

# 目 录

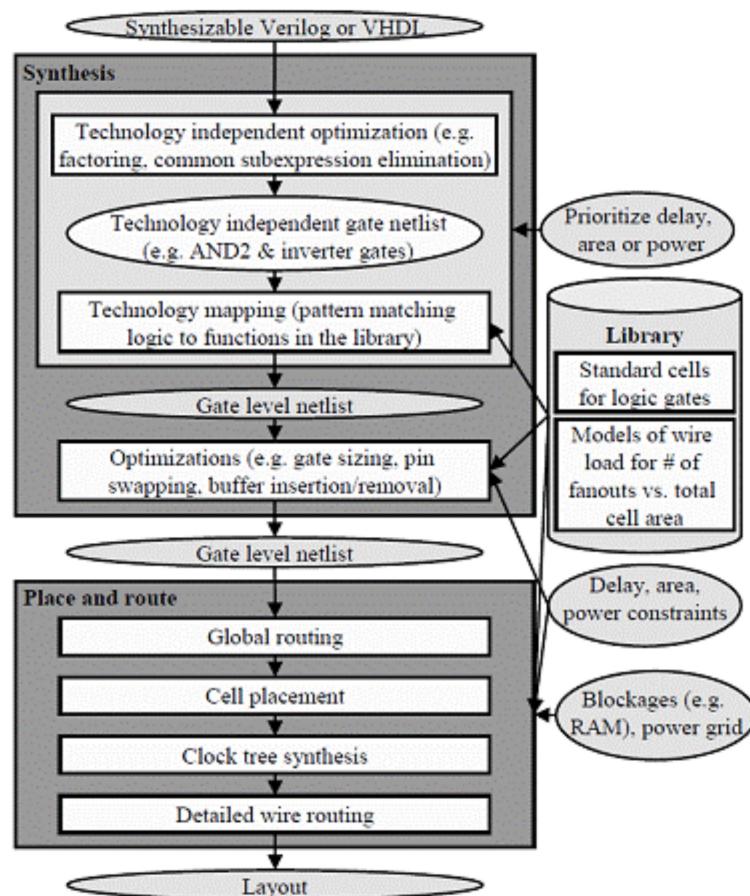
前言 .....	3
ASIC 专业实验要求 .....	7
第一部分 语言级仿真 .....	18
LAB1 简单的组合逻辑设计 .....	18
LAB2 简单时序逻辑电路的设计 .....	21
LAB3 简单时序逻辑电路的设计 .....	25
LAB4 用 always 块实现较复杂的组合逻辑电路 .....	30
LAB5 存储器电路的设计 .....	34
LAB6 设计时序逻辑时采用阻塞赋值与非阻塞赋值的区别 .....	37
LAB7 利用有限状态机进行复杂时序逻辑的设计 .....	40
LAB8 通过模块之间的调用实现自顶向下 CPU 的设计 .....	48
第二部分 电路综合 .....	54
第三部分 版图设计 .....	57
附录一 Verilog 语言要素 .....	77
附录二 关于 VI 编辑器 .....	100



# 前 言

随着工艺的发展，半导体芯片的集成化程度越来越高，设计的系统越来越复杂，规模越来越大，性能的需求越来越高，功耗也越来越大，给芯片设计工程师和EDA厂商带来了新的挑战。芯片的设计方法也随着发生了改变，经历了从早期的手工设计阶段、计算机辅助设计阶段，计算机辅助工程阶段，电子自动化设计阶段，发展到系统芯片阶段。

下图给出了芯片设计的典型流程，示例主要采用Synopsys公司的EDA工具：



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。然后设计者根据硬件设计所划分出的功能模块，进行模块设计或者复用已有的IP核，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。门级网表使用逻辑单元对电路进行描述，采用例化的方法组成电路，以及定义电路的层次结构。

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。

2、逻辑综合。建立设计和综合环境，将RTL源代码输入到综合工具，例如Design Compiler，给设计加上约束，然后对设计进行逻辑综合，得到满足设计要求的门级网表。门级网表可以以ddc的格式存放。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。

3、版图规划。在得到门级网表后，把结果输入到JupiterXT做设计的版图规划。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。

4、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。EDA工具广泛支持物理综合，即将布局和优化与逻辑综合统一起来，引入真实的连线信息，减少时序收敛所需要的迭代次数。把设计的版图规划和门级网表输入到物理综合工具，例如Physical Compiler进行物理综合和优化。在PC中，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。

5、静态时序分析（STA）、形式验证（FV）和可测性电路插入（DFT）。静态时序分析是一种穷尽分析方法，通过对提取的电路中所有路径的延迟信息的分析，计算出信号在时序路径上的延迟，找出违背时序约束的错误，如建立时间和保持时间是否满足要求。在后端设计的很多步骤完成后都要进行静态时序分析，如逻辑综合之后，布局优化之后，布线完成之后等。

形式验证是逻辑功能上的等效性检查，根据电路的结构判断两个设计在逻辑功能上是否相等，用于比较RTL代码之间、门级网表与RTL代码之间，以及门级网表之间在修改之前与修改之后功能的一致性。

可测性设计。通常，对于逻辑电路采用扫描链的可测性结构，对于芯片的输入/输出端口采用边界扫描的可测性结构，增加电路内部节点的可控性和可观测性，一般在逻辑综合或物理综合之后进行扫描电路的插入和优化。

6、后布局优化，时钟树综合和布线设计。在物理综合的基础上，可以采用Astro工具进一步进行后布局优化。在优化布局的基础上，进行时钟树的综合和布线。Astro在

设计的每一个阶段，都同时考虑时序、信号、功耗的完整性和面积的优化、布线的拥塞等问题。其能把物理优化、参数提取、分析融入到布局布线的每一个阶段，解决了设计中由于超深亚微米效应产生的相互关联的复杂问题。

7、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。有了设计的版图，使用Sign-Off参数提取的工具，如Star-RCXT进行寄生参数的提取，其可以设计进行RC参数的提取，然后输入到时序和功耗分析工具进行时序和功耗的分析。

8、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。如Primestime-SI能进行时序分析，以及信号完整性分析，可以做串扰延迟分析、IR drop（电压降）的分析和静态时序分析。在分析的基础上，如发现设计中还有时钟违规的路径，Primestime-SI可以自动为后端工具如Astro产生修复文件。PrimePower具有门级功耗的分析能力，能验证整个IC设计中的平均峰值功耗，帮助工程师选择正确的封装，决定散热和确证设计的功耗。在设计通过时序和功耗分析之后，PrimeRail以Star-RCXT、HSPICE、Nanosim和PrimeTime的技术为基础，为设计进行门级和晶体管级静态和动态的电压降分析，以及电迁移的分析。

9、ECO（工程修改命令）修改。当在设计的最后阶段发现个别路径有时序问题或者逻辑错误时，有必要对设计的部分进行小范围的修改和重新布线。ECO修改只对版图的一小部分进行修改而不影响到芯片其余部分的布局布线，保留了其他部分的时序信息没有改变。

10、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）。将版图输入Hercules，进行层次化的物理验证，以确保版图和线路图的一致性，其可以预防、及时发现和修正设计在设计中\_的问题。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。

在完成以上步骤之后，设计就可以签收、交付到芯片制造厂了（Tapeout）。

本次ASIC实验主要是完成一个简单CPU的设计，对CPU进行语言级仿真，对其中的控制模块进行综合，检查其功耗和最高工作速度，进行门级仿真，并保证门级仿真结果正

确之后，进行控制器的版图设计及验证。

对于这样一个CPU设计，实际工作中都是通过将其划分为相对独立的小模块进行设计，然后对这些模块分别通过验证，最后再将设计正确的模块集成起来，完成一个完整CPU的设计。为了方便同学们理解和设计，在CPU进行分模块设计的过程中，实验还将Verilog HDL中的关键语法知识进行了复习与运用。

根据上述原则，本实验将其分成八个步骤来完成，从简单计数器开始，到寄存器的设计，再到存储控制器设计，以及CPU状态控制器等，将依次用到组合逻辑与时序逻辑、阻塞赋值与非阻塞赋值、状态机的设计等。同时，每次课设计的小模块都将是最终CPU设计的一个组成部分，需要通过验证以保证最终CPU设计调用的正确性。

