

**电子工程学院**

**ASIC专业实验报告**



**班级：2016211205**

**姓名： 何健豪**

**学号：2016210893**

**班内序号：03**

目录

[第一部分 语言级仿真 2](#_Toc11349713)

[LAB 1：选择器设计 2](#_Toc11349714)

[LAB 2：程序计数器设计 5](#_Toc11349715)

[LAB 3：指令寄存器设计 9](#_Toc11349716)

[LAB 4：算术逻辑单元设计 13](#_Toc11349717)

[LAB 5：存储器设计 15](#_Toc11349718)

[LAB 6：设计时序逻辑时采用阻塞赋值与非阻塞赋值的区别 17](#_Toc11349719)

[LAB 7：状态控制机设计 19](#_Toc11349720)

[LAB 8：CPU集成设计及验证 25](#_Toc11349721)

[第二部分 电路综合 32](#_Toc11349722)

[第三部分 版图设计 38](#_Toc11349723)

**课程主要目标：**

本实验课总共分为大致三个部分

1. 使用Verilog语言完成 CPU 芯片的各个模块进行RTL级设计、仿真以及测试。包括LAB 1 到LAB 8 部分
2. 在完成了RTL级设计和功能验证后，利用综合工具和逻辑库对电路进行逻辑综合获得门级网表，并完成门级电路仿真
3. 对于电路综合结果进行布局布线，完成版图设计。

# 第一部分 语言级仿真

## LAB 1：选择器设计

1. **功能分析：**

地址多路选择器，有两种不同的指令执行状况：顺序执行和跳转执行。CPU需要区分指令地址的源头，当指令顺序执行时，PC地址自增2，即pc\_addr；当跳转执行时，下一条指令地址为ir\_addr，多路选择器在控制信号sel控制下，判断时pc\_addr还是ir\_addr，将下一条指令的地址送到存储器中，存储器再将下一条指令通过数据总线传到指令寄存器中。

1. **实验目的**

掌握基本组合逻辑电路的实现方法。

掌握二选一开关的基本原理及在CPU中的应用

1. **实验原理**

本实验设计的是一个可综合的二选一开关，实现的逻辑功能是当sel = 0时，则out = a，否则out = b。

1. **模块源代码：**

mux.v

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

parameter size=1;

output[size-1:0] out;

input[size-1:0]b,a;

input sel;

assign out = (!sel)?a:--二选一描述

(sel)?b:

{size{1'bx}};

endmodule

测试部分：

**mux\_test.v**

`define width 8

`timescale 1 ns/1 ns

module mux\_test;

reg[`width:1]a,b;

wire[`width:1]out;

reg sel;

scale\_mux#(`width)m1(.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}};

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

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

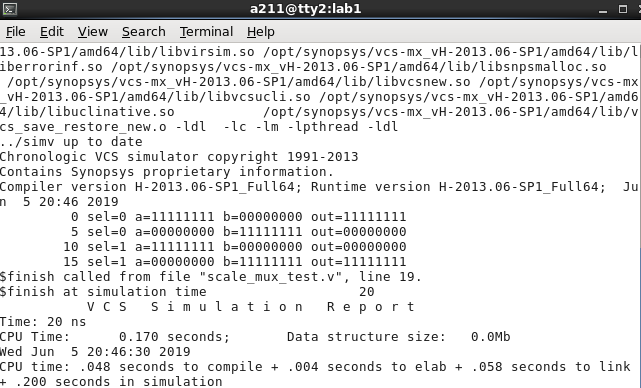
#5sel=1;b={`width{1'b1}};a={`width{1'b0}};

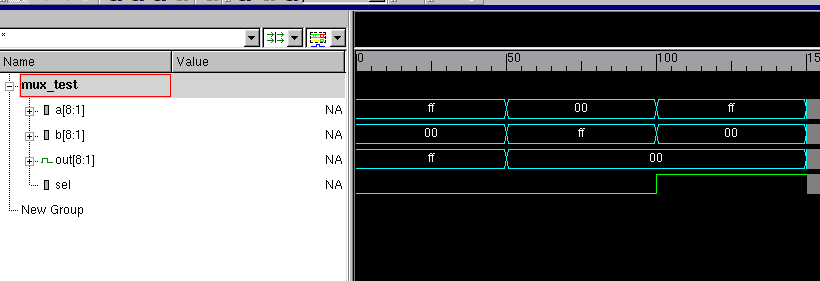
#5 $finish;

end

endmodule

1. **仿真结果与波形**

****



分析：可见，当sel = 0时，out = a，当sel=1时，out=b。

## LAB 2：程序计数器设计

1. **功能分析**

程序计数器，是一个存储状态单元，存放指令的地址，通过这个地址向存储器提取指令。顺序执行一条指令后，PC指向的地址自增2（每条指令两个字节），指向下一条指令。如果是转移指令，CPU控制器输出ld\_pc信号，通过load口将目标地址ir\_addr送入PC。

1. **实验目的**

掌握基本时序逻辑电路的实现。

掌握程序计数器的基本工作原理及其在CPU中的应用

1. **实验原理**

功能如下：当时钟上升沿或下降沿处，如果信号rst\_为低电平，则输出cnt为0；如果如果信号rst\_为高电平，当load=1时，输出cnt为总线上的数据，当load=0时，输出cnt自动加1。

在Verilog HDL中，相对于组合逻辑电路，时序逻辑电路也有规定的表述方式。在可综合的Verilog HDL模型中，我们常使用always块和@(posedge clk)或@(negedge clk)的结构来表述时序逻辑。

在always块中，被赋值的信号都必须定义为reg型，这是由时序逻辑电路的特点所决定的对于reg型数据，如果未对它进行赋值，仿真工具会认为它是不定态。为了正确地观察到仿真结果，在可综合的模块中我们通常定义一个复位信号rst-，当它为低电平时对电路中的寄存器进行复位。

1. **源代码**

counter.v

`timescale 1 ns/100 ps

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

output[4:0]cnt ;

input [4:0]data;

input clk;

input rst\_;

input load;

reg [4:0]cnt;

always@(posedge clk or negedge rst\_)—异步清零

if(!rst\_)

cnt<=0;

else

if(load)

cnt<= data;

else

cnt<= cnt + 1;

endmodule

counter\_test.v

`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;

#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;@(negedge clk)expect(5'h00);

{rst\_,load,data}=7'b1\_1\_11101;@(negedge clk)expect(5'h1D);

{rst\_,load,data}=7'b1\_0\_11101;

repeat(5)@(negedge clk);

expect(5'h02);

{rst\_,load,data}=7'b1\_1\_11111;@(negedge clk)expect(5'h1F);

{rst\_,load,data}=7'b0\_X\_XXXXX;@(negedge clk)expect(5'h00);

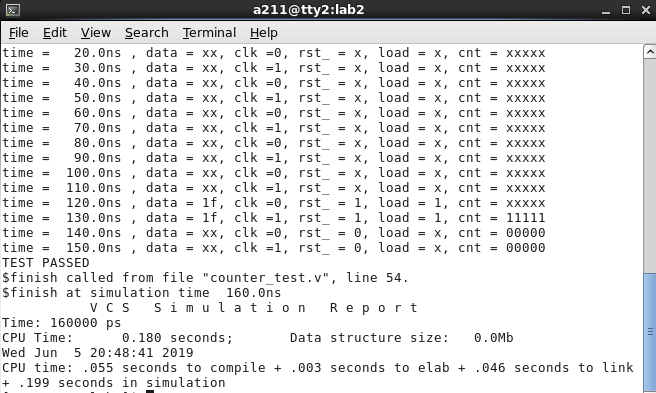
$display("TEST PASSED");

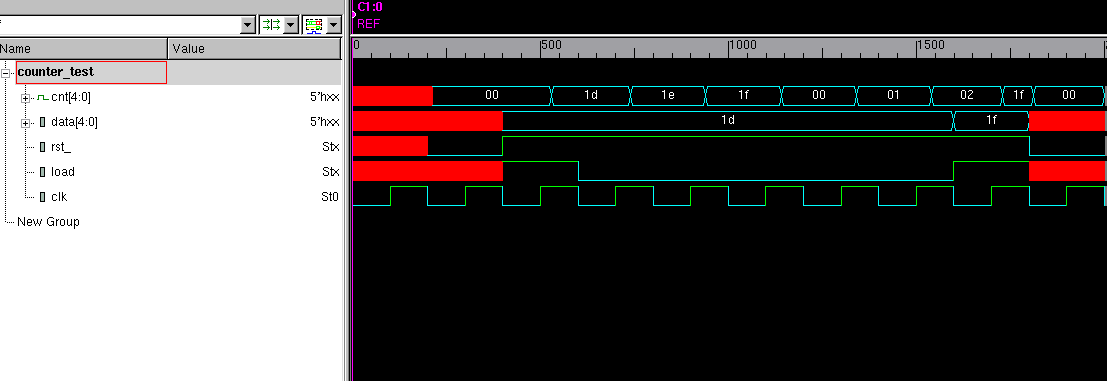
$finish;

end

endmodule

1. **仿真结果与波形**

****



分析：如果信号rst\_为低电平，则输出cnt为0；如果如果信号rst\_为高电平，当load=1时，输出cnt为总线上的数据，当load=0时，输出cnt自动加1。

1. **思考题**

该电路中，rst-是同步还是异步清零端？

在always块中将rst作为敏感信号，触发后直接清零与时钟无关。所以是异步清零端。

## LAB 3：指令寄存器设计

1. **功能分析**

指令寄存器，用来暂时存放当前正在执行指令的内容，当执行一条指令时，CPU控制器通过ld\_ir信号控制将指令内容从存储器中取出，通过数据总线读取到指令寄存器中。

1. **实验目的**

掌握基本时序逻辑电路的设计原理

掌握寄存器的基本原理及其在CPU中的应用

1. **实验原理**

每一位寄存器由一个二选一MUX和一个触发器dffr组成，当load=1，装载数据；当load=0，寄存器保持。

对于处理重复的电路，可用数组来描述

1. **源代码**

clock.v

`timescale 1 ns /1 ns

module clock(clk);

reg clk;

output clk;

initial begin

clk=0;

forever begin --时钟周期20ns

#10 clk=1'b1;

#10 clk=1'b0;

end

end

endmodule

mux及dffr模块调用代码

mux mux7 --8位移位寄存器

(.out(n1[7]),

.sel(load),

.b(data[7]),

.a(out[7])

);

dffr dffr7

(.q(out[7]),

.d(n1[7]),

.clk(clk),

.rst\_(rst\_)

);

mux mux6

(.out(n1[6]),

.sel(load),

.b(data[6]),

.a(out[6])

);

dffr dffr6

(.q(out[6]),

.d(n1[6]),

.clk(clk),

.rst\_(rst\_)

);

mux mux5

(.out(n1[5]),

.sel(load),

.b(data[5]),

.a(out[5])

);

dffr dffr5

(.q(out[5]),

.d(n1[5]),

.clk(clk),

.rst\_(rst\_)

);

mux mux4

(.out(n1[4]),

.sel(load),

.b(data[4]),

.a(out[4])

);

dffr dffr4

(.q(out[4]),

.d(n1[4]),

.clk(clk),

.rst\_(rst\_)

