第 1 章

SVA 介 绍

——学习 SVA 语法

1.1 什么是断言

断言是设计的属性的描述。

- 如果一个在模拟中被检查的属性(property)不像我们期望的那样表现，那么这个断言失败。
- 如果一个被禁止在设计中出现的属性在模拟过程中发生，那么这个断言失败。

一系列的属性可以从设计的功能描述中推知，并且被转换成断言。这些断言能在功能的模拟中不断地被监视。使用形式验证技术，相同的断言能被重用来验证设计。断言，又被称为监视器或者检验器，已经被用作一种调试技术的方式，在设计验证流程中使用了很长时间。传统上，它们由过程语言，比如 Verilog，来实现。它们也能用 PLI 和 C/C++的程序来实现。下面的代码显示了相互断言条件检查的 Verilog 实现，其中信号 a 和信号 b 不能同时为高电平。如果这种情况发生，则显示这是一个错误信息。

```
`ifdef ma
if(a & b)
$display("Error:  Mutually asserted check

failed.\n");
```

```
`endif
```

这种监视器仅作为模拟的一部分而存在，因此只有当需要时才被纳入设计环境中。这可以通过允许 Verilog 代码条件编译的指令“`ifdef”来实现。

1.2 为什么使用 SystemVerilog 断言(SVA)

虽然 Verilog 可以很容易地用来实现一些检查，它仍有一些不足之处：

(1) Verilog 是一种过程语言，因此并不能很好地控制时序。

(2) Verilog 是一种冗长的语言，随着断言的数量增加，维护代码将变得很困难。

(3) 语言的过程性这一本质使得测试同一时间段内发生的并行事件相当困难。在一些情况下，一个 Verilog 的检验器甚至可能无法捕捉到所有被触发的事件。

(4) Verilog 语言没有提供内嵌的机制来提供功能覆盖的数据。用户必须自己实现这部分代码。

SVA 是一种描述性语言，可以完美地描述时序相关的状况。语言的描述性本质提供了对时间卓越的控制。语言本身非常精确且易于维护。SVA 也提供了若干个内嵌函数来测试特定的设计情况，并且提供了一些构造来自动收集功能覆盖数据。

例子 1.1 显示了分别用 Verilog 和 SVA 实现的检验器。这个检验器验证当信号 a 在当前时钟周期为高电平时，下面 1~3 个时钟周期内，信号 b 应该变为高电平。

例子 1.1 分别用 Verilog 和 SVA 实现的断言实例

```
// Sample Verilog checker

always @(posedge a)
begin
```

```
repeat (1) @(posedge clk);
  fork: a_to_b

    begin
    @(posedge b)
    $display
   ("SUCCESS: b arrived in time\n", $time);
    disable a_to_b;
    end

    begin
    repeat (3) @(posedge clk);
    $display
("ERROR: b did not arrive in time\n", $time);
    disable a_to_b;
    end

  join
end

// SVA Checker

a_to_b_chk:
assert property
@(posedge clk) $rose(a) |-> ##[1: 3] $rose(b));
```

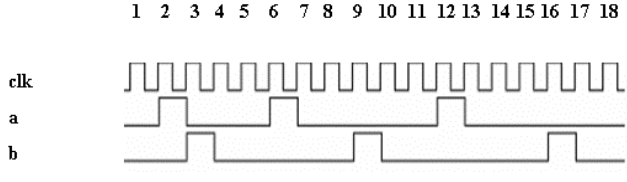

图 1-1 断言实例的波形

例子 1.1 很清楚地显示了 SVA 的优势。本章将讨论 SVA 语法。检验器代表着一种非常简单的协议。使用 SVA 实现只需要一行代码，而相同的协议描述用 Verilog 则需要好几行。此外，失败和成功的条件必须在 Verilog 里额外地被定义，而 SVA 中断言失败会

自动显示错误信息。例子模拟的结果如下：

```
SUCCESS:  b arrived in time 127
vtosva.a_to_b_chk:
started at 125s succeeded at 175s

SUCCESS:  b arrived in time 427
vtosva.a_to_b_chk:
started t 325s succeeded at 475s

ERROR:  b did not arrive in time 775
vtosva.a_to_b_chk:
started at 625s failed at 775s
     Offending '$rose(b)'
```

1.3 SystemVerilog 的调度

SystemVerilog 语言被定义成一种基于事件的执行模式。在每个时隙(time slot)，许多事件按照安排的顺序发生。这个事件的列表依照标准定义的算法执行。依照这个算法，模拟器可以防止任何在设计和测试平台互动中的不一致。断言的评估和执行包括以下三个阶段：

预备(Preponed) 在这个阶段，采样断言变量，而且信号(net)或变量(variable)的状态不能改变。这样确保在时隙开始的时候采样到最稳定的值。

观察(Observed) 在这个阶段，对所有的属性表达式求值。

响应(Reactive) 在这个阶段，调度评估属性成功或失败的代码。

图 1-2 显示了一个简化了的 SystemVerilog 事件进程安排流程表。要彻底地理解 SystemVerilog 的进程安排算法，请参考<<SystemVerilog 3.1a LRM>>[1]。

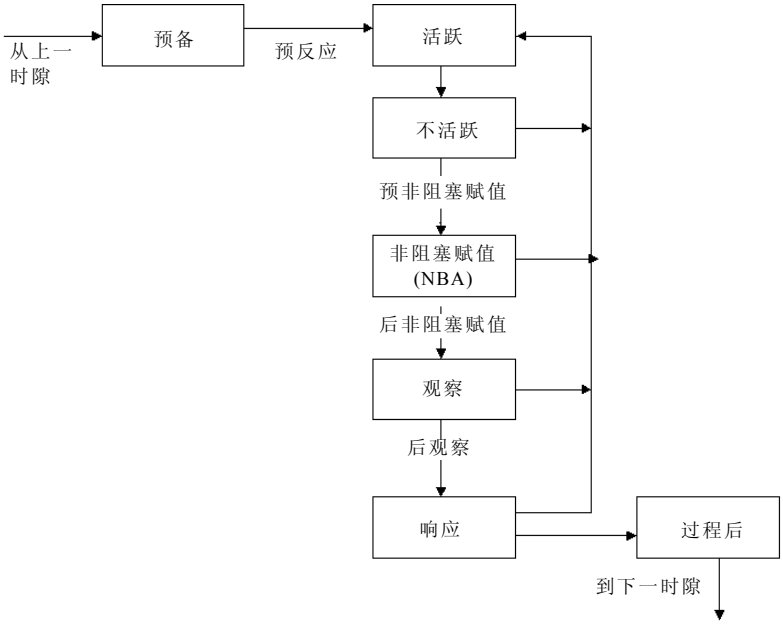

图 1-2 SV 事件调度流程简图

1.4 SVA 术语

SystemVerilog 语言中定义了两种断言：并发断言和即时断言。

1.4.1 并发断言

- 基于时钟周期。
- 在时钟边缘根据调用的变量的采样值计算测试表达式。
- 变量的采样在预备阶段完成，而表达式的计算在调度器的观察阶段完成。
- 可以被放到过程块(procedural block)、模块(module)、接口(interface)，或者一个程序(program)的定义中。
- 可以在静态(形式的)验证和动态验证(模拟)工具中使用。

一个并发断言的例子如下：

```
a_cc: assert property (@(posedge clk) not (a && b));
```

图 1-3 显示了并发断言 a_cc 的结果。所有的成功显示成向上的箭头，所有的失败显示成向下的箭头。这个例子的核心内容是属性在每一个时钟的上升沿都被检验，不论信号 a 和信号 b 是否有值的变化。

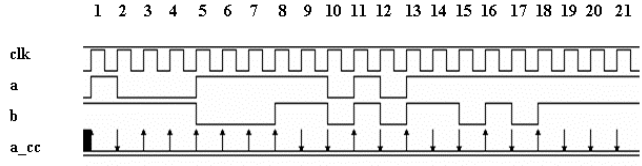

图 1-3 并发断言例子的波形

1.4.2 即时断言

- 基于模拟事件的语义。
- 测试表达式的求值就像在过程块中的其他 Verilog 的表达式一样。它们本质不是时序相关的，而且立即被求值。
- 必须放在过程块的定义中。
- 只能用于动态模拟。

一个即时断言的例子如下：

```
always_comb
begin
    a_ia:  assert (a && b);
end
```

即时断言 a_ia 被写成一个过程块的一部分，它遵循和信号 a、b 相同的事件调度。当信号 a 或者信号 b 发生变化时，always 块被执行。区别即时断言和并发断言的关键词是“**property**”。图 1-4 显示了即时断言 a_ia 的结果：

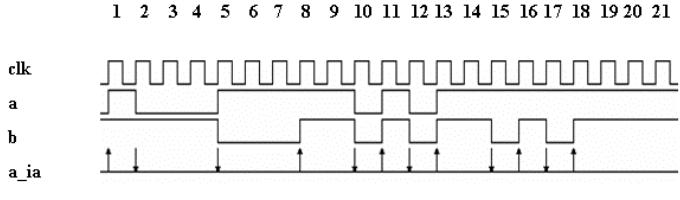

图 1-4 即时断言例子的波形

1.5 建立 SVA 块

在任何设计模型中，功能总是由多个逻辑事件的组合来表示的。这些事件可以是简单的同一个时钟边缘被求值的布尔表达式，或者是经过几个时钟周期的求值的事件。SVA 用关键词“sequence”来表示这些事件。序列(sequence)的基本语法是：

```
sequence name_of_sequence;
   <test expression>;
endsequence
```

许多序列可以逻辑或者有序地组合起来生成更复杂的序列。SVA 提供了一个关键词“property”来表示这些复杂的有序行为。属性(property)的基本语法是：

```
property name_of_property;
   <test expression>; or
   <complex sequence expressions>;
endproperty
```

属性是在模拟过程中被验证的单元。它必须在模拟过程中被断言来发挥作用。SVA 提供了关键词“assert”来检查属性。断言(assert)的基本语法是：

```
assertion_name： assert property (property_name);
```

建立 SVA 检验器的步骤如图 1-5 所示：

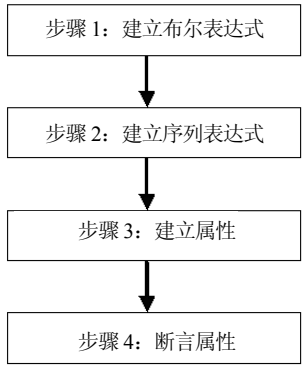

图 1-5 SVA 建立块图示

1.6 一个简单的序列

序列 s1 检查信号“a”在每个时钟上升沿都为高电平。如果信号“a”在任何一个时钟上升沿不为高电平，断言将失败。注意，这相当于“a == 1'b1”。

```
sequence s1;
   @(posedge clk) a;
endsequence
```

图 1-6 显示了信号“a”和序列在模拟中对这个信号响应的波形。信号“a”在第七个时钟上升沿变为 0。这一变化在第八个时钟周期被采样到。因为并行断言使用进程安排中预备(“prepond”)阶段采样到的值，在第七个时钟周期，序列 s1 采样到的信号“a”的最稳定的值是 1。因此序列成功。在第八个时钟周期，信号“a”被采样的值为 0，因此序列失败。一个向上的箭头表示一次成功，一个向下的箭头表示一次失败。表 1-1 总结了信号“a”每个时钟周期的采样值，直到第十五个时钟周期。

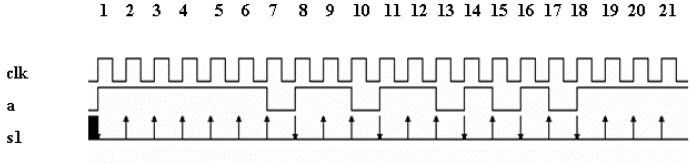

图 1-6 简单序列 s1 的波形

表 1-1 序列 s1 的真值表

时 钟 数	信号“a”的采样值
1	0
2	1
3	1
4	1
5	1
6	1
7	1

(续表)

时 钟 数	信号“a”的采样值
8	0
9	1
10	1
11	0
12	1
13	1
14	0
15	1

1.7 边沿定义的序列

序列 s1 使用的是信号的逻辑值。SVA 也内嵌了边缘表达式，以便用户监视信号值从一个时钟周期到另一时钟周期的跳变。这使得用户能检查边沿敏感的信号。三种这样有用的内嵌函数如下：

$rose(boolean expression or signal_name)

- 当信号/表达式的最低位变成 1 时返回真。

$fell(boolean expression or signal_name)

- 当信号/表达式的最低位变成 0 时返回真。

$stable(boolean expression or signal_name)

- 当表达式不发生变化时返回真。

序列 s2 检查信号“a”在每一个时钟上升沿都跳变成 1。如果跳变没有发生，断言失败。

```
sequence s2;
    @(posedge clk) $rose(a);
endsequence
```

图 1-7 显示序列 s2 响应信号“a”跳变的情况。标记 1 显示了序列 s2 的第一个成功。在时钟周期 1，信号“a”的值从 0 变到 1。

在这个时钟周期，信号“a”在序列中的采样值是 0。在时钟周期 1 之前，信号“a”没有被赋值，因此值被认定为“x”。值从 x 到 0 的转化不是上升沿，因此序列失败。在时钟周期 2，信号“a”在序列中的采样值是 1。值从 0 到 1 的转化是上升沿，因此序列 2 在时钟周期 2 成功。在时钟周期 9 的标记 2 显示了另一个成功。表 1-2 总结了信号“a”前 9 个时钟周期的转化以及序列如何采样和更新值。

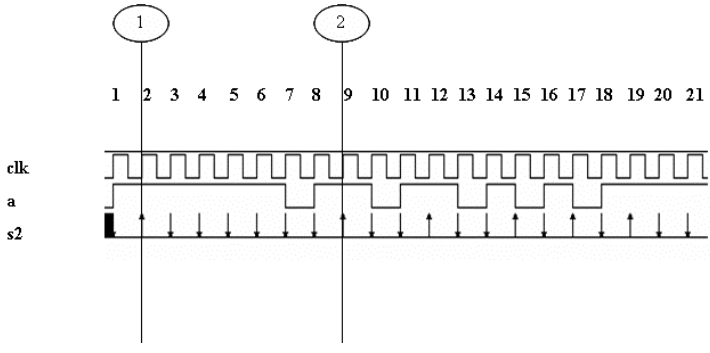

图 1-7　边沿定义的简单序列的波形

表 1-2　序列s2 的真值表

时钟数	信号“a”从上一个时钟周期得来的采样值	信号“a”在这个时钟周期的采样值	序列 s2 的状态
1	X	0	失败
2	0	1	成功
3	1	1	失败
4	1	1	失败
5	1	1	失败
6	1	1	失败
7	1	1	失败
8	1	0	失败
9	0	1	成功

1.8 逻辑关系的序列

序列 s3 检查每一个时钟上升沿，信号“a”或信号“b”是高电平。如果两个信号都是低电平，断言失败。

```
sequence s3;
@(posedge clk)  a || b;
endsequence
```

图 1-8 显示序列 S3 如何根据信号“a”和“b”做出反应。标记 1 显示了时钟周期 12，信号“a”和“b”的采样值都是 0，因此序列失败。同理，在标记 2 所在的时钟周期 17，序列也失败。在所有其他时钟周期，信号“a”和信号“b”至少有一个其值为 1，因此在这些时钟周期，序列都成功。

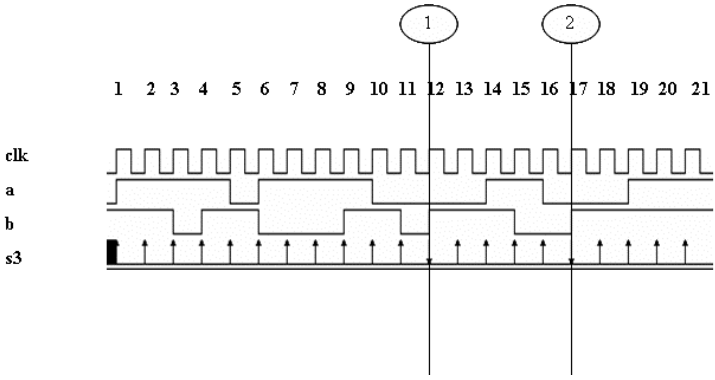

图 1-8 序列s3 的波形

1.9 序列表达式

通过在序列定义中定义形参，相同的序列能被重用到设计中具有相似行为的信号上。例如，我们可以定义下面这个序列

```
sequence s3_lib (a, b);
a || b;
endsequence
```

通用的序列 s3_lib 能重用在任何两个信号上。例如，我们有两个信号“req1”和“req2”，它们中至少一个信号应该在时钟周期的上升沿为 1，我们可以使用下列的序列。

```
sequence s3_lib_inst1;
   s3_lib (req1, req2);
endsequence
```

一些在设计中常见的通常的属性可以被开发成一个库以便于重用。比如，one-hot 状态机检查，等效性检查等都适合放在这样的检验器库中。

1.10 时序关系的序列

简单的布尔表达式在每个时钟边缘都会被检查。换句话说，它们只是简单的组合逻辑检查。很多时候，我们关心的是检查需要几个时钟周期才能完成的事件。也就是所谓的“时序检查”。在 SVA 中，时钟周期延迟用“##”来表示。例如，##3 表示 3 个时钟周期。

序列 s4 检查信号“a”在一个给定的时钟上升沿为高电平，如果信号“a”不是高电平，序列失败。如果信号“a”在任何一个给定的时钟上升沿为高电平，信号“b”应该在两个时钟周期后为高电平。如果信号“b”在两个时钟周期后不为 1，断言失败。注意，序列以信号“a”在时钟上升沿为高电平开始。

```
sequence s4;
  @(posedge clk)  a ##2 b;
endsequence
```

图 1-9 显示了序列 s4 在模拟过程中的响应。表 1-3 总结了信号“a”和信号“b”在每个时钟周期的采样值。

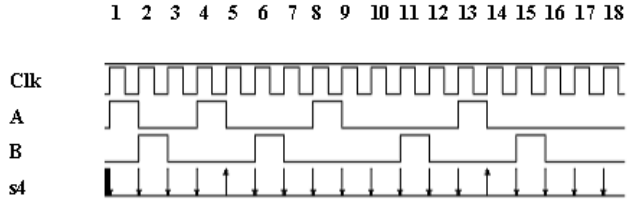

图 1-9 序列s4 的波形

与前面小节中的例子不同的是，序列 s4 的开始时间和结束时间不同。如果信号“a”在任何时钟周期不为高电平，序列在同一个时钟周期开始并失败。如果信号“a”是高电平，序列开始。在两个时钟周期后，如果信号“b”是高电平，序列成功(第 5 和第 14 时钟周期)。另一方面，如果在两个时钟周期后，信号“b”不是高电平，序列失败。应注意的是，在图中，成功的序列总是标注在序列开始的位置。

表 1-3 序列s4 的真值表

时钟数	“a”的采样值	“b”的采样值	S4 有效开始	S4 状态
1	0	0	否	失败
2	1	0	是	失败 (开始于 2，结束于 4)
3	0	1	否	失败
4	0	0	否	失败
5	1	0	是	成功(开始于 5，结束于 7)
6	0	0	否	失败
7	0	1	否	失败
8	0	0	否	失败
9	1	0	是	失败 (开始于 9，结束于 11)
10	0	0	否	失败
11	0	0	否	失败
12	0	1	否	失败
13	0	0	否	失败
14	1	0	是	成功(开始于 14，结束于 16)
15	0	0	否	失败
16	0	1	否	失败
17	0	0	否	失败

1.11 SVA 中的时钟定义

一个序列或者属性在模拟过程中本身并不能起什么作用。它们必须像下面的例子那样被断言才能发挥作用。

```
sequence s5;
  @(posedge clk)  a ##2 b;
endsequence

property p5;
  s5;
endproperty

a5 :  assert property(p5);
```

注意，序列 s5 中指定了时钟。这是一种把检查和时钟关联起来的方法，但是还有其他的方法。在序列、属性，甚至一个断言的语句中都可以定义时钟。下面的代码显示了在属性 p5a 的定义中指定时钟。

```
sequence s5a;
 a ##2 b;
endsequence

property p5a;
  @(posedge clk)  s5a;
endproperty

a5a :  assert property(p5a);
```

通常情况下，在属性(property)的定义中指定时钟，并保持序列(sequence)独立于时钟是一种好的编码风格。这可以提高基本序列定义的可重用性。

断言一个序列并不一定需要定义一个独立的属性。因为断言语句调用属性，在断言的语句中可以直接调用被检查的表达式，如下面的断言 a5b 所示。

```
sequence s5b;
 a ##2 b;
endsequence

a5b : assert property(@(posedge clk) s5b);
```

当我们在断言的陈述中要调用已经定义了时钟的序列，就不能再次在断言语句中定义时钟。下面的断言 a5c 就显示了这种错误的编程风格。

```
a5c : assert property(@(posedge clk) p5a); // Not
allowed
```

1.12 禁止属性

在之前的所有例子中，属性检查的都是正确的条件。属性也可以被禁止发生。换句话说，我们期望属性永远为假。当属性为真时，断言失败。

序列 s6 检查当信号“a”在给定的时钟上升沿为高电平，那么两个时钟周期以后，信号“b”不允许是高电平。关键词“**not**”用来表示属性应该永远不为真。

```
sequence s6;
@(posedge clk)  a ##2 b;
endsequence

property p6;
 not s6;
endproperty

a6 : assert property(p6);
```

图 1-10 显示了断言 a6 如何在模拟过程中响应。我们注意到检验器在标记 1 和 2 显示的两个位置(时钟 5 和 14)失败。在这两个时钟周期，发生了被禁止的序列，断言因此失败。

另一方面，在信号“a”有效的两个位置(时钟 2 和时钟 9)检验器成功。因为从这两个时钟周期开始检查，两个时钟周期以后

信号“b”不为高，因此检验器成功。在其他时钟周期中，信号“a”都不为高，因此检验器都自动成功。表 1-4 总结了信号“a”和信号“b”在每个时钟周期的采样值。

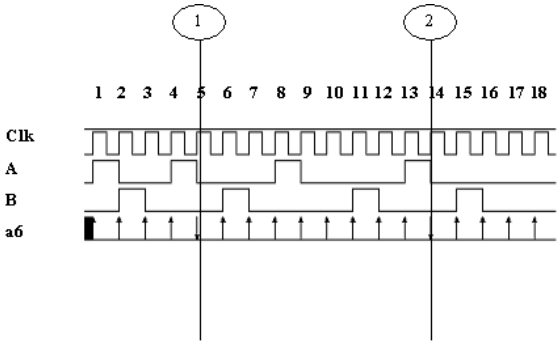

图 1-10　带禁止属性的SVA检验器的波形

表 1-4　属性p6 的真值表

时钟数	“a”的采样值	“b”的采样值	s6 有效开始	a6 状态
1	0	0	是	成功 (同一时钟周期)
2	1	0	是	成功 (开始于 2，结束于 4)
3	0	1	是	成功 (同一时钟周期)
4	0	0	是	成功 (同一时钟周期)
5	1	0	是	失败 (开始于 5，结束于 7)
6	0	0	是	成功 (同一时钟周期)
7	0	1	是	成功 (同一时钟周期)
8	0	0	是	成功 (同一时钟周期)
9	1	0	是	成功 (开始于 9，结束于 11)
10	0	0	是	成功 (同一时钟周期)
11	0	0	是	成功 (同一时钟周期)
12	0	1	是	成功 (同一时钟周期)
13	0	0	是	成功 (同一时钟周期)
14	1	0	是	失败 (开始于 14，结束于 16)
15	0	0	是	成功 (同一时钟周期)
16	0	1	是	成功 (同一时钟周期)
17	0	0	是	成功 (同一时钟周期)

1.13 一个简单的执行块

SystemVerilog 语言被定义成每当一个断言检查失败，模拟器在默认情况下都会打印出一条错误信息。模拟器不需要对成功的断言打印任何东西。读者同样也可以使用断言陈述中的“执行块”(action block)来打印自定义的成功或失败信息。执行块的基本语法如下所示。

```
assertion_name :
   assert property(property_name)
            <success message> ;
         else
            <fail message>;
```

下面显示的检验器 a7 在执行块中使用了简单的显示语句来打印成功和失败信息。

```
property p7;
    @(posedge clk)  a ##2 b;
endproperty

a7 : assert property(p7)
     $display("Property p7 succeeded\n");
     else
     $display("Property p7 failed\n");
```

执行块不仅仅局限于显示成功和失败。它可以有其他的应用，例如：控制模拟环境和收集功能覆盖数据。这些主题将在第 2 章详细讨论。

1.14 蕴含操作符

属性 p7 有下列特别之处：

(1) 属性在每一个时钟上升沿寻找序列的有效开始。在这种情况下，它在每个时钟上升沿检查信号“a”是否为高。

(2) 如果信号“a”在给定的任何时钟上升沿不为高，检验器将产生一个错误信息。这并不是一个有效的错误信息，因为我们并不关心只检查信号“a”的电平。这个错误只表明我们在这个时钟周期没有得到检验器的有效起始点。虽然这些错误是良性的，它们会在一段时间内产生大量的错误信息，因为检查在每个时钟周期都被执行。为了避免这些错误，某种约束技术需要被定义来在检查的起始点不有效时忽略这次检查。

SVA 提供了一项技术来实现这个目的。这项技术叫作“蕴含”(**Implication**)。

蕴含等效于一个 if-then 结构。蕴含的左边叫作“先行算子”(**antecedent**)，右边叫作“后续算子”(**consequent**)。先行算子是约束条件。当先行算子成功时，后续算子才会被计算。如果先行算子不成功，那么整个属性就默认地被认为成功。这叫作“空成功”(vacuous success)。蕴含结构只能被用在属性定义中，不能在序列中使用。

