摘要：本文介绍了基于标准单元库的深亚微米数字集成电路的自动化设计流程。此流程从设计的系统行为级描述或 RTL 级描述开始，依次通过系统行为级的功能验证，设计综合，综合后仿真，自动化布局布线，到最后的版图后仿真。在这里，我们用 Synopsys 公司的 VSS(VHDL System Simulator)工具进行各种仿真，用 Design Compiler 进行综合，用 Cadence 公司的 Silicon Ensemble 进行自动布局布线。对于最后的版图后仿真，由于输出文件的限制，我们改用 Active-HDL 工具进行验证。本文同时用一个实例 DDFS 对整个流程加以了举例说明。
关键词：库，仿真，约束，综合， floorplan,布局布线。

前 言

传统的芯片设计方法是手工全定制。随着半导体工艺的几何缩小，集成电路设计已经到了深亚微米的时代。在同一面积上，晶体管数目的迅猛增加，传统的芯片设计方已几乎变得不可能。再加上 time-to-market 的压力越来越大，用户要求芯片制造商在最短的时间内用最低的费用生产高性能产品。为了解决这些问题，新的方法学和工具得到了发展。近几年来，为了缓解 time-to-market 的压力和快速更新设计以满足用户的要求，一些高性能的工具和技术得到了发展。高级设计语言的应用，如 VHDL ， Verilog ，取代了手画电路图,并且提高了设计重用。技术更改指令 ECO(Engineering Change Orders)技术的发展更进一步的提高了设计重用。 Formal Verification 代替动态仿真，不仅提高了验证速度，更重要的是它摆脱了工艺的约束和仿真 test bench 的不完全性，更全面的检查了电路的功能。从行为级开始综合大大提高了设计者的设计灵活性，使设计者更进一步脱离了工艺与物理的限制。设计预算方法学的发展使设计者在较少的时间内得到了较好的 QOR ，并且提供了更好的环境约束。模块编译器简化和自动化了 data-path 设计，帮设计者解决复杂而没有规则的 data-path 设计。自动布局布线提高了版图生成的效率，减少了过多的人工干预所带来的不确定性。版图提取和分析加强了逻辑设计与物理设计之间的联系与信息交换，更进一步提高了逻辑综合时对版图的考虑。设计重用技术，验证技术，行为综合和逻辑综合，设计预算技术，模块编译技术，布局布线自动化，版图提取和分析等技术的应用大大提高了设计人员的设计能力，缩短了设计周期。

本文讲述的是基于标准单元库的数字集成电路的设计流程和方法学。它从行为级的 HDL 描述开始，依次进行系统行为级仿真， RTL 级仿真，逻辑综合，综合后仿真，自动化布局布线，最后是版图后仿真。所有这些步骤都是通过工具自动完成，快速而有效。

我用 Synopsys 公司的 VSS(VHDL System Simulator)工具进行各种仿真，用 Design Compiler 进行综合，用 Cadence 公司的 Silicon Ensemble 进行自动布局布线。对于最后的版图后仿真，由于输出文件的限制，我们改用 Active-HDL 工具进行验证。并且解决 clock tree 和版图后仿真的问题。本文用 DDFS ， I2C ， counter 等实例对整个流程加以了验证。

本文的第 1 章简要介绍了深亚微米数字集成电路的设计流程。从第 2 章开始我们将分章节详细介绍各个主要步骤。第 2 章介绍系统行为级仿真方法。第 3 章介绍行为级综合和模型编译。第 4 章解释了综合的概念，介绍了逻辑综合的实现及讨论了几个常见问题的解决方法。第 5 章解决了版图后仿真的实现问题，阐述了各种技术库的生成，比较了系统行为级仿真和综合后仿真的区别。第 6 章介绍了 Formal Verification 和其他辅助工具的应用。第 7 章详细讲述了自动化布局布线方法，解决了 clock tree 的生成问题。由于版图后仿真与综合后仿真在操作上没什么区别，这里就略去不讲。

第 1 章　　EDA 设计的概述

随着电路设计进入 VLSI ，甚至 ULSI 时代，电路规模迅速上升到几十万门甚至几百万门。根据摩尔定律，每十八个月增加一倍。而设计人员的设计能力只是一个线性增长的曲线，远远跟不上电路规模指数上升的速度。为了弥补这个差距，工业界对 EDA 软件和设计方法不断提出新的要求。在 80 年代，由美国国防部支持的 Very High Speed Integrated Circiut 发展计划促成了 VHDL 的诞生，并使之成为了国际标准。而 Cadence 公司的 Verilog HDL 在工业界获得了广泛的接受，并最终成为了国际标准。利用 HDL 进行设计大大方便了设计输入，提高了设计抽象程度，更有利于设计人员发挥聪明才智，因而可以大大提高设计效率，缩短了设计周期。

随着电路规模的增大和系统复杂度的增加，直接用电路实现已是不可能， RTL 级的 HDL 编码也变得越来越难以忍受。行为级综合技术的发展为设计者带来了曙光。它使设计者开始逐步摆脱繁重的 RTL 级编码，大大提高了设计者的设计灵活性和设计效率，减少了工艺及物理对设计的约束。

为了提高设计的速度和设计成功率，利用已验证正确的设计作为新设计的一部分是现在大规模设计的常用方法。随着时代的发展，人们对产品的要求越来越高。他们要求的不仅仅是新产品的出现，更多的是要求改善旧产品的性能，增加更多的功能。为此对旧的设计的修改是必须的。为了充分利用以前的成果，减少修改的工作量，加快设计修改速度，同时尽量不影响不变部分，提高修改的成功率，技术更改指令 ECO 被提了出来并得到了发展。

随着半导体工艺的不断进步，器件的特征尺寸越来越小，线宽越来越窄，器件的速度变得越来越快。但同时随着设计的越来越复杂，电路规模的越来越大，金属线的长度和层数不断增加，线宽也随之变小。这都导致了金属连线的延时变大。于是器件的延时不再是一个系统的主要延时，连线的延时变得越来越重要，甚至超过了器件的延时。因此以前设计系统时只考虑器件延时的观念已经行不通，设计时考虑连线的延时是必须的。设计者在设计时必须同时考虑到综合和版图，且使综合和版图尽量结合在一起。把综合后的时序信息前注释到布局布线，同时布局布线后提取寄生参数和时序延时信息后注释回综合，从而使逻辑设计和物理设计紧密的结合起来。考虑到连线延时，必须进行版图后仿真。版图后仿真必须后注释大量的版图时序延时信息。

电路规模的增大导致了时钟同步的问题。时钟到达不同子模块的延时不同，这成了一个系统失败的致命弱点。为了解决时钟延时的问题，在布局布线中 CLOCK TREE 的技术得到了极大的发展。它较好的解决了这时钟延时的问题。

随着系统规模的不断增大，功耗的问题变得越来越重要，散热成了人们的一大难题。为此，设计者在进行设计系统的时候必须考虑功耗的问题。在逻辑综合后必须进行功耗分析。

- 设计流程

基于标准单元库的数字集成电路设计方法主要流程为及工具使用如下：

1. 功能与规格要求；
2. 行为级编码，仿真 test bench 的准备及 DFT 存储器的 BIST 插入；
3. 用 VSS 进行系统行为级的功能验证；
4. 用 Behavioral Compiler 进行行为级综合，生成 RTL 级网表；
5. 用 VSS 进行 RTl 级仿真；
6. 用 Design Compiler 进行初级综合；
7. 用 Design Budgeter 进行设计约束的分配；
8. 用 Design Compiler 进行逻辑综合与测试扫描插入；
9. 用 VSS 进行综合后时序功能验证；
10. 用 Design Compiler 或 Prime Time 版图前静态时序分析；
11. 用 Power Compiler 进行功耗分析；
12. 用 Silicon Ensemble 进行 floorplan,布局,Clock Tree 的插入以及全局布线；
13. 插入 Clock Tree 后的网表重新读回 Design Compiler ；
14. 用 Formality 验证原来的综合后网表和插入 Clock Tree 后的网表；

15. 用 Prime Time 进行全局布线后静态时序分析；
16. 用 Silicon Ensemble 进行细节布线；
17. 用 Prime Timing 进行版图后静态时序分析；
18. 用 VCS 或其他门级电路的仿真器进行版图后时序功能验证；
19. 流片

大体的流程图如图 1.1 所示：

由于篇幅的限制，本文将只详细讲述几个重要的工具，其他工具只讲述其流程。

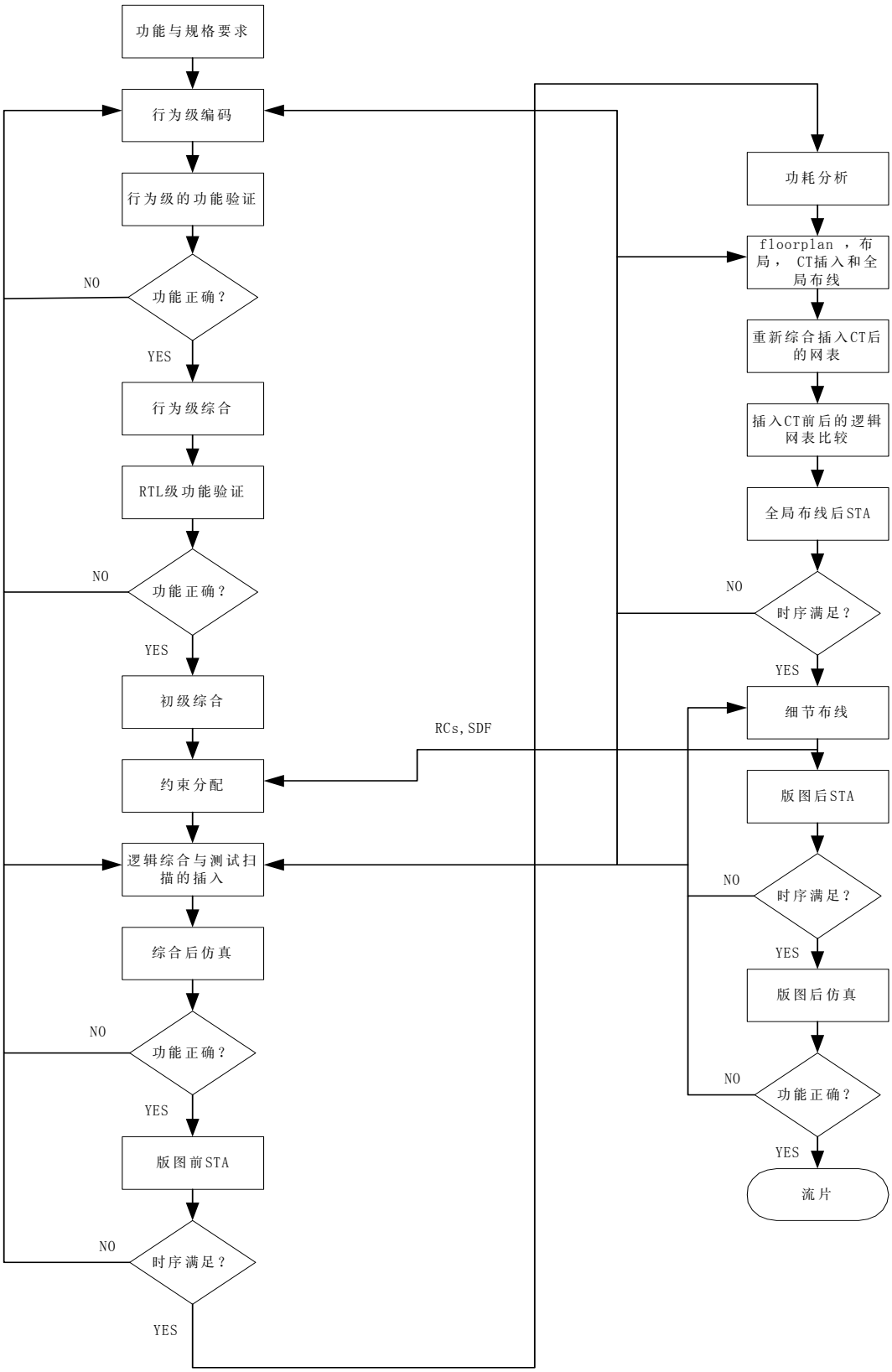


图 1.1 数字集成电路的设计流程

第 2 章　行为级仿真

2.1　行为级仿真简介

当设计完成后，为了验证功能是否正确，设计者必须对其设计源文件进行仿真。因为这时的设计文件为行为级的 HDL 文件，故称此仿真称为行为级仿真。当设计源文件经过行为综合，或手工编写，转换为 RTL 级设计文件后，设计者还必须进行 RTL 级仿真。因为 RTL 级仿真与行为级仿真在具体操作上没什么区别，本文将不再讲述 RTL 级仿真。

2.1.1　工具介绍：

synopsys 提供了数个仿真工具：**Scirocco 、 VHDL 仿真工具、 Verilog 仿真工具**。不同的仿真工具有不同用途和各自的优点。

1.　Scirocco

Scirocco 为 RTLO 级功能验证提供最快最高性能的 VHDL 仿真。 Scirocco 既支持基于周期(cycle-based)的仿真也支持事件驱动(event-driven)的仿真。 Scirocco 使基于周期(cycle-based)的仿真有着事件驱动(event-driven)仿真的灵活性。这个技术为综合的设计优化提供了最佳性能。它支持混合语言仿真。 Scirocco 支持各级的设计描述，但只对行为级和寄存器级进行优化。 Scirocco 支持后仿真机制。支持多语言，多平台，多仿真器。 Scirocco 有强大的纠错能力。

后仿真机制：就是通过把 VCD(a Value Change Dump)历史文件作为输入，对事件驱动仿真不再进行调试，而直接分析 VCD 文件里记录的仿真结果。

2.　VHDL 仿真工具

VHDL 仿真工具用于 Synopsys 高级设计学的功能验证阶段。它包括 Synopsys 系统仿真器(VSS)和 Cyclone 。 VSS 是一个事件驱动仿真器， Cyclone 是一个基于周期的仿真器。VSS 和 Cyclone 都可用于确认和验证寄存器级设计， VSS 还可用于验证门级设计。 VHDL 仿真器工具顾名思义只能对 VHDL 设计进行仿真，但是它可以产生 Verilog 目标文件，使其可用于 VCS(Verilog Compiled Simulator) 仿真器。同样对于 Verilog 设计，设计者也可通过 VCS 仿真器产生 VHDL 目标文件,使其用于 VSS 仿真器。这样就解决了混合语言仿真的问题。

基于周期仿真器：只在每个时钟的有效沿计算设计源代码的值，而对每个周期内的其他时序信息不与考虑，即对时钟周期间的事件不进行仿真。信号在时钟沿是被假设为稳定的，因此建立时间，保持时间，脉冲宽度的违约现象都被忽略。虽然两个时钟沿间的信号和时序信息被忽略了，但是它大大加快了仿真的速度和俭省了内存空间。对复杂的大型设计，这种仿真机制有着很大的优势。

事件驱动仿真器：在单个周期内，信号值在最后稳定之前，于逻辑路径的任何一点都可能改变数次。每当某个事件发生，仿真器就对信号值更新一次。仿真器在整个时钟周期都描绘出所有信号的值。

基于周期仿真和事件驱动仿真的区别如图 2.1 所示：

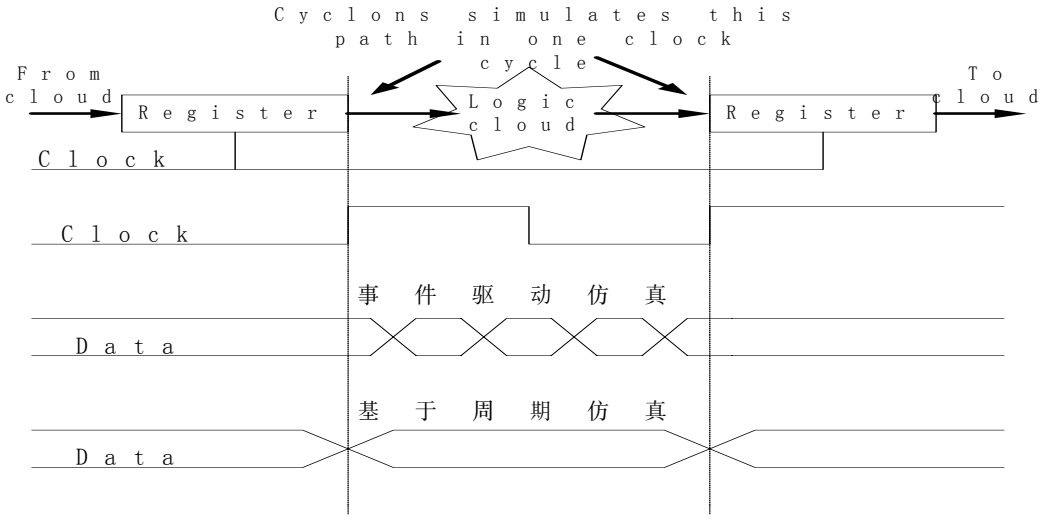


图 2.1 事件驱动与基于周期仿真的区别

1) **VSS 介绍**

VSS 是一种贯穿从概念到 ASIC sign off 的功能强大的仿真器。它可以用于仿真和调试综合前和综合后行为级，寄存器级和门级描述的 VHDL 设计。它是一个复杂的事件驱动仿真器。虽然高级设计的三个设计发展阶段：概念确认阶段，功能验证阶段，执行验证阶段，VSS 都可以使用，但是它更适合于概念确认阶段和执行验证阶段(门级仿真)。

它有三种仿真机制：

Interpreted : 具有充分的内嵌交互调试特性
Compiled : 具有最快的 RTL 和行为级仿真速度
Gate-level : 快速门级仿真和 ASIC sign off 验证

VSS 核心程序： VHDL 分析器(vhdl -event)、 VHDL 库分析器(liban)、 VHDL 仿真器(vhdlsim)、 VHDL 调试器(vhdldbx)、波形观察器(waves)。

2) **Cyclone 介绍：**

Cyclone 是一个快而功能强大的基于周期的仿真器。它直接计算和仿真 RTL 级源代码，而不是在仿真前先把 RTL 级代码综合成门级电路。它使用简单的逻辑和强度值。它把 std_logic package 里定义的 9 状态逻辑转换成了 2 状态逻辑(0 ， 1 ， Z)或 3 状态逻辑(0 ， 1 ， X ， Z)。它映射 L ， H ， W 弱逻辑状态为 0 ， 1 ， X 强逻辑状态，减少了仿真时逻辑值的数目。它忽略了逻辑延时值。 Cyclone 有着比 VSS 更高的性能和更少的运行仿真时间。在功能验证阶段，设计模型较稳定，仿真次数较少(同概念确认阶段相比)。 Cyclone 就是为这种较稳定模型的的冗长测试优化的。而且它具有良好的直觉性和交互性的调试性能，方便了设计者在功能验证阶段修改其设计。 Cyclone 与 Synopsys 的综合工具紧密的结合在一起,它将影响 HDL 源代码,使其更适合于综合。因此在 RTL 功能验证阶段,建议使用 Cyclone 仿真器。

1. Verilog 仿真工具

Verilog 仿真工具的具体应用和 VHDL 仿真工具类似，只不过它只适用于 Verilog 源代码的仿真，而 VHDL 仿真器只适合于 VHDL 源代码的仿真。 Verilog 仿真工具的仿真器为 VCS(Verilog Compiled Simulator)。它的具体介绍略。

以上各种软件的使用大同小异，特别是仿真命令几乎没什么区别。其中有点需要注意的是： Scirocco 与其他仿真工具在流程上有点小区别。那就是 Scirocco 因为即支持基于周期的仿真又支持事件驱动的仿真，所以在它用混合模式分析源文件后，进行混合模式仿真前,需要产生分块文件给它进行分块处理。如果不对需要进行基于周期仿真的顶层设计用分块命令语句指定，它将以事件驱动模式运行。分块命令语句为：

cycle *[option] design_root*

(design_root 为顶层设计的 configuration-name)。

下面的介绍我们主要以 VHDL 仿真工具中的 VSS 为例。

2.1.2 系统行为级仿真流程(如图 2.2 所示)

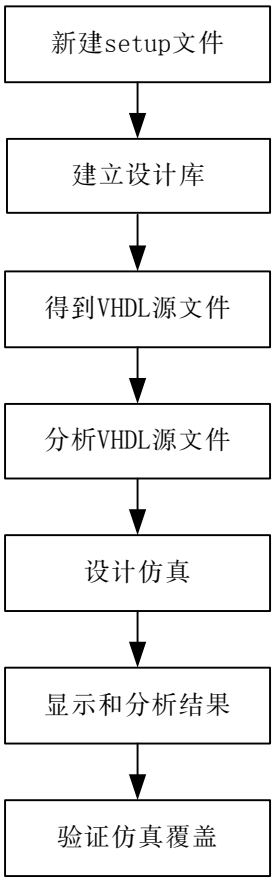


图 2.2 系统行为级仿真设计流程

1. 新建 setup 文件

VHDL 仿真工具的 setup 文件.synopsys_vss.setup 指定了你的 VHDL 设计库名与实际的 UNIX 目录的映射关系，建立了你的查找路径，并给仿真控制变量赋值。

2. 建立设计库

设计库用于存放源程序分析后的中间文件和设计所的 reference 库。

3. 得到 VHDL 源文件

4. 分析 VHDL 源文件

VHDL 分析器检查 VHDL 源文件的句法和语法错误。经检查没有错误的 VHDL 源文件转换为中间格式的文件存放于设计库中。

5. **设计仿真**

VHDL 仿真器根据设计库里的中间文件建立完整的层次结构，计算仿真值。

6. **显示和分析结果**

利用波形观察器观察和分析仿真结果。

7. **验证仿真的 coverage**

coverage 文件(.cov)列出了 VHDL 源文件每一行源代码的执行时间数。通过这些信息可以验证设计瓶颈和未被激励的面积。

2.2 setup 文件

仿真工具在每一次启动的时候都将读一次定义了环境变量的 setup 文件。VHDL 仿真工具有着三个同名的 setup 文件。一个为安装目录下的 setup 文件。它定义了缺省环境。一个为你的根目录下的 setup 文件，它定义了了你所有设计的共同 setup 信息。一个为你工

作目录下的 setup 文件，它定义了你这个设计的 setup 信息。当你启动仿真工具时它依次从你的安装目录，你的根目录，你的工作目录读入这三个 setup 文件。最后读的 setup 文件有最高的优先权，即相同的变量定义，后面读入的 setup 文件定义将覆盖前面的 setup 文件的定义。

如果想察看 setup 文件定义的变量可用命令 show_setup 列出所有变量。

2.2.1 Setup 文件的编辑

setup 文件的一个简单样本:

```
WORK > LIB1
LIB1 : user/design/lib1
TIMEBASE = ns
```

- **设计库的映射：**

```
library_logical_name > design_library_name
design_library_name : host_directory_name
```

library logical name：是指你在 VHDL 源文件中库语句所写的库名，既 LIBRARY，USE 语句中的库名。

design_library_name：是指仿真工具使用的中间库名。

host_directory_name：是指你的库所存放的实际目录 。

如：上华的库 csmc06 我们放在根目录的 csmchdlib 子目录下。我们任取一个有效的 design library name—libs，则设计库的映射写为：

```
csmc06 > libs
libs : /home/usr/csmchdlib
```

有关设计库映射的详细说明请见**建立设计库**那一节。

- **变量定义**

```
variable_name = value
如:TIMEBASE = ns
```

- **其他常用命令**

如果你的命令需要续行请用续行符号“\”

```
如： libs: /home/usr/ggh/sample_projects/example \
   /debugger/lib1
```

