

HDA 工 | 程 | 技 | 术 | 丛 | 书 |

国内首本系统论述Xilinx新一代集成设计环境Vivado和Vivado HLS设计流程与设计方法的中文著作  
Xilinx公司Vivado设计套件高级市场营销总监Ramine Roane作序

Xilinx大学计划推荐用书



THE DEFINITIVE GUIDE OF XILINX FPGA DEVELOPMENT  
BASED ON VIVADO INTEGRATED DEVELOPMENT ENVIRONMENT



# Xilinx FPGA设计权威指南

## Vivado集成设计环境

何宾 编著  
He Bin



清华大学出版社



THE DEFINITIVE GUIDE OF XILINX FPGA DEVELOPMENT  
BASED ON VIVADO INTEGRATED DEVELOPMENT ENVIRONMENT

# Xilinx FPGA设计权威指南

Vivado集成设计环境

Vivado是全球著名的可编程逻辑器件厂商Xilinx推出的新一代集成开发环境，其设计理念与其前身ISE相比有着显著的进步：更加强调以IP为中心的系统级设计思想；允许设计者在多个方案中探索最优的实现方法；提供了更高效的时序收敛能力；提供设计者对FPGA布局布线高效的控制能力等。另外，高级综合工具Vivado HLS也是Vivado集成设计环境的一大亮点，使得设计者可以使用高级编程语言对FPGA设计进行建模，并通过高级综合工具HLS将设计模型自动转换成RTL级的描述。

本书从逻辑设计、嵌入式系统设计、数字信号处理等不同的应用角度，通过最典型的设计实例，由浅入深地论述了Vivado的设计理念和设计方法；使读者通过对Vivado设计工具的系统学习，掌握其设计思想精髓，提高FPGA设计效率。

## 主要内容

- Vivado设计导论
- Vivado调试流程
- Vivado HLS信号处理流程
- Vivado部分可重配置设计流程
- Vivado工程模式和非工程模式设计流程
- 基于IP的嵌入式系统设计流程
- System Generator设计流程
- Vivado高级设计技术

## 学习资源



为了方便教学和自学，本书配套提供所有案例的完整设计文件。这些资源可以在清华大学出版社网站([www.tup.com.cn](http://www.tup.com.cn))本书页面下载。

## 作者简介

**何宾** 长期从事数字系统EDA方面教学与科研工作，在EDA教学与科研方面积累了丰富的经验。同时，与Xilinx、Cypress、Altium、MathWorks等知名建立了良好的合作关系，极力推动最新EDA设计技术在国内高校和业界的普及。目前，已出版《EDA原理及Verilog实现》、《EDA原理及VHDL实现》、《Xilinx FPGA设计权威指南》、《Xilinx All Programmable Zynq-7000 SoC设计指南》、《Altium Designer 13.0电路设计、仿真与验证权威指南》等20余部广受好评的EDA技术图书。

清华大学出版社数字出版网站

**WQBook** 书文  
www.wqbook.com



**HDA** 工 | 程 | 技 | 术 | 丛 | 书 |



THE DEFINITIVE GUIDE OF XILINX FPGA DEVELOPMENT  
BASED ON VIVADO INTEGRATED DEVELOPMENT ENVIRONMENT

# Xilinx FPGA设计权威指南

## Vivado集成设计环境

何宾 编著  
He Bin

清华大学出版社

## 内 容 简 介

本书全面系统地介绍了 Xilinx 新一代集成开发环境 Vivado 的设计方法、设计流程和具体实现。全书共分 8 章，内容包括：Vivado 设计导论、Vivado 工程模式和非工程模式设计流程、Vivado 调试流程、基于 IP 的嵌入式系统设计流程、Vivado HLS 设计流程、System Generator 设计流程、Vivado 部分可重配置设计流程和 Vivado 高级设计技术。本书参考了 Xilinx 公司提供的 Vivado 最新设计资料，理论与应用并重，将 Xilinx 公司最新的设计方法贯穿在具体的设计实现中。

本书可作为使用 Xilinx Vivado 集成开发环境进行 FPGA 设计的工程技术人员的参考用书，也可作为电子信息类专业高年级本科生和研究生的教学用书，同时也可作为 Xilinx 公司的培训教材。

本书封面贴有清华大学出版社防伪标签，无标签者不得销售。

版权所有，侵权必究。侵权举报电话：010-62782989 13701121933

### 图书在版编目(CIP)数据

Xilinx FPGA 设计权威指南：Vivado 集成设计环境 / 何宾编著. —北京：清华大学出版社，2014  
EDA 工程技术丛书  
ISBN 978-7-302-36688-1

I. ①X… II. ①何… III. ①可编程序逻辑器件—系统设计—指南 IV. ①TP332.1-62

中国版本图书馆 CIP 数据核字(2014)第 117171 号

责任编辑：盛东亮

封面设计：李召霞

责任校对：焦丽丽

责任印制：沈 露

出版发行：清华大学出版社

网 址：<http://www.tup.com.cn>, <http://www.wqbook.com>

地 址：北京清华大学学研大厦 A 座 邮 编：100084

社 总 机：010-62770175 邮 购：010-62786544

投稿与读者服务：010-62776969, c-service@tup.tsinghua.edu.cn

质 量 反 馈：010-62772015, zhiliang@tup.tsinghua.edu.cn

课 件 下 载：<http://www.tup.com.cn>, 010-62795954

印 装 者：北京密云胶印厂

经 销：全国新华书店

开 本：185mm×260mm 印 张：23.25 字 数：536 千字

版 次：2014 年 7 月第 1 版 印 次：2014 年 7 月第 1 次印刷

印 数：1~3000

定 价：69.00 元

---

产品编号：056522-01

# 序言

在今天日益复杂的电子系统中，高级算法正在拉伸密度、性能和功耗的边界。在逻辑、时钟和 IP 中持续扩展的复杂度，伴随着微缩处理节点（shrinking process nodes）上进行互连成为关键的瓶颈，这些因素成为设计团队在所分配的有限的预算范围内实现其最终目标的挑战。为了解决互连的瓶颈和加速生产力，Xilinx 正在发布领先一代的硅片结构，以及拥有尖端分析布线器的行业首个 ASIC 增强型 Vivado 设计套件。

然而，即使拥有了最好的硅片和工具，设计团队也必须采用融合工业中最佳设计实践的规范化设计方法。因此，为了更进一步增强 Vivado 设计套件，并使其能够加速和可预测设计周期，Xilinx 还推出了可编程业界内首个丰富的设计方法——UltraFast 设计方法。该方法由来自工业界专家精选的最佳实践所组成，而且该方法和一套综合的方法指南、第三方工具、IP 核和自学培训视频一起，部署在 Vivado 设计套件内建的一个自动化设计组合之中。

UltraFast 设计方法已经获得广泛的证明，可以将设计周期由数月减少到几周。了解更多 Xilinx UltraFast 设计信息，请访问 [www.xilinx.com/ultrafast](http://www.xilinx.com/ultrafast)。

衷心祝贺何宾教授出版行业首本 Vivado 中文图书，并为其在 Vivado 设计套件及 UltraFast 设计方法在中国工程界的推广和应用所做出的努力表示衷心的感谢，也期待着 Vivado 能够为中国工程界的设计创新带来强大的动力！

Ramine Roane  
Xilinx 公司 Vivado 设计套件高级市场营销总监  
2014 年 4 月

# 前言

全球知名的可编程逻辑器件生产厂商——美国 Xilinx 公司——于 2012 年发布了新一代的 Vivado 集成开发环境,使得新一代 FPGA 的设计环境和设计方法发生了重要变化。在 2014 年初,Xilinx 新一代 UltraScale 结构的 FPGA 也进入量产阶段。这些都标志着在高性能数据处理方面,FPGA 将发挥越来越重要的作用。同时,我们也很高兴看到 2014 年 Xilinx 公司迎来自己 30 岁的生日。在未来若干年内,Xilinx 将为全球信息技术的不断发展做出自己的贡献。

Xilinx 新一代集成开发环境 Vivado 突出基于知识产权(Intellectual Property, IP)核的设计方法,更加体现系统级设计的思想,进一步增强了设计者对 FPGA 底层布局和布线的干预能力。并且,允许设计者通过选择不同的设计策略,对不同的实现方法进行探索,从中找到最佳的实现解决方案。这些新的设计思想和设计方法,大大地降低了设计成本,同时也提高了 FPGA 的设计效率。

本书首次系统地以 Xilinx 公司新一代集成开发环境 Vivado 为平台,从逻辑设计、嵌入式系统设计和信号处理等几个方面,充分展现 Vivado 集成开发环境的特点和性能。全书共分为 8 章,内容包括:Vivado 设计导论、Vivado 工程模式和非工程模式设计流程、Vivado 调试流程、基于 IP 的嵌入式系统设计流程、Vivado HLS 设计流程、System Generator 设计流程、Vivado 部分可重配置设计流程、Vivado 高级设计技术。每章内容要点如下:

- (1) 第 1 章主要介绍了 Vivado 系统级设计流程、Vivado 功能和特性、Vivado 使用模式和最新的 UltraScale 结构。
- (2) 第 2 章主要介绍了工程模式设计流程和非工程模式设计流程。
- (3) 第 3 章主要介绍了设计调试原理和方法、创建新的 FIFO 调试工程、添加 FIFO IP 到设计中、添加顶层设计文件、FIFO 例化、添加约束文件、网表插入调试探测流程方法及实现、使用添加 HDL 属性调试探测流程、使用 HDL 例化调试核调试探测流程。
- (4) 第 4 章主要介绍了简单硬件系统设计、在 PL 内添加外设、创建和添加定制 IP、编写软件程序、软件控制定时器和调试、使用硬件分析仪调试。
- (5) 第 5 章主要介绍了高级综合工具概述、高级综合工具调度和绑定、Vivado HLS 工具的优势、C 代码的关键属性、时钟测量术语说明、HLS 关键优化策略、基于 HLS 的数字系统实现。
- (6) 第 6 章主要介绍了使用 System Generator 实现 FPGA 信号处理的方法、FPGA 模型设计模块、System Generator 运行环境的配置、信号模型的构建和实现、编译 MATLAB 到 FPGA、FIR 滤波器的设计与实现。
- (7) 第 7 章主要介绍了可重配置导论和可重配置的实现。
- (8) 第 8 章主要介绍了 Vivado 支持的属性、增量编译、修改布线和逻辑、布局约束、查看和分析时序报告,以及时序约束。

参加本书编写工作的人员还有李宝隆和张艳辉。李宝隆负责第3章和第6章设计实例的验证,张艳辉负责第7章设计实例的验证。全书由何宾统稿和定稿。

本书的编写得到Xilinx公司大中华区大学计划经理谢凯年博士和Xilinx公司亚太区市场传播经理张俊伟女士的大力支持和帮助,以及美国Digilent公司的大力支持和帮助,他们为本书的编写提供了设计资料和硬件设计平台。此外,Mathworks公司中国教育业务发展总监陈炜博士为该书的编写捐赠了正版的MATLAB R2013a软件,在此也向Mathworks公司表示感谢。正是由于他们的无私帮助和鼎力支持,才能使作者顺利完成本书的编写工作。同时,也要感谢清华大学出版社的编辑和相关工作人员,他们的辛勤工作保证了本书的高质量出版。

由于FPGA技术发展迅速,作者水平有限,书中难免会有疏漏之处,欢迎读者批评指正。

作 者

2014年5月于北京

# 目录

<b>第 1 章 Vivado 设计导论 .....</b>	<b>1</b>
1.1 Vivado 系统级设计流程 .....	1
1.2 Vivado 功能和特性 .....	3
1.3 Vivado 使用模式 .....	4
1.3.1 Vivado 工程模式和非工程模式不同点比较 .....	5
1.3.2 工程模式和非工程模式命令的不同 .....	5
1.4 最新的 UltraScale 结构 .....	7
1.4.1 可配置逻辑块 .....	7
1.4.2 时钟资源和时钟管理单元 .....	9
1.4.3 块存储器资源 .....	13
1.4.4 专用的 DSP 模块 .....	15
1.4.5 输入/输出块 .....	16
1.4.6 高速串行收发器 .....	17
1.4.7 PCI-E 模块 .....	18
1.4.8 Interlaken 集成块 .....	18
1.4.9 Ethernet 模块 .....	19
1.4.10 系统监控器模块 .....	19
1.4.11 配置模块 .....	19
1.4.12 互连资源 .....	20
<b>第 2 章 Vivado 工程模式和非工程模式设计流程 .....</b>	<b>21</b>
2.1 工程模式设计流程 .....	21
2.1.1 启动 Vivado 集成开发环境 .....	21
2.1.2 建立新的设计工程 .....	22
2.1.3 Vivado 设计主界面及功能 .....	26
2.1.4 创建并添加一个新的设计文件 .....	30
2.1.5 RTL 描述和分析 .....	34
2.1.6 设计综合和分析 .....	36
2.1.7 设计行为级仿真 .....	41
2.1.8 添加约束条件 .....	45
2.1.9 XDC 约束语法规则 .....	49
2.1.10 设计实现和分析 .....	50
2.1.11 设计时序仿真 .....	55
2.1.12 生成编程文件 .....	56

# 目录

2.1.13 下载比特流文件到 FPGA .....	57
<b>2.2 非工程模式设计流程 .....</b>	<b>61</b>
2.2.1 修改路径 .....	61
2.2.2 设置输出路径 .....	62
2.2.3 设置设计源文件和约束 .....	62
2.2.4 运行综合 .....	63
2.2.5 运行布局 .....	63
2.2.6 运行布线 .....	64
2.2.7 生成比特流文件 .....	64
<b>第3章 Vivado 调试流程 .....</b>	<b>65</b>
3.1 设计调试原理和方法 .....	65
3.2 创建新的 FIFO 调试工程 .....	66
3.3 添加 FIFO IP 到设计中 .....	67
3.4 添加顶层设计文件 .....	70
3.5 例化 FIFO .....	71
3.6 添加约束文件 .....	75
3.7 网表插入调试探测流程方法及实现 .....	77
3.7.1 网表插入调试探测流程的方法 .....	77
3.7.2 网表插入调试探测流程的实现 .....	79
3.8 使用添加 HDL 属性调试探测流程 .....	83
3.9 使用 HDL 例化调试核调试探测流程 .....	84
<b>第4章 基于 IP 的嵌入式系统设计流程 .....</b>	<b>89</b>
4.1 简单硬件系统设计 .....	89
4.1.1 创建新的工程 .....	90
4.1.2 使用 IP 集成器创建处理器系统 .....	91
4.1.3 生成顶层 HDL 和导出设计到 SDK .....	96
4.1.4 创建存储器测试程序 .....	98
4.1.5 验证设计 .....	100
4.2 在 PL 内添加外设 .....	102
4.2.1 打开工程 .....	102
4.2.2 添加两个 GPIO 实例 .....	102
4.2.3 连接外部 GPIO 外设 .....	108
4.2.4 生成比特流和导出到 SDK .....	112

# 目录

4.2.5	生成测试程序	112
4.2.6	验证设计	116
4.3	创建和添加定制 IP	116
4.3.1	使用外设模板创建定制 IP	116
4.3.2	使用 IP 封装器封装外设	122
4.3.3	修改工程设置	124
4.3.4	添加定制 IP 到设计	125
4.3.5	添加约束 XDC	128
4.3.6	添加 BRAM	129
4.4	编写软件程序	130
4.4.1	打开工程	130
4.4.2	创建应用工程	132
4.4.3	为 LED_IP 分配驱动	136
4.4.4	分析汇编目标文件	138
4.4.5	验证设计	138
4.5	软件控制定时器和调试	140
4.5.1	打开工程	140
4.5.2	创建 SDK 软件工程	140
4.5.3	在硬件上验证操作	143
4.5.4	启动调试器	144
4.6	使用硬件分析仪调试	146
4.6.1	ILA 核原理	147
4.6.2	VIO 核原理	150
4.6.3	打开工程	151
4.6.4	添加定制 IP	152
4.6.5	添加 ILA 和 VIO 核	152
4.6.6	标记和分配调试网络	154
4.6.7	生成比特流文件	156
4.6.8	生成测试程序	156
4.6.9	测试和调试	157
<b>第 5 章</b>	<b>Vivado HLS 设计流程</b>	<b>163</b>
5.1	高级综合工具概述	163
5.1.1	高级综合工具的功能和特点	163
5.1.2	不同的命令对 HLS 综合结果的影响	164

# 目录

5.1.3	从 C 模型中提取硬件结构	166
5.2	高级综合工具调度和绑定	168
5.2.1	高级综合工具调度	168
5.2.2	高级综合工具绑定	168
5.3	Vivado HLS 工具的优势	169
5.4	C 代码的关键属性	170
5.4.1	函数	171
5.4.2	类型	171
5.4.3	循环	178
5.4.4	数组	179
5.4.5	端口	180
5.4.6	操作符	181
5.5	时钟测量术语说明	182
5.6	HLS 关键优化策略	183
5.6.1	延迟和吞吐量	183
5.6.2	循环的处理	190
5.6.3	数组的处理	193
5.6.4	函数内联	198
5.6.5	命令和编译指示	200
5.7	基于 HLS 的数字系统实现	202
5.7.1	基于 HLS 实现组合逻辑	202
5.7.2	基于 HLS 实现时序逻辑	217
5.7.3	基于 HLS 实现矩阵相乘	223
<b>第 6 章</b>	<b>System Generator 设计流程</b>	<b>242</b>
6.1	FPGA 信号处理方法	242
6.2	FPGA 模型设计模块	244
6.2.1	Xilinx Blockset	244
6.2.2	Xilinx Reference Blockset	244
6.3	System Generator 运行环境的配置	245
6.4	信号模型的构建和实现	245
6.4.1	信号模型的构建	245
6.4.2	模型参数的设置	249
6.4.3	信号处理模型的仿真	252
6.4.4	生成模型子系统	253

# 目录

6.4.5	模型 HDL 代码的生成	254
6.4.6	打开生成设计文件并仿真	255
6.4.7	协同仿真的配置及实现	256
6.4.8	生成 IP 核	259
6.5	编译 MATLAB 到 FPGA	260
6.5.1	模型的设计原理	260
6.5.2	系统模型的建立	262
6.5.3	系统模型的仿真	264
6.6	FIR 滤波器的设计与实现	265
6.6.1	FIR 滤波器设计原理	265
6.6.2	生成 FIR 滤波器系数	266
6.6.3	建模 FIR 滤波器模型	267
6.6.4	仿真 FIR 滤波器模型	270
6.6.5	修改 FIR 滤波器模型	272
6.6.6	仿真修改后 FIR 滤波器模型	273
<b>第 7 章</b>	<b>Vivado 部分可重配置设计流程</b>	<b>274</b>
7.1	可重配置导论	274
7.1.1	可重配置的概念	274
7.1.2	可重配置的应用	275
7.1.3	可重配置的特点	278
7.1.4	可重配置术语解释	280
7.1.5	可重配置的要求	281
7.1.6	可重配置的标准	281
7.1.7	可重配置的流程	283
7.2	可重配置的实现	283
7.2.1	查看脚本	283
7.2.2	综合设计	284
7.2.3	实现第一个配置	285
7.2.4	实现第二个配置	290
7.2.5	验证配置	291
7.2.6	生成比特流	292
7.2.7	部分重配置 FPGA	293

# 目录

<b>第 8 章 Vivado 高级设计技术</b>	<b>295</b>
8.1 Vivado 支持的属性	295
8.1.1 ASYNC_REG	295
8.1.2 BLACK_BOX	295
8.1.3 BUFFER_TYPE	296
8.1.4 DONT_TOUCH	296
8.1.5 FSM_ENCODING	297
8.1.6 FSM_SAFE_STATE	298
8.1.7 FULL_CASE(Verilog Only)	298
8.1.8 GATED_CLOCK	298
8.1.9 IOB	299
8.1.10 KEEP	299
8.1.11 KEEP_HIERARCHY	300
8.1.12 MAX_FANOUT	300
8.1.13 PARALLEL_CASE(Verilog Only)	301
8.1.14 RAM_STYLE	301
8.1.15 ROM_STYLE	301
8.1.16 SHREG_EXTRACT	301
8.1.17 SRL_STYLE	302
8.1.18 TRANSLATE_OFF/TRANSLATE_ON	302
8.1.19 USE_DSP48	302
8.1.20 在 XDC 文件中使用属性	303
8.2 增量编译	303
8.2.1 增量编译流程	303
8.2.2 运行增量布局和布线	304
8.2.3 使用增量编译	305
8.2.4 增量编译高级分析	307
8.3 修改布线和逻辑	308
8.3.1 修改布线	308
8.3.2 修改逻辑	314
8.4 布局约束	316
8.5 查看和分析时序报告	317
8.5.1 时序检查基础	317
8.5.2 生成时序报告	323
8.5.3 分析时序报告	327

# 目 录

8.6 时序约束 .....	331
8.6.1 时钟定义 .....	331
8.6.2 时钟组 .....	336
8.6.3 I/O 延迟约束 .....	339
8.6.4 时序例外 .....	341
8.6.5 时序约束实现 .....	353
附录 XDC 中有效的命令 .....	355

自从 Xilinx 公司推出 ISE 集成开发环境以来的 16 年间, Xilinx 可编程逻辑器件应用工程师都是在这个熟悉的集成开发环境下完成设计的。Xilinx 公司于 2012 年发布了新一代的 Vivado 设计套件, 设计环境和设计方法发生了重要的变化。

本章对 Vivado 系统级设计流程、Vivado 功能和特性、Vivado 使用模式以及最新的 UltraScale 结构进行了概述, 以帮助读者从整体上正确把握 Vivado 的设计理念和设计方法, 从而在 Vivado 集成开发环境下进行高效率的设计。

## 1.1 Vivado 系统级设计流程

图 1.1 给出了 Vivado 系统级设计流程。除了传统上寄存器传输级(Register Transfer Level, RTL)到比特流的 FPGA 设计流程外, Vivado 设计套件新提供了系统级的设计集成流程, 该系统级设计的中心思想是基于知识产权(Intellectual Property, IP)核的设计。

从图 1.1 中可以看出:

(1) Vivado 设计套件提供了一个环境, 该环境用于配置、实现、验证和集成 IP。

(2) 通过 Vivado 提供的 IP 目录, 就可以快速地对 Xilinx IP、第三方 IP 和用户 IP 进行例化和配置。IP 的范围包括: 逻辑、嵌入式处理器、数字信号处理(Digital Signal Processing, DSP)模块或者基于 C 的 DSP 算法设计。一方面, 将用户 IP 进行封装, 并且使封装的 IP 符合 IP-XACT 协议。这样, 就可以在 Vivado IP 目录中使用它; 另一方面, Xilinx IP 利用 AXI4 互连标准, 从而实现更快速的系统级集成。在设计中, 设计者可以通过 RTL 或者网表格式使用这些已经存在的 IP。

(3) 可以在设计流程的任意一个阶段, 对设计进行分析和验证。

(4) 对设计进行分析, 包括: 逻辑仿真、I/O 和时钟规划、功耗分析、时序分析、设计规则检查(Design Rule Check, DRC)、设计逻辑的可视化、实现结果的分析和修改以及编程和调试。

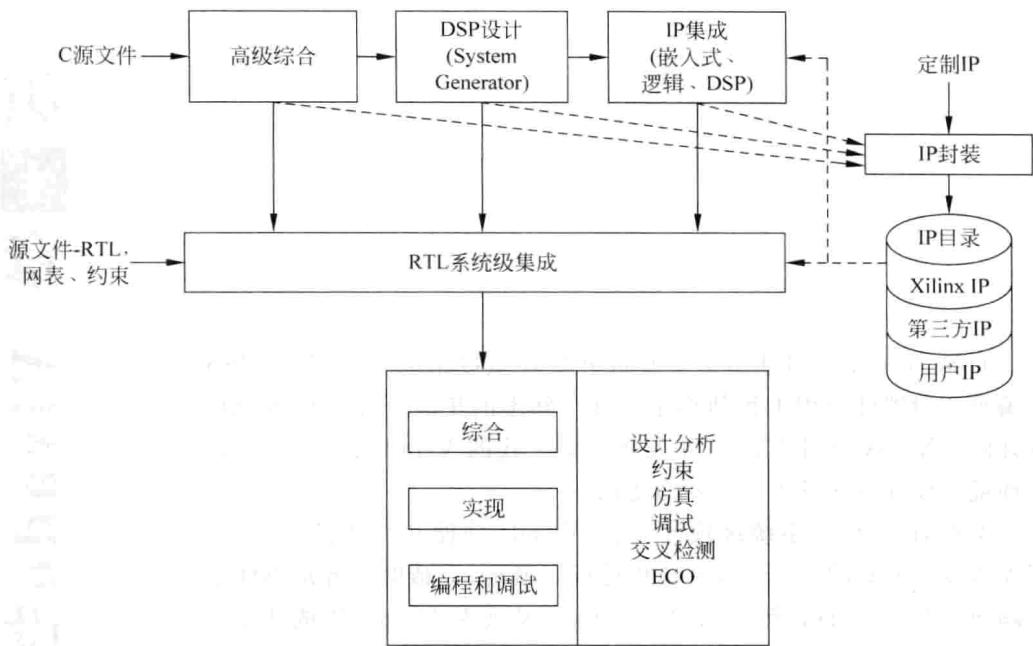


图 1.1 Vivado 系统级设计流程

(5) 通过 AMBA AXI4 互连协议, Vivado IP 集成器环境使得设计者能够将不同的 IP 组合在一起。设计者可以使用块图风格的接口交互式地配置和连接 IP, 并且可以像原理图那样, 通过绘制 DRC 助手很容易将整个接口连接在一起。然后, 对这些 IP 块设计进行封装, 将其当作单个的设计源。通过在一个设计工程或者在多个工程之间进行共享, 来使用设计块。

(6) Vivado IP 集成器环境是主要的接口, 通过使用 Zynq 器件或者 Microblaze 处理器, 创建嵌入式处理器设计。Vivado 设计套件也集成了传统的 XPS, 用于创建、配置和管理 MicroBlaze 微处理器软核。在 Vivado IDE 环境中, 集成和管理这些核。如果设计者选择编辑 XPS 的源设计, 将自动启动 XPS 工具。设计者也可以将 XPS 作为一个单独的工具运行, 然后将最终的输出文件作为 Vivado IDE 环境下的源文件。在 Vivado IDE 环境中, XPS 不能用于 Zynq 器件, 而是使用新的 IP 集成器环境。

(7) 对于数字信号处理方面的应用, Vivado 提供了两种设计方法:

① 使用 Xilinx System Generator 建模数字信号处理: Vivado 设计套件集成了 Xilinx System Generator 工具, 用于实现 DSP 的功能。当设计者编辑一个 DSP 源设计时, 自动启动 System Generator。设计者可以将 System Generator 作为一个独立运行的工具, 并且将其最终的输出文件作为 Vivado IDE 的源文件。

② 使用高级综合工具(High-Level Synthesis, HLS)建模数字信号处理: Vivado 设计套件集成了 Vivado HLS, 它提供了基于 C 语言的 DSP 功能, 来自 Vivado HLS 的 RTL 输出, 作为 Vivado IDE 的 RTL 源文件。在 Vivado IP 封装器中, 将 RTL 的输出封装成符合 IP-XACT 标准的 IP。这样, 在 Vivado IP 目录中就变成了可用的 IP。设计者

也可以在 System Generator 逻辑中使用 Vivado HLS 逻辑模块。

(8) Vivado 设计套件中包含 Vivado 综合、Vivado 实现、Vivado 时序分析、Vivado 功耗分析和比特流生成。通过下面的一种方式：

- ① Vivado IDE；
- ② 批处理 Tcl 脚本；
- ③ Vivado 设计套件的 Tcl Shell；
- ④ Vivado IDE Tcl 控制台下，输入 Tcl 命令。

设计者就可以运行整个的设计流程。

(9) 设计者可以创建多个运行，通过使用不同的综合选项、实现选项、时序约束、物理约束和设计配置来进行尝试。这样，可以帮助设计者改善设计结果，提高设计效率。

(10) Vivado 集成开发环境提供了 I/O 引脚规划环境，用于将 I/O 端口分配到指定的封装引脚上，或者分配到内部晶圆的焊盘上。通过使用 Vivado 引脚规划器内的视图和表格，设计者可以分析器件和设计相关的 I/O 数据。

Vivado IDE 提供了高级的布局规划能力，用于帮助改善实现的结果。设计者可以将一个指定的逻辑，强迫放到芯片内的某个特定区域。即：为了后面的运行，通过交互的方式，锁定到指定的位置或者布线。

(11) Vivado IDE 使设计者可以在对设计处理的每个阶段，对设计进行分析、验证和修改。通过对处理过程中所生成的中间结果进行分析，设计者可以提高设计的性能。在将设计转换成 RTL 后、综合后和实现后，就可以运行分析工具。

(12) Vivado 集成了 Vivado 仿真器，使得设计者可以在设计的每个阶段，运行行为级和结构级的逻辑仿真。仿真器支持 Verilog 和 VHDL 混合模式仿真，并且以波形的形式显示结果。此外，设计者也可以使用第三方的仿真器。

(13) 在 Vivado IDE 内，在对设计处理的每一个阶段，设计者都可以对结果进行交互分析。一些设计和分析特性包括：时序分析、功耗估计和分析、器件利用率统计、DRC、I/O 规划、布局规划和交互布局，以及布线分析。

(14) 当执行实现过程后，对器件进行编程。然后，在 Vivado 环境中对设计进行分析。在 RTL 内或者在综合之后，很容易地识别调试信号。在 RTL 或者综合网表中，插入和配置调试核。Vivado 逻辑分析仪也可以进行硬件验证。通过将接口设计成与 Vivado 仿真器一致，就可以使两者共享波形视图。

## 1.2 Vivado 功能和特性

Vivado Design Suite 提供全新构建的 SoC 增强型、以 IP 和系统为中心的下一代开发环境，以解决系统级集成和实现的生产力瓶颈。

与 Xilinx 前一代的设计平台 ISE 相比，Vivado 在各方面的性能都有了明显的提升。表 1.1 给出了 Vivado 在性能方面的提升。

注：随着 Vivado 环境的不断优化，其性能还会提高。

表 1.1 Vivado Design Suite 加速设计生产力

加速实现	加速集成和验证
(1) 实现速度提升 4 倍; (2) 器件利用率提升了 20%; (3) 最多 3 个速度级性能优势; (4) 功耗降低 35%; (5) 增量编译速度提高一倍	(1) IP 集成速度提高 4 倍; (2) RTL 仿真速度提高 3 倍; (3) C/C++/SystemC 至 RTL 的转换速度提高 4 倍

注：随着软件不断更新，性能还会不断改善。

Vivado 集成设计环境支持下面业界已经建立的设计标准：

- (1) Tcl;
- (2) AXI4, IP-XACT;
- (3) Synopsys 设计约束(Synopsys Design Constraints, SDC);
- (4) Verilog, VHDL, SystemVerilog;
- (5) SystemC, C, C++。

通过这些支持，使得：

- (1) 电子设计自动化(Electronic Design Automation, EDA)生态系统更好地支持 Vivado 设计套件。此外，Vivado 设计套件中集成了很多新的第三方工具。
- (2) 本质上，Vivado 设计套件是基于 Tcl 的脚本。因此支持 Synopsys 设计约束 (SDC) 和 Xilinx 设计约束(Xilinx Design Constraints, XDC) 格式。
- (3) 除了提供传统上对 Verilog、VHDL 和 SystemVerilog 综合的支持，Vivado 高级综合 HLS 也可以使用 C、C++ 或者 SystemC 语言定义逻辑。
- (4) 使用标准的 IP 互连协议，例如 AXI4 和 IP-XACT。这样，使得更快和更容易地实现系统级设计集成。

### 1.3 Vivado 使用模式

Vivado 设计套件允许设计者根据自己的习惯，使用不同的方法运行工具。设计者可以使用基于工程的方法自动管理设计过程和设计数据，也就是所说的工程模式(Project Mode)。当在工程模式下时，在磁盘上创建一个目录结构，用于管理设计源文件，运行结果和跟踪工程状态。通过一个运行结构，来管理自动地综合和实现过程，以及跟踪运行状态。可以通过单击鼠标，在 Vivado 集成环境内运行完整的设计流程。

另一种方法是，设计者可以选择基于 Tcl 脚本的编译风格方式。通过这种方式，设计者自己可以管理源文件和设计流程。这种方式也称为非工程模式(Non-Project Mode)。当在非工程模式下时，通过源文件当前所在的位置来访问源文件，以及通过存储器中的流程来编译设计。

- ① 通过使用 Tcl 命令，可以单独运行设计中的每一步；
- ② 使用 Tcl 命令，可以设置设计参数和实现选项；
- ③ 使用 Tcl 命令，设计者可以在设计处理的每个阶段，保存设计检查点和创建报告；

④ 此外,在每个设计阶段,设计者可以打开 Vivado 集成设计环境,用于设计分析和分配约束。

当设计者正在查看存储器中活动的设计时,会自动地提交流程中的变化。比如:设计者可以保存对新约束文件的更新或者设计检查点。

### 1.3.1 Vivado 工程模式和非工程模式不同点比较

在工程模式下,Vivado 集成设计环境跟踪设计历史,保存相关的设计信息。然而,在这种模式下,由于很多是过程自动处理的,所以设计者很少能控制处理的过程。例如在每次运行时,只是生成一组标准的报告文件。在工程模式下,提供了下面的自动处理功能:

- (1) 源文件管理和状态;
- (2) 通过 Vivado IP 目录和 Vivado 集成器,实现 IP 配置和集成;
- (3) 综合信息和自动生成标准的报告;
- (4) 保存和重用工具设置和设计配置;
- (5) 用多个综合和实现运行,进行探索;
- (6) 约束设置的使用和管理;
- (7) 运行结果的管理和状态;
- (8) 流程导航;
- (9) 工程总结。

在非工程模式下,通过使用 Tcl 命令,执行每个行为。在存储器中,执行所有的处理。因此,不会自动生成文件或者报告。当设计者每次对设计进行编译的时候,设计者必须定义所有的源文件,设置所有工具和设计配置的参数,启动所有的实现命令,以及指定所需要生成的报告文件。由于没有在磁盘上创建一个工程,源文件保留在它们最初的位置,只在设计者指定的位置创建运行输出。这个流程充分发挥 Tcl 命令的能力,可以充分地控制整个设计过程。

表 1.2 给出了工程模式和非工程模式特性的比较。

表 1.2 工程模式和非工程模式特性比较

流程元素	工程模式	非工程模式
设计源文件管理	自动	手动
流程导航	引导	手动
流程定制	有限	无限
报告	自动	手动
分析阶段	只有设计	设计和设计检查点

### 1.3.2 工程模式和非工程模式命令的不同

对于设计者来说,选择的模式不同,Tcl 命令就相应地有所不同。在非工程模式下,

所有的操作和工具设置都要求单独的 Tcl 命令,包括:设置工具选项,运行实现命令,产生报告和写设计检查点。在工程模式下,打包过的命令,用于每个综合、实现和报告命令。

例如在工程模式下,设计者使用 add\_files Tcl 命令将源文件添加到工程中。可以将源文件复制到工程中,这样在工程目录结构中保留一个独立的版本;或者通过远程方式引用。在非工程模式下,设计者使用 read\_verilog、read\_vhdl、read\_xdc 和 read... Tcl 命令,从当前的位置读取不同类型的源文件。

在工程模式下,带有预配置运行策略的 launch\_runs 命令,用于启动工具和生成标准报告。这样能够合并实现命令、标准报告、运行策略的使用、运行状态的跟踪。然而,设计者也可以在设计处理的每一步之前或者之后,运行定制的 Tcl 命令。在工程内,自动地保存和管理运行的结果。在非工程模式下,必须单独运行每个命令,例如 opt\_design、place\_design 和 route\_design。

如图 1.2 所示,很多 Tcl 命令既可用于工程模式,又可以用于非工程模式,例如报告命令。在一些情况下,将 Tcl 命令指定在工程模式或者非工程模式。当创建脚本的时候,指定为一个模式的命令,不能被混用。例如,如果使用了工程模式,设计者就不能使用基本级别命令,例如 synth\_design,这是因为将该命令指定为非工程模式。如果在工程模式下,使用了非工程模式的命令,数据库不会更新状态信息,并且不会自动生成报告。

**注:** 工程模式包含所有的 GUI 操作,这样导致在绝大多数情况下,执行一个 Tcl 命令。Tcl 命令显示在 Vivado 集成开发环境的控制台下,在 vivado.jou 文件进行捕获。设计者可以使用这个文件,来开发用于其中一种模式的脚本。

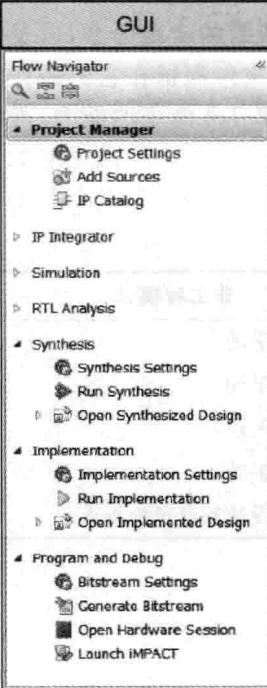
工程模式		非工程模式
GUI	Tcl Script	Tcl Script
	<pre> create_project ... add_files ... import_files ... ... launch_run synth_1 wait_on_run synth_1 open_run synth_1 report_timing_summary  launch_run impl_1 wait_on_run impl_1 open_run impl_1 report_timing_summary  launch_run impl_1 -to_step write_bitstream wait_on_run impl_1 </pre>	<pre> read_verilog ... read_vhdl ... read_ip ... read_xdc ... read_edif ... ... synth_design ... report_timing_summary write_checkpoint  opt_design write_checkpoint place_design write_checkpoint route_design report_timing_summary write_checkpoint  write_bitstream </pre>

图 1.2 工程模式和非工程模式命令

## 1.4 最新的 UltraScale 结构

UltraScale 架构是业界首款采用最先进的 ASIC 架构优化的 All Programmable 架构。该架构能从 20nm 平面 FET 结构扩展至 16nm 鳍式 FET 晶体管技术甚至更高的技术,同时还能从单芯片扩展到 3D IC。借助 Xilinx Vivado 设计套件的分析型协同优化,UltraScale 架构可以提供海量数据的布线功能,同时还能智能地解决先进工艺节点上的头号系统性能瓶颈。这种协同设计可以在不降低性能的前提下实现超过 90% 的利用率。

UltraScale 架构不仅能解决系统总吞吐量扩展和时延方面的局限性,而且还能直接应对先进工艺节点上的头号系统性能瓶颈,即互连问题。UltraScale 新一代互连架构的推出体现了可编程逻辑布线技术的真正突破。赛灵思致力于满足从多吉比特智能包处理到多太比特数据路径等新一代应用需求,即必须支持海量数据流。在实现宽总线逻辑模块(将总线宽度扩展至 512 位、1024 位甚至更高)的过程中,布线或互连拥塞问题一直是影响实现时序收敛和高质量结果的主要制约因素。过于拥堵的逻辑设计通常无法在早期器件架构中进行布线。即使工具能够对拥塞的设计进行布线,最终设计也经常需要在低于预期的时钟速率下运行。而 UltraScale 布线架构则能完全消除布线拥塞问题。结论很简单:只要设计合理,就能进行布线。

下面来做个类比。例如位于市中心的一个繁忙十字路口,交通流量的方向是从北到南,从南到北,从东到西,从西到东,有些车辆正试图掉头,所有交通车辆试图同时移动。这样通常就会造成大堵车。现在考虑一下将这样的十字路口精心设计为现代化高速公路或主干道,情况又会如何?道路设计人员设计出了专用坡道(快行道),用以将交通流量从主要高速路口的一端顺利地疏导至另一端。交通流量可以从高速路的一端全速移动到另一端,不存在堵车现象。

### 1.4.1 可配置逻辑块

可配置的逻辑块(Configurable Logic Block, CLB)是主要的逻辑资源,用于实现时序和组合逻辑电路。UltraScale 结构 CLB 提供了高级高性能和低功耗的可编程逻辑,其特点主要有:

- (1) 真正的 6 输入查找表;
- (2) 双 LUT5(5 输入 LUT)选项;
- (3) 分布式存储器和移位寄存器能力;
- (4) 用于算术功能的专用高速进位逻辑;
- (5) 多功能的多路复用器,用于高效的利用;
- (6) 专用的存储元素,能配置为带有灵活控制信号的触发器或者锁存器。

每个 CLB 连接到一个开关矩阵,用于访问通用的布线资源。一个 CLB 包含一个切片(Slice)。每个切片提供 8 个 6 输入的查找表和 16 个触发器。LUT 按列排列,它带有一个 8 位的进位链。多功能多路复用器将 LUT 组合在一起,构成 7、8 或者 9 输入的任意函数功能,或者带有最多 55 个输入的一些函数功能。SLICEL 用于描述支持上述功能

的 CLB 切片(L 表示逻辑)。在 SLICEM(M 表示存储器)内的 LUT 能配置为一个查找表,64 位的分布式存储器,或者一个 32 位的移位寄存器。表 1.3 给出了一个 CLB Slice 内的逻辑资源。

表 1.3 一个 CLB Slice 内的逻辑资源

CLB Slice	LUT	触发器	算术和进位链	多功能复用器	分布式 RAM	移位寄存器
SLICEL	8	16	1	F7,F8,F9	N/A	N/A
SLICEM	8	16	1	F7,F8,F9	512b	256bit

注: LUT 从底到顶用 A,B,C,D,E,F,G 和 H 标记。

### 1. 多路复用器

每个切片包含 7 个多路复用器,用于构建更宽的函数功能。不同多路复用器的功能包括:

(1) F7MUX\_AB、F7MUX\_CD、F7MUX\_EF 和 F7MUX\_GH

用于组合两个相邻的 LUT,以产生一个 7 输入的任何函数功能。

(2) F8MUX\_BOT 和 F8MUX\_TOP

用于组合两个相邻的 F7MUX,以产生 8 输入的任何函数功能。

(3) F9MUX

用于组合两个 F8MUX,以产生 9 输入的任何函数功能。

### 2. 存储元素

每个 CLB Slice 内有 16 个存储元素,其中的每一个都可以配置为边沿触发的 D 触发器或者电平触发的锁存器。在器件上半部(A~D)和下半部(E~H)提供锁存器选项。置位/复位有同步或者异步两种方式。

UltraScale 结构中,每个 CLB 为存储元素提供了两个时钟输入和两个置位/复位(SR)。图 1.3 给出了时钟和置位/复位的分配结构。

用于存储元素的初始化方式:

(1) SRLOW: 当 SR 信号有效时,同步或者异步复位;

(2) SRHIGH: 当 SR 信号有效时,同步或者异步置位;

(3) INIT0: 当上电时,异步复位;

(4) INIT1: 当上电时,异步置位。

### 3. 分布式 RAM(只有 SLICEM)

SLICEM 内的函数发生器(LUT)可以作为同步 RAM 资源,也称为分布式 RAM。SLICEM 内的多个 LUT 可组合构成最多 512 比特容量的 RAM。它可以配置为单端口、双端口、简单双端口、四端口、八端口模式。

### 4. 只读存储器(ROM)

SLICEM 和 SLICEL 内的每个 LUT 都可以实现一个  $64 \times 1$  位 ROM。它提供了四

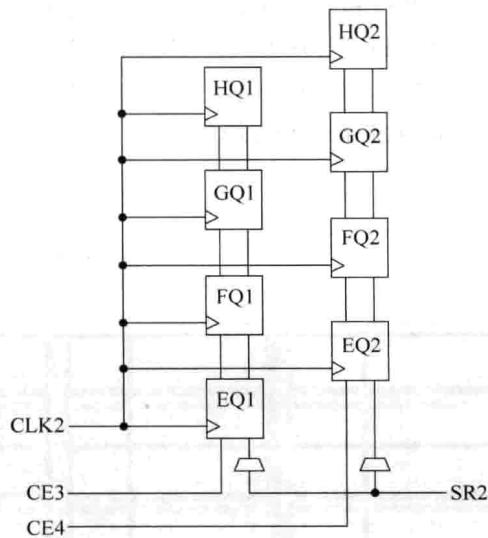


图 1.3 控制信号的分配

种 ROM 配置方式：

- (1)  $64 \times 1$ (1 个 LUT);
- (2)  $128 \times 1$ (2 个 LUT);
- (3)  $256 \times 1$ (4 个 LUT);
- (4)  $512 \times 1$ (8 个 LUT)。

### 5. 移位寄存器(只有 SLICEM)

一个 SLICEM 的函数发生器可以在不使用触发器的情况下，配置为一个 32 位的移位寄存器。当用作移位寄存器时，每个 LUT 可以将串行数据延迟  $1 \sim 32$  个时钟。一个 SLICEM 内的 8 个 LUT 可以级联产生最多 256 个时钟周期的延迟。图 1.4 给出了移位寄存器的描述。

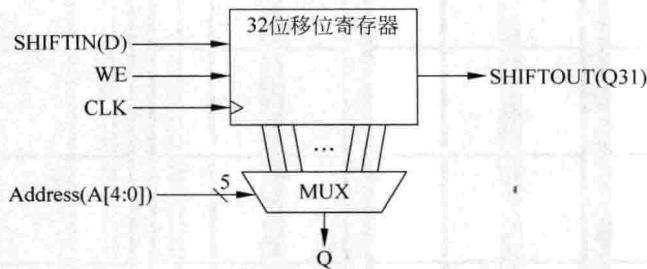


图 1.4 32 位移位寄存器

#### 1.4.2 时钟资源和时钟管理单元

图 1.5 给出了 UltraScale 结构的时钟结构图。从图中可以看出，基本的结构是由

CR 块构成(CR 表示分段时钟的行和列)。CR 以一个单元的方式排列,这样就可以构造出行和列。每个 CR 包含切片(Slice)、DSP、36K BRAM。每个 CR 中的这些资源在行方向上可能不同,但是在垂直方向是一致的。每个 CR 的高度是 60 个 CLB,24 个 DSP 和 12 个 BRAM。这样,在器件内建立了这些资源的列。

从图 1.5 中可以看出:

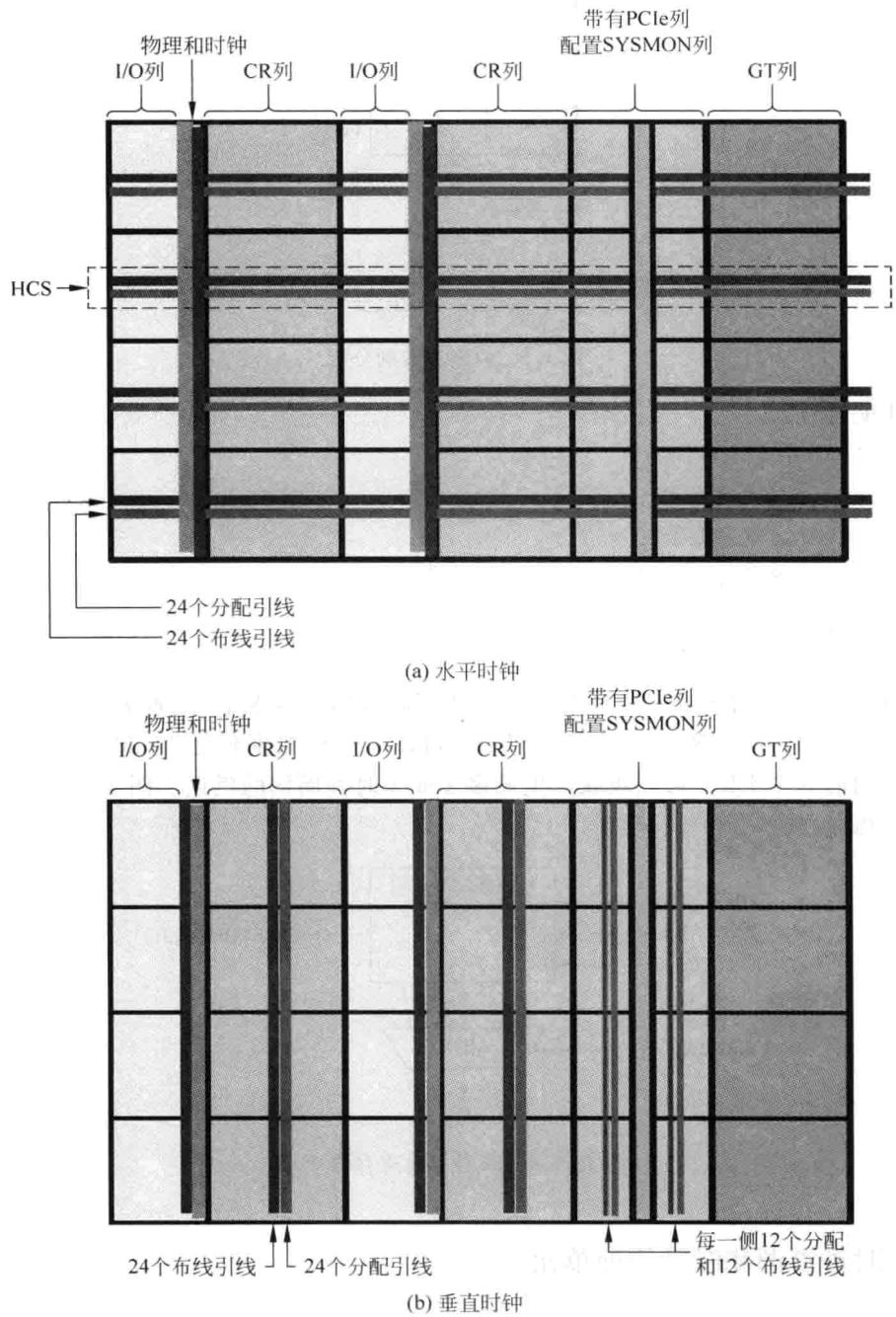


图 1.5 UltraScale 结构的时钟分布

- (1) 在 CR 的中间插入 I/O 和 GT 列。
- (2) 与 I/O 列相邻的是物理块, 带有时钟管理单元 (Clock Management Tiles, CMT)、全局时钟缓冲区、全局时钟复用结构和 I/O 逻辑管理功能。
- (3) 时钟结构存在一个单个的列, 它包含配置逻辑、系统监控器和 PCIE。
- (4) 一个水平时钟脊梁 (Horizontal Clock Spine, HCS) 穿过每行 CR、I/O 和 GT 的中间。HCS 包含水平布线和分布引线, 以及叶时钟缓冲区, 还有在水平/垂直布线和分配之间的时钟网络互连。
- (5) 垂直的布线引线和分配连接一列内所有的 CR, 而垂直布线跨越一个整个的 I/O 列。时钟结构包含 24 个水平布线和 24 个分布引线, 以及 24 个垂直布线和 24 个分布引线。
- (6) I/O 直接由来自物理块的时钟驱动, 或者通过布线引线由相邻的物理块驱动。

### 1. UltraScale 结构的时钟资源

UltraScale 结构内的时钟资源包括: 全局时钟输入、字节时钟输入、时钟缓冲和布线。

#### 1) 全局时钟输入

在每一个 I/O Bank 上有四个全局时钟 (Global Clock, GC) 引脚, 可以直接访问全局时钟缓冲区、混合模式的时钟管理器 (Mixed-mode Clock Manager, MMCM) 和相位锁相环 (Phase Lock Loop, PLL)。全局时钟 (Globle Clock, GC) 输入提供了高速访问全局和区域时钟资源的专用通道。

每个 I/O Bank 在一个单个时钟域内, 包含 52 个 I/O 引脚。

#### 2) 字节时钟输入

字节通道时钟 (DBC 和 QBC) 输入引脚对是专用的时钟输入, 直接驱动源同步的时钟到 I/O 块的比特切片。在存储器应用中, 这些信息称为双向数据选通 (Bidirectional Data Strobe, BDS)。

#### 3) 时钟缓冲和布线

物理全局时钟包含 BUFGCTRL、BUFGCE 和 BUFGCE\_DIV。它们中的每一个时钟缓冲, 都可以由来自相邻组 MMCM、同一物理块的 PLL 和互连直接驱动。时钟缓冲区驱动整个芯片内的布线和分配资源。每个物理块包含 24 个 BUFGCE、8 个 BUFGCTRL 和 4 个 BUFGCE\_DIV。但是, 在同一时刻, 只使用其中的 24 个缓冲区。

此外, 还包含 BUFCE\_LEAF 时钟缓冲区、BUFG 时钟缓冲区和 BUFG\_GT 时钟缓冲区。其中:

- (1) BUFG\_GT: 该时钟缓冲区提供对时钟 1~8 的分频, 该分频时钟用于收发器时钟;
- (2) BUFCE\_LEAF: 在叶子级别上, 该时钟缓冲区提供了时钟的门控能力。

### 2. UltraScale 时钟管理模块

UltraScale 结构内的每个 I/O 组包含一个时钟管理单元 (Clock Management Tile, CMT), 每个 CMT 包含下面的功能单元:

- (1) 一个混合模式的时钟管理器(MMCM)；
- (2) 两个相位锁相环(PLL)。其目的主要用于为 I/O 生成时钟。但是，它也包含了用于内部结构的 MMCM 的一些功能集。

图 1.6 给出了 MMCM 的结构图。MMCM 用于宽范围频率的合成、内部或者外部时钟的抖动过滤器。MMCM 的中心是一个压控振荡器(Voltage Controlled Oscillator, VCO)，根据来自相位频率检测器(Phase Frequency Detector, PFD)的电压，VCO 增加或者降低频率。

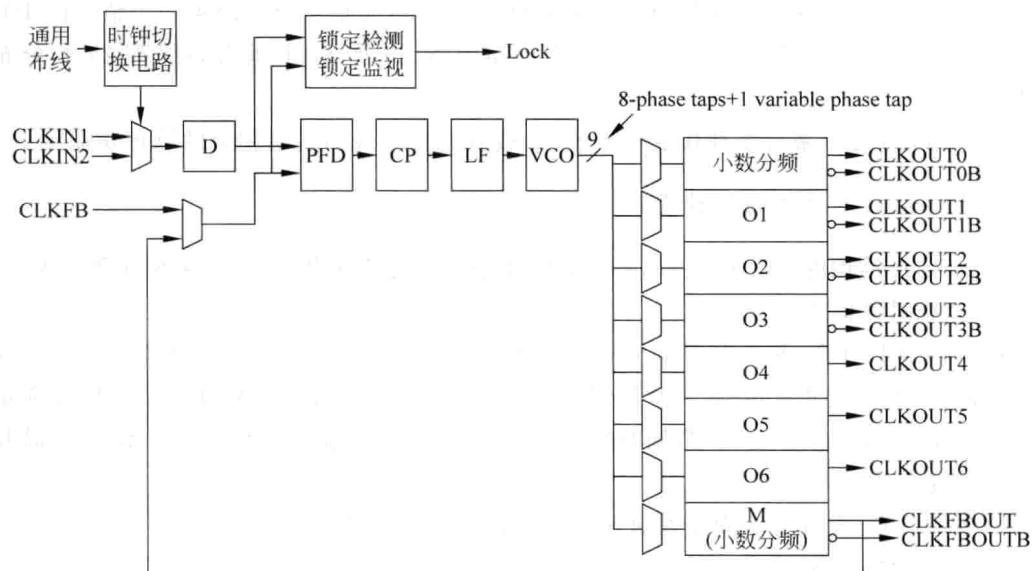


图 1.6 MMCM 的内部结构

MMCM 中有三个可编程的分频因子 D、M 和 O，通过动态配置端口，就可以在配置和正常操作期间，对它进行编程。预触发器 D，用于降低输入时钟的频率，然后将其送入到相位/频率比较器。反馈分频器 M，由于在将它送给相位比较器之前，其将 VCO 的输出频率进行分频，所以充当乘法器的角色。必须选择合适的 D 和 M，使得 VCO 处于其指定的频率范围内。VCO 有 8 个等间隔的输出相位( $0^\circ$ 、 $45^\circ$ 、 $90^\circ$ 、 $135^\circ$ 、 $180^\circ$ 、 $225^\circ$ 、 $270^\circ$ 和  $315^\circ$ )。设计者可以选择其中的一个相位，来驱动一个输出分频器。此外，设计者可以在配置时，对每个分频器进行编程，分频因子是 1~128 的任意整数。

MMCM 有三个输入抖动过滤器选项——低带宽、高带宽或者优化的模式。其中：

- (1) 低带宽模式有最好的抖动衰减；
- (2) 高带宽模式有最好的相位偏置；
- (3) 优化模式允许 Vivado 工具找到最好的设置。

MMCM 也有一个小数计数器，它在反馈路径上(充当乘法器)或者在输出路径上。小数计数器允许非整数的 1/8 增量。因此，将频率合成能力增加 8 倍。取决于 VCO 的频率，MMCM 也能够提供小幅度增量的固定或者动态的相位移动。在 1600MHz 时，相位移动时间增量为 11.2ps。

PLL 比 MMCM 的特性要少得多,在一个时钟管理单元内的两个 PLL,其基本上是为专用的存储器接口电路提供必要的时钟。在 PLL 中心的电路类似于 MMCM,带有 PFD、VCO,以及可编程的 M、D 和 O 计数器。每个 PLL 有两个到 FPGA 结构的分频输出时钟,以及到存储器接口电路的一个时钟和上一个使能信号。

### 1.4.3 块存储器资源

每个 UltraScale 结构的 FPGA 包含大量的 36Kb 的块存储器 (Block RAM, BRAM),每个 BRAM 都有完全独立的端口,共享保存的数据。每个 BRAM 可以配置成一个 36Kb RAM 或者两个独立的 18Kb RAM。由时钟控制每个存储器的访问(读或者写)。连接每个 BRAM 列的使能信号,可以在垂直相邻的 BRAM 之间实现级联。这样,提供了一种非常容易的方法,用于创建大容量、快速的存储阵列和 FIFO,大大地降低了功耗。

所有的输入、数据、地址、时钟使能和写使能都被寄存。输入地址总是由时钟驱动(除非关闭地址锁存),一直保留数据到下一个操作。在以消耗一个额外时钟延迟的代价下,一个可选的输出数据流水线寄存器允许更高的时钟速率。在写操作期间,数据的输出能反映出前面保存的数据或者新写入的数据,或者其保持不变。在设计者的设计中包含没有使用的 BARM 时,会自动断电,以降低功耗。每个 BARM 中有一个额外的引脚,用于控制动态功耗控制特性。

#### 1. 可编程的数据宽度

每个端口可以配置成  $32K \times 1$ 、 $16K \times 2$ 、 $8K \times 4$ 、 $4K \times 9$ (或 8)、 $2K \times 18$ (或 16)、 $1K \times 36$ (或 32)或者  $512 \times 72$ (或者 64)。不管配置成一个 RAM 块还是 FIFO,两个端口都可以有不同宽高比,并且没有任何限制。只有在简单双端口模式下,其数据宽度可以大于 18 位(18Kb RAM)或者 36 位(36Kb RAM)。在该模式下,一个端口专用于读操作,另一个端口专用于写操作。在简单双端口模式下,读/写一侧可以是变化的,而另一个端口固定为 32/36 或者 64/72。双端口 36Kb RAM 的所有侧都可以是可变的宽度的。图 1.7 给出了简单双端口和真正双端口 BRAM 的配置模式。

#### 2. 检错和纠错

每个 64 位宽的 BRAM 能产生、保存和使用 8 位额外的海明码。在读过程中,纠正单比特错误和检测两比特错误操作。在写或者读外部 64~72 位宽的存储器时,也可以使用 ECC 逻辑。