蕴含可以分为两类：交叠蕴含(Overlapped implication)和非交叠蕴含(Non-overlapped implication)。

1.14.1 交叠蕴含

交叠蕴含用符号“|->”表示。如果先行算子匹配，在同一个时钟周期计算后续算子表达式。下面用一个简单的例子解释。属性 p8 检查信号“a”在给定的时钟上升沿是否为高电平，如果 a 为高，信号“b”在相同的时钟边沿也必须为高。

```
property p8;
  @(posedge clk)  a |-> b;
endproperty

a8 :  assert property(p8);
```

图 1-11 显示了断言 a8 在模拟中的响应。表 1-5 总结了信号“a”和信号“b”的采样值和断言的状态。表中一共显示了三种结果。当信号“a”检测为有效的高电平，而且信号“b”在同一个时钟

沿也检测为高，这是一个真正的成功。若信号“a”不为高，断言默认地自动成功，则称为空成功。相应的，失败指的是信号“a”检测为高且在同一个时钟沿信号“b”未能检测为有效的高电平。

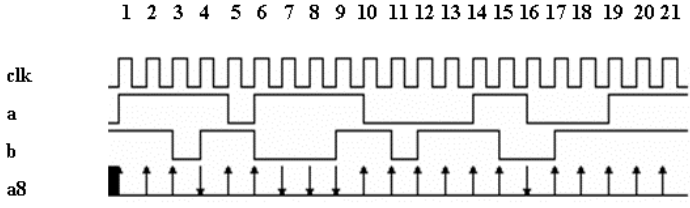

图 1-11 属性p8 的波形

表 1-5 属性p8 的真值表

时 钟 数	“a”的采样值	“b”的采样值	a8 的状态
1	0	1	空成功
2	1	1	真正的成功
3	1	1	真正的成功
4	1	0	失败
5	1	1	真正的成功
6	0	1	空成功
7	1	0	失败
8	1	0	失败
9	1	0	失败

1.14.2 非交叠蕴含

非交叠蕴含用符号“|=>”表示。如果先行算子匹配，那么在下一个时钟周期计算后续算子表达式。后续算子表达式的计算总是有一个时钟周期的延迟。下面以属性 p9 举个简单的例子。该属性检查信号“a”在给定的时钟上升沿是否为高，如果为高，信号“b”必须在下一个时钟边沿为高。

```
property p9;
  @(posedge clk)  a |=> b;
endproperty

a9 :  assert property(p9);
```

图 1-12 显示了断言 a9 在模拟中的响应。表 1-6 总结了信号“a”和信号“b”的采样值以及断言的状态。应注意的是，断言在当前时钟周期开始，在下一个时钟周期成功的情况才是真正的成功。相应的，如果属性有一个有效的开始(信号“a”为高)，且信号“b”在下一个时钟周期不为高，属性失败。

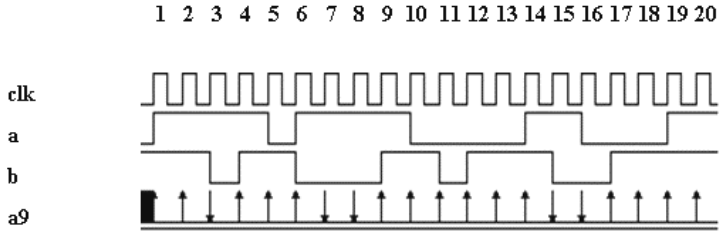

图 1-12 属性p9 的波形

表 1-6 属性p9 的真值表

时钟数	“a”的采样值	“b”的采样值	a9 的状态
1	0	1	空成功
2	1	1	真正的成功 (开始于 2，结束于 3)
3	1	1	败 (开始于 3，结束于 4)
4	1	0	真正的成功(开始于 4，结束于 5)
5	1	1	真正的成功 (开始于 5，结束于 6)
6	0	1	空成功
7	1	0	失败 (开始于 7，结束于 8)
8	1	0	失败(开始于 8，结束于 9)
9	1	0	真正的成功(开始于 9，结束于 10)

1.14.3 后续算子带固定延迟的蕴含

属性 p10 检查如果信号“a” 在给定时钟上升沿为高，在两个时钟周期后信号“b”应该为高。类似的检查在前面已经用不使用蕴含的方式介绍过了。使用蕴含使得所有误报的错误都被消除。只有属性有效开始(信号“a”为高)时，才进行后续算子的检查(信

号“a”)。图 1-13 显示了属性 p10 的一个模拟的例子。表 1-7 总结了属性 p10 中信号的采样值。

```
property p10;
  @(posedge clk)  a |-> ##2 b;
endproperty

a10 :  assert property(p10);
```

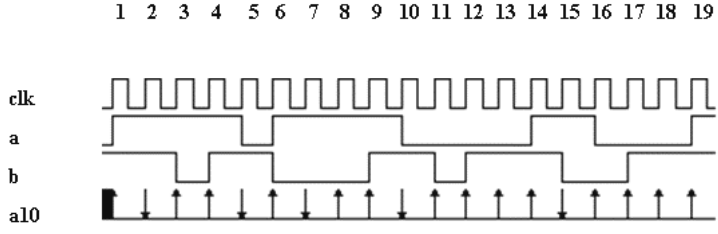

图 1-13 属性p10 的波形

表 1-7 属性p10 的真值表

时钟数	“a”的采样值	“b”的采样值	a10 的采样值
1	0	1	空成功
2	1	1	失败 (开始于 2，结束于 4)
3	1	1	成功 (开始于 3，结束于 5)
4	1	0	成功 (开始于 4，结束于 6)
5	1	1	失败 (开始于 5，结束于 7)
6	0	1	空成功
7	1	0	失败 (开始于 7，结束于 9)
8	1	0	成功 (开始于 8，结束于 10)
9	1	0	成功 (开始于 9，结束于 11)

1.14.4 使用序列作为先行算子的蕴含

属性 p10 在先行算子的位置使用的是信号。先行算子同样可以使用序列的定义。在这种情况下，仅当先行算子中的序列成功时，才计算后续算子中的序列或者布尔表达式。在任何给定的时

钟周期，序列 s11a 检查如果信号“a”和信号“b”都为高，一个时钟周期之后信号“c”应该为高。序列 s11b 检查当前时钟上升沿的两个时钟周期后，信号“d”应为低。最终的属性检查如果序列 s11a 成功，那么序列 s11b 被检查。如果没有监测到有效的序列 s11a，那么序列 s11b 将不被检查，属性检查得到一次空成功。

```
sequence s11a;
    @(posedge clk) (a && b) ##1 c;
endsequence

sequence s11b;
    @(posedge clk) ##2 !d;
endsequence

property p11;
    s11a |-> s11b;
endpeoperty
```

图 1-14 显示了断言 a11 在模拟中的表现。标记 1s 和 1e 表明了一个成功的属性检查的起始和结束。标记 2s 和 2e 标出了一个失败的起始和结束。在时钟周期 11，信号“a”和信号“b”都为高。这表明 2 个时钟周期以后，即时钟周期 14，信号“d”应该为低。但是在例子中的波形上信号“d”为高电平，因此属性失败。

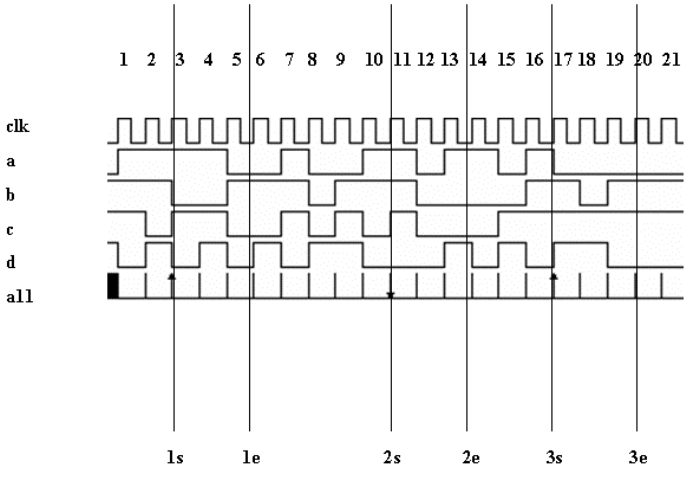

图 1-14 属性p11 的波形

图中所有的空成功都用简单的竖线表示。标记 3s 和 3e 显示了一个成功的属性检查的起始和结束。表达式“a && b”在时钟周期 17 为真，在一个时钟周期后，信号“c”像预期的一样为高。因此在时钟周期 18，序列 s11a 成功。正如被期望的那样，接着信号“d”在两个时钟周期后为低。因此，属性在时钟周期 20 成功。

1.15 SVA 检验器的时序窗口

到目前为止，带延迟的例子使用的都是固定的正延迟。在下面几个例子中，我们将讨论几种不同的描述延迟的方法。

属性 p12 检查布尔表达式“a && b”在任何给定的时钟上升沿为真。如果表达式为真，那么在接下去的 1~3 周期内，信号“c”应该至少在一个时钟周期为高。SVA 允许使用时序窗口来匹配后续算子。时序窗口表达式左手边的值必须小于右手边的值。左手边的值可以是 0。如果它是 0，表示后续算子必须在先行算子成功的那个时钟边沿开始计算。

```
property p12;
  @(posedge clk)  (a && b) |->  ##[1:3] c;
endproperty

a12 :  assert property(p12);
```

图 1-15 显示了属性 p12 在模拟中的响应。每声明一个时序窗口，就会在每个时钟沿上触发多个线程来检查所有可能的成功。p12 实际上以下面三个线程展开。

```
(a && b) |->  ##1 c 或
(a && b) |->  ##2 c 或
(a && b) |->  ##3 c
```

属性有三个机会成功。所有三个线程具有相同的起始点，但是一旦第一个成功的线程将使整个属性成功。应当注意，在任何时钟上升沿只能有一个有效的开始，但是可以有多个有效的结束。这是因为每个有效的起始可以有三个机会成功。

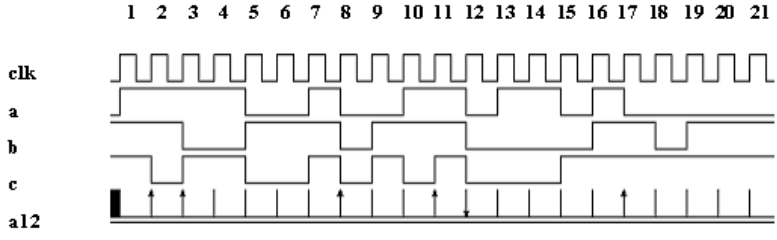

图 1-15　p12 的波形

表 1-8 总结了属性计算过程中所有相关信号的采样值。在任何给定的时钟上升沿，如果信号“a”和信号“b”不全为高，那么属性得到一个空成功。另一方面，如果信号“a”和信号“b”都为高，那么属性就有了一个有效的开始。如果信号“c”在之后的 1~3 个时钟周期内都没被检测到高电平，属性失败。

表 1-8　属性p12 的真值表

时钟数	“a”的采样值	“b”的采样值	“c”的采样值	p12 的有效开始	a12 的状态
1	0	1	1	否	空成功
2	1	1	1	是	真正的成功(开始于 2，结束于 4)
3	1	1	0	是	真正的成功(开始于 3，结束于 4)
4	1	0	1	否	空成功
5	1	0	1	否	空成功
6	0	1	0	否	空成功
7	0	1	0	否	空成功
8	1	1	1	是	真正的成功(开始于 8，结束于 10)
9	0	0	0	否	空成功
10	0	1	1	否	空成功
11	1	1	0	是	真正的成功(开始于 11，结束于 12)

(续表)

时钟数	“a”的采样值	“b”的采样值	“c”的采样值	p12 的有效开始	a12 的状态
12	1	1	1	是	失败 (开始于 12，结束于 15)
13	0	0	0	否	空成功
14	1	0	0	否	空成功
15	1	0	0	否	空成功
16	0	0	1	否	空成功
17	1	1	1	是	真正的成功(开始于 17，结束于 18)

注意，属性在时钟周期 2 和 3 都有有效的开始。这两个有效的开始同时在时钟周期 4 成功。时钟周期 2 开始的检查在两个时钟周期后检测到信号“c”为高。而时钟周期 2 开始的检查在一个时钟周期后检测到信号“c”为高。这两个都是有效情况，因此它们都成功了。在时钟周期 12 同样有一个有效的开始。属性在时钟周期 13，14 和 15 都检测信号“c”是否为高。由于信号“c”在这所有三个可能的时钟周期始终为低，检测失败。

1.15.1 重叠的时序窗口

属性 p13 与属性 p12 相似。两者最大的区别是 p13 的后续算子在先行算子成功的同一个时钟沿开始计算。

```
Property p13;
  @(posedge clk)  (a && b) |->  ##[0: 2] c;
endproperty

a13 :  assert property(p13);
```

图 1-16 显示了 p13 在模拟中的响应。与属性 p12 最大的区别在于一个成功的开始发生在时钟周期 12。这个成功是因为检查发生了重叠。信号“c”的值在先行算子成功的同一个时钟沿被检测为高。

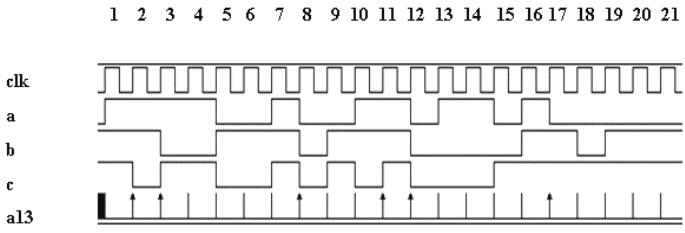

图 1-16　属性p13 的波形

1.15.2　无限的时序窗口

在时序窗口的窗口上限可以用符号“$”定义，这表明时序没有上限。这叫作“可能性”(eventuality)运算符。检验器不停地检查表达式是否成功直到模拟结束。因为会对模拟的性能产生巨大的负面影响，所以这不是编写 SVA 的一个高效的方式。最好总是使用有限的时序窗口上限。

属性 p14 在任何给定的时钟上升沿检查信号“a”是否为高。如果为高，那么信号“b”从下一个时钟周期往后最终将为高，而信号“c”在信号“b”为高的时钟周期开始往后最终将为高。

```
property p14;
  @(posedge clk)  a |->  ##[1: $] b ##[0: $] c;
endproperty

a14 :  assert property(p14);
```

图 1-17 显示了属性 p14 在模拟中的响应。表 1-9 总结了断言 a14 和相关信号的采样值。值得注意的是，真正的成功可能在任意个时钟周期后结束。如果一个有效的开始发生，而信号“b”或信号“c”在模拟结束前始终不为高，这些检查被报告为“未完成检验”(incomplete check)。因为信号“b”和信号“c”可以重叠地满足检验，整个检查有可能在一个时钟周期内结束。时钟周期 17 显示了这样一种情况，当信号“a”在时钟周期 17 被检测为高，且信号“b”和信号“c”在时钟周期 18 都被检测为高。

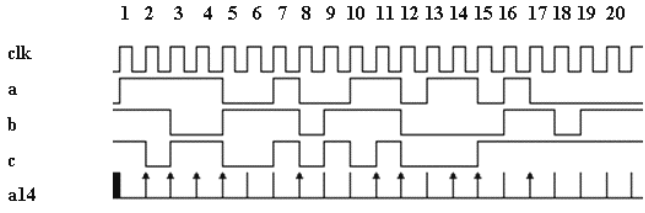

图 1-17 属性p14 的波形

表 1-9 p14 的真值表

时钟数	“a”的采样值	“b”的采样值	“c”的采样值	p14 有效开始	a14 的状态
1	0	1	1	否	空成功
2	1	1	1	是	真正的成功（开始于2，结束于 4)
3	1	1	0	是	真正的成功（开始于3，结束于 8)
4	1	0	1	是	真正的成功（开始于4，结束于 8)
5	1	0	1	是	真正的成功（开始于5，结束于 8)
6	0	1	0	否	空成功
7	0	1	0	否	空成功
8	1	1	1	是	真正的成功（开始于8，结束于 10)
9	0	0	0	否	空成功
10	0	1	1	否	空成功
11	1	1	0	是	真正的成功（开始于11，结束于 12)
12	1	1	1	是	真正的成功（开始于12，结束于 17)
13	0	0	0	否	空成功
14	1	0	0	是	真正的成功（开始于14，结束于 17)
15	1	0	0	是	真正的成功（开始于15，结束于 17)
16	0	0	1	否	空成功
17	1	1	1	是	真正的成功（开始于17，结束于 18)

1.16 “ended”结构

到目前为止，定义的序列都只是用了简单的连接(concatenation)的机制。换句话说，就是将多个序列以序列的起始点作为同步点，来组合成时间上连续的检查。SVA 还提供了另一种使用序列的结束点作为同步点的连接机制。这种机制通过给序列名字追加上关键词“ended”来表示。例如，s.ended 表示序列的结束点。关键词“ended”保存了一个布尔值，值的真假取决于序列是否在特定的时钟边沿匹配检验。这个 s.ended 的布尔值只有在相同时钟周期有效。

序列 s15a 和 s15b 是两个需要多个时钟周期来完成的简单序列，属性 p15a 检查序列 s15a 和序列 s15b 满足两者间隔一个时钟周期的延迟分别匹配检验。属性 p15b 检查相同的协议，但是使用了关键词“ended”。在这种情况下，两个序列在结束点同步。由于使用了结束点，两个序列间加上了两个时钟周期的延迟，来保证断言检验的协议与 p15a 相同。

```
sequence s15a;
    @(posedge clk) a ##1 b;
endsequence

sequence s15b;
    @(posedge clk) c ##1 d;
endsequence

property p15a;
    s15a |=> s15b;
endproperty

property p15b;
    s15a.ended |-> ##2 s15b.ended;
endproperty

a15a: assert property(p15a);
a15b: assert property(p15b);
```

图 1-18 显示了属性 p15a 和 p15b 在模拟中的响应。表 1-10 总结了断言 a15a 和 a15b 的状态。断言 a15a 的第一个真正的成功发生在时钟周期 2。当信号“a”被检测为高，检验在时钟周期 2 被激活。当信号“d”在时钟周期 5 检测为高时，检验完成。断言 a15b 的第一次真正成功出现在在时钟周期 3。在时钟周期 3，当序列 s15a 成功，即信号“b”被检测为高时检验被激活。接着在时钟周期 5 当序列 s15b 成功，或者说信号“d”被检测为高时，检验完成。

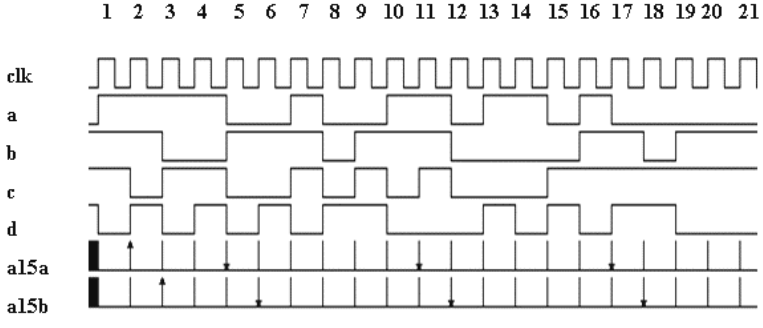

图 1-18 使用“ended”的SVA检验器的波形

断言 a15a 的第一次失败发生在时钟周期 5。当信号“a”在给定的时钟上升沿为高，且一个时钟周期以后(时钟周期 6)信号“b”紧接着为高时，检测到一个有效的起始点。这样就会检查后续算子，因为信号“c”在下一个时钟周期不为高，检验失败于时钟周期 7。

相应地，断言 a15b 的第一个失败出现在时钟周期 6。当序列 s15a 在时钟周期 6 成功结束时，检测到一个有效的起始点。接着检验后续算子在时钟周期 8 是否得到一个有效的结束点。因为信号“c”在时钟周期 7 不像期望的那样为高，序列的结束点的值为假，导致检验在时钟周期 8 为假。

上述例子中，我们用了两种不同的方法来实现统一个检验。第一种方法基于序列的起始点来同步序列。第二种方法基于序列的结束点来同步序列。

表 1-10 使用“ended”的SVA检验器的真值表

时钟数	“a”的采样值	“b”的采样值	“c”的采样值	“d”的采样值	a15a的状态	A15b 状态
1	0	1	1	1	空成功	空成功
2	1	1	1	0	真正的成功(开始于 2，结束于 5)	空成功
3	1	1	0	1	空成功	真正的成功(开始于 3，结束于 5)
4	1	0	1	0	空成功	空成功
5	1	0	1	1	失败 (开始于 5，结束于 7)	空成功
6	0	1	0	0	空成功	失败 (开始于 6，结束于 8)
7	0	1	0	1	空成功	空成功
8	1	1	1	0	空成功	空成功
9	0	0	0	1	空成功	空成功
10	0	1	1	1	空成功	空成功
11	1	1	0	0	失败 (开始于 11，结束于 13)	空成功
12	1	1	1	0	空成功	失败 (开始于 12，结束于 14)
13	0	0	0	0	空成功	空成功
14	1	0	0	1	空成功	空成功
15	1	0	0	0	空成功	空成功
16	0	0	1	1	空成功	空成功
17	1	1	1	0	失败(开始于 17，结束于 20)	空成功

1.17 使用参数的 SVA 检验器

SVA 允许像 Verilog 那样在检验器中使用参数(parameter)。这为创建可重用的属性提供了很大的灵活性。比如，两个信号间的延迟信息可以在检验器中用参数表示，那么这种检验器就可以在设计只有时序关系不同的情况中重用。例子 1.2 显示了一个带延迟默认值参数的检验器。如果这个检验器在设计中被调用，它使用一个时钟周期作为延迟默认值。如果在实例化时重设检验器中延迟参数值，那么同一个检验器就可以被重用。在例子 1.2 中，模块“top”有两个“generic_chk”的实例。实例 i1 将延迟参数改写为 2 个时钟周期，而实例 i2 使用默认的 1 个时钟周期。

例 1.2 使用参数的 SVA 检验器的例子

```
module generic_chk (input logic a, b, clk);

parameter delay = 1;

property p16;
    @(posedge clk) a |-> ##delay b;
endproperty

a16: assert property(p16);

endmodule

// call checker from the top level module

module top(…..);
logic clk, a, b, c, d;
.
.
generic_chk #(.delay(2)) i1 (a, b, clk);
generic_chk i2 (c, d, clk);
```

```
.
.
endmodule
```

图 1-19 显示了两个检验器实例 i1 和 i2 在模拟过程中对信号变化的响应。

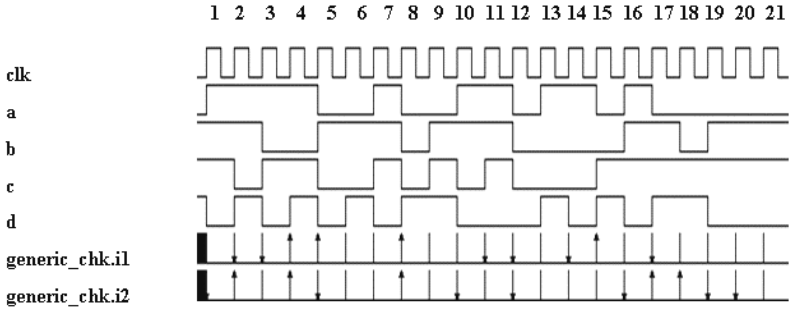

图 1-19 带参数的SVA检验器的波形

1.18 使用选择运算符的 SVA 检验器

SVA 允许在序列和属性中使用逻辑运算符。属性 p17 检查如果信号“c”为高，那么信号“d”的值与信号“a”的相等。如果信号“c”不为高，那么信号“d”的值与信号“b”的相等。这是一个组合的检验，在每个时钟上升沿被执行。

```
property p17;
@(posedge clk) c ? d == a: d == b;
endproperty

a17: assert property(p17);
```

图 1-20 显示了属性 p17 在模拟中的响应。表 1-11 总结了断言 a17 的状态和涉及的信号的采样值。在时钟周期 1，信号“c”被检测为高，因此检验期望信号“d”和信号“a”有相等的值。但是信号“d”被检测为高，而信号“a”为低，所以检验失败。

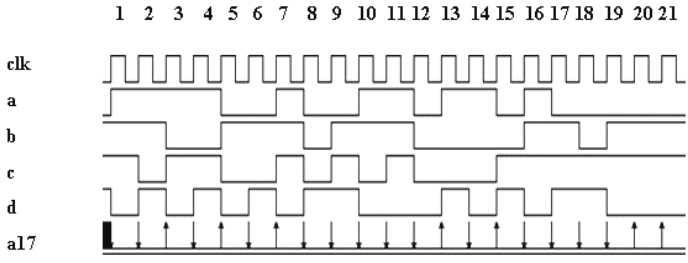

图 1-20 使用选择运算符的SVA检验器的波形

表 1-11 使用选择运算器的SVA检验器的真值表

时钟数	“a”的采样值	“b”的采样值	“c”的采样值	“d”的采样值	a17的状态
1	0	1	1	1	失败
2	1	1	1	0	失败
3	1	1	0	1	成功
4	1	0	1	0	失败
5	1	0	1	1	成功
6	0	1	0	0	失败
7	0	1	0	1	成功
8	1	1	1	0	失败
9	0	0	0	1	失败
10	0	1	1	1	失败
11	1	1	0	0	失败
12	1	1	1	0	失败
13	0	0	0	0	成功
14	1	0	0	1	失败
15	1	0	0	0	成功
16	0	0	1	1	失败
17	1	1	1	0	失败