如果你要注释一个语句,请用注释符“- -“

```
如：--this is time base
```

注意：如果你想改变你的仿真精度，则修改你的 TIMEBASE 变量。

2.3 建立设计库

设计库用于存放设计分析后的中间文件，仿真器从设计库里取出中间文件仿真。

设计库还用于存放你的设计所引用的 reference 库， reference 库在你的 VHDL 源文件里用库语句说明(USE, LIBRARY)。

设计库有三个名字：a logical name, a library name, a physical name。

a logical name：是 VHDL 源文件里调用的库的名字，既 library， use 语句里指明的库名。

a library name：是仿真工具使用的中间库名，它映射库的 logical name 到 physical name。

a physical name：是你实际的主机上目录，它用于存放你的分析后的中间文件，或你的资源库。

它们的关系如图 2.3 所示：

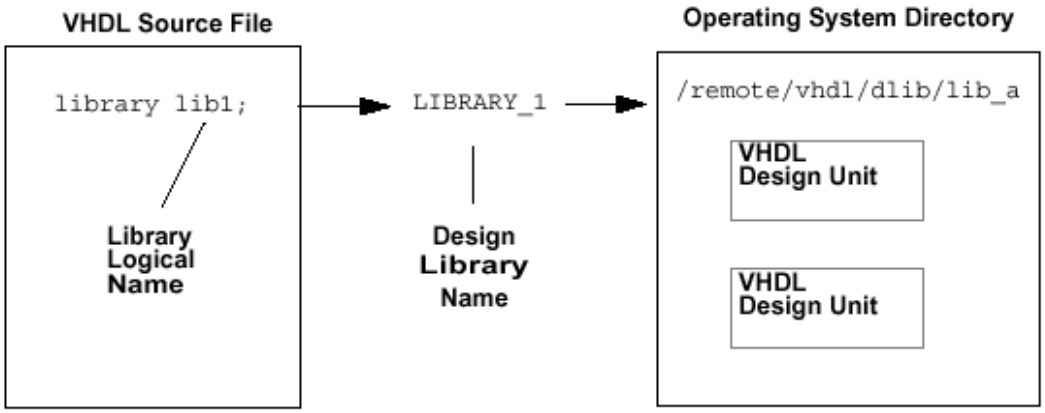


图 2.3 三个设计库的关系

设计库有两种,一种为存放仿真分析后的中间文件的工作库(work library),一种为存放你的设计调用的库的资源库(resource library)。

当你想改变你的 VHDL 源文件中调用的库时,我们可以通过改变 setup 文件中设计库的映射来方便的实现。既我们不用改变 VHDL 源文件中 LIBRARY 语句。而是在 setup 文件中更改设计库的 design library name 和 design physical name ，使其映射到新的目录，从而达到了改变 VHDL 源文件所调用的库的目的。

2.3.1 创建 work library

work library 的约定 design logical name 为 WORK ，当然我们也可在调用 VHDL 分析器的命令中利用-work 选项覆盖这个缺省的 design logical name 。我们任选一个合法的 design library name ，把 logical name 映射到这个名字：

WORK > design_library_name 。

我们再创建一个系统目录作为 design physical name ，在 setup 文件中把这系统目录映射到 design library name : design_library_name : design_physical_name 。这就完成了 work library 的创建。

2.3.2 创建 resource library

根据 VHDL 源文件中 LIBRARY 语句指定的库名，我们得到 design logical name 。如：LIBRARY csmc06 ，则这 csmc06 就是我们的 design logical name 。然后我们任意指定一个合法的 design library name, 在 setup 文件中把 design logical name 映射到 library name: design_logical_name > design_library_name 。再我们找到这资源库所存放的目录,把其作为我们的 design physical name ，在 setup 文件中映射： design_library_name : design_physical_name 。这就完成了 resource library 的创建。

2.3.3 特别注意:

a design library name 必须映射到一个存在的目录，而不是一个空目录，否则 VHDL 分析器在分析源文件的时候将报错，且停下来。

我们分析完源文件，在对中间文件进行仿真前，不能更改 setup 文件中设计库的映射关系，否则仿真器将会找不到你的设计。

一个 design library name 只能对应一个目录。如果你的设计库有多个名字。请用多个 design logical name 与之对应。

2.4 设计分析

调用 VHDL 分析器对设计源文件进行分析，我们可以使用 vhdlan 或 gvan 命令。

vhdlan 命令格式: vhdlan [options] filename_list

例如我们分析 mux.vhd 文件

%vhdlan mux.vhd

gvan 命令格式： gvan [options] filename_list

vhdlan 支持基于周期的仿真(-cycle)和事件驱动的仿真(-event)，但是-cycle 的分析必须是用 Cyclone 仿真器进行仿真。

gvan 命令调用 VHDL 分析器的图形界面,它把所有它检查到的错误都显示在错误浏览

窗口里。 gvan 命令不支持基于周期的仿真。

2.4.1 simdepends

当你的设计源代码或工作环境有所改变时，必须先重新分析这有改变的源文件，同时也要重新分析包含了这设计单元的其他源文件，对于一个庞大而复杂的设计来说，要找出这些有关联的源文件是一件非常困难的事。分析器的 simdepends 应用帮助你解决了这麻烦。 simdepends 列出了所有你相互之间有关联的文件列单，并且在重新分析源文件的同时帮你自动重新分析相关联的源文件。它的输出文件就是 UNIX 中的 makefile 文件。

运行 simdepends 产生依赖文件列单命令： simdepends [options] design_unit

注意：这里的 design_unit 必须是你的顶层设计单元，可以是 configuration ，package ， entity 。

simdepends 的使用步骤如下：

a) 用 vhdlan 或 gvan 分析你的设计: gvan *design1 design2 design3 ……*
b) 运行 simdepends 产生依赖列单: simdepends -o *makefile top_design_unit*
c) 仿真你的设计:
d) 修改你的设计:
e) 使用 UNIX 的 make 命令重新分析你的源文件和相关的源文件:
   make ANALYZER=gvan all

2.4.2 simcompiled

如果你是为 compiled 模式仿真分析源文件，且想知道将来哪个 compiled 选项在这个设计单元中使用，则在这个 design unit 上调用 simcompiled,它将为你提供这方面的 compiled 仿真消息。命令为：

simcompiled [options] design_unit [-u design_unit]

2.5 设计的仿真与结果分析

2.5.1 仿真机制

VSS 提供三种仿真机制：

Interpreted:

拥有强大的调试性能，在仿真过程中可以设置 monitors 和断点。它的调试可以是源代码级的，例如它可以监测变量和设计模块(block)中的源代码。

Compiled: 利用 compiled 模块最大化你的仿真速度。但是它只支持少数的调试功能，例如不能设置断点。

优化了的 FTGS 门级(Gate-Level)：

精确的门级仿真，优化了的 FTGS 门级仿真机制利用 ASIC 供应商的门级模型，精确的仿真设计中的物理器件。

在设计的早期，设计者需经常修改设计，建议使用 Interpreted 机制。当设计稳定下来以后，为加快仿真速度，建议使用 Compiled 机制。当然也可使用混合仿真机制，对未稳定的设计部分使用 Interpreted 机制，稳定了的设计部分使用 Compiled 机制。对于综合后的门级仿真和 ASIC 的完成验证当然是使用 Gate-Level 机制。

何时使用何种仿真机制， VSS 根据你的设计文件自动调用。例如目标文件(.o)VSS 自动调用 Compiled 机制，但是你也可以利用屏蔽调试功能对已编译设计强制使用 Interpreted 机制来进行进一步的调试。要使用屏蔽调试功能在启动 VSS 时加上-fi 或者-fi_all 选项即可。

2.5.2 VSS 的启动

启动 VSS 可以有两中方法，一种是 vhdlsim ，使用命令行界面；一种是 vhdldbx,使用图形窗口界面。

调用 vhdldbx: %vhdldbx *[options] design* 或%vhdldbx &再在 Vhdldbx-Select Simulator Arguments 窗口中选择仿真单元。

调用 vhdlsim: %vhdlsim [*options] design* 显示命令行提示符 #

design 可以是[*libname]cfgname*,也可以是*[libname] entname,[libname] entname archname* 。较常用的是*[libname] cfgname* 。

options 中有很多的选项和 setup 文件中定义的变量功能是相同的， options 中的变量优先级高于 setup 文件中的变量，即 options 的定义将覆盖 setup 文件中的变量定义。

常用 options:

-i *filename*　启动时调用命令文件 *filename*

-e *filename*　启动时调用命令文件，同时显示执行的命令

-sdf f*ilename.sdf*　读入 SDF 文件。

2.5.3 常用的命令行命令：

run [*n*]　运行 *n* 个时间单位

trace [*options*] *object_name_list*　绘出 *name* 的波形在波形观察器里

ls *[name]*　列出 name 匹配的对象

cd region　改变当前工作域

quit　结束仿真

restart　重启

include *filename*　调用命令文件 *filename*

help topic　特定主题和命令的帮助

assign (*VHDL_expression) {VHDL_object_name}*　给对象赋值

evaluate *vhdl_expression_list*　计算并显示表达式的值

statue [-t] {*process_name/dignal_name*}　列出准备运行或等待事件发生的进程

调用命令文件的方法有四：

include *filename*

或% vhdlsim –i *filename design_unit*

或% vhdlsim –e *filename design_unit*　/*执行时显示命令*/

或在. Synopsys_vss.setup 文件中设置： RUNREAD = *filename* VSS ，则每次调用都执行这个文件。如果调用 VSS 时使用－ i filename ， VSS 先执行 RUNREAD 指定的文件，再执行-i filename 指定的文件。

得到 configuration informations:　# environment > sim_config

!show_setup >> sim_config

暂时逃到操作系统执行数条命令方法：

```
#!
First command
Second command
.
.
last command
exit
#
```

纠错：# help vss-error_number

得到变量的值： $ variable

显示变量的值： echo $variable

设置变量的值： set　如： set PROMPT ^> ^

产生用户定义的变量：如： set MY_REGION　/CPU/alu/addwe/nand2

产生波形文件 .ow),在setup文件中设置： WAVEFORM=wif,同时调用Waveform Viewer则设置为： WAVEFORM=wif+waves

产生命令过程：　# comm *name*

```
>first command
> .
> .
> end
#
```

运行的话，只需打入过程的名字就行了。

产生监测(monitors)：　monitor [options] condition

编辑 monitors: #edit *monitor_name*

重导 monitor 的输出： redirect *device_tag monitor_list*

coverage -- 计算和显示在 interpreted simulation 中执行的 VHDL 源代码的每一行的时间数

#coverage [*options] [vhdl_source_file_list]* /*产生*/

% coverage *[options] {filename}* /*调用*/

VCD(a Value Change Dump) 文件：包含有时间量程，范围定义，堆放了的时间类型和随着时间的增加实际值的改变等信息的 ASCII 文件。

产生常规的 VCD file:

启动 vhdlsim ,

执行 vcdfiel vcdcomment,vcdaddobjectsvcddumpobjects 和其他 SCL 命令。

产生扩展的 VCD fi 文件同上。 SCL 命令不同： vcdaddports 等

也可同时产生这两个文件，用 vcdon ,vcdoff,vcdlimit 等

使用 WIF 文件作为激励文件：

% vhdlsim –iw *input_wif_filename.ow design_name*

产生 SAIF(a Switching Activity Interchange Format) 文件 :

% vhdlsim –saiffile *alarm_clock_saif cfg_tb_top_behavioral*

2.6 设计实例

对于 I2C ， DDFS ， couter 等涉及来说，操作的方法相同，由于篇幅的限制，本文只介绍 DDFS 设计。

实例： DDFS 设计源文件:

ddfs.vhd,froma.vhd,fromb.vhd,croma.vhd,cromb.vhd(注意这里的 VHDL 源文件不要忘了加 configuration 。)

testbench 文件为 DDFS_TB (源文件见附录)

1. 仿真前准备

转到工作目录，设工作目录为/home/usr/design

```
% cd  /home/usr/design
% mkdir vhdl
```

把设计源文件存放在 vhdl/目录下

2. 创建 setup 文件

在工作目录下创建 setup 文件， setup 文件的内容如下：

```
WORK > DEFAULT
DEFAULT : work
TIMEBASE = ps
```

3. 创建设计库

```
% mkdir work
```

4. 设计分析

产生设计分析的命令文件 analyze1.sh,内容为：

```
#!/bin/sh -f
      vhdlan -event  \
       vhdl/ddfs.vhd \
       vhdl/froma.vhd \
       vhdl/fromb.vhd \
       vhdl/croma.vhd \
       vhdl/cromb.vhd \
       vhdl/DDFS_TB.vhd
```

注；更好的命令文件是把创建设计库也写在里面，其内容见附录。

执行命令文件,对源文件进行分析:

```
% analyze1.sh
```

5. 设计调用仿真器进行设计仿真

产生仿真命令文件 simfile ，其最基本的内容为：

```
trace -wif -waves /DDFS__TB/*'signal
run 10000000
```

注：在这命令文件里，如果你想观察波形文件的话，切记不要写上 quit ，因为退出 vhdlsim 后，波形观察器同时也是关闭的。

调用 vhdlsim 仿真器进行仿真；

```
% vhdlsim -i simfile TESTBENCH_FOR_DDFS
```

注： TESTBENCH_FOR_DDFS 为 testbench 文件的 configuration 的名字。

仿真结果如图 2.4 所示:

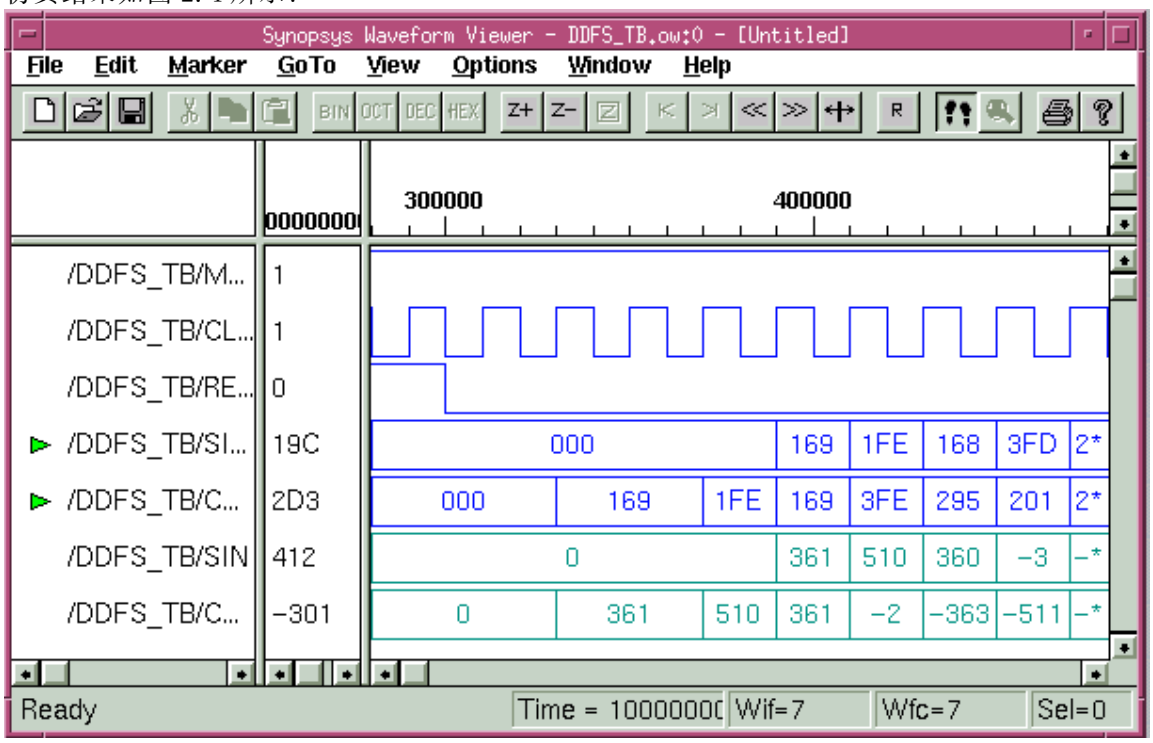


图 2.4 行为级仿真结果

第三章 Behavioral Compiler 和 Module Compiler

3.1 Behavioral Compiler

Behavioral Compiler 自动把行为级的 HDL 设计综合成了 RTL 级设计，提高了设计者的设计效率。

3.1.1 Behavioral Compiler 的设计流程图(如图 3.1 所示)

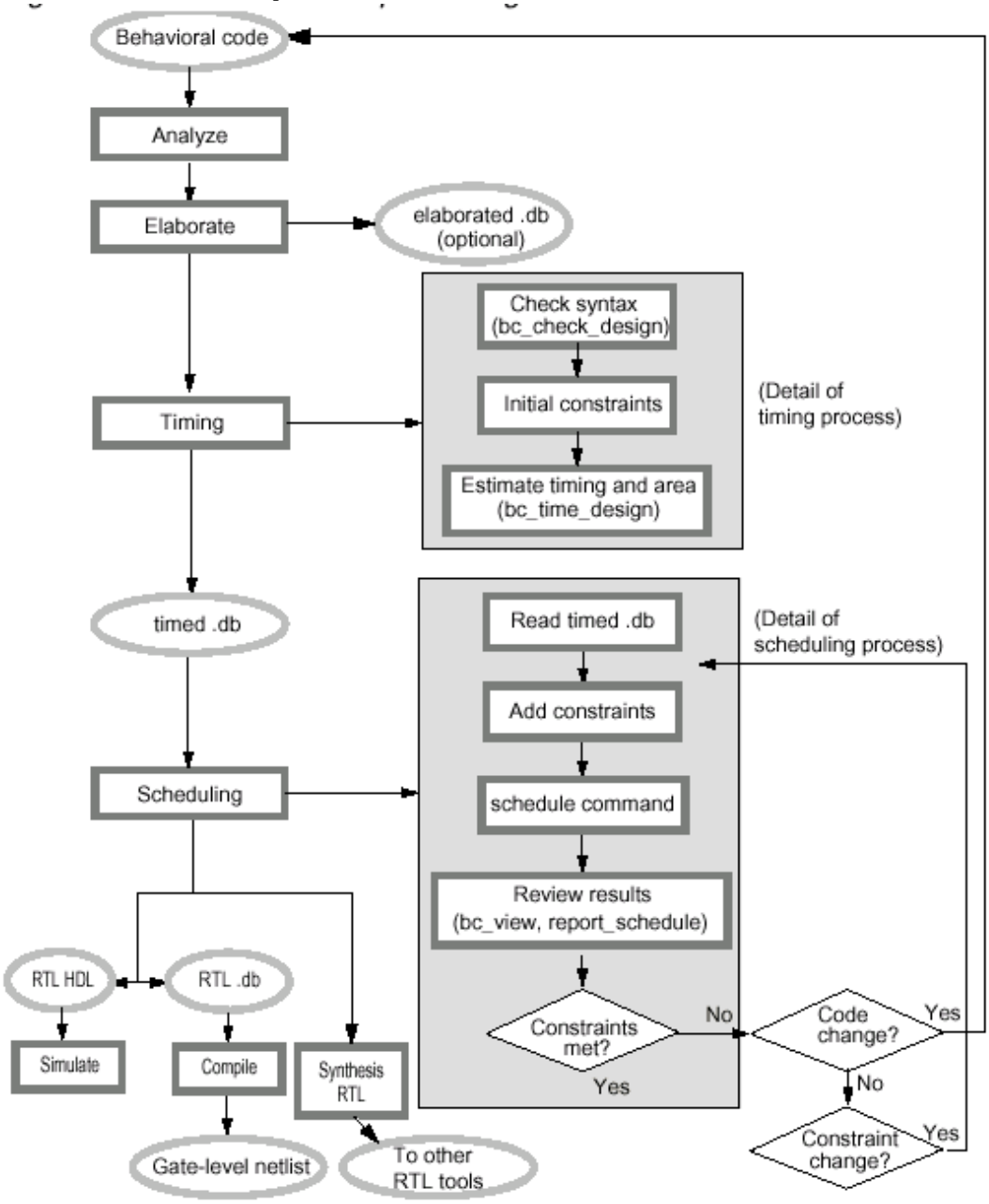


图 3.1 Behavioral Compiler 的设计流程

3.1.2 Behavioral Compiler 设计流程及相关命令

1. 分析设计：

   analyze [-library library_name] [-work library_name] [-format vhdl | verilog] [-create_update] [-update] [-define define_netlist] file_list

   如： analyze -format vhdl name_design.vhd

2. Elaborating 设计：

   elaborate design_name [-library library_name | -work library_name] [-architecture arch_name] [-parameters parameter_list] [-file_parameters

file_list] [-update] [-schedule]
如： elaborate -schedule entity_name -arch arch_name

3. 给设计时序约束：
初始约束： create_clock
set_dont_use
set_drive
set_driving_cell
set_input_delay
set_load
bc_margin
set_memory_input_delay
set_memory_output_delay
set_operating_conditions
set_wire_load_min_block_size
set_wire_load_mode
set_wire_load_model
set_wire_selection_mode
设计检查 ： bc_check_design
时序和面积预算： bc_time_design [-force] [-fastest]
产生时序面积的预算报告： report_resource_estimates
输出时序的.db **文件**： write

4. 约束安排： 约束：
set_cycles
set_max_cycles
set_min_cycles
set_memory_input_delay
set_memory_output_delay
pipeline_loop
驱动：
bc_fsm_coding_style
bc_dont_register_input_port
bc_dont_ungroup
chain_operations
dont_chain_operations
ignore_array_precedences
ignore_array_loop_precedences
ignore_memory_precedences
ignore_memory_loop_precedences
set_common_resource
set_exclusive_use
set_behavioral_reset
用 report_scheduling_constraints 显示当前设计上安排的约束。

5. 设计安排(scheduling the design)：
schedule [-effort zero | low | medium | high] [-io_mode
cycle_fixed | superstate_fixed] [-extend_latency] [-hostname]
[-arch remote_host_architecture] [-allocation_
effort zero | low| medium | high]
schedule 报告：
report_schedule [-process process_name] [-operations [-mask
[r][w][l][L][o][p]] [-start start_cycle] [-finish end_cycle] [-delimiter

"character"]] [-variables [-min min_width] [-max max_width] [-start start_cycle] [-finish end_cycle] [-delimiter "character"]] [-summary] [-abstract_fsm [-mask [r][w][o][s]]]
[-verbose_fsm [-mask [r][w][o][d][s]]]

6. 写出 RTL 级文件： write
   写 RTL 级的.db 文件：
   写出RTL 级HDL仿真文件: vhdlout_levelize = true
   write -format vhdl -hier -output file_name.vhd
7. 可综合的 RTL 输出： write -rtl_script script_file_name

3.2.3 脚本范例

```
analyze -f vhdl cmplx.vhd /* analyze the vhdl file */
elaborate -s cmplx/* elaborate design for scheduling */
create_clock clk -period 10
write -hier -out cmplx_elab.db /*save elaborated design */
bc_check_design -io super
/* Here you may wish to add operating conditions and */
/* wireloads before timing the design */
bc_time_design
write -hier -out cmplx_timed.db /*save the timed design */
/* Begin Scheduling Cycle */
/* specifying some scheduling constraints */
/*set_cycles ...*/
/* scheduling with superstate_fixed mode */
schedule -io_mode super
/* save scheduling reports to file */
report_schedule -op > cmplx_schd.rpt
report_schedule -summary >> cmplx_schd.rpt
/* End Scheduling Cycle */
write -hier -out cmplx_rtl.db /*save RTL design to a db*/
write -hier -f vhdl -rtl_script rtl.scr -out syn_rtl.vhd
/* saves synthesizable RTL */
vhdlout_levelize = true
/*write flattened, simulation-only RTL model*/
vhdlout_use_packages = {ieee.std_logic_1164}
/* include all packages used in the design */
write -hier -f vhdl -out cmplx_rtl.vhd /* saves simulatable rt level design */
```

3.2 Module Compiler

Moduler Compiler 提供了高性能的 data-path 综合和优化，简化和自动化了 data-path 设计，帮设计者解决复杂而没有规则的 data-path 设计。 Module Compiler 提供了大量常用函数来创建 data-path blocks.每一个函数都有设置参数的图形界面。由于篇幅的限制，它的具体操作和命令略。

第 4 章　　逻辑综合

4.1　逻辑综合概述

4.1.1　逻辑综合的概念

综合(synthesis)：就是把思想转换为实现欲想功能的可制造的设计。综合是约束驱动和基于路径的。

在这里，综合也就是把行为级或 RTL 级的 HDL 描述转换为门级电路的过程，用公式表示就是：

综合等于 = 翻译 + 优化 + 映射

(Synthesis = Transiation + Optimization + Mapping)

用图形表示就是：(见图 4.1)

