摘要:

本文介绍了数字集成电路设计中静态时序分析（**Static Timing Analysis**）和形式验证（**Formal Verification**）的一般方法和流程。这两项技术提高了时序分析和验证的速度，在一定程度上缩短了数字电路设计的周期。本文使用 **Synopsys** 公司的 **PrimeTime** 进行静态时序分析，用 **Formality** 进行形式验证。由于它们都是基于 **Tcl**（**Tool Command Language**）的工具，本文对 **Tcl** 也作了简单的介绍。

关键词:

静态时序分析　形式验证　**PrimeTime**　**Formality**　**Tcl**

目　录

第一章 绪论

我们知道，集成电路已经进入到了 VLSI 和 ULSI 的时代，电路的规模迅速上升到了几十万门以至几百万门。而 IC 设计人员的设计能力则只是一个线性增长的曲线，远远跟不上按照摩尔定律上升的电路规模和复杂度的要求。这促使了新的设计方法和高性能的 EDA 软件的不断发展。

Synopsys 公司的董事长兼首席执行官 Aart de Geus 曾经提到，对于现在的 IC 设计公司来说，面临着三个最大的问题：一是设计中的时序问题；二是验证时间太长；三是如何吸引并留住出色的设计工程师。他的话从一个侧面表明了，随着 IC 设计的规模和复杂度的不断增加，随着数百万系统门的设计变得越来越普遍，时序分析和设计验证方面的问题正日益成为限制 IC 设计人员的瓶颈。

对于这些问题，设计者们提出的策略有：创建物理综合技术、开发更快更方便的仿真器，使用静态时序分析和形式验证技术、推动 IP 的设计和应用等等。本文将着重于探讨其中的静态时序分析和形式验证两项技术，在集成电路设计日益繁复的背景下，它们为 IC 产品更快更成功地面对市场提供了可能。

§1.1 静态时序分析

一般来说，要分析或检验一个电路设计的时序方面的特征有两种主要手段：动态时序仿真（Dynamic Timing Simulation）和静态时序分析（Static Timing Ana-lysis）。

动态时序仿真的优点是比较精确，而且同后者相比较，它适用于更多的设计类型。但是它也存在着比较明显的缺点：首先是分析的速度比较慢；其次是它需要使用输入矢量，这使得它在分析的过程中有可能会遗漏一些关键路径(critical pat-hs)，因为输入矢量未必是对所有相关的路径都敏感的。

静态时序分析的分析速度比较快，而且它会对所有可能的路径都进行检查，不存在遗漏关键路径的问题。我们知道，IC 设计的最终目的是为了面对竞争日益激

烈的市场，Time-to-market 是设计者们不得不考虑的问题，因此对他们来说，分析速度的提高，或者说分析时间的缩短，是一个非常重要的优点。

§1.2 形式验证技术

我们知道，验证问题往往是 IC 产品开发中最耗费时间的过程之一，而且它需要相当多的计算资源。开发一个带有相应的测试向量的测试平台是很费时的工作，而且它要求开发者必须对设计行为有很好的很深入的理解。而形式验证技术，简单地说就是将两个设计－－或者说一个设计的两个不同阶段的版本－－进行等效性比较的技术，由于能够很有效地缩短为了解决关键的验证问题所花费的时间，正在逐渐地被更多的人接受和使用。这方面的工具有 Synopsys 公司的 Formality 和 Verp-lex 公司的 Conformal LEC 等。

本文将讨论使用 Synopsys 的工具 PrimeTime 和 Formality 进行静态时序分析和形式验证的一般方法和流程。本文的第二章简要介绍了 PrimeTime 的基本功能和特点。第三章介绍了 Tcl 在 PrimeTime 中的基本使用，重点是关于对象和属性的操作。第四章介绍了在进行静态时序分析之前要作的准备工作。第五章介绍了对一个具体例子进行静态时序分析的过程。第六章介绍了 Formality 的基本特点和验证流程。第七章介绍了对一个具体例子进行形式验证的过程。第八章介绍了对验证失败的设计进行 Debug 的各种技巧。

第二章 PrimeTime 简介

正如本文前面所提到的，静态时序分析方法由于有着更快的分析速度等优点，正在被更多的设计者们所重视。**PrimeTime** 是 **Synopsys** 的静态时序分析软件，常被用来分析大规模、同步、数字 **ASIC**。**PrimeTime** 适用于门级的电路设计，可以和 **Synopsys** 公司的其它 **EDA** 软件非常好的结合在一起使用。

这一章将简要介绍 **PrimeTime** 的基本功能和特点，以及使用 **PrimeTime** 进行静态时序分析的一般过程。

§2.1 PrimeTime 的特点和功能

作为专门的静态时序分析工具，**PrimeTime** 可以为一个设计提供以下的时序分析和设计检查：？？

- 建立和保持时间的检查**(setup and hold checks)**
- 时钟脉冲宽度的检查
- 时钟门的检查**(clock-gating checks)**
- **recovery and removal checks**
- **unclocked registers**
- 未约束的时序端点（**unconstrained timing endpoints**）
- **master-slave clock separation**
- **multiple clocked registers**
- 组合反馈回路（**combinational feedback loops**）
- 基于设计规则的检查，包括对最大电容、最大传输时间、最大扇出的检查等。

PrimeTime 具有下面的特点：

1）PrimeTime 是可以独立运行的软件，它不需要逻辑综合过程中所必需的各种数据结构，而且它对内存的要求相对比较低。

2）PrimeTime 特别适用于规模较大的、**SOC**（**system-on-chip**）的设计。

在数字集成电路设计的流程中，版图前、全局布线之后已经版图后，都可以使用 **PrimeTime** 进行静态时序分析。

§2.2 PrimeTime 进行时序分析的流程

使用 **PrimeTime** 对一个电路设计进行静态时序分析，一般要经过下面的步骤：

1）设置设计环境

在可以进行时序分析之前，首先要进行一些必要的设置和准备工作。具体来说包括了：

- 设置查找路径和链接路径
- 读入设计和库文件
- 链接顶层设计
- 对必要的操作条件进行设置，这里包括了线上负载的模型、端口负载、驱动、以及转换时间等
- 设置基本的时序约束并进行检查

2）指定时序约束 **(？？ timing assertions/constraints)**

包括定义时钟周期、波形、不确定度**(uncertainty)**、潜伏性**(latency)**，以及指明输入输出端口的延时等。

3）设置时序例外**(？？ timing exceptions)**：

这里包括了：

- 设置多循环路径（**multicycle paths**）
- 设置虚假路径（**false paths**）
- 定义最大最小延时、路径的分段**(path segmentation)**以及无效的 **arcs**

4）进行时序分析：

在作好以上准备工作的基础上，可以对电路进行静态时序分析，生成 **constra -int reports** 和 **path timing reports**。

以上仅仅是 **PrimeTime** 进行静态时序分析的简单流程，在本文以下的部份中将会有更详细的叙述。

§2.3 静态时序分析中所使用的例子

在本文中，进行静态时序分析时所用的例子是微处理器 **AMD 2910**，图 **2-2** 给出了它的顶层的电路图。

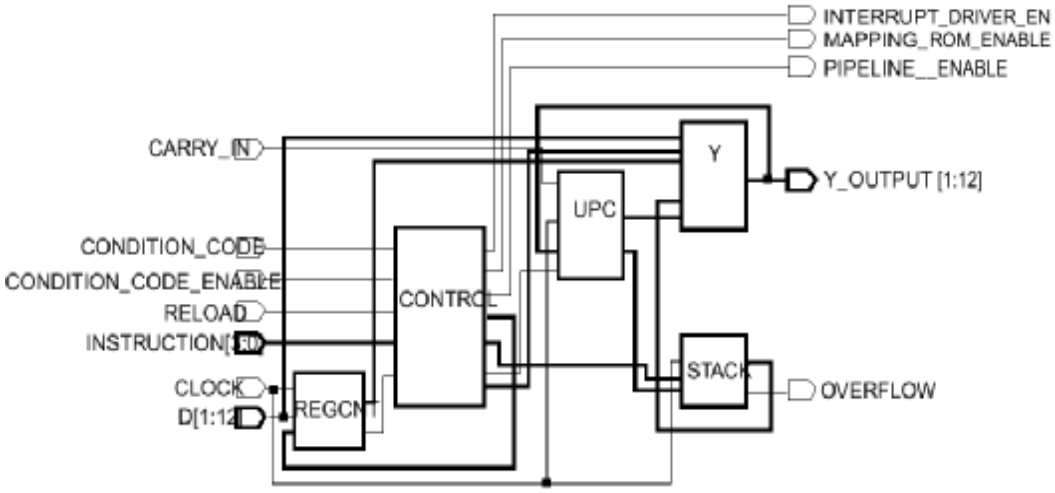


Figure2-2 AMD 2910 微处理器

§2.4 PrimeTime 的用户界面

PrimeTime 提供两种用户界面，图形用户界面 **GUI**（**Graphical User Interf -ace**）和基于 **Tcl** 的命令行界面 **pt_shell**，其运行方式分别是：

