Final Project

David Rohweder

CMPEN 331

May 7, 2022

Abstract

In the final project for CMPEN 331 we sought to implement a forwarding functionality to the existing CPU schema. The reason to modify the circuit was to prevent data hazards from occurring which prevents data coherence. To achieve this functionality two multiplexers are added to the circuit to choose what input source new instructions should pull data from if the control unit detects a potential hazard will occur for the given instruction operands.

Introduction

When designing this implementation of a CPU there are five distinct processing regions in which the instructions are executed. To begin, on each cycle a instruction from the instruction memory unit will read the instruction at the pc count then increment to the next pc + 4. Next, after the instruction is fetched it will be decoded in the instruction decode unit where the opcode, function, rs, rt, rd, and imm value will be read. After, the control unit will process the opcode to determine what instruction type it is (R-type or I-type). From this information the control unit will determine if the instruction is writing to the memory, writing from the memory to the register, or writing to the register. Furthermore, the control unit will determine the alu operation to perform on the rs and rt registers and if an I-type word to use the aluimm value. A new feature is forwarding and if the control unit detects a data hazard in rs or rt it will determine where to pull the most recent data from. The output of the control unit feeds into 3 multiplexors to determine if an I-type to use rt as the destination register, if a hazard is in rs to use which source of data, and if a hazard is in rt to use which source of data. Next, the data is passed to the exe phase where the ALU determines whether to use the aluimm or rt value in processing. After, the ALU performs the computation based on the ALU code determined by the control unit. The output of the exe phase is then sent to the memory phase where if the control unit determines we save the result of the ALU to the data memory or not. Next, the data is passed from the data memory phase to the write back phase, where either the data memory result or the ALU computation result is stored in the register, and then if the control unit wants that result to be saved to the register then that data is placed into either the determined rd or rt register place.

These commands are executed in a pipeline fashion where instructions are being computed together, each at a different phase of the phases described above (IF, ID, EXE, MEM, and WB) and as seen in *Figure 1*.

The benefit or an architecture as shown in *Figure 1* is a pipelined, data hazard free CPU which can process instructions much faster than if run serially and a data coherent system where the data will always be reflected correctly.

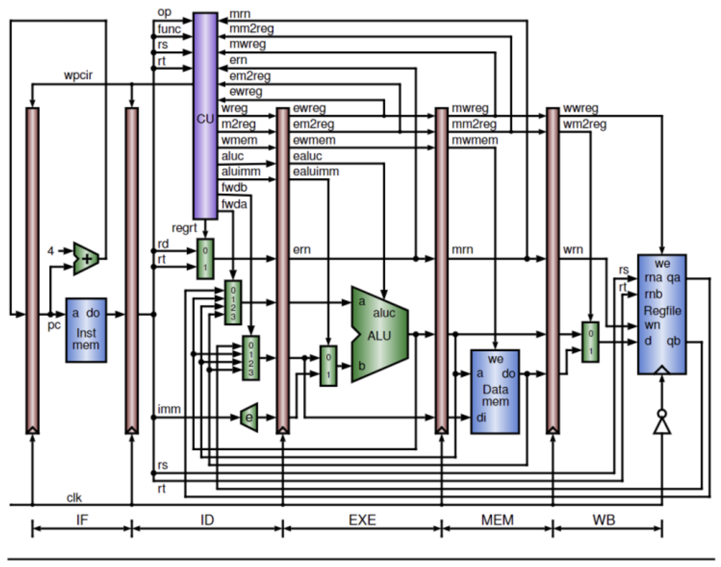


Figure 1: Circuit Schema

Verilog Design Code:

Device: XC7Z010CLG400-1

Test bench code:

