

**电子科学与技术专业实验2**

**实验内容: ASIC设计理论与实践**

**姓 名： 黎加骏**

**学 院： 电子工程学院**

**专 业： 电子科学与技术**

**班 级： 2020211209**

**学 号： 2020210905**

**班内序号： 13**

目录

[概述 1](#_Toc11075235)

[一、ASIC设计流程图 1](#_Toc11075236)

[二、设计流程的说明 2](#_Toc11075237)

[实验一：RISC\_CPU RTL级设计及仿真 4](#_Toc11075238)

[Lab1：选择器设计 4](#_Toc11075239)

[一、实验内容 4](#_Toc11075240)

[二、核心代码及注释 5](#_Toc11075241)

[三、实验结果截图及分析 5](#_Toc11075242)

[四、实验小结 4](#_Toc11075243)

[Lab2：程序计数器设计 6](#_Toc11075244)

[一、实验内容 6](#_Toc11075245)

[二、核心代码及注释 6](#_Toc11075246)

[三、实验结果截图及分析 8](#_Toc11075247)

[四、实验小结 6](#_Toc11075248)

[Lab3：指令寄存器设计 9](#_Toc11075249)

[一、实验内容 9](#_Toc11075250)

[二、自行添加的代码及注释 9](#_Toc11075251)

[三、实验结果截图及分析 12](#_Toc11075252)

[Lab4：算术逻辑单元设计 14](#_Toc11075254)

[一、实验内容 14](#_Toc11075255)

[二、核心代码及注释 14](#_Toc11075256)

[三、实验结果截图及分析 16](#_Toc11075257)

[Lab5：存储器设计 17](#_Toc11075259)

[一、实验内容 17](#_Toc11075260)

[二、自行添加代码及注释 17](#_Toc11075261)

[三、实验结果截图及分析 18](#_Toc11075262)

[Lab6：设计时序逻辑时采用阻塞赋值与非阻塞赋值的区别 19](#_Toc11075264)

[一、实验内容 19](#_Toc11075265)

[二、核心代码及注释 19](#_Toc11075266)

[三、实验结果截图及分析 21](#_Toc11075267)

[Lab7：状态控制器设计 22](#_Toc11075269)

[一、实验内容 22](#_Toc11075270)

[二、自行添加代码及注释 23](#_Toc11075271)

[三、实验结果截图及分析 24](#_Toc11075272)

[四、实验小结 25](#_Toc11075273)

[Lab8：CPU集成设计及验证 27](#_Toc11075274)

[一、实验内容 27](#_Toc11075275)

[二、核心代码及注释 27](#_Toc11075276)

[三、实验结果截图及分析 32](#_Toc11075277)

[实验二：逻辑综合 36](#_Toc11075279)

[一、加载库文件 3](#_Toc11075280)6

[1、简介 37](#_Toc11075281)

[2、实验内容 3](#_Toc11075282)7

[二、读入设计 37](#_Toc11075283)

[1、简介 37](#_Toc11075284)

[2、实验内容 37](#_Toc11075285)

[三、施加设计约束 38](#_Toc11075286)

[简介及实验内容 38](#_Toc11075287)

[四、定义环境属性 39](#_Toc11075288)

[简介及实验内容 39](#_Toc11075289)

[五、综合及结果输出 40](#_Toc11075290)

[1、简介 40](#_Toc11075291)

[2、实验内容 41](#_Toc11075292)

[六、结果分析 43](#_Toc11075293)

[1、运行结果 46](#_Toc11075294)

[实验三：版图设计 48](#_Toc11075296)

[一、数据准备 4](#_Toc11075297)9

[1、简介 52](#_Toc11075298)

[2、实验内容 5](#_Toc11075299)1

[二、布图规划 55](#_Toc11075300)

[1、简介 55](#_Toc11075301)

[2、实验结果 56](#_Toc11075302)

[三、布局 58](#_Toc11075304)

[1、简介 58](#_Toc11075305)

[2、实验结果 59](#_Toc11075306)

[四、时钟树综合 60](#_Toc11075308)

[1、简介 60](#_Toc11075309)

[2、实验结果 60](#_Toc11075310)

[五、布线 62](#_Toc11075311)

[1、简介 62](#_Toc11075312)

[2、实验结果 62](#_Toc11075313)

[六、参数提取和后仿真 63](#_Toc11075314)

[1、简介 63](#_Toc11075315)

[2、实验结果 64](#_Toc11075316)

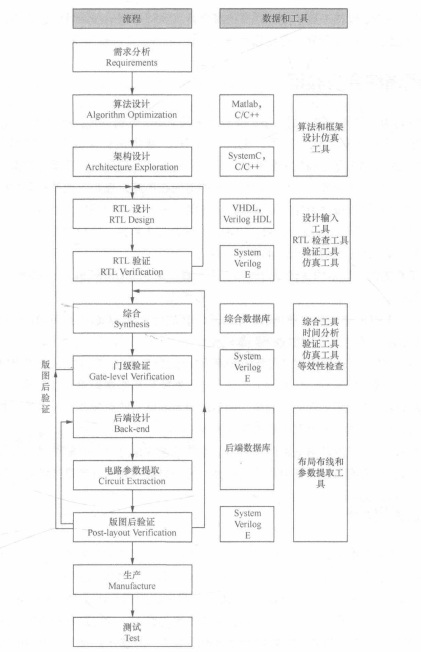
[七、物理验证 65](#_Toc11075317)

[实验体会 66](#_Toc11075318)

# 概述

## **一、**ASIC设计流程图

如图所示是芯片设计的典型流程，图中所有步骤均可采用Synopsys公司的EDA工具实现。



## **二、**设计流程的说明

一个复杂的ASIC芯片的设计流程包括需求分析、算法设计、架构设计、寄存器转换级( Register Transfer Lever, RTL)电路设计与验证、逻辑综合、布局布线、物理验证等部分，可以粗分为前端设计(也称逻辑设计)和后端设计(也称物理设计)。首先根据系统需求进行架构设计，针对关键模块提出或选择合适的实现算法，然后交由RTL设计者进行代码编写,并进行功能验证,对代码做进一步的修改和优化。接着利用EDA工具进行综合，得到门级网表，进行时序分析,验证设计是否符合时序要求。当验证完毕之后，相应的网表就会送到物理设计人员手中，进行布局和布线设计，最终进行芯片的流片和测试。

**1、设计定义和可综合的RTL代码**

设计定义描述芯片的总体结构、规格参数、模块划分、使用的接口等。通常使用硬件描述语言在寄存器传输级描述电路的行为，采用Verilog/VHDL描述各个逻辑单元的连接关系，以及输入/输出端口和逻辑单元之间的连接关系。

2、**前仿真**

也称为RTL级仿真或功能仿真。通过HDL仿真器验证电路逻辑功能是否有效，在前仿真时，通常与具体的电路实现无关，没有时序信息。

3、**逻辑综合**

电路的逻辑综合一般由三步组成：转化、逻辑优化和映射。首先将RTL源代码转化为通用的布尔等式（GTECH格式）；逻辑优化的过程尝试完成库单元的组合，使组合成的电路能最好的满足设计的功能、时序和面积的要求；最后使用目标工艺库的逻辑单元映射成门级网表，映射线路图的时候需要半导体厂商的工艺技术库来得到每个逻辑单元的延迟。综合后的结果包括了电路的时序和面积。

1. **版图规划**。

版图规划包含宏单元的位置摆放、电源网络的综合和分析、可布通性分析、布局优化和时序分析等。

1. **单元布局和优化**

单元布局和优化主要定义每个标准单元（Cell）的摆放位置，并根据摆放的位置进行优化。可以对设计在时序、功耗、面积和可布线性进行优化，达到最佳的结果质量。

1. **后布局优化**

时钟树综合和布线设计。在优化布局的基础上，进行时钟树的综合和布线。

1. **寄生参数的提取**

提取版图上内部互连所产生的寄生电阻和电容值。这些信息通常会转换成标准延迟的格式被反标回设计，用于静态时序分析和后仿真。

1. **后仿真**

以及时序和功耗分析。后仿真也叫门级仿真、时序仿真、带反标的仿真，需要利用局部布线后获得的精确延迟参数和网表进行仿真、验证网表的功能和时序是否正确。

1. **物理验证**

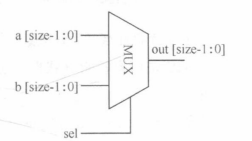
物理验证是对版图的设计规则检查（DRC）及逻辑图网表和版图网表比较（LVS）。其中DRC用以保证制造良率，LVS用以确认电路版图网表结构是否与其原始电路原理图（网表）一致。LVS可以在器件级及功能级进行网表比较，也可以对器件参数，如MOS电路沟道宽/长、电容/电阻值等进行比较。

# 实验一：RISC\_CPU RTL级设计及仿真

## Lab1：选择器设计

### 一、实验内容

本实验设计一个可综合的二选一开关，**结构示意图**如下：



**功能示意图**如下： 当sel=0 时，则out=a; 当sel=1 时，则out=b

### 二、功能分析

选择器的应用主要是作为地址多路选择器。有两种不同的指令执行状况：顺序执行和跳转执行。所以必须要增加一个部件来在多个数据源中选择一个提供给存储器利用，这一点就是通过选择器来实现的。此时，上面结构图中的a、b可分别指寄存器中地址变量ir\_addr、程序指针地址pc\_addr，选择后输出addr，最终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送到指令寄存器中。在CPU中这两位都是8位的，这也是我们在测试的时候选择width=8的原因。

### **三、**核心代码及注释

### （1）//mux.v core code

**assign** out **=** **(!**sel**)** **?** a **:**

**(**sel**)** **?** b **:**

**{**size**{**1'bx**}};** //使用assign结构描述组合逻辑，此语句实现了分支判断

//mux\_test.v core code

### （2）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**;**//按照功能要求设置了四种测试输入对应的输出

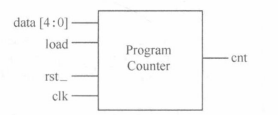
### **1_验证**四、实验结果截图及分析

**结果分析**：如图所示，当测试结果为sel=0时，out=a，sel=1时，out=b，与设计的目标电路相符合。

## Lab2：程序计数器设计

### 一、实验内容

本实验设计一个程序计数器，**结构图**如下：



功能示意图如图所示：

|  |  |  |  |
| --- | --- | --- | --- |
| **rst\_** | **load** | **clk** | **cnt** |
| **0** | **x** | **↑** | **0** |
| **1** | **1** | **↑** | **data** |
| **1** | **0** | **↑** | **1** |

### 

### 二、功能分析

在这个实验中，rst\_是复位信号，低电平有效，优先级最高；load是加载信号，高电平有效，指不做处理地输出原数据data；不复位不加载的时候就实现计数功能，即将输入数据data加1后输出。

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。

在分析该部分实验结果的时候需要考虑到模块的时延特性，即在某一时钟沿输入信号变化之后，输出不会立即变化，而是会出现一个延时。该实验的测试代码中使用了task，使得测试结果简明扼要，只需要结果显示“TEST PASSED”就可以判定rtl级仿真正确，非常方便。

### **三、**核心代码及注释

### （1）//counter.v core code

**always** **@** **(** **posedge** clk **or** **negedge** rst\_**)**//可以看出，在该代码中，rst\_信号一旦变为0，则cnt输出随即置零，而不用等到时钟沿的到来，所以rst\_是异步清零端。

**if** **(** **!**rst\_**)**

cnt**<=**0**;**

**else**

**if(**load**)**

cnt**<=**data**;**

**else**

cnt**<=**cnt**+**1**;**//这是一个时序逻辑电路，运用always模块描述

//counter\_test.v core code

**initial**

**begin**

**@(negedge** clk**)**

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

**@(negedge** clk**)expect(**5'h00**);**//rst\_的清零功能

**{**rst\_**,**load**,**data**}=**7'b1\_1\_11101**;**

**@(negedge** clk**)expect(**5'h1D**);**//load的加载功能

**{**rst\_**,**load**,**data**}=**7'b1\_0\_11101**;**//计数加1

**repeat(**5**)@(negedge** clk**);**//加5次

**expect(**5'h02**);**//h1D加5次1之后可得h02

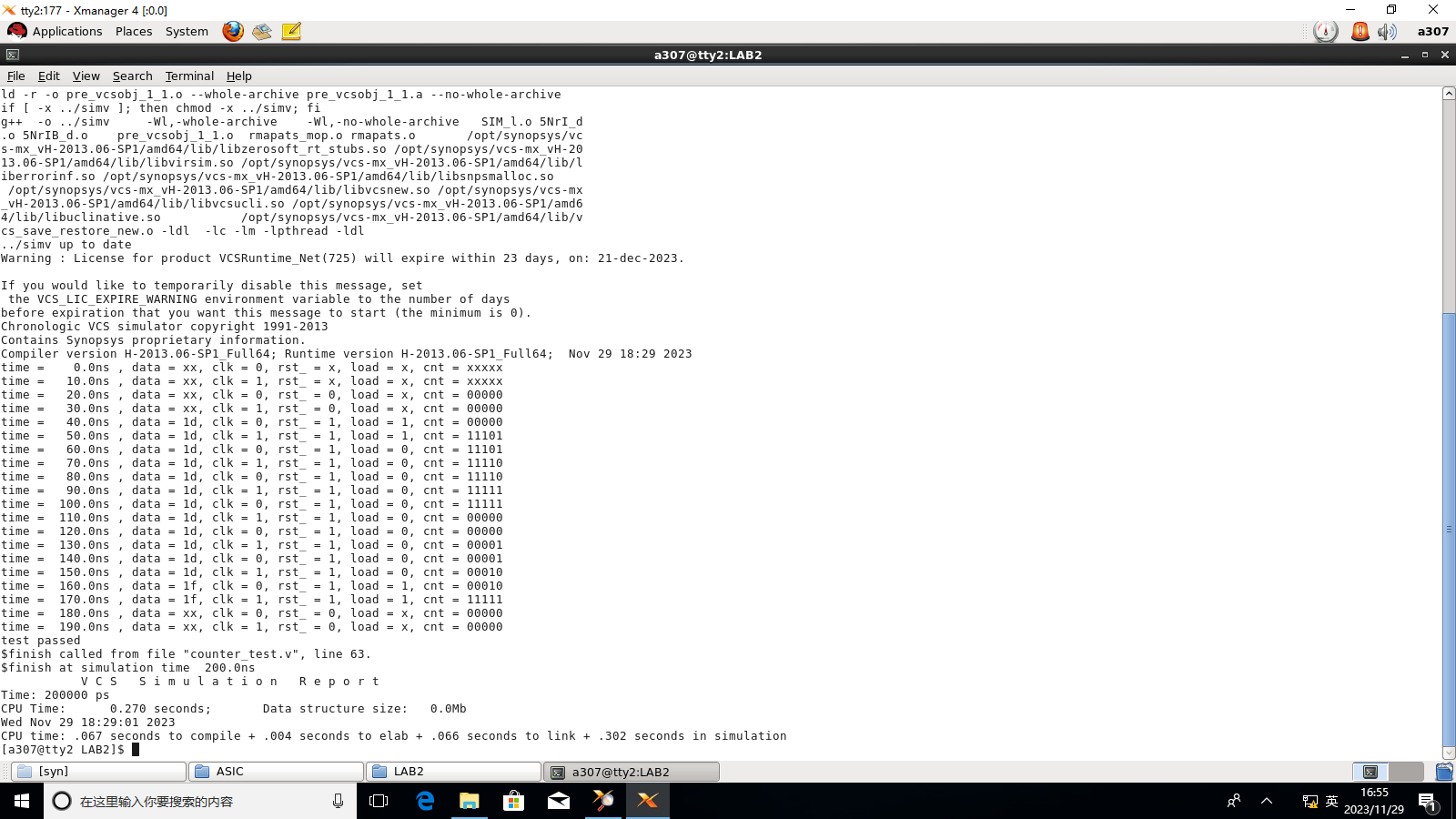
**{**rst\_**,**load**,**data**}=**7'b1\_1\_11111**;@(negedge** clk**)expect(**5'h1F**);**//加载

