

FPGA 设计流程指南

前言

本部门所承担的 FPGA 设计任务主要是两方面的作用：系统的原型实现和 ASIC 的原型验证。编写本流程的目的是：

- 在于规范整个设计流程，实现开发的合理性、一致性、高效性。
- 形成风格良好和完整的文档。
- 实现在 FPGA 不同厂家之间以及从 FPGA 到 ASIC 的顺利移植。
- 便于新员工快速掌握本部门 FPGA 的设计流程。

由于目前所用到的 FPGA 器件以 Altera 的为主，所以下面的例子也以 Altera 为例，工具组合为 modelsim + LeonardoSpectrum/FPGACompilerII + Quartus，但原则和方法对于其他厂家和工具也是基本适用的。

目 录

1. 基于 HDL 的 FPGA 设计流程概述.....	1
1.1 设计流程图.....	1
1.2 关键步骤的实现.....	2
1.2.1 功能仿真.....	2
1.2.2 逻辑综合.....	2
1.2.3 前仿真.....	3
1.2.4 布局布线.....	3
1.2.5 后仿真（时序仿真）.....	4
2. Verilog HDL 设计.....	4
2.1 编程风格（Coding Style）要求.....	4
2.1.1 文件.....	4
2.1.2 大小写.....	5
2.1.3 标识符.....	5
2.1.4 参数化设计.....	5
2.1.5 空行和空格.....	5
2.1.6 对齐和缩进.....	5
2.1.7 注释.....	5
2.1.8 参考 C 语言的资料.....	5
2.1.9 可视化设计方法.....	6
2.2 可综合设计.....	6
2.3 设计目录.....	6
3. 逻辑仿真.....	6
3.1 测试程序（test bench）.....	7
3.2 使用预编译库.....	7
4. 逻辑综合.....	8
4.1 逻辑综合的一些原则.....	8
4.1.1 关于 LeonardoSpectrum.....	8
4.1.1 大规模设计的综合.....	8
4.1.3 必须重视工具产生的警告信息.....	8
4.2 调用模块的黑盒子（Black box）方法.....	8
参考.....	10
修订纪录.....	10

1. 基于 HDL 的 FPGA 设计流程概述

1.1 设计流程图

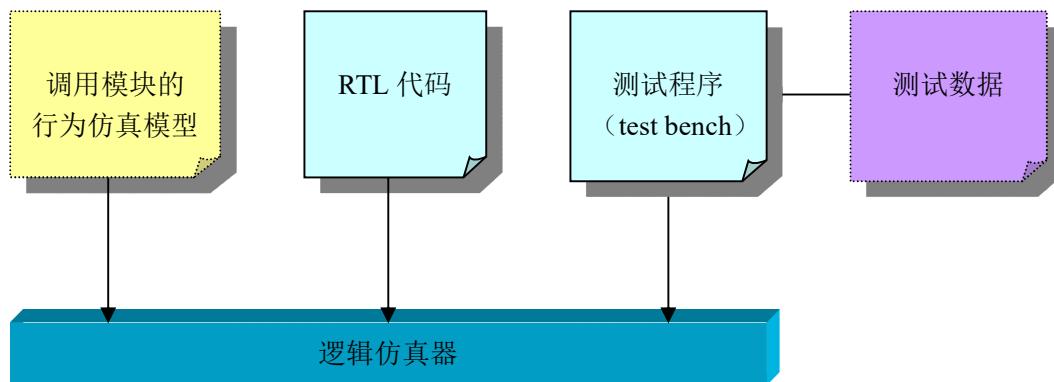


说明：

- 逻辑仿真器主要指 modelsim, Verilog-XL 等。
- 逻辑综合器主要指 LeonardoSpectrum、Synplify、FPGA Express/FPGA Compiler 等。
- FPGA 厂家工具指的是如 Altera 的 Max+PlusII、QuartusII, Xilinx 的 Foundation、Alliance、ISE4.1 等。

1.2 关键步骤的实现

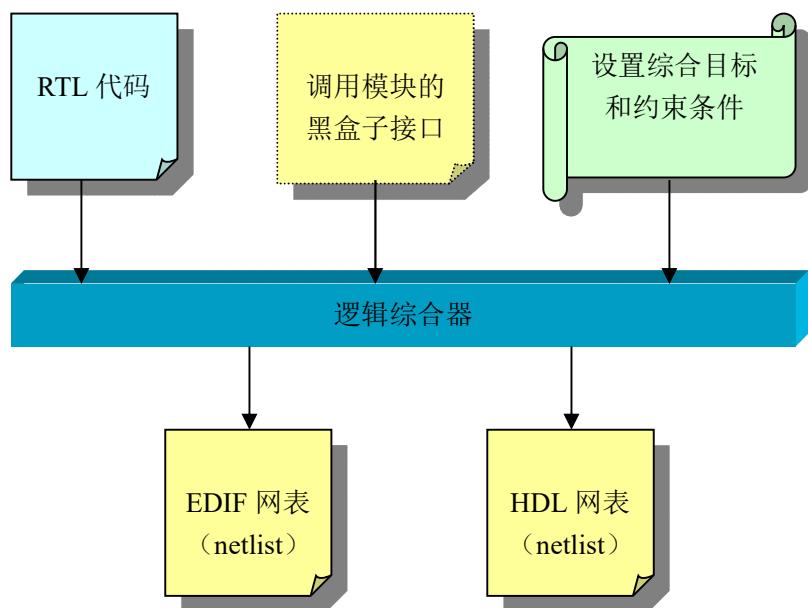
1.2.1 功能仿真



说明：

“调用模块的行为仿真模型”指的是 RTL 代码中引用的由厂家提供的宏模块/IP，如 Altera 提供的 LPM 库中的乘法器、存储器等部件的行为模型。

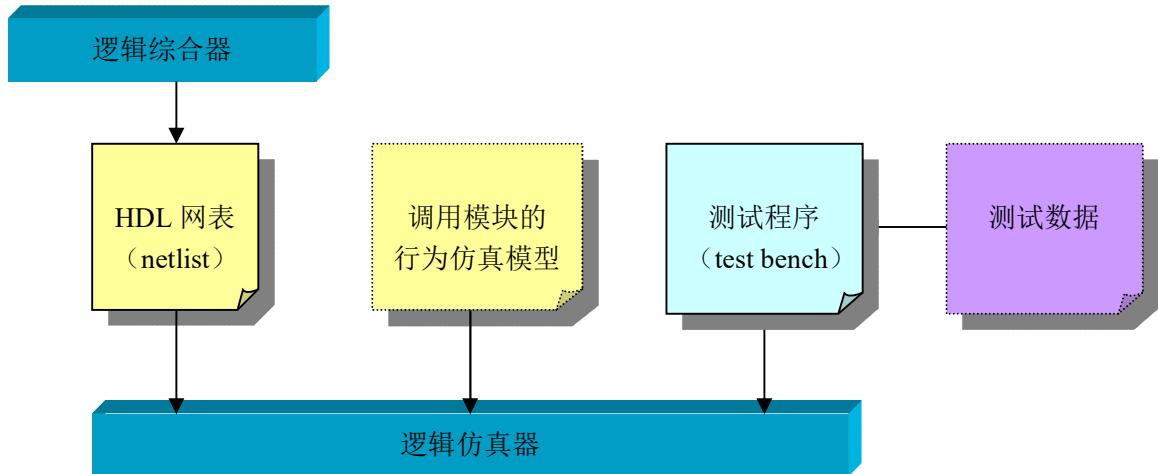
1.2.2 逻辑综合



说明：

“调用模块的黑盒子接口”的导入，是由于 RTL 代码调用了一些外部模块，而这些外部模块不能被综合或无需综合，但逻辑综合器需要其接口的定义来检查逻辑并保留这些模块的接口。

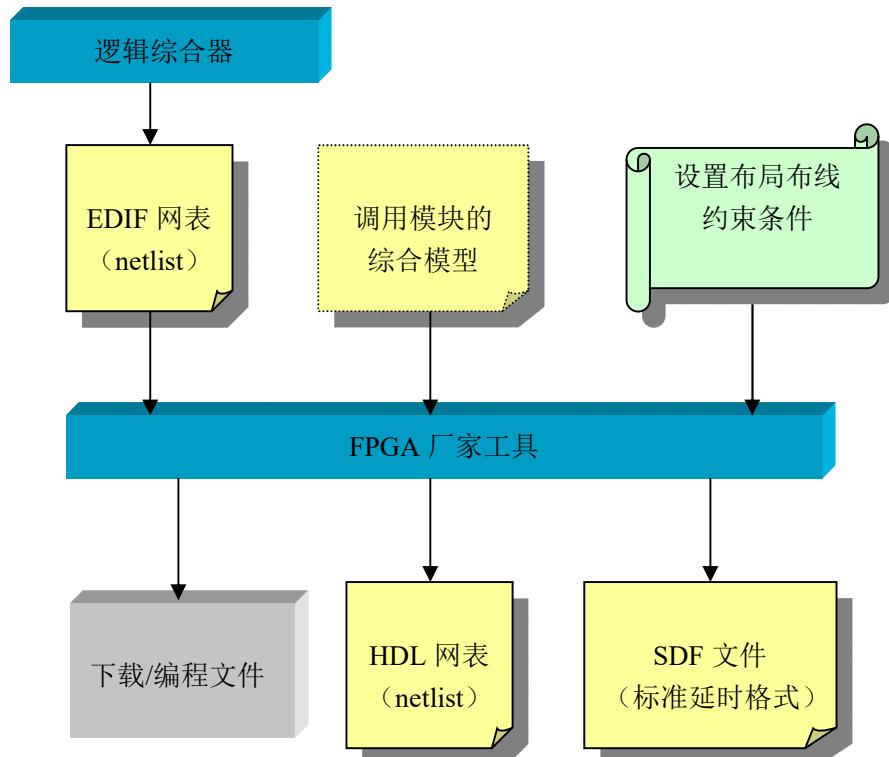
1.2.3 前仿真



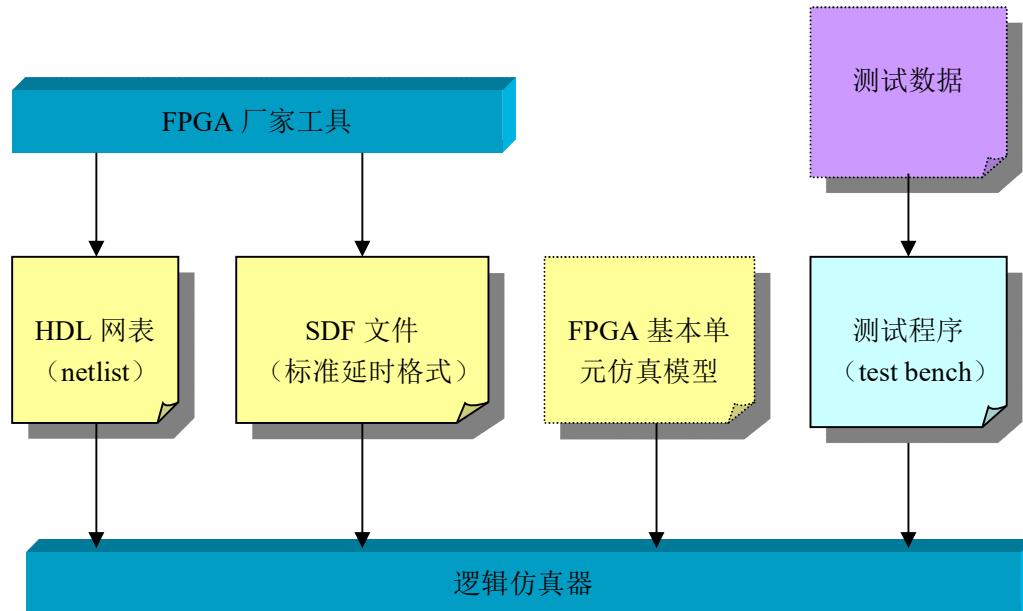
说明：

一般来说，对 FPGA 设计这一步可以跳过不做，但可用于 debug 综合有无问题。

1.2.4 布局布线



1.2.5 后仿真（时序仿真）



2. Verilog HDL 设计

基于将来设计转向 ASIC 的方便，本部门的设计统一采用 **Verilog HDL**，但针对混合设计和混合仿真的趋势，所有开发人员也应能读懂 VHDL。

Verilog HDL 的学习可参考[1][2]。

2.1 编程风格（Coding Style）要求

2.1.1 文件

- (1) 每个模块（module）一般应存在于单独的源文件中，通常源文件名与所包含模块名相同。
 - (2) 每个设计文件开头应包含如下注释内容：
 - 年份及公司名称。
 - 作者。
 - 文件名。
 - 所属项目。
 - 顶层模块。
 - 模块名称及其描述。
 - 修改纪录。
- 请参考标准示例程序[3]。

2.1.2 大小写

- (1) 如无特别需要，模块名和信号名一律采用小写字母。
- (2) 为醒目起见，常数（`define 定义）/参数（parameter 定义）采用大写字母。

2.1.3 标识符

- (1) 标识符采用传统 C 语言的命名方法，即在单词之间以“_”分开，如：max_delay、data_size 等等。
- (2) 采用有意义的、能反映对象特征、作用和性质的单词命名标识符，以增强程序的可读性。
- (3) 为避免标识符过于冗长，对较长单词的应当采用适当的缩写形式，如用 ‘buff’ 代替 ‘buffer’，‘ena’ 代替 ‘enable’，‘addr’ 代替 ‘address’ 等。

2.1.4 参数化设计

为了源代码的可读性和可移植性起见，不要在程序中直接写特定数值，尽可能采用 `define 语句或 parameter 语句定义常数或参数。

2.1.5 空行和空格

- (1) 适当地在代码的不同部分中插入空行，避免因程序拥挤不利阅读。
- (2) 在表达式中插入空格，避免代码拥挤，包括：

赋值符号两边要有空格；
双目运算符两边要有空格；
单目运算符和操作数之间可没有空格，
示例如下：

```
a <= b;  
c <= a + b;  
if (a == b) then ...  
a <= ~a & c;
```

2.1.6 对齐和缩进

- (1) 不要使用连续的空格来进行语句的对齐。
- (2) 采用制表符 Tab 对语句对齐和缩进，**Tab 键采用 4 个字符宽度**，可在编辑器中设置。
- (3) 各种嵌套语句尤其是 if...else 语句，必须严格的逐层缩进对齐。

2.1.7 注释

必须加入详细、清晰的注释行以增强代码的可读性和可移植性，注释内容占代码篇幅不应少于 30%。

2.1.8 参考 C 语言的资料

要形成良好的编程风格，有许多细节需要注意，可以参考资料[4]，虽然它是针对 C 语言的讨论，但由于 Verilog HDL 和 C 语言的形式非常近似，所以里面提到的很多原则都是可以借鉴的。

2.1.9 可视化设计方法

为提高设计效率和适应协同设计的方式，可采用可视化的设计方法，Mentor Graphics 的 Renoir 软件提供了非常好的设计模式。

2.2 可综合设计

用 HDL 实现电路，设计人员对可综合风格的 RTL 描述的掌握不仅会影响到仿真和综合的一致性，也是逻辑综合后电路可靠性和质量好坏最主要的因素，对此应当予以充分的重视。

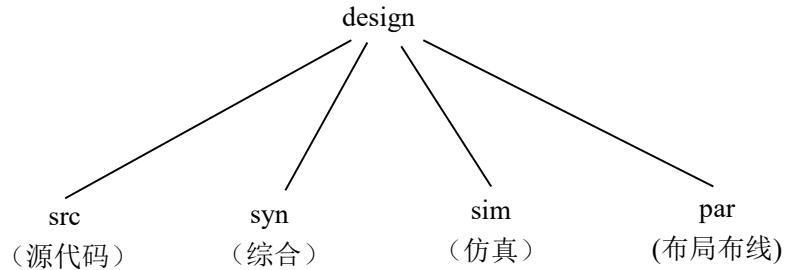
学习可综合的 HDL 请参考 [5][6][7]。

学习设计的模块划分请参考[8]。

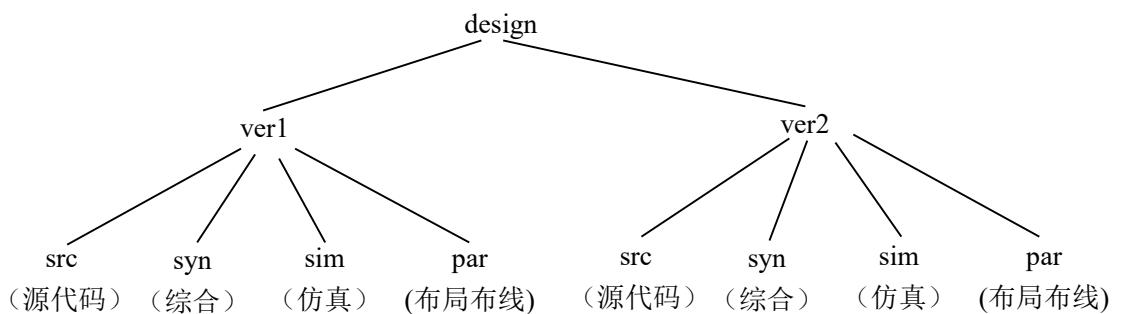
2.3 设计目录

采用合理、条理清晰的设计目录结构有助于提高设计的效率、可维护性。建议采用类似下面的目录结构：

(1)



(2)



3. 逻辑仿真

考虑到性能和易用性，首选的逻辑仿真器是 Mentor Graphics 的 modelsim。

3.1 测试程序 (test bench)

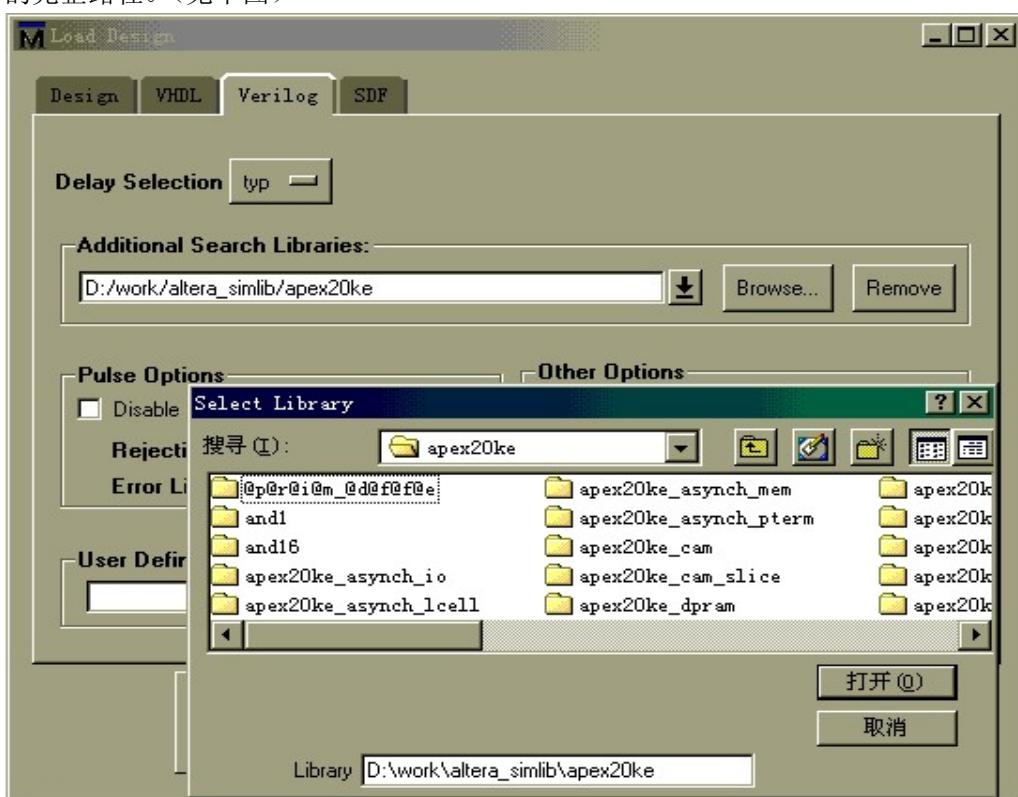
测试程序对于设计功能和时序的验证有着举足轻重的影响, 测试激励的完备性和真实性是关键所在, 有以下原则须遵循:

- (1) 测试激励输入和响应输出采集的时序应当兼顾功能仿真(无延时)和时序仿真(有延时)的情况。
- (2) 对于周期较多的测试, 为提高效率, 尽可能采用程序语句来判断响应与标准结果是否一致, 给出成功或出错标志, 而不是通过观察波形来判断。
- (3) 采用基于文件的测试是很好的办法, 即由 matlab 或 spw 等系统工具产生测试数据, 测试程序将其读入产生激励, 再把响应结果写入到文件, 再交给上述工具进行处理或分析。
- (4) 仿真器支持几乎所有的 Verilog HDL 语法, 而不仅仅是常用的 RTL 的描述, 应当利用这一点使测试程序尽可能简洁、清楚, 篇幅长的要尽量采用 task 来描述。

3.2 使用预编译库

在进行功能仿真和后仿真时都需要某些模块的行为仿真模型和门级仿真模型, 如 Altera Quartus 里的 220model.v (LPM 模块行为仿真模型) 和 apex20ke_atoms.v (20KE 系列门级仿真模型), 为避免在不同的设计目录中多次编译这些模型, 应当采用一次编译, 多次使用的方法。具体做法如下 (以 20KE 门级库为例):

- 1: 在某个工作目录下新建一库名 apex20ke, 将 apex20ke_atoms.v 编译到其中。
- 2: 在图形界面中的 Load Design 对话框中装入仿真设计时, 在 Verilog 标签下指定预编译库的完整路径。(见下图)



4. 逻辑综合

目前可用的 FPGA 综合工具有 Mentor Graphics 的 LeonardoSpectrum，Synplicity 的 Synplify 和 Synopsys 的 FPGA CompilerII/FPGA Express，LeonardoSpectrum 由于性能和速度最好，成为我们首选的综合器，FPGA CompilerII/FPGA Express 由于可以和 Design Compiler 代码兼容也可用。见参考[9]