% **PrimeTime**

% **pt_shell**

退出的命令是 **quit**、**exit** 或者**^d**。事实上，在 **GUI** 界面中通过菜单进行的每一个操作，都对应着相应的 **pt_shell** 的命令。因此，本文以下的章节都只针对于 **pt_ shell** 来完成。

第三章 Tcl 与 pt_shell 的使用

Tcl 是 **Tool Command Language** 的缩写，由于 **PrimeTime** 的命令语言是基于 **Tcl** 标准的，所以在这一章里我想大致介绍一下 **Tcl** 在 **PrimeTime** 中的基本使用。除了一些最常用的 **Tcl** 命令之外，主要介绍了 **pt_shell** 中有关对象和属性的操作。

事实上，大多数 **synopsys** 公司的 **EDA** 工具都是基于 **Tcl** 标准的。例如在第二章的图 **2-1** 中可以看到的综合软件 **Design Compiler** 也是。由于都基于 **Tcl** 标准，**PrimeTime** 中的大多数命令以及命令参数都和 **Design Compiler** 中是相同的。

§3.1 Tcl 中的变量

我们可以把 **Tcl** 看作是一种比较高级的语言，它很容易理解和使用，所以这里对它的介绍也是很简单的。

与变量有关的有下列操作：

1）定义变量：**set** 变量名 变量值

例如： **set clock_period 10**

2）引用变量：**$**变量名

例如： **echo $clock_period**

3）删除变量：**unset** 变量名

4）打印变量：**printvar** 变量名（无变量名时打印所有变量）

或者： **echo $**变量名

举个例子，在使用 **PrimeTime** 之前，我们可以把它设置成分页显示，以便于浏览在运行时生成的信息，此时可以使用如下的命令：

set sh_enable_page_mode true

如果希望每次运行时 **PrimeTime** 总是分页显示，可以到**.synopsys_pt.setup** 文件中去更改 **sh_enable_page_mode** 变量的设置。

§3.2 命令的嵌套

在使用 PrimtTime 的过程中，命令的嵌套经常会被用到。嵌套命令时，用方括号([])分隔开每一层的命令，例如：

命令 1 [命令 2 [命令 3]]

在这样一个例子中，命令 3 首先被执行，它的结果将被作为命令 2 的一个参数，然后依次执行下去。

§3.3 文本的引用

在 Tcl 中，可以使用两种方法来引用文本或者说字符串：

1）弱引用：使用双引号来引用文本。在双引号里出现的变量、命令和反斜杠不会被转义，仍然保持特殊意义。

2）强引用：使用大括号来引用文本。大括号中的字符串将按照字面上被引用。

例如：
```
set mydelay 10
echo "The value of mydelay is $mydelay"
```
得到的结果将是：The value of mydelay is 10，而
```
echo {The value of mydelay is $mydelay}
```
得到的结果将是：The value of mydelay is $mydelay。

除此之外，可以使用反斜杠来转义一个单一的特殊字符，以及使用 expr 命令来得到算术表达式的值。

§3.4 PrimeTime 中的对象

§3.4.1 对象的概念

在 IC 设计中，“对象(object)”是一个常用的概念。一般来说，一个设计会包含以下的对象：Design、Cell、Port、Pin、Net、Clock 等。在分析和验证的过程中，也经常要跟这些对象打交道。因此搞清楚这些概念，才不会在使用软件的过程中遇到不必要的障碍。

Design：有一定逻辑功能的电路描述，它可以是独立的，也可以包含有其他的子设计。虽然严格地来说子设计只是设计的一部份，但是 **Synopsys** 也把它看作是一个 **design**。

Cell：在 **Synopsys** 的术语中，**cell** 和 **instance** 被认为是同样的概念，都是 **design** 中例化的一个具体元件。

Port：指主要的 **input**、**output** 或者 **design** 的 **I/O** 管脚。

Pin：对应于设计中的 **cell** 的 **input**、**output** 或者 **I/O** 管脚。

Reference：**cell** 或者 **instance** 参考的源设计的定义。

Net：是指信号的名字，即通过连接 **ports** 与 **pins** 或者 **pins** 与 **pins** 而把一个设计连接在一起的金属线的名字。

Clock：作为时钟源的 **port** 或者 **pin**。

下面的例子是用 **VHDL** 语言描述的一个电路，包含了上面所说的各种对象：

Figure3-1

§3.4.2 在 PrimeTime 中使用对象

PrimeTime 提供了一个命令来选中这些对象，或者更准确地说是建立一个这些对象的 **collection**，命令的形式为：

```
get_objtype
```

其中 **objtype** 是这种对象的类型，可以使用“**help get_***”来查看。这个命令与 **Design Compiler** 中的 **find** 命令有点类似。

值得注意的是，这个命令并不是返回被选中对象的列表，而是建立一个指向被选中对象的 **collection**。这种方法比在 **memory** 中保存一个庞大的列表要节省时间和资源。要列举所选中的对象，可以用 **query_objects** 命令。通常可以把建立的 **collection** 设置为变量，以方便使用，例如：

```
set data_ports [get_ports D[*] ]
query_objects $data_ports
```

另外，也可以用 **all_objtype** 命令来建立某种对象的 **collection**，例如：**all_clocks**、**all_inputs**、**all_outputs**、**all_instances**、**all_registers**、

all_connected 等。其中 **all_connected** 的作用是列出与某对象连接的所有 **pin**、**port** 或者 **net**，例如要得到所有连接到 **CLOCK** 的对象，可以：

```
query_objects [all_connected [get_nets CLOCK]]
```

§3.4.3 针对 collection 的操作

1）foreach_in_collection

顾名思义，这个命令的作用是对一个 **collection** 中的所有的对象进行某种操作，其形式为

```
foreach_in_collection variable collection { body }
```

例如，要打印出端口总线 **Y_OUTPUT** 的电容：

```
foreach_in_collection outpin [get_ports Y_OUTPUT[*]] {
? set maxcap [get_attribute $outpin wire_capacitance_max]
? set pinnname [get_attribute $outpin full_name]
? echo "Max capacitance of port $pinname is $maxcap"
? }
```

说明：①问号表示命令尚未结束，出现在引号，大括号和中括号里。

②这个命令比较复杂，它的执行过程是这样的：执行 **get_ports** 命令得到 **collection**；依次定义三个新的变量：**outpin**、**maxcap** 以及 **pinname**；然后对于 **collection** 中的每一项，依次执行 **echo** 命令。

foreach_in_collection 命令是针对于 **collection** 的，对于一般的列表可以使用 **Tcl** 中的标准命令 **foreach**。

2）从 collection 中增加或删除对象

```
add_to_collection collection object
remove_from_collection collection object
```

其中的 **object** 表示要增加或者删除的对象，这两个命令将返回一个新的 **collection**。

3）collection 的过滤

根据一定的条件对 **collection** 进行过滤，可以使用 **filter_collection** 命令，它将在 **collection** 中寻找符合条件的对象，并返回一个新的 **collection**，如果没有匹配的对象的话将返回空的字符串。也可以 **collection** 命令中使用**-filter** 参数。

filter 命令中的条件表达式可以使用以下的运算符：**==,!=,>,<,>=,<=,=~**

例如，要列出名字是 **ND2,ND21,ND3,ND4p** 诸如此类的 **cells**：

```
query [get_cells * -hier -filter "ref_name =~ ND*"]
```