**{**rst\_**,**load**,**data**}=**7'b0\_X\_XXXXX**;@(negedge** clk**)expect(**5'h00**);**//清零

$display**(**"TAST PASSED"**);**

### 

### 四、实验结果截图及分析



**结果分析**：

最终输出显示“TEST PASSED”表明模块设计没有错误。

20ns时是第一个下降沿，rst\_置0，于是cnt置0；

40ns时是第一个下降沿，rst\_置1，load置1，由于有延迟，cnt没来得及转换，所以cnt仍为0；

……

加载5次后，160ns时load置1，由于有延迟，cnt没来得及转换，170ns时置位h1F，190ns时rst\_置0，cnt输出清零。

符合如下规律：

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。

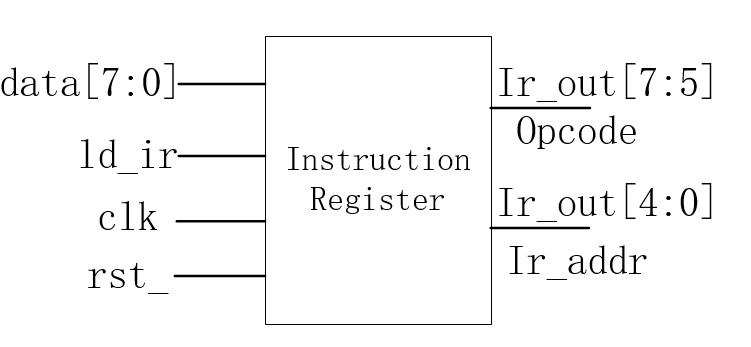
## 

## Lab3：指令寄存器设计

### **一、**实验内容

### 结构描述：寄存器在CPU中的结构框图如下

指令寄存器结构图如下：



**功能**描述：

### 结合寄存器的框图，可知每一位寄存器由一个二选一MUX和一个触发器dffr 组成，其功能实现如下:当load=1时，装载数据;当load=0时，寄存器保持。对于处理重复逻辑的电路，可用数组来描述。

**功能**示意图如图所示：

|  |  |
| --- | --- |
| **load** | **功能** |
| **1** | **转载数据** |
| **0** | **锁存数据** |

### **二、**自行添加的代码及注释

### 在该实验子目录中需要设计电路文件register.v、电路测试文件register\_test.v、一个周期为20ns的时钟电路clock.v、一个批处理文件run.f和一个 cells\_lib库文件夹。

### 其中的dffr和mux的调用代码需要自行添加，代码如下

### (1)//周期为20ns的时钟电路clock.v

`timescale 1 ns **/**1 ns

**module** clock**(**clk**);**

**output** clk**;**

**reg** clk**;**

**initial**

**begin**

clk**=**0**;**

**forever**

**begin**

**#**10 clk**=**1'b1**;**

**#**10 clk**=**1'b0**;**//通过延时后改变clk的值来生成一个时钟文件

**end**

**end**

**Endmodule**

(2)//register.v中dffr及mux模块的调用部分

dffr d0 **(.**q**(**out**[**0**]),.**d**(**n1**[**0**]),.**clk**(**clk**),.**rst\_**(**rst\_**));**

dffr d1 **(.**q**(**out**[**1**]),.**d**(**n1**[**1**]),.**clk**(**clk**),.**rst\_**(**rst\_**));**

dffr d2 **(.**q**(**out**[**2**]),.**d**(**n1**[**2**]),.**clk**(**clk**),.**rst\_**(**rst\_**));**

dffr d3 **(.**q**(**out**[**3**]),.**d**(**n1**[**3**]),.**clk**(**clk**),.**rst\_**(**rst\_**));**

dffr d4 **(.**q**(**out**[**4**]),.**d**(**n1**[**4**]),.**clk**(**clk**),.**rst\_**(**rst\_**));**

dffr d5 **(.**q**(**out**[**5**]),.**d**(**n1**[**5**]),.**clk**(**clk**),.**rst\_**(**rst\_**));**

dffr d6 **(.**q**(**out**[**6**]),.**d**(**n1**[**6**]),.**clk**(**clk**),.**rst\_**(**rst\_**));**

dffr d7 **(.**q**(**out**[**7**]),.**d**(**n1**[**7**]),.**clk**(**clk**),.**rst\_**(**rst\_**));**

mux m0 **(.**out**(**n1**[**0**]),.**sel**(**load**),.**b**(**data**[**0**]),.**a**(**out**[**0**]));**

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

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

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

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

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

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

mux m7 **(.**out**(**n1**[**7**]),.**sel**(**load**),.**b**(**data**[**7**]),.**a**(**out**[**7**]));**//模块调用过程中必须端口对应

//register\_test.v中单元实例化部分及信号监控模块

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

clock c **(.**clk**(**clk**));**//实例化过程中必须端口对应

**initial**

**begin**

$timeformat**(-**9**,**1**,**"ns"**,**9**);**//时间显示格式规定

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

$stime**,**clk**,**data**,**load**,**out**)**；//显示内容，其中$stime为系统函数，可显示当前运行时间

$dumpvars**(**2**,**register\_test**);**//波形记录文件

**End**

### 其中 run.f文件的内容如下:

//run.f

register\_test**.**v

register**.**v

clock**.**v

**-**y cells\_lib

**+**libext**+.**v

### 实验结果截图及分析

可以看出，指令寄存器的实际输出是分为两部分的，分别为操作码与操作数。

但执行一条指令时，应从存储器中经过数据总线把即将执行的指令读取到指令寄存器中，当指令寄存器渠道指令后，根据指令的操作码来判断指令将要执行的操作。指令寄存器的触发时钟是clk，由于数据总线上有时传输指令，有时传输数据，所以并不是每一个clk的上升沿都能寄存数据总线的指令，这是由控制器发送的ld\_ir信号控制的。当rst\_信号为高电平即复位后，指令寄存器被清零，继续准备接受下一指令。

### 3_验证

**结果分析**：这个实验的结果与书上提供的结果图有出入，我们小组成员讨论之后认为我们的实验结果符合代码输出，而书上的结果不符合对应代码的输出。原因在于系统将0ns时刻当做一个下降沿。

分析结果如下

Time=0ns，rst\_=0，data=XX，load=X，初始化输出为XX；

Time=10/20ns时，data=00，输出为00；

Time=30ns时第一个下降沿，rst\_=1，输出为0；

Time=50ns时第二个下降沿，data=h55，load=1，由于延迟，输出仍为0；

Time=60ns时，输出为h55；

Time=70ns时第三个下降沿，data=haa，load=1，由于延迟，输出仍为h55；

Time=80ns时，输出为haa；

Time=90ns时第四个下降沿，data=hcc，但load=0，所以输出不变，仍为haa；

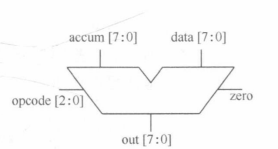
Time=100ns时输出为haa。

Time=110ns 时，仿真完成。

## 

## Lab4：算术逻辑单元设计

### 一、实验内容

本实验是设计一个算数逻辑单元，并结合ALU 的框图，设计一个模块名为alu.v的运算器电路,其**结构示意图**如下：

**其操作码与功能实现对照表**如下：

|  |  |  |
| --- | --- | --- |
| opcode | mnemonic | operation |
| 0 | PASSA | pass accumulator |
| 1 | PASSA | pass accumulator |
| 2 | ADD | data+accumulator |
| 3 | AND | data&accumulator |
| 4 | XOR | data^accumulator |
| 5 | PASSD | pass data |
| 6 | PASSA | pass accumulator |
| 7 | PASSA | pass accumulator |
| others | NONE | out=8'bx |

### 

### **二、**核心代码及注释

### 该电路通过对指令的判断，对输入数据执行相应的操作，包括加、与、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，结果都会做出及时的反应。

### 显然，如果采用assign 语句实现，表达起来将非常复杂。运用case结构来进行分支判断，不但设计思想得到直观的体现，而且代码看起来非常整齐、便于理解。

//alu.v

**always** **@(**opcode **or** data **or** accum**)**

**begin**

**if(**accum **==** 8'b0000\_0000**)**

**#**1.2 zero**=**1**;**//对zero信号的处理

**else**

**#**1.2 zero**=**0**;**

**case(**opcode**)**

PASS0**:#**3.5 out **=** accum**;**

PASS1**:#**3.5 out **=** accum**;**

ADD **:#**3.5 out **=** data **+** accum**;**

AND **:#**3.5 out **=** data**&**accum**;**

XOR **:#**3.5 out **=** data**^**accum**;**

PASSD **:#**3.5 out **=** data**;**

PASS6 **:#**3.5 out **=** accum**;**

PASS7 **:#**3.5 out **=** accum**;**//对各操作码对应的功能进行处理

**default** **:#**3.5 out **=** 8'bx**;**

**endcase**

**end**

### **三、**实验结果截图及分析

### 显示“TEST PASSED”说明仿真通过。

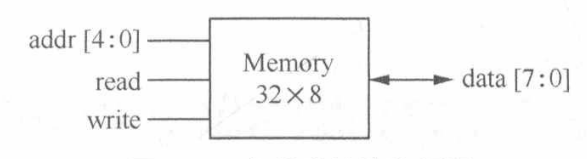
**结果分析**：通过查看操作码与功能实现的对照表，我们可以看懂输出结果，其中opcode=000,001,110,111输出都是accumulator(累加器)的值，101是输出data的值，010、011、100分别实现求和、与、异或的运算。

### 4_验证

## Lab5：存储器设计

### 一、实验内容

本实验设计了一个大小为32\*8的存储器，其**结构示意图**如下



**功能说明**：结合存储器的框图，可知存储器模型具有双向数据总线及异步处理功能。当read为高电平时，读出memory的数据到data总线上;在 write 的上升沿，将data总线上的数据写入memory。

**功能实现对照表**如下

|  |  |  |
| --- | --- | --- |
| read | write | 方向 |
| 1 |  | data->mem |
|  | ↑ | mem->data |

### 二、自行添加代码及注释

//alu.v设置read、write功能，此处代码需要自行添加

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

**assign** data **=** **(**read**)?** memory**[**addr**]** **:** 8'hz**;**//读出memory数据到data总线上

**always** **@(posedge** write**)**

**begin**

memory**[**addr**]** **<=** data**[**7**:**0**];**//将data总线上的数据写入memory

**end**

### 

### **三、**实验结果截图及分析

这次的实验遇见的主要问题是对reg [7:0] memory [0:31]这种形式的变量声明的理解。后来结合该存储器的大小是32\*8才理解到，这是声明了一个memory变量，该变量分为32个部分，每个部分表示一个字节，每个字节含有8位数据。之所以是32个字节是因为该CPU的地址位是5位。

**结果分析**：

在memory中先写入数据，再读出数据。

最终结果显示“TEST PASSED”说明仿真通过。

## Lab6：设计时序逻辑时采用阻塞赋值与非阻塞赋值的区别

### 一、实验内容

在always模块中，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赋值语句是并发执行的在实际的时序逻辑设计中，一般的情况下更多地使用非阻塞赋值语句，但有时为了在同一周期实现相互关联的操作，也会使用阻塞赋值语句。

实验中分别采用阻塞赋值语句和非阻塞赋值语句实现看上去非常相似的两个电路模块 blocking.v和non\_blocking.v，来比较两者之间的区别。另外，请读者观察由此所产生的电路的差异。

**其操作码与功能实现对照表**如下：

|  |  |  |
| --- | --- | --- |
| opcode | 助记符 | 操作 |
| 0 | HLT | 终止 |
| 1 | SKZ | 为0则跳过下一跳指令 |
| 2 | ADD | data+accum |
| 3 | AND | data&accum |
| 4 | XOR | data^accum |
| 5 | LDA | 把存储器中的数据加载到累加器中 |
| 6 | STO | 把累加器中的数据加载到存储器中 |
| 7 | JMP | 程序指针跳转 |

### 

### 二、核心代码及注释

//blocking.v

**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**;**//阻塞赋值

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

**end**

**endmodule**

//non\_blocking

**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**;**//非阻塞赋值

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

**end**

**endmodule**

//①修改blocking模块

**always@(posedge** clk**)**

**begin**

c **=**b**;**

b **=**a**;**

**end**

//②修改blocking模块

**always@(posedge** clk**)**

b **=**a**;**

**always@(posedge** clk**)**

c **=**b**;**

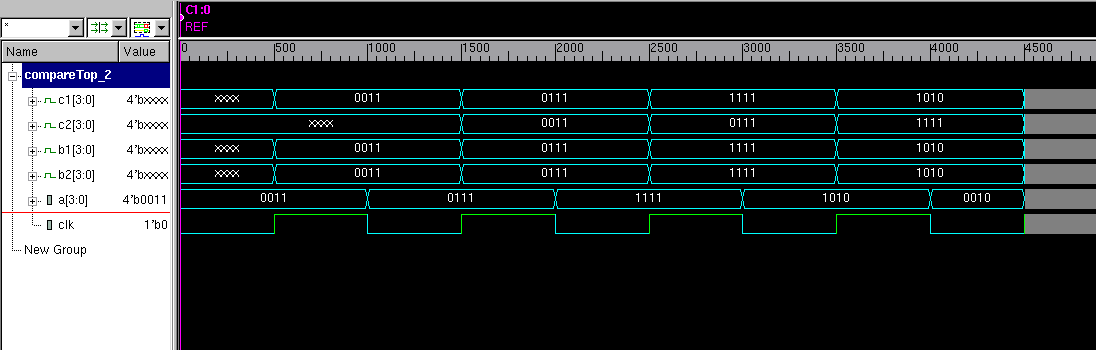
**end**

### 

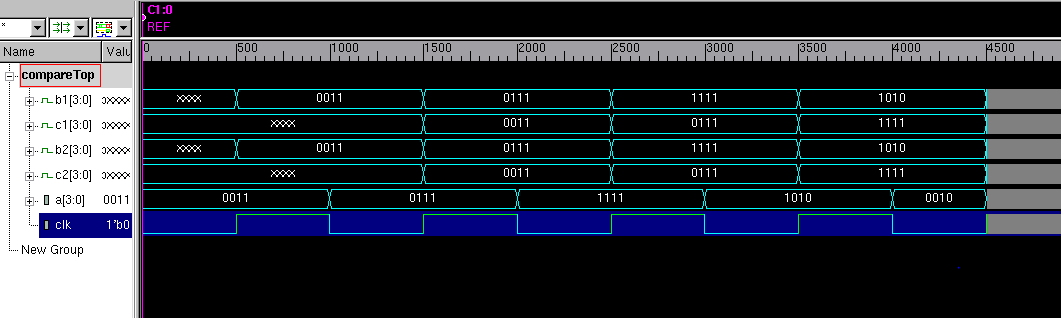
### 三、实验结果截图及分析

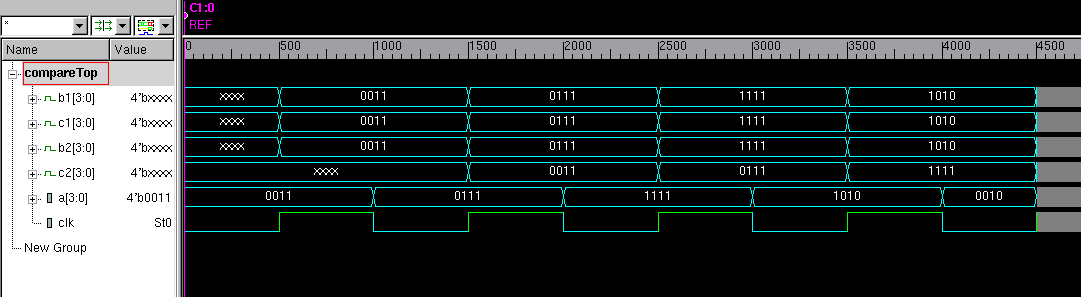
**（1）**

blocking.v的结果是阻塞赋值的结果，可以看出，两条赋值语句是同时进行的，即第一个时钟沿来的时候a的值赋给b的同时，将b在变化之后的值赋给c。

 nonblocking.v的结果a，b2，c2是非阻塞赋值的结果，可以看出，两条赋值语句是按顺序执行的，即第一个时钟沿来的时候先把a的值赋给b，等到第二个时钟沿来的时候再把b变化后的值赋给c。

**（2）：**这也是阻塞赋值，第一个时钟沿来到的时候，把b的值赋给c，然后把a的值赋给b。

**（3）**：两个always块执行的结果等同于阻塞赋值。



## Lab7：状态控制器设计

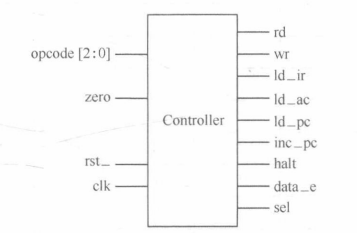
### 一、实验内容

本实验是要设计一个状态控制器，状态控制器是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何时读指令，何时进行RAM和IVO ( Input/Output)端口的读写等操作，都由控制器来控制。

这是一个有限状态机，其设计分为输入、输出、状态三部分，需要注意设定对应的代码段。

下图是其**结构示意图：**



**控制器指令变量操作对应表**如下：



### 

### 二、自行添加代码及注释

//control.v状态转移代码状态转移代码

**always@(**state**)**

nexstate **<=** state **+**1'h1**;**

//输入输出运算处理代码

**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**

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**;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**)**