4.1 逻辑综合的一些原则

HDL 代码综合后电路质量的好坏主要取决于三个方面：RTL 实现是否合理、对厂家器件特点的理解和对综合器掌握的程度。参考[10]中有比较全面的讨论。

4.1.1 关于 LeonardoSpectrum

LeonardoSpectrum 对综合的控制能力比较强，但使用也略为复杂，故需要在使用前尽量熟悉其功能，才能取得较好的综合结果。

当出现综合结果不能满足约束条件时，不要急于修改设计源文件，应当通过综合器提供的时序和面积分析命令找出关键所在，然后更改综合控制或修改代码。

在 LeonardoSpectrum 2000.1b 以前的版本输出的 .v 网表都不能用于仿真。

4.1.1 大规模设计的综合

- 分块综合

当设计规模很大时，综合也会耗费很多时间。如果设计只更改某个模块时，可以分块综合。如有设计 top.v 包含 a.v 和 b.v 两个模块，当只修改 a.v 的话，可以先单独综合 b.v，输出其网表 b.edf，编写一个 b 模块的黑盒子接口 b_syn.v，每次修改 a.v 后只综合 top.v、a.v、b_syn.v，将综合后的网表和 b.edf 送去布线，可以节约综合 b 模块的时间。

- 采用脚本命令

当设计规模比较大时，综合控制也许会比较复杂，可以考虑采用脚本控制文件的方式进行综合控制，modelsim、LeonardoSpectrum 和 Quartus 都支持 TCL (Tool Command Language) 语言，采用脚本控制可以提供比图形界面更灵活和更方便的控制手段。

4.1.3 必须重视工具产生的警告信息

综合工具对设计进行处理可能会产生各种警告信息，有些是可以忽略的，但设计者应该尽量去除，不去除必须确认每条警告的含义，避免因此使设计的实现产生隐患。

这个原则对仿真和布局布线同样适用。

4.2 调用模块的黑盒子（Black box）方法

使用黑盒子方法的原因主要有两点：

一是 HDL 代码中调用了一些 FPGA 厂家提供的模块（如 Altera 的 LPM 模块）或第三方提供的 IP，这些模块不需要综合，而且有些综合器也不能综合（如 FPGA CompilerII/FPGA Express 可以综合包含 LPM 的代码而 LeonardoSpectrum 不能）。因此须提供一个黑盒子接口给综合器，所调用的模块到布局布线时才进行连接。

二是方便代码的移植，由于厂家提供的模块或第三方提供的 IP 通常都是与工艺有关的，直接在代码中调用的话将不利于修改，影响代码移植。

下面以调用 Altera 的 LPM 库中的乘法器为例来说明。调用这样一个模块需要这样一个文件：mult8x8.v（可由 Quartus 的 MegaWizer Plug-in Manager 产生），代码如下：

```
// mult8x8.v

module mult8x8 (dataa, datab, result);

input [7:0] dataa;
input [7:0] datab;
output [15:0] result;

// exemplar translate_off

// synopsys translate_off

lpm_mult lpm_mult_component(
    .dataa  (dataa),
    .datab  (datab),
    .aclr   (1'b0),
    .clock   (1'b0),
    .clken   (1'b0),
    .sum     (1'b0),
    .result  (result)
);

defparam
    lpm_mult_component.lpm_widtha      = 8,
    lpm_mult_component.lpm_widthb      = 8,
```

```

lpm_mult_component.lpm_widths      = 16,
lpm_mult_component.lpm_widthp     = 16,
lpm_mult_component.lpm_representation = "SIGNED",
// exemplar translate_on
// synopsys translate_on
endmodule

```

注意上述的代码有两对编译指示:

```

// exemplar translate_off 和 // exemplar translate_on (LeonardoSpectrum 支持)
// synopsys translate_off 和 // synopsys translate_on ( LeonardoSpectrum 和 FPGA
CompilerII 都支持)

```

对于相应的综合器，在这些编译指示中间的语句将会被忽略，那我们可以看到在综合过程中模块 mult8x8 实际变成了一个只有 I/O 定义的空盒子（即 black box），所以该部分的代码没有连接，在 Quartus 布局布线的时候，lpm 模块的代码才连接到整个设计，在仿真的时候，编译指示不影响模块的完整性。

参考

- [1]: 台湾清华 Verilog HDL 教程
- [2]: Verilog HDL 硬件描述语言
- [3]: 文件头注释块示例
- [4]: C 语言的风格
- [5]: Verilog HDL Reference manual
- [6]: Actel HDL coding style guide
- [7]: LeonardoSpectrum HDL Synthesis
- [8]: ASIC Design Partitioning
- [9]: 三种 FPGA 综合工具的比较
- [10]: FPGA Synthesis Training Course

修订纪录

V2.0 (何辉, 2001-8-1)

修改了 4.2 节（黑盒子方法）的描述

V1.0 (何辉, 2001-3)

第一个版本



目 录

1 前言	8
2 综合工具与代码风格	8
2.1 理解综合两个过程	9
2.2 不同综合工具的性能	10
2.3 综合性能对Coding Style影响	10
3 FPGA器件结构（VirtexII）	10
3.1 器件结构对Coding Style的影响	11
3.1.1 FPGA结构	11
3.1.2 ASIC结构	11
3.1.3 Coding Style的对比	11
3.2 VirtexII功能概述	12
3.3 结构概述	12
3.3.1 CLB	13
3.3.2 Slice	14
3.3.3 LUT	15
3.3.4 Shift Register LUT（SRL）	16
3.3.5 MUXFX	16
3.3.6 Carry Logic 和 Arithmetic Logic Gates	17
3.3.7 SOP	20
3.3.8 FFX/FFY	21
3.4 Memory	21
3.4.1 Distributed RAM	21
3.4.2 Block RAM	23
3.5 乘法器资源	25
3.6 IOB	27
3.6.1 IOB结构	27
3.6.2 Select I/O	28
3.6.3 DCI	29
3.7 Clock Resource	29
3.7.1 Global Clock	29
3.7.2 CLK MUX	30
3.7.3 DCM	32
3.8 补充说明	33
3.8.1 LUT如何配置成组合逻辑电路：揭开“门数增加，逻辑级数未变，但资源占用减少，速度更快”之谜	34
3.8.2 解剖Block SelectRAM内部结构	35
4 设计技巧	37
4.1 合理选择加法电路	38
4.1.1 串行进位与超前进位	38
4.1.2 使用圆括号处理多个加法器	39
4.2 IF语句和Case语句：速度与面积的关系	40
4.3 减少关键路径的逻辑级数	41



4.3.1 通过等效电路，赋予关键路径最高优先级	41
4.3.2 调整if语句中条件的先后次序	42
4.4 合并if语句，提高设计速度	43
4.5 资源共享	44
4.5.1 if语句	44
4.5.2 loop语句	45
4.5.3 子表达式共享	46
4.5.4 综合工具与资源共享	46
4.6 流水线（Pipelining）	47
4.7 组合逻辑和时序逻辑分离	49
4.8 利用电路的等价性，巧妙地“分配”延时	52
4.9 复制电路，减少扇出（fanout），提高设计速度	52
4.10 多路选择器与三态电路	53
4.10.1 virtex以前的系列	53
4.10.2 virtex系列	54
4.11 利用LUT四输入特点，指导电路设计	54
4.12 高效利用IOB	55
4.13 Distributed RAM的使用	56
4.14 Block SelectRAM的使用	57
4.15 SRL的使用	57
4.16 LFSR加1计数器	57
5 如何使用后端工具	58
5.1 布局布线	58
5.1.1 设计前期（设计方案阶段），对关键电路的处理	58
5.1.2 布局布线策略，兼谈如何做第一次布局布线	58
5.1.3 正确看待map之后的资源占用报告	59
5.2 FPGA Editor的作用	59
5.3 FloorPlanner的作用	59
5.4 TimingAnalyzer的作用	60
6 综合运用	60
6.1 可能成为关键路径的电路	60
6.2 如何提高芯片速度	60
6.2.1 引入放松约束：TIG（False path）和Multi-Cycle-Path	60
6.2.2 对线延时比较大的net，设置Maxdelay和Maxskew	61
6.2.3 采用BUFGS	61
6.2.4 基本设计技巧	61
6.2.5 专有资源的利用	61
6.2.6 关键路径在同一个Module	61
6.2.7 关键路径单独综合，不与其它模块放在一起综合	61
6.2.8 针对关键路径，进行位置约束	61
6.2.9 迂回策略：降低非关键路径上的面积，为关键路径腾挪空间。	61
6.3 如何降低芯片面积	61
6.3.1 Distributed RAM代替BlockRAM	61
6.3.2 Distributed RAM代替通道计数器	61
6.3.3 专有资源的利用	62



6.3.4 基本设计技巧	62
7 感谢	62

表目录

表1 VirtexII 的分布式RAM 配置表	22
表2 VirtexII 的BlockRAM 分布表:	24
表3 带奇偶校验位的Block RAM配置表	25
表4 VirtexII 乘法器速度表（厂家数据）	27
表5 VirtexII 的DCM分布表	33

图目录

图1 使用二进制描述的Mux	9
图2 使用内部三态线描述的Mux	9
图3 VirtexII 结构示意图	13
图4 VirtexII 的CLB结构示意图	14
图5 SLICE结构示意图	14
图6 VirtexII 的Slice 结构图（上半部分）	15
图7 SRL的移位链	16
图8 VirtexII的MUXFX连接图	17
图9 进位链结构示意图	18
图10 使用进位链实现加法器	18
图11 使用进位链级联实现高速宽函数运算	19
图12 VirtexII 的两个独立进位链	20
图13 VirtexII 的SOP 链	21
图14 FFX/FFY结构示意图	21
图15 单端口32x1 RAM	22
图16 双端口16x1 RAM	23
图17 VirtexII 的Block RAM 分布规律	24
图18 Write first 模式	25
图19 Read first 模式	25
图20 No Change 模式	25
图21 乘法器与Block RAM	26
图22 XC2V40的乘法器	26
图23 乘法器块	26
图24 VirtexII的IOB	27
图25 VirtexII 的IOB中的DDR	28
图26 VirtexII 的IOB 实际结构	28
图27 VirtexII 的Clock Pads	29
图28 VirtexII 的时钟（顶部）	30
图29 VirtexII 的时钟资源分布原理	30



图30 VirtexII的BUFGMUX	31
图31 VirtexII的BUFG	31
图32 VirtexII 的BUFGCE	31
图33 VirtexII 的BUFGCE	31
图34 VirtexII 250 的DCM 位置	32
图35 VirtexII 的DCM	33
图36 门数增加，逻辑级数未变，但资源占用减少，速度更快	35
图37 完整的单端口Block Select RAM	36
图38 Read first mode	36
图39 Write first mode	37
图40 No-read-on-write mode	37
图41 串行进位	38
图42 超前进位	39
图43 串行加法电路	39
图44 并行加法电路	39
图45 if-else完成多路选择	40
图46 case语句完成电路选择	41
图47 critical信号经过2级逻辑	42
图48 critical信号只经过一级逻辑	42
图49 资源共享前，2个加法器	44
图50 资源共享后，1个加法器	45
图51 资源共享前4个加法器	45
图52 资源共享后一个加法器	46
图53 采用流水线之前电路结构	48
图54 采用流水线之后的电路结构	49
图55 Mealy状态机的基本结构	49
图56 组合逻辑（加法器）在后	52
图57 组合逻辑（加法器）在前	52
图58 扇出较大	53
图59 扇出较小	53
图60 多路选择	54
图61 采用三态电路实现电路选择	54
图62 VirtexE IOB结构示意图	55
图63 输入输出寄存器移入IOB中	55
图64 采用Distributed RAM实现多路加1计数器	57
图65 15位基本型LFSR计数器在VIRTEX器件中的实现	58



FPGA设计高级技巧（xilinx篇）

关键词：FPGA器件结构、速度与面积、关键路径、压缩线延时、降低LUT级数、腾挪空间

摘要：

本文从FPGA器件结构角度出发，以速度和面积为主题，描述在FPGA设计过程中应当注意的问题和可以采用的设计技巧。

缩略语清单：

ASIC: Application Specific Integrated Circuit

CLB: Configurable Logic Block

DCI: Digitally Controlled Impedance

DCM: Digital Clock Manager

DDR: Double Data Rate

DLL: Delay-Locked Loop

FPGA: Field Programmable Gate Array

GRM: General Routing Matrix

IOB: Input/Output Block

LFSR: Linear Feedback Shift Register

LUT: Look Up Table

SOP: Sum of Product

SRL: Shift Register LUT

UCF: Custom Constraints File

参考资料清单：

参考资料清单				
名称	作者	编号	发布日期	查阅地点或渠道
VHDL数字电路设计指导	周志坚/钱晶			中研基础
Virtex系列器件结构简介	苏文彪			中研基础
VerilogHDL编码入门指导	牛风举			中研基础
基于FPGA器件的编码规范	苏文彪			中研基础
UCF应用指导	喻志清/周志坚			中研基础
Xilinx后端工具使用指导	喻志清			中研基础
LFSR计数器原理及应用				中研基础



同步电路设计技术及规则	周志坚			中研基础
VirtexII时钟资源（初稿）	陈亮			中研基础
ds031(virtexII).pdf	xilinx			周志坚
lfsr.pdf	xilinx			周志坚
sp_block_mem.pdf	xilinx			周志坚
VirtexII_DesignConsideration.pdf	xilinx			周志坚
gensim.pdf	xilinx			周志坚

1 前言

随着HDL（Hardware Description Language，硬件描述语言）语言、综合工具及其它相关工具的推广，使广大设计工程师从以往烦琐的画原理图、连线等工作解脱开来，能够将工作重心转移到功能实现上，极大地提高了工作效率。

任何事务都是一分为二的，有利就有弊。我们发现，现在越来越多的工程师不关心自己的电路实现形式，以为“我只要将功能描述正确，其它事情交给工具就行了”。在这种思想影响下，工程师在用HDL语言描述电路时，脑袋里没有任何电路概念，或者非常模糊；也不清楚自己写的代码综合出来之后是什么样子，映射到芯片中又会是什么样子，有没有充分利用到FPGA的一些特殊资源。遇到问题，立刻想到的是换速度更快、容量更大的FPGA器件，导致物料成本上升；更为要命的是，由于不了解器件结构，更不了解与器件结构紧密相关的设计技巧，过分依赖综合等工具，工具不行，自己也就束手无策，导致问题迟迟不能解决，从而严重影响开发周期，导致开发成本急剧上升。

目前，我们的设计规模越来越庞大，动辄上百万门、几百万门的电路屡见不鲜。同时，我们所采用的器件工艺越来越先进，已经步入深亚微米时代。而在对待深亚微米的器件上，我们的设计方法将不可避免地发生变化，要更多地关注以前很少关注的线延时（我相信，ASIC设计以后也会如此）。此时，如果我们不在设计方法、设计技巧上有所提高，是无法面对这些庞大的基于深亚微米技术的电路设计。而且，现在的竞争越来越激励，从节约公司成本角度出发，也要求我们尽可能在比较小的器件里完成比较多的功能。

本文从澄清一些错误认识开始，从FPGA器件结构出发，以速度（路径延时大小）和面积（资源占用率）为主题，描述在FPGA设计过程中应当注意的问题和可以采用的设计技巧。

本文对读者的技能基本要求是：熟悉数字电路基本知识（如加法器、计数器、RAM等），熟悉基本的同步电路设计方法，熟悉HDL语言，对FPGA的结构有所了解，对FPGA设计流程比较了解。

2 综合工具与代码风格

硬件描述语言和综合工具的产生，极大地提高了工程师的工作效率。然而，随着它们的普及与推广，一种不好的现象也在逐步蔓延：在设计过程中，只关注功能是否实现，而不考虑或



很少考虑电路到底是如何实现的；过分依赖综合等工具来提高设计性能（如速度、面积等），而不是从设计本身来考虑自己的电路是否最佳。

如果将设计看成是一个化学变化，那么工具只是起到催化剂的作用，我们所掌握的背景知识、电路设计方法及有关技巧，才是参加化学反应的分子，是起决定作用的因素。

因此，设计遇到困难时，不能完全指望工具，更不能怪罪工具。只有我们才是决定设计成败的关键。

2.1 理解综合两个过程

不管何种综合工具，一般包括如下两个过程：

Synthesis 和 Optimization 。

前者是把行为级的描述通过一定的算法转化成门级的描述，该过程与设计的工艺库无关、与用户约束无关。

后者则是把已转化的门级描述在用户的约束下，通过算法映射到相应的工艺库中的器件上。对ASIC，是映射到厂商的Gate 库中，对FPGA，是映射到FPGA器件的单元结构中。

从上两个步骤可知，当设计代码的风格不一样时，则在综合第一步就已大部分决定了设计的性能（对ASIC来说，因为是转成Gate，器件库一般也是Gate，相对影响较少）。因此，我们不难理解Code Style 对FPGA设计的重要性。

举个例子，如下的16选1MUX（以4000系列为例），前者用二进制译码（如case语句）的描述，后者使用BUFT的描述（采用三态Buffer来实现多路选择器）：

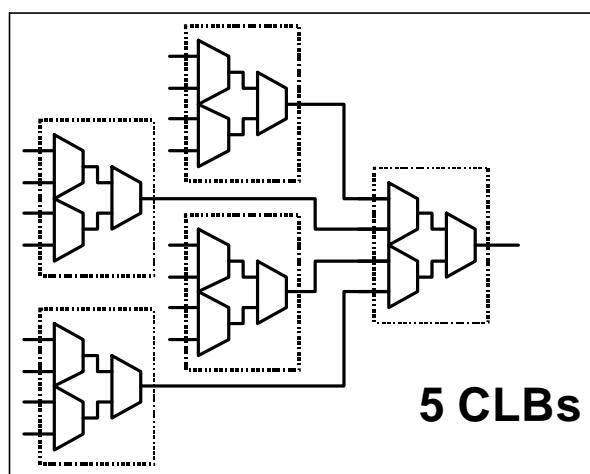


图1 使用二进制描述的Mux

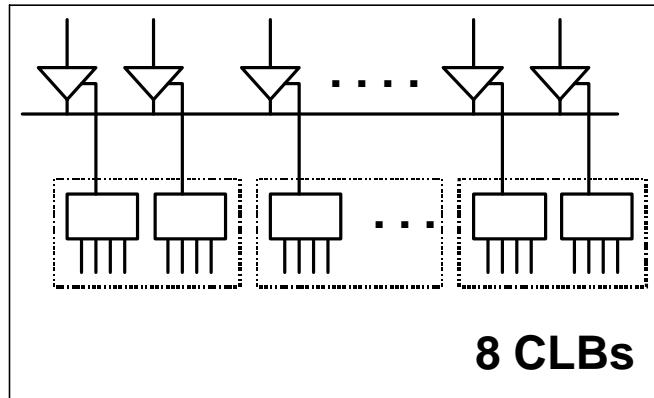


图2 使用内部三态线描述的Mux

2.2 不同综合工具的性能

不同综合工具的针对目标不一致和各综合工具的不同性能，导致了综合出来的结果也不同。目前，综合工具主要有 DC（Design Compiler）、FC2（FPGA Compiler II）、Synplify、Leonardo、Galileo 等综合工具。

其中DC主要是用于ASIC的综合工具，FC2是DC在FPGA综合方面的增强版。其中，FC2的开发队伍已解散。Leonardo 是做FPGA 综合工具的先驱。Synplify（Synplicity公司出品）是目前成长最快的综合工具。但无论哪家综合工具，对FPGA的综合，都必须紧密结合各FPGA厂家的FPGA结构，否则无法更好利用FPGA提供的优点。从目前来看，对做FPGA，优选Synplify 或 Leonardo 综合工具。DA的FPGA综合性能差但适宜ASIC综合。

到目前为止，第三方的综合工具都未能很好利用FPGA器件，如Virtex 系列的进位链，综合工具不管代码风格如何变就是利用不上（目前是这样，以后很难说，因为综合工具一直在升级），因此无法得到更好的性能。但若用Xilinx 的自己的综合工具XST，则可综合出来。

从我们目前掌握的情况来看，Synplicity公司的综合工具比较优秀一点，其最新推出的Amplify工具值得跟踪，不过价格太贵。

2.3 综合性能对Coding Style 影响

由于综合工具无法利用器件提供的优越性能，导致设计性能变差，这要求在FPGA设计上，若想得到更好的性能，最好对一些设计采用一些Core，但这种基于FPGA器件（特有工艺）的代码设计，将降低设计代码的重用性能、FPGA设计与ASIC设计的兼容性。

考虑到综合工具不是万能的，因此，我们在编写代码时，要采取恰当的风格，以提高电路性能。对FPGA设计（Xilinx为例）而言，其所要求的代码风格：

1. 资源共享的应用限制在同一个module里。这样，综合工具才能最大限度地发挥其资源共享综合作用。
2. 尽可能将Critical path上所有相关逻辑放在同一个module里。这样，综合工具能够发挥其最佳综合效果。
3. Critical path所在的module与其它module分别综合，对critical path采用速度优先的综合策略，对其它module采用面积优先的综合策略。



4. 尽可能Register所有的Output。做到这一点，对加约束比较方便；同时一条路径上的组合逻辑不可能分散在各个module里，这对综合非常有利。可以比较方便地达到面积、速度双赢的目的。

5. 一个module的size不能太大。具体大小，由各综合工具而定。

6. 一个module尽量只有一个时钟，或者整个设计只有一个时钟。

更多的代码风格，可参见中研基础部的“verilog代码书写规范”。

3 FPGA器件结构（VirtexII）

许多工程师在做设计时，并不了解器件的结构，不关心自己的电路是怎么实现的，过分的依赖综合工具和厂家的布线工具来完成设计，并且认为：我只要将功能描述正确就够了，至于它是如何实现的，自然由综合工具等相关工具来决定。这实际是“重功能、轻实现”思维的表现，其结果是在我们的设计中人为地制造了一大堆“垃圾”（本人的体会是，第一个设计有20~30%的优化可能。当然，这个结果是基于当时大家水平都不高。如果大家在一个高手入云的环境下进行熏陶，我想信会表现得好得多），人为地提高了芯片成本。

我们在前面提到，综合工具并不一定保证能够充分利用芯片结构特点以达到最优目的，而且工具本身也不一定非常智能，它永远不会变成最佳（想一想，如果能够最佳，那么它还有优化的余地吗？），因为设计本身是复杂多样的且一直在变化，问题总会越来越多。

