

第 11 章 后综合设计任务

在专用集成电路(ASIC)的设计流程中，后综合任务包括：后设计验证、时序验证、测试生成及故障模拟^①。设计验证确保综合后网表的功能与 RTL 模型的功能相一致；时序验证检查设计的物理实现是否满足时序规范，并且确定同步电路可工作的最高频率；测试生成用于生成激励模板，以便检查电路制造过程中产生的故障；故障模拟则决定利用激励查找电路故障的优劣程度。

11.1 后综合设计验证

后综合设计验证的目的不是验证功能的正确与否。因为在后综合之前，用 RTL 模型进行的功能验证比用门级模型更加有效。

设计验证有两种方法：形式验证和仿真。形式验证不在本书的讨论范围，本书只考虑在 ASIC 行业中广泛应用的仿真。这里的目的是希望通过后综合的设计验证，检查出 RTL 模型和门级网表之间在功能上是否相同，否则可能导致实际电路无法正常工作。通过对 RTL 模型和门级模型施加相同的激励进行仿真，用 Verilog 的系统任务记录各自的响应并将二者的响应进行对比。

一些潜在的因素导致 RTL 模型与门级模型的仿真结果不一致。门级模型使用的是标准单元，它们被嵌入了器件的描述工艺参数的传输延时，而 RTL 模型则无此延迟。因此，当工作时钟频率足够高时，门级模型中将出现时序违约错误，而 RTL 模型则无此问题。由于门级模型和 RTL 模型处理时序违约的描述方式的区别，使门级电路和 RTL 电路的信号传输可能有所不同。若时钟频率已给定，则 RTL 模型必须要重新建模及同步设计，以便实现这两种电路模型之间的时序匹配。

如果状态机的描述中存在软竞争，也可能发生两种电路模型仿真结果不一致的情况。通常，如果在 Verilog 的多个周期行为模型描述中对同一变量同时进行赋值操作，则会导致软竞争现象的出现。由于仿真器所执行的多个并发周期行为描述的执行顺序是不确定的，而且也没有一种办法不依赖仿真器就能确定对多个周期行为模型描述中的同一变量同时赋值的操作顺序，所以要特别当心某个变量在一个行为描述中被赋值而在另一个行为描述中该变量同时被引用的情形发生。在无锁存时序电路中，为了避免由软竞争所产生的模糊结果，可把所有进程赋值语句放在单独一个周期行为描述中，并按照正确的赋值顺序将语句排序。

如果从数据通道到控制数据通道的状态机之间存在反馈，则该时序电路就可能产生竞争。利用第 7 章中讨论的方法可以实现无冒险逻辑电路，即：在边沿敏感的行为描述中使用非阻塞赋值($<=$)，在电平敏感的行为描述中使用阻塞赋值($=$)，而且在多个阻塞赋值语句^②中没有某个变量被同时赋值和引用。

带锁存器的电路设计风格也可能产生竞争条件。如果锁存器处在一个重汇聚扇出节点，电路将存在竞争条件^③。例如，当锁存器的使能信号线和数据通道是一个相同的变量时，则使能信

① 见图 1.1。

② 参见 Howe^[1]有关在设计验证中仿真结果不一致的详细论述。

③ 如果存在多条通路从另外的节点到达某节点，则称该节点为重汇聚扇出节点。

号和数据可能同时改变，从而发生竞争。由于竞争结果的不确定性，因此要避免采用这种设计风格。

例 11.1 图 11.1(a) 中锁存器的输出和电平敏感(高电平锁存，低电平使能)事件控制表达式(敏感列表)都是以数据通道为启动条件的。RTL 模型 *Latch_Race_1* 和 *Latch_Race_2* 只是在周期行为描述代码的循环动作(always)中的顺序不同。代码的注释解释了要关注的事件执行顺序和输出波形^①。图 11.1(b) 中的仿真结果显示了 *D_out_1* 和 *D_out_2* 的不同之处。这些模型可综合成相同的电路结构，它是一个将 *D_in* 输入到数据端和使能输入端的硬件锁存器。这些例子说明了在数据通道和锁存器的使能信号间存在竞争的危险性。为了便于阐述，假设在 *Latch_Race_3* 和 *Latch_Race_4* 中是有延时的。这些模型也可综合成将 *D_in* 连接到数据端和使能输入端的一个锁存器。实际电路中存在的传输延时即为从输入到输出的相对延迟，实际电路的输出波形如 *D_out_3* 和 *D_out_4* 所示，从该波形可明显看出仿真结果的不一致。

```

module Latch_Race_1 (output reg D_out, input D_in);
  reg En;
  always @ (D_in) En = D_in;
  always @ (D_in, En) if (En == 0) D_out <= D_in;

  // D_in triggers second behavior, with residual En (Enabled-low)
  // D_out is scheduled to get D_in (1)
  // First behavior is triggered by D_in
  // En is updated (1)
  // Second behavior is triggered by En
  // D_out is latched, so no change is scheduled
  // Scheduled value of D_out is assigned (1)
endmodule

module Latch_Race_2 (output reg D_out, input D_in);
  reg En;
  always @ (D_in, En) if (En == 0) D_out <= D_in;
  always @ (D_in) En = D_in;

  // Second behavior is triggered by D_in changing (0 to 1)
  // En is updated (1)
  // Second behavior is also triggered by D_in, with updated En value (1)
  // D_out is latched, so no change
  // Waveform of D_out_2 is consistent with this assumption
endmodule

module Latch_Race_3 (output reg D_out, input D_in);
  wire En;
  buf #1 (En, D_in);
  always @ (D_in, En)
    if (En == 0) D_out <= D_in;
endmodule

  // Change in D_in schedules delayed change in En
  // Change in D_in schedules change in D_out
  // D_out is updated
  // Delayed change in En triggers cyclic behavior, but En has
  // D_out latched.
  // D_out_3 should exhibit change of D_out_3 coincident with D_in

module Latch_Race_4 (output reg D_out, input D_in);
  wire En, D_in_del;
  buf #2 (D_in_del, D_in);

```

^① 仿真器在第一个循环行为之前执行第二个循环行为。

```

buf (En, D_in);
always @ (D_in, En)
  if (En == 0) D_out <= D_in;
endmodule

```

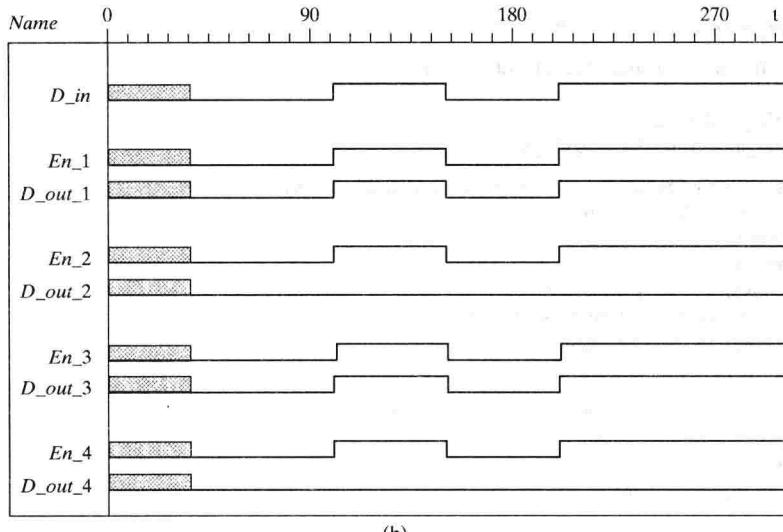
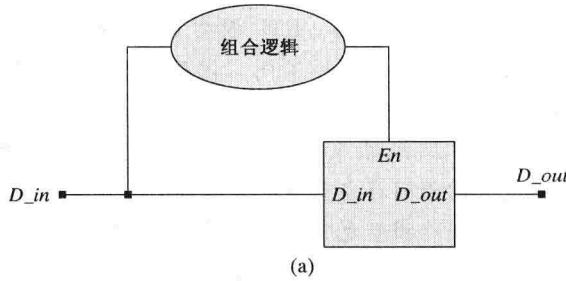


图 11.1 锁存电路:(a)在数据通道和使能输入之间有竞争;(b)在 RTL 模型中被锁存的值取决于在代码中赋值语句的执行顺序;实际电路中存在的传输延时取决于该器件输入到输出的相对延迟,应避免采用这种风格的设计

11.2 后综合时序验证

后综合时序验证是十分必要的,因为 RTL 模型的功能验证并没有考虑传输延时,因而不能验证该模型是否满足硬件时序的约束和输入/输出(I/O)时序的性能指标。综合工具将 RTL 设计映射成具体的物理实现并进行时序分析。由于综合工具仅考虑了设计中的线网互连和网线的负载电阻及寄生电容特性所带来的延时,并据此对预布局布线(如线负载模型)进行了数据估计,因此综合工具所进行的时序分析的精度有一定的局限性。由于映射工具中并没有获取到在布局布线步骤中所产生的实际延时,因此综合工具所产生的网表并不能准确描述所需的延时,此时的延时仅是实际延时的估计而已。线网互连的电阻和寄生电容的实际数值要从版图中提取,并对门电路的延时模型进行反标注,才能得到布局布线后准确的时序分析结果^①。

^① 有关 EDA 工具(通过给综合引擎提供布线信息来改善时序收敛)的更多的最新发展,参见 www.cadence.com。

电路能否正常工作，其本质上是受称为关键路径的最长逻辑通路的限制，以及受芯片中存储器件的物理约束或工作环境的影响。为了确保电路能够满足设计规定的时序规格及器件的约束条件，必须验证关键路径以及与关键路径延时相近的通路是否满足时序要求，这就必须要考虑逻辑门的传输延时、门之间的互连、时钟偏移、I/O 时序裕度以及器件约束（如建立时间、保持时间和触发器的时钟脉冲宽度）。如果边沿触发器的建立或保持时间这个约束条件被违反了，则触发器将进入亚稳态（参见第 5 章）。

时序验证利用电路的器件和互连模型来分析电路时序，以此来判断物理设计（版图设计前/后）是否能达到硬件的时序约束条件和输入/输出的时序规范。时序验证可直接仿真电路的行为，以此判断是否达到硬件约束和性能指标，或者不直接仿真电路的行为而是通过分析电路中所有可能的信号通路来间接判定是否满足时序约束条件。时序验证的这两种方法分别称为动态时序分析（DTA）和静态时序分析（STA）^[2,3]。表 11.1 比较了动态时序分析和静态时序分析的特点。

表 11.1 时序验证方法的比较

	时序验证方法	
	动态分析	静态分析
方法	仿真	路径分析
对测试模板的要求	需要	不需要
覆盖率	取决于测试模板	与测试模板无关
风险	警告丢失	警告错误
最大最小分析	不可行	可行
与综合配合	不可行	可行
CPU 运行时间	数目/数周	数小时
内存使用	大量	少量

DTA 利用电路的行为级、门级及开关级模型来仿真并分析该电路的功能通路，用模拟电路级仿真器^①来仿真晶体管级模型。STA 使用与 DTA 相同的模型，但 STA 系统在全面详尽地提取电路的门级描述的拓扑结构的基础上，计算所有通路上的传输延时，从而生成一个有向无环图（DAG）。如果电路的所有可能信号通路都符合时序约束和性能指标，那么对所有的输入激励而言该电路就都能满足实际工作时的设计需求。

DTA 和 STA 存在不同的风险及各自的代价。由于 DTA 依赖激励源的设计，因此若使用的激励源不完备，就可能漏掉电路中的最长延时通路（即关键路径），导致 DTA 漏报时序违约风险警告。但是对复杂电路而言，要设计一系列完备稳定的激励元组合来检查复杂电路的时序验证且不漏报，则是十分困难和不现实的。与之相比，STA 能全面详尽地考虑电路所有可能的拓扑路径，甚至可以检查并报告无效信号路径（即在运行中不可能用到的通路）上的时序违约，从而生成错误虚假的违约警告。此外，由于大规模电路的 DTA 事件驱动仿真需要很大的存储器空间，与 STA 相比，在分析设计和检查时序违约时，DTA 相对较慢。

时序收敛是指在一个设计中，其逻辑单元的布局布线、信号通路和时钟树是否满足时序规范的要求。由于综合工具中并未包含有关布局布线后的延时信息，当时序收敛不能满足时，就需要重新综合或者重新对电路进行布局布线，由此将产生额外的设计成本。因此一些综合工具通过

① 模拟仿真是很费时的，通常仅用来验证高性能电路的关键路径。

对包含物理综合的整个设计流程^①进行综合考虑，以此设置更为准确的内部互连负载模型，从而解决时序收敛的难题。

DTA 分析百万逻辑门规模以上电路的能力是有限的。在如图 11.2 所示的 SoC(片上系统)芯片中集成了多家公司的 IP(知识产权)核，此时 DTA 的应用则十分有限。由于把多个不同的 IP 核的测试模板整合在一起非常困难，因此就限制了仿真的覆盖能力。与此相比，STA 不受激励源和测试模板的限制，仅花费更少的仿真时间且具有更广的应用范围，是进行 SoC 设计时序分析和验证的十分适用的方法。由于 STA 适用于将一个复杂电路划分为多个子电路模块的设计应用，因此 STA 同样适用于数百万门规模的多核 SoC 设计。

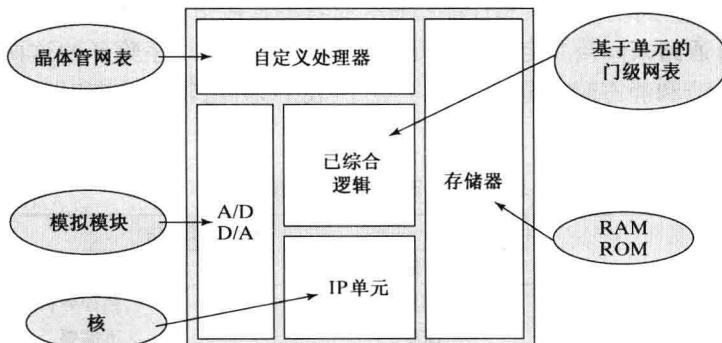


图 11.2 将多个 IP 集成在一个芯片内的 SoC 设计方法

11.2.1 静态时序分析

这里主要关注在同步电路设计时，整个芯片的门级静态时序分析。STA 由电路的网表形成一个 DAG。DAG 的节点代表逻辑门，DAG 的边代表信号通路。DAG 的拓扑通路包含电路的时序通路(即把激励模板信号加到电路的输入端所生成的信号传输通路)。DAG 的每条边上标注了每个通路的传输延时。DAG 必须是无环的，即无反馈通路。

例 11.2 图 11.3(a) 中电路的时序 DAG 如图 11.3(b) 所示。为简便起见，假设本例中该 DAG 的逻辑门的上升和下降延时均为对称的。

一个电路可能会有四种时序通路：

- (1) 从电路的原始输入端到存储单元的数据输入端的通路；
- (2) 从一个存储单元的输出端到另一个存储单元的输入端的通路；
- (3) 从存储单元的数据输出端到电路的原始输出端的通路；
- (4) 从电路的原始输入端到原始输出端的通路。

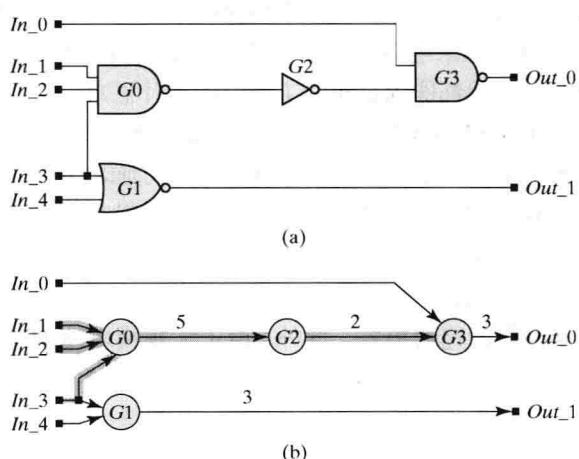


图 11.3 一个组合逻辑电路及其时序 DAG

^① 参见 www.magma.com 网站。

每一类通路都要经过组合逻辑部分。STA 检查从源到目标(通常称为起点和终点)之间的时序通路,如图 11.4 所示。

电路的时序通路的起点是原始输入端(即封装的输入引脚)和时序电路中存储单元的时钟引脚。在时序电路器件中的物理通路被连接到该器件的输出端,由于时钟是对沿着物理通路的信号传输进行初始化的,因而时序通路的起点是时钟引脚。电路的时序通路的终点是原始输出端(即封装引脚)和存储单元的数据输入端。并不是 DAG 的所有拓扑通路都是时序通路,在给定的起点和终点之间也可能存在不同的时序通路,这主要取决于信号的上升和下降是否有对称的传输延时。

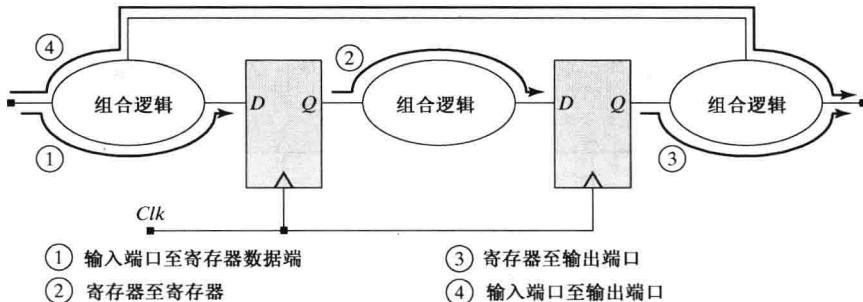


图 11.4 对同步电路进行时序分析时,信号通路的起点和终点

11.2.2 时序规范

性能规范可能约束与外接电路接口处的信号偏移以及内部通路的延时^①。输入延时(偏移)约束适用于从原始输入端(输入引脚)到电路中存储单元的信号时序通路,它规定了相对于触发信号的时钟有效沿,以及输入信号到达的最迟时间。在一个完全同步系统中,到达电路输入引脚的信号可通过与电路输入相连的时钟沿来触发。时序分析器使用规定的输入约束 $t_{\text{input_delay}}$ 来确定到达信号和下一个时钟有效沿之间的时序裕度 $t_{\text{input_margin}}$,如图 11.5 所示。 $t_{\text{input_margin}}$ 确定了输入信号通过电路内部组合逻辑到达通路终点的可用的时间裕度,以此满足触发器的建立时间要求。

输出延时(offset)约束适用于从存储单元的输出到原始输出的时序通路。输出约束指定了相对于起点处的时钟有效沿,从起点到终点信号传播的最长时间,如图 11.6 所示。时序分析器用 $t_{\text{output_delay}}$ 来计算 $t_{\text{output_margin}}$,从而使得输出信号在下一个时钟有效沿之前有足够的时问到达目的地。

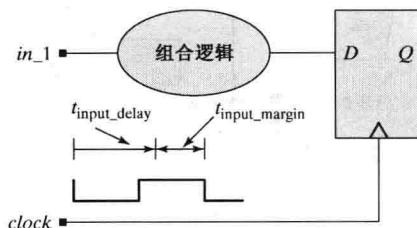


图 11.5 输入延时约束

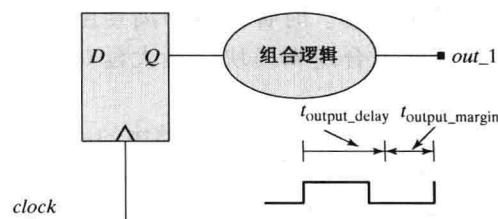


图 11.6 输出延时约束

^① 通路约束条件影响着综合工具综合出满足约束条件的实现所需要的时间,同时也影响着 FPGA 内部逻辑单元的布局。FPGA 设计初期应该不包括时序约束和引脚分配,以确定该设计能否与所选择的部件相适应。

输入-输出(I/O,引脚到引脚)约束适用于从原始输入端到原始输出端的通路。I/O 延时约束了从原始输入端到原始输出端之间组合逻辑通路的最大时序长度。

周期时间(时钟)约束规定了同步电路的最大时钟周期，它适用于寄存器之间的通路。该约束指定了一个指定的时钟信号的周期。时序分析器也认可关于时钟波形的占空比和偏移量的有关约束。如图 11.7 所示，如果一个电路有多个独立的时钟域^①，则可以按照时钟域中被时钟同步的存储单元来对通路进行分组。终点不是存储单元的数据输入端的通路分在默认组中。

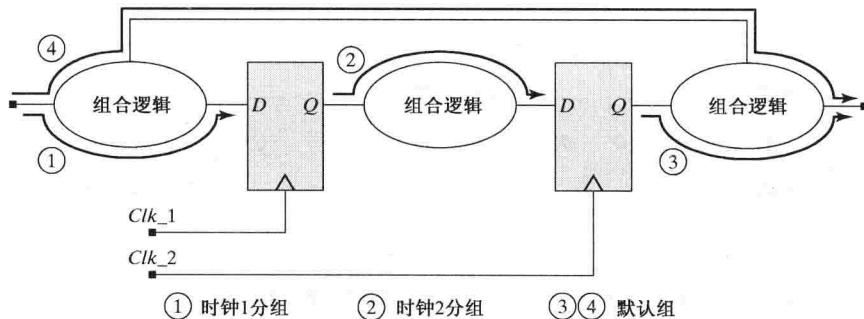


图 11.7 具有多时钟域的同步电路信号通路分组

时序分析用来确定电路中通路的时序约束是否已得到满足。在电路中不允许有任何组合形式的反馈环路存在，而且所有寄存器的反馈通路都在各自的时钟域内。对所有通过组合逻辑模块的四类通路，根据上升和下降沿的传输都可计算出每条通路上的延时，把从终点到起点的逆向通路上的所有延时累加，得到该通路上的总传输延时。通路按照其延时的长度进行分类，并验证通路的输入/输出时序约束是否满足。如果给定的器件传输延时是一个时间范围(即 min : max)，那么这个通路的延时也将表示为一个时间范围。

输入约束的验证要考虑到通路终点处时序器件的建立时间。同样，输出约束的验证也要考虑到时钟信号的时钟至输出的延时，此处的时钟将激励信号传输至原始输出。周期时间约束的验证必须考虑在通路终点器件的时钟至输出的延时、通路中组合逻辑的传输延时、通路终点器件的建立时间以及时钟的偏移。时钟的最小周期由存储单元之间的组合逻辑模块的最大延时通路决定。

上面的时序分析是基于电路中的通路是静态敏感的(statically sensitized)，即通路中逻辑门的其他输入都是固定的，并且不会阻塞信号通过该逻辑门的传输。例如，在图 11.8 中 NAND 门的其余输入必须是 1。

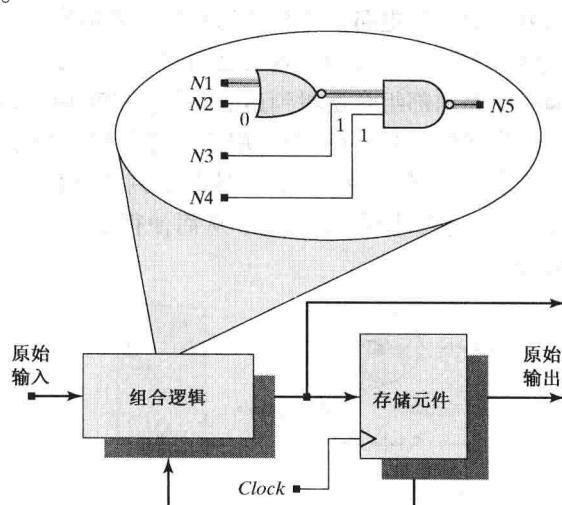


图 11.8 静态敏感电路上其余的门输入不会阻塞信号通过逻辑门

^① 时序分析器可能认为时钟信号是由同一个源(非独立时钟)得到的，但这里不予考虑。

11.2.3 影响时序的因素

如图11.9所示，典型同步电路中的信号通过组合逻辑从源寄存器向目的寄存器传输。通路传输延时包含信号从 clk 至传输信号的寄存器输出端之间的延时和信号通过该通路上的逻辑门的传输延时两部分。逻辑门的传输延时受到其驱动的输入引脚电容(即通路上的扇出负载)和由通路的电阻和分布电容产生的负载的影响。

连接到输出端的逻辑门和线网增加了其固有电容，从而输入端的逻辑门发生变化并在输出端反映出来时，使得传输延时进一步增加。同时，驱动逻辑门信号的斜率也会影响逻辑门的传输延时，因为如果输入信号有较长的上升时间，那么在RC网络中电容的充电过程相比于输入信号为阶跃信号的情况慢得多。在电路的给定时钟域中，寄存器应通过一个公共时钟来同步，但是由于时钟信号沿着不同的物理通路传输，所以实际的芯片中到达各寄存器的时钟脉冲的边沿不可能绝对一致(未对准)。在同步电路中时钟沿的未对准称为时钟偏移(*clock skew*)。时钟偏移减少了目的寄存器中数据和时钟信号之间的时序裕度。

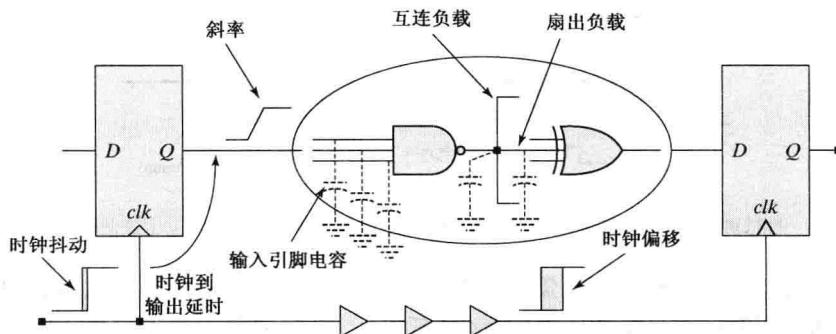


图11.9 影响同步电路时序的因素

电路中通路的最大延时由以下几部分组成：通路上的逻辑门和存储单元的内部传播延时^①，通路上逻辑门的扇出负载，信号通路上的互连负载以及信号的斜率。时钟周期必须与电路中寄存器之间的最大延时通路的延时相适应。

如果满足以下条件，则该通路为最大延时通路：

- (1) 通过最大延时通路上所有逻辑门的功能延时之和不能小于其他任何组合逻辑通路上的总延时；
- (2) 必须存在一个原始输入和存储单元逻辑值的向量模板，使输出的逻辑值依赖该通路上每个节点的逻辑值，即：由起点到终点的通路是敏感的，如果通路的终点是存储单元，则时钟信号的跳变对该器件使能。

以下术语和标记将用于描述影响同步电路最小时钟周期的因素：

$t_{clk_to_Q}$	时钟有效沿与被该时钟同步的触发器的有效输出之间的延时。
t_{comb_max}	通过组合逻辑的最长通路延时。
t_{setup}	由组合逻辑驱动的触发器的建立时间。
t_{skew}	时钟偏移。

t_{comb_max} 延时取决于固有的逻辑门延时、信号的转换速度、与扇出及与路径相关的互连所产生

① 逻辑门的固有延时独立于扇出负载。

的负载。在深亚微米设计中(即实际尺寸 $\leq 0.18 \mu\text{m}$)，互连延时起主要作用。时钟周期必须足够长，使得信号能够满足目的寄存器的建立时间裕度，即在寄存器建立时间内，信号必须及时稳定地到达寄存器数据输入端。换言之，最长通路必须满足下列约束： $t_{\text{comb_max}} < T_{\text{clock}} - t_{\text{clk_to_output}} - t_{\text{setup}} - t_{\text{skew}}$ ，或 $t_{\text{setup_time_margin}} > 0$ 。此处： $t_{\text{setup_time_margin}} = T_{\text{clock}} - t_{\text{clk_to_Q}} - t_{\text{setup}} - t_{\text{skew}} - t_{\text{comb_max}}$ 时，如果 $t_{\text{setup_time_margin}} \leq 0$ ，则电路违反了周期时间约束。

寄存器的保持时间裕度是对通过逻辑模块的最短通路的约束。最短通路必须满足下列约束： $t_{\text{comb_min}} > t_{\text{hold}} - t_{\text{clk_to_output}} + t_{\text{skew}}$ ，或 $t_{\text{hold_time_margin}} > 0$ 。此处： $t_{\text{hold_margin}} = t_{\text{comb_min}} - t_{\text{hold}} + t_{\text{clk_output}} - t_{\text{skew}}$ 。这个约束避免了通路中起点寄存器的输出与通路中终点寄存器^①数据输入之间的竞争。通路中起点的信号值的变化不能太快地到达终点寄存器。

图 11.10 显示了时钟周期 T_{clock} 与信号通路延时之间的关系，图中忽略了时钟偏移。时钟周期必须大于以下三种延时之和：时钟至输出的延时、组合通路的最大延时、终点器件(假定为触发器)的建立时间。通路的时间裕度 t_{slack} 是时钟周期与通路延时的差值。对于任何电路的通路，若 $t_{\text{slack}} \leq 0$ ，当激励施加于电路时，电路会发生建立时间的时序违约。

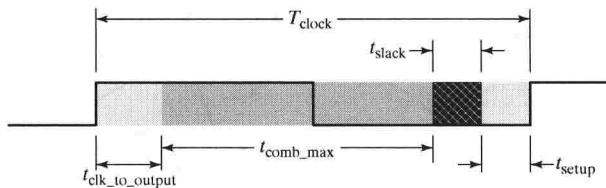


图 11.10 无时钟偏移的电路中时钟周期必须满足以下条件： $T_{\text{clock}} > t_{\text{clk_to_output}} + t_{\text{comb_max}} + t_{\text{setup}}$