1.19 使用 true 表达式的 SVA 检验器

使用 true 表达式，可以在时间上延长 SVA 检验器。这代表一种忽略的状态，它使得序列延长了一个时钟周期。这可以用来实现同时监视多个属性且需要同时成功的复杂协议。

序列 s18a 检查一个简单的条件。序列 s18a_ext 检查相同的条件，但是序列的成功被往后移了一个时钟周期。当这个序列被用在一个属性的现行算子时，它会造成一些差异。两个序列的结束点不同，因此开始检查后续算子的时钟周期也不一样。

属性 p18 检查先行算子中的 s18a.end，如果成功，两个时钟周期后，检查 s18b.end 是否成功。属性 p18_ext 检查 s18a_ext 在先行算子中是否成功。这个成功与 s18a.ended 的成功相同，但是早了一个时钟周期。因此属性 p18_ext 的后续算子需要在一个时钟周期而不是像 p18 中定义的两个时钟周期后成功。属性 p18 和 p18_ext 检查相同的情况，但是他们在先行算子中的成功点却不同。

```
`define true 1

sequence s18a;
    @(posedge clk) a ##1 b;
endsequence

sequence s18a_ext;
    @(posedge clk) a ##1 b ##1 `true;
endsequence

sequence s18b;
    @(posedge clk) c ##1 d;
endsequence

property p18
    @(posdge clk) s18a.ended |-> ##2 s18b.ended;
endproperty

property p18_ext
    @(posdge clk) s18a_ext.ended |=> s18b.ended;
```

```
endproperty

a18: assert property(p18);
a18_ext: assert property(p18_ext);
```

图 1-21 显示了属性 p18 和 p18_ext 在模拟中的响应。可以清楚地看到与断言 a18 比较，断言 a18_ext 的起始点被推迟了一个时钟周期。

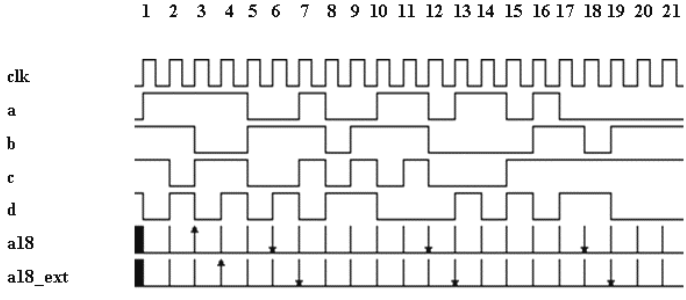

图 1-21 使用true表达式的SVA检验器的波形

1.20 “$past”构造

SVA 提供了一个内嵌的系统任务“$past”，它可以得到信号在几个时钟周期之前的值。在默认情况下，它提供信号在前一个时钟周期的值。结构的基本语法如下。

```
$past (signal_name, number of clock cycles)
```

这个任务能够有效地验证设计到达当前时钟周期的状态所采用的通路是正确的。属性 p19 检验的是在给定的时钟上升沿，如果表达式(c&&d)为真，那么两个周期前，表达式(a&&b)为真。

```
property p19;
  @(posedge clk) (c && d) |->
($past((a&&b), 2) == 1'b1);
endproperty

a19: assert property(p19);
```

图 1-22 显示了属性 p19 在模拟中的响应。表 1-12 总结了断言 a19 的状态和相关信号的采样值。断言在时钟周期 1 失败。在时钟周期 1，信号“c”和信号“d”都为高，断言有一个有效的开始。于是检验器的后续算子需要比较两个周期前的表达式(a&&b)的值。但是由于不可能得到两个信号在时钟周期 1 之前两个周期的历史，信号的值被当作“x”，因此检验器在时钟周期 1 失败。

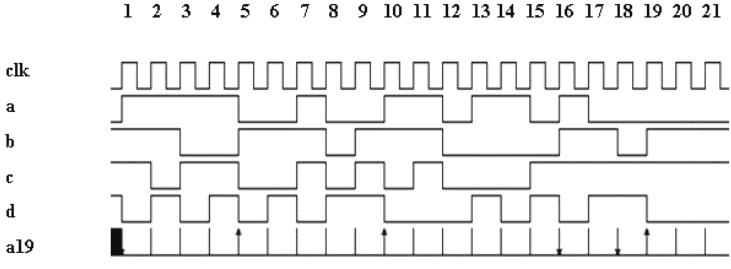

图 1-22 使用“$past”的SVA检验器的波形

检验在时钟周期 5 有一个真正的成功。在时钟周期 5，由于信号“a”和信号“b”都为高，断言有一个成功的开始。后续算子检查在时钟周期 3 时表达式(a&&b)是否为真。正如期望的那样，在时钟周期 3，信号“a”和信号“b”都被检测为高，因此检验成功。

检验在时钟周期 16 失败。在时钟周期 16，由于信号“a”和信号“b”都为高，断言也有一个成功的开始。后续算子检查在时钟周期 3 时表达式(a&&b)是否为真。信号“a”如期望的那样为高但是信号“b”为低。这使得表达式(a&&b)为假，检验失败。

表 1-12 使用“$past”的SVA检验器的真值表

时钟数	“a”的采样值	“b”的采样值	“c”的采样值	“d”的采样值	a19的状态
1	0	1	1	1	失败
2	1	1	1	0	空成功
3	1	1	0	1	空成功
4	1	0	1	0	空成功
5	1	0	1	1	真正的成功
6	0	1	0	0	空成功

(续表)

时钟数	“a”的采样值	“b”的采样值	“c”的采样值	“d”的采样值	a19的状态
7	0	1	0	1	空成功
8	1	1	1	0	空成功
9	0	0	0	1	空成功
10	0	1	1	1	真正的成功
11	1	1	0	0	空成功
12	1	1	1	0	空成功
13	0	0	0	0	空成功
14	1	0	0	1	空成功
15	1	0	0	0	空成功
16	0	0	1	1	失败
17	1	1	1	0	空成功

带时钟门控的$past 构造

$past 构造可以由一个门控信号(gating singal)控制。比如，在一个给定的时钟沿，只有当门控信号的值为真时才检查后续算子的状况。使用门控信号的$past 构造的基本语法如下：

```
$past (signal_name, number of clock cycles, gating signal)
```

属性 p20 与属性 p19 相似。但是只有当门控信号“e”在任意给定的时钟上升沿有效时检验才被激活。

```
property p20;
  @(posedge clk) (c && d) |->
            ($past((a&&b), 2, e) == 1'b1);
endproperty

a20:  assert property(p20);
```

1.21 重复运算符

如果信号“start”在任何给定的时钟上升沿跳变为高，接着从下一个时钟周期起，信号“a”保持三个连续时钟周期为高，然后下一个时钟周期，信号“stop”为高。

像上述描述的序列可以使用下面的 SVA 代码来检验。

```
@(posedge clk) $rose(start) |->
                ##1 a ##1 a ##1 a ##1 stop
```

如果信号“a”需要在很多个周期中保持高电平，编写这样一个检验器可能会非常冗长。而且这个例子要求信号“a”连续地保持高电平。当我们只希望检查信号“a”是否在被检测时保持为高，而不一定是三个连续的时钟周期的时候，协议就会变得复杂起来。换句话说，信号“a”需要连续地或者间歇地重复自己三次。

SVA 语言提供三种不同的重复运算符：连续重复(consecutive repetition)，跟随重复(go to repetition)，非连续重复(non-consecutive repetition)。

连续重复—— 允许用户表明信号或者序列将在指定数量的时钟周期内都连续地匹配。信号的每次匹配之间都有一个时钟周期的隐藏延迟。连续重复运算符的基本语法如下所示。

```
signal or sequence [*n]
```

“n”是表达式应该匹配的次数。

比如，a[*3]可以被展开成下面的式子。

```
a ##1 a ##1 a
```

而序列(a##1b)[*3]可以展开为

```
(a ##1 b) ##1 (a ##1 b) ##1 (a ##1 b)
```

跟随重复—— 允许用户表明一个表达式将匹配达到指定的次数，而且不一定在连续的时钟周期上发生。这些匹配可以是间歇的。跟随重复的主要要求是被检验重复的表达式的最后一个匹配

应该发生在整个序列匹配结束之前。跟随重复运算符的基本语法如下所示。

```
signal [->n]
```

参考下面的序列：

```
start ##1 a[->3] ##1 stop
```

这个序列需要信号“a”的匹配(即信号“a”的第三次，也就是最后一次重复的匹配)正好发生在“stop”成功之前。换句话说，信号“stop”在序列的最后一个时钟周期匹配，而且在前一个时钟周期，信号“a”有一次匹配。

非连续重复——与跟随重复相似，除了它并不要求信号的最后一次重复匹配发生在整个序列匹配前的那个时钟周期。非连续重复运算符的基本语法如下所示。

```
signal [=n]
```

在跟随重复和非连续重复中只允许使用表达式，不能使用序列。

1.21.1 连续重复运算符[*]

属性 p21 检查在检验有效地开始两个时钟周期后，信号“a”在连续的三个时钟周期为高，再过两个时钟周期，信号“stop”为高。下一个时钟周期，信号“stop”为低。

```
property p21;
  @(posedge clk) $rose(start) |->
                    ##2 (a[*3]) ##2 stop ##1 !stop;
endproperty

a21:  assert property(p21);
```

图 1-23 显示了属性 p21 在模拟中的响应。波形中显示了 2 个失败和 1 个真正的成功。其他成功都是空成功。

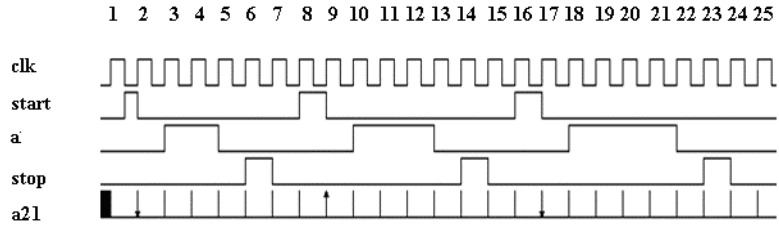

图 1-23 使用连续重复的SVA检验器的波形

断言失败于时钟周期 2——时钟周期 2 有一个有效的开始信号。检验器接着检验信号“a”是否从时钟周期 4 的上升沿开始有连续三个时钟周期为高。信号“a”在时钟周期 4 和 5 为高，但是在时钟周期 6 为低。因此检验失败。检查从时钟周期 2 开始，在时钟周期 6 失败。

断言成功于时钟周期 9——在时钟周期 9 检测到一个有效的开始。于是检验器检查信号“a”是否在时钟周期 11 开始的 3 个连续时钟周期都为高。信号“a”在时钟周期 11、12、13 都像预期的那样被检测为高。两个时钟周期后(时钟周期 15)，信号“stop”如预期地为高。一个时钟周期以后，“stop”被检测为低。至此检验成功。注意，检查从时钟周期 9 开始，结束于时钟周期 16。

断言失败于时钟周期 17——在时钟周期 17 检测到一个有效的开始。于是检验器检查信号“a”是否在时钟周期 19 开始的 3 个连续时钟周期都为高。信号“a”在时钟周期 19、20、21 都像预期的那样为高。接着检验器检查信号“stop”在时钟周期 23 是否为高，但是没检测到。因此检验失败。可以看到，信号“a”保持了 4 个时钟周期的高电平。但是检验器只需要检查 3 个重复，因此直接继续检查信号“stop”。整个检查从时钟周期 19 开始，失败于时钟周期 23。

1.21.2 用于序列的连续重复运算符[*]

属性 p22 检查有效开始的两个时钟周期以后，序列(a ##2 b)重复三次，接着再过两个时钟周期，信号“stop”为高。

```
property p22;
  @(posedge clk) $rose(start) |->
       ##2 ((a ##2 b)[*3]) ##2 stop;
endproperty

a22： assert property(p22);
```

图 1-24 显示了属性 p22 在模拟中的响应。图中共显示了 2 个失败和 1 个真正的成功。

失败 1——第一个失败由标记 1s 标出。有效的开始在这个点被检测到。两个时钟周期后，检验器期望序列(a ##2 b)重复 3 次。但是序列只重复了两次。因此检验器失败，失败点由标记 1e 标出。

成功 1——唯一一个真正的成功由标记 2s 标出。有效的开始在这个点被检测到。两个时钟周期后，检验器开始检查序列(a ##2 b)是否重复 3 次。序列如预期地重复了 3 次。在序列重复被检验后，再过两个时钟周期，信号“stop”也如期望地为高。因此检验器成功，成功点由标记 2e 标出。

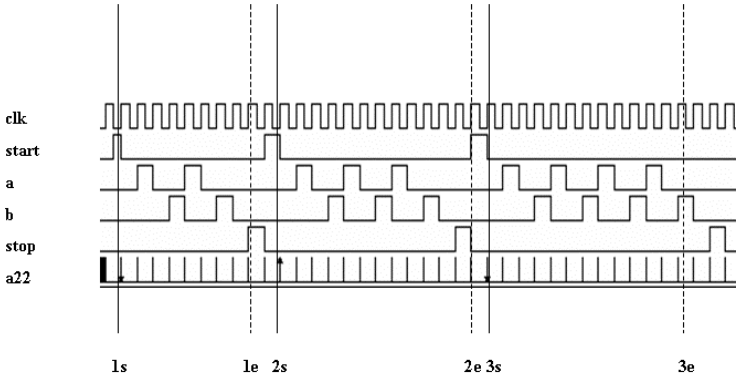

图 1-24 检查连续重复的序列的SVA检验器的波形

失败 2——第二个失败由标记 3s 标出。有效的开始在这个点被检测到。两个时钟周期后，检验器开始检查序列(a ##2 b)是否重复 3 次。序列如预期地重复了 3 次。当序列重复被检验到后，信号“stop”被期望在两个时钟周期后为高，但是失败了。因此检验器失败，失败点由标记 3e 标出。

1.21.3 用于带延迟窗口的序列的连续重复运算符[*]

属性 p23 检查在有效开始的两个时钟周期后，序列(a ##[1：4] b)重复 3 次，接着再过两个时钟周期，信号“stop”为高。实际上，这个序列有一个时序窗口，使得情况变得有些复杂。

```
property p23;
  @(posedge clk) $rose(start) |->
      ##2 ((a ##[1:4] b)[*3]) ##2 stop;
endproperty

a23： assert property(p23);
```

主序列(a ##[1：4] b)[*3]可以被扩展成：

```
((a ##1 b) or (a ##2 b) or (a ##3 b) or (a ##4 b)) ##1
((a ##1 b) or (a ##2 b) or (a ##3 b) or (a ##4 b)) ##1
((a ##1 b) or (a ##2 b) or (a ##3 b) or (a ##4 b))
```

图 1-25 显示了属性 p23 在模拟中的响应。图中有 2 个失败和 1 个真正的成功。

失败 1——第一个失败由标记 1s 标出。这一点有一个有效的开始。从这一点开始两个时钟周期后，检验器期望序列(a ##[1：4] b)重复 3 次。但是序列只重复了 2 次。因此检验器失败，失败点由标记 1e 标出。可以看到成功的两个重复分别是(a ##1 b)和(a ##2 b)。

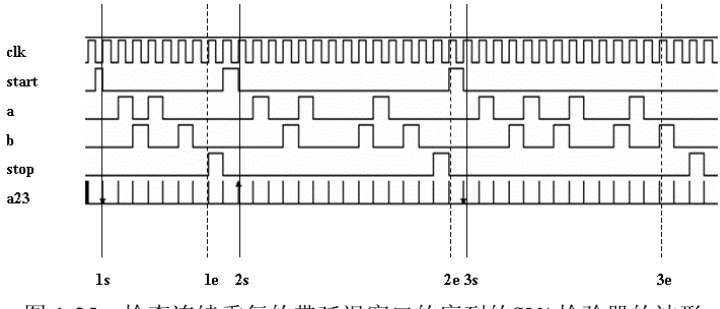

图 1-25 检查连续重复的带延迟窗口的序列的SVA检验器的波形

成功 1——唯一的真正成功由标记 2s 标出。这一点有一个有效的开始。从这一点开始两个时钟周期后，检验器期望序列(a

##[1：4] b)重复 3 次。序列如预期地重复了 3 次。在成功重复之后，信号“stop”如期望地在两个时钟周期后为高。因此检验器成功了，成功点由标记 2e 标出。可以看到成功的三个重复分别是(a ##2 b)，(a ##4 b)和(a ##2 b)。

失败 2——第二个失败由标记 3s 标出。这一点有一个有效的开始。从这一点开始两个时钟周期后，检验器期望序列(a ##[1：4] b)重复 3 次。序列如预期地重复了 3 次。在成功地重复之后，信号“stop”被期望地在两个时钟周期后为高，但是失败了。因此检验器失败，失败点由标记 3e 标出。可以看到成功的三个重复分别是(a ##2 b)，(a ##2 b)和(a ##3 b)。

1.21.4 连续运算符[*]和可能性运算符

属性 p23 指定了一个重复序列的时序窗口。同样的，重复的次数也可以是一个窗口。比如，a[*1：5]表示信号“a”从某个时钟周期开始重复 1~5 次。这个定义可以展开成下面的表达式。

```
a or
(a ##1 a) or
(a ##1 a ##1 a) or
(a ##1 a ##1 a ##1 a) or
(a ##1 a ##1 a ##1 a ##1 a)
```

重复窗口的边界规则与延迟窗口的相同。左边的值必须小于右边的值。右边的值可以是符号“$”，这表示没有重复次数的限制。

属性 p24 显示了一个带没有重复次数限制的有限的检查。它检验有效开始两个时钟周期后，信号“a”将保持为高，直到信号“stop”跳变为高。

```
property p24;
  @(posedge clk) $rose(start) |->
          ##2 (a[*1：$]) ##1 stop;
endproperty

a24： assert property(p24);
```

图 1-26 显示了属性 p24 在模拟中的响应。图中有 1 个失败和 1 个真正的成功。

失败 1——一个有效的开始发生在时钟周期 3，如标记 1s 所示。检查期望在两个时钟周期后，信号“a”将保持为高直到信号“stop”有效。信号“a”一直为高直到时钟周期 7。在时钟周期 8，“a”被检测为低，但是信号“stop”仍然不为高。因此检验在时钟周期 8 失败，如标记 1e 所示。

成功 1——一个有效的开始发生在时钟周期 11，如标记 1s 所示。检查期望在两个时钟周期后，信号“a”将保持为高直到信号“stop”有效。信号“a”一直为高直到时钟周期 15。在时钟周期 16，“a”被检测为低，但是信号“stop”如期望地为高。因此检验在时钟周期 16 成功，如标记 2e 所示。

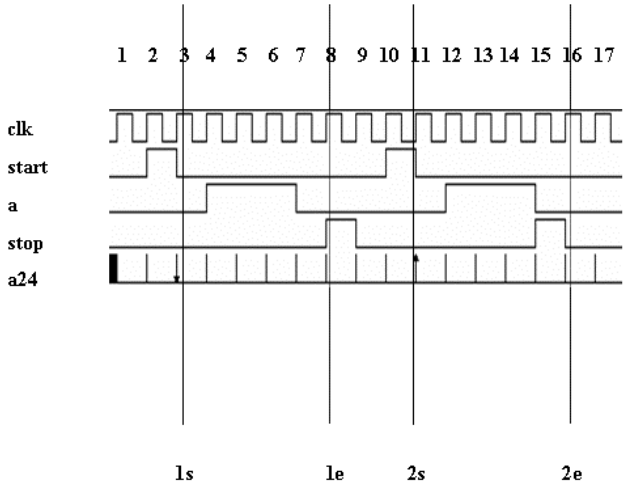

图 1-26　使用连续重复和可能性运算符的SVA检验器的波形

1.21.5　跟随重复运算符[->]

属性 p25 检查如果在任何时钟上升沿有有效的开始，两个时钟周期后，信号“a”连续或者间断地出现 3 次为高，接着信号“stop”在下一个时钟周期为高。

```
property p25;
  @(posedge clk) $rose(start) |->
```

```
                ##2 (a[->3]) ##1 stop;
    endproperty

    a25: assert property(p25);
```

图 1-27 显示了属性 p25 在模拟中的响应。图中显示共有 1 个失败、1 个成功和一个未完成的检查。

失败 1——标记 1s 标出了检验器的一个有效开始。检验器期望在有效开始的两个时钟周期后，信号“a”重复 3 次。信号如预期地重复 3 次，在信号“a”的第 3 次匹配后，信号“stop”没能如期望的那样在下一个时钟周期为高。因此检查在标记 1e 所示位置失败了。

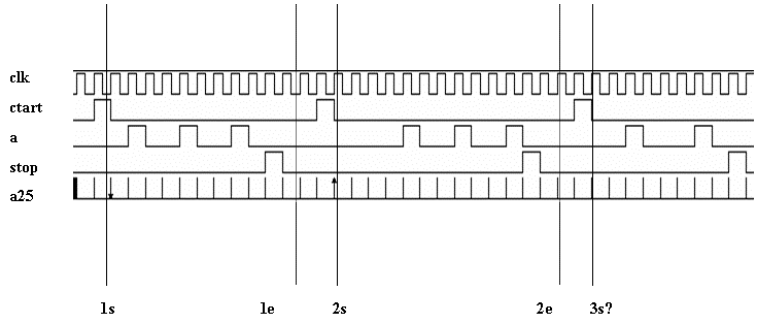

图 1-27 采用跟随重复操作符的SVA检验器的波形

成功 1——标记 2s 标出了检验器的一个有效开始。检验器期望在有效开始的两个时钟周期，信号“a”将重复 3 次。信号如预期地重复 3 次。在信号“a”的第 3 次匹配后，在下一个时钟周期信号“stop”如期望的那样为高。因此检查在标记 2e 所示位置成功了。

未完成 1——标记 3s 标出了检验器的一个有效开始。检验器期望在有效开始的两个时钟周期后，信号“a”重复 3 次。信号重复了两次，模拟就结束了。应注意到，在模拟周期结束前，信号“stop”出现了一次有效。由于重复语句还没有完成，这个有效的“stop”并没有发生任何作用。检验 3 个重复的语句阻塞了信号“stop”的检验。因此在模拟结束时这个检查未能完成。

1.21.6 非连续重复运算符[=]

属性 p26 检查如果在任何时钟上升沿有有效的开始信号，两个时钟周期后，在一个有效的“stop”信号前，信号“a”连续或者间断地出现 3 次为高，然后一个时钟周期后“stop”应该为低。p26 和 p25 做的是相同的检查，唯一的不同是 p26 使用的是非连续(non-consecutive)重复运算符而不是跟随(go to)重复运算符。这表示在属性 p26 中，在信号“stop”有效匹配的前一个时钟周期，信号“a”不一定需要有有效的匹配。

```
Property p26;
  @(posedge clk) $rose(start) |->
            ##2 (a[=3]) ##1 stop ##1 !stop;
endproperty

a26:  assert property(p26);
```

图 1-28 显示了属性 p26 在模拟中的响应。图中显示有 2 个真正的成功和 1 个未完成的检查。

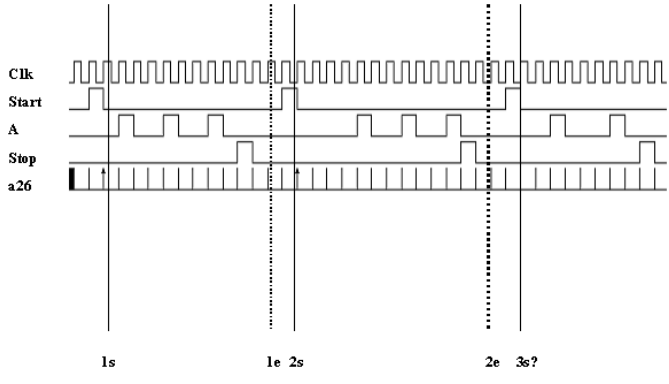

图 1-28 使用非连续重复运算符的SVA检验器的波形

成功 1——标记 1s 标出了检验器的一个有效开始。检验器期望在有效开始的两个时钟周期后，信号“a”重复 3 次。信号“a”如预期地重复 3 次，在“a”的第三个匹配后，期望一个有效的信号“stop”，但是不必在下一个时钟周期发生。实际上，在信号“a”的第三次匹配的两个时钟周期后有一个有效的信号“stop”，因此

检验如标记 1e 所示的成功了。这就是跟随重复和非连续重复的不同之处。在相同情况下，属性 p25 由于使用的是跟随重复而失败了。

成功 2——标记 2s 标出了检验器的一个有效开始。检验器期望在有效开始的两个时钟周期后，信号“a”重复 3 次。信号“a”如预期地重复 3 次，在“a”的第三次匹配后，期望一个有效的信号“stop”，但不必在下一个时钟周期发生。实际上，在信号“a”的第三次匹配的 1 个时钟周期后有一个有效的信号“stop”，因此检验如标记 2e 所示的成功了。

未完成 1——标记 3s 标出了检验器的一个有效开始。检验器期望在有效开始的两个时钟周期后，信号“a”重复 3 次。实际上，信号“a”重复了两次，在第 3 次重复出现前，模拟结束了。同样应注意，信号“stop”在模拟周期结束前曾经出现为高。因为重复语句还没有完成，所以这个“stop”并没有起到任何作用。信号“a”重复三次的语句阻塞了信号“stop”的检验。因此在模拟结束时检验未完成。这个行为与跟随重复(“go to” repetition) 相同。