因此，在这种情况下，我们必须了解我们的器件结构，了解我们的设计是如何实现的，它是否充分利用到了FPGA里面的特有资源（如进位链、shift register、IOB中的register等）。如果没有，则应当想办法充分利用（如修改代码，以适合FPGA结构特性；或者，采用coregen生成的module等）。这在许多场合，是一个非常行之有效的手段。

“只有工具与大脑完美结合，才能获得最佳效果”。

本章节主要以VirtexII为例进行说明，主要目的是想让读者知道了解FPGA器件结构对做好FPGA设计有多么重要。读者要想获得Virtex其它系列更多的知识，可参见“Virtex系列器件结构简介”一书，或者看xilinx提供的相关资料。

3.1 器件结构对Coding Style的影响

3.1.1 FPGA结构

Altera的FPGA、Xilinx的FPGA或其他公司的FPGA，一般的结构都是由一些CLB（或类似称为LE）的宏单元组成，其内的component一般是查找表（LUT）、时序单元（如寄存器），外加一些如进位链等先进的结构。如Altera的FPGA和Xilinx的FPGA都采用4输入的查找表（LUT）。在LE或CLB中，Component的延时是固定的、可预测的。我们知道，设计的延时包括器件的延时和线延时，对FPGA，器件的延时反映在所用的components的级数上，如利用了多少级的查找表、或使用多少级进位链等。线延时则反映在CLB与CLB的互连上，当一个逻辑采用的CLB级数越多，就需要越长的互连线，结果是线延时越长。



在FPGA中，一般都提供了一些先进结构（如进位链等），目的是减少对CLB数目的使用，以期得到最高的性能，如Virtex 系列中，时序单元本身具有同步复位的功能，但不占用LUT的资源。再如进位链，可以用来实现快速进位的加法器或宽输入的函数，且把LUT的使用减少到最少但速度更快。因此，在编码上，就应该考虑如何更好地利用FPGA器件中的这些特点。

3.1.2 ASIC结构

基于门阵列（或标准单元）的ASIC，其线延时不象FPGA那样，一般可作到较小，因此，设计的速度瓶颈往往在于Gate 的级数上（深亚微米级时，线延时是主要因素）。虽然一些厂家可能也提供类似FPGA的宏单元结构，但更多的是与门、非门等之类的一些gate 。因此，减少逻辑个数是提高设计速度的主要手段。

3.1.3 Coding Style的对比

由于器件结构的不同，导致在实现一个更高性能的设计时，针对ASIC和FPGA，需要的Coding Style 也不同。

FPGA器件的设计性能很大程度依赖于Coding ，而ASIC设计性能与Coding 相关较少。对 Gate Array（门阵列）或standard cell（标准单元）设计，我们可很容易达到较高速度，也不必要求很高的Coding 技术（同步规则或一些通用设计要求除外）。如采用Gate Array， 66M 就很容易实现。但在FPGA设计，我们很少看见几十层逻辑级的设计（7，8 级逻辑级一般只能实现到50M左右）。因此，对FPGA设计，要达到高速和好的性能则需要好的代码风格和好的设计策略。

我们往往有个误区，那就是提到Code Style 时往往忽略了对器件结构的了解，即沿用了ASIC设计思路----减少逻辑的门级就是提高速度。对FPGA而言，要想获得较高速度和节省面积，是以减少LUT的个数为主要手段；减少逻辑级数，不一定能提高速度和降低面积，有时反而会降低速度和增加面积。

注意：对FPGA而言，门数和面积（资源占用率）不一定成正比，有时候是反比。

至于为什么，请读者耐心看完随后章节内容，自然会明白。

3.2 VirtexII功能概述

VirtexII 系列器件提供的是1.5v 的FPGA系列，可提供如下功能：

- 1、 提供更高密度的FPGA资源，从40K（xc2v40） 到10M（xc2v10000）：最高支持420M内部时钟频率和840Mb/s 的I/O。
- 2、 支持19 种 single-ended 标准 的IO 和 9种差分IO 标准。与其他不同的是，VirtexII 具有XCITE 功能（数字控制电阻匹配）。
- 3、 IOB中集成了DDR （Double Data Rate）寄存器，用户管脚达1108 个，支持可编程的 sink current（2ma 到24ma）。
- 4、 在RAM上，每个块RAM有18Kbit 。对外RAM接口性能提高，支持：
400M b/s DDR-SDRAM 接口
400Mb/s FC RAM 接口



333Mb/sQDR-SRAM 接口

600Mb/s Sigma RAM 接口

5、集成有18bit x 18 bit 的乘法器专有模块。

6、具有水平级连结构。

7、在时钟方面，增强了以往DLL功能，有12个DCM。16个全局时钟，并有时钟选择器。

8、SRAM 结构

9、.15um 技术、8层金属。

3.3 结构概述

VirtexII 器件结构示意图如下：

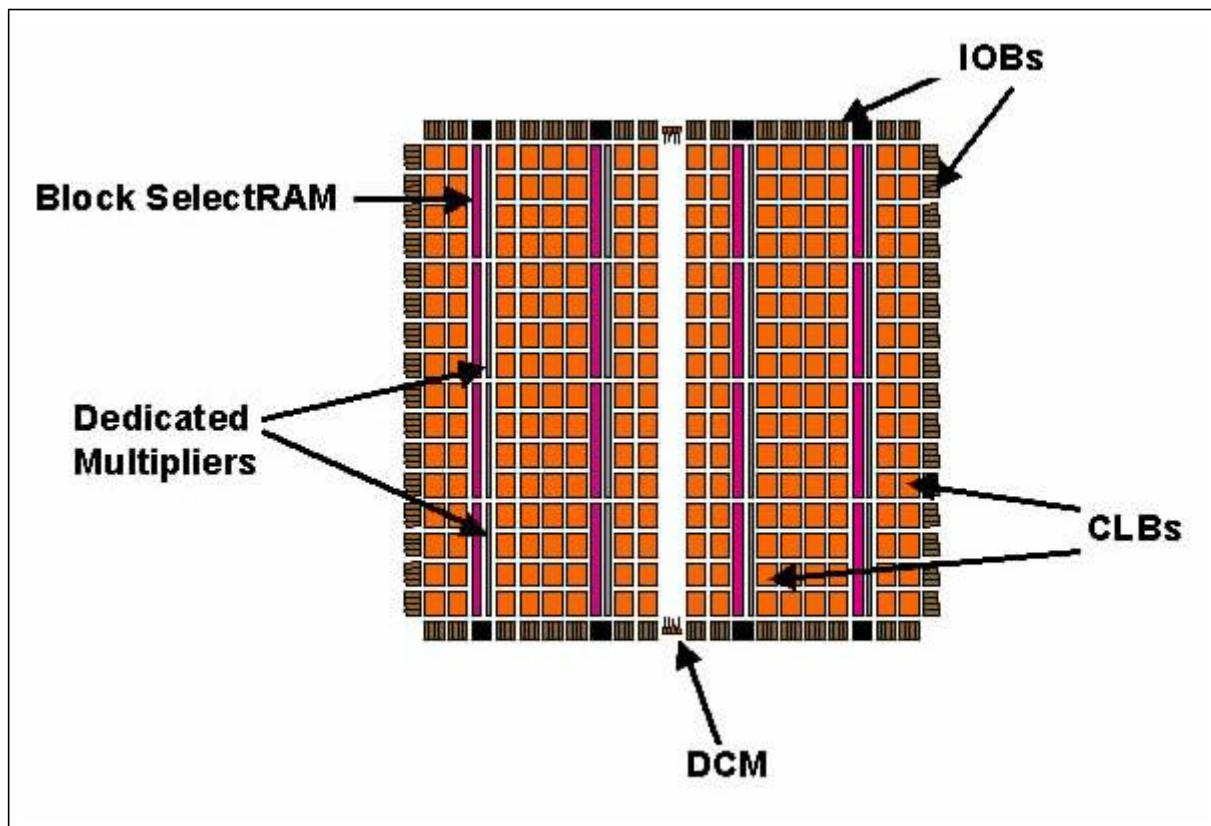


图3 VirtexII 结构示意图

VirtexII 器件在结构上与 Virtex 和 VirtexE 是相似的，也主要由 IOBs 和 CLBs 组成，但增加了一个专有乘法器结构，另外，在 IOB 和 CLB 中也有点不同，DCM 则是 DLL 的增强版。

3.3.1 CLB

VirtexII 的 CLB 与 Virtex Family 和 VirtexE Family 结构有点不一样。每个 CLB 含有 4 个相似的 Slice，在结构的安排上，采用阵列的安排。如下示意图：

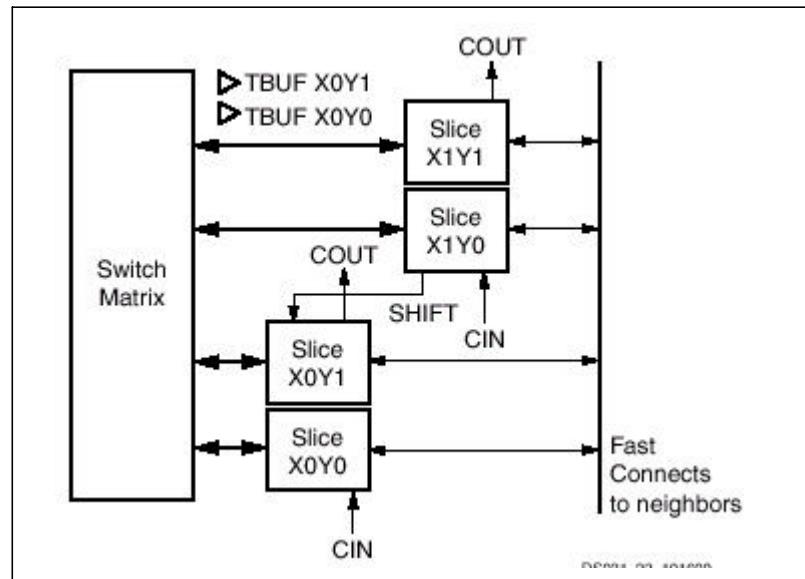


图4 VirtexII 的CLB结构示意图

与以往不同的是，每个CLB包含4个相似的Slice，4个Slice按照如上图的阵列排布。每个Slice都与一个开关矩阵紧密相接以便连到通用布线阵列（GRM）。在CLB中，还有内部的快速的互联线，保证4个slice之间快速的互联。

4个Slice分成两列，每列两个slice，每列有保留一条独立的进位链，但两列共用一个移位链（SHIFT）。如上图所示。

3.3.2 Slice

Slice基本元件包括：2个4输入LUT（G函数、F函数）、2个Storage element（FFX、FFY，一般用做D触发器）。另外，为了实现某些高性能电路，Slice中还集成了carry logic、 arithmetic logic gates、 multiplexers等元件。如下图所示：

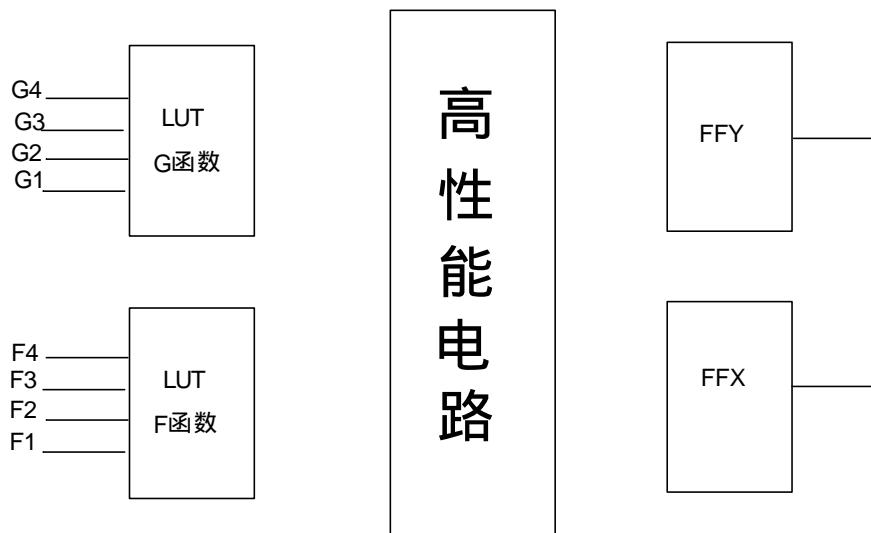


图5 SLICE结构示意图

值得大家注意的是：在许多的FPGA设计中，由于设计者没有注意利用Slice中的一些高速特性，导致设计速度上不去，或者FPGA资源实际利用率不高。

VirtexII 的slice具体结构参见下图：

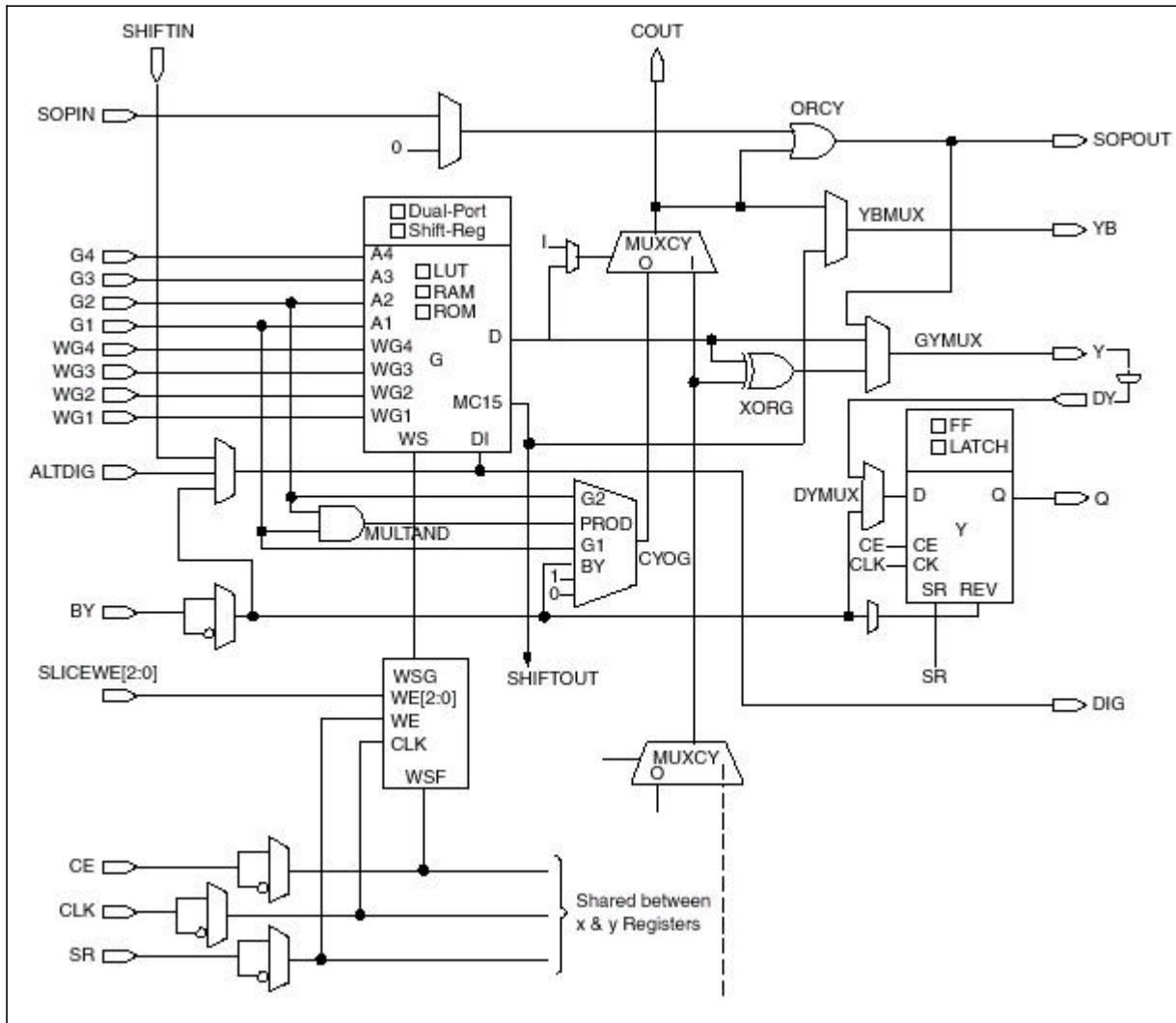


图6 VirtexII 的Slice 结构图（上半部分）

VirtexII 的Slice 增加了不少的结构，下面分别简要介绍。

3.3.3 LUT

每个Slice 包含两个4输入的LUT（分别是F函数和G函数），其最基本的功能是Function Generator（4000系列的功能），也就是当做组合逻辑电路。另外LUT还可实现RAM和移位寄存器的功能，这两个功能在随后的章节会详细介绍。

LUT本身是一个16X1的RAM结构，它的4个输入（G函数：G1~G4，F函数：F1~F4）其实是RAM的地址线。通过对RAM中各存储单元进行配置，可灵活配置成任意1~4输入任意组合逻辑；或者直接用它作RAM或ROM，这本身就是它原来的特点。

LUT即可配置成单口RAM，也可配置成双端口RAM（输入脚WG4~1与此有关），详情参见本章“Distributed RAM”部分。



要提醒大家注意的是：信号在CLB中通过LUT的延时是固定的，不管你是几输入的函数（最多4输入）。

关于LUT的使用，还可参见本章补充说明部分。

3.3.4 Shift Register LUT (SRL)

一个LUT可把16个移位寄存器放在一个LUT中实现，从而大大节省线延时和面积，提高设计速度和设计性能。

如下图所示，通过移位链，CLB的4个Slice的的SRL16移位输出可串成一个大的移位链，从而形成更大的移位寄存器组。LUT的MC15就是移位的输出。每个Slice 的G函数的MC15输出接到DIF_MUX的SHIFTIN端，作为F函数移存器的shiftin， F函数的MC15输出作为该slice的SHIFTOUT。

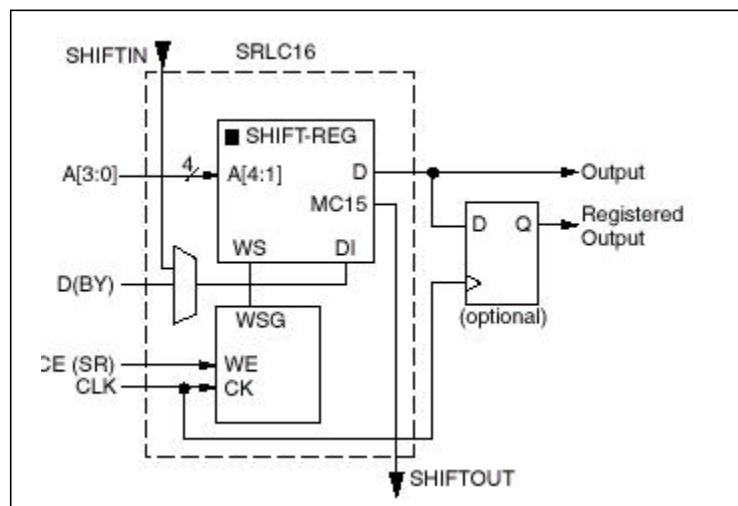


图7 SRL的移位链

在使用SRL时请注意：不能使用异步复位，一个Shift Registers LUT只能有一个数据输出和一个数据输入。

3.3.5 MUXFX

与基本的LUT相比，VirtexII 的Slice 增加了MUXF7、MUXF8的功能。

MUXF5：MUXF5用于MUX 每个Slice 中两个LUT (F、G) 的输出，使输入函数位宽达9个。可在CLB中实现4选一的MUX。这个功能一般的工具都可实现。

MUXFX：MUXFX 可实现MUXF6、MUXF7、MUXF8的功能，输入为FXINA、FXINB，输出为FX。MUXFX用成F6、F7或F8要看Slice 具体位置。

MUXF6：用于MUX相邻两个Slice 的MUXF5的输出，因此只能是X0Y0、X1Y0两个Slice 的MUXFX可例化成MUXF6 (F6) 。通过MUXF6可在CLB (两个Slice) 中实现达19位宽的输入函数。即可在一个CLB (两个Slice) 中实现8选1的MUX。这功能一般工具都可实现。

MUXF7：用于MUX两个F6的输出。因此只能是X0Y1 这个Slice 的MUXFX可例化成MUXF7。通过MUXF7可在CLB中实现 达39位宽的输入函数。实现大于8选1的MUX，但一



般工具无法直接利用该功能，因此需要在代码风格上加于实现。但通过F7可在CLB中的4个Slice 实现一个16选1的MUX。

MUXF8：用于MUX相邻两个Slice 的F7。因此只能是X1Y1这个Slice 的MUXFX可例化成 MUXF8。通过MUXF8实现更宽的函数。通过F8可在两个CLB中实现32选1的MUX。