在附录中，为大家提供了一些网上参考资料，便于大家查阅。当然更建议同学们带参考书和先修课程资料查阅，以巩固相关概念和知识点。

# ASIC 专业实验要求

## 1. 远程系统的登陆方法

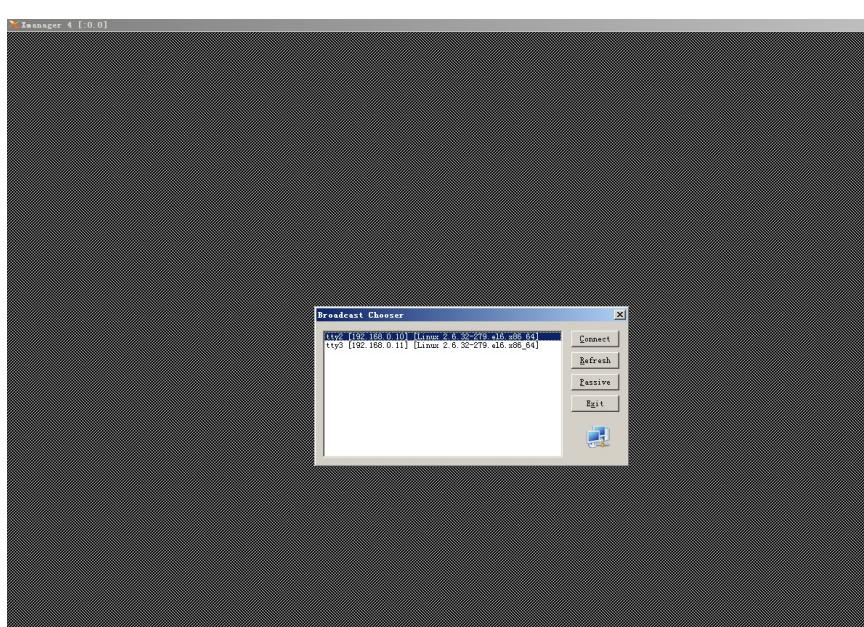
首先在电脑上找到下面图标双击打开



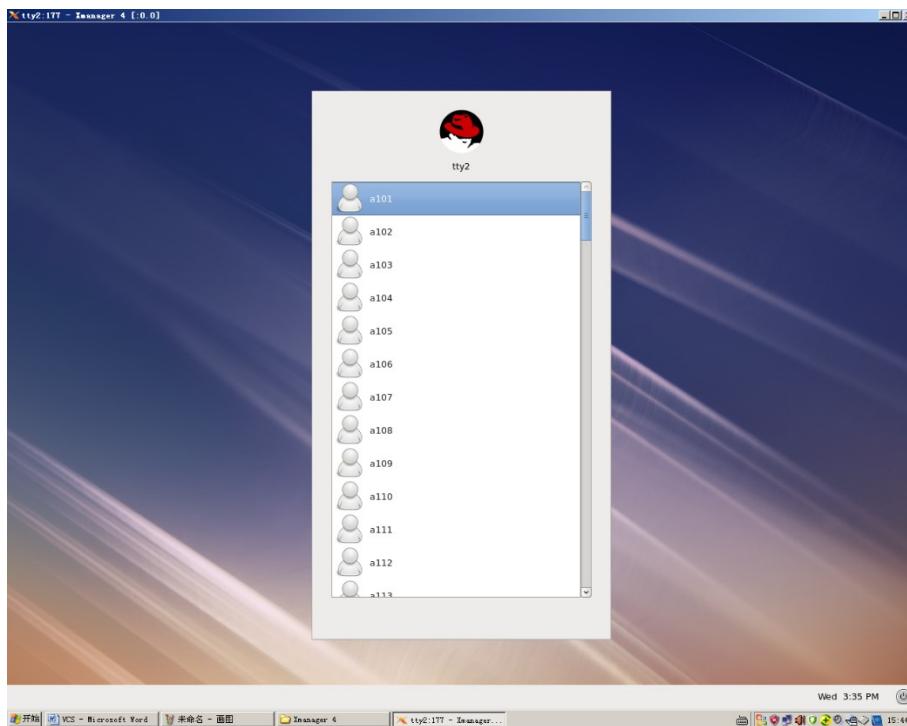
得到如下页面



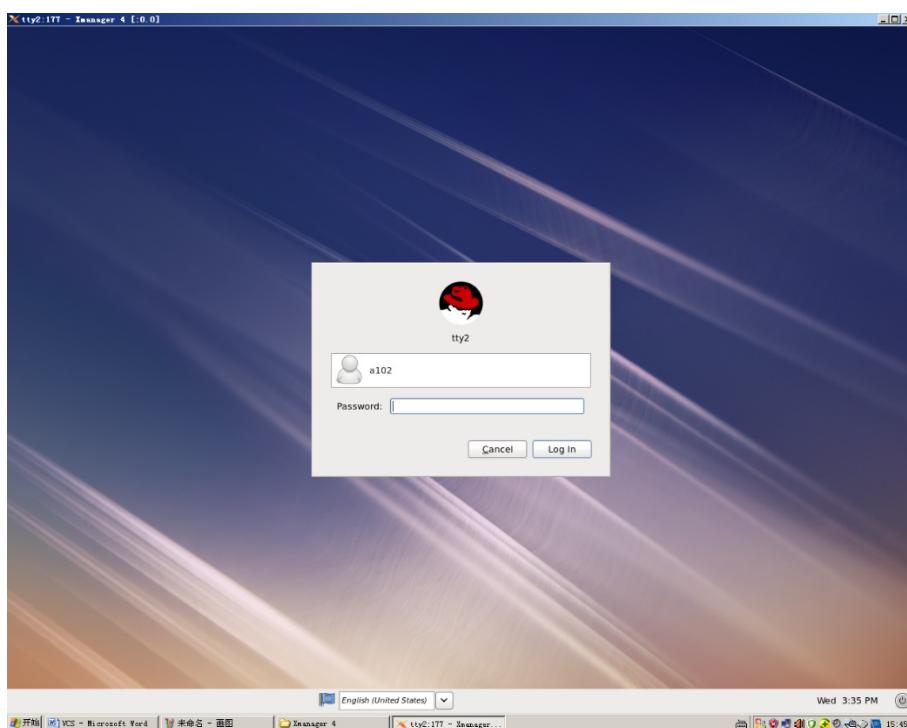
双击 进入远程登录页面



选择第一个或者第二个网址，点击 connect，进行连接，进入到 linux 登录页面

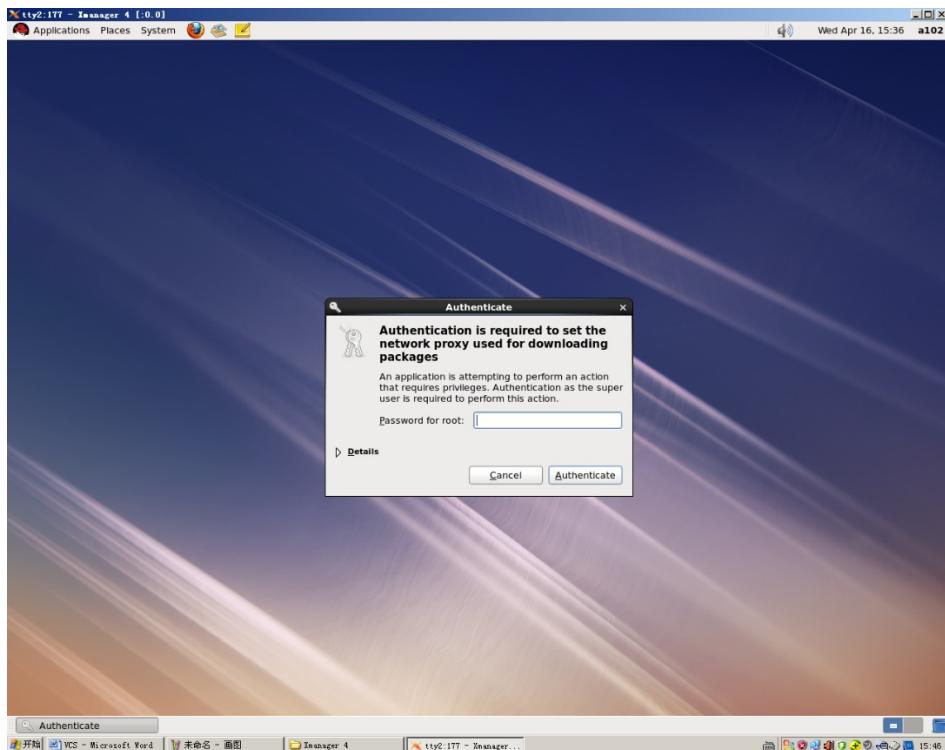


选择一个用户点击，登录



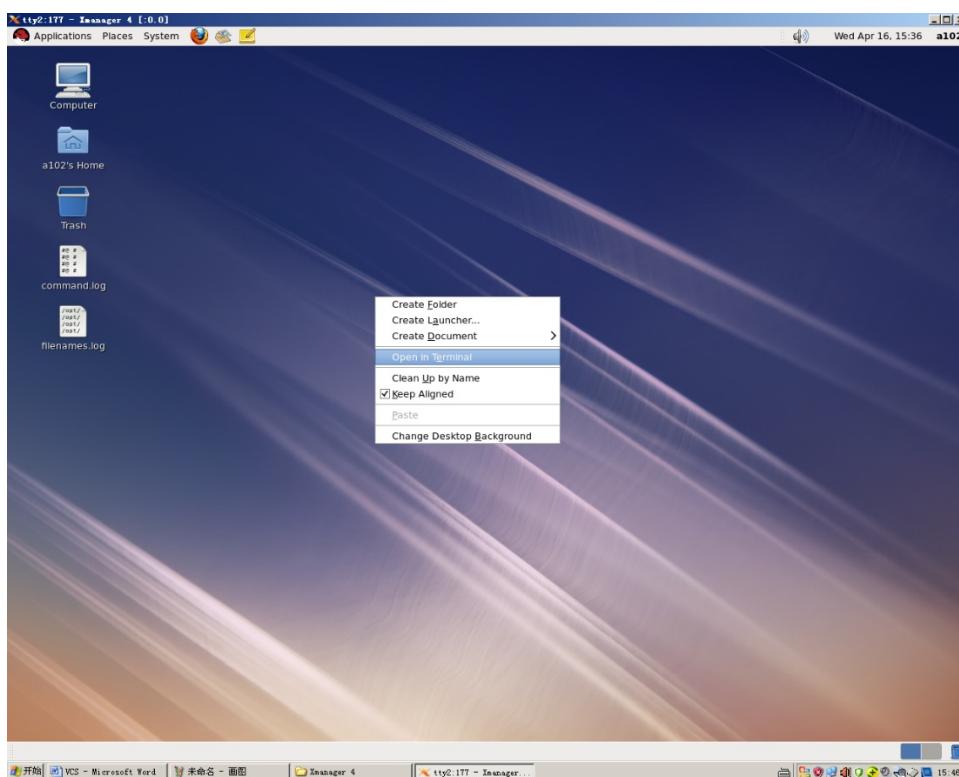
密码统一为：usrusr

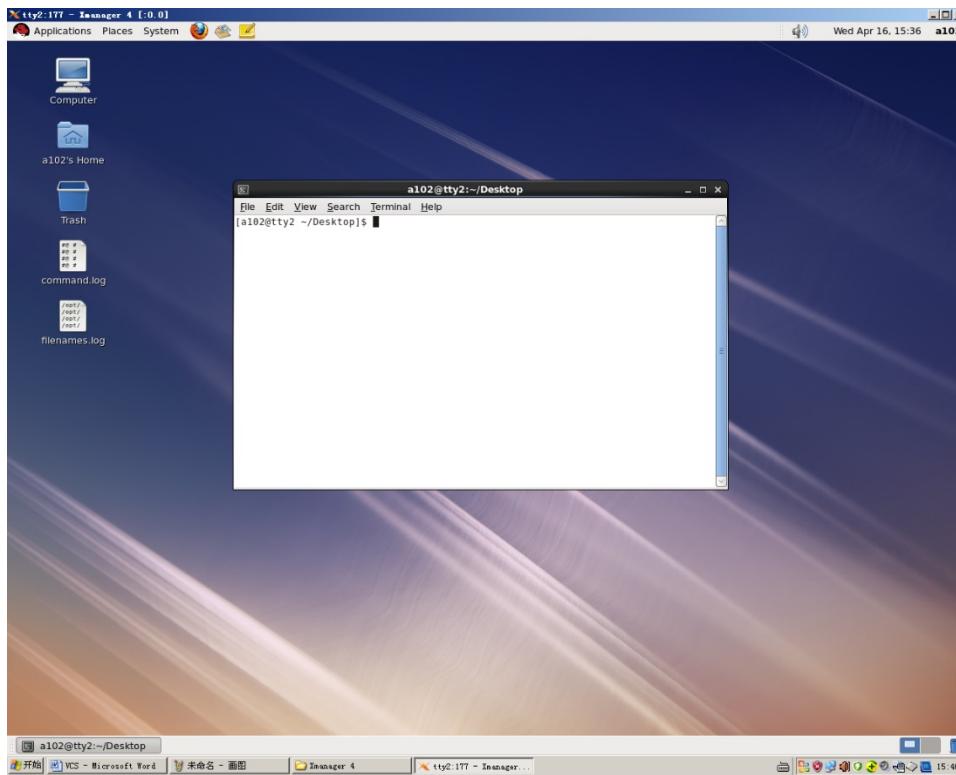
登录成功



点击 cancel 即可。至此远程登录成功。

右击鼠标，选择 open in Terminal，进入命令窗口；





输入: [a102@tty2 ~/Desktop]\$ cd ~

进入 a102' s Home (根据不同用户, 用户名可能不同)

d: 建立一个 ASIC 文件夹, 并且按照实验内容, 为每个实验建立子目录 lab\*, 请首先理解实验内容。

**文件的输入和编辑:** 用 vi 进行 verilog 程序(电路文件和测试文件)输入!

**注:**

**关于 Linux 的基本命令:**

mkdir tmpname	建一个目录
cd /tmp	进入目录 (文件夹)
cd	返回 home 下
cd ..	返回上一级目录
