**Final project- single cycle CPU**

**Compen331 Section002**

**Yiyi Wang**

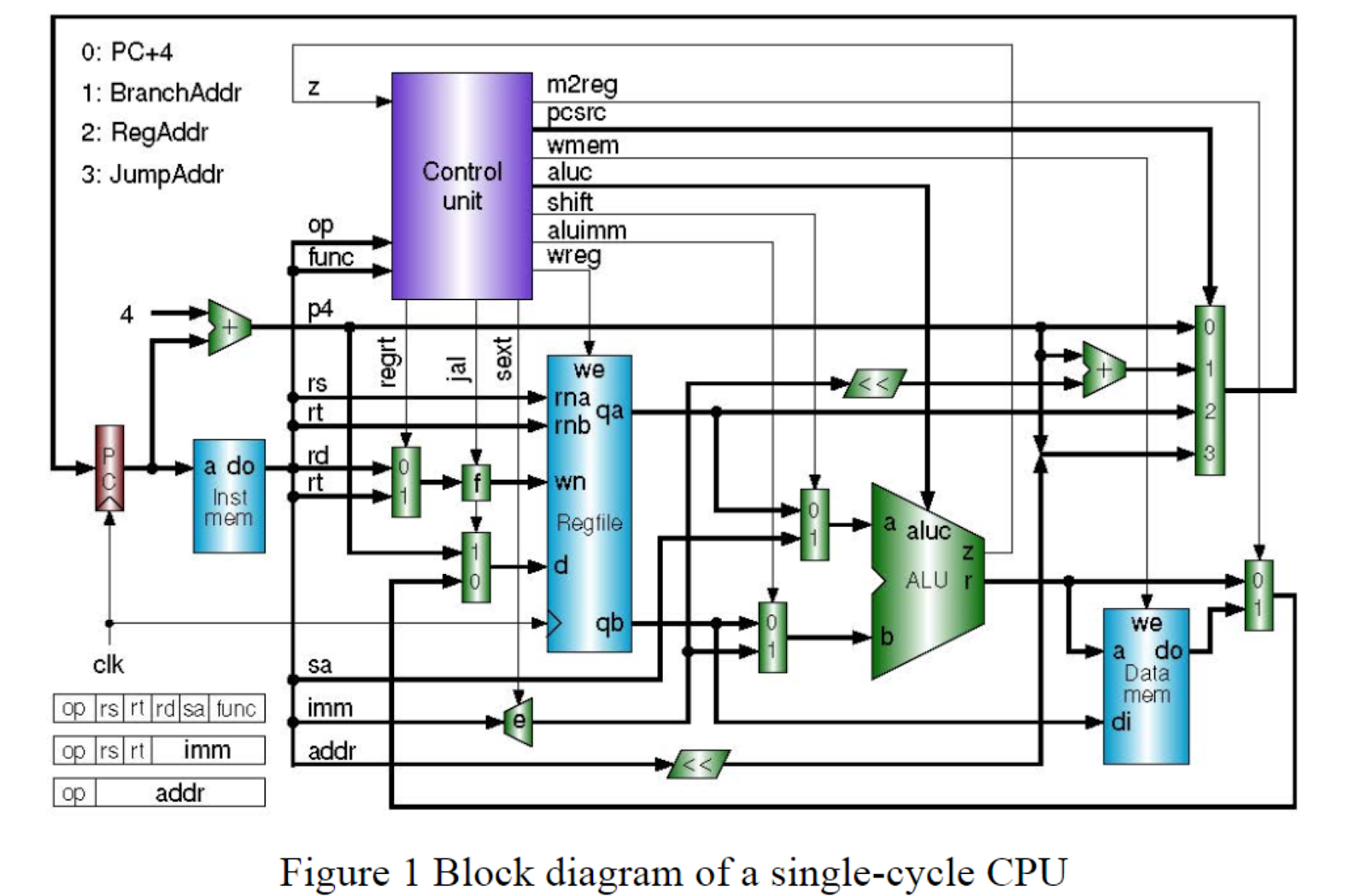
**Xiaodong Huo**

**4/28/18**

**Abstract**

This project is a summery to previous labs. This concludes contents from lab4,lab5 and lab6. Instruction that can execute includes instruction fetch, instruction execution, R-format instruction, I-format instruction, Branch instructions, Jump instructions, and shift instructions. Then we need to generate bit-stream and connect to board. When all the instructions are done(pc=h5c), led should light up.

**Introduction**



There are five stages including instruction fetch, decode, execution, memory access and writeback.

In the instruction fetch state, a **program counter** reads the address of the instruction and then access the **Instmem**. The value of the program counter is determined by the 2bit **pcsrc.** Determined by the instruction type.Then a 32 bit instruction is decomposed based on the instruction type.

In the decode stage, op code and function code is sent to the **control** unit, Rs and Rt is send to the **Regfile.** A mux determines the the destination register number rt or rd base on the type of instruction(rt if regrt = 1). A mux **f** determines the j-type instructions takes the jump address. If jal is a 1, a 5 bit patern 11111 (decimal 31) will be assigned to the destination register number wn. Otherwise, reg\_dest, which is rd or rt, will be assigned to wn. The mux below the **f** mux determines the destination address, it will be either Program counter+4 or the alu output memory or the output of the data memory. When the instruction is branch type, a **imm** address will be sign extended and the immediate is sign extended

(if sext  = 1), after the sign extension, a shift left logic with shift the address to the left and add whatever on the PC+4. Input 3 is the jump target address coming from addr and PC+4.

The output of regfile is qa and qb correspond to rna and rnb.

