**北京邮电大学电子工程学院**

**专业实验报告**



**班级:2016211205**

**姓名:闫昂霄**

**学号:2016210848**

**班内序号:02**

目录

[一、 RTL级设计与仿真 3](#_Toc11317992)

[1) 选择器设计 3](#_Toc11317993)

[2) 程序计数器设计 4](#_Toc11317994)

[3) 指令寄存器设计 6](#_Toc11317995)

[4) 算术逻辑单元设计 9](#_Toc11317996)

[5) 存储器设计 12](#_Toc11317997)

[6) 设计时序逻辑时采用阻塞赋值与非阻塞赋值的区别 16](#_Toc11317998)

[7) 状态控制机设计 18](#_Toc11317999)

[8) CPU集成设计及验证 26](#_Toc11318000)

[二、 电路综合与仿真 32](#_Toc11318001)

[1) 综合前准备 33](#_Toc11318002)

[2) 综合结果分析 33](#_Toc11318003)

[3) 综合后仿真 36](#_Toc11318004)

[三、 版图设计与仿真 38](#_Toc11318005)

[1) 数据准备 38](#_Toc11318006)

[2) 布图规划 38](#_Toc11318007)

[3) 布局 39](#_Toc11318008)

[4) 时钟树综合 40](#_Toc11318009)

[5) 布线 41](#_Toc11318010)

[6) 寄生参数导出和后仿真 42](#_Toc11318011)

[四、 实验总结 43](#_Toc11318012)

## RTL级设计与仿真

### 选择器设计

**功能描述**

**本模块实现了一个二选一选择器，该选择器用于选择输出下一条指令的存储地址（来自程序计数器）或者内存数据的读写地址（来自指令寄存器），具体选择以sel位确定。在cpu读取执行指令的8个时钟周期中，前四个周期sel为1(读取指令)，后四个周期sel为零(执行数据读写)**

**设计代码**

module scale\_mux (out,sel,b,a);

parameter size=1;//定义参数

output[size-1:0] out;//输出

input [size-1:0] b,a;//输入

input sel;//选择位

assign out =(!sel)?a:

(sel) ?b:

{size{1'bx}};//sel为1 输出b sel为0输出a

endmodule

**测试代码**

`define width 8//参数赋值-选择器位宽为8

`timescale 1 ns / 1 ns

module mux\_test;

reg [`width:1] a,b;

wire [`width:1] out;

reg sel;

scale\_mux #(`width) ml (.out(out), .sel(sel), .b(b), .a(a));

initial

begin

$monitor($stime,"sel=%b a=%b b=%b out =%b",sel,a,b,out);

$dumpvars(2,mux\_test);

sel=0;b={`width{1'b0}};a={`width{1'b1}};

#5 sel=0;b={`width{1'b1}};a={`width{1'b0}};

#5 sel=1;b={`width{1'b0}};a={`width{1'b1}};

#5 sel=1;b={`width{1'b1}};a={`width{1'b0}};

#5 $finish;

end

endmodule

**测试结果及分析**



由测试结果可以看到

0ns时 sel=0输出a=11111111

5ns时 sel=0输出a=00000000

10ns时 sel=1输出b=000000000

15ns时 sel=1 输出b=11111111 实现了二选一选择器的功能

### 程序计数器设计

**程序计数器用于用于存放要执行的指令在内存中的地址，若是顺序执行，程序计数器中的地址pc\_addr在时钟上升沿(inc\_pc上升沿)自增一（本实验中cpu的一条指令占一个字节），以指向下一条指令。若执行了转移指令，程序计数器将输出由ir\_addr输入的跳转地址（load为高电平时在时钟上升沿装入），当load为低电平时，则为顺序执行状态。**

**设计代码**

`timescale 1 ns / 100 ps

module counter ( cnt,clk,data,rst\_,load);

output [4:0] cnt;//输出5位内存地址

input [4:0] data;//输入要跳转到的内存地址

input clk;//时钟

input rst\_;//复位

input load;//装入转移地址使能

reg [4:0] cnt;

always @ (posedge clk or negedge rst\_)

if( !rst\_ )//复位

cnt<=0;

else

if(load)

cnt<=data;//转移执行

else

cnt<=cnt + 1;//顺序执行

endmodule

**测试代码**

`timescale 1 ns / 1 ns

module counter\_test;

wire [4:0] cnt;

reg [4:0] data;

reg rst\_;

reg load;

reg clk;

counter c1

(.cnt(cnt),.clk(clk),.data(data),.rst\_(rst\_),.load(load));

initial begin

clk=0;

forever begin

#10 clk =1'b1;//时钟周期20ns

#10 clk =1'b0;

end

end

initial

begin

$timeformat (-9,1,"ns ",9);

$monitor("time = %t,data = %h,clk = %b,rst\_=%b,load =%b, cnt=%b",

$stime, data,clk,rst\_,load,cnt);

$dumpvars(2,counter\_test);

end

task expect;

input [4:0] expects;

if (cnt !== expects) begin

$display ("At time %t cnt is%b and should be %b",$time,cnt,expects);

$display ("TEST FAILED");

$finish;

end

endtask

initial

begin

@(negedge clk)

{rst\_,load,data}=7'b0\_X\_XXXXX;//复位

@(negedge clk) expect (5'h00);//检查复位是否有效

{rst\_,load,data}=7'b1\_1\_11101;//load=1 data=11101 装载转移地址11101（1DH）

@(negedge clk) expect (5'h1D);//检查输出是否为装载地址1DH

{rst\_, load,data}=7'b1\_0\_11101;//load复位，顺序执行 输出地址加一

repeat(5) @(negedge clk);expect(5'h02);//经过5个上升沿，输出应为5‘h1D+5=5’h02

{rst\_, load, data}= 7'b1\_1\_11111; //装载转移地址 11111（1FH）

@(negedge clk) expect(5'h1F);//下降沿检查是否装载了新地址

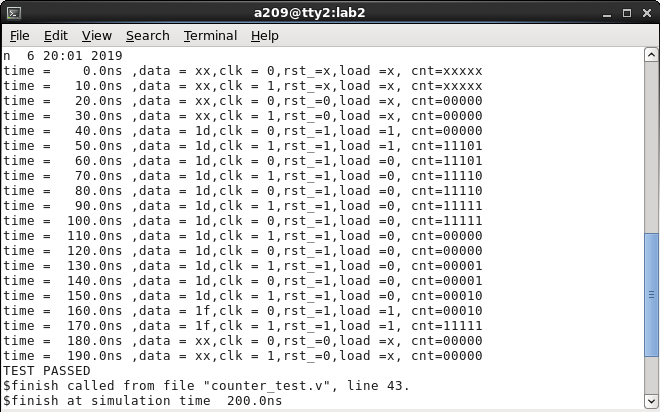
{rst\_, load, data}= 7'b0\_X\_XXXXX;//复位

@(negedge clk) expect(5'h00);//检查复位是否有效

$display( "TEST PASSED");

$finish;

**测试结果**



**可以看到 模块输出与预计相同，测试通过。**

### 指令寄存器设计

**本模块设计了一个八位指令寄存器。指令寄存器的作用是存放从内存中取出的将要执行的指令。在clk的上升沿，若ld\_ir(load)有效，则将指令由data端口存入指令寄存器并从端口out输出；rst\_为低电平复位信号，其有效时将使得寄存器输出置零。**

**对于cpu中另一个模块累加器，作用是存放算术逻辑单元的运算结果。其接口逻辑和位数与指令寄存器相同，故二者共用如下设计。**

**设计代码**

`timescale 1 ns/ 1 ns

module register(out, data, load, clk,rst\_);

output[7:0] out;

input [7:0] data;

input load;

input clk;

input rst\_;

wire [7:0] n1, n2;

wire [7:0] data;

wire [7:0] out;

wire load;

wire clk;

wire rst\_;

mux m0(.out(n2[0]),.sel(load),.b(data[0]),.a(n1[0]) );//二选一选择器（库文件中给出），负责确//定当前时刻是否存入新的指令data(取决于load)

mux m1(.out(n2[1]),.sel(load),.b(data[1]),.a(n1[1]) );

mux m2(.out(n2[2]),.sel(load),.b(data[2]),.a(n1[2]) );

mux m3(.out(n2[3]),.sel(load),.b(data[3]),.a(n1[3]) );

mux m4(.out(n2[4]),.sel(load),.b(data[4]),.a(n1[4]) );

mux m5(.out(n2[5]),.sel(load),.b(data[5]),.a(n1[5]) );

mux m6(.out(n2[6]),.sel(load),.b(data[6]),.a(n1[6]) );

mux m7(.out(n2[7]),.sel(load),.b(data[7]),.a(n1[7]) );

assign n1[0] = out[0];//将触发器的输出作为选择器的一个输入，用以实现指令保持。

assign n1[1] = out[1];

assign n1[2] = out[2];

assign n1[3] = out[3];

assign n1[4] = out[4];

assign n1[5] = out[5];

assign n1[6] = out[6];

assign n1[7] = out[7];

dffr d0(.q(out[0]),.q\_(),.d(n2[0]),.clk(clk),.rst\_(rst\_));//触发器例化

dffr d1(.q(out[1]),.q\_(),.d(n2[1]),.clk(clk),.rst\_(rst\_));

dffr d2(.q(out[2]),.q\_(),.d(n2[2]),.clk(clk),.rst\_(rst\_));

dffr d3(.q(out[3]),.q\_(),.d(n2[3]),.clk(clk),.rst\_(rst\_));

dffr d4(.q(out[4]),.q\_(),.d(n2[4]),.clk(clk),.rst\_(rst\_));

dffr d5(.q(out[5]),.q\_(),.d(n2[5]),.clk(clk),.rst\_(rst\_));

dffr d6(.q(out[6]),.q\_(),.d(n2[6]),.clk(clk),.rst\_(rst\_));

dffr d7(.q(out[7]),.q\_(),.d(n2[7]),.clk(clk),.rst\_(rst\_));

endmodule

**时钟模块**

`timescale 1 ns/1 ns

module clock(clk);

output clk;

reg clk;

initial

clk=0;

always

#10 clk = ~clk;//时钟周期20ns

endmodule

**测试代码**

`timescale 1 ns/ 1 ns

module register\_test;

wire [7:0] out;

reg[7:0] data;

reg load;

reg rst\_;

wire clk;

//例化指令寄存器

register r1(.out(out),.data(data),.load(load),.clk(clk),.rst\_(rst\_));

//例化时钟模块

clock c1(.clk(clk));

//monitor signals

initial

$monitor($time,,"clk=%b data=%h load=%b out=%h",clk,data[7:0],load,out[7:0]);

//apply stimulus

initial

begin

@(negedge clk)

rst\_ =0;//指令寄存器复位

data =0;

load =0;

@(negedge clk)

rst\_=1;

@(negedge clk)

data='h55;//第三个下降沿(40ns)装入指令55h，应在50ns（下一上升沿）处输出55h

load=1;

@(negedge clk) //第四个下降沿(60ns)装入指令AAh, 应在70ns（下一上升沿）处输出55h

data='hAA;

load=1;

@(negedge clk)

data = 'hCC;load=0;//第五个下降沿(80ns)，指令寄存器保持原有指令，输出不变

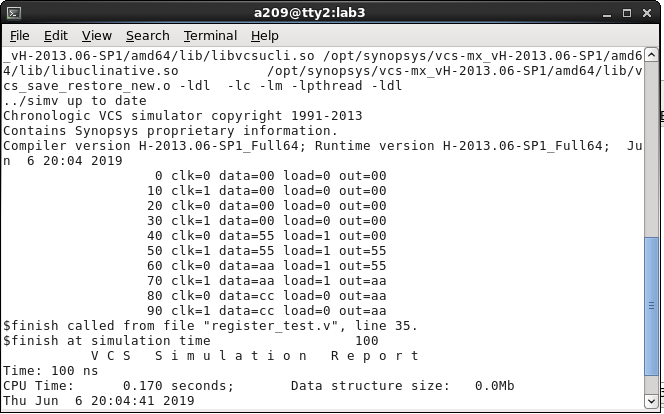
@(negedge clk)

$finish;

end

endmodule

**测试结果**



**由仿真结果可得，寄存器在50ns输出了前一下降沿装入的指令55h，在70ns输出了前一下降沿装入的指令aah，之后输出保持不变，符合预期结果。**

### 算术逻辑单元设计

**算术逻辑单元负责cpu中算术逻辑运算AND XOR ADD，以及LDA操作和STO操作，并输出操作结果ALU\_out到累加器中；并判断当前结果是否为零，将判断结果zero输出到控制器中，在执行SKZ操作时决定下一条指令地址。**

**设计代码**

`timescale 1 ns/ 100 ps

module alu(out,zero,opcode,data,accum);

input [7:0] data,accum;//输入为八位操作数和八位累加器输出

input [2:0] opcode;//操作码

output zero;//累加器当前数据是否为零(高电平表示输出为零)

output [7:0] out;//运算结果输出

reg [7:0] out;

reg zero;

parameter PASS0 = 3'b000,//HLT程序暂停

PASS1 = 3'b001,//SKZ 累加器当前数据为零则跳过下一条语句继续执行

ADD = 3'b010,//ADD data与accum相加

AND = 3'b011,//AND data与accum相与

XOR = 3'b100,//XOR data与accum异或

PASSD = 3'b101,//LDA 将指令中给出地址的数据放入累加器

PASS6 = 3'b110,//STO 将累加器中的数据放入指令中给出的地址

PASS7 = 3'b111;//JMP 跳转至指令给出地址继续执行

always@(data,accum,opcode)

begin

if(accum==8'h00)//判断累加器数据是否为零

zero=1;

else

zero=0;

case (opcode)

PASS0:out=accum;//HLT 累加器数据不变

PASS1:out=accum;//SKZ 累加器数据不变

ADD:out=accum+data;//相加

AND:out=accum&data;//相与

XOR:out=accum^data;//异或

PASSD:out=data;//LDA 累加器装入内存操作数

PASS6:out=accum;//STO Accumulator->BUS

PASS7:out=accum;//JMP 累加器数据不变

default: out=8'bx;//避免意外生成锁存器

endcase

end

endmodule

**测试文件**

`timescale 1 ns/ 1ns

`define DELAY 20

module alu\_test;

wire [7:0] out;

reg [7:0] data;

reg [7:0] accum;

reg [2:0] opcode;

integer i;

parameter PASS0 = 3'b000,

PASS1 = 3'b001,

ADD = 3'b010,

AND = 3'b011,

XOR = 3'b100,

PASSD = 3'b101,

PASS6 = 3'b110,

PASS7 = 3'b111;

alu alu1(.out(out),.zero(zero),.opcode(opcode),.data(data),.accum(accum));//算术逻辑单元例化

initial

begin

$display("<------------INPUTS------------><-OUTPUTS->");