ls	显示当前目录的内容
vi filename	打开或新建一个文件
more	查看文件内容 (安全)
find	查找文件
cp-i file1 file2	复制文件
rm -i file	删除文件
rm -rf tmp	删除目录

### 关于 Vi 编辑器的命令:

**vi<文件名称>** vi 可以自动帮你载入要编辑的文件或是开启一个新文件。

[ESC] 切换到命令模式。

:q, :wq 退出、存档后再退出（注意冒号）。

:set all 打印所有选项

:set number 每行前打印行号

:set showmode 显示是输入模式还是替换模式

:set noic 查找时忽略大小写

:set list 显示制表符(^I)和行尾符号

:set ts=8 为文本输入设置 tab stops

:set window=n 设置文本窗口显示 n 行

### 相关命令

	命 令	作 用		命 令	作 用
模 式	<a>	在光标后输入文本	复 制 粘 贴	</yw>	将光标所在单词拷入剪贴板
	<A>	在当前行末尾输入文本		<yy>	将当前行拷入剪贴板
	<i>	在光标前输入文本		<p>	将剪贴板中的内容粘贴在光 标后
	<I>	在当前行开始输入文本		<P>	将剪贴板中的内容粘贴在光 标前
	<o>	在当前行后输入新一行			
	<O>	在当前行前输入新一行			
标 移 动	<b>	移动到当前单词的开始	删	<x>	删除光标所在的字符
	<e>	移动到当前单词的结尾		<dw>	删除光标所在的单词
	<w>	向前移动一个单词		<dd>	删除当前行
	<h>	向前移动一个字符	查 询	</X>	向前查询 X
	<j>	向上移动一行		<?X>	向后查询 X
	<k>	向下移动一行		<n>	向前继续查询
	<l>	向后移动一个字符		<N>	向后继续查询