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**)**

ld\_pc**=**1**;**

**if(**opcode**==**`STO**)**

wr**=**1**;**

**end**//此部分参考控制信号时序关系写出

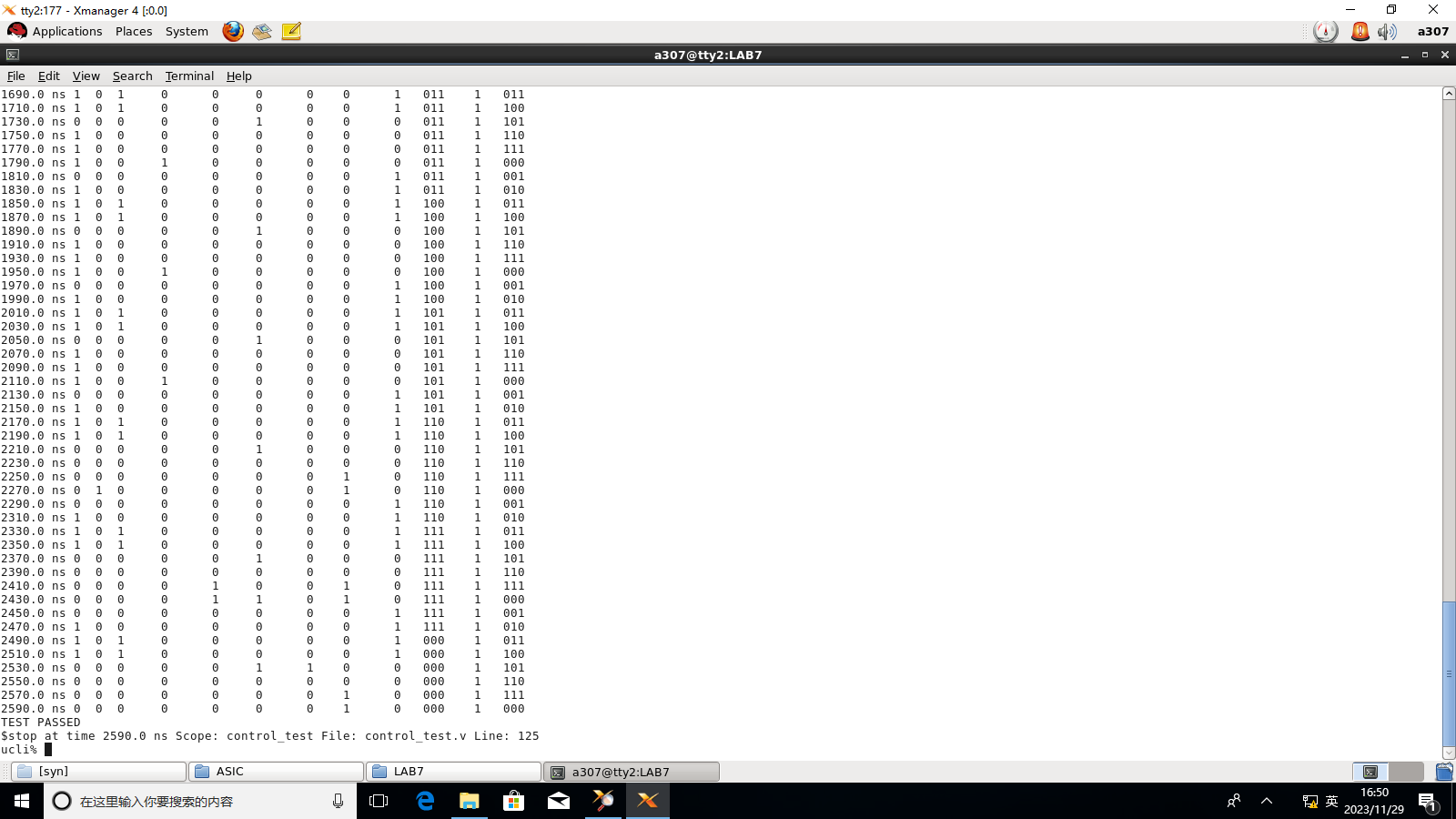
// default :begin sel=1'bZ;ld\_ir=1'bZ;inc\_pc=1'bZ;halt=1'bZ;ld\_pc=1'bZ;data\_e=1'bZ;ld\_ac=1'bZ;wr=1'bZ;end

**endcase**

### 

### 三、实验结果截图及分析

最终结果显示“TEST PASSED”，说明仿真正确。



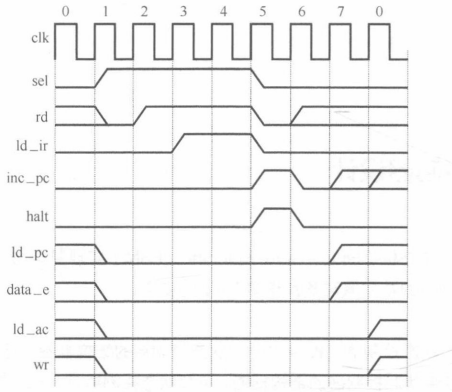
**结果分析**：程序读出激励文件stimulate.pat作为信号的输入，若输出结果与response.pat中的额数据一致则表示测试正确。

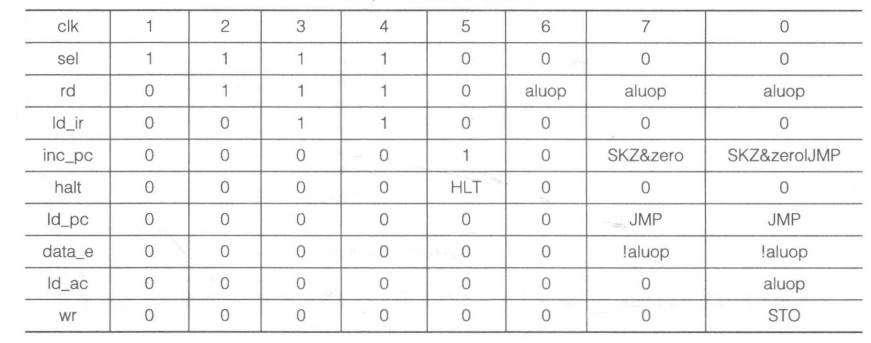
### 

### 四、结合LAB7控制器时序解释CPU的执行过程

本状态控制器由8个时钟周期组成，前4个时钟周期从存储器取数据，后4个时钟周期发出不同的控制信号来控制指令的执行。控制器在8个时钟周期内完成指令的获取和执行，并按照指令变量opcode和 alu来的zero标志产生以下控制信号。

控制信号的时序关系图如下：





这次实验的难点是看懂上面这个图，其中当执行ADD、AND、XOR和LDA

读懂每一个时间单位中CPU的执行过程有一些难度，如下所示：

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

## Lab8：CPU集成设计及验证

### 一、实验内容

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。

根据前7个模块的设计，可以根据下图的连接方式，完成一个8位RSIC\_CPU系统的设计。该CPU有32\*8的内存单元，能够实现8种指令。

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。

CPU内部的结构与信号连接关系如图所示：

### 

### 二、核心代码及注释

//cpu\_test.v

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

\* 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**(**"\* Enter \"scope cpu\_test; deposit test.N 4; task test; run\" \*"**);**

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

$display**(**"\* Enter \"scope cpu\_test; deposit test.N 5; task test; run\" \*"**);**//修改显示的提示内容

$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**(**"\* Enter \"call test(4);run\" to run the 4th diagnostic program. \*"**);**

$display**(**"\* Enter \"call test(5);run\" to run the 5th diagnostic program. \*"**);**//修改显示的提示内容

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

`endif

$stop **;**

**@** **(** **negedge** cpu1**.**clock **)**

rst\_ **=** 0**;**

**@** **(** **negedge** cpu1**.**clock **)**

rst\_ **=** 1**;**

**@** **(** **posedge** cpu1**.**halt **)**

$display **(** "HALTED AT PC = %h"**,** cpu1**.**pc\_addr **)** **;**

**disable** test **;**

**end**

// Define the test task

**task** test **;**

**input** **[**2**:**0**]** N **;**//修改N的二进制值的位数

**reg** **[**12**\***8**:**1**]** testfile **;**

**if** **(** 1**<=**N **&&** N**<=**5**)**//修改N的取值范围

**begin**

testfile **=** **{** "CPUtest"**,** 8'h30**+**N**,** ".dat" **}** **;**

$readmemb **(** testfile**,** cpu1**.**mem1**.**memory **)** **;**

**case** **(** N **)**

1**:**

**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**:**

**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**:**

**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**

### 三、 LAB8三个程序选择一个注解——选择CPUtest3 注解

**Test3主要功能是求一个Fibonacci数列，正确显示到144。**该测试文件是实现的功能是输出一个Fibonacci数列，期间遍历了8个操作码，根据显示的数据以及最后pc停止的位置来判断CPU设计是否合理，结果最后输出了0、1、1、2、3、5、8、13、21、34、55、89、144符合Fibonacci数列特征并且停止在PC=0C处，与预期一致，所以可以判断CPU设计合理。

//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

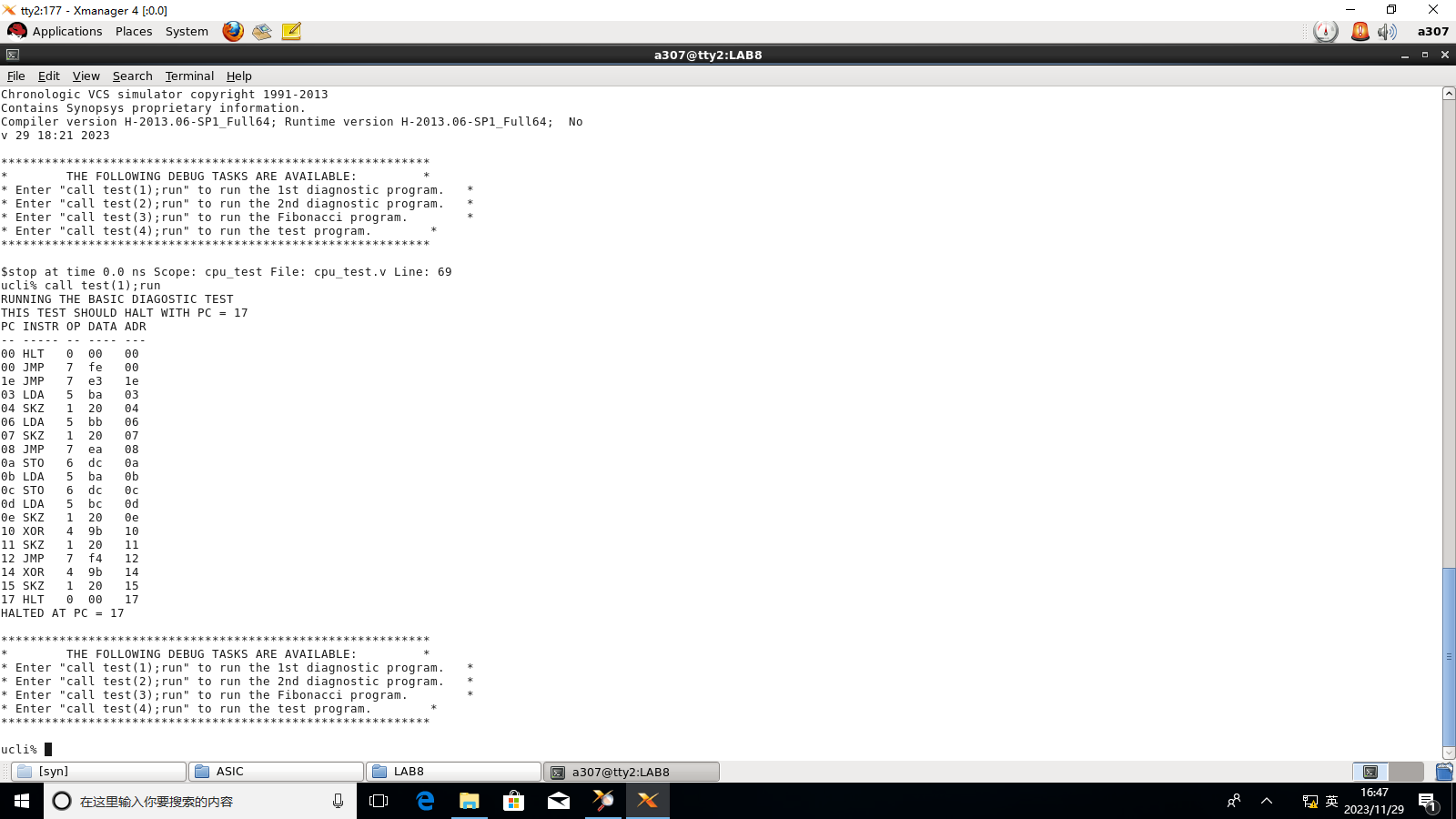
    00000001      //  1F   ONE:               //constant 1

