

研究管理部文档中心	文档编号	产品版本	密级
		2.1	内部公开
	产品名称：		共45页

Synplify工具使用指南

(征求意见稿，仅供内部使用)

文档作者：	_____	日期：	2001/ 03 /20
项目经理：	_____	日期：	____ / ____ / ____
研 究 部：	_____	日期：	____ / ____ / ____
总 体 组：	_____	日期：	____ / ____ / ____
文档管理员：	_____	日期：	____ / ____ / ____

修订记录

日期	修订版本	描述	作者
2000/08/20	1.00	初稿完成	
2000/9/20	1.10	第一次修订	
2000/3/20	2.00	第二次修订	
2000/4/2	2.10	第三次修订	

目 录

1前言	5
2基本概念	6
2.1综合	6
2.2工程文件	6
2.3Tcl 脚本	6
2.4约束文件	6
2.5宏库	6
2.6属性包	6
3基本工作流程	6
3.1PC版基本工作流程	7
3.2UNIX环境下窗口界面工作流程	7
3.2.1UNIX环境设置	7
3.2.2UNIX版用户界面简介	
1.工具条	7
2.用户界面的按钮	7
3.2.3 SCOPE窗口	8
1.Insert Wizard	8
2.Clock	9
3.Clock to Clock	10
4.Input/Output	10
5.Registers	11
6.Multicycle Paths	11
7.False Path	11
8.Attribute	11
9.Other	12
3.2.4生成的报告和信息	12
1.Log 文件	12
2.时间报告 (Timing Report)	12
3.资源使用报告	12
4.Net Buffering报告	12
3.3批处理工作模式流程	13
3.3.1运行工程文件	13
3.3.2运行一个Tcl文件	13
4使用HDL Analyst分析和调试设计	14
4.1HDL Analyst 简介	14
4.2HDL Analyst 的应用	14
4.2.1POP_UP信息	14
4.2.2状态条显示	14
4.2.3链接式选中目标	14
4.2.4查看延迟信息	14
5使用Symbolic FSM compiler	15
6怎样用Tcl语言执行批处理任务	16

6.1创建Tcl script 文件	16
6.2常用Tcl 命令说明	16
6.2.1工程命令	16
6.2.2添加文件的命令	17
6.2.3控制命令	17
6.2.4打开文件的命令	18
6.3Tcl 格式的script文件示例	18
6.3.1运行一个script 文件针对多个目标器件进行综合	18
6.3.2运行多个频率要求, 并存储为不同的log 文件	19
6.3.3设置控制选项及约束示例	19
6.3.4自底向上的综合示例	21
6.4运行script文件	22
7时间约束	22
7.1书写约束文件的一些规则	22
7.2Verilog对象表示语法	22
7.3HDL源代码中的约束	23
7.3.1通用时间约束	23
7.3.2黑盒时间约束	26
7.3.4特定厂家的时间约束文件	26
8综合属性 (Attributes) 和指示 (Directives)	26
8.1简介	26
8.2厂家提供属性	27
8.2.1Altera	27
8.2.1Xilinx	30
8.2.1综合指示 (Directives)	35
9实现对速度的优化	38
9.1一般性考虑	38
9.2怎样处理关键路径上不满足速度要求的延迟	38
9.3关于综合约束的建议	38
9.3.1时钟	38
9.3.2扇出限制	39

Synplify 快速入门

关键词: Synplify synthesis 综合 Script 脚本 Tcl FPGA Verilog

摘要: 本文的目的是为Synplify的使用提供帮助, 包括三个主要内容, 第一部分快速入门, 介绍基本的工作流程。第二部分Tcl使用指南, 包括运用批处理工作文件提高工作效率, 和怎样用时间约束文件使综合结果更加成功。第三部分是一些通用的以及专门针对Altera和Xilinx器件的综合策略。

缩略语清单:

SCOPE: Synthesis Constrains Optimization Environmemt

Tcl: Tool command language

FPGA: Field Programmable Gate Array

RTL: Register Transfer Level

参考资料清单				
名称	作者	编号	发布日期	查阅地点或渠道
Synplify Reference Manual	Synplicity co.		Oct. 2000	Online help
Synplify User guide and tutorial	Synplicity co.		Oct. 2000	Online help

1 前言

Synplify 和 Synplify Pro 是 Synplicity 公司提供的专门针对FPGA和CPLD实现的逻辑综合工具, 它支持VHDL93 (IEEE1076), 包括 std_logic_1164, Numeric_std, std_logic_Unsigned, std_logic_Signed, std_logic_Arith; 和Verilog95 (IEEE1364) 的可综合子集。

该软件提供的Symbolic FSM Compiler 是专门支持有效状态机优化的内嵌工具; SCOPE是管理 (包括输入和查看) 设计约束与属性, 提供活页式分类, 非常友好的表格界面; 用于文本输入的HDL语法敏感编辑窗口不仅提供了对综合错误的高亮显示, 结合图形化的分析和cross_probe工具HDL Analyst, 可以把源代码与综合的结果有机地链接起来, 帮助设计者迅速定位关键路径, 解决问题; 其提供的命令行界面, 可以通过使用Tcl脚本极大的提高工作效率。

Synplify Pro还增加提供了FSM Explorer, 可以在尝试不同的状态机优化方案后选定最佳结果; 以及FSM viewer, 用于查看状态机的详细迁移状况。

此外, 为了获得最佳的综合效果, Synplify还针对具体的厂家器件提供了较为丰富的综合属性 (Attributes) 和综合说明 (Directives)。

Synplify支持PC (WIN98/WIN2000/WIN NT 4.0)、Sun (Sun OS 5.6 and 5.7/Solaris 2.6 and 2.7)、HP-UX 10.20, 后文内容中3.1节针对PC版, 其余章节所述内容, 因为PC版本与工作站版本并无太大区别, 因此均以工作站版本为例, 如使用PC版本则可参照工作站版本相应部分。

本文针对Verilog HDL，以及特别增加针对Altera和Xilinx器件的内容，有关VHDL和其他厂家器件的信息请参阅Synplify Reference Manual。

目前，部门使用的是Synplify v5.3.1。

2 基本概念

2.1 综合

综合（Synthesis），简单地说就是将HDL代码转化为门级网表的过程。Synplify 对电路的综合包括三个步骤，表示如下：

- 1、HDL compilation：把HDL的描述编译成已知的结构元素。
- 2、Optimization：运用一些算法进行面积优化和性能优化，使设计在满足给定性能约束的前提下，面积尽可能的小。这里，Synplify进行的是基本的优化，与具体的目标器件技术无关。
- 3、Technology mapping：将设计映射到指定厂家的特定器件上，针对目标器件结构优化，生成作为布局布线工具输入的网表。

2.2 工程文件

工程文件（*.prj）以tcl 的格式保存以下信息：设计文件、约束文件、综合选项的设置情况等。

2.3 Tcl 脚本

Tcl（Tool Command Language）是一种非常流行的工业标准批处理描述语言，常用作软件应用的控制。

应用Synplify 的Tcl script 文件，设计者可以用批处理命令的形式执行一个综合，也可以一次执行同一设计多个综合，尝试不同的器件，不同的时延目标，不同的约束条件。

Synplify 的script 文件以（*.tcl）保存。

2.4 约束文件

约束文件采用Tcl，以（*.sdc）保存。用来提供设计者定义的时间约束，综合属性，供应商定义的属性等。

约束文件既可以通过SCOPE创建编辑，也可以使用正文编辑器创建编辑。可被添加到在工程窗口的代码菜单中，也可以被Tcl script 文件调用。

2.5 宏库

Synplify 在它内建的宏库中提供了由供应商给出的宏模块。比如一些门电路，计数器，寄存器，I/O模块等。你可以把这些宏模块直接例化到你的设计中去。

2.6 属性包

Synplify为VHDL提供了一个属性包，在Synplify_install_dir/lib/vhd/synattr.vhd。内容有时间约束，如对黑匣子的时间约束，供应商提供的一些属性，还有一些综合属性以帮助你实现你的综合目的。使用时只需在VHDL源文件的开头加入以下属性包调用语句。

```
library synplify;  
use synplify.attributes.all;
```

3 基本工作流程

3.1 PC版基本工作流程

3.1.1 Synplify的用户界面

Synplify是标准的windows应用程序，所有功能均可以通过菜单选择来实现。下面按照图中数字所标示的次序对其界面作简要介绍。图中1表示Synplify的主要工作窗口，在这个窗口中可以详细显示设计者所创建的工程的详细信息。包括工程包括的源文件，综合后的各种结果文件。同时如果综合完成后，每个源文件有多少错误或者警告都会在这个窗口显示出来。图中2表示TCL窗口，在这个窗口中设计者可以通过TCL命令而不是菜单来完成相应的功能。3是观察窗口，在这里可以观察设计被综合后的一些特性，比如最高工作频率等。4是状态窗口，它表示现在Synplify所处的状态，比如下图表示Synplify处于闲置状态。在综合过程中会显示“编译”状态、“映射”状态等等。5所示的一些复选框可以对将要综合的设计的一些特性进行设置，Synplify可以根据这些设置对设计进行相应的优化工作。6是运行按钮，当一个工程加入之后，按这个“RUN”按钮，Synplify就会对工程进行综合。7所示是Synplify的工具栏。

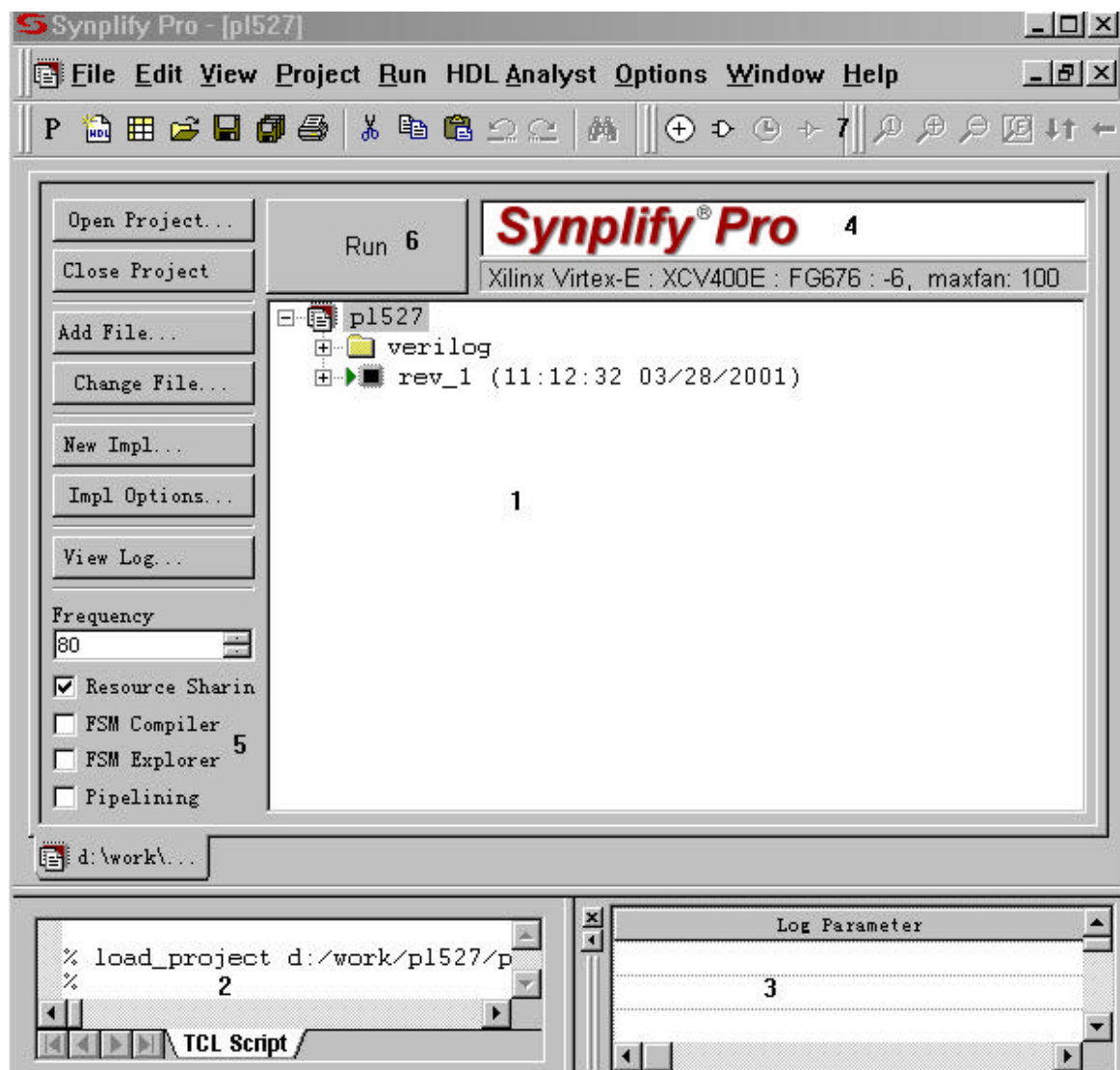


图1 Synplify工作主窗口

3.1.2 启动Synplify

在windows环境下，点击“开始”。依次选择“程序——Synplicity——Synplify”启动Synplify。在工程窗口中包含了以下内容：源文件信息、结果文件信息、目标器件信息。