1.22 “and”构造

二进制运算符“and”可以用来逻辑地组合两个序列。当两个序列都成功时整个属性才成功。两个序列必须具有相同的起始点，但是可以有不同的结束点。检验的起始点是第一个序列的成功时的起始点，而检验的结束点是使得属性最终成功的另一个序列成功时的点。

序列 s27a 和 s27b 是两个独立的序列。属性 p27 将两者用运算符“and”组合起来。当两个序列都成功时，属性成功。

```
sequence s27a;
    @(posedge clk) a##[1:2] b;
endsequence

sequence s27b;
```

```
    @(posedge clk) c##[2:3] d;
endsequence

property p27;
    @(posedge clk)  s27a and s27b;
endproperty

a27: assert property(p27);
```

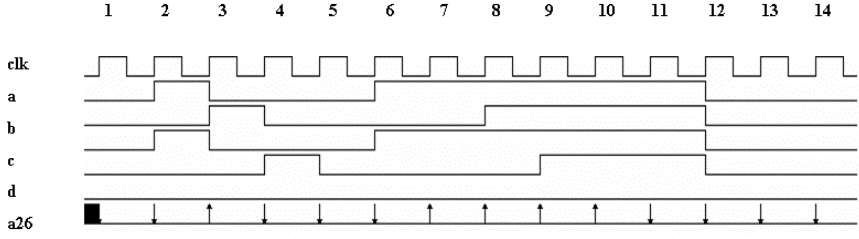

图 1-29　使用“and”构造的SVA检验器的波形

图 1-29 显示了属性 p27 在模拟中的响应。表 1-13 总结了断言 a27 的状态和所有相关信号的采样值。一共有三种结果。一种是没有有效开始所导致的失败。当在给定的时钟边沿信号“a”或者信号“c”不为高(时钟周期 1，2，4，5，6，13，14)。

表 1-13　使用“and”的SVA检验器的真值表

时钟数	“a”的采样值	“b”的采样值	“c”的采样值	“d”的采样值	有效开始	a27的状态
1	0	0	0	0	否	失败
2	0	0	0	0	否	失败
3	1	0	1	0	是	成功 (开始于 3，结束于 5)
4	0	1	0	0	否	失败
5	0	0	0	1	否	失败
6	0	0	0	0	否	失败
7	1	0	1	0	是	成功 (开始于 7，结束于 10)

(续表)

时钟数	“a”的采样值	“b”的采样值	“c”的采样值	“d”的采样值	有效开始	a27的状态
8	1	0	1	0	是	成功 (开始于 8，结束于 10)
9	1	1	1	0	是	成功 (开始于 9，结束于 11)
10	1	1	1	1	是	成功 (开始于 10，结束于 12)
11	1	1	1	1	是	失败 (开始于 11，结束于 14)
12	1	1	1	1	是	失败 (开始于 12，结束于 14)
13	0	0	0	0	否	失败
14	0	0	0	0	否	失败

还有 5 个不同的成功，它们各自的长度也不同。一个有效的检验开始于时钟周期 7 和时钟周期 8，但是它们同时结束于时钟周期 10。从时钟周期 7 开始的检验，信号“b”在时钟周期 9 为真，且信号“d”在时钟周期 10 为真。而从时钟周期 8 开始的检验，信号“b”在时钟周期 9 为真，且信号“d”在时钟周期 10 为真。

此外还有两个失败，分别在时钟周期 11 和 12。它们有相同的长度，但是失败的原因却不相同。对于从时钟周期 11 开始的检验，信号“b”在时钟周期 12 为真。但是信号“d”在时钟周期 13 和 14 都不为真，因此检验失败于时钟周期 14。对于从时钟周期 12 开始的检验，信号“b”在时钟周期 3 不为真。而且在时钟周期 14 信号“b”和信号“d”都不为真，因此检验在时钟周期 14 失败。

1.23 “intersect”构造

“intersect”运算符和“and”运算符很相似，它有一个额外要求。两个序列必须在相同时刻开始且结束于同一时刻。换句话说，两个序列的长度必须相等。

属性 p28 检验与属性 p27 相同的情况。唯一的区别是它使用的是“intersect”构造而不是“and”构造。

```
sequence s28a;
    @(posedge clk) a##[1：2] b;
endsequence

sequence s28b;
    @(posedge clk) c##[2：3] d;
endsequence

property p28;
    @(posedge clk)  s28a intersect s28b;
endproperty
a28：  assert property(p28);
```

图 1-30 显示了属性 p28 在模拟中的响应。表 1-14 总结了断言 p28 的状态和所有相关信号的采样值。图 1-30 也显示了在同一条件下使用“and”构造的断言 a27 的结果，从而可以帮助理解“and”和“intersect”的区别。

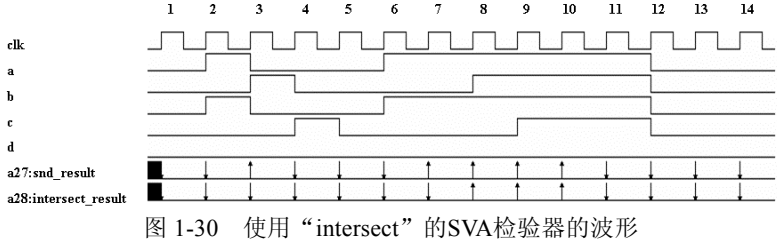

图 1-30　使用“intersect”的SVA检验器的波形

由于缺少有效开始而导致的失败与前一节相同。第二类的失败是由于属性中单个序列未能成功匹配。这类的失败发生在时钟

周期 11 和 12。第三类的失败是在属性中所有单个序列都匹配时发生的。这些失败是由于两个序列达到匹配的时间长度不一致而发生的。在时钟周期 3 显示的失败中，序列 s28a 花了一个时钟周期实现匹配(“a”在时钟周期 3 为真，且“b”在时钟周期 4 为真)，而序列 s28b 花了两个时钟周期实现匹配(“c”在时钟周期 3 为真，且“d”在时钟周期 5 为真)。在时钟周期 7 显示的失败中，s28a 花了两个时钟周期完成匹配(“a”在时钟周期 7 为真，且“b”在时钟周期 9 为真)，而 s28b 花了三个时钟周期完成匹配(“c”在时钟周期 7 为真，且“d”在时钟周期 10 为真)。

三次成功分别发生于时钟周期 8，9，10。在所有这三个例子中，两个序列都经过相同的时间长度匹配。

在时钟周期 8 显示的成功中，序列 s28a 匹配了两次，分别在时钟周期 9 和 10。序列 s28b 同样也匹配了两次，分别在时钟周期 10 和 11。两个序列匹配的共同时间长度是两个时钟周期。因此 intersect 在 s28a 和 s28b 都在时钟周期 10 时匹配成功，它们各自的长度是两个时钟周期。

表 1-14 使用“intersect”的SVA检验器的真值表

时钟数	“a”的采样值	“b”的采样值	“c”的采样值	“d”的采样值	有效开始	a28的状态
1	0	0	0	0	否	失败
2	0	0	0	0	否	失败
3	1	0	1	0	是	失败(序列匹配经历的长度不同)
4	0	1	0	0	否	失败
5	0	0	0	1	否	失败
6	0	0	0	0	否	失败
7	1	0	1	0	是	失败(序列匹配经历的长度不同)
8	1	0	1	0	是	成功 (开始于 8，结束于 10)
9	1	1	1	0	是	成功 (开始于 9，结束于 11)

(续表)

时钟数	“a”的采样值	“b”的采样值	“c”的采样值	“d”的采样值	有效开始	a28的状态
10	1	1	1	1	是	成功 (开始于 10，结束于 12)
11	1	1	1	1	是	失败 (开始于 11，结束于 13)
12	1	1	1	1	是	失败 (开始于 12，结束于 14)
13	0	0	0	0	否	失败
14	0	0	0	0	否	失败

在时钟周期 9 所示的成功中，序列 s28a 在时钟周期 10 和 11 各有一次匹配。序列 s28b 也有两次匹配，分别在时钟周期 11 和 12。两个序列匹配的通用长度为两个时钟周期。因此 intersect 构造因为 s28a 和 s28b 都在时钟周期 11 匹配而成功。每个序列的长度都是两个时钟周期。

在时钟周期 10 所示的成功中，序列 s28a 在时钟周期 11 和 12 各有一次匹配。序列 s28b 在时钟周期 12 也有一次成功。两个序列匹配的通用长度为两个时钟周期。因此 intersect 构造因为 s28a 和 s28b 都在时钟周期 12 有匹配而成功。每个序列的长度都是两个时钟周期。

1.24 “or”构造

二进制运算符“or”可以用来逻辑地组合两个序列。只要其中一个序列成功，整个属性就成功。

序列 s29a 和 s29b 是两个独立的序列。属性 p29 将两者用“or”运算符组合起来。当其中任一序列成功时，属性就成功。

```
sequence s29a;
    @(posedge clk) a##[1:2] b;

endsequence

sequence s29b;
    @(posedge clk) c##[2:3] d;
endsequence

property p29;
    @(posedge clk)  s28a or s28b;
endproperty

a29:  assert property(p29);
```

图 1-31 显示了属性 p29 在模拟中的响应。表 1-15 总结了断言 a29 的状态和所有相关信号的采样值。图 1-31 也显示了使用“and”构造的断言 a27 的结果，比较两个结果就很容易理解“and”构造和“or”构造的区别。由于没有有效开始而发生的失败和前面小节介绍的相同，第二类失败是由于其中的序列都不匹配引起的。发生在时钟周期 12 的失败就是这种失败，两个序列在它们各自的时序窗口都没能匹配，因此检验失败。

使用“and”构造和“or”构造的成功几乎是相同的，主要的区别在于匹配的时间。当序列 s29a 成功时“or”运算符就匹配了，而不需要等待序列 p29b 结束。

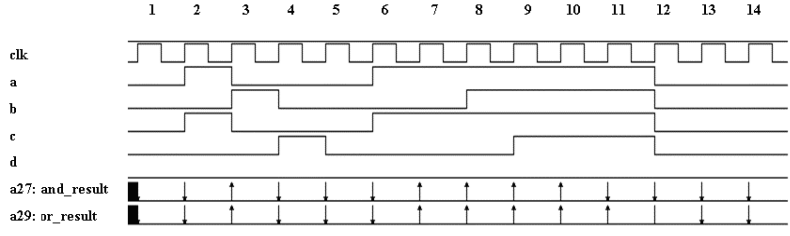

图 1-31　使用“or”的SVA检验器的波形

表 1-15 使用“or”构造的SVA检验器的波形

时钟数	“a”的采样值	“b”的采样值	“c”的采样值	“d”的采样值	有效开始	a29的状态
1	0	0	0	0	否	失败
2	0	0	0	0	否	失败
3	1	0	1	0	是	成功(开始于 3，结束于 4)
4	0	1	0	0	否	失败
5	0	0	0	1	否	失败
6	0	0	0	0	否	失败
7	1	0	1	0	是	成功(开始于 7，结束于 9)
8	1	0	1	0	是	成功(开始于 8，结束于 9)
9	1	1	1	0	是	成功(开始于 9，结束于 10)
10	1	1	1	1	是	成功(开始于 10，结束于 11)
11	1	1	1	1	是	成功(开始于 11，结束于 12)
12	1	1	1	1	是	失败(开始于 12，结束于 14)
13	0	0	0	0	否	失败
14	0	0	0	0	否	失败

其中一个使用“and”构造的失败发生在时钟周期 11，但是当使用“or”构造时，属性却成功了一次。原因是属性的第一部分序列 s29a 在时钟周期 12 匹配，使得属性立即成功。而在使用“and”构造时，单是这个成功并不是充分条件，第二部分的序列也必须匹配，但是在给定的时序窗口内，并没能出现。因此，相同的条件下，属性 p27 在时钟周期 14 失败了。

1.25 “first_match”构造

任何时候使用了逻辑运算符(如“and”和“or”)的序列中指定了时间窗，就有可能出现同一个检验具有多个匹配的情况。“first_match”构造可以确保只用第一次序列匹配，而丢弃其他的匹配。当多个序列被组合在一起，其中只需时间窗内的第一次匹配来检验属性剩余的部分时，“first_match”构造非常有用。

在下面的例子中，属性用运算符“or”将两个序列组合在一起。这个属性的几个可能的匹配如下所示。

```
a ##1 b;
a ##2 b;
c ##2 d;
a ##3 b;
c ##3 d;
```

当检验属性 p30 时，第一次匹配保留下来，其他匹配都被丢弃了。

```
sequence s30a;
    @(posedge clk) a ##[1:3] b;
endsequence

sequence s30b;
    @(posedge clk) c ##[2:3] d;
endsequence

property p30;
    @(posedge clk) first_match(s30a or s30b);
endproperty

a30: assert property(p30);
```

图 1-32 显示了属性 p30 在模拟中的响应。图中显示了两次成功，分别在时钟周期 3 和 9。在时钟周期 3 的成功基于序列(c ##2 d)的匹配。在时钟周期 9 的成功基于序列(a ##1 b)的匹配。在这两种情况中，第一次序列匹配就使得整个属性成功。

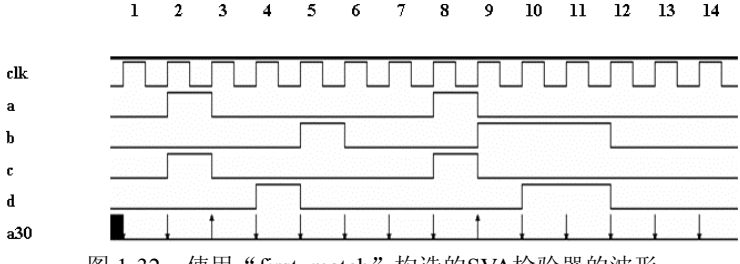

图 1-32 使用“first_match”构造的SVA检验器的波形

1.26 “throughout”构造

蕴含(implication)是目前讨论到的允许定义前提条件的一项技术。例如，要对一个指定的序列进行检验，必须某个前提条件为真。也有这样的情况，要求在检验序列的整个过程中，某个条件必须一直为真。蕴含只在时钟边沿检验前提条件一次，然后就开始检验后续算子部分，因此它不检测先行算子是否一直保持为真。为了保证某些条件在整个序列的验证过程中一直为真，可以使用“throughout”运算符。运算符“throughout”的基本语法如下所示：

```
(expression) throughout (sequence definition)
```

属性 p31 检查下列内容：

a. 在信号“start”的下降沿开始检查。
b. 检查表达式((!a&&!b) ##1 (c[->3]) ##1 (a&&b))。
c. 序列检查在信号“a”和“b”的下降沿与信号“a”和“b”的上升沿之间，信号“c”应该连续或间断地出现 3 次为高电平。
d. 在整个检验过程中，信号“start”保持为低。

```
property p31;
    @(posedge clk) $fell(start) |->
         (!start) throughout
         (##1 (!a&&!b) ##1 (c[->3]) ##1 (a&&b));
endproperty

a31:  assert property(p31);
```

图 1-33 显示了属性 p31 在模拟中的响应。检验在时钟周期 3 成功，在时钟周期 16 失败。

成功 1——信号“start”在时钟周期 3 被检测到一个下降沿，因此属性的先行算子成功。一个周期后，信号“a”和信号“b”如期在时钟周期 4 为低。之后，信号“c”如期望地分别在时钟周期 6，9，11 重复三次。接着在时钟周期 12，信号“a”和信号“b”如期为高。因此属性从时钟周期 3 开始，在时钟周期 12 成功。注

意到，信号“start”从时钟周期 3 一直到时钟周期 12 保持低电平。这是本次检验成功的关键。

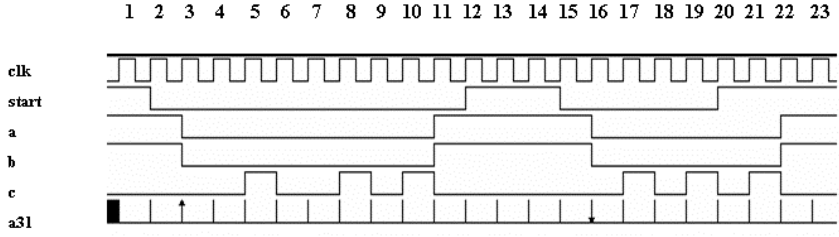

图 1-33 使用“throughout”构造的SVA检验器的波形

失败 1——信号“start”在时钟周期 16 检测到一个下降沿，因此属性的先行算子成功。一个周期后，信号“a”和信号“b”如期在时钟周期 17 为低。之后，期望信号“c”重复为高三次。我们分别在时钟周期 18 和 20 发现两次重复。但是在时钟周期 21，信号“c”的第三次重复还没出现，信号“start”就被检测出高电平，因此检验在时钟周期 21 失败。违背了“throughout”的条件导致了整个检验的失败。

1.27 “within”构造

“**within**”构造允许在一个序列中定义另一个序列。

```
seq1 within seq2
```

这表示 seq1 在 seq2 的开始到结束的范围内发生，且序列 seq2 的开始匹配点必须在 seq1 的开始匹配点之前发生，序列 seq1 的结束匹配点必须在 seq2 的结束匹配点之前结束。属性 p32 检查序列 s32a 在信号“start”的上升沿和下降沿之间发生。信号“start”的上升和下降由序列 s32b 定义。

```
sequence s32a;
    @(posedge clk)
        ((!a&&!b) ##1 (c[->3]) ##1 (a&&b));
endsequence
```

```
sequence s32b;
    @(posedge clk)
        $fell(start) ##[5:10] $rose(start);
endsequence

sequence s32;
    @(posedge clk) s32a within s32b;
endsequence

property p32;
    @(posedge clk) $fell(start) |-> s32;
endproperty

a32: assert property(p32);
```

图 1-34 使用了与 throughout 运算符用的例子相同的设计条件来显示属性 p32 在模拟中的响应。检验有两个有效的开始：一个在时钟周期 3，另一个在时钟周期 16。在这两个点，检测到信号“start”的下降沿。

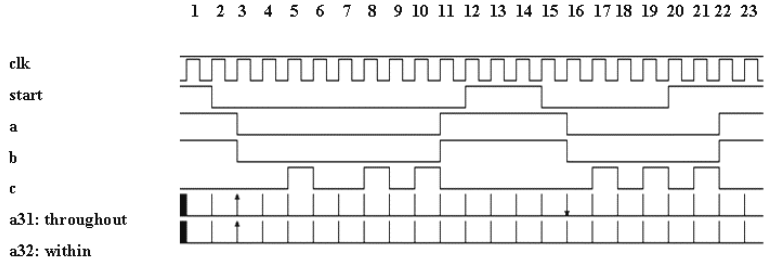

图 1-34 使用“within”构造的SVA检验器的波形

成功 1——从时钟周期 3 开始的检验成功了。信号“start”的下降沿在时钟周期 3，上升沿在时钟周期 13。在这两个时钟周期间，信号“c”分别在时钟周期 6，9，11 被检测到三次高电平。因此检验成功。

未完成 1——从时钟周期 16 开始的检验未能完成。信号“start”的下降沿在时钟周期 16，上升沿在时钟周期 21。在这两个时钟周期间，信号“c”分别在时钟周期 18 和 20 被检测到两次高电平。信号“c”的第三次重复出现在时钟周期 22，但是在时钟周期 21

检测到信号“start”为高。这是一个失败，但是由于信号“c”使用的是跟随重复(“go to”repetition)运算符，它按照阻塞序列的规则来执行。这使得检查失败并且在模拟中发出了一个未完成的信息。

1.28 内建的系统函数

SVA 提供了几个内建的函数来检查一些最常用的设计条件。

$onehot(expression)——检验表达式满足“one-hot”，换句话说，就是在任意给定的时钟沿，表达式只有一位为高。

$onehot0(expression)——检验表达式满足“zero one-hot”，换句话说，就是在任意给定的时钟沿，表达式只有一位为高或者没有任何位为高。

$isunknown(expression)——检验表达式的任何位是否是 X 或者 Z。

$countones(expression)——计算向量中为高的位的数量。

断言语句 a33a 检验向量“state”是“one-hot”。断言语句 a33b 检验向量“state”是“zero one-hot”，断言语句 a33c 检验向量“bus”是否有任何位为 X 或 Z。断言语句 a33d 检验向量“bus”中等于 1 的位的个数大于 1。

```
a33a: assert
      property(@(posedge clk) $onehot(state));
a33b: assert
      property(@(posedge clk) $onehot0(state));
a33c: assert
      property(@(posedge clk) $isunknown(bus));
a33d: assert
      property(@(posedge clk) $countones(bus)> 1);
```

图 1-35 显示了上述断言在模拟中的响应。表 1-16 总结了每个断言的状态和向量“state”和“bus”的采样值。注意，断言 a33a

在时钟周期 2 失败，因为所有位都为零。“one-hot”要求在任何时钟上升沿都只有一位为高。另一方面，断言 a33b 成功因为它检查“zero one-hot”，而对于这种构造，所有位都为零是合法的。a33a 和 a33b 都在时钟周期 5，6，7，8 失败，因为有超过一位为高。断言 a33c 在任何时候向量“bus”的值不为 X 或 Z 时失败。它在时钟周期 5，6，7 成功，因为向量的值为 Z。断言 a33d 在时钟周期 2，3，5，6，7 失败，因为值为高的位的个数没超过 1。断言 a33d 在时钟周期 4，8 成功，因为向量“bus”在这两个时刻都有两位为高。

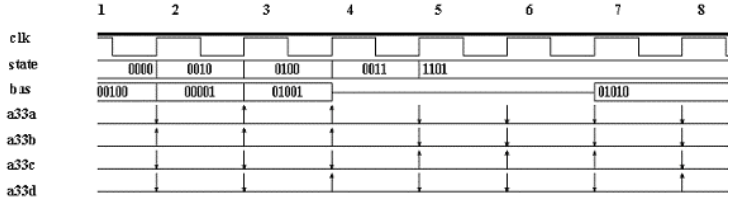

图 1-35　使用内建的系统函数的SVA检验器波形

表 1-16　使用内建函数的SVA检验器的真值表

时钟数	“state”的采样值	“bus”的采样值	a33a - $onehot 状态	a33b - $onehot0 状态	a33c - $isunknown 状态	a33d - $countones 状态
2	0000	00100	失败	成功	失败	失败
3	0010	00001	成功	成功	失败	失败
4	0100	01001	成功	成功	失败	成功
5	0011	Z	失败	失败	成功	失败
6	1101	Z	失败	失败	成功	失败
7	1101	Z	失败	失败	成功	失败
8	1101	01010	失败	失败	失败	成功

1.29 “disable iff”构造

在某些设计情况中，如果一些条件为真，则我们不想执行检验。换句话说，这就像是一个异步的复位，使得检验在当前时刻不工作。SVA 提供了关键词“disable iff”来实现这种检验器的异步复位。“disable iff”的基本语法如下。

```
disable iff (expression) < property definition>
```

属性 p34 检查在有效开始后，信号“a”重复两次，且 1 个周期之后，信号“b”重复两次，再过一个时钟周期，信号“start”为低。在整个序列过程中，如果“reset”被检测为高，检验器会停止并默认地发出一个空成功的信号。

```
property p34;
    @(posedge clk)
  disable iff (reset)
  $rose(start) |=> a[=2] ##1 b[=2] ##1 !start ;
endproperty

a34:  assert property(p34);
```

图 1-36 显示了属性 p34 在模拟中的响应。标记 1s 标出了一个有效的开始，在有效开始后，信号“a”重复为高两次，接着信号“b”重复为高两次，然后信号“start”如期望的为低。

在整个序列的过程中，信号“reset”如期望的始终不被激活，因此检验在标记 1e 处成功。第二个有效开始由标记 2s 标出。在有效开始后，信号“a”重复为高两次，接着复位信号“reset”在信号“b”重复两次之前被激活。这使得检查失效，属性得到一个空成功。

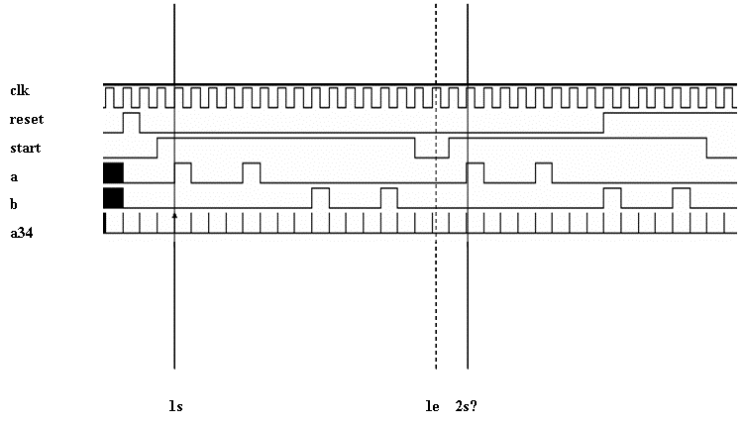

图 1-36 使用“disable iff”构造的SVA检验器的波形

1.30 使用“intersect”控制序列的长度

在 1.23 节讨论的“intersect”运算符可以有效地控制序列的长度，尤其是在时序窗口未定义上界的情况。每当使用一个可能性(eventuality)运算符时，检验器成功所需的时钟周期数没有限制。运算符 intersect 提供了一个定义可能性运算符可以使用的最小和最大时钟周期数的机制。