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company: PSU CMPEN 331
// Engineer: David Rohweder
//
// Create Date: 03/21/2022 04:25:25 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 clock;
    wire [31:0]pc;
    wire [31:0]dinstOut;
    wire ewreg;
    wire em2reg;
    wire ewmem;
    wire [3:0]ealuc;
    wire ealuimm;
    wire [4:0]ern;
    wire [31:0]eqa;
    wire [31:0]eqb;
    wire [31:0]eimm32;

    wire mwreg;

    wire mm2reg;

    wire mwmem;

    wire [4:0]mrn;

    wire [31:0]mr;

    wire [31:0]mqb;

    wire wwreg;

    wire [4:0]wrn;

    wire wm2reg;

    wire [31:0]wr;

    wire [31:0]wdo;

    wire [31:0]r; // project

    wire [31:0]mdo; // project


    initial begin

        clock = 0;

    end


    datapath dataP(clock, pc, dinstOut, ewreg, em2reg, ewmem, ealuc, ealuimm, ern, eqa, eqb, eimm32, mwreg, mm2reg, mwmem, mrn, mr, mqb, wwreg, wrn,
wm2reg, wr, wdo, r, mdo);

    always begin

        #5

        clock = ~clock;

    end


endmodule
```

Datapath code:

```
`timescale 1ns / 1ps

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

// University: Penn State

// Engineer: David Rohweder

//

// Create Date: 03/21/2022 06:08:38 PM

// Design Name: 331 final project

// Module Name: datapath

// Project Name:

// Target Devices:

// Tool Versions:

// Description:
```

```
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module datapath(
   input 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]ern, // project
   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]mrn,// project
   output wire [31:0]mr,
   output wire [31:0]mqb,
   output wire wwreg,
   output wire [4:0]wrn, // project
   output wire wm2reg,
   output wire [31:0]wr,
   output wire [31:0]wdo,
   output wire [31:0]r, // project
   output wire [31:0]mdo // project
   );

   wire [31:0]nextPc;
   wire [31:0]instOut;
```

```
    programCounter pcc(nextPc, clock, pc);

    pcAddr pcA(pc, nextPc);

    instructionMemory instMem(pc, instOut);

    ifid_pipelineRegister ifidRP(instOut, clock, dinstOut);


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

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

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

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

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


    wire wreg;

    wire m2reg;

    wire wmem;

    wire [3:0]aluc;

    wire aluimm;

    wire regrt;

    wire [1:0]fwdb; // project

    wire [1:0]fwda; // project

    controlUnit contUnit(opcode, func, rs, rd, rt, mrn, mm2reg, mwreg, ern, em2reg, ewreg, wreg, m2reg, wmem, aluc, aluimm, regrt, fwdb, fwda);


    wire [31:0]wbData;

    wire [4:0]destReg;

    wire [31:0]qa;

    wire [31:0]qb;

    muxRegrt muxR(rt, rd, regrt, destReg);

    regFile regF(rs, rt, wrn, wbData, wwreg, ~clock, qa, qb);


    wire [31:0]fqa;

    wire [31:0]fqb;

    muxForwardA fdmxa(fwda, qa, r, mr, mdo, fqa);

    muxForwardB fdmxb(fwdb, qb, r, mr, mdo, fqb);


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

    wire [31:0]imm32;

    immediateExtender immediateExt(imm, imm32);

    idexe_pipelineRegister idexeRP(wreg, m2reg, wmem, aluc, aluimm, destReg, fqa, fqb, imm32, clock, ewreg, em2reg, ewmem, ealuc, ealuimm, ern, eqa, eqb,
eimm32);


    wire [31:0]b;
    aluMux i_aluMux(eqb, eimm32, ealuimm, b);

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

    exeMem i_exeMem(ewreg, em2reg, ewmem, ern, r, eqb, clock, mwreg, mm2reg, mwmem, mrn, mr, mqb);

    dataMemory i_dataMemory(mr, mqb, mwmem, clock, mdo);

    memWB i_memWB(mwreg, mm2reg, mrn, mr, mdo, clock, wwreg, wm2reg, wrn, wr, wdo);


    wbData wbMux(wr, wdo, wm2reg, wbData);


endmodule



// **************************************

// Region Project

// **************************************




module muxForwardA(

    input [1:0]fwda,

    input [31:0]qa,

    input [31:0]r,

    input [31:0]mr,

    input [31:0]mdo,

    output reg [31:0]fqa

    );


    always @(*) begin


        case (fwda)