3.1.3 建立工程

缺省情况下，当Synplify启动时将自动建立一个新工程。这时可以选择将工程以新名字保存。如果结束了一个工程的操作，想新建一个工程，则可以选择“FILE—>NEW”，然后选择“工程文件”就可以建立一个新的工程。这项操作也可以通过工具条来进行：单击工具条的“P”图标，则弹出对话框，选择“工程文件”即可。

3.1.4 添加源文件

新建工程之后，需要将源文件添加进来。点击“ADD FILE”按钮添加源文件和约束文件。

3.1.5 选择顶层设计

Synplify把最后编译的module/entity and the architecture作为顶层设计。故把你所要的顶层设计文件用左键拖拉到源文件菜单的末尾处或者点击“Impl Options”按钮，在“verilog”属性页中设置顶层模块的名称。

3.1.6 设置工程属性

点击“Impl option”按钮，出现属性页对话框，打开“Device”属性页。分别设置器件厂家、器件型号、速度级别和封装信息。

根据设计的速度和面积要求，可以设置最大扇出系数，缺省是100。根据该工程所属模块是否和片外有信号联系，选中或者不选中“Disable I/O insert”。如果选中则告诉synplify不要为输入输出信号加buf。缺省不选中。



图2 设置器件属性属性页

点击“options/Constraints”属性页，作进一步设置。选中“Symbolic FSM Compiler”，即告诉synplify在综合过程中启动有限状态机编译器，对设计中的状态机进行优化。选中“Resource Sharing”选项，则启动资源共享。一般说来，设置了这个选项之后，设计的最高工作频率会低于不选中的情况，但是资源则比不选中要节约好多。在设计能够满足时钟频率要求的情况下，一般选中，以节省资源。选中“Use FSM Explorer Data”选项即可以用synplify内置的状态机浏览器观察状态机的各种属性。选中“Pipelining”即启动“流水”，在高速时钟设计中，如果其他措施都不能达到目标频率，则最好选中此项。

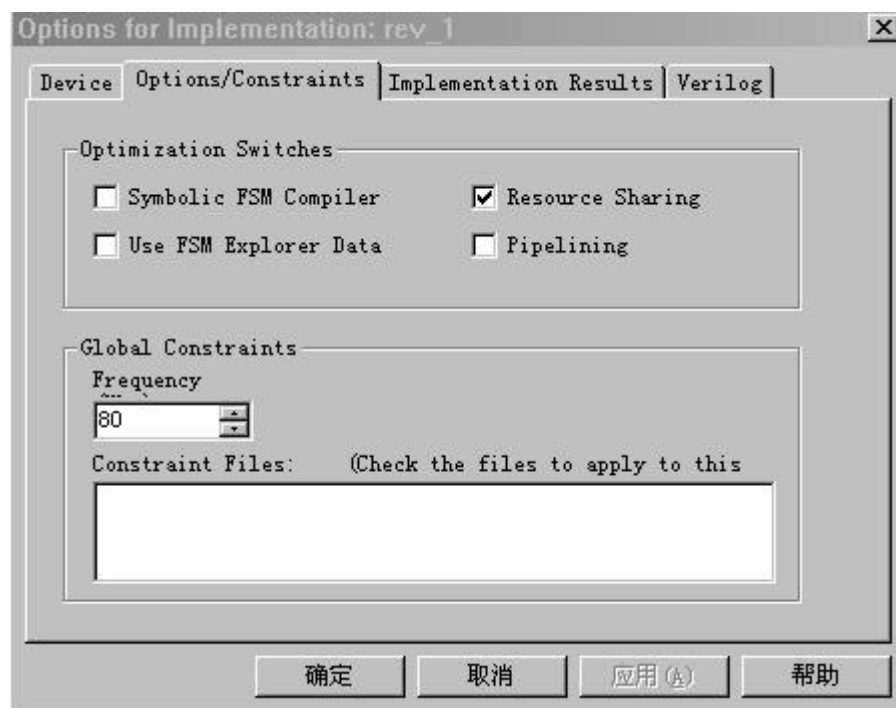


图3 设置选项和约束属性页

点击“Implementation Results”属性页，设置综合结果放置的目录，综合结果的文件名称。同时一定要将“Write Vendor Constraint”选项选中。

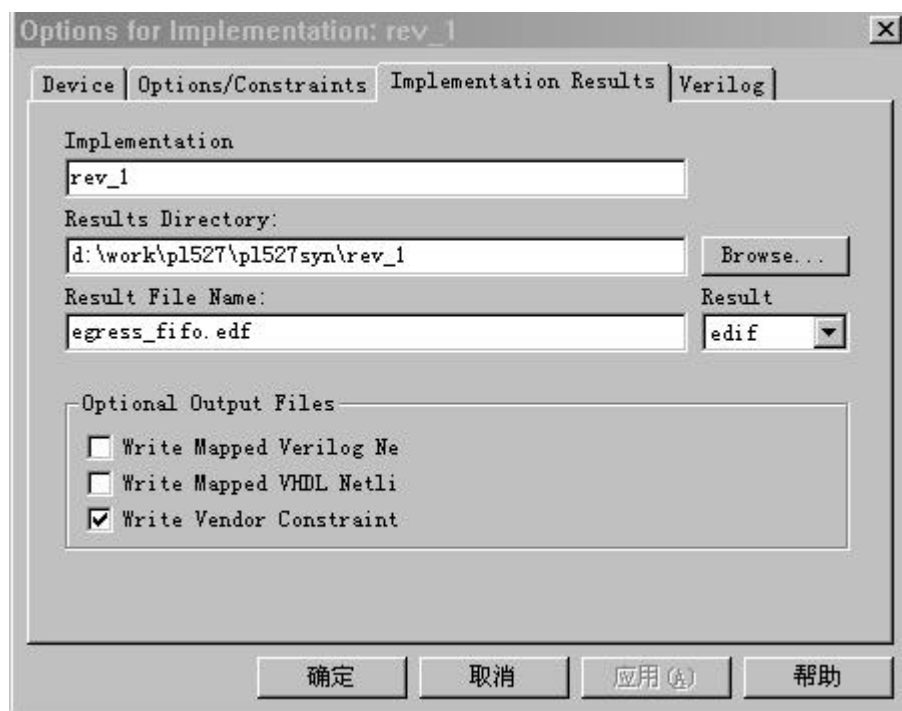



图4 设置综合结果属性页

做完所有设置之后，回到图4的主窗口，点击6处的“RUN”按钮，开始综合即可。

3.2 UNIX环境下的窗口界面工作流程

- 1、在UNIX环境下，键入synplify。系统弹出工程窗口。
- 2、创建新工程和读入已经存在的工程。点击工具条上图标P，选择“New Project ”或“Existing Project”。
- 3、添加源文件，在Source Files区域添加源文件。注意在某添加的源文件中使用include引用的文件不要加入。如果有约束文件也在这里添加。
- 4、选择顶层设计。Synplify把最后编译的module/ entity & architecture作为顶层设计。故把你所要的顶层设计文件用左键拖拉到源文件菜单的末尾处。
- 5、选择目标器件，设置相关选项。点击Change Target，在弹出的菜单中填入相应选择。
- 6、添加时间约束（如果需要的话）。点击工具条上图标，输入时间约束并保存。关于时间约束请参见有关章节。
- 5、综合。点击RUN即可。
- 6、保存工程文件，选择File--->Save AS。

3.2.1 UNIX环境设置

在第一次使用Synplify之前，要在.cshrc文件中进行路径和license设置。请确认在你的.cshrc文件中有如下内容：

```
setenv SYNPLIFY /edatools/synplify/
if ($?LM_LICENSE_FILE) then
    setenv LM_LICENSE_FILE 1709@LICSrv:$LM_LICENSE_FILE
else
    setenv LM_LICENSE_FILE 1709@LICSrv
endif
set SYNPLIFY_BIN = ($SYNPLIFY/bin)
set path = ($path $SYNPLIFY_BIN)
```

3.2.2 UNIX版本用户界面简介

1. 工具条

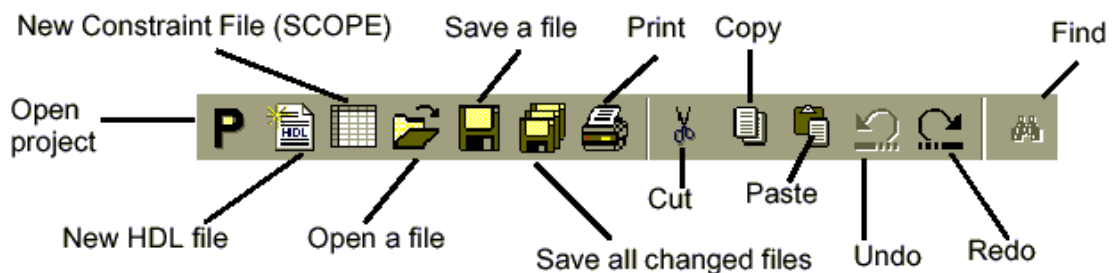


图5 Project 工具条

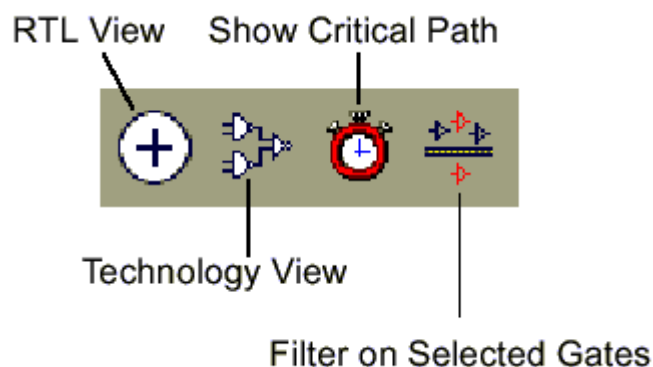


图6 Analyst 工具条

RTL View : 打开一个已编译设计的RTL级层次结构的电路视图。

Technology View: 打开一个已映射（已综合）设计的基于目标器件技术的层次结构的电路视图。

Show Critical Path: 高亮显示Technology View中的关键路径上的器件。

Filter On Selected Gate: 重新显示RTL、Technology View，只显示选中的器件。再次点击恢复。

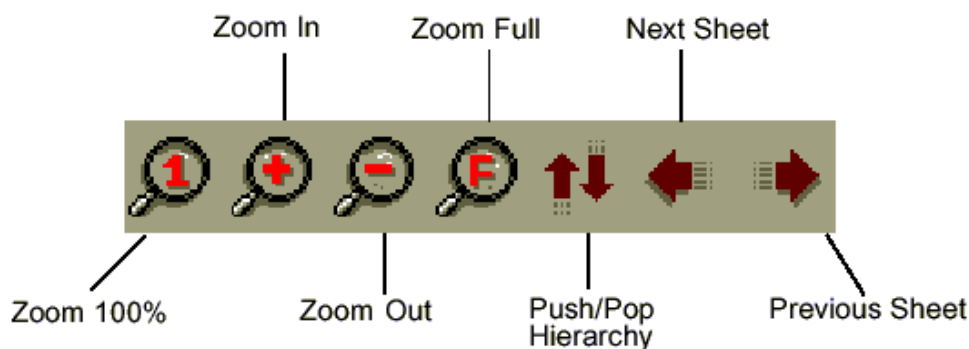


图7 View 工具条

Push/Pop Hierarchy: 用于进入层次结构中的不同的层次。选中后移动鼠标，若光标在某位置显示下箭头表示可进入低一层，点击左键即可，上箭头反之，如果光标显示为叉的话，表示只有唯一层次。