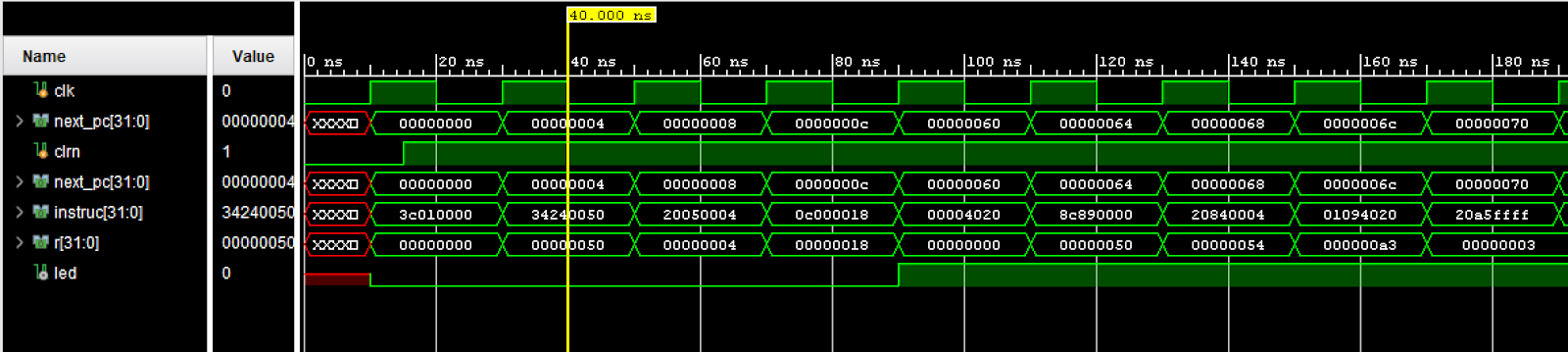
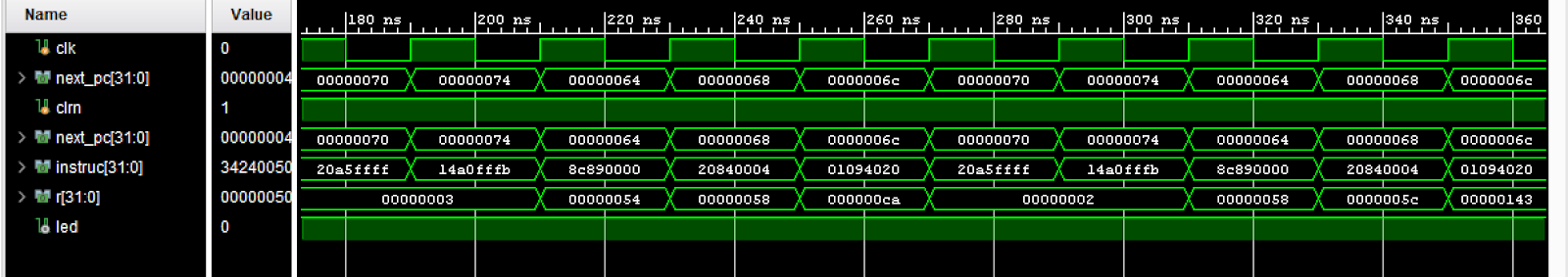
In the excution state, two mux determines one operand of the addition comes from register rs of the register file (shift = 0); the other operand is the immediate (aluimm = 1) for immediate operations. In the alu, the alu control determines the type of operations excuted. It can be add, subtract, and, or, xor, lui, sll,srl, sra based on the 4bit **aluc** code.

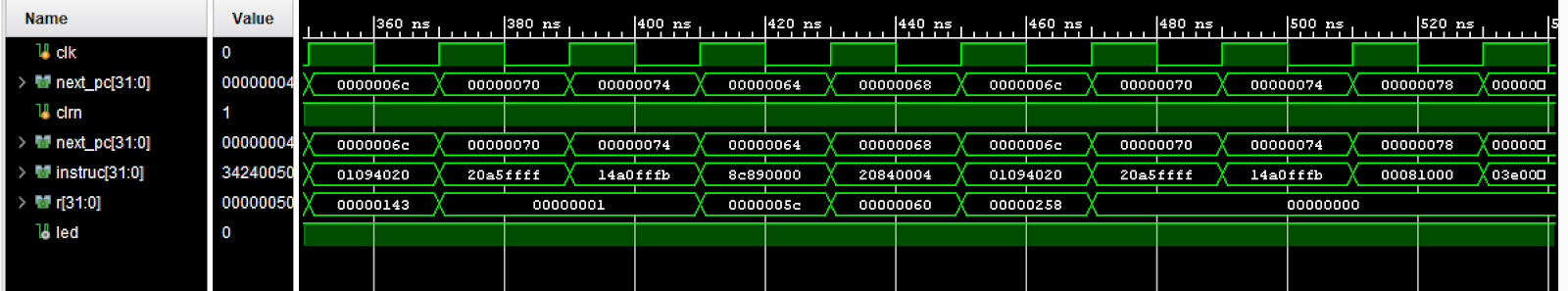
In the Memory access stage, **we** determines the write enable, whether to write in the memory or not.

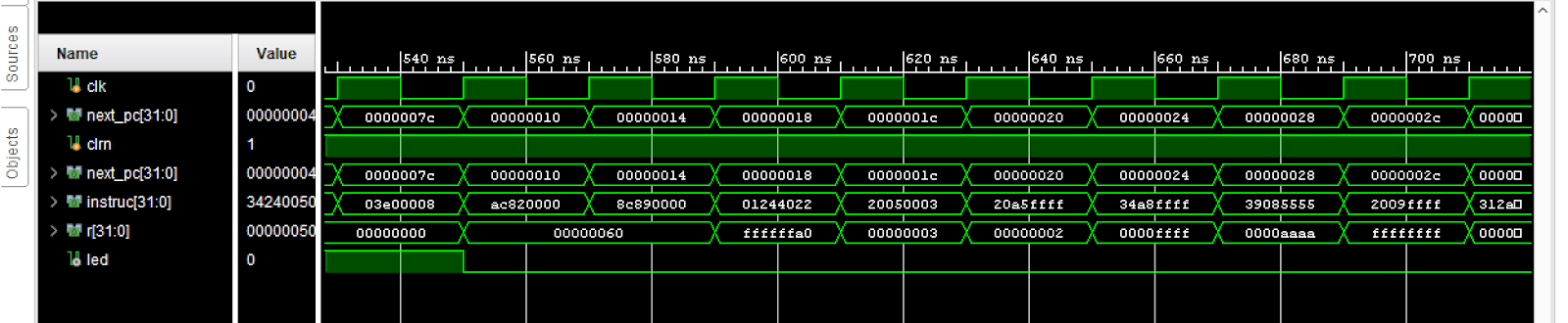
In the write back stage, the wreg singla controls whether to write back to the **regfile** or not.