);

mux mux3

(.out(n1[3]),

.sel(load),

.b(data[3]),

.a(out[3])

);

dffr dffr3

(.q(out[3]),

.d(n1[3]),

.clk(clk),

.rst\_(rst\_)

);

mux mux2

(.out(n1[2]),

.sel(load),

.b(data[2]),

.a(out[2])

);

dffr dffr2

(.q(out[2]),

.d(n1[2]),

.clk(clk),

.rst\_(rst\_)

);

mux mux1

(.out(n1[1]),

.sel(load),

.b(data[1]),

.a(out[1])

);

dffr dffr1

(.q(out[1]),

.d(n1[1]),

.clk(clk),

.rst\_(rst\_)

);

mux mux0

(.out(n1[0]),

.sel(load),

.b(data[0]),

.a(out[0])

);

dffr dffr0

(.q(out[0]),

.d(n1[0]),

.clk(clk),

.rst\_(rst\_)

);

例化寄存器

register r1

(

.data(data),

.out(out),

.load(load),

.clk(clk),

.rst\_(rst\_)

);

例化时钟

clock c1

(

.clk(clk)

);

添加检测信号

initial

begin

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

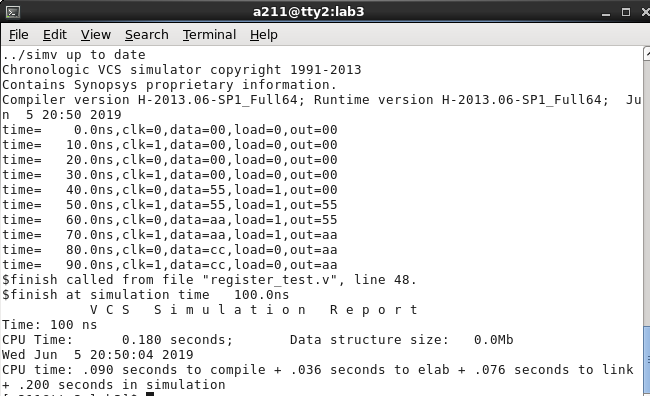
$monitor("time=%t,clk=%b,data=%h,load=%b,out=%h",

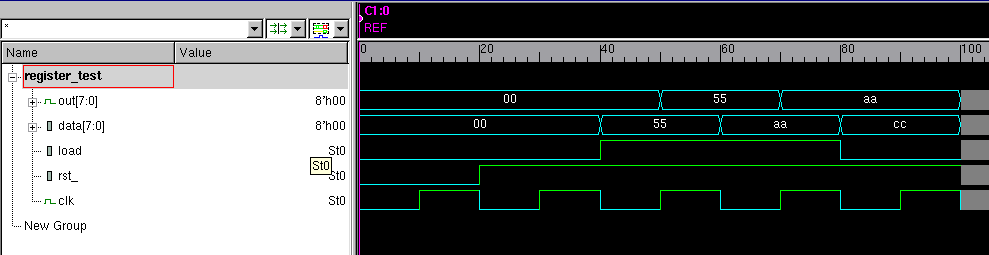
$stime,clk,data,load,out);

$dumpvars(2,register\_test);

end

1. **仿真结果与波形**





分析：可见当load=1，装载数据；当load=0，寄存器保持。

## LAB 4：算术逻辑单元设计

1. **功能分析**

算术逻辑单元，与累加器配合进行相加、相与，异或等算术逻辑运算。

1. **实验目的**

掌握用always实现组合逻辑电路的方法；

了解assign与always两种组合逻辑电路实现方法之间的区别。

掌握ALU的基本原理及其在CPU中的应用

1. **实验原理**

本实验描述的是一个简单的ALU指令译码电路的设计示例。它通过对指令的判断，对输入数据执行相应的操作，包括加、减、或和传数据，并且无论是指令作用的数据还是指令本身发生变化，结果都要做出及时的反应。

当accum=0,zero=1;否则zero-0;

示例中使用了电平敏感的always块，电平敏感的触发条件是指在@后括号内电平列表的任何一个电平发生变化就能触发always块的动作，并且运用了case结构来进行分支判断。

在always中适当运用default（在case结构中）和else（子if…else结构中），通常可以综合为纯组合逻辑，尽管被赋值的变量一定要定义为reg型。如果不使用default或else对缺省项进行说明，易产生意想不到的锁存器。

1. **源代码**

电路描述

补充描述部分

always@(data,accum,opcode)

begin

if(accum==8'h00)

zero=1;

else

zero=0;

case (opcode)—case语句实现

PASS0:out=accum;//HLT

PASS1:out=accum;//SKZ

ADD:out=accum+data;

AND:out=accum&data;

XOR:out=accum^data;

PASSD:out=data;//LDA

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

PASS7:out=accum;//JMP

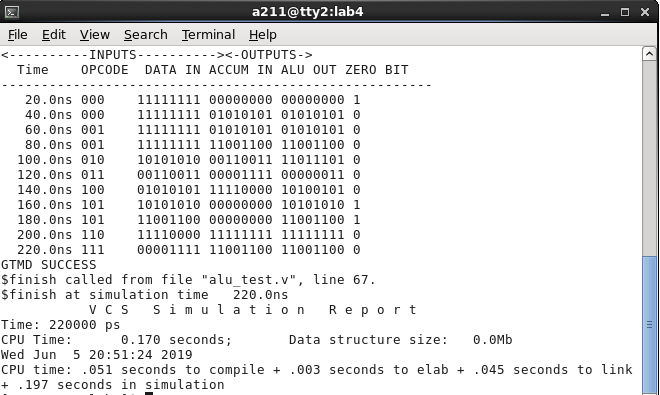
default: out=8'bx;

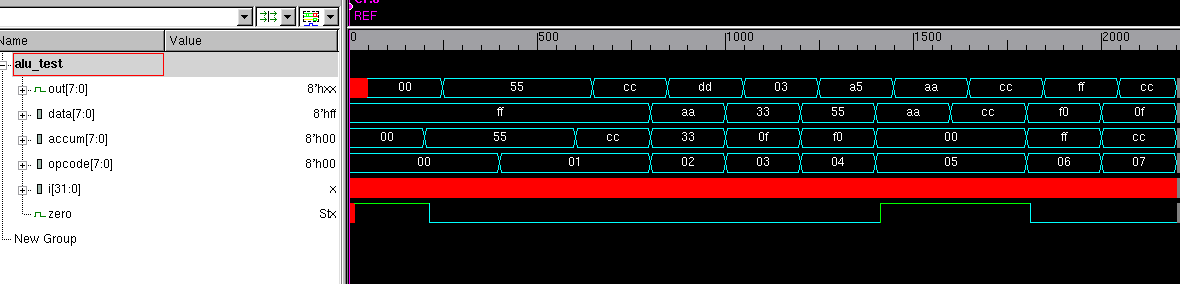
endcase

end

endmodule

1. **仿真结果与波形**

****



分析：测试ALU的逻辑运算功能，遍历PASSA,PASSD,ADD,AND,XOR等逻辑运算功能，波形可反应正确的结果。

## LAB 5：存储器设计

1. **功能分析**

存储器，存储程序指令，设计的CPU为8位RISC\_CPU，指令地址为4：0共5位，存储器大小8\*32bit。存储器为并行设计，能够读出read信号与write信号。

1. **实验目的**

掌握存储器存储程序和数据的双向读写的基本工作原理

掌握存储器在CPU中的应用

1. **实验原理**

当read为高电平时，读出memory的数据到data总线上；在wire的上升沿，将data总线的数据写入memory。

本实验中，设计一个模块名为mem的存储器仿真模型，该存储器具有双线数据总线及异步处理功能。由于数据是双向的，所以要注意，对memory的读写在时序上要错开。

1. **源代码**

**自行添加的代码**

assign data= (read)?memory[addr]:8'hZ;--read高电平，数据从memory到data总线

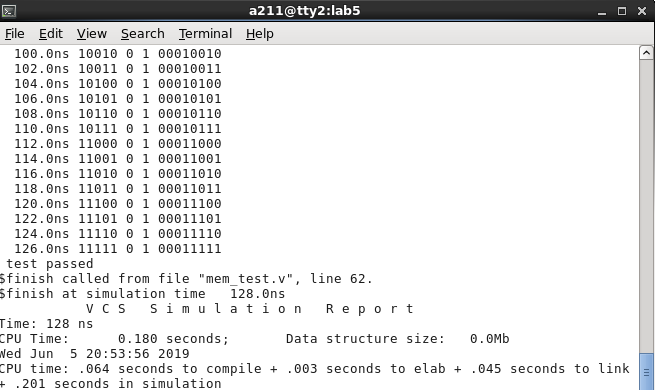
always @(posedge write)—在write上升沿，将data总线上的数据写入memory

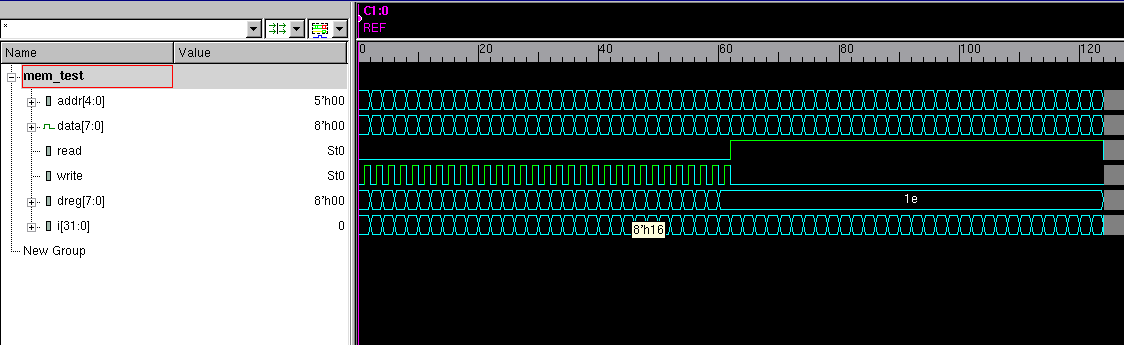
begin

memory[addr]<=data[7:0];

end

1. **仿真结果与波形**

****



**分析：**当read为高电平时，读出memory的数据到data总线上；在wire的上升沿，将data总线的数据写入memory。

## LAB 6：设计时序逻辑时采用阻塞赋值与非阻塞赋值的区别

1. **实验目的**

明确掌握阻塞赋值与非阻塞赋值的概念和区别；

了解阻塞赋值的应用场景

1. **实验原理**

在always块中，阻塞赋值可以理解为赋值语句是顺序执行的，而非阻塞赋值可以理解为并发执行的。实际时序逻辑设计中，一般情况下非阻塞赋值语句被更多的使用，有时为了在同一周期实现相互关联的操作，也使用阻塞赋值语句。

1. **源代码**

blocking.v

`timescale 1 ns/ 100 ps

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;

c =b;

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

end

endmodule

non\_blocking.v

`timescale 1 ns/ 100 ps

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;

c<=b;

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

end

endmodule

compareTop.v

`timescale 1 ns/ 100 ps

module compareTop;

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

reg[3:0]a;

reg clk;

initial

begin

clk=0;

forever #50 clk=~clk;

end

initial

$dumpvars (2,compareTop);

initial

begin

a=4'h3;