Next/Previous Sheet: 多幅的情况下，显示下（上）一张。

2. 用户界面的按钮

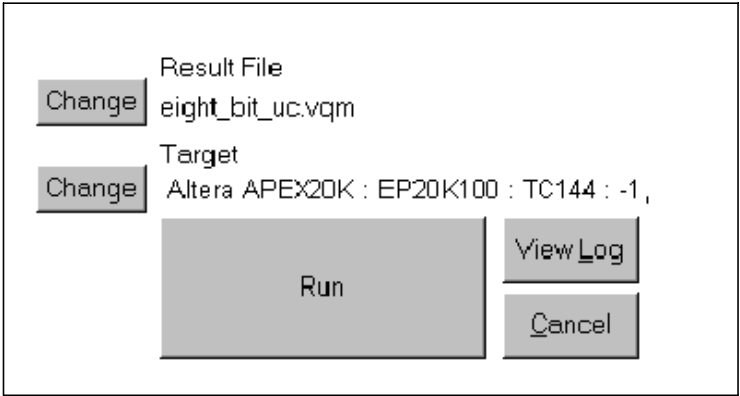


图5 用户界面局部1

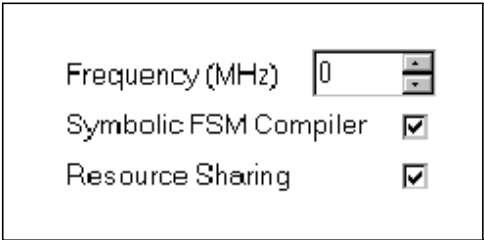


图6 用户界面局部2

表1 用户界面部分按钮描述

Button	描述
Run	运行综合（编译和映射），和菜单中的Run ->Synthesize一样。
Cancel	停止当前的命令
Frequency	设置全局频率，可以通过定义局部属性（Attribute）在局部重新定义。
Symbolic FSM Compiler	决定是否使能专用的FSM优化器。
Resource Sharing	决定综合是否采用资源共享技术。
Change (Result File)	改变输出网表文件的文件名或目录。
Change (Target)	设置目标器件选项。

3.2.3 SCOPE窗口

SCOPE是一个电子表格界面，用于管理设计的时间约束和综合属性。输入的内容保存为一个扩展名为.sdc的文件，也就是设计的约束文件。也可以采用正文编辑器生成约束文件，然后加入源文件列表。对一些需要指定属性施加对象的情况，一个更简单的方法是用鼠标将一个对象从RTL视窗中直接拽到SCOPE相应栏中。

关于时间约束和综合属性将在第n节和第n节中分别详细介绍。

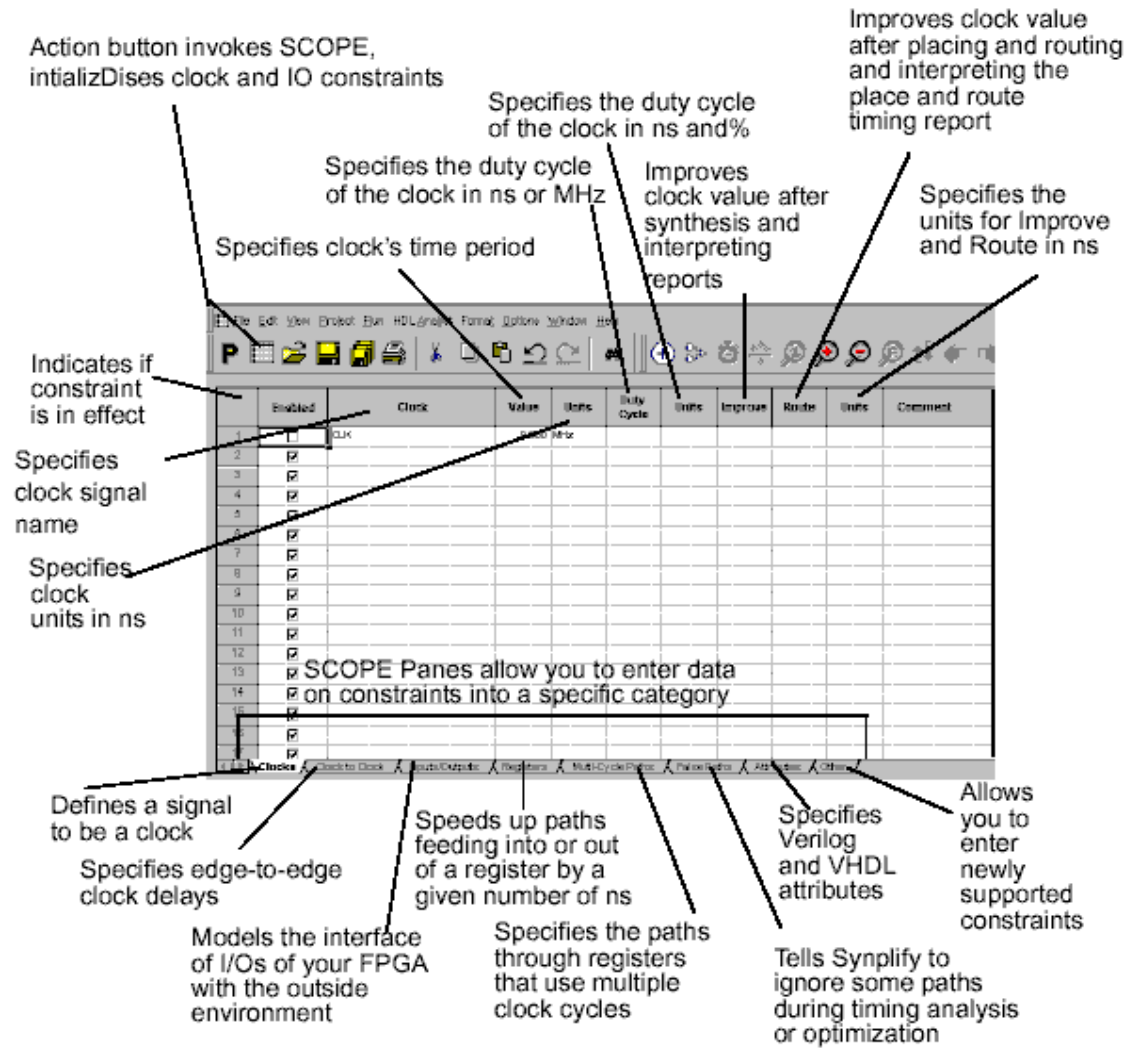


图10 SCOPE 窗口

1. Insert Wizard

每一个SCOPE栏目都有自己的向导帮助你选择对象和进行缺省设置。选中希望的标签，点击右键选择Insert Wizard。

2. Clock

定义一个信号作为时钟（参见第n节define_clock）。Clock域包括

Enable — 标志约束是否生效

Clock — 说明时钟信号名

Value — 说明时钟的数值，单位为Mhz或ns。

Units — 说明Value一栏的单位，Mhz或ns。

Duty cycle — 说明时钟的占空比，单位是ns或%。

Units — 说明时钟的占空比的单位，ns或%。

Improve — 希望改善由这个时钟控制的寄存器的路径延迟的数量。这个数值可以根据Synplify的时间报告中关于相关寄存器的路径延迟的负裕量给出，这是一个高级用户选项。

Route — 希望改善由这个时钟控制的寄存器的路径延迟的数量。与Improve有所不同的是，这一栏的数值应填入布局布线工具的时间报告与Synplify的时间报告相差的数值，这也是高级用户选项。

Improve和Route可以同时使用。

Units — 说明Improve和Route选项的单位，只能是ns。

Comments — 允许你加入一些注释。

3. Clock to Clock

说明不同时钟间沿到沿的延迟（参见define_clock_delay）。可以用来定义不同时钟控制的触发器之间的最大允许延迟，说明一条时钟间的False Path，或是描述一个不对称占空情况的时钟。

Clock1 — 说明第一个时钟的名字

Edge1 — 说明是第一个时钟的上升或下降沿

Clock2 — 说明第二个时钟的名字

Edge2 — 说明是第二个时钟的上升或下降沿

Value — 说明两个沿之间的延迟值，或“false”，false选项指定两个沿之间的路径将被忽略。

4. Input/Output

建立FPGA的I/O端口与外部环境界面的模型，缺省的FPGA外部延迟为0。（参见define_input_delay和define_output_delay）。

Port — 说明端口名

Type — 说明端口类型，Input 或 Output

Value、Improve、Route等与前面的相同或类似。

5. Registers

这个约束的目的是按照给定的时间缩短进入一个寄存器或由其输出的路径延迟。（参见define_input_delay和define_output_delay）

Type — 说明是输入路径还是输出路径

Object — 说明指定寄存器的名字

其他如Improve、Route、Units等与前相同或类似。

6. Multicycle Paths

说明通过寄存器的多时钟周期路径。通过这项约束，你可以为输入或由一个寄存器输出的，或是通过一条连线（net）的所有路径添加额外的时钟周期以放宽时间约束。（参见define_multicycle_path）

Type — 说明路径是输入或输出指定的端口或寄存器或连线
 Port/Register/Net — 说明指定路径时依据的端口或寄存器名
 Value — 说明为该路径提供的全部时钟周期的总数，必须是整数
 Units — 说明Value的单位，只能是个〈周期〉
 其他与前相同或类似

7. False Path

定义在时间分析或优化忽略的路径。〈参见define_false_path〉
 其中Port/Register/Net、Type等与前相同或类似。

8. Attribute

在这里，你可以说明设计属性。其中Object栏和Attribute栏的下拉式菜单是同步的。如果你在Object栏的下拉式菜单里选定一个对象，则Attribute栏的下拉式菜单中只显示可以施加于该对象的属性，反之亦然。

Object Filter — 说明属性施加对象的类型。你可以用这一栏筛选和选择对象。
 Object — 说明施加属性的对象名
 Attribute — 说明施加的属性
 Value — 说明所施加属性的值
 Val Type — 说明属性值的正确类型
 Description — 包含关于该属性的一个简短描述
 其他与前相同或类似。

9. Other

这一栏是为了让高级用户输入新近支持的约束文件命令。这些命令优化和时间分析并不支持的，但是他们会被作为约束传递给布局布线工具。可以在这里使用的约束文件命令包括：

〈Multicycle〉-from 〈从一个寄存器或输入管脚〉-to 〈到一个寄存器或输出管脚〉。适用于Xilinx的M2布局布线工具。

〈False Paths〉-from 〈从一个寄存器或输入管脚〉-to 〈到一个寄存器或输出管脚〉。适用于Xilinx的M2布局布线工具。

3.2.4 生成的报告和信息

1. Log 文件

Synplify将所有综合时产生的报告和信息写入Log文件，Log文件被写入工作目录下，文件名为“*project_name.srr*”。Log文件包括：

被编译的文件列表
 语法或综合的warnings,errors,和Notes
 用户的综合选项设定列表

如果综合时打开了Symbolic FSM Compiler，则会显示抽取出的状态机信息，包括可到达的状态列表

时间报告

资源使用报告

Net Buffering报告

2. 时间报告 (Timing Report)

时间报告包含在Log文件的性能总结 (Perference Summary) 部分, 帮助分析关键路径、调整设计, 增加约束以达到面积或速度目标。

应该正确理解的是, 综合产生的时间报告是估计值, 设计的实际时序状况极大程度的依赖于布局布线工具。如果你调整布局布线工具的时间约束, 可以很容易的让设计的操作频率在10%—20%的范围内变化。

时间报告包括:

- 所有时钟的性能总结
- 所有I/O端口的界面信息, 包括用户的约束, 实际的到达时间和要求值以及裕量。
- 每一个时钟的详细时间报告包括
 - 一个所有在最坏时间裕量一定范围内的路径的起点、终点列表, 最多报告10个起点和10终点。
 - 一个关键路径报告, 包括前面列表报告的所有起点所在的路径。在每一个报告的末尾是该关键路径需要的建立时间。如果时钟频率达到要求, 只报告一条关键路径。

3. 资源使用报告

资源使用报告提供以下信息:

- 设计使用的单元的总数, 和组合逻辑、时序逻辑分别占用的单元的总数。
- 时钟驱动和I/O单元的数量
- 设计中每一个使用单元类型的详细报告

4. Net Buffering报告

Net Buffering报告提供以下信息:

- 被插入缓冲或数据源被复制的Nets
- 上面那些nets被分成的段数
- 插入的缓冲的总数
- 复制数据源增加的寄存器、查找表或其他类型单元的数量

Net Buffering Report Example

```
Net buffering Report:
Badd_c[2] - loads: 24, segments 2, buffering source
Badd_c[1] - loads: 32, segments 2, buffering source
Badd_c[0] - loads: 48, segments 3, buffering source
Aadd_c[0] - loads: 32, segments 3, buffering source
Added 10 Buffers
Added 0 Registers via replication
Added 0 LUTs via replication
```

3.3 批处理工作模式流程

批处理工作有两种方式，分别使用工程文件和Tcl文件。使用批处理方式必须拥有浮动license。

3.3.1 运行工程文件

- 1、启动Synplify 工程窗口。
- 2、设置你的工程选项。
- 3、设置好工程文件：源代码文件，约束文件，Tcl scripts文件。
- 4、保存工程文件*.prj。
- 5、运行：

```
synplify -batch project_file_name.prj
```