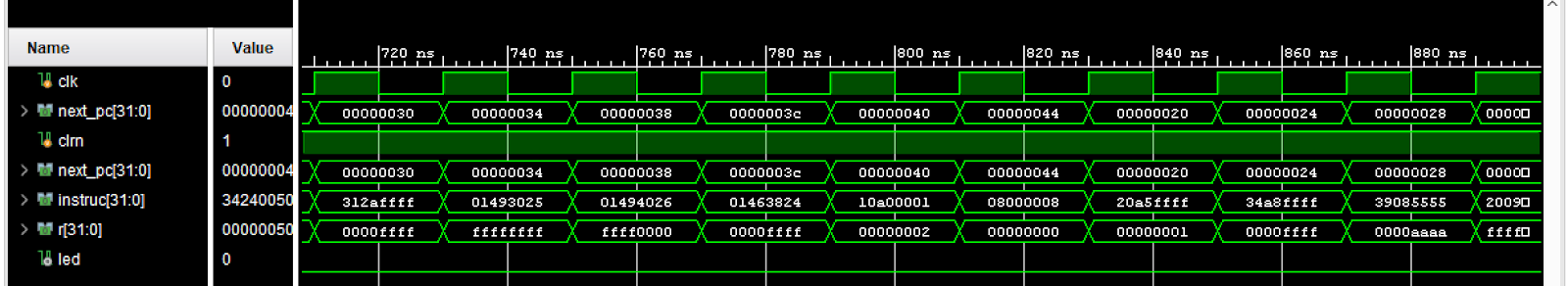
The benefits of our design is we separated the data memory and instruction memory so that we can read and write from the memory at the same time.(read at positive edge and write at negative edge)