实验要求:

### 基础实验

按照实验内容指导书的要求，在规定时间内完成。要求对 CPU 进行语言级系统仿真结果正确之后，对其中的控制器电路进行综合，并检查 timing 和 power，进行门级仿真，在保证门级仿真结果正确，最后完成控制器电路的版图设计及验证。

### 拓展实验

完成整个 CPU 版图设计。或一个帧同步检测电路版图设计

验收方式:

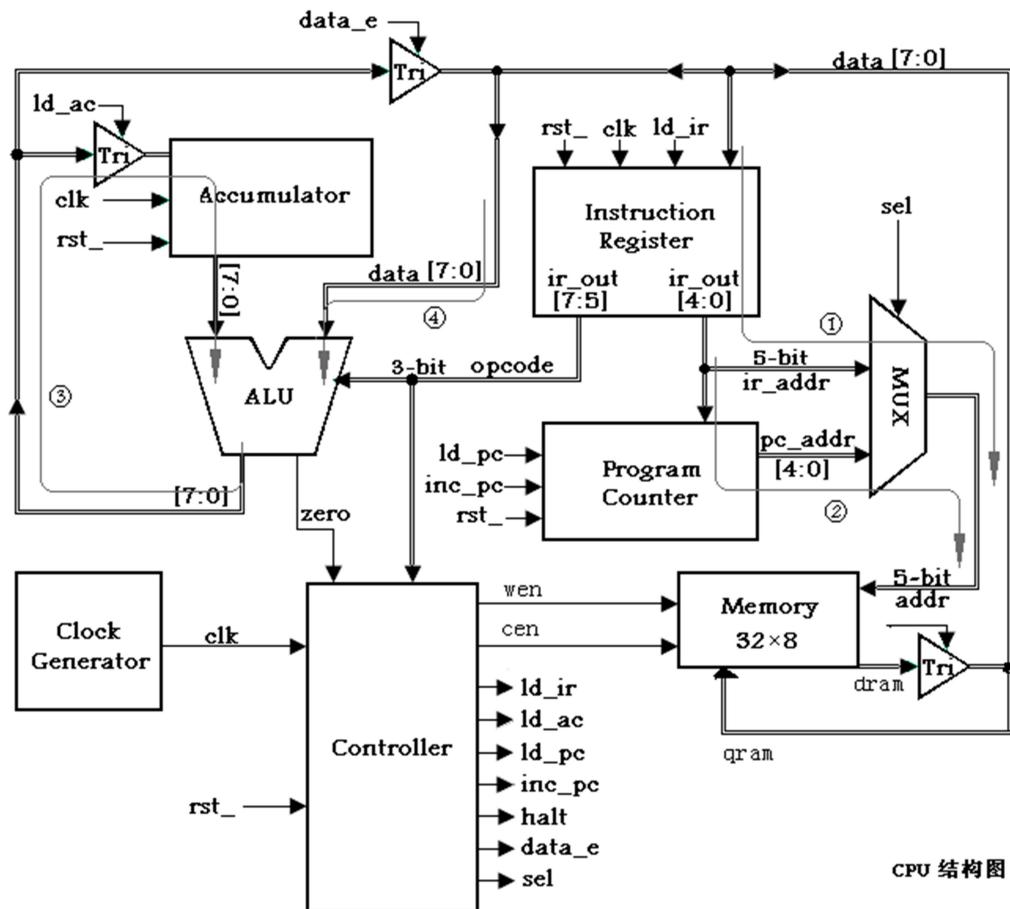
报告	30%
在线验收	70%

实验内容:

设计一个 8 位 RISC\_CPU 系统。(RISC: Reduced Instruction Set Computer) 它是一种 80 年代才出现的 CPU，与一般的 CPU 相比，不仅只是简化了指令系统，而且通过简化指令系统使计算机的结构更加简单合理，从而提高了运算速度。从实现的方法上，它的时序控制信号部件使用了硬布线逻辑，而不是采用微程序控制方式，故产生控制序列的速度要快的多，因为省去了读取微指令的时间。它由八个独立的逻辑部件所组成:

- (1) 时钟发生器
- (2) 指令寄存器
- (3) 累加器
- (4) ALU
- (5) 数据控制器
- (6) 状态控制器
- (7) 程序计数器
- (8) 地址选择器

其连接关系如下图：



CPU 结构图

vcs - full164 filetest1.v filename2.v -R 进行语法分析和仿真。

vcs - full164 -c submod.v 对子模块仅作语法分析。

vcs - full164 -c -f run.f 按批处理的方式作语法分析。

vcs - full164 -f run.f -R 按批处理的方式作仿真。

dve& 进入交互式仿真环境。

注：\*run.f 文件是批处理文件，一般文件中包含需要进行仿真的测试文件、电路文件和一些需要调用的库文件。是在仿真调试过程中简化命令输入的一种方式。

\*另外需要提示同学们的是，在工具做完仿真后，会自动生成波形文件和一些记录文件及仿真报告，所以同学们要重复观察以仿真的波形或仿真报告时，可以打开相关文件，不必重复仿真。