MUXF5、MUXFX的使用和连接如下图示意：

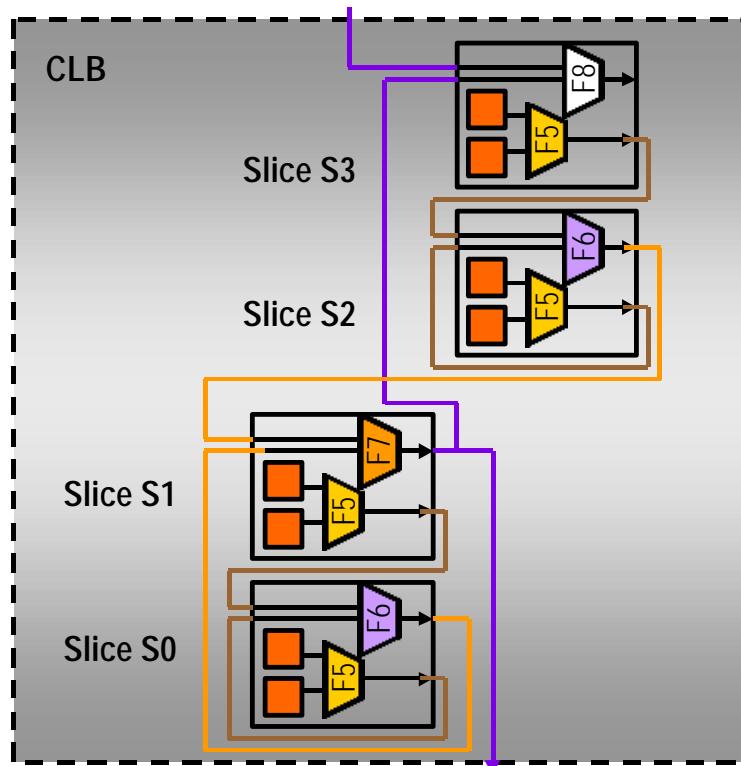


图8 VirtexII的MUXFX连接图

3.3.6 Carry Logic 和Arithmetic Logic Gates

Arithmetic Logic Gates包括一个XOR和一个MULTIAND，所处位置参见“VirtexII 的Slice 结构图（上半部分）”。

与基本Slice 中的进位链结构一样，VirtexII 通过MUXCY和XOR门可实现快速进位的计数器和通过MUXCY实现宽函数输入的级联（数据流从下往上）。

进位链结构如下图所示：

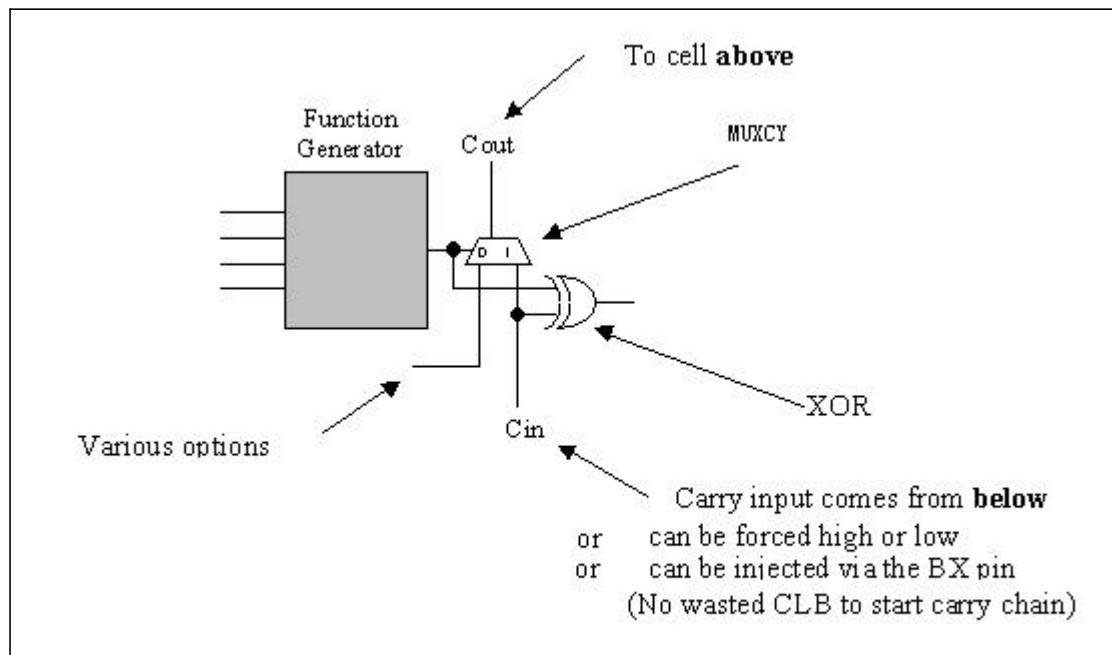


图9 进位链结构示意图

下图是一个采用进位链实现3bit全加器示意图：

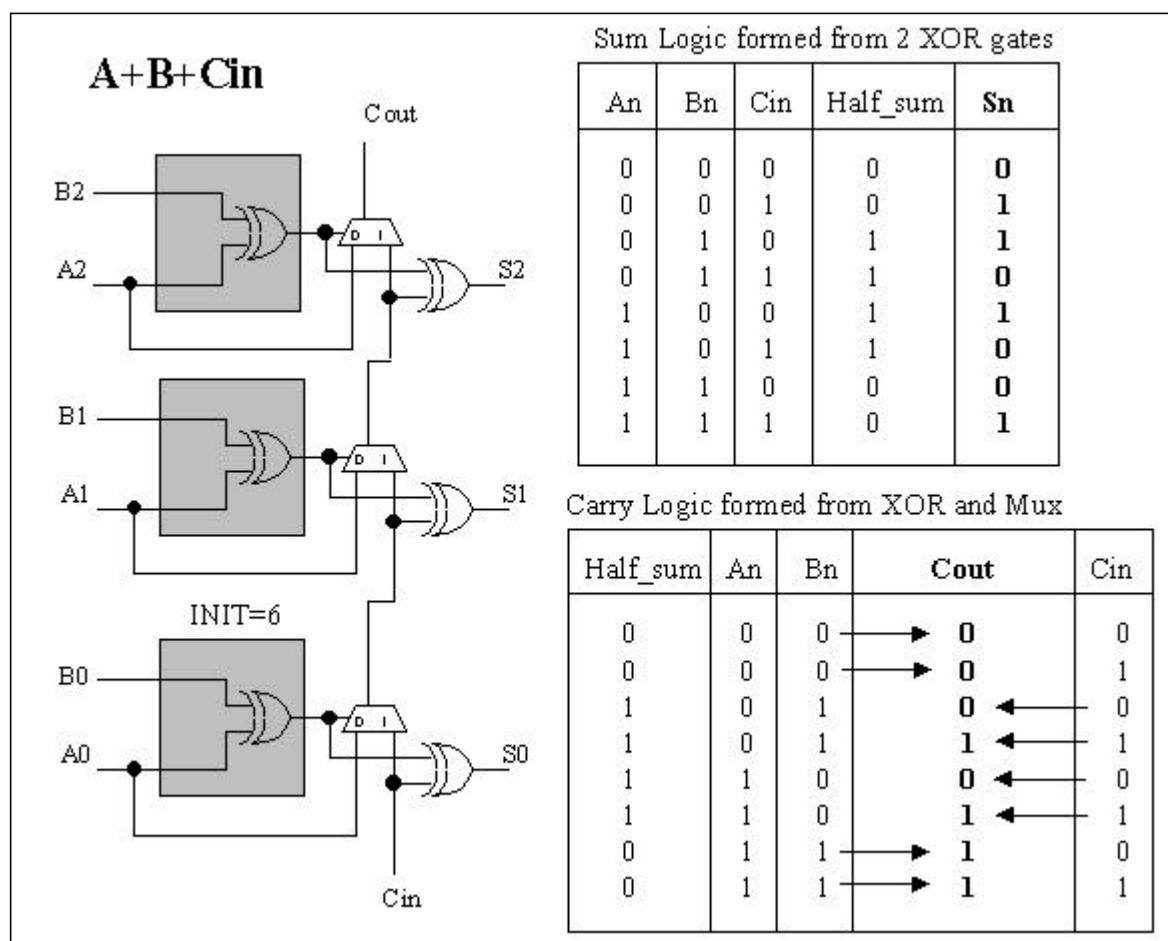


图10 使用进位链实现加法器

采用进位链，还可以实现一些特定的大的组合电路，如下图所示：

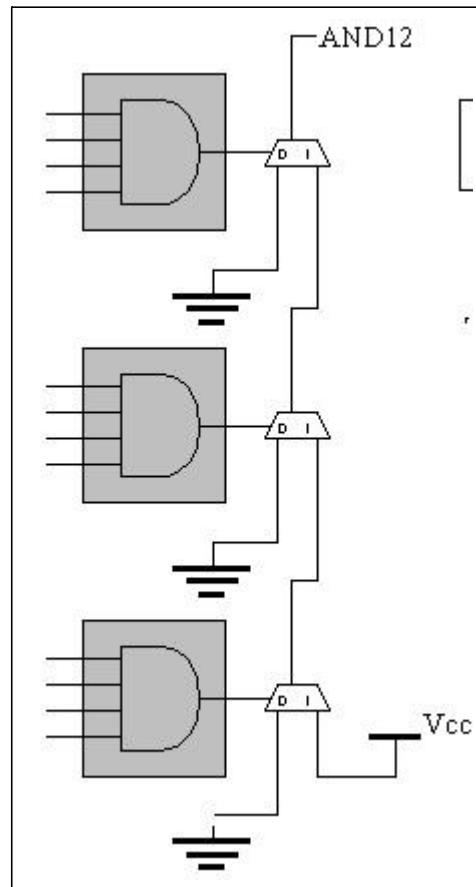


图11 使用进位链级联实现高速宽函数运算

由于乘法器可看成累加器，因此，使用专有进位链还可实现乘法器。

在VirtexII中，与以往的器件不同的是，每个CLB提供了两条独立的进位链，如下图所示：

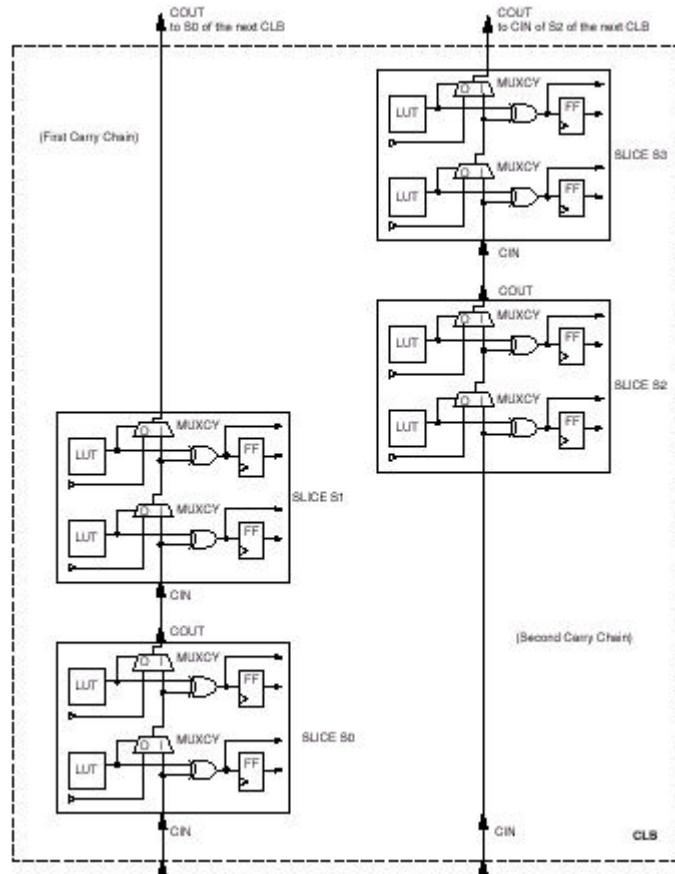


图12 VirtexII 的两个独立进位链

注意：一般的综合工具目前无法直接利用该进位链。

3.3.7 SOP

VirtexII 的每个Slice 中有一个OR，名叫ORCY。用于把Slice 中的进位链在水平方向上级联起来，形成一个大的、灵活的SOP链。如下示意图：

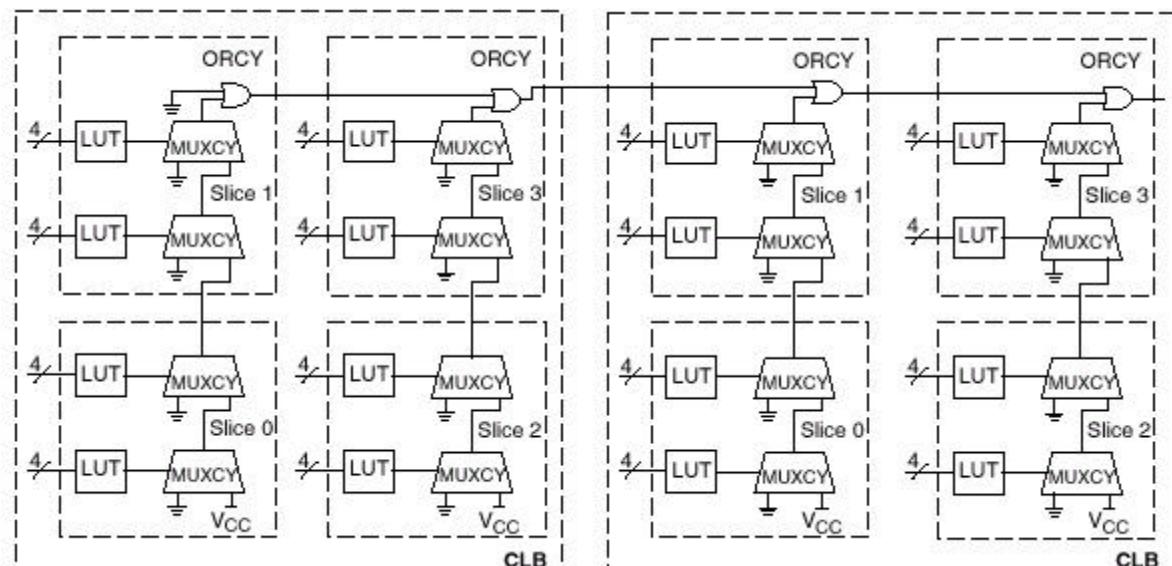


图13 VirtexII 的SOP 链

上图中，每一路纵向结构的4个LUT和4个MUXCY形成一个16输入的与门，横向的ORCY连接成4输入的或门。

3.3.8 FFX/FFY

VirtexII 的FFX 和FFY 与基本的FFX/FFY单元的功能基本一致。只是提供了4个attribute：SRHIGH、SRLOW、INIT1和INIT0。前两者用于描述SR（即外部复位信号）控制的复位属性：复位为0或1。后两者用于描述在没有外部复位信号时，芯片在上电配置时（configuration 或通过全局的GSR网络）的存储单元复位属性：复位为0或为1，这些属性可通过UCF描绘来实现。

存储单元结构如下示意：

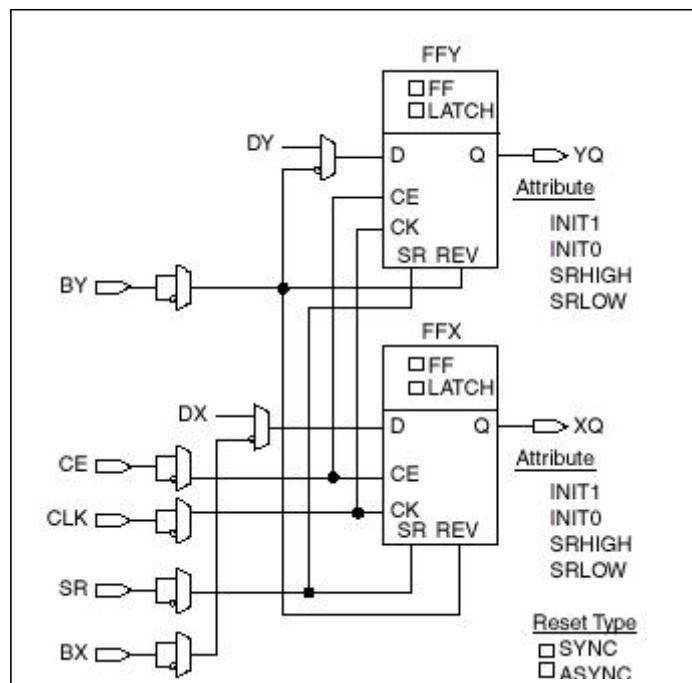


图14 FFX/FFY结构示意图

上图的DY是G函数发生器（LUT）输出信号Y在Slice 外部直接反馈进来的信号（可参见“VirtexII 的Slice 结构示意图”）。

3.4 Memory

3.4.1 Distributed RAM

VirtexII的LUT除了可产生单端RAM外还可产生双端RAM。当配置成单端RAM时，是同步写异步读的模式；当配置成双端RAM时，也是同步写异步读模式，且是一端口可读可写（R/W），另一端口只读。

一个VirtexII的CLB含4个Slice，可配置成如下的分布式RAM结构：

表1 VirtexII 的分布式RAM 配置表

模式	型号(深度X宽度)	Lut 使用个数
单端RAM	16 x 1	1
	32 x 1	2
	64 x 1	4
	128 x 1	8
双端RAM	16 x 1	2
	32 x 1	4
	64 x 1	8

下面两图是Distributed RAM的应用例子：

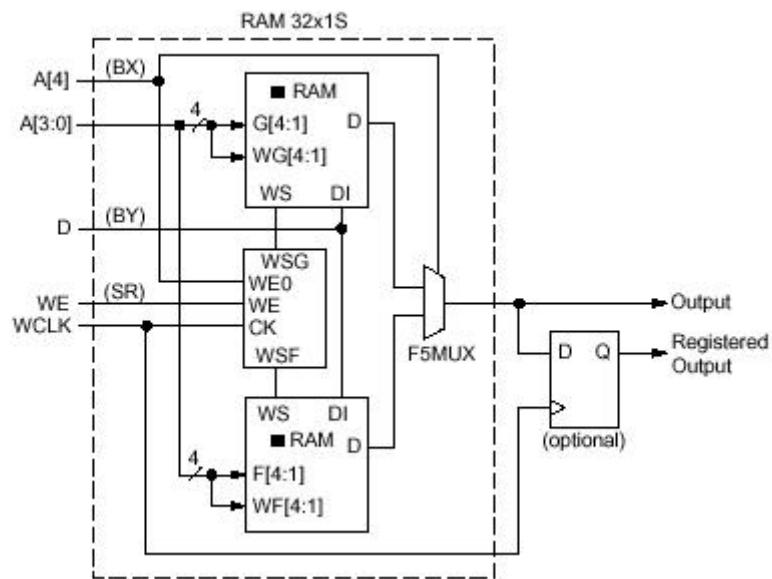


图15 单端口32x1 RAM

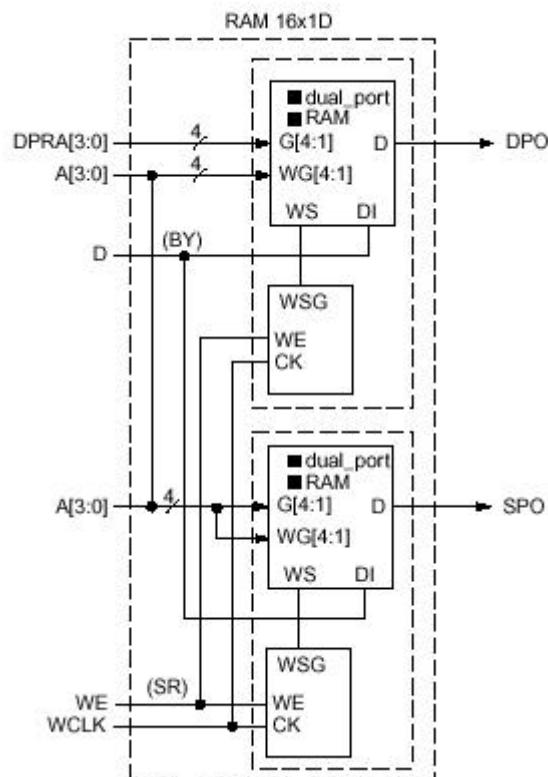


图16 双端口16x1 RAM

3.4.2 Block RAM

VirtexII 的Block RAM资源比以往的增加很多。每个Block RAM 有18k bit 。在整个VirtexII 系列中，Block RAM一般按如两列、4列或6列的规律进行分布：

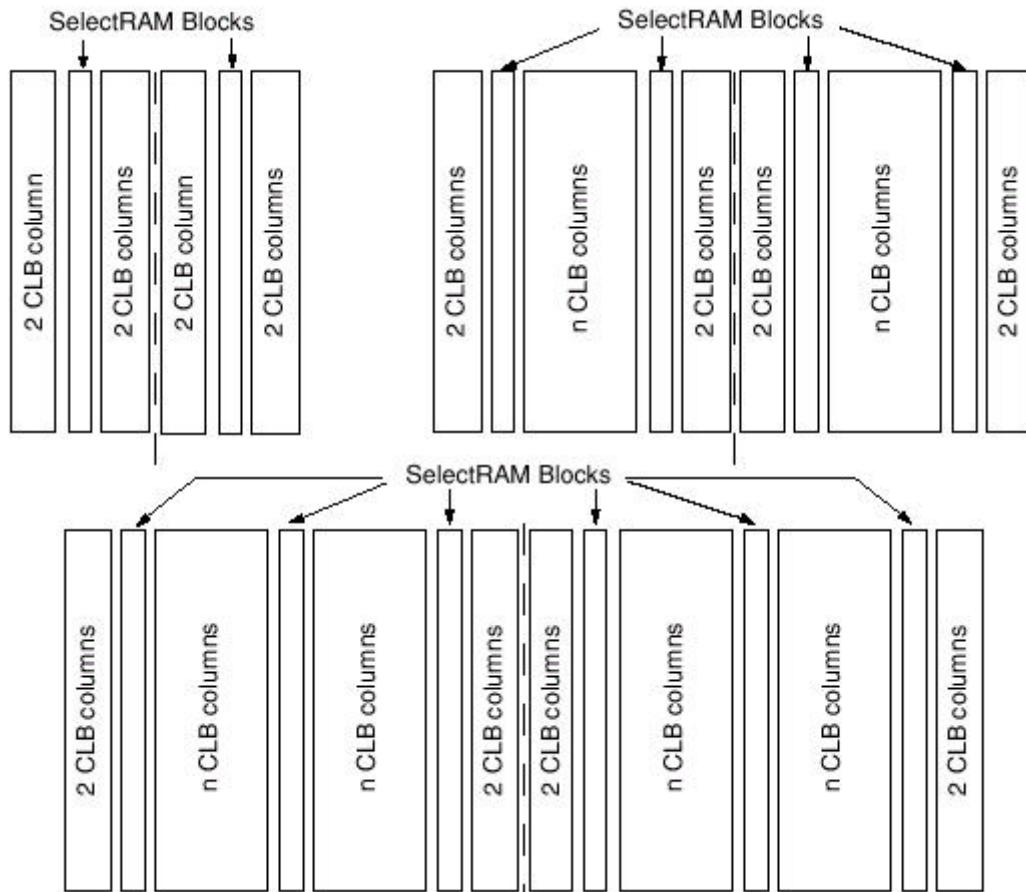


图17 VirtexII 的Block RAM 分布规律

其中的N 等于该器件CLB的列数除以4。具体的各系列的块数如下表：

表2 VirtexII 的BlockRAM 分布表：

Device	Columns	SelectRAM Blocks	
		Per Column	Total
XC2V40	2	2	4
XC2V80	2	4	8
XC2V250	4	6	24
XC2V500	4	8	32
XC2V1000	4	10	40
XC2V1500	4	12	48
XC2V2000	4	14	56
XC2V3000	6	16	96
XC2V4000	6	20	120
XC2V6000	6	24	144
XC2V8000	6	28	168
XC2V10000	6	32	192