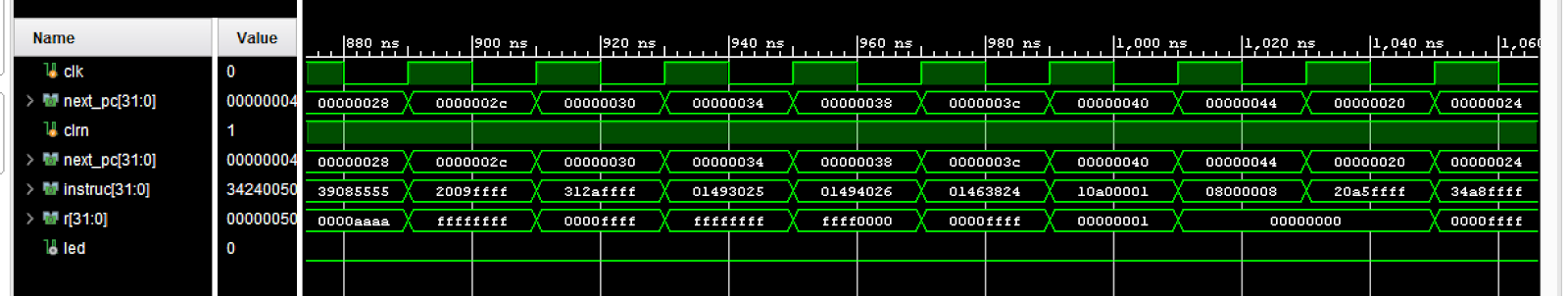
**RTL analysis**

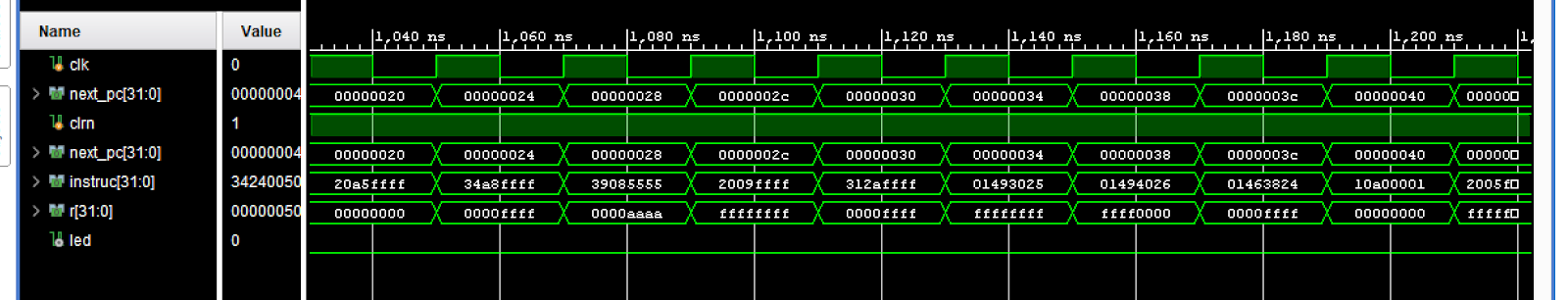
****

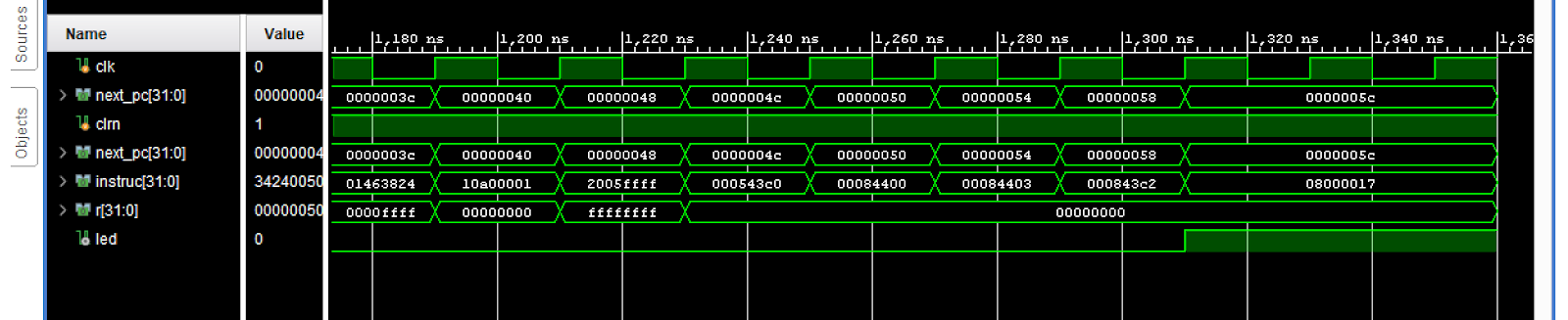
****

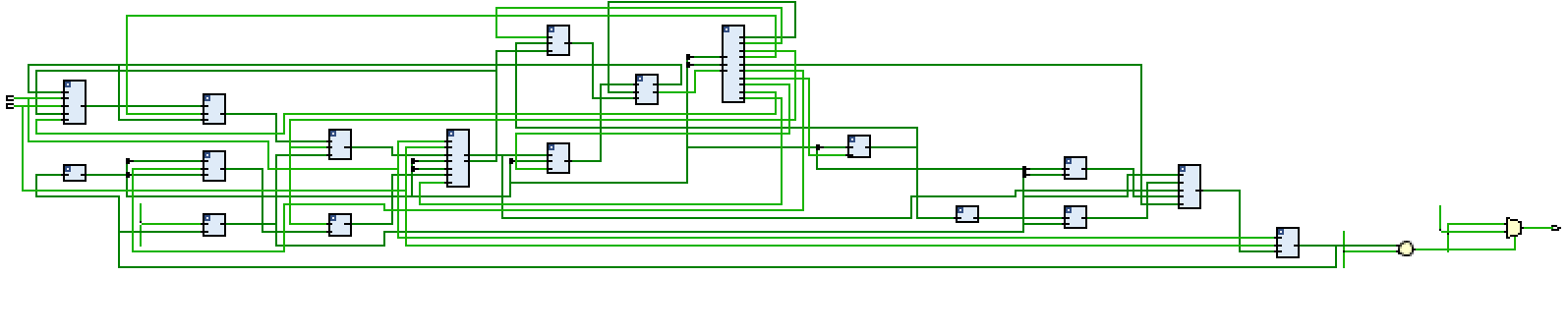
****

****

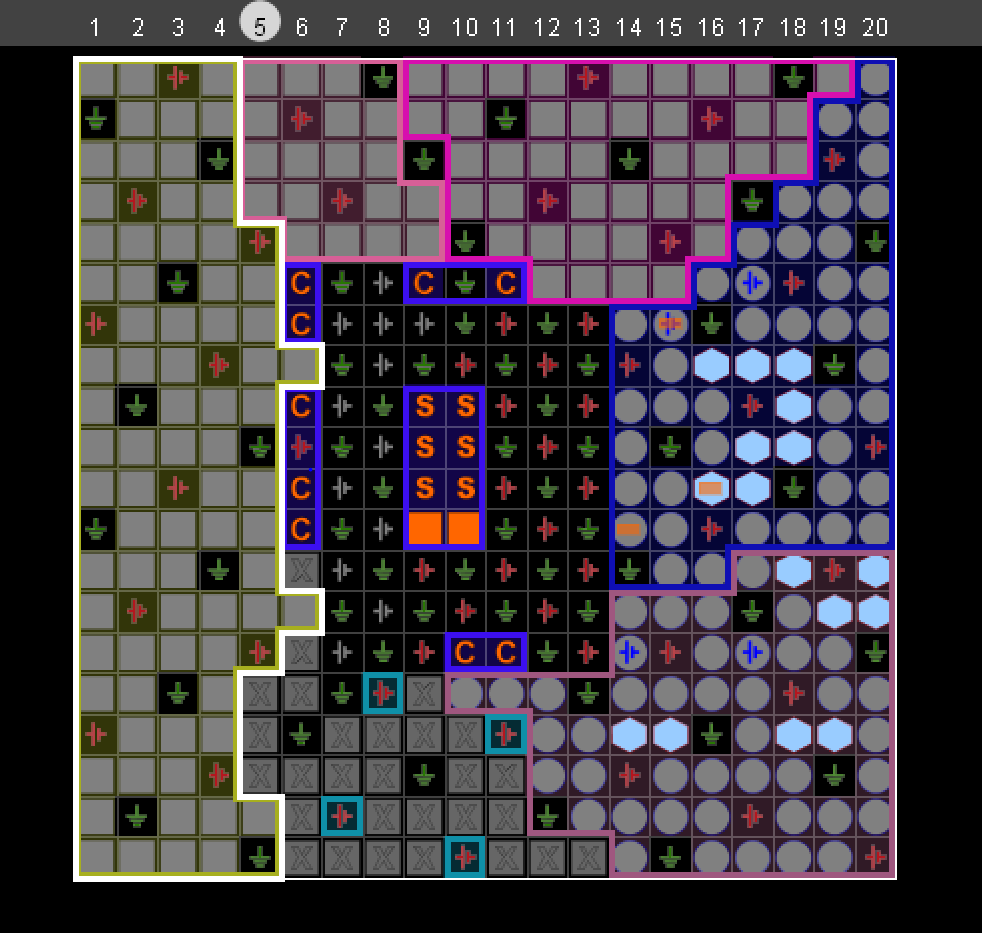
****

****

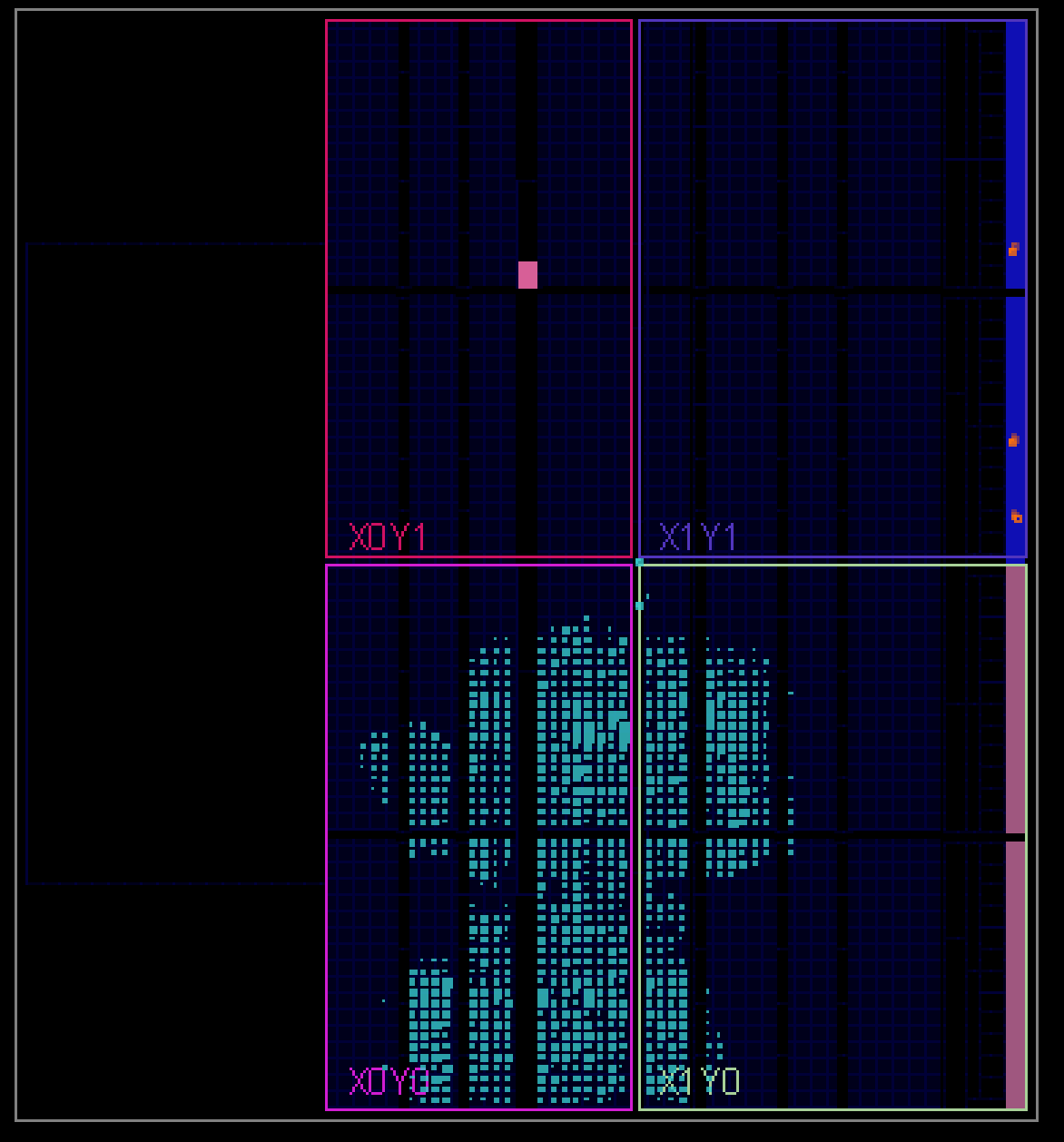
****



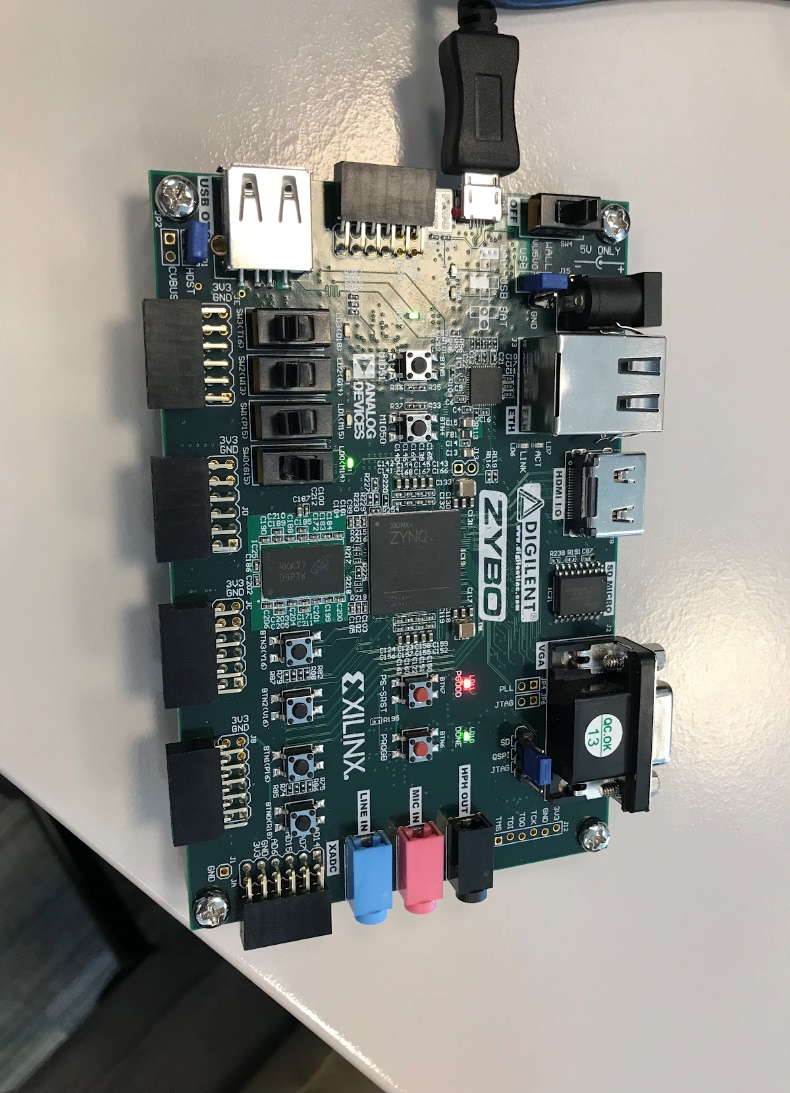
**I/O planning**



**Floor Planning**



**Zybo**

****

**TOP Execute**

`timescale 1ns / 1ps

module execute(clk, clrn, led);

   input clk, clrn;

   wire [31:0] instruc;

   wire [31:0] r;

   wire [31:0] pc, next\_pc;

   wire m2reg, wmem, shift, aluimm, wreg, regrt, sext,z;

   wire [3:0] aluc;

   wire [1:0] pcsrc;

   wire [4:0] wn, wn\_new;

   wire [31:0] d, d\_new, do, qa, qb, a, b, imm\_exp, imm\_exp\_shift, addr\_shift, p1, p2, addr\_exp, jal;

   output led;