$display(" TIME OPCODE DATA IN ACCUM IN ALU OUT ZERO BIT");

$display("--------- ------ -------- -------- -------- --------");

$timeformat (-9,1," ns",9);

$dumpvars(2,alu\_test);

end

task expect;

input [8:0] expects;

begin

$display("%t %b %b %b %b %b", $time,opcode, data,accum, out, zero);

if({zero,out} !==expects)

begin

$display("At time %t:zero is %b and should be %b, out is %b and should be %b",

$time, zero, expects[8],out,expects[7:0]);

$display("TEST FAILED" );

$finish;

end

end

endtask

initial

begin

//HLT指令，输出结果等于来自累加器的输入

{opcode,accum,data} = {PASS0,8'h00,8'hFF}; #(`DELAY) expect({1'b1,accum});//zero=1

{opcode,accum,data} = {PASS0,8'h55,8'hFF}; #(`DELAY) expect({1'b0,accum});//zero=0

//SKZ指令，输出结果等于来自累加器的输入

{opcode,accum,data} = {PASS1,8'h55,8'hFF}; #(`DELAY) expect({1'b0,accum});//zero=0

{opcode,accum,data} = {PASS1,8'hCC,8'hFF}; #(`DELAY) expect({1'b0,accum});//zero=0

//ADD指令，输出结果等于累加器输入+操作数data

{opcode,accum,data} = {ADD,8'h33,8'hAA}; #(`DELAY) expect({1'b0,accum+data});//zero=0

// AND指令，输出结果等于累加器输入&操作数data

{opcode,accum,data} = {AND,8'h0F,8'h33}; #(`DELAY) expect({1'b0,accum&data});//zero=0

//XOR指令，输出结果等于累加器输入^操作数data

{opcode,accum,data} = {XOR,8'hF0,8'h55}; #(`DELAY) expect({1'b0,accum^data});//zero=0

//LDA指令，输出结果为内存操作数data

{opcode,accum,data} = {PASSD,8'h00,8'hAA}; #(`DELAY) expect({1'b1,data});//zero=1

{opcode,accum,data} = {PASSD,8'h00,8'hCC}; #(`DELAY) expect({1'b1,data});//zero=1

//JMP指令，输出结果等于来自累加器的输入

{opcode,accum,data} = {PASS6,8'hFF,8'hF0}; #(`DELAY) expect({1'b0,accum});//zero=0

{opcode,accum,data} = {PASS7,8'hCC,8'h0F}; #(`DELAY) expect({1'b0,accum});//zero=0

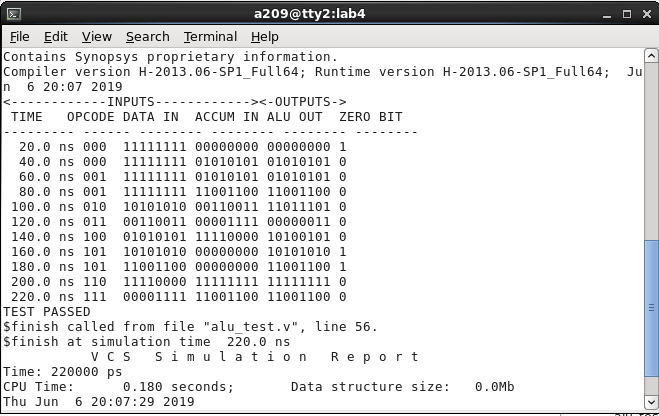
$display("TEST PASSED");

$finish;

end

endmodule

**测试结果**



**由仿真结果可以看到，输出结果与预期相符合，实现了算术逻辑单元的功能。**

### 存储器设计

**存储器用来存储指令和数据。此8位cpu指令地址位为5位，所以存储器大小位25Byte=32字节。存储器输入端口有读信号read，写信号write，地址信号addr；以及输入输出端口data。在write的上升沿，数据data从总线写入存储器；当read为高电平，数据data从存储器输出到总线上，由此实现数据的存取。**

**设计代码**

`timescale 1 ns/1 ns

module mem(data, addr,read,write);

inout [7:0] data;//8位总线

input [4:0] addr;//5位存储器地址

input read;//读使能

input write;//写使能

reg[7:0] memory[0:31];//存储区

//若read为高电平，输出地址addr处一个字节数据到总线上，

//否则data为高阻态以读取总线数据

assign data = read ? memory[addr]:8'hZ;

always@(posedge write)//上升沿写入

begin

memory[addr]<=data;

end

endmodule

**测试模块**

`timescale 1 ns/1 ns

module mem\_test;

reg read;

reg write;

reg [4:0] addr;

reg [7:0] dreg;

wire [7:0] data=(!read)?dreg:8'hZ;//read为高，则data为高阻以读取存储器数据；反之则输//出dreg以写入到存储器

integer i;

mem m1 (.data(data), .addr(addr), .read(read), .write(write));//存储器例化

initial

begin

$timeformat (-9,1," ns",9);

$display(" TIME ADDR WR RD DATA ");

$display("--------- ----- -- -- --------");

$monitor("%t %b %b %b %b", $time, addr, write, read, data);

$dumpvars(2,mem\_test);

end

task write\_val;//写任务

input [4:0] addr;

input [7:0] data;

begin

mem\_test.addr = addr;//装载地址

mem\_test.dreg=data;//装载数据总线

#1 write =1;//写入

#1 write =0;

end

endtask

task read\_val;//读任务

input [4:0] addr;//存储器读取地址

input [7:0] data;//预期读入数据

begin

mem\_test.addr =addr;//装载地址

mem\_test.read =1;//读入数据

#1 if (mem\_test.data !==data)//读入数据与预期不符

begin

$display( "At time %t and addr %b, data is %b and should be %b",

$time, addr, mem\_test.data, data);

$display ("TEST FAILED");

$finish;

end

#1 read=0;

end

endtask

initial

begin

write =0;read =0;

for(i=0;i<=31;i=i+1)

write\_val(i,i);//在地址i处写入数据i

for(i=0;i<=31;i=i+1)

read\_val(i,i);//读取地址i处数据，预期结果为i

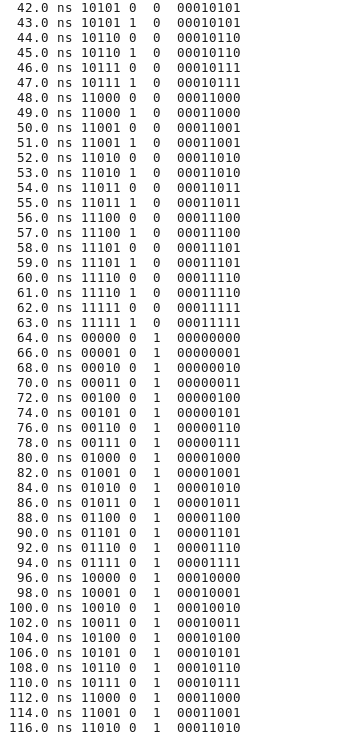
$display ("TEST PASSED");

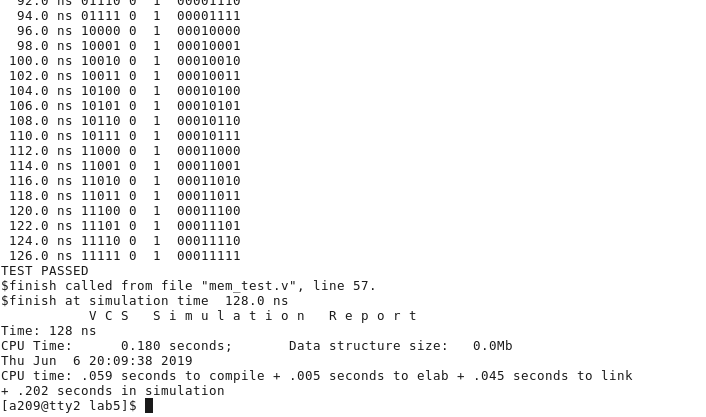
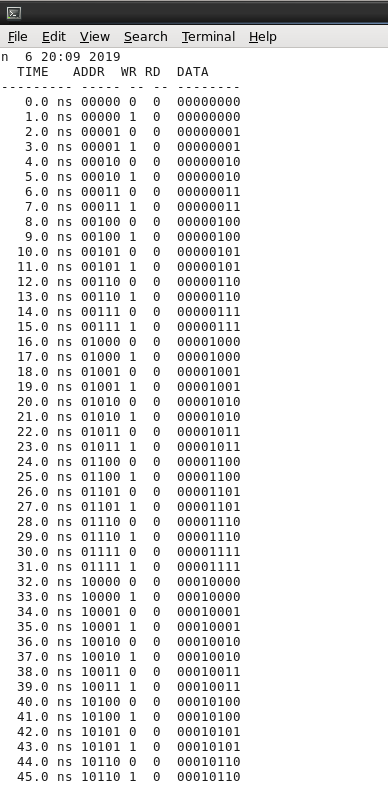
$finish;

end

endmodule

**测试结果**





**由测试结果可知，存储器能够正确写入数据且能被正确读出，实现了存储器的功能。**

### 设计时序逻辑时采用阻塞赋值与非阻塞赋值的区别

**本设计用以体现时序逻辑中阻塞赋值与非阻塞赋值各自的特点，即阻塞赋值是即时顺序执行的，而非阻塞赋值是在触发沿处并行执行的。**

module blocking(clk,a,b,c);

output [3:0]b,c;

input [3:0] a;

input clk;

reg [3:0]b,c;

always @(posedge clk)

begin

b=a;//阻塞赋值，顺序执行

c=b;//执行结果c=b=a

$display("Blocking: a = %d,b = %d, c=%d.",a,b,c);

end

endmodule

module non\_blocking(clk,a,b,c);

output [3:0]b,c;

input [3:0]a;

input clk;

reg [3:0]b,c;

always@(posedge clk)

begin

b<=a;//非阻塞赋值，并行执行

c<=b;//执行结果 b=a(original) c=b(original) ，即c和b分别为b和a前一时刻的值

$display("Non\_Blocking:a = %d, b = %d, c = %d.",a,b,c);

end

endmodule

**测试模块**

`timescale 1 ns/100 ps

module compareTop;

wire [3:0] b1,c1,b2,c2;

reg [3:0] a;

reg clk;

//时钟周期100ns

initial

begin

clk =0;

forever #50 clk =~clk;

end

initial

$dumpvars(2,compareTop);

initial

begin

a = 4'h3;

$display("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_");

# 100 a = 4'h7;//下降沿处改变a的值

$display("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_");

# 100 a = 4'hf;

$display("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_");

# 100 a = 4'ha;

$display("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_");

# 100 a = 4'h2;

$display("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_");

# 100 $display("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_");

# 100 $finish;

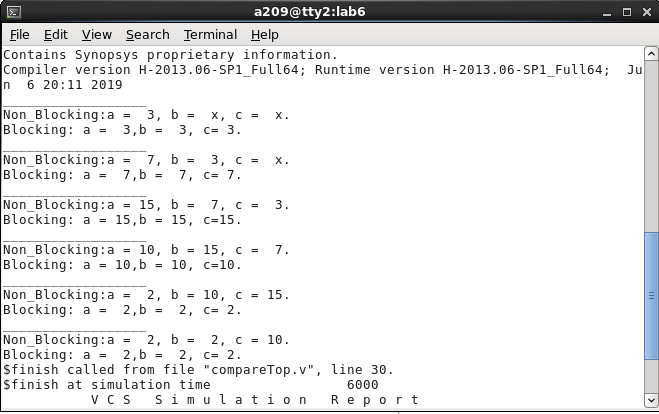
end

non\_blocking non\_blocking(clk, a,b2,c2);

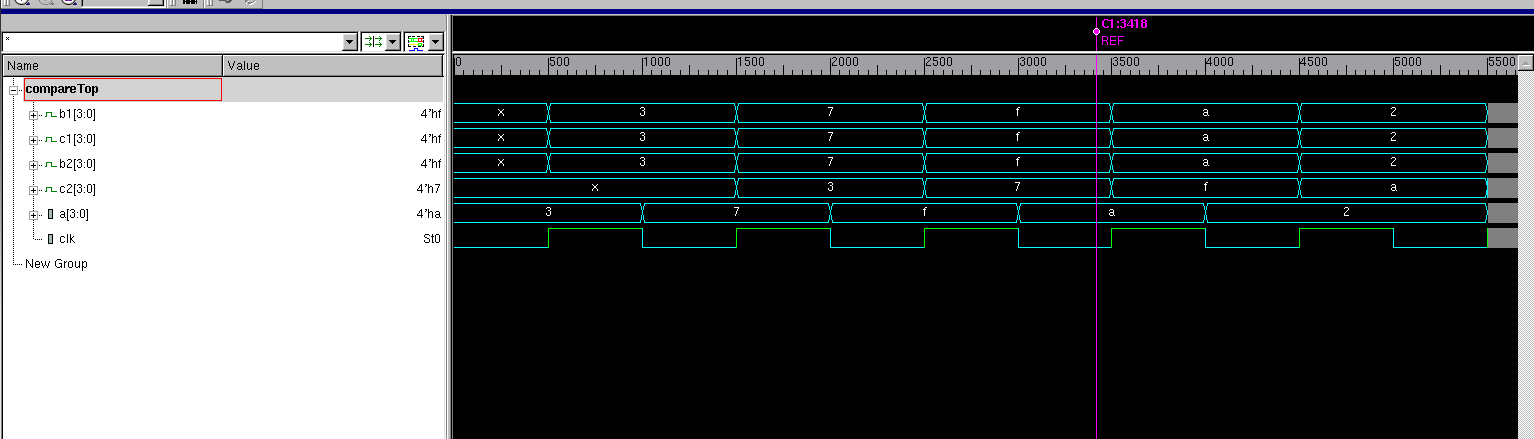
blocking blocking(clk,a,b1,c1);

endmodule

**测试结果**



**仿真波形**



**由仿真结果可得，对于非阻塞赋值(b2,c2)，其赋值为并行执行，故输入a的变化只能影响到b2, c2的值取决于上一时刻的b2。而对于阻塞赋值(b1,c1)，其赋值为顺序执行，在一个上升沿处，先执行b1=a,再执行c1=b1，故b1，c1的值均受a影响。**