            2'b00: begin

                fqa = qa;

            end


            2'b01: begin

                fqa = r;

            end


            2'b10: begin

                fqa = mr;

            end
            2'b11: begin
                fqa = mdo;
            end

        endcase
    end

endmodule



module muxForwardB(
    input [1:0]fwdb,
    input [31:0]qb,
    input [31:0]r,
    input [31:0]mr,
    input [31:0]mdo,
    output reg [31:0]fqb
    );

    always @(*) begin

        case (fwdb)
            2'b00: begin
                fqb = qb;
            end

            2'b01: begin
                fqb = r;
            end

            2'b10: begin
                fqb = mr;
            end

            2'b11: begin
                fqb = mdo;
            end

        endcase
```

```
   end

endmodule



// **************************************
// End Region Project
// **************************************



// **************************************
// Region Lab 5
// **************************************



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

   always  @(*) begin
      if (wm2reg == 0) begin
         wbData = wr;
      end else if (wm2reg == 1) begin
         wbData = wdo;
      end
   end

endmodule



// **************************************
// Region Lab 4
// **************************************



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

    always  @(*) begin
        if (ealuimm == 0) begin
            b = eqb;
        end else if (ealuimm == 1) begin
            b = eimm32;
        end
    end

endmodule



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

    always @(*) begin

        case (ealuc)
            4'b0010: begin
                r = eqa + b; // add
            end

            4'b0110: begin
                r = eqa - b; // sub
            end

            4'b0001: begin
                r = eqa | b; // or
            end

            4'b1001: begin
                r = eqa ^ b; // xor
            end

            4'b0000: begin
                r = eqa & b; // and
            end

        endcase
    end

endmodule


module exeMem(
    input ewreg,
    input em2reg,
    input ewmem,
    input [4:0]ern,
    input [31:0]r,
    input [31:0]eqb,
    input clock,
    output reg mwreg,
    output reg mm2reg,
    output reg mwmem,
    output reg [4:0]mrn,
    output reg [31:0]mr,
    output reg [31:0]mqb
    );

    always @(posedge clock) begin
        mwreg = ewreg;
        mm2reg = em2reg;
        mwmem = ewmem;
        mrn = ern;
        mr = r;
        mqb = eqb;
    end

endmodule
```

```
module dataMemory(
    input [31:0]mr,
    input [31:0]mqb,
    input mwmem,
    input clock,
    output reg [31:0]mdo
    );
    reg [31:0] memory[0:63];

    initial begin
        memory[0] = 32'h00000000;
        memory[1] = 32'hA00000AA;
        memory[2] = 32'h10000011;
        memory[3] = 32'h20000022;
        memory[4] = 32'h30000033;
        memory[5] = 32'h40000044;
        memory[6] = 32'h50000055;
        memory[7] = 32'h60000066;
        memory[8] = 32'h70000077;
        memory[9] = 32'h80000088;
        memory[10] = 32'h90000099;
    end

    always @(*) begin
        mdo = memory[mr[7:2]];
    end

    always @(negedge clock) begin
        if (mwmem == 1) begin
            memory[mr] = mqb;
        end
    end

endmodule


module memWB(
    input mwreg,
    input mm2reg,
    input [4:0]mrn,
    input [31:0]mr,
    input [31:0]mdo,
    input clock,
    output reg wwreg,
    output reg wm2reg,
    output reg [4:0]wrn,
    output reg [31:0]wr,
    output reg [31:0]wdo
    );

    always @(posedge clock) begin
        wwreg = mwreg;
        wm2reg = mm2reg;
        wrn = mrn;
        wr = mr;
        wdo = mdo;
    end

endmodule


// *************************************
// End Region Lab 4
// *************************************


