

# 集成电路设计流程

## 参考书目

- 《Verilog HDL设计与验证》人民邮电出版社，EDA先锋工作室 吴继华
- 《Verilog HDL数字设计与综合》电子工业出版社，Samir Palnitkar，夏宇闻等译。
- 《硬件描述语言Verilog》清华大学出版社，Thomas & Moorby，刘明业等译

## VMware虚拟机

- VMWare虚拟机软件可以在一台机器上同时运行两个以上Windows、LINUX操作系统。多启动系统在一个时刻只能运行一个系统，在系统切换时需要重新启动机器。
- VMWare是真正“同时”运行多个操作系统在主系统的平台上，就象标准Windows应用程序那样切换。而且每个操作系统你都可以进行虚拟的分区、配置而不影响真实硬盘的数据，可以通过网卡将几台虚拟机用网卡连接为一个局域网。

## Icarus 仿真工具

- Icarus仿真工具是一个轻量、免费、开源的Verilog编译器，基于C++实现。安装文件中已经包含GTKWave，支持Verilog/VHDL文件的编译和仿真，以命令行方式操作，通过testbench文件可以生成对应的仿真波形数据文件，通过自带的GTKWave可以查看仿真波形图，支持将Verilog转换为VHDL文件。
- 可以检查Verilog文件的语法错误，并进行一些基本的时序仿真。Icarus Verilog显得极其小巧，最新版安装包大小仅有17MB，支持全平台：Windows+Linux+MacOS，并且源代码开源。

## Yosys综合工具

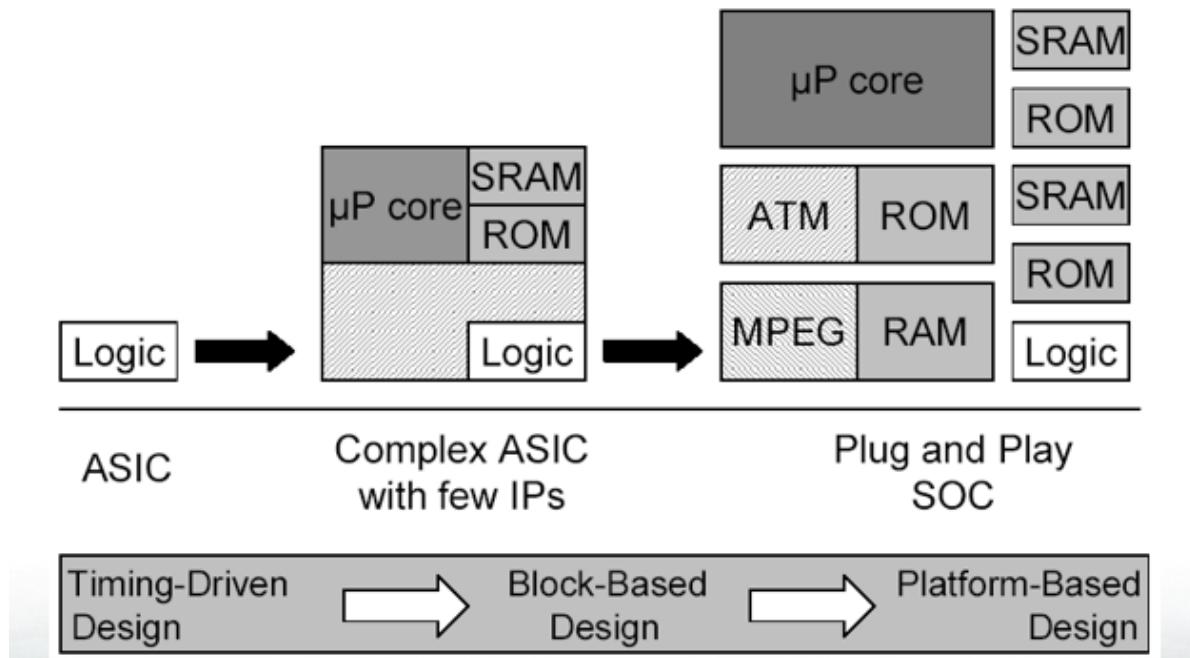
- <http://eduhub.cn>
- <http://www.clifford.at/yosys/about.html>
- 支持 Verilog-2005 的可综合的设计
- 将Verilog转换为其他的设计格式 (BLIF/EDIF/BTOR/ SMT-LIB/ simple RTL Verilog)
- 内建的Formal功能
- 将RTL综合为针对AISC标准单元库的门级网表 (在有ASIC标准单元的Liberty库的前提下)
- 将RTL设计综合映射为Xilinx 7系和Lattice iCE40 FPGA

## IC设计的挑战-技术问题

- 设计复杂度的提高
  - 硬件复杂度
  - 软件复杂度

规模，特征尺寸的缩小，时钟频率提高，低功耗问题，可测试性

IC设计方法的变化



## 集成电路设计的一些基本概念

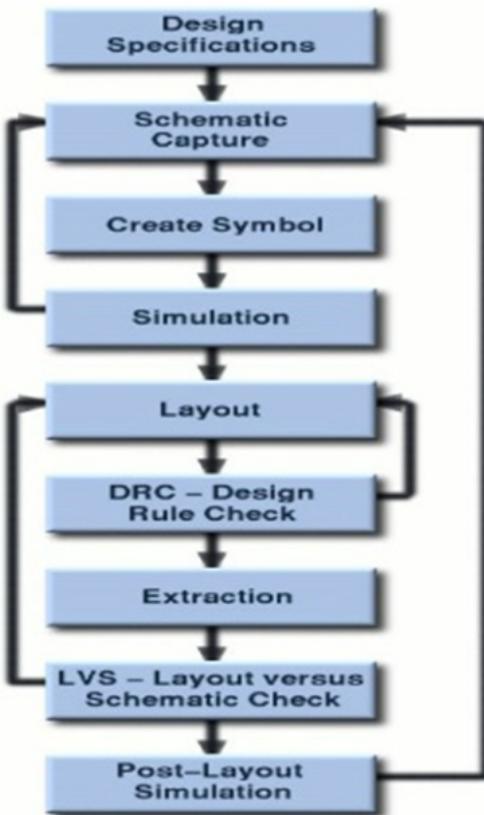
- **nonrecurring-engineering ( NRE )**: 集成电路产品的研制开发费，设计人工费，计算机软硬件设备折旧费以及试制过程中所需的制版、工艺加工、测试分析等研发过程中的一次性开支。
- **wafer size**: 4英寸, 6英寸, 8英寸, 12英寸
- **feature size**: 特征尺寸
- **die size**: 芯片裸粒尺寸
- **Moore's Law**: 加工能力每18个月翻一番
- **等效门**: 一个等效门是一个二输入NAND门
- **RTL**: 寄存器传输级
- **HDL**: 硬件描述语言
- **defect density**: 缺陷密度，影响成品率
- **yield**: 成品率 (良率)

## 集成电路设计方法分类

- 全定制设计
- 基于门阵列的设计
- 基于现场可编程阵列FPGA的设计 (Field Programmable Gate Array )
- 基于标准单元的设计

## 全定制设计(Full-custom)

- 全定制设计：从晶体管开始手工完成集成电路的电路设计、仿真、版图设计的一种方法。
- 设计的精度很高，可以最大程度优化芯片的性能，不会浪费太多芯片资源。
- 花费更多的人力和时间成本。



## 基于门阵列的设计方法

- 门阵列是指由半导体厂商准备出已经在硅片上形成了被称为基本单元的逻辑门的母板,通过按照用户希望的电路进行布线,在母板上形成电路的半客户定制品芯片。
- 门阵列可分为有信道和无信道两种。

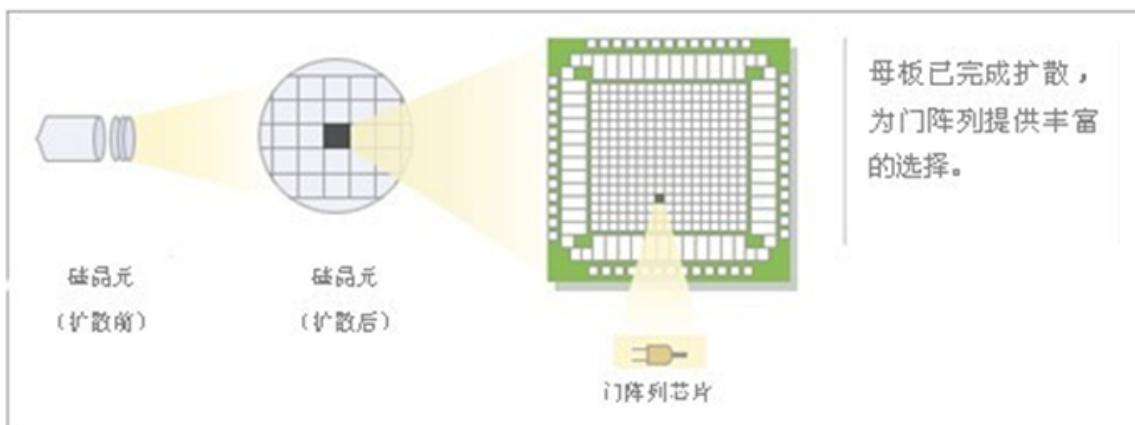


图1 门阵列产品

### 特点：

- 可大幅度的缩短生产工期；
- 低成本：即使是小批量的数字电路，仍能实现低成本，原因在于实现相同的功能，FPGA使用LUT等浪费的资源比较多，而门阵列则不存在这个问题，所以实现相同的功能，门阵列所使用的资源数量要远远小于FPGA，从而减少电路面积，降低总成本。

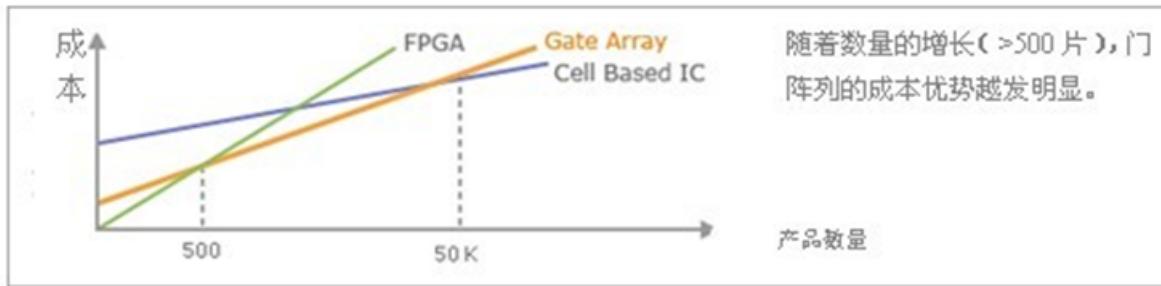


图 2 门阵列与其他产品的成本竞争

- 安全性：FPGA 的电路编程数据存储在 ROM。可通过监控启动时 ROM 和 FPGA 之间的位流，截取和复制电路数据。门阵列的专用电路设计在半定制 IC 上硬连线实现，从而使其不可能被复制。

产品：

- 瑞萨的门阵列产品
- NEC 电子近日推出了一种可将微控制器和门阵列封入到一个封装中的新型 ASIC “PFESiP”

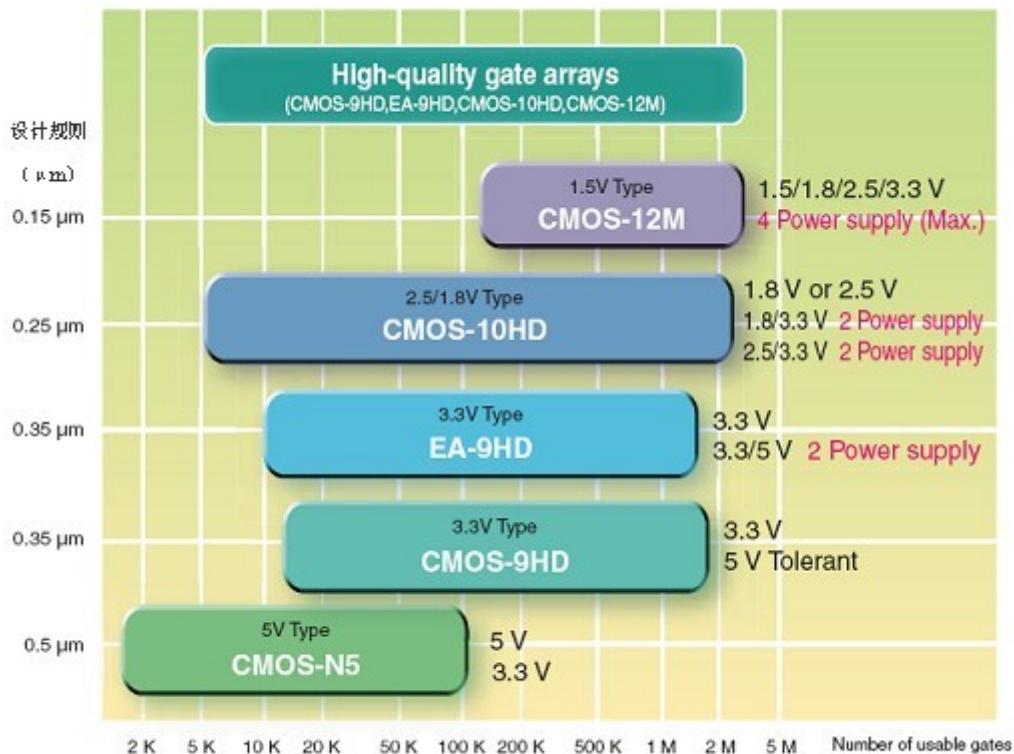


图 3 Renesas 简便门阵列产品阵容

## 现场可编程阵列 FPGA

- 主要特点：

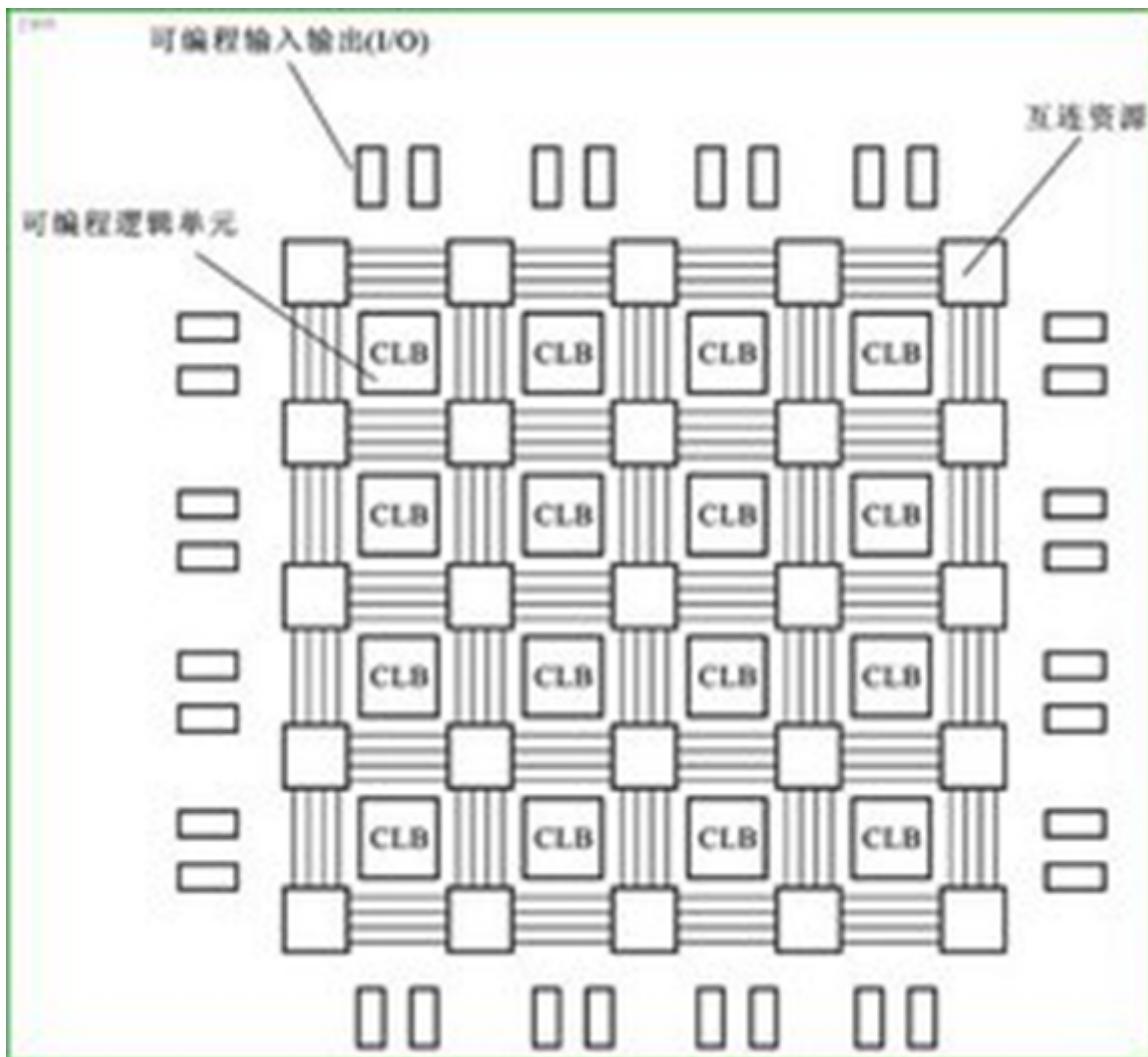
- 小批量系统提高系统集成度、可靠性的最佳选择之一。
- 不需要投片生产，就能得到可用的芯片。
- 可做其它全定制或半定制 ASIC 电路的中试样片。
- 设计周期短、开发费用最低、风险最小

- 主要提供商：

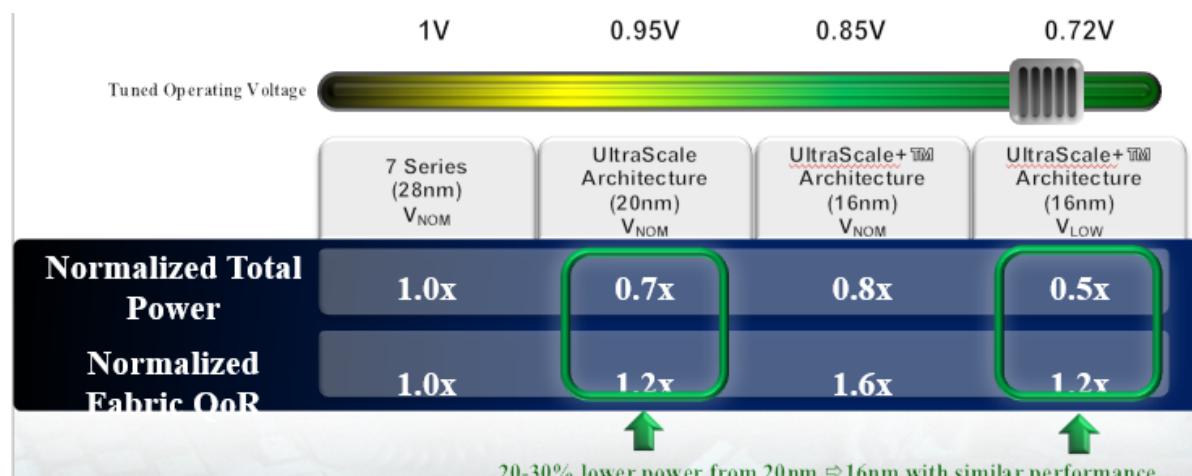
- Altera PLD 的发明者
- Xilinx FPGA 的发明者
- Actel 采用非易失工艺 (反熔丝或 Flash 工艺)

- 结构

- FPGA芯片集成了丰富的门及各种IO
- 存储器SRAM
- 高速存储器接口DDR4
- 时钟 PLL 及DCM
- DSP IP
- XADC
- 高速通信接口PCIe
- CPU
- IP种类丰富

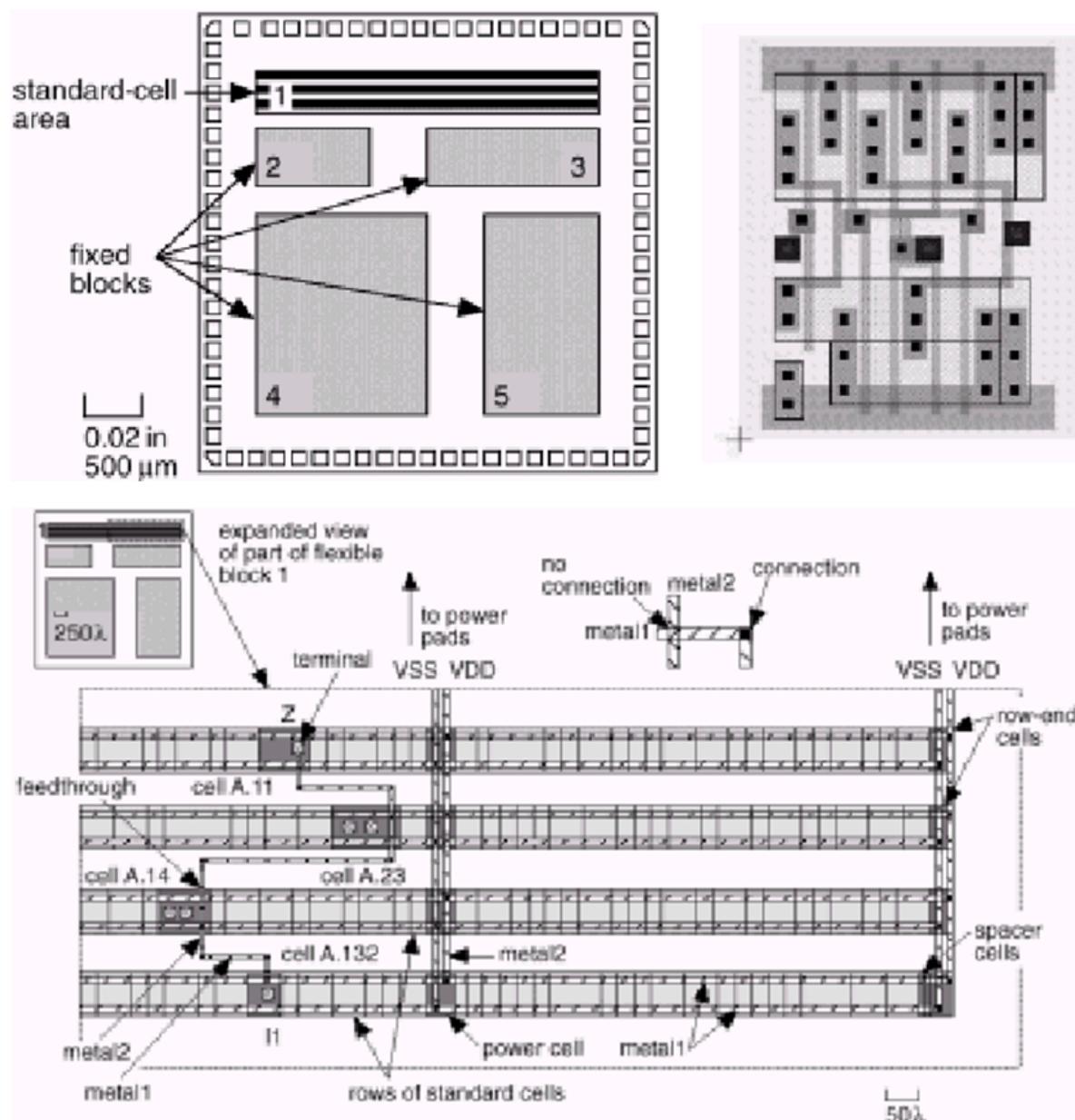


- Xilinx公司成立于1984年，1985年推出第一片FPGA，已经有近30年的历史



## 标准单元库

- 单元库是进行集成电路设计的一个重要部分，可由制造厂家，或第三方提供，也可自己开发。
- 为了支持不同层次的设计，单元库的内容包括：
  - 行为模型 (behavioral model )
  - Verilog/VHDL模型 (verilog/VHDL model )
  - 电路原理图 (circuit schematic )
  - 单元符号 (cell icon, symbol )
  - 详细的时序模型 (detailed timing model )
  - 测试策略 (test strategy)
  - 线负载模型 (wire-load model )
  - 布线模型 (routing model )
  - 物理版图 (physical layout )



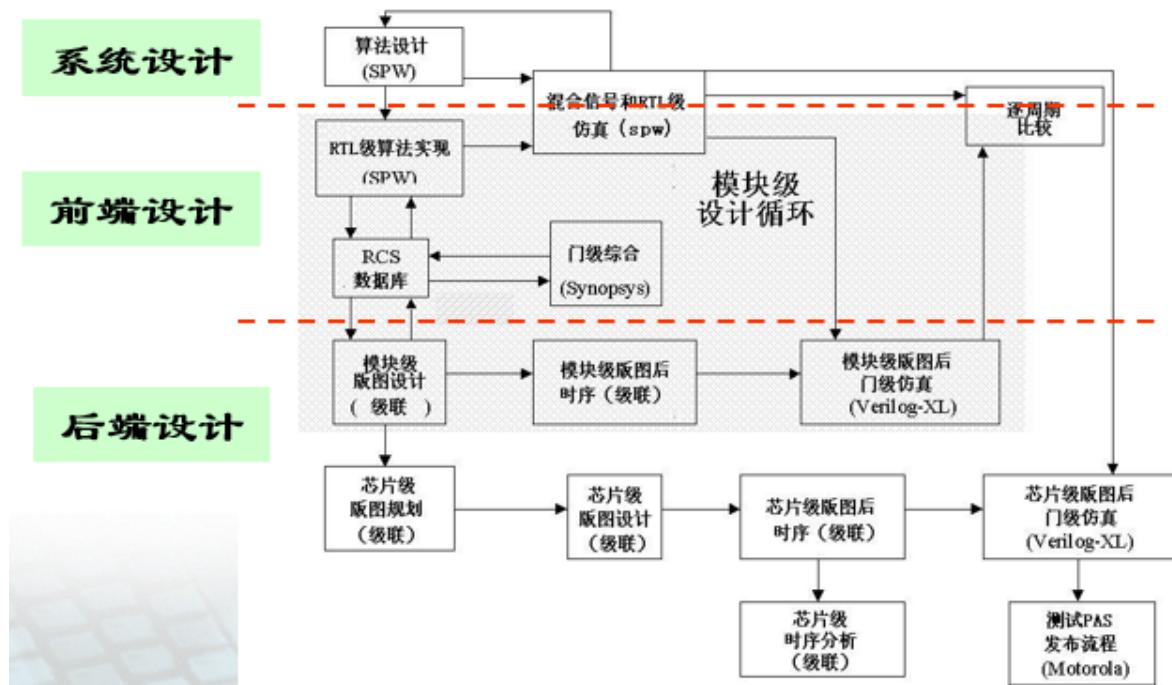
## EDA工具提供商

- Cadence
- Synopsys
  - 电路设计及仿真版图输出及验证、逻辑综合、版图自动设计
- Mentor

- FPGA设计、Modelsim、Calibre、DFT PCB设计
- 华大九天
  - 电路设计及仿真版图输出及验证

## 集成电路设计流程

- IC设计涉及到大量的EDA工具，要求每个工程师对这些工具都有些了解



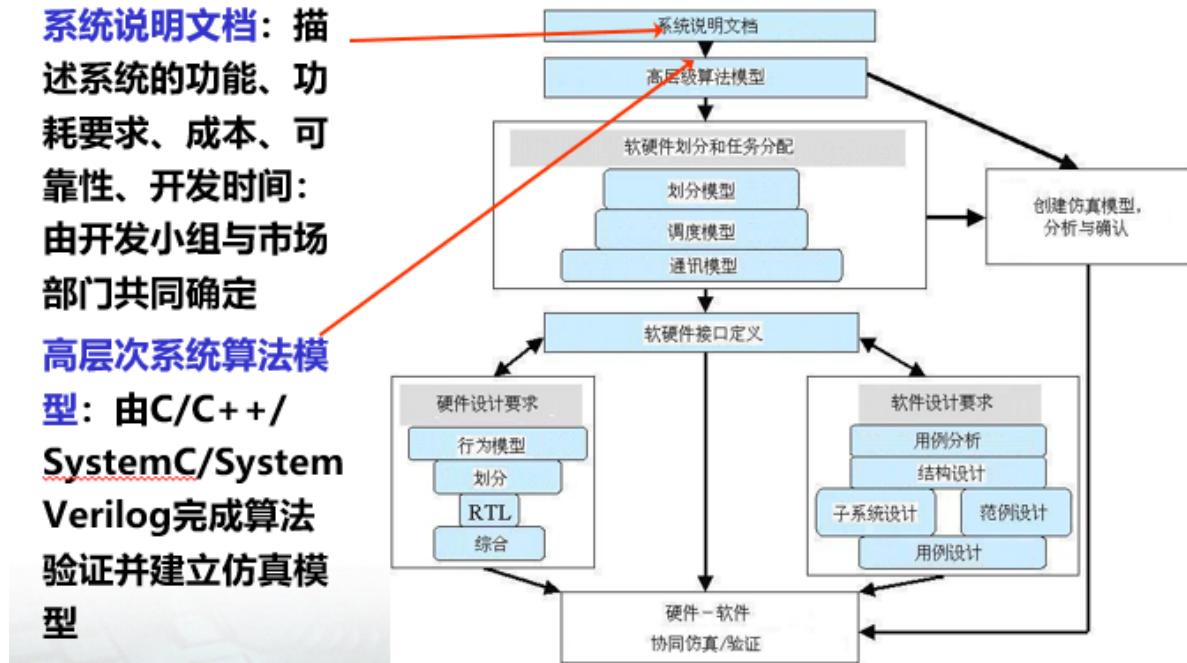
完成一个IC设计，从系统的角度来看，主要分为三个阶段：

- 系统级设计：确定算法，完成行为级模型并对系统功能进行验证，完成系统软硬件划分，确定系统功能框图。由系统工程师完成。要求系统工程师熟悉硬件设计、软件设计、PCB设计等。主要工具：SPW、matlab等。
- 前端设计：根据系统框图和算法，完成RTL设计及其验证。将RTL设计进行综合，并完成综合后验证，得到门级网表。由前端工程师完成。主要工具：schematic editor、Verilog/VHDL simulator、Synopsys Design Compiler、Power Compiler、Prime Time、DFT、Formality等。
- 后端设计：根据网表及综合约束，完成版图设计及验证。由后端工程师完成，主要工具：ultra/SoC Encounter、Prime Time /layout editor、DRC、ERC、LVS的Calibra。

## 软/硬件协同设计技术

**系统说明文档：描述系统的功能、功耗要求、成本、可靠性、开发时间：由开发小组与市场部门共同确定**

**高层次系统算法模型：由C/C++/SystemC/SystemVerilog完成算法验证并建立仿真模型**



软硬件详细设计完成划分后的软件和硬件的设计实现。

硬件综合是在厂家综合库的支持下，完成行为级、RTL以及逻辑级的综合。

代码优化完成对设计实现后的系统进行优化，主要是与处理器相关的优化和与处理器无关的优化。与处理器相关的优化受不同的处理器类型影响很大，一般根据处理器进行代码选择、主要是指令的选择；指令的调度(并行、流水线等)、寄存器的分配策略等；与处理器无关的优化主要有常量优化、变量优化和代换、表达式优化、消除无用变量、控制流优化和循环内优化等。

软硬件协同仿真和验证完成设计好的系统的仿真和验证，保证目标系统的功能实现、满足性能要求和限制条件，从整体上验证整个系统。

软硬件协同设计在实际应用中表现为软硬件协同设计平台的开发。从系统组成的角度，可以用图1来表述软硬件协同设计平台的系统组成。其中设计空间搜索部分由体系结构库、设计库、成本库、系统功能描述和系统设计约束条件组成。设计空间搜索的任务是对不同的目标要求找到恰当的解决办法。体系结构库是存放协同设计支持的各种体系结构数据库，一般是通过不同的模型表现出来。到目前为止，使用较多的模型有状态转换模型(有限状态机)、事件驱动模型、物理结构组成模型、数据流程模型和混合模型等。

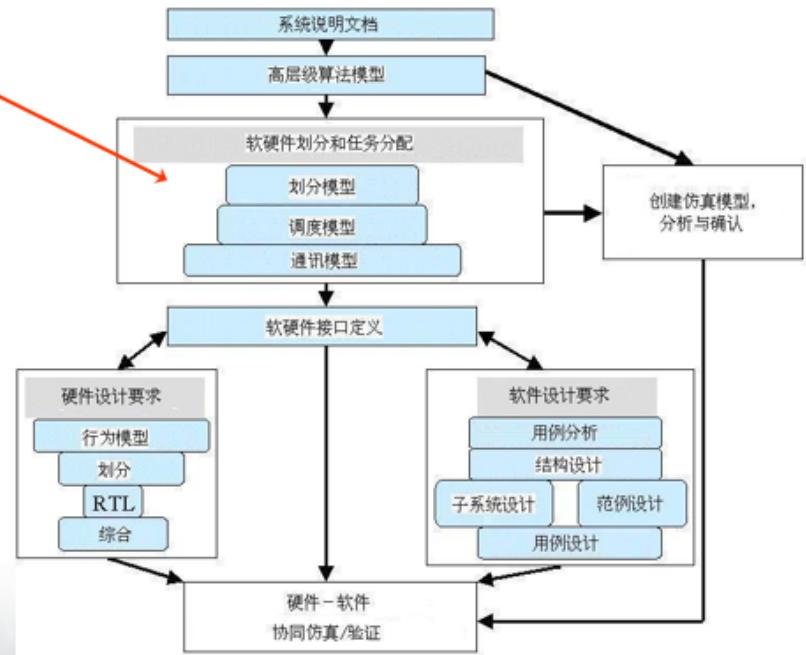
体系结构的丰富程度决定了对目标系统的软硬件协同设计的支持力度。设计库中包含可以使用的程序或网表的设计执行数据库，为新的设计提供参考依据。成本库中提供设计成本的计算方法以及由目标系统的资源消耗、电源消耗、芯片面积、实时要求等组成的数据库，是工作在给定平台上的明确界定。

**主要由有经验的  
工程师完成**

**划分过程就是性  
能与成本的折衷**

— 成本包括什么?

**划分结果是产生  
软件和硬件的详  
细说明文档，并  
定义软件与硬件  
之间的接口**

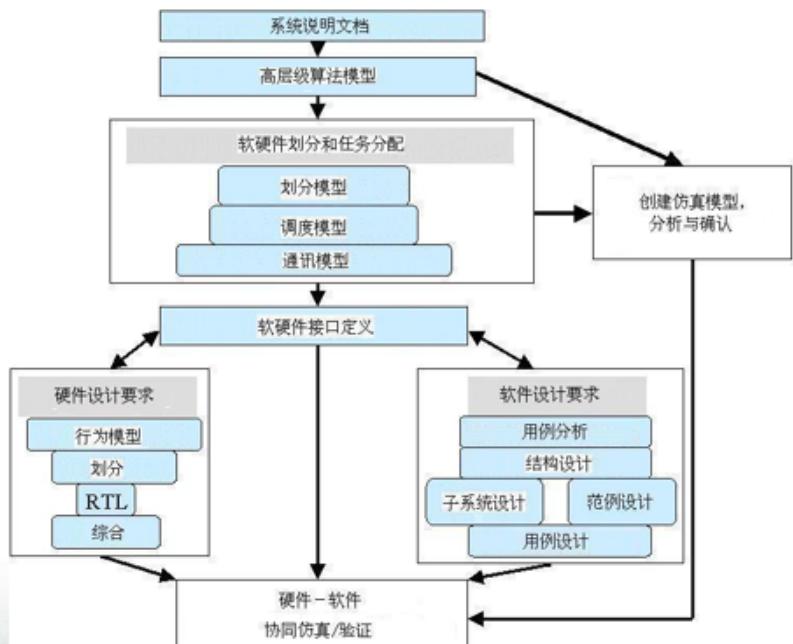


调度模型主要确定软件各个任务子程序之间执行次序。

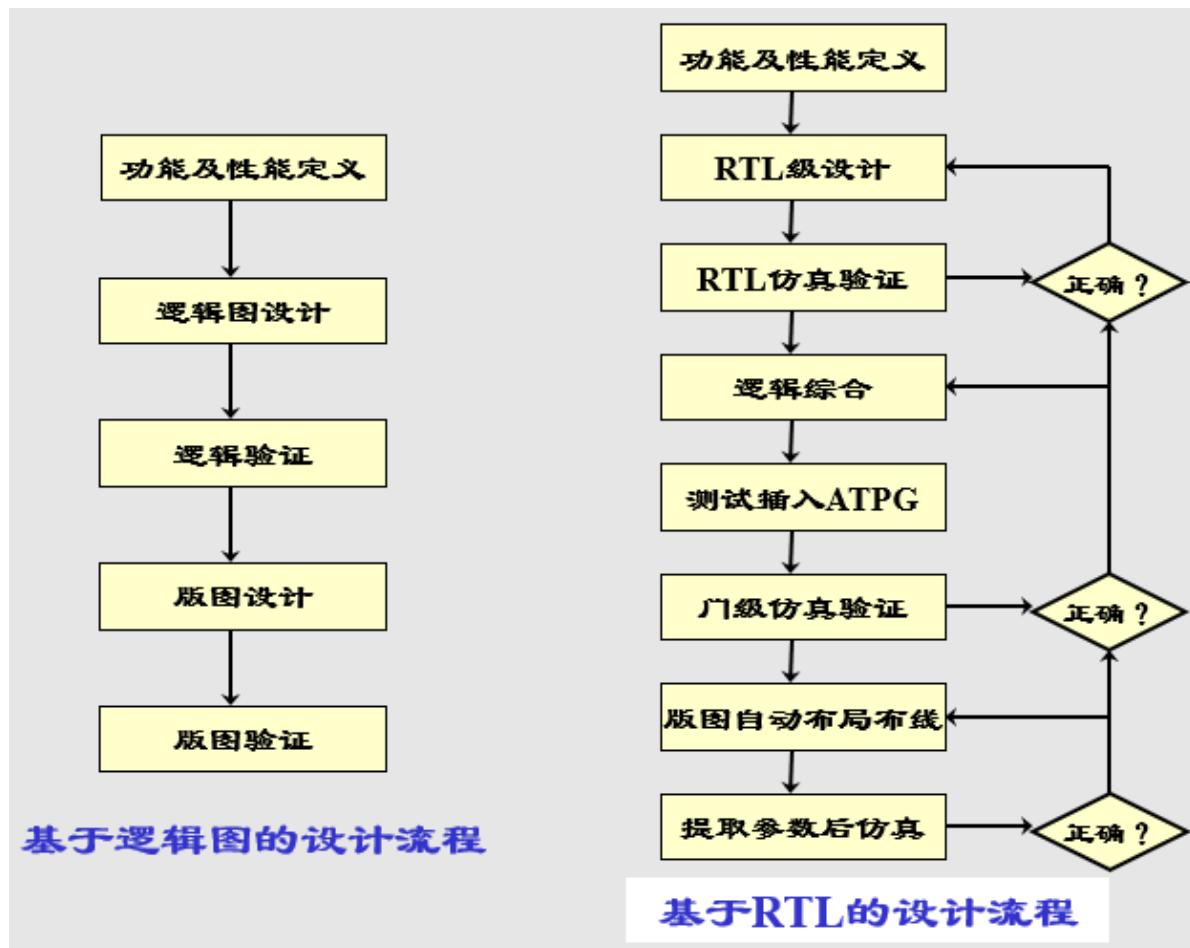
通信模型主要确定硬件之间的通信和软件与硬件之间的通信模式。

**软/硬件划分完成后  
建立硬件行为模型  
和软件工作原型  
通过协同仿真使硬  
件与软件进一步优  
化和精确**

**软件、硬件结构确  
定后，分别进入标  
准的软件开发流程  
和ASIC开发流程**



**基于标准单元的设计流程**



到目前为止，集成电路设计流程有发展有四个阶段在：

- (1) 基于逻辑图的设计流程
- (2) 基于RTL的设计流程，80年代之后
- (3) 深亚微米集成电路设计流程
- (4) 基于IP的设计流程

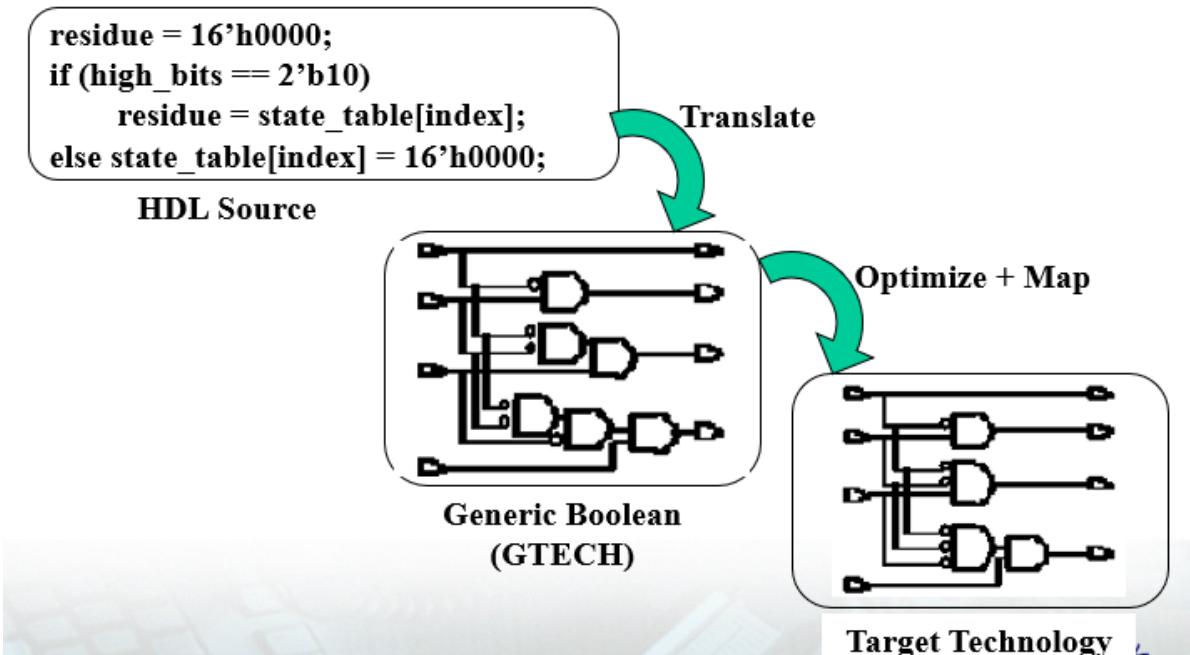
后三个流程大体相似，后两者需要更多的验证。

## RTL设计及验证

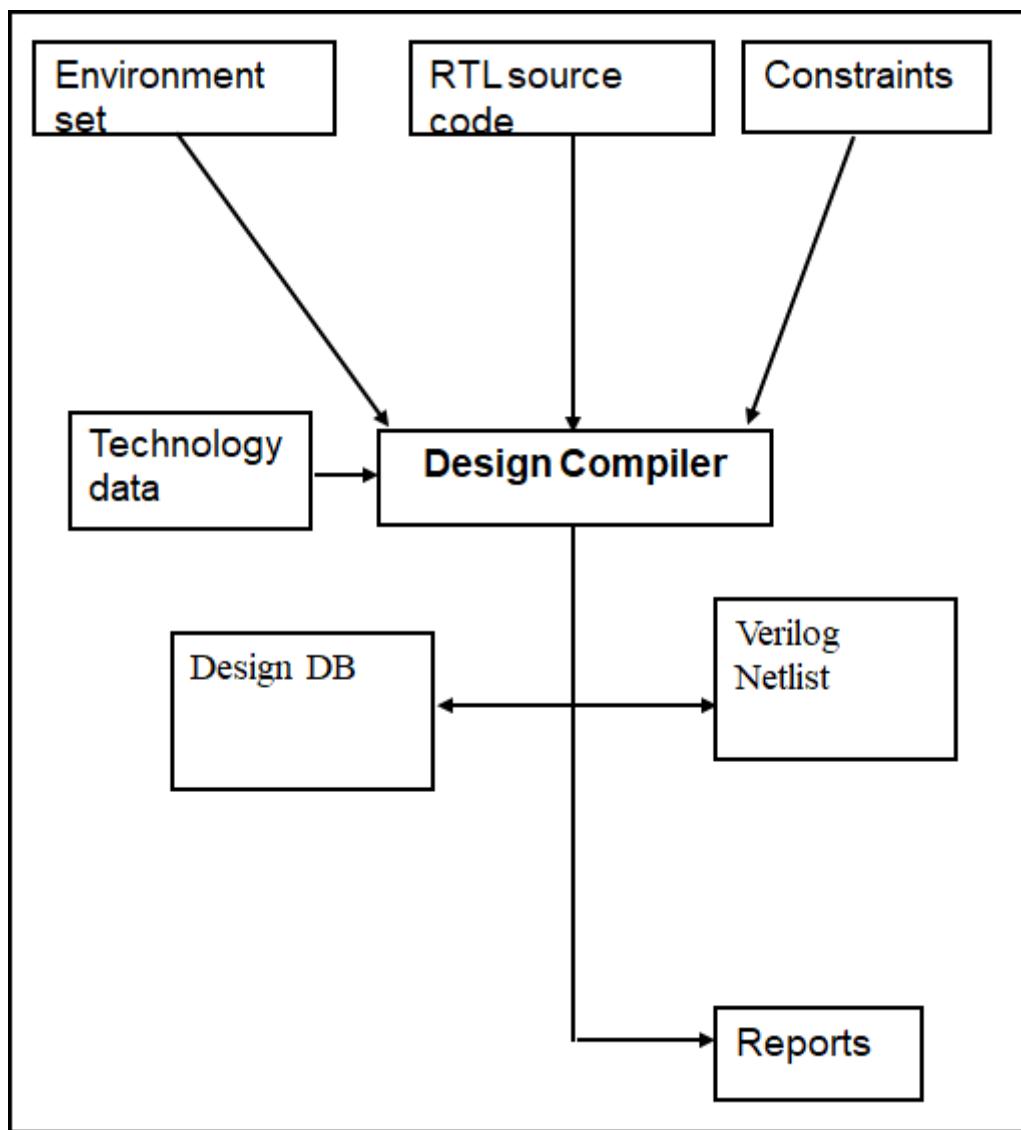
- 采用HDL(Hardware Description Language)完成电路设计
- HDL主要有两种：Verilog和VHDL
- Verilog 1983年由Phil Moorby所创
  - 1995年制定IEEE-1364标准。
  - 2001年推出Verilog-2001标准
  - 2005年推出Verilog-2005标准，即SystemVerilog
- VHDL: IEEE 1706-1985标准。

什么是逻辑综合？

Synthesis = translation + optimization + mapping



## 逻辑综合-Synthesis

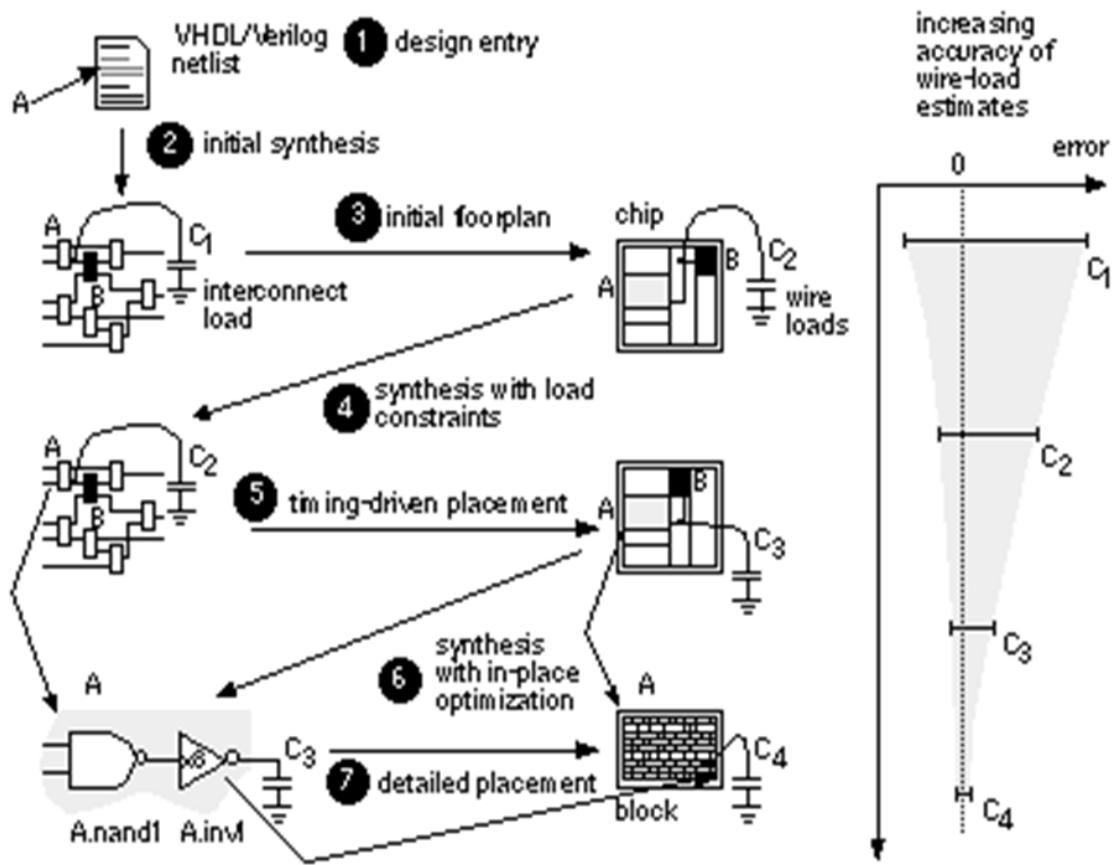


- RTL代码 + 设计约束 + 设计库
- 提交综合工具
- 综合工具主要: synopsys: design compiler/physical compiler

cadence: build gates/RTL compiler

magma: blaster create

## 物理综合



对于深亚微米 (0.18um以下) 设计，设计过程不同。在之前，延时主要在门上。连接线上的延时主要是按照wireload模型进行估算。

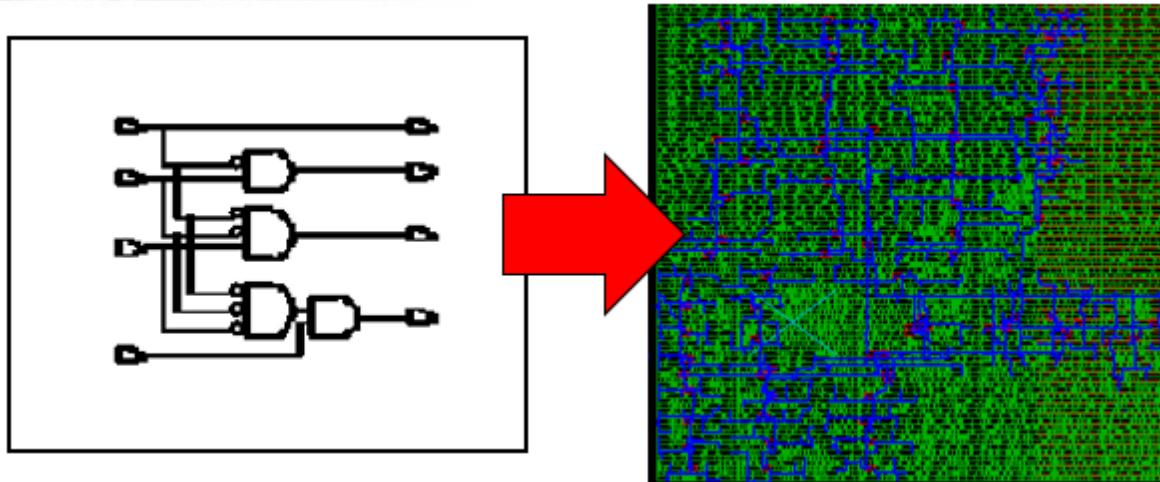
到0.18um,特别是0.13um以下，连接线延时占70%以上，而工作时钟频率往往也比较高，在400MHz以上。因此必须在综合时能够比较精确估算延时。

采用物理综合流程进行设计。

首先是进行预布局，得到门的位置后，再估算连接延时。这些延时反标注后，再进行电路综合。

预布局的过程是：面积估算、IP自动布局、手工调整。门逻辑预布局。这个时候，可以对电路性能进行初步估算。如果达不到性能要求→修改设计，因为此时往往面积估得会小一些。

## 版图设计 (layout)

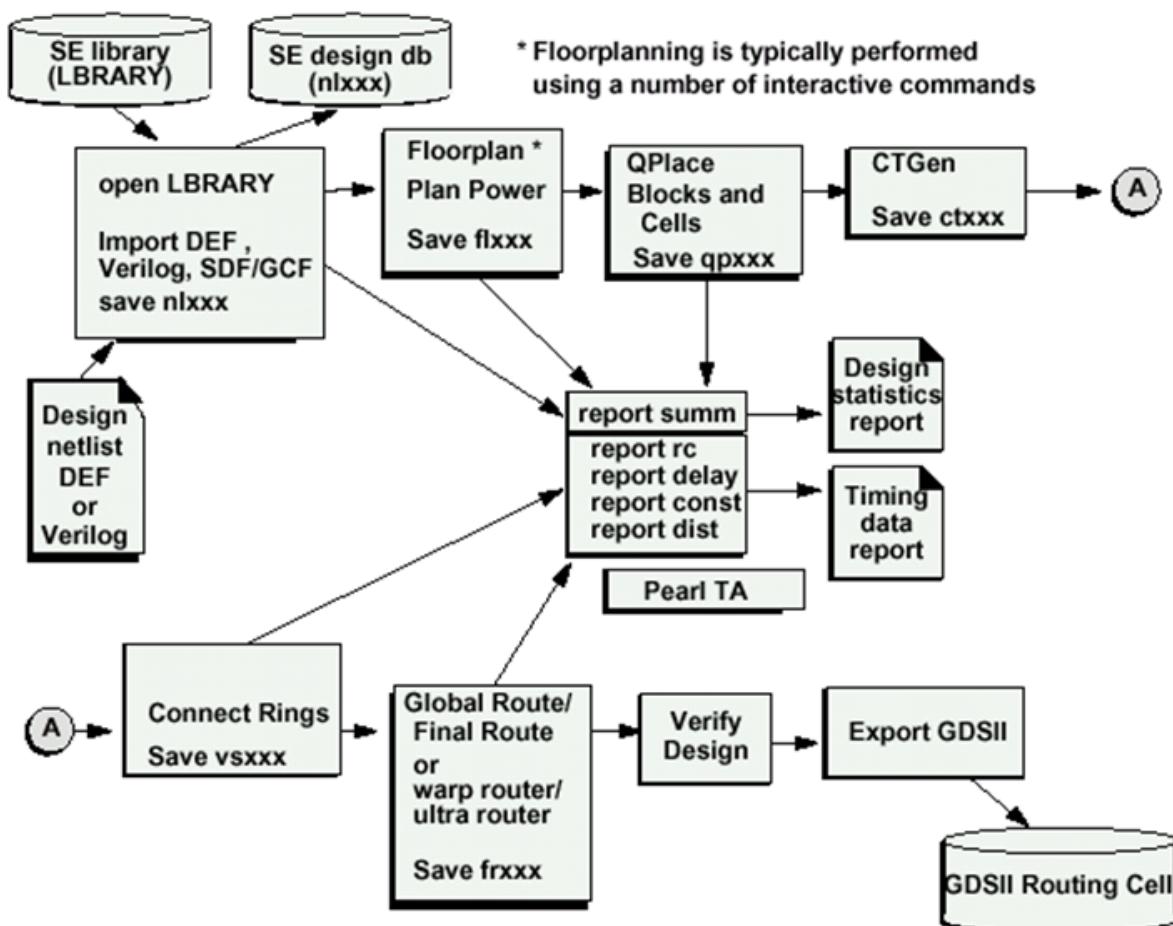


后端设计工具很多，synopsys:applo/ultra、cadence:SE/SocENcounter, magma/blaster fusion, 都是很好的工具。没有优劣。差别再于细节处理不同。

Magma简单易用，内部有一个设计流程，并把设计流程中许多细节包括在一个大的命令（实际是TCL）中。对于专家，可以修改这些命令。

工艺支持最好的是synopsys的ultra。有人抱怨SE不好用，没有undo。

## 后端Layout设计



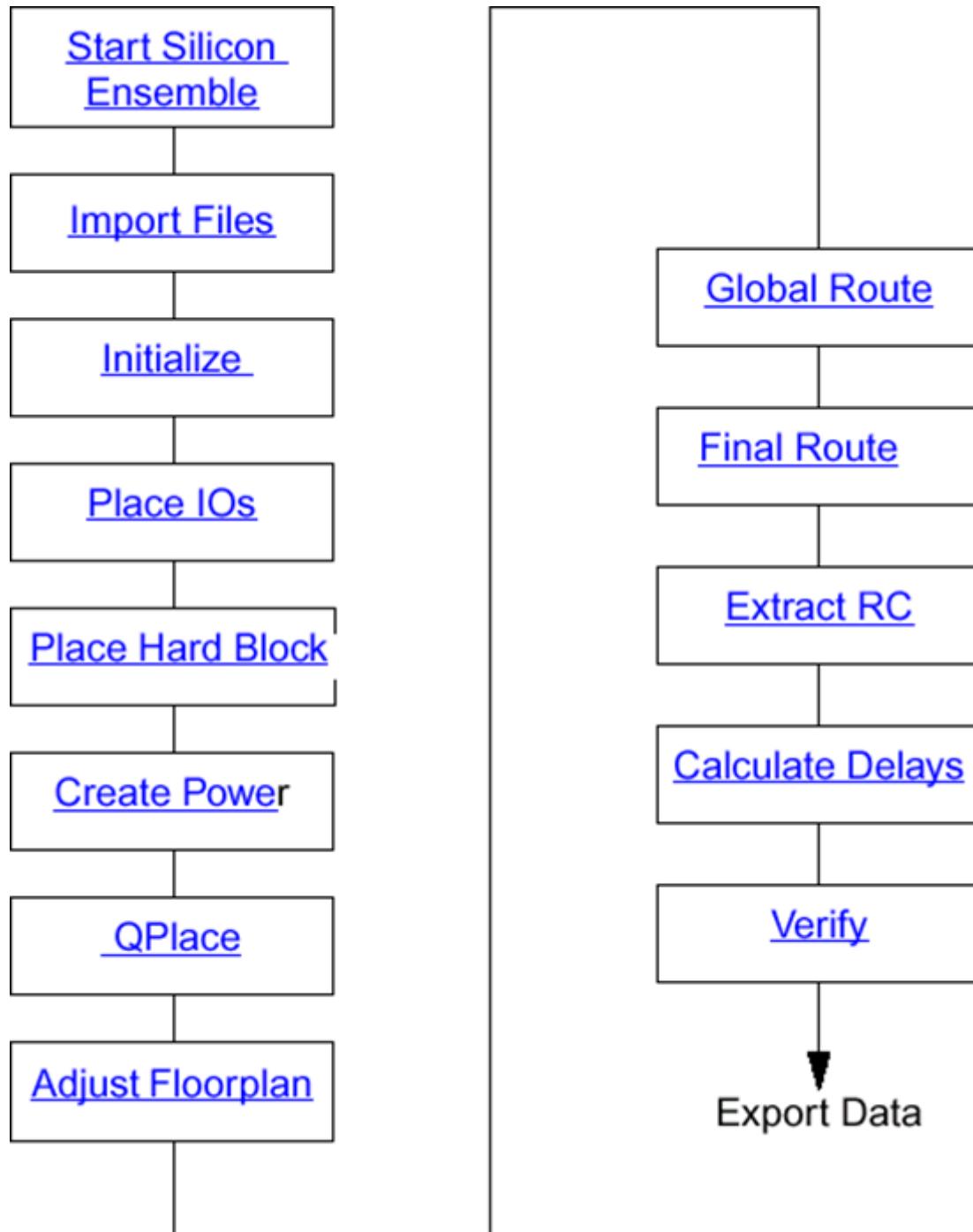
后端设计基本上包括以下几个步骤：

- 设计准备
- 布局
- 时钟产生
- 布线
- 分析

- 验证

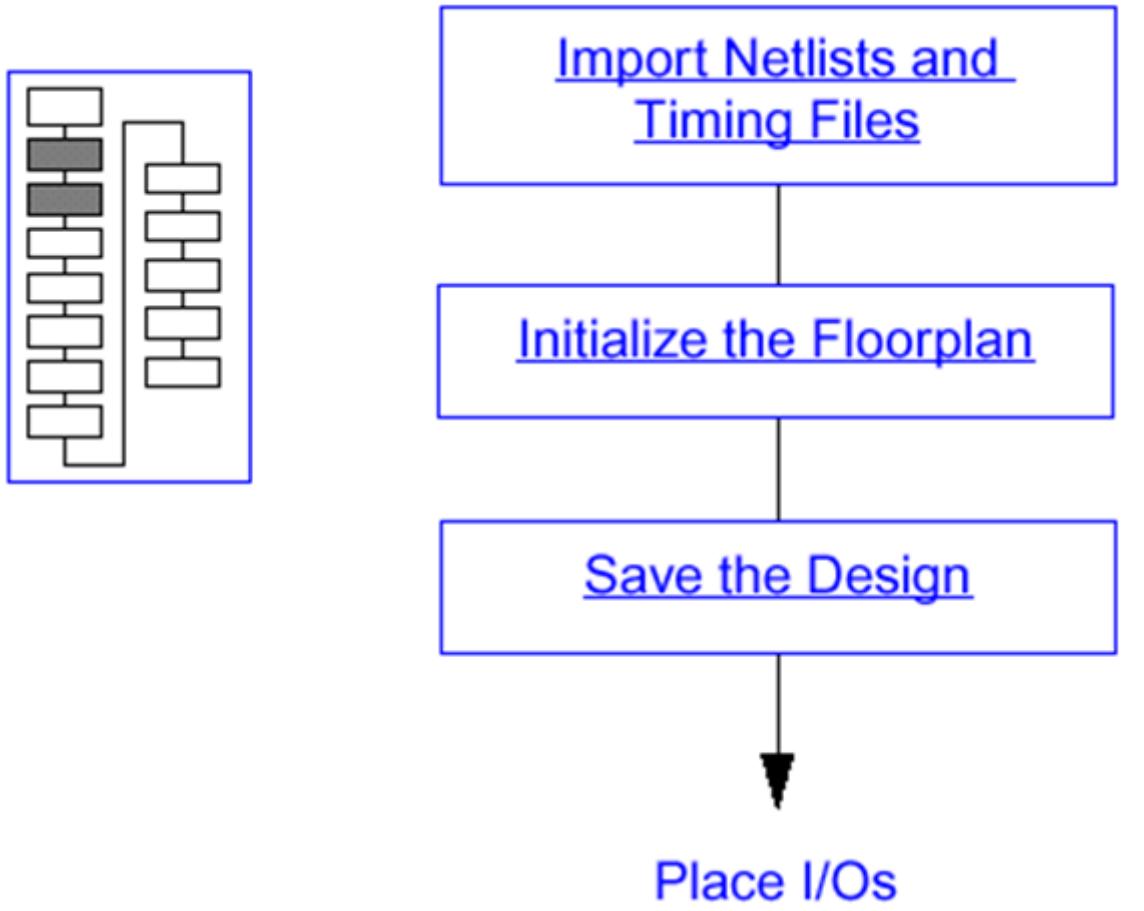
## 设计流程

版图设计是一个基于时序的 floorplan, place、route工具



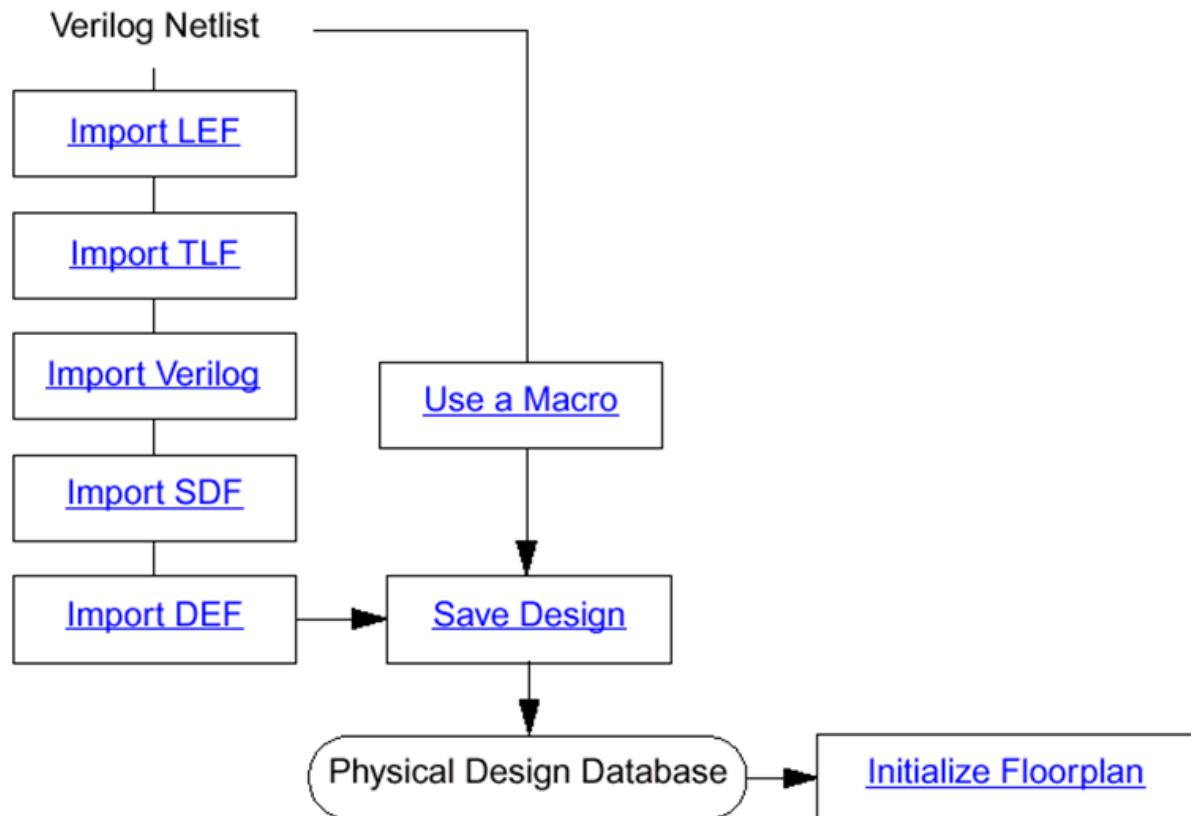
## 设计准备

- 输入数据并对设计进行初始化



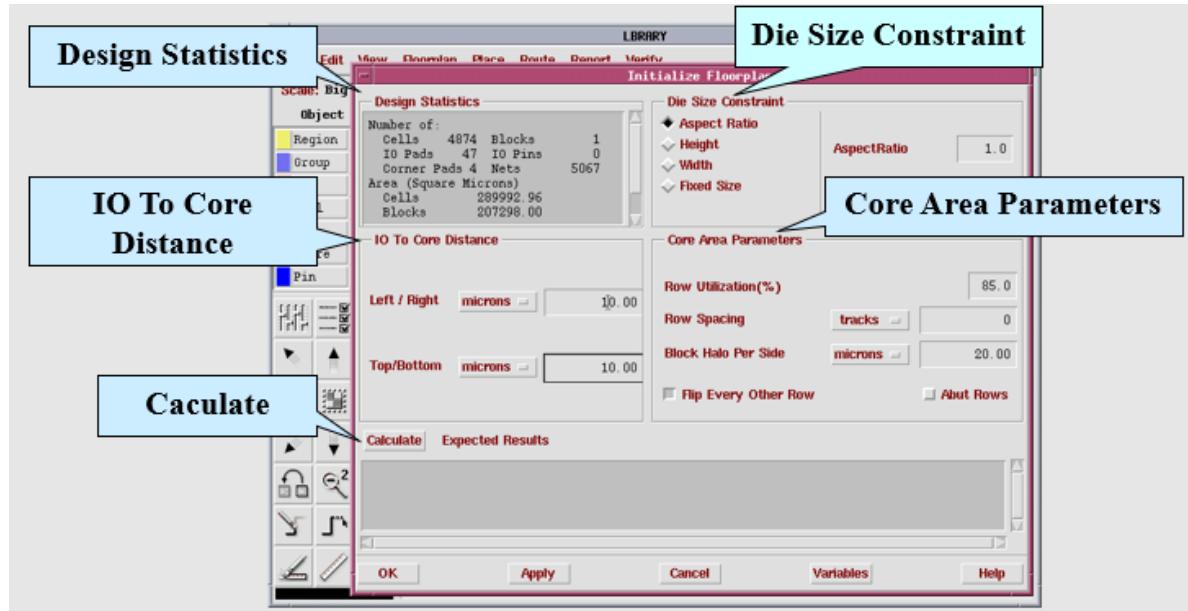
## 导入网表及时序文件

- 建立物理数据库：导入库，网表，以及时序信息

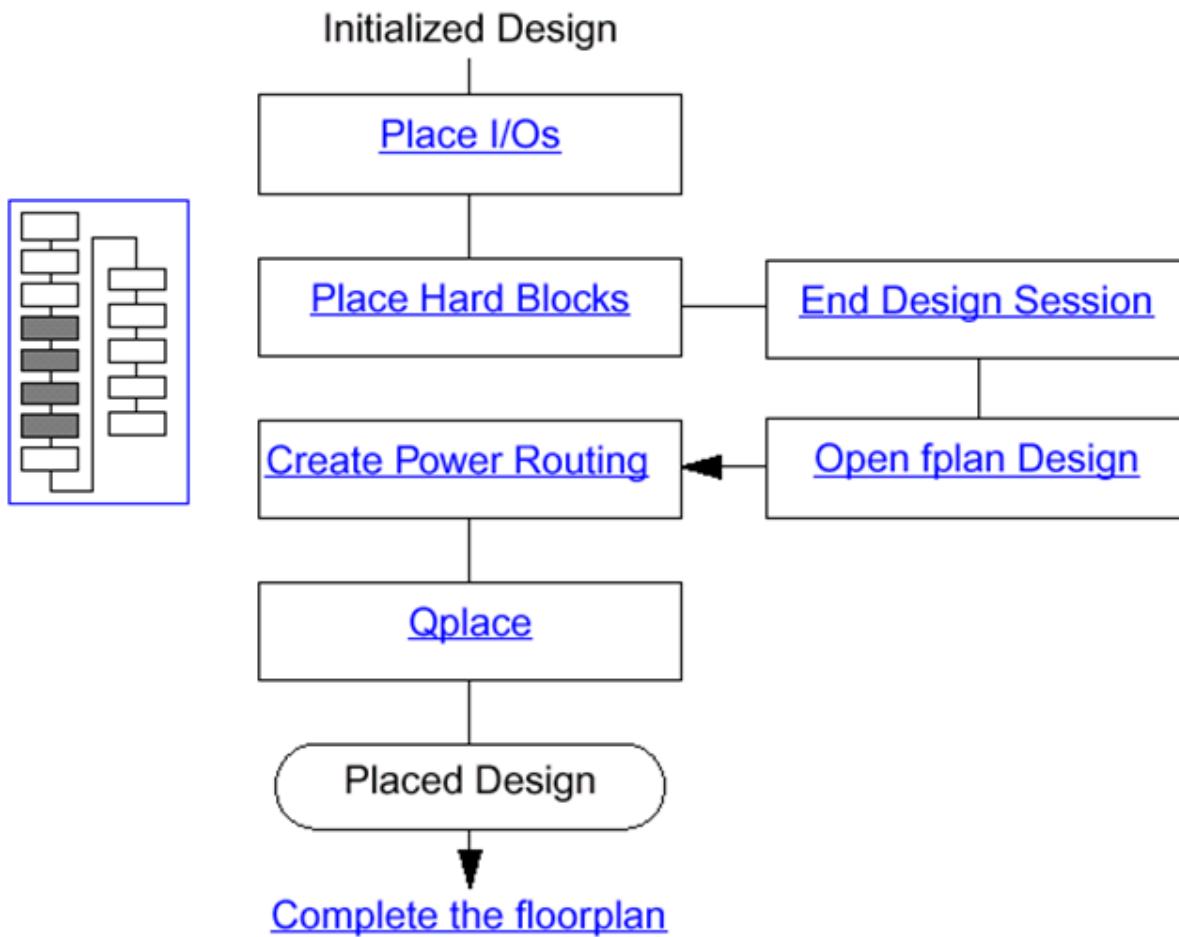


## Floorplan

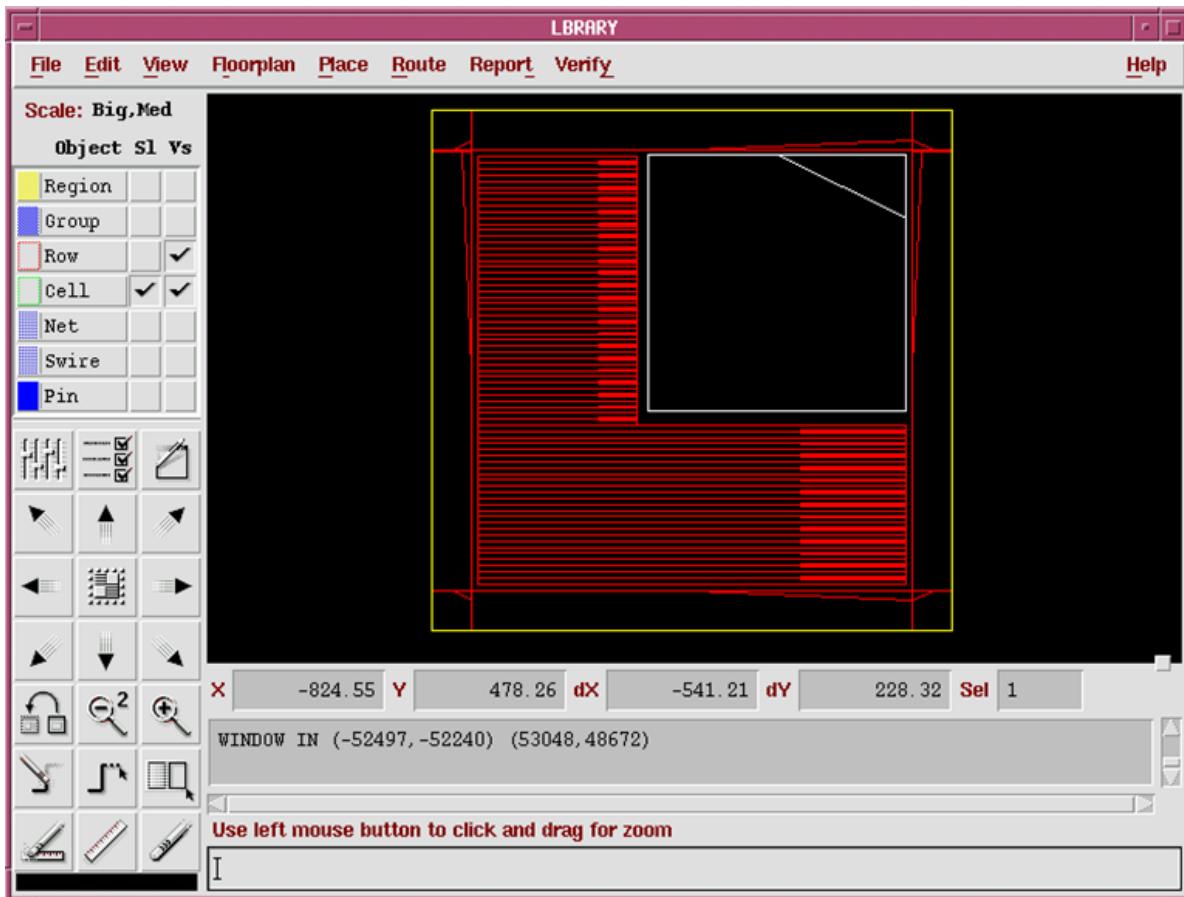
- 不放置任何单元，只控制芯片的芯片的大小、长宽比率，建立横向或纵向的core和I/O row



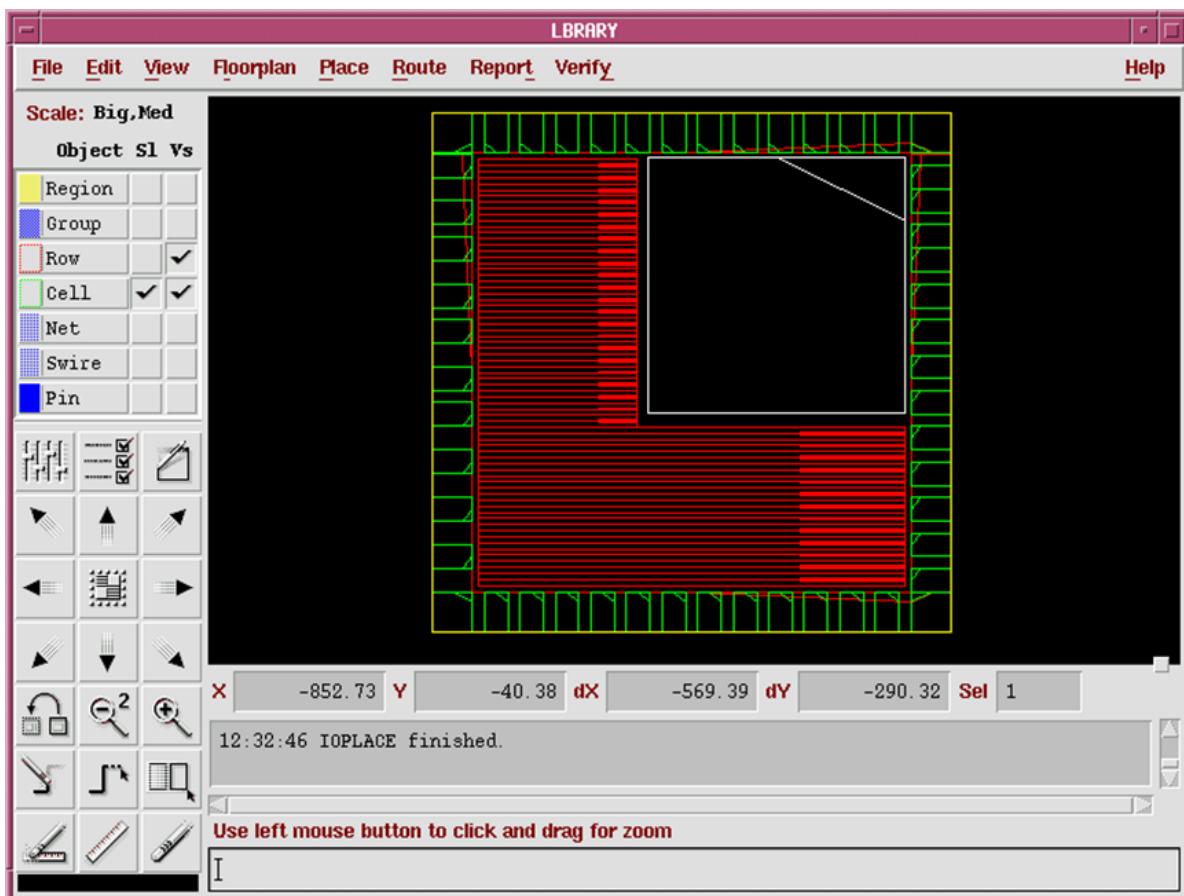
## Building the Floorplan



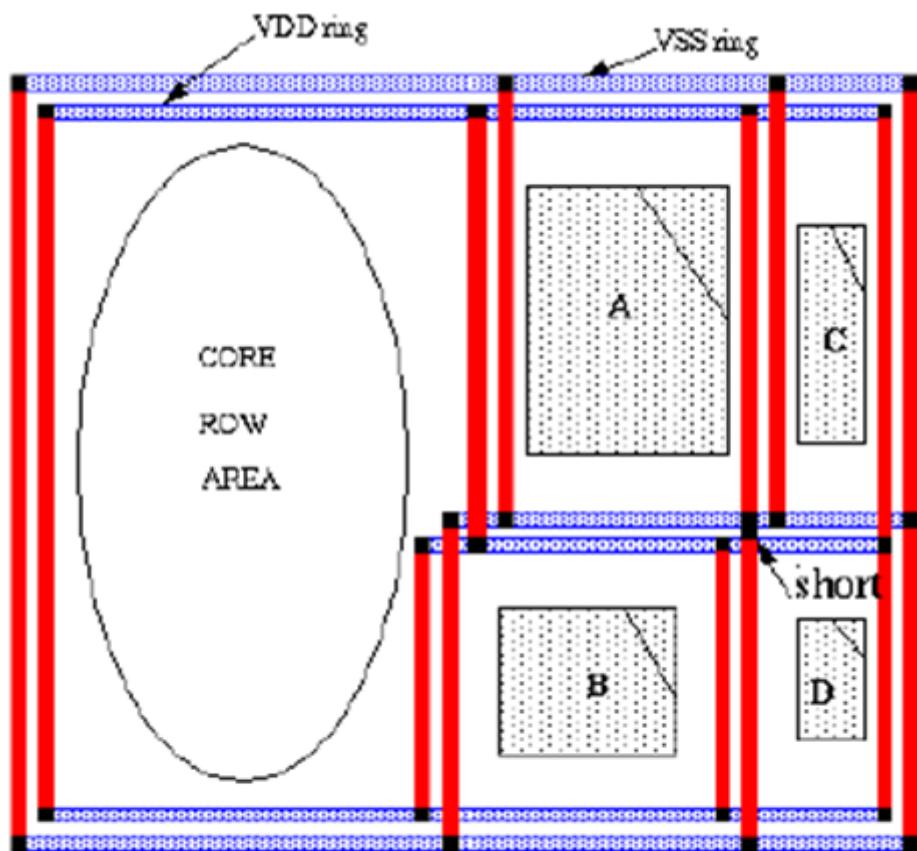
## Placing Block



## I/O Placed

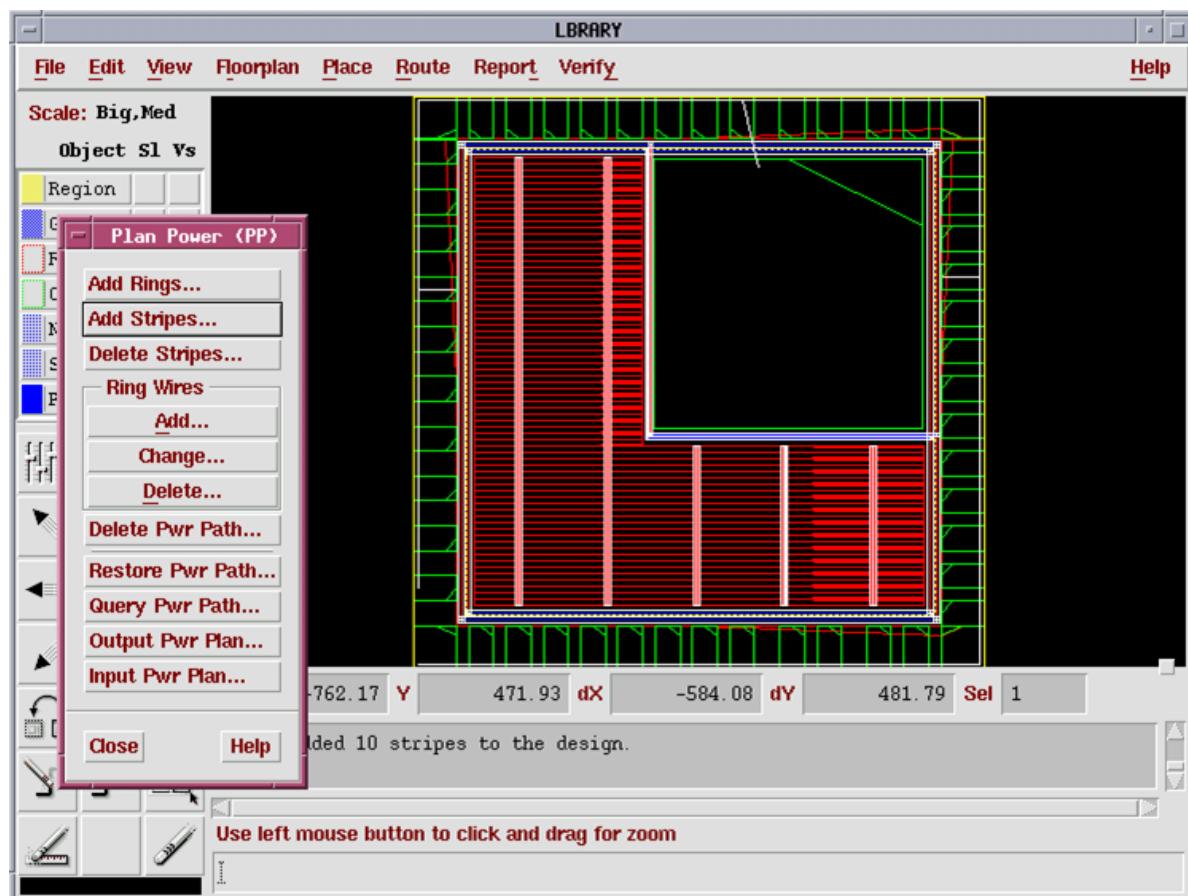


## Power Planning

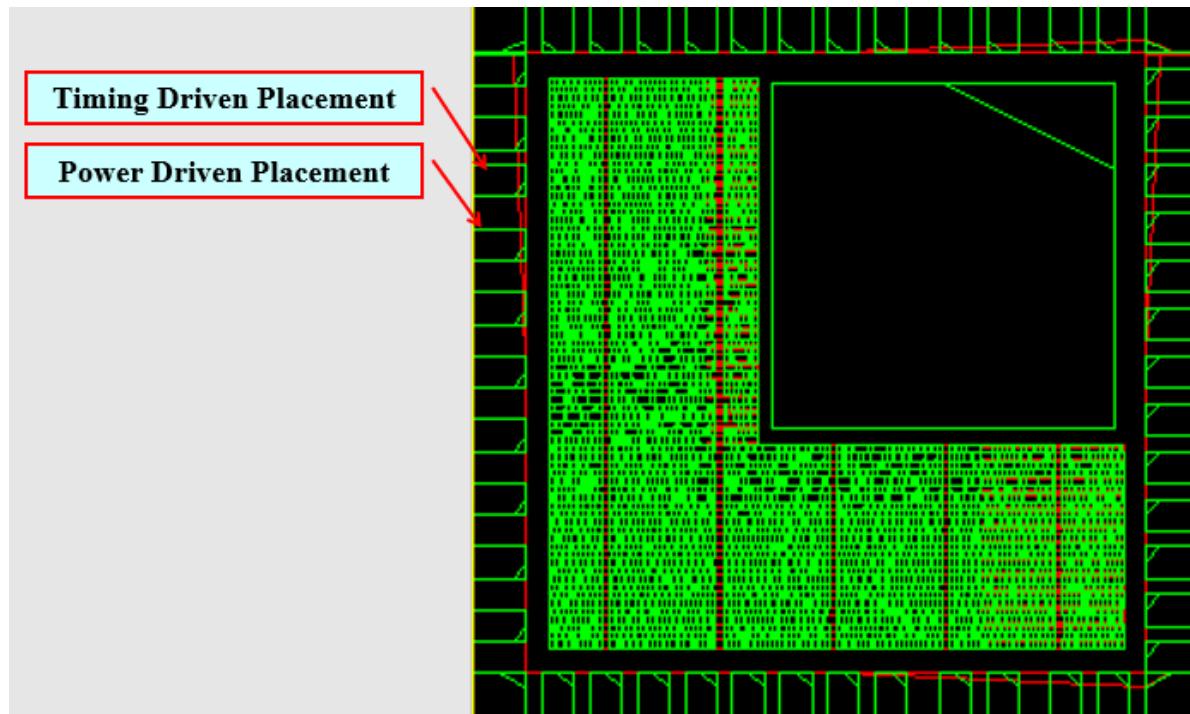


Metal1  
Metal2  
Vt

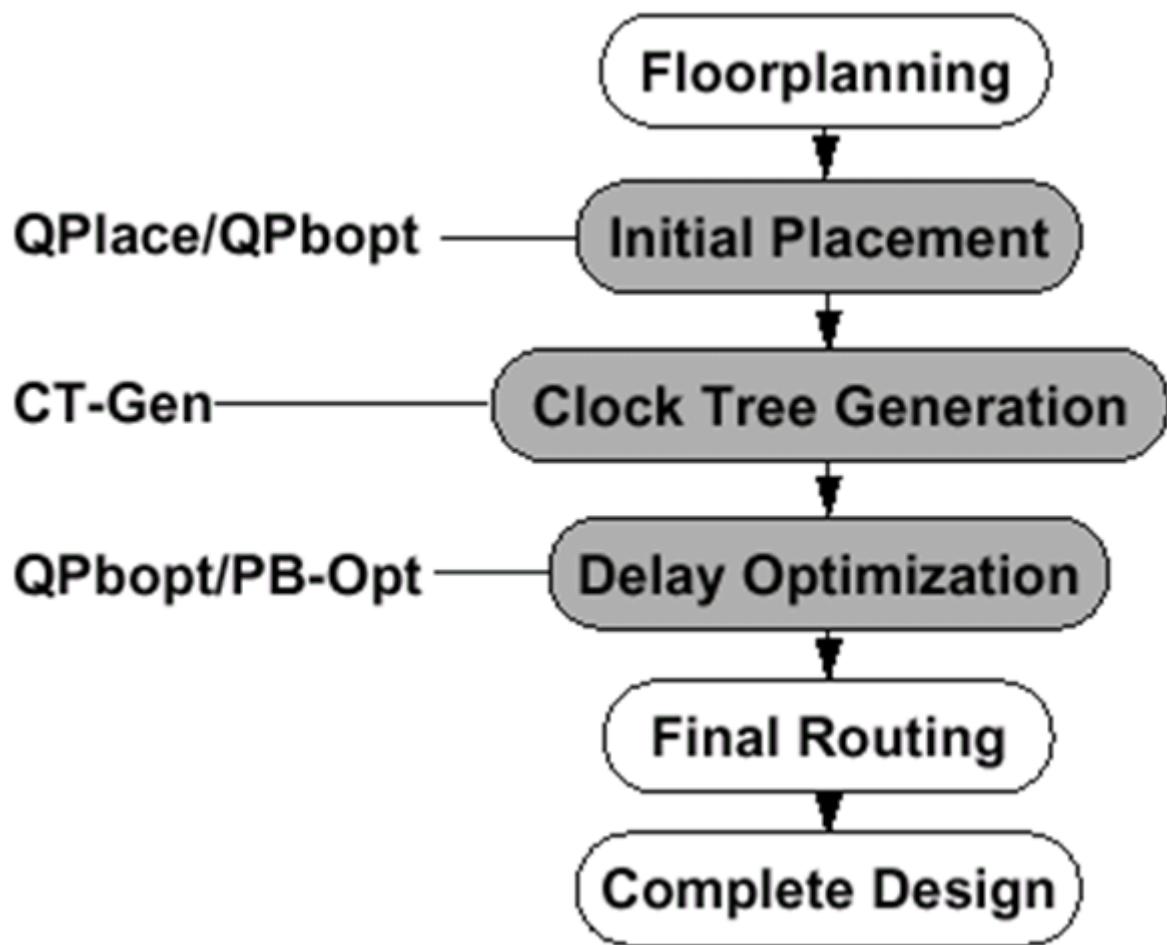
## Plan Power – Strips



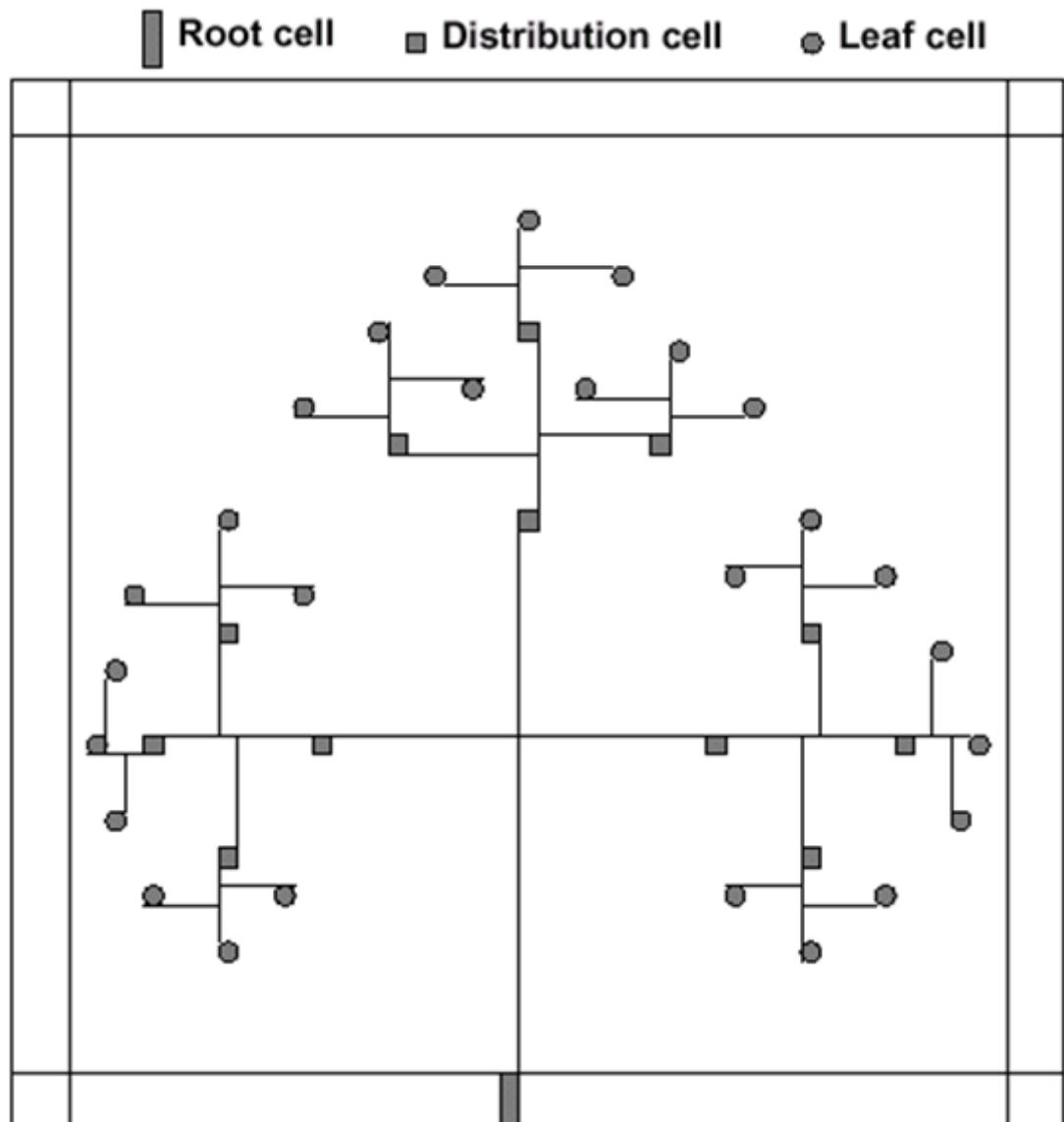
## Place Cells



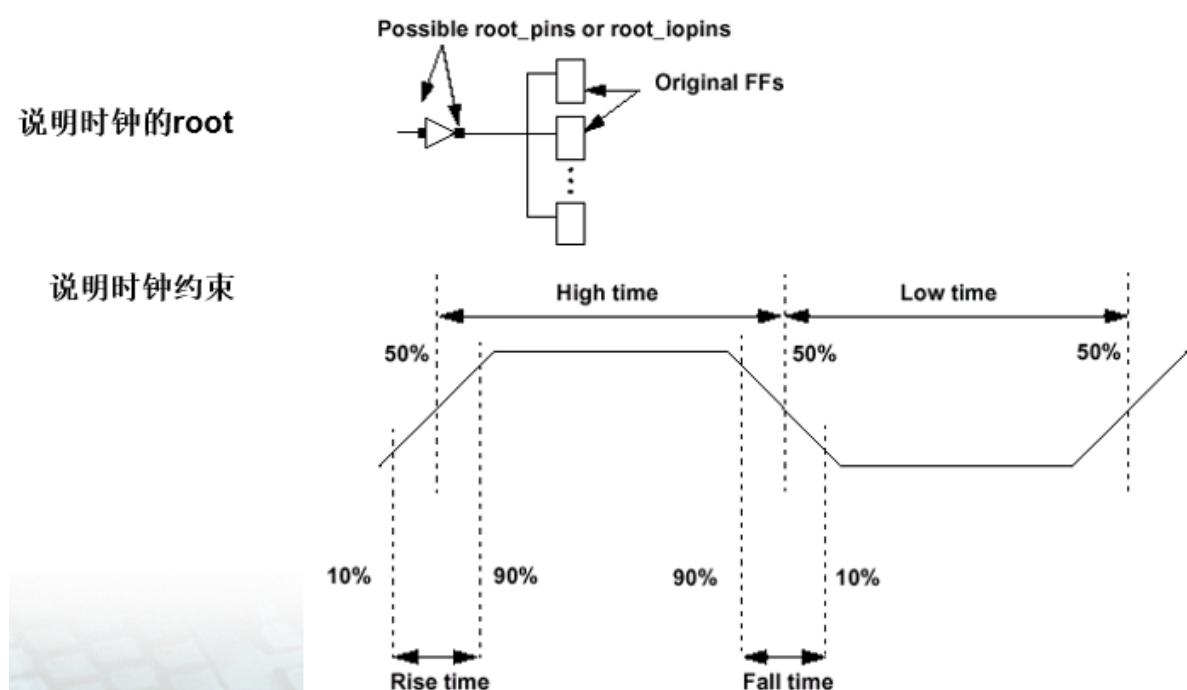
## 时钟树设计流程



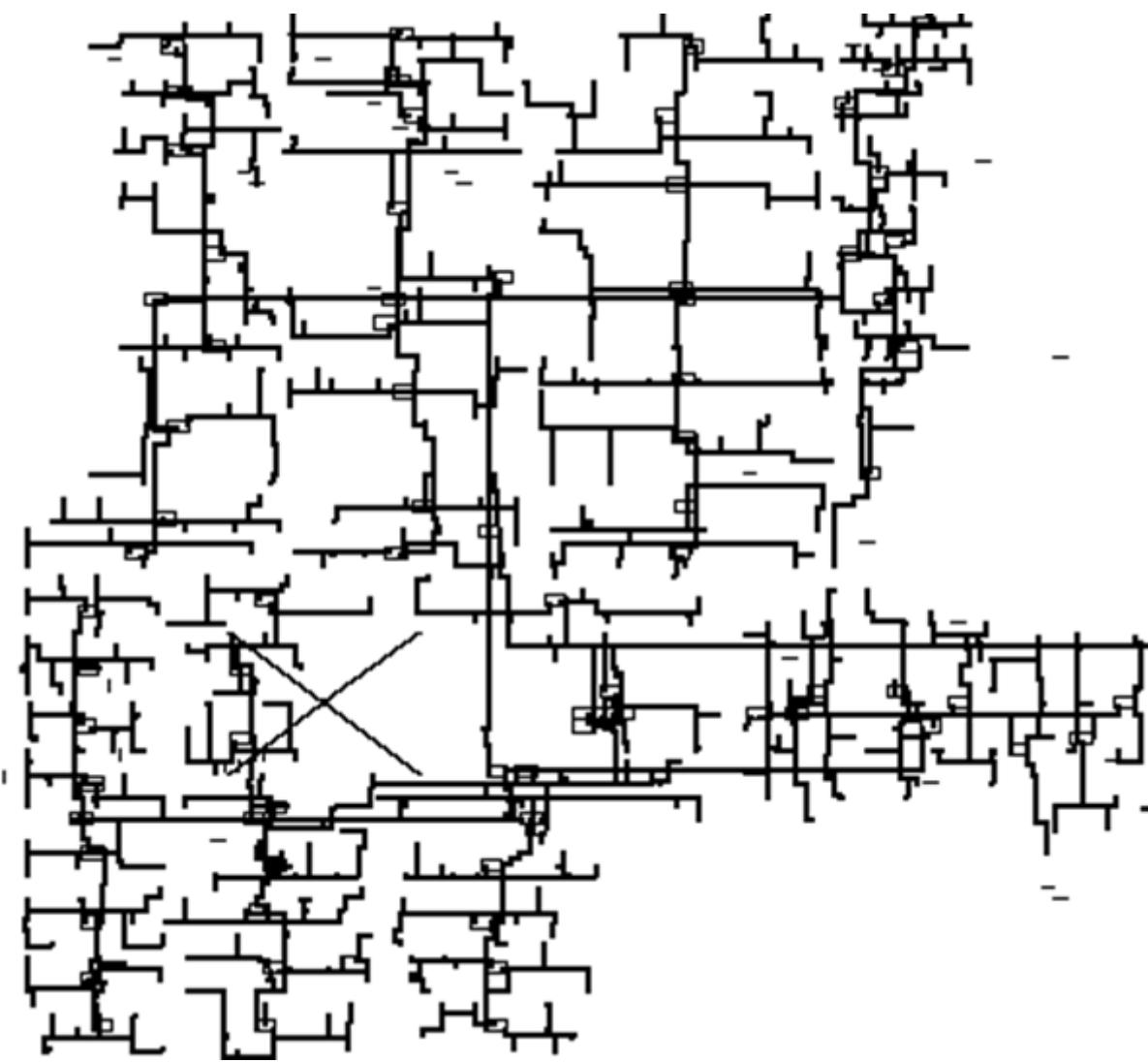
## 时钟树设计



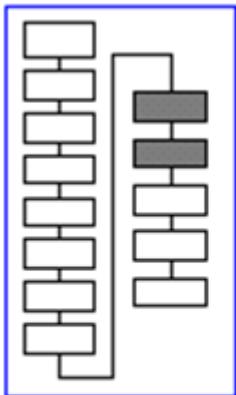
## Using the CT- Gen Integration in SE



## Setting the Clock Tree Constraints



Routing flow



Completed floorplan

Connect Special Nets

Global Route

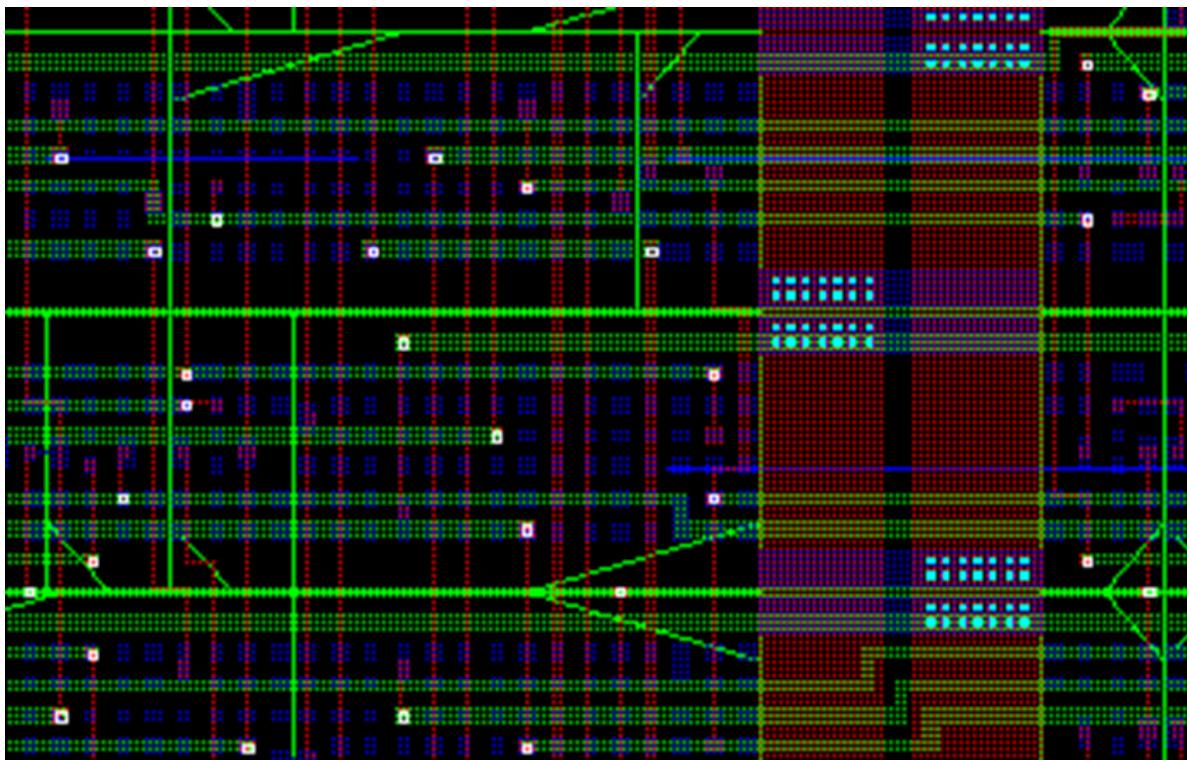
Final Route

Fix Violations

Routed Design

Post-Routing Checks

## Routed Design

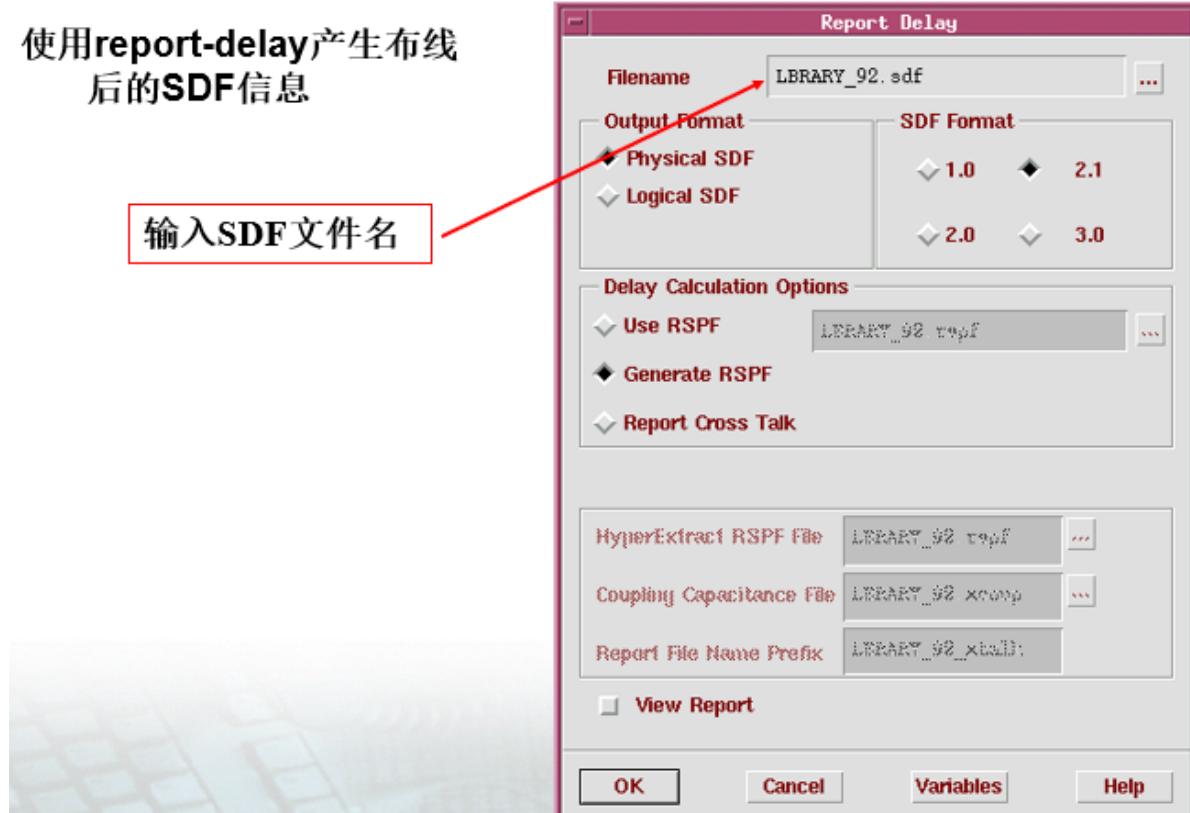


## Verify

完成布线后，如果有violation，final router将在数据库中保留信息标记。这时，可以用Verify Connectivity和Verify Geometry命令来做进一步的检查。

- Verify Connectivity检查是否完成了所有的连接，并在连接线上加入信息 (info)。
- 使用Report Info 来产生信息报告。
- Verify Geometry检查短路和设计规则violation并产生图形化的信息标记。用户应在执行了 Verify Connectivity和Verify Geometry后产生一个报告。

## Report Delay(SDF)



## 以GDSII格式保存设计

输入GDSII文件名

输入map文件名

输入顶层模块名



# MPW(Multi-Project Wafer)

---

- MPW将多种具有相同工艺的集成电路设计放在同一个硅圆片。
- 实验费用就由所有参加多项目晶圆的项目按照各自所占的芯片面积分摊，极大地降低了实验成本(90%)。
- 可以得到数十片芯片样品，用于设计开发阶段的实验、测试。
- 多项目晶圆提高了设计效率，降低了开发成本。

## Verilog语言简介

---

### 术语定义(terms and definitions)

---

- 硬件描述语言HDL：描述电路硬件及时序的一种编程语言
- 仿真器：读入HDL并进行解释及执行的一种软件
- 抽象级：描述方式的详细程度，如行为级和门级
- ASIC：专用集成电路(Application Specific Integrated Circuit)
- ASIC Vendor：芯片制造商，开发并提供单元库
- Bottom-up design flow：一种先构建底层单元，然后由底层单元构造更大的系统的设计方法。
- Top-down design flow：一种设计方法，先用高抽象级构造系统，然后再设计下层单元。
- RTL level：寄存器传输级(Register Transfer Level)，可综合的一种设计抽象级
- Tcl：Tool command Language, 向交互程序输入命令的描述语言

### 什么是硬件描述语言HDL

- 具有特殊结构能够对硬件逻辑电路的功能进行描述的一种高级编程语言
- 这种特殊结构能够：
  - 描述电路的连接
  - 描述电路的功能
  - 以不同的抽象级描述电路
  - 描述电路的时序
  - 描述具有并行性
- HDL主要有两种：Verilog和VHDL
  - Verilog起源于C语言，因此非常类似于C语言，容易掌握
  - VHDL起源于ADA语言，格式严谨，不易学习
  - VHDL出现较晚，但标准化早。IEEE 1706-1985标准

### Verilog的历史

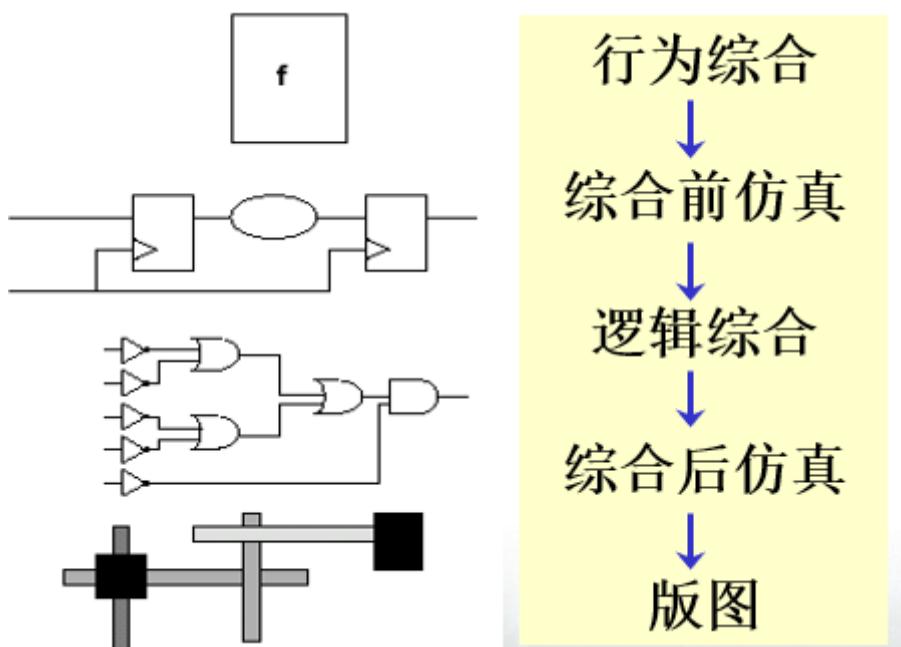
- Verilog HDL是在1983年由GDA(GateWay Design Automation)公司的Phil Moorby所创。Phil Moorby后来成为Verilog-XL的主要设计者和Cadence公司的第一个合伙人。
- 在1984~1985年间，Moorby设计出了第一个Verilog-XL的仿真器。
- 1986年，Moorby提出了用于快速门级仿真的XL算法。
- 1990年，Cadence公司收购了GDA公司
- 1991年，Cadence公司公开发表Verilog语言，成立了OVI(Open Verilog International)组织来负责Verilog HDL语言的发展。
- 1995年制定了Verilog HDL的IEEE标准，即IEEE1364。
- 2001年推出Verilog-2001标准
- 2005年推出Verilog-2005标准，即SystemVerilog
- IEEE Std 1800-xxxx：SystemVerilog现行标准
- Verilog-AMS：模拟及混合信号描述的Verilog扩展

# Verilog的用途

- Verilog的主要应用包括：
  - ASIC和FPGA工程师编写可综合的RTL代码
  - 使用高抽象级描述仿真系统，进行系统建模开发
  - 测试工程师用于编写各种层次的测试程序
  - 用于ASIC和FPGA单元或更高层次的模块的仿真模型开发

## 抽象级(Levels of Abstraction)

- Verilog既是一种行为描述的语言也是一种结构描述语言。
- Verilog模型可以是实际电路的不同级别的抽象。
- 这些抽象的级别包括：
  - 行为级(系统说明)：设计文档/算法描述
  - RTL/功能级：Verilog
  - 门级/结构级：Verilog
  - 版图/物理级：几何图形



Verilog可以在三种抽象级上进行描述：

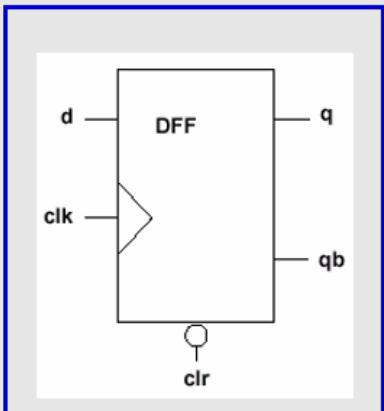
- 行为级
  - 用功能块之间的数据流对系统进行描述
  - 在需要时在函数块之间进行调度赋值。
- RTL级/功能级
  - 用功能块内部或功能块之间的数据流和控制信号描述系统
  - 基于一个已定义的时钟的周期来定义系统模型
- 结构级/门级
  - 用基本单元(primitive)或低层元件(component)的连接来描述系统以得到更高的精确性，特别是时序方面。
  - 逻辑综合时用特定工艺和低层元件将RTL描述映射到门级网表

## Verilog语言结构

module是层次化设计的基本构件

端口信号及内部信号  
数据类型:

net  
register  
parameter



module DFF (q, qb, d, clk, clr);

// 端口说明

output q, qb;

input d, // input data

clk, /\*input clock \*/ clr;

reg q;

wire qb, d, clk, clr;

/\*

clk is posedge and clr is active low

\*/

assign qb = ! q;

always @ (posedge clk or negedge clr)

if (!clr)

q <= 0;

else

q <= d;

endmodule

端口在模块名字后的括号中列出

端口可以说明为  
*input*, *output*及  
*inout*

功能描述放在  
**module**内部

## Verilog-2001语言结构

```
module DFF (
    // 端口说明
    output reg     q   ,
    output wire    qb  ,
    input  wire    d   ,      // input data
                           clk,      /*input clock */
                           clr
);
/*
  clk is posedge and clr is active low
*/
assign qb = !q;
always @ (posedge clk, negedge clr)
  if (!clr)
    q <= 0;
  else
    q <= d;
endmodule
```

## SystemVerilog

```
module DFF (
    // 端口说明
    output reg     q   ,
    output wire    qb  ,
    input  wire    d   ,      // input data
                           clk,      /*input clock */
                           clr
```

```

);
/*
  clk is posedge and clr is active low
*/
  assign qb = !q;
  always @(posedge clk or negedge clr)
    if(!clr)
      q <= 0;
    else
      q <= d;
endmodule

```

## 空白符和注释

```

module DFF (
  // 端口说明
  output reg q ,
  output wire qb,
  input wire d , // input data
  clk, /*input clock */
  clr
);
/*
  clk is posedge and
  clr is active low
*/
  assign qb = !q;
  always @(posedge clk , negedge clr)
    if(!clr)
      q <= 0;
    else
      q <= d;
endmodule

```

单行注释  
到行末结束

多行注释，在/\* \*/内

### 格式自由

使用空白符(空格，换行符) 提高可读性及代码组织。Verilog忽略空白符除非用于分开其它的语言标记。

```
reg cnt01,cnt02,cnt03,reg0;
```

```
assign qb=!q;
```

```
reg cnt01, cnt02, cnt03, reg0;
```

```
assign qb = ! q;
```

## 某标准制定的注释规则

1. 注释每一个功能逻辑，说明该功能逻辑的功能以及意图；
2. 注释的文本内容与注释符之间至少留有一个空格；
3. 验证修改时产生的无用代码应删除，而不能仅与以变更为注释；
4. 对所有的端口、信号、变量、函数、过程、常量、任务声明进行注释，应说明其意义、方向、有效值、作用等，宜注释与声明同行，如果注释过长，则将注释置于声明的上一行；
5. 注释应放在靠近被注释的代码附近，注释应简洁、精练；
6. 过程、任务、函数的注释应放在其代码之前，不应插入其中，避免中断代码的连贯性；
7. 在结束标识符、关键字 (end、endcase等) 后注明当前结束哪一个程序段,注释与结束标识符之间留有适当的间隔；
8. 局部区域内的注释应列对齐，以增强代码的可读性，如下图：

```

always @( posedge in_clk or negedge reset_n )
  if (reset_n == 1'b0)
    count_r <= 4'b0000 ;          //reset counter
  else if (count == 4'b1111)
    count_r <= 4'b0000 ;          //roll the counter over
  else
    count_r <= count_r + 4'b0001 ; //increment counter

```

## 属性(Attribute)

(\* begins an attribute, terminated by a \*).