电路的同步运行要求所有的存储器单元都要在相同的时钟沿进行同步。时钟偏移是到达目的地的时钟沿相对于时钟信号源的边沿的变化(延迟)。时钟偏移的产生是由于时钟本身固有的抖动或由于受时钟信号驱动的单元在布线时引入的不同传输延时所引起的。布线所引入的时钟偏移不仅与负载(电阻电容的金属互连线及存储单元)有关，还与时钟分配通路上的缓冲器链路有关。金属互连线引入了与其线长成正比的传输延时。通路所产生的时钟偏移是不可避免的，必须予以考虑^[4]。

图 11.11 解释了一个被偏移的时钟其边沿的模糊和不确定性。抖动在时钟边沿的正常位置产生了一个模糊区域。时钟的实际跳变发生在阴影区域，但具体位置却是不确定的。图 11.12 给出了可以确定出具有时钟偏移的最高频率时钟的依据。与没有时钟偏移的时钟相比，时钟偏移使得时钟的最小周期增大。当存在时钟偏移时，时钟周期必须满足如下约束： $T_{\text{clock}} > t_{\text{clk_to_output}} + t_{\text{comb_max}} + t_{\text{skew}}$ 。

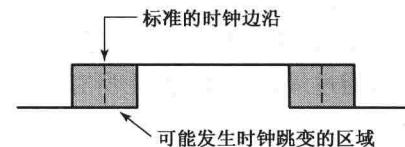


图 11.11 时钟偏移导致的时钟边沿模糊

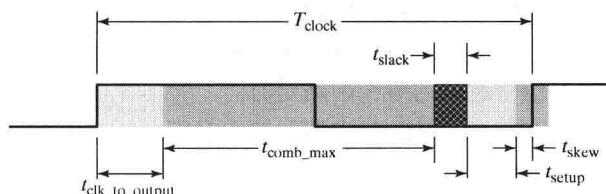


图 11.12 时钟周期必须增大以补偿时钟偏移，并应满足条件： $T_{\text{clock}} > t_{\text{clk_to_output}} + t_{\text{comb_max}} + t_{\text{skew}}$

^① 注意，最小通路延时能够用于保持时间约束。

时钟信号通路上的缓冲器和其他逻辑引入了时钟偏移。在图 11.13 中, 由于时钟边沿在每个寄存器处出现的时间不同, 因此时钟边沿处的模糊区会沿着通路逐渐累加。对于给定的时钟周期, 允许的时钟偏移的边界为: $T_{clock} > t_{clk_to_output} + t_{comb_max} + t_{setup}$ 。

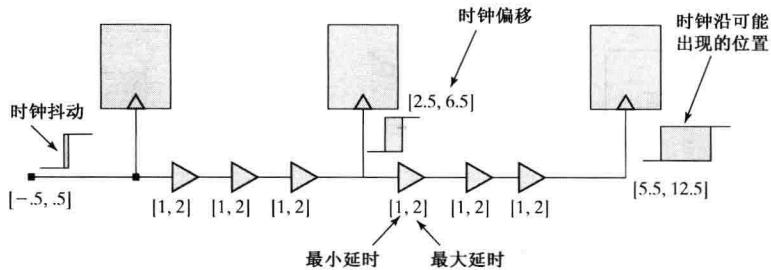


图 11.13 时钟通路上的缓冲器递增了目的寄存器处的时钟偏移

例 11.3 图 11.14(a)中的移位寄存器在时钟分配线网中是非均衡的(即具有不同的缓冲延时)。图 11.14(b)中的仿真结果比较了均衡延时和非均衡延时下寄存器的输出。在非均衡延时的寄存器中, D_{in} 通过带有时钟偏移的寄存器时经历了 3 个时钟周期, 而非 4 个时钟周期。

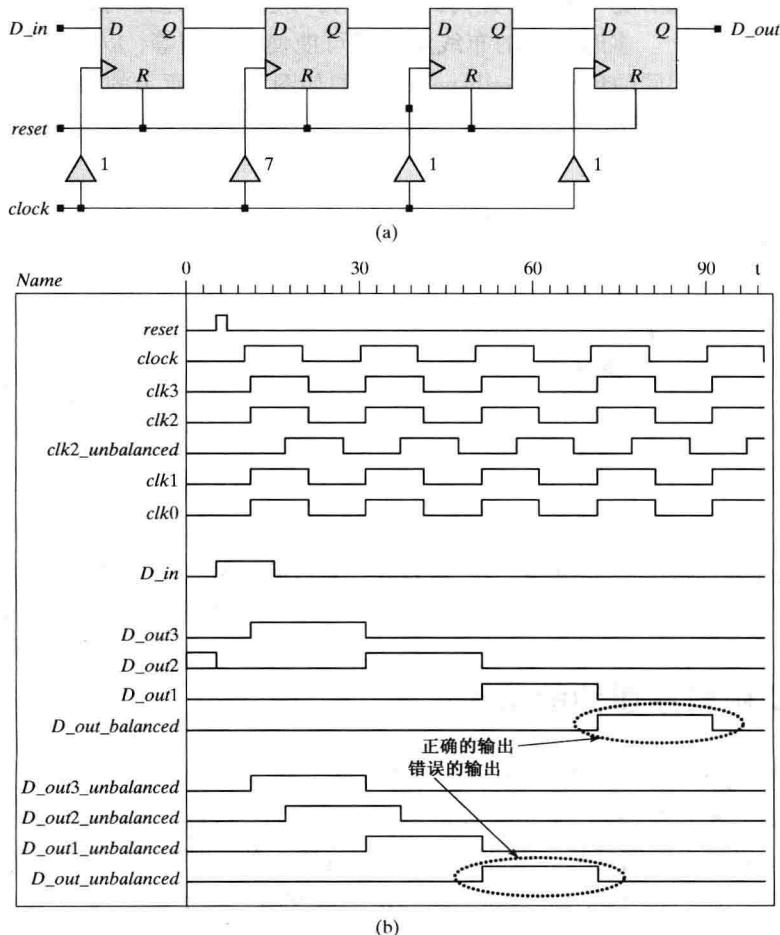


图 11.14 时钟偏移的作用: (a) 非均衡时钟分配条件下的移位寄存器; (b) 错误的寄存器输出波形

例 11.4 图 11.15(a)的时钟分频器中由 *clock* 产生了 *clock_by_2*, 但缓冲器的延时使 *clock_by_2*滞后 *clock*。图 11.15(b)所示的推荐电路^[5]为触发器分配了一个共用的时钟, 还省掉了一个缓冲器。

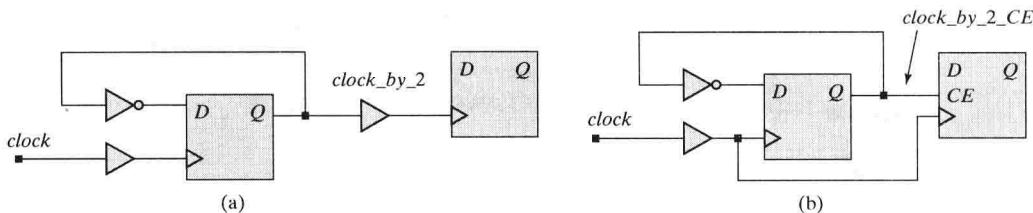


图 11.15 两种时钟分频电路: (a)有时钟偏移的 *clock_by_2*; (b)两个触发器共用一个时钟同步, 无时钟偏移

对于 ASIC 的时钟分频网络, 必须仔细设计, 实现用尽可能短的过渡延时来使时钟偏移的影响最小化之目的。如图 11.16 所示, 使时钟引脚到所有存储单元的延迟相等, 以便将时钟偏移降到最小。在对时钟树进行布局布线设计时应使用专用的编译器。平衡的时钟树设计可以实现对称的物理和时序的拓扑结构, 从而达到时钟边沿同步之目的。否则距离时钟源最远的时钟边沿要比最近的时钟边沿出现的晚一些。时钟树(也称 H 树)应该使经过树的延时都相等, 并且能够减小负载端的峰值电流^[4]。多时钟树的布线应该尽可能使负载相等, 以减小时钟相位的偏移。每一个时钟分支应该有相同的负载。Testbench 可监测如图 11.17 所示的信号, 以确认信号没有偏离得太远; 用静态时序分析软件可以确定时钟信号到达时钟树所有终点的时间, 以确保时序的同步。

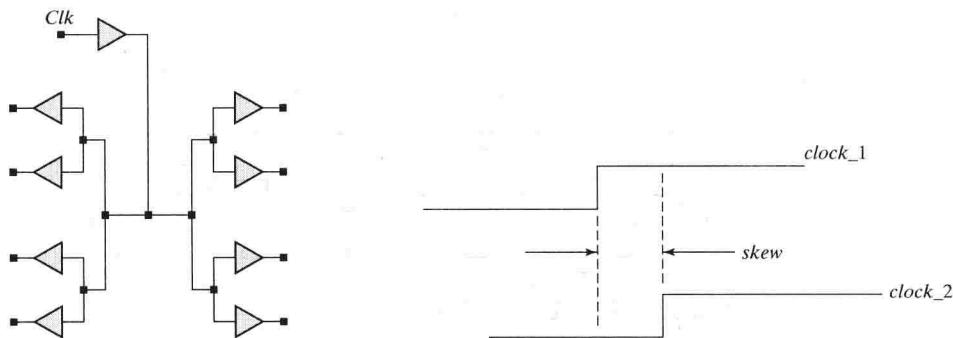


图 11.16 时钟树使时钟信号从时钟源到目的地的延时保持均衡

图 11.17 测试平台检测是否违反最大时钟偏移约束
以监测设计中的两个时钟信号之间的偏移

11.3 ASIC 中时序违约的消除

表 11.2 总结了能够消除 ASIC 中时序违约的可选方案。最简单的方案就是延长时钟周期。如果最大时钟周期没有加以限制, 此方案即可消除时序违约。否则, 一种可选方案是对电路的关键路径重新布线, 以减少线网延时。也可以通过调整通路上的器件尺寸来减小通路延时, 并在元件库所提供的器件范围内选择建立和保持时间较短的触发器。除了表 11.2 中提到的方案, 还可以通过对行为模型的重新检查, 看是否能通过对模型的改进综合出更快的逻辑。例如, 用 *case* 语句代替 *if...else* 语句可能综合出并行逻辑; 也可以改变状态编码(如使用 one-hot(独热)编码), 以便综合出速度更快的电路。

表 11.2 设计 ASIC 中消除时序违约的可选方案

ASIC 中时序违约的消除	
方案	作用
延长时钟周期	在性能指标约束内，消除时序违约
调整关键路径	减小线网延时
调整器件尺寸或更换器件	减小器件延时，改善建立和保持裕度
重新设计时钟树	减小时钟偏移
更换算法	减小通路延时(如用超前进位代替行波进位)
更改系统结构	减小通路延时(如流水线操作)
改变工艺	减小器件及通路的延时

由于 FPGA 的体系结构是固定的，因此消除 FPGA 时序违约的构架方案很少。然而，FPGA 包含快速进位逻辑(改善时序裕度)、专用时钟缓冲器网络和延迟锁定环(减小设计中的时钟偏移)。在 FPGA 中含有大量的寄存器，因此一种行之有效的方案就是用流水线来增加多级组合逻辑的吞吐量。由于 FPGA 中的 I/O 触发器具有可编程的输入延时和输出偏移的功能，因此保障了其规定的建立时间、保持时间以及时钟至输出时间。例如，Xilinx 器件的可编程输入延时保证了器件具有零保持时间和延长的建立时间。软件工具会把一个设计综合为适合 FPGA 的文件，使其满足 FPGA 中的 I/O 和内部的时序约束。对于一个给定器件，如果时序约束不能满足，可行的方法就只有延长时钟周期或者更换成具有较高的时钟频率的器件了。

没有错误的同步运行是时序验证的关键。设计中所用到的物理存储器件必须满足电路要求的建立时间约束、保持时间约束和脉冲宽度约束。建立时间约束、最长通路延时、实际布图和电路中时钟分配四者的相互影响决定了时钟偏移约束。例如，STA 可以确定电路是否有足够的裕度来满足建立条件。Testbench 也可以监视其他约束条件，如一些毛刺和信号之间的相关偏移(如存储器控制线之间的偏移)。STA 可以对通路长度的分配进行分析，决定在不违反时序约束的情况下是否可以减小器件的尺寸。

第 6 章讨论了具有优先权的功能 *if…else* 嵌套语句，并可综合成优先编码器。如果不避免地要用到这个结构，就要把关键的时序信号放到语句的第一个条件下，因为它要驱动最后一级的逻辑模块，并且到达输出端的通路最短。*case* 语句综合可得到更小更快的电路，但是 *if…else* 语句更加灵活且可实现优先级逻辑，可以用来调整最后到达的信号。嵌套的 *if…else* 语句和嵌套的 *case* 语句可综合出多级逻辑，但其性能上将有所折中。

11.4 虚假路径

电路中所有的物理通路并不一定能被全部运行。如果静态通路分析软件不顾通路的实际功能，工具软件可能会发出虚假警告。

例 11.5 在图 11.8 所示的电路中，若通路 $In_1 - w0 - w1$ 被激活，则 in_0 和 in_2 必须为 1，此时将使电路中的反相器输出为 0，从而阻塞了驱动 $w2$ 通路的逻辑门，使其不可能被激活。因此图 11.8 中的 $In_1 - w0 - w1 - w2 - Out_0$ 是一个不可能被激活的通路。注意到 In_2 可通过两条不同的通路到达 $w2$ ，此时称其为再汇聚扇出状态。当电路存在再汇聚扇出时，由于信号通过再汇聚逻辑门时，该逻辑门的其他输入值要视在通路中传播的信号值来决定，故再汇聚的逻辑门不可能被独立地激活，因此经过那些信号再汇聚的逻辑门的通路不可能是静态激活通路，如果不考虑信号在通路中传输的极性，拓扑路径延时的报告可能是不正确的。若只是把通路上的逻辑门

的最大或最小延时加起来，得到的最大或最小通路延时值效果是不准确的，并且会使得在消除时序违约时浪费很多的精力。因此在计算通路延时时，必须恰当地选择器件的上升延迟或下降延迟进行分析和计算。

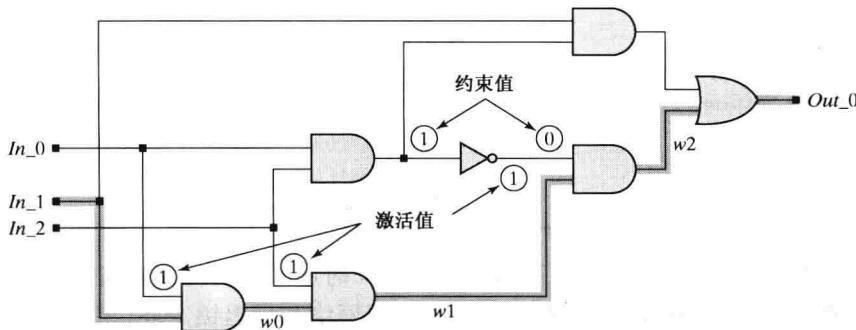


图 11.18 通路 $In_1 - w0 - w1 - w2 - Out_0$ 不是静态激活通路

例 11.6 对图 11.19 所示的通路进行延时分析，若将通路上逻辑门延时的最大值简单相加，可得到通路最大上升延时 $t_{delay_rising} = 15$ 。但是考虑到传输信号的极性，则得到 $t_{delay_rising} = 5 + 3 + 5 = 13$ 。

如果在原始输入作用下，一条拓扑通路没有任何逻辑功能，则称该通路为虚假路径。例如，若互相排斥的条件控制数据通道时，此时时序分析器的通路报告将是虚假路径。因此，软件中的控制逻辑将消除某些不应该报告的通路。

例 11.7 图 11.20 中引导数据通道的两个多路复用器的控制信号彼此相互依赖，有两条虚假路径是不会被运行的。电路中最大拓扑延时 $t_{topological} = 30 + 30 = 60$ ，而最大功能延时 $t_{max} = 15 + 30 = 45$ 。



图 11.19 反相器通路上的上升沿跳变的最大延时必须考虑通路上信号跳变的极性

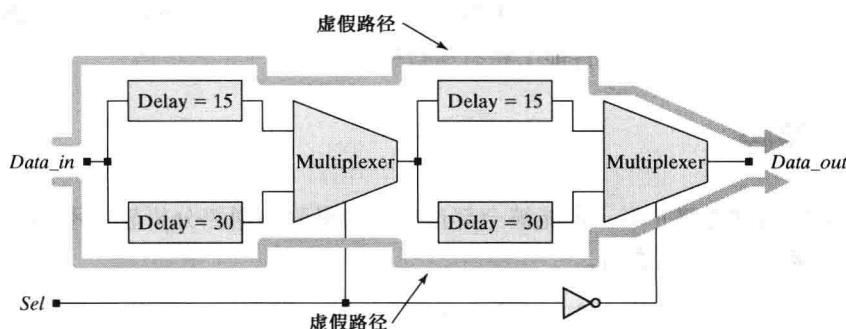


图 11.20 有虚假路径的电路

为了减少通路延时在设计中加入冗余逻辑，时序分析器可能报告虚假路径延时。如果时序分析器没有考虑到冗余逻辑所带来的延时减小，则该报告的延时结果是不准确的。同样，时序分析器若不能识别出多周期循环路径，则也可能引发错误警告。

11.5 用于时序验证的系统任务

在仿真过程中 Verilog 有一些可以进行时序检查的内置任务，其中有些任务可以包含在模块中完成以下功能：

- (1) 自动监测仿真行为；
- (2) 检测时序违约；
- (3) 报告时序违约^①。

11.5.1 时序检查：建立时间条件

如果在时钟触发的前后边沿，输入端的数据在足够的时间内不能保持稳定，则边沿触发器不能正常工作。建立和保持时间是对存储单元正确运行的逻辑约束。如果违反了存储单元的建立和保持时间约束，存储单元的不确定行为将导致系统错误。图 11.21 显示了在每个时钟的有效沿之前的建立时间间隔。检查器件建立时间违约的系统任务的语法结构为：`$setup(data_event, reference_event, limit)`。

在与 `reference_event` 相关的 `limit` 范围内，`data_event` 若不稳定，就会发生建立时间违约。实际电路中，在触发器的时钟有效沿之前，数据必须保持稳定。

建立时间违约的原因是通路延时相对于时钟周期过长。为了消除建立时间的违约，必须减小最后到达的数据的延时，或者延长时钟周期(参见表 11.2)。

例 11.8 图 11.22 显示了 `sys_clk`, `sig_a` 和 `sig_b` 的波形。应该注意 `sig_a` 满足建立时间约束，但是 `sig_b` 不满足。时序检查通过任务 `$setup(sig_a, posedge sys_clk, 5)` 和 `$setup(sig_b, posedge sys_clk, 5)` 来激活；后一个检查会报告有时序违约。

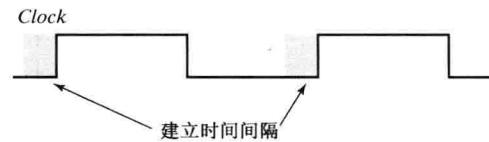


图 11.21 在时钟有效沿之前的建立时间间隔内，触发器的数据必须保持稳定

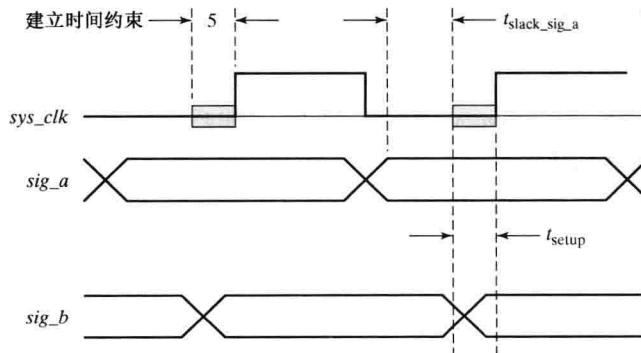


图 11.22 在时钟有效沿之前的建立时间间隔内，`sys_clk` 的建立时间约束要求 `sig_a` 和 `sig_b` 保持稳定，而 `sig_b` 违反了建立时间约束

^① 器件的时序检查通常是通过模块中的 `specify...endspecify` 块被包含在该器件的标准单元库模块中。

11.5.2 时序检查：保持时间约束

为了使触发器能正常工作，触发器输入端的数据必须在时钟有效沿之后足够长的时间内保持稳定。如果触发器的数据通道太短，导致通路中起始点触发器输出的数据传输到通路终点触发器输入端的数据变化速度太快，就会引起保持时间违约。图 11.23 显示了触发器的数据在保持时间间隔内必须稳定。

经过组合逻辑的较短通路由综合工具自动延长，以减小时间裕度并满足时序约束。设计中最理想的情况是能达到某种平衡，使得信号在通路中的传输不快不慢，刚好达到要求。不必要的快速通路会浪费硅片面积。



检查器件保持时间违约的系统任务的语法结构为：**图 11.23 在时钟有效沿之后的保持时间间隔内，触发器的数据必须保持稳定**
 $\$hold(reference_event, data_event, limit)$ 。在相对于 $reference_event$ 这个参考事件的 $limit$ 特定时间范围内， $data_event$ 若不稳定，就会发生保持时间违约。

例 11.9 图 11.24 的波形中，由于 sig_a 在 sys_clock 的保持时间间隔内不稳定，任务 $\$hold(posedge sys_clock, sig_a, 5)$ 报告了时序的违约。

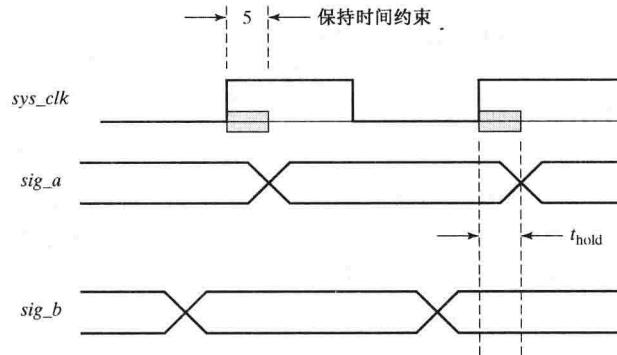


图 11.24 sig_a 不满足 sys_clock 的保持时间约束

11.5.3 时序检查：建立时间和保持时间约束

若用于同步某个器件的跳变边沿为 $reference_event$ ，按照 $\$setuphold$ 的语句规则，任务 $\$setuphold(reference_event, data_event, setup_limit, hold_limit)$ 能监测 $reference_event$ 和 $data_event$ 之间是否有建立时间和保持时间的违约。

例 11.10 图 11.25 中 sig_a 和 sig_b 均满足建立时间和保持时间约束。

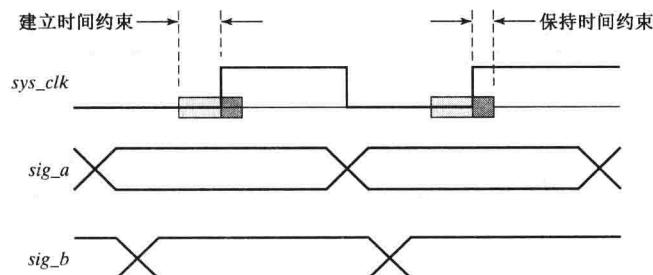


图 11.25 sig_a 和 sig_b 都满足建立时间和保持时间约束

11.5.4 时钟检查：脉冲宽度约束

时序器件的最小时钟脉度是有限制的。边沿触发器的时钟必须持续足够的时间来对内部信号节点充电。任务 `$width(reference_event, limit)` 检查最小脉冲宽度是否违约。例如，`$width(posedge clk, 4)` 对时钟脉冲进行违约检查，如果 `clk` 的上升沿与紧随其后的下降沿之间的间隔小于 4，则会出现违约。在仿真中该任务也可检查潜在的毛刺和被退化(过窄的)时钟脉冲。

例 11.11 图 11.26 说明 `clock_a` 满足最小脉冲宽度约束，`clock_b` 不满足。

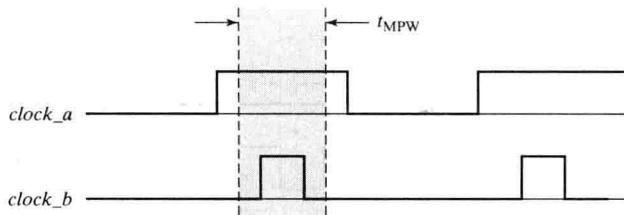


图 11.26 任务 `$pulsewidth` 利用 `clock_b` 来检查脉冲宽度违约

11.5.5 时序检查：信号偏移约束

在系统性能指标中，时钟偏移是一个很关键的问题。它是由不对称的时钟树以及建立和保持时间裕度的衰减造成的。两个信号间的时钟偏移可通过任务 `$skew(reference_event, data_event, limit)` 进行监测。如果 `reference_event` 和 `data_event` 之间的间隔超过了 `limit` 规定的值，任务会报告有违约情况。

例 11.12 在图 11.27 中，`$skew(posedge clk1, posedge clk2, 3)` 对两时钟进行违约检查，如果 `clk1` 和 `clk2` 之间的间隔超过了 3，就会出现违约。

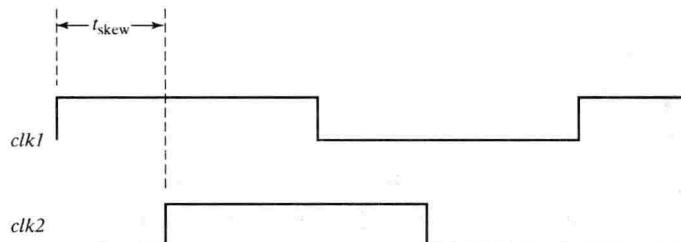


图 11.27 系统任务 `$skew` 检查时钟信号边沿之间的偏移

11.5.6 时序检查：时钟周期

时钟周期由波形的连续有效沿之间的间隔决定。任务 `$period(reference_event, limit)` 监测边沿触发器的 `reference_event` 的连续有效边沿，当连续有效边沿之间的间隔小于 `limit` 的规定值时，将报告有时序违约情况。在 SoC 的环境下可重复使用的设计中，这种时序检查将验证一个核心单元能否在指定的时钟周期下安全运行。如果满足时序验证的要求，则说明时钟周期至少和该核心单元需要的最小周期一样长。

例 11.13 图 11.28 中任务 `$period(posedge clock_a, 25)` 检查出 `clock_a` 没有时序违约。

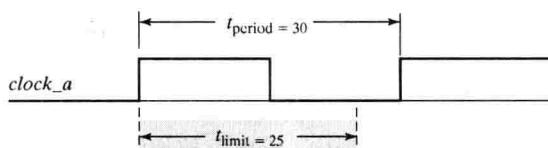


图 11.28 `clock_a` 满足其最小周期约束

11.5.7 时序检查：恢复时间

任务 `$recovery(reference_event, data_event, limit)` 将检查在复位(异步)或清零条件失效以后同步动作重新恢复所需要的时间。任务指定了异步输入在时钟有效沿之前必须稳定的最短时间。参数 `limit` 给出了 `reference_event` 的失效沿和 `data_event` 下一个有效沿之间的时间。

例 11.14 如图 11.29 中, 任务 `$recovery(negedge set, posedge clock_a, 3)` 检查 `set` 的无效沿是否超前 `clock_a` 的上升沿 3 个时间单位。

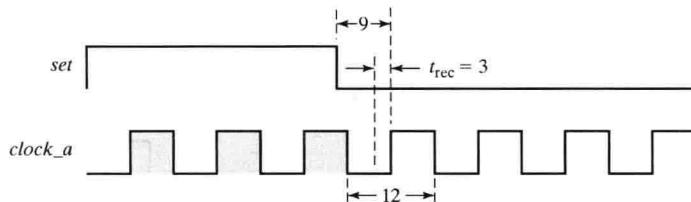


图 11.29 `set` 的下降沿(失效沿)与 `clock_a` 下一个上升沿之间的时间间隔满足恢复时间的约束

11.6 故障模拟及制造测试

若一个设计正确的电路在实际运行中仍然可能出现故障, 此时对这些故障的诊断和修复的代价是高昂的。电路的故障可能是永久的、间歇的或暂时的。永久的故障意味着电路一直不能正常工作。间歇的故障则会随机出现, 并且只持续一段有限的时间。暂时的故障一般发生在某些环境下, 在该环境下器件的性能可能会发生变化。例如, 高温或放射性环境。以下几种条件可能导致故障的发生:

- (1) 晶片缺陷;
- (2) 净室受到污染(如周围的尘埃);
- (3) 制造过程中融入了气体、水和化学杂质;
- (4) 光掩膜位置偏差。

制造过程引入的产品缺陷可能导致节点漏电流变高, 接触电阻变大, 开路, 短路, 或超出规定的门限电压。我们所关注的是那些在运行中会引起功能性错误的缺陷。

净室环境的污染会给制造出来的电路带来缺陷。净室中的空气可能包括尘埃颗粒、烟雾、挥发的清洗液或其他气态物质。晶片可能会受到没有被清洗掉的残留物和化学物质的影响。净室需要非常洁净的空气, 然而设备操作人员和技术人员在室内的活动不可避免会形成污染。净室的地板是特制的, 以尽可能减小因震动而使光掩膜对不准的概率, 如果光掩膜没有对准, 会造成器件的非正常曝光, 违反版图设计的空间约束。