### 

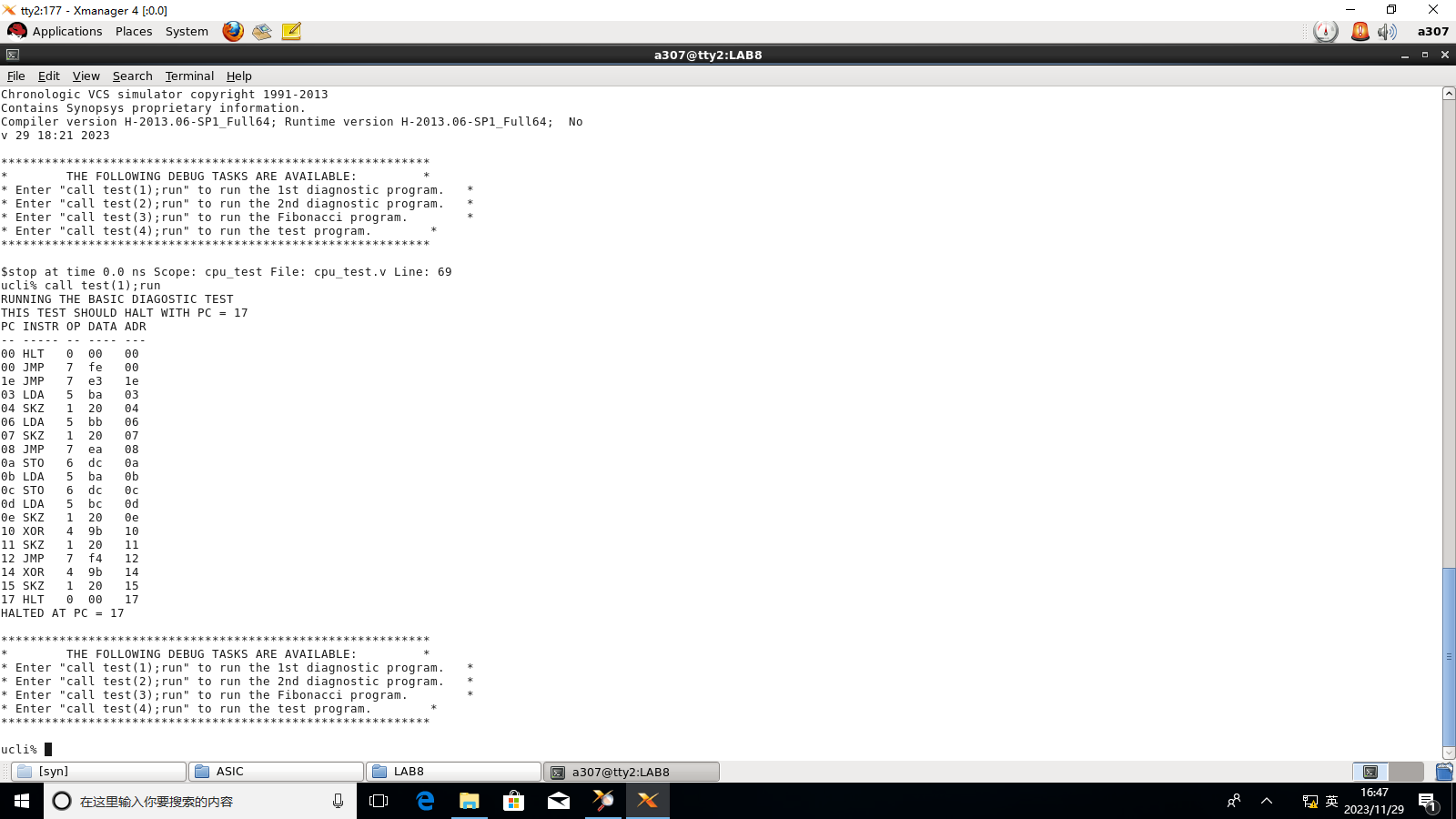
### 四、实验结果截图及分析

**结果分析**：此为调试成功后的显示界面。

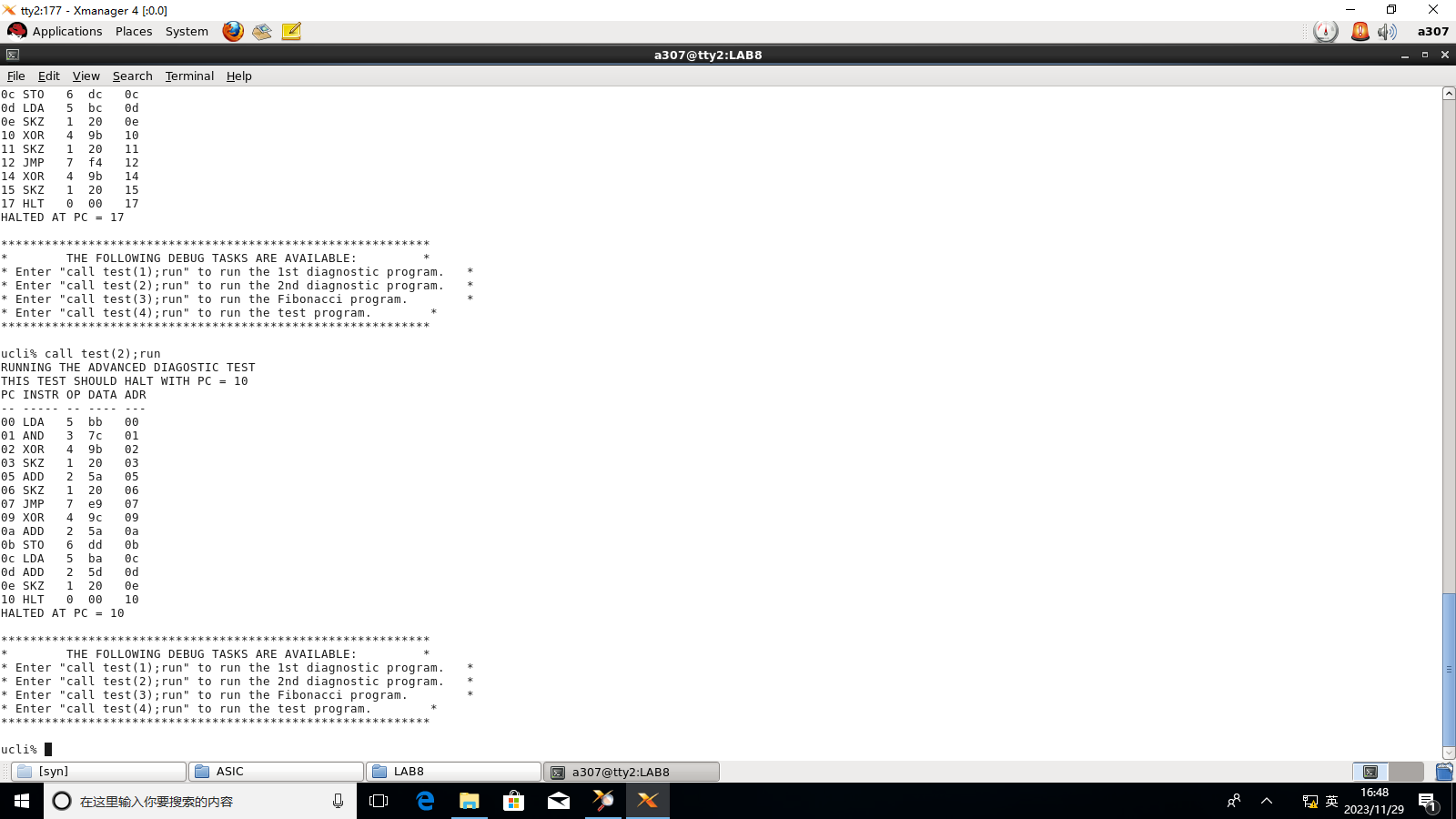
显示为： THE FOLLOWING DEBUG TASKS ARE AVALIABLE



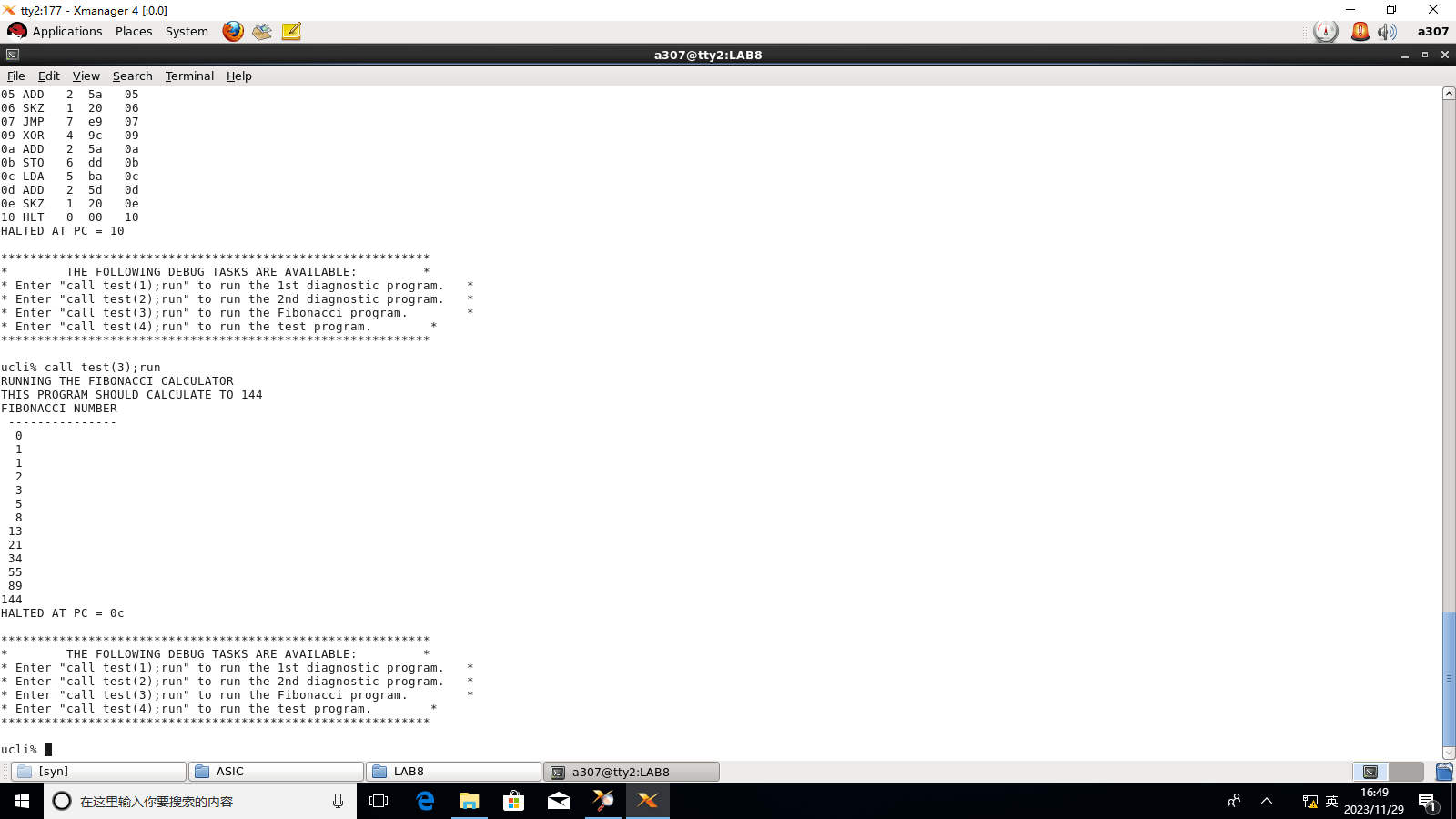
**结果分析**：这是CPU\_test1.dat运行结果，该测试文件是遍历了8个操作码，根据最后pc停止的位置来判断CPU设计是否合理，结果最后停止在PC=17处，与预期一致，所以可以判断CPU设计合理。



**结果分析**：这是CPU\_test2.dat运行结果，该测试文件是遍历了8个操作码，根据最后pc停止的位置来判断CPU设计是否合理，结果最后停止在PC=10处，与预期一致，所以可以判断CPU设计合理。



**结果分析**：这是CPU\_test.dat3运行结果，该测试文件是实现的功能是输出一个Fibonacci数列，期间遍历了8个操作码，根据显示的数据以及最后pc停止的位置来判断CPU设计是否合理，结果最后输出了0、1、1、2、3、5、8、13、21、34、55、89、144符合Fibonacci数列特征并且停止在PC=0C处，与预期一致，所以可以判断CPU设计合理。



# 

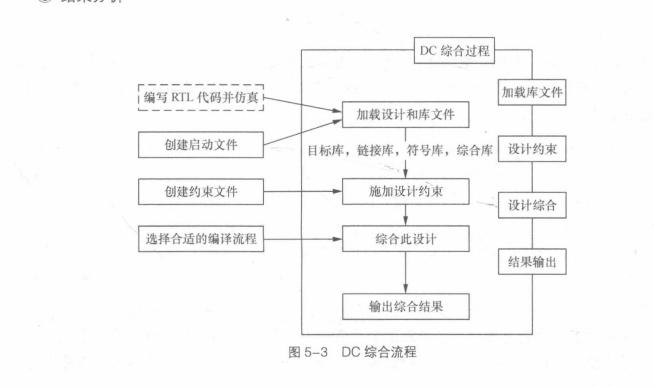
# 实验二：电路综合

# 描述逻辑综合流程

### 逻辑综合流程

综合是前端模块设计中的重要步骤之一，综合的过程是将行为级描述的电路和RTL级的电路转换到门级的过程。Design Compiler(简称DC )是Synopsys公司用于电路综合的核心工具，综合可以方便地将HDL语言描述的电路转换成基于工艺库的门级网表。本节将初步描述使用Design Compiler完成电路综合的全过程。

逻辑综合流程主要包括以下步骤：1.加载库文件；2.读入设计；3.设计约束；4.设计综合；5.结果输出；6.结果分析 流程图见下图：



### 

### 一、加载库文件

主要涉及4种库文件

①单元库：单元库是综合后电路网表要最终映射到的库，读入的HDL代码首先有Synopsys自带的GTECH库转换成DC内部交换的格式，然后经过映射到工艺库并优化生成门级网表。单元库包含了各个门级单元的行为、引脚、面积、以及时序信息，DC在综合是就是根据工艺库中给出的单元电路的延迟信息来计算路径的延迟的，并根据各个单元延时、面积和驱动能力的不同选择合适的单元来优化电路。

②链接库：用来设置模块或者单元电路的引用。

③符号库：是定义单元电路显示的Schematic的库。

④IP库：对于扩展的DesignWare库需要在IP库中设置。

## 二、读入设计

### 1、简介

DC支持多种硬件描述的格式，包括.db、.v、.vhd等等，对于TCL的工作模式来说，读取不同的文件格式需要使用不同的命令。

### 2、实验内容

在本实验中，读入设计的命令写在scripts目录下的脚本文件dc\_scripts.tcl中。

代码为

read\_file –format verilog ./rtl/control.v

read\_file –format verilog ./rtl/control\_pad.v。

## 

## 三、施加设计约束

### 1.简介及实验内容

DC需要根据这些约束将RTL模板综合成门级网表，然后分析综合出的网表是否满足约束条件，如果不满足就要修改约束条件，甚至重写RTL代码。

时序约束要达到的苗木表是约束电路中所有时序路径，使其延迟符合设计要求。可分为三类

①时钟约束

定义时钟时必须定义它的时钟源，时钟源可以是端口也可以是管脚，还必须定义时钟的周期

在本实验中，读入设计的命令写在scripts目录下的脚本文件dc\_scripts.tcl中。

代码为creat\_clock –period 20 [get ports clk]

set\_dont\_touch i\_rst true。

第一行代码定义一个周期为5ns的时钟，时钟源是一个称为clk的端口。第二行代码把所有定义的时钟网络设置为don’t touch，即综合的时候不对clk信号优化。代码中对所有端口都进行了这样的设置，，如果不进行这步操作，DC会根据clk的负载自动产生Buffer。

②约束输入路径

主要通过定义输入延迟实现。输入延迟是指被综合模块外的寄存器触发的信号在到达被综合模块之前经过的延迟。

在本实验中，该命令写在scripts目录下的脚本文件dc\_scripts.tcl中。

代码为set\_input\_delay 0.1 –max –clock clk [remove\_from\_collection [all\_inputs] [get\_portsclk]]。

上面语句指出了被综合模块的所有输出端口的最大延时为0.1ns。

③约束输出路径

主要通过定义输出延时来实现。输出延迟是指本电路端口到达本电路之外电路触发器所需要的延时。

在本实验中，该命令写在scripts目录下的脚本文件dc\_scripts.tcl中。

代码为Set\_output\_delay 1 –max –clock clk [all\_output]。

该语句指出了被综合模块的所有输出端口的最大输出延时为1ns。

## 

## 四、定义环境属性

### 简介及实验内容

主要包括以下四个方面：

1>设置工作条件

Set\_operating\_conditions用于设置工作条件，通常有max、type、min三种类型

2>设定输出负载

综合出来的电路必须要驱动下级电路，如果负载取得过小，下级电路将无法正常工作；如果负载过大，则会增大上一级电路的难度。

在本实验中，该命令写在scripts目录下的脚本文件dc\_scripts.tcl中。

代码为Set\_load [expr[load\_of $lib\_name/AND2X4/A\*15][all\_outputs]。

3>设置输入驱动

如果驱动太大，综合出来的电路的负载很大；如果驱动太小，信号的变化边沿会很差。

在本实验中，该命令写在scripts目录下的脚本文件dc\_scripts.tcl中

代码为Set \_sriving\_cell –library $lib\_name –lib\_cell AND2X4 [remove\_from\_collection[all\_inputs] [get\_ports clk]]

4>设置连线负载模型

连线负载模型基于连线的扇出、估计电阻电容等寄生参数，均由Foundry提供。根据连线负载模型，DC可以估算出连线的电容、电阻和面积，找出这条线驱动的负载，得出其扇出数，通过网表就可以查出相应的线的长度。

## 

## 五、综合及结果输出

### 1、简介

综合是指在满足设计电路的功能、速度及面积等限制条件下，实现将行为级描述转化为指定的技术库中的单元电路连接，完成RTL级向门级网表的转换。实际情况下，综合会实现门的延时、导线的延时、信号的转换时间及时钟信号到达各个触发器的时间不同等情况。