- 属性指定Verilog的对象或语句的特殊属性，供特定软件工具（如逻辑综合）使用。属性是在Verilog-2001标准增加的。
- 属性可以作为声明、module、语句或端口的前缀出现。
- 属性可以作为运算符的或对函数调用的后缀出现。
- 属性可以被赋值。如果未指定值，则默认值为1。
- 可以指定多个属性，用逗号隔开。
- Verilog-2001标准没有定义标准属性；软件工具或其他标准可以根据需要定义属性。

```

(* full_case, parallel_case *) case (state)
  ...
endcase

assign sum = a + (* CLA=1 *) b;

```

## 标识符(identifiers)

- 标识符是用户在描述时给Verilog对象 定义的名称
- 标识符必须以字母(a-z, A-Z)或( \_ )开头，后面可以是字母、数字、\$ 或 \_。
- 最长可以是1023个字符；标识符**区分大小写**，sel和SEL是不同的标识符；模块、端口和实例的名称都是标识符

```

module DFF(
    // 端口说明
    output reg q,
    output wire qb,
    input wire d, // input data
    clk, /*input clock */
    clr
);
/*
    clk is posedge and clr is active low
*/
assign qb = !q;
always @(posedge clk or negedge clr)
    if(!clr)
        q <= 0;
    else
        q <= d;
endmodule

```

Verilog标识符

有效标识符举例:

```

shift_reg_a
busa_index
_bus3

```

无效标识符举例:

```

34net      // 开头不是字母或“_”
a*b_net   // 包含了非字母或数字, “$” “_”
n@238     // 包含了非字母或数字, “$” “_”

```

Verilog区分大小写, 所有Verilog关键词使用小写字母。

转义符, 主要是EDA工具使用

```

\34net      // 合法
\`a*b_net  // 合法
\`n@238    // 合法

```

## 某标准中的信号命名规则

1. 信号、变量名称不应超过32个(英文/数字)字符;
2. 信号、变量名称一律采用小写英文字母或数字; clk\_32M, mux4x1
3. 同一个时钟源驱动的时钟在不同模块和设计层级采用相同的时钟信号名称;
4. 使用同一复位信号的所有模块采用一致的复位信号名称;
5. 定义总线时, 采用一致的位排列顺序。对于Verilog, 使用[x:0];
6. 低电平有效信号名称后缀为“\_n”。例如: enable\_n, reset\_n;
7. 时钟信号名称前缀为“clk”。例如: clk\_cpu;
8. 复位信号名称前缀为“rst”。例如: rst\_n;

9. 三态信号名称后缀为“\_z”。例如：data\_z；大写？
10. 异步信号名称后缀为“\_a”。例如：strobe\_a； asyn
11. 寄存器输出信号名称后缀为“\_r”。例如：count\_r； reg
12. 状态变量命名使用不同的后缀。例如cs当前状态, ns下一状态；
13. 寄存器数据输入信号如果与寄存器同名，则添加后缀“i”或者“in”；
14. 测试相关信号名称后缀为“\_t”；
15. 模块输入信号添加前缀“i”，输出信号添加前缀“o”。

## 整数常数和实数常数

Verilog中，常量(literals)可是整数也可以是实数

- 整数的大小可以定义也可以不定义。整数表示为：

```
<size>'<base><value>
```

其中 size : 十进制数表示的二进制位数(bit)，缺省为32位

base: 数基，可为b、o、d、h进制，缺省为10进制

value: 是所选数基内任意有效数字，包括X、Z。

- 实数常量可以用十进制或科学表示法表示。

12	无符号十进制数（32位）
'H83a	无符号16进制数(32位)
8'b1100_0001	位二进制数
3'b1010_1101	3位2进制数，截为3'b101
64'hff01	64位16进制数
9'017	9位8进制数
32'b01x	32位2进制数
6.3	十进制实数
32e-4	科学法表示 0.0032
4.1E3	科学法表示 4100

- 整数的大小可以定义也可以不定义。整数表示为：
  - 数字中（\_）忽略，便于阅读，但不能出现在数字的首位。
  - 没有定义大小(size)整数缺省为32位
  - 缺省数基为十进制
  - 数基(base)和数字(16进制)中的字母无大小写之分
  - 当数值value大于指定的位数时，截去高位。如 2'b1101表示的是2'b01；小于指定的位数？
- 实数常量
  - 实数可用科学表示法或十进制表示
  - 科学表示法表示方式：

<尾数><e或E><指数>， 表示：尾数×10^指数

## 字符串 (string)

Verilog中，字符串大多用于显示信息的命令中。Verilog没有字符串数据类型。

- 字符串要在一行中用双引号括起来,不能跨行。
- 字符串中可以使用一些C语言转义(escape)符, 如\t \n
- 可以使用一些C语言格式符(如%b)在仿真时产生格式化输出:

" This is a normal string"

"This string has a \t tab and ends with a new line\n"

"This string formats a value: val = %b"

转义符及格式符将在验证支持部分讨论

### 格式符

%h	%o	%d	%b	%c	%s	%v	%m	%t
hex	oct	dec	bin	ASCII	string	strength	module	time

### 转义符

\t	\n	\	\"	<1-3 digit octal number>			
tab	换行	反斜杠	双引号	ASCII representation of above			

格式符%0d表示没有前导0的十进制数

## 输入输出信号及数据类型

### 三种端口:

- input
- output
- inout

### 三类(class)数据类型:

- net (连线) : 表示器件之间的物理连接
- register (寄存器) : 表示抽象存储元件
- parameter(参数) : 运行时的常数(run-time constants)

```
module DFF (
    // 端口说明
    output reg     q   ,
    output wire   qb  ,
    input  wire   d   ,      // input data
                    clk,      /*input clock */
                    clr
);
/*
    clk is posedge and clr is active low
*/
    assign qb = !q;
    always @(posedge clk , negedge clr)
        if(!clr)
```

```

q <= 0;
else
q <= d;
endmodule

```

## net的分类

- 有多种net类型用于设计(design-specific)建模和工艺(technology-specific)建模



net类型	功 能
wire, tri	标准内部连接线(缺省)
supply1, supply0	电源和地
wor, trior wand, triand trireg tri1, tri0	多驱动源线或 多驱动源线与 能保存电荷的net 无驱动时上拉/下拉

- 没有声明的net的缺省类型为1位无符号wire类型。但这个缺省类型可由下面的编译指令改变：

```
`default_nettype <nettype>
```

## register类的类型

- 寄存器类有五种数据类型

### 寄存器类型 功能

<b>reg</b>	可定义的无符号整数变量，可以是标量(1位)或矢量，是最常用的寄存器类型
<b>integer</b>	32位有符号整数变量，算术操作产生二进制补码形式的结果。通常用作不会由硬件实现的数据处理。
<b>real</b>	双精度的带符号浮点变量，用法与 <b>integer</b> 相同。
<b>time</b>	64位无符号整数变量，用于仿真时间的保存与处理
<b>realtime</b>	与 <b>real</b> 内容一致，但可以用作实数仿真时间的保存与处理

- 不要混淆寄存器数据类型与结构级存储元件，如udp\_dff

## Verilog中net和register声明语法

- net 声明 [signed unsigned]
 

```
<net_type> [range] [delay] <net_name>[,  
net_name];
```

net\_type: net类型  
 range: 矢量范围, 以 [MSB: LSB] 格式  
 delay: 定义与net相关的延时 (注意是固有延时)  
 net\_name: net名称, 一次可定义多个net, 用逗号分开。
- 寄存器声明 [signed unsigned]
 

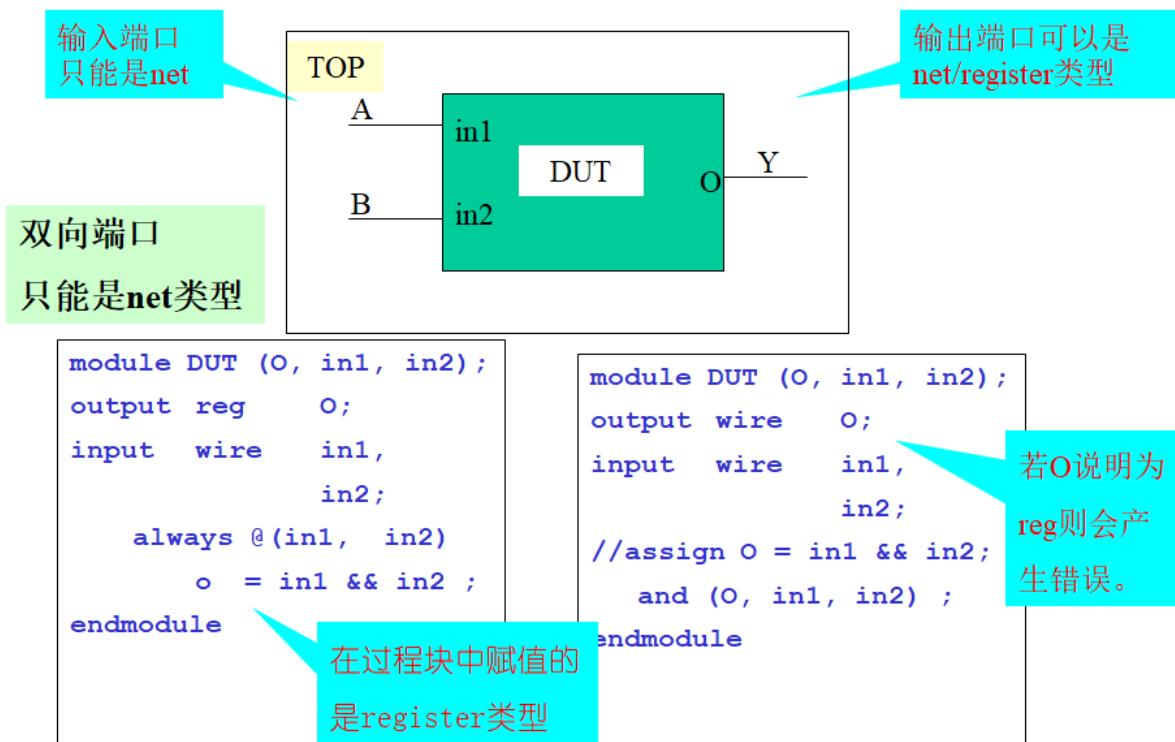
```
<reg_type> [range]  
<reg_name>[, reg_name];
```

reg\_type: 寄存器类型  
 range: 矢量范围, 以 [MSB: LSB] 格式。只对reg类型有效  
 reg\_name : 寄存器名称, 一次可定义多个寄存器, 用逗号分开

举例:

```
reg          a;           //一个标量寄存器
wand         w;           // 一个标量wand类型net
wire signed [3 : 0] d;   // 一个标量wand类型net
reg  [ 3 : 0] v;         // 从MSB到LSB的4位寄存器向量
reg  signed [ 7 : 0] m, n; // 两个有符号的8位寄存器
tri   [15: 0] busa;     // 16位三态总线
wire   [ 0: 31] w1, w2; // 两个32位wire, MSB为bit0
integer i;             // 32位整数i
```

## 选择正确的数据类型



- 输入只能是net
- 输出没有限制
- 过程赋值必须reg

## 参数 (parameters)

- 用参数声明一个可变常量，常用于定义延时及宽度变量。
- 参数定义的语法：parameter <赋值语句列表>;
- 可一次定义多个参数，用逗号隔开。
- 在使用常数(literal)的地方都可以使用参数。
- 参数的定义是局部的，只在当前模块中有效。
- 参数定义可使用以前定义的整数和实数参数。

```

module DFF (q, qb, d, clk, clr);
    parameter n = 4,
              m = n;
    output [n - 1 : 0] q, qb;
    input  [m - 1 : 0] d;
    input   clk,   clr;
    reg  [n - 1 : 0] q, qb;
    wire  [m - 1 : 0] d;
    wire   clk,   clr;

    assign qb = ~q;

    always @(posedge clk or negedge clr)
        if(!clr)
            q <= 0;
        else
            q <= d;

endmodule

```

```

module DFF #(q, qb, d, clk, clr);
parameter n = 4;
output [n - 1 : 0] q, qb;
input [n - 1 : 0] d;
input clk, clr;
reg [n - 1 : 0] q;
wire [n - 1 : 0] qb, d;
wire clk, clr;

assign qb = ~q;

always @(posedge clk or negedge clr)
  if(!clr)
    q <= 0;
  else
    q <= d;

endmodule

```

2001标准

```

module DFF #(parameter n = 4
) (
  output reg [n - 1 : 0] q,
  output wire [n - 1 : 0] qb,
  input wire [n - 1 : 0] d,
  input wire clk,
  clr
);

assign qb = ~q;

always @(posedge clk or negedge clr)
  if(!clr)
    q <= 0;
  else
    q <= d;

endmodule

```

## 参数重载 (overriding)

Defparam语句 (现在综合工具还不支持)

- 可用defparam语句在编译时重载参数值。
- defparam语句引用参数的层次化名称。
- 使用defparam语句可单独重载任何参数值。

```

module test;
  .
  .
  DFF Udff(q, qb, d, clk, clr);
  defparam
    Udff.n = 8;
  .
  .
endmodule

```

```

module test;
  .
  .
  DFF #(8, 8) Udff(q, qb, d, clk,
  clr);
  .
  .
endmodule

```

如果有多个参数，用逗号隔开，次序与原说明相同

```

module DFF #(
  parameter n = 4,
  m = 4
) (
  output reg [n - 1 : 0] q,
  wire [n - 1 : 0] qb,
  input wire [m - 1 : 0] d,
  input wire clk,
  clr
);

assign qb = ~q;

always @(posedge clk or negedge clr)
  if(!clr)
    q <= 0;
  else
    q <= d;

endmodule

```

## 功能描述

- 持续赋值 assign

- 过程块

- *initial*
- **always**

```
module DFF (
    // 端口说明
    output reg q,
    output wire qb,
    input wire d, // input data
                  clk, /*input clock */
                  clr
);
    assign qb = !q;

    always @ (posedge clk, negedge clr)
        if (!clr)
            q <= 0;
        else
            q <= d;
endmodule
```

## 持续赋值(continuous assignment)

- 描述的是**组合逻辑**
  - 在过程块外部使用。
  - 对net类型的信号赋值。
  - 在等式左边可以有一个简单延时说明。
  - 只限于在表达式左边用#delay形式
  - 可以是显式或隐含表示。

语法:

```
<assign> [#delay] [strength] <net_name> = <expressions>;
```

```
wire      out;  
assign #2 out  = a & b; // 显式  
wire      inv = ~in; // 隐含
```

## 过程块语句

块语句用来将多个语句组织在一起，使得他们在语法上如同一个语句。

块语句分为两类：

- 顺序块：语句置于关键字begin和end之间，块中的语句以顺序方式执行。
  - 并行块：关键字fork和join之间的是并行块语句，块中的语句并行执行。

```

initial c
begin
c _____
end

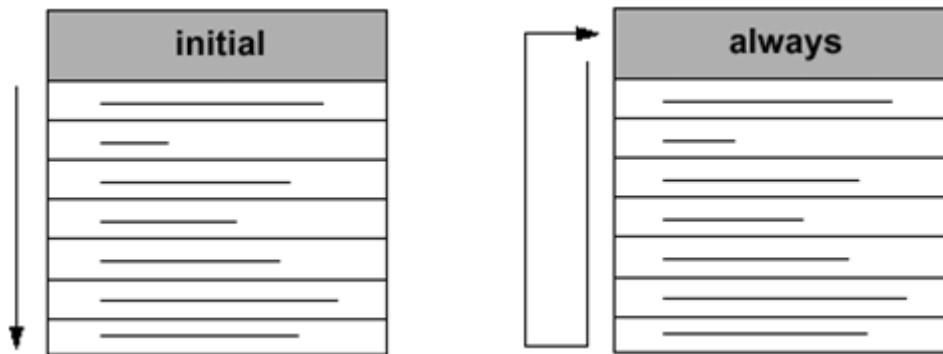
```

- Fork和join语句常用于test bench描述。这是因为可以一起给出矢量及其绝对时间，而不必描述所有先前事件的时间。

## 过程块(procedural block)

过程语句有两种：

- **initial**：只执行一次
- **always**：循环执行



所有过程在时间0执行一次

- 过程块之间
- assign语句之间并行执行
- 过程块与assign语句

```

module DFF (
    output reg q ,
    output wire qb ,
    input wire d , // input data
    clk, /*input clock */
    clr
);

assign qb = !q;

always (时序控制)
begin
    过程赋值语句;
    系统任务和自定义任务;
    高级描述语句
    {
        if语句;
        case语句;
        循环语句;
    }
end
endmodule

```

```

module DFF (
    output reg [3 : 0] q ,
    input wire sel ,
    input wire [3 : 0] dina ,
                dinb ,
                dinc
);
reg [ 3 : 0] data;

always @(dina, dinb, dinc) begin
    data = dina + dinb;
    if(sel)
        q = dina;
    else
        q = data;
    $display("result is %d", q);
end
endmodule

```

错误!

```

module DFF (
    output reg [3 : 0] q ,
    input wire sel ,
    input wire [3 : 0] dina ,
                dinb ,
                dinc
);
wire [ 3 : 0] data;

assign data = dina + dinb;

if(sel)
    q = dina;
else
    q = data;

$display("result is %d", q);
endmodule

```

## 过程赋值(procedural assignment)

- 在过程块中的赋值称为过程赋值。
- 表达式左边的信号必须是寄存器类型 (如reg类型)
- 等式右边可以是任何有效的表达式，数据类型也没有限制。
- 如果信号没有声明则缺省为wire类型。使用过程赋值语句给wire赋值会产生错误。

```
module adder (
    input wire      a      ,
                    b      ,
                    cin   ,
    output reg [1:0] out half_carry
);
    reg           half_sum;

    always @(*( a or b or cin)) begin
        half_sum = a ^ b ^ cin ; // OK
        half_carry = a & b | a & !b & cin | !a & b & cin ;
// ERROR!
        out          = {half_carry, half_sum} ;
    end
endmodule
```

## 持续赋值描述

```
module adder (
    input wire      a      ,
                    b      ,
                    cin   ,
    output wire [1:0] out
);
// reg           half_sum;

    assign half_sum = a ^ b ^ cin ; // OK
    assign half_carry = a & b | a & !b & cin | !a & b & cin ;
    assign out       = {half_carry, half_sum} ;

/*
always @(*( a or b or cin)) begin
    half_sum = a ^ b ^ cin ; // OK
    half_carry = a & b | a & !b & cin | !a & b & cin ;
    out       = {half_carry, half_sum} ;
end
*/
endmodule
```

```

module adder (out, a, b, cin);
    input a, b, cin;
    output [1:0] out;
    reg half_sum, half_carry;
    reg [1:0] out;
    always @ (a, b, cin) 时序控制
        begin
            half_sum = a ^ b ^ cin ;
            half_carry = a & b | a & !b & cin | !a & b & cin ;
            out = {half_carry, half_sum} ;
        end
endmodule

```

```

module adder (
    input wire a, b, cin,
    output reg [1:0] out
);
    reg half_sum, half_carry;
    always
    begin
        half_sum = a ^ b ^ cin ;
        half_carry = a & b | a & !b & cin | !a & b
& cin ;
        out = {half_carry, half_sum} ;
    end
endmodule

```

没有时序  
控制

当事件队列中所有事件结束  
后仿真器向前推进。但在零  
延时循环中，事件在同一时  
间片不断加入，使仿真器停  
滞后那个时片。

## 过程时序控制的种类

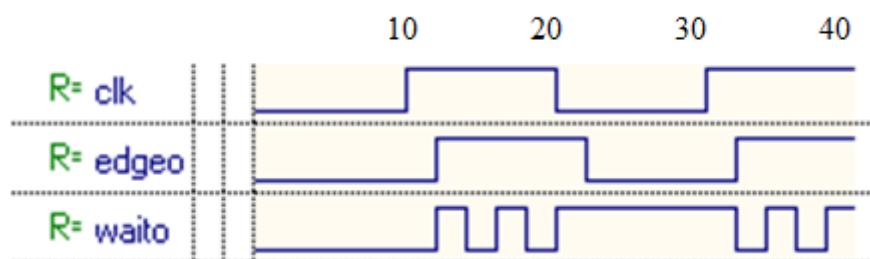
在过程块中可以说明过程时序。过程时序控制有三类：

- 简单延时(#delay): 延迟指定时间步后执行
- 边沿敏感的时序控制: `@(<signal>)`
  - 在信号发生翻转后执行。
  - 可以说明信号有效沿是上升沿(posedge)还是下降沿(negedge)。
  - 可以用关键字 or 或， 指定多个参数。
- 电平敏感的时序控制: `wait(<expr>)`
  - 直至expr值为真时 (非零) 才执行。
  - 若expr已经为真则立即执行。

```

module wait_test;
    reg clk, waito, edgeo;
    initial begin clk=0; edgeo=0; waito=0; end
    always #10 clk = ~clk;
    always @(clk) #2 edgeo = ~edgeo;
    always wait(clk) #2 waito = ~waito;
endmodule

```



## 简单延时

在test bench中使用简单延时 (#延时) 施加激励，或在行为模型中模拟实际延时。

```
module muxtwo (
    input      a, b, s1;
    output     reg out
);
always @(* s1 or a or b)
    if (! s1)
        #10 out = a;
    // 从a到out延时10个时间单位
    else
        #12 out = b;
    //从b到out延时12个时间单位
endmodule
```

**在简单延时中可以使用模块参数**parameter**:**

```
module clock_gen (
    output reg clk
);

parameter cycle = 20;
initial clk = 0;

always
    #(cycle/2) clk = ~clk;

endmodule
```

## 编译指令(Compiler Directives)

- (`)符号说明一个编译指令
- 这些编译指令使仿真编译器进行一些特殊的操作
- 编译指令一直保持有效直到被覆盖或解除
- `resetall 复位所有的编译指令为缺省值，应该在其它编译指令之前使用

## `timescale

- `timescale 说明时间单位及精度

格式: `timescale <time\_unit> / <time\_precision>

如: `timescale 1 ns / 100 ps

time\_unit: 延时或时间的测量单位

time\_precision: 延时值超出精度要先舍入后使用

- `timescale 必须在模块之前出现

**`timescale 1 ns / 10 ps**

```
module DFF (
    output reg q ,
    output wire qb ,
    input  wire d , // input data
                  clk, /*input clock */
                  clr
);

assign #2 qb = !q;
always @ (posedge clk or negedge clr)
    if (!clr)
        q <= 0;
    else
        q <= d;
endmodule
```

- time\_precision不能大于time\_unit
- time\_precision和time\_unit的表示方法: integer unit\_string
  - integer : 可以是1, 10, 100
  - unit\_string: 可以是s(second), ms(millisecond), us(microsecond), ns(nanosecond), ps(picosecond), fs(femtosecond)
  - 以上integer和unit\_string可任意组合
- precision的时间单位应尽量与设计的实际精度相同。
  - precision是仿真器的仿真时间步。
  - 若time\_unit与precision\_unit差别很大将严重影响仿真速度。
  - 如说明一个timescale 1s / 1ps, 则仿真器在1秒内要扫描其事件序列1012次; 而timescale 1s/1ms则只需扫描103次。
- 如果没有timescale说明将使用缺省值, 一般是s。

所有timescale中的最小值决定仿真时的最小时间单位。

这是因为仿真器必须对整个设计进行精确仿真

在下面的例子中，仿真时间单位（STU）为100fs

```
`timescale 1ns/ 10ps
module1 (. . .);
    not #1.23 (. . .) // 1.23ns or 12300 STUs
    .
endmodule

`timescale 100ns/ 1ns
module2 (. . .);
    not #1.23 (. . .) // 123ns or 1230000 STUs
    .
endmodule

`timescale 1ps/ 100fs
module3 (. . .);
    not #1.23 (. . .) // 1.23ps or 12 STUs (rounded off)
    .
endmodule
```

## 边沿敏感时序

时序控制@可以用在RTL级或行为级组合逻辑或时序逻辑描述中。可以用关键字posedge和negedge限定信号敏感边沿。敏感表中可以有多个信号，用关键字or或，连接。

```
module reg_adder (out, a, b, clk);
    input clk;
    input [2: 0] a, b;
    output [3: 0] out;
    reg [3: 0] out;
    reg [3: 0] sum;
    always @ (a or b) // 若a或b发生任何变化，执行
        #5 sum = a + b;
    always @ (posedge clk) // 在clk上升沿执行
        out = sum;
endmodule
```

注：事件控制符or和位或操作符|及逻辑或操作符||没有任何关系。

注2：如果信号列表中一个信号指定了边沿，则其它信号也必须指定边沿。

```
module reg_adder (
    input wire clk, rst;
    input wire [2 : 0] a, b;
    output [3: 0] out
);
    reg [3: 0] sum;
/*
    always @ (posedge clk, rst) // 错误
        if(!rst) out <= 0;
        else      out <= a + b;
*/
    always @ (posedge clk, negedge rst) // 正确

```

```

    if(!rst)  out <= 0;
    else      out <= a + b;
endmodule

```

## wait语句

wait用于行为级代码中电平敏感的时序控制。

下面 的输出锁存的加法器的行为描述中，使用了用关键字or的边沿敏感时序以及用wait语句描述的电平敏感时序。

```

module latch_adder (
    input  wire           enable,
    input  wire [2: 0]    a,
                        b,
    output reg [3: 0]    out
);
    always @(*( a or b)) begin
        wait (!enable) // 当enable为低电平时执行加法
        out = a + b;
    end
endmodule

```

注：综合工具还不支持wait语句。

## 条件语句（if分支语句）

*if* 和 *if-else* 语句：

```

always #20
    if (index > 0) // 开始外层 if
        if (regA > regB) // 开始内层第一层 if
            result = regA;
        else
            result = 0; // 结束内层第一层 if
    else if (index == 0)
        begin
            $display(" Note : Index is zero");
            result = regB;
        end
    else
        $display(" Note : Index is negative");

```

描述方式：

*if* (逻辑表达式)

begin

.....

end

*else*

begin

.....

end

- 可以多层嵌套。在嵌套if序列中，else和前面最近的if相关。
- 为提高可读性及确保正确关联，使用begin...end块语句指定其作用域。

## 条件语句（case分支语句）

在Verilog中重复说明case项是合法的，因为Verilog的case语句只执行第一个符合项。

```
module compute (
    input wire [7: 0] rega      ,
    regb      ,
    input wire [2: 0] opcode ,
    output reg [7: 0] result
);

always @(* rega , regb , opcode)
    case (opcode)
        3'b000 : result = rega + regb;
        3'b001 : result = rega - regb;
        3'b010 ,           // 多个case有同一个结果
        3'b100 : result = rega / regb;
        default : begin
            result = 'bx;
            $display (" no match");
        end
    endcase
endmodule
```

case语句是测试表达式与另外一系列表达式分支是否匹配的多路条件语句。

- case语句进行逐位比较以求完全匹配（包括x和z）。
- default语句可选，在没有任何条件成立时执行。此时如果未说明default，Verilog不执行任何动作。
- 多个default语句是非法的。

重要内容：

使用default语句是一个很好的编程习惯，特别是用于检测x和z。

Casez和casex为case语句的变体，允许比较无关(don't-care) 值。

- case表达式的任何位为无关值时，在比较过程中该位不予考虑。
- 在casez语句中，? 和 z 被当作无关值。
- 在casex语句中，?, z 和 x 被当作无关值。

```
case (表达式)
    <表达式>, <表达式>; 赋值语句或空语句;
    <表达式>, <表达式>; 赋值语句或空语句;
                           default: 赋值语句或空语句;
endcase
```

### casez语句

```
module coding3_8(
    input wire [7:0] a,
    output reg [2:0] o
);
```

```

always @(a)
casez(a)
  8'b1???_????:   o = 3'b111;
  8'b01??_????:   o = 3'b110;
  8'b001?_????:   o = 3'b101;
  8'b0001_????:   o = 3'b100;
  8'b0000_1????: o = 3'b011;
  8'b0000_01???: o = 3'b010;
  8'b0000_001?:  o = 3'b001;
  8'b0000_0001:   o = 3'b000;
  default:           o = 3'bx;
endcase
endmodule

```

## casex语句

```

module coding3_8(
  input wire [7:0] a,
  output reg [2:0] o
);

always @(a)
casex(a)
  8'b1xxx_xxxx:   o = 3'b111;
  8'b01xx_xxxx:   o = 3'b110;
  8'b001x_xxxx:   o = 3'b101;
  8'b0001_xxxx:   o = 3'b100;
  8'b0000_1xxx:   o = 3'b011;
  8'b0000_01xx:   o = 3'b010;
  8'b0000_001x:   o = 3'b001;
  8'b0000_0001:   o = 3'b000;
  default:           o = 3'bx;
endcase
endmodule

```

## 循环(looping)语句

有四种循环语句：

**repeat**: 将一块语句循环执行确定次数。

**repeat** (次数表达式) <语句>

**while**: 在条件表达式为真时一直循环执行

**while** (条件表达式) <语句>

**forever**: 重复执行直到仿真结束

**forever** <语句>

**for**: 在执行过程中对变量进行计算和判断，在条件满足时执行

**for**(赋初值; 条件表达式; 计算) <语句>

IEEE1364  
 综合工具  
 还不支持  
 在2001标  
 准以后支  
 持while

## 循环(looping)语句-repeat

repeat: 将一块语句循环执行确定次数。

repeat (次数表达式) 语句

```
module multiplier #(  
    parameter size = 8  
)  
begin  
    input  wire [size : 1]      op_a, op_b,  
    output reg [2 * size : 1]   result  
);  
reg [2 * size : 1] shift_opa;  
reg [size : 1] shift_opb;  
always @(* op_a or op_b) begin  
    result = 0;  
    shift_opa = op_a; // 零扩展至16位  
    shift_opb = op_b;  
    repeat (size) begin  
        #10 if (shift_opb[1]) result = result + shift_opa;  
        shift_opa = shift_opa << 1; // Shift left  
        shift_opb = shift_opb >> 1; // Shift right  
    end  
end  
endmodule
```

上述程序是模拟手工乘法

```
5 0 1 0 1    opa  
3 0 0 1 1    opb -> shift_opb  
-----  
          0 1 0 1    shift_opa  
          0 1 0 1  
          0 0 0 0  
          0 0 0 0  
-----  
          0 0 0 0 1 1 1 1    15 <- result
```

## 循环语句--while

while: 只要表达式为真(不为0), 则重复执行一条语句(或语句块)

```
module tmp(  
    input wire [7:0] a,  
    output reg [3:0] count  
);  
reg [7: 0] tempreg;  
  
always @(*) begin  
    count = 0;  
    tempreg = a;  
    while (tempreg) // 统计tempreg中 1 的个数  
    begin  
        if (tempreg[ 0]) count = count + 1;  
        tempreg = tempreg >> 1; // 右移  
    end  
end
```

```
end  
endmodule
```

注: While(A): A如果是个二进制串, 中每一位都不为1的时候跳出

## 循环语句--forever

*forever*: 一直执行到仿真结束

*forever*应该是过程块中最后一条语句。其后的语句将永远不会执行。

*forever*语句不可综合, 通常用于test bench描述。

```
...  
reg clk;  
initial  
begin  
    clk = 0;  
    forever  
        begin  
            #10 clk = 1;  
            #10 clk = 0;  
        end  
    end  
...
```

- 这种行为描述方式可以非常灵活的描述时钟, 可以控制时钟的开始时间及周期占空比。仿真效率也高。
- Forever 可以实现各种占空比的时钟信号

## 循环语句--for

*for*: 只要条件为真就一直执行

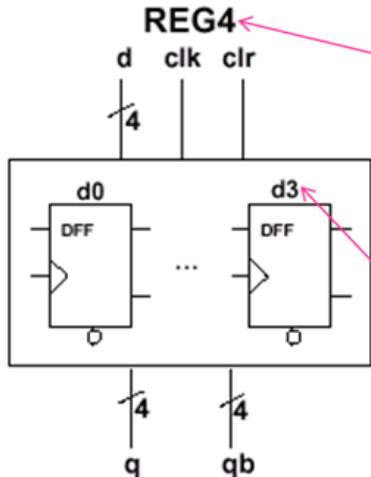
条件表达式若是简单的与0比较通常处理得更快一些。但综合工具可能不支持与0的比较。(循环变量可以声明为integer或reg矢量)

```
integer index, i, j;  
// X检测  
Initial begin  
    for (index = 0; index < size; index = index + 1)  
        if (val[index] === 1'bX)  
            $display (" found an X");  
  
    // 存储器初始化; “!= 0”仿真效率高  
    for (i = size; i != 0; i = i - 1)  
        memory[i-1] = 0;  
  
    // 阶乘序列  
    factorial = 1;  
    for (j = num; j != 0; j = j - 1)  
        factorial = factorial * j;  
end
```

注意: verilog 不支持 *i++* 的写法, *i=i+1* 代替

## 模块实例化 (module instances)

通过模块实例化构造设计的层次体系



```
module DFF (d, clk, clr, q, qb);
    ....
endmodule

module REG4(
    output wire [3: 0] q ,
    qb,
    input wire [3: 0] d ,
    input wire clk,
    clr );
    DFF d0 (d[0], clk, clr, q[0], qb[0]);
    DFF d1 (d[1], clk, clr, q[1], qb[1]);
    DFF d2 (d[2], clk, clr, q[2], qb[2]);
    DFF d3 (d[3], clk, clr, q[3], qb[3]);
endmodule
```

## Verilog操作符

### 操作符类型

下表以优先级顺序列出了Verilog操作符。注意“与”操作符的优先级总是比相同类型的“或”操作符高。本章将对每个操作符用一个例子作出解释。

操作符类型	符号	
连接及复制操作符	{ } {()}	最高
一元操作符	! ~ &   ^	
算术操作符	* / % + -	
逻辑移位操作符	<< >> >>> <<<	
关系操作符	> < >= <=	
相等操作符	== === != !==	
按位操作符	& ^ ~^	
逻辑操作符	&& 	
条件操作符	? :	最低

## 变量的位数与符号

- Verilog根据表达式中变量的长度对表达式的值自动地进行调整。
- Verilog自动截断或扩展赋值语句中右边的值以适应左边变量的长度。
- 当一个负数赋值给无符号变量如reg时，Verilog自动完成二进制补码计算

```
module sign_size;
    reg [3:0] a,
              b;
    reg [15:0] c;
    initial begin
        a = -1; // a是无符号数，因此其值为4'b1111
        b = 8; c = 8; // b = c = 4'b1000
        #10 b = b + a; // 结果4'b10111截断，b=4'b0111
        #10 c = c + a; // c = 16'b0000_0000_0001_0111
        #10 c = b + a;
    end
endmodule
```

```
module sign_size;
    reg signed [ 3 : 0] a;
    reg signed [15 : 0] c;
    initial begin
        a = -1;
        c = 8;
        c = c + a;
    end
endmodule
```

## 算术操作符

+	加
-	减
*	乘
/	除
%	模

- 将负数赋值给reg或其它无符号变量使用2的补码算术。
- 如果操作数的某一位是x或z，则结果为x
- 在整数除法中，余数舍弃
- 模运算中使用第一个操作数的符号

```
module arithops;
    parameter five = 5;
    integer ans, int;
    reg [3: 0] rega, regb;
    reg [3: 0] num;
    initial begin
        rega = 3;
        regb = 4'b1010;
        int      = -3;      //int = 1111.....1111_1101
    end
    initial begin
        #10 ans    = five * int;           // ans = -15
    end
endmodule
```

```

#10 ans    = (int + 5) / 2;      // ans = 1
#10 ans    = five / int;        // ans = -1
#10 num = rega + regb;        // num = 1101
#10 num = rega + 1;           // num = 0100
#10 num = int;                // num = 1101
#10 num = regb % rega;       // num = 1
#80 $finish;
end
endmodule

```

注意：integer和reg类型在算术运算时的差别。integer是有符号数，而reg是无符号数。

## 除法及模余

```

`timescale 1 ns / 1 ns

module divider ();
    integer a, b;
    initial begin
        a = 7; b = 4;
        $display("%0d/%0d quot and rem is %0d %0d", a, b, a/b, a%b);
        a = 7; b = -4;
        $display("%0d/%0d quot and rem is %0d %0d", a, b, a/b, a%b);
        a = -7; b = 4;
        $display("%0d/%0d quot and rem is %0d %0d", a, b, a/b, a%b);
        a = -7; b = -4;
        $display("%0d/%0d quot and rem is %0d %0d", a, b, a/b, a%b);
        $finish;
    end
endmodule

```

7/4 quotient and remainder is 1 3

7/-4 quotient and remainder is -1 3

-7/4 quotient and remainder is -1 -3

-7/-4 quotient and remainder is 1 -3

- 在整数除法中，余数舍弃
- 模运算中使用第一个操作数的符号

## 有符号数与无符号数混合运算

```

`timescale 1ns/1ns
module operate ();
    reg signed[3: 0] rega;
    reg          [3: 0] regb;
    reg signed[7: 0] regc;
    initial begin
        rega = 4'b1001;      // -8'd7, 0扩展至8'b0000_1001
        regb = 4'b1110;      // 8'd14, 0扩展至8'b0000_1110
        regc = 8'b0;
        #100    regc = rega + regb; //regc = 8'b0001_0111
        $display($time, " regc = %d %b", regc, regc);
        #100 $finish;
    end
endmodule

```

- 进行有有符号数计算，强制regb是有符号数

如果在设计中regb虽然声明为无符号数，但它实际上是有符号数。在这种情况下，我们需要使用系统函数\$signed将其转换为有符号数再参与计算。

```
`timescale 1ns/1ns
module operate ();
    reg signed[3: 0] rega;
    reg [3: 0] regb;
    reg signed[7: 0] regc;
    initial begin
        rega = 4'b1001;      // -8'd7, 符号扩展至8'b1111_1001
        regb = 4'b1110;      // -8'd2, 符号扩展至8'b1111_1110
        regc = 8'b0;
        #100 regc = rega + $signed (regb); // regc = 8'b1111_0111
        $display($time, " regc = %d %b", regc, regc);
        #100 $finish;
    end
endmodule
```

- 进行有符号数计算，但regb是无符号数，即正数

如果无符号数要与有符号数进行计算，由于无符号数是正数，因此我们需要将无符号数扩展1位，即高位补0，这个补入的0作为符号参与计算。如前例中，regb是一个无符号数，即正数，要参与计算需先0扩展，再转换为有符号数，如下例所示，regc的计算结果为7。

```
`timescale 1ns/1ns
module operate ();
    reg signed[3: 0] rega;
    reg [3: 0] regb;
    reg signed[7: 0] regc;
    initial begin
        rega = 4'b1001;      // -8'd7, 符号扩展至8'b1111_1001
        regb = 4'b1110;      // 8'd14, 符号扩展至8'b0000_1110
        regc = 8'b0;
        #100 regc = rega + $signed ({1'b0, regb}); // regc = 8'b0000_0111
        $display($time, " regc = %d %b", regc, regc);
        #100 $finish;
    end
endmodule
```

## SYNOPSYS DesignWare Library

- DesignWare Library是一组可重用的、可综合的IP块，集成在Synopsys综合环境中。
- 缺省的DesignWare库是standard.sldb，包括：
  - adder: +, +1
  - subtractor: -, -1
  - comparator: ==, !=, <, <=, >, >=
  - mulpilier, divider
  - sin cos tan
  - Sqrt FIR IIR



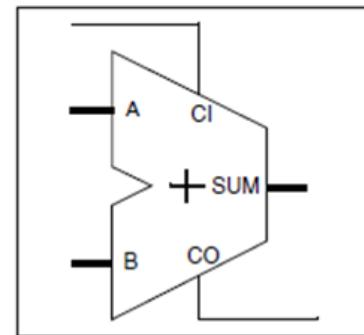
## DW01\_add

### Adder

[Version, STAR and Download Information: IP Directory](#)

### Features and Benefits

- Parameterized word length
- Carry-in and carry-out signals



### Description

DW01\_add adds two operands A and B with a carry-in CI to produce the output SUM with a carry-out CO.

Table 1-1 Pin Description

Pin Name	Width	Direction	Function
A	width bit(s)	Input	Input data
B	width bit(s)	Input	Input data
CI	1 bit	Input	Carry-in
SUM	width bit(s)	Output	Sum of (A + B + CI)
CO	1 bit	Output	Carry-out

### Usage Through Operator Inferencing

```
module DW01_add_oper (in1, in2, sum);
    parameter wordlength = 8;
    input [wordlength-1:0] in1, in2;
    output [wordlength-1:0] sum;

    assign sum = in1 + in2;
endmodule
```

```
module DW01_add_inst( inst_A, inst_B, inst_CI, SUM_inst, CO_inst );
    parameter width = 8;
    input [width-1 : 0] inst_A;
    input [width-1 : 0] inst_B;
    input inst_CI;
    output [width-1 : 0] SUM_inst;
    output CO_inst;

// Instance of DW01_add
DW01_add #(width)
    U1 (.A(inst_A), .B(inst_B), .CI(inst_CI), .SUM(SUM_inst),
    .CO(CO_inst) );

endmodule
```

- 如果用户在设计中使用了**DesignWare**的元件，用户可以使用约束来改变其实现方式

- 实现方式查看**DW**手册

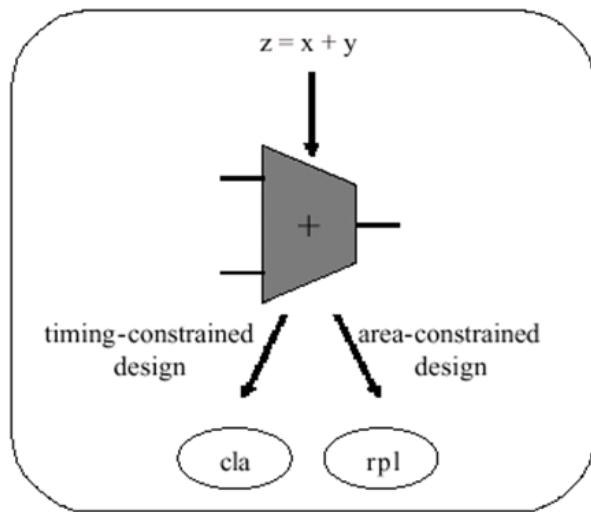


Table 4 - Synthesis Implementations

Implementation Name	Function	License Required
rpl	Ripple carry synthesis model	DesignWare-Basic
cla	Carry look-ahead synthesis model	DesignWare-Basic
clf	Fast carry look-ahead synthesis model	DesignWare-Foundation
bk <sup>1</sup>	Brent-Kung architecture synthesis model	DesignWare-Foundation
csm <sup>2</sup>	Conditional sum synthesis model	DesignWare-Foundation

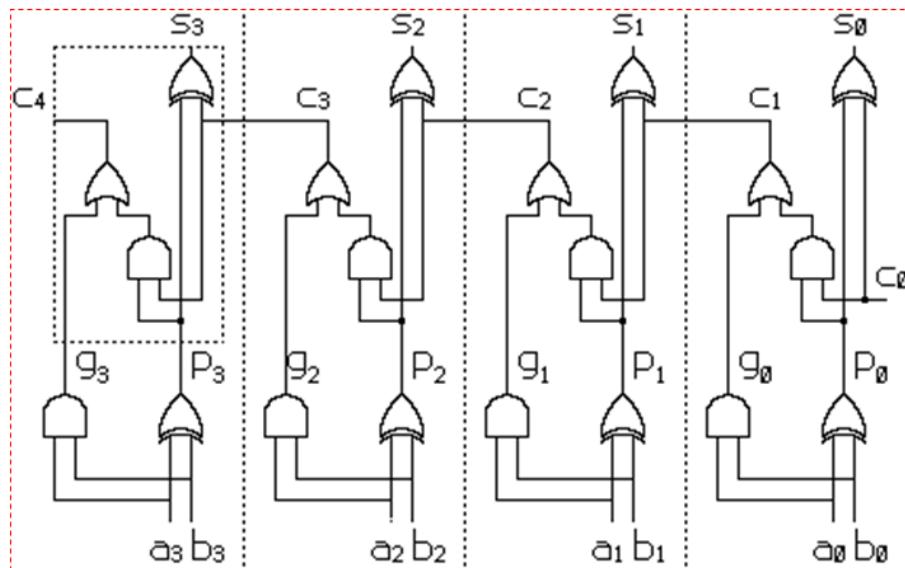
## 加法器设计

### 行波进位加法器 (RPL/RCA)

Ripple Carry Adder

$$\begin{aligned}
 c &= a \cdot b + a \cdot ci + b \cdot ci \\
 &= a \cdot b + (a + b) \cdot ci \\
 &= a \cdot b + (a \cdot b + a \cdot \bar{b} + \bar{a} \cdot b) \cdot ci \\
 &= a \cdot b + (a \oplus b) \cdot ci \\
 s &= a \oplus b \oplus ci
 \end{aligned}$$

$$\begin{aligned}
 g &= a \cdot b; && \text{generate} \\
 p &= a \oplus b; && \text{propagate} \\
 c &= g + p \cdot ci \\
 s &= p \oplus ci
 \end{aligned}$$



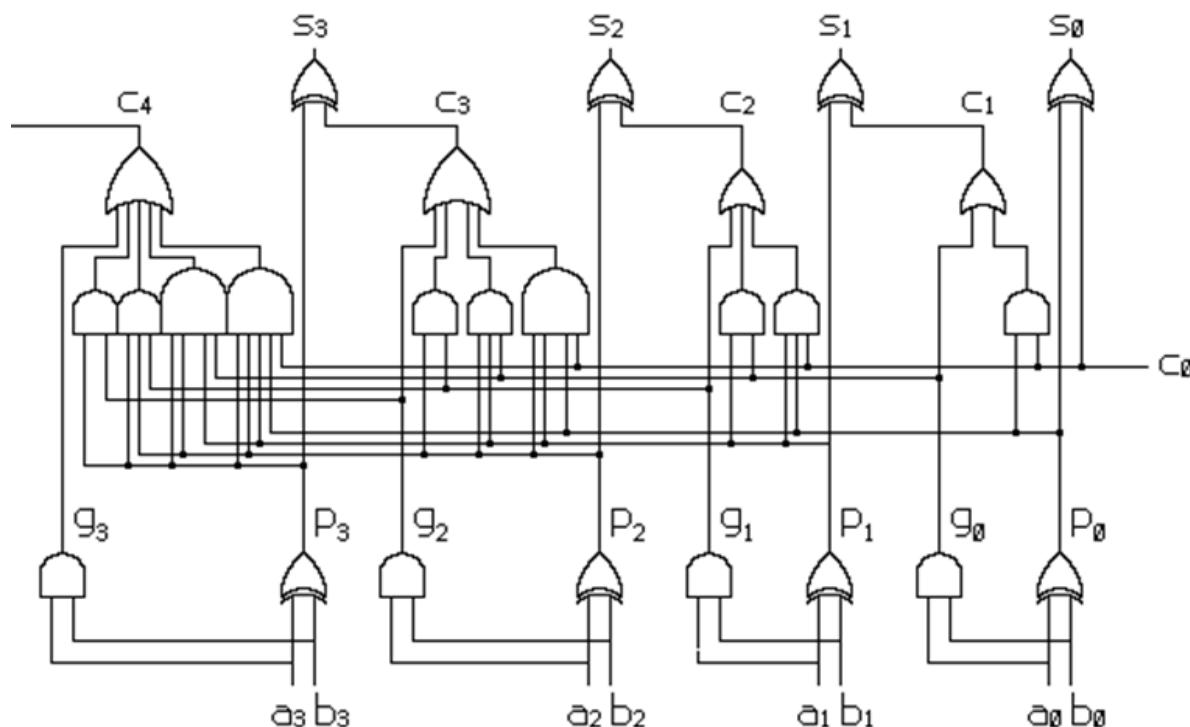
## 超前进位加法器

CLA(Carry Look-ahead Adder)

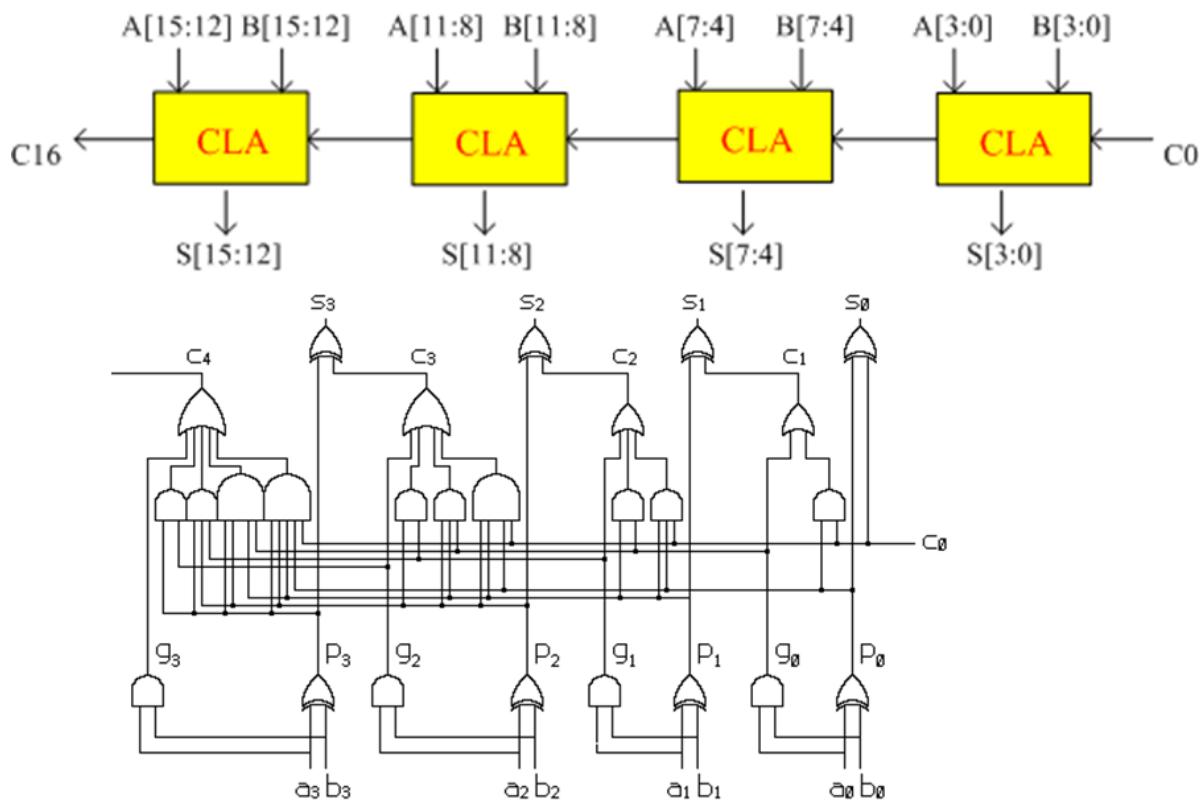
$$\begin{aligned}
 c_0 &= c_i \\
 c_1 &= g_0 + p_0 c_0 \\
 c_2 &= g_1 + p_1 c_1 \\
 &= g_1 + p_1(g_0 + p_0 c_0) \\
 &= g_1 + p_1 g_0 + p_1 p_0 c_0 \\
 c_3 &= g_2 + p_2 c_2 \\
 &= g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_0 \\
 c_4 &= g_3 + p_3 c_3 \\
 &= g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0 + p_3 p_2 p_1 p_0 c_0 \\
 c_o &= c_4
 \end{aligned}$$

$$\begin{aligned}
 c_0 &= c_i \\
 c_1 &= g_0 + p_0 c_0 \\
 G_0 &= g_0; P_0 = p_0 \\
 c_2 &= G_1 + P_1 c_0 \\
 G_1 &= p_1 g_0; P_1 = p_1 p_0; \\
 c_3 &= G_2 + P_2 c_0 \\
 G_2 &= g_2 + p_2 g_1 + p_2 p_1 g_0; \\
 P_2 &= p_2 p_1 p_0; \\
 c_4 &= G_3 + P_3 c_0 \\
 G_3 &= g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0; \\
 P_3 &= p_3 p_2 p_1 p_0;
 \end{aligned}$$

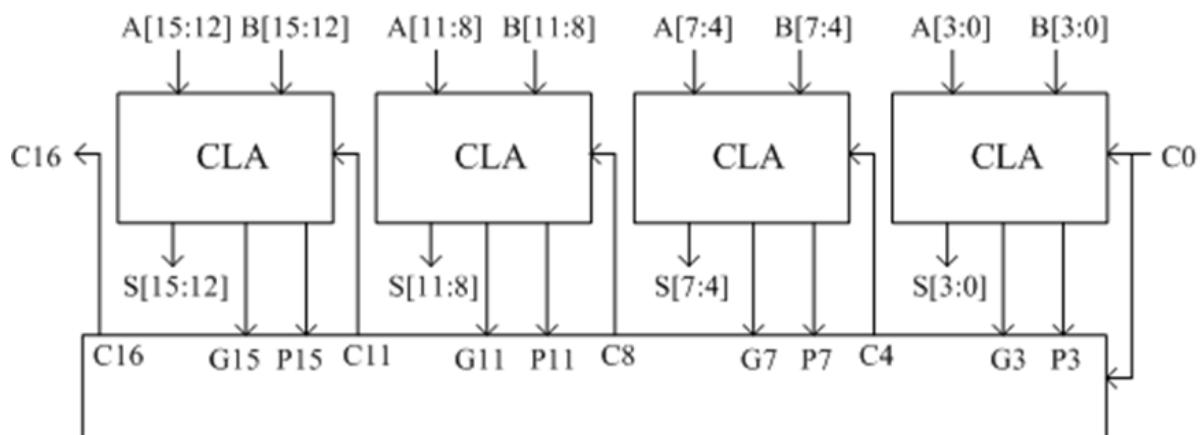
### 4位超前进位加法器 (CLA)



### 16位CLA加法器



块间超前进位加法器

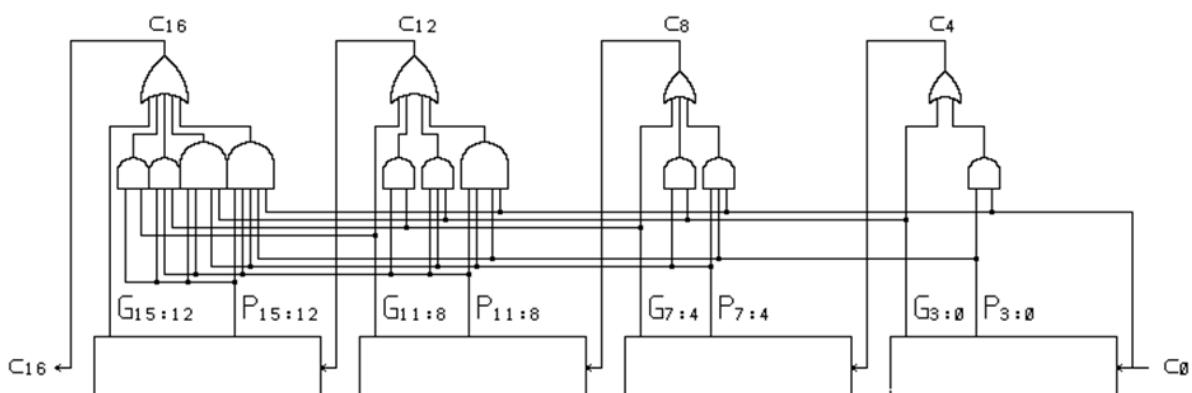


$$c_{16} = G_{15} + P_{15}c_0$$

$$G_{15} = g_{15} + p_{15}g_{11} + p_{15}p_{11}g_7 + p_{15}p_{11}p_7g_3 ;$$

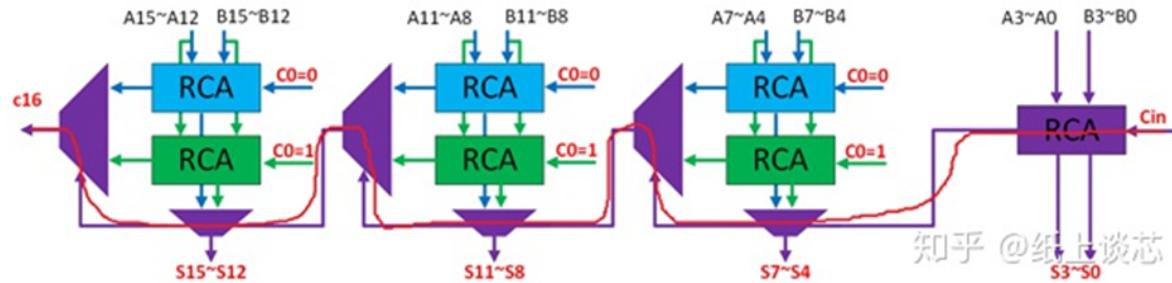
$$P_{15} = p_{15}p_{11}p_7 \cdot p_3;$$

快速16位CLA加法器



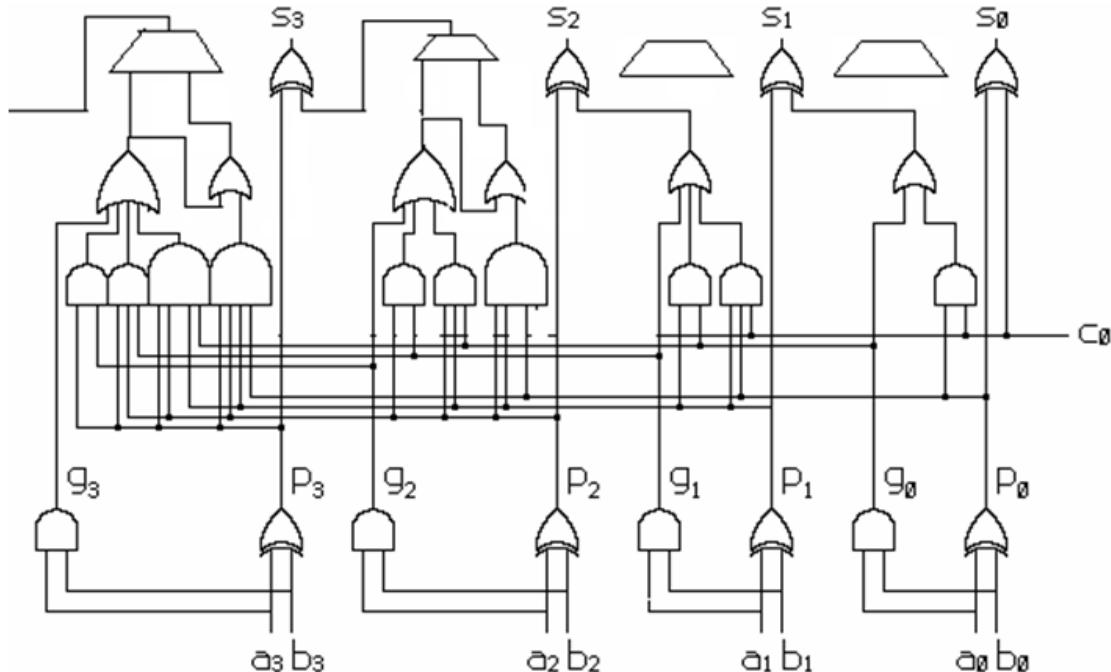
## 16位选择进位加法器 (CSA)

CSA (Carry Select Adder)



$$c_4 = G_3 + P_3 c_0$$

$$c_4 = G_3 \quad \text{当 } c_0 = 0 \\ = G_3 + P_3 \quad \text{当 } c_0 = 1$$



## 定点乘法器设计

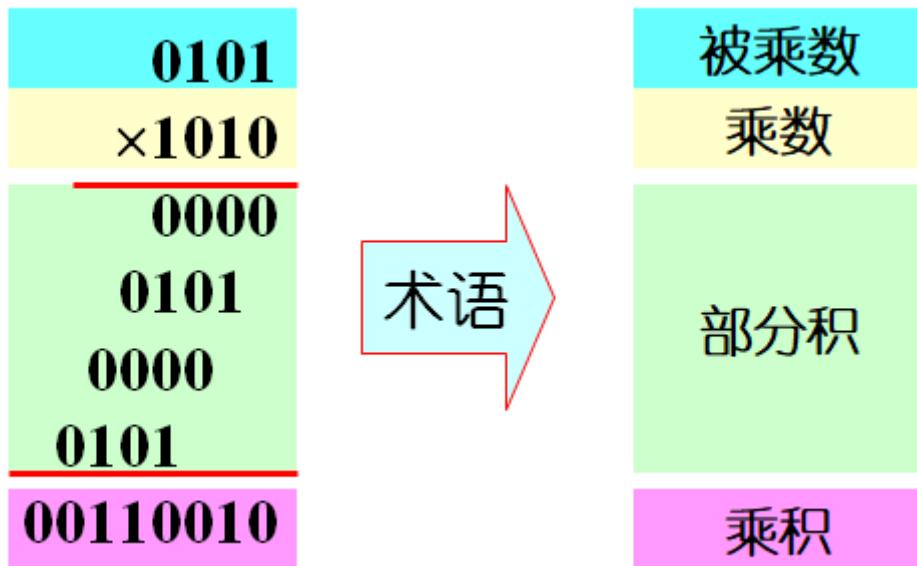
- 乘法器是高性能微处理器中的关键部件，是进行高速计算特别是信号处理等方面应用时所必须的。

被乘数:  $A = a_{n-1}a_{n-2} \dots a_0$

乘 数:  $B = b_{n-1}b_{n-2} \dots b_0$

乘 积:  $P = A \times B$

## 乘法计算方法



- 乘法计算过程

- 部分积产生
- 部分积相加

### 乘法运算的关键

- 要提高乘法计算速度，需要：
  - 加快部分积的形成
  - 减少部分积数目
    - 采用多位扫描、跳过连续的0/1串和对乘数重编码（如Booth算法）等处理方法
  - 加快部分积加法运算的速度
  - 一般是利用进位保留加法器（CSA, Carry Save Adder）先使参与操作的部分积形成两个数（这两个数分别是偶和与局部进位）

### 二阶Booth算法

乘 数：  $B = b_{n-1}b_{n-2} \dots b_0$

$$B = -b_{n-1}2^{n-1} + \sum_{i=0}^{n-2} b_i 2^i$$

乘数必须  
是偶数位

$$= -b_{n-1}2^{n-1} + b_{n-2}2^{n-2} + b_{n-3}2^{n-3} + b_{n-4}2^{n-4} + \dots + b_02^0$$

$$= -2b_{n-1}2^{n-2} + b_{n-2}2^{n-2} + b_{n-3}2^{n-2} - b_{n-3}2^{n-3} + b_{n-4}2^{n-4} + \dots + b_02^0$$

$$= (-2b_{n-1} + b_{n-2} + b_{n-3})2^{n-2} - b_{n-3}2^{n-3} + b_{n-4}2^{n-4} + \dots + b_02^0$$

$$B = \sum_{i=0}^j (-2b_{2i+1} + b_{2i} + b_{2i-1})2^{2i}$$

$$5 \times 31_h = F5_h$$

最低位补  
一位0

00110001

0 0 1 1 0 0 0 1 0

1 1 1  
-2<sup>1</sup> 2<sup>0</sup> 2<sup>0</sup>

二阶Booth编码

$b_{i+1} b_i b_{i-1}$	重编码	操作
000	0	加0; 中间结果右移2位
001	1	加A; 中间结果右移2位
010	1	加A; 中间结果右移2位
011	2	加2A; 中间结果右移2位
100	-2	减2A; 中间结果右移2位
101	-1	减A; 中间结果右移2位
110	-1	减A; 中间结果右移2位
111	0	加0; 中间结果右移2位

0101 0  
0101  
0000  
1010  
0101 1  
—————  
0111110101

67×67位乘法器的改进四阶Booth算法实现 西安交通大学电子与信息工程学院

$$X = -x_{n-1}(2)^{n-1} + \sum_{i=0}^{n-2} x_i 2^i = 2^{n-4}(-2^3 x_{n-1} + 2^2 x_{n-2} + 2 x_{n-3} + x_{n-4} + x_{n-5}) + 2^{n-8}(-2^3 x_{n-5} + 2^2 x_{n-6} + 2 x_{n-8} + x_{n-9}) + \dots \quad (1)$$

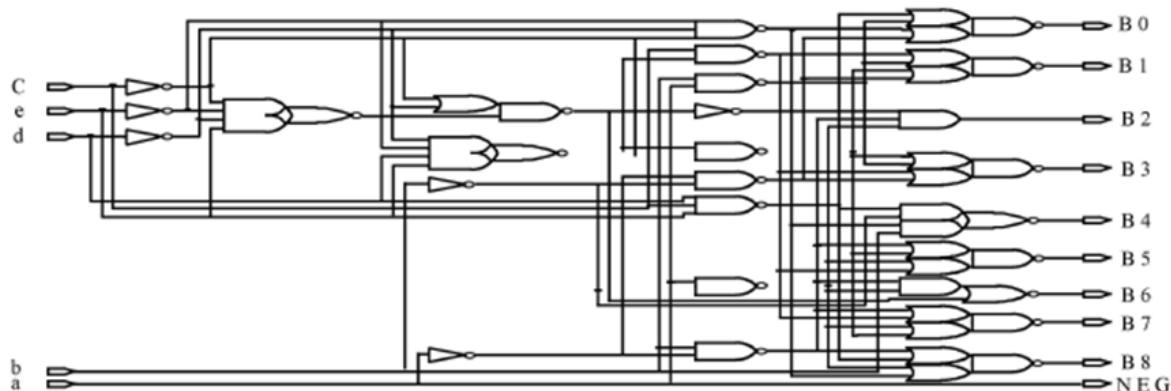


图 1 四阶 Booth 编码电路

**表 1 四阶 Booth 编码表**

二进制代码	部分积	二进制代码	部分积
0000(0)	0 M	1000(0)	-8 M
0000(1)	+ M	1000(1)	-7 M
0000(0)	+ M	1001(0)	-7 M
0000(1)	+2 M	1001(1)	-6 M
0010(0)	+2 M	1010(0)	-6 M
0010(1)	+3 M	1010(1)	-5 M
0011(0)	+3 M	1011(0)	-5 M
0011(1)	+4 M	1011(1)	-4 M
0100(0)	+4 M	1100(0)	-4 M
0100(1)	+5 M	1100(1)	-3 M
0101(0)	+5 M	1101(0)	-3 M
0101(1)	+6 M	1101(1)	-2 M
0110(0)	+6 M	1110(0)	-2 M
0110(1)	+7 M	1110(1)	- M
0111(0)	+7 M	1111(0)	- M
0111(1)	+8 M	1111(1)	0 M

TSMC0.18μm的工艺

**表 4 不同方法性能比较(67位宽)**

算法	延时/ns	面积/ $\mu\text{m}^2$
DesignWare	15.58	589943.812500
改进的 Booth4 算法	10.85	375625.937500

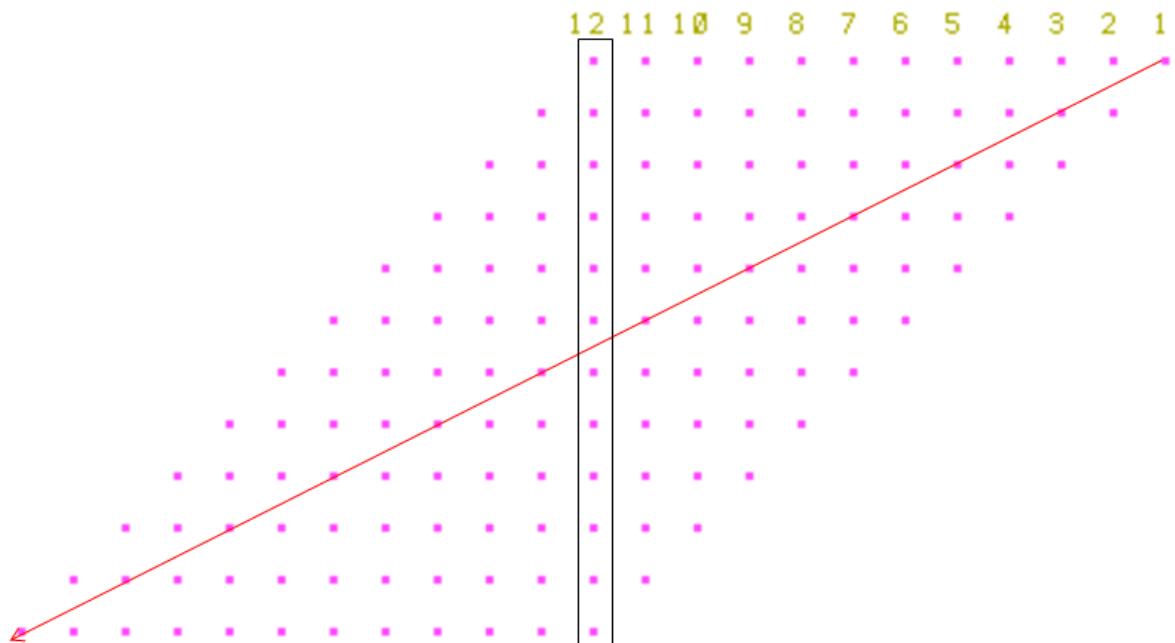
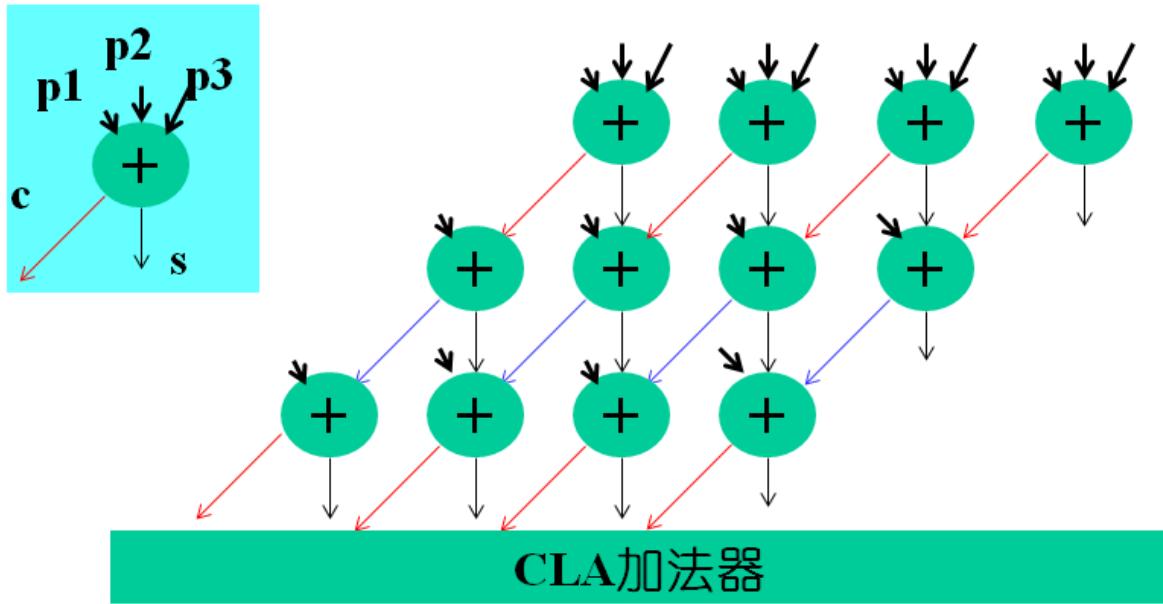
Altera的FPGA EP1C20F400C8.

**表 5 考虑布线的各算法的 FPGA 实现**

算法	Total delay/ns	Total logic elements
DesignWare	154.785	9,227
改进的 Booth4 算法	125.866	10,792

## 加法器树-保留进位加法器

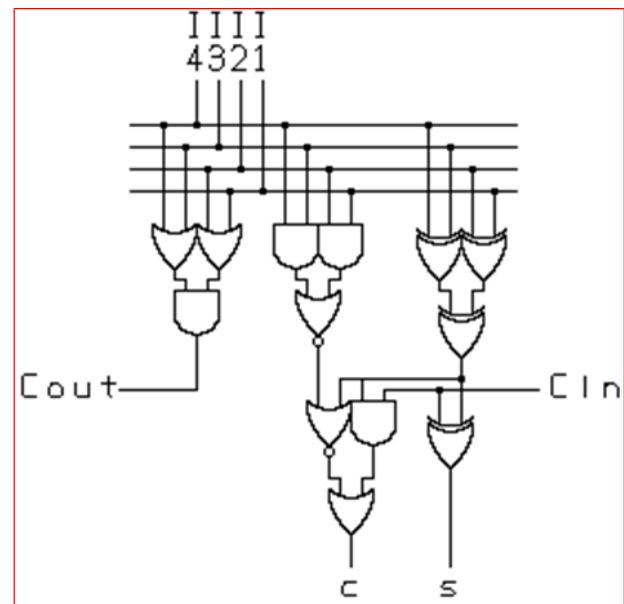
- 基本的加法器单元多采用(2,2)计数器（半加器）、(3,2)计数器、(5,3)计数器和(7,3)计数器等



## 加法器树—4-2压缩加法器

- 4-2压缩器的加法器单元在乘法器的设计中得到了广泛采用。它可以较快地完成中间伪和的产生，而逻辑又比较简单

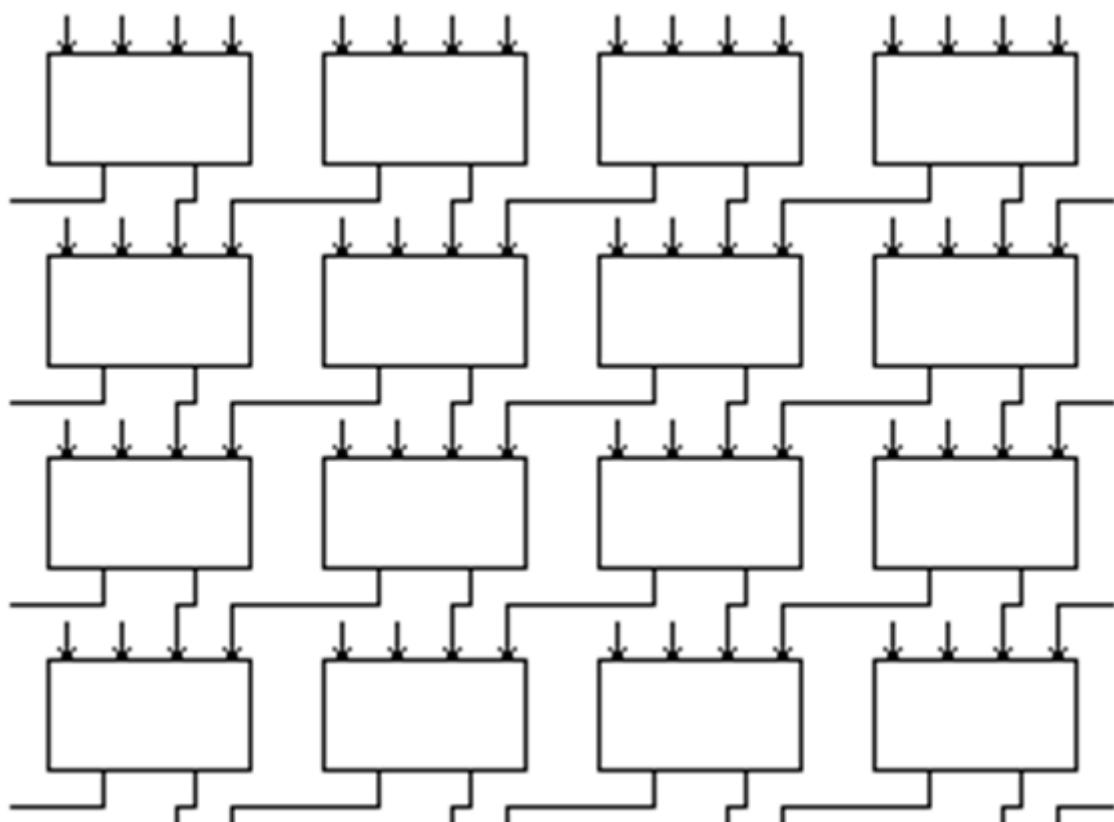
$$\begin{aligned}
 S &= I_1 \oplus I_2 \oplus I_3 \oplus I_4 \oplus C_{in} \\
 C &= (I_1 \oplus I_2 \oplus I_3 \oplus I_4) \cdot C_{in} + \\
 &\quad (I_1 \cdot I_2 + I_3 \cdot I_4) \\
 C_{out} &= (I_1 + I_2) \cdot (I_3 + I_4)
 \end{aligned}$$



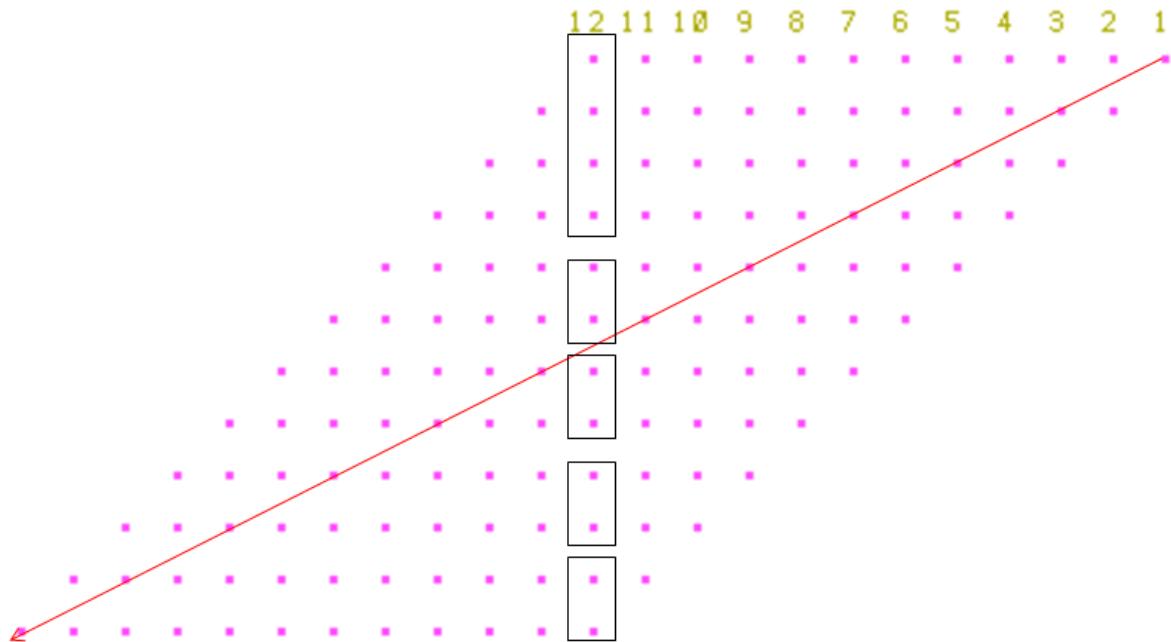
### 加法器树—线性阵列

- 结构最简单规整，速度最慢，需要8级4-2加法器串联

18个部分积：18-4-2-2-2-2-2-2=0，共8级



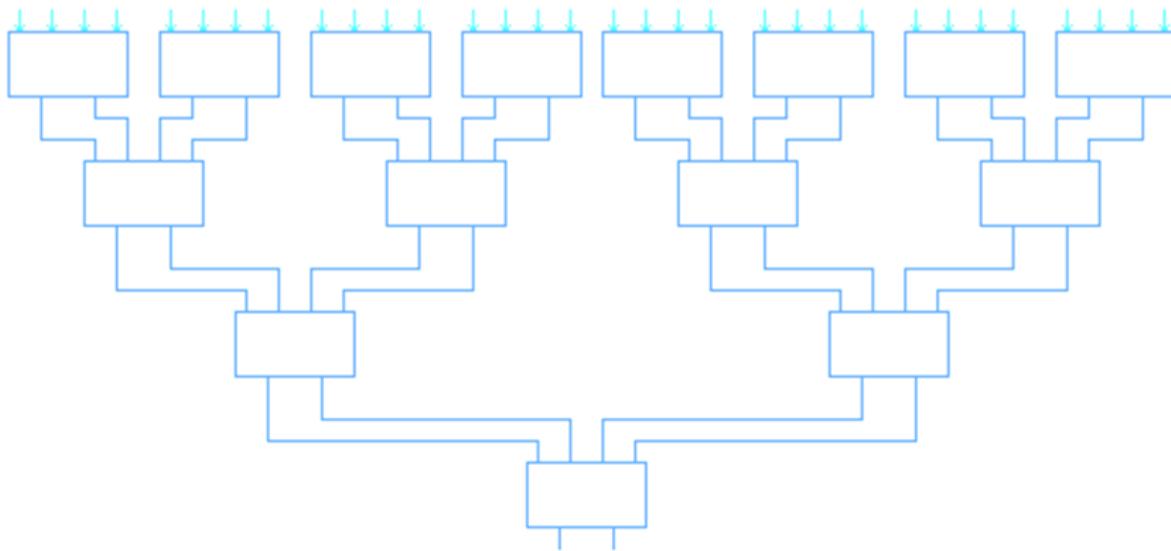
!



## 加法器树—Wallace Tree

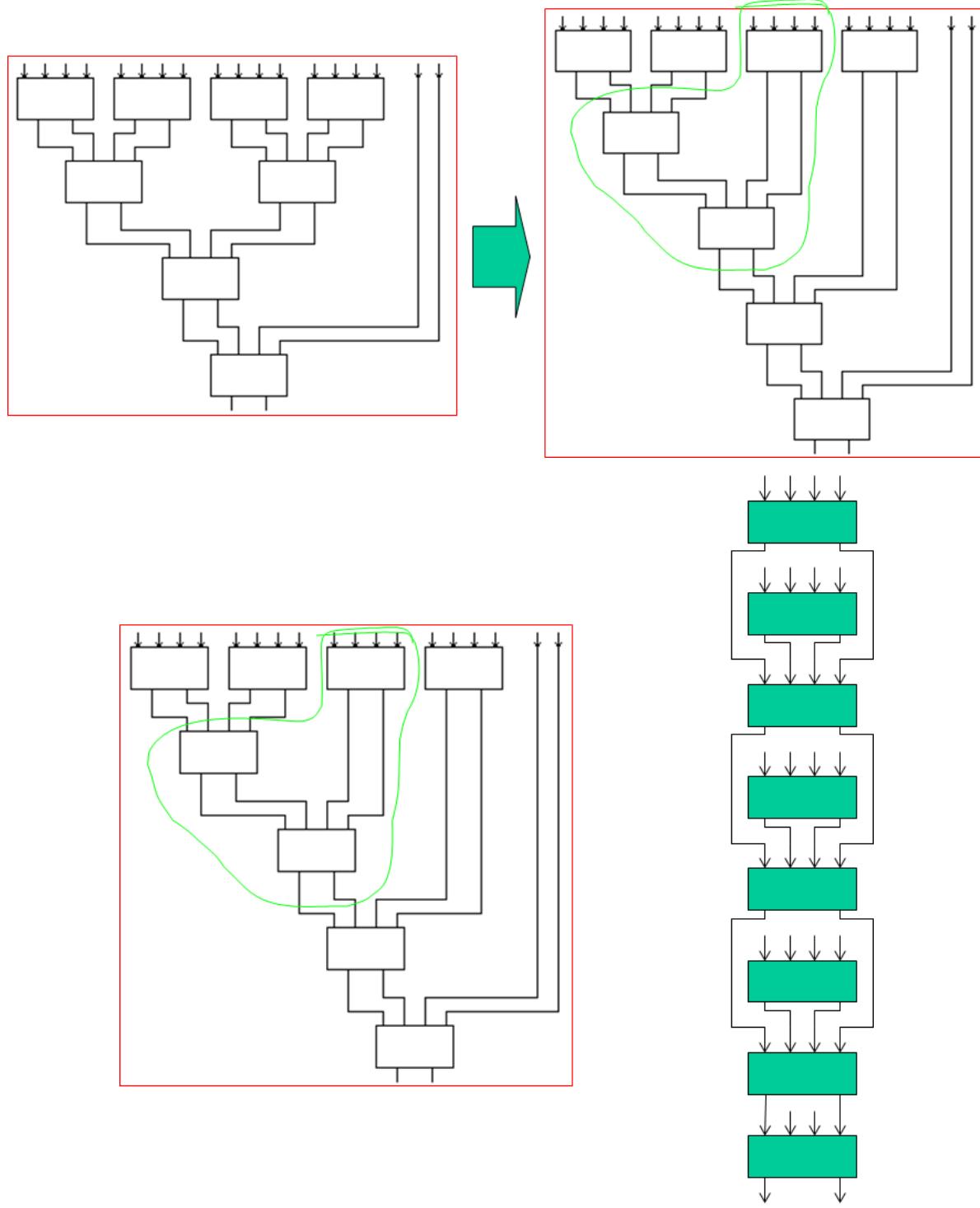
- 速度最快，但结构不规则

$$18 = 4 \times 4 + 2 \rightarrow 8 + 2 = 4 \times 2 + 2 \rightarrow 4 + 2 \rightarrow 2 + 2 \rightarrow 2, \text{ 共4级}$$

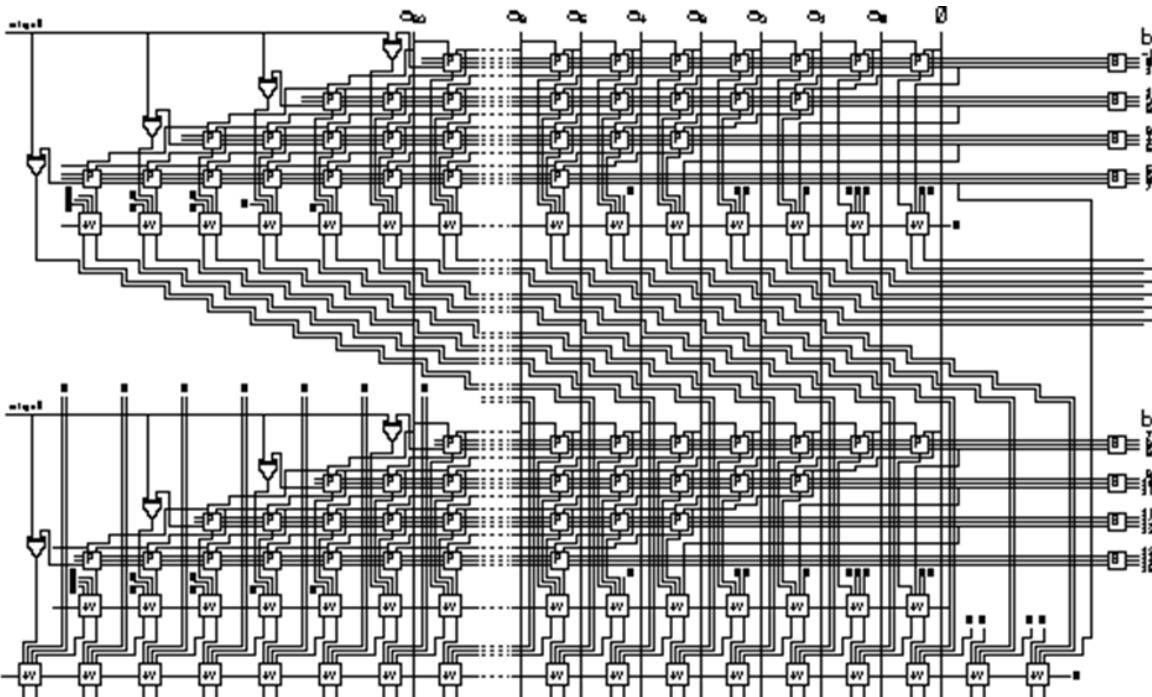


## 加法器树—折衷结构

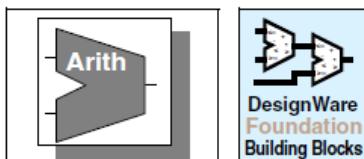
- 32位乘法器只有18个部分积，可以采用折衷结构：在速度和规则性进行折衷  
共5级



乘法器结构示意图



## 二级流水乘法器



DW02\_mult\_2\_stage

Two-Stage Pipelined Multiplier

Version, STAR and Download Information: [IP Directory](#)

### Features and Benefits

- Parameterized word length
- Unsigned and signed (two's-complement) data operation
- Two-stage pipelined architecture
- Automatic pipeline retiming

### Description

DW02\_mult\_2\_stage is a two-stage pipelined multiplier.  
DW02\_mult\_2\_stage multiplies the operand A by B to produce a product (PRODUCT) with a latency of one clock (CLK) cycle.