鉴于器件故障的代价是昂贵的, 半导体厂商必须在产品上市前对器件进行全面测试。由于设计的错误应该在生成用于加工的掩膜文件之前, 在设计流程中的验证步骤中就被发现, 因此这种测试并不是重新验证功能特性或者检查最初设计是否有错。事实上, 用于验证器件功能的激励模板只能检查出被制造出的电路中一小部分缺陷。对制造的故障分析可以指出设计上的错误, 但这并不是目的。

在产品测试过程中, 给电路施加测试信号并监测电路的响应, 来判断电路是否有缺陷出现。施加到电路的测试模板集必须能够测试出芯片的已知故障模型, 并能够检查出其中的所有缺陷。利用测试模板(功能验证的)对产品进行测试是制造测试的第一步, 但是当测试模板不能反映某

些电路的故障状态时，该测试模板就不适用了。对具有嵌入式处理器、存储器或许还包含一个内嵌 DSP 的电路，具有 500 000 ~ 1 000 000 个逻辑门的较复杂电路时，此时设计验证用的测试模板集对复杂电路就不可能有足够的覆盖率。

产品测试的主要目的是发现因制造工艺缺陷造成的永久性故障。它包括两个主要步骤：测试生成和故障模拟。在决定如何测试之前首先要弄清楚应测试什么。在产品测试中的测试就是试图检查电路内部故障的模型(称之为故障)。测试模板的生成要与故障模拟相结合，故障模拟就是确定一个测试能否检测出电路中某个特定位置的故障。故障覆盖率可以反映测试模板集的优劣。

11.6.1 电路缺陷和故障

电路的故障状态模型要考虑因实际的物理故障造成的逻辑错误。基于 Smith 的著作^[4]中的有关讨论，把主要的物理故障总结于表 11.3 中。假定物理故障和由其导致的逻辑错误在硅片上的分布是均匀的。

表 11.3 集成电路中的物理故障及电路级影响

物理故障	衰变故障*	开路故障	短路故障
芯片级故障			
封装引线间的泄漏和短接	X		X
断裂，未对准，虚焊		X	
表面污染，潮湿	X		
金属离子迁移，应力，剥落		X	X
金属化(开路或短路)		X	X
门级故障			
触点断开		X	
栅极与源/漏极短接	X		X
场效应氧化无源器件	X		X
栅氧化裂缝、毛刺	X		X
掩膜未对准	X		X

* 参量故障(门限电压偏移)或延时故障

当电路不能正常工作时，电路实现的逻辑就会与设计所要求的逻辑不同。有各种故障模型来检测数字电路的故障，其中一种最常见的故障模型就是信号线与电源线或地线短接，这种故障模型称为粘接(stuck)故障，粘接的位置为故障点。CMOS 电路中，如果一个晶体管的栅极始终是导通的，就会出现粘接故障。这样的缺陷可能出现在光刻掩膜时，由于物理震动导致掩膜未对准，进而引起信号通路的导线发生偏移并与相邻的信号通路的导线挨得太近。也可能出现在蚀刻光刻胶时改变了导线的物理位置。为了降低故障发生率，业界统计制定了制造工艺的技术操作规范和流程，若违反了该规范就会导致此种缺陷。

逻辑单元内部晶体管互连线的短路故障称为桥接(bridging)故障。桥接故障可以通过检测电路的静态电流 I_{DDQ} 。测试静态电流要比测试粘接故障花费更多的时间。桥接故障的检测必须要测试静态电流 I_{DDQ} ，因为粘接故障的测试往往并不能检测出桥接故障，相比于错误的逻辑值，桥接故障会表现为一个很大的静态电流。

例 11.15 图 11.30(a)说明了两输入(CMOS)或非门由于两个上拉晶体管的栅极相连导致的桥接故障。正常情况下，在器件输出端发生跳变的那段极短的时间内，CMOS 器件的电源和地

线是短路的。经过电路的静态电流 I_{DDQ} 为 0，但是，当 $x_1 = 0$ 且 $x_2 = 1$ 时，电路的上拉逻辑会出现桥接故障，导致电源和地之间流过一个很大的电流，造成热损耗和器件寿命缩短。通过监测静态电流，可以检查出桥接故障^[6]。

图 11.30(b)中， x_1 的下拉晶体管为 1，即有粘接故障，导致此或非门有缺陷。由于施加到上拉电路栅极 x_1 的信号与施加到下拉电路栅极 x_1 的信号不能独立激活，这种情况下是不能通过检查电路的逻辑功能从外部检测的。而且注意，当 $x_1 = 0$ 且 $x_2 = 0$ 时 $y = 1$ ，但有粘接故障的下拉晶体管栅极把 y 拉到 0。粘接故障会引起器件的输出节点连续放电，使得上拉动作变得比无故障时缓慢。另外，经过下拉晶体管的大电流将导致热损耗并缩短电路寿命。

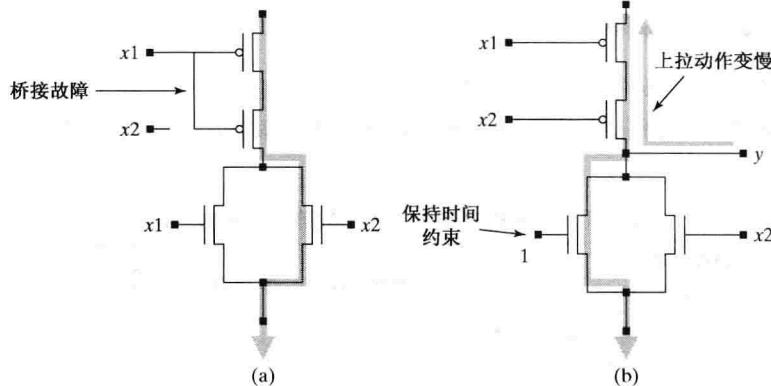


图 11.30 两输入或非门：(a) 桥接故障；(b) 粘接故障

与电源短路引起的故障称粘接 1 故障，与地线短路引起的故障称粘接 0 故障。电路中一个给定节点固定在逻辑 1 上，记为 $s\text{-}a\text{-}1$ ，固定在 0 上记为 $s\text{-}a\text{-}0$ 。存在 $s\text{-}a\text{-}1$ 和 $s\text{-}a\text{-}0$ 就足以危害电路的逻辑功能的实现了。

例 11.16 对图 11.31 所示的三输入与非门，(a) 中为无故障，(b) 中的一个输入有 $s\text{-}a\text{-}0$ 故障，(c) 中的一个输入有 $s\text{-}a\text{-}1$ 故障。无故障电路的组合门逻辑为 $y_{good} = (x_1 x_2 x_3)'$ ，但是当 x_1 有 $s\text{-}a\text{-}0$ 故障时， $y_{x_1 s\text{-}a\text{-}0} = 1$ ，电路的输出也粘接在逻辑 1 上。当 x_1 有 $s\text{-}a\text{-}1$ 故障时， $y_{good} = (x_1 s\text{-}a\text{-}1) = (x_2 x_3)'$ (即实现了两输入与非逻辑)。故障电路与无故障电路实现的逻辑是不同的^①。

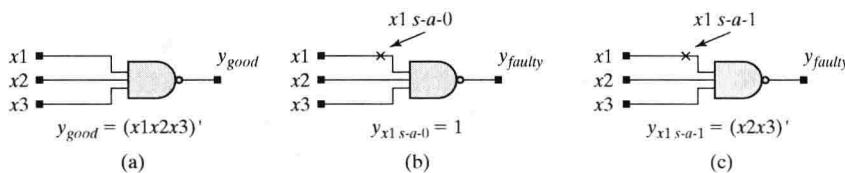


图 11.31 三输入与非门：(a) 无故障；(b) x_1 有 $s\text{-}a\text{-}0$ 故障；(c) x_1 有 $s\text{-}a\text{-}1$ 故障

逻辑电路中所有与逻辑门的输入和输出相连的线网都有可能发生 $s\text{-}a\text{-}0$ 和 $s\text{-}a\text{-}1$ 故障。粘接故障使已实现逻辑中的变量或者蕴含项消失，或者使该逻辑功能限于某个固定值。因此，测试应该能够检查出无故障电路与有 $s\text{-}a\text{-}0$ 或 $s\text{-}a\text{-}1$ 故障电路在逻辑功能上的不同。图 11.31(b)中， $x_1 = 1, x_2 = 1, x_3 = 1$ 时可检测出 $s\text{-}a\text{-}0$ 故障，如果 $y_{x_1=1, x_2=1, x_3=1} = 1$ ，将检查出有故障，因为

^① 注意，图 11.31(b)中的内部桥接故障影响了由 x_1 驱动的下拉晶体管，而对于相应的上拉晶体管没有影响。在 x_1 处的粘接故障则对上拉、下拉两个晶体管电路均有影响。

$y_{x1=1, x2=1, x3=1} = 1$ 与 $y_{good} = 0$ 不一致。同样，若图 11.31(c) 中， $x1 = 0, x2 = 1, x3 = 1$ 时可检测出 $s-a-1$ 故障，也是因为当有故障存在时， $y_{x1=0, x2=1, x3=1} = 0$ 与 $y_{good} = 1$ 不一致。

图 11.31 中，在逻辑门输入端加激励信号并在输出端进行监测，三输入与非门的故障可以容易地被检测到。这种测试方案并不具有代表性，因为电路中的绝大多数逻辑门的输入和输出并不能直接从芯片的引脚处测得。另外，一个芯片最多有几百个引脚，而内部可能存在数百万个故障点。尽管如此，测试单个器件所用的原理同样也可用来研究如何测试芯片内嵌的故障点。以下两点同时满足时，可以检查内嵌故障：

- (1) 原始输入能够断言(确定)故障点的逻辑值；
- (2) 该输入与那些能把故障传输到原始输出端的输入是兼容的。

11.6.2 故障检测与测试

如果在组合逻辑电路中， $s-a-0$ 和 $s-a-1$ 故障是可测的，则可给输入端施加一系列已知的测试模板，并观察有故障的电路和无故障的电路的输出是否有所不同。图 11.32 给出了测试电路粘接故障的基本原理图。给无故障电路和在特定位置注入故障的电路施加同样的输入信号，将这些电路的输出结果进行比较，可以判断是否存在差别。如果有错误的信号出现，则该施加的输入信号可以作为注入故障电路的测试模板；如果没有差别，则该模板不能区别有故障和无故障电路。

例 11.17 图 11.33(a) 所示电路中， $x1$ 的 $s-a-0$ 故障的测试模板列于图 11.33(b) 中。当测试模板加到电路的输入端时，故障电路和无故障电路的输出不同，因此测试模板 $(x1\ x2\ x3) = (1\ 1\ 1)$ 能够检测到故障，是对 $x1$ 的 $s-a-0$ 故障的测试。

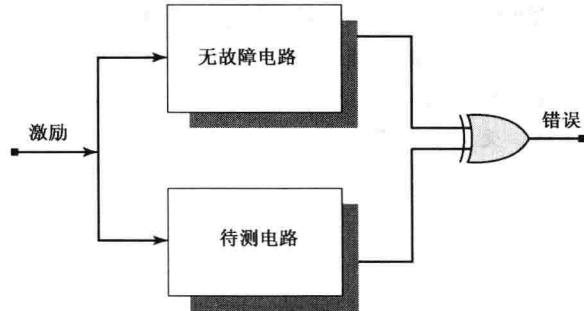


图 11.32 对无故障电路和有故障电路的响应进行比较的故障模拟框图

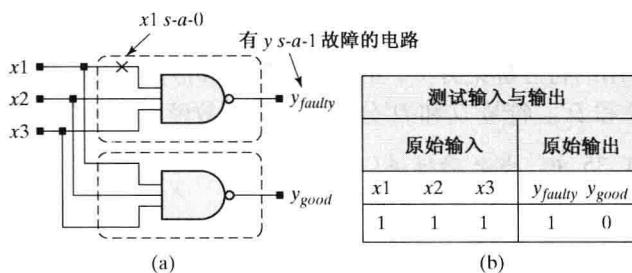


图 11.33 对 $x1$ 的 $s-a-0$ 故障的测试显示出有故障和无故障电路的输出不同

注意到，如果 $x1$ 粘接到逻辑 0，图 11.33 中电路的输出会是 1，输出将与输入无关。通常用一个能判断故障点(无粘接时)的激励源来对某一故障进行测试，并将故障电路和无故障电路的输出进行比较。如果输出不一致，该激励模板将检测出制造的电路中的此类故障。

组合逻辑电路中的 $s-a-0$ 和 $s-a-1$ 的故障测试需要一个信号输入模板，该测试可按如下四步进行：

- (a) 在电路中注入一个故障。

(b) 验证该故障(即加入激励, 在故障点得到逻辑值)。

(c) 激活一条或多条通路将故障的影响传输到输出。

(d) 将激活的输出与无故障电路的输出进行比较。

为了测试组合逻辑电路的故障, 电路的原始输入必须选择为:

(1) 能够确定(断言)故障点的假定值;

(2) 故障的影响可以传输到基本输出端(即故障可以激活输出)。

如果该测试想要检测某个节点是否粘接在逻辑1的情形, 则应设计一个原始输入能使无故障电路中与故障节点相对应节点的值为0。同时, 这个原始输入还能将故障点的信号值传输到原始输出端。当激活的输出与无故障电路输出不同时, 则该测试就检测出有故障存在。因为一种测试可能会检测出不止一个故障, 因此能测出故障的测试不一定要将故障的位置隔离在某一特定点。

例 11.18 图 11.35 所示的两个相同电路, 其中一个电路 x_1 有 $s-a-0$ 故障, 被称为故障电路。为了检测故障, 激励模板将 x_1 的值置为 1 来调整故障。与非门的其余输入(x_2, x_3)全置为 1, 使得与非门的输出对 x_1 敏感。将或门的另外输入(x_0)置为 0, 来激活传输 x_1 的值到电路原始输出的通路。无故障电路 $y_{good} = 0$, x_1 有 $s-a-0$ 故障的电路 $y_{faulty} = 1$ 。激励模板 $(x_0 \ x_1 \ x_2 \ x_3) = (0 \ 1 \ 1 \ 1)$ 是对 x_1 的 $s-a-0$ 故障的一种测试。

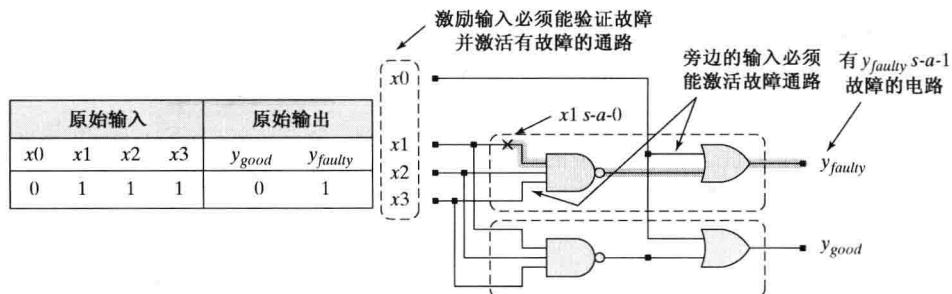


图 11.34 测试模板 $(x_0 \ x_1 \ x_2 \ x_3) = (0 \ 1 \ 1 \ 1)$ 通过验证故障和激活故障传输通路来检测 x_1 的 $s-a-0$ 故障

11.6.3 D 标记法

D 标记法是一种专用的符号标记方法, 用于测试生成和故障检测。在 D 标记法中, 电路中信号的逻辑值被标记为 D 和 D' 。符号 D 和 D' 分别代表在无故障电路中信号值为 1 和 0 的情况。

例 11.19 在图 11.35 中, 当激励模板 $(x_1 \ x_2 \ x_3) = (1 \ 1 \ 1)$ 时, 信号 x_1 的值为 D , 表示在无故障电路中 $x_1 = 1$, 有 $x_1 s-a-0$ 故障电路中 $x_1 = 0$ 。同样, 无故障电路的输出为 $y_{good} = D'$, 故障电路的输出为 $y_{faulty} = D$ 。

当多个故障能被同样的测试模板检测出来并且要减少必须施加于电路的测试模板的数量时, D 标记法对故障检测是十分有用的。在需要用少量的测试模板检测大量的故障时的应用场合。D 标记法是适合的。

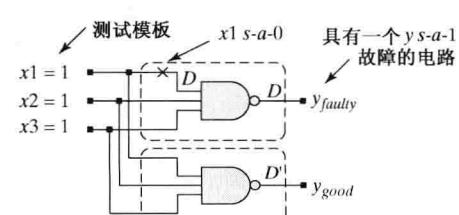
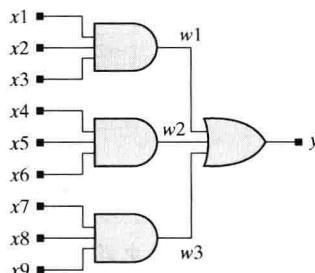


图 11.35 D 标记法表示在无故障电路中线网值为 1, 在故障电路中值为 0

例 11.20 图 11.36(a) 中的两级电路在其 9 个基本输入端、3 个内部线网和 1 个原始输出上

有故障点。图 11.36(b)和(c)中的表格用 D 标记法标出了测试模板^①指定的情况下内部线网和原始输出线网的值。表格已进行了注释以便于识别由测试模板检测出来的故障。测试是在假定电路中只有一个故障的情况下进行的(所谓单粘接故障模型^[6])，但是给定的激励模板可能会检测到多个故障。在这个电路中分别只需 3 个测试就能检测出所有可能的 $s-a-1$ 故障，而另外的 3 个测试能检测出所有可能的 $s-a-0$ 故障。



(a)

s-a-1 故障测试						
#1	site	#2	site	#3	site	
x_1	0	x_1	1		1	
x_2	1		0	x_2	1	
x_3	1		1		0	x_3
x_4	0	x_4	1		1	
x_5	1		0	x_5	1	
x_6	1		1		0	x_6
x_7	0	x_7	1		1	
x_8	1		0	x_8	1	
x_9	1		1		0	x_9
w_1	D'	w_1	D'	w_1	D'	w_1
w_2	D'	w_2	D'	w_2	D'	w_2
w_3	D'	w_3	D'	w_3	D'	w_3
y	D'	y	D'	y	D'	y

(b)

s-a-0 故障测试						
#1	site	#2	site	#3	site	
x_1	1	x_1	0		x	
x_2	1	x_2	x		x	
x_3	1	x_3	x		0	
x_4	0			1	x_4	x
x_5	x			1	x_5	x
x_6	x			1	x_6	0
x_7	0			0		x_7
x_8	x			x		x_8
x_9	x			x		x_9
w_1	D	w_1	D'		D'	
w_2	D'		D	w_2	D'	
w_3	D'		D'		D	w_3
y	D	y	D	y	D	y

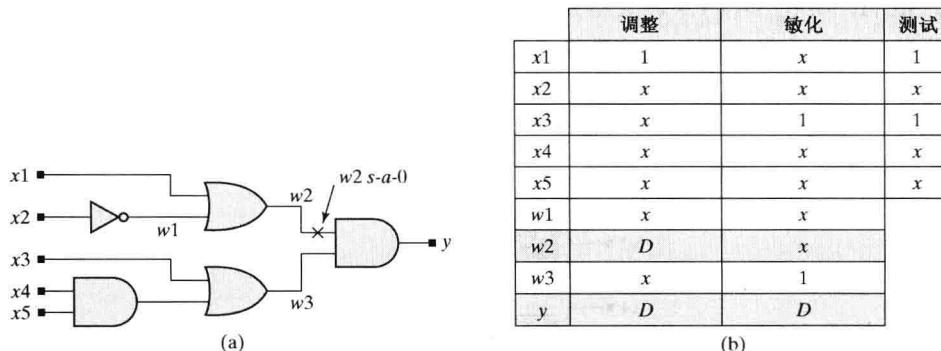
(c)

图 11.36 激励模板(b)和(c)可以检测出电路(a)中所有的 $s-a-1$ 故障和 $s-a-0$ 故障

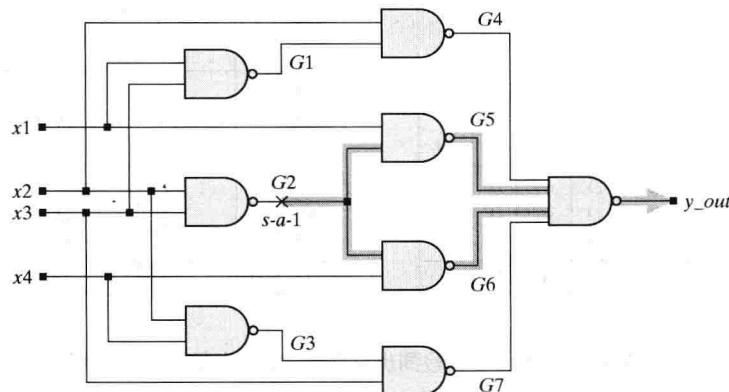
例 11.21 当用一种测试来检测图 11.37(a)所示的多级组合逻辑电路中的 w_2 的 $s-a-0$ 时，用 D 标记法标记的信号值列于图 11.37(b)的表格中。在所有的故障测试模板中，其中一个为 $(x_1\ x_2\ x_3\ x_4\ x_5) = (1\ x\ 1\ x\ x)$ ，这里的 x 为任意值。只要激活故障的原始输入值不与验证故障的值发生冲突，该测试模板就可以检测到故障。无故障电路的输出为 $y=1$ 。

多级网络中的故障激活可以通过追踪从故障点到基本输出端的前向通路来完成，并将通路上所有逻辑门的其余输入值置为其激活值(例如，与非门的其余输入端置为 1)。然后通过追踪从其他输入到基本输入的反向通路，来建立从故障点到原始输出端的可以激活通路的原始输入值。这一步称为线性辨识。通常还必须考虑多通路激活问题，因为单通路的激活可能无法产生对某一可测故障的测试。

① 在本书中，符号 x 表示任意信号值。

图 11.37 故障检测: (a)组合逻辑电路;(b)为了验证和激活 $w2\ s-a-0$ 故障的 D 标记法的信号值

例 11.22 图 11.38 中的电路(称为 Schneider 电路^[6])可以通过设定 $x2 = 1$ 和 $x3 = 1$ 来检测 $G2$ 的 $s-a-1$ 故障。为了激活从故障点到 y_{out} 的通路, 设定 $x1 = 1$, 以驱动 $G5$ 的输出为 1。前面对 $x1$, $x2$ 和 $x3$ 的选择与形成 y_{out} 的与非门的其他输入置为 1 是一致的, 但是将 $G6$ 置为 1 则要求将 $x4$ 置为 0, 因为 $G2$ 的值是未知的。当 $x3 = 1$, $x4 = 0$ 时, $G7$ 的值又变为 0, 从而阻塞了故障通路。如果选择通过 $G6$ 的单一通路来传输故障, 相同条件仍将阻塞故障向输出端传输。因此, 只激活从故障点到电路输出的单一通路不能检测到 $G2$ 的 $s-a-1$ 故障。只有用激励模板($x1\ x2\ x3\ x4$)=(1 1 1 1), 同时激活通过 $G5$ 和 $G6$ 的通路时才能检测到该故障。

图 11.38 故障 $G2\ s-a-1$ 不能通过一个通路激活

11.6.4 组合电路的自动测试模板生成

依靠人工方法进行大规模组合电路的测试是很不可取的。图 11.37 中的故障 $w2\ s-a-0$ 测试之所以能够通过人工方法进行, 是因为该电路比较简单。测试组合逻辑电路的另外一种方法是给电路施加所有可能的输入, 并且观察电路的输出是否与无故障电路输出相同。虽然枚举所有测试模板的方法比较简单, 但当电路中有大量的输入时就变得不适合且无益处了。在例 11.2 中可以看到只需要 6 组测试模板就能检测到全部的故障, 枚举所有测试模板会产生 512 组测试模板。

有这样一些算法, 它们能够自动、高效地找到或构建少量的测试模板, 这些少量的测试模板能够检测大部分组合逻辑电路的故障。用于生成测试模板的商用工具包含多种算法不同的特性。前面提到的 D 标记法的 D 算法^[7]应用比较广泛。它用于测试内嵌在组合电路内部所有信号线的粘接故障。D 算法已经被集成在自动生成测试模板(ATPG)的软件中, 它采用多路径激活方法,

并且保证在发现组合电路逻辑故障的测试模板存在的情况下，能够找到这个测试模板，但是当电路中存在大量的异或门时，寻找测试模板的效率将显著降低。

还有两种可选择的算法，一种是面向通路判决法 PODEM (path-oriented decision making)^[8]，另一种是面向扇出的测试模板生成算法 FAN(fanout-oriented test-generation algorithm)^[9]，这两种算法要比 D 算法效率更高。PODEM 算法从电路的基本输入向前推算，取代 D 算法中的交替追踪和向前传输的步骤。FAN 算法采用附加策略来减少反复追踪并且比 PODEM 算法更有效。关于这些算法请参见 Abramovici et al.^[6], fujiwara 与 shimono^[10] 的相关讨论。

ATPG 能找到一个检测给定故障的测试模板但却不能对存在再汇聚扇出的组合电路进行测试。有些故障是不可测的，不可测的原因有以下几点：

- (1)冗余逻辑(见本章习题 14)^①;
- (2)不可控的线网;
- (3)不规则的线网。

当故障所在的线网不可控，或者没有输出通路能够被激活用来观察故障时，这种故障是不能被检测的。

例 11.23 图 11.39 中对故障 $w2\ s-a-0$ 的测试就不存在。信号 $x5$ 在 y 处存在扇出重汇聚，所以由 $x5$ 所影响的其余输入不能独立设置。 $w3$ 的激活需要将 $x5$ 置为 1，但是 y 的激活则需要将 $x5$ 置为 0。表 11.4 列出了使用 D 标记法的信号值，而且也列出了激活 $w3$ 和 y 所需要的基本输入值之间的冲突， $x5$ 的扇出重汇聚迫使 y 为 0，而与故障 $w2\ s-a-0$ 无关。这种测试无法区分故障电路和无故障电路。

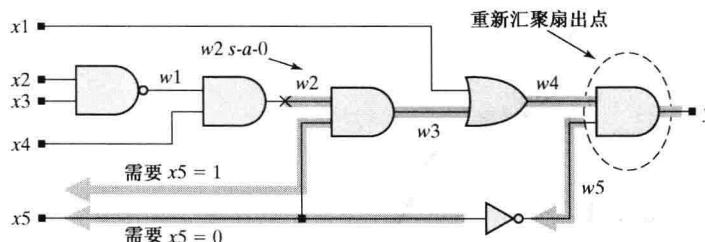


图 11.39 对于故障 $w2\ s-a-0$ 的测试不存在。 $x5$ 的扇出重汇聚使测试故障 $w2\ s-a-0$ 的条件处于矛盾状态

表 11.4 扇出重汇聚使激活故障 $w2\ s-a-0$ 所需的 $x5$ 的值之间产生冲突

	调 整	激 活	测 试
$x1$	x	0	0
$x2$	0	x	0
$x3$	x	x	x
$x4$	1	x	1
$x5$	x	$w3$ 为 1 y 为 0	冲突
$w1$	1	x	
$w2$	D	D	
$w3$	x	D	
$w4$	x	D	
$w5$	x	冲突	
y	x	0	

① 虽然综合工具能够消除冗余逻辑，但综合后的电路可以通过增加冗余逻辑来提高电路的速度或消除竞争。这种修改对测试模板生成是有影响的，因为它会导致测试模板生成的效率下降。

11.6.5 故障覆盖和缺陷级别

故障测试确保了交付给客户的产品质量。交付有缺陷器件的概率 W 与测试覆盖 T 和相对制造成品率 Y 的关系式为：

$$W = 1 - Y^{(1-T)}$$

此处， Y 表示生产芯片的 ASIC 制造过程中的相对成品率(例如 $Y=0.75$)， T 表示故障测试覆盖率^[10,11]。高的相对成品率和故障测试覆盖率是人们所期待的。表 11.5 和图 11.40 说明了在给定的成品率范围内，平均缺陷率是如何由故障覆盖率决定的。为了达到提高覆盖率的目的，该曲线给出了一个定量的估测。对于半导体厂商来说，保证电路中故障覆盖率要超过 99% 是很平常的事。

表 11.5 未检测的有缺陷部件的质量取决于制造工艺的成熟程度和测试模板覆盖率，不成熟的工艺明显地需要非常高的故障覆盖率以降低平均缺陷级别

生产成品率	测试模板覆盖率		
	70%	90%	99%
未测试缺陷的百分比			
先进工艺	10%	50%	21%
正在完备中的工艺	50%	19%	7%
完备工艺	90%	3%	1%

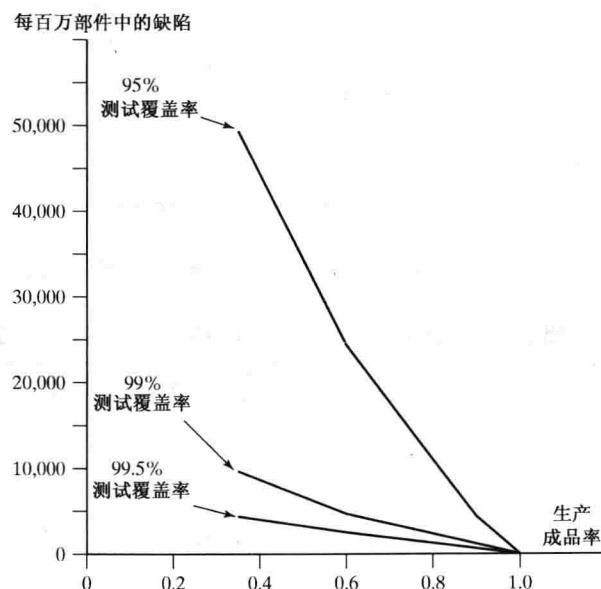


图 11.40 每百万部件中的缺陷率相对于测试覆盖率和生产成品率的关系曲线

11.6.6 时序电路的测试生成

对时序电路进行直接生成是很困难的，其原因是该测试需要很长的输入信号序列来驱动内部时序器件达到一个已知状态，这个已知的状态能够验证故障点或激活一条通路。想要验证两个时序电路具有同样的功能(即通过施加输入序列并观察输出序列说明两个电路是等同的)是不切实际的，一种可选的方法是将时序电路视为迭代网络。