综合的过程具体包括如下三步

1>代码转化:把有硬件描述语言描述的电路转化为用GTECH库单元组成的逻辑电路的过程，即实现从源代码到逻辑电路的过程

2>逻辑优化：即设计者根据迪那路路设定的延时和面积等约束条件对电路进行优化设计的过程，它通过各种方法尽量满足设计者对电路的要求。

3>门级映射：把用GTECH库元件组成的电路映射到某一固定厂家的工艺库上。

### 

### 2、实验内容

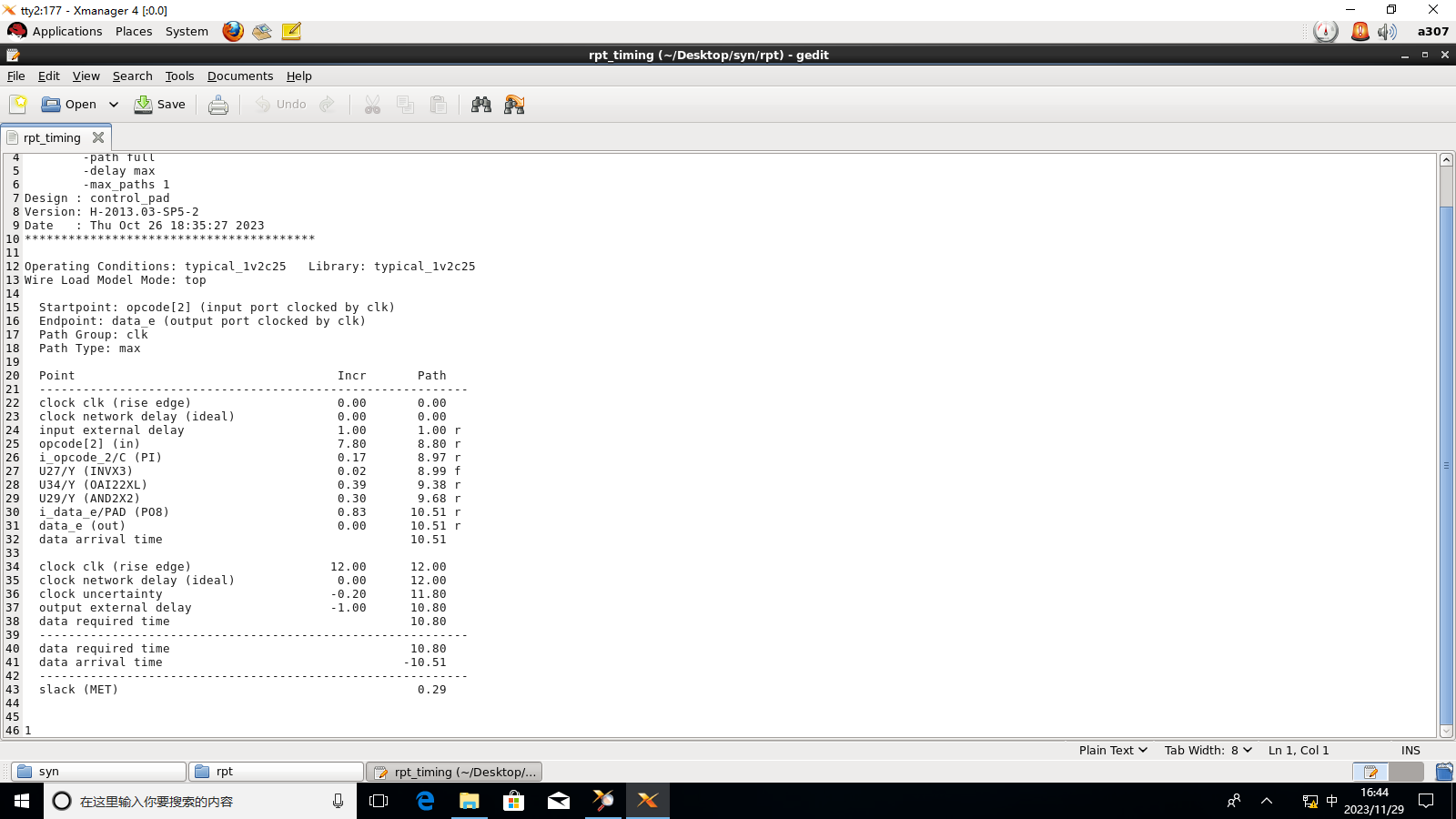
综合后会输出几份报告。包括延迟报告、面积报告、功耗报告。

在本实验中，该命令写在scripts目录下的脚本文件dc\_scripts.tcl中。

1>report\_timing > .rpt/rpt/rpt\_timing

输出延迟报告如下：

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10.51ns

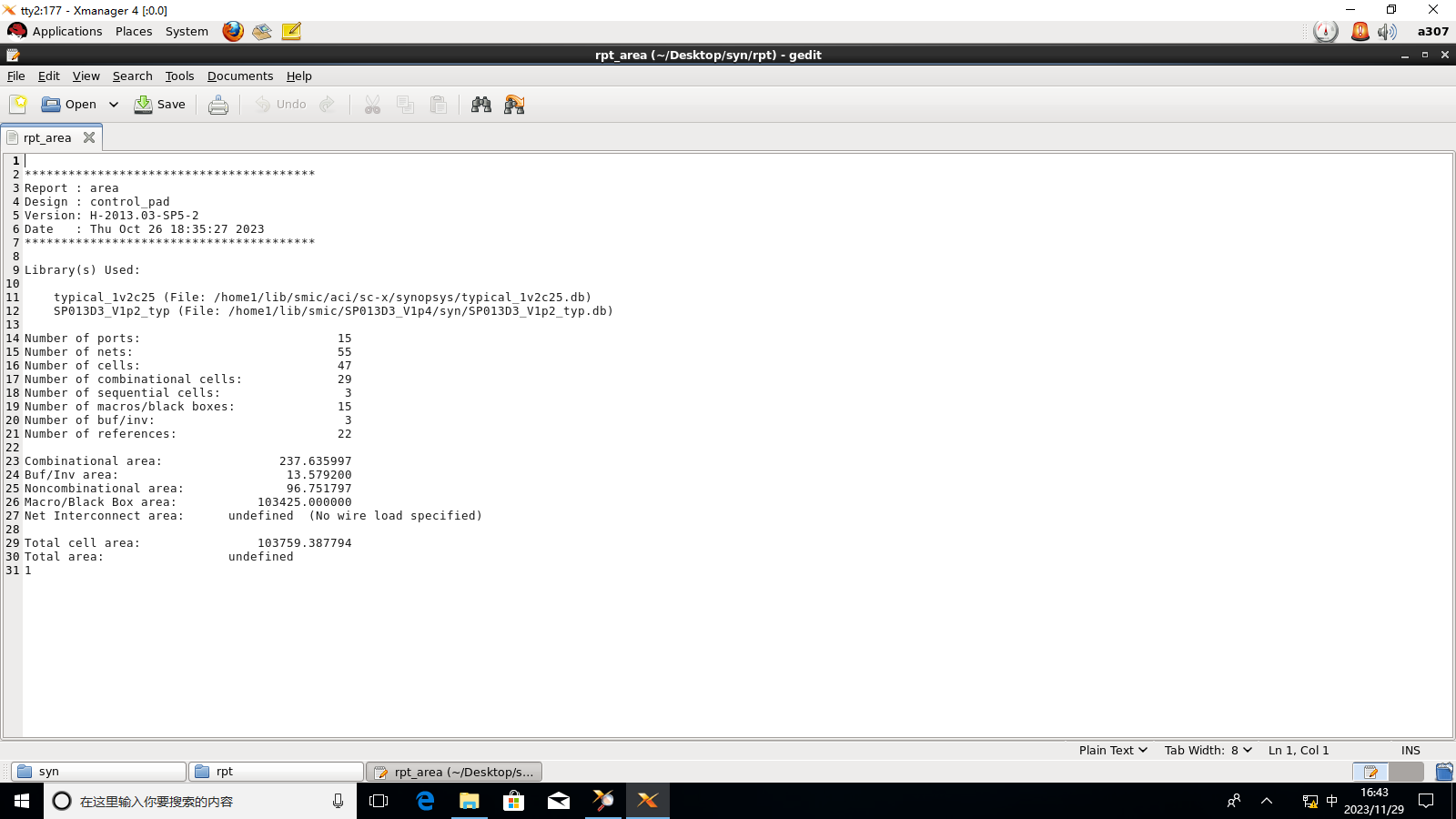


1. report\_area > .rpt/rpt/rpt\_area

DC可以导出时序文件：时序描述文件（描述设计的时序情况主要用于进行综合后的动态仿真）、时序约束文件（带有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），都是SDF的格式。

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约为103759.个面积单位。

输出面积报告如下：



1. report\_power > .rpt/rpt/rpt\_power

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），占了总功耗的约99%。

## 功耗分析报告

### **4、**时序描述文件

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。

### 时序描述文件

### **5、**时序约束文件

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。

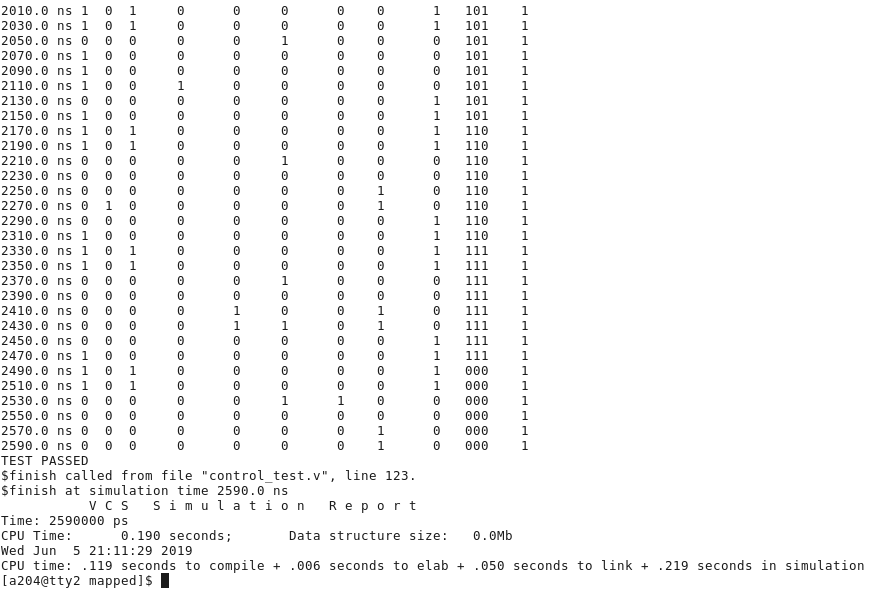
### 时序约束文件

## 六、结果分析

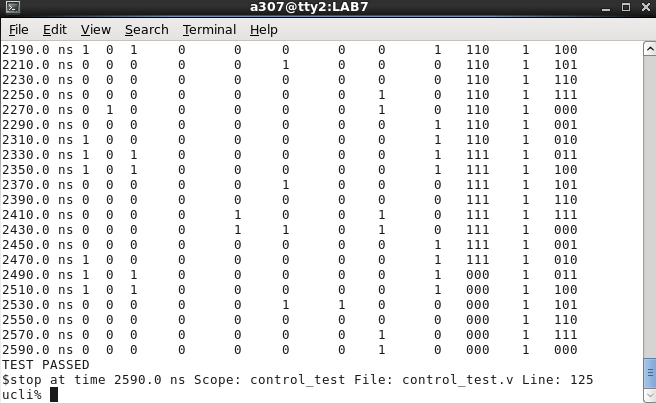
### 1、运行结果

门级电路仿真结果：

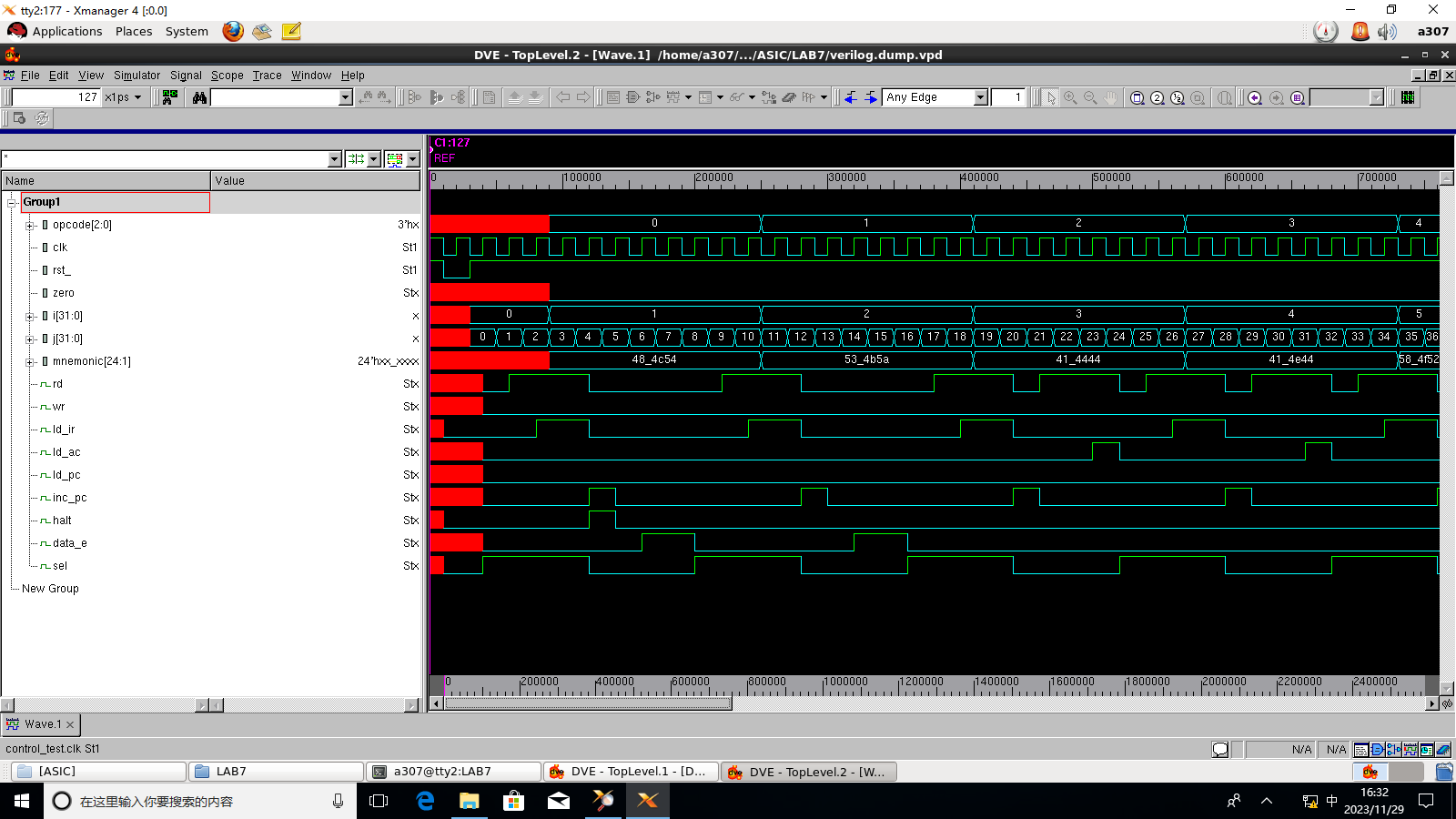
输出“TEST PASSED” 则仿真成功



RTL级仿真结果



RTL级仿真时序图如下



# 数字集成电路时序的概念

### 数字集成电路时序是指数字电路中各个信号的时序关系，即信号的产生、传输和响应的时间关系。在数字电路中，信号的传输是有一定延迟的，因此需要考虑信号的时序关系，以确保电路的正确性和稳定性。

### 当前集成电路设计工具大都采用时序驱动的电路设计方法，其思路就是对电路的各种时序路径施加时序约束。在对电路进行综合的过程中，集成电路设计工具根据用户对电路施加的时序约束和静态时序分析技术对电路进行编译和优化。

### 在时序驱动电路设计方法中，综合工具根据用户对电路路径的约束的要求、建立时间与保持时间对信号传输时间的要求，计算出电路路径的最大和最小传输延迟，作为电路逻辑综合的依据。

### 建立时间对电路的要求

### 为了实现正确的同步，寄存器的信号输入端必须满足建立时间的要求。在时钟信号到达前一级寄存器R1后输出逻辑信号，输出的逻辑信号经过组合逻辑电路后到达寄存器R2。逻辑信号到达 R2的时刻应该至少比下一个周期时钟信号到达R2的时刻早一个建立时间的长度。

### 用公式表示为:

### 整理后得到：

### 可以看到，集成电路的时钟周期受到电路延迟、建立时间和时钟偏斜的限制。为提高芯片的频率，时钟周期越小越好。时钟频率取决于两个寄存器之间最大的传输延迟。每两个时序逻辑单元之间的信号延迟都需要满足公式的要求，在所有路径中的最大延时就是限制时钟周期的最关键因素，这条路径称为“关键路径”。找出各个寄存器间组合逻辑的最长路径是逻辑综合工具和静态时序分析工具需要解决的主要问题之一。

### 保持时间对电路的要求

### 由保持时间的定义可知，在时钟信号到达寄存器并对寄存器输入端采样之后，数据输入端仍需维持一段时间才能保证寄存器输出端的正确输出。一般情况下，后级寄存器在本时钟周期边沿收到前级寄存器在上一个时钟周期边沿处发出的信号，为保证后级寄存器上的输入信号在本周期时钟到达后仍能稳定一段时间，前级寄存器在本时钟周期边沿发出的信号到达后级寄存器的时间不能太早，前级寄存器本时钟周期边沿的输出信号应当在后级寄存器本时钟周期边沿经过保持时间后才到达后级寄存器。

### 用公式表示为:

### 整理后得到：

### 从上式中可以看到,当δ为正时,δ一定要小于tR1 + t logic一 T hold ;另外需要强调的是，一般情况下建立时间是在下一个时钟边沿处比较，而保持时间是与两个寄存器的同一个时钟边沿比较。

### 静态时序分析

### 在电路综合过程中和综合完成后，需要对时序进行分析或验证。

### 传统电路的验证方法是在验证功能的同时验证时序，需要输入向量作为激励。随着规模增大，所需向量数量增长，验证所需时间大大提高，最大的问题则是难以保证足够的覆盖率。所以，这种方法越来越少地被用于时序验证，取而代之的是静态时序分析技术。

### 静态时序分析技术主要是检查建立时间和保持时间是否满足要求，通过计算信号在路径上的延迟传播找出违背时序约束的错误，是一种穷尽分析方法，运行速度很快，占用内存很少，它克服了动态时序验证的缺陷，节省了设计时间。静态时序分析较难的地方是存在超长路径的设计，有时甚至需要修改代码来满足时序要求。对于具有多时钟的设计，还需要对时钟进行分离，分别对每个时钟域进行静态时序分析。很多时候设计中会存在伪路径，在时序分析时要注意找出关键伪路径，在施加约束时把其设为false path。