### 状态控制机设计

**状态控制器是CPU的控制核心，它负责在八个时钟周期内，完成一条指令的读取与执行。其中，前四个周期用于从内存中读取指令，后四个周期通过发出控制信号执行指令并完成相应数据读写。控制器发出的控制信号有wr（写内存触发信号）,rd（读内存使能）,ld\_ir（指令寄存器装载指令使能）,ld\_ac（写入累加器使能）,ld\_pc（程序计数器装载转移地址使能）,inc\_pc（程序计数器自增时钟信号）,halt（程序暂停信号）,data\_e（总线高阻/alu\_out选择信号）,sel（指令存储地址or数据读写地址选择信号）。**

**八个时钟周期，控制器及CPU的动作如下所述：**

1. **sel为高，其余控制信号为低，选择器选择指令存储地址输出到内存中。**
2. **sel为高，rd为高，其余为低，从内存中读取指令到总线上。**
3. **sel，rd，ld\_ir为高，由于需要在clk的上升沿将指令存到指令寄存器中，所以需要在3、4周期分界处时钟上升沿将指令读入，即指令的读取需要两个时钟周期。**
4. **与第三周期相同。**
5. **inc\_pc为高电平，提供一个计数器自增上升沿，使得程序执行地址+1；halt=HLT,用以指示程序是否暂停；其余信号为低电平。**
6. **rd=aluop(aluop在执行ADD,AND,XOR,LDA指令时为高电平)，aluop为高时，执行的指令需要从内存读取操作数，故rd为高电平，内存数据输出到总线上；其余信号为低电平。**
7. **rd=aluop；inc\_pc=SKZ&zero，即若执行了SKZ且累加器中数据为零，则跳过下一条指令继续执行（inc\_pc产生一个上升沿使程序计数器自增一）；ld\_pc=JMP,即若执行了JMP指令，则将指令中地址装入程序计数器中作为新的指令（另需要inc\_pc提供上升沿(第八周期)）；data\_e=!aluop,即若执行的是ADD,AND,XOR,LDA指令，需要从内存读取数据，将总线状态置为高阻；若执行了HLT,JMP,STO,SKZ指令，可能要向内存写入数据(for STO)，故将alu运算结果输出到总线上；其余信号为低电平。**
8. **rd=aluop；inc\_pc=SKZ&zero|JMP，即若执行JMP指令，为将转移地址装入到指令寄存器，inc\_pc提供一个上升沿，若执行SKZ且zero=1，则inc\_pc的上升沿在上一周期，本周期保持高电平无上升沿;ld\_pc=JMP，指令寄存器写入使能；data\_e=!aluop,原因同上个周期；ld\_ac=aluop，即若执行了ADD,AND,XOR,LDA操作，须将运算结果从ALU写入到累加器中；wr=STO，即执行STO时，提供一写信号上升沿以向内存写入数据。**

**设计代码**

`timescale 1 ns/ 1 ns

`define HLT 3'b000

`define SKZ 3'b001

`define ADD 3'b010

`define AND 3'b011

`define XOR 3'b100

`define LDA 3'b101

`define STO 3'b110

`define JMP 3'b111

module control

(rd,wr,ld\_ir,ld\_ac,ld\_pc,inc\_pc,halt,data\_e,sel,opcode,zero,clk,rst\_);

output rd;

output wr;

output ld\_ir;

output ld\_ac;

output ld\_pc;

output inc\_pc;

output halt;

output data\_e;

output sel;

input[2:0]opcode;

input zero;

input clk;

input rst\_;

reg rd;

reg wr;

reg ld\_ir;

reg ld\_ac;

reg ld\_pc;

reg inc\_pc;

reg halt;

reg data\_e;

reg sel;

reg [2:0] nexstate;

reg [2:0] state;

always @ (posedge clk or negedge rst\_)

begin

if(!rst\_)

state<=3'b000;//复位

else

state<=nexstate;//状态转移(8个时钟周期对应8个状态)

end

always @(state)

nexstate=state+1;//下一时钟周期

always@(opcode or state or zero)

begin:blk

reg alu\_op;

alu\_op = opcode==`ADD||opcode==`AND||opcode==`XOR||opcode==`LDA;//读内存指令

case(state)

1:begin sel=1;rd=0;ld\_ir=0;inc\_pc=0;halt=0;ld\_pc=0;data\_e=0;ld\_ac=0;wr=0;end//第一周期

2:begin sel=1;rd=1;ld\_ir=0;inc\_pc=0;halt=0;ld\_pc=0;data\_e=0;ld\_ac=0;wr=0;end//第二周期

3:begin sel=1;rd=1;ld\_ir=1;inc\_pc=0;halt=0;ld\_pc=0;data\_e=0;ld\_ac=0;wr=0;end//第三周期

4:begin sel=1;rd=1;ld\_ir=1;inc\_pc=0;halt=0;ld\_pc=0;data\_e=0;ld\_ac=0;wr=0;end//第四周期

//第五周期

5:begin sel=0;rd=0;ld\_ir=0;inc\_pc=1;halt=(opcode==`HLT);ld\_pc=0;data\_e=0;ld\_ac=0;wr=0;end

//第六周期

6:begin sel=0;rd=alu\_op;ld\_ir=0;inc\_pc=0;halt=0;ld\_pc=0;data\_e=0;ld\_ac=0;wr=0;end

//第七周期

7:begin sel=0;rd=alu\_op;ld\_ir=0;inc\_pc=(opcode==`SKZ)&zero;halt=0;ld\_pc=(opcode==`JMP);data\_e=!alu\_op;ld\_ac=0;wr=0;end

//第八周期

0:begin sel=0;rd=alu\_op;ld\_ir=0;inc\_pc=(opcode==`SKZ)&zero|(opcode==`JMP);halt=0;ld\_pc=(opcode==`JMP);data\_e=!alu\_op;ld\_ac=alu\_op;wr=(opcode==`STO);end

default:begin sel=1;rd=0;ld\_ir=0;inc\_pc=0;halt=0;ld\_pc=0;data\_e=0;ld\_ac=0;wr=0;end

//避免意外生成锁存器

endcase

end

endmodule

**测试代码**

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* TEST BENCH FOR CONTROLLER \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

`timescale 1 ns / 1 ns

module control\_test ;

reg [8:0] response [0:127];//从文件中读取输出预期

reg [3:0] stimulus [0:15];//从文件中读取激励

reg [2:0] opcode;

reg clk;

reg rst\_;

reg zero;

integer i,j;

reg[(3\*8):1] mnemonic;//三字符

// Instantiate controller

control c1

(

rd ,

wr ,

ld\_ir ,

ld\_ac ,

ld\_pc ,

inc\_pc ,

halt ,

data\_e ,

sel ,

opcode ,

zero ,

clk ,

rst\_

);

// Define clock 时钟周期20ns

initial begin

clk = 1 ;

forever begin

#10 clk = 0 ;

#10 clk = 1 ;

end

end

// Generate mnemonic for debugging purposes

always @ ( opcode )

begin

case ( opcode )

3'h0 : mnemonic = "HLT" ;

3'h1 : mnemonic = "SKZ" ;

3'h2 : mnemonic = "ADD" ;

3'h3 : mnemonic = "AND" ;

3'h4 : mnemonic = "XOR" ;

3'h5 : mnemonic = "LDA" ;

3'h6 : mnemonic = "STO" ;

3'h7 : mnemonic = "JMP" ;

default : mnemonic = "???" ;

endcase

end

// Monitor signals

initial

begin

$timeformat ( -9, 1, " ns", 9 ) ;

$display ( " time rd wr ld\_ir ld\_ac ld\_pc inc\_pc halt data\_e sel opcode zero state" ) ;

$display ( "--------- -- -- ----- ----- ----- ------ ---- ------ --- ------ ---- -----" ) ;

// $shm\_open ( "waves.shm" ) ;

// $shm\_probe ( "A" ) ;

// $shm\_probe ( c1.state ) ;

$dumpvars(0,control\_test);

end

// Apply stimulus

initial

begin

$readmemb ( "stimulus.pat", stimulus ) ;//读取激励文件

rst\_=1;

@ ( negedge clk ) rst\_ = 0 ;

@ ( negedge clk ) rst\_ = 1 ;//复位

for ( i=0; i<=15; i=i+1 )

@ ( posedge ld\_ir )

@ ( negedge clk )

{ opcode, zero } = stimulus[i] ;//下降沿施加激励

end

// Check response

initial

begin

$readmemb ( "response.pat", response ) ;

@ ( posedge rst\_ )

for ( j=0; j<=127; j=j+1 )

@ ( negedge clk )//下降沿检测输出

begin

$display("%t %b %b %b %b %b %b %b %b %b %b %b %b",$time,rd,wr,ld\_ir,ld\_ac,ld\_pc,inc\_pc,halt,data\_e,sel,opcode,zero,c1.state ) ;

if ( {rd,wr,ld\_ir,ld\_ac,ld\_pc,inc\_pc,halt,data\_e,sel} !==

response[j] )//输出与预计不同

begin : blk

reg [8:0] r;

r = response[j];

$display ( "ERROR - response should be:" ) ;

$display ( "%t %b %b %b %b %b %b %b %b %b",$time,r[8],r[7],r[6],r[5],r[4],r[3],r[2],r[1],r[0] ) ;

$display ( "TEST FAILED" ) ;

$stop;

$finish ;

end

end

$display ( "TEST PASSED" ) ;

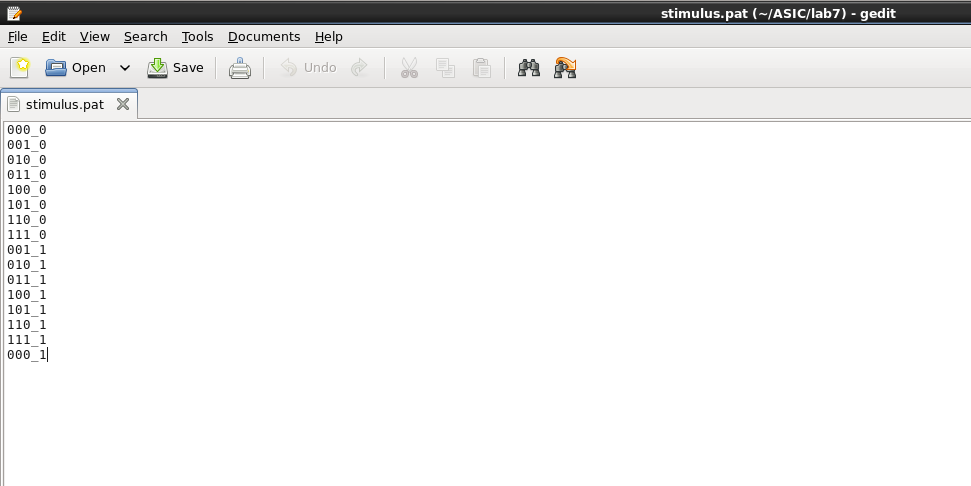
$stop;

$finish ;