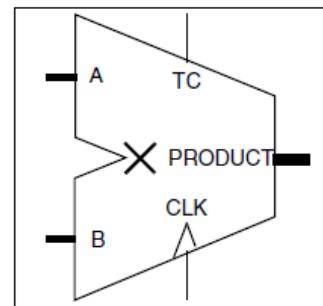
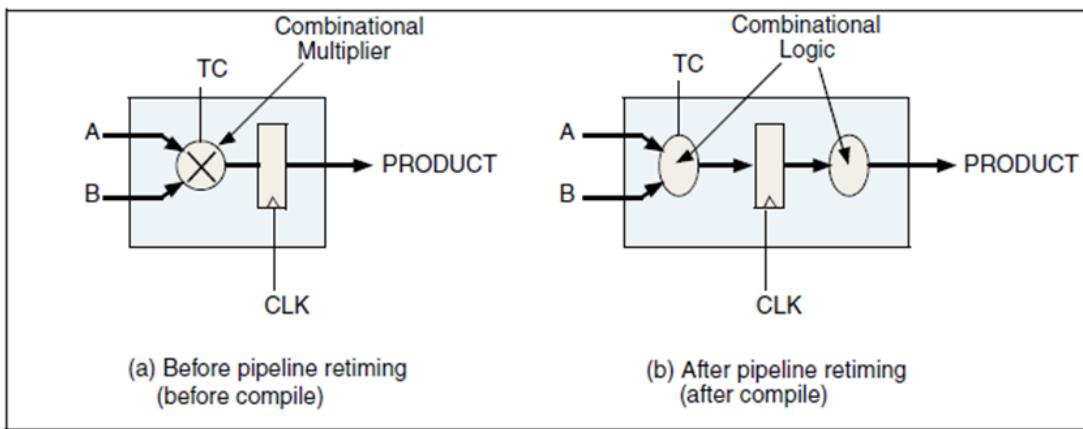


Table 1-1 Pin Description

Pin Name	Width	Direction	Function
A	$A\_width$ bit(s)	Input	Multiplier
B	$B\_width$ bit(s)	Input	Multiplicand
TC	1 bit	Input	Two's complement control 0 = unsigned 1 = signed
CLK	1 bit	Input	Clock
PRODUCT	$A\_width + B\_width$ bit(s)	Output	Product ( $A \times B$ )

Figure 1-1 DW02\_mult\_2\_stage Block Diagram



```

module DW02_mult_2_stage_inst #((
    parameter A_width = 8,
    parameter B_width = 8
)
(
    input [A_width-1 : 0] inst_A,
    input [B_width-1 : 0] inst_B,
    input                   inst_TC,
    input                   inst_CLK,
    output [A_width+B_width-1 : 0] PRODUCT_inst
);
    // Instance of DW02_mult_2_stage
    DW02_mult_2_stage #(A_width, B_width) U1 (
        .A (inst_A),
        .B (inst_B),
        .TC (inst_TC),
        .CLK (inst_CLK),
        .PRODUCT(PRODUCT_inst)
    );
endmodule

```

## 除法器设计

### 串行除法-恢复余数除法

被除数 (Dividend) : 124
除数(Divisor) : 3
商 (Quotient) : 41
余数 (Remainder) : 1

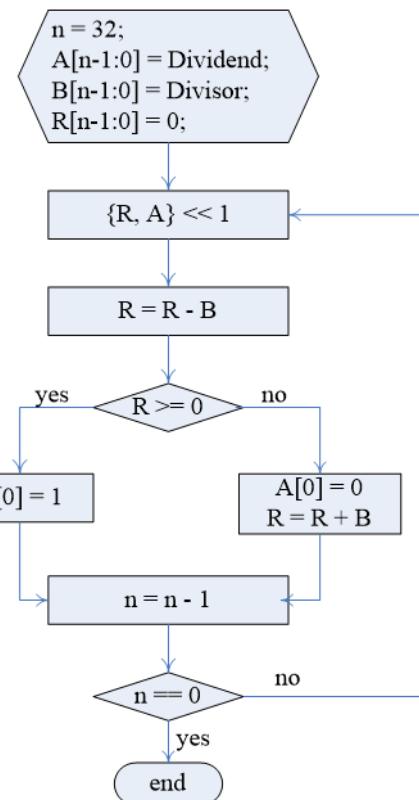
$$\begin{array}{r}
 & 0\ 4\ 1 & \text{Quotient} \\
 \text{Divisor } 3 & \overline{)1\ 2\ 4} & \text{Dividend} \\
 & -0 & \\
 & \underline{1\ 2} & \\
 & -1\ 2 & \\
 & \underline{0\ 0\ 4} & \\
 & -3 & \\
 & \underline{1} & \text{Remainder}
 \end{array}$$

恢复余数除法 (Restoring Division Algorithm) :

1. 将被除数最高位移入，作为被减数R
2.  $R = R - D$ 。如果R为负，恢复被减数R。若R为正，则保留余数。

## 恢复余数除法计算流程

n	操作	A值	结论
4	左移A	A = 00010100	
	A = A - D	A = 11100100	A<0 A[0]=0 A = A + D = 11100100 + 00110000 = 00010100 (restore)
3	左移A	A = 00101000	
	A = A - D	A = 11111000	A<0 A[0]=0 A = A + D = 11111000 + 00110000 = 00101000 (restore)
2	左移A	A = 01010000	
	A = A - D	A = 00100000	A>0 A[0]=1 A = 00100001
1	左移A	A = 01000010	
	A = A - D	A = 00010010	A>0 A[0]=1 A = 00010011
0			Q = 0011(3) R = 0001(1)



## 不恢复余数除法(Non-RDA)

恢复余数法中：

若第*i*次减法计算后的余数小于0，则恢复余数法：

$$R_p = R_i + B;$$

其中B是除数，

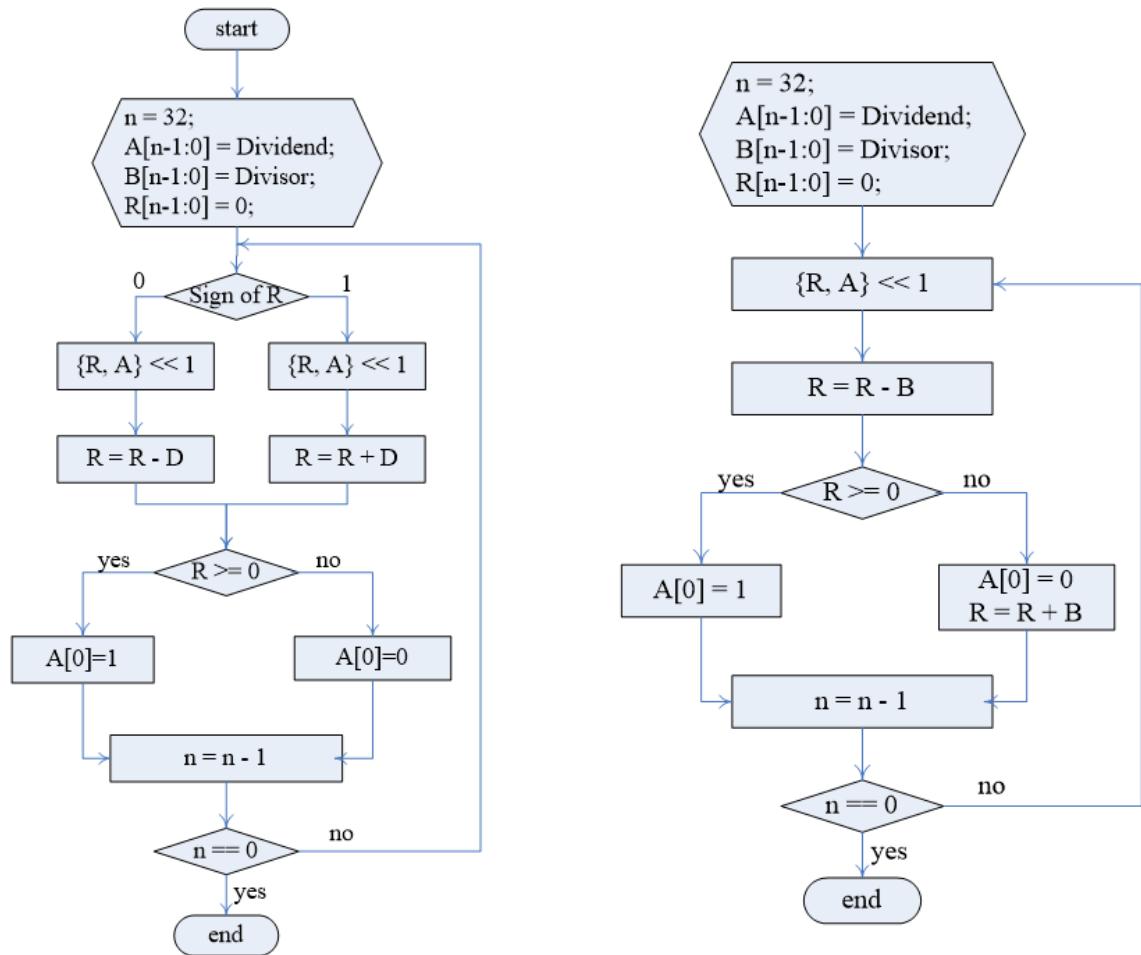
则第*i+1*次的余数为：

$$R_{i+1} = 2R_p - B = 2(R_i + B) - B = 2R_i + B;$$

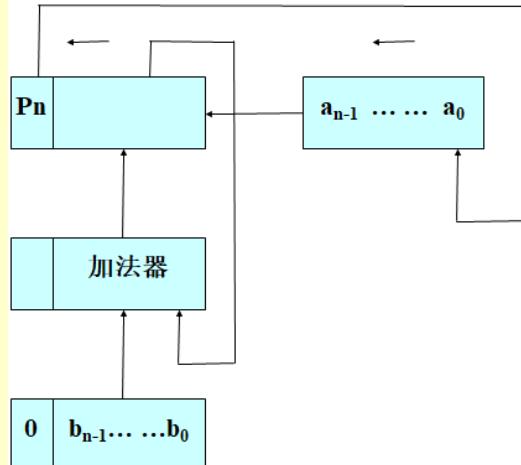
因此不恢复余数除法采用 $R_{i+1} = 2R_i + Y$ 的公式，其结果与恢复余数后左移一位再减B是等效的。

原码交替加减法的通用公式为：

$$R_{i+1} = \begin{cases} 2R_i - Y & \text{上一次余数符号为正} \\ 2R_i + Y & \text{上一次余数符号为负} \end{cases}$$

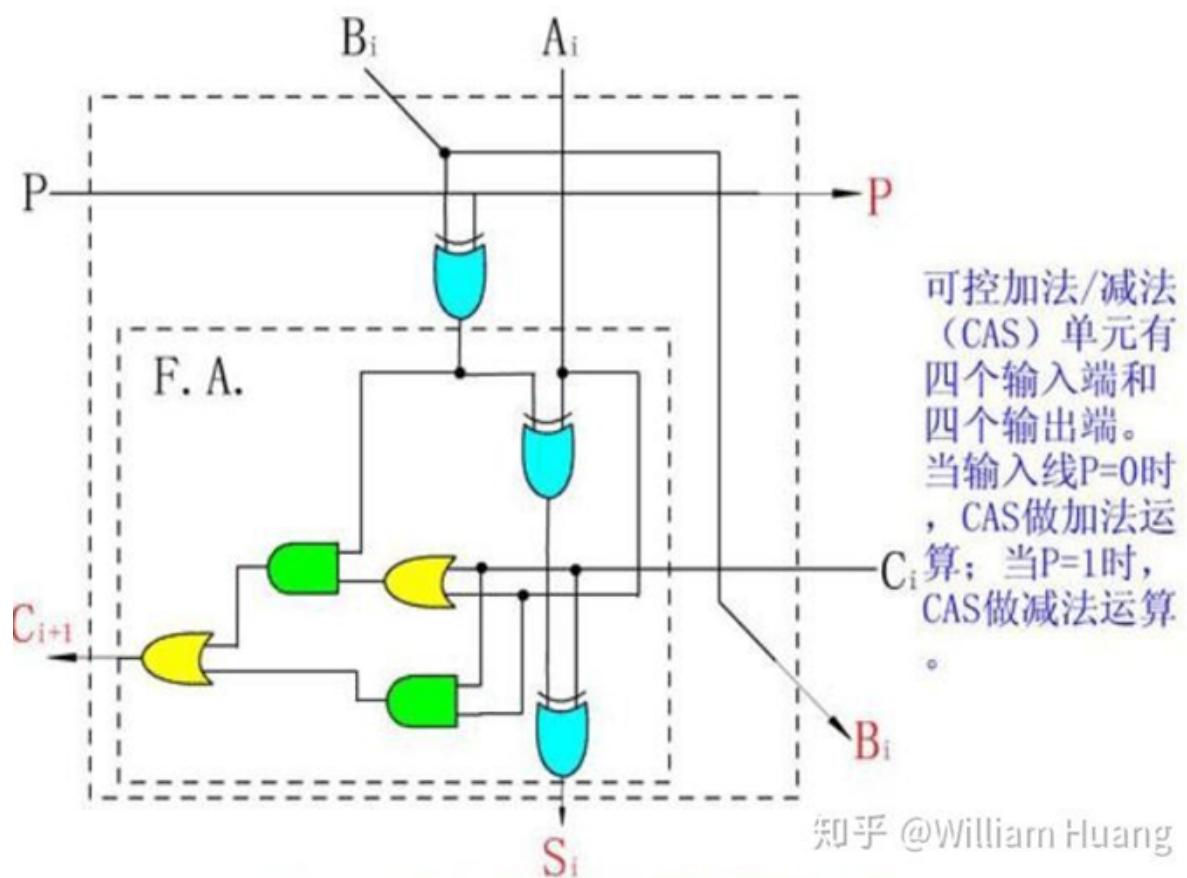


$A=14=1110$ 除 $B=0011$	
$00000 \ 1110$	
$00001 \ 110$	左移一位
$+ \ 11101$	减( $B$ )
<hr/>	
$11110 \ 1100$	$P$ 是负的, 商 “0”
$11101 \ 100$	左移一位
$+ \ 00011$	加( $B$ )
<hr/>	
$00000 \ 1001$	$P$ 是非负的, 商 “1”
$00001 \ 001$	左移一位
$+ \ 11101$	减( $B$ )
<hr/>	
$11110 \ 0010$	$P$ 是负的, 商 “0”
$11100 \ 010$	左移一位
$+ \ 00011$	加( $B$ )
<hr/>	
$11111 \ 0100$	$P$ 是负的, 商 “0”
$+ \ 00011$	余数是负的, 作最后恢复
<hr/>	
$00010$	商是 0100 , 余数是 0010



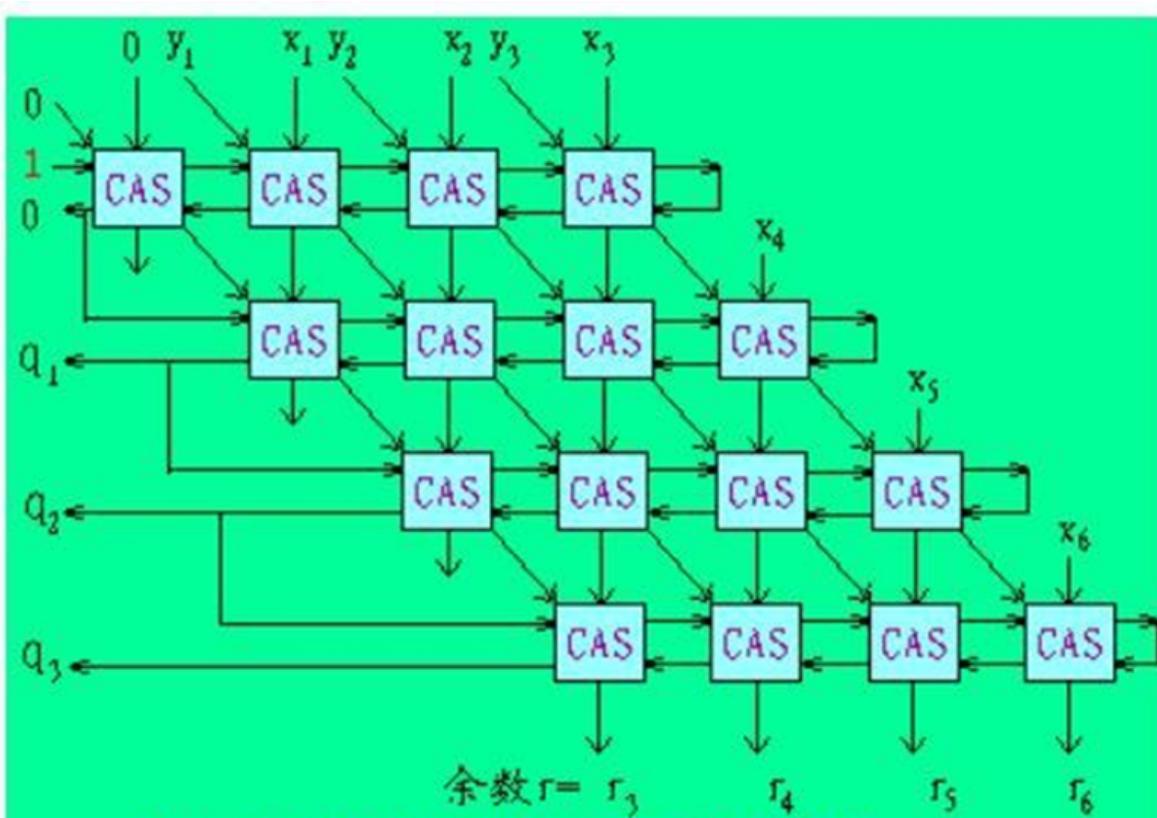
**P:** 余数寄存器  
**a:** 被除数寄存器 (有左移位功能)  
**b:** 除数寄存器

## 脉动阵列除法-加法单元



知乎 @William Huang

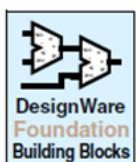
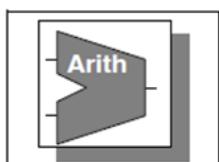
$$\begin{aligned} C_{i+1} &= A_i \cdot B_i + B_i \cdot C_i + A_i \cdot C_i \\ &= (A_i + C_i) \cdot B_i + A_i \cdot C_i \end{aligned}$$



(b) 4位除4位阵列除法器逻辑结构图

知乎 @William Huang

# 并行除法器



## DW\_div

### Combinational Divider

Version, STAR and Download Information: [IP Directory](#)

### Features and Benefits

- Parameterized word lengths
- Unsigned and signed (two's complement) data operation
- Remainder or modulus as second output
- Inferable using a function call

### Description

DW\_div is a combinational integer divider with both quotient and remainder outputs. This component divides the dividend *a* by the divisor *b* to produce the quotient and remainder. Optionally, the remainder output computes the modulus.

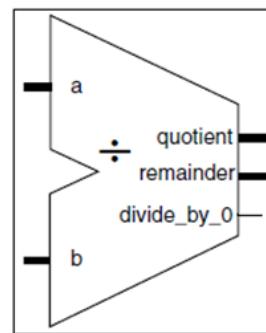


Table 1-1 Pin Description

Pin Name	Width	Direction	Function
<i>a</i>	<i>a_width</i> bit(s)	Input	Dividend
<i>b</i>	<i>b_width</i> bit(s)	Input	Divisor
quotient	<i>a_width</i> bit(s)	Output	Quotient
remainder	<i>b_width</i> bit(s)	Output	Remainder / modulus
divide_by_0	1 bit	Output	Indicates if <i>b</i> equals 0



## DW\_div\_seq

### Sequential Divider

Version, STAR and Download Information: [IP Directory](#)

### Features and Benefits

- Parameterized word length
- Parameterized number of clock cycles
- Unsigned and signed (two's complement) data division
- Registered or un-registered inputs and outputs
- Provides minPower benefits with the DesignWare-LP license.

### Description

DW\_div\_seq is a sequential divider designed for low area, area-time trade-off, or high frequency (small cycle time) applications. DW\_div\_seq is an integer divider with both quotient and remainder outputs.

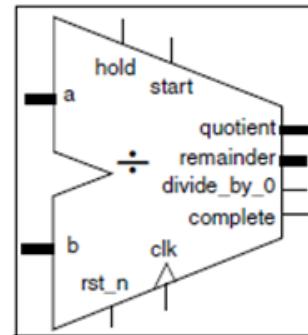


Table 1-1 Pin Description

Pin Name	Width	Direction	Function
clk	1 bit	Input	Clock
rst_n	1 bit	Input	Reset, active low
hold	1 bit	Input	Hold current operation (=1)
start	1 bit	Input	Start operation (=1). A new operation is started by setting start=1 for one clock cycle.
a	$a\_width$ bit(s)	Input	Dividend
b	$b\_width$ bit(s)	Input	Divisor
complete	1 bit	Output	Operation completed (=1)
divide_by_0	1 bit	Output	Indicates if b equals 0
quotient	$a\_width$ bit(s)	Output	Quotient
remainder	$b\_width$ bit(s)	Output	Remainder

## 算术运算的性能

- SMIC 130nm CMOS工艺32位加法器
- 门数是以最小2输入与非门的面积统计的 (面积: 5.0922)

	最慢	6ns	3ns	最快
面积( $\mu\text{m}^2$ )	1071	1468	1660	3792
门数	210	288	326	745
速度(ns)	13.56	6	3	1.23

- SMIC 130nm CMOS工艺32位 乘法器

	最慢	8ns	4ns	最快
面积( $\mu\text{m}^2$ )	14003	15140	28215	36268
门数	2750	2793	5541	7122
速度(ns)	15	7.75	3.83	3.31

# 按位逻辑操作符

```
~  not
&  and
|  or
^  xor
~ ^ xnor
^ ~ xnor
```

按位操作符对矢量中相对应位运算。

```
regb = 4'b1 0 1 0
regc = 4'b1 x 1 0
num = regb & regc = 1 0 1 0 ;
```

位值为x时不一定产生x结果。如#50时的or计算。

**当两个操作数位数不同时，位数少的操作数零扩展到相同位数。**

```
a = 4'b1011;
b = 8'b01010011;
c = a | b; // a零扩展为 8'b00001011
```

```
module bitwise ();
    reg [3: 0] rega, regb, regc;
    reg [3: 0] num;
    initial begin
        rega = 4'b1001;
        regb = 4'b1010;
        regc = 4'b11x0;
    end
    initial begin
        #10 num = rega & 0;           // num = 0000
        #10 num = rega & regb;      // num = 1000
        #10 num = rega | regb;      // num = 1011
        #10 num = regb & regc;      // num = 10x0
        #10 num = regb | regc;      // num = 1110
        #10 num = rega | regc;      // num = ?
        #10 $finish;
    end
endmodule
```

```
module bitwise (
    input wire [3: 0] rega ,
                           regb ,
                           regc ,
    output reg [3: 0] num
);
always @(rega, regb, regc) begin
    num = ~rega;
    num = rega & regb;
    num = rega ^ regb; •      num = rega | regb | regc;
    num = rega | regb & regc;
    num = ( rega | regb) & regc;
```

```
end  
endmodule
```

## 逻辑操作符

```
!    not  
&&   and  
||   or
```

- 逻辑操作符的结果为一位1, 0或x。
- 逻辑操作符只对逻辑值运算。
- 如操作数为全0, 则其逻辑值为false
- 如操作数有一位为1, 则其逻辑值为true
- 若操作数只包含0、x、z, 则逻辑值为x

逻辑反操作符将操作数的逻辑值取反。例如, 若操作数为全0, 则其逻辑值为0, 逻辑反操作值为1。

```
module logical 0;  
  parameter five = 5;  
  reg ans;  
  reg [3: 0] rega, regb, regc;  
  initial  
    begin  
      rega = 4'b0011;           //逻辑值为 “1”  
      regb = 4'b10xz;           //逻辑值为 “1”  
      regc = 4'b0z0x;           //逻辑值为 “x”  
    end  
  initial begin  
    #10 ans = rega && 0;      // ans = 0  
    #10 ans = rega || 0;        // ans = 1  
    #10 ans = rega && five;    // ans = 1  
    #10 ans = regb && rega;    // ans = 1  
    #10 ans = regc || 0;        // ans = x  
    #10 ans = rega || regc;     // ans = ?  
    #10 $finish;  
  end  
endmodule
```

```

module logical (
    input wire [3 : 0] rega ,
                           regb ,
    output reg [3 : 0] ans
);

    always @(rega, regb) begin
        ans = regb && rega;
        ans = regb || rega;
        ans = ! regb
    end
endmodule

```

## 逻辑反与位反的对比

! logical not	逻辑反
$\sim$ bit-wise not	位反

逻辑反的结果为一位1, 0或x。位反的结果与操作数的位数相同

逻辑反操作符将操作数的逻辑值取反。例如, 若操作数为全0, 则其逻辑值为0, 逻辑反操作值为1。

```

module negation();
    reg [3: 0] rega, regb;
    reg [3: 0] bit;
    reg log;

    initial begin
        rega = 4'b1011;
        regb = 4'b0000;
    end

    initial begin
        #10 bit = ~rega; // bit = 0
        #10 bit = ~regb; // bit = 1
        #10 log = ! rega; // log = 0
        #10 log = ! regb; // log = 1
        #10 bit = ! regb; // bit = ?
        #10 $finish;
    end
endmodule

```

## 一元归约操作符

```
& and  
| or  
^ xor  
~ ^ xnor  
^ ~ xnor
```

- 归约操作符的操作数只有一个。
- 对操作数的所有位进行位操作。
- 结果只有一位，可以是0, 1, X。

```
module reduction();
    reg val;
    reg [3: 0] rega, regb, regc;
    initial begin
        rega = 4'b0100;
        regb = 4'b1111;
    end
    initial begin
        #10 val = & rega ;      // val = 0
        #10 val = | rega ;     // val = 1
        #10 val = & regb ;     // val = 1
        #10 val = | regb ;     // val = 1
        #10 val = ^ rega ;     // val = 0
        #10 val = ^ regb ;     // val = 0
        #10 val = ~| rega;     // (nor) val = 0
        #10 val = ~& rega;     // (nand) val = 1
        #10 val = ^regb && &regb; // val = 1
        $finish;
    end
endmodule
```

```
module reduction(
    input wire [3 : 0] rega ,
                           regb ,
    output reg           val
);
    always @(rega, regb) begin
        val = & rega ;
        val = | rega ;
        val = ^ rega ;
        val = ^ rega && & regb;
    end
endmodule
```

## 移位操作种类

```
>> 逻辑右移  
<< 逻辑左移  
>>> 算术右移  
<<< 算术左移
```

第二个操作数（移位位数）是无符号数

若第二个操作数是x或z则结果为x

### 算术右移：高位补符号位

S	a <sub>30</sub>	a <sub>29</sub>	...	a <sub>1</sub>	a <sub>0</sub>
---	-----------------	-----------------	-----	----------------	----------------

S	S	a <sub>30</sub>	...	a <sub>2</sub>	a <sub>1</sub>
---	---	-----------------	-----	----------------	----------------

### 逻辑右移：高位补0

S	a <sub>30</sub>	a <sub>29</sub>	...	a <sub>1</sub>	a <sub>0</sub>
---	-----------------	-----------------	-----	----------------	----------------

0	S	a <sub>30</sub>	...	a <sub>2</sub>	a <sub>1</sub>
---	---	-----------------	-----	----------------	----------------

### 算术左移

### 逻辑左移：低位补0

S	a <sub>30</sub>	a <sub>29</sub>	...	a <sub>1</sub>	a <sub>0</sub>
---	-----------------	-----------------	-----	----------------	----------------

a <sub>30</sub>	a <sub>29</sub>	a <sub>28</sub>	...	a <sub>0</sub>	0
-----------------	-----------------	-----------------	-----	----------------	---

### 循环右移：低位移出的

### 补入高位

S	a <sub>30</sub>	a <sub>29</sub>	...	a <sub>1</sub>	a <sub>0</sub>
---	-----------------	-----------------	-----	----------------	----------------

a <sub>0</sub>	S	a <sub>30</sub>	a <sub>29</sub>	...	a <sub>1</sub>
----------------	---	-----------------	-----------------	-----	----------------

举例：

1000\_1110

算术右移两位： 1110\_0011

逻辑右移两位： 0010\_0011

逻辑左移两位： 0011\_1000

循环右移两位： 1010\_0011

若=左右符号位数不一致，则先全部扩展

建议：表达式左右位数一致

```
module shift ;
    reg [9: 0] num, num1;
    reg [7: 0] rega, regb;
    initial      rega = 8'b00001100;
    initial begin
        #10 num = rega << 5 ;           // num = 01_1000_0000
        #10 regb = rega << 5 ;           // regb =      1000_0000
        #10 num = rega >> 3;            // num = 00_0000_0001
        #10 regb = rega >> 3 ;           // regb =      0000_0001
        #10 num = 10'b11_1111_0000;
        #10 rega = num << 2;             // rega =      1100_0000
        #10 num1 = num << 2;             // num1=11_1100_0000
        #10 rega = num >> 2;             // rega =      1111_1100
        #10 num1 = num >> 2; //num1= ?
    endmodule
```

```

`timescale 1 ns / 1 ns          //Logic shift
module shifter ();
    reg      [7 : 0] rega,
              regb;
    reg clk = 0;
    always #5 clk = !clk;
    initial begin
        rega = 8'b1100_1010;
        @(posedge clk) regb = rega >> 1;
        @(posedge clk) regb = rega >>> 1;
        @(posedge clk) regb = rega << 2;
        @(posedge clk) regb = rega <<< 2;
        @(posedge clk) $finish;
    end
endmodule

```

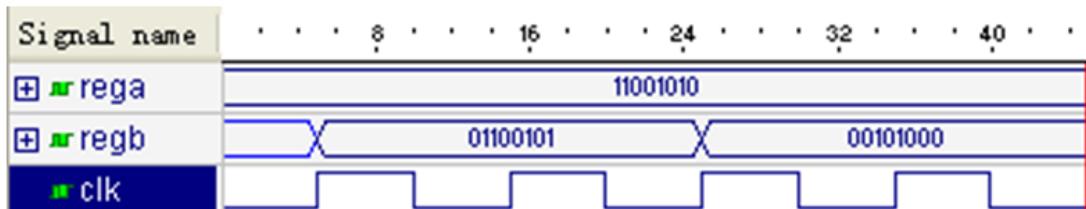
```

`timescale 1 ns / 1 ns          //Arith shift
module shifter ();
    reg signed [7 : 0] rega,
                  regb;
    reg clk = 0;
    always #5 clk = !clk;
    initial begin
        rega = 8'b1100_1010;
        @(posedge clk) regb = rega >> 1;
        @(posedge clk) regb = rega >>> 1;
        @(posedge clk) regb = rega << 2;
        @(posedge clk) regb = rega <<< 2;
        @(posedge clk) $finish;
    end
endmodule

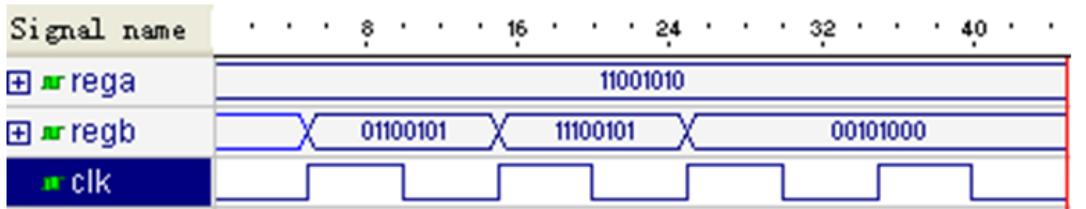
```

- 算术右移必须等式左右两边的变量都是有符号数，如果都是无符号数相当于逻辑右移，尽量不要在无符号数使用算术右移
- 移位的电路实现

## Logic shift



## Arith shift

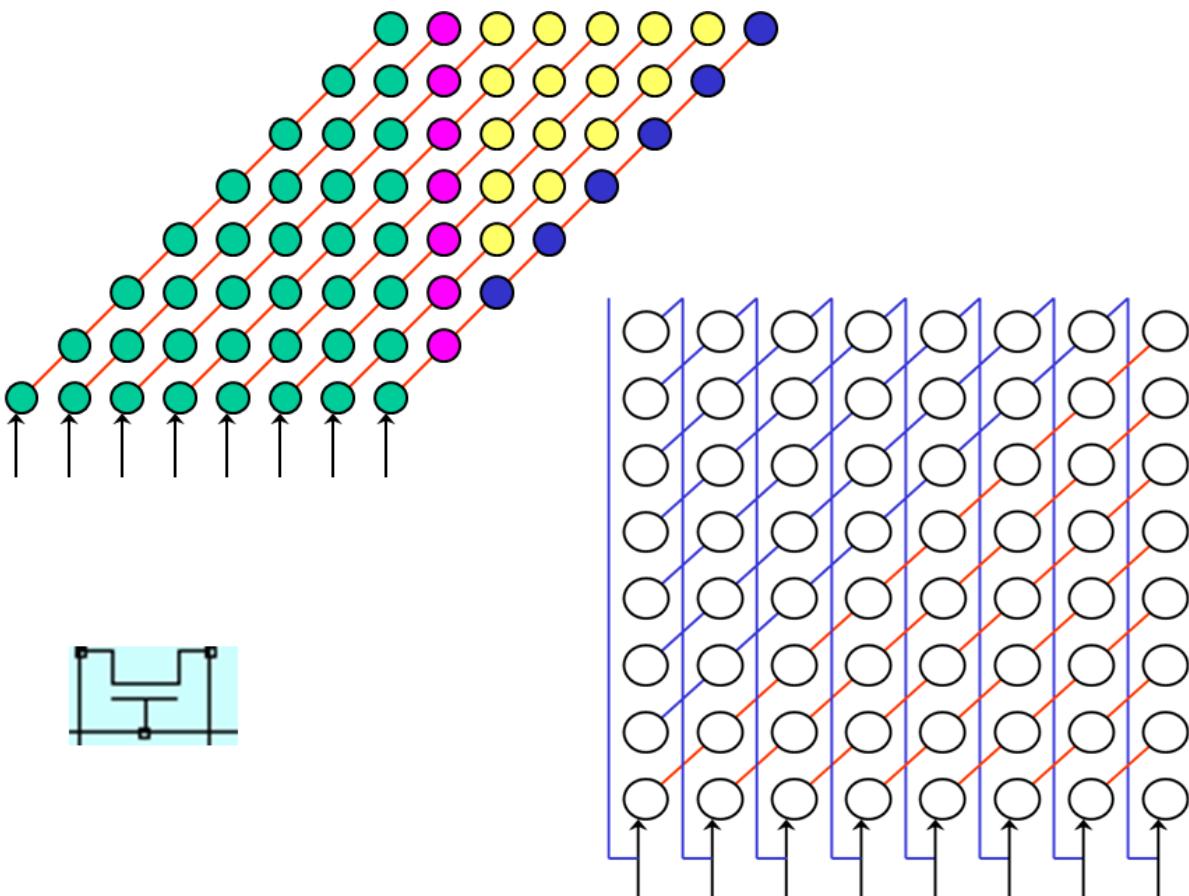


```

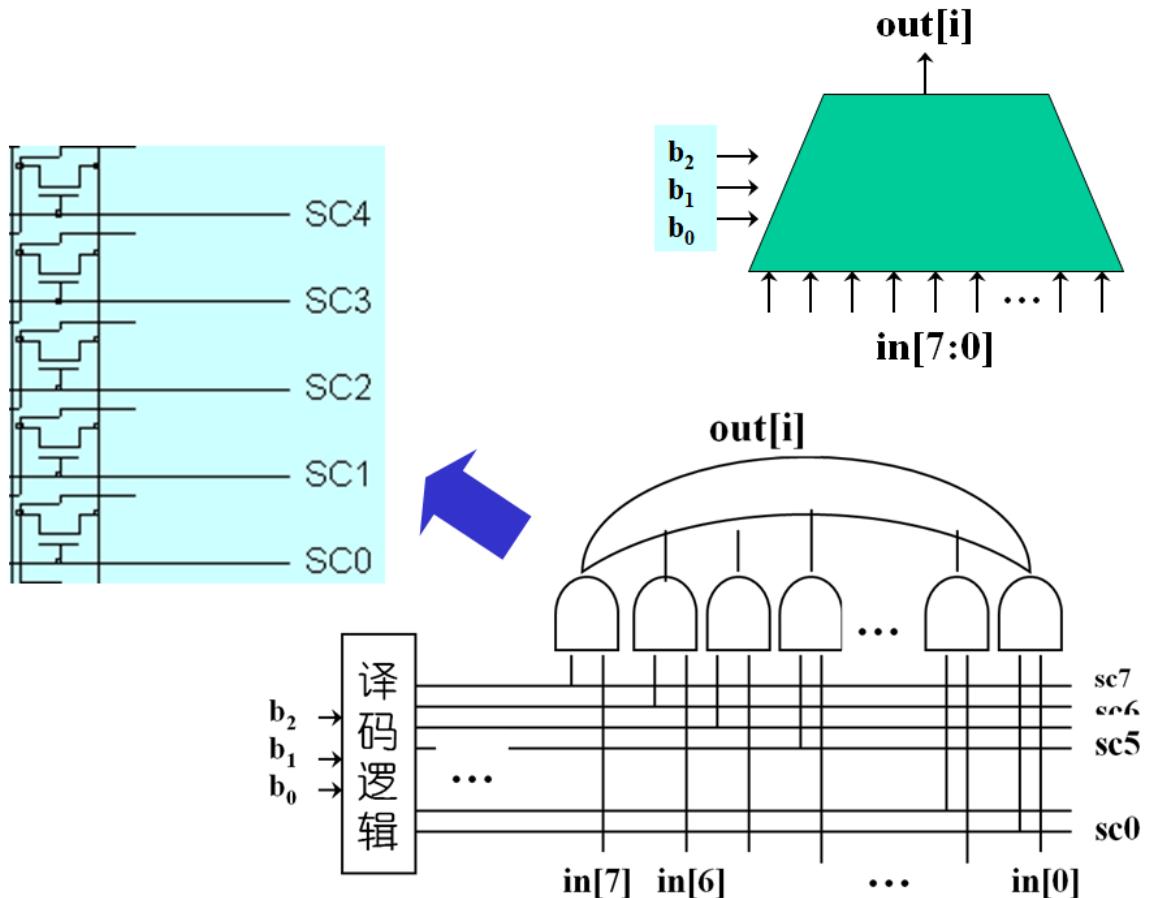
module shift (
    input wire [7 : 0] rega ,
    output reg [7 : 0] num1 ,
    input wire [2: 0] index
);
    always @(rega, regb, index) begin
        num1 = rega << 2;
        num1 = rega >> 2 ;
        num1 = rega >>> 2 ;
        num1 = rega <<< 2 ;
        num2 = rega >> index ;
    end
endmodule

```

## 循环右移

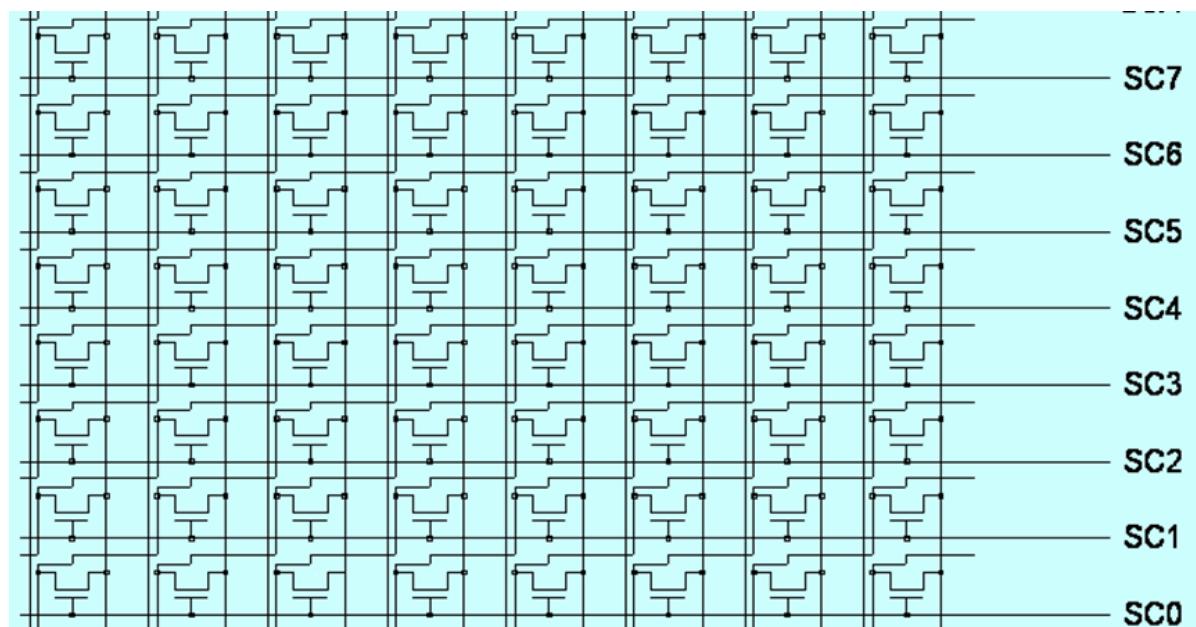


## BS的实现（全译码）



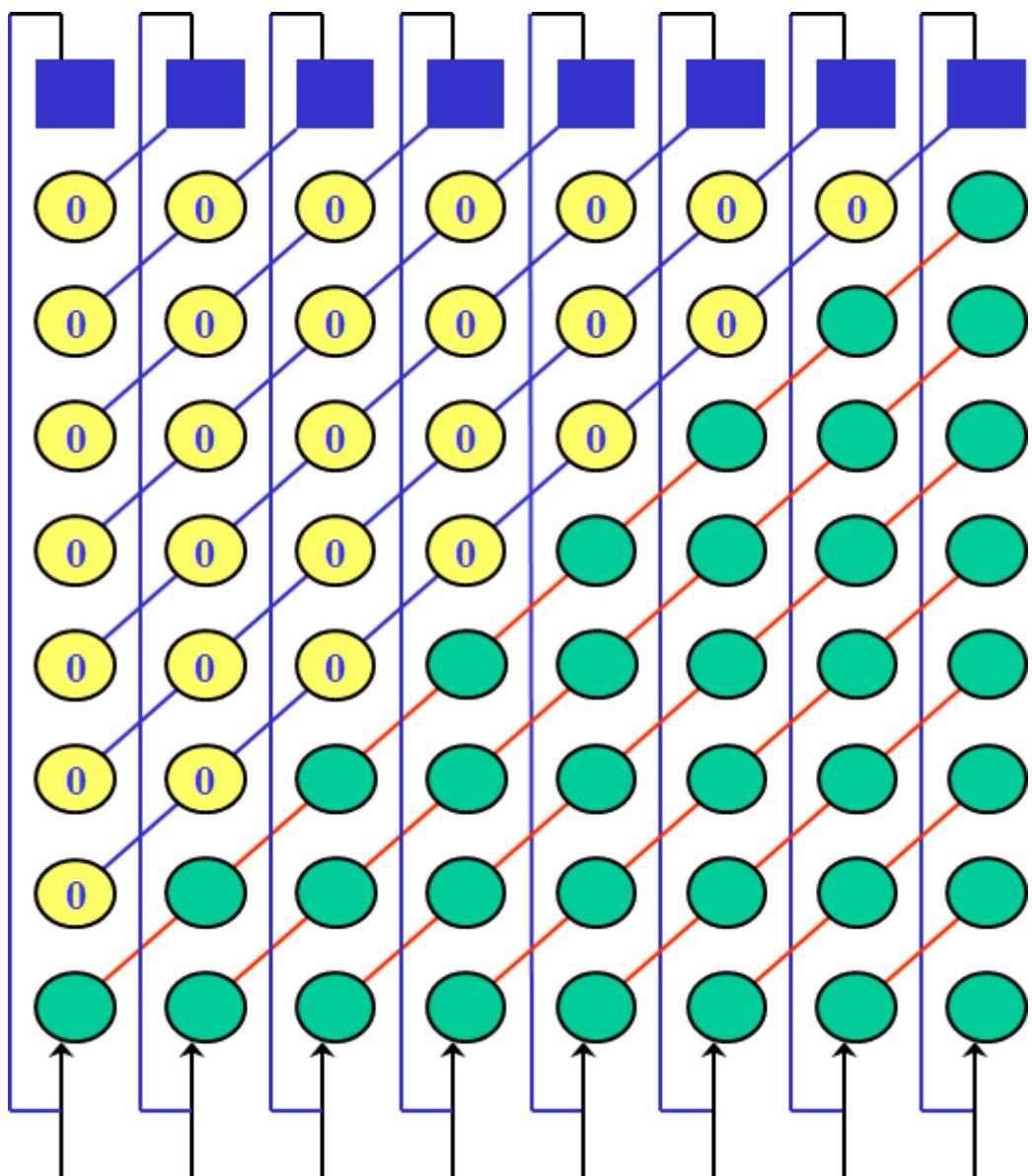
- 1位: 8选1
- 8位: 8个8选1
- 用管逻辑电路实现

### 循环右移

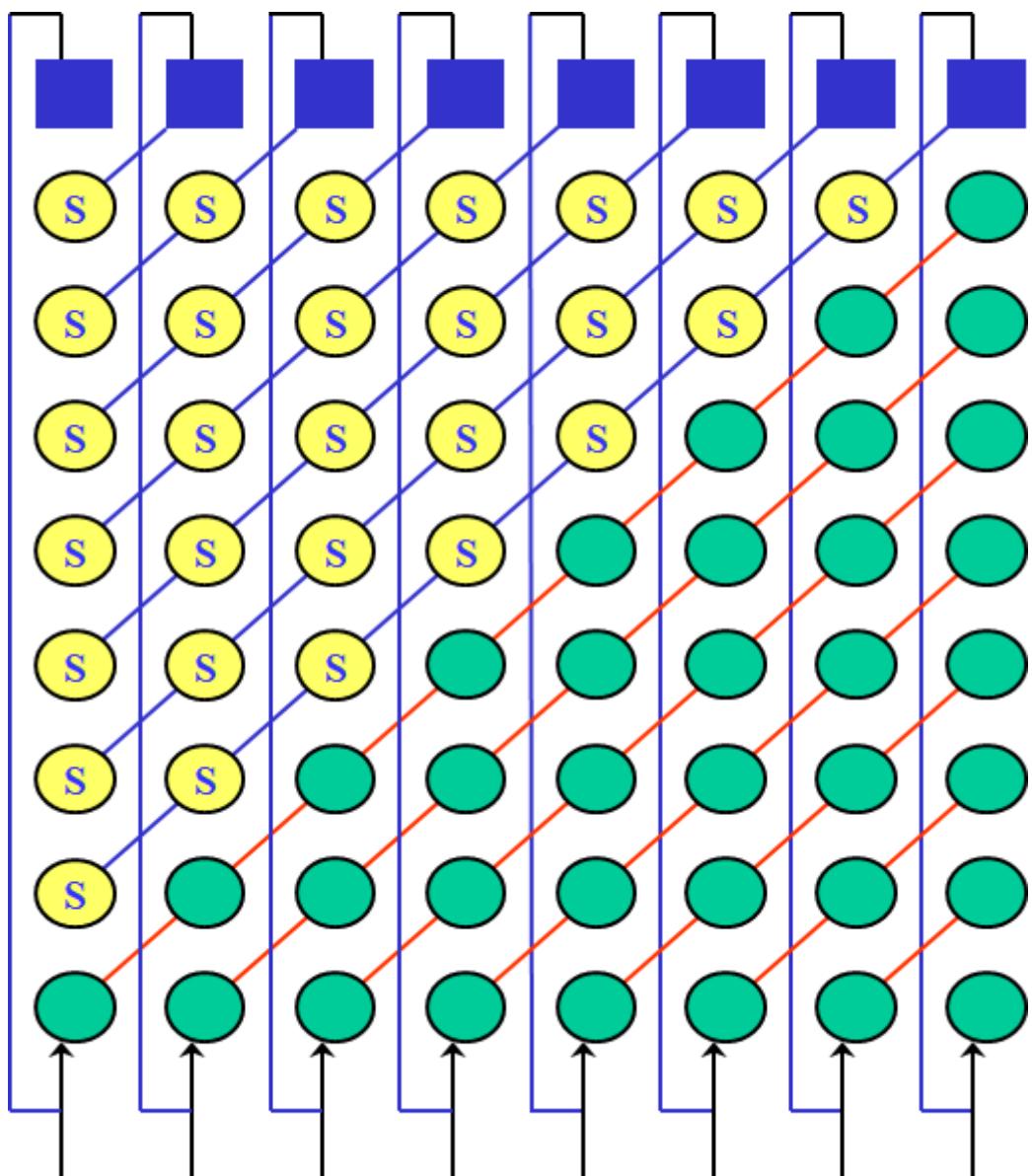


- CMOS电路本质是对电容充放电的过程

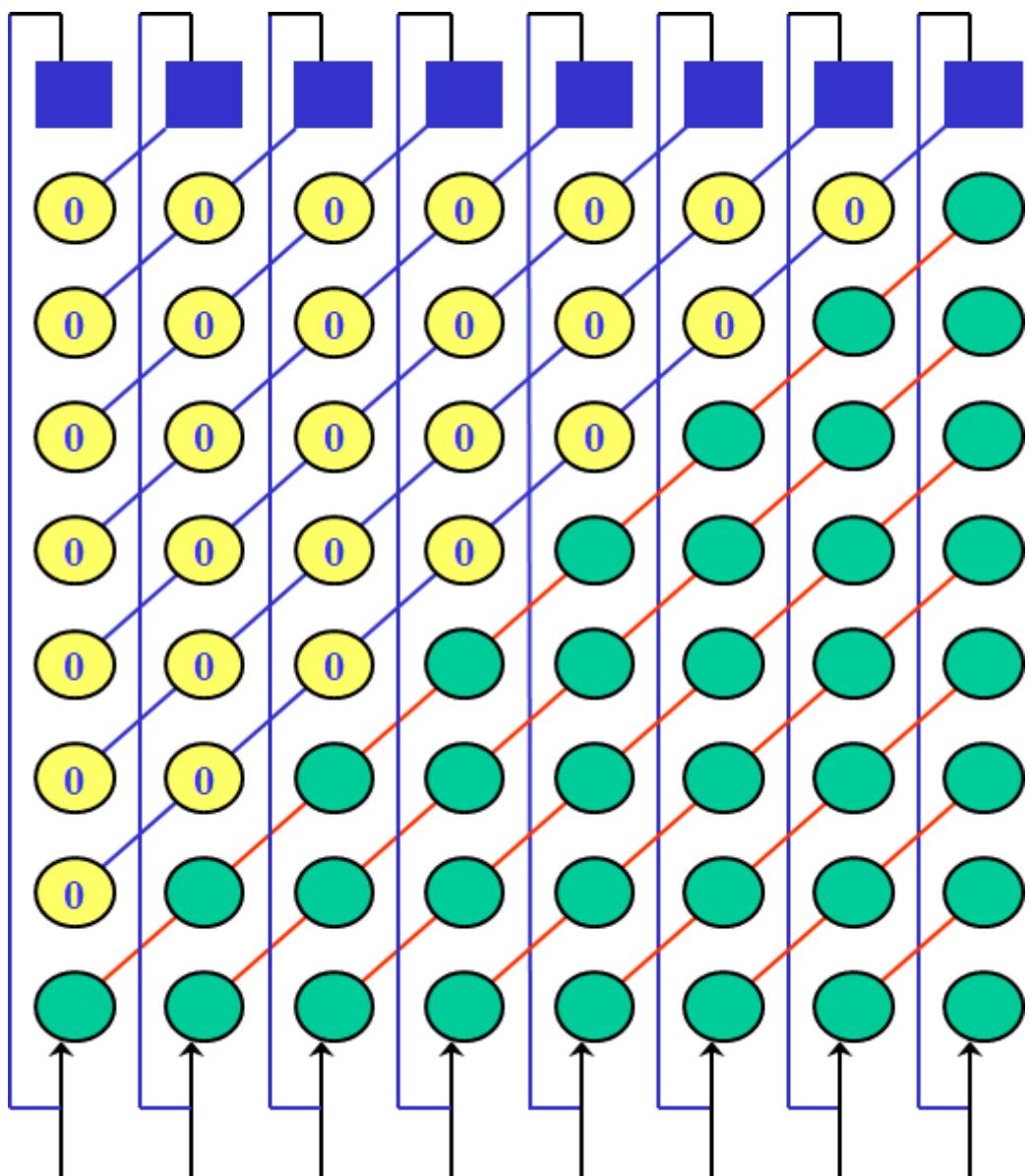
### 逻辑右移



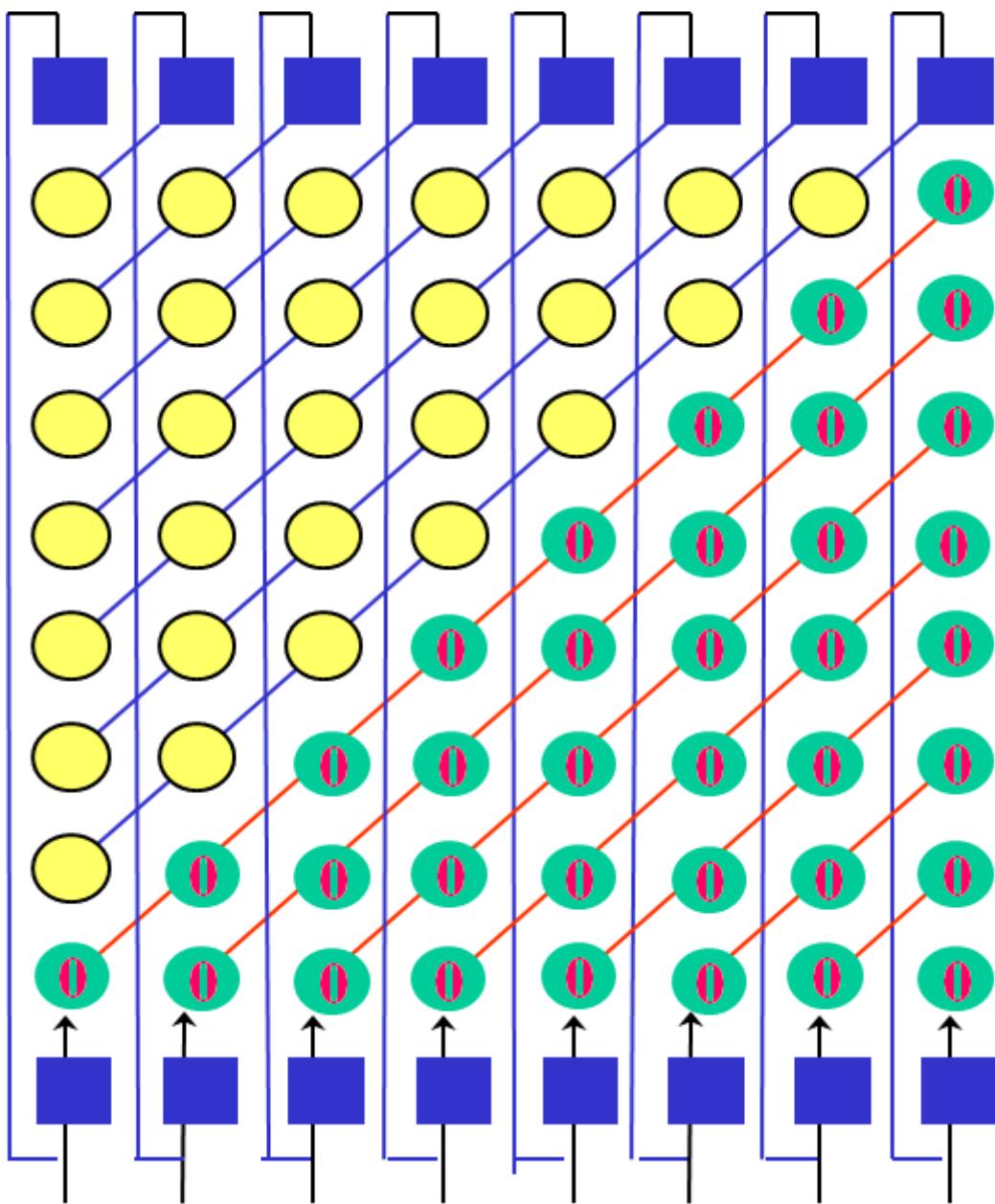
算术右移



右移控制



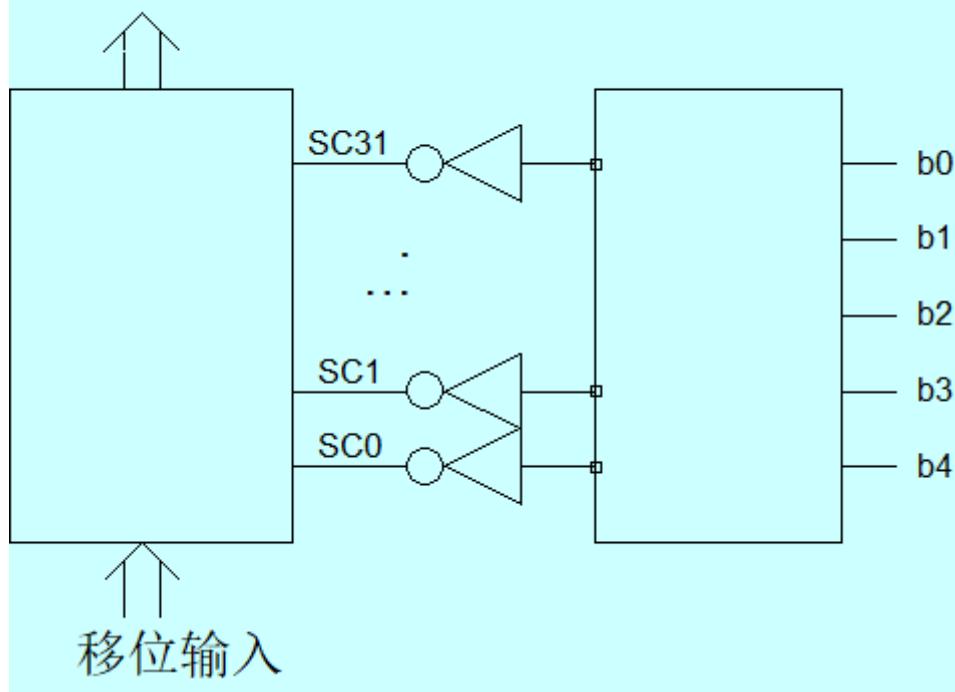
逻辑左移



## 全译码方式

- 对表示移位次数的二进制位进行完全译码，分别给出各种移位的单独控制线。
- 对于32位字长来说，移位部分有32根控制线SC31~ SC0分别控制移31~0位时的操作

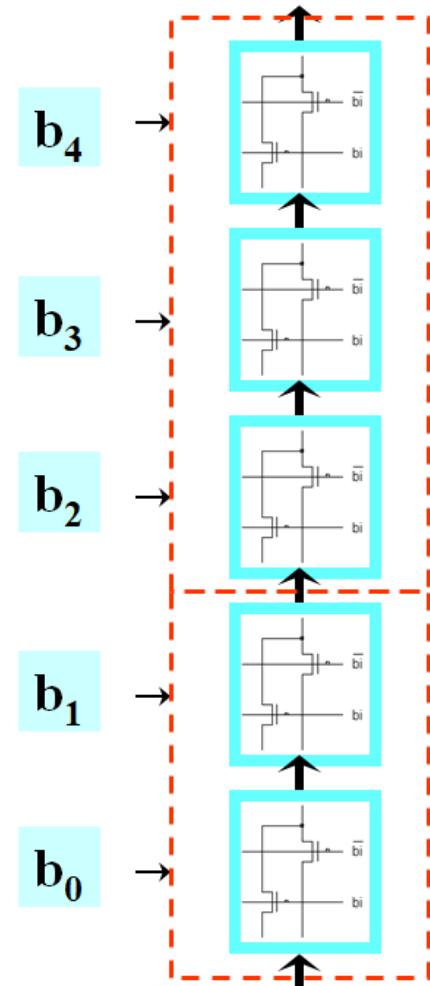
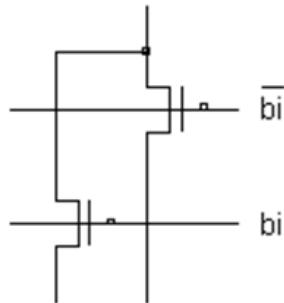
## 移位输出



- 桶形移位寄存器，开关阵列规则，电路最规则，最慢

## BS的实现（全编码）

$b_4 = 1$  移16位 =0 不移位  
 $b_3 = 1$  移 8 位 =0 不移位  
 $b_2 = 1$  移 4 位 =0 不移位  
 $b_1 = 1$  移 2 位 =0 不移位  
 $b_0 = 1$  移 1 位 =0 不移位

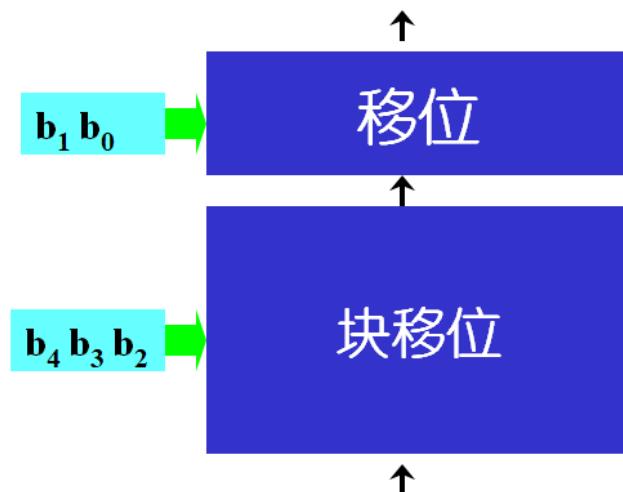


- 5个叠加就是要移位位数，电路最不规则，最快

## BS的实现 (部分译码)

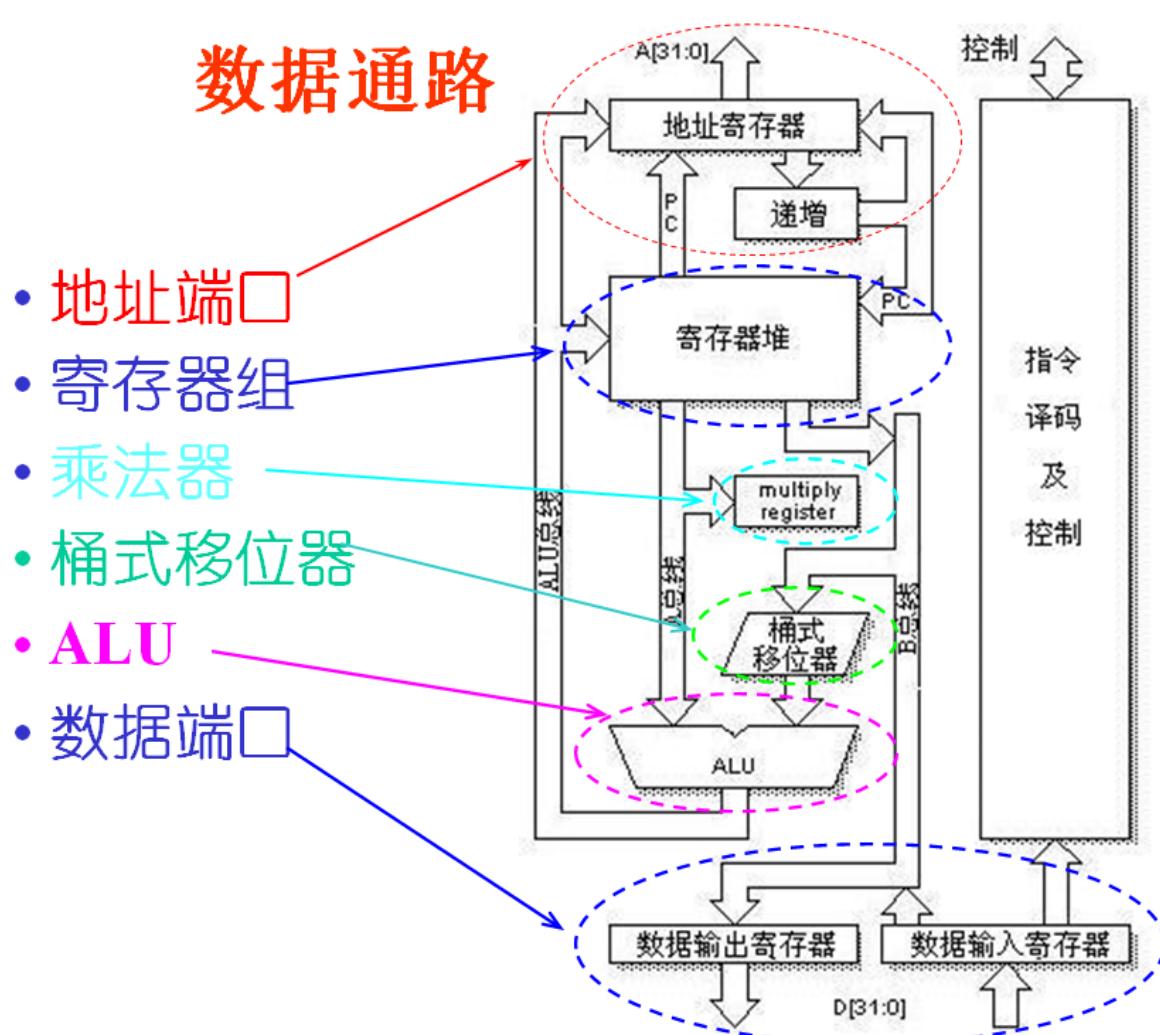
$b_4 b_3 b_2 = 000$	移 0位
001	移 4位
010	移 8位
011	移12位
100	移16位
101	移20位
110	移24位
111	移28位

$b_1 b_0 = 00$	移 0 位
01	移 1 位
10	移 2 位
11	移 3 位



- 折衷： $4 \times 32$  和  $8 \times 32$  的开关阵列组成，快位移每次固定移动四位

## 数据通路



## 桶式移位器BS(Barrel Shifter)

- 是高速微处理器中的常用部件
- 能在单周期内完成多种方式、各种位数的移位操作。
- 用于实现移位指令、浮点计算中的小数点对齐等。

## 关系操作符

```
> 大于  
< 小于  
>= 大于等于  
<= 小于等于
```

其结果是

1'b1

1'b0

或 1'bx。

什么时候  
出现x值？

```
module relational0;  
    reg [3: 0] rega, regb, regc;  
    reg val;  
    initial begin  
        rega = 4'b0011;  
        regb = 4'b1010;  
        regc = 4'b0x10;  
    end  
    initial begin  
        #10 val = regc > rega ; // val = x  
        #10 val = regb < rega ; // val = 0  
        #10 val = regb >= rega ; // val = 1  
        #10 val = regb > regc ; // val = 1  
        #10 $finish;  
    end  
endmodule
```

rega和regc的  
关系取决于x

无论x为何值，  
regb>regc

```
module relational0  
    input wire [3: 0] rega ,  
           regb ,  
    output reg          val  
);  
  
    always @(rega, regb) begin  
        val = rega > regb ; //符号位0, 且不是全零  
        val = rega < regb ; //符号位1  
        val = rega == regb ; //全0  
        val = rega >= regb ; //符号位0  
        val = rega <= regb ; //符号位1, 或全零  
    end  
endmodule
```

- 判断大小的电路是减法器来实现
- 上述代码的实现电路需要5位来实现，增加一位符号位

## 相等操作符

**==** 逻辑等  
**!=** 逻辑不等

其结果是1'b1、1'b0或1'bx。

如果左边及右边为确定值并且相等，则结果为1。

如果左边及右边为确定值并且不相等，则结果为0。

如果左边及右边有值不能确定的位，但值确定的位相等，则结果为x。

!=的结果与==相反

值确定是指所有的位为0或1。  
不确定值是有值为x或z的位。

```
module equalities1();
    reg [3: 0] rega, regb, regc;
    reg val;
initial begin
    rega = 4'b0011;
    regb = 4'b1010;
    regc = 4'b1x10;
end
initial begin
#10 val= rega == regb ; // val= 0
#10 val= rega != regc; // val= 1
#10 val= regb != regc; // val= x
#10 val= regc == regc; // val= x
#10 $finish;
end
endmodule
```

- == 的电路规模小

## 相同操作符

**====** 相同(case等)  
**!==** 不相同(case不等)

其结果是1'b1、1'b0。

如果左边及右边的值相同（包括x、z），则结果为1。

如果左边及右边的值不相同，则结果为0。

!==的结果与 === 相反

综合工具不支持

```
module equalities2();
    reg [3: 0] rega, regb, regc;
    reg val;
initial begin
    rega = 4'b0011;
    regb = 4'b1010;
    regc = 4'b1x10;
end
initial begin
#10 val= rega === regb ; // val= 0
#10 val= rega !== regc; // val= 1
#10 val= regb === regc; // val= 0
#10 val= regc === regc; // val= 1
#10 $finish;
end
endmodule
```

- ==== 字母比较
- ==== 唯一一个不能综合的操作符



赋值操作符，将等式右边表达式的值拷贝到左边。

== 逻辑等				
==	0	1	x	z
0	1	0	x	x
1	0	1	x	x
x	x	x	x	x
z	x	x	x	x

```
a = 2'b1x;
b = 2'b1x;
if (a == b)
    $display(" a is equal to b");
else
    $display(" a is not equal to b");
```

注意逻辑等与

case等的差别

2'b1x==2'b0x

值为0，因为不相等

2'b1x==2'b1x

值为x，因为可能不相等，也可能相等

==== case等				
==	0	1	x	z
0	1	0	0	0
1	0	1	0	0
x	0	0	1	0
z	0	0	0	1

```
a = 2'b1x;
b = 2'b1x;
if (a === b)
    $display(" a is identical to b");
else
    $display(" a is not identical to b");
```

2'b1x==2'b0x

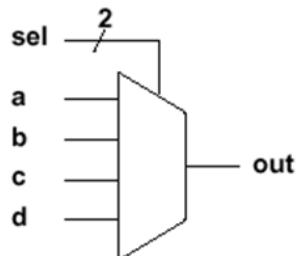
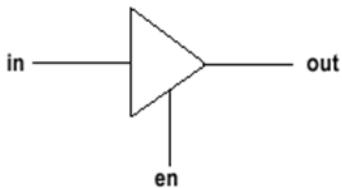
值为0，因为不相同

2'b1x==2'b1x

值为1，因为相同

## 条件操作符

? : 条件操作符



module likebufif(

input wire in,  
en,  
output wire out  
);

assign out = en ? in : 1'bz;

endmodule

module like4to1(

input wire a, b, c, d,  
input wire [1:0] sel,  
output wire out);  
assign out = sel == 2'b00 ? a :  
sel == 2'b01 ? b :  
sel == 2'b10 ? c : d;

endmodule

如果条件值为x或z，则结果可能为x或z

- 三态门实现
- 数据总线上后来用二选一电路替代三态门
- 三态门给双向数据端口建模

条件操作符的语法为：

```
<LHS> = <condition> ? <true_expression> : <false_expression>
```

其意思是：if condition为真，则 LHS=true\_expression, 否则 LHS = false\_expression

每个条件操作符必须有三个参数，缺少任何一个都会产生错误。

```
register = condition ? true_value : false_value;
```

上式中，若condition为真则register等于true\_value；若condition为假则register等于false\_value。一个很有意思的地方是，如果条件值不确定，且true\_value和false\_value不相等，则

```
例如: assign out = (sel == 0) ? a : b;
```

若sel为0则out =a；若sel为1则out = b。如果sel为x或z，若a = b =0，则out = 0；若a≠b，则out值不确定。

## 级联操作符



### 级联或称为拼接

可以从不同的矢量中选择位  
并用它们组成一个新的矢量。  
用于位的重组和矢量构造

在级联和复制时，必须位数确定，否则将产生错误。

下面是类似错误的例子：

```
a[7:0] = {4{`b10}};  
b[7:0] = {2{5}};  
c[3:0] = {3`b011, `b0};
```

级联时不限定操作数的数目。  
在级联符号{}中，用逗号将  
操作数分开。例如：

```
{A, B, C, D}
```

```
module concatenation;  
reg [7: 0] rega, regb, regc, regd;  
reg [7: 0] new;  
initial begin  
rega = 8'b0000_0011;  
regb = 8'b0000_0100;  
regc = 8'b0001_1000;  
regd = 8'b1110_0000;  
end  
initial begin  
#10 new = { regc[4:3], regd[7:5],  
            regb[2], rega[1:0] };  
// new = 8'b11111111  
#20 $finish;  
end  
endmodule
```

- 必须指定位数，不能缺省！如'b0

## 复制操作符

{ } 复制

复制一个变量或在{}中的值

前两个{符号之间的正整数指定复制次数。

```
module replicate();
    reg [3: 0] rega;
    reg [1: 0] regb, regc;
    reg [7: 0] bus;
initial begin
    rega = 4'b1001;
    regb = 2'b11;
    regc = 2'b00;
end
initial fork
    #10 bus <= {4{ regb}}; // bus= 8'b11111111
    // regb 复制4次。
    #20 bus <= { {2{ regb}}, {2{ regc}} };
    // regc 和 regb 复制后的结果级联
    // bus = 8'b11110000
    #30 bus <= { {4{ rega[1]}}, rega };
    // bus = 8'b00001001
    #40 $finish;
join
endmodule
```

复制

## 拼接与复制

```
module replicate (
    input wire [1 : 0] rega ,
                           regb ,
    output reg [3 : 0] out   ,
                           bus
);

always @(rega, regb) begin
    bus = {2{ regb}};
    out = {rega, regb};
end
endmodule
```

{ } 复制

{ } 级联 或称为拼接

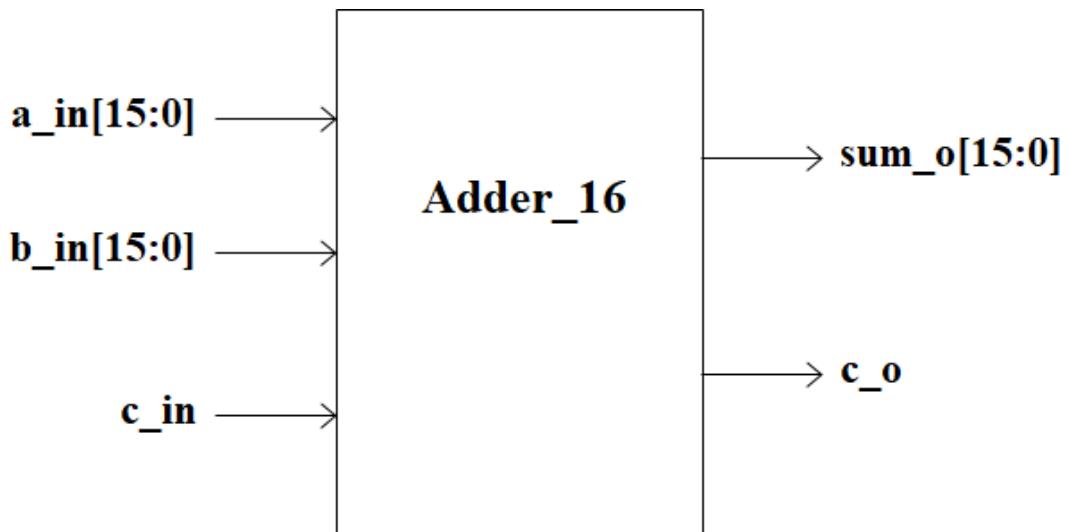
## 作业1

请使用verilog描述一个16位CSA加法器adder\_16，如下图所示。其中：

a\_in和b\_in是输入数据，sum\_o是和输出信号，

c\_in是进位输入，c\_o是进位输出信号。

要求：4位一组，组内使用CLA进位链。



仿真工具：

- Modesim
- ActiveHDL
- iverilog

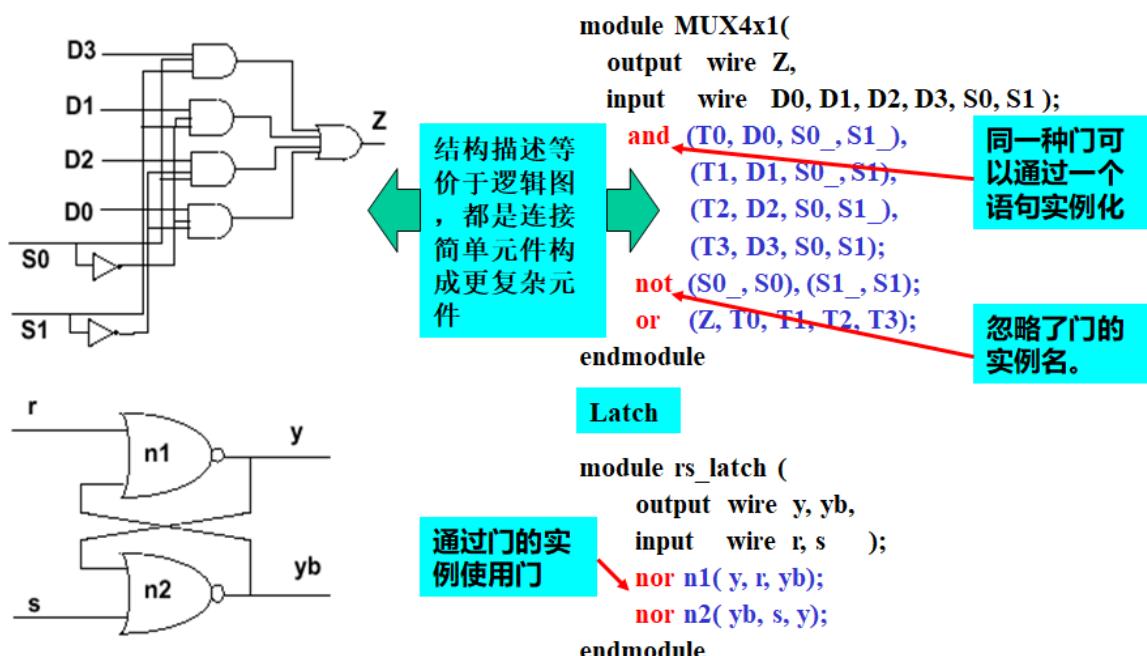
## Verilog语言的结构级描述

### 术语及定义 (terms and definitions)

- 结构描述：用门及门的连接描述器件的功能
- primitives(基本单元)：Verilog语言已定义的具有简单逻辑功能的功能模型(models)

### 结构描述

- Verilog结构描述表示一个逻辑图
- 结构描述用已有的元件构造。



- 结构描述等价于逻辑图。它们都是连接简单元件来构成更为复杂的元件。Verilog使用其连接特性完成简单元件的连接。
- 在描述中使用元件时，通过建立这些元件的实例来完成。
- 上面的例子中MUX是没有反馈的组合电路，使用中间或内部信号将门连接起来。描述中忽略了门的实例名，并且同一种门的所有实例可以在一个语句中实例化。
- 上面的锁存器(latch)是一个时序元件，其输出反馈到输入上。它没有使用任何内部信号。它使用了实例名并且对两个nor门使用了分开的实例化语句。

## Verilog基本单元 (primitives)

- Verilog基本单元提供基本的逻辑功能，这些基本单元是预定义的，不需要用户定义。
- 大多数ASIC和FPGA元件库是用这些基本单元开发的。基本单元库是自下而上的设计方法的一部分。

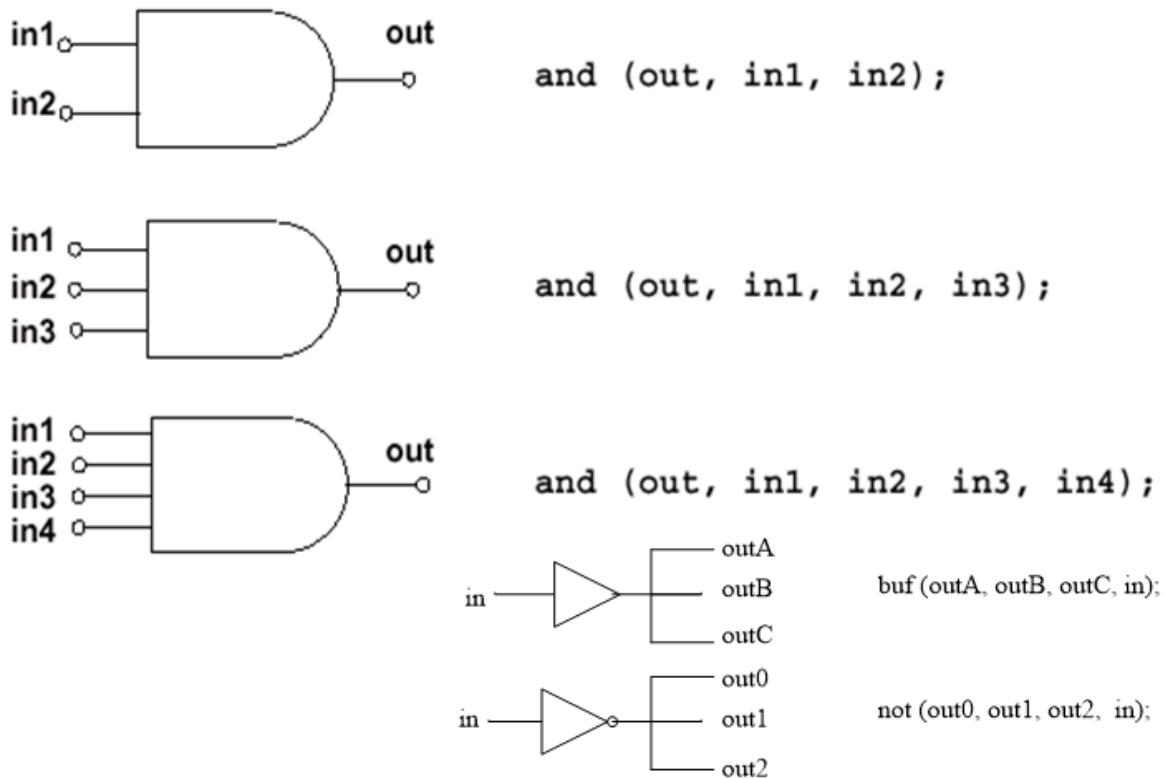
基本单元名称	功能
and	Logical And
or	Logical Or
not	Inverter
buf	Buffer
xor	Logical Exclusive Or
nand	Logical And Inverted
nor	Logical Or Inverted
xnor	Logical Exclusive Or Inverted

## 用于底层建模的基本单元

- 上拉、下拉电阻：  
-pullup pulldown
- MOS管单元：  
-cmos rcmos //r是带电阻的  
-pmos rpmos  
-nmos rnmos
- 传输线单元：  
-tran rtran  
-tranif0 rtranif0 //带三态门  
-tranif1 rtranif1

## 基本单元的引脚 (pin)的可扩展性

- 基本单元引脚数目由连接到门上的net的数量决定。因此当基本单元输入或输出的数量变化时用户不需要重定义一个新的逻辑功能。
- 所有门（除了not和buf）可以有多个输入，但只能有一个输出。
- not和buf门可以有多个输出，但只能有一个输入。



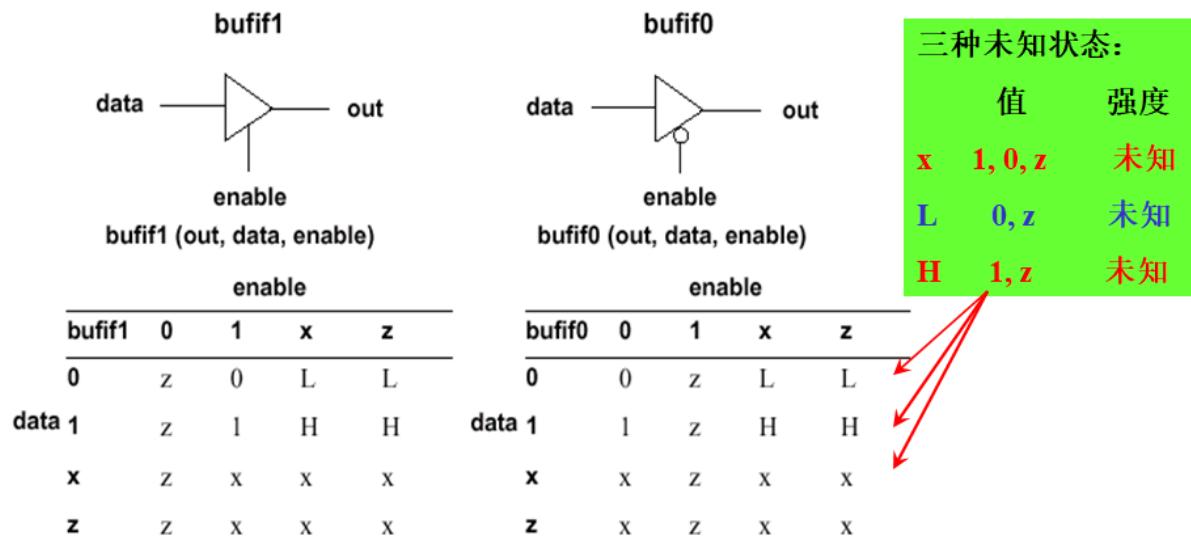
## 带条件的基本单元

- Verilog有四种不同类型的条件基本单元
- 这四种基本单元只能有三个引脚: output, input, enable
- 这些单元由enable引脚使能。

当条件基本单元使能信号无效时，输出高阻态Z。

基本单元名称	功能
<code>bufif1</code>	条件缓冲器, 逻辑 1 使能
<code>bufif0</code>	条件缓冲器, 逻辑 0 使能
<code>notif1</code>	条件反相器, 逻辑 1 使能
<code>notif0</code>	条件反相器, 逻辑 0 使能

- 条件基本单元有三个端口：输出、数据输入、使能输入



## 基本单元实例化

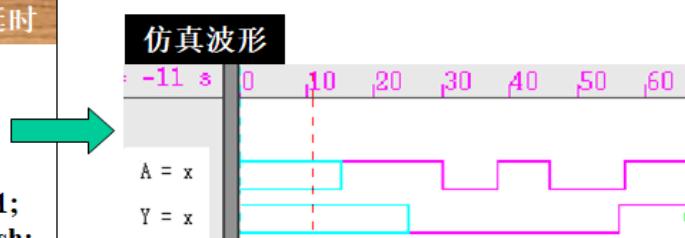
- 在端口列表中，先说明输出端口，然后是输入端口
- 实例化时实例的名字是可选项

```
and (out, in1, in2, in3, in4); // unnamed instance
buf b1 (out1, out2, in); // named instance
```

- 延时说明是可选项。所说明的延时是固有延时。输出信号经过所说明的延时才变化。没有说明时延时为0。

```
notif0 #3.1 n1 (out, in, cntrl); // delay specified
• 信号强度说明是可选项
not (strong1, weak0) n1 (inv, bit); // strength specified
```

```
module intr_sample;
  reg A;  wire Y;
  not #10 intrinsic (Y, A);
initial begin
  #15 A = 1;  #15 A = 0;  #8 A = 1;
  #8 A = 0;  #11 A = 1;  #10 $finish;
end
endmodule
```



Intrinsic 反相器  $Y = \sim A$

惯性，小于#10不取反，起到过滤作用（惯性延迟）

## Module实例化(module instantiation)

- 模块实例化时实例必须有一个名字。
- 使用位置映射时，端口次序与模块的说明相同。
- 使用名称映射时，端口次序与位置无关
- 没有连接的输入端口初始化值为x。

```

module comp (o1, o2, i1, i2);
    output o1, o2;
    input i1, i2;
    ...
endmodule

module test;
    comp c1 (Q, R, J, K); // Positional mapping
    comp c2 (.i2(K), .o1(Q), .o2(R), .i1(J)); // Named mapping
    comp c3 (Q, , J, K); // One port left unconnected
    comp c4 (.i1(J), .o1(Q)); // Named, two unconnected ports
endmodule

```

没有连接时通常会产生警告

名称映射的语法:  
.内部信号 (外部信号)

## 实例数组(Array of Instances)

- 实例名字后有范围说明时会创建一个实例数组。在说明实例数组时，实例必须有一个名字(包括基本单元实例)。其说明语法为：

<模块名字> <实例名字> <范围> (<端口>);

```

module driver (in, out, en);
    input [2: 0] in;
    output [2: 0] out;
    input en;
    bufif0 u[2:0] (out, in, en); // array of buffers
endmodule

```

```

module driver_equiv (in, out, en);
    input [2: 0] in;
    output [2: 0] out;
    input en;
    // Each primitive instantiation is done separately
    bufif0 u2 (out[2], in[2], en);
    bufif0 u1 (out[1], in[1], en);
    bufif0 u0 (out[0], in[0], en);
endmodule

```

两个模块功能完全等价

- 如果范围内MSB与LSB相同，则只产生一个实例。
- 一个实例名字只能有一个范围。
- 下面以模块comp为例说明这些情况

```

module oops;
    wire y1, a1, b1;
    wire [7: 0] a2, b2, y2, a3, b3, y3;

    comp u1 [5: 5] (y1, a1, b1); // 只产生一个comp实例
    comp m1 [0: 3] (y2, a2, b2);
    comp m1 [4: 7] (y3, a3, b3); // 非法
endmodule

```

m1作为实例阵列  
名字使用了两次  
，非法

generate

```

module .....
generate
    genvar i;
    for (i = 0; i <= 7; i = i + 1)      begin: u          //u 命名block
        adder8 add (sum[i], co[i+1], a1[i], a2[i], ci[i]);
    end
endgenerate
.....
endmodule

parameter WIDTH =1;
generate
    case(WIDTH)
        1: add1bit x1(co, sum, a1, a2, ci);
        2: add2bit x1(co, sum, a1, a2, ci);
        default: add_c #(WIDTH) x1(co, sum, a1, a2, ci);
    endcase
endgenerate

```

```

generate
    if (a1_width < 8)
        mult_8bit u1 (a1, a2, prt);
    else
        mult_16bit u1 (a1, a2, prt);

```

## 逻辑强度(strength)模型

- Verilog提供多级逻辑强度。
- 逻辑强度模型决定信号组合值是可知还是未知的，以更精确的描述硬件的行为。
- 下面这些情况是常见的需要信号强度才能精确建模的例子。
  - 开极输出(Open collector output)(需要上拉)
  - 多个三态驱动器驱动一个信号
  - MOS充电存储
  - ECL门 (emitter dotting)
- 逻辑强度是Verilog模型的一个重要部分。通常用于元件建模，如ASIC和FPGA库开发工程师才使用这么详细的强度级。但电路设计工程师使用这些精细的模型仿真也应该对此了解。

## 信号强度值系统

	Level	Type	%v formats		Specification
Supply	7	Drive	Su0	Su1	supply0, supply1
Strong	6	Drive( <b>default</b> )	St0	St1	strong0, strong1
Pull	5	Drive	Pu0	Pu1	pull0, pull1
Large	4	Capacitive	La0	La1	large
Weak	3	Drive	We0	We1	weak0, weak1
Medium	2	Capacitive	Me0	Me1	medium
Small	1	Capacitive	Sm0	Sm1	small
High Z	0	Impedance	Hi0	Hi1	highz0, highz1

- 用户可以给基本单元实例或net定义强度。

- 基本单元强度说明语法：

<基本单元名> <强度> <延时> <实例名> (<端口>) ;

例： nand (strong1, pull0) #( 2: 3: 4) n1 (o, a, b); // strength and delay

or (supply0, highz1) (out, in1, in2, in3); // no instance name

- 用户可以用%v格式符显示net的强度值

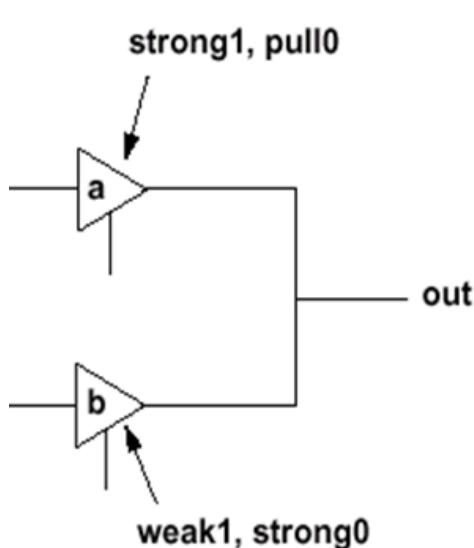
```
$monitor ($time,, " output = %v", f);
```

- 电容强度(large, medium, small)只能用于net类型trireg和基本单元tran

例如： trireg (small) tl;

## Verilog多种强度决断

- 在Verilog中，级别高的强度覆盖级别低的强度



a's output	b's output	out
strong1	strong0	strongX
pull0	weak1	pull0
pull0	strong0	strong0
strong1	weak1	strong1
pull0	HiZ	pull0
HiZ	weak1	weak1
HiZ	HiZ	HiZ

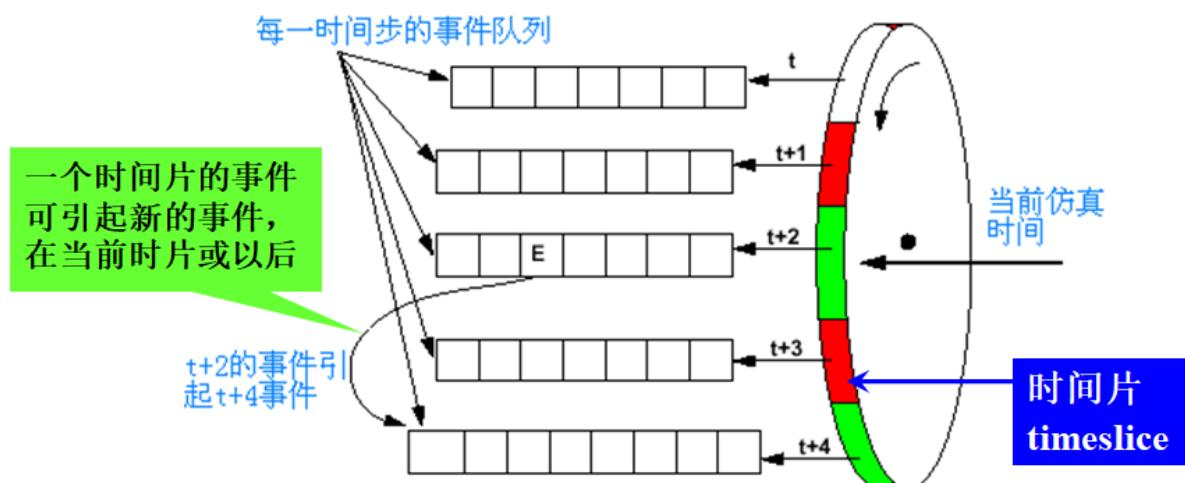
## 仿真工具及testbench编写

- 主要有三种仿真算法
  - 基于时间的(SPICE仿真器)
  - 基于事件的(Verilog-XL和NC Verilog仿真器)
  - 基于周期的(cycle)

# 仿真算法

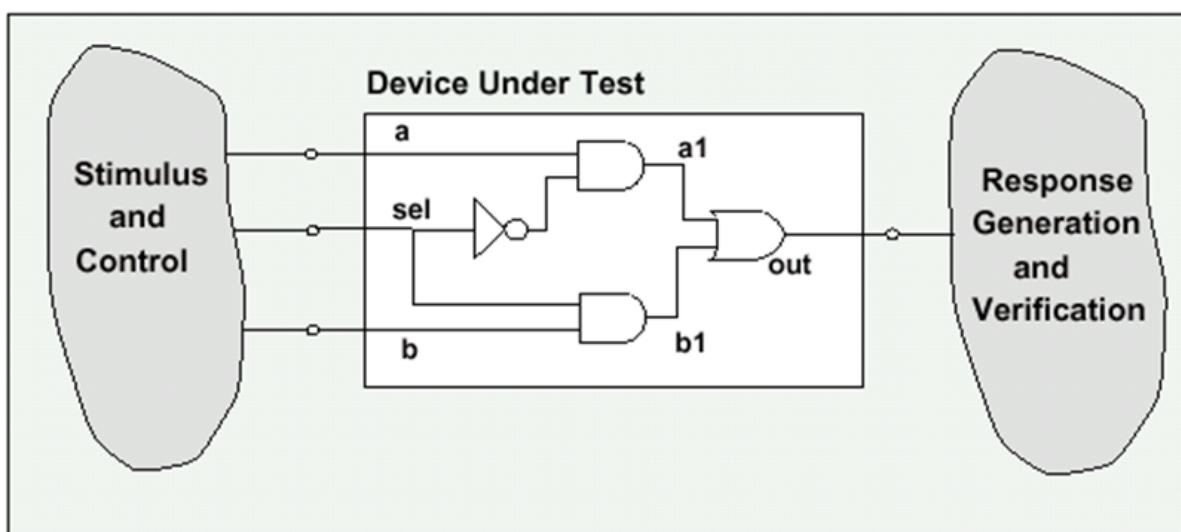
- 基于时间的算法用于处理连续的时间及变量
  - 在每一个时间点对所有电路元件进行计算
  - 效率低。在一个时间点只有约2~10%的电路活动
- 基于事件的算法处理离散的时间、状态和变量
  - 只有电路状态发生变化时才进行处理，只模拟哪些可能引起电路状态改变的元件。仿真器响应输入引脚上的事件，并将值在电路中向前传播。
  - 是应用最为广泛的仿真算法
  - 效率高。“evaluate when necessary”
- 基于周期的仿真以时钟周期为处理单位(与时间无关)
  - 只在时钟边沿进行计算，不管时钟周期内的时序
  - 使用两值逻辑 (1, 0)
  - 只关心电路功能而不关心时序，对于大型设计，效率高
  - 仅适用于同步电路。

## 基于事件仿真的时轮(time wheel)



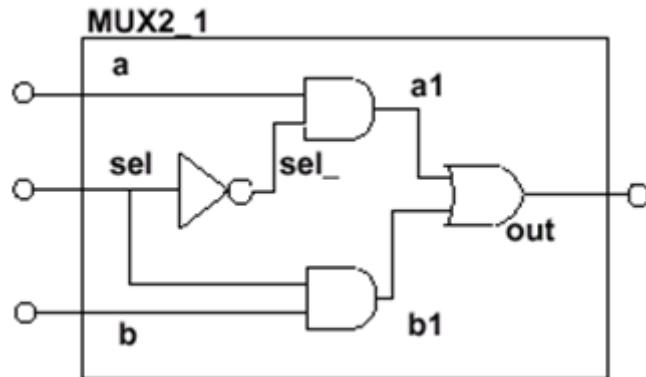
- 仿真器在编译数据结构时建立一个事件队列。
- 只有当前时间片中所有事件都处理完成后，时间才能向前。
- 仿真从时间0开始，而且时轮只能向前推进。只有时间0的事件处理完后才能进入下一时间片。
- 在同一个时间片内发生的事件在硬件上是并行的。
- 理论上时间片可以无限。但实际上受硬件及软件的限制。

## 一个完整的简单例子 test fixture



- 被测试器件DUT是一个二选一多路器。测试程序(test fixture)提供测试激励及验证机制。
- Test fixture使用行为级描述， DUT采用门级描述。下面将给出Test fixture的描述、 DUT的描述及如何进行混合仿真。

## DUT 被测器件 (device under test)



- a, b, sel是输入端口， out是输出端口。所有信号通过这些端口从模块输入/输出。
- 另一个模块可以通过模块名及端口说明使用多路器。实例化多路器时不需要知道其实现细节。这正是自上而下设计方法的一个重要特点。模块的实现可以是行为级也可以是门级，但并不影响高层次模块对它的使用。

```
module MUX2_1 (
    // Port declarations
    output wire out,
    input    wire     a,
                b,
                sel
);
    wire sel_, a1, b1;

    not (sel_, sel);
    and (a1, a, sel_);
    and (b1, b, sel);
    or   (out, a1, b1);
endmodule
```

## Test Fixture — 如何说明实例

```

module testfixture;
    // Data type declaration

    // Instantiate modules
    MUX2_1 mux (o, a, b, s);

    // Apply stimulus

    // Display results

endmodule

```

多路器实例化语句

```

module MUX2_1(
    // Port declarations
    output wire out,
    input  wire  a, b, sel
);
    wire sel_,a1,b1;

    not (sel_,sel);
    and (a1,a,sel_);
    and (b1,b,sel);
    or  (out,a1,b1);
endmodule

```

MUX的实例化语句包括：

- 模块名称：与引用模块相同
- 实例名称：任意，但要符合标记命名规则
- 端口列表：与引用模块的次序相同

## Test fixture 激励描述

```

module testfixture;
    // Data type declaration
    reg   a, d, s;
    wire o;
    // MUX instance
    MUX2_1 mux (o, a, d, s);
    // Apply stimulus
    initial
    begin
        a = 0; d = 1; s = 0;
        #5 d = 0;
        #5 d = 1; s = 1;
        #5 a = 1;
        #5 $finish;
    end
    // Display results
endmodule

```

Time	Values		
	a	d	s
0	0	1	0
5	0	0	0
10	0	1	1
15	1	1	1

- 例子中，`a`, `d`, `s`说明为`reg`类数据。`reg`类数据是寄存器类数据信号，在重新赋值前一直保持当前数据。
- `#5` 用于指示等待5个时间单位。
- `$finish`是结束仿真的系统任务。

## Test Fixture 响应产生

Verilog提供了一些系统任务和系统函数，包括：

- **`$time`** 系统函数，返回当前仿真时间
- **`$monitor`** 系统任务，若参数列表中的参数值发生变化，则在时间单位末显示参数值。

`$monitor(["format_specifiers"], <arguments>);`

例如：

```
$monitor($time, o, in1, in2);
$monitor($time, out, a, b, sel);
$monitor($time, "%b %h %d %o", sig1, sig2, sig3, sig4);
```

注意不能  
有空格

## 语言专用标记( tokens)

系统任务及函数

`$<identifier>`

- `$`符号指示这是系统任务和函数
- 系统函数有很多，如：
  - 返回当前仿真时间`$time`
  - 显示/监视信号值(`$display`, `$monitor`)
  - 停止仿真`$stop`
  - 结束仿真`$finish`

```
$monitor($time, "a = %b, b = %h", a, b);
```

```
module DFF (q, qb, d, clk, clr);
  // 端口说明
  output q, qb;
  input d, // input data
        clk, /*input clock */ clr;
  reg q;
  wire qb, d, clk, clr;
/*
  clk is posedge and
  clr is active low
*/
  assign qb = !q;
  always @(posedge clk or negedge clr)
    if(!clr)
```

```
q <= 0;  
else  
q <= d;  
endmodule
```

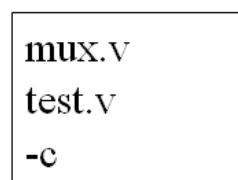
## 完整的Test Fixture

```
module testfixture;  
// 数据类型说明  
reg a, b, s;  
wire o;  
// MUX2_1 实例化  
MUX2_1 mux (o, a, b, s);  
// 施加激励  
initial begin  
a = 0; b = 1; s = 0;  
#5 b = 0;  
#5 b = 1; s = 1;  
#5 a = 1;  
#5 $finish;  
end  
// 显示结果  
initial  
$monitor($time,, " o=%b a=%b b=%b s=%b", o, a, b, s);  
endmodule
```

## 结果输出

```
0    o= 0  a= 0  b= 1  s= 0  
5    o= 0  a= 0  b= 0  s= 0  
10   o= 1  a= 0  b= 1  s= 1  
15   o= 1  a= 1  b= 1  s= 1
```

## 启动Verilog-XL

- 在命令窗口启动Verilog-XL:  
`verilog [verilog-xl_options] design_files`
- 没有option启动的例子  
`verilog mux.v test.v`
- 使用 `-c`选项只对设计进行语法和连接检查  
`verilog -c mux.v test.v`
- 使用`-f`选项指定一个包含命令行参数的文件  
`verilog -f run.f`  
run.f文件的内容 → 

```
mux.v
test.v
-c
```

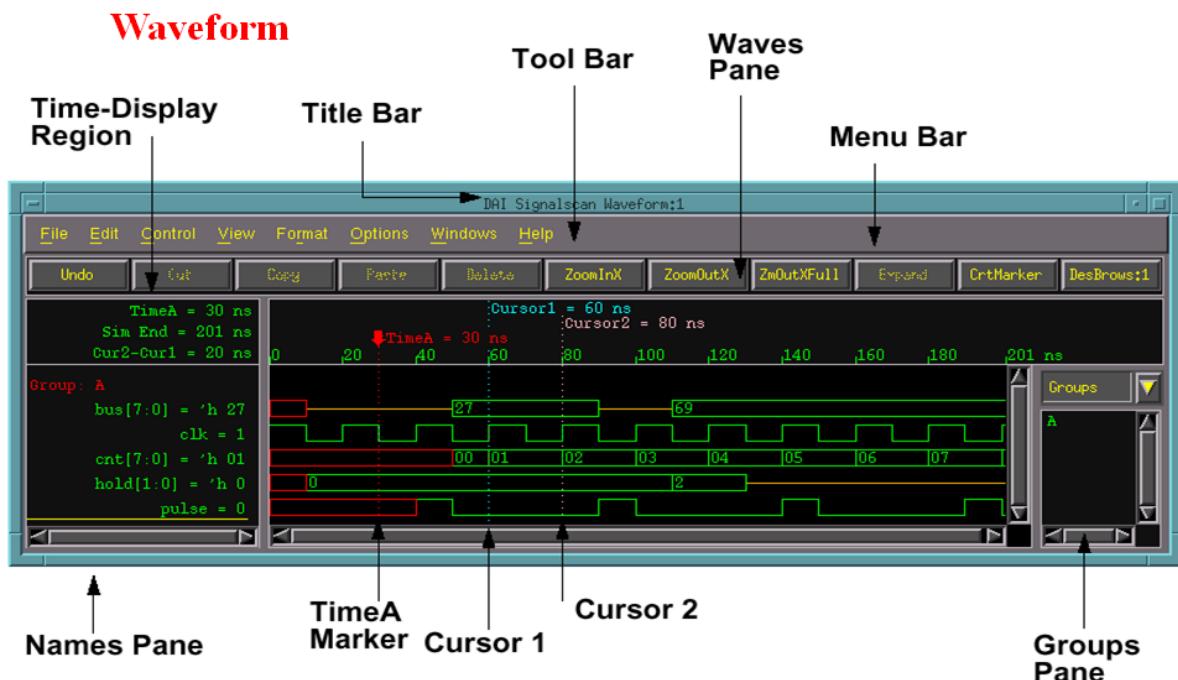
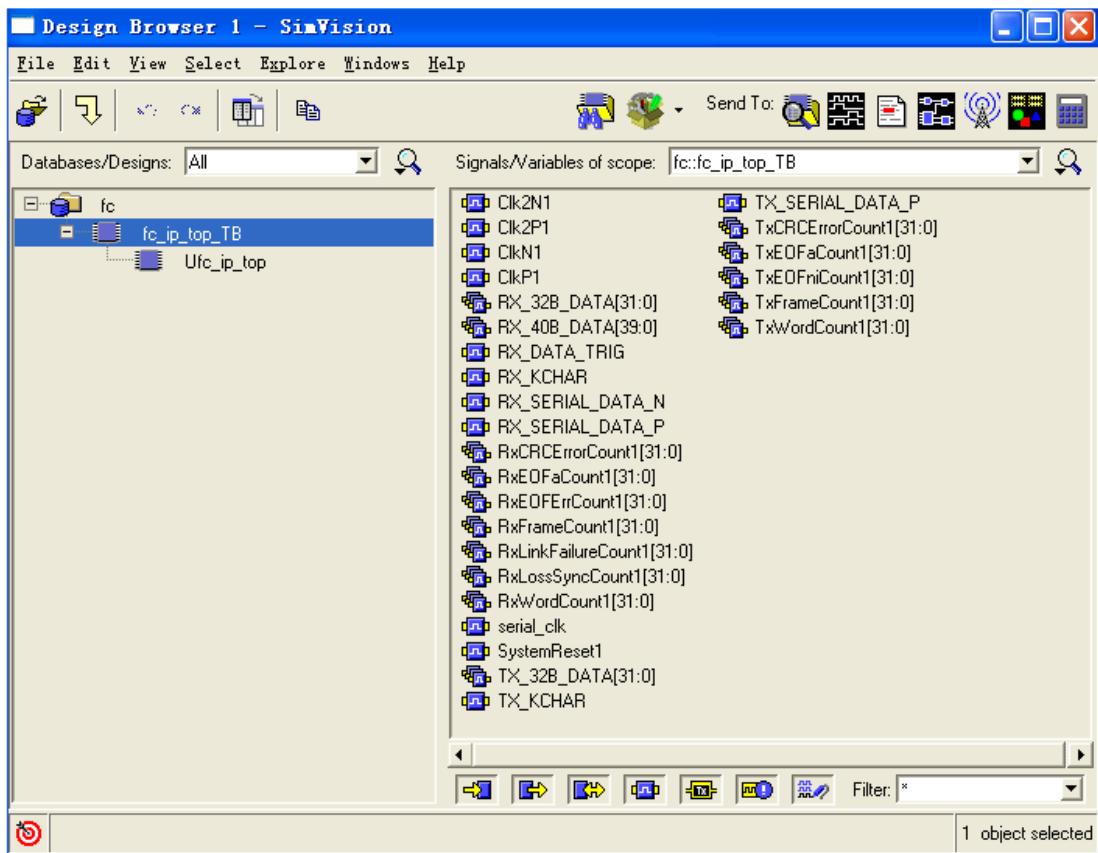
Verilog-XL将所有终端输出保存到名为`verilog.log`的文件