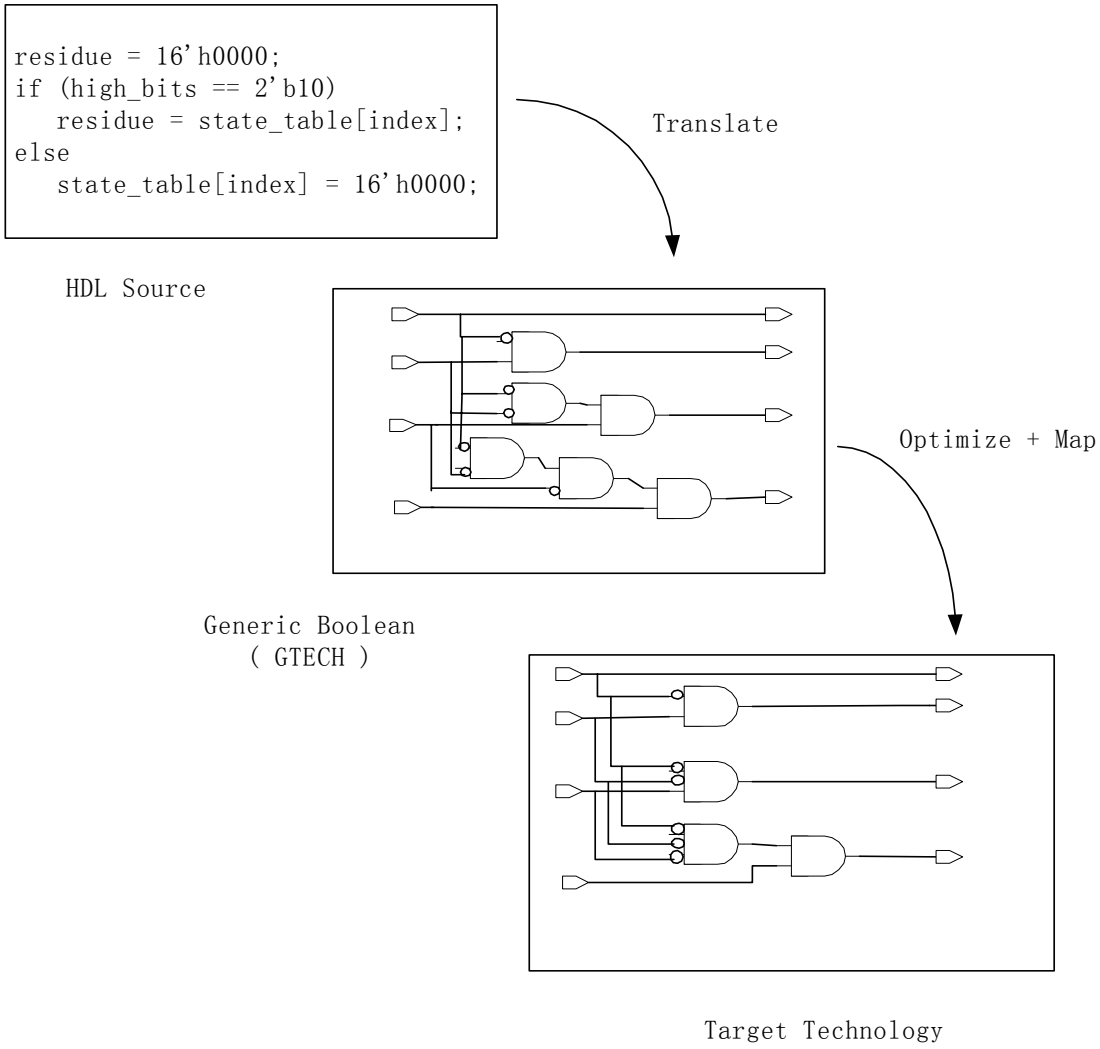


图 4.1 综合的概念

4.1.2　逻辑综合的工具介绍

- **工具操作界面**

设计编译器(Design Compiler 简称 DC)是 Synopsys 综合工具的核心。综合一个设计时，可以选用两种界面： A 。设计分析器(Design Analyzer 简称 DA)-图形窗口界面。 B 。dc_shell—命令行界面。

DA 图形窗口界面的启动：%da

dc_shell 命令行界面的启动；%dc_shell

dc_shell 界面的提示符为:dc_shell >

dc_shell 命令行界面支持两种脚本语言:dcsh 模式和 dctcl 模式。

dcsh 是使用源于 Synopsys 的语言。 dctcl 使用工具命令语言(Tool Command Langugae)。

dcsh 模式和 dctcl 模式比较

tcl 是一种开放型的工业标准语言。它比 dc_shell 更加强大。

启动 dcsh 模式用 dc_shell 命令,启动 dctcl 模式用 dc_shell -t

Dctcl	dcsh
广泛的在线帮助	有限的在线帮助
工业标准的开放式接口	Synopsys 的专有接口
广泛的文件和字符串操作能力	有限或无文件和字符串操作
可用户定义过程	无用户定义过程
支持变量数列	不支持数列
不被 DA 支持	为 DA 支持
有限的脚本或知识基础	有脚本或知识基础

如果你已经有了 dcsh 的 setup 文件或脚本文件(.scr),你想转换为 Tcl 的 setup 文件和约束文件,则我们只需执行下面命令即可.

setup 文件的转换:

设 dcsh 的 setup 文件为.synopsys_dc.setup.old,要转换为 Tcl 的 setup 文件.synopsys_dc.setup 则

% dc-transcript .synopsys_dc.setup.old .synopsys_dc.setup

脚本文件的转换:

设 dcsh 的约束文件为 old_scriptfile.scr,dctcl 的约束文件为 tcl_script.tcl ，则

% dc-transcript old_scriptfile.scr tcl_script.tcl

由于 dcsh 和 dctcl 是可以转换的，以下的介绍中，在支持 dcsh 的地方，将都用 dcsh 命令。

- **Synopsys 格式**

大多数 Synopsys 产品都支持和共享一个公用的中间结构--"db"格式。 db 文件是描述文本数据的二进制已编译表格式。 DC 可以读和写以下的所有格式： Verilog ， VHDL ， EDIF 。

4.1.3 逻辑综合的流程

流程图如图 4.2 ：

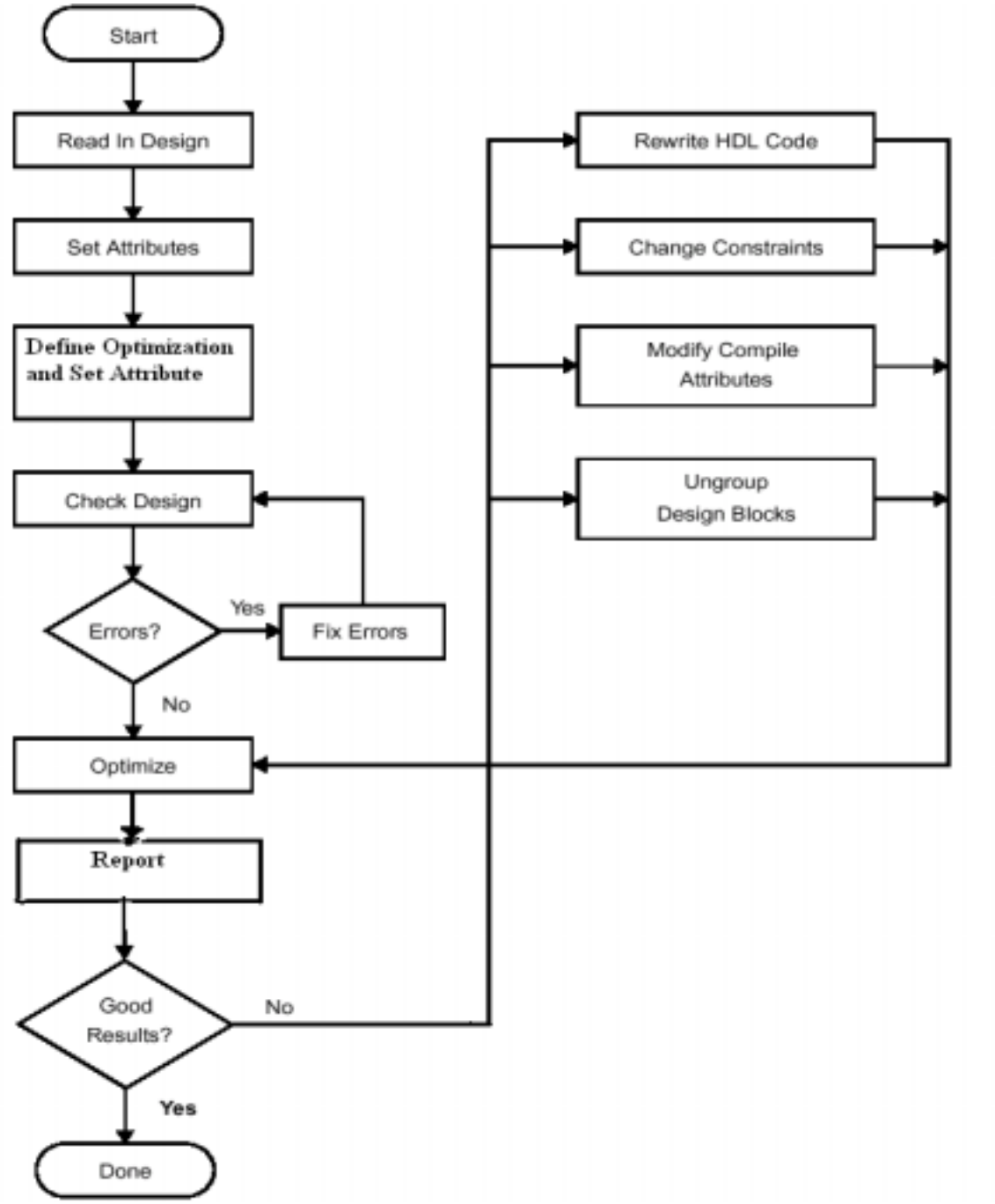


图 4.2 逻辑综合设计流程

4.2 setup 文件,库以及一些基本概念

4.2.1 setup 文件

把行为级描述转换为门级电路，在映射过程种，必须有技术库的支持，否则它将找不参照的元器件。为此我们在综合前必须在 setup 文件中设置好综合所需的技术库。这里所需的技术库的格式为.db 和.sdb 格式。技术库的描述略。

Synopsys 的设计编译器(Design Compiler,简称 DC)提供了三个同名的 setup 文件.synopsys_dc.setup 。一个为安装目录下的 setup 文件，它提供系统管理员指定的系统变量设置。一个为你根目录下的 setup 文件，它提供你的工作环境变量的设置：公司的名字，你的名字，背景色。这是由用户指定的 DC 值。

一个为你工作目录下的 setup 文件，它规定了你设计所指定的 DC 值，如：

search path, target library, link library , symbol library.

启动 DC 工具时，它依次读入这三个文件，且读的越后的文件优先级别更高，即相同的

变量，后面的设置值将覆盖前面的设置。

根目录下的 setup 例子如下：

```
company = “ your_company”;
designer = “your_name”;
view_blackground = “black”;
```

工作目录下的 setup 文件例子如下：

dcsh 模式：

```
searsh_path = {}  + search_path
link_library = {MTC45000.db} ;
target_library = {MTC45000.db} ;
symbol_library = {MTC45000.sdb} ;
define_design_lib work -path work ;
```

Tcl 模式：

```
set search_path [concat [list] $search_path]
set link_library [list MTC45000.DB]
set target_library [list MTC45000.db]
set symbol_library [list MTC45000.sdb]
define_design_lib work -path work
```

说明：

search_path:为 DC 提供未分析设计标准的搜寻路径，亦即你的技术库的搜寻路径。

如果你的库不是放在 DC 的安装目录下的库的目录下,则你还需修改你的 search+path,指定库的目录。方法是:

dcsh 模式：

```
search_path = {directory}  + search_path
```

dctcl 模式：

```
    set search_path [concat [list directory] $search_path]
    link_library:
```

指明了你的设计所参照的子设计的位置。 DC 根据 link_library 寻找它所参照的设计。如果参考设计的完整名字在 link_library 里没有定义，则需在 search_path 中包括这参考设计的路径。 link_library 定义了被单独使用的元器件的库的名字。即，link_library 里的元器件是不被 DC 所 inferred 的。

4.2.2 库

target_library:

指明了在你优化设计时用到的元器件的库。

symbol library:指明了含技术库元件的图形描述的库。

4.2.3 对象

在进行综合时，我们经常会遇到一些对象的概念。搞清楚这些概念具体是指代什么是很有必要的。

Design: 对应于执行一定逻辑功能的电路描述。 design 可以是独立的一个,也可以含有其他的子设计。子设计虽然可以是设计的一部分,但是 Synopsys 也把它看成是一个设计。

Cell: 是 design 中的子设计的一个 instance 。在 Synopsys 的术语中， cell 和 instance 被认为是一样的。

Reference: cell 或 instance 参考的源设计的定义。

Port: 指主要 inputs,outputs 或 design 的 IO 管脚。

Pin: 对应于设计中的 cell 的 input,output,或 IO 管脚。

Net: 这是信号的名字,即通过连接 ports 与 pins 或 pins 与 pins 而把一个设计连在一起的的金属线的名字。

Clock: 作为时钟源的 port 或 pin..

library: 对应于设计的综合目标或参考连接的工艺指定单元的集合。

具体示例如图 4.3 ：

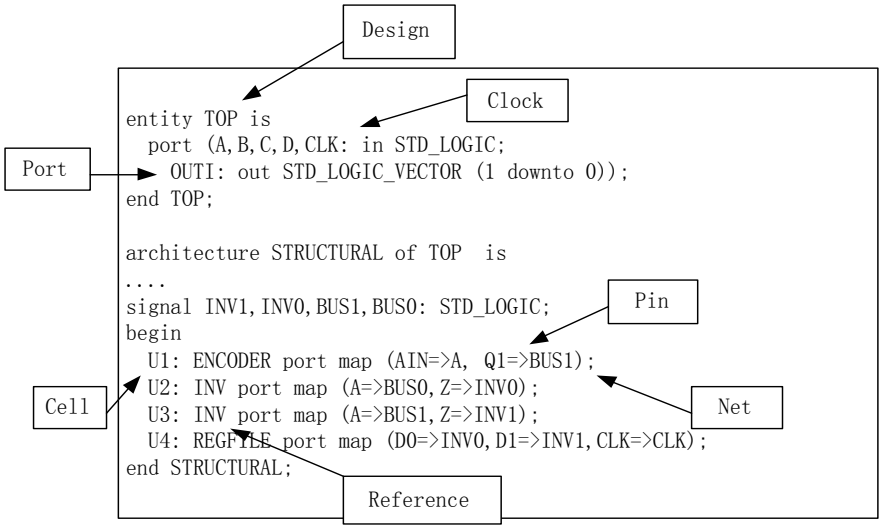


图 4.3 对象

4.3 设计分块

分块(partitioning)：把复杂的设计分成各个小部分的过程。

Partitioning = Divide +Conquer

概念如图 4.4

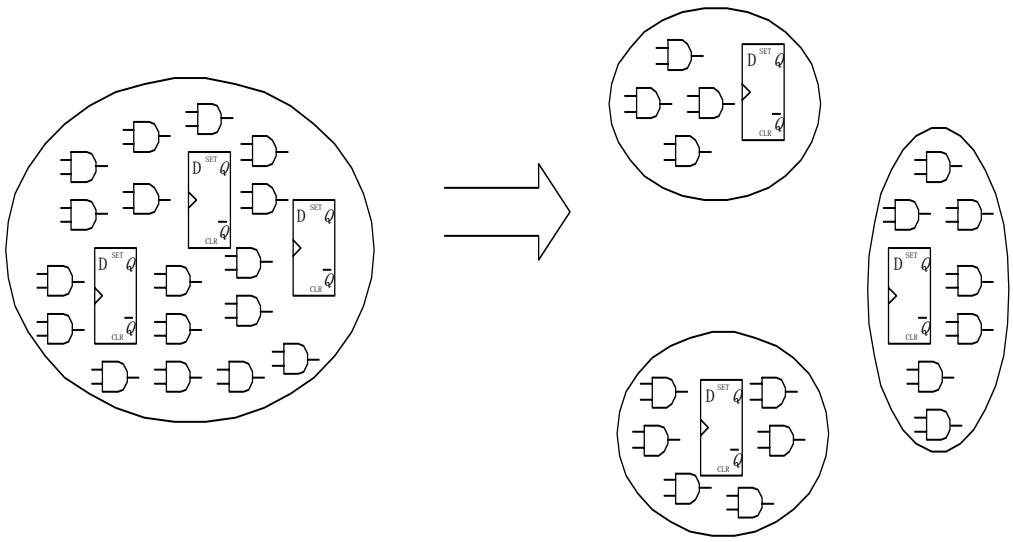


图 4.4 分块的概念

分块是成功的进行综合和布局布线的关键。传统上的分块是根据逻辑功能,而不考虑综合的。固定的边界降低了综合结果的质量,使优化难以进行。正确的给设计分块能大大的增强设计结果，而且降低编译的时间和简化脚本的管理。

以下是分块的几条建议：

1. 把相关的组合逻辑保留在同一模块中；
2. 考虑设计的重用；
3. 根据它们的功能划分模块；
4. 把结构逻辑和随机逻辑分开；
5. 合理的块大小(每个块最大大约为 10K 个门)；
6. 把顶层分块出来(独立 I/OPads,边界扫描 Boundary Scan ，核心逻辑,Clocks)；

7. 顶层避免存在 glue-logic;
8. 把状态机和其他逻辑独立开来；
9. 避免在一个块中存在多时钟；
10. 把用来同步多个时钟的块独立出来；
11. 分块时考虑你的版图设计。

块的产生： entity 和 module 语句定义了层次的块。 entity 或 module 的示例也产生了一个新层。算术电路(+,- ,*,。。)的 inference 也能产生新层。 process 和 always 语句不会产生层次。

逻辑优化并不能穿过块的界线。最好的分块是把相关的组合逻辑和它的目的寄存器聚集在同一个块中。这样组合优化技术能被完全的利用，时序优化也可以吸收一些组合逻辑到复杂的触发器中(JK,T,Clock-enabled)。好的分块如图 4.5 所示：

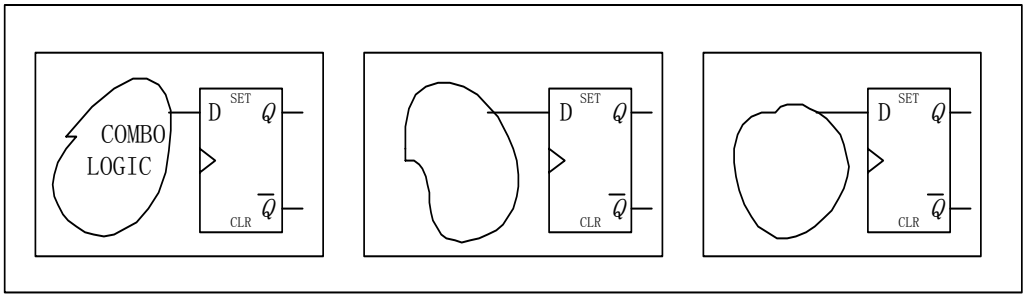


图 4.5 最佳的分块

什么是 glue-logic :就是用一个组合逻辑把几个块结合起来，这个组合逻辑就是 glue logic.由于这 glue logic 不能被吸收，优化是被限制的，要删除这 glue logic,我们把这 NAND 放进后面的逻辑电路中。这样 glue logic 就可以和其他逻辑一起优化了。顶层设计只能是一个结构的网表，不需要被编译。换句话说，就是顶层不能含 glue logic ，当然其他层最好也是不要存在 glue logic 。

分块时各个块的大小最好不要相差太大，因为，太小多余的边界将限制优化，太大编译的时间又太长。尽量减少不必要的层。编写 HDL 源代码时每一个块的输出最好都是通过寄存器实现，这样可以可以简化约束的说明和方便优化。 input delay 和 output delay 可以不用设置。块的名字必须和文件名相同，否则在分析是将会出错。

初始的分块由 HDL 定义。不满意的话， DC 可以进行调整。

DC 分块的命令是 group 与 ungroup.

例如我们把两个块 U1,U2 合并为一个块 U23,则转到当前层后

```
group {U1,U2}  -cell_name U23
ungroup {U23}
```

命令的具体应用请查看在线帮助。

4.4 读进设计

DC 的输入格式可以是 Verilog HDL,VHDL 等硬件描述语言，可编程逻辑阵列(PLA)，EDIF2000 ，格式。

对于 HDL 格式， DC 要求用 analyze 和 elaborate 读进设计。

analyze ：读进 VHDL,或 Verilog 文件，检查语法和可综合逻辑，并把设计已中间格式存在设计工作库(WORK)中。 analyze 后，在 DA(Design Analyzer)中并看不到有什么东西出现。 analyze 命令可以同时对若干个文件执行操作。

elaborate ：从工作库中把 analyze 后的中间文件转换为一个设计。 elaborate 命令用综合的操作符代替 HDL 的操作符，且决定正确的总线大小。 elaborate 命令后，在 DA 中你可以看到出来了一个一个的模块。一个文件一个，即一个 entity 或 module 一个。 elaborate 一次只能对一个文件进行操作。这点需要注意。

在这里我们需要提醒一下的是， entity 或 module 的名字，即设计的名字必须和文件

名相同。因为 analyze 后它存在 WORK 里的名字是设计的名字(entity or module)而不是你源程序的文件名，所以如果你 elaborate 的是文件名，那么如果设计和文件名不同， DC 将找不到你的设计，出现 error 。当然如果你不厌其烦的特别让 elaborate 他的设计名，这问题就不存在了。

对于其他非 HDL 文件， DC 用 read 命令读进设计。

read 命令并不产生中间文件，而是之间把他转换为了 DC 里的符号。

理论上， read 可以读进所以的设计，不管是不是 HDL 文件，但是建议对 HDL 文件用 analyze 和 elaborate 读。

对于 VHDL 的 package 文件，我们应该用 read 命令读，且必须在读进 VHDL 源程序前读。

4.5 设计约束

为了从 DC 中能得到合适的结果，设计者必须通过描述其的设计环境，目标任务和设计规则来系统的约束其设计。约束包含时序和面积信息。它们通常是从规格说明中提取出来的。 DC 用这些约束去综合和优化设计以符合其目标任务。本文只讲些最常用的约束命令，且不会讲的很详细。约束命令的具体用法请参照在线帮助。

4.5.1 设计环境

几个环境变量含义如图 4.6 所示：

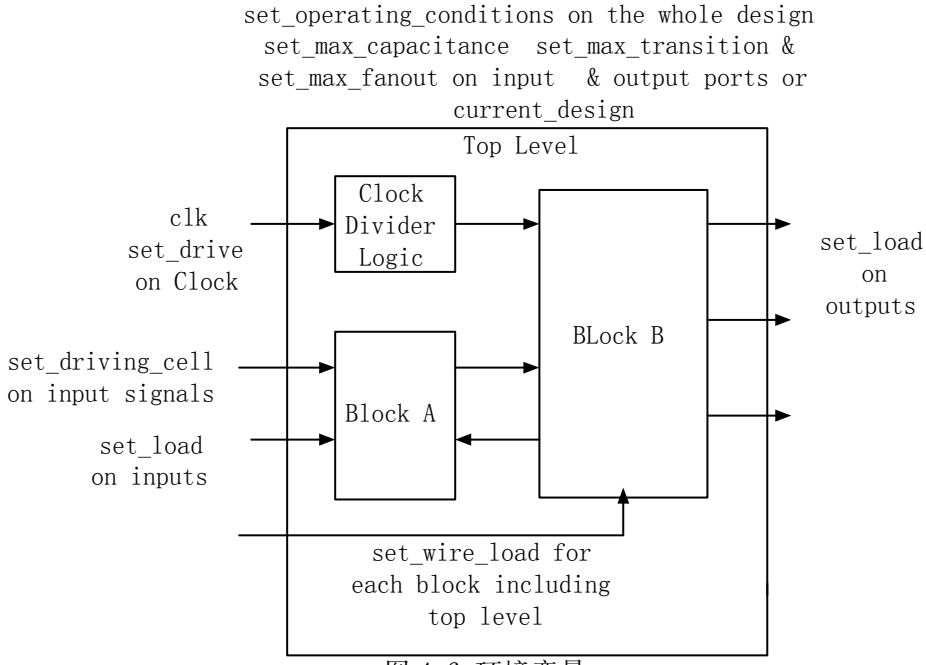


图 4.6 环境变量

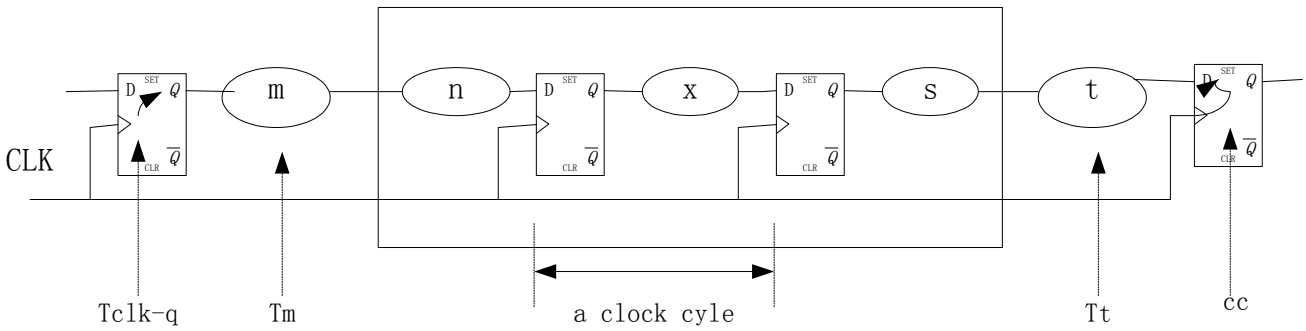


图 4.7 时序图

set_operating_conditions ：描述了设计的工艺，电压和温度等条件。通常以 WORST,TYPICAL,BEST 的情况进行描述。技术库通常具有正常情况的的特性。为了查找芯片