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

# 100 a =4'h7;

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

# 100 a =4'hf;

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

# 100 a =4'ha;

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

# 100 a =4'h2;

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

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

$finish;

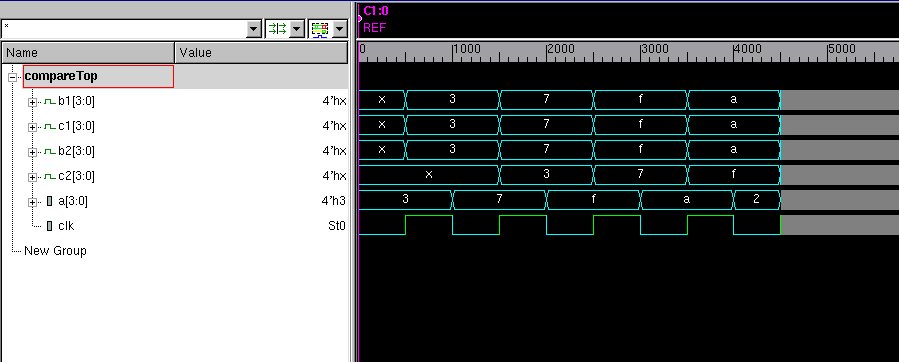
end

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

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

endmodule

1. **仿真结果与波形**



## LAB 7：状态控制机设计

1. **功能分析**

状态控制机，用来控制PC,ALU,IR,MEMORY等部件的协调运作，控制何时启动，停止部件，何时读取指令，写入数据等操作。CPU在8个周期内完成指令的或许与执行，前4个周期用来从存储器读取数据，后4个周期发出不同的控制信号。

Clk1：存储器接收多路选择器送来的指令地址。

Clk2:存储器从数据总线读取指令

Clk3：IR读取指令。

Clk4:IR接着读取指令，8位RISC\_CPU指令读取需要2个时钟周期

Clk5:CPU输出inc\_pc信号到程序计数器，halt=HLT,如果hlt高电平，执行暂行操作。如果HLT低电平,pc自增1.

Clk6:逻辑运算判断，rd=aluop，当执行ADD,AND,XOR,LOA操作时,aluop高电平；当执行HLT,SKZ,STO,JMP操作时，aluop为低电平。

Clk7与CLK8:判断并执行SKZ,JMP,LDA、STO等具体指令，参考具体代码实现

1. **实验目的**

掌握利用有限状态机(FSM)实现复杂时序逻辑的方法。

掌握状态控制器的基本原理及其在CPU中的应用。

1. **实验原理**

控制器是CPU的控制核心，用于产生一系列的控制信号，启动或停止某些部件。CPU何时进行读指令，何时进行RAM和I/O端口的读写操作等，都由控制器来控制。

1. **源代码**

**补充代码**

nexstate<=state+1'h01;--描述状态转移

case(state)—判断CPU控制信号

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

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

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

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

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

if(opcode==`HLT)—halt=1则执行暂停操作

halt=1;

end

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

7:begin sel=0;rd=alu\_op;ld\_ir=0;halt=0;data\_e=!alu\_op;ld\_ac=0;wr=0;

if(opcode==`SKZ)—SKZ指令

inc\_pc<=zero;

if(opcode==`JMP)

ld\_pc=1;

end

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

if(opcode==`JMP)--JMP

ld\_pc=1;

if(opcode==`STO)--STO

wr=1;

end

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

endcase

end

control\_test.v

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

\* TEST BENCH FOR CONTROLLER \*

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

`timescale 1 ns / 1 ns

module control\_test ;

reg [8:0] response [0:127];

reg [3:0] stimulus [0:15];

reg [2:0] opcode;

reg clk;

reg rst\_;

reg zero;

integer i,j;

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

// Instantiate controller

control c1

(

rd ,

wr ,

ld\_ir ,

ld\_ac ,

ld\_pc ,

inc\_pc ,

halt ,

data\_e ,

sel ,

opcode ,

zero ,

clk ,

rst\_

);

// Define clock

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 ) ;

end

// Apply stimulus

initial

begin

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

rst\_=1;

@ ( negedge clk ) rst\_ = 0 ;

@ ( negedge clk ) rst\_ = 1 ;

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

@ ( posedge ld\_ir )

@ ( negedge clk )

{ opcode, zero } = stimulus[i] ;

end

// Check response

initial

begin

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

@ ( posedge rst\_ )

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

@ ( negedge clk )

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] )

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" ) ;

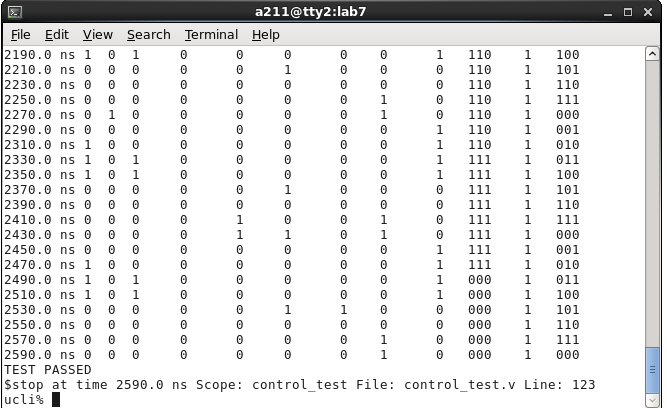
$stop;

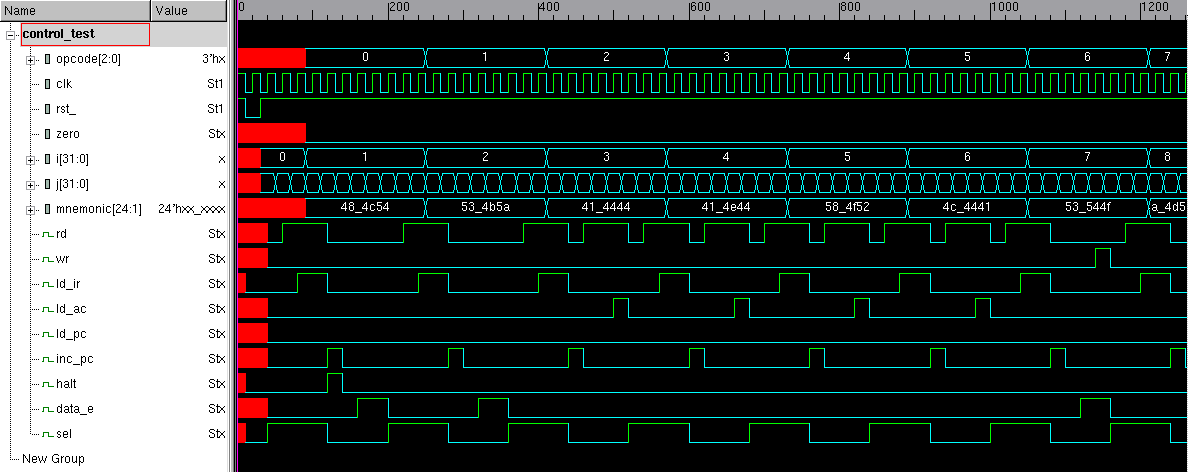
$finish ;

end

endmodule

1. **仿真结果与波形**

****



分析：遍历了各种逻辑功能，波形可反应正确的结果

## LAB 8：CPU集成设计及验证

1. **功能分析：**

此部分通过例化调用之前设计并验证过的计数器、选择器等CPU各部分组件，通过测试文件对CPU的RTL级设计进行语法以及功能验证，最终完成前仿真。

1. **实验目的**

学习和使用层次化、结构化设计方法。

掌握CPU的基本原理与设计思想。

1. **实验原理**

大规模集成电路往往由模块多层次的引用和组合构成。层次化，结构化的设计过程使复杂系统更容易控制和调试。Verilog HDL中，上层模块引用下层模块与C语言中程序调用有些类似，被引用的子模块在综合时作为其父模块的一部分被综合，形成相应的电路结构。

在进行模块实例引用时，必须注意的是模块之间对应的端口，即子模块端口与父模块的内部信号必须是一一对应。同时调用的模块名与设计时的模块名也必须一致，否则会出现调用失败。导致功能无法实现。

在语法仿真通过后，本实验采用运行使用CPU基本指令集编写的程序段，通过验证是否正确实现了测试程序段的功能，从而验证CPU功能是否正确。

1. **源代码**

**CPUtest1.dat**

**Test1主要功能是将CPU的指令做了一个遍历，如果系统在哪一行停止，则可据此推测出现问题的功能从而进行修正。**

//opcode\_operand // addr assembly code

//-------------- // ---- -----------------------------------------------

@00 111\_11110 // 00 BEGIN: JMP TST\_JMP

000\_00000 // 01 HLT //JMP did not work at all

000\_00000 // 02 HLT //JMP did not load PC, it skipped

101\_11010 // 03 JMP\_OK: LDA DATA\_1

001\_00000 // 04 SKZ

000\_00000 // 05 HLT //SKZ or LDA did not work

101\_11011 // 06 LDA DATA\_2

001\_00000 // 07 SKZ

111\_01010 // 08 JMP SKZ\_OK

000\_00000 // 09 HLT //SKZ or LDA did not work

110\_11100 // 0A SKZ\_OK: STO TEMP //store non-zero value in TEMP

101\_11010 // 0B LDA DATA\_1

110\_11100 // 0C STO TEMP //store zero value in TEMP

101\_11100 // 0D LDA TEMP

001\_00000 // 0E SKZ //check to see if STO worked

000\_00000 // 0F HLT //STO did not work

100\_11011 // 10 XOR DATA\_2

001\_00000 // 11 SKZ //check to see if XOR worked

111\_10100 // 12 JMP XOR\_OK

000\_00000 // 13 HLT //XOR did not work at all

100\_11011 // 14 XOR\_OK: XOR DATA\_2

001\_00000 // 15 SKZ

000\_00000 // 16 HLT //XOR did not switch all bits

000\_00000 // 17 END: HLT //CONGRATULATIONS - TEST1 PASSED!