   pc PC(clk, clrn, pc,next\_pc);

   read\_instruc readInstruc(next\_pc, instruc);

   pc\_adder p1Adder(next\_pc,32'd4, p1);

   addr\_e addrE(instruc[25:0], p1[31:28], addr\_shift);

   imm\_e immE(sext, instruc[15:0], imm\_exp);

   shift\_left immShift(imm\_exp, imm\_exp\_shift);

   pc\_adder p2Adder(p1, imm\_exp\_shift, p2);

   control\_unit controlUnit(instruc[31:26], instruc[5:0],z, m2reg, wmem, aluimm, pcsrc, aluc, shift, wreg, regrt, sext, jal);

   regrt\_mux regrtMux(regrt, instruc[15:11], instruc[20:16], wn);

   wn\_mux wnMux(jal, wn, wn\_new);

   d\_mux dMux(jal,d, p1, d\_new);

   regfile regFile(clk, clrn, wreg, instruc[25:21],instruc[20:16], wn\_new, d\_new, qa, qb);

   pc\_mux pcMux(pcsrc, p1, p2, qa, addr\_shift, pc);

   shift\_mux shiftMux(shift, qa, instruc[10:6], a);

   qb\_imm\_mux qbImmMux(aluimm, qb, imm\_exp, b);