// *************************************
// Region Lab 3
// *************************************


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

    initial begin
        pc <= 100;
    end
```

```
   // at pos edge of clock pc = nextPc
    always @(posedge clock) begin
      pc = nextPc;
    end

endmodule


// memory implemented as 2D reg arr, on any signal change instOut is set to value of meory array at pc pos
// implementation - 256 bytes or 64 words (4b / word): byte addr dif from word addr: pc = 100 == word addr 25
// only need to init memory where insturct is located -- words 25 and 26 ++ any signal change
module instructionMemory (
   input [31:0]pc,
   output reg [31:0]instOut
   );
   reg [31:0] memory[0:63];

   initial begin // format 6-bit opcode, 5-bit rs, 5-bit rt, 5-bit rd, 5-bit offset, 6-bit unused
      memory[25] = {6'b000000, 5'b00001, 5'b00010, 5'b00011, 5'b00000, 6'b100000}; // add $3, %1, $2
      memory[26] = {6'b000000, 5'b01001, 5'b00011, 5'b00100, 5'b00000, 6'b100010}; // sub $4, %9, $3
      memory[27] = {6'b000000, 5'b00011, 5'b01001, 5'b00101, 5'b00000, 6'b100101}; // or $5, %3, $9
      memory[28] = {6'b000000, 5'b00011, 5'b01001, 5'b00110, 5'b00000, 6'b100110}; // xor $6, %3, $9
      memory[29] = {6'b000000, 5'b00011, 5'b01001, 5'b00111, 5'b00000, 6'b100100}; // and $7, %3, $9
   end

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

endmodule


// on any signal change
module pcAddr (
   input [31:0]pc,
   output reg [31:0]nextPc
   );
   reg [31:0]hardwired;
```

```
    initial  begin
        hardwired = 32'b00000000000000000000000000000100;
    end

    always @(*) begin
        nextPc = pc + hardwired;
    end

endmodule



// on pos edge
module ifid_pipelineRegister (
    input [31:0]instOut,
    input clock,
    output reg [31:0]dinstOut
    );

    always @ (posedge clock) begin
        dinstOut = instOut;
    end

endmodule



module controlUnit (
    input [5:0]opcode, // bits 31:26 of dinstOut -- 6 bits
    input [5:0]func, // bits 5:0 of dinstOut -- 6 bits
    input [4:0]rs,
    input [4:0]rd,
    input [4:0]rt,
    input [4:0]mrn,
    input mm2reg,
    input mwreg,
    input [4:0]ern,
    input em2reg,
    input ewreg,
    output reg wreg, // write reg - when 1 means we write to 1 of 32 registers in regfile
    output reg m2reg, // mem to reg -- when 1 means we are loading from mem to reg in regfile
    output reg wmem, // write mem -- when 1 means we are write to data mem
```

```
  output reg [3:0]aluc, // 4 bits - op we intend to exe on ALu

  output reg aluimm, // alu immediate when 1, indicates we want to use sign extended im over qb in ALU, if we want qb aluimm = 0

  output reg regrt, // reg rt - when 1 indicates we are writing to rt not rd in regfile. if rd regrt - 0. regrt -> dont care val if wreg = 0

  output reg [1:0]fwdb,

  output reg [1:0]fwda

  );


  always @(*)begin


    if (mm2reg == 1 || mwreg == 1 || em2reg == 1 || ewreg == 1) begin // does hazard potentially exist || if em2reg = 1 then we stall bc we have not read
mdo yet

      if (ern == rs || ern == rt || mrn == rs || mrn == rt) begin // does any of the registers have a hazard