例 11.24 图 11.41 中的电路需要三个激励模板组成的序列来检测故障 $w1\ s-a-1$ 。生成测试模板是按照相反的顺序进行的，先从周期 3 中必须有的线网的值开始，并从 y 处可以观测故障的影响，然后逆向推导确定周期 2 中必须存在的值，依次类推，让故障的影响可以在最后一个周期中被观测到。表 11.6 显示了测试的每一个周期中的信号值。

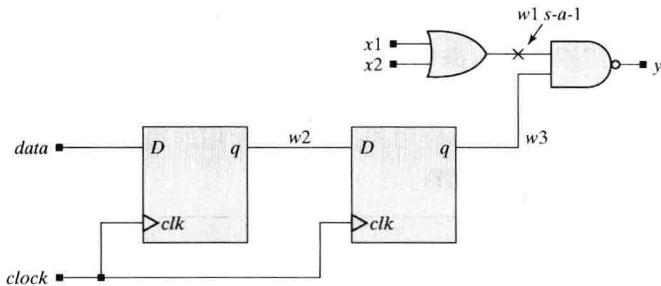


图 11.41 三组测试模板组成的序列可检测到故障 $w1\ s-a-1$

表 11.6 用于测试图 11.41 中的故障 $w1\ s-a-1$ 的测试模板序列。该测试序列必须传输一个 1 信号经由移位寄存器（以激活输出 y ）到达故障处

		验 证	激 活	测 试
周期 3	$w1$	D'	D'	
	$w2$	x	x	
	$w3$	x	1	
	$x1$	0	x	0
	$x2$	0	\bar{x}	0
	$data$	x	x	x
	y	D	D	D
周期 2	$w1$	x	x	
	$w2$	x	1	
	$w3$	x	x	
	$x1$	x	x	x
	$x2$	x	x	x
	$data$	x	x	x
	y	x	x	x
周期 1	$w1$	x	x	
	$w2$	x	x	
	$w3$	x	x	
	$x1$	x	x	x
	$x2$	x	x	x
	$data$	x	1	1
	y	x	x	x

由于为时序电路直接生成测试模板是十分困难的，因此通常考虑采用路径扫描的方法。电子行业中经常用扫描的方法验证电路，将用于组合电路的测试方法用于时序电路，实现其可观测性。

路径扫描有多种方法，方法的取舍取决于电路的寄存器在扫描链路中的连接状况。ASIC 使用的是部分或者完全扫描方法，取决于在其内部的部分或者全部触发器是否都可以用扫描单元替代，并且连接起来形成一个或多个可由外部测试器控制的移位寄存器。而采用部分扫描方法其实是一种折中，它平衡了扫描链路所提供的故障覆盖率和形成扫描链路所需要的附加逻辑之间的关系。

扫描设计是用扫描触发器替代普通的触发器来构成一个双端口寄存器，即所谓的扫描寄存器，它使电路更加可控和/或可观测。扫描寄存器可以通过串行端口使数据移位，并通过并行端口装载数据。在测试模式中，测试模板的逻辑值移入触发器中。已装入触发器的逻辑值在一个时钟周期内驱动组合逻辑通路，并在下一个时钟周期内能并行捕获到该通路的目的端口的值。捕获的输出数据可以从寄存器移出并加以分析，来检测逻辑通路的内部故障。

图 11.42 给出了用一组双端口扫描单元连接起来构成一个四比特的扫描寄存器。正常工作情况下， $T=0$ 时数据经由 $D[3:0]$ 并行装入。在测试模式 $T=1$ 周期内，数据通过寄存器从 $x_in3_scan_in$ 移位到 $y3_scan_out$ 。根据具体应用和扫描单元使用的程度，通过串行扫描通路，逻辑电路的内部节点可做到 100% 的可控与可观测。

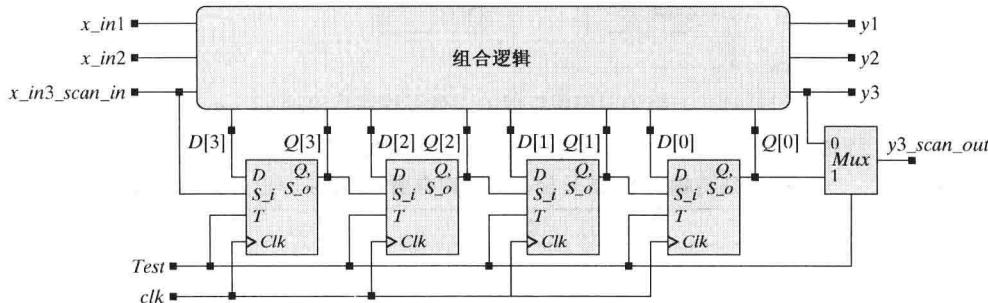


图 11.42 由双端口寄存器构成的扫描寄存器

完全扫描是用扫描单元替代设计中的全部触发器。边界扫描是将扫描单元放在 ASIC 的 I/O 端口处，并把它们连接起来构成一个边界扫描链路用于板级测试。由于具有完全扫描特性的 ASIC 核的重构电路具有如图 11.43 所示的结构，其测试模板与用来测试其组合逻辑电路的测试模板一样。使用完全扫描的电路测试过程是先测试扫描通路，然后再测试组合逻辑。扫描通路的测试是将电路置于测试模式 ($T=1$) 并触发时钟 $n+1$ 次，以通过链路传输测试序列。使电路处于

测试模式，将测试模板的逻辑值移入扫描寄存器，并施加原始输入信号，为测试组合逻辑电路做好准备。把测试模板施加到扫描寄存器后，模式被置为正常状态 ($T=0$)，并在基本输出端和扫描寄存器的输入端观察到电路的响应。时钟再次触发，将并行输入锁存到扫描寄存器中。然后电路又将置于测试模式下，并将已捕获的样本从寄存器中移出用于分析。与此同时，另一组模板被移入寄存器。

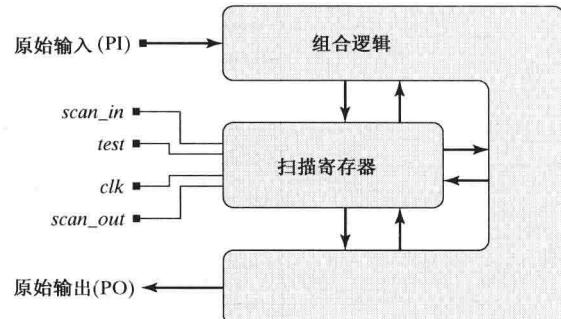


图 11.43 完全扫描测试的电路结构

11.7 故障模拟

故障模拟是将故障电路的运行状态与无故障电路的运行状态进行比较。如果施加相同的激励模板，两电路的输出不同，该激励模板称为故障检测的测试模板。故障模拟决定了给定激励模

板(测试模板)能检测故障的程度。故障模拟现在已经集成在 ATPG^①的工具中。不过,这里将讨论一些主要的概念,它对理解整个设计流程中的重要方面将有所帮助。

下面只考虑测试组合逻辑中的单粘接故障。故障覆盖率,即一组激励模板能够检查电路中可能存在的故障的程度,可由以下表达式定义:

$$\text{故障覆盖率} = \frac{\text{检测到的故障数}}{\text{故障总数}}$$

为了确保交付给用户的器件没有缺陷,提供的测试模板必须有很高的故障覆盖率,这一点很重要。故障等级^②通过检查故障是否可测、是否能够找到该故障的一个测试模板来确定测试覆盖范围。大多数应用中对于电路包含的单粘接故障,故障测试的范围必须大于99.5%。

故障模拟器可通过搜集故障点、注入故障、施加激励模板,并将该电路与无故障电路的输出比较来检测电路故障。不能检测故障的激励模板是不可用的。在具有高覆盖率的激励模板测试模板生成的配合下,故障模拟器进行故障检测。有多种方法来减少测试生成所需要的时间和精力。

11.7.1 故障解析

测试的效率会影响昂贵的测试仪器的折旧摊销。不必要的测试会浪费测试仪器的资源和寿命,因此只测试那些需要测试的故障,并能尽快找出能检测尽可能多的故障的测试模板是很重要的。高效的故障模拟器使用故障分类形成等价故障类,等价故障类可用相同的测试模板检测。通过相同测试模板检测到的故障是没有区别的,其称为等价故障。故障模拟器只在一个等价故障类中测试一个故障,避免了对其他同类故障进行不必要的仿真。一类等价故障可用相同的测试模板测试,所以等价故障类只需检测其中的一个故障即可。

例 11.25 图 11.44 中故障 $x1\ s-a-0$ 与故障 $y\ s-a-1$ 是无法区分开的,所以检测 $x1\ s-a-0$ 的测试模板同样能够检测 $y\ s-a-1$ 。两个故障属于等价故障。

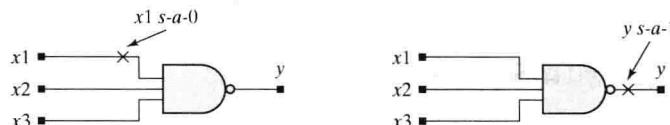


图 11.44 故障 $x1\ s-a-0$ 与故障 $y\ s-a-1$ 为等价故障

有三种主要的故障仿真方法:串行故障模拟、并行故障模拟和并发性故障模拟^[5]。串行故障模拟是三种方法中最慢的一种,但也是最容易理解和实现的一种方法。

11.7.2 串行故障模拟

串行故障模拟一次只考虑一个电路故障,并按照下列步骤确定所采用的激励源是否能够发现故障:

- (1) 创建故障点列表;
- (2) 将一个故障引入到电路中并将其从列表中去除;
- (3) While(故障列表不为空);

{

施加激励模板并对无故障电路和引入故障的电路进行仿真;

比较两个电路的输出;

① 参见 www.synopsys.com 网站的 TetraMAX ATPG 工具。

② 也称故障覆盖率分析。

```

if(输出存在差异)
    该模板检测出故障;
else
    该模板没有检测到故障;
引入另一个故障并将它从列表中去除;
}

```

11.7.3 并行故障模拟

并行故障模拟同时对多个副本电路进行仿真，每一个副本电路都引入不同的故障，并将其响应与无故障电路的响应加以比较。虽然它比串行故障模拟要快，但需要更多的存储空间和高效的存储管理技术来处理多个电路。

11.7.4 并发性故障模拟

并发性故障模拟是使用最广泛的算法。它将故障模拟的范围限制在部分电路中，从故障点后向验证(扇入)和从故障点前向激活(扇出)。并发性故障模拟要求对电路进行完善的拓扑分析来限制用于验证和激活故障的搜索范围。其结果是并发性故障模拟要比串行故障模拟和并行故障模拟的速度更快。

11.7.5 概率性故障模拟

还有另外一种故障模拟的方法，称为概率性故障模拟，它可以识别具有高触发覆盖率的测试模板，并将它们作为检测故障的测试模板的基础。触发电路中大量节点的测试模板与能检测大量故障的测试模板之间具有很高的相关性^[4]。触发测试比其他故障检测方法更简单，运行速度也更快。

11.8 JTAG 端口^①和可测性设计

可测性设计(Design For Testability, DFT)确保对已制造的电路能够进行缺陷测试。DFT 通常要求设计的电路能够支持测试，因为一般芯片上可用的 I/O 引脚是相当少的，要测试内部的大量节点，这些引脚是不够的。通过衡量控制和观察内部节点的难易程度来评价一个芯片的可测性，这样的方法有很多^[12]。虽然也有一些能改善电路可测性的方法^[6]，但这里只关注基于扫描的方法，这种方法扩展了 11.7.6 节中所介绍的测试时序电路的概念。这样做是因为基于扫描的方法已经被广泛地应用并且十分重要，它们不仅支持对电路缺陷的测试，而且支持在软件开发中对嵌入式处理器的调试，以及支持 CPLD 和 FPGA 的现场编程。

在芯片和板级测试中存在一些实际的问题：

- (1) 时序电路难以进行测试，因为测试需要一组测试模板序列；
- (2) 大规模电路的内部节点不能在输出引脚处观测，也不容易通过可用的输入引脚对它们进行控制；
- (3) 印制电路板的制作过程使用铜线作为信号通路，如果这些铜线被短接或开路，电路就会存在缺陷；
- (4) 电路板与 ASIC 芯片引脚之间或引脚与核心逻辑之间可能会接触不良；

^① JTAG 端口是以制定 IEEE 1149.1 和 IEEE 1149.1a 标准的行业专家组 Joint Test Action Group 命名的。

- (5) 安装到电路板上的芯片的核心逻辑只能现场测试，而不能将其从电路板上拿下来单独测试；
 (6) 有必要将故障位置与特定的 ASIC 或模块隔离，以减小维修成本。

通过采用一种基于扫描链的标准的电路接口对板级和芯片级电路进行测试，电子行业界已经解决了这些问题。

11.8.1 边界扫描和 JTAG 端口

边界扫描是为了测试时序电路而对 11.7.6 节中所讨论的扫描寄存器概念进行的扩展。通过在 ASIC^① 的 I/O 引脚处插入边界扫描单元(Boundary Scan Cells, BSC)，并在芯片周围将它们连接成移位寄存器，从而将扫描链路加入到 ASIC 的网表(netlists)中。同样的单元也可以用来替代核心逻辑电路中的触发器，以构成内部扫描路径，这条内部的扫描路径由一个或多个测试数据的寄存器(test-data register)级联而成。当在内部使用时，这些单元被称为数据寄存器(Data Register, DR)单元。

一个典型的 BSC 或 DR 单元如图 11.45 所示。这些单元允许在不影响芯片正常工作的情况下扫描数据(例如在线监测芯片的工作)。两个多路选择器控制了单元的数据通路。输入多路选择器决定了捕获/扫描触发器是连接到 *data_in* 还是连接到串行输入 *scan_in*。输出多路选择器决定了是把 *data_in* 还是输出触发器连接到 *data_out*。

mode = 0 时单元处于正常模式，*data_in* 通过多路选择器传输到 *data_out* 和捕获/扫描触发器，由 *clockDR* 脉冲处加载数据。捕获/扫描触发器支持边界扫描链路；当新数据被扫描进入捕获/扫描触发器时，输出寄存器的数据保持不变。BSC 的 *data_in* 和 *data_out* 分别连接到 ASIC 核心逻辑的输入和输出。在测试模式下，测试模板在 *clockDR* 的控制下被移入捕获/扫描触发器。若扫描链路处在正常状态，通过 *updateDR* 信号触发，捕获/扫描触发器中的数据可以并行装入，以更新输出寄存器。

如果 BSC 寄存器单元连接到芯片的输入引脚，则 *data_in* 连接到芯片的输入引脚，*data_out* 连接到 ASIC 核心逻辑的输入引脚。如果 BSC 寄存器单元作为输出，则 ASIC 核心逻辑与 *data_in* 相连，并且 *data_out* 连接到 ASIC 的输出引脚。BSC 单元的 Verilog 模型如下：

```
module BSC_Cell (output data_out, output reg scan_out,
  input data_in, mode, scan_in, shiftDR, updateDR, clockDR
);
  reg update_reg;

  always @ (posedge clockDR) begin
    scan_out <= shiftDR ? scan_in : data_in;
  end

  always @ (posedge updateDR) update_reg <= scan_out;
  assign data_out = mode ? update_reg : data_in;
endmodule
```

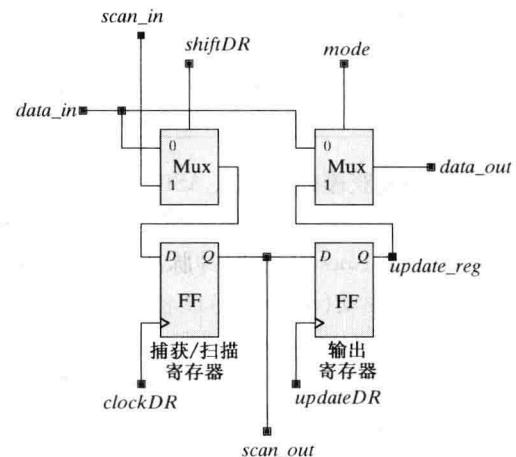


图 11.45 用来实现边界扫描测试寄存器和测试数据寄存器的 DR 单元包括一个捕获/扫描触发器和一个输出触发器

① 这里将讨论 ASIC 中的扫描链路，但是它们也可用于 FPGA 和其他器件中。

11.8.2 JTGA 操作模式

表 11.7 中总结了边界扫描单元(数据寄存器单元)的操作方式。在正常模式下($mode = 0$)数据直接通过单元从 $data_in$ 传输到 $data_out$ 。驱动 $data_out$ 的多路选择器在信号通路上增加了可忽略的传输延时。在测试模式下, $mode = 1$, 输出寄存器($update_reg$)驱动单元的 $data_out$ 。

在扫描模式下, 把一个单元的 $scan_out$ 与链路中下一个单元的 $scan_in$ 相连, 边界扫描单元便连接成移位寄存器。测试模板能够移入寄存器中, 随后装载进输出寄存器, 在 $data_out$ 处建立逻辑值, 这些逻辑值是用来测试核心逻辑的。当 $shiftDR = 1$ 时, 数据在 $clockDR$ 的上升沿经过捕获/扫描寄存器, 从 $scan_in$ 移到 $scan_out$ 。

表 11.7 BSC 的操作模式

模式	操作
通常	当 $mode = 0$, $data_in$ 直接连到 $data_out$, 扫描链不影响 ASIC 工作
扫描	当 $shiftDR = 1$, 数据在 $clockDR$ 的有效沿通过 $scan_in$, 并通过 $data_out$ 输出
捕获	当 $shiftDR = 1$, 在 $clockDR$ 的有效沿, 数据被载入捕获寄存器
更新	当 $mode = 1$, 在 $updateDR$ 的有效沿捕获寄存器的输出被移入更新寄存器

单元的捕获模式可以从 ASIC 中获取数据而不打断它的操作。当芯片正在工作时, 数据可以在稍后被扫描出来。该模式要求工作在 $shiftDR = 0$ 的条件下, 此时扫描通路与捕获/扫描寄存器相连。下一个 $clockDR$ 的时钟脉冲期间将 $data_in$ 装入到扫描寄存器。在这种模式下, $data_out$ 可以由 $data_in$ 驱动($mode = 0$)或者由输出触发器驱动($mode = 1$)。

更新模式下, $mode = 1$ 时, 由输出寄存器的内容来驱动 $data_out$ 。施加 $updateDR$ 脉冲, 将扫描寄存器的内容加载到输出寄存器。如果 $data_out$ 与 ASIC 的输入引脚相连接, 这个激励模板可以作为该芯片的一个测试模板。当 $shiftDR = 0$ 时, 芯片的响应可通过 $clockDR$ 脉冲获取。

边界扫描方法可以测试 PC 板上的多片芯片、芯片间的板上布线、芯片引脚和核心逻辑之间的连接等。测试器可以将内嵌有边界扫描电路和专用测试存取端口(TAP, 也称为 JTAG 端口)的 ASIC 核心逻辑模块分离出来, 并对其进行测试。TAP 控制器是一种有限状态机, 它能控制 TAP。JTAG 标准 IEEE 1149.1 [13] 和 IEEE1149.1a 中规定了 TAP 的实现方法。芯片的 JTAG 端口可以与另一个芯片的 JTAG 端口串行连接, 这样扫描链路可以把板上的所有芯片都连接起来。如果板上的芯片集成了 JTAG 端口和边界扫描链路, 测试器就可以测试电路板布线的开路和短路故障, 这些布线包括芯片的 I/O 引脚和电路板之间的布线, 以及 ASIC 的核心逻辑和引脚间的布线。外部测试器可以利用 JTAG 端口检测 ASIC 的内部故障。

JTAG 端口除了可以测试 ASIC 和印制电路板(PC)的制造缺陷之外, 还有很多其他用途。JTAG 端口可用来对可配置的 PLD^[14] 和 FPGA 器件编程^①, 还可以通过控制处理器和访问内部寄存器^②, 对嵌入式处理器的软件进行开发和调试。

11.8.3 JTAG 寄存器

每一个应用 JTAG 方法的芯片必须包括边界扫描寄存器(由连接 BSC 构成)、旁路寄存器

① 参见 www.altera.com 和 www.xilinx.com。

② 参见 www.agilent.com。

和指令寄存器，这些命名的寄存器如图 11.46 所示。旁路寄存器保存一个比特。指令寄存器和其他数据寄存器的大小可以根据应用设定。可以使用一个可选的 32 位宽的器件识别寄存器来存储数据，存储的数据包括器件的序号、制造厂商名和其他可由外部测试器读取的信息。指令寄存器的当前指令决定了哪个寄存器是连接在测试数据输入 (TDI) 与测试数据输出 (TDO) 之间的。实际的寄存器可以由内部扫描链路上的一个或多个测试数据寄存器 (TDR) 连接而成。

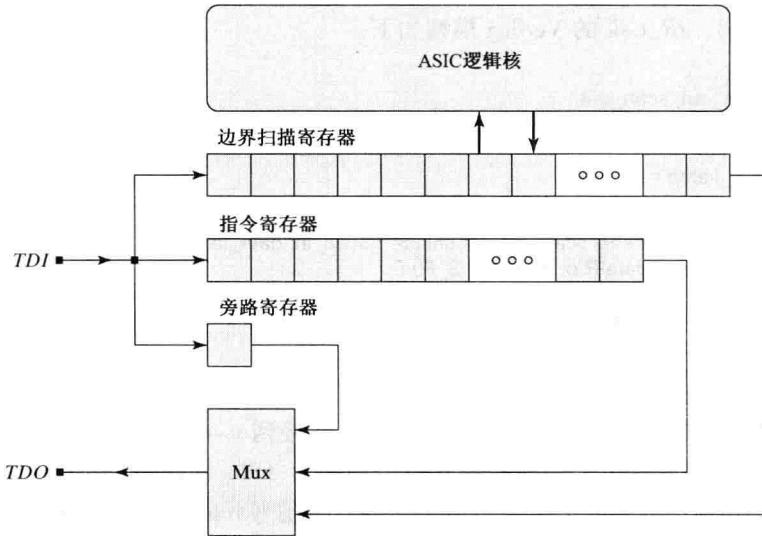


图 11.46 JTAG 规定的寄存器结构

单比特旁路寄存器是具有图 11.47 所示结构的单元。指令寄存器的单元结构如图 11.48 所示。在激励模板加到给定的扫描寄存器之前，旁路寄存器可以绕过 PC 板上扫描链路中的一个 ASIC，通过减少移位的次数来减少测试的长度。

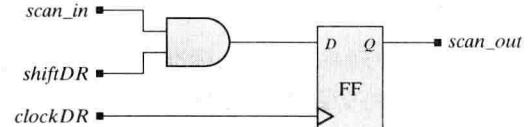


图 11.47 边界扫描的旁路寄存器 (BR) 单元

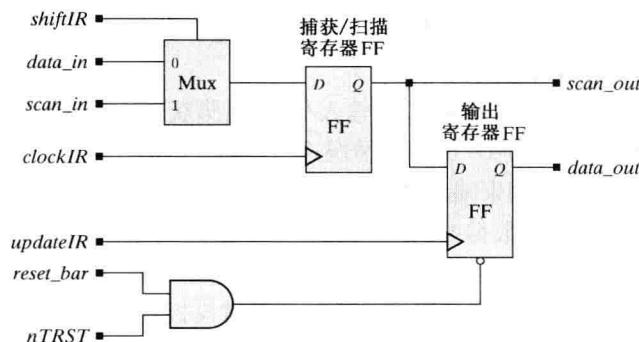


图 11.48 边界扫描的指令寄存器 (IR) 单元

旁路寄存器的 Verilog 模型由下面给出的 *BR_Cell* 模块来描述。其中，信号 *shiftDR* 门控扫描通路，*clockDR* 为单元提供同步脉冲。

```
module BR_Cell (output reg scan_out, input scan_in, shiftDR, clockDR);
  always @ (posedge clockDR) scan_out <= scan_in & shiftDR;
endmodule
```

指令寄存器指定了 TAP 内部指令和控制 TAP 的数据通路。指令定义了扫描操作过程中在 *TDI* 和 *TDO* 之间连接的一系列测试数据寄存器通路。指令寄存器单元具有异步置位/复位功能，根据当 TAP 状态机进入复位状态时保持的指令，可以通过参数 *SR_value* 进行编程来确定在输出端是 0 还是 1。指令寄存器具有通过 *scan_in/scan_out* 的串行输入/输出，以及通过 *data_in/data_out* 的并行输入/输出。*IR_Cell* 的 Verilog 模型如下：

```
module IR_Cell (
  output reg data_out, scan_out,
  input data_in, scan_in, shiftIR, reset_bar, nTRST, clockIR, updateIR
);
  parameter SR_value = 0;
  wire S_R = reset_bar & nTRST;

  always @ (posedge clockIR) scan_out <= shiftIR ? scan_in: data_in;
  always @ (posedge updateIR or negedge S_R)
    if (S_R == 0) data_out <= SR_value;
    else data_out <= scan_out;
endmodule
```

请注意，当输出寄存器中存储了一个指令时，则可将一个新的指令移送到扫描寄存器。信号 *shiftIR* 选择了该单元的输入数据通路，该通路可以是连接到 *scan_in* 的串行通路，或者是连接到 *data_in* 的并行通路。后者提供了一种能够在指令中包含 ASIC 的数据(如：状态位)的方法。当 TAP 控制器进入复位状态时，由 TAP 控制器同步地产生信号 *reset_bar*。*nTRST* 是可选的第 5 个异步输入端口，其低电平有效。

11.8.4 JTAG 指令

表 11.8 中列举了 JTAG 标准所规定的指令。BYPASS 指令扫描从 *TDI* 到 *TDO* 的数据，此处是指通过 1 比特旁路寄存器的而不是通过整个边界扫描链路的数据。这样就绕过了还没有被测试的芯片，并缩短了测试其他部件所必需的扫描链。

EXTEST(外部测试)指令可用来测试芯片外部的互连。将测试模板扫描进获取/扫描寄存器，然后再将该数据并行装入到边界扫描链路的输出寄存器。当芯片置于测试模式时，测试模板就出现在芯片的输出引脚，并驱动测试信号与其他芯片互连。其他芯片上的信号值被获取并扫描出来用以进行互连结构的完整性分析。

SAMPLE/PRELOAD 指令可以从 ASIC 的输入/输出引脚获取数据，而不影响其正常操作。已读取的数据可以扫描出来用以对芯片的工作情况进行分析。

INTEST(内部测试)指令可用来隔离并测试电路板上每个元件的内部电路。该指令把测试模板扫描进捕获寄存器，然后再将数据并行装入边界扫描链路的输出寄存器。当芯片处于测试模式时，连接到 ASIC 的输入端口的输出寄存器单元将对芯片的逻辑进行激活，芯片输出可从捕获/扫描寄存器单元获取，将其扫描出来并进行分析。在测试模板被扫描出来的同时，另一组测试模板扫描进寄存器。

由 TAP 所实现的指令代码的一部分由 JTAG 标准规定。BYPASS 指令的代码要求全为 1，EXTEST 指令的代码则全为 0。TAP 也可以包含可选的测试数据寄存器以进行内部扫描和其他测试。每一个测试数据扫描寄存器对应一个内部扫描链路，这条链路在 TAP 和指令寄存器的控制下可由外部测试器对其激活。

表 11.8 IEEE 标准 1149.1 所规定的指令

指令	行为方式
BYPASS	通过单一单元旁路寄存器传输数据，绕过 ASIC 的边界扫描链路，缩短了测试其他元件所必需的扫描链路
EXTEST	将已知值驱动到 ASIC 的输出引脚，测试电路板级连接和 ASIC 的外部逻辑模块
SAMPLE/PRELOAD	SAMPLE 获取系统引脚的数据值，将数据并行装入捕获寄存器。PRELOAD 将测试数据放入输出寄存器
INTEST*	对 ASIC 逻辑模块应用测试模板，并从逻辑模块获取响应。仅在 TDI 和 TDO 之间连接扫描寄存器
RUNBIST*	当 TAP 控制器处于 <i>S_Run_Idle</i> 时，主 ASIC 进行自检。
IDCODE*	将 IDCODE 寄存器(器件识别寄存器)中的数据移出，给测试器提供制造商名称、部件序列号和其他数据。如果在 TAP 中不存在 IDCODE 寄存器，指令将默认送到 BYPASS 寄存器

* 表示可选指令。

11.8.5 TAP 结构

TAP 的结构如图 11.49 所示。为了符合 JTAG 的要求，TDI，TMS 和 nTRST 输入端口具有上拉电路，比如 TDI 输入端悬空，则未被驱动的输入将产生与输入逻辑 1 相同的响应，这也隐含着 TAP 控制状态机的运行状态，我们将在后面讨论。