商提供的运行环境，我们可以用 report_lib libname 命令。

set_operating_conditions <name of operating conditions>

例子：

dc_shell> current_design = “addtwo”

dc_shell> set_operating_conditions -max “typ_124_4.50”

最坏(即最慢)的运行条件是最低的电压和最高的温度。

最好(即最快)的运行条件是最高的电压和最低的温度。

对于设置 the best case 的运行条件是：

set_operating_conditions –max WORSTC_OPCONDS –min BESTC_OPCONDS

如果你想检查你的设置，你可以用 report_port -verbose,write_script 和 report_design 命令。

set_load: 定义了输出单元总的驱动能力。

set_load <value> <object list>

例子：

直接指定一个值：dc_shell> set_load 4 find (port OUT1)

利用 load_of(lib/cell/pin)命令把技术库的门作为负载数的衡量：

dc_shell> set_load load_of(CBA/AN2/A) find (port OUT1)

load_of(lib/cell/pin)可以乘以系数：

dc_shell> set_load load_of(CBA/IVA/A) * 3 find (port OUT1)

set_driving_cell: 模拟了驱动输入管脚的的驱动单元的驱动电阻。定义了信号到达输入管脚的传输时间，可以直接指明驱动输入管脚的外部实际单元。

set_driving_cell -cell <cell name> -pin <pin name> <object list>

例子：

dc_shell> set_driving_cell -cell BUFF1 -pin Z all_inputs()

set_drive: 指明了输入管脚的驱动强度。模拟了输入管脚的外部驱动电阻。它一般只用在 clock 管脚上。

set_drive <value> <object list>

例子：

dc_shell> set_drive 0 {CLK RST}

set_wire_load: 用来提供估计的统计线载(wire load)信息，反过来也用线载信息模拟 net 延时。线载模型是基于 net 扇出的 net 参数估计。

set_wire_load <wire-load model> -mode <top | enclosed | segmented >

例子:

set_wire_load MEDIUM –mode top

wire load 模型是芯片制造商产生的,但是也可以你自己产生。自己产生的方法举例如下:

设库名为 Extra.lib:

Library (“ extraa ”) {

operating_conditions (“ SLOW ”) {

process : 1.75;

temperature : 100;

voltage :4.66;

tree_type : “ worst_case_tree ”;}

}

再执行以下命令：

dc_shelll> read_lib extra.lib

dc_shelll> write_lib extra –output extra.lib

dc_shell> link_library = {“ *”,tech_library extra.lib}

dc_sehll> set_operating_conditions “ SLOW ” –library “ extra ”

这样就完成了运行条件的产生。

4.5.2 设计规则

set_max_fanout, set_nax_transition, set_max_fanout 设计规则在技术库中设置，为工艺参数所决定。

set_max_transition <value> <object list>

set_max_capacitance <value> <object list>

set_max_fanout <value> <object list>

例子：

dc_shell> set_max_transition 0.3 current_design

dc_shell> set_max_capacitance 1.5 find(port,"out1")

dc_shell> set_max_fanout 3.0 all_outputs()

4.5.3 时序和面积约束：

各种变量的含义如图 4.8 所示:

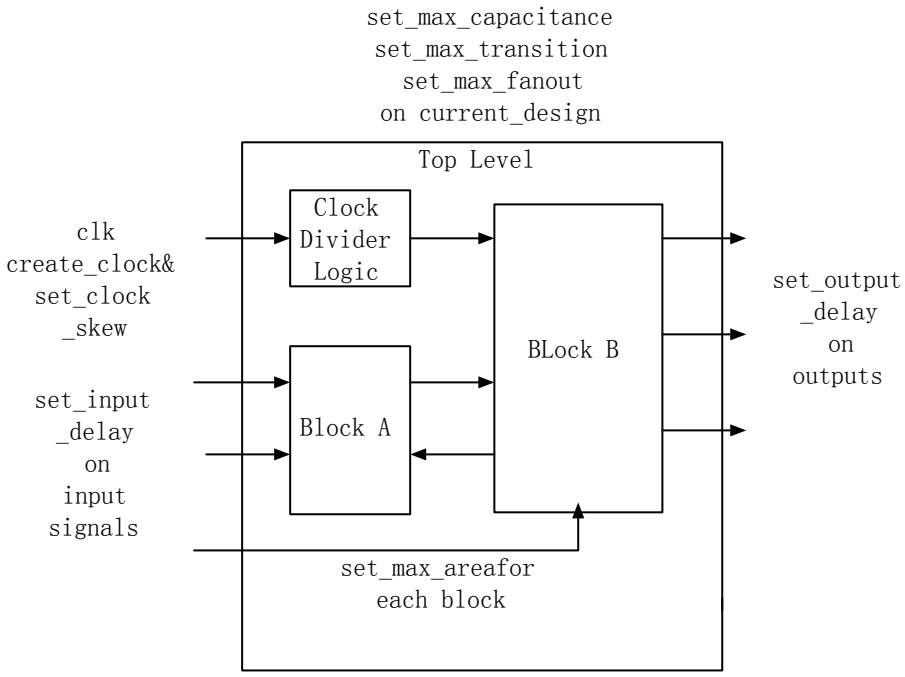


图 4.8 时序和面积约束

create_clock: 定义时钟周期.

例子:

dc_shell> create_clock –period 10 find(port,Clk)

set_dont_touch_network :告诉 DC 不要缓冲 clock net,即使它驱动了很多的触发器。一般此命令用于 clock 网络和 reset 。

例子：

dc_shell> create_clock -period 20 find(port,Clk)

dc_shell> set_dont_touch_network find(clock ,Clk)

set_input_delay: 限制输入路径的延时。

input delay = Tclk-q + Tm

set_input_delay delay_value [-clock clock_name] [-clock_fall] [level_sensitive] [-rise] [-fall] [-max] [-min] [-add_delay] port_pin_list

例子：

set_input_delay -max 4 -clock Clk find(port,“ A ”)

set_input_delay 命令的参数是前面电路的 delay 数。如： clock 的周期为 10 。我们

限制输入 ports 的延时是时钟周期的 80%则参数应为 10 － 8 ＝ 2.不想包括的管脚可用－除掉，注意前后都有空格，

如： set_input_delay 2.0 –max –clock Clk all_inputs() – Clk

set_output_delay ： 约束输出路径的延时。用法和 set_input_delay 相似。

output delay = Tt + Tsetup

set_output_delay delay_value [-clock clock_name] [-clock_fall] [-level_sensitive] [-rise] [-fall] [-min] [-max] [-add_delay] [-group_path group_name] port_pin_list

例子：

set_output_delay -max 5.0 -clock Clk find(port,"s")

与 set_input_delay 同样 set_outputdelay 的延时数是后一个电路的建立时间。也应该时钟周期减去 outputs 的延时数。

你想看你的约束设置，你可以用 report_clock, report_port -verbose, write_script, report_design 命令。

如果你的约束设置错了，你想修改，你可以用 reset_design, remove_design 命令删除你的约束或设计重新来过。

4.5.4 时钟

DC 并不综合时钟树，时钟树的综合基于物理布局，由芯片制造商完成。为了模拟(model)clock 分支的不确定性，我们使用 set_clock_uncertainty 命令。

对于同步的多时钟设计，我们采用虚拟时钟的方法。即在你的设计中选择所有时钟的最小公倍数作为你的虚拟时钟的频率，然后把它分频为你设计中的多时钟服务。产生虚拟时钟的方法如下：

create_clock -name vTEMP_CLK -period 20

注意：必须是-name ，而且没有源的 pin 或 port 。

对于异步多时钟设计，凡是经过异步边界的路径，我们都应该不进行基于时序的综合。为此，我们使用 set_false_path 命令。

例子：

set_false_path -from find(clock CLKA) -to find(clock CLKB)

检查你所加的时序约束，用 check_timing 命令。

4.5.5 时序分析

DC 有一个内嵌的静态时序分析器叫设计时间(DesignTime)。用这个 DesignTime ，你可以对设计进行时序分析。你也可以用 Synosys 的辅助工具 PrimeTime 进行静态时序分析。这里我们就以 DC 内嵌的静态时序分析器为例。命令为 report_timing 。输出的时序报告文件按一条一条信号路径显示。每一个时序路径都计算两次，一次为上升沿输入，一次为下降沿输入。时序报告文件具体如何看，请参考有关资料。

DesignTime 把设计分成一个个信号路径。每个路径的起点(startpoints)可以是 input ports,也可以是触发器的的 clock pin 。终点(endpoints)可以是 output ports,也可以是时序器件的数据输入脚。路径的划分由 clock 决定。

4.6 设计综合和优化

当设置好所有的约束条件后，就可以开始进行设计综合和优化了。综合优化的命令为 compiler 。在优化的过程中，时序和面积是两个矛盾的约束。我们把时序约束放宽了，则能得到比较小的面积。时序约束要求越高则芯片的面积要求将越大。在以前的版本中， DC 总是优先考虑面积，总是为得到最小面积的最快逻辑(design space exploration)来分析设计。现在由于时序方便的问题更重要，已经把时序约束放在了比面积更重要的地位。

为了能够得到比较好的综合结果，最好的是在你写源程序时就考虑了综合。

优化分为三个阶段:结构优化阶段，逻辑优化阶段，门级优化阶段。

结构优化：结构的高级综合指：设计构件(DesignWare)的更佳选择，

共享子表达式，资源共享，操作符的重新排列。

高级综合是约束驱动的，即基于约束和编码系统。通过高级综合， DC 产生了符合时序要求的高效面积结果。需要注意的是，高级综合只能在优化未映射设计时发生。

逻辑优化：在高级优化以后，电路功能在 GTECH（ general technology ）部分已经出现。在逻辑优化期间两个优化处理可以选择： structuring ， flattening 。

structuring ：产生中间结构来改善设计，是基于约束的，可以同时改善设计的面积和速度。选择 structuring 综合： set_structure true

flattening: 删除中间结构,以减少积和电路设计。独立于约束，在速度的优化上很有效，但是需要较大的面积。 Flattening 由于库的限制并不能保证实际映射一定为两级的积和电路。选择 flattening 综合： set_flatten true

门级优化: 主要任务是映射。映射包括组合电路的映射，时序电路的映射。

4.6.1 约束和时序检查：

最常用的约束和时序检查命令：

report_constraint -all_violators ：报告设计约束的所有违约(violators)。

report_timing -delay_max ：报告每一个路径组建立时间（ setup time ）约束的最差时序路径。

report_timing -delay_min ：报告每一个路径组保持时间（ hold time ）约束的最差时序路径。

4.6.2 违约的解决

如果你的是约束没有满足，则需要重新综合。执行连续的编译是没用的，你需要改变一些东西，如调整你的约束，改变 set_structure 或 set_flatten 选项，或改变编译的映射尝试（ map effort ）。可同时 enable the Design Ware Foundation library components, in order to enable faster adders:

```
synthetic_library = {dw_foundation.sldb}
link_library = link_library + dw_foundation.sldb
```

/*这里的 dw.fondation.sldb 是否用{} 括起来没关系

Compile -map_effort (low | medium | high)

缺省选择为 medium.重新编译时一般选择 high.

使用 compile -incremental_mapping 重新编译设计 。使用此命令时只有门级优化是进行的，在运行速度上比常规的编译要快。

当你有设计规则的违约时,用 compile -only_design_rule 命令.

由于综合时总是偏于最小面积和最快速度,所以 setup 时序一般不会有问题，有问题的话也只能去该你的设计了。但是由于数据到来的太快， hold 时序往往会出现违约（ violator ）。为了确定保持时间的违约，需要设置为最好情况下的约束：

```
set_clock_uncertainty -hold
set_input_delay -min
set_output_delay -min
set_operating_conditions -min
```

在缺省情况下， DC 并不确定保持时间的违约，我们使用 set_fix_hold 和 compile -only_design_rule 让 DC 去确定保持时间的违约。

使用 characterize

characterize 计算周围环境强加在设计上面的属性和约束,然后放这些约束在设计上，write_script 输出当前设计的所有约束到一个脚本文件上。

characterize 的使用方法实例：

```
current_design TOP
characterize -constraints find(cell, "U1")
current_design A
write_script > A_w.scr
```

characterize 工作在 cell ，而不是 design,而 write_script 则显示 current_design 的所有 constraints,故要 write_script 是需先执行 current-design 命令，使它转换到泥需的 design 上， write 后再转回去。

限制：

characterize 只能在所有的块都已映射为了门级电路的设计上使用，不能用来作为第一次编译的设计预算(budgets).characterize 一次只能对一个块进行处理。

✧ 假设关键路径的大多数组合电路在 ALU 设计中。则你可以：

对于小违约现象(10-30%):

```
current_design ALU
compile –map_effort high –incremental
current_design = RISC_CORE      /*current_design 后面有无＝一样*/
report_constraint all_violators
```

对于大违约现象 (>40%):

```
remove_design ALU
read unmapped/ALU.db
current_design ALU
include scripts/ALU_w.scr
synthetic_library = {dw_foundation.sldb}
link_library = link_library + {dw_foundation.sldb}
compile –map_effort high
current_design RISC_CORE
report_constraint –all_violators       /*可检查有无 violators*/
```

如果你尝试了以上的方法后还不行，对不起，请回去修改你自己的源代码。

4.6.3 大型设计和层次设计主要综合方法学

对于层次设计， DC compile 的时候先不考虑约束把每一个 block 映射成门级电路，然后再优化逻辑以符合时序面积的约束，并通过周围的块确定违约。

当你的设计中存在多 instance 问题(一个实例在不同的地方被多次使用)时，你可以用 check_design 命令检查，同时这命令还可以帮你检查是不是有管脚没连好。对于存在多实例的设计，我们可以用 3 种方法解决： uniquify, compile + dont_touch ， ungroup 。

uniquif ：为每一个多次引用的 instance 做备份，一个 instance 一个备份，这样每一个 instance 都得到了一个单独的设计名。 DC 可以为每一个 instance 根据它的指定环境进行映射。使用方法为：先 uniquify,在 compile.

compile + dont_touch ： set_dont_touch 设置 dont touch 属性给设计对象，它阻止了这个设计对象在编译时的调整。(注意：如果这个设计对象是未映射的，那幺它还将是未映射的)使用方法：我们先约束和编译这个设计对象，然后设置 dont_touch 属性在这个设计上，最后再编译整个大的设计。

ungroup ：删除所有的层，为当前设计的所有 cells 和 references 产生单一的名字。

例子：设 A_des 是一个多实例对象。 D_design 是顶层设计。

```
read -f db unmapped/A_des.db
current_design = Ades
link
include Aconstraints.scr
compile
read -f db unmapped/D_design.db
current_design = D_design
set_dont_touch find(design Ades)
include Dconstraints.scr
compile
```

如果你不考虑编译运行时间，内存的限制，以及一次性就要放好这个块，你可以选择 uniquify.

注意：对于多 instance 设计，如果你不进行以上几个操作中的一项， DC 将出现 ERROR ，提示你解决多 instance 问题。

综合一个设计。我们可以采用 bottom-up ，也可以采用 top-down 。

top-down 综合方法：

1. 读进所有的设计，
2. 解决多实例问题

3. 应用顶层约束
4. 编译
5. 查看结果
6. 保存设计

top-down 综合方法的优点是模块之间的依赖关系被自动维护。人花在工具运行的时间比较少。

bottom-up 综合方法：

1. 单独约束和编译每一个子模块，
2. 确定所有的子模块都符合了它们的初始约束，
3. 读进完整的设计，并应用顶层约束，
4. 检查约束报告，如果都通过了，那你就完成了。

bottom-up 综合方法的优点是大型的设计通过"divide and conquer"方法编译了，不受有效内存的限制。缺点是需要人们反复的干预直到块与块之间的接口稳定下来。需要仔细的修改控制。

4.7 设计输出

设计综合好，且所有的约束(可以除了 hold time)都满足以后，我们就可以输出综合后的文件了。

DC 可以输出.db, VHDL, Verilog, EDIF, PLA, State Table, LSI, XNF 等格式。

为了以后重新读此设计的方便，我们一般要把设计存为.db 格式放在 mapped/目录下。为了在 VSS 中进行版图后的门级仿真，我们要把设计存为 VHDL 格式。为了在 SE(Silicon Ensemble)中进行自动布局布线，我们又要求把设计存为 Verilog 格式。

在综合后仿真中我们必须前注释延时的时序信息(SDF 文件)到 VSS 工具中，这时序信息由 DC 得到。得延时时序信息的命令为：

write_sdf *design_name*.sdf

进行自动布局布线，要求我们前注释约束信息到版图工具中去。这约束信息也叫 SDF 文件，注意与前面的 SDF 文件的区别。产生此约束信息的 SDF 文件的命令如：

write_constraints -format sdf-v2.1 -max_nets 0.05 -net_priorities -max_path_timing -max_paths 1 -hierarchy -output *design_name*.sdf

4.8 脚本的编写与执行

约束脚本的编写(constraints.scr) (这里假设所有的输出都已锁存，故不需要设置 input delay 和 output delay)：

1. Reset the design
2. Create a clock object
3. Set input delays on all input ports except the clock port
(若无 Assume 则＋ set output delays on all output ports)
4. Set the operating condition
5. Set the wire load model
6. Define the cell driving all input ports except the clock port
7. Define the maximum capacitance allowed on the input ports
8. Define the pin capacitive load on all output ports

总的综合脚本的编写(runit.scr) (假设设计明和文件名相同)：

1. Read a list of designs into dc_shell
2. For each design, set the *current_design* variable
3. For each design, link
4. For each design, apply the *constraints.scr* script file
5. For each design, compile
6. For each design, generate and save the results from a constraint report to the *reports* directory, using a meaningful name
7. For each design, save the mapped design under the *mapped* directory under a meaningful name

8. Quit dc_shell

context_check 必需在完整的 script 文件上才使用，所以对约束文件 constraints.scr 不能用 context_check

产生和检查约束文件 constraints.scr ：

dc_shell -f scripts/constraints.scr -syntax_check

产生和检查总的脚本文件 runit.scr ：

dc_shell -f scripts/runit.scr -syntax_check

dc_shell -f scripts/runit.scr -context_check | tee context.log

运行脚本综合你的设计:

dc_shell -f scripts/runit.scr | tee runit.log

constraints.scr 和 runit.scr 的样本见附录

- 一些注意点及常用命令的用法：

  执行 *group, ungroup* 命令需在 Schematic View

  执行 *constraint, compile* 命令需在 Symbol View ，也可在 Schematic View 其实指定单元即可

  寻找 critical path 需在 Schematic View

  command.log 文件可以当作 script 文件,特别是在你忘记 save your work 时,你可把他当作 script 文件重新执行.切记要从命名 command.log 文件,否则重其 dc 是会被覆盖掉.

  compare_design A B 命令需在 analyze 和 elaborate 之后

  grep library command.log 查找指定格式的文件

  library 这个参数是指你要查的信息，如要查 error,则改为 error

  list -libraries

  remove_all : Edit>Reset

  不想包括的管脚可用一除掉，注意前后都有空格，

  如： set_input_delay 2.0 -max -clock Clk all_inputs() - Clk

  path type 为 max 暗示 setup time,min i 暗示 hold time

  应该用 best case 的 operating conditions 来进行 hold-time checking

  一个无效的路径是指没有时序约束的路径。

  find 命令：

  dc_shell>find port 或 find (port,"*") /*find 与(之间的空格可有可无。( ) 里的逗号可用空格代替*/

  dc_shelll>find (cell,"*DW*") /*list all the cells that contain letters "DW"*/

  dc_shell>find (pin,cba_core/and2a0/*) /*list all the pins of the AN2 gate in the cba_core library*/

  dc_shell>set_load 5 find (net,"CLK") /*place a 5 pF load on the net CLK, 这里""可不要*/

  dc_shell>find (pin /Q) /*list all of the " Q" pins in the desig*/

4.9 Prime Time

Prime Time(简称 PT)时 Synopsys 公司的一个 sign-off 静态时序分析工具。它只支持 TCL 脚本语言。

PT 的调用：

命令行界面：%pt_shell

GUI 界面： primetime

PT 的 setup 文件.synopsy_pt.setup:

set search_path [list ~ jzhu/csmchdlib/lib]

```
set link_path [list {*}  csmc06core,csmc06pad]
```

在这里 set search-path 等价于 DC 的 setup 文件中的 search_path ， set link_path 等价于 link_path 。

PT 的使用命令与 DC 区别不大，由于篇幅的限制，这里不再详细讲解 PT 命令的使用，只以实例说明。

版图前 setup-time 分析脚本：

```
#*************************************************************************
****
#Define top-level design name
    set active_design tap_controller
#Design entry in db format,netlist only-no constraints
    read_db -netlist_only $active_design.db
    current_design $active_design
#Design environment
    set_wire_load large
    set_wire_load_mode top
    set_operating_conditions WORST
    set_load 50.0 [all_outputs]
    set_driving_cell -cell BUFF1X -pin Z [all_inputs]
#Clock specification and design constraints
    create_clock -period 33 -waveform [0 16.5] tck
    set_clock_latency 2.0 [get_clocks tck]
    set_clock_transition 0.2 [get_clocks tck]
    set_clock_uncertainty 3.0 -setup [get_clocks tck]
    set_input_delay 20.0 -clock tck [all_inputs]
    set_output_delay 10.0 -clock tck [all_outputs]
#Timing analysis commands
    report_constraint -all_violators
    report_timing -to [all_registers -data_pins]
    report_timing -to [all_outputs]
```

4.10 设计实例

我们继续以 DDFS 设计为例。

4.10.1 Design Analyzer 操作步骤：

❖ **设置路径及环境变量：**

在自己的工作路径下创建.synopsys_dc.setup 文件，并在里面设置自己的环境变量。

.synopsys_dc.setup 文件：

```
**************************************************************************
search_path = {~ jzhu/csmchdlib/lib}  + search_path
link_library = {csmc06core.db csmcs06pad.db}
target_library = {csmc06core.db}
symbol_library = {csmc06core.sdb csmc06pad.sdb}
synthetic_library = {standard.sldb dw_foundation.sldb}
link_library = link_library +synthetic_library
define_design_lib work –path work;
/*为输出的 VHDL 网表写上使用的 packages*/
vhdlout_use_packages ={ \
                       “IEEE.std_logic_1164”, \
                        “csmc06core.VCOMPONENTS” \
                         }
/*.synopsys-dc.setup1*/
**************************************************************************
```

在根目录下执行环境设置文件：

```
su7>source .cshrc_syn
```

转到自己的工作路径下启动 synopsys 工具。

```
su7>da
```

❖ **读进 design**

1. **Read in 各层的 design**

用 **Read** 读进 PLA 等各种非 HDL 文件；即： File- Read

对 HDL 文件，先用 **Analyze** 读进，转化为中间格式的文件，再用 **Elaborate** 把中间格转化为 synopsys 工具所用的操作符。具体操作为：