## 其它常用选项

- `verilog`  
显示其帮助 (所有可用选项)
- `verilog +gui`  
启动图形界面
- `verilog -v -y:`  
定义仿真库
- `verilog +define+<macro>`  
重新定义宏变量
- `verilog +sdf_file`  
重新指定sdf文件

## 波形显示工具—simvision

# Design Browser : simvision lab.shm &



## SHM: 波形数据库

波形显示工具从数据库，如SHM数据库中读取数据。使用下面的系统任务可以对SHM数据库进行操作：

系统任务	描述
\$shm_open("waves.shm");	打开一个仿真数据库。同时只能打开一个库写入。
\$shm_probe();	选择信号，当它们的值变化时写入仿真库
\$shm_close; \$shm_save;	关闭仿真库 将仿真数据库写到磁盘

```

initial
begin
    $shm_open("./data/lab.shm");
    $shm_probe();
end

```

## 用\$shm\_probe设置信号探针

- \$shm\_probe的语法：

\$shm\_probe(scope0, node0, scope1, node1, ...);

- 每个node都是基于前面scope的说明(层次化的)
- scope参数缺省值为当前范围(scope)。node参数缺省值为指定范围的所有输入、输出及输入输出。

---

### node说明 保存到数据库存的信号

---

“A” 指定范围的所有节点(包括端口(port))

“S” 指定范围及其以下所有端口，不包括库单元内部

“C” 指定范围及其以下所有端口，包括库单元内部

“AS” 指定范围及其以下所有节点(包括端口),不包括库单元内部

“AC” 指定范围及其以下所有节点(包括端口),包括库单元内部

---

## 模块实例化

```

module testfixture;

// Instantiate modules
CPU top (o, a, b, s);

initial begin
    $shm_open("./data/top.shm");
    $shm_probe(top.umul, top.udiv );
    $shm_probe("S", top.alu, "AC");
end

endmodule

```

```

module CPU (out, a, b, sel);
    // Port declarations
    output out;
    input a, b, sel;
    wire out, a, b, sel;
    wire sel_, a1, b1;

    alu ualu (sel_, sel);
    mul umul(a1, a, sel_);
    div udiv (b1, b, sel);
    dsp udsp (out, a1, b1);

initial begin
    $shm_open("./data/cpu.shm");
    $shm_probe();
end

endmodule

```

在\$shm\_probe中使用scope/node对作为参数。参数可以使用缺省值或两个参数都设置。例如：

- \$shm\_probe( ); 观测当前范围(scope)所有端口
- \$shm\_probe("A"); 观测当前范围所有节点
- \$shm\_probe(alu, adder); 观测实例alu和adder的所有端口
- \$shm\_probe("S", top.alu, "AC"); 观测：
  - (1): 当前范围及其以下所有端口，除库单元
  - (2): top.alu模块及其以下所有节点，包括库单元

## VCD数据库

Verilog提供一系列系统任务用于记录信号值变化保存到标准的VCD(Value Change Dump)格式数据库中。大多数波形显示工具支持VCD格式。

系统任务	功能
\$dumpfile("file.dump");	打开一个VCD数据库用于记录
\$dumpvars();	选择要记录的信号
\$dumpflush;	将VCD数据保存到磁盘
\$dumpoff;	停止记录
\$dumpon;	重新开始记录
\$dumplimit(<file_size>);	限制VCD文件的大小(以字节为单位)
\$dumpall;	记录所有指定的信号值

VCD数据库是仿真过程中数据信号变化的记录。它只记录用户指定的信号。

- 用户可以用\$dump系统任务打开一个数据库，保存信号并控制信号的保存。除\$dumpvars外，其它任务的作用都比较直观。\$dumpvars将在后面详细描述。
- 必须首先使用\$dumpfile系统任务，并且在一次仿真中只能打开一个VCD数据库。

- 在仿真前(时间0前)必须先指定要观测的信号，这样才能看到信号完整的变化过程。
- 仿真时定期的将数据保存到磁盘是一个好的习惯，万一系统出现问题数据也不会全部丢失。
- VCD数据库不记录仿真结束时的数据。因此如果希望看到最后一次数据变化后的波形，必须在结束仿真前使用\$dumpall。

要给\$dumpvars提供层次(levels)及范围(scope)参数，例如

```
$dumpvars; // Dump所有层次的信号
$dumpvars (1, top); // Dump top模块中的所有信号
$dumpvars (2, top.u1); // Dump实例top. u1及其下一层的信号
$dumpvars (0, top.u2, top.u1.u13.q); // Dump top.u2及其以下所有信号，以及信号
top.u1.u13.q。
$dumpvars (3, top.u2, top.u1); // Dump top.u1和top.u2及其下两层中的所有信号。
```

用下面的代码可以代替前面test fixture的\$monitor命令：

```
initial
begin
    $dumpfile ("verilog.dump");
    $dumpvars (0, testfixture);
end
```

## \$dumpvars语法

**\$dumpvars[(< levels>, <scope>\*)];**

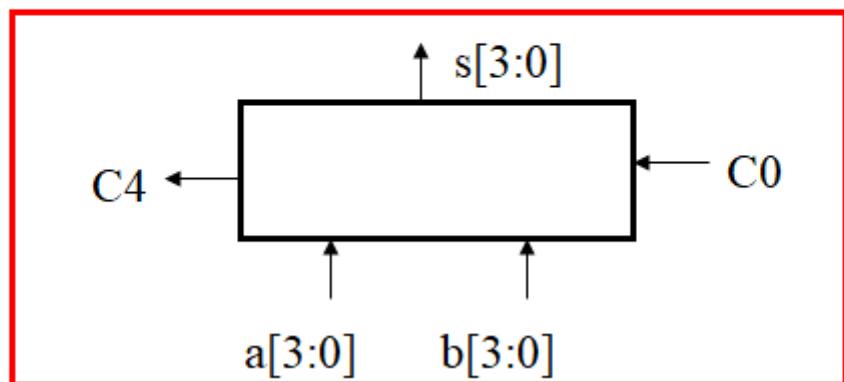
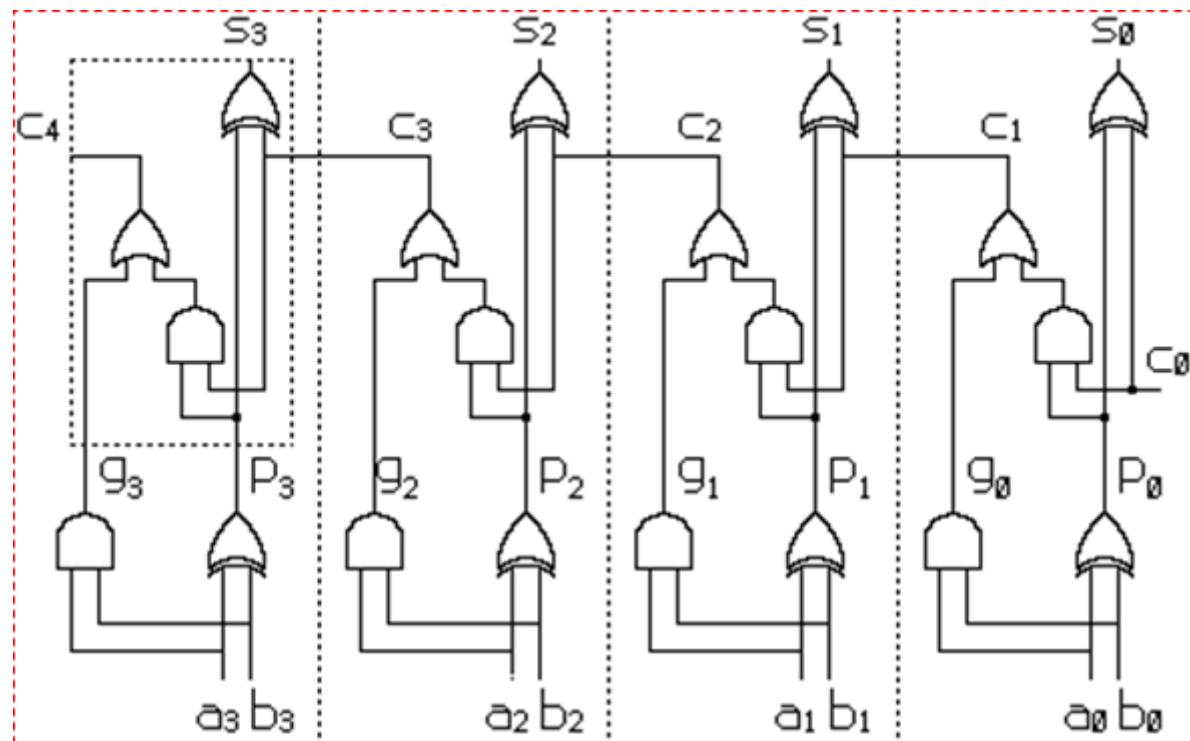
*scope*可以是层次中的信号，实例或模块。

- 仿真时所有信号必须在同一时间下使用\$dumpvars。
- 就是说可以使用多条\$dumpvars语句，但必须从同一时间开始。  
如：

```
initial begin
    $dumpfile ("verilog.dump");
    $dumpvars (0, testfixture.a);
#1 $dumpvars (0, testfixture.b);
end
```

此语句将引起一个警  
告信息并被忽略

## 练习



```

module adder_rp1(
    input wire [3 : 0]    a,
    input wire            b,
    input wire             ci,
    output wire            co,
    output reg [3 : 0]    s
);
    reg [3 : 0]    g,
                  p;
    reg [4 : 0]    c;

    assign co = c[4];
    always @(a, b, ci) begin
        c[0] = ci;
        g     = a & b;
        p     = a ^ b;
        c[4:1] = g | (p & c[3:0]);
        s      = p ^ c[3:0];
    end
endmodule

```

```

module adder_rp1(
    input wire [3 : 0]    a,

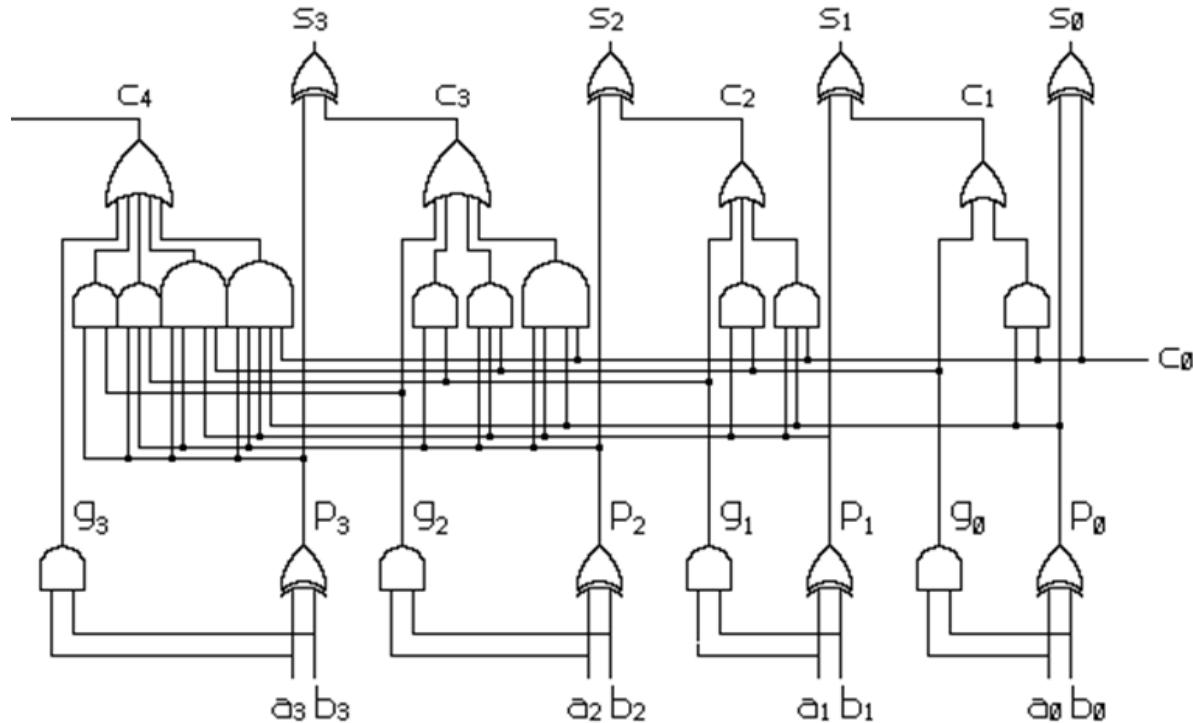
```

```

    b,
input wire          ci,
output wire         co,
output reg [3 : 0]  s
);
reg [3 : 0]  g,
               p;
reg [4 : 0]  c;
integer   i;
assign co = c[4];
always @(a, b, ci) begin
  c[0] = ci;
  for(i = 0; i < n; i = i+1) begin
    g[i]    = a[i]&b[i];
    p[i]    = a[i]^b[i];
    c[i+1]  = g[i] | (p[i]&c[i]);
    s[i]    = p[i]^c[i];
  end
endmodule

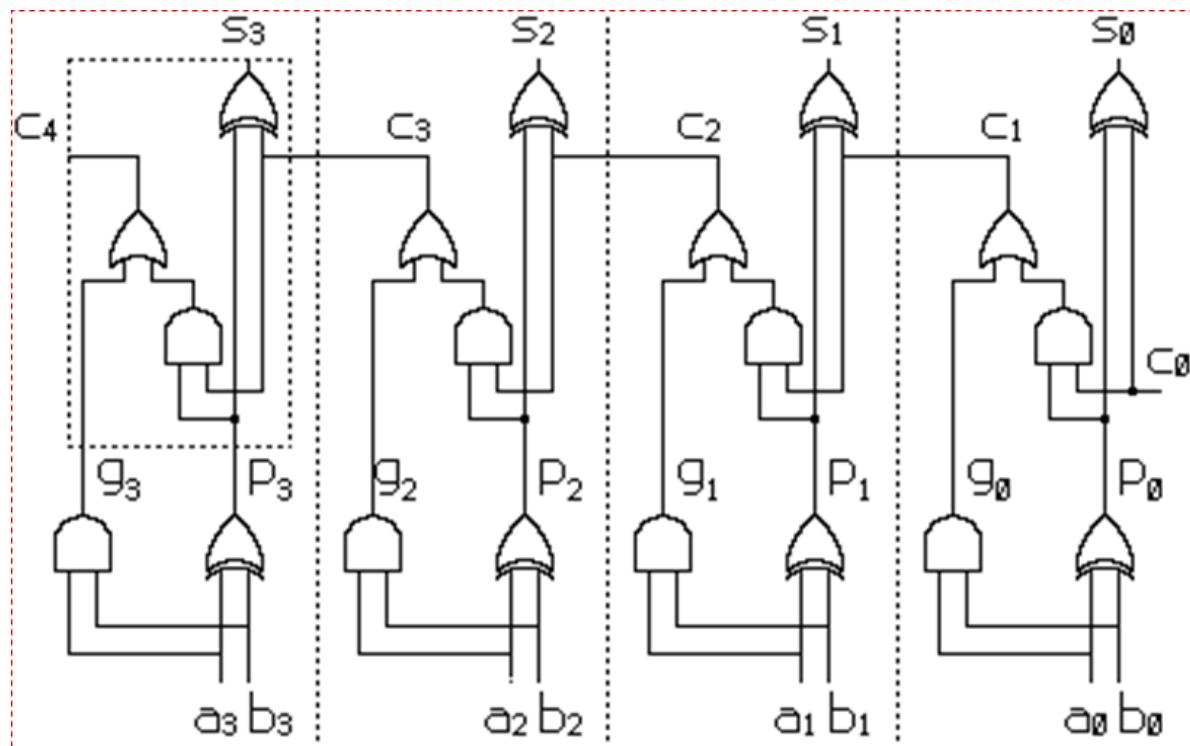
```

请分别给出下面电路的结构级和行为级模型



## ALU的功能

- 算术运算
  - 加、减法运算
- 逻辑运算
  - 按位逻辑计算
  - and nand or nor xor xnor buf not
  - 输出全0, 全1



## 逻辑运算

- 逻辑操作是按位完成的。
- 逻辑操作可以通过最小项的组合来完成。

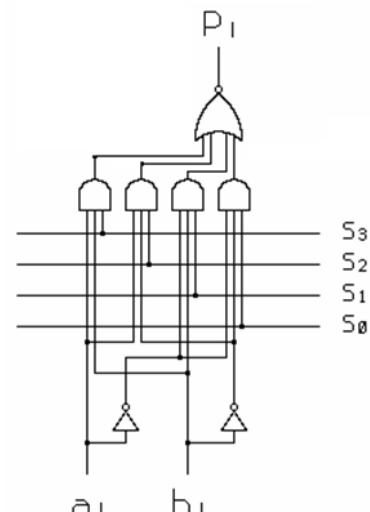
两个被操作数  $a$  与  $b$  可以有四个最小项:

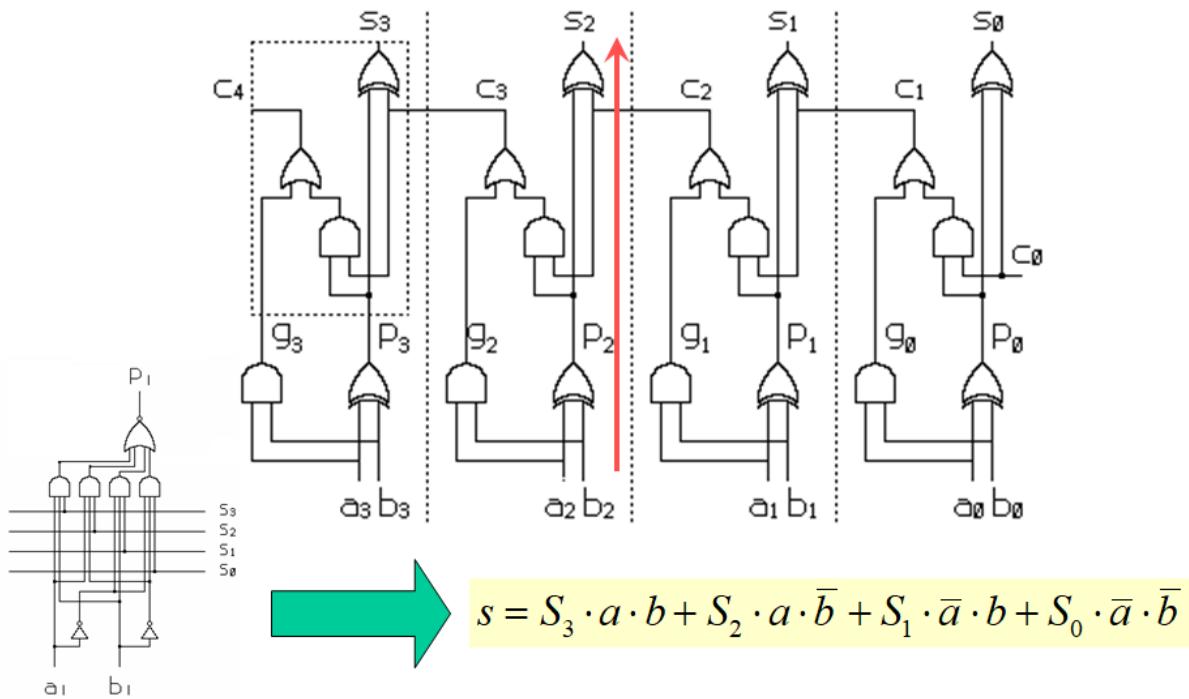
$$\bar{a} \cdot \bar{b} \quad a \cdot \bar{b} \quad \bar{a} \cdot b \quad a \cdot b$$

- 这些最小项分别通过选择控制信号  $S_i$  ( $i=0,1,2,3$ ) 进行控制，并为了便于产生算术操作时的进位传递信号取其逻辑反信号，就可以有公式:

$$p = \overline{S_3 \cdot a \cdot b + S_2 \cdot a \cdot \bar{b} + S_1 \cdot \bar{a} \cdot b + S_0 \cdot \bar{a} \cdot \bar{b}}$$

如何实现逻辑运算?





## 算术运算

- 所有算术操作都是通过加法与减法操作来实现的，加法操作的进位产生信号 $g_+$ 与进位传递信号 $p_+$ 分别为：

$$g_+ = a \cdot b$$

$$p_+ = a \cdot \bar{b} + \bar{a} \cdot b$$

- 减法操作的进位产生信号 $g_-$ 与进位传递信号 $p_-$ 分别为：

$$g_- = a \cdot \bar{b}$$

$$p_- = a \cdot b + \bar{a} \cdot \bar{b}$$

$$p = \overline{S_3 \cdot a \cdot b + S_2 \cdot a \cdot \bar{b} + S_1 \cdot \bar{a} \cdot b + S_0 \cdot \bar{a} \cdot \bar{b}}$$

$$S = p \cdot \bar{c} + \bar{p} \cdot c$$

$$S = S_3 \cdot a \cdot b + S_2 \cdot a \cdot \bar{b} + S_1 \cdot \bar{a} \cdot b + S_0 \cdot \bar{a} \cdot \bar{b}$$

$$g = S_3 \cdot a \cdot b + S_2 \cdot a \cdot \bar{b} \quad \rightarrow \quad g = S_3 \cdot a \cdot b + S_2 \cdot a \cdot \bar{b} + \bar{M}$$

S3	S2	S1	S0	M	C0
$g_+ = a \cdot b$	1	0	0	1	1
$p_+ = a \cdot \bar{b} + \bar{a} \cdot b$					
$g_- = a \cdot \bar{b}$	0	1	1	0	1
$p_- = a \cdot b + \bar{a} \cdot \bar{b}$					

注：异或取反是同或，同或取反是异或

$$p = S_3 \cdot a \cdot b + S_2 \cdot a \cdot \bar{b} + S_1 \cdot \bar{a} \cdot b + S_0 \cdot \bar{a} \cdot \bar{b}$$

$$S = p \cdot \bar{c} + \bar{p} \cdot c$$

$$S = S_3 \cdot a \cdot b + S_2 \cdot a \cdot \bar{b} + S_1 \cdot \bar{a} \cdot b + S_0 \cdot \bar{a} \cdot \bar{b}$$

$$g = S_2 \cdot a \cdot b + S_3 \cdot a \cdot \bar{b} \quad \rightarrow \quad g = (S_2 \cdot a \cdot b + S_3 \cdot a \cdot \bar{b}) \bar{M}$$

S3	S2	S1	S0	M	C0
$g_+ = a \cdot b$	0	1	1	0	0
$p_+ = a \cdot \bar{b} + \bar{a} \cdot b$					
$g_- = a \cdot \bar{b}$	1	0	0	1	0
$p_- = a \cdot b + \bar{a} \cdot \bar{b}$					

- 对于加法，使选择线 $S_0$ 与 $S_3$ 有效，就可以得到

$$p_+ = a \cdot \bar{b} + \bar{a} \cdot b$$

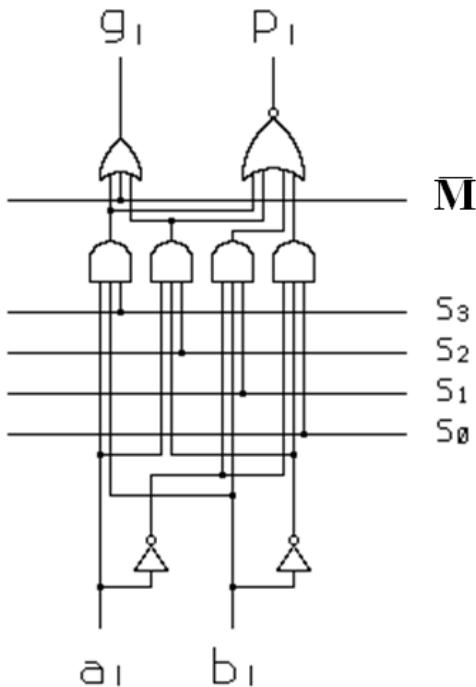
- 对于减法，使选择控制线 $S_1$ 与 $S_2$ 有效，就可以得到

$$p_- = a \cdot b + \bar{a} \cdot \bar{b}$$

- $g_+$ 信号与 $g_-$ 信号分别是由 $S_3$ 与 $S_2$ 控制的，它们的逻辑或进位产生信号 $g$ 。为了使ALU在进行逻辑运算时进位信号不产生影响，使用 $M$ 信号来封锁进位产生信号（为“0”时封锁），这样进位产生信号的逻辑表达式为：

$$g = S_3 \cdot a \cdot b + S_2 \cdot a \cdot \bar{b} + \bar{M}$$

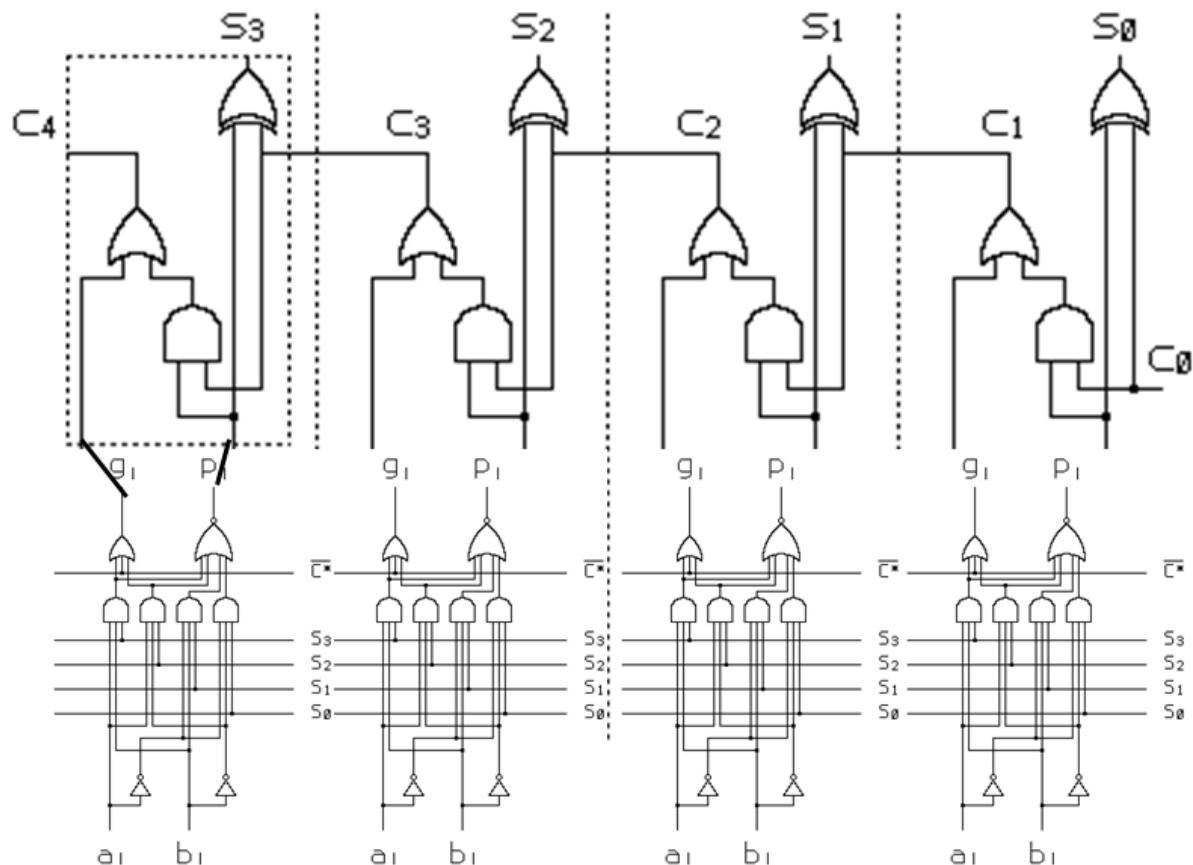
- 按照这种设计，通过对四条选择控制线 $S_i$  ( $i=0,1,2,3$ ) 和向末位的进位信号 $c_0$ 与进位链控制信号 $M$ 可以产生16种逻辑操作与两种算术操作，如表所示。



用前面介绍的进位链产生各位进位。各位结果为：

$$s_i = p_i \oplus c_i$$

注：M是控制是逻辑运算还是算术运算，对于第一种方式：0：逻辑运算，1：算术运算



选择控制码						逻辑	功能
S <sub>3</sub>	S <sub>2</sub>	S <sub>1</sub>	S <sub>0</sub>	c <sub>in</sub>	M		
0	0	0	0	1	0	0	置全“0”
0	0	0	1	1	0	(not a)·(not b)	nor
0	0	1	0	1	0	(not a)·b	notand
0	0	1	1	1	0	not a	not
0	1	0	0	1	0	a·(not b)	andnot
0	1	0	1	1	0	not b	not
0	1	1	0	1	0	a·(not b) + (not a)·b	xor
0	1	1	1	1	0	(not a) + (not b)	nand
1	0	0	0	1	0	a·b	and
1	0	0	1	1	0	a·b + (not a)·(not b)	xnor
1	0	1	0	1	0	b	传送b
1	0	1	1	1	0	a·b + (not a).b + (not a)·(not b)	notor
1	1	0	0	1	0	a	传送a
1	1	0	1	1	0	a·b + a·(not b) + (not a)·(not b)	ornot
1	1	1	0	1	0	a·b + a·(not b) + (not a)·b	or
1	1	1	1	1	0	1	置全“1”
1	0	0	1	0	1	a xor b xor c	加法
0	1	1	0	1	1	(a xnor b) xor c	减法

### 作业3

## ➤ 完成一个32位ALU设计并进行功能验证。

```
module alu_core #(
    parameter n = 32
) (
    input wire [n-1 : 0] opA, //操作数A
    input wire [n-1 : 0] opB, //操作数B
    input wire [3 : 0] S , //工作模式选择信号
    input wire M , //逻辑操作控制信号
    input wire Cin , //进位输入信号
    output reg [n-1 : 0] DO , //数据输出
    output reg C , //进位输出
    output reg V , //溢出指示输出信号
    output reg N , // DO符号位输出信号
    output reg Z , //DO为全0指示信号
);

```

## ALU计算溢出的判断方法

- 溢出是指：运算结果超出数的表示范围。

不同符号的数相加不会产生溢出。

- 溢出分正溢和负溢。正溢：正数相加溢出；负溢：相加绝对值溢出。

$$\begin{array}{r} 0110 \\ + 0011 \\ \hline 1001 \end{array}$$

- 判断溢出的方法：

设两个操作数分别为A[31: 0]、B[31: 0]，结果为S[31: 0]，

符号位分别是A[31]、B[31]、S[31]

$$\begin{array}{r} 1010 \\ + 1100 \\ \hline 10110 \end{array}$$

**方法1**：正溢：{A[31], B[31], S[31]} = 3'b001

负溢：{A[31], B[31], S[31]} = 3'b110

**方法2**：第30位向第31位的进位C[31]，和 第31位向第32位的进位C[32]

C[31] 与 C[32] 不同

**方法3**：将ALU扩展为33位计算，最高两位符号位不同。

## Verilog的可综合描述风格

### 描述风格简介

- 如果逻辑输出在任何时候都直接由当前输入组合决定，则为组合逻辑。
- 如果输出在任何给定时刻不能由输入的状态决定，则为时序逻辑。

通常综合输出不会只是一个纯组合或纯时序逻辑。

一定要清楚

所写的源代码会综合出什么类型逻辑，至少要知道其拓扑结构

要得到指定的逻辑结构，需要如何描述

| 这是非常重要的。

## 不支持的Verilog结构

综合工具通常不支持下列Verilog结构：

**initial**

**UDP**

**循环：**

**fork...join块**

**repeat**

**wait**

**forever**

**过程持续赋值：**

**while(2001标准后支持但有限制)**

**assign deassign**

**非结构化的for语句**

**force release**

**数据类型：**

**操作符：**

**event**

**====**

**real**

**! ==**

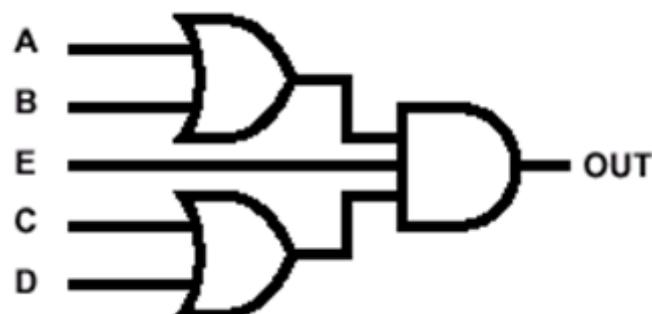
**time**

## 持续赋值

持续赋值驱动值到net上。因为驱动是持续的，所以输出将随任意输入的改变而随时更新，因此将产生组合逻辑。

```
module operand (out, a, b, c, d, e);
    input a, b, c, d, e;
    output out;
    assign out = e & (a | b) & (c | d);
endmodule
```

```
module operand (); //另类的assign
    input wire a,
           en,
    output wire out);
    assign out = en ? a : out;
endmodule
```



## 延迟赋值语句

语法: LHS = <timing\_control> RHS;

- 时序控制延迟的是赋值而不是右边表达式的计算。
- 在延迟赋值语句中RHS表达式的值都有一个隐含的临时存储。
- 可以用来简单精确地模拟寄存器交换和移位。

```
begin
    temp= b;
    @ (posedge clk) a =
temp;
end
```

等价语句

```
a = @ ( posedge clk)
b;
```

LHS: Left-hand-side  
RHS: Right-hand-side

## 非阻塞过程赋值语句

```
module swap_vals;
    reg a, b, clk;
initial begin
    a = 0; b = 1; clk = 0;
end
always #5 clk = ~clk;
always @ ( posedge clk)
begin
    a <= b; // 非阻塞过程赋值
    b <= a; // 交换a和b值
end
endmodule
```

过程赋值有两类  
阻塞过程赋值  
非阻塞过程赋值

阻塞过程赋值执行完成后再执行在顺序块内下一条语句。

非阻塞赋值不阻塞过程流，仿真器读入一条赋值语句并对它进行调度之后，就可以处理下一条赋值语句。

若过程块中的所有赋值都是非阻塞的，赋值按两步进行：

1. 仿真器计算所有RHS表达式的值，保存结果，并进行调度在时序控制指定时间的赋值。
2. 在经过相应的延迟后，仿真器通过将保存的值赋给LHS表达式完成赋值。

### 阻塞与非阻塞赋值语句行为差别举例

```

module non_block1;
    reg a, b, c, d, e, f;
    initial begin // 阻塞赋值
        a = #10 1; // time 10
        b = #2 0; // time 12
        c = #4 1; // time 16
    end
    initial begin // 非阻塞赋值
        d <= #10 1; // time 10
        e <= #2 0; // time 2
        f <= #4 1; // time 4
    end
    initial begin
        $monitor("$ time,, " a= %b b= %b c= %b d= %b e= %b f= %b",
a, b, c, d, e, f);
        #100 $finish;
    end
endmodule

```

输出结果:

```

0  a=x b=x c=x d=x e=x f=x
2  a=x b=x c=x d=x e=0 f=x
4  a=x b=x c=x d=x e=0 f=1
10 a=1 b=x c=x d=1 e=0 f=1
12 a=1 b=0 c=x d=1 e=0 f=1
16 a=1 b=0 c=1 d=1 e=0 f=1

```

## 过程块语句

- 任意边沿

- 在所有输入信号的任意边沿进入的过程块称为组合块。

```

always @(a or b) // 与门
y = a & b;

```

- 单个边沿

- 在一个控制信号的单个边沿上进入的过程块产生同步逻辑。这种过程块称为同步块。

```

always @ (posedge clk) // D flip-flop
q <= d;

```

- 同步块也可以对异步复位信号的变化产生敏感

```

always @ (posedge clk or negedge rst_)
if (!rst_)
    q <= 0;
else
    q <= d;

```

## 组合块描述风格

```

module DFF (q, qb, d, clk, clr);
    output q, qb;
    input d, /* input data
               clk, /* input clock */ clr;
    reg q;
    wire qb, d, clk, clr;

    assign qb = !q;

    always @ (a, b, c...) begin
        // 过程赋值语句;
        // 高级描述语句
    end
endmodule

```

```

if语句;
case语句;
循环语句;

end
endmodule

```

- 任意边沿

- 在所有输入信号的任意边沿进入的过程块称为组合块。

```
always @( a or b)
```

```
y = a & b;
```

- 敏感列表要完全**
- 采用阻塞赋值语句
- 可能产生**组合逻辑或锁存器**
- 过程赋值语句产生组合逻辑
- IF或CASE语句可能产生**锁存器**
- 循环语句?

## 组合块：敏感列表要完全

在下面的例子，a, b, sl是块的输入

- sl用作条件，a、b用在过程赋值语句的右边

### 敏感表不完全：

```

module sens (q, a, b, sl);
input a, b, sl;
output q;
reg q;
always @( sl)
begin
  if (! sl)
    q = a;
  else
    q = b;
end
endmodule

```

### 完全的敏感列表

```

module sensc (q, a, b, sl);
input a, b, sl;
output q;
reg q;
always @( sl or a or b)
begin
  if (! sl)
    q = a;
  else
    q = b;
end
endmodule

```

将块的所有输入都列入敏感表是很好的描述习惯。不同的综合工具对不完全敏感表的处理有所不同。有的将不完全敏感表当作非法。其他的则产生一个警告并假设敏感表是完全的。在这种情况下，综合输出和RTL描述的仿真结果可能不一致。

## 2001标准对敏感列表的简化

```

module sens (q, a, b, sl);
    input a, b, sl;
    output q;
    reg q;

    always @(*)
        begin
            if (!sl)
                q = a;
            else
                q = b;
        end
endmodule

```

相当于

```

module sensc (q, a, b, sl);
    input a, b, sl;
    output q;
    reg q;

    always @( sl or a or b)
        begin
            if (!sl)
                q = a;
            else
                q = b;
        end
endmodule

```

## 条件语句

*if* 和 *if-else* 语句：

```

always #20
    if (index > 0) // 开始外层 if
        if (regA > regB) // 开始内层第一层 if
            result = regA;
        else
            result = 0; // 结束内层第一层 if
    else
        if (index == 0)
            begin
                $display(" Note : Index is zero");
                result = regB;
            end
        else
            $display(" Note : Index is negative");

```

描述方式：

```

if (逻辑表达式)
begin
    .....
end
else
begin
    .....
end

```

- 可以多层嵌套。在嵌套*if*序列中，*else*和前面最近的*if*相关。
- 为提高可读性及确保正确关联，使用*begin...end*块语句指定其作用域。

```

module compute (result, regA, regB, opcode);
    input [7: 0] regA, regB;
    input [2: 0] opcode;
    output [7: 0] result;
    reg [7: 0] result;
    always @(*)
        case (opcode)
            3'b000 : result = regA + regB;
            3'b001 : result = regA - regB;
            3'b001 , // 多个case有同一个结果
            3'b010 : result = regA / regB;
            default : begin
                result = 'bx;
                $display (" no match");
            end
        endcase
    end
endmodule

```

```

    end
endcase
endmodule

```

在Verilog中重复说明case项是合法的，因为Verilog的case语句只执行第一个符合项。

### case、if语句可互相代替

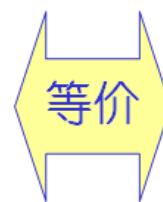
#### 条件互斥

```

module tmp(a, b, c, d, s, o);
    input wire a, b, c, d;
    input wire [1:0] s;
    output reg o;

    always @ (a, b, c, d, s)
        case(s)
            2'b00:      o = a;
            2'b01:      o = b;
            2'b10:      o = c;
            2'b11:      o = d;
            default:    o = 1'bx;
        endcase
    endmodule

```



```

module tmp(a, b, c, d, s, o);
    input wire a, b, c, d;
    input wire [1:0] s;
    output reg o;

    always @ (a, b, c, d, s)
        if(s == 2'b00)
            o = a;
        else if(s == 2'b01)
            o = b;
        else if(s == 2'b10)
            o = c;
        else if(s == 2'b11)
            o = d;
        else
            o = 1'bx;
    endmodule

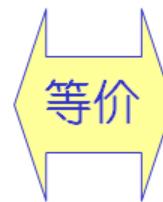
```

```

module tmp(a, b, c, d, s, o);
    input wire a, b, c, d;
    input wire [1:0] s;
    output reg o;

    always @ (a, b, c, d, s)
        case(1'b1)
            a:          o = c;
            b:          o = d;
            default:    o = s;
        endcase
    endmodule

```



```

module tmp(a, b, c, d, s, o);
    input wire a, b, c, d;
    input wire [1:0] s;
    output reg o;

    always @ (a, b, c, d, s)
        if( a )          o = c;
        else if( b )    o = d;
        else             o = s;
    endmodule

```

**One-hot style**

## 会产生锁存器 (latch)

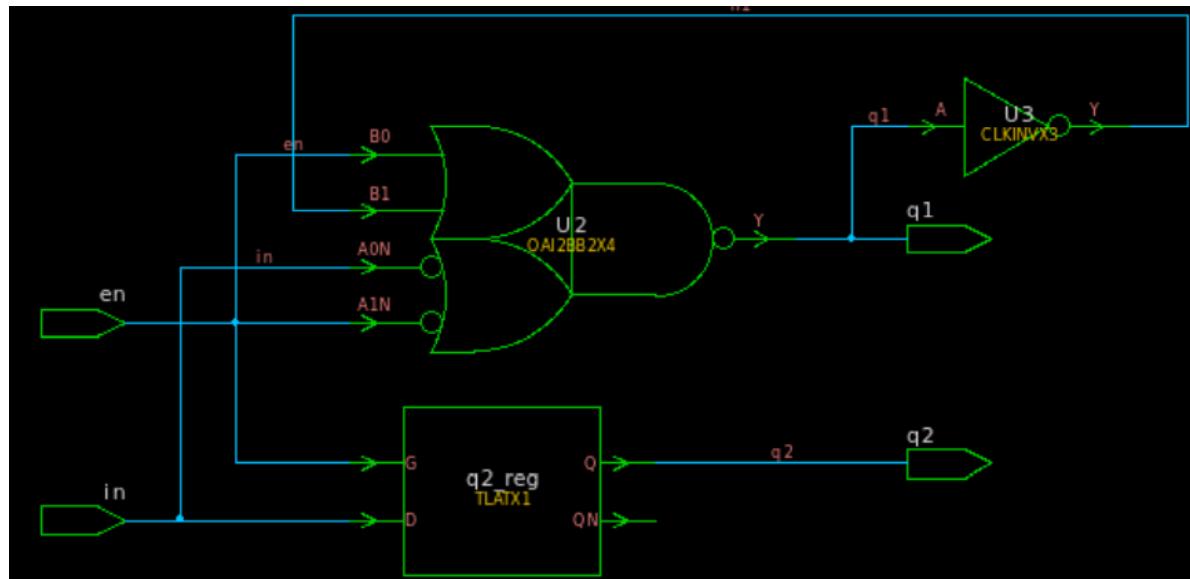
在always块中，条件语句如果没有说明所有条件，即条件不完全，将产生latch。

在下面的例子中，由于没有定义enable为低电平时data的状态，因此enable为低电平时data的值必须保持，综合时将产生一个锁存器

```
module latch (
    input wire      data,
    input wire      enable,
    output reg      q
);

    always @(* enable, data)
        if (enable)
            q = data;
endmodule
```

注：assign q = en? in, q //FPGA：锁存器；ASIC：反馈震荡电路



## 自然完全的条件语句

```
module comcase (
    input wire a, b, c, d,
    output reg e
);

    always @(* a or b or c or d)
        case ({ a, b })
            2'b11: e = d;
            2'b10: e = ~c;
            2'b01: e = 1'b0;
            2'b00: e = 1'b1;
        endcase
endmodule

module compif (
    input wire a, b, c, d,
    output reg e
);
```

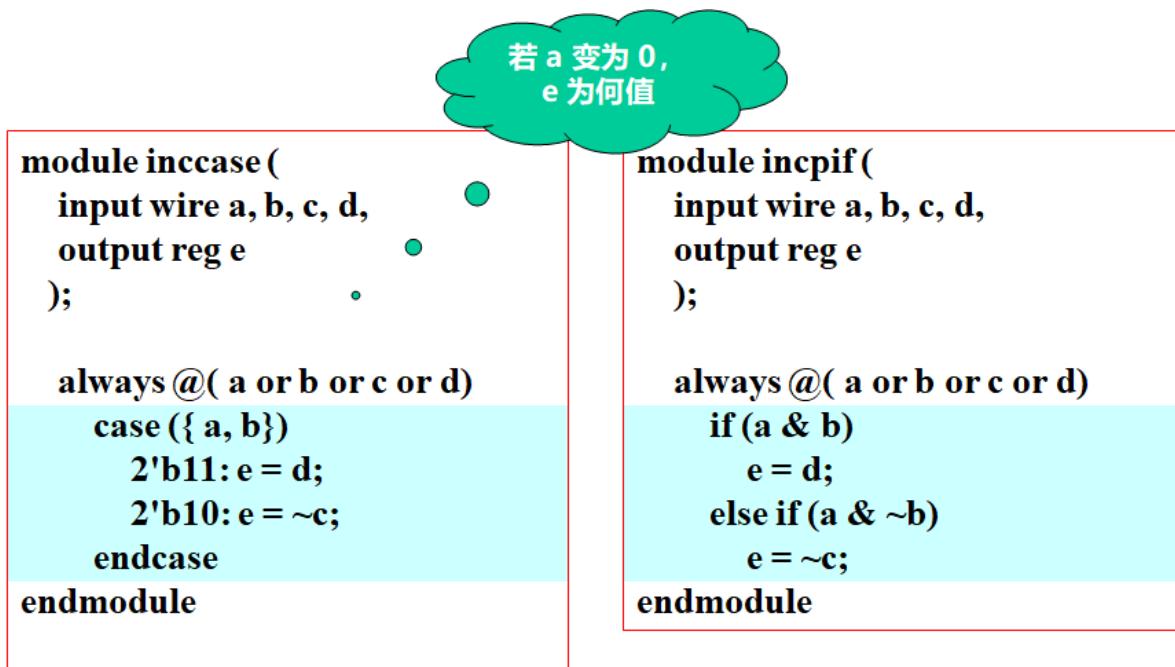
```

always @( a or b or c or d)
  if (a & b)
    e = d;
  else if (a & ~b)
    e = ~c;
  else if (~a & b)
    e = 1'b0;
  else if (~a & ~b)
    e = 1'b1;
endmodule

```

例中定义了所有可能的选项，综合结果是纯组合逻辑，没有不期望的锁存器产生。

## 不完全条件语句



在上面的例子中，当a变为零时，不对e赋新值。因此e保存其值直到a变为1。这是锁存器的特性。

## default完全条件语句

```

module comcase(
  input wire a, b, c, d,
  output reg e
);

always @( a, b, c, d)
  case ({ a, b})
    2'b11: e = d;
    2'b10: e = ~c;
    default: e = 'bx;
  endcase
endmodule

```

```

module compif(
  input wire a, b, c, d,
  output reg e
);

always @( a, b, c, d)
  if (a & b)
    e = d;
  else if (a & ~b)
    e = ~c;
  else
    e = 'bx;
endmodule

```

综合工具将 'bx作为无关值，因此if语句类似于“full case”，可以进行更好的优化。

例中没有定义所有选项，但对没有定义的项给出了缺省行为。同样，其综合结果为纯组合逻辑——没有不期望的锁存器产生。

## 指示完全条件语句

```
module dircase (a, b, c, d);
    input b, c;
    input [1: 0] a;
    output reg d;

    always @(* a or b or c)
        case (a) // synopsys synthesis case = full synopsys: 公司的名字, 综合指令指示
            完全, 最好不要用, 产生了工具依赖

                2'b00: d = b;
                2'b01: d = c;
            endcase
    endmodule
```

和前例一样，没有定义所有case项，但综合指令通知优化器缺少的case项不会发生。结果也为纯组合逻辑，没有不期望锁存器产生。注意如果缺少的case项发生，而其结果未定义，综合结果和RTL的描述的行为可能不同。

## 综合指令

- 大多数综合工具都能处理综合指令。
- 综合指令可以嵌在Verilog注释中，因此他们在Verilog仿真时忽略，只在综合工具解析时有意义。
- 不同工具使用的综合指令在语法上不同。但其目的相同，都是在RTL代码内部进行最优化。
- 通常综合指令中包含工具或公司的名称。例如，下面介绍的Envia synopsys synthesis工具的编译指示都以synopsys synthesis开头。

### 综合指令举例

- 这里列出部分Cadence综合工具中综合指令。这些与其他工具，如Synopsys Design Compiler，中的指令很相似。

```
// synopsys synthesis on
// synopsys synthesis off
// synopsys synthesis case = full, parallel, mux
• 结构指令
// synopsys synthesis architecture = cla or rpl
• FSM指令
// synopsys synthesis enum xyz
// synopsys synthesis state_vector sig state_vector_flag
```

### 综合指令 — case指示

- case语句通常综合为一个优先级编码器，列表中每个case项都比后面的case项的优先级高。
- case指令按下面所示指示优化器：
  - //synopsys synthesis case = parallel  
建立并行的编码逻辑，彼此无优先级。
  - //synopsys synthesis case = mux

若库中有多路器，使用多路器建立编码逻辑。

o //synopsis synthesis case = full

假定所有缺少的case项都是“无关”项，使逻辑更为优化并避免产生锁存器。

## 条件完全的例外

有时使用了case full指示，case语句也可能综合出latch。

下面的描述综合时产生了一个latch。

要求：条件语句中各分支的赋值对象一致。

```
module select (
    input wire [1: 0] sl,
    output reg a, b
);

    always @(*)
        case (sl) // synopsys synthesis case = full
            2'b00: begin a = 0; b = 0; end
            2'b01: begin a = 1; b = 1; end
            2'b10: begin a = 0; b = 1; end
            2'b11: begin a = 1; b = 0; end
            default: begin a = 'bx; b = 'bx; end
        endcase
    endmodule
```

```
module select (
    input wire sl, i1, i2,
    output reg a, b
);

    always @(*)
        if(sl)
            a = i1;
        else
            b = i2;
    endmodule
```

注：条件完全语句指的是同一组信号

## 带复位、置位的锁存器latch的描述示例

下面的例子给出了一个复杂一些的复位分支。由于是一个latch，因此敏感表是完全的。

```
module latch (
    input wire enable, d, set, clr,
    output reg q
);

    always @(*)
        begin
            if (set)
                q = 1;
            else if (clr)
                q = 0;
            else if (enable)
                q = d;
        end
    endmodule
```

## 异端

```
module latch (
    input wire enable ,
    d ,
    output reg q
```

```

);
  always @( *)
    q = enable ? d : q;
endmodule

module latch (
  input wire enable ,
  d ,
  output wire q
);
  assign q = enable ? d : q;
endmodule

module latch (
  input wire enable ,
  d ,
  output reg q
);
  always @( *)
    if( enable )
      q = d ;
    else
      q = q;
endmodule

```

## 条件互斥与条件非互斥

### 条件互斥

```

module tmp(
  input wire a, b, c, d,
  input wire [1:0] s,
  output reg o
);

  always @*(a, b, c, d, s)
    if(s == 2'b00)
      o = a;
    else if(s == 2'b01)
      o = b;
    else if(s == 2'b10)
      o = c;
    else if(s == 2'b11)
      o = d;
    else
      o = 1'bx;
endmodule

```

### 非条件互斥

```

module tmp(
  input wire a, b, c, d,
  input wire e,
  output reg o
);

  always @*(a, b, c, d, e)
    if( a )
      o = c;
    else if( b )
      o = d;
    else
      o = e;
endmodule

```

注：条件互斥产生并行电路，条件不互斥产生带优先级的串行电路

## 条件非互斥的if语句

### 例a if语句

```
module single_if(a, b, c, d, sel, z);
    input a, b, c, d;
    input [3:0] sel;
    output z;
    reg z;

    always @(*) begin
        if (sel[3])      z = d;
        else if (sel[2]) z = c;
        else if (sel[1]) z = b;
        else if (sel[0]) z = a;
        else             z = 0;
    end
endmodule
```

推荐方式

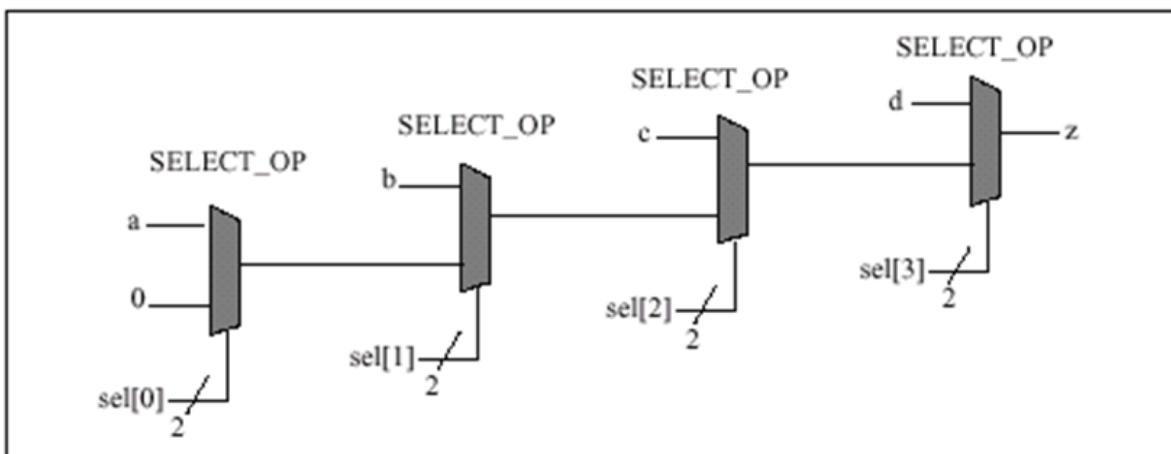
### 例b case语句

```
module case1(a, b, c, d, sel, z);
    input a, b, c, d;
    input [3:0] sel;
    output z;
    reg z;

    always @(*) begin
        casex (sel)
            4'b1xxx: z = d;
            4'b01xx: z = c;
            4'b001x: z = b;
            4'b0001: z = a;
            default: z = 1'b0;
        endcase
    end
endmodule
```

注意代码  
的优先级

## 电路结构

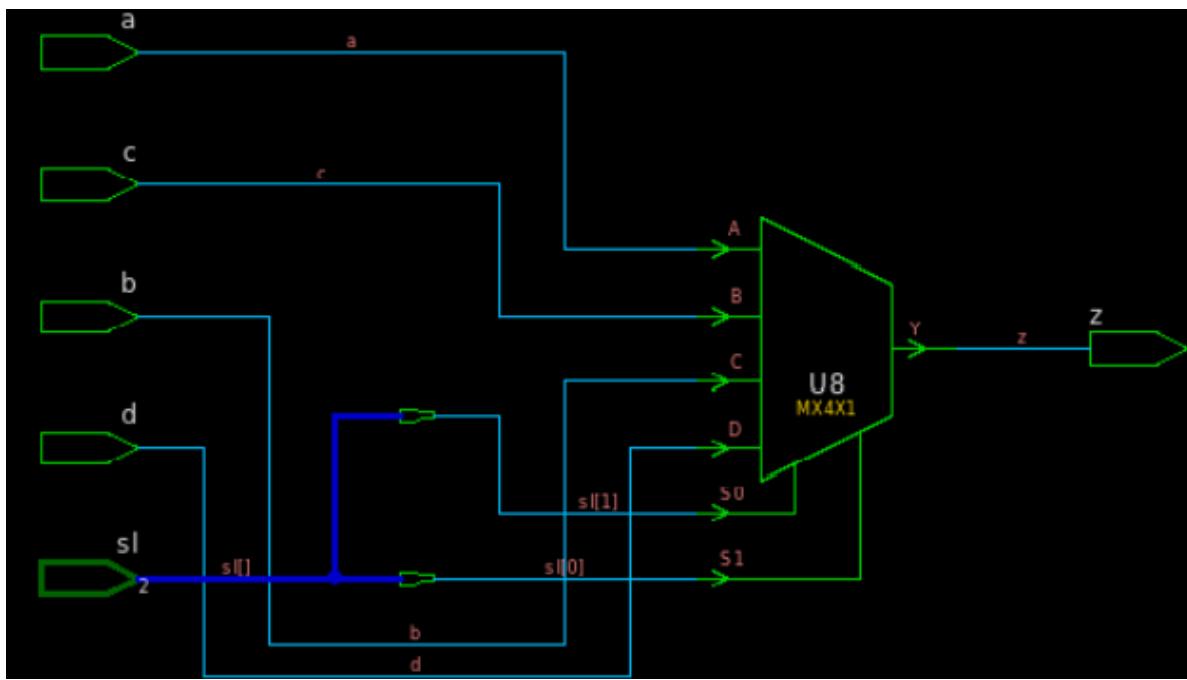


## 互斥的非互斥表达:

```
module single_if(a, b, c, d, s1, z);
    input a, b, c, d;
    input [1:0] s1;
    output reg z;
    wire [3:0] sel = s1 == 2'b11 ? 4'b1000:
                    s1 == 2'b10 ? 4'b0100:
                    s1 == 2'b01 ? 4'b0010:
                                  4'b0001;

    always @(*) begin
        if (sel[3])      z = d;
        else if (sel[2]) z = c;
        else if (sel[1]) z = b;
        else if (sel[0]) z = a;
        else             z = 0;
    end
endmodule
```

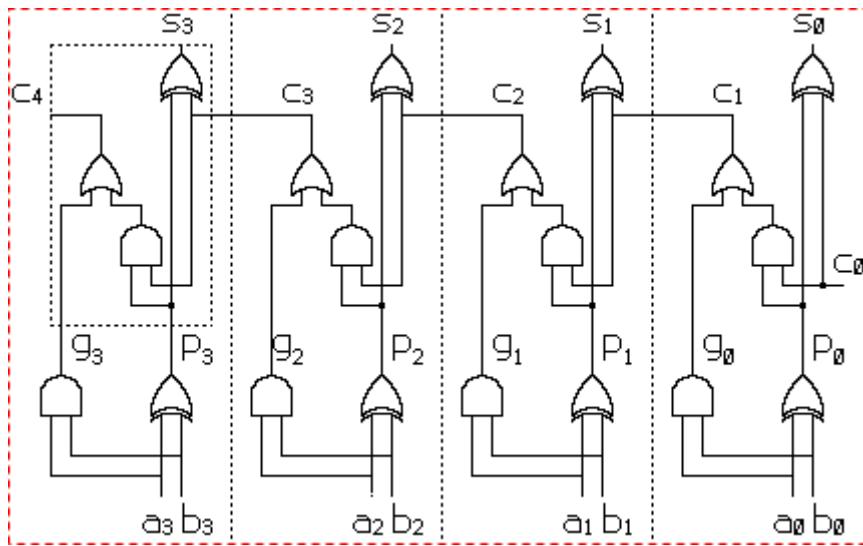
```
endmodule
```



总结：If语句条件完全：逻辑组合电路，不完全：锁存器；如果完全，互斥，并行电路；完全但是不互斥，串行带优先级电路

## for 循环语句

```
module adder_rpl(a, b, ci, co, s);
    parameter n = 4;
    input wire [n-1 : 0] a, b;
    input wire ci;
    output wire co;
    output reg [n-1 : 0] s;
    reg [n-1 : 0] g, p;
    reg [n : 0] c;
    integer i;
    assign co = c[n];
    always @(*) begin
        c[0] = ci;
        for(i = 0; i < n; i = i+1) begin
            g[i] = a[i] & b[i];
            p[i] = a[i] ^ b[i];
            c[i+1] = g[i] | (p[i] & c[i]);
            s[i] = p[i] ^ c[i];
        end
    end
endmodule
```

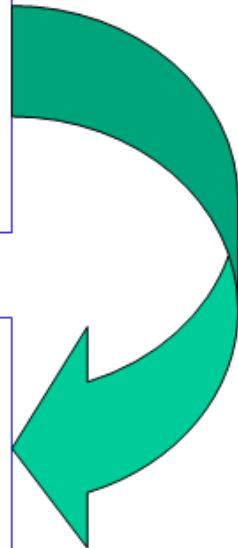


## 非结构化的for循环

综合工具处理循环的方法是将循环内的结构重复。在循环中包含不变化的表达式会使综合工具花很多时间优化这些冗余逻辑。

```
for( i =0; i<4; i=i+1) begin
    sig1 = sig2; //无变化的语句
    data_out[i] = data_in[i];
end
```

```
sig1 = sig2; //无变化的语句
for( i =0; i<4; i=i+1)
    data_out[i] = data_in[i];
```



## 同步块描述风格

```
module DFF (q, qb, d, clk, clr);
    output q, qb;
    input d, // input data
          clk, /*input clock */ clr;
    reg q;
    wire qb, d, clk, clr;

    assign qb = !q;

    always @(posedge clk) ;
    begin
        过程赋值语句;
        高级描述语句
        if语句;
        case语句;
    end
endmodule
```

```

    循环语句;
end
endmodule

```

- 在一个控制信号的单个边沿上进入的过程块产生同步逻辑。这种过程块称为**同步块**。

```
always @(posedge clk)
```

```
    q <= d;
```

- **同步块会产生D触发器**: 并不是所有在同步块中赋值的信号都会产生D触发器，这依赖于描述风格
- 同步块RTL代码中使用**非阻塞赋值**
- 组合块的RTL代码中使用**阻塞赋值**

## 阻塞、非阻塞语句区别

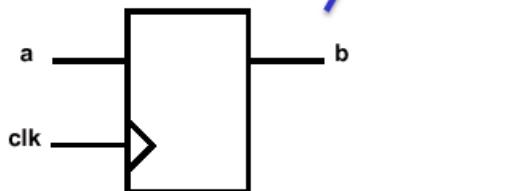
非阻塞赋值语句并行执行，因此临时变量不可避免地在一个周期中被赋值，在下一个周期中被采样。

**使用阻塞赋值，此描述综合出一个D flip-flop:**

```

module bloc (clk, a, b);
    input clk, a;
    output b;
    reg y;
    reg b;
    always @(posedge clk)
    begin
        y = a;
        b = y;
    end
endmodule

```

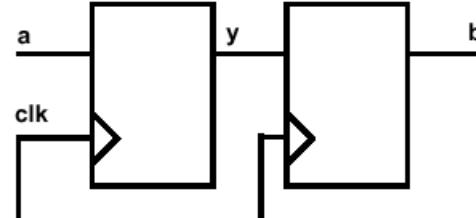


**使用非阻塞赋值，此描述将综合出两个D Flip-flop。**

```

module nonbloc (clk, a, b);
    input clk, a;
    output b;
    reg y;
    reg b;
    always @(posedge clk)
    begin
        y <= a;
        b <= y;
    end
endmodule

```



## 非阻塞过程赋值区别

```

module dffa(clk, a, b, c);
    input wire clk;
    input wire q;
    output reg b, c;

    always @ ( posedge clk)
        c <= b;

    always @(posedge clk)
        b <= a;

endmodule

```

```

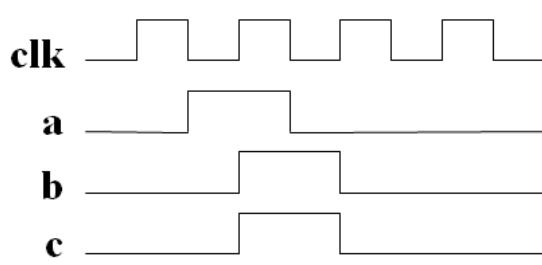
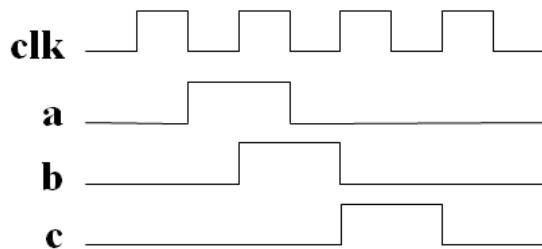
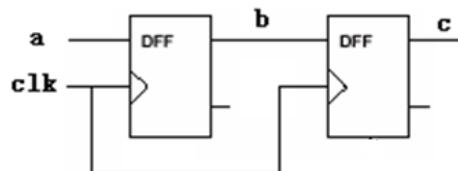
module dffa(clk, a, b, c);
    input wire clk;
    input wire a;
    output reg b, c;

    always @ ( posedge clk)
        c = b;

    always @(posedge clk)
        b = a;

endmodule

```



```

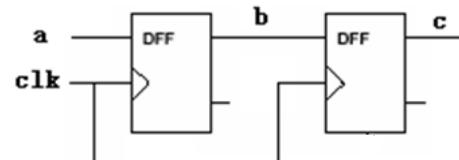
module dffa(clk, a, b, c);
    input wire clk;
    input wire a;
    output reg b, c;

    always @ ( posedge clk)
        b = a;

    always @(posedge clk)
        c = b;

endmodule

```



仿真器计算所有RHS表达式的值，保存结果，并进行调度在时序控制指定时间的赋值。

非阻塞赋值：信号先采样，后变化

阻塞与非阻塞语句：不要混合使用

```

module dffa(clk, a, e);
    input wire clk;
    input wire a;
    output reg e;

    reg b, c, d;

    always @( b ) begin
        c = b;
        d = c;
        e = d;
    end

    always @ (posedge clk)
        b <= a;

endmodule

```

```

module dffa(clk, a, e);
    input wire clk;
    input wire a;
    output reg e;

    reg b, c, d;

    always @( b ) begin
        c = b;
        d <= c;
        e = d;
    end

    always @ (posedge clk)
        b <= a;

endmodule

```

X

### 同步块中的条件语句-带使能的寄存器

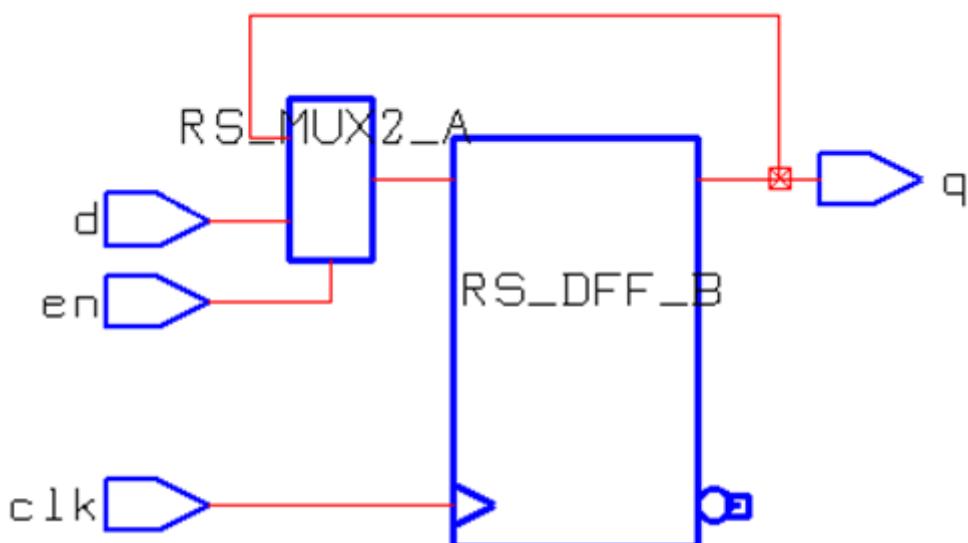
- 条件不完全的条件语句产生带使能端的D触发器；条件完全的条件语句在D触发器前产生选择逻辑。
- 在寄存器的描述中，敏感列表是不完全的。综合时会产生warning

```

module dffn (
    input   wire  d    ,
    clk    ,
    en    ,
    output reg   q);

    always @ (posedge clk)
        if (en)
            q <= d;
endmodule

```



## 复位

复位是可综合编码风格的重要环节。状态机中一般都有复位。

### 同步复位

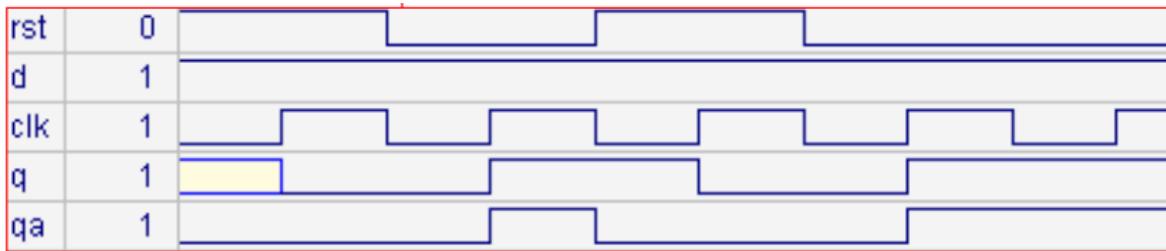
```
module sync(
    input wire ck, d, rst,
    output reg q
);
always @(posedge ck)
    if (rst)
        q <= 0;
    else
        q <= d;
endmodule
```

### 同步块的异步复位

```
module async(
    input wire ck, d, r,
    output reg q
);
always @(posedge ck, posedge r)
    if (r)
        q <= 0;
    else
        q <= d;
endmodule
```

- 同步复位描述：在同步块内，当复位信号有效时，进行复位操作；当复位信号无效时，执行该块的同步行为。如果将复位信号作为条件语句的条件，且在第一个分支中进行复位，综合工具可以更容易的识别复位信号。
- 异步复位：在同步块的敏感表中包含复位信号的激活边沿。在块内，复位描述方式与同步方式相同。

```
'timescale 1ns / 1ns
module reset_tb;
    reg clk, d, rst ;
    wire q, qa;
    sync u0 ( clk, d, rst , q );
    async u1 ( clk, d, rst , qa );
    always #5 clk = !clk;
    initial begin
        clk = 0; d = 1;
        rst = 1;
        @(posedge clk);
        @(negedge clk);
        rst = 0;
        @(posedge clk);
        @(negedge clk);
        rst = 1;
        d = 1;
        @(negedge clk);
        rst = 0;
        d = 1;
        @(negedge clk);
        #10 $finish;
    end
endmodule
```



低电平有效的复位

```
module sync(
    input wire clk, d, rst,
    output reg q
);

    always @(*)
        if ( !rst )
            q <= 0;
        else
            q <= d;
endmodule
```

高电平有效的复位

```
module async(
    input wire clk, d, rst,
    output reg q
);

    always @(*)
        if ( rst )
            q <= 0;
        else
            q <= d;
endmodule
```

## 不好的复位描述方式

下面的异步复位描述（异步复位和同步块分开）是一种很不好的描述风格，并且有的综合工具不支持。在仿真中，如果r和ck在同一时刻改变，则结果不可确定。

### 不好的异步复位描述方式

```
module async( q, ck, r, d);
    input ck, d, r;
    output q;
    reg q;

    always @(*)
        if( !r ) q <= d;

    always @(*)
        q <= 0;

endmodule
```

## 综合工具不能胜任的工作

- 时钟树
- 复杂的时钟方案

- 组合逻辑反馈循环和脉冲发生器
- 存储器, IO
- 专用宏单元
- 做的和人工做的一样好

综合工具善于优化组合逻辑。但设计中有很大一部分不是组合逻辑。

- 例如, 时钟树。时钟树是全局的、芯片范围的问题。在没有版图布局信息的情况下, 要给出较优的结果, 综合工具对块的大小有一定的限制。
- 综合工具不能很好地处理复杂时钟。通常, 只允许要综合的块含有一个时钟。但设计中经常使用两相时钟或在双沿时钟。
- 综合工具不易实现脉冲产生逻辑, 如单个脉冲, 或结果依赖于反馈路径延迟的组合反馈逻辑。对这种情况, 插入延迟元件使一个信号推迟到达的效果并不好。
- 不能用综合产生大块存储器, 因为综合工具会用flip-flop实现。
- 不是所有的综合工具都能很好地从工艺库里挑选大的单元或宏单元, 这需要用户人工实例化。一些宏单元, 例如大的结构规则的数据通路元件, 最好使用生产商提供的硅编译器产生。
- 综合工具不保证产生最小结果。通常综合结果不如人工结果, 只要你有足够的时间。

## 代码练习

### 练习1

#### CLA Adder 4bit RTL代码

```

`timescale 1ns/1ns
module CLA_Adder(
  input wire [3:0] a ,
  b ,
  input wire ci ,
  output wire [3:0] s ,
  output wire co
);
  wire [3:0] p , g;
  wire [3:0] c;
  assign g = a&b ;
  assign p = a^b ;
  assign c[0] = g[0]|(p[0]&ci) ;
  assign c[1] = g[1]|(p[1]&g[0])|(p[1]&p[0]&ci) ;
  assign c[2] = g[2]|(p[2]&g[1])|(p[2]&p[1]&g[0])|(p[2]&p[1]&p[0]&ci) ;
  assign c[3] = g[3]|(p[3]&g[2])|(p[3]&p[2]&g[1])|(p[3]&p[2]&p[1]&g[0])|
(p[3]&p[2]&p[1]&p[0]&ci) ;
  assign co = c[3] ;
  assign s[3] = p[3]^c[2];
  assign s[2] = p[2]^c[1];
  assign s[1] = p[1]^c[0];
  assign s[0] = p[0]^ci ;
endmodule

```

#### ALU 4bit CLA RTL代码

```

module ALU_CLA4_G(
  input wire [3:0] a ,
  b ,
  input wire ci ,
  input wire M ,

```

```

input wire [3:0] S ,
output wire [3:0] Do ,
output wire co ,
output wire V ,
Z
);
reg [3 : 0] g , p ;
wire [3 : 0] G , P ;
wire [4 : 0] c ;
integer i;
always @(*)
begin
  for( i = 0; i < 4; i = i + 1) begin
    g[i] = ( S[3] & a[i] & b[i] ) ||
            ( S[2] & a[i] & !b[i] ) ||
            !M;
    p[i] = !( ( S[3] & a[i] & b[i] ) ||
               ( S[2] & a[i] & !b[i] ) ||
               ( S[1] & !a[i] & b[i] ) ||
               ( S[0] & !a[i] & !b[i] ) );
  end

  assign G[0] = g[0] ;
  assign G[1] = g[1] | ( p[1] & G[0] ) ;
  assign G[2] = g[2] | ( p[2] & G[1] ) ;
  assign G[3] = g[3] | ( p[3] & G[2] ) ;
  assign P[0] = p[0] ;
  assign P[1] = p[1] & P[0] ;
  assign P[2] = p[2] & P[1] ;
  assign P[3] = p[3] & P[2] ;
  assign c[0] = ci ;
  assign c[4 : 1] = G | ( P & {4{c[0]}} );
  assign Do = p ^ c[3 : 0];
  assign co = c[4];
  assign V = c[4] ^ c[3];
  assign Z = ! ( | Do);
endmodule

```

## ALU CLA 4bit 测试程序

```

`timescale 1ns/1ns
module ALU_CLA4_G_tb();
reg [3:0] a ,
b ;
reg ci;
reg M ;
reg [3:0] S ;
wire [3:0] Do ;
wire co;
wire V ,
Z ;
ALU_CLA4_G uG(
.a ( a ),
.b ( b ),
.ci ( ci ),
.M ( M ),
.S ( S ),

```

```

.Do ( Do ),
.co ( co ),
.v ( v ),
.z ( z )
);

integer i;
reg error ;
integer func ;
reg [3 : 0] DO_ex;
reg [3 : 0] opA,
            opB;
initial begin
    //logic test
    func = 0;
    error = 0;
    a = 4'b1100;
    b = 4'b1010;
    ci= 1'b1;
    M = 1'b0;
    for(i = 0; i < 16; i = i+1) begin
        S = i;
        #10 if( Do != DO_ex)
            error = 1;
        else
            error = 0;
    end

    //add test
    func = func + 1;
    S = 4'b1001; ci = 1'b0; M = 1'b1;
    for(i = 0; i < 256; i = i+1) begin
        {a, b} = i;
        #10 if(Do != DO_ex)
            error = 1;
        else
            error = 0;
    end

    //sub test
    func = func + 1;
    S = 4'b0110; ci = 1'b1; M = 1'b1;
    a = 4'b1100;
    b = 4'b1010;
    #10 if(Do != DO_ex)
        error = 1;
    else
        error = 0;

    //carry test
    func = func + 1;
    S = 4'b1001; ci = 1'b0; M = 1'b1;
    a = 4'b1100;
    b = 4'b0110;
    #10 if(co == 1'b0)
        error = 1;
    else
        error = 0;

```

```

a = 4'b0100;
b = 4'b0110;
#10 if(co == 1'b1)
    error = 1;
else
    error = 0;

//v test
//a + b
// a = 4'b0111; b = 4'b0010 v = 1
// a = 4'b0111; b = 4'b1010 v = 0
// a = 4'b1100; b = 4'b1010 v = 1
// a = 4'b1100; b = 4'b0110 v = 0
func = func + 1;
S = 4'b1001; ci = 1'b0; M = 1'b1;
a = 4'b0111;
b = 4'b0010;
#10 if(v == 1'b0)
    error = 1;
else
    error = 0;
a = 4'b0111;
b = 4'b1010;
#10 if(v == 1'b1)
    error = 1;
else
    error = 0;
a = 4'b1100;
b = 4'b1010;
#10 if(v == 1'b0)
    error = 1;
else
    error = 0;

a = 4'b1100;
b = 4'b0110;
#10 if(v == 1'b1)
    error = 1;
else
    error = 0;
//a - b
// a = 4'b0111; b = 4'b0010 v = 0
// a = 4'b0111; b = 4'b1010 v = 1
// a = 4'b1100; b = 4'b1010 v = 0
// a = 4'b1100; b = 4'b0110 v = 1
func = func + 1;
S = 4'b0110; ci = 1'b1; M = 1'b1;
a = 4'b0111;
b = 4'b0010;
#10 if(v == 1'b1)
    error = 1;
else
    error = 0;
a = 4'b0111;
b = 4'b1010;
#10 if(v == 1'b0)
    error = 1;
else

```

```

    error = 0;

a = 4'b1100;
b = 4'b1010;
#10 if(v == 1'b1)
    error = 1;
else
    error = 0;
a = 4'b1100;
b = 4'b0110;
#10 if(v == 1'b0)
    error = 1;
else
    error = 0;
#10 $finish;
end

//DO expected
always @(*) begin
    opA = a;
    opB = b;
    case({S, ci, M})
//0000 10 0 置全0
//0001 10 !A & !B nor
//0010 10 !A & B notand
//0011 10 !A not A
    6'b0000_10: DO_ex = 32'b0;
    6'b0001_10: DO_ex = ~opA & ~opB;
    6'b0010_10: DO_ex = ~opA & opB;
    6'b0011_10: DO_ex = ~opA;
//0100 10 A & !B andnot
//0101 10 !B not B
//0110 10 A&!B | !A&B xor
//0111 10 !A | !B nand
    6'b0100_10: DO_ex = opA & ~opB;
    6'b0101_10: DO_ex = ~opB;
    6'b0110_10: DO_ex = opA ^ opB;
    6'b0111_10: DO_ex = ~opA | ~opB;
//1000 10 A & B and
//1001 10 A&B | !A & !B xnor
//1010 10 B 传送B
//1011 10 A&B | !A&B | !A&!B notor

    6'b1000_10: DO_ex = opA & opB;
    6'b1001_10: DO_ex = opA ~^ opB;
    6'b1010_10: DO_ex = opB;
    6'b1011_10: DO_ex = ~(opA & ~opB);
//1100 10 A 传送A
//1101 10 A&B | A&!B | !A&!B or not
//1110 10 A&B | A&!B | !A&B or
//1111 10 1 置全1
    6'b1100_10: DO_ex = opA ;
    6'b1101_10: DO_ex = ~(~opA & opB);
    6'b1110_10: DO_ex = opA | opB;
    6'b1111_10: DO_ex = {32{1'b1}};
//1001 01 A ^ B ^ C 加法
//0110 11 ( A ~^ B) ^ C 减法
    6'b1001_01: DO_ex = opA + opB;

```

```

6'b0110_11: DO_ex = opA - opB;
      default: DO_ex = 32'bx;
endcase
end
endmodule

```

## ALU CLA 32bit RTL代码

```

`timescale 1ns/1ns
module ALU_CLA32 (
    input wire [31 : 0] opA , //操作数A
    opB , //操作数B
    input wire [3 : 0] S , //工作模式
    input wire M , //逻辑操作
    Cin , //进位输入
    output wire [31 : 0] DO , //数据输出
    output wire N ,
    Z ,
    V ,
    C
);
wire [8 : 0] Co;
wire [7 : 0] Zo, Vo;
assign Co[0] = Cin;

generate
genvar i;
for( i = 0; i < 8; i = i + 1 ) begin : u
    ALU_CLA4_G uALU_CLA4_G(
        .a ( opA[i*4 + 3 : i*4] ),
        .b ( opB[i*4 + 3 : i*4] ),
        .ci ( Co[i] ),
        .M ( M ),
        .S ( S ),
        .Do ( DO[i*4 + 3 : i*4] ),
        .co ( Co[ i+1 ] ),
        .V ( Vo[ i ] ),
        .Z ( Zo[ i ] )
    );
end
endgenerate
assign V = Vo[ 7 ];
assign N = DO[31];
assign C = Co[ 8 ];
assign Z = & Zo;
endmodule

```

## ALU CLA 32bit 测试程序

```

`timescale 1ns/1ns
module ALU_CLA32_tb();
reg [31 : 0] opA ,
opB ;
reg [3 : 0] S ;

```

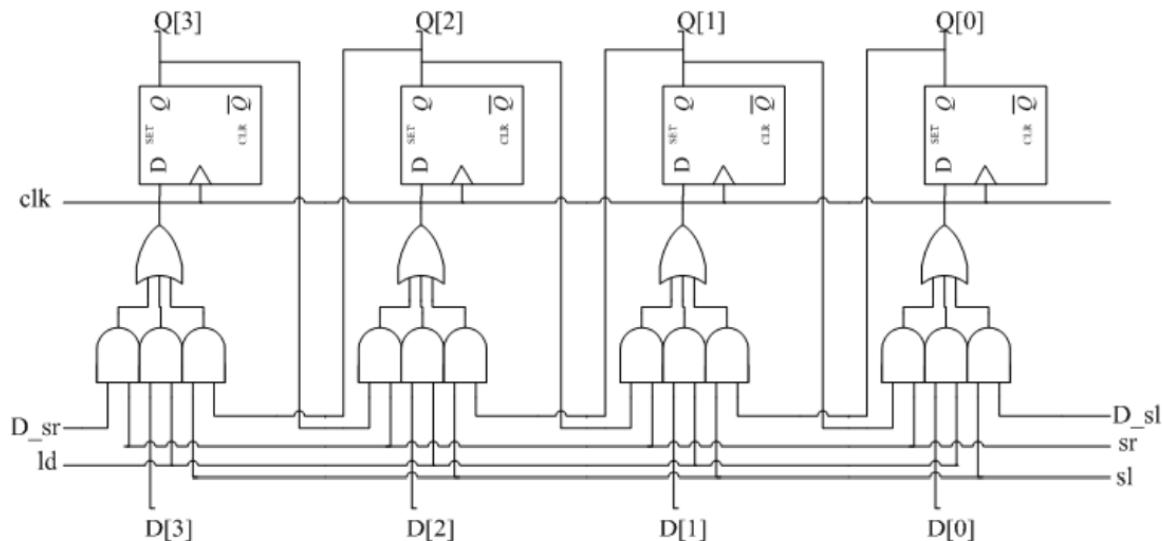
```

reg M ,
cin ;
wire [31 : 0] DO ;
wire N , Z ,
V , C ;
ALU_CLA32 uALU32(
.opA ( opA ), //操作数A
.opB ( opB ), //操作数B
.S ( S ), //工作模式选择信号
.M ( M ), //逻辑操作控制信号
.Cin ( Cin ), //进位输入信号
.DO ( DO ), //数据输出
.N ( N ),
.Z ( Z ),
.V ( V ),
.C ( C )
);

initial begin
opA = 32'hFFFF_FFFF;
opB = 32'h0000_0001;
S = 4'b1001;
M = 1'b1;
Cin = 1'b0;
#10 opA = 32'hFFFF_FFFF;
opB = 32'h0000_0000;
#10 opA = 32'hFFFF_FFFF;
opB = 32'hFFFF_FFFF;
#10 opA = 32'h7FFF_FFFF;
opB = 32'h0FFF_FFFF;
#10 opA = 32'h3FFF_FFFF;
opB = 32'h0FFF_FFFF;
#10 $finish;
end
endmodule

```

## 练习2

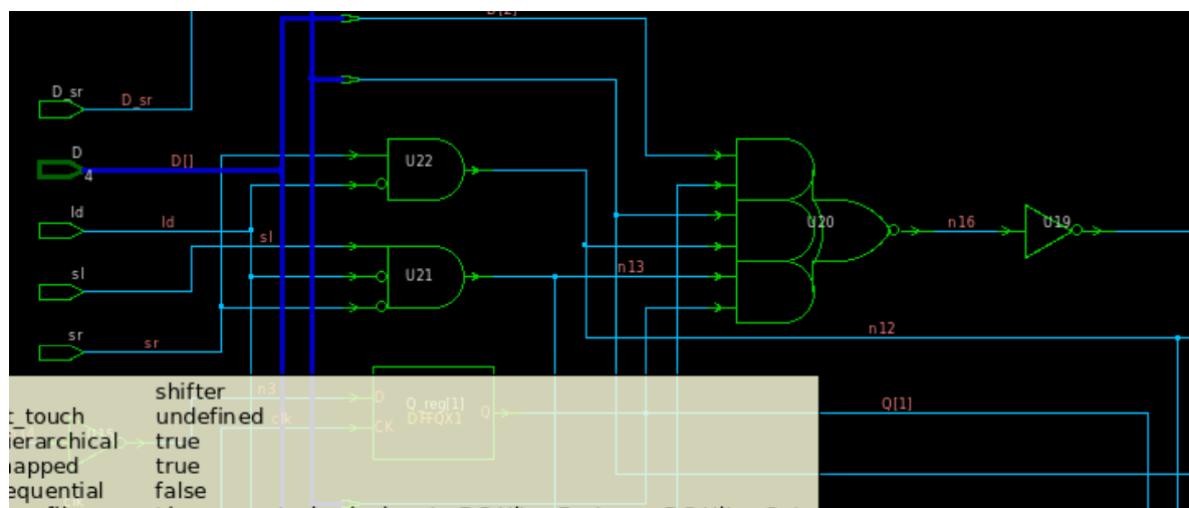


## 移位器RTL代码

```

`timescale 1ns/1ns
module shifter (
    input wire clk ,
    input wire D_sr ,
    sr ,
    input wire D_s1 ,
    s1 ,
    input wire [n-1 : 0] D ,
    input wire ld ,
    output reg [n-1 : 0] Q
);
    always @(posedge clk)
        if ( ld )
            Q <= D;
        else if( sr )
            Q <= {D_sr, Q[3 : 1]};
        else if( s1 )
            Q <= {Q[2 : 0], D_s1 };
        else
            Q <= 0;
endmodule

```

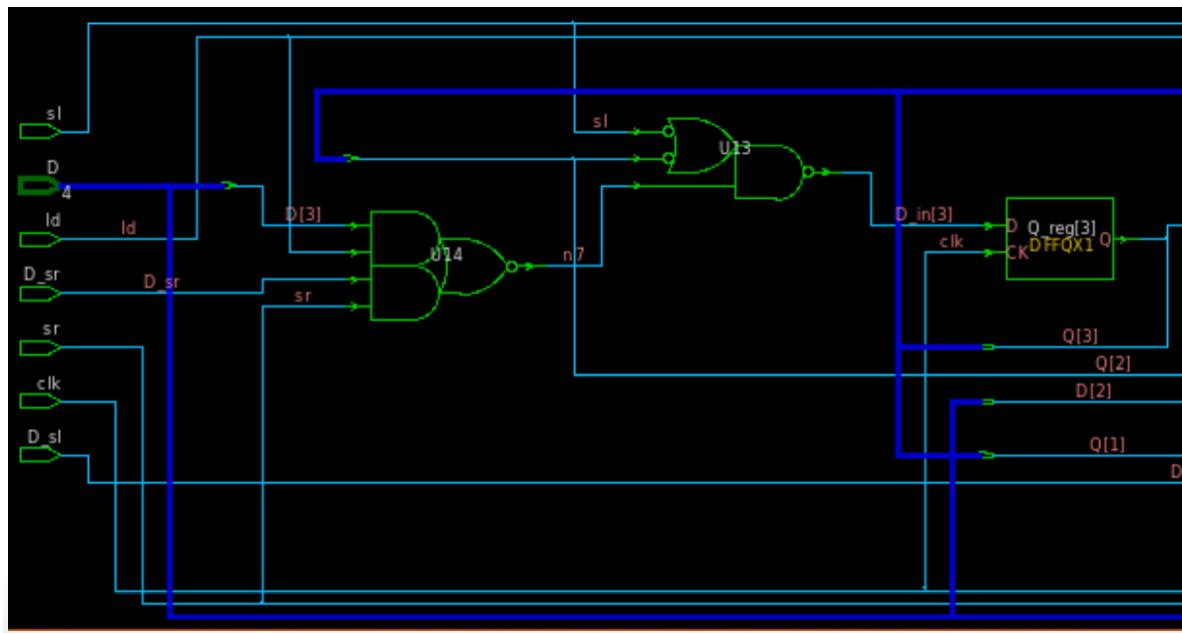


```

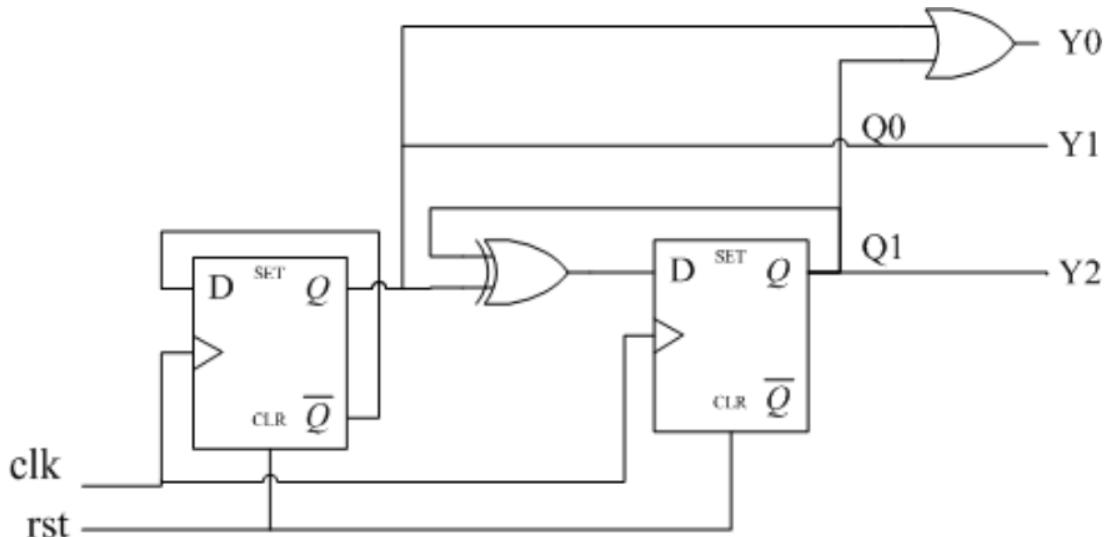
`timescale 1ns/1ns
module shifter (
    input wire clk ,
    input wire D_sr ,
    sr ,
    input wire D_s1 ,
    s1 ,
    input wire [n-1 : 0] D ,
    input wire ld ,
    output reg [n-1 : 0] Q
);
    wire [n-1 : 0] D_in;
    assign D_in =  ( {4{ld}} & D ) |
                  ( {4{sr}} & {D_sr, Q[n-1 : 1]} ) |
                  ( {4{s1}} & {Q[n-2 : 0], D_s1 } );
    always @(posedge clk)
        Q <= D_in;

```

```
endmodule
```



### 练习3: counter3



### RTL代码

```
`timescale 1ns/1ns
module fsm (
    input wire clk ,
    rst ,
    output wire Y2 ,
    Y1 ,
    Y0
);
always @ (posedge clk, negedge rst)
    if( !rst ) begin
        Q1 <= 0;
        Q0 <= 0;
        end
    else begin
        Q1 <= Q1 ^ Q0;
        Q0 <= ! Q0;
    end
endmodule
```

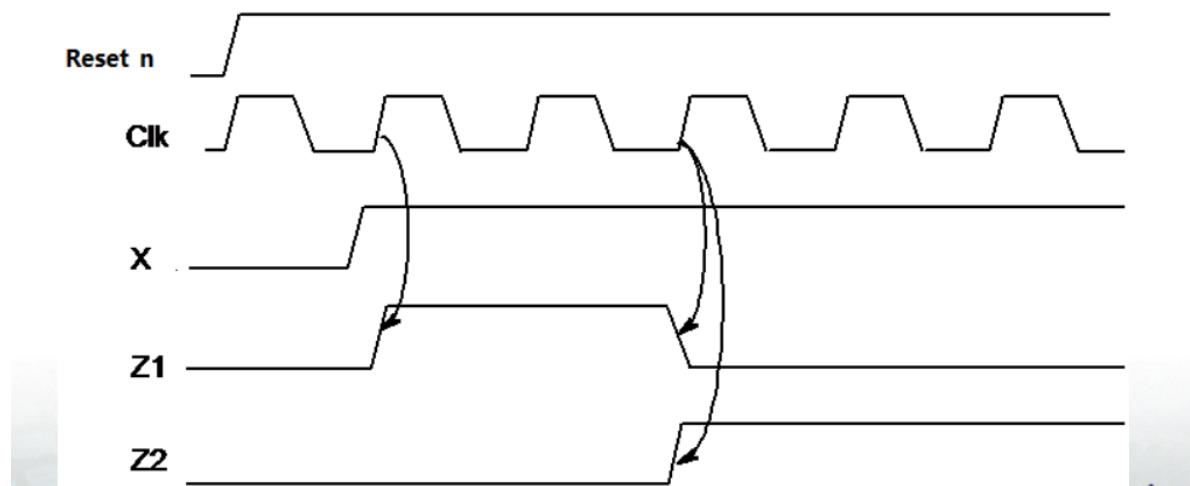
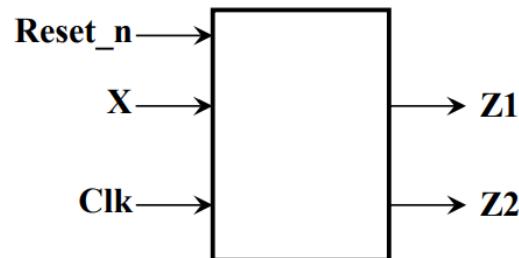
```

end
assign Y2 = Q1;
assign Y1 = Q0;
assign Y0 = Q1 | Q0;
endmodule

```

## 练习 4

一个电路的时序如下图所示  
。请用Verilog语言完成此电  
路设计。**Reset\_n**是低电平有  
效的异步复位信号。



### FSM 方法1

```

`timescale 1ns/1ns
module fsm1 (
  input wire clk ,
  reset_n ,
  X ,
  output wire z1 ,
  z2
);
reg [1: 0] cnt;
always @ (posedge clk, negedge reset_n)
  if( ! reset_n )
    cnt <= 2'b00;
  else if( X ) begin
    if(cnt != 2'b11)
      cnt <= cnt + 1'b1;
  end
  assign z1 = ^ cnt ;
  assign z2 = & cnt ;
endmodule

```

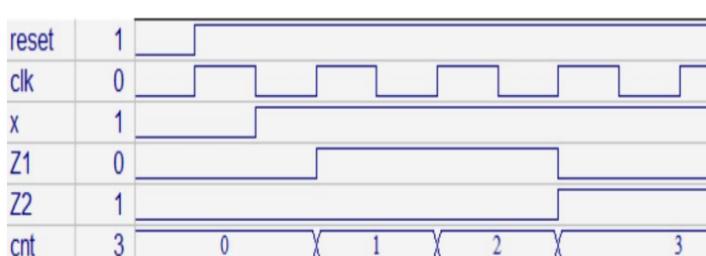
## FSM 测试程序

```

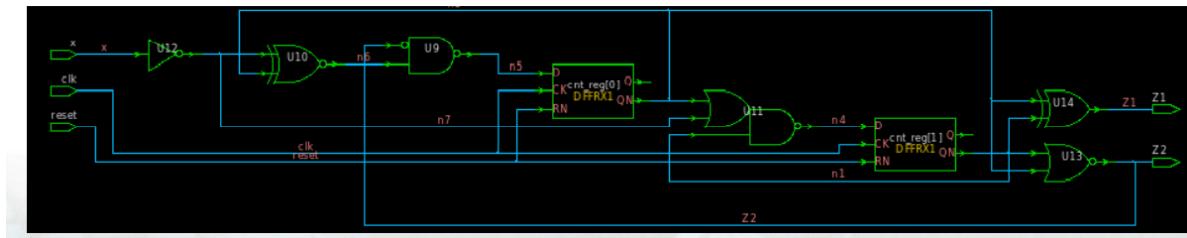
`timescale 1ns/1ns
module fsm1_tb();
reg clk ,
reset ,
x ;
wire z1 ,
z2 ;
fsm3 u fsm3(
.clk (clk),
.reset_n (reset),
.x (x),
.z1 (z1),
.z2 (z2)
);
always #5 clk = !clk;

initial begin
clk = 0;
reset = 0;
x = 0;
@(posedge clk);
reset = 1'b1;
@(negedge clk);
x = 1;
repeat(5) @(posedge clk);
$finish;
end
endmodule

```

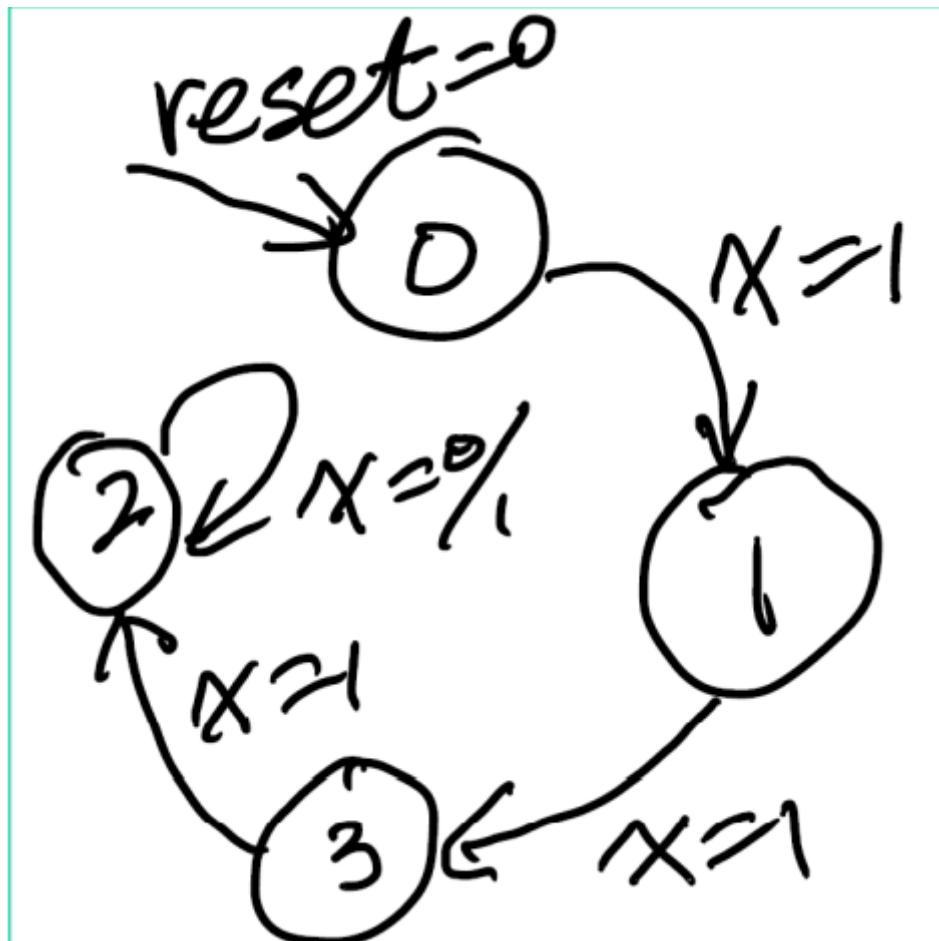


Cell	Reference	Area
U9	NAND2BX1	6.789600
U10	XNOR2X1	11.881800
U11	OAI21XL	6.789600
U12	CLKINVX1	3.394800
U13	NOR2X1	5.092200
U14	XOR2X1	11.881800
cnt_reg[0]	DFFRX1	32.250599
cnt_reg[1]	DFFRX1	32.250599
<b>Total 8 cells</b>		
<b>110.330997</b>		

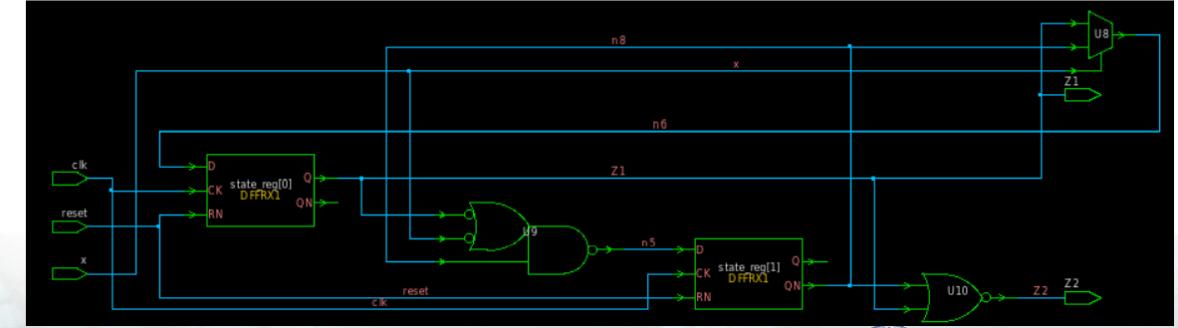


## FSM 方法 2

```
module fsm2 (
    input wire clk
    ,
    reset ,
    x ,
    output wire z1 ,
    z2
);
reg [1: 0] state;
parameter ST0 = 2'b00, ST1 = 2'b01,
ST2 = 2'b11, ST3 = 2'b10;
always @(posedge clk, negedge reset)
    if(!reset)
        state <= ST0;
    else if( x )
        case(state)
            ST0 : state <= ST1;
            ST1 : state <= ST2;
            ST2 : state <= ST3;
            default: state <= ST3;
        endcase
assign z1 = state == ST1 || state == ST2 ;
assign z2 = state == ST3;
endmodule
```



	Cell	Reference	Area
reset	U8	CLKMX2X2	13.579200
clk	U9	OAI2BB1X1	8.487000
x	U10	NOR2X1	5.092200
Z1	state_reg[0]	DFFRX1	32.250599
Z2	state_reg[1]	DFFRX1	32.250599
state	Total 5 cells		91.659598



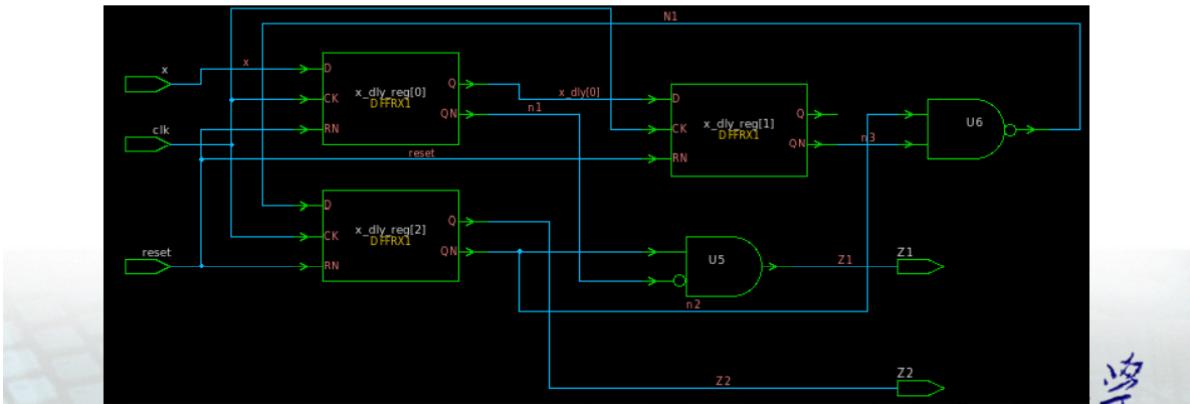
### FSM 方法3

```

module fsm3 (
  input wire clk ,
  reset ,
  x ,
  output wire z1 ,
  z2
);
reg [2: 0] x_dly;
always @ (posedge clk, negedge reset)
  if (!reset)
    x_dly <= 0;
  else begin
    x_dly[0] <= x;
    x_dly[1] <= x_dly[0];
    x_dly[2] <= x_dly[1] | x_dly[2];
  end
assign z1 = x_dly[0] & !x_dly[2];
assign z2 = x_dly[2] ;
endmodule

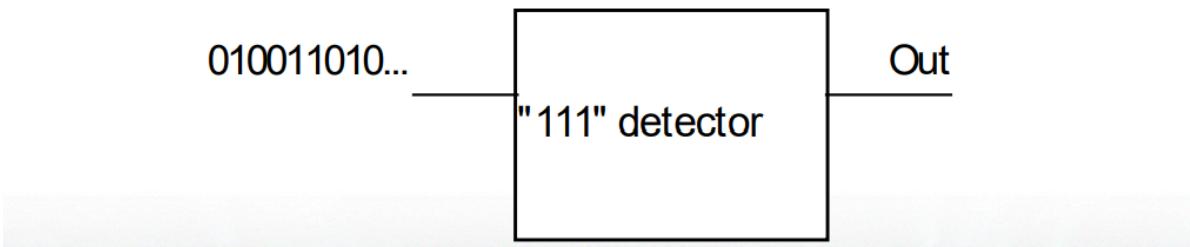
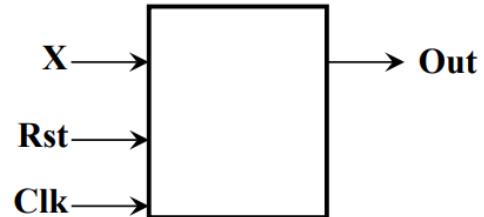
```

	Cell	Reference	Area
reset	U5	NOR2BX1	6.789600
clk	U6	NAND2X1	5.092200
x	x_dly_reg[0]	DFFRX1	32.250599
Z1	x_dly_reg[1]	DFFRX1	32.250599
Z2	x_dly_reg[2]	DFFRX1	32.250599
x_dly			
■ X_...	Total 5 cells		108.633596
■ X_...			
■ X_...			