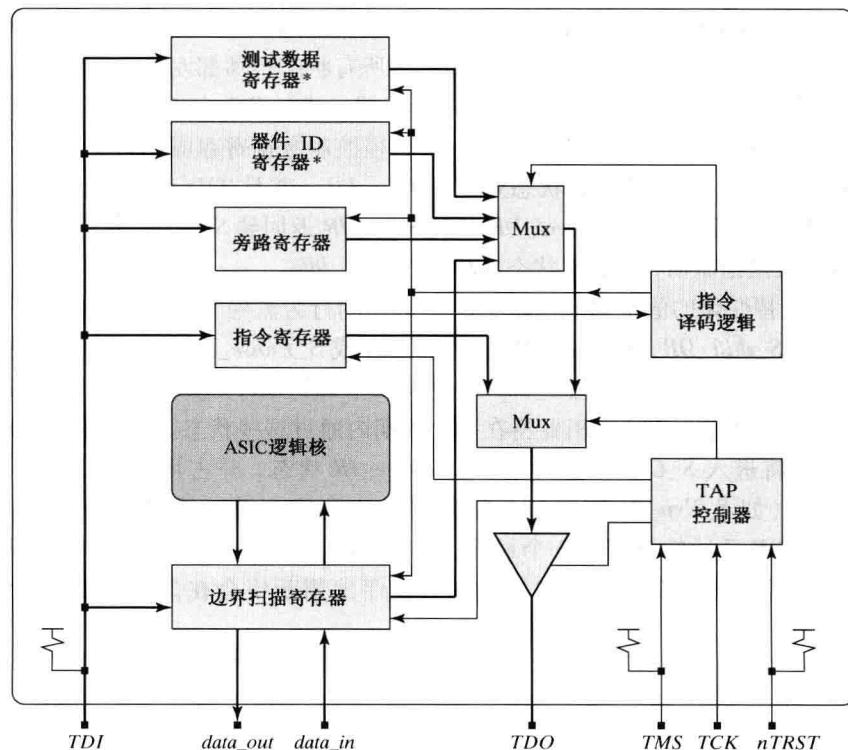


图 11.49 带有 JTAG 测试接入端口的芯片结构(* 表示可选寄存器。低电平有效输入 nTRST 也是可选的)

具有 JTAG 端口的 ASIC 或其他器件需要一条由 4 个专用引脚(*TDI*, *TDO*, *TMS* 和 *TCK*)组成的测试总线, 来支持边界扫描和内部测试^①。TAP 的 *TDI* 和 *TDO* 引脚分别连接到边界扫描寄存器链路中的第一个和最后一个单元, 用作芯片的接口。测试数据输入(*TDI*)引脚作为输入将测试模板以串行方式施加于端口, 测试数据输出(*TDO*)引脚用做串行输出端口。TAP 的工作模式可通过测试模式选择(*TMS*)的输入端来控制。测试用的主时钟信号应加到测试时钟(*TCK*)输入引脚。能够实现 JTAG 结构的 PC 板需要 4 个附加的引脚, 用来提供 *TDI*、*TDO*、*TMS* 和 *TCK* 信号, 或许还需要给 *nTRST* 提供另外一个引脚, 如图 11.49 所示。每一个 ASIC 作为总线从设备, 每一个外部代理作为总线主设备, 并利用 *TMS* 和 *TCK* 来控制从属设备。

PC 板上每一片 ASIC 的 TAP 包括了一个 TAP 控制器和一个状态机(其有 4 个引脚专门与 JTAG 连接)。*TMS* 的输入能够控制 TAP 控制器的状态转移, 而且每一次转移都发生在 *TCK*^② 上升沿。由 TAP 控制器产生的信号驱动寄存器单元的输入 *shiftDR*, *mode*, *clockDR*, *updateDR*, *shiftDR*, *clockIR* 和 *updateIR*。图 11.50(a)所示的一块 PC 板由两片配有扫描链路的 ASIC 环形相连构成^[6]。为简便起见, TAP 控制信号没有标出。在环形结构中, 每个芯片都由相同的 TAP 信号驱动。在更为常见的星形结构中(见图 11.50(b)), 芯片的串行端口是串行连接的, 而链路中的每个芯片都有它自己的 *TMS* 信号。总线主设备通过控制各自的 *TMS* 信号控制 TAP, 因此允许各自的 TAP 控制器单独受控。

11.8.6 TAP 控制器状态机

指令寄存器和 TAP 控制器状态机控制了 TAP 的数据通道。图 11.51 所示为该状态机的算法状态机(ASM), 图中采用十进制标注来表示状态码^③所有状态转移都发生在 *TCK* 的上升沿; ASIC 中测试逻辑的行为或者发生在控制器每个状态的上升沿, 或者发生在每个状态的下降沿。

TAP 控制器的 ASM 图基本上是对称的, 一条路径控制数据寄存器的行为, 另一条路径控制 TAP 指令寄存器的行为。如果机器的状态处在 *S_Run_Idle*, 并且 *TMS* 已被激活, 如果 *TMS* 保持激活两个时钟周期, 状态将经由 *S_Select_DR* 和 *S_Select_IR* 返回到 *S_Reset*, 然后它将停留在 *S_Reset* 状态直到 *TMS* 被撤销激活, 才能使状态转移到 *S_Run_Idle*。

请注意 *TMS* 交替变化的值是如何控制 TAP 控制器的行为流程的, 即那些能使状态转移到 *S_Reset*, *S_Run_Idle*, *S_shift_DR*, *S_Pause_DR*, *S_shift_IR* 或 *S_Pause_IR* 的 *TMS* 值将使机器保持在原有状态, 直到 *TMS* 的值改变。状态 *S_Capture_DR*, *S_Exit1_DR*, *S_Exit2_DR*, *S_Capture_IR*, *S_Exit1_IR*, *S_Exit2_IR* 为临时状态, 机器将在一个周期内通过这些状态。当对应的捕获/扫描寄存器被装载的时候, 将进入 *S_Capture_DR* 和 *S_Capture_IR* 状态, 并占用一个周期的时间。注意到所谓的退出状态(如 *S_Exit2*)使得一个单控制信号 *TMS* 有效地控制着状态机的行为流程。例如, 从 *S_Pause_DR* 开始的流程有 3 个最终目的地(如 *S_Pause_DR*, *S_Update_DR* 或 *S_shift_DR*)。单比特控制信号通过在两个时钟周期内有次序地判断将会在 3 种可能情况中做出转移目标的选择。

① 可选的第 5 个引脚可以用作异步低电平有效地测试复位输入信号(*nTRST*), 对 TAP 控制器进行复位。与 *TMS* 和 *TDI* 一样, *Ntrst* 必须要与上拉器件相连接。

② JTAG 标准要求的。

③ JTAG 标准并没有规定 TAP 的状态码。为了清楚起见, 图 11.51 所示的 TAP 控制器的状态名前增加了前缀 S_。为了简化 JTAG 标准中规定的状态 *Test-Logic-Reset*, *Run-Test-Idle*, *Select-DR-Scan* 和 *Select-IR-Scan*, 图 11.51 和 TAP 控制器的模型中把它们分别命名为 *S_Reset*, *S_Run_Idle*, *S_Select_DR* 和 *S_Select_IR*。

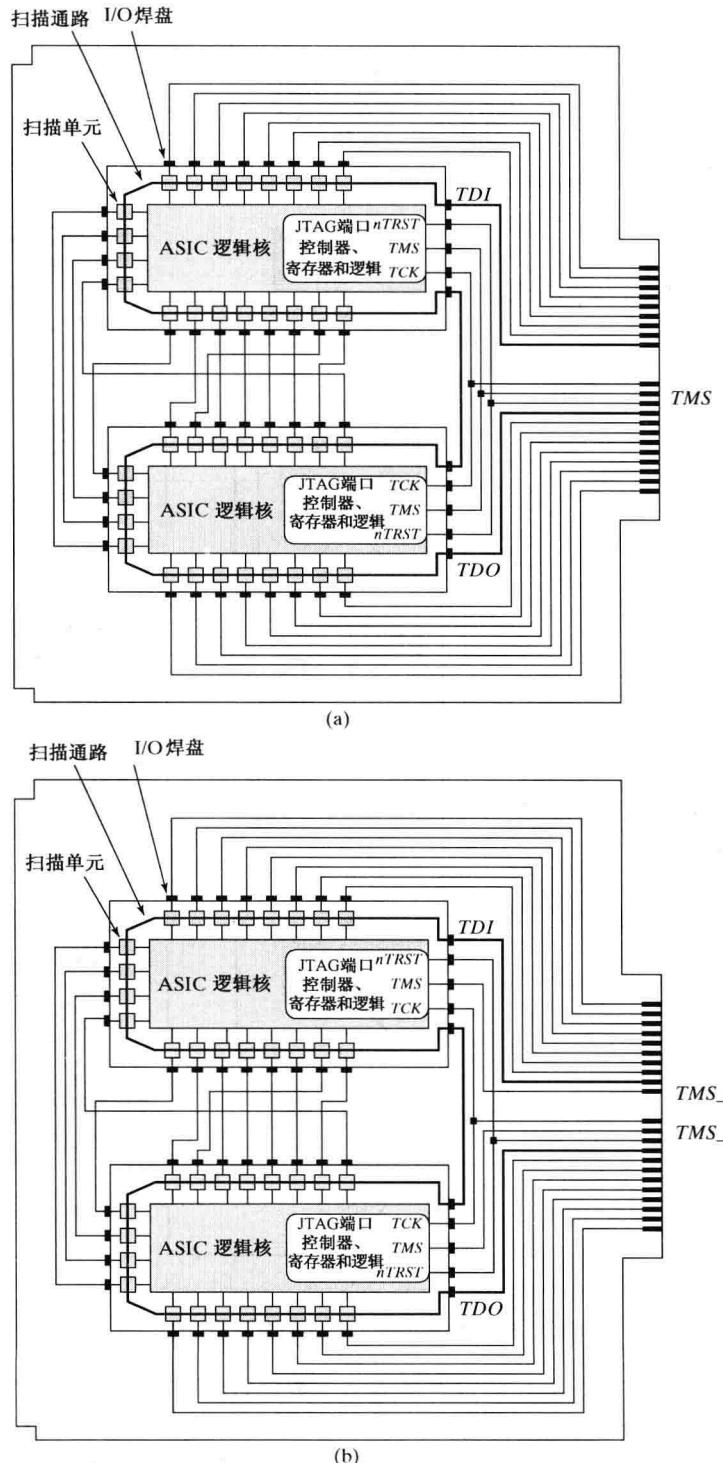


图 11.50 配有边界扫描单元电路和 JTAG 端口的 ASIC 构成的 PC 板：
 (a) 在测试模式下板上芯片连接成环形结构的菊花链以用作 TMS；
 (b) 在测试模式下板上芯片串行连接成星形结构以用作 TMS

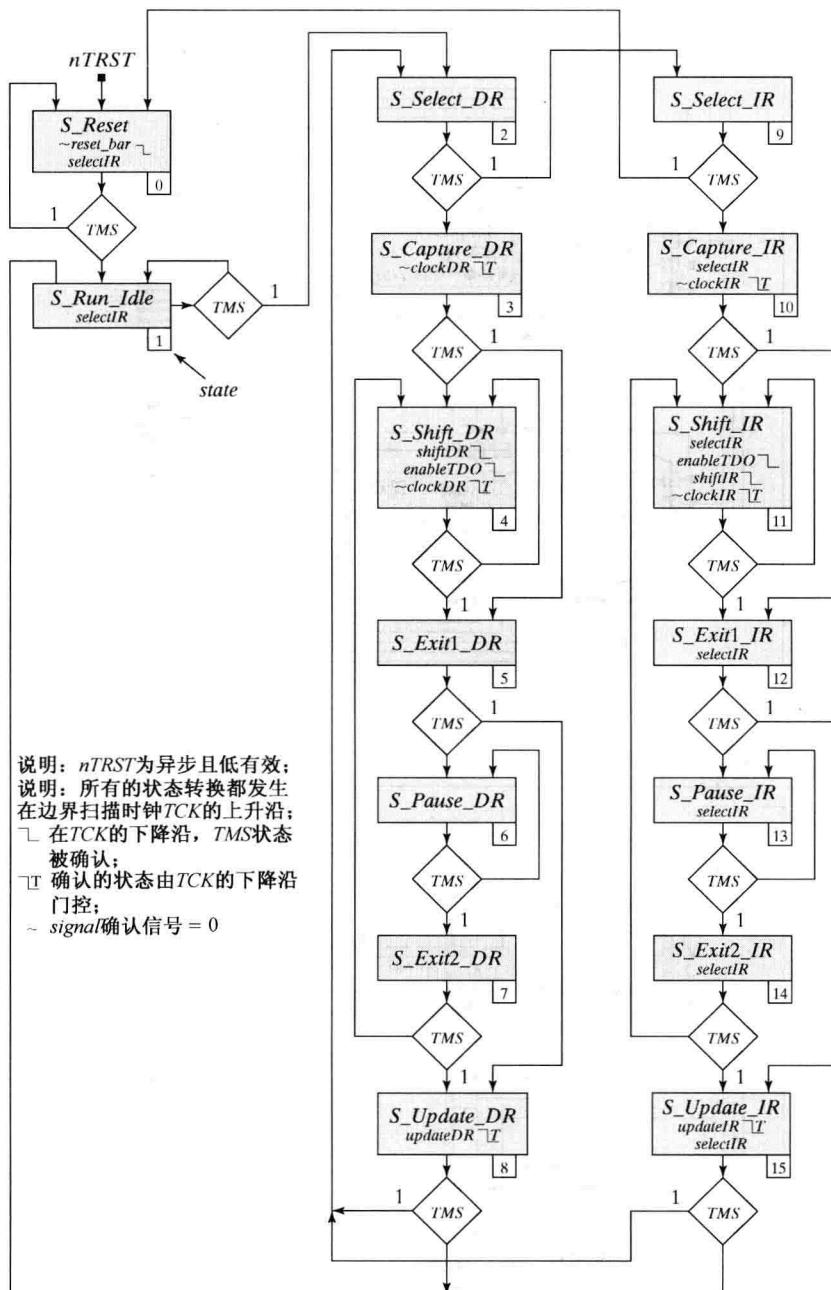


图 11.51 TAP 控制器状态机的 ASM 图

指令寄存器的内容决定了边界扫描寄存器或测试数据寄存器中的一个是否会因控制器的操作受到影响。也要注意, TAP 控制器的输入 $nTRST$ 是可选的, 因为通过确认 TMS , 至多需要 5 个时钟周期, 就可以从其他任何状态到达 S_{Reset} 状态。在综合后仿真中可能就需要用 $nTRST$ 来驱动 TAP 控制器的门级模型进入到一个已知的初始状态。

影响数据寄存器的控制状态描述见表 11.13, 影响指令寄存器的控制状态的描述与之类似。

表 11.9 TAP 控制器的状态机的控制状态

状态	行为事件
<i>S_Reset</i>	TAP 控制器的复位状态。TAP 的测试逻辑被禁止，主 ASIC 正常工作。如果机器中有一个器件识别寄存器，则将 IDCODE 指令装入指令寄存器；否则将装入 BYPASS 指令
<i>S_Run_Idle</i>	当主 ASIC 执行内部测试(如 BIST)期间，TAP 控制器保持在 <i>S_Run_Idle</i> 状态。指令寄存器必须提前装入支持测试的信息
<i>S_Select_DR</i>	在控制器处于 <i>S_Run_Idle</i> 状态时，TMS 的激活值将驱动状态进入到 <i>S_Select_DR</i> 状态，并驻留一个时钟周期，之后过渡到 <i>S_Capture_DR</i> 来启动一个扫描数据序列，或进入到 <i>S_Select_IR</i> 状态启动运行序列以更新指令寄存器或终止运行
<i>S_Capture_DR</i>	当状态驻留在 <i>S_Capture_DR</i> 时，由现有指令指定的边界扫描寄存器或测试数据寄存器中的捕获/扫描寄存器可以被并行加载(通过 <i>data_in</i>)，并在 <i>clockDR</i> 脉冲和 <i>shift_DR</i> 低电平的作用下开始捕获数据
<i>S_Shift_DR</i>	由指令寄存器所选定的测试数据寄存器在每个 <i>TCK</i> 的有效沿处向串行输出端口移动一个单元。一个数据位从 <i>TDI</i> 端口读入并从 <i>TDO</i> 端口读出。驱动 <i>TDO</i> 的缓冲器只有在移位时才有效
<i>S_Exit1_DR</i>	由 <i>S_Shift_DR</i> 状态(在移动序列之后)或 <i>S_Capture_DR</i> 状态(跳过初始移动序列)进入的暂时状态。一个周期后状态转移到 <i>S_Pause_DR</i> 并暂停，直到 <i>TMS</i> 被再次确认，或转移到 <i>S_Update_DR</i> 状态，在此状态下捕获的数据应被装入输出寄存器
<i>S_Pause_DR</i>	状态停留在 <i>S_Pause_DR</i> ，在 <i>TMS = 0</i> 时暂时停止扫描过程，直到 <i>TMS</i> 再次被激活，而捕获/扫描寄存器将保存其状态
<i>S_Exit2_DR</i>	暂时状态。在转移到 <i>S_Shift_DR</i> 开始扫描序列之前，或者转移到 <i>S_Update_DR</i> 终止扫描并将已捕获的数据装入输出寄存器之前，状态会停留在 <i>S_Exit2_DR</i> 一个周期
<i>S_Update_DR</i>	在状态转移到 <i>S_Select_DR</i> ，开始扫描序列或指令序列之前，或者在状态转移到 <i>S_Run_Idle</i> ，且在 ASIC 执行操作期间并保持该状态之前，而且在从捕获/扫描寄存器向输出寄存器装载数据的时钟周期之后，状态停留在 <i>S_Update_DR</i> 一个时钟周期。在测试模式下，输出寄存器的值驱动到并行输出

TAP 控制器状态机所产生的输出信号示于表 11.10 中，用它们可以控制扫描寄存器的工作状态。

表 11.10 由 TAP 控制状态机所产生的 Moore 型输出

输出	功能
<i>reset_bar</i>	将指令寄存器(IR)复位到 IDCODE 或 BYPASS
<i>shiftIR</i>	为指令寄存器中的捕获/扫描触发器选择串行输入
<i>clockIR</i>	在 IR 的输入端获取数据，或者将 IR 的内容移至测试数据输出端。该动作可由 <i>TCK</i> 的下降沿控制
<i>updateIR</i>	将 IR 中捕获触发器的内容装入输出寄存器。该动作可由 <i>TCK</i> 下降沿控制
<i>shiftDR</i>	为 <i>TDR</i> 单元中的捕获/扫描触发器选择串行输入
<i>clockDR</i>	在 IR 输入端获取数据或者将 <i>TDR</i> 的内容移至测试数据输出端。该动作可由 <i>TCK</i> 的下降沿控制
<i>updateDR</i>	将 <i>TDR</i> 捕获/扫描触发器的内容装入输出寄存器。该动作可由 <i>TCK</i> 的上升沿控制
<i>selectIR</i>	在 TAP 的 <i>TDI</i> 和 <i>TDO</i> 引脚间，选择连接指令寄存器还是测试数据寄存器
<i>enableTDO</i>	使能驱动测试数据输出(<i>TDO</i>)的三态缓冲器

11.8.7 设计实例：JTAG 测试

这里的例子说明了 JTAG 如何对一个带有边界扫描链路和 TAP 控制器的 ASIC 进行扩充，接着说明 BYPASS 和 INTEST 指令。本章末的练习将涉及控制器的附加特性。这里的 ASIC 为一个简单的 4 位加法器，但这里的方法对于更加复杂的 ASIC 也适用。此处主要考虑其结构上和操作上的某些细节。

使用 JTAG 测试 ASIC 需要几个步骤。例如，要测试一个包含组合逻辑的 ASIC 核，状态机的状态必须指向 *S_Shift_DR*，并保持该状态多个时钟周期，直至测试模板被移入边界扫描寄存器。在移入操作结束时，测试输入将保存在驱动芯片输入的捕获/扫描寄存器单元中。触发信号 *update_DR* 将把捕获/扫描寄存器的内容传输到输出寄存器。在测试模式下，输出寄存器的测试模板将驱动 ASIC 的输入引脚。ASIC 的响应将呈现在与 ASIC 输出引脚相连的捕获/扫描寄存器单元的 *data_in* 引脚上。随着 *shiftDR* 的撤销，触发信号 *clockDR* 便会从输入引脚获得数据，并且把电路对测试模板的响应装载到捕获/扫描寄存器中。然后，在 *shiftDR* 有效时，连续触发 *clockDR* 便会从扫描链路中扫出数据。在前一个模板被扫描出去的同时，将另一个模板扫入 IR。

图 11.52 中给出了经过修改的包含 TAP 的 ASIC 的完整结构。为了简便起见，ASIC 为一个内嵌的 4 位加法器。TAP 控制器和 TAP 的控制信号在本例中被忽略，但被包含在增强 JTAG 的 ASIC 模型中。

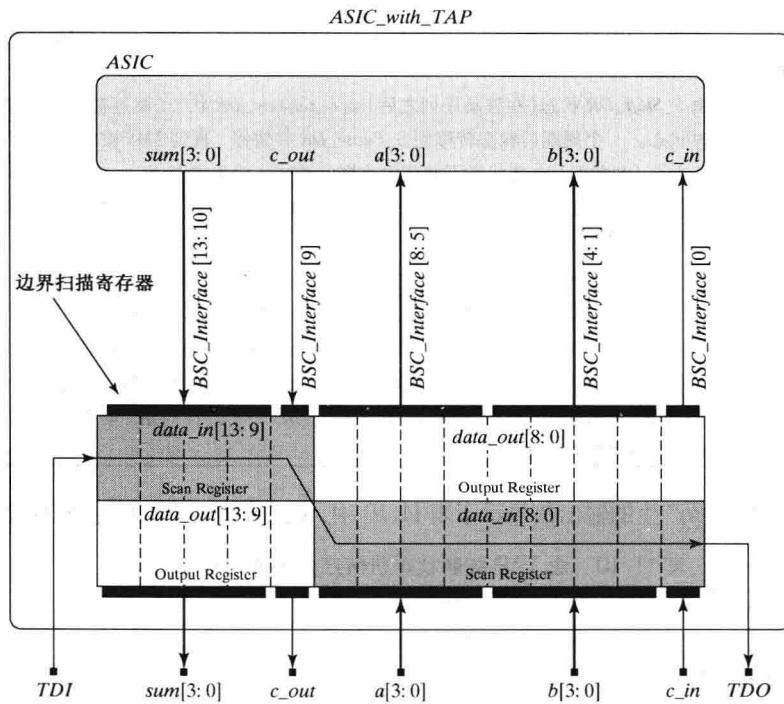


图 11.52 边界扫描寄存器和端口接口结构

在实现时最关键的一步是在 ASIC 和它的工作平台之间创建接口。请注意，ASIC 保留了它的端口结构，但直接通过总线 *BSC_Interface[13:0]* 连接到扫描寄存器，从而完成了 ASIC、边界扫描寄存器、工作平台之间的端口接口匹配。ASIC 的端口模式决定了 *BSC_Interface* 的连线是连接到边界扫描器的输入端口还是输出端口。ASIC 的输出连接到边界扫描寄存器的输入引脚，边界扫描寄存器又通过 *data_in[13:9]* 连接到捕获/扫描寄存器。对应的输出寄存器单元被连接到 *ASIC-with_TAP* 的输出端(原始输出端)。同样，在 *data_out[8:0]* 处的输出寄存器将驱动 ASIC 的输入，而相应的捕获/扫描寄存器将通过 *data_in[8:0]* 由芯片的外部(原始)输入驱动。

边界扫描寄存器单元的捕获/扫描寄存器和输出寄存器被单独表示出来，并由一条通过它们的数据通路来表示扫描链路。扫描寄存器(阴影部分)被连接到 ASIC 的输出端和芯片的原始输入引脚，输出寄存器被连接到基本输出引脚和 ASIC 输入端。例如，ASIC 的 *Sum[3:0]* 和 *c_out* 端口

被连接到 `data_in[13:9]` 端口，而 `data_out[13:9]` 被连接到 `ASIC_with_TAP` 和主机环境的接口 `{sum[3:0], c_out}`。

图 11.52 所示的结构是灵活的，而且接口信号的顺序也是任意的。有几点值得注意：

- (1) 可以对端口结构进行修改以适合不同 ASIC 端口的要求；
- (2) 边界扫描寄存器的大小也可以重新调整；
- (3) `BSC_Interface` 可以与 ASIC 的输入/输出端口匹配。

作为 `ASIC_with_TAP` 总体设计和验证过程中的第一步，给出下面的指令寄存器和 8 位边界扫描寄存器的模型以及仿真结果，展示了边界扫描寄存器的并行和串行 I/O 模式。在这个例子中，`shiftDR`, `clockDR`, `updateDR` 和 `mode` 都可以由测试平台控制。图 11.53 中的仿真结果被标成高亮以显示操作的正常工作模式、测试模式和寄存器动作的形式。`clockDR` 的第一个脉冲期间，将在 `data_in(8'haa)` 处获取并行数据并将其装入 `BSC_Scan_Register`。当 `shiftDR` 撤销后，`updateDR` 的第一个脉冲表示了扫描寄存器(`8'haa`)的数据被装入输出寄存器中，而没有影响到正常操作(`data_in` 和 `data_out` 没有影响)。当 `shiftDR` 激活时，`clockDR` 脉冲将 1 读进扫描寄存器，而数据通过 `scan_out` 退出。`updateDR` 的第二个脉冲将捕获/扫描寄存器(`8'hff`)的值装入输出寄存器。当进入测试模式后，输出寄存器的值驱动总线 `data_out`。当测试模式撤销后，`data_out` 返回 `data_in` 的值到 `8'haa`。`clockDR` 的最后一个脉冲读取 `data_in` 的数据，并将值 `8'haa` 再次装入扫描寄存器中。

```

module Boundary_Scan_Register #(parameter size = 14)(  
    output [size -1: 0] data_out,  
    output scan_out,  
    input [size -1: 0] data_in,  
    input scan_in,  
    input shiftDR, mode, clockDR, updateDR  
);  
  
reg [size -1: 0] BSC_Scan_Register, BSC_Output_Register;  
  
always @ (posedge clockDR)  
    BSC_Scan_Register <= shiftDR ? {scan_in, BSC_Scan_Register [ size -1: 1]} :  
        data_in;  
  
always @ (posedge updateDR) BSC_Output_Register <= BSC_Scan_Register;  
  
assign scan_out = BSC_Scan_Register [0];  
assign data_out = mode ? BSC_Output_Register : data_in;  
endmodule  
  
module Instruction_Register #(parameter IR_size = 3)(  
    output [IR_size -1: 0] data_out,  
    output scan_out,  
    input [IR_size -1: 0] data_in,  
    input scan_in,shiftIR, clockIR, updateIR, reset_bar  
);  
reg [IR_size -1: 0] IR_Scan_Register, IR_Output_Register;  
  
assign data_out = IR_Output_Register;  
assign scan_out = IR_Scan_Register [0];  
  
always @ (posedge clockIR)  
    IR_Scan_Register <= shiftIR ? {scan_in, IR_Scan_Register [IR_size - 1: 1]} : data_in;  
always @ (posedge updateIR, negedge reset_bar) // Asynchronous per  
                                                // 1140.1a.  
    if (reset_bar == 0) IR_Output_Register <= ~(0); // Fills IR with 1s  
                                                // for BYPASS instruction  
    else IR_Output_Register <= IR_Scan_Register;  
endmodule

```

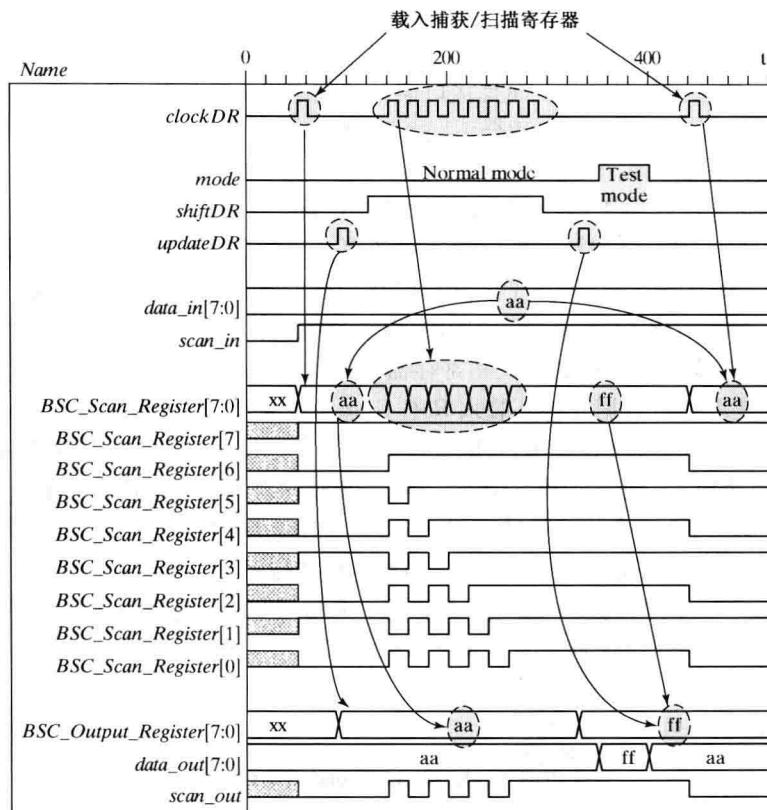


图 11.53 验证 8 位边界扫描寄存器操作正常的仿真结果

以下是 *TAP_Controller* 状态机的模块。状态机的状态为二进制编码。同样为了简便起见，门控时钟信号由 *clckDR*, *updateDR*, *clckIR* 和 *updateIR* 产生。为了能够真正实现，信号连接到具有一个复用的专用触发器的时钟输入引脚(参见 6.11 节)。

```

module TAP_Controller (
  output reg reset_bar, selectIR, shiftIR,
  output reg shiftDR, enableTDO,
  output   clockDR, updateDR, clockIR, updateIR,
  input    TMS, TCK
);
  parameter S_Reset          = 0,
           S_Run_Idle      = 1,
           S_Select_DR     = 2,
           S_Capture_DR    = 3,
           S_Shift_DR      = 4,
           S_Exit1_DR      = 5,
           S_Pause_DR      = 6,
           S_Exit2_DR      = 7,
           S_Update_DR     = 8,
           S_Select_IR     = 9,
           S_Capture_IR    = 10,
           S_Shift_IR      = 11,
           S_Exit1_IR      = 12,
           S_Pause_IR      = 13,
           S_Exit2_IR      = 14,
           S_Update_IR     = 15;