3.3.2 运行一个Tcl文件

- 1、编写一个Tcl 文件。格式如下：

```
project -new
#all your other Tcl commands will to be there
project -run
exit
```
- 2、输入你的Tcl 命令。
- 3、保存你的文件。把它与你的工程文件、源代码文件、约束文件放在一起。
- 4、运行：

```
synplify -batch Tcl_script_name.tcl。
```

4 使用HDL Analyst分析和调试设计

4.1 HDL Analyst 简介

HDL Analyst是synplify提供给设计者查看结果，提高设计速度特性和优化面积的强有力的层次结构可视化图形工具。HDL Analyst包含两个原理图视窗，RTL视窗（RTL View）和目标技术视窗（Technology View）。

RTL视窗显示的是高层的与目标技术无关的原理图，是对编译结果的可视化显示。目标技术视窗提供的是相对低层的、特定厂家器件实现的原理图，是对映射结果的可视化显示，它显示的基本元是与特定厂家器件技术有关的诸如查找表、级联和进位链、多路器、触发器等等。

在设计者把他的设计映射到一个器件后，HDL Analyst自动生成层次化的RTL级和基本门级网表。用HDL Analyst 打开你的设计后，你就可以在你的源代码与你的逻辑图之间进行交叉索引（cross_probe）了，你可以查看源代码中一段代码编译或映射后产生的结果是否和预期相符，也可以查看图中关键路径对应的是那一段源代码以做修改。

4.2 HDL Analyst 的应用

4.2.1 POP_UP信息

把鼠标停留在一个目标上片刻，系统会在鼠标附近显示目标的名字（instance, net, port, sheet connector等）。在门级显示的情况下，打开show critical path还可以显示该路径的延迟信息。

4.2.2 状态条显示

如果你打开了View-->Status Bar信息，则把鼠标停留在一个目标上片刻，会同时在状态条上显示目标信息。

4.2.3 链接式选中目标

可以在源代码中选中几行代码处于高亮状态，如一个always块，在相关的逻辑图中你可以看到相应的逻辑图显示高亮。反之，如果你首先选中逻辑图中的一部分使它处于高亮，则相应的代码在编辑窗口中显示高亮。

注意如果设计的代码及综合结果均处于打开状态，则只需用左键单击目标即可在另一个窗口中高亮显示相关的东西。而要求系统弹出另一种显示状态时，则需用左键双击目标。

注意并非所有的代码都有对应的逻辑显示，这是因为在编译或映射时有可能被优化掉了。

4.2.4 查看延迟信息

对以正确综合的设计：

1. 选择HDL-->Technology view 或在工具栏点击相应图标打开综合生成的门级网表。
2. 选择HDL Analyst -->show critical path 或在工具栏点击Show Critical Path 或在逻辑图显示区域按右键弹出菜单选择Show Critical Path。此时关键路径上的部件及网表节点处于高亮状态，所有的延迟信息也标在了instance上面，不过要放大才能看见。
3. 把关键路径孤立出来。选择HDL Analyst-->Filter Schematic 或在工具栏选取按钮Filter Schematic等。此时系统会把关键路径上的所有元素“搜集”到一张逻辑图上，而不管这些元素原来分布在那些逻辑图或那个层次的逻辑图中。再次选择 Filter Schematic 命令可以把你原始的逻辑图重新装进来。

可以灵活应用Slack Margin 命令把你想要的查看的一些关键路径而不只是一条最大延迟的路径显示到一张逻辑图中。选择Analyst -->Set Slack Margin，输入一个超出设计要求的延迟范围，比如是10ns，则所有比你定义的时钟周期大10ns的延迟路径都会显示出来。

正确理解关键路径上的时间延迟显示信息，如 “ out[0] (dfm7a), delay : 12.9 ns, slack: -10.5ns”，表示路径延迟累积到此（寄存器dfm7a，对应设计中的out[0]）为12.9ns，到此已超出时间要求10.5ns。

5 使用Symbolic FSM compiler

在工程窗口中打开Symboic FSM Compiler, 则Synplify在对设计优化时, 自动搜索设计中的状态机, 在不需要改动源代码设计的情况下针对状态机进行优化。

该工具针对状态机的优化包括: 对状态机设计重新选择编码方式 (One-hot 或 Binary) 加以实现; 为你的状态机设计确定一个更恰当的起始状态。究竟选取什么样的编码实现取决于你对时序和对面积的要求的折中。一般情况下, One-hot 的自动机可以达到更快速度, 但有时, 如需要自动机状态寄存器输出进行译码时, 可能会由于译码输入过多造成译码逻辑的级数增加, 反而增大了路径延迟, 降低了设计的速度, 也浪费了面积。而Symbolic FSM compiler 可以帮助你决定使用那种编码方式更合适。

该工具还可以把状态机中多余的状态逻辑删除。例如你设计了一个只有10个状态的四位状态机, 如果“1100”的状态你没有用到, 则Synplify在综合时调用Symbolic FSM compiler对该状态机优化, 删除与“1100”状态有关的无用逻辑。当然, 这也可以通过在源代码中使用综合指示 (Directives) “full_case”实现。

该工具还可对状态机的各个状态的可达到性进行分析, 对一些不能到达的状态加以删除。注意在设计状态机时一个最常见的错误便是存在永远不可到达的状态。

如果在综合时使用了Symbolic FSM compiler, 则在log file中可以查看每一个状态机的综合结果, 以及每一个状态机的可达到的状态。有助于你对状态机设计实现加深认识。

一种值得推荐的使用方式是在初始的综合中使用Symbolic FSM compiler 以获得最优的自动机设计方式, 并相应修改源代码, 而在最终生成结果的综合中禁止Symbolic FSM compiler。

你可以全局使能或全局禁止该工具, 也可以通过在源代码中使用综合指示在局部针对某寄存器调用或禁止该工具。例如:

Verilog Examples:

针对Current_state 使能FSM compiler:

```
reg [3:0] current_state /*synthesis syn_state_machine=1 */;
```

或 reg [3:0] current_state //synthesis syn_state_machine=1 ;

针对Current_state 禁止FSM compiler:

```
reg [3:0] current_state /*synthesis syn_state_machine=0 */;
```

或 reg [3:0] current_state //synthesis syn_state_machine=0 ;

6 怎样用Tcl语言执行批处理任务

Tcl (Tool Command Language) 是一种非常流行的工业标准批处理描述语言, 常用作软件应用的控制。Tcl 是大小写敏感的语言。

应用Synplify 的Tcl script 文件, 设计者可以用批处理命令的形式执行一个综合, 也可以一次执行同一设计多个综合, 尝试不同的器件, 不同的时延目标, 不同的约束条件。

Synplify 的script 文件以 (*.tcl) 保存。

6.1 创建Tcl script 文件

1、建立新工程:

```
project -new
```

2、添加源文件:

```
add_file -verilog
```

```
或 add_file -vhdl
```

3、用综合控制命令去设置目标器件、设计速度目标等。调用symbolic FSM compiler及其他option设置。

```
set_option
```

4、用供应商提供的vendor --specific Tcl 命令去设置目标工艺器件、封装、速度等级，还可以改变一些隐含设置如fan-out 等。

```
vendor--specific
```

5、添加约束文件。

```
add_file --constraint
```

6、执行综合命令

```
project --run
```

7、保存文件为 *.tcl

说明: Tcl 文件的注释行以 # 开头

对文件中的路径名和文件名要用双引号括起来

6.2 常用Tcl 命令说明

6.2.1 工程命令

1、project -new

创建一个新工程。该新工程的名字有project --save 命令指定。

你必须在运行其他Tcl 命令之前运行 project --new 和 project --load 两者之一。

2、project -load "project_name.prj"

装载一个工程文件。

3、project -log_file "new_log_filename.srr "

指定一个新的 log 文件名代替隐含的 log 文件名project_name.srr 。该log文件包含了以下信息: 简单的编译信息, 技术映射信息, 资源利用信息和时序报告。

4、project -result_file "result_filename "

指定一个新的综合结果文件名代替隐含的综合结果文件。

注意: 文件名要小写。

改变文件的扩展名并不能改变综合结果文件的数据格式。

5、project -result_format "result_file_format "

改变综合结果文件的数据格式。

注意: 改变文件的数据格式并不能自动改变文件的扩展名, 仍需用project --result_file。

6、project -compile

编译你的设计工程而不进行技术映射。

系统执行语法检查，可综合性检查，生成RTL级综合结果。

7、project -run

编译并综合你的设计工程。

8、project -save "project_filename.prj "

保存工程文件。

6.2.2 添加文件的命令

1、add_file -verilog "verilog_filename.v "

添加verilog格式的HDL源文件。你可以用 "*.v" 代表你所有的文件。

2、add_file -vhdl [-lib library_name] "vhdl_filename.vhd "

添加vhdl格式的HDL源文件。

3、add_file -constraint "constraint_filename.sdc"

添加约束文件。

6.2.3 控制命令

1、set_option -top_module {verilog_module | vhdl_entity | vhdl_entity.arch}

指定顶层设计

2、set_option -write_verilog {true | false}

把综合生成的网表存成一个verilog格式的文件，以便综合后仿真调用。

3、set_option -write_vhdl {true | false}

把综合生成的网表存成一个vhdl格式的文件，以便综合后仿真调用。

4、set_option -write_apr_constraint {true | false}

把综合中的约束信息生成一个指导布局布线的约束文件，文件的格式与所选器件有关。

5、set_option -frequency MHz_frequency

指定时间约束目标。

6、set_option -technology {vendor_technology}

指定综合选用的器件系列。

7、set_option -part {vendor_part_name}

指定综合选用的器件名。

8、set_option -package {vendor_package_name}

指定综合选用器件的封装。

9、set_option -speedgrade {number}

指定综合选用器件的速度等级。

10、set_option -symbolic_fsm_compiler {true | false}

选择是否在综合时调用针对状态机优化的强有力的工具symbolic FSM compiler。

11、set_option -default_enum_encoding {onehot | sequential | gray}

指定在对设计中的状态机进行综合时选用的隐含编码方式: one-hot, sequential, gray 等。

6.2.4 打开文件的命令

1、open_file -edit_file "filename"

在Synplify内嵌的编辑窗口打开源文件或其他文本文件。

2、open_file -rtl_view

在HDL Analyst窗口中打开当前工程综合结果的RTL级显示。

3、open_file -technology_view

在HDL Analyst窗口中打开当前工程映射后的结果显示。

6.3 Tcl 格式的script文件示例

6.3.1 运行一个script 文件针对多个目标器件进行综合

多次综合以尝试不同的器件

建立一个新Project.

project -new

设定目标速度为 33.3 MHz.

set_option -frequency 33.3

加入Verilog文件到源文件列表

add_file -verilog "andorxor.v"

创建一个Tcl变量\$try_these, 用来对不同的器件进行多次综合

```
set try_these {  
    FLEX8000  
    FLEX10K  
    XC4000  
    XC4000E  
    XC9500      }
```

循环尝试每个器件

foreach technology \$try_these {

根据\$try_these列表设定目标器件

set_option -technology \$technology

运行综合

project -run

打开Technology视窗, 显示实现结果

open_file -technology_view }

打开RTL视窗, 显示RTL电路图。每次综合的RTL级电路是一样的, 因为设计并没有改变

open_file -rtl_view

6.3.2 运行多个频率要求, 并保存为不同的log 文件

#对同一设计在3个不同的时钟频率目标下运行3次综合，比较一下速度/面积的折中结果
#读入一个存在的project。

```
project -load "design.prj"
```

创建一个Tcl 变量\$try_these,用来以不同的频率综合设计

```
set try_these {  
    20.0  
    24.0  
    28.0    }
```

循环尝试每个频率

```
foreach frequency $try_these {
```

根据try_these列表设置频率

```
set_option -frequency $frequency
```

使用<\$frequency>.srr作为log文件名，因为希望保留所有Log文件，否则使用
#<project_name>.srr作为文件名，会使前面的log文件被覆盖。

```
project -log_file $frequency.srr
```

```
project -run
```

显示每次综合的Log文件

```
open_file -edit_file $frequency.srr    }
```

6.3.3 设置控制选项及约束示例

```
project -new
```

设定目标器件，类型，封装和速度级别选项

```
set_option -technology XC4000E  
set_option -part XC4013E  
set_option -package PC84  
set_option -speed_grade -1
```

读入VHDL文件，最后读入顶层设计

```
add_file -vhdl "rotate.vhd"  
add_file -vhdl "memory.vhd"  
add_file -vhdl "top_level.vhd"
```

