

数字集成电路设计入门

--从HDL到版图

于敦山

北大微电子学系

课程内容(一)

- 介绍Verilog HDL, 内容包括:
 - Verilog应用
 - Verilog语言的构成元素
 - 结构级描述及仿真
 - 行为级描述及仿真
 - 延时的特点及说明
 - 介绍Verilog testbench
 - 激励和控制描述
 - 结果的产生及验证
 - 任务task及函数function
 - 用户定义的基本单元(primitive)
 - 可综合的Verilog描述风格

课程内容(二)

- 介绍Cadence Verilog仿真器, 内容包括:
 - 设计的编译及仿真
 - 源库(source libraries)的使用
 - 用Verilog-XL命令行界面进行调试
 - 用NC Verilog Tcl界面进行调试
 - 图形用户界面(GUI)调试
 - 延时的计算及反标注(annotation)
 - 性能仿真描述
 - 如何使用NC Verilog仿真器进行编译及仿真
 - 如何将设计环境传送给NC Verilog
 - 周期(cycle)仿真

课程内容(三)

- 逻辑综合的介绍
 - 简介
 - 设计对象
 - 静态时序分析 (STA)
 - **design analyzer**环境
 - 可综合的**HDL**编码风格
- 可综合的**Verilog HDL**
 - **Verilog HDL**中的一些窍门
 - **Designware**库
 - 综合划分
- 实验 (1)

课程内容(四)

- 设计约束 (Constraint)
 - 设置设计环境
 - 设置设计约束
- 设计优化
 - 设计编译
 - FSM的优化
- 产生并分析报告
- 实验 (2)

课程内容(五)

- 自动布局布线工具(Silicon Ensemble)简介

课程安排

- 共54学时 (18)
- 讲课, 27学时
 - Verilog (5)
 - Synthesis (3)
 - Place &Route (1)
- 实验, 24学时
 - Verilog (5)
 - Synthesis (2)
 - Place &Route (1)
- 考试, 3学时

参考书目

- Cadence Verilog Language and Simulation
- Verilog-XL Simulation with Synthesis
- Envisia Ambit Synthesis
- 《硬件描述语言Verilog》 清华大学出版社, Thomas & Moorby, 刘明业等译, 2001.8

第二章 Verilog 应用

- 学习内容
 - 使用HDL设计的先进性
 - Verilog的主要用途
 - Verilog的历史
 - 如何从抽象级(levels of abstraction)理解
 - 电路设计
 - Verilog描述

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: 描述风格的详细程度, 如行为级和门级
- **ASIC**: 专用集成电路(Application Specific Integrated Circuit)
- **ASIC Vender**: 芯片制造商, 开发并提供单元库
- 自下而上的设计流程: 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。
- 自顶向下的设计流程: 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级**: 寄存器传输级(Register Transfer Level), 用于设计的可综合的一种抽象级
- **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
 - 在不同抽象级上描述电路
 - 描述电路的时序
 - 表达具有并行性
- 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。

为什么使用HDL

- 使用**HDL**描述设计具有下列优点：
 - 设计在高层次进行，与具体实现无关
 - 设计开发更加容易
 - 早在设计期间就能发现问题
 - 能够自动的将高级描述映射到具体工艺实现
 - 在具体实现时才做出某些决定
- **HDL**具有更大的灵活性
 - 可重用
 - 可以选择工具及生产厂
- **HDL**能够利用先进的软件
 - 更快的输入
 - 易于管理

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。

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
 - 高抽象级系统仿真进行系统结构开发
 - 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)

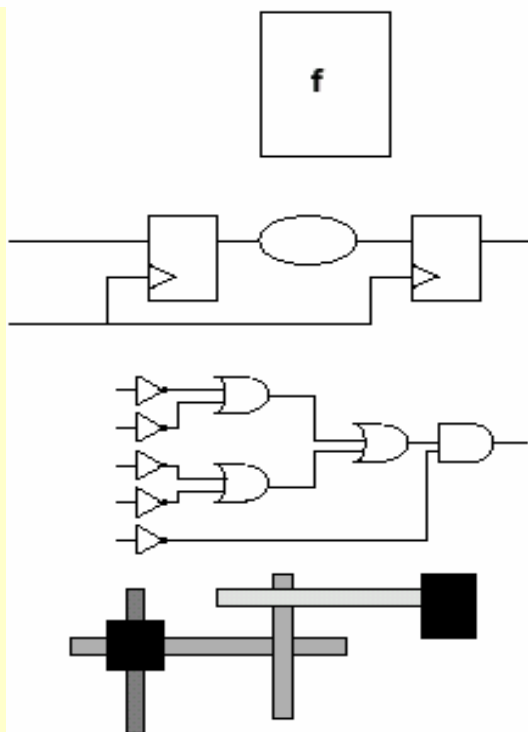
- Verilog既是一种行为描述的语言也是一种结构描述语言。Verilog模型可以是实际电路的不同级别的抽象。这些抽象的级别包括：

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



行为综合



综合前仿真



逻辑综合



综合后仿真



版图

抽象级(Levels of Abstraction)

- 在抽象级上需要进行折衷



抽象级(Levels of Abstraction)

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。
 - 在综合时用特定工艺和低层元件将**RTL**描述映射到门级网表

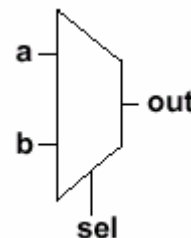
抽象级(Levels of Abstraction)

- 设计工程师在不同的设计阶段采用不同的抽象级
 - 首先在行为级描述各功能块，以降低描述难度，提高仿真速度。
 - 在综合前将各功能模块进行**RTL**级描述。
 - 用于综合的库中的大多数单元采用结构级描述。在本教程中的结构级描述部分将对结构级(门级)描述进行更详细的说明。
- Verilog还有一定的晶体管级描述能力及算法级描述能力

行为级和RTL级

- MUX的行为可以描述为：只要信号a或b或sel发生变化，如果sel为0则选择a输出；否则选择b输出。

```
module muxtwo (out, a, b,  
sel);  
  input a, b, sel;  
  output out; reg out;  
  always @( sel or a or b)  
    if (! sel) out = a;  
    else    out = b;  
endmodule
```



这个行为级RTL描述不处理X和Z状态输入，并且没有延时。

在行为级模型中，逻辑功能描述采用高级语言结构，如@, while, wait, if, case。

Testbench(test fixture)通常采用行为级描述。所有行为级结构在testbench描述中都可以采用。

RTL模型中数据流都是基于时钟的。任何时钟元件在时钟沿处的行为都要精确描述。RTL级描述是行为级Verilog的子集。

结构级描述

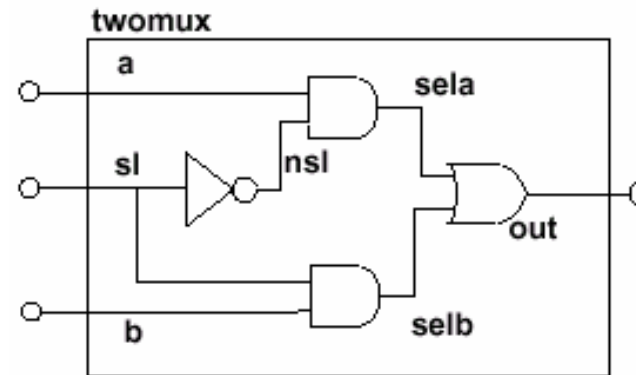
综合不支持!

结构级Verilog适合开发小规模元件，如ASIC和FPGA的单元

- Verilog内部带有描述基本逻辑功能的基本单元(primitive)，如and门。
- 用户可以定义自己的基本单元UDP(User Defined Primitives)
- 综合产生的结果网表通常是结构级的。用户可以用结构级描述粘接(glue)逻辑。

- 下面是MUX的结构级描述，采用Verilog基本单元(门)描述。描述中含有传输延时。

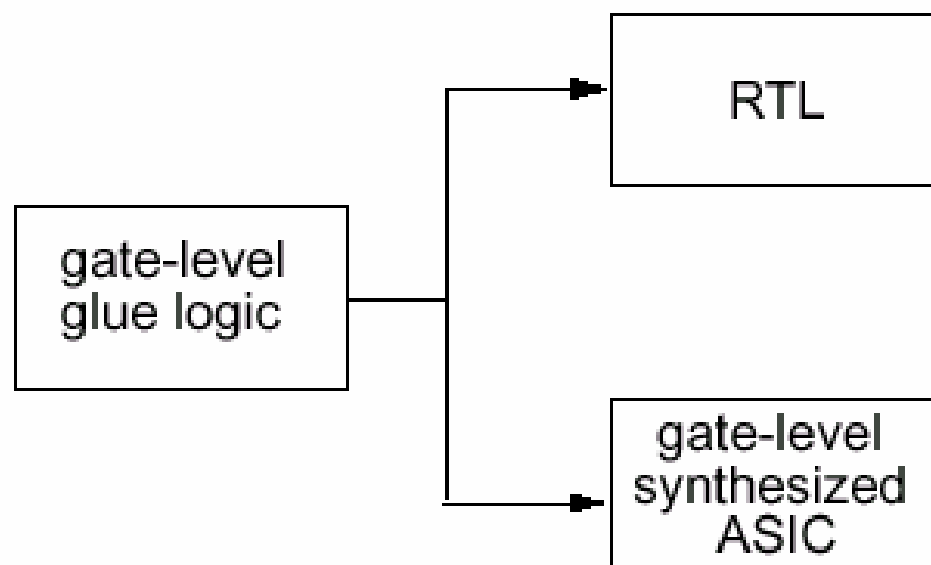
```
module twomux (out, a, b, sl);  
  input a, b, sl;  
  output out;  
  not u1 (nsl, sl );  
  and #1 u2 (sela, a, nsl);  
  and #1 u3 (selb, b, sl);  
  or #2 u4 (out, sela, selb);  
endmodule
```



仅需一种语言

- Verilog的一个主要特点是可应用于各种抽象级。建模时可采用门级和RTL级混合描述，在开发testfixture时可以采用行为级描述。

Top-level behavioral test bench



复习

- 什么是**Verilog** ?
- **Verilog**是公开的吗?
- 设计时什么时候采用**Verilog RTL**级描述?
- **Verilog**适合做什么样的设计?

解答:

- **Verilog**是用于硬件描述的具有时间概念的并行编程语言
- **Verilog**是一种公开语言， 由**OVI**负责组织， 有**IEEE1394**标准
- **RTL**描述用于综合， 或用于必须精确到每个时钟周期的模型的建模。
- **Verilog**适用于各种抽象级模型的开发及验证

第三章 Cadence仿真器

- 学习内容
 - 逻辑仿真算法
 - 如何启动**Verilog-XL**和**NC Verilog**仿真器
 - 如何显示波形

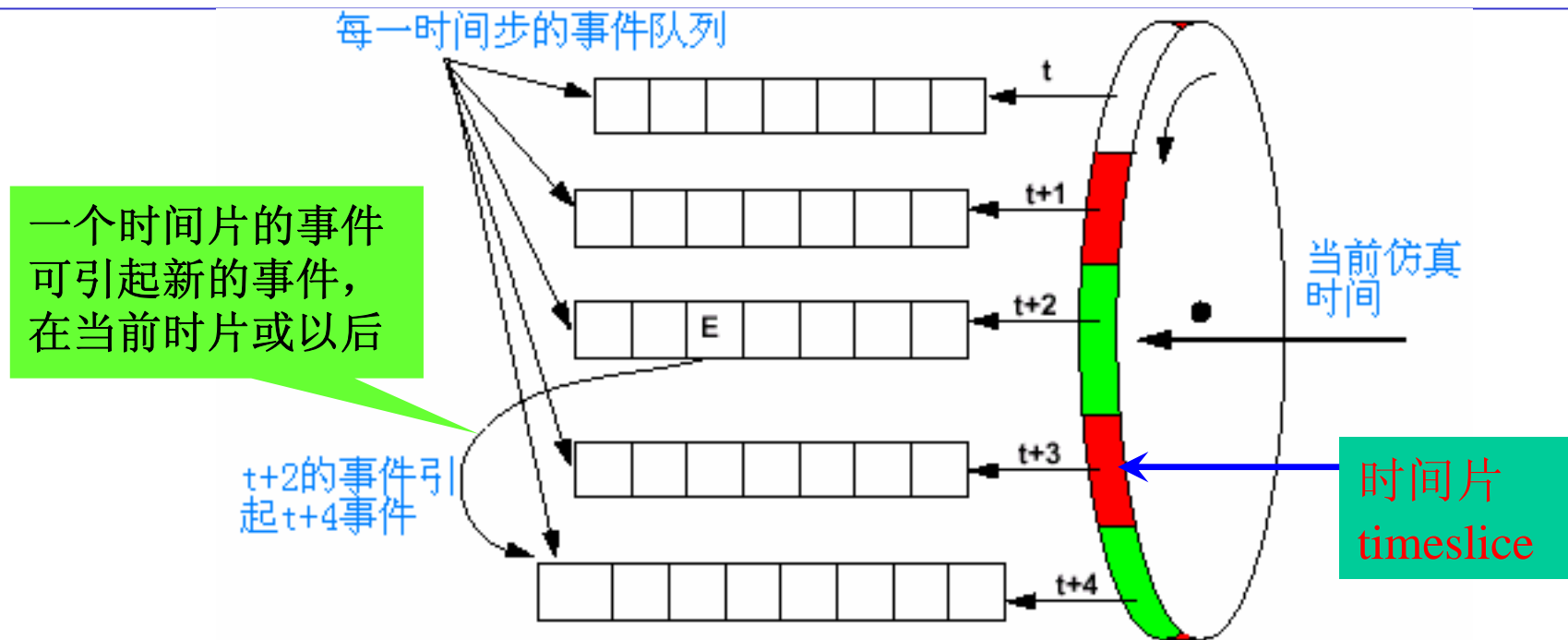
仿真算法

- 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。

Cadence Verilog仿真器

- **Verilog-XL和NC Verilog仿真器都是基于事件算法的仿真器。仿真器读入Verilog HDL描述并进行仿真以反映实际硬件的行为。**
- **Verilog-XL和NC Verilog仿真器遵循IEEE 1364 Verilog规范制定的基于事件的调度语义**
- 仿真器可用于
 - 确定想法的可行性
 - 用不同的方法解决设计问题
 - 功能验证
 - 确定设计错误

仿真过程

- Verilog仿真分下列步骤:

- 编译

- ✓ 读入设计描述，处理编译指导(**compiler directive**)，建立一个数据结构定义设计的层次结构
 - ✓ 这一步有时分为两步: **compilation, elaboration**

- 初始化

- ✓ 参数初始化；没有驱动的**Net**缺省值为**Z**；其它节点初始值为**X**。这些值延着设计层次传播。

- 仿真

- ✓ 刚开始时间为**0**时，仿真器将**initial**和**always**中的语句执行一次，遇到有时序控制时停止。这些赋值可产生在时间**0**或其后时间的事件。
 - ✓ 随着时间推进，被调度事件的执行引起更多的调度事件，直至仿真结束。

Versus 交互式编译仿真器

- **Verilog-XL**是一个交互式仿真器，过程如下：
 1. 读入**Verilog**描述，进行语义语法检查，处理编译指导(**compiler directive**)
 2. 在内存中将设计编译为中间格式，将所有模块和实例组装成层次结构(设计数据结构)。源代码中的每个元件都被重新表示并能在产生的数据结构 找到。
 3. 决定仿真的时间精度，在内存中构造一个事件队列的时间数据结构(时轮)。
 4. 读入、调度并根据事件执行每一个语句

Verilog-XL采用多种加速算法提高各种抽象级的仿真速度。

每次重新启动**Verilog-XL**，将重复上述步骤。

当进入交互模式时，可以输入**Verilog HDL**语句并加到设计的数据结构中。

Versus 交互式编译仿真

- **Verilog-XL**仿真器是与**Verilog HDL**同时开发的，因此它成为**Verilog HDL**仿真器的事实上的标准。
- **Verilog-XL**采用了多种加速算法，对每种抽象级描述都能很好的仿真。这些加速算法包括**Turbo**算法，**XL**算法及**Switch-XL**算法。在后面的教程中将对这些算法进行更为详尽的介绍。

NC Verilog-全编译仿真

- **NC Verilog**是全编译仿真器，它直接将**Verilog**代码编译为机器码执行。其过程为：
 - **ncvlog**编译**Verilog**源文件，按照编译指导(**compile directive**)检查语义及语法，产生中间数据。
 - **ncelab**按照设计指示构造设计的数据结构，产生可执行代码。除非对优化进行限制，否则源代码中的元件(**element**)可能被优化丢失。产生中间数据。
 - **ncsim**启动仿真核。核调入设计的数据结构，构造事件序列（时轮），调度并执行事件的机器码。有些事件可能消失(从不执行)除非限制优化过程。

编译后的所有代码的执行使用同一个核。

当重新启动仿真时，要对修改过的模块重新编译。省略这个手工过程的方法是直接对设计进行仿真，这将自动地对修改过的模块进行重新编译。

当采用交互模式时，可以使用**Tcl**命令和针对**NC Verilog**的**Tcl**扩展命令。

NC Verilog全编译仿真

- **NC Verilog**是最近才开发的，但其对描述的仿真与**Verilog-XL**完全相同
- **NC Verilog**仿真器用同一个核(**kernel**)对所有抽象级进行混合仿真，也就是说用户可以采用各种不同抽象级混合设计。但在门级仿真的效率差一些。
- **NC Verilog**仿真器对源代码采用增量编译方式，减少了编译时间。
- 在交互模式下，可以使用**Tcl**命令及其针对**NC Verilog**的扩展命令来修改设计和控制仿真。这将在后面进行详细描述。

对Verilog语言的支持

- **Verilog-XL和NC Verilog计划支持Verilog语言全集。**
用户可依据下列标准进行设计：
 - **IEEE1364-1995 Verilog语言参考手册**
 - **OVI 2.0 Verilog语言参考手册，但不支持：**
 - ✓ **Attributes: Verilog描述中对象的属性。**
 - ✓ **函数中output或inout变元(argument): OVI2.0允许函数中output和inout变元值能够返回。**

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 

<pre>mux.v test.v -c</pre>

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

启动NC Verilog

- 虽然NC Verilog仿真过程包括三个分立的步骤(ncvlog, ncelab, ncsim), 但仿真时不需要三个命令, 可以用带有命令行参数的ncverilog命令启动NC Verilog:

ncverilog [ncverilog_options] verilog-xl_arguments

Examples:

- ✓ ncverilog mux.v test.v
- ✓ ncverilog -c mux.v test.v
- ✓ ncverilog -f run.f

run.f文件的内容 →

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

NC Verilog将所有终端输出保存到名为ncverilog.log的文件

NC Verilog有什么不同？

NC Verilog为编译的元件及其它文件建立一个库结构。增量编译依赖于源文件、**SDF**文件和命令行参数。

- 除**+gui**、**-q**和**-s**这些只影响运行时间的参数外，其它任何命令行参数的改变将使设计重新编译、**elaborate**及仿真。
- 如果更新了源文件及仿真时用到的**SDF**文件，则与它们相关的文件将重新编译，设计也将重新**elaborate**和仿真。

ncverilog还有其它一些命令行参数，如

- 在调试时有完全的读、写及连接操作，用
+access + argument
ncverilog -f run.f + access+RWC
- 要得到源文件行操作能力，用**+linedebug**
ncverilog -f run.f +linedebug
- 强制重编译所有设计单元，使用**+nouupdate**

NC Verilog有什么不同？

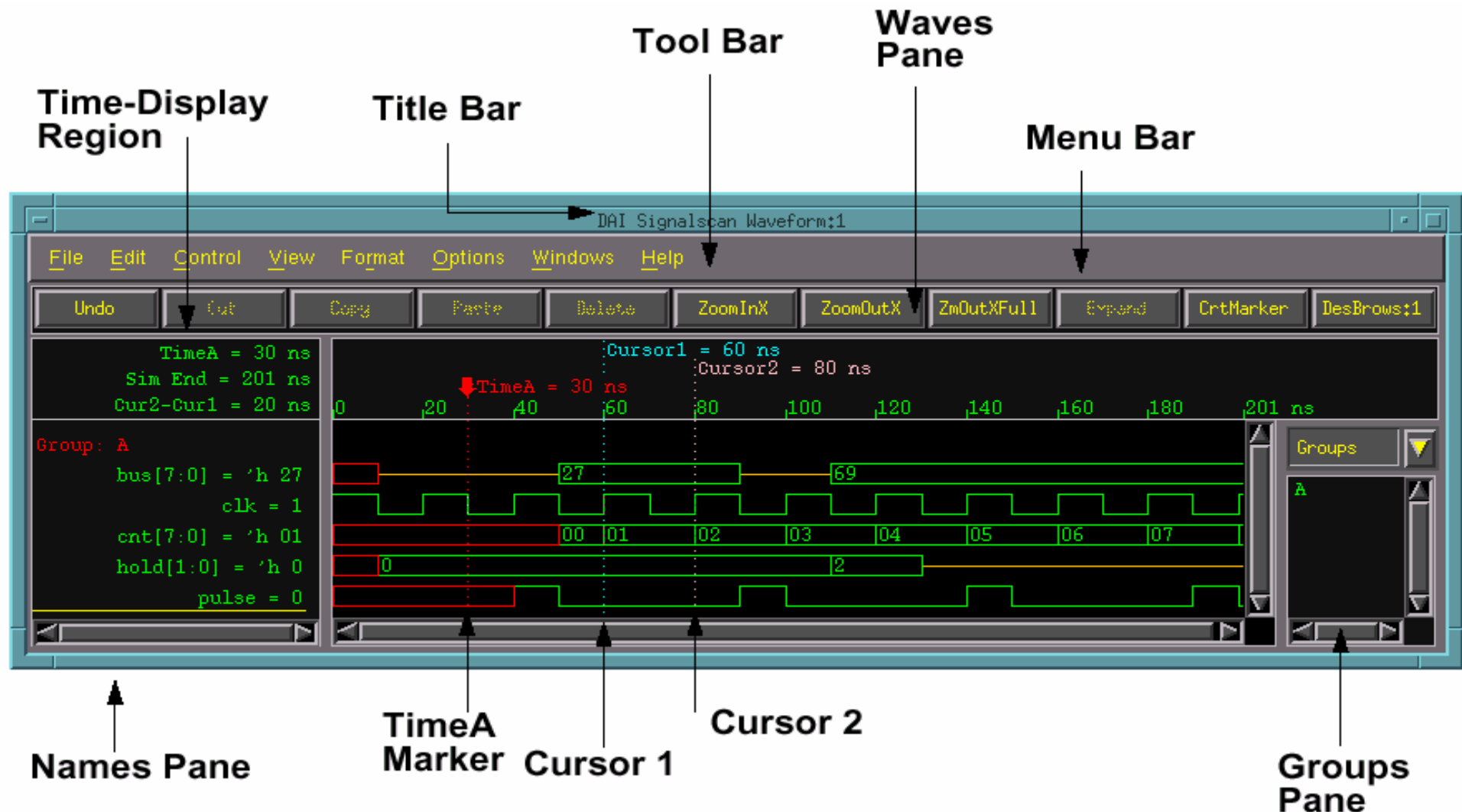
使用**+access**选项可以设置对所有对象的缺省操作。对象的缺省设置是无操作。用**+access+<args>**打开操作，**+access-<args>**关掉操作。**args**可以是R、W、C的任何组合。使用**+linedebug**可以打开R、W、C，同时可对源文件行进行操作，如在行上设置一断点。

使用**+nouupdate**强制重编译整个设计。缺省时只重新编译修改过的文件。只有当库可能被破坏时才这样做。

+gui选项启动图形界面；**-q**选项抑制标识信息；**-s**选项使仿真器在时间0时停止，进入交互模式。

波形显示工具—SignalScan

> signalscan & 或 signalscan 数据库文件名 &



波形显示工具—SignalScan

在命令行输入**signalscan**启动。SignalScan窗口包括：

- **Title Bar:** 显示这是**SignalScan**窗口并以数字编号。若启动几个**SignalScan**窗口它们将顺序编号。
- **Menu Bar:** 通过菜单可以执行所有基本命令。
- **Tool Bar**中的按钮有：**copy, cut, paste, undo, delete, zoom, create marker, expand buses, launch the Design Brower**等等。用户可以自定义。

注：必须用**Design Brower**在波形窗口中添加信号。

- **Groups Pane**列出用户建立的波形组
- **Waveforms Region**显示加入信号的波形
- **Names Pane**在波形的左边显示信号名。这些信号名可以拖拽，在**pane**中双击右键可以移动插入的**marker**
- **Time-Display Region**显示两个指针的时间值及其时间差

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

例子：

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


SHM: 波形数据库

仿真历史管理器(**Simulation History Manager, SHM**)数据库记录用户的设计在仿真时数据信号的变化。只记录用户要观察(**probe**)的信号。

用户可以用`$shm_`系统任务打开一个**SHM**数据库，设置信号探针并将结果保存到数据库中。这些系统任务的功能除`$shm_probe`外都非常直观。对`$shm_probe`将在下面详细讨论。

用户必须在仿真前(时间0前)设置探针信号才能看到信号在仿真过程中全部变化。

用\$shm_probe设置信号探针

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

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

相关工具

与Cadence Verilog仿真器相关的工具有：

- Affirma NC VHDL仿真器
- Envisia Ambit综合工具
- Verilog-XL故障仿真器, 用于评价用户测试向量的有效性
- SignalScan-TX图形界面调试工具包
- Affirma equivalence checker完成门级设计之间或门级与RTL级之间的静态功能验证
- Affirma model checker形式验证工具，将Verilog或VHDL描述与设计说明进行验证
- Affirma model packager，用户的Verilog, VHDL或C语言可执行模型分发时进行编译及分发许可证
- Affirma Advanced Analysis Environment includes CoverScan, a code profiler, and HAL, a lint checker

总结

本章学习内容

- 逻辑仿真
- 运行Verilog-XL和NC Verilog仿真器
- 探测及显示波形

复习

1. 基于事件的仿真器是如何做到并行的？
2. 时间 t 的事件能否调度同一时间 t 的事件？
3. NC Verilog仿真器不支持IEEE 1364 Verilog LRM的什么元件？

1. 通过调度在一个给定的时间片内发生的所有事件来得到并行性。实际上仿真器串行处理给定时间片内的事件，但理论上它们都是在同一时间片内发生的。
2. 任何时间片的事件能够调度在同一时间片或其以后产生的事件。
3. NC Verilog希望支持IEEE 1364 LRM规范全集。目前主要还不支持实例阵列 (array of instances)。请参见产品发布手册。

第四章 设计举例

学习目标:

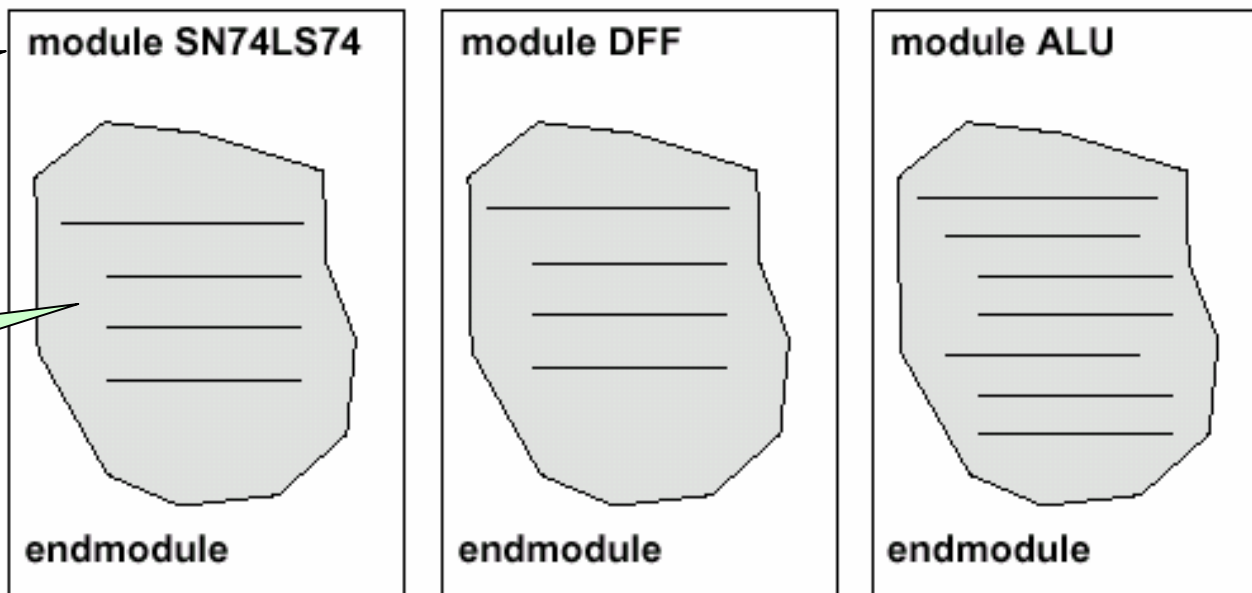
1. 进一步学习Verilog的结构描述和行为描述
2. Verilog混合（抽象）级仿真

语言的主要特点

module(模块)

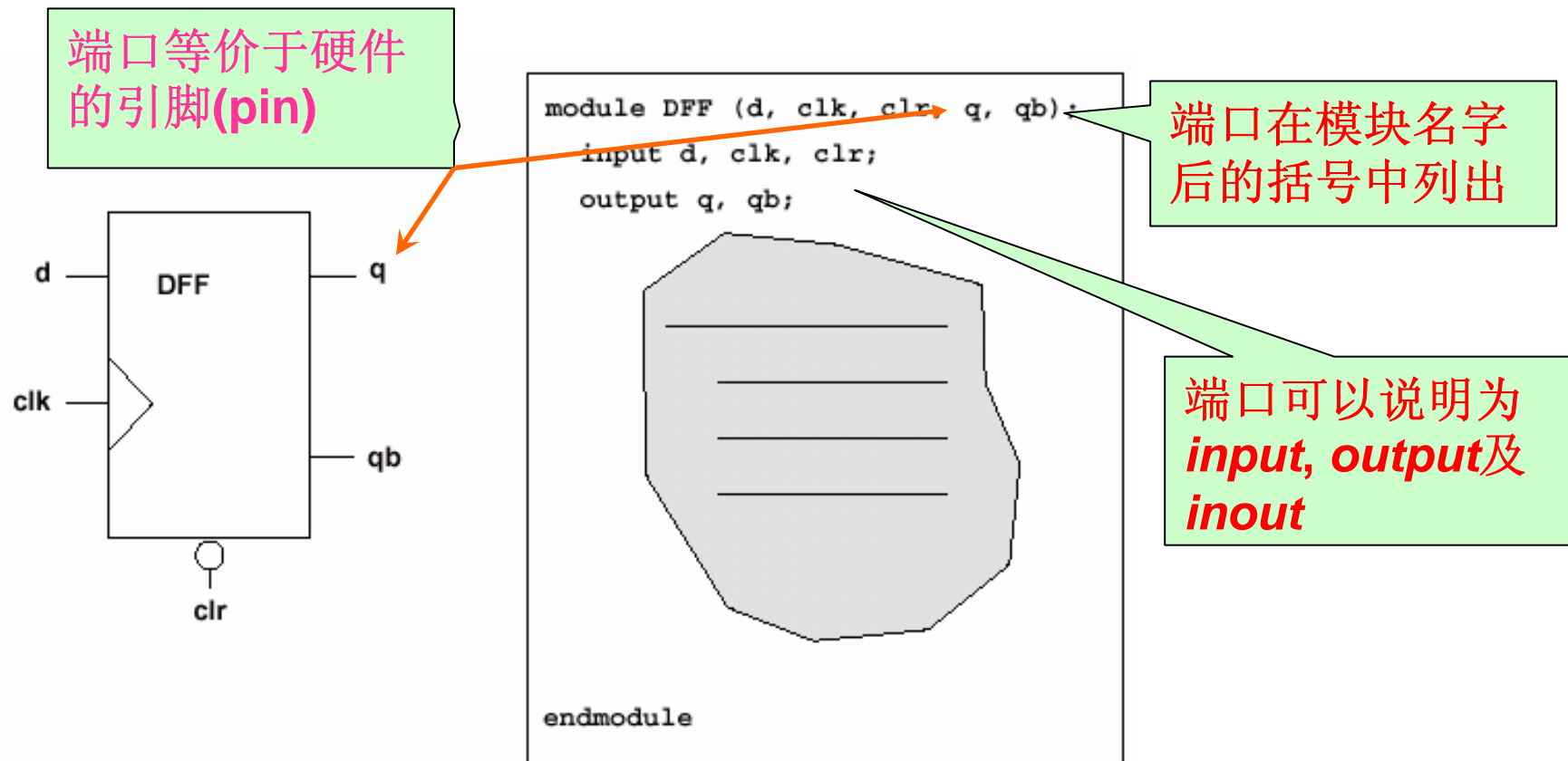
module是层次化设计的基本构件

逻辑描述放在**module**内部



- **module**能够表示：
 - 物理块，如**IC**或**ASIC**单元
 - 逻辑块，如一个**CPU**设计的**ALU**部分
 - 整个系统
- 每一个模块的描述从关键词**module**开始，有一个名称（如**SN74LS74**，**DFF**，**ALU**等等），由关键词**endmodule**结束。

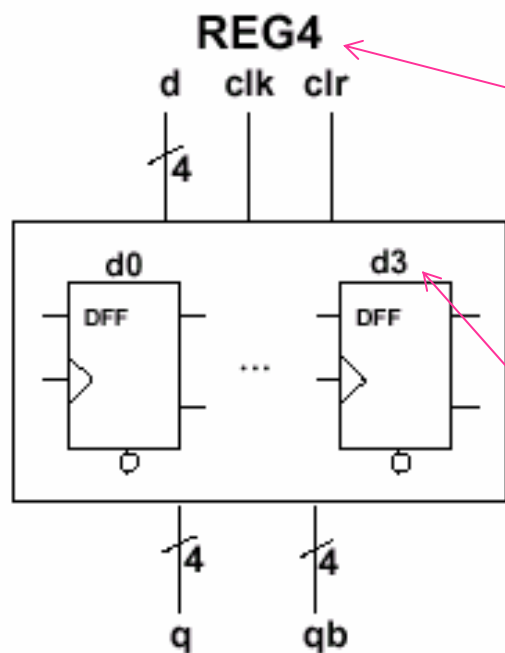
语言的主要特点—模块端口(module ports)



- 注意模块的名称**DFF**，端口列表及说明
- 模块通过端口与外部通信

语言的主要特点

模块实例化(module instances)



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

```
....
```

```
endmodule
```

```
module REG4( d, clk, clr, q, qb);
```

```
output [3: 0] q, qb;
```

```
input [3: 0] d;
```

```
input 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]);
```

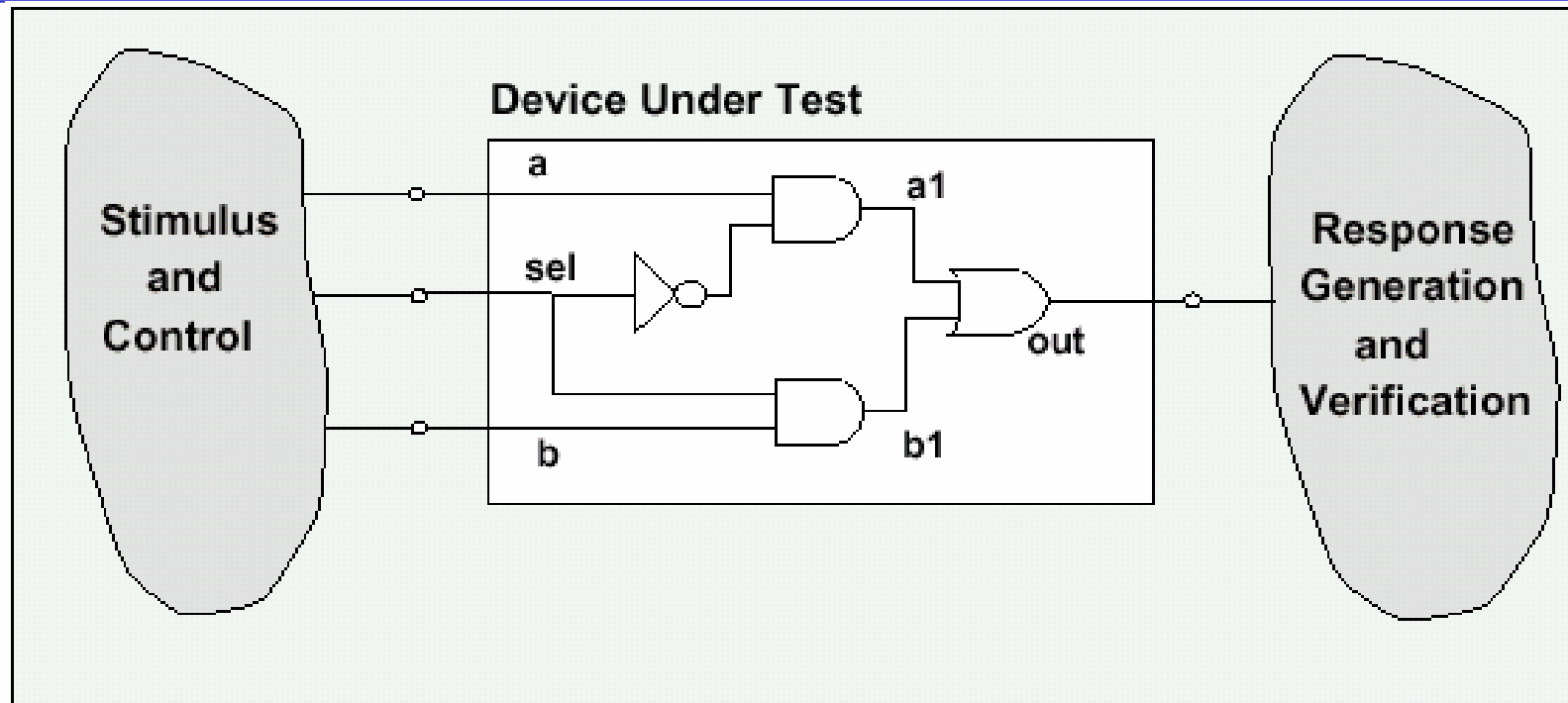
```
endmodule
```

语言的主要特点

模块实例化(module instances)

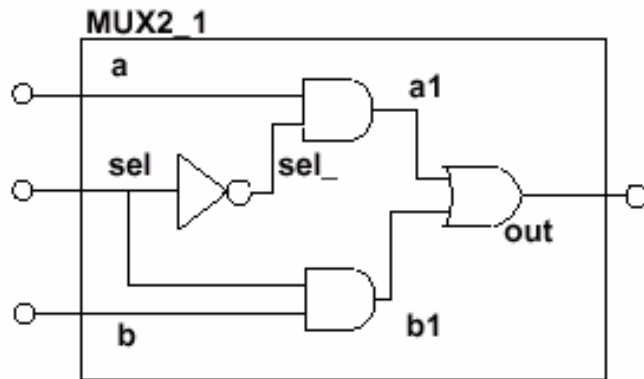
- 可以将模块的实例通过端口连接起来构成一个大的系统或元件。
- 在上面的例子中，**REG4**有模块**DFF**的四个实例。注意，每个实例都有自己的名字(**d0, d1, d2, d3**)。实例名是每个对象唯一的标记，通过这个标记可以查看每个实例的内部。
- 实例中端口的次序与模块定义的次序相同。
- 模块实例化与调用程序不同。每个实例都是模块的一个完全的拷贝，相互独立、并行。

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。测试装置(**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 (out, a, b,  
sel);
```

注释行

```
// Port declarations
```

```
output out;
```

```
input a, b, sel;
```

```
wire out, a, b, sel;
```

```
wire sel_, a1, b1;
```

```
// The netlist
```

```
not (sel_, sel);
```

```
and (a1, a, sel_);
```

```
and (b1, b, sel);
```

```
or (out, a1, b1);
```

```
endmodule
```

多路器由关键词 **module** 和 **endmodule** 开始及结束。

已定义的 **Verilog** 基本单元的实例

Test Fixture *template*

```
module testfixture;  
    // Data type declaration  
  
    // Instantiate modules  
  
    // Apply stimulus  
  
    // Display results  
  
endmodule
```



由于**testfixture**是最顶层模块，不会被其它模块实例化。因此不需要有端口。

Test Fixture — 如何说明实例

```
module testfixture;  
    // Data type declaration
```

```
    // Instantiate modules
```

```
        MUX2_1 mux (out, a, b,  
sel);
```

```
    // Apply stimulus
```

```
    // Display results
```

```
endmodule
```

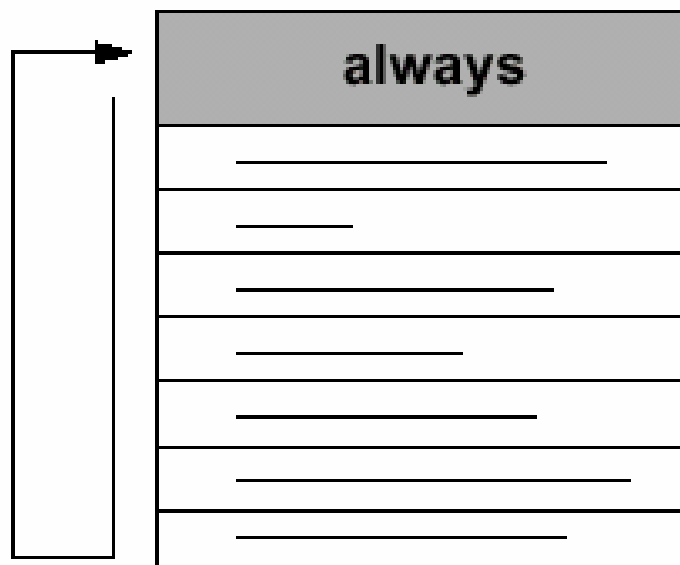
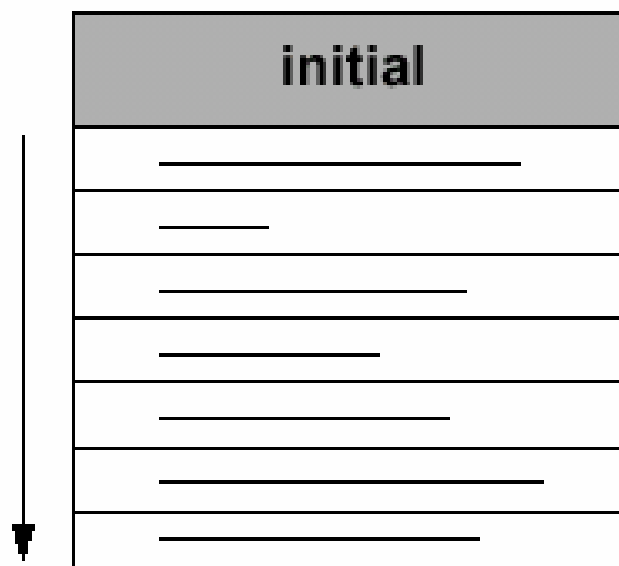
多路器实例化语句

MUX的实例化语句包括：

- 模块名字：与引用模块相同
- 实例名字：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 —过程(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
过程之间是并行执行的

Test Fixture —过程(procedural block)

- 通常采用过程语句进行行为级描述。test fixture的激励信号在一个过程语句中描述。
- 过程语句的活动与执行是有差别的
 - 所有过程在时间0处于活动状态，并根据用户定义的条件等待执行；
 - 所有过程并行执行，以描述硬件内在的并行性；

Test fixture 激励描述

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

Time	Values		
	a	b	sel
0	0	1	0
5	0	0	0
10	0	1	1
15	1	1	1

- 例子中，a, b, sel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，给出当前仿真时间
- *\$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

Test Fixture 响应产生

- *\$time*是一个系统函数，返回当前返回仿真时间。时间用64位整数表示。
- *\$monitor* 在时间单位末，若参数列表中的参数值发生变化则显示所列参数的值。由\$time引起的变化不会显示。
- \$monitor系统任务支持不同的数基。缺省数基是十进制。支持的数基还有二进制、八进制、十进制。

完整的Test Fixture