### 静态时序分析是大规模集成电路设计中非常重要的一个技术。在电路设计过程中，为得到一个最佳的电路设计，在结构逻辑、电路布局布线等方面时序分析都起着关键性的作用。静态时序分析既要检验电路的最大延迟，以保证电路在指定的频率下能够满足建立时间的要求;同时又要检验电路的最小延迟，以满足保持时间的需求。芯片的设计只有通过了静态时序分析才能真正完成，甚至从逻辑综合开始后的每一个步骤的结果都需要满足或部分满足时序的要求。随着芯片尺寸的减小和集成度密集化的增强、电路设计复杂度的增加、电路性能要求的提高等，各种新的变化都对芯片内的时序分析提出了更高的要求。

# 逻辑综合脚本注解

### 逻辑综合脚本与注解如下

# dc\_ scripts. tcl

read\_ file -format verilog ./rt1/contro1.v      //读取Verilog源代码文件`control.v`

read\_ file -format verilog ./rt1/contro1\_ pad.v //读取Verilog文件`./rtl/control pad.v

write -hierarchy -f ddc -out unmapped/control\_ pad . ddc //以Design Compiler的格式将设计写入文件`unmapped/control pad.ddc`，包括层次结构信息。

list\_ designs                                    //列出当前设计中的所有设计单元和库

list\_ libs

set lib\_ name typical\_ 1v2c25

current\_ design control\_ pad

link

write -hierarchy -f ddc -out unmapped/control\_ pad. ddc

list\_ designs

list\_ libs

# Create clock object and set uncertainty

create\_ \_clock -period 20 [get\_ ports clk]       //创建一个时钟对象，并设置时钟周期为20，时钟端口为`clk`。

set\_ clock\_ \_uncertainty 0.2 [get\_ clocks c1k]   //设置时钟的不确定性为0.2。

# Set constraints on input ports

suppress\_ \_message UID-401

set\_ \_driving\_ ce1l -library $lib\_ name -1ib\_ cell AND2X4 [remove\_ from\_ collection

[all\_ inputs] [get\_ ports c1k]]

set\_ input\_ delay 0.1 -max -clock clk [remove\_ from\_ collection [all\_ inputs] [get\_ ports

c1k]]

#set\_ input\_ \_delay 1.2 -max -clock clock [get\_ ports Neg\_ Flag]

# Set constraints on output ports

set\_ output\_ delay 1 -max -clock c1k [all\_ outputs]

set\_ load [expr [load\_ of $lib\_ name/AND2X4/A] \* 15] [al1\_ outputs]

set\_ dont\_ touch i\_ rst true                     //设置一系列输入端口`rst`为不进行优化

set\_ \_dont\_ touch i\_ clk true

set\_ dont\_ touch i\_ zero true

set\_ \_dont\_ touch i\_ opcode\_ 0 true

set\_ \_dont\_ \_touch i\_ opcode\_ 1 true

set\_ dont\_ touch i\_ opcode\_ 2 true

set\_ dont\_ touch i\_ sel true

set\_ dont\_ touch i\_ data le, true

set\_ dont\_ touch i\_ inc\_ pC true

set\_ dont\_ \_touch i\_ 1d\_ \_pc true

set\_ dont\_ touch 1\_ 1d\_ ac true

set\_ dont\_ touch i\_ ld\_ ir true

set\_ dont\_ touch i\_ wr true

set\_ dont\_ touch i\_ rd true

set\_ dont\_ touch i\_ halt true

set\_ dont\_ touch\_ network opcode [0]

set\_ dont\_ touch\_ network opcode [1]

set\_ dont \_touch\_ network opcode [2]

compile\_ \_ultra                                    //使用Ultra编译模式进行综合和优化。

report\_ constraint -all > ./rpt/rpt\_ consitraints  //生成约束报告，包括所有约束信息，并将结果写入文件`./rpt/rpt constraints`。

report\_ timing > . /rpt/rpt\_ timing                //生成时序报告。

report\_ area                                       //生成面积报告

> ./rpt/rpt\_ area

report\_ power > . / rpt/rpt\_ power                 //生成功耗报告。

write -hierarchy -format ddc -output . /mapped/control\_ pad . ddc  //将映射后的设计层次结构写入文件`./mapped/control pad.ddc`。

write -hierarchy -format verilog -output . /mapped/control\_ pad.v  //将映射后的Verilog代码写入文件`./mapped/control pad.v`。

write\_ sdc . /mapped/control\_ pad. sdc .           //将时序约束文件写入文件`./mapped/control pad.sdc`。

write\_ sdf . /mapped/control\_ pad. sdf             //将时序延迟文件写入文件`./mapped/control pad.sdf`。

list\_ designs              //列出当前设计中的所有设计单元与库

list\_ libs

# 实验三：版图设计

# 描述版图设计的概念

### 1、版图设计的概念

集成电路版图设计是依据综合后的门级网表、工艺库文件、时序约束文件等定义掩膜版图的过程，是电路设计的物理描述。

它的目标是根据电路的功能和性能要求，以及工艺条件的限制，设计工艺制造过程中必需的光刻掩膜版图。通过集成电路版图设计，可以将立体的电路系统变为一个二维的平面图形，再经过制造工艺加工还原为基于硅材料的立体结构。因此，版图设计是一个上承电路系统、下接集成电路芯片制造的中间桥梁，是电路系统与集成电路工艺之间的重要环节。

输出包含三个文件，分别是GDSII文件、SPEF 寄生参数文件和版图之后的.v文件，以下分别进行介绍。

### **2、**版图设计的输入与输出

### 版图设计的输入和输出均包含多个文件，其中，输入包含五个文件，分别是综合后的网表文件、时序约束文件( sdc文件)参考库文件( reference文件）。工艺库文件( tech 文件)和寄生参数库文件( tlup文件);

# 版图设计所用库及其概念

### 1、版图设计所用库及其概念

版图设计中用到的库文件主要包括：

物理单元库（reference库)、逻辑库( .db ) 、RC提取库（ Tluplus )、工艺库（ .tf )。

**物理单元库（ reference库）**包含标准单元、宏单元和IO单元的物理信息，用于布局和布线。

**逻辑库（ logic library** )包含标准单元和宏单元的功能和时序信息，并包含单元的驱动和负载设计规则。

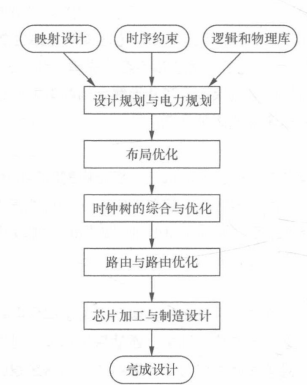
**RC提取库（ TLU+）**用于计算连线延迟的模型文件。

**工艺库( tf 文件)**提供了特定工艺信息，例如:每层金属的名称、物理和电学特性。另外，tf 文件还可以包含设计规则。tf文件可由write\_mw\_lib\_files指令写出。

# 实验内容

## 零、版图设计流程

整个版图设计的流程如下图所示，包括数据准备、布图规划、布局、时钟树综合、布线等关键步骤。



## 数据准备（Design Setup ):设计库和最初设计单元的创建、网表和sdc文件的读入、reference库和工艺库的设置,逻辑“1”和逻辑“O”的处理等。

## 布图规划（Floorplan ):对芯片（ die ) 大小和形状的规划、芯片输入/输出单元（I/O）的规划、宏单元的规划、电源网络的设计等。

## 布局（ Place ):主要是对标准单元的布局，即时序驱动的过程，其目标是在满足延迟的要求下，尽量减少布线的拥挤度。

## 时钟树综合（CTS ):创建时钟树结构，通过一级一级的 buffer来驱动负载（寄存器)，其主要目的是减少时钟偏斜（ clock skew )o

## 布线(Route ):完成电源线、信号线和电路单元pin 的互连，并优化互连结果。

## 一、数据准备

### 1、简介

创建一个Milkyway库，同时在库里创建一个设计，并把需要的库文件和Milkyway关联起来。通过read\_verilog命令读入一个网表，并保存为一个设计。通过set\_tlu\_plus\_files命令输入tluplus RC寄生参数，通过read\_sdc命令设置采用的时序约束文件。通过derive\_pg\_connection对门级网表中的0和1信号进行处理，并与电源进行逻辑关联。

本实验中，这部分的代码在scripts目录下的脚本文件design\_setup.tcl中。

# 二、floorplan的过程

### 1、简介

布图规划是用来确定整个芯片的形状、大小、pad摆放顺序，包括新生成的pad并且提供电源供电方案，包括core的供电和pad的供电。

作用如下：

1、产生原来不存在的单元，如电源pad、cornor pad等；

2、约束pad的摆放位置；

3、整个空间的形状大小。

本实验中，布图规划前三部分的相关代码在scripts目录下的脚本文件floorplan.tcl中。

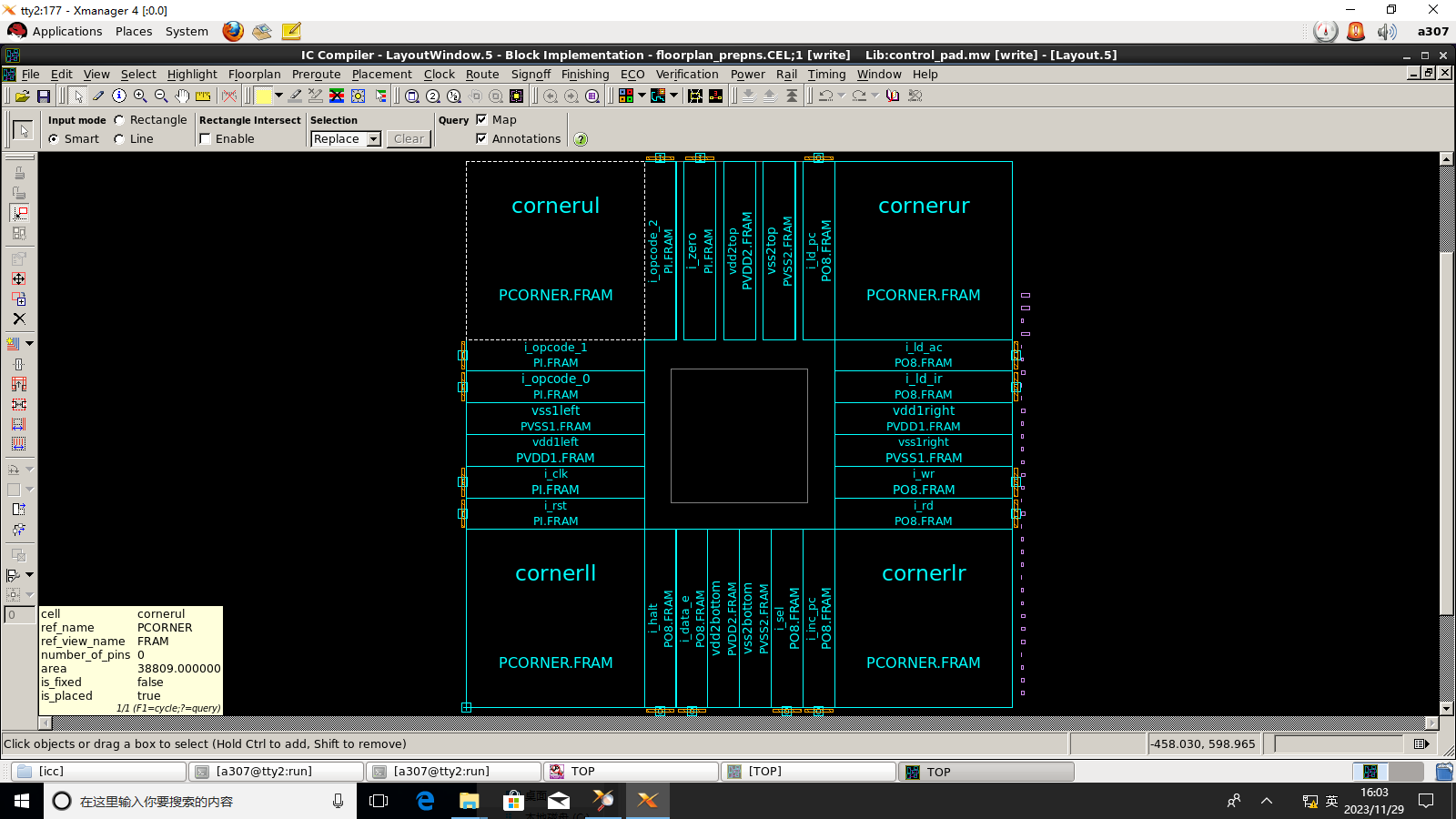
4、供电方案

首先要插入填充单元，用来填充I/O单元和I/O单元之间的缝隙，作用主要是把扩散层连接起来满足DRC规则和设计需要，并形成电源线和地线轨道。然后需要形成电源环，在核和I/O单元之间加入pad电源环。