说明：在使用 **PrimeTime** 时可以使用缩略的命令，这里就使用了 **query** 来代替 **query_objects**。

§3.5 属性

属性**(arritubes)**可以是 **PrimeTime** 预定义的，也可以是从综合软件如 **Design Compiler** 继承下来的**(**例如时钟周期、输入延迟、**net** 的电容等**)**，也可以是由用户定义的。

与属性相关的命令有：

list_arritubes

get_arritube

report_arritube

define_user_arritube

set_user_arritube

remove_user_arritube

利用这些命令可以很方便地了解设计，例如想知道 **AM2910** 设计中的最大电压，可以：

```
get_attribute [get_designs AM2910] voltage_max
```

§3.6 查看命令

与 **Unix** 相似，可以使用 **help** 或者 **man** 来查看命令的用法。例如查看与 **clock** 相关的命令可以：

```
help *clock
```

查看命令的参数可以用如下的命令：

```
help -verbose
```

其他的命令这里就不再赘述了，在后面的章节中使用到时再一一作介绍。

第四章　静态时序分析前的准备工作

从第二章里的时序分析流程可以看到，在对一个设计进行静态时序分析之前，首先要作一些基本的环境设置和准备工作，包括：

- 设置查找和链接路径；
- 读入并链接所要分析的设计；
- 设置操作条件和线上负载模型**(wire load model??)**；
- 设置基本的时序约束**(??timing assertions)**；
- 检查所设置的约束以及该设计的结构。

这一章将依次介绍这些内容。

§4.1　编译时序模型

§4.1.1　编译 Stamp Model

Stamp model 是针对于复杂模块——例如 **DSP(digital signal processing)**的核心或者 **RAM**——而建立的静态时序模型。它一般是为晶体管级的设计而创建的，在设计中没有门级的网表。**Stamp model** 中可以包含的时序信息有：**pin-to-pin** 的 **arcs(??)**、建立和保持时间、**pin** 的电容和负载、以及三态输出、锁存器、内建时钟等。

一个 **Stamp model** 包括两个源文件：

1）**.mod** 文件，包含对 **pin-to-pin** 的 **arc** 的描述（不包括延时）。

2）**.data** 文件，包括**.mod** 文件中描述的每一个 **arc** 的延迟数据。

例如，对于 **AM2910** 设计中的 **Y** 模块（见图 **2-2**），编译其 **Stamp model**：

compile_stamp_model -model_file Y.mod -data_file Y.data -output Y

其中 **Y.mod** 和 **Y.data** 是源文件，编译后生成了两个**.db** 文件：**Y_lib.db** 和 **Y.db**。其中 **Y_lib.db** 是一个库文件，其中包括一个被称为 **core** 的单元；而 **Y.db** 是一个设计文件，是 **Y_lib.db** 中的单元 **core** 的例化。正是由于它们之间的这种关系，为了在链接时能够正确地例化 **Y.db**，库 **Y_lib.db** 必须要加入到链接路径**(link**

_path)中去。

§4.1.2 编译快速时序模型

对于设计中的某些未完工的模块——??比如说，该模块的 HDL，或者完整的 stamp model 没有完成——你可以创建一个快速时序模型来进行分析。快速时序模型是一个临时性的模型，可以提供进行时序分析而需要的时序信息。实际上，快速时序模型是包含一系列 pt_shell 命令的文件，而不是语言。这样比使用 Stamp mo -del 的语言去写一个模型花费的时间更少。

例如为 AM2910 中的 STACK 模块创建一个快速时序模型：

```
source -echo stack.qtm.pt
report_qtm_model;
save_qtm_model -output STACK -format db
```

其中 stack.qtm.pt 是一个脚本文件，描述了建立快速时序模型所需要的所有 pt_shell 命令，所以在这里直接 source 就可以了。它的具体内容参见附录[1]。

创建好之后，用 save_qtm_model 命令把该模型保存为.db 格式的文件。同其他类型的模型一样，PrimeTime 也创建了两个.db 文件，STACK_lib.db 和 STACK .db。

最后需要说明的是，§4.1 节中的操作并非对所有的设计都是必需的，所有我没有把它加入到本章开头的流程中去。

§4.2 设置查找路径和链接路径

查找路径和链接路径在 PrimeTime 中对应着两个变量：search_path 和 link_path，用 set 命令对它们进行设置。

设置查找路径：set search_path "."

设置链接路径：set link_path "* pt_lib.db STACK_lib.db Y_lib.db"

我是在存放 AMD 2910 的设计文件的目录下运行 PrimeTime 的，所以把 search _path 设置成当前目录“.”。

link_path 中的符号“*”的意思是，当 **PrimeTime** 在链接时，它会使用内存中的设计文件和库文件。**pt_lib.db**、**STACK_lib.db**、**Y_lib.db** 是该设计中用到的库文件。

§4.3 读入设计文件

下面的表格给出了 **PrimeTime** 可以接受的文件类型，以及读入每一种类型的设计时，所使用的不同命令。

PrimeTime 可以接受的文件类型	命令
Synopsys 数据库文件（**.db**）	**read_db**
Verilog 网表文件	**read_verilog**
EDIF 网表文件 （**Electronic Design Interchange Format**）	**read_edif**
VHDL 网表文件	**read_vhdl**

本文中分析的例子使用的是第一种格式，这是大部份 **Synopsys** 工具都支持和共享的一种公用的中间结构，它是描述文本数据的二进制已编译表格式。例子中顶层的设计文件是 **AM2910.db**，所以这样读入该设计：

```
read_db AM2910.db
```

§4.4 链接

链接过程就在库文件中寻找到设计中所需要的元件，并将该设计例化的过程。**PrimeTime** 首先调用 **link_path** 中指定的所有的库文件，然后进行链接，该设计中的五个子模块 **CONTROL**、**REGCNT**、**STACK**、**UPC**、**Y** 会依次被读入进来。可以看到，我们在定义 **link_path** 的时候，一定要把所有的技术库（**pt_lib.db**）和模型库文件（**STACK_lib.db**、**Y_lib.db**）都写在里面，否则会导致链接的失败。

链接使用如下的命令：

link_design AM2910

说明：

1）在上面的例子中，我没有把子模块的设计文件读入到 **Memory**，因为在链接的时候 **PrimeTime** 可以自动读入它们。但是注意这样做的前提是 **PrimeTime** 能够在 **search_path** 中找到所有的子设计文件。

2）如果在 **link_design** 命令中，没有指定需要链接的设计名，**PrimeTime** 会链接当前设计。如果没有当前设计，**PrimeTime** 会读入并链接最近的设计。

3）在默认的情况下，变量 **link_create_black_boxes** 的值是 **true**。如果链接过程中 **PrimeTime** 无法找到某个元件的 **reference**，即无法将其例化，它将为该元件建立一个 **black box**。假如变量 **link_create_black_boxes** 被设置为 **false**，链接将会失败。

链接完成之后我们可以用下面的命令来查看当前读入的设计：

list_design

还可以查看当前已经读入的单元的信息：

report_cell

需要注意的是，**memory** 中只能存在一个已链接的设计，当你链接一个新的设计后，以前的设计将变成未链接的，此时所有的时序信息都将丢失，系统会给一个 **warning**。你可以在链接新的设计之前使用命令

write_script

生成一个脚本（**.pt** 文件），以后可以通过运行这个脚本来重新链接。

§4.5 设置操作条件和线上负载

```
set_operating_conditions -library pt_lib -min BCCOM -max WCCOM
set_wire_load_mode top
set_wire_load_model -library pt_lib -name 05x05 -min
set_wire_load_model -library pt_lib -name 20x20 -max
```

PrimeTime 在产生 **setup timing report** 的时候使用最大的操作条件和线上负载，在产生 **hold timing reports** 的时候则使用最小的操作条件和线上负载。

如果最大和最小的操作条件在两个不同的库中，可以使用命令

set_min_library

来建立两个库之间的联系。可以用 **list_libraries** 命令来查看所有的库，然后对需要详细了解的库，用“**report_lib** 库名”来查看**.**