```
module testfixture;  
  // Data type declaration  
  reg a, b, sel;  
  wire out;  
  // MUX instance  
  MUX2_1 mux (out, a, b, sel);  
  // Apply stimulus  
  initial begin  
    a = 0; b = 1; sel = 0;  
    #5 b = 0; #5 b = 1; sel = 1;  
    #5 a = 1;  
    #5 $finish;  
  end  
  // Display results  
  initial  
    $monitor($time, " out=%b a=%b b=%b sel=%b", out, a, b,  
    sel);  
endmodule
```

结果输出

```
0    out= 0 a= 0 b= 1 sel=  
0  
5    out= 0 a= 0 b= 0 sel=  
0  
10   out= 1 a= 0 b= 1 sel=  
1  
15   out= 1 a= 1 b= 1 sel=  
1
```

时间单位末的概念

```
`timescale 1ns/1ns  
module testfixture;  
  // Data type declaration  
    reg a, b, sel;  
    wire out;  
  // MUX instance  
    MUX2_1 mux (out, a, b, sel);  
  // Apply stimulus  
    initial begin  
      a = 0; b = 1; sel = 0;  
      #5.7 b = 0; #5 b = 1; sel = 1;  
      #5 a = 1; #5 $finish;  
    end  
  // Display results  
    initial  
      $monitor($time,," out=%b a=%b b=%b sel=%b", out, a, b,  
sel);  
endmodule
```

结果输出

```
0    out= 0  a= 0  b= 1  sel=  
0  
6    out= 0  a= 0  b= 0  sel=  
0  
11   out= 1  a= 0  b= 1  sel=  
1  
16   out= 1  a= 1  b= 1  sel=  
1
```

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

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波形，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`。

\$dumpvars

\$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

\$dumpvars

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
```

复习

1. **Verilog**的基本构建模块是什么？是如何构成一个系统的？
 2. **module**怎样与其它模块通信？
 3. 仿真时两个性质不同的模块是什么？
 4. 在**test fixture**中两类不同的过程语句是什么？它们有什么不同？
 5. 用什么方法能以文本格式显示仿真结果？
1. **module**是基本构建单元。在**module**中实例化另一个**module**可以构成一个复杂的层次化系统。
 2. **module**之间通过端口的连接进行互相通信
 3. 两个模块是设计模块和激励模块。设计模块又称为**DUT**，激励模块又称为**testbench**或**test fixture**。测试模块用于设计模块验证
 4. 在**testbench**中用到的两类过程语句是**initial**和**always**。其不同处是**initial**只执行一次，而**always**循环执行。
 5. **\$monitor**语句以文本格式显示仿真结果

第五章 Verilog的词汇约定(Lexical convention)

学习内容:

1. 理解**Verilog**中使用的词汇约定
2. 认识语言专用标记(**tokens**)
3. 学习**timescale**

术语及定义

1. 空白符：空格、**tabs**及换行
2. **Identifier**: 标志符，**Verilog**中对象(如模块或端口)的名字
3. **Lexical**: 语言中的字或词汇，或与其相关。由其文法（**grammar**）或语法(**syntax**)区分。
4. **LSB**: 最低有效位(**Least significant bit**)
5. **MSB**: 最高有效位(**Most significant bit**)

空白符和注释

```
module MUX2_1 (out, a, b, sel);
```

```
  // Port declarations ← 单行注释  
  output out;
```

到行末结束

```
  input sel, // control input  
          b, /* data inputs */ a;
```

```
/*
```

```
  The netlist logic selects input "a" when  
  sel = 0 and it selects "b" when sel = 1. ←
```

多行注释，在/* */内

```
*/
```

```
  not (sel_, sel);  
  and (a1, a, sel_), (b1, b, sel); // What does this  
line do?  
  or (out, a1, b1);  
endmodule
```

格式自由

使用空白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。

整数常量和实数常量

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** : 大小，由十进制数表示的位数(**bit**)表示。缺省为**32**位

base: 数基，可为**2(b)**、**8(o)**、**10(d)**、**16(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	unsized decimal (zero-extended to 32 bits)
'H83a	unsized hexadecimal (zero- extended to 32 bits)
8'b1100_0001	8-bit binary
64'hff01	64-bit hexadecimal (zero- extended to 64 bits)
9'O17	9-bit octal
32'bz01x	Z-extended to 32 bits
3'b1010_1101	3-bit number, truncated to 3'b101
6.3	decimal notation
32e- 4	scientific notation for 0.0032
4.1E3	scientific notation for 4100

整数常量和实数常量

- 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，便于查看
 - 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**大于指定的大小时，截去高位。如 **2'b1101**表示的是**2'b01**
- 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 $\times 10^{\text{指数}}$

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，也就是不能跨行。
- 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"

字符串 (string)

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

标识符(identifiers)

- 标识符是用户在描述时给**Verilog**对象起的名字
- 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、端口和实例的名字都是标识符

```
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
```



Verilog标识符

标识符(identifiers)

- 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。

转义标识符(Escaped identifiers)

转义标识符由反斜杠“\”开始，空白符结束

- 可以包含任何可打印字符
- 反斜杠及空白符不是标识符的一部分

```
module \2:1MUX (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
```

Escaped Identifiers

- 使用转义符可能会产生一些问题，并且不是所有工具都支持。有时用转义符完成一些转换，如产生逻辑图的**Verilog**网表。综合工具输出综合网表时也使用转义符。不建议使用转义符。

转义标识符(Escaped identifiers)

转义标识符允许用户在标识符中使用非法字符。如：

`\~#@sel`

`\busa+ index`

`\{A,B}`

`top.\ 3inst .net1` // 在层次化名字中转义符



转义标识符必须以空格结束

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);
```

当信号a或b的值发生变化时，系统任务\$monitor显示当前仿真时间，信号a值(二进制格式)，信号b值（16进制格式）。

语言专用标记(tokens)

延时说明

- “#”用于说明过程(procedural)语句和门的实例的延时，但不能用于模块的实例化。

```
module MUX2_1 (out, a, b, sel) ;  
output out ;  
input a, b, sel ;  
    not #1 not1( sel_, sel);  
    and #2 and1( a1, a, sel_);  
    and #2 and2( b1, b, sel);  
    or #1 or1( out, a1, b1);  
endmodule
```

- 门延时有很多类名字：门延时(gate delay)，传输延时(propagation delay)，固有延时(intrinsic delay)，对象内在延时(intra-object delay)

编译指导(Compiler Directives)

- (`\`)符号说明一个编译指导
- 这些编译指导使仿真编译器进行一些特殊的操作
- 编译指导一直保持有效直到被覆盖或解除
- `\resetall` 复位所有的编译指导为缺省值，应该在其它编译指导之前使用

文本替换(substitution) - `define

编译指导`define提供了一种简单的文本替换的功能

`define <macro_name> <macro_text>

在编译时<macro_text>替换<macro_name>。可提高描述的可读性。

```
`define not_delay #1
`define and_delay #2
`define or_delay #1
module MUX2_1 (out, a, b, sel);
output out;
input a, b, sel;
    not `not_delay not1( sel_,
    sel);
    and `and_delay and1( a1, a,
    sel_);
    and `and_delay and2( b1, b,
    sel);
    or `or_delay or1( out, a1,
    b1);
```

定义not_delay

使用not_delay

文本替换(substitution)

- 解除定义的宏，使用
 - ``undef macro_name`
- 使用编译指导**`define**，可以
 - 提高描述的可读性
 - 定义全局设计参数，如延时和矢量的位数。这些参数可以定义在同一位置。这样，当要修改设计配置时，只需要在一个地方修改。
 - 定义Verilog命令的简写形式
 - ``define vectors_file "/usr1/chrisz/library/vectors"`
 - ``define results_file "/usr1/chrisz/library/results"`
- 可以将**`define**放在一个文件中，与其它文件一起编译。

文本包含(inclusion) - ``include`

- 编译指导**``include`**在当前内容中插入一个文件

格式: ``include "<file_name>"`

如 ``include "global.v"`

``include "parts/count. v"`

``include "../..../library/mux. v"`



可以是相对路径或绝对路径

- **``include`**可用于:
 - `include`保存在文件中的全局的或经常用到的一些定义, 如文本宏
 - 在模块内部**`include`**一些任务 (**tasks**), 提高代码的可维护性。

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  
// All time units are in multiples of 1 nanosecond  
module MUX2_1 (out, a, b, sel);  
output out;  
input a, b, sel;  
    not #1 not1( sel_, sel);  
    and #2 and1( a1, a, sel_);  
    and #2 and2( b1, b, sel);  
    or #1 or1( out, a1, b1);  
endmodule
```

Timescale

- **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 10^{12} 次; 而`timescale 1s/1ms则只需扫描 10^3 次。
- 如果没有**timescale**说明将使用缺省值, 一般是**ns**。

Timescale

- 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
```

复习

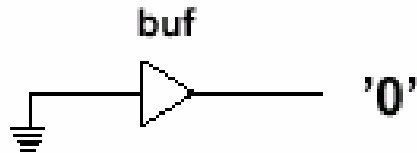
1. Verilog中的空白符总是忽略的吗？
 2. 在源代码中插入注释有哪两种方法？
 3. 整数常数的尺寸如何指定？缺省的尺寸及数基是多少？
 4. 设置的编译指导如何解除？
 5. 编译指导影响全局吗？
 6. 在仿真时为什么要用接近实际的最大timescale精度？
1. 是的。空白符用于隔开标识符及关键词，多余的忽略
 2. //用于单行注释，/* */用于多行注释
 3. 整数常量的尺寸由10进制数表示的位数确定。缺省为32位，缺省的数基为十进制。
 4. 使用`resetall解除
 5. 编译指导是全局的。编译时遇到编译指导后开始有效，直至复位或被覆盖，可能影响多个文件。
 6. 使用尽可能大的精度。精度越小，仿真时间步越小，仿真时间越长。使用适当的精度，既达到必要的精度，又不会仿真太慢。

第六章 Verilog的数据类型及逻辑系统

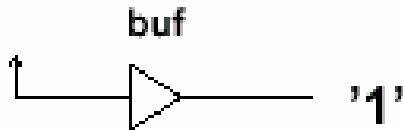
学习内容:

- 学习Verilog逻辑值系统
- 学习Verilog中不同类的数据类型
- 理解每种数据类型的用途及用法
- 数据类型说明的语法

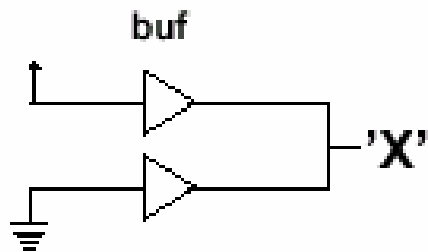
Verilog采用的四值逻辑系统



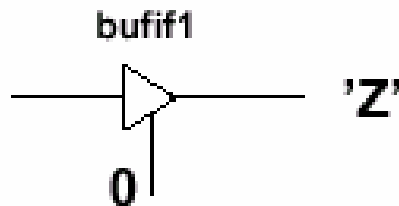
'0', Low, False, Logic Low, Ground, VSS, Negative Assertion



'1', High, True, Logic High, Power, VDD, VCC, Positive Assertion



'X' Unknown: Occurs at Logical Which Cannot be Resolved Conflict



HiZ, High Impedance, Tri- Stated, Disabled Driver (Unknown)

主要数据类型

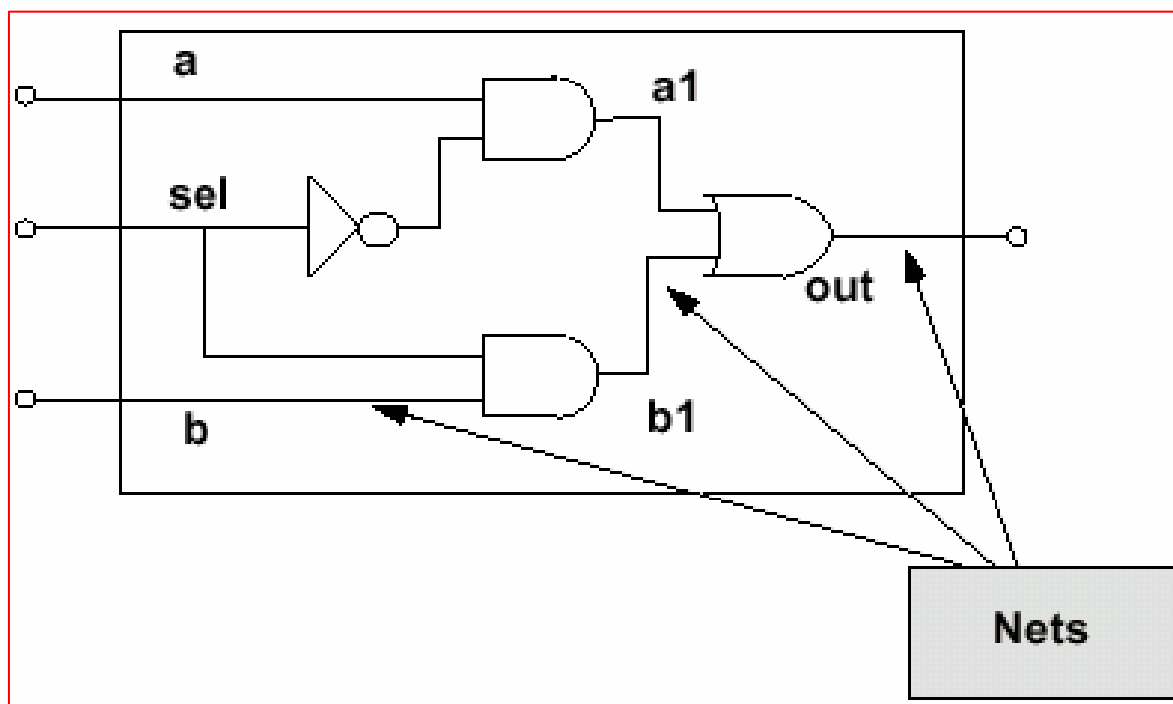
Verilog主要有三类(class)数据类型:

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

net（线网）

net需要被持续的驱动，驱动它的可以是门和模块。

当**net**驱动器的值发生变化时，**Verilog**自动的将新值传送到**net**上。在例子中，线网**out**由**or**门驱动。当**or**门的输入信号置位时将传输到线网**net**上。



net类的类型（线网）

- 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	多驱动源线或
wand, triand	多驱动源线与
trireg	能保存电荷的net
tri1, tri0	无驱动时上拉/下拉

- 没有声明的net的缺省类型为 1 位(标量)wire类型。但这个缺省类型可由下面的编译指导改变:

```
`default_nettype <nettype>
```

net类的类型（线网）

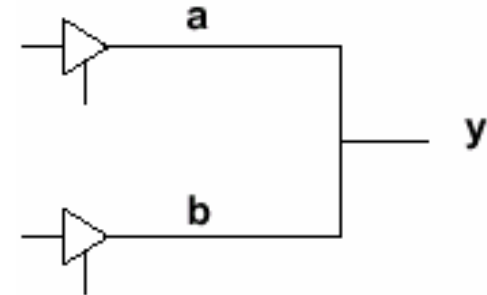
- **wire**类型是最常用的类型，只有连接功能。
- **wire**和**tri**类型有相同的功能。用户可根据需要将线网定义为**wire**或**tri**以提高可读性。例如，可以用**tri**类型表示一个**net**有多个驱动源。或者将一个**net**声明为**tri**以指示这个**net**可以是高阻态Z(hign-impedance)。可推广至**wand**和**triand**、**wor**和**trior**
- **wand**、**wor**有线逻辑功能；与**wire**的区别见下页的表。
- **triereg**类型很象**wire**类型，但**triereg**类型在没有驱动时保持以前的值。这个值的强度随时间减弱。
- 修改**net**缺省类型的编译指导：

``default_nettype <nettype>`

nettype不能是**supply1**和**supply0**。

net类在发生逻辑冲突时的决断

- Verilog有预定义的决断函数
- 支持与工艺无关的逻辑冲突决断
 - wire-and用于集电极开路电路
 - wire-or用于射极耦合电路



Wire/Tri

$\begin{smallmatrix} b \\ \backslash a \end{smallmatrix}$	0	1	x	z
0	0	x	x	0
1	x	1	x	1
x	x	x	x	x
z	0	1	x	z

y

Wand/Triand

$\begin{smallmatrix} b \\ \backslash a \end{smallmatrix}$	0	1	x	z
0	0	0	0	0
1	0	1	x	1
x	0	x	x	x
z	0	1	x	z

y

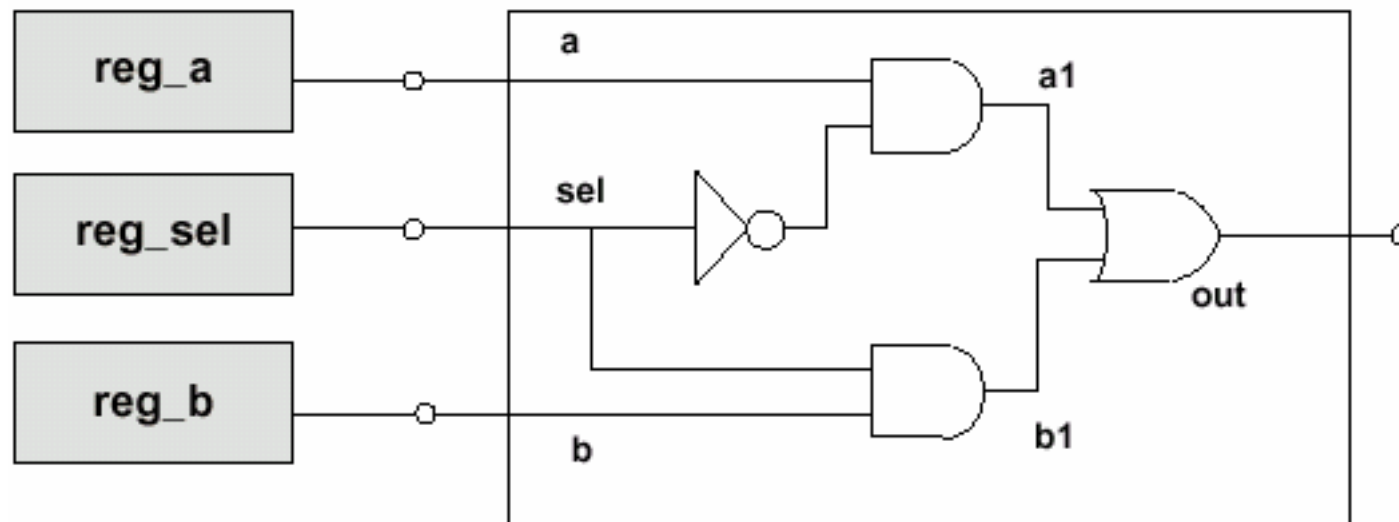
Wor/Trior

$\begin{smallmatrix} b \\ \backslash a \end{smallmatrix}$	0	1	x	z
0	0	1	x	0
1	1	1	1	1
x	x	1	x	x
z	0	1	x	z

y

寄存器类 (register)

- 寄存器类型在赋新值以前保持原值
- 寄存器类型大量应用于行为模型描述及激励描述。在下面的例子中，**reg_a**、**reg_b**、**reg_sel**用于施加激励给2:1多路器。
- 用行为描述结构给寄存器类型赋值。给**reg**类型赋值是在过程块中。



寄存器类的类型

- 寄存器类有四种数据类型

寄存器类型	功能
-------	----

reg	可定义的可符号整数变量，可以是标量(1位)或矢量，是最常用的寄存器类型
integer	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的数据处理。
real	双精度的带符号浮点变量，用法与integer相同。
time	64位无符号整数变量，用于仿真时间的保存与处理
realtime	与real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**

<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

net_name: net名称，一次可定义多个net, 用逗号分开。

- **寄存器声明**

<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

Verilog中net和register声明语法

- 举例:

reg a; // 一个标量寄存器

wand w; // 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 [7: 0] m, n; // 两个8位寄存器

tri [15: 0] busa; // 16位三态总线

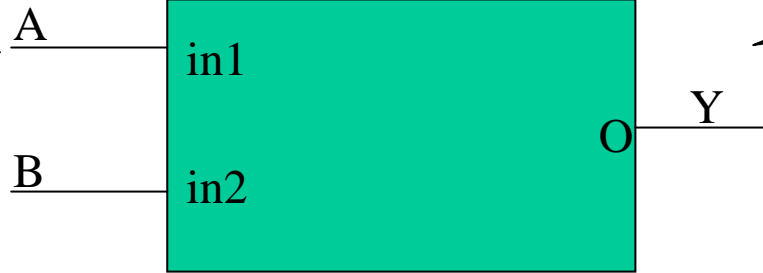
wire [0: 31] w1, w2; // 两个32位wire, MSB为bit0

选择正确的数据类型

输入端口可以由
net/register驱动，但输入
端口只能是**net**

输出端口可以是
net/register类型，输出
端口只能驱动**net**

双向端口输入/输出只
能是**net**类型



```
module top;
wire y;
reg a, b;
  DUT u1 (y, a, b) ;
  initial begin
    a = 0; b = 0;
    #5 a = 1;
  end
endmodule
```

在过程块中只能给
register类型赋值

```
module DUT (Y, A, B);
output Y;
input A, B;
wire Y, A, B;
  and (Y, A, B) ;
endmodule
```

若**Y, A, B**说明为
reg则会产生错误。

选择数据类型时常犯的错误

信号类型确定方法总结如下：

- 信号可以分为端口信号和内部信号。出现在端口列表中的信号是端口信号，其它的信号为内部信号。
- 对于端口信号，输入端口只能是net类型。输出端口可以是net类型，也可以是register类型。若输出端口在过程块中赋值则为register类型；若在过程块外赋值(包括实例化语句)，则为net类型。
- 内部信号类型与输出端口相同，可以是net或register类型。判断方法也与输出端口相同。若在过程块中赋值，则为register类型；若在过程块外赋值，则为net类型。
- 若信号既需要在过程块中赋值，又需要在过程块外赋值。这种情况是有可能出现的，如决断信号。这时需要一个中间信号转换。

下面所列是常出的错误及相应的错误信息(error message)

- 用过程语句给一个net类型的或忘记声明类型的信号赋值。
信息: illegal assignment.
- 将实例的输出连接到声明为register类型的信号上。
信息: <name> has illegal output port specification.
- 将模块的输入信号声明为register类型。
信息: incompatible declaration, <signal name>

选择数据类型时常犯的错误举例

example.v

修改前:

```
module example(o1, o2, a, b, c, d);  
    input a, b, c, d;  
    output o1, o2;  
    reg c, d;  
    reg o2  
    and u1(o2, c, d);  
    always @(a or b)  
        if (a) o1 = b; else o1 = 0;  
endmodule
```

修改后:

```
module example(o1, o2, a, b, c, d);  
    input a, b, c, d;  
    output o1, o2;  
    // reg c, d;  
    // reg o2  
    reg o1;  
    and u1(o2, c, d);  
    always @(a or b)  
        if (a) o1 = b; else o1 = 0;  
endmodule
```

选择数据类型时常犯的错误举例

Compiling source file "example.v"

**Error! Incompatible declaration, (c) defined as input
at line 2 [Verilog-IDDIL]**

"example.v", 5:

**Error! Incompatible declaration, (d) defined as input
at line 2 [Verilog-IDDIL]**

"example.v", 5:

Error! Gate (u1) has illegal output specification [Verilog-GHIOS]

"example.v", 8:

3 errors

第一次编译信息

verilog -c example.v

Compiling source file "example.v"

Error! Illegal left-hand-side assignment

[Verilog-ILHSA]

"example.v", 11: o1 = b;

Error! Illegal left-hand-side assignment

[Verilog-ILHSA]

"example.v", 12: o1 = 0;

2 errors

第二次编译信息

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 <list_of_assignment>;**
- 可一次定义多个参数，用逗号隔开。
- 在使用文字(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 mod1( out, in1, in2);  
    ...  
    parameter cycle = 20, prop_del = 3,  
                setup = cycle/2 - prop_del,  
                p1 = 8,  
                x_word = 16'bx,  
                file = "/usr1/jdough/design/mem_file.dat";  
    ...  
    wire [p1: 0] w1; // A wire declaration using parameter  
    ...  
endmodule
```

注意：参数file不是string，而是一个整数，其值是所有字母的扩展ASCII值。若file="AB"，则file值为8'h4142。用法：

```
$fopen(file);  
  
$display("%s", file);
```


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;  
...  
  mod1 l1( out, in1, in2);  
  defparam  
    l1. p1 = 6,  
    l1. file = "../ my_mem.dat";  
...  
endmodule
```

```
module mod1( out, in1, in2);  
...  
  parameter p1 = 8,  
             real_constant = 2.039,  
             x_word = 16'bx,  
             file =  
"/usr1/jdough/design/mem_file.dat";  
...  
endmodule
```

参数重载 (overriding)

模块实例化时参数重载

```
module mod1( out, in1, in2);  
...  
parameter p1 = 8,  
           real_constant = 2.039,  
           x_word = 16'bx,  
           file = "/usr1/jdough/design/mem_file.dat";  
...  
endmodule  
module top;  
...  
  mod1 #( 5, 3.0, 16'bx, "../ my_mem. dat") l1( out, in1, in2);  
...  
endmodule
```

次序与原说明相同

使用#

因为#说明延时的时候只能用于gate或过程语句，不能用于模块实例。

gate (primitives)在实例化时只能有延时，不能有模块参数。

为什么编译器认为这是参数而不是延时呢？

寄存器数组(Register Arrays)

- 在Verilog中可以说明一个寄存器数组。

`integer NUMS [7: 0]; // 包含8个整数数组变量`

`time t_vals [3: 0]; // 4个时间数组变量`

- reg**类型的数组通常用于描述存储器

其语法为: `reg [MSB:LSB] <memory_name> [first_addr:last_addr];`

`[MSB:LSB]`定义存储器字的位数

`[first_addr:last_addr]`定义存储器的深度

例如:

`reg [15: 0] MEM [0:1023]; // 1K x 16存储器`

`reg [7: 0] PREP ['hFFFE: 'hFFFF]; // 2 x 8存储器`

- 描述存储器时可以使用参数或任何合法表达式

`parameter wordsize = 16;`

`parameter memsize = 1024;`

`reg [wordsize-1: 0] MEM3 [memsize-1: 0];`

存储器寻址(Memory addressing)

- 存储器元素可以通过存储器索引 (**index**)寻址，也就是给出元素在存储器的位置来寻址。

mem_name [addr_expr]

- Verilog**不支持多维数组。也就是说只能对存储器字进行寻址，而不能对存储器中一个字的位寻址。

```
module mems;  
  reg [8: 1] mema [0: 255]; // declare memory called mema  
  reg [8: 1] mem_word;      // temp register called mem_word  
  ...  
  initial  
    begin  
      $displayb( mema[5]);    //显示存储器中第6个字的内容  
      mem_word = mema[5];  
      $displayb( mem_word[8]); //显示第6个字的最高有效位  
    end  
endmodule
```

若要对存储器字的某些位存取，只能通过暂存器传递

复习(review)

问题:

1. 在Verilog中, 什么情况下输出端会输出X值?
2. net和register类型的主要区别是什么?
3. 在Verilog中如何定义一个常数?

解答:

1. 若输出端输出X值, 一种可能是输出net上发生驱动冲突, 二是由一个未知值传递到net上引起。
2. register有存储功能, 而net必须持续驱动。
3. 在Verilog中使用parameter定义一个常数。文本宏也是常数的一种形式。

第7章 结构描述(structural modeling)

学习内容:

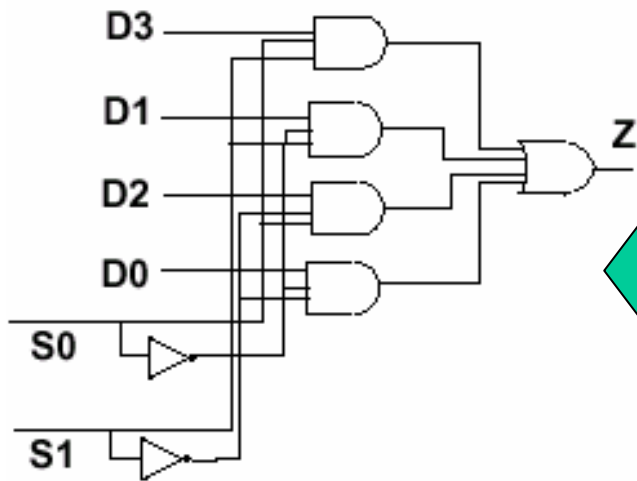
- 如何使用Verilog的基本单元(primitives)
- 如何构造层次化设计
- 了解Verilog的逻辑强度系统

术语及定义 (terms and definitions)

- 结构描述：用门来描述器件的功能
- **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。

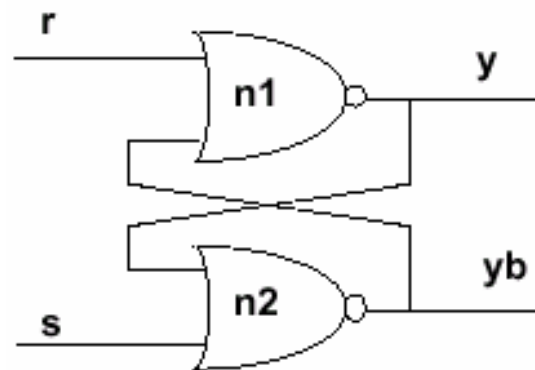


结构描述等价于逻辑图，都是连接简单元件构成更复杂元件

```
module MUX4x1( Z, D0, D1, D2, D3, S0, S1);
    output Z;
    input D0, D1, D2, D3, S0, S1;
    and (T0, D0, S0_, S1_),
        (T1, D1, S0_, S1_),
        (T2, D2, S0_, S1_),
        (T3, D3, S0_, S1_);
    not (S0_, S0), (S1_, S1);
    or (Z, T0, T1, T2, T3);
endmodule
```

同一种门可以通过一个语句实例化

忽略了门的实例名。



通过门的实例使用门

```
Latch
module rs_latch (y, yb, r, s);
    output y, yb;
    input r, s;
    nor n1( y, r, yb);
    nor n2( yb, s, y);
endmodule
```


结构描述（续）

- 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，也就是说这些逻辑功能是预定义的，用户不需要再定义这些基本功能。
- 基本单元是Verilog开发库的一部分。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

基本单元的引脚 (pin)的可扩展性

- 基本单元引脚的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。



```
and (out, in1, in2);
```



```
and (out, in1, in2, in3);
```



```
and (out, in1, in2, in3, in4);
```

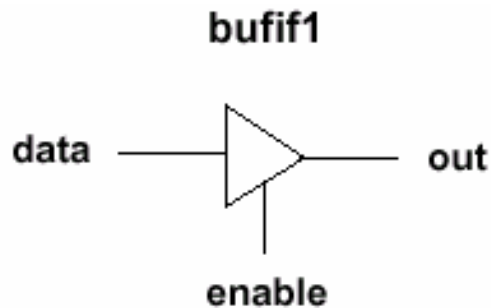
带条件的基本单元

- **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。

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

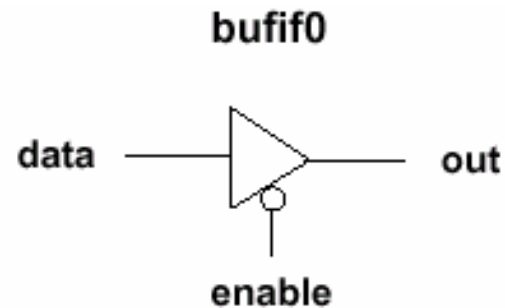
带条件的基本单元（续）

- 条件基本单元有三个端口：输出、数据输入、使能输入



bufif1 (out, data, enable)

		enable			
bufif1		0	1	x	z
data	0	z	0	L	L
	1	z	1	H	H
	x	z	x	x	x
	z	z	x	x	x

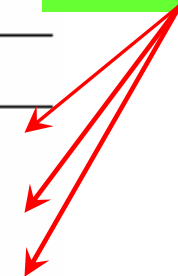


bufif0 (out, data, enable)

		enable			
bufif0		0	1	x	z
data	0	0	z	L	L
	1	1	z	H	H
	x	x	z	x	x
	z	x	z	x	x

三种未知状态:

	值	强度
x	1, 0, z	未知
L	0, z	未知
H	1, z	未知



基本单元实例化

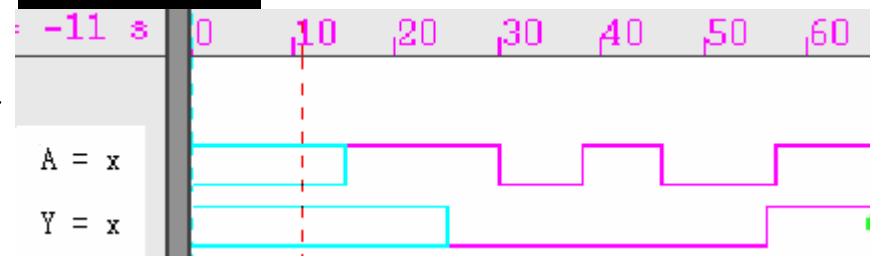
- 在端口列表中，先说明输出端口，然后是输入端口
- 实例化时实例的名字是可选项
 `and (out, in1, in2, in3, in4); // unnamed instance`
 `buf b1 (out1, out2, in); // named instance`
- 延时说明是可选项。所说明的延时是固有延时。输出信号经过所说明的延时才变化。没有说明时延时为0。
 `notif0 #3.1 n1 (out, in, cntrl); // delay specified`
- 信号强度说明是可选项
 `not (strong1, weak0) n1 (inv, bit); // strength specified`

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

固有延时



仿真波形



模块实例化(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
```

没有连接时通常会产生警告

```
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
```

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
```

范围说明语法：
[MSB : LSB]

```
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

实例数组(Array of Instances)(续)

- 如果范围中**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 [3: 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
```

现有综合
工具还不
支持实例
数组

m1作为实例阵列名字使用了两次

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。

逻辑强度(strength)模型（续）

- 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**triereg**和基本单元**tran**

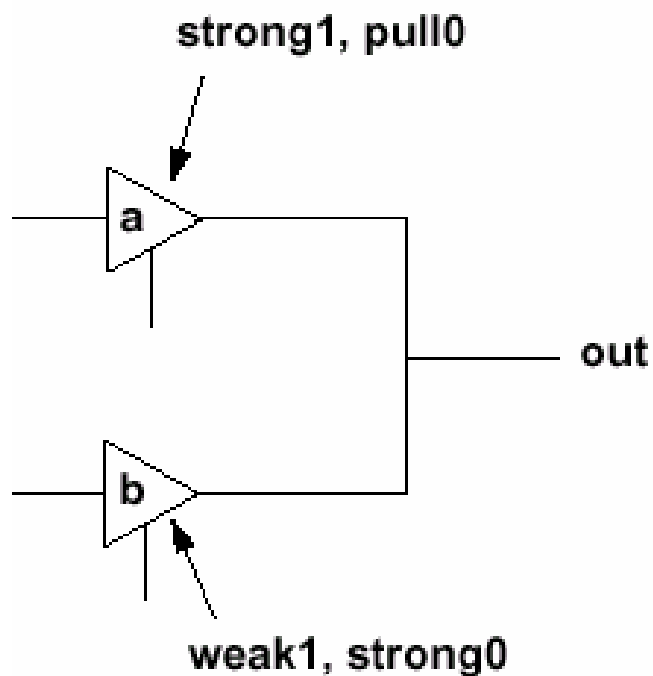
例如： **triereg (small) tl;**

信号强度值系统

	Level	Type	%v formats		Specification
Supply	7	Drive	Su0	Su1	supply0, supply1
Strong	6	Drive(default)	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

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

复习

问题：

1. 什么是**Verilog**中的结构化描述？
2. 连接模块端口有哪两种方法？哪一种更通用一些？
3. 什么是实例数组？
4. 什么时候实例名字是可选的？

解答：

1. 结构化描述是使用**Verilog**基本单元或单元(**cell**)级元件对设计进行描述，与逻辑图很相似。
2. 可以根据次序（位置映射）或名字（名称映射）来映射端口。虽然在这个教程中大多使用位置映射，这主要是为了节省空间。在实际设计中，名称映射可能更通用一些。
3. 实例数组用一条语句创建模型(模块或基本单元)的多个实例。
4. 基本单元实例化时实例名是可选的，说明基本单元数组时除外。当实例化模块时，实例名是必须的。

第8章 延时模型

学习内容:

1. 如何说明块延时
2. 如何说明分布延时
3. 如何说明路径延时
4. 怎样在模块中说明时序检查
5. 标准延时格式**SDF (Standard Delay Format)**

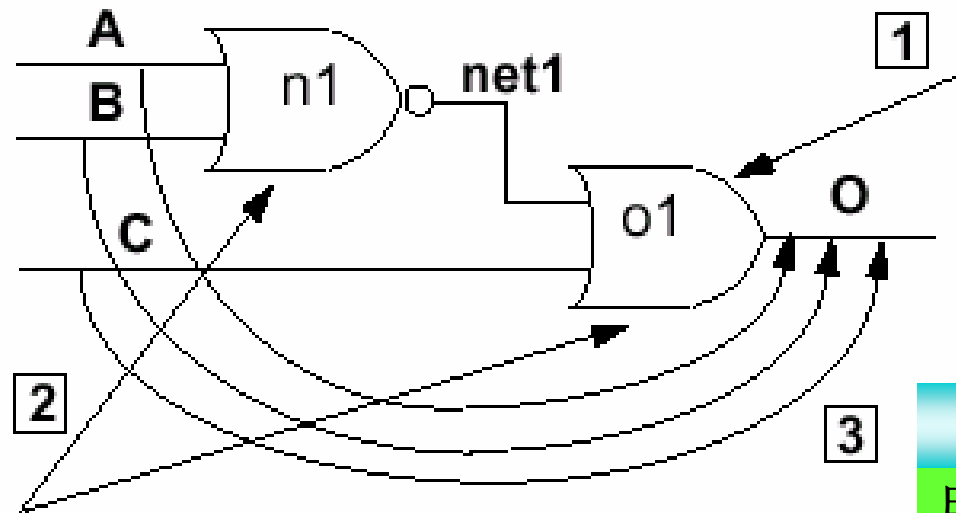
术语及定义

- **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
- **PLI:** 编程语言接口，提供 Verilog 数据结构的过程访问。
- **时序检查(timing check):** 监视两个输入信号的关系并检查的系统任务，以保证电路能正确工作。
- **时序驱动设计(timing driven design):** 从前端到后端的完整设计流程中，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：

noror ASIC 单元



块延时

将全部延时集中到最后一个门

路径延时

用专用块说明每一个路径pin-to-pin延时

分布延时

延时分布在每一个门上

典型的延时说明：

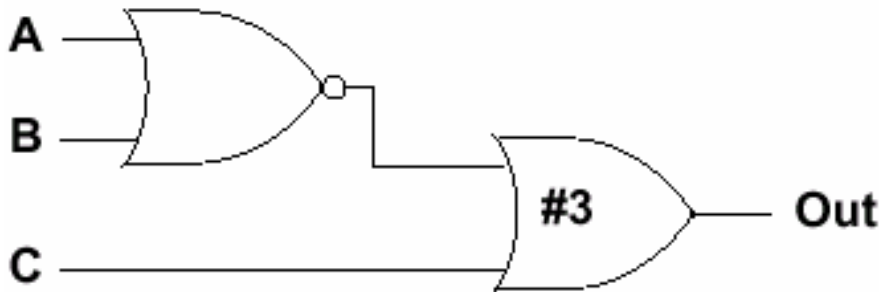
Delay from A to O = 2

Delay from B to O = 3

Delay from C to O = 1

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：

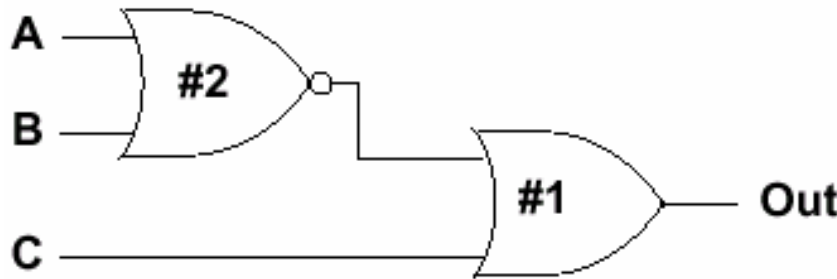
A -> Out is 3 ns

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：

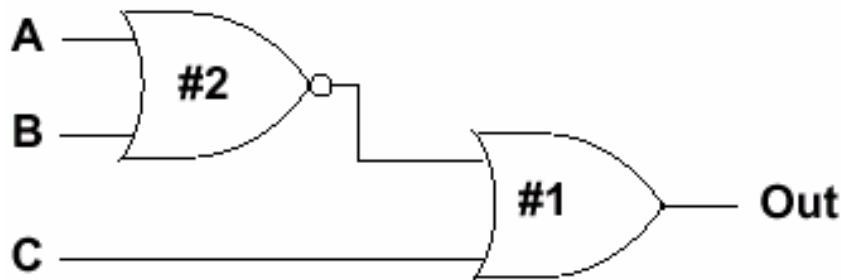
A -> Out is 3 ns

B -> Out is 3 ns

C -> Out is 1 ns

模块路径延时(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：时序与功能分开说明



例中各路径延时为：

A -> Out is 2 ns

B -> Out is 3 ns

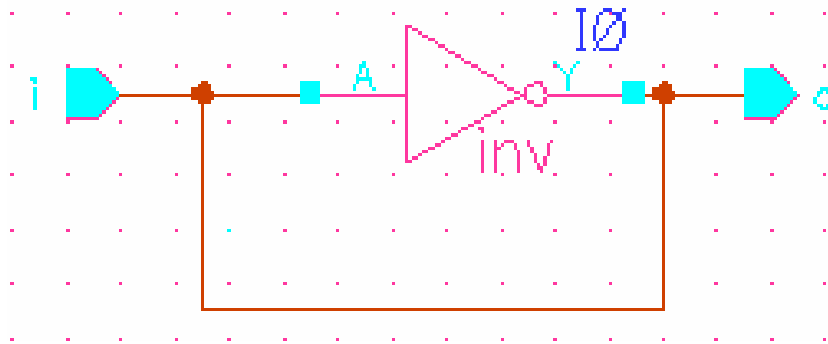
C -> Out is 1 ns

路径必须括
在圆括号内