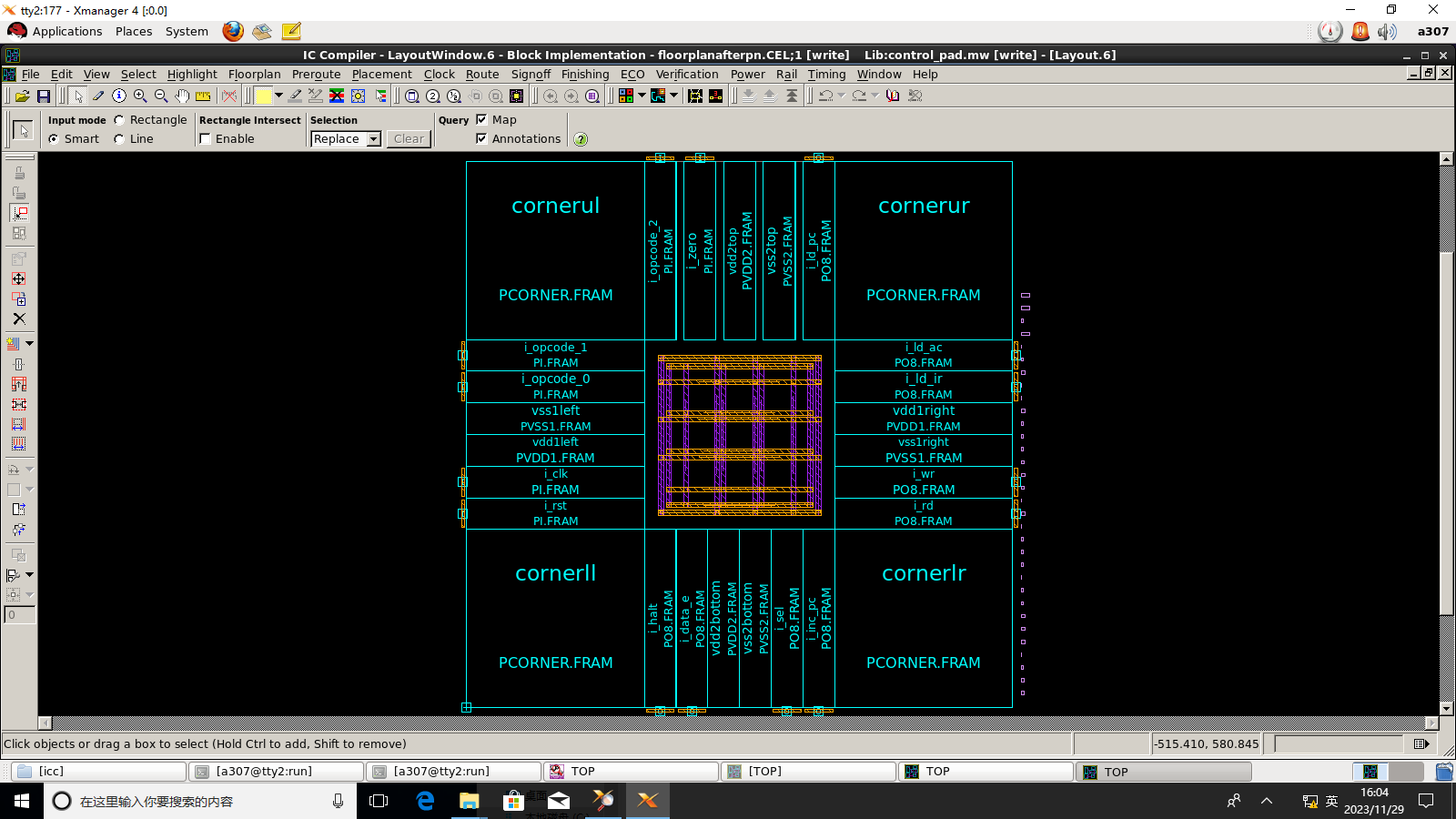
### 2、实验结果

各阶段图形界面如下：

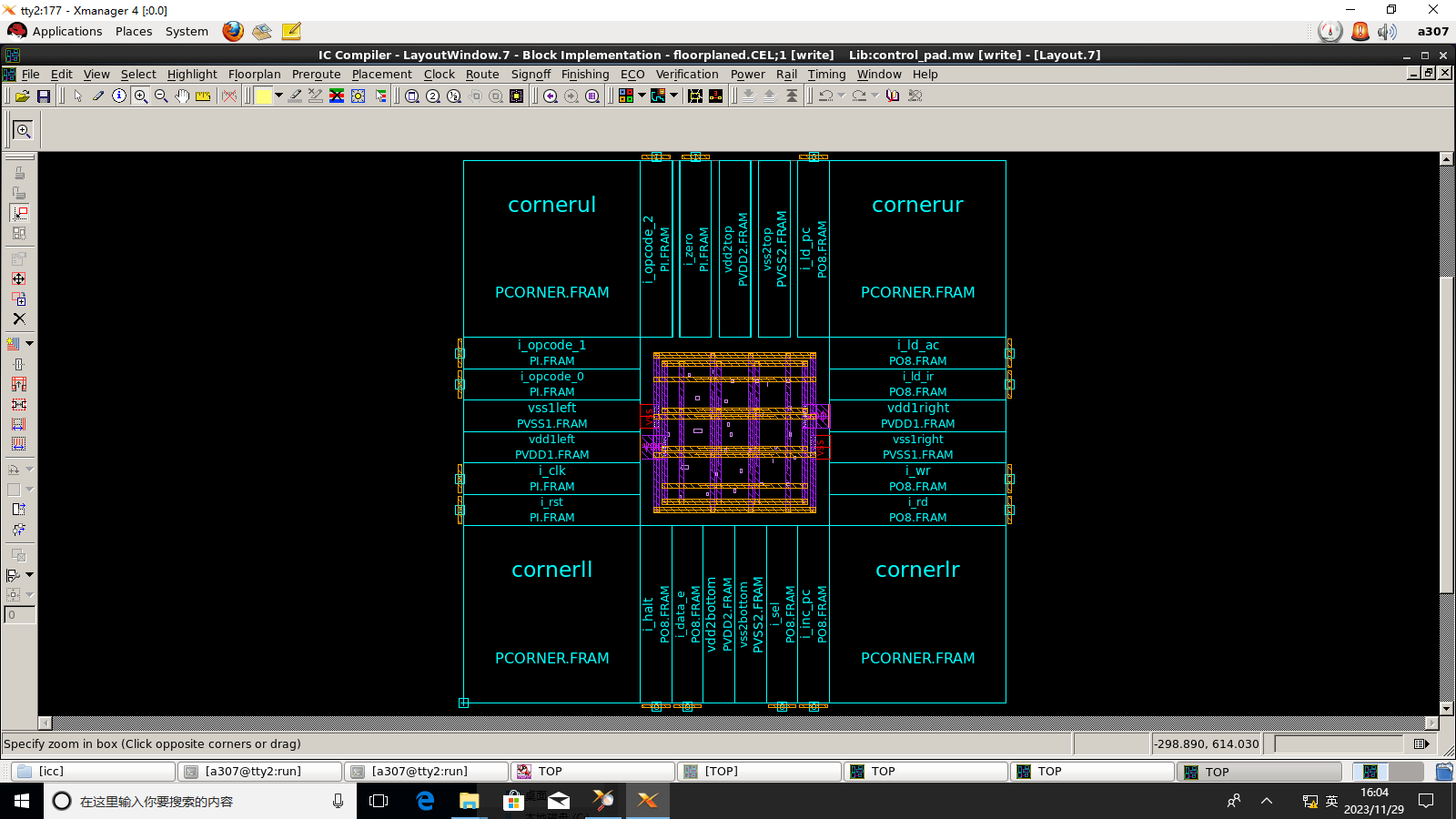
未产生core ring和mesh：



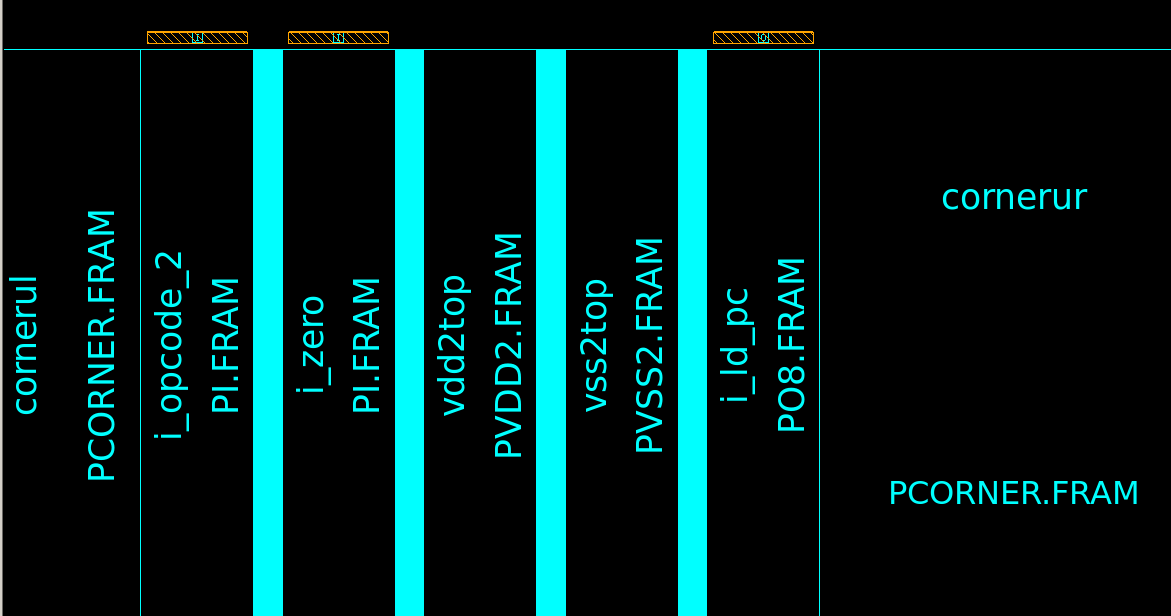
产生core ring和mesh：



电源线和电源pad连接后：



标准填充单元pad填充后：



本实验中，在scripts目录下创建ring和mesh，分别形成环状和网状的电源分配。

## 

## 三、布局

### 1、简介

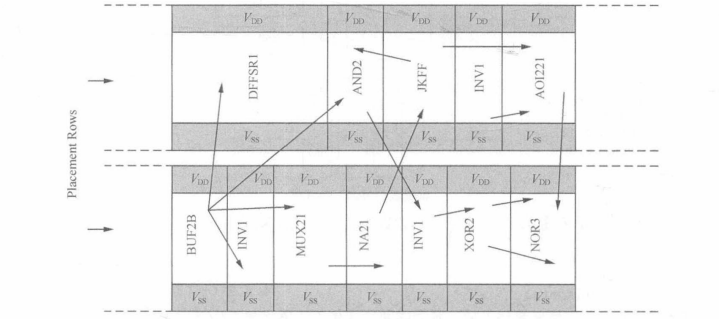
在布图规划结束后，芯片的大小、电源网络、macro 的位置已经确定了，接下来的工作是进行标准单元的布局。

布局工作是时序驱动的，即布局出来的结果要满足时序的要求。ICC时序分析采用静态时序分析（STA），STA必须要获得单元延时和线延时。

core area标准单元并不是可以随意摆放的，所有的标准单元都被设计成等高不等宽，因此可以被放入同样的placement row里面。

本实验中，在scripts目录下创建place.tcl，完成布局内容。

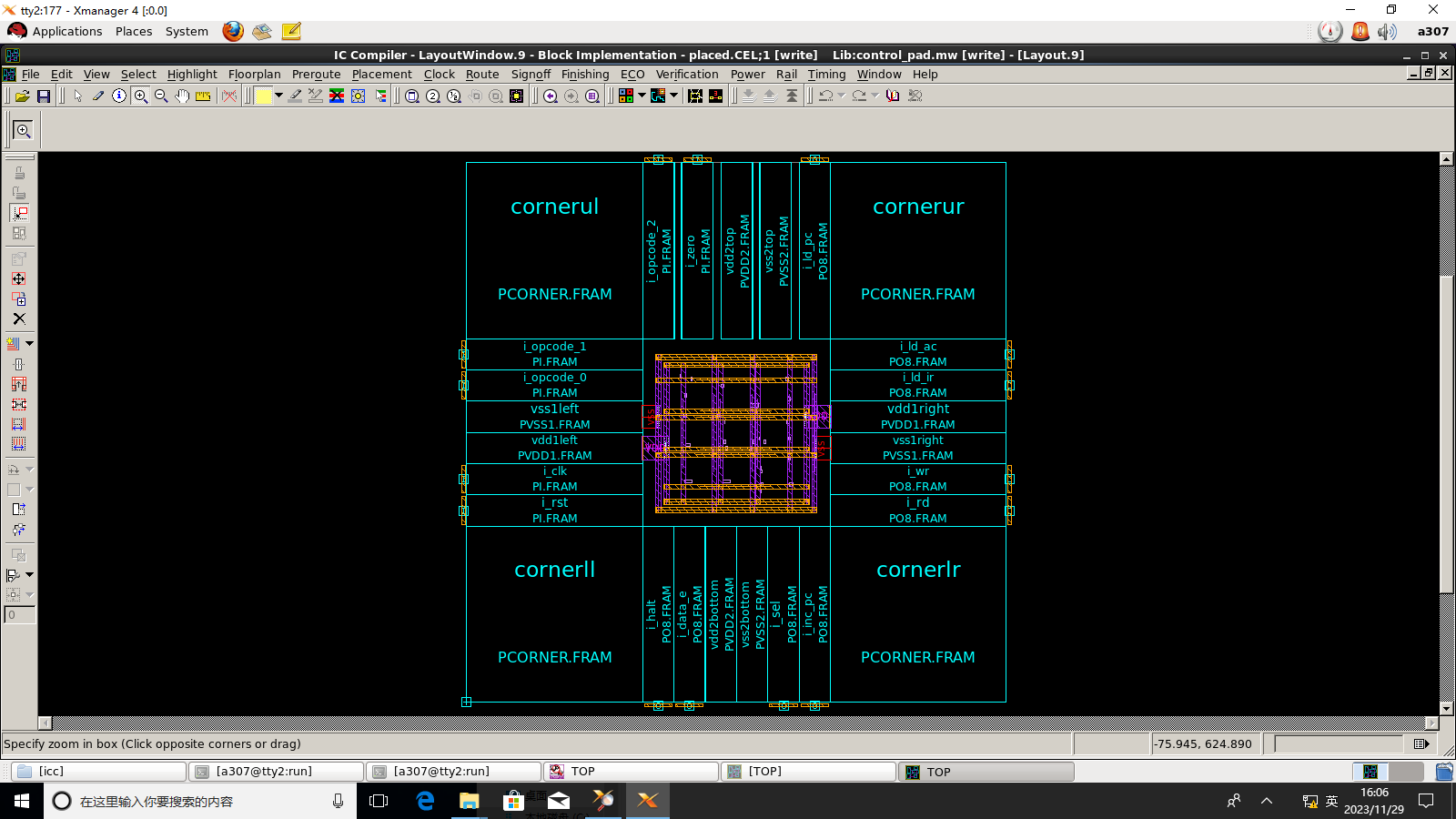
core area标准单元如图所示



### 

### 2、实验结果

布局后的图形页面如下：



## 

## 四、时钟树综合

### 1、简介

决定时钟频率的主要因素有两个：意识组合逻辑部分的最长电路延迟。而是同步原件内的时钟偏斜。

ICC时钟树综合分为以下三个步骤：

1、设置时钟综合目标；

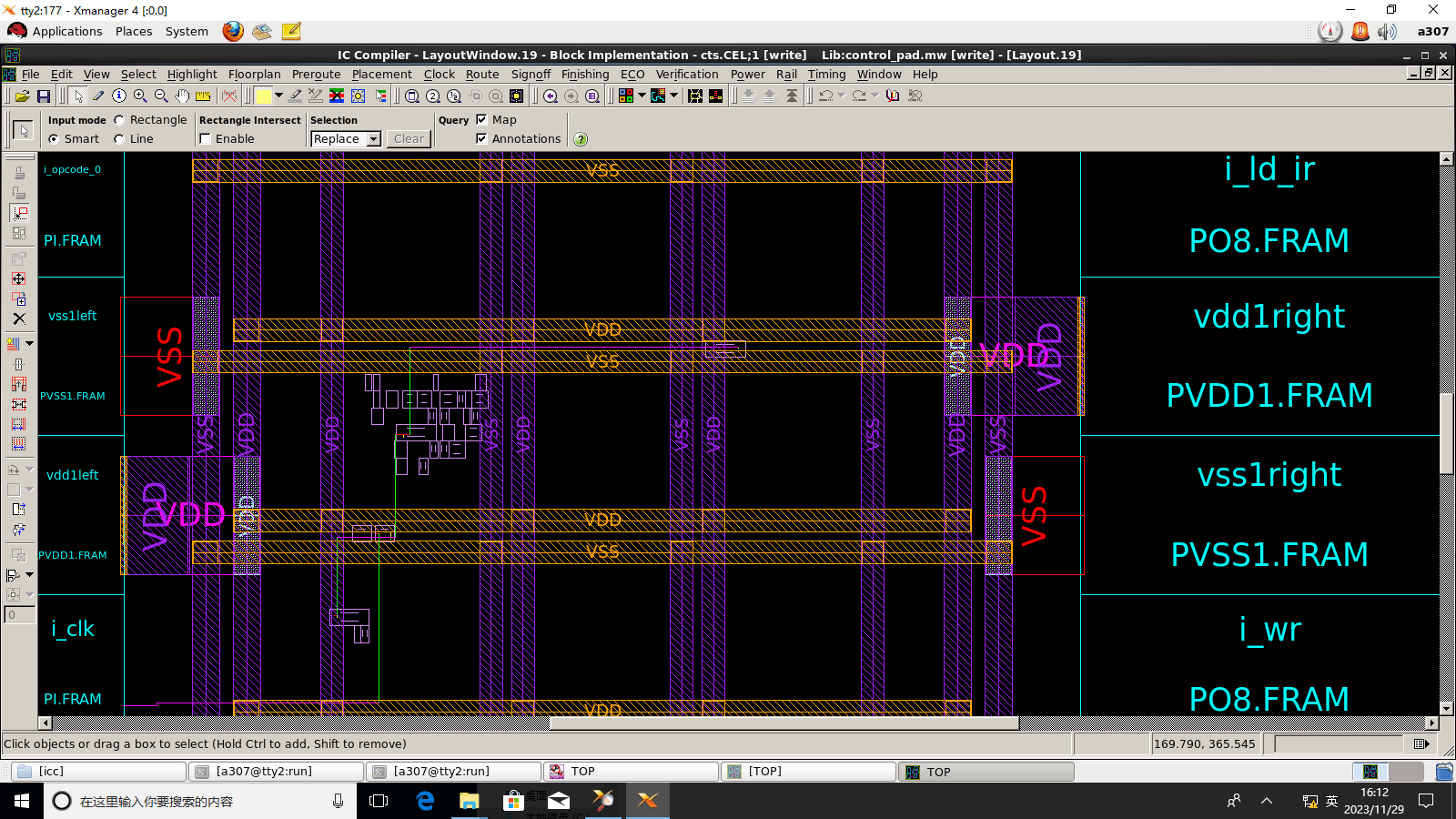
2、设置时钟树综合过程中优先使用的buffer；

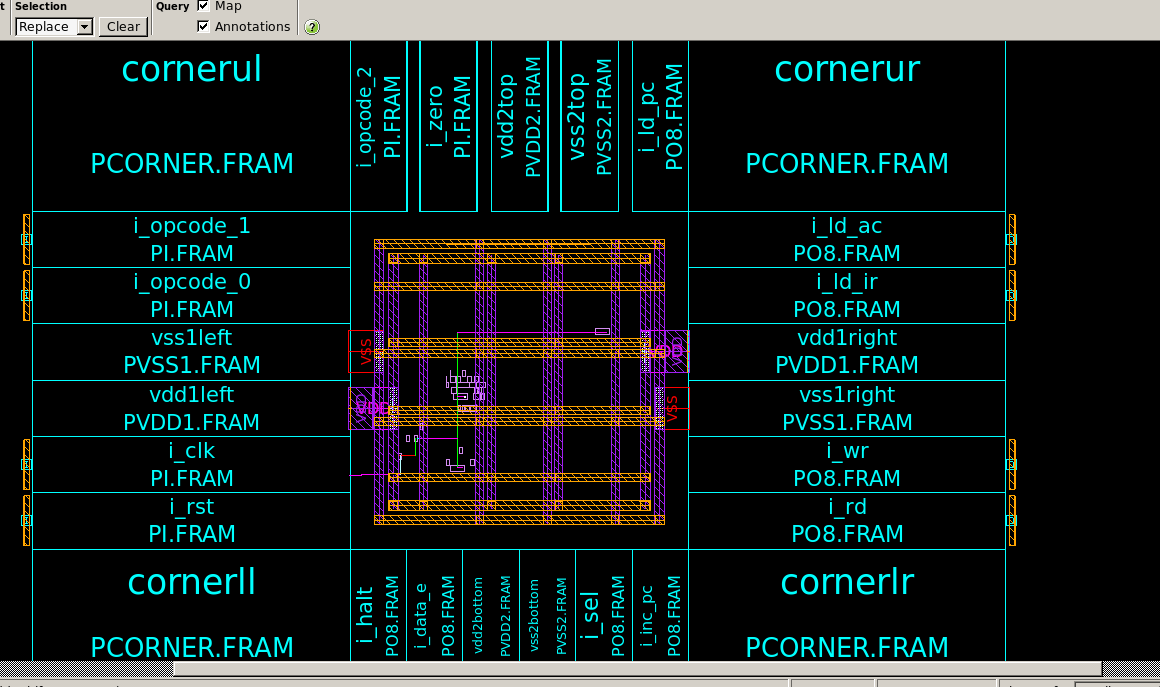
3、进行具体的CTS综合阶段。一是满足CTS设计规则要求，而是时钟树优化，三是有新的buffer会插入以满足插入延迟的需要。