111\_00000 // 18 JMP BEGIN //run test again

@1A 00000000 // 1A DATA\_1: //constant 00(hex)

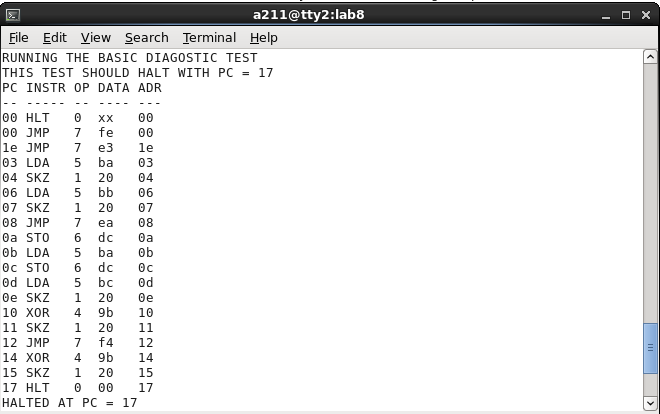
11111111 // 1B DATA\_2: //constant FF(hex)

10101010 // 1C TEMP: //variable - inititially AA(hex)

@1E 111\_00011 // 1E TST\_JMP: JMP JMP\_OK

000\_00000 // 1F HLT //JMP is broken

Test1验证通过：



指针正确停在了17

**CPUtest2.dat**

**Test2主要功能是将3个data数据进行与，异或运算，看CPU是否能完成正确的跳转。**

//opcode\_operand // addr assembly code

//-------------- // ---- -----------------------------------------------

@00 101\_11011 // 00 BEGIN: LDA DATA\_2

011\_11100 // 01 AND DATA\_3

100\_11011 // 02 XOR DATA\_2

001\_00000 // 03 SKZ –得零所以跳过下一条指令

000\_00000 // 04 HLT //AND doesn't work

010\_11010 // 05 ADD DATA\_1

001\_00000 // 06 SKZ

111\_01001 // 07 JMP ADD\_OK

000\_00000 // 08 HLT //ADD doesn't work

100\_11100 // 09 XOR DATA\_3

010\_11010 // 0A ADD DATA\_1 //FF plus 1 makes -1

110\_11101 // 0B STO TEMP

101\_11010 // 0C LDA DATA\_1

010\_11101 // 0D ADD TEMP //-1 plus 1 should make zero

001\_00000 // 0E SKZ

000\_00000 // 0F HLT //ADD Doesn't work

000\_00000 // 10 END: HLT //CONGRATULATIONS - TEST2 PASSED!

111\_00000 // 11 JMP BEGIN //run test again

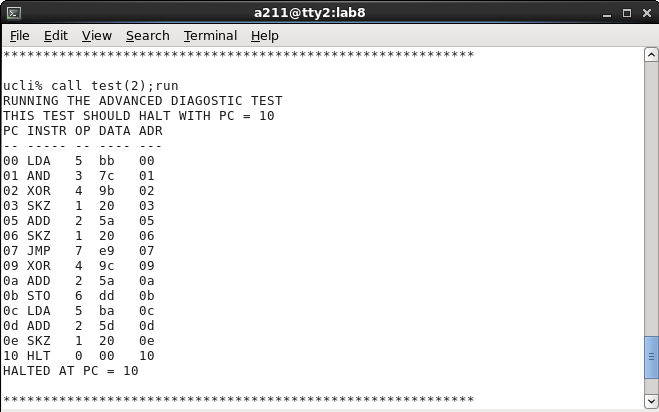
@1A 00000001 // 1A DATA\_1: //constant 1(hex)

10101010 // 1B DATA\_2: //constant AA(hex)

11111111 // 1C DATA\_3: //constant FF(hex)

00000000 // 1D TEMP:

Test2验证通过



指针正确停在了10.

**CPUtest3.dat**

**Test3主要功能是求一个Fibonacci数列，正确显示到144**

**遇到的主要问题是数列重复将每个数字进行了重复输出，错误出在testbench的输出部分，将输出敏感操作符错误定义，斐波那契数列每进行一次相加才会得到新的项，所以将输出敏感的操作符改为ADD即opcode=2，数列输出正确。**

//opcode\_operand // addr assembly code

//-------------- // ---- ----------------------------------------------------

111\_00011 // 00 JMP LOOP //jump to the address of LOOP

@03 101\_11011 // 03 LOOP: LDA FN2 //load value in FN2 into accum

110\_11100 // 04 STO TEMP //store accumulator in TEMP

010\_11010 // 05 ADD FN1 //add value in FN1 to accumulator

110\_11011 // 06 STO FN2 //store result in FN2

101\_11100 // 07 LDA TEMP //load TEMP into the accumulator

110\_11010 // 08 STO FN1 //store accumulator in FN1

100\_11101 // 09 XOR LIMIT //compare accumulator to LIMIT

001\_00000 // 0A SKZ //if accum = 0, skip to DONE

111\_00011 // 0B JMP LOOP //jump to address of LOOP

000\_00000 // 0C DONE: HLT //end of program

101\_11111 // 0D AGAIN: LDA ONE

110\_11010 // 0E STO FN1

101\_11110 // 0F LDA ZERO

110\_11011 // 10 STO FN2

111\_00011 // 11 JMP LOOP //jump to address of LOOP

@1A 00000001 // 1A FN1: //variable - stores 1st Fib. No.

00000000 // 1B FN2: //variable - stores 2nd Fib. No.

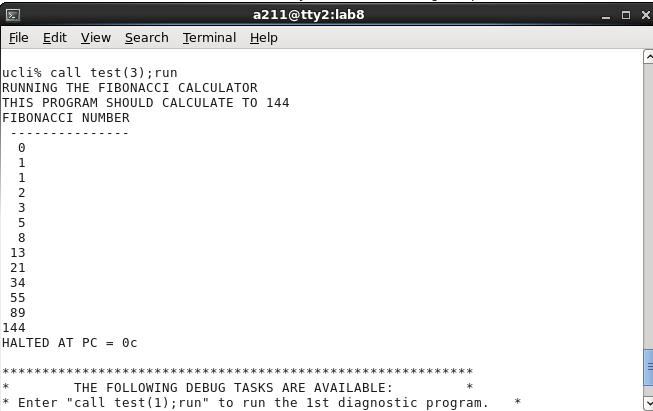
00000000 // 1C TEMP: //temporary variable

10010000 // 1D LIMIT: //constant 144 - max value

00000000 // 1E ZERO: //constant 0

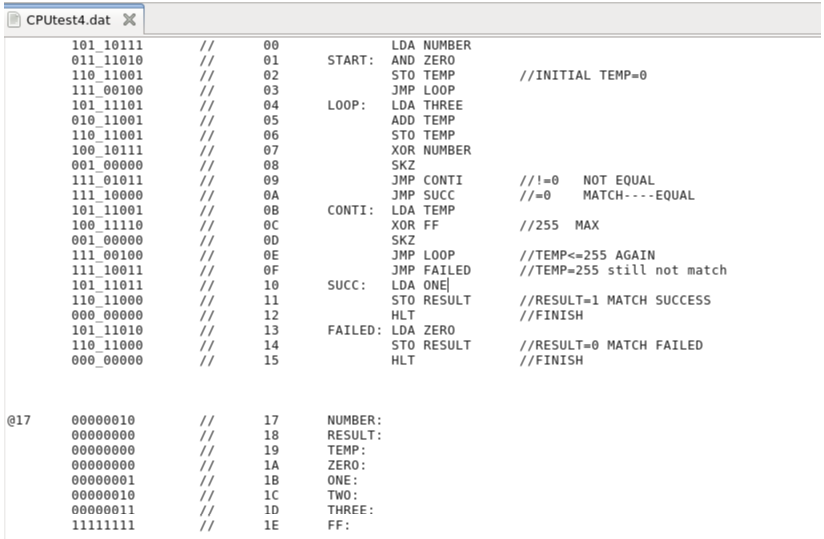
00000001 // 1F ONE: //constant 1

**Test3验证通过**

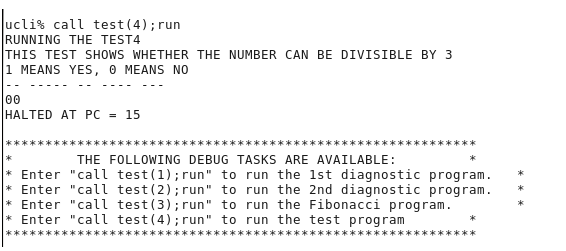


**CPUtest4.dat 为自己编写的CPUtest，实现3的倍数判断。**

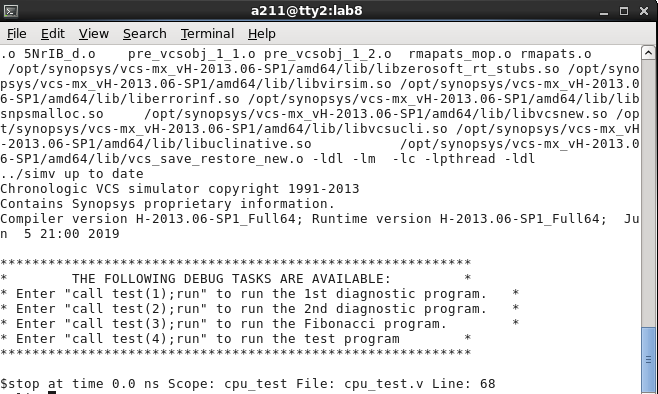
遍历了所有的指令，通过不断加3直至溢出的方式，遍历数据范围内3的倍数并与待判断目标数进行比较，如果相等则跳出循环。由于命令集简单功能不够强大，算法的进一步优化较为困难。具体的实现逻辑见注释。