```
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
```

结构描述的零延时反馈(Loop)

- 当事件队列中所有事件结束时仿真前进一个时片。在某种零延时反馈情况下，新事件在同一片不断的加入，致使仿真停滞在那个时片。
- 若在结构描述中出现从输出到输入的零反馈情况，多数仿真器会检测到这个反馈并产生错误信息。**Verilog的lint checker**对这种情况会提出警告。
- 解决这个问题的方法是在电路中加入分布延时。路径延时不能解决零延时振荡问题，因为输出信号在反馈前不会离开模块。



精确延时控制

在Verilog中，可以：

- 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
user_module #(1:2:3, 2:3:4) (.....) ;在Cadence Verilog中还不支持
(b => y) = (2: 3: 4, 3: 4: 6, 4: 5: 8); // min: typ: max for rise, fall, and turnoff

精确延时控制（续）

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。

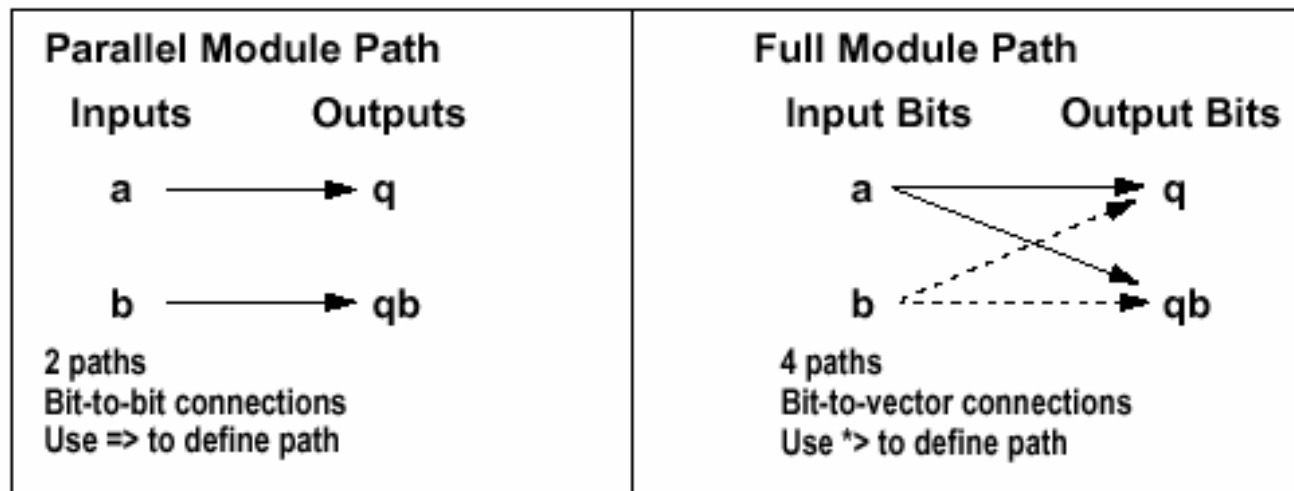
- 上升延时是输出转换为1时的延时
- 下降延时是输出转换为0时的延时
- 关断延时输出转换为三态时的延时
- 到X的转换延时是最小延时，而从X到其它值的转换使用最坏延时
 - 如果说明了上升、下降和关断延时，则1->X的转换延时使用上升和关断延时的最小值。X->0的延时为下降延时；X->Z的转换为关断延时。
 - 如果只说明了上升和下降延时，则1->X和X->0使用下降延时，X->Z使用上升和下降延时的最小延时
 - 如果只说明了一个延时，则所有转换使用这个延时。
 - 如果说明了六个延时，则1->X使用1->X和1->Z中最小延时；X->0使用1->0和X->0的最大延时；X->Z使用1->Z和0->Z中的最大延时。

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。
 - 设计的功能描述中的延时，如`#delay`在综合时不起作用
- **specify**块由**specify**开始， 到**endspecify**结束， 并且在模块内部
- 使用关键字**specparam**在**specify**中进行参数声明。不要同模块参数(由**parameter**说明)混淆。**specparam**只能在**specify**块内声明参数并使用；而**parameter**也只能在**specify**外声明参数并使用。
- **specify**块可以：
 - 描述穿过模块的路径及其延时
 - 描述时序检查以保证器件的时序约束能够得到满足
 - 定义特定模块或特定模块路径的时钟过滤限制

模块路径的并行连接和全连接（specify续）

- `*>`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;`

模块路径的并行连接和全连接（specify续）

这里有一些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;

// 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

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

使用specparam定义参数的例子：

```
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
```

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。

```
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
```

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，**a**到**out**的延时依赖于**b**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。

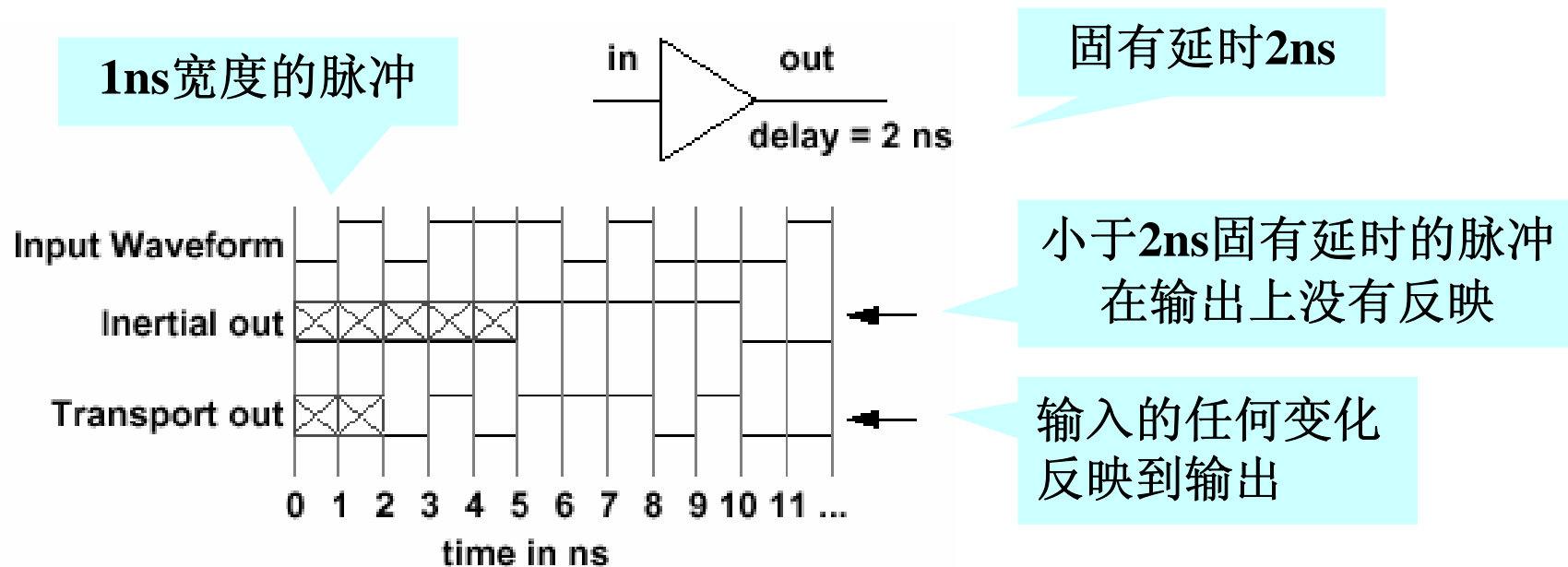
条件有一些限制，但许多仿真器并不遵循**IEEE**标准的限制。

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导指定延迟模型。

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

`(en => q) = 12;`

`(data => q) = 10;`

`(clr, pre *> q) = 4;`

specparam

`PATHPULSE$ = 3,`

`PATHPULSEenq = (2, 9) ,`

`PATHPULSEclrq = 1 ;`

endspecify

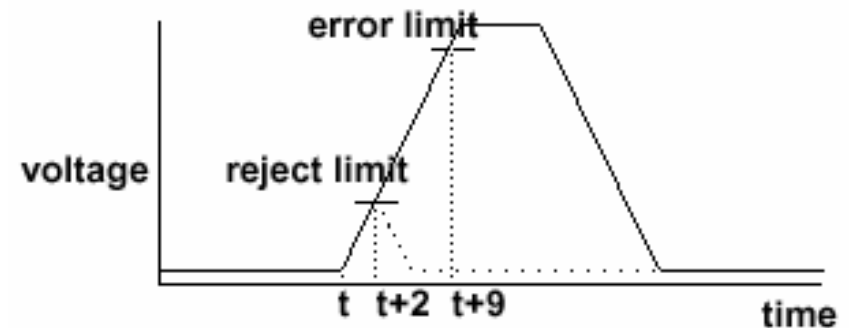
例子



en输入波形

输出q带倾斜波形

硬件可能产生的波形



路径脉冲控制

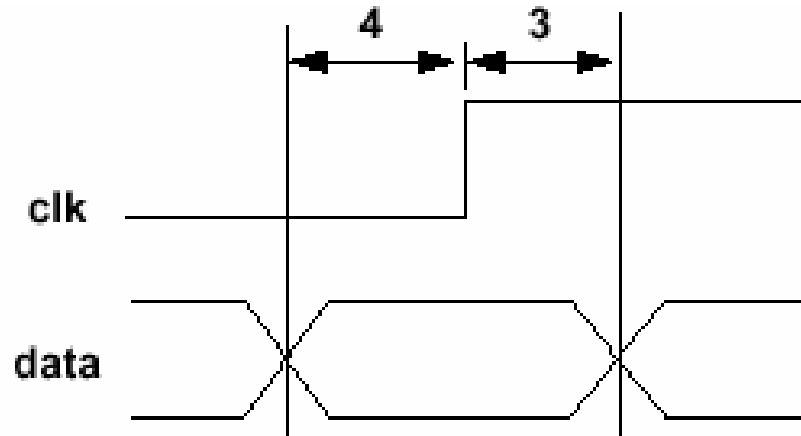
- 可以用`PATHPULSE$`声明的 *specparam* 参数覆盖全局脉冲控制
- `PATHPULSE$`声明的 *specparam* 参数缩小了指定模块或模块内特定路径的模块路径延时的范围。
- 只声明一个值时，`error_value`和`reject_value`相同，如
`PATHPULSE$ = 3;` 等价于 `PATHPULSE$ = (3, 3);`
- 脉冲宽度小于`reject_value`的信号将被滤掉，而小于`error_value`的值会使输出产生不定状态。
- 由上面带斜率的波形可以看出，模块中`en`信号在时间`t`发生变化并开始影响`q`；若`en`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)

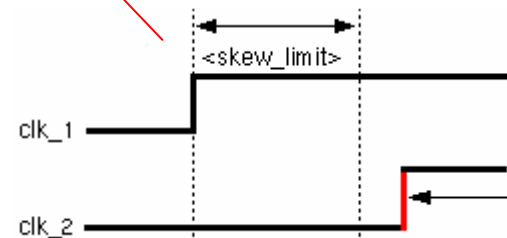
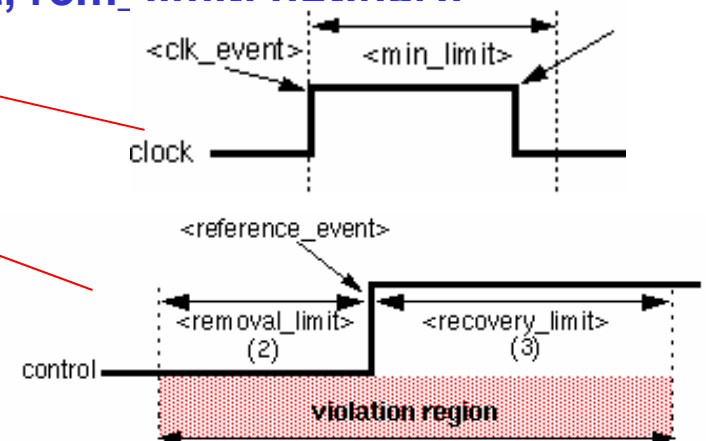
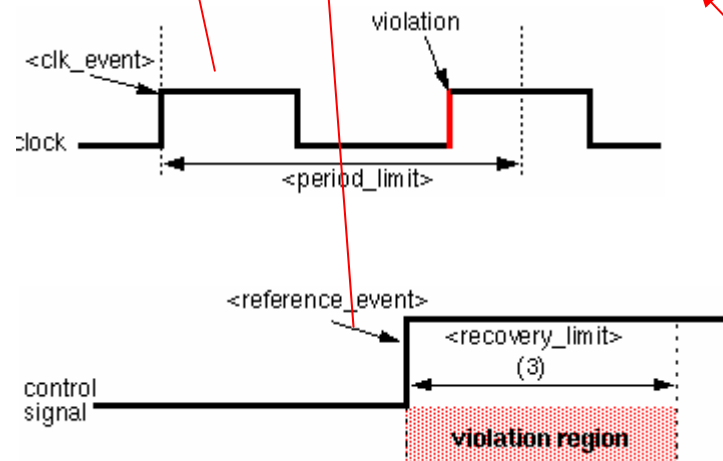
Verilog时序检查（续）

- 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 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 violation,如
\$hold(posedge clk, data, 3);
- \$setuphold**是**\$setup**和**\$hold**的联合。
\$setuphold(posedge clk, data, 4, 3);



Verilog时序检查（续）

- 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);`
- `$removal(ctrl_event1, ctrl_event2, limit, notifier);`
- `$recrem(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);`



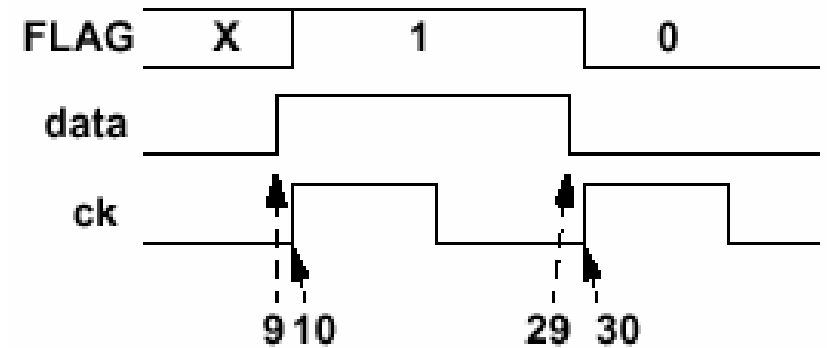
时序检查中的通知(notifier) (续)

- 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，不需要为**notifier**声明一个端口也可以对其进行操作。

notifier举例

```
`timescale 1ns/ 1ns
module dff_notifier (q, ck, d, rst);
    input ck, d, rst;
    output q;
    reg FLAG; // 1-bit notifier
    // dff netlist goes here
    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);
    // stimulus and response checking goes
    here
    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。

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

时序检查 — 条件时序检查

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。

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。

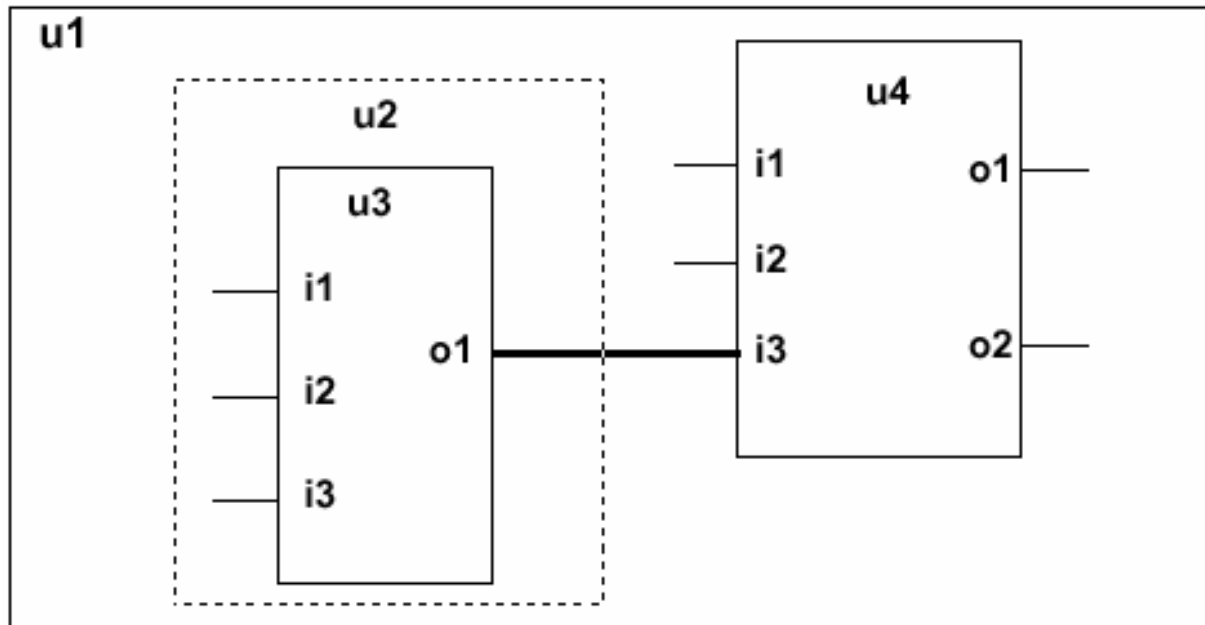
应注意的是，Verilog仿真器必须能够将SDF文件中的数据标注用于仿真。这些数据包括：

- 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。

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) )  
  )  
)
```


内部连接延时

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。

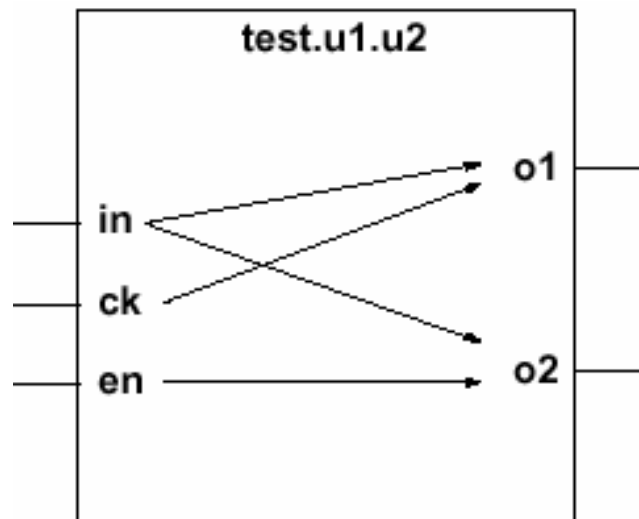
延时分上升、下降和关断延时，每种延时又有最好、典型和最坏值。

- 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) )
  )
)
```

IOPATH延时

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

注意：

SDF文件中的时序信息覆盖
specify块中的时序信息

复习

问题：

- 哪一种延时说明是最精确的？
- 怎样防止一个结构化零延时反馈？
- 为什么不能在**specify**块中使用模块参数(**parameter**)？
- **Verilog**仿真时在哪里说明一个线延时？
- **Verilog**仿真时在哪里说明时序检查？

解答：

- 在**specify**块中说明的延时信息是最精确的。因为可以说明**pin-to-pin**的路径延时
- 在门加上一个#延时来防止零延时组合逻辑反馈。
- 模块延时在每次实例化时都复制一份。而低层元件必须大量应用参数，并且其实例化次数也非常多。因此使用**specparam**节省了大量存储器。
- 在**SDF**文件中。在编译或仿真时标注这些延时
- 在**specify**块中描述，可以由**SDF**文件中的时序检修改或代替。

第九章 编译控制的使用

学习目标:

- 开发商提供的**Verilog**库
- 用**Verilog**库仿真
- **Verilog**源代码加密
- 其它仿真器相关的问题

术语及定义

- **PLI**: 编程语言接口, 基于C的过程访问Verilog数据结构
- **UDP**: 用户定义的基本单元, 用户定义的门级组合的及时序 的Verilog基本单元。
- **VHDL**: **VHSIC HDL**, 类似Ada的高级VLSI设计语言

Verilog模型库

开发商提供了大量的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开发商开发并提供工艺专用库
- 设计人员以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Ambit）
- 时序分析器（如Pearl）
- 故障仿真（Verifault-XL）

元件库建模

建立Verilog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**

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

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**

元件库建模（续）

库单元的特点：

- 每个库单元的描述在编译指导 *``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导`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。

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
```

```
module dff(...);
```

```
...
```

```
endmodule
```

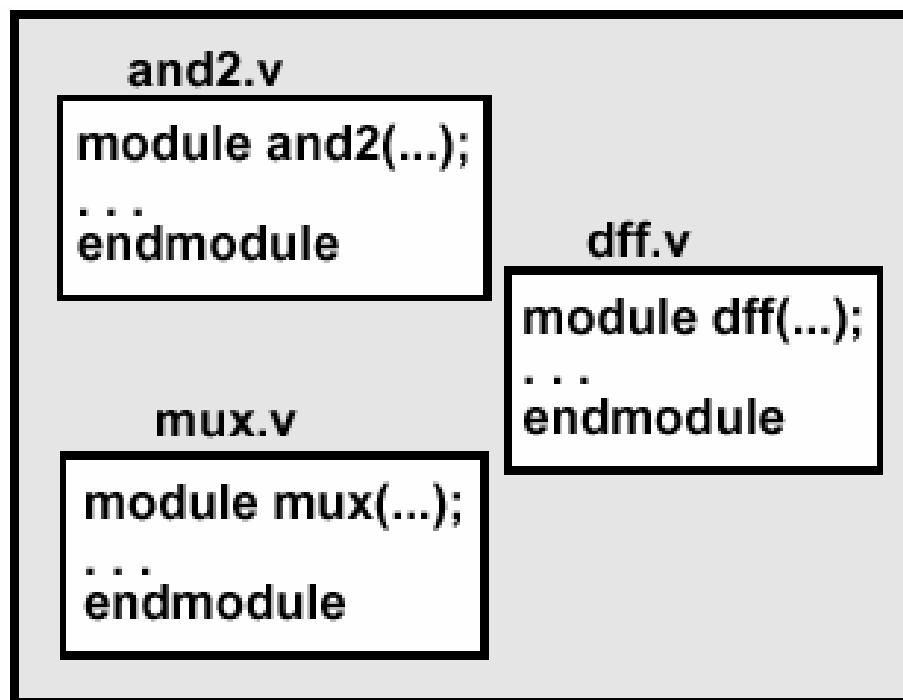
库目录扫描

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

Verilog模型库目录



编译指导`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/goo/libext=.v
```

编译指导`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

编译指导`uselib

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。

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

编译指导

尽管编译指导是**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`将编译指导变为缺省值。
- 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`编译指导时，使用`+incdir`命令行选项指定所包含文件的查找路径。

`+incdir+directory1+ directory2+... directoryN`

- 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。

编译指导

编译指导从出现时开始有效，直到被覆盖或使其失效。因此编译指导是全局的。

下列编译指导是**Verilog IEEE**标准中的：

<ul style="list-style-type: none">库单元分界 <code>`celldefine</code> <code>`endcelldefine</code>	<ul style="list-style-type: none">复合编译指导 <code>`default_nettype</code> <code>`include</code> <code>`unconnected_drive</code> <code>`nounconnected_drive</code> <code>`resetall</code> <code>`timescale</code>
<ul style="list-style-type: none">定义文本宏其和基于文本宏的转换 <code>`define</code> <code>`undef</code> <code>`ifdef</code> <code>`else</code> <code>`endif</code>	

定义文本宏

用**+define+**命令行参数定义文本宏

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

- 例子

```
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
```

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导指定单位延时、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，门延时为单位或零
- 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，因此使用编译指导影响设计中的指定块。

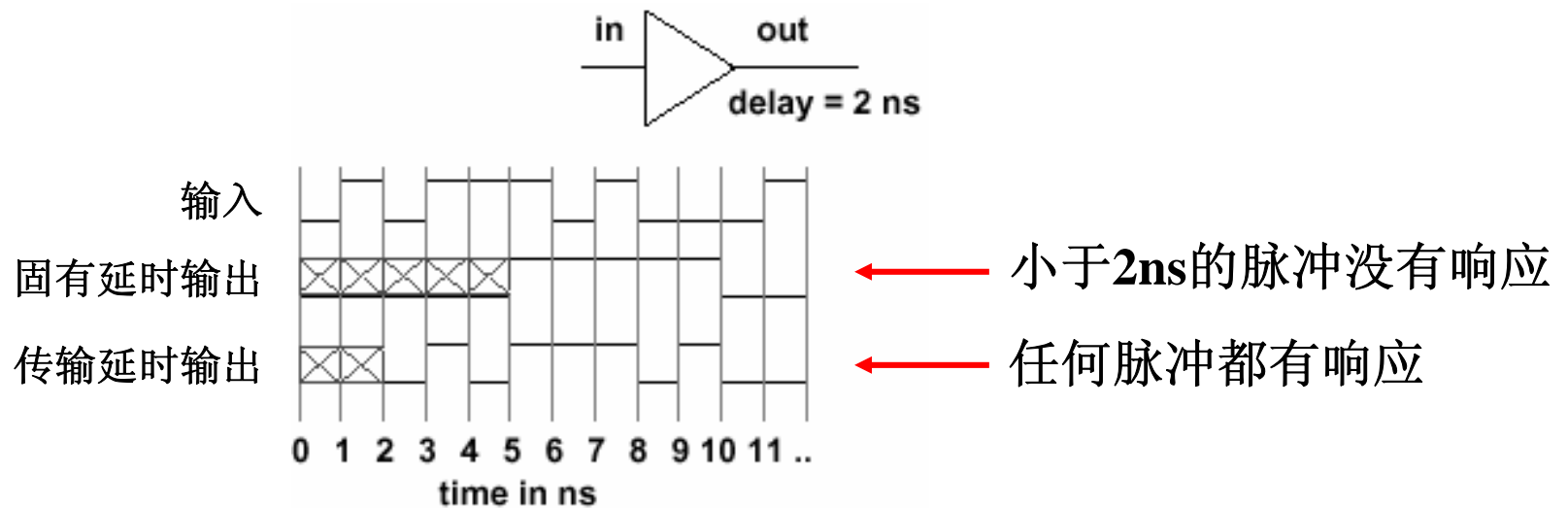
单位指的是时间精度**precision**

在设定单位延时和零延时模型时
不影响过程中时序控制

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

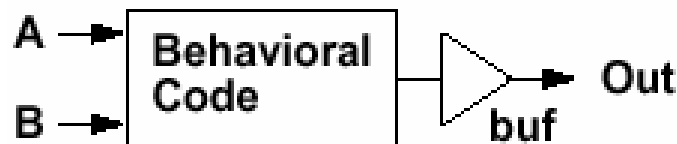


注意：记住使用`+pathpulse`用于路径延时控制

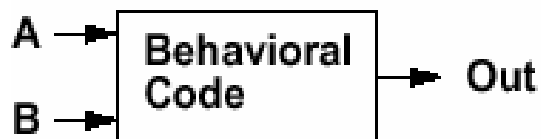
Verilog-XL路径延时的限制

模块路径延时由**specify**块说明。**Verilog-XL**中路径输出端必须是一个可以加速的基本单元。**NC Verilog**没有这个限制。

1. 路径输出端口必须由一个可加速的门驱动。



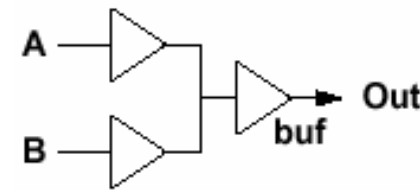
合法



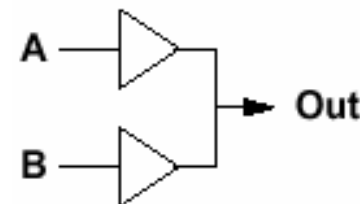
非法

解决方法：
在输出加基本单元buf

2. 输出端口在模块内部只能有一个驱动。



合法



非法

不是可加速基本单元或是行为描述时：

Error! Path delay output is not accelerated due to...

...[Verilog- PDOMBA< number>]

Error! Multiple path delays defined to node <path_ destination>;

Path delay outputs must have only one driver within the module

[Verilog- PDOMOD]

Verilog-XL可加速的对象

Verilog-XL可加速net和基本单元，但也有例外情况：

- 不能加速的net:

- 矢量net
- 具有非零延时的net
- 被force语句驱动或驱动过的net
- net被连续赋值（除非使用+caxl）

系统任务可以显示
所有forced net

- 不能加速的基本单元

- 双向基本单元tran tranif0 tranif1 rtran rtranif0 itranif1
- buf和not门有多个输出
- 基本单元的输入连接到寄存器数组的一位或一部分
- 基本单元的延时有非常数表达式
- 基本单元的一个输入是表达式（除非使用+caxl）
- 基本单元的不同延时超过65535

不能由XL算法加速，但可
以由Switch-XL加速

延时必须真正的不同。延时
#(3,4)和#(3,2)是不同的，而
#3和#(3,3)是相同的

- 系统任务\$shownonxl()寻找非XL结构。其参数设定一个模块的实例时查找该实例。若没有参数则在整个设计层次中查找。

用Verilog-XL加密源代码

用户可以保护设计中的所有权信息

- 有两种方法保护**Verilog**源描述：
 - 用**+autoprotect**命令行选项保护所有模块
 - 结合编译指导**`protect**和**`endprotect**，使用**+protect**选项保护所选模块或区域。
- 仿真器对源文件的加密是将**`protect**和**`endprotect**之间的源代码封装起来。
- 保护机制建立一个加密文件。仿真加密模块时，这些编译指导指示仿真器将源代码解密并仿真，但用户不能访问加密的数据结构。
- 可以用**Affirma Model Packager**保护私有模块用于**NC Verilog**和**NC VHDL**仿真。但对其它**Verilog**仿真器不会很理想。

保护所有Verilog源代码

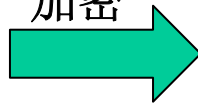
用**+autoprotect**命令行选项进行编译可以保护所有**Verilog**源代码。编译后建立一个新的只有模块名字可读的源文件。

verilog design.v +autoprotect

```
module AND2( a, b, c);  
    output a;  
    input b, c;  
    and a1( a, b, c);  
    specify  
        (b => a) = (1.2,2.0);  
        (c => a) = (1.4,1.8);  
    endspecify  
endmodule
```

design.v

加密



```
module AND2 `protected  
8Q@ RMSQH5DT^< oAXO3N^ VVhaD89ldTiYhe  
@n== 0Pi@ Q\ JifJ[ Z< ai766? ReA0RPaQ94H  
GHN60= Y[ KQgBdW4dO3662XYUXK=; CM= EVc  
cZo2@ 65`Lm< XL: 0VR[ CAUbShNPHne2  
IVFe3ZFa; pB5fO6kpT_< h$  
`endprotected endmodule
```

模块名不保护

design.vp

可以用**+autoprotect**选项指定输出文件扩展名

verilog design.v +autoprotect.port

输出的保护文件名为: **design.v.prot**

Verilog-XL缺省的输出文件名为原文件名后加“p”

保护选择的源描述

为了保护源描述的某一部分，用编译指导`protect和`endprotect将这部分包含起来。在编译源文件时使用命令行选项+protect。

verilog design.v +protect

```
module AND2( a, b, c);  
  output a;  
  input b, c;  
  `protect  
  and a1( a, b, c)  保护区域  
  specify  
    (b => a) = (1.2,2.0);  
    (c => a) = (1.4,1.8);  
  endspecify  
  `endprotect  
endmodule
```

不变

加密

```
module AND2( a, b, c);  
  output a;  
  input b, c;  
  `protected  
  RFdeISQV5DT^< ilbl= YI2AiTMmV  
  mKD^ Bg23WoqMIQbbREfQ`hfXF9S  
  G: M94J5E6SC079qZAZVgd$  
  `endprotected  
endmodule
```

design.vp

不保护模块名
及端口列表

提高了可
移植性

可以用+protect选项指定输出文件扩展名
verilog design.v +protect.prot
输出的保护文件名为: **design.v.prot**

在Verilog-XL中输入(import)VHDL模型

在**Verilog**设计中可以包含**VHDL**模型，并用**Cadence VHDL仿真器(leapfrog)**进行协同仿真。这称为**VHDL import**。

- 将**VHDL**模型进行预编译并放在Leapfrog的work库中。
- 执行vhdlshell产生每个**VHDL**模型的**verilog**包装(shell)。
- 确保从Leapfrog的license有效
- 执行**Verilog**仿真。它自动启动Leapfrog仿真器。

verilog包装的一个shell例子

```
module MULTIPLIER (IN1, IN2, OUT1, CLK)
  (* integer simulator = "Leapfrog";
   integer model = "WORKLIB.
MULTIPLIER";*)
  input [15: 0] IN1, IN2;
  input CLK;
  output [31: 0] OUT1;
endmodule
```

(**)中的命令是属性，是**Verilog**语言的一部分，还没有标准化。

在Verilog-XL中引入VHDL模型

从Leapfrog仿真器与常规的Leapfrog仿真器使用不同的license。如果从Leapfrog仿真器的license有效，并且设计中有VHDL模型，Verilog-XL自动启动、连接到从Leapfrog仿真器。Verilog-XL仿真器控制从仿真器何时启动，何时停止，并与之同步。

使用Leapfrog协同仿真时要注意以下几点：

- 对于Verilog-XL，VHDL模型是黑盒子。除能观察其端口外，不能观测其内部信号。为了能够观察其内部结构，需要在VHDL模型中设置附加的端口。这将影响仿真性能。
- shell模块中不能实例化其它模块
- shell中端口次序、大小、类型及指针方向必须与VHDL模型匹配。
- VHDL模块中不能实例化Verilog模块。
- 不能在引入的VHD模块之间传输双向信号
- 不能向VHDL inout端口传送Verilog的tran门信号。
- 所有Verilog模块必须有`timescale
- 使用Leapfrog协同仿真器时不能使用\$save和\$restart

使用INCA协同执行Verilog和VHDL

在Verilog设计中可以包含VHDL模型，使用NC仿真器协同执行。协同执行(co-execute)使用单一过程，比协同仿真(co-simulation)更有效。

- 用ncvhdl预编译VHDL模型
- 可以执行ncshell产生VHDL模型的Verilog封装。也可以直接实例化VHDL对象（不通过shell），通过使用VHDL定义的端口名称及次序，而不需要转换端口类型。
- 确保一个NC CoEx仿真器的license有效。
- 执行NC仿真，使用ncxlmode。

VHDL模型的Verilog封装的一个例子

```
module MULTIPLIER (IN1, IN2, OUT1, CLK)
  (* integer foreign = "VHDL( event) WORKLIB.MULTIPLIER"; *)
  input [15: 0] IN1, IN2;
  input CLK;
  output [31: 0] OUT1;
endmodule
```

使用INCA协同执行Verilog和VHDL(续)

CoEx选项的**license**不同于**NC Verilog**和**NC VHDL**仿真器。如果**license**有效，并且设计中有**VHDL**模型，仿真器自动协同执行。由于使用同一个过程，同步是非常有效的。

在协同执行时要注意下列几点：

- 当进入并观测实例模型内部时，必须遵循协同执行及**VHDL**的访问规则。
- **shell**端口次序、大小、类型及指针方向必须与**VHDL**模块匹配。
- 不能向**VHDL inout**端口传送**Verilog**的**tran**门信号。
- 所有**Verilog**模块必须有`**timescale**
- 协同执行时不能使用**\$save**、**\$restart**和**\$reset**。
- 可以在**Verilog**设计层次中引用层次名（使用**Verilog**模块外引用或**OOMR**）。这些引用可以穿过**VHDL**对象（但不能作为目标）。如果任何路径中的**VHDL**对象名称使用大写或非法字符，或**Verilog**关键字，必须使用**nmp**工具确定其**Verilog**名称。同样，必须注意将**VHDL**实例中指针(**n**)改变为**Verilog**指针[**n**]。

注意：可以使用**ncshell**工具产生**shell**以引入(**import**)**LMSFI**、**FMI**或**Swift**模型到**VHDL**设计中。产生的**shell**也可用于**Verilog-XL**或**Leapfrog**仿真。

小结

在本章我们学习了：

- 生产商提供的**Verilog**库
- 使用**Verilog**库仿真
- **Verilog**源代码加密
- 其它与仿真相关的话题

复习

问题：

- 当仿真器遇到编译指导`resetall时将所有编译指导置为缺省值吗？
- 使用什么选项指定库的名字？
- 如果仿真器没有在编译指导`uselib指定的库中找到实例的定义，它会去哪里寻找？

解答：

- 不是。当使用编译指导`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，使用编译指导`undef。
- 使用-v选项和/或-y及+libext+选项。
- 不会再去别的位置查找。

第10章 Verilog操作符

学习内容:

- 熟悉Verilog语言的操作符

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	{ } { { } }
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	 ? :

最高

↑

优先级

↓

最低

Verilog中的大小(size)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
1111
        b = 8; c = 8; // b = c = 1000
        #10 b = b + a; // 结果10111截断, b = 0111
        #10 c = c + a; // c = 10111
    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 fork
        #10 ans = five * int;    // ans = -15
        #20 ans = (int + 5)/ 2;  // ans = 1
        #30 ans = five/ int;    // ans = -1
        #40 num = rega + regb;  // num = 1101
        #50 num = rega + 1;    // num = 0100
        #60 num = int;        // num = 1101
        #70 num = regb % rega; // num = 1
        #80 $finish;
    join
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。

按位操作符

~	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 fork
        #10 num = rega & 0;    // num = 0000
        #20 num = rega & regb; // num = 1000
        #30 num = rega | regb;  // num = 1011
        #40 num = regb & regc; // num = 10x0
        #50 num = regb | regc;  // num = 1110
        #60 $finish;
    join
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 ();
    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 fork
        #10 ans = rega && 0;           // ans = 0
        #20 ans = rega || 0;           // ans = 1
        #30 ans = rega && five;         // ans = 1
        #40 ans = regb && rega;         // ans = 1
        #50 ans = regc || 0;           // ans = x
        #60 $finish;
    join
endmodule
```

逻辑反与位反的对比

! logical not 逻辑反
~ 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 fork
        #10 bit = ~rega; // num = 0100
        #20 bit = ~regb; // num = 1111
        #30 log = !rega; // num = 0
        #40 log = !regb; // num = 1
        #50 $finish;
    join
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;
    initial begin
        rega = 4'b0100;
        regb = 4'b1111;
    end
    initial fork
        #10 val = & rega ; // val = 0
        #20 val = | rega ; // val = 1
        #30 val = & regb ; // val = 1
        #40 val = | regb ; // val = 1
        #50 val = ^ rega ; // val = 1
        #60 val = ^ regb ; // val = 0
        #70 val = ~| rega; // (nor) val = 0
        #80 val = ~& rega; // (nand) val = 1
        #90 val = ^ rega && &regb; // val = 1
        $finish;
    join
endmodule
```


移位操作符

>> 逻辑右移
<< 逻辑左移

- 移位操作符对其左边的操作数进行向左或向右的位移操作。
- 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

<< 将左边的操作数左移右边操作数指定的位数

>> 将左边的操作数右移右边操作数指定的位数

在赋值语句中，如果右边(RHS)的结果：
位宽大于左边，则把最高位截去
位宽小于左边，则零扩展

```
module shift ();  
    reg [9: 0] num, num1;  
    reg [7: 0] rega, regb;  
    initial    rega = 8'b00001100;  
    initial fork  
        #10 num <= rega << 5 ; // num = 01_1000_0000  
        #10 regb <= rega << 5 ; // regb = 1000_0000  
        #20 num <= rega >> 3; // num = 00_0000_0001  
        #20 regb <= rega >> 3 ; // regb = 0000_0001  
        #30 num <= 10'b11_1111_0000;  
        #40 rega <= num << 2; //rega = 1100_0000  
        #40 num1 <= num << 2; //num1=11_1100_0000  
        #50 rega <= num >> 2; //rega = 1111_1100  
        #50 num1 <= num >> 2; //num1=00_1111_1100  
        #60 $finish;  
    join  
endmodule
```

左移先补后移

右移先移后补

建议：表达式左右位数一致

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。

```
module relationals ();
    reg [3: 0] rega, regb, regc;
    reg val;
    initial begin
        rega = 4'b0011;
        regb = 4'b1010;
        regc = 4'b0x10;
    end
    initial fork
        #10 val = regc > rega ; // val = x
        #20 val = regb < rega ; // val = 0
        #30 val = regb >= rega ; // val = 1
        #40 val = regb > regc ; // val = 1
        #50 $finish;
    join
endmodule
```

rega和regc的关系取决于x

无论x为何值，
regb>regc

相等操作符

二 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

Case等只能用于行为描述，不能用于RTL描述。

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 fork  
        #10 val = rega == regb ; // val = 0  
        #20 val = rega != regc;  // val = 1  
        #30 val = regb != regc;  // val = x  
        #40 val = regc == regc;  // val = x  
        #50 $finish;  
    join  
endmodule
```

相等操作符

===

相同(case等)

!==

不相同(case不等)

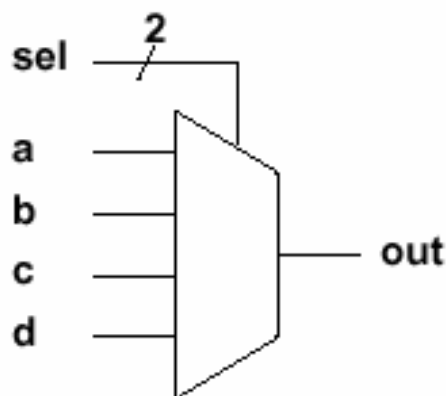
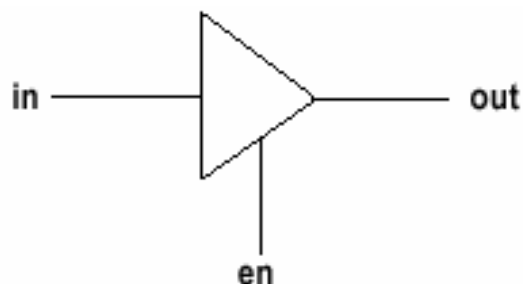
- 其结果是1'b1、1'b0或1'bx。
- 如果左边及右边的值相同（包括x、z），则结果为1。
- 如果左边及右边的值不相同，则结果为0。
- !==的结果与 === 相反

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 fork  
        #10 val = rega === regb ; // val = 0  
        #20 val = rega !== regc;  // val = 1  
        #30 val = regb === regc;  // val = 0  
        #40 val = regc === regc;  // val = 1  
        #50 $finish;  
    join  
endmodule
```