#### 3. FIFO 控制器

图 1.8 给出了 UltraScale 的内建 FIFO 结构。每个 BRAM 能配置成一个 36Kb FIFO 或者一个 18Kb FIFO。内建的 FIFO 控制器用于单时钟(同步)或者双时钟(异步或者多数据率)操作,内部递增地址。此外,提供了四个握手信号:满、空、可编程满和编程空。可编程的标志允许设计者指定 FIFO 计数器的值,这个值用于控制标志变成活动

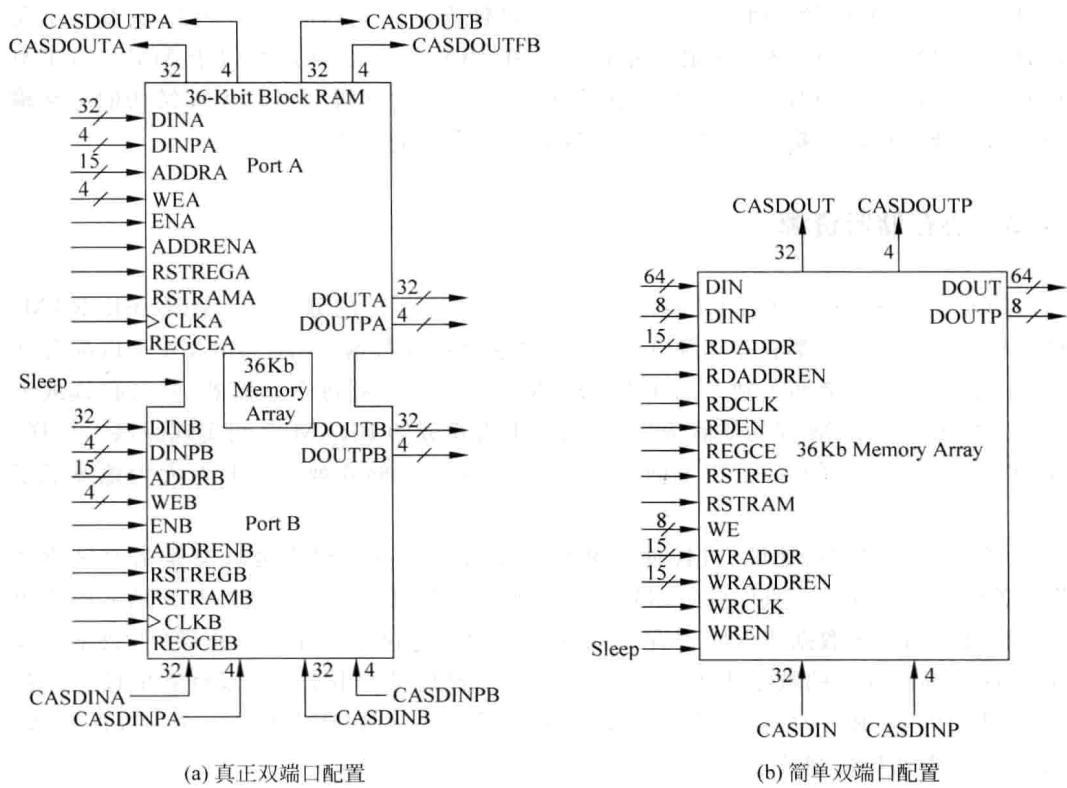


图 1.7 双端口 RAM 的两种配置方式

状态。设计者可以对 FIFO 的宽度和深度进行编程,以支持在一个 FIFO 上不同的读端口和写端口宽度。一个专用的级联路径允许容易创建更深的 FIFO。

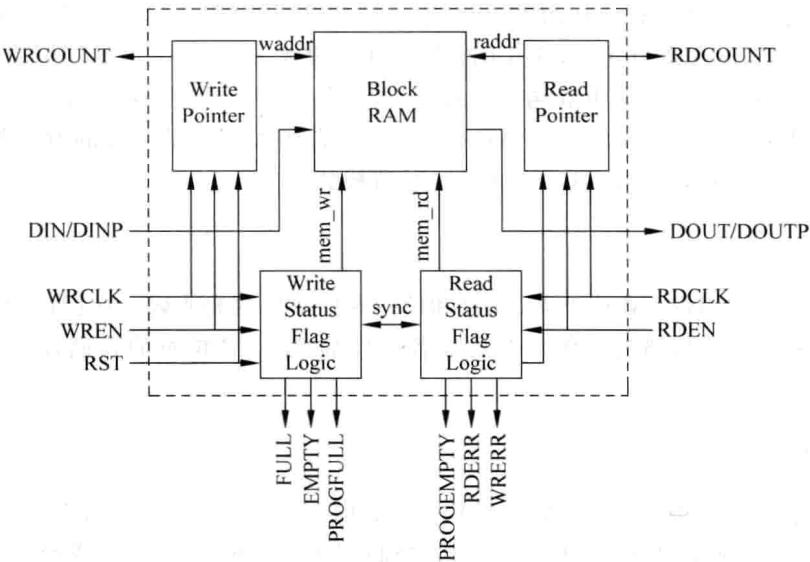


图 1.8 内建 FIFO 的结构图

#### 1.4.4 专用的 DSP 模块

DSP 应用使用大量的二进制乘法器和累加器。这些操作可以在专用的 DSP 切片中实现。如图 1.9 所示, UltraScale 结构内专用的、低功耗 DSP 切片。

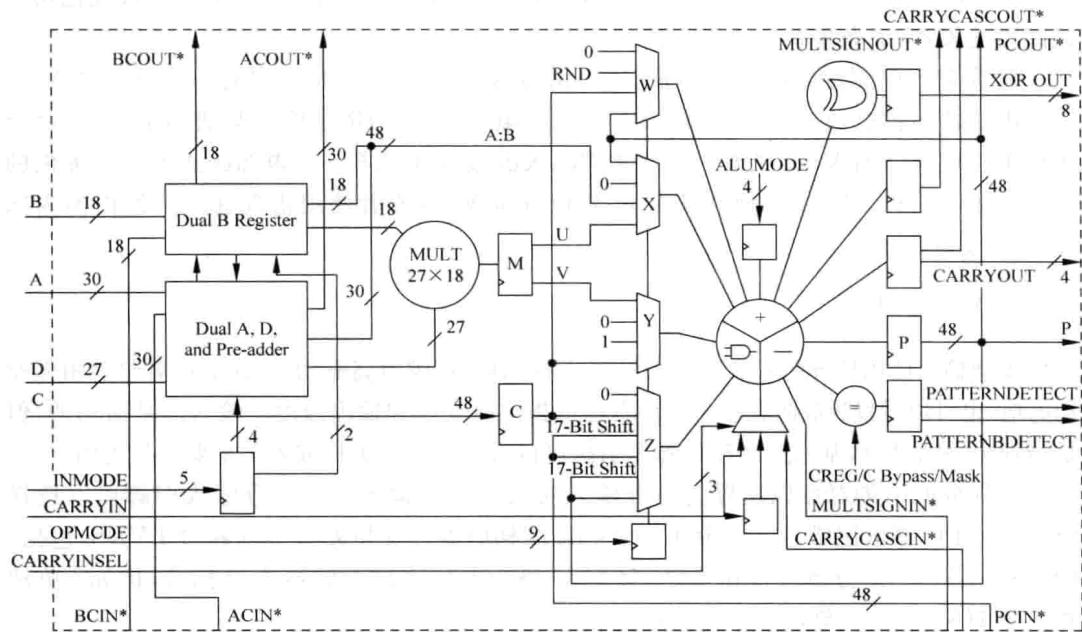


图 1.9 XtremeDSP DSP48A1 DSP 模块内部结构

每个 DSP 切片包含:

- (1) 一个专用的  $27 \times 18$  位二进制补码乘法器;
- (2) 一个 48 位的累加器。

在应用时,可以动态旁路一个乘法器,两个 48 位的输入能送到一个单指令多数据(Single Instruction Multiple Data, SIMD)算术单元(两个 24 位加法/减法/累加或者四个 12 位加法/减法/累加)或者一个逻辑单元。该逻辑单元可以产生使用两个操作数的 10 种不同逻辑功能中任何一种功能。

DSP 包含一个额外的预加法器。典型地,用于对称滤波器。在一个高密度封装的设计中,这个预加法器改善了性能。同时,将 DSP 切片的数量最多减少了 50%。96 位宽度 XOR 功能,可以编程为 12、24、48 或者 96 位宽度。这样,当执行向前纠错和循环冗余检查算法时,改善了实现性能。

DSP 也包含一个 48 位宽度的模式检测器,它可以用于收敛或者对称的四舍五入。当与逻辑单元相结合时,模式检测器也可以实现 96 位宽度的逻辑功能。

### 1.4.5 输入/输出块

I/O 引脚的数目取决于器件和封装, 每个 I/O 都是可配置的, 并且遵守大量的 I/O 标准。I/O 可以分成宽范围(High Range, HR)或者高性能(High Performance, HP)两种类型。HR I/O 提供了最宽范围的电压支持, 其范围为 1.0~3.3V。HP I/O 被优化用于最高性能操作, 其范围为 1.0~1.8V。

所有 I/O 引脚按组(Bank)构成, 每个组包含 52 个 I/O。每个组有一个公共的 V<sub>CCO</sub> 为输出缓冲区供电, 它也为一些输入缓冲区供电。此外, HR 组能分割成两个半组, 每个半组由它们自己的 V<sub>CCO</sub> 供电。一些单端输入的缓冲区要求一个内部的或者外部提供的参考电压(Vref)。Vref 引脚可以由 PCB 板直接驱动, 或者使用出现在每个组内的内部 Vref 生成器产生。

#### 1. I/O 电特性

单端输出使用传统的 CMOS 下拉或者上拉结构, 将逻辑高驱动为 V<sub>CCO</sub> 或者将逻辑低驱动为地, 也可以设置为高阻状态。设计者可以指定抖动率和输出强度。输入总是活动的, 但是当输出是活动时, 常常被忽略。每个引脚可以选择一个弱上拉或者一个弱下拉电阻。

大部分的引脚对可以配置为差分输入对或者差分输出对。差分输入引脚可以选择使用 100Ω 内部电阻端接。所有 UltraScale 结构的器件支持差分标准超过 LVDS, 包括: RS422、RS485、差分 SSTL 和差分 HSTL。每个 I/O 支持存储器 I/O 标准, 比如: 单端和差分 HSTL, 以及单端和差分 SSTL。

一个三态的数字控制阻抗(T\_DCI)能控制输出驱动阻抗(串行端接)或者能提供输入信号到 V<sub>CCO</sub> 的并行端接, 或者分割(Thevenin)端接到 V<sub>CCO</sub>/2。这样, 设计者就不需要使用片外信号端接方法。除了节约空间外, 当处于输出模式或者三态时, 自动关闭端接。与片外端接相比, 降低了系统功耗。I/O 为 IBUF 和 IDELAY 提供了更好降低功耗的方式, 特别是在实现存储器接口方面。

#### 2. I/O 逻辑

##### 1) 输入和输出延迟

所有的输入和输出都可以配置成组合或者寄存模式。同时, 所有的输入和输出均支持双数据率(Double Data Rate, DDR)。任何一个输入和输出都可以单独配置到最大 1250 个 ps, 其分辨率为 5~15 个 ps。实现这个延迟功能的是 IDELAY 和 ODELAY 单元。可以在配置时, 设置延迟步长的数目。也可以在使用时, 递增或者递减。IDEDELAY 和 ODELAY 可以级联在一起, 在单一方向达到两倍的延迟。

##### 2) ISERDES 和 OSERDES

很多应用包含器件外的高速位串行 I/O 传输, 以及器件内的低速并行操作。因此, 这就要求在 I/O 逻辑内有一个串行化器和一个解串行化器。每个带有 IOSERDES (ISERDES 和 OSERDES) 的 I/O 能执行 2、4、或者 8 位的串行到并行, 以及并行到串行的转换。因此, 这些 I/O 逻辑特性能用作高性能的接口, 例如吉比特以太网/1000BaseX/

SGMII。

### 1.4.6 高速串行收发器

在同一个 PCB、背板和跨越较长距离的超快速串行数据传输中,高速串行收发变得日益重要。在这些传输中,其速度可以达到 100G/s 和 400G/s 的线卡。专用片上电路和差分 I/O,使得能够在这样高速的条件下满足信号完整性的要求。

UltraScale 结构的 FPGA 内,提供了两种类型的收发器,即 GTH 和 GTY。它们按照四个一组分配,称为一个四收发器。每个串行收发器包含一个发送器和一个接收器。表 1.4 给出了 UltraScale 结构 FPGA 收发器的信息。

表 1.4 UltraScale 结构 FPGA 收发器的信息

参数	Kintex UltraScale	Virtex UltraScale	
收发器类型	GTH	GTH	GTY
收发器数量	16~64	20~52	0~52
最大数据率	16.3Gb/s	16.3Gb/s	32.75Gb/s
最小数据率	0.5Gb/s	0.5Gb/s	0.5Gb/s
应用例子	背板,PCI-E Gen 4	背板,PCI-E Gen 4	100G+光纤,片到片,25G+背板

串行发送器和接收器有单独的电路,它使用高级 PLL 结构。通过 4~25 之间的可编程倍频因子,将输入参考时钟进行倍频,将其变成位串行数据时钟。每个收发器都有大量用户定义的特性和参数。这些参数可以在器件配置时设置,也可以在操作期间修改。

#### 1. 发送器

发送器的基本功能是一个并行到串行的转换器,其转换因子为:16、20、32、40、64 或者 80(用于 GTH);16、20、32、40、64、80 和 128 或者 160(用于 GTY)。这样,允许在高性能的设计中,设计者可以在数据路径宽度和时间裕度之间进行权衡。这些发送器的输出用于驱动带有单通道差分输出信号的 PC 板。TXOUTCLK 是一个经过合理分频的串行数据时钟,可以直接将来自内部逻辑的并行数据进行寄存。输入的并行数据可以通过可选的 FIFO,额外的电路用于支持 8B/10B,64B/66B 或者 64B/67B 的编码策略。位串行输出信号驱动带有差分信号的两个封装引脚。这个输出信号对有可编程的信号摆率和可编程的预加重和后加重,用于补偿 PC 板的损耗和其他互连特性。对于较短的通道,可以降低摆率以降低功耗。

#### 2. 接收器

发送器的基本功能是一个串行到并行的转换器,将输入的位串行差分数据转换为并行的字,其转换因子为:16、20、32、40、64 或者 80(用于 GTH);16、20、32、40、64、80、128 或者 160(用于 GTY)。这样允许在高性能的设计中,设计者可以在数据路径宽度和时间裕度之间进行权衡。接收器提取输入的差分数据流,将其送入可编程的 DC 自动增益控制、线性和判决返回均衡器(用于补偿 PC 板、电缆、光纤和其他互连特性)。同时,使用参

考时钟输入来初始化时钟识别(这里不需要单独的时钟线)。数据模式使用非归零码(Non-return-to-zero, NRZ)编码,以及可选的用于保证数据充分跳变的编码策略。通过使用 RXUSERCLK 时钟将数据传输到 FPGA 的内部逻辑中。对于短的通道,收发器提供了一个特殊的低功耗模式,可将功耗降低大约 30%。对于 DC 自动增益控制、线性和判决返回均衡器,可以选择使用“自适应”来自动地理解和补偿不同的互连特性。这样,使能更高的裕度。

### 3. 带外信号

收发器提供了带外信号,经常用于从发送器发送低速信号到接收器,而高速串行数据发送没有处于活动状态。典型地,用于低功耗模式下地连接,或者没有初始化的情况。这对于 PCI-E、SATA/SAS 和 QPI 应用来说,都是很有好处的。

#### 1.4.7 PCI-E 模块

所有 UltraScale 结构的 FPGA 集成至少一个 PCI-E 模块,它能配置为一个端点或者根端口。该模块遵守 PCI-E 基本规范 3.0 版本。根端口能用于构建用于兼容根联合体的基础,以允许通过 PCI-E 协议定制 FPGA 到 FPGA 的通信,以及添加到 FPGA 的 ASSP 设备,例如:以太网控制器或者光纤通道 HBA。

设计者可充分配置这个模块,用于满足系统要求。在 2.5Gb/s,5.0Gb/s 或 8.0Gb/s 的速率下,可以配置操作 1、2、4 或者 8 个通道的数据。对于高性能的应用来说,该模块提供的高级缓冲技术提供了一个灵活的载荷,最大为 1024 个字节的数据。集成块到集成高速收发器的接口,用于串行连接;与 BRAM 的接口,用于数据缓冲。总之,这些元素用于实现 PCI-E 协议的物理层、数据链路层和交易层。

Xilinx 提供了轻量级、可配置、容易使用的 IP,它将各种构件块(PCI-E 集成块、收发器、BRAM 和时钟资源)集成到端点或者根设备的应用中。系统设计人员可以控制很多可配置的参数:通道宽度、最大负载大小、FPGA 逻辑接口速度、参考时钟频率,以及基地址寄存器译码和过滤。

#### 1.4.8 Interlaken 集成块

一些 UltraScale 结构的 FPGA 包含了 Interlaken 集成块。Interlaken 是一个可扩展的片到片的互连协议,其传输速度从 10Gb/s 到 150Gb/s。该集成模块遵守 Interlaken 规范 1.2 版本,在 1~12 个通道上可以剥离数据和聚集数据。允许的配置是:

- (1) 10.3125Gb/s 速率下,1~12 个通道;
- (2) 12.5Gb/s 速率下,1~12 个通道;
- (3) 25.78125Gb/s 速率下,1~6 个通道。

每个模块灵活地支持最大 150Gb/s 的速度。使用多个 Interlaken 块,某些 UltraScale 结构的 FPGA 很容易可靠地实现 Interlaken 开关和桥。

### 1.4.9 Ethernet 模块

UltraScale 结构的 FPGA 内集成的 100G 以太网模块,该模块遵守 IEEE Std802.3ba 规范。提供了 100Gb/s 的以太网端口,带有设计者宽范围的定制和统计搜集。集成以太网模块支持  $10 \times 10.3125\text{Gb/s}$ (CAUI) 和  $4 \times 25.78125\text{Gb/s}$ (CAUI-4) 配置。此外,集成以太网模块包含 100GMAX 和 PCS 逻辑,支持 IEEE Std1588v2 1-step 和 2-step PTP 时间戳。

### 1.4.10 系统监控器模块

如图 1.10 所示,UltraScale 结构的 FPGA 内部提供了系统监控器模块,通过片上供电传感器和温度传感器对环境的监控,扩展了系统的整体安全性和可靠性。此外,系统监控器还提供了最多 17 个设计者分配的外部模拟输入。系统监控器支持监控器件内部的主要供电电压(例如  $V_{CCINT}$ 、 $V_{CCAUX}$ 、 $V_{CCBRAM}$  和  $V_{CCO}$ )。

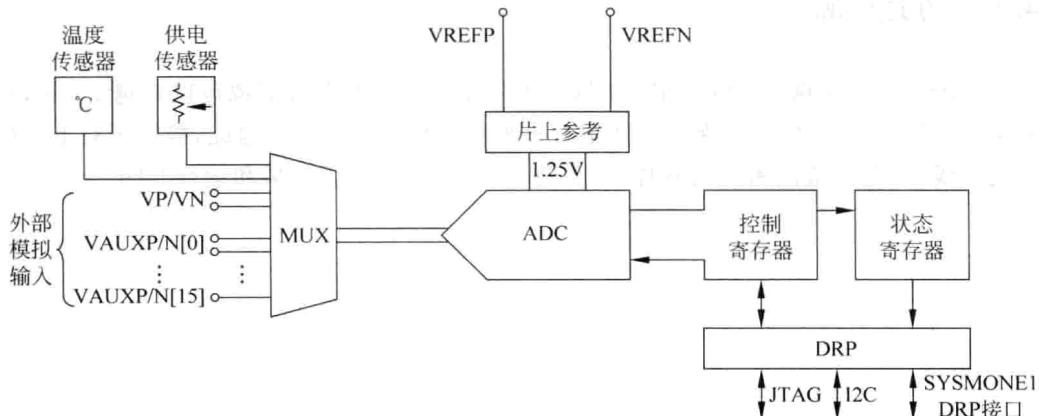


图 1.10 系统监控器框图

通过使用 10 位 200kSPS 的 ADC,将传感器的输出和模拟输入数字化,并且将测量的结果保存在寄存器中。它能通过 FPGA 内部、JTAG 或者 I2C 接口,访问这些结构,通过 I2C 接口,系统管理器/主机可以在配置前对片上监控进行访问。

### 1.4.11 配置模块

对于 UltraScale 结构的 FPGA 来说,SRAM 类型的内部锁存器用于保存定制的配置数据。由于配置保存是易失性的,所以当 FPGA 上电的时候,需要重新加载数据。任何时候,通过将 FPGA 的 PROGRAM\_B 拉低,就可以重新加载保存的配置数据。UltraScale 结构的 FPGA 有三个模式引脚,用于确定加载配置数据的方法,其他专用的配置数据引脚用来简化配置的过程。

SPI(串行 NOR)接口( $\times 1$ , $\times 2$ , $\times 4$  和双 $\times 4$ 模式)和 BPI(并行 NOR)接口( $\times 8$  和

×16 模式)是两种用于配合 FPGA 的非常普通的方法。设计者能将 SPI 或者 BPI Flash 直接连接到 FPGA, FPGA 的内部配置逻辑读取来自外部 Flash 的比特流, 然后配置自己。在配置的过程中 FPGA 自动检测总线的宽度, 无须使用外部控制或者开关进行识别。较大的数据宽度增加配置的速度, 减少了配置 FPGA 需要花费的时间。

在主模式下, 通过 FPGA 内的一个内部生成时钟, FPGA 能驱动配置时钟; 或者为了更高速度的配置, FPGA 也可以使用一个外部的配置时钟源。这样, 允许高速的配置器件。并且, 容易使用主模式的特性。FPGA 配置也支持从模式, 其数据宽度最多到 32 位, 这对于使用处理器驱动的配置非常有用。此外, 新的媒体控制访问端口 (Media Control Access Port, MCAP) 提供了在 PCI-E 集成模块和配置逻辑之间的直接连接。这样, 简化了 PCI-E 的配置过程。

使用 SPI 或者 BPI Flash, FPGA 可以使用不同的镜像重新配置自己。这样, 无须使用外部的控制器。当数据发送过程中出现错误时, FPGA 能重新加载它最初的设计, 用于确保在该过程结束时 FPGA 可用。当最终产品出货后, 对产品进行升级时, 这种方法是非常有用的。

#### 1.4.12 互连资源

在 UltraScale 结构的 FPGA 内, 不同长度的垂直和水平布线资源可以跨越 1、2、4、5、12 或者 16 个 CLB。这样, 确保信号能很容易地从源传输到目的。因此, 提供了对下一代宽总线布线(甚至是最高密度的器件)的支持。同时, 也改善了结果和运行时间。



大量的布线资源使得设计者能够轻松地处理复杂的布线需求。例如, 在设计中可能需要将一个信号从一个远端位置传送到另一个远端位置, 或者在一个大范围内进行信号交换。通过利用这些广泛的布线资源, 设计者可以有效地解决这些问题, 提高设计的灵活性和效率。

### 本章小结

本章主要介绍了 UltraScale 器件的架构, 包括其核心组件 CLB、IOB 和全局布线, 以及它们如何协同工作以实现高性能和低功耗。我们还探讨了如何利用 Vivado Design Suite 中的综合器和布局布线器来优化设计, 并通过具体的例子展示了设计流程。最后, 我们简要介绍了 FPGA 的配置机制, 包括 SPI 和 BPI Flash 两种常见的配置方式。

本章将通过一个设计例子,介绍 Vivado 工程模式和非工程模式设计流程,以帮助读者掌握不同模式下的 Vivado 设计方法。

## 2.1 工程模式设计流程

本节将分几个部分来介绍 Vivado 工程模式设计流程。

### 2.1.1 启动 Vivado 集成开发环境

下面给出启动 Vivado 集成开发环境的四种方法,主要包括:

- (1) 在 Windows 7 操作系统主界面下,选择开始→所有程序→Xilinx Design Tools→Vivado 2013.3→Vivado 2013.3。
- (2) 在 Windows 7 操作系统桌面上,单击如图 2.1 所示的图标。



图 2.1 Vivado 桌面图标

- (3) 如图 2.2 所示,在 Window 主界面左下角的命令行中,输入 Vivado。然后,按 Enter 键。

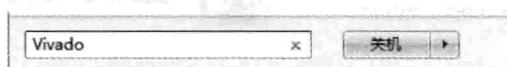


图 2.2 命令行输入

**注:** 当输入 Vivado 命令后,系统自动运行 Vivado - mode gui,启动 Vivado 集成开发环境。如果设计者需要帮助,则输入 Vivado - help 命令。

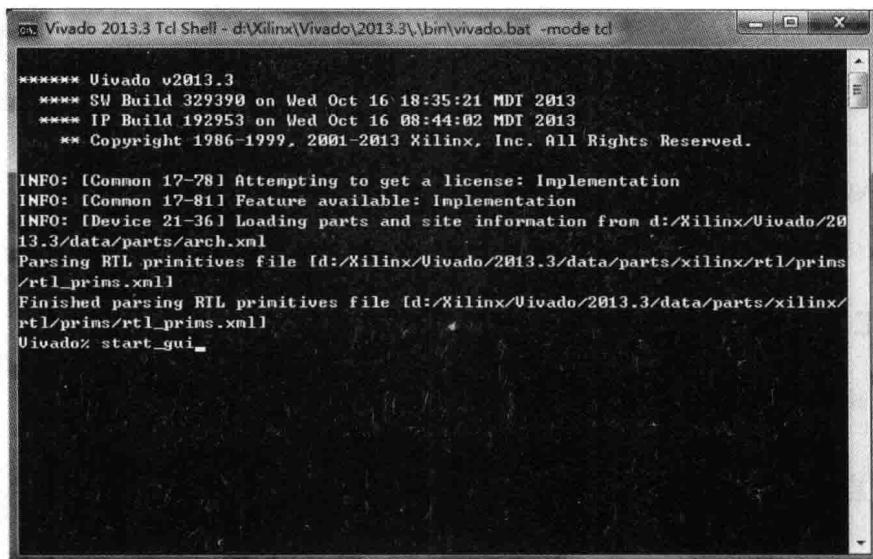
- (4) 在 Windows 7 操作系统主界面下,选择开始→所有程序→Xilinx Design Tools→Vivado 2013.3→Vivado 2013.3 Tcl Shell。

- ① 出现如图 2.3 所示的 Vivado 2013.3 Tcl Shell 对话框界面。

② 在 Vivado %命令提示符的后面,输入

```
start_gui
```

然后,按回车键。系统将启动 Vivado 集成开发环境。



```
Vivado 2013.3 Tcl Shell - d:\Xilinx\Vivado\2013.3\bin\vivado.bat -mode tcl

*****
**** Vivado v2013.3
**** SW Build 329390 on Wed Oct 16 18:35:21 MDT 2013
**** IP Build 192953 on Wed Oct 16 08:44:02 MDT 2013
** Copyright 1986-1999, 2001-2013 Xilinx, Inc. All Rights Reserved.

INFO: [Common 17-78] Attempting to get a license: Implementation
INFO: [Common 17-81] Feature available: Implementation
INFO: [Device 21-36] Loading parts and site information from d:/Xilinx/Vivado/2013.3/data/parts/arch.xml
Parsing RTL primitives file [d:/Xilinx/Vivado/2013.3/data/parts/xilinx/rtl/prims/rtl_prims.xml]
Finished parsing RTL primitives file [d:/Xilinx/Vivado/2013.3/data/parts/xilinx/rtl/prims/rtl_prims.xml]
Vivado> start_gui
```

图 2.3 启动 Vivado 2013.2 Tcl Shell

### 2.1.2 建立新的设计工程

建立新的设计工程的步骤主要包括:

(1) 进入到 Vivado 2013.3 主界面。如图 2.4 所示,该界面分为 Getting Started 和 Documentation 两个主要的入口。

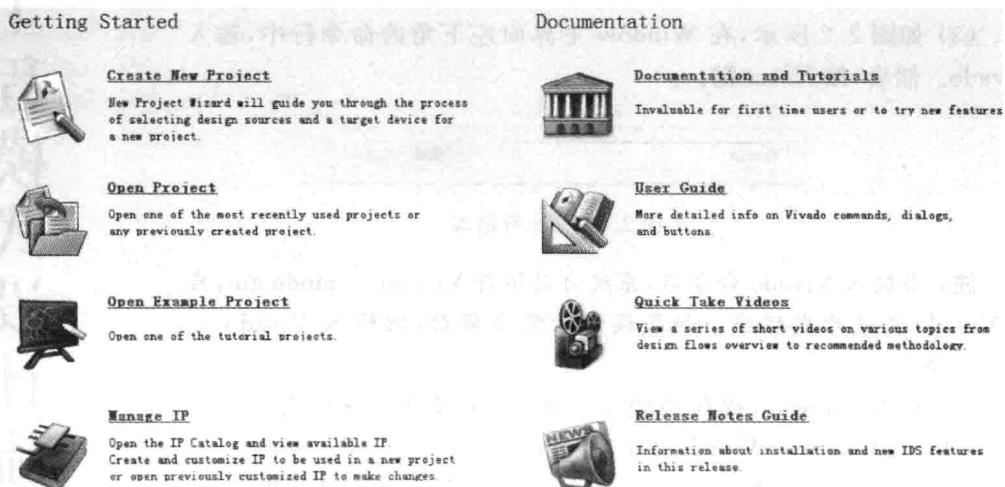


图 2.4 Vivado 2013.3 主界面

① 在 Getting Started 入口下,提供了以下功能:

Create New Project(创建新的工程): 该选项将打开创建新工程向导,指导设计者创建不同类型的工程。设计者也可以通过使用该向导,导入通过 PlanAhead 工具所创建的工程(.ppr 扩展名)或者通过 ISE 设计套件所创建的工程(.xise 扩展名)。

Open Project(打开工程): 打开浏览器,设计者可以打开 Vivado 集成环境工程文件(.xpr 扩展名)。它也显示最后 10 个以前打开的工程。

**注:** 10 个是默认值。如果设计者想改变这个数字,需要在 Vivado 主界面主菜单下,选择 Tools→Options。出现如图 2.5 所示的界面,选择 General 标签。在 Number of recent projects to list(列出最近的工程的个数)右侧的下拉框中,修改数字。

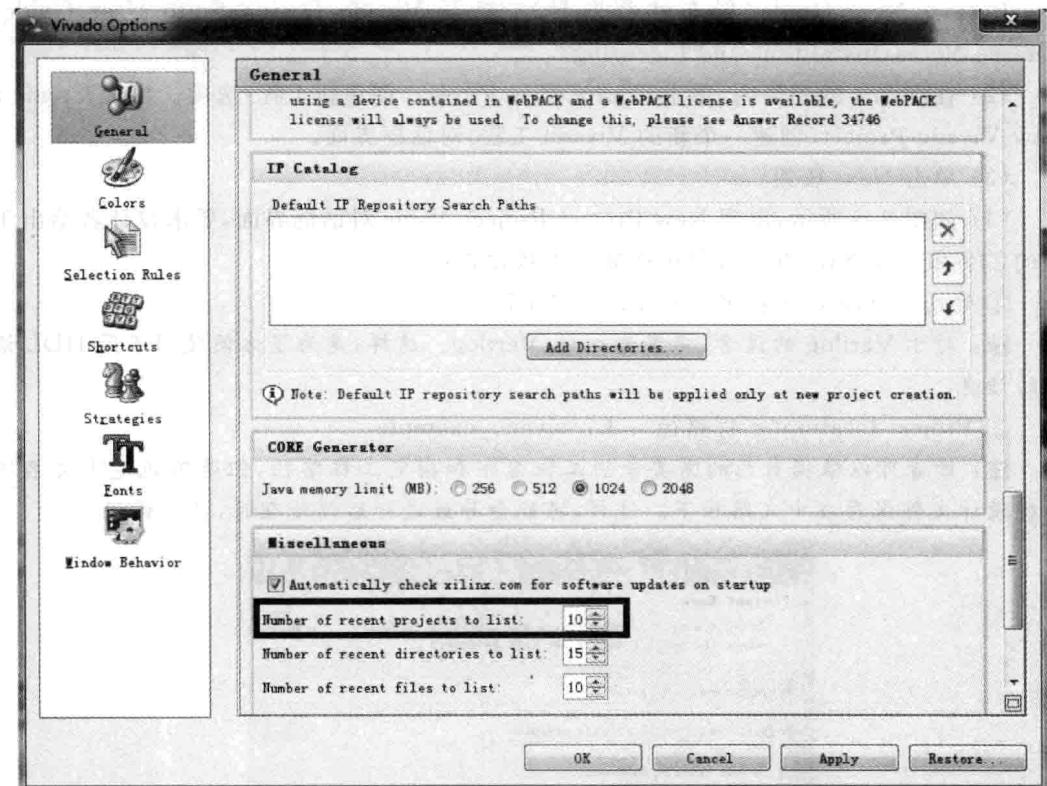


图 2.5 修改默认的打开工程个数

Open Example Project(打开实例工程),打开下面的例子工程:

BFT Core——小的 RTL 工程。

CPU(HDL)——大的,混合语言 RTL 工程。

CPU(综合的)——大的,综合的网表工程。

Wave(HDL)——小的工程,包括三个嵌入的 IP 核;设计者可以通过这个设计,学习如何使用集成的 IP 核。

Zynq System——基于 ZC702 评估板的 Vivado IP 集成器 Zynq 器件。使设计者在 Vivado 集成开发环境下完成设计,生成比特流。然后,在软件开发工具 SDK 中,开发软件应用程序代码。

Microblaze System——基于 KC705 评估板的小 Vivado IP 集成器 Microblaze 处理器设计。在 SDK 中开发应用程序代码。在 Vivado 设计套件中, 使用 SDK 产生的 ELF 文件, 对设计进行仿真。

Manage IP(管理 IP)——在一个已存在的工程外, 打开 IP 目录。IP 目录显示了 Xilinx、第三方和用户定制的 IP。设计者可以查看或者重新定制已经存在的 IP 核。

- ② 在 Documentation 入口下, 提供了以下的功能:
  - Documentation and Tutorials(文档和教程): 打开 Xilinx 的教程和支持设计数据。
  - User Guide(用户指南): 打开 Vivado Design Suite User Guide。
  - Quick Take Videos(快速打开视频): 打开 Xilinx 视频教程。
  - Release Note Guide(发布注释向导): 打开 Vivado Design Suite User Guide: Release Notes, Installation, and Licensing。

(2) 在图 2.4 的界面内, 单击 Create New Project(创建新工程)选项。出现 Create a New Vivado Project(创建一个新的 Vivado 工程)对话框界面。

- (3) 单击 Next 按钮。
- (4) 如图 2.6 所示, 出现 New Project-Project Name 对话框界面, 要求设计者给出工程的名字和工程路径, 在该设计中按如下参数设置:

- ① Project name(工程名字): gate\_VHDL。  
注: 对于 Verilog 的读者, 名字为 gate\_Verilog。这样, 是为了方便使用不同 HDL 语言的读者。
- ② Project location(工程路径): E:/vivado\_example  
注: 读者可以根据自己的需要命明工程名字和指定工程路径, 但是不能起中文名字和将设计文件保存在中文路径下。这样, 可能会导致进行后续处理时, 产生错误。

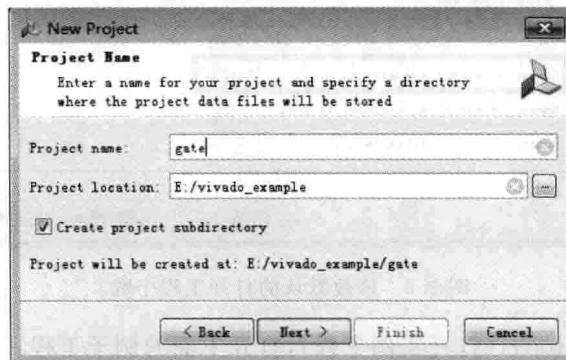


图 2.6 给出工程路径和工程名字

- (5) 单击 Next 按钮。
- (6) 如图 2.7 所示, 出现 New Project-Project Type(新工程-工程类型)对话框界面。在该界面内提供了下面可选的工程类型:
  - ① RTL Project: 选择该选项, 设计者可以添加源文件、生成 IP、运行寄存器传输级(RTL)分析、综合、实现、设计规划和分析。
  - ② Post-synthesis Project: 选择该选项, 设计者可以添加源文件、查看器件资源、运

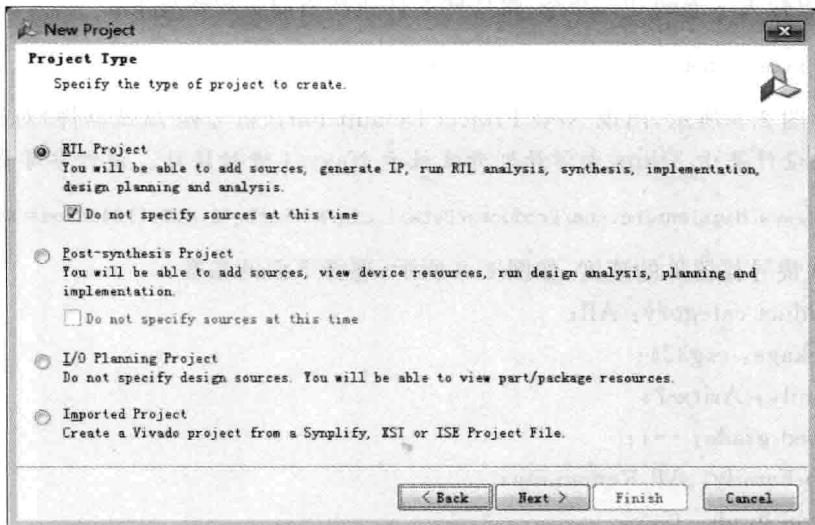


图 2.7 指定工程的类型

行设计分析、规划和实现。

③ I/O Planning Project：选择该选项，不能指定设计源文件，但是可以查看器件/封装资源。

④ Imported Project：从 symplify、xst 或者 ISE 工程文件中，创建一个 vivado 工程。在该设计中，按如下参数设置：  
 ① 选中 RTL Project；  
 ② 选中 Do not specify sources at this time(此次不指定源文件，表示在生成工程后，再添加设计源文件到工程中)。

(7) 单击 Next 按钮。

注：如图 2.8 所示，设计者也可以在 Vivado 主界面下面的 Tcl Console(Tcl 控制台)窗口下，输入 Tcl 命令，来创建工程。



图 2.8 Tcl 控制台界面

① 下面给出用于创建工程的 Tcl 命令格式模板(读者可以根据情况进行修改)：

```
create_project project_Name /exampleDesigns/project_8 - part xc7vx485tffg1157 - 1
```

默认工程类型是 RTL。

② 如果读者想创建一个网表工程，按照下面 Tcl 模板格式输入命令：

```
set_property design_mode GateLvl [current_fileset]
```

③ 可以输入下面的 Tcl 命令，在工程中添加设计源文件：

```
add_files -norecurse -scan_for_includes ./designs/oneFlop.v
```

④ 可以输入下面的 Tcl 命令, 将这些文件放到当前工程路径下:

```
import_files -norecurse ./designs/oneFlop.v
```

(8) 如图 2.9 所示, 出现 New Project-Default Part(新工程-默认器件)对话框界面。

注: 该设计基于 Xilinx 大学计划开发板卡 Nexys4 进行设计。设计参考资料见下:

<https://www.digilentinc.com/Products/Detail.cfm?NavPath=2,400,1184&Prod=NEXYS4>

为了加快寻找器件的速度, 如图 2.9 所示, 选择下面的参数:

- ① Product category: All;
- ② Package: csg324;
- ③ Family: Artx-7;
- ④ Speed grade: -1;
- ⑤ Sub-Family: All Remaining;
- ⑥ Temp grade: C。

可以看到, 所选择器件的型号是 xc7a100tcsg324-1。

(9) 单击 Next 按钮。

(10) 出现 New Project-New Project Summary(新工程-新工程总结)对话框界面。该对话框给出了工程类型、工程名字和器件信息说明。

(11) 单击 Finish 按钮。

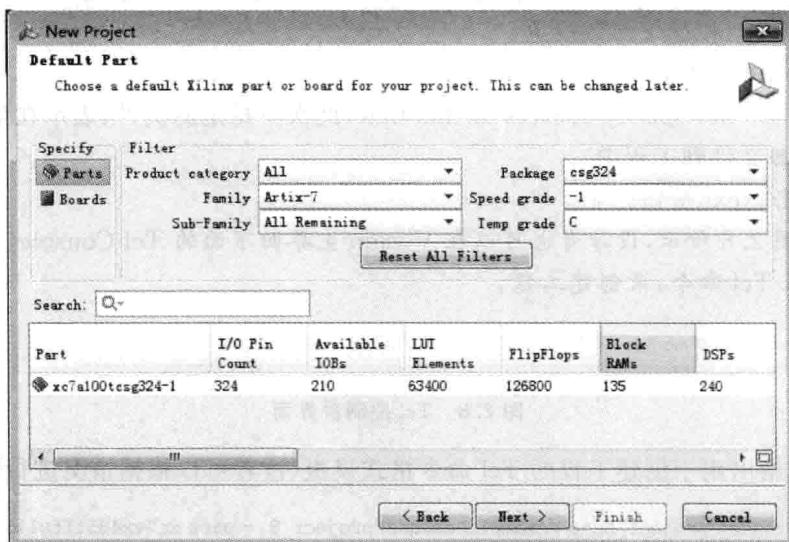


图 2.9 器件类型选择界面

### 2.1.3 Vivado 设计主界面及功能

本节介绍 Vivado 设计主界面及功能。内容包括: 流程处理主界面及功能、工程管理器主界面及功能、工作区窗口和设计运行窗口。

## 1. 流程处理主界面及功能

如图 2.10 所示,在 Vivado 左侧的 Flow Navigator(流程向导)界面中给出了处理的主要流程,包括:

### (1) Project Manager(工程管理器):

- ① Project Settings(工程设置);
- ② Add Sources(添加源文件);
- ③ IP Catalog(IP 目录)。

### (2) IP Integrator(IP 集成器):

- ① Create Block Design(创建块设计);
- ② Open Block Design(打开块设计);
- ③ Generator Block Design: 生成模块设计。

### (3) Simulation(仿真):

- ① Simulation Settings(仿真设置);
- ② Run Simulation(运行仿真)。

### (4) RTL Analysis(RTL 分析):

- ① Open Elaborated Design(打开详细的设计)。

### (5) Synthesis(综合)

- ① Synthesis Settings(综合设置);
- ② Run Synthesis(运行综合);
- ③ Open Synthesized Design(打开综合后的设计)。

### (6) Implementation(实现):

- ① Implementation Settings(实现设置);
- ② Run Implementation(运行实现);
- ③ Open Implemented Design(打开实现后的设计)。

### (7) Program and Debug(编程和调试):

- ① Bitstream Settings(比特流设置);
- ② Generate Bitstream(生成比特流);
- ③ Open Hardware Manager(打开硬件管理器);
- ④ Launch iMPact(启动 iMPACT 工具)。

## 2. 工程管理器主界面及功能

如图 2.11 所示,该窗口为 Project Manager(工程管理器窗口界面),所有的设计文件及类型,以及这些设计文件之间的关系均显示在该界面窗口下。

(1) Sources(源窗口)——该窗口允许设计者管理工程源文件,包括:添加、删除和对源文件重新排序,用于满足指定的设计要求。当下面作为工程的一部分时,显示它们:

① Design Source(设计源文件)——显示源文件类型,这些源文件类型包括:Verilog、VHDL、NGC/NGO、EDIF、IP 核、数字信号处理(DSP)模块、嵌入式处理器和

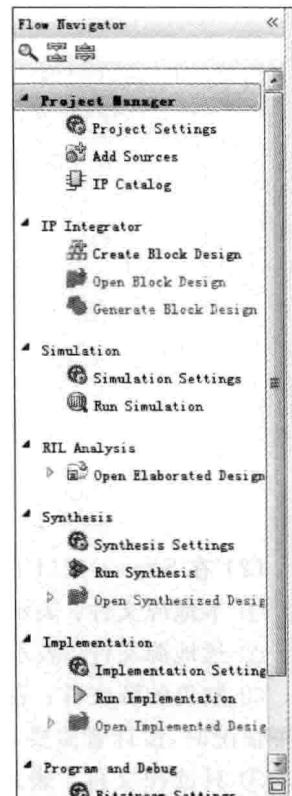


图 2.10 Flow Navigator

管理器界面

XDC/SDC 约束文件。

② Constraint file(约束文件)——显示用于对设计进行约束的约束文件。

③ Simulation Sources(仿真源文件)——显示用于仿真的测试源文件。

④ IP Core(IP 核)。

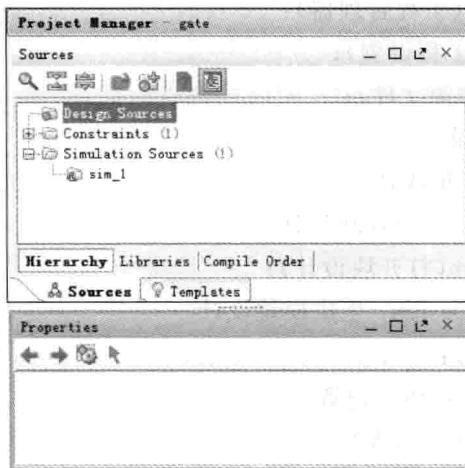


图 2.11 Project Manager 窗口界面

(2) 在 Source 窗口下, 使用下面的图标:

① 本地源文件: 表示文件存在于本地工程路径下。

② 远地源文件: 表示使用的文件并不在当前的本地工程路径下。

③ 缺失的源文件: 表示找不到工程设计中所使用的设计源文件。当在工程中遇到这种情况时, 设计者需要人工添加缺失的源文件。

④ 只读源文件: 表示该文件在 Vivado 集成开发环境下为只读文件, 不可以进行修改。

(3) 源文件窗口视图: 如图 2.11 所示, 源文件窗口提供了下面的视图, 用于显示不同的源文件。

① Hierarchy (层次): 层次视图显示了设计模块和例化的层次。顶层模块定义了用于编译、综合和实现的设计层次。Vivado 集成开发环境自动地检测顶层的模块, 但是设计者可以使用 Set as Top 命令手工定义顶层模块。

② IP Source(IP 源): IP 源文件显示了由 IP 核所定义的所有文件。

③ Library(库): 库视图显示了保存到各种库的源文件。

④ Compile Order(编译顺序): 该视图从最开始到结束, 显示了所有需要编译的源文件顺序。通常地, 顶层模块是编译的最后文件。基于定义的顶层模块和精细的设计, 设计者可以允许 Vivado 集成环境自动确定编译顺序。此外, 通过使用 Hierarchy Update 浮动菜单命令, 设计者可以人工控制设计的编译顺序, 即重新安排源文件的顺序。

(4) 源窗口工具栏命令

① 图标: 单击该图标, 将打开查找工具条, 允许快速的定位源文件窗口内的对象。

② 图标: 单击该图标, 将在源窗口中展开层次设计中所有设计文件。

- ③ 图标：单击该图标，将所有的设计源文件都缩回去，只显示顶层对象。
- ④ 图标：单击该图标，更新源文件窗口，将其聚焦在当前所选择的对象上。在包含很多源文件的大的设计中，这非常有用。
- ⑤ 图标：单击该图标，将添加或者创建 RTL 源文件、仿真源文件、约束文件、DSP 模块、嵌入式处理器或者已经存在的 IP。

### 3. 工作区窗口

如图 2.12 所示，该窗口下，给出了设计报告总结，并且可以实现设计输入和设计查看。

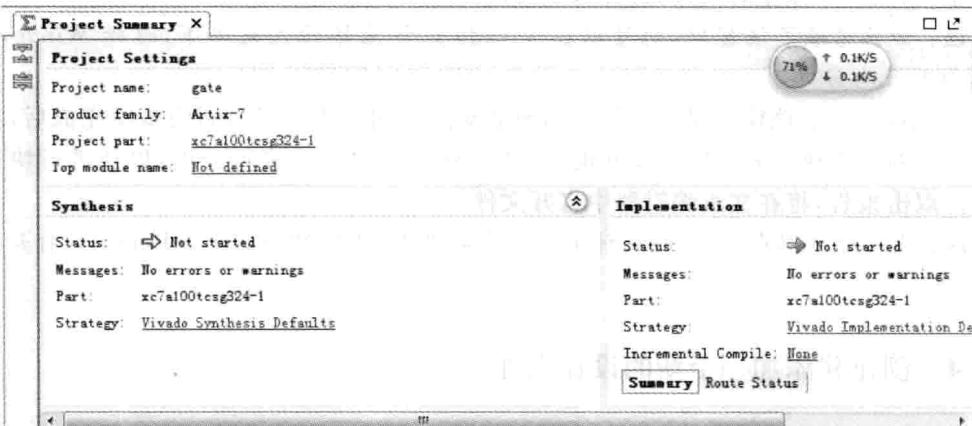


图 2.12 工作区窗口界面

### 4. 设计运行窗口

如图 2.13 所示，给出了 Design Runs(设计运行)对话框界面。该界面提供了下面的标签窗口。

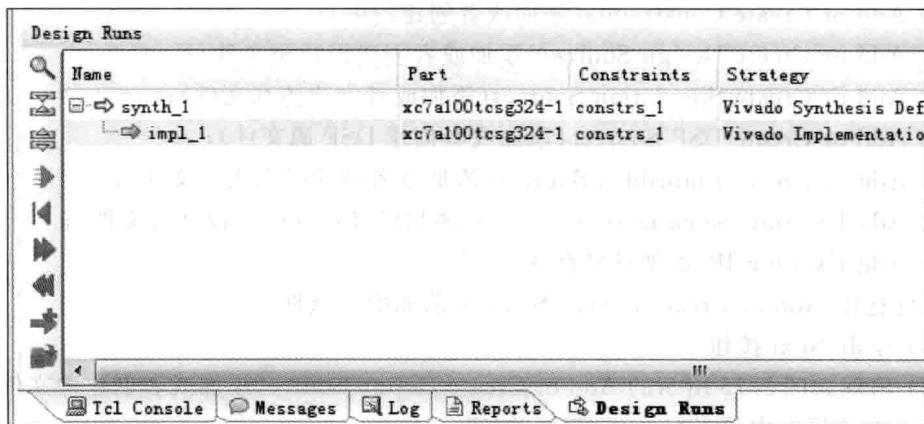


图 2.13 设计运行窗口界面

(1) **Tcl Console**: Tcl 控制台界面,可以在该界面下输入 Tcl 命令,来控制设计流程的每一步。

(2) **Message**: 消息窗口显示了设计和报告消息。通过不同的头部,对消息进行分组,以便设计者可以从不同的工具或者处理过程中快速地定位消息。所显示的消息有一个到相关文件的链接。设计者可以单击链接,在文本编辑器内打开 RTL 源文件。

**注:** 设计者可以在 Vivado 设计主界面主菜单下,选择 Windows→Messages, 打开消息窗口。

(3) **Log**: 显示对设计进行编译命令活动的输出状态。这些命令用于综合、实现和仿真。输出显示连续滚动格式,当新的命令运行时,就会覆盖输出显示。当在一个活动的运行中,启动一个命令时,自动打开这个窗口。

**注:** 如果隐藏了该窗口,则可以在 Vivado 设计主界面主菜单下,选择 Windows→Log, 打开日志窗口。

(4) **Reports**: 该窗口显示了用于当前活动运行的报告。当不同的步骤完成后,对报告进行更新。当执行完不同的步骤时,用不同的头部对报告进行分组,以便进行快速的定位。双击报告,将在文本编辑器中打开文件。

**注:** 设计者可以在 Vivado 设计主界面主菜单下,选择 Windows→Reports, 打开报告窗口。

## 2.1.4 创建并添加一个新的设计文件

本节将为该设计创建一个 VHDL/Verilog 设计文件。创建 HDL 设计文件的步骤主要包括:

(1) 在 Sources 窗口下,单击 按钮; 或者单击右键,出现浮动菜单,选择 Add Source...; 或者在 Vivado 主界面主菜单下,选择 File→Add Source...。

(2) 出现如图 2.14 所示的 Add Sources(添加源文件)对话框界面。该对话框界面提供了下面的选项:

- ① Add or Create Constraints(添加或者创建约束);
- ② Add or Create Design Sources(添加或者创建设计源文件);
- ③ Add or Create Simulation Sources(添加或者创建仿真文件);
- ④ Add or Create DSP Sources(添加或者创建 DSP 源文件);
- ⑤ Add or Create Embedded Sources(添加或者创建嵌入式源文件);
- ⑥ Add Existing Block Design Sources(添加已经存在的块设计源文件);
- ⑦ Add Existing IP(添加已经存在的 IP )。

在此选中 Add or Create Design Sources 前面的复选框。

(3) 单击 Next 按钮。

(4) 出现如图 2.15 所示的 Add or Create Design Source(添加或者创建源文件)对话框界面。在该界面中单击 Create File...按钮。

(5) 如图 2.16 所示,出现 Create Source File(创建源文件)对话框界面。在该界面内选择添加文件的类型和输入文件的名字。按下面参数进行设置:

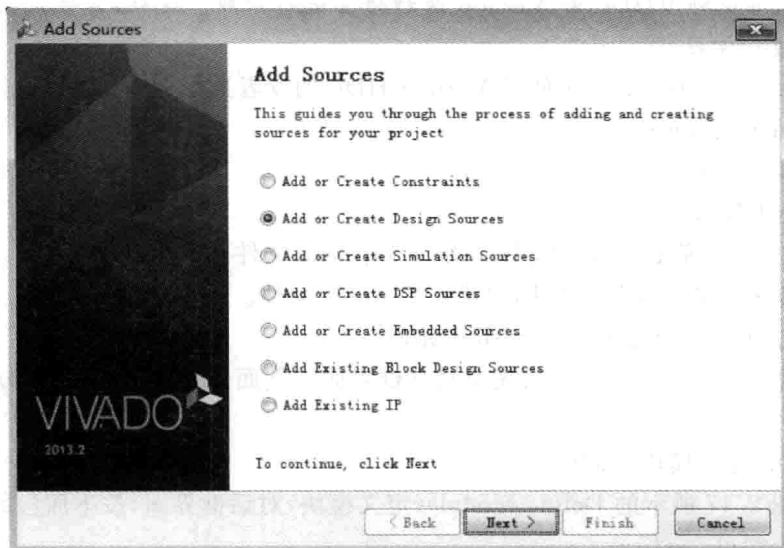


图 2.14 创建新的设计文件

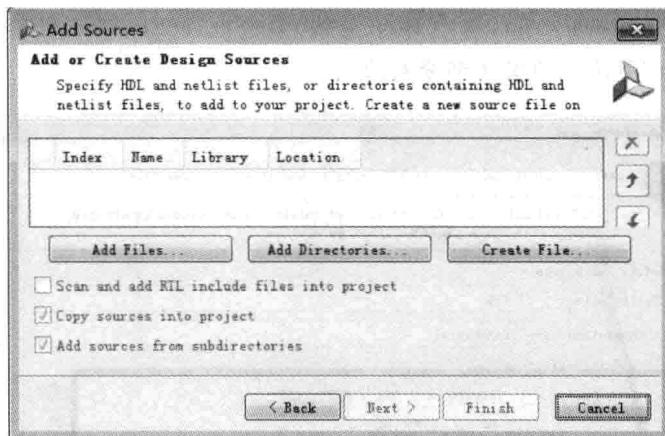


图 2.15 添加或者创建新文件选择对话框

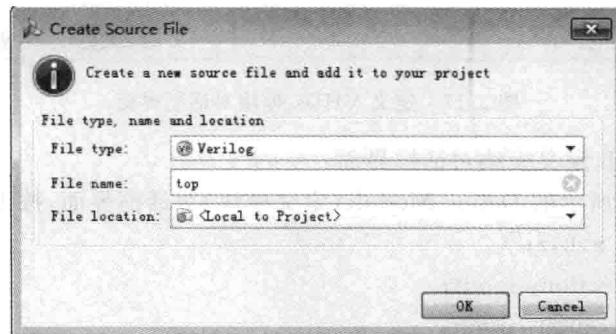


图 2.16 创建文件对话框界面

**注：**在本书中将 VHDL 和 Verilog 两种语言进行对照。这样，方便使用不同 HDL 语言的读者进行学习。

① File type: VHDL(对于使用 Verilog HDL 的读者选择 Verilog)；

② File name: top；

③ File location: Local to Project。

(6) 单击 OK 按钮。

(7) 在图 2.15 所示的对话框中，添加了 top.vhd 文件。

**注：**前面选择了 Verilog 文件，则生成了 top.v 文件。

(8) 单击图 2.15 界面中的 Finish 按钮。

(9) 出现 Define Module(定义模块对话框界面)，下面分 VHDL 和 Verilog 两个设计界面说明。

① VHDL 定义模块对话框界面。

出现如图 2.17 所示的 Define Module(定义模块)对话框界面，按下面参数设置：

添加三个端口：a,b,z；

对于端口 a,Direction : in；

对于端口 b,Direction: in；

对于端口 z,Direction: out, 选中 Bus(总线)复选框, MSB: 5, LSB: 0。

**注：**该声明和 VHDL 的实体部分对应。

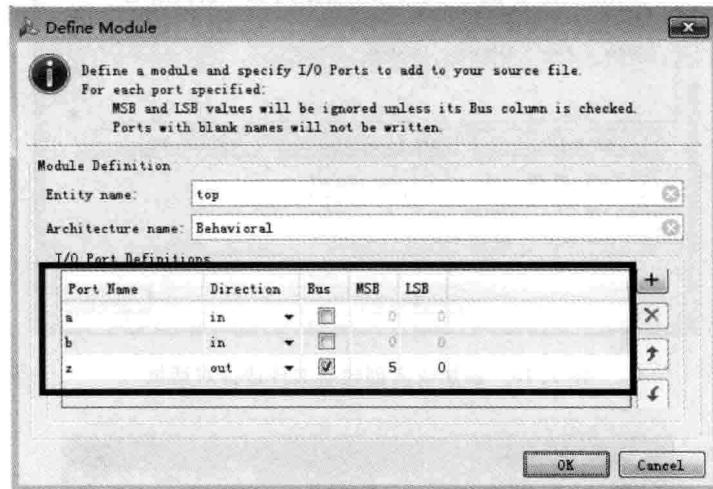


图 2.17 定义 VHDL 模块对话框界面

② Verilog HDL 定义模块对话框界面。

出现如图 2.18 所示的 Define Module(定义模块)对话框界面，按下面参数设置：

添加三个端口：a,b,z；

对于端口 a,Direction: input；

对于端口 b,Direction: input；

对于端口 z,Direction: output, 选中 Bus(总线)复选框, MSB: 5, LSB: 0。

**注：**该声明和 Verilog HDL 模块内的端口声明相对应。

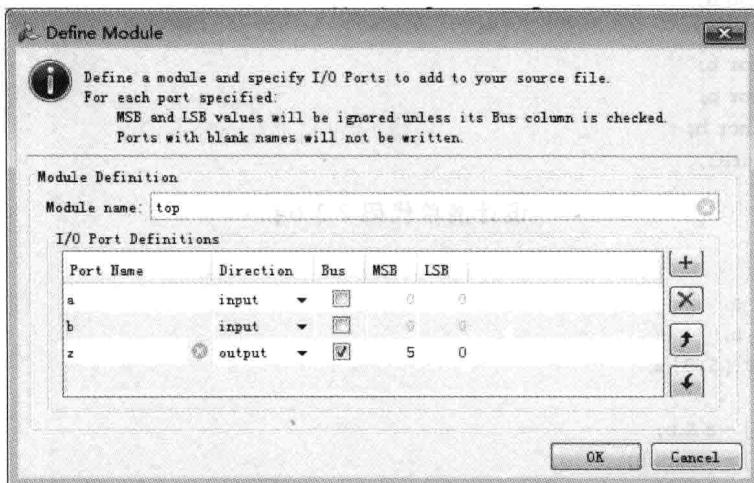


图 2.18 定义 Verilog 模块对话框界面

(10) 单击 OK 按钮。

(11) 如图 2.19 所示,在源文件窗口中,添加了 top.vhd 或者 top.v 文件。

(12) 双击源文件窗口的 top.vhd 文件或者 top.v 文件。打开设计模板,修改设计模板,并添加设计代码。该设计中,两个逻辑量 a 和 b,进行 6 种逻辑运算,产生 6 种输出到 z(5)~z(0)。