由于块RAM有18bit，因此可支持奇偶校验的功能，每个端口可配成如下结构：

表3 带奇偶校验位的Block RAM配置表

Configuration	Depth	Data bits	Parity bits
16K x 1	16Kb	1	0
8K x 2	8Kb	2	0
4K x 4	4Kb	4	0
2K x 9	2Kb	8	1
1K x 18	1Kb	16	2
512 x 36	512	32	4

VirtexII 的block RAM 支持三种写模式:

Write first : new data is written , and then appears on the RAM output 。



图18 Write first 模式

Read first: Previous data value is read from the WRITE address, then held on the output during the WRITE operation 。

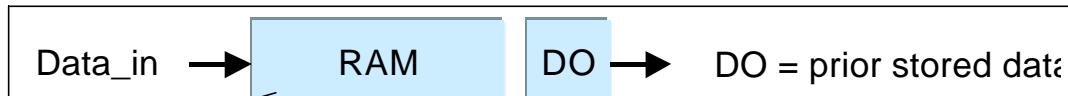


图19 Read first 模式

NO CHANGE: RAM output only changes when WE is inactive。

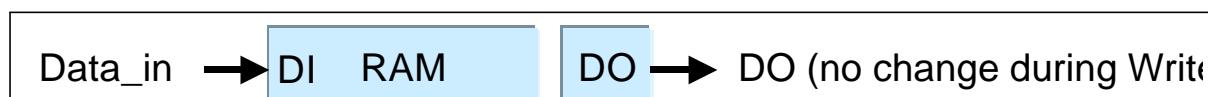


图20 No Change 模式

关于Block RAM更多的内容，可参见“sp_block_mem.pdf”。

3.5 乘法器资源

VirtexII 系列提供有专门的乘法器结构：18bits x 18bits 。 VirtexII 的乘法器资源分布图与 Block RAM 的分布图一样，每个乘法器块紧靠着Block RAM，共用4个开关矩阵，如下示意：

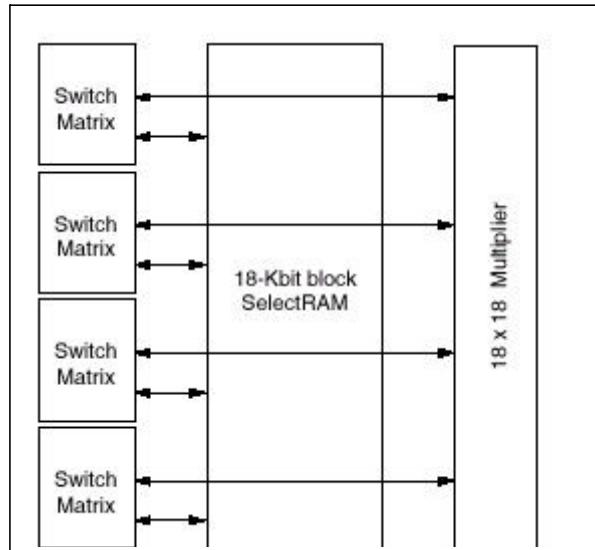


图21 乘法器与Block RAM

器件中乘法器位置如下图：

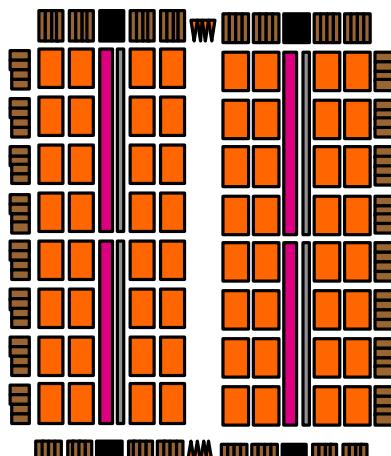


图22 XC2V40的乘法器

乘法器结构如下：

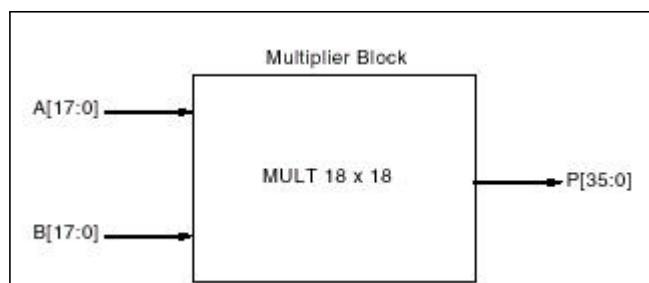


图23 乘法器块

乘法器可实现高速的低工耗的乘法器、累加器，速度如下表：



表4 VirtexII 乘法器速度表（厂家数据）

4x4 signed	~255 MHz
8x8 signed	~210 MHz
12x12 signed	~170 MHz
18x18 signed	~140 MHz

3.6 IOB

一般的IOB，都包含存储单元、I/Obuf、输入延时线DELAY。

存储单元包括：输入寄存、输出寄存、三态输出控制线的寄存。输入、输出、三态控制线也可以不经过寄存，直接通过I/O buf输入输出。

IOB中提供5中I/O Buf：输入buf（IBUF）、输出buf（OBUF）、三态输出buf（OBUFT）、双向buf（IOBUF）、全局时钟输入buf（IBUFG）。

Virtex II 的IOB 基本结构与基本的IOB一样，只是增加了DDR的功能、增加了一些IO标准和DCI功能。

3.6.1 IOB结构

在VirtexII 的器件结构中，IOB的位置有较大的改变，这是因为，VirtexII的所有用户IO 可配成差分信号，一个差分信号需要一对IO 口，因此，在器件结构中，5个IOB共用一个开关矩阵。4个IOB可形成两对差分IO，如下示意图：

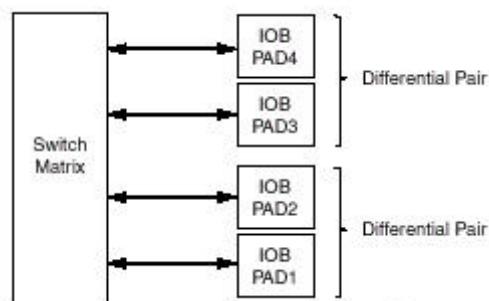


图24 VirtexII的IOB

VirtexII的IOB 与基本的IOB 结构相似，主要由三组存储单元、I/Obuf 和 输入延时线DELAY构成。

每组存储单元都由两个存储单元组成，用于实现DDR。输入和输出单元使用不同的时钟信号。两个存储单元通过DDR MUX来实现DDR，一般地，要求通过DCM来产生DDR的正反沿时钟信号。IOB的三组存储单元结构示意图如下：

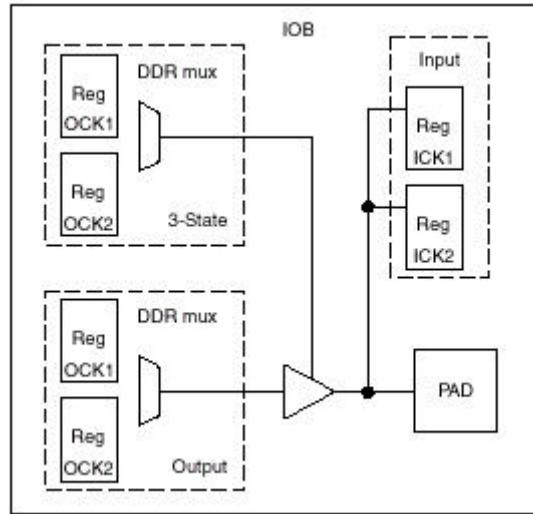


图25 VirtexII 的IOB中的DDR

具体的结构如下：

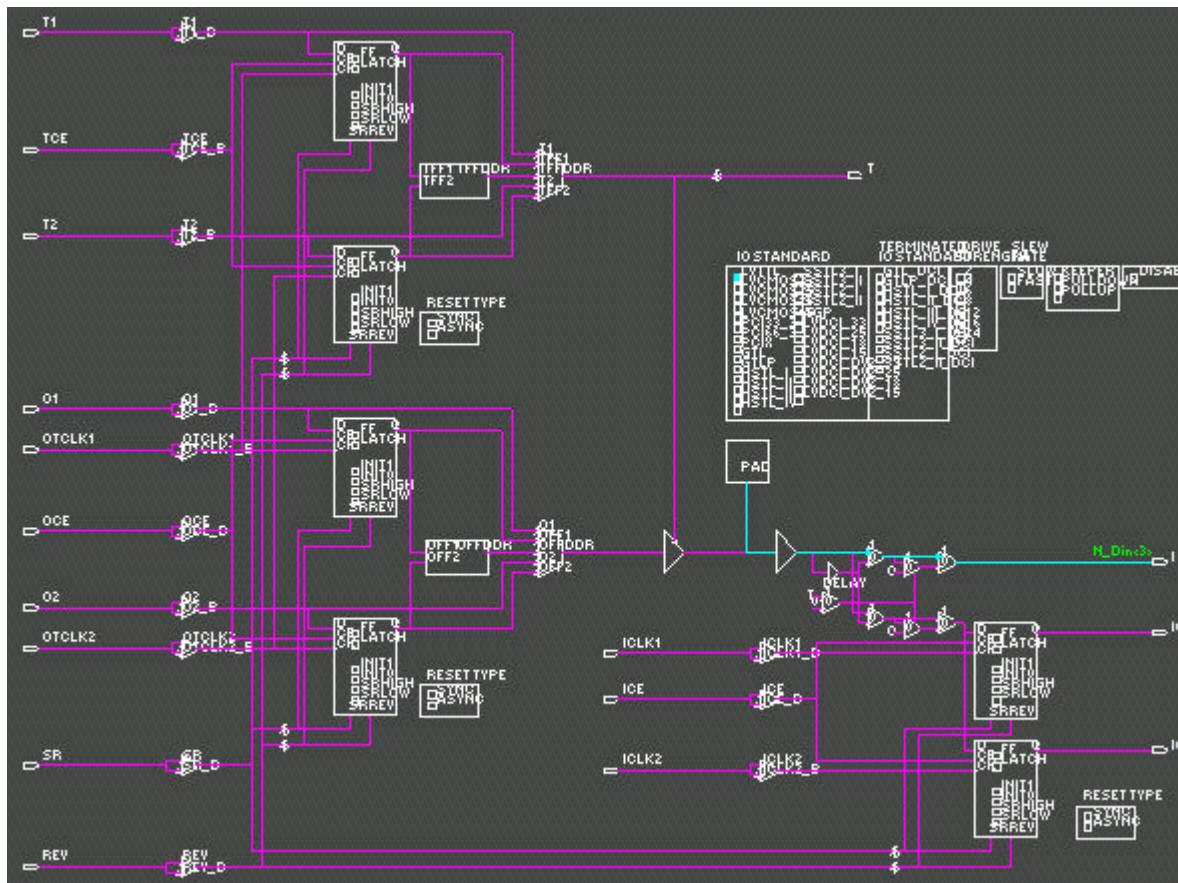


图26 VirtexII 的IOB 实际结构

3.6.2 Select I/O

VirtexII 的Select I/O 支持的标准有所增加。VirtexII 同样把器件分成8 个banks，每个banks 提供VRN和VRP参考电压，根据不同参考电压提供不同的 IO标准。



VirtexII 提供19 种signal-ended IO 标准，具体如下：

- LVTTL, LVCMOS (3.3V, 2.5V, 1.8V, and 1.5V)
- PCI-X at 133MHz, PCI (3.3V at 33MHz and 66MHz)
- GTL, GTLP
- HSTL (Class I, II, III, and IV)
- SSTL (3.3V and 2.5V, Class I and II)
- AGP-2X

提供如下的差分标准：

- LVDS, BLVDS, ULVDS
- LDT
- LVPECL

3.6.3 DCI

与以往不同的是， VirtexII 集成了DCI（Digital Controlled Impedance）功能，即通过特定的参考电压，在芯片内部提供IO管脚的特定匹配电阻，这样在单板上就不必增加匹配电阻，简化单板设计，增加设计的集成度。

3.7 Clock Resource

VirtexII 的时钟资源比以往增加了很多，最多可达一个芯片上提供16个全局时钟信号。

如果想要了解更多的信息，可参见“Virtex-II时钟资源（初稿）”。

3.7.1 Global Clock

VirtexII 芯片提供有16个时钟管脚，8个分布在芯片的顶部，8个分布在芯片的底部。这些时钟管脚还可以当作普通管脚使用，这点是与以往器件较不一样的。以顶部时钟为例，8个时钟管脚都与一个开关矩阵相连，由开关矩阵切换出16个时钟信号线，16时钟信号线既可以是顶部8个时钟脚或底部8个时钟脚或内部产生的时钟信号。16根时钟信号线通过8个时钟MUX，MUX出8个全局时钟信号，与底部的8个全局时钟信号组成全芯片的16个全局时钟信号。

芯片的16个时钟管脚 布局如下示意：

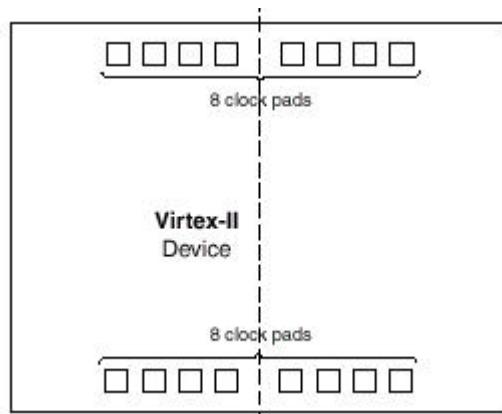


图27 VirtexII 的Clock Pads



具体的结构如下图，在顶部中央两边，各有一个开关矩阵，每个带4个时钟管脚，8个时钟信号连到顶部的开关矩阵切换出16个时钟信号连到下面的8个时钟MUX上。

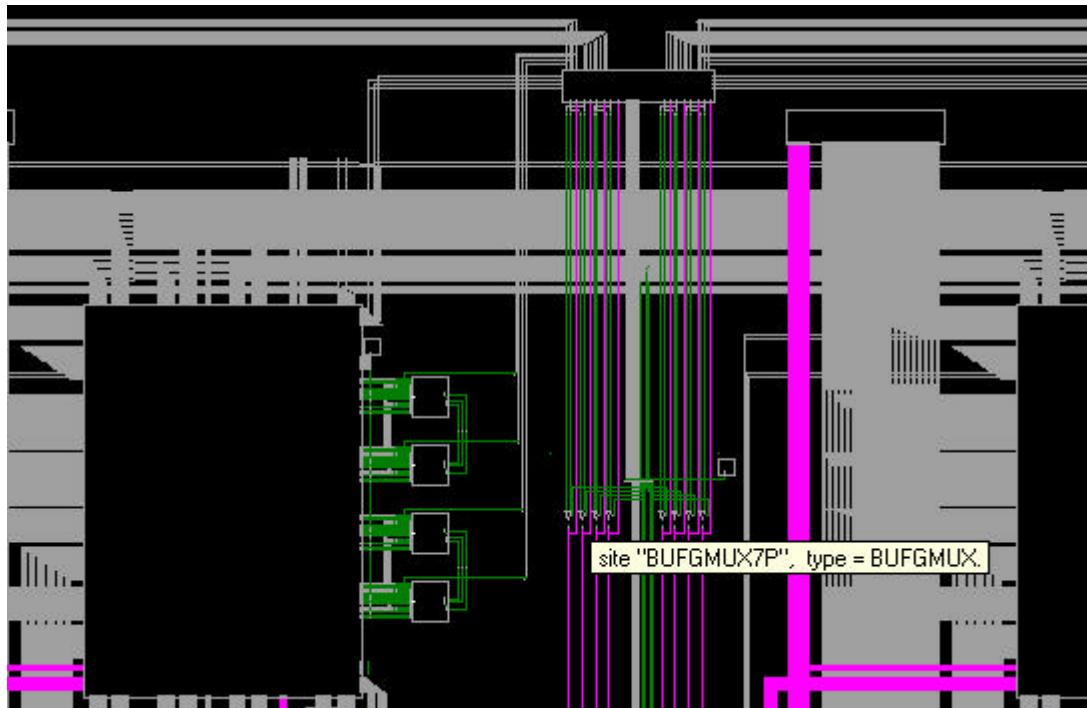


图28 VirtexII 的时钟（顶部）

在VirtexII 的器件中，16根的全局时钟信号通过时钟MUX 和一定的布线规律，可以保证芯片的4个区域内最多都可以获得8个全局时钟信号，因此，在安排时钟管脚时必须考虑一下。

一下是VirtexII 时钟资源分布原理：

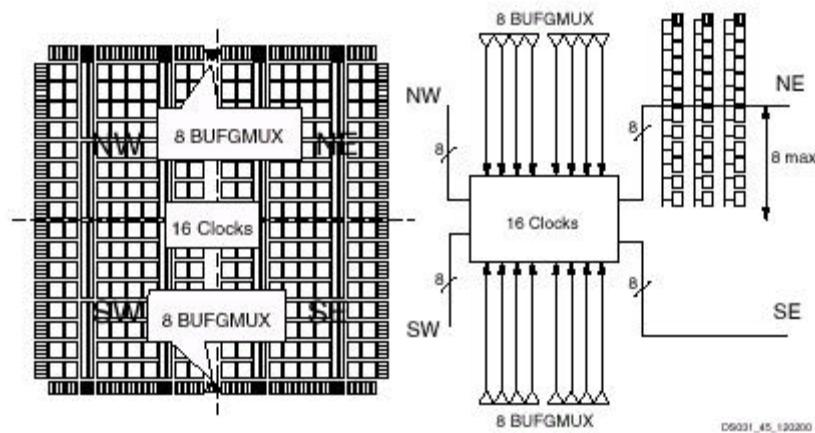


图29 VirtexII 的时钟资源分布原理

3.7.2 CLK MUX

在VirtexII 的器件中，作为时钟MUX的BUFGMUX 可配成多种形式，因此全局时钟资源可由时钟管脚、内部信号或DCM的输出来驱动。

BUFGMUX的结构如下：（顶部）

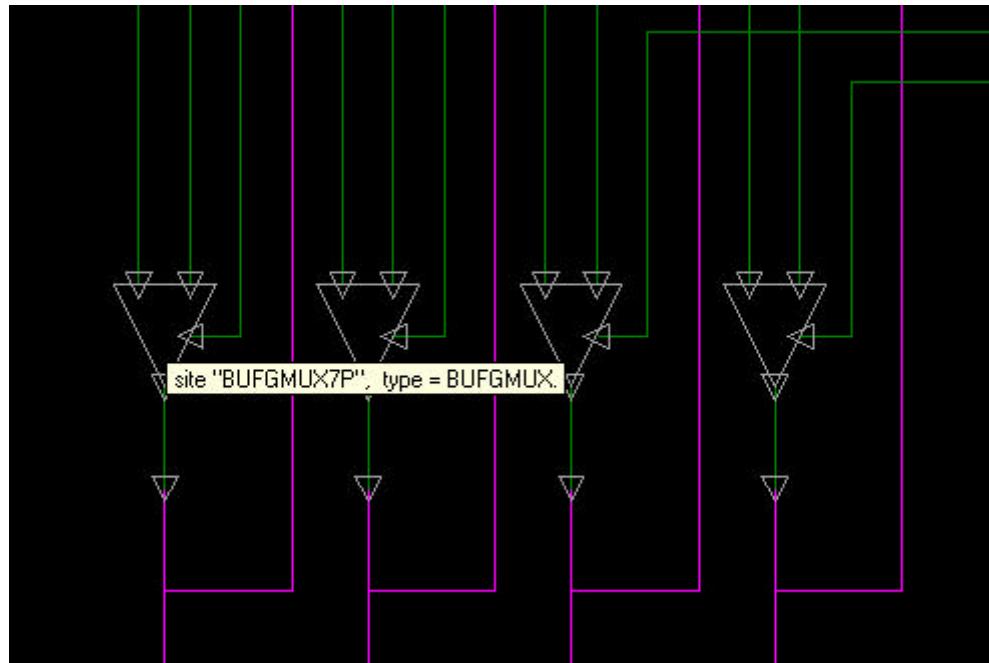


图30 VirtexII的BUFGMUX

该BUFGMUX可有如下几种配置：

BUFG: 即普通的全局时钟BUF。如下结构图：

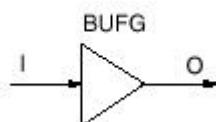


图31 VirtexII的BUFG

BUFGCE: 带时钟Enable 的全局时钟Buf。如下结构图：

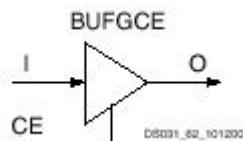


图32 VirtexII 的BUFGCE

BUFGMUX: 时钟MUX。如下结构图：

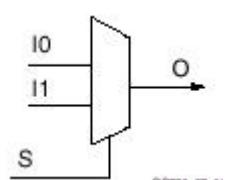


图33 VirtexII 的BUFGCE

3.7.3 DCM

VirtexII 器件结构对Virtex 的DLL做了增强，即为DCM： Digital Clock Manager 。 VirtexII 器件最多可达12个DCM。DCM一般分布在芯片的底部和顶部，每隔几列CLB列插入一个DCM。如下画出V2250芯片的8个DCM，4个在顶层，4个在底部。箭头指示的是底部从右边往左边第一个DCM。