条件操作符

?: 条件



```
module likebufif( in, en, out);  
    input in;  
    input en;  
    output out;  
    assign out = (en == 1) ? in : 'bz;  
endmodule
```

```
module like4to1( a, b, c, d, sel, out);  
    input a, b, c, d;  
    input [1: 0] sel;  
    output 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

条件操作符

条件操作符的语法为：

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

其意思是：if condition is TRUE, then LHS=true_expression, else 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。

最后一个操作数作为缺省值。

registger = 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，则输出不确定值。

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

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，必须指定位数，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。
在操作符符号{ }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 fork
        #10 new = {regc[ 4: 3], regd[ 7: 5],
                    regb[ 2], rega[ 1: 0]};
        // new = 8'b11111111
        #20 $finish;
    join
endmodule
```


复制

{ } 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 = 11111111
        // regb is replicated 4 times.
        #20 bus <= { {2{ regb}}, {2{ regc}} };
        // bus = 11110000. regc and regb are each
        // replicated, and the resulting vectors
        // are concatenated together
        #30 bus <= { {4{ rega[1]}}, rega };
        // bus = 00001001. rega is sign-extended
        #40 $finish;
    join
endmodule
```

复习

问题:

- 1、~ 和 ! 有什么不同?
- 2、&& 和 & 有什么不同?
- 3、用复制操作符复制一个数据时，对数据有什么要求?

解答

- 1、~ 进行1的补码操作，将矢量中的每一位取反
! 将一个操作数归约为一位true或false结果
- 2、& 将操作数从低到高的对应位的进行与操作
&& 将每个操作数归约为一位true或false，然后对归约结果进行与操作
- 3、要复制的操作数必须指定位数，例如
{3{'b1}}是非法的，因此'b1没有指定位数。而{3{1'b1}}是合法的

第11章 行为建模

学习内容:

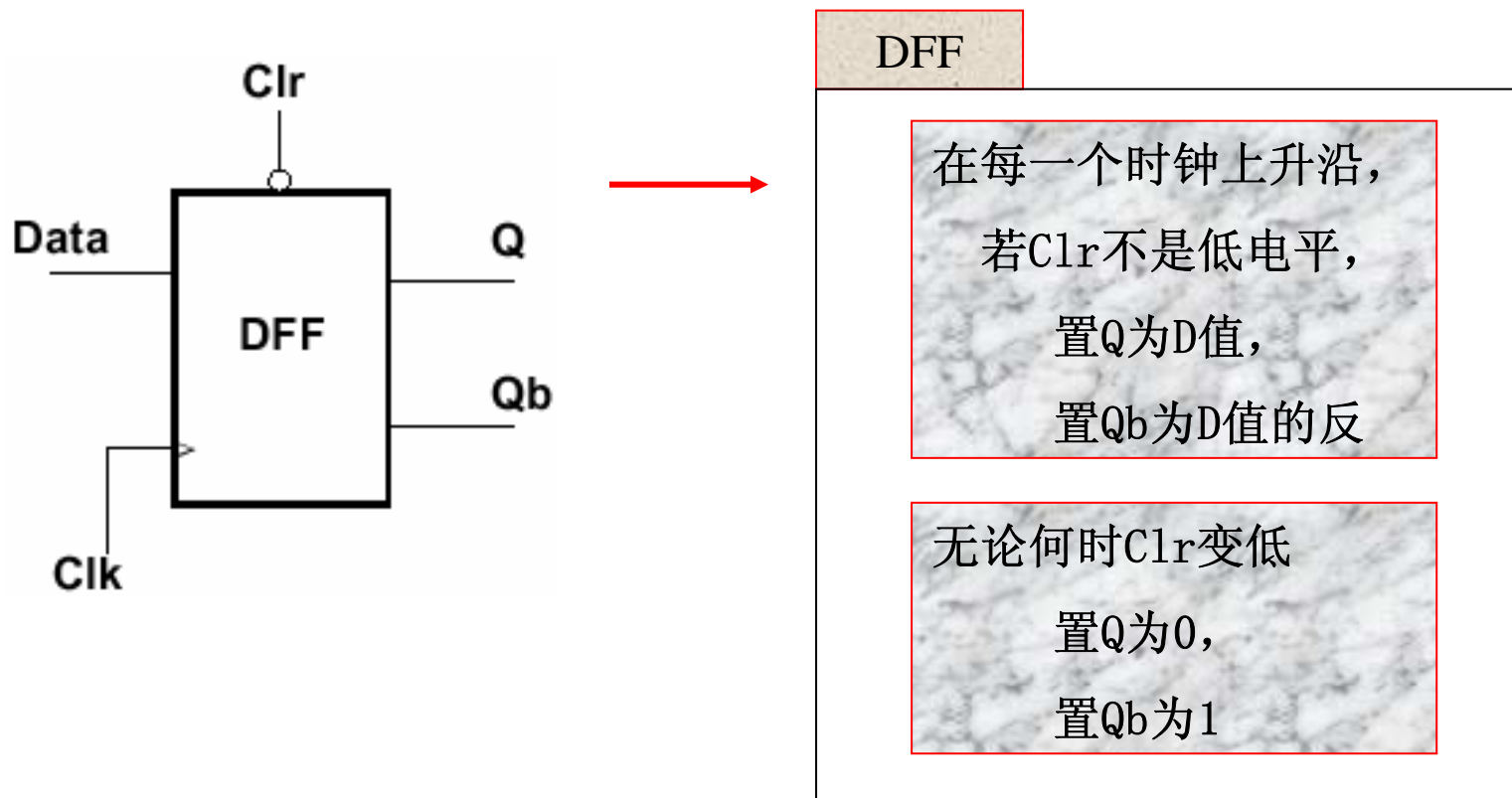
- 行为建模的基本概念
- Verilog中高级编程语言结构
- 如何使用连续赋值

注:

RTL描述方式是行为描述方式的子集。在本章中的综合部分将详细介绍哪些行为级结构同样可以用于RTL描述。

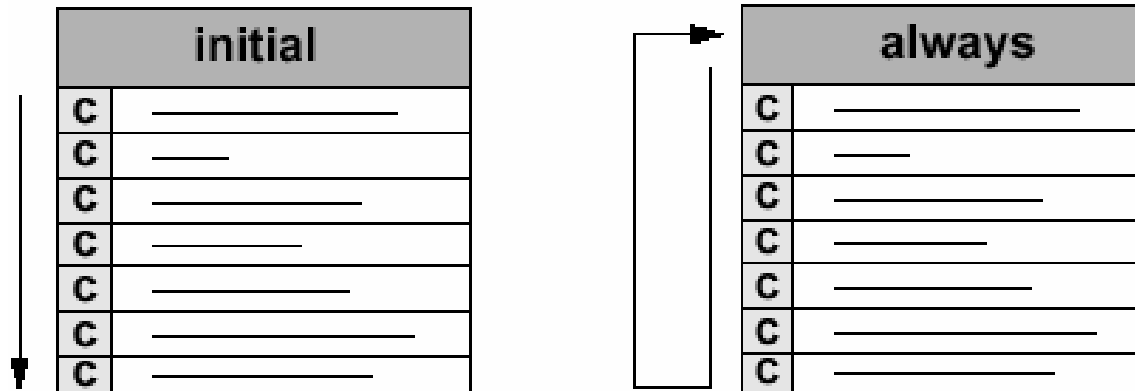
行为描述

- 行为级描述是对系统的高抽象级描述。在这个抽象级，注重的是整个系统的功能而不是实现。
- Verilog有高级编程语言结构用于行为描述，包括：
wait, while, if then, case和forever
- Verilog的行为建模是用一系列以高级编程语言编写的并行的、动态的过程块来描述系统的工作。



过程(procedural)块

- 过程块是行为模型的基础。
- 过程块有两种：
 - **initial**块，只能执行一次
 - **always**块，循环执行
- 过程块中有下列部件
 - 过程赋值语句：在描述过程块中的数据流
 - 高级结构（循环，条件语句）：描述块的功能
 - 时序控制：控制块的执行及块中的语句。



过程赋值(procedural assignment)

- 在过程块中的赋值称为过程赋值。
- 在过程赋值语句中表达式**左边**的信号必须是**寄存器类型**（如**reg**类型）
- 在过程赋值语句等式**右边**可以是任何有效的表达式，**数据类型也没有限制**。
- 如果一个信号没有声明则**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 (out, a, b, cin);  
    input a, b, cin;  
    output [1:0] out;  
    wire a, b, cin;  
    reg half_sum;  
    reg [1: 0] out;  
    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
```

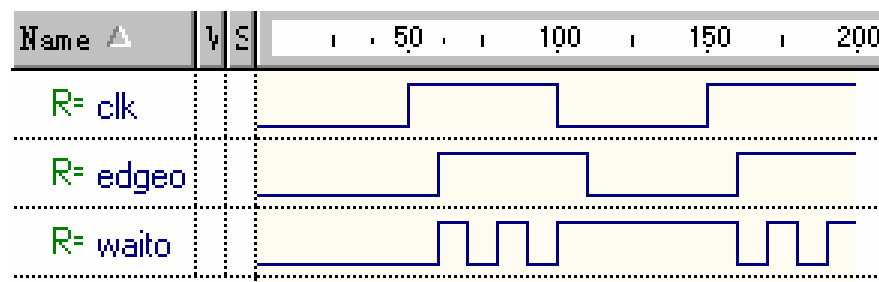
half_carry
没有声明

过程时序控制

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**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  
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 (out, a, b, sl);
    input a, b, sl;
    output out;
    reg out;
    always @( sl or a or b)
        if (! sl)
            #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 (clk);
    output clk;
    reg clk;
    parameter cycle = 20;
    initial clk = 0;
    always
        #(cycle/2) clk = ~clk;
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`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 @( negedge clk) // 在clk下降沿执行  
        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。

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 (out, a, b, enable);  
    input enable;  
    input [2: 0] a, b;  
    output [3: 0] out;  
    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。

命名事件(named event)

在行为代码中定义一个命名事件可以触发一个活动。命名事件不可综合。

```
module add_mult (out, a, b);
  input [2: 0] a, b;
  output [3: 0] out;
  reg [3: 0] out;
  /**define events**
  event add, mult;
  always@ (a or b)
    if (a > b)
      -> add; // *** trigger event ***
    else
      -> mult; // *** trigger event ***
  // *** respond to an event trigger ***
  always @( add)
    out = a + b;
  // *** respond to an event trigger ***
  always @( mult)
    out = a * b;
endmodule
```

在例子中，事件**add**和**mult**不是端口，但定义为事件，它们没有对应的硬件实现。

- 是一种数据类型，能在过程块中触发一个使能。
- 在引用前必须声明
- 没有持续时间，也不具有任何值
- 只能在过程块中触发一个事件。
- **->**操作符用来触发命名事件。

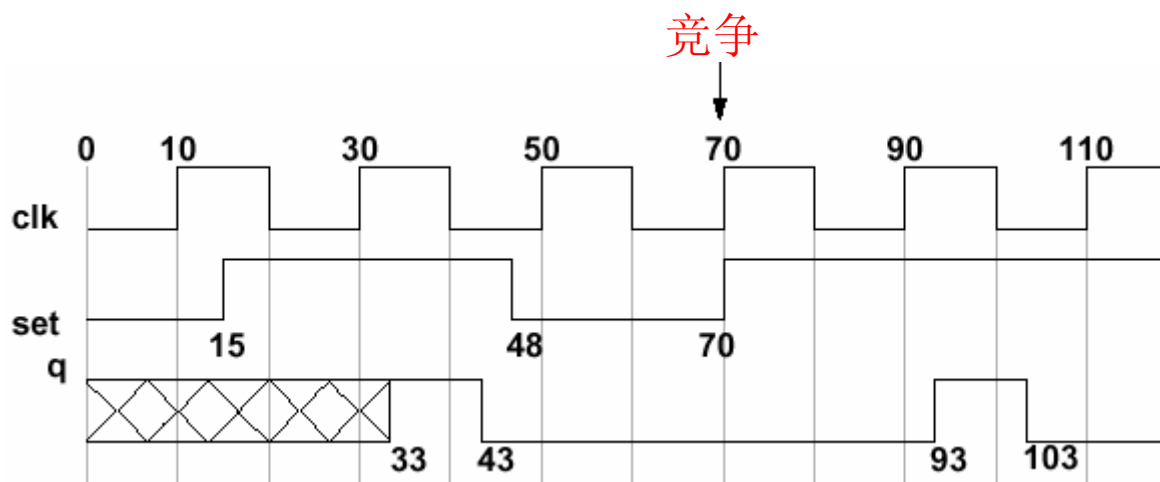
a大于**b**，事件**add**被触发，控制传递到等待**add**的**always**块。

如果**a**小于或等于**b**，事件**mult**被触发，控制被传送到等待**mult**的**always**块。

行为描述举例

```
always wait (set)
begin
    @( posedge clk) #3 q = 1;
                    #10 q = 0;

    wait (! set);
end
```



在上面的例子中发生下面顺序的事件：

1. 等待set=1，忽略时刻10的clk的posedge。
2. 等待下一个clk的posedge，它将在时刻30发生。
3. 等待3个时间单位，在时刻33（30+3）置q=1。
4. 等待10个时间单位，在时刻43（33+10）置q=0。
5. 等待在时刻48发生的set=0。
6. 等待在时刻70发生且与clk的上升沿同时发生的set=1。
7. 等待下一个上升沿。时刻70的边沿被忽略，因为到达该语句时时间已经过去了，如例子所示，clk=1。

重要内容：在实际硬件设计中，事件6应该被视为一个竞争（**race condition**）。在仿真过程中，值的确定依赖于顺序，所以是不可预测的。这是不推荐的建模类型。

RTL描述举例

下面的RTL例子中只使用单个边沿敏感时序控制。

```
module dff (q, qb, d, clk);  
    output q, qb;  
    input d, clk;  
    reg q, qb;  
    always @( posedge clk)  
    begin  
        q = d;  
        qb = ~d;  
    end  
endmodule
```

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。

always		c
begin		
c	_____	
c	_____	
c	_____	
c	_____	
c	_____	
c	_____	
c	_____	
end		

always		c
fork		
c	_____	
c	_____	
c	_____	
c	_____	
c	_____	
c	_____	
c	_____	
join		

initial		c
begin		
c	_____	
c	_____	
c	_____	
c	_____	
c	_____	
c	_____	
c	_____	
end		

initial		c
fork		
c	_____	
c	_____	
c	_____	
c	_____	
c	_____	
c	_____	
c	_____	
join		

- **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。

块语句（续）

- 在顺序块中，语句一条接一条地计算执行。
- 在并行块中，所有语句在各自的延迟之后立即计算执行。

```
begin
```

```
    #5 a = 3;
```

```
    #5 a = 5;
```

```
    #5 a = 4;
```

```
end
```

```
fork
```

```
    #5 a = 3;
```

```
    #15 a = 4;
```

```
    #10 a = 5;
```

```
join
```

上面的两个例子在功能上是等价的。**Fork-join**例子中的赋值故意打乱顺序是为了强调顺序是没有关系的。

注意**fork-join**块是典型的不可综合语句，并且在一些仿真器时效率较差。

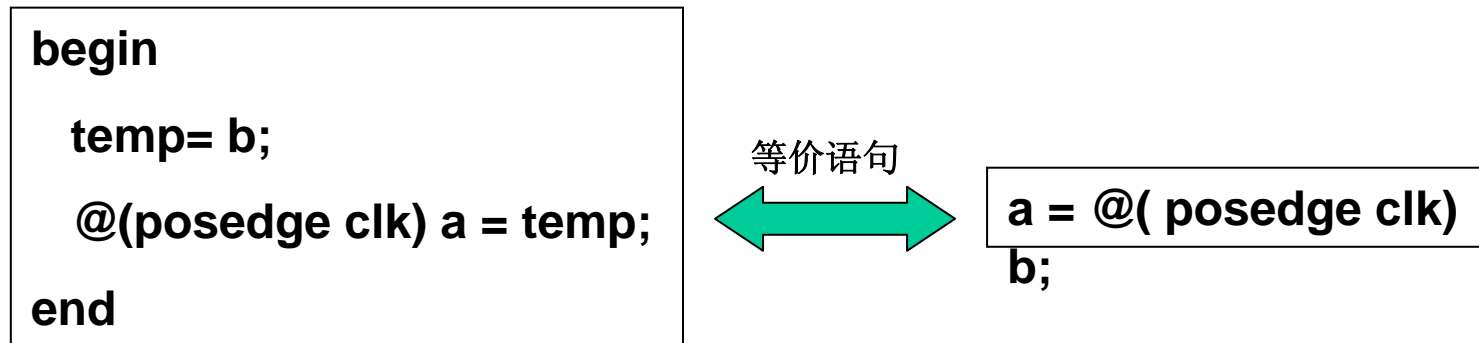
延迟赋值语句

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。



LHS: Left-hand-side

RHS: Right-hand-side

延迟赋值语句

并行语句在同一时间步发生，但由仿真器在另外一个时间执行。

在下面的每个例子中，**a**和**b**的值什么时候被采样？

在下面的每个例子中，什么时候给**a**和**b**赋值？

b值拷贝到a然后回传

```
begin
  a = #5 b;
  b = #5 a;
  #10 $display(a, b);
end
```

a和b值安全交换

```
fork
  a = #5 b;
  b = #5 a;
  #10 $display(a, b);
join
```

在左边的例子中，**b**的值被立即采样（时刻0），这个值在时刻5赋给**a**。**a**的值在时刻5被采样，这个值在时刻10赋给**b**。

注意，另一个过程块可能在时刻0到时刻5之间影响**b**的值，或在时刻5到时刻10之间影响**a**的值。

在右边的例子中，**b**和**a**的值被立即采样（时刻0），保存的值在时刻5被赋值给他们各自的目标。这是一个安全传输。

注意，另一个过程块可以在时刻0到时刻5之间影响**a**和**b**的值。

非阻塞过程赋值

```
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。

非阻塞过程赋值（续）

阻塞与非阻塞赋值语句行为差别举例1

```
module non_block1;
    reg a, b, c, d, e, f;
    initial begin // blocking assignments
        a = #10 1; // time 10
        b = #2 0; // time 12
        c = #4 1; // time 16
    end
    initial begin // non- blocking assignments
        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
```

非阻塞过程赋值（续）

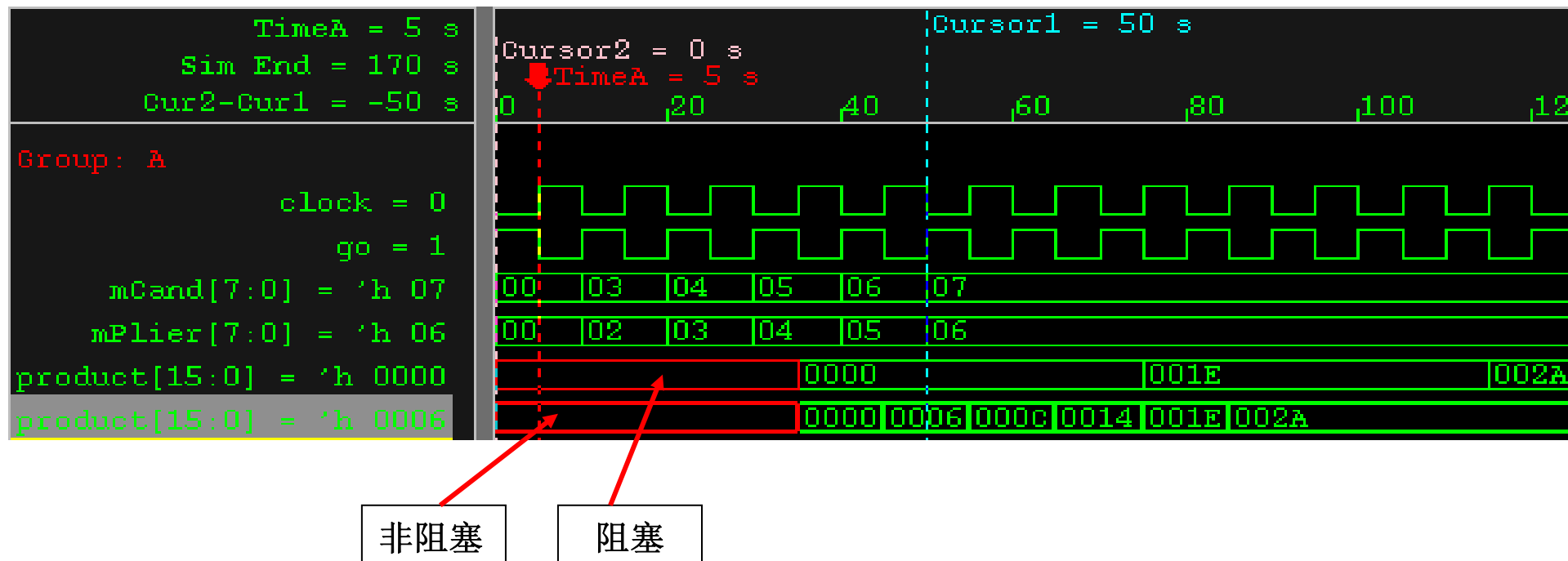
阻塞与非阻塞赋值语句行为差别举例2

```
module pipeMult(product, mPlier, mCand, go, clock);  
input      go, clock;  
input [7:0] mPlier, mCand;  
output [15:0] product;  
reg [15:0] product;  
always @(posedge go)  
    product = repeat (4) @(posedge clock) mPlier * mCand;  
endmodule
```

```
module pipeMult(product, mPlier, mCand, go, clock);  
input      go, clock;  
input [7:0] mPlier, mCand;  
output [15:0] product;  
reg [15:0] product;  
always @(posedge go)  
    product <= repeat (4) @(posedge clock) mPlier * mCand;  
endmodule
```

非阻塞过程赋值（续）

阻塞与非阻塞赋值语句行为差别举例2波形

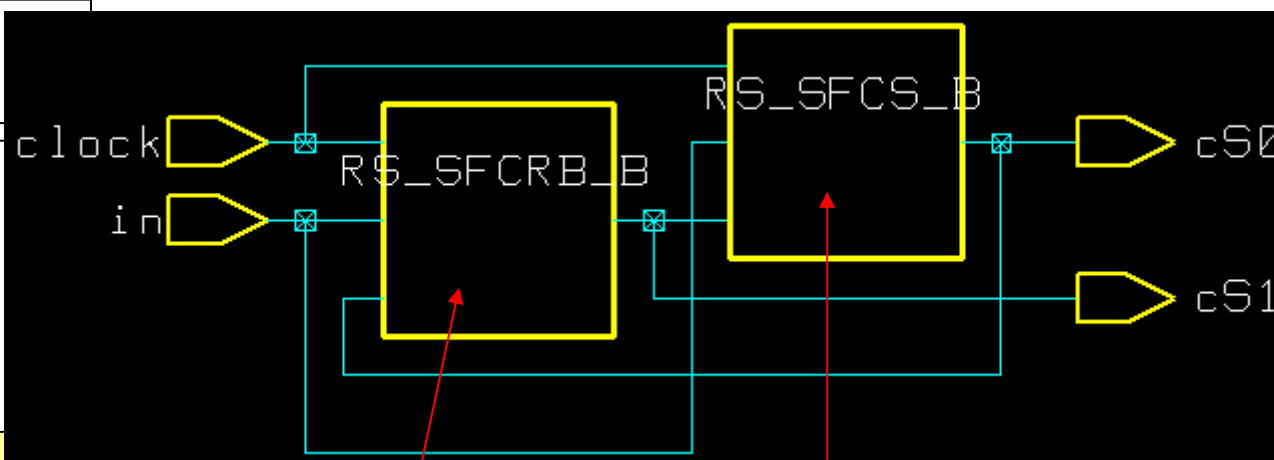
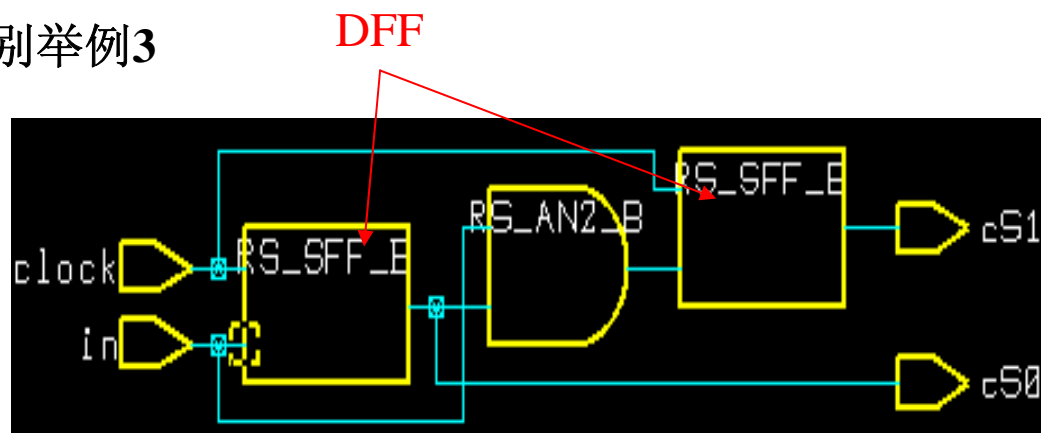


非阻塞过程赋值（续）

阻塞与非阻塞赋值语句行为差别举例3

```
module fsm(cS1, cS0, in, clock);  
input in , clock;  
output cS1, cS0;  
reg cS1, cS0;  
always @(posedge clock) begin  
    cS1 = in & cS0; //同步复位  
    cS0 = in | cS1; //cS0 = in  
end  
endmodule
```

```
module fsm(cS1, cS0, in, clock);  
input in , clock;  
output cS1, cS0;  
reg cS1, cS0;  
always @(posedge clock) begin  
    cS1 <= in & cS0; //同步复位  
    cS0 <= in | cS1; //同步置位  
end  
endmodule
```



同步复位DFF

同步置位DFF

非阻塞过程赋值（续）

举例4：非阻塞赋值语句中延时在左边和右边的差别

```
module exchange;
    reg[3:0] a, b;
    initial begin
        a=1; b=4;
        #2 a=3; b=2;
        #20 $finish;
    end
    initial
    $monitor($time, "\t%h\t%h", a, b);
    initial begin
        #5 a <= b;
        #5 b <= a;
    end
end
endmodule
```

time	a	b
0	1	4
2	3	2
5	2	2

```
module exchange;
    reg[3:0] a, b;
    initial begin
        a=1; b=4;
        #2 a=3; b=2;
        #20 $finish;
    end
    initial
    $monitor($time, "\t%h\t%h", a, b);
    initial begin
        a <= #5 b;
        b <= #5 a;
    end
end
endmodule
```

time	a	b
0	1	4
2	3	2
5	4	1

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 (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）

*case*语句:

```
module compute (result, rega, regb, opcode);
input [7: 0] rega, regb;
input [2: 0] opcode;
output [7: 0] result;
reg [7: 0] result;
always @( rega or regb or opcode)
    case (opcode)
        3'b000 : result = rega + regb;
        3'b001 : result = rega - regb;
        3'b010 , // specify multiple cases with the same
result
        3'b100 : result = rega / regb;
        default : begin
            result = 'bx;
            $display (" no match");
        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。

条件语句-case语句

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一个多路条件语句。

- **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**表达式或**case**项中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语法：

```
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;
```

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>



综合工具
还不支持

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

```
// Parameterizable shift and add multiplier
module multiplier( result, op_a, op_b);
    parameter size = 8;
    input [size:1] op_a, op_b;
    output [2* size:1] result;
    reg [2* size:1] shift_opa, result;
    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
end
endmodule
```

为什么要说明一个
shift_opb变量?

循环(looping)语句

while: 只要表达式为真(不为0), 则重复执行一条语句(或语句块)

```
...  
reg [7: 0] tempreg;  
reg [3: 0] count;  
...  
    count = 0;  
    while (tempreg) // 统计tempreg中 1 的个数  
    begin  
        if (tempreg[ 0]) count = count + 1;  
        tempreg = tempreg >> 1; // Shift right  
    end  
end  
...
```

循环(looping)语句

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  
    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。

循环(looping)语句

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。

```
// X检测
```

```
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;
```

行为级零延时循环

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，使仿真器停滞在那个时片。

在下面的例子中，对事件进行了仿真但仿真时间不会推进。当**always**块和**forever**块中没有时序控制时就会发生这种事情。

```
module comparator( out, in1, in2);  
  output [1: 0] out;  
  input [7: 0] in1, in2;  
  reg [1: 0] out;  
  always  
    if (in1 == in2)  
      out = 2'b00;  
    else if (in1 > in2)  
      out = 2'b01;  
    else  
      out = 2'b10;  
  initial  
    #10 $finish;  
endmodule
```


持续赋值(continuous assignment)

- 可以用持续赋值语句描述组合逻辑，代替用门及其连接描述方式。
- 持续赋值在**过程块外部使用**。
- 持续赋值用于**net**驱动。
- 持续赋值只能在等式左边有一个简单延时说明。
 - 只限于在表达式左边用**#delay**形式
- 持续赋值可以是显式或隐含的。

语法：

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

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

持续赋值(continuous assignment)(续)

持续赋值的例子

```
module assigns (o1, o2, eq, AND, OR, even, odd, one, SUM, COUT,
               a, b, in, sel, A, B, CIN);
    output [7:0] o1, o2;
    output [31:0] SUM;
    output eq, AND, OR, even, odd, one, COUT;
    input a, b, CIN;
    input [1:0] sel;
    input [7:0] in;
    input [31:0] A, B;

    wire [7:0] #3 o2;           // 没有说明，但设置了延时
    tri AND = a&b, OR = a|b; // 两个隐含赋值
    wire #10 eq = (a == b);    // 隐含赋值，并说明了延时
    wire [7:0] (strong1, weak0) #(3,5,2) o1 = in; // 强度及延时
    assign o2[7:4] = in[3:0], o2[3:0] = in[7:4]; // 部分选择
    tri #5 even = ^in, odd = ~^in; // 延时，两个赋值
    wire one = 1'b1; // 常数赋值
    assign {COUT, SUM} = A + B + CIN; // 给级联赋值
endmodule
```

持续赋值(continuous assignment)(续)

in的值赋给**o1**，但其每位赋值的强度及延迟可能不同。如果**o1**是一个标量(**scalar**)信号，则其延迟和前面的条件缓冲器上的门延迟相同。对向量线网(**net**)的赋值上的延迟情况不同。**0**赋值使用下降延迟，**Z**赋值使用关断延迟，所有其他赋值使用上升延迟。

上面的例子显示出持续赋值的灵活性和简单性。持续赋值可以：

- 隐含或显式赋值
- 给任何**net**类型赋值
- 给矢量**net**的位或部分赋值
- 设置延时
- 设置强度
- 用级联同时给几个**net**类变量赋值
- 使用条件操作符
- 使用用户定义的函数的返回值
- 可以是任意表达式，包括常数表达式

持续赋值(continuous assignment)(续)

使用条件操作符的例子：

```
module cond_assigns (MUX1, MUX2, a, b, c, d);  
    output MUX1, MUX2;  
    input a, b, c, d;  
    assign MUX1 = sel == 2'b00 ? a :  
                sel == 2'b01 ? b :  
                sel == 2'b10 ? c : d;  
    tri1 MUX2 = sel == 0 ? a : 'bz, MUX2 = sel == 1 ? b : 'bz,  
                MUX2 = sel == 2 ? c : 'bz, MUX2 = sel == 3 ? d : 'bz;  
endmodule
```

- 从上面的例子可以看出，持续赋值的功能很强。可以使用条件操作符，也可以对一个net多重赋值(驱动)。
- 在任何时间里只有一个赋值驱动MUX2到一个非三态值。如果所有驱动都为三态，则mux2缺省为一个上拉强度的1值。

复习

问题:

- 在哪里放置**always**块? **forever**循环呢?
- 持续赋值通常给哪种类型的逻辑建模?
- 在**begin...end**之间使用非阻塞赋值和**fork...join**块有哪些区别?
- **Verilog**中**posedge**是什么意思?
- **Verilog**中存在哪些条件结构?

解答:

- **always**是块语句, 在**module**内使用。
forever循环是在过程块、**task**或**function**内使用。
- 持续赋值只能给组合逻辑建模, 因为他们只包含简单延迟。过程块可以包含@和**wait**时序控制。
- **fork...join**块
 - 可以使用等式左边延迟
 - 可以包含并行语句(循环, 条件语句, 任务, 系统任务, 事件触发器), 不仅仅是并行赋值。
 - 是典型的不可综合语句
- **posedge**是任何可能从低到高的跳变 (**0->1,0->z,0->x,z->x,x->z,z->1,x->1**)。
- **Case**、**if-else**语句以及**?:**条件操作符。

第12章 TUI调试

学习内容:

在本章中将学习用**Verilog-XL TUI**(Textual User Interface)和**NC Verilog TUI**调试

- 进入交互式仿真模式
- 控制并观察仿真
- 浏览设计层次
- 检查 (**checkpointing**) 和退出仿真
- 对设计进行临时修补
- 动态的单步及跟踪仿真
- 使用命令历史列表

术语及定义

- **SHM** 仿真历史管理器(**Simulation History Manager**)。一个管理由**SimWave**显示的仿真对象值数据跳变的工具
- **CLI** **Verilog-XL**命令行界面(**command line interface**)，通过它你可以控制仿真并对**Verilog**过程语句执行调试操作
- **Tcl** 工具命令语言 (**Tool Command Language**)。用于对交互式程序提出命令的脚本语言
- **VCD** (**Value Change Dump**)。存储对象值跳变数据的文件格式

纵览

- 什么是**CLI(Command Line Interface)**?
 - 一个允许输入**Verilog HDL**过程命令的**Verilog-XL**仿真器的TUI
 - **CLI**命令是一个**Verilog**过程语句或语句块。**Verilog**过程语句包括过程赋值，循环，条件语句和任务以及功能调用。可以在一行里输入多个**Verilog**语句，语句之间由一个分号他开，如同在源代码中那样。
 - **Verilog-XL**有源代码调试命令，这些命令不在**IEEE**规范中，可以在**Verilog**描述中使用，但这不很必要。
- 什么是**Tcl (Tool Command Language)**
 - 一个对许多软件工具包括**NC Verilog**的文本用户界面。
 - **NC Verilog**仿真器的面向对象的TUI。
 - 一个**Tcl**命令包括一个或多个字（命令名，后面是命令的**argument**）。字之间由空格或**tab**分隔。可以在一个命令行中输入多个命令，中间用分号分开。
 - 可以从网上、技术参考书或图书馆得到标准**Tcl**命令的信息。
 - **NC Verilog**有专用标准**Tcl**命令集扩展用于设计调试(在**ncsim**命令窗口输入)。
 - 本章将对大部分**NC Verilog**专用扩展作简单描述。联机文档中有更详细解释。
- **CLI**和**Tcl**命令可以在命令行交互式输入，也可以由源脚本或**keyfile**输入。
- 注意：单击工具按钮或选择菜单时，**SimVision GUI**自动发出**CLI**和**Tcl**命令作出响应。
- 在本章中将通过实际调试**CLI**和**Tcl**的实例来学习二者的界面。

进入交互模式

有三种方法中断仿真，进入交互模式：

- 使用**-s**命令行选项在仿真前（时间**0**）停止仿真，立即进入交互模式
- 在仿真过程中输入一个**^C**异步中断
- 到达一个断点或在源代码里的**\$stop**系统任务。

当中断**Verilog-XL**时，仿真器进入交互式模式并给出提示符：

C1 >

当中断**NC Verilog**时，进入交互式模式并给出提示符：

ncsim >

- 此时，仿真器暂时挂起。可以在命令行提示符处输入交互式命令，然后继续仿真。

进入交互模式

仿真器允许在离散的时间点中断仿真并与设计进行交流。有三种方法进入交互模式：

- 使用-s命令行选项在时间0停止仿真
 - 输入一个^C异步中断
 - 可以用测试基准中的\$stop系统任务使仿真在指定的时间（或基于一个指定的事件）进入交互式模式。
- 当仿真器被中断时，它进入交互式模式并给出输入命令的提示符。
 - 在Verilog-XL CLI中，可以输入任何可以放在一个过程块内的语句，并输入一些只用于调试环境的特殊命令。
 - 在NC Verilog Tcl界面中，可以输入标准Tcl命令，Tcl的NC Verilog扩展，或将被传送到操作系统的shell命令。
 - 仿真只是暂时挂起。所有信号保持他们当前状态直到继续仿真。

退出仿真

如果以**batch**模式（没有停止或进入交互式模式）运行仿真，则当仿真器在源代码中遇到一个**\$finish**时，仿真退出。

在交互式模式中，**Verilog-XL**和**NC Verilog**有不同工作方式：

- 在**Verilog-XL**中，退出仿真的方式有：
 - 在交互式窗口中输入**\$finish**；或**\$finish[0|1|2]**。可以提供一个参数显示仿真时间和存储器/**CPU**使用统计。
 - 在交互式窗口中按[^]**D**。
 - 仿真时在遇到源代码中的一个**\$finish**。
- 在**NC Verilog**中，仿真退出有下列途径：
 - 在交互式提示符输入**\$finish[0|1|2]**
 - 在交互式提示符输入**exit**
 - 在交互式提示符按连续按[^]**D**两次

注意：在**NC Verilog**中，一旦进入了交互式模式，如果继续仿真并在源代码中遇到一个**\$finish**，则返回到交互式模式。

注：\$finish[0|1|2]：0：没有输出；1：输出仿真时间和位置；2：输出仿真时间和位置；输出仿真时使用的存储器大小和**CPU**时间。

退出仿真

如果在一个零延迟循环中或如果有太多仿真事件被列入队列以至系统不能及时对一个键盘中断做出响应，你可以终止仿真器进程来停止仿真。如果仿真继续运行，它将用去越来越多的存储空间。在NC Verilog中，你可以选择重复使用^C（5或6次）直到仿真停止。

可以发送KILL信号给一个UNIX进程来终止它，典型为^|，或从另一个shell进程，输入：

```
kill -9 pid
```

这里pid为仿真器进程的进程ID。

可以用Task Manager终止一个NT进程。

用**finish**命令或**\$finish**系统任务来退出仿真并返回控制到操作系统，可选择地显示仿真时间和存储器/CPU使用统计。

用Verilog-XL调试

下面是IEEE标准Verilog过程命令，在调试时特别有用：

force	release	\$display	\$finish	\$history	[\$inc]save
\$input	\$list	\$monitor	[\$real]time	\$reset	\$restart
\$scope	\$showscopes	\$showvars	\$stop	\$strobe	

Verilog-XL还提供了其它一些在调试时比较有用的命令

\$db_break[after before]time	[\$db_]cleartrace	\$cputime
\$db_break[once]atline	\$db_help	\$deposit
\$db_break[once]onnegedge	[\$db_]setfocus	\$history
\$db_break[once]onposedge	[\$db_] settrace	\$showallinstances
\$db_break[once]when	\$db_showbreak	\$showvariables
\$db_[delete enable disable]break	\$db_showfocus	;, : . ?
\$db_[delete enable disable]focus	\$db_step[time]	

注意：在交互模式下输入？或\$db_help显示调试命令的帮助，或查看联机文档

用Verilog-XL调试

一些系统任务的功能:

- **\$db_step** — 单步执行一条或多条语句
- **\$db_steptime** — 执行指定的时间 (**units**)
- **\$db_setfocus** — 指定\$db_命令作用范围
- **\$db_disablefocus, \$db_enablefocus, \$db_deletefocus** — 范围操作
- **[\$db_]settrace, [\$db_]cleartrace** — 跟踪或取消跟踪
- **\$db_breakatline** — 在一行之前或某个基本单元后设置断点
- **\$db_breakbeforetime, \$db_breakaftertime** — 在某一仿真时间之前/之后设置断点
- **\$db_break[once]when** — 变量值发生变化设置断点
- **\$db_break[once]onnegedge, \$db_break[once]onposedge** — 信号沿断点
- **\$db_disablebreak, \$db_enablebreak, \$db_deletebreak** — 断点操作
- **\$db_showbreak, \$db_showfocus** — 显示断点

NC Verilog调试

调试非常有用的标准Tcl命令:

break	case	close	concat	eval	exit	for
foreach	format	gets	history	if	incr	open
puts	return	set	source	switch	unset	while

NC Verilog提供的调试用的扩展Tcl命令:

alias	call	database	deposit	describe	drivers
finish	fmibkpt	force	help	omi	probe
process	release	reset	restart	run	save
scope	status	stop	task	time	value
version	where				

注: 在交互模式下, 输入**help**或**help <command_name>**来获得帮助, 或参考联机文档

NC Verilog调试

这些命令的功能为：

- **deposit:** 给对象设置一个值
- **force:** 给对象强制赋值或维持原值。
- **release:** 取消在对象上的强制赋值
- **database**和**probe:** SHM或VCD数据库控制
- **value:** 显示信号值
- **describe:** 显示仿真对象的信息
- **process:** 显示过程块信息
- **time:** 显示当前仿真时间
- **drivers:** 显示对象的详细驱动信息
- **scope:** 显示及游历模型层次信息
- **run:** 执行、单步及跟踪一个仿真
- **stop:** 多种类型断点处理
- **save:** 保存当前仿真的状态
- **reset:** 从0开始重新仿真, **restart:** 从上次保存的状态开始仿真
- **status:** 显示仿真状态
- **finish:** 结束仿真并显示仿真状态

Verilog的断点

在交互模式中，使用系统任务\$stop或其它源代码级调试任务设置、控制断点。输入点“.”来继续仿真。

```
C1 > #10 $stop;           //10个时间单位后停止仿真
C2 > forever @( posedge clk) $stop; //在clk的每个上升沿停止仿真
C3 > .
C1: $stop at simulation time 10
C3 > if (en1 | en2)        //如果en1或en2不为0,
> @( negedge clk)         //则在clk的每个时钟上升沿停止仿真
> $stop;
C4 > $db_ breakatline(7);   //在当前模块的第七行停止仿真
Set break (1) at line 7, scope test, file ulatch. v
C5 > $db_ showbreak;        //设置断点，其编号为1
Status enabled break( 1) at line 7, scope test, file ulatch. v
C6 > $db_ delete( 1);      //删除断点1
```

注意：所有命令要用“;”结束，如同源代码。在命令执行结束后，仿真器给出提示符“>”

Verilog的断点

- 可以用延时再加上**\$stop**设置一个简单断点
- 可以使用**Verilog**语言的行为结构设置一个复杂断点。
- 设置断点之后，**Verilog-XL**不会自动继续执行。输入一个“.”来继续仿真操作。
- 交互式命令输入完成后，**Verilog-XL**才会以“>”代替Cn>来响应。
- 在提示符后输入的每个**Verilog**语句或语句块，**Verilog-XL**认为是在下面语句后输入的：

initial #(\$time)

也就是说，输入的任何过程语句或在一个**begin/end**块中的几个语句集合，被安排在当前仿真时间发生。甚至可以使用一些编译器指令。

但不能实例基本单元或模块、设置**timescale**、声明一个任务、函数或模块、对一个线网(**net**)进行持续赋值，或输入一个新的**initial**或**always**块。