File-Analyze

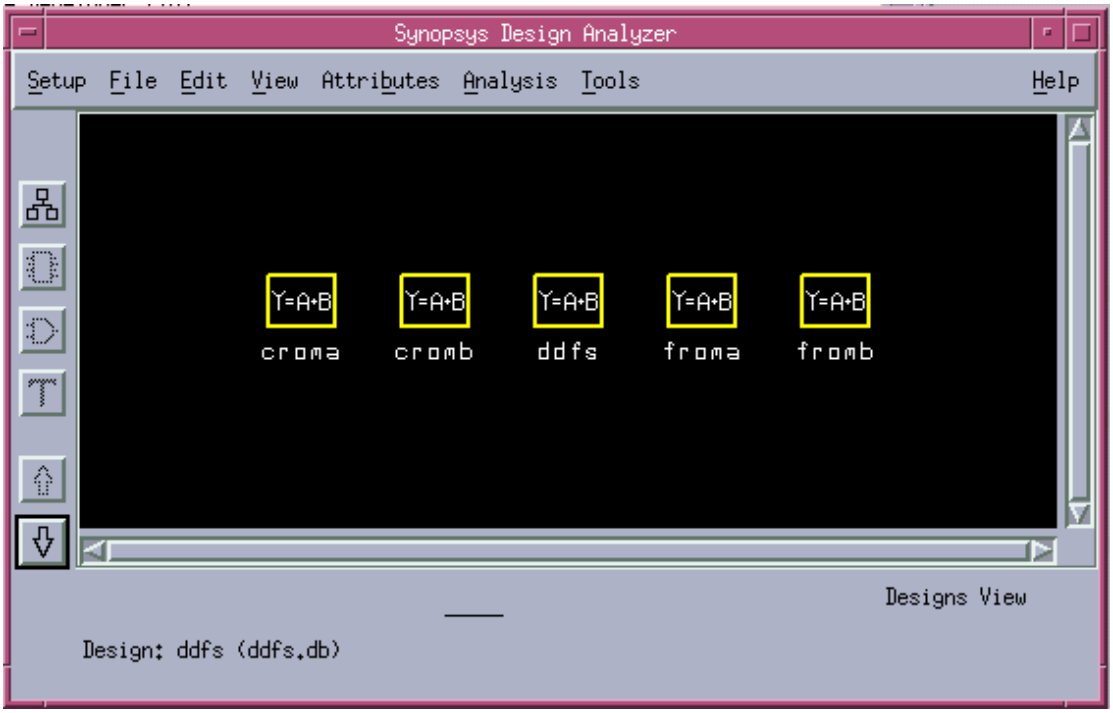
打开 Analyze File 窗口，在存放设计源文件的目录 vhdl/下用鼠标中键选择所有的设计，

在 Library 栏中选择 WORK 库，然后 OK 。

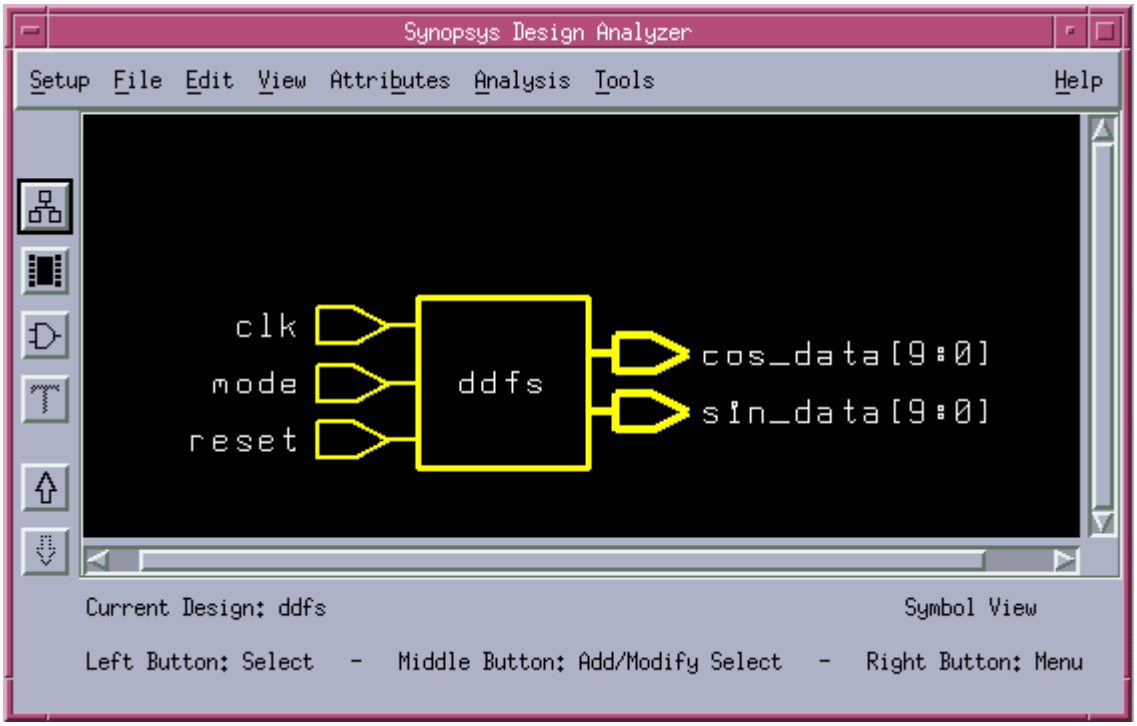
File-Elaborate

在 WORK 库中逐个 elaborate 已分析的设计。注意： analyze 后 DA 窗口中并不显示任何东西。设计在 elaborate 后才在窗口中显示出来。

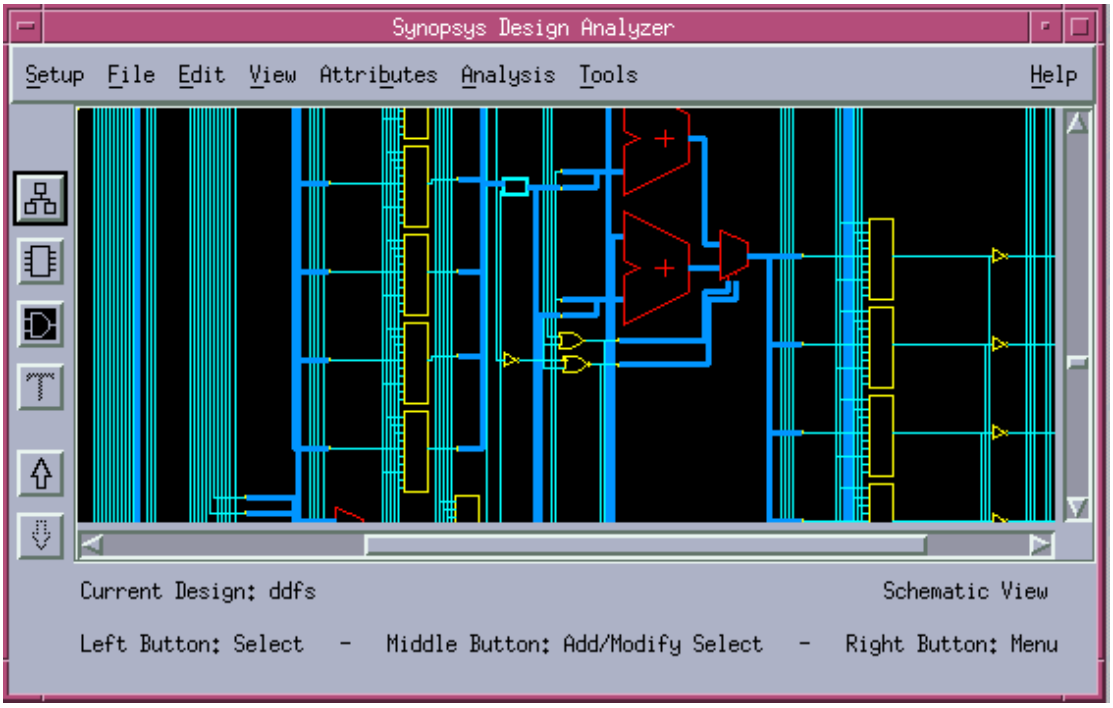
此时 Designs View 为：



其中顶层的 Symbol View 为：



Schematic View 为：



可见此时电路不是门级的。

❖ **设置 Attributes 与 Constraints**

设置属性 attributes:

在 Attribute-Operating Environment -…下设置各种属性。如：

Attribute-Operating Environment-Drive Strength 设置 input strength

Attribute-Operating Environment-Load 设置输出驱动能力

Attribute-Operating Environment-Wire Load 设置 wire load。

Attribute-Operating Environment-Operating Conditions 设置 operating conditions。

•••••••

设置约束 constraints:

1. **设置 Area Constraint**
2. **设置 CLK Constrains**

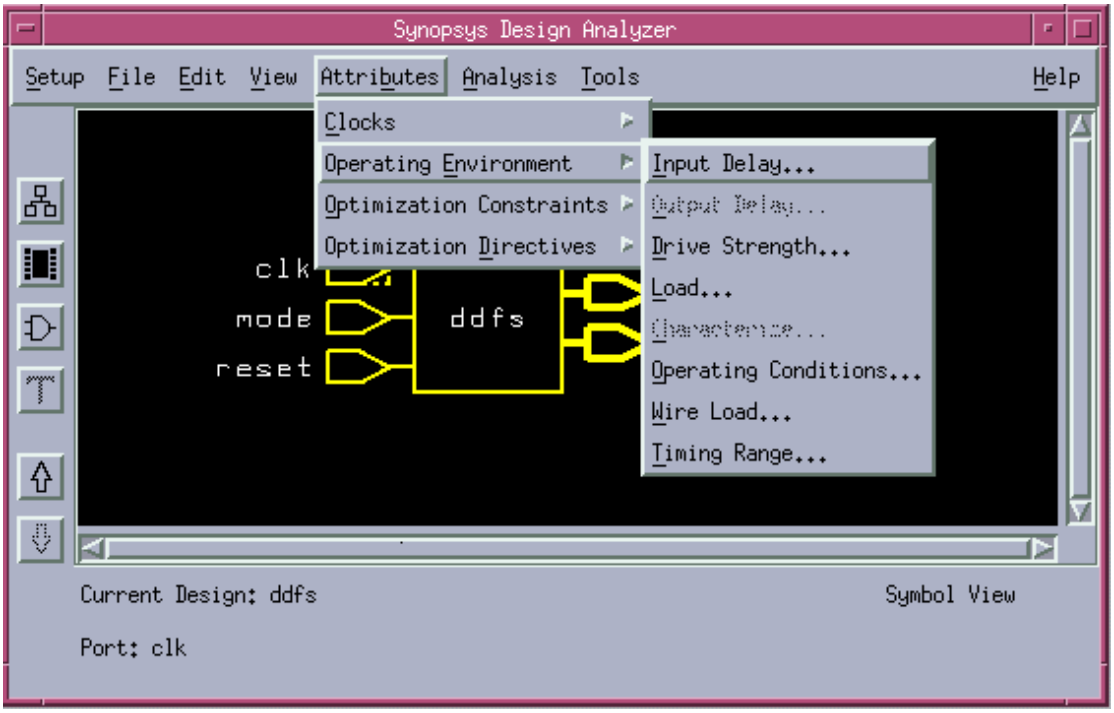
即：选中 CLK， Attribute-Clocks-Specify

3. **设置 delay constraints**

即：选中 output ports 设置 Attribute-Operating Environment-Output Delay

选中 input ports 设置 Attribute-Operating Environment-Input Delay

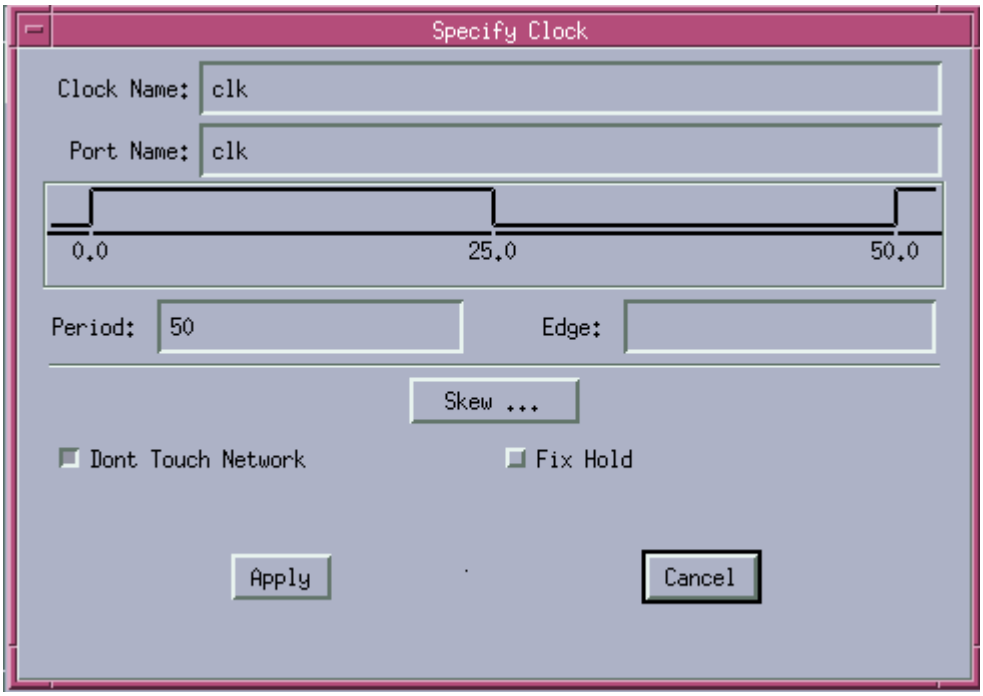
。。。。。。



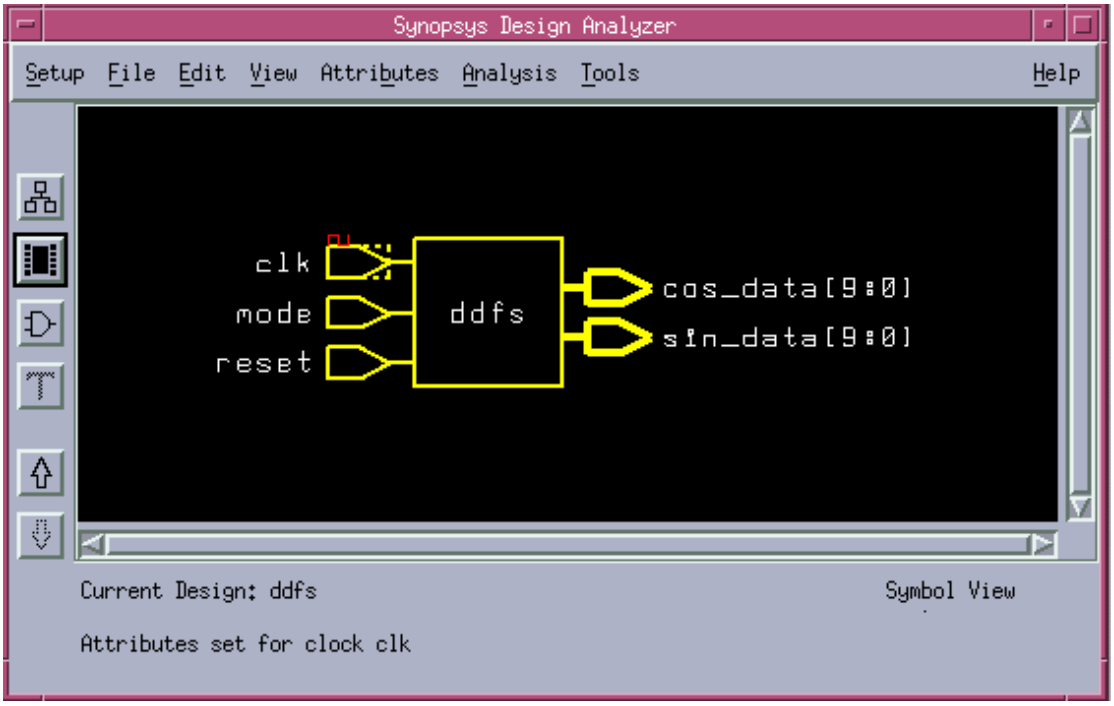
DDFS 设计的输入输出我们都是通过寄存器的，这样 input delay，output delay 就不需设置了。

对于这个 DDFS 设计我们只指定 CLOCK 周期:

在设计的 Synbol View 中选中 clk 管脚，然后 Attribute-Clocks-Specify 打开 Specify Clock 窗口。在窗口中的 period 一栏中写上指定的周期 50 ，这说明你所加的 clock 周期为 20M 。选中 Dont Touch Network，然后 apply，并关掉此 Specfy Clock 窗口。



我们可以看到这时 clk 管脚上出现了一个波形图标。



对于 multiple design instances 我们有三种解决方法：

uniquify

set_dont_touch

ungroup

这些命令在工具栏中没有，我们需用

命令窗口 Command Window 执行。

例如在执行完 uniquify 命令后，我们可能发

现 Design View 窗口中多了几个新的图标。

我们在命令窗口打入 uniquify。

❖ **查错**

选中顶层设计，执行 Analyzing-Check Design，
检查和纠正各种问题。
在这里我们选中 Check Timing 以检查时序属性有无问题。

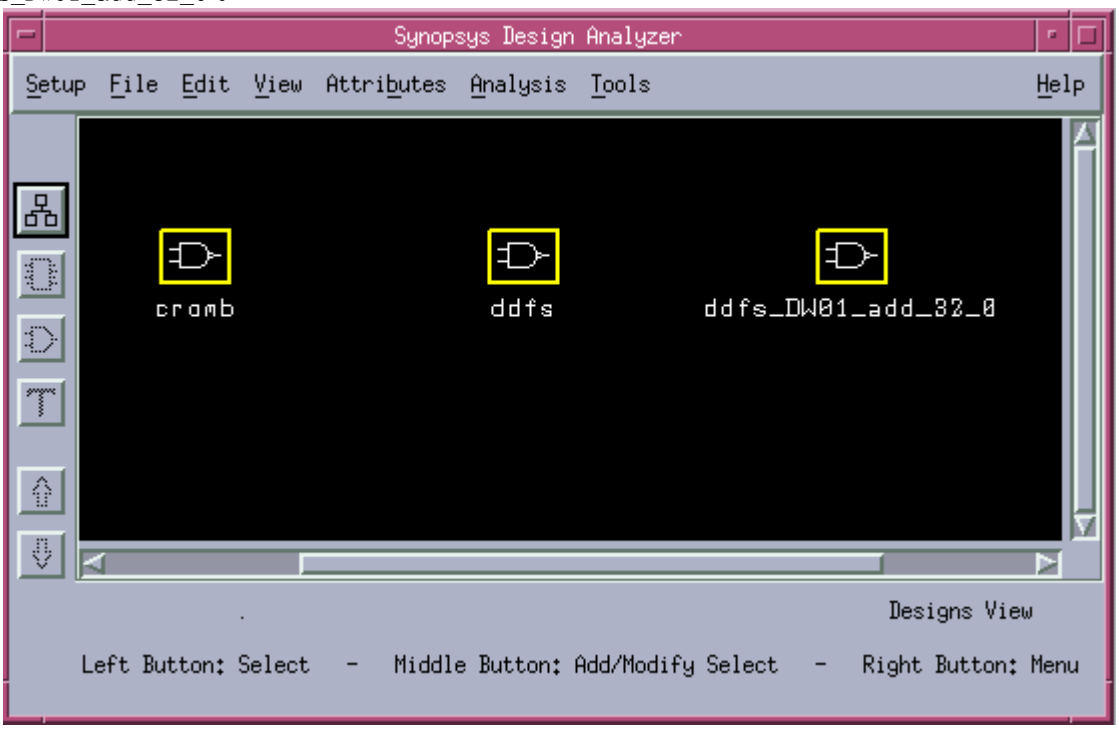
❖ **Compile Design**

选中 DDFS， Tools-Design Optimization ，检查和设置 options ，然后执行 ok,即执行 compile 。

compile 后 the Schematic View of design 就转换成了门级电路。

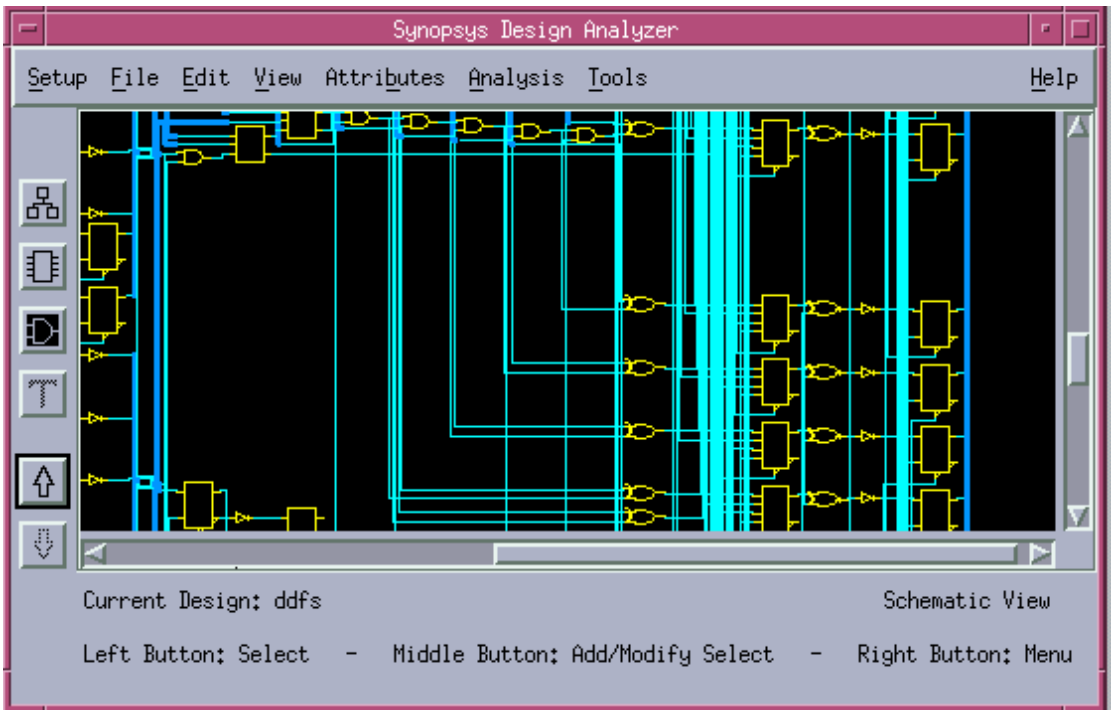
此时的 Designs View 如下图所示：

从中我们可以看到模块的标志已经用门级表示，而且多了一个模块 ddfs_DW01_add_32_0 。



此时 ddfs 的 Schematic View 为：

我们可以看到 ddfs 的 Schematic View 已经转换为了门级电路。



❖ **产生 Report ，分析设计结果。**

1. 产生各种 Attribute Report

2. 产生各种 Analyzing Report

其中 Timing Report 可检查 critical path

Report-Timing，然后在 Analysis Reports 中选中 constraints， timing,再在 set options 中选中 All Violations。 选择输出到文件中，在 File 一栏中打入你的文件名,然后 Apply。

产生报告文件后，检查是否有 violations 存在。如果没有存在，且时序都满足要求，则综合优化完成。

❖ **输出网表文件和 SDF 文件**

完成综合优化后，我们选中顶层设计 ddfs ，然后把设计保存为 db 格式放在 mapped/中，切忌别忘了选中 Save All Designs in Hierarchy。

为了在VSS中进行综合后仿真,我们需把设计存为VHDL格式(不要忘了选中Save All Designs in Hierarchy)，同时输出时序延时信息的 SDF 文件。输出此 SDF 文件： File-Save Info-Design Timing ，打开 Save Timing Information 窗口后，选中 SDF V2.1 格式，并在 Output File Name 里打入你的文件名，然后 ok 即可。

❖ 为了在版图工具 SE 中进行自动布局布线，我们要把设计存为 Verilog 格式的网表(同样要选中 Save All Designs in Hierarchy)。同时我们还需输出约束文件.sdf 文件。此 SDF 文件与前者不同。它的保存方法为： File-Save Info-Constraints 打开 Save Constraint Information 窗口后，选中 SDF V2.1 格式。并选中 Max Path Timing Constraints, Net Constraints, Design Hierarchy 。然后 ok 即可。

4.10.2 dc_shell 命令行批处理

执行批处理前我们需准备好脚本文件：约束文件 constraint.scr,执行文件 runit1.scr。具体文件内容见附录。

检查约束文件 constraint.scr ：

```
dc_shell -f scripts/constraint.scr -syntax_check
```

检查执行文件 runit.scr ：

```
dc_shell -f scripts/runit1.scr -syntax_check
dc_shell -f scripts/runit1.scr -context_check | tee context.log
```

运行脚本综合你的设计:

```
dc_shell -f scripts/runit1.scr | tee runit.log
```

执行完批处理文件以后，检查报告文件。如果报告文件中没有 violations，时序满足要求，则综合工作就完成了。否则，你需修改约束条件或尝试其他方法重新进行综合。