3. 依照实验内容做实验，必须将实验在规定时间内作完。在对 CPU 进行语言级系统仿真结果正确之后，对其中的控制器电路进行综合，并进行门级仿真，并且保证门级

仿真结果正确，最后完成控制器电路的版图设计。

#### 4. 验收方式:

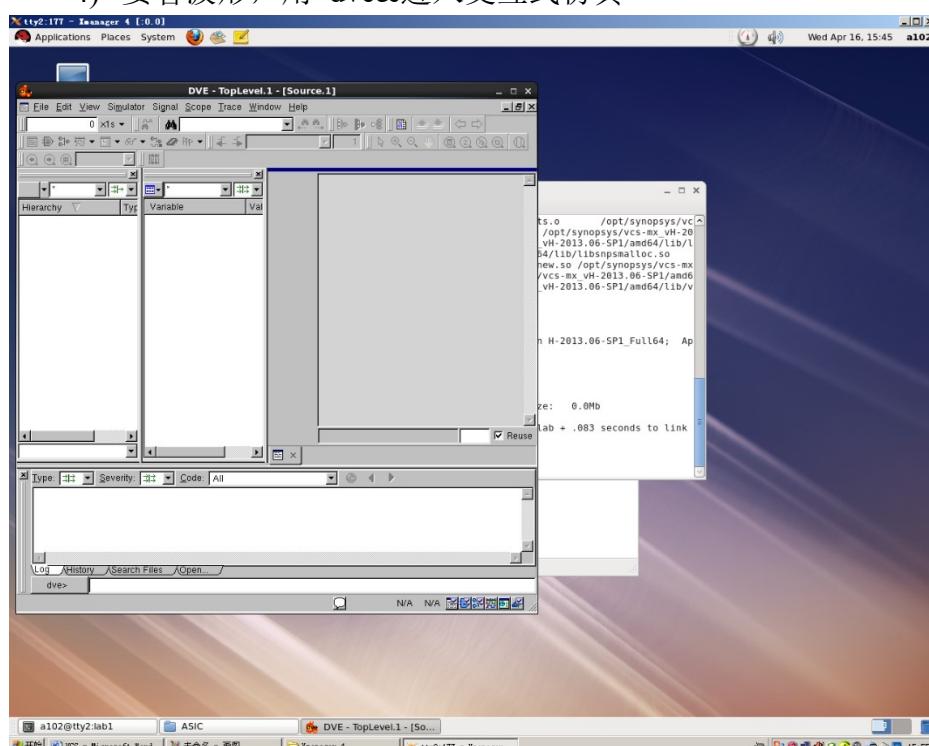
a:最后写出一份实验报告。

b:在线验证。

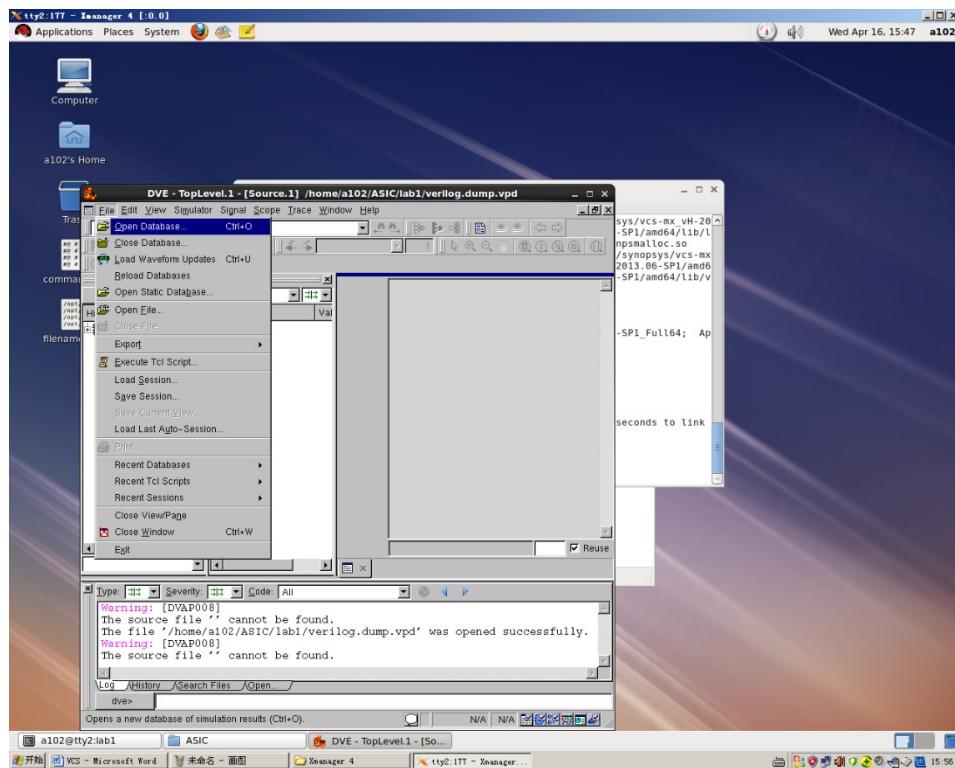
#### 5. 上机指导:

##### (一) Verilog 语言级仿真流程:

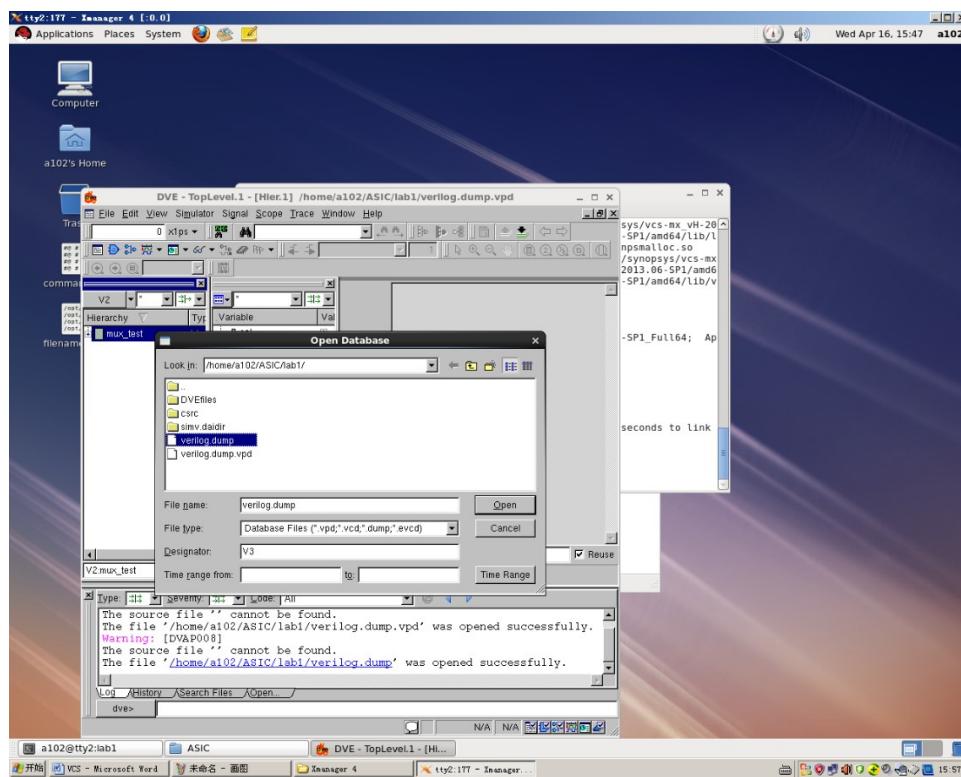
- 1) 按鼠标右键——hosts——terminal console 打开命令窗口,在命令窗口中可以输入文件和运行执行文件。
- 2) 用“mkdir 目录名”在当前目录下创建一个目录。用“vi 文件名”输入文件内容或者是打开一个文件。
- 3) 用 vcs -full64 mux\_test.v mux.v -R 进行仿真。
- 4) 要看波形, 用 dve&进入交互式仿真