§4.6 设置基本的时序约束

§4.6.1 对有关时钟的参数进行设置

```
create_clock -period 30 [get_ports CLOCK]
set clock [get_clock CLOCK]
set_clock_uncertainty 0.5 $clock
set_clock_latency -min 3.5 $clock
set_clock_latency -max 5.5 $clock
set_clock_transition -min 0.25 $clock
set_clock_transition -max 0.3 $clock
```

如果设计中具有完全 **back-annotated??(**注释**)**的时钟网络的话，上面的参数如 **uncertainty**、**transition** 等都可以使用下面的命令自动得到：

```
set_propagated_clock clock_object_list
```

§4.6.2 设置时钟一门校验(clock-gating checks)

设定时钟一门的建立和保持时间的数值，以及最小的脉冲宽度。

```
set_clock_gating_check -setup 0.5 -hold 0.1 $clock
set_min_pulse_width 2.0 $clock
```

如果该设计是 **back-annotated** 的，**PrimeTime** 会从 **SDF(standard delay format**，标准延迟格式**)**中取得以上参数。

§4.6.3 查看对该设计所作的设置

使用命令：

```
report_design
```

可以得到该设计的最大、最小的操作条件和线上负载。

```
report_reference
```

可以得到每个模块及其面积的信息。而且更重要的是，它能在各个模块中识别出 **stamp** 模型、以及快速时序模型，其余的模块都是门级的网表。

§4.7 检查所设置的约束以及该设计的结构

在开始静态时序分析之前，注意要执行下面的指令进行检查：

check_timing

该命令会检查该设计的结构和约束，给出可能存在的时序问题，如果存在问题的话则给出 **errors** 或者 **warnings**。可以在 **check_timing** 命令中设定参数，以得到有关 **error** 和 **warning** 的更多信息。具体的参数可以用命令“**check_timing -help**”查看，限于篇幅这里不再赘述。

在本文所分析的设计 **AM2910** 中，**check_timing** 之后产生了一些 **warnings**，这是因为设计中存在的无约束的端口引起的。解决这个问题的方法，将在下一章中讲述。

第五章　静态时序分析

这一章将讲述对 AM2910 进行静态时序分析的过程。

§5.1　设置端口延迟并检验时序

对于所有与时钟相关的端口，都要设置输入、输出的延迟。

```
set_input_delay 0.0 [all_inputs] -clock $clock
set_output_delay 2.0 [get_port INTERRUPT_DRIVER_ENABLE] -clock $clock
set_output_delay 1.25 [get_port MAPPING_ROM_ENABLE] -clock $clock
set_output_delay 0.5 [get_port OVERFLOW] -clock $clock
set_output_delay 1.0 [get_port PIPELINE_ENABLE] -clock $clock
set_output_delay 1.0 [get_port Y_OUTPUT] -clock $clock
```

除此之外，还要对所有的输入端设置一个驱动单元，对所有的输出端设置电容负载。

```
set_driving_cell -lib_cell IV -library pt_lib [all_inputs]
set_capacitance 0.5 [all_outputs]
```

说明：clock 是在第四章中定义的变量（set clock [get_clock CLOCK]），以后出现将不再说明。

完成以上的设置之后，可以再运行一次检查：

```
check_timing
```

在上一章中我们看到，在对设计 AM2910 运行 check_timing 的时候出现了错误信息，产生的原因是设计中存在着无约束的端口。现在再次检查，没有 warnings 出现，该设计已经是完全约束的了。

§5.2　保存以上的设置

使用 write_script 命令将所作的设置保存到一个脚本文件中，这样在下一次运行的时候可以直接通过该文件来完成所有的设置。

该命令可以生成三种格式的文件：

1）Design Compiler 的 dcsh 格式（.dcsh）

2）Design Compiler 的 dctcl 格式（.tcl）

3）PrimeTime 的文件格式：（.pt）

命令的形式为：

```
write_script -format dctc1 -output AM2910.tc1
write_script -format dcsh -output AM2910.dcsh
write_script -format ptsh -output AM2910.pt
```

事实上这种脚本文件，也是 PrimeTime 和 Design Compiler 传递数据的一种主要方法。对于使用 Design compiler 来综合的电路设计，可以把一些重要的设置直接继承到 PrimeTime 中来。

在生成的脚本文件中，包含了以下的信息：

Clocks	names、waveforms、latency、uncertainty
Timing Exceptions	false paths、multicycle paths、path groups、minimun and maximum delays
Delays	input and output delays、timing checks、all delay annotations
Net and Port	capacitance、resistance、fanout
Design Environment	wire load model、operation condition、drive、driving cell、transition
Design Rules	minimum and maximum capacitance、fanout、transition

可以看到，表 5-1 中包括了我们前面所作的所有的设置。

调用脚本的方法是：

```
source -echo 脚本文件名
```

§5.3 基本分析

运行 report_constraint 命令，得到的 constraint report 中包括了对整个设计的时序信息的总结。通过它，可以检查设计中存在的？？timing violations 和 constrains violations。

具体来说 **constraint report** 中包含了以下的内容：

- 最大和最小延迟（**min and max delay**）
- 最小时钟脉冲宽度（**min clock pulse width**）
- 最小周期（**min period**）
- **??recovery and removal on registers**
- 时钟一门的建立和保持（**clock-gating setup and hold**）
- 最大和最小电容（**min and max capacitance**）
- 最大和最小转换周期（**min and max transition**）
- 最大和最小扇出（**min and max fanout**）

如果在命令中加入**-verbose** 参数，将在 **report** 中得到更详细的细节。如果加入**-all_violators** 参数，在 **report** 中会列出对于每一项约束，设计中违反最严重的端点。

§5.4 生成 path timing report

使用 **report_timing** 命令，生成基于路径的 **timing report**。

在没有任何命令参数时，在 **report** 中列出的是对于每个 **path group**，该设计中最长的最大路径。如果需要的是该设计中最短的最小路径的话，可以在命令中加上**-delay min** 参数。

report_timing 命令是一个很灵活的命令，可以用**-help** 来查看其它的参数。

§5.5 设置时序中的例外??(timing exceptions)

Timing exceptions 包括了：错误路径**(false paths)**、多循环路径**(multic -ycle paths)**、用户定义的最大最小延迟约束以及无效的时序 **arcs??**。必须正确地定义 **timing exceptions**，否则它们不会被 **PrimeTime** 接受。例如错误路径和多循环路径必须指定一个完整的，有效的路径，包括正确的起点和终点。其中起点应该是主要的输入端口、时钟、管脚或者单元，而终点应该是主要的输出口、时钟、管脚或者单元。

需要注意：

1）假如没有进行 update，PrimeTime 不会检验 timing exceptions 的正确性。可以使用 report_exceptions 命令，来确定它们是否是正确的。

2）在该命令中加上-ignored 参数，看看是否有 exceptions 因为不正确而被 PrimeTime 忽略了。更正被忽略的 exceptions，否则它们将不起作用。

下面的是为 AM2910 设置 timing exceptions 的命令，我们在两个时钟间建立一个多循环路径。

```
set_false_path -from U3/OUTPUT_reg[*]/CP -to U2/OUTPUT_reg[*]/D
set_multicycle_path -setup 2 -from INSTRUCTION[*] -to U2/OUTPUT_reg[*]
set_multicycle_path -hold 1 -from INSTRUCTION[*] -to U2/OUTPUT_reg[*]
update_timing
report_exceptions -ignored
```

没有被忽略的 exceptions，说明对 timing exceptions 的定义是正确的。。

§5.6 再次进行分析

在定义好 timing exceptions 之后，再次进行分析，生成新的 constraint report 和 timing report。

```
report_constraint -all_violators
report_timing
```

可以看到设置了 exceptions 之后，viloators 的个数，以及 slack 的数值都减少了。

从第四章到这里，讲述了用 PrimeTime 对 AM2910 进行静态时序分析的过程，其中的 pt_shell 命令保存在了一个脚本文件中，详见附录[2]。

第六章 Formality 简介

在现在的数字集成电路设计流程中，有很多步骤都需要进行逆向的验证。随着数字集成电路的规模、复杂度，以及在验证过程中需要的模拟矢量的不断增加，用传统的模拟器进行逆向验证越来越成为了整个设计过程中的瓶颈之所在。