## 练习 5

试设计一个如图所示的状态机。同步复位信号Rst高有效。复位后输出out为低电平。如果在时钟Clk上升沿探测到输入位串X有三个连续的“1”则在out端输出高电平并保持不变。



### “111”检测

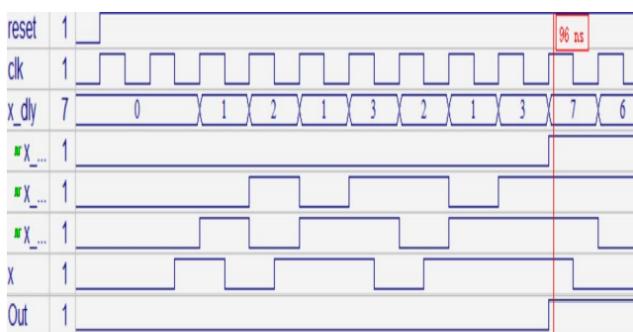
```
module detect111(
  input wire clk ,
  Rst ,
  x ,
  output wire out
);
  reg [2: 0] x_dly;
  always @ (posedge clk)
    if(Rst)
      x_dly <= 0;
```

```

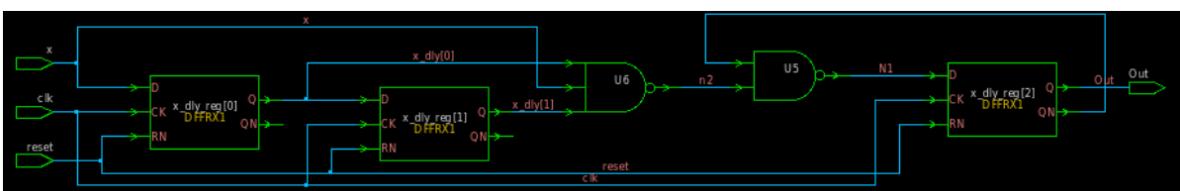
else begin
    x_dly[0] <= x;
    x_dly[1] <= x_dly[0];
    x_dly[2] <= (&x_dly[1:0] & x) | x_dly[2];
end
assign out = x_dly[2] ;
endmodule

`timescale 1ns/1ns
module detect111_tb();
reg clk ,
reset ,
x ;
wire out ;
detect111 udetect(
.clk (clk),
.Rst (reset),
.x (x),
.out (out)
);
always #5 clk = !clk;
reg [9:0] xin;
initial begin
clk = 0;
reset = 0;
xin = 10'b01_0110_1110;
x = 0;
@(posedge clk);
reset = 1'b1;
repeat(12) begin
@(negedge clk);
x = xin[9];
xin = xin << 1;
end
$finish;
end
endmodule

```



Cell	Reference	Area
U5	NAND2X1	5.092200
U6	NAND3X1	6.789600
x_dly_reg[0]	DFFRX1	32.250599
x_dly_reg[1]	DFFRX1	32.250599
x_dly_reg[2]	DFFRX1	32.250599
Total 5 cells		108.633596



```

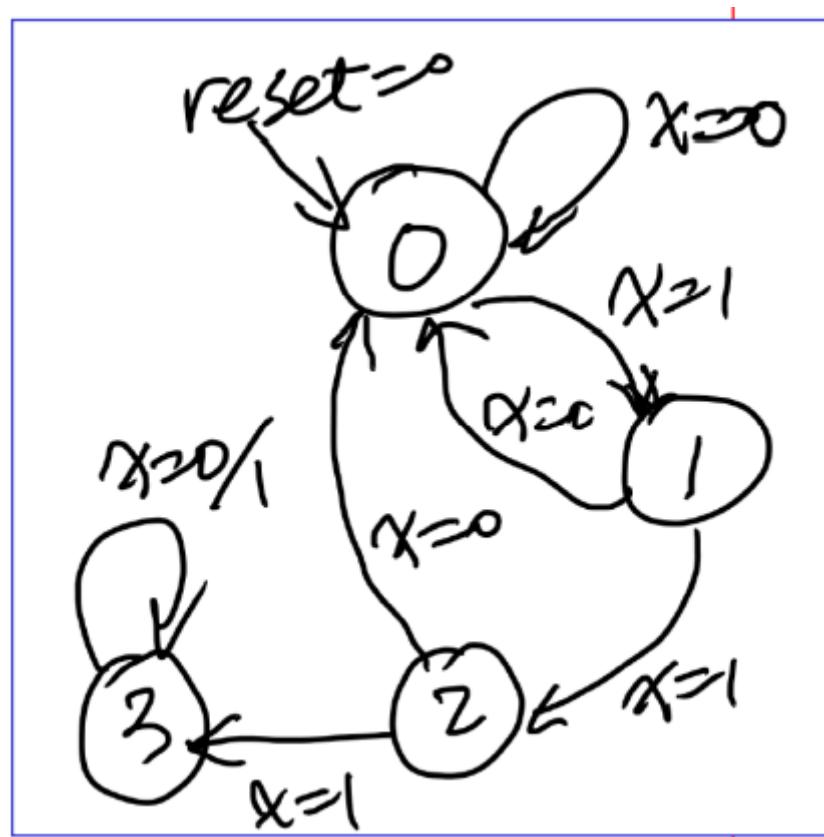
module detect_fsm(
input wire clk ,

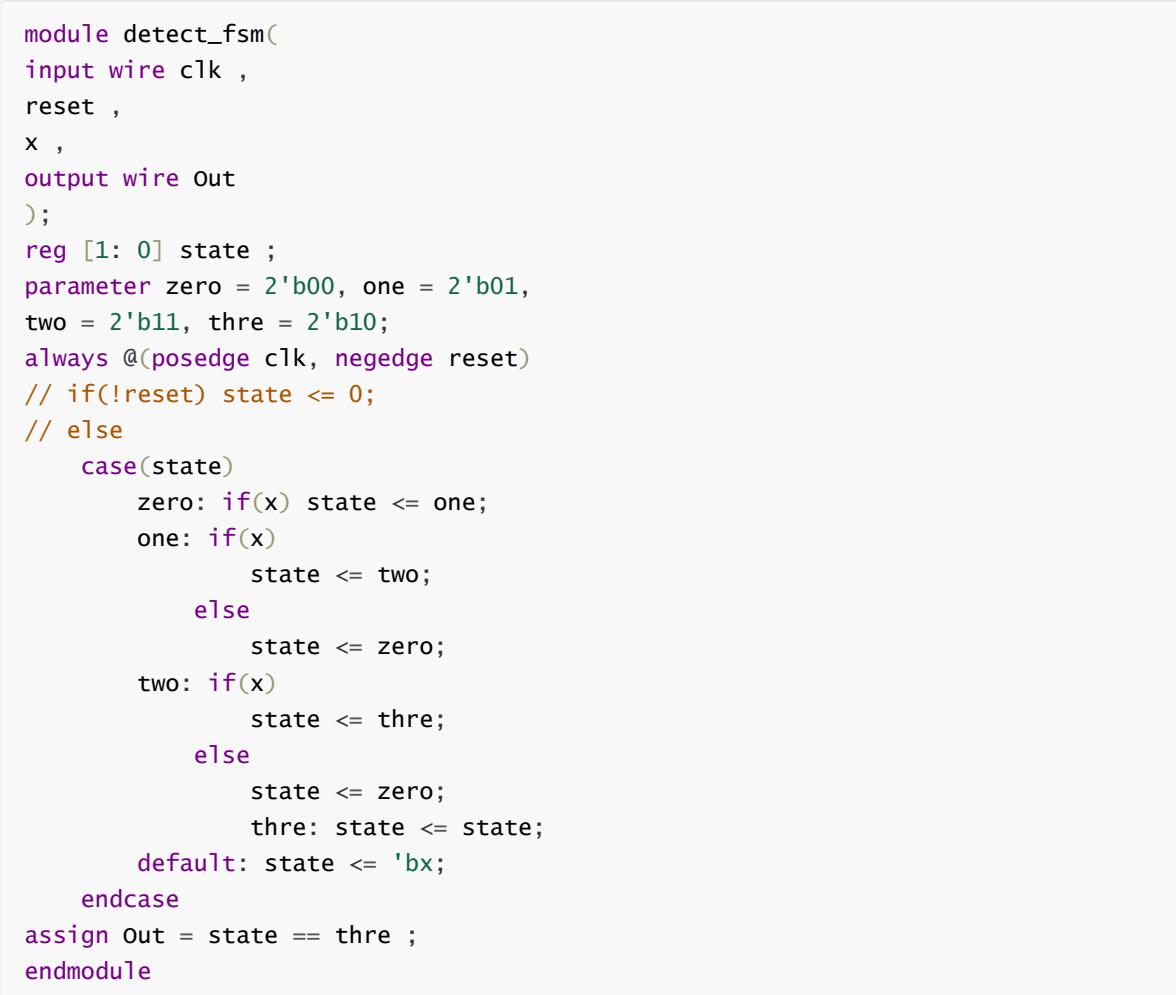
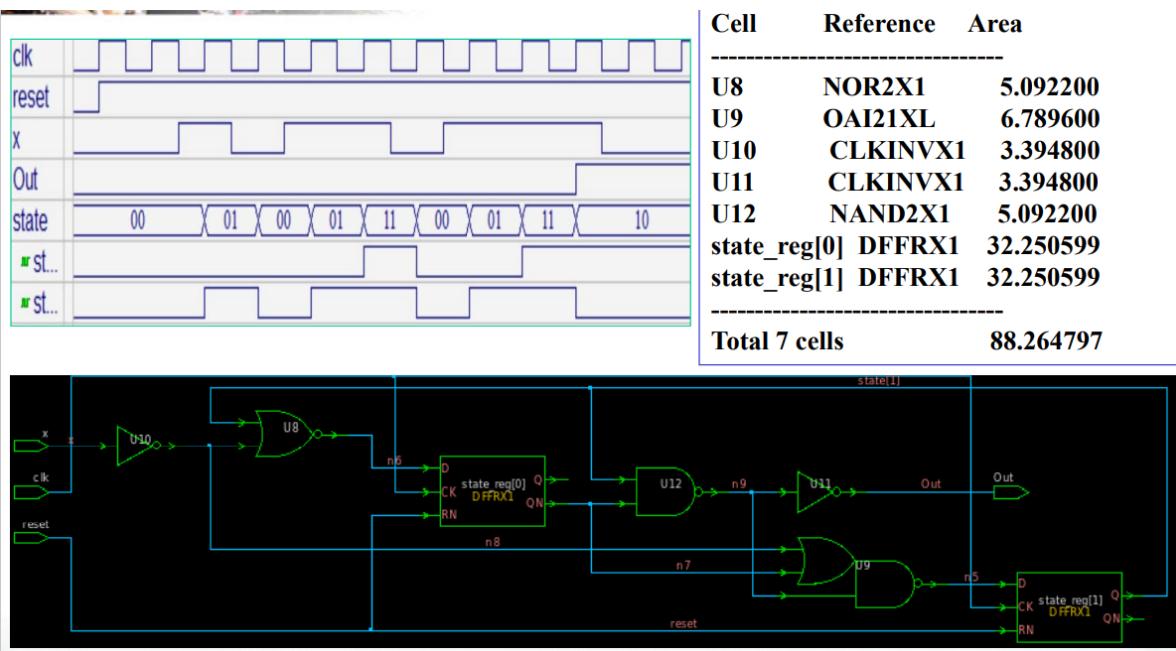
```

```

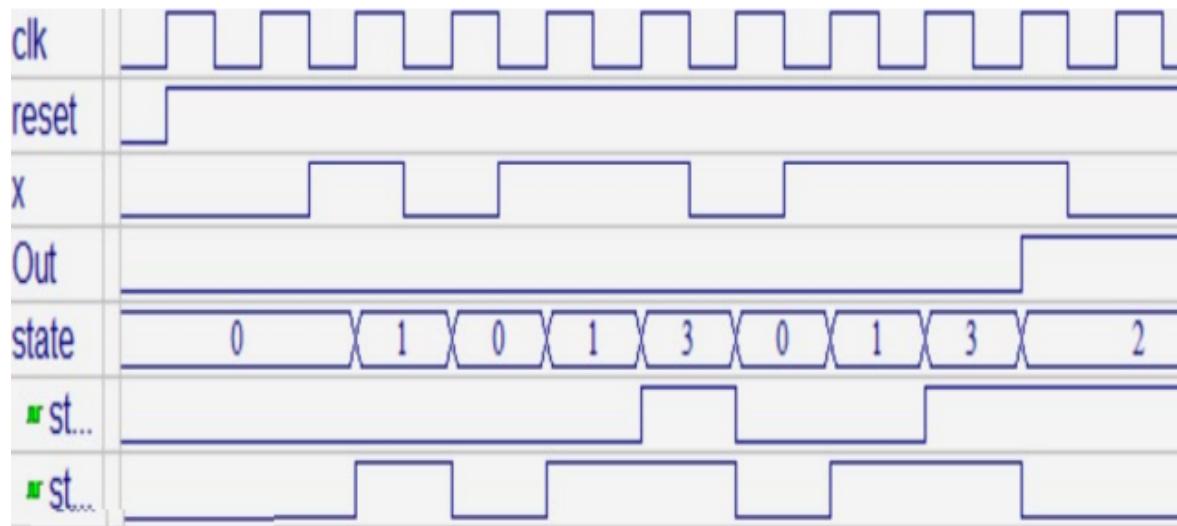
reset ,
x ,
output wire out
);
reg [1: 0] state;
parameter zero = 2'b00, one = 2'b01,
two = 2'b11, thre = 2'b10;
always @(posedge clk, negedge reset)
  if(!reset)
    state <= 0;
  else
    case(state)
      zero: if(x) state <= one;
      one: if(x)
            state <= two;
      else
            state <= zero;
      two: if(x)
            state <= thre;
      else
            state <= zero;
      default: state <= thre;
    endcase
  assign Out = state == thre ;
endmodule

```

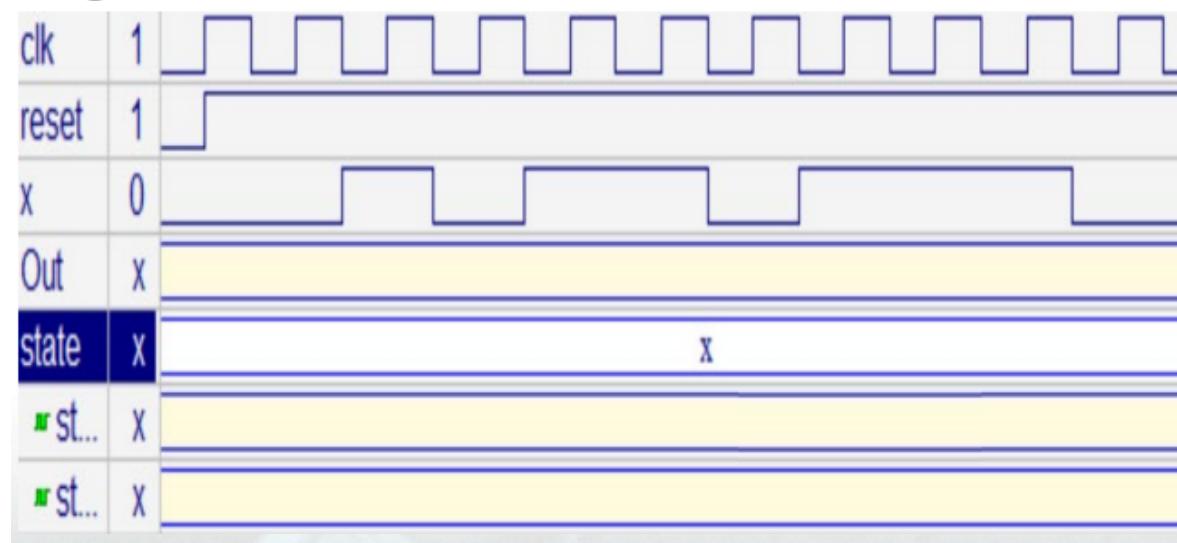




```
reg state = 2'b00;
```

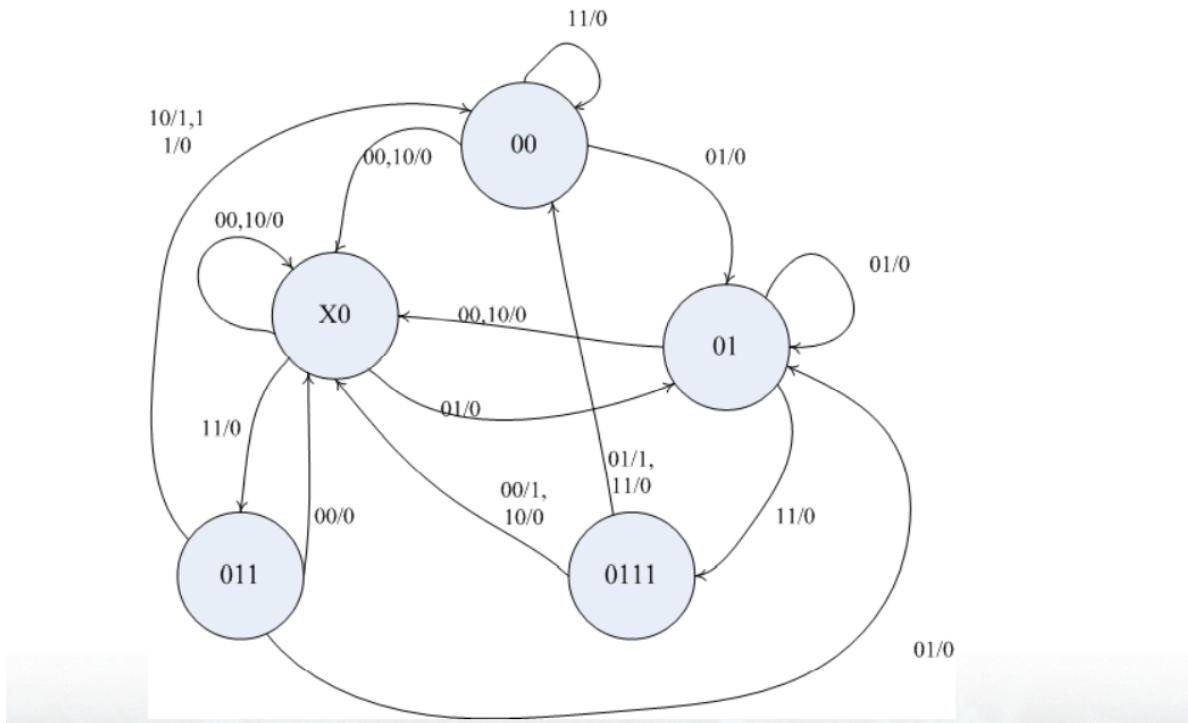
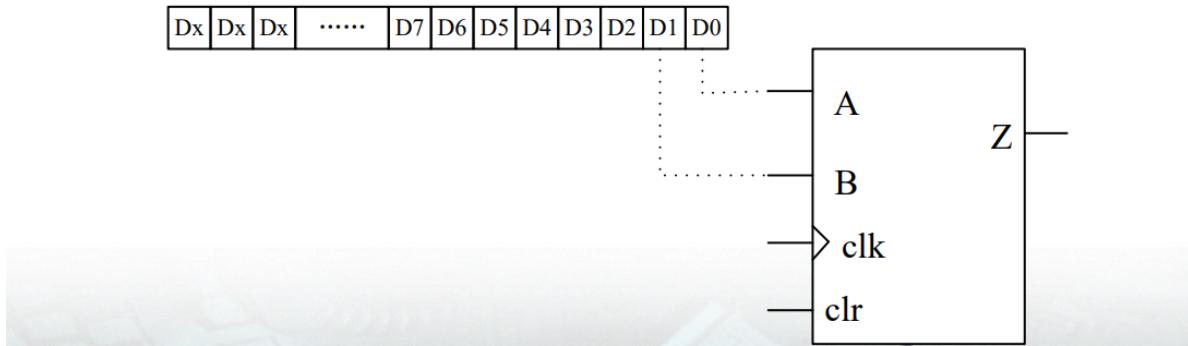


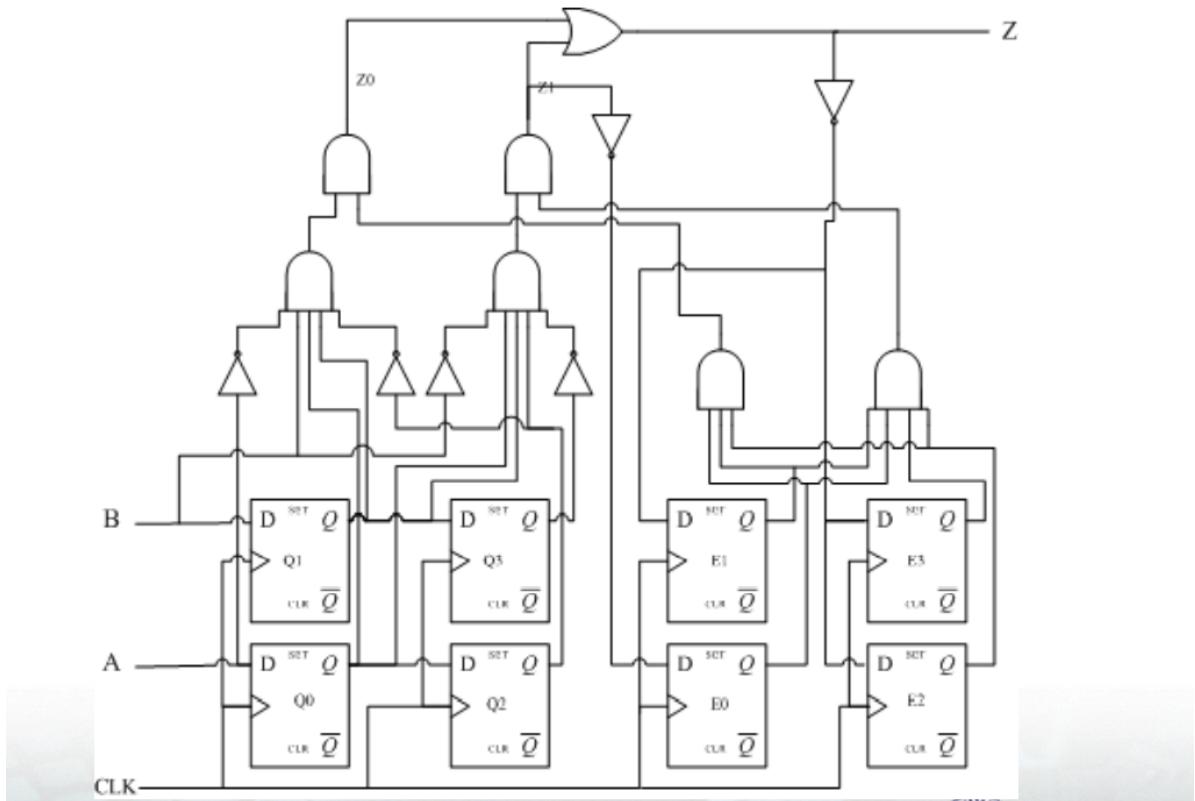
```
reg state ;
```



作业

请设计一个序列检测功能的时序电路，如图所示。其输入信号有clk、clr、A、B，输出信号Z。其中clk为时钟信号，clr是低电平有效的异步复位信号，A、B是输入数据信号。有一个二进制串行数据D<sub>0</sub>D<sub>1</sub>D<sub>2</sub>D<sub>3</sub>……D<sub>x</sub>，以两位为一组顺序送入电路，D<sub>0</sub>送入A，D<sub>1</sub>送到B，以此类推。电路检测此串行数据中是否存在“01110”序列，每发现一个序列则在Z输出一个时钟周期宽度的高电平脉冲。注意相邻“01110”序列数据位不重叠。



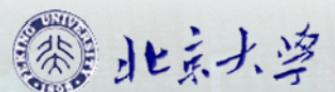
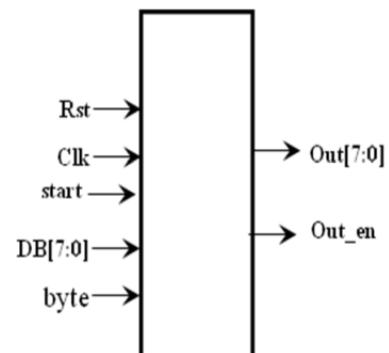


某电路如图所示，其中：

- Rst是低电平有效的系统复位信号，
- start是数据指示信号，
- Clk是时钟信号。
- DB[7:0]是数据输入信号，
- byte是高电平有效的指示信号。

系统复位结束后，当start信号为1时表示从DB送来有效数据。当byte为1时表示DB的8位数据都是有效数据位，为0时表示只有DB[3:0]是有效数据位。这些有效数据位以8位方式从Out[7:0]送出，当Out数据有效时，输出信号Out\_en为1。

请用verilog语言描述此电路。



## Verilog中的高级结构

### task定义

```

module top;
    .....
task task_name;
    <parameter_declaration>; //参数声明
    <input_declaration>; //输入声明
    <output_declaration>; //输出声明
    <inout_declaration>; //inout声明
    <reg_declaration>; //内部变量声明
    <time_declaration>;
    <integer_declaration>;
    <event_declaration>;
begin
    statement;
    statement;
end
endtask

initial task_name(argu_list); // 任务调用
endmodule

```

类似过程块描述

中间变量只能是寄存器类型

#### Task调用:

只能在过程块内调用  
变量列表按定义顺序  
输入可以是任何类型及表达式  
输出只能是寄存器类型

## 任务的主要特点

- 任务可以有input,output 和 inout参数。
- 传送到任务的参量和与任务I/O说明顺序相同。尽管传送到任务的参量名称与任务内部I/O说明的名字可以相同，但在实际中这通常不好。参量名的唯一性可以使任务具有好的模块性。
- 可以在任务内使用时序控制。
- 在Verilog中任务定义一个新范围 (scope)
- 要禁止任务，使用关键字disable。

**从代码中多处调用任务时要小心。**因为任务的局部变量只有一个拷贝，并行调用任务可能导致错误的结果。在任务中使用时序控制时这种情况时常发生。

**在任务中引用module的变量时要小心。**如果想使任务能多个过程块中调用，则所有在任务或函数内部用到的变量都必须列在端口列表中。

## 任务的简单例子

```

module mult (
    input                                clk,
    input                                en_mult,
    input [3: 0]   a,
    input [3: 0]   b,
    output reg [7: 0] out
);
always @( posedge clk)
    multme (a, b, out); // 任务调用

task multme; // 任务定义
    input [3: 0] xme,
    input [3: 0] tome;
    output [7: 0] result;

    wait (en_mult)
        result = xme * tome;
    endtask
endmodule

```

注意a,b, out  
的类型

下面的任务中有输入、输出、时序控制和一个内部变量，并且引用了一个**module** 变量。

任务调用时的参数按任务定义的顺序列出。

Xme和en\_mult  
什么不同？  
为什么？

- 调用那一时刻传递值，传递新值需要等待下一次调用
- 等task完成一个任务才相应上升沿
- 注意参数覆盖

## 任务的测试

```

`timescale 1ns/1ns
module mult_tb();
    reg clk ,
    en_mult ;
    reg [3: 0] a ,
    b ;
    wire [7: 0] out;
    mult umult(
        .clk (clk),
        .en_mult (en_mult),
        .a (a),
        .b (b),
        .out (out)
    );
    always #5 clk = !clk;

initial begin
    clk = 0;
    a = 2;
    b = 10;
    en_mult = 0;
    @(negedge clk)
        a = 4;
        b = 8;
    @(negedge clk)
        a = 5;
        b = 6;
    en_mult = 1;
endinitial

```

```

@(negedge clk)
a = 7;
b = 8;
#10 $finish;
end
endmodule

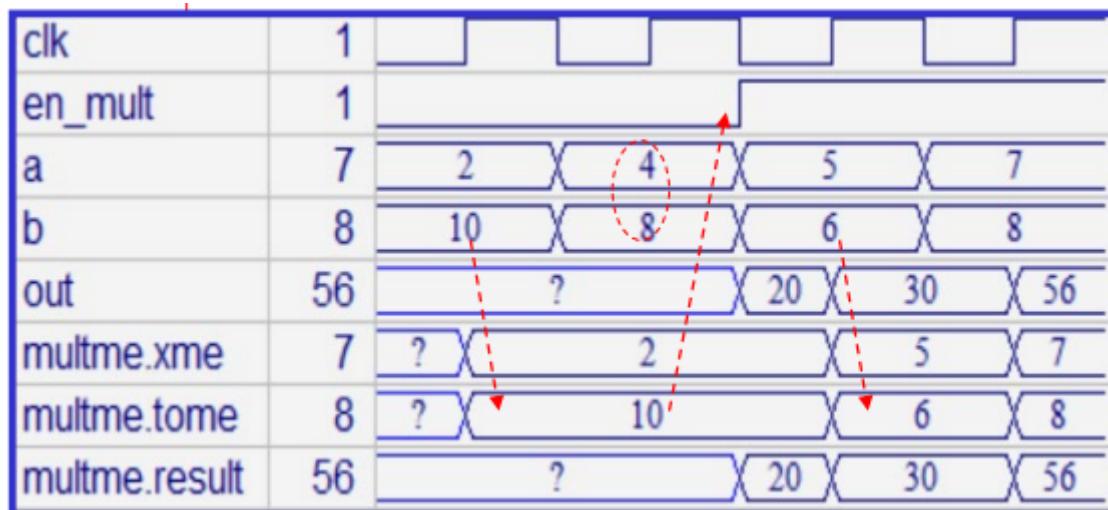
```

## 参数传递

```

module mult (
    input wire clk ,
    en_mult ,
    input wire [3: 0] a ,
    b ,
    output reg [7: 0] out
);
    always @C posedge clk)
        multme (a, b, out);
    task multme; // 任务定义
        input [3: 0] xme,
        tome;
        output [7: 0] result;
        wait (en_mult)
        result = xme * tome;
    endtask
endmodule

```



## 静态任务：参数覆盖

```

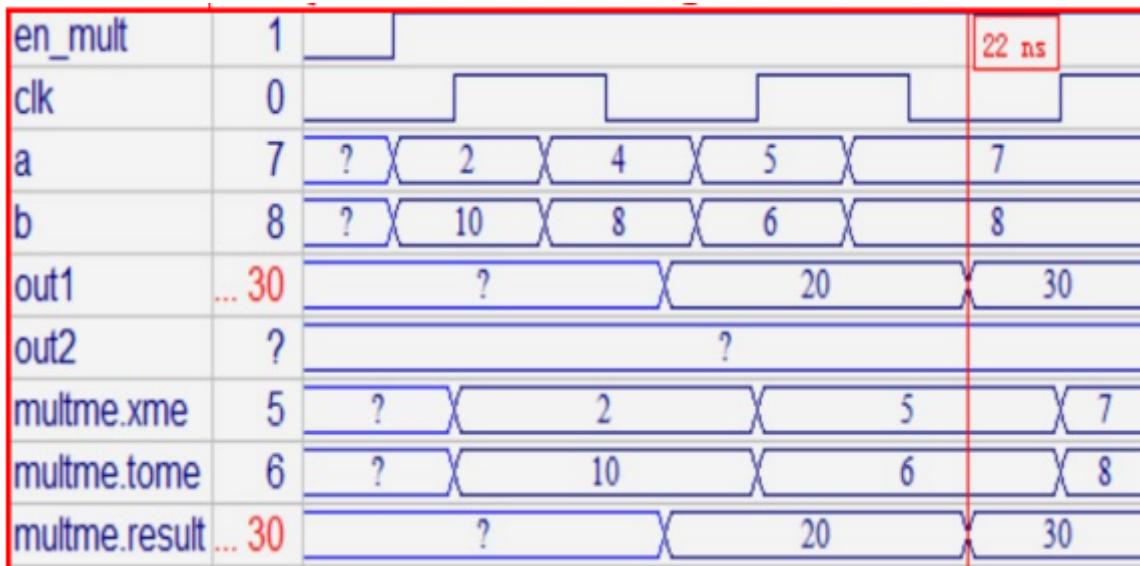
`timescale 1ns/1ns
module mult (
    input wire clk ,
    en_mult ,
    input wire [3: 0] a , b
    output reg [7: 0] out1 ,
    out2
);
    always @C posedge clk)
        multme (a, b, out1);
    // always @C negedge clk)

```

```

// multme (a, b, out2);
task multme;
    input [3: 0] xme,
              tome;
    output [7: 0] result;
    begin
        wait(en_mult);
        #7 result = xme * tome;
    end
endtask
endmodule

```



```

`timescale 1ns/1ns
module mult (
    input wire clk ,
    en_mult ,
    input wire [3: 0] a , b
    output reg [7: 0] out1 ,
    out2
);
    always @(`posedge clk)
        multme (a, b, out1);
    always @(`negedge clk)
        multme (a, b, out2);
    task multme;
        input [3: 0] xme,
                  tome;
        output [7: 0] result;
        begin
            wait(en_mult);
            #7 result = xme * tome;
        end
    endtask
endmodule

```

en_mult	1	_____					22 ns	
clk	0	_____	_____	_____	_____	_____		
a	7	? X 2 X 4 X 5 X	7					
b	8	? X 10 X 8 X 6 X	8					
out1	... 30	?	_____	20	_____	30		
out2	?	_____	?	_____	_____	_____		
multme.xme	5	? X 2 X 5 X	7					
multme.tome	6	? X 10 X 6 X	8					
multme.result	... 30	?	_____	20	_____	30		

en_mult	1	_____					22 ns	
clk	0	_____	_____	_____	_____	_____		
a	7	? X 2 X 4 X 5 X	7					
b	8	? X 10 X 8 X 6 X	8					
out1	... 56	?	_____	32	_____	56		
out2	30	?	_____	20	_____	30		
multme.xme	7	? X 2 X 4 X 5 X	7					
multme.tome	8	? X 10 X 6 X	8					
multme.result	... 56	?	_____	20	32	30	56	

## 动态任务：参数传递

```

`timescale 1ns/1ns
module mult (
    input wire clk ,
    en_mult ,
    input wire [3: 0] a , b
    output reg [7: 0] out1 ,
    out2
);
    always @(`posedge clk)
        multme (a, b, out1);
    always @(`negedge clk)
        multme (a, b, out2);
    task automatic multme;
        input [3: 0] xme,
                    tome;
        output [7: 0] result;
        begin
            wait(en_mult);
            #7 result = xme * tome;
        end
    endtask
endmodule

```

## 函数定义

type: integer  
real time

```
module function_example;
    ...
    function <range or type> <name_of_function>;
        <parameter_declaration>;
        <input_declaration>;
        <reg_declaration>;
        <time_declaration>;
        <integer_declaration>;
        <real_declaration>;
        <event_declaration>;
        begin
            statement;
            statement;
        end
    endfunction
    assign word = control ? name_of_function(argu_list) : 0;
    initial bye = name_of_function(argu_list);
endmodule
```

## 函数的特点

- 函数定义中不能包含任何时序控制语句。
- 函数至少有一个输入，不能包含任何输出或双向端口。
- 函数只返回一个数据，其缺省为register类型。
- 传送到函数的参数顺序和函数输入参数的说明顺序相同。
- 函数在模块 (module) 内部定义且只能在模块内部调用。
- 函数不能调用任务，但任务可以调用函数。
- 函数在Verilog中定义了一个新的范围 (scope)。
- 虽然函数只返回单个值，但返回的值可以使用{信号}赋值。这在需要有多个输出时非常有效。
  - {o1, o2, o3, o4} = f\_or\_and (a, b, c, d, e);

## 函数举例

```
module operand (
    input [7: 0] a, b, c, d, e,
    output [7 : 0] out
);
    reg [7: 0] out;
    always @(
        a or b or c or d or e)
        out = f_or_and (a, b, c, d, e); // 函数调用
    function [7 : 0] f_or_and;
        input [7 : 0] a, b, c, d, e;
        if (e == 1)
```

```

        f_or_and = (a | b) & (c | d);
    else
        f_or_and = 0;
endfunction
endmodule

```

- 函数中不能有时序控制，但调用它的过程语句可以有时序控制。
- 函数名f\_or\_and在函数中作为register使用

## 返回向量的函数

- 要返回一个向量值（多于一位），在函数定义时在函数名前说明范围。函数中有多条语句时用begin和end。
- 不管在函数内对函数名进行多少次赋值，值只返回一次。下例中，函数还在内部声明了一个整数。

```

module foo(
    input [7: 0] loo,
    output [3: 0] goo);
// 可以持续赋值中调用函数
    wire [3: 0] goo = zero_count ( loo );
    function [3: 0] zero_count;
        input [7: 0] in_bus;
        integer I;
        begin
            zero_count = 0;
            for (I = 0; I < 8; I = I + 1)
                if (!in_bus[ I ])
                    zero_count = zero_count + 1;
        end
    endfunction
endmodule

```

## 函数返回类型

- 函数返回值可以声明为其它register类型：integer, real, 或time。
- 在任何表达式中都可调用函数

```

module checksub (
    output reg neg,
    input wire [7 : 0] a, b
);
    function integer subtr;
        input [7: 0] in_a, in_b;
        subtr = in_a - in_b; // 结果可能为负
    endfunction
    always @ (a, b)
        if (subtr( a, b) < 0)
            neg = 1;
        else
            neg = 0;
endmodule

```

- 递归函数必须声明为动态的，因为第二次调用会覆盖第一次的值

- 逻辑综合工具不支持递归调用

## 递归函数-阶乘函数

```

`timescale 1ns/1ns 递归函数-阶乘函数
module fact();
reg [7 : 0] in_a;
integer out1,
out2;
initial begin
    in_a = 7;
    out1 = factorial_R( in_a );
    out2 = factorial ( in_a );
#10 in_a = 14;
    out1 = factorial_R( in_a );
    out2 = factorial ( in_a );
#10 $finish;
end
initial $monitor("%d %d %d", in_a,
out1, out2);

function automatic integer factorial_R;
input [7 : 0] oper;
if( oper >= 2)
    factorial_R = oper * factorial_R( oper - 1 );
else
    factorial_R = 1;
endfunction

function integer factorial;
input [7 : 0] oper;
integer i;
begin
factorial = 1;
for(i = 2; i <= oper; i = i + 1)
    factorial = factorial * i;
end
endfunction
endmodule

```

## 递归函数(recursive Function)

```

`timescale 1ns /1ns
module factorial(
    input wire [7 : 0] in_a,
    output integer    out
);
always @(*)
    out = factorial_R(in_a);

function automatic integer factorial_R;
    input [7 : 0] oper;
    if( oper >= 2)
        factorial_R = oper * factorial_R( oper - 1 );
    else
        factorial_R = 1;
endfunction
endmodule

```

**Quartus:**  
**Error (10210): Verilog HDL unsupported feature error at f.v(9): recursive Function Call in Function Declaration is not supported**

**Synopsys:**  
**Error: factorial.v:12: Function call stack exceeded maximum depth. (ELAB-901).**

## 参数化函数

- 函数中可以对返回值的个别位进行赋值。
- 函数值的位数、函数端口甚至函数功能都可以参数化。

```

. . .
parameter MAX_BITS = 8;
reg [MAX_BITS: 1] D;
function [MAX_BITS: 1] reverse_bits;
    input [MAX_BITS-1: 0] data;
    integer K;
    for (K = 0; K < MAX_BITS; K = K + 1)
        reverse_bits [MAX_BITS - K] = data [K];
endfunction
always @ (posedge clk)
    D = reverse_bits (D) ;
. . .

```

## 多module调用函数

```

module t1( );
...
parameter MAX_BITS = 8;
reg [MAX_BITS: 1] D;

function [MAX_BITS: 1] reverse_bits;
    input [MAX_BITS-1: 0] data;
    integer K;
    for (K = 0; K < MAX_BITS; K = K + 1)
        reverse_bits [MAX_BITS - K] = data [K];
endfunction

always @ (posedge clk)
    D = reverse_bits (D);
...
endmodule

```

```

module t2( );
...
reg [7 : 0] C;

always @ (posedge clk)
    C = reverse_bits (C);
...
endmodule

```

- 一个模块定义的function不能直接调用
- 一般把函数单独一个文件存储，调用使用'include

**t1.v**

```

module t1( );
...
parameter MAX_BITS = 8;
reg [MAX_BITS: 1] D;

`include "reverse_bits.v"

always @ (posedge clk)
    D = reverse_bits (D);
...
endmodule

```

**t2.v**

```

module t2( );
...
reg [7 : 0] C;

`include "reverse_bits.v"

always @ (posedge clk)
    C = reverse_bits (C);
...
endmodule

```

**reverse\_bits.v**

```

function [7: 0] reverse_bits;
    input [MAX_BITS-1: 0] data;
    integer K;
    for (K = 0; K < MAX_BITS; K = K + 1)
        reverse_bits [MAX_BITS - K] = data [K];
endfunction

```

## Verilog的任务及函数

- 函数(function)

- ✓ 通常用于计算，或描述组合逻辑
- ✓ 不能包含时序控制；函数仿真时间为0
- ✓ 只含有input变量并由函数名返回一个结果
- ✓ 可以调用其他函数，但不能调用任务

- 任务 (task)

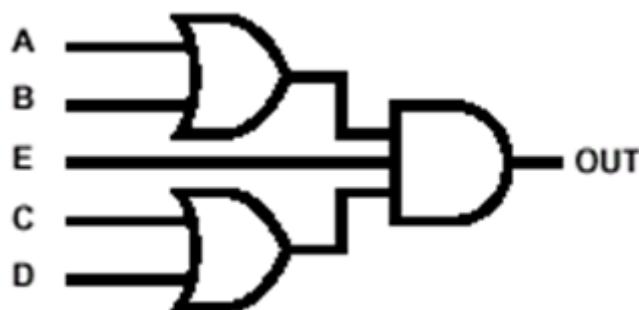
- ✓ 通常用于测试程序，或对硬件进行行为描述
- ✓ 可以包含时序控制 (#延迟, @, wait)
- ✓ 可以有 input, output, 和 inout参数
- ✓ 可以调用其他任务或函数

- 任务和函数必须在module内部定义和调用
- 在任务和函数中不能声明wire
- 所有输入/输出都是局部寄存器
- 任务/函数执行完成后才返回结果。
- 例如，若任务/函数中有forever语句，则永远不会返回结果

## 函数

- 函数没有时序控制，因此综合结果为组合逻辑。函数可以在过程块内或持续赋值语句中调用。
- 下例中的or/and块由持续赋值语句调用函数实现

```
module orand (
    input a, b, c, d, e,
    output wire out
);
    assign out = forand (a, b, c, d, e);
    function forand;
        input a, b, c, d, e;
        if (e == 1)
            forand = (a| b) & (c| d);
        else
            forand = 0;
    endfunction
endmodule
```

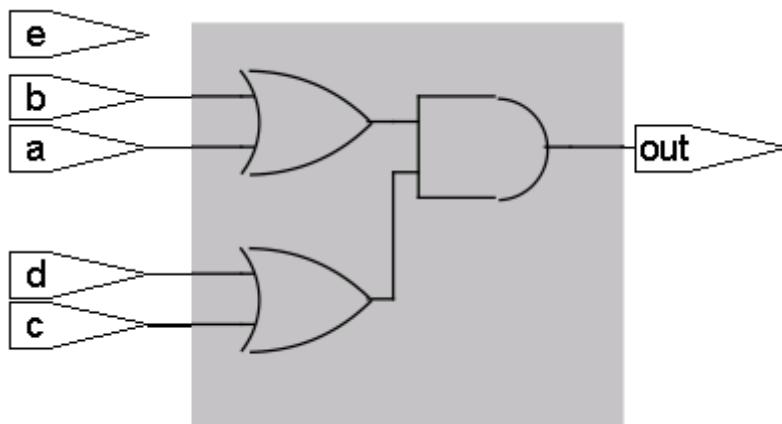


函数—条件不完全？

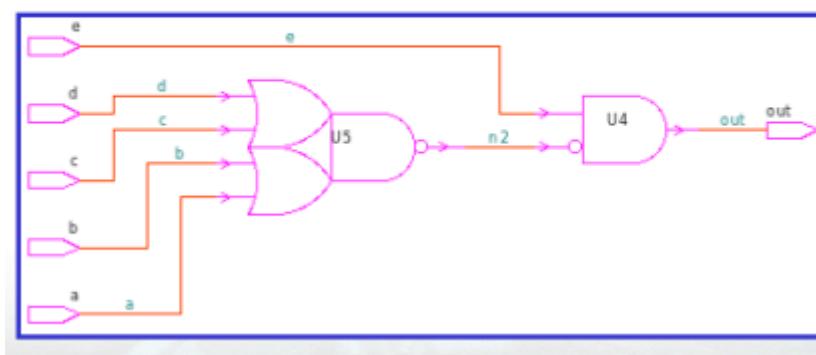
```

module operand (
    input a, b, c, d, e,
    output wire out
);
    assign out = forand (a, b, c, d, e);
    function forand;
        input a, b, c, d, e;
        if (e == 1)
            forand = (a| b) & (c| d);
        // else
        //     forand = 0;
    endfunction
endmodule

```



条件不完全会产生什么？



## 任务

```

module operandtask (out, a, b, c, d, e);
    input a, b, c, d, e;
    output out; reg out;
    always @(*)
        operand (out, a, b, c, d, e);
    task operand;
        input a, b, c, d, e;
        output out;
        if (e == 1)
            out = (a| b) & (c| d);
        else
            out = 0;
    endtask

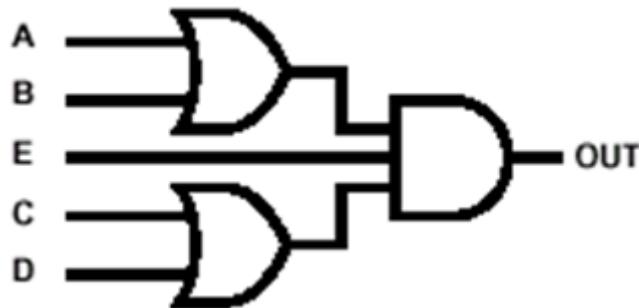
```

```
endmodule
```

任务一般只在测试程序中使用，因为：

- 没有时序控制的任务如同函数
- 带有时序控制的任务不可综合

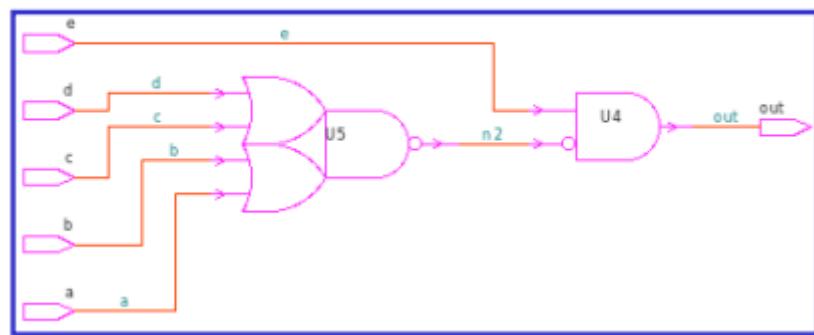
下面是用任务描述的or/and块：



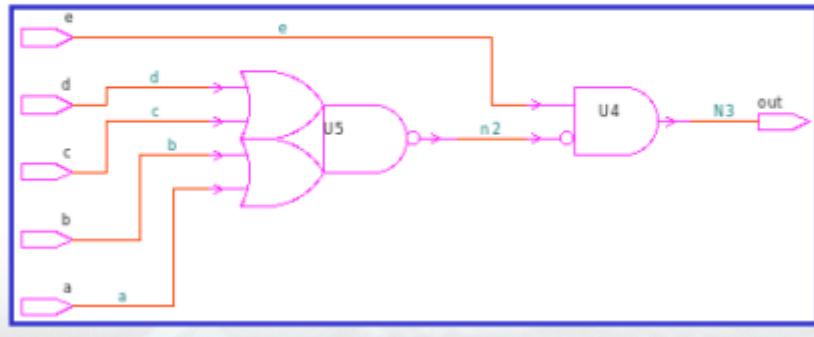
条件不完全会产生什么？

```
module orandtask (
  input a, b, c, d, e,
  output reg out
);
  always @(*)
    orand (a, b, c, d, e, out);
  task orand;
    input a, b, c, d, e;
    output out;
    if (e == 1)
      out = (a| b) & (c| d);
    // else
    //   out = 0;
  endtask
endmodule
```

## 条件完全



## 条件不完全会产生什么？



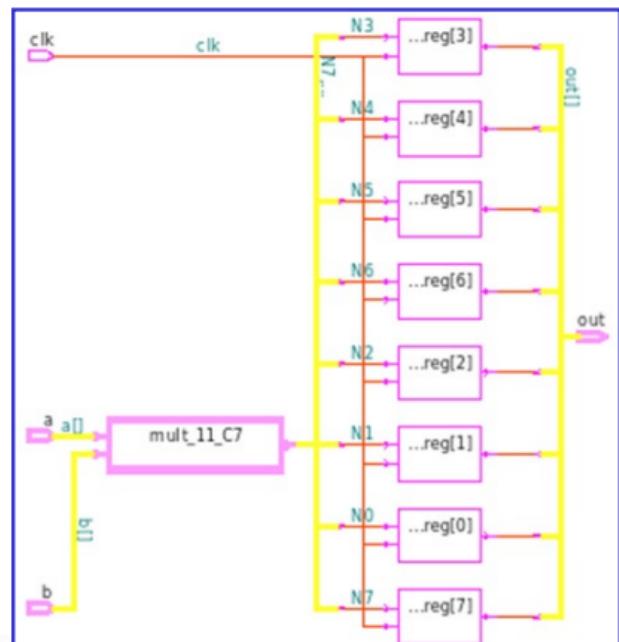
## 任务：电路1

```
module mult (
    input wire      clk,
    input wire [3: 0] a,
    b,
    output reg [7: 0] out
);

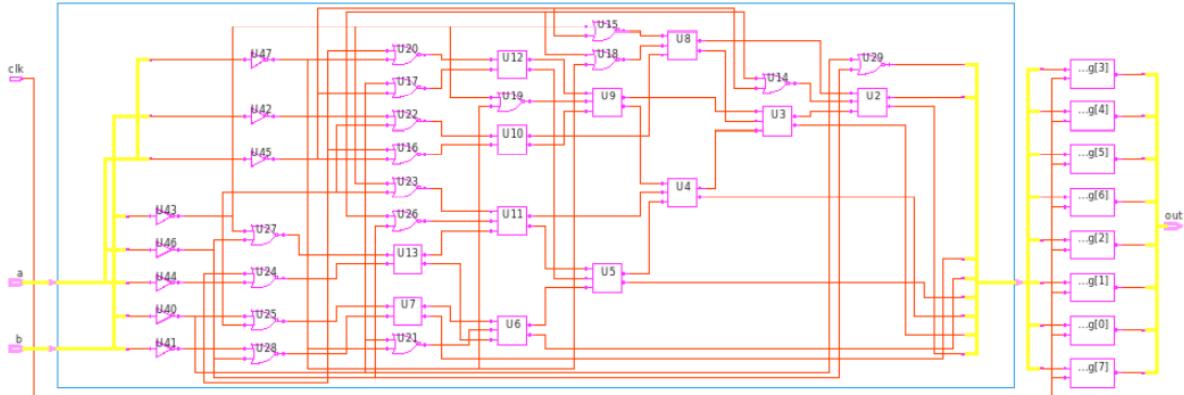
    always @( posedge clk)
        multme (a, b, out);

    task multme; // 任务定义
        input [3: 0] xme, tome;
        output [7: 0] result;
        result = xme * tome;
    endtask

endmodule
```



## 任务：电路2



## 命名块(named block)

- 在关键词begin或fork后加上：`<块名称>` 对块进行命名

```
module named_blk;
    .
    .
    begin : seq_blk
    .
    .
    end
    .
    .
    fork : par_blk
    .
    .
    join
    .
    .
endmodule
```

- 在命名块中可以声明局部变量
- 可以使用关键词**disable**禁止一个命名块
- 命名块定义了一个新的范围
- 命名块会降低仿真速度

## 禁止命名块和任务

```
module do_arith (out, a, b, c, d, e, clk, en_mult);
    input clk, en_mult;
    input [7: 0] a, b, c, d, e;
    output [15: 0] out;
    reg [15: 0] out;
    always @(* posedge clk)
        begin : arith_block // *** 命名块 ***
            reg [3: 0] tmp1, tmp2; // *** 局部变量 ***
            {tmp1, tmp2} = f_or_and (a, b, c, d, e); // 函数调用
            if (en_mult) multme (tmp1, tmp2, out); // 任务调用
        end
    always @(* negedge en_mult) begin // 中止运算
```

```

        disable multme ; // *** 禁止任务 ***
        disable arith_block; // *** 禁止命名块 ***
    end
// 下面是[定义任务和函数
.....
endmodule

```

- disable语句终结一个命名块或任务的所有活动。也就是说，在一个命名块或任务中的所有语句执行完之前就返回。

语法：

disable <块名称>

或

disable <任务名称>

- 当命名块或任务被禁止时，所有因他们调度的事件将从事件队列中清除
- **disable是典型的不可综合语句。**
- 在前面的例子中，只禁止命名块也可以达到同样的目的：所有由命名块、任务及其中的函数调度的事件都被取消。

## 对验证的支持

### 验证系统中的任务(task)及函数(function)

- Verilog读取当前仿真时间的系统函数

\$time : 返回一个64位整数时间值。

\$stime : 返回一个32位整数时间值。

\$realtime : 返回一个实数时间值。

返回值使用调用模块中`timescale定义的时间单位

- Verilog支持文本输出的系统任务：

\$display

\$strobe

\$write

\$monitor

## 输出格式化时间信息

- 若使用多个`timescale，以最小的时间精度显示时间值。
- 可用系统任务\$timeformat结合格式符%t全局控制时间显示方式。
- \$timeformat系统任务的语法为：
  - \$timeformat(<unit>,<precision>,<suffix>,<min\_width>);

```

`timescale 10ns / 100ps
module top;
    reg in1;
    not m1( o1, in1);
    initial begin
        $timeformat(-9, 2, "ns", 10);
        in1 = 0;
        #10 in1 = 1;
        #10 $finish;
    end
    initial
        $monitor($time, "%t %b %b", $realtime,
in1, o1);
endmodule

```

在这个例子中，显示的时间为：180.00 ns

**unit**: 0(s)到-15(fs)之间的整数，表示时间度量

**precision**: 要显示的十进制小数位数。

**suffix**: 在时间值后显示的字符串

**min\_width**: 显示前三项的最小宽度

对#延迟，Verilog将延迟值舍入最近(四舍五入)时间精度值。

例如，上面的例子修改为：

```

`timescale 1ns/ 100ps
not #9.49 m1 (o1, in1);

```

结果为：

time	realtime	stime	in1	o1
0	0.00ns	0	0	x
9	9.50ns	9	0	1
10	10.00ns	10	1	1
19	19.50ns	19	1	0

```

`timescale 1ns/ 100ps
not #9.42 m1 (o1, in1);

```

结果为：

time	realtime	stime	in1	o1
0	0.00ns	0	0	x
9	9.40ns	9	0	1
10	10.00ns	10	1	1
19	19.40ns	19	1	0

## 显示信号值 — \$display

- \$display输出参数列表中信号的当前值。

语法：\$display([" format\_specifiers"], <argument\_list>)

- \$display输出时自动换行。

```
$display ($time, "%b \t %h \t %d \t %o", sig1, sig2, sig3, sig4);
```

```
$display ($time, "%b \t", sig1, "%h \t", sig2, "% d \t", sig3, "%o", sig4);
```

- \$display支持二进制、八进制、十进制和十六进制。缺省基数为十进制。

```
$display (sig1, sig2, sig3, sig4);
```

```
$displayb (sig1, sig2, sig3, sig4);
```

```
$displayo (sig1, sig2, sig3, sig4);
```

```
$displayh (sig1, sig2, sig3, sig4);
```

## 格式符

%h	%o	%d	%b	%c	%s	%v	%m	%t
hex	octal	decimal	binary	ASCII	string	strength	module	time

---

\t	\n	\\"	\\"	< 1-3 digit octal number>	%0d
tab	换行	反斜杠	双引号	上述的ASCII表示	无前导0的十进制数

## 显示信号值—\$write和\$strobe

- \$write与\$display相同，不同的是**不会自动换行**。

```
$write($time, "%b \t %h \t %d \t %o \t", sig1, sig2, sig3, sig4);
```

- \$strobe与\$display相同，不同的是在仿真时间前进之前的信号值。而**\$display和\$write立即显示信号值**。也就是说**\$strobe显示稳定状态信号值**，而\$display和\$write可以显示信号的中间状态值。

```
$strobe($time, "%b \t %h \t %d \t %o \t", sig1, sig2, sig3, sig4);
```

- \$write和\$strobe都支持多种数基，缺省为十进制。
  - \$writeb \$strobeb
  - \$writeo \$strobo
  - \$writeh \$strobh

```
module textio;
reg flag;
reg [31: 0] data;
initial
begin
    $writeb("%d", $time, "%h \t",
           data, , flag, "\n");
    #15 flag = 1; data = 16;
    $displayh($time, ,data, , flag);
end
initial begin
    #10 data = 20;
    $strobe($time, "st is", data);
    $display($time, "dis is", data);
    data = 30;
end
endmodule
```

下面是模块textio仿真的输出：

- \$writeb输出：

```
0 xxxxxxxx x
```

注意data是32位数据，由8位十六进制数表示。时间以没有前导零的十进制形式输出。

缺省情况下，值以十进制显示，忽略前导零，与%0d格式符相同。可以在一个格式化符前插入一个0使Verilog忽略开头的零。

- \$displayh：

```
000000000000000f 00000010 1
```

注意当前时间，一个64位量，需要16个十六进制的数。

```
10 dis is 20
10 st is 30
```

## 监视信号值—\$monitor

- \$monitor持续监视参数列表中的变量。
- 在一个时间片中，参数表中任何信号发生变化，\$monitor将在仿真时间前进前显示参数表的信号值。
- 后面的\$monitor将覆盖前面的\$monitor。
- 可以用系统任务\$monitoron和\$monitoroff控制持续监视。

- \$monitor支持多种基数。缺省为十进制。

```
\$monitor (\$time, "%b \t %h \t %d \t %o", sig1, sig2, sig3, sig4);
```

- \$monitor是唯一的不断输出信号值的系统任务。其它系统任务在返回值之后就结束。
- \$monitor和\$strobe一样，显示参数列表中信号的稳定状态值，也就是在仿真时间前进之前显示信号。**在一个时间步中，参数列表中信号值的任何变化将触发\$monitor。但\$time,\$stime,\$realtime不能触发。**
- 任何后续的\$monitor覆盖前面调用的\$monitor。只有新的\$monitor的参数列表中的信号被监视，而前面的\$monitor的参数则不被监视。
- 可以用\$monitoron和\$monitoroff系统任务控制持续监视，使用户可以在仿真时只监视特定时间段的信号。
- \$monitor参数列表的形式与\$display相同。
- \$monitor支持多种基数。缺省为十进制。
  - \$monitorb
  - \$monitoro
  - \$monitorh

## 文件输出

```
.
.
integer MCD1;
MCD1 = $fopen("<name_of_file>");
$fdisplay( MCD1, P1, P2, ..., Pn);
$fwrite( MCD1, P1, P2, ..., Pn);
$fstrobe( MCD1, P1, P2, ..., Pn);
$fmonitor( MCD1, P1, P2, ..., Pn);
fclose( MCD1);
.
.
```

- \$fopen打开一个文件并返回一个多通道描述符（MCD）。
  - MCD是与文件唯一对应的32位无符号整数。
  - 如果文件不能打开并进行写操作，MCD等于0。
  - 如果文件成功打开，MCD中的一位被置位。
- 以\$f开始的显示系统任务将输出写入与MCD相对应的文件中。
- \$fopen打开参数中指定的文件并返回一个32位无符号整数MCD，MCD是与文件一一对应的多通道描述符。如果文件不能打开并进行写操作，它返回0。
- \$fclose关闭MCD指定的通道。
- 输出信息到log文件和标准输出的四个格式化显示任务(\$display, \$write, \$monitor, \$strobe)都有相对应的任务用于向指定文件输出。
- 这些对应的任务 (\$fdisplay,\$fwrite,\$fmonitor,\$fstrobe) 的参数形式与对应的任务相同，只有一个例外：第一个参数必须是一个指定向何哪个文件输出的MCD。MCD可以是一个表达式，但其值必须是一个32位的无符号整数。这个值决定了该任务向哪个打开的文件写入。
- MCD可以看作由32个标志构成的组，每个标志代表一个单一的输出通道。

```

...
integer messages, broadcast, cpu_chann, alu_chann;
initial
begin
    cpu_chann = $fopen("cpu.dat"); if(!cpu_chann) $finish;
    alu_chann = $fopen("alu.dat"); if(!alu_chann) $finish;
    // channel to both cpu.dat and alu.dat
    messages = cpu_chann | alu_chann;
    // channel to both files, standard out, and verilog.log
    broadcast = 1 | messages; ← 通道0（编号为1）为
end                                         标准输出及verilog.log
always @(posedge clock) // print the following to alu.dat
    $fdisplay(alu_chann, "acc=%h f=%h a=%h b=%h", acc, f, a, b);
/* at every reset print a message to alu.dat, cpu.dat, standard output
and the verilog.log file */
always @(negedge reset)
    $fdisplay(broadcast, "system reset at time %d", $time);
...

```

## 文件输入

- Verilog中有两个系统任务可以将数据文件读入寄存器组。一个读取二进制数据，另一个读取十六进制数据：

- \$readmemb

```

$readmemb ("file_name", <memory_name>);
$readmemb ("file_name", <memory_name>, <start_addr>);
$readmemb ("file_name", <memory_name>, <start_addr>, <finish_addr>);

```

- \$readmemh

```

$readmemh ("file_name", <memory_name>);
$readmemh ("file_name", <memory_name>, <start_addr>);
$readmemh ("file_name", <memory_name>, <start_addr>, <finish_addr>);

```

系统任务\$readmemb和\$readmemh从一个文本文件读取数据并写入存储器。

- 如果数据为二进制，使用\$readmemb；如果数据为十六进制，使用\$readmemh。
- filename指定要读入的文件。
- mem\_name指定存储器信号名称。
- start和finish给出存储器加载的地址。Start为开始地址，finish为结束地址。如果不指定开始和结束地址，\$readmem按从低端开始读入数据，与说明顺序无关。

\$readmemb和\$readmemh的文件格式：

```
$readmemb("mem_file.txt", mema);
```

- 可以指定二进制 (b) 或十六进制 (h) 数
- 用下划线“\_”提高可读性。

- 可以包含单行或多行注释。
- 可以用空格和换行区分各个数据。
- 可以给后面的值设定一个特定的地址，格式为：
  - @ (hex\_address)
  - 十六进制地址的大小写不敏感。
  - 在@和数字之间不允许有空格。

文本文档：m\_file.txt

```
0000_0000
0110_0001 0011_0010
// 地址3~15没有定义
@10 // hex
1111_1100
// 地址17~30没有定义
@1F
1110_0010
```

	0	1	2	3	4	5	6	7
00	00	61	32	xx	xx	xx	xx	xx
08	xx							
10	FC	xx						
18	xx	E2						
20	xx							
28	xx							
30	xx							
38	xx							

```
module memread;
reg [7 : 0] mema [63 : 0];
initial $readmemb("m_file.txt", mema);
endmodule
```

文本文档：m\_file.txt

```
0000_0000
0110_0001 0011_0010
// 地址3~15没有定义
@10 // hex
1111_1100
// 地址17~30没有定义
@1F
1110_0010
```

	0	1	2	3	4	5	6	7
00	xx	00	61	32	xx	xx	xx	xx
08	xx							
10	FC	xx						
18	xx	E2						
20	xx							
28	xx							
30	xx							
38	xx							

```
module memread;
reg [7 : 0] mema [63 : 0];
initial $readmemb("m_file.txt", mema, 1 );
endmodule
```

### 文本文件: m\_file.txt

```
0000_0000
0110_0001 0011_0010
// 地址3~15没有定义
@10 // hex
1111_1100
//地址17~30没有定义
@1F
1110_0010
```

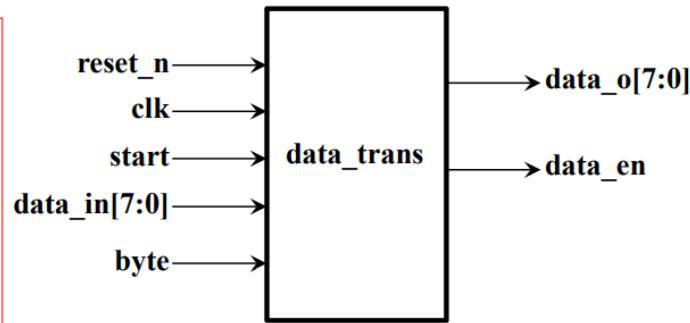
	0	1	2	3	4	5	6	7
00	xx	00	61	32	xx	xx	xx	xx
08	xx							
10	FC	xx						
18	xx							
20	xx							
28	xx							
30	xx							
38	xx							

```
module memread;
reg [7 : 0] mema [63 : 0];
initial $readmemb("m_file.txt", mema, 1, 16);
endmodule
```

## 作业：数据转换器

某电路如图所示，其中：

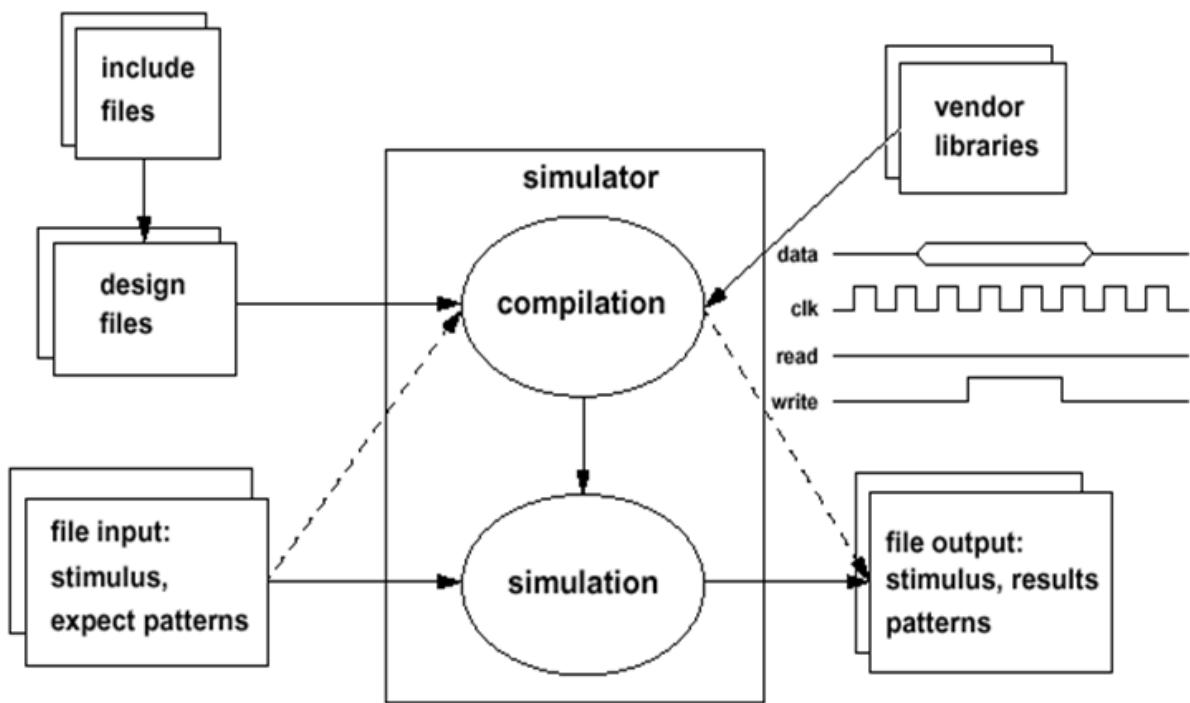
- reset\_n: 低电平异步复位信号
- start: 启动信号
- clk: 时钟信号,
- data\_in[7:0]: 数据输入
- byte: 字节有效的指示



电路复位结束后，当start信号为1时表示开始从data\_in送来数据。当byte为1时表示8位数据都是有效数据位，为0时表示只有[3:0]是有效数据位。这些有效数据位以8位方式从data\_o送出，当在data\_o输出有效数据时，信号data\_en为1，否则为0。

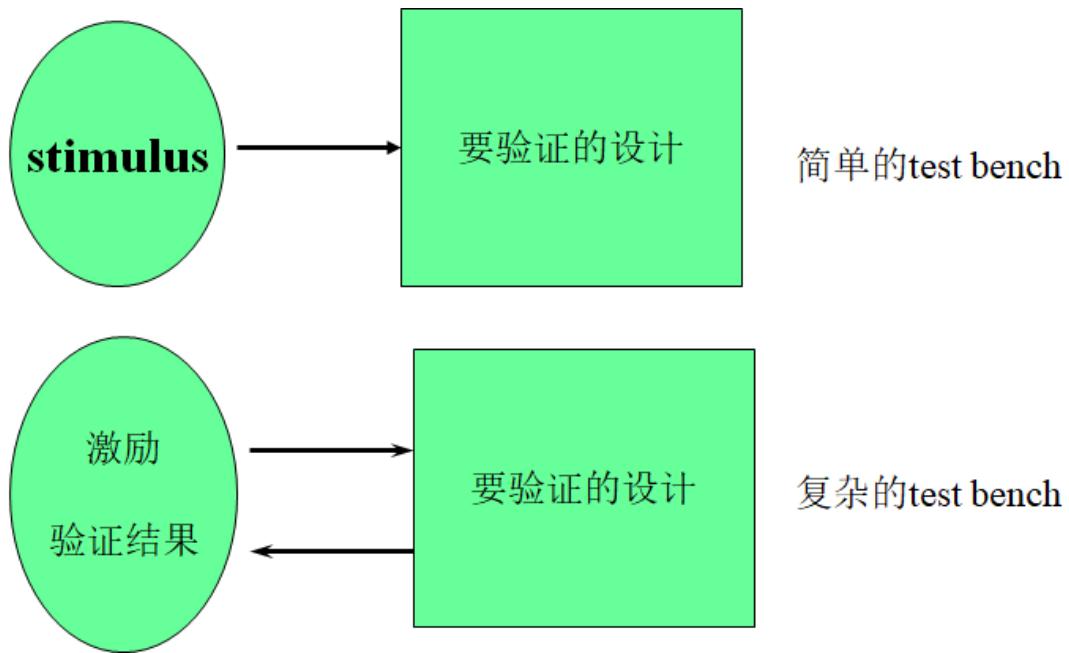
## Verilog Test Bench使用简介

### 设计组织



虚线表示编译时检测输入文件是否存在及可读并允许生成输出文件。

## test bench组织



- 简单的test bench向要验证的设计提供向量，人工验证输出。
- 复杂test bench是自检测的，其结果自动验证。

## 施加激励

产生激励并加到设计有很多 种方法。一些常用的方法有：

- 从一个initial块中施加线激励
- 从一个循环或always块施加激励
- 从一个向量或整数数组施加激励
- 记录一个仿真过程，然后在另一个仿真中回放施加激励

## 线性激励

线性激励有以下特性：

- 只有变量的值改变时才列出
- 易于定义复杂的时序关系
- 对一个复杂的测试，测试程序(test bench)可能非常大

```
module inline_tb;
    reg [7: 0] data_bus, addr;
    wire [7: 0] results;
    DUT u1 (results, data_bus, addr);
    initial
        fork
            data_bus = 8'h00;
            addr = 8'h3f;
            #10 data_bus = 8'h45;
            #15 addr = 8'hf0;
            #40 data_bus = 8'h0f;
            #60 $finish;
        join
endmodule
```

## 循环激励

从循环产生激励有以下特性：

- 在每一次循环，修改同一组激励变量
- 有固定的时序关系
- 代码紧凑

```
`timescale 1ns/1ns
module loop_tb;
    reg clk;
    reg [7:0] stimulus;
    wire [7:0] results;
    integer i;
    DUT u1 (results, stimulus);
    always begin // clock generation
        clk = 1; #5 clk = 0; #5
    end
    initial begin
        for (i = 0; i < 256; i = i + 1)
            @(`negedge` clk) stimulus = i;
        #20 $finish;
    end
endmodule
```

## 数组激励

从数组产生激励有以下特性：

- 在每次反复中，修改同一组激励变量
- 激励数组可以直接从文件中读取

```
module array_tb;
```

```

reg [7: 0] stim_array[ 15: 0]; // 数组
reg [7 : 0] stimulus;
wire result;
integer i;
DUT u1 (results, stimulus);
initial begin
    // 从数组读入数据
    #20 stimulus = stim_array[0];
    #30 stimulus = stim_array[15]; // 线激励
    #20 stimulus = stim_array[1];
    for (i = 14; i > 1; i = i - 1) // 循环
        #50 stimulus = stim_array[i];
    #30 $finish;
end
endmodule

```

## 矢量采样

在仿真过程中可以对激励和响应矢量进行采样，作为其它仿真的激励和期望结果。

```

module capture_tb;
parameter period = 20
reg [7:0] in_vec, out_vec;
integer RESULTS, STIMULUS;
DUT u1 (out_vec, in_vec);
initial begin
    STIMULUS = $fopen("vec.txt");
    if (STIMULUS != 0) forever #(period/2)
        $fstrobeb(STIMULUS, "%b", in_vec);
end
initial begin
RESULTS = $fopen("results.txt");
if (RESULTS != 0) #(period/2) forever
    #(period/2)
        $fstrobeb(RESULTS, "%b", out_vec);
end
endmodule

```

## 矢量回放

- 保存在文件中的矢量反过来可以作为激励

```

module read_file_tb;
    parameter num_vecs = 256;
    reg [7:0] data_bus;
    reg [7:0] stim [num_vecs-1:0];
    integer i;
    DUT u1 (results, data_bus)
    initial
        begin // Vectors are loaded
            $readmemb ("vec.txt", stim);
            for (i = 0; i < num_vecs ; i = i + 1)
                #50 data_bus = stim[i];
        end
endmodule

```

// 激励文件vec.txt

**00111000**

**00111001**

**00111010**

**00111100**

**00110000**

**00101000**

**00011000**

**01111000**

**10111000**

.

.

- 使用矢量文件输入/输出的优点:
  - 激励修改简单
  - 设计反复验证时直接使用工具比较矢量文件。

## 错误及警告报告

- 使用文本或文件输出类的系统任务报告错误及警告

```

always @(
  posedge par_err)
  $display (" error-bus parity errors detected");
always @(
  posedge cor_err)
  $display("warning-correctable error detected");

```

一个更为复杂的test bench可以:

- 不但能报告错误，而能进行一些动作，如取消一个激励块并跳转到下一个激励。
- 在内部保持错误跟踪，并在每次测试结束时产生一个错误报告。

## 建立时钟

例1：虽然有时候在设计中给出时钟，但通常时钟是测试基准中建立。

下面介绍如何产生不同的时钟波形。同时给出用门级和行为级描述方法

下面是一个简单对称时钟的例子：

```
reg ck;  
  
always begin  
    #( period/2) ck = 0;  
    #( period/2) ck = 1;  
end
```

```
reg go; wire ck;  
  
nand #( period/2) u1 (ck, ck, go);  
  
initial begin  
    go = 0;  
    #( period/2) go = 1;  
end
```

产生的波形（假定period为20）



注意：在一些仿真器中，时钟与设计使用相同的抽象级描述时，仿真性能会好一些。

例2：有启动延时的对称时钟的例子：

```
reg ck;  
  
initial begin  
    #( period/2)  
    ck = 0;  
    forever  
        #( period/2) ck = !ck;  
end
```

```
reg go; wire ck;  
  
nand #( period/2) u1 (ck, ck, go);  
  
initial  
begin  
    go = 0;  
    #(period) go = 1;  
end
```

产生的波形（假定period为20）



注意：在行为描述中，在时间0将CK初始化为0；而在结构描述中，直到period/2才影响CK值。当go信号在时间0初始化时，CK值到period/2才变化。可以使用特殊命令force和release立即影响CK值。

### 例3：有不规则启动延时的不对称时钟的例子：

```
reg ck;  
initial begin  
    #(period + 1) ck = 1;  
    #(period/2 - 1)  
    forever begin  
        #(period/4) ck = 0;  
        #(3*period/4) ck = 1;  
    end  
end
```

```
reg go; wire ck;  
nand #(3*period/4, period/4)  
    u1(ck, ck, go);  
initial begin  
    #(period/4 + 1) go = 0;  
    #(5*period/4 - 1) go = 1;  
end
```

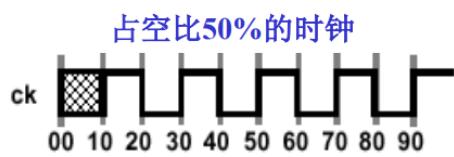
产生的波形（假定period为20）



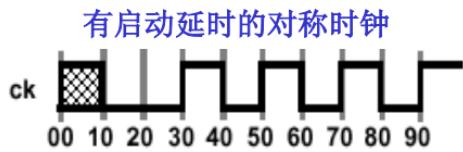
注意：在行为描述中，CK值立刻被影响；而在结构描述中，在传播延时后才输出正确波形。

### 建立时钟 (period = 20)

```
reg ck;  
always begin  
    #( period/2) ck = 0;  
    #( period/2) ck = 1;  
end
```



```
reg ck;  
initial begin  
    #( period/2)  
    ck = 0;  
    forever  
        #( period/2) ck = !ck;  
end
```



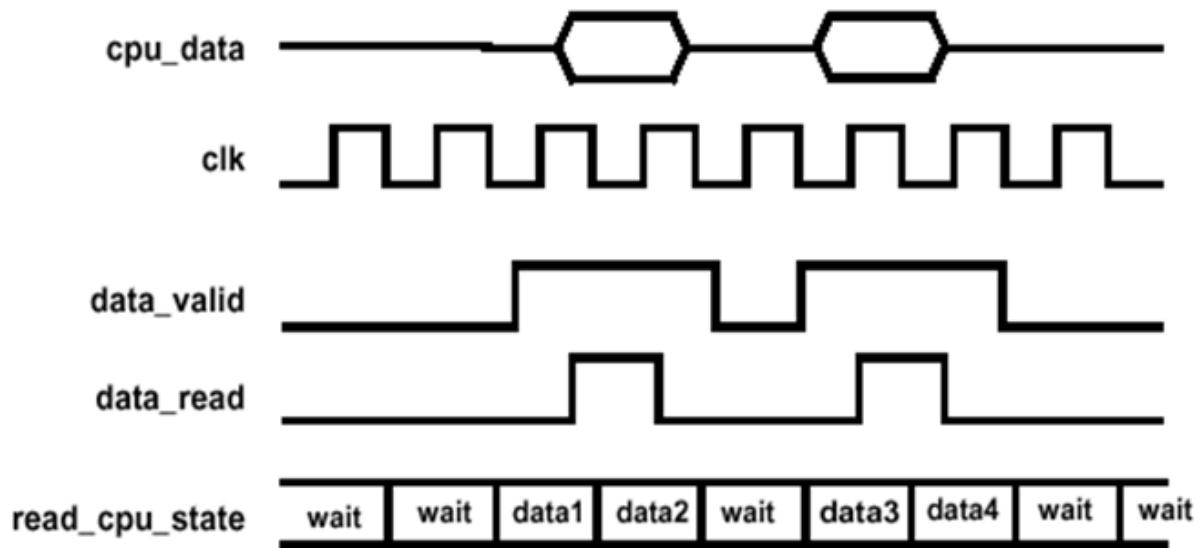
```
reg ck;  
initial begin  
    #(period + 1) ck = 1;  
    #(period/2 - 1)  
    forever begin  
        #(period/4) ck = 0;  
        #(3*period/4) ck = 1;  
    end  
end
```

不规则启动延时的占空比25%的时钟



北京大学

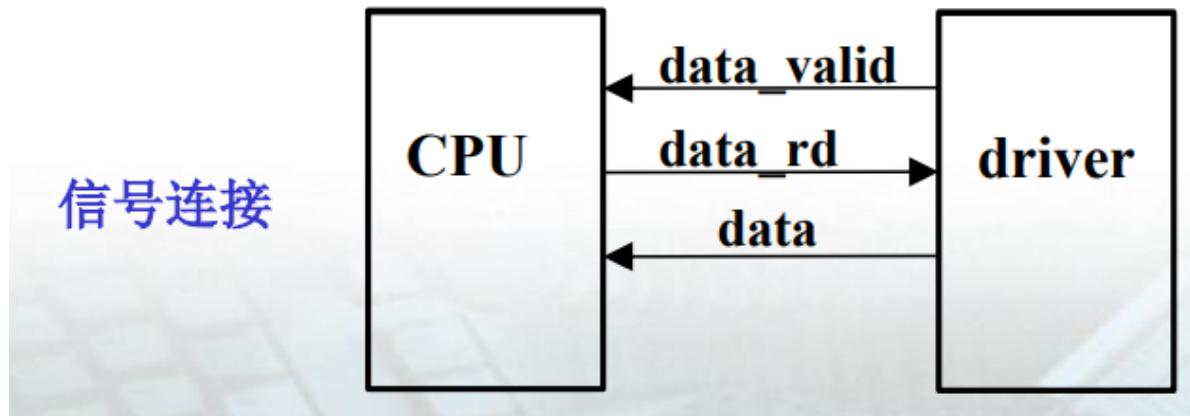
### 使用task



在test bench中使用task可以压缩重复操作，提高代码效率。

```
module bus_ctrl_tb;
reg [7: 0] data;
reg data_valid, data_rd;
cpu u1 (data_valid, data,data_rd);
initial begin
    cpu_driver (8'b0000_0000);
    cpu_driver (8'b1010_1010);
    cpu_driver (8'b0101_0101);
end
```

```
task cpu_driver;
input [7:0] data_in;
begin
#30 data_valid = 1;
wait (data_rd == 1);
#20 data = data_in;
wait (data_rd == 0);
#20 data = 8'hzz;
#30 data_valid = 0;
end
endtask
endmodule
```



## 使用task压缩重复操作，提高代码效率。

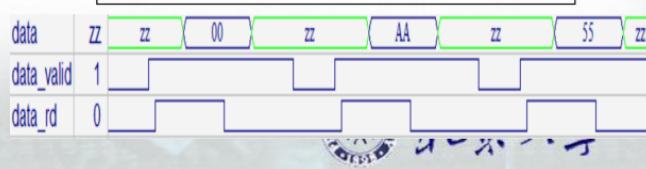
```
'timescale 1ns/1ns
module cpu (
    input [7: 0] data,
    input     data_valid,
    output reg  data_rd
);

initial data_rd = 1'b0;
always @(posedge data_valid)
begin
    #5 data_rd = 1'b1;
    #30 data_rd = 1'b0;
end
endmodule
```

```
'timescale 1ns/1ns
module bus_ctrl_tb;
    reg [7: 0] data;
    reg     data_valid,
            data_rd;
```

```
cpu u1 (data, data_valid, data_rd);
```

```
initial begin
    cpu_driver (8'b0000_0000);
    cpu_driver (8'b1010_1010);
    cpu_driver (8'b0101_0101);
end
task cpu_driver;
input [7:0] data_in;
begin
    #30 data_valid = 1;
    wait (data_rd == 1);
    #20 data = data_in;
    wait (data_rd == 0);
    #20 data = 8'hzz;
    #30 data_valid = 0;
end
endtask
endmodule
```



问题：

- 什么操作可以容易的在fork...join块做到，而不容易在begin...end块做到？
- 通常怎样产生规则激励和不规则激励？
- 从一个文件中读取激励时使用什么数据类型？
- 在行为级时钟模型中能做哪些在门级时钟模型中很难或不能作到的事？

解答：

- fork...join块中不但能够赋值，还可以并行执行循环、条件语句、任务或函数调用。
- 循环或always块能有效地产生规则激励，不规则激励适合用在initial块产生。
- 用寄存器组（存储器）并用\$readmem系统任务从一个文件以读取向量。
- 行为级代码可以很容易地产生一个启动时间不规则的时钟波形，并且可以在时刻零初始化时钟。

## 存储器建模

描述存储器必须做两件事：

- 说明一个适当容量的存储器。
- 提供内容访问的方式，例如：
  - 只读
  - 读和写
  - 写同时读
  - 多个读操作，同时进行单个写操作
  - 同时有多个读和多个写操作，有保证一致性的方法

## 简单ROM描述

下面的ROM描述中使用二维寄存器组定义了一个存储器mem。ROM的数据单独保存在文件my\_rom\_data中，如右边所示。通常用这种方法使ROM数据独立于ROM描述。

```

`timescale 1ns/10ps
module myrom (
    input wire      read_en_,
    input wire [3:0] addr,
    output reg [3:0] read_data
);
reg [3 : 0] mem [15 : 0];

initial
    $readmemb ("my_rom_data", mem);

always @(*( addr or read_en_))
    if (! read_en_)
        read_data = mem[addr];
endmodule

```

my_rom_data
0000
0101
1100
0011
1101
0010
0011
1111
1000
1001
1000
0001
1101
1010
0001
1101

```

reg [7 : 0] mem [15 : 0]; // mem为16x8的寄存器数组
reg MemA [3 : 0]; //MemA是4x1位寄存器数组
time Events [3 : 0]; //time数组
integer int [3 : 0]; //整数数组int[3 : 0]

```

Verilog-1995标准中，不能直接访问一维数组中的某个寄存器数据中的一位，比如要取出下面 array[7][5]，需要将 array[7] 赋给一个reg变量，再从这个变量中取出第5位：

```

reg [7 : 0] array [15 : 0];
reg [7 : 0] arrayreg;
reg reg5;
initial begin
arrayreg = array[7];
reg5 = arrayreg[5];
end

```

## 寄存器和数组

```

`timescale 1ns/1ns
module reg_tb;
reg [3 : 0] RegA; //一个4位的reg类型数据
reg MemA [3 : 0]; //一个reg类型的 4x1 的一维数组
initial begin
RegA = 4'b1101; //合法
MemA = 4'b0010; //非法
MemA[3] = 1'b0;
MemA[2] = 1'b0;
MemA[1] = 1'b1;
MemA[0] = 1'b0;
$display("RegA = %b ", RegA);
$display("MemA = %b ", {MemA[3], MemA[2], MemA[1], MemA[0]});
#100 $finish;

```

```
    end  
endmodule
```

## Verilog-2001标准的数组声明

```
`timescale 1ns /1ns  
module array;  
reg [7 : 0] mem [15 : 0][7 : 0];  
reg [7 : 0] out;  
integer i, j, k;  
initial begin  
for (i = 0; i < 16; i = i + 1)  
for(j = 0; j < 8; j = j + 1)  
mem[i][j] = i + j;  
for (i = 0; i < 16; i = i + 1) begin  
for(j = 0; j < 8; j = j + 1) begin  
out = mem[i][j];  
$write("\t %0d", out);  
end  
$write("\n");  
end  
$display("out = %b", out);  
$write("out = ");  
for( k = 0; k < 8; k = k + 1)  
$write("%b", mem[i-1][j-1][7 - k]);  
$write("\n");  
$finish;  
end  
endmodule
```

可声明包括**wire**类型在内的多维数组。

可任意访问多维数组中数据的**1位或多位**。

0	1	2	3	4	5	6	7
1	2	3	4	5	6	7	8
2	3	4	5	6	7	8	9
3	4	5	6	7	8	9	10
4	5	6	7	8	9	10	11
5	6	7	8	9	10	11	12
6	7	8	9	10	11	12	13
7	8	9	10	11	12	13	14
8	9	10	11	12	13	14	15
9	10	11	12	13	14	15	16
10	11	12	13	14	15	16	17
11	12	13	14	15	16	17	18
12	13	14	15	16	17	18	19
13	14	15	16	17	18	19	20
14	15	16	17	18	19	20	21
15	16	17	18	19	20	21	22
i = 16 j = 8							
out = 00010110							
out = 00010110							

MSB为0的数组

```

module data;
    reg [0 : 15] data; // [MSB : LSB]为[0 : 7]
                    // 0是寄存器高位
    reg [0 : 3] result;

initial begin
    data = 16'b0100_0011_0010_00001; //16'h4321

    for(integer i = 0; i < 16; i++)
        $display(" data[%0d] = %b ", i, data[i]);

    for(integer i = 3; i >= 0; i --) begin
        result = data[i*4 + : 4];
        $display(" result = %b ", result);
    end

    $finish;
end
endmodule

```

**result = 0001**  
**result = 0010**  
**result = 0011**  
**result = 0100**

```

data[ 0] = 0
data[ 1] = 1
data[ 2] = 0
data[ 3] = 0
data[ 4] = 0
data[ 5] = 0
data[ 6] = 1
data[ 7] = 1
data[ 8] = 0
data[ 9] = 0
data[10] = 1
data[11] = 0
data[12] = 0
data[13] = 0
data[14] = 0
data[15] = 1

```

## 存储器数据装入

可以使用循环或系统任务给存储器装入初始化数据

- 用循环给存储器的每个字赋值

```

for (i= 0; i < memsize; i = i+ 1) // initialize memory
    mema[ i] = {wordsize{ 1'b1}};

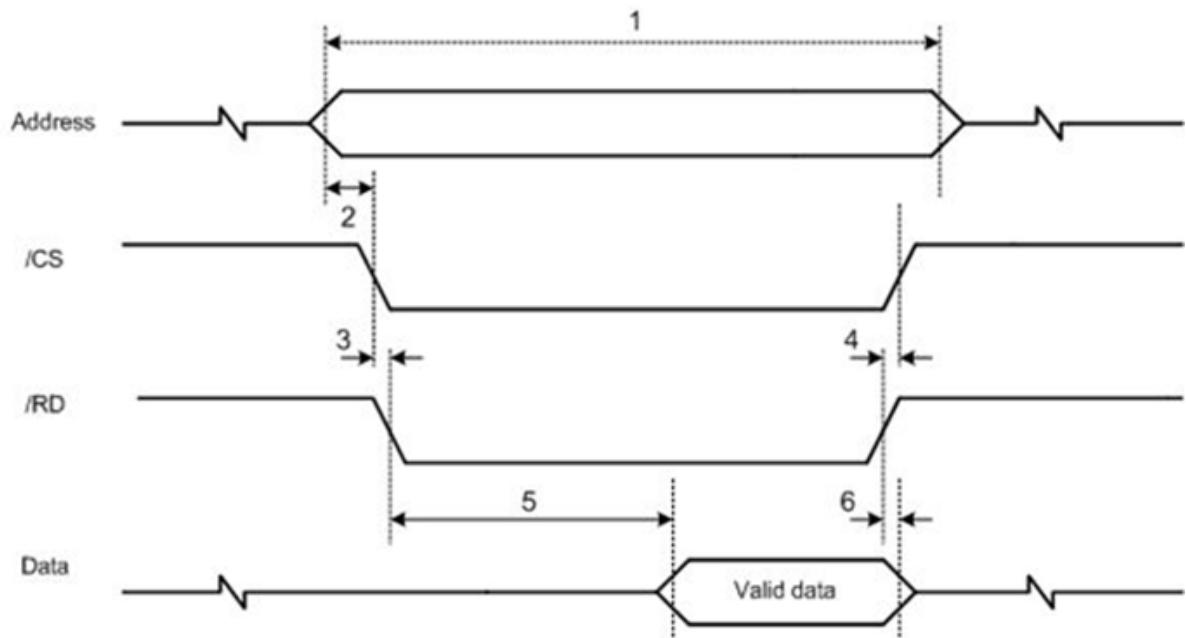
```

- 调用系统任务\$readmem

```
$readmemb("mem_file. txt", mema);
```

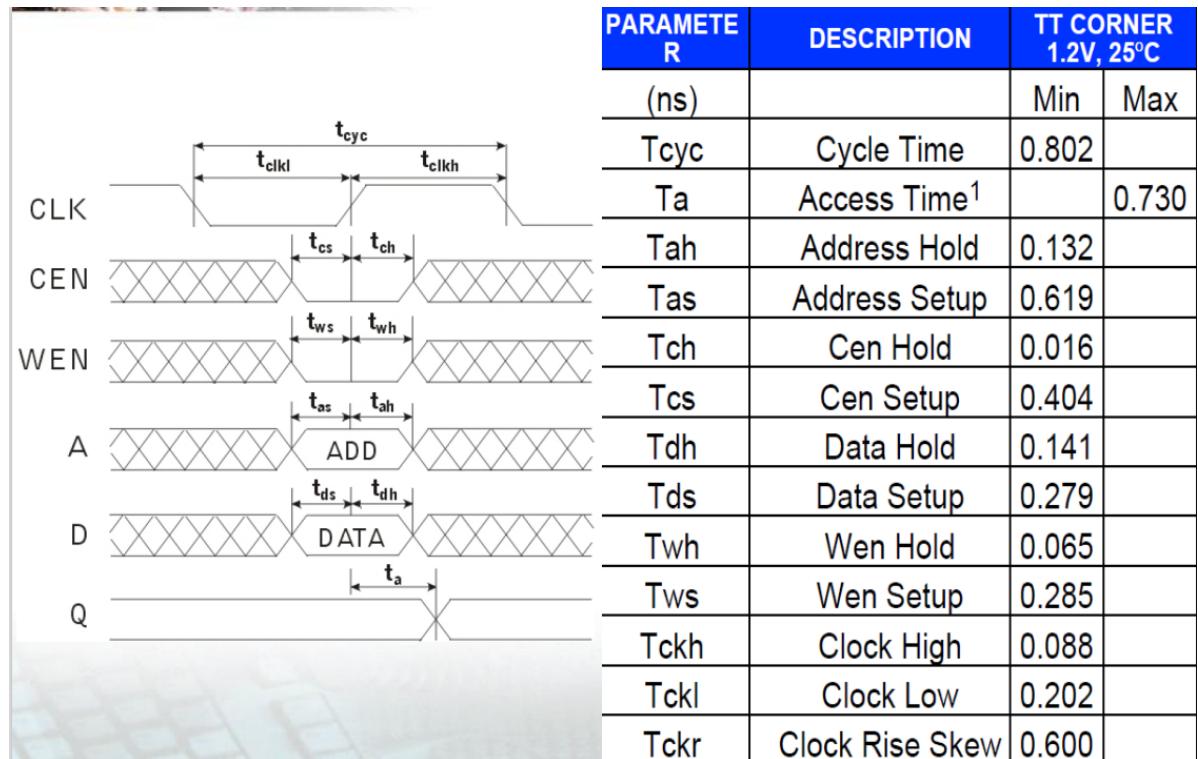
可以用系统任务\$readmem给一个ROM或RAM加载数据。对于ROM，开始时写入的数据就是其实际内容。对于RAM，可以通过初始化，而不是用不同的写周期给每个字装入数据以减少仿真时间。

## 异步存储器接口时序



	描述	最小值	最大值
1	读时序周期	80 ns	-
2	有效地址至 /CS置低时间	8 ns	-
3	/CS置低到 /RD置低时间	-	1 ns
4	/RD置高到 /CS置高时间	-	1 ns
5	/RD置低到有效数据输出时间	-	80 ns
6	/RD置高到数据高阻输出时间	-	1 ns

## 同步SRAM存储器接口时序 - SMIC65nm\_1Kx32



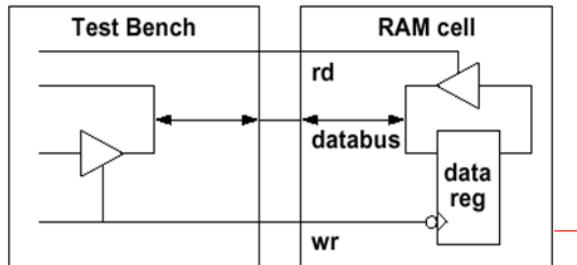
## 简单的异步RAM描述

RAM描述比ROM略微复杂，因为必须既有读功能又有写功能，而读写通常使用同一数据总线。这要求使用新的处理双向数据线的建模技术。在下面的例子中，若读端口未使能，则模型不驱动数据总线；此时若数据总线没有写数据驱动，则总线为高阻态Z。这避免了RAM写入时的冲突。

```
`timescale 1ns /1ns
module mymem (
    inout wire [3:0] data,
    input wire [3:0] addr,
    input wire      rd,
    output wire     wr
);
reg [3:0] memory [15:0]; // 16*4
// 读
assign data = rd ? memory[addr] : 4'bz;
// 写
always @(posedge wr)
    memory[addr] <= data;
endmodule
```

这个描述可综合，但许多工具仅仅产生一个寄存器堆，因此与一个真正的存储器相比耗费更多的面积。

## 异步RAM测试



```
module mymem (
    inout wire [3:0] data,
    input wire [3:0] addr,
    input wire      rd,
    output wire     wr
);
reg [3:0] memory [15:0]; // 16*4
// 读
assign data = rd ? memory[addr] : 4'bz;
// 写
always @(posedge wr)
    memory[addr] <= data;
endmodule
```

```
`timescale 1ns /1ns
module mymem_tb ;
    wire [3:0] data;
    reg [3:0] addr;
    reg      rd,
            wr;

    mymem umymem(
        data,
        addr,
        rd,
        wr
    );
    reg [3 : 0] wdata;
    assign data = wr ? wdata : 4'bz;
initial begin
    wdata = 4'b1010;
    rd = 1'b0;
    wr = 1'b0
    .....
endmodule
```

## 参数化存储器描述

在下面的例子中，给出如何定义一个字长和地址均参数化的只读存储器件。

```
module scalable_ROM #(  
    parameter addr_bits = 8, // 地址总线宽度  
    wordsize = 8, // 字宽  
    words = (1 << addr_bits) // mem容量  
)  
(  
    output [wordsize - 1 : 0] mem_word, // 存储器字  
    input [addr_bits - 1 : 0] address // 地址总线  
,  
);  
  
reg [wordsize - 1 : 0] mem [words - 1 : 0]; // mem声明  
  
// 输出存储器的一个字  
assign mem_word = mem[address];  
  
endmodule
```

例中存储器字范围从0而不是1开始，因为存储器直接用地址线确定地址。也可以用下面的方式声明存储器并寻址。

```
reg [wordsize:1] mem [1:words]; // 从地址1开始的存储器  
// 存储器寻址时地址必须加1  
wire [wordsize:1] mem_word = mem[ address + 1];
```

## 存储器数据赋值

可以使用循环或系统任务给存储器装入初始化数据

- 用循环给存储器的每个字赋值

```
for (i= 0; i < memsize; i = i+ 1) // initialize memory  
  
    mema[ i] = {wordsize{ 1'b1}};
```

- 调用系统任务\$readmem

```
$readmemb("mem_file.txt", mema);
```

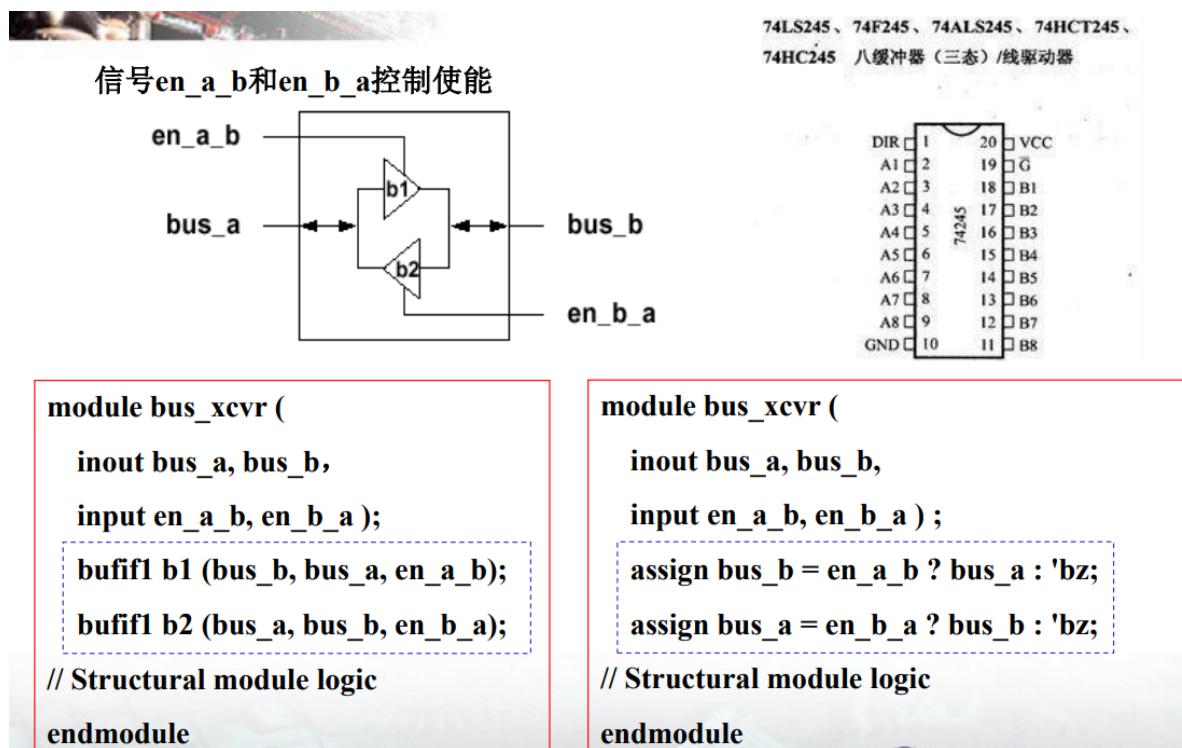
可以用系统任务\$readmem给一个ROM或RAM加载数据。对于ROM，开始时写入的数据就是其实际内容。对于RAM，可以通过初始化，而不是用不同的写周期给每个字装入数据以减少仿真时间

## 使用双向端口

- 用关键词inout声明一个双向端口
  - inout [7:0] databus;
- 双向端口声明遵循下列规则：
  - inout端口不能声明为寄存器类型，只能是net类型。

- 这样仿真器若多个驱动时可以确定结果值。
  - 对inout端口可以从任意一个方向驱动数据。端口数据类型缺省为net类型。不能对net进行过程赋值，只能在过程块外部持续赋值，或将它连接到基本单元。
- 在同一时间应只从一个方向驱动inout端口。
  - 例如：在RAM模型中，如果使用双向数据总线读取RAM数据，同时在数据总线上驱动写数据，则会产生逻辑冲突，使数据总线变为未知。
  - 必须设计与inout端口相关的逻辑以确保正确操作。当把该端口作为输入使用时，必须禁止输出逻辑