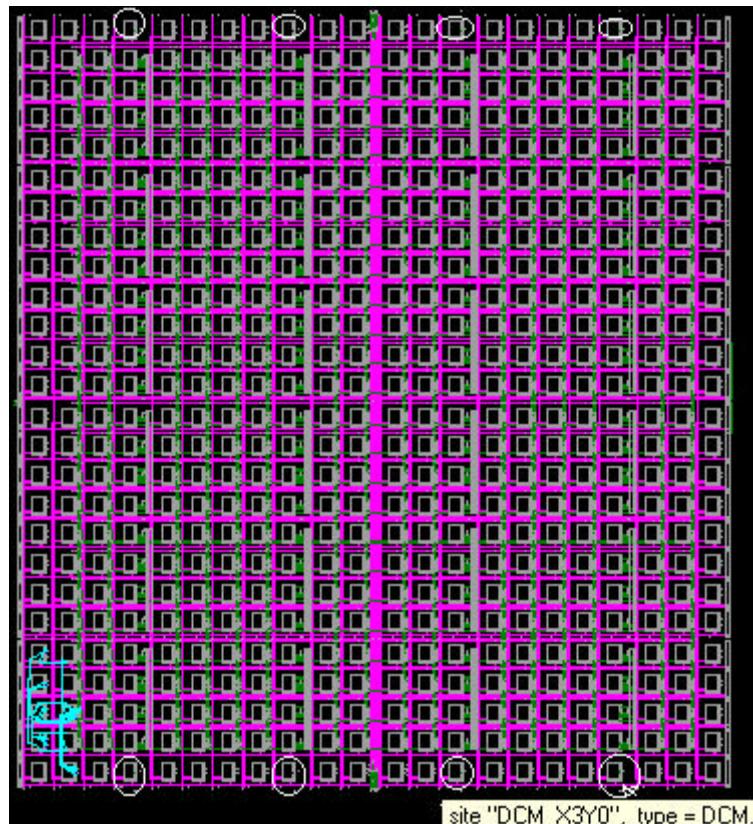


图34 VirtexII 250 的DCM 位置

VirtexII 系列器件的DCM分布表如下：



表5 VirtexII 的DCM分布表

Device	Columns	DCMs
XC2V40	2	4
XC2V80	2	4
XC2V250	4	8
XC2V500	4	8
XC2V1000	4	8
XC2V1500	4	8
XC2V2000	4	8
XC2V3000	6	12
XC2V4000	6	12
XC2V6000	6	12
XC2V8000	6	12
XC2V10000	6	12

VirtexII 的DCM 的符号如下：

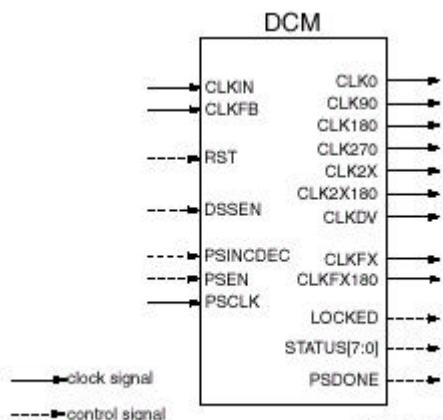


图35 VirtexII 的DCM

DCM是对DLL的增强，因此提供如下的功能：

DLL: Delay-Locked Loop。延时锁相环。具体原理与基本DLL是一致的。通过该延时锁相环可保证DCM的输入、输出时钟的0相位延时并提供时钟驱动。

DPS: Digital Phase Shifter。数字相移器。DCM可提供4相位的相移。

DFS: Digital Frequency Synthesizer。数字频率合成器。可提供如下公式的频率合成：

$$\text{FREQ}_{\text{CLKFX}} = (\text{M}/\text{D}) * \text{FREQ}_{\text{CLKIN}}$$

DSS: Digital Spread Spectrum: 数字扩频。

3.8 补充说明

注意：本节所有内容来自个人推测。



3.8.1 LUT如何配置成组合逻辑电路：揭开“门数增加，逻辑级数未变，但资源占用减少，速度更快”之谜

前面本文提到，LUT本身是一个16X1的RAM结构，它的4个输入其实是RAM的地址线。那么，它是怎么实现组合电路的呢？

假定我们要完成如下一个3输入与门（以LUT中的F函数为例）

$$F = F_4 \cdot F_3 \cdot F_2$$

该函数等价于：

$$F = F_4 \cdot F_3 \cdot F_2 \cdot F_1 + F_4 \cdot F_3 \cdot F_2 \cdot /F_1$$

式中，“/”表示非运算。该式表示当 $F_4 \sim 1 = "1111"$ or $"1110"$ 时， $F = "1"$ ，其它的值都是“0”，也就是3输入（ $F_4 \sim 2$ ）与门。

我们知道， $F_4 \sim 1$ 是RAM（LUT）的地址输入信号，我们只要将地址为“1111”和“1110”的存储单元置为“1”，其它的所有存储单元置为“0”，则该RAM的功能实际就是“ $F_4 \cdot F_3 \cdot F_2$ ”与门。不过，在实际实现时，只要将“1111”单元和“1110”单元中任意一个置“1”就可以了，同时 F_1 固定接“1”或“0”。

从中，我们可以得出结论：

1. 不管我们用LUT实现几输入的组合电路，在实际实现时都会变成4输入的组合电路。
2. 正是由于第1点，对于在一个LUT内可以实现的组合电路，不管你怎样化简，对LUT而言，毫无用处。
3. 只要是在一个LUT内实现的逻辑，不管是几输入，逻辑延时基本一样。

我们知道，面积优化、速度优化都是针对组合逻辑而言的，对Xilinx而言，就是LUT。根据上述结论，对FPGA设计而言，如果想速度更快，则应当努力减少路径上LUT的个数，而不是逻辑级数；如果想面积更小，则应当努力减少LUT的个数，而不是逻辑门数。这一点与ASIC设计完全不一样。

我们来看下面这个例子：

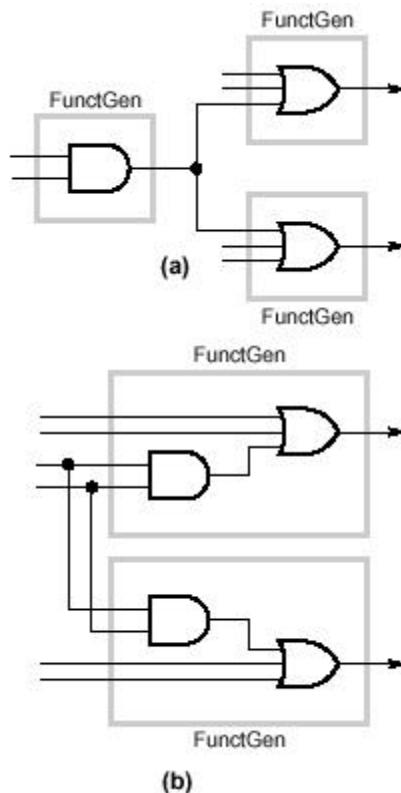


图36 门数增加，逻辑级数未变，但资源占用减少，速度更快

采用图a结构，我们知道一个LUT只有一个输出，因此前面的2输入与门要占用一个LUT，后面的2个三输入或门要各占用一个LUT，总共占用3个LUT，LUT级数是2级。

采用图b等效结构，虽然增加了一个2输入与门，并且逻辑级数与图a一样，也是2级，但我们根据LUT特点，它只占用2个LUT（2输入与门和3输入或门由一个LUT实现），LUT级数只有1级。

显然，这是一个“门数增加，逻辑级数未变，但资源占用减少，速度更快”典型案例。

3.8.2 解剖Block SelectRAM内部结构

对Block SelectRAM而言，我们只要掌握了单端口RAM，就很容易掌握双端口RAM。本节以单端口RAM为例进行说明。

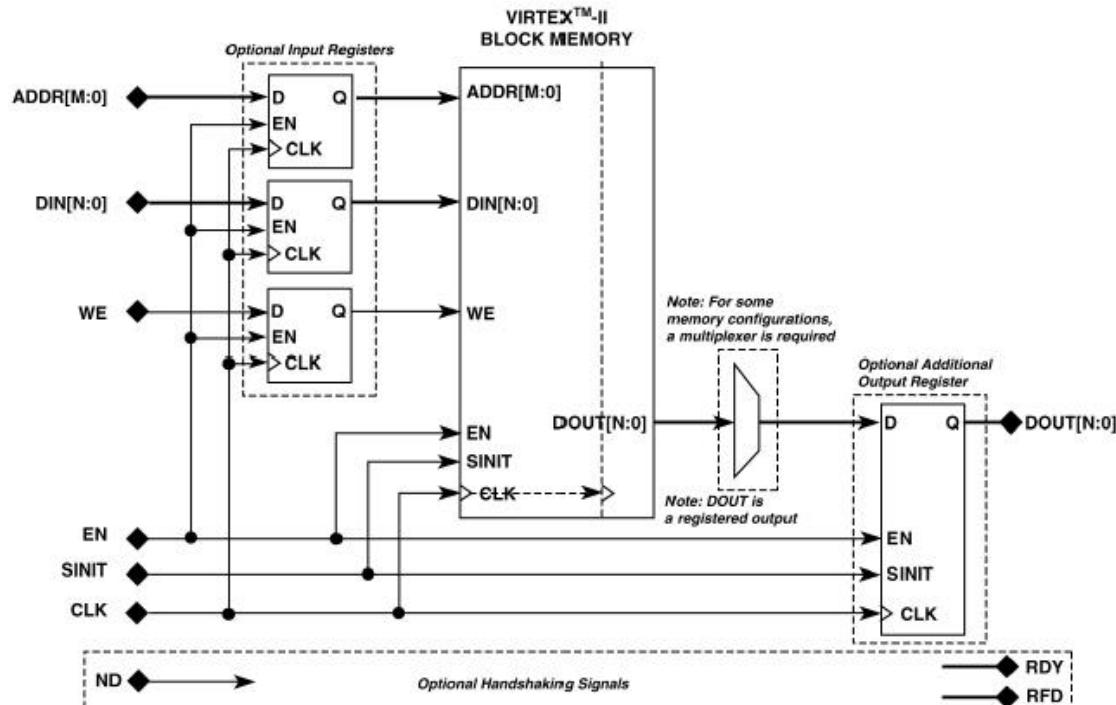


图37 完整的单端口 Block Select RAM

上图是一个完整的单端口RAM结构，它包括外围一些可选电路。我们这里准备讲的单端口RAM是上图中的核心部分Block Memory。

对 VirtexII Block Memory 而言，它的写有三种操作模式：Write First、Read First、No-read-on-write（No Change）。根据其输入输出信号相位关系，我们猜测其对应的电路分别如下：

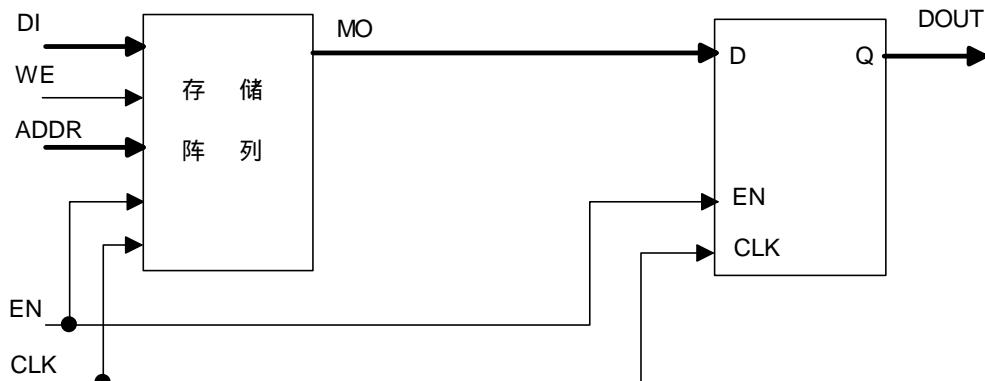


图38 Read first mode

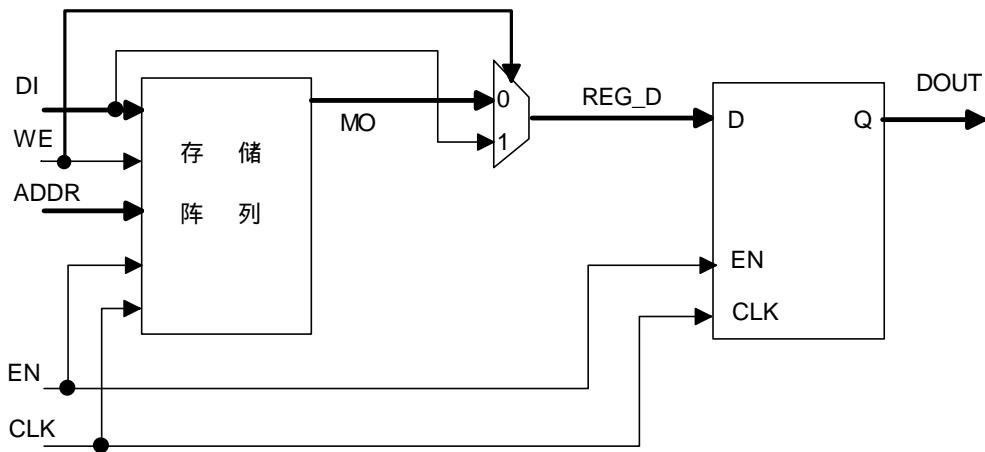


图39 Write first mode

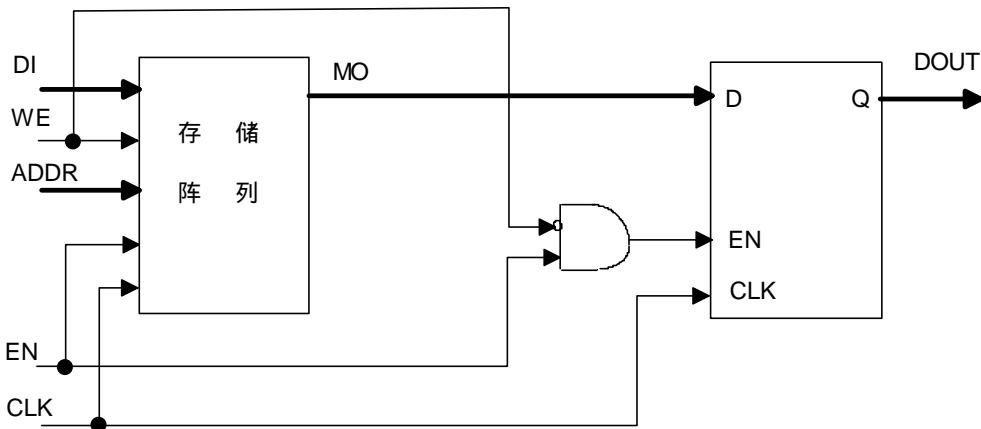


图40 No-read-on-write mode

图中，存储阵列的工作原理和Distributed RAM一样，可以等价看成Distributed RAM。

4 设计技巧

在设计过程中，经常遇到速度或面积问题：在功能基本正确之后，设计要么速度不满足要求，要么面积太大，或者两者都不满足设计要求。经常在速度和面积上花费大量的时间。

本章着重从速度和面积角度出发，考虑如何编写代码或设计电路，以获得最佳的效果。但是，有些方法是以牺牲面积来换取速度，而有些方法是以牺牲速度来换取面积，也有些方法可同时获得速度和面积的好处。具体如何操作，应当依据实际情况而定。

在处理速度与面积问题的一个原则是：向关键路径（部分）要时间，向非关键路径（部分）要面积。为了获得更高的速度，应当尽量减少关键路径上的LUT级数，尽量压缩线延时；为了获得更小的面积，在非关键路径（部分）上尽量优化电路结构，压缩面积。



特别提醒：由于工具的发展，本文提到的一些设计技巧可能在绝大部分情况下已经失效（工具可以代劳）。不过，在一些复杂电路，估计工具可能还无能为力。因此，本文提供这些设计技巧的一个主要目的，是为了让大家在遇到困难时，不至于束手无策，可以尝试本文所提供的设计技巧，看看是否可以解决问题。

注意：由于历史原因，本章节所举的代码都采用的是VHDL语言。不过，我们认为语言是次要的，关键是电路设计技巧。另外，本章节中有部分内容已经转换成verilog格式，可参见“VerilogHDL编码入门指导”。

4.1 合理选择加法电路

4.1.1 串行进位与超前进位

改变赋值语句的顺序和使用信号或变量可以控制设计的结构。每一个VHDL信号赋值、进程或元件的引用对应着特定的逻辑。每个信号代表一条信号线。使用这些结构，能将不同的实体连接起来，实现不同的结构。下面的VHDL实例为加法器的进位链电路的两种可能的描述。

例、串行进位链

```
-- A is the addend
-- B is the augend
-- C is the carry
-- Cin is the carry in
C0 <= (A0 and B0) or
    ((A0 or B0) and Cin);
C1 <= (A1 and B1) or
    ((A1 or B1) and C0);
```

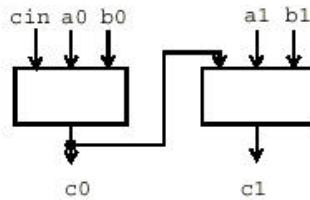


图41 串行进位

例、超前进位链（并行结构）

```
-- Ps are propagate
-- Gs are generate
p0 <= a0 or b0;
g0 <= a0 and b0;
p1 <= a1 or b1;
g1 <= a1 and b1;
c0 <= g0 or (p0 and cin);
```



$c1 \leq g1 \text{ or } (p1 \text{ and } g0) \text{ or }$

$(p1 \text{ and } p0 \text{ and } cin);$

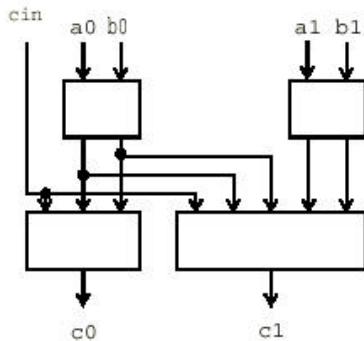


图42 超前进位

显然，第二种方法速度快，但面积大；第一种方法速度慢，但面积小。

Virtex系列FPGA结构中本身含有进位链结构，从其实现结构来看，应当是第一种方式。由于进位链是FPGA的专有资源，因此其实现速度比一般的串行进位快多了。

4.1.2 使用圆括号处理多个加法器

控制设计结构的另一种方法是使用圆括号来定义逻辑分组。下面的例子描述了一个4输入的加法器分组及其实现结果。

例、 $Z \leq A + B + C + D;$

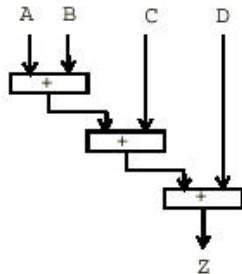


图43 串行加法电路

用圆括号重新构造的加法器分组如下所示。

例、 $Z \leq (A + B) + (C + D);$

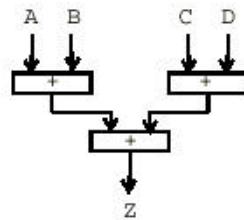


图44 并行加法电路

上述两种方法的在速度和面积上的区别是：



第一种方法（不带括号）：面积小，但整体速度慢。但是，如果信号D是关键路径，其它信号是非关键路径；或者，设计中关键路径与A、B、C、D无关，则应当采用这种方法。

第二种方法（带括号）：面积大，但整体速度快。如果对A、B、C、D的时序要求都比较苛刻，应当采用这种方法。

4.2 IF语句和Case语句：速度与面积的关系

IF语句指定了一个优先级编码逻辑，而Case语句生成的逻辑是并行的，不具有优先级。IF语句可以包含一套不同的表达式，而Case语句比较的是一个公共的控制表达式。通常，Case结构速度较快，但占用面积较大，所以用Case语句实现对速度要求较高的编解码电路。IF-Else结构速度较慢，但占用的面积小。如果对速度没有特殊要求，而对面积有较高要求，则可用IF-Else语句完成编解码。不正确的使用嵌套的IF语句会导致设计需要更大的延时。为了避免较大的路径延时，不要使用特别长的嵌套IF结构。用IF语句实现对延时要求苛刻的路径（speed-critical paths）时，应将最高优先级给最迟到达的关键信号（Critical Signal）。有时，为了兼顾面积和速度，可以将IF和Case语句合用。

例、用IF-Then-Else完成8选1多路选择器

```
MUX6to1:process(sel,in)
begin
    if(sel= "000") then
        out <= in(0);
    elseif(sel = "001") then
        out <= in(1);
    elseif(sel = "010") then
        out <= in(2);
    elseif(sel = "011") then
        out <= in(3);
    elseif(sel = "100") then
        out <= in(4);
    else
        out <= in(5);
    end if;
end process;
```

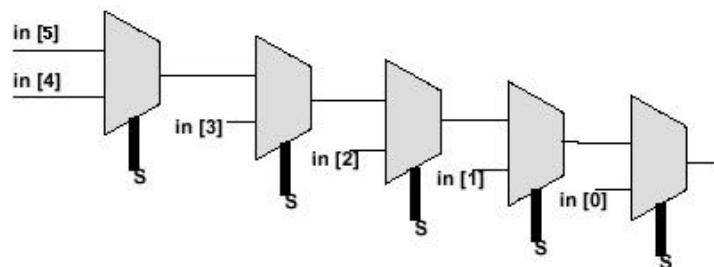




图45 if-else完成多路选择

下面的例子是用Case语句完成8选1多路选择器的VHDL实例。在大多数FPGA结构中能够在单个CLB中完成一个4选1的多路选择器，Virtex可以在单个CLB中完成一个8选1的多路选择器。而用IF-Else语句需要多个CLB才能完成相同功能。因此，Case语句生成的设计速度更快延时更小。

例、用Case实现8选1多路选择器

```
MUX8to1: process( C, D, E, F, G, H, I, J, S )  
begin  
    case S is  
        when "000" => Z <= C;  
        when "001" => Z <= D;  
        when "010" => Z <= E;  
        when "011" => Z <= F;  
        when "100" => Z <= G;  
        when "101" => Z <= H;  
        when "110" => Z <= I;  
        when others => Z <= J;  
    end case;  
end process;
```

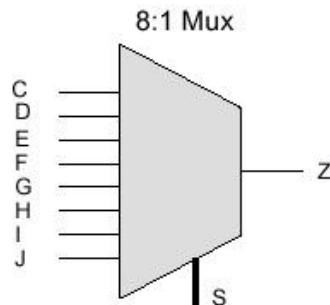


图46 case语句完成电路选择

4.3 减少关键路径的逻辑级数

在FPGA中，关键路径（critical path）上的每一级逻辑都会增加延时。为了保证能满足时间约束，就必须在对设计的行为进行描述时考虑逻辑的级数。减少关键路径延时的常用方法是给最迟到达的信号最高的优先级，这样能减少关键路径的逻辑级数。下面的实例描述了如何减少关键路径上的逻辑级数（注：前面提到的串行加法器也是一个案例）。

4.3.1 通过等效电路，赋予关键路径最高优先级

例、此例中critical信号经过了2级逻辑

```
if (clk'event and clk ='1') then
```



```
if (non_critical='1' and critical='1') then
    out1 <= in1;
else
    out1 <= in2;
end if;
end if;
```

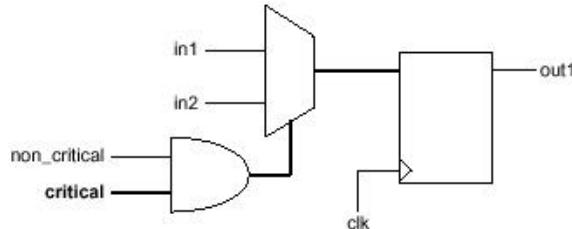


图47 critical信号经过2级逻辑

为了减少critical路径的逻辑级数，将电路修改如下，critical信号只经过了一级逻辑。

```
signal out_temp : std_logic;
process (non_critical, in1, in2)
begin
    if (non_critical='1') then
        out_temp <= in1;
    else
        out_temp <= in2;
    end if;
end process;
process(clk)
begin
    if (clk'event and clk ='1') then
        if (critical='1') then
            out1 <= out_temp;
        else
            out1 <= in2;
        end if;
    end if;
end process;
```

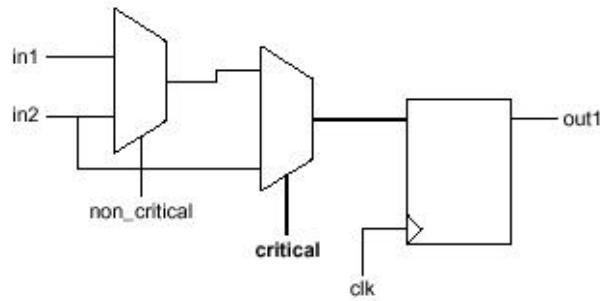


图48 critical信号只经过一级逻辑

注意：上述电路对FPGA而言基本没什么影响（4输入LUT特点），但对ASIC而言，却可行。

4.3.2 调整if语句中条件的先后次序

设计者习惯用if语句来描述电路功能，即便是在没有优先级的电路中，也采用有优先级概念的if语句来描述。

例如：

```
If 条件1 then
    Do action1
Else if 条件2 then
    Do action2
Else if 条件3 then
    Do action3
    .
    .
    .