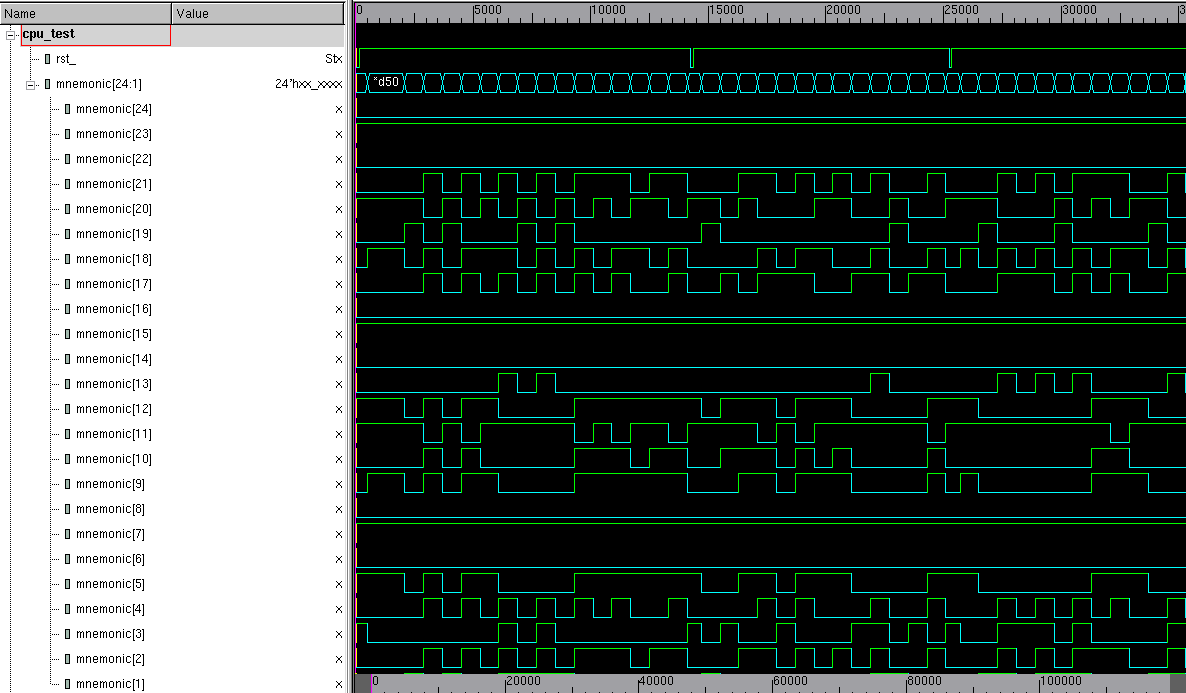


**Test4验证通过**



1. **RTL仿真结果与波形**

****



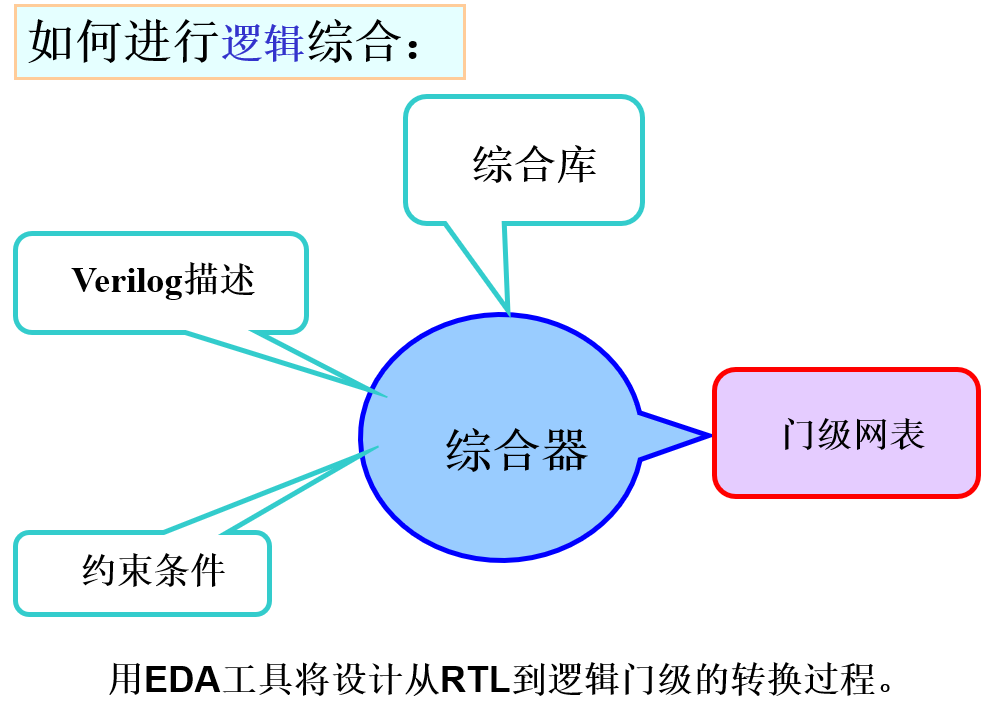
**分析：此为没有延时信息的前仿真波形。**

# 第二部分 电路综合

1. **实验目的**

利用综合工具和厂家的逻辑库对电路进行逻辑综合，获得与特定工艺相关的门级网表。熟悉采用Design Compiler进行逻辑综合的基本方法，获得门级电路后仿真。

逻辑综合的目的是产生电路门级结构的同时寻求时序、面积、功耗的平衡。



综合可以分成三个部分：Synthesis = translation + mapping + optimize

转换Translation: 将HDL转化为通用的、独立于工艺的通用元件库(GTECH)组成的逻辑单元。

映射Mapping：将通用元件库映射到目标单元库（target library）上，此时电路相关网表包含相关的工艺参数。

优化Optimize：按设定的延迟、面积、线负载模型等综合约束条件，对电路网表进一步优化。

1. **实验内容**

采用SYNOPSYS公司的综合工具Design Compiler对实验7的control.v做综合。

综合的过程主要分成三个过程：

* 1. 代码转化：将源代码转化为逻辑门电路。
  2. 逻辑优化：添加延时面积等约束条件对电路进行优化。
  3. 门级映射：将逻辑门电路映射到具体的工艺库上。

在综合完成之后，可以得到带延时信息的门级网表，输出结果报告，包括延时报告

面积报告和功耗报告等。

1. **实验主要步骤以及理解**
2. **首先配置DC运行所用的库文件**

首先是单元库target\_library，mapping时用的特定工艺元件库。包括元件功能、延迟、面积及其他相关信息。链接库连接到对应的库上。符号库定义了单元电路现实的Schematic库

1. **读入设计文件**

综合工具读入HTL代码并进行分析

1. **施加设计约束**

约束条件是为了在逻辑综合和优化过程中实现对设计结果的控制。约束条件不同，将综合出不同的电路结构，电路结果是与设计约束密切相关的。一般综合器中使用的约束条件包括面积、延时、功耗的约束。而对于基于时钟的数字逻辑电路来说，时序约束是最重要的。时序路径可以分为三类：输入到寄存器的路径，寄存器到寄存器的路径，寄存器到输出的路径。

芯片面积直接关系到芯片的成本，面积越大成本越高。但是芯片面积优化的代价是牺牲其他参数，因此优化的程度需要进行斟酌与多方面考虑后，根据实际情况进行决定。

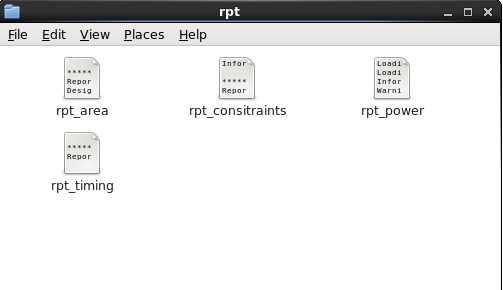
1. **定义环境属性**

在设计约束中仅仅约束了输入输出的延时，没有考虑电平转换时间和互连线的延时。同时综合模块周围环境的变化也没有考虑在内，例如温度变化或者电路的供电电压发生变化。这些方面的影响在实际电路中是不可忽视的，因此这些环境属性也必须考虑到。

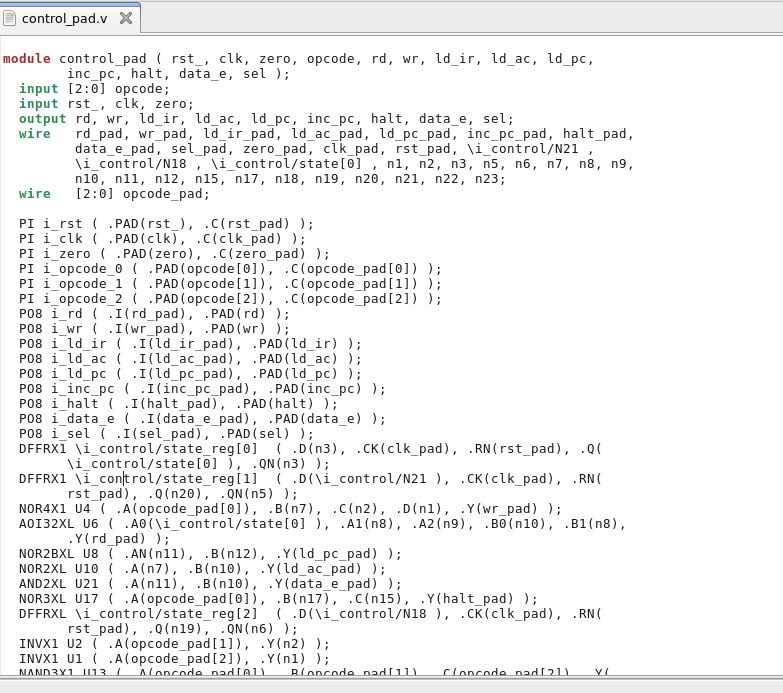
设置环境属性主要有设置工作条件、输出负载、输入驱动和连线负载模型几个方面。

1. **综合以及结果输出**

综合是在满足了各项约束条件的情况下，实现将行为级描述转化为指定的技术库中单元电路的连接，完成RTL到门级网表的转换。通过代码转化、逻辑优化、门级映射散步得到一个功能和时序都满足要求的网表，并达到面积最小化、功耗最小化和性能最大化。同时也会输出时序报告、时序约束文件、面积报告和功耗报告如下图所示