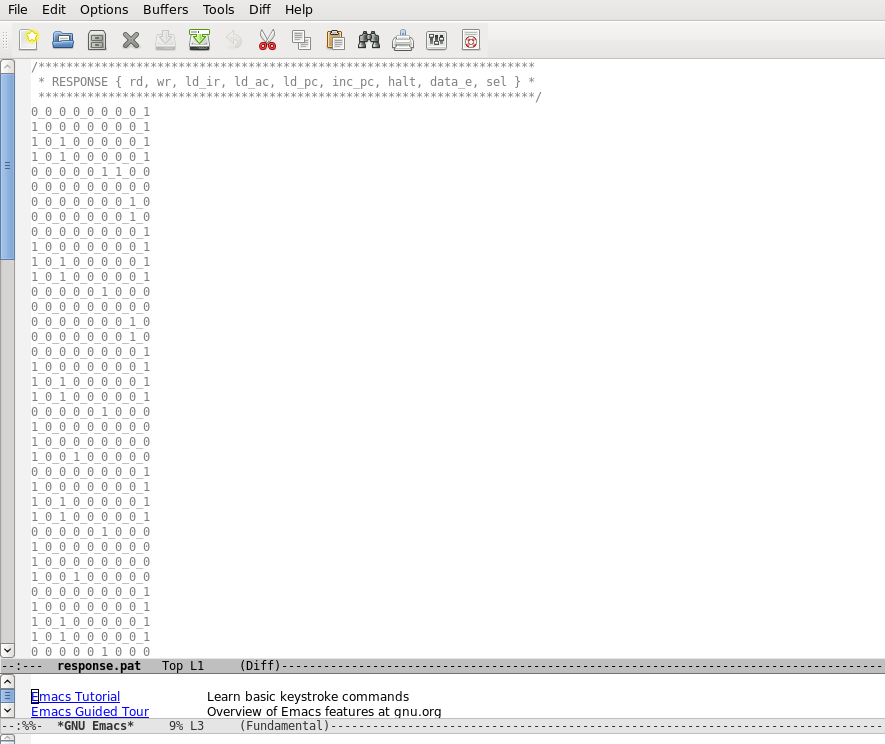
end

endmodule

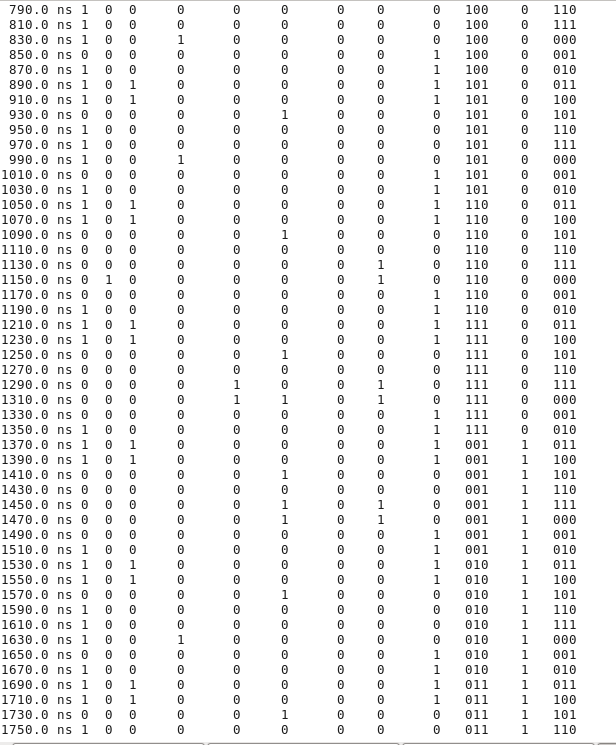
**激励文件**

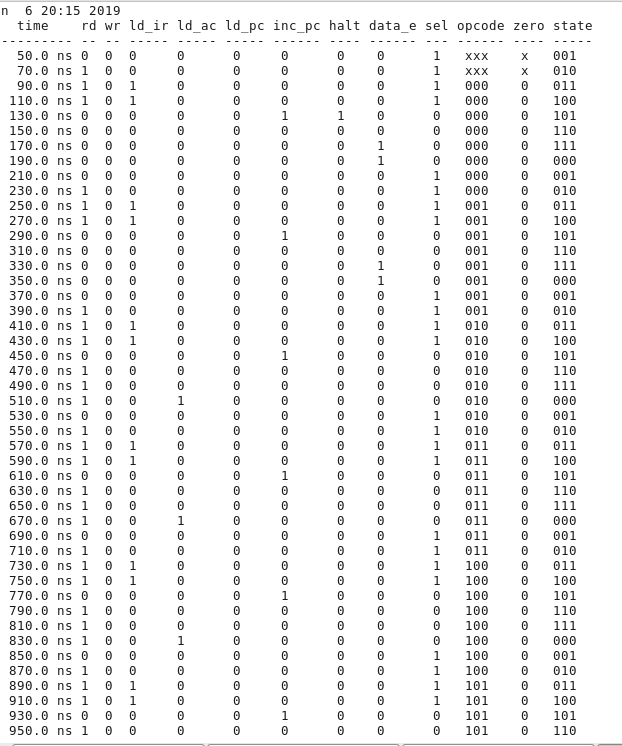


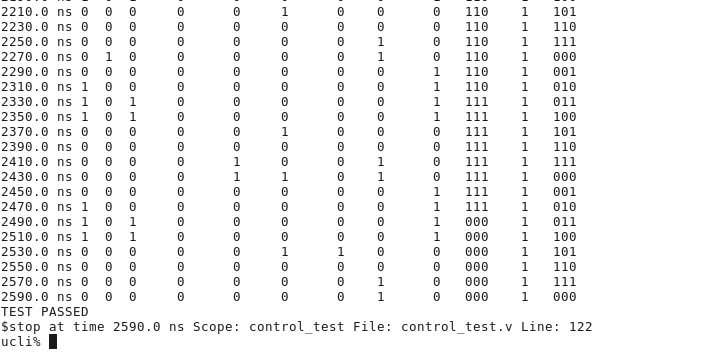
**预期输出**

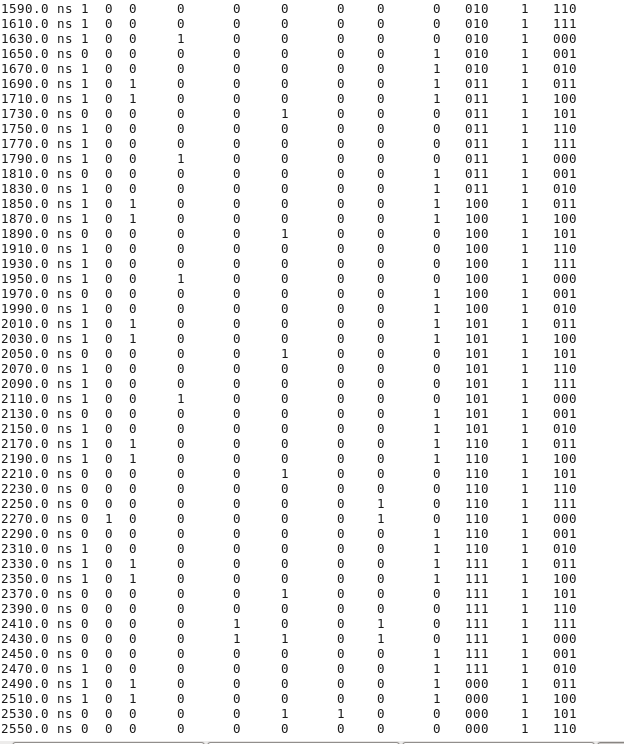


**仿真结果**









可以看到，实际输出与预计相同，实现了状态控制器的设计功能。

### CPU集成设计及验证

**本节为CPU顶层设计文件，将状态控制器，算术逻辑单元，指令寄存器，累加器，存储器，程序计数器，地址二选一选择器例化并连接对应端口。**

/\*\*\*\*\*\*\*

\* CPU \*

\*\*\*\*\*\*\*/

`timescale 1 ns / 1 ns

module cpu

(

rst\_

);

input rst\_;//复位信号

wire [7:0] data ;//总线

wire [7:0] alu\_out ;//算术逻辑单元运算结果

wire [7:0] ir\_out ;//指令寄存器输出

wire [7:0] ac\_out ;//累加器数据输出

wire [4:0] pc\_addr ;//程序计数器输出

wire [4:0] ir\_addr ;//指令寄存器输出指令低五位（操作数地址）

wire [4:0] addr ;//地址二选一选择器输出（存储器地址输入）

wire [2:0] opcode;//指令寄存器输出指令高三位（操作码）

assign opcode = ir\_out[7:5];

assign ir\_addr = ir\_out[4:0];

//Instantiate design components

control ctl1//状态控制器例化

(

.rd (rd ),

.wr (wr ),

.ld\_ir (ld\_ir ),

.ld\_ac (ld\_ac ),

.ld\_pc (ld\_pc ),

.inc\_pc (inc\_pc ),

.halt (halt ),

.data\_e (data\_e ),

.sel (sel ),

.opcode (opcode ),

.zero (zero ),

.clk (clock ),

.rst\_ (rst\_ )

);

alu alu1//算术逻辑单元例化

(

.out (alu\_out ),

.zero (zero ),

.opcode (opcode ),

.data (data ),

.accum (ac\_out )

);

register ac//累加器例化

(

.out (ac\_out ),

.data (alu\_out ),

.load (ld\_ac ),

.clk (clock ),

.rst\_ (rst\_ )

);

register ir//指令寄存器例化

(

.out (ir\_out ),

.data (data ),

.load (ld\_ir ),

.clk (clock ),

.rst\_ (rst\_ )

);

scale\_mux #5 smx//地址二选一选择器例化

(

.out (addr ),

.sel (sel ),

.b (pc\_addr ),

.a (ir\_addr )

);

mem mem1//存储器例化

(

.data (data ),

.addr (addr ),

.read (rd ),

.write (wr )

);

counter pc//程序计数器例化

(

.cnt (pc\_addr ),

.data (ir\_addr ),

.load (ld\_pc ),

.clk (inc\_pc ),

.rst\_ (rst\_ )

);

clkgen clk//时钟模块例化

(

.clk (clock )

);

//Glue logic

assign data = (data\_e) ? alu\_out: 8'bz;//总线状态控制

endmodule

/\*\*\*\*\*\*\*\*\*

\* CLOCK \*

\*\*\*\*\*\*\*\*\*/

`timescale 1 ns / 1 ns

`define period 10

module clkgen

(

clk

);

output clk;

reg clk;

initial

begin

clk = 1;

forever

begin

#(`period/2) clk = 0;//时钟周期10ns

#(`period/2) clk = 1;

end

end

endmodule

**测试模块**

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* TEST BENCH FOR VeriRISC CPU \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

`timescale 1 ns / 1 ns

module cpu\_test;

reg rst\_ ;

reg [(3\*8):1] mnemonic;

// Instantiate the VeriRISC CPU

cpu cpu1 ( rst\_ ) ;

// Generate mnemonic

always @ ( cpu1.opcode )

case ( cpu1.opcode )

3'h0 : mnemonic = "HLT" ;

3'h1 : mnemonic = "SKZ" ;

3'h2 : mnemonic = "ADD" ;

3'h3 : mnemonic = "AND" ;

3'h4 : mnemonic = "XOR" ;

3'h5 : mnemonic = "LDA" ;

3'h6 : mnemonic = "STO" ;

3'h7 : mnemonic = "JMP" ;

default : mnemonic = "???" ;

endcase

// Monitor signals

initial

begin

$timeformat ( -9, 1, " ns", 12 ) ;

// $shm\_open ( "waves.shm" ) ;

// $shm\_probe ( mnemonic, cpu1, "A" ) ;

$dumpvars (0,cpu\_test);

end

// Apply stimulus

always

begin

`ifdef INCA

$display("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

$display("\* THE FOLLOWING DEBUG TASKS ARE AVAILABLE: \*");

$display("\* Enter \"scope cpu\_test; deposit test.N 1; task test; run\" \*");

$display("\* to run the 1st diagnostic program. \*");

$display("\* Enter \"scope cpu\_test; deposit test.N 2; task test; run\" \*");

$display("\* to run the 2nd diagnostic program. \*");

$display("\* Enter \"scope cpu\_test; deposit test.N 3; task test; run\" \*");

$display("\* to run the Fibonacci program. \*");

$display("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

`else

$display("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

$display("\* THE FOLLOWING DEBUG TASKS ARE AVAILABLE: \*");

$display("\* Enter \"call test(1);run\" to run the 1st diagnostic program. \*");

$display("\* Enter \"call test(2);run\" to run the 2nd diagnostic program. \*");

$display("\* Enter \"call test(3);run\" to run the Fibonacci program. \*");

$display("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

`endif

$stop ;

@ ( negedge cpu1.clock )//复位

rst\_ = 0;

@ ( negedge cpu1.clock )

rst\_ = 1;

@ ( posedge cpu1.halt )

$display ( "HALTED AT PC = %h", cpu1.pc\_addr ) ;//程序停止

disable test ;

end

// Define the test task

task test ;

input [2:0] N ;

reg [12\*8:1] testfile ;

if ( 1<=N && N<=4 )

begin

testfile = { "CPUtest", 8'h30+N, ".dat" } ;//由测试文件编号N生成测试文件名

$readmemb ( testfile, cpu1.mem1.memory ) ;//读入测试程序到存储器中

case ( N )

1://测试一

begin

$display ( "RUNNING THE BASIC DIAGOSTIC TEST" ) ;

$display ( "THIS TEST SHOULD HALT WITH PC = 17" ) ;

$display ( "PC INSTR OP DATA ADR" ) ;

$display ( "-- ----- -- ---- ---" ) ;

forever @ ( cpu1.opcode or cpu1.ir\_addr )

$strobe ( "%h %s %h %h %h",

cpu1.pc\_addr, mnemonic, cpu1.opcode, cpu1.data, cpu1.addr ) ;

end

2://测试二

begin

$display ( "RUNNING THE ADVANCED DIAGOSTIC TEST" ) ;

$display ( "THIS TEST SHOULD HALT WITH PC = 10" ) ;

$display ( "PC INSTR OP DATA ADR" ) ;

$display ( "-- ----- -- ---- ---" ) ;

forever @ ( cpu1.opcode or cpu1.ir\_addr )

$strobe ( "%h %s %h %h %h",

cpu1.pc\_addr, mnemonic, cpu1.opcode, cpu1.data, cpu1.addr ) ;

end

3://测试三

begin

$display ( "RUNNING THE FIBONACCI CALCULATOR" ) ;

$display ( "THIS PROGRAM SHOULD CALCULATE TO 144" ) ;

$display ( "FIBONACCI NUMBER" ) ;

$display ( " ---------------" ) ;

forever @ ( cpu1.opcode )

if (cpu1.opcode == 3'h2)

$strobe ( "%d", cpu1.mem1.memory[5'h1B] ) ;

end

4://测试四

begin

$display ( "RUNNING THE MULTIPLIER" ) ;

$display ( " ---------------" ) ;

@(cpu1.opcode);

forever @ ( cpu1.opcode )

if (cpu1.opcode == 3'h0)

$strobe ( "%b", cpu1.mem1.memory[5'h1C] ) ;

end

endcase

end

else

begin

$display("Not a valid test number. Please try again." ) ;

$stop ;

end

endtask

endmodule

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* 个人编写测试程序4

\* Test program 4 for the VeriRisc CPU system, Verilog Training Course.