```

在实际情况中，如果条件1、条件2、条件3不可能同时成立，则上述if语句无所谓谁优先。我们假定条件3所涉及的信号穿过的路径比较多，是关键路径，则上述写法会使设计更糟，应当改成：

```
If 条件3 then
    Do action3
Else if 条件1 then
    Do action1
Else if 条件2 then
    Do action2
    .
    .
    .

```

4.4 合并if语句，提高设计速度

前面提到，设计者习惯用if语句来描述电路功能，即便是在没有优先级的电路中，也采用有优先级概念的if语句来描述。

例如：

```
If 条件1 then
```



```
    信号置1
Else if 条件2 then
    信号置0
Else if 条件3 then
    信号置1
Else if 条件4 then
    信号置0
.
.
```

如果上述条件没有优先级，则为了获得更高设计速度，我们建议合并if语句中各条件：

```
If 条件1 or 条件3 then
    信号置1
Else if 条件2 or 条件4 then
    信号置0
.
.
```

4.5 资源共享

4.5.1 if语句

资源共享能够减少HDL设计所用逻辑模块的数量。否则，每个HDL描述都要建立一套独立的电路。下面的VHDL实例说明如何使用资源共享来减少逻辑模块的数量。

例、没有资源共享时用了4个加法器完成

```
if (...(siz = "0001")...) then
    count <= count + "0001";
else if (...((siz = "0010")...) then
    count <= count + "0010";
else if (...(siz = "0011")...) then
    count <= count + "0011";
else if (...(siz == "0000")...)then
    count <= count + "0100";
end if;
```

利用资源共享，采用下面的代码，可以节省2个加法器

```
if (...(siz = "0000")...) then
    count <= count + "0100";
else if (...) then
    count <= count + siz;
end if;
```

例、没有利用资源共享时用了2个加法器实现



```
if (select = '1') then
    sum<=A +B;
else
    sum<=C +D;
end if;
```

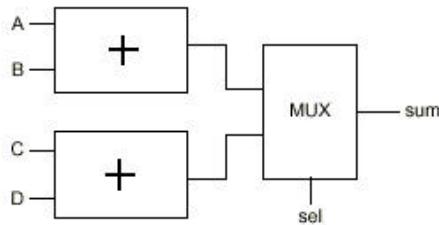


图49 资源共享前, 2个加法器

加法器要占用宝贵的资源。利用资源共享，修改代码如下，只用2个选择器和1个加法器实现，减少了资源占用。

```
if (sel ='1') then
    temp1 <=A;
    temp2 <=B;
else
    temp1 <=C;
    temp2 <=D;
end if;
sum <= temp1 + temp2;
```

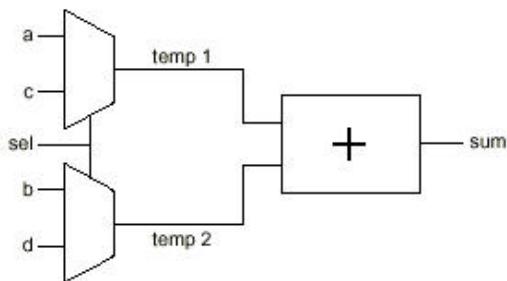


图50 资源共享后, 1个加法器

4.5.2 loop语句

与选择器相比，运算符占用更多的资源。如果在循环语句中有一个运算符，综合工具必须对所有的条件求值。下面的VHDL实例，综合工具用4个加法器和一个选择器实现。只有当“req”信号为关键信号时，才建议采用这种方法。

```
for i in 0 to 3 loop
    if (req(i)='1') then
        sum <= vsum + offset(i);
```



```
end if;  
end loop;
```

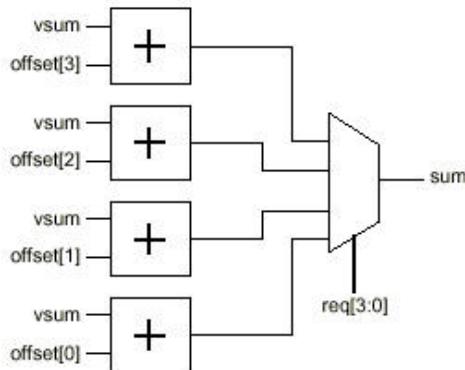


图51 资源共享前4个加法器

如果“req”信号不是关键信号，运算符应当移到循环语句的外部。这样在执行加法运算前，综合工具可以对数据信号进行选择。修改代码如下，用一个多路选择器和一个加法器即可实现。

```
for i in 0 to 3 loop  
    if (req(i)='1') then  
        offset_1 <= offset(i);  
    end if;  
end loop;  
sum <= vsum + offset_1;
```

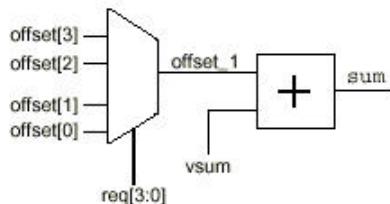


图52 资源共享后一个加法器

4.5.3 子表达式共享

一个表达式中子表达式包含2个或更多的变量。如果相同的子表达式在多个等式中出现，应共享这些运算，减少电路的面积。通过声明一个临时变量存储子表达式，然后在任何需要重复使用这个子表达式的地方用临时变量代替。

下面的VHDL实例描述了用相同的子表达式（a+b）完成一组简单的加法运算。

```
temp <= a + b;  
x <= temp;  
y <= temp + c;
```



4.5.4 综合工具与资源共享

通过设置FPGA CompilerII/FPGA Express的相应选项（缺省为共享），可以让综合工具自动决定是否要共享相同的子表达式，而不需声明一个临时变量存储子表达式。

但是综合工具对共享的代码编写规则有限制，如下：

1. 可共享的操作符为 * + - > < >= <=
2. 子表达式具有相同position

例如：

```
sum1 <= A + B + C;  
sum2 <= D + A +B;  
sum3 <= E + (A +B);
```

则sum1和sum3 可共享(A +B),但与sum2不共享，原因是表达式的计算是从左到右。

3. 必须在同一block（process）中，并且是在同一条件下控制，如下：

```
if (cond1 = ..) then  
    S1 <= A +B ;  
else  
    if (cond2 ....) then  
        S2 <= E + F ;  
    else  
        S3 <= G+ H ;  
    end if ;  
end if ;
```

则 S2 和S3可共享一个加法器，但与S1不可共享。

为了使代码更加具备通用性和可移植性，最好尽量自行编写共享资源代码。尤其是在进行ASIC设计时，所采用的综合工具在FPGA阶段和转ASIC阶段可能不同。

4.6 流水线（Pipelining）

流水线能动态的提升器件性能，它的基本思想是：对经过多级逻辑的长数据通路进行重新构造，把原来必须在一个时钟周期内完成的操作分成多个周期完成。这种方法允许更高的工作频率，因此提高了数据吞吐量。因为FPGA的寄存器资源非常丰富，所以对FPGA设计而言，流水线通常是一种先进的结构，而又不耗费过多的器件资源。然而，采用流水线后，数据通道变成多时钟周期，必须特别考虑设计的其余部分，解决增加通路带来的延迟。在定义这些路径的延时约束时必须特别小心。

当一个设计的寄存器之间存在多级逻辑时，其延时为源触发器的clock-to-out时间、多级逻辑的延时、多级逻辑的走线延时和目的寄存器的建立时间之和。工作时钟频率的提高受到这个延时的限制。采用流水线，减少了寄存器间的逻辑的级数。最终的结果是系统的工作频率提高了。

例、采用流水线前的电路



```
process(clk, a, b, c) begin
    if(clk'event and clk = '1') then
        a_temp <= a;
        b_temp <= b;
        c_temp <= c;
    end if;
end process;

Process(clk, a_temp, b_temp, c_temp)
begin
    if(clk'event and clk = '1') then
        out <= (a_temp * b_temp) + c_temp;
    end if;
end process;
```

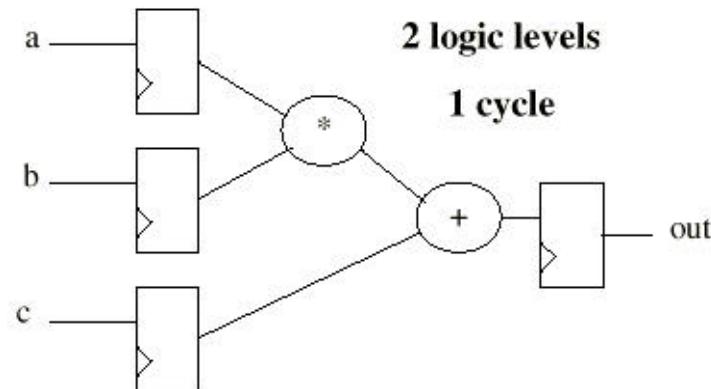


图53 采用流水线之前电路结构

例、采用流水线后的电路

```
process(clk, a, b, c) begin
    if(clk'event and clk = '1') then
        a_temp <= a;
        b_temp <= b;
        c_temp1 <= c;
    end if;
end process;

process(clk, a_temp, b_temp, c_temp1)
begin
    if(clk'event and clk = '1') then
        mult_temp <= a_temp * b_temp
    end if;
end process;
```



```
c_temp2 <= c_temp1;  
end if;  
end process;  
  
process(clk, mult_temp, c_temp2)  
begin  
if(clk'event and clk = '1') then  
    out <= mult_temp + c_temp2;  
end if;  
end process;
```

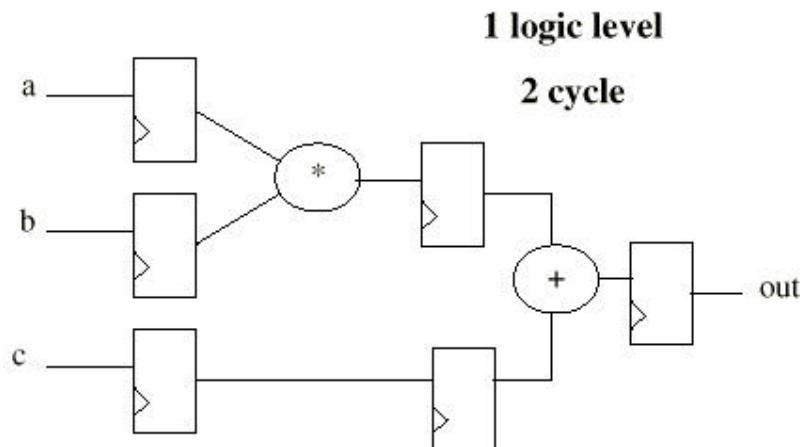


图54 采用流水线之后的电路结构

4.7 组合逻辑和时序逻辑分离

包含寄存器的同步存储电路和异步组合逻辑应分别在独立的进程中完成，组合逻辑中关联性强的信号应放在一个进程中，这样在综合后面积和速度指标较高。这种方法常用在设计Mealy状态机中。Mealy状态机的基本结构如下图所示。

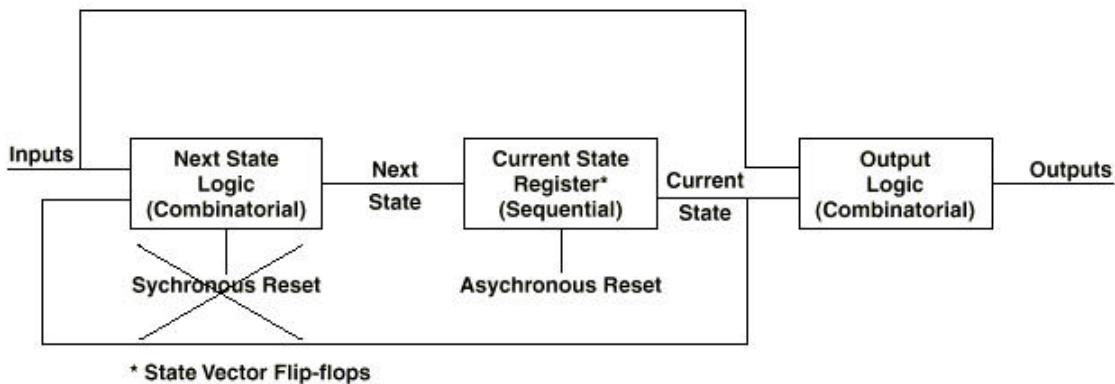


图55 Mealy状态机的基本结构



由图可看出，Mealy状态机由次状态逻辑、当前状态寄存器和输出逻辑三部分组成，其中次状态逻辑和输出逻辑为组合逻辑，当前状态寄存器为时序逻辑。因此，Mealy机可由三个进程实现，如下面VHDL实例。

例、5个状态的Mealy状态机

```
-- Example of a 5-state Mealy FSM
```

```
library ieee;
```

```
use ieee.std_logic_1164.all;
```

```
entity mealy is
```

```
    port (clock, reset: in std_logic;
          data_out: out std_logic;
          data_in: in std_logic_vector (1 downto 0));
end mealy;
```

```
architecture behave of mealy is
```

```
    type state_values is (st0, st1, st2, st3, st4);
```

```
    signal pres_state, next_state: state_values;
```

```
begin
```

```
    -- FSM register
```

```
    statereg: process (clock, reset)
```

```
    begin
```

```
        if (reset = '0') then
```

```
            pres_state <= st0;
```

```
        elsif (clock'event and clock ='1') then
```

```
            pres_state <= next_state;
```

```
        end if;
```

```
    end process statereg;
```

```
    -- FSM combinational block
```

```
    fsm: process (pres_state, data_in)
```

```
    begin
```

```
        case pres_state is
```

```
            when st0 =>
```

```
                case data_in is
```

```
                    when "00" => next_state <= st0;
```

```
                    when "01" => next_state <= st4;
```

```
                    when "10" => next_state <= st1;
```



```
        when "11" => next_state <= st2;
        when others => next_state <= (others <= 'x');

    end case;

when st1 =>

    case data_in is
        when "00" => next_state <= st0;
        when "10" => next_state <= st2;
        when others => next_state <= st1;

    end case;

when st2 =>

    case data_in is
        when "00" => next_state <= st1;
        when "01" => next_state <= st1;
        when "10" => next_state <= st3;
        when "11" => next_state <= st3;
        when others => next_state <= (others <= 'x');

    end case;

when st3 =>

    case data_in is
        when "01" => next_state <= st4;
        when "11" => next_state <= st4;
        when others => next_state <= st3;

    end case;

when st4 =>

    case data_in is
        when "11" => next_state <= st4;
        when others => next_state <= st0;

    end case;

when others => next_state <= st0;

end case;

end process fsm;

-- Mealy output definition using pres_state w/ data_in
outputs: process (pres_state, data_in)
begin
    case pres_state is
        when st0 =>
```



```
case data_in is
    when "00" => data_out <= '0';
    when others => data_out <= '1';
end case;
when st1 => data_out <= '0';
when st2 =>
    case data_in is
        when "00" => data_out <= '0';
        when "01" => data_out <= '0';
        when others => data_out <= '1';
    end case;
when st3 => data_out <= '1';
when st4 =>
    case data_in is
        when "10" => data_out <= '1';
        when "11" => data_out <= '1';
        when others => data_out <= '0';
    end case;
when others => data_out <= '0';
end case;
end process outputs;
end behave;
```

4.8 利用电路的等价性，巧妙地“分配”延时

在功能等价的情况下，我们可以根据时序需要，安排组合逻辑电路在寄存器前后的位置，合理分配延时。

例、组合逻辑在寄存器之后

如下图所示，假定a、b信号的延时非常大，则

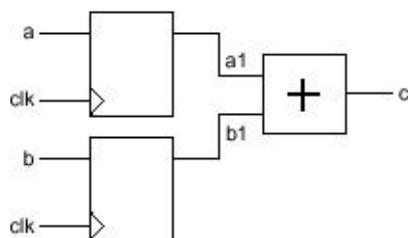


图56 组合逻辑（加法器）在后

例、组合逻辑放在寄存器之前

如果a、b信号的延时并不大，而寄存器c信号经过的逻辑比较多，延时大，则

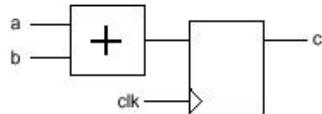


图57 组合逻辑（加法器）在前

这种处理方法的实质是：将关键路径中的部分延时“挪到”其它非关键路径上。

4.9 复制电路，减少扇出（fanout），提高设计速度

提高关键路径速度的一个常用方法是复制电路，减少关键路径的扇出。当一个信号网络所带负载增加时，其路径延时也相应增加。这对复位信号网络可能影响不大，但对象三态使能信号是不能容忍的。扇出对关键路径延时的影响甚至超过了逻辑的级延时，确保一个网络的扇出少于一定值（例如16，表示某个信号所驱动的基本器件不超过16个）是很重要的。下面的VHDL实例中，信号“Tri_en”的扇出为24。

例、寄存器扇出为24

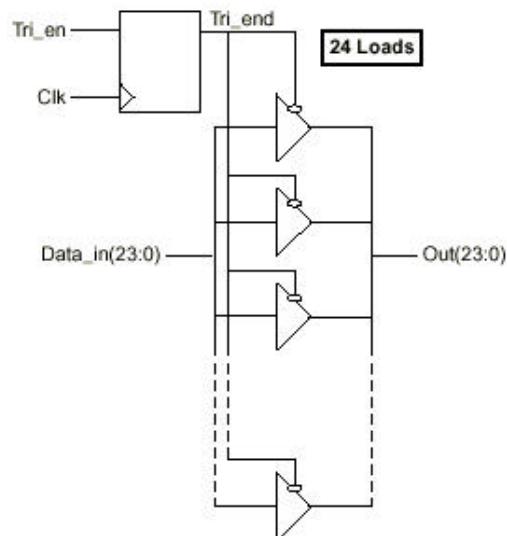


图58 扇出较大

为了将扇出降低一半，增加一个寄存器使负载分成两部分，每个寄存器扇出为12。

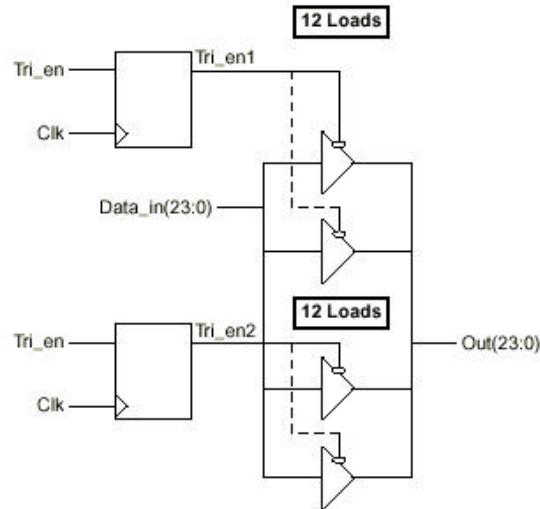


图59 扇出较小

类似地，我们还可以复制组合逻辑电路、网络上插入buffer等手段，减少扇出，提高速度。

4.10 多路选择器与三态电路

4.10.1 virtex以前的系列

多路选择器一般用CLB中的函数发生器实现。一个4选1（6输入信号，4个数据输入信号，2个选择信号）的多路选择器可以在XC4000或Spartan的单个CLB中实现，因此效率较高。6个输入信号使用函数发生器的F、G和H。但是，如果多路复用器规模大于4选1，则将超过一个CLB的容量。例如，16选1的多路复用器需要有2级逻辑，占用了5个CLB，结果增加了面积和延时。因此，Xilinx推荐使用内部的三态buffer（BUFT）实现规模较大的多路复用器。用BUFT实现规模较大的多路复用器具有下面一些优点：

- 1、在输入宽度变化时，面积和延时特性几乎不变。
- 2、多路选择器的最大输入宽度与目标器件的每条水平长线的三态buffer数目相等。
- 3、BUFT在CLB之外，是专门用于三态电路的，不占CLB资源，不用白不用。
- 4、选择信号采用one-hot编码。