****

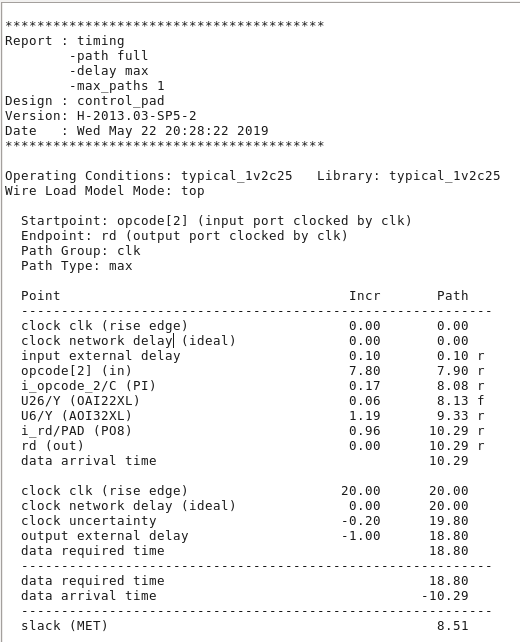
下图为综合后，门级映射后mapped文件夹中生成的门级网表



以及产生的SDF文件，作为标准延时文件，他描述了设计中的时序信息，指明了模块管脚和管脚间的延迟、时钟到数据的延迟和内部链接延迟。



1. **时序报告分析**

****

分析这个时序报告可以看到：

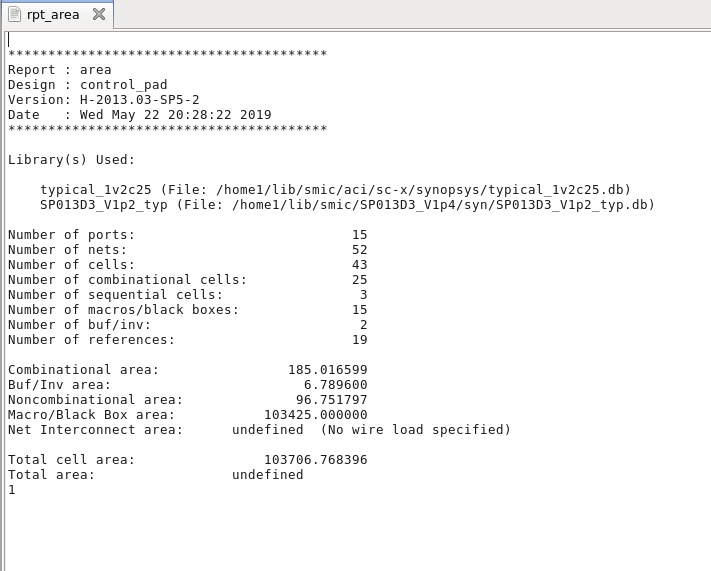
数据经过这条路径需要的时间是10.29ns

而这一条路径数据延时要求时间由时钟上升沿减去不确定因素延时，数据延时要求最多为18.80ns

两者相减最后结果为8.51 为正值

综合结果可行

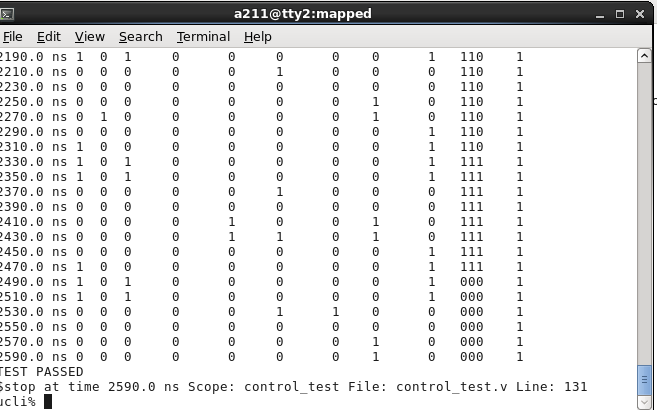
**面积报告分析**

****

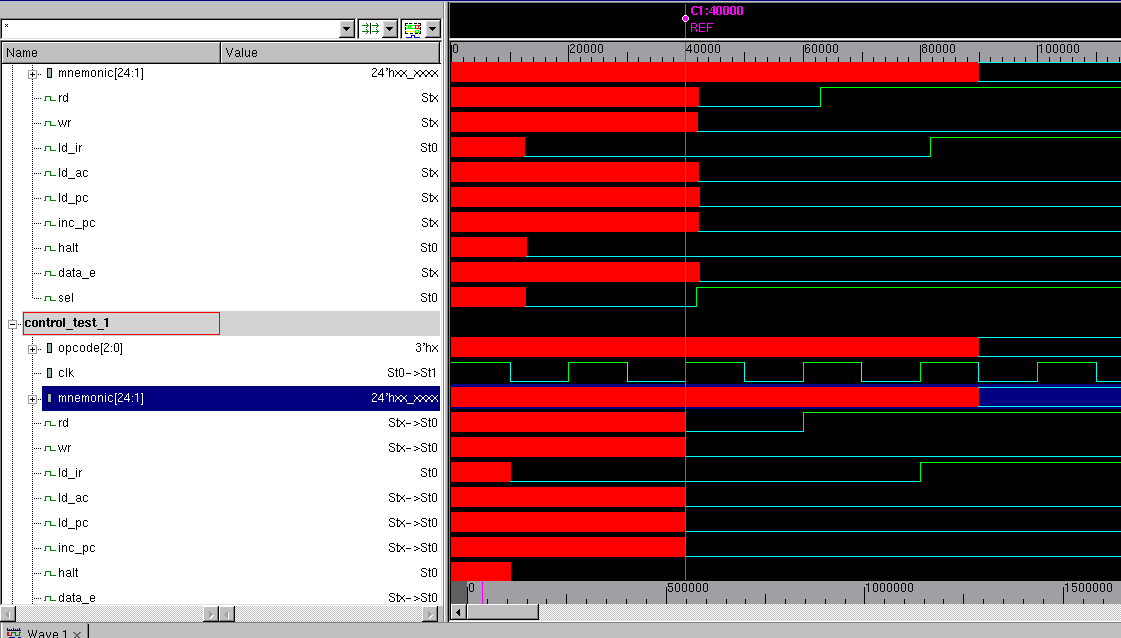
1. **综合后仿真**

为了验证综合后生成的门级网表的功能，需要进行后仿真。

**综合后仿真通过**

****

**综合后延时仿真波形：上半部分为综合后延时波形，下半部分为综合前波形**

****

1. **延时信息分析！**

综合后仿真的SDF文件里包括了设计的延时信息，在综合后仿真的波形中可以看出延时的体现。为了验证延时的正确性，我们选择rd信号作为分析对象进行计算。从输出的rd信号开始倒推关键路径，在生成的门级网表中查找到：

PO8 i\_rd ( .I(rd\_pad), .PAD(rd) );

这是第一个门，从sdf文件中查找到此原件的延时信息:

(CELL

(CELLTYPE "PO8")

(INSTANCE i\_rd)

(DELAY

(ABSOLUTE

(IOPATH I PAD (0.956:0.959:0.959) (0.944:0.974:0.974))

)

)

)

可以得知这个门的延时是0.96，与报告吻合。

而后继续倒推，下一个门是U6

(CELL

(CELLTYPE "AOI32XL")

(INSTANCE U6)

(DELAY

(ABSOLUTE

(IOPATH A0 Y (1.207:1.209:1.209) (0.592:0.592:0.592))

(IOPATH A1 Y (1.215:1.215:1.215) (0.591:0.591:0.591))

(IOPATH A2 Y (1.240:1.249:1.249) (0.601:0.603:0.603))

(IOPATH B0 Y (1.193:1.195:1.195) (0.501:0.514:0.514))

(IOPATH B1 Y (1.198:1.199:1.199) (0.469:0.469:0.469))

)

)

)

此门延时为1.19

而后下一个门是OAI22XL

以此类推，不断地沿着路径累加到时钟，即可获得此路径的总延时

计算得到总延时为10.28，与报告相吻合。

思考题

1. control\_pad.v文件是verilog语言级的描述还是结构化的描述？

是结构化的描述。

1. control\_pad.sdf文件中，对触发器的延迟包括哪些信息？

包括对逻辑单元和管脚的上升/下降时延的最大值、最小值和典型值。

# 第三部分 版图设计

1. **实验目的**

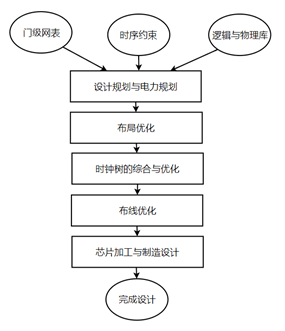
掌握版图设计的基本概念和流程，熟悉采用Sysnopsys ICC工具进行版图设计的方法。

1. **实验内容和理解**

对电路综合输出的门级网表control\_pad.v进行布局布线，完成寄生参数的导出和后仿真。

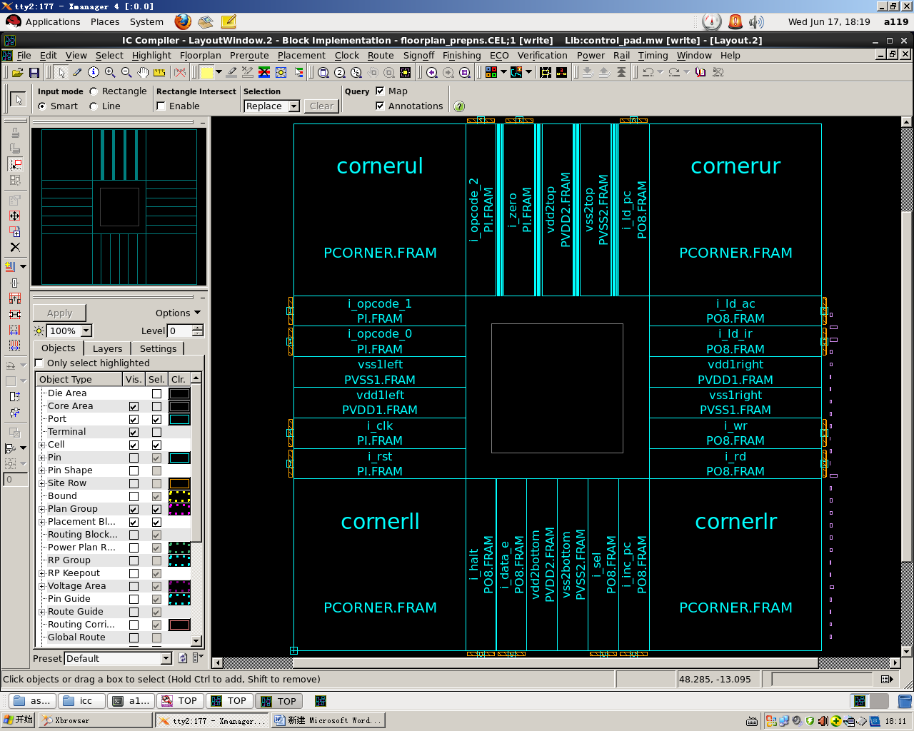
在逻辑综合得到与特定工艺库相关联的门级网表后，需要进行集成电路版图设计。版图是集成电路制造所用的掩模上的图形，各层图形对应的工艺步骤不同，每层版图对应不同图形，版图设计与工艺密切相关。