## 双向驱动器建模



## 存储器端口建模

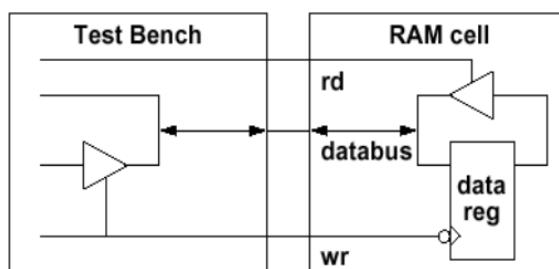
```

'timescale 1ns /1ns
module ram_tb;
  wire [3 : 0] databus;
  reg [3 : 0] addr    ;
  reg rd, wr ;
  reg [3 : 0] data;

  ram uram (databus, addr, rd, wr );
  assign databus = wr ? data : 4'bz;

initial begin
  rd = 0; wr = 0;
  data = 4'hA;
  addr = 4'h5;
#5 wr = 1;
#5 wr = 0;
#5 rd = 1;
#5 rd = 0;
#5 $finish;
end
endmodule

```



```

module ram (
  inout [3 : 0] databus,
  input [3 : 0] addr    ,
  input rd, wr
);
  reg [3 : 0] mem [15 : 0];
  assign databus = rd ? mem[addr] : 4'bzz;
  always @(*( negedge wr))
    mem[addr] <= databus;
endmodule

```

问题：

- 在Verilog中用什么结构定义一个存储器组？
- 如何向存储器加载数据？
- 如何通过一个双向 (inout) 端口传送数据？

解答：

- 在Verilog中将存储器声明为一个一个2维寄存器阵列。
- 可以用系统任务\$readmem或\$readmemb或用过程赋值向存储器加载数据
- 因为inout两端信号必须都是net数据类型，因此只能使用基本单元，子模块，或持续赋值驱动数据。同时还必须注意确保在任何一端不要发生驱动冲突。

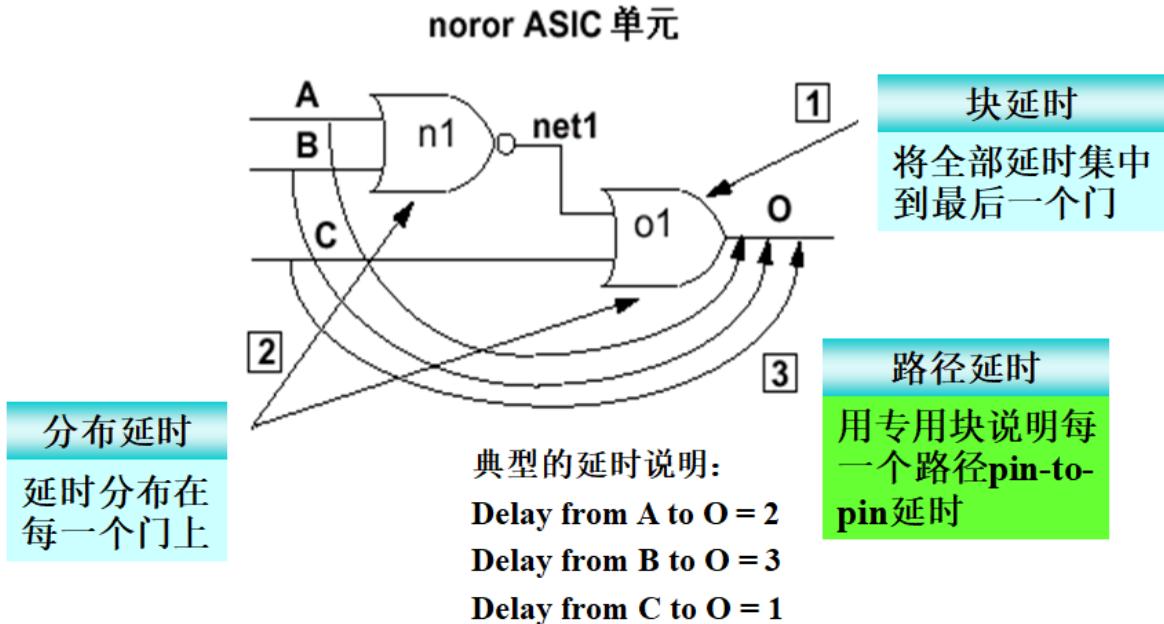
## 延时模型

### 术语及定义

- 模块路径(module path): 穿过模块，连接模块输入(input端口或inout端口) 到模块输出(output端口或inout端口) 的路径。
- 路径延时(path delay): 与特定路径相关的延时
- 时序检查(timing check): 监视两个输入信号的时间关系并进行检查的系统任务，以保证电路能正确工作。
- 时序驱动设计(timing driven design): 从前端到后端的整个设计流程中，用时序信息连接不同的设计阶段

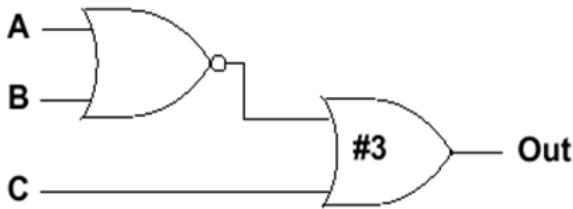
### 延时模型类型(Delay Modeling Types)

延时有三种描述模型：



### 块延时(Lumped Delay)

- 块延时方法是将全部延时集中到最后一个门上。这种模型简单但不够精确，只适用于简单电路。因为当到输出端有多个路径时不能描述不同路径的不同延时。
- 可以用这种方法描述器件的传输延时，并且使用最坏情况下的延时(最大延时)。



```

`timescale 1ns/ 1ns
module noror(Out, A, B, C);
    output Out;
    input A, B, C;
    nor n1 (net1, A, B);
    or #3 o1 (Out, C, net1);
endmodule

```

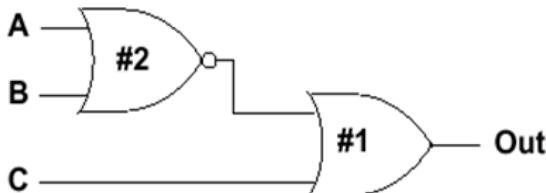
用块延时描述时，不同路径的延时完全相同，左边例中各路径延时为：

A -> Out is 3 ns  
B -> Out is 3 ns  
C -> Out is 3 ns

## 分布延时(Distributed Delays)

分布延时方法是将延时分散到每一个门。在相同的输出端上，不同的路径有不同的延时。分布延时有两个缺点：

- 在结构描述中随规模的增大而变得异常复杂。
- 仍然不能描述基本单元(primitive)中不同引脚上的不同延时。



```

`timescale 1ns/ 1ns
module noror(Out, A, B, C);
    output Out;
    input A, B, C;
    nor #2 n1 (net1, A, B);
    or #1 o1 (Out, C, net1);
endmodule

```

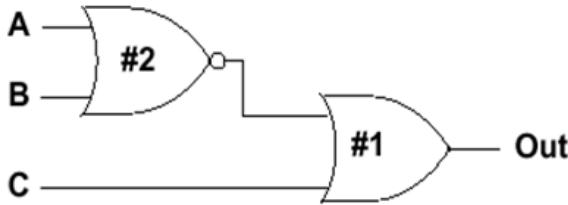
这种描述方法描述的不同路径的延时。例中各路径延时为：

A -> Out is 3 ns  
B -> Out is 3 ns  
C -> Out is 1 ns

## 路径延时(Module Path Delays)

在专用的specify块描述模块从输入端到输出端的路径延时。

- 精确性：所有路径延时都能精确说明。
- 模块性：时序说明与功能描述分开说明
  - 功能验证独立于时序验证。
  - 在不同的抽象级中保持不变。



例中各路径延时为：

- A -> Out is 2 ns
- B -> Out is 3 ns
- C -> Out is 1 ns

```
module noror( O, A, B, C);
    output O;
    input A, B, C;
    nor n1 (net1, A, B);
    or o1 (O, C, net1);
    specify
        (A => O) = 2;
        (B => O) = 3;
        (C => O) = 1;
    endspecify
endmodule
```

## Specify块

- specify块定义了模块的时序部分
  - 时序信息和功能在不同的块中描述，这样功能验证独立于时序验证。specify块在不同的抽象级中保持不变。
  - 功能描述中的延时，如#delay在综合时不起作用
- 由specify开始，到endspecify结束，并且在模块内部
- specify块可以：
  - 描述穿过模块的路径及其延时
  - 使用关键字specparam在specify中进行参数声明
  - 定义特定模块或特定模块路径的脉冲过滤限制
  - 描述时序检查以保证器件的时序约束能够得到满足

## 精确延时控制

- 说明门和模块路径的上升(rise)、下降(fall)和关断(turn-off)延时

```
and #(2, 3)  (out, in1, in2, in3); // rise, fall
bufif0 #( 3, 3, 7) (out, in, ctrl); // rise, fall, turn- off
(in => out) = (1, 2); // rise, fall
(a => b) = (5, 4, 7); // rise, fall, turn-off
```

- 在路径延时中可以说明六个延时值(0 ->1, 1 ->0, 0 ->Z, Z ->1, 1 ->Z, Z ->0)

```
(C => Q) = (5, 12, 17, 10, 6, 22);
```

- 在路径延时中说明所有12个延时值(0 ->1, 1 ->0, 0 ->Z, Z ->1, 1 ->Z, Z ->0, 0 ->X, X ->1, 1 ->X, X ->0, X ->Z, Z ->X)

```
(C => Q) = (5, 12, 17, 10, 6, 22, 11, 8, 9, 17, 12, 16);
```

- 上面所说明的每一个延时还可细分为最好、典型、最坏延时。

```
or #( 3.2:4.0:6.3) o1( out, in1, in2); // min: typ: max
not #( 1:2:3, 2:3:5) (o, in); // min: typ: max for rise, fall
(b => y) = (2:3:4, 3:4:6, 4:5:8); // min: typ: max for rise, fall, and turnoff
```

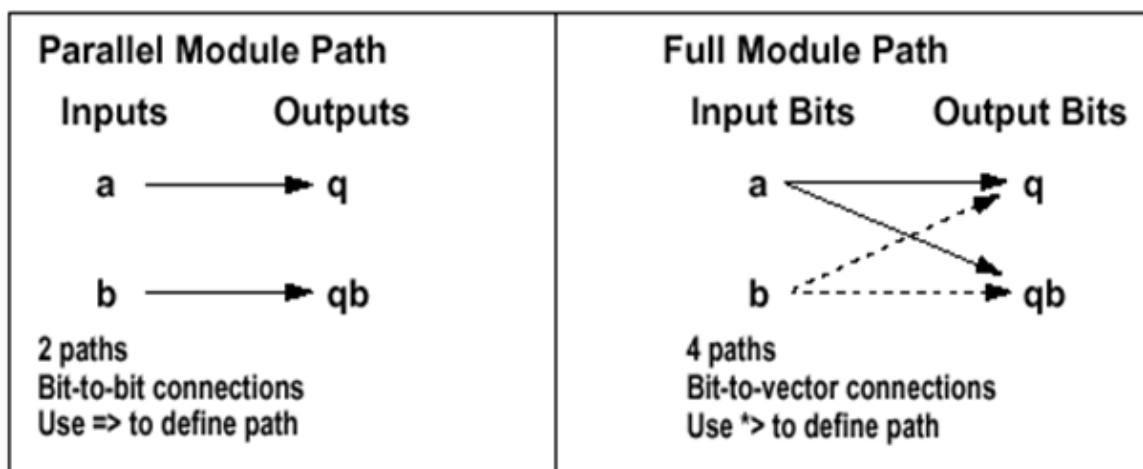
## 最坏延时检测set up, 最好检测hold time

延时说明定义的是门或模块的**固有延时**。输入上的任何变化要经过说明的延时才能在输出端反映出来。如果没有延时说明，则**基本单元**的延时为0。分布**关断延时**只对三态基本单元有效。

- 上升延时是输出转换为 1 时的延时
- 下降延时是输出转换为 0 时的延时
- 关断延时输出转换为 Z 时的延时
- 到X的延时是最小延时，而从X到其它值的转换使用最大延时
  - 如果说明了上升、下降和关断延时，则1->X的延时使用上升和关断延时的最小值。X->0的延时为下降延时；X->Z的延时为关断延时。
  - 如果只说明了上升和下降延时，则1->X和X->0使用下降延时，X->Z使用上升和下降延时的最小延时
  - 如果只说明了一个延时，则所有跳变使用这个延时。
  - 如果说明了六个延时，则1->X使用1->X和1->Z中最小延时；X->0使用1->0和X->0的最大延时；X->Z使用1->Z和0->Z中的最大延时。

## 模块路径的并行连接和全连接

- 路径说明必须括在圆括号内
- \*->表示全连接，也就是所有输入连接到所有输出
- =>表示并行连接，也就是信号对之间的连接



(a, b => q, qb) = 15;  
等价于：  
(a => q) = 15;  
(b => qb) = 15;

(a, b \*-> q, qb) = 15;  
等价于：  
(a => q) = 15;  
(b => q) = 15;  
(a => qb) = 15;  
(b => qb) = 15;

路径延时说明的例子：

```

// 从 a 到 out 和从 b 到 out的路径延时说明
(a, b => out) = 2.2;
(a => out1, out2) = 2.2; // 使用错误, 可使用 (a *> out1, out2) = 2.2;

// 从 r 到 o1 和 o2 的上升、下降延时说明
(r *> o1, o2) = (1, 2);

// 从 a[1] 到 b[1] 和 从 a[0] 到 b[0] 的路径延时说明
(a[ 1: 0] => b[ 1: 0]) = 3; // 并行连接

// 从 a 到 o 的全路径延时说明
(a[7: 0] *> o[7: 0]) = 6.3; // full connection

```

## 路径的极性声明

简单路径可以声明极性。

- “+”表示正极性 (positive polarity)
- “-”表示负极性 (negative polarity)
- 极性声明是可选项
- 正极性表示从输入到输出是同相, 即不会发生信号反转
- 负极性表示异相, 信号会发生反转
- 极性声明不会影响仿真, 但可能会被其它EDA工具, 如时序验证工具 使用。

例如: (V + => W) = (3, 4, 5) ;

## 边沿敏感路径声明

边沿敏感路径, 是指输入端口是时钟信号, 并且需指定触发边沿 (posedge /negedge) 。

例1: (posedge clk => (out +: in)) = (1, 2);

表示从clk的上升沿到out的模块路径, 其上升延时是1, 下降延时是2。 “+: in”表示 out = in, 且是同相传输。

例2: (negedge clk => (out -: in)) = (1, 2);

表示从clk的下降沿到out的模块路径, 其上升延时是1, 下降延时是2。 (out -: in)表示out= -in, 是反相传输。

例3: (clk => (out : in)) = (1,2);

表示从clk到out的模块路径, 其上升延时是1, 下降延时是2。 从in到out的数据路径的传输极性是不确定的, 同相或者反相。

## 状态依赖路径延时SDPD

状态依赖路径延时在说明的条件成立时赋予路径一个延时。

有时路径延时可能依赖于其它输入的逻辑值。 SDPD就是用于说明这种情况。 在例子中, b到x的延时依赖于a的状态。

SDPD说明语法:

*if* <condition> 路径延时说明;

SDPD说明不使用*else*子句。 条件值为X或Z则认为条件成立。 当一个路径中有多个条件成立时使用最小值。

所有输入状态都应说明。若没有说明则使用分布延时(若说明了分布延时)，否则使用零延时。

条件有一些限制，但许多仿真器并不遵循IEEE标准的限制。

```
module XOR2 (x, a, b);
    input a, b;
    output x;
    xor (x, a, b);
specify
    if (a) (b=> x) = (5: 6: 7);
    if (!a) (b=> x) = (5: 7: 8);
    if (b) (a=> x) = (4: 5: 7);
    if (!b) (a=> x) = (5: 7: 9);
endspecify
endmodule
```

## specify块参数

specify块中的参数由关键字`specparam`说明。`specparam`参数和模块中`parameter`定义的参数作用范围不同，并且`specparam`定义的参数不能重载。下面总结了两种参数的差别：

- specify参数
  - 关键字为`specparam`声明
  - 必须在`specify`块内声明
  - 只能在`specify`块内使用
  - 不能使用`defparam`重载
- 模块参数
  - 使用关键字`parameter`声明
  - 必须在`specify`块外声明
  - 只能在`specify`块外使用
  - 可以用`defparam`重载
  - 占用存储器，因为在每个模块实例中复制

```
module noror (O, A, B, C);
    output O;
    input A, B, C;
    nor n1 (net1, A, B);
    or o1 (O, C, net1);
specify
    specparam ao = 2, bo = 3, co = 1;
    (A => O) = ao;
    (B => O) = bo;
    (C => O) = co;
endspecify
endmodule
```

## 路径脉冲控制

使用`specparam`参数`PATHPULSE$`控制模块路径对脉冲的处理。

语法：

```

PATHPULSE$ = (< reject_value>, <error_value>?)
PATHPULSE$< path_source>$< path_destination> =
(< reject_value>, <error_value>?)

```

**specify**

**例子**

```

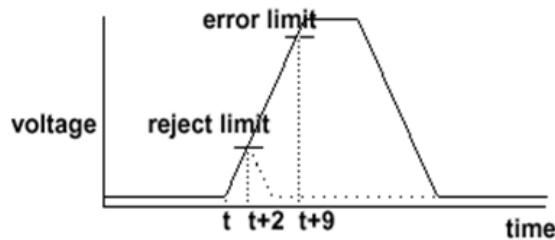
(en => q) = 12;
(data => q) = 10;
(clr, pre *> q) = 4;
specparam
  PATHPULSE$ = 3,
  PATHPULSE$en$q = ( 2, 9 ),
  PATHPULSE$clr$q = 1 ;
endspecify

```



en输入波形

输出q带倾斜波形  
硬件可能产生的波形



- 可以用PATHPULSE\$声明的 specparam参数说明全局脉冲控制
- PATHPULSE\$声明的 specparam参数缩小了指定模块或模块内特定路径的模块路径延时的范围。
- 只声明一个值时, error\_value和reject\_value相同, 如

```
PATHPULSE$ = 3; 等价于 PATHPULSE$ = (3, 3);
```

- 脉冲宽度小于reject\_value的信号将被滤掉, 而小于error\_value的值会使输出产生不定状态。
- 由上面带斜率的波形可以看出, 模块中en信号在时间t发生变化并开始影响q; 若en脉冲在时间t+2结束, 则q没有被完全驱动, q将恢复原值, 如点波形所示。若en脉冲在时间t+9结束, q则可能完成驱动, 也可能没有, 处于未知状态。如果en到t+9一直有效, q将输出新值。

## Verilog时序检查

- 使用时序检查以验证设计的时序
- 时序检查完成下列工作:
  - 确定两个指定事件之间的时差
  - 比较时差与指定的时限
  - 如果时差超过指定时限则产生时序不能满足的报告。这个报告只是一个警告信息, 不影响模块的输出
- Verilog支持的时序检查有:
  - setup(建立时间)
  - hold (保持时间)
  - pulse width (脉冲宽度)
  - clock period (时钟周期)
  - skew (倾斜)
  - recovery (覆盖)

• 系统任务\$setup在数据变化到时钟沿的时差小于时限则报告一个timing violation, 如

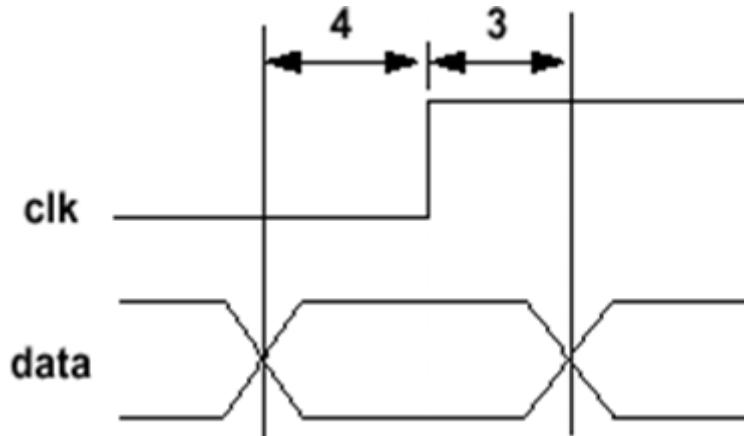
```
$setup( data, posedge clk, 4);
```

• 系统任务\$hold在时钟沿到数据变化的时差小于时限则报告一个timing violation, 如

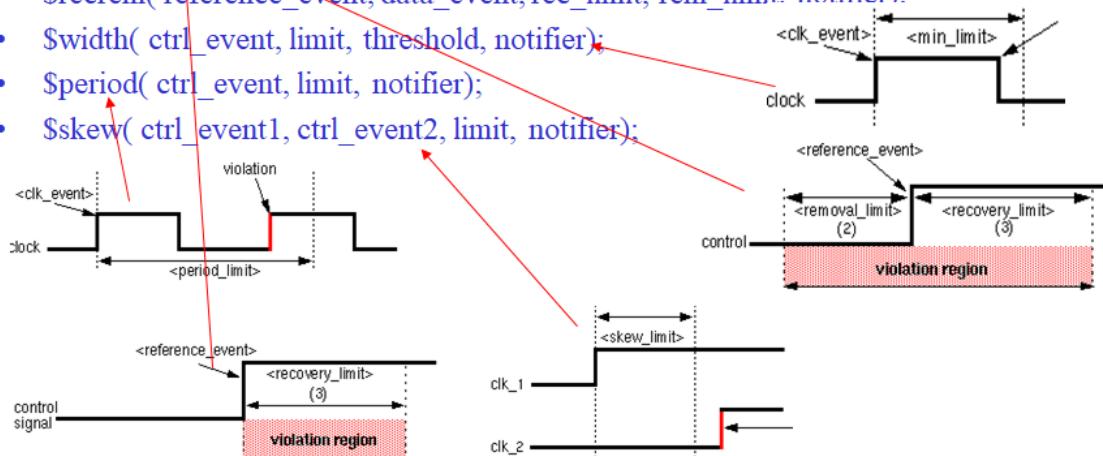
```
$hold( posedge clk, data, 3);
```

- \$setuphold 是 \$setup 和 \$hold 的联合。

```
$setuphold( posedge clk, data, 4, 3);
```



- 建立时间: \$setup( data\_event, clk\_event, limit, notifier);
- 保持时间: \$hold( clk\_event, data\_event, limit, notifier);
- 建立/保持时间: \$setuphold( clk\_event, data\_event, s\_limit, h\_limit, notifier);
- 覆盖: \$recovery(reference\_event, data\_event, limit, notifier);
- \$removal( ctrl\_event1, ctrl\_event2, limit, notifier);
- \$removal( reference\_event, data\_event, rec\_limit, rem\_limit, notifier);
- \$width( ctrl\_event, limit, threshold, notifier);
- \$period( ctrl\_event, limit, notifier);
- \$skew( ctrl\_event1, ctrl\_event2, limit, notifier);



## 时序检查 – 条件时序检查

在条件时序检查中，是否进行时序检查取决于条件表达式的计算值

```

module dff (data, clk, rst, q, qb);
    input data, clk, rst;
    output q, qb;
// instantiate the primitives for the basic flip-flop
    udp_dff( q_int, data, clk, rst);
    buf b1( q, q_int);
    not n1( qb, q_int);
// create timing checks
    specify
        $setup( data, posedge clk &&& rst, 12);
        $hold( posedge clk, data &&& rst, 5);
        $width( posedge clk, 25);
    endspecify
endmodule

```

专用操作符`&&&`在时序检查中设置条件。

只当条件表达式为真时才进行时序检查

当rst为高时进行  
setup和hold检查

width检查  
与rst无关

条件表达式中条件只能是一个标量信号，这个信号可以：

- 用位反操作符 (~) 取反。
- 用等于操作符 (= =或! =) 与一个标量常量进行比较
- 用相同操作符(==或! ==) 与一个标量常量进行比较
- 若条件表达式计算值为1、x或z则认为条件成立。

由于条件时序检查的条件表达式中只能有一个信号，因此需要多个信号产生条件时必须使用哑逻辑使将它们表达为一个内部信号表示才能用于条件时序检查。

- 可以说明并使用一个notifier来显示时序不满足(violation)

```
$setuphold( ref_event, data_event, s_limit, h_limit, NOTIFY);
```

- notifier是可选的
- notifier是一个1位的寄存器
- 时序检查产生violation时，Verilog报告信息并使notifier翻转
- 当时序violation产生时，可以用notifier使输出变为未定义值。
- 有两种方法使notifier影响输出值
- 将notifier作为UDP的一个输入端口
- 在高级行为模块中，不需要将notifier声明为端口也可以对其进行操作。

可能导致所有D触发器全部置x

## notifier举例

```

`timescale 1ns/ 1ns
module dff_notifier (q, ck, d, rst, FLAG);
    input ck, d, rst;
    output q, FLAG;
    reg FLAG; // 1-bit notifier
// dff 网表
    .....
    specify
        (ck => q) = (2: 3: 4);
        $setup( d, posedge ck , 2, FLAG);
    endspecify

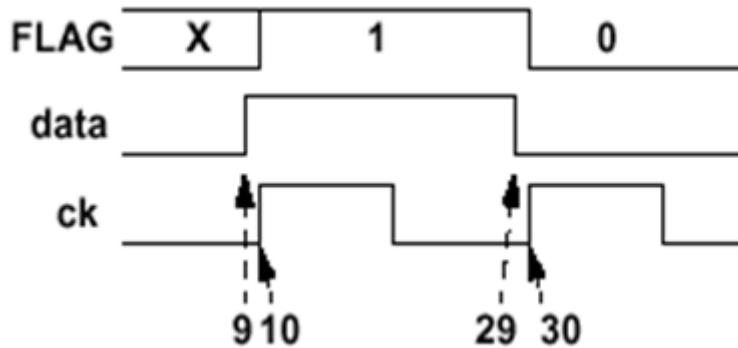
```

```

endmodule

module test;
    reg ck, d, rst;
    dff_notifier (q, ck, d, rst, notifier);
// 产生激励并检查响应
    always @(* notifier) begin
        rst = 1; #10 rst = 0;
    end
endmodule

```



`notifier`初始值为`X`; 第一个产生时序violation时, 其值变为1。其后每次产生时序violation, 其值翻转。

## D触发器时序检查举例

```

`timescale 10 ps / 1 ps
`celldefine
module LVT_DGRNHDV0 ( Q, QN, CK, D, RN );
input CK, D, RN;
output Q, QN;
reg NOTIFIER, NOTIFIERW, NOTIFIERS, NOTIFIERh, NOTIFIERC, NOTIFIERm;
supply1 xsn, EN;
buf X0 (xrn, RN);
buf IC (clk, CK);
udp_edfft I0 (n0, D, clk, xrn, xsn, EN, NOTIFIER);
buf I1 (Q, n0);
not I2 (QN, n0);
and I4 (Deff, D, xrn);
specify
(posedge CK => (Q : D)) = (1.0, 1.0); // arc CK --> Q
(posedge CK => (QN : D)) = (1.0, 1.0); // arc CK --> QN
$width(posedge CK, 1.0, 0, NOTIFIERW);
$setup( D, posedge CK, 1.0, NOTIFIERS);
$hold (posedge CK, D, 1.0, NOTIFIERh);
$recovery(RN, posedge CK, 1.0, NOTIFIERC);
$removal (RN, negedge CK, 1.0, NOTIFIERm);
end

```

```

endspecify
endmodule
`endcelldefine

```

## UDP udp\_edfft定义

```

primitive udp_edfft (out, in, clk, clr_, set_, en, NOTIFIER);
    output out;
    input in, clk, clr_, set_, en, NOTIFIER;

```

### table

```

//in clk clr_ set_ en NOT :Qt:Qt+1
? r 0 1 ? ? :? :0 ;// clock in 0
0 r ? 1 1 ? :? :0 ;// clock in 0
? r ? 0 ? ? :? :1 ;// clock in 1
1 r 1 ? 1 ? :? :1 ;// clock in 1
? * 1 1 0 ? :? :- ;// no changes, not enabled
? * ? 1 0 ? :0 :0 ;// no changes, not enabled
? * 1 ? 0 ? :1 :1 ;// no changes, not enabled
? (x0) ? ? ? :? :- ;// no changes
? (x1) ? 0 ? ? :1 :1 ;// no changes
1 * 1 ? ? ? :1 :1 ;// reduce pessimism
0 * ? 1 ? ? :0 :0 ;// reduce pessimism
? f ? ? ? ? :? :- ;// no changes on negedge clk
* b ? ? ? ? :? :- ;// no changes when in switches
1 x 1 ? ? ? :1 :1 ;// no changes when in switches
? x 1 ? 0 ? ? :1 :1 ;// no changes when in switches
0 x ? 1 ? ? ? :0 :0 ;// no changes when in switches
? x ? 1 0 ? ? :0 :0 ;// no changes when in switches
? b ? ? * ? ? :? :- ;// no changes when en switches
? b * ? ? ? ? :? :- ;// no changes when clr_ switches
? x 0 1 ? ? ? :0 :0 ;// no changes when clr_ switches
? b ? * ? ? ? :? :- ;// no changes when set_ switches
? x ? 0 ? ? ? :1 :1 ;// no changes when set_ switches
? ? ? ? ? * :? :x ;// any NOTIFIER changed

```

### endtable

```
endprimitive // udp_edfft
```

## UDP udp\_edfft 定义

## DFF测试程序

```

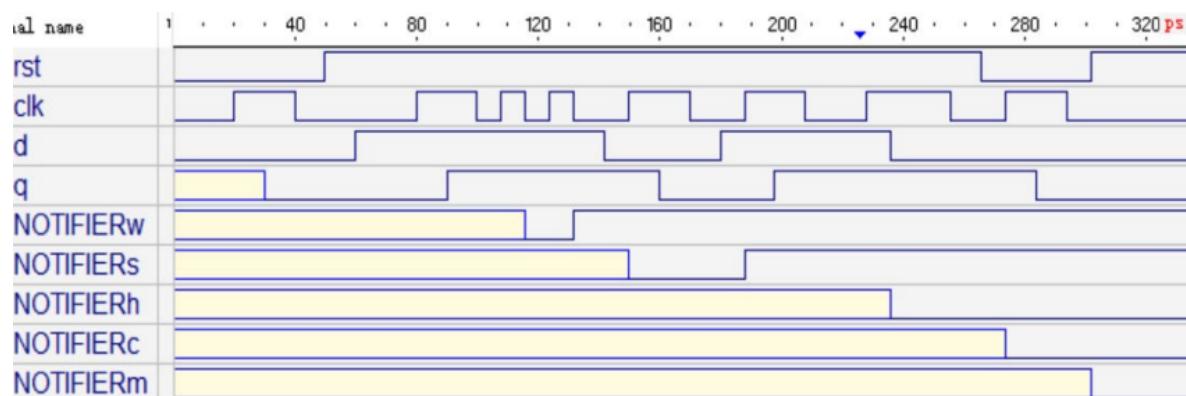
`timescale 10ps / 1ps
module dffa_tb ();
reg d, clk, rst;
;
wire q, qn;
;
dffsr1_udff (
.D (d), .CK (clk), .RN (rst),
.Q (q),
.QN(qn)
);
//
// always #5 clk = !clk
;
initial begin
clk = 0;
d = 0;
rst = 0;
#2 clk = 1;
#2 clk = 0;
#1 rst = 1;
#1 d = 1;

```

```

#2 clk = 1;
#2 clk = 0;
#0.8 clk = 1;
#0.8 clk = 0;
#0.8 clk = 1;
#0.8 clk = 0;
#1 d = 0;
#0.8 clk = 1;
#2 clk = 0;
#1 d = 1;
#0.8 clk = 1;
#2 clk = 0;
#2 clk = 1;
#0.8 d = 0;
#2 clk = 0;
#1 rst = 0;
#0.8 clk = 1;
#2 clk = 0;
#0.8 rst = 1;
#10 $finish;
end
endmodule

```



<b>initial begin</b>		
clk = 0;	#0.8 clk = 1;	#1 rst = 0;
d = 0;	#0.8 clk = 0;	#0.8 clk = 1;
rst = 0;	#1 d = 0;	#2 clk = 0;
	#0.8 clk = 1;	#0.8 rst = 1;
#2 clk = 1;	#2 clk = 0;	#10 \$finish;
#2 clk = 0;	#1 d = 1;	
#1 rst = 1;	#0.8 clk = 1;	
	#2 clk = 1;	
#1 d = 1;	#0.8 clk = 0;	
#2 clk = 1;	#2 clk = 0;	
#2 clk = 0;	#1 d = 1;	
#0.8 clk = 1;	#0.8 clk = 0;	
#0.8 clk = 0;	#2 clk = 1;	



北京大学

## 选择仿真延迟模型

## 选择延时值

- 用下列命令行选项选择延时模型
  - +mindelays
  - +typdelays
  - +maxdelays
- 用下列命令行选项或编译指令指定单位延时、零延时、分布延时或路径延时
  - +delay\_mode\_unit                   `delay\_mode\_unit
  - +delay\_mode\_zero                   `delay\_mode\_zero
  - +delay\_mode\_path                   `delay\_mode\_path
  - +delay\_mode\_distributed           `delay\_mode\_distributed
  - 在单位和零延时模型中，仿真器忽略所有specify块，门延时为单位延时或零延时
  - 分布延时忽略所有specify块，只保留门延时
  - 路径延时忽略门延时，只保留specify块中延时
  - 零延时和路径延时可造成结构零延时反馈，因此使用编译指令影响设计中的指定块。

单位指的是时间精度precision

在设定单位延时和零延时模型时不影响过程中时序控制

# SDF时序标注

## 术语及定义

- CTF: (Compiled Timing Library Format) 编译的时序库格式。特定工艺元件数据的标准格式。
- GCF: (General constraint Format)通用约束格式。约束数据的标准格式。
- MIPD: (Module Input Port Delay)模块输入端口延时。模块输入或输入输出端口的固有互连延时
- MITD: (Multi-source Interconnect Transport Delay)多重互连传输延时。与SITD相似，但支持多个来源的不同延时。
- PLI: (Programming Language Interface) 编程语言界面。基于C的对Verilog数据结构的程序访问。
- SDF: Standard Delay Format.(标准延迟格式)。时序数据OVI标准格式。
- SITD: Single-Source Interconnect Transprot Delay，单一源互连传输延迟。和MIPD相似，但支持带脉冲控制的传输延迟。
- SPF: Standard Parasitic Format. (标准寄生参数格式) 。提取的寄生参数数据的标准格式。

## 精确时序仿真--时序标注

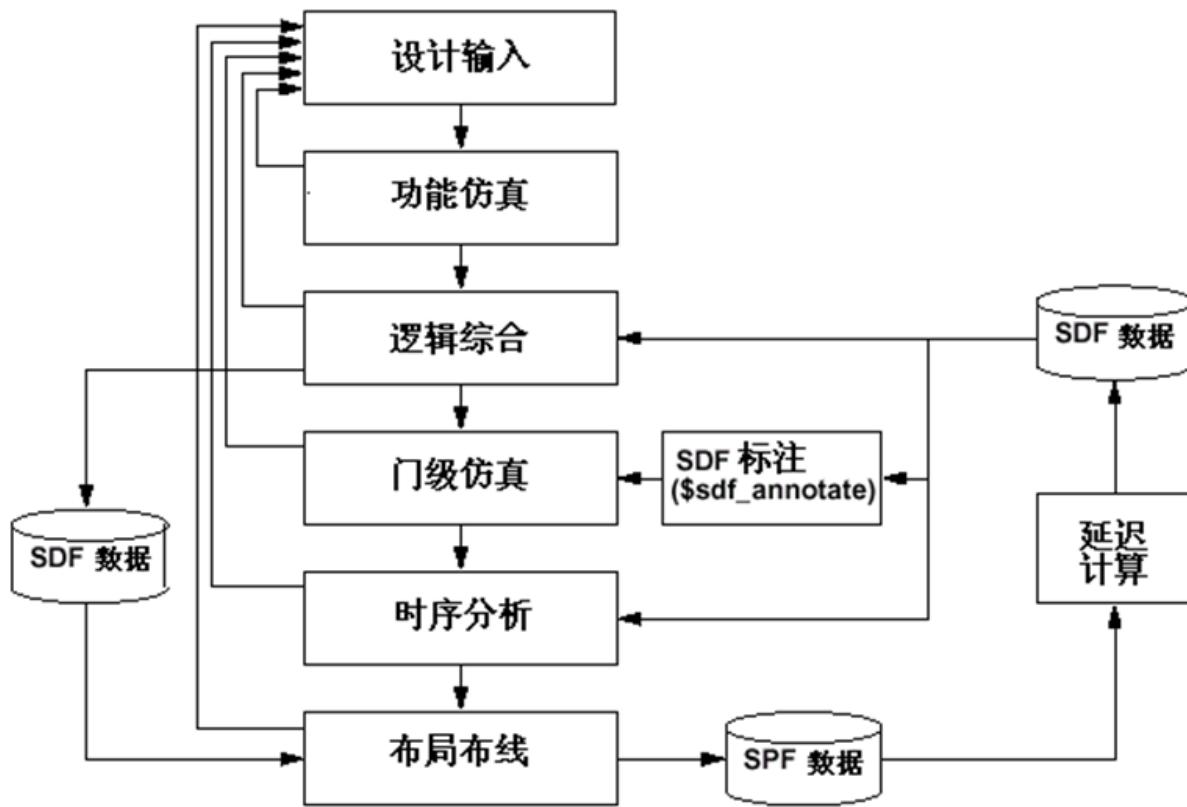
通常的Verilog元件库仅包含固定时序数据。

若要进行精确的时序仿真，还需要的数据有：

- 输入传输时间
- 固有延迟
- 驱动强度
- 总负载
- 互连寄生
- 环境因子
  - 过程
  - 温度
  - 电压

同时还需要仿真最坏情况下的数据和最佳情况下时钟，反过来也要做一次。在没有时序标注时Verilog仿真器做不到这一点。

## 时序数据流



## 时序数据流程

延时计算器需要：

- 综合出来的网表
- 布局布线工具产生的简化的寄生参数

延迟计算器可以产生：

- 粗略延时，仅基于设计连线和层次
- 详细延时，由后端工具提取的寄生参数信息

有时序驱动的自顶而下的设计方法中，时序约束贯穿整个设计流程。与时序数据仅向后反馈的情况，如从布线布线工具反馈到综合工具，相比，这种方法时序收敛速度快。

前端和后端工具使用统一的延迟计算器会提高时序收敛速度。

大多数EDA工具接受标准延迟格式（SDF）。

## SDF（标准延迟格式）

标准延迟格式（SDF）是统一的时序信息表示方法，与工具无关。它可以表示：

- 模块路径延时——条件的和无条件的
- 器件延时
- 互连延时
- 端口延时
- 时序检查

- 路径和net时序约束

注意：在specify块中不能说明互连延迟或输入端口延迟。要用互连延迟仿真，必须进行时序标注。

模块输入端口延迟 (MIPD) 描述的是到模块输入端口或双向端口的延迟。延迟为惯性的且影响三种跳变：到1，到0，和到z。

单一源输入传输延迟 (SITD) 和MIPD相似，但使用传输延迟并且有全局和局部脉冲控制。SITD影响6种跳变：0到1，1到0，0到z，z到0，1到z，z到1。

多重输入传输延迟 (MITDs) 和SITD相似，但允许为每个源-负载通路说明独立延迟。

## SDF(Stand Delay Format)文件

标准延时格式 (SDF) 是一种标准的，与工具无关的表示时序数据的文本格式。SDF文件通常用于Verilog仿真。教程不对SDF做详细介绍。

应注意的是，Verilog仿真器必须能够将SDF文件中的数据标注用于仿真。这些数据包括：

- 增量或绝对延时，如模块路径，器件、内部连接和端口(包括输入端口延时)
- 时序检查，如setup, hold, recovery, skew, width period
- 时序约束，如path
- 条件或无条件模块路径延时
- 设计、实例、类型或库的专用数据
- 比例、环境、工艺及用户定义基本单元

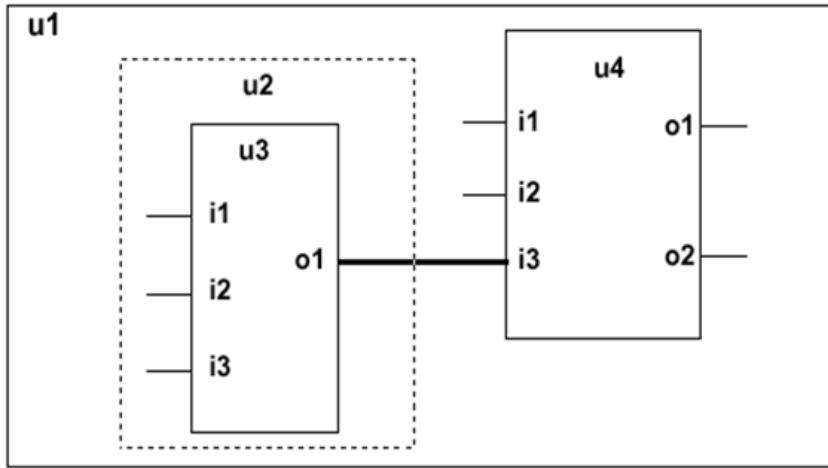
SDF允许不同工具共享延时数据。可以将关键路径信息由综合器传递给布局布线工具，也可将内部连接线延时信息由布局布线工具反传给仿真器。

## SDF举例

```
(DELAYFILE
  (DESIGN "system")
  (DATE "Mon Jun 1 14:54:29 PST 1992")
  (VENDOR "Cadence")
  (PROGRAM "delay_calc")
  (VERSION "1.6a, 4")
  (DIVIDER /) /* hierarchical divider */
  (VOLTAGE 4.5:5.0: 5.5)
  (PROCESS "worst")
  (TIMESCALE 1ns) /* delay time units */
  (CELL (CELLTYPE "system") (INSTANCE block_1) /* top level blocks */
    (DELAY (ABSOLUTE
      (INTERCONNECT D1/z P3/i (. 155::: 155) (. 130::: 130))))))
  (CELL (CELLTYPE "INV") (INSTANCE ) /* all instances of "INV" */
    (DELAY (INCREMENT
      (IOPATH i z (. 345::: 348) (. 325::: 329))))) 可以指定某种单元
    (IOPATH i2 z (. 300::: 300) (. 325::: 325)))) 所有的实例或某个实例
  (CELL (CELLTYPE "OR2") (INSTANCE B1/C1) /* this instances of "OR2" */
    (DELAY (ABSOLUTE
      (IOPATH i1 z (. 300::: 300) (. 325::: 325))
      (IOPATH i2 z (. 300::: 300) (. 325::: 325))))) 延迟可以是绝对的或相对的
  ) // end delay file)
```

## 内部连接延时

内部连接延时是对器件之间连接线延时的估算。例如：



```

(INSTANCE )
(DELAY
(ABSOLUTE
(INTERCONNECT u1.u2.u3.o1 u1.u4.i3 (5:6:7) (5.5:6:6.5) )
)
)
    
```

上面的例子中的内部连接延时说明了一个input到output连接的线延时。

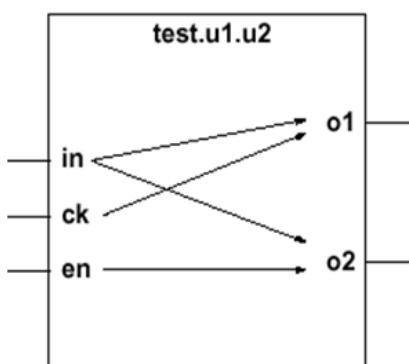
延时分上升、下降和关断延时，每种延时又有最好、典型和最坏值。

- 限定于敏感边沿的iopath，时钟到输出；用上升、下降的最好、典型、最坏值说明。
- 条件iopath, input到output；用上升、下降和关断的最好、典型、最坏值说明

## IOPATH延时

IOPATH延时是器件从输入端口到输出端口的一个合法路径上的延时。

例如：



```

(INSTANCE test.u1.u2)
(DELAY (ABSOLUTE
(IOPATH in o1 (1:2:3) (1:3:4))
(IOPATH(posedge ck) o1 (2:3:4) (4:5:6))
( COND en (IOPATH in o2(2:4:5) (4:5:6) (4:5:7) )
)
)
    
```

在上面IOPATH延时的例子中包括：

- 端口IOPATH，从输入到输出；用最好、典型和最坏值说明上升和下降延时。
- 限定敏感边沿的IOPATH，从时钟到输出，用最好、典型和最坏值描述其上升和下降延时。

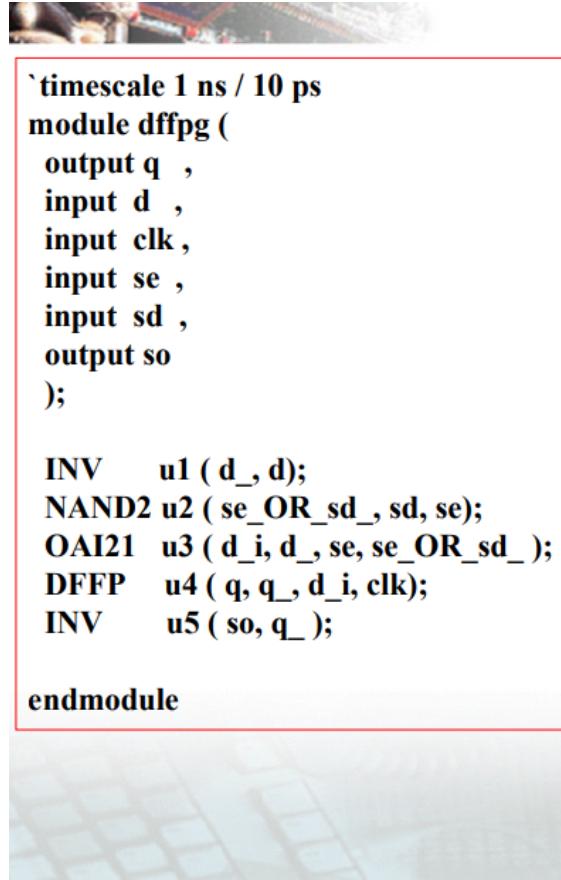
- 条件IOPATH，从输入到输出；用最好、典型和最坏值描述其上升、下降和关断延时。

在上面IOPATH延时的例子中，实例test.u1.u2 需要一个如下面所示的specify块用于反标注。

```
specify
  ( in => o1) = (1:2:3, 1:3:4);
  ( ck => o1) = (2:3:4, 4:5:6);
  if ( en ) ( in => o2) = (2:4:5, 4:5:6, 4:5:7);
endspecify
```

注：SDF文件中的时序信息覆盖specify块中的时序信息

## SDF实例-dffpg模块及测试程序



```

`timescale 1 ns / 10 ps
module dffpg (
  output q ,
  input d ,
  input clk ,
  input se ,
  input sd ,
  output so
);

INV    u1 ( d_,d);
NAND2 u2 ( se_OR_sd_,sd,se);
OAI21 u3 ( d_i,d_,se,se_OR_sd_ );
DFFP   u4 ( q,q_,d_i,clk);
INV    u5 ( so,q_ );

endmodule

```

```

`timescale 1 ns / 1 ns
module DFFP_test;
  wire q , q2;
  wire so, so2;
  reg d ;
  reg clk ;
  reg se ;
  reg sd ;

dffpg udffp (q, d,clk, se, sd, so );
dffpg udffp2 (q2, d,clk , se, sd, so2);

initial $ssdf_annotation( "dffp.sdf", udffp);

initial clk = 0;
always #5 clk = ~clk;
initial begin
  se = 0;
  sd = 0;
  d = 0;
  repeat(5)
    @(negedge clk) d = !d;
#10 $finish;
  end
endmodule

```



```
`timescale 1 ns / 10 ps
`celldefine
module OAI21 ( y, a1, a2, b );
    output y;
    input a1;
    input a2;
    input b;

    wire y_i = ~((a1 | a2) & b);
    buf ( y, y_i );

    specify
        ( a1 => y ) = ( 1.10, 1.10 );
        ( a2 => y ) = ( 1.10, 1.10 );
        ( b => y ) = ( 1.10, 1.10 );
    endspecify

endmodule

`endcelldefine
```

```
`timescale 1 ns / 10 ps
`celldefine
module INV ( y, a );
    output y;
    input a;
    wire y_i = ~a;
    buf ( y, y_i );
specify
    ( a => y ) = ( 1.10, 1.10 );
endspecify
endmodule
```

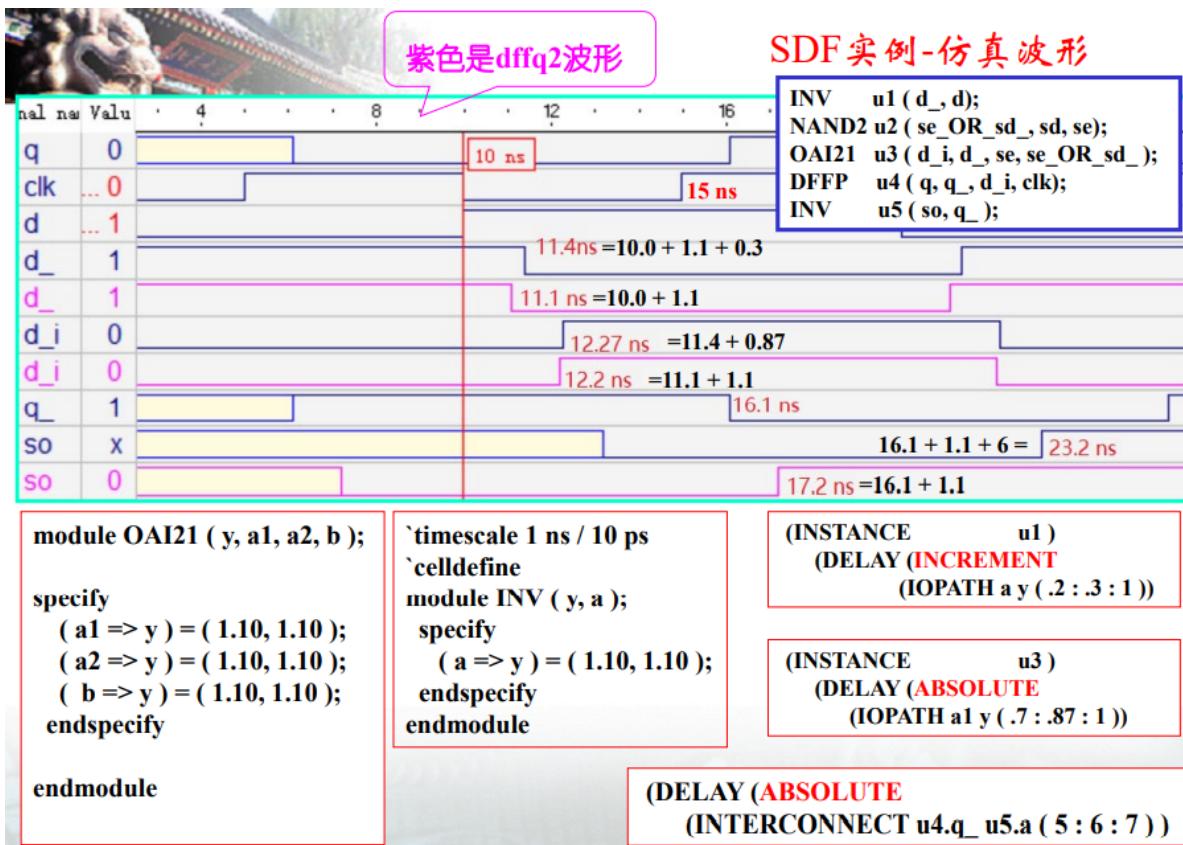
标准单元库

```
module DFFP ( q, q_, d, clk );
    output reg q ;
    output q_ ;
    input d ;
    input clk ;
    always @ ( posedge clk )
        q <= d;
    not ( q_, q );
specify
    ( clk => q ) = ( 1.10, 1.10 );
    ( clk => q_ ) = ( 1.10, 1.10 );
    $setup ( d, posedge clk, 1.0 );
    $hold ( posedge clk, d, 1.0 );
endspecify
endmodule
```

## SDF实例

```
(DELAYFILE
  (SDFVERSION      "SDF Interface version 1.3")
  (DESIGN          "dffp_test")
  (DATE            "Tue Nov 10 09:00:38 EST 1998")
  (VENDOR          "Cadence Design Systems")
  (PROGRAM         "VERILOG")
  (VERSION         "VERITIME 1.3.1")
  (DIVIDER         .)
  (VOLTAGE         )
  (PROCESS          "WCCOM")
  (TEMPERATURE     )
  (TIMESCALE       1ns)
  (CELL
    (CELLTYPE      "INV")
    (INSTANCE       u1)
    (DELAY (INCREMENT
            (IOPATH a y (.2:.3:1))
           ))
  )
)

(CELL
  (CELLTYPE      "OAI21")
  (INSTANCE       u3)
  (DELAY (ABSOLUTE
          (IOPATH a1 y (.7:.87:1))
          (IOPATH a2 y (.7:.87:1))
          (IOPATH b y (.7:.87:1))
        ))
  )
  (CELL
    (CELLTYPE      "")
    (INSTANCE       )
    (DELAY (ABSOLUTE
            (INTERCONNECT u4.q_ u5.a (5:6:7))
          ))
  )
)
```



## SDF标注工具

用系统任务\$*sdf\_annotation*标注SDF时序信息。

可以交互式界面调用这个任务，或在源代码中调任务。

```
$sdf_annotation ("sdf_file", [module_instance,
    "config_file", "log_file", "mtm_spec",
    "scale_factors", "scale_type"]);
```

1. *sdf\_file*: SDF文件名称和绝对或相对路径
2. *module\_instance*: 标注范围。缺省为调用\$*sdf\_annotation*所在的范围
3. *config\_file*: 配置文件的绝对或相对路径。缺省使用预设的设置。
4. *Log\_file*: 日志文件名，缺省为*sdf.log*。可以用+*sdf\_verbose*选项生成一个日志文件。
5. *Mtm\_spec*: 选择标注的时序值，可以是{MINIMUM, TYPICAL, MAXIMUM, TOOL\_CONTROL}之一。缺省为TOOL\_CONTROL(命令行选项)。这个参数覆盖配置文件中MTM关键字。
6. *Scale\_factors*: min:typ:max格式的比例因子，缺省为1.0:1.0:1.0。这个参数覆盖配置文件SCALE\_FACTORS关键字。
7. *Scale\_type*: 选择比例因子；可以是{FROM\_MINIMUM, FROM\_TYPICAL, FROM\_MAXIMUM, FROM\_MTM}之一。缺省为FROM\_MTM。这个参数覆盖配置文件中SCALE\_TYPE关键字。

注意：除*sdf\_file*的所有参数可以忽略。*sdf\_file*可以是任意名字，然后在运行时使用命令行选项+*sdf\_file*选项指定一个*sdf\_file*。

## 执行SDF标注

在下面的例子中，在设计的最顶层进行带比例的SDF标注

```

module top;
    . . .
    initial      $sdf_annotate ("my.sdf", , , , , 1.6:1.4:1.2);
    . . .
endmodule

```

在下面的例子中，对不同的实例分开标注

```

module top;
    . . .
    cpu u1 (. . .
    fpu u2 (. . .
    dma u3 (. . .

    . . .
    initial begin
        $sdf_annotate ("sdffiles/cpu.sdf", u1, , "logfiles/cpu_sdf.log");
        $sdf_annotate ("sdffiles/fpu.sdf", u2, , "logfiles/fpu_sdf.log");
        $sdf_annotate ("sdffiles/dma.sdf", u3, , "logfiles/dma_sdf.log");
    end
    . . .
endmodule

```

和SDF标注相关的命令行选项：

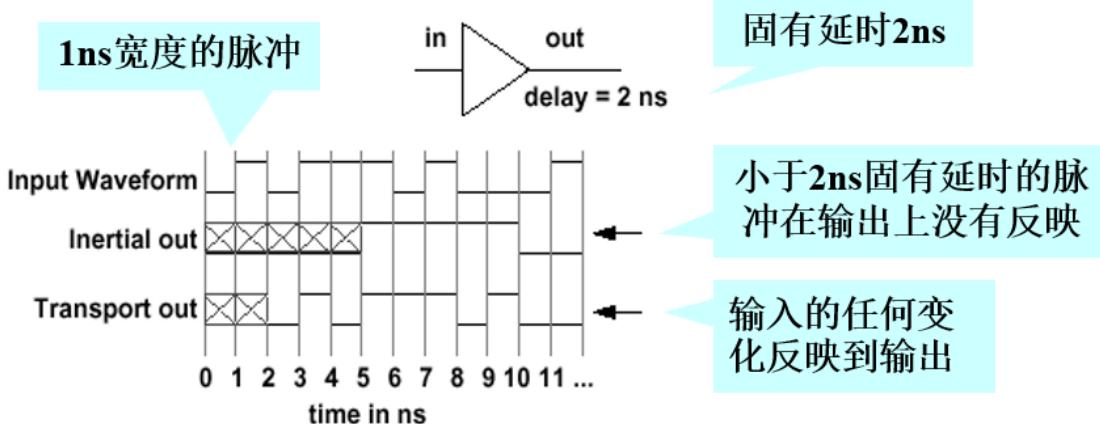
命令	解释
+sdf_cputime	记录用于标注的CPU秒数
+sdf_error_info	显示PLI标注工具错误信息
+sdf_file<filename>	覆盖系统任务\$sdf_annotate中的文件名
+sdf_nocheck_celltype	禁止逐个实例进行单元类型确认
+sdf_no_errors	禁止SDF标注的错误信息
+sdf_nomsrc_int	通知标注工具没有MITD；可以提高性能
+sdf_no_warnings	禁止SDF标注的警告信息
+sdf_verbose	详细记录标注的过程信息

## 惯性(inertial)和传输(transport)延时模型

对于惯性延迟，若路径延时小于门的固有延时，信号会被淹没。

对于传输延迟，输入上每个变化都会反映到输出上。

## 惯性延迟与传输延时的比较

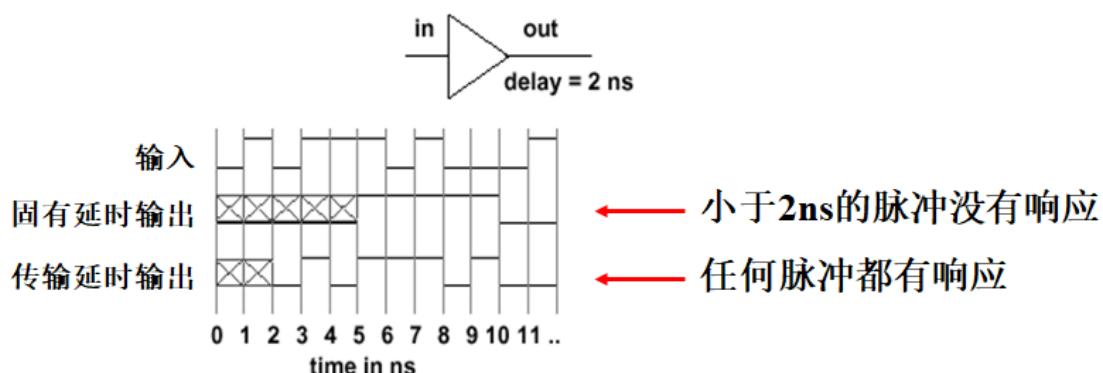


仿真器使用缺省的延迟模型，有的可以用命令行选项，有的用仿真器专用的编译指令指定延迟模型。

## 固有延时和传输延时模型

仿真时可以用固时延时模型或传输延时模型

- 固有延时模型(缺省模型)不传送脉冲宽度小于电路延时的信号。这是开关电路的行为特性
- 在传输延时模型中，输入上的所有变化在路径延时之后反映到输出上。这是传输线的行为特性。
- 采用命令行选项+transport\_path\_delays设置传输延时模型



注意：记住使用+pathpulse用于路径延时控制

## 编译控制的使用

### Verilog模型库

生产商提供了大量的Verilog库。这些库并不是Verilog仿真器专用的，但其库管理格式都基于Verilog-XL风格。

库中每个元件都包括功能及工具专用的时序及工艺信息。

- ASIC和FPGA生产商开发并提供工艺专用库
- 设计人员用库中的元件建立网表
- 仿真器在编译时扫描模型库寻找实例化模块

合成库可以支持多种工具，例如它可以包含下列工具所需要的信息

- 仿真器（如Verilog-XL和NC Verilog）
- 综合器（如Synopsys）
- 时序分析器（如Pearl）

- 故障仿真 (Verifault-XL)

## 单元库建模

建立Verilog单元模型库，需要：

- 每个元件（或单元）用一个module描述
- 将相关的module放在同一个文件或同一个目录中

可以用两种抽象级描述库单元

- 结构级
  - 用 Verilog基本单元或UDP
  - 用于描述组合逻辑或简单的时序逻辑
- 行为级
  - 用过程块或赋值语句
  - 用于描述大的或复杂的元件，如RAM或ROM

库单元的特点：

- 每个库单元的描述在编译指令`celldefine和`endcelldefine之间
- 每个库单元的描述有两部分：
  - 功能描述
  - 时序描述

```

`celldefine
`timescale 1ns / 100ps ←
module full_adder( cout, sum, a_in, b_in, c_in);
  input a_in, b_in, c_in;
  output cout, sum;
// 功能描述
...
// 时序描述
...
endmodule
`endcelldefine

```

在模块定义之前  
插入`timescale定  
义单元所使用  
的时间单位和精度

## Verilog库的使用

在Cadence Verilog仿真器中使用Verilog库：

- 使用库文件
  - 在命令行中使用选项：-v file\_name
- 使用库目录
  - 在命令行中使用选项 -y directory\_name
  - 在命令行中使用选项 +libext+file\_extension

在使用库目录时，如果每个文件都有一个扩展名，则在Cadence Verilog仿真器必须用+libext选项指定其扩展名。仿真器中没有缺省地使用.v作扩展名

使用-v或-y选项指定库时，只编译那些设计中用到的模块。如果在命令行中直接输入库文件名而没有使用-v选项（或在文件中使用编译指令`include`），则库中所有模块都被编译。使用选项大大压缩编译时间及内存空间。在NC Verilog中也压缩了使用的磁盘空间。

如果没有使用-v选项，而是：

- 在命令行中直接输入库文件名，
- 或在文件中使用编译指令`include`，

则库中所有module都被编译。使用选项大大压缩编译时间及内存空间。在 NC Verilog中也压缩了使用的磁盘空间。

## 库文件扫描

每一个-v选项指定一个库文件

```
verilog test.v design.v -v library_file.v
```

```
library_file.v
module and2(...);
    ...
endmodule
module mux(...);
    ...
endmodule
moduledff(...);
    ...
endmodule
```

```
`timescale 1ns/1ps
`celldefine
module BUFX2 (Y, A);
output Y;
input A;
buf I0(Y, A);
specify
// delay parameters
    specparam
        tph0$A$Y = 1.0,
        tph1$A$Y = 1.0;
// path delays
    (A *-> Y) = (tph0$A$Y, tph1$A$Y);
endspecify
endmodule // BUFX2
`endcelldefine
```

## 库目录扫描

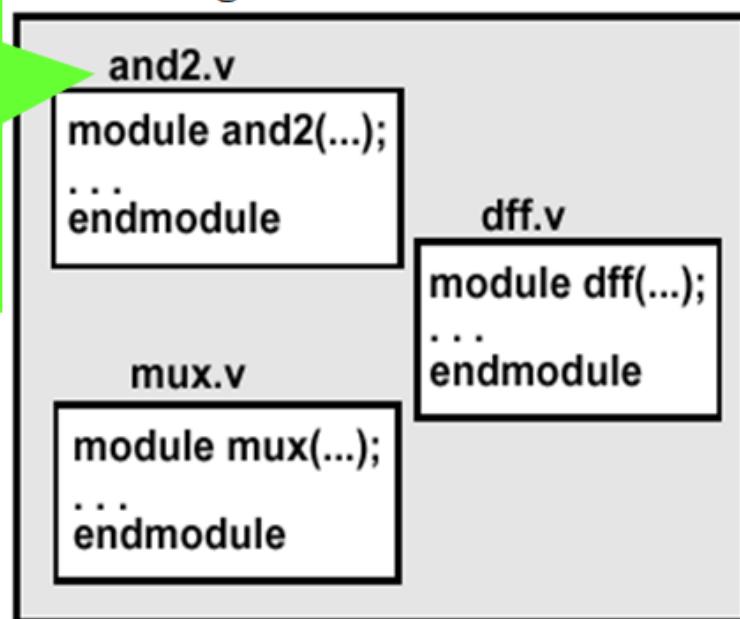
每一个-y选项指定一个库目录。

+libext+选项指定有效的文件扩展名。

```
verilog test.v design.v -y library_directory +libext+.v
```

当把module放到同一个目录时，文件名应与module名相同。文件名的扩展名是可选的

### Verilog模型库目录



## 编译指令`uselib

- 定义设计中使用的库元件（包括UDP）的位置
- 一直有效，直到遇到另一个`uselib或`resetall
- 覆盖任何命令行选项中库的设置。也就是说如果不能在`uselib指定的位置找到元件，仿真器不会再按命令行中-v或-y选项去寻找。

### `uselib语法

```
'uselib library_reference library_reference
```

其中，library\_reference可以是：

file = file\_name\_path  
dir = directory\_name\_path libext = .file\_extension

注意：`uselib使用.v作为缺省扩展名，如-y命令选项不同

使用空`uselib或  
`resetall会产生什么  
效果？

恢复命令行  
-v和-y设置

在`uselib中可以使用“\”进行多行说明

```
'uselib file=/usr1/chrisz/libs/foo.lib \
          dir=/usr1/chrisz/libs/go libext=.v
```

## 编译指令`uselib`使用举例

```
module adder (c_out, sum, a, b, c_in);
    output c_out, sum;
    input a, b, c_in;
`uselib dir=/libs/FAST_LIB/
SN7486 u1 (half_sum, a, b);
`uselib dir=/libs/TTL libext=.v file=/libs/TTL_U/udp.lib
    SN7408 u2 (half_c, a, b);
    SN7408 u3 (tmp, c_in, half_sum);
SN7486 u4 (sum, c_in, half_sum);
    SN7432 u5 (c_out, tmp, half_c);
endmodule
`uselib
```

指定目录库FAST\_LIB  
中寻找实例u1的定义

指定目录库TTL及文件  
库udp.lib寻找其它实例  
的定义

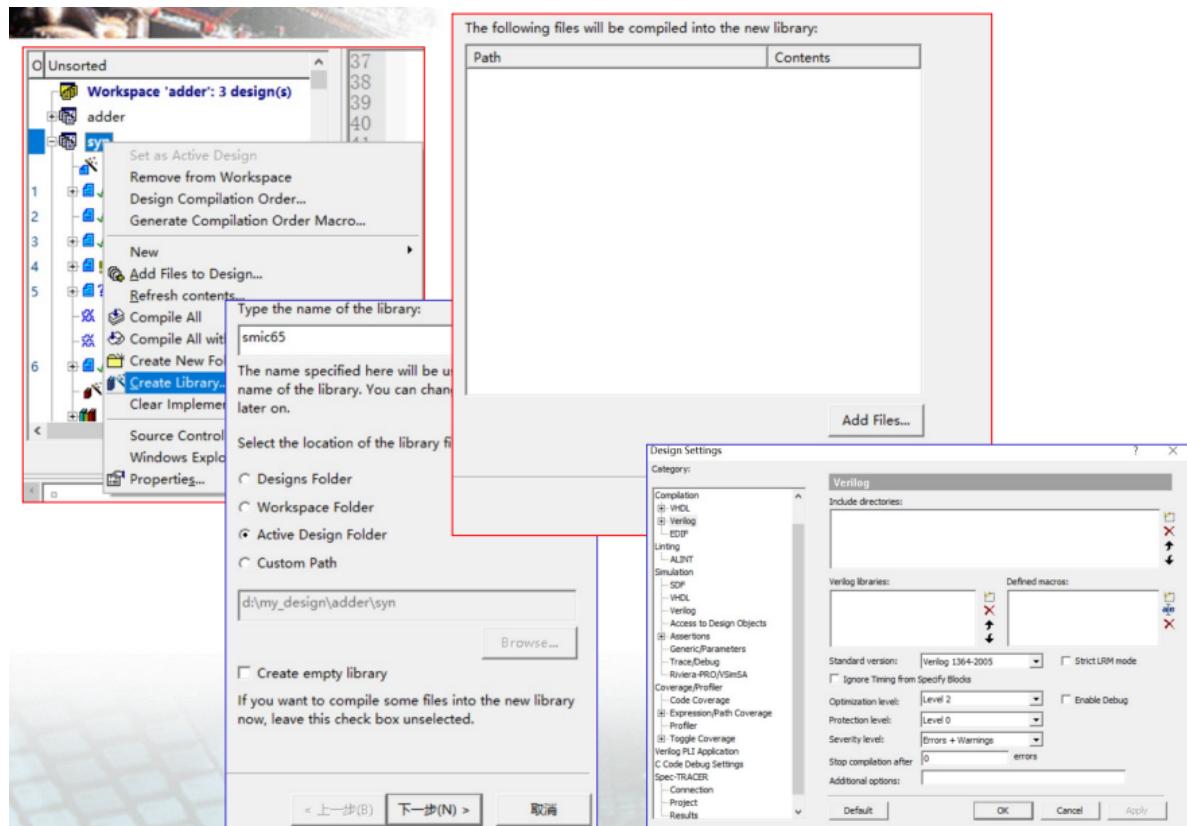
目录库TTL中的单元可以使用文件库udp.lib中定义的单元

在`uselib`中库的指定可以使用由`define`定义的宏进行文本替换。

```
`define TTL_LIB      dir=/libs/TTL  libext=.v
`define TTL_UDP      file=/ libs/ TTL_U/udp.lib
`uselib `TTL_LIB   `TTL_UDP
```

在命令行中用+define+选项给宏一个值。设计易于管理，可移植性高。

## 某Windows仿真器—建库



## 编写与大小无关的源代码

Verilog是对大小写敏感的语言，如sel和SEL是不同的标识符

- Verilog 关键字均使用小写，如input, output
- 标识符中大小写都可以使用，但Sel和sel是不同的标识符
- 仿真时使用-u选项进入大小写不敏感模式。仿真器将所有标识符转换为大写形式，而关键字仍保持为小写。

```
module MUX2_1 (out, a, b, sel);
    output out;
    input a, b, sel;
    not not1( SEL, sel);
    and and1( a1, a, SEL);
    and and2( b1, b, sel);
    or or1( out, a1, b1);
endmodule
```

在正常情况下，左边例子中sel和SEL是不同的信号。若使用-u选项，sel和SEL变为相同的信号。

将产生错误的仿真结果。

如果在大小写不敏感的工具中使用这个模型，则用-u选项可以找出错误。

可以用-d选项输出-u选项产生的大小写不敏感的描述

## 编译指令

尽管编译指令是Verilog语言的一部分，但其作用取决于编译器，因此不同的仿真器中其作用可能不同。

- `resetall将编译指令变为缺省值。
- Cadence Verilog仿真器在遇到`resetall时，文本宏定义不变。要清除文本宏定义，使用`undef macro\_name
- 在使用`include编译指令时，使用+incdir命令行选项指定所包含文件的查找路径。  
+incdir+directory1+directory2+...directoryN

Include不要用路径

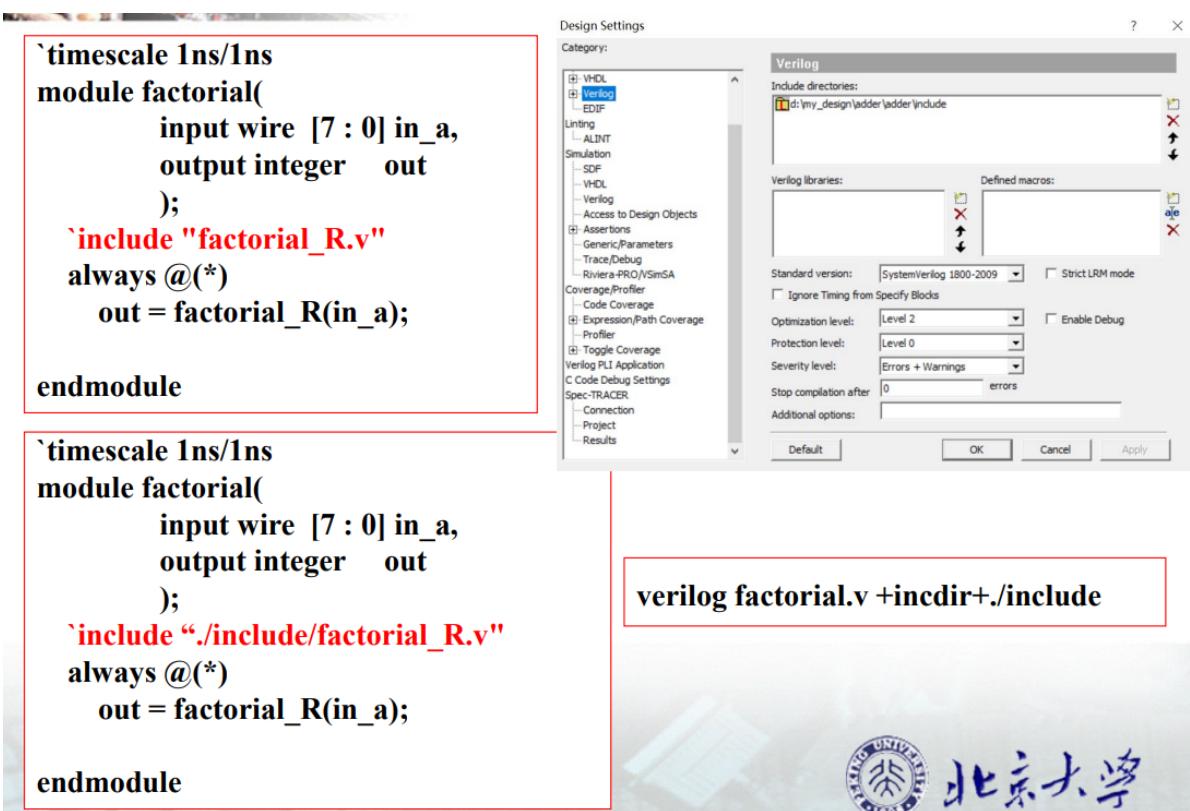
- 仿真器首先查找当前目录，若没有找到再沿指定路径顺序查找。

编译指令从出现时开始有效，直到被覆盖或使其失效。因此编译指令是全局的。

下列编译指令是Verilog IEEE标准中的：

- |  |   |
|--|---|
| <ul style="list-style-type: none"> <li>• 库单元分界<br/>           `celldefine<br/>           `endcelldefine</li> </ul> | <ul style="list-style-type: none"> <li>• 复合编译指令<br/>           `default_nettype<br/>           `include<br/>           `unconnected_drive<br/>           `nounconnected_drive<br/>           `resetall<br/>           `timescale</li> </ul> |
|--|---|

## `include举例



The screenshot shows a Verilog editor interface. On the left, there are two code snippets. The top snippet is:

```

`timescale 1ns/1ns
module factorial(
    input wire [7 : 0] in_a,
    output integer    out
);
`include "factorial_R.v"
always @(*)
    out = factorial_R(in_a);

endmodule

```

The bottom snippet is identical to the top one, also enclosed in a red box.

On the right, a 'Design Settings' dialog box is open, specifically for the 'Verilog' tab. It shows the following settings:

- Include directories:** d:\my\_design\adder\adder\include
- Verilog libraries:** (empty)
- Defined macros:** (empty)
- Standard version:** SystemVerilog 1800-2009
- Optimization level:** Level 2
- Protection level:** Level 0
- Severity level:** Errors + Warnings
- Stop compilation after:** 0 errors
- Additional options:** (empty)

A red box highlights the command **verilog factorial.v +inadir+./include** at the bottom of the editor window.

## 定义文本宏文件举例

FIFO\_control\_define.v

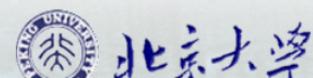
```

`ifndef FIFO_CONTROL

`define FIFO_CONTROL

//Tx register address
`define TXSOF0      8'b0000_0000
`define TxFramehead00 8'b0000_0001

```



```

`define TxFramehead01 8'b0000_0010
`define TxFramehead02 8'b0000_0011
`define TxFramehead03 8'b0000_0100
`define TxFramehead04 8'b0000_0101
`define TxFramehead05 8'b0000_0110
`define TXEOF0       8'b0000_0111

`endif

```

verilog FIFO.v FIFO\_ctrl.v +inmdir+./include

```

`timescale 1ns /1ns
`include "FIFO_control_define.v"
module FIFO (
.....
);
.....
endmodule

```

```

`timescale 1ns /1ns
`include "FIFO_control_define.v"
module FIFO_ctrl (
.....
);
.....
endmodule

```

## 实例化不同module

```

`ifdef FPGA //实例化 Alrera FPGA的64x8 RAM2P
SRAM64x8 uSRAM64x8 (
    .byteena_a ( 1'b0 ),
    .clock ( clk ),
    .data ( DB ),
    .rdaddress ( AA ),
    .rdn ( CENA ),
    .wraddress ( AB ),
    .wren ( CENB ),
    .q ( QA )
);
`else //实例化SMIC SRAM2P
S65NLLHS2PH64x8 uS65NLLHS2PH64x8 (
    .QA ( QA ),
    .CLKA ( clk ),
    .CLKB ( clk ),
    .CENA ( CENA ),
    .CENB ( CENB ),
    .BWENB ( 8'b0 ),
    .AA ( AA ),
    .AB ( AB ),
    .DB ( DB )
);
`endif

```

# 定义文本宏

在命令行定义文本宏：+define+命令行参数

语法：+define+MACRO\_NAME="MACRO\_TEXT"

注意：文本宏的覆盖可能影响设计的结构，可能强制NC Verilog重新编译全部或部分设计

- 文本宏中字符串长度没有限制。
- 清除文本宏定义，使用：

`undef macro\_name

- 清除所有文本宏定义，使用

`undefall

```
verilog test. v +define+gate="or"
`define gate and
module test;
    reg a, b;
    `gate (c, a, b);
    initial
        begin
            a= 0; b= 1;
            $monitor ($time,, c, a, b);
            #1 $finish;
        end
endmodule
```

## 复习

问题：

- 当仿真器遇到编译指令`resetall时将所有编译指令置为缺省值吗？
- 使用什么选项指定库的名字？
- 如果仿真器没有在编译指令`uselib指定的库中找到实例的定义，它会去哪里寻找？

解答：

- 不是。当使用编译指令`resetall时，IEEE规范没有说明如何处理文本宏。Cadence Verilog仿真器对文本宏不作处理。要重文本宏，使用编译指令`undef。
- 使用-v选项和/或-y及+libext+选项。
- 不会再去别的位置查找。

# 高性能编码风格

## if语句

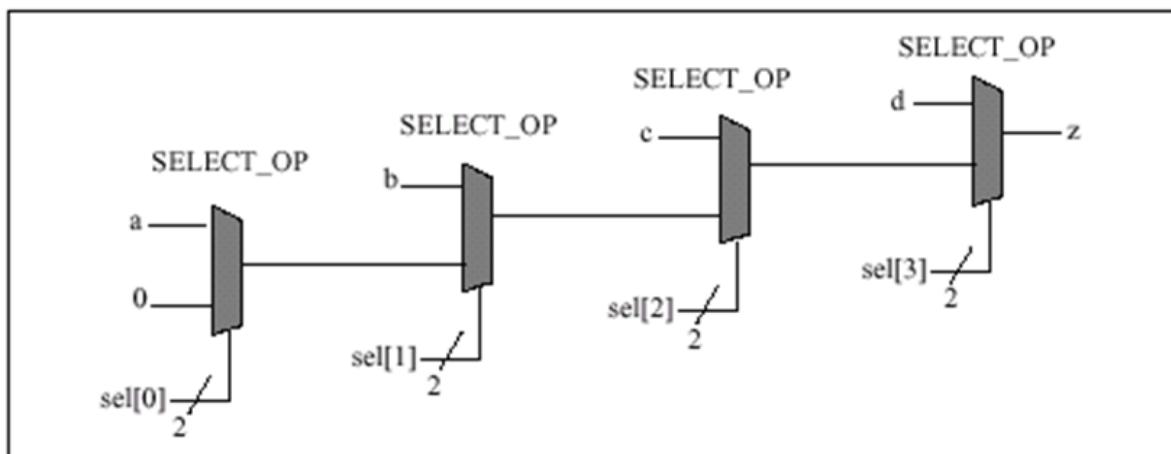
### 例1.1a 单个if语句

```
module single_if(  
    input wire      a,  
                b,  
                c,  
                d,  
    input wire [3:0] sel,  
    output reg       z  
);  
  
    always @(*) begin  
        if (sel[3])      z = d;  
        else if (sel[2]) z = c;  
        else if (sel[1]) z = b;  
        else if (sel[0]) z = a;  
        else             z = 0;  
    end  
endmodule
```

### 例1.1b 多重if语句

```
module single_if(  
    input wire      a,  
                b,  
                c,  
                d,  
    input wire [3:0] sel,  
    output reg       z  
);  
  
    always @(*) begin  
        z = 0;  
        if (sel[0]) z = a;  
        if (sel[1]) z = b;  
        if (sel[2]) z = c;  
        if (sel[3]) z = d;  
    end  
endmodule
```

注意代码的优先级



### case语句

## 例1.2 case语句

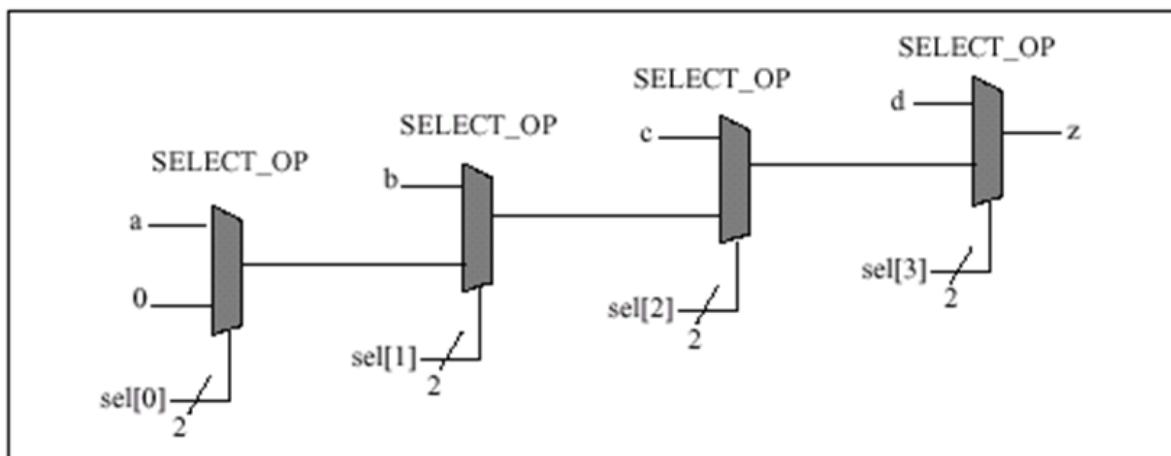
```
module single_if(  
    input wire      a,  
                  b,  
                  c,  
                  d,  
    input wire [3:0] sel,  
    output reg       z  
);  
    always @(*) begin  
        casex (sel)  
            4'b1xxx: z = d;  
            4'b01xx: z = c;  
            4'b001x: z = b;  
            4'b0001: z = a;  
            default: z = 1'b0;  
        endcase  
    end  
endmodule
```

casex具有使用无关项的优点，不用列出sel的所有组合。

## 晚到达信号处理

设计时通常知道哪一个信号到达的时间要晚一些。这些信息可用于构造HDL，使到达晚的信号离输出近一些。

下面的例子中，针对晚到达信号重新构造if和case语句，以提高逻辑性能。



## 晚到达的是数据信号-无优先级

顺序if语句可以根据关键信号构造HDL。在例1.1a中，输入信号d处于选择链的最后一级，也就是说d最靠近输出。

假如信号b\_is\_late是晚到达信号，我们就要重新构造例1.1a使其最优化。

### 原if结构

```
module single_if(a, b, c, d, sel, z);
    input a, b, c, d;
    input [3:0] sel;
    output z;
    reg z;

    always @ (a or b or c or d or sel)
        begin
            if (sel[3])      z = d;
            else if (sel[2]) z = c;
            else if (sel[1]) z = b;
            else if (sel[0]) z = a;
            else             z = 0;
        end
    endmodule
```

### 无优先级的if结构

```
module single_if(a, b, c, d, sel, z);
    input a, b, c, d;
    input [3:0] sel;
    output z;
    reg z;

    always @ (a or b or c or d or sel)
        begin
            if (sel[1])      z = b_is_late;
            else if (sel[3]) z = d;
            else if (sel[2]) z = c;
            else if (sel[0]) z = a;
            else             z = 0;
        end
    endmodule
```

## 晚到达的是数据信号-保持优先级

顺序if语句可以根据关键信号构造HDL。在例1.1a中，输入信号d处于选择链的最后一级，也就是说d最靠近输出。

假如信号b\_is\_late是晚到达信号，我们就要重新构造例1.1a使其最优化。

### 保持优先级的if结构

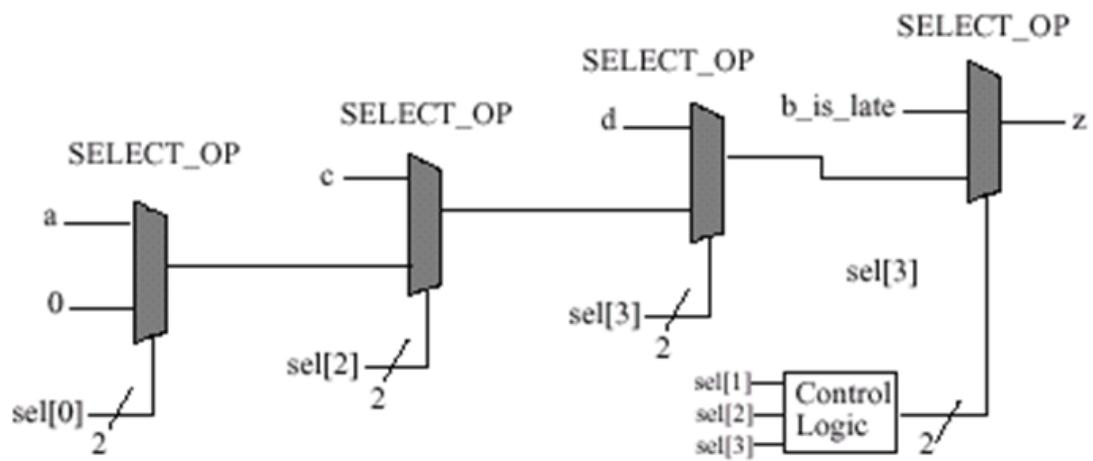
```
module mult_if_improved(a, b_is_late, c, d, sel, z);
    input a, b_is_late, c, d;
    input [3:0] sel;
    output z;
    reg z, z1;

    always @ (a or b_is_late or c or d or sel) begin
        if (sel[1] & ~ (sel[2] | sel[3])) z = b_is_late;
        else if (sel[3]) z = d;
        else if (sel[2]) z = c;
        else if (sel[0]) z = a;
        else             z = 0;
    end
endmodule
```

### 原if结构

```
module single_if(a, b, c, d, sel, z);
    input a, b, c, d;
    input [3:0] sel;
    output z;
    reg z;

    always @ (a or b or c or d or sel)
        begin
            if (sel[3])      z = d;
            else if (sel[2]) z = c;
            else if (sel[1]) z = b;
            else if (sel[0]) z = a;
            else             z = 0;
        end
    endmodule
```



## 晚到达的是控制信号

如果晚到达信号作为if语句条件分支的条件，也应使这个信号离输出最近。在下面的例子中，`CTRL_is_late`是晚到达的控制信号

```

module single_if_late(A, C, CTRL_is_late, Z);
    input [6:1] A;
    input [5:1] C;
    input CTRL_is_late;
    output Z; reg Z;

    always @(C or A or CTRL_is_late)
        if (C[1] == 1'b1) Z = A[1];
        else if (C[2] == 1'b0) Z = A[2];
        else if (C[3] == 1'b1) Z = A[3];
        else if (C[4] == 1'b1 && CTRL_is_late == 1'b0)
            // if条件中晚到达的信号
            Z = A[4];
        else if (C[5] == 1'b0) Z = A[5];
        else
            Z = A[6];

```

**endmodule**

```
module single_if_late(A, C, CTRL_is_late, Z);
    input [6:1] A;
    input [5:1] C;
    input CTRL_is_late;
    output Z; reg Z;
    always @(C or A or CTRL_is_late)
        // if条件中晚到达的信号
        if (C[4] == 1'b1 && CTRL_is_late == 1'b0)
            Z = A[4];
        else if (C[1] == 1'b1) Z = A[1];
        else if (C[2] == 1'b0) Z = A[2];
        else if (C[3] == 1'b1) Z = A[3];
        else if (C[5] == 1'b0) Z = A[5];
        else Z = A[6];
endmodule
```

## if-case嵌套语句

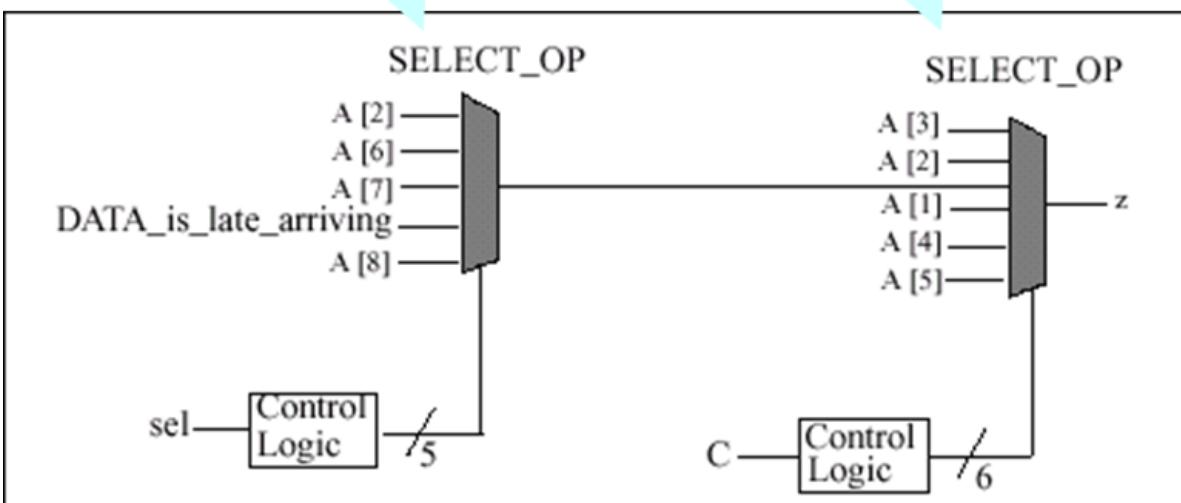
```

module case_in_if_01(A, DATA_is_late_arriving, C, sel, Z);
    input [8:1] A;
    input DATA_is_late_arriving;
    input [2:0] sel;
    input [5:1] C;
    output Z; reg Z;
    always @ (sel or C or A or DATA_is_late_arriving)
        if (C[1])           Z = A[5];
        else if (C[2] == 1'b0) Z = A[4];
        else if (C[3])           Z = A[1];
        else if (C[4])
            case (sel)
                3'b010: Z = A[8];
                3'b011: Z = DATA_is_late_arriving;
                3'b101: Z = A[7];
                3'b110: Z = A[6];
                default: Z = A[2];
        endcase
        else if (C[5] == 1'b0) Z = A[2];
        else Z = A[3];
endmodule

```

Case语句

if语句



```
always @(sel or C or A or DATA_is_late_arriving) begin
```

```
    if (C[1])          Z1 = A[5];  
    else if (C[2] == 1'b0) Z1 = A[4];  
    else if (C[3])          Z1 = A[1];  
    else if (C[4])  
        case (sel)  
            3'b010: Z1 = A[8];  
            //3'b011: Z1 = DATA_is_late_arriving;  
            3'b101: Z1 = A[7];  
            3'b110: Z1 = A[6];  
            default: Z1 = A[2];  
        endcase  
    else if (C[5] == 1'b0) Z1 = A[2];  
    else                      Z1 = A[3];
```

```
FIRST_IF = (C[1] == 1'b1) || (C[2] == 1'b0) || (C[3] == 1'b1);
```

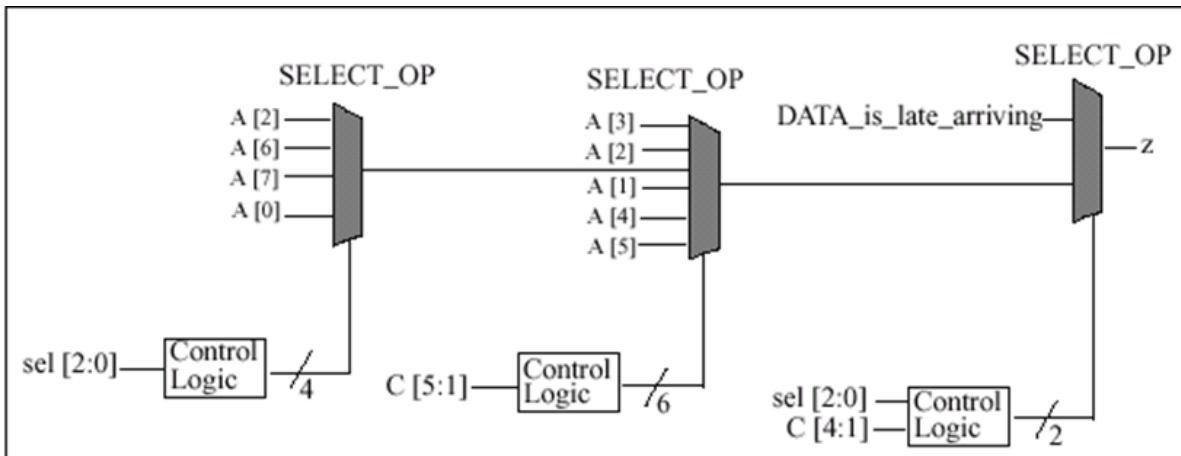
```
if (!FIRST_IF && C[4] && (sel == 3'b011))
```

```
    Z = DATA_is_late_arriving;
```

```
else
```

```
    Z = Z1;
```

```
end
```



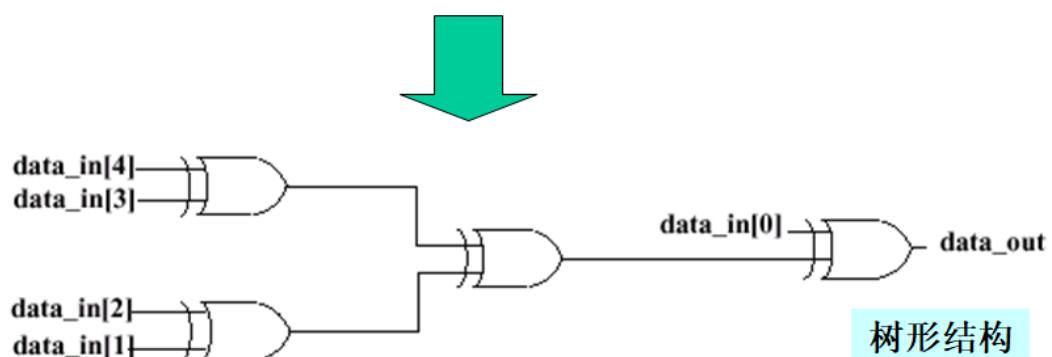
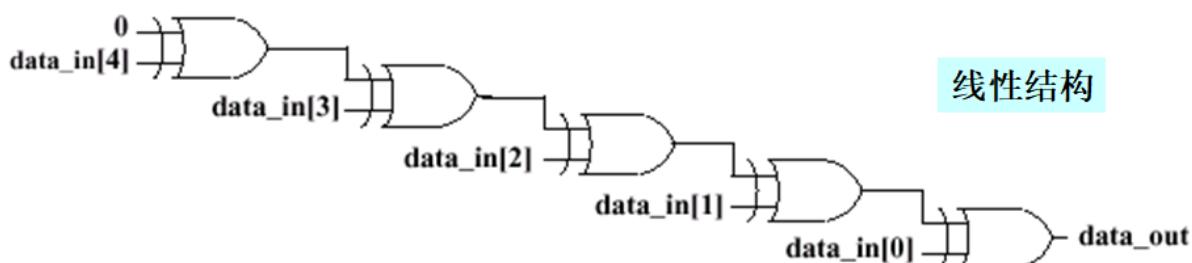
归约XOR

## 线性结构

```
module XOR_reduce (data_in, data_out);
parameter N = 5;
input [N-1:0] data_in;
output data_out;
reg data_out;

function XOR_reduce_func;
input [N-1:0] data;
integer I;
begin
    XOR_reduce_func = 0;
    for (I = N-1; I >= 0; I=I-1)
        XOR_reduce_func = XOR_reduce_func ^ data[I];
end
endfunction

always @(data_in)
    data_out <= XOR_reduce_func(data_in);
endmodule
```



树形结构实现

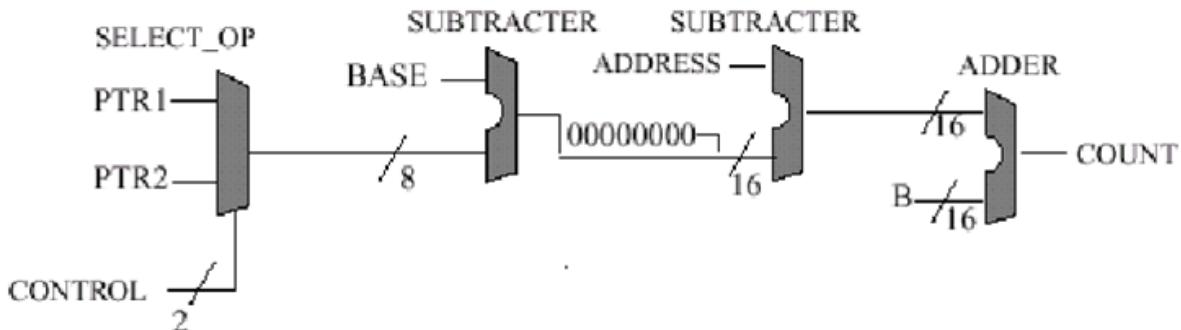
## 树形结构

```
module XOR_tree(data_in, data_out);
    parameter N = 5;
    parameter logN = 3;
    input [N-1:0] data_in;
    output data_out;  reg data_out;
    function even;
        input [31:0] num;
        even = ~num[0];
    endfunction
    function XOR_tree_func;
        input [N-1:0] data;
        integer I, J, K, NUM;
        reg [N-1:0] temp, result;
        begin
            temp[N-1:0] = data_in[N-1:0];
            NUM = N;
            for (K=logN-1; K>=0; K=K-1)
                begin
                    J = (NUM+1)/2;
                    J = J-1;

```

```
if (even(NUM))
    for (I=NUM-1; I>=0; I=I-2)
begin
    result[J] = temp[I] ^ temp[I-1];
    J = J-1;
end
else begin
    for (I=NUM-1; I>=1; I=I-2) begin
        result[J] = temp[I] ^ temp[I-1];
        J = J-1;
    end
    result[0] = temp[0];
end
temp[N-1:0] = result[N-1:0];
NUM = (NUM+1)/2;
end
XOR_tree_func = result[0];
end
endfunction
always @(data_in)
    data_out <= XOR_tree_func(data_in);
endmodule
```

## 高性能编码技术



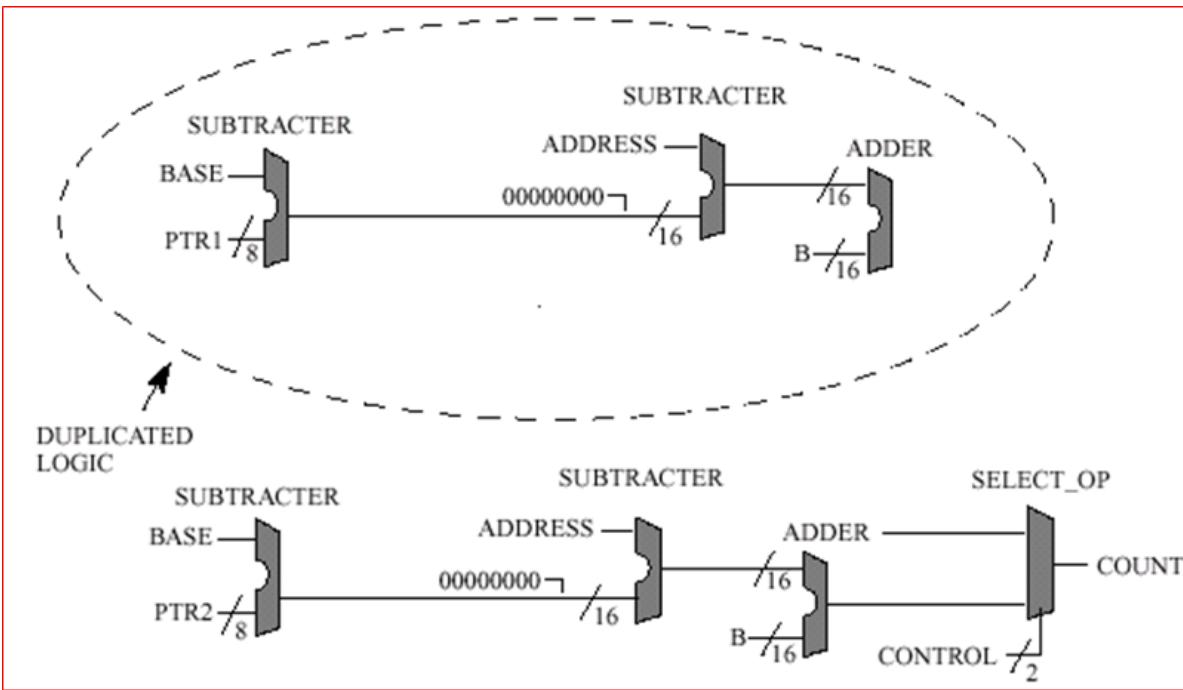
```
module BEFORE (ADDRESS, PTR1, PTR2, B, CONTROL, COUNT);
    input [7:0] PTR1, PTR2;
    input [15:0] ADDRESS, B;
    input CONTROL; // CONTROL is late arriving
    output [15:0] COUNT;
    parameter [7:0] BASE = 8'b10000000;
    wire [7:0] PTR, OFFSET;
    wire [15:0] ADDR;

    assign PTR = (CONTROL) ? PTR1 : PTR2;
    assign OFFSET = BASE - PTR;
    assign ADDR = ADDRESS - {8'h00, OFFSET};
    assign COUNT = ADDR + B;

endmodule
```

在某些情况下，可以通过重复逻辑来提高速度。

在下面的例子中，CONTROL是一个晚到达的输入信号。要提高性能，就要减少CONTROL到输出之间的逻辑。



```

module PRECOMPUTED (ADDRESS, PTR1, PTR2, B, CONTROL, COUNT);
    input [7:0] PTR1, PTR2;
    input [15:0] ADDRESS, B;
    input CONTROL;
    output [15:0] COUNT;
    parameter [7:0] BASE = 8'b10000000;
    wire [7:0] OFFSET1,OFFSET2;
    wire [15:0] ADDR1,ADDR2,COUNT1,COUNT2;

    assign OFFSET1 = BASE - PTR1; // Could be f(BASE,PTR)
    assign OFFSET2 = BASE - PTR2; // Could be f(BASE,PTR)
    assign ADDR1 = ADDRESS - {8'h00 , OFFSET1};
    assign ADDR2 = ADDRESS - {8'h00 , OFFSET2};
    assign COUNT1 = ADDR1 + B;
    assign COUNT2 = ADDR2 + B;
    assign COUNT = (CONTROL == 1'b1) ? COUNT1 : COUNT2;

endmodule

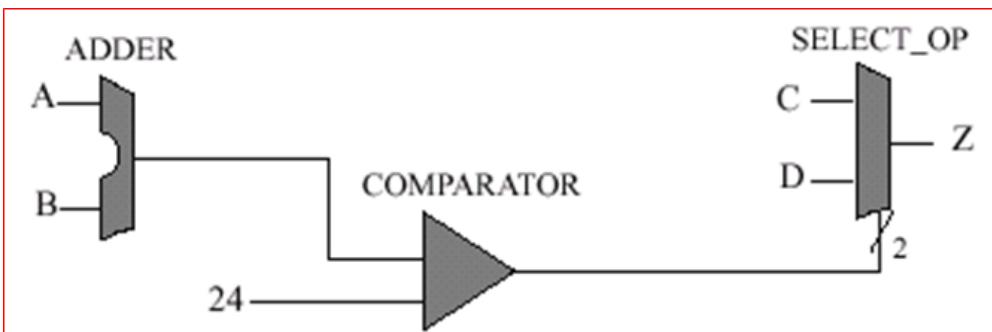
```