加入时间约束文件和厂家提供的属性

```
add_file -constraint "design.sdc"
```

顶层文件"top_level.vhd"有两个不同的设计，最后一个是缺省的顶层。这次设置第一个
(design1) 作为顶层模块。在VHDL中，也可以用<entity>.<arch>来说明

```
set_option -top_module design1
```

打开Symbolic FSM Compiler

```
set_option -symbolic_fsm_compiler true
```

```
set_option -frequency 30.0
```

#保存Project，缺省的结果文件是“<project_name>.<ext>”。如希望被保存为别的名字，诸如


```
# “design.xnf” , 可以使用 “ project -result_file "<name>.xnf"” 命令
project -save "design.prj"

project -run
open_file -rtl_view
open_file -technology_view

#
-----
# 下面是约束文件 “design.sdc”
#
-----
# Timing Constraints:
# -----
#除clk_fast需要66.0Mhz, 其余都是缺省的目标频率30.0 MHz, 故覆盖clk_fast的频率
define_clock {clk_fast} -freq 66.0

#输入延迟 4 ns
define_input_delay -default 4.0

# 信号 “sel” 例外, 输入延迟 8 ns
define_input_delay {sel} 8.0

# 片外输出延迟 3.0 ns
define_output_delay -default 3.0

# 之前的结果发现关键路径是输入到inst3.q[0] (存储器内) 的路径, 希望减少该路径的延迟3.0 ns.
define_reg_input_delay {inst3.q[0]} -improve 3.0

# -----
# Xilinx提供的属性约束
# -----
# 指定标量端口 “sel” 的位置。这些VHDL对象名是大小写敏感的
define_attribute {sel} xc_loc "P139"

# 指定一个总线的所有位的pad位置
define_attribute {b[7:0]} xc_loc "P14, P12, P11, P5, P21, P18, P16, P15"

# 指定一个pad为快速转换类型
define_attribute {a[5]} xc_fast 1

# 对clk_slow使用普通buffer而不是时钟buffer, 保留时钟buffer供其他高速时钟使用
define_attribute {clk_slow} syn_noclockbuf 1

#设置最大扇出限制为10000.
define_attribute {clk_slow} syn_maxfan 10000
```

6.3.4 自底向上的综合示例

```
# 从其他Tcl脚本读入命令, 每一个脚本编译一个独立的模块并有它自己的约束文件
source "statemach.tcl"
source "fifo.tcl"
```

```

source "cherstrp.tcl"

# 综合完独立的逻辑模块，创建一个顶级模块，加入顶层文件、约束文件及选项
project -new
add_file -vhdl top_level.vhd
add_file -constraint top_level.sdc

set_option -technology FLEX10K
set_option -part EPF10K70
set_option -speed_grade -3
set_option -frequency 50.0
set_option -symbolic_fsm_compiler true

# 设置输出信息
project -result_file top_level.edf
project -log_file top_level.srr

project -save top_level.prj
project -run

open_file -rtl_view
open_file -technology_view

# -----
# 以下为statemach.tcl文件，被bottom_up.tcl用命令“source statemach.tcl”读入
# -----

project -new
add_file -vhdl statemach.vhd
add_file -constraint statemach.sdc

set_option -technology FLEX10K
set_option -part EPF10K70
set_option -speed_grade -3
set_option -frequency 50.0
set_option -symbolic_fsm_compiler true

project -result_file statemach.edf
project -log_file statemach.srr

project -save statemach.prj

project -run

# -----
# 下面是statemach.sdc
# -----
# Timing Constraints:
# -----

define_input_delay -default -100
define_output_delay -default -100
define_input_delay RESET -10
define_reg_input_delay {q[8]} -improve 4.0

# -----
# Altera-Specific Attributes:

```

```
# -----
      define_attribute {inst1.sqrt8} altera_implement_in_eab 1
```

6.4 运行script文件

- 1、在UNIX环境下执行synplify -batch tcl_script_name.tcl
- 2、在工程窗口中选择File --> Run Tcl script

7 时间约束

定义时间约束是为了让综合结果满足预期的时序要求，时间约束通常分为两类，一是通用时间约束，用于目标结构的时序要求；二是黑盒时间约束，用于在设计中指定为黑盒的模块。

时间约束可以通过三种方式加入：

- 通过SCOPE，会在保存输入的结果时生成一个约束文件。这是最推荐的方式，能输入绝大多数约束，但无法输入需要在源代码中加入综合指示（Directives）的约束。
- 直接使用正文编辑器创建生成约束文件。
- 在源代码中直接加入综合属性和指示，应该只在非此不可的时候才采用。

7.1 书写约束文件的一些规则

- 1、采用Tcl，大小写敏感。
- 2、提供正确的对象（如signals，ports，instances等）名，要和源代码中的名字一致。
- 3、所有的对象要用大括号括起来，如{foo[2]}。
- 4、用圆点符（.）作为层次连接符，连接层次结构设计中不同层次实例名。
- 5、用好恰当的前缀：i：表示instance。p：表示port。n：表示内部nets。

7.2 Verilog对象表示语法

这里介绍的是关于对象名的语法，对象包括Verilog的Module、component实例、端口和内部网络，用于在约束文件中添加时间约束。具体如下：

1、不允许在名字中间出现空格。可以使用“?”或“*”作为通配符，但是不匹配层次区分符“.”。

2、对Verilog中的module名采用下面的语法

v:cell

这里

- v: 定义一个观察对象
- cell 是Verilog中的module名

3、对Instance, ports 和net 名采用下面的语法：

cell typespec: object_path

其中

- cell 是Verilog module名

- typespec 为单个字母后跟一个冒号，用于设计中有两个或更多的对象同名的情况下进行区分，其可选值如下：
 - i: instance名。
 - p: 完整的port名，表示端口自己。
 - b: 部分port名，表示端口的位片断。
 - n: 内部net名。定义所有内部nets都需要这个类型说明。
- object_path 为一实例路径，用点(.)作为层次区分符，以目标对象名结尾。

示例：

表示 module statemod中的instance statereg 的所有位：
 statemod|i:statereg[*]
 表示仅次于顶层的层次中名字以state开头的实例：
 i:*.state*
 表示port mybus[19:0] 而不是驱动该端口的同名寄存器：
 p:mybus[19:0]
 表示顶层 port mybus的第一比特而不是驱动该端口的同名寄存器：
 B:mybus[1]
 下面是约束文件中上述例子使用的情况：
 define_attribute {statemod|i:statereg[*]} syn_encoding sequential
 define multicvcl path -to {*.slowred[*]} 2

7.3 HDL源代码中的约束

不管怎样，这种方法是不推荐采用的，通常只有对黑盒时间约束采用这样的方法。对于加入源代码的时间约束，只要做任意改动都要重新编译，对大型的设计来说是难以接受的。

HDL源代码中的约束以注释形式加入。

时间约束具有唯一的名字，使用实数时间值，并与input、output、register信号相关联。下面是示例：

```
// 缩短输入register_a 的路径延迟3.0ns
reg register_a /* synthesis syn_reg_input_delay_improve=3.0 */;
// 定义通过input_a 的路径的输入延迟为10.0ns
input input_a; //synthesis syn_input_delay=10.0
/* 增加额外的布线延迟2.0ns (在布局布线中确定的) */
input input_a; //synthesis syn_input_delay_route=2.0
```

7.1 通用时间约束

1、define_clock

语法：**define_clock** [-disable] *clock_name* [{-freq MHz|-period ns}{-duty_ns ns| -duty_pct percent}}] [-improve ns] [-route ns]

说明：

-improve：用于提高由该时钟控制的寄存器延迟，这个数值可以根据Synplify的时间报告中关于相关寄存器的路径延迟的负裕量给出，这是一个高级用户选项。

-route: 用于提高由该时钟控制的寄存器延迟, 以满足布局布线时序报告中的时延与综合时序报告时延的差值, 与Improve有所不同的是, 这一栏的数值应填入布局布线工具的时间报告与Synplify的时间报告相差的数值, 这也是高级用户选项。

-duty_ns 和-duty_pct 分别定义用ns或百分比说明的时钟占空比。

define_clock 的优先级要比set_option -frequency高。

对内部时钟生成器生成的时钟, 要用define_clock定义, 时钟名就是输出最终时钟的寄存器实例名。

```
示例:    define_clock clock_a -freq 33.0
          define_clock clock_b -period 100.0
          define_clock ten_mhz_internal_clock -freq 10.0
```

2、define_clock_delay

语法: **define_clock_delay [-rise | -fall] clock1 [-rise | -fall] clock2 delay**

说明: 该命令可以定义不同时钟之间的最大延迟; 定义两时钟之间是伪路径 (delay的值取 -false); 定义占空比不对称 (即不为0.5) 的时钟。

delay的值是两个沿之间的最小值。

例子: //例1: clka 与clkb 的第一个上升沿同步, clkb 的频率是 clka 的两倍。

```
define_clock clka -period 100.0
define_clock clkb -period 50.0
define_clock_delay -rise clka -rise clkb 50.0
define_clock_delay -rise clkb -fall clka 50.0
define_clock_delay -fall clka -rise clkb 50.0
define_clock_delay -fall clkb -rise clka 25.0
```

//例2: 定义clka的上升沿与clkb的上升沿的路径为伪路径。

```
define_clock_delay -rise clka -rise clkb -false
```

//例3: 定义clka的高电平占空比为25%。

```
define_clock clka -perioc 100
define_clock_delay -rise clka -fall clka 25
define_clock_delay -fall clka -rise clka 75
```

3、define_reg_input_delay

语法: **define_reg_input_delay register_name [-improve ns] [-route ns]**

说明: 期望输入某寄存器的路径延迟要改善的时间值。register_name是单比特寄存器而不能是寄存器总线。

```
例子:    define_reg_input_delay reg_a -improve 2.0
          define_reg_input_delay reg_a -route 1.8
```

4、define_reg_output_delay

语法: **define_reg_output_delay register_name [-improve ns] [-route ns]**

说明: 期望从某寄存器输出的路径延迟要改善的时间值。register_name是单比特寄存器而不能是寄存器总线。

例子: `define_reg_output_delay reg_a -improve 1.0`
`define_reg_output_delay reg_a -route 1.8`

5、define_input_delay

语法: `define_input_delay {input_port_name | -default} ns [-improve ns] [-route ns]`

说明: 定义输入延迟, 该延迟为信号在片外到达芯片管脚相对于时钟的第一个边沿消耗的时间。无定义时的缺省值是0ns。

例子: //例1: 对所有的输入端口设置3ns延迟
`define_input_delay -default 3.0`
 //例2: 对输入端口input_a设置10ns 延迟
`define_input_delay input_a 10.0`
 //例3:
`define_input_delay input_a 0 -improve 2.0`
`define_input_delay input_a 5.0 -improve 3.0`
`define_input_delay input_a 10.0 -route 1.8`

6、define_output_delay

语法: `define_output_delay {out_port_name | -default} ns [-improve ns] [-route ns]`

说明: 定义输出延迟, 该延迟为信号输出芯片管脚到达被驱动的逻辑需要的时间。无定义时的缺省值是0ns。

例子: `define_output_delay -default 8.0`
`define_output_delay output_a 10.0`
`define_output_delay output_a 0 -improve 2.0`
`define_output_delay output_a 5.0 -improve 3.0`
`define_output_delay output_a 10.0 -route 1.8`

7、define_multicycle_path

语法: `define_multicylce_path {-from reg_or_input | -to reg_or_output | -through net } clock_cycles`

说明: 定义多时钟周期路径。不要将-through选项和其他选项一起使用, 如果用-through开关选项, 则需要在源代码使用综合指示“syn_keep”保证关联的信号名始终不会改变。

例子: `define_multicycle_path -from register_a 2`
`define_multicycle_path -from register_b 2`
`define_multicycle_path -to register_c 2`

8、define_false_path

语法: `define_false_path { -from reg_or_input | -to reg_or_output | -through reg }`

说明：定义在时间分析和优化时可以忽略得路径。不要将-through选项和其他选项一起使用，如果用-through开关选项，则需要在源代码使用综合指示“syn_keep”保证关联的信号名始终不会改变。

例子：define_flase_path -from register_a
define_false_path -to register_c

9、syn_reference_clock

语法：define_attribute register syn_reference_clock clock

说明：该命令用来定义一个参考时钟。比如一个寄存器的使能信号每两个时钟周期使能一次，则可对该寄存器设置一个参考时钟为原来时钟频率的两倍。

例子：define_clock dummy -period 40.0
define_attribute myreg[31:0] syn_reference_clock dummy
对所有使能信号为en40的寄存器设置一个参考时钟。
define_attribute {find -reg -enable en40} syn_reference_clock ck2