第 5 章　综合后仿真

5.1　工具介绍

RTL 级的源设计综合成门级电路后，我们需要对这生成的门级网表进行门级仿真，以验证功能是否正确。Synopsys 的 VSS 工具提供了门级仿真的仿真机制。VSS 工具的介绍请参照系统行为级仿真中的介绍。对于门级仿真， VSS 的操作没什幺大的区别。门级电路仿真与系统行为级仿真的区别在于网表的不同以及所加的工艺库的不同。

5.2　工艺库的产生

5.2.1　工艺库介绍

综合后仿真与系统行为级仿真的主要区别是它需要加入库的延时信息。即在综合后的源文件上加上时序工艺库的说明。综合后仿真所需的时序工艺库可以是 FTGS(full-timing gate-level simulation)库，也可以是 VITAL(VHDL initiative toward ASIC library)。

仿真工具用到的工艺库模型有五种：

FTBM(full-timing behavioral model)：提供非常精确的模型，但是有最慢的仿真速度。

UDSM (unit-delay structural model)：不考虑时序，用于功能仿真.组合单元实例有 1ns 的 rise/fall delay.时序单元实例有 2ns 的 rise/fall delay 。即它用一个常数的时序延时信息来模拟各个单元的功能。它有着比 FTBM 更快的仿真速度，但是时序精度不及。冒险监测，时序约束检查和 X-Generation 都不能执行。

FTSM (full-timing structural model)：用于功能验证和普通的时序验证。它用精确的传播延时模拟每一个单元。延时模型包括 0 延时，功能网络上的传输线延时和管脚间延时。它的仿真速度比 UDSM 略慢，时序的精确度比 FTBM 略差。只能进行有限的时序违约检查，如最小的脉冲宽度，建立时间，保持时间，和恢复时间等检查。时序约束违约作为警告报告。它不能执行冒险监测和 X-Generation 。

FTGS (full-timing gate-level simulation model)：给门级设计提供了快而又非常精确的仿真模型(和 FTBM 一样)。延时模型包括传输线延时和管脚间延时。除了警告,可选的输出 X 也被预定为时序约束违约和电路的冒险。这使得 FTGS 对 sigh-off 验证和细节验证一样有用。

VITAL(VHDL initiative toward ASIC library)：给 ASIC 库提供一个 VITAL 适应仿真模型.它的延时模型包括传输先验时和管脚间延时.除了警告，可选输出 X 被安排为时序约束违约和电路冒险。

各种仿真库模型的用法比较：

Table 3-1　Design Stage and Simulation Engine Required

Design Stage	Activities	Simulation Engine	Simulation Models
Behavioral VHDL validation, RTL VHDL validation (unit test)	Explore functional alternatives interactively, debug source code.	Interpreted, compiled	High-level behavioral, RTL
RTL VHDL validation	Refine design, test "corner cases," prepare for synthesis, verify ASIC in context of entire system.	Compiled, interpreted	RTL, FTSM (with XP), UDSM (with XP)
Gate-level verification	Fast, accurate batch simulation to verify timing, achieve sign-off with silicon vendor.	Optimized gate-level	FTGS

如果芯片制造商没有给你提供仿真所需的工艺库，你可以从工艺库的.lib 或.db 格式产生你所需的仿真所需的工艺库。在这里我们将主要介绍 FTGS 和 VITAL 库的产生。先前我

们实验室不能进行综合后，版图后等时许仿真就是因为没有时序的 VITAL 库。

5.2.2 综合库转换为仿真库

Synopsys 的库分析器可以帮你把综合库(.db)转换为加密的 VHDL 仿真库。 VSS 环境中的库分析器如图 5.1 所示：

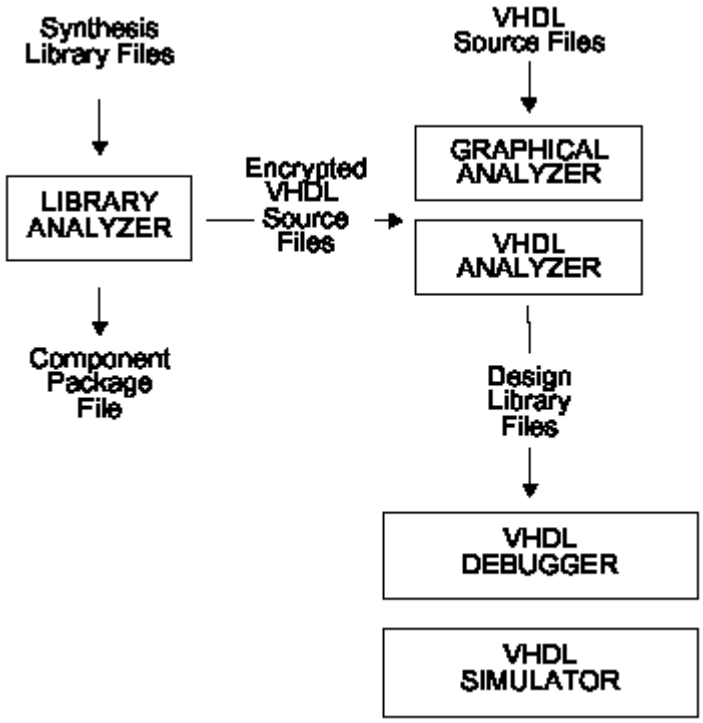


图 5.1 VSS 环境中的库分析器

库分析器的输入文件为综合的库文件(.db)，输出文件为各种仿真所需的 VHDL 工艺库： FTBM， USDM， FTSM， FTGS， VITAL。

FTBM， USDM， FTSM， FTGS 模型的输出文件是：

libname_model.E(加密后的 VHDL 库文件)

libname_component.vhd(器件文件)

VITAL 模型的输出文件是：

libname_VITAL,vhd.E

libname_Vtables.vhd.E

libname_Vcomponents,vhd(器件文件)

调用库分析器 liban:

% liban [*options*] *lib_name*.db

lib_name.db 为综合的库文件。

options 包括你想生成的时序模型，冒险处理算法以及是否打开 X-Generation.

库分析器能产生的时序模型有 5 种： FTBM， UDSM， FTSM， FTGS， VITAL。各种模型的含义见前面的说明。各种时序模型选项为：-arch ftbm | udsm | ftsm | ftgs | vital

确省为 ftbm 模型。

冒险处理算法：-hazard spike | glitch | inertial | transport

使用 X-Generation: -xgen

分析加密的 VHDL 库文件：

例如：

% vhdlan -w ASICLIB asiclib_FTSM.vhd.E asiclib_components.vhd

这样生成后 vhdl 库文件在 VHDL 设计源文件中的调用为：

library asiclib;

use asiclib.components.all;

实例：

编译 FTGS library:

liban –arch FTGS –output ???_FTGS 库名.db > ???.log

vhdlan –spc –w ???_FTGS 库名_cmponents.vhd

vhdlan –w ???_FTGS 库名_FTGS.vhdl.E

5.2.3 工艺库.lib 格式转换为仿真库

如果你没有综合库(.db)，则你也可以用库编译器从最初的工艺库.lib 格式产生你仿真所需的仿真库。这库编译器(Library Compiler)能产生四种类型的仿真库： UDSM ，FTSM， FTGS， VITAL。它们的介绍详见前面。

VHDL 仿真需要后注释时序信息到你的设计中去。注释时序信息到模型中有三种方法：

1. 通过使用外部的时序计算器(如 DC)，直接后注释或前注释到设计的每一个单元中。
2. 使用 VHDL 标准配置块(standard configurationblock)进行后注释。你在 VHDL 代码的配置块中加入延时信息，然后重新编译。
3. Synopsys 的 VSS 工具也可用 SDF 文件来作为后注释。

下面我们将只介绍 FTGS 和 VITAL 库的产生，其他库的产生方法雷同，这里不在一一介绍。

FTGS 库的创建(图 5.2)

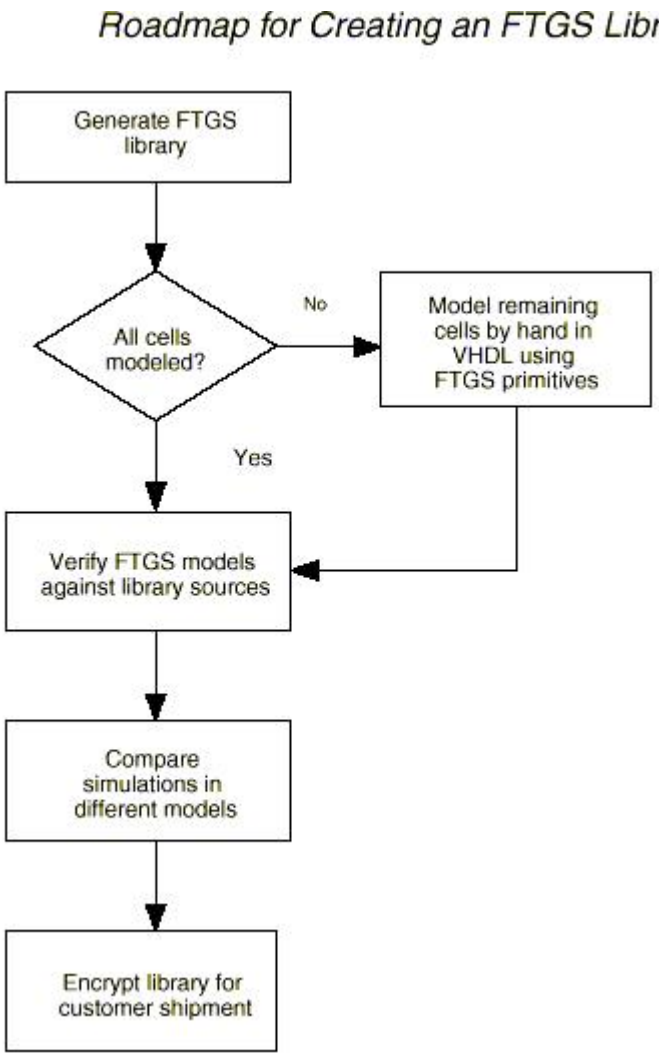


图 5.2 FTGS 库的创建

1. 产生 FTGS 库：(图 5.3)

在编写 FTGS 模型代码时，你必须确定.lib 文件中的描述的时序特性和.sdf 文件中的一样。

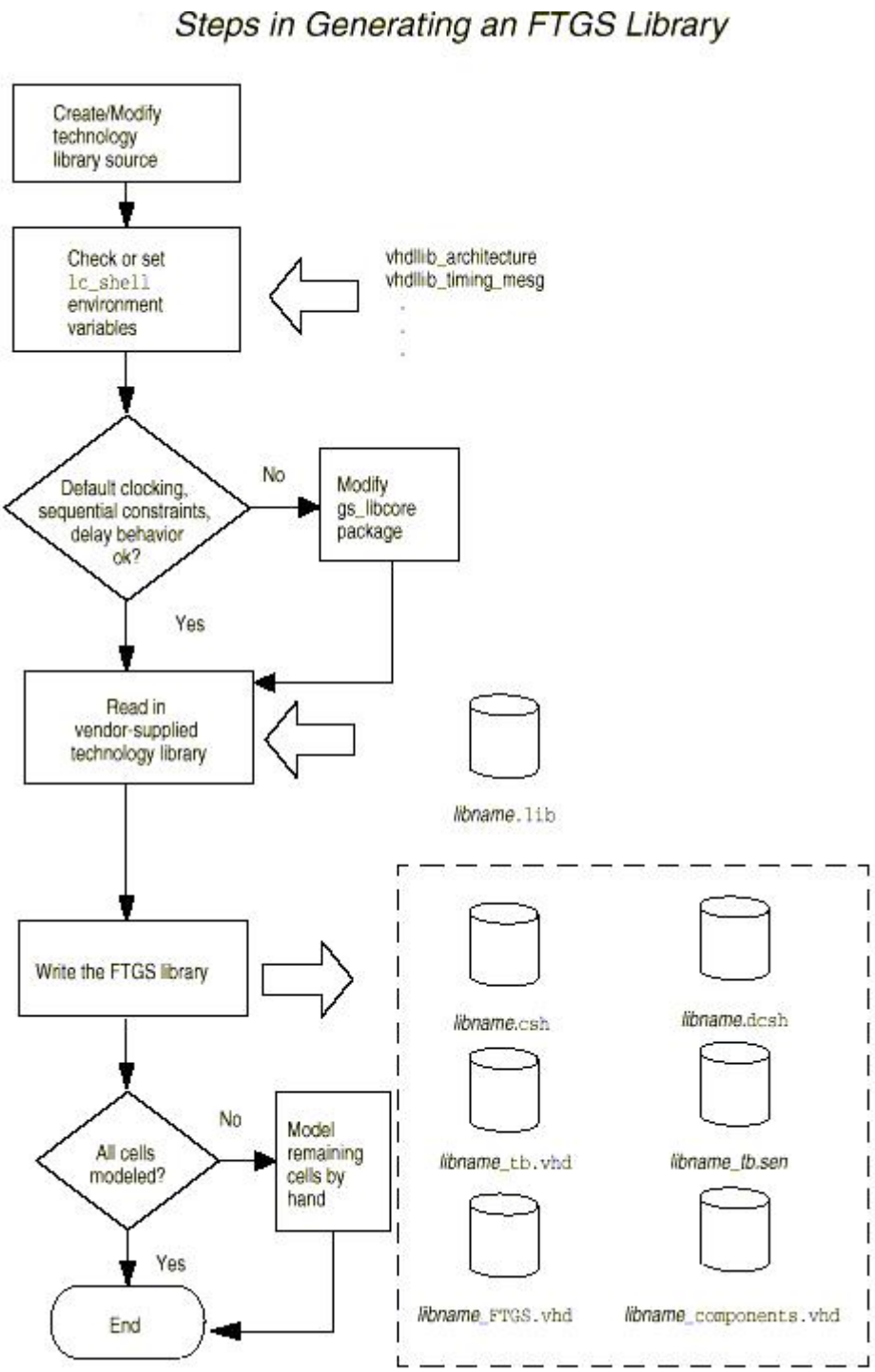


图 5.3 产生 FTGS 库

1) 调整工艺库的源文件(可选)

编写依赖状态的时序特性

工艺库的条件时序检查

2) 检查或设置环境变量

vhdllib_architecture variable vhdllib_timing_mesg
vhdllib_timing_xgen vhdllib_glitch_handle
-……

3) 调整 gs_libcore 包 (可选)

read_lib vendor_library_source_name.lib

4) 写 Synopsys FTGS VHDL 库

write_lib -format vhdl vendor_library_source_name

5) 确定唯一的包名称(可选)

2 。验证 FTGS 模型(图 5.4)

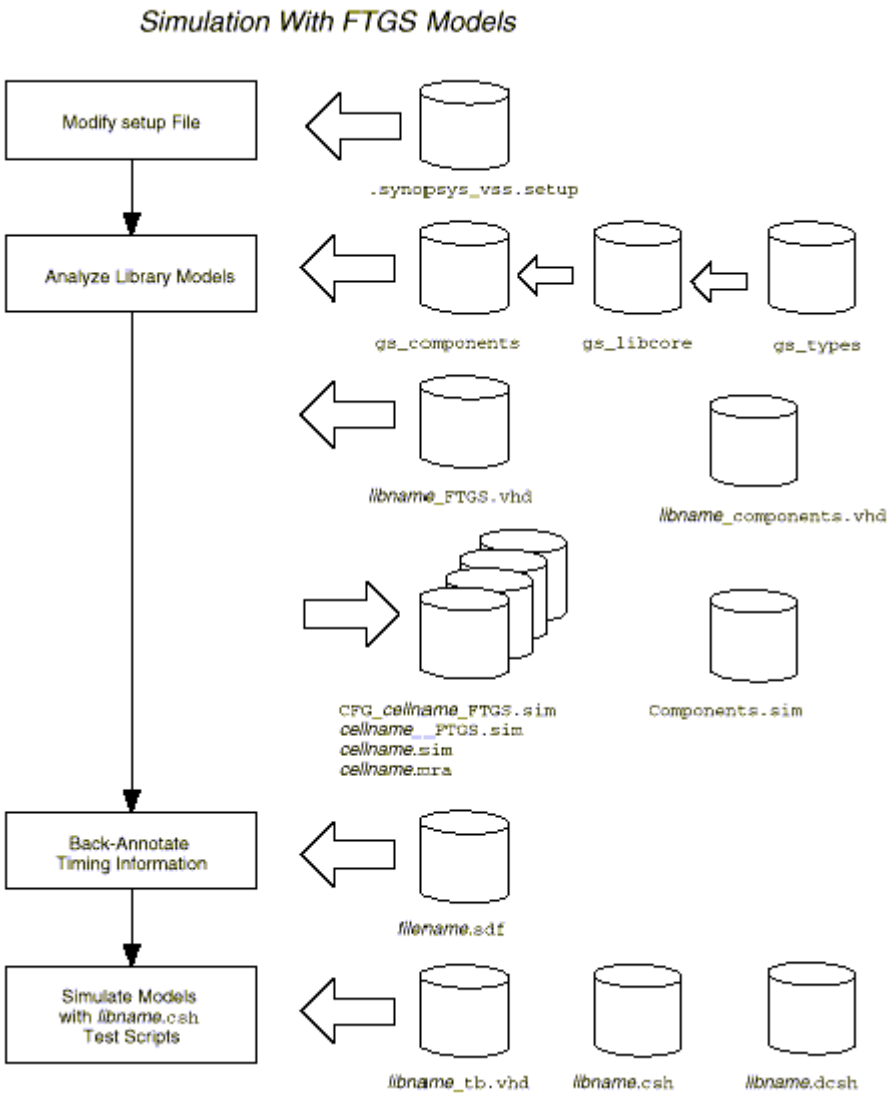


图 5.4 FTGS 库的验证

1) 调整 setup file,使 vhdllib_logical_name 变量设置的 logicalname 付值为分析了的仿真文件存放的子目录。

分析工艺库:

在模型上运行 vhdlan 去产生可执行的仿真代码。具体的语法规则请参照 VSS 手册。

2) 后注释时序信息

vhdlsim -sdf sdf_filename design_name .

你必须在命令行中后注释，而不能使用仿真器的 include 命令。

3) 在 VHDL 代码的配置块(configuration block)中加入延时语句。仿真模型

3 。比较不同模型的仿真结果

4 。加密文件(可选)

使用库分析器(liban)呵 encrytp_lib 命令产生 VHDL 库文件。

VITAL 库的创建

创建 VITAL 库的方法和 FTGS 库的方法雷同，只是输出文件不同。这里将只给出流程。(见图 5.5)

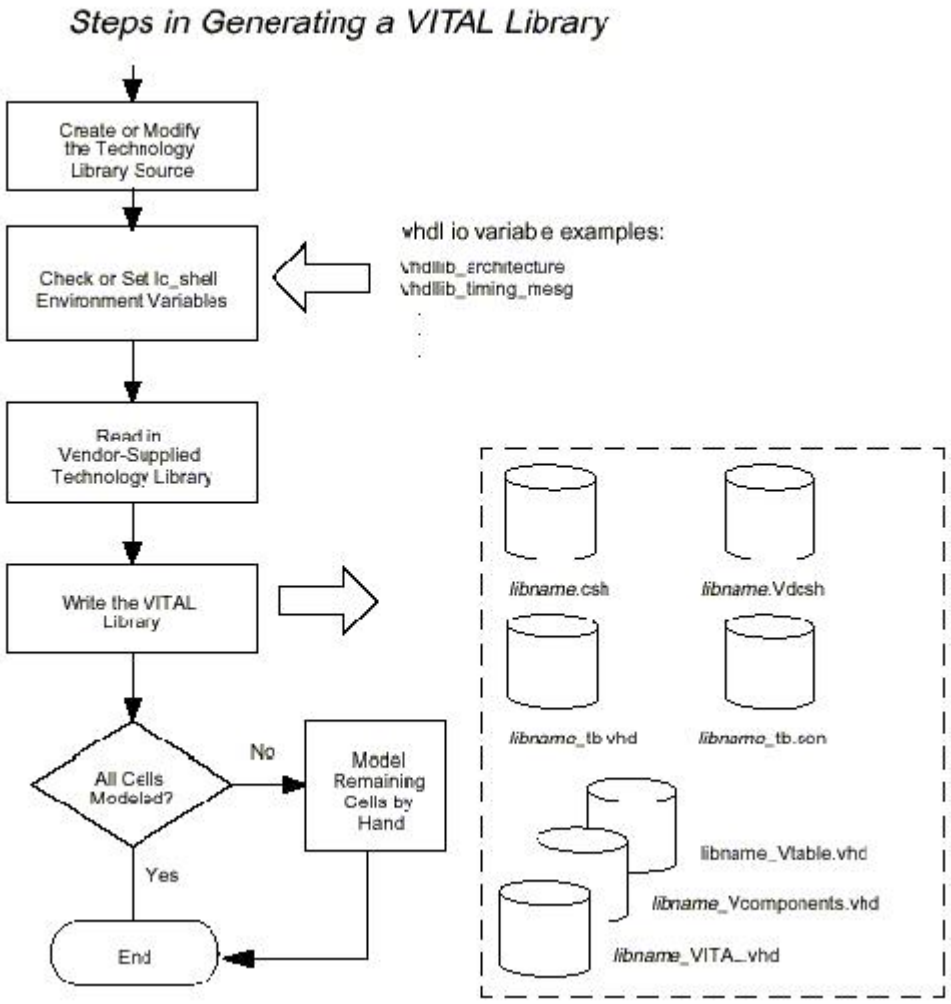


图 5.5 VITAL 库的创建

实例：

生成 VITAL 库：

1. 创建你的目录和目录文件：

   目录文件： run.inc /*运行 simulator 的批处理文件；至少包含 run, quit 两个命令*/

   .synopsys_vss.setup： 含： vhdllib_logical_name : LIBVUOF

   Set the SDF naming file for VITAL model SDF back-annotation.