\* 本程序以累加方法计算X\*Y（存储在内存中），并将结果存放在RESULT字节里

\* 本程序未考虑乘法溢出情况

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

@1C

00000000 //RESULT 11100 乘积存放地址 结果应为11111111

xxxxxxxx // FF 11101 程序将此字节变为FFh后，用以将Y减一。

00110011 //mul1 X 51 11110 乘因子X

00000101 //mul2 Y 05 11111 乘因子Y

@00

101\_11101 //LDA FF 00000

011\_11100 //AND 00 00001 FF=00000000

100\_11100 //XOR 00(BUILD FF) 00010 FF=11111111

101\_11111 // LDA Y 00011 BEGIN

001\_01011 // SKZ 00100 Y==0则结束累加

111\_00111 // JMP CONTINUE 00101 Y!=0继续累加

000\_00000 //END 00110

010\_11101 // ADD FF (Y=Y-1) 00111 CONTINUE

110\_11111 // STO Y 01000

101\_11110 // LDA X 01001

010\_11100 // ADD RESULT 01010

110\_11011 // STO RESULT 01011

111\_00011 // JMP BEGIN 01100

## 电路综合与仿真

**本部分主要涉及到将前面RTL级电路进行逻辑综合转化为门级电路的过程。综合主要分为三个阶段：转换，映射，优化。转换是将HDL的描述变为一个独立于工艺的RTL级网表。映射是将RTL级网表映射到指定工艺库上，成为门级网表。在综合准备阶段，设计者要根据实际需求进行时序，面积等约束条件的设置，综合软件将根据约束条件对生成的门级网表进行优化。**

**生成的门级网表与生产商的工艺库有着一一对应的映射关系，具体电路的延迟、面积信息等都在工艺库中有所描述，综合软件将根据这些信息分析估算电路中各条路径的延迟信息，以及电路面积信息和电路功耗信息等，最终生成门级电路的综合报告。**

**在综合完成后，由于生成的门级网表考虑了实际电路的延迟，需要再次对电路功能进行仿真，确保设计正确。逻辑综合可以导出门电路和互连线的估算延迟，并输出到.sdf文件中，综合后仿真就是将该延时文件反标到门级电路中，基于时延信息进行再次仿真，其仿真结果较RTL级仿真更加接近于实际电路工作情况。**

**本实验综合针对CPU中的状态控制器模块进行**

### 综合前准备

* 1. **设置工作环境**

配置DC的运行环境，并指定综合所需要的单元库，符号库，链接库，IP库。

* 1. **添加PAD**

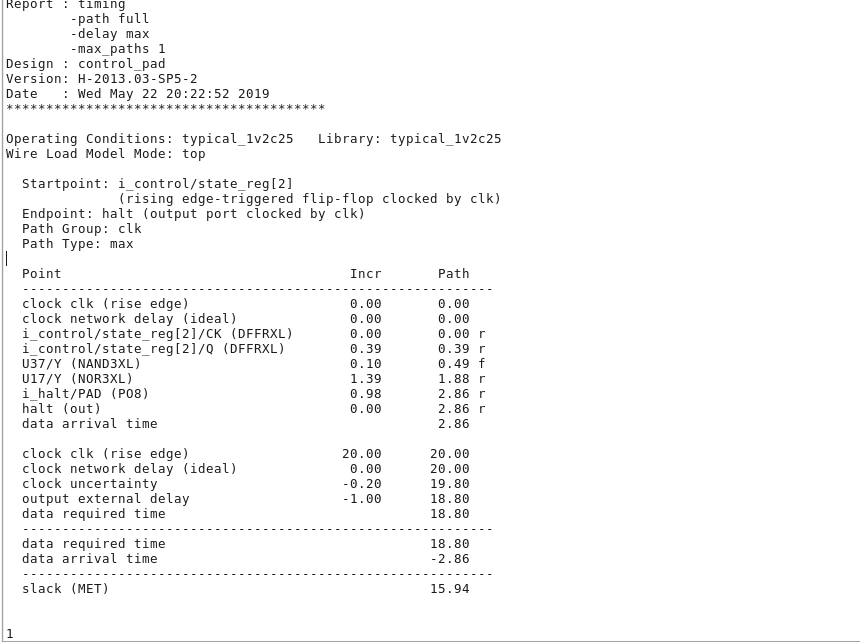
PAD是输入输出缓冲器，用以将内部信号与外部引脚相连接，用以增加电路驱动能力，并对内部电路起保护作用。

* 1. **编写综合脚本**

综合脚本为.tcl文件，用以读入设计文件，施加设计约束（时序约束，面积约束），定义环境属性（即工作条件，输出负载与输入驱动等）。

### 综合结果分析

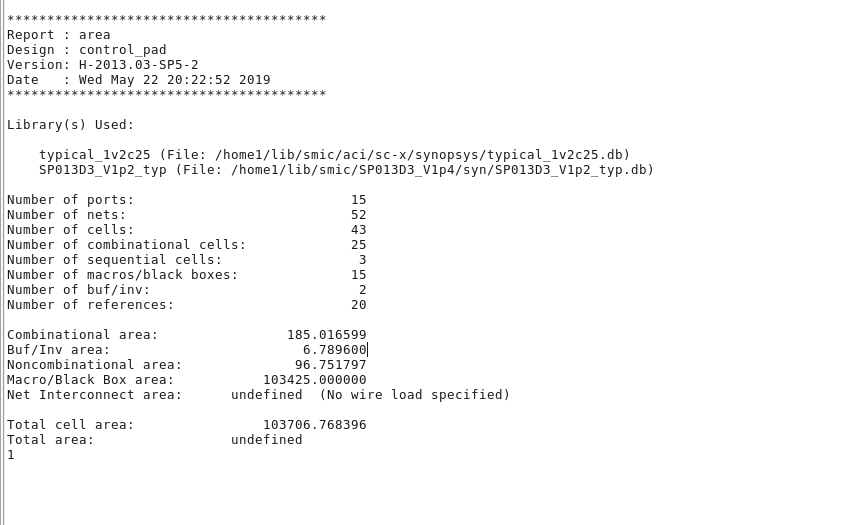
* 1. **时序分析报告**



该时序报告计算了在时钟clock的上升沿触发下从i\_control/state\_reg[2]端口到halt端口的时延，其中 path type为max，表示各结点之间延迟选取最大值。Incr表示，上一结点到本结点的时延，Path为从路径起点到本结点的累计延迟，可以看到，该路径最大延迟为2.86ns

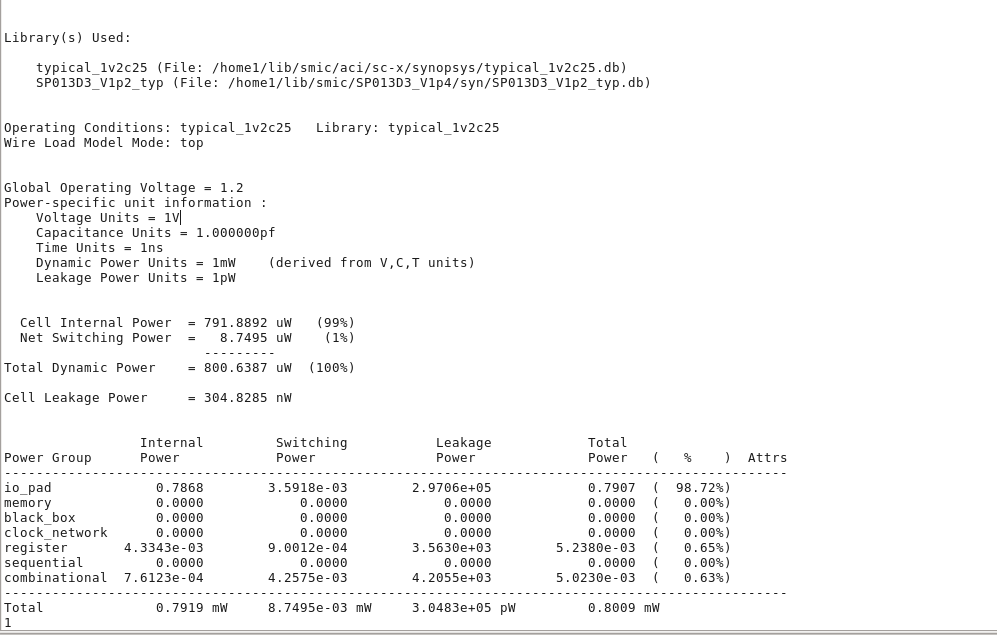
报告后半部分计算了该路径要求的最大延迟时间data required time(18.8ns)，最大延迟时间一般与寄存器的建立时间有关。此路径最大延迟由时钟周期减去时钟抖动（clock uncertainty）和外部输出延迟(output external delay)得到。data required time – data arrival time =slack（松弛时间），松弛时间较大(15.94ns)，表明电路可在满足时序要求的情况下对面积作进一步优化。

* 1. **面积分析报告**



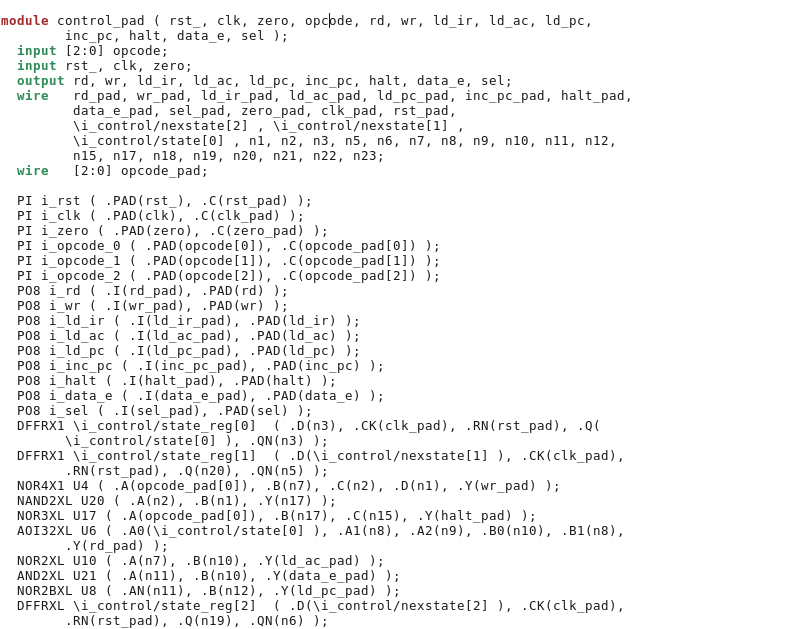
面积分析报告统计了端口数，连线数，组合/时序逻辑单元数等信息，并且计算得出了组合逻辑面积，非组合逻辑面积等面积信息，最终得出电路总面积为103706.768396个面积单位。

* 1. **功耗分析报告**



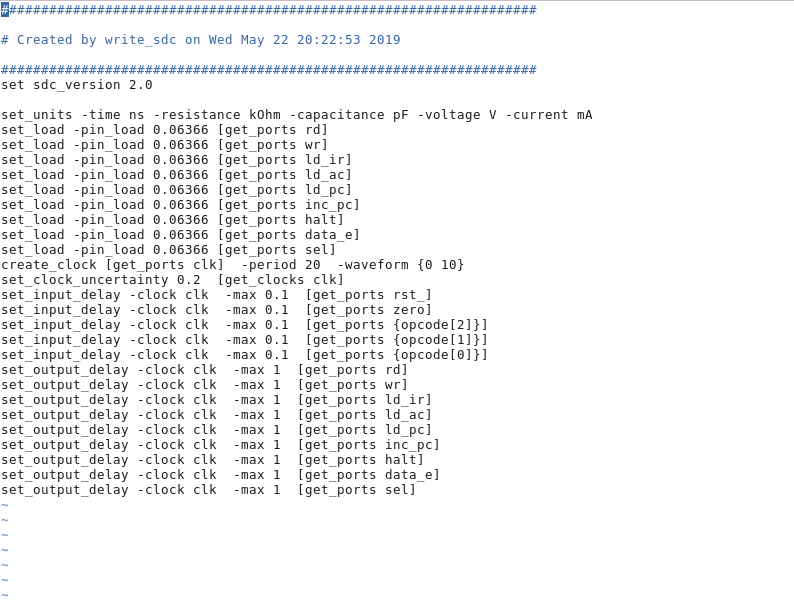
功耗分析报告计算了电路各模块的功耗情况，包括io\_pad,register,时序逻辑电路，组合逻辑电路等，可以看出，电路主要的功耗在于io\_pad（输入输出缓冲器），占了总功耗的98.72%。

* 1. **生成门级网表**



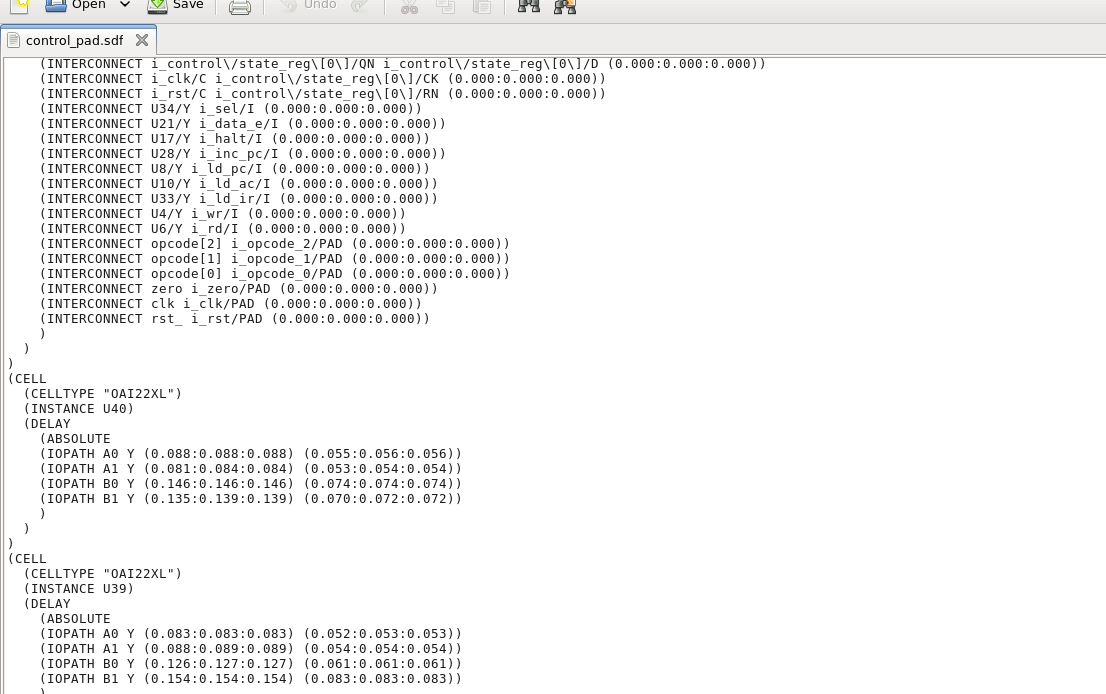
可以看出，综合之后，原有的RTL级描述变成了工艺库中各种逻辑单元的例化和互连，实现了RTL级网表到工艺库的映射。

* 1. **时序约束文件**



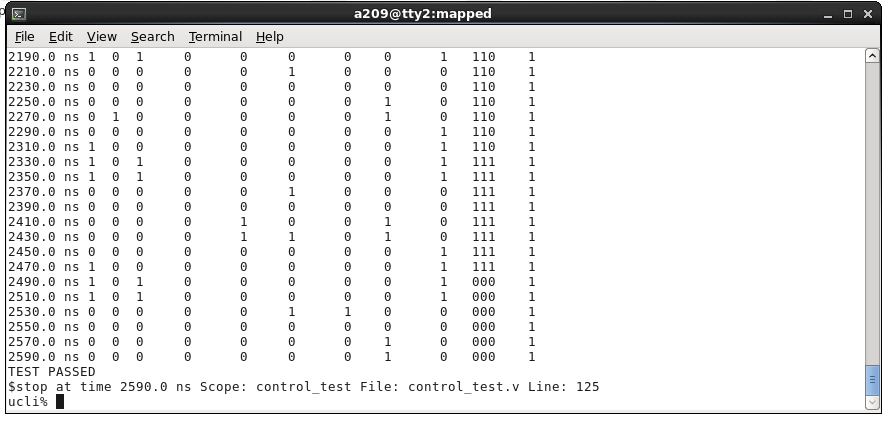
时序约束文件(control\_pad.sdc)由综合脚本导出，包含关键路径的延迟约束信息，主要供后端工具进行延迟驱动的布局布线。