```

```

reg [3:0] state, next_state;
pullup (TMS); // Required by IEEE 1149.1a; ensures that an undriven input
pullup (TDI); // produces a response identical to the application of a logic 1."
// Program for Xilinx implementation

always @ (negedge TCK) reset_bar <= (state == S_Reset) ? 0 : 1; // Registered
// active low

always @ (negedge TCK) begin
    shiftDR <= (state == S_Shift_DR) ? 1 : 0; // Registered select for scan mode
    shiftIR <= (state == S_Shift_IR) ? 1 : 0; // Registered output enable
    enableTDO <= ((state == S_Shift_DR) || (state == S_Shift_IR)) ? 1 : 0;
end

// Gated clocks for capture registers
assign clockDR = !(((state == S_Capture_DR) || (state == S_Shift_DR)) &&
(TCK == 0));
assign clockIR = !(((state == S_Capture_IR) || (state == S_Shift_IR)) &&
(TCK == 0));

// Gated clocks for output registers
assign updateDR = (state == S_Update_DR) && (TCK == 0);
assign updateIR = (state == S_Update_IR) && (TCK == 0);

always @ (posedge TCK) state <= next_state;

always @ (state, TMS) begin
    selectIR = 0;
    next_state = state;

    case (state)
        S_Reset: begin
            selectIR = 1;
            if (TMS == 0) next_state = S_Run_Idle;
        end
        S_Run_Idle: begin
            selectIR = 1; if (TMS) next_state = S_Select_DR; end
            next_state = TMS ? S_Select_IR: S_Capture_DR;
        end
        S_Capture_DR: begin
            next_state = TMS ? S_Exit1_DR: S_Shift_DR; end
            if (TMS) next_state = S_Exit1_DR;
            next_state = TMS ? S_Update_DR: S_Pause_DR;
            if (TMS) next_state = S_Exit2_DR;
            next_state = TMS ? S_Update_DR: S_Shift_DR;
        begin
            next_state = TMS ? S_Select_DR: S_Run_Idle; end
        begin
            next_state = TMS ? S_Reset: S_Capture_IR;
        end
        S_Capture_IR: begin
            selectIR = 1;
            next_state = TMS ? S_Exit1_IR: S_Shift_IR;
        end
        S_Shift_IR: begin
            selectIR = 1; if (TMS) next_state = S_Exit1_IR; end
            begin
                selectIR = 1;
                next_state = TMS ? S_Update_IR: S_Pause_IR;
            end
            begin
                selectIR = 1; if (TMS) next_state = S_Exit2_IR; end
            begin
                selectIR = 1;
                next_state = TMS ? S_Update_IR: S_Shift_IR;
            end
        begin
            selectIR = 1;
            next_state = TMS ? S_Select_DR: S_Run_Idle;
        end
    endcase
end

```

```

default          next_state = S_Reset;
endcase
end
endmodule

```

下面列出的参数化模块 *ASIC_with_TAP* 例化了如下模块：*ASIC*, *TAP_Controller*, *Boundary_Scan_Register*, *Instruction_Register* 和 *Instruction_Decoder*。通常情况下, TAP 的指令寄存器在 *S_Capture_IR* 状态下从并行数据通道(*data_in*)中读取数据, 在此状态下由主控部件^①产生的具体设计信息提供给 TAP。在这个例子中, 通过 *data_in* 端口传输 *Dummy_data* = 3'b001。

```

module ASIC_with_TAP #(parameter size = 4)
  output      [size -1: 0] sum,      // ASIC interface I/O
  output      c_out,
  input       [size -1: 0] a, b,
  input       c_in,
  output      TDO,        // TAP interface signals
  input      TDI, TMS, TCK
);

parameter    BSR_size = 14;
parameter    IR_size = 3;
wire         [BSR_size -1: 0] BSC_Interface; // Declarations for boundary scan register I/O
wire         reset_bar,      // TAP controller outputs
               selectIR, enableTDO,
               shiftIR, clockIR, updateIR,
               shiftDR, clockDR, updateDR;

wire         test_mode, select_BR;
wire         TDR_out;      // Test data register serial datapath
wire         [IR_size -1: 0] Dummy_data = 3'b001; // Captured in S_Capture_IR
wire         [IR_size -1: 0] instruction;
wire         IR_scan_out; // Instruction register
wire         BSR_scan_out; // Boundary scan register
wire         BR_scan_out; // Bypass register

assign      TDO = enableTDO ? selectIR ? IR_scan_out : TDR_out : 1'bz;
assign      TDR_out = select_BR ? BR_scan_out : BSR_scan_out;

ASIC M0 (
  .sum (BSC_Interface [13: 10]),
  .c_out (BSC_Interface [9]),
  .a (BSC_Interface [8: 5]),
  .b (BSC_Interface [4: 1]),
  .c_in (BSC_Interface [0]));
Bypass_Register M1(
  .scan_out (BR_scan_out),
  .scan_in (TDI),
  .shiftDR (shift_BR),
  .clockDR (clock_BR));

```

^① JTAG 标准要求在 *S_Capture_IR* 状态时, 指令寄存器中的两个最低单元应该装入模板 2'b01。剩下的位固定为 0 或 1, 这取决于有关的应用值。

```

Boundary_Scan_Register M2(
    .data_out ({sum, c_out, BSC_Interface [8: 5], BSC_Interface [4: 1],
               BSC_Interface [0]}),
    .data_in ({BSC_Interface [13: 10], BSC_Interface [9], a, b, c_in}),
    .scan_out (BSR_scan_out),
    .scan_in (TDI),
    .shiftDR (shiftDR),
    .mode (test_mode),
    .clockDR (clock_BSC_Reg),
    .updateDR (update_BSC_Reg));

Instruction_Register M3 (
    .data_out (instruction),
    .data_in (Dummy_data),
    .scan_out (IR_scan_out),
    .scan_in (TDI),
    .shiftIR (shiftIR),
    .clockIR (clockIR),
    .updateIR (updateIR),
    .reset_bar (reset_bar));

Instruction_Decoder M4 (
    .mode (test_mode),
    .select_BR (select_BR),
    .shift_BR (shift_BR),
    .clock_BR (clock_BR),
    .shift_BSC_Reg (shift_BSC_Reg),
    .clock_BSC_Reg (clock_BSC_Reg),
    .update_BSC_Reg (update_BSC_Reg),
    .instruction (instruction),
    .shiftDR (shiftDR),
    .clockDR (clockDR),
    .updateDR (updateDR));

TAP_Controller M5 (
    .reset_bar(reset_bar),
    .selectIR (selectIR),
    .shiftIR (shiftIR),
    .clockIR (clockIR),
    .updateIR (updateIR),
    .shiftDR (shiftDR),
    .clockDR (clockDR),
    .updateDR (updateDR),
    .enableTDO (enableTDO),
    .TMS (TMS),
    .TCK (TCK));

endmodule

module ASIC #(parameter size = 4) (
    output      [size -1: 0]      sum,
    output      [size -1: 0]      c_out,
    input       [size -1: 0]      a, b,
    input       [size -1: 0]      c_in
);
    assign {c_out, sum} = a + b + c_in;
endmodule

module Bypass_Register (
    output reg scan_out,
    input      scan_in, shiftDR, clockDR
);

```

```

always @ (posedge clockDR) scan_out <= scan_in & shiftDR;
endmodule

module Instruction_Decoder #(parameter IR_size = 3) (
    output reg mode, select_BR, clock_BR, clock_BSC_Reg,
        update_BSC_Reg,
    output shift_BR, shift_BSC_Reg,
    input [IR_size -1: 0] instruction,
    input shiftDR, clockDR, updateDR
);
    parameter BYPASS = 3'b111; // Required by 1149.1a
    parameter EXTEST = 3'b000; // Required by 1149.1a
    parameter SAMPLE_PRELOAD = 3'b010;
    parameter INTEST = 3'b011;
    parameter RUNBIST = 3'b100;
    parameter IDCODE = 3'b101;

    assign shift_BR = shiftDR;
    assign shift_BSC_Reg = shiftDR;

    always @ (instruction, clockDR, updateDR) begin
        mode = 0; select_BR = 0; // default is test-data register
        clock_BR = 1; clock_BSC_Reg = 1;
        update_BSC_Reg = 0;

        case (instruction)
            EXTEST: begin mode = 1; clock_BSC_Reg = clockDR;
                update_BSC_Reg = updateDR; end
            INTEST: begin mode = 1; clock_BSC_Reg = clockDR;
                update_BSC_Reg = updateDR; end
            SAMPLE_PRELOAD: begin clock_BSC_Reg = clockDR;
                update_BSC_Reg = updateDR; end
            RUNBIST: begin end
            IDCODE: begin select_BR = 1; clock_BR = clockDR; end
            BYPASS: begin select_BR = 1; clock_BR = clockDR; end
            default: begin select_BR = 1; end

        endcase
    end
endmodule

```

图 11.54 中给出了用于 *ASIC_with_TAP* 的测试平台 (*t ASIC_with_TAP*) 结构。两个 *Array_of_TAP_Instructions* 和 *Array_of_ASIC_Test_Patterns* 数组把扫描指令模板和测试模板存入边界扫描寄存器中。测试序列将从这些寄存器中选择一个模板并装入 *Pattern_Register*。当测试序列确认一个读取信号成立时，存储在 *Pattern_Register* 中的值被装入 *TDI_Generator* 中的寄存器 *TDI_Reg* 中。同时 TAP 控制器将把该模板从 *TDI_Reg* 扫描到 TAP 的 *TDI* 端口和 *Pattern_Buffer_1*。TAP 的 *TDO* 端口中的值被扫描进 *TDO_Monitor* 中的 *TDO_Reg*，同时，*Pattern_Buffer_1* 中的值被扫描进 *Pattern_Buffer_2*。当扫描完成后，将对 *TDO_Reg* 和 *Pattern_Buffer_2* 的内容加以比较来检测错误。

以下是 *t ASIC_with_TAP* 的测试平台，以及一些对需要进行验证的功能特性的注解。

```

module t ASIC_with_TAP (); // Testbench
    parameter size = 4;
    parameter BSC_Reg_size = 14;
    parameter IR_Reg_size = 3;
    parameter N_ASIC_Patterns = 8;
    parameter N_TAP_Instructions = 8;
    parameter Pause_Time = 40;
    parameter End_of_Test = 1500;
    parameter time_1 = 350, time_2 = 550;

    wire [size -1: 0] sum;
    wire [size -1: 0] sum_fr_ASIC = M0.BSC_Interface [13: 10];

```

```

wire          c_out;
wire          c_out_fr_ASIC = M0.BSC_Interface [9];
reg [size -1: 0] a, b;
reg          c_in;
wire          a_to_ASIC = M0.BSC_Interface [8: 5];
wire          b_to_ASIC = M0.BSC_Interface [4: 1];
wire          c_in_to_ASIC = M0.BSC_Interface [0];

reg          TMS, TCK;
wire          TDI;
wire          TDO;
reg          load_TDI_Generator;
reg          Error, strobe;
integer        pattern_ptr;
reg [BSC_Reg_size -1: 0] Array_of_ASIC_Test_Patterns
                     [0: N_ASIC_Patterns -1];
reg [IR_Reg_size -1: 0] Array_of_TAP_Instructions
                     [0: N_TAP_Instructions -1];
reg [BSC_Reg_size -1: 0] Pattern_Register; // Size to maximum
                                         TDR
reg          enable_bypass_pattern;

ASIC_with_TAP M0 (sum, c_out, a, b, c_in, TDO, TDI, TMS, TCK);

TDI_Generator M1(
.to_TDI (TDI),
.scan_pattern (Pattern_Register),
.load (load_TDI_Generator),
.enable_bypass_pattern (enable_bypass_pattern),
.TCK (TCK));
TDO_Monitor M3 (
.to_TDI (TDI),
.from_TDO (TDO),
.strobe (strobe),
.TCK (TCK));

initial #End_of_Test $finish;
initial begin TCK = 0; forever #5 TCK = ~TCK; end
/* Summary of a basic test plan for ASIC_with TAP
Verify default to bypass instruction
Verify bypass register action: Scan 10 cycles, with pause before exiting
Verify pull up action on TMS and TDI
Reset to S_Reset after five assertions of TMS
Boundary scan in, pause, update, return to S_Run_Idle
Boundary scan in, pause, resume scan in, pause, update, return to S_Run_Idle
Instruction scan in, pause, update, return to S_Run_Idle
Instruction scan in, pause, resume scan in, pause, update, return to S_Run_Idle
*/
// TEST PATTERNS
// External I/O for normal operation
initial fork
  // {a, b, c_in} = 9'b0;
  {a, b, c_in} = 9'b_1010_0101_0; // sum = F, c_out = 0, a = A, b = 5, c_in = 0
join
/* Option to force error to test fault detection
initial begin :Force_Error
  force M0.BSC_Interface [13: 10] = 4'b0;
end
*/
initial begin // Test sequence: Scan, pause, return to S_Run_Idle
  strobe = 0;

```

```

Declare_Array_of_TAP_Instructions;
Declare_Array_of ASIC_Test_Patterns;
Wait_to_enter_S_Reset;

// Test for power-up and default to BYPASS instruction (all 1s in IR), with default path
// through the Bypass Register, with BSC register remaining in wakeup state (all x).
// ASIC test pattern is scanned serially, entering at TDI, passing through the
// bypass register,
// and exiting at TDO. The BSC register and the IR are not changed.

pattern_ptr = 0;
Load ASIC_Test_Pattern;
Go_to_S_Run_Idle;
Go_to_S_Select_DR;
Go_to_S_Capture_DR;
Go_to_S_Shift_DR;
enable_bypass_pattern = 1;
Scan_Ten_Cycles;
enable_bypass_pattern = 0;
Go_to_S_Exit1_DR;
Go_to_S_Pause_DR;
Pause;
Go_to_S_Exit2_DR;
/*
Go_to_S_Shift_DR;
Load ASIC_Test_Pattern;      // option to re-load same pattern and scan again
enable_bypass_pattern = 1;
Scan_Ten_Cycles;
enable_bypass_pattern = 0;
Go_to_S_Exit1_DR;
Go_to_S_Pause_DR;
Pause;
Go_to_S_Exit2_DR;
*/
Go_to_S_Update_DR;
Go_to_S_Run_Idle;
end

// Test to load instruction register with INTEST instruction

initial #time_1 begin
pattern_ptr = 3;
strobe = 0;
Load_TAP_Instruction;
Go_to_S_Run_Idle;
Go_to_S_Select_DR;
Go_to_S_Select_IR;
Go_to_S_Capture_IR;           // Capture dummy data (3'b011)
repeat (IR_Reg_size) Go_to_S_Shift_IR;
Go_to_S_Exit1_IR;
Go_to_S_Pause_IR;
Pause;
Go_to_S_Exit2_IR;
Go_to_S_Update_IR;
Go_to_S_Run_Idle;
end

// Load ASIC test pattern
initial #time_2 begin
pattern_ptr = 0;
Load ASIC_Test_Pattern;
Go_to_S_Run_Idle;
Go_to_S_Select_DR;
Go_to_S_Capture_DR;
repeat (BSC_Reg_size) Go_to_S_Shift_DR;

```

```

Go_to_S_Exit1_DR;
Go_to_S_Pause_DR;
Pause;
Go_to_S_Exit2_DR;
Go_to_S_Update_DR;
Go_to_S_Run_Idle;

// Capture data and scan out while scanning in another pattern
pattern_ptr = 2;
Load ASIC Test Pattern;
Go_to_S_Select_DR;
Go_to_S_Capture_DR;
strobe = 1;
repeat (BSC_Reg_size) Go_to_S_Shift_DR;
Go_to_S_Exit1_DR;
Go_to_S_Pause_DR;
Go_to_S_Exit2_DR;
Go_to_S_Update_DR;
strobe = 0;
Go_to_S_Run_Idle;
end

***** TAP CONTROLLER TASKS *****
task Wait_to_enter_S_Reset;
begin
@ (negedge TCK) TMS = 1;
end
endtask

task Reset_TAP;
begin
TMS = 1;
repeat (5) @ (negedge TCK);
end
endtask

task Pause;
begin #Pause_Time;
end endtask

task Go_to_S_Reset;
begin @ (negedge TCK) TMS = 1; end endtask
task Go_to_S_Run_Idle;
begin @ (negedge TCK) TMS = 0; end endtask

task Go_to_S_Select_DR;
begin @ (negedge TCK) TMS = 1; end endtask
task Go_to_S_Capture_DR;
begin @ (negedge TCK) TMS = 0; end endtask
task Go_to_S_Shift_DR;
begin @ (negedge TCK) TMS = 0; end endtask
task Go_to_S_Exit1_DR;
begin @ (negedge TCK) TMS = 1; end endtask
task Go_to_S_Pause_DR;
begin @ (negedge TCK) TMS = 0; end endtask
task Go_to_S_Exit2_DR;
begin @ (negedge TCK) TMS = 1; end endtask
task Go_to_S_Update_DR;
begin @ (negedge TCK) TMS = 1; end endtask

task Go_to_S_Select_IR;
begin @ (negedge TCK) TMS = 1; end endtask
task Go_to_S_Capture_IR;
begin @ (negedge TCK) TMS = 0; end endtask

task Go_to_S_Shift_IR;
begin @ (negedge TCK) TMS = 0; end endtask
task Go_to_S_Exit1_IR;
begin @ (negedge TCK) TMS = 1; end endtask
task Go_to_S_Pause_IR;
begin @ (negedge TCK) TMS = 0; end endtask
task Go_to_S_Exit2_IR;
begin @ (negedge TCK) TMS = 1; end endtask
task Go_to_S_Update_IR;
begin @ (negedge TCK) TMS = 1; end endtask
task Scan_Ten_Cycles;
begin repeat (10) begin @ (negedge TCK)
TMS = 0;
@ (posedge TCK) TMS = 1; end end endtask

***** ASIC TEST PATTERNS *****
task Load ASIC Test Pattern;
begin
Pattern_Register = Array_of ASIC Test Patterns [pattern_ptr];
@ (negedge TCK ) load_TDI_Generator = 1;
@ (negedge TCK ) load_TDI_Generator = 0;

```

```

end
endtask

task Declare_Array_of ASIC_Test_Patterns;
begin
//s3 s2 s1 s0_c0_a3 a2 a1 a0_b3 b2 b1 b0_c_in;

Array_of ASIC_Test_Patterns [0] = 14'b0100_1_1010_1010_0;
Array_of ASIC_Test_Patterns [1] = 14'b0000_0_0000_0000_0;
Array_of ASIC_Test_Patterns [2] = 14'b1111_1_1111_1111_1;
Array_of ASIC_Test_Patterns [3] = 14'b0100_1_0101_0101_0;
end endtask

/******************************************* INSTRUCTION PATTERNS *****/
parameter BYPASS= 3'b111;// pattern_ptr = 0
parameter EXTEST= 3'b001;// pattern_ptr = 1
parameter SAMPLE_PRELOAD= 3'b010;// pattern_ptr = 2
parameter INTTEST= 3'b011;// pattern_ptr = 3
parameter RUNBIST= 4'b100;// pattern_ptr = 4
parameter IDCODE= 5'b101;// pattern_ptr = 5

task Load_TAP_Instruction;
begin
Pattern_Register = Array_of_TAP_Instructions [pattern_ptr];
@(posedge TCK) load_TDI_Generator = 1;
@(posedge TCK) load_TDI_Generator = 0;
end
endtask

task Declare_Array_of_TAP_Instructions;
begin
Array_of_TAP_Instructions [0] = BYPASS;
Array_of_TAP_Instructions [1] = EXTEST;
Array_of_TAP_Instructions [2] = SAMPLE_PRELOAD;
Array_of_TAP_Instructions [3] = INTTEST;
Array_of_TAP_Instructions [4] = RUNBIST;
Array_of_TAP_Instructions [5] = IDCODE;
end
endtask
endmodule

module TDI_Generator #(parameter BSC_Reg_size = 14)(
output to_TDI,
input [BSC_Reg_size -1: 0] scan_pattern,
input load, enable_bypass_pattern, TCK
);
reg [BSC_Reg_size -1: 0] TDI_Reg;
wire enableTDO = t_ASIC_with_TAP.M0.enable
TDO;
assign to_TDI = TDI_Reg [0];

always @ (posedge TCK) if (load) TDI_Reg <= scan_pattern;
else if (enableTDO || enable_bypass_pattern)
TDI_Reg <= TDI_Reg >> 1;
endmodule

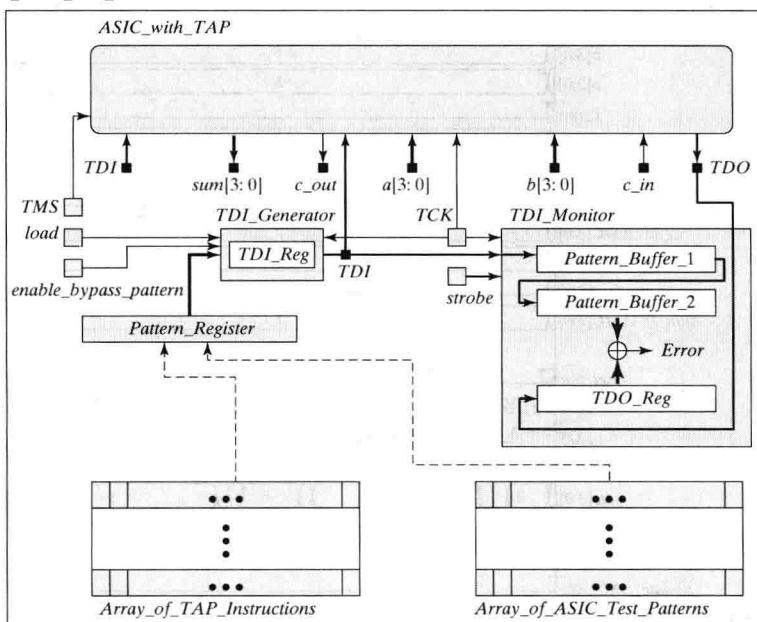
module TDO_Monitor #(parameter BSC_Reg_size = 14)(
input to_TDI,
input from_TDO, strobe, TCK
);
reg [BSC_Reg_size -1: 0] TDI_Reg, Pattern_Buffer_1,
Pattern_Buffer_2,
Captured_Pattern, TDO_Reg;
reg Error;
parameter test_width = 5;
wire enableTDO = t_ASIC_with_TAP.M0.enable
TDO;

```

```

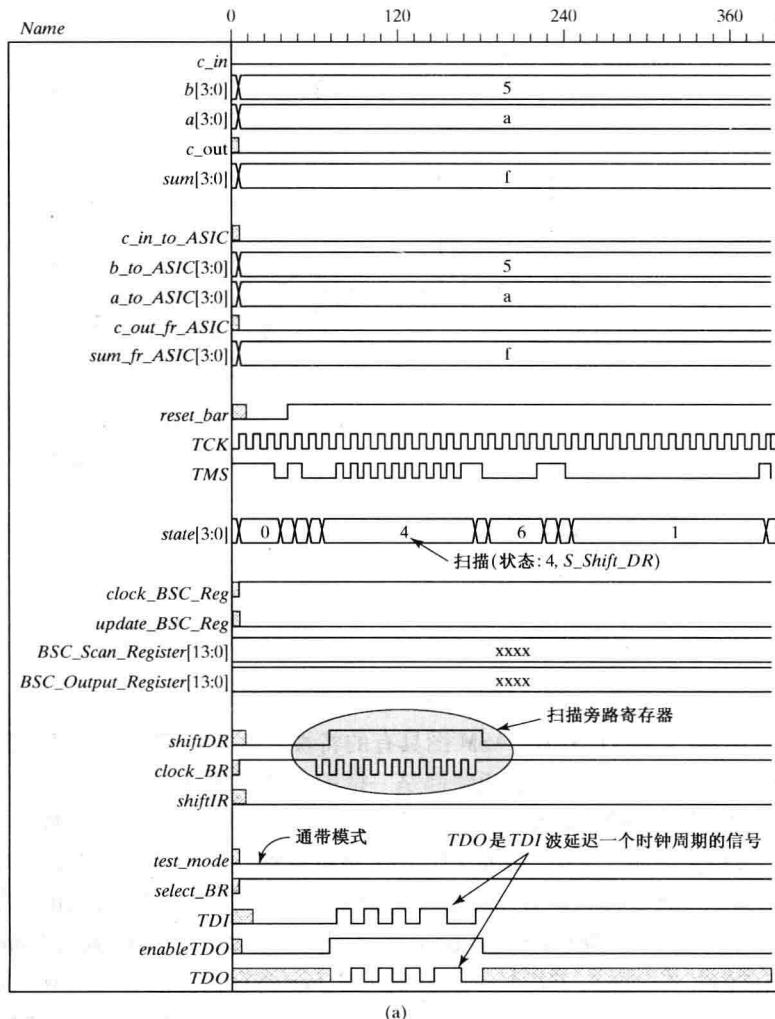
wire [test_width -1: 0] Expected_out =
Pattern_Buffer_2 [BSC_Reg_size -1:
:BSC_Reg_size - test_width];
wire [test_width -1: 0] ASIC_out =
TDO_Reg [BSC_Reg_size - 1:
BSC_Reg_size - test_width];
initial Error = 0;
always @ (negedge enableTDO) if (strobe == 1) Error = |(Expected_out ^
ASIC_out);
always @ (posedge TCK) if (enableTDO) begin
Pattern_Buffer_1 <= {to_TDI, Pattern_Buffer_1 [BSC_Reg_size -1: 1]};
Pattern_Buffer_2 <= {Pattern_Buffer_1 [0], Pattern_Buffer_2 [BSC_Reg_size -1: 1]};
TDO_Reg <= {from_TDO, TDO_Reg [BSC_Reg_size -1: 1]};
end
endmodule

```

t_ASIC_with_TAP图 11.54 *ASIC_with_TAP* 的测试平台的结构

请注意 TAP 控制器(见图 11.51)的 ASM 图具有的特性, 即 *TMS* 值(该值使现态进入状态转移图中的一个状态)对于进入该状态的所有通路都是一样的。通过观察写出了一组测试平台任务, 来指定一串输入序列, 使其能够控制流程沿着 ASM 图执行。该测试平台的测试模板与图 11.54 所示的端口结构相适应。该模板说明了 *ASIC_with_TAP* 中的数据流程, 并说明了测试平台检测到一个已被引入到电路中的错误。该指令模板与在 *Instruction_Decoder* 模型中的指令代码相一致。下面给出了由 *TDI_Generator* 产生并由 *TDO_Monitor* 显示的测试序列。任务 *Load_ASIC_Test_Pattern* 执行一个测试序列, 以选择扫描值并将其装入 *TDI_Generator* 的 *TDI_Reg* 寄存器。当 *enableTDO* 或 *enable_bypass_pattern* 激活时, 这个值可从 *TDI_Generator* 中扫描出来。*TDO_Monitor* 包括一个两级流水线缓冲器, 其输入级接收要移入 ASIC 中的模板。第一级保存 *ASIC_with_TAP* 边界扫描寄存器中的现有值, 第二级存储由 *ASIC_with_TAP* 所保存的前一个值, 使得从 *ASIC_with_TAP* 中扫描到的实际值与期望值能够进行比较。将与 ASIC 对应的输出边界扫描寄存器单元中得到的数据和测试模板数据加以比较, 若两者不一致则表明有错误存在。测试平台包括一段可选部分的代码, 在这段代码内插入一些误差, 使得它能够与加法器产生的和相对应, 并且检查 *TDO_Monitor* 是否能检测出这个错误。

图 11.55 所示的仿真结果表明默认指令就是 BYPASS 指令。信号 c_{in} , b , a , c_{out} 和 sum 是 $ASIC_with_TAP$ 的外部端口; $c_{in_to_ASIC}$, b_{to_ASIC} 和 a_{to_ASIC} 是 ASIC 的输入端口, $c_{out_fr_ASIC}$ 和 sum_asic 是 ASIC 的输出端口。当仿真从 $t=0$ 开始时, 系统处于未知状态。在 TCK 的第一个有效沿, 状态机进入到 S_Reset(0) 状态^①, 并保持该状态(见图 11.55(a))直到 TMS 的输入序列通过 TAP(见图 11.55(b))^②扫描了 Pattern_Register(1354_H) 中的 10 个比特。在测试平台中 Pattern_Register 保存了 pattern_ptr 选取的模板。Load_TDI_Generator 脉冲将该模板装入 TDI_Reg (TDI_Generator 中)。TDI_Reg 的 LSB 驱动 TDI。在状态机进入 S_Shift_DR(4)之后, shiftDR 成立的同时, TCK 的 10 个周期通过旁路寄存器扫描了模板中的 10 个位。请注意, 图 11.55(a)中, 状态的转移发生在 TCK 的上升沿, 而当 enableTDO 激活时, TDO 的波形是 TDI 波形延迟一个周期的副本。同样应注意, clock_BSC_Reg 是固定的(即边界扫描寄存器处于空闲)。