版图是根据逻辑与电路功能和性能要求以及工艺水平要求来设计光刻用的掩模版图，实现IC设计的最终输出。下图是版图的基本设计流程

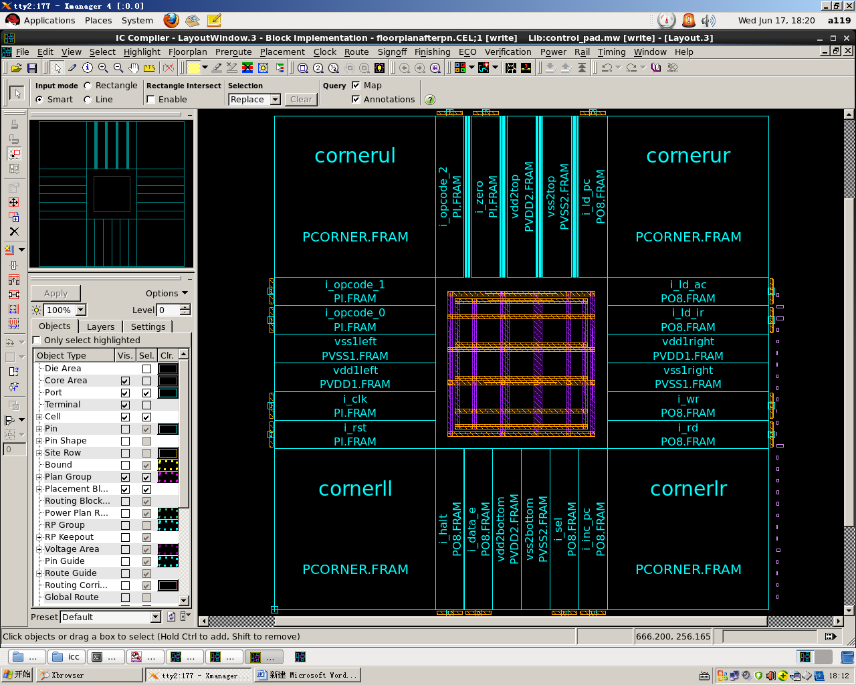


1. **实验主要步骤**
2. **布局规划**

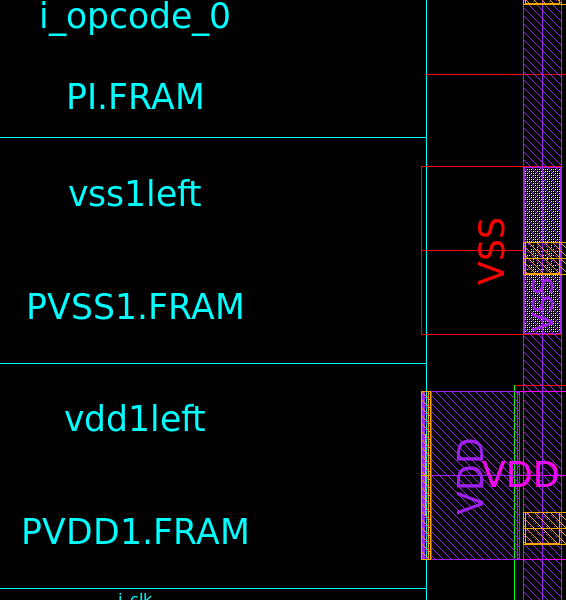
未产生core ring和mesh前的图形界面



产生core ring和mesh后的图形界面



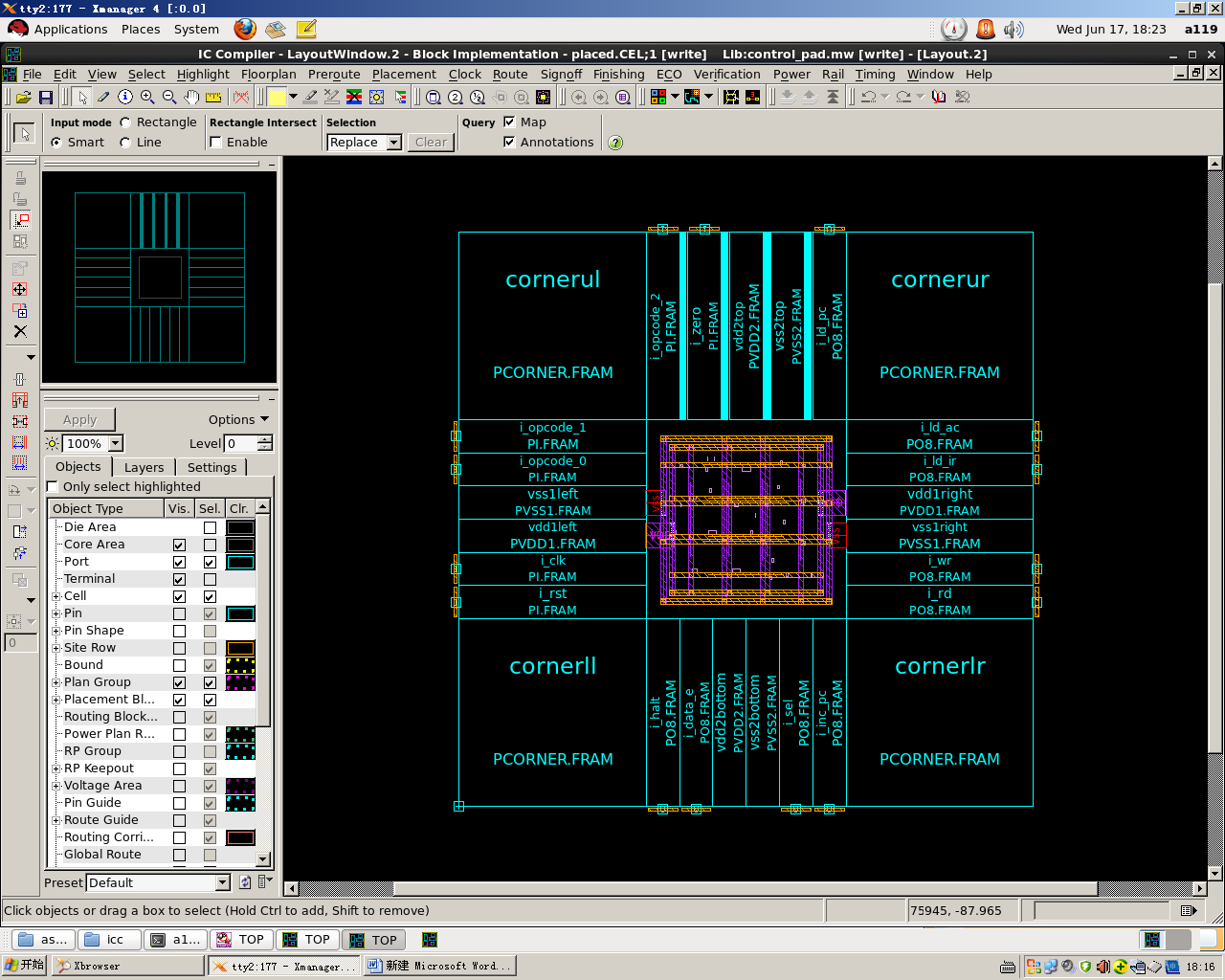
电源线和电影PAD连接后



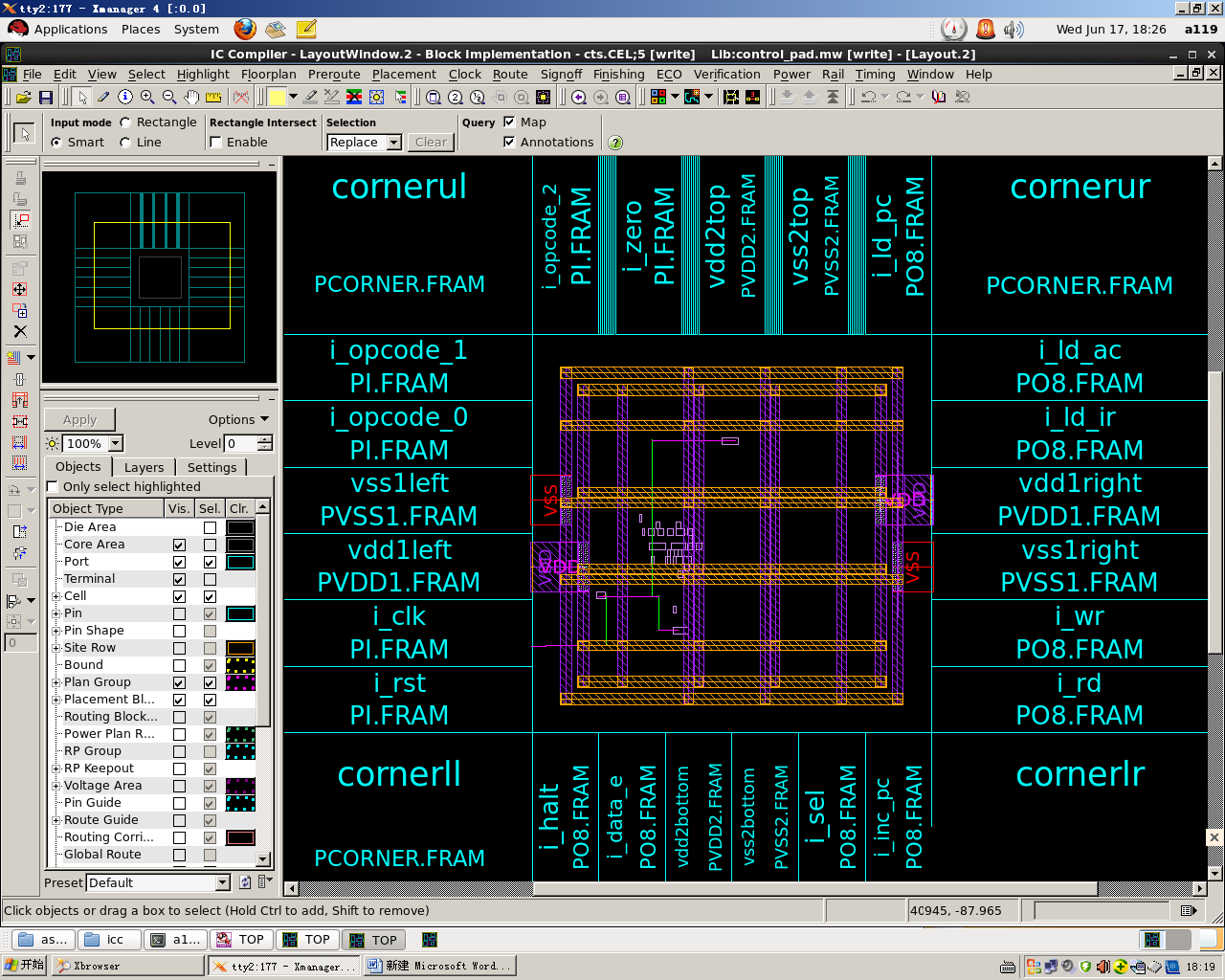
filler PAD填充后



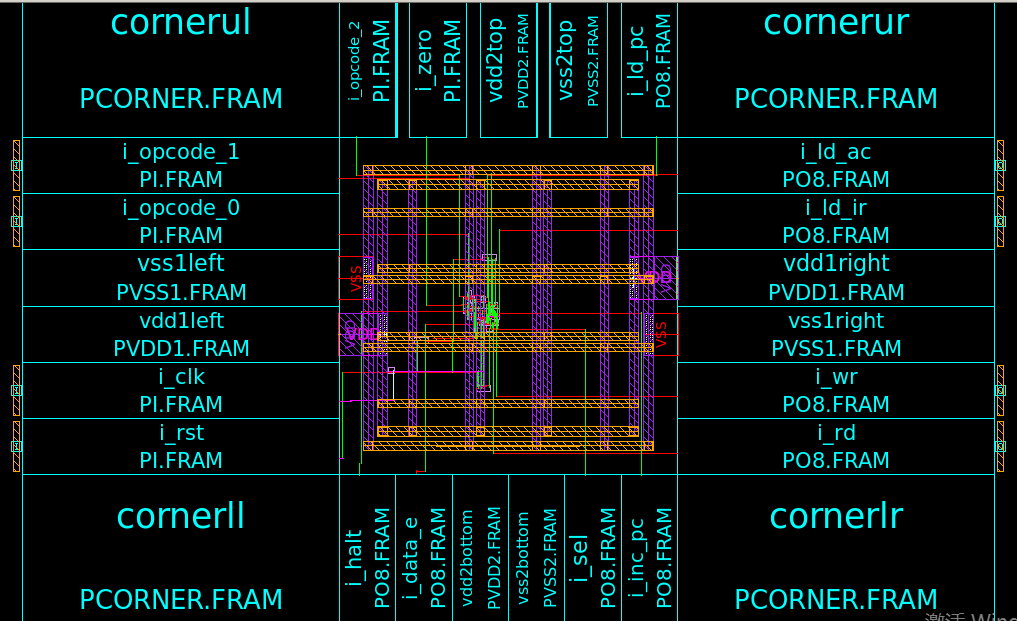
1. **布局**



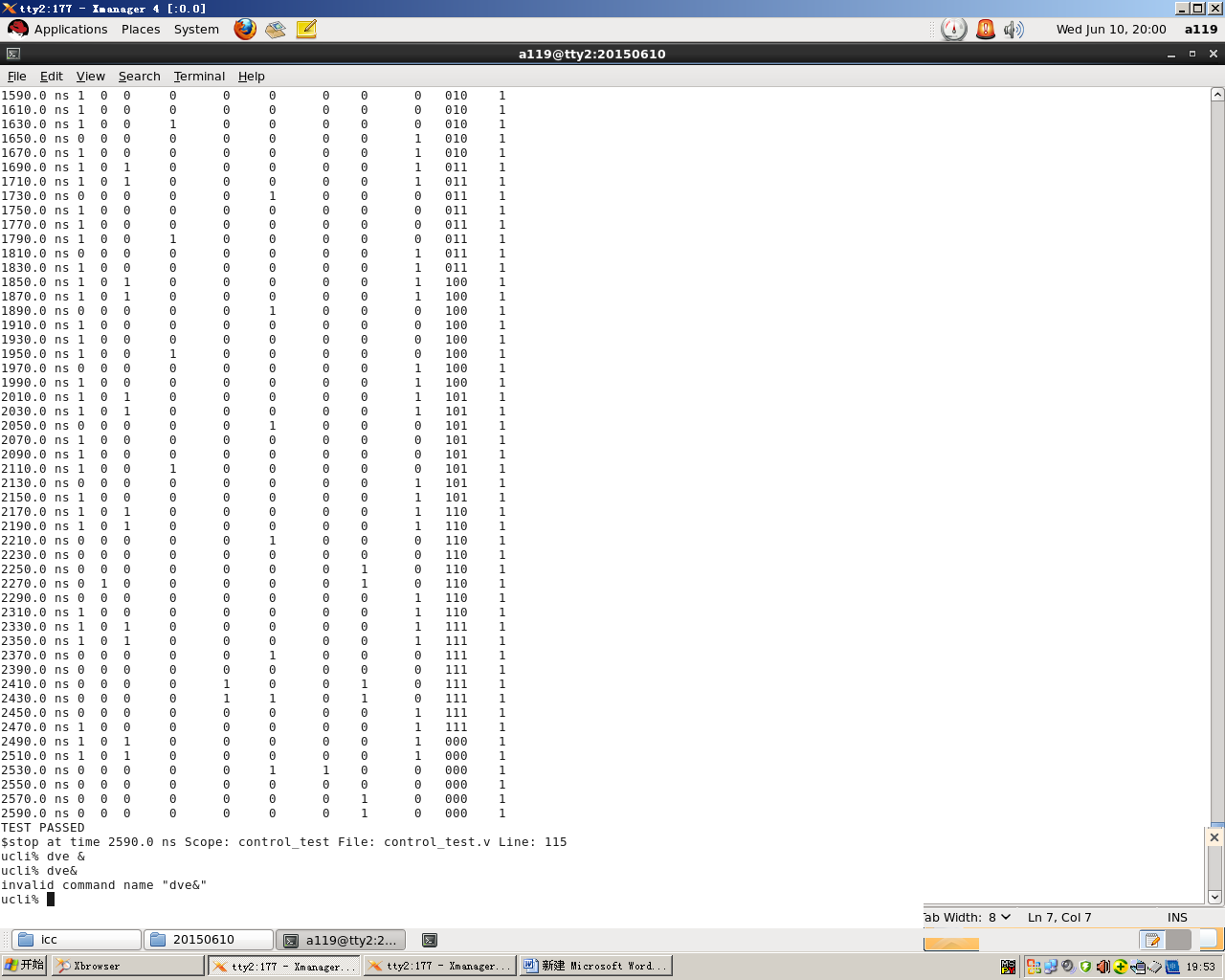
1. **时钟树综合**



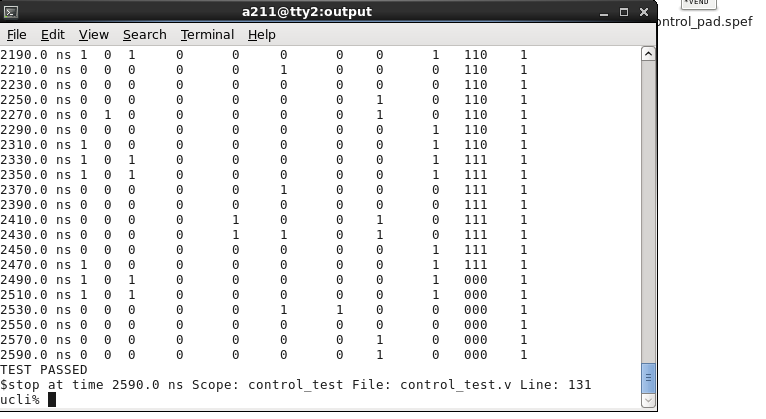
1. **布线**

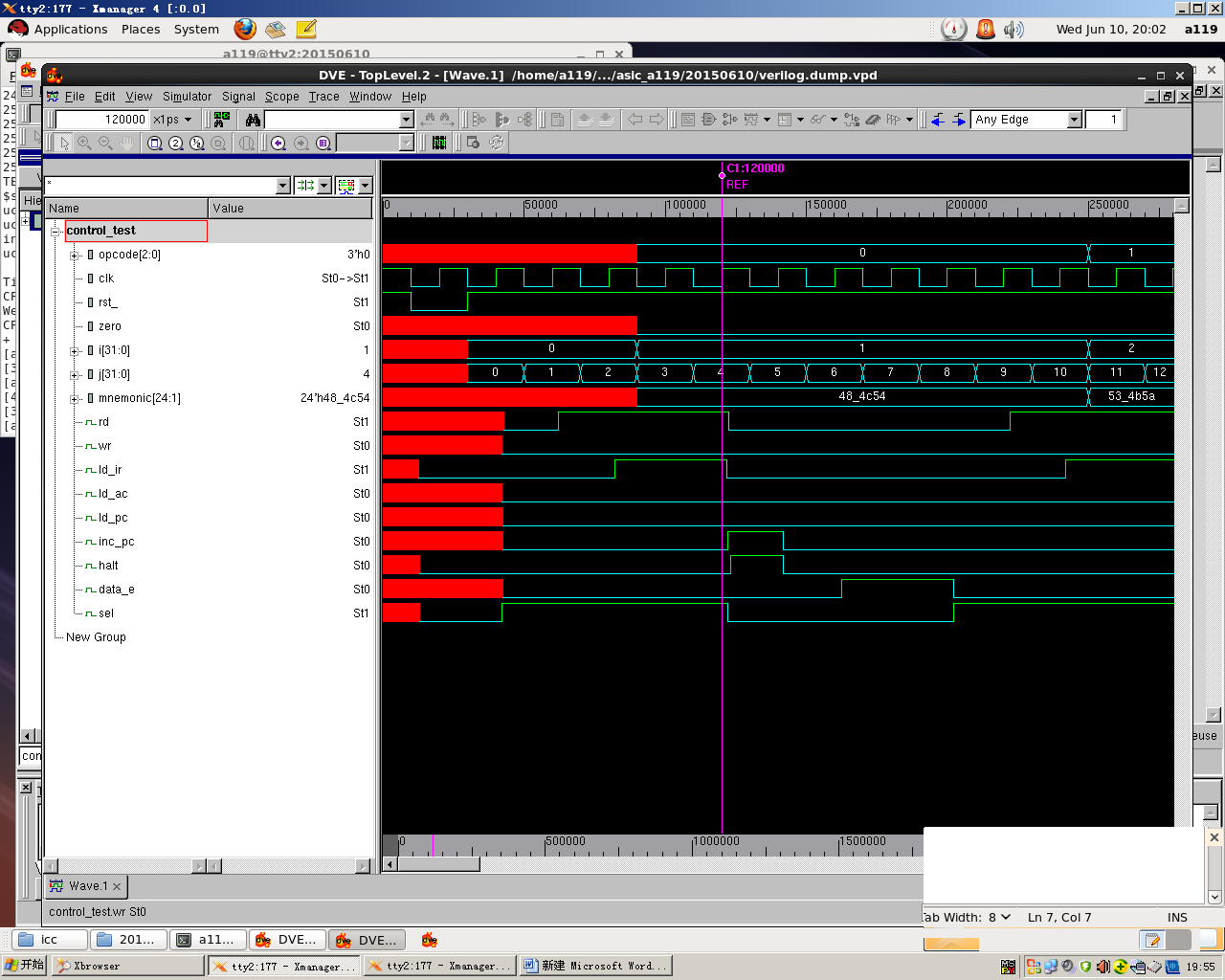
****

1. **寄生参数的导出**



1. **后仿真**

****



1. **思考题**
2. 简述ICC在design setup阶段的主要工作。

创建设计库，读取网表文件并创建设计单元，提供并检查时间约束，检查时钟。在对之前的数据与信息进行读取与检查后保存设计单元。

1. 为什么要填充标准填充单元？

filler pad把分散的pad单元连接起来，把pad I/O区域供电连成一个整体。使它们得到持续供电并提高ESD保护能力。

1. derive\_pg\_connection的作用是什么？

描述有关电源连接的信息。

1. 简述布图规划的主要任务。

对芯片大小、输入输出单元、宏模块进行规划，对电源网络进行设计。

1. 简述布局阶段的主要任务。

对电路中的延时进行估计与分析，模拟时钟树的影响，按照时序要求，对标准化单元进行布局。

1. 简述CTS的主要步骤。

设置时钟树公共选项；综合时钟树；重新连接扫描链；使能传播时钟；Post-CTS布局优化；优化时钟偏移；优化时序。