   ALU alu(aluc, a, b, r, z);

   data\_mem dataMem(clk, clrn, wmem, r, qb, do);

   m2reg\_mux m2regMux(m2reg, r, do, d);

   //assign led = !(next\_pc == pc)? 0:1;

   assign led = !(next\_pc >= 32'h0000005c) ? 0:1;

   //assign led = clrn;

endmodule

**Test Bench**

`timescale 1ns / 1ps

module test\_bench;

   reg clk = 0;

   reg clrn;

   wire led;

   execute PCexecute(clk, clrn,led);

   always #10 clk = ~clk;

   initial begin

   #0 clk = 0;

      clrn = 0;

   #15 clrn = 1;

   #1350 $finish;

   end

endmodule

**PC**

`timescale 1ns / 1ps

module pc(clk, clrn, input\_pc, next\_pc);

   input clk;

   input clrn;

   input [31:0] input\_pc;

   output reg [31:0] next\_pc;

   //initial begin

   //  next\_pc <= 32'b0;

   //end

   always @(posedge clk) begin

       if (clrn) begin

           next\_pc <= input\_pc;

       end

       else begin

           next\_pc <= 32'h00000000;

       end

   end

endmodule

**Read instruction**

`timescale 1ns / 1ps

module read\_instruc(pc, instruc);

   input [31:0] pc;

   output reg [31:0] instruc;

   reg [31:0] instruc\_mem [0:31];

   initial begin

       instruc\_mem[0] <= 32'h3c010000;

       instruc\_mem[1] <= 32'h34240050;

       instruc\_mem[2] <= 32'h20050004;

       instruc\_mem[3] <= 32'h0c000018;

       instruc\_mem[4] <= 32'hac820000;

       instruc\_mem[5] <= 32'h8c890000;

       instruc\_mem[6] <= 32'h01244022;

       instruc\_mem[7] <= 32'h20050003;

       instruc\_mem[8] <= 32'h20a5ffff;

       instruc\_mem[9] <= 32'h34a8ffff;

       instruc\_mem[10] <= 32'h39085555;

       instruc\_mem[11] <= 32'h2009ffff;

       instruc\_mem[12] <= 32'h312affff;

       instruc\_mem[13] <= 32'h01493025;

       instruc\_mem[14] <= 32'h01494026;

       instruc\_mem[15] <= 32'h01463824;

       instruc\_mem[16] <= 32'h10a00001;

       instruc\_mem[17] <= 32'h08000008;

       instruc\_mem[18] <= 32'h2005ffff;

       instruc\_mem[19] <= 32'h000543c0;

       instruc\_mem[20] <= 32'h00084400;

       instruc\_mem[21] <= 32'h00084403;

       instruc\_mem[22] <= 32'h000843c2;

       instruc\_mem[23] <= 32'h08000017;

       instruc\_mem[24] <= 32'h00004020;

       instruc\_mem[25] <= 32'h8c890000;

       instruc\_mem[26] <= 32'h20840004;

       instruc\_mem[27] <= 32'h01094020;

       instruc\_mem[28] <= 32'h20a5ffff;

       instruc\_mem[29] <= 32'h14a0fffb;

       instruc\_mem[30] <= 32'h00081000;

       instruc\_mem[31] <= 32'h03e00008;

   end

   always @(\*) begin

       instruc <= instruc\_mem[pc>>2];

   end

endmodule

**P1 Adder**

`timescale 1ns / 1ps

module pc\_adder(current\_pc, adder, next\_pc);

   input [31:0] current\_pc;

   input [31:0] adder;

   output reg [31:0] next\_pc;

   always @(\*) begin

       next\_pc <= current\_pc + adder;

   end

endmodule

**Adder**

`timescale 1ns / 1ps

module addr\_e(addr, pc, addr\_exp);

   input [25:0] addr;

   input [3:0] pc;

   output [31:0] addr\_exp;

   assign addr\_exp[31:28] = pc;

   assign addr\_exp[27:2] = addr;

   assign addr\_exp[1:0] = 2'b00;

endmodule

**Imme**

`timescale 1ns / 1ps

module imm\_e(sext, imm, imm\_exp);

   input sext;

   input [15:0] imm;

   output reg [31:0] imm\_exp;

   always @(\*) begin

       if (sext == 1) begin

           if (imm[15] == 0) begin

               imm\_exp[31:16] <= 16'b0;

               imm\_exp[15:0] <= imm;

           end

           else begin

               imm\_exp[31:16] <= 16'hffff;

               imm\_exp[15:0] <= imm;

           end

       end

       else begin

           imm\_exp[31:16] <= 16'b0;

           imm\_exp[15:0] <= imm;

       end

   end

endmodule

**Immshift**

`timescale 1ns / 1ps

module shift\_left(data\_in, data\_out);

   input [31:0] data\_in;

   output reg [31:0] data\_out;

   always @(\*) begin

       data\_out <= (data\_in<<2) & 32'hffffffff;

   end

endmodule

**P2 adder**

   `timescale 1ns / 1ps

   module pc\_adder(current\_pc, adder, next\_pc);

       input [31:0] current\_pc;

       input [31:0] adder;

       output reg [31:0] next\_pc;

       always @(\*) begin

           next\_pc <= current\_pc + adder;

       end

   endmodule

**Control unit**

`timescale 1ns / 1ps

module control\_unit(op, func, z, m2reg, wmem, aluimm, pcsrc, aluc, shift, wreg, regrt, sext, jal);

   input [5:0] op;

   input [5:0] func;

   input z;

   output reg m2reg;

   output reg wmem;

   output reg aluimm;

   output reg [1:0] pcsrc;

   output reg [3:0] aluc;

   output reg shift;

   output reg wreg;

   output reg regrt;

   output reg sext;

   output reg jal;

   always @(\*) begin

       if (op == 6'b000000)