* 1. **生成时序描述文件**

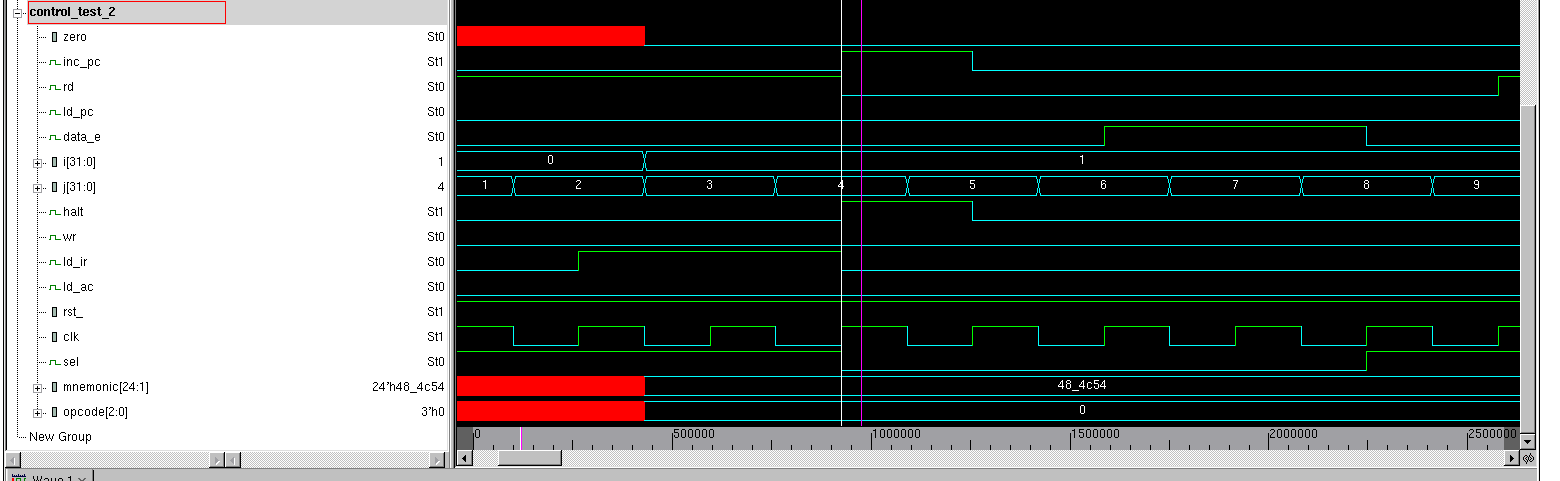
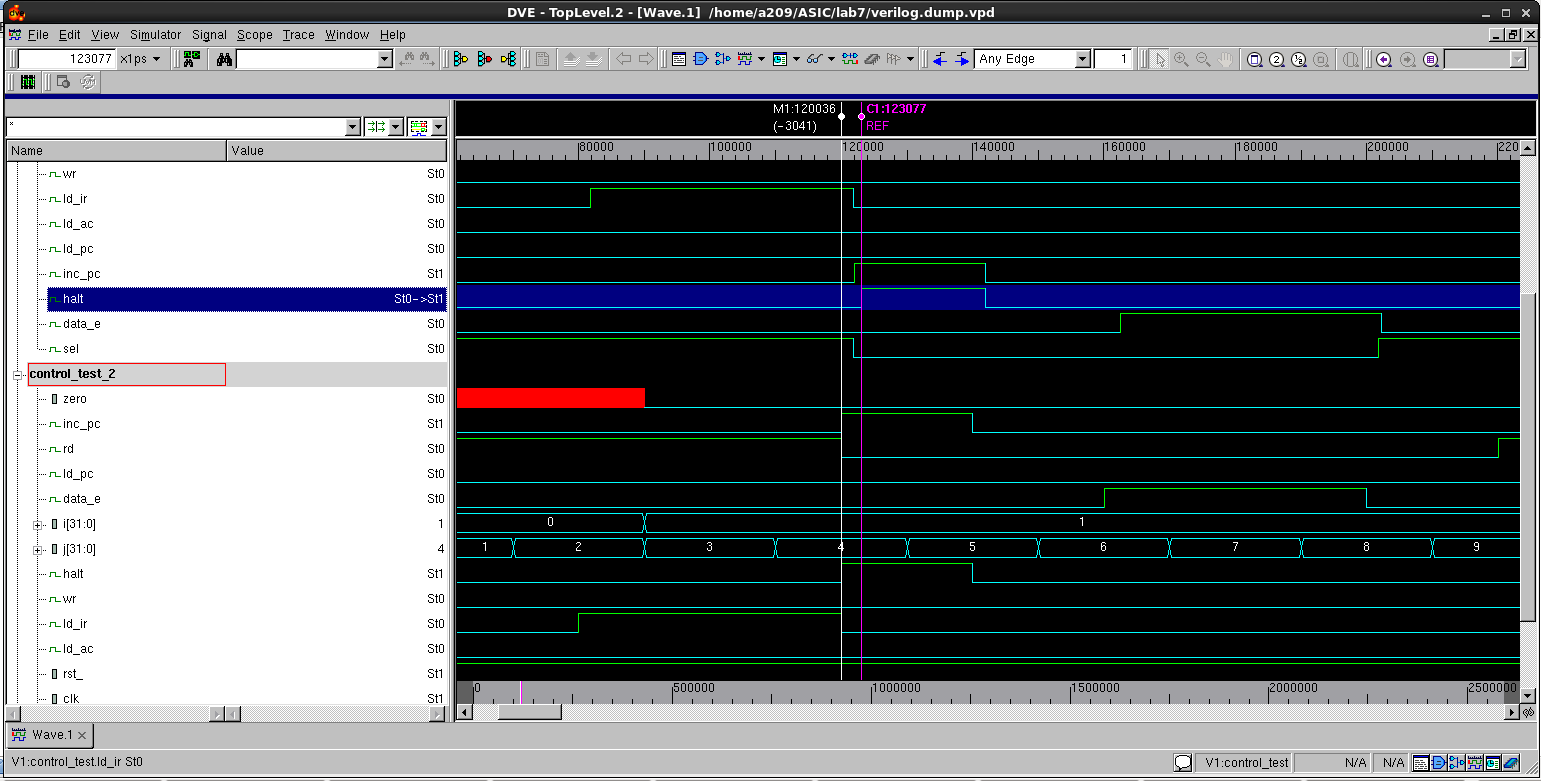


时序描述文件(control\_pad.sdf)描述了工艺库各逻辑单元的时延信息、互连线延迟信息，以及时序单元的建立/保持时间等信息，主要用于反标门级网表进行综合后仿真。

### 综合后仿真



可以看到，在反标时延信息之后，电路的时序仍能满足要求，可以达到设计要求。



**仿真波形图**

与时序分析报告中的关键路径相对应，在这里将光标标在综合后仿真(control\_test1)和RTL仿真（control\_test2）halt信号上（二者时钟同步）。其中RTL仿真结果里的halt上升沿与时钟上升沿重合，而综合后仿真结果里的halt上升沿滞后于时钟信号123.077-120.036=3.041ns，此为halt动态仿真输出延迟，大于静态时序分析延迟2.86ns。

## 版图设计与仿真

**版图设计是由综合生成的门级网表，时序约束文件和工艺库、参考库文件定义掩模版图的过程，是电路设计的物理描述。设计出的掩模版图将应用于集成电路的光刻生产中。**

**版图布线完成后，需要利用寄生参数库文件导出其寄生参数以获取最接近实际的时延信息，以进行版图设计后仿真。**

**版图设计流程包括数据准备，布图规划，布局，时钟树综合，布线五个步骤。在版图设计完成后，应对版图进行后仿真，以确保电路功能的正确性。**

### 数据准备

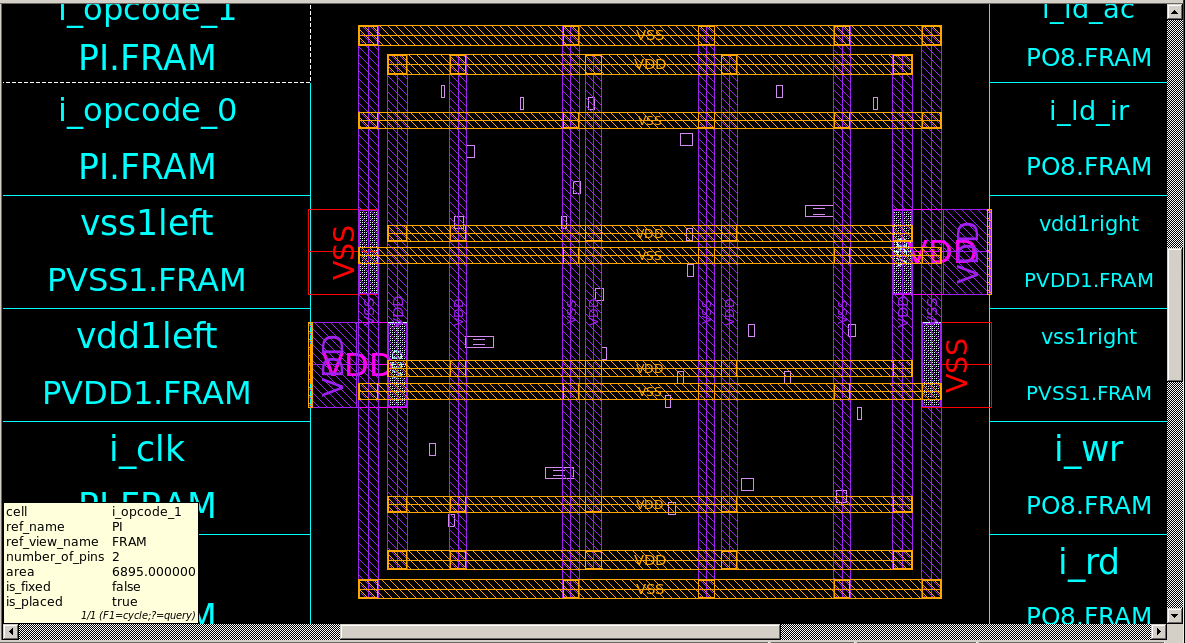
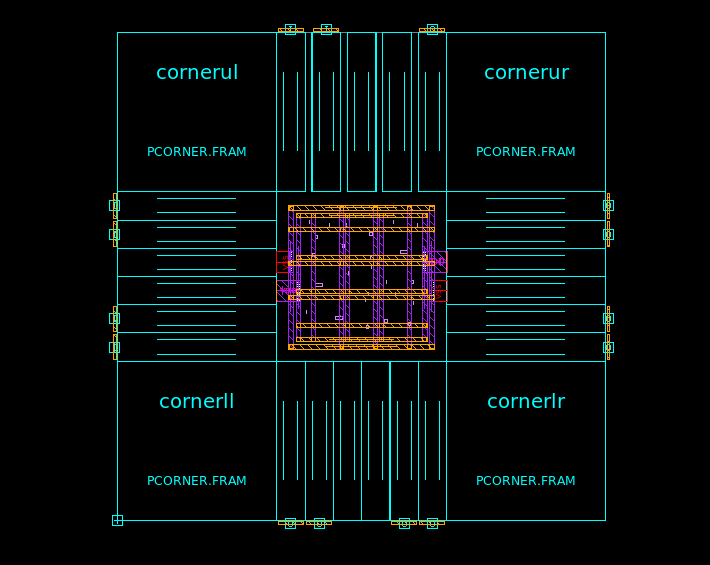
建立milkyway库，通过read\_verilog命令读入综合后的网表；通过set\_tlu\_plus\_files命令输入tluplus RC寄生参数库；通过read\_sdc命令设置时序约束文件（综合过程中导出的.sdc文件）；通过derive\_pg\_connection对门级网表中的0和1信号进行处理，并和电源地进行逻辑关联。

### 布图规划

布图规划用来确定整个芯片的形状，大小，pad的摆放顺序，并提供电源供电方案。

在布图过程中，需要添加原网表中不存在的电源pad、cornor pad等单元，并对所有pad的排放位置进行设定。另外，还需插入填充单元到I/O单元之间的间隙中以满足DRC规则；在core和IO单元之间加入pad电源环以对core和pad供电。