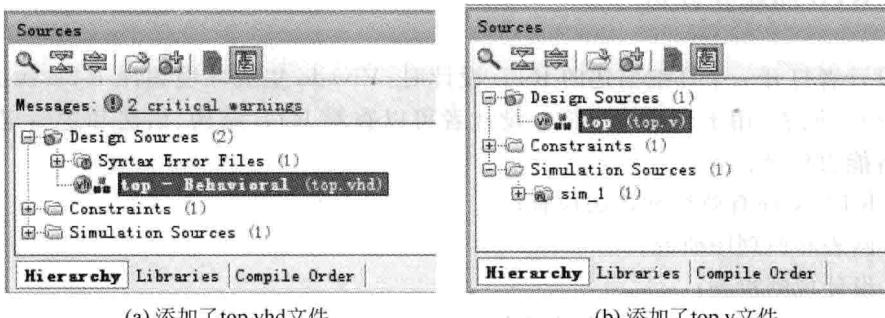


图 2.19 新添加了 top.vhd/top.v 文件

### 设计代码清单 2-1 top.vhd

```
-- entity top is
entity top is
    Port( a:in STD_LOGIC;
           b:in STD_LOGIC;
           z:out STD_LOGIC_VECTOR(5 downto 0);
    );
end top;

architecture Behavioral of top is
begin
    z(0)<= a and b;
```

```

z(1)<= a and b;
z(2)<= a or b;
z(3)<= a nor b;
z(4)<= a xor b;
z(5)<= a xnor b;
end Behavioral;

```

### 设计清单代码 2-2 top.v

```

module top(
    input a,
    input b,
    output [5:0] z
);
assign z[0] = a & b;
assign z[1] = ~ (a & b);
assign z[2] = a | b ;
assign z[3] = ~ (a| b);
assign z[4] = a ^b;
assign z[5] = a ~^b;
endmodule

```

(13) 添加完设计代码后,保存设计文件。

## 2.1.5 RTL 描述和分析

当设计者打开一个详细描述的 RTL 设计时,Vivado 集成环境编译 RTL 源文件,并且加载 RTL 网表,用于交互式分析。设计者可以查看 RTL 结构、语法和逻辑定义。分析和报告能力包括:

- (1) RTL 编译有效性和语法检查;
- (2) 网表和原理图研究;
- (3) 设计规则检查;
- (4) 使用一个 RTL 端口列表的早期 I/O 引脚规划;
- (5) 可以在一个视图中,选择一个对象,并且交叉检测其他视图中的一个对象。

RTL 描述和分析的步骤主要包括:

- (1) 在源窗口下,选择 top.vhd 或者 top.v 文件。
- (2) 如图 2.20 所示,在 Vivado 左侧的流程管理窗口,找到 RTL Analysis(RTL 分析),并展开。
- (3) 看到 Elaborated Design,并展开。在展开项中,选择并双击 Open Elaborated Design。
- (4) Vivado 开始运行 Elaborated Design。如图 2.21 所示,当运行完后,可以看到 Open Elaborated Design 标题变成了 Elaborated Design。

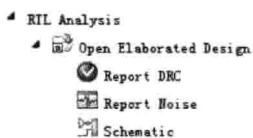


图 2.20 执行 RTL 分析

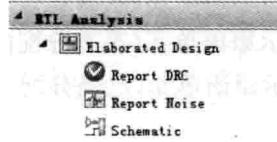


图 2.21 执行完 RTL 分析

(5) 如图 2.22 所示,自动打开 RTL Schematic。可以看到对 HDL 描述翻译进行后,得到的 RTL 级连接结构。

注:如果设计者重新运行 Elaborated Design,则在如图 2.21 所示的界面内,选择 Elaborated Design,单击右键,出现浮动菜单。在浮动菜单内,选择 Reload Design。

(6) 查看 RTL 级网表。如图 2.23 所示,在源窗口内,选择 RTL Netlist 标签。可以看到网表逻辑结构。

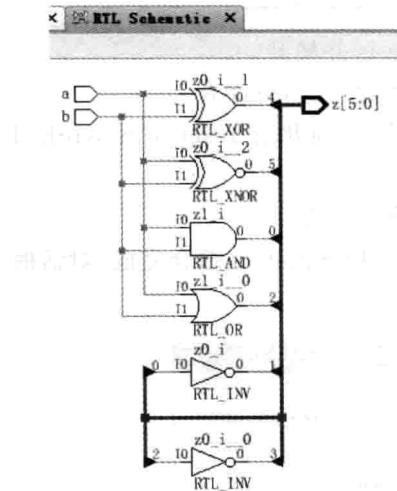


图 2.22 打开 RTL 原理图符号

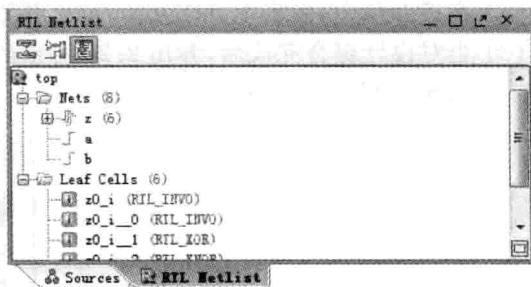


图 2.23 查看 RTL 级网表

下面对图标含义进行说明:

- ① : 表示总线;
- ② : 表示 IO 总线;
- ③ : 表示网络;
- ④ : 表示 IO 网络;
- ⑤ : 表示层次化单元(逻辑);
- ⑥ : 表示层次化单元(黑盒)。

注:对于那些不包含网表或者逻辑内容的层次化单元,Vivado 将理解为黑盒。一个层次化的单元可能是一个设计的黑盒,也可能是编码错误或者丢失文件。

- ⑦ : 表示层次化单元(分配到 Pblock)。
- ⑧ : 表示层次化单元(黑盒分配到 Pblock)。
- ⑨ : 表示原语单元(分配到 Pblock)。

- ⑩ 表示原语单元(放置并且分配到 Pblock)。
- ⑪ 表示原语单元(没有分配的布局约束)。
- ⑫ 表示原语单元(已经分配了布局约束)。

## 2.1.6 设计综合和分析

本节将对设计进行综合。综合就是将 RTL 级的设计描述转换成门级的描述。Vivado 集成环境下的综合工具基于时间驱动机制,专门为存储器的利用率和性能进行了优化。综合工具支持 SystemVerilog,以及 VHDL 和 Verilog 混合语言描述。该综合工具支持 Xilinx 设计约束 XDC(该约束是基于工业标准的 Synopsys 设计约束 SDC 格式)。

### 1. 执行设计综合

实现设计综合的步骤主要包括:

(1) 如图 2.24 所示,在流程处理窗口下,找到 Synthesis 并展开。

或者在 Tcl 命令行中,输入 launch\_runs synth\_1 脚本命令,运行综合。

**注:** 如果前面已经运行过综合,需要重新运行综合前,必须执行 reset\_run synth\_1 脚本命令,然后再执行 launch\_runs synth\_1 脚本命令。

(2) 在展开项中,单击 Run Synthesis。开始对设计进行综合。

(3) 当对设计综合完成后,弹出 2.25 所示的 Synthesis Completed(综合完成)对话框界面。该界面提供了三个选项:

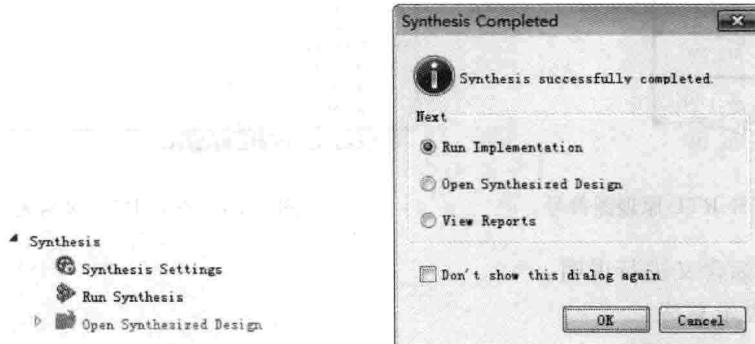


图 2.24 查找 Synthesis

图 2.25 综合完成提示对话框界面

- ① Run Implementation(运行实现过程);
- ② Open Synthesized Design(打开综合后的设计);
- ③ View Reports(查看报告)。

选择 Open Synthesized Design 选项。

- (4) 单击 OK 按钮。
- (5) 如图 2.26 所示,出现对话框,提示关闭前面执行 Elaborated Design 所打开的原理图界面,单击 Yes 按钮。Vivado 开始执行综合过程。
- (6) 当执行完综合后,如图 2.27 所示,可以展开 Synthesis Design。提供了下面的选项:

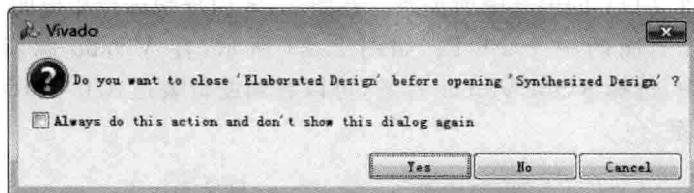


图 2.26 提示关闭前面界面对话框界面

- ① Edit Timing Constraints(编辑时序约束);
- ② Report Timing Summary(报告时序总结);
- ③ Report Clock Networks(报告时钟网络);
- ④ Report Clock Interaction(报告时钟相互作用);
- ⑤ Report DRC(报告 DRC);
- ⑥ Report Noise(报告噪声);
- ⑦ Report Utilization(报告利用率);
- ⑧ Report Power(报告功耗);
- ⑨ Schematic(原理图)。

下面给出综合后的报告利用率和原理图。

(7) 在图 2.27 所示的界面内,单击 Schematic 选项。

(8) 如图 2.28 所示,显示了该设计的综合后的网表结构。

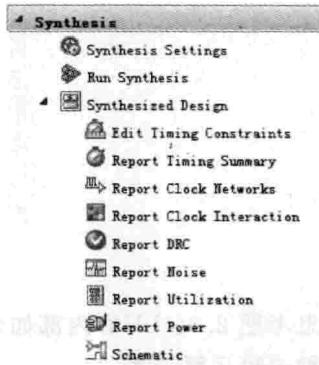


图 2.27 展开 Synthesis Design

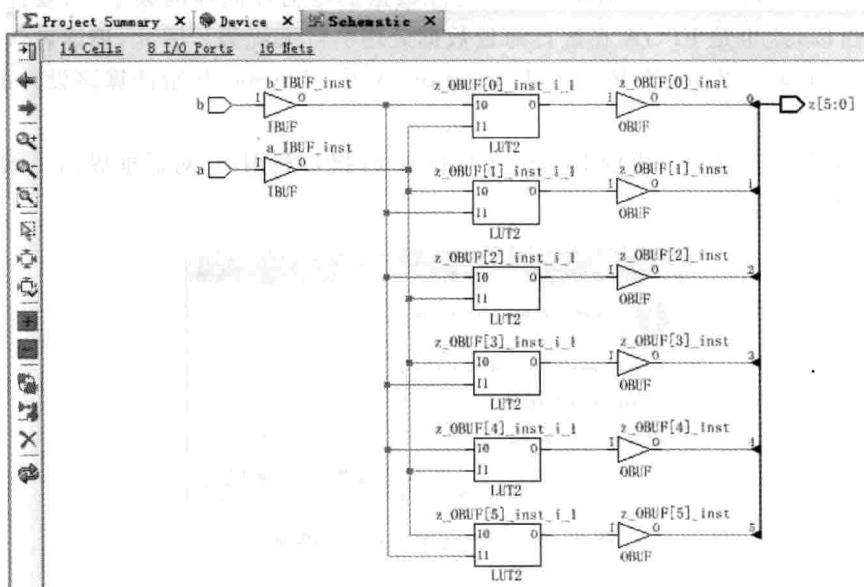


图 2.28 该设计的完整网表结构

思考题 2.1: 为什么是这种结构(提示: FPGA 是基于查找表结构)。

思考题 2.2: 观察输入缓冲区和输出缓冲区的结构并分析该设计的结构。

(9) 查看每个 LUT 的内部映射关系。在图 2.28 内分别选择相应的 LUT,总共 6 个 LUT。先选择最上面的一个 LUT。如图 2.29 所示,在 Vivado 源窗口下方的 Cell Properties 窗口中,选择 ROM Values 标签,可以看到逻辑表达式  $0=I_0 \& I_1$ 。图 2.29 给出真值表映射关系。

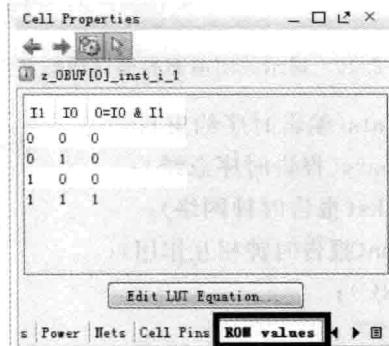


图 2.29 LUT 的内部映射关系

思考题 2.3: LUT 内部如何实现逻辑关系? (提示: 在每个查找表内实现了该设计的每种逻辑运算关系。)

思考题 2.4: 在前面的设计中,可以看到 Verilog 的描述方法,类似于 C 语言。C 语言通过按位逻辑运算符,实现逻辑运算。用 FPGA 实现和 C 语言在 CPU 上实现有什么本质的区别?(C 语言在 CPU 上实现,靠程序计数器,串行执行;而 FPGA 上的数字逻辑的实现是并行实现,是由逻辑流推动。)其处理数据的能力在同样的频率下,要比 CPU 快若干倍。所以,这也是 FPGA 在进行海量数据处理的巨大优势。因此,请读者认真体会。

(10) 单击图 2.27 内的 Report Utilization 选项。Vivado 开始计算该设计的资源消耗量。

(11) 如图 2.30 所示,出现 Report Utilization(报告利用率)对话框界面。默认地,报告名字是 utilization\_1。

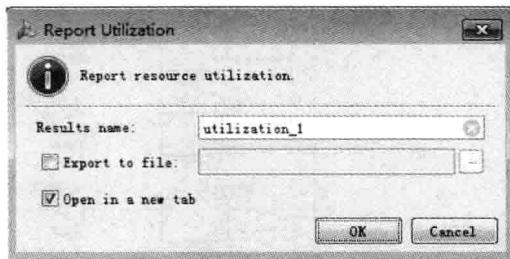


图 2.30 报告利用率对话框界面

(12) 单击 OK 按钮。

(13) 在 Vivado 下方打开 Utilization-utilization\_1 标签窗口。如图 2.31 所示,给出了该设计的资源利用率:

① Slice Logic(切片逻辑)使用了 3 个,总共 64300,利用率 1%;

② IO and GIX Specific(IO 和指定的 GIX)使用了 8 个,总共有 212 个,利用率 4%。

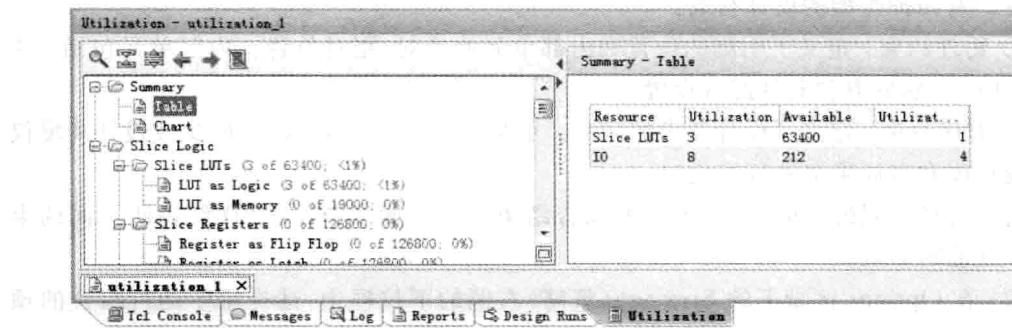


图 2.31 设计利用率报告

## 2. 设计综合选项

本节将介绍设置综合选项参数的含义。便于后面在修改综合选项参数时，理解这些参数的含义。在如图 2.24 所示的界面中，选择 Synthesis Settings。如图 2.32 所示，出现综合属性设置对话框界面。

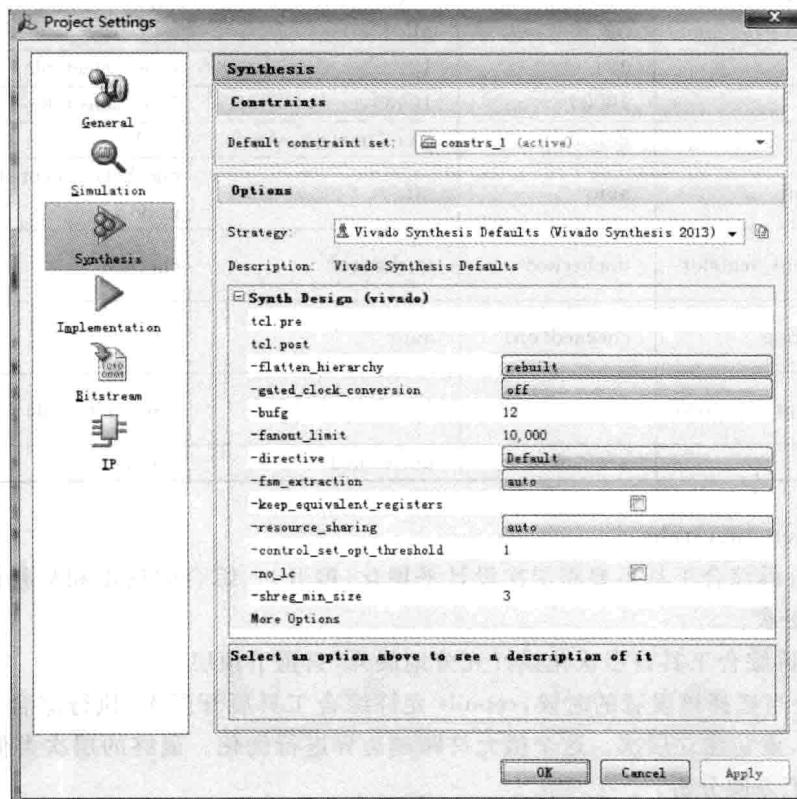


图 2.32 综合属性设置界面

- (1) 在 Default constraint set 右侧通过下拉框，可以选择用于综合的多个不同设计约束集合。一个约束集合是多个文件的集合，它包含 XDC 文件中用于该设计的设计约

束条件。有两种类型的设计约束：

① 物理约束：定义了引脚的位置和内部单元的绝对/相对位置。内部单元包括：块 RAM、LUT、触发器和器件配置设置。

② 时序约束：定义了设计要求的频率。如果没有时序约束，Vivado 集成设计环境仅对布线长度和布局阻塞进行优化。

通过选择不同的约束，得到不同的综合结果。在后面章节中，会详细说明不同约束对设计性能的影响。

(2) 在 Options 区域下的 Strategy(策略)右侧的下拉框中，选择用于运行综合的预定义综合策略。设计者也可以定义自己的策略。后面会介绍如何创建一个新的策略。表 2.1 给出了运行策略选项、默认设置和其他选项。

表 2.1 运行策略选项、默认设置和其他选项

运行策略选项	Vivado Synthesis-Defaults 默认设置	Flow_RuntimeOptimized 默认设置	其他选项
-flatten_hierarchy	rebuilt	none	full
-gated_clock_conversion	off	off	on
-bufg	12	12	User-selectable
-fanout_limit	10000	10000	User-selectable
-directive	default	RunTimeOptimized	N/A
-fsm_extraction	auto	off	one_hot, sequential, Johnson, gray
-keep_equivalent_registers	unchecked	unchecked	checked
-resource_sharing	checked(on)	auto	off
-control_set_opt_threshold	1	1	User-selectable
-no_lc	unchecked	unchecked	checked

① -flatten\_hierarchy：

none：告诉综合工具不要将层次设计平坦化(展开)。综合的输出和最初的 RTL 描述有相同的层次。

full：告诉综合工具将层次化设计充分地展开，只留下顶层。

rebuilt：当选择该设置的时候，rebuilt 允许综合工具展开层次，执行综合，然后基于最初的 RTL，重新建立层次。这个值允许跨越边界进行优化。最终的层次类似于 RTL，这主要是为了方便分析。

② -gate\_clock\_conversion：该选项打开或者关闭综合工具的能力。该能力提供了对带有使能时钟逻辑的转换。使用门控时钟转换也要求使用 RTL 属性。

③ -bufg：该选项控制综合工具推断设计中需要 BUFG 的个数。在网表内，当设计中使用的其他 BUFG 对综合过程不可见时，使用这个选项。

③ -bufg 后面的数字所定,工具所能推断出的 BUFG 个数。例如,如果-bufg 选项设置为最多 12 个,在 RTL 内例化了 3 个 BUFG,则工具还能推断出 9 个 BUFG。

④ -fanout\_limit: 指定在开始复制逻辑前,信号必须驱动的负载个数。这个目标限制,通常是引导性的。当工具确定必须复制逻辑时,就会忽略该选项。

**注:** 该选项不影响控制信号,例如置位、复位和时钟使能。如果需要的话,则使用 MAX\_FANOUT 来复制这些信号。

⑤ -directive: 代替 effort\_level 选项。当指定时,这个选项用不同的优化运行 Vivado 综合过程。它的值是 Default 和 RuntimeOptimized 时,更快地运行综合,进行较少的优化。

⑥ -fsm\_extraction: 该选项控制如何提取和映射有限自动状态机。当该选项为 off 时,将状态机综合为逻辑。此外,设计者也可以从下面的选项中指定状态机的编码类型。即: one\_hot、sequential、johnson、gray 或者 auto。

⑦ -keep\_equivalent\_registers: 该选项将阻止合并带有相同逻辑输入的寄存器。

⑧ -resource\_sharing: 该选项用于在不同的信号间共享算术操作符。可选的值为 auto、on 和 off。

**注:** 选择 auto 时,取决于设计要求的时序,决定是否采用资源共享; on 表示总是进行资源共享,off 表示总是关闭资源共享。

⑨ -control\_set\_opt\_threshold: 该选项设置用于时钟使能优化的门限,目的在于降低控制设置的个数。默认地,该值设置为 1。给定的值是扇出的个数,Vivado 工具将把这些控制设置移动到一个 D 触发器的逻辑中。如果扇出比这个值多,则工具尝试让信号驱动寄存器上的 control\_set\_pin。

⑩ -no\_lc: 当选中的时候,这个选项将关闭 LUT 的组合。

**注:** 可以使用 KEEP 属性,阻止对寄存器地合并。

(3) tcl.pre 和 tcl.post,这两个选项和 tcl 文件关联,用于综合前和综合后地立即运行。

**注:**

① tcl.pre 和 tcl.post 脚本中的路径是相对于当前工程的路径:

```
<project>/<project.runs>/<run_name>
```

② 可以使用当前工程或者当前运行的 DIRECTORY 属性来定义脚本中的相对路径:

```
get_property DIRECTOTY [current_project]
get_property DIRECTOTY [current_run]
```

## 2.1.7 设计行为级仿真

本节将执行对设计的行为级仿真。执行对设计进行行为级仿真的步骤主要包含:

(1) 按照前面的操作方法,启动添加新文件命令。也可以在源文件窗口内选择 Simulation Sources,单击右键,出现浮动菜单。在浮动菜单内,选择 Edit Simulation Sets...。

(2) 如图 2.14 所示, 出现 Add Source(添加源文件)对话框界面。在该界面内选择 Add or Create Simulation Source(添加或者创建仿真源文件)选项。

(3) 单击 Next 按钮。

(4) 出现 Add or Create Simulation Sources(添加或者创建仿真源文件)对话框界面。在该界面内, 单击 Create File 按钮。

(5) 下面分 VHDL/Verilog 两种设计语言介绍:

① 对于 VHDL 设计流程, 如图 2.33 所示, 按下面参数设置:

File type: VHDL;

File name: test;

File location: Local to Project。

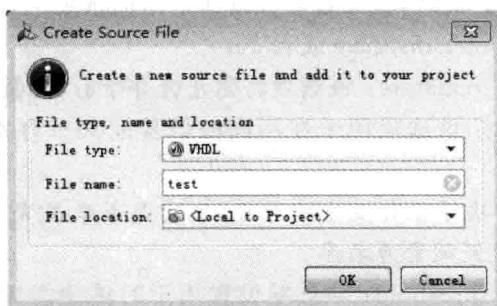


图 2.33 添加 VHDL 类型的仿真文件

② 对于 Verilog 设计流程, 如图 2.34 所示, 按下面参数设置:

File type: Verilog;

File name: test;

File location: Local to Project。

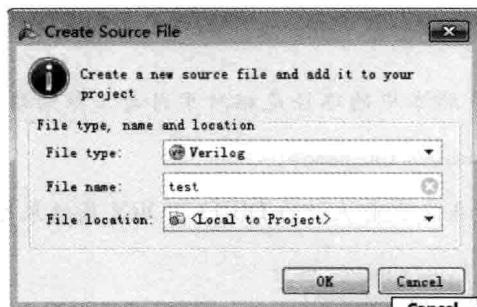


图 2.34 添加 Verilog 类型的仿真文件

(6) 单击 OK 按钮。

(7) 在 Add Sources 对话框界面中, 新添加了名字为 test.vhd/test.v 的仿真源文件。

(8) 在 Add Sources 对话框界面中, 单击 Finish 按钮。

(9) 出现 Define Module(定义模块)对话框界面。直接单击 OK 按钮。

(10) 出现 Define Module(定义模块)提示对话框界面。直接单击 Yes 按钮。

(11) 如图 2.35 所示,在源文件窗口的 simulation Sources 下添加了 test.vhd 或者 test.v 文件,它们下面包含设计文件 top.vhd 或者 top.v,它们分别作为仿真测试文件的设计源文件。

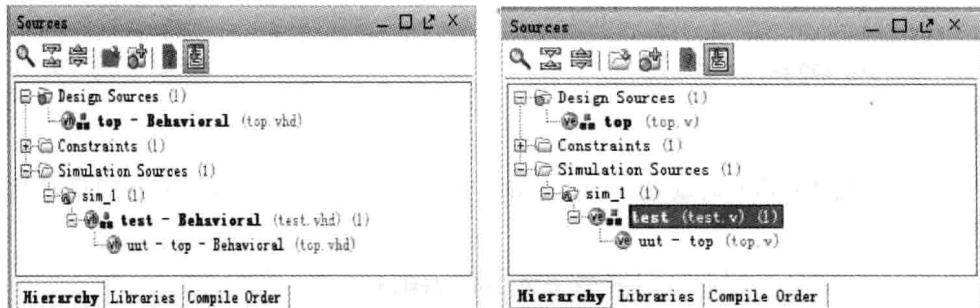


图 2.35 添加 HDL 仿真文件后的源窗口

(12) 按照 VHDL/Verilog 语法添加测试代码。

① 对于使用 VHDL 的读者,打开 test.vhd 文件。在 test.vhd 中添加下面的设计代码,作为仿真源文件。

#### 代码清单 2-3 test.vhd

```
entity test is
end test;

architecture Behavioral of test is
component top
    Port ( a : in STD_LOGIC;
           b : in STD_LOGIC;
           z : out STD_LOGIC_VECTOR (5 downto 0)
    );
end component;
signal a : std_logic := '0';
signal b : std_logic := '0';
signal z : std_logic_vector(5 downto 0);
begin
    uut : top port map(
        a => a,
        b => b,
        z => z
    );
process
begin
    a <= '0';
    b <= '0';
    wait for 200 ns;
    a <= '0';
    b <= '1';
end process;
end;
```

```

    wait for 200 ns;
    a<='1';
    b<='0';
    wait for 200 ns;
    a<='1';
    b<='1';
    wait for 200 ns;
end process;
end;

```

② 对于使用 Verilog 的读者, 打开 test.v 文件。在 test.v 中添加下面的设计代码。作为仿真源文件。

**代码清单 2-4 test.v**

```

'timescale 1ns / 1ps

module test;
reg a;
reg b;
wire [5:0] z;
top uut(
    .a(a),
    .b(b),
    .z(z)
);
initial
begin
    while(1)
        begin
            a = 0;
            b = 0;
            #100;
            a = 0;
            b = 1;
            #100;
            a = 1;
            b = 0;
            #100;
            a = 1;
            b = 1;
            #100;
        end
    end
endmodule

```

- (13) 保存 test.vhd 或者 test.v 文件。
- (14) 在源文件窗口选择 test.vhd 或者 test.v 文件(取决于读者所选择的设计语言)。
- (15) 如图 2.36 所示, 在 Vivado 设计界面左侧的流程向导窗口内, 找到并展开 Simulation。

(16) 单击 Run Simulation(运行仿真)。出现浮动菜单,选择 Run Behavioral Simulation(运行行为仿真选项)。此后,Vivado 开始运行行为仿真。

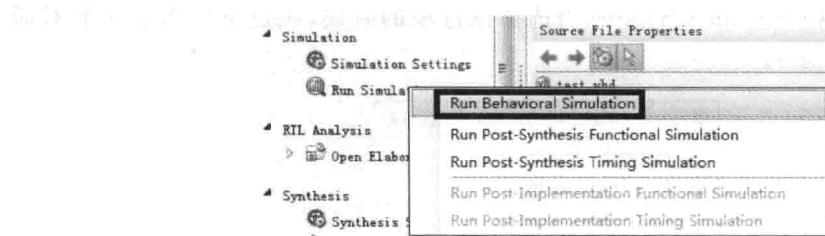


图 2.36 选择运行行为仿真

(17) 如图 2.37 所示,出现行为仿真波形界面。

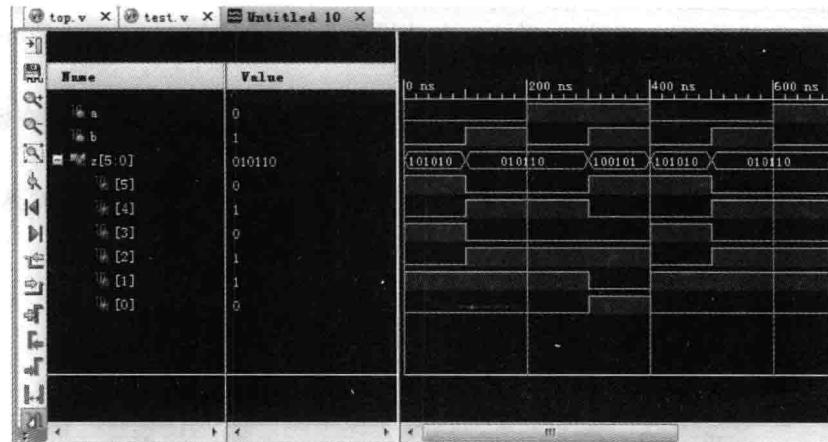


图 2.37 行为仿真波形图

注:

- ① 单击图 2.37 左侧一列工具栏中的 (放大)或者 (缩小)按钮,将波形调整到合适的显示大小。
- ② 单击工具栏中的 按钮,添加若干标尺,使得可以测量某两个逻辑信号跳变之间的时间间隔。
- ③ 如图 2.38 所示,单击 Vivado 上方的工具栏内的按钮,可以控制仿真的运行时间。

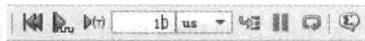


图 2.38 控制仿真运行时间

(18) 退出行为仿真界面。

## 2.1.8 添加约束条件

本节将为设计指定引脚约束,该引脚约束参考 Nexys4 板子的设计图纸和相关资料。使用 Nexys4 的两个开关作为 a 和 b 的逻辑输入量,6 个 LED 灯作为 z0~z5 的逻辑输出

量。添加约束条件的步骤包括：

(1) 在 Vivado 源文件窗口下,单击右键,如图 2.39 所示,出现浮动菜单。选择 Add Sources...,或者 Edit Constraints Sets... (当选择该选项的时候,跳过下面第 2 步的对话框界面,直接进入第 3 步)。

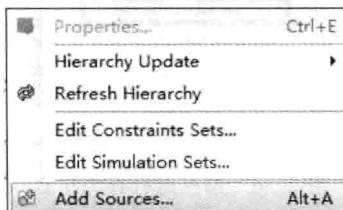


图 2.39 添加约束文件选项

(2) 出现 Add Sources 对话框界面。在该界面下,选择 Add or Create Constraints。

(3) 单击 Next 按钮。

(4) 出现 Add or Create Constraints(添加或者创建约束)对话框界面。在该界面下,单击 Create File...按钮。

(5) 如图 2.40 所示,出现 Create Constraints File(创建约束文件)对话框界面。按下面参数设置:

- ① File type: XDC;
- ② File name: top;
- ③ File location: Local to Project。

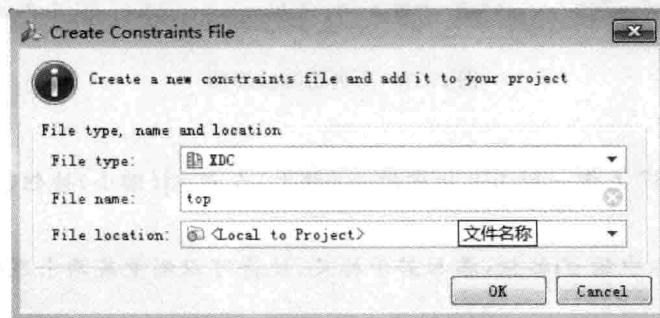


图 2.40 添加约束文件对话框界面

(6) 单击 OK 按钮,返回到添加或者创建约束对话框界面。可以看到添加了名字为 top.XDC 的约束文件。

**注:** 在 Vivado 中,约束文件的后缀名是. xdc,该文件取代 ISE 集成设计环境中的 .ucf 文件。

(7) 单击 Finish 按钮。如图 2.41 所示,可以看到添加了 top.xdc 文件。

(8) 如图 2.42 所示,在下拉框中选择 I/O Planning(I/O 规划)。

**注:** 需要执行完前面的综合,并且在综合完后,选择 Open Synthesized Design。

(9) 如图 2.43 所示,在 Vivado 下方出现 I/O Ports(I/O 端口)对话框界面。按

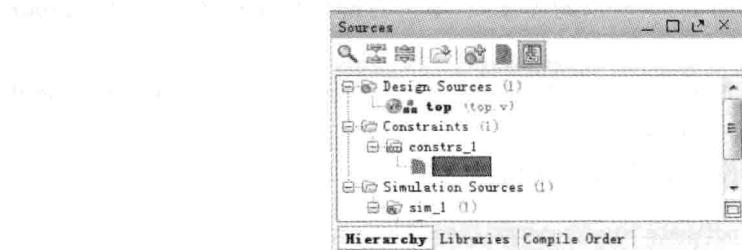


图 2.41 添加约束文件后的源文件窗口

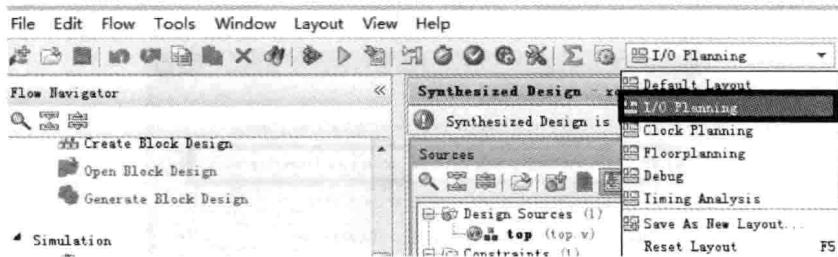


图 2.42 选择 I/O Planning

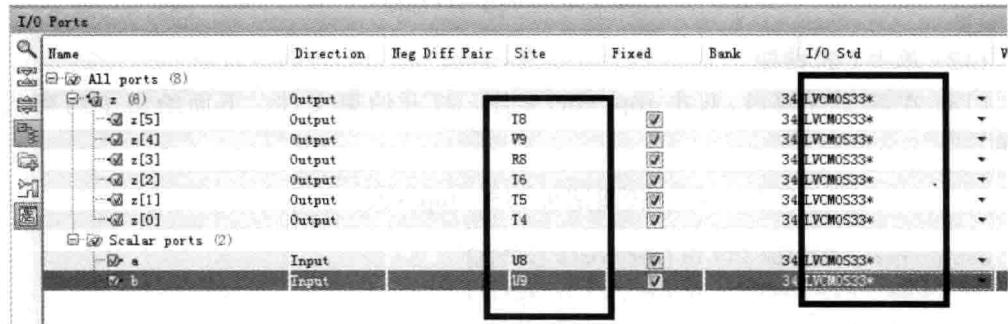


图 2.43 进行 I/O 规划

表 2.2 所示,在 Site 下输入设计中所定义的每个逻辑端口在 FPGA 上引脚的位置。此外,在 I/O Std(I/O 标准)下,为每个逻辑端口定义其 I/O 电气标准。

表 2.2 逻辑端口的 I/O 约束

逻辑端口	FPGA 引脚位置	I/O 标准
Z[5]	T8	LVCMOS33
Z[4]	V9	LVCMOS33
Z[3]	R8	LVCMOS33
Z[2]	T6	LVCMOS33
Z[1]	T5	LVCMOS33
Z[0]	T4	LVCMOS33
a	U8	LVCMOS33
b	U9	LVCMOS33

(10) 完成输入后，在如图 2.43 所示的界面内，单击右键出现浮动菜单，选择 Export I/O Ports…。

(11) 如图 2.44 所示，出现 Export I/O Ports(导出 I/O 端口)对话框界面。选中 XDC，并将导出路径指向当前工程路径。

① 对于 VHDL 语言的设计流程，路径指向：

E:\vivado\_example\gate\_vhdl\gate\_srcs\constrs\_1\new\top.xdc

② 对于 Verilog 语言的设计流程，路径指向：

E:\vivado\_example\gate\_verilog\gate\_verilog\_srcs\constrs\_1\new\top.xdc

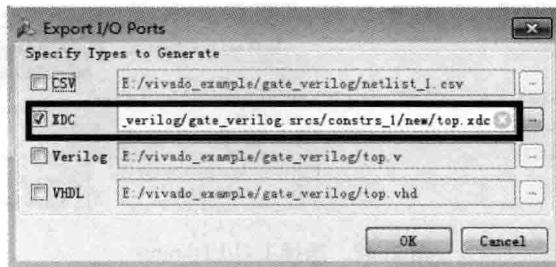


图 2.44 导出 I/O 规划

(12) 单击 OK 按钮。

(13) 在源文件窗口，双击 top.xdc 文件。打开约束文件。下面给出了约束代码清单。

#### 代码清单 2.5 top.xdc

```
set_property PACKAGE_PIN T8 [get_ports {z[5]}]
set_property PACKAGE_PIN V9 [get_ports {z[4]}]
set_property PACKAGE_PIN R8 [get_ports {z[3]}]
set_property PACKAGE_PIN T6 [get_ports {z[2]}]
set_property PACKAGE_PIN T5 [get_ports {z[1]}]
set_property PACKAGE_PIN T4 [get_ports {z[0]}]
set_property PACKAGE_PIN U8 [get_ports a]
set_property PACKAGE_PIN U9 [get_ports b]
set_property DIRECTION OUT [get_ports {z[5]}]
set_property IOSTANDARD LVCMOS33 [get_ports {z[5]}]
set_property DRIVE 12 [get_ports {z[5]}]
set_property SLEW SLOW [get_ports {z[5]}]
set_property DIRECTION OUT [get_ports {z[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {z[4]}]
set_property DRIVE 12 [get_ports {z[4]}]
set_property SLEW SLOW [get_ports {z[4]}]
set_property DIRECTION OUT [get_ports {z[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {z[3]}]
set_property DRIVE 12 [get_ports {z[3]}]
set_property SLEW SLOW [get_ports {z[3]}]
set_property DIRECTION OUT [get_ports {z[2]}]
```

```

set_property IOSTANDARD LVCMOS33 [get_ports {z[2]}]
set_property DRIVE 12 [get_ports {z[2]}]
set_property SLEW SLOW [get_ports {z[2]}]
set_property DIRECTION OUT [get_ports {z[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {z[1]}]
set_property DRIVE 12 [get_ports {z[1]}]
set_property SLEW SLOW [get_ports {z[1]}]
set_property DIRECTION OUT [get_ports {z[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {z[0]}]
set_property DRIVE 12 [get_ports {z[0]}]
set_property SLEW SLOW [get_ports {z[0]}]
set_property DIRECTION IN [get_ports a]
set_property IOSTANDARD LVCMOS33 [get_ports a]
set_property DIRECTION IN [get_ports b]
set_property IOSTANDARD LVCMOS33 [get_ports b]

```

注：下面一节将对 XDC 的语法规则进行简单的说明。

(14) 如图 2.43 所示，将选项再次切换到 Default Layout。

### 2.1.9 XDC 约束语法规则

本节将介绍 XDC 约束相关的语法规则。内容包括 XDC 和 UCF 的区别、UCF 到 XDC 映射、约束顺序。

#### 1. XDC 和 UCF 的区别

XDC 和 UCF 约束的区别主要包括：

- (1) XDC 是顺序语言，它是一个带有明确优先级的规则。
- (2) 一般来说，UCF 应用于网络，而 XDC 可以应用到引脚、端口和单元对象(Cell Object)。
- (3) UCF 的 PERIOD 约束和 XDC 的 create\_clock 命令并不等效，这将导致不同的时序结果。
- (4) 默认地，对于 UCF 来说，在异步时钟组之间无时序关系；但是，对于 XDC 来说，所有时钟之间都存在联系，也就是存在时序关系(除非有其他约束)(set\_clock\_groups)。
- (5) 在 XDC 中，相同的对象中存在多个时钟。

#### 2. UCF 到 XDC 映射

表 2.3 给出 UCF 约束到 XDC 命令之间的映射。

表 2.3 UCF 约束到 XDC 命令之间的映射

UCF	XDC
TIMESPEC PERIOD	create_clock create_generated_clock
OFFSET = IN <x> BEFORE <clk>	set_input_delay
OFFSET = OUT <x> BEFORE <clk>	set_output_delay

续表

UCF	XDC
FROM:TO "TS_}*2	set_multicycle_path
FROM:TO	set_max_delay
TIG	set_false_path
NET "clk_p" LOC = AD12	set_property LOC AD12 [get_ports clk_p]
NET "clk_p" IOSTANDARD = LVDS	set_property IOSTANDARD LVDS [get_ports clk_p]

注：不管设计者在设计中，使用了一个还是多个 XDC 文件，Xilinx 推荐设计者使用下面的顺序来组织约束：

```

## Timing Assertions Section
# Primary clocks
# Virtual clocks
# Generated clocks
# Clock Groups
# Input and output delay constraints

## Timing Exceptions Section (sorted by precedence)
# False Paths
# Max Delay / Min Delay
# Multicycle Paths
# Case Analysis
# Disable Timing

## Physical Constraints Section
# located anywhere in the file, preferably before or after the timing
constraints
# or stored in a separate XDC file

```

## 2.1.10 设计实现和分析

Vivado 集成设计环境的实现处理过程，包括：对设计的逻辑和物理转换。该实现过程包含下面子过程：

- (1) 优化设计(Opt Design)：对逻辑设计进行优化，使其容易适配到目标 Xilinx 器件。
- (2) 功耗优化设计(Power Opt Design)：对设计元素进行优化，以降低目标 Xilinx 器件的功耗要求。
- 注：该步是可选的。
- (3) 布局设计(Place Design)：在目标 Xilinx 器件上，对设计进行布局。
- (4) 物理优化设计(Phys Opt Design)：对高扇出网络驱动器进行复制，对其负载进行分配(即降低将高扇出负载量)，以优化设计时序。
- (5) 布线设计(Route Design)：在目标 Xilinx 器件上对设计进行布线。
- (6) 写比特流(Write Bitstream)：生成用于配置 Xilinx 器件的比特流。

## 1. 运行设计实现

本节将执行设计实现过程。设计实现的步骤主要包括：

(1) 在源窗口中,选择 top.vhd/top.v 文件(读者根据自己设计所使用的 HDL 语言选择)。

(2) 如图 2.45 所示,在 Vivado 左侧的流程处理窗口下,找到并展开 Implementation。

(3) 在展开项里,单击 Run Implementation 选项。Vivado 开始执行设计实现过程;或者在 Tcl 命令行中,输入 launch\_runs impl\_1 脚本命令,运行实现。

**注:** 如果前面已经运行过实现,需要重新运行实现之前,必须执行 reset\_run impl\_1 脚本命令,然后再执行 launch\_runs impl\_1 脚本命令。

(4) 如图 2.46 所示,出现 Bitstream Generation Completed(比特流生成完成)对话框界面。在该界面内提供了三个选项:

- ① Open Implemented Design(打开实现后的设计);
- ② View Reports(查看报告);
- ③ Launch iMPACT(启动 iMPACT)。

在该设计中,选择 Open Implemented Design 选项。

(5) 单击 OK 按钮。  
 (6) 出现 Critical Messages(危险消息)对话框界面。在该设计中不用考虑。  
 (7) 单击 OK 按钮。  
 (8) 如图 2.47 所示,在 Vivado 右上角出现器件的结构图。

(9) 单击图 2.47 界面左侧一列工具栏内的 (放大视图)按钮,放大该器件视图。并调整视图在窗口的位置。



图 2.46 实现处理完成后的对话框界面

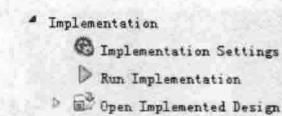


图 2.45 执行实现处理过程

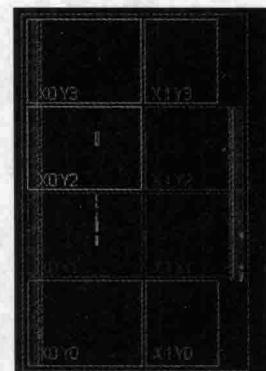


图 2.47 Artix FPGA 器件内部结构图

(10) 如图 2.48 所示,在器件右下角视图上,看到标有橙色颜色方块的引脚。表示该设计中已经使用。

(11) 单击图 2.47 界面左侧一列工具栏内的 (显示布线资源)按钮和 (缩小视

图)按钮。并调整视图在窗口的位置。

(12) 如图 2.49 所示,显示了该设计的布线。

注: 其中绿色的线表示设计中使用的互连线资源。

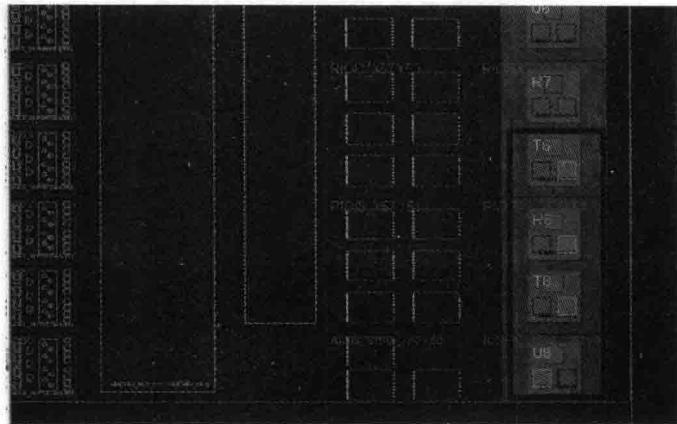


图 2.48 Artix FPGA 器件所用 IO 引脚



图 2.49 该设计的布线关系

(13) 放大视图,并调整该视图在窗口的位置。如图 2.49 所示,给出了该设计所使用的逻辑设计资源。在该视图中,可以看到其内部的结构,包括: 查找表 LUT、多路复用器 MUX、快速进位链、触发器资源。

思考题 2.5: 结合作者《Xilinx FPGA 设计权威指南》一书中,对 Xilinx 7 系列 FPGA 内部结构的介绍。仔细查找时钟管理单元、块存储器、数字信号处理模块,以及其他设计资源在 FPGA 硅片上的布局。

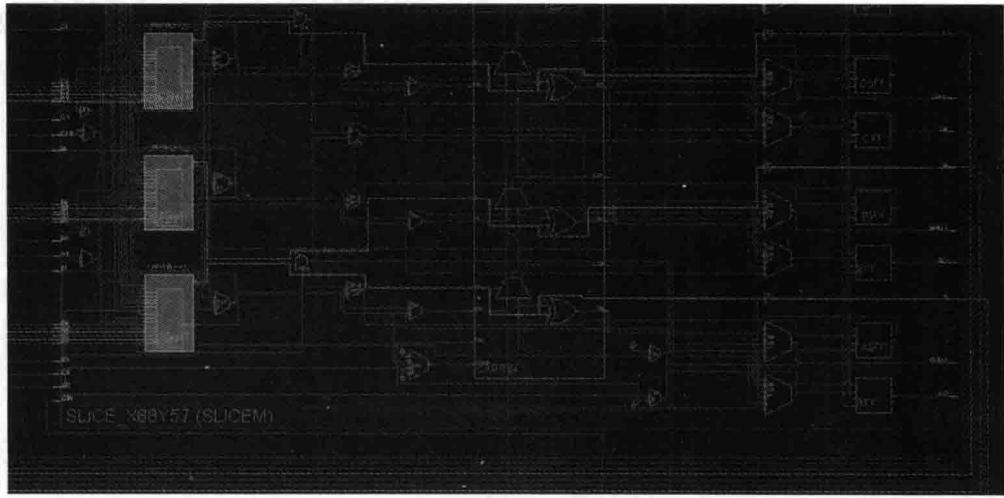


图 2.50 该设计使用的逻辑资源

思考题 2.6: 把鼠标放到每个逻辑资源上,查看 Xilinx 对每个逻辑资源的位置定义。这对于读者进行相应的位置约束也至关重要。

(14) 关闭器件视图界面。

## 2. 设计实现选项

本节将介绍设计实现选项参数,以便在后面对设计实现选项进行修改时,能理解这些参数的含义。如图 2.45 所示,单击 Implementation Settings 选项。打开图 2.51 所示的实现选项对话框界面。

由图 2.51 中可以看出,Vivado 集成设计套件包含预定义的策略集。此外,设计者也可以创建自己的策略(后面章节会详细介绍)。

根据策略的目的,将其分解为不同的类。类的名字作为策略的前缀。表 2.4 给出了类及其用途。表 2.5 给出了实现策略的种类和功能描述。

表 2.4 类及其用途

类 别	目 的
Performance	提高设计性能
Area	减少 LUT 个数
Power	添加整体功耗优化
Flow	修改流程步骤
Congestion	减少阻塞和相关的问题

表 2.5 实现策略(Implementation Strategies)

实现策略名字	描 述
Vivado Implementation Defaults	平衡运行时间,努力实现时序收敛
Performance_Explore	使用多个算法进行优化、布局和布线,这是为了得到潜在较好的优化结果
Performance_RefinePlacement	在布局后的优化阶段内,提高布局器的努力程度,防止在布线器内出现时序发散(时序的不收敛)
Performance_WLBlockPlacement	忽略用于布局 BRAM 和 DSP 的时序约束,取而代之的是使用线长
Performance_WLBlockPlacementFanoutOpt	忽略用于布局 BRAM 和 DSP 的时序约束,取而代之的是使用线长。并且,执行对高扇出驱动器的复制
Performance_LateBlockPlacement	使用大概的 BRAM 和 DSP 的布局,直到布局的后期阶段。因此可能产生更好的整体布局
Performance_NetDelay_high	补偿乐观的延迟估计。为长距离和高扇出的连接,添加额外的延迟代价(high 设置,最悲观的)
Performance_NetDelay_medium	补偿乐观的延迟估计。为长距离和高扇出的连接,添加额外的延迟代价(medium 设置)
Performance_NetDelay_low	补偿乐观的延迟估计。为长距离和高扇出的连接,添加额外的延迟代价(low 设置,最不悲观的)
Performance_ExploreSLLs	探索 SLR 的重新分配,以改善整体的时序余量
Area_Explore	使用多个优化算法,以使用更少的 LUT
Area_DefalutOpt	添加功耗优化(power_opt_design),以降低功耗

续表

实现策略名字	描述
Area_RunPhysOpt	类似于 Implementation Run Defaults,但是使能物理优化步骤 (phys_opt_design)
Flow_RuntimeOptimized	每个实现步骤用设计性能换取了更好的设计时间,禁止物理优化 (phys_opt_design)
Flow_Quick	只运行布局和布线,禁止所有的优化和时间驱动的行为
Congestion_SpreadLogic_high	将逻辑分散到整个器件,以避免创建阻塞区域。High 表示最高程度的分散
Congestion_SpreadLogic_medium	将逻辑分散到整个器件,以避免创建阻塞区域。Medium 表示中等程度的分散
Congestion_SpreadLogic_low	将逻辑分散到整个器件,以避免创建阻塞区域。low 表示最低程度的分散
Congestion_SpreadLogicSLLs	分配 SLL,这样所有逻辑能分配到所有的 SLR,以避免在 SLR 内创建阻塞区域
Congestion_BalanceSLLs	分配 SLL,这样没有两个 SLR 要求一个不成比例的较多的 SLL,因此减少了那些 SLR 内的阻塞。
Congestion_BalanceSLRs	分区,这样每个 SLR 有相似的区域,以避免在一个 SLR 内创建阻塞区域
Congestion_CompressSLRs	用较高 SLR 利用率分区,以降低整体 SLL 的数量

注: 包含 SLL 或者 SLR 的策略,只用于 SSI 器件。

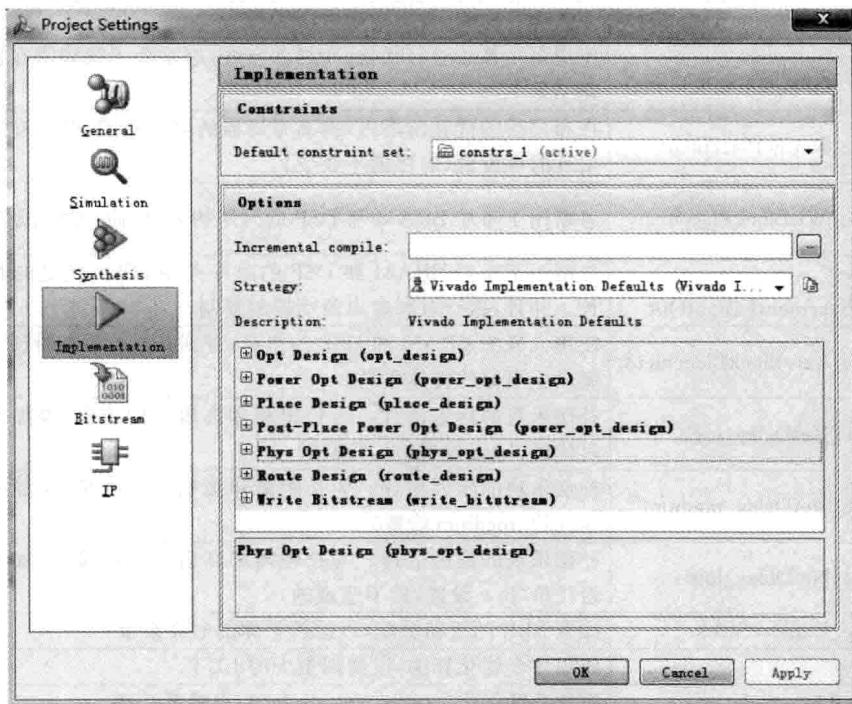


图 2.51 实现设计设置对话框界面

## 2.1.11 设计时序仿真

本节将对设计执行时序仿真。时序仿真和行为级仿真最大的不同点在于，时序仿真带有标准延迟格式(Standard Delay Format, SDF)信息，而行为级仿真不带有时序信息。毛刺、竞争冒险等时序问题都会表现在设计时序仿真中。执行时序仿真的步骤主要包含：

- (1) 在源文件窗口选择 test.vhd/test.v 文件(取决于读者所选择的设计语言)。
- (2) 在 Vivado 设计界面左侧的 Flow Explorer 内，找到 Simulation 并展开。
- (3) 如图 2.52 所示，单击 Run Simulation(运行仿真)。出现浮动菜单，选择 Run Post-Implementation Timing Simulation(运行实现后时序仿真选项)。在浮动菜单内，Vivado 开始运行实现后仿真程序。

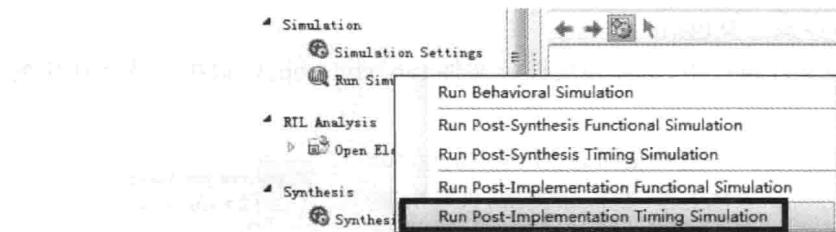


图 2.52 选择实现后时序仿真

- (4) 如图 2.53 所示，出现仿真后的波形界面。在其左侧一列的工具栏中，单击 (放大)按钮，对仿真波形局部放大，并且调整仿真波形在窗口中的位置。在进行适当调整后，出现实现后的仿真波形界面。

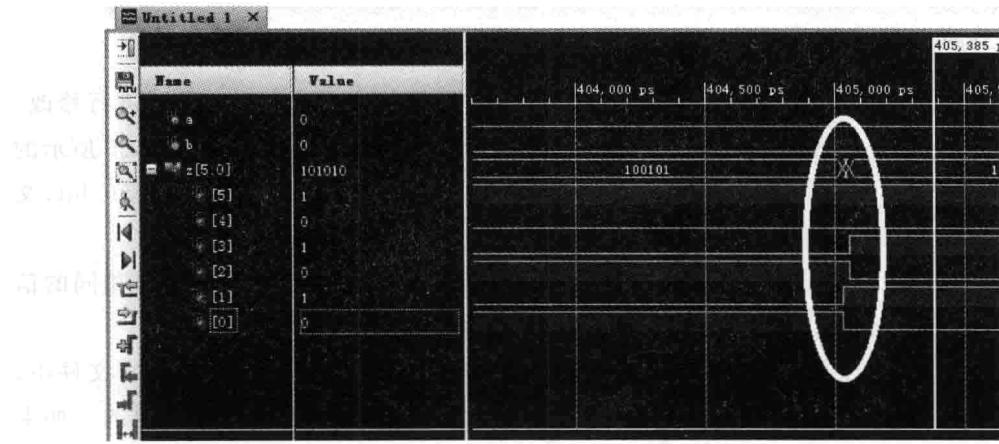


图 2.53 实现后时序仿真波形图

思考题 2.7：请读者仔细观察图中白色椭圆圈内的信号变化情况。说明毛刺的产生机理。

(提示：不同的逻辑量从输入到输出经过芯片内互连线传输延迟和逻辑门的翻转延

迟。这些延迟时间是不一样的,从图中看到延迟是 ps。所以,可以看到 z[5:0]逻辑信号上有一个很小的过渡区域,也就是毛刺)。

思考题 2.8: 查阅相关资料,说明在 FPGA 内关键路径的定义,以及为什么关键路径对设计性能有很大的影响。在设计中,采取何种措施缩短关键路径,以提高设计性能。