在下面的例子中，if语句的条件表达中包含有操作符。

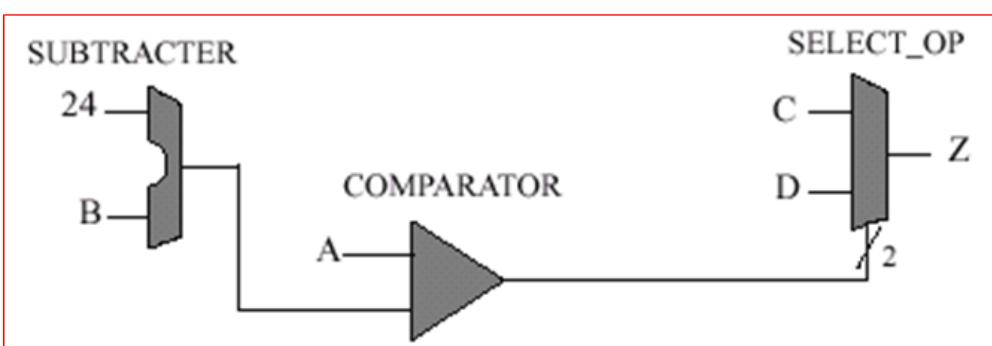
```

module cond_oper(A, B, C, D, Z);
    parameter N = 8;
    input [N-1:0] A, B, C, D; //A is late arriving
    output [N-1:0] Z;
    reg [N-1:0] Z;
    always @(A or B or C or D)
    begin
        if (A + B < 24)
            Z = C;
        else
            Z = D;
    end
endmodule

```



若条件表达式中的信号A是晚到达信号。因此要移动信号A使其离输出近一些。



```

module cond_oper_improved (A, B, C, D, Z);

parameter N = 8;

input [N-1:0] A, B, C, D; // A is late arriving

output [N-1:0] Z;

reg [N-1:0] Z;

always @ (A or B or C or D)

begin

    if (A < 24 - B)

        Z <= C;

    else

        Z <= D;

end

endmodule

```

## 其它要注意的问题

- 不要引入不必要的latch
- 敏感表要完整
- 非结构化的for循环
- 资源共享

## 不要产生不需要的latch

条件分支不完全的条件语句（if和case语句）将会产生锁存器

```

always @ (cond_1, data_in)
begin
  if (cond_1)
    data_out = data_in;
end

```

```

always @ (sel or a or b or c or d)
begin
  case (sel)
    2'b00: a = b;
    2'b01: a = c;
    2'b10: a = d;
  endcase
end

```

## 敏感表要完整

不完整的的敏感表将引起综合后网表的仿真结果与以前的不一致。

```

always @ (d or clr)
if (clr)
  q = 1'b0
else if (e)
  q = d;

```

```

always @ (d, clr, e)
if (clr)
  q = 1'b0
else if (e)
  q = d;

```

```

always @ (*)
if (clr)
  q = 1'b0
else if (e)
  q = d;

```

## 资源共享

资源共享是指多节代码共享一组逻辑。例如：

```
always @(*( a or b or c or d))
```

```
if (a)
```

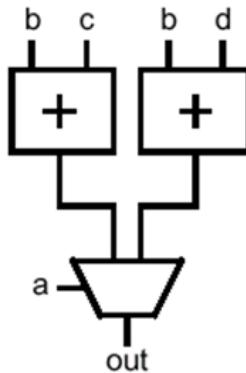
```
    out = b + c;
```

```
else
```

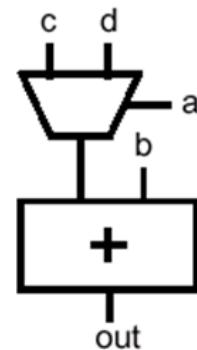
```
    out = b + d;
```

资源共享与所用综合工具有关。但通常，要共享资源，表达式必须在同一个**always**块中的同一个条件语句中。

没有资源共享



资源共享



资源共享可以由RTL代码控制。

例如，可以改变编码风格强制资源共享。

原始代码

```
if (a)
    out = b + c;
else
    out = b + d;
```

强制资源共享

```
temp = a ? c : d;
out = b + temp;
或
out = b + (a ? c : d);
```

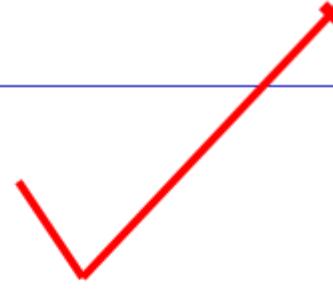
只有在同一个条件语句(if和case)不同的分支中的算术操作才会共享。

条件操作符 ?: 中的算术操作不共享。

```

if (cond)
    z = a + b;
else
    z = c + d;

```

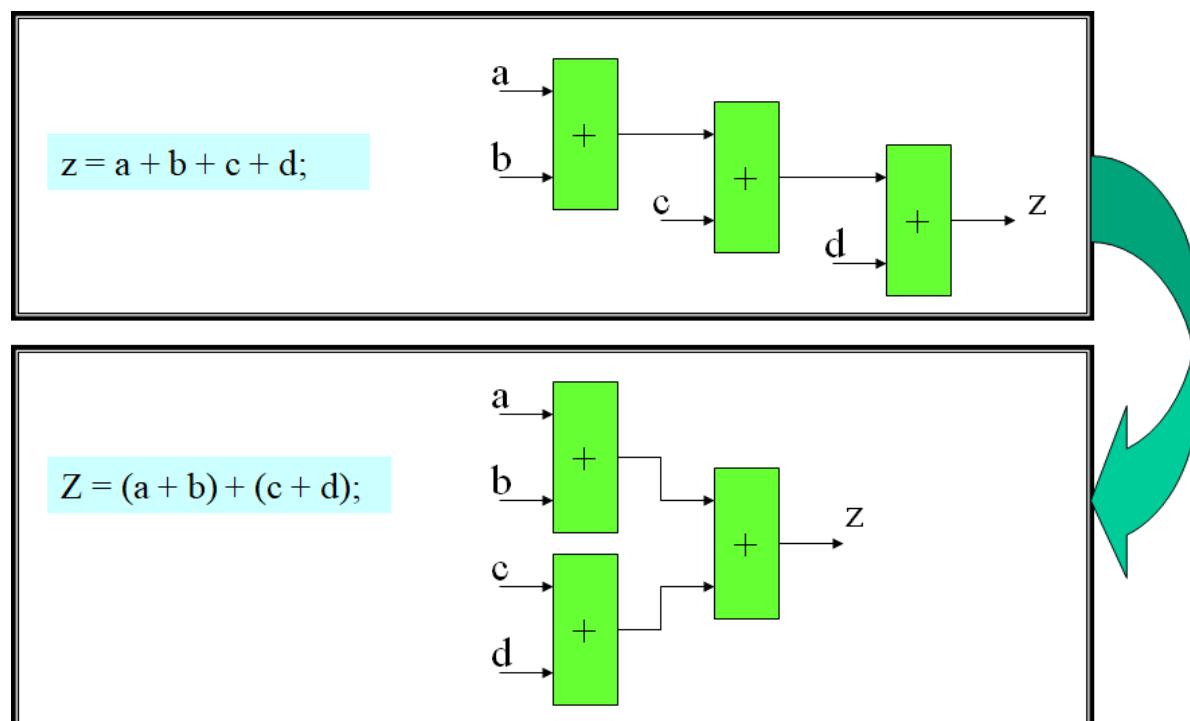


$Z = (\text{cond}) ? (a + b) : (c + d);$



## 括号的作用

利用括号分割逻辑



## 逻辑构造块的编码格式

下面介绍某些常用逻辑块，如译码器的不同的编码格式。每种块给出了一个通常格式和建议格式。  
所有的例子的位宽都是参数化的。

## 3-8译码器

### index方式

```
module decoder_index (in1, out1);
    parameter N = 8;
    parameter log2N = 3;
    input [log2N-1:0] in1;
    output [N-1:0] out1;
    reg [N-1:0] out1;

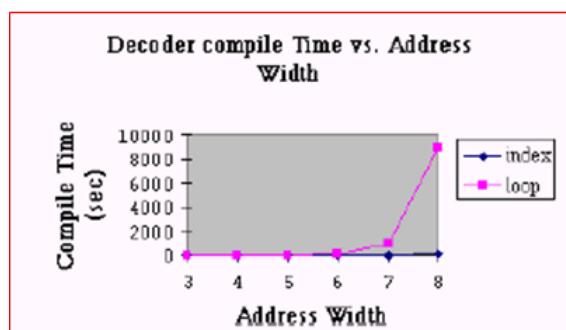
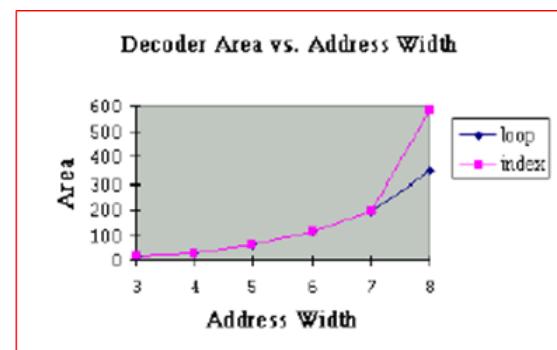
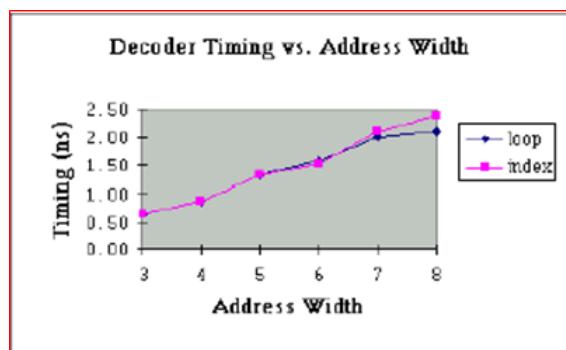
    always @(in1)
        begin
            out1 = 0;
            out1[in1] = 1'b1;
        end
    endmodule
```

### loop方式

```
module decoder38_loop (in1, out1);
    parameter N = 8;
    parameter log2N = 3;
    input [log2N-1:0] in1;
    output [N-1:0] out1;
    reg [N-1:0] out1;
    integer i;

    always @(in1) begin
        for(i=0;i<N;i=i+1)
            out1[i] = (in1 == i);
    end
endmodule
```

## 译码器



## 优先级编码器—高位优先

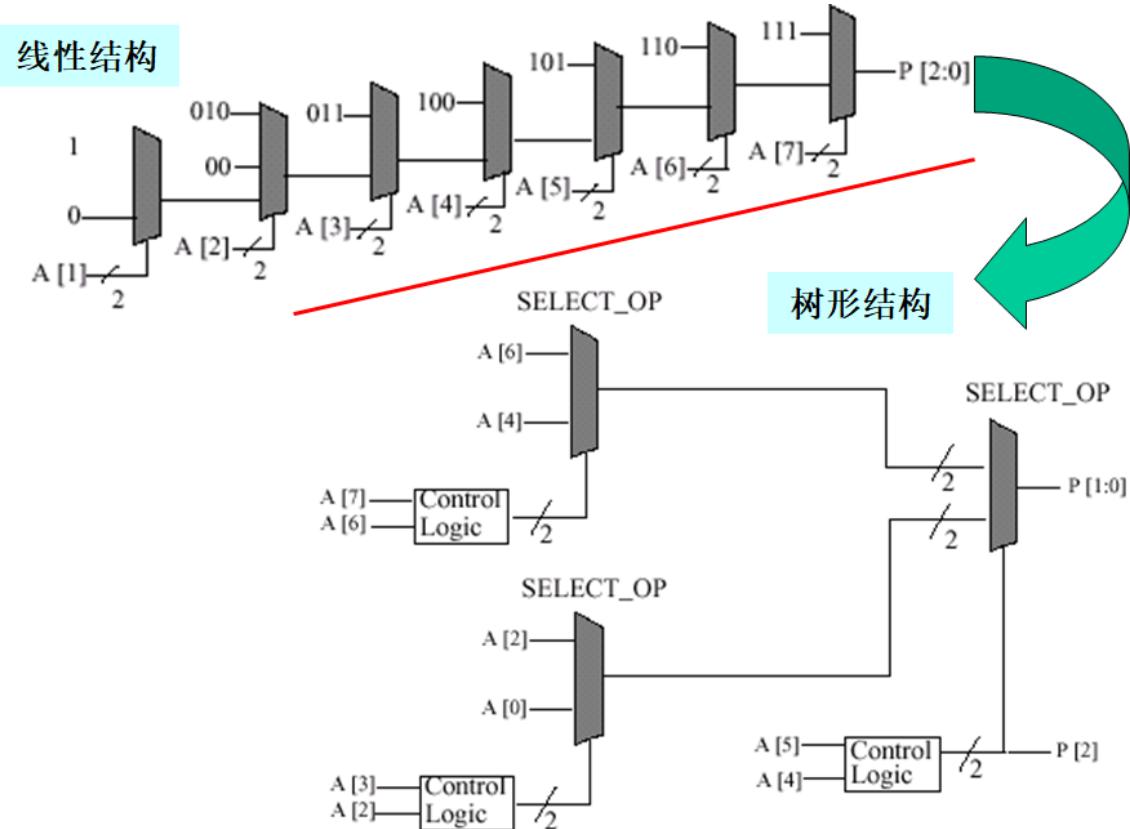
## 线性结构描述

```
1???_????: 111
01??_????: 110
001?_????: 101
0001_????: 100
0000_1??? : 011
0000_01?? : 010
0000_001? : 001
0000_000? : 000
```



```
module priority_low_high (A, P);
parameter N = 8;
parameter log2N = 3;
input [N-1:0] A; //Input Vector
output [log2N-1:0] P; // High Priority Index
reg [log2N-1:0] P;
function [log2N-1:0] priority;
    input [N-1:0] A;
    integer I;
    begin
        priority = 3'b0;
        for (I=0; I<N; I=I+1)
            if (A[I])
                priority = I; // 覆盖前面的值
    end
endfunction
always @(A)
    P = priority(A);
endmodule
```

## 优先级编码器



## 显式有限状态机

```

`timescale 1ns/100ps
module state4 (clock, reset, out);
    input reset, clock;
    output [1: 0] out;
    reg [1: 0] out;
    parameter //状态变量枚举
        stateA = 2'b00,
        stateB = 2'b01,
        stateC = 2'b10,
        stateD = 2'b11;
    reg [1: 0] state; //状态寄存器
    reg [1: 0] nextstate;
    always @(posedge clock)
        if (reset) //同步复位
            state <= stateA;
        else
            state <= nextstate;

```

```

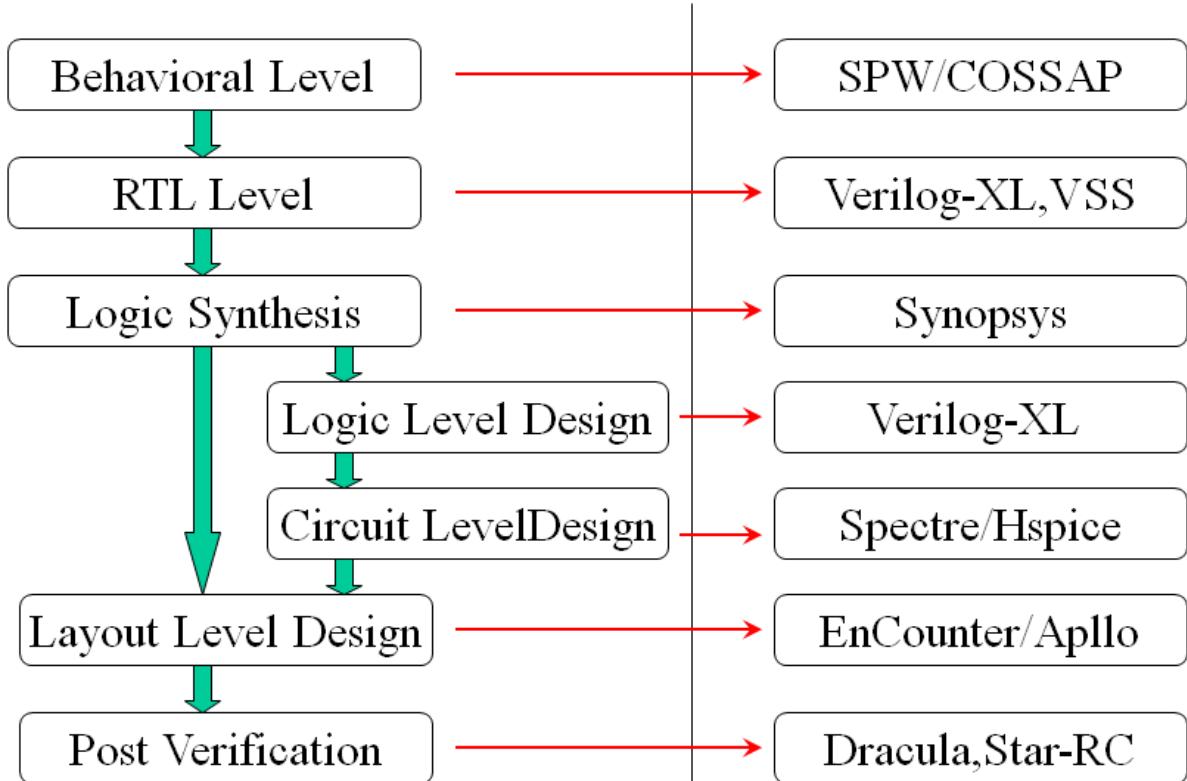
always @(*( state)) // 定义下一状态的组合逻辑
case (state)
    stateA: begin
        nextstate = stateB;
        out = 2'b00; // 输出决定于当前状态
    end
    stateB: begin
        nextstate = stateC;
        out = 2'b11;
    end
    stateC: begin
        nextstate = stateD;
        out = 2'b10;
    end
    stateD: begin
        nextstate = stateA;
        out = 2'b00;
    end
endcase
endmodule

```

## 逻辑综合

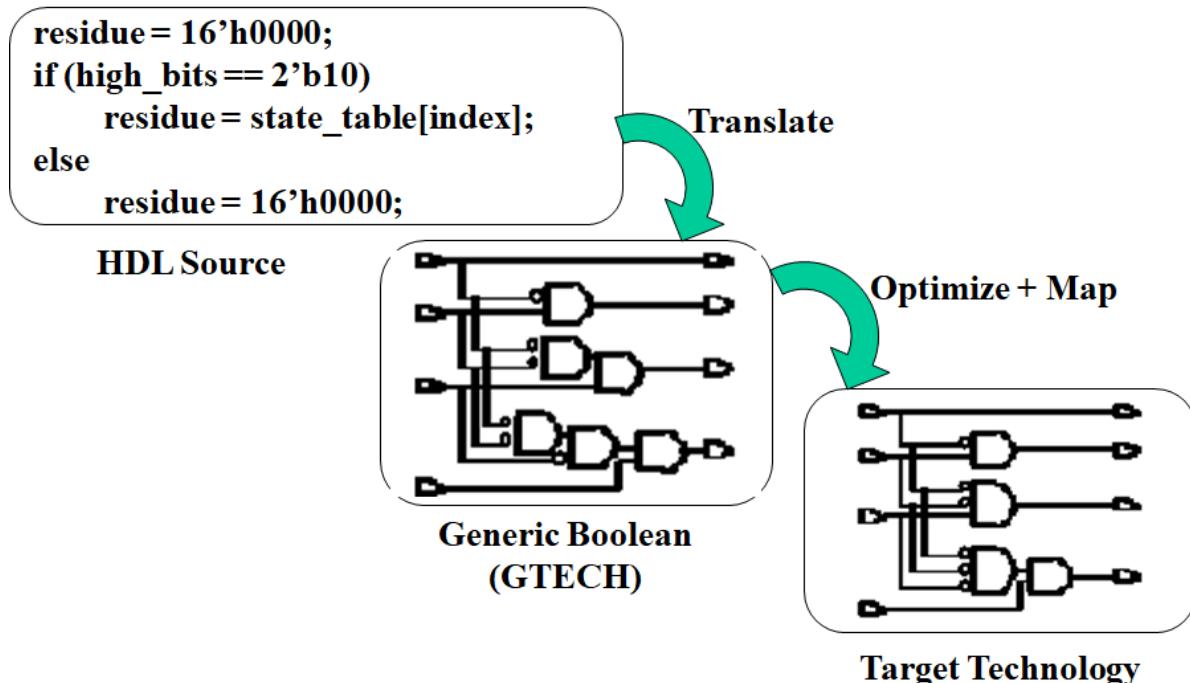
### 逻辑综合介绍

#### 基于标准单元的设计流程



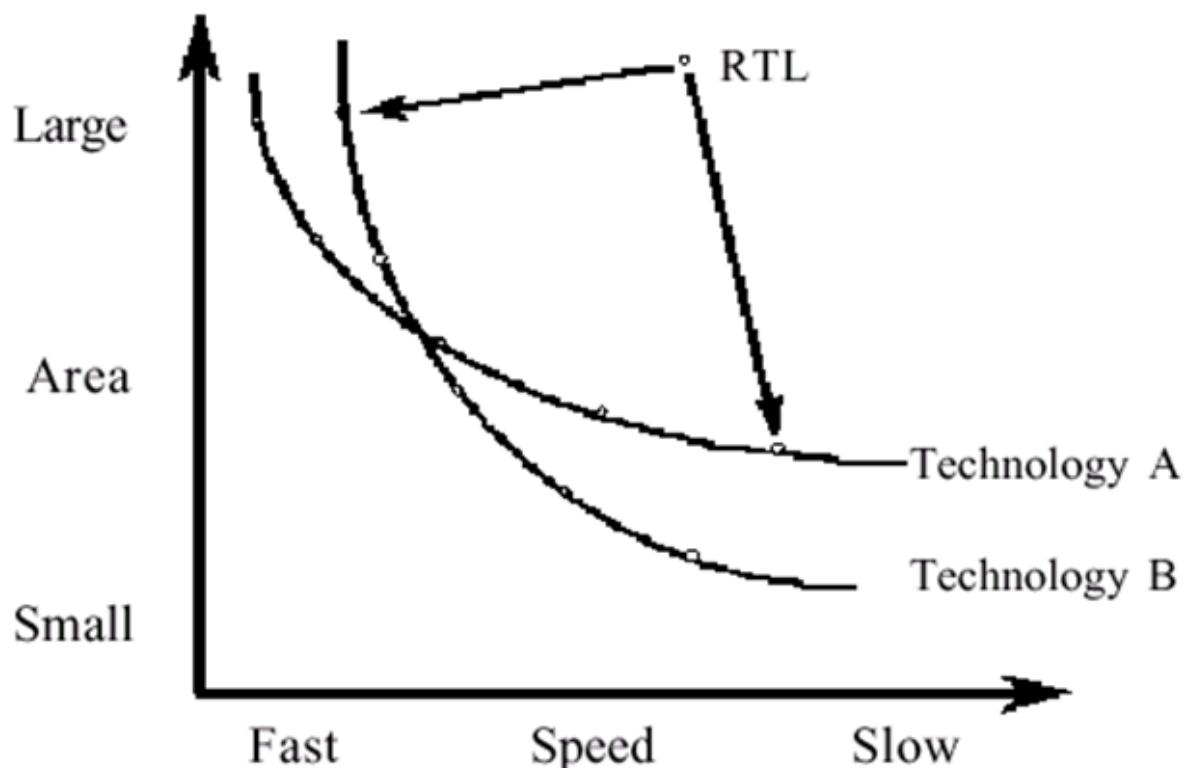
## 什么是综合?

Synthesis = translation + optimization + mapping

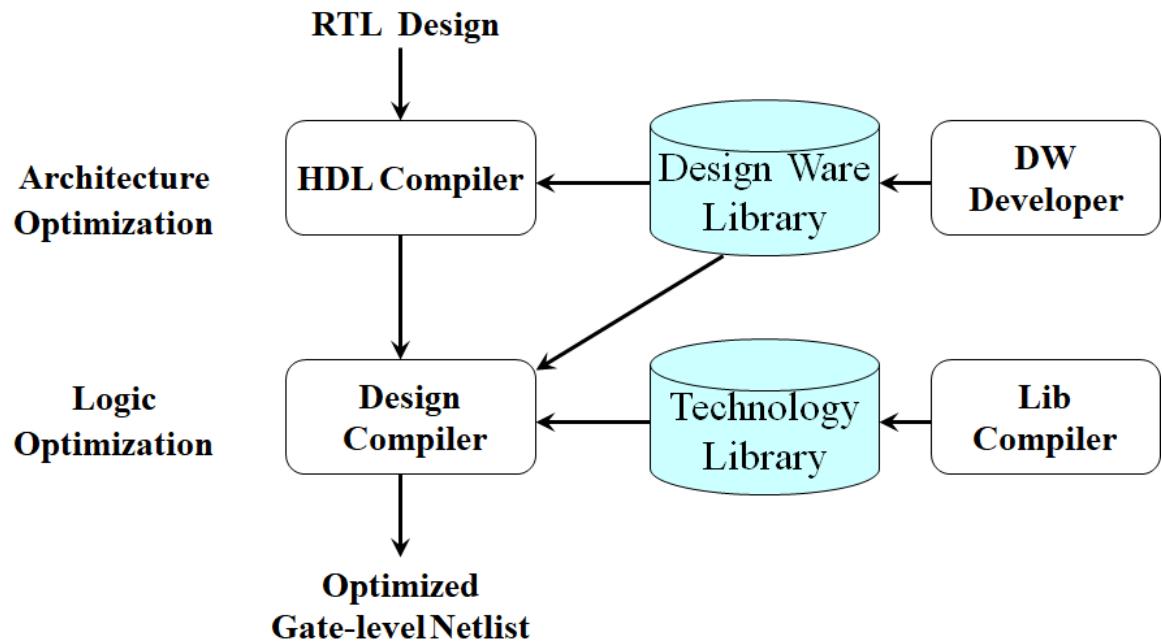


## 工艺无关性

设计可以转换到任何工艺上。

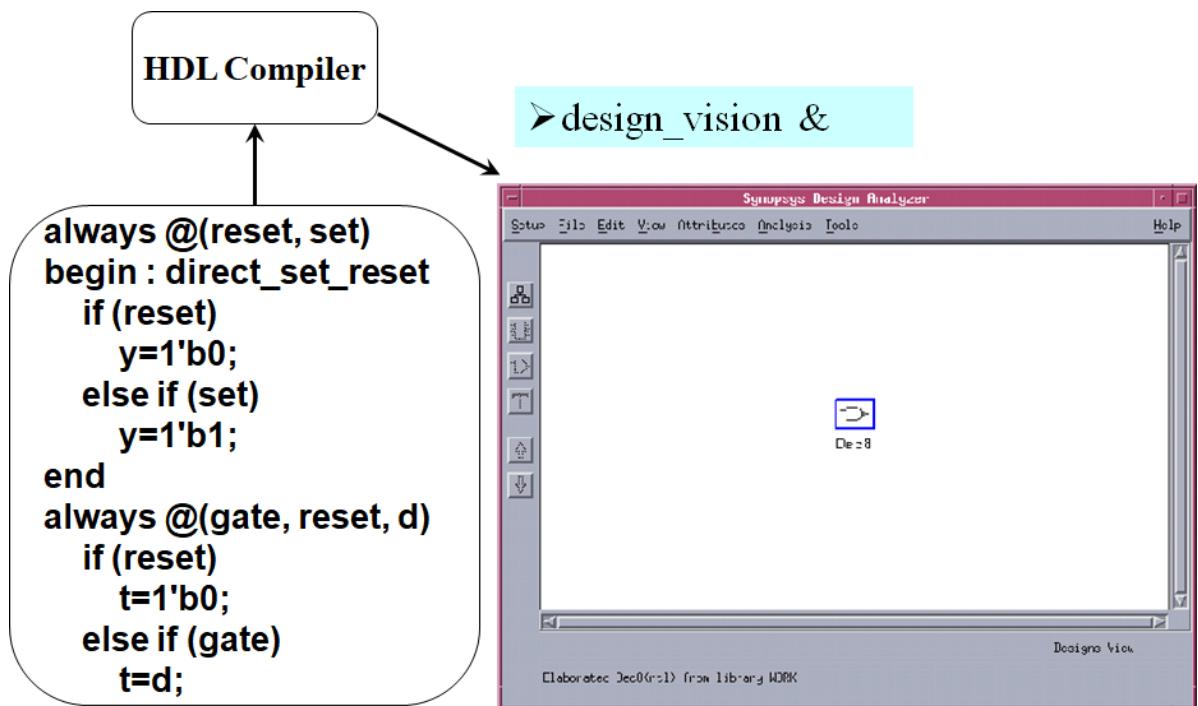


## Logic Synthesis Overview

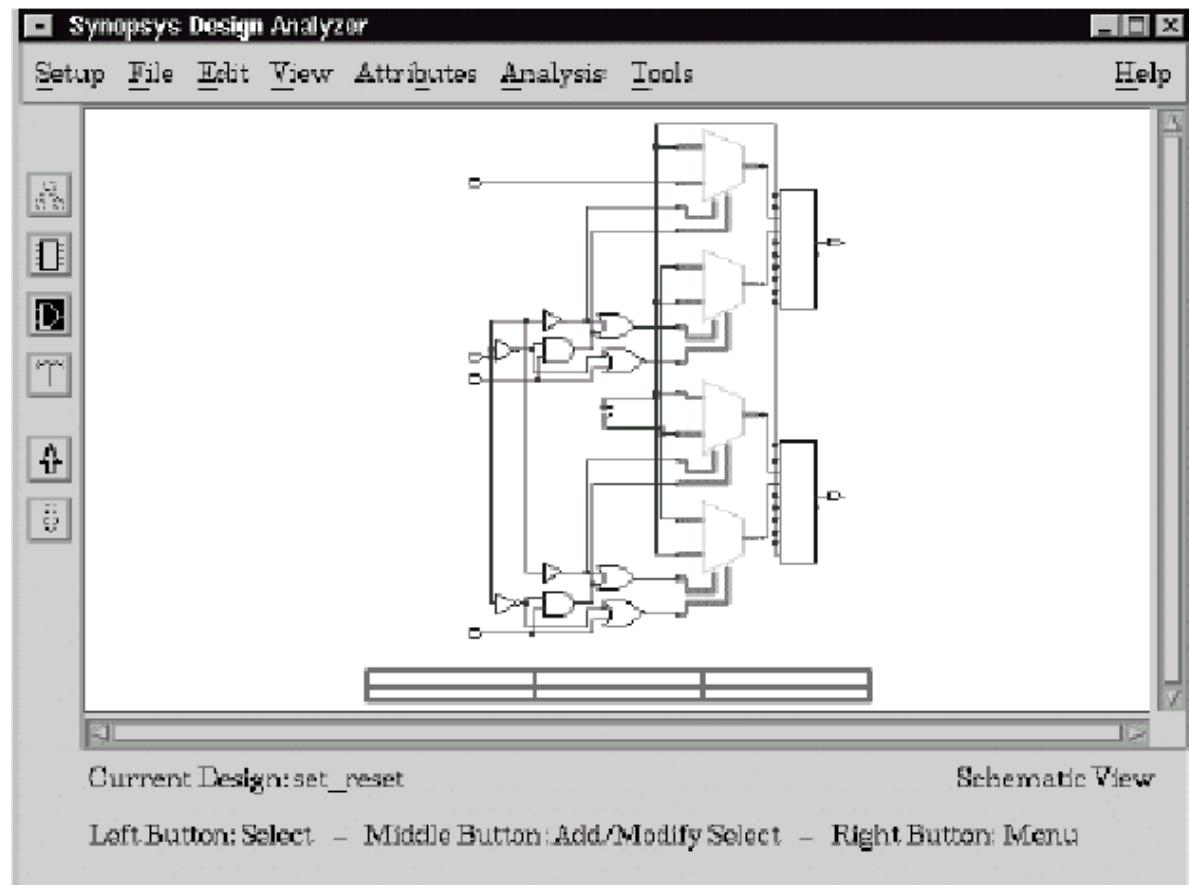


## HDL Compiler

HDL Compiler将HDL描述转换为Synopsys设计块，并传送给Design Compiler

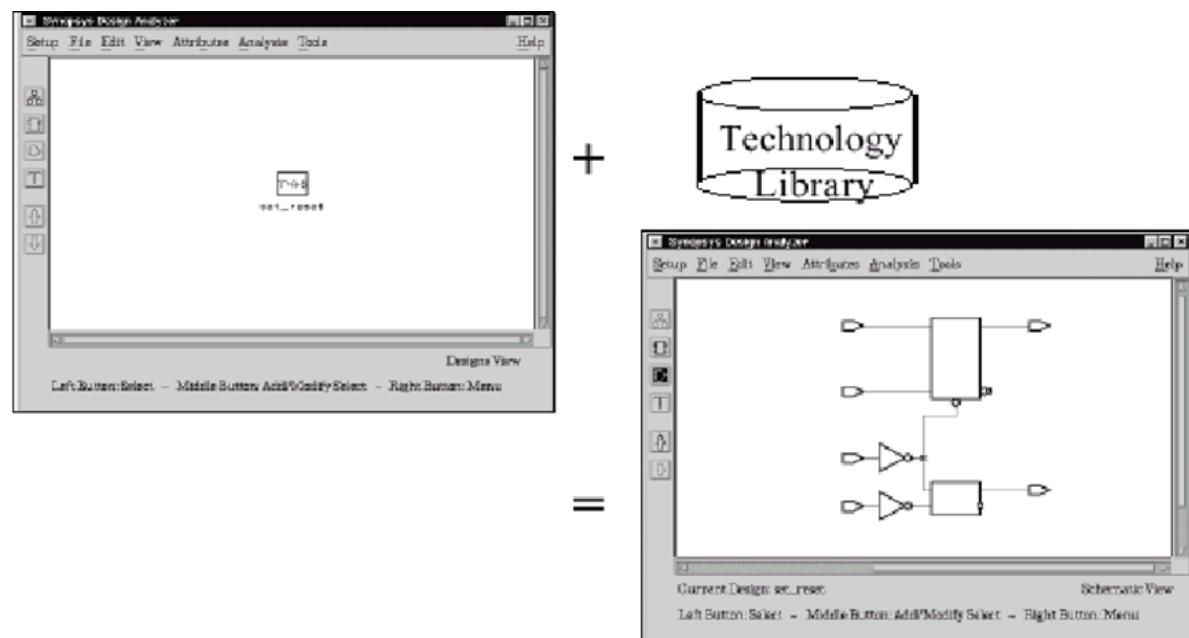


在逻辑图中，我们可以看到，verilog文件已经转换为 GTECH 库(the synopsys default)



## Design Compiler

Design Compiler将Synopsys设计块映射到用户指定库的门级设计



## Synopsys相关文件

文 件	用 途
.cshrc	设置path、环境变量以及license文件
.synopsys_dc.setup	<p>设置Design Compiler的相关变量</p> <p>1、系统级（不要修改） \$SYNOPSYS/admin/setup/.synopsys_dc.setup</p> <p>2、用户登陆(home)目录</p> <p>3、用户当前工作目录</p>

注意：

- 1.这三个文件总是按所列顺序读取
- 2.对于相同的设置项，后面的设置覆盖前面的设置

### .synopsys\_dc.setup中定义的内容

- link\_library：这个库用来解释所输入的设计描述
  - 在用户HDL代码中的门级网表或实例化的单元
- target\_library：所要映射到的ASIC工艺库
  - 在综合过程中使用的wire load或Operating condition模型
- symbol\_library：用来生成逻辑图
- search\_path：定义所引用的库或设计的查找路径
- synthetic\_library：要用到的designware库
- 其它变量

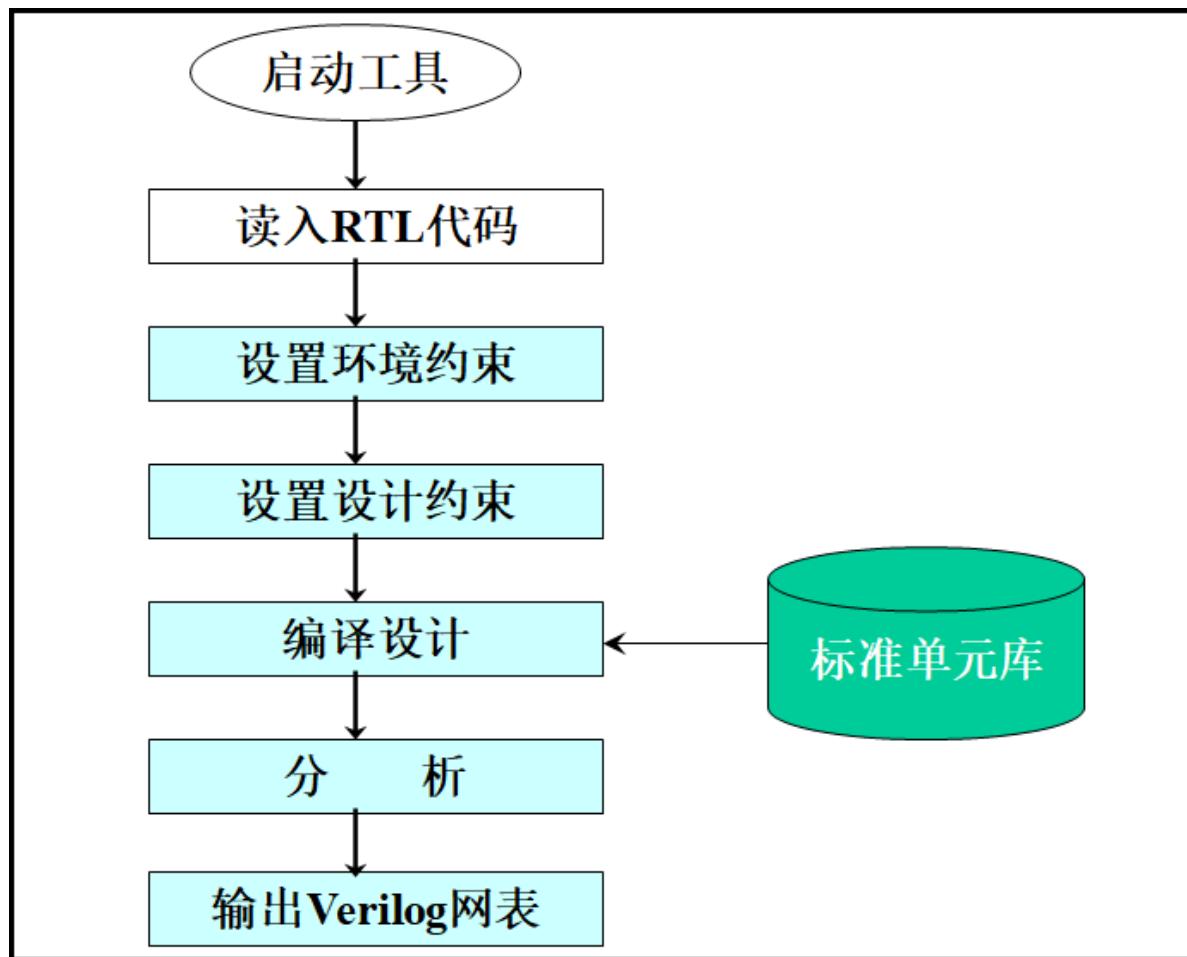
```

search_path = {. /my_library} + search_path;
link_library = {"*", "my_link.db", "dw_foundation.sldb"} ;
target_library = {my_target.db};
symbol_library = {generic.sdb} ;
synthetic_library = {"dw_foundation.sldb"} ;

hdlin_translate_off_skip_text = "TRUE"
edifout_netlist_only = "TRUE"
verilogout_no_tri = true ;
plot_command = "lpr -Plp" ;
view_script_submenu_items =
{"Avoid assign statement", "set_fix_multiple_port_nets -all -buffer_constant", \
"Change Naming Rule", "change_names -rule verilog -hierarchy", \
"Write SDF", "write_sdf -version 1.0 -context verilog chip.sdf"}

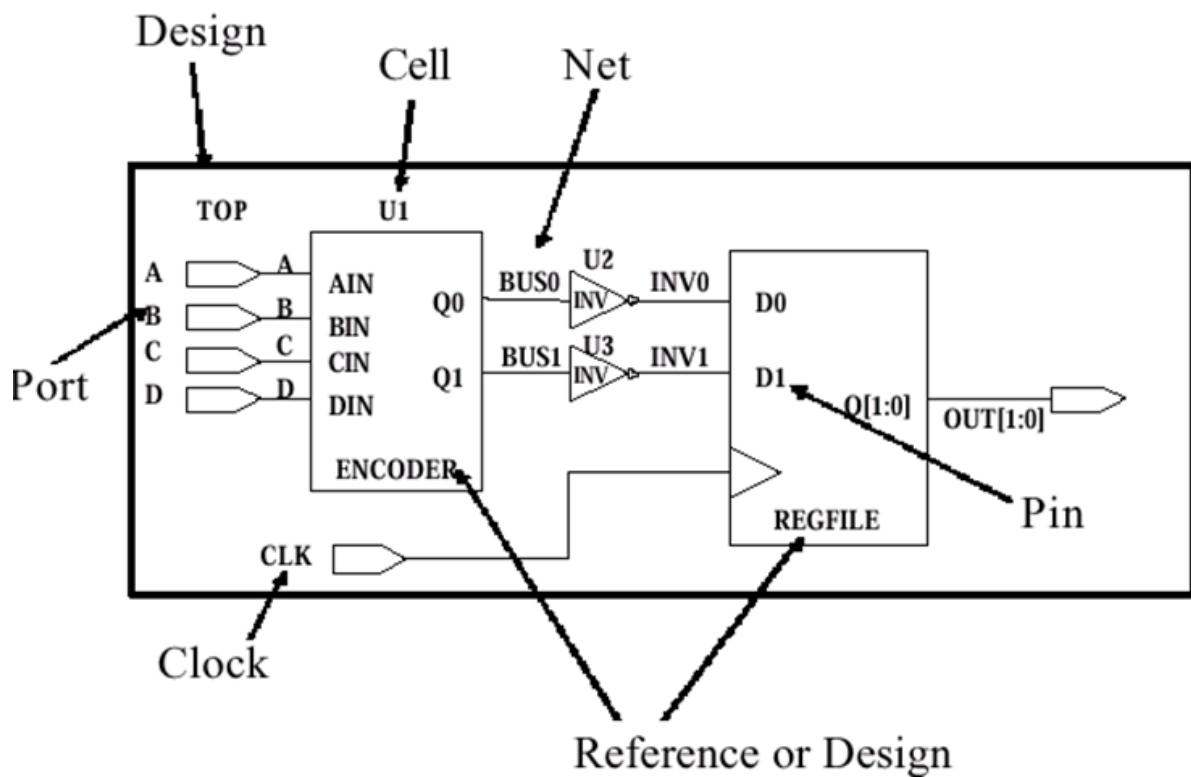
```

## ASIC综合的设计流程



## 综合对象

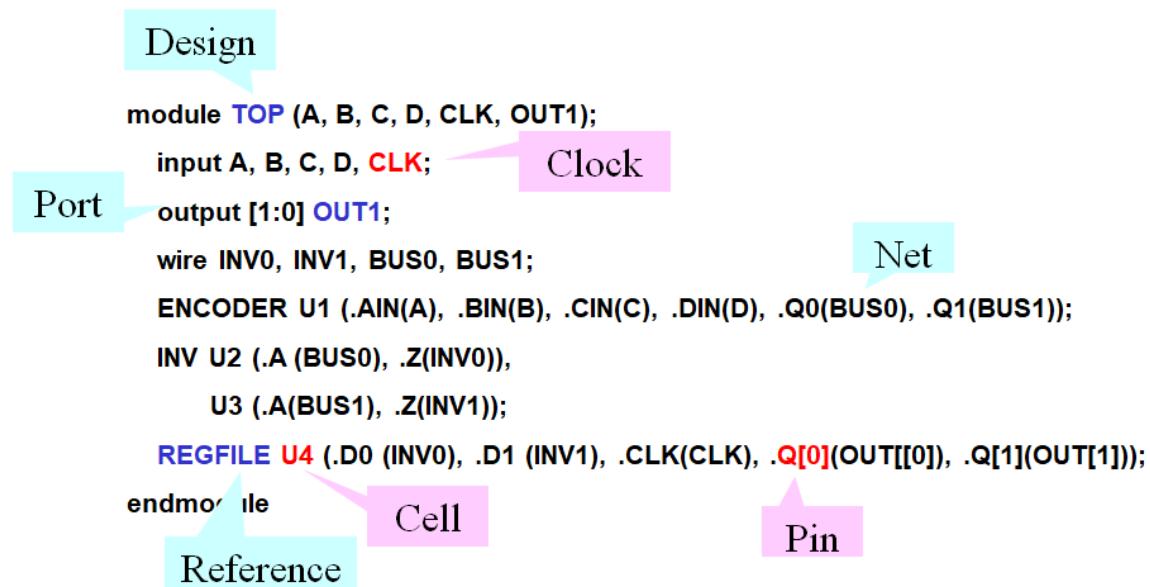
### 设计对象 (逻辑图示)



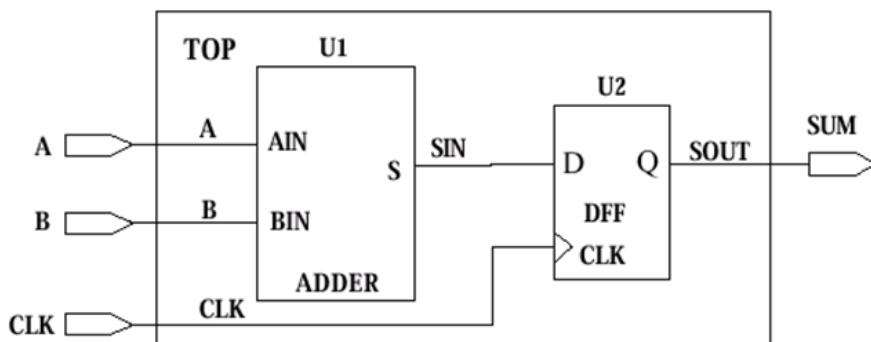
- Design(设计)：完成一个或多个逻辑功能 的电路
- Cell (单元)：一个设计在另一个设计中的实例
- Reference (引用)：单元(Cell)指向的原始设计
- Port (端口)：设计的输入或输出

- Pin (引脚) : 单元(Cell)的输入或输出
- Net (网线) : Port-Pin或Pin-Pin之间的连接线
- Clock (时钟) : 指定为时钟源的Port或Pin上所加的波形

## 设计对象 (Verilog图示)



## 练习

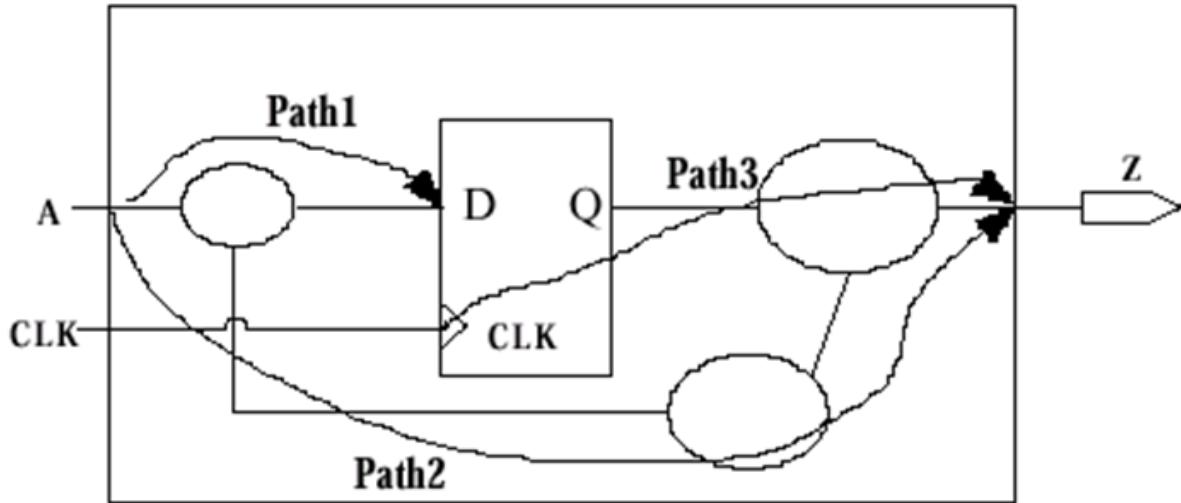


1. 列出设计中所有的端口 (Port)? { **A B SUM** }
2. 列出名字中含有字母 “U”的所有单元(Cell)? { **U1 U2** }
3. 列出以CLK结尾的所有网线(net)? { **CLK** }
4. 列出设计中所有的 “Q”引脚 (Pin)? { **Q** }
5. 列出所有的引用(Reference)? { **ADDER DFF** }

## 静态时序分析

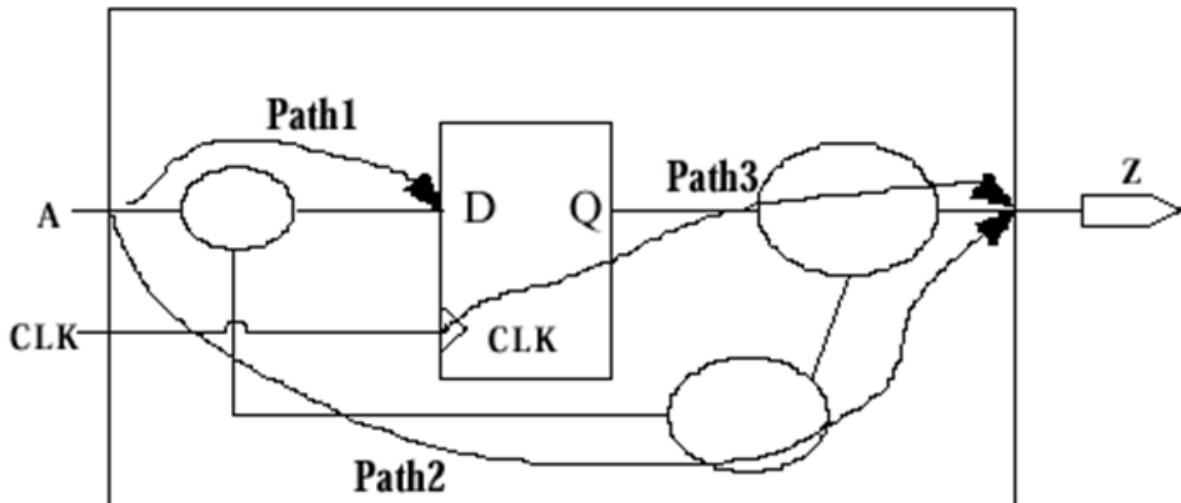
### 静态时序分析(Design Time)

- 分析电路是否符合时序约束( timing constraint), 不需要仿真
  - 将设计划分为一系列时序路径( timing path )
  - 计算每一个路径的延时
  - 检查所有的路径延时是否符合时序约束



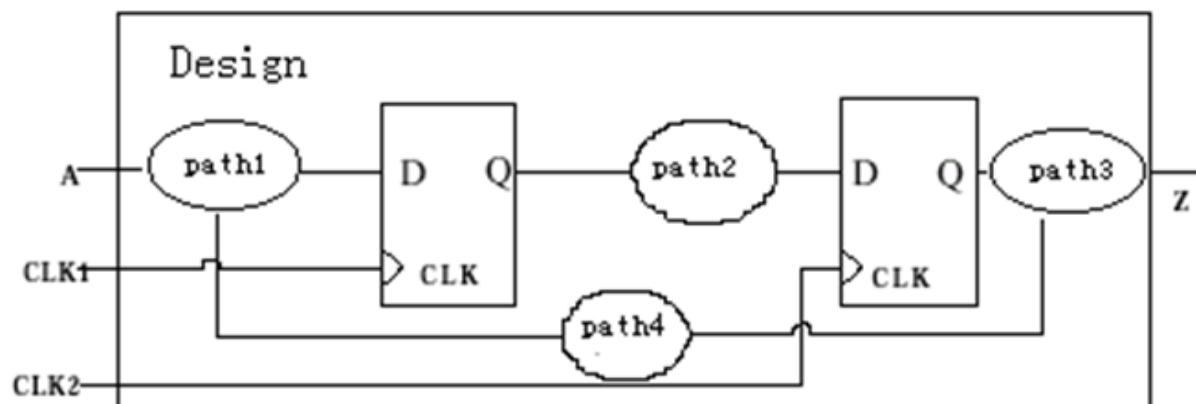
## Design Compiler中的时序路径

- Design Time将设计划分为一系列的信号路径，每一路都有起点和终点
  - 起点：输入端口(Port)和时序器件的数据输入引脚(pin)
  - 终点：输出端口(port)和时序器件的数据输出引脚(pin)



## 时序路径组(group)

如何将时序路径构成一组？



时序路径按控制终点的时钟划分成组

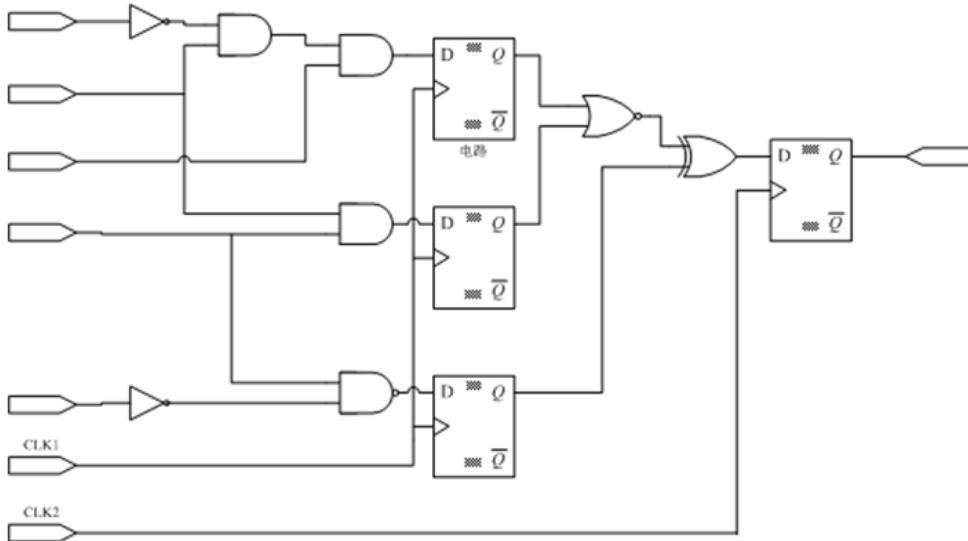
与每一个时钟相关联的一组路径构成一个组

缺省情况下，其它与时钟无关的所有路径构成一个组



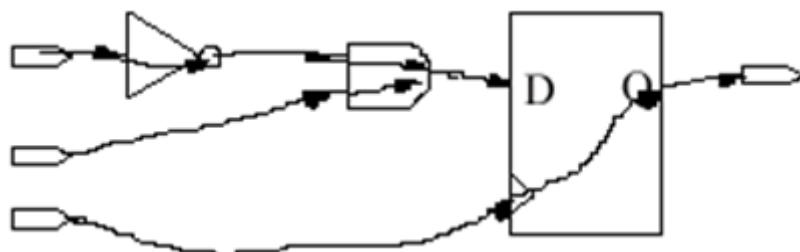
练习

- 下图有多少个时序路径? **11**
- 有多少个路径组? **3**

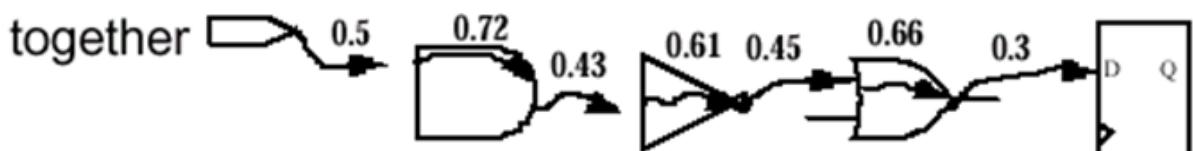


## 逻辑图转换为时序图

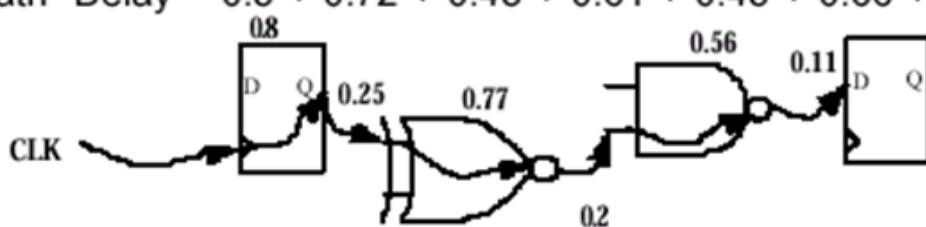
- 为了计算总的延时, Design Time将每一条路径划分为时序弧(timing arc)。
- 时序弧: 一段连接线(net)延时或一个单元(cell)延时



- 计算路径延时举例
  - 将所有延着路径的网线和单元的时序弧加起来。



$$\text{Path Delay} = 0.5 + 0.72 + 0.43 + 0.61 + 0.45 + 0.66 + 0.3 = 3.67 \text{ ns}$$



$$\text{Path Delay} = 0.8 + 0.25 + 0.77 + 0.2 + 0.56 + 0.11 = 2.51 \text{ ns}$$

# IP Library

## DesignWare Library

- DesignWare Library是一组可重用的、可综合的IP块，集成在Synopsys综合环境中。
- 缺省的DesignWare库是standard.sldb，包括：
  - adder: +, +1
  - subtractor: -, -1
  - comparator: ==, !=, <, <=, >, >=
  - mulpilier, divider
  - sin cos tan
  - Sqrt FIR IIR
- 如果用户在设计中使用了DesignWare的元件，用户可以使用约束来改变其实现方式
- 实现方式查看DW手册

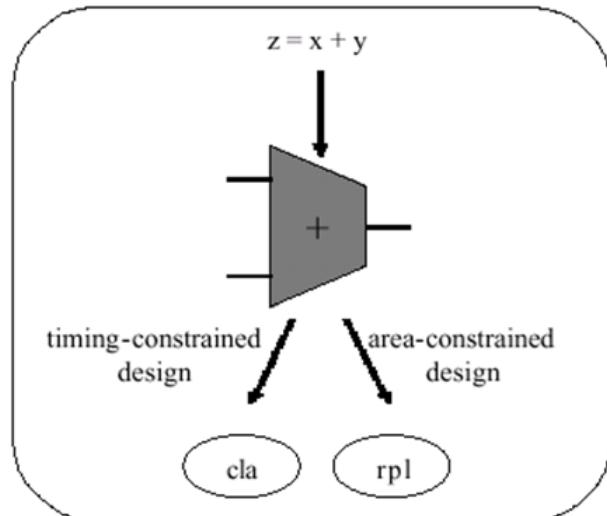


Table 4 - Synthesis Implementations

Implementation Name	Function	License Required
rpl	Ripple carry synthesis model	DesignWare-Basic
cla	Carry look-ahead synthesis model	DesignWare-Basic
clf	Fast carry look-ahead synthesis model	DesignWare-Foundation
bk <sup>1</sup>	Brent-Kung architecture synthesis model	DesignWare-Foundation
csm <sup>2</sup>	Conditional sum synthesis model	DesignWare-Foundation

- 如果用户要使用DesignWare库，必须在.synopsys\_dc.setup中设置“synthetic\_library”和“search\_path”
- Example: `synthetic_library = {"dw_foundation.sldb"}`
- 如果模块在不同的库中都有并且有相同的名字，则使用所列的第一个库中的模块。

## DesignWare Part

- 有两种方法使用DesignWare中的元件：
  - 推断：由design compiler根据约束选择DesignWare元件
  - 实例化：显式的实例化synthetic模块
- Example

推断

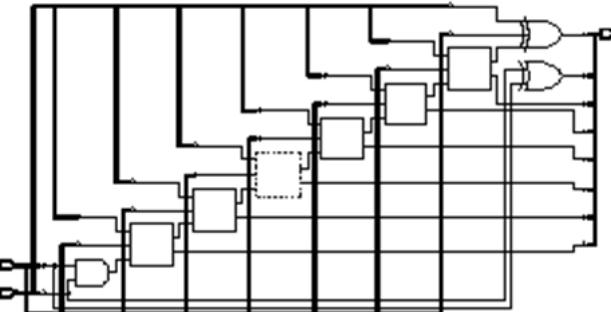
```
module adder(in1,in2,sum);
  parameter wordlength = 8;
  input [wordlength-1:0] in1,in2
  output [wordlength-1:0] sum;
  assign sum = in1 + in2;
endmodule
```

实例化

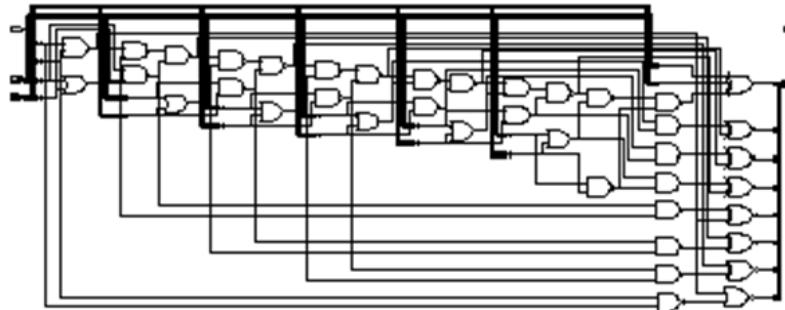
```
module adder(in1, in2, carry_in, sum, carry_out);
  parameter wordlength = 8;
  input [wordlength-1:0] in1, in2;
  input carry_in;
  output [wordlength-1:0] sum;
  output carry_out;
  DW01_add #(wordlength) U1(in1,in2,carry_in,sum,carry_out);
endmodule
```

- Example : assign c = a + b;

面积优化  
8-bit行波进位加法器  
(rpl)



速度优化  
8-bit cla加法器



## DW-Select implementation

- 在使用DesignWare库中的元件时, 我们可以选择实现方式。例如, 当使用元件 “dw01\_add”时, 可以明确指定这个加法器是一个cla-adder 或是一个 rpl -adder。
- 如何指定?
  - 在RTL 代码中指定 (嵌入式)
  - 使用dc\_shell命令“set\_implementation”
- 与DesignWare相关的所有信息包含在Synopsys联机文档(SOLD)中的 “DesignWare”部分。

Synthesis Implementation

Component	Standard Library Implementations	Operator/Function	Foundation Library Implementation
DW01_absval	rpl, cla	DW_absval	clf
DW01_add	rpl, cla	+	clf, bk, csm
DW01_addsub	rpl, cla	+,-	clf, bk, csm
DW01_cmp2	rpl	<, >	clf, bk
DW01_cmp6	rpl	<, >, <=, >=	clf, bk
DW01_dec	rpl, cla	-	clf
DW01_inc	rpl, cla	+	clf
DW01_incdec	rpl, cla	+,-	clf
DW01_sub	rpl, cla	-	clf, bk, csm
DW02_mult	csa	X	wall

## Implementation – 嵌入式

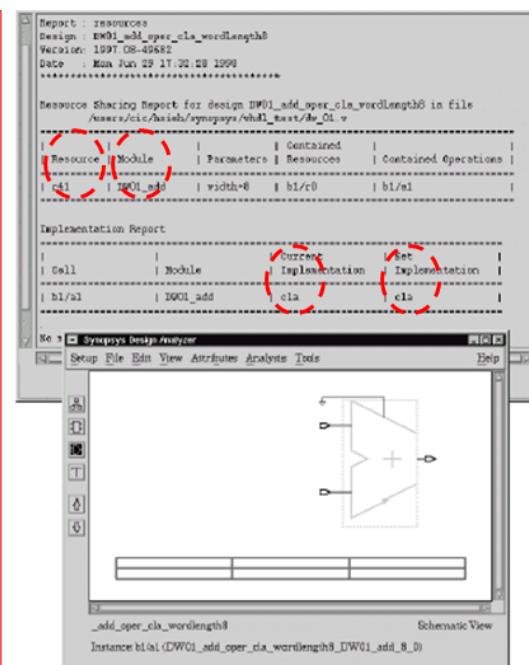
- 选择需要的实现
- 代码格式 – 只能在**always**块中使用
  - /\* synopsys resource *resource\_name* :
  - 选择元件  
*map\_to\_module* = “*module\_name*”,
  - 选择实现的方式  
*implementation* = “*impl\_name*”,
  - 绑定标记的操作到指定的模型及实现  
*ops* = “*label\_name*”;\*/
  - 标记操作  
***z = a + b; // synopsys label *label\_name****

- Example – 加法器，采用元件“dw01\_add”，实现方式“cla”

```

module DW01_add_oper_cla(in1,in2,sum);
    parameter wordlength = 8;
    input [wordlength-1:0] in1,in2;
    output [wordlength-1:0] sum;
    reg [wordlength-1:0] sum;
    always @ (in1 or in2) begin :b1
        /* synopsys resource r0:
            map_to_module= "DW01_add",
            implementation = "cla ",
            ops = "a1"; */
        sum = in1 + in2; //synopsys label a1
    end
endmodule

```

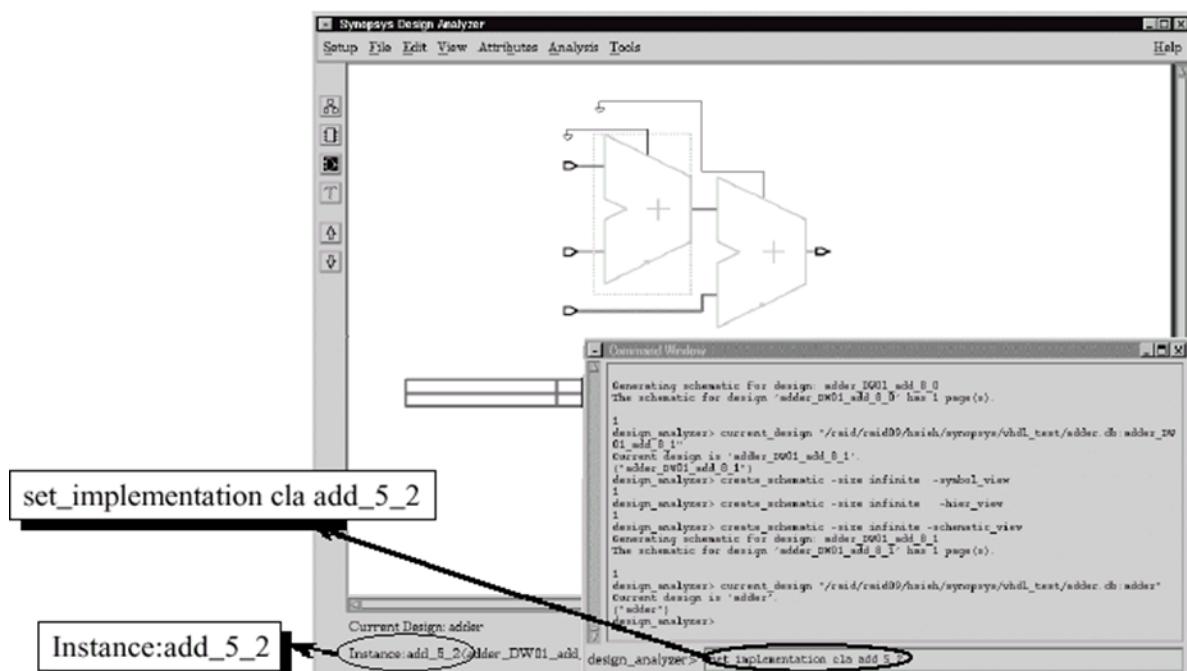


## DesignWare – set\_implementation

- 指定DesignWare元件的实现方式
  - 选择要设定实现方式的元件，查看其实例名是什么。
  - 根据Synopsys联机文档的DesignWare部分选择实现方式
  - 使用dc\_shell命令

*set\_implementation implementation\_name instance\_name*

- compile
- report -resource



## DesignWare 仿真

- 推断方式：和以前一样，不需要任何专门处理。
- 实例化方式：在\$SYNOPSYS目录，可以找到designware的仿真模型，在做仿真时加上这个仿真模型  
- 其实际目录是：
  - » \$SYNOPSYS/dw/dw0x/src -- for VHDL
  - » \$SYNOPSYS/dw/dw0x/src\_ver -- for verilog

```
//synopsys translate_off
`include "/synopsys/synthesis/cur/dw/dw01/src_ver/DW01_sub.v"
`include "/synopsys/synthesis/cur/dw/dw02/src_ver/DW02_mult.v"
`include "/synopsys/synthesis/cur/dw/dw02/src_ver/DW_div.v"
`include "/synopsys/synthesis/cur/dw/dw02/src_ver/DW_mac.v"
`include "/synopsys/synthesis/cur/dw/dw02/src_ver/DW_square.v"
//synopsys translate_on
```

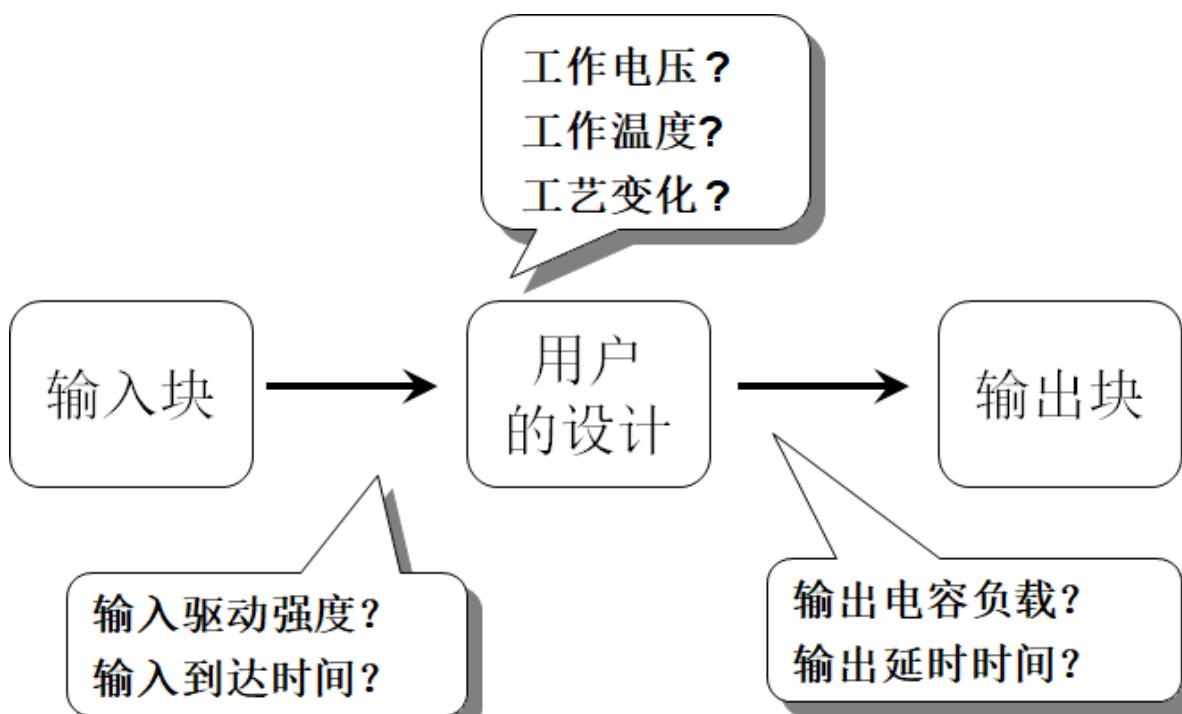
## 设置设计约束

### 设置环境约束

#### 为什么要描述一个真实的外部环境

- 必须要清楚，缺省的条件是不实际的
  - 输入驱动不是无限大。
  - 负载电容通常不是0
  - 要考虑工艺，温度，及电压的变化
- 工作环境影响目标库元件的选择以及设计的时序
- 用户定义的真实的环境 描述电路的工作条件。

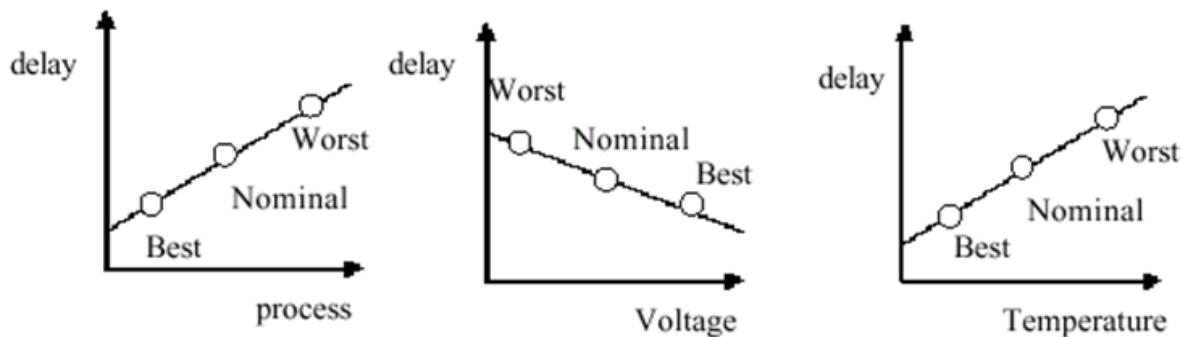
#### 设计环境-- *Operating Environment*



## 工作条件—在综合库中定义

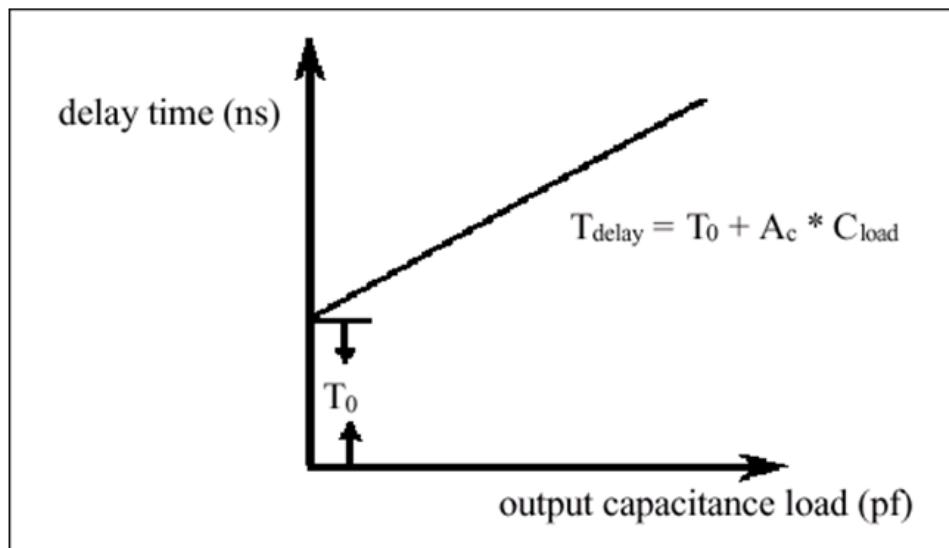
- 工作条件模型 scale 元件延时,
- 指导优化器模拟工艺、温度和电压变化。

Name	Process	Temp	Volt	Interconnection Model
WCCOM	1.32	100.00	2.7	worst_case_tree
BCCOM	0.73	0.00	3.6	best_case_tree
NCCOM	1.00	25.00	3.3	balance_tree



## 输入驱动强度—驱动电阻

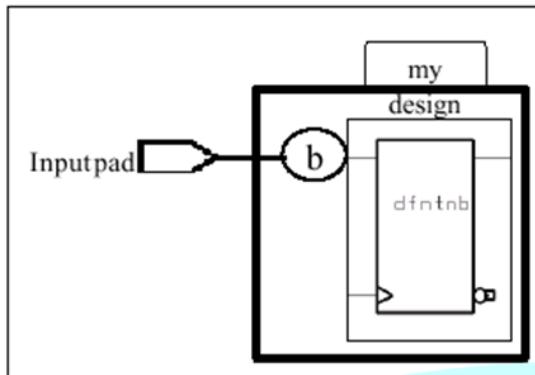
- $T_{\text{delay}} = T_0 + A_c * C_{\text{load}}$ 
  - $T_0$ : 单元的pin-pin固有延时
  - $A_c$ : 驱动电阻 (单位: ns/pf)



## PAD的输入驱动强度

- 如果设计如下:

假定使用的输入PAD为PC3D01, 如下图所示。我们可以设置输入驱动强度为0.2468 (ns/pf)



### 3V CMOS Input Only Pads PC3D01, PC3D11, PC3D21, and PC3D31

#### Performance Equations

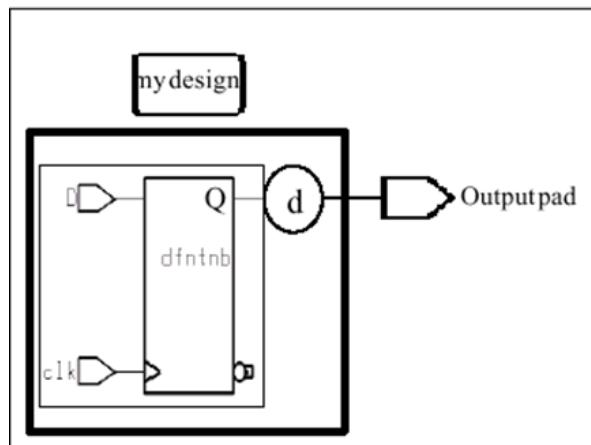
PC3D01		RSE	FALL
iCD	PAD->CIN	$0.3377 + 0.1133 + 0.2468 \cdot C_{ld}$	$0.2566 + 0.0305 + 0.1978 \cdot C_{lc}$
PC3D11		RSE	
iCD	PAD->CIN	$0.3261 + 0.0232 + 0.2265 \cdot C_{ld}$	$0.2582 + 0.0255 + 0.1990 \cdot C_{lc}$
PC3D21		RSE	
iCD	PAD->CIN	$0.7915 + 0.0374 + 0.2618 \cdot C_{ld}$	$0.6064 + 0.0395 + 0.2763 \cdot C_{lc}$
PC3D31		RSE	
		$0.5478 + 0.0289 + 0.2256 \cdot C_{ld}$	
		$0.2971 + 0.0255 + 0.1976 \cdot C_{lc}$	

$$0.3377 + 0.1133 + 0.2468 \cdot C_{ld}$$

## PAD的输出负载

- 如果设计如下

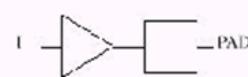
假设使用的输出PAD为PC3O01, 如下列所示。我们可以设置输出负载为0.096 (pf)



### 3V CMOS Output Pads

#### PC3O01 through PC3O05

PC3O01 through PC3O05 cells are CMOS output pads with AC drive capabilities ranging from 1x to 5x.



Function Table	
INPUT	OUTPUT
I	PAD
L	L
H	H

#### Cell Description

Macro Name:	PC3O01	PC3O02	PC3O03	PC3O04	PC3O05
Drive Capability	1x	2x	3x	4x	5x
Width (mils)	3.2	5.4	5.4	5.4	5.4
Power (µW/MHz)	198.55	193.59	193.75	198.31	197.58

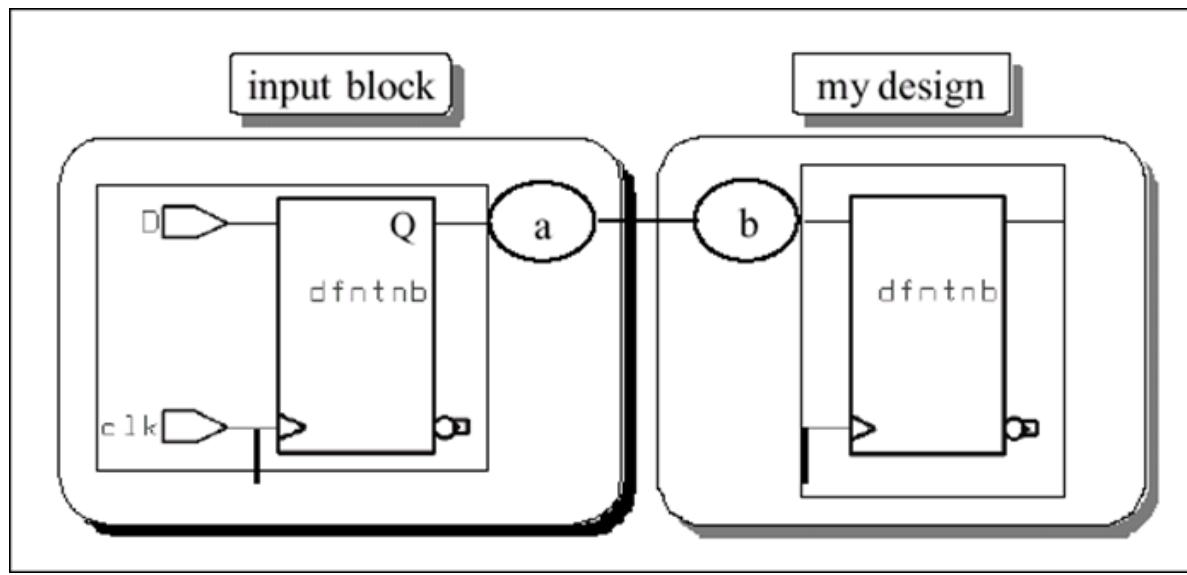
#### Pin Description

NAME	Capacitance (fF)					Description
	PC3O01	PC3O02	PC3O03	PC3O04	PC3O05	
I	0.096	0.096	0.096	0.132	0.133	Input
PAD	8.51	8.51	8.51	8.50	8.51	Output

## 负载预算--Load Budget

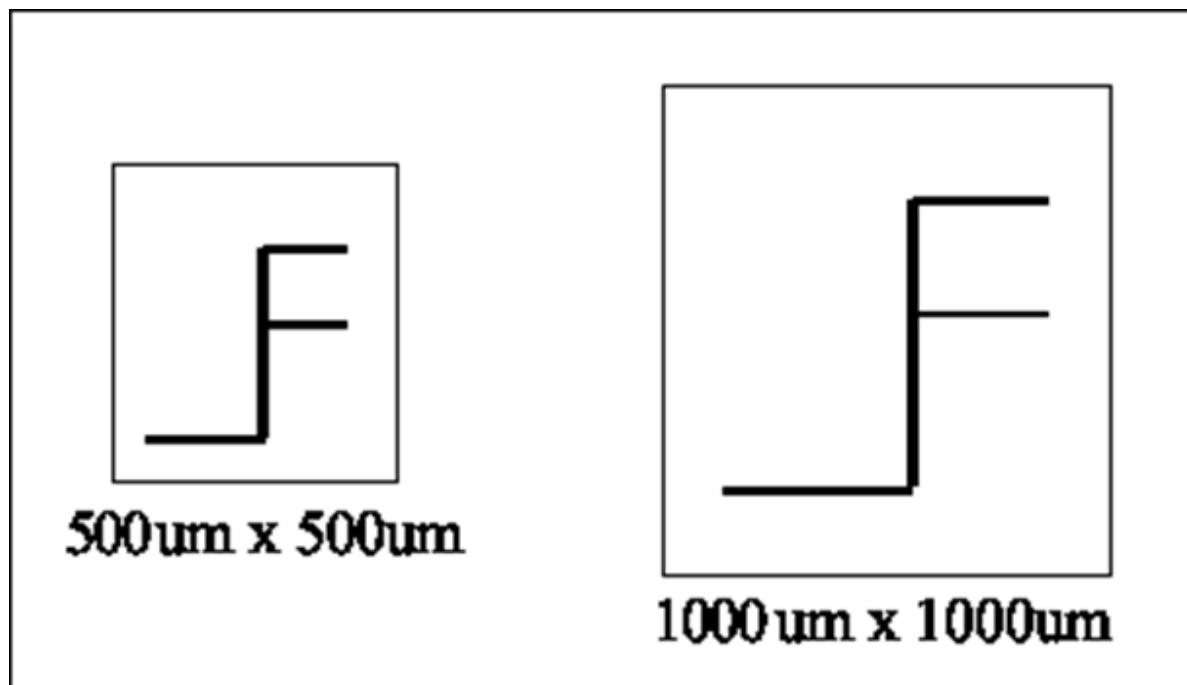
- 可以采用下列规则：

- 假设输出端口是由驱动能力弱的单元驱动（最小的反相器）
- 限制每个输入端口的输入电容（如负载不大于10个AND2的电容）
- 估算输出端口驱动模块的数目

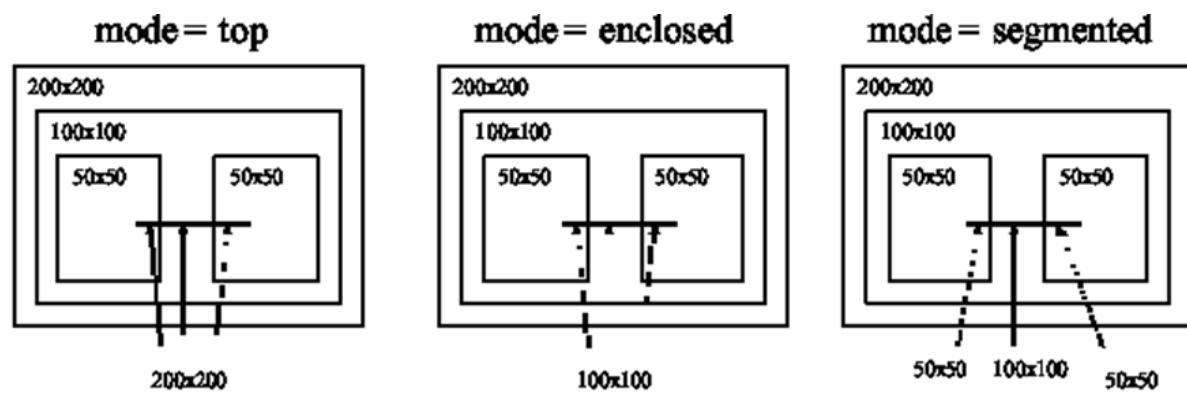


### Wire Load Model

- wire load model根据芯片面积和标准单元的扇出估算连接线上的电容
- 在编译时设置这项信息，可以更精确的建立设计模型。



三种模型



## Wire Load Model举例(Design Time)

- 要计算连接线的R、C及连线面积
- 确定连接线的fanout数
  - 在wire\_load\_model的fanout-length对中查找连接线长度
  - 长度乘上电容 (或R或面积) 系数  
(fanout = 3 ® length = 2.8)

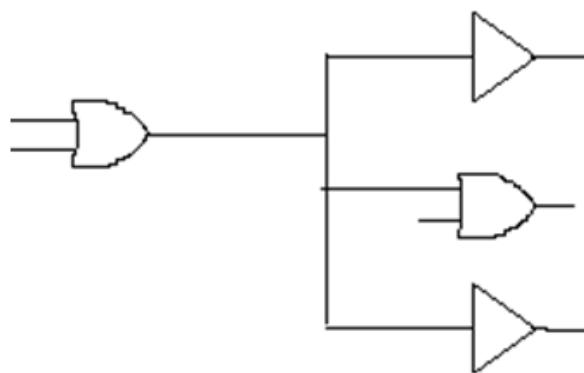
Cwire = length(2.8) ' capacitance coefficent (1.3) =3.64 load units

Rwire = length (2.8) ' resistance coefficient (3.0) = 8.4 resistance units

Net area = length (2.8) ' area coefficient (0.04) = 0.112 net area units

Cwire = (3.3 + (7-4)\*0.15) ' capacitance coefficent (1.3)

```
wire_load ("500x500") (
  resistance : 3.0
  capacitance: 1.3
  area: 0.04
  slop: 0.15
  fanout_length ( 1 , 2.1 )
  fanout_length ( 2 , 2.5 )
  fanout_length ( 3 , 2.8 )
  fanout_length ( 4 , 3.3 )
```



## 设置设计约束

### 约束(constraint)

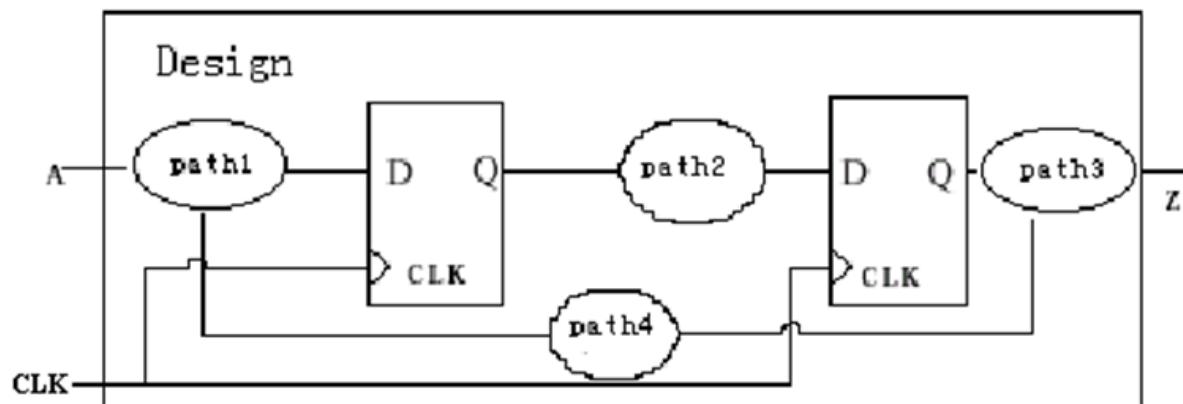
- 约束是Design Compiler优化一个设计到目标工艺库的目标
- 设计规则约束：与工艺相关的限制，如最大的传输时间、最大的扇出、最大电容等。
- 优化约束：设计目标及要求。如最大延时、最小延时、最大面积、最大功耗等。
- 在编译时，Design Compiler试图满足所有约束。

### 优化约束

- 按关注的次序，优化约束有：

- 最大延时
  - 最小延时
  - 最大功耗
  - 最大面积
- 对于组合电路，我们在时序上只需设置最大延时和最小延时

## 需要什么设计约束?

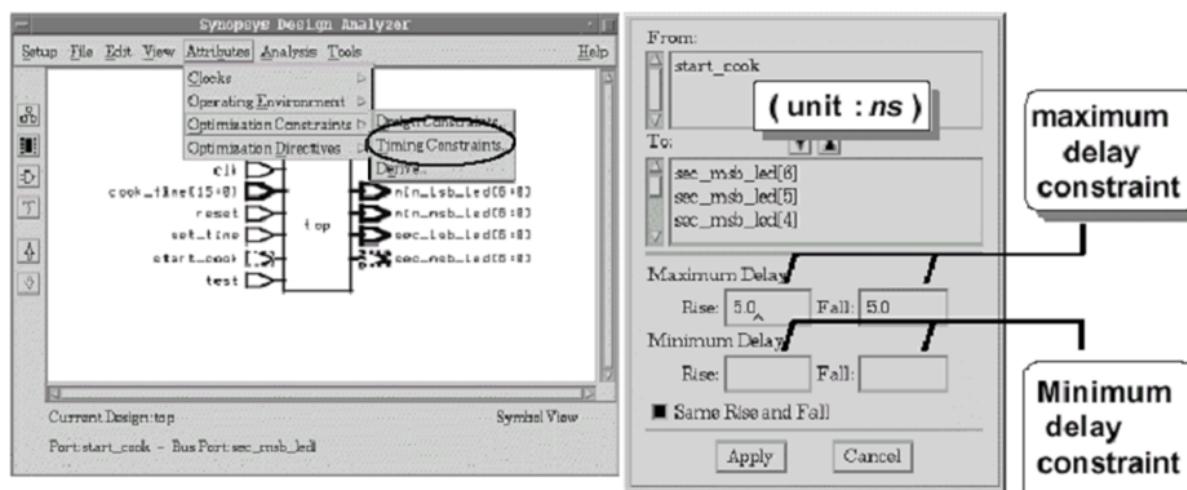


需要约束的时序路径有四类:

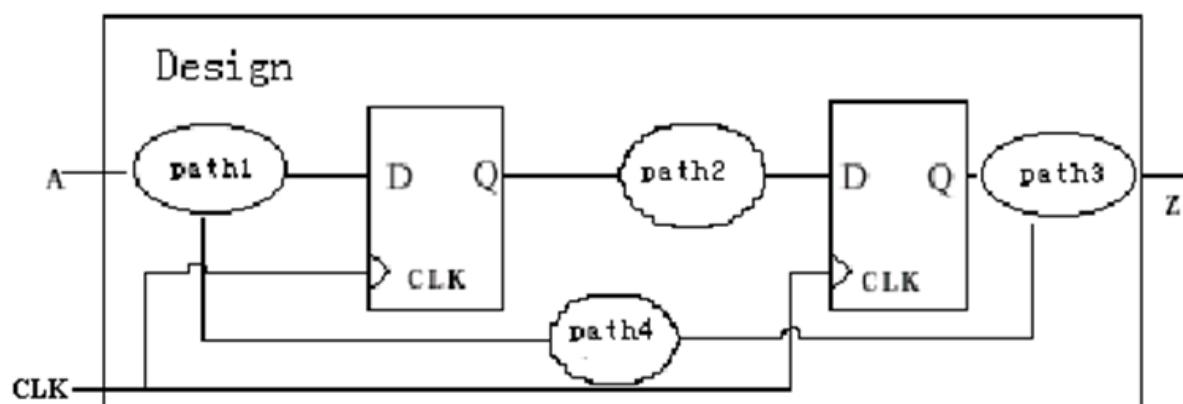
1. 输入端口到输出端口的组合逻辑
2. 输入端口到时序元件的数据输入端
3. 时序元件的时钟端到时序元件的数据输入端
4. 时序元件的时钟端到输出端口

### Maximum Delay Constraint

- 对于组合电路，主要是
  - 选择时序通路起点和终点。
  - *Attributes/Optimization Constraints/Timing Constraints*



### 时序元件相关的路径

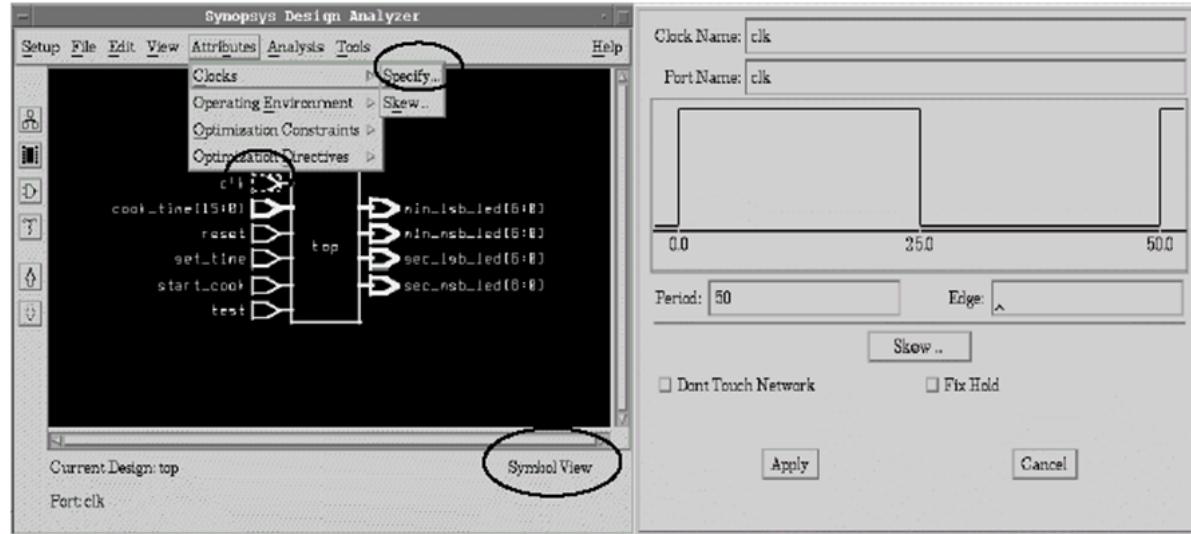


时序元件相关的时序路径有三类：

- 1.输入端口到时序元件的数据输入端
- 2.时序元件的时钟端到时序元件的数据输入端
- 3.时序元件的时钟端到输出端口

## 时序电路->指定时钟

- 选择时钟端口
- *Attributes/Clocks/Specify*



- **creat\_clock:** 定义时钟的波形，并考虑所有时钟触发器(**flip-flop**)的建立时间(**set-up time**)要求。

```
dc_shell> create_clock "clk"-period 50 -waveform {0 25}
```

- **set\_fix\_hold:** 考虑所有时钟触发器的保持时间(**hold time**)要求。

```
dc_shell> set_fix_hold clk
```

- **set\_dont\_touch\_network:** 不在时钟网络上设置缓冲器

```
dc_shell> set_dont_touch_network clk
```

## 时钟树建模

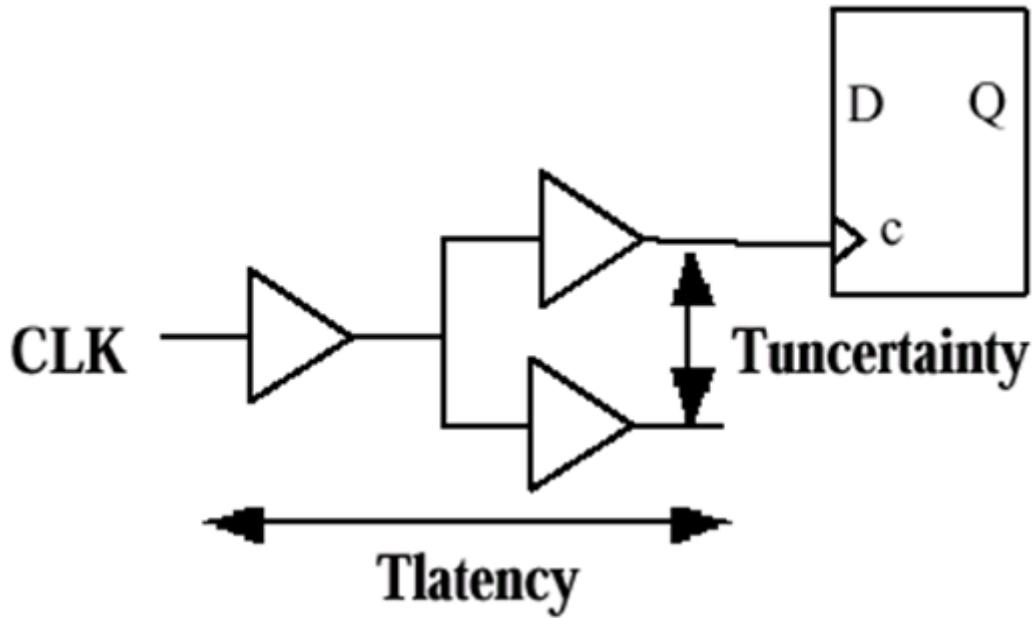
- 时钟建模需要两个参数

1.指定时钟网络的延迟

```
» set_clock_latency -rise tr -fall tf find (clock, CLK)
```

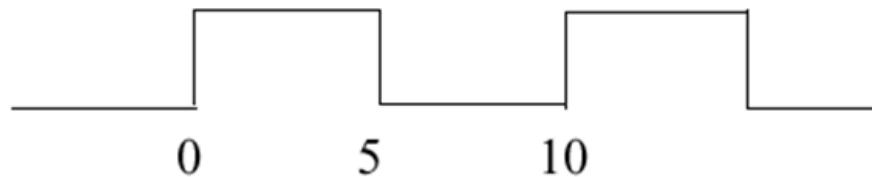
2.指定时钟网络的倾斜(uncertainty)

```
» set_clock_uncertainty -rise tp -fall tm find (clock, CLK)
```



时钟树建模举例

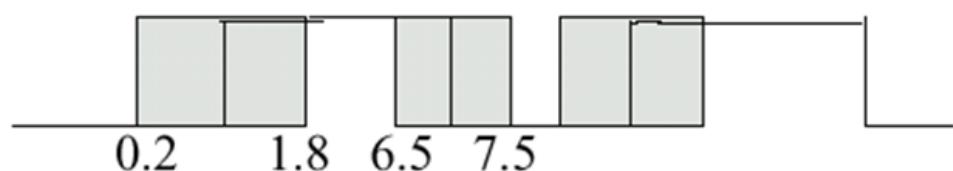
`create_clock -period 10 -waveform {0 5} find (port CLK)`



`set_clock_latency -rise 1 -fall 2 find (port CLK)`



`set_clock_uncertainty -rise 0.8 -fall 0.5 find (port CLK)`



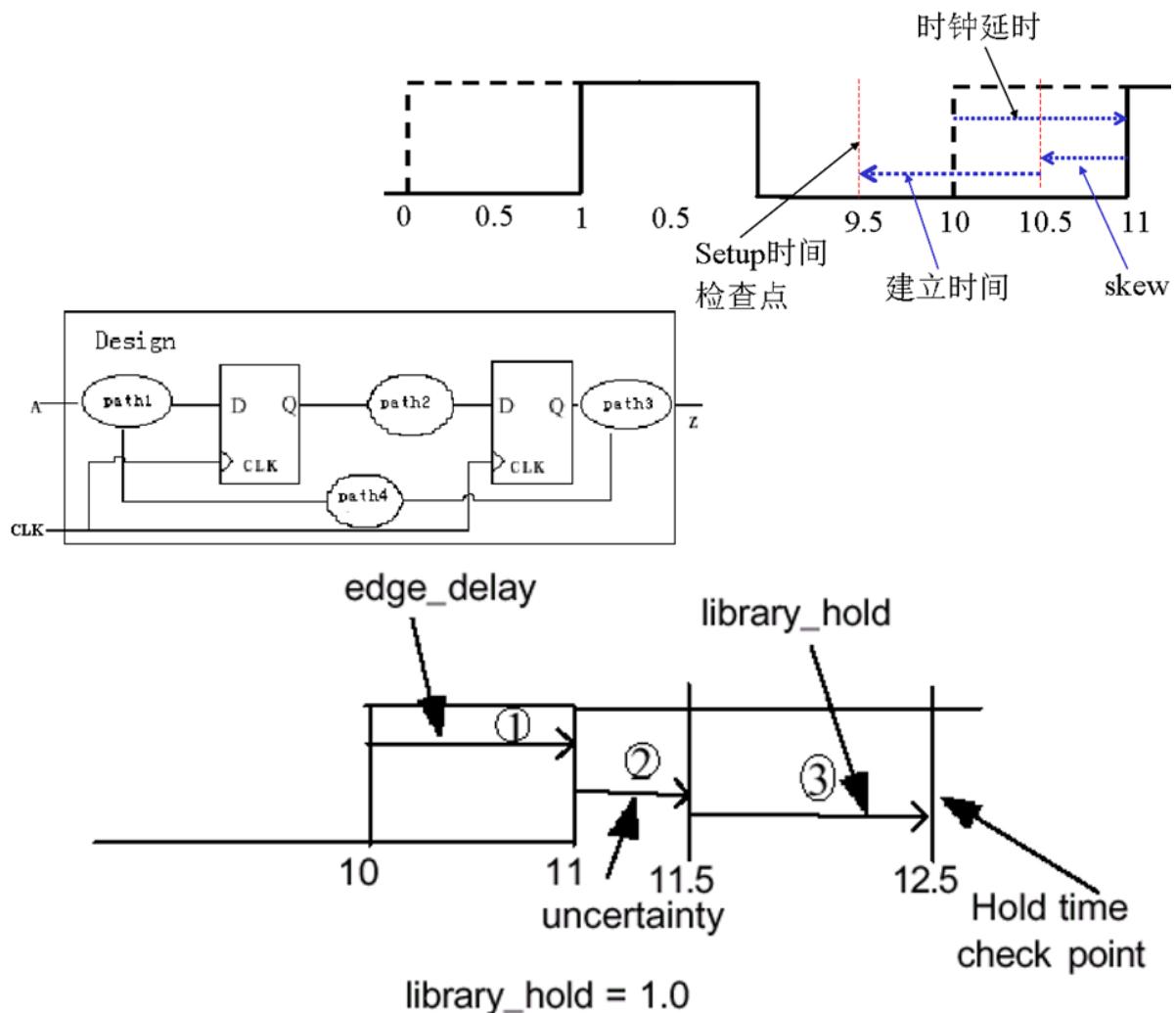
时钟树建模对建立时间的影响

假设库中上升沿D触发器(Flip Flop) setup time = 1ns

`create_clock -period 10 -waveform {0 5} find (port CLK)`

`set_clock_latency -rise 1 -fall 2 find (port CLK)`

`set_clock_uncertainty -rise 0.5 -fall 0.7 find (port CLK)`



$$\text{Hold time check} = (\text{clock\_edge} + \text{edge\_delay} + \text{uncertainty} + \text{lib\_hold})$$