这主要是因为：为了确保设计达到所需要的各方面的要求，需要数量众多的模拟矢量。而数量众多的矢量、日益增大的设计尺寸，都增大了验证过程中需要交换和处理的数据量。此外，由于电路尺寸和复杂度的增加，对于每一个激励，逻辑模拟工具都要进行更多的处理，也是导致这个瓶颈的因素之一。

在这样一种背景下，形式验证（**Formal Verification**）技术显示出了较多的优点。这一章将对 **Synopsys** 的形式验证工具 **Formality** 作一个简单的介绍。

§6.1 Formality 的基本特点

所谓形式验证，就是通过比较两个设计在逻辑功能是否等同的方法来验证电路的功能。这种方法的优点在于它不仅提高了验证的速度，可以在相当大的程度上缩短数字设计的周期，而且更重要的是，它摆脱了工艺的约束和仿真 **test bench** 的不完全性，更加全面地检查了电路的功能。

Formality 是形式验证的工具，你可以用它来比较一个修改后的设计和它原来的版本，或者一个 **RTL** 级的设计和它的门级网表在功能上是否一致。

Formality 有下面一些特点：

- 跟事件驱动的模拟器相比，能要快验证出两个设计在功能上是否等同；
- 不依赖于矢量，因此能提供更完全的验证；
- 可以实现 **RTL-to-RTL**、**RTL-to-gate**、**gate-to-gate** 之间的验证；
- 有定位功能，可以帮助你找出两个设计之间功能不等同的原因；
- 可以使用的文件格式有 **VHDL**、**Verilog**、**Synopsys** 的 **.db** 格式，以及 **EDIF** 网表等；
- 可以实现自动的分层验证；

- 使用 **Design Compiler** 的技术库；
- 同 **PrimeTime** 一样提供两种界面：图形用户界面 **GUI** 和命令行界面 **fm_shell**；

§6.2 Formality 在数字设计过程中的应用

在现在的 **EDA** 设计方法中，**Formality** 可以很好地取代传统的模拟工具去完成逆向验证。由于 **Formality** 在验证时不需要任何输入矢量，所以会带来两个显著的优点：更短的验证时间、更完全的验证结果。它与静态时序分析工具结合在一起，可以在相当大的程度上改善数字电路的设计过程。

任何时候对一个电路设计进行了改动之后，都可以使用 **Formaliyt** 来验证这种改动是否影响或者改变了该设计的逻辑功能。如果证实了改动后的设计和源设计是等价的之后，就可以把修改后的设计作为下一次验证时的“源设计”。由于结构相似的设计所需要的比较时间较短，这样也就节省了花费在验证上的时间。