NC Verilog断点

在交互模式时，使用**stop**命令来生成、命名、禁止、使能、删除或显示不同类型的断点。输入**run**来继续仿真。

```
ncsim > stop -time 10ns -delbreak 1    //10个时间单位后停止仿真
Created stop 1
ncsim > stop -condition {# clk == 1}    //在每个clk时钟上升沿停止仿真
Created stop 2
ncsim > run
10NS + 0 (stop 1)
ncsim > if {[ expr #en1] || [expr #en2]} \ //如果en1或en2为1，
> {stop -condition {# clk == 1}}        //在每个clk时钟上升沿停止仿真
Created stop 3
ncsim > stop -line 7 -name linbr         //在当前范围第7行停止仿真，并命名
这个
Created stop 4                          //断点的名字为linbr
ncsim > stop -show
2 Enabled Condition {# clk == 1}
3 Enabled Condition {# clk == 1}
linbr Enabled Line: ./ ulatch. v: 7 (scope: test)
ncsim > stop -delete 2                  //删除断点2
ncsim > stop -disable linbr            //禁止命名断点linbr
ncsim >
```

NC Verilog断点

在交互模式时，使用**stop**命令来生成、命名、禁止、使能、删除或显示不同类型的断点。断点可以为条件、行、对象和时间。

设置断点后，仿真器不会自动继续执行。输入**run**来继续仿真。

输入交互命令时，**Verilog-XL**的提示符为“>”。完成后为**ncsim>**。

如果使用完全访问，可以访问行，事件，值和连接信息，并修改信号值。

显示及设置调试作用域

使用系统任务\$showscopes;或Tcl命令scope -show显示当前级定义的作用域。

Directory of scopes at current scope level:

module (adder), instance (u1)

module (latch), instance (u2)

Current scope is (top)

Highest level modules:

top

使用系统任务\$scope或Tcl命令scope设置调试范围。

C2 > \$scope(u1);

ncsim > scope u1

使用CLI命令冒号 (:)或Tcl命令where显示当前仿真位置

Line 6, file "./ inter_bufif. v", scope (top)

Scope is (top)

显示及设置调试作用域

刚进入交互模式时，当前调试范围为最高层。

使用系统任务**\$scope**改变调试范围到层次中指定(名字) 级

使用**\$scope**命令设置当前范围，显示当前范围的全部或部分源代码，或显示一个特定范围的信息。

使用**\$scope<scope_name>**命令来改变调试范围到层次中命名的层次。

可以使用层次范围名。可以直接引用当前层次上定义的对象而无须他们的全部层次名。

设计反编译

可以反编译当前范围（缺省）或一个命名的范围。

- 在Verilog-XL中，使用\$list系统任务。用-d命令行选项来反编译整个设计。
- 在NC Verilog中，使用scope -list命令。

产生的输出在所有当前活动的过程语句旁边带有一个星号（*）。

```
// mods. v
```

```
1  module dut( q, d, c);
2      output
2      q; // = 1'h0, 0
3      input
3      d, // = St1
3      c; // = St0
4      reg
4      q; // = 1'h0, 0
5*   always
6      begin
7          @( posedge c)
8              q <= #( 0) d;
9      end
10   endmodule
```

声明变量的
当前 状态

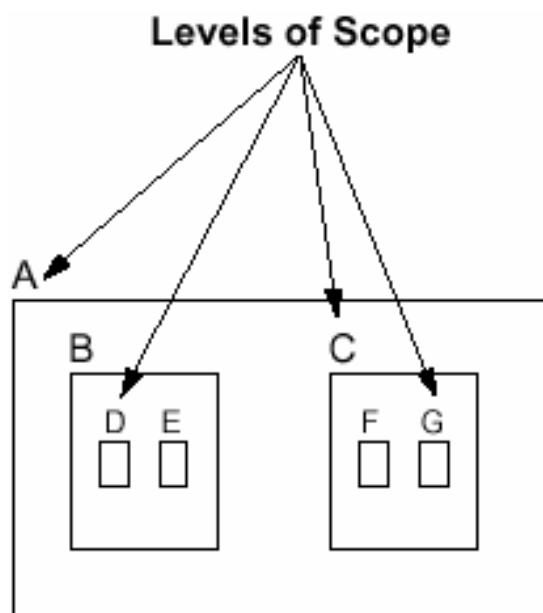
*表示当前
活动的语句

遍历设计

例中层次化描述的顶层为A。

刚进入交互式模式时：

- **scope -show**或**\$showscopes**；确定模块A为当前范围，列出模块实例B和C为A内的范围
- **scope -list**或**\$list**；反编译当前范围（模块A）
- **scope -list C**或**\$list(C)**；反编译模块实例C
- **scope B**或**\$scope(B)**；设置当前交互范围为模块实例B



等价的Verilog-XL命令

```
$list(A.B.D);  
$scope(A.B.D);
```

```
$scope(A.B);  
$list(D);
```

```
$list;
```

等价的NC Verilog命令

```
scope -list A.B.D
```

```
scope A.B
```

```
scope -list D
```

```
scope A.B.D
```

```
scope -list
```


显示信号值

在Verilog-XL中:

- 使用**\$display**,**\$monitor**和**\$strobe**来输出信息或值
- 使用**\$time**和**\$realtime**系统功能显示仿真时间, 用**\$cputime**系统功能来显示到目前为止实际使用的仿真时间, 以0.1秒计。

```
C1 > $monitor("%m %0d", $time, " o1 = %h", o1);
```

```
top 100.0 o1 = X
```

```
C2 > $display(" Simulation took %g/ 10 seconds", $cputime);
```

```
Simulation took 6/ 10 seconds
```

在NC Verilog中:

- 使用**value**, **describe**和**scope -describe**来输出信息或值。
- 使用**time**来显示当前仿真时间。

```
ncsim > describe o1 m1
```

```
o1..... wire (wire/ tri) = StX
```

```
m1 ..... instance of mod1
```

```
ncsim > value %h o1
```

```
1'hx
```

```
ncsim > time
```

```
10 NS
```

显示信号值

- 使用**value**命令可以显示一个或多个仿真对象的值。可以用**\$vlog_format**设置变量的缺省显示格式。如果不设置，它缺省为**%h**。必须在**elaboration**时将对象设置为可读取访问才能显示其值。
- 用**describe**命令显示仿真对象的简短描述。
- 使用**time**命令显示当前仿真时间。用Tcl命令**\$display_unit**设置时间单位，缺省为**auto**（最大的基准，显示一个不小于1的时间），并包括**delta**周期记数（零延迟事件被传输的次数）。

显示信号驱动

在Verilog-XL中:

- 使用系统任务\$showvars或\$showvariables得到线网或寄存器的详细信息。

```
C1 > $showvars( o1);
```

```
o1 (top) wire = StX, schedule = StX
```

```
    HiZ <- (top. m1): bufif1 g3( o1, i3, c3);
```

```
    schedule = St1
```

```
    St1 <- (top. m1): bufif1 g2( o1, i2, c2);
```

```
    St0 <- (top. m1): bufif1 g1( o1, i1, c1);
```

- 使用系统任务\$countdrivers列出一个线网的所有驱动。这个系统任务不太常用，在联机文档中有相关说明。

在NC Verilog中:

- 使用drivers或scope -drivers显示有关线网或寄存器的详细信息。

```
ncsim > drivers o1
```

```
o1..... wire (wire/ tri) = StX
```

```
    HiZ <- (top. m1): bufif1 g3( o1, i3, c3)
```

```
    St1 <- (top. m1): bufif1 g2( o1, i2, c2)
```

```
    St0 <- (top. m1): bufif1 g1( o1, i1, c1)
```

显示信号驱动

使用Tcl命令**drivers**或系统任务**\$showvars**列出一个或多个信号值的所有驱动。也可以列出范围，查看在这个范围中的信息。如果没有参数，则报出当前范围内所有信号的以下信息：

- 变量名
- 当前值
- 变量范围
- 变量是否被强制(forced)
- 变量类型
- 反编译出的驱动器及其输出值

要显示NC Verilog中的驱动，必须已经在elaboration时将对象设置连接访问。

\$showvars除报出和**drivers**同样的信息外，还有：

- 要调度的将来值
- 驱动器将来值，如果已经调度

可以给 **\$showvariables** 提供一个从0到7的值。

\$showvariables(0)；使用同**\$showvars**；

- 1 在整个层次中向下递归地报出变量信息。
- 2 抑制驱动器信息。 3递归地抑制驱动器信息。
- 4 只报出含有未知值的变量。 5递归进行4。
- 6 只报出还有未知值的变量，且抑制驱动器信息。 7递归进行6。

在Verilog-XL中修补设计

force和**release**命令用一个值或表达式覆盖信号的驱动。

C1 > \$reset; // 复位返回到仿真时间为0时的状态

C2 > force o1 = in1 | in2;

...

C14 > release o1; //清除net或寄存器上所有强制值

注意：可以用系统任务**\$list_forces**列出所有当前活动的force。

用**\$deposit**系统任务在线网(net)或寄存器上放置一个值(可以是0、1、x、z)，这个值向前传播。可以用一个简单的赋值设置值。

\$deposit(sig, 1);

\$deposit(bus, 'hAx2);

信号保持其值直到下一次改变。例如：

```
module patch;
    reg in;
    buf (tmp, in);
    buf (out, tmp);
    initial begin
        #5 in = 0;      //in = 0; tmp = 0; out = 0;
        #5 $deposit (tmp, 1'b1); //in = 0; tmp=1; out=1;
    end
endmodule

#5 in = 1; //in=1; tmp=1; out=1;
#5 in = 0; //in=0; tmp=0; out=0;
#5 $finish;
```

这些系统任务
Verilog-XL专用

在NC Verilog中修补设计

force和**release**命令用一个新值覆盖信号的驱动。

```
ncsim > reset // resets the simulation to time zero
```

```
ncsim > force o1 = 1'b0
```

```
ncsim > ...
```

```
...
```

```
ncsim > release o1 -keepvalue
```

注意：force命令等号右边可以是一个表达式，但这个表达式只计算一次。

用**\$deposit**系统任务在线网(net)或寄存器上放置一个值(可以是0、1、x、z)，这个值向前传播。

```
ncsim > deposit sig 1
```

```
ncsim > deposit bus 'hAx2
```

信号保持其值直到下一次改变。不能向由force赋值的信号进行deposit。

注意：可以用scope -list命令列出当前范围的信号的所有驱动，包括当前活动的force。deposit不是一个驱动。

在NC Verilog中修补设计

reset命令将设计复位到仿真时刻0的逻辑状态。

电路修补的通常用途为：

- 将电路的部分或全部初始化
- 设置电路到一个确定状态
- 覆盖错误值使得仿真可以继续进行
- 打断反馈循环并设置它到一个确定状态

用**force**命令来在调试时临时修补设计。只能对全部寄存器或线网矢量和展开后的向量线网的位或选择部分强制赋值。被赋的值必须为Verilog立即数。**force**覆盖所有其他驱动，并保持有效直到被其它Tcl或Verilog的**force**替代，或被Tcl或Verilog中的**release**取消。这个命令和Verilog **force**语句等价。

release命令取消Tcl或Verilog中一个或多个信号上的**force**。这个命令与Verilog-XL **release**语句等价。

deposit命令设置指定的net，寄存器，存储器或存储元件的值。被赋的值必须为Verilog立即灵敏。设置的值的变化向前传输。

要修补一个信号，必须已经在elaboration时将对象设置为写访问。

跟踪仿真动态

在NC Verilog中没有跟踪信息。在Verilog-XL中：

- **\$settrace**和**\$cleartrace**系统任务开启和关闭跟踪。
- **\$db_settrace**和**\$db_cleartrace**系统任务在设计中关心部分开启和关闭跟踪。
- 用**-t**选项使整个仿真过程中开启跟踪。

```
C1 > forever begin
```

```
> @enable $settrace; #2 $cleartrace; $stop; end
```

```
C2 > .
```

```
SIMULATION TIME IS 300
```

```
L8 "run_ ta. v": buf u4 >>> XL GATE = St0
```

```
L6 "run_ ta. v": wire clk >>> FROMXL NET = St0
```

```
L12 "latch. v" (run_ ta. u3): @( negedge enable) >>> CONTINUE
```

```
L13 "latch. v" (run_ ta. u3): r1 = #( 1) data; >>> = #( 32'h1, 1) 1'hx, x;
```

```
C1: #2 >>> CONTINUE
```

```
C1: $cleartrace;
```


跟踪仿真动态

用**\$settrace**和**\$cleartrace**系统任务来开启和关闭仿真跟踪。

用**-t**选项来为整个仿真来开启跟踪

跟踪显示事件、文件名以及在源描述的行号。

仿真跟踪在将一个时序问题追踪到一个小时时间窗口时，例如一个竞争情况，非常有用。

注意： 这些系统任务为Verilog-XL仿真器专用。NC Verilog没有一个跟踪模型。

跟踪仿真动态

- 在Verilog-XL中，单步跟踪可以为：
 - 分号（；）单步执行仿真
 - 逗号（，）单步执行仿真并跟踪
 - **\$db_step**单步仿真，跳过所有不在调试范围内的部分
 - **\$db_steptime**仿真指定的时间。

```
C1 > ,           //单步执行并跟踪
C1: forever
C1 > ,,
C1: begin
C1: @sel
C2 > ;;          //单步仿真，没有跟踪，除非使用$settrace或-t
```

- 在NC Verilog中，单步跟踪可以为：
 - **run -step**: 单步仿真，执行每一个语句。
 - **run- next**: 单步仿真，执行每个语句并跳过子程序调用（任务和函数）。
 - **run timeval**: 运行仿真一段指定的时间。

跟踪仿真动态

如果没有指定范围并开启了跟踪，分号、逗号和`$db_step` 等价。

run命令开始仿真或继续一个以前暂停的仿真。仿真根据指定的参数进行。时间可以是相对或绝对的。使用不带参数的**run**命令运行仿真直到结束或直到一个断点或发生一个错误。

命令历史列表

系统任务\$history或Tcl命令history可以列出以前执行的交互命令。

- 在NC Verilog中输入!命令编号或**history redo** -命令编号可以重新执行命令。
- 在Verilog-XL中，输入命令编号重新执行命令或-命令编号取消命令。

```
C4 > $history; Verilog-XL
```

```
Command history:
```

```
C1* forever
```

```
    @( posedge clk)
```

```
    $stop;
```

```
C2 $list;
```

```
C3 $display( clear);
```

```
C4* $history;
```

```
C5 > -1
```

```
C6 > 2
```

处于活动状态的命令由(*)标记

```
ncsim > history NC Verilog
```

```
1 stop -time 10ns -delibreak 1
```

```
2 stop -condition {# clk == 1}
```

```
3 run
```

```
4 if {# en1 || #en2} {
```

```
    stop -condition {# clk== 1}
```

```
}
```

```
5 stop -line 7 -name linbr
```

```
6 stop -show
```

```
7 stop -delete 2
```

```
8 stop -disable linbr
```

```
9 history
```

```
ncsim > history redo -2
```

```
Created stop 2
```

用**history**命令修改历史机制将保持的命令数来重新执行一个指定的以前的命令，并在重新执行前替代一个以前命令的一些部分。**history**命令是一个标准Tcl命令。

保存及重启仿真

在Verilog-XL中，可以在交互式模式或在源代码中使用下列命令：

- **\$save**和**\$incsave**命令保存整个仿真数据结构。
- **\$restart**或**-r**命令行选项可以从保存时间重新开始仿真。
- 这样可以不必重新编译并重新仿真到该时间点。
- **\$reset**重新设置仿真状态到时刻0的状态。
- 系统任务\$history或Tcl命令history可以列出以前执行的交互命令。

```
C1> #500 $save("save.dat"); // 在500 ns后保存仿真
```

```
C2> forever #100000 // 每个100000 ns后
```

```
> $incsave(" inc. dat"); // 增量式保存仿真
```

在NC Verilog中使用这些Tcl交互命令：

- **save**和**restart**保存并从一个仿真snapshot重新开始。
- **reset**重新从时刻0snapshot开始。
- 由于NC Verilog总是保存所有数据，snapshot只是简单的保存初始值。这使保存和重新开始更快更可靠。

```
ncsim > stop -time 100000NS -execute {save try1}
```

```
ncsim > restart try1
```

保存及重启仿真

可以将仿真数据结构保存到一个文件并在以后再次启动仿真。

用这个特性可以定点检查一个长仿真并：

- 在机器崩溃时保护仿真（machine failure）
- 执行快速“what if”关
- 系统任务**\$save**保存仿真数据结构。提供一个文件名参数用于存储仿真数据结构。
- 系统任务**\$incsave**进行增量保存。增量保存使用较少空间。
- 系统任务**\$restart**或**-r**命令行选项从一个以前保存的数据结构重新开始仿真。需要提供储存仿真数据结构的文件名作为参数。
- 注意使用**-r** 命令行选项时，不用指定设计文件。
- 仿真器执行一个过程语句中间不能保存。用**run -clean**到达仿真中的下一个“clean”点，然后执行 **save**命令。可以以后重新装载snapshot继续仿真；
- Tcl环境的状态(包括指针)并不和snapshot一起保存。要保存Tcl环境，必须单独执行一个**save -environment**。要从储存的Tcl环境**restart**，必须编译保存的环境文件。

在Verilog-XL中执行playing back TUI命令

系统任务**\$input**或**-i**命令行选项可以执行一个脚本文件。

仿真器在交互模式下执行命令脚本。

keyfile. txt

```
$display(" Executed keyfile at %g",$ realtime);
```

```
verilog mods. v -i keyfile. txt -q -s //在交互模式下，并在时间0执行keyfile.txt
```

```
C1 > $display(" Executed keyfile at %g",$ realtime);
```

```
Executed keyfile at 0
```

```
C2 > begin $input(" keyfile. txt"); #10 $stop; end . //停止时执行keyfile.txt
```

```
C2: $stop at simulation time 10
```

```
C3 > $display(" Executed keyfile at %g",$ realtime);
```

```
Executed keyfile at 10
```

```
C4 >
```

注意：在交互式模式下，**begin**和**end**之间语句顺序执行。通常的，在交互模式提示符处输入的语句在仿真继续进行时为并行执行。

在Verilog-XL中重新执行TUI命令

仿真器将交互命令写到一个key文件(key file)，其缺省名为verilog.key。

可以用**-k**选项重新命名key文件。

可以用**\$input**系统任务或用**-i**命令行选项开始Verilog-XL，读取并执行一个包含交互式命令的文件，例如该key文件。

仿真器在交互模式下读取并执行keyfile。

在NC Verilog中重新执行TUI命令

在交互模式中使用**source file_name**命令或**+tcl+file_name**命令行选项执行一个Tcl命令脚本，或keyfile。

commands. tcl

run 2NS

concat {Executed keyfile at} [eval time NS]

进入交互模式，并在时间0执行keyfile

ncverilog mods. v +tcl+ commands. tcl -s

+access+ rwc

ncsim > run 2NS

ncsim > concat {Executed keyfile at} [eval time NS]

Executed keyfile at 2 NS

ncsim > source commands. tcl

ncsim > run 2NS

ncsim > concat {Executed keyfile at} [eval time NS]

Executed keyfile at 4 NS

在NC Verilog中重新执行TUI命令

- 仿真器将交互命令写入一个key文件，缺省名为ncsim.key。
- 可以用**-k**选项来重新命名key文件。在交互模式中输入

save -commands tcl_file

或 **save-environment tcl_file** 来保存一个拷贝。

- 可以输入

source tcl_file命令

或 **+tcl+tcl_file**

或 **+ncinput+tcl_file**

或 **+nckeyfile+tcl_file**命令行选项

重启仿真器，读取并执行一个含有交互命令的文件，例如key文件。

Source命令只显示最后一个执行的命令的结果。Verilog-XL选项**-i**被忽略。

建立波形数据库

在Verilog-XL交互模式下创建并操作一个波形数据库，使用数据库类型相关的系统任务。

在NC Verilog交互模式下创建一个波形数据库，使用以下命令：

- 打开、禁止、使能、关闭或显示SHM或VCD数据库的信息：

`database [-options]`

例如，打开数据库mywaves.shm并命名其波形：

ncsim> database -open waves -into mywaves. shm

- 创建、禁止、使能、删除或显示探针的信息：

`probe [-options]`

例如，探测从实例u1到页单元的所有端口到一个SHM数据库，命名探针为seek，并打开波形工具：

ncsim> probe -create -name peek u1 -ports -depth to_ cells -waves

例如，监视被探测的信号rega并将其值打印到一个文件：

ncsim> probe -screen u1. rega -redirect mon_rega. txt

database命令打开、关闭、禁止或使能一个SHM或VCD数据库，或显示有关信息。

probe命令创建、删除、禁止、使能或显示一个指针。当被探测时，对象值的每一个变化都被保存到一个数据库。

其它Tcl命令

- alias [-options] 设置、取消或显示一个仿真命令别名
ncsim> alias -set go {run 10 ns; value sum}
- call [-systf] task_name [-options] 调用用户自定义系统任务或函数（PLI 应用）
ncsim> call {\$myprinter} {"set value to "} 8'hc
- process 确定正在执行或将要安排执行的过程块。
- status 显示资源使用和仿真时间。
ncsim> status
- task 用名字调用一个Verilog任务。
- version 显示仿真器的版本。

其它Tcl命令

- **Alias**命令是一个标准Tcl命令。可以设置、取消或显示一个仿真命令别名
- **call**命令调用一个用户自定义系统任务或函数（Verilog），或一个C-接口函数（VHDL）。插入**-systf**或**-predefined**选项在协同执行Verilog和VHDL仿真时区分名字相同的函数。
- **process**命令确定正在执行或将要安排执行的过程块（或VHDL过程）。
- **status**命令显示存储器使用、CPU使用和仿真时间。
- **task**命令调用一个Verilog任务。注意不能用这个命令调用一个Verilog函数或系统任务/函数。要传递参数值给该任务，在调用之前在其输入和双向端口**deposit**值。例如，如果任务test的输入为N，则可以用如下方法调用它：

```
ncsim> deposit test.N 1; task test
```
- 用**version**命令来显示仿真器版本。

总结

在这一章学习了：

- 控制并观察仿真
- 进入交互仿真模式
- 浏览设计层次
- 设置断点
- 显示信号波形
- 点检测(checkpointing)、退出及重新启动仿真
- 临时修补设计
- 单步和跟踪仿真行为
- 使用命令历史列表

复习

问题：

1. 怎样创建自己的命令使仿真前进**1ns**并显示一个寄存器的内容？
2. 怎样进入交互模式？
3. 可以用什么命令来设置缺省当前范围层？
4. 怎样得到一个信号的当前值？
5. 解释在每个仿真器内**deposit**和**force**之间的区别。
6. 在**Verilog**源代码中，是否当**ncsim**遇到**\$finish**系统任务时就会终结并返回到主提示符？

解答：

除非特别注明，以**\$**开头的命令用于**Verilog-XL**，其他命令用于**NC Verilog**。

1. 在**Tcl**中，可以使用**alias**命令创建一个命令。例如：**alias -set go1 {run 1 ns; value reg1}**

在**Verilog-XL**中，可以为命令定义一个文本宏。例如：**`define set #1 \$display(reg1); .**

2. 用**-s**命令行选项，按[^]C或在源代码中遇到**\$stop**。
3. 用**\$scope** 或**scope**来设置缺省当前范围层。
4. 用**\$display**，**value**或**describe**。可以用**\$showvars**或**drivers**获得一个信号的值和强度，以及它的所有驱动。
5. 在两个仿真器中，**force**保持一个值直到它被**release**，而一个**deposited**值保持其值仅到信号更新。在**Verilog-XL**中，**force**命令使**force**节点减速，但可以在一个表达式和一个信号之间创建一个连续关系。它可以为一个位或部分赋值，而**\$deposit**只能影响整个线网或寄存器。

第13章 使用图形调试环境

学习内容:

这一章将学习有关**SimVision**图形环境:

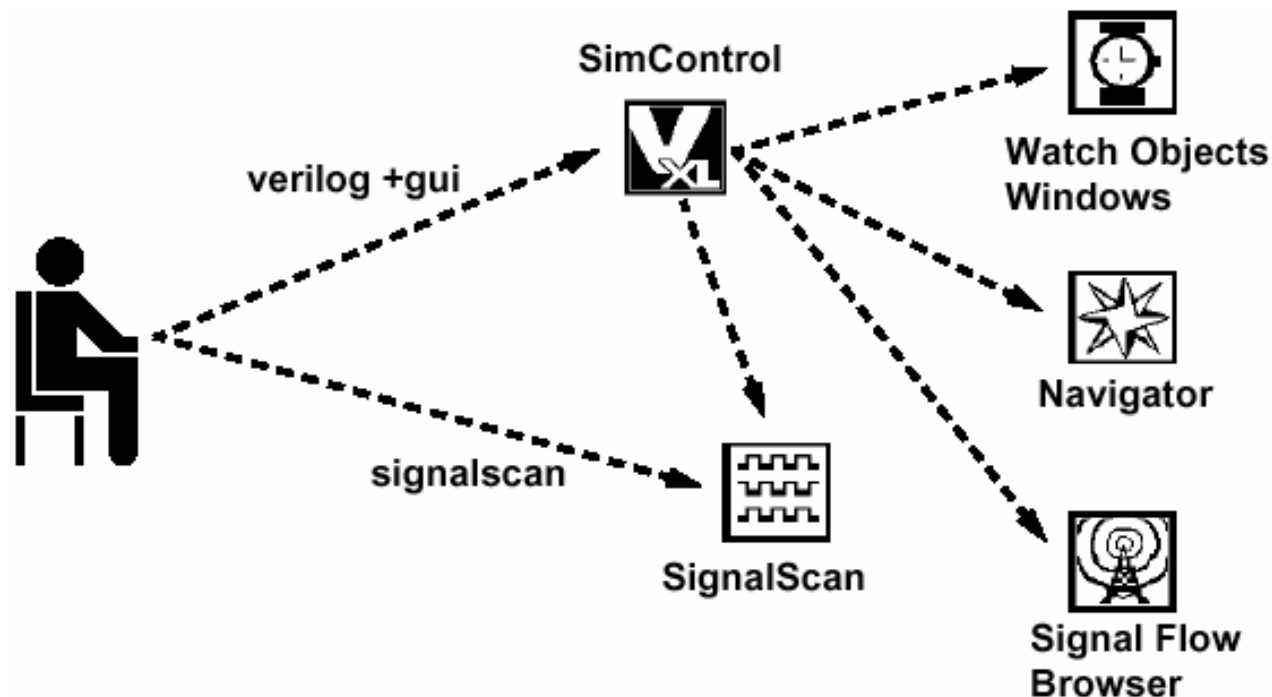
1. **SimControl**
2. **Navigator**
3. **Signal Flow Browser (SFB):** 信号流浏览器
4. **Watch Objects Windows:** 对象观察窗口

术语及定义

- **SimControl:** 图形仿真器接口；用SimControl来推进或中断仿真，打断并改变仿真，控制范围，等等。
- **SignalScan:** SimControl图形波形观察器
- **Navigator:** 显示设计层次和一个范围内对象信息的图形工具
- **Signal Flow Browser(SFB):** 在设计中从一个信号反向跟踪到其驱动源的图形工具
- **Watch Objects Windows :** 监视信号组及其值的窗口
- **Object:** 在SimControl中，任何一个信号或范围
- **Double-click:** 将光标放在一个项目上并快速点击鼠标左键两次
- **Right-select:** 将光标放在一个项目上并点击鼠标右键

启动图形环境

使用命令行选项+gui启动SimControl。在SimControl中可以访问SimVision所有部件。



仿真器在SimControl主窗口的控制下运行。当第一次启动时，SimControl主窗口显示顶层模块的源代码。

可以从SimControl工具菜单或工具栏访问SimVision环境（SignalScan, Navigator, Signal Flow Browser及Watch Objects windows）的其他部件。

SimControl

包括8个下拉菜单

固定菜单按钮可以快速访问常用命令

一个空的用户可定义按钮
当前仿真时间

显示正在调试的模块的源代码，可以选择这个区域的文本，但不能编辑

显示范围并浏览。在同一时间只能显示一下范围。若一个模块在多个文件出现，可以在subscope选择文件

显示交互仿真的输出。可以提示符处输入交互命令

显示信息，如仿真器的状态。

Menu Bar

Tool Bar

Source
Browser

Scopes
Region

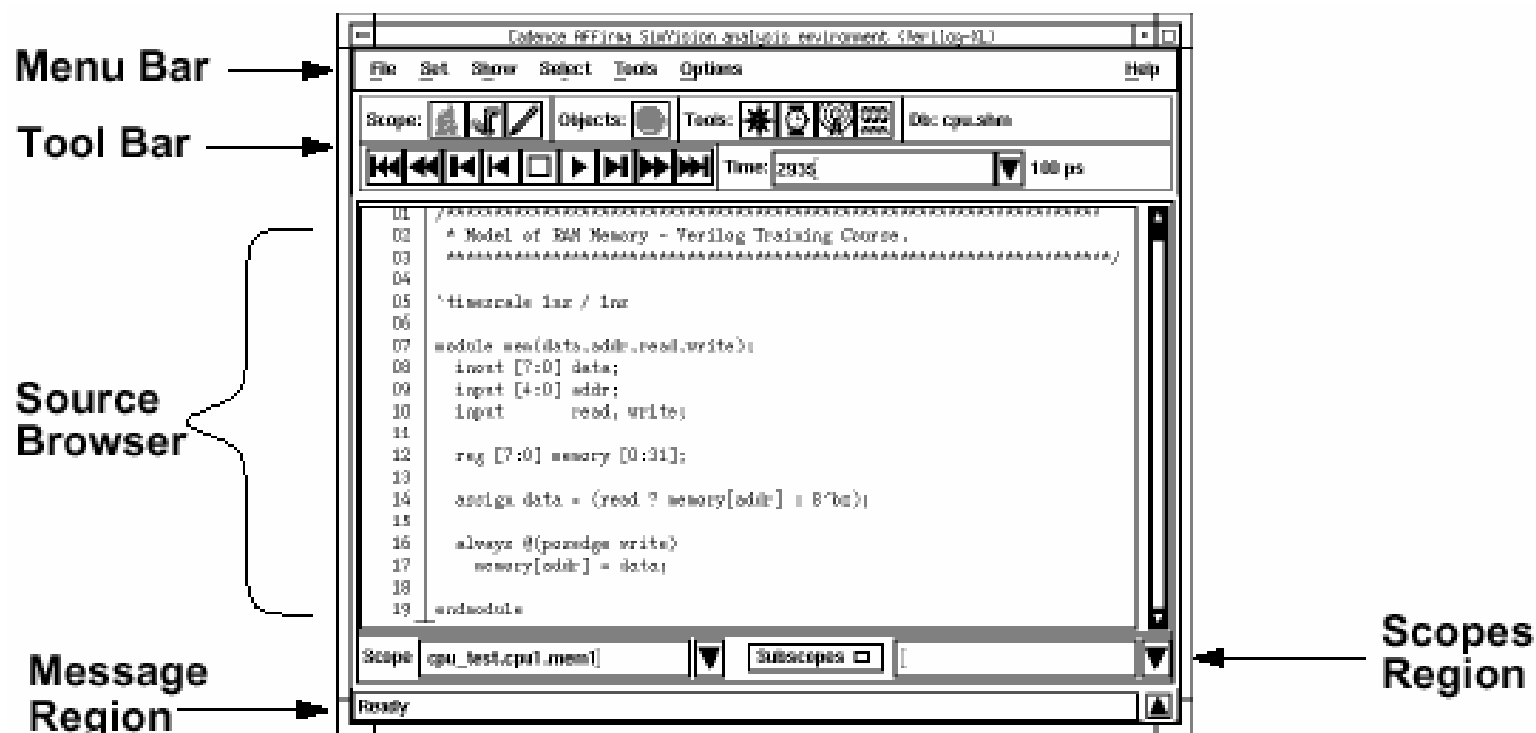
I/O Region

Message
Region



详细的菜单命令及其它SimVision细节请参考联机文档

后处理环境



在仿真并将探测信号放到一个SHM数据库之后，可以对设计进行后处理。

可以从GUI或通过使用+**ppe**选项进入后处理模式。

打开仿真当前设计时生成的SHM数据库。

交互及后处理

用+gui选项启动SimVision时，进入仿真器交互模式。进入后处理模式时，将仿真器释放并对保存在波形数据库中的值进行操作。

在交互模式中：

- 时间只能向前推进。
- 可以读取并修改仿真值，访问连接并设置行断点，取决于使能的是何种访问。
- 可以将值保存到一个波形数据库。
- 可以执行交互命令和源交互式脚本。
- 可以保存、重新开始、重新设置、运行和停止仿真。

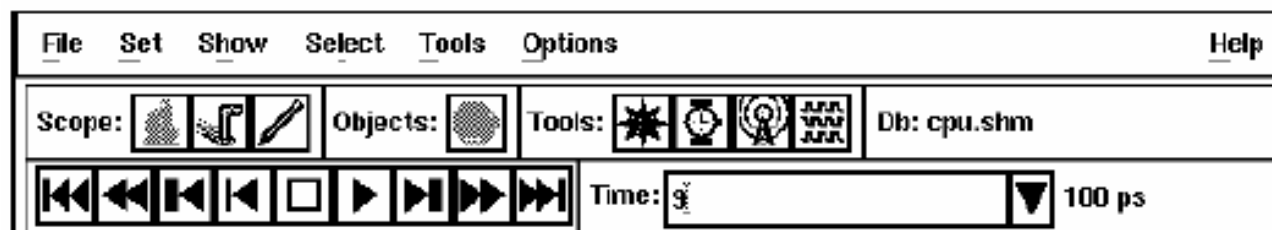
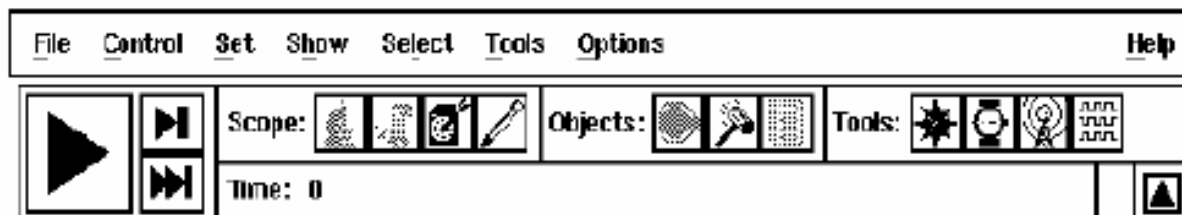
在后处理模式中：

- 可以扫描、单步、执行，或按时间向前或向后跳转。
- 可以读取仿真值并访问连接，取决于在仿真过程中使能的访问及在波形数据库中保存的信息。

SimControl菜单及工具条

SimControl下拉菜单和固定菜单按钮可以：

- 编辑源文件，打开数据库，和查找文本。在交互模式中，还可以source命令文件、保存并重新开始仿真。
- 执行、单步、停止和复位到开始时刻。在后处理模式中，还可以向后执行，并可以向前或向后扫描。
- 设置并显示断点和范围。在交互模式中，还可以设置并显示forces和探针
- 选择范围，端口和信号。
- 启动其他SimVision工具。
- 按照用户的意愿调整SimControl的行为

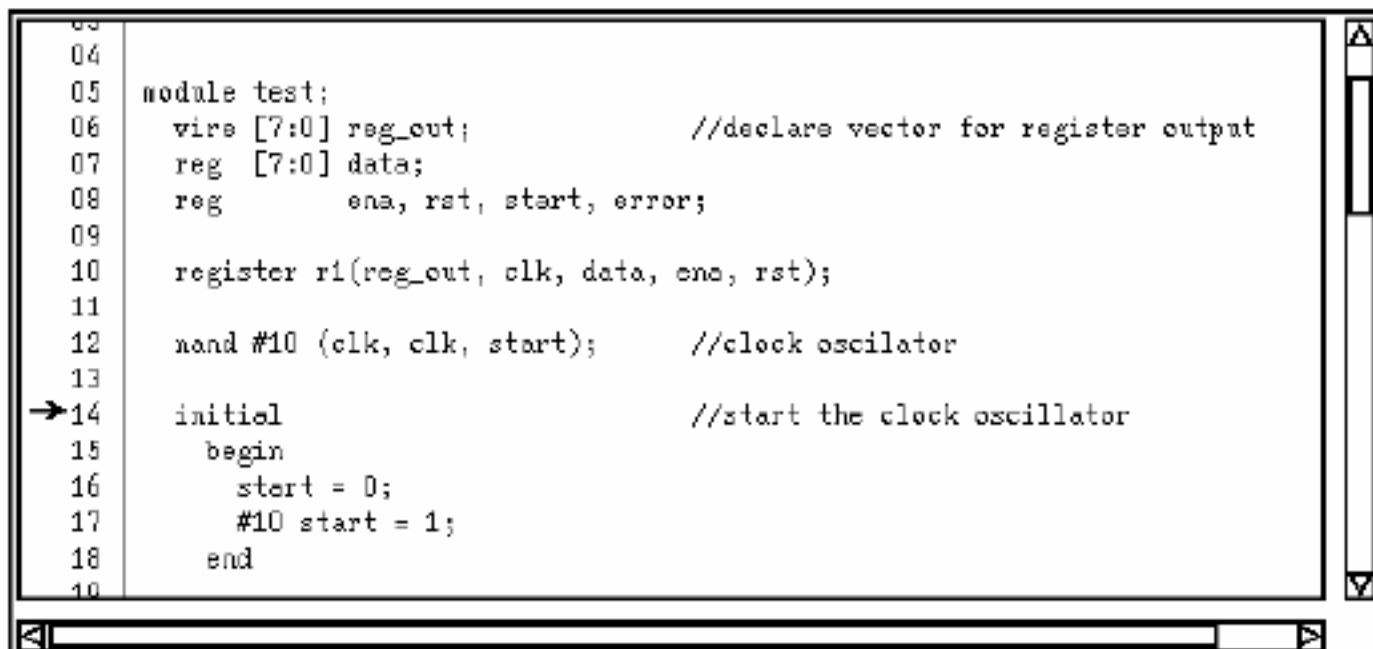


Source Browser

Source Browser显示在当前调试模块的源代码。

可以在**Source Browser**中选择任意对象（寄存器，**net**，实例或线）

在**Source Browser**中右击一个对象或行号弹出可以对该对象或行进行操作的命令菜单

A screenshot of the Source Browser window in a digital logic design tool. The window displays a Verilog module named 'test'. The code includes declarations for a 7-bit output vector 'reg_out', a 7-bit data register 'data', and a register 'reg' with inputs 'ena', 'rst', 'start', and 'error'. It also shows an instance of a 'register' component 'r1' and a clock oscillator circuit using a NAND gate and a delay. An initial block is present to start the clock oscillator. The line numbers 03 through 19 are visible on the left side of the code editor. A mouse cursor is pointing at line 14. The window has a standard scrollbar on the right and a status bar at the bottom.

```
03  
04  
05 module test;  
06     wire [7:0] reg_out;           //declare vector for register output  
07     reg [7:0] data;  
08     reg      ena, rst, start, error;  
09  
10     register r1(reg_out, clk, data, ena, rst);  
11  
12     nand #10 (clk, clk, start);    //clock oscillator  
13  
→14     initial                      //start the clock oscillator  
15         begin  
16             start = 0;  
17             #10 start = 1;  
18         end  
19
```

选择对象

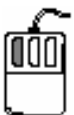
在SimVision环境中，选择和弹出菜单使鼠标成为一个有力的工具。



左键可以在所有**SimVision**窗口中选择对象



右击鼠标右键选择对象可以弹出一个菜单。



按住**Control**键再按左键可以选择多个对象或取消选择。



中键可以在窗口之间拖动对象

选择对象

在一个SimVision窗口选择一个或多个对象时，其他SimVision窗口的这些对象也均被选择。

当右击选择一个对象时，弹出一个弹出菜单，包含用于该对象的一组常用命令。在不同的SimVision工具中弹出的不同的菜单。还可以在源浏览器中右击选择一个行号。

按control键并用鼠标左键点击一个对象，可以选定该对象而不影响其他被选对象。

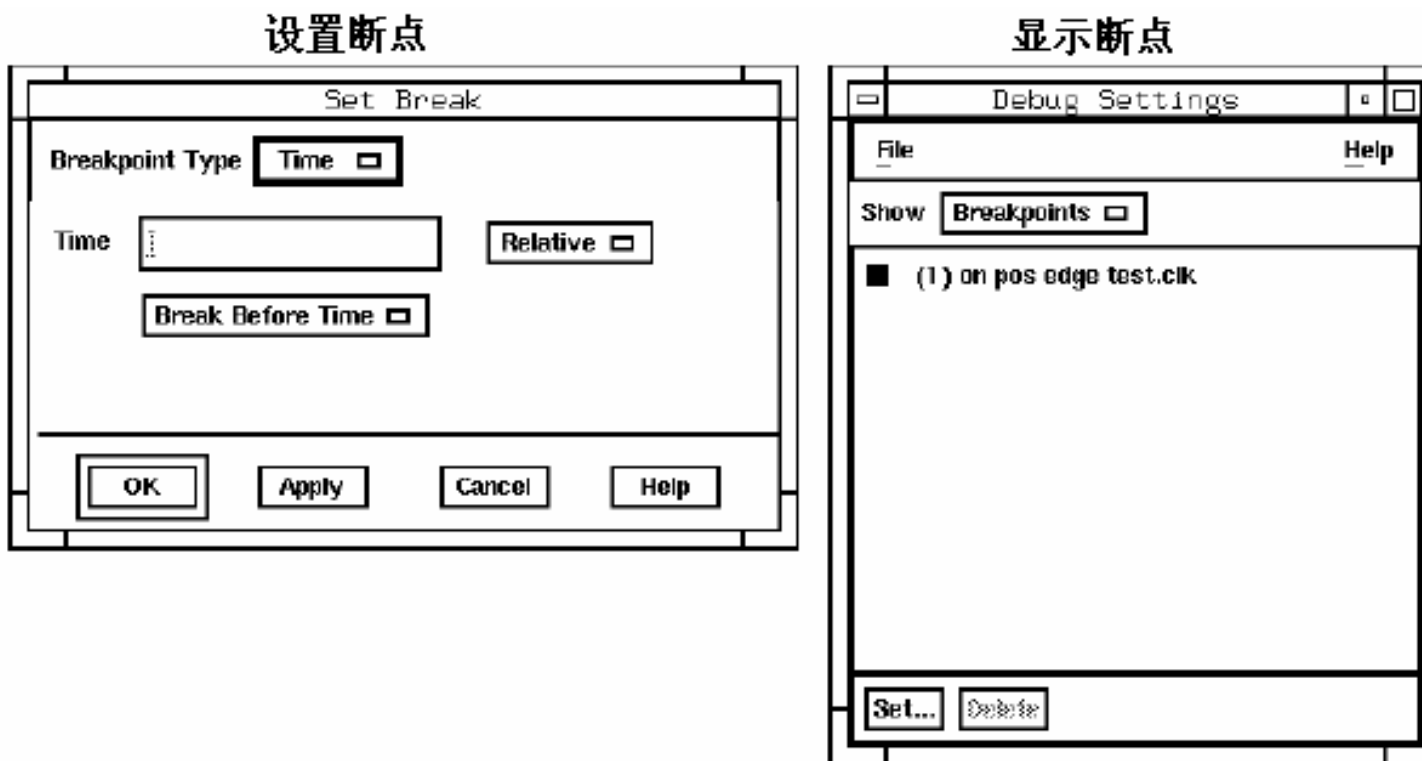
添加选择：选择一个对象后，可以选择其它更多对象。

取消选择：当一个对象被选定，可以用这个办法来取消对它的选择。

点击拖动：当用鼠标中间的按钮选择一个对象时，可以拖动这个对象到另一个位置。如果鼠标只有两个按钮，可以用左键点击拖动。

设置断点

在下面的对话框中可以设置、使能、取消、列出，和删除断点。断点可以是仿真时间、值的改变、条件或代码行。断点的功能视仿真器而不同。




设置断点

断点是使仿真停止的事件。有三种类型的断点：


- 基于时间：当仿真到一个指定时间停止。此为缺省。
- 基于行：当仿真到源代码一个指定的行时停止。必须指定范围，文件名，和行号。只用于交互模式。
- 基于对象：当指定信号的值发生变化或指定跳变发生时停止。在NC Verilog中，不能指定一个单一跳变。
- 基于条件：当指定的Tcl表达式值为真时停止。