本实验中，时钟树综合部分的代码为scripts目录下的cts.tcl文件。

### 2、实验结果

时钟树综合后的图形界面如下：





时钟树综合的细节如下图所示：



## 五、布线

### 1、简介

主要有以下四个步骤

1、全局布线：是为合计中还没有布线的连线规划处布线路径，确定其答题位置及走向，并不做实际的连接，全局布线已经把布线路径映射到了特定的铝线层；

2、布线通道分配：把每一连线分配到一定的布线通道上，并且对连线进行实际布线；

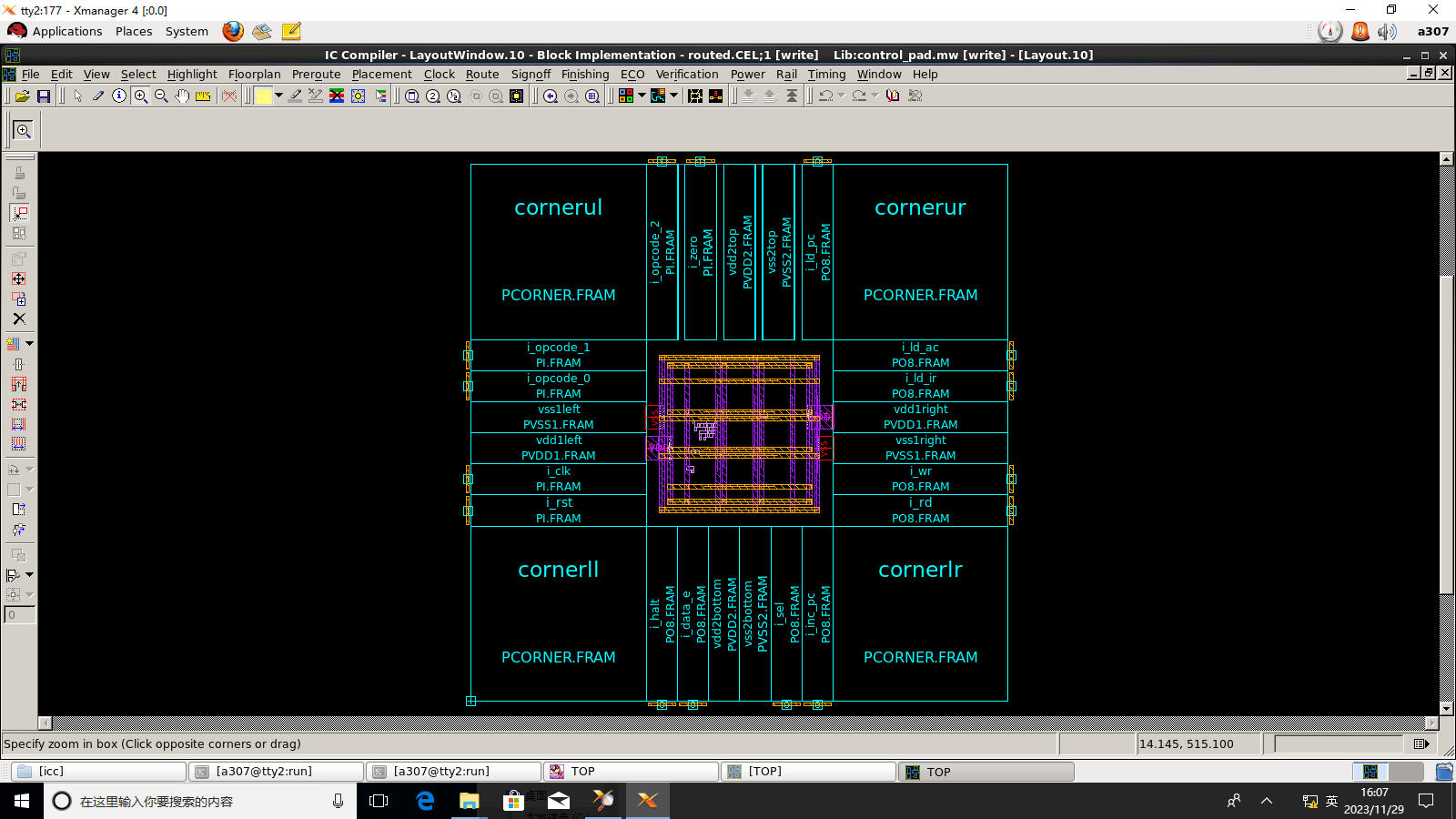
3、详细布线：使用全局布线和布线通道分配过程中产生的路径进行布线和步孔；

4、布线修补：用于修复在详细布线中没有完全消除的DRC违规。

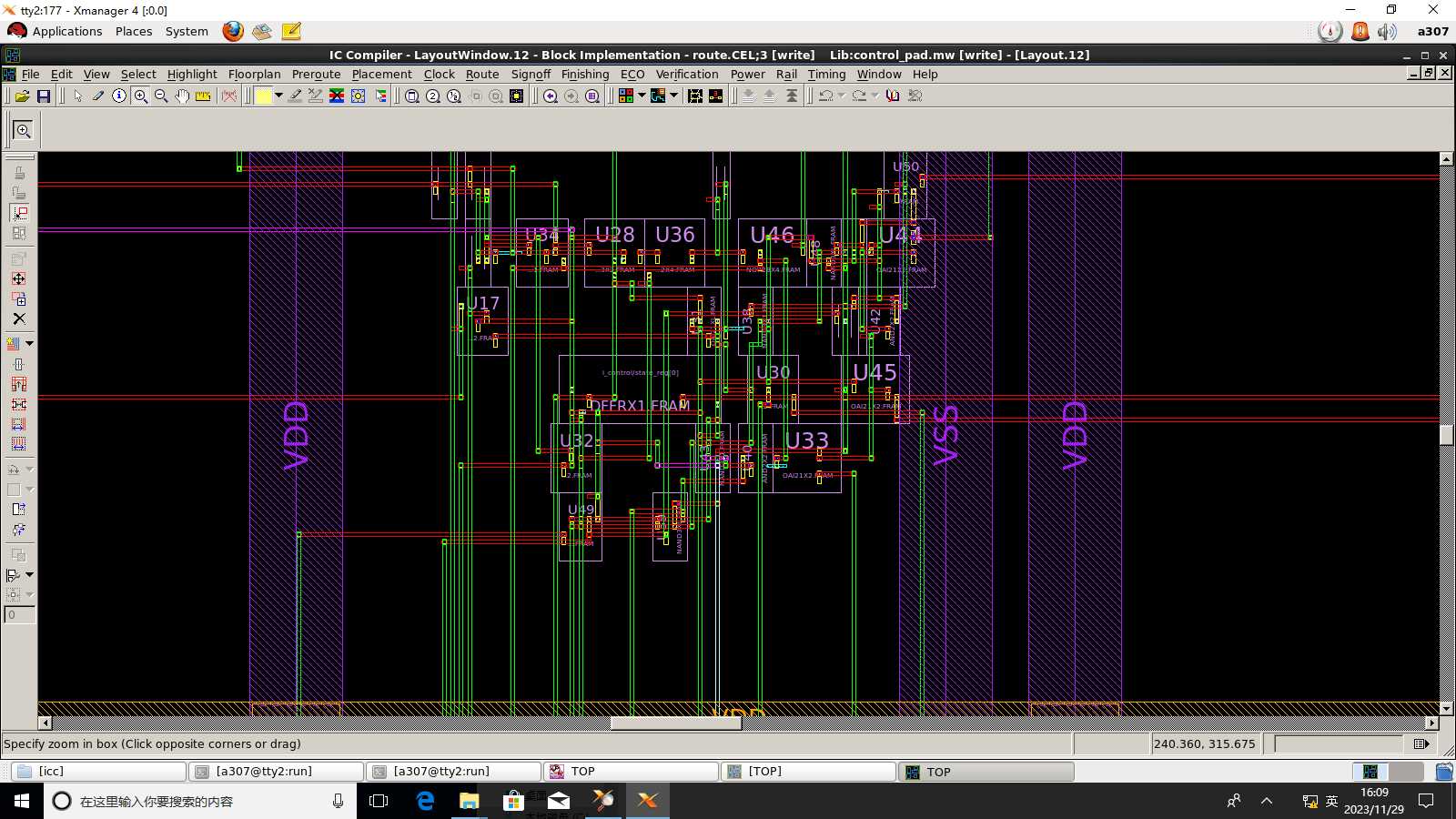
本实验中，时钟树综合部分的代码为scripts目录下的cts.tcl文件。

### 2、实验结果

布线后的图形界面如下：



布线的细节如下图所示：



## 

## 六、参数提取和后仿真

### 1、简介

**参数提取**：

用Synopsys的ICC工具能编写SPEF文件，利用prime times时序分析工具可以把SPEF文件转化为SDF文件。

**后仿真**：

版图设计完成后提取芯片内部寄生参数得到最准确的门延时和互连线延时的仿真，包括逻辑仿真、时序仿真、功耗仿真、电路可靠性分析等。。

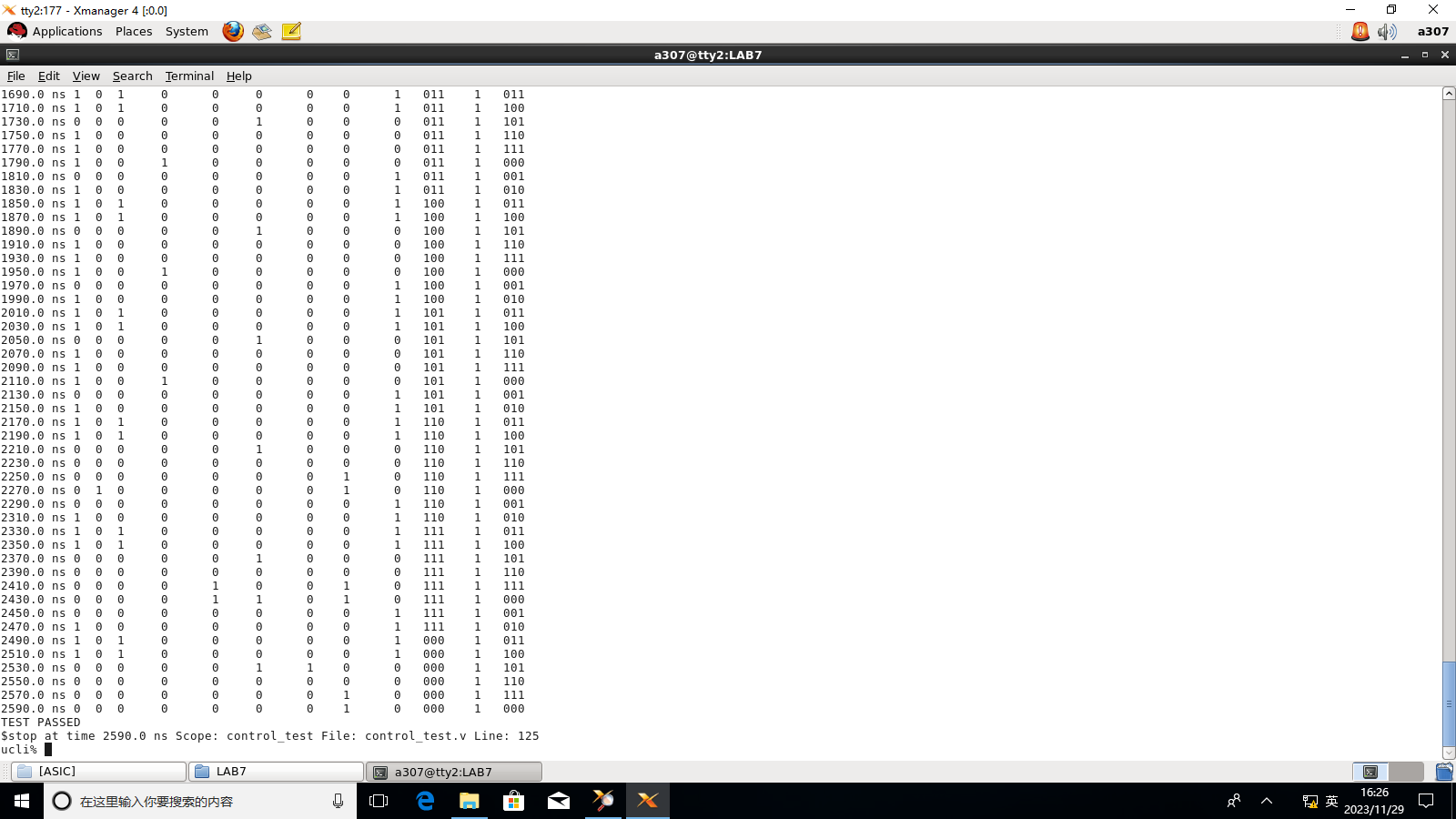
实验中，利用命令导出布线的网表和寄生参（SPEF）后生成SDF延迟信息。

### 

### 2、实验结果

仿真结果如下图所示：

**仿真分析**：结果显示”TEST PASSED”说明之前的设计操作都是正确的。



## 版图仿真.波形PNG时钟波形如图所示：

## 七、物理验证

物理验证主要有：

①DRC（设计规则检查）：检查版图的几何图形是否符合工艺规则的要求，以便芯片能在工艺线上生产出来；

②ERC（电气规则检查）： 主要是检查版图电性能（如衬底是否正确接电源或地、有无栅极悬空）以保证各器件能正常工作；

③LVS（电路规则检查）：把设计得到的版图和逻辑网表进行比较并检查各器件大小和连接关系是否一致。首先从版图中抽取出版图所确定的网编文件，然后将抽取出的网表文件与电路网表文件比较。

# 

# 七、实验总结

这次实验并不太难，重点在于考察我们对集成芯片设计过程的理解。、

尽管我学过ASIC设计课程，但在专业实验中，我还是学到了很多关于verilog语法，尤其是verilog系统函数，芯片设计和芯片测试的基础知识。从观看第一节课到成功设计出版图，我受益匪浅。特别是在课堂上编写CPU的测试文件，这个过程迫使我们去理解设计出这个CPU的功能和方法。当然，在学习过程中遇到了很多问题，我经常向同学请教，非常感谢他们对我提出的一些简单问题的耐心解答和讲解。不同的同学可能会遇到不同的问题，有时候我们互相帮助查找错误，结果两边的问题都得到了解决。这告诉我，在学习和实验中，一个人的努力可能会有缺陷，但对其他人来说可能并非如此，所以我们一定要学会合作，在帮助同学的过程中不仅能得到别人的帮助，还能提升自己的能力。实验的参考书写得非常好，书中解答了很多疑惑的问题，包括逻辑编写还是细节处理的问题，但是可惜的是第六章有些代码空格有错误，LAB3测试与书上结果有出入。感谢老师的付出！