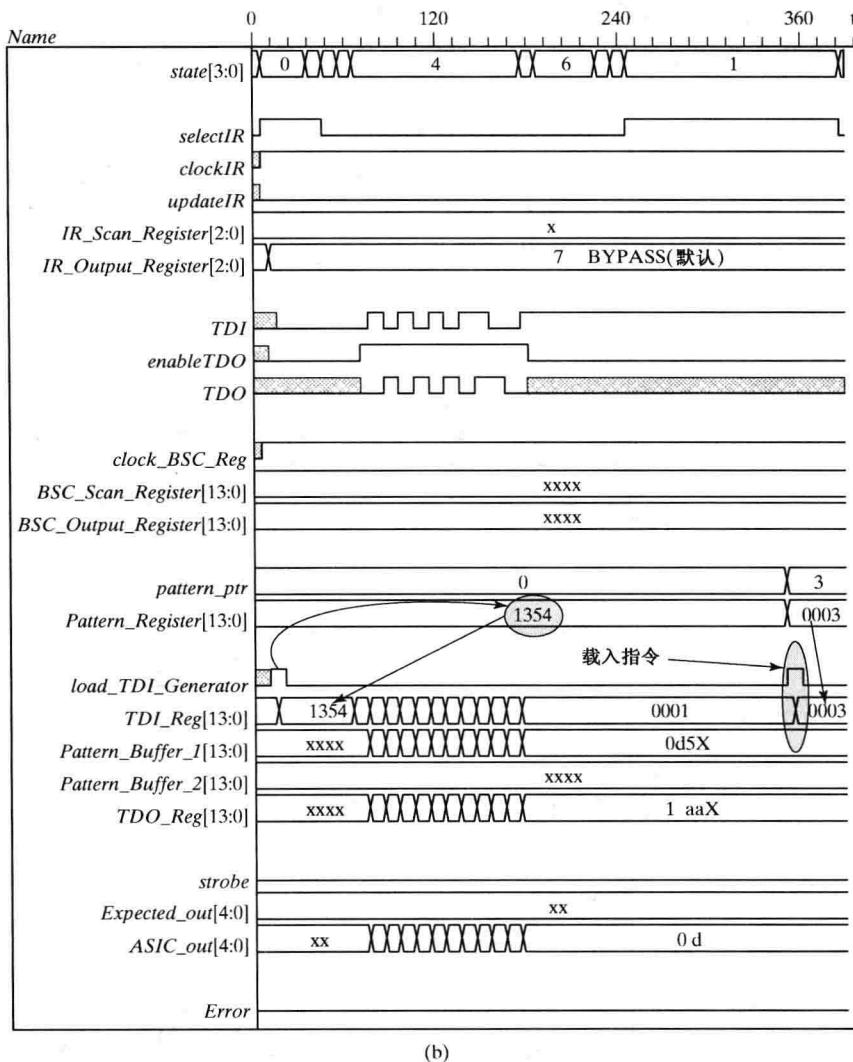


(a)

图 11.55 加电后通过 $ASIC_with_TAP$ 的旁路寄存器扫描模板的仿真结果: (a) 模板扫描通过芯片且延时一个时钟周期; (b) 控制信号、TAP 寄存器和测试平台寄存器

^① 参见本章末的习题 21。

^② 数据模板和测试序列间隔已在 TAP 的工作描述中予以说明。



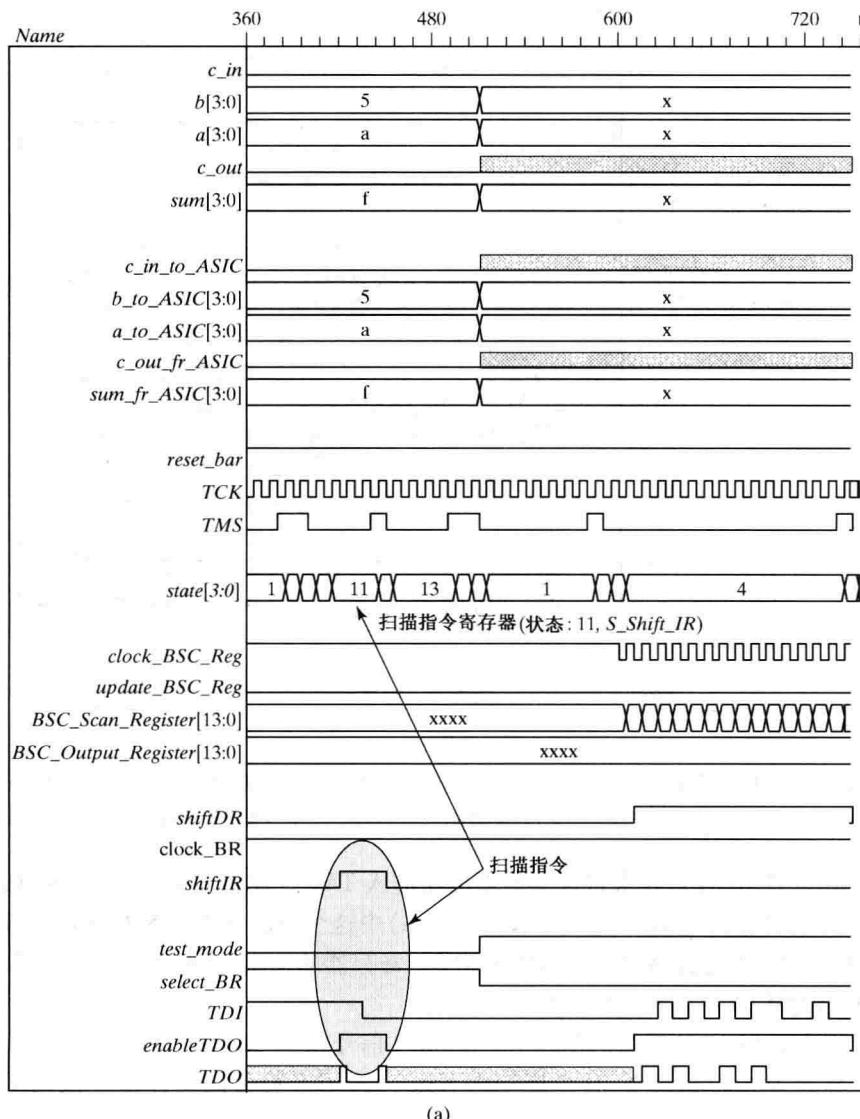
(b)

图 11.55(续) 加电后通过 *ASIC_with_TAP* 的旁路寄存器扫描模板的仿真结果: (a) 模板扫描通过芯片且延时一个时钟周期; (b) 控制信号、TAP寄存器和测试平台寄存器

旁路寄存器的 JTAG 规定要求寄存器的输出在进入 TAP 控制器状态 *S_Capture_DR* 之后的第一个 *TCK* 的上升沿时置为逻辑 0。注意, 图 11.56(a) 中这个边沿发生在 *S_Capture_DR* 和 *S_Shift_DR* 状态转换的时刻, 而且旁路寄存器的输出为 0。寄存器的输出是在 *TDO_enable* 激活之后的下一个 *TCK* 上升沿时从 *TDO* 中扫描出的值。

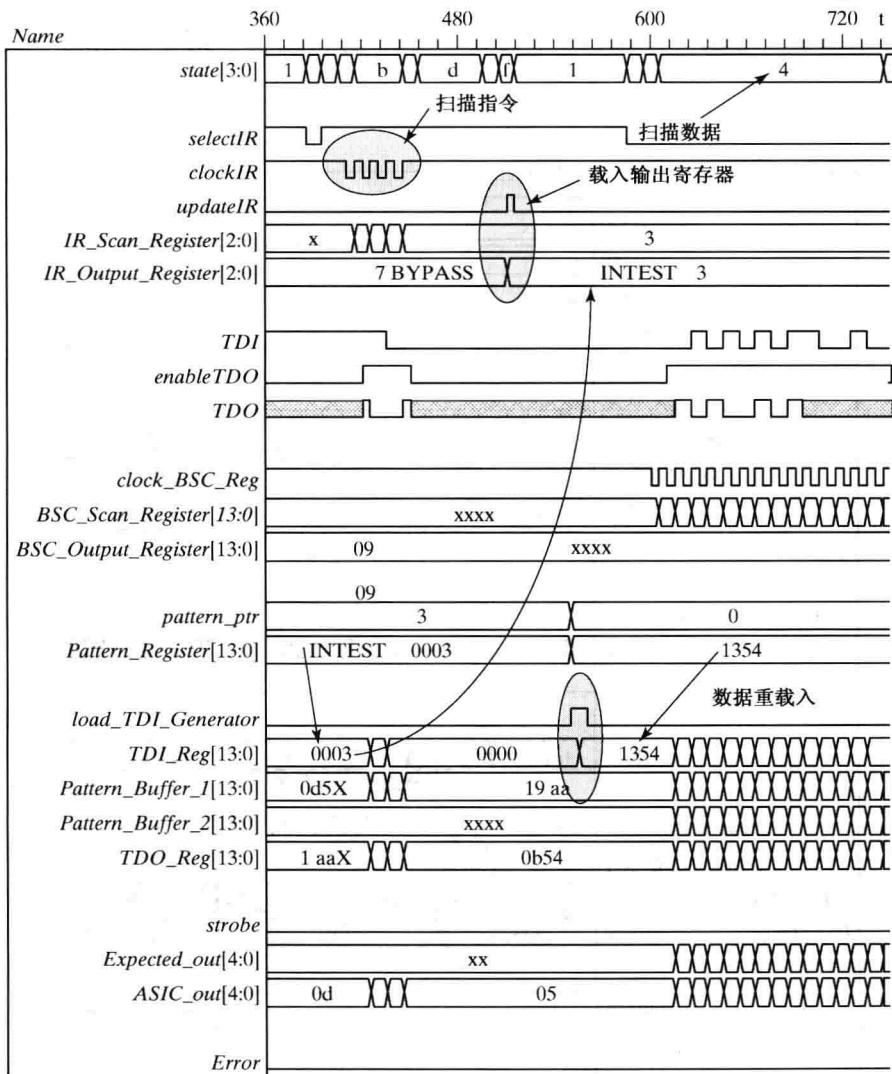
扫描过程不会影响 ASIC 端口的信号值。*clockBR* 信号在 *S_Capture_DR*(3) 状态下一个周期内有效, 而在 *S_Shift_DR*(4) 状态下 10 个周期内有效。*selectBR* 将旁路寄存器连接到 *TDO* 和 *TDI*。*BSC_Scan_Register* 和 *BSC_output_Register* 在没有接收数据时保持为 $14'Hx$ 。*reset_bar* 有效时(低电平有效)且在 *S_Reset*(0) 状态将 3 位指令寄存器置位为全 1(BYPASS 指令)。*TDO* 的后续位将会对 *TDI* 的波形进行复制。

图 11.57(a)所示的仿真结果表明：当机器处于状态 $S_Shift_IR(11)$ ，且 $shiftIR$ 和 $enableTDO$ 有效时，BYPASS 指令被移出 TAP，同时将 INTEST 指令装入 TAP。测试平台将 *Pattern_Register* 装载到 INTEST，然后激活 *Load_TDI_Generator*，将 *TDI_Reg* 载入到 INTEST。在状态 $S_Shift_IR(11)$ 下，当 BYPASS 通过 *TDO* 将寄存器内容扫描出来的同时，指令将扫描进寄存器 *IR_SCAN_Register*（见图 11.57(b)）。当 TAP 控制器进入状态 S_Update_IR 时，指令 INTEST 被装入 *IR_Output_Register*。图 11.57(b)所示的波形同样显示了测试平台将 1354_H 重新读入 *Pattern_Register*，将该模板传送到 *TDI_Reg*，并在状态处于 S_Shift_IR 时将模板扫描进 *BSC_Scan_Register*。同样要注意 *TDO* 的三态行为与 JTAG 标准相符合。



(a)

图 11.56 将指令 INTEST 装入指令寄存器的仿真结果：(a) $enableTDO$ 只在扫描时有效(否则 *TDO* 处于高阻态)；(b) 指令 INTEST 被装入，然后再装入数据模板，经由 *TDI* 从 *TDI_Generator* 扫描到 *ASIC_with_TAP*



(b)

图 11.56(续) 将指令 INTEST 装入指令寄存器的仿真结果: (a) enableTDO 只在扫描时有效(否则 TDO 处于高阻态); (b) 指令 INTEST 被装入, 然后再装入数据模板, 经由 TDI 从 TDI_Generator 扫描到 ASIC_with_TAP

随着 *IR_Output_Register* 保存 INTEST 指令, 模板(1354_{H})被装入图 11.56(b)中的 *BSC_Scan_Register*, 并被送入图 11.57(a)所示的 *BSC_Update_Register*, 用以对 ASIC 进行内部测试。请注意, *c_in_to ASIC*, *b_to ASIC*, *a_to ASIC* 的值会变成应用测试模板所指定的值, *c_out_fr ASIC* 和 *sum_fr ASIC* 由 ASIC^① 中的加法器产生。下一个测试模板(1354_{H})被装入 *TDI_Reg*(见图 11.57(b)), 并在前一个测试模板的结果被扫描出去后移入 *BSC_Scan_Register*。

① 本例中的加法器延时为 0。

测试过程在图 11.58 中完成。新的测试模板装入 *BSC_Output_Register*(见图 11.58(a)), 然后将 *Expected_out* 与图 11.58(b) 中的 *ASIC_out* 进行比较。两个模板相一致, 不存在 *Error*^①。

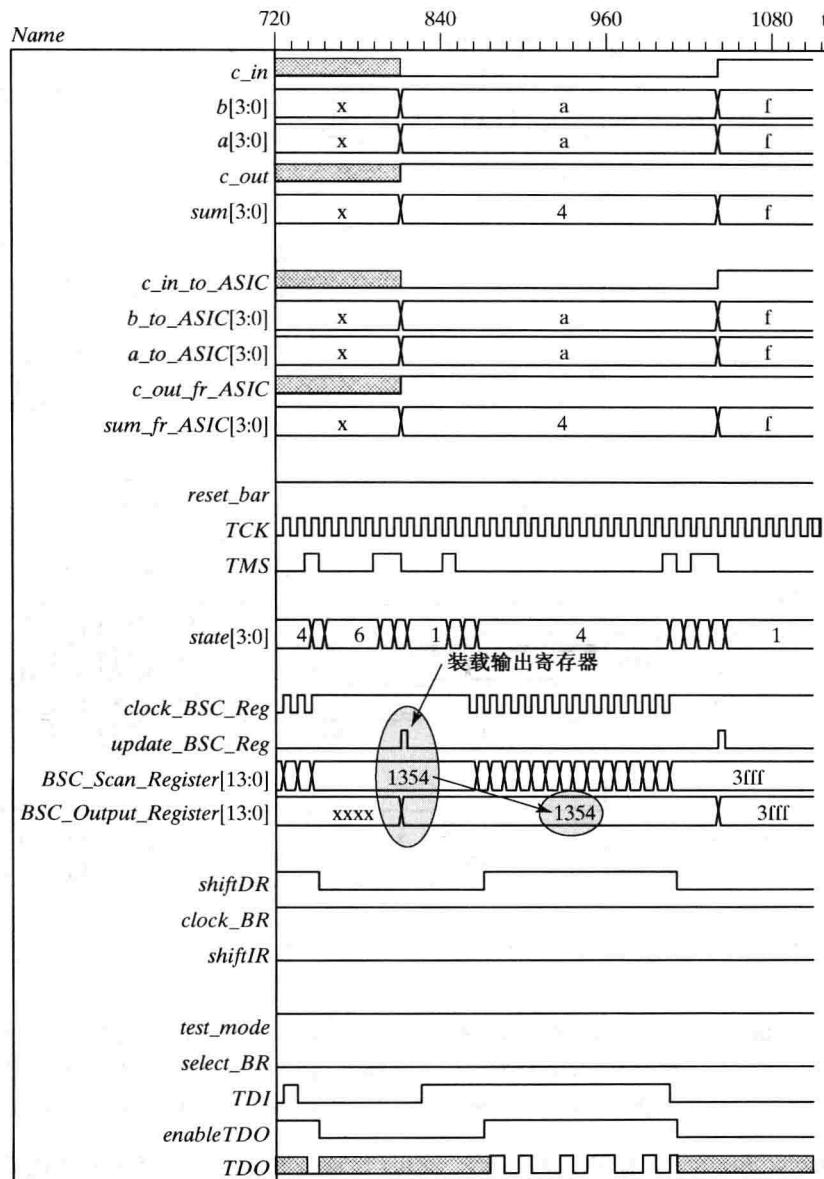


图 11.57 仿真结果: (a) 在测试模板 1354_H 扫描进捕获/扫描寄存器后, 载入边界扫描输出寄存器, 并将测试施加到 ASIC; (b) 经由 *TDO*, ASIC 的输出被捕获并扫描输出, 再移入 *TDO_Reg* 用于与 *Pattern_Buffer_2* 进行比较(见图 11.66)

① 测试平台包括一个将故障引入 ASIC 并通过施加测试模板将其检测出来的实例。

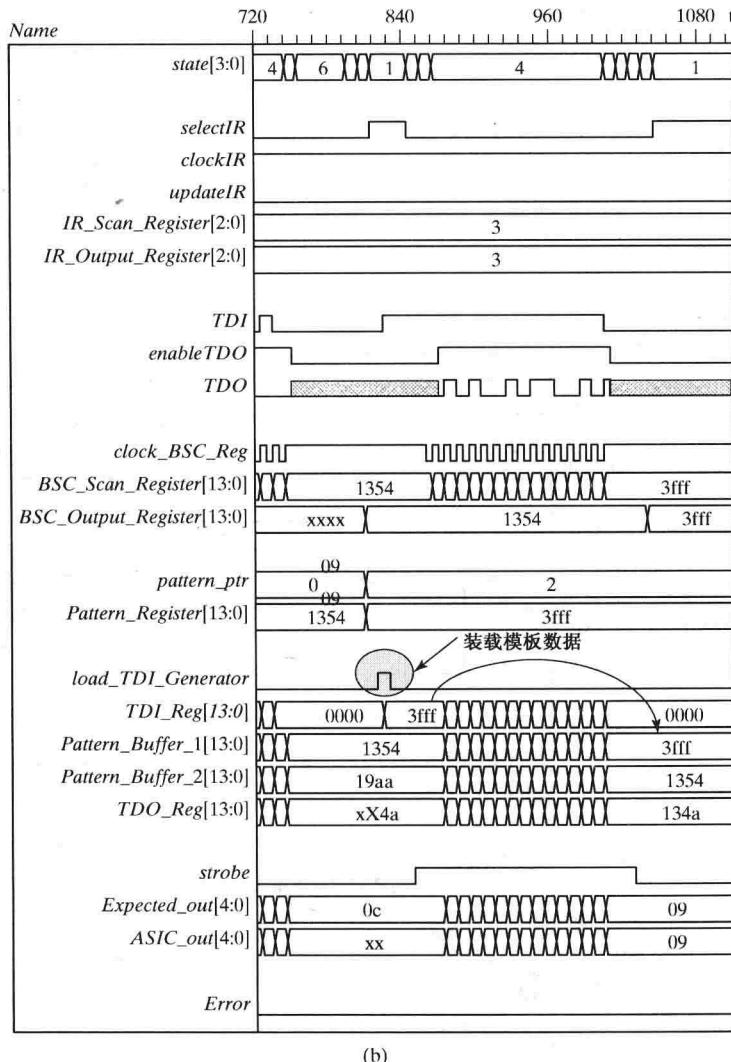


图 11.57(续) 仿真结果: (a) 在测试模板 1354_H 扫描进捕获/扫描寄存器后, 载入边界扫描输出寄存器, 并将测试施加到 ASIC; (b) 经由 TDO , ASIC 的输出被捕获并扫描输出, 再移入 TDO_Reg 用于与 $Pattern_Buffer_2$ 进行比较(见图11.66)

11.8.8 设计实例：内建自测试

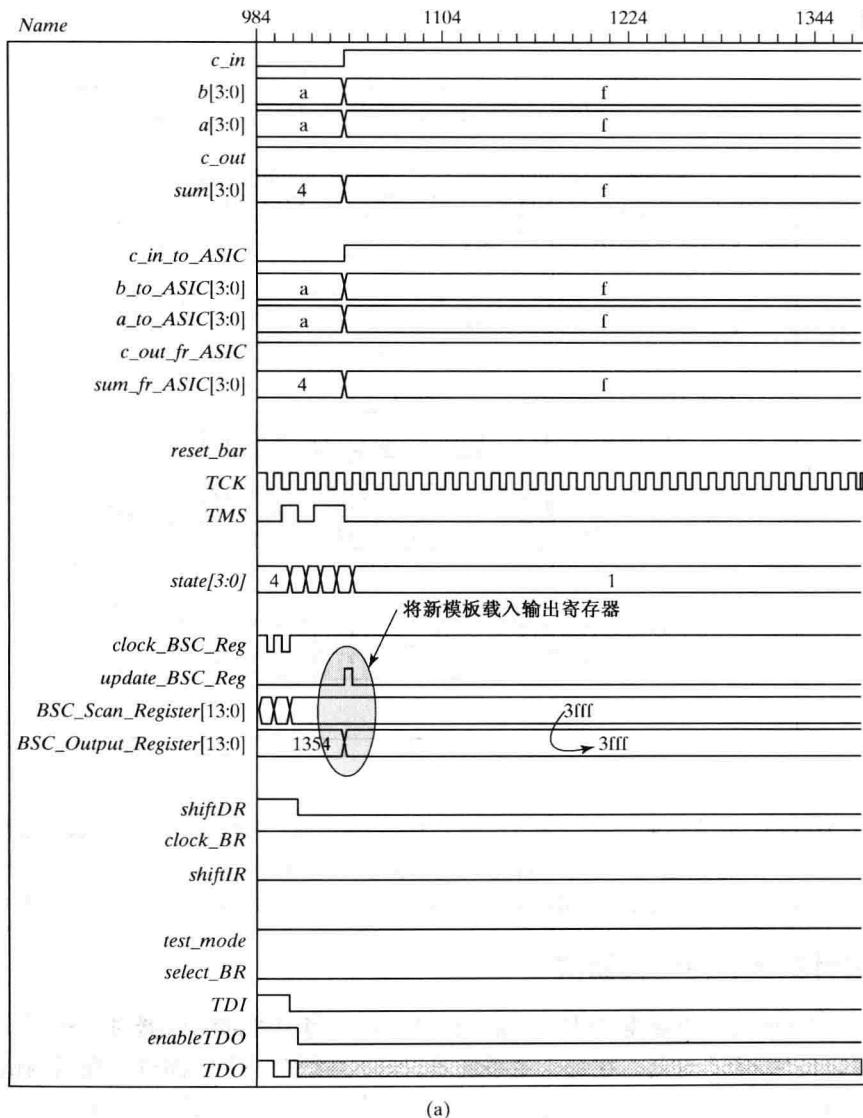
内建自测试(BIST)逻辑使得 ASIC 可以进行自测试。当用外部测试器对 ASIC 进行测试显得不可行时, 可以使用 BIST 电路。在每次主系统重启时, 必须对一些电路进行现场测试, 其他部分必须作为主电路板的外设进行测试。例如, 计算机和其他时序系统在启动时要用 BIST 检测 RAM 模块。

具有 BIST 硬件的结构如图 11.59 所示。在正常模式下待测试单元(UUT)由外部(原始)输入驱动, 但在测试模式下, 由内置电路产生测试模板并应用于该电路。电路的响应由附加硬件监测, 并与输入模板的期望响应进行比较。期望响应与实际响应之间的差别表明了存在内部故障。控制器(状态机)控制着施加激励和观察响应的全部过程。

将激励模板存入存储器中, 在测试模式下将其取出, 这样可以得到 BIST 的激励生成器。这

种方法和其他方法相比，需要相当大的存储量。下面将考虑采用线性反馈移位寄存器(LFSR)^①作为伪随机模板生成器(PRPG)，并采用一个多输入特征寄存器(MISR)来监视模板。采用 LFSR 作为 PRPG，是因为它们只需要很少的硬件就可生成大量的测试模板。

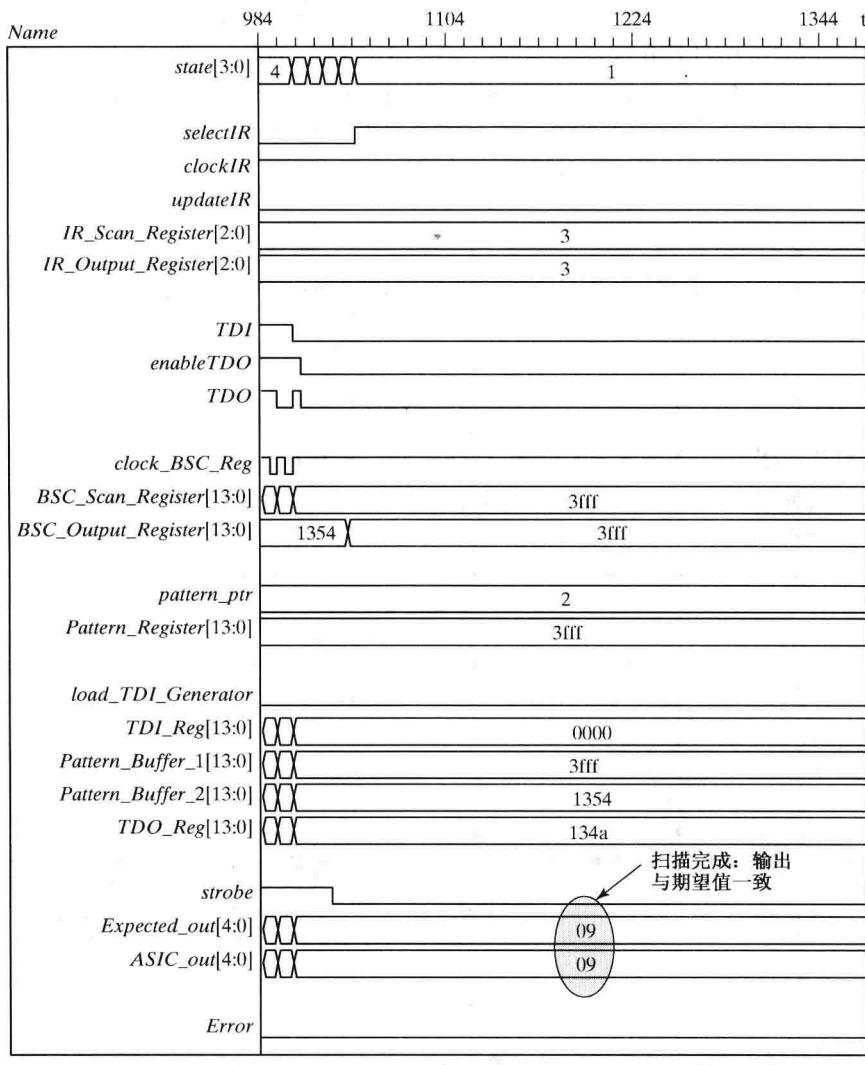
n 位自主 LFSR 的系数可以进行选择，用于产生重复周期为 $2^n - 1$ 的 n 位的伪随机序列模板(即模板序列是循环的)。这种生成模板的方法较为可取，因为生成模板序列所需要的硬件明显少于将相同模板存储于存储器中所需要的硬件。具有一个最简基本特性多项式的 LFSR 能够产生最大长度的模板序列^[16]。



(a)

图 11.58 将第二个数据扫描进 *ASIC_with_TAP* 之后的仿真结果：(a) 将模板(3fff_{16})装入 *BSC_Output_Register*；(b) *ASIC* 的期望输出与实际输出相一致

① 见第 5 章。



(b)

图 11.58(续) 将第二个数据扫描进 *ASIC_with_TAP* 之后的仿真结果: (a) 将模板(3fff_{H})装入 *BSC_Output_Register*; (b) ASIC 的期望输出与实际输出相一致

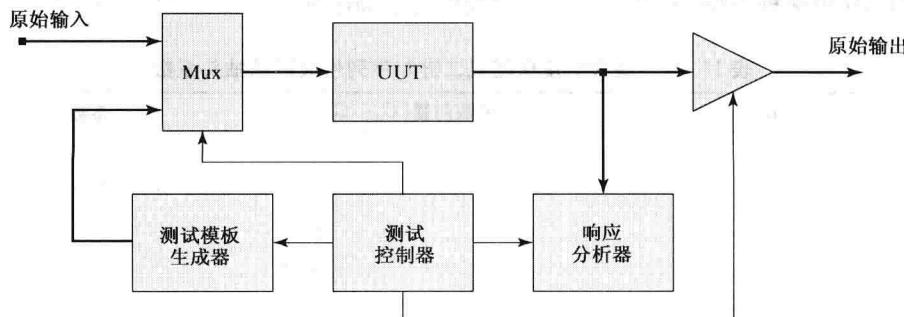


图 11.59 BIST 逻辑的结构图

图 11.60 给出了两种类型的 LFSR。类型 1 的 LFSR 是使用“外部”异或门对通用移位寄存器扩展得到的，这个类型的 LFSR 能够将同样的寄存器用于普通操作。图 11.60(a) 中类型 1 的移位寄存器分出一条通路将单元的输出反馈到链路中的第一个单元(最左边)。图 11.60(b) 所示类型 2 的结构在抽头系数为 1 的位置上的移位寄存器通路上有异或门。两种结构都能产生最大长度的伪随机二进制序列，具体取决于抽头系数。