下图是一个典型的 **ASIC** 的验证过程，从中我们可以清楚地看到 **Formality** 在数字设计过程中的作用。

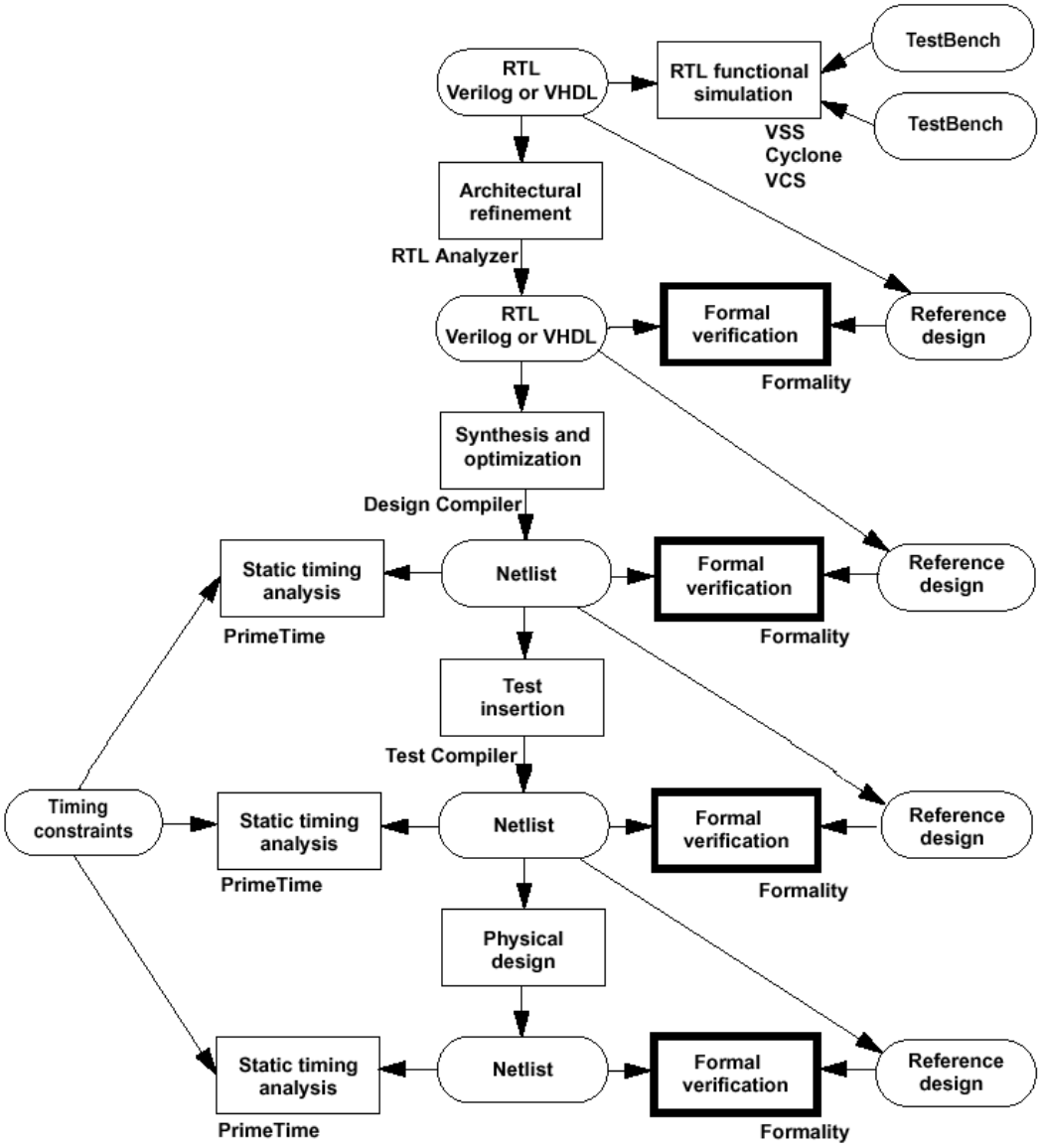


Figure6-1 ASIC 的验证过程

§6.3 Formality 的功能

我们可以把 **Formality** 的功能大致划分为四个方面，如图 **6-2** 所示。

1）设计管理

设计管理指的是你可以对需要验证的设计进行管理和控制，例如读入设计、设置参数、保存和再次调用设置等等。

2）验证

Formality 的主要功能。

3）生成报告

在进行验证的过程中，**Formality** 会生成好几种类型的报告，从中你可以得到关于验证、诊断的结果等等有用的信息。

4）诊断

当验证的结果是两个设计并不等同时，你可以使用诊断功能去寻找不等同的原因。关于诊断等功能的更详细的细节将在下面的章节中讲述。

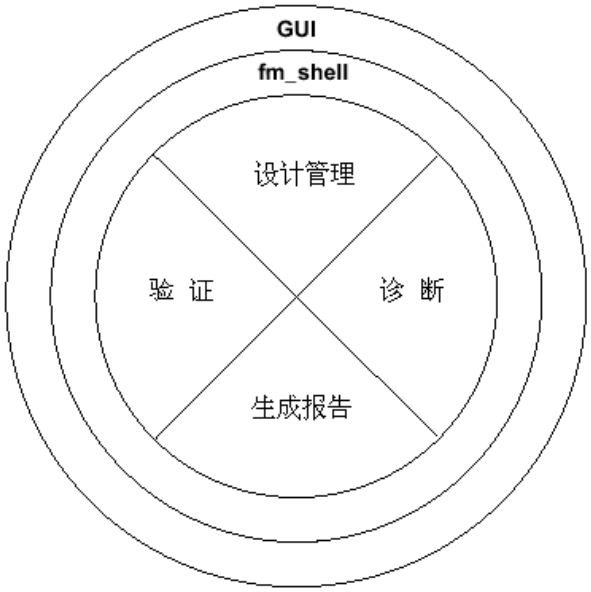
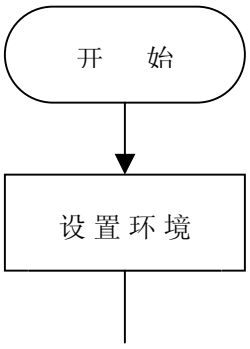
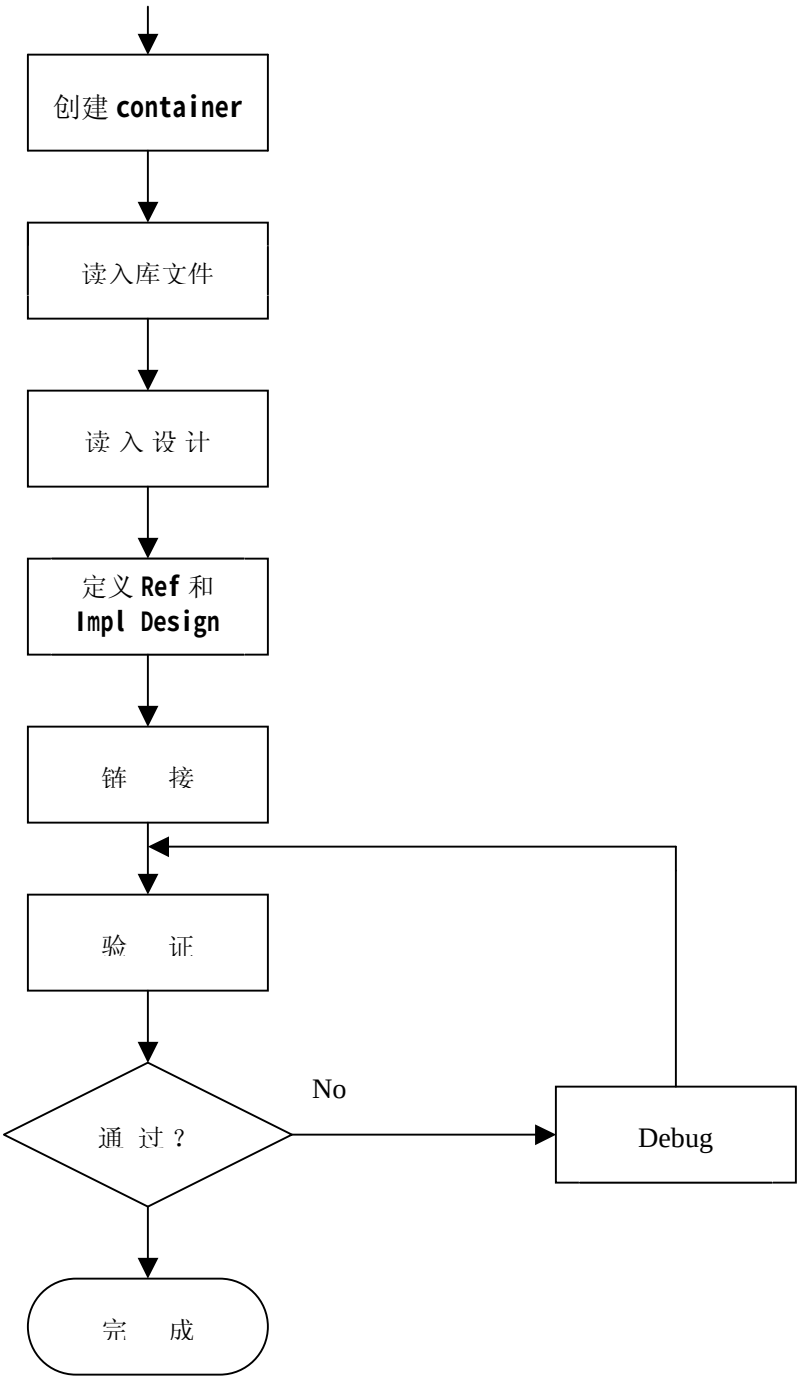


Figure6-2 Formality 的主要功能

§6.4 验证流程

下图给出了使用 **Formality** 进行形式验证的一般流程：





一般的情况下从开始一直到“运行分析”这一步骤，都是使用 **fm_shell** 的命令行模式来完成；其后的步骤则通常使用 **GUI** 完成。

第七章　形式验证

这一章将讲述使用 **Formality** 进行形式验证的过程。

§7.1　fm_shell 命令

与 **PrimeTime** 一样，**Formality** 的命令也是基于 **Tcl** 的，所以关于命令使用的基本知识这里就不再重复了。在遇到不了解的命令时同样可以使用 **help** 和 **man** 命令或者**-help** 参数来查看具体信息。

需要说明的是，很多 **UNIX** 的命令在 **fm_shell** 中可以直接使用，例如 **ls** 等；但是另外一些命令，例如 **cat**，需要用下面的方式来使用：

```
sh cat
```

§7.2　一些基本概念

§7.2.1　Reference Design 和 Implementation Design

从前面的介绍可以知道，在形式验证的过程中涉及到两个设计：一个是标准的、其逻辑功能符合要求的设计，在 **Synopsys** 的术语中称之为 **Reference Desi-gn**；另一个是修改后的、其逻辑功能尚待验证的设计，称之为 **Implementation Design**。在本文所使用的例子中，**Reference Design** 和 **Implementation Design** 分别放在我的工作目录的 **dbs** 和 **netlists** 子目录中，后者是前者的一个修改后的版本，其中包含了测试电路。它们使用的库文件放在 **lib** 目录中。所以，首先把 **lib**、**dbs**、**netlists** 目录加入到查询路径中：

```
set search_path ". ./lib ./dbs ./netlists"
```

§7.2.2　container

我们可以把 **container** 理解为 **Formality** 用来读入设计的一个空间，或者说一个“集装箱”。一般情况下要建立两个 **container** 来分别保存 **Reference Design** 和 **Implementation Design**。对于一个 **container**，可以进行命名、删除、关闭等操作。一个 **container** 中包含有一个设计，以及该设计所需要的所有的技术库和设计库，如下图所示：

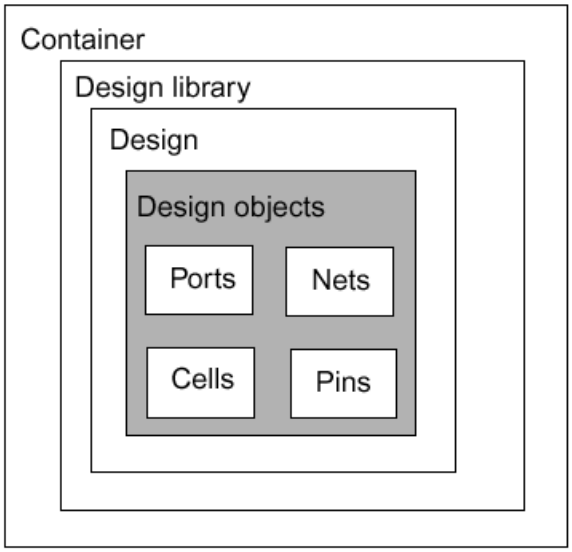


Figure7-1 container 示意图

§7.3 读入共享技术库

在开始验证流程之前，首先要读入所有的会被用到的共享技术库。留意一下运行 **Formality** 时显示的信息，可以看到它已经自动读入了一个通用的共享技术库：**gtech.db**。在本文所使用的例子中，还需要读入位于**./lib** 目录下的 **cba_core.db** 这个库：

```
read_db cba_core.db
```

使用下面的命令可以查看已经读入的库的情况：

```
report_libraries -short
report_libraries
```

前一个命令将给出每个库包含的单元数，以及它们是否共享的情况。后者将列出库中包含的所有单元。

说明：实际上，在键入一个命令及其参数时，只要你输入的字符能把该命令同其他命令区分开来就可以了，不必输入所有的字符，例如上面的第一个命令就可以缩略成下面的形式：

```
report_li -s
```