(5) 退出实现后仿真界面。

## 2.1.12 生成编程文件

编程文件用于对 FPGA 进行配置。通过调试主机和目标 FPGA 之间的 JTAG 通道,将编程文件下载到目标 FPGA 中。

### 1. 执行生成可编程文件

生成编程文件的步骤主要包括:

(1) 在 Vivado 源文件窗口中,选择顶层设计文件 top.vhd/top.v(取决于读者使用的设计流程)。

(2) 如图 2.54 所示,在 Vivado 左侧的 Flow Navigator 窗口下方,找到 Program and Debug(编程和调试)选项并展开。

(3) 在展开项中,单击 Generate Bitstream(生成比特流)选项。开始生成编程文件。

(4) 当生成编程文件后,准备将生成的编程文件下载到硬件中,对设计进行验证。



图 2.54 生成编程文件选项

### 2. 生成编程文件选项

本节将介绍生成编程文件的选项的功能,以便读者在后续对比特流设置进行修改。如图 2.54 所示,在该界面内选择并单击 Bitstream Settings 选项。打开如图 2.55 所示的比特流配置界面。默认地,write\_bitstream Tcl 命令只生成一个二进制比特流(.bit)文件。通过使用下面的命令开关来改变产生的文件格式。

(1) -raw\_bitfile: 该选项产生原始比特文件,该文件包含和二进制比特流相同的信息,但它是 ASCII 格式。输出文件名字为文件名.rbt。

(2) -mask\_file: 该选项产生一个掩码文件,有掩码数据,其配置数据在比特文件中。这个文件定义了比特流文件中的哪一位应该和回读数据进行比较,用于验证目的。如果掩码为 0,需要验证比特流中的该位,否则,不需要验证。输出文件的名字是文件名.msk。

(3) -no\_binary\_bitfile: 不产生二进制比特流文件。当需要生成 ASCII 格式比特流或者掩码文件时,或者生成一个比特流文件时,使用该选项。

(4) -logic\_location\_file: 创建一个 ASCII 逻辑定位文件(.ll),该文件给出了锁存器、

LUT、BRAM 和 I/O 块输入和输出的比特流位置。帧参考比特，位置文件中的比特个数，用于帮助设计者观察 FPGA 寄存器的内容。

(5) -bin\_file：创建一个二进制文件(. bin)，只包含器件编程数据，而没有标准比特流文件中的头部信息。

(6) -reference\_bitfile<arg>：读一个参考比特文件，输出一个增量比特流文件。该文件只包含与指定参考文件不同之处。这个部分比特流文件用于增量编程一个已经存在编程文件的器件，也就是更新。

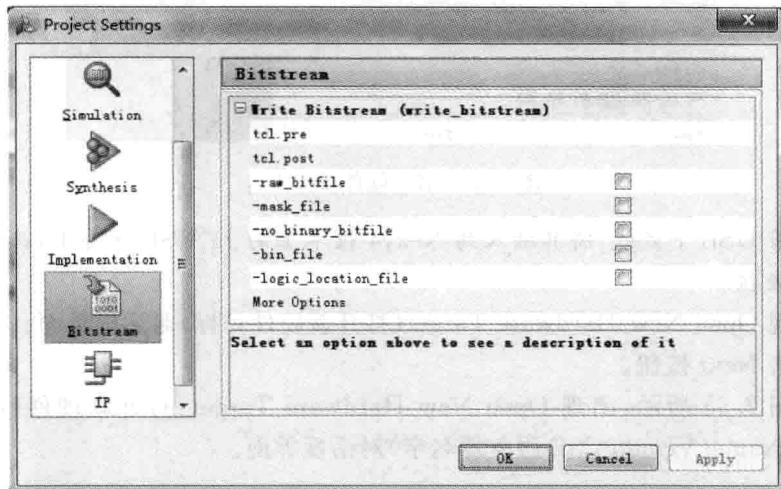


图 2.55 生成编程文件选项

### 2.1.13 下载比特流文件到 FPGA

当生成用于编程 FPGA 的比特流数据后，就可以将比特流数据下载到目标 FPGA 器件中。Vivado 工具提供了两种不同的方法，用于将比特流下载到目标 FPGA 中，即：

(1) 打开 Vivado 集成环境内构建的系统内器件编程功能-hardware session(硬件会话)；

(2) 打开 Vivado 集成环境内提供的 iMPACT 器件编程器工具。

**注：**iMPACT 作为 ISE 环境下的下载工具，本书不建议在 Vivado 集成环境下使用。

Vivado 集成工具，允许设计者对一个或多个 FPGA 进行编程，同时和这些 FPGA 进行交互。可以通过 Vivado 集成环境用户接口或者使用 Tcl 命令，连接 FPGA 硬件系统。在这两种方式下，连接目标 FPGA 器件的步骤都相同。包括：

- ① 打开硬件会话(Open hardware session)；
- ② 通过运行在主机上的硬件服务器(hardware server)，打开硬件目标器件；
- ③ 给需要编程的目标 FPGA 器件分配相应的比特流编程文件；
- ④ 将文件编程或者下载到目标器件中。

使用 Vivado 硬件会话编程 FPGA 的步骤主要包括：

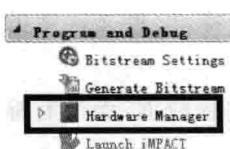


图 2.56 下载编程文件选项

(1) 如图 2.56 所示, 在 Vivado 左侧的 Flow Navigator 窗口下方, 找到 Program and Debug(编程和调试) 选项并展开。在展开项中, 选择 Hardware Manager。

(2) 如图 2.57 所示, 在 Vivado 界面上方出现 Hardware Session(硬件会话)界面。在该界面的上方单击 Open a new hardware target 打开一个新的目标。

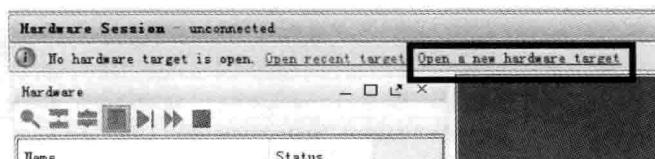


图 2.57 启动硬件目标入口

注: 使用 USB 下载线, 将其插入到 NEXY4 板卡上的 J6 USB 插座上, 建立 FPGA 和主机之间的连接。

- (3) 出现 Open New Hardware Target(打开新硬件目标)对话框界面。
- (4) 单击 Next 按钮。
- (5) 如图 2.58 所示, 出现 Open New Hardware Target(打开新硬件目标)-Vivado CSE Server Name(Vivado CSE 服务器名字)对话框界面。

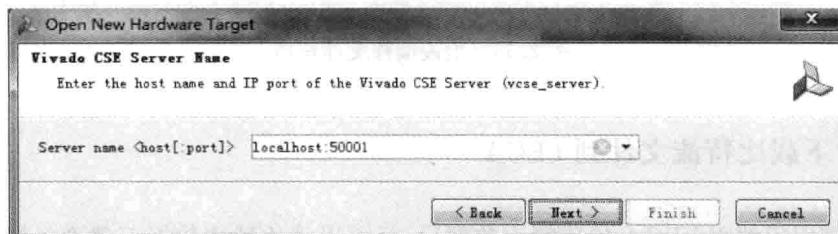


图 2.58 输入端口号界面

- (6) 单击 Next 按钮。
- (7) 如图 2.59 所示, 出现 Open Server(打开服务器)对话框界面, 表示正在打开服务器。

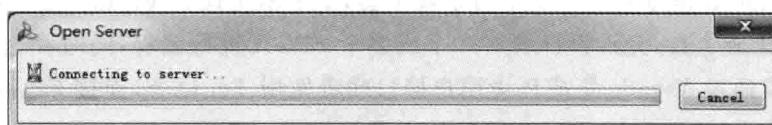


图 2.59 正在打开服务器界面

- (8) 成功打开服务器后, 如图 2.60 所示, 在 Hardware Targets 下面给出了如下信息:

① Type: xilinx\_tcf;

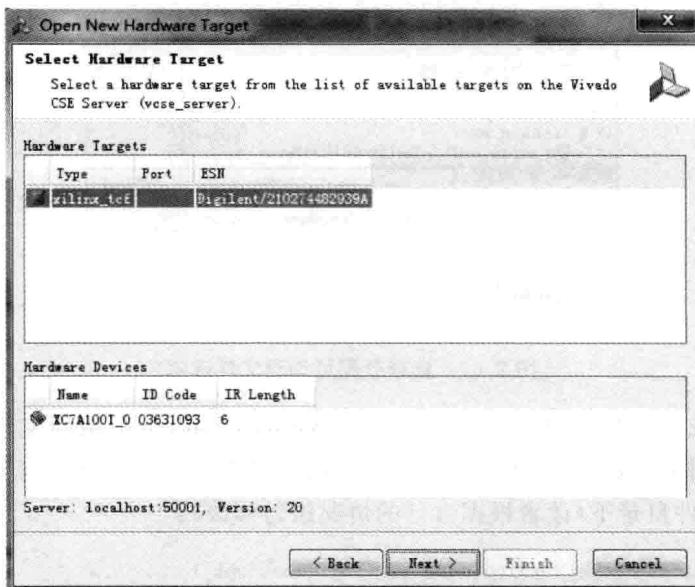


图 2.60 打开新硬件目标对话框界面

(9) 单击 Next 按钮。

(10) 如图 2.61 所示, 出现 Open New Hardware Target(打开新硬件目标)-Set Hardware Target Properties(设置硬件目标属性)对话框界面。设计者可以设置下载频率(注: 使用默认设置)。

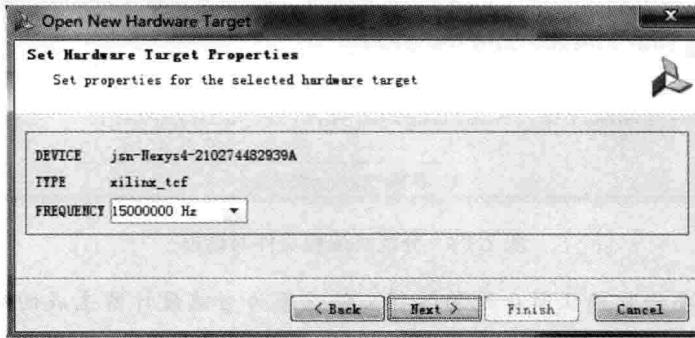


图 2.61 出现设置频率窗口

(11) 单击 Next 按钮。

(12) 出现 Open New Hardware Target(打开新硬件目标)-Open Hardware Target Summary(打开硬件目标总结)对话框界面。

(13) 单击 Finish 按钮。

(14) 出现如图 2.62 所示的 Hardware Session(硬件会话)对话框界面。在该界面中,选中 XC7A100T 器件,单击右键,出现浮动菜单,选择 Assign Programming File…(分配编程文件)选项。

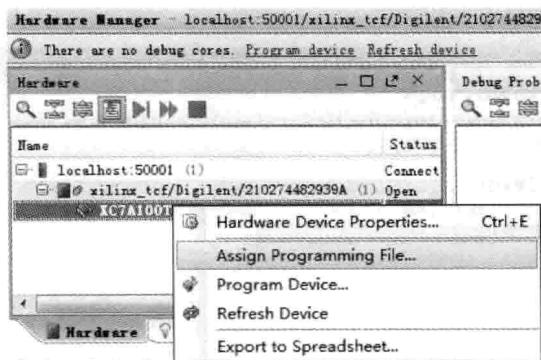


图 2.62 选择分配可编程文件选项

(15) 如图 2.63 所示,出现 Assign Programming File(分配编程文件)对话框界面。在该界面中,单击  按钮。出现 Open File 对话框界面,在该界面中,将分配文件指向当前生成的 bit 文件目录下(读者根据自己的情况进行修改):

```
E:\vivado_example\gate_verilog\gate_verilog.runs\impl_1
```

(16) 如图 2.63 所示,单击 Apply 按钮。

(17) 在如图 2.61 所示的界面下,再次选中 XC7A100T 器件。然后,单击右键,出现浮动菜单。在浮动菜单内,选择 Program Device…(编程器件)。

(18) 如图 2.64 所示,出现 Program Device(编程器件)-Select Bitstream file for device XC7A100T(选择用于 XC7A100T 的比特流文件)对话框界面。

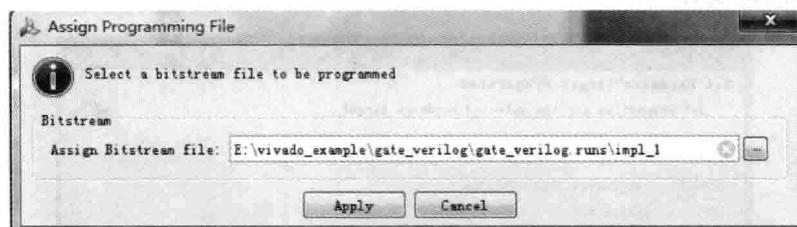


图 2.63 分配可编程文件对话框

**注:** 此处需要仔细确认所分配的编程文件是否为当前设计所生成的比特流文件。

(19) 单击 Program 按钮。

(20) 如图 2.65 所示,出现 Program Xilinx FPGA Device(编程 Xilinx FPGA 器件)对话框界面。

(21) 编程成功后,在 Nexys4 板卡上,验证该设计的正确性。

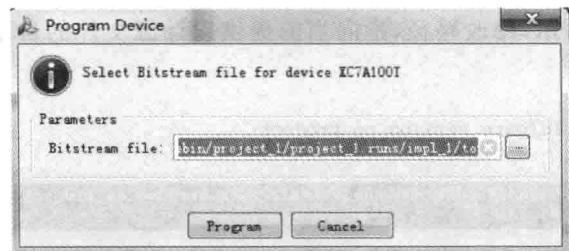


图 2.64 确认选择的编程文件对话框界面

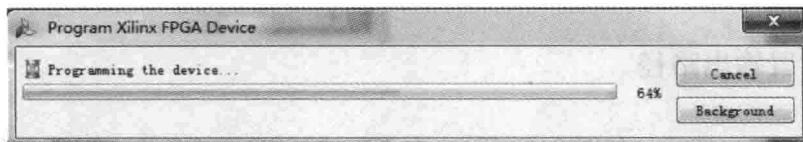


图 2.65 编程 FPGA 对话框界面

## 2.2 非工程模式设计流程

本节将分几个部分来介绍 Vivado 非工程模式设计流程。

### 2.2.1 修改路径

本节将启动 Vivado TCL 环境，并修改路径，将路径指向设计文件所在的目录。修改路径的步骤主要包括：

- (1) 在 Window 7 操作系统主界面下，选择开始→所有程序→Xilinx Design Tools→Vivado 2013.3→Vivado 2013.3 Tcl Shell。
- (2) 如图 2.66 所示，出现 Vivado 2013.3 Tcl Shell 界面。在出现一系列提示信息后，出现提示符：

Vivado %

```
cat Vivado 2013.3 Tcl Shell - C:\Xilinx\Vivado\2013.3\bin\vivado.bat -mode tcl

*****
**** Vivado v2013.3
**** SW Build 329390 on Wed Oct 16 18:35:21 MDT 2013
**** IP Build 192953 on Wed Oct 16 08:44:02 MDT 2013
** Copyright 1986-1999, 2001-2013 Xilinx, Inc. All Rights Reserved.

INFO: [Common 17-78] Attempting to get a license: Implementation
INFO: [Common 17-81] Feature available: Implementation
INFO: [Device 21-36] Loading parts and site information from C:/Xilinx/Vivado/2013.3/data/parts/arch.xml
Parsing RTL primitives file [C:/Xilinx/Vivado/2013.3/data/parts/xilinx/rtl/prims/rtl_prims.xml]
Finished parsing RTL primitives file [C:/Xilinx/Vivado/2013.3/data/parts/xilinx/rtl/prims/rtl_prims.xml]
Vivado%
```

图 2.66 启动 Vivado 2013.3 Tcl Shell 界面

(3) 如图 2.67 所示,修改路径,指向当前提供设计源文件的目录。在 Vivado% 提示符后面输入命令:

```
cd e:/vivado_example/gate_verilog_no_project
```

```
Vivado% cd e:/vivado_example/gate_verilog_no_project
Vivado%
```

图 2.67 修改设计路径

注: 读者根据情况修改路径。

## 2.2.2 设置输出路径

下面设置输出路径。如图 2.68 所示,在 Vivado % 提示符后输入下面命令:

```
Set outputDir ./gate_Created_Data/top_output
File mkdir $outputDir
```

```
Vivado% set outputDir ./gate_Created_Data/top_output
./gate_Created_Data/top_output
Vivado% file mkdir $outputDir
Vivado%
```

图 2.68 设置输出路径

## 2.2.3 设置设计源文件和约束

下面将设置设计源文件和约束。如图 2.69 所示,在 Vivado % 提示符后输入下面命令:

```
read_verilog top.v
read_xdc top.xdc
```

```
Vivado% read_verilog top.v
e:/vivado_example/gate_verilog_no_project/top.v
Vivado% read_xdc top.xdc
e:/vivado_example/gate_verilog_no_project/top.xdc
Vivado%
```

图 2.69 设置设计源文件和约束

注:

- (1) read\_edif——将一个 EDIF 或者 NGC 网表读入到当前工程的源文件中;
- (2) read\_verilog——读入 Verilog(.v) 和 System Verilog(.sv) 源文件, 用于非工程模式;
- (3) read\_vhdl——读入 VHDL(.vhd 或者 .vhdl) 源文件, 用于非工程模式;
- (4) read\_xdc——读入 .sdc/.xdc 格式的约束文件, 用于非工程模式。

## 2.2.4 运行综合

下面对设计进行综合。在 Vivado% 提示符后依次输入下面命令：

```
synth_design - top top - part xc7a100tcsg324-1
write_checkpoint - force $outputDir/post_synth
report_timing_summary - file $outputDir/post_synth_timing_summary.rpt
report_power - file $outputDir/post_synth_power.rpt
```

**注：**命令格式为：

(1) synth\_design：

```
synth_design [ - name <arg> ] [ - part <arg> ] [ - constrset <arg> ] [ - top <arg> ] [ - include
_dirs <args> ] [ - generic <args> ] [ - verilog_define <args> ] [ - flatten_hierarchy <arg> ]
[ - gated_clock_conversion <arg> ] [ - directive <arg> ] [ - rtl ] [ - bufg <arg> ] [ - no_lc ]
[ - fanout_limit <arg> ] [ - mode <arg> ] [ - fsm_extraction <arg> ] [ - keep_equivalent_
registers ] [ - resource_sharing <arg> ] [ - control_set_opt_threshold <arg> ] [ - quiet ]
[ - verbose ]
```

(2) write\_checkpoint/read\_checkpoint：

在处理流程的任何一点保存设计。一个设计检查点(checkpoint)由设计流程中该点带有任何优化的网表和约束构成。

(3) report\_\*：

运行多个标准的报告，可以在设计流程的任何阶段运行该命令。

## 2.2.5 运行布局

下面对设计运行布局和逻辑优化，报告利用率和时序估计，写检查点设计。在 Vivado% 提示符后依次输入下面命令：

```
opt_design
power_opt_design
place_design
phys_opt_design
write_checkpoint - force $outputDir/post_place
report_timing_summary - file $outputDir/post_place_timing_summary.rpt
```

**注：**命令格式为：

(1) opt\_design——执行高层次设计优化。

(2) power\_opt\_design——执行智能时钟门控，用于降低系统整体功耗(该步骤可选)。

(3) place\_design——对设计进行布局。

(4) phys\_opt\_design——执行物理逻辑优化，以改善时序或者鲁棒性(该步骤可选)。

## 2.2.6 运行布线

下面对设计运行布线器, 报告真正的利用率和时序, 写检查点设计, 运行 DRC, 写 verilog 和 xdc。在 Vivado% 提示符后依次输入下面命令:

```
route_design
write_checkpoint - force $outputDir/post_route
report_timing_summary - file $outputDir/post_route_timing_summary.rpt
report_timing - sort_by group - max_paths 100 - path_type summary - file
$outputDir/post_route_timing.rpt
report_clock_utilization - file $outputDir/clock_util.rpt
report_utilization - file $outputDir/post_route_util.rpt
report_power - file $outputDir/post_route_powet.rpt
report_drc - file $outputDir/post_imp_drc.rpt
write_verilog - force $outputDir/top_impl_netlist.v
write_xdc - no_fixed_only - force $outputDir/top_impl.xdc
```

注: 命令格式为:

route\_design——对设计进行布线。

## 2.2.7 生成比特流文件

下面生成比特流, 在 Vivado% 提示符后依次输入下面命令:

```
write_bitstream - force $outputDir/top.bit
```

注: 命令格式为:

write\_bitstream——生成比特流文件并运行 DRC。

# 第3章 Vivado 调试流程

本章调用 Vivado 提供的 IP 核,生成用户定制的 IP,并通过 HDL 语言进行例化。然后,通过 Vivado 提供的调试功能对设计进行调试,并分析调试结果。

## 3.1 设计调试原理和方法

对 FPGA 的调试,是一个反复迭代,直到满足设计功能和设计时序的过程。对于 FPGA 这样比较复杂数字系统的调试,就是将其分解成一个个很小的部分。然后,对设计中的每个很小的部分通过仿真或者调试进行验证。这样要比在一个复杂设计完成后,再进行仿真或者调试的效率要高得多。设计者可以通过使用下面的设计和调试方法,来保证设计的正确性:

- (1) RTL 级的设计仿真;
- (2) 实现后设计仿真;
- (3) 系统内调试。

对于前面两种方法,在本书第 2 章进行了详细的说明。本章将通过一个设计实例,详细介绍系统内调试方法。

(1) 系统内逻辑设计调试: Vivado 集成设计环境包含逻辑分析特性,使设计者可以对一个实现后的 FPGA 器件进行系统内调试。在系统内对设计进行调试的好处包括: 在真正的系统环境下,以系统要求的速度,调试设计的时序准确性和实现后的设计。系统内调试的局限性包括: 与使用仿真模型相比,稍微降低了调试信号的可视性,潜在地延长了设计/实现/调试迭代的时间。这个时间取决于设计的规模和复杂度。

通常,Vivado 工具提供了不同的方法,用于调试设计。设计者可以根据需要,使用这些方法。

(2) 系统内串行 I/O 设计调试: 为了实现系统内对串行 I/O 验证和调试,Vivado 集成开发环境包括一个串行的 I/O 分析特性。这样,设计者可以在基于 FPGA 的系统中,测量并且优化高速串行 I/O 连接。这个特性,可以解决大范围的系统内调试和验证问题,范围从简

单的时钟和连接问题,到复杂的余量分析和通道优化问题。与外部测量仪器技术相比,使用 Vivado 内的串行 I/O 分析仪可以测量接收器对接收信号进行均衡后的信号质量。这样,就可以在 Tx 到 Rx 通道的最优点进行测量。因此,就可以确保得到真实和准确的数据。

Vivado 工具提供了用于生成设计的工具。该设计应用吉比特收发器端点和实时软件进行测量,帮助设计者优化高速串行 I/O 通道。

系统内调试包括三个重要的阶段:

(1) 探测阶段(Probing phase): 用于标识需要对设计中的哪个信号进行探测,以及探测的方法;

(2) 实现阶段(Implementation phase): 实现设计,包括将额外的调试 IP 连接到被标识为探测的网络;

(3) 分析阶段(Analysis phase): 通过与设计中的调试 IP 进行交互,调试和验证设计功能。

本节介绍探测设计用于系统内调试,关于实现和分析,在后面会详细介绍。

在探测阶段分为两个步骤:

(1) 识别需要探测的信号或者网络;

(2) 确认将调试核添加到设计中的方法。

很多时候,设计者决定需要探测的信号,以及探测这些信号的方法。它们之间互相影响。设计者可以手工添加调试 IP 元件例化到设计源代码中(称为 HDL 例化探测流程),设计者也可以让 Vivado 工具自动地将调试核插入到综合后的网表中(称为网表插入探测流程)。表 3.1 给出了不同调试方法的优势和权衡。

表 3.1 调试策略

调试目标	推荐的调试编程流程
在 HDL 源代码中识别调试信号,同时保留灵活性,用于流程后面使能或者禁止调试	(1) 在 HDL 中,使用 mark_debug 属性标记需要调试的信号; (2) 使用 Set up Debug 向导来引导设计者通过网表插入探测流程
在综合后的设计网表中识别调试网络,不需要修改 HDL 源代码	(1) 使用 Mark Debug 单击菜单选项,选择在综合设计的网表中需要调试的网络; (2) 使用 Set up Debug 向导来引导设计者通过网表插入探测流程
使用 Tcl 命令,自动调试探测流程	(1) 使用 set_property Tcl 命令,在调试网络上设置 mark_debug 属性; (2) 使用网表插入探测流程 Tcl 命令,创建调试核,并将其连接到调试网络
明确在 HDL 语言中,将信号添加到 ILA 调试核中	(1) 识别用于调试的 HDL 信号; (2) 使用 HDL 例化探测流程产生和例化一个集成逻辑分析仪(ILA)核,并且将它连接到设计中的调试信号

## 3.2 创建新的 FIFO 调试工程

本节将创建一个新的 FIFO 调试工程。创建新的 FIFO 调试工程的步骤主要包括:

- (1) 在 Vivado 主界面主菜单下,选择 File→New Project。
- (2) 出现 New Project-Create a New Vivado Project 对话框界面。
- (3) 单击 Next 按钮。
- (4) 出现 New Project-Project Name 对话框界面。按下面参数设置:  
① Project name: fifo\_vhdl 或者 fifo\_verilog。  
注: 使用 VHDL 的读者设置名字为 fifo\_vhdl, 使用 Verilog 的读者设置名字为 fifo\_verilog。
- (5) 单击 Next 按钮。
- (6) 出现 New Project-Project Type 对话框界面,在该界面内选择 RTL Project。选中 Do not specify source at this time 前的复选框。
- (7) 单击 Next 按钮。
- (8) 出现 New Project-Default Part 对话框界面,在该界面内选择: xc7a100tcsg324-1。
- (9) 单击 Next 按钮。
- (10) 出现 New Project-New Project Summary 对话框界面。
- (11) 单击 Finish 按钮。

### 3.3 添加 FIFO IP 到设计中

本节将使用 IP 生成器,例化一个 FIFO IP。添加 FIFO IP 到设计中的步骤主要包括:

- (1) 如图 3.1 所示,在 Vivado 主界面左侧的 Flow Navigator 窗口下,选择并展开 Project Manager。在展开项中,单击 IP Catalog。
- (2) 如图 3.2 所示,在 Vivado 主界面右侧出现 IP Catalog 标签窗口。在该标签窗口下,列出了可供使用的 IP 核。在该窗口界面中,找到并展开 Memories & Storage Elements。在展开项中找到 FIFOs。

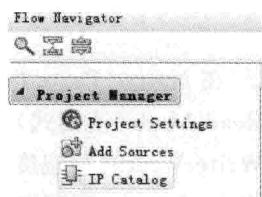


图 3.1 选择 IP Catalog 入口

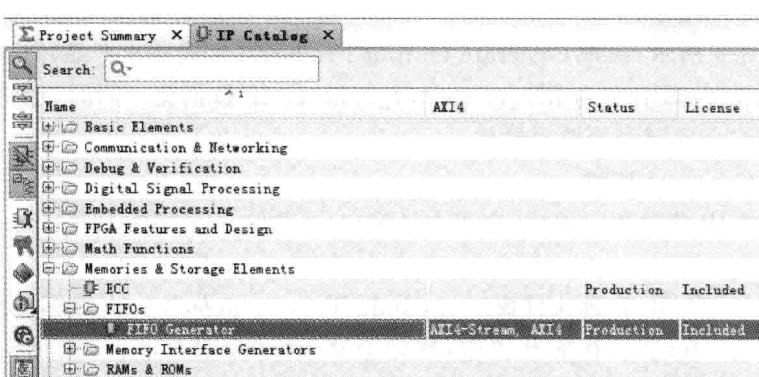


图 3.2 生成 FIFO IP 核入口

- (3) 展开 FIFOs。选择并双击 FIFO Generator 选项。
- (4) 如图 3.3 所示,出现 FIFO Generator(FIFO 生成器)对话框界面。在该界面中,按如下参数设置。

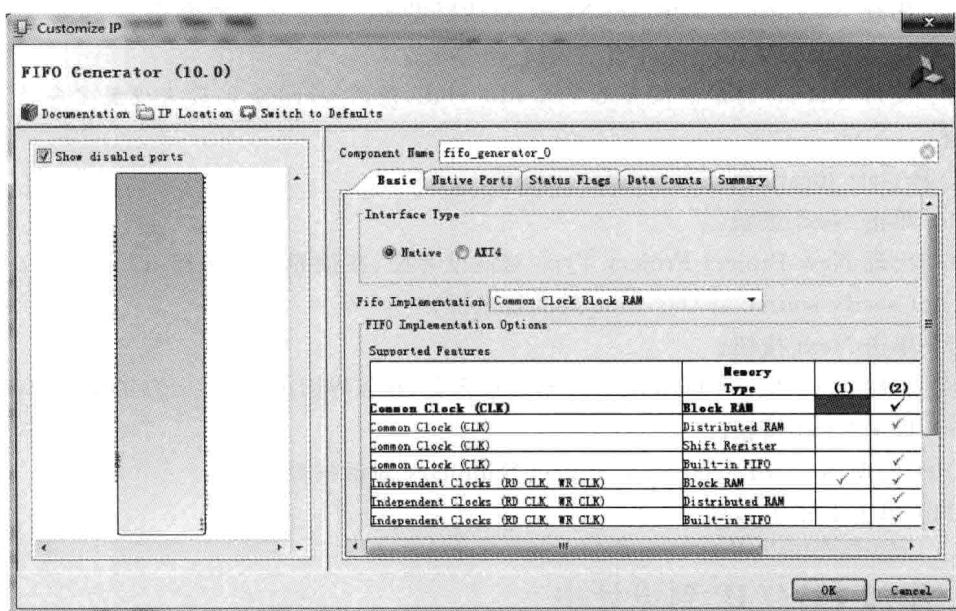


图 3.3 FIFO 生成器界面

① 在 Basic 标签窗口下:

Interface Type: Native;

Fifo Implementation(Fifo 实现): Common Clock Block RAM(公共时钟 BRAM)。

② 在 Native Ports 标签窗口下:

Read Mode(读模式): Standard FIFO;

Write Width(写宽度): 8;

Write Depth(写深度): 16。

其余按默认参数设置。

(5) 单击 OK 按钮。

(6) 如图 3.4 所示,出现 Generate Output Products(生成输出产品)对话框界面。在该界面中,提示将生成名字为 fifo\_generator\_0.xci 的 IP 核例化。同时,也将生成例化模板、工程例子、综合后网表等输出文件。

(7) 单击 Generate 按钮。

(8) 当生成 IP 核过程完成后,在源文件窗口内,添加了名字为 fifo\_generator\_0 的 IP 核设计。

(9) 如图 3.5 所示,在源文件窗口中,单击 IP Sources 标签。在该标签窗口下,单击并展开 fifo\_generator\_0。在展开项中,找到 Instantiation Template 例化模板。对于使用 VHDL 语言的读者,如图 3.5(a)所示,选择并单击 fifo\_generator\_0.vho 文件;对于使用 Verilog 语言的读者,如图 3.5(b)所示,选择并单击 fifo\_generator\_0.veo 文件。

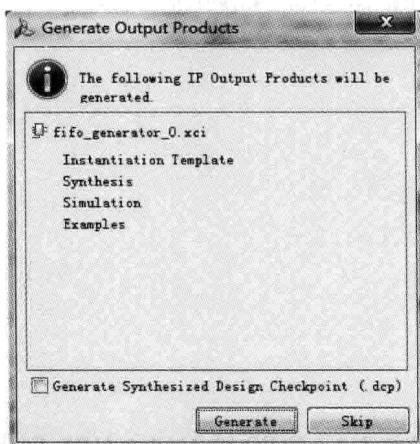


图 3.4 生成 FIFO 界面

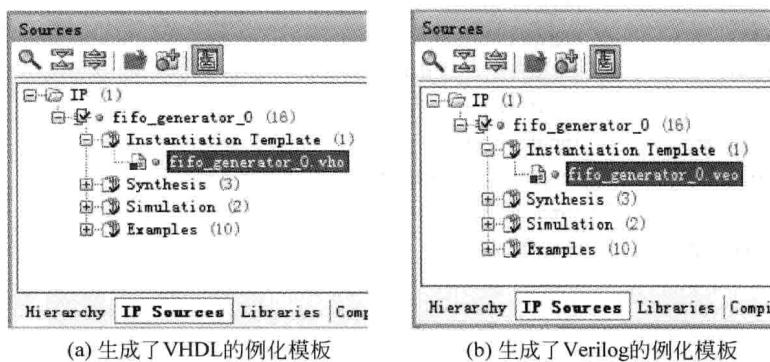


图 3.5 FIFO 的不同 HDL 语言例化模板

### 代码清单 3-1 FIFO 的 VHDL 例化模板

```

COMPONENT fifo_generator_0
PORT (
    clk : IN STD_LOGIC;
    rst : IN STD_LOGIC;
    din : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
    wr_en : IN STD_LOGIC;
    rd_en : IN STD_LOGIC;
    dout : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
    full : OUT STD_LOGIC;
    empty : OUT STD_LOGIC
);
END COMPONENT;
-- COMP_TAG_END ----- End COMPOENT Declaration -----
-- The following code must appear in the VHDL architecture
-- body. Substitute your own instance name and net names.

```

```
-- Begin Cut here for INSTANTIATION Template ----- INST_TAG
your_instance_name : fifo_generator_0
PORT MAP (
    clk => clk,
    rst => rst,
    din => din,
    wr_en => wr_en,
    rd_en => rd_en,
    dout => dout,
    full => full,
    empty => empty
);

```

代码清单 3-2 FIFO 的 Verilog HDL 例化模板

```
fifo_generator_0 your_instance_name (
    .clk(clk),           // input clk
    .rst(rst),           // input rst
    .din(din),           // input [7 : 0] din
    .wr_en(wr_en),       // input wr_en
    .rd_en(rd_en),       // input rd_en
    .dout(dout),         // output [7 : 0] dout
    .full(full),         // output full
    .empty(empty)         // output empty
);

```

### 3.4 添加顶层设计文件

本节将使用 VHDL/Verilog 语言生成顶层设计文件，并且通过 VHDL/Verilog 语言的例化语句将 FIFO 核添加到设计中。

- (1) 在源文件窗口中，将标签重新切换到 Hierarchy 标签。
- (2) 在该窗口界面内，单击右键出现浮动菜单，选择 Add Sources…。
- (3) 出现 Add Sources(添加源文件)窗口界面。在该窗口界面内，选择 Add or Create Design Sources(添加或者创建设计源文件)选项。
- (4) 单击 Next 按钮。
- (5) 出现 Add or Create Design Sources(添加或者创建设计源文件)对话框界面。在该界面内，单击 Create File…按钮。
- (6) 出现 Create Source File(创建源文件)对话框界面。在该界面中按下面参数设置：

- ① File type: VHDL/Verilog；

注：使用 VHDL 语言设计的读者选择 VHDL；使用 Verilog 语言设计的读者选择 Verilog。

- ② File name: top。

- (7) 单击 OK 按钮。
- (8) 退回到 Add Sources 对话框界面。此时,已经添加 top.vhd 或者 top.v 文件。
- (9) 单击 Finish 按钮。
- (10) 出现 Define Module(定义模块)对话框界面。不做任何操作,直接单击 OK 按钮。
- (11) 使用 VHDL 语言的读者,在源文件窗口下,选择并打开 top.vhd 文件; 使用 Verilog 语言的读者,在源文件窗口下,选择并打开 top.v 文件。

### 3.5 例化 FIFO

输入 VHDL/Verilog 语言,完成 top.vhd/top.v 文件代码。使用 VHDL 语言的读者参考 top.vhd 代码清单,使用 Verilog 语言的读者参考 top.v 代码清单。

代码清单 3-3 top.vhd 代码

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
-- use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
-- library UNISIM;
-- use UNISIM.VComponents.all;

entity top is
Port (
    rd_trig      : in  std_logic;          -- read trigger signal
    rst          : in  std_logic;          -- reset signal
    clk          : in  std_logic;          -- clock input signal
    wr_trig      : in  std_logic;          -- write trigger signal
    dout         : out std_logic_vector(7 downto 0); -- fifo data output
    empty        : out std_logic;          -- fifo empty flag
    full         : out std_logic;          -- fifo full flag
);
end top;

architecture Behavioral of top is
type fifo_data is array(15 downto 0) of std_logic_vector(7 downto 0);
signal data_in  : fifo_data:= (x"00",x"01",x"02",x"03",x"04",x"05",x"06",x"07",
                               x"08",x"09",x"0a",x"0b",x"0c",x"0d",x"0e",x"0f");
type state is(ini,wr_fifo,ready,rd_fifo);
signal next_state : state ;

```

```

signal rd_en      :  std_logic;
signal wr_en      :  std_logic;
signal j          :  integer range 0 to 15;
signal din        :  std_logic_vector(7 downto 0);
COMPONENT fifo_generator_0           --- copy from Instantiation Template
PORT (
    ' clk : IN STD_LOGIC;
    ' rst : IN STD_LOGIC;
    din : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
    wr_en : IN STD_LOGIC;
    rd_en : IN STD_LOGIC;
    dout : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
    full : OUT STD_LOGIC;
    empty : OUT STD_LOGIC
);
END COMPONENT;
begin
Inst_fifo : fifo_generator_0
PORT MAP (
    clk => clk,
    rst => rst,
    din => din,
    wr_en => wr_en,
    rd_en => rd_en,
    dout => dout,
    full => full,
    empty => empty
);
process(rst,clk)
begin
    if(rst = '1') then
        next_state<= ini;
        j<= 0;
        rd_en<= '0';
        wr_en<= '0';
    elsif rising_edge(clk) then
        case next_state is
            when ini=>
                j<= 0;
                rd_en<= '0';
                if(wr_trig = '1')then
                    next_state<= wr_fifo;
                else
                    next_state<= ini;
                end if;
            when wr_fifo=>
                din<= data_in(j);
                if(j = 15) then
                    next_state<= ready;
                else
                    j <= j + 1;
                end if;
        end case;
    end if;
end process;
end;

```

```

        wr_en<= '1';
        next_state<= wr_fifo;
    end if;
when ready =>
    j <= 0;
    wr_en<= '0';
    if(rd_trig = '1') then
        next_state<= ini;
    else
        next_state<= rd_fifo;
    end if;
when rd_fifo =>
    if(j = 15) then
        next_state <= ini;
    else
        j <= j + 1;
        rd_en<= '1';
        next_state <= rd_fifo;
    end if;
end case;
end if;
end process;
end Behavioral;

```

代码清单 3-4 top.v 代码

```

module top(
    input wire rd_trig,
    input wire rst,
    input wire clk,
    input wire wr_trig,
    output wire dout,
    output wire empty,
    output wire full
);
reg [7:0] data_in [15:0];
initial
begin
    data_in[15] = 8'h0f; data_in[14] = 8'h0e; data_in[13] = 8'h0d; data_in[12] = 8'h0c;
    data_in[11] = 8'h0b; data_in[10] = 8'h0a; data_in[9] = 8'h09; data_in[8] = 8'h08;
    data_in[7] = 8'h07; data_in[6] = 8'h06; data_in[5] = 8'h05; data_in[4] = 8'h04;
    data_in[6] = 8'h03; data_in[5] = 8'h02; data_in[4] = 8'h01; data_in[3] = 8'h00;
end
reg [1:0] next_state;
parameter ini = 2'b00, wr_fifo = 2'b01, ready = 2'b11, rd_fifo = 2'b10;
reg wr_en;
reg rd_en;
reg[7:0] din;
reg[3:0] j;
fifo_generator_0 Inst_fifo1 (
    .clk(clk), // input clk
    .rst(rst), // input rst

```

```

    .din(din),           // input [7 : 0] din
    .wr_en(wr_en),       // input wr_en
    .rd_en(rd_en),       // input rd_en
    .dout(dout),         // output [7 : 0] dout
    .full(full),         // output full
    .empty(empty)        // output empty
);
always @(posedge rst or posedge clk)
begin
    if(rst)
        begin
            next_state<= ini;
            j <= 0;
            rd_en <= 1'b0;
            wr_en <= 1'b0;
        end
    else
        begin
            case (next_state)
                ini :
                    begin
                        j <= 0;
                        rd_en <= 1'b0;
                        if(wr_trig == 1'b1)
                            next_state <= wr_fifo;
                    end
                wr_fifo:
                    begin
                        din <= data_in[j];
                        if(j == 15)
                            next_state <= ready;
                        else
                            begin
                                j <= j + 1;
                                wr_en <= 1'b1;
                                next_state <= wr_fifo;
                            end
                    end
                ready:
                    begin
                        j <= 0;
                        wr_en <= 1'b0;
                        if(rd_trig == 1'b1)
                            next_state <= rd_fifo;
                        else
                            next_state <= ready;
                    end
                rd_fifo:
                    begin
                        if(j == 15)
                            next_state <= ini;
                        else
                            begin

```

```

        j <= j + 1;
        rd_en <= 1'b1;
        next_state <= rd_fifo;
    end
end
endcase
end
end
endmodule

```

## 3.6 添加约束文件

本节将为设计指定引脚约束,该引脚约束参考Nexys4板子的设计图纸和相关资料。添加约束条件的步骤包括:

- (1) 在Vivado源文件窗口下,单击右键,出现浮动菜单。在浮动菜单内,选择Add Sources...,或者Edit Constraints Sets... (当选择该选项的时候,跳过下面第二步的对话框界面,直接进入第(3)步)。
- (2) 出现Add Sources对话框界面。在该界面下,选择Add or Create Constraints。
- (3) 单击Next按钮。
- (4) 出现Add or Create Constraints(添加或者创建约束)对话框界面。在该界面下,单击Create File...按钮。
- (5) 出现Create Constraints File(创建约束文件)对话框界面。按下面参数设置:
  - ① File type: XDC;
  - ② File name: top;
  - ③ File location: Local to Project。
- (6) 单击OK按钮,返回到添加或者创建约束对话框界面。可以看到添加了名字为top.xdc的约束文件。
- (7) 单击Finish按钮。可以看到在源文件窗口下添加了top.xdc文件。
- (8) 如图3.6所示,在下拉框中选择I/O Planning(I/O规划)。

**注:**需要执行完前面的综合过程,并且,在综合完后,选择Open Synthesized Design。

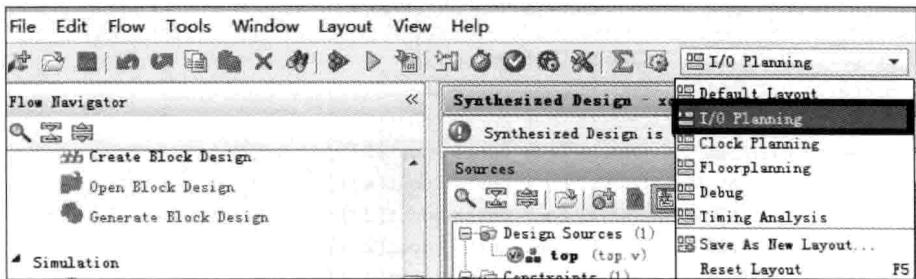


图3.6 选择I/O Planning

(9) 在 Vivado 主界面下方出现 I/O Ports(I/O 端口)对话框界面。如图 3.7 所示,在 Site 下输入每个逻辑端口在 FPGA 上所对应的引脚位置。此外,在 I/O Std(I/O 标准)下,为每个逻辑端口定义 I/O 电气标准。

(10) 输入完成后,在如图 3.7 所示的界面内,单击右键出现浮动菜单。在浮动菜单内,选择 Export I/O Ports…。

Name	Direction	Neg Diff Pair	Site	Fixed	Bank	I/O Std
dout (8)	Output					34 LVCMOS33*
dout[7]	Output		U6	<input checked="" type="checkbox"/>		34 LVCMOS33*
dout[6]	Output		U7	<input checked="" type="checkbox"/>		34 LVCMOS33*
dout[5]	Output		T4	<input checked="" type="checkbox"/>		34 LVCMOS33*
dout[4]	Output		T5	<input checked="" type="checkbox"/>		34 LVCMOS33*
dout[3]	Output		T6	<input checked="" type="checkbox"/>		34 LVCMOS33*
dout[2]	Output		R8	<input checked="" type="checkbox"/>		34 LVCMOS33*
dout[1]	Output		V9	<input checked="" type="checkbox"/>		34 LVCMOS33*
dout[0]	Output		T8	<input checked="" type="checkbox"/>		34 LVCMOS33*
Scalar ports (6)						
clk	Input		E3	<input checked="" type="checkbox"/>		35 LVCMOS33*
empty	Output		P2	<input checked="" type="checkbox"/>		34 LVCMOS33*
full	Output		R2	<input checked="" type="checkbox"/>		34 LVCMOS33*
rd_trig	Input		U8	<input checked="" type="checkbox"/>		34 LVCMOS33*
rst	Input		R7	<input checked="" type="checkbox"/>		34 LVCMOS33*
wr_trig	Input		U9	<input checked="" type="checkbox"/>		34 LVCMOS33*

图 3.7 I/O 引脚约束

(11) 出现 Export I/O Ports(导出 I/O 端口)对话框界面。选中 XDC,并将导出路径指向当前工程的 top.xdc。

① 对于 VHDL 语言的设计流程,路径指向:

E:\vivado\_example\gate\_vhdl\gate.srcc\constrs\_1\new\top.xdc

② 对于 Verilog 语言的设计流程,路径指向:

E:\vivado\_example\gate\_verilog\gate\_verilog.srcc\constrs\_1\new\top.xdc

(12) 单击 OK 按钮。

(13) 如图 3.6 所示,将选项再次切换到 Default Layout。

(14) 在源文件窗口,双击 top.xdc 文件。打开约束文件。下面给出了约束代码清单。

#### 代码清单 3-5 top.xdc

```
set_property IOSTANDARD LVCMOS33 [get_ports clk]
set_property IOSTANDARD LVCMOS33 [get_ports {dout[7]}]
set_property IOSTANDARD LVCMOS33 [get_ports {dout[6]}]
set_property IOSTANDARD LVCMOS33 [get_ports {dout[5]}]
set_property IOSTANDARD LVCMOS33 [get_ports {dout[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {dout[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {dout[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {dout[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {dout[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports empty]
set_property IOSTANDARD LVCMOS33 [get_ports rst]
```

```

set_property IOSTANDARD LVCMOS33 [get_ports full]
set_property IOSTANDARD LVCMOS33 [get_ports wr_trig]
set_property IOSTANDARD LVCMOS33 [get_ports rd_trig]
set_property PACKAGE_PIN T8 [get_ports {dout[0]}]
set_property PACKAGE_PIN V9 [get_ports {dout[1]}]
set_property PACKAGE_PIN R8 [get_ports {dout[2]}]
set_property PACKAGE_PIN T6 [get_ports {dout[3]}]
set_property PACKAGE_PIN T5 [get_ports {dout[4]}]
set_property PACKAGE_PIN T4 [get_ports {dout[5]}]
set_property PACKAGE_PIN U7 [get_ports {dout[6]}]
set_property PACKAGE_PIN U6 [get_ports {dout[7]}]
set_property PACKAGE_PIN R7 [get_ports rst]
set_property PACKAGE_PIN U8 [get_ports rd_trig]
set_property PACKAGE_PIN U9 [get_ports wr_trig]
set_property PACKAGE_PIN P2 [get_ports empty]
set_property PACKAGE_PIN R2 [get_ports full]
set_property PACKAGE_PIN E3 [get_ports clk]

```

## 3.7 网表插入调试探测流程方法及实现

本节首先介绍网表插入调试探测流程,然后基于该设计例子详细介绍其实现过程。

### 3.7.1 网表插入调试探测流程的方法

在分层设计时,插入 Vivado 工具中的调试核,用于解决 Vivado 设计者的不同需求:

- (1) 最高级是在一个简单的向导内,基于所选择的用于调试的网络集合,自动创建和配置集成逻辑分析仪(Integrated Logic Analyzer,ILA)V2.1 核。
- (2) 下一个层次是,在 Vivado 主界面主菜单下选择 Window→Debug。这个窗口内,允许控制单个调试核、端口和它们的属性。
- (3) 最低层次是 Tcl 调试命令集合,设计者可以手工输入 Tcl 或者将 Tcl 作为脚本反复使用。

对于设计者来说,可以根据情况,混合使用这三种方法插入和定制调试核。

#### 1. 标记用于调试的 HDL 信号

在综合前,通过使用 mark\_debug 约束,设计者可以在 HDL 源文件中标识用于调试的信号。在 HDL 中,标识用于调试的信号所对应的网络,将在 Debug 窗口中自动列出这些信号。

用于调试所标记网络的过程,取决于基于 RTL 级的工程还是综合后的网表工程。

(1) 对于一个 RTL 网表工程:

① 在 VHDL 或者 Verilog 源文件中,mark\_debug 约束标记可以用于调试的 HDL 信号。用于 mark\_debug 约束的有效值是 TRUE 或者 FALSE。Vivado 综合特性不支持 SOFT 值。

② 使用 Xilinx 综合技术(Xilinx Synthesis Technology,XST),设计者可以在 VHDL

和 Verilog 源文件中,通过 mark\_debug 约束来标记用于调试的网络。此外,有效的值是 TRUE 或者 FALSE。如果可能的话,SOFT 值允许软件优化指定的网络。

### (2) 对于一个综合后的网表工程

① 使用 Synopsys Synplify 综合工具,设计者可以在 VHDL 或者 Verilog 文件内,使用 mark\_debug 和 syn\_keep 约束。或者在 SDC 文件中单独使用 mark\_debug 约束,来标记用于调试的网络。当该行为由 syn\_kepp 属性控制时,Synplify 不支持 SOFT 值。

② 使用 Mentor Graphics Precision 综合工具,设计者可以在 VHDL 或者 Verilog 文件中,使用 mark\_debug 约束来标记用于调试的网络。

## 2. 标记用于调试的 HDL 信号的例子

### (1) Vivado 综合 mark\_debug 语法例子:

#### ① VHDL 语法例子:

```
attribute mark_debug : string;
attribute mark_debug of char_fifo_dout: signal is "true";
```

#### ② Verilog 语法例子:

```
(* mark_debug = "true" *) wire [7:0] char_fifo_dout;
```

### (2) XST mark\_debug 语法例子:

#### ① VHDL 语法例子:

```
attribute mark_debug : string;
attribute mark_debug of char_fifo_dout: signal is "true";
```

#### ② Verilog 语法例子:

```
(* mark_debug = "true" *) wire [7:0] char_fifo_dout;
```

### (3) Synplify mark\_debug 语法例子

#### ① VHDL 语法例子:

```
attribute syn_keep : boolean;
attribute mark_debug : string;
attribute syn_keep of char_fifo_dout: signal is true;
attribute mark_debug of char_fifo_dout: signal is "true";
```

#### ② Verilog 语法例子:

```
(* syn_keep = "true", mark_debug = "true" *) wire [7:0] char_fifo_dout;
```

#### ③ SDC 语法例子:

```
define_attribute {n:char_fifo_din[*]} {mark_debug} {"true"}
```

### (4) Precision mark\_debug 语法例子:

#### ① VHDL 语法例子:

```
attribute mark_debug : string;
```

```
attribute mark_debug of char_fifo_dout: signal is "true";
```

② Verilog 语法例子：

```
(* mark_debug = "true" *) wire [7:0] char_fifo_dout;
```

### 3.7.2 网表插入调试探测流程的实现

本节将介绍网表插入调试探测流程的实现过程。

#### 1. 添加测试点

本节将为调试添加测试点。添加测试点的步骤主要包括：

- (1) 选中顶层 top.v 或者 top.VHD 文件。
- (2) 在 Vivado 主界面左侧的 Flow Navigator 窗口下,选择并展开 Synthesis。在展开项中,选择并双击 Run Synthesis。开始对设计进行综合。
- (3) 等待综合完成后,出现 Synthesis Completed 对话框界面。在该界面中,选择 Open Synthesized Design。
- (4) 单击 OK 按钮。

**注:** 在打开综合后设计的过程中,可能出现 Critical Messages 对话框界面,当出现该对话框界面时,单击 Ok 按钮。

- (5) 如图 3.8 所示,在 Netlist 窗口下,列出了当前设计中存在的所有网络节点。

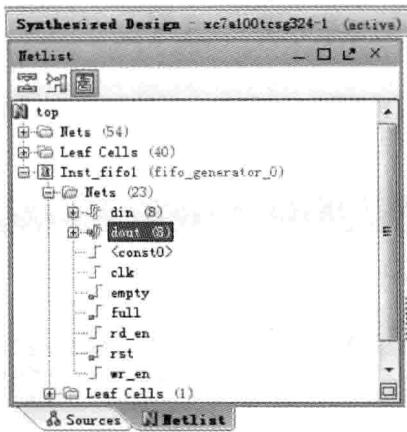


图 3.8 Netlist 窗口界面

- (6) 下面从所列出的网络节点中,选择需要调试的端口,并且进行标记。如图 3.9 所示,在 Netlist 窗口下,找到并展开 Inst\_fifo1。在展开项中,找到并选中 dout(8)。然后,单击右键,出现浮动菜单。在浮动菜单内,选择 Mark Debug。

- (7) 出现 Confirm Debug Net(s) 对话框界面,单击 OK 按钮。
- (8) 按照前面的方法,分别选择 din、rd\_en、wr\_en,将它们设置为 MarkDebug。
- (9) 图 3.10 给出了添加完调试端口后的 Netlist 窗口。

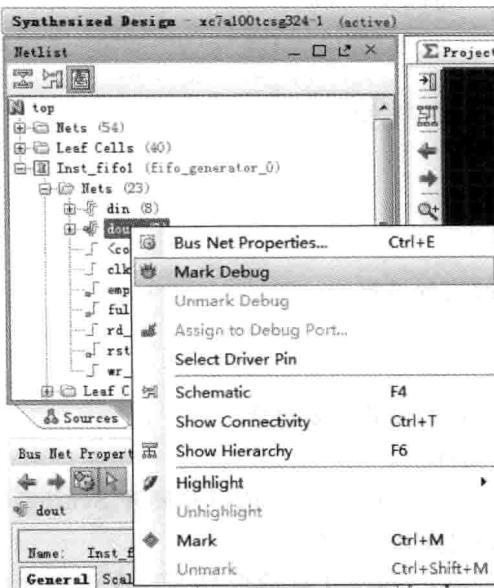


图 3.9 标记调试端口

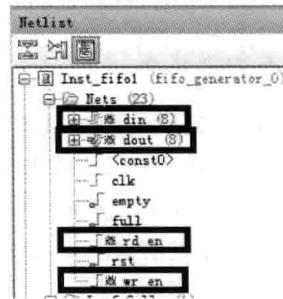


图 3.10 添加完成调试端口

## 2. 设置调试内核参数

本节将设置调试内核参数。设置调试内核参数的步骤主要包括：

(1) 在 Vivado 主界面主菜单下,选择 Tools→Set up Debug…。

(2) 出现 Set up Debug 对话框界面。

(3) 单击 Next 按钮。

(4) 如图 3.11 所示,可以看到 din、dout、rd\_en、wr\_en 所对应的 Clock Domain 显示红色的 undefined。

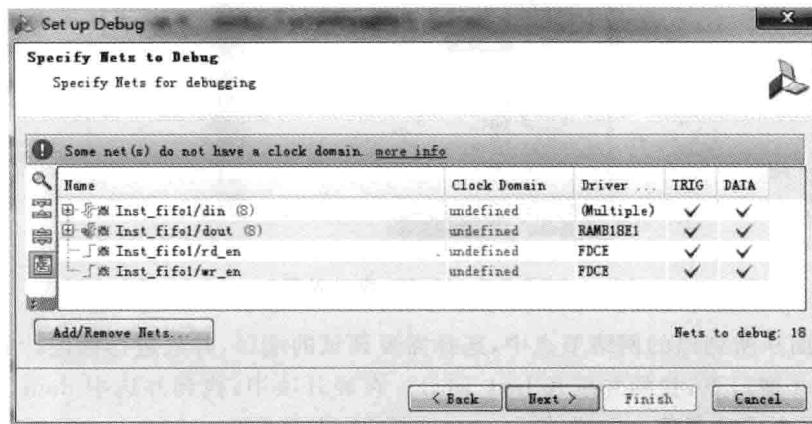


图 3.11 指定需要调试的网络

(5) 如图 3.12 所示,单击左键,同时按下 Shift 按键,选中所有四行需要调试的网络信号。然后,单击右键,出现浮动菜单。在浮动菜单内,选择 Select Clock Domain…选项。

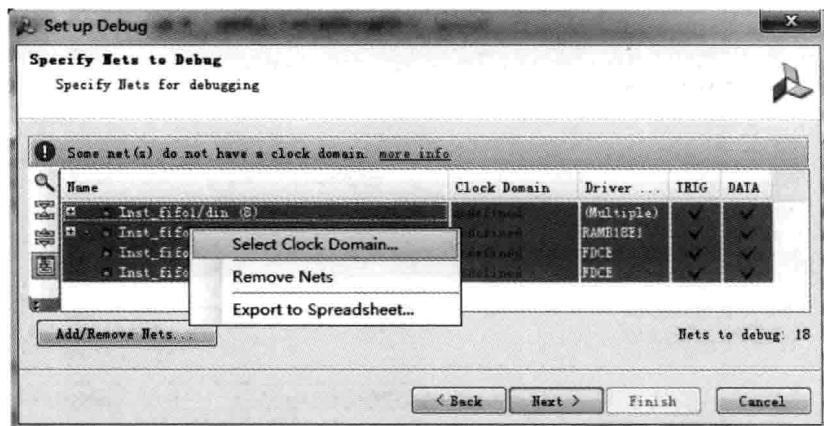


图 3.12 为需要调试的网络指定时钟网络

(6) 如图 3.13 所示, 出现 Select Clock Domain 对话框界面。在列表中, 选择 Inst\_fifo/clk。

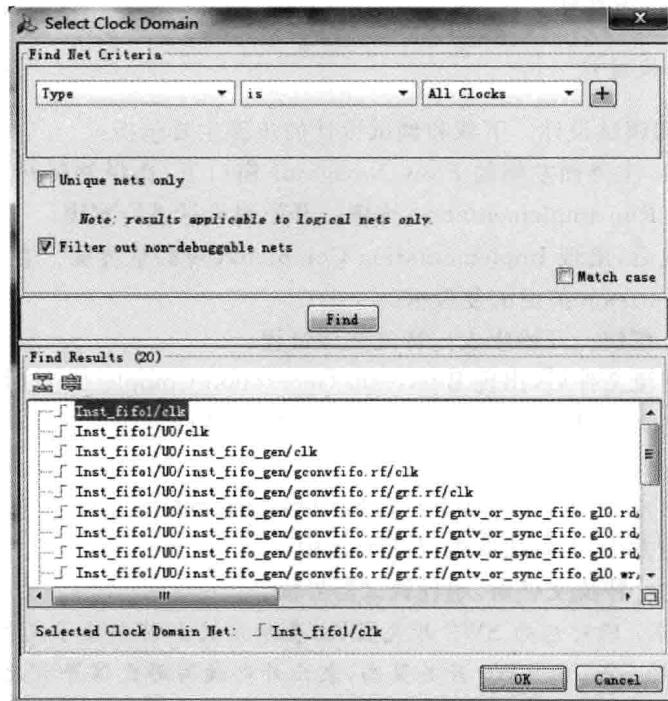


图 3.13 选择时钟域对话框界面

(7) 单击 OK 按钮。

(8) 如图 3.14 所示, 可以看到为每个调试网络都选择了时钟域。

(9) 单击 Next 按钮。

(10) 出现 Set up Debug-Trigger and Capture Modes(设置调试一触发器和捕获模式)对话框界面。在该界面下, 分别选中 Enable advanced trigger mode 和 Enable basic capture mode 前面的复选框。

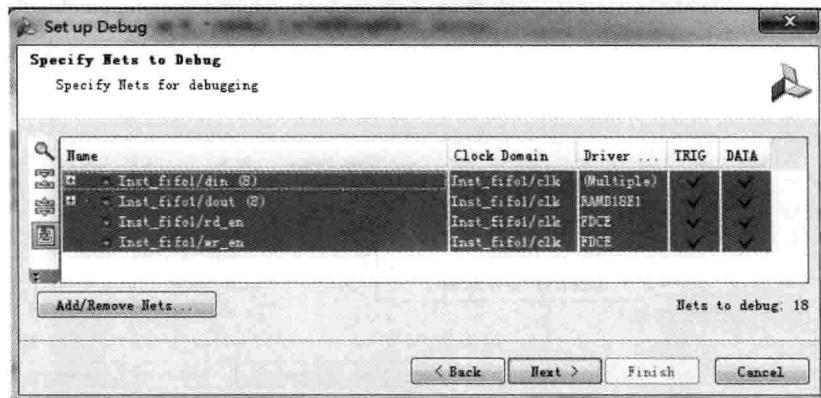


图 3.14 完成时钟域设置界面

- (11) 单击 Next 按钮。
- (12) 出现 Set up Debug Summary 对话框界面。
- (13) 单击 Finish 按钮。

### 3. 下载和调试设计

本节将下载和调试设计。下载和调试设计的步骤主要包括：

- (1) 在 Vivado 主界面左侧的 Flow Navigator 窗口下,选择并展开 Implementation。在展开项中,双击 Run Implementation 选项。开始对设计进行实现。
- (2) 实现结束后,出现 Implementation Completed 对话框界面。在该对话框界面内选中 Generate Bitstream 前面的复选框。
- (3) 单击 OK 按钮。开始生成比特流文件过程。
- (4) 生成比特流文件后,出现 Bitstream Generation Completed 对话框界面。在该对话框界面中,选中 Open Hardware Manager 前面的复选框。
- (5) 单击 OK 按钮。
- (6) 按前面的方法,单击 Open a new hardware target。
- (7) 按前面的方法,下载比特流到 FPGA 中。
- (8) 当下载完比特流文件后,出现调试器界面。  
注：此时,将 R7 所对应的 SW2 开关置低,表示系统当前不处于复位状态。同时,将 U8 和 U9 所对应的 SW1 和 SW0 开关置高,表示外部读写触发信号有效。
- (9) 在调试窗口中,找到如图 3.15 所示的 Basic Trigger Setup(基本触发设置)对话框界面。在该界面中,可以看到默认 Inst\_fifo1/rd\_en 的比较值为‘1’,Inst\_fifo1/wr\_en 的比较值为‘1’。单击 图标,出现浮动菜单选项。在浮动菜单内,选择 Set Trigger Condition to ‘Global OR’选项。该选项表示当 rd\_en=‘1’或者 wr\_en=‘1’时,满足触发条件,用于捕获数据。
- (10) 如图 3.16 所示,在调试界面内,找到 ILA Properties 窗口界面。在该界面中,单击 按钮。

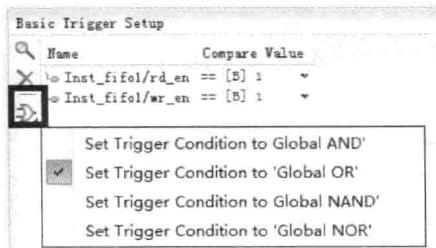


图 3.15 基本触发器设置界面

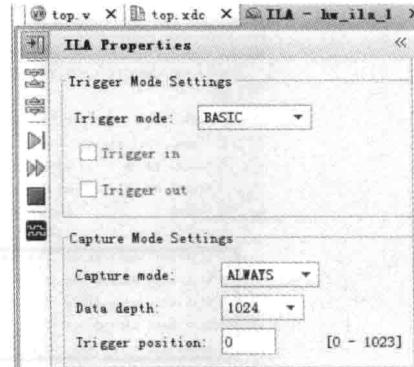


图 3.16 ILA 属性设置界面

(11) 出现如图 3.17 所示的波形窗口界面。在该界面中,单击 和 按钮。将波形调整到合适的位置。图 3.18 给出了调整后的波形窗口界面。

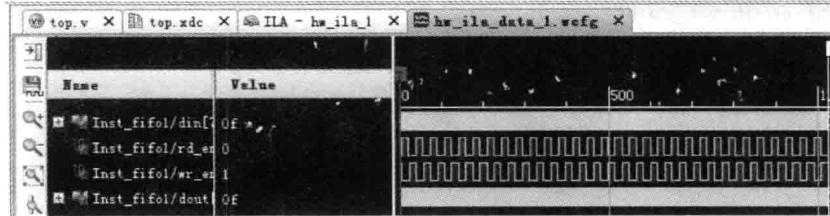


图 3.17 波形窗口界面

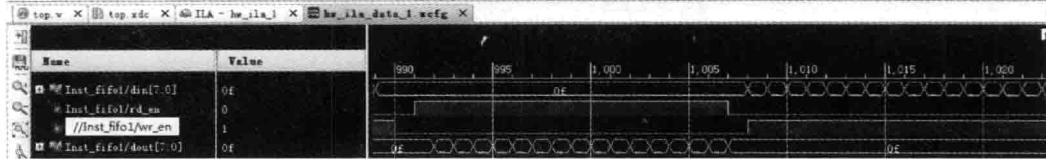


图 3.18 调整后的波形窗口界面

思考题 3.1: 请读者分析调试的结果和预先设计的要求是否一致?

### 3.8 使用添加 HDL 属性调试探测流程

下面以 VHDL 语言设计为例,在设计中添加 HDL 属性设置参数,对于 Verilog 读者,参考前面的 Verilog 属性设置语法进行参考设置。

(1) 选中 top.vhd 文件。

(2) 如图 3.19 所示,在 top.vhd 文件中,添加下面的属性声明语句。

注: 该属性声明语句表示,将 rd\_en、wr\_en、din、dout 网络添加到探测网络中。

(3) 保存文件。

(4) 在 Vivado 主界面左侧的 Flow Navigator 窗口内,找到并展开 Synthesis。在展开项中,选择并单击 Run Synthesis。