7.2 黑盒时间约束

黑盒是在设计中用综合指示“syn_black_box”声明或实例化的模块。对黑盒的时间约束只能通过通过在源代码中加入综合指示完成。下面是黑盒时间约束：

syn_tpd_n：定义穿过黑盒的组合逻辑延迟

syn_tsu_n：定义黑盒的输入延迟。

syn_tcon：定义黑盒的输出延迟。

n为1－10的整数。

用[!]clock表示clock的负沿。

例子：

```
module ram32x4(z, d, addr, we, clk);
/* synthesis black_box
syn_tpd1="addr[3:0]->z[3:0]=8.0"
syn_tsu1="addr[3:0]->[!]clk=2.0"
syn_tsu2="we->clk=3.0" */
output [3:0] z;
input [3:0] d;
input [3:0] addr;
input we;
input clk;
Endmodule
```

7.4 特定厂家的时间约束文件

Synplify可以生成作为厂家布局布线工具输入的时间约束文件，使能或禁止这项功能，在project视窗下，选中或取消选中Option->Write vendor constraints。

Synplify 为 Altera Max+plus II 或 Quatus 生成的约束文件为 “.acf ”。为 Xilinx 布局布线工具生成的约束文件为 “.ncf ”。有关具体内容参见《Synplify Reference Manual》中 “Forward Annotation” 章节。

8 综合属性 (Attributes) 和指示 (Directives)

8.1 简介

综合指示用于控制综合中编译阶段的设计分析，因而必须加入到源代码中。属性是在编译后读入的，因而可以既可以在源程序中说明也可以在约束文件中说明。约束文件提供了较大的灵活性，可以使得修改约束而不用重新编译源程序，因而是强烈推荐采用的方法。

在 Verilog 源程序中说明指示（或属性）采用注释的方法，语法如下：

```
// synthesis directive|attribute = "value"
或 /* synthesis directive |attribute = "value" */
```

8.2 厂家提供属性

8.2.1 Altera

表2 Altera 提供的属性

Attribute	描述
altera_area	<p>说明一个 Altera 的 Component 占据的大概面积，通常用来定义一个黑盒的面积，其属性值为估计的要使用的 LCELL 的个数，语法如下：</p> <p>.sdc 文件</p> <pre>define_attribute {arch_component_module} altera_area {integer}</pre> <p>Verilog 源代码</p> <pre>object /*synthesis altera_area = integer*/ ;</pre> <p>在 Verilog 中，这个属性只能放在 module 处，例如：</p> <pre>module de(clk,out,a)/*synthesis altera_area=12*/ ; /* Other coding */</pre>
altera_auto_use_eab	<p>控制在 FLEX10K 和 ACEX 的设计中，RAMs 和 ROMs 是否由扩展阵列块（EABs）实现。对设计中的 RAMs 和 ROMs，Altera 映射器自动使用 EABs 实现，但是如果你使用了 altera_implement_in_eab 属性或该资源在黑盒中或在实例化 LPMs 时被使用，映射器因为不知道资源已被使用而可能造成重复使用。将 altera_auto_use_eab 的属性值设为 0 或 false，映射时将不使用 EABs 实现。你可以用 syn_ramstyle、syn_romstyle 指定存储器实现的方法。语法如下：</p> <p>.sdc 文件中</p> <pre>define_global_attribute altera_auto_use_eab {0 1}</pre> <p>Verilog 文件中</p> <pre>object /* synthesis altera_auto_use_eab = 0 1 */ ;</pre>
altera_auto_use_esb	<p>控制在 APEX20K 设计中，RAMs 和 ROMs 是否由扩展系统块（ESBs）中的 PTERMs 实现，语法如下：</p> <p>.sdc 文件中</p> <pre>define_global_attribute altera_auto_use_esb {0 1}</pre> <p>Verilog 中</p> <pre>object /* synthesis altera_auto_use_esb = 0 1 */ ;</pre>

altera_chip_pin_lc	<p>指定Altera的I/O管脚的位置，语法如下：</p> <p>.sdc 文件中</p> <pre>define_attribute {port_name} altera_chip_pin_lc {@pin_number}</pre> <p>Verilog 中</p> <pre>object /*synthesis altera_chip_pin_lc = "@pin_number" */;</pre> <p>注意在APEX中管脚名不用@。</p>
altera_implement_in_eab	<p>在FLEX10K或ACEX 设计中，指定一个逻辑设计单元由扩展阵列块（EABs）实现，语法如下：</p> <p>.sdc 文件中</p> <pre>define_attribute {v:module_or_architecture_name} altera_implement_in_eab {1}</pre> <p>Verilog 中</p> <pre>object /* synthesis altera_implement_in_eab 1 */;</pre>
altera_implement_in_esb	<p>在FLEX10K 或ACEX 设计中，指定一个逻辑设计单元由扩展系统块（ESBs）中的PTERM实现</p> <p>.sdc 文件中</p> <pre>define_attribute {v:module_or_architecture_name} altera_implement_in_esb {1}</pre> <p>Verilog 中</p> <pre>object /* synthesis altera_implement_in_esb 1 */;</pre>
altera_io_opendrain	<p>设置APEX20KE 结构中输出和双向端口的漏极开路模式，该属性的缺省值为0，即不使用漏极开路模式，语法如下</p> <p>.sdc 文件中</p> <pre>define_attribute {output bidir} altera_io_opendrain {1 0}</pre> <p>Verilog 中</p> <pre>object /* synthesis altera_io_opendrain = 1 0 */;</pre>
altera_io_powerup	<p>控制 APEX20KE I/ O 寄存器中未预定义置位或复位状态的上电模式，只能在未对该寄存器定义上电置位或复位状态时使用，Quartus缺省设置I/O寄存器的上电状态为低。如果该寄存器无法使用I/O寄存器，Synplify警告并忽略该属性。语法如下：</p> <p>.sdc 文件中</p> <pre>define_attribute {port} altera_io_powerup {high low}</pre> <p>Verilog 中</p> <pre>object /*synthesis altera_io_powerup = "high low" */;</pre> <p>注意在SCOPE中，除了对port外，你还可以对bus或bus的片段使用这一属性。</p>
syn_direct_enable	<p>控制在APEX、FLEX 和ACEX 设计中是否指定一个时钟使能net分配 给指定存储单元的使能端，缺省情况下软件会作最优选择，如果在有多个选择时，通过将该属性的值置为1，你可以为综合工具指定一个确定的node作为唯一的选择，语法如下：</p> <p>.sdc 文件中</p> <pre>define_attribute {n:net_name} syn_direct_enable {1 0}</pre>
syn_forward_io_constraints	<p>使能或禁止 I/ O 约束（define_input_dealy、define_output_delay）被写入生成的布局布线工具的时间约束文件，缺省值是1，语法如下：</p> <p>.sdc 文件中</p> <pre>define_global_attribute syn_forward_io_constraints = {1 0}</pre> <p>Verilog 中</p> <pre>object /* synthesis syn_forward_io_constraints = 1 0 */;</pre>

syn_hier	<p>控制在优化和映射时对一个Module或Component实例的边界的层次化处理。这个属性可以用于设计中的多个模块。syn_hier的值包括:</p> <p>soft — 确省值, Synplify决定最优的边界, 只影响指定的设计单元</p> <p>firm — 保留设计单元的界面, 允许cell跨越边界, 只影响指定的单元</p> <p>hard — 严格保留设计单元的界面, 只影响指定的单元</p> <p>remove — 去除声明了该属性的层次, 但不影响更低层次的单元</p> <p>macro — 严格保留设计的界面和内容</p> <p>flatten — 去除所有低于该属性所在层次的层次, 属性所在层次不动</p> <p>你可以将flatten和其他值组合使用</p> <p>flatten,soft — 等于flatten</p> <p>flatten,firm — 平面化低于属性所在层次的所有层次, 保留设计单元的界面, 保留对跨越边界的cell的优化</p> <p>flatten,remove — 平面化包括属性所在层次和低于该层次的所有层次</p> <p>在低层声明的该属性取代在高层声明的这一属性。该属性的语法如下:</p> <p>.sdc 文件中</p> <pre>define_attribute {object} syn_hier {value}</pre> <p>Verilog 中</p> <pre>object /* synthesis syn_hier = "value" */;</pre>
syn_maxfan	<p>设置一个单独的输入脚或寄存器的输出端的扇出限制, 如希望禁止该功能, 输入一大数, 例如1000。若将其应用于一个net, 元件会生成一个KEEPBUFFER,并将该属性附加其上。如应用于了一个低层模块, 该模块在综合时被优化, 则属性将被附加于它的上一层模块。如不希望如此, 使用syn_hier的hard。该属性语法如下:</p> <p>.sdc 文件中</p> <pre>define_attribute {object} syn_maxfan {integer}</pre> <p>Verilog 中</p> <pre>object /* synthesis syn_maxfan = "value" */;</pre>
syn_netlist_hierarchy	<p>决定输出的EDIF网表是层次化的还是平面化的, 是施加于顶层模块的全局属性, 其语法如下:</p> <p>.sdc 中</p> <pre>define_global_attribute syn_netlist_hierarchy {0 1}</pre> <p>Verilog 中</p> <pre>object /* synthesis syn_netlist_hierarchy = 0 1 */;</pre>
syn_noarrayports	<p>防止端口在输出的EDIF网表中被加入信号组, 而让它保持独立, 其语法如下:</p> <p>.sdc 文件中</p> <pre>define_global_attribute syn_noarrayports {0 1}</pre> <p>Verilog 中</p> <pre>object /* synthesis syn_noarrayports = 0 1 */;</pre>
syn_pipeline	<p>在APEX、FLEX和ACEX设计中, 允许将跟在乘法器后面的寄存器移入乘法器实现以提高频率, 语法如下:</p> <p>.sdc 文件中</p> <pre>define_attribute {register} syn_pipeline {1 0}</pre> <p>Verilog 中</p> <pre>object /* synthesis syn_pipeline = 1 0 */;</pre>
syn_probe	<p>加入探测点以供测试调试, 测试点可以使内部信号作为端口在顶层出现。</p> <p>.sdc 文件中</p> <pre>define_attribute {n:netName} syn_probe {0 1}</pre> <p>Verilog 中</p> <pre>object /* synthesis syn_probe = 0 1 */;</pre>

syn_ramstyle	<p>确定在APEX、FLEX10K和ACEX设计中RAMs的实现方法，如寄存器或内存块等，你可以将这个属性施加于RAM驱动的寄存器信号或RAM实例名，可选值为：</p> <p>registers — RAM由寄存器实现</p> <p>block_ram — 由厂家器件提供的方式实现RAM</p> <p>这是一个与具体厂家有关的属性，参见表3。其语法如下：</p> <p>.sdc 文件中</p> <pre>define_attribute {signal_name[bit_range]} syn_ramstyle {string}</pre> <p>Verilog 中</p> <pre>object /* synthesis syn_ramstyle = "string" */;</pre>
syn_reference_clock	<p>声明一个有别于该寄存器时钟脚上的时钟频率的时钟频率，对APEX、FLEX、和ACEX设计有效。参见8.1的9。</p>
syn_romstyle (Altera)	<p>确定在APEX、FLEX10K和ACEX设计中ROM的实现架构。缺省情况下Synplify采用逻辑方法实现小（地址线小于7）ROM，采用EABs（对FLEX、ACEX）或ESBs（对APEX）实现大（地址线不小于7）ROM。可选值为logic或block_rom，语法如下：</p> <p>.sdc 文件中</p> <pre>define_attribute {rom_primitive} syn_romstyle {logic block_rom}</pre> <p>Verilog 中</p> <pre>object /* synthesis syn_romstyle = "logic block_rom" */;</pre> <p>For example</p> <pre>Reg [3:0] z /*synthesis syn_romstyle="block_rom"*/;</pre>
syn_useenables	<p>防止产生带时钟使能端的寄存器，语法如下：</p> <p>.sdc 文件中</p> <pre>define_attribute {register_instance} syn_seenables {0 1}</pre> <p>Verilog 中</p> <pre>object /* synthesis syn_useenables = 0 1 */;</pre>
syn_useioff (Altera)	<p>在设计中使用I/O触发器，只适用于APEX，你可以对一个端口或全局使用这个属性，缺省值是禁止使用，语法如下：</p> <p>.sdc 文件中</p> <pre>define_attribute {port_name} syn_useioff {1 0}</pre> <pre>define_global_attribute syn_useioff {1 0}</pre> <p>Verilog 中</p> <pre>object /* synthesis syn_useioff = 1 0 */;</pre>