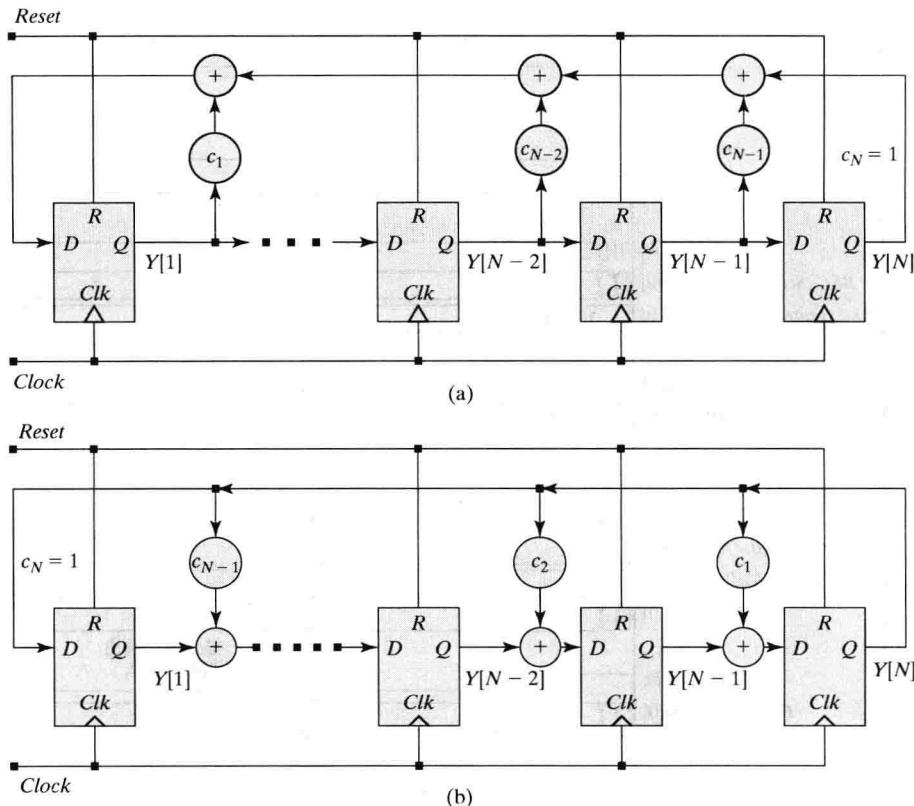


图 11.60 线性反馈寄存器：(a)类型 1(外部异或门)；(b)类型 2(内部异或门)

在测试中可首选类型 2 LFSR，因为它产生的模板比类型 1 的 LFSR 所产生的模板更加随机(相关性小)^[4]。生成的最大长度伪随机二进制序列的移位寄存器抽头系数由表 11.11 给出。请注意，图 11.60 中两种类型 LFSR 的抽头系数按照递增顺序标出，其顺序与表 11.11 相反。

表 11.11 最大长度伪随机二进制序列生成器的抽头系数

n	系数向量($C_n \cdots C_2 \ C_1$)	系数
2	11	C2C1
3	101	C3C1
4	1001	C4C1
5	1_0010	C5C2
6	10_0001	C6C1
7	100_0100	C7C3

(续表)

<i>n</i>	系数向量($C_n \cdots C_2 \ C_1$)	系数
8	1000_1110	C8C4C3C2
9	1_0000_1000	C9C4
10	10_0000_0100	C10C3
11	100_0000_0010	C11C2
12	1000_0010_1001	C12C6C4C1
13	1_0000_0000_1101	C13C4C3C1
14	10_0010_0010_0001	C14C10C6C1
15	100_0000_0000_0001	C15C1
16	1000_1000_0000_0101	C16C12C3C1
32	1000_0000_0010_0000_0000_0000_0011	C32C22C2C1

将基于 BIST 驱动的电路的响应与期望响应比较来判断该电路的工作是否正常。不必存储所期望的响应模板，而是可以用 MISR 将该电路产生的多个模板压缩而形成一个特征信号^[6]。将正确电路的响应所产生的特征信号存储起来用以与实际响应进行比较。因此，采用 MISR 电路和该特征信号就不再需要对各个不同测试模板的响应进行监控和比较了。图 11.61 中的 MISR 可通过该电路的响应模板来驱动。施加一个激励模板以后，电路的状态 Y 就是电路的特征信号。

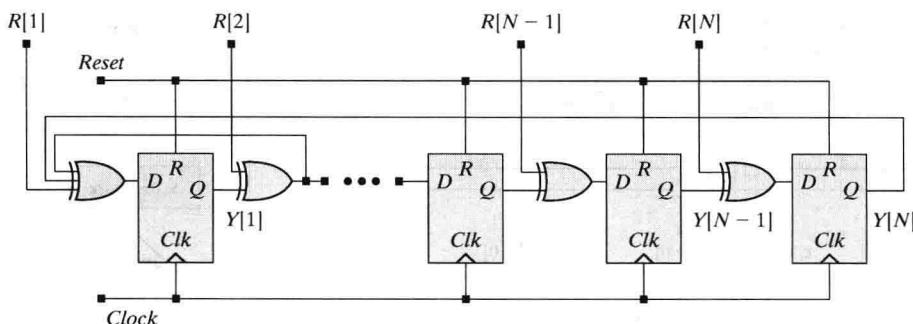


图 11.61 多输入线性反馈移位寄存器(MISR)

ASIC_with_BIST 电路显示了 ASIC 如何与其他硬件结合起来形成内建自测试电路。为了简便起见，其中的 ASIC 为一个 4 位带进位输入和输出的加法器。图 11.62 给出了 *ASIC_with_BIST* 的结构，其中包括加法器数据通路端口、*test_mode* 信号（能控制 *ASIC_with_BIST* 是测试模式还是普通模式）和一个能驱动内部状态机到复位状态的 *reset* 信号。*done* 信号在一个时钟周期内有效，表示 BIST 测试序列已经完成；*error* 表示 *response_Analyzer* 所产生的特征信号与存储中期望的特征信号不一致，因为测试模板序列是由 BIST 电路产生的。

ASIC_with_BIST 模型包括的 Verilog 模块有：*ASIC*，*Pattern_Generator*，*Response_Analyzer* 和 *BIST_Control_Unit*。BIST 的实现不会对 ASIC 做任何修正，ASIC 电路用 BIST 硬件进行测试。*Pattern_Generator* 是一个自定义的 LFSR，参数 *size* 用以指定 ASIC 中加法器的数据通路的大小；参数 *Length* 指定移位寄存器的长度，参数 *initial_state* 指定当外部复位信号有效时 LFSR 所处的状态。*Pattern_Generator* 中的最长 LFSR 将产生激励序列模板，*Response_Analyzer* 中的 MISR 将产生一个特征信号。在测试序列的末尾，*BIST_Control_Unit* 将比较该特征信号与所存储的模板，并在两者

不一致时激活 *error* 信号。图 11.62 中的复用器和三态输出缓冲器可通过 *ASIC_with_BIST* 的 Verilog 连续赋值语句来建模。

图 11.63 中的 ASM 图描述了 *ASIC_with_BIST* 的状态机控制器。*clock*, *reset* 和 *test_mode* 信号由主控平台驱动。BIST 电路包括一个可以决定测试序列长度的计数器。当施加测试模板时状态保持在 *S_test*, 然后转移到 *S_compare*, 在该状态把 *Response_Analyzer* 产生的特征信号与 *stored_pattern* 进行比较, 如果两者结果一致, 状态将转移到 *S_done*, 在一个时钟周期内激活 Moore 型输出信号 *done*, 然后再回到 *S_idle*; 如果两者不一致, 状态将转移到 *S_error*, 并保持该状态直到 *reset* 重新有效。

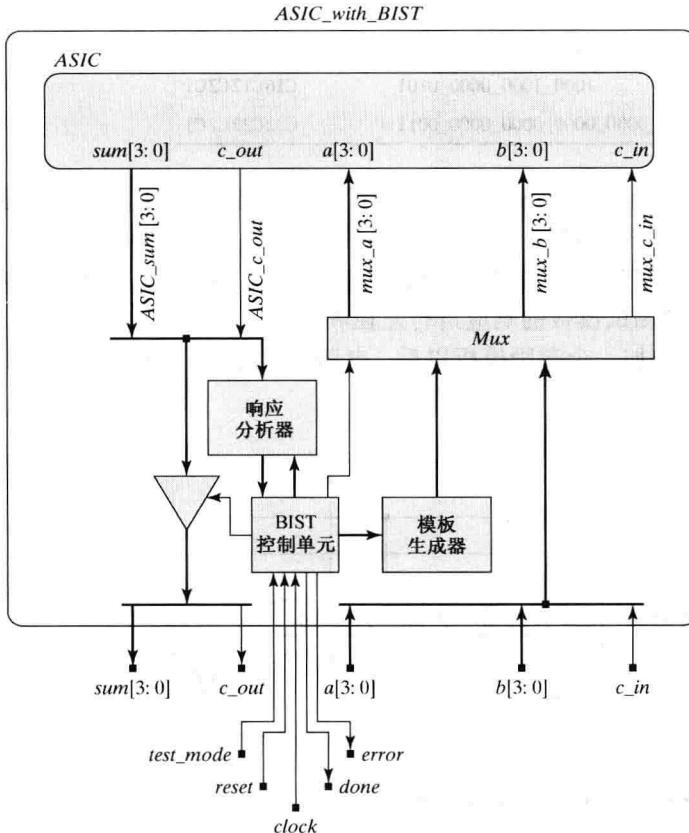


图 11.62 *ASIC_with_BIST* 的结构

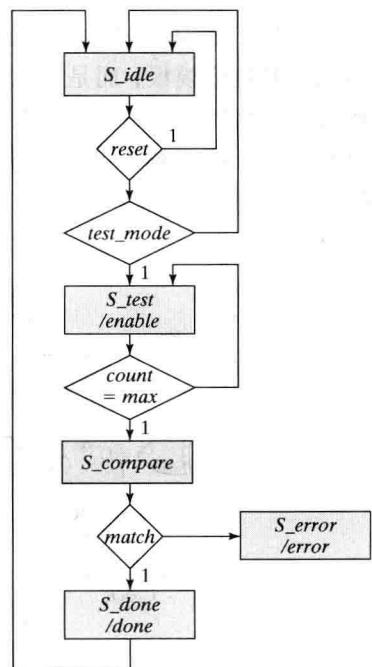


图 11.63 *ASIC_with_BIST* 中控制器的 ASM 图

```

module ASIC_with_BIST #(parameter size = 4)(
    output [size -1: 0]      sum, // ASIC interface I/O
    output                  c_out,
    input  [size -1: 0]      a, b,
    input  [size -1: 0]      c_in,
    output                  done, error,
    input   test_mode, clock, reset
);
    wire      [size -1: 0]  ASIC_sum;
    wire      [size -1: 0]  ASIC_c_out;
    wire      [size -1: 0]  LFSR_a, LFSR_b;

```

```

wire          LFSR_c_in;
wire [size -1: 0] mux_a, mux_b;
wire          mux_c_in;
wire          enable;
wire [1: size +1] signature;
assign {sum, c_out} = (test_mode) ? 'bz : {ASIC_sum, ASIC_c_out};
assign {mux_a, mux_b, mux_c_in} = (enable == 0) ? {a, b, c_in} :
{LFSR_a, LFSR_b, LFSR_c_in};

ASIC M0 (
.sum (ASIC_sum),
.c_out (ASIC_c_out),
.a (mux_a),
.b (mux_b),
.c_in (mux_c_in));
Pattern_Generator M1 (
.a (LFSR_a),
.b (LFSR_b),
.c_in (LFSR_c_in),
.enable (enable),
.clock (clock),
.reset (reset)
);
Response_Analyzer M2 (
.MISR_Y (signature),
.R_in ({ASIC_sum, ASIC_c_out}),
.enable (enable),
.clock (clock),
.reset (reset));
BIST_Control_Unit M3 (done, error, enable, signature, test_mode, clock, reset);
endmodule

module ASIC #(parameter size = 4)(
output [size -1: 0] sum,
output          c_out,
input  [size -1: 0] a, b,
input  c_in
);
assign {c_out, sum} = a + b + c_in;
endmodule

module Response_Analyzer #(parameter size = 5)(
input          [1: size] R_in,
input          enable, clock, reset
);
always @ (posedge clock)
if (reset == 0) MISR_Y <= 0;
end
endmodule

module Pattern_Generator #(parameter size = 4, Length = 9)(
output [size -1: 0] a, b,
output          c_in,
input  enable, clock, reset
);
reg      [1: Length] LFSR_Y;
parameter [1: Length] initial_state = 9'b1_1111_1111;
parameter [Length: 1] Tap_Coefficient = 9'b1_0000_1000;
integer    k;
assign a = LFSR_Y[2: size + 1];
assign b = LFSR_Y[size + 2: Length];
assign c_in = LFSR_Y[1];

```

```

always @ (posedge clock)
  if (reset == 1'b0) LFSR_Y <= initial_state;
  else if (enable) begin
    for (k = 2; k <= Length; k = k + 1)
      LFSR_Y[k] <= Tap_Coefficient[Length -k +1]
      ? LFSR_Y[k -1] ^ LFSR_Y[Length] : LFSR_Y[k -1];
    LFSR_Y[1] <= LFSR_Y[Length];
  end
endmodule

module BIST_Control_Unit #( parameter sig_size = 5, c_size = 10, size = 3,
  c_max = 510)(
  output reg      done, error, enable,
  input [1: sig_size]   signature,
  input           test_mode, clock, reset
);
  parameter stored_pattern = 5'h1a; // signature if fault-free
  parameter S_idle = 0,
  S_test = 1,
  S_compare = 2,
  S_done = 3,
  S_error = 4;
  reg      [size -1: 0]      state, next_state;
  reg      [c_size -1: 0]    count;
  wire     match = (signature == stored_pattern);

  always @ (posedge clock) if (reset == 0) count <= 0;
  else if (count == c_max) count <= 0;
  else if (enable) count <= count + 1;

  always @ (posedge clock) if (reset == 0) state <= S_idle;
  else state <= next_state;

  always @ (state, test_mode, count, match) begin
    done = 0;
    error = 0;
    enable = 0;
    next_state = S_error;
    case (state)
      S_idle:   if (test_mode) next_state = S_test; else next_state = S_idle;
      S_test:   begin enable = 1; if (count == c_max -1) next_state = S_compare;
                 else next_state = S_test; end
      S_compare: if (match) next_state = S_done;
                 else next_state = S_error;
      S_done:   begin done = 1; next_state = S_idle; end
      S_error:  begin done = 1; error = 1; end
    endcase
  end
endmodule

```

ASIC_with_BIST 的测试平台的操作如下：

- (1) 上电复位；
- (2) 热复位；
- (3) *sum* 和 *c_out* 的三态动作及 *test_mode* 有效时，选通输入数据通路；
- (4) 当 *enable* 通过 *BIST_Control_Unit* 得到激活时 LFSR 模板生成器和 MISR 动作的初始化；
- (5) 对 ASIC 的一个输入引脚引入的故障进行检测。

```

module t ASIC_with_BIST #(parameter size = 4, End_of_Test = 11000);
  wire      [size -1: 0]      sum;    // ASIC interface I/O
  wire      c_out;
  reg       c_in;
  wire      done, error;
  reg       test_mode, clock, reset;
  reg       Error_flag = 1;

  ASIC_with_BIST M0 (sum, c_out, a, b, c_in, done, error, test_mode, clock, reset);

  initial begin Error_flag = 0; forever @ (negedge clock) if ( M0.error)
    Error_flag = 1; end
  initial #End_of_Test $finish;

  initial begin clock = 0; forever #5 clock = ~clock; end
  // Declare external inputs
  initial fork
    a = 4'h5;
    b = 4'hA;
    c_in = 0;
    #500 c_in = 1;
  join
  // Test power-up reset and launch of test mode
  initial fork
    #2 reset = 0;
    #10 reset = 1;
    #30 test_mode = 0;
    #60 test_mode = 1;
  join
  // Test action of reset on-the-fly
  initial fork
    #150 reset = 0;
    #160 test_mode = 0;
  join
  // Generate signature of fault-free circuit
  initial fork
    #180 test_mode = 1;
    #200 reset = 1;
  join
  // Test for an injected fault
  initial fork
    #5350 release M0.mux_b [2];
    #5360 force M0.mux_b[0] = 0;
    #5360 begin reset = 0; test_mode = 1; end
    #5370 reset = 1;
  join
endmodule

```

图 11.64 的仿真结果表明上电复位将 *BIST_Control_Unit* 的状态驱动到 *S_idle*(0)，并将 LFSR 的状态复位为 1ffH。当 *test_mode* 有效后，*enable* 有效，该状态转移到 *S_test*(1)。*enable* 的激活连通了 LFSR(参见 *mux_a*, *mux_b* 和 *mux_c_in*)到 ASIC 的端口的数据通路，并将 *ASIC_with_BIST* 的输出数据通路(参见 *sum* 和 *c_out*)置为高阻态。在 *enable* 有效时，LFSR 可产生驱动 *ASIC_sum* 和 *ASIC_c_out* 内部数据通路的模板，*Response_Analyzer* 中的 MISR 生成初始特征信号。*reset* 的第二次激活表明机器在进行热复位。

- (1) 上电复位；
- (2) 热复位；
- (3) *sum* 和 *c_out* 的三态动作及 *test_mode* 有效时，选通输入数据通路；
- (4) 当 *test_mode* 有效时 LFSR 和 MISR 行为的初始化。

图 11.65 的仿真结果表明无故障电路的特征信号与所存模板相一致。在 510 个时钟周期之后，机器状态进入 *S_compare*(2)，检测到结果是匹配的，然后状态又进入到 *S_done*(3) 并在该状态保持一个周期后返回到 *S_idle*。当施加多测试序列来检测引入的故障时，测试平台包括用来检测该故障的 *Error_flag*。在 ASIC 的一个输入引脚引入的故障，其仿真结果如图 11.66 所示。当电路引入故障后，仿真结果表明：无故障电路的 *storage_pattern* 与 MISR 产生的特征信号之间不一致。

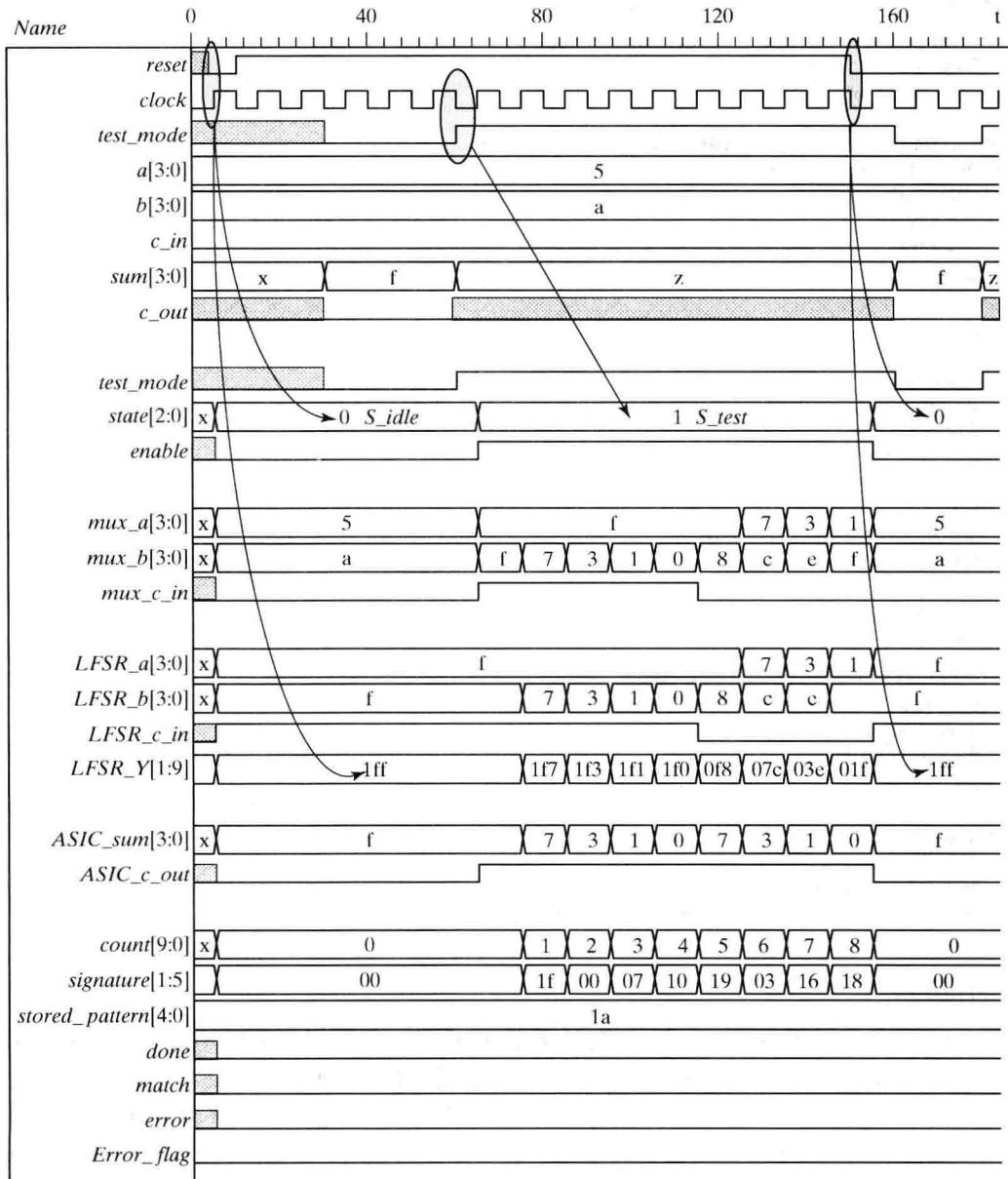


图 11.64 仿真结果表明：(1)上电复位；(2)热复位；(3)当 *test_mode* 有效时，*sum* 和 *c_out* 进入三态，选择输入数据通路；(4)当 *test_mode* 有效时，LFSR 和 MISR 初始化

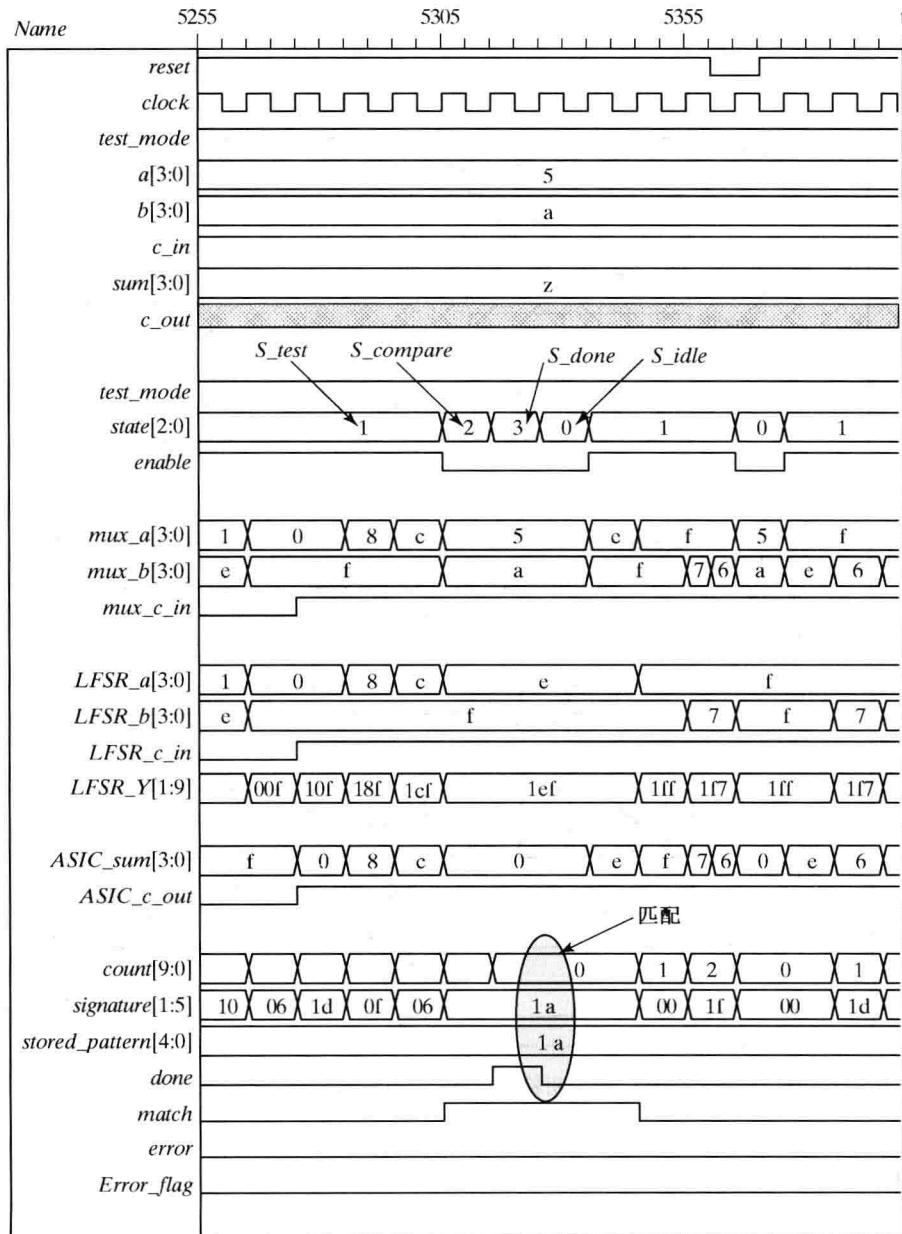
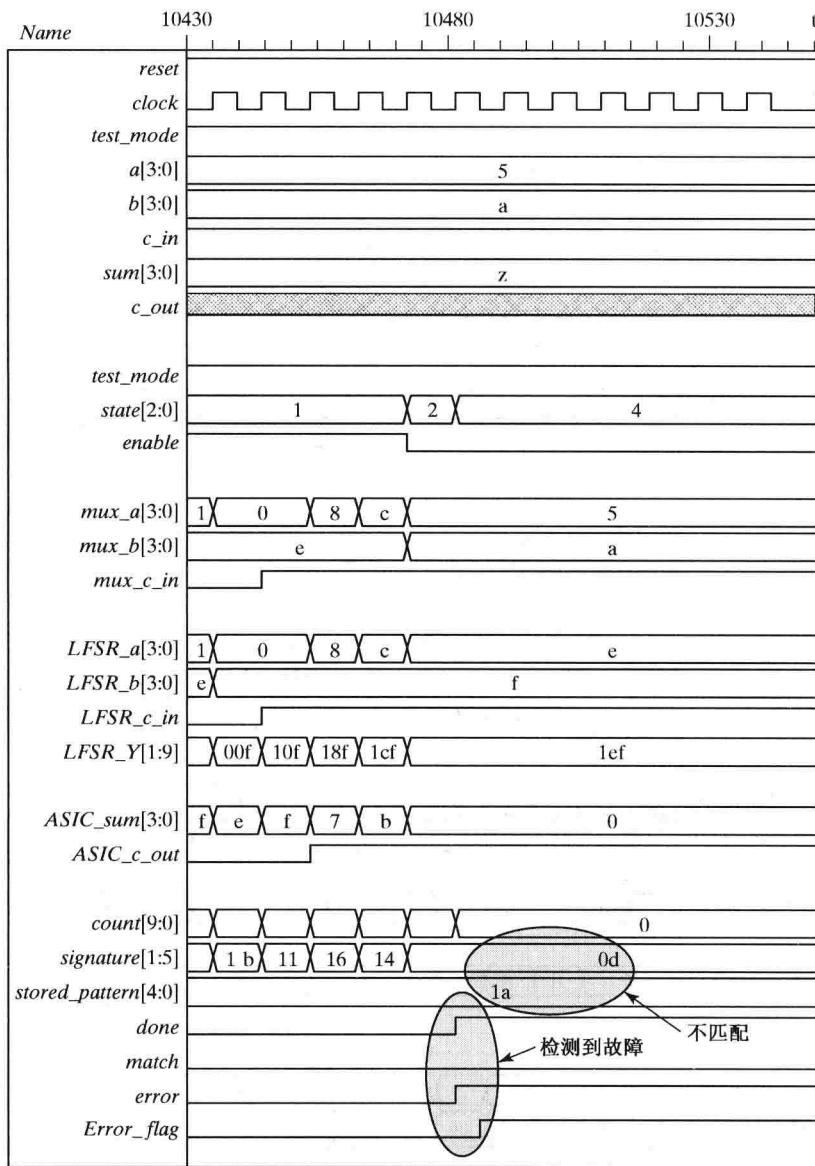


图 11.65 仿真结果表明 stored_pattern 和无故障电路的特征信号匹配

图 11.66 仿真结果表明检测到 *ASIC_with_BIST* 的一个引入故障

参考文献

1. Howe H. "Pre-and Postsynthesis Simulation Mismatches." Proceedings of the Sixth International Verilog HDL Conference, March 31-April 3, 1997, Santa Clara, CA.
2. McWilliams TM. "Verification of Timing Constraints on Large Digital Systems." Ph. D. Thesis, Stanford University, 1980.
3. Osterhout JK. "Crystal: A Timing Analyzer for nMOS VLSI Circuits." In: Bryant R, ed. *Proceedings of the Third Caltech Conference on VLSI*. Rockville, MD: Computer Science Press, 1983, 57-69.