        ///

        if (ern == rs || ern == rt) begin


          if (ern == rs) begin


            fwda = 2'b01; // load qa = r

            fwdb = 2'b00; // load qb = qb


          end


          if (ern == rt) begin


            fwda = 2'b00; // load qa = qa

            fwdb = 2'b01; // load qb = r


          end


        end

        ///


        ///

        if (mrn == rs || mrn == rt) begin


          if (mrn == rs) begin


            fwda = 2'b10; // load qa = mr

            fwdb = 2'b00; // load qb = qb
        end

        if (mrn == rt) begin

          fwda = 2'b00; // load qa = qa
          fwdb = 2'b10; // load qb = mr

        end

      end
      ///

      // chefs note: since there is a not clock on reg file that on neg edge trigger values are read so if a write happens on pos then no hazzard occurs..

    end
  end else begin
    fwda = 2'b00; // load qa = qa
    fwdb = 2'b00; // load qa = qb
  end

  case (opcode)
    6'b000000: begin // r-type

      case(func)
        6'b100000: begin  // add func
          wreg = 1;
          m2reg = 0;
          wmem = 0;
          aluimm = 0;
          regrt = 0;
          aluc = 4'b0010;
        end

        6'b100010: begin  // sub
          wreg = 1;
          m2reg = 0;
          wmem = 0;
          aluimm = 0;
          regrt = 0;
```

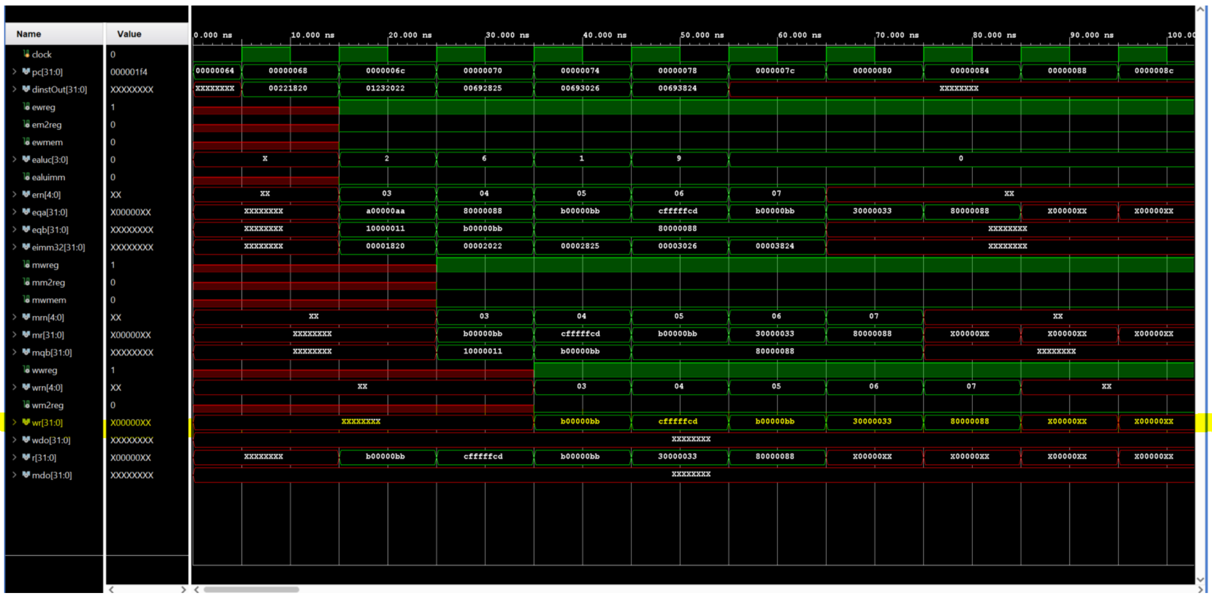
```
            aluc = 4'b0110;
        end

        6'b100101: begin  // or
            wreg = 1;
            m2reg = 0;
            wmem = 0;
            aluimm = 0;
            regrt = 0;
            aluc = 4'b0001;
        end

        6'b100110: begin  // xor func
            wreg = 1;
            m2reg = 0;
            wmem = 0;
            aluimm = 0;
            regrt = 0;
            aluc = 4'b1001;
        end