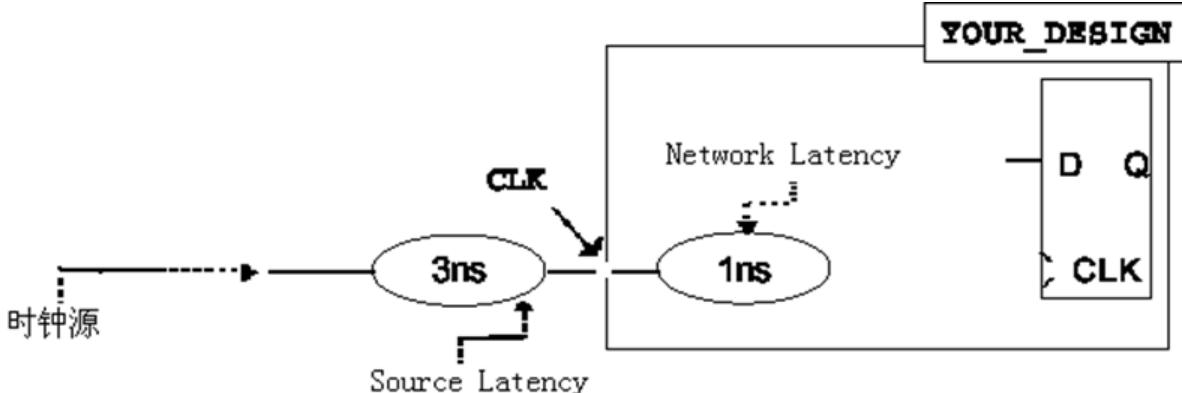
### 时钟源延迟模型

- 源延迟是从实际时钟源到设计的时钟定义点的传播延迟

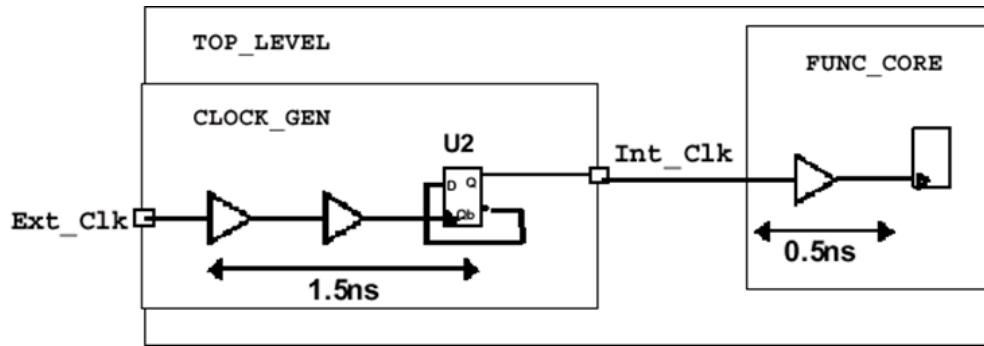
```
create_clock -period 10 find(port, CLK)
```

```
set_clock_latency -source 3 find(port, CLK)
```

```
set_clock_latency 1 find(port, CLK)
```



## 衍生时钟(Derived Clock)



方法 I

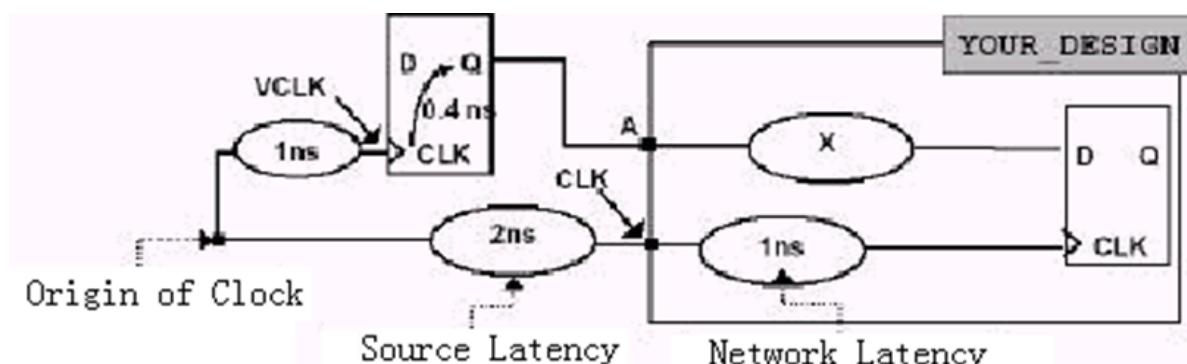
```
create_clock -p 50 find(port, Ext_Clk)
create_clock -name Int_Clk -p 100 find(pin, CLK_GEN/U2/Q)
set_clock_latency -source 1.5 find (pin, CLK_GEN/U2/Q)
set_clock_latency 0.5 find (pin, CLK_GEN/U2/Q)
```

方法 II

```
create_clock -p 50 find(port, Ext_Clk)
create_generated_clock -name Int_Clk -source Ext_Clk \
    -divide_by 2 find (pin, CLK_GEN/U2/Q)
set_clock_latency -source 1.5 find (pin, CLK_GEN/U2/Q)
set_clock_latency 0.5 find (pin, CLK_GEN/U2/Q)
```

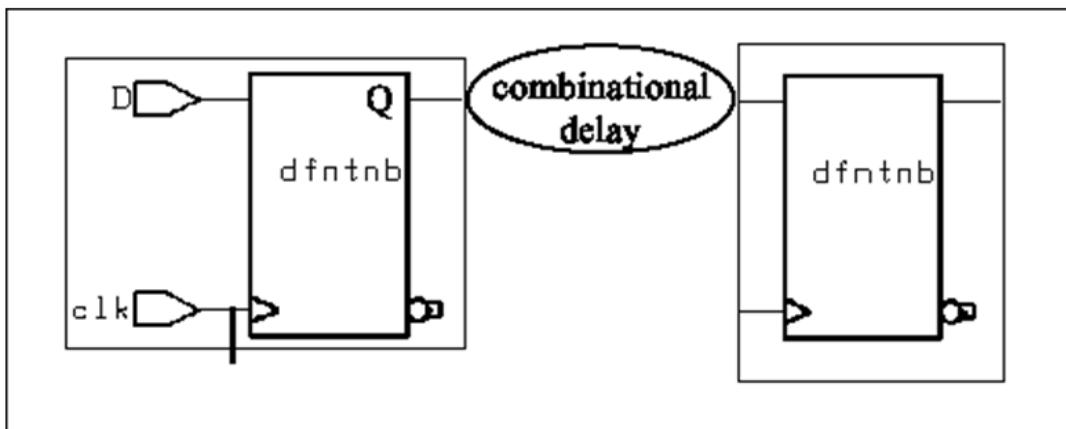
## 外部时钟延迟(举例)

```
current_design my_design
create_clock -p 10 find(port, CLK)
create_clock -p 10 -name VCLK      //虚拟时钟
set_clock_latency -source 2 find(clock, CLK)
set_clock_latency -source 1 find(clock,VCLK)
set_clock_latency 1 find(clock, CLK)
/* set_propagated_clock all_clocks() */ /*For post-layout synthesis*/
set_input_delay 0.4 -clock VCLK find(port, A)
```



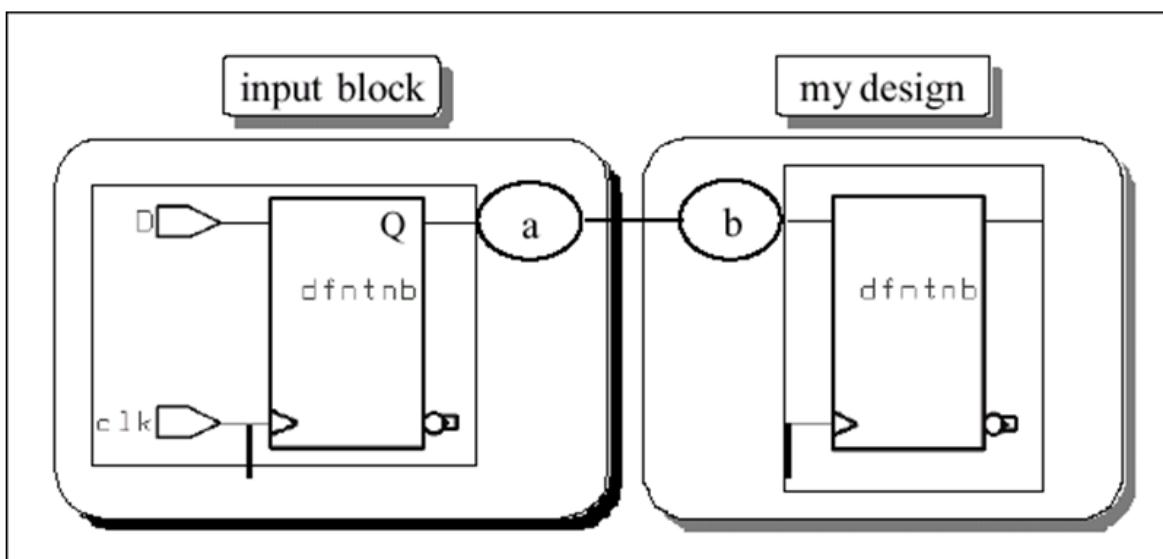
## 时序电路

- 时序电路通常包含一个指定的时钟
- $\text{clock cycle} \geq \text{DFF}_{\text{clk-Qdelay}} + \text{combinational delay} + \text{DFF}_{\text{setup}}$

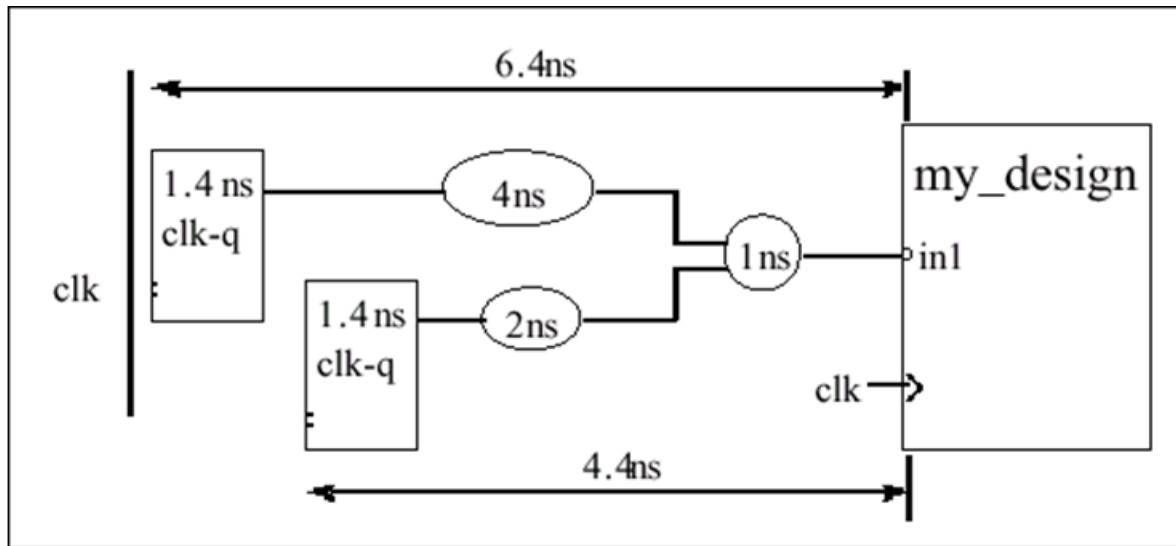


输入延时模型

- $\text{Clock cycle} \geq \text{DFF}_{\text{clk-Qdelay}} + a + b + \text{DFF}_{\text{setup}}$
- $\text{Input delay} = \text{DFF}_{\text{clk-Qdelay}} + a$



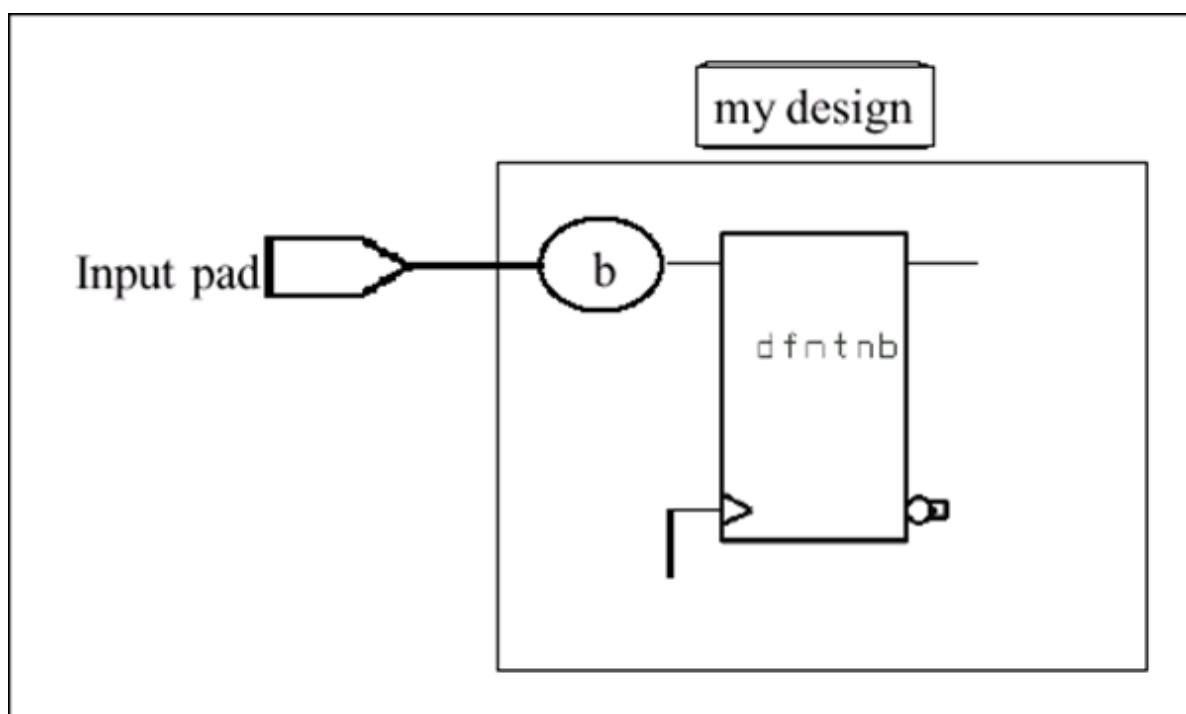
设置输入延时



```
dc_shell> set_input_delay -clock clk -max 6.4 in1
```

```
dc_shell> set_input_delay -clock clk -min 4.4 in1
```

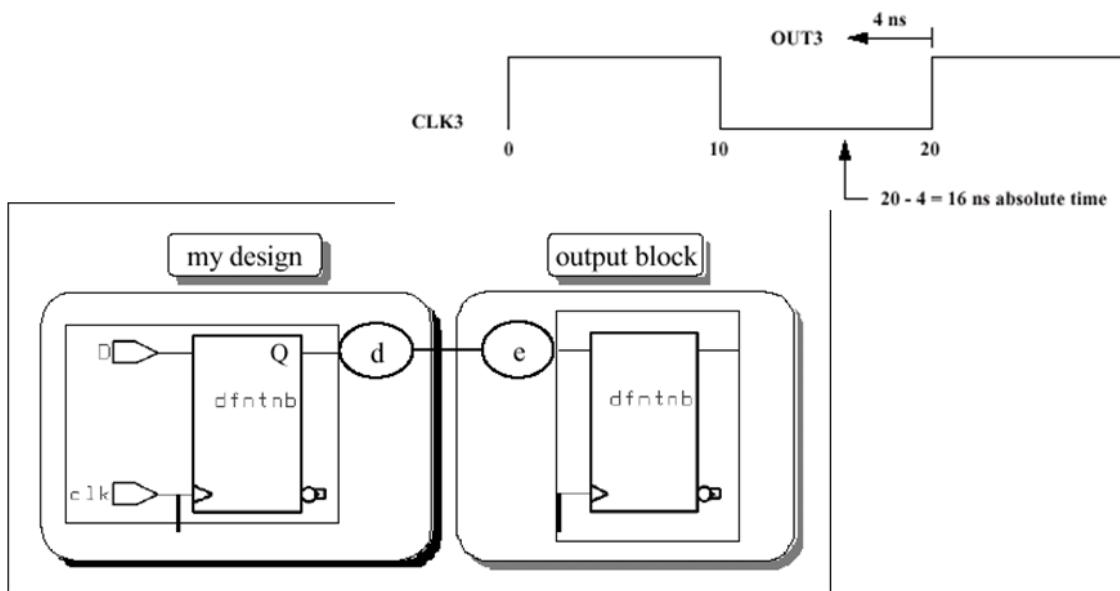
- 如果输入是PAD( top level), 则将输入延时设置一个适当值。 (参考PAD的数据手册或使用 *characterize*命令)



- 时钟周期  $\geq \text{DFF}_{\text{clk-Qdelay}} + d + e + \text{DFF}_{\text{setup}}$

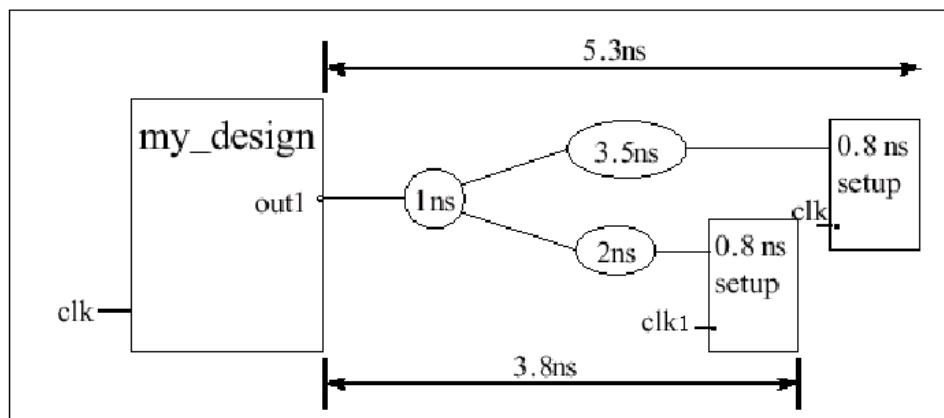
- 输出延时 =  $e + \text{DFF}_{\text{setup}}$

Set\_output\_delay 4 out3 -clock clk



#### 设置输出延时

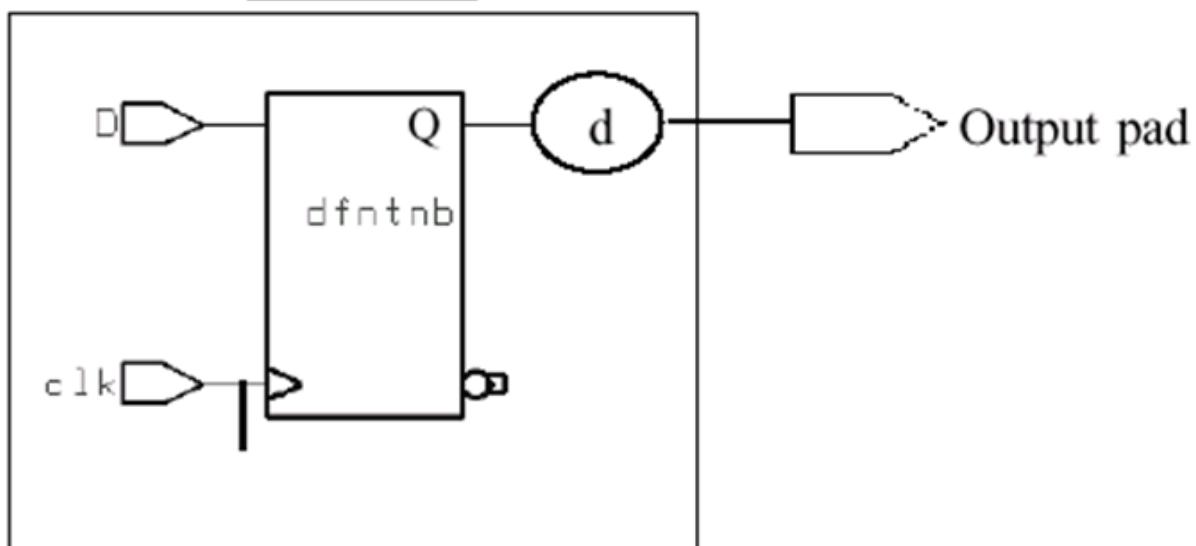
- `dc_shell> create_clock -period 10 -waveform {0 5} clk1`
- `dc_shell> set_clock_latency 0.4 clk1`
- `dc_shell> set_clock_uncertainty 0.1 clk1`
- $\text{output\_delay} = \text{path\_delay} + \text{uncertainty} + \text{setup} - \text{clock\_delay}$   
 $= 4.5 + 0.1 + 0.8 - 0.4 = 5.0$



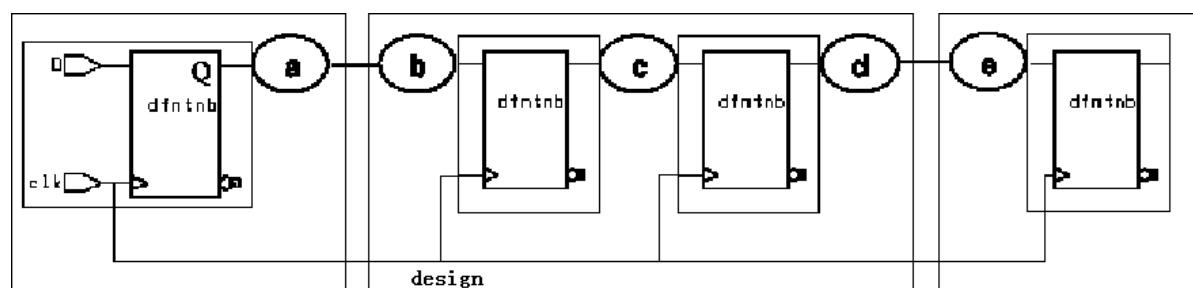
`dc_shell>set_output_delay -clock clk -max 5.0 out1`

- 如果设计的输出连接到PAD(top level), 将输出延时设置为一个适当的值。(参考PAD数据手册或使用`characterize`命令)

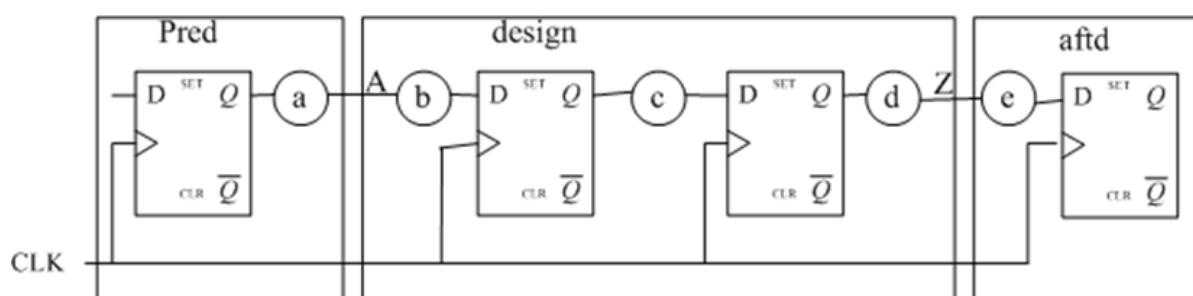
## my design



What have we modeled?



- 假设 clock cycle = p
- 输入延时 = a ; a + b < p
- 输出 延时= e ; d + e < p



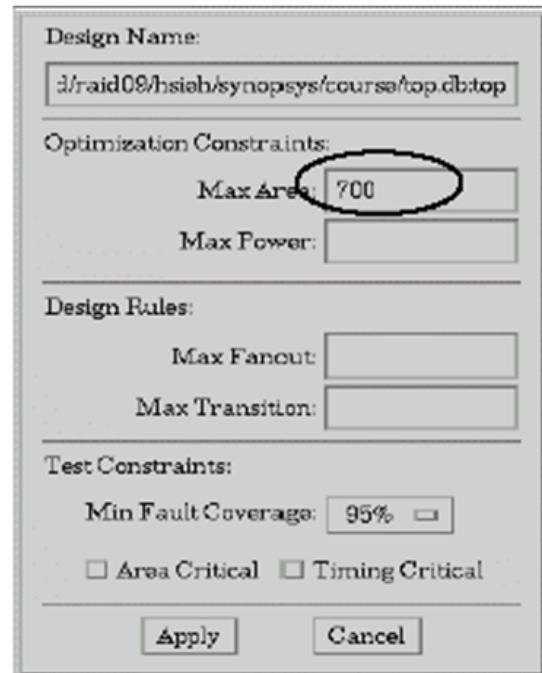
- 假设时钟CLK的时钟周期(clock cycle) = p
- 输入延时 = a ; a + b < p
- 输出 延时= e ; d + e < p

设置面积约束

- **Attributes/Optimization Constraints/Design Constraints**
- 如果只考虑面积,不关心时序, 可以使用下面的约束描述:
  - ***remove constraints –all***
  - ***set max\_area 0***
  - ***compile –effort medium***

面积单位:

- (1) 等效门数  
 (2) *um x um*



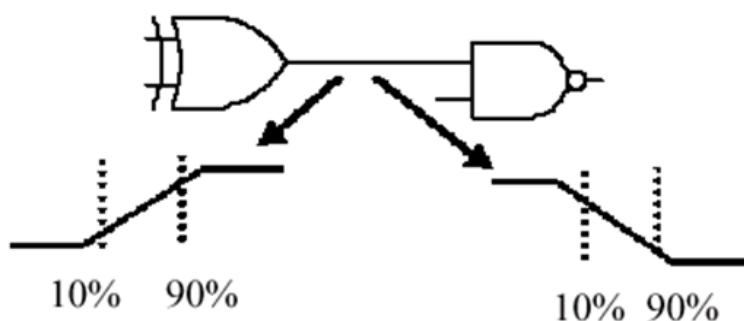
## 设计规则约束

- 设计规则 (design rule)是绝对不能违反的, 即使时间及面积约束不能满足也在所不惜。
- 设计规则约束有三类:
  1. *set\_max\_capacitance*
  2. *set\_max\_transition*
  3. *set\_max\_fanout*

设置最大转换时间

- ***set\_max\_transition***
  - 设置端口或设计中的最大转换时间.
  - **Example:**

```
set_max_transition 5 all_inputs()
set_max_transition 3 all_outputs()
```

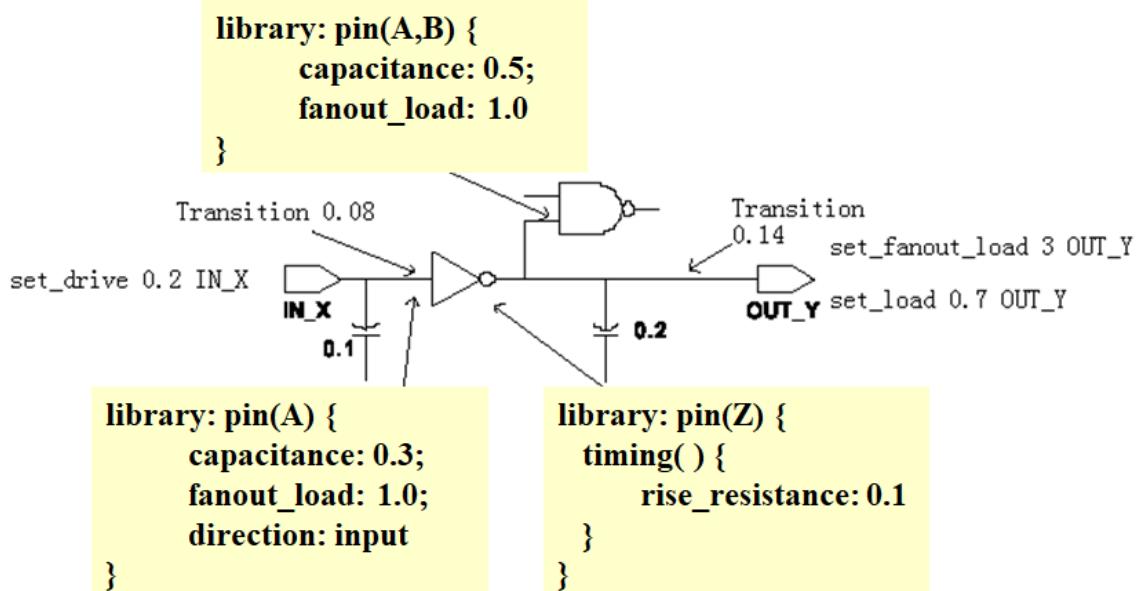


Rising edge on a signal

Falling edge on a signal

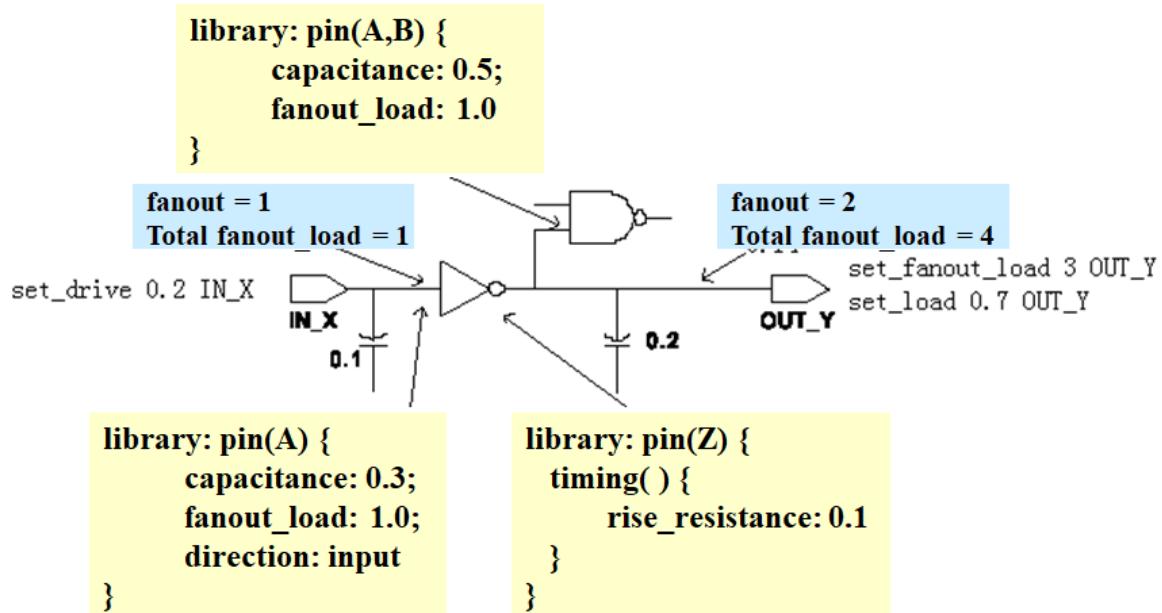
计算最大转换时间

- $\text{Transition Time} = \text{Drive (resistance)} * \text{Max_Cap}$   
 $= \text{Drive (resistance)} * \text{Load} (\Sigma C_{pins} + C_{wireload})$



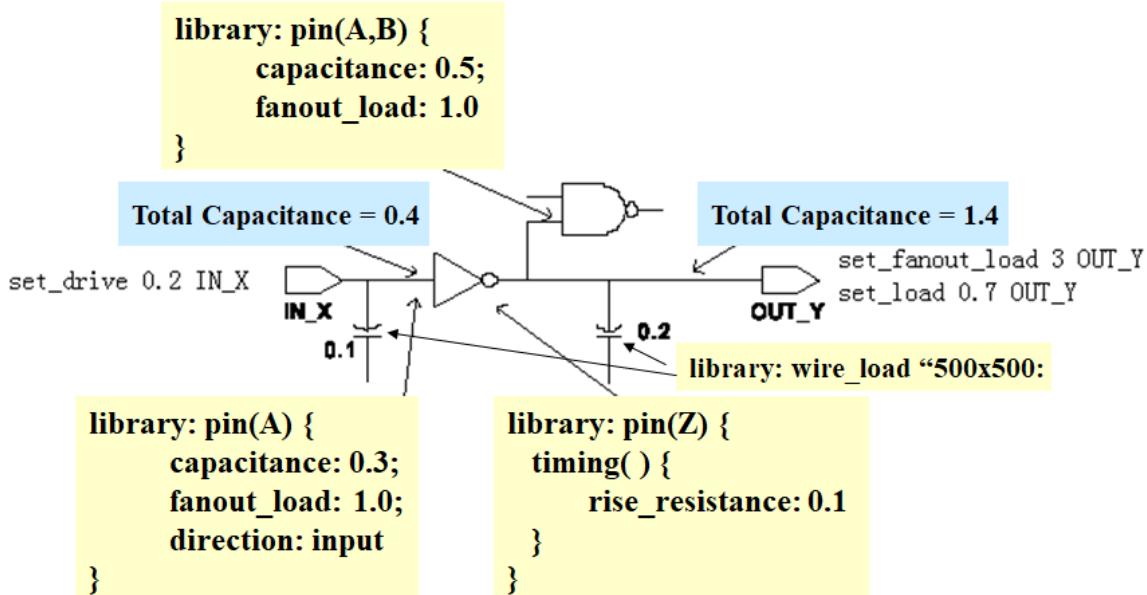
计算最大的fanout\_load

- $\text{Maximum fanout_load} = \Sigma \text{fanout_loads}$ 
  - set\_max\_fanout value object (object是输入端口或设计)



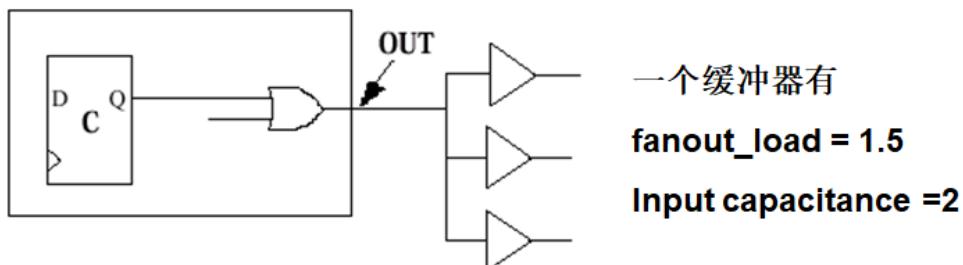
计算最大电容

- Maximum capacitance = Load( $\sum C_{pins} + C_{wireload}$ )
- set\_max\_capacitance capacitance\_value object



设置扇出负载

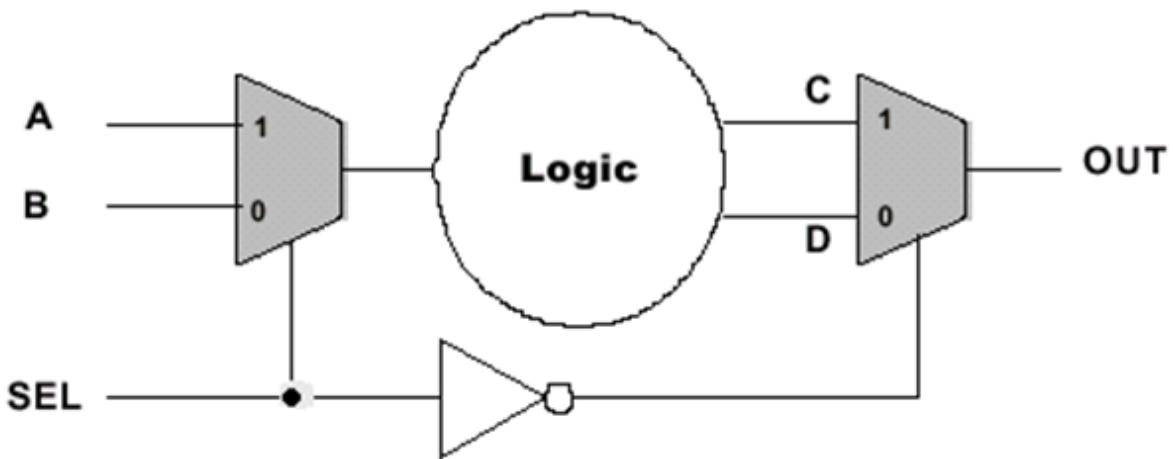
- 使用 **fanout\_load** 调节 **max\_fanout**
- 语法: **set\_fanout\_load fanout portlist** (输出端口列表)
- 设计规则: 外部 **fanout\_load** + 内部 **fanout\_load** 必须小于 **max\_fanout\_load**
  - “OUT”的外部 **fanout\_load** 为 4.5 ( $1.5 \times 3$ )
  - “OUT”的外部电容为 6 ( $2 \times 3$ )
  - “OUT”的 **fanout** 数为 3 (3 buffers)



dc\_shell> set\_fanout\_load 4.5 find(port, out)

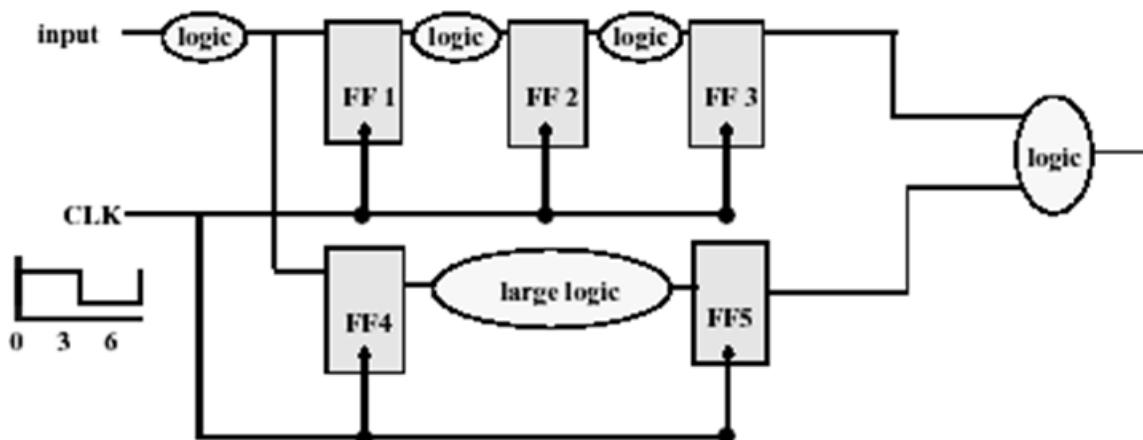
## False Path

- *false path* 是不传播信号的通路，或是忽略这个通路上的时序约束。
- **set\_false\_path** 可以禁止一个通路的基于时序的综合
- 它主要用于:
  - 约束异步通路
  - 从逻辑上约束 *false path*
- **set\_false\_path -from {A} -through {C} -to {OUT}**
- **set\_false\_path -from {B} -through {D} -to {OUT}**



## 多周期通路

- 在有些情况下，两个寄存器之间的组合逻辑延迟可能要求多于一个时钟周期，这些通路应设置为 *multicycle* 通路。



## 设计检查

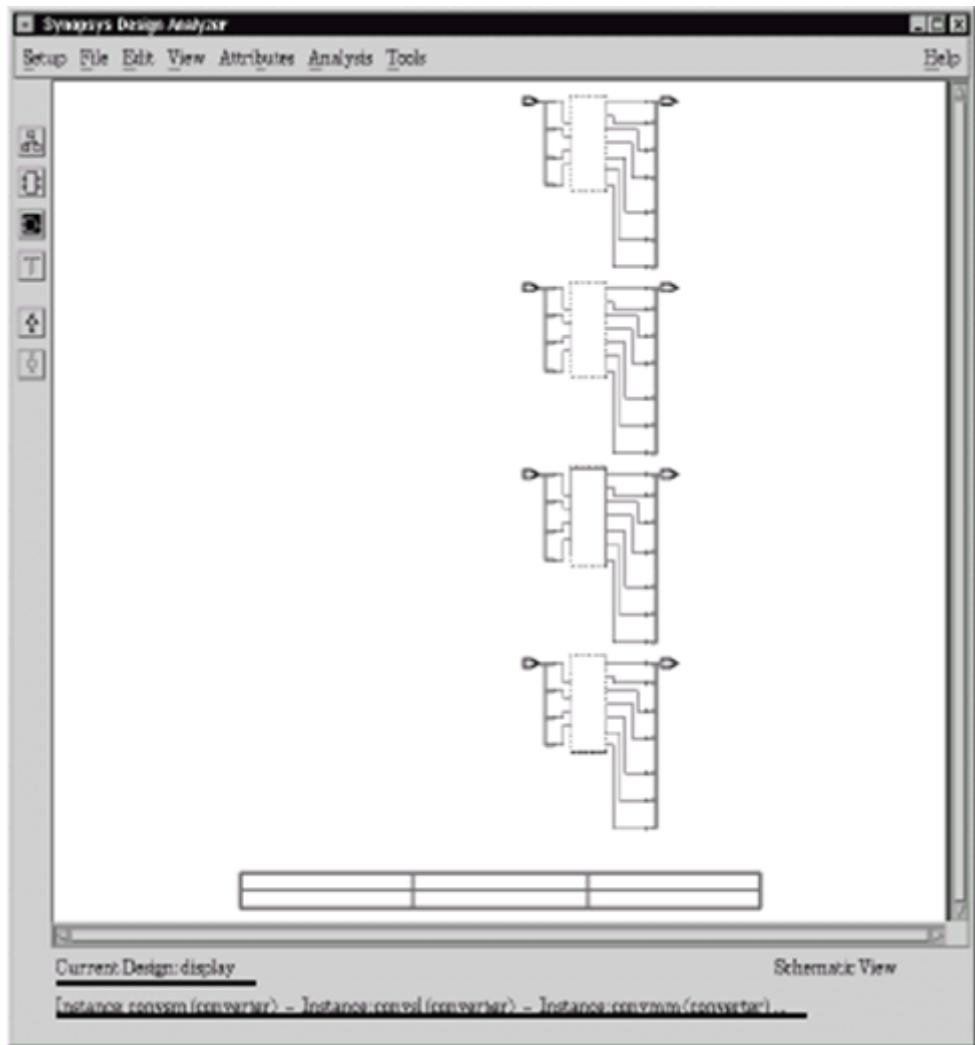
- 在设置设计属性和设计约束后，建议下一步对设计进行检查。
- Analysis/Check Design
- 你可能遇到下列警告：

```

design_analyzer> Warning: Design 'convector' is instantiated 4 times. (LINT-45)
      Cell 'convval' in design 'display'
      Cell 'convval' in design 'display'
      Cell 'convval' in design 'display'
      Cell 'convval' in design 'display'
1
design_analyzer>

```

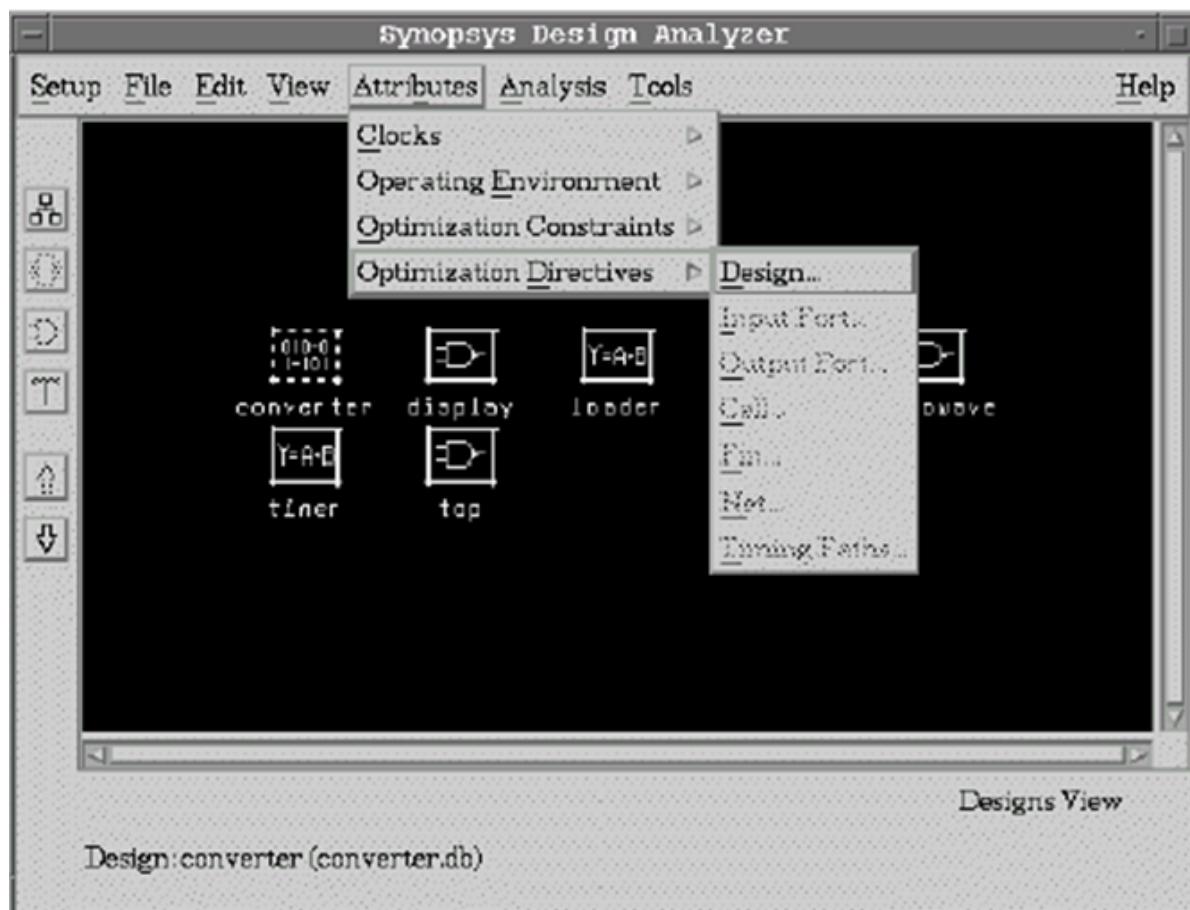
- 这个警告信息称为 “multiple design instance”，它产生的原因是用相同的HDL描述多次实例化 (instance)。
- 如何处理？
  - dont\_touch
  - ungroup
  - uniquify



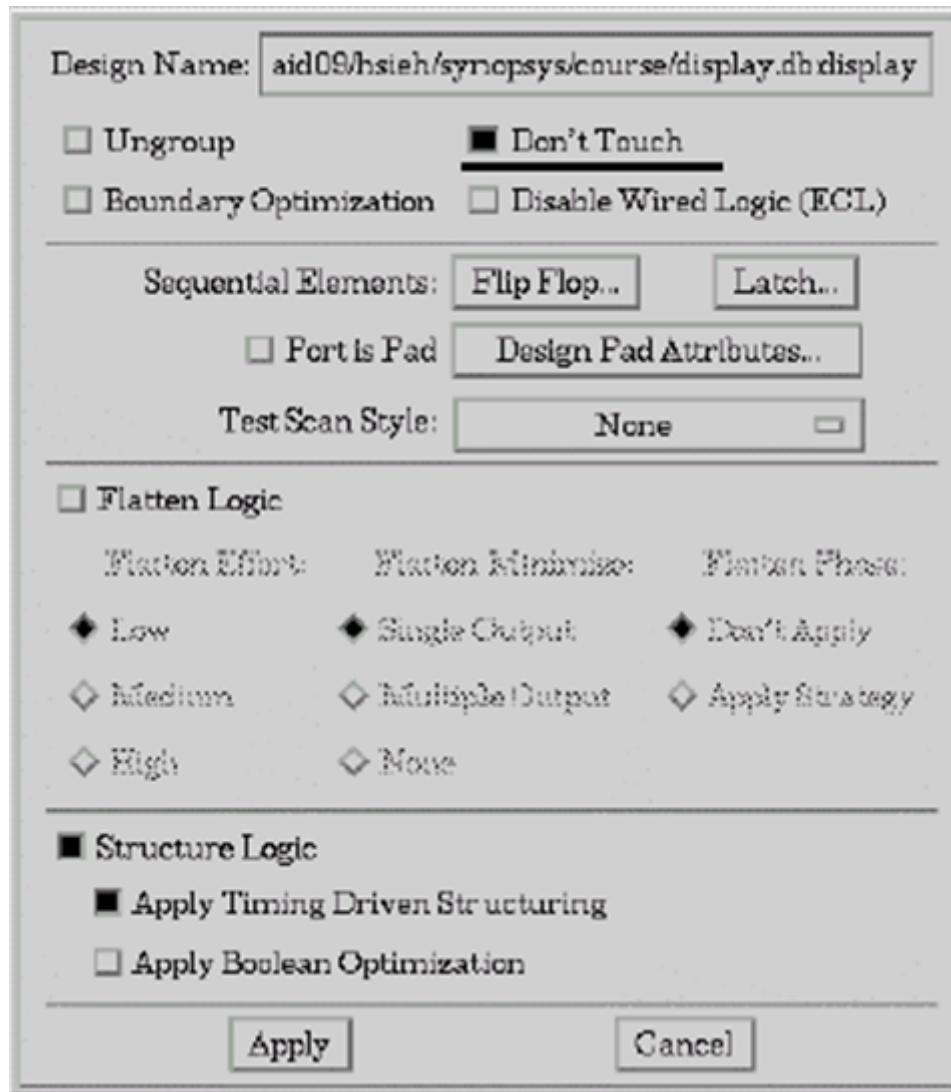
## **dont\_touch**

- 停止对低层设计的重新编译
- 层次将被保留
- 单个设计描述可以共享
- 用于那些不需要用户做优化的块
- 在设计优化过程中, *dont\_touch* 块不会再优化
- 如果*dont\_touch*设置在一个*unmapped* 设计, 设计将保持*unmapped*

*Attributes/Optimization Directives/Design*



- 使用过程
  - 对块进行约束
  - 块编译
  - 选择设计中要多实例化的块
  - Attributes/Optimization Directives/Design, 设置dont\_touch按钮
  - 用层次编译方式编译整个设计



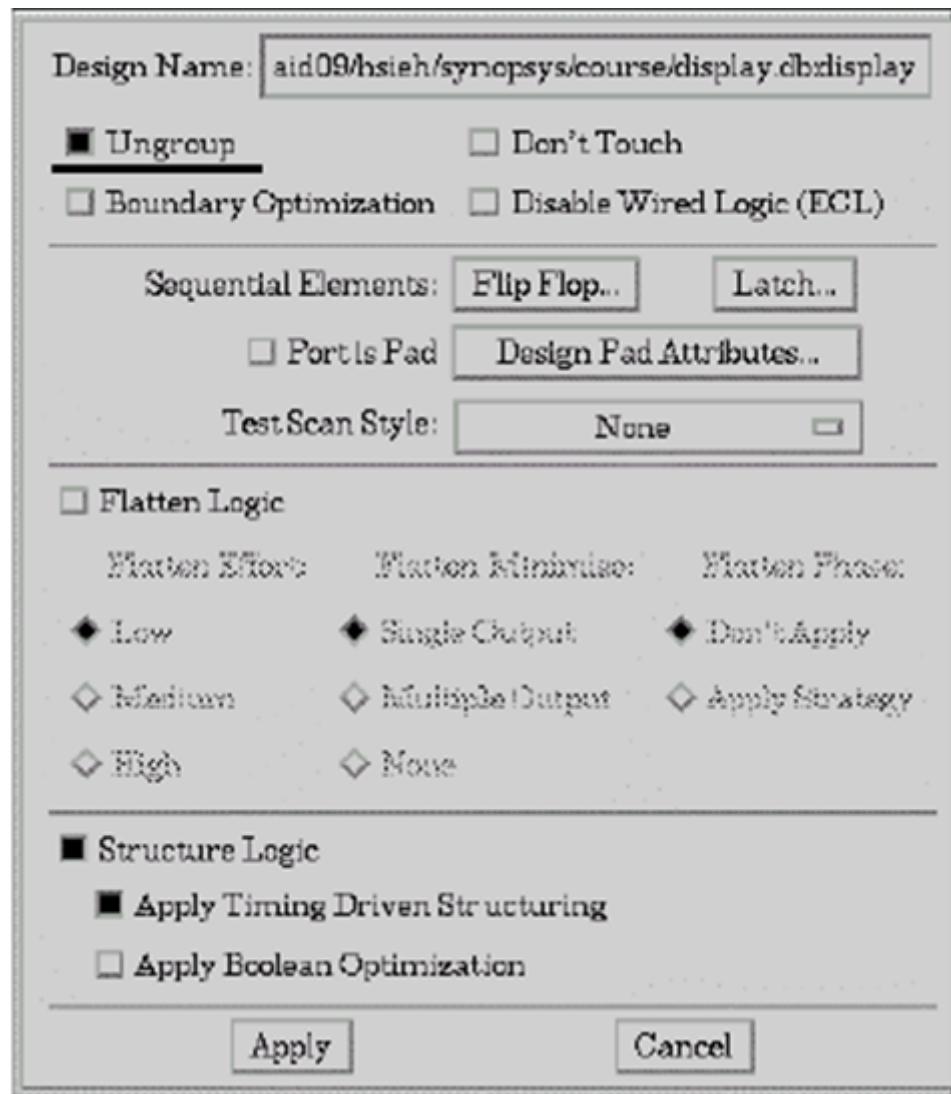
## Ungroup

- 过程
  - 选中多实例化设计块
  - *Attributes/Optimization*

*Directives/Design* 并设置

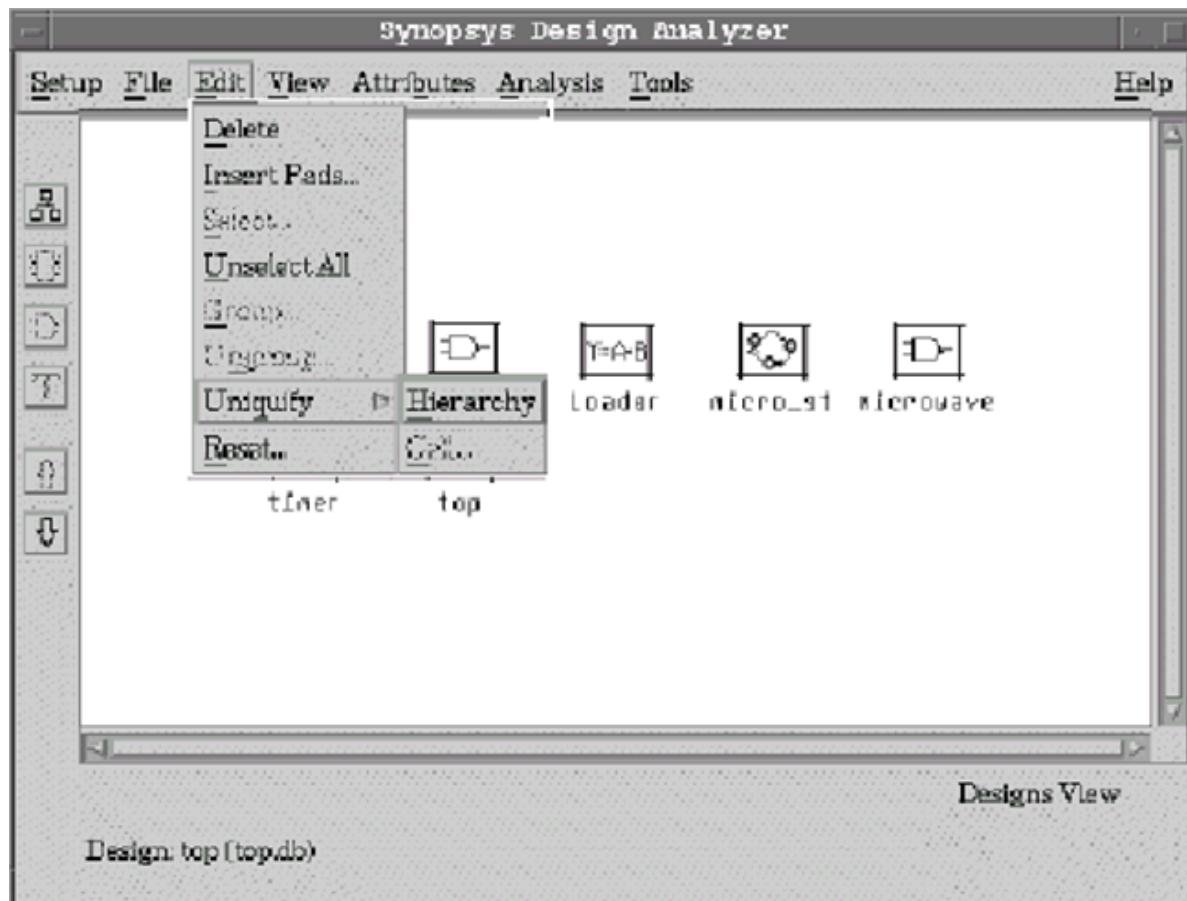
Ungroup按钮

- 用层次化方式编译整个设计
  - 展开设计层次
  - 不保留设计层次
  - 消耗更多的存储器
  - 更多的编译时间
  - 产生最好的设计结果

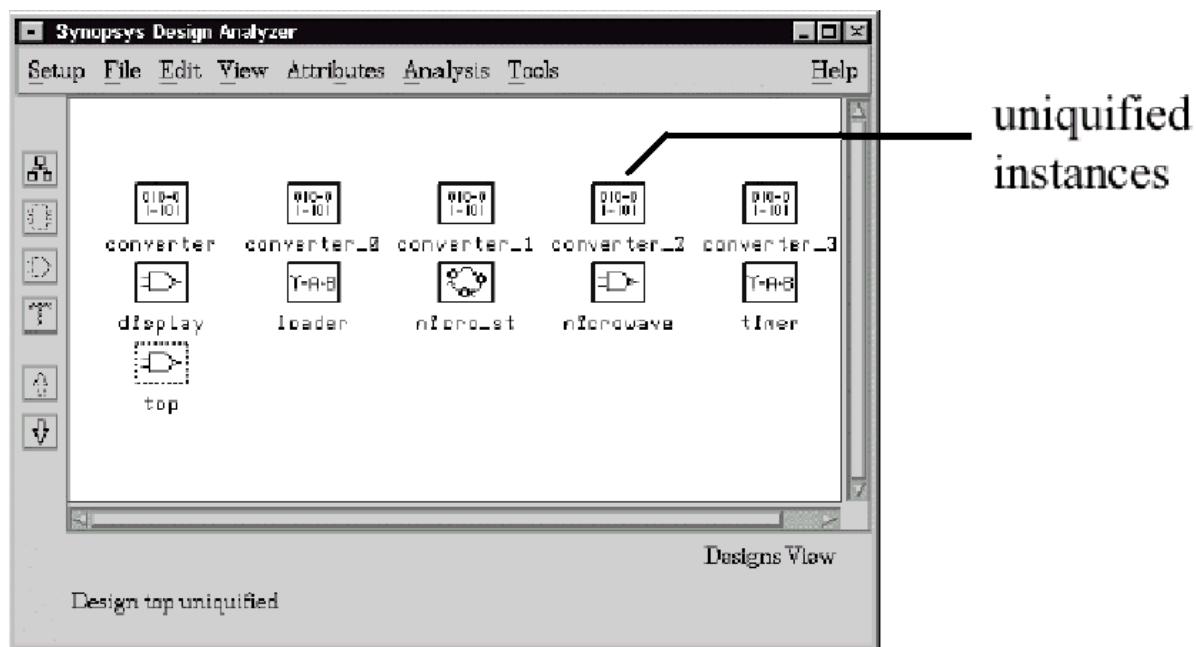


## Uniquify

- 为每一个实例建立一个单独的文件
  - 可以选择一个单元或整个设计层次进行uniquify
  - 允许设计定制自己的界面
  - 如果环境的变化很大，使用*uniquify\*\**，而不使用*Compile+dont\_touch*
  - 与compile+dont\_tough相比，Uniquify 使用更多的存储器和更长的编译时间
- 
- 选择设计层次的最顶层 (top)
  - Edit/Uniquify/Hierarchy*



- 使用变量`uniquify_naming_style**`创建新的设计名称, 缺省为`%s_%d`



## 多设计实例 (总结)

- 采用“`dont_touch`, `ungroup`, `uniquify`”解决
- 最简单的方法是`uniquify`, 但需要较多的存储器和编译时间
- 如果希望保留设计层次并资源共享, 使用`dont_touch`
- 如果希望得到一个最好的结果, 推荐使用`ungroup`, 但需要的存储器和编译时间最多。

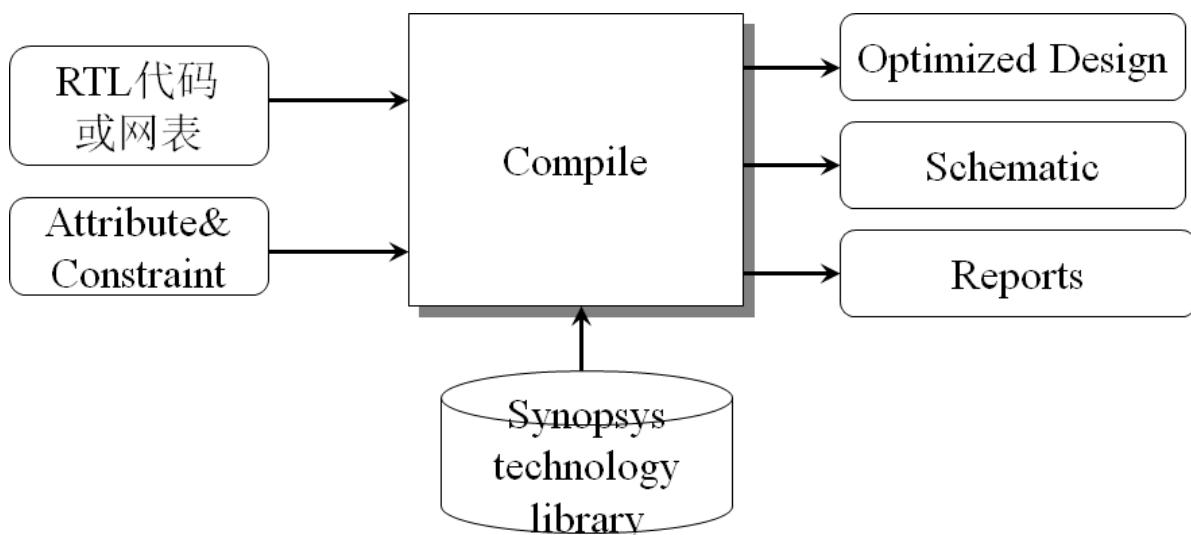
## 总结

- 设计实际的设计环境
  - 输入延时, 输出延时
  - 输入驱动强度, 输出负载
  - 工作条件
  - Wireload model
- 设置Design Rule约束(DRC)
  - Maximum fanout
  - Maximum transition time
  - Maximum capacitance
- **设置设计约束**
  - 最大延时, 最小延时
  - 说明时钟
  - 最大面积
  - False path, multi-cycle path和multi-frequency时钟
  - 动态功耗, 泄漏功耗
- 处理多实例化问题
  - **dont\_touch**
  - **ungroup**
  - **uniquify**
- 在编译设计前进行检查
- 保存设置文件并执行脚本文件

## 设计优化

### 设计编译

Compile: the “art”of Synthesis



## Architectural-Level Optimization

- 设计结构的选择(DesignWare)

- 数据通路的优化

- 共享子表达式

$$\begin{aligned} \text{Sum1} &= A + B + C; \\ \text{Sum2} &= A + B + D; \end{aligned}$$

- 资源共享

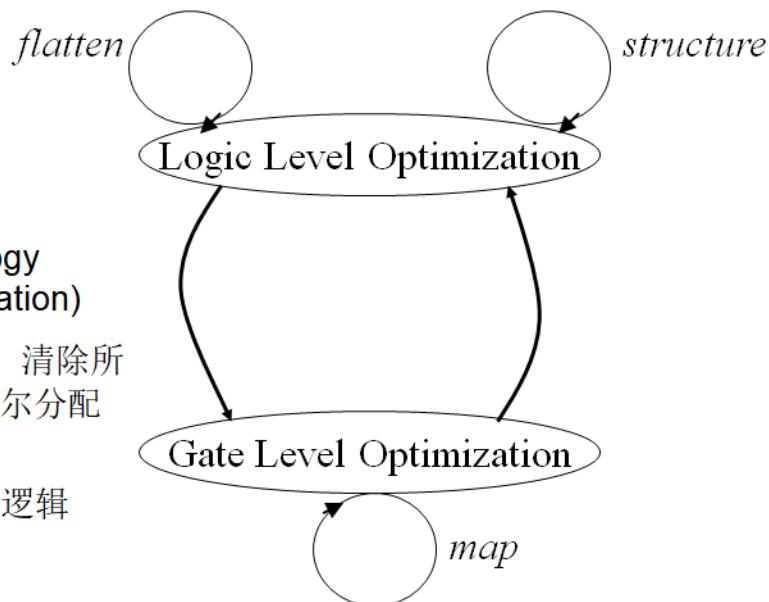
- 计算重新排序

```
if(a)
    out = b + c;
else
    out = b + d;
```

$$\begin{aligned} \text{Temp} &= A + B; \\
\text{Sum1} &= \text{Temp} + C \\
\text{Sum2} &= \text{Temp} + D; \end{aligned}$$

$$\text{Sum} = A * B + C * D + E + F + G;$$

$$\text{Sum} = E + F + G + A * B + C * D;$$



- 逻辑级优化(technology independent optimization)

– *flatten* (缺省为关)：清除所有中间变量，使用布尔分配定律去除所有括号。

– *structure* : 提取公用逻辑

- 门级优化

– *map* : 使设计与工艺相关

## 逻辑级优化(Logic-Level)

- 对电路的布尔表达式进行优化

- 对整个设计面积/速度特性有全局的影响

- 策略

– *structure*

– *Flatten*

– 如果两者都选用, 则设计先*flatten*后*structure*

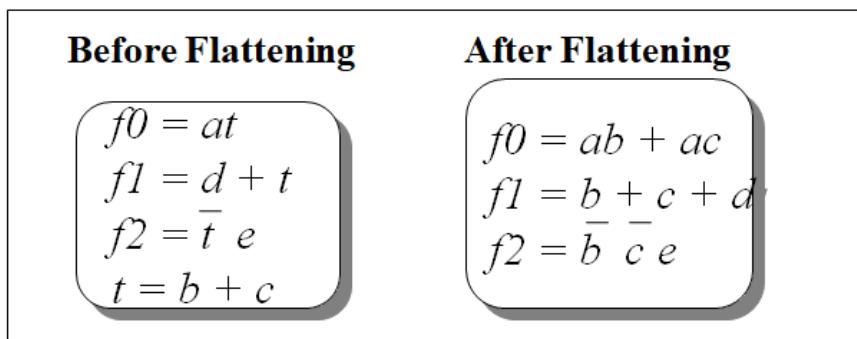
## structuring和flattening

### Structuring

- 提取共因式作为中间变量
- 时序驱动**structuring**: 在**structuring**过程中考虑时序约束并减小设计面积。
- 布尔优化(缺省为关)产生最可能小的设计。
- 适合于结构化电路

### flattening

- **Flatten**缺省为关
- 消除所有中间变量
- 结果为两级积-和式
- 适合于非结构化电路



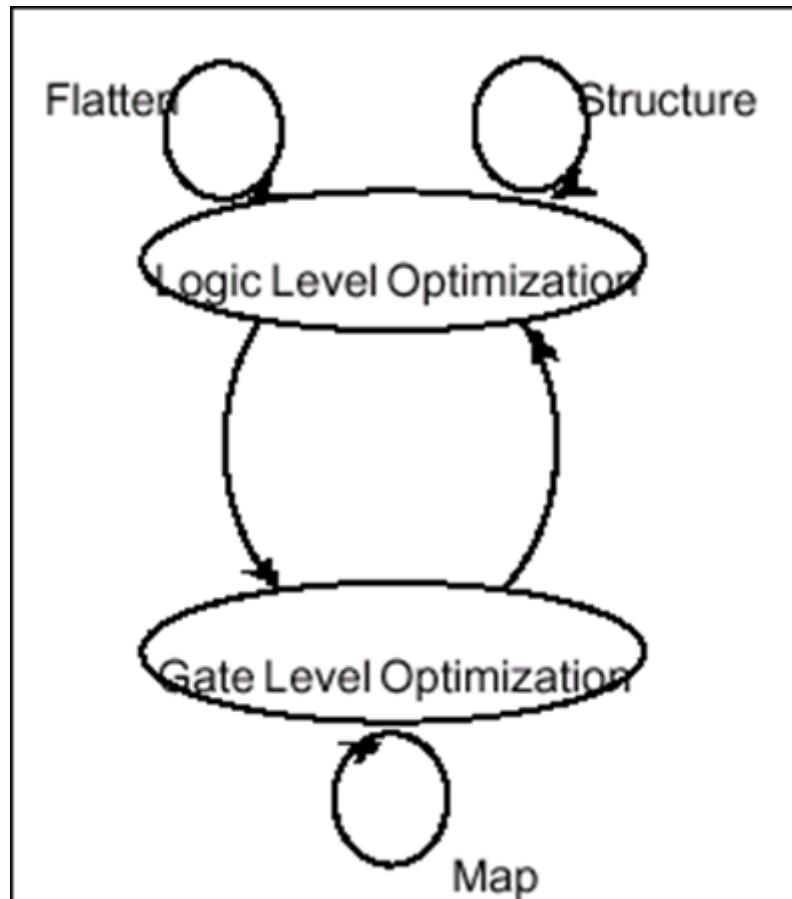
## 门级优化--映射

- 使用逻辑优化产生的结构
- 从工艺库中选择元件以实现这些逻辑结构，以满足电路指定的时间、设计规则和面积目标
- 局部影响设计的area/speed特性

Mapping flow

-寻找基本配置的门来实现逻辑结构

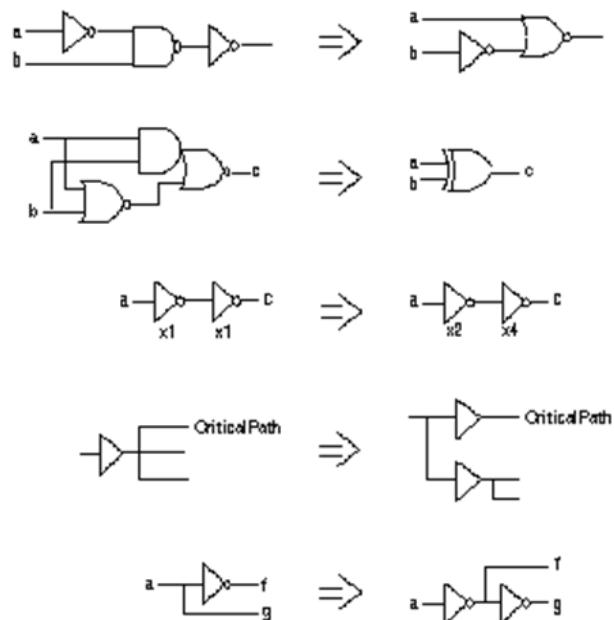
-从逻辑上重新安排元件，以满足设计规则、面积、速度和功耗目标。



## 组合电路映射

### 组合逻辑映射

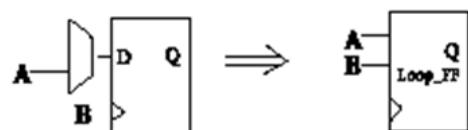
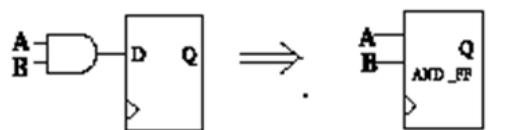
- 映射重新安排元件，组合和重组合逻辑到不同的元件中
- 可以采用不同的算法，如 **cloning, resizing or buffering**
- 试图满足设计规则、约束以及时间/面积目标。



## 时序电路映射

## 时序逻辑映射

- 对到工艺库中元件的映射进行优化
- 分析时序单元周围的组织逻辑，看是否能吸收HDL中的逻辑属性。
- 利用可选的用户首选的时序单元的嵌入式描述。



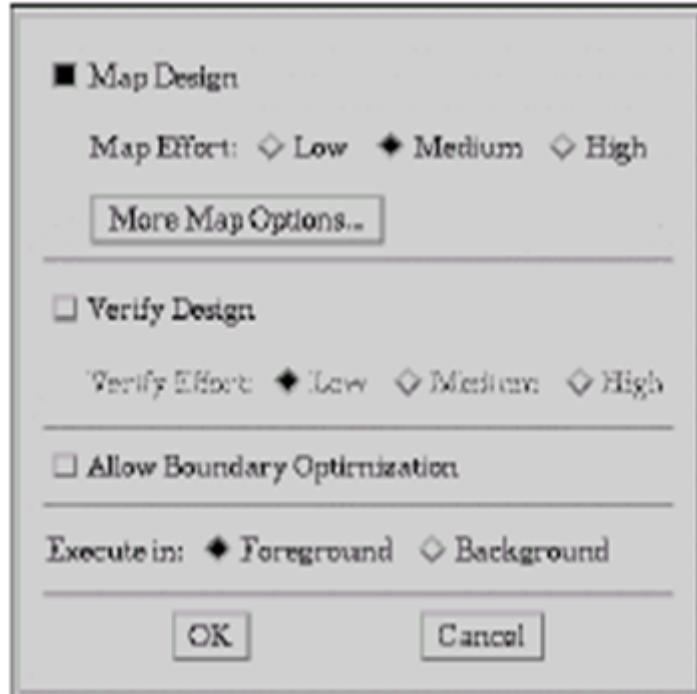
## 映射强度

- 有三种强度, *low*, *medium*, *high*, 决定编译的映射过程所花费的CPU时间的相对数
  - low* - 较快的综合, 不做所有算法
  - medium* - 缺省, 对大多数设计是足够的
  - high* - 将关键路径重新综合, 但会花费较多的CPU时间; 在有些情况下, 编译会陷入死循环。

- Note: 在Synopsys的编译过程

需要很长时间, 因此在进行编译

之前, 请确认设置的属性和约束。



## Compile Summary

### 逻辑级优化

- 优化电路的布尔表达式优化
- Flattening (off by default)

» 消除所有的结构

» 当设计输入少于20个时，关闭structure, 打开flattening

#### - Structuring

» 寻找共因子以减小面积

» 时序驱动的 (default)

» 布尔优化 (area optimization only)

#### 门级优化

##### - Mapping ( medium effort is default )

» 从工艺库中选择元件

» 使用逻辑级优化产生的结构

#### 层次化编译技术

##### - top-down, bottom up and characterize

##### - Timing budgeting

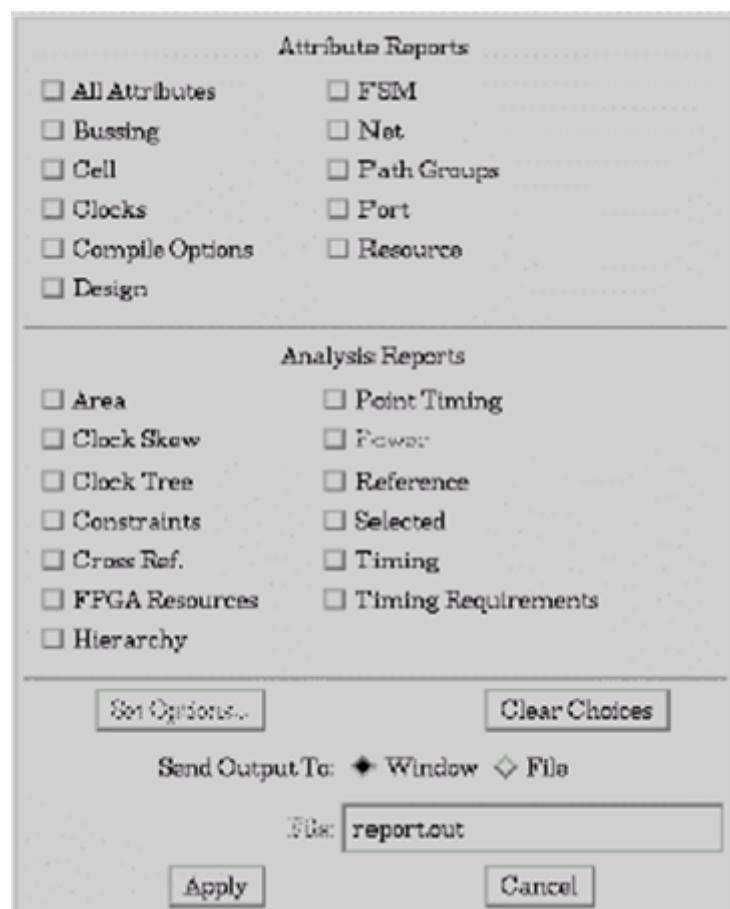
» Design Budgeting

» Automatic Chip Synthesis (ACS)

## Synthesis Report and Analysis

- Analysis/Report

- 通过报告及分析，可以查看设置的属性及优化后的结果



## Report We will Generate

- 属性报告

-所有的属性、clock、端口、设计及net

- 分析报告

-面积、层次、约束、时间、节点时间

## 网线(net) Report

- 网线报告显示每一条net的静态结果

The screenshot shows a software interface for generating a net report. At the top, it displays the report parameters: Report : net, Design : top, Version: v3.4b, and Date : Thu Jan 30 09:40:44 1997. Below this, operating conditions are specified: Operating Conditions: wcom Library: cb60hp230d\_typ and Wire Loading Model Mode: top. The main area contains a table with columns: Net, Fanout, Fanin, Load, Resistance, Pins, and Attributes. The table lists several nets, each with its corresponding values. A column header 'Load' is circled in red. At the bottom of the table are buttons for Show, Next, Previous, and Cancel.

Net	Fanout	Fanin	Load	Resistance	Pins	Attributes
clk	19	1	0.74	0.00	20	
cook	6	1	0.27	0.00	7	
cook_time[0]	1	1	0.07	0.00	2	
cook_time[1]	1	1	0.07	0.00	2	
cook_time[2]	1	1	0.07	0.00	2	
cook_time[3]	1	1	0.06	0.00	2	

## 端口(port) Report

- dc\_shell命令

report\_port -verbose { port\_list }

或在选项菜单中选择verbose (详细)

The screenshot shows two side-by-side command windows. The left window is titled 'Command Window' and displays a 'Port Report' with the following parameters: Report : port, -verbose, Design : top, Version: v3.4b, and Date : Fri Jan 31 03:07 1997. It includes a table of port characteristics with columns: Port, Dir, Pin, Wire, Max, Min, Connection, Class, and Attrs. The right window is also titled 'Command Window' and displays a 'Pin Drive' report with columns: Input Port, Pin Drive, Wire Drive, Min Trans, Min Cap, and Min Fanout. Both windows have a status bar at the bottom indicating they are 'design\_analyzer>'.

Port	Dir	Pin	Wire	Max	Min	Connection	Class	Attrs
sec_msb_led[6]	out	0.0190	0.0000	--	--	--		
sec_msb_led[5]	out	0.0190	0.0000	--	--	--		
sec_msb_led[4]	out	0.0190	0.0000	--	--	--		
sec_msb_led[3]	out	0.0190	0.0000	--	--	--		
sec_msb_led[2]	out	0.0190	0.0000	--	--	--		
sec_msb_led[1]	out	0.0190	0.0000	--	--	--		
sec_msb_led[0]	out	0.0190	0.0000	--	--	--		
cook_time[15]	in	0.0000	0.0000	--	--	--		
cook_time[14]	in	0.0000	0.0000	--	--	--		
cook_time[13]	in	0.0000	0.0000	--	--	--		
cook_time[12]	in	0.0000	0.0000	--	--	--		
cook_time[11]	in	0.0000	0.0000	--	--	--		
cook_time[10]	in	0.0000	0.0000	--	--	--		
cook_time[9]	in	0.0000	0.0000	--	--	--		
cook_time[8]	in	0.0000	0.0000	--	--	--		
cook_time[7]	in	0.0000	0.0000	--	--	--		
cook_time[6]	in	0.0000	0.0000	--	--	--		

## 面积(area) Report

- 面积报告显示设计的门数

```
design_analyzer>*****
Report : area
Design : top
Version: v3.4b
Date   : Thu Jan 30 09:42:31 1997
*****  
  
Library(s) Used:
  cb60hp230d_typ (File: /raid/caid09/hsieh/lib/LIB06opdm/Synopoya/cb60hp230d_typ)  
  
Number of ports:          49
Number of nets:           84
Number of cells:          3
Number of references:     3  
  
Combinational area:      502.000000
Noncombinational area:    138.250000
Net Interconnect area:   1.626000  
  
Total cell area:         640.850000
Total area:               641.875977  
  
1
design_analyzer>
```

是指设计大  
约为600门

## 层次(hierarchy) Report

- 层次报告显示每块的使用的元件及其层次

```
*****  
Report : hierarchy
Design : top
Version: v3.4b
Date   : Thu Jan 30 09:47:52 1997
*****  
  
top
  display
    converter_0
      an02d1          cb60hp230d_typ
      fa05d1          cb60hp230d_typ
      in01d0          cb60hp230d_typ
      in01d1          cb60hp230d_typ
      in02d1          cb60hp230d_typ
      ni21d1          cb60hp230d_typ
      rx21d1          cb60hp230d_typ
      nd02d0          cb60hp230d_typ
      nd03d0          cb60hp230d_typ
      nd04d0          cb60hp230d_typ
      ni01d1          cb60hp230d_typ
      nr02d0          cb60hp230d_typ
      nr02d2          cb60hp230d_typ
      nr03d1          cb60hp230d_typ
```

## 引用(reference) Report

- 引用报告显示设计中引用的静态结果

Reference	Library	Unit Area	Count	Total Area	Attributes
display		208.500000	1	208.500000	b, n
microwave		96.750000	1	96.750000	b, n
timer		335.000000	1	335.000000	b, n
Total 3 references					640.250000
Show	Next	Previous	Cancel		

## 约束(constraint) Report

- 约束报告显示编译过的设计是否满足约束
- dc\_shell> report\_constraint -all\_violators

clk	0.00	1.00	0.00
default	0.00	1.00	0.00
<hr/>			
max_delay/setup			
Group (min_delay/hold)	Cost	Weight	Weighted Cost
clk (no fix_hold)	0.00	1.00	0.00
default	0.00	1.00	0.00
<hr/>			
min_delay/hold			
<hr/>			
Constraint	<hr/>		
max_delay/setup	0.00 (MET)		
max_transition	0.00 (MET)		
max_fanout	0.00 (MET)		
max_area	41.88 (VIOLATED)		
<hr/>			
Test Constraint	<hr/>		
min_fault_coverage	None (UNKNOWN)		
Show	Next	Previous	Cancel

## 时序(timing) Report

- 用户应知道report\_timing产生的路径信息

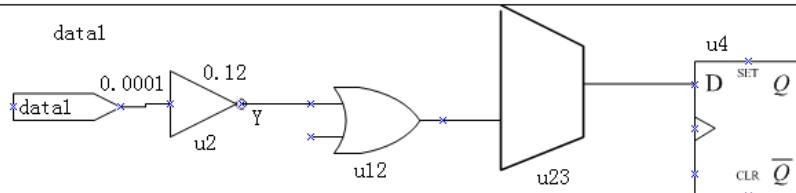
The screenshot shows the 'Report Output' window from a design analyzer. The window displays the following information:

```
design_analyzer>
*****
Report: timing
  -path short
  -delay max
  -max_paths 1
Design : pmult32
Version: 1999.10-5
Date : Wed Nov 15 09:46:06 2000
*****
Operating Conditions: NCCOM Library: cb35os142
Wire Load Model Mode: enclosed
*****
Startpoint: A2_reg[5] (rising edge-triggered flip-flop clocked by clock)
Endpoint: out_reg[55]
  (rising edge-triggered flip-flop clocked by clock)
Path Group: clock
Path Type: max
*****
Des/Clust/Port      Wire Load Model      Library
pmult32          22000                  cb35os142
pmult32_lv_bthenc_21
```

At the bottom of the window are buttons for 'Show', 'Next', 'Previous', and 'Cancel'.

- 路径延时、路径要求时间和总结部分

Point	Incr	path
Clock clk (rise edge)	0.00	0.00
Clock network delay(ideal)	0.00	0.00
Input external delay	1.00	1.00 f
Data1(in)	0.00	1.00 f
U2/Y(inv1)	0.12	1.12 r
U12/Y(or2x1)	0.26	1.38 r
U23/Y(mx2d2)	0.23	1.61 f
U4/D(dffx1)	0.00	1.61 f
<b>Data arrival time</b>		<b>1.61</b>



## 时序(timing) 报告

- 时序报告显示设计的最大或最小路径延时，缺省时显示一个最大延时路径。

<b>Point</b>	<b>Incr</b>	<b>path</b>
<b>Clock clk (rise edge)</b>	<b>5.00</b>	<b>5.00</b>
<b>Clock network delay(ideal)</b>	<b>0.00</b>	<b>5.00</b>
<b>U4/clk(dffx1)</b>	<b>0.00</b>	<b>5.00</b>
<b>Library setup time</b>	<b>-0.19</b>	<b>4.81</b>
<b>Data required time</b>		<b>4.81</b>
<b>Data required time</b>		<b>4.81</b>
<b>Data arrival time</b>		<b>-1.61</b>
<b>Slack(MET)</b>		<b>3.20</b>

## What is slack?

- slack是要求时间与实际到达时间的差值结果。
- slack为正或0表示约束满足
- slack为负表示约束没有得到满足。

## Report Power

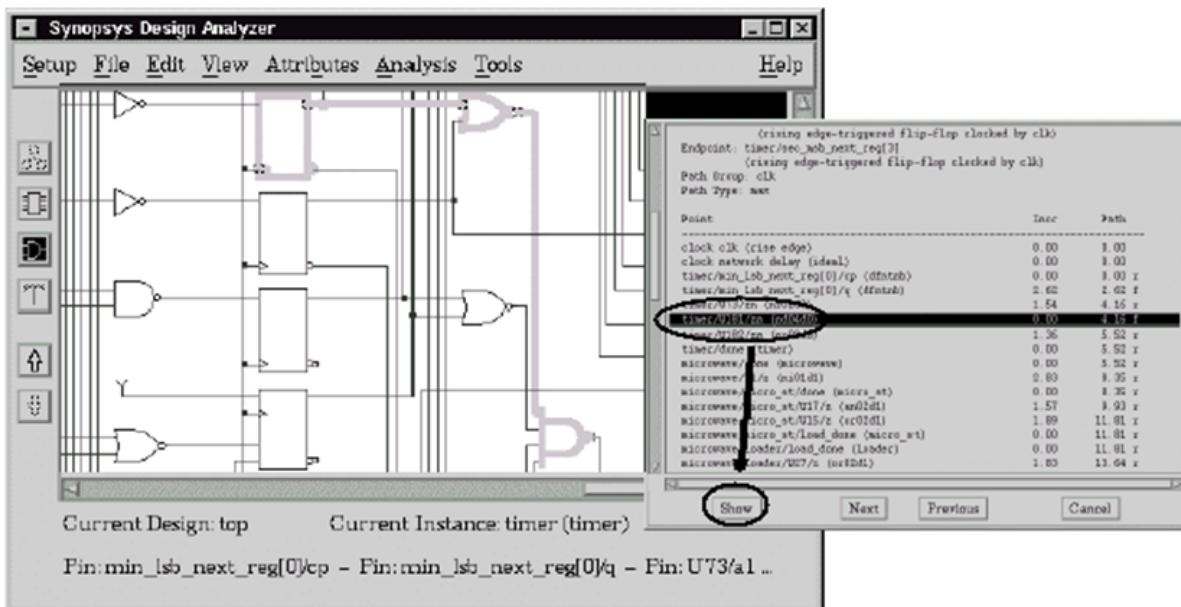
- dc\_shell> report\_power

```
Global Operating Voltage = 2.5
Power-specific unit information :
  Voltage Units = 1V
  Capacitance Units = 1.000000pf
  Time Units = 1ns

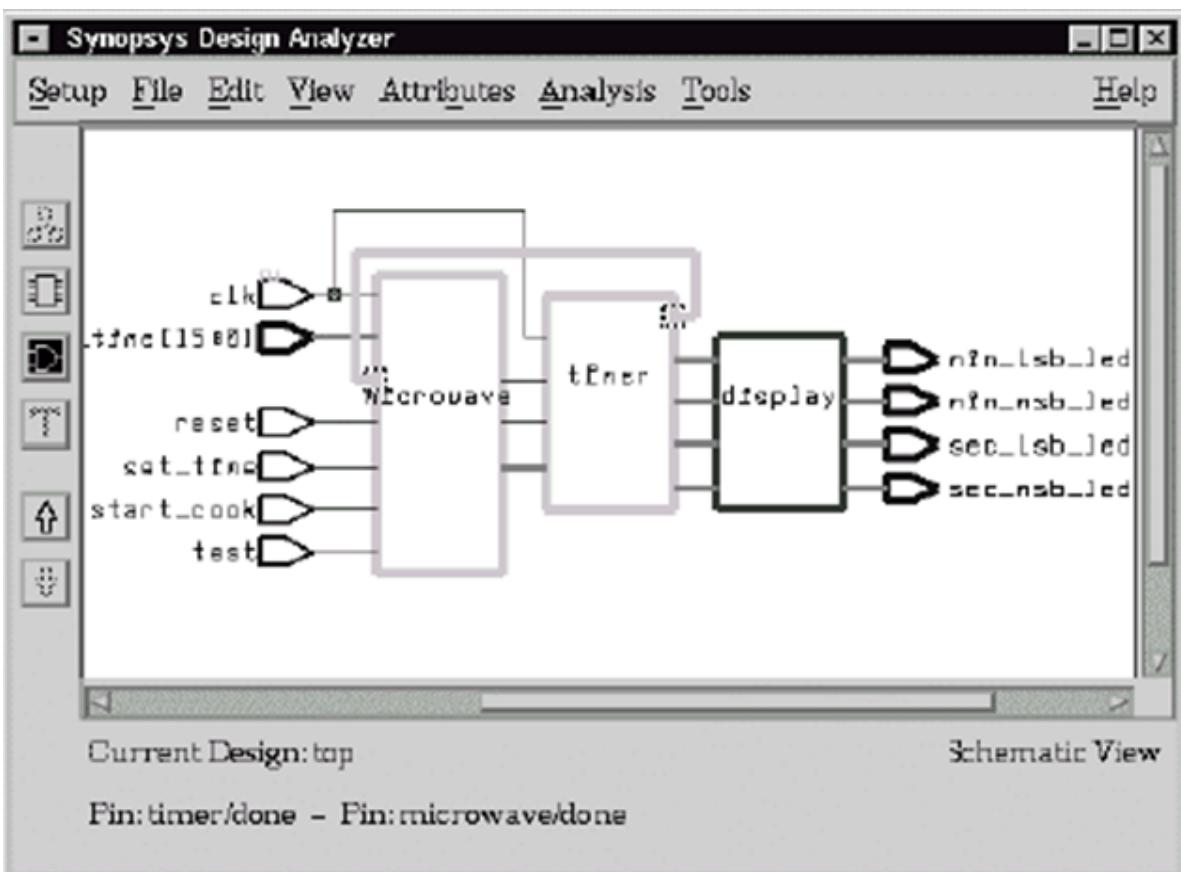
  Dynamic Power Units = 1mW (derived from V,C,T units)
  Leakage Power Units = 1nW
  Cell Internal Power = 13.4819 mW (58%)
  Net Switching Power = 9.7291 mW (42%)
  Total Dynamic Power = 23.2109 mW (100%)
  Cell Leakage Power = 123.8077 nW
```

## 交互式显示

- 从报告中选择一条信息，相应的项会在逻辑图中高亮度显示(按show按钮)

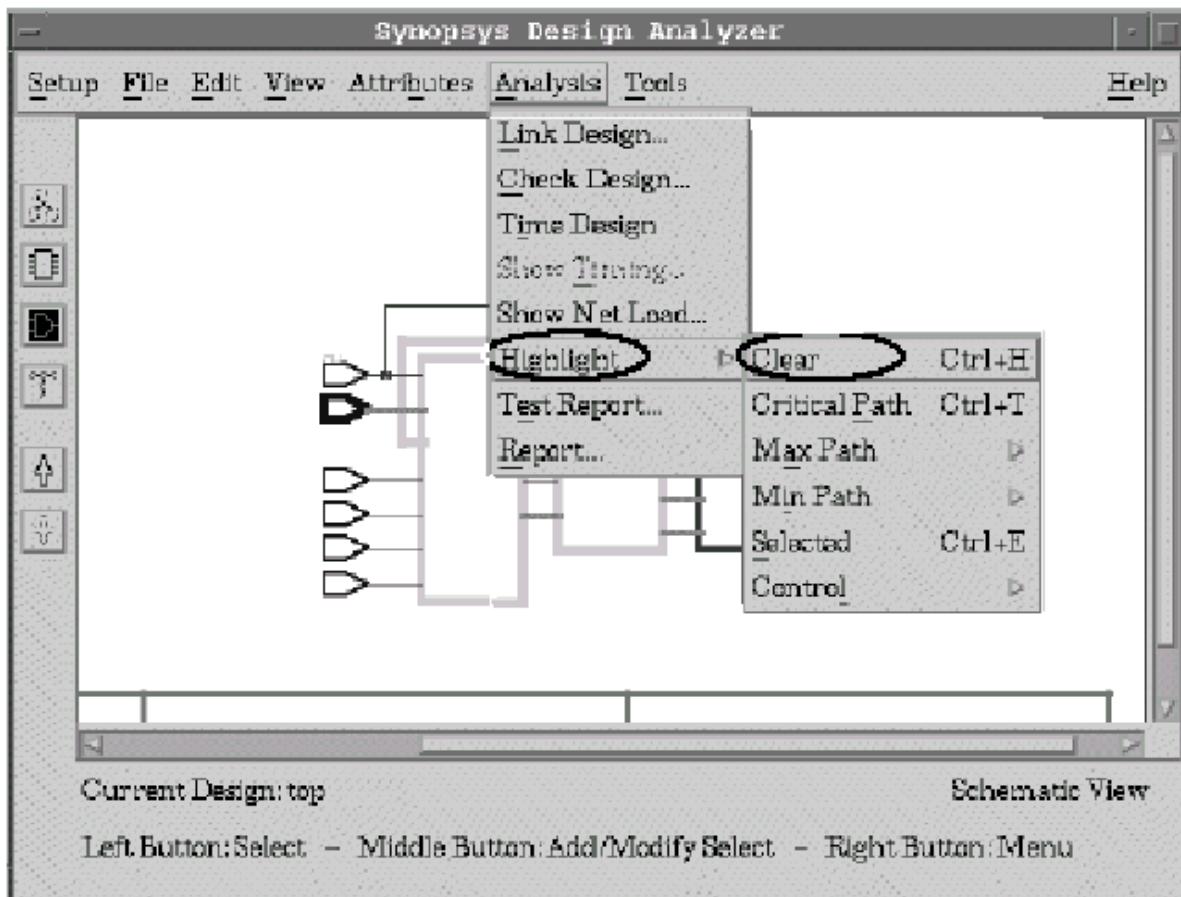


- 高亮度显示可以穿过层次



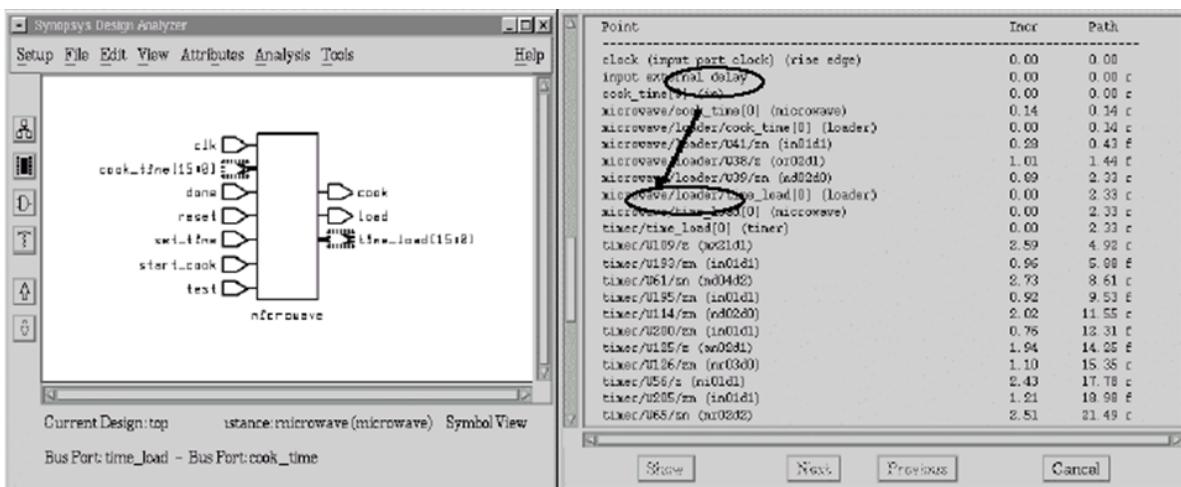
## Highlight

- 显示最长和最短路径的另一种方法是：Analysis/Highlight



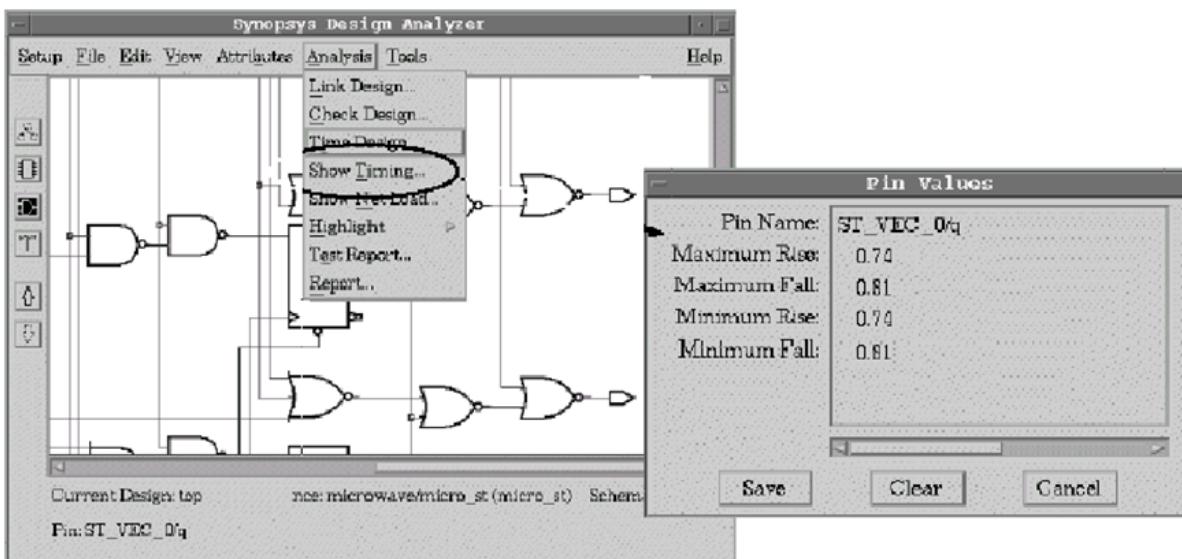
## Point Timing Report

- Point timing report 显示逻辑图中两个所选的点之间的时序信息。
- Analysis/report @point timing



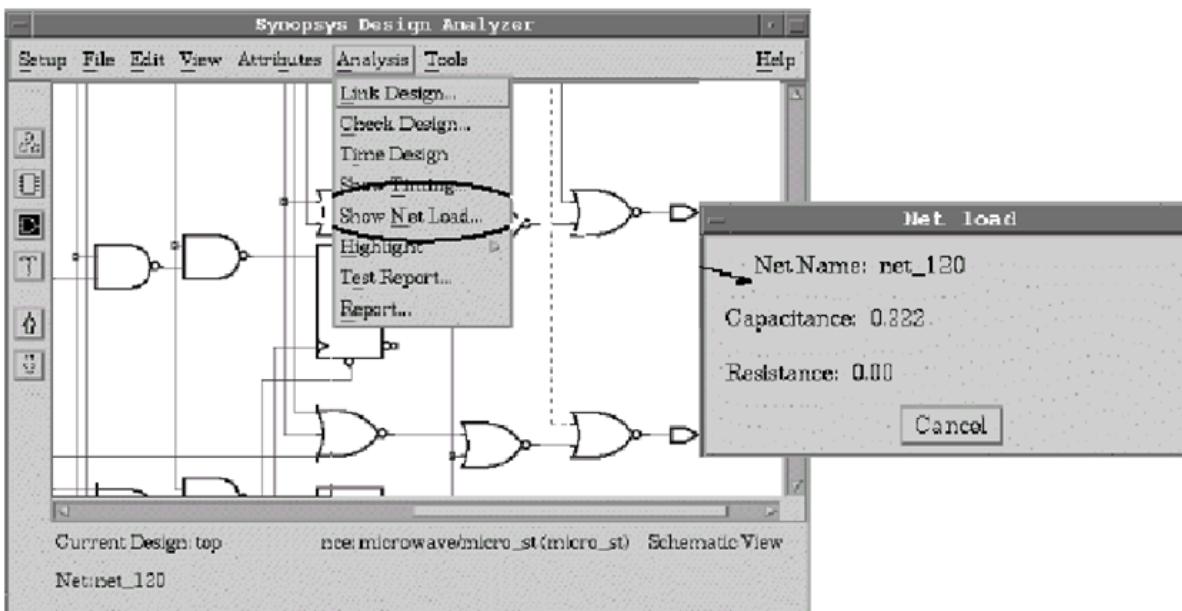
## 用逻辑图分析电路

- 确定逻辑图中所选的引脚上的信号的到达时间



- 确定在逻辑图中选择的网线的负载

#### *Analysis>Show Net Load*



## 保存设计

- 在退出Design Analyzer之前，将设计保存到文件
- *File / Save* 以DB格式保存设计
- *File / Save as* 可以其它的输出格式保存设计

-Synopsys formats

» equation: .eq

» state table: .st

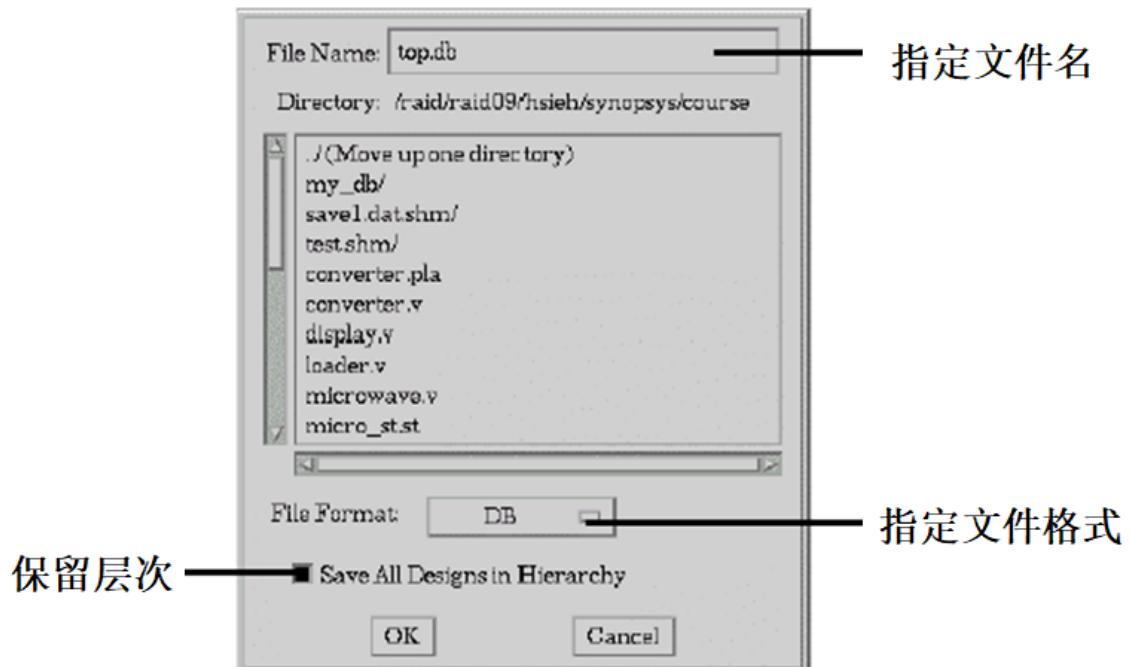
-Verilog: .v

-VHDL: .vhd

-PLA( Berkeley Espresso): .pla

-EDIF

- *File / Save as*



- 将文件保存为Verilog格式，进行门级仿真，并用Verilog in接口将其转换为OPU数据库用来布局布线
- 如果不能 Verilog in，请检查assign问题
- 如果存在任何assign问题，选择这个块并用下面的dc\_shell命令修复：

-set\_fix\_multiple\_port\_nets -all -buffer\_constants

-compile -map\_effort medium

### Fix multiple Port Net

- 清除verilog assignment问题

-set\_fix\_multiple\_port\_nets -all -buffer\_constants

-compile -map\_effort medium

```
assign \A[19] = A[19];
assign \A[18] = A[18];
assign \A[17] = A[17];
assign \A[16] = A[16];
assign \A[15] = A[15];
assign ABSVAL[19] = \A[19];
assign ABSVAL[18] = \A[18];
assign ABSVAL[17] = \A[17];
assign ABSVAL[16] = \A[16];
assign ABSVAL[15] = \A[15];
```



```
buffda X37X (.I(A[19]), .Z(ABSVAL[19]));
buffd1 X38X (.I(A[18]), .Z(ABSVAL[18]));
buffd1 X39X (.I(A[17]), .Z(ABSVAL[17]));
buffd1 X40X (.I(A[16]), .Z(ABSVAL[16]));
buffd1 X41X (.I(A[15]), .Z(ABSVAL[15]));
```

### 门级仿真 (verilog)

- 输出门级网表（有两种方法）

1.File / Save As ® Verilog ( for File format)

2.dc\_shell> write -format verilog -hierarchy -output chip.vg

- 产生SDF（有两种方法）

1.File /Save info ® Design timing

2.dc\_shell> write\_sdf -version 1.0 -context verilogchip.sdf

- 修改testfixture文件

```
$sdf_annotate("the_SDF_file_name",
    the_top_level_module_instance_name);
```

For example: \$sdf\_annotate("chip.sdf", top);

- 用Verilog-XL进行仿真

©北京大学 [JackHCC](#)