```

49 signal data_in : fifo_data := (x"00", x"01", x"02", x"03", x"04", x"05", x"06", x"07",
50                                         x"08", x"09", x"0a", x"0b", x"0c", x"0d", x"0e", x"0f");
51 type state is (ini, wr_fifo, ready, rd_fifo);
52 signal next_state : state;
53 signal rd_en : std_logic;
54 signal wr_en : std_logic;
55 signal j : integer range 0 to 15;
56 signal din : std_logic_vector(7 downto 0);
// 
57 attribute mark_debug : string;
58 attribute mark_debug of din : signal is "true";
59 attribute mark_debug of dout : signal is "true";
60 attribute mark_debug of rd_en : signal is "true";
61 attribute mark_debug of wr_en : signal is "true";
62 COMPONENT fifo_generator_0
  -- copy from Instantiation Template

```

图 3.19 添加属性声明语句

(5) 等待综合完成后, 打开综合后的设计。

(6) 如图 3.20 所示, 在 Netlist 界面内, 找到并展开 Nets。在展开项中, 可以看到将 din、dout、rd\_en 和 wr\_en 网络标记为调试。



图 3.20 已经标记为 debug 的网络

### 3.9 使用 HDL 例化调试核调试探测流程

本节将使用 HDL 例化调试核调试探测。使用 HDL 例化调试核调试探测的步骤主要包括:

- (1) 打开本书提供的 fifo\_verilog\_1 目录下的 Vivado 设计工程。
- (2) 在 Vivado 主界面左侧的 Flow Navigator 窗口下, 找到并单击 IP Catalog。
- (3) 在 Vivado 主界面右侧出现 IP Catalog 标签窗口界面。
- (4) 如图 3.21 所示, 找到并展开 Debug&Verification。在展开项中, 找到并展开 Debug。在展开项中, 双击 ILA(Integrated Logic Analyzer)。
- (5) 如图 3.22 所示, 将 Component Name 所对应的名字改为 fifo\_debug0。
- (6) 如图 3.22 所示, 选择 General Options 标签。在设标签窗口下, 按下面参数设置:

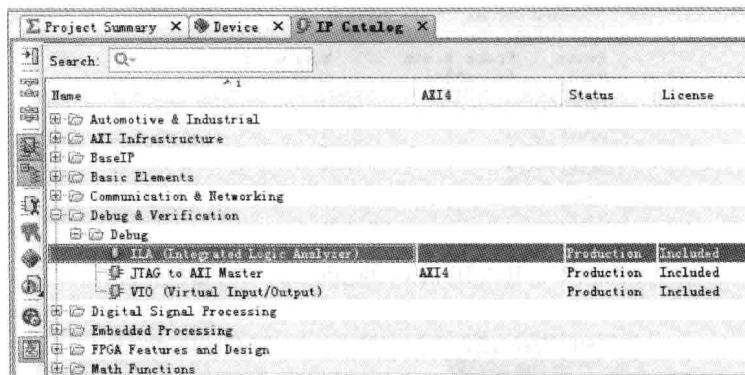


图 3.21 ILA IP 核入口

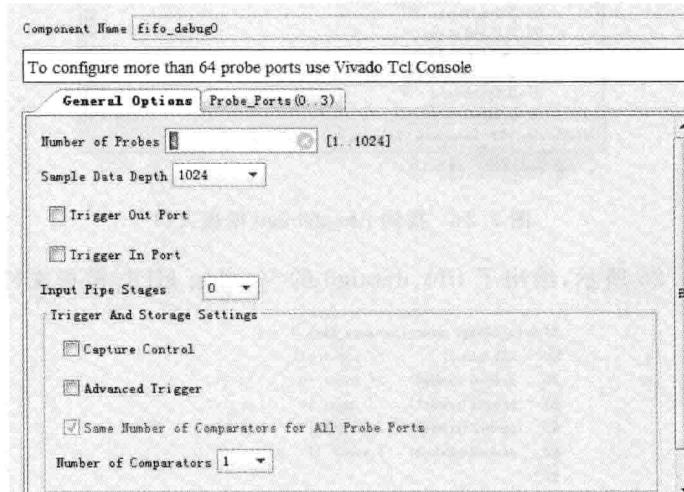


图 3.22 ILA IP 核 General Options 选项界面

- ① Number of Probes: 4;
- ② 不选中 Trigger Out Port 和 Trigger In Port 前面的复选框；
- ③ 其余按默认参数设置。

(7) 如图 3.23 所示,选择 Probe\_Ports(0..3)标签,按下面设置参数:

- ① PROBE0: 1(Probe Width);
- ② PROBE1: 1(Probe Width);
- ③ PROBE2: 8(Probe Width);
- ④ PROBE3: 8(Probe Width)。

(8) 单击 OK 按钮。

(9) 出现 Generate Output Products 对话框界面。

(10) 单击 Generate 按钮。

(11) 如图 3.24 所示,在 Sources 窗口内,选择 IP Sources 标签。在该标签界面内,找到并双击 fifo\_debug.veo。

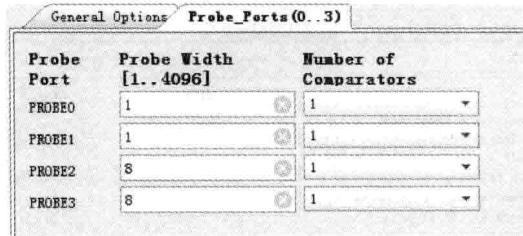


图 3.23 ILA IP 核 Probe\_Ports(0..3) 选项界面

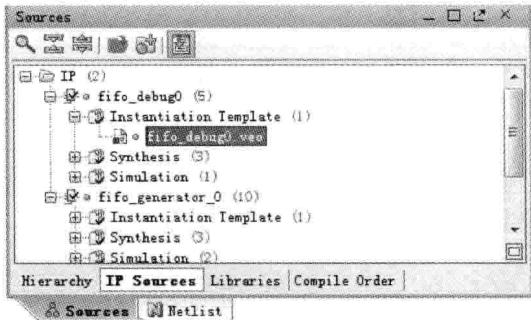


图 3.24 找到 fifo\_debug0 模板入口

(12) 如图 3.25 所示,给出了 fifo\_debug0 的 Verilog HDL 模板文件。

```

57 fifo_debug0 your_instance_name (
58   .clk(clk),           // input clk
59   .probe0(probe0),    // input [0 : 0] probe0
60   .probe1(probe1),    // input [0 : 0] probe1
61   .probe2(probe2),    // input [7 : 0] probe2
62   .probe3(probe3)    // input [7 : 0] probe3
63 );

```

图 3.25 fifo\_debug0 Verilog HDL 模板

(13) 双击打开 top.v 文件。如图 3.26 所示,在该文件中添加 fifo\_debug0 的例化语句。

```

45 wire[7:0] doutl;
46 reg[3:0] j;
47 assign dout=doutl;
48 fifo_debug0 inst_fifo_debug0 (
49   .clk(clk),           // input clk
50   .probe0(wr_en),     // input [0 : 0] probe0
51   .probe1(rd_en),     // input [0 : 0] probe1
52   .probe2(din),       // input [7 : 0] probe2
53   .probe3(doutl)      // input [7 : 0] probe3
54 );
55 fifo_generator_0 Inst_fifo1 (
56   .clk(clk),           // input clk

```

图 3.26 添加 fifo\_debug0 例化语句

(14) 在 Vivado 主界面左侧的 Flow Navigator 窗口下,找到并展开 Implementation。在展开项中,找到并单击 Run Implementation,对设计进行实现。

(15) 实现结束后,出现 Implementation Completed 对话框界面。在该对话框界面内选中 Generate Bitstream 前面的复选框。

(16) 单击 OK 按钮。等待生成比特流文件。

(17) 出现 Bitstream Generation Completed 对话框界面。在该对话框界面中,选中 Open Hardware Manager 前面的复选框。

(18) 单击 OK 按钮。

(19) 当下载完比特流文件后,出现调试器界面。

注:此时将 R7 所对应的 SW2 开关置低,表示系统当前不处于复位状态。同时,将 U8 和 U9 所对应的 SW1 和 SW0 开关置高,表示外部读写触发信号有效。

(20) 如图 3.27 所示,在 Debug Probes 窗口下,选择 rd\_en 和 wr\_en 信号,按照箭头所示方向,将其拖曳添加到 Basic Trigger Setup 窗口中。

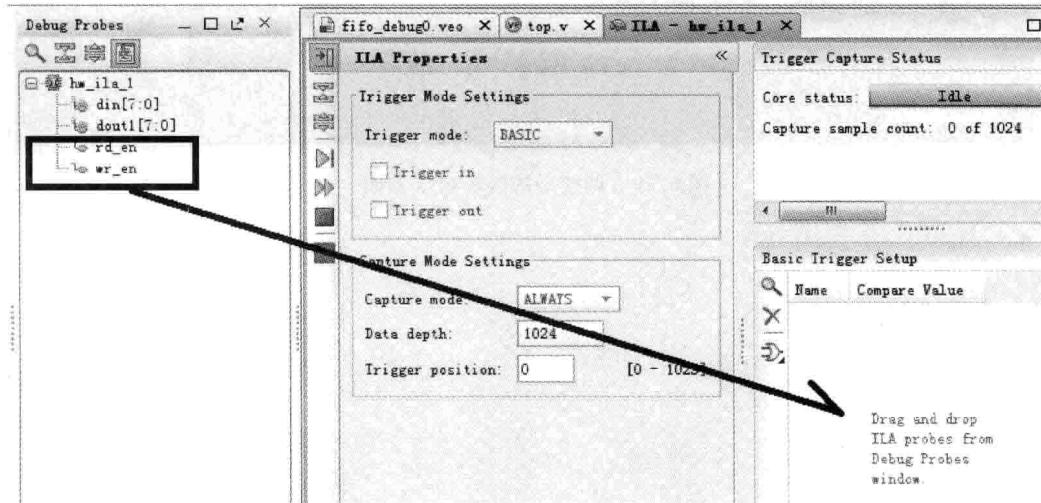


图 3.27 添加基本触发条件

(21) 如图 3.28 所示,在 Basic Trigger Setup 窗口中,添加了两个触发信号。

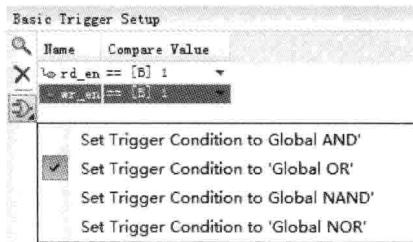


图 3.28 设置触发条件(1)

(22) 如图 3.28 所示,分别单击 rd\_en 和 wr\_en 右侧的下拉框,出现图 3.29 所示的设置触发条件对话框界面。在该界面中将 Value 右侧的值改为 1。

(23) 如图 3.28 所示,在 Basic Trigger Setup 窗口内,单击 按钮,出现浮动菜单。在浮动菜单内,选择 Set Trigger Condition to ‘Global OR’。该选项表示当 rd\_en = ‘1’

或者 wr\_en = ‘1’时，满足触发条件，用于捕获数据。

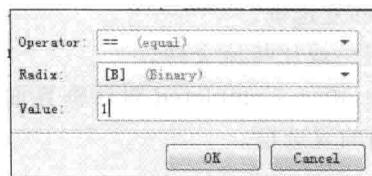


图 3.29 设置触发条件(2)

- (24) 在调试界面内，找到 ILA Properties 窗口界面。在该界面中，单击 按钮。
- (25) 单击 和 按钮，将波形调整到合适的位置，如图 3.30 所示，给出了调整后的波形。

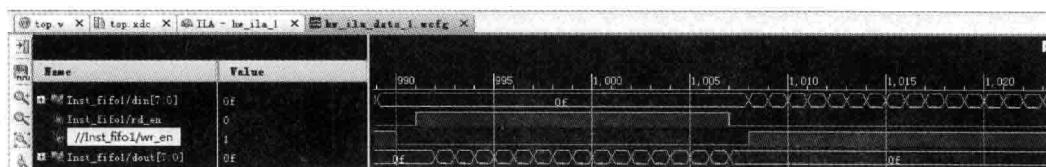


图 3.30 调整后的波形窗口界面

本章介绍 Vivado 环境下的 Xilinx Zynq-7000 SoC 嵌入式系统的设计流程。内容包括：简单硬件系统设计、在 PL 内添加外设、创建和添加定制 IP、软件控制定时器和调试，使用硬件分析仪调试。

该流程使用 Vivado 集成开发环境创建一个硬件系统，使用软件开发工具(Software Development Kit,SDK)创建应用程序验证硬件功能。

注：关于 Zynq-7000 更详细的知识，参考《Xilinx All Programmable Zynq-7000 SoC 设计指南》一书，该书由清华大学出版社出版。

图 4.1 给出了该系统实现的完整结构。本章将使用 IP 核集成器 (IP Integrator) 创建一个基于处理器系统 (Processing System, PS) 的设计，该系统主要包含：

- (1) ARM Cortex-A9(PS)；
- (2) 用于串口通信的 UART；
- (3) 用于外部 DDR3 SDRAM 的存储器控制器。

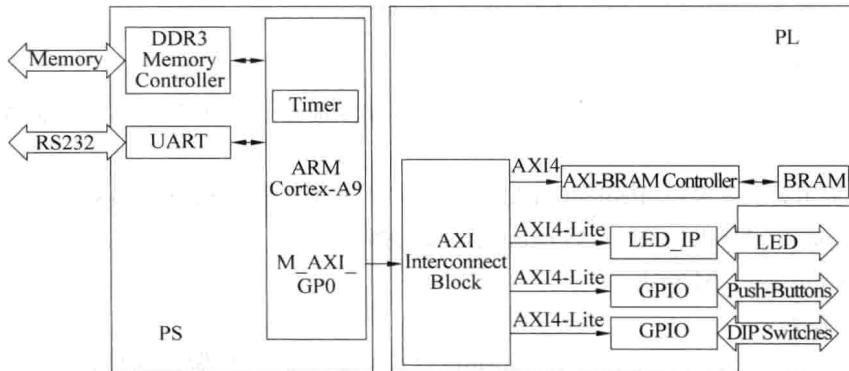


图 4.1 完整嵌入式系统结构图

## 4.1 简单硬件系统设计

本节将介绍简单硬件系统的设计。内容包括：创建新的工程、使用 IP 集成器创建处理器系统、生成顶层 HDL 和导出设计到 SDK、创建存储器测试程序、验证设计。

### 4.1.1 创建新的工程

本节将基于 Vivado 创建新的设计工程。创建新工程的步骤主要包括：

- (1) 在 Windows 7 操作系统主界面左下角选择开始→所有程序→Xilinx Design Tools→Vivado 2013.3→Vivado 2013.3。
- (2) 在 Vivado 主环境下,选择 Create New Project,启动建立新工程向导。
- (3) 出现 Create a New Vivado Project 界面,单击 Next 按钮。
- (4) 如图 4.2 所示,出现 New Project-Project Name 对话框界面,按下面参数设置:
  - ① Project name: lab1;
  - ② Project location: E:/vivado\_example/zynq;
  - ③ 选中 Create project subdirectory。



图 4.2 输入工程名字和路径界面

- (5) 单击 Next 按钮。
- (6) 出现 New Project-Project Type 对话框界面,在该界面中选中 RTL Project 前面的复选框。
- (7) 如图 4.3 所示,出现 New Project-Add Sources 对话框界面。按下面参数设置:
  - ① Target language: VHDL(对于 Verilog 读者,选择 Verilog);
  - ② Simulator language: Mixed。
- (8) 单击 Next 按钮。
- (9) 出现 New Project-Add Existing IP(optional)对话框界面。不做任何修改。
- (10) 单击 Next 按钮。
- (11) 出现 New Project-Add Constraints(optional)对话框界面。不做任何修改。
- (12) 单击 Next 按钮。
- (13) 如图 4.4 所示,出现 New Project-Default Part 对话框界面。在 Specify 下的窗口选择 Boards。在 Board 窗口下,选择 Zedboard Zynq Evaluation and Development Kit 选项。
- (14) 单击 Next 按钮。
- (15) 出现 New Project-New Project Summary 对话框界面。
- (16) 单击 Finish 按钮。

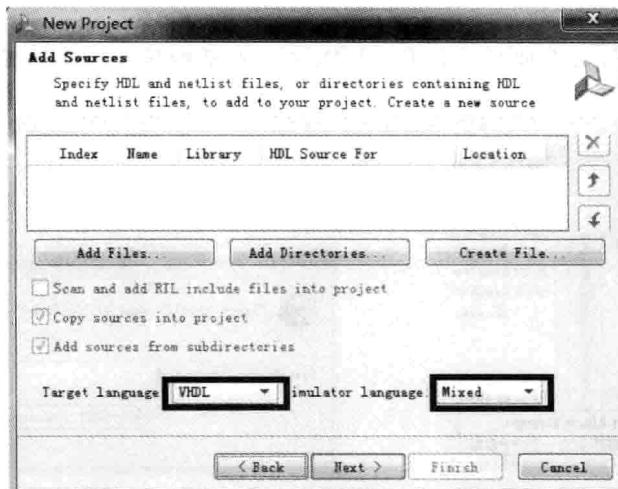


图 4.3 Add Sources 界面

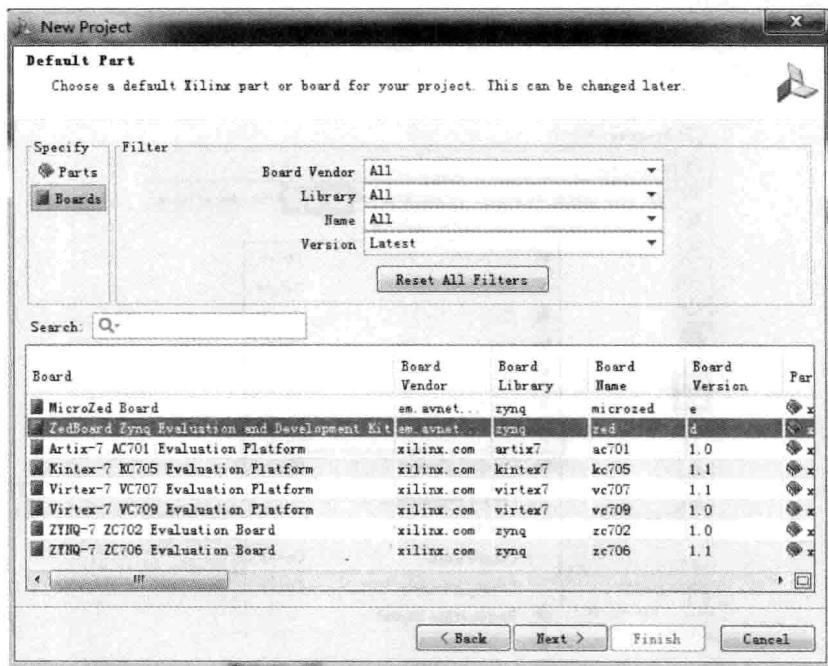


图 4.4 选择器件界面

#### 4.1.2 使用 IP 集成器创建处理器系统

本节将使用 IP 集成器创建一个新的设计块,用于生成基于 ARM Cortex-A9 处理器的硬件系统。该硬件系统用于 Xilinx XUP 提供的 Zedboard 开发平台。创建处理器系统的步骤主要包括:

- (1) 如图 4.5 所示,在 Vivado 主界面左侧的 Flow Navigator 窗口下,找到并展开 IP

Integrator。在展开项中,找到并单击 Create Block Design。

(2) 如图 4.6 所示,出现 Create Block Design 对话框界面。在该界面内,输入该设计块的名字: System。

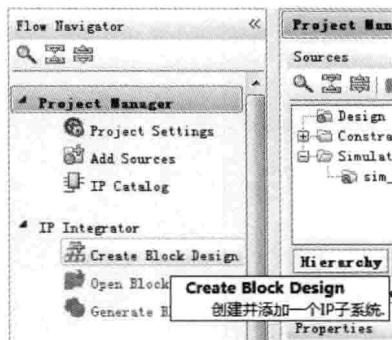


图 4.5 IP 集成器入口

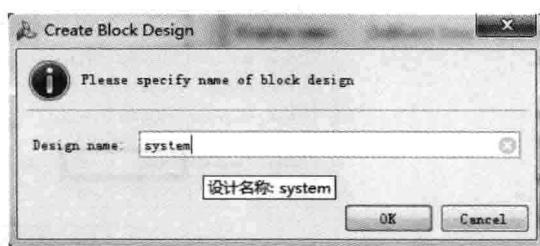


图 4.6 输入设计块的名字

(3) 单击 OK 按钮。

(4) Vivado 提供下面的方法,用于从目录中添加 IP:

① 如图 4.7 所示,在 Diagram 面板窗口最上方,单击 Add IP。

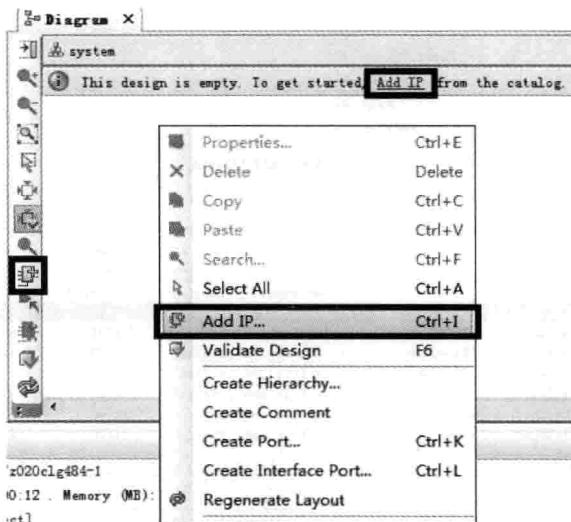


图 4.7 添加 IP 入口

② 如图 4.7 所示,在 Diagram 面板窗口左侧的一列工具栏内,单击 按钮。

③ 按 Ctrl+I 组合按键。

④ 如图 4.7 所示,在 Diagram 窗口界面内,单击右键,出现浮动菜单。在浮动菜单内,选择 Add IP。

(5) 如图 4.8 所示,出现 IP 列表界面。在 Search 右侧输入框中,输入 z,在下面的 IP 列表中,找到并双击 ZYNQ7 Processing System。或者选中该选项,然后按下 Enter 键。这样,就将 ZYNQ7 IP 添加到设计中。

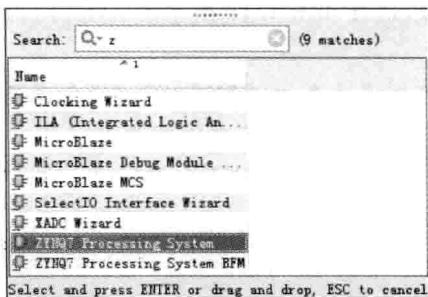


图 4.8 选择 ZYNQ 处理器系统 IP

(6) 如图 4.9 所示,在 Diagram 窗口顶部,出现消息 Designer Assistance available。单击 Run Block Automation 选项。并且,选择/processing\_system\_7\_0。

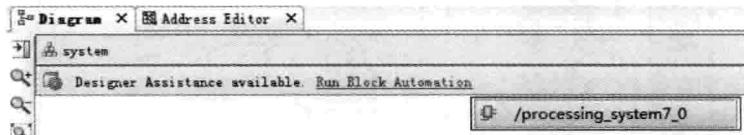


图 4.9 选择 Run Block Automation 入口

(7) 如图 4.10 所示,出现 Run Block Automation 对话框界面。

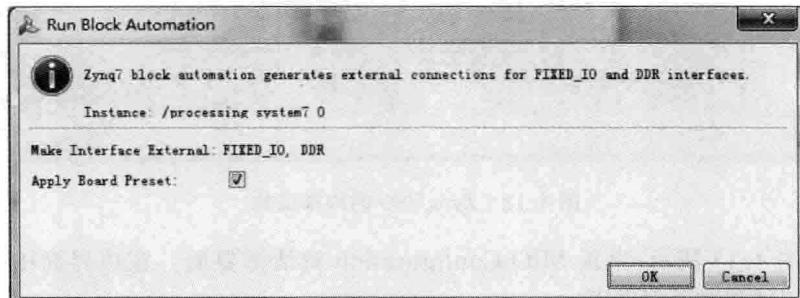


图 4.10 Run Block Automation 入口对话框界面

(8) 单击 OK 按钮。

如图 4.11 所示,当完成 Run Block Automation 时,可以看到为 DDR 和固定 IO 自动添加了端口。此外,也显示了其他可用的端口。下面准备修改 ZYNQ 的默认设置。

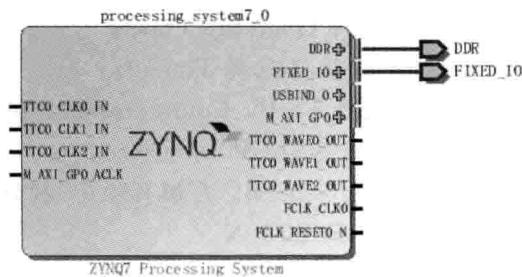


图 4.11 完成 DDR 和 FIXED\_IO

(9) 双击 ZYNQ 块符号, 打开如图 4.12 所示的 Zynq 内部结构图, 该图显示了 PS 内各种可配置的块。

注: 在这一步, 设计者可以点击各种可配置的块(绿色高亮), 修改系统配置。

注: 在该设计中只使用 UART1。下面将修改设置。

(10) 双击图 4.12 内的绿色高亮标记的 I/O Peripherals。

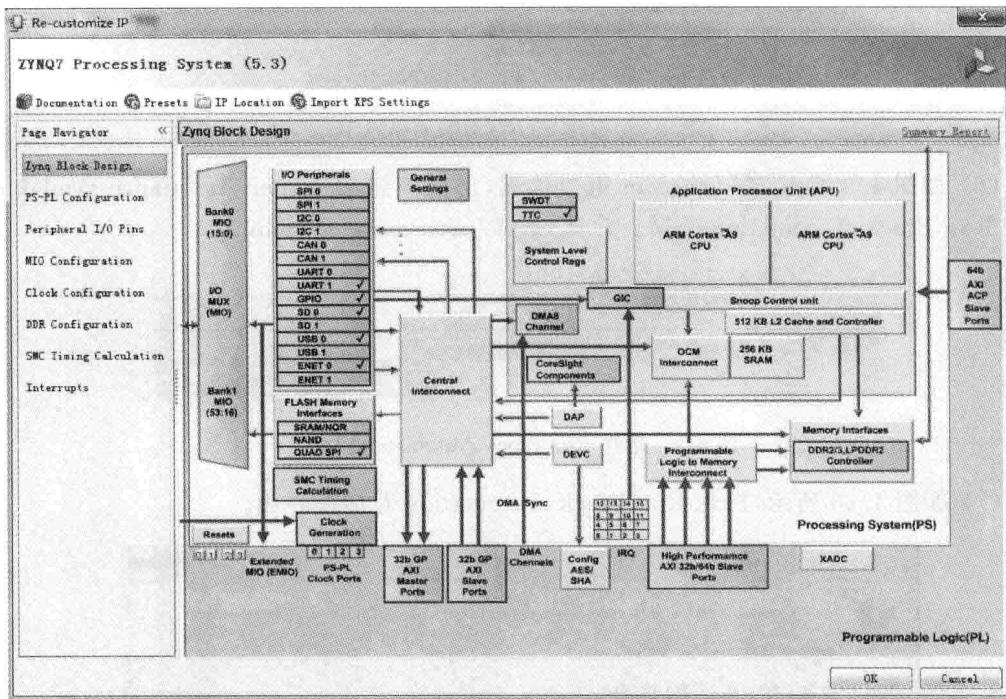


图 4.12 Zynq7000 的内部结构

(11) 如图 4.13 所示, 出现 MIO Configuration 对话框界面。在该界面中, 除了选择 UART1 外, 不选择其他的外设。

例如:

- ① 去掉 ENET0;
- ② 去掉 USB0;
- ③ 去掉 SD0;
- ④ 展开 GPIO, 不选择 GPIO MIO;
- ⑤ 展开 Memory Interfaces, 不选择 Quad SPI Flash;
- ⑥ 展开 Application Processor Unit, 不选择 Timer0。

(12) 如图 4.14 所示, 在左侧选择 PS-PL Configuration。在右侧展开 GP Master AXI Interface。在展开项中, 不选中 M AXI GP0 interface。

(13) 如图 4.15 所示, 选择并展开 General。在展开项中, 找到并展开 Enable Clock Resets。在展开项中, 不选中 FCLK\_RESET0\_N。

(14) 如图 4.16 所示, 在左侧选择 Clock Configuration。在右侧选择并展开 PL Fabric Clocks。在展开项中, 不选中 FCLK\_CLK0。

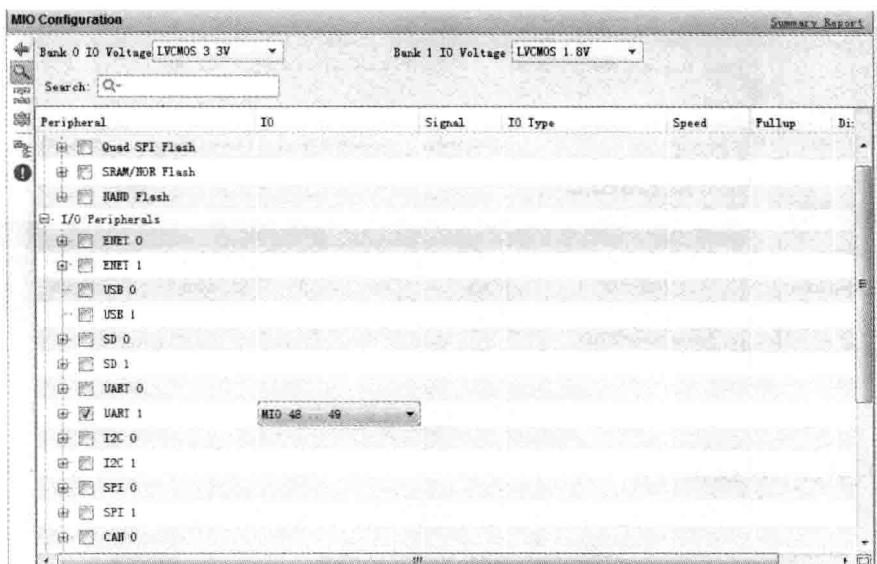


图 4.13 Zynq7000 MIO 选择界面

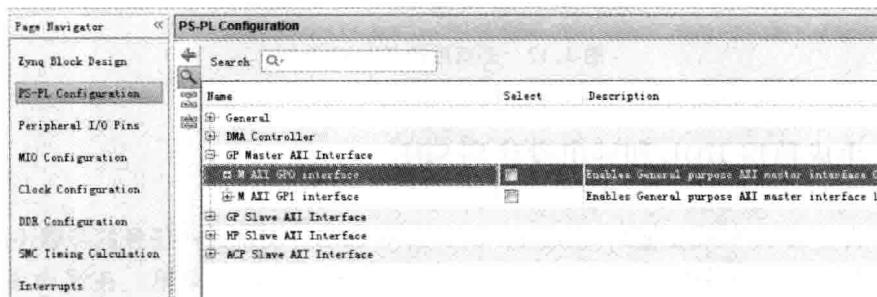


图 4.14 PS-PL 设置界面(1)

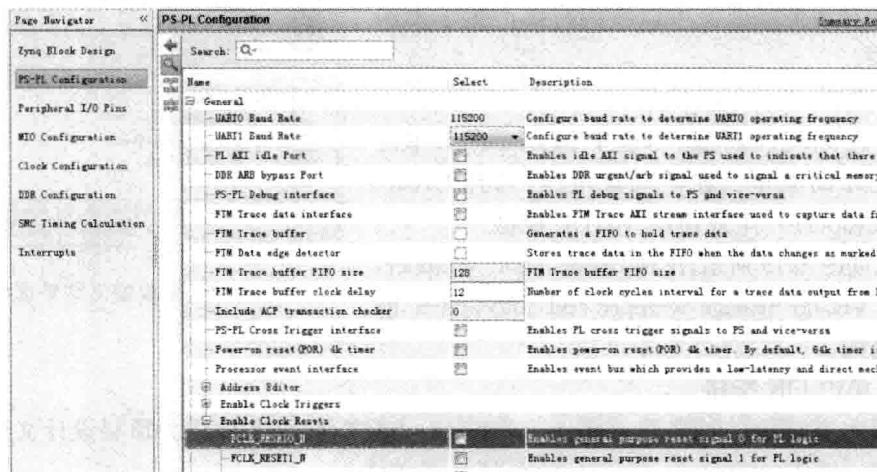


图 4.15 PS-PL 设置界面(2)

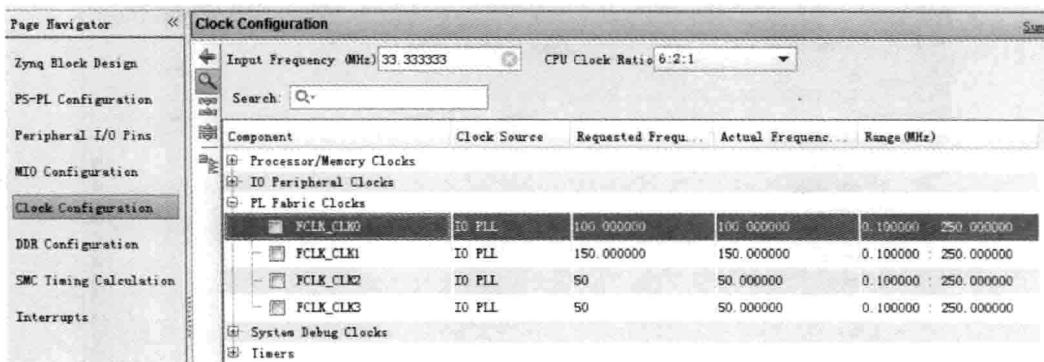


图 4.16 Clock 设置界面

(15) 单击 OK 按钮。

(16) 如图 4.17 所示,给出了更新配置后的 Zynq 嵌入式系统。

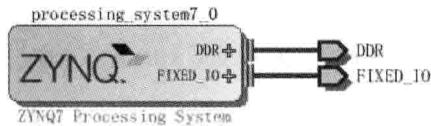


图 4.17 更新后的 ZYNQ 系统

### 4.1.3 生成顶层 HDL 和导出设计到 SDK

本节将生成 IP 集成器输出,顶层 HDL,并启动 SDK。实现这些任务的步骤主要包括:

(1) 在 Source 标签下,选中 system.bd,单击右键出现浮动菜单。在浮动菜单内,选择 Generate Output Products...。

(2) 如图 4.18 所示,出现 Generate Output Products 对话框界面。可以看到将要产生下面的输出结果。

(3) 单击 Generate 按钮。

(4) 在 Source 标签下,选中 system.bd,单击右键出现浮动菜单。在浮动菜单内,选择 Create HDL Wrapper...,生成顶层 VHDL 模型。

(5) 如图 4.19 所示,出现 Create HDL Wrapper,选中 Let Vivado manage wrapper and auto-update 前面的复选框。

(6) 单击 OK 按钮。

如图 4.20 所示,创建了 system\_wrapper.vhd 文件,该文件作为顶层设计文件,用`top`符号标记。双击 system\_wrapper.vhd,打开该文件。

**注:** 在将设计导出到 SDK 前,需要打开块设计。如果没有打开,则在 Vivado 主界面下的 Flow Navigator 下,找到并展开 IP Integrator。在展开项中,单击 Open Block Design。

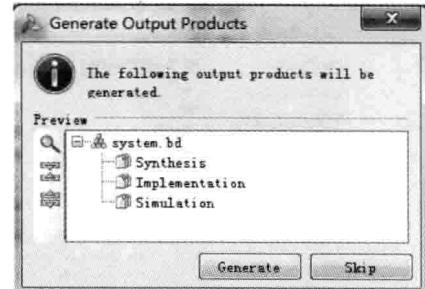


图 4.18 生成输出结果选项

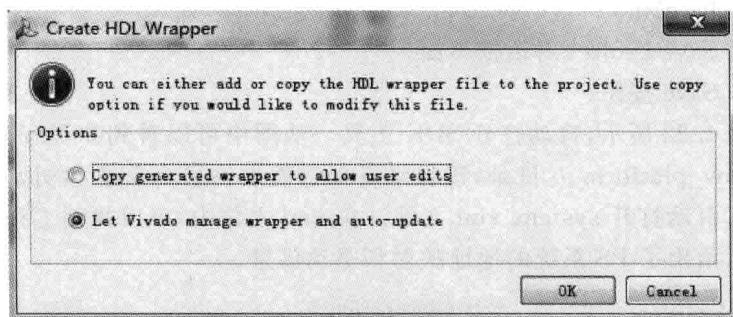


图 4.19 创建 HDL 界面

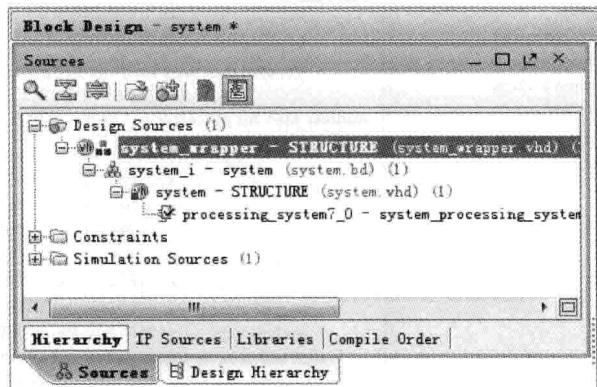


图 4.20 创建 HDL 文件

- (7) 在 Vivado 主界面主菜单下,选择 File→Export→Export hardware for SDK…。  
 (8) 如图 4.21 所示,出现 Export Hardware for SDK 对话框界面。在该界面中确认已经选中 Export Hardware 和 Lanuch SDK 前面的复选框。

**注:** 由于当前的设计,没有使用 PL 内的任何资源,所以没有生成比特流。因此,可以看到不可使用 Include bitstream 选项。



图 4.21 导出硬件到 SDK 对话框界面

- (9) 单击 OK 按钮。
- (10) 出现 Save Project 对话框界面。
- (11) 单击 Save 按钮。
- (12) 如图 4.22 所示,自动打开 SDK 工具。从图中可以看到在 Project Explorer 窗口下,创建了 hw\_platform\_0 目录,该目录下保存着硬件平台信息和初始化文件。在右侧窗口界面中,自动打开 system.xml 文件。在 xml 文件中,可以找到工程硬件配置的基本信息。同时,给出了 PS 系统的地址映射和驱动信息。

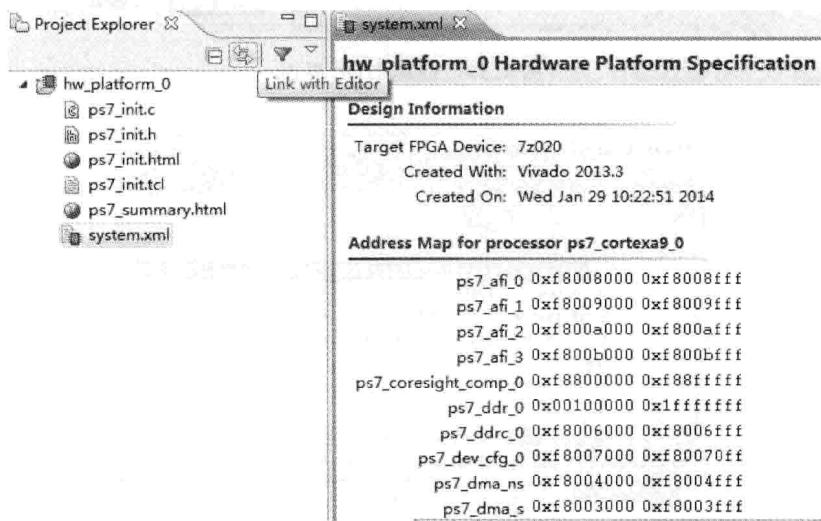


图 4.22 打开 SDK 主界面

注: 如果出现 Welcome 窗口,关闭或者最小化该窗口。

#### 4.1.4 创建存储器测试程序

本节将使用标准的工程模板,生成存储器测试应用程序。创建存储器测试程序的步骤主要包括:

- (1) 在 SDK 主界面主菜单下,选择 File→New→Application Project。
- (2) 出现 New Project 对话框界面,按下面参数设置:
  - ① Project name: mem\_test;
  - ② 选中 Create New 前面的复选框。保留默认的 mem\_test\_bsp。
  - (3) 单击 Next 按钮。
- (4) 如图 4.24 所示,出现 Now Project 对话框界面。在该界面左侧的 Available Templates 窗口下,选择 Memeory Tests。
- (5) 单击 Finish 按钮。
- (6) 将自动创建 mem\_test 工程和板支持包工程 mem\_test\_bsp。可以在 SDK 主界面的 Project Explorer 窗口下看到这两个工程目录。并且,SDK 将自动对这两个工程进行编译。在 Console 窗口内,设计者可以查看编译过程的信息。

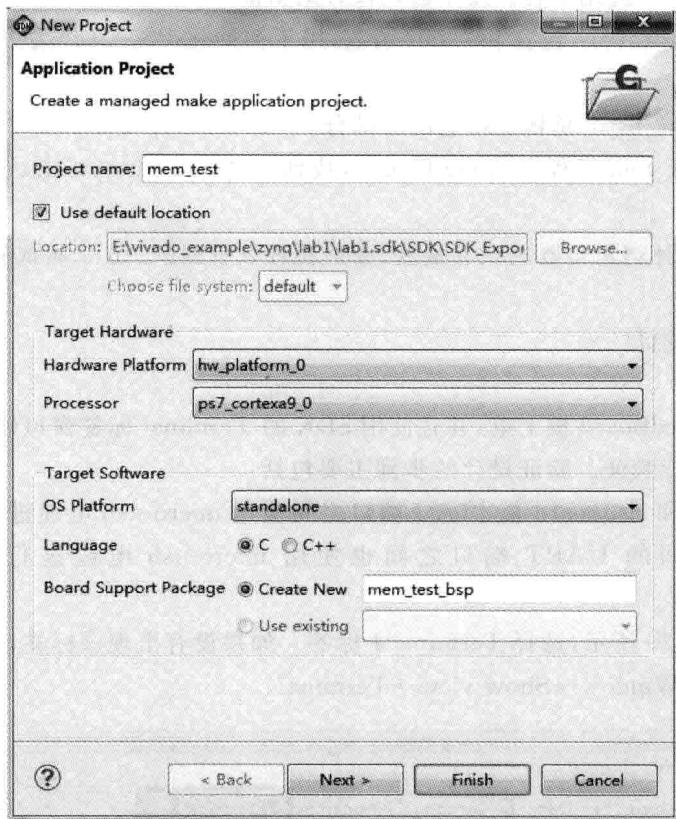


图 4.23 创建应用工程对话框界面

(7) 如图 4.25 所示,在 SDK 主界面左侧的 Project Explorer 窗口下,展开目录,查看以下三个工程: hw\_platform\_0、mem\_test\_bsp 和 mem\_test。

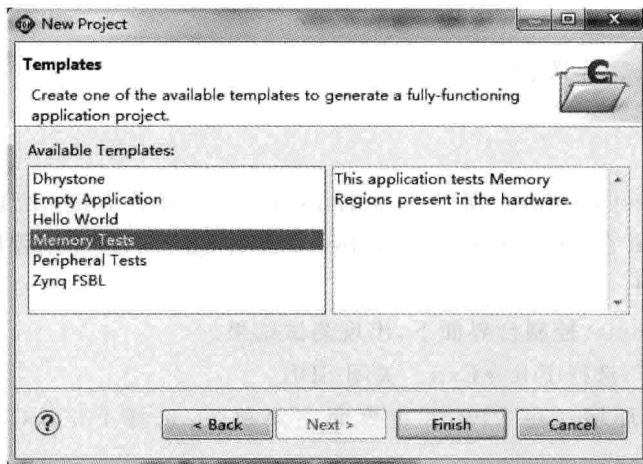


图 4.24 选择应用工程模板

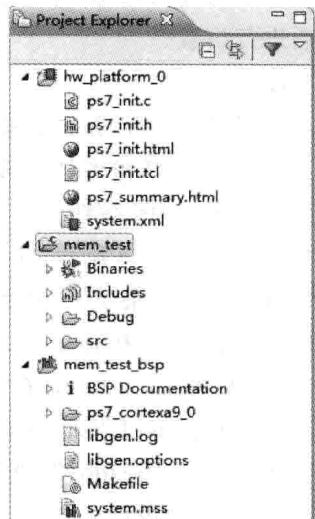


图 4.25 工程窗口

- ① mem\_test：应用工程。用于验证设计的功能。
- ② hw\_platform\_0：包含 ps7\_init 函数，用于初始化 PS。其作为第一级启动引导的一部分。
- ③ mem\_test\_bsp：是板支持包的一部分。

(8) 在 mem\_test 工程下的 src 目录下，找到并打开 memorytest.c 文件，并且查看文件内容。

注：这个文件调用函数测试存储器，分别执行 8 位测试、16 位测试和 32 位测试。

#### 4.1.5 验证设计

本节将给 ZedBoard 板上电，并且使用 SDK 的 Terminal 标签窗口建立和主机的串口通信，对设计进行验证。验证设计的步骤主要包括：

- (1) 在 PC 和 Zedboard 板 JTAG 端口之间使用 micro-usb 电缆进行连接，在 PC 和 Zedboard 板之间的 UART 端口之间也使用 micro-usb 电缆进行连接。然后，给 Zedboard 板上电。
- (2) 如图 4.26 所示，选择 Terminal 1 标签。如果没有出现该标签，则在 SDK 主界面主菜单下，选择 Window→Show view→Terminal。

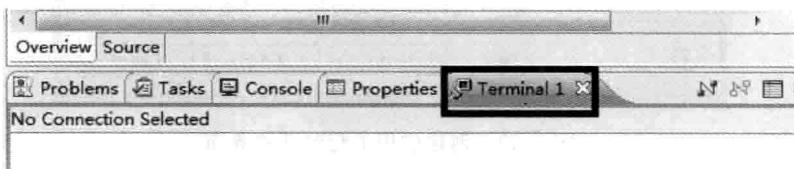


图 4.26 选择 Terminal 1 标签

- (3) 如图 4.26 所示，单击 按钮。出现图 4.27 所示的串口参数设置窗口。按图中参数进行设置。

注：

- ① 需要确保计算机成功安装驱动程序；
- ② 串口号根据主机不同而有所不同。

(4) 单击 OK 按钮。

- (5) 如图 4.28 所示，在 Project Explorer 窗口下，选择 mem\_test。单击右键，出现浮动菜单。在浮动菜单下，选择 Run As→Launch on Hardware 选项，用于下载应用程序，并且执行 ps\_init 和 mem\_test.elf。

(6) 如图 4.29 所示，在 Terminal 控制台界面下，出现测试结果。

(7) 在 SDK 主界面主菜单下，选择 File→Exit。关闭 SDK。

- (8) 进入 Vivado 集成开发环境。在 Vivado 主界面主菜单下，选择 File→Close Project，关闭该设计工程。

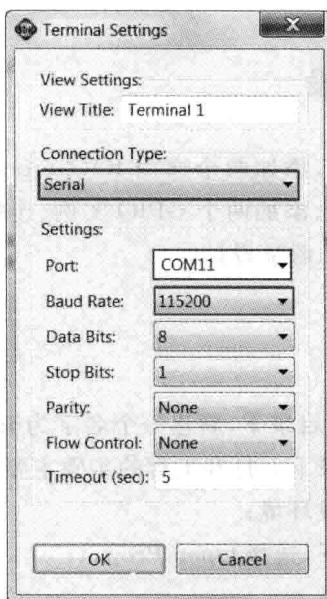


图 4.27 串口参数设置窗口

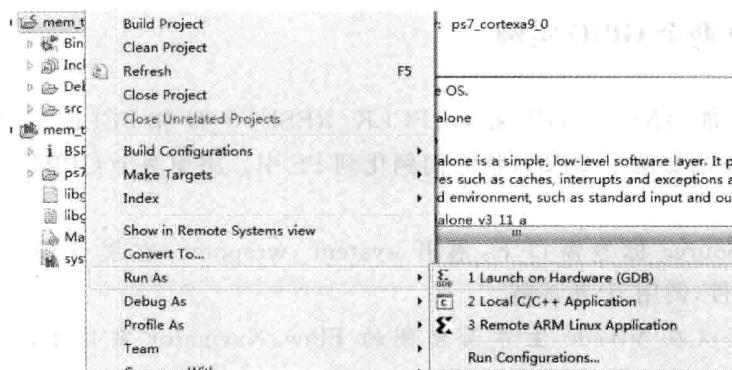


图 4.28 执行应用程序入口