属性 p35 定义了一个序列来检验在给定时钟边沿，如果信号“a”为高，那么从下一个时钟周期开始信号“b”最终将为高，接着在下一个时钟周期开始信号“c”最终也会为高。这个序列每当信号“a”为高时就开始，并且可能一直到整个模拟结束时才成功。这可以使用带 1[*2:5]的 intersect 运算符来加以约束。这个 intersect 的定义检查从序列的有效开始点(信号“a”为高)，到序列成功的结束点(信号“c”为高)，一共经过 2~5 个时钟周期。

```
property p35;
   (@(posedge clk)  1[*2:5] intersect
                    (a ##[1:$] b ##[1:$] c));
endproperty

a35:  assert property(p35);
```

图 1-37 显示了属性 p35 在模拟中的响应。表 1-17 总结了断言 a35 的状态和相关信号的采样值。在一个给定的时钟边沿，如果信号“a”未被检测为高，那么这是一个失败。这种情况发生在时钟周期 1，3，4，5，11 和 13，这些时刻没有有效开始。

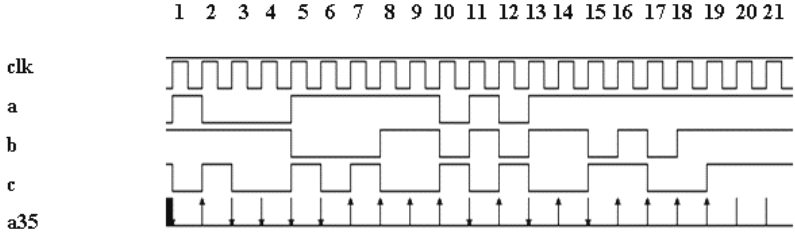

图 1-37 使用intersect来控制序列长度的SVA检验器的波形

检验在时钟周期 2，7，8，9，10，12 和 14 成功。可以看到序列从开始到结束至多花了 5 个时钟周期。检验在时钟周期 6 有一个真正的失败。在时钟周期 6 检测到信号“a”为高，而且在时钟周期 9 信号“b”为高。但是在整个检查达到允许的最大长度，即时钟周期 10，信号“c”依然未能为高，因此检验在时钟周期 10 失败。可以看到信号“c”在时钟周期 11 为高，但是这已经太晚了。

表 1-17 使用intersect构造来控制序列长度的SVA检验器的真值表

时钟数	“a”的采样值	“b”的采样值	“c”的采样值	有效开始	a35 的状态
1	0	1	1	否	失败
2	1	1	0	是	成功 (开始于 2，结束于 6)
3	0	1	1	否	失败
4	0	1	0	否	失败
5	0	1	0	否	失败
6	1	0	1	是	失败 (开始于 6，结束于 10)
7	1	0	0	是	成功 (开始于 7，结束于 11)
8	1	0	1	是	成功 (开始于 8，结束于 11)
9	1	1	0	是	成功 (开始于 9，结束于 11)
10	1	1	0	是	成功 (开始于 10，结束于 13)

(续表)

时钟数	“a”的采样值	“b”的采样值	“c”的采样值	有效开始	a35 的状态
11	0	0	1	否	失败
12	1	1	0	是	成功 (开始于 12，结束于 16)
13	0	0	1	否	失败
14	1	1	0	是	成功 (开始于 14，结束于 16)
15	1	1	0	是	失败
16	1	0	1	是	成功
17	1	1	1	是	成功

1.31 在属性中使用形参

可以用定义形参(formal arguments)的方式来重用一些常用的属性。属性“arb”使用了 4 个形参，并且根据这些形参进行检验。其中还定义了特定的时钟。SVA 允许使用属性的形参来定义时钟。这样，属性可以应用在使用不同时钟的相似设计模块中。同样的，时序延迟也可以参数化，这使得属性的定义更具有普遍性。

属性首先检查有效开始。在给定的时钟上升沿，如果在信号“a”的下降沿后的 2~5 个时钟周期内出现信号“b”的下降沿，那么这就是一个有效开始。如果先行算子匹配，那么属性接着检查信号“c”和信号“d”的下降沿在下一个时钟周期出现，并且确保它们在 4 个连续的周期都保持为低。接着一个周期后，必须检测到信号“c”和信号“d”都为高，且再过一个时钟周期应该检测到信号“b”为高。

假定这是处理三个具有相似信号的不同主控设备的仲裁器的协议，可以很容易地重用前面定义的属性来检验所有 3 个主控设备的接口。断言 a36_1，a36_2 和 a36_3 定义了每个主控接口用的检验器，分别使用了各个接口对应的信号作为属性 arb 的参数。

```
property arb (a, b, c, d);
    @(posedge clk) ($fell(a) ##[2:5] $fell(b)) |->
```

```
        ##1 ($fell(c) && $fell(d)) ##0
      (!c&&!d) [*4] ##1 (c&&d) ##1 b;
endproperty

a36_1: assert property(arb(a1, b1, c1, d1));
a36_2: assert property(arb(a2, b2, c2, d2));
a36_3: assert property(arb(a3, b3, c3, d3));
```

图 1-38 显示了每个接口对应的断言在模拟过程中的响应。断言 a36_1 有一个有效开始，并且检验成功。断言 a36_3 有一个有效开始但是检验失败了。

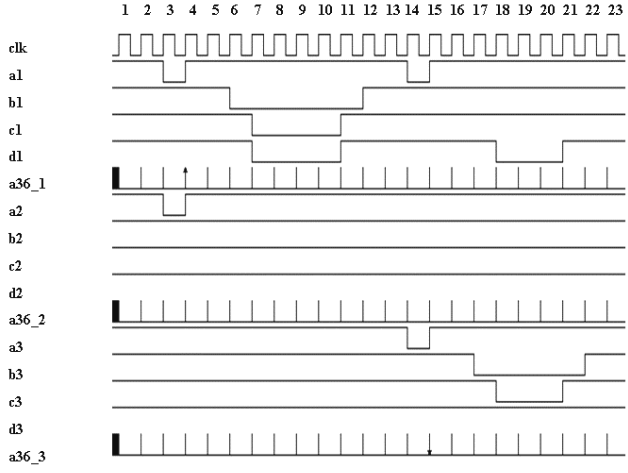

图 1-38 在属性中使用形参的SVA检验器的波形

成功 1(a36_1)——在时钟周期 4，当信号“a1”的下降沿出现时，检验开始。期望在 2~5 个时钟周期内信号“b1”出现一个下降沿，下降沿在时钟周期 7 如期出现。在下一个时钟周期，信号“c1”和信号“d1”如期望的为低，并且必须保持为低 4 个时钟周期。它们在时钟周期 8~11 都为低。接着在时钟周期 12，信号“c1”和“d1”如期望的为高。然后在时钟周期 13，信号“b1”如期望地为高。因此检验开始于时钟周期 4，成功于时钟周期 13。

失败 1(a36_3)——在时钟周期 15，当信号“a3”的下降沿出现时，检验开始。期望在 2~5 个时钟周期内信号“b3”出现一个下降沿，下降沿如期在时钟周期 18 出现。在下一个时钟周期，信

号“c3”和“d3”应该为低。由于未检测到信号“d3”为低，检验失败于时钟周期 19。

1.32 嵌套的蕴含

SVA 允许使用嵌套的蕴含。当我们有多个门限条件指向一个最终的后续算子时，这种构造十分有用。

属性 p_nest 检验如果信号“a”有一个下降沿，则是一个有效开始，接着在一个周期后，信号“b”，“c”和“d”应该为低电平有效信号以保持这个有效开始。如果第二个条件匹配，那么在 6 到 10 个周期内期望“free”为真。注意，当且仅当信号“b”，“c”和“d”都匹配时，在后续状况(consequent condition)“free”才会被检验是否为真。

```
`define free (a && b && c && d)

property p_nest;
       @(posedge clk) $fell(a) |->
                     ##1 (!b && !c && !d) |->
                                          ##[6: 10]
                                          `free;
endproperty

a_nest: assert property(p_nest);
```

同一个属性可以被重写成不使用嵌套蕴含的方式，如下所示。

```
property p_nest1;
       @(posedge clk) $fell(a) ##1 (!b && !c && !d)
                     |-> ##[6:10] `free;
endproperty

a_nest1: assert property(p_nest1);
```

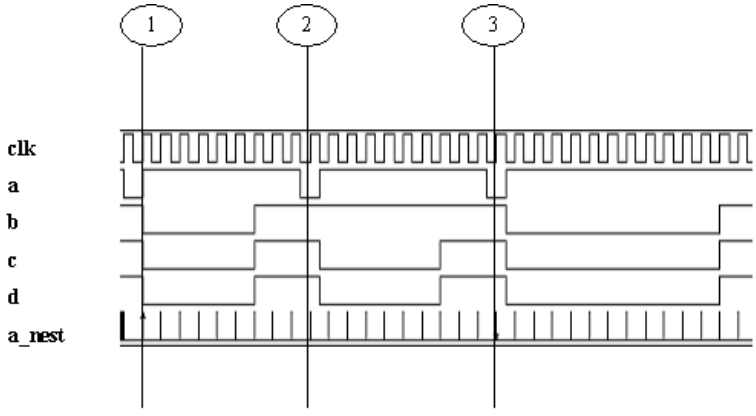

图 1-39 使用嵌套蕴含的SVA检验器

注意：

使用嵌套蕴含的属性 p_nest 中没有“else”情况，因此属性很容易就能重写成如 p_nest1 所示的形式。

图 1-39 显示了断言 a_nest 在模拟中的表现。标记 1 显示了检验器的第一次成功。当检测到信号“a”的下降沿时，出现一个有效开始。一个时钟周期后，信号“b”、“c”和“d”如预期地被检测为低。因此检验保持有效，后续算子被检验。在 6 个时钟周期后，检测到状况“free”为真，因此检验成功。

第二个标记指出了下一个有效开始，在此检测到信号“a”的下降沿。一个周期后，检测到信号“c”和“b”为低，但是信号“b”不为低。因此检验没能保持有效，得到一个空成功。

第三个标记也标出了一个有效开始，在此检测到信号“a”的下降沿。一个周期后，如期望的检测到信号“b”，“c”和“d”为低，因此检验保持有效，后续算子被检验。在 6~10 个时钟周期内，没有检验到状况“free”为真，因此检验失败。

1.33 在蕴含中使用 if/else

SVA 允许在使用蕴含的属性的后续算子中使用“if/else”语句。

属性 p_if_else 检查如果信号“start”的下降沿被检测到，就是一个有效开始，接着一个时钟周期后，信号“a”或者信号“b”为高。在现行算子成功匹配时，后续算子有两个可能的路径。

1. 如果信号“a”为高，那么信号“c”必须间歇地重复两次为高，且在下一个时钟周期，信号“e”必须为高。

2. 如果信号“a”不为高，那么信号“d”必须间歇地重复两次为高，且在下一个时钟周期，信号“f”必须为高。

注意，在检验信号“a”的后续算子中有优先级。

```
property p_if_else;
    @(posedge clk)
  ($fell(start) ##1 (a||b)) |->
        if(a)
      (c[->2] ##1 e)
        else
      (d[->2] ##1 f);
endproperty

a_if_else: assert property(p_if_else);
```

如果不用“if/else”构造来重写这个属性，需要用三个单独的属性来实现。由于“if/else”有优先级，两个信号导致了如下所示的三种不同的可能结果：

a	b	分支(Leaf)
1	0	a
0	1	b
1	1	a

可以发现，如果信号“a”和“b”都为高，那么会执行信号“a”的“if”块，因为它的优先级高。重写的三个属性如下所示。

```
property p_if_else_leaf1;
    @(posedge clk)
        ($fell(start) ##1 a) |->
                              (c[->2] ##1 e);

endproperty
```

```
a_if_else_leaf1:
       assert property(p_if_else_leaf1);

property p_if_else_leaf2;
    @(posedge clk)
        ($fell(start) ##1 b) |->
                            (d[->2] ##1 f);
endproperty

a_if_else_leaf2:
       assert property(p_if_else_leaf2);

property p_if_else_leaf3;
    @(posedge clk)
        ($fell(start) ##1 (a &&b)) |->
                             (c[->2] ##1 e);

endproperty

a_if_else_leaf3:
       assert property(p_if_else_leaf3);
```

1.34 SVA 中的多时钟定义

SVA 允许序列或者属性使用多个时钟定义来采样独立的信号或者子序列。SVA 会自动地同步不同信号或子序列使用的时钟域。下面的代码显示了一个序列使用多个时钟的简单例子。

```
sequence s_multiple_clocks;
        @(posedge clk1) a ##1 @(posedge clk2) b;
endsequence
```

序列 s_multiple_clocks 检验在时钟“clk1”的任何上升沿，信号“a”为高，接着在时钟“clk2”的上升沿，信号“b”为高。当信号“a”在时钟“clk1”的任意给定上升沿为高时，序列开始匹配。接着“##1”延迟构造将检验时间移到时钟“clk2”的最近的

上升沿，检查信号“b”是否为高。当在一个序列中使用了多个时钟信号时，只允许使用“##1”延迟构造。序列 s_multiple_clocks 不能被重写成下面这种形式。

```
sequence s_multiple_clocks_illegal1;
        @(posedge clk1) a ##0 @(posedge clk2) b;
endsequence

sequence s_multiple_clocks_illegal2;
        @(posedge clk1) a ##2 @(posedge clk2) b;
endsequence
```

使用“##0”会产生混淆，即在信号“a”匹配后究竟哪个时钟信号才是最近的时钟。这将引起竞争，因此不允许使用。使用##2 也不允许，因为不可能同步到时钟“clk2”的最近的上升沿。

相似的技术可以用来建立具有多个时钟的属性。如下面的例子所示：

```
property p_multiple_clocks;
    @(posedge clk1) s1 ##1 @(posedge clk2) s2;
endproperty
```

它假定序列 s1 没有被时钟驱动，或者它的时钟定义和“clk1”一样。它又假定序列 s2 没有被时钟驱动，或者它的时钟定义和“clk2”一样。同样的，属性可以在序列定义之间使用非交叠蕴含运算符。下面是一个简单的例子：

```
property p_multiple_clocks_implied;
    @(posedge clk1) s1 |=> @(posedge clk2) s2;
endproperty
```

禁止在两个不同时钟驱动的序列之间使用交叠蕴含运算符。因为先行算子的结束和后续算子的开始重叠，可能引起竞争的情况，这是非法的。下面的代码显示了这种非法的编码方式：

```
property p_multiple_clocks_implied_illegal;
    @(posedge clk1) s1 |-> @(posedge clk2) s2;
endproperty
```

1.35 “matched”构造

任何时候如果一个序列定义了多个时钟，构造“**matched**”可以用来监测第一个子序列的结束点。序列 s_a 查找信号“a”的上升沿。而信号“a”是根据时钟“clk1”来采样的。序列 s_b 查找信号“b”的上升沿。信号“b”则是根据时钟“clk2”来采样的。属性 p_match 验证在给定的时钟“clk2”的上升沿，如果序列 s_a 匹配，那么在一个周期后，序列 s_b 也必须为真。

```
sequence s_a;
    @(posedge clk1) $rose(a);
endsequence

sequence s_b;
    @(posedge clk2) $rose(b);
endsequence

property p_match;
    @(posedge clk2) s_a.matched |=> s_b;
endproperty

a_match: assert property(p_match);
```

图 1-40 显示了断言 a_match 在模拟中的表现。属性在序列 s_a 匹配的时候得到有效开始。注意，虽然序列 s_a 是根据时钟“clk1”来采样的，但我们只在时钟“clk2”的每个上升沿查找这种匹配。

在“clk1”的时钟周期 3，信号“a”有一次有效的上升。这将更新序列 s_a 的匹配值为真。这个值一直被保持到 “clk2”的最近的时钟上升沿，也就是“clk2”的时钟周期 2。在这一个时间点，属性被激活，并且在“clk2”的下一个时钟周期，期望序列 s_b 匹配。因此属性的第一次成功开始于“clk2”的时钟周期 2，结束于“clk2”的时钟周期 3。

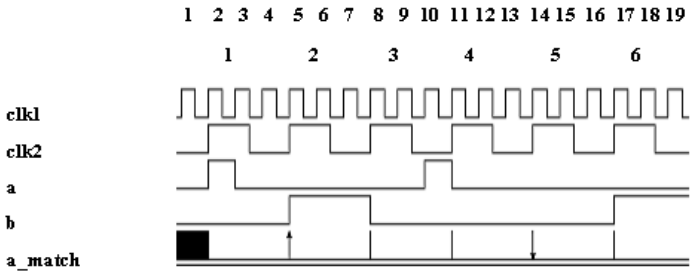

图 1-40 使用“matched”构造的SVA检验器

信号“a”的另一个有效上升发生在信号“clk1”的时钟周期11，并且在“clk2”的时钟周期 5 被属性采样到。属性在这个点被激活，并且期望在“clk2”的时钟周期 6，序列 s_b 匹配。但是这次，信号“b”的上升沿没能出现，因此属性失败。理解“matched”构造的使用方法的关键在于，被采样到的匹配的值一直被保存到另一个序列最近的下一个时钟边沿。

1.36 “expect”构造

SVA 支持一种叫“expect”的构造，它与 Verilog 中的等待构造相似，关键的区别在于 expect 语句等待的是属性的成功检验。expect 构造后面的代码是作为一个阻塞的语句来执行。expect 构造的语法与 assert 构造很相似。expect 语句允许在一个属性成功或者失败后使用一个执行块(action block)。使用 expect 构造的实例如下所示。

```
initial

begin
@(posedge clk);
#2ns cpu_ready = 1'b1;
expect(@(posedge clk) ##[1: 16]
                        memory_ready == 1'b1)
    $display("Hand shake successful\n");
else
    begin
```

```
        $display("Hand shake failed: exiting\n")
        $finish();
        end

    for(i=0; i<64; i++)
    begin
        send_packet();
        $display("PACKET %0d sent\n", i);
    end

    end
```

注意，在信号“cpu_ready”被断言后，expect 语句等待信号“memory_ready”在 1~16 个周期内的任意周期被断言。如果信号“memory_ready”如预期的被断言，那么显示一个成功信息，并且开始执行“for”循环代码。如果信号“memory_ready”没能如预期的被断言，那么显示错误信息，且模拟结束。

1.37 使用局部变量的 SVA

在序列或者属性的内部可以局部定义变量，而且可以对这种变量进行赋值。变量接着子序列放置，用逗号隔开。如果子序列匹配，那么变量赋值语句执行。每次序列被尝试匹配时，会产生变量的一个新的备份。

```
property p_local_var1;
int lvar1;
@(posedge clk)
($rose(enable1), lvar1 = a) |->
          ##4 (aa == (lvar1*lvar1*lvar1));
endproperty

a_local_var1: assert property(p_local_var1);
```

属性 p_local_var1 查找信号“enable1”的上升沿。如果找到，

局部变量“lvar1”保存设计中向量“a”的值。在 4 个周期后，检查设计的输出向量“aa”是否与局部变量的值的立方相等。属性的后续算子等待设计满足延迟(4 个时钟周期)，然后将设计的实际输出和属性局部计算的值比较。图 1-41 显示了检验在模拟中的响应。

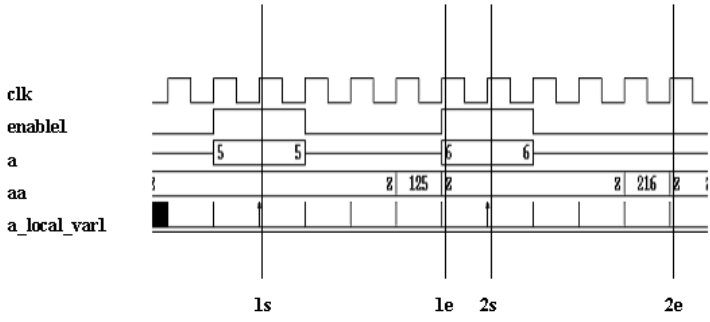

图 1-41　使用局部变量的SVA的波形

标记 1s 显示了信号“enable1”的上升沿被采样到的点，在这个点，向量“a”的值为 5，被保存在局部变量“lvar1”中。标记 1e 标出了输出被采样的点，它在输入值被保存的 4 个时钟周期之后。在标记 1e 的点，因为输出值(125)与局部变量“lvar1”的值的立方相等，断言成功。类似地，标记 2s 显示了下一个输入数据被保存的时刻，标记 2e 标出了输出被采样并且与局部变量“lvar1”的立方值比较的时间点。

可以在 SVA 中保存和操作局部变量。

```
property p_lvar_accum;
int lvar;
@(posedge clk) $rose(start)|=>
(enable1 ##2 enable2, lvar = lvar +aa) [*4]
##1 (stop && (aout == lvar));
endproperty

a_lvar_accum : assert property(p_lvar_accum);
```

属性 p_lvar_accum 检查下列内容：

(1) 在任意给定的时钟上升沿，如果检测到信号“start”的上升沿，标志一个有效开始。

(2) 在一个周期后，寻找一个特定的模型或者子序列。信号“enable1”必须被检测为高，且两个周期后，“enable2”应该被检测为高。这个子序列必须连续重复 4 次。

(3) 在子序列的每次重复中，向量“aa”的值在序列内部被累加。在重复结束时，局部变量保存着向量“aa”累加 4 次的值。

(4) 在重复结束的下一个时钟周期，期望信号“stop”为高，且局部变量保存的值与输出向量“aout”的值相等。

图 1-42 显示了检验在模拟中的响应。

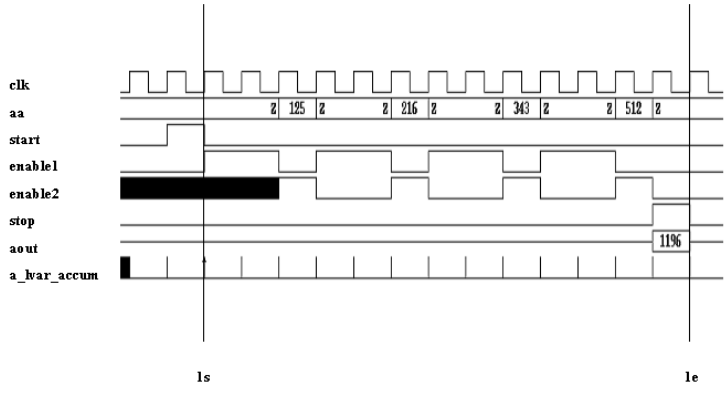

图 1-42 使用局部变量赋值的SVA

标记 1s 显示了当信号“start”被检测为高时所产生的一个有效开始。标记 1e 显示了检验的结束点。信号“enable*”成功地重复 4 次并且在一个时钟周期后，信号“stop”如期望的被检测为高。局部变量保存的值与输出向量“aout”值相同，因此检验在标记 1e 处成功。

1.38 在序列匹配时调用子程序

SVA 可以在序列每次成功匹配时调用子程序。同一序列中定义的局部变量可以作为参数传给这些子程序。对于序列的每次匹配，子程序调用的执行与它们在序列定义中的顺序相同。

```
sequence s_display1;
@(posedge clk)
   ($rose(a), $display("Signal a arrived at %t\n",
$time));
endsequence

sequence s_display2;
@(posedge clk)
   ($rose(b), $display("Signal b arrived at %t\n",
$time));
endsequence

property p_display_window;
@(posedge clk)
s_display1 |-> ##[2: 5] s_display2;
endproperty

a_display_window :
      assert property(p_display_window);
```

序列 s_display1 查找信号“a”的上升沿。如果匹配，就执行 display 语句。序列 s_display2 对信号“b”作类似的检查。属性 p_display_window 检验如果序列 s_display1 出现，那么序列 s_display2 必须在 2~5 个时钟周期之间的某个时刻出现。使用 display 语句，用户可以得到精确的信息，了解后续序列经过多少个时钟周期完成。图 1-43 显示了检验在模拟中的响应。

标记 1s 显示了由于检测到信号“a”的上升沿而得到的一个检验器的有效开始。在这一点，SVA 执行序列 s_display1 的 display 语句。标记 1e 显示了信号“b”出现上升沿的点。因为它出现在 3 个时钟周期后，所以检验成功。在这个点上，执行序列 s_display2 的 display 语句。

标记 2s 显示了由于检测到信号“a”的上升沿而得到的检验器的另一个有效开始。在这一点，SVA 执行序列 s_display1 的 display 语句。标记 2e 显示了检验器的结束点。信号“b”的有效上升沿没能在 2~5 个时钟周期内出现，因此检验失败。由于第二个序列没有匹配，序列相关的 display 语句没有执行。SVA 发出一

个默认的出错信息。

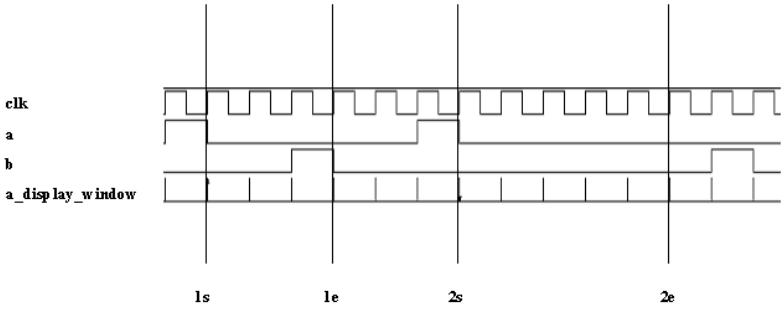

图 1-43 在序列匹配时使用子程序的SVA

一个模拟日志的实例如下所示。

```
Signal a arrived at                125

Signal b arrived at                275

"sub.v", 45:  sub.a_display_window:
started at 125s succeeded at 275s

Signal a arrived at                425

"sub.v", 45:  sub.a_display_window:
started at 425s failed at 675s
       Offending '$rose(b)'
```