表3 Altera支持的RAM实现类型

器件	属性值	实现方法
Flex10K, ACEX	无	缺省情况：小RAM由寄存器实现。大的RAM，如果altera_auto_ead=1或ture，则由EABs实现。
	registers	RAMs由寄存器实现
	block_ram	RAMs 由EABs实现
APEX系列	无	缺省情况：小RAM由寄存器实现。大的RAM，如果altera_auto_ead=1或ture，则由EABs实现。
	registers	RAMs由寄存器实现
	block_ram	RAMs 由EABs实现

8.2.1 Xilinx

表4 Xilinx提供的属性

Attribute	描述
syn_direct_enable	<p>控制在Spartan2、XC4000、XC5000和Virtex系列设计中是否指定一个时钟使能net分配给指定存储单元的使能端，缺省情况下软件会作最优选择，如果在有多个选择时，通过将该属性的值置为1，你可以为综合工具指定一个确定的node作为唯一的选择，语法如下：</p> <p>.sdc 文件中</p> <pre>define_attribute {n:net_name} syn_direct_enable {1 0}</pre>
syn_edif_bit_format	<p>全局字符格式属性，控制EDIF输出文件中总线信号名、端口名和向量范围的命名格式与风格，第一个参数（%n，%d，或%u）控制名字的大小写，第二个（%i）控制向量范围的格式。缺省的名字的大小写与源代码中的相同。其中%u：整个名字变成大写；%d：整个名字变成小写；%n：保持不变；该属性语法如下：</p> <p>.sdc 文件中</p> <pre>define_global_attribute syn_edif_bit_format {%n %u %d [char]%i[char]}</pre> <p>Verilog 中</p> <pre>object /* synthesis syn_edif_bit_format = "%n %u %d [char]%i[char]" */;</pre>
syn_edif_scalar_format	<p>全局字符格式属性，控制EDIF输出文件中端口名和标量信号的命名格式与风格，%u将名字变成大写，%d将名字变成小写，%n保持源代码中的方式不变。其语法如下：</p> <p>.sdc 文件中</p> <pre>define_global_attribute syn_edif_scalar_format {%u %d %n}</pre> <p>Verilog 中</p> <pre>object /* synthesis syn_edif_scalar_format = " %u %d %n" */;</pre>
syn_hier	<p>控制在优化和映射时对一个Module或Component实例的边界的层次化处理。这个属性可以用于设计中的多个模块。syn_hier的值包括：</p> <p>soft — 确省值，Synplify决定最优的边界，只影响指定的设计单元</p> <p>firm — 保留设计单元的界面，允许cell跨越边界，只影响指定的单元</p> <p>hard — 严格保留设计单元的界面，只影响指定的单元</p> <p>remove — 去除声明了该属性的层次，但不影响更低层次的单元</p> <p>flatten — 去除所有低于该属性所在层次的层次，属性所在层次不动</p> <p>你可以将flatten和其他值组合使用</p> <p>flatten,soft — 等于flatten</p> <p>flatten,firm — 平面化低于属性所在层次的所有层次，保留设计单元的界面，保留对跨越边界的cell的优化</p> <p>flatten,remove — 平面化包括属性所在层次和低于该层次的所有层次在低层声明的该属性取代在高层声明的这一属性。该属性的语法如下：</p> <p>.sdc 文件中</p> <pre>define_attribute {object} syn_hier {value}</pre> <p>Verilog 中</p> <pre>object /* synthesis syn_hier = "value" */;</pre>
syn_maxfan	<p>设置一个单独的输入脚或寄存器的输出端的扇出限制，如希望禁止该功能，输入一大数，例如1000。若将其应用于一个net，元件会生成一个KEEPBUFFER,并将该属性附加其上。如应用于了一个低层模块，该模块在综合时被优化，则属性将被附加于它的上一层模块。如不希望如此，使用syn_hier的hard。该属性语法如下：</p> <p>.sdc 文件中</p> <pre>define_attribute {object} syn_maxfan {integer}</pre> <p>Verilog 中</p> <pre>object /* synthesis syn_maxfan = "value" */;</pre>

syn_netlist_hierarchy	<p>决定输出的EDIF网表是层次化的还是平面化的，是施加于顶层模块的全局属性，其语法如下：</p> <p>.sdc 中</p> <pre>define_global_attribute syn_netlist_hierarchy {0 1}</pre> <p>Verilog 中</p> <pre><i>object</i> /* synthesis syn_netlist_hierarchy = 0 1 */;</pre>
syn_noarrayports	<p>防止端口在输出的EDIF网表中被加入信号组，而让它保持独立，其语法如下：</p> <p>.sdc 文件中</p> <pre>define_global_attribute syn_noarrayports {0 1}</pre> <p>Verilog 中</p> <pre><i>object</i> /* synthesis syn_noarrayports = 0 1 */;</pre>
syn_noclockbuf	<p>控制全局时钟Buffer的自动插入，设属性值为0，则不插入全局时钟Buffer，其语法如下：</p> <p>.sdc 文件中</p> <pre>define_attribute {clock_port} syn_noclockbuf {1 0}</pre> <p>Verilog 中</p> <pre><i>object</i> /* synthesis syn_noclockbuf = 1 0 */;</pre>
syn_pipeline	<p>在XC4000和Virtex系列设计中，允许将跟在乘法器后面的寄存器移入乘法器实现以提高频率，其语法如下：</p> <p>.sdc 文件中</p> <pre>define_attribute {register} syn_pipeline {1 0}</pre> <p>Verilog 中</p> <pre><i>object</i> /* synthesis syn_pipeline = 1 0 */;</pre>
syn_probe	<p>加入探测点以供测试调试，测试点可以使内部信号作为端口在顶层出现。</p> <p>.sdc 文件中</p> <pre>define_attribute {n:netName} syn_probe {0 1}</pre> <p>Verilog 中</p> <pre><i>object</i> /* synthesis syn_probe = 0 1 */;</pre>
syn_ramstyle	<p>确定设计中RAMs的实现方法，如寄存器或内存块等，你可以将这个属性施加于RAM驱动的寄存器信号或RAM实例名，可选值为：</p> <p>registers — RAM由寄存器实现</p> <p>block_ram — 由厂家器件提供的方式实现RAM</p> <p>select_ram — 由分布在CLBs中的RAMs实现，这是Xilinx器件的缺省值</p> <p>no_rw_check — 由Virtex Block SelectRAM+实现。如果是双端口RAM将不插入旁路逻辑。仅当设计中不会出现同时读写同一地址的情况才可以使用这个属性值。在无旁路逻辑的情况下，如果同时读写Block SelectRAM+的同一地址，输出将是不确定的，也会导致RTL级仿真和综合后仿真的结果不一致。不过如果确定设计不会出现这种情况，可以使用这个值以简化逻辑。</p> <p>这是一个与具体厂家有关的属性，参见表5。其语法如下：</p> <p>.sdc 文件中</p> <pre>define_attribute {signal_name[bit_range]} syn_ramstyle {string}</pre> <p>Verilog 中</p> <pre><i>object</i> /* synthesis syn_ramstyle = "string" */;</pre>
syn_reference_clock	<p>声明一个有别于该寄存器时钟脚上的时钟频率的时钟频率，对XC4000系列、XC5200和Virtex系列设计有效。参见8.1的9。</p>

syn_romstyle (Xilinx)	<p>确定在XC4000和Virtex系列设计中ROM的实现架构，在Synplify中用case语句生成ROMs，至少要有一半的地址被赋予有效值。可以选择采用逻辑或是分布ROM实现ROM。其语法如下：</p> <p>.sdc 文件中</p> <pre>define_attribute {rom_primitive} syn_romstyle {select_rom logic}</pre> <p>Verilog 中</p> <pre>object /* syn_romstyle = "select_rom logic" */;</pre>
syn_srlstyle (Xilinx)	<p>决定怎样实现移位时序（SeqShift）逻辑，只适用于Virtex系列。可选值： registers—用寄存器实现 select_srl—用移位寄存器查找表（SRL）16原语实现</p> <p>.sdc 文件中</p> <pre>define_attribute {object} syn_srlstyle {registers select_srl}</pre>
syn_tristatetomux	<p>让软件检查所有三态驱动的网络，将输出数目小于属性值的三态驱动替换为多路器，只适用于Virtex系列。其语法如下：</p> <p>.sdc 文件中</p> <pre>define_attribute {object} syn_tristatetomux {integer}</pre> <p>Verilog 中</p> <pre>object /* synthesis syn_tristatetomux = integer */;</pre>
syn_useenables	<p>防止产生带时钟使能端的寄存器，语法如下：</p> <p>.sdc 文件中</p> <pre>define_attribute {register_instance} syn_useenables {0 1}</pre> <p>Verilog 中</p> <pre>object /* synthesis syn_useenables = 0 1 */;</pre>
syn_useioff (Xilinx)	<p>在设计中使用I/O触发器，只适用于XC4000和Virtex系列，其语法如下：</p> <p>.sdc 文件中</p> <pre>define_attribute {port} syn_useioff {1 0} define_global_attribute syn_useioff {1 0}</pre> <p>Verilog 中</p> <pre>object /* synthesis syn_useioff = 1 0 */;</pre>
xc_alias	<p>更换XNF中的Cell名，Virtex系列的除外，语法如下：</p> <p>.sdc 文件中</p> <pre>define_attribute {cell_name} xc_alias {alias_name}</pre> <p>Verilog 中</p> <pre>object /* synthesis xc_alias "string" */;</pre>
xc_clockbuftype	<p>在Virtex设计中声明时钟端口使用时钟延迟锁相环CLKDLL，其语法如下：</p> <p>.sdc 文件中</p> <pre>define_attribute {port} xc_clockbuftype {BUFGDLL}</pre> <p>Verilog 中</p> <pre>object /* synthesis xc_clockbuftype = "BUFGDLL" */;</pre>
xc_fast	<p>加速XC4000系列设计中输出管脚的转换速度，缺省的输出管脚的转换速度是低。Synplify可以将属性传递到供布局布线工具使用的XNF或EDIF文件中，xc_fast就是可以被传递的属性之一，其语法如下：</p> <p>.sdc 文件中</p> <pre>define_attribute {output_port} xc_fast {1}</pre> <p>Verilog 中</p> <pre>object /* synthesis xc_fast = 0 1 */;</pre>

xc_fast_auto	<p>控制Virtex设计中高速输出Buffer的实例化。缺省值是1，高速输出缓冲OBUF_F_24、OBUFT_F_24和IOBUF_F_24将被用于实例化。在HDL源代码，在顶级Module中声明这一属性。可以用xc_padtype覆盖它。其语法如下：</p> <p>.sdc 文件中</p> <pre>define_global_attribute xc_fast_auto {0 1}</pre> <p>Verilog 中</p> <pre>object /* synthesis xc_fast_auto = 0 1 */;</pre>
xc_global_buffers	<p>控制在设计中使用的全局Buffer的数量，如若在黑盒中使用了全局Buffer，可以用这个属性防止使用的全局Buffer数量超过了资源限度，这个属性不等用于源代码中，其语法如下：</p> <p>.sdc 文件中</p> <pre>define_global_attribute xc_global_buffers {maximum}</pre>
xc_isgsr	<p>在非Virtex设计中，声明一个黑盒的端口和内部STARTUP块相连，防止Synplify再生成一个STARTUP块，其语法如下：</p> <p>.sdc 文件中</p> <pre>define_attribute {instance.reset_port} xc_isgsr {1 0}</pre> <p>Verilog 中</p> <pre>object /* synthesis xc_isgsr = {1 0} */;</pre>
xc_loc	<p>声明端口或设计单元的位置，关于有效的位置参见Xilinx databook。可以用xc_loc声明向量总线（端口）的片段的位置。其语法如下：</p> <p>.sdc 文件中</p> <pre>define_attribute {port_design_name} xc_loc {placements}</pre> <p>Verilog 中</p> <pre>object /* synthesis xc_loc = "placements" */;</pre>
xc_map	<p>在设计中声明一个设计单元使用fmap、hmap或LUT，适用于XC4000和Virtex系列，只能和xc_uset、xc_rloc一起使用，参见xc_uset和xc_rloc。其语法如下：</p> <p>.sdc 文件中</p> <pre>define_attribute {v:primitive_name} xc_map {fmap hmap lut}</pre> <p>Verilog 中</p> <pre>object /* synthesis xc_map = "fmap hmap lut" */;</pre>
xc_modular_region	<p>声明一个Module在FPGA上的物理位置，用于在模块化流程中综合一个单独的Module。用行数和列数作为属性值，在Virtex2设计中使用关键字SLICE，在Virtex设计中使用关键字CLB。语法如下：</p> <p>.sdc 文件中</p> <pre>define_attribute xc_modular_region {CLB SLICE_R<n>C<n>:CLB SLICE_R<n>C<n>}</pre> <p>Verilog中</p> <pre>object /* xc_modular_region= "CLB SLICE_R<n>C<n>:CLB SLICE_R<n>C<n>" */;</pre>
xc_ncf_auto_relax	<p>控制生成*.ncf文件时对约束的自动放松，只适用于XC4000、XC5200和Virtex系列，缺省值是1，即自动放松。其语法如下：</p> <p>.sdc 文件中</p> <pre>define_global_attribute xc_ncf_auto_relax { 0 1}</pre> <p>Verilog 中</p> <pre>object /* synthesis xc_ncf_auto_relax = 0 1 */;</pre>
xc_nodelay	<p>删除Xilinx在XC4000系列中为触发器和锁存器插入的输入延迟，该输入延迟的插入是为了保证XC4000系列中INFF、INLAT和INREG要求的保持时间。</p> <p>.sdc 文件中</p> <pre>define_attribute {input_port_name} xc_nodelay {1 0}</pre>