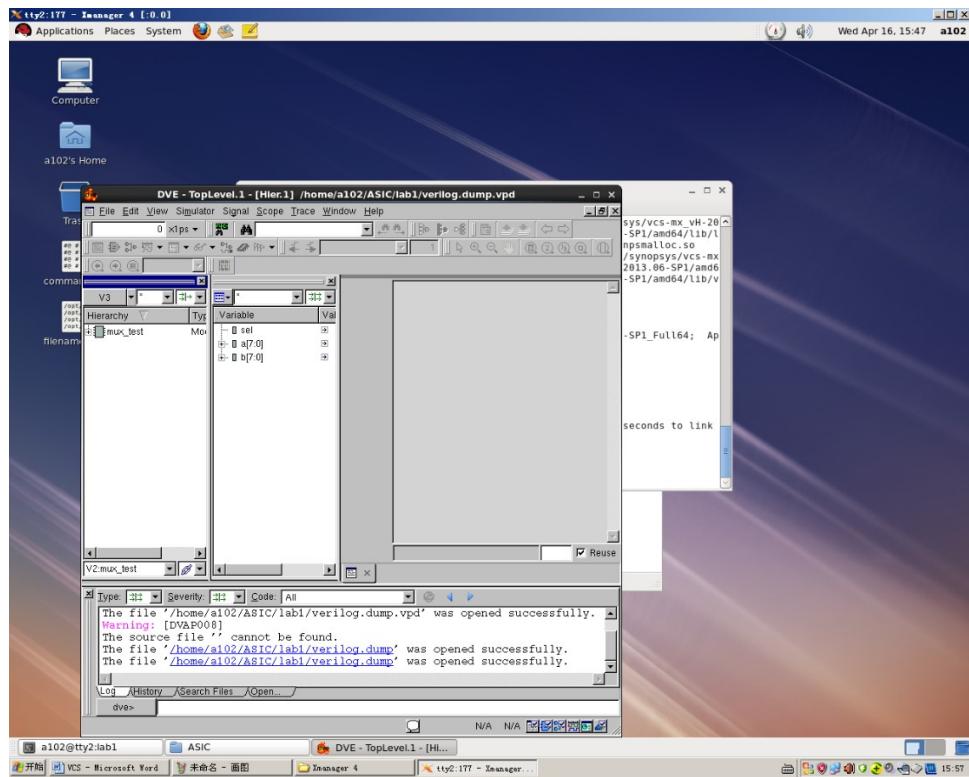


打开 file->open database

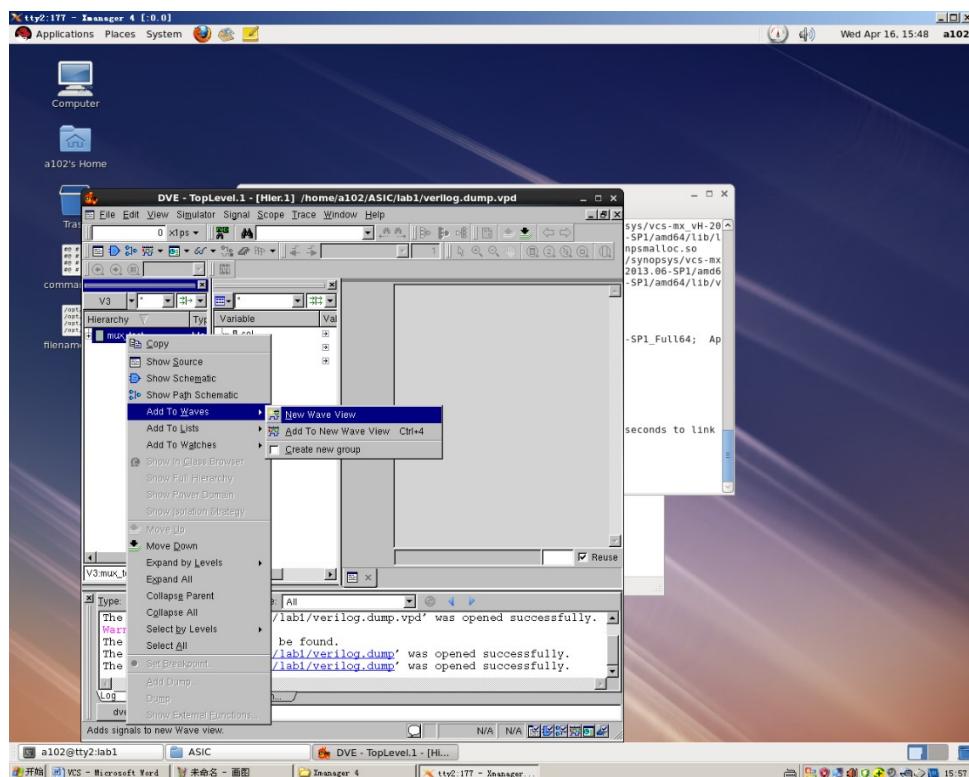


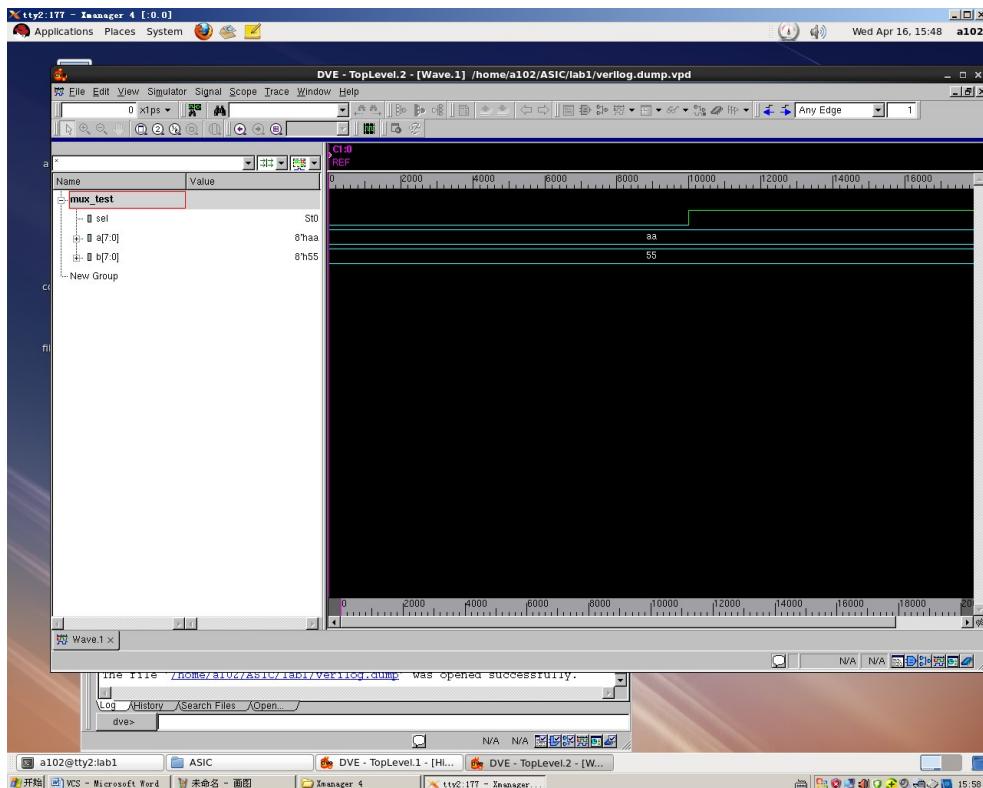
选择如图 verilog.dump 文件





右击 mux\_test 图标，选择 Add to wave ->new wave view;





至此仿真完成；

## 第一部分 语言级仿真

# LAB 1：简单的组合逻辑设计

模块源代码：

```
//-----mux.v -----
*****
* 2-TO-1 N-BIT WIDE SCALABLE MUX *
*****
```

```
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 :
(sel) ? b :
{size{1'bx}} ;
endmodule
```

测试模块用于检测模块设计得正确与否，它给出模块的输入信号，观察模块的内部信号和输出信号，如果发现结果与预期的有所偏差，则要对设计模块进行修改。

测试模块源代码：

```
*****
* TEST BENCH FOR SCALABLE MUX *
*****
```

```
`define width 8
`timescale 1 ns / 1 ns //定义时间单位。

module mux_test ;
reg [`width:1] a, b;
wire [`width:1] out;
reg sel;
```

```
// Instantiate the mux. Named mapping allows the designer to have freedom
//      with the order of port declarations. #8 overrides the parameter (NOT
//      A DELAY), and gives the designer flexibility naming the parameter.

scale_mux #(`width) m1 (.out(out), .sel(sel), .b(b), .a(a));

initial
begin

// Display results to the screen, and store them in an SHM database
$monitor($stime,, "sel=%b a=%b b=%b out=%b", sel, a, b, out);
$dumpvars(2,mux_test);

// Provide stimulus for the design
    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
```

## LAB2 简单时序逻辑电路的设计

```
//5-bit counter
'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_ )
  if ( !rst_ )
    #1.2 cnt <= 0 ;
  else
    if ( load )
      cnt <= #3 data ;
    else
      cnt <= #4 cnt + 1 ;

endmodule
```

```
*****
* TEST BENCH FOR 5-BIT COUNTER *
*****
```

```
'timescale 1 ns / 1 ns

module counter_test ;

wire [4:0] cnt ;
reg  [4:0] data ;
reg      rst_ ;
reg      load ;
reg      clk ;
```

// 例化 counter 模块

```
counter c1
(
  .cnt (cnt ),
  .clk (clk ),
  .data(data),
  .rst_ (rst_ ),
  .load(load)
);
```

// 产生仿真时钟 周期为 20ns

```
initial begin
  clk= 0 ;
  forever begin
    #10 clk = 1'b1 ;
    #10 clk = 1'b0 ;
  end
end
```

//添加监测信号

```
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
```

// 验证输出结果，若不符则输出 TEST FAILED

```
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
```

// 仿真主进程

```
initial
```

```
begin
    // SYNCHRONIZE INTERFACE TO INACTIVE CLOCK EDGE
    @(negedge clk)

    // RESET
    {rst_, load, data} = 7'b0_XXXXX; @(negedge clk) expect(5'h00);

    // LOAD 1D
    {rst_, load, data} = 7'b1_1_11101; @(negedge clk) expect(5'h1D);

    // COUNT +5
    {rst_, load, data} = 7'b1_0_11101;
    repeat(5) @(negedge clk);
    expect(5'h02);

    // LOAD 1F
    {rst_, load, data} = 7'b1_1_11111; @(negedge clk) expect(5'h1F);

    // RESET
    {rst_, load, data} = 7'b0_XXXXX; @(negedge clk) expect(5'h00);

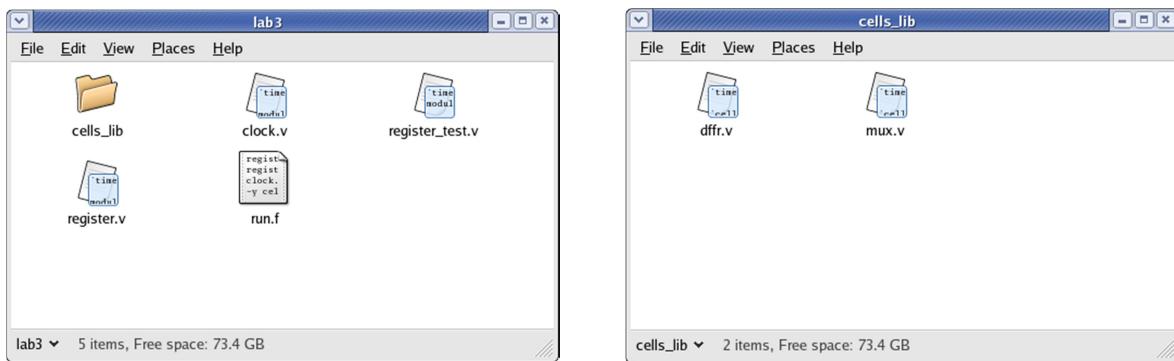
    $display( "TEST PASSED" );
    $finish;

end

endmodule
```