**实验遇到的主要问题与总结**

1.在Lab8的CPUtest1中命令停止在了第一行，经过长时间的debug最终发现问题出在CPU的模块例化中，例化调用选择器的模块名与设计文件中的模块名不匹配，导致无法调用该选择器模块。经过统一了各部分模块名后通过了测试。

2.在Lab8的CPUtest3中，斐波那契数列的输出测试出现了问题，数列的每一项都重复输出了多次，经过分析，数列的输出源于测试文件，输出逻辑是没遇到一次敏感操作码，就输出一次当前寄存器的内容，由于多次输出数列项，猜测可能是每一次数列项相加的循环中，由于敏感操作码的设置不当进行了多次输出，根据斐波那契数列的逻辑，将敏感操作码改为了opcode=2，即ADD操作，问题得以解决。

3.自己动手编写CPUtest4时，由于是第一次使用操作码进行编写，想要通过仅仅8条操作码实现复杂功能具有一定的挑战性。经过长时间的思考才决定好代码功能，最终完成了自己的算法设计。

4.实验遇到的主要问题是在CPU的RTL级设计中，部分源代码是逻辑不可综合的，导致在后期的综合过程中遇到了重重困难。其中包括各种不规范的赋值以及不可综合的选择支嵌套等。由于是综合时才出现的问题导致发现修改都变得十分困难。所以也使我加深了对书写规范重要性的认识。

本次实验使我完整的经历了集成电路的设计流程，从设计CPU架构，到依次设计编写CPU的各部分组件并测试验证功能，再到集成功能的CPU功能验证，而后进行电路的逻辑综合，最后将综合所得的门级网表进行布局布线，生成版图。一开始我的注意力仅仅放在了代码的正确性上，并没有深入地理解每一步的意义，而后在进行实际动手设计操作的时候，不停地犯错与改正使我慢慢的对于综合、布局布线有了进一步的理解。尤其是对时序报告进行分析的时候，只有通过一步一步的分析理解，才能够彻底地了解RTL设计是如何一步步变成带有各种延时信息的门级网表的。

一学期的实验不仅仅是高屋建瓴的一览完整的IC设计，更是深入到每一部分都由自己动手设计操作，机会十分难得。并且通过积极地参与到每次实验中，对于Verilog设计与各种仿真命令的操作越发的熟练，迅速完成自己实验项目的同时还能够给予班级其他同学指导，使我对自己在集成电路设计上的能力更加自信也更感兴趣。

在本次实验中不断的发现问题，不断的改正，不断地学习，不断地思考。回顾此次集成电路设计实验，将专业多门课程所学的知识通过实际应用的方式加深了理解，也引发了我对集成电路设计行业的兴趣与向往。近段时间不断地涌现出国内公司遭遇美国断供的新闻，中国集成电路行业的发展刻不容缓。我们作为电子专业的学生，能够通过本次实验加深对集成电路设计的理解与认知，受益匪浅！