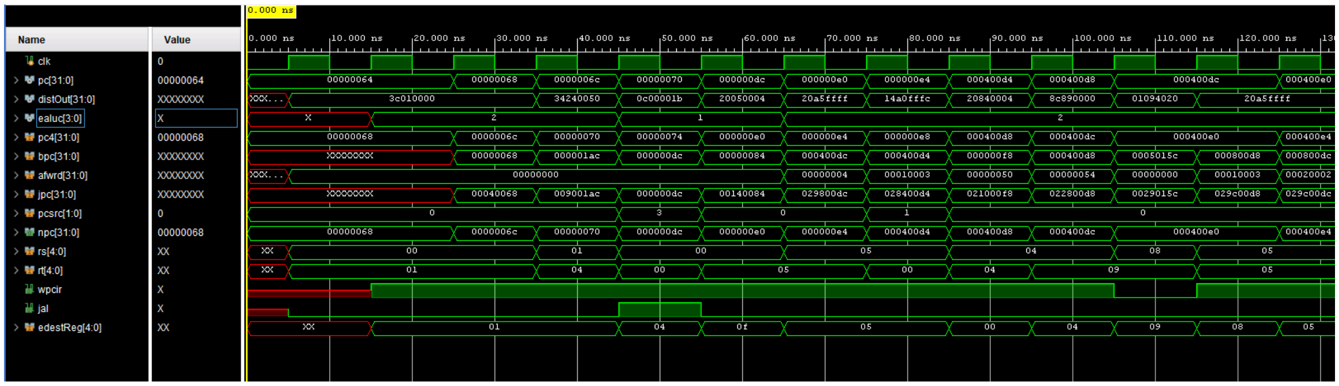


Figure 2: Waveform produced from simulation of the design code showing asked for regs.

7. I/O Planning

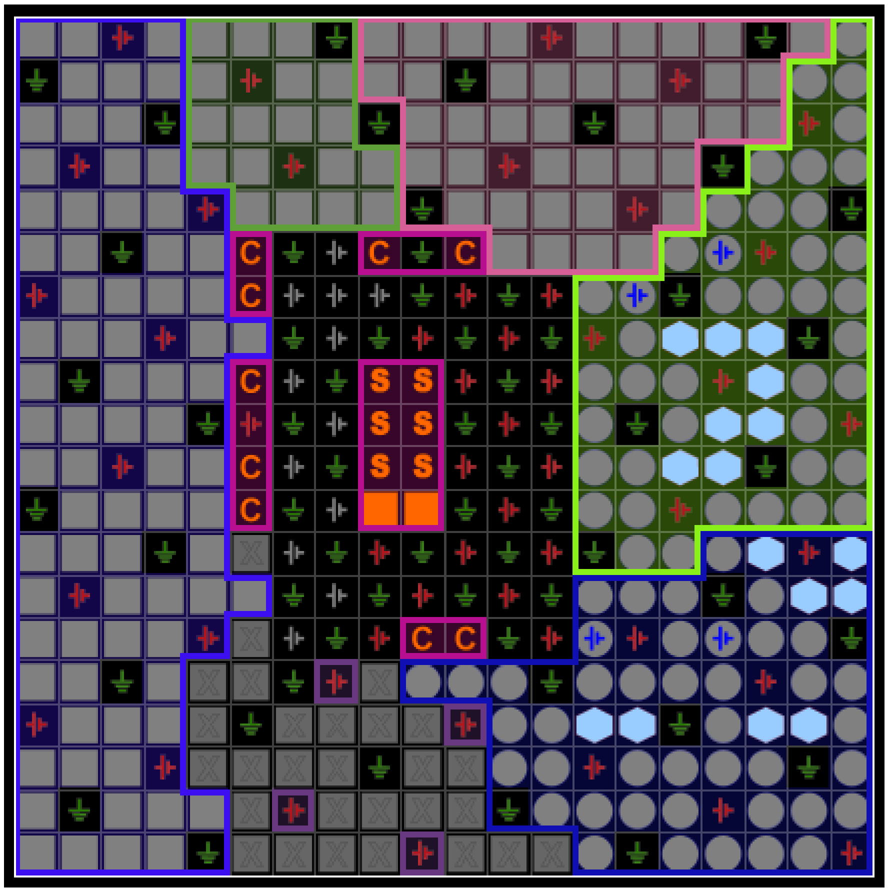


Figure 3: I/O planning generated from Xilinx of the design.

8. Floor Planning

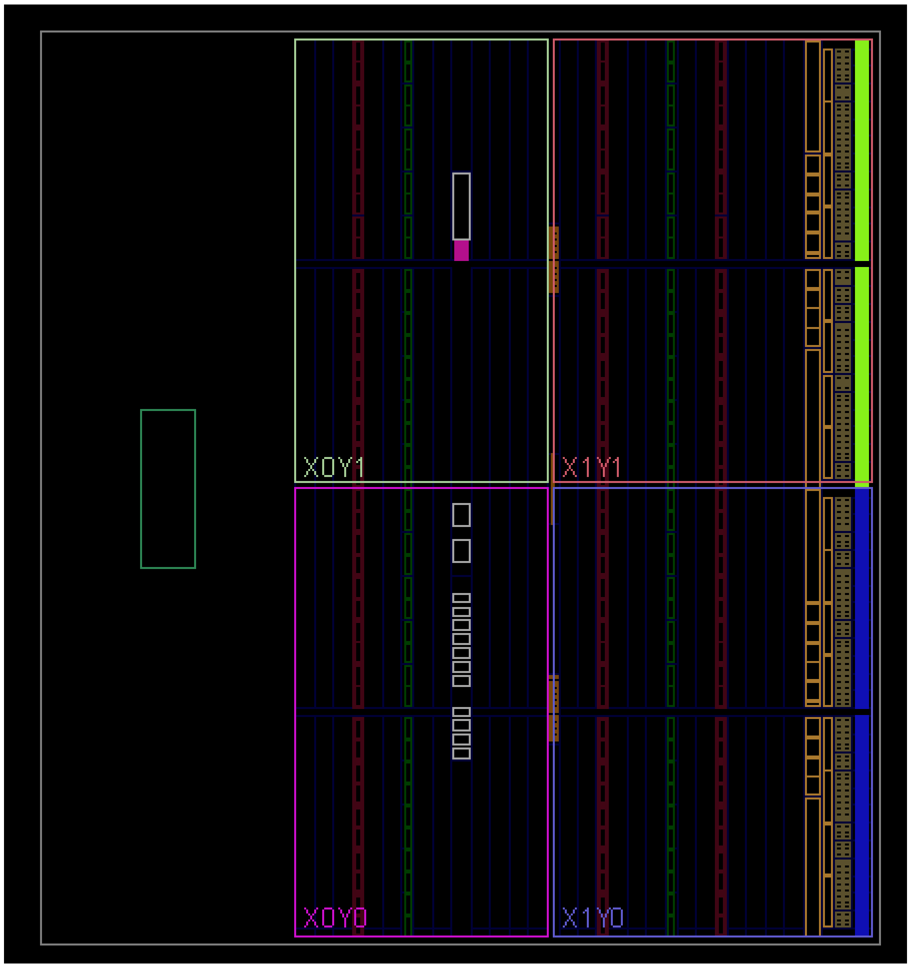


Figure 4: Floor Planning planning generated from Xilinx of the design.