# LAB3 简单时序逻辑电路的设计

目录结构如下图



在 cells-lib 目录下有 mux.v 和 dffr.v 两个模块文件

mux.v:

```
'timescale 1 ns / 1 ns
```

```
`celldefine
```

```
module mux ( out, sel, b, a );
    output out ;
    input sel ;
    input b ;
    input a ;
    not ( sel_, sel ) ;
    and ( selb , sel_ , b ) ;
    and ( sela , sel_ , a ) ;
    or ( out , selb , sela ) ;
endmodule
`endcelldefine
```

dffr.v:

```
*****
```

```
* D FLIPFLOP *
```

```
*****
```

```
'timescale 1 ns / 1 ns
```

```
`celldefine
```

```
module dffr ( q, q_, d, clk, rst_ );
    outputq ;
    outputq_ ;
    input d ;

```

```

input clk ;
input rst_ ;

nand n1 ( de, dl, qe ) ;
nand n2 ( qe, clk, de, rst_ ) ;
nand n3 ( dl, d, dl_, rst_ ) ;
nand n4 ( dl_, dl, clk, qe ) ;
nand n5 ( q, qe, q_ ) ;
nand n6 ( q_, dl_, q, rst_ ) ;

endmodule
`endcelldefine

```

电路文件： register.v:

```

*****  

* 8-bit REGISTER *  

*****  

`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 ;

//自行添加 dffr 及 mux 模块调用代码
dffr d0(out[0],,n1[0],clk,rst_),
      d1(out[1],,n1[1],clk,rst_),
      d2(out[2],,n1[2],clk,rst_),
      d3(out[3],,n1[3],clk,rst_),
      d4(out[4],,n1[4],clk,rst_),
      d5(out[5],,n1[5],clk,rst_),
      d6(out[6],,n1[6],clk,rst_),
      d7(out[7],,n1[7],clk,rst_);

mux m0(n1[0],load,data[0],out[0]),
      m1(n1[1],load,data[1],out[1]),
      m2(n1[2],load,data[2],out[2]),
      m3(n1[3],load,data[3],out[3]),

```

```
m4(n1[4],load,data[4],out[4]),  
m5(n1[5],load,data[5],out[5]),  
m6(n1[6],load,data[6],out[6]),  
m7(n1[7],load,data[7],out[7]);  
endmodule
```

注意：由于该模块 register 将用于以后的实验，请确保该模块仿真的准确。

以下为测试文件 register-test.v

```
*****  
* TEST BENCH FOR 8-BIT REGISTER *  
*****  
'timescale 1 ns / 1 ns  
module register_test;  
  
    wire [7:0] out ;  
    reg   [7:0] data ;  
    reg      load ;  
    reg      rst_ ;  
  
    //自行添加以下代码  
    // Instantiate register 例化寄存器  
    register r1(.out(out),.data(data),.load(load),.clk(clk),.rst_(rst_));  
    // Instantiate clock 例化时钟  
    initial  
    begin  
        clk = 0;  
        forever begin  
            #10 clk = 1'b1;  
            #10 clk = 1'b0;  
        end  
    end  
    // Monitor signals 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  
  
    // Apply stimulus  
  
    initial  
    begin
```

```
// INSERT STIMULUS HERE
/*To prevent clock/data races,ensure that you don't transition the stimulus on the
active(positive)edge of the clock */

@( negedge clk ) // Initialize signals
    rst_ = 0 ;
    data = 0 ;
    load = 0 ;

@( negedge clk ) // Release reset
    rst_ = 1 ;

@( negedge clk ) // Load hex 55
    data = 'h55 ;
    load = 1 ;

@( negedge clk ) // Load hex AA
    data = 'hAA ;
    load = 1 ;

@( negedge clk ) // Disable load but register
    data = 'hCC ;
    load = 0 ;

@( negedge clk ) // Terminate simulation
    $finish ;
end

endmodule
```

编辑完成后，在命令行中键入 vcs -full64 -f run.f -R，应得到结果：

# LAB4 用 always 块实现较复杂的组合逻辑电路

```
`timescale 1ns/100ps
module alu(out, zero, opcode, data, accum) ;
input [7:0] data, accum;
input [2:0] opcode;
output zero;
output [7:0] out;
reg [7:0] out;
//reg zero;

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;
//请自行添加电路的描述部分
always @(accum or data or opcode)
begin
  case(opcode)
    PASS0:  out = #3.5 accum;
    PASS1:  out = #3.5 accum;
    ADD   :  out = #3.5 accum + data;
    AND   :  out = #3.5 accum & data;
    XOR   :  out = #3.5 accum ^ data;
    PASSD:  out = #3.5 data;
    PASS6:  out = #3.5 accum;
    PASS7:  out = #3.5 accum;
    default:out = #3.5 accum;
  endcase
end
assign #1.2 zero = (accum == 0)? 1'b1 : 1'b0;      //时延按照要求设置
endmodule
```

```
*****
* TEST BENCXH FOR ALU *
*****
```

```
'timescale 1 ns / 1 ns
```

```
// Define the delay from stimulus to response check
`define DELAY      20
```

```
module alu_test ;
```

```
    wire [7:0] out    ;
    reg   [7:0] data   ;
    reg   [7:0] accum  ;
    reg   [2:0] opcode ;
    integer     i      ;
```

```
// Define opcodes
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;
```

```
// Instantiate the ALU.
```

```
// Use explicit port mapping.
```

```
alu alu1
(
    .out (out ),
    .zero(zero  ),
    .opcode (opcode ),
    .data(data),
    .accum (accum )
);
```

```
// Monitor signals
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
// Verify response
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
// Apply stimulus
initial
begin
{opcode,accum,data} = {PASS0,8'h00,8'hFF}; #(`DELAY) expect({1'b1,accum      });
{opcode,accum,data} = {PASS0,8'h55,8'hFF}; #(`DELAY) expect({1'b0,accum      });
{opcode,accum,data} = {PASS1,8'h55,8'hFF}; #(`DELAY) expect({1'b0,accum      });
{opcode,accum,data} = {PASS1,8'hCC,8'hFF}; #(`DELAY) expect({1'b0,accum      });
{opcode,accum,data} = {ADD  ,8'h33,8'hAA}; #(`DELAY) expect({1'b0,accum+data});
{opcode,accum,data} = {AND  ,8'h0F,8'h33}; #(`DELAY) expect({1'b0,accum&data});
{opcode,accum,data} = {XOR  ,8'hF0,8'h55}; #(`DELAY) expect({1'b0,accum^data});
{opcode,accum,data} = {PASSD,8'h00,8'hAA}; #(`DELAY) expect({1'b1,      data});
{opcode,accum,data} = {PASSD,8'h00,8'hCC}; #(`DELAY) expect({1'b1,      data});
{opcode,accum,data} = {PASS6,8'hFF,8'hF0}; #(`DELAY) expect({1'b0,accum      });
{opcode,accum,data} = {PASS7,8'hCC,8'h0F}; #(`DELAY) expect({1'b0,accum      });

$display("TEST PASSED");
$finish;
end
endmodule
```