```
--Starting Memory Test Application--
NOTE: This application runs with D-Cache disabled. As a result,
be generated
Testing memory region: ps7_ddr_0
    Memory Controller: ps7_ddr_
        Base Address: 0x00100000
        Size: 0x1ff00000 bytes
        32-bit test: PASSED!
        16-bit test: PASSED!
        8-bit test: PASSED!
Testing memory region: ps7_ram_1
    Memory Controller: ps7_ram_
        Base Address: 0xfffff0000
        Size: 0x0000fe00 bytes
        32-bit test: PASSED!
        16-bit test: PASSED!
        8-bit test: PASSED!
--Memory Test Application Complete--
```

图 4.29 测试结果

## 4.2 在 PL 内添加外设

本节将在基本 PS 系统上,添加两个通用 IO(Gereral Port Input/Output, GPIO)IP 核。本节内容包括: 打开工程、添加两个 GPIO 实例、连接外部 GPIO 外设、生成比特流和导出到 SDK、生成测试程序、验证设计。

### 4.2.1 打开工程

在\ vivado\_example\zynq 目录下,新建一个名字为 lab2 的目录。并且,将 lab1 目录下的所有文件复制到 lab2 目录下。打开工程的步骤主要包括:

- (1) 启动 Vivado 集成开发环境;
- (2) 在 Get Started 页面下,选择 Open Project;
- (3) 选择 Browse Project;
- (4) 定位到目录\zynq\lab2。在该目录下,选择并单击 lab1. prj。

### 4.2.2 添加两个 GPIO 实例

本节将使能 AXI\_M\_GP0 接口、FCLK\_RESET0\_N 和 FCLK\_CLK0 端口,从 IP catalog 选择并添加两个 GPIO,将它们例化到 PS 中。添加两个 GPIO 实例的步骤主要包括:

- (1) 在 Source 标签窗口下,展开 system\_wrapper。在展开项中,选择并双击 system. bd 文件,调用 IP 集成器。

**注:** 也可以在 Vivado 主界面左侧的 Flow Navigator 窗口下,找到并展开 IP Integrator。在展开项中,单击 Open Block Design。这样,也可以调用 IP 集成器工具。

- (2) 在 Diagram 窗口下,双击 ZYNQ 块图符号,打开 Zynq 配置窗口。
- (3) 在配置界面左侧,找到 Zynq Block Design。在右侧可以看到 Zynq 的内部配置界面。如图 4.30 所示,单击 32b GP AXI Master Ports 符号块。

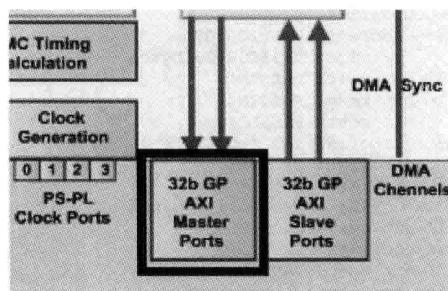


图 4.30 AXI 模块符号

(4) 如图 4.31 所示, 出现 PS-PL Configuration 界面。在该界面中, 选中 AXI GP0 interface 后面的复选框。此外, 可以展开 M AXI GP0 interface, 查看其参数配置。

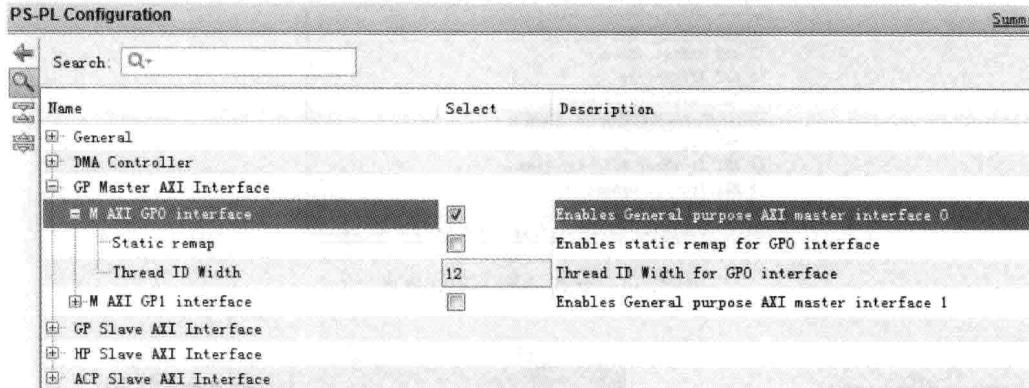


图 4.31 AXI 模块配置界面

(5) 展开图 4.31 内的 General→Enable Clock Resets 条目。在展开项中, 选中 FCLK\_RESET0\_N 后面的复选框。

(6) 选择页面左侧的 Clock Configuration 标签。在右侧窗口中, 找到并展开 PL Fabric Clock。在展开项中找到并选中 FCLK\_CLK0 后面的复选框(设时钟所对应的频率为 100MHz)。

(7) 单击 OK 按钮。

(8) 如图 4.32 所示, 可以看到新添加了 M\_AXI\_GP0 端口, 还新添加了 M\_AXI\_GP0\_ACLK、FCLK\_CLK0 和 FCLK\_RESET0\_N 端口。

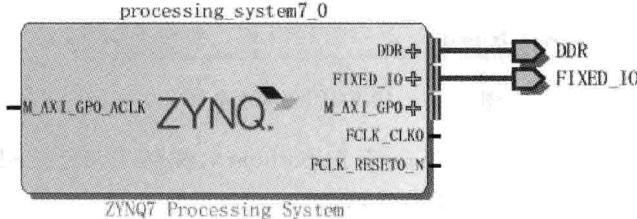


图 4.32 新的 ZYNQ 符号界面

注: 读者可以单击 Diagram 窗口左侧一列工具栏内的 按钮, 重新绘制 ZYNQ 符号。

(9) 单击 Diagram 左侧一列工具栏内的 按钮。

(10) 如图 4.33 所示, 在 Search 后侧输入 g。在下面窗口中找到 AXI GPIO IP 核。

(11) 双击 AXI GPIO 条目, 将其添加到设计中。如图 4.34 所示, 可以看到 AXI GPIO 已经添加到设计中。

(12) 单击图 4.34 内的 axi\_gpio\_0 块符号。如图 4.35 所示, 在 Block Properties 窗口内, 将 Name 后面的名字改为 sw\_8bit。

(13) 双击图 4.34 内的 axi\_gpio\_0 块图标, 打开 Re-customize IP 对话框界面。

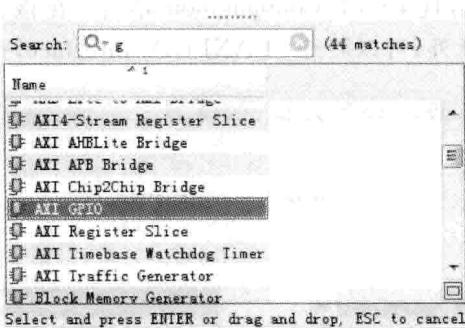


图 4.33 查找 AXI GPIO 界面

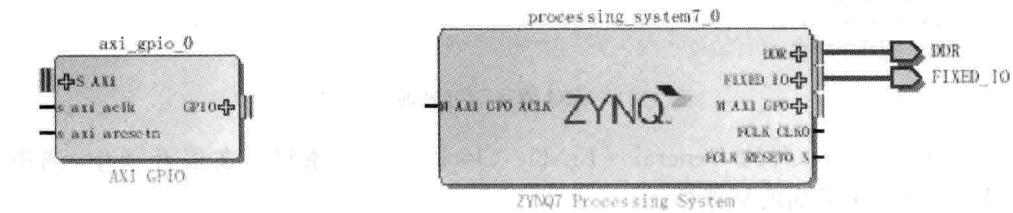


图 4.34 添加 AXI GPIO 到设计中

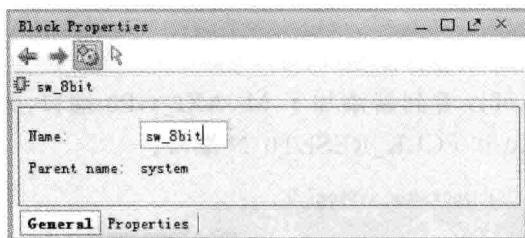


图 4.35 AXI GPIO 属性窗口

**注：**由于在前面建立工程时，已经选择了 Zedboard，提供了用于 Zedboard 的板级支持包，所以 Vivado 知道板子可以利用的资源。

(14) 如图 4.36 所示，按下面参数设置：

- ① 选中 Generate Board based IO Constraints；
- ② 在 GPIO 右侧的下拉框中选择 sws\_8bits。

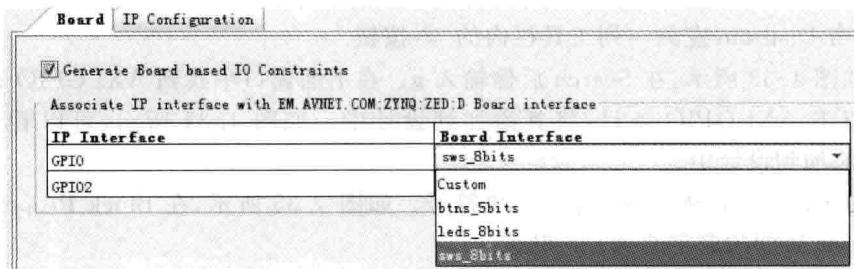


图 4.36 修改 AXI GPIO 属性(1)

(15) 如图 4.37 所示,单击 IP Configuration。按如下参数设置:

- ① GPIO Width: 8;
- ② 不选中 Enable Dual Channel 前面的复选框;
- ③ 不选中 Enable Interrupt 前面的复选框。

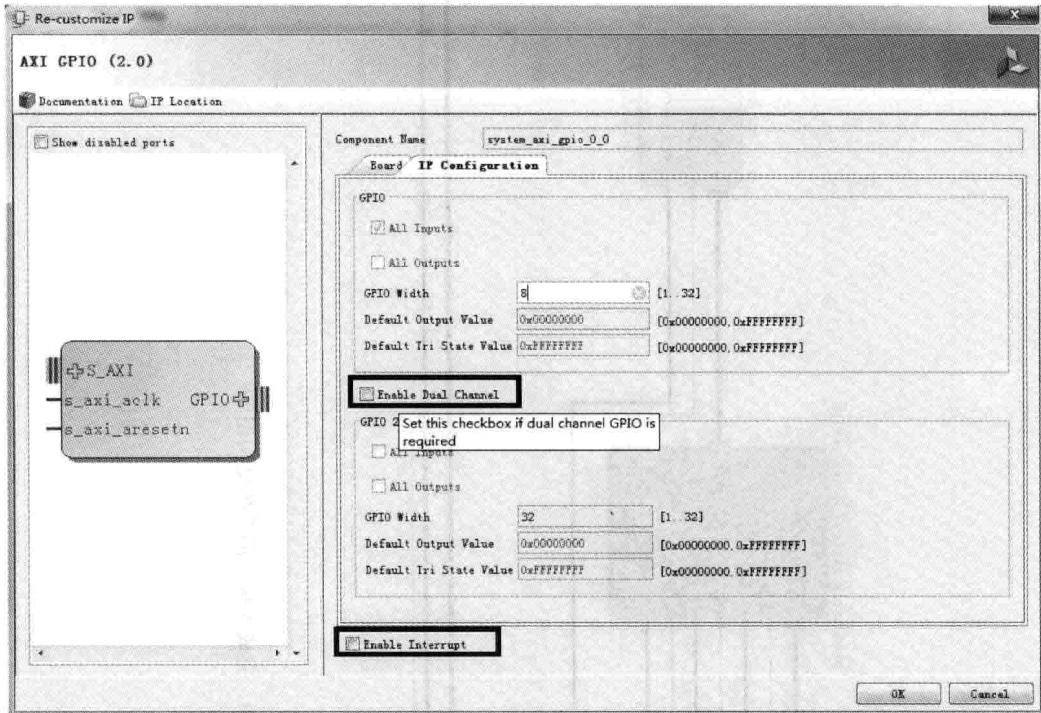


图 4.37 修改 AXI GPIO 属性(2)

(16) 单击 OK 按钮。

(17) 在 Diagram 窗口最上面出现 Designer Assistance is available 消息。单击该消息右侧的 Run Connection Automation 条目,并且选择/sw\_8bit/S\_AXI。

(18) 如图 4.38 所示,出现 Run Connection Automation 对话框界面。

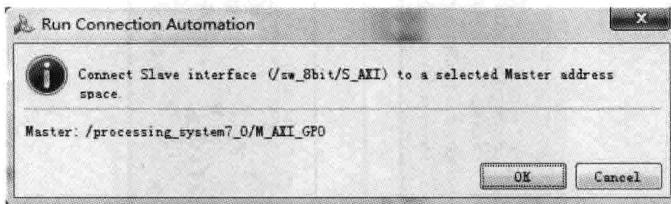


图 4.38 连接提示

(19) 单击 OK 按钮。

(20) 如图 4.39 所示,可以看到连接完成后的系统结构。在该图中,新添加了两个模块,即:

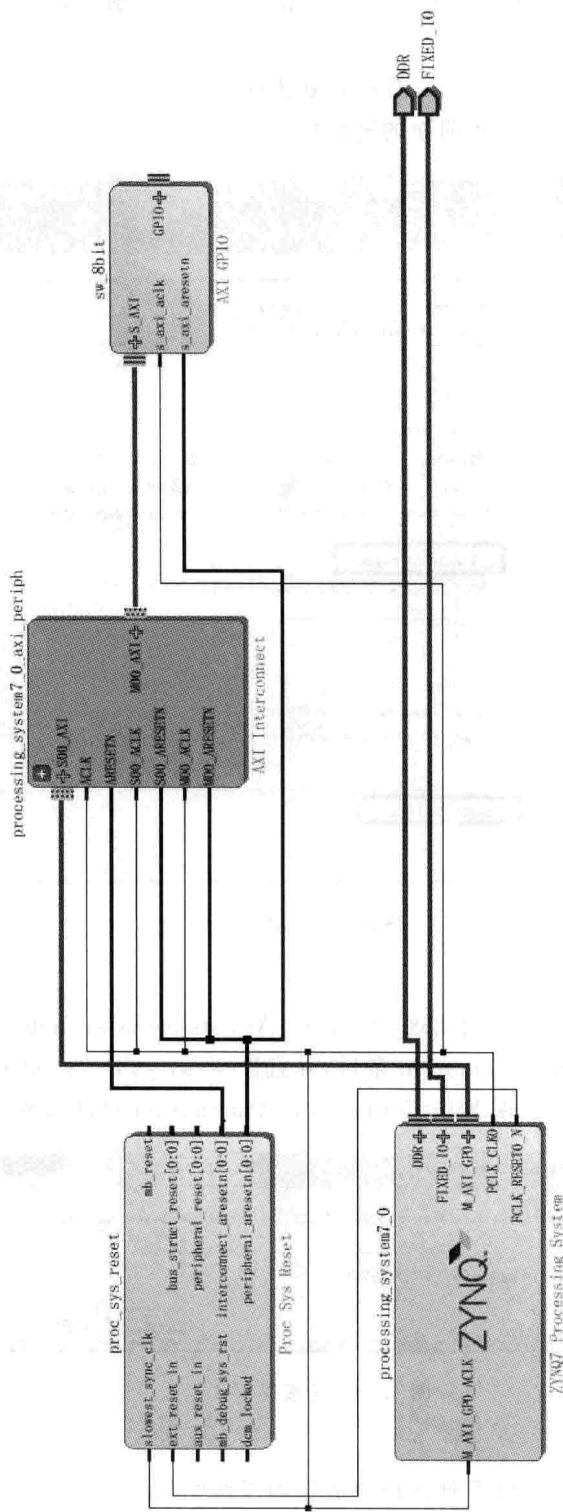


图 4.39 完成连接后的系统结构图

① Proc Sys Reset;

② AXI Interconnect。

注：可以调整和重新放置图中每个模块的位置。图 4.39 给出的结构是调整模块位置后的图。

(21) 单击 Diagram 左侧一列工具栏内的 按钮。

(22) 出现 IP Catalog 对话框界面，在该界面的 Search 右侧输入 g。在下面窗口中找到 AXI GPIO IP 核。

(23) 双击 AXI GPIO，将其添加到设计中。如图 4.40 所示，可以看到 AXI GPIO 已经添加到设计中。

(24) 单击图中内新添加的 AXI GPIO 模块符号。按前面的方法，在 Block Properties 窗口内，将 Name 后面的名字改为 btns\_5bit。

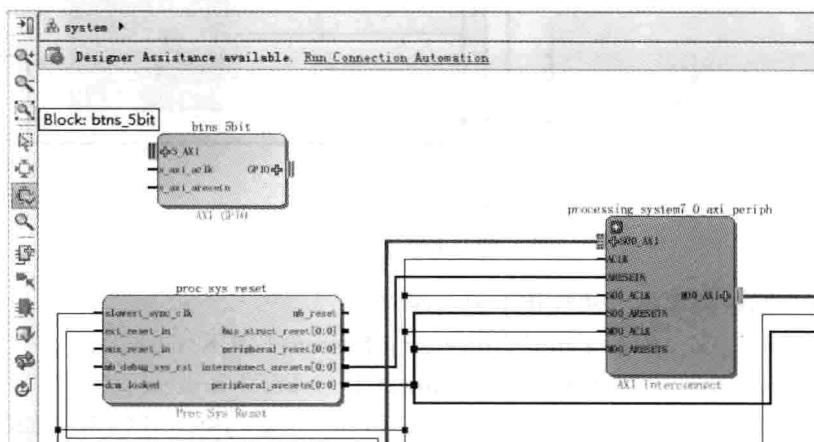


图 4.40 添加另一个 AXI GPIO 到设计中

(25) 双击图 4.40 内的 AXI Interconnect 块图标，打开如图 4.41 所示的 Recustomize IP 对话框界面。在该对话框界面的 Top Level Settings 标签窗口下，将 Number of Master Interfaces 的值改为 2。

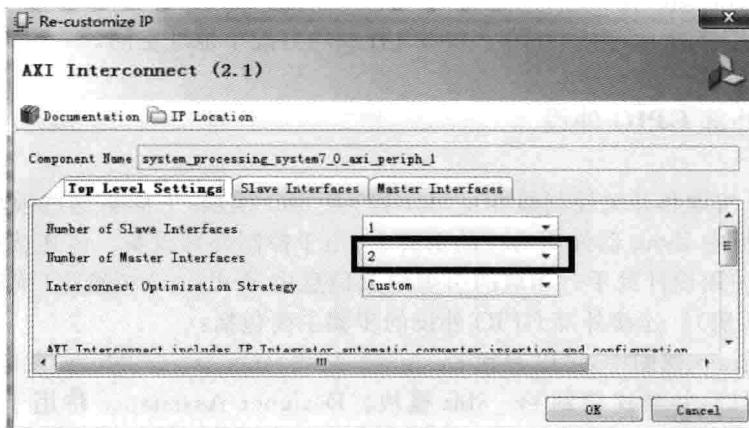


图 4.41 AXI Interconnect 对话框界面

(26) 单击 OK 按钮。

(27) 如图 4.42 所示,单击 btms\_5bit 模块符号内的 S\_AXI 端口,出现一个指针箭头,拖曳并一直按下左键。出现 Found 1 interface 消息,同时在 AXI Interconnect 的 M01\_AXI 端口右边出现一个勾,表示这是一个有效的连接端口。拖曳鼠标到这个端口,然后释放鼠标按键。此时 btms\_5bit 模块和 AXI 端口相连接。

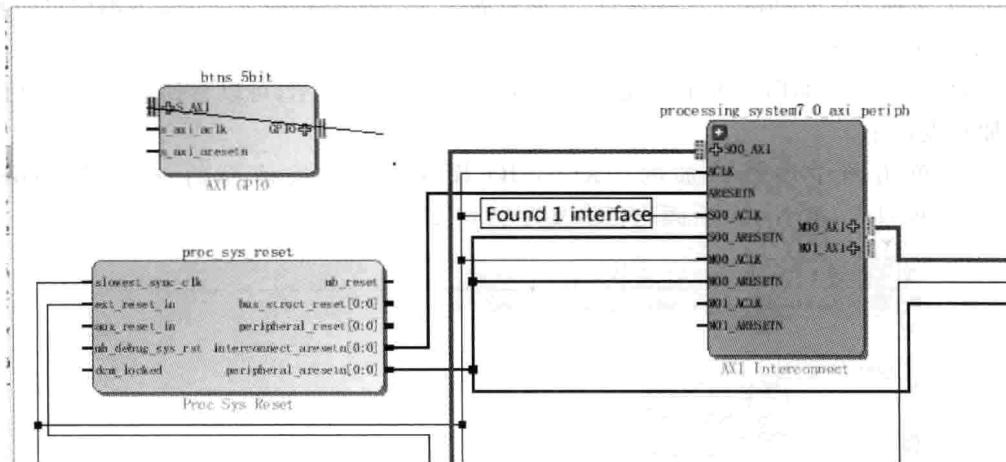


图 4.42 手工连接模块

(28) 类似地,按前面的方法连接下面的端口:

- ① btms\_5bit s\_axi\_aclk→Zynq7 Processing System FCLK\_CLK0;
- ② btms\_5bit s\_axi\_aresetn→Proc Sys Reset Peripheral\_aresetn;
- ③ AXI Interconnect M01\_ACLK→Zynq7 Processing System FCLK\_CLK0;
- ④ AXI Interconnect M01\_ARESETN→Proc Sys Reset peripheral\_aresetn。

图 4.43 给出了手工连接完成后的系统结构图。

(29) 如图 4.44 所示,选择 Address Editor 标签。在该标签窗口中,展开 processing\_system7\_0→Data→Unmapped Slaves。可以看到已经为 sw\_8bit 模块自动分配了地址,但是并没有为 btms\_5bit 模块分配地址。选中 btms\_5bit,单击右键,出现浮动菜单。在浮动菜单内,选择 Assign Address。

(30) 如图 4.45 所示,为所有两个 GPIO 外设均分配了地址空间。

### 4.2.3 连接外部 GPIO 外设

前面添加了按键和开关控制器 btms\_5bit 和 sw\_8bit 模块,下面将这些模块的端口连接到 Zedboard 板中 Zynq 器件所对应的引脚上,用于控制外部设备。该连接过程可以手工完成,也可以使用设计助手。相应的引脚约束信息由 Zedboard 板给出(读者可以查看 Zedboard 用户手册)。连接外部 GPIO 外设的步骤主要包括:

(1) 在 Diagram 视图中,可以看到 Designer Assistance is available。忽略该提示,将手工创建一个端口,并且连接到 sw\_8bit 模块。Designer Assistance 将用于连接 btms\_5bit 模块。

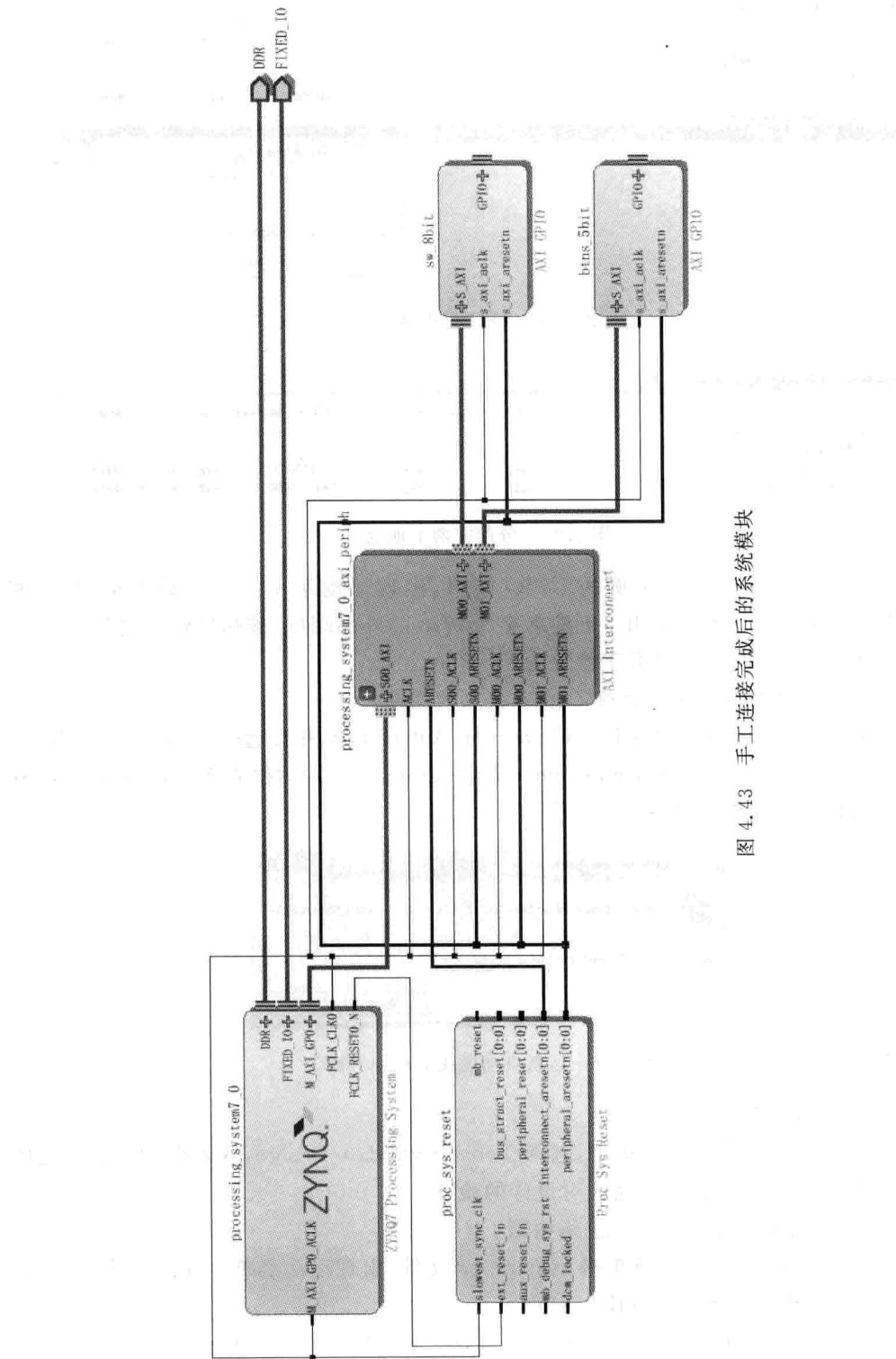


图4.43 手工连接完成后的系统模块

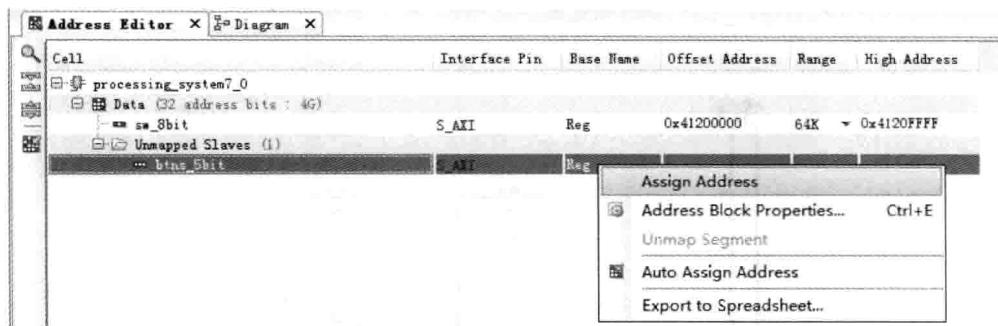


图 4.44 手工分配地址

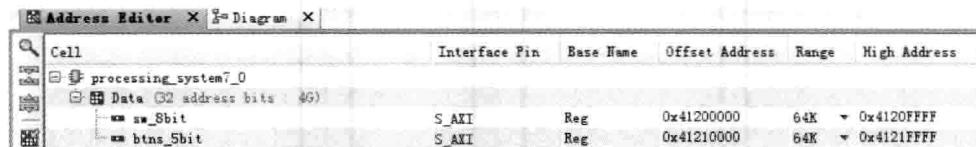


图 4.45 分配完两个地址

(2) 右键单击 sw\_8bit 实例的 GPIO 端口, 出现浮动菜单。在浮动菜单内, 选择 Make External 选项。该选项用于创建名字为 gpio 的外部端口, 该端口将连接到外设。

(3) 选择 gpio 端口, 将其名字修改为 sw\_8bit。

**注:** 由上面模块的宽度自动确定 gpio 接口的宽度。

(4) 单击 Diagram 窗口中的 Run Connection Automation, 并且选择 /btms\_5bit/GPIO。

(5) 如图 4.46 所示, 出现 Run Connection Automation 对话框界面。在 Select Board Interface 右侧的下拉框中选择 btms\_5bits。

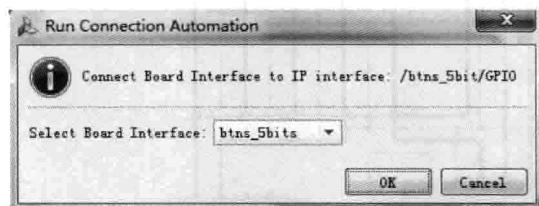


图 4.46 运行自动连接对话框界面

(6) 单击 OK 按钮。

(7) 在 Vivado 主界面主菜单中, 选择 Tools→Validate Design, 或者在 Diagram 左侧的一列工具栏内单击 按钮。完成对设计的验证。

图 4.47 给出了设计完成的系统结构图。

(8) 在 Source 窗口内, 右键单击 system.bd 文件, 出现浮动菜单。在浮动菜单内, 选择 Create HDL Wrapper, 更新 HDL 文件。

(9) 出现 Create HDL Wrapper 对话框界面, 选中 Let Vivado manage wrapper and auto-update 前面的复选框。

(10) 单击 OK 按钮。

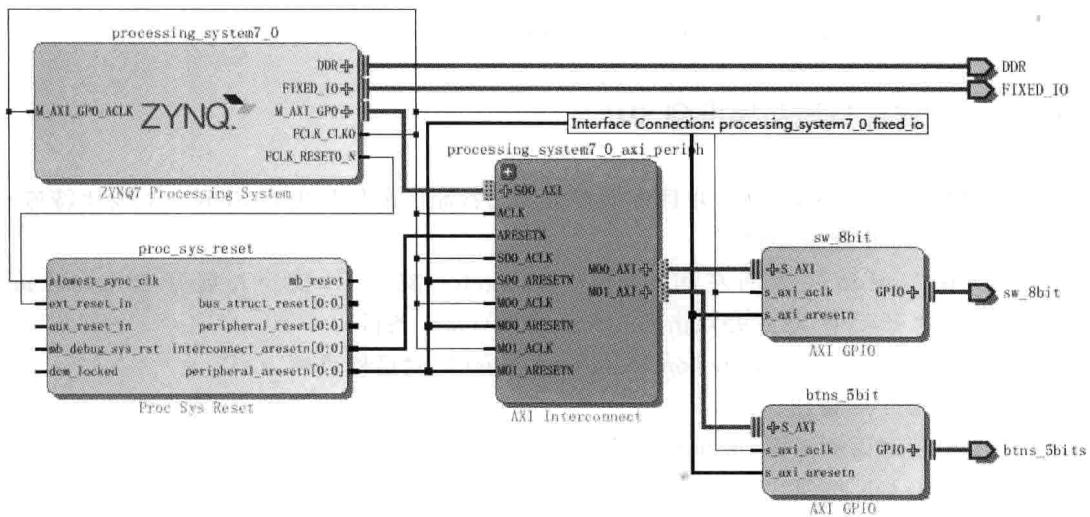


图 4.47 系统设计完成后的结构图

(11) 在 Viva0 主界面左侧的 Flow Navigator 窗口下,找到并展开 Synthesis。在展开项中,找到并双击 Run Synthesis。

(12) 出现 Save Project 对话框界面。

(13) 单击 Save 按钮。

(14) 当综合完成后,出现对话框,选择 Open Synthesized Design 选项。

(15) 单击 OK 按钮。

(16) 如图 4.48 所示,在快捷栏下拉框下,选择 I/O Planning。

(17) 如图 4.49 所示,在 I/O Port 标签窗口下,展开 btms\_5bit\_tri\_i,可以看到已经为该控制器分配了引脚。引脚信息也包含在板级支持包内。当 IP 自动连接到端口时,自动分配引脚。从图中可以看到,也自动分配了 sw\_8bit\_tri\_i 引脚的位置。

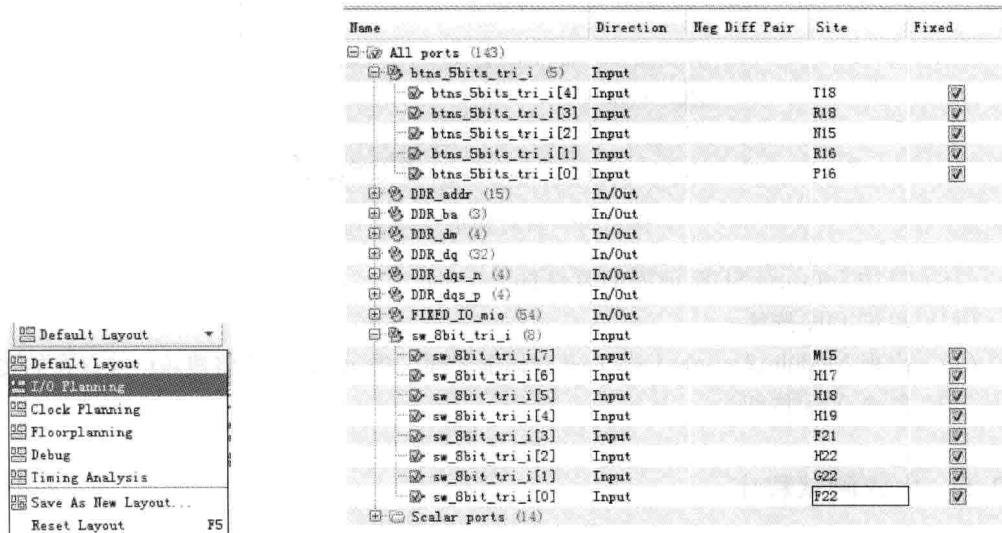


图4.48 I/O引脚约束入口

图 4.49 I/O 引脚约束界面

(18) 将窗口再次切换到 Default Layout。

#### 4.2.4 生成比特流和导出到 SDK

本节将生成比特流文件，并且将带有生成比特流的硬件导出到 SDK。生成比特流和导出到 SDK 的步骤主要包括：

(1) 在 Vivado 主界面左侧的 Flow Navigator 窗口下，找到并展开 Program and Debug。在展开项中，找到并双击 Generate Bitstream 条目。

(2) 出现 No Implementation Results Available 对话框界面。

(3) 单击 Yes 按钮。

(4) 出现对话框界面，选择 Open Implemented Design 选项。

(5) 单击 OK 按钮。

**注：**在导出硬件到 SDK 前，需要打开块设计和实现后的设计。

(6) 出现 Vivodo 对话框界面。

(7) 单击 Yes 按钮。

(8) 在 Vivado 主界面主菜单下，选择 File→Export→Export Hardware for SDK。

(9) 如图 4.50 所示，出现 Export Hardware for SDK(为 SDK 导出硬件)对话框界面。

**注：**由于在 PL 内构建了 GPIO，所以需要生成比特流。默认地，选中 Include bitstream 前面的复选框。

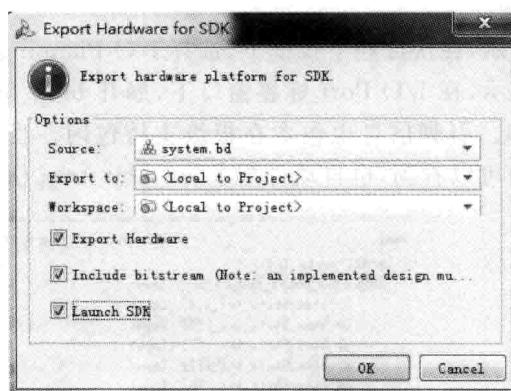


图 4.50 启动 SDK 入口界面

(10) 选中 Lanuch SDK 前面的复选框。

(11) 单击 OK 按钮。

(12) 出现 Module Already Exported 对话框界面。提示：该过程将覆盖已经存在的文件。

(13) 单击 Yes 按钮。

#### 4.2.5 生成测试程序

本节将生成带有默认设置和默认名字的软件应用工程。生成测试程序的步骤主要

包括：

- (1) 在 SDK 主界面左侧的 Project Explorer 窗口下，分别选中 mem\_test 和 mem\_test\_bsp。单击右键，出现浮动菜单。在浮动菜单内，选择 Delete(该操作将删除前面生成的应用工程)。
- (2) 出现 Delete Resources 对话框界面。在该界面选中 Delete project contents on disk(cannot be undone)前面的复选框(表示彻底删除这两个工程)。
- (3) 单击 OK 按钮。
- (4) 在 SDK 主界面主菜单下，选择 File→New→Board Support Package。
- (5) 如图 4.51 所示，出现 New Board Support Package Project 对话框界面。在该界面中，将 Project name 后面的名字改为 standalone\_bsp。
- (6) 单击 Finish 按钮。
- (7) 出现 Board Support Package Settings 对话框界面。
- (8) 单击 OK 按钮。
- (9) 自动编译 BSP 工程。
- (10) 在 SDK 主界面主菜单下，选择 File→New→Application Project。

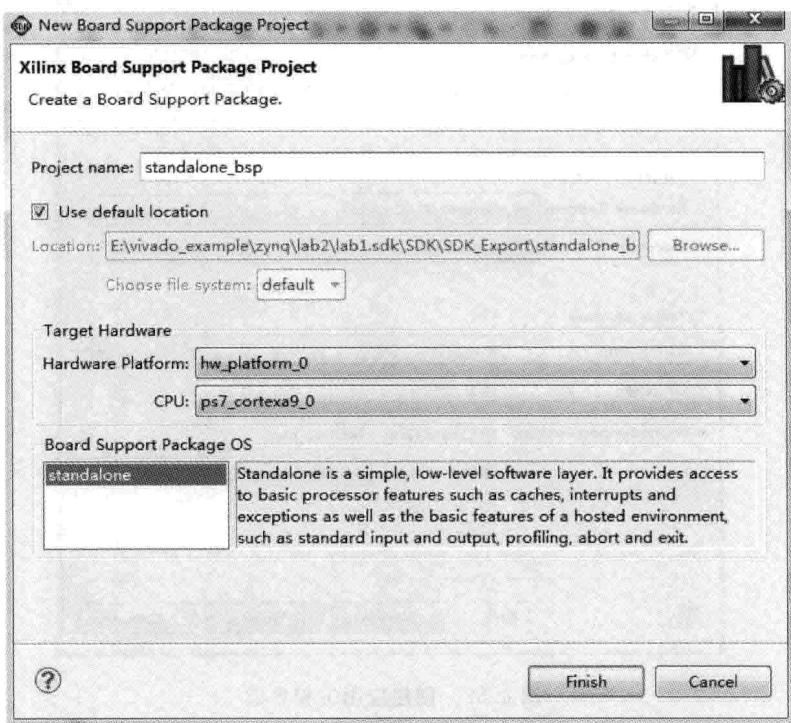


图 4.51 创建 BSP 界面

- (11) 如图 4.52 所示，出现 New Project 对话框界面。在该界面下按下面设置参数：
  - ① Project name：TestApp；
  - ② 在 Board Support Package 标题栏下，选中 Use existing 前面的复选框。
- (12) 单击 Next 按钮。

(13) 出现 New Project-Template 对话框界面。在 Available Templates 下面选择 Empty Application。

(14) 单击 Finish 按钮。

(15) 在后台运行库生成器。并且,在下面给出的路径中生成 xparameter.h 文件:

e:\vivado\_exmaple\zyng\lab2\lab1.sdk\SDK\SDK\_Export\standalone\_bsp\ps7\_cortex9\_0\include

(16) 在 SDK 主界面的 Project Explorer 窗口下,展开 TestApp→src。选中 src,单击右键,出现浮动菜单。在浮动菜单内,选择 Import。

(17) 出现 Import 对话框界面。在该对话框界面中,展开 General。在展开项中,选择 File System。

(18) 单击 Next 按钮。

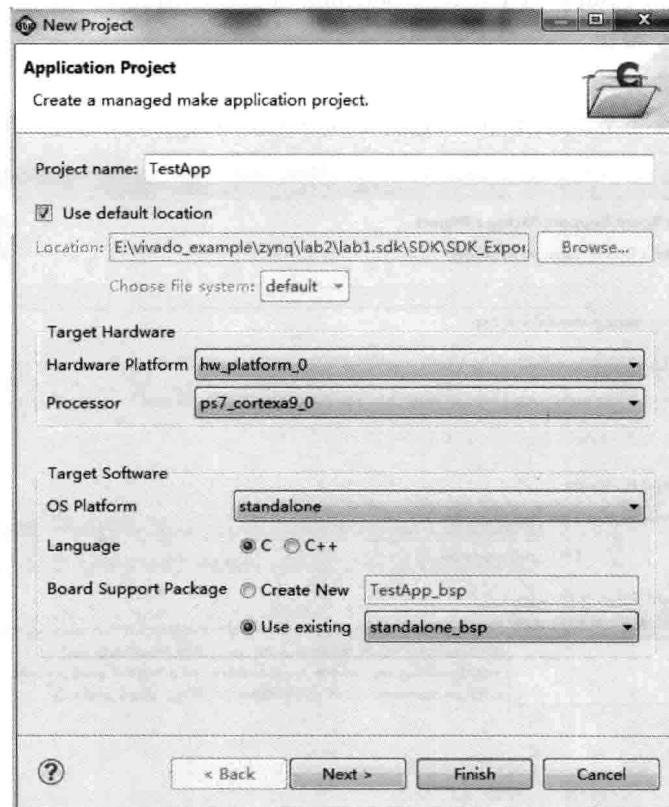


图 4.52 创建应用工程界面

(19) 如图 4.53 所示,出现 Import(导入)对话框界面。在该界面内,单击  按钮。将路径定位到:

E:\vivado\_exmaple\zyng\source

注: 该路径下保存着设计中所需要的源文件。

在图 4.53 中,选中 lab2.c 前面的复选框。

- (20) 单击 Finish 按钮。
- (21) 如图 4.54 所示,给出了 lab2.c 源文件代码清单。

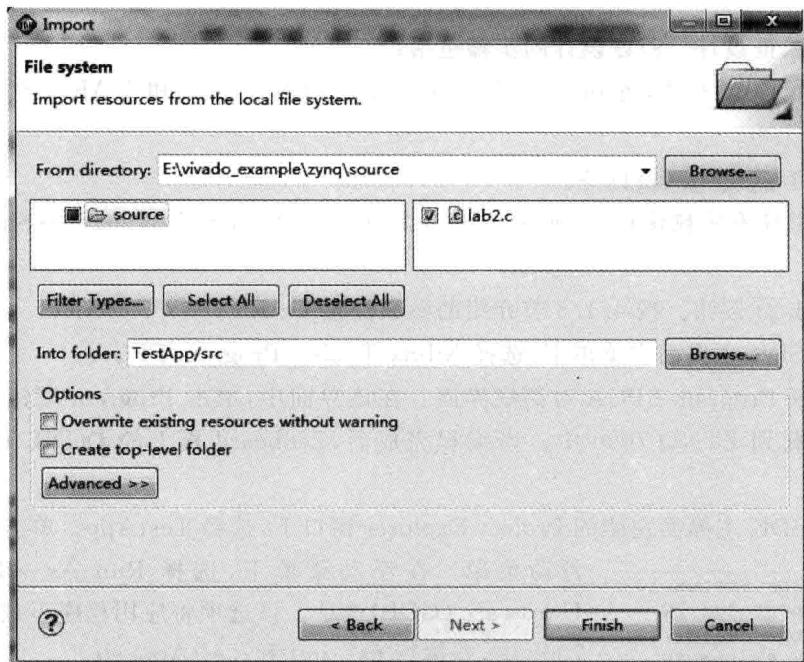


图 4.53 定位源文件界面

```

system.xml system.mss lab2.c
#include "xparameters.h"
#include "xgpio.h"
#include "xutil.h"

//=====
int main (void)
{
    XGpio dip, push;
    int i, psb_check, dip_check;

    xil_printf("-- Start of the Program --\r\n");
    XGpio_Initialize(&dip, XPAR_SW_8BIT_DEVICE_ID);
    XGpio_SetDataDirection(&dip, 1, 0xffffffff);

    XGpio_Initialize(&push, XPAR_BTNS_5BIT_DEVICE_ID);
    XGpio_SetDataDirection(&push, 1, 0xffffffff);

    while (1)
    {
        psb_check = XGpio_DiscreteRead(&push, 1);
        xil_printf("Push Buttons Status %x\r\n", psb_check);
        dip_check = XGpio_DiscreteRead(&dip, 1);
        xil_printf("DIP Switch Status %x\r\n", dip_check);

        for (i=0; i<9999999; i++);
    }
}

```

图 4.54 lab2.c 源文件

#### 4.2.6 验证设计

本节将验证设计。验证设计的步骤包括：

(1) 按照前面方法,连接PC和Zedboard之间的JTAG和UART电缆,并且给Zedboard板上电。

(2) 选择 标签。

**注:**如果没有出现该标签,则在SDK主界面主菜单下,选择Window→Show view→Terminal。

(3) 单击 按钮。按4.1.5节介绍的参数配置UART。

(4) 在SDK主界面主菜单下,选择Xilinx Tools→Program FPGA。

(5) 出现Program FPGA对话框界面。在该界面中,单击Program按钮,将硬件比特流文件下载到ZYNQ-7000中。当编程完成后,Zedboard板上的DONE LED灯(蓝色)变亮。

(6) 在SDK主界面左侧的Project Explorer窗口下,选择TestApp。单击右键,出现浮动菜单。在浮动菜单下,选择Run As→Launch on Hardware(GDB)选项。该选项将应用程序下载到ZYNQ-7000中,并执行ps7\_init和TestApp.elf。

(7) 如图4.55所示,看到在Terminal1标签窗口下,显示下面的结果。

(8) 选择Console标签,单击 按钮,停止运行程序。

(9) 关闭SDK和Vivado集成开发环境。

```
Push Buttons Status 10
DIP Switch Status 28
```

图4.55 lab2.c运行结果

#### 4.3 创建和添加定制IP

本节将通过使用Vivado IP封装器,创建并添加一个定制的外设到一个已经存在的处理器系统中。本节内容包括:使用外设模板创建定制IP、使用IP封装器封装外设、修改工程设置、添加IP到设计、添加约束XDC和添加BRAM。

##### 4.3.1 使用外设模板创建定制IP

本节将使用Vivado提供的axi\_lite从外设模板和定制IP源代码,创建一个满足设计要求的定制IP。使用外设模板创建定制IP的步骤主要包括:

(1) 在Windows 7操作系统主界面左下角选择开始→所有程序→Xilinx Design Tools→Vivado 2013.3→Vivado 2013.3。

(2) 在Vivado开始界面中,单击Manage IP,选择New IP Location...。

(3) 出现New IP Location>Create a New Customized IP Location对话框界面。

(4) 单击Next按钮。

(5) 如图 4.56 所示, 出现 New IP Location-Manage IP Settings 对话框界面。按下面设置参数:

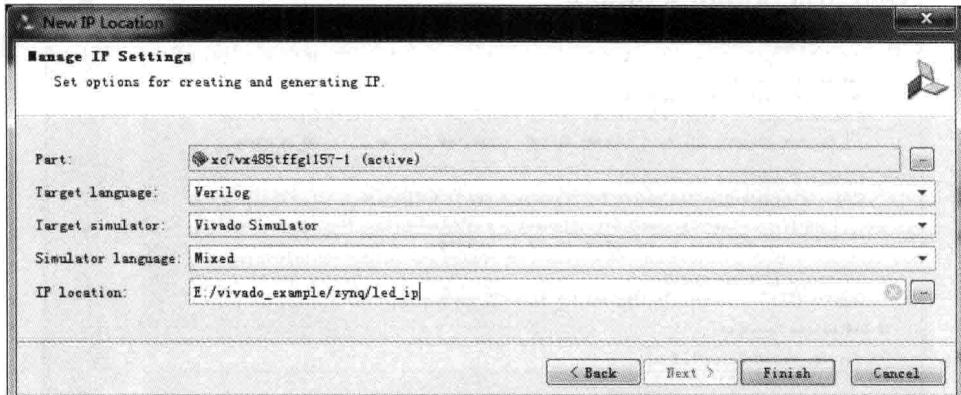


图 4.56 新 IP 的设置界面

- ① Target language: Verilog;
- ② Simulator language: Mixed;
- ③ IP location: E:/vivado\_example/zynq/led\_ip

**注:** 虽然该工程中需要选择一个 Virtex-7 器件。但是,之后可以添加条件用于满足对其他器件的兼容性。

- (6) 单击 Finish 按钮。
- (7) 出现 Create Directory 对话框界面。在该界面中提示创建一个目录。
- (8) 单击 OK 按钮。
- (9) 在 Vivado 主界面主菜单下, 选择 Tools→Create and Package IP。
- (10) 出现 Create And Package New IP 对话框界面。
- (11) 单击 Next 按钮。
- (12) 如图 4.57 所示, 出现 Create And Package New IP-Choose Create Peripheral or Package IP 对话框界面。在该界面内, 按下面设置参数:

- ① 选中 Create new AXI4 peripheral 前面的复选框;
- ② IP Definition Location: E:/vivado\_example/zynq/led\_ip。

- (13) 单击 Next 按钮。
- (14) 如图 4.58 所示, 出现 Create And Package New IP-Peripheral Details 对话框界面, 在该对话框界面中, 按下面设置参数:

- ① Vendor: BUCT;
- ② Name: led\_ip;
- ③ Display Name: led\_ip\_v1\_0;
- ④ Vendor Display Name: He Bin。

- (15) 单击 Next 按钮。
- (16) 如图 4.59 所示, 出现 Create And Package New IP-Add Interface 对话框界面。

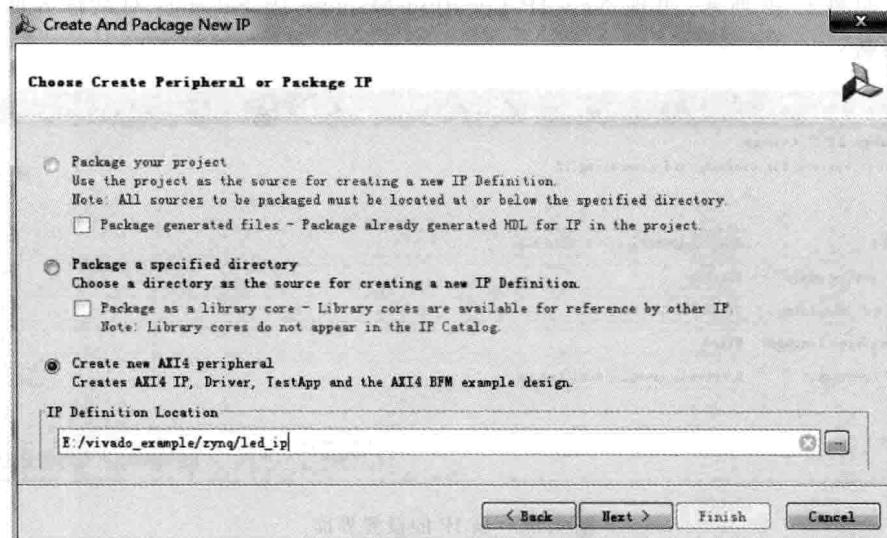


图 4.57 选择创建或者封装 IP 界面

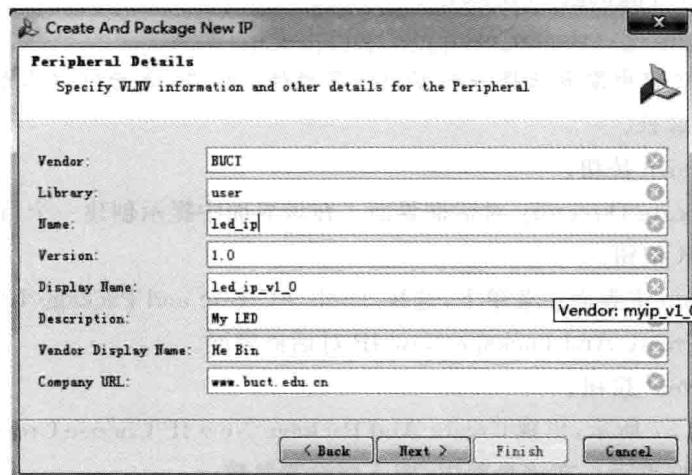


图 4.58 填充 IP 信息界面

在该对话框界面内,按下面设置参数:

- ① Name: S\_AXI;
- ② Interface Type: Lite;
- ③ Interface Mode: Slave;
- ④ Data Width: 32;
- ⑤ Number of Register: 4。

(17) 单击 Next 按钮。

(18) 如图 4.60 所示,出现 Create And Package New IP-Create Peripheral: Generation Options 对话框界面。在该界面中,选中 Create Drivers(Only for Full/Lite Slave interfaces)前面的复选框。

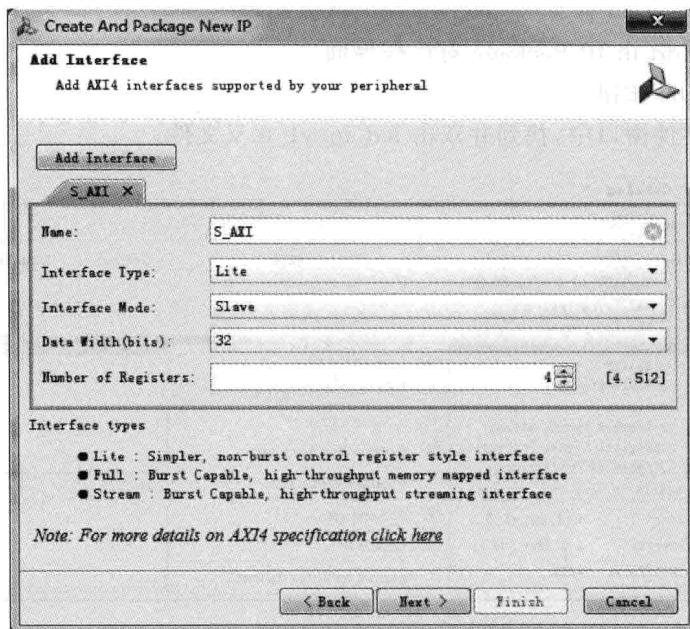


图 4.59 选择接口界面

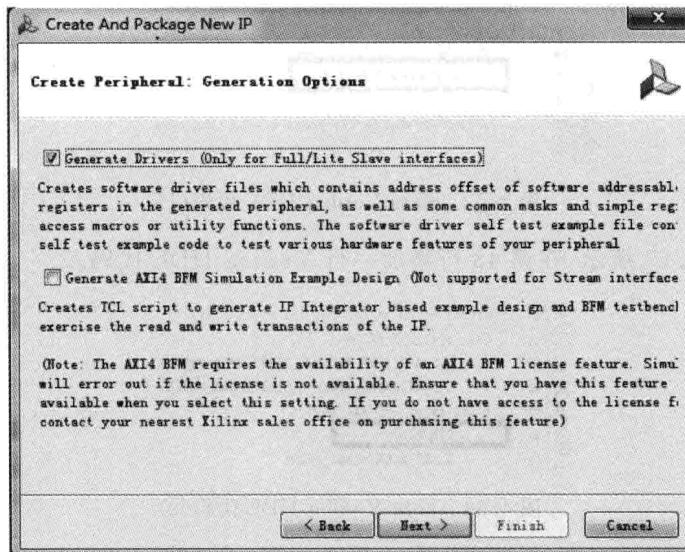


图 4.60 外设生成选项

- (19) 单击 Next 按钮。
- (20) 出现 Create And Package New IP-Begin Peripheral Creation 对话框界面。在该界面中,选中 Add IP to Catalog and open IP in editing session 前面的复选框。
- (21) 单击 Finish 按钮。
- (22) 如图 4.61 所示,在 IP 列表中,找到并展开 AXI Peripheral。在展开项中,找到并选择 led\_ip\_v1\_0。单击右键,出现浮动菜单。在浮动菜单内,选择 Edit in IP Packager

选项。

- (23) 出现 Edit in IP Package 对话框界面。
- (24) 单击 OK 按钮。
- (25) 在源文件窗口中,找到并双击 led\_ip\_v1\_0.v 文件。

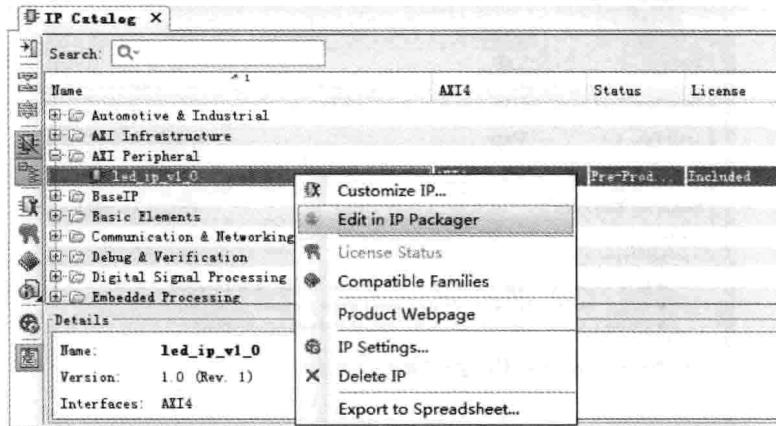


图 4.61 打开 IP 封装器入口

- (26) 如图 4.62 所示,在第 15 行添加一行 Verilog HDL 代码。

```

13 // 
14 // Users to add ports here
15 output wire[7:0] LED;
16 // User ports ends
17 // Do not modify the ports beyond this line
18

```

图 4.62 添加 Verilog HDL 代码(1)

- (27) 如图 4.63 所示,在第 48 行添加一行 Verilog HDL 代码。

```

43 // Instantiation of Axi Bus Interface S_AXI
44   led_ip_v1_0_S_AXI #(
45     .C_S_AXI_DATA_WIDTH(C_S_AXI_DATA_WIDTH),
46     .C_S_AXI_ADDR_WIDTH(C_S_AXI_ADDR_WIDTH)
47   ) led_ip_v1_0_S_AXI_inst (
48     .LED(LED),
49     .S_AXI_ACLK(s_axi_aclk),

```

图 4.63 添加 Verilog HDL 代码(2)

- (28) 按下 Ctrl+S 按键,保存修改后的文件。

- (29) 在源文件窗口下,找到并双击 led\_ip\_v1\_0\_S\_AXI.v 文件。打开该文件。

- (30) 如图 4.64 所示,在第 15 行添加 Verilog HDL 代码。

- (31) 如图 4.65 所示,在第 397 行添加下面的代码,用于例化逻辑,该逻辑用于 LED IP。

**注:** 也可以在/zynq/source 目录下,找到并打开 user\_logic\_instantiation.txt 文件。将该文件的代码复制到第 397 行开始的位置。

- (32) 在 Vivado 主界面左侧的 Flow Navigator 窗口下,找到并展开 Project Manager。在展开项中,选择 Add Sources。

```

13      (
14          // Users to add ports here
15          output wire [7:0] LED;
16          // User ports ends
17          // Do not modify the ports beyond this line

```

图 4.64 添加 Verilog HDL 代码(3)

```

// 396 // Add user logic here
397 lab3_user_logic U1(
398     .S_AXI_ACLK(S_AXI_ACLK),
399     .slv_reg_wren(slv_reg_wren),
400     .axi_awaddr(axi_awaddr[C_S_AXI_ADDR_WIDTH-1:ADDR_LSB]),
401     .S_AXI_WDATA(S_AXI_WDATA),
402     .S_AXI_ARESETN(S_AXI_ARESETN),
403     .LED(LED)
404 );

```

图 4.65 添加 Verilog HDL 代码(4)

(33) 出现 Add Sources 对话框界面。在该界面下,选中前面的 Add or Create Design Sources 复选框。

(34) 单击 Next 按钮。

(35) 出现 Add Sources-Add or Create Design Sources 对话框界面。在该界面下,单击 **Add Files...** 按钮,定位到路径/vivado\_example/zynq/source/lab3\_user\_logic.v。

(36) 单击 OK 按钮。

(37) 单击 Finish 按钮。

(38) 按 Ctrl+S 保存设计。

(39) 如图 4.66 所示,可以看到定制 IP 核完整的文件层次结构。下面给出了 lab3\_user\_logic 的代码。请读者仔细分析该段代码实现的功能。

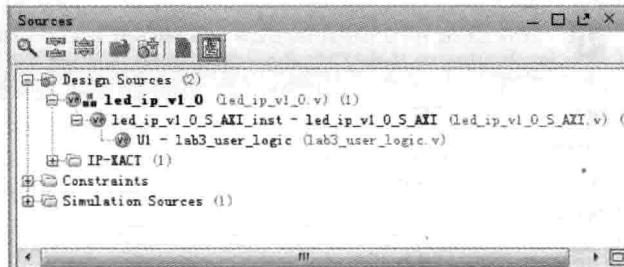


图 4.66 定制 IP 的文件层次

#### 代码清单 4-1 lab3\_user\_logic.v

```

'timescale 1ns / 1ps
///////////////////////////////
// Module Name: lab3_user_logic
/////////////////////////////

```