§7.4 设置 Reference Design

1）建立一个新的 container

在读入一个设计之前，首先要为它建立一个 container：

```
create_container ref
```

这里“ref”是 container 的名字。

2）读入门级网表

Formality 可以接受以下几种格式的设计：

- Synopsys 的.db 格式
- Verilog
- VHDl
- EDIF

例子中，Reference 设计（synth.db 文件中的 mR4000）是.db 文件，Implemen-tation 设计（clk_insert1.v）是 verilog 网表文件。把 synth.db 读入到当前的 container：

```
read_db synth.db
```

假如是使用 GUI 界面的话，Formality 会弹出一个 container 窗口，可以清楚地看到其中包含了两个共享设计库 cba_core 和 gtech，以及刚刚读入的设计，它被自动命名为 WORK。我们将要进行验证的 Reference Design 是其中的一个名字为 mR4000 的子设计。

3）确认该设计为 Reference Design

```
set_reference_design ref:/WORK/mR4000
```

说明：在命令中要键入设计全名(full designID)。设计全名的一般形式是：

container 名：/库名/设计名

在完成这一步后，Formality 生成了一个 ref 变量，指代 Reference Design，即上面的设计全名。

4）链接 Reference Design

事实上，Formality 在进行形式验证时会自动进行链接。但是为了发现可能出现的错误，例如设计名或者输入输出的不匹配等等，可以人工进行一下链接。

```
link $ref
```

§7.5 设置 Implementation Design

这个过程跟 Reference Design 的设置过程是类似的。

1）建立一个名为 impl 的 container，读入 clk_insert1.v 文件

```
read_verilog -c impl -netlist clk_insert1.v
```

说明：-netlist 参数说明了需要读入的文件是一个网表文件，其中不含有 RTL 的内容。这样可以减少 Formality 读入文件所花的时间。

2）确认 Implementation Design

```
set_implementation_design impl:/WORK/mR4000
```

3）链接该设计

```
link $impl
```

4）把该设计设置为当前设计，然后把其中的 test_se 端口设置为 0。：

```
current_design §impl
set_constant test_se 0
```

设置当前设计的好处是在命令中不需要键入设计全名。前面提到 Implementa -tion Design 比 Reference Design 多加入了测试电路，而 test_se 是其中顶层的输入信号，把它的逻辑状态设为 0，就消除了测试电路对于验证的影响。

§7.6 保存及恢复所作的设置

可以说到这一个步骤，已经完成了形式验证前必需的准备工作。一般来说，下面的步骤，特别是 Debug，在 GUI 界面下完成会比较方便，当然使用 fm_shell 也是完全可以的。所以这里先将上面所作的设置保存到.fss 文件中：

```
save_session -replace -full fm_shell_session
exit
```

然后运行 GUI，并恢复设置：

```
restore_session fm_shell_session.fss
```

§7.7 验证

运行 **verify** 命令，**Formality** 将根据所作的设置，对 **ref** 和 **impl** 中的两个设计进行验证。如果验证通过的话，就说明在两个设计在逻辑功能上是等同的。

实际验证的结果是：

```
verification failed
3 Failing compare points
```

即有三个不匹配的点导致了验证的失败。下一章将利用 **Formality** 的诊断功能，找出这些点并进行 **debug**。

第八章　对验证失败的设计进行 Debug

上一章对一个具体的数字设计进行了形式验证，验证的结果是发现了在 **Refe-rence Desing** 和 **Implementation Design** 中存在着不匹配的点，从而导致了它们功能的不等同。这一章将分别利用 **fm_shell** 的诊断程序和 **GUI** 的逻辑锥图进行 **De-bug**，其中后者更加直观，是推荐使用的方法。

§8.1　查看不匹配点的详细信息

在验证失败之后，我们需要找出导致两个设计功能差异的原因并将其修正。首先可以查看不匹配点的详细信息：

report_failing_points

可以看到在两个设计中，不匹配的对象有三个：**Instruction_reg[10]**、**Instruction_reg[16]**和 **state_reg[9]**。

§8.2　诊断程序

在第六章的图 **6-2** 中可以看到，**Formality** 的功能中包括一个叫做“诊断”的部份，所谓“诊断”就是说对于 **Implementation** 和 **Reference Design** 的不匹配，**Formality** 可以识别出其中可能的原因。在“诊断”过程中，**Formality** 分析电路的行为，并计算当改变设计中的某一个 **net** 时，原来不匹配的部份获得改善的可能性。

运行诊断程序，并查看其产生的 **report**：

diagnose

report_error_candidates

结果如下：

```
Error candidates:

  %      type    Name
-----    ----    ----
 89.5    net     Clk
 66.4    net     n1023
 52.6    net     cntrl/n1236
 52.6    net     cntrl/state[9]
 52.6    net     n990
 31.6    net     Instruction[15]
 31.6    net     n1012
 31.6    net     n1030
 31.6    net     test_se
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
Max error coverage found: 89.5% (1 instances and 0 library
 cells
```

在生成的报告文件中，**Formality** 列出了当某个 **net** 或者 **cell** 的逻辑功能被改变时，不匹配部份通过的百分比。例如，如果一个 **net** 的百分比是 **100%**，那么只要对它进行修正，所有的不匹配都会消失。

在本文的例子中，可以通过改变 **CLK** 的功能来修正大部份的不匹配。但是在一般的情况下，我们不希望改动时钟信号，因为这样很可能会带来更多的问题。而且可以看到，没有一个网点的百分比是 **100%**，所以问题的症结肯定不是只存在于一个网点上。所以这个时候最好分别对每一个不匹配的点分别进行诊断和 **debug**，以便更精确地找到问题。

首先我们对第一个不匹配点，即 **instruction_reg[10]**进行诊断：

```
diagnose $impl/Instruction_reg[10]
report_error_candidates
```

其结果是：

```
Error candidates:
  %      type    Name
-----    ----    ----
100.0    net     Clk
100.0    net     n1023
100.0    net     n42
```

此时修改 **Clk**，**n1023**，**n42** 三个 **net** 中的每一个都可以解决 **Instrution_reg [10]**处的不匹配问题。当然一般对 **Clk** 不作改动，可以在另外两者中选择一个。

§8.3 逻辑锥

§8.3.1 逻辑锥的概念

图 **8—1** 可以比较形象地说明“逻辑锥（**Logic Cone**）”这个概念，图中最右

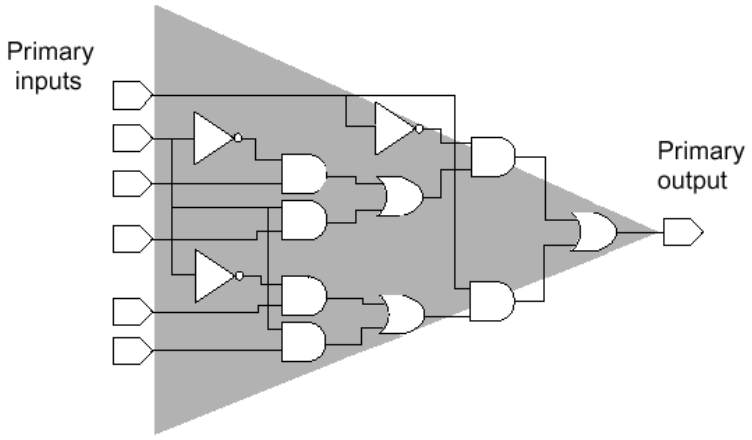


Figure8-1 逻辑锥

边是一个输出端口，在形式验证的过程中它将被作为一个比较点，与另一个设计中的相应的对象进行比较，比较的过程实际上就是考察它们的逻辑锥是否等价。所以对于不匹配的点，我们也应该通过对它们逻辑锥的分析，来找出具体的原因。

§8.3.2 查看不匹配点的逻辑锥

8.1 节和 **8.2** 节是利用 **fm_shell** 对不匹配的原因进行分析，利用 **GUI** 来查看两个设计中不匹配点的逻辑锥，可以更直观地找出其中的原因。

在 **GUI** 界面的 **Report** 菜单中选中 **Failing Points** 选项，在弹出 **Report** 窗口的 **Edit** 菜单中选中 **Show Size**，显示每个不匹配点的大小，这实际上也就是 **fm_**

shell 中的 **report_failing_points** 命令。一般的习惯是从最小的点开始分析，右击 **impl:/WORK/mR4000/Instruction_reg[10]**，选择 **view logic cone**。如下图所示，在上下两部份窗口中分别显示了 **Reference Design** 和 **Implementation Desi -gn** 的逻辑锥。