# LAB5 存储器电路的设计

```

*****
* 32X8 MEMORY *
*****


`timescale 1 ns / 1 ns

module mem ( data, addr, read, write ) ;
inout [7:0] data  ;
input [4:0] addr  ;
input      read  ;
input      write ;

reg [7:0] memory [0:31] ;
/*添加代码，使当 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'bZ;
always @ (posedge write)
begin
    memory[addr] = data;
end

endmodule

```

注意：由于该模块 mem.v 将用于以后的实验，请确保该模块仿真的准确。

```

*****
* TEST BENCH FOR MEMORY *
*****


`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;
integer   i      ;


```

```
// Instantiate memory submodule

mem m1 ( .data(data), .addr(addr), .read(read), .write(write) ) ;

// Monitor signals

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

// Define write task

task write_val;
input [4:0] addr ;
input [7:0] data ;
begin
mem_test.addr = addr ;
mem_test.dreg= data ;
#1 write = 1 ;
#1 write = 0 ;
end
endtask

// Define read task

task read_val ;
input [4:0] addr ;
input [7:0] data ;
begin
mem_test.addr = addr ;
mem_test.read = 1 ;
#1 if ( mem_test.data !== data )
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
end
```

```
#1 read = 0 ;
end
endtask

// Apply stimulus

initial
begin
    // INITIALIZE CONTROL SIGNALS
    write = 0 ; read = 0 ;
    //请注意 memory 的工作方式,必须先给存储器写入数据,然后才读出.
    // WRITE DATA = ADDR
    for ( i=0; i<=31; i=i+1 )
        write_val ( i, i );
    // READ DATA = ADDR
    for ( i=0; i<=31; i=i+1 )
        read_val ( i, i );
    $display ( "TEST PASSED" );
    $finish ;
end

endmodule
```

# LAB6 设计时序逻辑时采用阻塞赋值与非阻塞赋值的区别

模块源代码：

```
// ----- 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;
        $display("Blocking: a = %d, b = %d, c = %d.",a,b,c);
    end
endmodule

//----- non_blocking.v -----
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
```

测试模块源代码：

```
//----- compareTop.v -----
```

```
'timescale 1ns/100ps
```

```
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  non_blocking(clk,a,b2,c2);
        blocking      blocking(clk,a,b1,c1);
    endmodule
```

# LAB7 利用有限状态机进行复杂时序逻辑的设计

请按以上要求，完成以下状态控制器：

```
*****  
* CONTROLLER *  
*****  
'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_)
  if(!rst_)
    state<=3'b000;
  else
    state<=nexstate;

//请补充状态转移的代码
always @ (negedge clk) //建议使用格雷码编码，否则综合不过，独热码也可以，但综合效率低
begin
  case(state)
    3'b000 : nexstate <= 3'b001;
    3'b001 : nexstate <= 3'b011 ;
    3'b011 : nexstate <= 3'b010 ;
    3'b010 : nexstate <= 3'b110 ;
    3'b110 : nexstate <= 3'b111 ;
    3'b111 : nexstate <= 3'b101 ;
    3'b101 : nexstate <= 3'b100 ;
    3'b100 : nexstate <= 3'b000 ;
  endcase
end

always @ (opcode or state or zero)
begin:blk
  reg alu_op;
  alu_op = opcode=='ADD||opcode=='AND||opcode=='XOR||opcode=='LDA;
//请补全剩余代码

```

```
case (state)
3'b000:
begin
sel    <= 1'b0;
rd     <= alu_op;
ld_ir <= 1'b0;
inc_pc<= opcode=='SKZ & zero || opcode=='JMP;
halt   <= 1'b0;
ld_pc <= opcode=='JMP;
data_e<= !alu_op;
ld_ac <= alu_op;
wr     <= opcode=='STO;
end

3'b001:
begin
sel    <= 1'b1;
rd     <= 1'b0;
ld_ir <= 1'b0;
inc_pc<= 1'b0;
halt   <= 1'b0;
ld_pc <= 1'b0;
data_e<= 1'b0;
ld_ac <= 1'b0;
wr     <= 1'b0;
end

3'b011:
begin
sel    <= 1'b1;
rd     <= 1'b1;
ld_ir <= 1'b0;
inc_pc<= 1'b0;
halt   <= 1'b0;
ld_pc <= 1'b0;
data_e<= 1'b0;
ld_ac <= 1'b0;
wr     <= 1'b0;
end

3'b010:
begin
sel    <= 1'b1;
rd     <= 1'b1;
ld_ir <= 1'b1;
inc_pc<= 1'b0;
halt   <= 1'b0;
```

```
    ld_pc <= 1'b0;  
    data_e<= 1'b0;  
    ld_ac <= 1'b0;  
    wr      <= 1'b0;  
end
```

3'b110:

```
begin  
sel   <= 1'b1;  
rd    <= 1'b1;  
ld_ir <= 1'b1;  
inc_pc<= 1'b0;  
halt  <= 1'b0;  
ld_pc <= 1'b0;  
data_e<= 1'b0;  
ld_ac <= 1'b0;  
wr    <= 1'b0;
```

end

3'b111:

```
begin  
sel   <= 1'b0;  
rd    <= 1'b0;  
ld_ir <= 1'b0;  
inc_pc<= 1'b1;  
halt  <= opcode=='HLT;  
ld_pc <= 1'b0;  
data_e<= 1'b0;  
ld_ac <= 1'b0;  
wr    <= 1'b0;
```

end

3'b101:

```
begin  
sel   <= 1'b0;  
rd    <= alu_op;  
ld_ir <= 1'b0;  
inc_pc<= 1'b0;  
halt  <= 1'b0;  
ld_pc <= 1'b0;  
data_e<= 1'b0;  
ld_ac <= 1'b0;  
wr    <= 1'b0;
```

end

3'b100:

```
begin  
sel   <= 1'b0;
```

```
    rd      <= alu_op;
    ld_ir <= 1'b0;
    inc_pc<= opcode=='SKZ & zero;
    halt   <= 1'b0;
    ld_pc <= opcode=='JMP;
    data_e<= !alu_op;
    ld_ac <= 1'b0;
    wr     <= 1'b0;
  end
endcase
end
endmodule

/*****************/
* 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 );
$dumpvars(0,control_test);
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",
        $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",
        $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
```

## LAB8：通过模块之间的调用实现自顶向下 CPU 的设计

cpu.v 程序如下：

```
*****  
* CPU *  
*****`timescale 1 ns / 1 ns  
  
module cpu  
(  
    rst_  
);  
  
input rst_;  
  
wire [7:0] data ;  
wire [7:0] alu_out ;  
wire [7:0] ir_out ;  
wire [7:0] ac_out;  
wire [4:0] pc_addr ;  
wire [4:0] ir_addr ;  
wire [4:0] addr ;  
wire [2:0] opcode ;  
  
assign opcode = ir_out[7:5];  
assign ir_addr = ir_out[4:0];  
  
//Instantiate design components  
  
control      ctl1          //例化控制模块  
(  
    .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           //例化 ALU
(
.out (alu_out ),
.zero(zero      ),
.opcode (opcode      ),
.data(data      ),
.accum (ac_out      )
);

register    ac            //例化累加器
(
.out (ac_out      ),
.data(alu_out ),
.load(ld_ac      ),
.clk (clock      ),
.rst_(rst_      )
);

register    ir            //例化 IR 寄存器
(
.out (ir_out      ),
.data(data      ),
.load(ld_ir      ),
.clk (clock      ),
.rst_(rst_      )
);

scale_mux #5 smx           //例化 mux
(
.out (addr      ),
.sel (sel      ),
.b      (pc_addr ),
.a      (ir_addr )
);

mem          mem1           //例化存储器
```

```
(  
    .data(data      ),  
    .addr  (addr      ),  
    .read(rd      ),  
    .write  (wr      )  
);  
  
counter      pc          //例化程序计数器  
(  
    .cnt (pc_addr ),  
    .data(ir_addr ),  
    .load(ld_pc     ),  
    .clk  (inc_pc   ),  
    .rst_ (rst_     )  
);  
  
clkgen      clk          //例化时钟源  
(  
    .clk (clock     )  
);  
  
//Glue logic  
assign data = (data_e) ? alu_out: 8'bz;  
  
endmodule
```