1.39 将 SVA 与设计连接

有两种方法可以将 SVA 检验器连接到设计中。

(1) 在模块(module)定义中内建或者内联检验器。

(2) 将检验器与模块、模块的实例或者一个模块的多个实例绑定。

有的工程师不喜欢在设计中加任何验证代码。在这种情况下，在外部绑定 SVA 检验器是很好的选择。SVA 代码可以内建在模块定义中的任何地方。下面的例子显示了内联在模块中的 SVA。

```
module inline(clk, a, b, d1, d2, d);

input logic clk, a, b;
input logic [7: 0] d1, d2;
output logic [7: 0] d;

always@(posedge clk)
begin
if(a)
    d <= d1;
if(b)
    d <= d2;
end

property p_mutex;
@(posedge clk) not (a && b);
endproperty

a_mutex: assert property(p_mutex);

endmodule
```

如果用户决定将 SVA 检验器与设计代码分离，那么就需要建立一个独立的检验器模块。定义独立的检验器模块，增强了检验器的可重用性。下面所示的是一个检验器模块的代码实例。

```
module mutex_chk(a, b, clk);

input logic a, b, clk;
property p_mutex;
    @(posedge clk) not (a && b);
endproperty

a_mutex: assert property(p_mutex);

endmodule
```

注意，定义检验器模块时，它是一个独立的实体。检验器用来检验一组通用的信号，检验器可以与设计中任何的模块(module)或者实例(instance)绑定，绑定的语法如下所示。

```
bind <module_name or instance name>
        <checker name> <checker instance name>
                <design signals>;
```

在上面的检验器例子中，绑定可以用下面的方式实现。

```
bind inline mutex_chk i2 (a, b, clk);
```

在实现绑定时，使用的是设计中的实际信号。

比如，我们有一个如下所示的顶层模块。

```
module top (..);

inline u1 (clk, a, b, in1, in2, out1);
inline u2 (clk, c, d, in3, in4, out2);

endmodule
```

检验器 mutex_chk 可以用下面的方式与顶层模块中内联(inline)的两个模块实例绑定。

```
bind top.u1 mutex_chk i1(a, b, clk);
bind top.u2 mutex_chk i2(c, d, clk);
```

与检验器绑定的设计信号可以包含绑定实例中的任何信号的跨模块引用(cross module reference)。

1.40 SVA 与功能覆盖

功能覆盖是按照设计规范衡量验证状态的一个标准，它可以分成两类。

a. 协议覆盖。

b. 测试计划覆盖。

断言可以用来获得有关协议覆盖的穷举信息。SVA 提供了关键词“cover”来实现这一功能，cover 语句的基本语法如下所示。

```
<cover_name> :  cover property(property_name)
```

“cover_name”是用户提供的名称，用来标明覆盖语句，

“property_name”是用户想获得覆盖信息的属性名。例如，在1.39节定义的检验器“mutex_chk”，可以如下所示来检查它的覆盖情况。

```
c_mutex： cover property(p_mutex);
```

cover语句的结果包含下面的信息：

(1) 属性被尝试检验的次数。

(2) 属性成功的次数。

(3) 属性失败的次数。

(4) 属性空成功的次数。

检验器“mutex_chk”在一次模拟中的覆盖日志的实例如下所示。

```
c_mutex，12 attempts，12 match，0 vacuous match
```

就像断言(assert)语句一样，覆盖(cover)语句可以有执行块。在一个覆盖成功匹配时，可以调用一个函数(function)或者任务(task)，或者更新一个局部变量。

第 2 章 SVA 模拟方法论

在第 1 章中，通过举例详细介绍了 SVA 语言的结构。所有例子都阐明了两个或更多的通用信号间的关系，而没有涉及任何实际设计的详情。在第 2 章中，用一个虚拟的系统表示实际设计的情况。本章将逐步讨论协议的析取和断言的开发过程，并讨论多种能显著提高基于断言的验证(ABV)生产率的模拟方法，本章还将详细讨论功能覆盖和互动测试平台的开发。

2.1 一个被验证的实例系统

这个被验证的实例系统如图 2-1 所示。它有 3 个主控设备(master device)和 2 个目标设备(target device)。主控设备和目标设备之间通过一个中间设备(mediator)相连。在同一时刻，只能有一个主控设备与一个目标设备交互。任何主控设备都可以和任何一个目标设备交互。事务可以是一个读操作或写操作。中间设备有一个仲裁逻辑来决定哪个主控设备管理事务，仲裁器采用一种简单的循环技术。中间设备还包含一个胶合逻辑(glue logic)，实际上将主控设备的信息解码提供给目标设备，反之亦然。胶合逻辑有助于建立一个指定的主控设备和目标设备之间的连接，及成功管理这个事务。

2.1.1 主控设备

图 2-2 显示了有输入和输出端口的主控设备的方框图。主控

设备可以执行读和写操作。它支持在单个系统中控制两个目标设备。当主控设备收到指令“ask_for_it”(取信息)，就准备开始一个事务，发出一个低电平有效的脉冲到信号“req”上，然后等待一个“gnt”信号。“gnt”信号是一个低电平有效信号。如果在 2~5 个时钟周期内没有等到“gnt”信号，稍后主控设备将重试这个过程。如果在 2~5 个时钟周期内获得了“gnt”信号，主控设备将立即断言(assert)信号“frame”和“irdy”，用来确认“gnt”信号的到达(“frame”和“irdy”信号是低电平有效信号)。在同一个时钟周期，主控设备也要选择与它建立事务的目标设备。主控设备用输出信号“rsel”来表示，如果信号“rsel”设为 1，则主控设备要与目标设备 1 建立事务，如果信号“rsel”设为 0，则主控设备要与目标设备 0 建立事务。

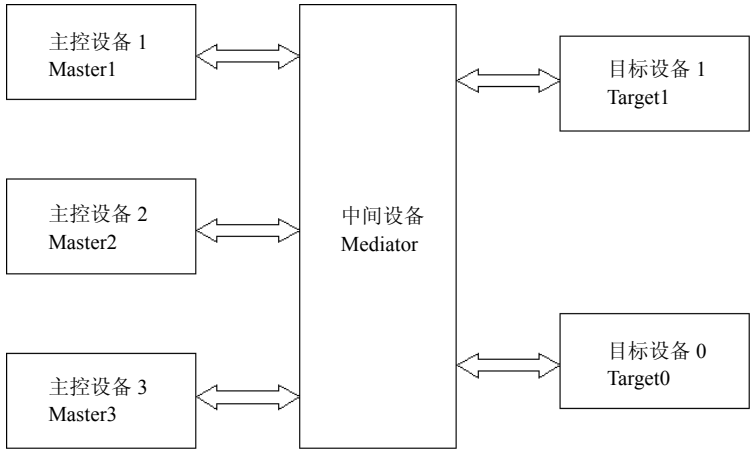

图 2-1 一个实例系统

一旦“rsel”信号被更新，目标设备需要向主控设备确认自己响应了选中。目标设备用信号“trdy”表明它已准备就绪。如果在“rsel”被更新后的 3 个时钟周期内目标设备没有确认它自身，这就是出错的情况。如果目标设备确认了它自身，那么主控设备要决定是否读或写。主控设备通过“datac”总线发送数据和指令来读或写数据。

图 2-2 主控设备实例

最高位是指令位(如波形图中的信号“rw”所示)，如果指令位为 1，则主控设备将执行写操作。如果它为 0，则主控设备执行读操作。如果要执行一个写操作，最低的 8 位数据被写到目标设备。如果要执行一个读操作，则从目标设备读入的数据将会出现在输入总线“datao”上。主控设备的每次事务将会精确地持续 8 个时钟周期。换言之，在一个事务中主控设备要么读 8 个字节要么写 8 个字节。由于没有指定的地址生成方案，主控设备将会写到目标设备中最新更新的写指针所指的地址。同样地，主控设备会从目标设备中最新更新的读指针指向的地址读取。图 2-3 所示的是主控设备执行写事务的实例波形。图 2-4 所示的是主控设备执行读事务的实例波形。

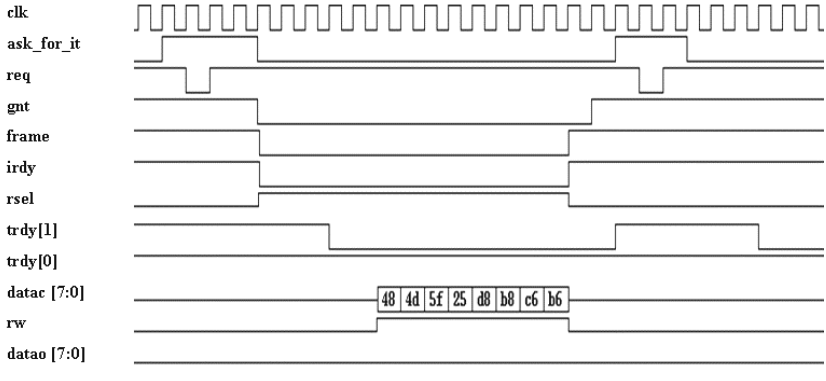

图2-3 一个主控设备的写事务

一旦读或写操作完成，主控设备在下一个时钟周期将信号“frame”和“irdy”的断言解除以示操作完成，它也把“rsel”信号设为三态(tri-state)。在下一时钟，仲裁器确认“rsel”信号，并

将“gnt”信号解除断言。一旦仲裁器清除了“gnt”信号，目标设备通过将“trdy”信号解除断言来确认整个事务的完成。

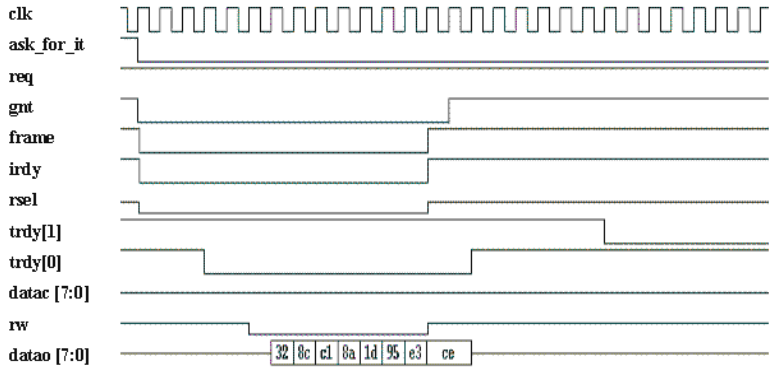

图 2-4 一个主控设备的读事务实例

2.1.2 中间设备

图 2-5 显示了有输入和输出端口的中间设备的方框图。中间设备负责两个重要的任务：

(1) 提供仲裁逻辑来决定哪个主控设备管理与目标设备之间的事务。

(2) 建立一个特定的主控设备和一个目标设备之间的连接。在一个给定的时间，许多主控设备可以通过断言各自的“req”信号来请求执行事务。

仲裁器使用轮转(round-robin)算法来决定哪个主控设备获得管理权。当仲裁器做出决定，它会将对应的主控设备的“gnt”信号断言。仲裁器可以在 2~5 个时钟周期内做出决定。仲裁器的内部逻辑可以用一个简单的零有效 one-hot 状态机(zero one-hot state machine)来描述。

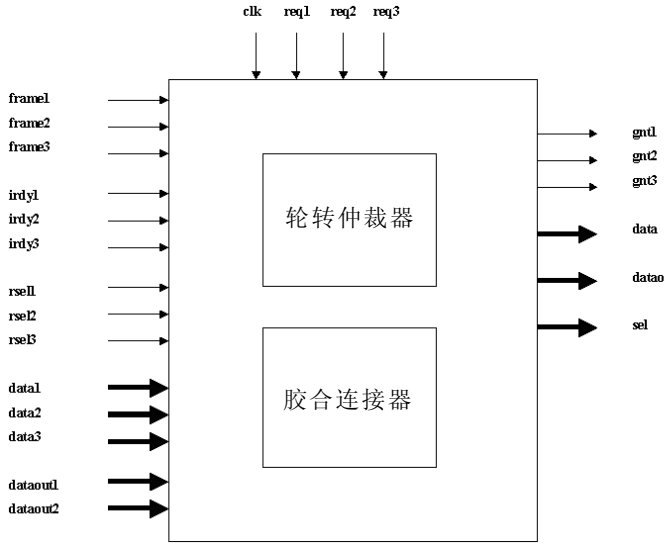

图 2-5 中间设备实例

主控设备选定要与其建立事务的目标设备后，中间设备将提供到指定的目标设备的信息。由于三个主控设备都能与任何一个目标设备建立事务，中间设备就不得不监控三个主控设备的“rsel”信号。在任何给定时间，要么所有三个“rsel”信号都处于三态，要么其中两个处于三态。如果所有三个“rsel”都是三态，那么在此时没有事务请求。如果有一个事务请求，那么其中的一个“rsel”信号的值为 0 或 1，具体的值依赖于选择的目标设备。如果信号“rsel”是 1，那么信号“sel”的最高位(MSB)置为高，表示目标设备 1 被选中，如果信号“rsel”是 0，那么信号“sel”的最低位(LSB)置为高，表示目标设备 0 被选中。

中间设备也为读和写事务选择正确的数据信号。如果是一个写事务，那么中间设备监控哪个主控设备的“rsel”信号是有效的，然后将这个主控设备的数据分配到选定的目标设备输入端。例如，如果主控设备 1 要对目标设备 0 做一个写操作，那么信号“rsel1”将置为低，并且总线“data1”将被分配到中间设备的输出总线“data”，这个输出将被导入选定的目标设备的输入端。在一个读事务中，中间设备则把来自目标设备的正确的输出数据分配到主

控设备。又比如一个读事务要从目标设备 1 读出数据，那么总线“dataout1”将被分配到总线“datao”。图 2-6 显示了中间设备功能的实例波形。

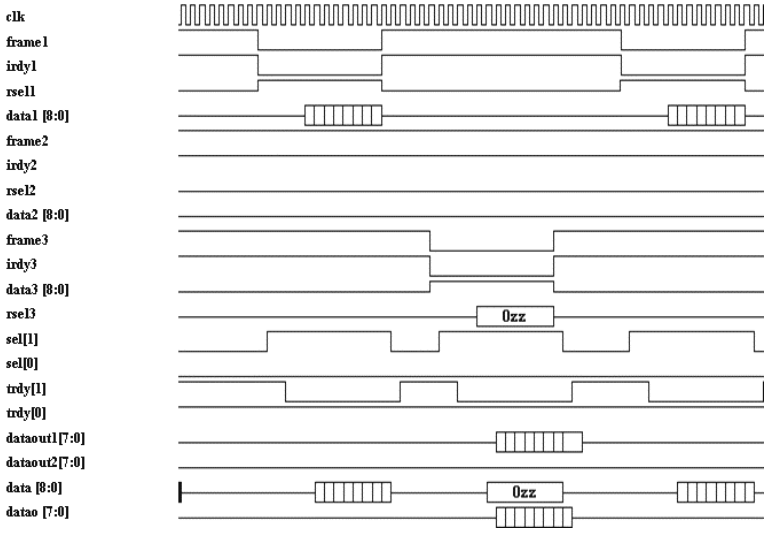

图 2-6 中间设备功能的波形

2.1.3 目标设备

图 2-7 显示了有输入输出端口的目标设备的方框图。目标设备有一个先进先出(FIFO)的内存，可以存储最多 64 字节的数据。

图 2-7 目标设备的实例

目标设备等待信号“sel_bit”的断言。一旦信号“sel_bit”被断言，目标设备要在两个时钟周期后通过断言信号“trdy”来确认它。断言了信号“trdy”后，如果是一个写事务，目标设备等待一

个有效的数据和有效的写信号。一旦发现一个有效的写信号，输入的数据从目标设备的写指针寄存器(wi)的最新更新的值所指向的地址开始存储。如果是一个读事务，那么目标设备从当前读指针(ri)指向的内存的位置开始读出 8 个数据。

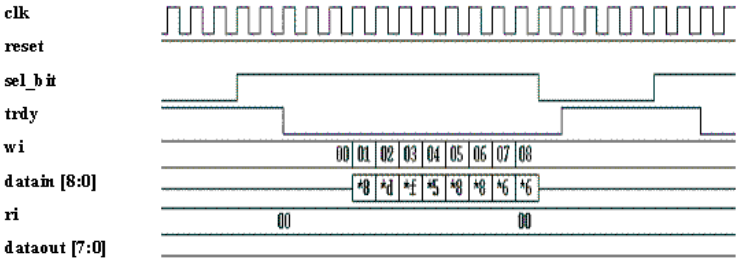

图 2-8 目标设备的写事务

事务的类型由总线“datain”的最高位(MSB)来表示。在一个读事务中，读的数据出现在总线向量“dataout”中。当事务完成，解除对信号“sel_bit”的断言，一个时钟周期后，解除对信号“trdy”的断言。图 2-8 显示了目标设备写操作的实例波形。图 2-9 显示了目标设备读操作的实例波形。

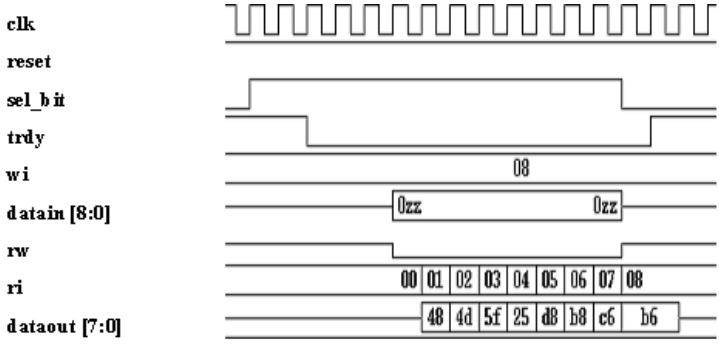

图 2-9 目标设备的读事务

2.2 块级验证

当每个独立设计模块完成后，应该对它们进行彻底的测试。块的穷举验证能提前发现边缘缺陷(corner case bugs)。在系统集成

前必须发现这些缺陷，因为在系统级发现这些缺陷是非常困难的。而且，系统级的出错很难识别定位，在系统级调试边缘缺陷是一项巨大的挑战。SVA 可以很容易地用来有效地测试独立设计块。在模块级，模拟工作量比较少，因此缺陷容易被跟踪并快速修复。在实例系统中有四个独立的设计块需要验证：

(1) 主控设备
(2) 目标设备
(3) 仲裁器
(4) 胶合(Glue)

还有两个需要彻底测试的块级接口：

(1) 主控设备和中间设备
(2) 目标设备和中间设备

2.2.1 SVA 在设计块中的应用

在用 SVA 进行块级验证中推荐下面的一些技巧：

- 所有为块级设计而写的 SVA 检验都应该是内嵌的。块级断言经常要访问设计的内部寄存器，因此把检验内嵌到设计模块更为有效率。
- 模块级的 SVA 检验的引用应该由设计模块内定义的参数来控制。这样，根据模拟的不同需求，可以自由地打开或关闭这些检验。
- 模块级的 SVA 检验的严重级别应该由设计模块内定义的参数来控制。SVA 里默认的严重级别将打印一个出错信息，然后继续模拟。
- 每个模块级的 SVA 检验都应该被执行并覆盖到。所有模块级检验一定要至少有一次真正的成功，这是必需的。

2.2.2 仲裁器的验证

基于 2.1.2 节描述的仲裁器的算法，可以析取出下面的 SVA 检验。在仲裁器的检验中重复使用的一些通用表达式，可以用 assign 语句做如下的定义：

```
assign frame = frame1 && frame2 && frame3;
assign irdy = irdy1 && irdy2 && irdy3;
assign gnt = !gnt1 || !gnt2 || !gnt3;
assign req = !req1 || !req2 || !req3;
```

信号“frame”和“irdy”都是低电平有效信号。每个主控设备有唯一的一个“frame”和“irdy”信号，它们是仲裁器模块的输入。如果主控设备在激活状态，它把信号“frame”和“irdy”都置为低。因此，把几个“frame”信号进行“与”操作，如果结果是低，则我们可以知道总线是激活的。同样地，把所有“irdy”信号进行“与”操作，如果结果是低，则我们可以知道总线是激活的。如果信号“frame”和“irdy”分别进行“与”操作的值都是高，则没有主控设备是激活的。

每个主控设备有唯一的一个“req”信号，用来请求总线，而且仲裁器提供唯一的“gnt”信号。通过把所有的“req”信号进行“或”操作，我们能知道即使只有一个主控设备有有效的请求，仲裁器也会判断这个请求。同样地，通过把“gnt”信号进行“或”操作，我们可以知道一个主控设备获得许可。创建这样的中间表达式使 SVA 检验器具有更好的可读性。

Arb_chk1：在任何给定的时钟边缘，仲裁器的内部状态变化应该是一个零有效“one-hot”状态机。

```
property p_arb_onehot0;
    @(posedge clk) $onehot0(state);
endproperty
```

Arb_chk2：一旦主控设备有一个有效的请求，仲裁器应该在2~5 个时钟周期内提供一个许可。

```
property p_req_gnt;
    @(posedge clk) $rose (req) |->
                  ##[2:5] $rose (gnt);
endproperty
```

Arb_chk3：一旦授予了许可，主控设备应该在同一时钟周期内通过断言“frame”和“irdy”信号来确认它接受这个许可。

```
property p_gnt_frame;
    @(posedge clk) $rose (gnt) |->
                   $fell (frame && irdy);
endproperty
```

Arb_chk4：一旦主控设备完成这个事务，它要解除对信号“frame”和“irdy”的断言，接着，仲裁器要在下一时钟周期解除对“gnt”信号的断言。

```
property p_frame_gnt;
    @(posedge clk) $rose(frame && irdy)
                                |=> $fell (gnt);
endproperty
```

2.2.3 模拟中针对仲裁器的 SVA 检验

2.2.2 节中所示的四个检验应该内嵌到仲裁器模块。根据需要，应该有一个规则来开关这些特性的检验，以及为这些特性设置严重等级。下面这些代码显示它是怎么实现的。

```
module arbiter(…..);

// port declarations

parameter arb_sva = 1'b1;
parameter arb_sva_severity = 1'b1;

// Arbiter design description
// SVA property description

// SVA Checks

always@(posedge clk)
begin
if(arb_sva)
begin

a_arb_onehot0:
     assert property(p_arb_onehot0)
     else if(arb_sva_severity) $fatal;
```

```
a_req_gnt:
      assert property(p_req_gnt)
      else if(arb_sva_severity) $fatal;

a_gnt_frame :
      assert property(p_gnt_frame)
      else if(arb_sva_severity) $fatal;

a_frame_gnt:

      assert property(p_frame_gnt)
      else if(arb_sva_severity) $fatal;

c_arb_onehot0: cover property(p_arb_onehot0);
c_req_gnt: cover property(p_req_gnt);
c_gnt_frame: cover property(p_gnt_frame);
c_frame_gnt: cover property(p_frame_gnt);

end
end

endmodule
```

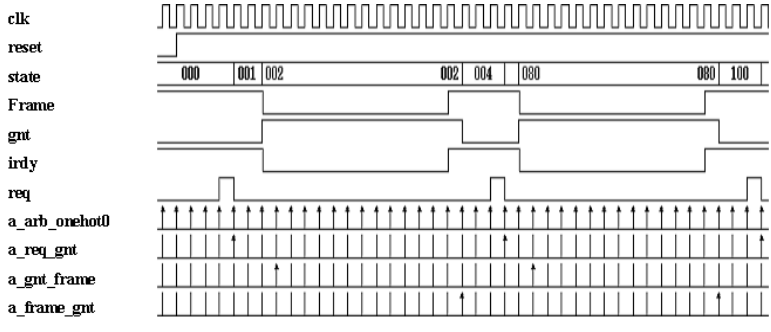

图 2-10 模拟中仲裁器的检验

要把检验包括进一个模拟中，参数“arb_sva”要设为 1。参数“arb_sva_severity”控制模拟中所做的动作。在这个例子里，如果参数设为 1，则严重度被设为$fatal。这意味着，只要任何检验失败一次，就要退出模拟。把参数设为 0，检验会使用默认的设定，

即当失败时打印出错信息，然后继续模拟。图 2-10 显示的是实例模拟的一个波形。

2.2.4 主控设备的验证

基于 2.1.1 节中描述的主控设备的协议，可以析取出下面的 SVA 检验。注意每个主控设备只有一个“req”、“gnt”、“frame”和“irdy”信号。在主控设备的检验器中提到的这些信号并不是在仲裁器检验中定义的表达式中的信号。它们只是出现在每个主控设备中的单独的信号。

Master_chk1：一旦主控设备有一个有效的请求，许可信号要在 2~5 个时钟内到达。如果是这样，并且信号“r_sel”为高，那么在同一时钟周期，主控设备应该断言信号“frame”和“irdy”。三个时钟周期后，目标设备 1 应该通过断言信号“trdy”确认这个选择。

```
property p_master_start1;
@(posedge clk)
  ($fell (req) ##[2:5] ($fell(gnt) && r_sel)) |->
                  (!frame && !irdy) ##3 !trdy[1];
endproperty
```