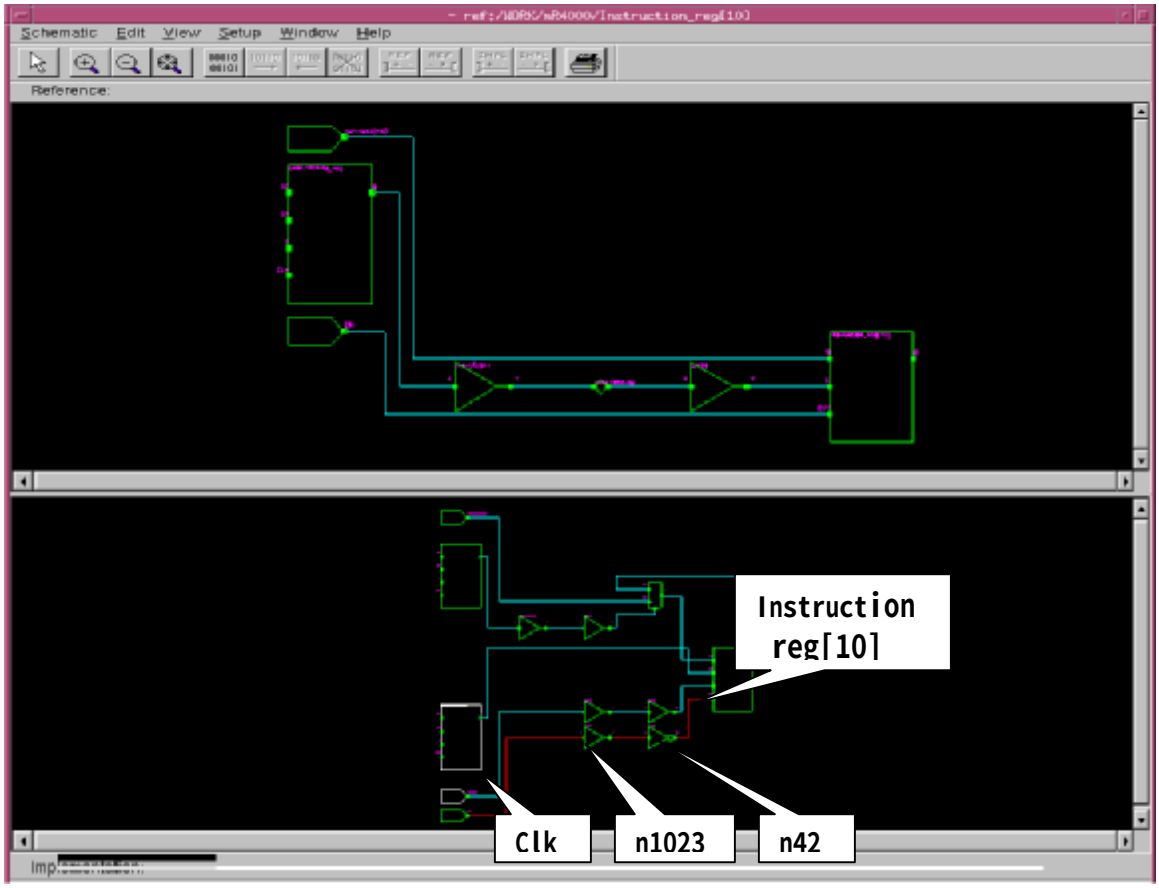


Figure7-3 不匹配点的逻辑锥

在 **8.2** 节中诊断之后得到的报告中，给出了与不匹配相关的百分比，这个百分比是我们解决不匹配的问题的很重要的依据。在这里的逻辑锥图中，线条的颜色也表示了这种百分比，其对应关系是：

亮蓝色	**0**%
灰白色	**<=24**%
绿色	**25**%－**49**%

紫色	50%－89%
红色	90%－100%

可以看到，连接到 **Instruction_reg[10]**模块的 **Clk** 输入端的线条是红色的，其中包括了 **Clk**、**n1023**、**n42** 三个 **net**，其他的 **net** 则是亮蓝色的。结合上面的表格可以知道，这跟我们在 **8.2** 节中得到诊断结论也是一致的。

说明：图中白色的元件表示在待验证设计中有，而在参考设计没有的元件。

§8.3.3 使用逻辑锥来 Debug

放大图形，直到看清楚 **instruction_reg[10]**的管脚。选中与 **CLK** 脚相连的 **net**，右击并选中 **Isolate Subcone**，此时只显示与 **CLK** 脚相关的逻辑锥。这样我们可以对比在 **Reference Design** 和 **Implementation Design** 中，驱动 **Clk** 脚的逻辑。如下图所示，在 **Implementation** 中多出了一个 **buffer** 和一个反相器，这就是导致这一点没有通过验证的原因。

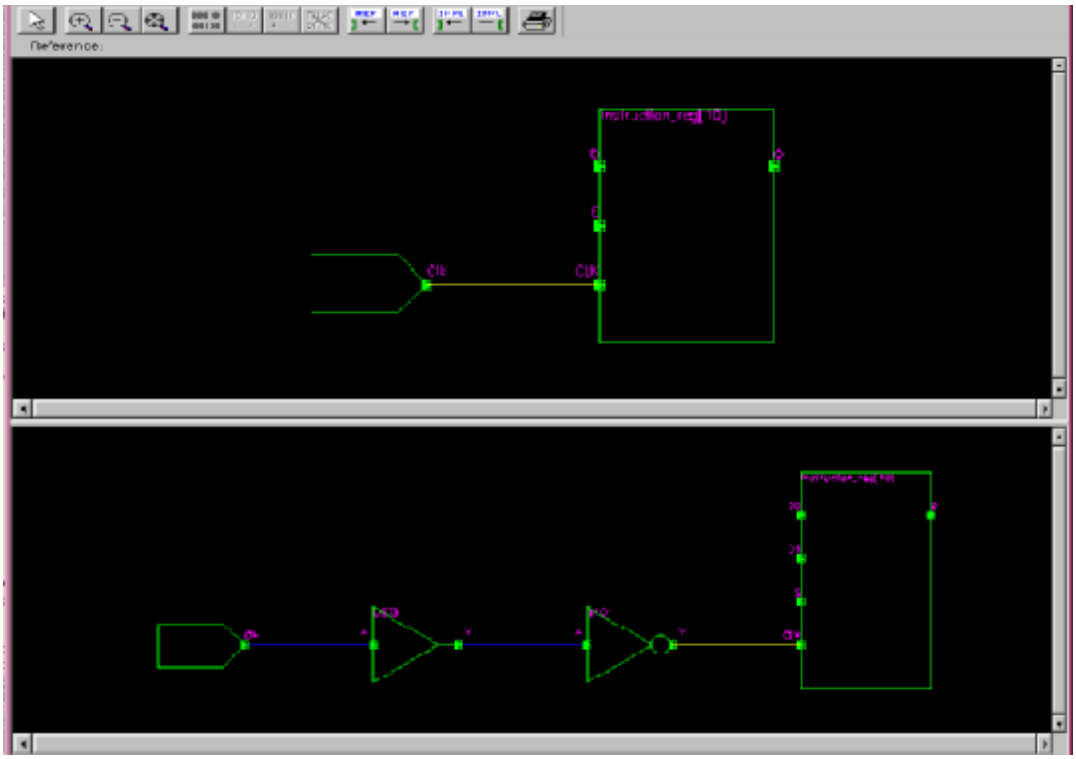


Figure7-4 使用逻辑锥 **Debug**

§8.3.4 通过逻辑值来分析

在验证失败的时候，**Formality** 会生成一组输入矢量来描述具体的逻辑情况。可以在逻辑锥图查看每一个输入矢量的逻辑值，方法是在逻辑锥窗口中的 **View menu** 菜单中选中 **Apply First Pattern**，就可以看到元件的管脚和连线旁边都标注了逻辑值。比较两个设计中逻辑值的差异，从中寻找有助于 **debug** 的信息。限于篇幅，这部份的内容就不深入讲述了。