一般的多路选择器：

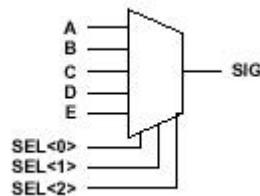


图60 多路选择

用BUFT实现的多路选择如下：

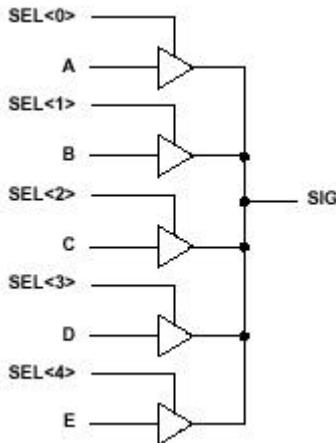


图61 采用三态电路实现电路选择

注意：xilinx的三态电路是低电平导通，高电平三态。若设计中采用高电平导通，低电平三态，则相应的三态控制信号会增加一个反向器，有可能会增加一级CLB（例如三态控制信号直接来自寄存器输出），导致不必要的增加电路延时。因此，设计中尽量采用低电平导通、高电平三态方式。

4.10.2 virtex系列

在virtex系列里，有专门的MUX资源，通过它可在CLB中实现较大的多路选择器。例如，对VirtexII而言，由于一个CLB由4个slice组成，因此一个16选1的多路选择器只需一个CLB，32选1的多路选择器只需2个CLB（通过MUXF8）。

在设计设计中，应当根据周边的实际情况选择恰当的多路选择器实现方式。

4.11 利用LUT四输入特点，指导电路设计

前面我们提到，如果想速度更快，则应当努力减少路径上LUT的个数，而不是逻辑级数；如果想面积更小，则应当努力减少LUT的个数，而不是逻辑门数。如何更好地利用LUT四输入特点进行电路优化，是提高电路性能的关键。

具体内容，参见上一章“补充说明”相关部分。

4.12 高效利用IOB

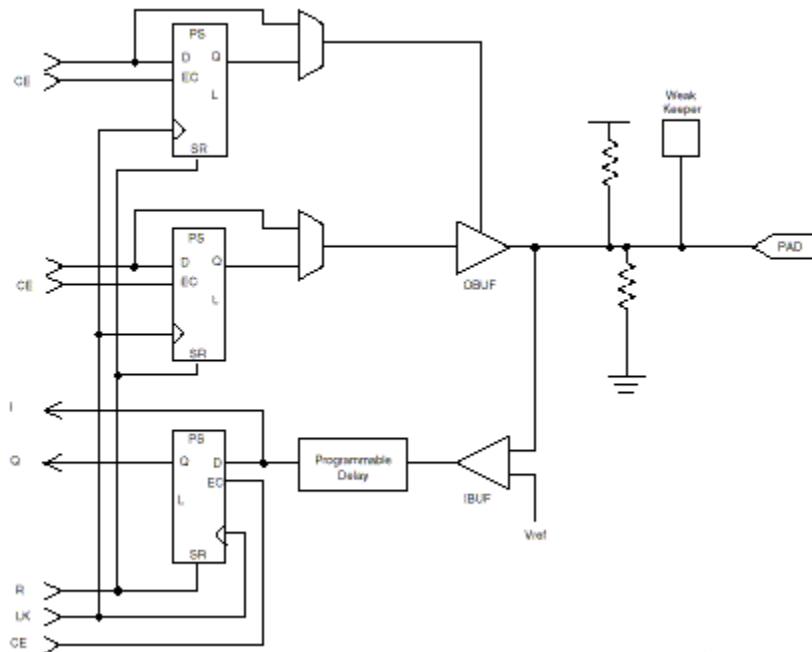
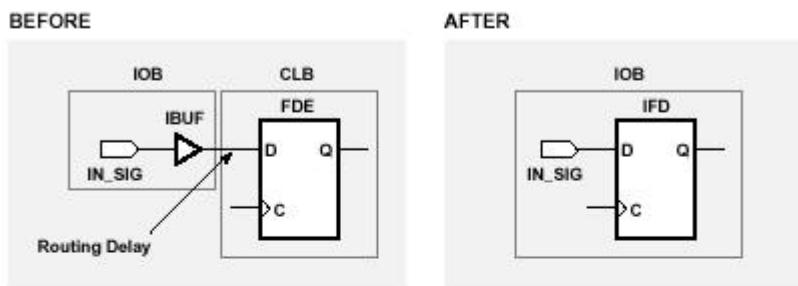


图62 VirtexE IOB结构示意图

IOB中包含有输入寄存器（或锁存器）、输出寄存器和输出三态控制寄存器。在设计中，应当尽量利用IOB中的寄存器（可在布局布线工具中进行设置），以减少CLB的使用量，如下图所示。

Input Register



Output Register

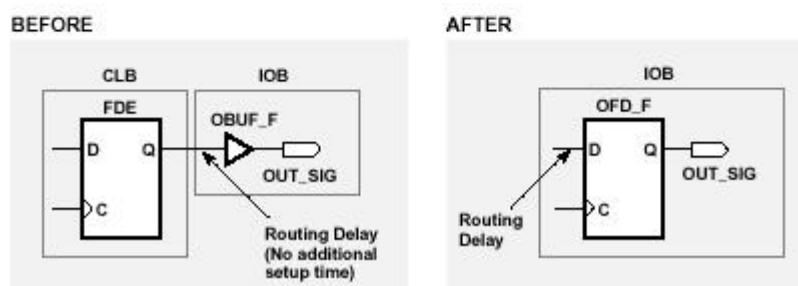


图63 输入输出寄存器移入IOB中

在设计中，我们要求：进入FPGA内部的信号，尽量经过Register；从FPGA出来的信号，也尽量经过Register。这是因为：



1. IOB本身就有register，不用白不用。
2. FPGA内部线延时难以控制，如果不经过Register，将增加Input Delay和Output Delay，对外部器件速度的要求更加苛刻，时序容限缩小。

另外，为了满足外部保持时间要求，在外部引脚和IOB输入触发器（或锁存器）的D输入端之间有一个延时块。如果不需要，可以通过NODELAY设置（可通过UCF）去掉输入寄存器的延时块。

通过综合工具可以控制I/O管脚的边沿速率（Slew rate）。不管I/O配置成输入、输出还是双向，都能在IOB中获得上拉和下拉电阻。缺省时，所有未用的IOB都配置为接上拉电阻的输入信号。

为了充分合理利用IOB资源，我们建议：

1. 所有的输入输出信号都经过寄存器处理。这么做能够放宽对外部电路或其它芯片的时序要求，提高单板设计速度。并且，IOB中的寄存器不用白不用。
2. 在处理双向口时，为了将三态控制寄存器、输入输出寄存器、三态电路都移入IOB中，减少CLB的使用量，应当：
 - (1) 采用同样的时钟触发；
 - (2) 同样的复位电路；
 - (3) 每个输出信号都有独立的三态控制信号（一个输出信号对应一个三态控制寄存器），并且三态控制信号直接来自寄存器。三态控制信号不许共用；
 - (4) 三态控制低电平有效，否则三态寄存器无法引进IOB，增加信号输出延时。

注意：上述方式只适合FPGA设计，对ASIC设计不适用。

另外，xilinx器件有专门的时钟IOB，内设时钟驱动Buffer（IBUFG），有关时钟IOB，可参考“Virtex-II时钟资源”。

4.13 Distributed RAM的使用

一般地，我们将Distributed RAM用作同步RAM。

虽然，Virtex系列器件里有许多Block RAM可供设计者使用，但是，其资源毕竟是有限的，许多情况下我们需要应用Distributed RAM来补充Block RAM的不足。

由于Distributed RAM是采用LUT实现的，当RAM容量大到一定程度时，例如64X32的同步RAM，此时每个地址线驱动的基本单元都比较多，电路扇出很大，并且实现RAM的LUT比较分散，线延时也会增加许多，因此我们建议：

- (1) 地址线直接来自专门地址寄存器
- (2) RAM的输出直接接寄存器

这样，保证地址线的扇出较小，且按照流水线设计，可获得较高频率。

下图是一个采用Distributed RAM实现多路（多通道）加1计数器，其中的地址就是通道号。这种电路经常用在：设计的通道比较多（例如256），每个通道需要一个地址指针（例如8bits加1计数器）。如果采用Block RAM，则十分浪费。

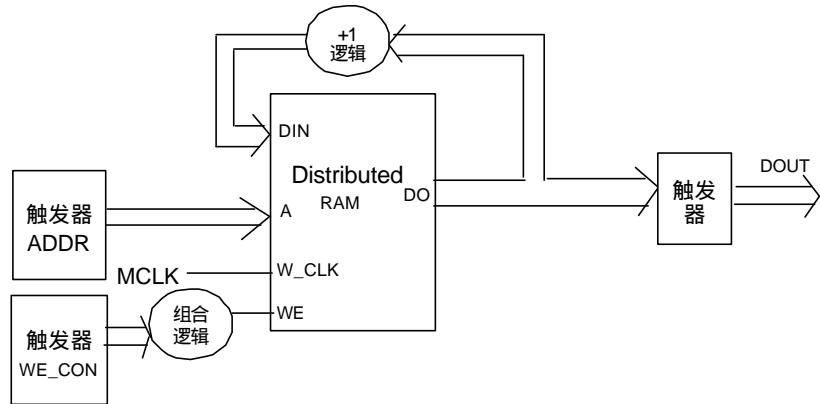


图64 采用Distributed RAM实现多路加1计数器

当然，是否在Distributed RAM前后加寄存器，要具体情况具体分析。因此，我们要求设计者在设计之前，必须了解所采用的RAM最快速率是多少，以决定是否加和如何加寄存器。

注意：如果能够采用RAM来做计数器，会大大降低资源占用率。例如：一个LUT可以替代16个registers。

4.14 Block SelectRAM的使用

Block SelectRAM本身是一个双端口RAM，如果只用作单端口RAM，则至少有一半的容量被浪费掉。因此，许多设计者在使用Block SelectRAM时，都采用双端口形式，哪怕采用单端口也能实现。反正不用白不用。

如果设计仅仅用在FPGA设计，上述做法无可厚非；如果FPGA最终要转变成ASIC，则这种做法不好。应当是，能采用单端口的尽量采用单端口方式，理由是：

1. 如果采用双端口RAM，则转ASIC时，RAM容量会增加一倍，增加ASIC芯片成本。
2. RAM的增加，可能会导致ASIC成品率下降。
3. 如果FPGA设计采用双端口，转ASIC时再变成单端口，则处理麻烦，转ASIC出错概率增大。

另外，在使用内部RAM做设计时，还有一个问题：不管RAM大小，一律采用Block SelectRAM。前面我们提到，BlockRAM是有限的，这种做法很容易导致：Block SelectRAM不足，而LUT等资源利用率很低，但依然不得不更换到容量更大的器件，增加物料成本。

因此，对一些容量较小的RAM，应当依据实际情况来决定是采用Block SelectRAM还是Distributed RAM。

4.15 SRL的使用

前面我们提到，一个LUT可把16个移位寄存器放在一个LUT中实现，大大节省线延时和面积，提高设计速度和设计性能，具体情况可以参见前面的有关章节。

4.16 LFSR加1计数器

通常的加1计数器的计算值（输出序列）是“0，1，2，3，……”有规律地变化。而LFSR计数器基于线性“异或”或者“同或（异或非）”反馈，以牺牲输出序列的顺序性为代价，具有结构简单，速度快，占用资源少的特点，可用作对序列顺序要求不严格の場合，如FIFO，随机码发生器，任意分频计数器等。

值得注意的是，LFSR采用了移位寄存器的结构。因此，我们在实现LFSR时，可采用SRL结构，如下图所示的15bit LFSR计数器。当然，由于SRL的特殊性，如：16个寄存器只能有一个输出信号，不能带复位信号等，限制了SRL的应用。

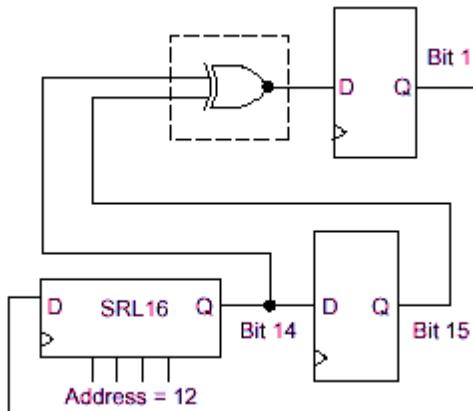


图65 15位基本型LFSR计数器在VIRTEX器件中的实现

有关LFSR计数器的详细资料，可参见“LFSR计数器原理及应用”或者“lfsr.pdf”。

5 如何使用后端工具

本章节主要说明Xilinx的一些后端工具能为我们做什么，在什么情况下我们考虑使用这些工具。至于这些工具具体如何使用，可以看Xilinx提供的相关文件（可从www.xilinx.com下载，也可和本人联系）。

5.1 布局布线

布局布线是干什么的，我不用多说。我只想讲讲几个值得注意的问题

5.1.1 设计前期（设计方案阶段），对关键电路的处理

一个设计能否成功，关键是在设计方案阶段相关问题是否考虑完善。其中，一个非常重要的工作是确定关键路径（或者关键模块、关键算法等）能否在芯片中实现，其实现的结果如何（如速度是否满足？面积是否太大等）。

因此，我们要求大家在做方案时，要对所有的可能的关键路径（或关键部分）心中有数，而且一定要在正式开始编代码之前，要将这一部分的“评估代码”完成，并经过布局布线的检验，以考察其可实现性，从而对设计方案的风险有一个确定的认识。

5.1.2 布局布线策略，兼谈如何做第一次布局布线

一个设计想要成功完成布局布线，必需满足：



1. 设计规模在芯片容量限制之内
2. 线资源等资源不能超过芯片现有资源
3. 时序要满足要求。

据我观察，在进行布局布线时，绝大部分设计者做法是：设定时钟约束及必要的管脚约束，选定器件，然后开始布局布线。这种做法在小规模设计时，一般可以过。不过，当面对大型设计时，却往往行不通，浪费设计时间。在这种情况下，比较好的做法是：

1. 第一次布线时，不加任何约束，或者放松时钟约束。

这么做的目的是：确保我所选的器件是符合容量要求的；并快速提供一个参考结果。目前，我们经常遇到布线几十个小时，仍然没有结果的情况。因为没有结果，我们也就无法利用相关的工具进行分析，只能干等；同时，也不知道这么长时间没结果，是布线布不通呢，还是资源不够用。结果，白白浪费时间。

2. 在第一次布线结果分析基础上，适当增加约束条件，如时钟约束。注意，时钟约束要根据上一次分析结果确定是否一步到位。

约束条件如果设置不好，十分浪费时间，尤其是对规模庞大的设计。因此，我们的约束要恰当好处，即能发现关键路径（不满足约束条件），又能较快的布出一个结果来。

3. 如果布线结果仍不满意，则应当努力找到尽可能多的“放松”约束（TIG，Multi-Cycle-Path等），同时，根据实际情况，决定是否要进行设计修改。可以尝试“设计技巧”章节里提到的各种手段。

需要注意的是：约束文件（如ucf）只是一种微调作用，若要从根本上解决延时问题，应多从设计本身考虑。当时序实现与要求差别很大时，是不能依靠约束来解决问题。

5.1.3 正确看待map之后的资源占用报告

在使用Virtex系列进行FPGA设计时，经常发现map报告，说资源利用率已经到达100%。然而，真实情况确实如此吗？未必！

我们知道，slice内部包含LUT、Registers和快速进位链及其它快速性能电路。其中，对资源占用影响最深的应当是LUT和Registers。因此，我们在看报告时，应当看LUT占用了多少，Registers占用了多少，当然，也应当看Block RAM占用多少，时钟资源占用多少。这些东西，才是我们下决策时要考虑的因素。

因此，我们在估计一个设计是否能被某个器件装下时，不能笼统地只看Slice使用状况。

5.2 FPGA Editor的作用

一定要看“版图”，了解我们的电路是如何实现的，如资源是否共享、特殊结构（如进位链等）是否利用等。

请注意：我们现在得出的经验只是基于现有的器件和设计手段，将来情况发生变化时，本文提到的一些所谓技巧可能要失灵，效果反而会更糟。因此，通过FPGA Editor查看电路实现形式，以确保“这就是我所想要的”，同时还可检验以前的经验是否还管用。

一个比较好的习惯：在正式编写代码之前，对于自己不能确定的东西，先写一个“评估代码”，经综合、布局布线，再通过FPGA Editor看一看该电路是否就是自己想要的。



在设计前期，FPGA Editor对关键路径论证非常有帮助。

5.3 FloorPlanner的作用

FloorPlanner主要是用来人工place设计各部分在芯片中的位置，最终形成位置约束（ucf或mfp格式）文件，以指导Place and Route。它即可一个module一个module的place，也可一个LUT一个LUT的place；即可将设计全部place，也可部分place。

通常情况下，我们可以用它来手工place关键路径，以压缩线延时。偶尔，可以尝试手工place一个关键模块或者整个设计，不过，会经常遇到情况反而变得更糟糕的现象。

5.4 TimingAnalyzer的作用

TimingAnalyzer是一个静态时序分析工具。实际上，在布局布线过程中，布局布线工具已经根据约束文件进行时序分析工作了，并将一些基本的分析结果保存下来，供用户使用。

不过，布局布线工具只能根据约束文件进行时序分析，不够灵活。当我们想看一看其它路径时序情况，或者了解更多的未满足约束条件的其它路径信息时，可通过TimingAnalyzer进行。

由于布局布线工具的时序分析报告内容比较简单，当我们想了解更多的信息时，例如分析到底有哪些路径不满足某一约束条件，这些路径的特点是什么等，这时用TimingAnalyzer就比较方便。

在设计前期对关键电路进行预分析，以及设计后期对关键路径分析时，经常会用到TimingAnalyzer。

6 综合运用

下面提到的解决问题技巧，由于受目前认识的限制和现有的技术水准，不应当把它们看成是“万能”的，尤其是随着技术的发展，其中的许多手段肯定会不合适了。

6.1 可能成为关键路径的电路

在做详细设计方案或者总体方案的时候，一定要考虑到设计中哪些电路可能成为关键路径，如果不考虑这些，很可能导致设计实现失败，或者要更改设计，或者要更换器件。

在FPGA设计里，常常影响到设计无法（或者很难）实现的电路有（指设计速度方面）：比较器、多路选择器、Distributed RAM、乘法器、加法器等，尤其是在位宽比较大的情况下。

建议：在做方案时，针对上述电路先进行速度评估，以决定是否要Pipeline。

6.2 如何提高芯片速度

提高芯片速度的根本方法是：减少组合逻辑LUT的级数；尽量压缩关键路径上的线延时。

6.2.1 引入放松约束：TIG（False path）和Multi-Cycle-Path

许多设计者觉得设置TIG和Multi-Cycle-Path意义不大，因为它们不直接对关键路径发生作用。



这种想法是错误的，虽然它们不直接对关键路径发生作用，但可以起到“让非关键路径散开”的作用，让这些非关键路径滚得越远越好。这样，就“为关键路径腾挪出空间”，从而与关键路径相关的LUT有可能尽量压缩在一起，从而到达“压缩关键路径上线延时”的目的。这实际是一种“曲线救国”的策略（用词可能不太好，不过我想不出更好的了）。

实践证明：这种方法非常行之有效，而且它的一个最大好处是不用更改设计。

6.2.2 对线延时比较大的net，设置Maxdelay和Maxskew

迫使工具利用驱动能力比较强的长线资源，以减少线延时。

6.2.3 采用BUFGS

对于一些扇出特别多、线延时特别大的net，可以直接引用BUFGS，以提高驱动能力。例如，对时钟使能信号，采用BUFGS进行驱动。

6.2.4 基本设计技巧

在“设计技巧”章节里，我们提到了许多设计技巧，其中许多与设计速度相关，现在整理如下：

copy逻辑电路减少fanout；加法器处理；case代替if语句；合并if语句；减少关键路径上的LUT级数；去掉资源共享；pipeline；组合逻辑与时序逻辑分离；利用电路的等价性，巧妙分配延时；利用LUT四输入特点进行优化

6.2.5 专有资源的利用

如进位链、MUX、SRL、乘法器等，可利用Coregen产生宏单元。

利用专有资源虽然可以提高速度，但有一缺点：降低代码的可移植性；如果是准备转ASIC，则需对专有资源进行代码改动，增加出错的可能性。因此，在做ASIC设计时，采用这种方法要仔细权衡。

6.2.6 关键路径在同一个Module。

这样，在综合时，可以或得最佳效果。

6.2.7 关键路径单独综合，不与其它模块放在一起综合

对关键路径所在模块，采取速度优先策略；对非关键路径模块，采用面积优先策略。

6.2.8 针对关键路径，进行位置约束

如果发现关键路径相关LUT距离太远，可通过floorplanner手工布线，并形成位置约束文件，以指导布局布线。

6.2.9 迂回策略：降低非关键路径上的面积，为关键路径腾挪空间。

尽可能优化非关键路径上的面积，以尽量多给关键路径留空间，以便将关键路径相关LUT压缩在一起，降低线延时。

该方法体现了“向非关键路径要面积，向关键路径要时间”的设计思想。

6.3 如何降低芯片面积



6.3.1 Distributed RAM代替BlockRAM

在设计中，如果LUT足够多，而BLOCK RAM不够，则可考虑采用Distributed RAM代替BlockRAM。

这种情况在设计中有时会碰到。

6.3.2 Distributed RAM代替通道计数器

这个在前面的章节已经提到，这里不多说。

6.3.3 专有资源的利用

情况同“6.2.5”。

6.3.4 基本设计技巧

在“设计技巧”章节里，我们提到了许多设计技巧，其中许多与设计面积相关，现在整理如下：

加法器处理；if代替case语句；资源共享；组合逻辑与时序逻辑分离；利用LUT四输入特点进行优化；高效利用IOB；采用单端口BlockRAM，为ASIC做准备

另外，在代码风格方面，请参考“2.3 综合性能对Coding Style 影响”相关内容。

7 感谢

本文在制订过程中，陈亮、谭锐等人提供了许多好的建议，并指出其中的不足与错误，这里特别表示感谢。

希望广大读者勇于发表不同意见，共同提高设计水平。