SimControl提供四种设置断点的方法：

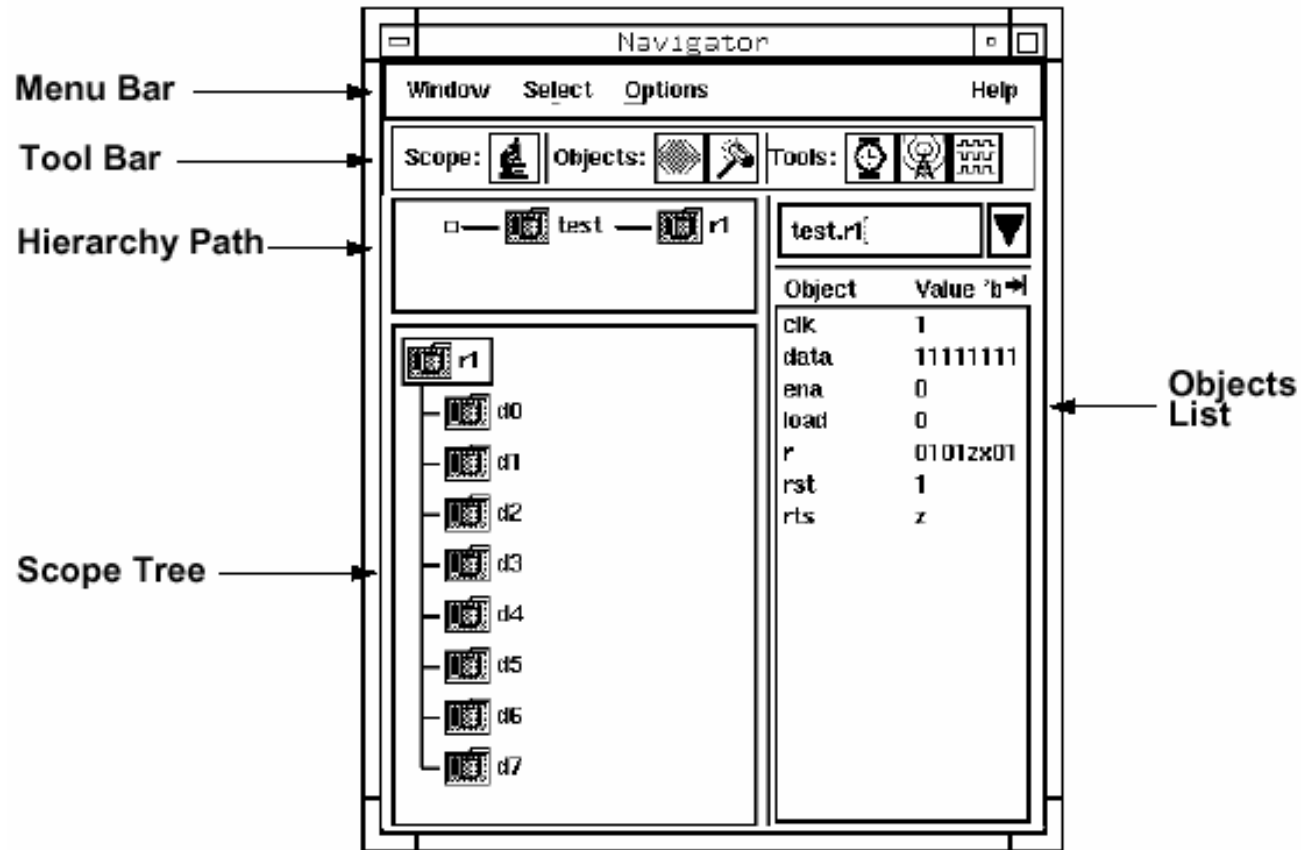
- 在菜单选择**Set-Breakpoints**
- 在**Show-Breakpoints**对话框，按**Set**按钮。
- 按**Set Breakpoints**按钮  (只用于基于对象的断点设置)。
- 在任何一个SimControl窗口中右击选择一个信号名，弹出一个有**Set Break**的菜单。

Navigator


用navigator来查看设计层次和当前范围的对象。

要启动navigator，在SimControl窗口使用**Tools**菜单下**Navigator**，或按navigator按钮 

Navigator工具条是主窗口工具条的子集



Navigator

- 浏览设计层次时，navigator生成一个树结构，在这个树结构中每个节点为设计层次中一个范围。
 - 双击一个没有展开的节点以显示子层
 - 双击一个展开的节点隐藏子层
- 在navigator中有两种方法设计当前范围(scope):
 - 选择一个节点，然后按**Scope**按钮 ，或者在SimControl窗口中选择**Set-Scope**。
 - 右击选择一个节点并从弹出菜单中选择**Set Debug Scope**，或双击该节点。
- **注意：**由于所有SimVision窗口交互作用，从任何SimVision窗口设置范围时，该范围的源代码自动读入源浏览器source browser。
- 用Options菜单设置显示选项
 - 可以隐藏范围树或对象列表。
 - 启动Object List Options框或Scope Tree Options框指定显示及每个区域的内容。

Signal Flow Browser（信号流浏览器）

使用信号流浏览器signal flow browser跟踪信号的驱动，设置信号属性。

可以从SimControl启动信号浏览器SFB：

菜单条：通过选择**Tools-Signal Flow Browser**

工具条：通过选择Signal Flow Browser 按钮 

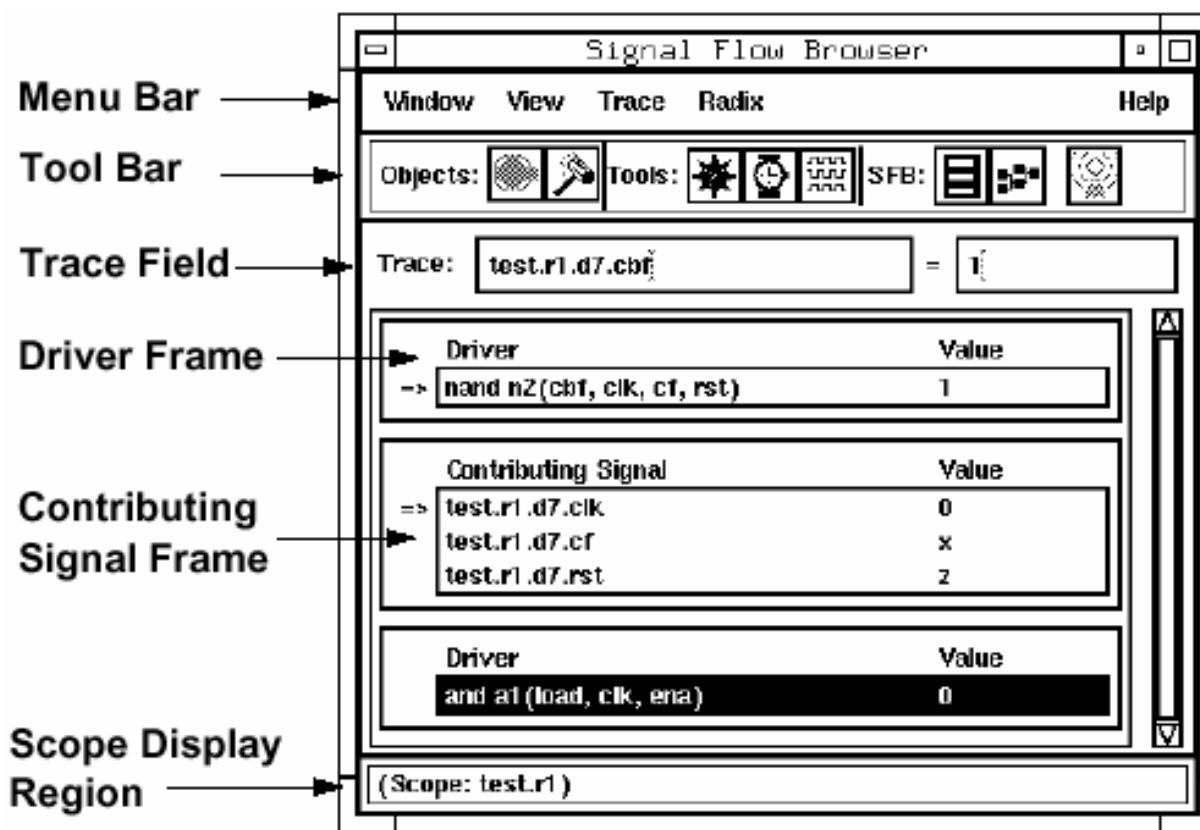
SFB的工具条包括：

主窗口工具条的子集

Trace Back ：跟踪选择信号，代替当前跟踪

Stack View 和Trace View  翻转SFB外观

右图是一个Stack View形式



Signal Flow Browser（信号流浏览器）

信号流浏览器可以交互地跟踪一个信号的驱动以及对这些驱动所起的作用

选择一个将被跟踪的信号：

打开信号流浏览器前：

在源浏览器中选择信号。在任何窗口中所做的选择会传递到其他所有窗口

打开信号流浏览器后：

— 在Trace区输入一个层次信号名。

— 在其他Simvision窗口中选择一个信号并用鼠标中键将其拖到信号流浏览器中。

Signal Flow Browser（信号流浏览器）

- 信号流浏览器（SFB）是一个高效的设计调试环境。
- 用SFB可以从一个行为反常的信号开始，向后跟踪其驱动和作用信号直到发现行为反常的原因。
- 信号流浏览器可以执行下面操作：
 - 以选择的基数显示一个信号的值。
 - 显示信号的驱动。
 - 查看信号的输入或驱动的细节。
 - 显示所有对一个驱动起作用的信号。
 - 跟踪一个模块端口到一个较低的层次。

Signal Flow Browser（信号流浏览器）



- 必须首先选择要跟踪的信号并将它输入或点击拖放到SFB中。此时出现一个驱动器（**Driver**）框，显示出所有该信号的驱动。
- 可以在一个**Driver**框里跟踪任何驱动并显示其输入；或者通过首先选择该驱动器，然后从菜单中选择**Trace-Show Inputs**；或者简单地双击该驱动。
- 可以在一个**Contributing Signal**框中跟踪任何信号以显示其所有驱动，或者通过首先选择该信号，然后从菜单总选择**Trace-Show Drivers**；或者简单地双击该信号。
- 如果一个信号的驱动为一个模块实例的端口，可以跟踪它并显示在这个模块实例内部的信号源。首先选择该驱动器，然后从菜单中选择**Trace-Descend**。
- 例如，如果在一个驱动器框中有下面这个模块实例：

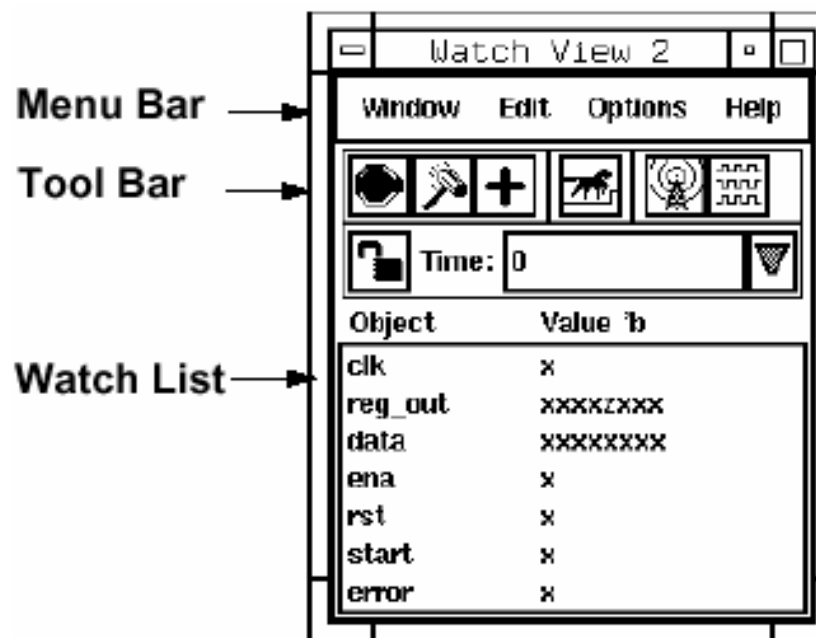
```
register r1(. r( reg_out) ...)
```

选择reg_out并使用**Trace-Descend**将显示 r 为信号源。




- 可以通过首先选择该驱动器，然后从菜单中选择**View-Driver Info**显示一个驱动的细节。

Watch Objects Windows (信号观察窗口)

- 可以打开任意多个对象观察窗口。每个窗口包含一个信号表和当前仿真值。
- 要在一个对象观察窗口中添加对象，用点击信号并拖动或按**Add Objects** 
- 可以重新命名、锁定、删除、关闭，**iconify**，或克隆每个对象观察窗口。
- 点击**Find Next Edge** 使仿真到下一个跳变。



Watch Objects Windows（信号观察窗口）

- 打开多个观察窗口
 - 每个观察窗口在关闭时会自动保存。
 - 可以打开一个新窗口或一个以前保存过的窗口。打开的第一个窗口缺省名“View1”。第二个窗口为“View2”，依次类推。
 - 可以重新命名一个窗口。
 - 可以克隆一个窗口，生成一个完全拷贝。
- 要从另一个SimVision窗口添加对象到一个观察窗口，用点击拖动或者选择它们然后按Add Objec
 - 添加一个信号是将其添加到观察列表中。
 - 添加一个范围是添加在此范围中的信号到观察列表。
- 用观察窗口的选项（Option）菜单为每个窗口定制信号命名和值显示的方式。
- Find Next Edge 使仿真前进到一个信号的下一个跳变，或到下一个用户设置的断点。
- Lock 可以锁定一个观察窗口，使它不会随时间更新其显示。不能向一个被锁定的窗口添加新信号或从它前进时间。当解锁后，它将立即更新到反映当前时间的当前值。
- Time区在后处理模式中可编辑，并只为该特定观察窗口改变时间值。

总结

- 这一章学习了**SimVision**图形环境：
 - **SimControl**
 - **Navigator**
 - **Signal Flow Browser(SFB)**信号流浏览器
 - **Watch Objects Windows**

复习

问题

1. **SimVision**的五个主要基于窗口的元件是什么？
2. 怎样显示设计层次？
3. 怎样在你的设计中显示和监视一组信号？、
4. 可以从观察对象窗口中在一个对象上设置一个断点？
5. 怎样确定一个驱动的作用信号？

解答：

1. **SimVision**的五个主要基于窗口的元件为**SimControl**，**Navigator**，**Watch Objects**，**Signal Flow Browser**和**SignalScan** 波形观察器。
2. 可以用**navigator**显示并在设计层次中浏览。**Navigator**显示范围，对象和对象值。可以在**Source Browser**中显示每个范围的源代码。
3. 可以用对象观察窗口查看一组信号及其值。
4. 是的，可以从对象观察窗口在一个对象上设置一个断点，或可以用**Find Next Edge**按钮高效地设置一个断点，仿真并去除断点。
5. 可以通过用信号流浏览器跟踪一个驱动的作用信号来找出这些信号。这在确定一个问题信号的源头很有用。可以反向跟踪驱动及其作用信号直到发现与预期行为不同信号。

第14章 对验证的支持

学习内容

1. 理解Verilog文本输出
2. 理解不同的读取仿真时间的系统函数
3. 理解 Verilog文件I/O功能

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

\$stime

\$realtime

- **Verilog**支持文本输出的系统任务:

\$display

\$strobe

\$write

\$monitor

仿真时间

访问仿真时间

- **\$time**, **\$realtime**, 和 **\$stime** 函数返回当前仿真时间。
- 这些函数的返回值使用调用模块中 **timescale** 定义的时间单位
- **\$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。

\$stime 函数返回一个 **32** 位整数时间值。对大于 2^{32} 的时间，返回模 2^{32} 的值。使用它可以节省显示及打印空间。

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;
```

```
        #8  in1 = 1;
```

```
        #10 $display("%t %b %b", $realtime, in1,  
o1);
```

```
        #10 $finish;
```

```
    end
```

```
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

输出格式化时间信息

```
`timescale 1 ns / 10 ps
module top;
  reg in1;
  not #9.53 n1 (o1, in1);
  initial
  begin
    $display("time realtime stime \t in1 \t o1 ");
    $timeformat(-9, 2, "ns", 10);
    $monitor("%d %t %d \t %b \t %b", $time, $realtime,
              $stime, in1, o1);

    in1 = 0;
    #10 in1 = 1;
    #10 $finish;
  end
endmodule
```

time	realtime	stime	in1
0	0.00ns	0	0
10	9.53ns	10	0

1			
10	10.00ns	10	1
1			
20	19.53ns	20	1
0			

输出格式化时间信息

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 n1 (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 n1 (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

转义符

\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	\$strobeo
\$writeh	\$strobeh

显示信号值—\$write和\$strobe

```
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, , data);
    $display($time, , data);
    data = 30;
  end
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:

0000000000000000f 00000010 1

注意当前时间, 一个64位量, 需要16个十六进制的数。

- \$display: 10 20

- \$strobe: 10 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);

监示信号值—\$monitor

- **\$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**\$monitroff**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

\$monitro

\$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将等于0。
 - 如果文件成功打开，MCD中的一位将被置位。
- 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。

文件输出

- **\$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。

文件输出

...

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;

end

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);

必须声明为integer

通道0（编号为1）为
标准输出及verilog.log

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>);

文件输入

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**指定要调入的文件。
- **mem_name**指定存储器名。
- **start**和**finish**决定存储器将被装载的地址。**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。

文件输入

\$readmemb和**\$readmemh**的文件格式：

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

声明的存储器组

reg [0:7] mema[0:1023]

文本文件：mem_file.txt

0000_0000

0110_0001 0011_0010

// 地址3~255没有定义

@100 // hex

1111_1100

//地址257~1022没有定义

@3FF

1110_0010

00000000	0
01100001	1
00110010	3
11111100	256
11100010	1023
0	7

```
module readmem;  
  reg [0:7] mema [0:1023]  
  initial  
    $readmemb("mem_file.txt",  
      mema);  
endmodule
```

文件输入

`$readmemb`和`$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存储器字。
- 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。

复习

问题：

- 哪个系统任务显示参数列表中信号的稳定状态值？
- 每次能打开多少个输出文件？

解答：

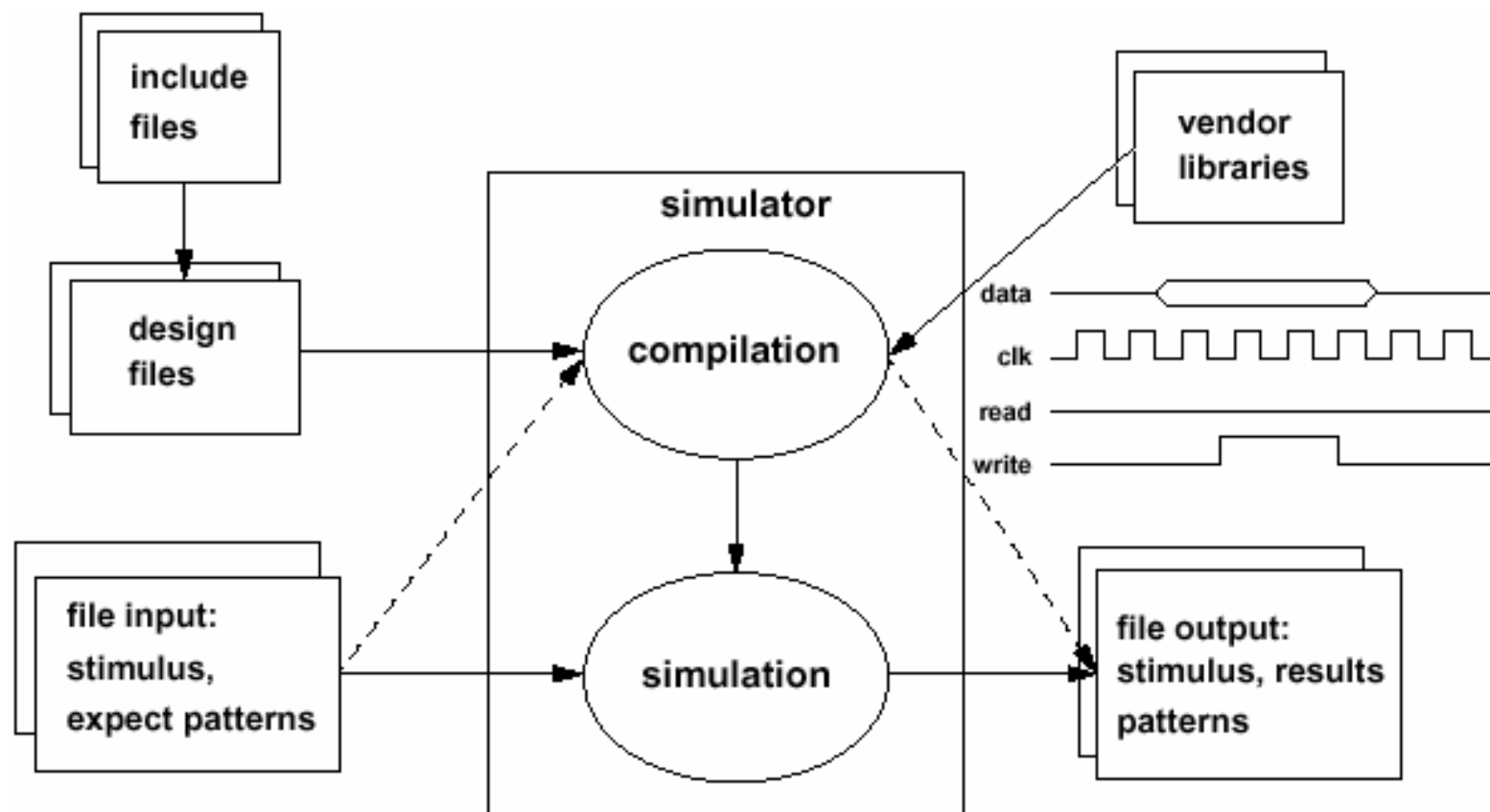
- 系统任务**\$monitor**和**\$strobe**显示参数列表中信号的稳定状态值。
这些任务在时间前进之前输出信号值。
- 每次只能打开一个输出文件，包括已由仿真器打开的任何**log**文件。

第十五章 Verilog Test Bench使用简介

学习内容:

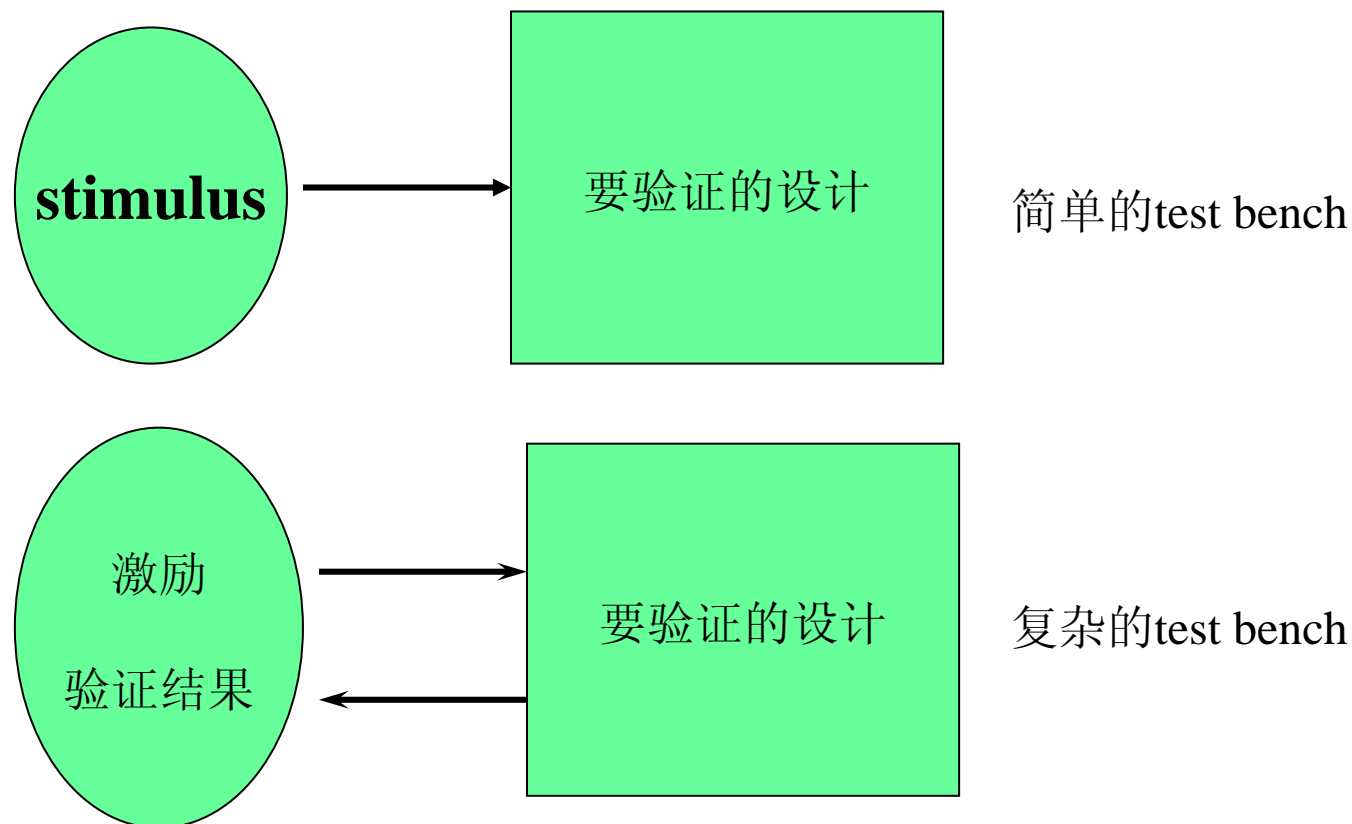
- 用一个复杂的**test bench**复习设计的组织与仿真
- 建立**test bench**通常使用的编码风格及方法

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。
- 复杂的test bench是自检测的，其结果自动验证。

并行块

- **fork...join**块在测试文件中很常用。他们的并行特性使用户可以说明绝对时间，并且可以并行的执行复杂的过程结构，如循环或任务。

```
module inline_tb;
  reg [7: 0] data_bus;
  // instance of DUT
  initial fork
    data_bus = 8'b00;
    #10 data_bus = 8'h45;
    #20 repeat (10) #10 data_bus = data_bus
+ 1;
    #25 repeat (5) #20 data_bus = data_bus
<< 1;
    #140 data_bus = 8'h0f;
  join
endmodule
```

上面的两个repeat循环从不同时间开始，并行执行。象这样的特殊的激励集在单个的**begin...end**块中将很难实现。

Time	data_bus
0	8'b0000_0000
10	8'b0100_0101
30	8'b0100_0110
40	8'b0100_0111
45	8'b1000_1110
50	8'b1000_1111
60	8'b1001_0000
65	8'b0010_0000
70	8'b0010_0001
80	8'b0010_0010
85	8'b0100_0100
90	8'b0100_0101
100	8'b0100_0110
105	8'b1000_1100
110	8'b1000_1101
120	8'b1000_1110
125	8'b0001_1100
140	8'b0000_1111

包含文件

- 包含文件用于读入代码的重复部分或公共数据。

```
module clk_gen (clk);
output clk; reg clk;
`include "common.txt"
initial begin
    while ($ time < sim_end)
    begin
        clk =
initial_clock;
        #(period/2) clk = !initial_clock;
        #(period/2);
    end
    $finish;
end
endmodule
```

```
// common. txt
// clock and simulator constants
parameter initial_clock = 1;
parameter period = 15;
parameter max_cyc = 100;
parameter sim_end = period *
max_cyc
```

在上面的例子中，公共参数在一个独立的文件中定义。此文件在不同的仿真中可被不同的测试文件调用。

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，测试基准(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
 - ✓ 时序关系规则
 - ✓ 代码紧凑

```
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] data_ bus, stim_ array[ 0: 15]; // 数
    组
    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("stimulus. txt") ;
      RESULTS = $fopen("results. txt") ;

      fork
        if (STIMULUS != 0 ) forever #( period/2)
          $fstrobeb (STIMULUS, "%b", in_vec);
        if (RESULTS != 0 ) #( period/2) forever #( period/2)
          $fstrobeb (RESULTS, "%b", out_vec);
      join
    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

<code>always @(posedge par_err)</code>
<code>\$display (" error-bus parity errors detected");</code>
<code>always @(posedge cor_err)</code>
<code>\$display("warning-correctable error detected");</code>

- 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。

强制激励

- 在过程块中，可以用两种持续赋值语句驱动一个值或表达式到一个信号。
 - 过程持续赋值通常不可综合，所以它们通常用于测试基准描述。
 - 对每一种持续赋值，都有对应的命令停止信号赋值。
 - 不允许在赋值语句内部出现时序控制。
- 对一个寄存器使用**assign**和**deassign**，将覆盖所有其他在该信号上的赋值。这个寄存器可以是RTL设计中的一个节点或测试基准中在多个地方赋值的信号等。

```
initial begin
    #10 assign top.dut.fsm1.state_reg = `init_state;
    #20 deassign top.dut.fsm1.state_reg ;
end
```

- 在**register**和**net**上（例如一个门级扫描寄存器的输出）使用**force**和**release**，将覆盖该信号上的所有其他驱动。

```
initial begin
    #10 force top. dut. counter. scan_reg. q = 0 ;
    #20 release top. dut. counter. scan_reg. q ;
end
```

强制激励

在上面两个例子中，在 **net**或**register**上所赋的常数值，覆盖所有在时刻**10**和时刻**20**之间可能发生在该信号上的其他任何赋值或驱动。如果所赋值是一个表达式，则该表达式将被持续计算。

- 可以强制(**force**)并释放一个信号的指定位、部分位或连接，但位的指定不能是一个变量（例如**out_vec[i]**）
- 不能对**register**的一位或部分位使用**assign**和**deassign**
- 对同一个信号，**force**覆盖**assign**。
- 后面的**assign**或**force**语句覆盖以前相同类型的语句。
- 如果对一个信号先**assign**然后**force**，它将保持**force**值。在对其进行**release**后，信号为**assign**值。
- 如果在一个信号上**force**多个值，然后**release**该信号，则不出现任何**force**值。

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。

建立时钟

例2：有启动延时的对称时钟的例子：

```
reg ck;  
initial begin  
    ck = 0;  
    #( period)  
    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
```



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。

建立时钟

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。

使用task

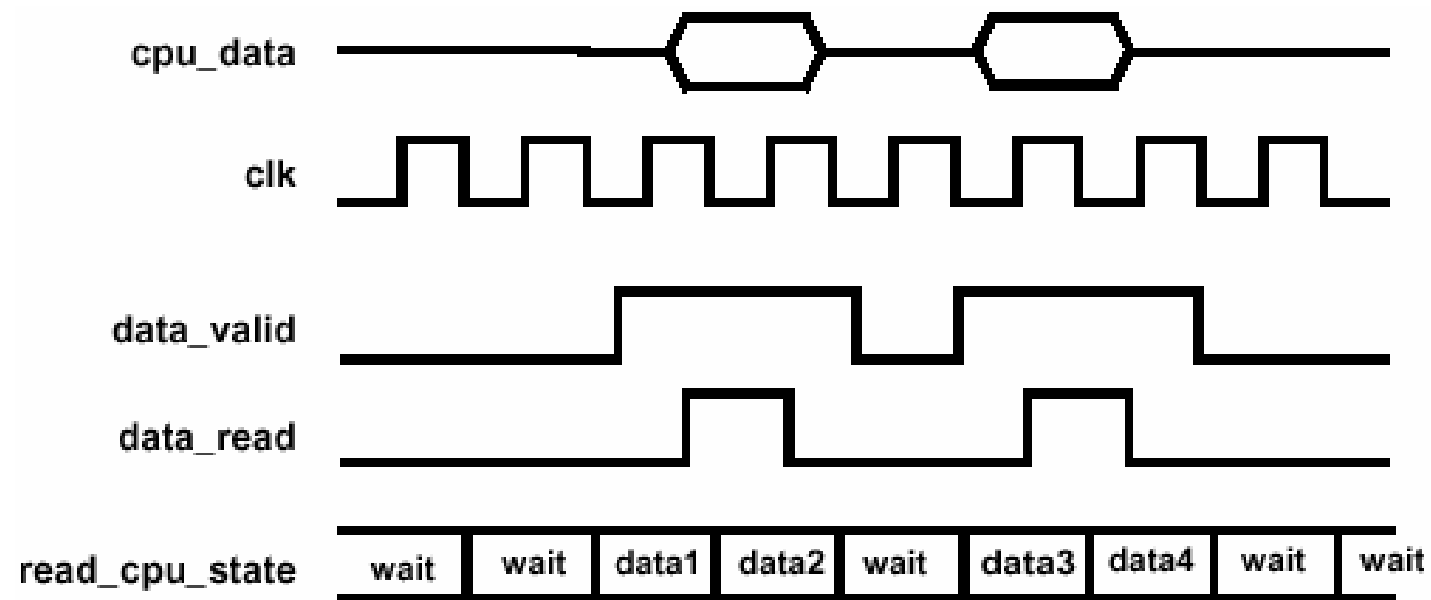
在test bench中使用task可以压缩重复操作，提高代码效率。

```
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
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

产生的波形



复习

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。

第16章 存储器建模

学习内容:

- 如何描述存储器
- 如何描述双向端口

存储器件建模

描述存储器必须做两件事：

- 说明一个适当容量的存储器。
- 提供内容访问的级别，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 (read_data, addr,
read_en_);
    input read_en_;
    input [3:0] addr;
    output [3:0] read_data;
    reg [3:0] read_data;
    reg [3:0] mem [0:15];
    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

简单的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 (data, addr, read, write);  
    inout [3:0] data;  
    input [3:0] addr;  
    input read, write;  
    reg [3:0] memory [0:15]; // 16*4  
// 读  
    assign data = read ? memory[addr] :  
4'bz;  
// 写  
    always @(posedge write)  
        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。

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

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。

使用双向端口

用关键词**inout**声明一个双向端口

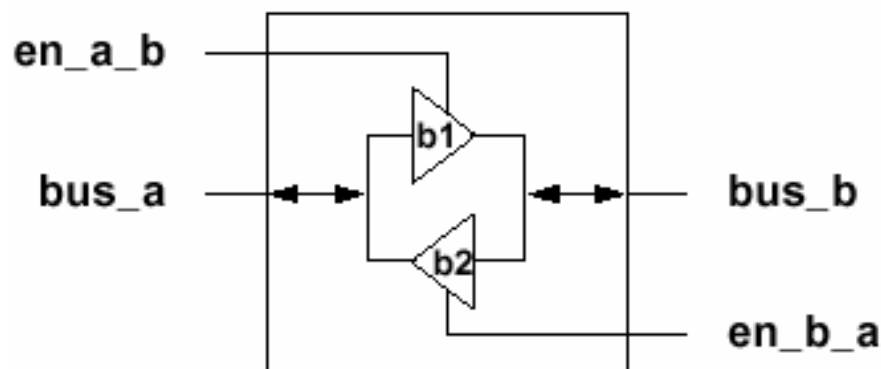
```
inout [7:0] databus;
```

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。
 - ✓ 这样仿真器若有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。

双向端口建模 — 使用基本单元建模

信号en_a_b和en_b_a控制使能



```
module bus_xcvr( bus_a, bus_b, en_a_b,  
en_b_a);
```

```
    inout bus_a, bus_b;
```

```
    input en_a_b, en_b_a;
```

```
    bufif1 b1 (bus_b, bus_a, en_a_b);
```

```
    bufif1 b2 (bus_a, bus_b, en_b_a);
```

```
    // Structural module logic
```

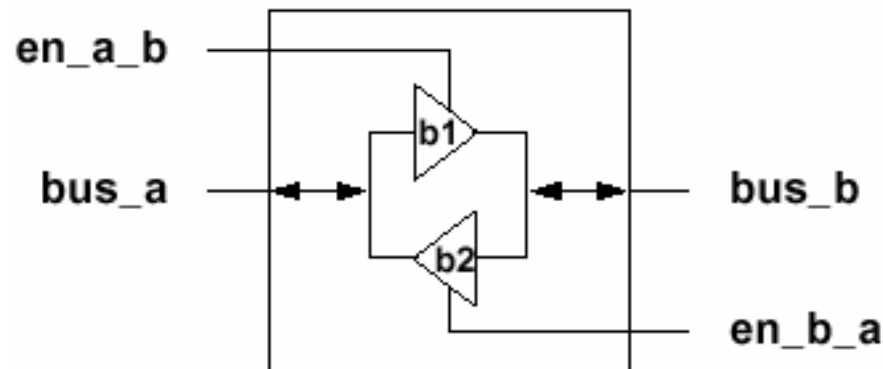
```
endmodule
```

若en_a_b=1，基本单元b1使能，bus_a数据传送到bus_b

若en_b_a=1，基本单元b2使能，bus_b数据传送到bus_a

双向端口建模 — 使用持续赋值建模

信号en_a_b和en_b_a控制使能



```
module bus_xcvr( bus_a, bus_b, en_a_b,  
en_b_a);
```

```
    inout bus_a, bus_b;
```

```
    input en_a_b, en_b_a;
```

```
    assign bus_b = en_a_b ? bus_a : 'bz;
```

```
    assign bus_a = en_b_a ? bus_b : 'bz;
```

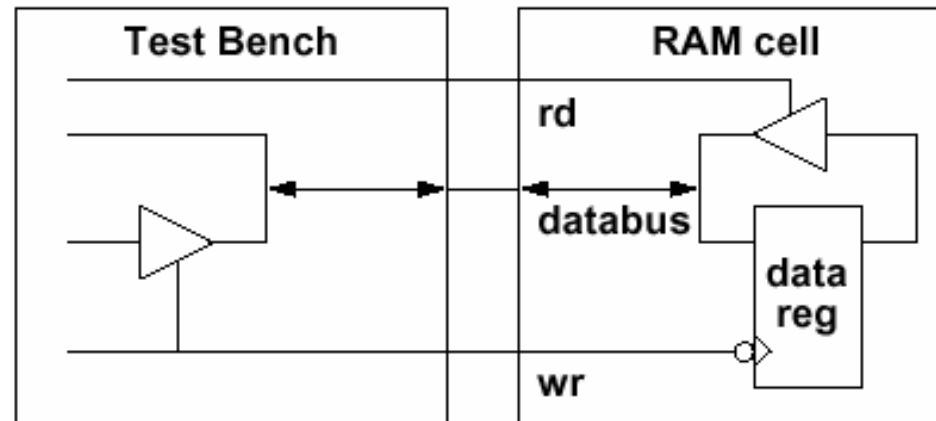
```
// Structural module logic
```

```
endmodule
```

若en_a_b=1，赋值语句
驱动bus_a数据到bus_b

若en_b_a=1，赋值语句
驱动bus_b值到bus_a

双向端口建模 — 存储器端口建模



```
module ram_cell( databus, rd, wr);  
    inout databus;  
    input rd, wr;  
    reg datareg;  
    assign databus = rd ? datareg :  
        'bz;  
    always @( negedge wr)  
        datareg <= databus;  
endmodule
```

当rd=1时，datareg的
值赋值databus

在wr下降沿，databus
数据写入datareg

复习

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。

第17章 Verilog中的高级结构

学习内容:

- 任务和函数的定义和调用
- 怎样使用命名块
- 怎样禁止命名块和任务
- 有限状态机（FSM）及建模

Verilog的任务及函数

结构化设计是将任务分解为较小的，更易管理的单元，并将可重用代码进行封装。这通过将设计分成模块，或任务和函数实现。

- 任务 (**task**)
 - ✓ 通常用于调试，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
- 函数(**function**)
 - ✓ 通常用于计算，或描述组合逻辑
 - ✓ 不能包含任何延迟；函数仿真时间为0
 - ✓ 只含有**input**参数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

Verilog的任务及函数

- 任务和函数必须在**module**内调用
- 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

任务

下面的任务中含有时序控制和一个输入，并引用了一个**module**变量，但没有输出、输入输出和内部变量，也不显示任何结果。

时序控制中使用的信号（例如**ck**）一定不能作为任务的输入，因为输入值只向该任务传送一次。

```
module top;
  reg clk, a, b;
  DUT u1 (out, a, b, clk);
  always #5 clk = !clk;
  task neg_clocks;
    input [31:0] number_of_edges;
    repeat( number_of_edges) @( negedge clk);
  endtask
  initial begin
    clk = 0; a = 1; b = 1;
    neg_clocks(3); // 任务调用
    a = 0; neg_clocks (5);
    b = 0;
  end
endmodule
```

任务

主要特点:

- 任务可以有- 传送到任务的参数和与任务I/O说明顺序相同。尽管传送到任务的参数名称与任务内部I/O说明的名字可以相同，但在实际中这通常不是一个好的方法。参数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的只有一个拷贝，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。

在任务或函数中引用调用模块的变量时要小心。如果想使任务或函数能从另一个模块调用，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

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。但没有双向端口，也没有显示。

任务调用时的参数按任务定义的顺序列出。

```
module mult (clk, a, b, out, en_mult);  
  input clk, en_mult;  
  input [3: 0] a, b;  
  output [7: 0] out;  
  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, tome;  
    output [7: 0] result;  
    wait (en_mult)  
    result = xme * tome;  
  endtask  
endmodule
```

函数 (function)

```
module orand (a, b, c, d, e, out);  
    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，但调用它的过程可以有时序控制。

函数名 **f_or_and** 在函数中作为 **register** 使用

函数

主要特性:

- 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**reg**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。
- 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，但返回的值可以直接给信号连接赋值。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

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。函数中需要多条语句时用**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 [7: 0] goo;
    // 可以持续赋值中调用函数
    wire [7: 0] goo = zero_count ( loo );
    function [3: 0] zero_count;
        input [7: 0] in_ bus;
        integer l;
        begin
            zero_count = 0;
            for (l = 0; l < 8; l = l + 1)
                if (! in_ bus[ l ])
                    zero_count = zero_count + 1;
        end
    endfunction
endmodule
```


函数

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 (neg, a, b);
```

```
    output neg;
```

```
    reg neg;
```

```
    input 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 or b)
```

```
    if (subtr( a, b) < 0)
```

```
        neg = 1;
```

```
    else
```

```
        neg = 0;
```

```
endmodule
```

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+ 1)] = data  
[K];  
endfunction  
always @ (posedge clk)  
    D = reverse_bits (D) ;  
...
```

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[定义任务和函数
```

.....