**布图之后生成的版图**

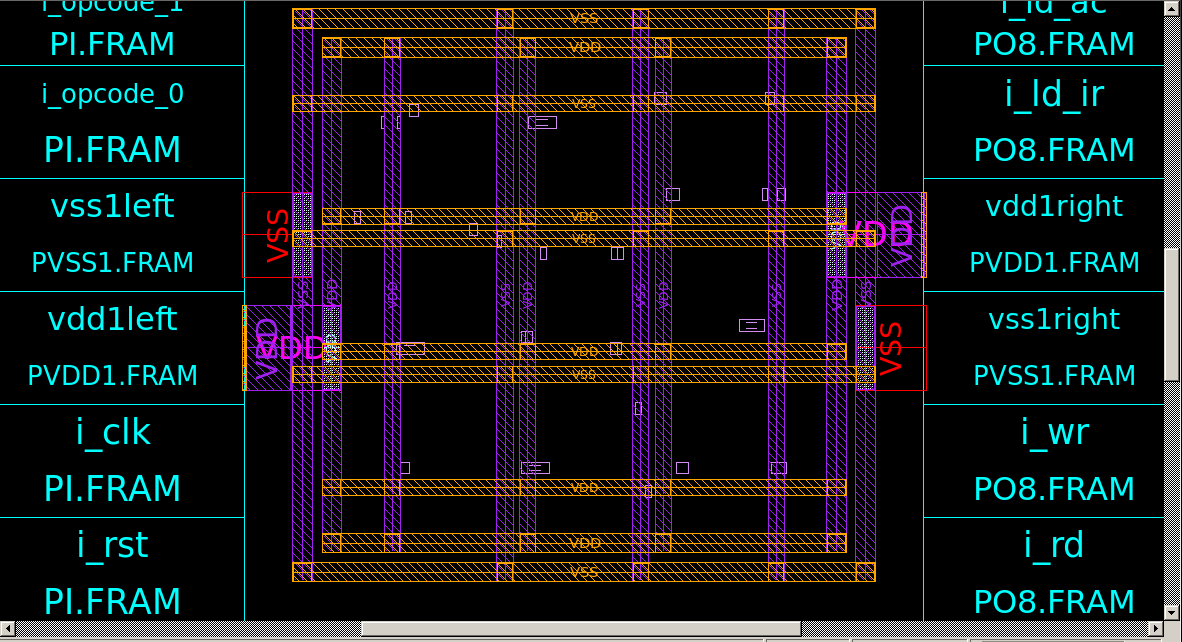


可以看到，布图后，出现了在综合后网表中不存在的电源pad和cornor pad 等单元，各个pad的摆放位置确定。整个芯片的电源网络也已经形成。

### 布局

布图之后，芯片的大小，电源网络，macro的位置便已确定。布局是在布图之后进行标准单元的布局。

在布局规划的脚本文件中可以用place\_opt命令进行布局设置，以尽量减少布线的拥挤度。布局后的版图应满足时序约束文件(.sdc)的约束要求。



可以看到版图中的标准单元在布局之后位置发生了变化

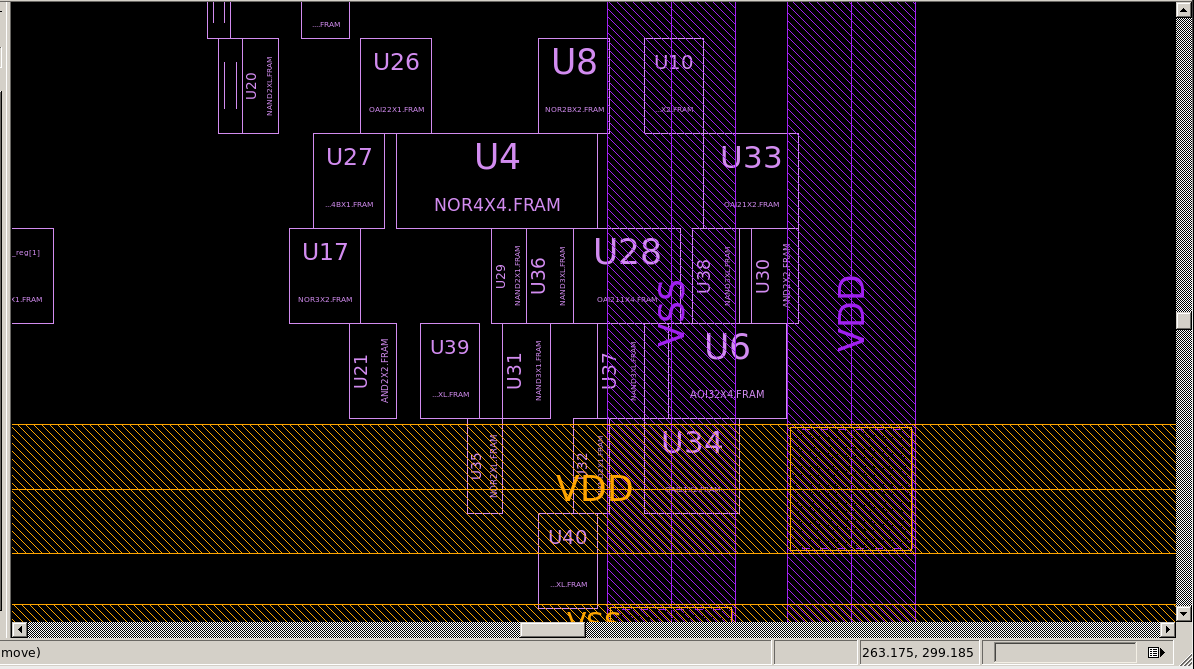
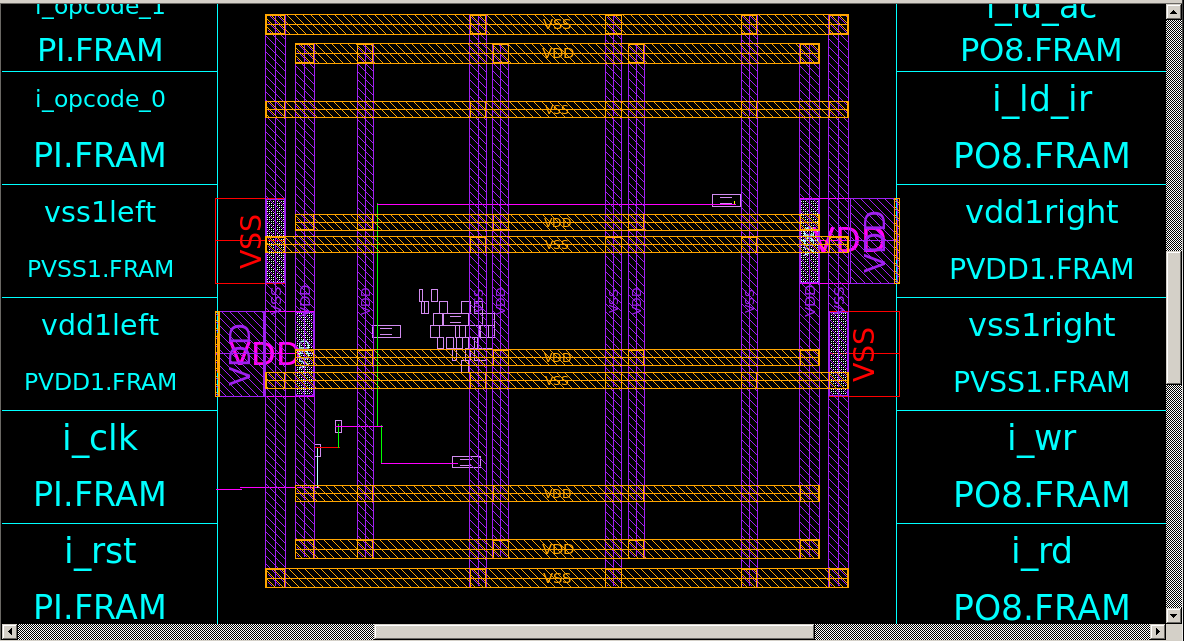
### 时钟树综合

由于电路中存在很多寄存器，这些寄存器都要有时钟信号输入，而一个时钟源信号到达各个寄存器的时间将有所偏差，称为时钟偏斜。为了减少时钟偏斜，应建立时钟树结构，通过buffer来驱动寄存器。

icc时钟树综合分为三个步骤

1. 设置时钟偏斜的目标（set\_clock\_tree\_options命令）
2. 设置在时钟树综合过程中优先使用的buffer(set\_clock\_tree\_references命令)
3. 进行时钟树综合

**时钟树综合后的版图**



由版图可以看出，时钟树综合后，时钟信号线（绿色）上加入了buffer出现在了版图中，并连接到各个寄存器单元，形成了时钟树。

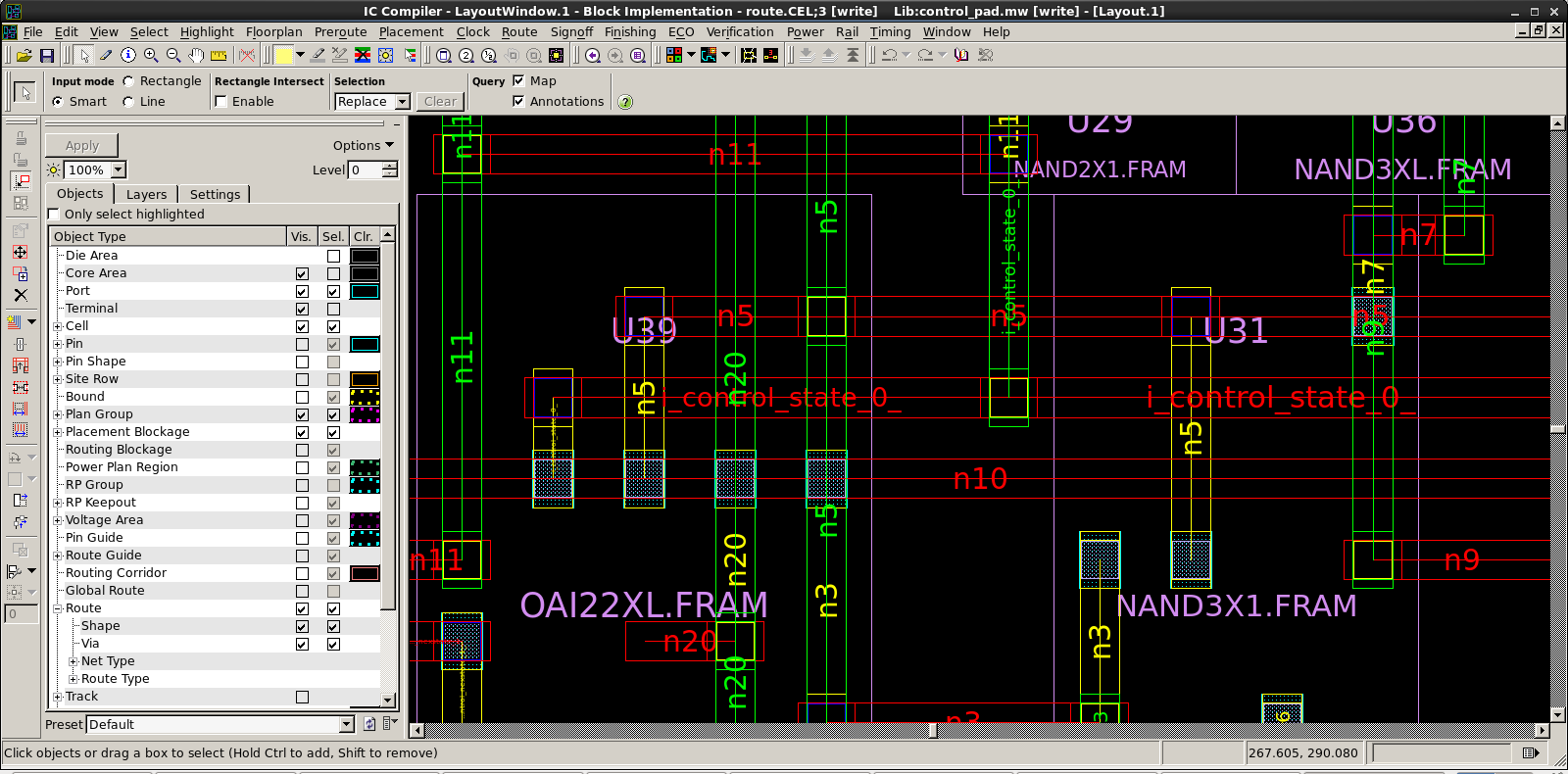
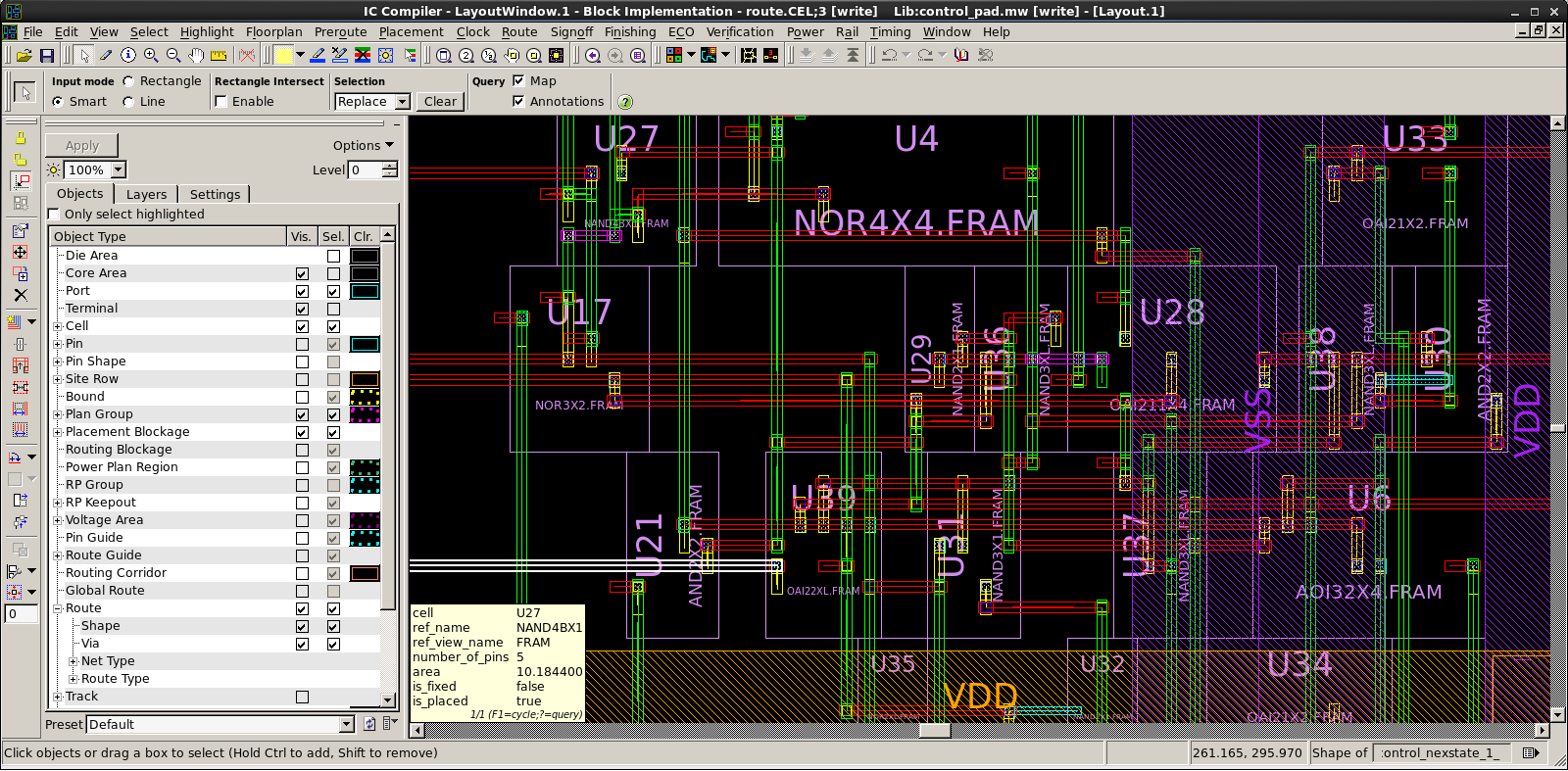
### 布线