        6'b100100: begin  // and func
            wreg = 1;
            m2reg = 0;
            wmem = 0;
            aluimm = 0;
            regrt = 0;
            aluc = 4'b0000;
        end

    endcase
end

6'b100011: begin // I-type and lw
    wreg = 1; // write register
    m2reg = 1; // memory to register
    wmem = 0; // write memory
    aluimm = 1; // alu immediate - means we want to use sign extended immediate over value of qb in alu, if we want to use qb, aluimm should be 0
    regrt = 1;
    aluc = 4'b0010; // add offset ;)
```

```
            end

        endcase

    end

endmodule



module muxRegrt (
    input [4:0]rt, // bits (20:16) of dinstOut
    input [4:0]rd, // bits 15:11 of dinstOut
    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 regFile (
    input [4:0]rs, // bits [25:21] dinstOut
    input [4:0]rt, // bits [20:16] of dinstOut
    input [4:0]wrn,
    input [31:0]wbData,
    input wwreg,
    input clock,
    output reg [31:0]qa,
    output reg [31:0]qb
    );
    reg [31:0] registers [0:31];
    integer index; // needs to be out of loop or verilog err
    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 (index = 11; index < 32; index = index + 1) begin
            registers[index] = 32'b00000000000000000000000000000000;
        end

    end

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

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

endmodule



// imm32 is set to sign extended value of imm
module immediateExtender (
    input [15:0]imm,
    output reg [31:0]imm32
    );

    always @(*) begin
```

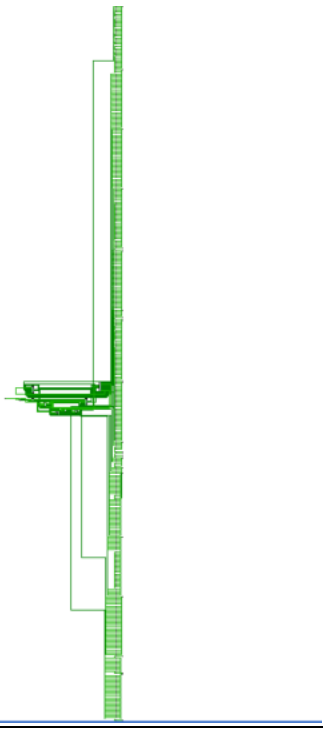
```
        imm32 <= { {16{imm[15]}}, imm[15:0]}; // note to self verilog pg 60 rev 1
    end

endmodule


module idexe_pipelineRegister(
    input wreg,
    input m2reg,
    input wmem,
    input [3:0]aluc,
    input aluimm,
    input [4:0]destReg, // 5 bits
    input [31:0]fqa,
    input [31:0]fqb,
    input [31:0]imm32,
    input clock,
    output reg ewreg,
    output reg em2reg,
    output reg ewmem,
    output reg [3:0]ealuc,
    output reg ealuimm,
    output reg [4:0]ern,
    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;
        ern = destReg;
        eqa = fqa;
        eqb = fqb;
        eimm32 = imm32;
    end
```