   目录： LIBVUOF, WORK
2. %lc_shell
3. lc_shell> read_lib std_cell.lib
4. lc_shell> vhdllib_architecture=vital
5. lc_shell> vhdllib_tb_x_eq_dontcare=false
6. lc_shell>vhdllib_tb_compare=5 /*不设的话 testbench 不产生，各个值的含义请参

照在线帮助*/

7. lc_shell> write_lib -f vhdl std_cell
8. 运行 testbench，仿真库文件

library.csh

5.3 综合后的门级仿真

综合后的门级仿真在工具或命令的操作上与系统行为级的仿真没什么区别，故这里对命令的执行不再详诉，只给出简单的流程。

1. 修改综合好的 VHDL 网表文件

a) 给 VHDL 网表文件加 configuration；

假设设计文件为 design.vhd ,entity 的名字为 design1,architecture 名字为 behavior，则加 configuration 为:

```
configuration cfg_design1 of design1 is
for syn_behavior
    end for;
end cfg_design1;
```

b) 在使用这个库的 VHDL 原文件开头指定时序库文件:

```
library asiclib;
use asiclib.components.all;
```

2。设置 setup 文件

在 setup 文件里映射泥的 VHDL 文件里 的 logic library 到 the VITAL ASIC library 目录

在 setup 文件中设置 the simulation timebase 使之与 ASIC vendor library 一致

3。产生 testbench

什 testbench 文件为 tb_design,则在 tb_design.vhd architecture statement 中加 the using entity name:

```
for uut: design1 use entity work.DESIGN1(SYN_behavior)
```

4。创建设计库

5。分析各种源文件：

调用 VHDL Analyzer 分析 VHDL 库文件:

例如： % vhdlan -w ASICLIB asiclib_FTSM.vhd.E asiclib_components.vhd

asiclib_components 对应于使用这个库的 VHDL 原文件开头指定的时序库文件

```
library asiclib;
use asiclib.components.all;
```

调用 VHDL 分析器分析各个修改后的 VHDL 网表文件

6。调用 VSS 仿真器进行门级仿真:

vhdlsim -i filename -sdf_top /testbench_entity_name/uut -sdf design.sdf design_unit

其中 design_unit 为 testbench 的 configuration 的名字

注意：

1. setup 文件中指定的 design_lib 一定要与你创建的库名一样，大小写是有区别的。
2. 目录中不能有同字母，不同大小的目录，否则他会无法删除和创建
3. testbench 文件中的 wait 命令中时间单位前 必须有空格
4. 在 setup 文件中设定 timebase 可以修改仿真精度,或在 vhdlsim 命令中用-t base_time 选项指定
5. testbench 里的 uut 后面的名字是最顶层的 design 的名字

5.4 设计实例

修改 DC 生成的网表文件：

为每一个 entity 加上 configuration 说明：

configuration *configuration_name* of *entity_name* is

```
    for architecture_name
    end for;
end configuration_name ;
```

为每一个 package 和 entity 加上使用的仿真 VITAL 库的说明：

```
library csmc06core;
use csmc06core.VCOMPONENTS.all;
```

这里因为我们已经在 DC 的 setup 文件.synopsys_dc.setup 文件里加了加这些语句说明的命令，所以不必加了。如果没有则必须加上。

创建 setup 文件

在工作目录上我们需创建 VSS 的 setup 文件，其内容如下；

```
WORK > DEFAULT
DEFAULT : work
csmc06core : ../csmchdlib/lib
TIMEBASE : ps
```

创建设计库

```
% mkdir work
```

分析各种源文件

分析各种源文件的命令，我们写在了一个脚本文件 analyze2.sh 里,内容如下：

```
**************************************************************************
#!/bin/csh -f
if(-d work) then
rm -rf work
echo "------"
echo "Creating work directory"
mkdir work
endif

vhdlan -optimize -event -w xsmc06core \
        ../csmchdlib/vital/csmc06core_Vcomponents.vhd \
        ../csmchdlib/vital/csmc06core_Vtables.vhd \
        ../csmchdlib/vital/csmc06core_VITAL.vhd

vhdlan -event \
        ddfs.vhd \
      vhdl/DDFS_TB.vhd
**************************************************************************
```

在命令行你执行此 analyze2.sh

```
% ananlyze2.sh
```

启动 VSS 工具进行仿真

仿真的命令文件同系统行为级仿真一样，放在 simfile 文件里，其内容最基本的为：

```
trace -wif -waves /DDFS_TB/*'signal
run 100000000
```

启动 VSS 工具开始仿真：

```
% vhdlsim -i simfile -sdf_top ddfs_tb/uut -sdf ddfs.sdf TESTBENCH_FOR_DDFS
```

其仿真结果为：(见图 5.6)

从此波形中我们可以看到，信号已经有了延时。对比系统行为级仿真，系统行为级仿真时是没有延时的，因为我们没有加入有延时信息的库，同时也没有加入延时信息的 SDF 文件。这门级仿真的延时由库的延时和 DC 生成的延时时序 SDF 文件共同产生。

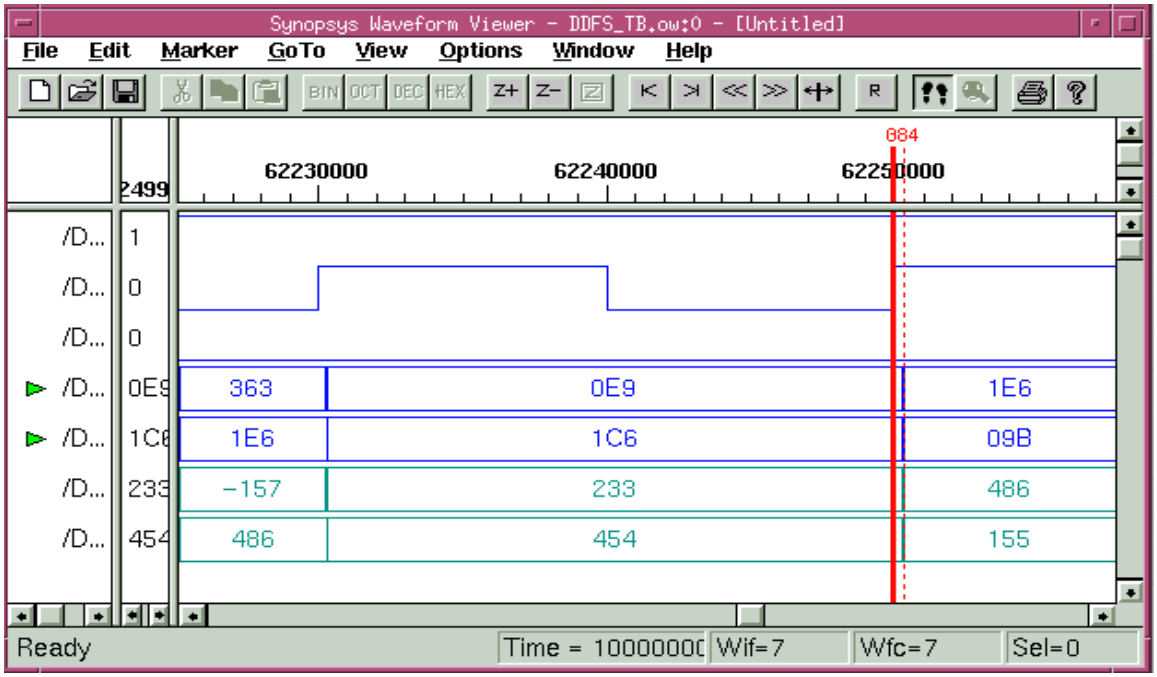


图 5.6 综合后仿真的仿真结果

第6章 Formal Verification及其其他辅助工具

6.1 Formality

Formal Verification就是通过比较两个设计功能是否相同来验证电路的功能，不仅提高了验证速度，更重要的是它摆脱了工艺的约束和仿真test bench的不完全性，更全面的检查了电路的功能。Formality是Formal Verification的工具。

1. 启动Formality: % fm_shell
   fm_shell >
2. 读进共享工艺库：fm_shell> read_db cba_core.db
3. 产生reference设计准备验证：
   产生新container：fm_shell> create_container ref
   读进门级网表到containter：fm_shell> read_db synth.db
   建立reference 设计：fm_shell> set_reference_design ref:/WORK/mR4000
   link reference 设计：fm_shell> link $ref
4. 读入调整了的reference 设计：fm_shell> read_verilog -c impl -netlist clk_insert1.v
   fm_shell> set_implementation_design impl:/*/mR4000
   fm_shell> link $impl
   fm_shell> current_design $impl
   fm_shell> set_constant test_se 0
5. 验证调整了的reference设计：fm_shell> verify
6. 调试失败了的验证：
   报告失败位置：fm_shell> report_failing_points
   诊断运行： fm_shell> diagnose
   报告错误 candidates ： fm_shell> report_error_candidates
   保存： fm_shell> save_session -replace -full fm_shell_session
   推出 Formality ； fm_shell> exit

6.2 ECO Compiler

使用 ECO Compiler 重用先前一综合和布局布线了的网表，大大加快了设计修改速度，提高了设计重用。 ECO 通常在 design cycle 的末期， post-layout 阶段使用。只对 functional requirement 敏感，如果只是改变 constraints,function 没变，则 ECO 使用整个老网表。

6.2.1 ECO Compiler 的使用流程(如图 6.1)：

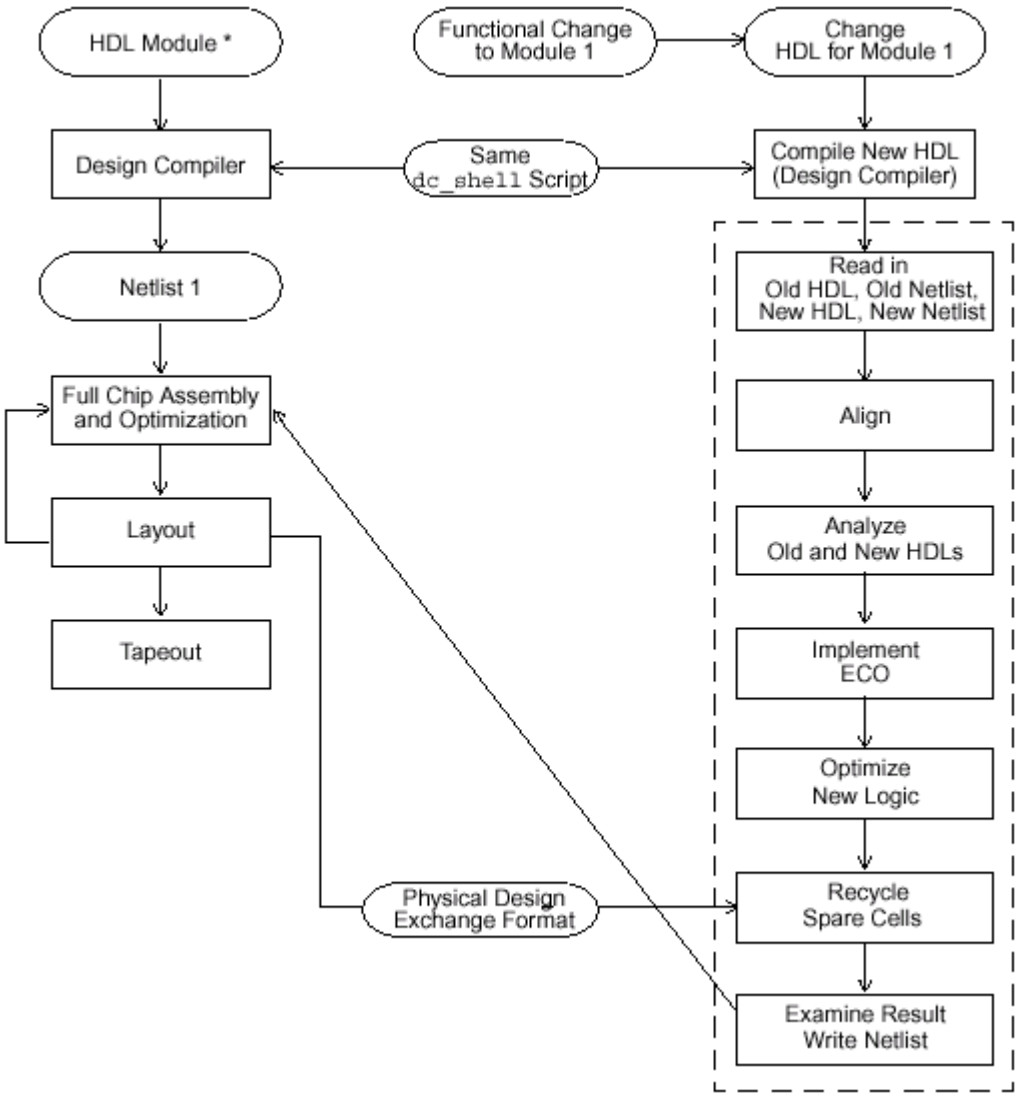


图 6.1 ECO Compiler 的使用流程

1. 调用 DC ，使用编译了旧 HDL 设计的同一脚本文件编译新的 HDL 设计，并保存设计为.db 格式；
2. 读入旧 HDL ， 旧网表， 新 HDL ， 新网表四个设计；
3. 使用 ECO Compiler 检查四个设计的排列；
4. 检查新旧 HDL 间的功能变化；
5. 产生 ECO 网表并检查结果；
6. 使用 DC 优化新的逻辑以满足约束条件；
7. 用旧网表中的未重用单元和共享单元反复替代 ECO 网表中的新逻辑；
8. 检查结果并输出网表

注意点：当新旧网表中寄存器阵列的寄存器类型有不同时，为使 eco_implement 继续下去，且作为新逻辑的一部分综合这寄存器，设置变量 eco_allow_register_type_different 为 true. ECO Compiler 不支持非阵列的黑盒子.一旦侦察到就停止.

6.2.2 设计实例

```
/*读入设计*/
%dc_shell
dc_shell> eco_instance_name_prefix = "ECO"
dc_shell> compile_instance_name_prefix = eco_instance_name_prefix
dc_shell> read old_hdl.db
```

```
dc_shell> current_design old_hdl.db:uart
dc_shell> link
dc_shell> read new_hdl.db
dc_shell> current_design new_hdl.db:uart
dc_shell> link
dc_shell> read old_netlist.db
dc_shell> current_design old_netlist.db:uart
dc_shell> link
dc_shell> read new_netlist.db
dc_shell> current_design new_netlist.db:uart
dc_shell> link
/*设计排列*/
dc_shell> eco_current_design_pair -old_hdl old_hdl.db:uart \
-old_netlist old_netlist.db:uart
dc_shell> eco_align_design
dc_shell> eco_current_design_pair -new_hdl new_hdl.db:uart \
-new_netlist new_netlist.db:uart
dc_shell> eco_align_design
dc_shell> eco_current_design_pair -old_hdl old_hdl.db:uart \
-new_hdl new_hdl.db:uart
dc_shell> set_eco_unique second cell "baud64_reg"
dc_shell> set_eco_unique second pin "rcv/csrc"
dc_shell> eco_align_design
/*分析 HDLS*/
dc_shell> eco_current_design_pair
dc_shell> eco_analyze_design
dc_shell> remove_design old_hdl.db:uart -hier
dc_shell> remove_design new_hdl.db:uart —hier
Aligning the Netlists:
dc_shell> eco_current_design_pair \
-old_netlist old_netlist.db:uart \
-new_netlist new_netlist.db:uart
dc_shell> set_eco_unique second cell "baud64_reg"
dc_shell> set_eco_unique second pin "rcv/csrc"
dc_shell> set_eco_reuse txs txs
dc_shell> set_eco_align cell add_71 add_71
dc_shell> eco_align_design
/*执行 Initial Engineering Change Order 并产生结果报告*/
dc_shell> eco_current_design_pair
dc_shell> eco_implement -preserve_obsolete
dc_shell> write -hier -f db -out db/eco_implemented.db
dc_shell> remove_design -d
/*Recycling Spare Cells and Obsolete Cells, andFinishing the ECO*/
dc_shell> read old_c74.db
dc_shell> remove_design uart -hier
dc_shell> read db/eco_implemented.db
dc_shell> current_design c74
dc_shell> link
dc_shell> read_clusters -design c74 scripts/c74.pdef
dc_shell> current_design c74
dc_shell> eco_recycle_verbose = true
```

```
dc_shell> eco_recycle -post_tapeout
dc_shell> write_clusters -design c74 -output reports/new_c74.pdef
dc_shell> write -hier -out db/eco_c74.db
dc_shell> write uart -hier -out db/eco_recycled.db
dc_shell> remove_design -d
dc_shell> read old_c74.db
dc_shell> link
dc_shell> read eco_c74.db
dc_shell> link
dc_shell> eco_current_design_pair -old_netlist old_c74.db:c74 \
-eco_netlist eco_c74.db:c74
dc_shell> eco_netlist_diff > reports/c74_net.diff
dc_shell> quit
```

6.3 Floorplan Manager

Floorplan Manager 是一个处理逻辑设计环境与物理设计环境间数据交换的工具，它提供了 Synopsys 的 links-to-layout 方法学。 links-to-layout 方法学降低了综合与版图的反复。注意:Floorplan Manage 不是一个 floorplanner 。

6.3.1 Floorplan Manager 设计流程(如图 6.2)

6.3.2 Links-to-Layout 方法学流程

1. Develop physical information:

   使用 set_load 和 set_resistance 命令后注释参数。

   使用 read_sdf 命令后注释时序信息

   使用 create_wire_load 命令产生 wire load models

   使用典型的 constraitns 快速综合 HDL 设计，使用 compile map effort (low,medium,high), 使用 vendor wire load models

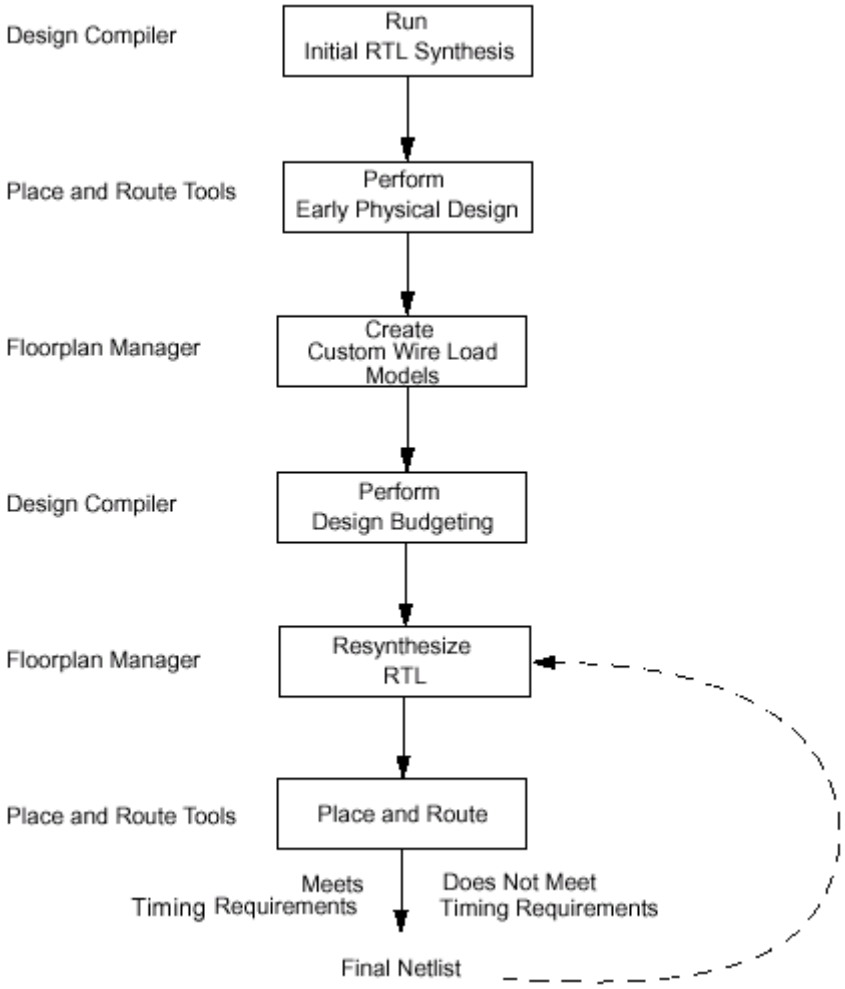


图 6.2 Floorplan Manager 的使用流程

2. Performing detailed implementation
3. Gain timing closure

后注释：用 read_clusters 命令去读和后注释 PDEF 文件，用 read-sdf 命令读 SDF 文件. 用 set_load 和 set__resistance 读 net parasitics.

用 reoptimize_design 命令重新综合设计，综合完后输出 netlist 和 PDEF 文件（ write_clusters 命令）

必要时使用 define_name_rules 和 change_names 命令改变 net 或 port names

在 Floorplan Manager 和 layout tools 中传递的文件:网表文件(EDIF， VHDL， Verilog) SDF,Parasitic back-annonatation_(optionally write SPEF).Physical Design Exchange Format(PDEF）

Design constraints 用 write_constraints 命令产生，

对于 write_constraints 的选项：如果设计太大，可使用-cover_design 和-cover_nets (注意两者不能同时使用)

annonatating delay information:用 write_sdf 命令；读用 read_sdf 命令

写 parasitic information 用 write_parasitics 命令(可选)(floorplan manager 产生) parasitic back-annotation 由 layout tools 产生， load parasitic back-annotation information 到 floorplan manager 用 include filename 命令。验证用 check_design –post_layout 命令。

产生 wire load models 用 create_wire_load 命令,选择 wire load models 用 set_wire_load 命令

第 7 章　　版图生成

7.1　工具介绍

在这个流程中我们 floorplan 和布局布线采用的工具是 Silicon Ensemble，简称 SE。

Silicon Ensemble 是一个自动版图生成系统。它执行基于标准单元集成电路和子电路设计的平面布置(floorplan),单元放置和互连线的连接。用户既可以使用 ANSI(命令行界面)也可以使用窗口界面。 Silicon Ensemble 包含几个功能块：平面布置器(floorplanners)，布局器(placers)，布线器(routers)，系统支持工具(system support tools)。

floorplanners: 为器件放置预备行空间。

placer: 包括基于连通性把单元分组，自动放置单元，用 Qplace 放置单元，放置单元微调，布局优化等命令。

router: 使用全局(global)，最后(final),能量(power)，时钟，和特定布线器进行布线。当然你也可使用 WarpRoute 选项进行快速全局和最后的布线。

system support tools: 让你进行读写数据，验证数据，和手工放编辑单元的放置及连线。

Silicon Ensemble 支持多层金属布线(Mutilayer metal routing),混合库的支持(Mixed library support)， Correct-by-construction layout,自动参数调节(APT),技术更改选项(ECO)。

7.2　自动布局布线流程(见图 7.1 所示)

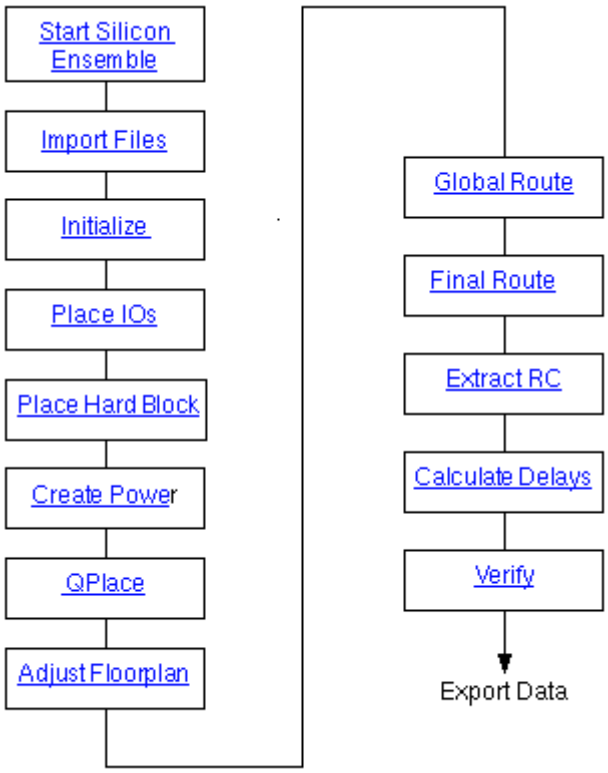


图 7.1 自动布局布线流程

7.3　SE 前准备

开始启动 SE 工具进行自动布局布线之前，我们需要准备好设计所需的工艺库，设计数据， setup 文件。

7.3.1　工艺库

LEF 文件(Library Exchange Format): 描述了工艺技术和宏单元的 ASCII 文件。

TLF 文件(Timing Library Format)和 CTLF 文件(Compiled Timing Library Format)：CTLF 是 TLF 的编译化版本。 TLF 包含了库的时序信息。

GCF 文件(General Constraints Format)：包含了压强，电压，温度等要求并指向 CTLF 时序库。

Verilog 文件：如果你想进行布局优化和生成 clock tree,就需要此 Verilog 除非你的 Verilog 网表含有了显式连接或没有模块是有 bus pins 的。

7.3.2 设计数据

DEF(Design Exchange Format) **网表**:设计数据的 ASCII 描述。它不仅包含了设计的网表，还包含了设计的物理约束。对于 DEF 网表和 Verilog 网表，你可以只要一个。当然你也可以在 incremental DEF 网表中增加你的附加信息。产生 DEF 文件，你可以是自己手工写，也可以是用 Preview 之类工具产生。

Verilog 网表：可以在读完此文件以后在读入 incremental DEF 文件，来导入附加的设计数据。

GCF(General Constraints Format)**文件**：

SDF 约束文件：

setup 文件：

se.ini: 设置了环境变量，也可以作为一个自动执行的脚本文件。它在 SE 工具启动时从工作目录或逻辑目录中读入此文件。

se.env: 设置了系统运行的环境变量。如果你想设置控制系统运行的变量或设置数个用户的工作环境，则需把此文件放在当前工作目录下。软件在启动的时候将在当前工作目录下搜索此文件。

se.fin: 软件在关掉之前读入此文件。

dlc.init: 如果你没有使用 GCF 文件,为了初始化 the Central Delay Calculator(CDC),需要这个文件。

7.4 时间驱动设计流程：

7.4.1 设计流程(见图 7.2)