Master_chk2：一旦主控设备有一个有效的请求，许可信号要在 2~5 个时钟周期内到达。如果是这样，并且信号“r_sel”为低，那么在同一时钟周期内，主控设备应该断言信号“frame”和“irdy”。三个时钟周期后，目标设备 0 应该通过断言信号“trdy”确认这个选择。

```
property p_master_start2;
    @(posedge clk)
 ($fell (req) ##[2:5] ($fell(gnt) && !r_sel))|->
            (!frame && !irdy) ##3 !trdy[0];
endproperty
```

Master_chk3：一旦目标设备确认了它的选择，主控设备应该在 10 个时钟周期内完成这个事务。通过解除对信号“frame”和“irdy” 的断言表示这个事务完成。一个时钟周期后，应该解除

对信号“gnt”的断言。

```
property p_master_stop1;
    @(posedge clk)
$fell (trdy[1]) |-> ##10 (frame && irdy) ##1 gnt;
endproperty

property p_master_stop2;
    @(posedge clk)
$fell (trdy[0]) |-> ##10 (frame && irdy) ##1 gnt;
endproperty
```

注意，我们用两个独立的属性来检验这个事务是否完成，每个目标设备各有一个。

Master_chk4：如果主控设备正处在一个写事务中，那么总线的数据(data_c)不应该是三态，应该是有效的数据。

```
property p_master_data1;
    @(posedge clk)
     ($fell (trdy[1]) ##2 rw) |->
          ($isunknown (data) == 0) [*7];
endproperty

property p_master_data2;
    @(posedge clk)
     ($fell (trdy[0]) ##2 rw) |->
         ($isunknown (data) == 0) [*7];
endproperty
```

- 注意，这两个独立的属性分别对应一个目标设备，检验写事务中数据的正确性。
- 注意，如果信号“rw”为高，那么主控设备正在操作一个写事务。

Master_chk5：如果主控设备处在一个读事务，那么总线的数据(data_o)不应该是三态，应该是有效的数据。

```
property p_master_datao1;
    @(posedge clk)
       ($fell (trdy[1]) ##3 !rw) |=>
            ($isunknown (data_o) == 0) [*7];
```

```
endproperty

property p_master_datao2;
    @(posedge clk)
       ($fell (trdy[0]) ##3 !rw) |=>
          ($isunknown (data_o) == 0) [*7];
endproperty
```

- 注意，这两个独立的属性分别对应一个目标设备，检验读事务中数据的正确性。
- 注意，如果信号“rw”为低，那么主控设备正在操作一个读事务。

2.2.5 模拟中针对主控设备的 SVA 检验

2.2.4 节中所示的五个检验应该内嵌到主控设备模块。根据需要，应该有一个规则来控制这些属性，下面这些代码显示它是怎么实现的。

```
module master(…..);

// port declarations

parameter master_sva = 1'b1;
parameter master_sva_severity = 1'b1;

// Master design description

// SVA property description

// SVA Checks

always@(posedge clk)

begin

if(master_sva)

begin

a_master_start1:
     assert property(p_master_start1)
     else if(master_sva_severity) $fatal;
```

```
a_master_start2:
     assert property(p_master_start2)
     else if(master_sva_severity) $fatal;

a_master_stop1:
     assert property(p_master_stop1)
     else if(master_sva_severity) $fatal;

a_master_stop2:
     assert property(p_master_stop2)
     else if(master_sva_severity) $fatal;

a_master_data1:
     assert property(p_master_data1)
     else if(master_sva_severity) $fatal;

a_master_data2:
     assert property(p_master_data2)
     else if(master_sva_severity) $fatal;

a_master_datao1:
     assert property(p_master_datao1)
     else if(master_sva_severity) $fatal;

a_master_datao2:
     assert property(p_master_datao2)
     else if(master_sva_severity) $fatal;

c_master_start1: cover property(p_master_start1);
c_master_start2: cover property(p_master_start2);
c_master_stop1: cover property(p_master_stop1);
c_master_stop2: cover property(p_master_stop2);
c_master_data1: cover property(p_master_data1);
c_master_data2: cover property(p_master_data2);
c_master_datao1: cover property(p_master_datao1);
c_master_datao2: cover property(p_master_datao2);

end

end

endmodule
```

图 2-11 显示的是这些主控设备检验的模拟波形的实例。

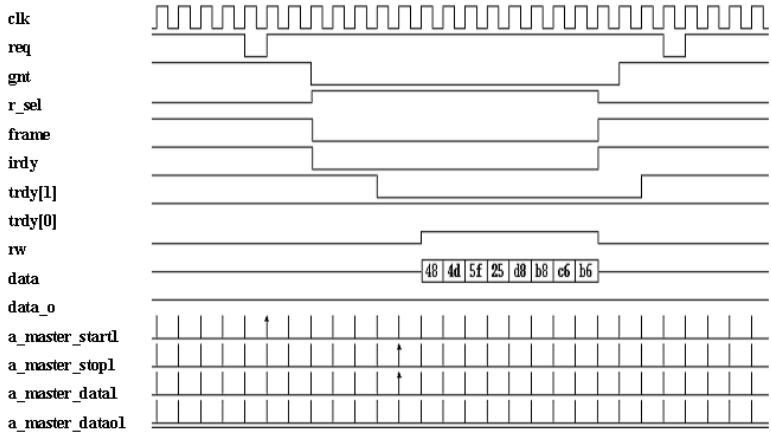

图 2-11 模拟中对目标设备 1 的主控设备的检验

2.2.6 胶合(Glue)的验证

基于 2.1.2 节描述的胶合逻辑(glue logic)的协议，可以析取出下面的 SVA 检验。

Glue_chk1：如果任何一个主控设备的选择信号“sel1”、“sel2”和“sel3”为高，那么目标设备 1 被选中。

```
property p_sel_1;
  @(posedge clk)
    (rsel1 || rsel2 || rsel3) |=> sel == 2'b10;
endproperty
```

Glue_chk2：如果任何一个主控设备的选择信号“sel1”、“sel2”和“sel3”为低，那么目标设备 0 被选中。

```
property p_sel_0;
    @(posedge clk)
    (!rsel1 || !rsel2 || !rsel3) |=> sel == 2'b01;
endproperty
```

Glue_chk3：在一个写事务中，如果信号“rsel1”不是三态，那么从主控设备 1 来的数据应该写入相应的目标设备。

```
property p_rsel1_write;
    @(posedge clk)
((rsel1 || !rsel1) ##3 ($fell (trdy[1]) ||
$fell(trdy[0])) ##3 data1[8]) |->
        (data == $past(data1)) [*7];
endproperty
```

- 注意，通过使用总线数据的最高位，我们可以确定事务的种类(读/写)。
- 如果总线数据的最高位是高，那么这是一个写事务。
- 如果总线数据的最高位是低，那么这是一个读事务。
- 在主控设备内，事务的种类由信号“rw”来确定。这个信号复制了总线数据的最高位。信号“rw”是主控设备的局部信号。外部接口应该由总线数据的最高位推断出事务的种类。

Glue_chk4：在一个写事务中，如果信号“rsel2”不是三态，那么从主控设备 2 来的数据应该写到对应的目标设备。

```
property p_rsel2_write;
    @(posedge clk)
((rsel2 || !rsel2) ##3 ($fell (trdy[1]) ||
$fell(trdy[0])) ##3 data2[8]) |->
        (data == $past(data2)) [*7];
endproperty
```

Glue_chk5：在一个写事务中，如果信号“rsel3”不是三态，那么从主控设备 3 来的数据应该写到对应的目标设备。

```
property p_rsel3_write;
    @(posedge clk)
((rsel3 || !rsel3) ##3 ($fell (trdy[1]) ||
$fell(trdy[0])) ##3 data3[8]) |->
          (data == $past(data3)) [*7];
endproperty
```

Glue_chk6：在一个读事务中，如果目标设备 1 被选中，从目标设备 1(dataout1)中读出的数据应该读入到对应的主控设备。

```
property p_read1;
    @(posedge clk)
($fell (trdy[1]) ##4 !data[8]) |->
                (dataout1 == datao) [*7];
endproperty
```

Glue_chk7：在一个读事务中，如果目标设备 0 被选中，从目标设备 0(dataout2)中读出的数据应该读入到对应的主控设备。

```
property p_read0;
    @(posedge clk)
($fell (trdy[0]) ##4 !data[8]) |->
                (dataout2 == datao) [*7];
endproperty
```

2.2.7 模拟中针对胶合逻辑(glue logic)的 SVA 检验

2.2.6 节中所示的七个检验应该内嵌到胶合模块内。根据需要，应该有一个规则来控制这些属性。下面这些代码显示它是怎么实现的。

```
module glue(…..);

// port declarations

parameter glue_sva = 1'b1;
parameter glue_sva_severity = 1'b1;

// glue design description
// glue SVA property description

// SVA Checks

always@(posedge clk)
begin
if(glue_sva)
begin

a_sel_1:
      assert property(p_sel_1)
            else if(glue_sva_severity) $fatal;
```

```
a_sel_0:
      assert property(p_sel_0)
            else if(glue_sva_severity) $fatal;

a_rsel1_write:
      assert property(p_rsel1_write)
            else if(glue_sva_severity) $fatal;

a_rsel2_write:
      assert property(p_rsel2_write)
            else if(glue_sva_severity) $fatal;

a_rsel3_write:
      assert property(p_rsel3_write)
            else if(glue_sva_severity) $fatal;

a_read1:
      assert property(p_read1)
            else if(glue_sva_severity) $fatal;

a_read0:
      assert property(p_read0)
            else if(glue_sva_severity) $fatal;

c_sel_1: cover property(p_sel_1);
c_sel_0: cover property(p_sel_0);
c_rsel1_write: cover property(p_rsel1_write);
c_rsel2_write: cover property(p_rsel2_write);
c_rsel3_write: cover property(p_rsel3_write);
c_read1: cover property(p_read1);
c_read0: cover property(p_read0);

end
end

endmodule
```

图 2-12 显示的是胶合检验的模拟波形的实例。

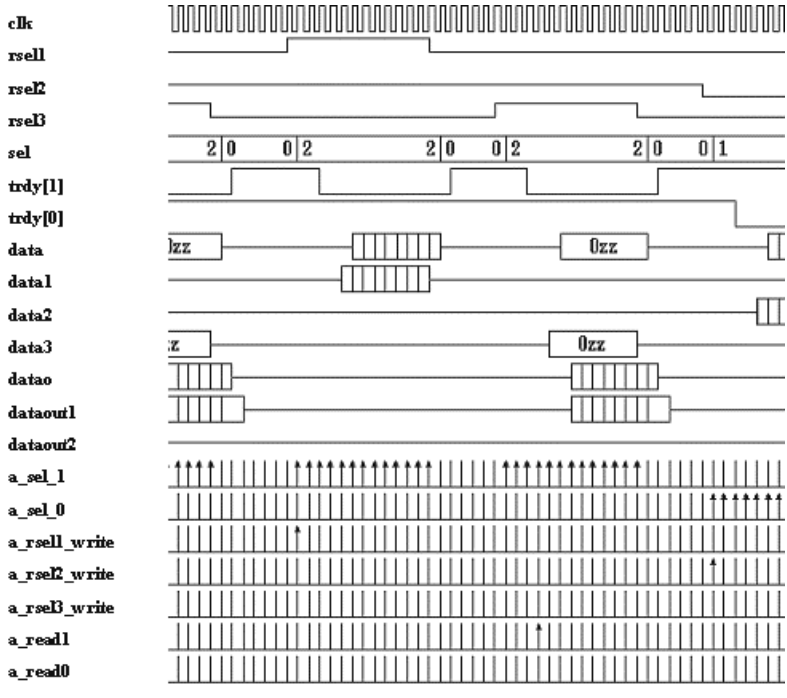

图 2-12 模拟中的胶合检验

2.2.8 目标设备的验证

基于 2.1.3 节中描述的目标设备的协议，可以析取出下面的 SVA 检验。

Target_chk1：如果选中一个目标设备，那么它应该在两个时钟周期后断言信号“trdy”。

```
property p_sel_trdy_start;
    @(posedge clk) $rose (sel_bit) |->
                          ##1 trdy ##1 !trdy;
endproperty
```

Target_chk2：在一个事务结束时，信号“sel_bit”被解除断言。一个时钟周期后，信号“trdy”应该被解除断言。

```
property p_sel_trdy_stop;
    @(posedge clk) $fell (sel_bit) |=> trdy;
endproperty
```

Target_chk3：在一个写事务中，每次在一个时钟周期后写指针应该加 1，以保证每次有效地写入一个唯一的地址。

```
property p_write;
    @(posedge clk)
(datain[8] && sel_bit && (wi != 63)) |->
                        (wi == ($past(wi) + 1));
endproperty
```

- 注意，地址指针累加会从 63 回到 0，再从 0 开始累加。因此，如果在一个给定的时钟边缘写指针在 63 的位置，这个检验不适用。
- 可以写一个不同的检验来验证指针总是会正确地从 63 回到 0。

Target_chk4：在一个读事务中，每次在一个时钟周期内从唯一的地址读出有效的数据完成后，读指针应该加 1。

```
property p_read;
    @(posedge clk)
(!datain[8] && sel_bit && (ri != 63)) |=>
                      (ri == ($past (ri) + 1));
endproperty
```

- 注意，对于读指针，当指针在 63 的位置，这个检验不适用。
- 读操作有一个时钟周期的延时，因此我们用非重叠蕴含操作符。
- 由于采用非重叠操作符，检验往后移了一个周期，与前一周期的地址作比较。
- 例如，在一个给定的时钟边缘，如果蕴含表达式的先行算子为真，则检验移到下一时钟周期。当指针在 63，如果指针加 1，那么检验会移到指针 0，然后比较 63 和 0 之间的加 1 关系。这是不正确的。因此，在一个给定的时钟边缘，如果读指针的值为 63，不能执行这个检验。
- 可以写一个独立的检验来确认指针能从 63 回到 0。

Target_chk5：在一个有效的读或写的事务中，从目标设备读出或写入的数据应该是有效的。

```
property p_target_datain;
    @(posedge clk)
  ($fell (trdy) ##3 (datain[8])) |->
            not ($isunknown (datain)) [*7];
endproperty

property p_target_dataout;
    @(posedge clk)
 ($fell (trdy) ##3 (!datain[8])) |=>
          not (($isunknown(dataout)) [*7]);
endproperty
```

2.2.9 模拟中针对目标设备的 SVA 检验

2.2.8 节中所示的五个检验应该内嵌到目标设备的模块内。根据需要，应该有一个规则来控制这些属性。下面这些代码显示它是怎么实现的。

```
module target(…..);

// port declarations

parameter target_sva = 1'b1;
parameter target_sva_severity = 1'b1;

// target design description
// target SVA property description
// SVA Checks

always@(posedge clk)
begin
if(target_sva)
begin

a_sel_trdy_start:
    assert property(p_sel_trdy_start)
    else    if(target_sva_severity) $fatal;
a_sel_trdy_stop:
     assert property(p_sel_trdy_stop)
     else    if(target_sva_severity) $fatal;

a_write:
      assert property(p_write)
```

```
        else if(target_sva_severity) $fatal;

a_read:
        assert property(p_read)
        else if(target_sva_severity) $fatal;

a_target_datain:
        assert property(p_target_datain)
        else if(target_sva_severity) $fatal;

a_target_dataout:
        assert property(p_target_dataout)
        else if(target_sva_severity) $fatal;

c_sel_trdy_start:
                cover property(p_sel_trdy_start);
c_sel_trdy_stop: cover property(p_sel_trdy_stop);
c_write: cover property(p_write);
c_read: cover property(p_read);
c_target_datain: cover property(p_target_datain);
c_target_dataout: cover property(p_target_dataout);


end
end
endmodule
```

图 2-13 所示的是针对目标设备的检验做的模拟波形的例子。

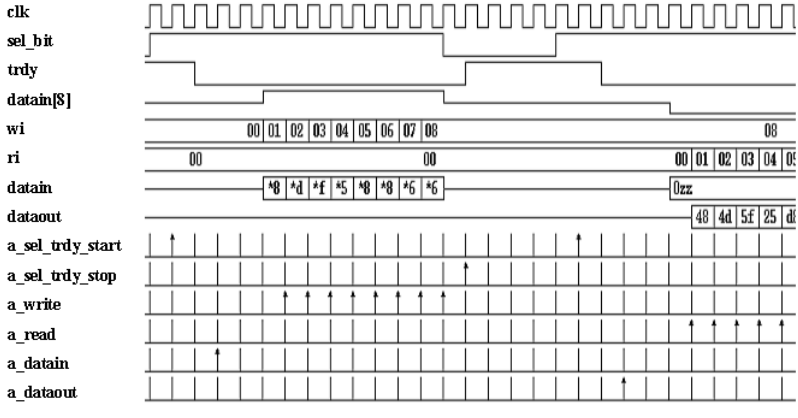

图 2-13 模拟中针对目标设备的检验

2.3 系统级验证

在这个系统中，一个中间设备实例连着三个主控设备和两个目标设备。系统的顶层连接如下所示。

```
module top(.., ..,    );

// port declarations

master u1 (ask[2], clk, req1, gnt1, frame1,
irdy1, trdy, data1, rsel1, datao);

master u2 (ask[1], clk, req2, gnt2, frame2,
irdy2, trdy, data2, rsel2, datao);

master u3 (ask[0], clk, req3, gnt3, frame3,
irdy3, trdy, data3, rsel3, datao);

arbiter u4(clk, reset, frame, irdy, req1, req2,
req3, gnt1, gnt2, gnt3);

glue u5 (clk, frame1, irdy1, frame2, irdy2,
frame3, irdy3, trdy, rsel1, rsel2, rsel3, data1,
data2, data3, sel, data, dataout1, dataout2,
datao);

target u6 (clk, reset, sel[1], trdy[1], data,
dataout1);

target u7 (clk, reset, sel[0], trdy[0], data,
dataout2);

endmodule
```

在用 SVA 进行系统级验证时推荐下面一些技巧：

- 由于单个块的内部功能已经仔细验证过，所以默认状态下，在系统级验证时不把块级断言包含进去。这么做主要是为了提高性能。

- 如果性能不是瓶颈，默认情况下，在系统级验证中把块级断言包含进去。系统接口提供了更加真实的和一些期望之外的输入情况，而模块级断言要能对此作出正确的反应。
- 如果有断言失败，验证环境要提供机制来激活模块级断言。例如，在我们的实例系统中，如果主控设备 1 和目标设备 0 之间的事务出错，把针对主控设备 1 和目标设备 0 写的块级 SVA 检验包括进去，重跑系统级模拟。
- 在系统级，要写一些新的断言来验证系统的连接，这些检查更应该关注接口规则而不是块内部细节。

针对系统级验证的 SVA 检验

基于系统的连接和协议，针对系统级验证可以写出如下一些检验。

Ss_shk1：在任何给定的时间点，只能断言一个“trdy”信号。换言之，在任何给定的时间，一个事务中只能有一个目标设备。

```
property p_target;
    @(posedge clk) not (!trdy[0] && !trdy[1]);
endproperty
```

Ss_chk2：在任何给定的时钟周期内，只能断言一对“frame”和“irdy”信号。换言之，在任何给定的时间，一个事务中只能有一个主控设备。

```
property p_frame;
    @(posedge clk)
      $countones({frame1, frame2, frame3}) >1;
endproperty

property p_irdy;
    @(posedge clk)
      $countones({irdy1, irdy2, irdy3}) >1;
endproperty
```

Ss_chk3：在任何给定的时间，只能断言一个“gnt”信号。换而言之，仲裁器一次只能让一个主控设备开始事务。

```
property p_gnt;
    @(posedge clk)
        $countones({gnt1, gnt2, gnt3}) > 1;
endproperty
```

Ss_chk4：在任何给定的时钟周期，只能激活一个“rw”信号，其他的“rw”信号应该是三态(“rw”信号是主控设备数据输出总线的最高位)。

```
property p_rw;
    @(posedge clk)
 ($isunknown(rw1)    &&    $isunknown(rw2)    &&
 $isunknown(rw3) ) ||
 ((rw1==1'b1 || rw1==1'b0) && $isunknown (rw2) &&
 $isunknown(rw3)) ||
 ((rw2==1'b1 || rw2==1'b0) && $isunknown (rw1) &&
 $isunknown(rw3)) ||
 ((rw3==1'b1 || rw3==1'b0) && $isunknown (rw2) &&
 $isunknown(rw2));
endproperty
```

Ss_chk5：在任何给定的时钟周期，只能激活一个“rsel”信号，其他“rsel”信号应该是三态。

```
property p_rsel;
    @(posedge clk)
$isunknown(rsel1)   &&   $isunknown(rsel2)   &&
$isunknown(rsel3) ) ||
  ((rsel1==1'b1 || rsel1==1'b0) && $isunknown (rsel2)
&& $isunknown(rsel3)) ||
  ((rsel2==1'b1 || rsel2==1'b0) && $isunknown (rsel1)
&& $isunknown(rsel3)) ||
  ((rsel3==1'b1 || rsel3==1'b0) && $isunknown (rsel2)
&& $isunknown(rsel1));
 endproperty
```

Ss_chk6：一旦一个主控设备发出一个正确的请求，在 2~5 个时钟周期内，一个正确的“gnt”应该到达。

```
assign req = !req1 || !req2 || !req3;
assign gnt = !gnt1 || !gnt2 || !gnt3;

property p_req_gnt_w;
```

```
    @(posedge clk)
        $rose (req) |-> ##[2:5] $rose(gnt);
endproperty
```

Ss_chk7：在任何给定的时钟，如果一个主控设备的“frame”和“irdy”信号被断言了，那么 3 个时钟周期后，相关的“trdy”信号应该被断言。

```
assign frame_ = !frame1 || !frame2 || !frame3;
assign irdy_ = !irdy1 || !irdy2 || !irdy3;

property p_start_frame;
    @(posedge clk)
      $rose(frame_ && irdy_)|->##3 $rose(trdy_);
endproperty
```

Ss_chk8：在任何给定的时钟，如果主控设备的“frame”和“irdy”信号被解除断言，那么两个时钟周期后，相关的“trdy”信号应该被解除断言。

```
assign trdyp = trdy[1] && trdy[0];

property p_end_frame;
    @(posedge clk)
        $rose(frame && irdy)|->##2 $rose(trdyp);
endproperty
```

Ss_chk9：在任何给定的时钟，如果没有有效的事务，那么总线“data”和“datao”应该是三态。

```
property p_bus_not_in_use;
    @(posedge clk)
  trdyp |->
        ($isunknown(data) && $isunknown(datao));
endproperty

a_target : assert property(p_target);
a_frame: assert property(p_frame);
a_irdy: assert property(p_irdy);
a_rsel: assert property(p_rsel);
a_rw: assert property(p_rw);
a_gnt: assert property(p_gnt);
a_req_gnt_w : assert property(p_req_gnt_w);
```

```
a_start_frame: assert property(p_start_frame);
a_end_frame: assert property(p_end_frame);
a_bus_in_use: assert property(p_bus_not_in_use);

c_target : cover property(p_target);
c_frame: cover property(p_frame);
c_irdy: cover property(p_irdy);
c_rsel: cover property(p_rsel);
c_rw: cover property(p_rw);
c_gnt: cover property(p_gnt);
c_req_gnt_w : cover property(p_req_gnt_w);
c_start_frame: cover property(p_start_frame);
c_end_frame: cover property(p_end_frame);
c_bus_in_use: cover property(p_bus_not_in_use);
```

在系统级模拟中，最顶层的模块应该配置参数设定，这样可以关闭所有块级断言。在我们的实例系统中，由于每个设计块都有一个参数，如果需要可以把相关的 SVA 检验包括进来。如下所示，我们可以轻松地配置系统级的顶层模块。

```
module top(.., ..,    );

// port declarations

master
#(.master_sva(1'b0), .master_sva_severity(1'b0))
u1(ask[2], clk, req1, gnt1, frame1, irdy1, trdy,
data1, rsel1, datao);

master
#(.master_sva(1'b0), .master_sva_severity(1'b0))
u2(ask[1], clk, req2, gnt2, frame2, irdy2, trdy,
data2, rsel2, datao);

master
#(.master_sva(1'b0), .master_sva_severity(1'b0))
u3(ask[0], clk, req3, gnt3, frame3, irdy3, trdy,
data3, rsel3, datao);

arbiter
#(.arb_sva(1'b0), .arb_sva_severity(1'b0))
u4(clk, reset, frame, irdy, req1, req2, req3,
```

```
gnt1, gnt2, gnt3);

glue
#(.glue_sva(1'b0), .glue_sva_severity(1'b0))
u5(clk, frame1, irdy1, frame2, irdy2, frame3,


irdy3, trdy, rsel1, rsel2, rsel3, data1, data2,
data3, sel, data, dataout1, dataout2, datao);

target
#(.target_sva(1'b0), .target_sva_severity(1'b0))
u6(clk, reset, sel[1], trdy[1], data, dataout1);

target
#(.target_sva(1'b0), .target_sva_severity(1'b0))
u7(clk, reset, sel[0], trdy[0], data, dataout2);
endmodule
```