禁止命名块和任务

- **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

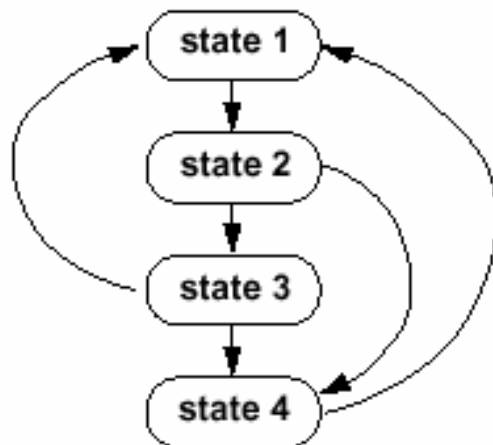
disable <任务名称>

- 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。

有限状态机

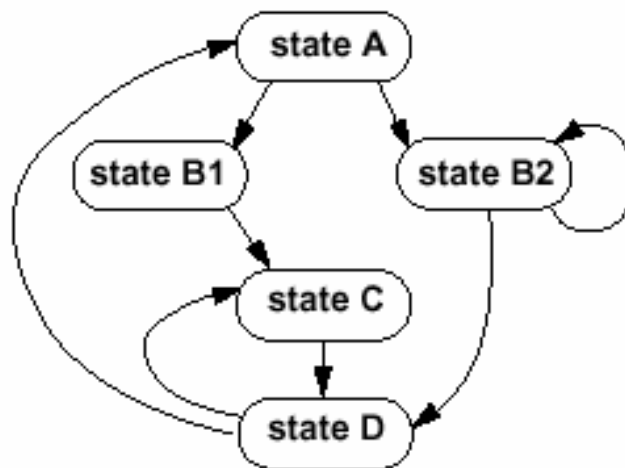
隐式状态机FSM

- 不需要声明状态寄存器
- 仿真效率高
- 只适合于线性的状态改变
- 大多数综合工具不能处理



显式FSM:

- 利于结构化
- 易于处理缺省条件
- 能处理复杂的状态改变
- 所有综合工具都支持



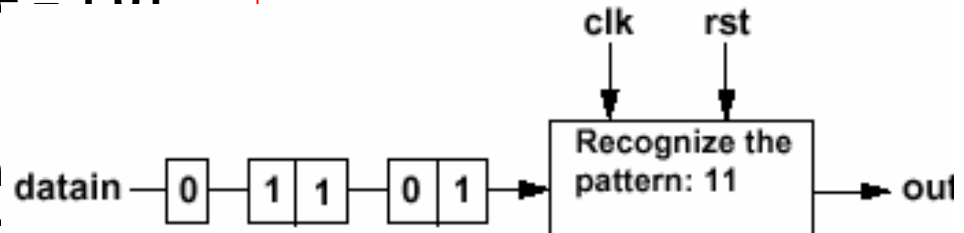
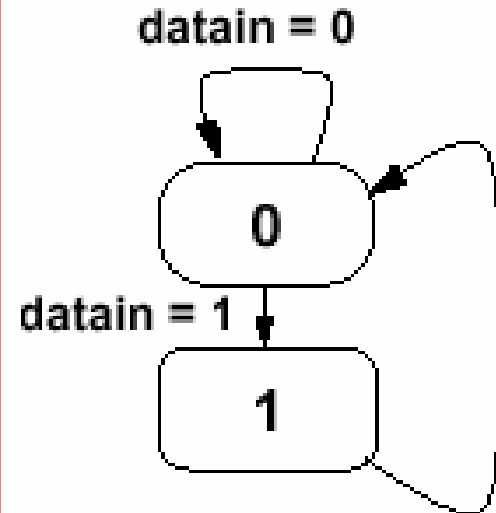
有限状态机

- 在隐式FSM中，只要数据在一个时钟沿写入并在另一个周期读出，则会生成寄存器。
- 所有FSM必须有复位，且状态改变必须在单一时钟信号的同一边沿。
- 通常，如果状态改变简单明确，且综合工具接受隐式状态机，就可以使用隐式类型。如果状态改变很复杂，则显式类型更加有效。
- 隐式状态机是一个行为级而非RTL代码的典型例子。这种代码依赖循环和内嵌时序控制，有时也有命名事件、**wait**和**disable**语句。因此，隐式状态机在综合时通常不被支持。
- 线性FSM是指从一个状态到下一个状态的转换不需要任何条件。

显式有限状态机

```

module exp (out, datain, clk, rst);
  input clk, rst, datain;
  output out; reg out;
  reg state;
  always @( posedge clk or posedge rst)
    if (rst) {state, out} = 2'b00;
    else
      case (state)
        1'b0: begin
          out = 1'b0;
          if (! datain) state =
1'b0;
          else state = 1'b1;
        end
        1'b1: begin
          out = datain;
          state = 1'b0;
        end
        default: {state, out} = 2'b00;
      endcase
endmodule
  
```



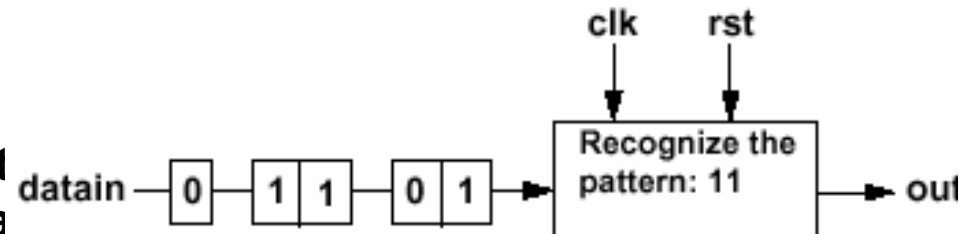
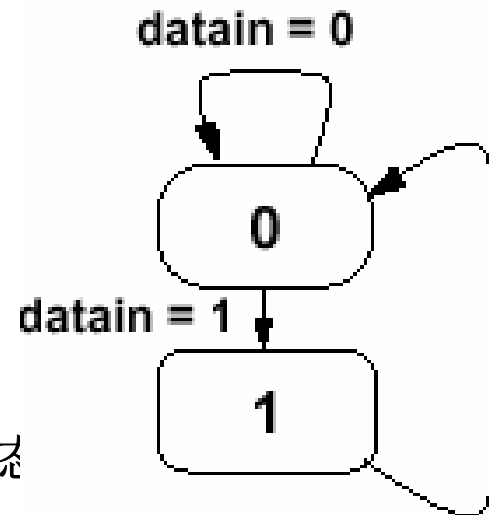
显式有限状态机

- 可以在过程块中用单一时钟边沿和**case**语句显式地描述**FSM**。
- 必须声明定义状态机的状态状态变量。
- 要改变当前状态，必须在时钟边沿改变状态变量的值。
- 给通常不会发生的条件指定缺省动作是一个很好的描述方式。

隐式有限状态机

```

module imp (out, datain, clk, rst);
    output out; reg out;
    input clk, datain, rst;
    always @( rst) // Synergy reset method
        if (rst) assign out = 1'b0;
    else
        begin
            deassign out;
            disable seq_block; //返回初始状态
        end
    always @( posedge clk)
    begin: seq_block
        out = 1'b0;
        if (!datain) // 状态1: output =
            disable seq_block;
        // 状态2: output = 2nd bit
        @( posedge clk) out = da
    end
endmodule
    
```



隐式有限状态机

- 可以在过程块中用多个时钟边沿（每个状态一个），条件语句，循环，和**disable**语句隐式地描述一个FSM。通常不可综合。
- 不必声明状态变量。
- 在下一个时钟边沿改变状态，除非迫使状态重复。例如在一个循环中或用一个**disable**语句。下一个状态可以由条件语句决定。

复习

1. 在Verilog中什么结构能产生一个新的“范围”？
2. 哪些结构可以被禁止？
3. 什么时候一个函数比一个任务更合适？反过来呢？

解答

1. 模块，任务，函数，和命名块。Verilog中模块作为主要层次分割方法。函数和任务提供附加的代码分割和封装方法。
2. 命名块和任务可以被禁止。
3. 函数更适用于组合逻辑描述，并且使用灵活（例如在一个持续赋值的右边或在一个端口列表里）。如果需要时序控制，则任务更适合。任务还可以被禁止。

第18章 用户定义基本单元

学习内容:

- 学习如何使用用户定义基本单元进行逻辑设计

术语及定义

- **UDP:** 用户定义基本单元，其行为和Verilog内部的基本单元相似。其功能用真值表定义。

什么是UDP

在Verilog结构级描述中，可以使用：

- 二十多个内部门级基本单元
- 用户自定义基本单元

UDP在ASIC库单元开发、中小型芯片设计中很有用

- 可以使用**UDP**扩充已定义的基本单元集
- **UDP**是自包容的，也就是不需要实例化其它模块
- **UDP**可以表示时序元件和组合元件
- **UDP**的行为由真值表表示
- **UDP**实例化与基本单元实例化相同

什么是UDP

- 可以使用**UDP**扩充已定义的基本单元集
- **UDP**是一种非常紧凑的逻辑表示方法。
- **UDP**可以减少消极（**pessimism**）因素，因为一个**input**上的**x**不会像基本单元那样自动传送到**output**。
- 一个**UDP**可以替代多个基本单元构成的逻辑，因此可以大幅减少仿真时间和存储需求。相同逻辑的行为级模型甚至可以更快，这取决于仿真器。

UDP的特点

- **UDP只能有一个输出**

如果在功能上要求有多个输出，则需要在**UDP**输出端连接其它的基本单元，或者同时使用几个**UDP**。

- **UDP可以有1到10个输入**

若输入端口超过**5**，存储需求会大幅增加。下表列出输入端口数与存储需求的关系。

#输入	存储器（KB）	#输入	存储器（KB）
1-5	1	8	56
6	5	9	187
7	17	10	623

- 所有端口必须为标量且不允许双向端口
- 不支持逻辑值**Z**
- 输出端口必须列为端口列表的第一个
- 时序**UDP**输出端可以用**initial**语句初始化为一个确定值。
- **UDP**不可综合

组合逻辑举例：2-1多路器

UDP名称

```
primitive multiplexer (o, a, b, s);
```

```
output o;
```

```
input s, a, b;
```

```
table
```

```
// a b s : o
```

```
0 ? 1 : 0;
```

```
1 ? 1 : 1;
```

```
? 0 0 : 0;
```

```
? 1 0 : 1;
```

```
0 0 x : 0;
```

```
1 1 x : 1;
```

```
endtable
```

```
endprimitive
```

输出端口

真值表中?
表示的逻辑
值为: 0、1
或x

这两行表示不管b为何值, 若s为1, o输出a值

这两行表示不管a为何值, 若s为0, o输出b值

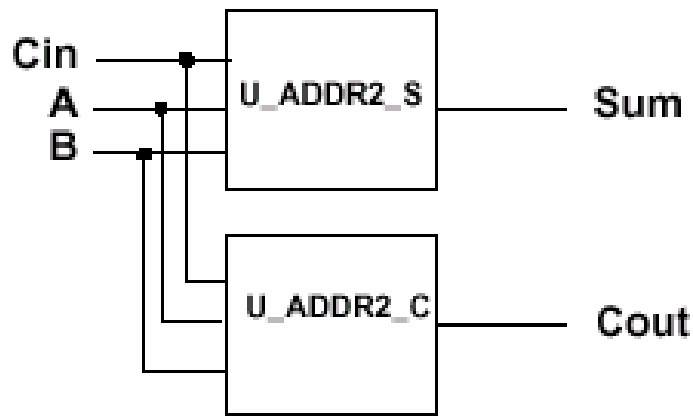
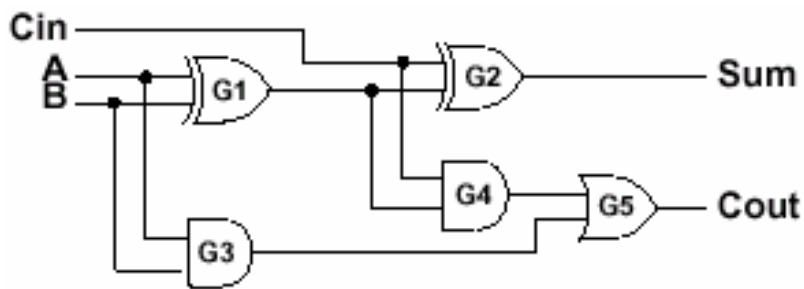
这两行用于减少消极因素。表示若a, b有相同逻辑值, 即使sel=x, o也输出与a, b相同的值。Verilog内部基本单元不能描述这种行为。

UDP将X作为真实世界的未知值(0或1), 而不是Verilog值, 描述也更为精确。

- UDP在模块(module)外部定义。
- 没有在真值表中说明的输入组合, 输出X。
- 真值表中输入信号按端口列表顺序给出。

组合逻辑举例：全加器

全加器可以由两个组合逻辑UDP实现



```
// FULL ADDER CARRY-OUT TERM
primitive U_ADDR2_C (CO, A, B, CI);
  output CO;
  input A, B, Ci;
  table // A B CI : CO
    1 1 ? : 1;
    1 ? 1 : 1;
    ? 1 1 : 1;
    0 0 ? : 0;
    0 ? 0 : 0;
    ? 0 0 : 0;

  endtable
endprimitive
```

```
// FULL ADDER SUM TERM
primitive U_ADDR2_S (S, A, B, CI);
  output S;
  input A, B, Ci;
  table // A B CI : S
    0 0 0 : 0;
    0 0 1 : 1;
    0 1 0 : 1;
    0 1 1 : 0;
    1 0 0 : 1;
    1 0 1 : 0;
    1 1 0 : 0;
    1 1 1 : 1;

  endtable
endprimitive
```

组合逻辑举例：全加器

全加器可以由两个组合逻辑**UDP**实现，而不使用内部基本单元。

- 当需要大量全加器时，可以大幅度减少存储器需求
- 大幅减小事件数目。使用内部基本单元时，事件通过**3**个基本单元后才能到达进位输出；而使用**UDP**，事件只需经过一个基本单元。

电平敏感时序元件举例：锁存器latch

```
primitive latch (q, clock, data);  
  output q;  
  reg q;  
  input clock, data;  
  initial q = 1'b1;  
  table  
    // clock data current state next state  
    //      0    1    :    ?    :    1    ;  
    //      0    0    :    ?    :    0    ;  
    //      1    ?    :    ?    :    ?    ;  
  endtable  
endprimitive
```

输出必须声明为reg
以保存前一状态

时序UDP初始化语句，将输出初始化为

用另一个场表示下一状态

‘-’状态值表示
输出没有变化

输入及当前状态中的
的? 表示无关值

- 锁存器的行为如下：
 - ✓ 当时钟输入为0时，data输入的值传送到输出。
 - ✓ 当时钟输入为1时，输出不变。
- 这种加电初始化在实际元件中很少见，但在UDP功能测试时很有用。

边沿敏感时序元件举例：D触发器

```
primitive d_edge_ff (q, clk,
data);
    output q;
    input clk, data;
    reg q;
    table // clk dat state next
        (01) 0 : ? : 0 ;
        (01) 1 : ? : 1 ;
        (0x) 1 : 1 : 1 ;
        (0x) 0 : 0 : 0 ;
        (x1) 0 : 0 : 0 ;
        (x1) 1 : 1 : 1 ;
    //忽略时钟下降沿
        (?0) ? : ? : - ;
        (1x) ? : ? : - ;
    //时钟稳定时忽略data变化
        ?  (??) : ? : - ;
    endtable
```

endprimitive

- 表里有边沿项表示输入跳变。
- 在一条入口语句中只能说明一个输入跳变，因为Verilog仿真是基于事件，一次只允许一个事件发生。
- 在每个时间步中，电平入口优先于边沿入口，因为电平最后处理。因此，下面的出口：

(? 0) ? : ? : - ;

可由下式取代：

0 ? : ? : - ;

两个都给出时，只有后者起作用

- 在任何真值表入口语句中只能说明一个输入跳变。
- 如果说明了任何输入跳变，则必须说明所有输入上的所有跳变。

提高可读性的简写形式

Verilog中有一些符号可用于**UDP**真值表中以提高可读性

符号	表示	解释
-	没有变化	时序元件输出的下一个值与当前值相同
?	0、1或x	任何值
b	0或1	任何确定值
r	(01)	0->1跳变
f	(10)	1->0跳变
p	(01)、(0x)或(x1)	任何上升沿(posedge)
n	(10)、(1x)或(x0)	任何下降沿(negedge)
*	(??)	任何跳变

提高可读性的简写形式

```
table
// clk dat state next
  r  0 : ?  : 0 ;
  r  1 : ?  : 1 ;
  (0x) 1 : 1  : 1 ;
  (0x) 0 : 0  : 0 ;
  (x1) 1 : 1  : 1 ;
  (x1) 0 : 0  : 0 ;
// 忽略时钟的下降沿
  n  ?  : ?  : - ;
// 忽略时钟稳定时的任何数据变化
  ? *  : ?  : - ;
endtable
```


带同步复位的D触发器

```
primitive U_ff_p_cl( q, d, clk, cl);
```

```
    input d, clk, cl;
```

```
    output q;
```

```
    reg q;
```

```
table
```

// d	clk	cl	:q	:q+ 1
1	r	1	:?	: 1;
0	r	?	:?	: 0;
?	r	0	:?	: 0;
?	p	0	:0	: -;
1	p	1	:1	: -;
0	p	?	:0	: -;
?	n	?	:?	: -;
*	?	?	:?	: -;
d				

```
// clock 1
```

```
// clock 0
```

```
// reset
```

```
// reducing pessimism
```

```
// ignore falling clk
```

```
// ignore changes on
```

```
    ?      ?      *      :?      : -; // ignore changes on
```

```
clk
```

```
endtable
```

```
endprimitive
```

带使能和复位的锁存器

当使能g为高(H)时，锁存器锁存d；只有当g为低时复位信号cl才有效（高有效）。

```
primitive u_latch_cl (q, d, g, cl);
```

```
output q;
```

```
input d, g, cl;
```

```
reg q;
```

```
table
```

```
// d      g      cl      :q      :q+ 1
```

```
0      1      ?      :?      : 0;
```

```
1      1      ?      :?      : 1;
```

```
1      ?      0      :1      : 1;
```

```
// reducing
```

```
pessimism
```

```
0      ?      0      :0      : 0;
```

```
// reducing
```

```
pessimism
```

```
0      ?      1      :?      : 0;
```

```
// reducing
```

```
pessimism
```

```
0      ?      ?      :0      : 0;
```

```
// reducing
```

```
pessimism
```

```
?      0      0      :?      : -;
```

```
// latch disabled
```

```
?      0      1      :?      : 0;
```

```
// clear
```

```
endtable
```

```
endprimitive
```

使用通报符(notifier)的寄存器

下面的例子是异步复位的上升沿D触发器，有时序检查和路径延迟。
这个模型使用了一个UDP，并将通报符作为UDP的一个输入。

```
`timescale 1ns/ 1ns
module dff_nt (q, ck, d, rst);
input ck, d, rst;
output q;
reg nt;
U_FFD_RB i1 (q, d, ck, rst, nt);
specify
    specparam tsu = 2;
    (ck => q) = (2: 3: 4);
    $setup(d, posedge ck, tsu,
nt);
endspecify
endmodule
```

```
primitive U_FFD_RB (Q, D, CP, RB,NT);
output Q; reg Q;
input D, CP, RB, NT;
table
```

// D	CP	RB	NT	:Q	:Q+1
0	r	?	?	:?	: 0; // clock a 0
1	r	1	?	:?	: 1; // clock a 1
1	p	1	?	:1	: -; // reducing pessimism
0	p	?	?	:0	: -; // reducing pessimism
?	?	0	?	:?	: 0; // asynchronous reset
?	?	x	?	:0	: -; // reducing pessimism
?	n	?	?	:?	: -; // ignore falling clock
*	?	?	?	:?	: -; // ignore rising edges
?	?	*	?	:?	: -; // ignore changes on

第19章 Verilog的可综合描述风格

学习目标:

学习组合逻辑和时序逻辑的可综合的描述风格及技术，包括：

- 不支持的**Verilog**结构
- 过程块
- 寄存器
- 敏感列表
- 持续赋值
- 综合指导
- 条件结构
- 阻塞及非阻塞赋值
- 锁存器/**MUX**推断
- 函数**function**
- 任务**task**
- 复位
- 有限状态机**FSM**
- 宏库及设计复用

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**。

如果逻辑暗示存储则为**时序逻辑**。如果输出在任何给定时刻不能由输入的状态决定，则暗示存储。

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产生什么类型输出，

并能够反过来确定为什么所用的综合工具产生这个输出，

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

assign deassign

非结构化的**for**语句

force release

数据类型:

操作符:

event

==

real

!=

time

过程块

- 任意边沿

- 在所有输入信号的任意边沿进入的过程块产生组合逻辑。这种过程块称为组合块。

```
always @( a or b) // 与门  
    y = a & b;
```

- 单个边沿

- 在一个控制信号的单一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;
```

过程块中的寄存器类型

若同步块中使用一个`reg`，则：

- 如果在一个时钟周期赋值并在另一个周期被采样，则只能以硬件寄存器实现。
- 如果`reg`还是一个基本输出，它会出现于综合网表中，但不一定是一个硬件寄存器。
- 若两者都不是，该信号可能被优化掉。

若组合块中使用一个`reg`，则：

- 如果`reg`值随块的任何一个输入的变化而改变，则在综合时不会产生硬件寄存器。
- 如果`reg`值并不总是随块的输入变化而改变，则综合时会产生一个锁存器。

同步寄存器举例

在这个例子中，**rega**只作暂存，因此会被优化掉。

```
module ex1reg (d, clk, q);  
  input d, clk;  
  output q;  
  reg q, rega;  
  always @(posedge clk)  
  begin  
    rega = 0;  
    if (d) rega = 1;  
    q = rega;  
  end  
endmodule
```

在这个例子中，**rega**产生一个寄存器，不会被优化掉。

```
module ex2reg (d, clk, q);  
  input d, clk;  
  output q;  
  reg q, rega;  
  always @(posedge clk)  
  begin  
    rega = 0;  
    if (d) rega = 1;  
  end  
  always @(posedge clk)  
  q = rega;  
endmodule
```

组合逻辑中的寄存器类型举例

在下面的例子， **rega**是暂存变量，并被优化掉

在这个例子中，**y**和**rega**总是赋新值，因此产生一个纯组合逻辑。

```
module ex3reg (y, a, b, c);
input a, b, c;
output y;
reg y, rega;
always @( a or b or c)
begin
    if (a & b)
        rega = c;
    else
        rega = 0;
    y = rega;
end
endmodule
```

在这个例子中，**rega**不总是产生新值，因此会产生一个锁存器，**y**是锁存器的输出

```
module ex4reg (y, a, b, c);
input a, b, c;
output y;
reg y, rega;
always @( a or b or c)
begin
    if (a & b)
        rega = c;
    y = rega;
end
endmodule
```

敏感列表

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 (a, q, 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。

敏感列表

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。

上述两例综合结果（**SYNOPSYS**）相同，但**RTL**描述的仿真结果不同。也就是左边的敏感表不完全的例子的**RTL**描述和综合出的网表的仿真结果不同。

```
module sens_t;
    reg a, b, sl;
    sens u1(a, q, b, sl);
    sensc u2(qc, a, b, sl);
    initial
    begin
        $monitor($time, " %b %b %b %b %b", a, b, sl, q, qc);
        a =0;b=0;sl = 0;
        #10 a =1;
        #10 sl = 1;
        #10 sl = 0;
        #10 $finish;
    end
endmodule
```

0	0	0	0	0	0
---	---	---	---	---	---

10	1	0	0	0	1
----	---	---	---	---	---

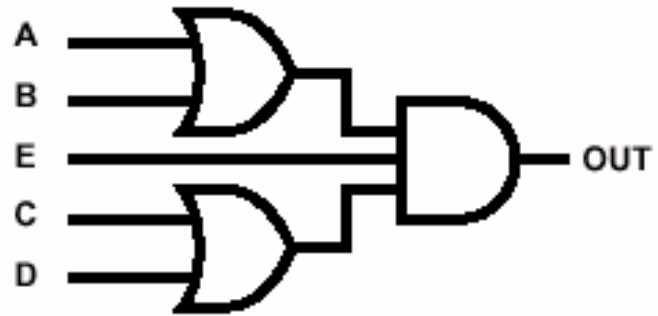
20	1	0	1	0	0
----	---	---	---	---	---

30	1	0	0	1	1
----	---	---	---	---	---

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



过程持续赋值

过程持续赋值是在过程块（**always**或**initial**）内给一个寄存器数据类型进行的持续赋值。这在大多数综合工具中是非法的。

```
module latch_quasi (q, en, d);  
    input en, d;  
    output q;  
    reg q;  
    always @( en)  
        if (en)  
            assign q = d;  
        else  
            deassign q;  
endmodule
```

综合指示

- 大多数综合工具都能处理综合指示。
- 综合指示可以嵌在**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示在语法上不同。但其目的相同，都是在**RTL**代码内部进行最优化。
- 通常综合指示中包含工具或公司的名称。例如，下面介绍的**Envisia Ambit synthesis**工具的编译指示都以**ambit synthesis**开头。

综合指示

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

```
// ambit synthesis on
```

```
// ambit synthesis off
```

```
// ambit synthesis case = full, parallel, mux
```

结构指示

```
// ambit synthesis architecture = cla or rpl
```

FSM指示

```
// ambit synthesis enum xyz
```

```
// ambit synthesis state_vector sig state_vector_flag
```


综合指示 — 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指示按下面所示指示优化器：

- **//ambit synthesis case = parallel**
 - 建立并行的编码逻辑，彼此无优先级。
- **//ambit synthesis case = mux**
 - 若库中有多路器，使用多路器建立编码逻辑。
- **//ambit 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。

条件语句

自然完全的条件语句

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

若 a 变为
0,

```
module inccase (a, b, c, d, e);  
  input a, b, c, d;  
  output e;  
  reg e;  
  always @( a or b or c or d)  
    case ({ a, b})  
      2'b11: e = d;  
      2'b10: e = ~c;  
    endcase  
endmodule
```

```
module incpif (a, b, c, d, e);  
  input a, b, c, d;  
  output e;  
  reg e;  
  always @( a or b or c or d)  
    if (a & b)  
      e = d;  
    else if (a & ~b)  
      e = ~c;  
endmodule
```

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

```
module compif (a, b, c, d, e);  
  input a, b, c, d;  
  output e;  
  reg e;  
  always @( a or b or c or 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 d;  
    reg d;  
    always @( a or b or c)  
        case (a) // ambit synthesis case  
        = full  
            2'b00: d = b;  
            2'b01: d = c;  
        endcase  
endmodule
```

和前例一样，没有定义所有case项，但综合指示通知优化器缺少的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。

case指示例外

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**。

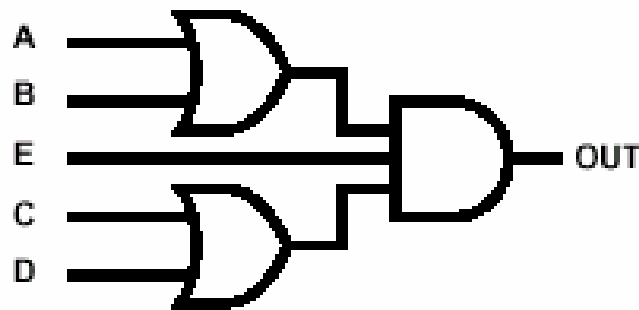
```
module select (a, b, sl);  
    input [1: 0] sl;  
    output a, b;  
    reg a, b;  
    always @( sl)  
        case (sl) // ambit synthesis case  
        = full  
            2b'00: begin a = 0; b = 0; end  
            2b'01: begin a = 1; b = 1; end  
            2b'10: begin a = 0; b = 1; end  
            2b'11: b = 1;  
            default: begin a = 'bx; b = 'bx;  
        end  
    endcase  
endmodule
```

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 (out, a, b, c, d, e);  
  input a, b, c, d, e;  
  output out; 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
```



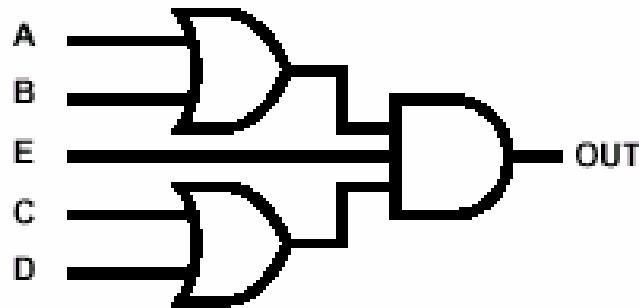
任务

任务一般只在测试基准使用，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：

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



锁存器（latch）推断

在**always**块中，如果没有说明所有条件，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，综合时将产生一个存储元件

```
module latch (q, data, enable);  
    input data, enable;  
    output q;  
    reg q;  
    always @( enable or data)  
        if (enable)  
            q = data;  
endmodule
```

同步反馈(feedback)推断

综合工具一般不支持组合逻辑反馈，但支持同步反馈。

在同步过程块中，如果条件语句的一个分支没有给所有输出赋值，则推断出反馈。

有反馈：

```
module dffn (q, d, clk, en);  
  input d, clk, en;  
  output q;  
  reg q;  
  always @( negedge clk)  
    if (en)  
      q <= d;  
endmodule
```

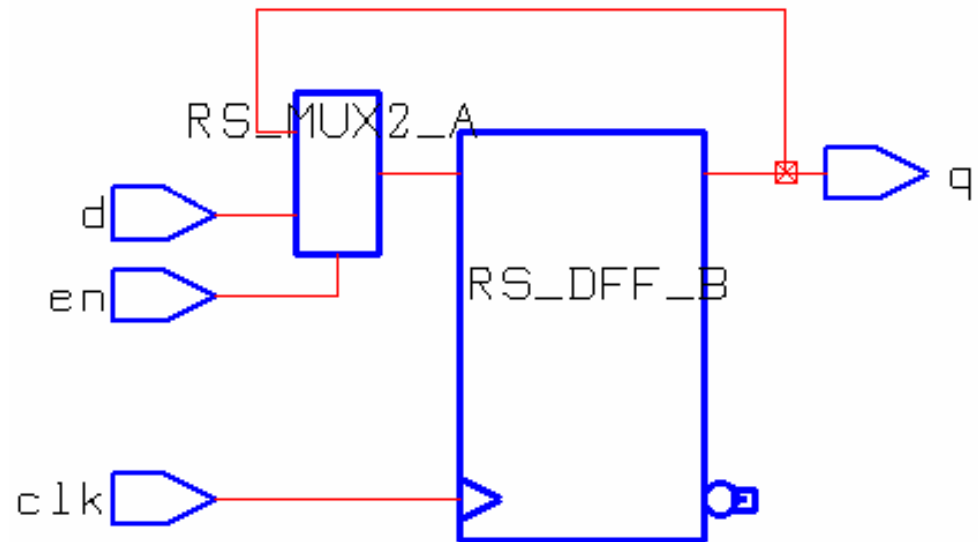
无反馈：

```
module dffn (q, d, clk, en);  
  input d, clk, en;  
  output q;  
  reg q;  
  always @( negedge clk)  
    if (en)  
      q <= d;  
    else  
      q <= 'bx;  
endmodule
```

带使能的寄存器

上述带反馈的描述用于带使能端的寄存器的描述。在寄存器的描述中，敏感列表是不完全的。

```
module dffn (q, d, clk, en);  
  input d, clk, en;  
  output q;  
  reg q;  
  always @( negedge clk)  
    if (en)  
      q <= d;  
endmodule
```



阻塞或非阻塞

使用的赋值类型依赖于所描述的逻辑类型：

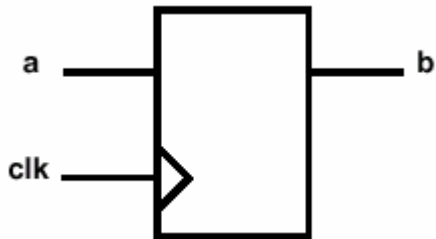
- 在时序块**RTL**代码中使用非阻塞赋值
 - 非阻塞赋值保存值直到时间片段的结束，从而避免仿真时的竞争情况或结果的不确定性
- 在组合的**RTL**代码中使用阻塞赋值
 - 阻塞赋值立即执行

阻塞、非阻塞对比

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。

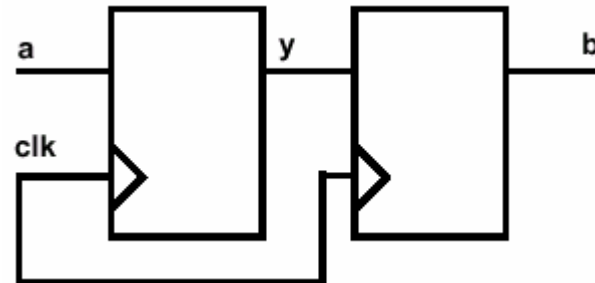
使用阻塞赋值，此描述综合出一个D flip-flop:

```
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
```



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( q, ck, r, d);  
  input ck, d, rst;  
  output q;  
  reg q;  
  always @( negedge ck)  
    if (r)  
      q <= 0;  
    else  
      q <= d;  
endmodule
```

同步块的异步复位

```
module async( q, ck, r, d);  
  input ck, d, r;  
  output q;  
  reg q;  
  always @( negedge ck or  
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。

复位

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 @( negedge ck)  
        if ( !r ) q <= d;  
    always @( posedge r)  
        q <= 0;  
endmodule
```

带复位、置位的锁存器latch

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 (q, enable, set, clr, d);  
    input enable, d, set, clr;  
    output q;  
    reg q;  
    always @( enable or set or clr or  
d)  
    begin  
        if (set)  
            q <= 1;  
        else if (clr)  
            q <= 0;  
        else if (enable)  
            q <= d;  
    end  
endmodule
```


有限状态机

有限状态机有两种不同类型 — 显式和隐式。

- 隐式状态机用多个 `@(posedge clk)` 语句指出状态跳变。
 - 隐式状态机的抽象级比显式状态机高，通常不可综合。
- 显式状态机用 `case` 语句显式定义每个可能状态。
 - 通常，显式状态机用于可综合代码描述。

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
```

有限状态机FSM指导

状态机的描述也有综合指导。在RTL代码中，**FSM**指导向优化器传递状态机有关的特性信息。

这些指导有：

- **enum**指导
 - 状态赋值枚举，也用来将状态赋值捆绑到状态向量。
- **state_vector**指导
 - 定义状态寄存器和编码类型

FSM指导

```
`timescale 1ns/ 100ps
module state4 (clock, reset, out);
    input reset, clock;
    output [1: 0] out;
    reg [1: 0] out;
    parameter /* ambit synthesis enum state_info */
        stateA = 2'b00,
        stateB = 2'b01,
        stateC = 2'b10,
        stateD = 2'b11;
    reg [1: 0] /* ambit synthesis enum state_info */ state;
    reg [1: 0] /* ambit synthesis enum state_info */ nextstate;
    always @(posedge clock)
        /* ambit synthesis state_vector state -encoding one_hot
        */
        if (reset)
            state <= stateA;
        else
            state <= nextstate;
    ...
endmodule
```

枚举名称定义

枚举名称限用于state、
nextstate向量

定义状态寄存器
并指定编码格式

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)
```

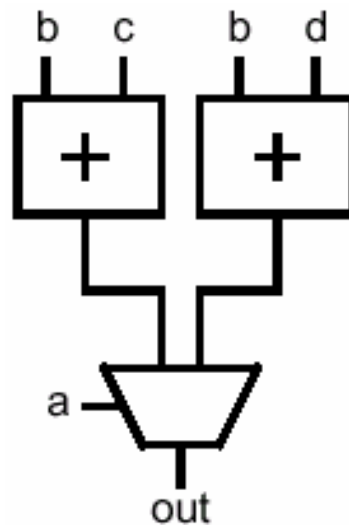
```
    out = b + c;
```

```
  else
```

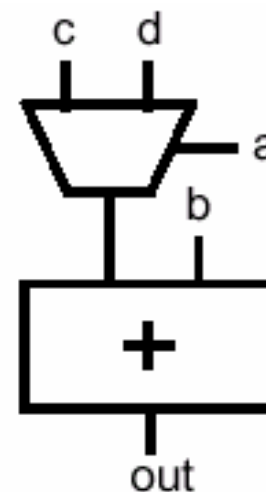
```
    out = b + d;
```

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

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);
```

复杂操作符

复杂操作符是可以被识别为高层操作并被直接映射到一个向量库内以存在单元的操作。例如：

out = a * b;

- 大多数工具可以将它映射为一个乘法器。
- 专用宏单元库中可能有乘法器。宏单元库中的元件的复杂程度要比常规单元库高。
- 宏单元库可以包含部分可重用设计，如**FIFO**，加法器，减法器（各种结构），移位寄存器，计数器和解码器等。
- 宏单元库还可以包括用户自定义的可重用块，由用户自己设计并综合。

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
- 总做得和你一样好

综合工具不能胜任的工作

综合工具善于优化组合逻辑。但设计中有很大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择大的单元或宏单元，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，只要有足够的时间。

可编程逻辑器件相关问题

- 迄今为止，很多注释假定综合为**ASIC**。对**FPGA**，存在一些不同问题。所有**ASIC**综合工具以同样的方式工作，使用同样的优化算法，且目标工艺库的单元也是相似的。众所周知，**FPGA**使用不同的技术，**EEPROM**，**SRAM**和**anti-fuse**，且每个开发商有不同的构造块，这些块比**ASIC**使用的基本门要大很多。
- 对特定的**FPGA**，其结构是固定的，限制了静态时序分析的效用，因为**ASIC**的路径延迟变化的范围很大，而**FPGA**是可预测的。而且，**FPGA**设计时通常使用专用结构，而这些专用结构很难由综合工具实现。
- 因此**Verilog**代码需要技术与工艺专用指导，这些指导由**FPGA**开发商提供的软件的算法识别，而仿真时忽略。这些指导通常由用户自定义属性，注释或直接实例化专用单元。这会限制**Verilog**的技术独立性和可移植性。
- 关键问题是综合工具能够多好地处理所选的技术。要得到一个好的结果，需要结构专用算法。

第21章 SDF时序标注

学习内容:

- 延迟计算器
- 标准延迟格式（**Standard Delay Format**）(**SDF**)
- 标注**SDF**数据

术语及定义

- **CTLF**: (**C**ompiled **T**iming **L**ibrary **F**ormat) 编译的时序库格式。特定工艺元件数据的标准格式。
- **GCF**: (**G**eneral **c**onstraint **F**ormat)通用约束格式。约束数据的标准格式。
- **MIPD**: (**M**odule **I**nterconnect **P**ort **D**elay)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**: (**M**ulti-source **I**nterconnect **T**ransport **D**elay)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**: (**P**rogramming **L**anguage **I**nterface) 编程语言界面。基于C的对Verilog数据结构的程序访问。
- **SDF**: **S**tandard **D**elay **F**ormat.(标准延迟格式)。时序数据OVI标准格式。
- **SITD**: **S**ingle-**S**ource **I**nterconnect **T**ransport **D**elay, 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**: **S**tandard **P**arasitic **F**ormat. (标准寄生参数格式)。提取的寄生参数数据的标准格式。

时序标注

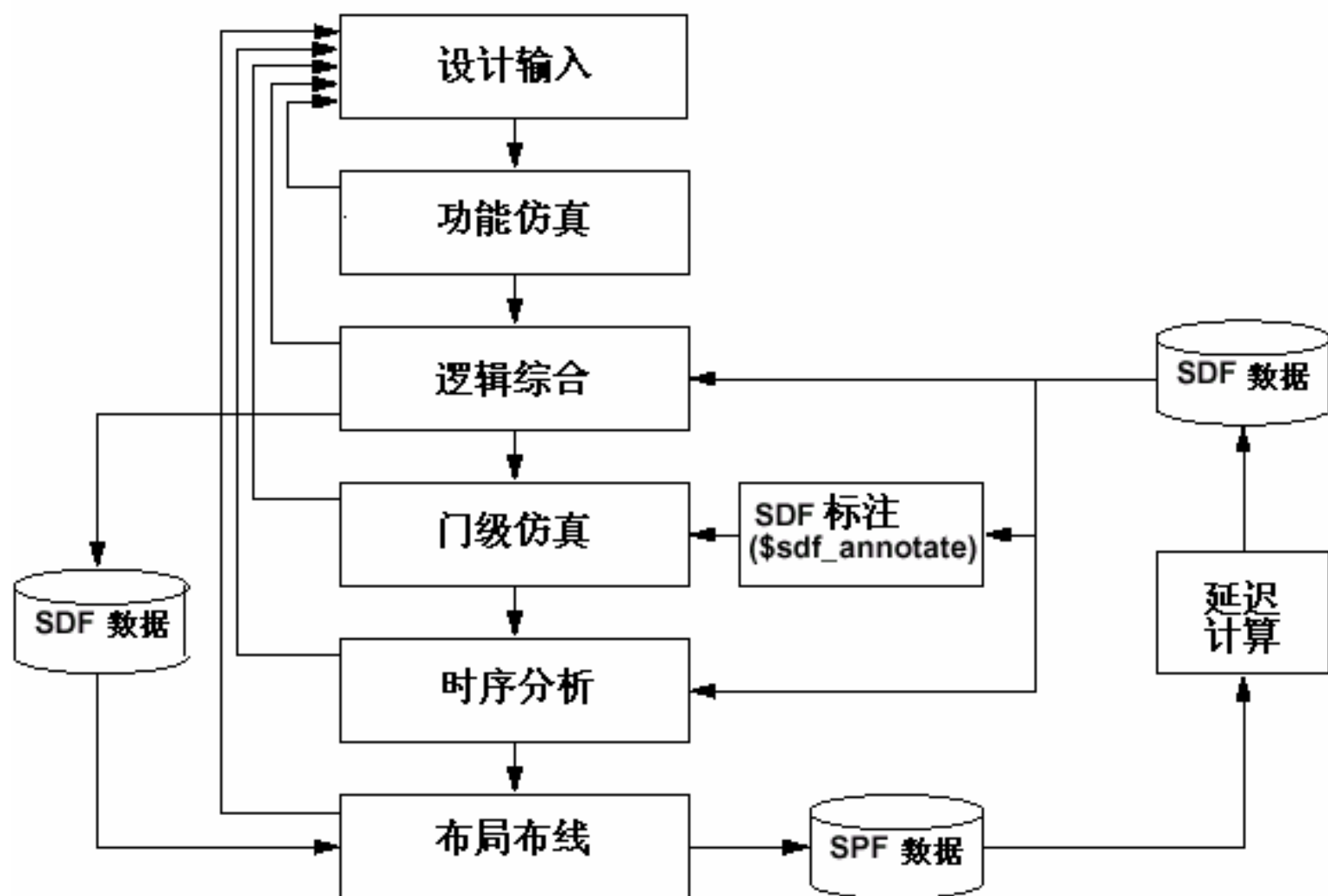
通常的Verilog元件库仅包含固有时序数据。

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：

- 粗略延迟，仅基于设计连线和层次
- 详细延迟，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**）。

延迟计算器

延时计算器主要有两类：

- 嵌入在工具中的延迟计算器
- 用户延迟计算器
 - 用户自定义
 - 开发商提供

延迟计算器可以产生**SDF**数据，或直接使用**PLI**标注时序数据。

延迟计算器可以自定义，但必须选择一个合适的延迟公式。

大多数**ASIC**生产商提供自己的生产工艺的延迟计算器。这些延迟计算器通常用**PLI**编写并直接在仿真时标注到设计中。但计算器也可以是独立的程序，产生的**SDF**由内嵌的延迟标注工具进行标注。

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：

- 模块通路延迟——条件的和无条件的
- 器件延迟
- 互连延迟
- 端口延迟
- 时序检查
- 通路和net时序约束

注意：在specity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 */
```