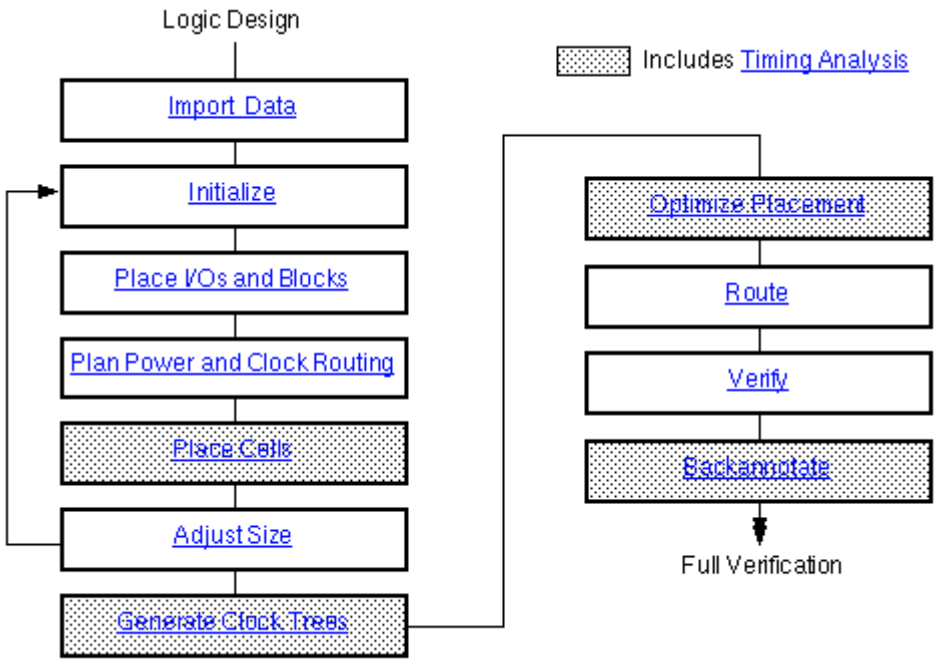


图 7.2 时间驱动设计流程

7.4.2 启动工具

启动 SE 工具：

se | sedsm | seultra [-b] [-j=journal] [-gd=drive] [-e=environmentalFile] [-m=memorySize] [“cmds”] [&] [-version]

-b: 指定运行批处理模式。

-gd=ANSI: 调用非图形界面。提示符为 SE>

-gd-x: 调用图形界面。缺省为图形界面。

执行 ANSI 的批处理：

se | sedsm =b -gd=ansi “EXECUTE script.com; “ &

7.4.3 导入数据文件

1. 导入 LEF 文件： LEF 文件包含了工艺和库的信息。你可以重复读入多个 LEF 文件。如果你有 GDSII 库信息，可以使用 AutoAbgen 来产生你的库。图形窗口界面操作为： File-Import-LEF;命令行界面操作为： INPUT LEF 。后面的命令操作将不再具体指出哪个是图形界面操作，哪个是命令行界面操作，凡是有“-”就是图形界面操作，否则就是命令行界面操作。

2. 导入时序库信息：你即可以导入 GCF 文件也可以导入 CTLF 文件。 GCF 文件里含有指向 CTLF 的语句。如果你导入的是 CTLF 文件，则你还需 dlc.init 文件。命令为： File-Import-Timing Library 或 INPUT CTLF 。

3. 读入 Verilog 库文件(可选)：如果你想进行布局优化及生成 Clock tree,你就需要读入此文件，否则不必。命令为： File-Import-Verilog 或 INPUT VERILOG 。

4. 读入你的设计网表：读入你的设计网表，你可以有两种方式：

i) Verilog 网表： File-Import-Verilog 或 INPUT VERILOG.

ii) DEF 文件:File-Import-DEF 或 INPUT DEF 。

5. 读入 DEF 文件(可选)；如果你有增加的信息，如有关芯片边缘的单元等信息，你就需要读入含此信息的 DEF 文件。这文件一般手写。命令为： File-Import-DEF 或 INPUT DEF 。

6. 读入边界条件(可选)：边界条件是指外部管脚的驱动和负载。对于边界条件你必须使用 GCF 文件。边界条件在所有的时序命令中都会被使用。如果你想产生 clodk tree ，你必须有此边界约束。你也可以使用同样的 GCF 文件读入下面一步中的系统约束。

7. 读入 GCF 或 SDF 格式的系统约束文件:此 SDF 文件可以是 DC 产生的约束文件。

8. 产生 Verilog 文件(可选)：此 Verilog 文件包含了所有的设计网表信息，包括边缘单元等。它可以用来和逻辑设计比较。

7.4.4 初始的 floorplan

为你的物理设计进行初始化。它产生一个只有行或列的 core,放置 I/O 的行或列则放在 core 的四周，同时产生 GCell 和 RGrid 轨迹。它不放置任何单元，只是初步定义了芯片的大小，行的利用率。

命令为： Floorplan-Initialize Floorplan 或 INITIALIZE FLOORPLAN 。

如果必要你可以手工对行，单元等用 EDIT 或 MOVE 等等进行编辑。

7.4.5 放置 I/O 单元和 blocks

一般来说在放置 blocks 之前先放好 I/O 单元，如果你有放置 I/O pads 的约束文件。否则你在放置单元的时候一起放置你的 I/O pins 。放置 I/Opads 的命令为： Place-I/Os 或 PLACE IO.

手工(Edit-Move 或 MOVE)或自动(Place-Blocks 或 QPLACE BLOCKS 放置你的关键 blocks 。

对 blocks 周围的行进行修剪。 Floorplan -Update Core Rows 或 CUT ROW.

7.4.6 规划电源布线：

Route-Plan Power 打开工具栏。如果有必要你就先编辑电源路径(-Delete Pwr Path 或-Restore Pwr Path).然后加入 power rings(-Add Rings 或 CONSTRUCT RING)和 power stripes(-Add Stripes 或 ADD STRIPE)。如果有必要你在编辑电源线(RingWires-Add | Change | Delete).

7.4.7 放置单元(cell)

如果必要，你应该先产生放置 cell,然后使用时序选项去放置你的 cell 。同时为了下

面的时序分析，你应该选中产生 RSPF 文件的选项。这 RSPF 文件是基于全局布线预算的。如果你没有放置 I/O pins,则需在放置 cell 时选中 pin placement 选项。

检查拥挤映射以估计可布通性。如果你的设计没有布通，你必须重新 floorplan 。你用 Floorplan-Compact Floorplan(Vsize)或 VSIZE 调整你的设计。

7.4.8 时序分析

SE 使用静态时序分析器和 Central Delay Calculator 计算精确的时序信息。用 Report-Timing Analysia 或 REPORT TIMING 检查你的设计是否满足时序要求。用 Report-Timing Path 或 REPORT TIMING PATH 检查各个路径。

如果你想用第三方的珍珠延时计算器或时序分析工具来验证你的设计是否满足时序要求，你需要输出以下信息并在相应的软件中读入这些信息：

用 Report-RC 或 REPORT RC 输出寄生信息；

用 Report-Delay 或 DELAY 输出延时信息，写成 SDF 文件。

7.4.9 调整尺寸

到了这一步，你可以估计你的芯片大小并在可布通的条件下调整你的设计尺寸。调整设计尺寸大小的命令为 Floorplan-Compact Floorplan(Vsize)或 VSIZE 。

如果你调整的程度每一维都超过了 10%，则你需要重新你的 floorplan 。

7.4.10 生成 clock trees(可选)

如果你使用命令行界面，你需要一个 CTGen 格式的约束文件。用 Place-Clock Tree Generate 或 CTGen 设置 skew budget 和插入延时限制，然后运行。如果你的 clock tree 不符合你的 skew 限制，你需反复执行这些步骤。接下去再用 File-Import-DEF 或 INPUT DEF ECO 读回 clock tree 。

产生报告查看 clock tree 是不是满足你的要求。用 Report-Clock Skew 或 REPORT CLOCK SKEW 命令检查基于全局布线的 clock skew 分析。用 Report-Timing Analyzsis 或 REPORT TIMING ANALYSIS 命令分析设计的保持建立时间，负载和传输电压。如果有违约(violations),你可以用布局优化器进行纠正。此时你可以用后注释更新你的逻辑设计。

7.4.11 布局优化(可选)

布局优化重新调整了门的尺寸，插入了缓冲器以纠正时序和电气的 violations 。进行此步骤的前提是你有时序库文件和 GCF 系统约束。布局优化你可以使用 PBOpt 工具(命令为 Place-Placement Optimization-PBOpt 或 PBS)，也可以使用 QPlace 优化器(Place-Placement Optimization-QPlace Optimization)进行优化。

如果你有很多的缓冲器，则你还需用 Floorplan-Compact Floorplan(Vsize)或 VSIZE 命令进行调整。这时你也可以用后注释来更新你的逻辑设计。

7.4.12 布线

使用 Route-Connect Ring 或 CONNECT RING 命令对 power nets 进行布线。

使用 Route-Clock Route 或 CLOCK ROUTE 命令对 clock trees 进行布线。

对剩下的 nets 进行布线： Route-Warp Route 或 WROUTE,或者是按传统的方法：先 Route-Globals Route 或 GROUTE 再 Route-Final Route 或 FROUTE 。用 Search 和 Repair 模式可以自动确定一些 violations 。

7.4.13 验证

检查时序，方法同上。

用 Verify-Geometry 或 VERIFY GEOMETRY 命令检查有没有几何或物理的版图 violations 。

用 Verify-Connectivity 或 VERIFY CONNECTIVITY 命令检查有没有连接的 violations 。

用 Verify-Antennas 或 VERIFY ANTENNA 命令检查工艺 antennas 。

用 search 和 repair 模式 FROUTE ,或重新对 wire 和 net 进行布线,以确定 violationss,然后用 Edit-Wire 手工纠正这些 violations 。

7.4.14 后注释

用 Report-RC 或 REPORT RC 命令提取 RSPF 格式的寄生参数。

用 Report-Delay 或 REPORT DELAYS 命令写 SDF 延时文件。

产生新的 Verilog 网表文件： File-Export-Verilog 或 OUTPUT VERILOG 。

7.5 数据路径设计(datapath design)的自动化布局布线

datapath design 的布局布线和时序驱动设计相似。只有微小的区别。就是 datapath 设计需要用 File-Import-Verilog 或 INPUT VERILOG 读入.vip 文件来作为 Verilog 网表。而在完成 floorplan 之后，进行其他操作之前需放置 datapath 结构。放置 datapath 结构的命令为：Floorplan-Datapath Toolbox-Initial Functions;
Floorplan-Datapath Toolbox-Add Regions;
Floorplan-Datapath Toolboxl-Place Regions 。

7.6 ECO 处理

1. 用 File-Import-DEF ECO(INPUT DEF ECO)或 File-Import-Verilog(INPUT VERILOG ECO)读进改变了的网表。
2. 根据设计状态决定你是否应该重新布局。
3. 对有所改变的 nets 或面积重新布线。

注意：GUI 中小数点后的两位算的，如：GUI 中 10.00 ，则命令行中将是 1000
由 DA 得到的 Verilog-HDL 需加入 PAD 的说明。
附加的 DEF 和 IO constraint file (.ioc)和 generate clock tree 的命令文件**.ctgen.cmd 需 自己写.

7.7 设计实例

7.7.1 布局布线前准备

1. 修改 DC 生成的 Verilog 文件，为其加上 core 的 pad 及电源与地的 pad 的说明。当然也可写成 DEF 格式。
2. 编辑 IO 的约束文件(.ioc)文件
3. 如果你是用脚本文件做批处理，则还需写好生成 clock tree 所需的约束文件(.constraint)和命令文件(.ctgen.cmd)

7.7.2 启动 SE 进行布局布线

转到工作目录
%cd work
把启动工具写成一个命令文件 InitWorkDir.csh 放在非工作目录 scripts/：

```
**************************************************************************
#!/bin/csh -f

if(`basename $cwd` == "work") then
  # Remove all previous files.
  \rm -r dbs *.cfg *.dtp *.gds *.info *.ini *.jnl *.rpt *.summary *.wdb
  \rm -r *
  # Prepare the directory.
  mkdir dbs
  cp ~ chwtang/CSMC/csmchdlib/se/se.ini ./se.ini
  # Start se in the background.
  sedsm -m=500 &
else
  echo "Not in work directory."
endif
**************************************************************************
*
```

执行启动工具的命令文件：
% ../scripts/InitWorkDir.csh
启动了 SE 后读入数据文件：
File-Import-LEF ：读入 csmc06.lef ；
File-Import-Timing Library ：读入 csmc06.gcf ；
File-Import-Verilog :读入设计文件 ddfs.v ， ddfstop.v 和库的 verilog 文件 csmc06.v ；
File-Import-SDF ：读入 SDF 文件 ddfs_constraints.sdf ；

File-Import-DEF ；读入电源和地的 pads DEF 文件；
到这里为止， SE 窗口都看不到什么东西。
下面各个参数和变量的设置不详述，具体请看脚本文件。
Initialize Floorplan ：
Floorplan-Initialize Floorplan ：设置各种参数
SE 窗口中出现 rows 。
放置 Iopads:
Place-IOs:读入 IO pads 的约束文件 ddfstop.ioc ；
SE 窗口中显示出 IO pads 的放置。
power planning:
Route-Plan Power-Add Rings:
Route-Plan Power-Add Stripes:
放置标准单元：
Place-Cells:设置好时序变量。生成 clock tree:
Place-Clock Tree Generate(CTGEN):设置 clock tree 的约束文件，产生 clock tree 。
读回 clock tree 的 def 文件:
File-Import-DEF ECO:
place filler core cells:
Place-Filler Cells-Add Cells ：
布线：
Route-Connect Ring:
Route-Wroute:
输出各种文件：
File-Export-GDS II:输出 gdsII 网表；
File-Verilog:输出 Verilog 文件用于版图后仿真；
Report-Delay:输出 SDF 文件，用于版图后仿真。

- **END**

脚本文件见附录。

结束语

本文的设计实例从 RTL 级编码开始，经过 RTL 级仿真，逻辑综合，综合后仿真，自动化布局布线，到(版图候仿真)，构成了一个较简单的完整深亚微米数字集成电路自动化流程。由于 DDFS 实例的规模较小，本实例选择了动态仿真，而非 Formal Verification 。同时也直接从 RTL 级开始编码，跳过了行为级综合，且忽略了各种辅助工具的使用。

由于工艺库和实验室条件的限制，设计流程的某些关节尚存一定的问题：

1. 含双向端口的 I2C 设计无法用 VSS 进行仿真，可能是实验室的 VSS 存在问题，也可能是设计源代码本身不够完善，虽然在 Active-HDL 仿真器中仿真结果是对的。由于时间的限制及实例练习的太少，问题出在哪，尚未发现。这有待后人的解决。
2. 由于库的限制，在 DC 自动生成的 V2.1 版本的 SDF 延时文件中，所有的非 clock 的 posedge,negedge generic 都没有定义。这样的 SDF 文件， VSS 在导入过程中将报 error ，从而使 SDF 文件导入失败。目前的解决办法是手工修改仿真库文件。注意：实验室的 VSS 只支持 V2.1 版本的 SDF 文件。
3. 由于 GCF 工艺库的问题，在用 SE 进行自动布局布线的时候不能进行时序分析。

附录

行为级仿真分析文件 analyze1.sh

```
#*****************************************************************************
#!/bin/csh -f

if(-d work) then
   rm -rf work
   echo "-----"
   echo "Creating work directory"
   echo "-----"
   mkdir work
endif

vhdlan -event\
    vhdl/ddfs.vhd \
    vhdl/froma.vhd \
    vhdl/fromb.vhd \
    vhdl/croma.vhd \
    vhdl/cromb.vhd \
    vhdl/DDFS_TB.vhd
#*****************************************************************************
*
```

仿真命令文件 simfile

```
*****************************************************************************
trace -wif -waves /DDFS_TB/*'signal
run 100000000
*****************************************************************************
```

约束文件 constraint.scr

```
/*****************************************************************************
/
reset_design
create_clock -period 100 find(port,clk)
set_dont_touch_network find(port,clk)
/*****************************************************************************
/
```

较为详细的约束文件 constraints.scr

```
/*****************************************************************************/
MAX_INPUT_LOAD = load_of(cba_core/and2a0/A) * 5

/* read -format db  unmapped/PRGRM_CNT_TOP.db */
reset_design
create_clock -period 10 find(port, Clk)
set_dont_touch_network find(port,Clk)
set_clock_uncertainty 0.3 find(port,Clk)
set_input_delay 2.0 -max -clock Clk  all_inputs() - find(port, Clk)
set_operating_conditions -max slow_120_4.50 -min fast_0_5.50
set_wire_load tc6a120m2
```

```
set_driving_cell -lib_cell fde1a1 -pin Q  all_inputs() - find(port, Clk)
set_max_capacitance MAX_INPUT_LOAD  all_inputs() - find(port, Clk)
set_load MAX_INPUT_lOAD * 3 all_outputs()
/*****************************************************************************
/
```

DC 运行脚本文件 runit.scr

```
/*****************************************************************************
/
SOURCE_DIR = "vhdl/"
SCRIPT_DIR = "scripts/"
MAPPED_DIR = "mapped/"
REPORTS_DIR = "reports/"
TOP = "ddfs"
DESIGN_LIST = {ddfs,froma,fromb,croma,cromb}

foreach (module, DESIGN_LIST) {
        analyze -lib work -format vhdl SOURCE_DIR + module + ".vhd"
        elaborate module
        }
current_design = TOP
link
include SCRIPT_DIR + constraint.scr
uniquify
compile
current_design = TOP
change_names -rules vhdl -hierarchy
write -format db -output MAPPED_DIR + TOP + .db -hierarchy
report_constraint -all_violators > REPORTS_DIR + TOP + .rpt
report_timing -delay max -path full >> REPORTS_DIR + TOP + .rpt
write -format vhdl -hierarchy -output TOP + .vhd
write_sdf TOP + .sdf
write_constraints  -format  sdf-v2.1  -max_nets  0.05    -net_priorities  -
max_path_timing -max_paths 1  -hierarchy -output ddfs_constraints.sdf
write -format verilog -hierarchy -output TOP + .v
quit
/*****************************************************************************
/
```

综合后仿真的分析文件 analyze2.sh

```
#*****************************************************************************
*
#!/bin/csh -f

rm -rf ../csmchdlib/lib
   echo "-----"
   echo "Creating ../csmchdlib/libs directory"
```

```
   echo "-----"
mkdir ../csmchdlib/lib

if(-d work) then
   rm -rf work
   echo "-----"
   echo "Creating work directory"
   echo "-----"
   mkdir work
endif

vhdlan -optimize -event -w csmc06core \
    ../csmchdlib/vital/csmc06core_Vcomponents.vhd \
    ../csmchdlib/vital/csmc06core_Vtables.vhd \
       ../csmchdlib/vital/csmc06core_VITAL.vhd \

vhdlan -event \
        ddfs.vhd \
        vhdl/DDFS_TB.vhd
#*****************************************************************************
*
```

SE 的启动命令文件 InitWorkDir.csh

```
#*****************************************************************************
*
#!/bin/csh -f

if(`basename $cwd` == "work") then
  # Remove all previous files.
  \rm -r dbs *.cfg *.dtp *.gds *.info *.ini *.jnl *.rpt *.summary *.wdb
  \rm -r *
  # Prepare the directory.
  mkdir dbs
  cp ~ chwtang/CSMC/csmchdlib/se/se.ini ./se.ini
  # Start se in the background.
  sedsm -m=500 &
else
  echo "Not in work directory."
endif
#*****************************************************************************
*
```

SE 的脚本文件 TOP.mac

```
#*****************************************************************************
*
# Read LEF files.
INPUT LEF
  FILENAME "../../csmchdlib/lef/csmc06.lef"
  REPORTFILE "./inpLef_csmc06.rpt"
;
```

```
# Read CTLF files.
INPUT CTLF
  INITFILE "../../csmchdlib/gcf/csmc06.gcf"
  REPORTFILE "./inpCtlf.rpt"
;

# Write DEF netlists.
INPUT VERILOG
  FILE "../ddfs.v ../ddfstop.v  ../../csmchdlib/verilog/csmc06.verilog"
  LIB "TOP"
  REFLIB "TOP "
  DESIGN "TOP.TOP:hdl"
#  REPORTFILE "./inpTOP.rpt"
;


# Input constraints sdf.
INPUT SDF FILENAME "../../../synthesis/slave/mapped/ddfs_constraints.sdf"
  REPORTFILE "./inpSDF.rpt"
;

# Input DEF
INPUT DEF
  FILENAME "../../source/ddfssuPads.def"
  REPORTFILE "./inpDEF.rpt"
;


# Save design.
SAVE DESIGN "TOP_netlist" ;
FLOAD DESIGN "TOP_netlist" ;

# Floor plan.
FINITIALIZE FLOORPLAN
  ROWUTILIZATION 0.85
  ROWSPACING 2000
# BLOCKHALO 20000
#  ABUT
#  FLIP
#  ASPECTRATIO 1
  X 300000 Y 300000
  XIO 4000
  YIO 4000
;

#Place IO cells.
 IOPLACE FILENAME "../../source/ddfstop.ioc" STYLE EVEN
;
```

```
# Power planning.
BUILD CHANNEL ;
CONSTRUCT RING
  NET "gnd!"
  NET "vdd!"
  LAYER metal1 CORERINGWIDTH 2000 SPACING CENTER BLOCKRINGWIDTH 0
  LAYER metal2 CORERINGWIDTH 2300 SPACING CENTER BLOCKRINGWIDTH 0
;
ADD STRIPE
  NET "gnd!"
  NET "vdd!"
  DIRECTION Vertical
  LAYER metal2 WIDTH 2300 SPACING 230  COUNT 2 ALL
;
DISPOSE CHANNEL ;

# Save design
SAVE DESIGN "TOP_floorplan" ;
FLOAD DESIGN "TOP_floorplan" ;

#place standard core cells.
SET VAR QPLACE.TIMING.MODE "TRUE";
SET VAR QPLACE.PLACE.INCREMENTAL "FALSE";
QPLACE NOCONFIG ;

SET VAR QPLACE.PLACE.INCREMENTAL "TRUE";
QPLACE NOCONFIG ;

#SAVE;
# Save design
SAVE DESIGN "TOP_placecell" ;
FLOAD DESIGN "TOP_placecell" ;


# Clock Tree Generation

OUTPUT LEF FILENAME TOP4ctgen.lef;

SET V OUTPUT.DEF.SPNET.WILDCARD TRUE;
OUTPUT DEF FILENAME TOP4ctgen.def;
SET V OUTPUT.DEF.SPNET.WILDCARD FALSE;

CTGEN FILENAME "../../source/TOP.ctgen.cmd";

SET VAR ALLOW.EC "TRUE";
INPUT DEF ECO FILENAME "TOPCTGenRun/TOPCtgen.def";

#SAVE design
SAVE DESIGN "TOP_ctgen";
FLOAD DESIGN "TOP_ctgen";
```

```
#place filler core cells.
SROUTE ADDCELL
  MODEL FEEDTHRU
  PREFIX FT
  NO FN SO FS
#  SPIN vdd! NET vdd!
#  SPIN gnd! NET gnd!
  AREA ( -150000 -150000 ) ( 150000 150000 )
;

#save design
SAVE DESIGN "TOP_placefiller" ;
FLOAD DESIGN "TOP_placefiller" ;

#Route core power.
CONNECT RING
  NET "gnd!"
  NET "vdd!"
  STRIPE
  BLOCK ALLPORT
  IOPAD ALLPORT
  IORING
  FOLLOWPIN
;

#Route core power.
CONNECT RING
  NET "gnd2!"
  NET "gnd1!"
  NET "vdd1!"
#  STRIPE
#  BLOCK ALLPORT
#  IOPAD ALLPORT
  IORING
#  FOLLOWPIN
;


#Route signal nets.
SET VAR WROUTE.SEARCHREPAIR TRUE ;
SET VAR WROUTE.TIMING.DRIVEN "FALSE" ;
#SET VAR WROUTE.TIMING.DRIVEN "TRUE" ;
SET VAR WROUTE.FINAL TRUE ;
SET VAR WROUTE.GLOBAL TRUE ;
SET VAR WROUTE.INCREMENTAL.FINAL FALSE ;
WROUTE NOCONFIG ;

SET VAR WROUTE.INCREMENTAL.GLOBAL "TRUE";
SET VAR WROUTE.FINAL TRUE ;
SET VAR WROUTE.GLOBAL FALSE ;
SET VAR WROUTE.INCREMENTAL.FINAL TRUE ;
```

```
WROUTE NOCONFIG INCREMENTAL ;


# Save design.
SAVE DESIGN "TOP_complete" ;
FLOAD DESIGN "TOP_complete" ;

# Export GDSII file
OUTPUT GDSII MAPFILE "../../../../csmchdlib/map/gds2.map"
 STRUCTURENAME TOP
 FILE "../../module/TOP.gds2"
 REPORTFILE TOP_complete.gds2.jnl  ;

# Save verilog file for post-layout simulation
SET VAR OUTPUT.VERILOG.PWR.AND.GND.PORTS "FALSE";
OUTPUT VERILOG FILE "../../postsource/ddfs_post.v" ;

# Save sdf file
REPORT DELAY SDFOUTPUT FILENAME "../../postsource/ddfs.sdf" ;
```