xc_padtype	<p>在Virtex设计中声明I/O buffer的标准。 buffer的可选值如下: BUFGP IBUFG IBUF IOBUF OBUFT OBUF 标准的可选值如下: AGP CTT F_2 F_4 F_6 F_8 F_12 F_16 F_24 GTL GTLP HSTL_I HSTL_III HSTL_IV LVCMOS2 PCI33_3 PCI33_5 PCI66_3 S_2 S_4 S_6 S_8 S_12 S_16 S_24 SSTL2_I SSTL2_II SSTL3_I SSTL3_II 其语法如下: .sdc 文件中 define_attribute {port} xc_padtype {buffers_standards} Verilog 中 object /* synthesis xc_padtype = "buffers_standards" */;</p>
xc_props	<p>声明标注到门级网表的Xilinx属性, 其语法如下: .sdc 文件中 define_attribute {i:instance} xc_props {property=value} For example: define_attribute {i:RAM1} xc_props {INIT=FFFF} Verilog 中 object /* synthesis xc_props = "property=value" */; For example: RAM16X1S RAM1(...)/* synthesis xc_props="INIT=0000" */;</p>
xc_pseudo_locs	<p>在模块化流程中用于声明管脚位置</p>
xc_pullup/xc_pulldown	<p>声明端口是上拉或下拉, 适用于XC4000、XC5200和Virtex, 其语法如下: .sdc 文件 define_attribute {port_name} xc_pullup {1} define_attribute {port_name} xc_pulldown {1} Verilog 中 object /* synthesis xc_pulldown = 1 0 */;</p>
xc_rloc	<p>指定所有xc_uset 属性值相同的实例的相对位置, 适用于XC4000和Virtex系列, 参见xc_uset 和xc_map。 .sdc 文件中 define_attribute {design_name} xc_rloc {instance_name} Verilog 中 object /* synthesis xc_rloc = "instance_name" */;</p>
xc_slow	<p>指定一个输出端口为低转换速度驱动, 适用于XC4000, 这一属性可以被传递到XNF或EDIF文件中, 其语法如下: .sdc 文件中 define_attribute {output_port} xc_slow {1 0} Verilog 中 object /* synthesis xc_slow = 1 0 */;</p>
xc_uset	<p>分配给component实例一个组名, 参见xc_rloc和xc_map。其语法如下: .sdc 文件中 define_attribute {instance_name} xc_uset {group_name} Verilog 中 object /* synthesis xc_uset = "group_name" */;</p>

表5 Xilinx支持的内存类型

器件	属性值	实现方法
XC4000系列	registers	RAMs由寄存器实现
	select_ ram	缺省情况: RAMs由分布RAMs实现
Virtex 系列	registers	RAMs 由寄存器实现
	block_ ram & no_ rw_ check	映射到 Block SelectRAM+
	select_ ram	缺省情况: RAMs由分布RAMs实现

8.2.1 综合指示 (Directives)

black_box_pad_pin

声明用户定义的黑盒的管脚，作为外部环境可见的I/O pad，如果有不止一个端口，列在双引号内，以逗号分开。一般不需要这一属性，Synplify提供了预定义的I/Os。其语法如下

```
object /* synthesis syn_black_box black_box_pad_pin = "port_list" */;
```

例如:

```
module BS(D,IN,PAD,Q) /*synthesis syn_black_box black_box_pad_pin="PAD" */;
```

block_box_tri_pins

声明黑盒的一个输出端口是三态，如不止一个列在双引号内以逗号分开。其语法如下:

```
object /* synthesis syn_black_box black_box_tri_pins = "port_list" */;
```

例如:

```
module BBDLHS(D,E,GIN,GOUT,PAD,Q) /* synthesis syn_black_box  
black_box_tri_pins="PAD" */;
```

full_case

仅用于Verilog，与case 语句一起使用，表明所有可能的状态都已经给出，不需要其他逻辑保持信号的值。语法如下:

```
object /* synthesis full_case */
```

其中object可以是case、casex、 casez statements declaration

parallel_case

仅用于Verilog，与case 语句一起使用，强制生成一个并行的多路选择结构而不是一个优先译码结构。其语法:

```
object /* synthesis parallel_case */
```

其中object可以是case、casex、 casez statements declaration

syn_block_box

说明一个module或component为黑盒，仅利用其界面进行综合而不管是否内部为空，也不进行优化。一般应用于厂家原语或宏，或IP等用户定义的宏。对于厂家I/O或其他一些厂家的宏，通常不需此属性，Synplify提供预定义的黑盒。其语法如下:

```
object /* synthesis syn_black_box */;
```

其中object可以是module declaration.

syn_encoding

强制选择自动机实现的方式，其可选值如下：

default—综合根据状态的数量选择编码方式，编码方式可以是：onehot、gray、sequential

onehot—采用onehot编码方式

gray—采用格雷码

sequential—采用自然码

safe—如果不能到达任一个状态时，让其回到复位态

其语法如下：

```
object /* synthesis syn_encoding = "value" */;
```

其中object是状态寄存器定义

syn_isclock

说明黑盒的一个输入是时钟信号。对名字为clk、rclk、wclk的黑盒输入信号，软件自动当作时钟，可以用这个属性说明任意输入信号为时钟信号。其语法如下：

```
object /* synthesis syn_isclock = 0|1 */;
```

其中object是黑盒的input port

例如：

```
module ram4(myclk, out, opcode, a, b) /* synthesis syn_black_box */;
output [7:0] out;
input myclk /* synthesis syn_isclock = 1 */;
input [2:0] opcode;
input [7:0] a, b;
/* Other coding */
```

syn_keep

保证被指定的wire在综合中保持不动，不会被优化掉。用于在define_multicycle_path或define_false_path用了-through 选项。如果你使用了这一属性，将生成一个keepbuf，可以对其定义时间约束，这个Buffer只占用一个位置，不出现在门级网表里。其语法如下：

```
object /* synthesis syn_keep = 0|1 */;
```

其中object是wire或reg声明

syn_noprune

用来保持一个或多个component的实例，即使其输出不能映射。一般无此指示的情况下，有着未用输出端口的实例会从EDIF文件中删除。它可被置于约束文件中，其语法如下：

.sdc 文件中

```
define_attribute {module|instance} syn_noprune {0|1}
```

Verilog 中

```
object /* synthesis syn_noprune = 0|1 */;
```

其中object可以是module declaration也可以是实例。

syn_preserve

用在某些独立的寄存器上，或module（则相当于施加于module中的所有寄存器）上，使其在优化时保持不动。也可用于保持某个自动机在优化时不动。其语法如下：

```
object /* synthesis syn_preserve = 0|1 */;
```

其中object可以是寄存器定义信号，也可以是Module

syn_sharing

使能/禁止综合时对运算符进行资源共享。缺省值是off，也可以在project视窗里设置这一选项。其语法如下：

```
object /* synthesis syn_sharing = " on|off " */;
```

其中object可以是module定义语句。

syn_state_machine

使能对设计中的某组状态寄存器进行自动机优化。其语法如下：

```
object /* synthesis syn_state_machine = 0|1 */;
```

其中object是该组状态寄存器

syn_tco<n>

提供黑盒的输出延迟信息，参见8.2。语法如下：

```
object /* syn_tcon = "[!]clock -> bundle = value" */;
```

其中bundle是总线或标量信号的集合

syn_tpd<n>

提供穿过黑盒的组合逻辑的传输延迟信息，参见8.2。语法如下：

```
object /* syn_tpdn = "[!]clock -> bundle = value" */;
```

其中bundle是总线或标量信号的集合

syn_tristate

指定黑盒的一个输出端口为三态。其语法如下：

```
object /* synthesis syn_tristate = 0|1 */;
```

其中object可以是黑盒的output port

syn_tsu<n>

说明一个黑盒的输入要求的建立时间。参见8.2。其语法如下：

```
object /* syn_tsun = "[!]clock -> bundle = value" */;
```

translate_on/translate_off

用于与其他综合软件的兼容。在这两个指示中间的所有代码将在综合时被忽略。也可以用于在源代码中插入一段仿真代码。其语法如下：

```
/* synthesis translate_off */
```

综合时忽略的代码。。。

```
/* synthesis translate_on */
```

9 实现对速度的优化

9.1 一般性考虑

在不设置任何约束的情况下，进行第一次综合，然后

第一步：用view log查看Timing信息。如果与设计目标差距在5%到10%。可先布局布线看是否可以满足要求，再用define_reg_input_delay -route, define_reg_output_delay -route等约束指导重新综合。如果与设计目标差距在10%到20%之间，可以考虑下面的方法。如在20%以上的話，一般就需要考虑修改设计或是选用更高速的器件了。

第二步：在工程窗口中设置synplify频率设置而不是采用隐含值0MHz

第三步：如果设计中存在多个时钟，可在约束文件中用define_clock定义多个时钟，以覆盖在工程窗口中设置的工作频率。

第四步：用约束文件适当减少相关路径的fan_out数目可加快该路径工作速度。

第五步：如果在设计中用了有限状态机，在约束文件中打开symbolic FSM Compiler option。

第六步：用HDL Analyst查看综合结果，找到关键路径，在相应的源代码中加入约束和属性：
define_reg_input_delay -improve, define_reg_output_delay -improve, define_input_delay, define_output_delay等。对黑匣子进行时间属性约束：syn_tpd, syn_tsu, syn_tco等。

9.2 怎样处理关键路径上不满足速度要求的延迟

1，关键路径的起点不是输入端就是寄存器，而结束点不是输出端就是寄存器。如果起点少就选择起点添加时间约束和时间属性。如果结束点较少就选择结束点添加时间约束和时间属性。

2，对输入端口添加约束。如果关键路径的起点是输入端口，而且该路径的slack时间是-2.4ns，则可在输入端添加约束define_input_delay -improve 2.4。

3，如果关键路径的起点是寄存器，需对该寄存器添加时间约束和时间属性。例如：
define_reg_output_delay -improve 2.4。

4，如果关键路径的终点是输出端口，则可对该该端口添加时间约束：define_output_delay 2.4

5，如果关键路径的终点是寄存器，则可添加时间约束：define_reg_input_delay 2.4。

6，对添加了约束的设计重新综合，查看延迟信息，如果时间延迟的超出部分在设计目标的5%到10%之间，则可考虑布局布线，并考虑在布局布线时添加有相应的供应商提供的约束条件。

9.3 关于综合约束的建议

设置恰如其分的约束，是获得最佳综合结果的关键。过严或是过松的约束都达不到最佳的效果。一般可先尝试通用的约束，如时钟、扇出限制等。如果没有达到要求可加入一些严格的具体约束，同时注意放松一些可以放松的约束。综合约束的结果是估计值，应该以布局布线的结果为准。Synplify可以生成布局布线工具使用的约束文件，可以充分利用这一功能。

9.3.1 时钟

采用同步设计，使用尽可能少的时钟。可以先尝试使用缺省时钟进行综合，如在project视窗或在project文件或Tcl 脚本文件中用set option -frequency设定，缺省时钟等于最快的时钟，若不能达到要求再使用 define_clock、 define_clock_delay 详细设置所有时钟，同时使用 define_multicycle_path、 define_false_path、 syn_reference_clock 放松可以放松的约束，使得综合工具可以充分让关键路径优先使用资源。

9.3.2 扇出限制

根据所选器件确定一个恰当的扇出限制，扇出过大会使延迟加大，扇出过小又会使某些高扇出网络增加过多的缓冲而造成延迟加大。对于高扇出网络，综合工具可以通过复制数据源或插入缓冲降低扇出，复制数据源的数量和插入的缓冲数量都会列在log文件中。最好通过修改设计是负载平衡，对于速度要求高又无法平衡的网络可以考虑手工插入全局时钟缓冲作为高扇出缓冲。