SDF文件配置信息

```
(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
的所有实例或某个实例

```
(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
```

SDF标注工具

用系统任务\$**sdf_annotate**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_annotate ("sdf_file", [module_instance,  
    "config_file", "log_file", "mtm_spec",  
    "scale_factors", "scale_type"]);
```

1. **sdf_file**: SDF文件的绝对或相对路径
2. **module_instance**: 标注范围。缺省为调用\$**sdf_annotate**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
```

执行SDF标注

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

总结

在本章中学习了：

- 延迟计算器
- 标准延迟格式**SDF**
- **SDF**数据标注

复习

问题：

1. 什么情况下要进行时序标注？
2. 延迟计算器通常需要哪些输入？
3. 在设计的什么地方可以调用`$sdf_annotate`系统任务？

解答：

1. 使用互连延迟仿真时进行时序标注，对同一个模块的不同实例使用不同的时序，这些时序是由元件物理特性计算出来的。
2. 任何延迟计算器都需要物理连接和层次信息、生产商元件技术库、元件环境信息以及用户的指导（如，想让它做什么？）。另外，计算器根据后端工具提取的简化的寄生参数，可以提供更好的延迟估算。
3. 通常，用户在设计的最顶层或`testbench`的`initial`块中调用`$sdf_annotate`系统任务，这样任务在时刻0时执行一次。也可以在交互式模式执行系统任务`$sdf_annotate`。
仿真器并不限制在哪里使用这个系统任务。

第22章 Coding Styles for Synthesis

主要内容:

1. if语句和case语句的编码风格
2. if语句和case语句中晚到达信号的处理
3. 逻辑块的编码风格
4. 高性能编码技术
5. 其它问题

if 语句

例1.1a 单个 *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;

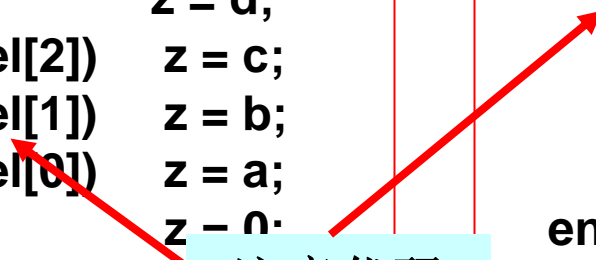
  always @(a or b or c or d or
sel)
    begin
      if (sel[3])      z = d;
      else if (sel[2]) z = c;
      else if (sel[1]) z = b;
      else if (sel[0]) z = a;
      else             z = 0;
    end
endmodule
```

例1.1b 多重 *if* 语句

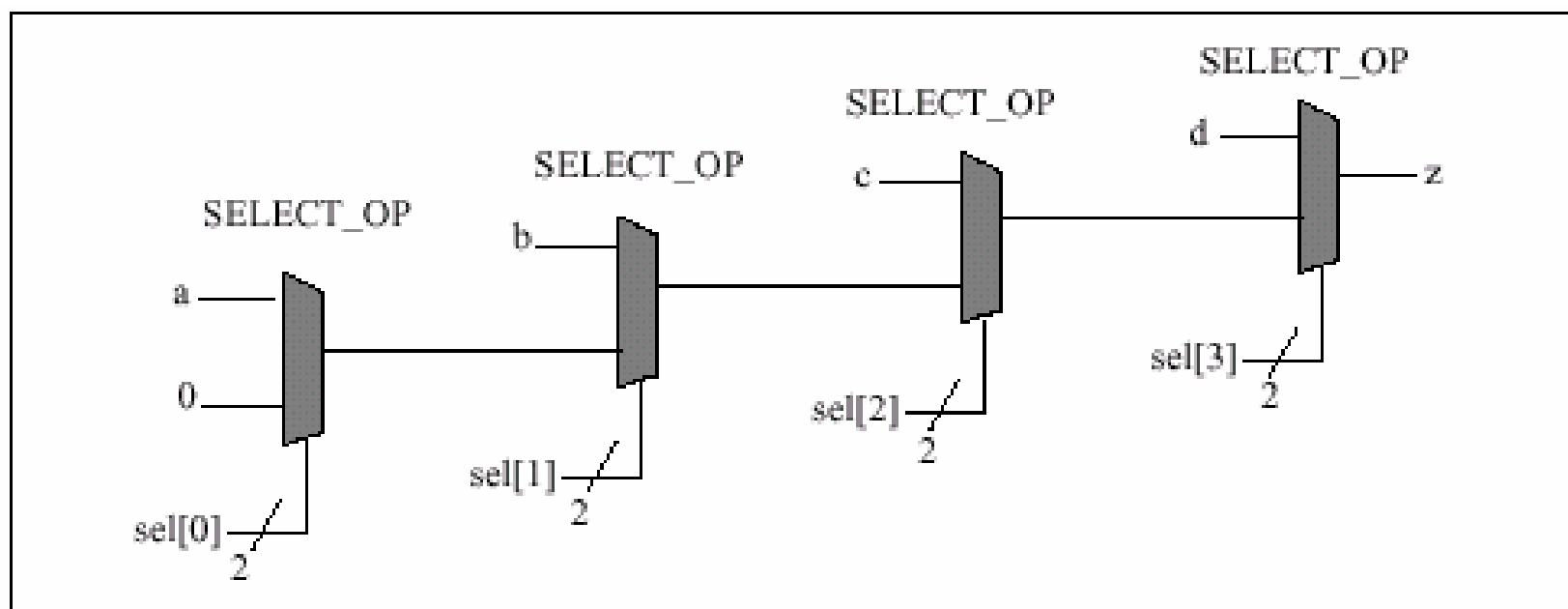
```
module mult_if(a, b, c, d, sel, z);
  input a, b, c, d;
  input [3:0] sel;
  output z;
  reg z;

  always @(a or b or c or d or
sel)
    begin
      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



if语句



case语句

例1.2 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 @(a or b or c or d or sel)  
    begin  
        case (sel)  
            4'b1xxx: z = d;  
            4'bx1xx: z = c;  
            4'bxx1x: z = b;  
            4'bxxx1: 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

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。

具有优先级的if结构

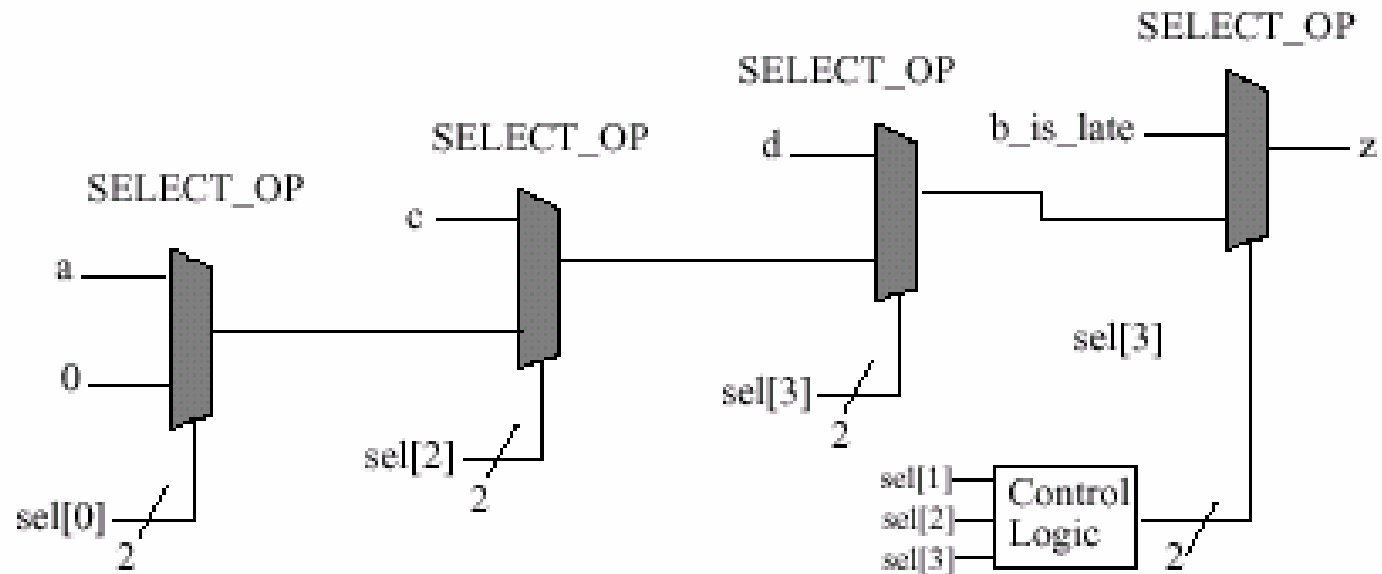
```
module mult_if_improved(a, b_is_late, c, d,
sel, z);
  input a, b_is_late, c, d;
  input [3:0] sel;
  output z;
  reg z, z1;
  always @(a or b_is_late or c or d or sel)
begin
  z1 = 0;
  if (sel[0]) z1 = a;
  if (sel[2]) z1 = c;
  if (sel[3]) z1 = d;
  if (sel[1] & ~(sel[2]|sel[3]))
    z = b_is_late;
  else
    z = z1;
end
```

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[2])  z = c;
    else if (sel[3])  z = d;
    else if (sel[0])  z = a;
    else              z = 0;
  end
```

晚到达的是数据信号



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)
            // late arriving signal in if condition
            Z = A[4];
        else if (C[5] == 1'b0) Z = A[5];
        else                    Z = A[6];
endmodule
```

晚到达的是控制信号

```
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)  
        // late arriving signal in if condition  
        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
```

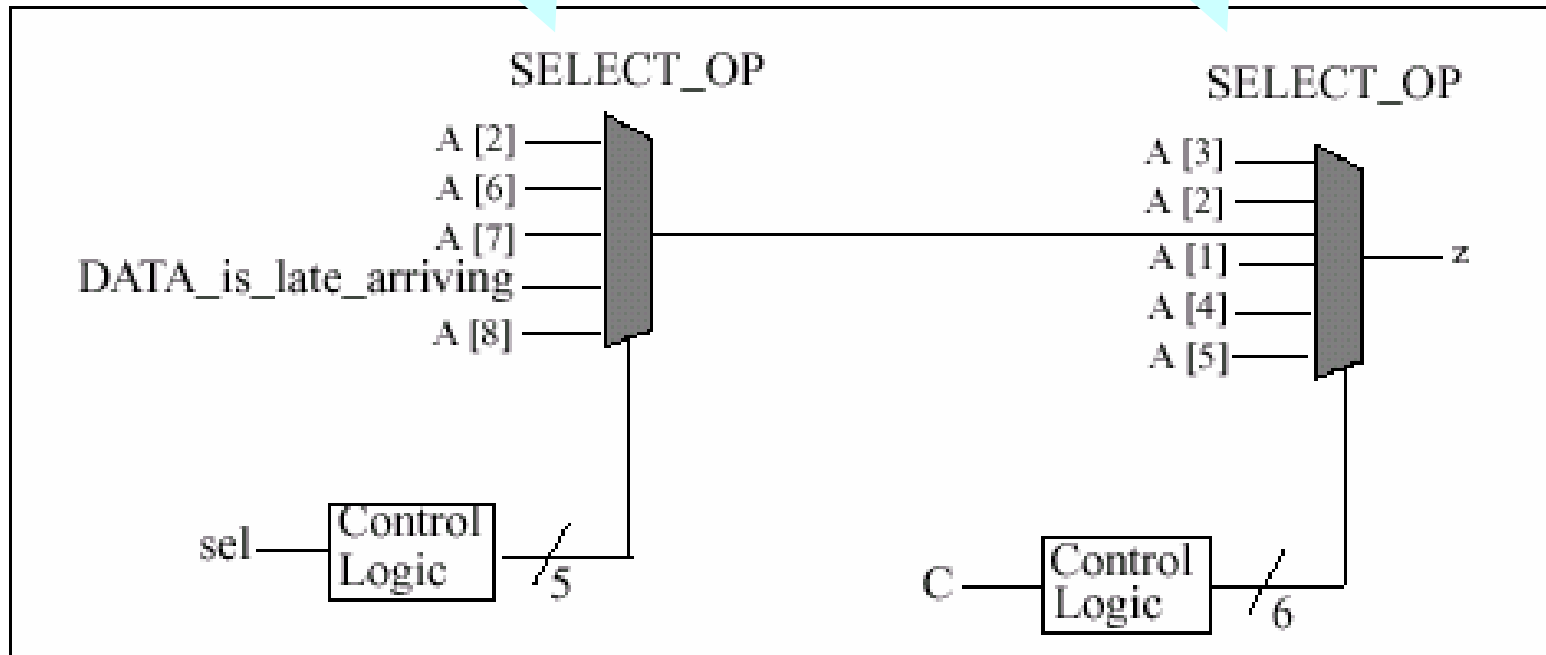

if-case嵌套语句

```
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
```

if-case嵌套语句

Case语句

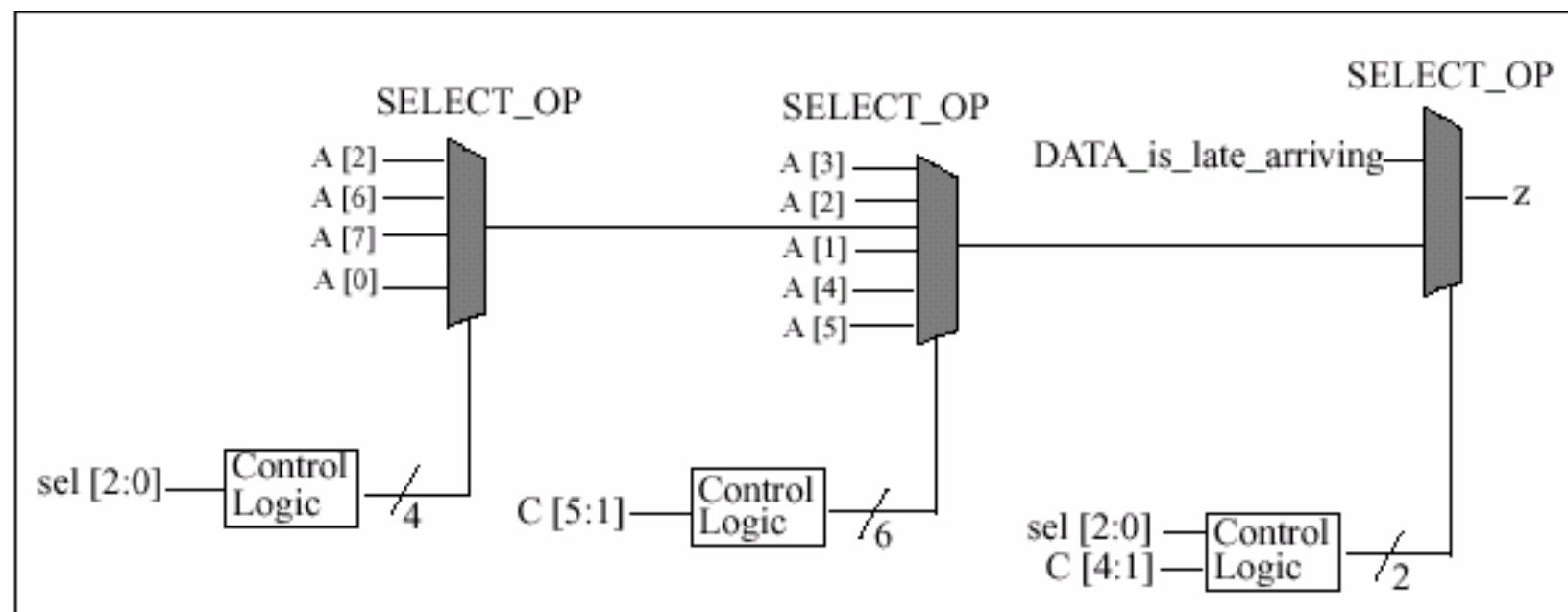
if语句



if-case嵌套语句—修改后

```
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
```

if-case嵌套语句—修改后



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)
```

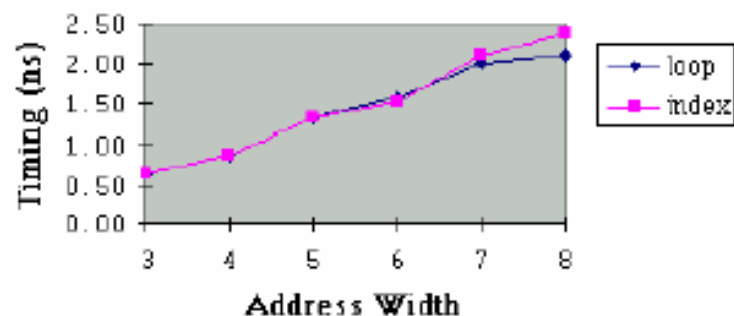
```
            out1[i] = (in1 == i);
```

```
    end
```

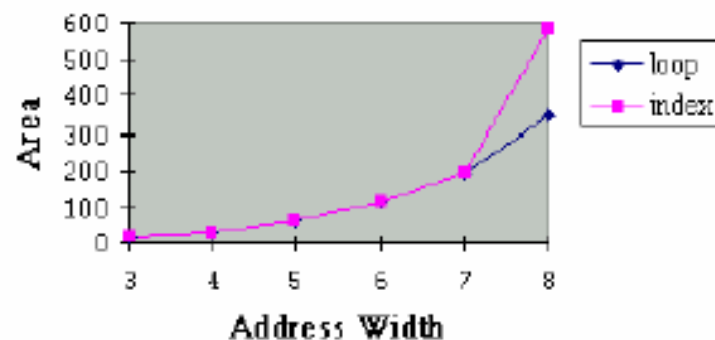
```
endmodule
```

译码器

Decoder Timing vs. Address Width



Decoder Area vs. Address Width



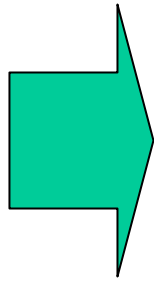
Decoder compile Time vs. Address Width



优先级编码器—高位优先

线性结构描述

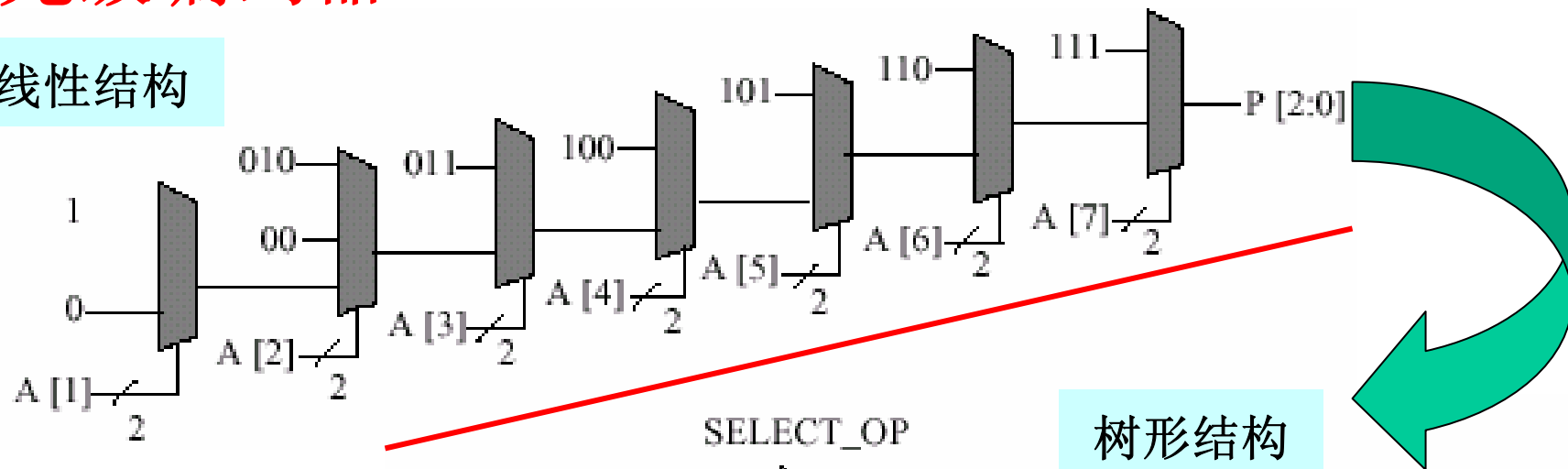
1???_???? : 111
01??_???? : 110
001?_???? : 101
0001_???? : 100
0000_1??? : 011
0000_01?? : 010
0000_001? : 001
0000_000? : 000



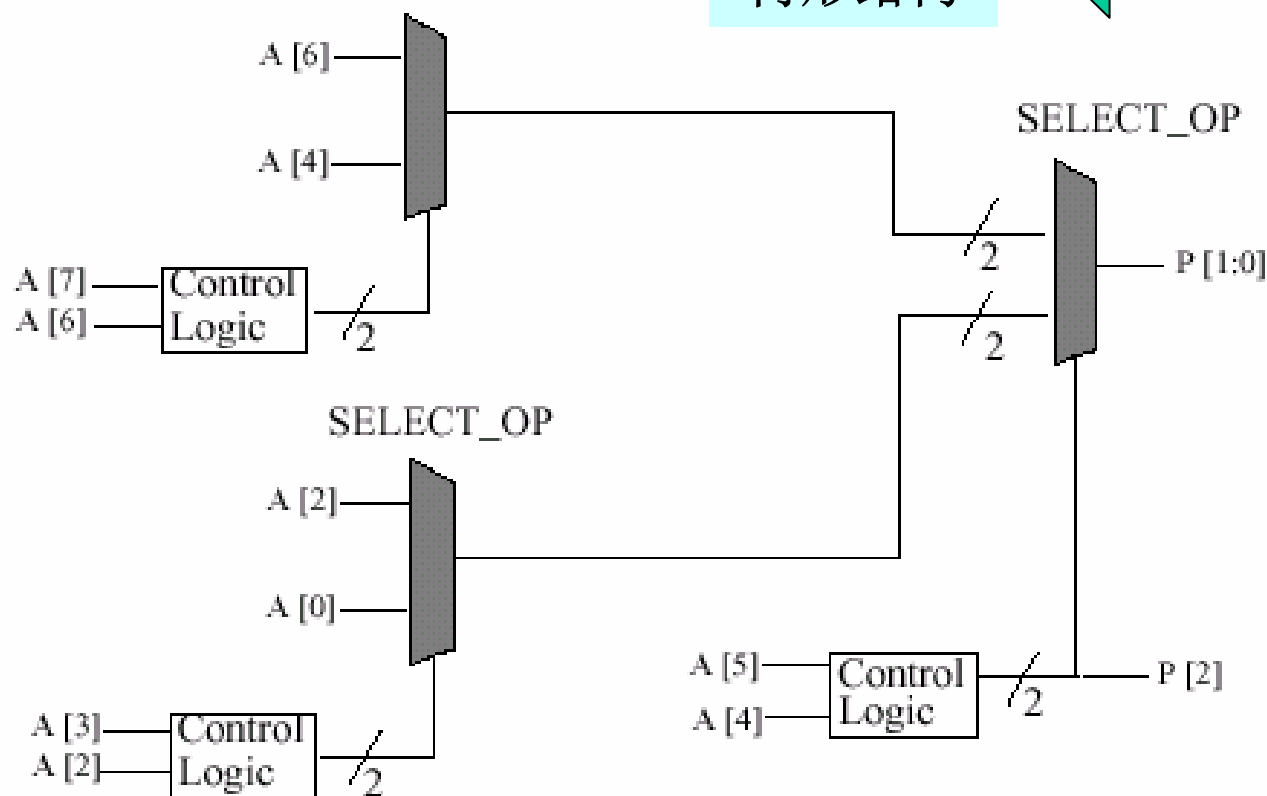
```
module priority_low_high (A, P);  
    parameter N = 8;  
    parameter log2N = 3;  
    input [N-1:0] A; //Input Vector  
    output [log2N-1:0] P; // High Priority Index  
    reg [log2N-1:0] P;  
  
    function [log2N-1:0] priority;  
        input [N-1:0] A;  
        integer I;  
        begin  
            priority = 3'b0;  
            for (I=0; I<N; I=I+1)  
                if (A[I])  
                    priority = I; // Override previous  
        end  
    index  
end  
  
endfunction  
always @(A)  
    P <= priority(A);  
endmodule
```


优先级编码器

线性结构



树形结构

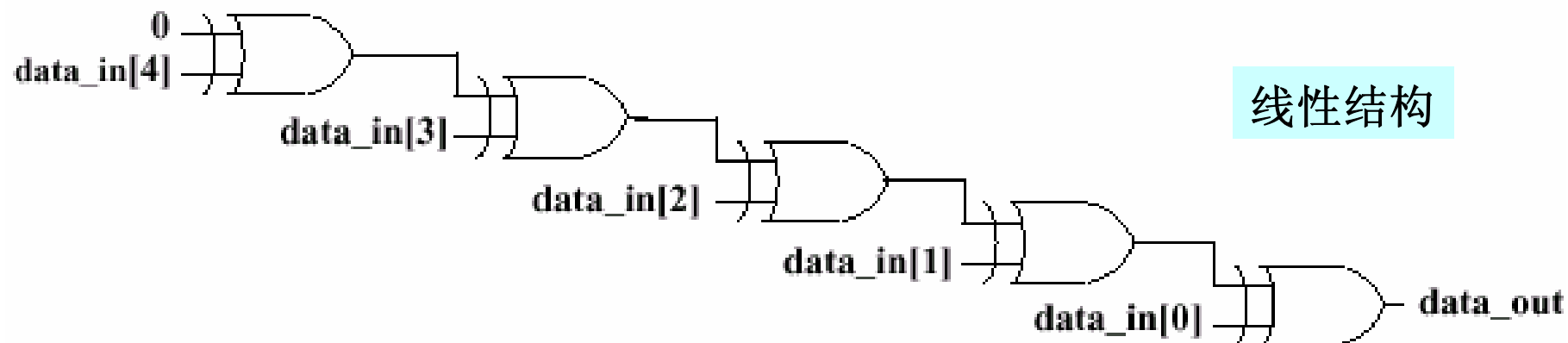


归约XOR

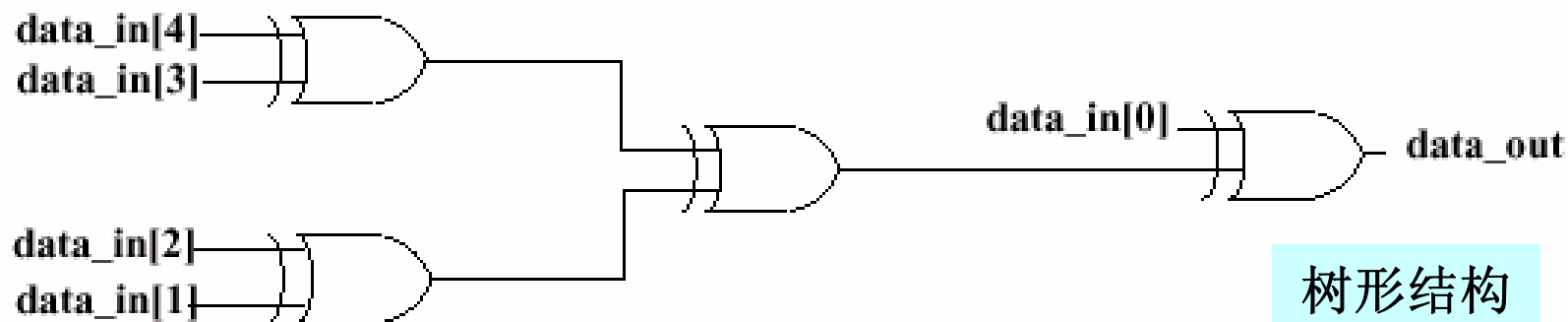
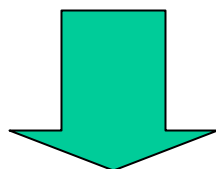
线性结构

```
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
```

归约XOR



线性结构



树形结构

归约XOR

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;
```

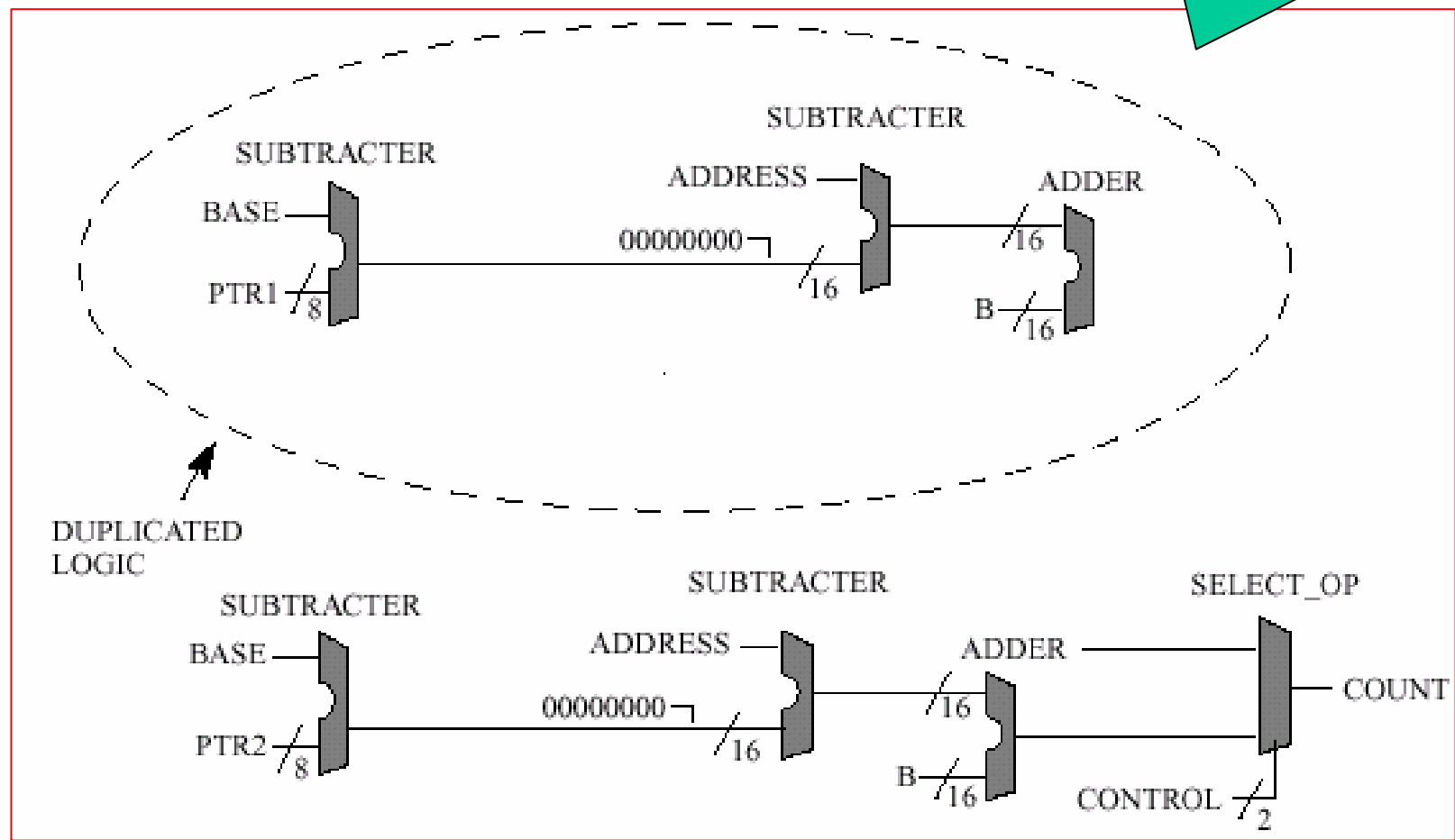
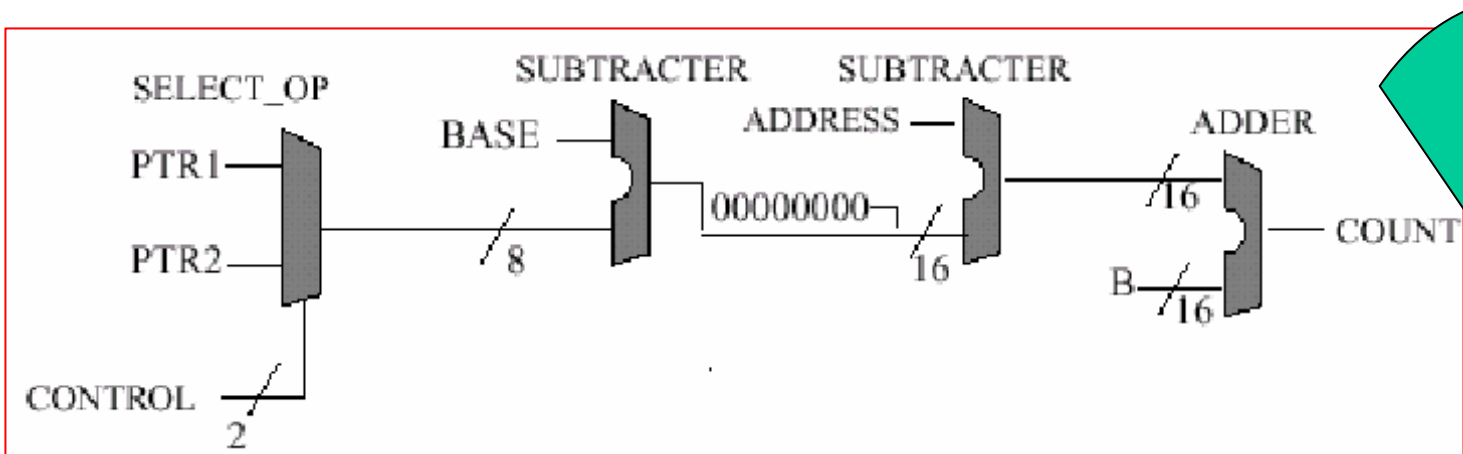
```
if (even(NUM))
    for (I=NUM-1; I>=0; I=I-2)
        begin
            result[J] = temp[I] ^ temp[I-1];
            J = J-1;
        end
    else    begin
        for (I=NUM-1; I>=1; I=I-2)    begin
            result[J] = temp[I] ^ temp[I-1];
            J = J-1;
        end
        result[0] = 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);
endendmodule
```

高性能编码技术

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 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 == 1'b1) ? PTR1 : PTR2;  
    assign OFFSET = BASE - PTR;  
    assign ADDR = ADDRESS - {8'h00, OFFSET};  
    assign COUNT = ADDR + B;  
endmodule
```



高性能编码技术

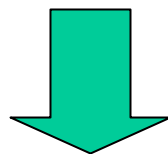
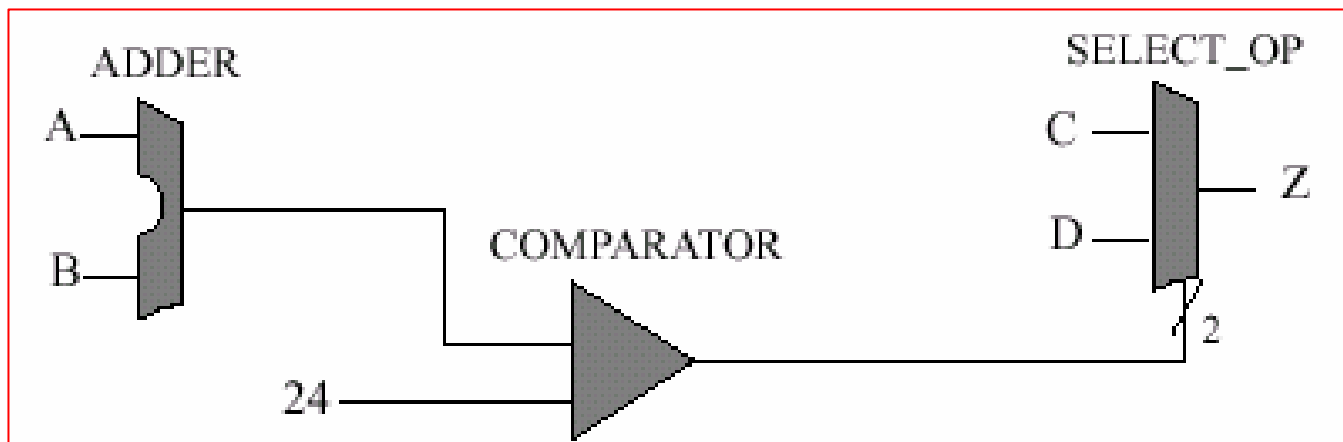
```
module PRECOMPUTED (ADDRESS, PTR1, PTR2, B, CONTROL,
COUNT);
    input [7:0] PTR1, PTR2;
    input [15:0] ADDRESS, B;
    input CONTROL;
    output [15:0] COUNT;
    parameter [7:0] BASE = 8'b10000000;
    wire [7:0] OFFSET1,OFFSET2;
    wire [15:0] ADDR1,ADDR2,COUNT1,COUNT2;
    assign OFFSET1 = 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
```

高性能编码技术

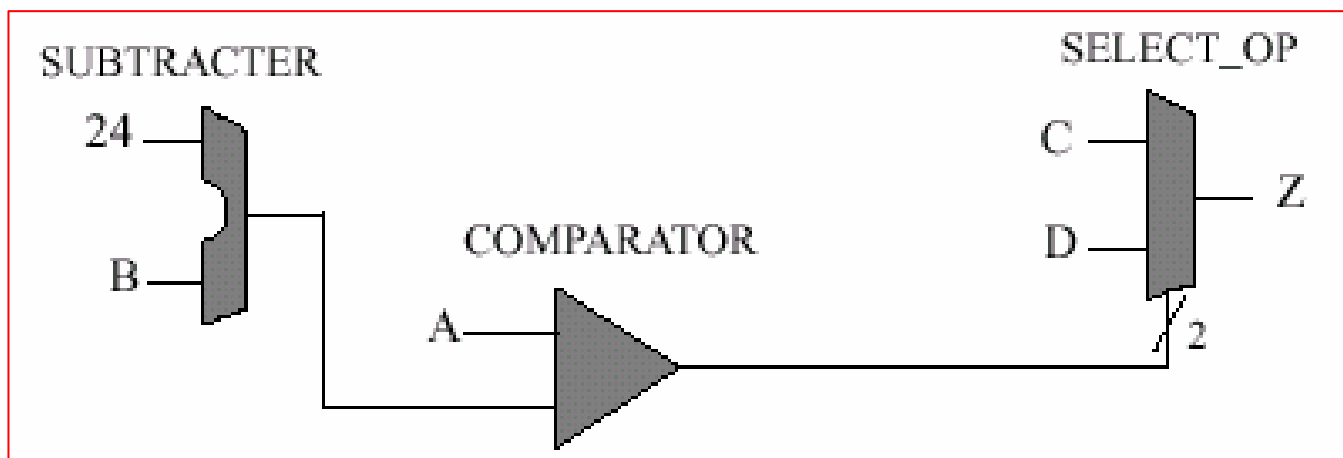
在下面的例子中，**if**语句的条件表达中包含有操作符。

```
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
```


高性能编码技术



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。



高性能编码技术

```
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)
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;
    end
```

敏感表要完整

不完整的敏感表将引起综合后网表的仿真结果与以前的不一致。

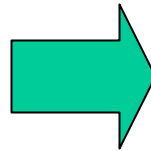
```
always @(d or clr)
```

```
  if (clr)
```

```
    q = 1'b0
```

```
  else if (e)
```

```
    q = d;
```



```
always @(d or clr or e)
```

```
  if (clr)
```

```
    q = 1'b0
```

```
  else if (e)
```

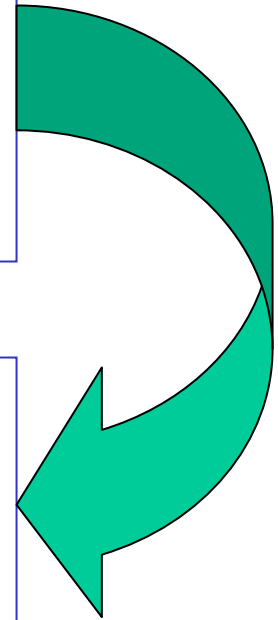
```
    q = d;
```

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( l =0; i<4; i=i+1) begin  
    sig1 = sig2; -- unchanging statement  
    data_out(l) = data_in(l);  
end
```

```
sig1 = sig2; -- unchanging statement  
for( l =0; i<4; i=i+1)  
    data_out(l) = data_in(l);
```

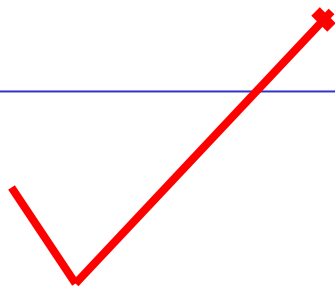


资源共享

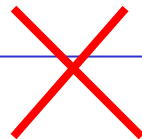
只有在同一个条件语句(**if**和**case**)不同的分支中的算术操作才会共享。

条件操作符 **?:** 中的算术操作不共享。

```
if (cond)
    z = a + b;
else
    z = c + d;
```



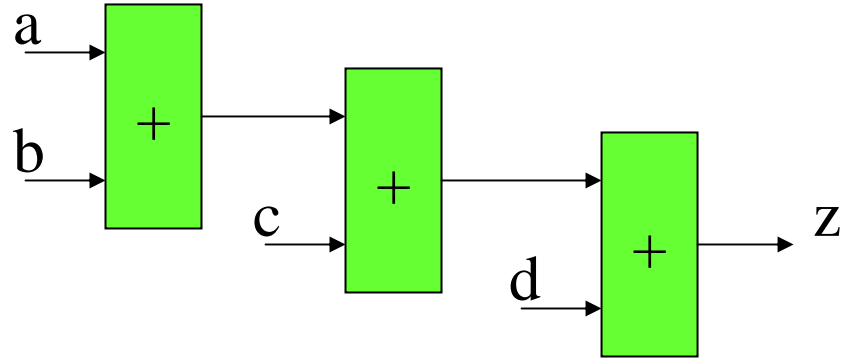
```
Z = (cond) ? (a + b) : (c + d);
```



括号的作用

利用括号分割逻辑。

$z = a + b + c + d;$



$Z = (a + b) + (c + d);$