```

module lab3_user_logic(
    input S_AXI_ACLK,
    input slv_reg_wren,
    input [2:0] axi_awaddr,
    input [31:0] S_AXI_WDATA,
    input S_AXI_ARESETN,
    output reg [7:0] LED
);

always @ (posedge S_AXI_ACLK)
begin
    if (S_AXI_ARESETN == 1'b0)
        LED <= 8'b0;
    else
        if (slv_reg_wren && (axi_awaddr == 3'h0))
            LED <= S_AXI_WDATA[7:0];
end
endmodule

```

### 4.3.2 使用 IP 封装器封装外设

本节将使用 IP 封装器对外设进行进一步的封装处理。使用 IP 封装器封装外设的步骤主要包括：

- (1) 在 Vivado 主界面左侧的 Flow Navigator 窗口下，找到并展开 Project Manager，在展开项中，找到并单击 Package IP。
- (2) 如图 4.67 所示，出现 Package IP-led\_ip 对话框界面。

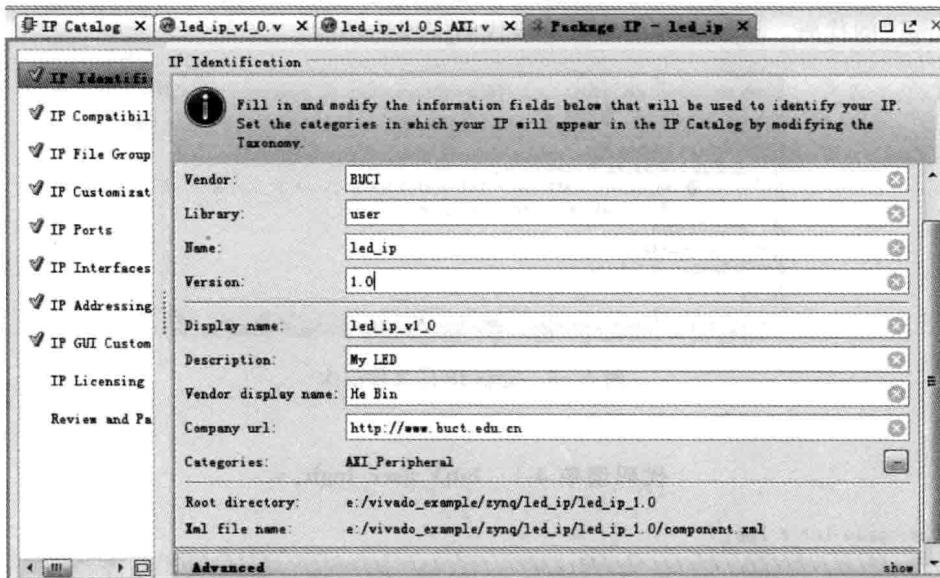


图 4.67 定制 IP 管理窗口

(3) 如图 4.67 所示,单击该界面左侧的 IP Compatibility。如图 4.68 所示,出现 Family Support 界面。该界面中,只有 Virtex7。可以增加其他可以支持的器件。单击右键,出现浮动菜单。在该浮动菜单内,选择 Add Family... 选项。

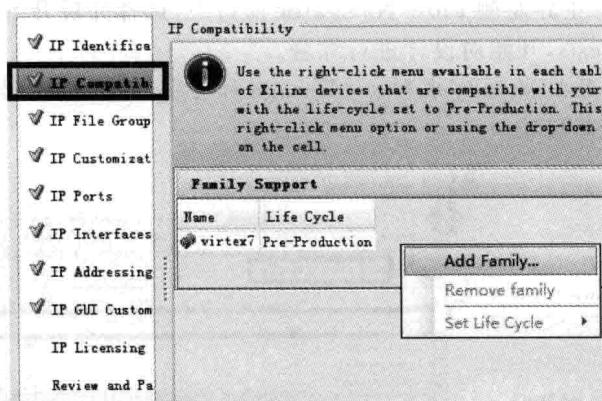


图 4.68 修改 IP 兼容性窗口

(4) 如图 4.69 所示,出现 Choose Family Support 对话框界面。在该界面选中 zynq 前面的复选框,该选项表示比 IP 支持 zynq 器件。

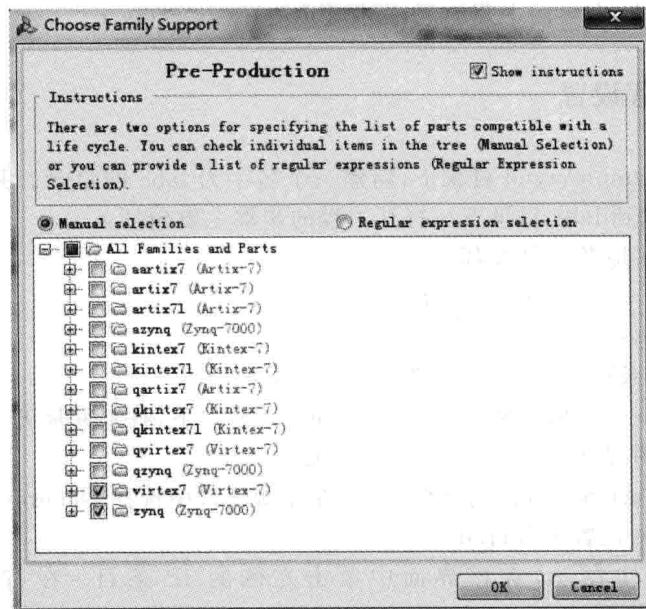


图 4.69 选择器件对话框界面

(5) 单击 OK 按钮。

(6) 如图 4.70 所示,在该界面左侧窗口内选择 IP Ports。然后,在右侧窗口内单击 Port Import Dialog。

(7) 等待完成端口导入后,可以看到在图 4.70 的界面中出现了名字为 LED 的新端口。

- (8) 选中图中左侧的 IP Customization Parameters 选项,进行导入参数的操作。
- (9) 选中图中左侧的 IP File Groups 选项,并且单击 Merge changes from IP File Groups Wizard。

(10) 在 Vivado 主界左侧 Flow Navigator 窗口下,选择并展开 Synthesis。在展开项中,双击 Run Synthesis,开始对设计进行综合。

**注:** 如果出错,则在 Vivado 底部的 Message 标签窗口下,查看错误消息。返回到前面的步骤,修改相应的错误。

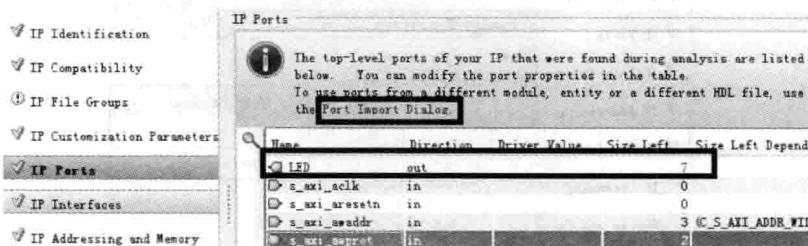


图 4.70 IP 端口修改界面

(11) 在 Package IP 左侧窗口内,选择 Review and Package 选项。在右侧窗口内,单击 Re-Package IP 按钮。

(12) 在 Vivado 主界面主菜单下,选择 File→Close Project。

### 4.3.3 修改工程设置

在\ vivado\_example\zynq 目录下,新建一个名字为 lab3 的目录。并且,将 lab2 目录下的所有文件复制到 lab3 目录下。打开工程的步骤主要包括:

- (1) 启动 Vivado 集成开发环境。
- (2) 在 Get Started 页面下,选择 Open Project。
- (3) 选择 Browse Project。
- (4) 定位到目录\zynq\lab3 下,选择并单击 lab1. prj。
- (5) 在 Vivado 主界面左侧的 Flow Navigator 窗口下,找到并展开 Project Manager。在展开项中,选择并单击 Project Settings 选项。
- (6) 出现 Project Settings 对话框界面,在左侧窗口中选择 General。在右侧窗口中,将 Target language 设置为 VHDL。
- (7) 如图 4.71 所示,在该界面中单击左侧的 IP 条目。在右侧窗口内,单击 Add Repository... 按钮。

定位到定制 IP 所在的路径:

e:/vivado\_example/zynq/led\_ip.

可以看到该 IP 路径出现在 IP Repositories 窗口下。同时,在 IP in Selected Repository 窗口下也显示出 IP 核的名字 led\_ip\_v1\_0。

(8) 单击 OK 按钮。

(9) 出现 Create New Run 对话框界面,选择 No(表示不创建新的运行)。

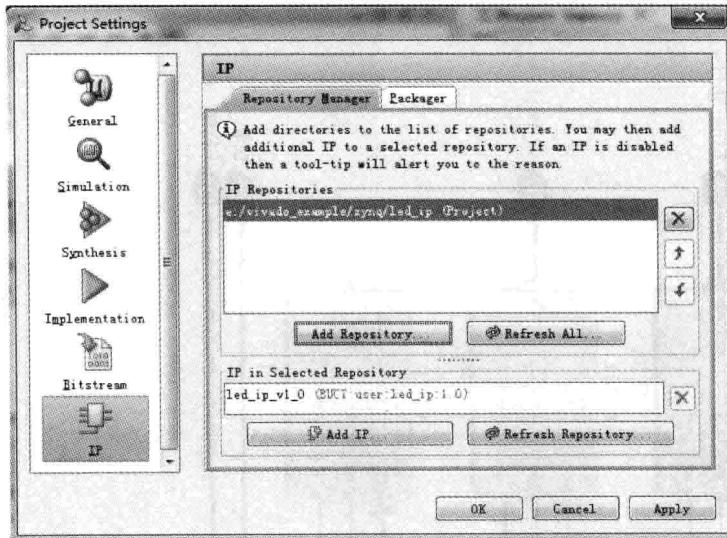


图 4.71 添加 IP 路径

#### 4.3.4 添加定制 IP 到设计

本节将添加 led\_ip，并且连接到处理器系统的 AXI4 Lite 接口。同时，将进行内部和外部端口的连接。此外，建立 LED 端口将其作为外部 FPGA 的引脚。添加定制 IP 的步骤主要包括：

- (1) 在 Vivado 主界面左侧的 Flow Navigator 窗口下，找到并展开 IP Integrator。在展开项中，单击 Open Block Design，或者在源文件窗口中选择 system.bd 文件，打开 IP 集成器。
- (2) 在 Diagram 窗口左侧的一列工具栏中，单击 按钮。
- (3) 如图 4.72 所示，出现 IP Catalog 对话框界面。在 Search 中输入 led，可以看到下面窗口显示 led\_ip\_v1\_0。
- (4) 双击 led\_ip\_v1\_0，将其添加到设计中。
- (5) 在 Diagram 窗口下，选中 led\_ip\_0 块符号。然后，在 Block Properties 窗口内将其名字改为 led\_ip。
- (6) 单击 Diagram 窗口内上方的 Run Connection Automation。然后，选择 /led\_ip/S\_AXI。
- (7) 出现 Run Connection Automation 对话框界面。
- (8) 单击 OK 按钮。
- (9) 单击 Diagram 左侧一列的工具栏内的 按钮，重新绘制图系统结构。图 4.73 给出了重新绘制完成后的系统结构图。
- (10) 在图 4.73 所示的界面内，选择 led\_ip 模块的 LED[7:0] 端口，单击右键，出现浮动菜单。在浮动菜单内，选择 Make External 选项。

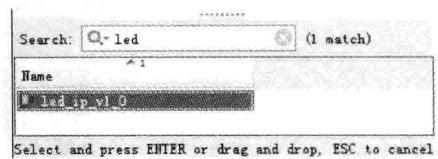


图 4.72 led\_ip 入口

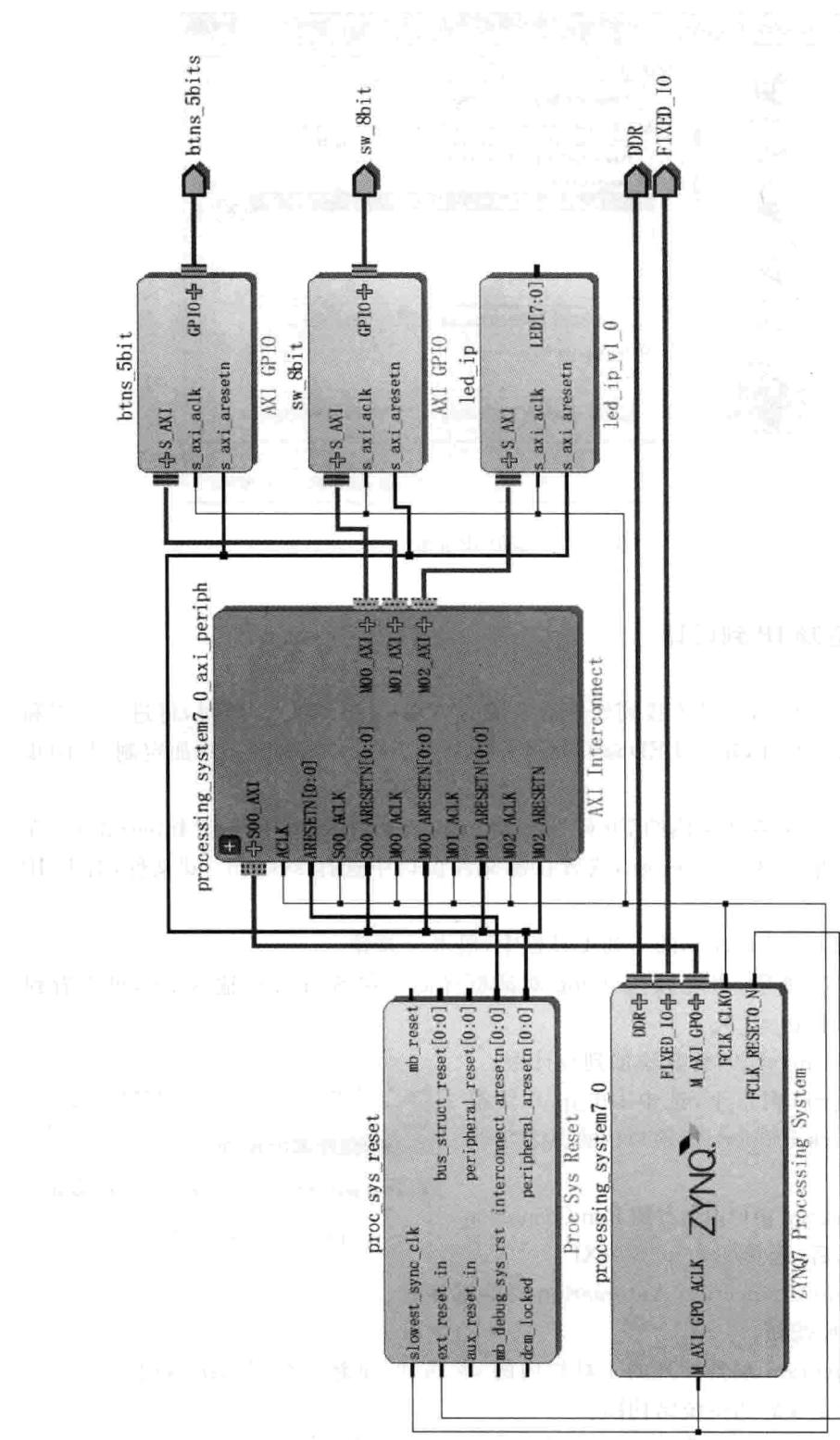


图 4.73 重新绘制后的系统结构图

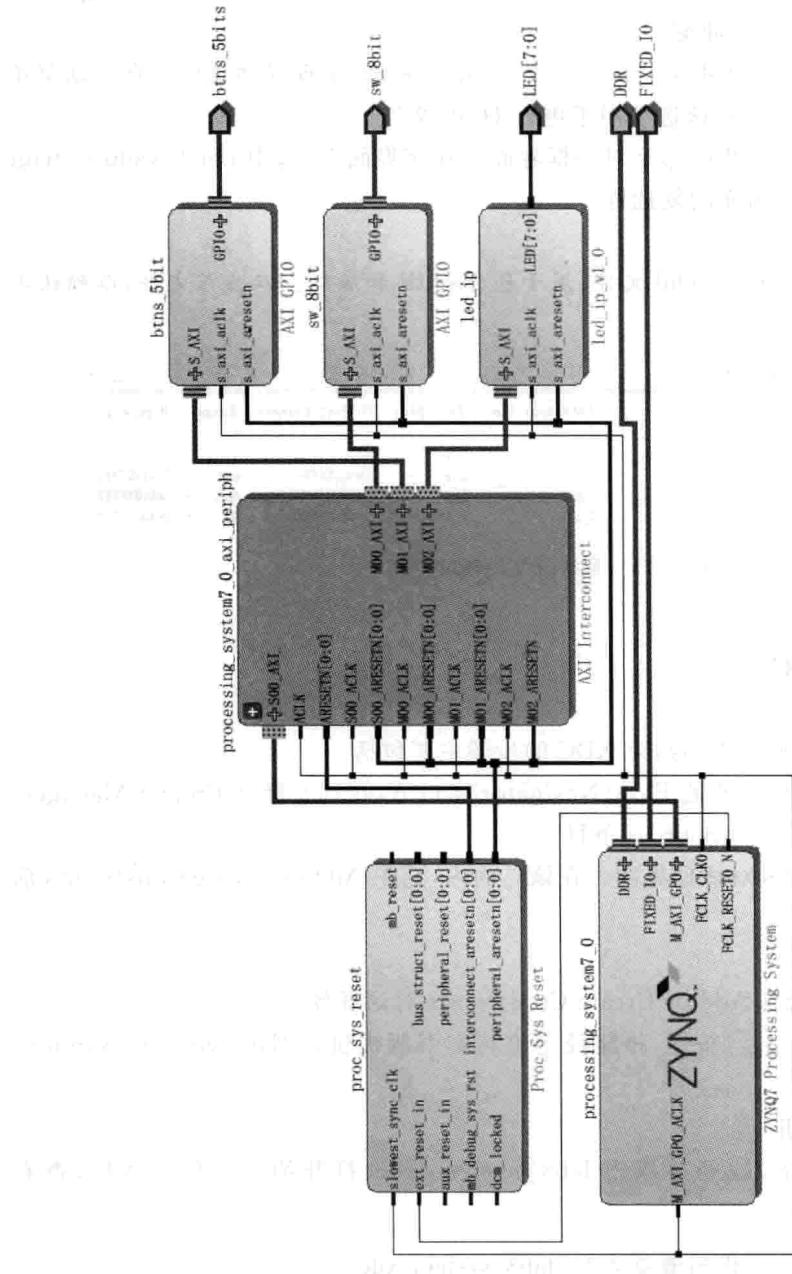


图 4.74 添加端口后的系统结构图

- (11) 如图 4.74 所示,给出了添加完 LED[7:0]端口后的系统结构图。
- (12) 如图 4.75 所示,选择 Address Editor 标签,可以看到系统已经为 LED\_IP 分配了地址空间。
- (13) 在 Vivado 主界面主菜单下,选择 Tools→Validate Design,运行设计规则检查。在系统设计中,不应该有任何冲突。
- (14) 在源文件窗口中,选中 system.bd 文件,单击右键,出现浮动菜单。在浮动菜单内选择 Create HDL Wrapper,该选项用于更新 HDL 文件。
- (15) 出现 Create HDL Wrapper 对话框界面。在该界面下,选中 Let Vivado manage wrapper and auto-update 前面的复选框。
- (16) 单击 OK 按钮。

**注:** 更新 system\_wrapper.vhd 文件,用于包含新 IP 和端口。双击该文件,以确认添加了 LED 端口。

Cell	Interface Pin	Base Name	Offset Address	Range	High Address
processing_system7_0					
Data C2 address bits : 4G					
sw_8bit	S_AXI	Reg	0x41200000	64K	0x4120FFFF
btns_8bit	S_AXI	Reg	0x41210000	64K	0x4121FFFF
led_ip	S_AXI	S_AXI_Reg	0x43C00000	4K	0x43C00FFF

图 4.75 系统分配的地址空间

### 4.3.5 添加约束 XDC

本节将添加约束 XDC。添加约束 XDC 的步骤主要包括:

- (1) 在 Vivado 主界面左侧的 Flow Navigator 窗口下,找到并展开 Project Manager。在展开项中,找到并单击 Add Sources 条目。
- (2) 出现 Add Sources 对话框界面。在该界面内,选中 Add or Create Constraints 前面的复选框。
- (3) 单击 Next 按钮。
- (4) 出现 Add Sources-Add or Create Constraints 对话框界面。
- (5) 单击 **Add Files...** 按钮,将路径定位到本书所提供资料的\ vivado\_example\ zynq\source 下。
- (6) 单击 Finish 按钮。
- (7) 在源文件窗口下,找到并双击 lab3\_system.xdc,打开约束文件。然后,查看 XDC 文件内容。

#### 代码清单 4-2 lab3\_system.xdc

```
# ######
# On-board LED
#
######
set_property PACKAGE_PIN T22 [get_ports LED[0]]
set_property IOSTANDARD LVCMOS33 [get_ports LED[0]]
```

```

set_property PACKAGE_PIN T21 [get_ports LED[1]]
set_property IOSTANDARD LVC MOS33 [get_ports LED[1]]
set_property PACKAGE_PIN U22 [get_ports LED[2]]
set_property IOSTANDARD LVC MOS33 [get_ports LED[2]]
set_property PACKAGE_PIN U21 [get_ports LED[3]]
set_property IOSTANDARD LVC MOS33 [get_ports LED[3]]
set_property PACKAGE_PIN V22 [get_ports LED[4]]
set_property IOSTANDARD LVC MOS33 [get_ports LED[4]]
set_property PACKAGE_PIN W22 [get_ports LED[5]]
set_property IOSTANDARD LVC MOS33 [get_ports LED[5]]
set_property PACKAGE_PIN U19 [get_ports LED[6]]
set_property IOSTANDARD LVC MOS33 [get_ports LED[6]]
set_property PACKAGE_PIN U14 [get_ports LED[7]]
set_property IOSTANDARD LVC MOS33 [get_ports LED[7]]

```

#### 4.3.6 添加 BRAM

本节将添加 BRAM 到设计中。添加 BRAM 的步骤主要包括：

- (1) 单击 Diagram 窗口左侧工具栏内的 按钮。
- (2) 出现 IP Catalog 对话框界面，在 Search 右侧输入 BRAM。找到并双击 AXI BRAM Controller，将 BRAM 控制器 IP 核的一个例化添加到设计中。
- (3) 单击 Diagram 上面的 Run Connection Automation，并选择 axi\_bram\_ctrl\_0/S\_AXI。
- (4) 出现 Run Connection Automation 对话框界面。
- (5) 单击 OK 按钮。
- (6) 选择并双击图中的 axi\_bram\_ctrl\_0 块符号。
- (7) 如图 4.76 所示，出现 Re-customize IP 对话框界面。将 Number of BRAM interface 的值改为 1。

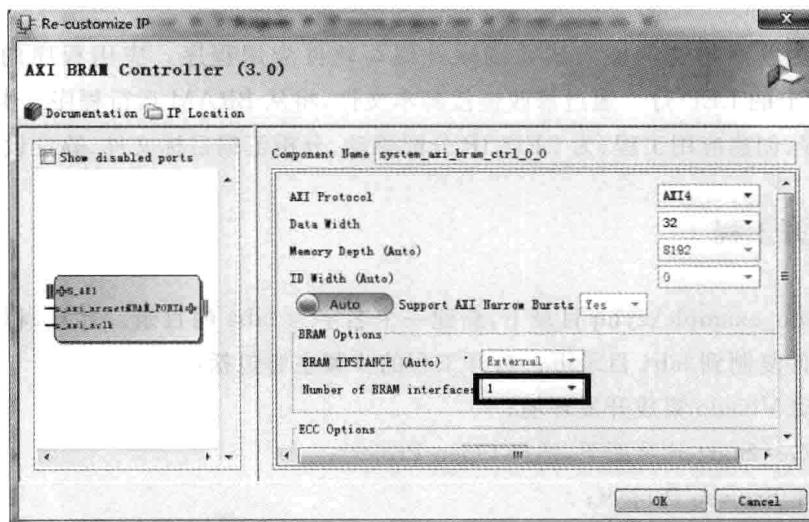


图 4.76 AXI BARM Controller 参数设置界面

(8) 单击 OK 按钮。

**注：**使用 AXI4 协议而不使用 AXI4Lite 协议，这样就能为 BRAM 控制器提供更高的访问带宽。同时，控制器能支持猝发交易。

(9) 单击 Diagram 顶层的 Run Connection Automation，选择 /axi\_bram\_ctrl\_0/BARM\_PORTA。

(10) 出现 Run Connection Automation 对话框界面。

(11) 单击 OK 按钮。

(12) 单击 Diagram 左侧工具栏内的 按钮，重新绘制结构图。

图 4.77 给出了添加完 BARM 后的系统结构图。

(13) 如图 4.78 所示，单击 Address Editor 标签。在该标签窗口内，将 axi\_bram\_ctrl\_0 所对应的 Range(范围)改成 8K。

(14) 在源文件窗口，右键单击 system.bd 文件，出现浮动菜单。在浮动菜单内，选择 Create HDL wrapper。

(15) 出现 Create HDL Wrapper 对话框界面。在该界面内选中，Let Vivado manage wrapper and auto-update 前面的复选框。

(16) 单击 OK 按钮。

(17) 按 F6 按键，最后一次验证设计的有效性。

(18) 在 Vivado 主界面左侧 Flow Navigator 窗口下，找到并展开 Program and Debug。在展开项中，单击 Generate Bitstream 选项。开始对设计进行综合、实现和生成比特流的过程。

**注：**在过程中，出现一些提示信息，按照前面所介绍的方法来处理这些提示信息。

## 4.4 编写软件程序

本节将为前面构建的嵌入式系统硬件编写软件应用程序。应用程序的功能是写 ZedBoard 板上的 LED 灯。通过修改链接脚本文件，将从 BRAM 运行程序。本节内容包括：打开工程、创建应用工程、为 LED\_IP 分配驱动、分析汇编目标文件、验证设计。

### 4.4.1 打开工程

在 \vivado\_example\zynq 目录下，新建一个名字为 lab4 的目录。并且，将 lab3 目录下的所有文件复制到 lab4 目录下。打开工程的步骤主要包括：

(1) 启动 Vivado 集成开发环境；

(2) 在 Get Started 页面下，选择 Open Project；

(3) 选择 Browse Project；

(4) 定位到目录 \zynq\lab4。在该目录下，选择并单击 lab1.prj。

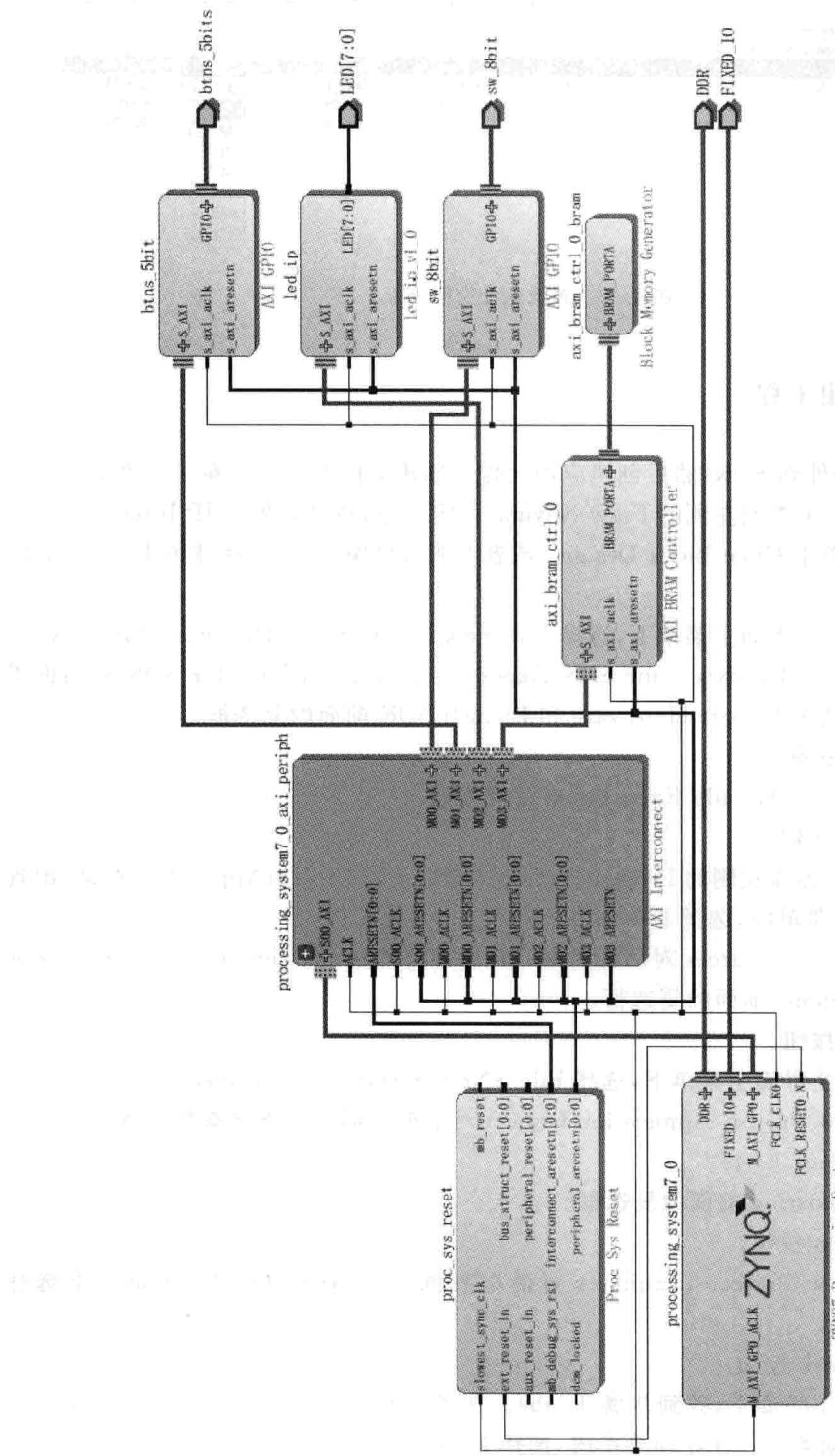


图 4.77 添加加完 BARM 后的系统结构图

Cell	Interface Pin	Base Name	Offset Address	Range	High Address
processing_system7_0					
Data (32 address bits : 4G)					
axi_bram_ctrl_0	S_AXI	Mem0	0x40000000	8K	0x40001FFF
btms_Sbit	S_AXI	Reg	0x41210000	4K	0x4121FFFF
led_ip	S_AXI	S_AXI_Reg	0x43C00000	8K	0x43C00FFF
sw_Sbit	S_AXI	Reg	0x41200000	16K	0x4120FFFF
				32K	
				64K	
				128K	
				256K	
				512K	

图 4.78 系统地址空间分配

#### 4.4.2 创建应用工程

本节将导出硬件到 SDK，然后创建应用工程。创建应用工程的步骤主要包括：

- (1) 在 Vivado 主界面左侧的 Flow Navigator 窗口下，找到并展开 IP Integrator。在展开项中，找到并单击 Open Block Design；或者在源文件窗口下，选择并单击 system.bd 文件。
- (2) 在 Vivado 主界面主菜单下，选择 File→Export→Export Hardware for SDK…。
- (3) 出现 Export Hardware for SDK-Export hardware platform for SDK 对话框界面。在该界面中，选中 Export Hardware 和 Launch SDK 前面的复选框。
- (4) 单击 OK 按钮。
- (5) 出现 Module Already Exported 对话框界面。
- (6) 单击 Yes 按钮。
- (7) 在 SDK 主界面左侧的 Project Explorer 窗口下，选中 TetApp。单击右键，出现浮动菜单。在浮动菜单内，选择 Delete。
- (8) 出现 Delete Resources 对话框界面。在该界面内选中 Delete project contents on disk(cannot be undone)前面的复选框。
- (9) 单击 OK 按钮。
- (10) 在 SDK 主界面主菜单下，选择 File→New→Application Project。
- (11) 出现 New Project-Application Project 对话框界面。按下面参数设置：
  - ① Project name: lab4；
  - ② 选中 Use existing 前面的复选框。
- (12) 单击 Next 按钮。
- (13) 出现 New Project-Templates 对话框界面。在 Available Templates 下选择 Empty Application。
- (14) 单击 Finish 按钮。
- (15) 在 SDK 主界面下，找到并展开 lab4。在展开项中，找到 src 子目录。选中 src，单击右键，出现浮动菜单。在浮动菜单内，选择 Import。
- (16) 出现 Import 对话框界面。在该界面中，展开 General。在展开项中，找到并选中 File System。

- (17) 单击 Next 按钮。
- (18) 出现 Import 对话框界面。在该界面内,单击 **Browse...** 按钮。定位到 E:\vivado\_example\zynq\source
- (19) 如图 4.79 所示,从右侧窗口中选择 lab4.c 文件。

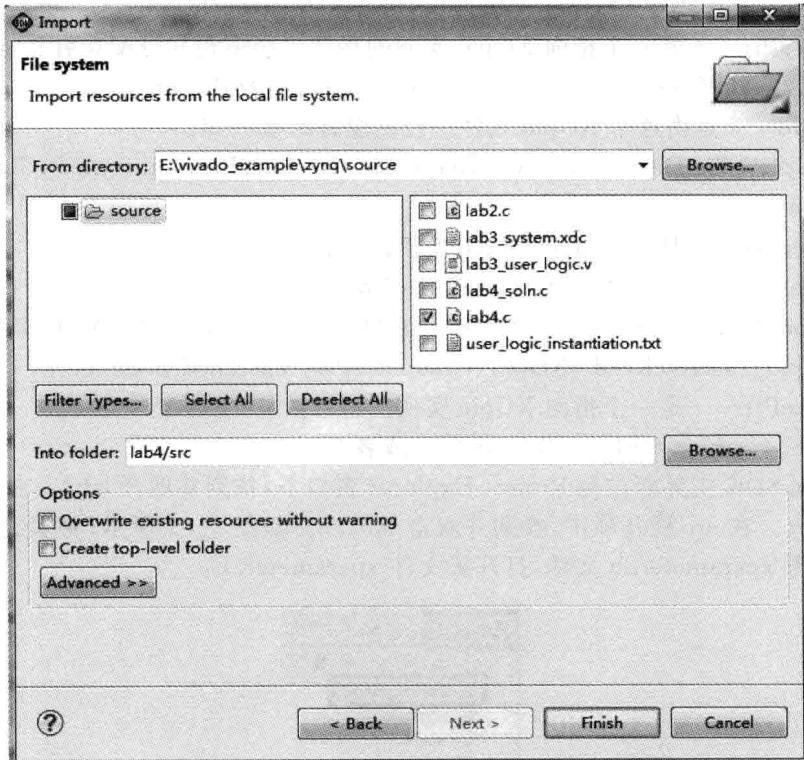


图 4.79 导入 lab4.c 文件

- (20) 单击 Finish 按钮(忽略编译过程中的任何错误)。
- (21) 在 SDK 主界面左侧 Project Explorer 窗口下,找到并展开 standalone\_bsp。在展开项中,选择 system.mss 文件。
- (22) 如图 4.80 所示,在 Peripheral Drivers 窗口下,单击 btns\_5bit 右侧的 Documentation。在一个默认的浏览器窗口中,打开文档。

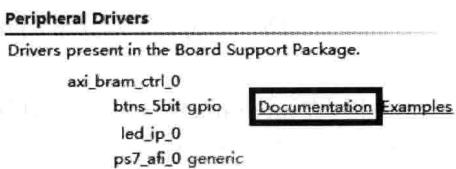


图 4.80 打开文档入口

- (23) 在浏览器窗口上面,单击 Files,查看和 GPIO IP 核相关的不同的 C 和头文件。
- (24) 单击头文件 xgpio.h,查看用于 GPIO 的可用的函数调用。

在软件应用程序中,通过下面的步骤,执行从 GPIO 的读:

- ① 初始化 GPIO;
- ② 设置数据方向;
- ③ 读数据。

单击下面的函数,查看函数的描述:

- ① XGpio\_Initialize (XGpio \* InstancePtr, u16 DeviceId);

InstancePtr——是一个指向 XGpio 实例的指针。必须预先分配指针指向的存储器。

使用这个指针,就可以通过 XGpio API 实现更多对元件操作的调用。

DeviceId——是由这个 XGpio 元件所控制设备的唯一 id。

- ② XGpio\_SetDataDirection (XGpio \* InstancePtr, unsigned Channel, u32 DirectionMask);

InstancePtr——是一个指向 XGpio 实例的指针。

Channel——包含 GPIO 工作的通道(1 或者 2)。

DirectionMask——方向标志,说明某一位是输入还是输出。0 表示输出,1 表示输入。

- ③ XGpio\_DiscreteRead (XGpio \* InstancePtr, unsigned Channel);

InstancePtr——是一个指向 XGpio 实例的指针。

Channel——包含 GPIO 工作的通道(1 或者 2)。

(25) 在 SDK 主界面左侧 Project Explorer 窗口下,找到并展开 lab4。在展开项中找到并展开 src。在 src 展开项中,找到并双击 lab4.c。如图 4.81 所示,将填充 Outline 标签窗口,双击 xparameter.h 文件,打开头文件 xparameter.h。

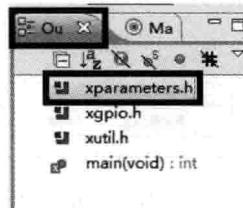


图 4.81 生成的文件

xparameter.h 文件包含了系统内外设的地址映射和设备号。这个文件从 Vivado 所生成的硬件平台描述中生成。找到下面的#define,用于标识开关外设。

```
#define XPAR_SW_8BIT_DEVICE_ID 1
```

**注:** 数字可能不同。

(26) 如图 4.82 所示,修改 lab4.c 的第 15 行,在 XGpio\_Initialize 函数中使用这个宏(#define)。

(27) 对于 BTNS\_5BIT 也一样,在 xparameters.h 中找到用于 BTNS\_5BIT 外设的宏(#define)。修改 lab4.c 的第 18 行,并保存文件。

(28) 重新建立文件。

**注:**

- ① 如果出现错误,则修改代码。
- ② 代码最终读开关的值,将其输出到 led\_ip。

```

#include "xparameters.h"
#include "xgpio.h"
#include "xutil.h"

// =====

int main (void)
{
    XGpio dip, push;
    int i, psb_check, dip_check;

    xil_printf("-- Start of the Program --\r\n");
    XGpio_Initialize(&dip, XPAR_DIP_DEVICE_ID); // Modify this
    XGpio_SetDataDirection(&dip, 1, 0xffffffff);

    XGpio_Initialize(&push, XPAR_PUSH_DEVICE_ID); // Modify this
    XGpio_SetDataDirection(&push, 1, 0xffffffff);

    while (1)
    {
        psb_check = XGpio_DiscreteRead(&push, 1);
        xil_printf("Push Buttons Status %x\r\n", psb_check);
        dip_check = XGpio_DiscreteRead(&dip, 1);
        xil_printf("DIP Switch Status %x\r\n", dip_check);

        // output dip switches value on LED_ip device

        for (i=0; i<9999999; i++);
    }
}

```

图 4.82 需要修改的文件

## 代码清单 4-3 修改后的 lab4.c

```

#include "xparameters.h"
#include "xgpio.h"
#include "xutil.h"

// =====

int main (void)
{
    XGpio dip, push;
    int i, psb_check, dip_check;

    xil_printf("-- Start of the Program --\r\n");

    XGpio_Initialize(&dip, XPAR_SW_8BIT_DEVICE_ID);
    XGpio_SetDataDirection(&dip, 1, 0xffffffff);

    XGpio_Initialize(&push, XPAR_BTNS_5BIT_DEVICE_ID);
    XGpio_SetDataDirection(&push, 1, 0xffffffff);

    while (1)
    {
        psb_check = XGpio_DiscreteRead(&push, 1);

```

```

xil_printf("Push Buttons Status %x\r\n", psb_check);
dip_check = XGpio_DiscreteRead(&dip, 1);
xil_printf("DIP Switch Status %x\r\n", dip_check);

// output dip switches value on LED_ip device

for (i = 0; i < 9999999; i++);
}
}

```

#### 4.4.3 为 LED\_IP 分配驱动

从驱动目录中,找到并分配 led\_ip 驱动到 led\_ip 实例。为 led\_ip 分配驱动的步骤主要包括:

- (1) 在 SDK 主界面主菜单下,选择 Xilinx Tools→Reposiories。
- (2) 出现 Preferences 对话框界面,单击 **New...** 按钮。将路径定位到:

E:\vivado\_example\zynq\led\_ip\led\_ip\_1.0

- (3) 单击 OK 按钮。
- (4) 在 SDK 主界面主菜单下,选择 Xilinx→Board Support Package Settings。
- (5) 出现 Select a board support package 对话框界面。在该界面内,选择 standalone\_bsp。
- (6) 单击 OK 按钮。
- (7) 出现 Board Support Package Settings 对话框界面。在该界面左侧选择 drivers。如图 4.83 所示,在 axi\_bram\_ctrl\_0 所对应的 Driver 列下面,通过下拉框选择 bram。在 led\_ip\_0 所对应的 Driver 列下面,通过下拉框选择 led\_ip。

Component	Component Type	Driver	Dri...
ps7_cortexa9_0	ps7_cortexa9	cpu_cortexa9	1.0...
axi_bram_ctrl_0	axi_bram_ctrl	bram	3.0...
btms_5bit	axi_gpio	gpio	3.0...
led_ip_0	led_ip	led_ip	1.0...

图 4.83 为外设分配驱动文件

- (8) 单击 OK 按钮。

当创建 IP 模板时,自动创建驱动程序代码。驱动程序包含更高级的函数,这些函数可以通过用户程序进行调用。驱动程序用于实现低层的控制功能,用于对外设进行控制。

- (9) 在 windows 浏览器中,定位到:

e:\vivado\_example\zynq\led\_ip\led\_ip\_1.0\driver\led\_ip\_v1\_00\_a\src

在这个目录下,找到并打开 led\_ip.c。这个文件只包含用于 IP 的头文件。

- (10) 关闭 led\_ip.c 文件,打开头文件 led\_ip.h,查看下面的宏:

LED\_IP\_mWriteReg( ... )

```
LED_IP_mReadReg( ... )
```

(11) 修改 lab4.c 文件。

(12) 包含头文件：

```
# include "led_ip.h"
```

(13) 包含写 IP 的函数：

```
LED_IP_mWriteReg(XPAR_LED_IP_0_S_AXI_BASEADDR, 0, dip_check);
```

(14) 保存文件。

#### 代码清单 4-4 最终的 lab4.c

```
# include "xparameters.h"
# include "xgpio.h"
# include "xutil.h"
# include "led_ip.h"
// =====

int main (void)
{
    XGpio dip, push;
    int i, psb_check, dip_check;

    xil_printf(" -- Start of the Program -- \r\n");

    XGpio_Initialize(&dip, XPAR_SW_8BIT_DEVICE_ID);
    XGpio_SetDataDirection(&dip, 1, 0xffffffff);

    XGpio_Initialize(&push, XPAR_BTNS_5BIT_DEVICE_ID);
    XGpio_SetDataDirection(&push, 1, 0xffffffff);

    while (1)
    {
        psb_check = XGpio_DiscreteRead(&push, 1);
        xil_printf("Push Buttons Status %x\r\n", psb_check);
        dip_check = XGpio_DiscreteRead(&dip, 1);
        xil_printf("DIP Switch Status %x\r\n", dip_check);

        // output dip switches value on LED_ip device
        LED_IP_mWriteReg(XPAR_LED_IP_0_S_AXI_BASEADDR, 0, dip_check);

        for (i = 0; i < 9999999; i++);
    }
}
```

#### 4.4.4 分析汇编目标文件

本节启动 shell 和 objdump lab4.elf 文件,用于查看文件中不同段的位置。分析汇编目标文件的步骤包括:

(1) 在 SDK 主界面主菜单下,选择 Xilinx Tools→Launch Shell。

(2) 如图 4.84 所示,先后输入命令:

```
cd lab4
cd debug
```

进入 debug 目录。

```
E:\vivado_example\zynq\lab4\lab1.sdk\SDK>cd sdk_export
E:\vivado_example\zynq\lab4\lab1.sdk\SDK\SDK_Export>cd lab4
E:\vivado_example\zynq\lab4\lab1.sdk\SDK\SDK_Export\lab4>cd debug
E:\vivado_example\zynq\lab4\lab1.sdk\SDK\SDK_Export\lab4\Debug>arm-xilinx-eabi-objdump -h lab4.elf
'arm-xilinx-eabi-objdump' 不是内部或外部命令, 也不是可运行的程序
或批处理文件。

E:\vivado_example\zynq\lab4\lab1.sdk\SDK\SDK_Export\lab4\Debug>arm-xilinx-eabi-objdump -h lab4.elf
```

图 4.84 命令界面

(3) 输入下面的命令

```
Arm - xilinx - eabi - objdump - h lab4.elf
```

如图 4.85 所示,可以看到不同的段分配。

#### 4.4.5 验证设计

本节将对设计进行验证。验证设计的步骤包括:

(1) 按照前面方法,连接 PC 和 Zedboard 之间的 JTAG 和 UART 电缆,并且给 Zedboard 上电。

(2) 选择 **Terminal 1** 标签。

**注:**如果没有出现该标签,则在 SDK 主界面主菜单下,选择 Window→Show view→Terminal。

(3) 单击 **N** 按钮。按 4.1.5 节介绍的参数配置 UART。

(4) 在 SDK 主界面主菜单下,选择 Xilinx Tools→Program FPGA。

(5) 出现 Program FPGA 对话框界面。在该界面中,单击 Program 按钮,将比特流文件下载到 ZYNQ-7000 中。当完成编程后,Zedboard 板上的 DONE LED 灯(蓝色)变亮。

```
管理员: C:\Windows\System32\cmd.exe  
或批处理文件。  
E:\vivado_example\zyng\lab4\lab1.sdk\SDK\SDK_Export\lab4\Debug>arm-xilinx-eabi-objdump -h lab4.elf  
  
lab4.elf:      file format elf32-littlearm  
  
Sections:  
Idx Name      Size    VMA     LMA     File off   Align  
0 .text       00001b60  00100000  00100000  00008000  2**2  
              CONTENTS, ALLOC, LOAD, READONLY, CODE  
1 .init       00000018  00101b60  00101b60  00009b60  2**2  
              CONTENTS, ALLOC, LOAD, READONLY, CODE  
2 .fini       00000018  00101b78  00101b78  00009b78  2**2  
              CONTENTS, ALLOC, LOAD, READONLY, CODE  
3 .rodata      0000018c  00101b90  00101b90  00009b90  2**2  
              CONTENTS, ALLOC, LOAD, READONLY, DATA  
4 .data        00000464  00101d20  00101d20  00009d20  2**3  
              CONTENTS, ALLOC, LOAD, DATA  
5 .eh_frame    00000004  00102184  00102184  0000a184  2**2  
              CONTENTS, ALLOC, LOAD, READONLY, DATA  
6 .mmu_tbl     00004000  00104000  00104000  0000c000  2**0  
              CONTENTS, ALLOC, LOAD, READONLY, DATA  
7 .init_array  00000008  00108000  00108000  00010000  2**2  
              CONTENTS, ALLOC, LOAD, DATA  
8 .fini_array  00000004  00108008  00108008  00010008  2**2  
              CONTENTS, ALLOC, LOAD, DATA  
9 .ARM.attributes 00000033  0010800c  0010800c  0001000c  2**0  
              CONTENTS, READONLY  
10 .bss         0000005c  0010800c  0010800c  0001000c  2**2  
              ALLOC  
11 .heap        00002008  00108068  00108068  0001000c  2**0  
              ALLOC  
12 .stack       00005400  0010a070  0010a070  0001000c  2**0  
              ALLOC  
13 .debug_info  000020e7  00000000  00000000  0001003f  2**0  
              CONTENTS, READONLY, DEBUGGING  
14 .debug_abbrev 000002b  00000000  00000000  00012126  2**0  
              CONTENTS, READONLY, DEBUGGING  
15 .debug_loc   00001229  00000000  00000000  00012f51  2**0  
              CONTENTS, READONLY, DEBUGGING  
16 .debug_aranges 00000218  00000000  00000000  00014180  2**3  
              CONTENTS, READONLY, DEBUGGING
```

图 4.85 段的分配

(6) 在 SDK 主界左侧的 Project Explorer 窗口下,选择 TestApp。单击右键,出现浮动菜单。在浮动菜单下,选择 Run As→Launch on Hardware (GDB)。将应用程序下载到 ZYNQ-7000 中,并且执行 ps7\_init 和 lab4.elf。

(7) 如图 4.86 所示,看到在 Terminal 1 标签窗口下,显示下面的结果。

图 4.86 lab4.c 运行结果

## 4.5 软件控制定时器和调试

本节将写软件应用程序,该应用程序使用 CPU 的私有定时器。读者可以参考 SDK 内所提供的定时器 API 函数,并创建和调试软件应用程序。该应用程序还将监控 dip 开关值,然后在 LED 上显示递增的计数值。当按下按键时,将退出应用程序。

本节内容包括: 打开工程、创建 SDK 软件工程、在硬件上验证操作和启动调试器。

### 4.5.1 打开工程

在\ vivado\_example\zynq 目录下,新建一个名字为 lab5 的目录。并且,将 lab4 目录下的所有文件复制到 lab5 目录下。打开工程的步骤主要包括:

- (1) 启动 Vivado 集成开发环境;
- (2) 在 Get Started 页面下,选择 Open Project;
- (3) 选择 Browse Project;
- (4) 定位到目录\zynq\lab5 下,选择并单击 lab1.prj。

### 4.5.2 创建 SDK 软件工程

本节将导出硬件到 SDK,然后创建应用工程。创建应用工程的步骤主要包括:

- (1) 在 Vivado 主界面左侧的 Flow Navigator 窗口下,找到并展开 IP Integrator。在展开项中,找到并单击 Open Block Design; 或者在源文件窗口下,选择并单击 system.bd 文件。
- (2) 在 Vivado 主界面主菜单下,选择 File→Export→Export Hardware for SDK…。
- (3) 出现 Export Hardware for SDK-Export hardware platform for SDK 对话框界面。在该界面中,选中 Export Hardware 和 Launch SDK 前面的复选框。
- (4) 单击 OK 按钮。
- (5) 出现 Module Already Exported 对话框界面。
- (6) 单击 Yes 按钮。
- (7) 在 SDK 主界面左侧的 Project Explorer 窗口下,选中 lab4。单击右键,出现浮动菜单。在浮动菜单内,选择 Delete。
- (8) 出现 Delete Resources 对话框界面。在该界面选中 Delete project contents on disk(cannot be undone)前面的复选框。
- (9) 单击 OK 按钮。
- (10) 在 SDK 主界面主菜单下,选择 File→New→Application Project。
- (11) 出现 New Project-Application Project 对话框界面。在该对话框界面下,按下面设置参数:
  - ① Project name: lab5;
  - ② 选中 Use existing 前面的复选框。
- (12) 单击 Next 按钮。

(13) 出现 New Project-Templates 对话框界面。在 Available Templates 下选择 Empty Application。

(14) 单击 Finish 按钮。

(15) 在 SDK 主界面下,找到并展开 lab5。在展开项中,找到 src 目录。选中 src,单击右键,出现浮动菜单。在浮动菜单内,选择 Import。

(16) 出现 Import 对话框界面。在该界面中,展开 General。在展开项中,找到并选中 File System。

(17) 单击 Next 按钮。

(18) 在 Import 对话框界面内,单击 **Browse...** 按钮。定位到下面的路径。

E:\vivado\_example\zynq\source

(20) 从右侧窗口中选择 lab5.c 文件。

(21) 单击 Finish 按钮。

**注:** 在编译的过程中,有很多的错误。这个文件并没有完成,下面将完成代码。

(22) 选择 system.mss 标签。

**注:** 如果没有打开,则在 SDK 主界面左侧 Project Explorer 窗口下,找到并展开 standalone\_bsp。在展开项中,找到并双击 system.mss。

(23) 如图 4.87 所示,找到 ps7\_scutimer\_0,单击 Documentation。

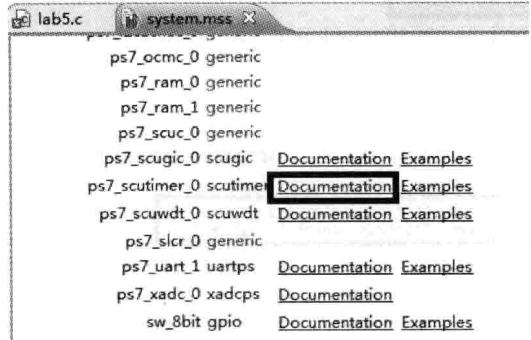


图 4.87 单击 Documentation

(24) 在浏览器窗口中自动打开 Xilinx Processor IP Library。单击 Files,查看与私有 API 函数相关的可用文件。

(25) 单击 xsctimer.h 链接,查看 API 内可用的各种函数和数据结构。图 4.88 给出了和定时器硬件交互的各种函数。

(26) 在 SDK 主界面的下方,单击 Problems 标签,双击第一个红色的位置类型名字 x。这将打开文件,帮助读者定位错误。

(27) 如图 4.89 所示,为了消除错误,添加命令用于包含头文件:

```
# include "XScuTimer.h"
```

(28) 保存文件,错误消失。

(29) 向下浏览该文件。如图 4.90 所示,可以看到几个空行上面有一些注释。这些注释用于引导读者添加设计代码。

```

Defines
#define XScuTimer_IsExpired(InstancePtr)
#define XScuTimer_RestartTimer(InstancePtr)
#define XScuTimer_LoadTimer(InstancePtr, Value)
#define XScuTimer_GetCounterValue(InstancePtr)
#define XScuTimer_EnableAutoReload(InstancePtr)
#define XScuTimer_DisableAutoReload(InstancePtr)
#define XScuTimer_EnableInterrupt(InstancePtr)
#define XScuTimer_DisableInterrupt(InstancePtr)
#define XScuTimer_GetInterruptStatus(InstancePtr)
#define XScuTimer_ClearInterruptStatus(InstancePtr)

Functions
XScuTimer_Config * XScuTimer_LookupConfig(u16 DeviceId)
int XScuTimer_SelfTest(XScuTimer *InstancePtr)
int XScuTimer_CfgInitialize(XScuTimer *InstancePtr, XScuTimer_Config *ConfigPtr, u32 EffectiveAddress)
void XScuTimer_Start(XScuTimer *InstancePtr)
void XScuTimer_Stop(XScuTimer *InstancePtr)
void XScuTimer_SetPrescaler(XScuTimer *InstancePtr, u8 PrescalerValue)
u8 XScuTimer_GetPrescaler(XScuTimer *InstancePtr)

```

图 4.88 scutimer 的 API 函数

```

lab5.c  system.mss
#include "xparameters.h"
#include "xgpio.h"
#include "xutil.h"
#include "led_ip.h"
// Include scutimer header file

// -----
XScuTimer_Timer; /* Cortex-A9 SCU_PrivateTimer Instance */

#define ONE_SECOND 33300000 // half of the CPU clock speed

int main (void)
{
    XGpio dip, push;
    int psb_check, dip_check, dip_check_prev, count, Status;

    // PS Timer related definitions
    XScuTimer_Config *ConfigPtr;
    XScuTimer *TimerInstancePtr = &timer;
    xil_printf("-- Start of the Program --\r\n");
}

```

图 4.89 程序错误(1)

```

count = 0;

// Initialize the timer

// Read dip switch values
dip_check_prev = XGpio_DiscreteRead(&dip, 1);
// Load timer with delay in multiple of ONE_SECOND

// Set AutoLoad mode

// Start the timer

```

图 4.90 添加代码指示

- (30) 如图 4.91 所示,给出了添加完成后的代码。
- (31) 向下继续浏览该文件,读者可以看到几个空行上面有一些注释,这些注释用于帮助读者添加设计代码。如图 4.92 所示,给出了添加完成后的代码。
- (32) 保存文件。

```

// Initialize the timer
ConfigPtr = XScuTimer_LookupConfig (XPAR_PS7_SCUTIMER_0_DEVICE_ID);
Status = XScuTimer_CfgInitialize(TimerInstancePtr, ConfigPtr, ConfigPtr->BaseAddr);
if(Status != XST_SUCCESS){
    xil_printf("Timer init() failed\r\n");
    return XST_FAILURE;
}

// Read dip switch values
dip_check_prev = XGpio_DiscreteRead(&dip, 1);
// Load timer with delay in multiple of ONE_SECOND
XScuTimer_LoadTimer(TimerInstancePtr, ONE_SECOND*dip_check_prev);
// Set AutoLoad mode
XScuTimer_EnableAutoReload(TimerInstancePtr);
// Start the timer
XScuTimer_Start (TimerInstancePtr);

while (1)
{
}

```

图 4.91 添加完后的代码(1)

```

if (dip_check != dip_check_prev) {
    xil_printf("DIP Switch Status %x, %x\r\n", dip_check_prev, dip_check);
    dip_check_prev = dip_check;
    // load timer with the new switch settings
    XScuTimer_LoadTimer(TimerInstancePtr, ONE_SECOND*dip_check);
    count = 0;
}
if(XScuTimer_IsExpired(TimerInstancePtr)) {
    // clear status bit
    XScuTimer_ClearInterruptStatus(TimerInstancePtr);
    // output the count to LED and increment the count
    LED_IP_mWriteReg(XPAR_LED_IP_0_S_AXI_BASEADDR, 0, count);
    count++;
}
return 0;

```

图 4.92 添加完后的代码(2)

### 4.5.3 在硬件上验证操作

本节将对设计进行验证。验证设计的步骤包括：

(1) 按照前面方法，连接 PC 和 Zedboard 之间的 JTAG 和 UART 电缆，并且给 Zedboard 上电。

(2) 选择 **Terminal 1** 标签。

注：如果没有出现该标签，则在 SDK 主界面主菜单下，选择 Window→Show View→Terminal。

(3) 单击 **N** 按钮。按 4.1.5 节介绍的参数配置 UART。

(4) 在 SDK 主界面主菜单下，选择 Xilinx Tools→Program FPGA。

(5) 出现 Program FPGA 对话框界面。在该界面中，单击 Program 按钮，将比特流文件下载到 ZYNQ-7000 中。当完成编程后，Zedboard 板上的 DONE LED 灯（蓝色）变亮。

(6) 在 SDK 主界左侧的 Project Explorer 窗口下，选择 TestApp。单击右键，出现浮动菜单。在浮动菜单下，选择 Run As→Launch on Hardware (GDB)。将应用程序下载到 ZYNQ-7000 中，并执行 ps7\_init 和 lab5.elf。

(7) 如图 4.93 所示，看到在 **Terminal 1** 标签窗口下，显示下面的结果。取决于开关

的设置,可以看到适当延迟的 LED 上的计数。

```
-- Start of the Program --
DIP Switch Status 1, 3
DIP Switch Status 3, 7
```

图 4.93 lab5.c 运行结果

#### 4.5.4 启动调试器

本节将启动调试器,对软件进行调试。启动调试器步骤主要包括:

- (1) 在 SDK 主界面左侧的 Project Explorer 窗口下,找到并选中 Lab5 工程。单击右键,出现浮动菜单。在浮动菜单内,选择 Debug As→Launch on Hardware(GDB)。
- (2) 出现 Confirm Perspective Switch 对话框界面。
- (3) 单击 Yes 按钮。该操作将停止正在执行的程序,进入到 Debug 调试界面。

读者可以使用下面的方法添加全局变量。在 Variable 标签下,单击右键,出现浮动菜单。在浮动菜单内,选择 Add Global Variables。将显示出所有的全局变量,读者选择所期望的变量。由于该程序中没有全局变量,所以没有选择全局变量的操作。

(4) 如图 4.94 和图 4.95 所示,双击该行左侧空的边界,设置断点。当设置所有的断点后,可以看到断点标记。各断点的位置为:

- ① 第一个断点设置在初始化 count 为 0 的位置;
- ② 第二个断点设置在定时器初始化失败的位置;
- ③ 第三个断点设置在程序准备读 dip 开关设置的位置;
- ④ 第四个断点设置在用于按下按键,终止程序执行的位置;
- ⑤ 第五个断点设置在当定时器超时,准备写 LED 的位置。

```
28 XGpio_Initialize(&push, XPAR_BTNS_5BIT_DEVICE_ID);
29 XGpio_SetDataDirection(&push, 1, 0xffffffff);
30
31 count = 0;
32
33 // Initialize the timer
34 ConfigPtr = XScuTimer_LookupConfig(XPAR_PS7_SCUTIMER_0_DEVICE_ID);
35 Status = XScuTimer_CfgInitialize(TimerInstancePtr, ConfigPtr, ConfigPtr->BaseAddr);
36 if(Status != XST_SUCCESS){
37     xil_printf("Timer init() failed\r\n");
38     return XST_FAILURE;
39 }
40
41 // Read dip switch values
42 dip_check_prev = XGpio_DiscreteRead(&dip, 1);
43 // Load timer with delay in multiple of ONE_SECOND
44 XScuTimer_LoadTimer(TimerInstancePtr, ONE_SECOND*dip_check_prev);
45
46 // Set AutoLoad mode
47 XScuTimer_EnableAutoReload(TimerInstancePtr);
```

图 4.94 设计断点(1)

(5) 单击 Resume 按钮 ,继续执行程序,一直到运行到第一个断点的位置。

注:在 Variable 标签下,注意到 count 的值不等于 0。

(6) 单击 Step Over 按钮  或者按 F6,执行一条语句。当执行 Step Over 时,读者