注意，每个设计块实例化时，都传入了参数的值。在每个实例化中，第一个参数“*_sva”都被设为 0，这就表示块级断言将不被执行，系统级模拟只做系统级的检验。

假设在系统级模拟时“Ss_chk6”报告了一些失败。这个检验是查找主控设备和仲裁器之间是否存在接口错误。要调试这类错误，可以重新运行模拟并把主控设备和仲裁器相关的块级检验包括进去。这个模拟的顶层模块的配置如下所示：

```
module top(.., ..,    );

// port declarations

master
#(.master_sva(1'b1), .master_sva_severity(1'b0))
u1(ask[2], clk, req1, gnt1, frame1, irdy1, trdy,
data1, rsel1, datao);

master
#(.master_sva(1'b1), .master_sva_severity(1'b0))
u2(ask[1], clk, req2, gnt2, frame2, irdy2, trdy,
data2, rsel2, datao);

master
```

```
#(.master_sva(1'b1), .master_sva_severity(1'b0))
u3(ask[0], clk, req3, gnt3, frame3, irdy3, trdy,
data3, rsel3, datao);

arbiter
#(.arb_sva(1'b1), .arb_sva_severity(1'b0))
u4(clk, reset, frame, irdy, req1, req2, req3,
gnt1, gnt2, gnt3);

glue
#(.glue_sva(1'b0), .glue_sva_severity(1'b0))
u5(clk, frame1, irdy1, frame2, irdy2, frame3,
irdy3, trdy, rsel1, rsel2, rsel3, data1, data2,
data3, sel, data, dataout1, dataout2, datao);

target
#(.target_sva(1'b0), .target_sva_severity(1'b0))
u6(clk, reset, sel[1], trdy[1], data, dataout1);

target
#(.target_sva(1'b0), .target_sva_severity(1'b0))
u7(clk, reset, sel[0], trdy[0], data, dataout2);

endmodule
```

注意，这个配置中，参数“master_sva”和“arb_sva”被设为1。在基本的设计块中，用“ ifdef - endif”结构可以有条件地把SVA检验包括进去。通过条件编译SVA的代码，用户可以在模块的所有实例中做这些检验，也可以不做这些检验。这个方法的不足之处是它是一个全局控制机制，会影响所有的模块级检验。通过使用参数，这个不足可以被克服，用户在模拟中可以更加灵活地选择需要的块级检验。

2.4 功能覆盖

到目前为止，编写的系统级检验寻求的是任何可能存在的违反指定的协议的情况。模拟中，确信这些检验至少被执行一次，很大程度上提高了系统功能的可信等级。功能覆盖的另一方面是

从测试平台角度考虑，在模拟中覆盖系统功能的所有可能的情景。在模拟中需要覆盖的情景应该成为测试计划的一部分。

为动态模拟写的 SVA 检验的效果仍然依赖于输入激励。如果输入向量不促使系统执行某些情景，那么它们就不会被测试到。许多测试平台采用随机技术产生模拟的输入激励。一个非常通用的方法是运行许多预先定义的事务，然后测量对某些情景集合的覆盖。通过约束控制输入激励的随机产生，会更加有效地覆盖到各种情景。关键是在最少数量的周期内，达到最大的功能覆盖。从 SVA 收集来的覆盖信息能有效地用来建立反应验证环境(reactive verification environments)。

2.4.1 实例系统的覆盖率计划

本章中讨论的实例系统有很多重要的功能，应该作为功能验证的一部分被覆盖到。

1. 请求情景(Request Scenario)

“所有可能的请求情景都应该被覆盖到”。

在任何给定的时间，有三个主控设备可以请求访问。这就意味着主控设备的“req”信号有七种可能的组合，如表 2-1 所示。

表 2-1 主控设备请求情景

Req1	Req2	Req3
0	1	1
1	0	1
1	1	0
0	0	1
1	0	0
0	1	0
0	0	0

表中的 0 表示主控设备正在请求总线。测试平台应该在模拟中生成所有这些可能的输入组合。

下面的代码举例说明了如何用功能覆盖的数据来控制模拟环境。所有七种可能的请求组合的属性定义如下。

```
property p_req1; // master 1 requesting
    @(posedge clk) $fell (req1) && req2 && req3;
endproperty

property p_req2; // master 2 requesting
    @(posedge clk) $fell (req2) && req1 && req3;
endproperty

property p_req3; // master 3 requesting
    @(posedge clk) $fell (req3) && req1 && req2;
endproperty

property p_req12; // master 1&2 requesting
    @(posedge clk)
    $fell (req1) && $fell(req2)&& req3;
endproperty

property p_req23; // master 2&3 rtequesting
    @(posedge clk)
    $fell (req2) && $fell(req3) && req1;
endproperty

property p_req31; // master 1&3 requesting
    @(posedge clk)
    $fell (req3) && $fell(req1) && req2;
endproperty

property p_req123; // master 1&2&3 requesting
    @(posedge clk)
  $fell (req1) && $fell(req2) && $fell(req3);
endproperty
```

每个属性应该有一个相连的如下所示的覆盖(cover)语句。覆盖语句的执行块能用来更新寄存器的标识。对于这种情况，每次覆盖到属性，一个局部的寄存器计数就会加 1。在同一个时钟，我们检验计数器的值是否已经达到 3。如果是，那么与这个属性相连的标识要被断言。换言之，在模拟中，期望每种请求组合会出现三次，当出现三次时，一个与指定的请求组合相连的标识将被断言。

```
c_req1: cover property(p_req1)
        begin
            creq1++;
            if(creq1 == 3) creq1_flag = 1'b1;
        end

c_req2: cover property(p_req2)
        begin
            creq2++;
            if(creq2 == 3) creq2_flag = 1'b1;
        end

c_req3: cover property(p_req3)
        begin
            creq3++;
            if(creq3 == 3) creq3_flag = 1'b1;
        end
c_req12: cover property(p_req12)
        begin
            creq12++;
            if(creq12 == 3) creq12_flag = 1'b1;
        end

c_req23: cover property(p_req23)
        begin
            creq23++;
            if(creq23 == 3) creq23_flag = 1'b1;
        end

c_req31: cover property(p_req31)
        begin
            creq31++;
            if(creq31 == 3) creq31_flag = 1'b1;
        end

c_req123: cover property(p_req123)
        begin
            creq123++;
            if(creq123 == 3) creq123_flag = 1'b1;
        end
```

这种覆盖信息可以用来有效地控制模拟环境。在实例系统的随机测试平台中，预先确定的许多事务一个接一个地执行，当所

有的事务完成了，模拟也将结束。下面的代码显示如何利用功能覆盖的信息来终止一个模拟。

```
always@(posedge clk)
begin

if(creq1_flag && creq2_flag && creq3_flag &&
creq12_flag && creq23_flag && creq31_flag &&
creq123_flag)

begin

    $display("FC: All possible request scenarios
    covered 3 times each\n");
    $finish();

end
end
```

在这段代码中，有两种方法来终止一个模拟：

(1) 随机运行完给定数量的事务，然后退出。

(2) 如果所有可能的请求情景每种都覆盖过三次，那么退出。

无论哪种情况先发生都将终止模拟。

2. 主控设备到目标设备的事务

每个主控设备应该对每个目标设备执行一个读事务和一次写事务。

系统中有三个主控设备和两个目标设备。这样就生成了 12 种可能的情景，如表 2-2 所示。所有 12 种可能的事务组合的属性定义生成如下所示。

表 2-2 主控设备到目标设备的事务

主控设备	目标设备	事务
M1	T1	读
M1	T1	写
M1	T0	读
M1	T0	写
M2	T1	读

(续表)

主 控 设 备	目 标 设 备	事 务
M2	T1	写
M2	T0	读
M2	T0	写
M3	T1	读
M3	T1	写
M3	T0	读
M3	T0	写

```
property p_m1t1r;
// master1 reading from target 1
  @(posedge clk)
  $fell (frame1 && irdy1) |->
         ##3 ($fell (trdy[1])) ##3 !data[8];
endproperty

property p_m1t1w;
// master 1 writing to target 1
  @(posedge clk)
  $fell (frame1 && irdy1) |->
         ##3 ($fell (trdy[1])) ##3 data[8];
endproperty

property p_m1t0r;
// master 1 reading from target 0
  @(posedge clk)
  $fell (frame1 && irdy1) |->
         ##3 ($fell (trdy[0])) ##3 !data[8];
endproperty

property p_m1t0w;
// master 1 writing to target 0
  @(posedge clk)
  $fell(frame1 && irdy1) |->
         ##3 ($fell(trdy[0])) ##3 data[8];
endproperty

property p_m2t1r;
// master 2 reading from target 1
  @(posedge clk)
```

```
  $fell (frame2 && irdy2) |->
        ##3 ($fell(trdy[1])) ##3 !data[8];
endproperty

property p_m2t1w;
// master 2 writing to target 1
  @(posedge clk)
  $fell (frame2 && irdy2) |->
        ##3 ($fell (trdy[1])) ##3 data[8];
endproperty

property p_m2t0r;
// master 2 reading from target 0
  @(posedge clk)
  $fell (frame2 && irdy2) |->
        ##3 ($felltrdy[0])) ##3 !data[8];
endproperty

property p_m2t0w;
// master 2 writing to target 0
  @(posedge clk)
  $fell (frame2 && irdy2) |->
        ##3 ($fell (trdy[0])) ##3 data[8];
endproperty

property p_m3t1r;
// master 3 reading from target 1
  @(posedge clk)
  $fell (frame3 && irdy3) |->
        ##3 ($fell (trdy[1])) ##3 !data[8];
endproperty

property p_m3t1w;
// master 3 writing to target 1
  @(posedge clk)
  $fell (frame3 && irdy3) |->
        ##3 ($fell (trdy[1])) ##3 data[8];
endproperty

property p_m3t0r;
// master 3 reading from target 0
  @(posedge clk)
  $fell (frame3 && irdy3) |->
        ##3 ($fell (trdy[0])) ##3 !data[8];
```

```
endproperty

property p_m3t0w;
// master 3 writing to target 0
  @(posedge clk)
  $fell (frame3 && irdy3) |->
         ##3 ($fell (trdy[0])) ##3 data[8];
endproperty
```

每个属性应该有一个与之相连的如下所示的覆盖语句。采用与前面“请求情景”中的相同技巧来计算情景出现的次数。

```
c_m1t1r: cover property(p_m1t1r)
        begin
            m1_t1_r++;
            if(m1_t1_r == 3) m1_t1_r_flag = 1'b1;
        end

c_m1t1w: cover property(p_m1t1w)
        begin
            m1_t1_w++;
            if(m1_t1_w == 3) m1_t1_w_flag = 1'b1;
        end

c_m1t0r: cover property(p_m1t0r)
        begin
            m1_t0_r++;
            if(m1_t0_r == 3) m1_t0_r_flag = 1'b1;
        end

c_m1t0w: cover property(p_m1t0w)
        begin
            m1_t0_w++;
            if(m1_t0_w == 3) m1_t0_w_flag = 1'b1;
        end

c_m2t1r: cover property(p_m2t1r)
        begin
            m2_t1_r++;
            if(m2_t1_r == 3) m2_t1_r_flag = 1'b1;
        end

c_m2t1w: cover property(p_m2t1w)
        begin
```

```
            m2_t1_w++;
            if(m2_t1_w == 3) m2_t1_w_flag = 1'b1;
        end

c_m2t0r: cover property(p_m2t0r)
        begin
            m2_t0_r++;
            if(m2_t0_r == 3) m2_t0_r_flag = 1'b1;
        end

c_m2t0w: cover property(p_m2t0w)
        begin
            m2_t0_w++;
            if(m2_t0_w == 3) m2_t0_w_flag = 1'b1;
        end

c_m3t1r: cover property(p_m3t1r)
        begin
            m3_t1_r++;
            if(m3_t1_r == 3) m3_t1_r_flag = 1'b1;
        end

c_m3t1w: cover property(p_m3t1w)
        begin
            m3_t1_w++;
            if(m3_t1_w == 3) m3_t1_w_flag = 1'b1;
        end

c_m3t0r: cover property(p_m3t0r)
        begin
            m3_t0_r++;
            if(m3_t0_r == 3) m3_t0_r_flag = 1'b1;
        end

c_m3t0w: cover property(p_m3t0w)
        begin
            m3_t0_w++;
            if(m3_t0_w == 3) m3_t0_w_flag = 1'b1;
        end
```

“请求情景”和“主控设备到目标设备的事务”中的覆盖信息能用来有效地控制模拟环境。在下面所示的这段代码中，有两种方法可以终止这个模拟：

(1) 随机运行完预先确定数量的事务，然后退出。

(2) 如果所有可能的请求情景被覆盖了三次和所有可能的主控设备到目标设备的事务覆盖了三次，那么退出这个模拟。

无论哪种情况先出现都将终止这个模拟。

```
always@(posedge clk)
begin

if(creq1_flag && creq2_flag && creq3_flag &&
creq12_flag && creq23_flag && creq31_flag &&
creq123_flag && m1_t1_r_flag && m1_t1_w_flag
&& m1_t0_r_flag && m1_t0_w_flag && m2_t1_r_flag
&& m2_t1_w_flag && m2_t0_r_flag && m2_t0_w_flag
&& m3_t1_r_flag && m3_t1_w_flag && m3_t0_r_flag
&& m3_t0_w_flag)

begin

    $display("FC: All possible request scenarios
    covered 3 times\n");

    $display("FC: All possible transactions
    covered 3 times\n");

    $finish();

end
end
```

3. 高级覆盖选项

有另一种数据点可以用来衡量系统的功能覆盖。

“目标设备的每个内存位置都应该被每个主控设备至少写入一次和从中读出一次”。

这个信息需要穷举测试才能获得。每个主控设备应该监控目标设备的每个地址空间的使用情况。在做功能覆盖时 SVA 并不是唯一的选择。如果功能覆盖涉及到穷举测试计划的覆盖点，那么使用支持面向对象编程的测试平台语言，效率会更高。当运行长

的回归测试时，应该使用这样的穷举功能覆盖点。

2.4.2 功能覆盖小结

功能覆盖的衡量保证了对所有必需检验的情景的测试。这种衡量能有效地用来控制模拟环境。一种方法是当达到功能覆盖的目标时终止模拟。在本节实例系统中，可以观察到下面的结果：

- 在测试平台中随机事务的数量默认设置为 500。
- 完成“请求情景”中所示的请求情景，只需要执行 46 个事务就终止了模拟。
- 完成“请求情景”中所示的请求情景和“主控设备到目标设备的事务”中所示的主控设备到目标设备的事务，只执行了 63 个事务就终止了模拟。

获得的功能覆盖的数据也能用来动态重定向测试平台。在随机的测试平台里，可以用一些约束来控制生成的事务的类型。在模拟开始时，为了实现随机分布，给这些约束分配了某些权重。在模拟中根据获得的功能覆盖的信息，动态调整这些权重，可以更快地达到功能覆盖的目标。

2.5 用于创建事务日志的 SVA

SVA 可以用来创建优秀的日志文件。模拟中，SVA 检验器侦听任何违背设计属性的情况。如果检验器能记录下它侦听到的信息，那么它也可以被称为“监视器”。在一个复杂的系统中，它对按照时间顺序生成事务的日志是非常有帮助的。在我们的实例系统中，SVA 生成一个关于所有读和写事务的日志，记录下事务在什么设备之间发生和何时发生，是非常重要的调试资源。

SVA 有一个选项可以在检验器范围内使用类似 Verilog 的功能。每个检验器的执行块或者覆盖语句能有效地创建日志文件。创建日志文件的一种方法是针对一个断言的成功或者一个覆盖语句显示信息，另外一种方法是调用一个任务(task)或函数(function)。调用一个任务或函数扩展了 SVA 检验器的功能。除了

在一个任务内显示信息外，也能有效地进行数据检验。下面的代码显示了这个实例系统的事务日志是如何按照时间顺序创建的。

```
// open a file to document transactions

integer h_mt;
initial
begin
    h_mt = $fopen("mt.dat");
end

// calling task for documentation

`ifdef slv_doc

c_m1t1w_doc:
    cover property(p_m1t1w) master_xaction(1, 1);
c_m1t1r_doc:
    cover property(p_m1t1r) master_xaction(1, 1);
c_m1t2w_doc:
    cover property(p_m1t0w) master_xaction(1, 0);
c_m1t2r_doc:
    cover property(p_m1t0r) master_xaction(1, 0);
c_m2t1w_doc:
    cover property(p_m2t1w) master_xaction(2, 1);
c_m2t1r_doc:
    cover property(p_m2t1r) master_xaction(2, 1);
c_m2t2w_doc:
    cover property(p_m2t0w) master_xaction(2, 0);
c_m2t2r_doc:
    cover property(p_m2t0r) master_xaction(2, 0);
c_m3t1w_doc:
    cover property(p_m3t1w) master_xaction(3, 1);
c_m3t1r_doc:
    cover property(p_m3t1r) master_xaction(3, 1);
c_m3t2w_doc:
    cover property(p_m3t0w) master_xaction(3, 0);
c_m3t2r_doc:
    cover property(p_m3t0r) master_xaction(3, 0);

`endif

task master_xaction(
```

```
  input int m_identity, input int t_identity);

integer i;

begin

if(data[8])
begin
    for(i=0; i<8; i++)
    begin

    $fwrite(h_mt, "WRITE:
Master %0d writing to Target %0d = %0d at
%0t\n", m_identity, t_identity, data[7:0], $time);

    @(posedge clk);
    end
    end

    if(!data[8])
    begin
    @(posedge clk);
    for(i=0; i<8; i++)
    begin
    $fwrite(h_mt, "READ:
    Master %0d reading from Target %0d = %0d at
    %0t\n", m_identity, t_identity, datao, $time);

    @(posedge clk);
    end
    end

    end

endtask
```

“主控设备到目标设备的事务”小节中定义的功能覆盖的属性被重用以创建事务日志。如果覆盖语句成功，调用一个名为“master_xaction”的任务。这个任务要有两个输入参数，一个用来确定主控设备，另一个用来确定目标设备。通过使用这些参数，可以创建一个通用的任务来精确记录事务。

事务被记录到一个独立的文件称为“mt.dat”。在模拟开始的

时候，用$fopen 语句可以打开这个文件。一旦调用这个任务，这个任务要么执行代码的读出模块或代码的写入模块。由于我们的实例系统按照 8 字节的整数倍进行一次读或写，所以在这个任务中用了一个“for”循环。它循环 8 次，每次循环对应的读或写的数据被$fwrite 语句记入到文件“mt.dat”中。下面是用这些代码创建的实例系统的日志的一部分。

```
WRITE: Master 1 writing to Target 1 = 72 at 775
WRITE: Master 1 writing to Target 1 = 77 at 825
WRITE: Master 1 writing to Target 1 = 95 at 875
WRITE: Master 1 writing to Target 1 = 37 at 925
WRITE: Master 1 writing to Target 1 = 216 at 975
WRITE: Master 1 writing to Target 1 = 184 at 1025
WRITE: Master 1 writing to Target 1 = 198 at 1075
WRITE: Master 1 writing to Target 1 = 182 at 1125
READ: Master 3 reading from Target 1 = 72 at 1725
READ: Master 3 reading from Target 1 = 77 at 1775
READ: Master 3 reading from Target 1 = 95 at 1825
READ: Master 3 reading from Target 1 = 37 at 1875
READ:Master 3 reading from Target 1 = 216 at 1925
READ:Master 3 reading from Target 1 = 184 at 1975
READ:Master 3 reading from Target 1 = 198 at 2025
READ:Master 3 reading from Target 1 = 182 at 2075
```

事务的日志可以做得更高级，更有助于调试用户的应用程序。注意这部分代码包含在’ifdef - ’ endif 块中。在长的回归测试中，不需要这么详细的事务日志，因此应该有条件地包括进去。

2.6 用于 FPGA 原型测试的 SVA

目前存在多种多样的高级验证方法，它们有助于更快地发现缺陷。在这些方法中，约束随机测试平台和断言是其中的重要一块。为了确信所有可能的功能都已经正确测试到而写出成千的测试是很常见的。虽然在 RTL 验证阶段能发现绝大多数缺陷，在已实现了的门级验证中仍发现功能的缺陷也是很常见的。门级模拟一直是一个性能瓶颈，将来也是如此。在门级运行所有在 RTL 验

证阶段开发的测试不是很实际。由于门级的模拟相当慢，越来越多的验证工作组依靠其他的验证方法，如：形式验证(formal verification)、FPGA 原型(prototype)测试等，如图 2-14 所示。在实际的硅片上进行验证，可以很大程度上加快验证过程。它允许在实际的硅片上穷举地运行 RTL 阶段开发的回归测试。

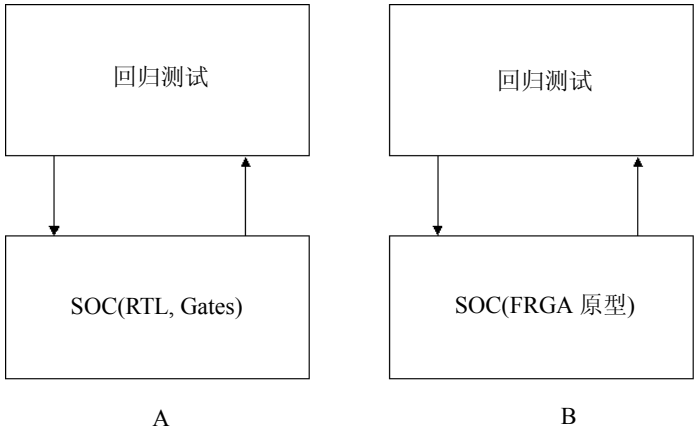

图 2-14 FPGA原型测试

在实际的硅片原型上运行测试的一个最主要的挑战是调试。在这方面 SVA 能有很大的帮助。通过综合检验器与设计，可以使调试过程更加容易一点。检验器是按照功能规范而写的，有它们监视实际硅片中的设计将使验证受益匪浅。为了适应这些断言，设计需要稍作更改。如果一个断言失败了，设计要用一个输出端口通知外部环境。使用断言的执行块，输出端口的结果就能被更新。在绝大多数实时测试中，可以在这些输出端口设置断点，当这些调试端口的其中一个失败时，可以停止验证，以便进行更深入的分析。实例系统的主控设备如图 2-2，它只包含了设计相关的默认端口。主控设备的 Verilog 代码实例如下。

```
module master(ask_for_it, clk, req, gnt, frame,
irdy, trdy, data_c, r_sel, data_o);

  input clk, gnt, ask_for_it;
  input [1:0] trdy;
```

```
output req, frame, irdy, r_sel;
output [8:0] data_c;
input [7:0] data_o;

parameter master_sva = 1'b1;
parameter master_sva_severity = 1'b1;

// functional description of master

// Block level SVA checks

endmodule
```

块级断言应该成为设计的一部分，这样有助于 FPGA 的原型测试。每个块级断言都应该有一个与之相连的调试输出端口。如果断言失败，调试输出端口应该被断言。下面描述的代码显示这是如何实现的。

```
module master (ask_for_it, clk, req, gnt, frame,
irdy , trdy , data_c , r_sel , data_o ,
a_master_start1_flag, a_master_start2_flag,
a_master_stop1_flag, a_master_stop2_flag,
a_master_data1_flag, a_master_data2_flag,
a_master_datao1_flag, a_master_datao2_flag);

  input clk, gnt, ask_for_it;
  input [1:0] trdy;
  output req, frame, irdy, r_sel;
  output [8:0] data_c;
  input [7:0] data_o;

  // debug pins for FPGA prototyping
  output a_master_start1_flag;
  output a_master_start2_flag;
  output a_master_stop1_flag;
  output a_master_stop2_flag;
  output a_master_data1_flag;
  output a_master_data2_flag;
  output a_master_datao1_flag;
  output a_master_datao2_flag;

  parameter master_sva = 1'b1;
  parameter master_sva_severity = 1'b1;
```

```
// functional description of master

// Block level checks for prototype debugging

`ifdef master_debug

d_a_master_start1:
    assert property(p_master_start1)
      else
                a_master_start1_flag = 1'b1;
d_a_master_start2:
      assert property(p_master_start2)
      else
                a_master_start2_flag = 1'b1;
d_a_master_stop1:
      assert property(p_master_stop1)
      else
                a_master_stop1_flag = 1'b1;
d_a_master_stop2:
      assert property(p_master_stop2)
      else
                a_master_stop2_flag = 1'b1;
d_a_master_data1:
      assert property(p_master_data1)
      else
                a_master_data1_flag = 1'b1;
d_a_master_data2:
      assert property(p_master_data2)
      else
                a_master_data2_flag = 1'b1;
d_a_master_datao1:
      assert property(p_master_datao1)
      else
                a_master_datao1_flag = 1'b1;
d_a_master_datao2:
      assert property(p_master_datao2)
      else
                a_master_datao2_flag = 1'b1;

`endif

endmodule
```

注意，当出现失败时，对应的输出端口的标识将被断言。由于这些断言是并发的，它们会在每个时钟边缘寻找有效的起始点。如果在断言失败时硅片测试机制不能提供一种方法设置断点，那么就要求这个失败能被锁存。否则，如果在后续的时钟周期断言成功了，这个失败就会丢失。

2.7 SVA 模拟方法的小结

- SVA 对测试平台环境的补充使动态模拟的效率更高。
- 设计者非常熟悉设计的内部功能，因此，他们应该把 SVA 检验器内嵌到相应的设计模块中。
- 验证工程师负责集成和验证系统，他们应该添加系统级断言，来彻底验证接口协议。
- 验证工程师应该能从验证环境控制或配置块级断言(如果需要，他能打开或关闭断言)。
- 采用 SVA，几乎不费吹灰之力就可以收集到功能覆盖的信息。应该有效利用这些信息来建立反应测试平台。
- 由于 SVA 在模拟全过程中监视设计的协议，它能用来创建信息丰富的日志文件。
- 按照可综合的编码风格编写 SVA 检验器，使 SVA 检验器能成为网表的一部分，可以用于调试原型或混合模拟中的失败。