布线是根据电路连接的关系，在满足工艺规则和电学性能的要求下，对版图完成所有需要的连线（电源地网络的布线在布图阶段已经完成）。

布线分为四个步骤

1. 全局布线：把版图区域划分为不同的布线单元，同时建立布线通道，对连线的网络连接方向和占用的布线资源（布线通道和过孔），连线的最短路径进行确定，对布线的拥塞程度进行估计，调整连线过度拥塞的部分。
2. 布线通道分配：把每一连线分配到一定的布线通道上，并对连线进行实际布线。
3. 详细布线：使用布线通道分配产生的路径进行布线和布孔。在此步进行DRC检查，用固定尺寸的sbox来修复违规。
4. 布线修补：通过尺寸逐渐加大的sbox来寻找和修复详细布线中遗漏的DRC违规。

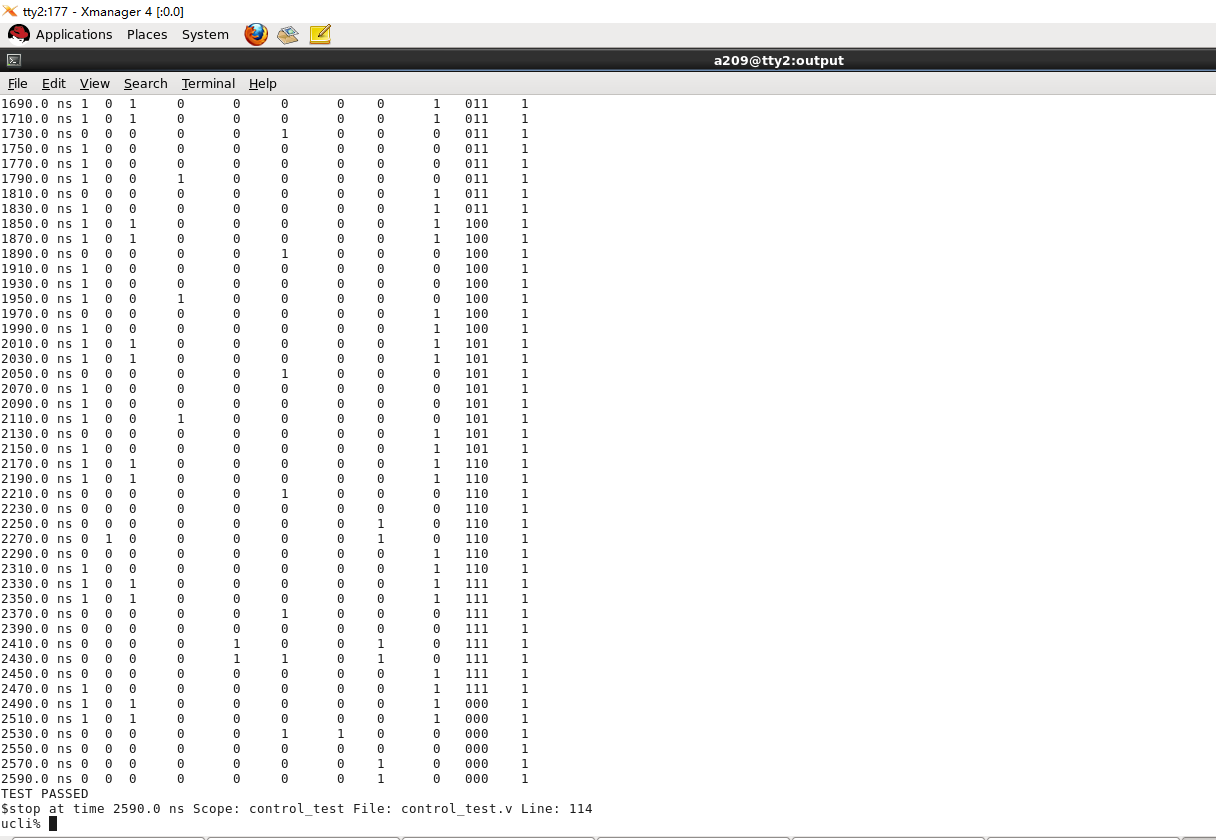
**布线结果**



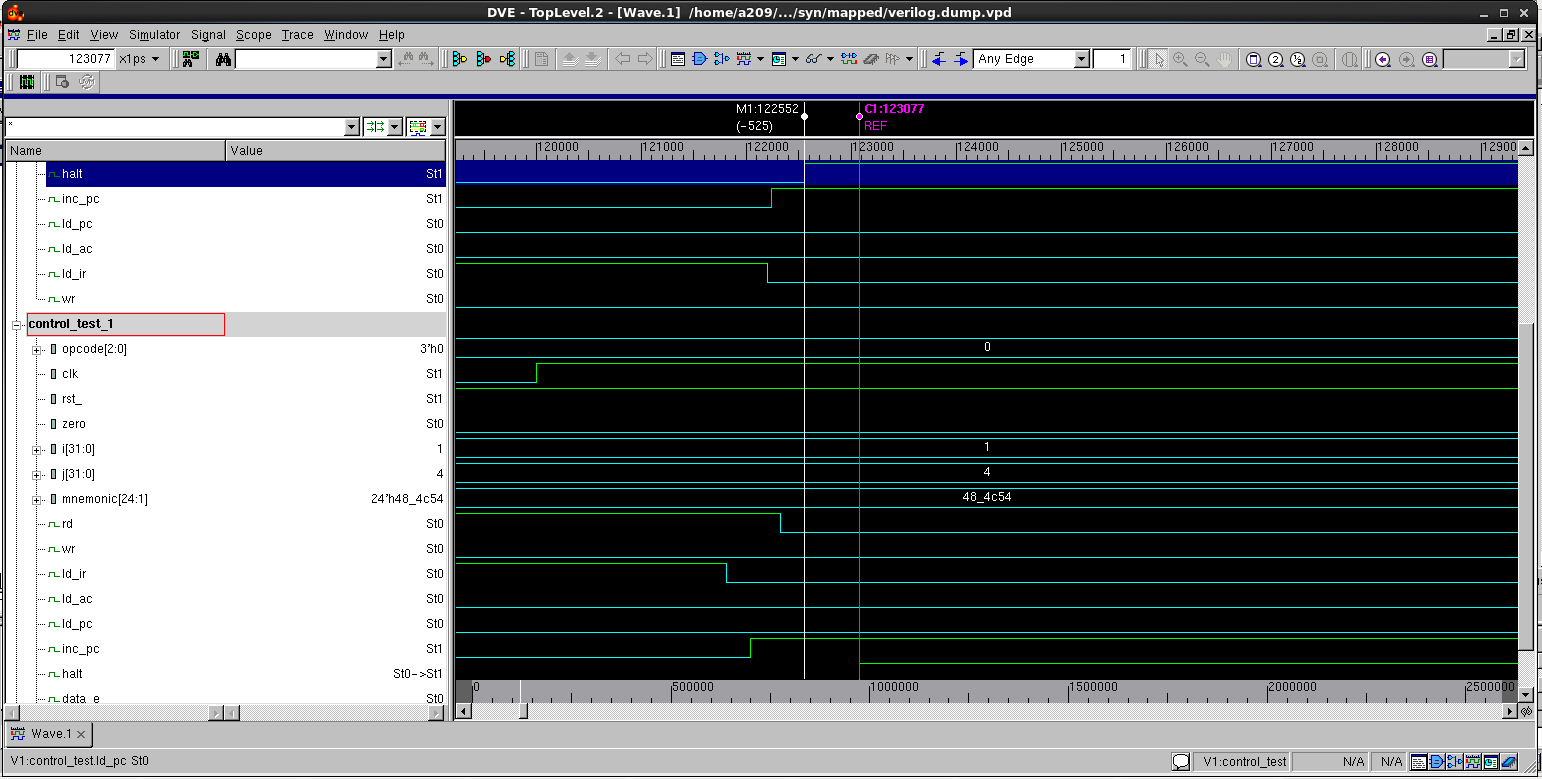
可以看到，布线之后，各基本单元之间的互连线出现在了版图中，由此完成了版图的设计（尚未验证）。

### 寄生参数导出和后仿真

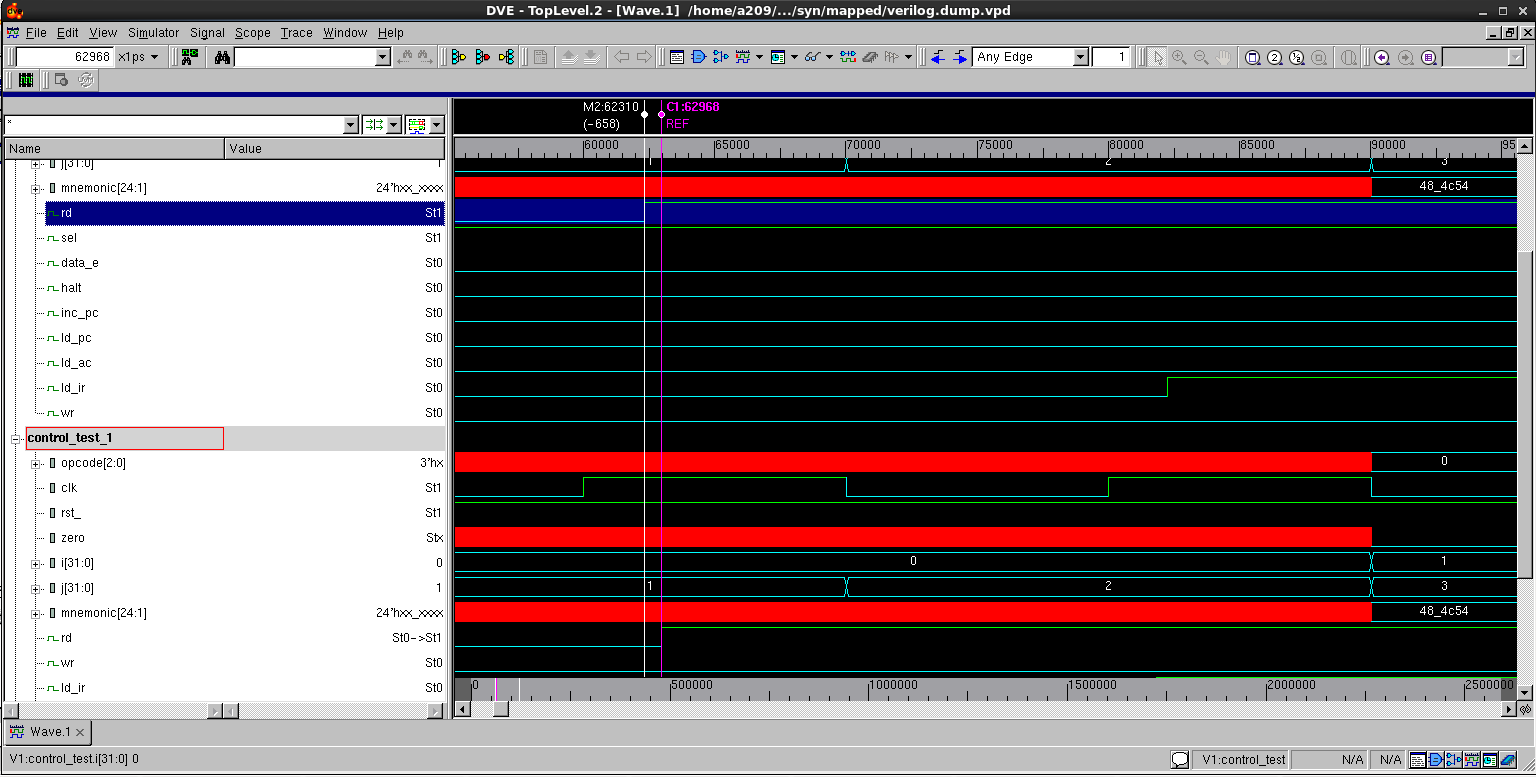
版图布线完成之后，可以导出布线之后的网表和寄生参数（电路中的寄生电阻，寄生电容），寄生参数文件为.spef文件。由寄生参数文件可以计算得出芯片在最佳、典型、最差三种条件下的延迟，结果由.sdf.文件表示。此文件与综合后仿真时反标到门级网表的时序描述文件类型相同，区别在于这里的时序描述文件由版图结构直接计算得出，而综合后仿真时采用的时序描述文件来自于工艺库中各单元的延时信息。所以利用这里的.sdf时序描述文件反标到网表中进行后仿真得到的结果与实际结果更为接近。



可见，版图电路功能与设计功能相符合



**halt 信号上升沿在两个后仿真中的时间差**



**rd 信号上升沿在两个后仿真中的时间差**

可以看出，对版图设计进行后仿真的信号时延与综合后仿真有差别，例如光标标出的halt信号，版图仿真波形中，其上升沿出现在122.552ns处，比综合后仿真的123.077ns要早。对于第二张截图中的rd信号，版图仿真波形上升沿出现在62.31ns处，早于综合后仿真的62.968ns。可见电路实际延迟要小于综合后仿真得到的延迟。

## 实验总结

这学期的专业实验，我们利用Synopsys工具完成了一个8位CPU的RTL级设计，并将其中的状态控制器模块进行了综合和版图设计。我理解了数字集成电路设计的各个阶段的工作内容：掌握了利用Verilog语言描述大规模集成电路、并编写测试代码进行测试检验的技能；对RTL级语言到门级网表的逻辑综合过程、门级网表到版图的物理综合过程有了深入体会和理解。

通过这次专业实验，我对微电子学科的研究有了新的认识和体会；专业素养大大提升，为今后在此方向更深层次的学习研究奠定了基础。