       begin

           if (func == 6'b100000)

           begin          // add

                pcsrc <= 2'b00;

                aluc <= 4'bx000;

                shift <= 0;

                wreg <= 1;

                m2reg <= 0;

                wmem <= 0;

                aluimm <= 0;

                sext <= 0;

                regrt <= 0;

                jal <= 0;

           end

           else if (func == 6'b100010) begin // subtract

                pcsrc <= 2'b00;

                aluc <= 4'bx100;

                shift <= 0;

                wreg <= 1;

                m2reg <= 0;

                wmem <= 0;

                aluimm <= 0;

                sext <= 0;

                regrt <= 0;

                jal <= 0;

           end

           else if (func == 6'b100100) begin // and

                pcsrc <= 2'b00;

                aluc <= 4'bx001;

                shift <= 0;

                wreg <= 1;

                m2reg <= 0;

                wmem <= 0;

                aluimm <= 0;

                sext <= 0;

                regrt <= 0;

                jal <= 0;

           end

           else if (func == 6'b100101) begin // or

                pcsrc <= 2'b00;

                aluc <= 4'bx101;

                shift <= 0;

                wreg <= 1;

                m2reg <= 0;

                wmem <= 0;

                aluimm <= 0;

                sext <= 0;

                regrt <= 0;

                jal <= 0;

           end

           else if (func == 6'b100110) begin // xor

                pcsrc = 2'b00;

                aluc = 4'bx010;

                shift = 0;

                wreg = 0;

                m2reg <= 0;

                wmem <= 0;

                aluimm <= 0;

                sext <= 0;

                regrt <= 0;

                jal <= 0;

           end

           else if (func == 6'b000000) begin  // sll

                pcsrc <= 2'b00;

                aluc <= 4'b0011;

                shift <= 1;

                wreg <= 1;

                m2reg <= 0;

                wmem <= 0;

                aluimm <= 0;

                sext <= 0;

                regrt <= 0;

                jal <= 0;

           end

           else if (func == 6'b000010) begin // srl

                pcsrc <= 2'b00;

                aluc <= 4'b0111;

                shift <= 1;

                wreg <= 1;

                m2reg <= 0;

                wmem <= 0;

                aluimm <= 0;

                regrt <= 0;

                jal <= 0;

           end

           else if (func == 6'b000011) begin  // sra

               pcsrc <= 2'b00;

               aluc <= 4'b1111;

               shift <= 1;

               wreg <= 1;

               m2reg <= 0;

               wmem <= 0;

               aluimm <= 0;

               regrt <= 0;

               jal <= 0;

           end

           else if (func == 6'b001000) begin  //jr

               pcsrc <= 2'b10;

               wmem <= 0;

               wreg <= 0;

           end

       end

       else if (op == 6'b001000) begin  // addi

           pcsrc[1:0] <= 2'b00;

           aluc <= 4'bx000;

           shift <= 0;

           wreg <= 1;

           m2reg <= 0;

           wmem <= 0;

           aluimm <= 1;

           sext <= 1;

           regrt <= 1;

           jal <= 0;

       end

       else if (op == 6'b001100) begin  // andi

           pcsrc[1:0] <= 2'b00;

           aluc <= 4'bx001;

           shift <= 0;

           wreg <= 1;

           m2reg <= 0;

           wmem <= 0;

           aluimm <= 1;

           sext <= 0;

           regrt <= 1;

           jal <= 0;

       end

       else if (op == 6'b001101) begin  // ori

           pcsrc[1:0] <= 2'b00;

           aluc <= 4'bx101;

           shift <= 0;

           wreg <= 1;

           m2reg <= 0;

           wmem <= 0;

           aluimm <= 1;

           sext <= 0;

           regrt <= 1;

           jal <= 0;

       end

       else if (op == 6'b001110) begin  // xori

           pcsrc[1:0] <= 2'b00;

           aluc <= 4'bx010;

           shift <= 0;

           wreg <= 1;

           m2reg <= 0;

           wmem <= 0;

           aluimm <= 1;

           sext <= 0;

           regrt <= 1;

           jal <= 0;

       end

       else if (op == 6'b100011) begin  // lw

           pcsrc[1:0] <= 2'b00;

           aluc <= 4'bx000;

           shift <= 0;

           wreg <= 1;

           m2reg <= 1;

           wmem <= 0;

           aluimm <= 1;

           sext <= 1;

           regrt <= 1;

           jal <= 0;

       end

       else if (op == 6'b101011) begin  // sw

           pcsrc[1:0] <= 2'b00;

           aluc<= 4'bx000;

           shift <= 0;

           wreg <= 0;

           wmem <= 1;

           aluimm <= 1;

           sext <= 1;

       end

       else if (op == 6'b000100) begin  // beq

           if (z == 0) begin  // z = 0

               pcsrc[1:0] <= 2'b00;

               aluc <= 4'bx010;

               shift <= 0;

               wreg <= 0;

               wmem <= 0;

               aluimm <= 0;

               sext <= 1;

           end

           else begin // z = 1

               pcsrc[1:0] <= 2'b01;

               aluc <= 4'bx010;

               shift <= 0;

               wreg <= 0;

               wmem <= 0;

               aluimm <= 0;

               sext <= 1;

           end

       end

       else if (op == 6'b000101) begin  // bne

           if (z == 0) begin  // z = 0

               pcsrc[1:0] <= 2'b01;

               aluc <= 4'bx010;

               shift <= 0;

               wreg <= 0;

               wmem <= 0;

               aluimm <= 0;

               sext <= 1;

           end

           else begin // z = 1

               pcsrc[1:0] <= 2'b00;

               aluc <= 4'bx010;

               shift <= 0;

               wreg <= 0;

               wmem <= 0;

               aluimm <= 0;

               sext <= 1;

           end

       end

       else if (op == 6'b001111) begin  // lui

           pcsrc[1:0] <= 2'b00;

           aluc <= 4'bx110;

           wreg <= 1;

           m2reg <= 0;

           wmem <= 0;

           aluimm <= 1;

           regrt <= 1;

           jal <= 0;

       end

       else if (op == 6'b000010) begin  // j

           pcsrc[1:0] <= 2'b11;

           wreg <= 0;

           wmem <= 0;

       end

       else if (op == 6'b000011) begin  // jal

           pcsrc[1:0] <= 2'b11;

           wreg <= 1;

           wmem <= 0;

           jal <= 1;