```
endmodule



// **************************************

// End Region Lab 3

// **************************************
```

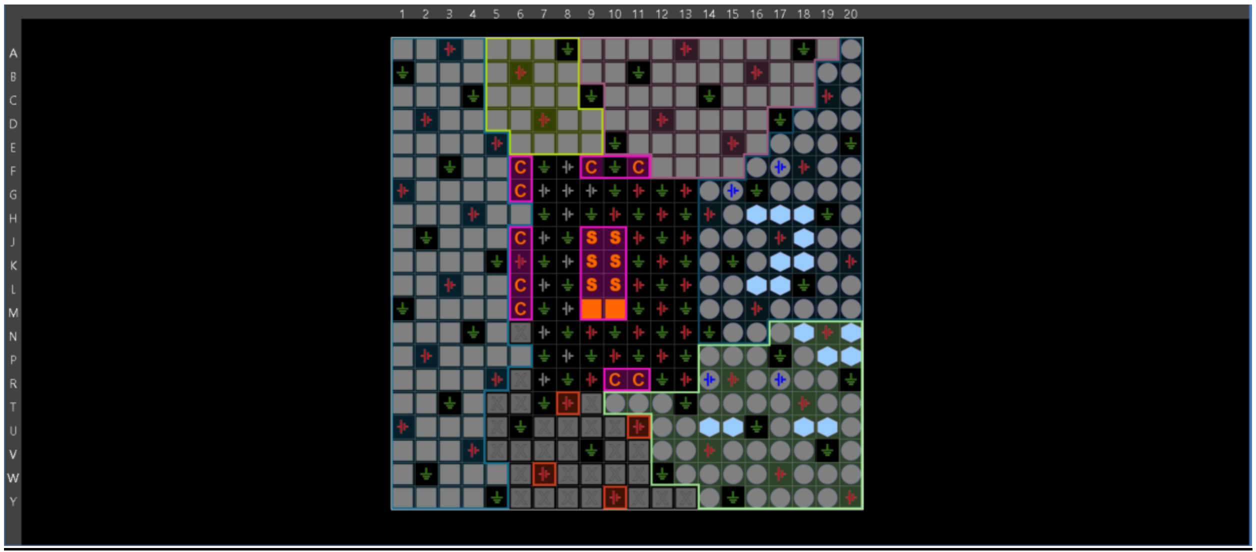
Waveforms:



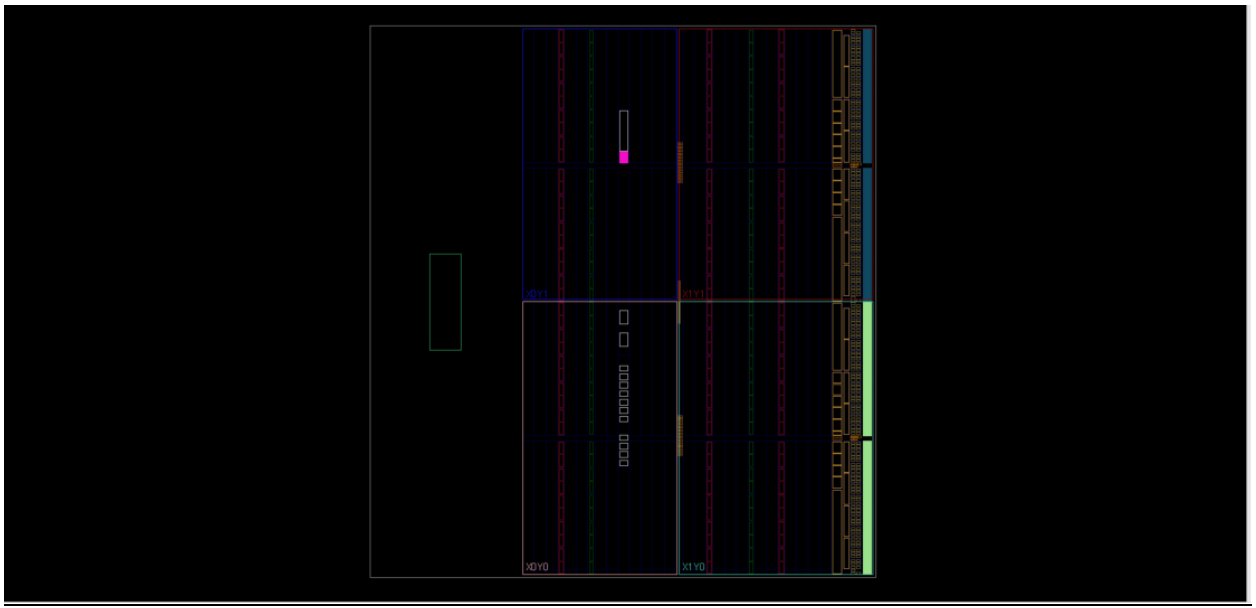
Design Schematics:



I/O Planning:



Floor Planning:


X0Y1
X1Y1
X0Y0
X1Y0