       end

   end

endmodule

`timescale 1ns / 1ps

**RegrtMux**

module regrt\_mux(regrt, rd, rt, wn);

   input regrt;

   input [4:0] rd;

   input [4:0] rt;

   output reg [4:0] wn;

   always @(\*) begin

       if (regrt == 1)

           wn <= rt;

       else

           wn <= rd;

   end

endmodule

**wnMux**

`timescale 1ns / 1ps

module wn\_mux(jal, wn\_in, wn\_out);

   input jal;

   input [4:0] wn\_in;

   output reg [4:0] wn\_out;

   always @(\*) begin

       if (jal == 0)

           wn\_out <= wn\_in;

       else

           wn\_out <= 5'b11111;

   end

endmodule

**Dmux**

`timescale 1ns / 1ps

module d\_mux(jal, d\_in, pcPlus4, d\_out);

   input jal;

   input [31:0] d\_in, pcPlus4;

   output reg [31:0] d\_out;

   always @(\*) begin

       if (jal == 0)

           d\_out <= d\_in;

       else

           d\_out <= pcPlus4;

   end

endmodule

**Refile**

`timescale 1ns / 1ps

module regfile(clk,clrn, wreg, rna, rnb, wn, d, qa, qb);

   input clk;

   input clrn;

   input wreg;

   input [4:0] rna;

   input [4:0] rnb;

   input [4:0] wn;

   input [31:0] d;

   output [31:0] qa, qb;

   reg[31:0] register [0:31];

   assign qa = (rna == 0)? 0:register[rna];

   assign qb = (rna == 0)? 0:register[rnb];

   integer i;

   //initial begin

   //  for (i=0; i<32; i=i+1)

   //      register[i] <= 32'b0;

   //  end

   always @(posedge clk) begin

       if (clrn == 0)begin

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

               register[i] <= 32'b0;

           end

       end

       if ((wn != 0) && (wreg == 1))

           register[wn] <= d;

   end

endmodule

**PC mux**

`timescale 1ns / 1ps

module pc\_mux(pcsrc, pc\_1, pc\_2, pc\_3, pc\_4, next\_pc);

   input [1:0] pcsrc;

   input [31:0] pc\_1;

   input [31:0] pc\_2;

   input [31:0] pc\_3;

   input [31:0] pc\_4;

   output reg[31:0] next\_pc;

   always @(\*) begin

       case (pcsrc)

           'b00: next\_pc = pc\_1;

           'b01: next\_pc = pc\_2;

           'b10: next\_pc = pc\_3;

           'b11: next\_pc = pc\_4;

       endcase

   end

endmodule

**Shift Mux**

`timescale 1ns / 1ps

module shift\_mux(shift, qa, sa, a);

   input shift;

   input [31:0] qa;

   input [4:0] sa;

   output reg [31:0] a;

   always @(qa or sa or shift)

   begin

       if (shift)

       begin

           a[31:5] <= 27'b0;

           a[4:0] <= sa;

       end

       else

           a <= qa;

   end

endmodule

**qblmmMux**

`timescale 1ns / 1ps

module qb\_imm\_mux(aluimm, qb, imm, b);

   input aluimm;

   input [31:0] qb;

   input [31:0] imm;

   output reg [31:0] b;

   always @(\*) begin

       if (aluimm == 1)

           b <= imm;

       else

           b <= qb;

   end

endmodule

**ALU**

`timescale 1ns / 1ps

module ALU(aluc, a, b, r, z);

   input [3:0] aluc;

   input [31:0] a;

   input [31:0] b;

   output reg [31:0] r;

   output reg z;

   integer i;

   reg [31:0] x;

   always @(aluc, a, b)

   begin

       if (aluc[2:0] == 3'b000) // add

           r <= a + b;

       else if (aluc[2:0] == 3'b100) //sub

           r <= a - b;

       else if (aluc[2:0] == 3'b001) //and

           r <= a & b;

       else if (aluc[2:0] == 3'b101) // or

           r <= a | b;

       else if (aluc[2:0] == 3'b010) begin //xor

           r <= a ^ b;

           if (a == b)

               z <= 1;

           else

               z <= 0;

       end

       else if (aluc[2:0] == 3'b110) begin // lui

           r[31:16] <= b[15:0];

           r[15:0] <= 16'b0;

       end

       else if (aluc == 4'b0011) // sll

           r <= b << a;

       else if (aluc == 4'b0111) // srl

           r <= b >> a;

       else if (aluc == 4'b1111) begin // sra

           r <= b >>> a;

       end

   end

endmodule

**Data Mem**

`timescale 1ns / 1ps

module data\_mem(clk, clrn, wmem, a, di, do);

   input clk, clrn, wmem;

   input [31:0] a;

   input [31:0] di;

   output [31:0] do;

   reg [31:0] dataMem [0:31];

   integer i;

   assign do = dataMem[a>>2];

   always @(posedge clk)begin

       if(clrn == 0) begin

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

               dataMem[i] = 0;

           end

           dataMem[5'h14] = 32'h000000a3;

           dataMem[5'h15] = 32'h00000027;

           dataMem[5'h16] = 32'h00000079;

           dataMem[5'h17] = 32'h00000115;

       end

       else begin

           if (wmem == 1) begin

               dataMem[a>>2] <= di;

           end

       end

   end

endmodule

**m2regMux**

`timescale 1ns / 1ps

module m2reg\_mux(m2reg, r, do, d);

   input m2reg;

   input [31:0] r;

   input [31:0] do;

   output reg [31:0] d;

   always @(\*) begin

       if (m2reg == 1)

           d <= do;

       else

           d <= r;

   end

endmodule

**Constraint**

##Clock signal

set\_property -dict { PACKAGE\_PIN L16   IOSTANDARD LVCMOS33 } [get\_ports { clk }]; #IO\_L11P\_T1\_SRCC\_35 Sch=sysclk

create\_clock -add -name sys\_clk\_pin -period 8.00 -waveform {0 4} [get\_ports { clk }];

##Switches

set\_property -dict { PACKAGE\_PIN G15   IOSTANDARD LVCMOS33 } [get\_ports { clrn }]; #IO\_L19N\_T3\_VREF\_35 Sch=SW0

##LEDs

set\_property -dict { PACKAGE\_PIN M14   IOSTANDARD LVCMOS33 } [get\_ports { led }]; #IO\_L23P\_T3\_35 Sch=LED0