```

56     if(psb_check & 0x1)
57     {
58         XScuTimer_Stop(TimerInstancePtr);
59         break;
60     }
61     dip_check = XGpio_DiscreteRead(&dip, 1);
62     if (dip_check != dip_check_prev) {
63         xil_printf("DIP Switch Status %x, %x\r\n", dip_check_prev, dip_check);
64         dip_check_prev = dip_check;
65         // load timer with the new switch settings
66         XScuTimer_LoadTimer(TimerInstancePtr, ONE_SECOND*dip_check);
67         count = 0;
68     }
69     if(XScuTimer_IsExpired(TimerInstancePtr)) {
70         // clear status bit
71         XScuTimer_ClearInterruptStatus(TimerInstancePtr);
72         // output the count to LED and increment the count
73         LED_IP_mWriteReg(XPAR_LED_IP_S_AXI_BASEADDR, 0, count);
74         count++;
75     }

```

图 4.95 设计断点(2)

可以注意到 count 的变量值改变为 0。

(7) 再次单击 Resume 按钮 ，可以看到执行了一些行，然后执行到第三个断点停下来。由于成功地初始化定时器，所以跳过了第二个断点。

(8) 单击 Step Over 按钮 或者按 F6，执行一条语句。当执行 Step Over 时，读者可以注意到，根据于板上开关设置，dip\_check\_prev 变量的值也发生了改变。

(9) 单击 memory 标签。

**注：**如果没有出现该标签，则在 SDK 主界面主菜单下，选择 Window→Show View→Memory。

(10) 如图 4.96 所示，单击 符号，添加一个存储器监控器。

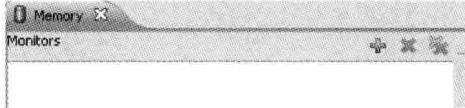


图 4.96 存储器监控界面

(11) 如图 4.97 所示，输入私有计数器加载寄存器(0xF8F00600)的地址。

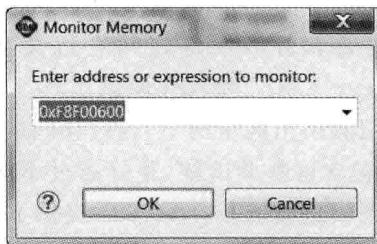


图 4.97 输入存储器地址界面

(12) 单击 OK 按钮。

**注：**读者可以在 xparameters.h 文件中找到该地址(# XPAR\_PS7\_SCUTIMER\_0\_BASEADDR)。首先，通过在 outline 窗口内双击 xscutimer.h。然后，在展开项中找到并

双击 xscutimer\_hw.h。最后,如图 4.98 所示,选择 XSCUTIMER\_LOAD\_OFFSET。

```
# XSCUTIMER_HW_H
# xil_types.h
# xil_io.h
# XSCUTIMER_LOAD_OFFSET
# XSCUTIMER_COUNTER_OFFSET
# XSCUTIMER_CONTROL_OFFSET
# XSCUTIMER_ISR_OFFSET
```

图 4.98 偏移地址界面

(13) 单击 Step Over 按钮 或者按 F6,执行一条语句。该语句将加载定时器寄存器。

**注:** 由于内容发生变化,所以地址 0xF8F00604 变成红色。验证内容和值 dip\_check\_prev \* 333000000 相同。

(14) 再次单击 Resume 按钮 ,可以看到执行了程序的一些行,停到写 LED 端口的断点。由于没有按 Zedboard 中心的按键,所以跳过第四个断点。

(15) 单击 Step Over 按钮 或者按 F6,执行一条语句。该语句将写 LED 端口,关闭 LED。

(16) 双击第五个断点,程序能自由地执行。

(17) 再次单击 Resume 按钮 ,继续执行程序。这次按照开关设置地速度,将连续地运行程序改变 LED 的亮模式。

(18) 触发开关,改变延迟,观察其效果。

(19) 按下 Zedboard 开发板的中心按键,观察程序停到第四个断点。由于调用停止定时器函数,计数器的值和控制寄存器的值发生改变。所以,定时器寄存器内容和控制寄存器(偏移 0x08)变红。

(20) 单击 按钮,终止任务。

(21) 关闭 SDK 和 Vivado。

## 4.6 使用硬件分析仪调试

在一个嵌入式系统中,需要软件和硬件之间进行交互。Xilinx SDK 包含 GNU、Xilinx Microprocessor Debugger(XMD)和软件调试工具。硬件分析仪允许设计者在不需要将内部逻辑通过 FPGA 引脚引出的情况下,通过访问内部的信号实现硬件调试。在器件的可编程逻辑部分,允许包含这些调试核,并且通过不同的方式对这些核进行配置。这样,就可以对 PL 内的硬件逻辑进行监控。Vivado 提供了在设计不同阶段标记任何网络的能力。

本节将在设计中完成下面的功能:

- (1) 添加 VIO 核;
- (2) 使用 VIO 插入激励源和监视响应;
- (3) 标记需要调试的网络来监视 AXI 的交易;

- (4) 在 Vivado 中,添加 ILA 核;
- (5) 使用硬件分析仪,执行硬件调试;
- (6) 使用 SDK,执行软件调试。

如图 4.99 所示,添加一个定制核执行一个简单的 8 位加法功能。这个核预先使用 Vivado 的 IP 封装器开发,作为该设计的一部分。这个核有一些额外的端口,用于引入激励源和监视响应。通过这个方法就可以在不使用 PS 或者软件应用程序的情况下,测试该定制 IP 核。

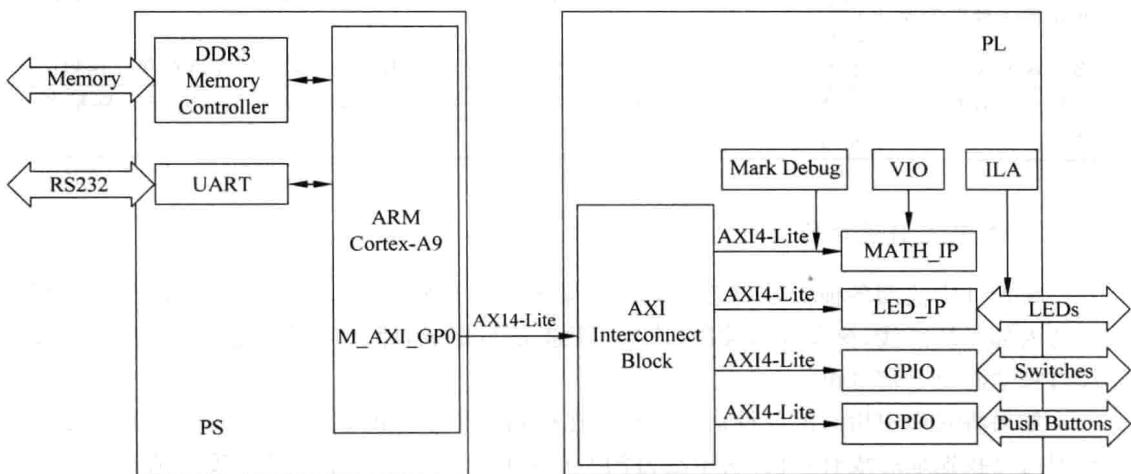


图 4.99 加入调试模块的系统结构

**注:** 关于逻辑分析工具更详细的资料,请参考《Xilinx FPGA 设计权威指南》,清华大学出版社,2012。

ILA 核是一个可用于监测设计中的任意内部信号的定制逻辑分析仪核。由于 ILA 核与被监测的设计同步,因此应用于设计中的所有设计时钟的约束也可应用于 ILA 核的内部元件。ILA 核包括三个主要组成部分:

- (1) 触发器输入和输出逻辑,其中包括:触发输入逻辑检测详细触发事件;触发输出逻辑触发外部测试设备和其他逻辑。
- (2) 数据捕获逻辑,ILA 核使用片上块 RAM 资源来捕获并存储跟踪数据信息。
- (3) 控制和状态逻辑,管理 ILA 核的操作。

#### 4.6.1 ILA 核原理

##### 1. ILA 触发器输入逻辑

表 4.1 给出了 ILA 触发器的特征。ILA 核的触发能力包括许多特征,这些特征是检测详细触发事件所必需的。

表 4.1 ILA 触发器的特征

特    征	描    述
宽触发器端口	每个触发器端口能够达到 1~256 位宽
多触发器端口	每个 ILA 核最多可以有 16 个触发端口。复杂系统中不同的信号类型或总线，需要使用各自的匹配单元来监测。因此，在系统中支持多触发端口是必要的
每个触发端口有多 个匹配单元	每个触发端口可以连接多达 16 个匹配单元。这个特征使得可以对触发器端口信号进行多个比较
布尔等式触发条件	触发条件由一个布尔 AND 或 OR 等式组成，该等式包含多达 16 个匹配单元函数
多级触发时序器	触发条件由多级触发序列器组成，该触发序列器包含多达 16 个匹配单元函数
布尔等式存储限定 条件	存储限定条件由一个布尔 AND 或 OR 等式组成，该等式包含多达 16 个匹配单元函数

## 2. 多触发器端口的使用

在设计中监测各种信号和总线的能力需要使用多触发端口。如果正在测量一个设计的内部系统总线，该设计由控制、地址和数据信号组成，那么可以指定一个单独的触发端口来监测每个信号组。

如果将它们中所有的各种信号和总线连接到一个单一的触发端口，那么当在指定的范围内寻找地址总线时，将无法单独监测 CE、WE 和 OE 信号上的位跳变。通过从不同匹配单元类型中进行灵活的选择，从而可以定制 ILA 核来满足触发需求。同时，保持使用最少的逻辑设计资源。

## 3. 使用触发器和存储限定条件

触发条件是一个布尔值或事件的连续组合，由连接在核触发器端匹配单元的比较器进行监测。在数据捕获窗口，触发条件用来标记一个明显的初始点，该点可设在数据捕获窗口的开始、结束或窗口中的任何位置。

同样，存储限制条件也是事件的布尔组合，这些事件由连接到核触发端口的匹配单元比较器检测。然而，存储限制条件不同于触发条件，它评估触发端口匹配单元事件，以决定是否要捕获和存储每个数据样本。触发和存储限制条件可一起使用，以确定开始捕获过程的时间以及捕获数据的类型。

如图 4.100 所示，假设有如下任务：

- (1) 第一个存储器写周期触发(CE=上升边沿, WE=1, OE= 0 )目标地址 = 0xFF0000。
- (2) 存储器读周期(CE=上升沿, WE=0, OE=1)从地址 = 0x23AAC 捕获，其数据值在 0x00000000 和 0x1000FFFF 之间。

为了成功地实现这些条件，需要确保 TRIG0 和 TRIG1 触发器端口都连接两个匹配单元：一个用于触发条件，一个用于存储限制条件。

下面将建立触发器和存储限制等式以及每个匹配单元，以满足上述条件要求：

- (1) 触发条件 = M0 && M2，其中：

① M0 [2: 0]=CE, WE, OE= “R10”(其中 R 代表“上升沿”);

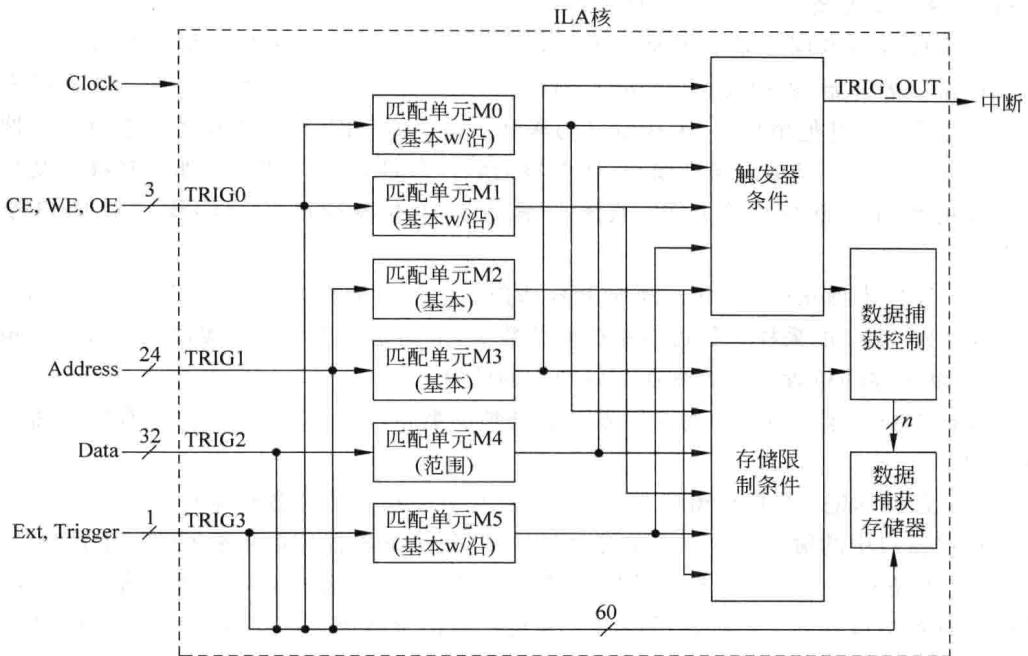


图 4.100 ILA 核例子

② M2[23:0] 地址 = “FF0000”。

(2) 存储限定条件 = M1&&M3&&M4, 其中：

① M1[2:0]=CE,WE,OE=“R10”(其中 R 代表“上升沿”);

② M3[23:0] 地址 = “23AAAC”;

③ M4[31:0] 数据 = 范围为 0x00000000~0x1000FFFF。

#### 4. ILA 触发器输出逻辑

ILA 核实现了触发器输出端口 TRIG\_OUT。TRIG\_OUT 端口输出触发条件, 由分析仪在运行时建立。触发输出的类型(电平或脉冲)和敏感信号(高电平有效或低电平有效)也可在运行时控制。与输入触发端口相关的 TRIG\_OUT 端口的延迟是 10 个时钟周期。

TRIG\_OUT 端口非常灵活且有许多用途。连接 TRIG\_OUT 端口到器件引脚以触发外部测试设备, 如示波器和逻辑分析仪。连接 TRIG\_OUT 端口到嵌入式处理器的中断线, 从而产生软件事件。通过将一个核的 TRIG\_OUT 端口连接到另一个核的触发输入端口, 就可以扩展用于片上调试解决方案的触发和数据采集能力。

#### 5. ILA 数据捕获逻辑

在设计中, 每个 ILA 核能够使用独立于其他核的片上 BRAM 资源来捕获数据。每个 ILA 核可以使用两种捕获模式来捕获数据：窗口和 N 样本。

(1) 窗口捕获模式：在窗口捕获模式中, 样本缓冲区可分为一个或多个同等规模样本窗口。窗口捕获模式使用一个单一的事件触发条件(即个别触发器匹配单元事件的布

尔组合)来收集足够的数据,填充样本窗口。

当采样窗口的深度是 2 的幂次方时,可以达到 131072 次采样。触发位置可以设置在采样窗口的开始(先触发,再收集)、采样窗口的末尾(收集直到触发事件)、或采样窗口的任何地方:在其他情况下,采样窗口的深度不是 2 的幂次方,触发位置只能设定在抽样窗口的起始点。一旦填满采样窗口,ILA 核的触发条件就会自动重新加载并继续监测触发条件的事件。重复这个过程,直到填满所有样本缓冲区的抽样窗口或用户停止 ILA 核。

(2) N 样本捕获模式: N 样本捕获模式类似于窗口捕捉模式,除了以下两个差别:

- ① 每个窗口的采样个数可以是任何整数 N,范围是从 1 到样本缓冲深度减 1 的值;
- ② 触发器的位置必须始终在窗口中 0 的位置上。

N 样本捕获模式有利于每个触发器捕获精确数量的样本,同时不浪费有用的捕获存储资源。

(3) 触发标记: 在抽样窗口中,标记与触发事件相一致的数据采样。该触发标志通知分析仪窗口中的触发位置。在抽样缓冲区中,每个触发标记消耗额外的一个比特位。

(4) 数据端口: ILA 核提供了捕获端口数据的能力,这些端口与用于执行触发器函数的触发器端口分开。该特性可以将被捕获的数据量限制到一个相对小的个数,这是因为不需要捕获和查看相同的用于触发核的信息。

然而,在许多情况下,捕获和观察用来触发核的相同的数据是有用的。在这种情况下,设计者可以选择来自一个或多个触发端口的数据。此特性可节省资源,同时提供了选择感兴趣的触发信息进行捕获的灵活性。

## 6. ILA 控制与状态逻辑

ILA 核含有少量的控制和状态逻辑,用来保证核正常操作。识别 ILA 核并与之通信的所有逻辑由控制和状态逻辑实现。

### 4.6.2 VIO 核原理

**虚拟输入/输出(Virtual Input/Output, VIO)**核是一个可定制的核,可以实时监测和驱动 FPGA 内部信号。与 ILA 和 IBA 核不同,不需要片上或片外 RAM。图 4.101 给出了 VIO 的内部结构图。

在 VIO 核中有 4 种信号:

- (1) 异步输入: 使用由 JTAG 电缆驱动的 JTAG 时钟信号进行采样,定期读取输入值并将其显示在分析仪上;
- (2) 同步输入: 使用设计时钟进行采样,定期读取输入值并将其显示在分析仪上;
- (3) 异步输出: 在分析仪中,由用户定义,从核输出到周围的设计。每个独立的异步输出口可以定义成逻辑 1 或逻辑 0;
- (4) 同步输出: 在分析仪中,由用户定义,与设计时钟同步,从核输出到周围的设计。单独的同步输出可以定义成逻辑 1 或逻辑 0。16 个时钟周期的脉冲序列(1 和/或 0 的)也可以被定义成同步输出。

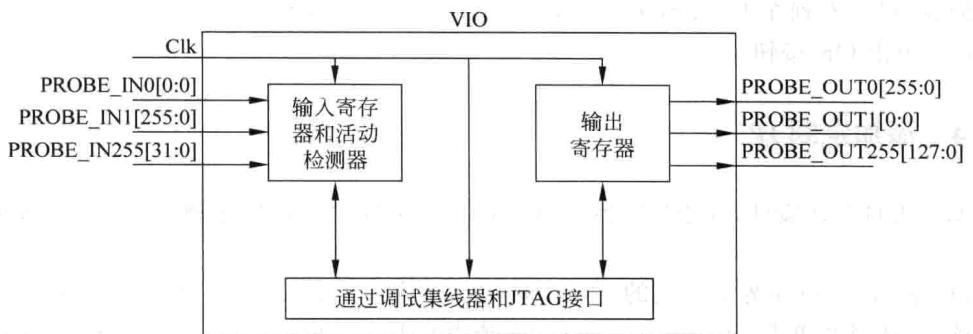


图 4.101 VIO 块图

### 1. 活动检测器

每个 VIO 核输入有额外的单元捕获输入上的跳变。由于设计时钟可能比分析仪的采样周期更快,被监测信号可能在连续采样间跳变很多次。活动探测器捕获这些行为,显示结果。如果是同步输入,使用能够监测异步和同步事件的活动单元。这些特性可用于检测毛刺和同步输入信号上的同步转换。

### 2. 脉冲序列

每个 VIO 同步输出可以输出静态 1、静态 0,或者连续值的脉冲序列。脉冲序列是一个 16 个时钟周期的 1 和 0 交替变化的序列。在连续设计时钟周期中,从核中输出该序列。在分析仪中定义脉冲序列并在装入核之后仅执行一次。

此外,根据与 FPGA 接口的要求,定制输入和输出端口的个数和宽度。由于 VIO 与正在监视事件和/或者驱动事件相同步,所有应用到设计中的时钟约束,也应用到 VIO 核内的元件。VIO 核的实时交互需要使用 Vivado 逻辑分析仪的特性。

## 4.6.3 打开工程

在 \vivado\_example\zynq 目录下,新建一个名字为 lab6 的目录。并且,将 lab5 目录下的所有文件复制到 lab6 目录下。打开工程的步骤主要包括:

- (1) 启动 Vivado 集成开发环境。
- (2) 在 Get Started 页面下,选择 Open Project。
- (3) 选择 Browse Project。
- (4) 定位到目录\zynq\lab6。在该目录下,双击 lab1.prj。
- (5) 在 Vivado 主界面左侧的 Flow Navigator 窗口下,找到并展开 Project Manager。在展开项中,找到并单击 Project Settings。
- (6) 出现 Project Settings 对话框界面。在该界左侧窗口中,选择 IP 条目。在右侧窗口中,单击 **Add Repository...** 按钮,定位到下面的路径:

e:/vivado\_example/zynq/math\_ip

同时,可以看到在 IP in Selected Repository 窗口下,出现 math\_ip\_v1\_0。

(7) 单击 OK 按钮。

#### 4.6.4 添加定制 IP

本节将打开块设计,并添加定制的 math IP 到系统中。添加定制 math IP 的步骤主要包括:

- (1) 在 Vivado 主界面左侧的 Flow Navigator 窗口下,找到并展开 IP Integrator。在展开项中,选择并单击 Open Block Design; 或者在源文件窗口内,选择并双击 system.bd 文件。通过这两种方法打开块设计。
- (2) 在 Diagram 窗口左侧的一列工具栏内,单击 按钮。
- (3) 出现 IP Catalog 对话框界面,在 Search 后面输入 math,找到并双击 math\_ip\_v1\_0。
- (4) 在 Diagram 窗口顶部,找到并单击 Run Connection Automation 消息,并且选择/math\_ip\_0/S\_AXI。

**注:** 该定制 IP 由层次化设计构成,底层的模块执行加法操作。高层模块包含两个从寄存器。

图 4.102 给出了定制 math IP 的加法器模块结构。

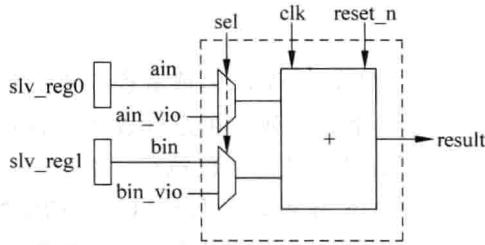


图 4.102 定制 math IP 的加法模块

#### 4.6.5 添加 ILA 和 VIO 核

本节将添加 ILA 核和 VIO 核,添加 ILA 和 VIO 核的步骤主要包括:

- (1) 在 Diagram 窗口左侧的一列工具栏内,单击 按钮。
- (2) 出现 IP Catalog 对话框界面,在 Search 后面输入 ILA,找到并双击 ILA (Integrated Logic Analyzer),将它的一个例化 ila\_0 模块添加到设计中。
- (3) 在 Diagram 窗口内找到并双击 ila\_0 模块符号。
- (4) 出现 Re-customize IP - ILA(Integrated Logic Analyzer)(3.0)对话框界面。在该界面中,单击 Probe\_Ports(0..0)标签。在该标签下,将 Probe Width[1..4096]的值改成 8。
- (5) 单击 OK 按钮。
- (6) 使用绘图工具,将 ila\_0 模块的 PROBE0 端口连接到 led\_ip 模块的 LED 端口。
- (7) 使用绘图工具,将 ila\_0 模块的 CLK 端口连接到其他模块的 s\_axi\_aclk 端口。
- (8) 在 Diagram 窗口左侧的一列工具栏内,单击 按钮。

(9) 出现 IP Catalog 对话框界面,在 Search 后面输入 vio,找到并双击 VIO(Virtual Input/Output),将它的一个例化 vio\_0 模块添加到设计中。

(10) 在 Diagram 窗口下,找到并双击 vio\_0 实例。

(11) 出现 Re-customize IP-VIO (Virtual Input/Output) (3.0) 对话框界面。在 General Options 标签窗口下,按下面设置参数:

① Input Probe Count: 1;

② Output Probe Count: 3。

在 PROBE\_IN Ports(0..0)标签窗口下,按下面设置参数:

① PROBE\_IN0 所对应的 Probe Width: 9。

在 PROBE\_OUT Ports(0..0)标签窗口下,按下面设置参数:

① PROBE\_OUT0 所对应的 Probe Width: 1;

② PROBE\_OUT1 所对应的 Probe Width: 8;

③ PROBE\_OUT2 所对应的 Probe Width: 8。

(12) 使用绘图工具,连接下面的端口:

① PROBE\_IN→result;

② PROBE\_OUT0→sel;

③ PROBE\_OUT1→ain\_vio;

④ PROBE\_OUT2→bin\_vio。

(13) 使用绘图工具,将 vio\_0 模块的 CLK 端口连接到其他模块的 s\_axi\_aclk 端口。

(14) 在 Diagram 窗口的一列工具栏内,单击 按钮,重新绘制系统结构图。图 4.103 给出了重新绘制后的系统结构图。

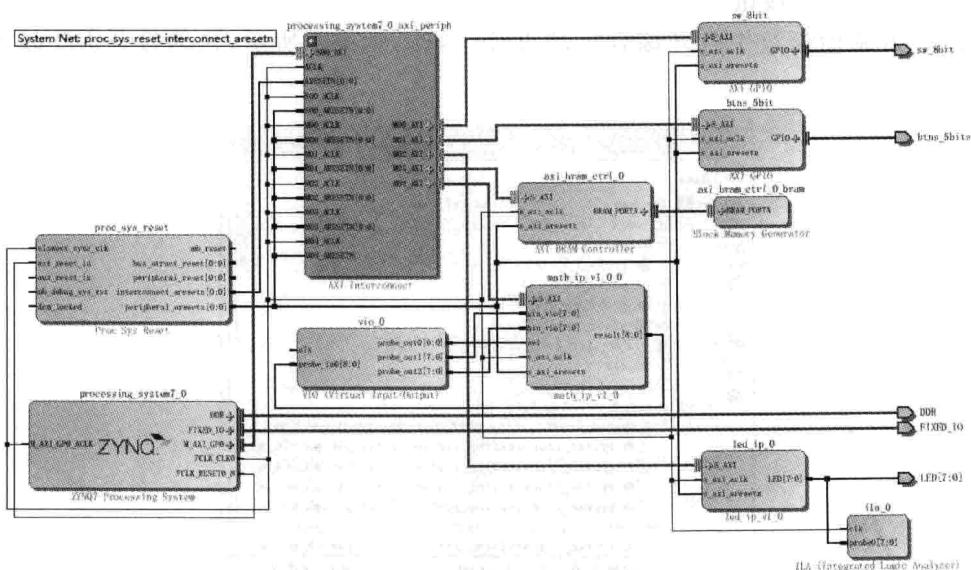


图 4.103 重新绘制后的系统结构图

#### 4.6.6 标记和分配调试网络

本节将 AXI 互连和 math\_ip\_0 实例中的 S\_AXI 连接标记为调试，并验证设计有效性。标记调试的步骤主要包括：

- (1) 选择 AXI 互连和 math\_ip\_0 之间的 S\_AXI 互连。单击右键，出现浮动菜单。在浮动菜单内，选择 Mark Debug 选项，该选项用于监视 AXI4Lite 交易。
- (2) 在 Vivado 主界面下，选择 Tools→Validate Design，以运行设计规则检查。
- (3) 出现 Validate Design 对话框界面，单击 OK 按钮。
- (4) 出现 Create HDL Wrapper 对话框界面。在该界面内，选中 Let Vivado manage wrapper and auto-update 前面的复选框。
- (5) 单击 OK 按钮。
- (6) 在 Vivado 主界面左侧的 Flow Navigator 窗口下，找到并展开 Synthesis。在展开项中，找到并单击 Run Synthesis。
- (7) 出现 Run Synthesis 对话框界面。提示重新运行综合。
- (8) 单击 OK 按钮。
- (9) 出现 Save Project 对话框界面。
- (10) 单击 Save 按钮。
- (11) 等待综合完成后，出现 Synthesis Completed 对话框界面。在该对话框界面中，选中 Open Synthesized Design 复选框。
- (12) 单击 OK 按钮。
- (13) 在辅助面板中打开综合后的设计。如图 4.104 所示，自动打开 Debug 标签窗口。

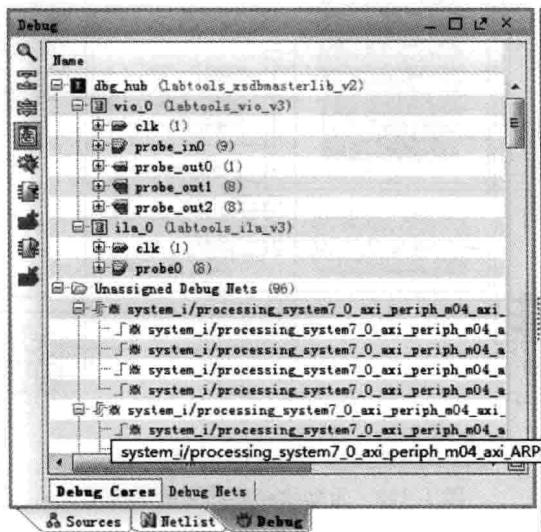


图 4.104 Debug 标签界面

注：如果没有出现 Debug 标签，则在 Vivado 主界面主菜单下，选择 Window→Debug。从图中可以看出，可以被调试的网络分成已分配的和未分配的组。已分配的组包含与 VIO 和 ILA 核相关联的网络。然而，未分配的网络组包含与 S\_AXI 相关的网络。

(14) 选中 Unassigned Debug Nets，单击右键，出现浮动菜单。在浮动菜单内，选择 Set up Debug...选项。

(15) 出现 Setup Debug 对话框界面。

(16) 单击 Next 按钮。

(17) 如图 4.105 所示，出现 Set up Debug-Specify Nets to Debug(设置调试-指定调试的网络)对话框界面。

(18) 在该图中，分别选中 BRESP 和 RRESP(由 GND 驱动)，单击右键，出现浮动菜单。在浮动菜单内，选择 Remove Nets。

(19) 单击 Next 按钮。

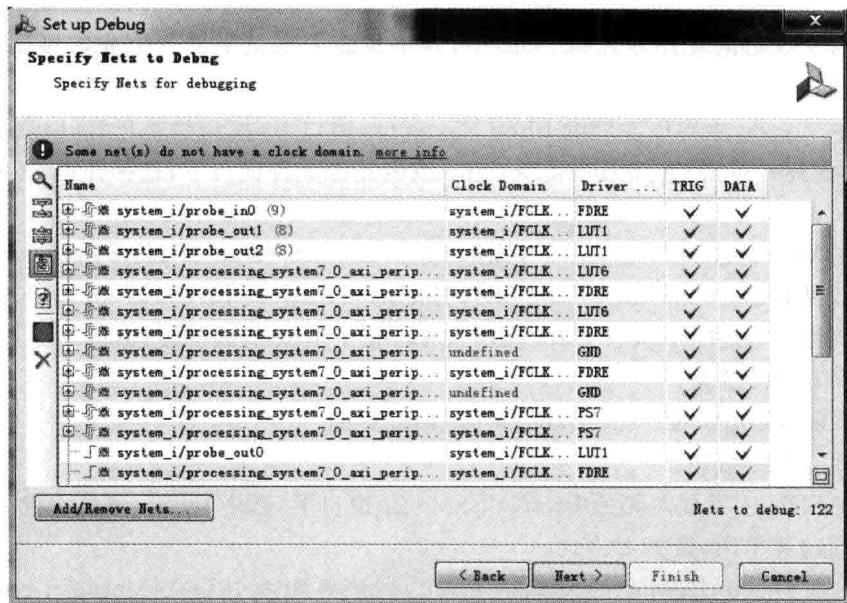


图 4.105 设置网络对话框界面

(20) 出现 Set up Debug-Trigger and Capture Modes(设置调试-触发器和捕获模式)对话框界面。不选择任何选项。

(21) 单击 Next 按钮。

(22) 出现 Set up Debug-Set up Debug Summary(设置调试-设置调试总结)对话框界面。

(23) 单击 Finish 按钮。

(24) 按 Ctrl+S 按键，保存设计。

(25) 对设计进行重新综合。

#### 4.6.7 生成比特流文件

本节将生成比特流文件。生成比特流文件的步骤主要包括：

- (1) 在 Vivado 主界面左侧的 Flow Navigator 窗口下, 选择并展开 Program and Debug。在展开项中, 选择并单击 Generate Bitstream, 运行实现和生成比特流的过程。
- (2) 出现对话框界面, 单击 Save 按钮, 保存工程。
- (3) 出现对话框界面, 单击 Yes 按钮, 运行设计。
- (4) 当成功生成比特流后, 出现对话框界面。在该界面中, 选中 Open Implemented Design 前面的复选框。
- (5) 单击 OK 按钮。

#### 4.6.8 生成测试程序

本节将实现后的设计导入到 SDK 中, 并生成新的测试工程。生成新测试工程的步骤主要包括：

- (1) 在 Vivado 主界面左侧的 Flow Navigator 窗口下, 找到并展开 ID Integrator。在展开项中, 找到并单击 Open Block Design; 或者在源文件窗口下, 选择并单击 system.bd 文件。
- (2) 在 Vivado 主界面主菜单下, 选择 File→Export→Export Hardware for SDK。
- (3) 出现 Export Hardware for SDK-Export hardware platform for SDK 对话框界面。在该界面中, 选中 Export Hardware 和 Launch SDK 前面的复选框。
- (4) 单击 OK 按钮。
- (5) 出现 Module Already Exported 对话框界面。
- (6) 单击 Yes 按钮。
- (7) 在 SDK 主界面左侧的 Project Explorer 窗口下, 选中 lab5。单击右键, 出现浮动菜单。在浮动菜单内, 选择 Delete。
- (8) 出现 Delete Resources 对话框界面。在该界面选中 Delete project contents on disk(cannot be undone)前面的复选框。
- (9) 单击 OK 按钮。
- (10) 在 SDK 主界面主菜单下, 选择 Xilinx Tools→Repositories。
- (11) 出现 Preferences 对话框界面。在该对话框界面下, 单击 **New...** 按钮。将路径定位到:  
E:\vivado\_example\zyng\math\_ip\math\_ip\_v1\_0\_1.0
- (12) 单击 OK 按钮。
- (13) 在 SDK 主界面主菜单下, 选择 Xilinx→Board Support Package Settings。
- (14) 出现 Select a board support package(选择一个板级支持包)对话框界面。在该界面内, 选择 standalone\_bsp。

- (15) 单击 OK 按钮。
- (16) 出现 Board Support Package Settings(板级支持包)对话框界面。在该界面左侧选择 drivers。在 led\_ip\_0 所对应的 Driver 列下面,通过下拉框选择 generic。
- (17) 单击 OK 按钮。
- (18) 在 SDK 主界面主菜单下,选择 File→New→Application Project。
- (19) 出现 New Project-Application Project 对话框界面。按下面设置参数:
- ① Project name: lab6;
  - ② 选中 Use existing 前面的复选框。
- (20) 单击 Next 按钮。
- (21) 出现 New Project-Templates 对话框界面。在该对话框界面中的 Available Templates 下,选择 Empty Application。
- (22) 单击 Finish 按钮。
- (23) 在 SDK 主界面下,找到并展开 lab6。在展开项中,找到 src。选中 src,单击右键,出现浮动菜单。在浮动菜单内,选择 Import。
- (24) 出现 Import 对话框界面。在该界面中,展开 General。在展开项中,找到并选中 File System。
- (25) 单击 Next 按钮。
- (26) 在 Import 对话框界面中,单击  按钮。定位到:  
E:\vivado\_example\zynq\source
- (27) 从该界面的右侧窗口中,选择 lab6.c 文件。
- (28) 单击 Finish 按钮。

如图 4.106 所示,写操作数到定制的核,读结果,然后打印结果。下面将使用写交易作为 ChipScope Analyzer 触发器条件。

```
xil_printf("-- Change slide switches to see corresponding output "
"on LEDs --\r\n");
Xil_Out32(XPAR_MATH_IP_V1_0_0_S_AXI_BASEADDR, 0x12);
Xil_Out32(XPAR_MATH_IP_V1_0_0_S_AXI_BASEADDR+4, 0x34);
i=Xil_In32(XPAR_LED_IP_0_S_AXI_BASEADDR);
xil_printf("result=%x\r\n",i);

while ([1])
{
    sw_check = XGpio_DiscreteRead(&sw, 1);
    LED_IP_mWriteReg(XPAR_LED_IP_0_S_AXI_BASEADDR, 0, sw_check);
    for (i=0; i<9999999; i++) // delay loop
}
}
```

图 4.106 代码片段

#### 4.6.9 测试和调试

本节将对设计进行验证。验证设计的步骤包括:

- (1) 按照前面方法,连接 PC 和 Zedboard 之间的 JTAG 和 UART 电缆,并且给

Zedboard 上电。

(2) 选择 标签。

**注：**如果没有出现该标签，则在 SDK 主界面主菜单下，选择 Window→Show View→Terminal。

(3) 单击 按钮。按 4.1.5 节介绍的参数配置 UART。

(4) 在 SDK 主界面主菜单下，选择 Xilinx Tools→Program FPGA。

(5) 出现 Program FPGA 对话框界面。在该界面中，单击 Program 按钮，将比特流文件下载到 ZYNQ-7000 中。当完成编程后，Zedboard 板上的 DONE LED 灯(蓝色)变亮。

(6) 在 SDK 主界面左侧的 Project Explorer 窗口下，选择 TestApp。单击右键，出现浮动菜单。在浮动菜单下，选择 Debug As→Launch on Hardware (GDB)，执行应用程序。

(7) 出现一个对话框界面，询问是否切换到调试器界面。

(8) 单击 YES 按钮，并切换到调试器界面。开始执行程序，并且停到入口点。

(9) 在 Vivado 主界面左侧的 Flow Navigator 窗口下，找到并展开 Program and Debug。在展开项中，找到并单击 Hardware Manager。

(10) 按前面的方法，单击 Open a New Hardware Target，建立主机和 Zedboard 之间的连接。

(11) 出现 Open New Hardware Target 对话框界面。

(12) 单击 Next 按钮。

(13) 出现 Open New Hardware-Vivado CSE Server Name 对话框界面。

(14) 单击 Next 按钮。

**注：**将扫描 JTAG，检测到器件。

(15) 出现 Open New Hardware Target-Select Hardware Target(打开新硬件目标-选择硬件目标)对话框界面。

(16) 单击 Next 按钮。

(17) 出现 Open New Hardware Target-Set Hardware Target Properties(打开新硬件目标-设置硬件目标属性)对话框界面。

(18) 单击 Next 按钮。

(19) 出现 Open New Hardware Target-Open Hardware Target Summary(打开新硬件目标-打开硬件目标总结)对话框界面。

(20) 单击 Finish 按钮。

(21) 如图 4.107 所示，在 Debug Probes 标签下，打开硬件任务。此时，hardware session 状态窗口也打开了，显示已经编程 FPGA(在 SDK 中执行)，有三个颜色，两个 ila 处于空闲状态。

(22) 选择 XC7Z020-1，单击 Run Trigger Immediate 按钮，然后在波形窗口中观察波形。

(23) 在 Debug Probe 窗口下，选择 AWADDR 总线并将其拖拽到 ILA-hw\_il\_2 窗口，释放以便添加它到窗口中，设置条件。

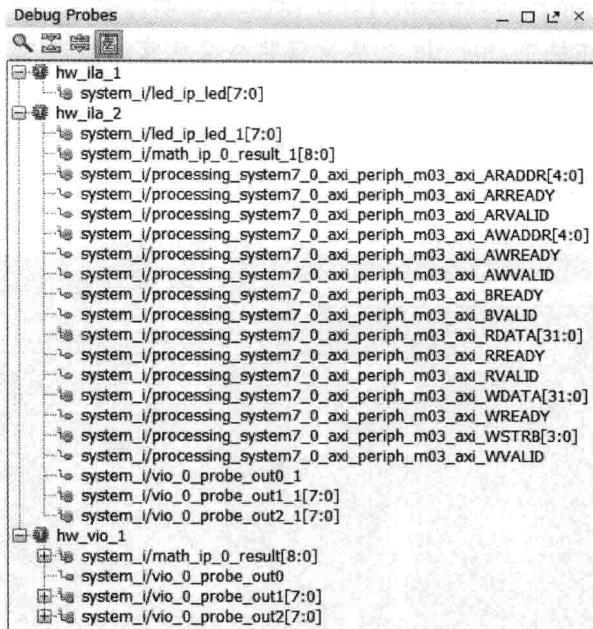


图 4.107 Debug Probes 窗口

- (24) 将值从 xx 改变到 04 (math\_1 实例的 slave\_reg2 地址)。
- (25) 类似地,添加 WSTRB 和 WVALID 信号,将其值从 xxxx 改成 xxx1,以及 1。
- (26) 如图 4.108 所示,将 hw\_ilas\_2 触发器的位置设置为 512。

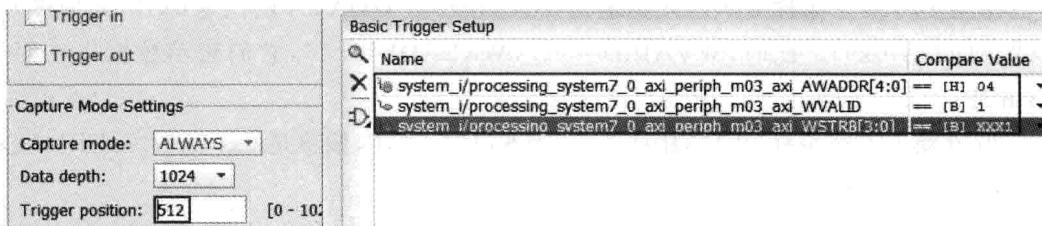


图 4.108 设置触发器位置窗口

- (27) 类似地,将 hw\_ilas\_1 触发器的位置设置为 512。
- (28) 在 Hardware 窗口下,选择 hw\_ilas\_2。
- (29) 单击 Run Trigger 按键,观察 hw\_ilas\_2 核,此时显示 Waiting For Trigger 状态。
- (30) 如图 4.109 所示,在 SDK 的 lab6.c 文件中,按图所示的位置设置断点。

```
Xil_Out32(XPAR_MATH_IP_V1_0_0_S_AXI_BASEADDR, 0x12);
Xil_Out32(XPAR_MATH_IP_V1_0_0_S_AXI_BASEADDR+4, 0x34);
i=xil_In32(XPAR_LED_IP_0_S_AXI_BASEADDR);
xil_printf("result=%x\r\n",i);
```

图 4.109 设置断点

(31) 单击 Resume 按钮,继续执行程序,然后停在断点的位置。

**注:** 在 Vivado 环境下,hw\_il\_ila\_2 从捕获状态变成空闲状态,波形显示了触发后的输出。

(32) 如图 4.110 所示,将光标移动到靠近触发器点的位置。然后,单击 按钮。不断单击 Zoom In 按钮,查看触发点附近的活动行为。

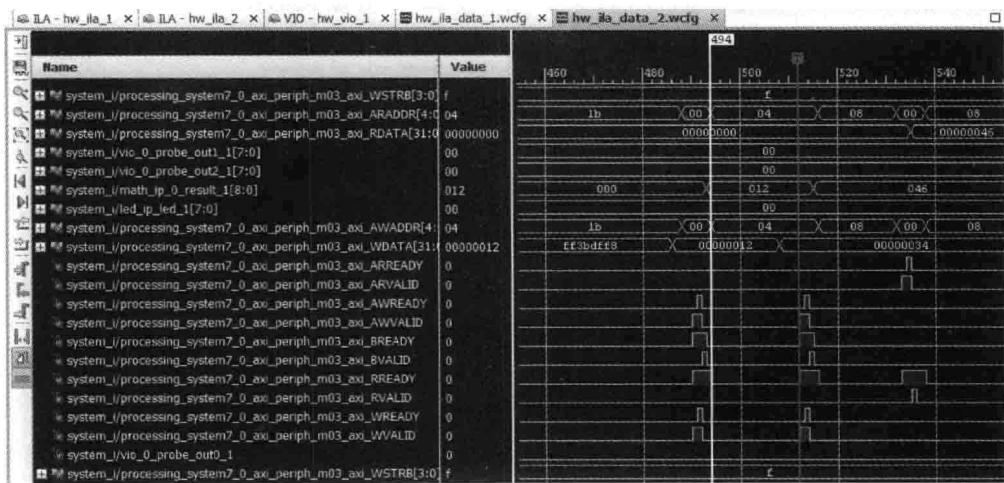


图 4.110 波形窗口

观察下面的采样:

① 在第 490 个采样 RDATA 的值是 0x00,正在写 WDATA 的值是 0x12,其偏移为 0 (AWADDR=0x0)。此时, WVALID=‘1’, WREADY=‘1’, 它们表示数据正在写入 math IP;

② 在第 512 个采样, WVALID=‘1’, WSTRB=0xF, 偏移是 0x4(AWADDR), 表示正在写入的数字为 0x034;

③ 在第 536 个采样, RREADY 和 RVAILD 为‘1’, 表示正在从 IP 读出数据, 其偏移为 0(ARADDR)。

(33) 观察在超级终端上的输出。

在 Vivado 环境中,选择 VIO 核,添加信号到 VIO-hw\_vio\_1 窗口,设置 vio\_0\_probe\_out0。这样,就能通过 VIO 人工控制 math\_ip 的输入。尝试为两个操作数输入不同的值,在 Console 窗口下,观察 math\_ip\_0\_result 端口上输出的结果。实现操作 Math IP 的步骤主要包括:

(1) 在 Debug Probes 下,选择 VIO Cores 的所有信号,将它们拖曳到 VIO-hw\_vio\_1 窗口。

(2) 如图 4.111 所示,选择 vio\_0\_probe\_out0。然后将其值修改为 1。这样就可以通过 VIO 的核控制 math\_ip 的输入。

(3) 如图 4.112 所示,单击 vio\_0\_probe\_out1 右侧,将其值改为 55(十六进制)。类似地,单击 vio\_0\_probe\_out2 右侧,将其值改为 44(十六进制)。

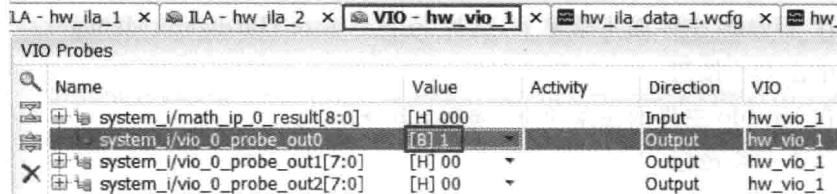


图 4.111 设置 probe0 窗口

system_i/math_ip_0_result[8:0]	[H] 099
system_i/vio_0_probe_out0	[B] 1
system_i/vio_0_probe_out1[7:0]	[H] 55
system_i/vio_0_probe_out2[7:0]	[H] 44

图 4.112 设置操作数窗口

注：一个短暂时刻后，在活动列中出现一个蓝色的向上的箭头，结果变成 099（十六进制）。

- (4) 尝试其他操作数，并观察结果。
- (5) 选择 vio\_0\_probe\_out0，将其值变为 0。这样，将断开 VIO 的核和 math\_ip 的交互。
- (6) 在 ILA Cores 标签窗口下，选择 hw\_il\_1。然后，将 led\_ip\_led 信号拖到 ILA-hw\_il\_1 窗口。
- (7) 如图 4.113 所示，设置 LED 输出值为 55 时，hw\_il\_1 的触发条件触发捕获。

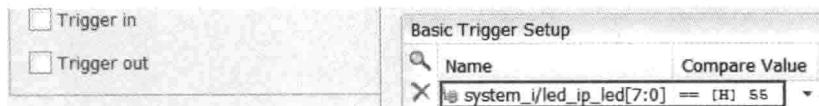


图 4.113 设置触发条件

- (8) 将 hw\_il\_1 触发器的位置设置为 512。确认开关没有设置到 01010101。
- (9) 在 Hardware 窗口下，选择 hw\_il\_1，单击右键。在浮动菜单内，选择 Run Trigger。
- (10) 在 SDK 窗口下，单击 Resume 按钮。
- (11) 改变开关，观察相应的 LED 的 ON 或者 OFF 状态。
- (12) 当满足条件(0x55)时，出现如图 4.114 所示的波形图。

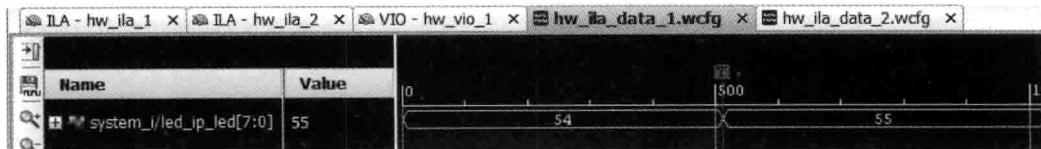


图 4.114 满足条件的触发波形

- (13) 观察完现象后，单击 ■ 按钮，停止程序的执行。

- (14) 在 SDK 主界面主菜单下,选择 File→Exit,退出 SDK。
- (15) 在 Vivado 主界面主菜单下,选择 File→Close Hardware Manager。
- (16) 单击 OK 按钮。
- (17) 在 Vivado 主界面主菜单下,选择 File→Exit。
- (18) 单击 OK 按钮。

# 第5章 Vivado HLS 设计流程

本章介绍了高级综合工具 HLS 的设计流程。内容包括：高级综合工具概述、高级综合工具调度和绑定、Vivado HLS 工具的优势、C 代码的关键属性、时钟测量术语说明、HLS 关键优化策略、基于 HLS 的数字系统实现。

Vivado HLS 工具的出现和发展，突破了以往在使用 FPGA 时，使用 HDL 语言进行设计实现的瓶颈。通过 C/C++/System C 对信号或数字处理直接建模，然后通过 Vivado HLS 工具将 C/C++/System C 所建立的模型直接转换为 RTL 级的模型。这样，大大提高了 FPGA 设计的效率，加速了 FPGA 在高性能信号和数据处理领域的应用和推广。

读者通过本章内容的学习，可以很清楚地建立“软件”和“硬件逻辑”之间的对应关系。

## 5.1 高级综合工具概述

高级综合工具概述部分，包括：高级综合工具的功能和特点、不同的命令对 HLS 综合结构的影响，以及从 C 中模型提取硬件结构。

### 5.1.1 高级综合工具的功能和特点

高级综合(High Level Synthesis, HLS)是 Xilinx 公司推出的最新一代的 FPGA 设计工具。图 5.1 给出了 HLS 的设计流程图。该综合工具的主要功能包括：

- (1) 从 C 语言描述级的源代码中创建一个 RTL 级实现；
- (2) 从 C 源代码提取出控制和数据流；
- (3) 基于默认的和用户定义的命令，实现设计；
- (4) 从相同的源代码描述中，可以实现很多的设计，即：
  - ① 实现较小的设计，较快速的设计，优化的设计；
  - ② 可以进行设计方法的“探索”，从而找到最佳的解决方案。

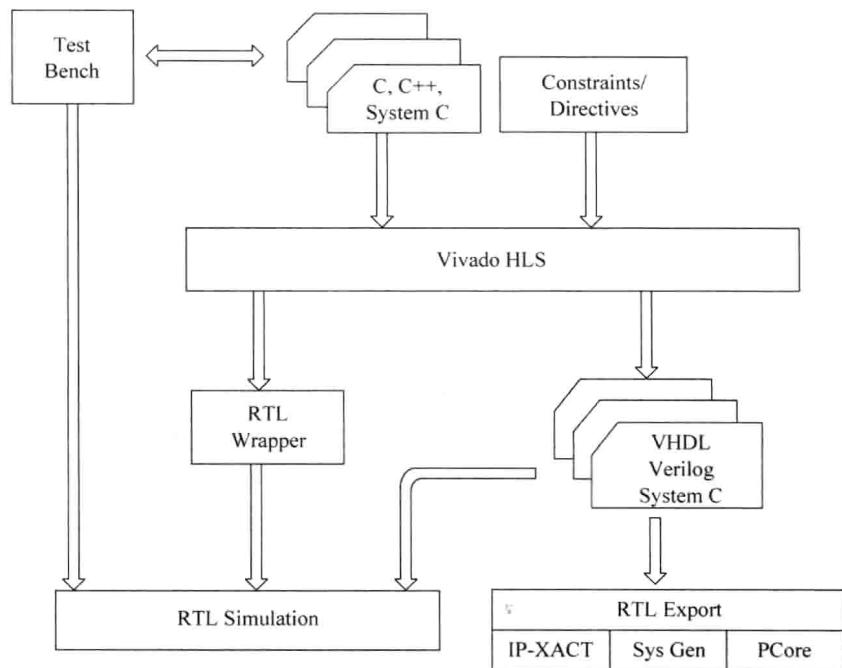


图 5.1 HLS 的流程图

## 5.1.2 不同的命令对 HLS 综合结果的影响

下面给出了一段 for 循环语句代码。

代码清单 5-1 for 循环

```

...
loop: for (i = 3;i >= 0;i --) {
    if (i == 0) {
        acc += x * c[0];
        shift_reg[0] = x;
    } else {
        shift_reg[i] = shift_reg[i - 1];
        acc += shift_reg[i] * c[i];
    }
}
...
  
```

情况 1：如图 5.2 所示，对于每个迭代，使用相同的硬件。在这种情况下：

- (1) 使用较少的区域(逻辑资源)；
- (2) 产生较长的延迟；
- (3) 获得较低的吞吐量；

情况 2：如图 5.3 所示，对于每个迭代，使用不同的硬件。在这种情况下：

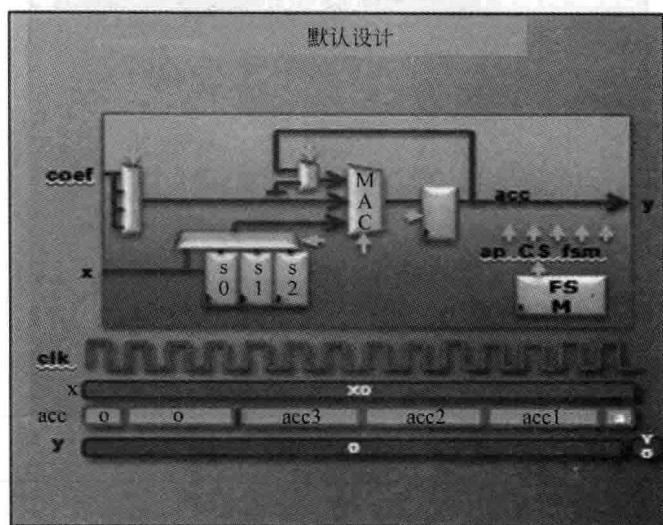


图 5.2 情况 1 下默认设计的硬件结构

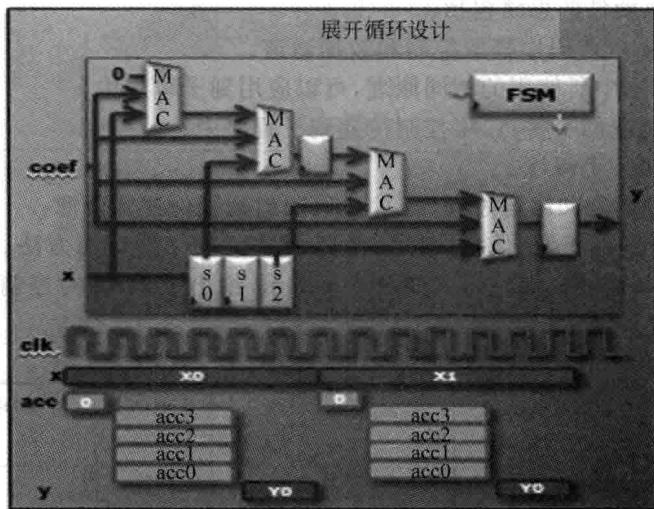


图 5.3 情况 2 下的设计结构

- (1) 占用较大的面积(逻辑资源);
- (2) 导致较短的延迟;
- (3) 产生更好的吞吐量。

情况 3: 如图 5.4 所示,不同的迭代并发运行。在这种情况下:

- (1) 占用较大的面积(逻辑资源);
- (2) 导致较短的延迟;
- (3) 产生最好的吞吐量。

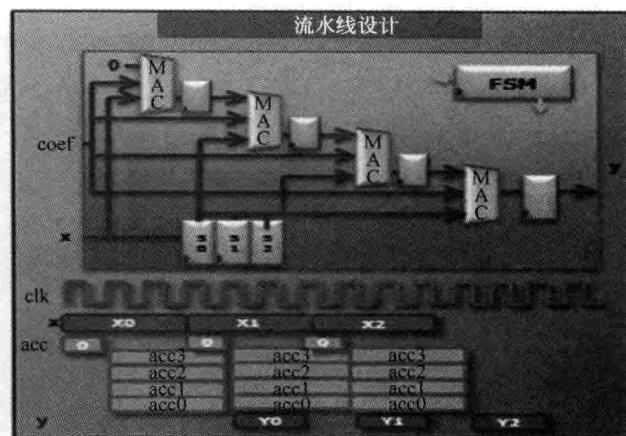


图 5.4 情况 3 下的设计结构

### 5.1.3 从 C 模型中提取硬件结构

从 C 代码提取硬件的思路包括：

- (1) 在顶层，从 C 代码中提取控制和数据通道。
- (2) 下面的例子中所使用的相同原理，可以应用到子程序中。
  - ① 在顶层控制流的一些点，将控制传递到子程序中；
  - ② 能并行地执行子程序。
- (3) HLS 通过调度和绑定过程，将 C 代码映射到硬件逻辑资源。

下面将通过一些例子说明映射函数、循环、数组和 IO 端口的方法。如图 5.5 所示，给出了 HLS 对控制流的提取过程。如图 5.6 所示，给出了 HLS 对控制和数据通路的提取过程。

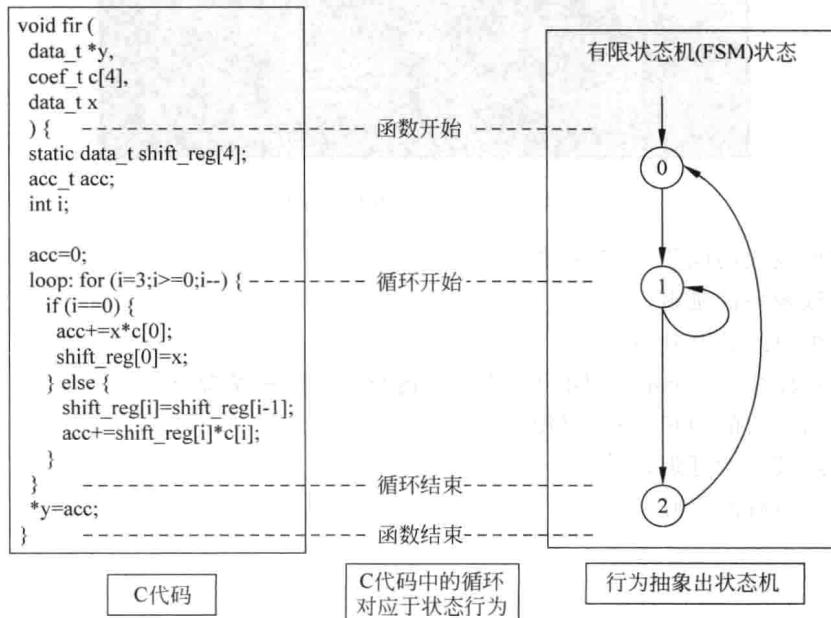


图 5.5 HLS 控制提取

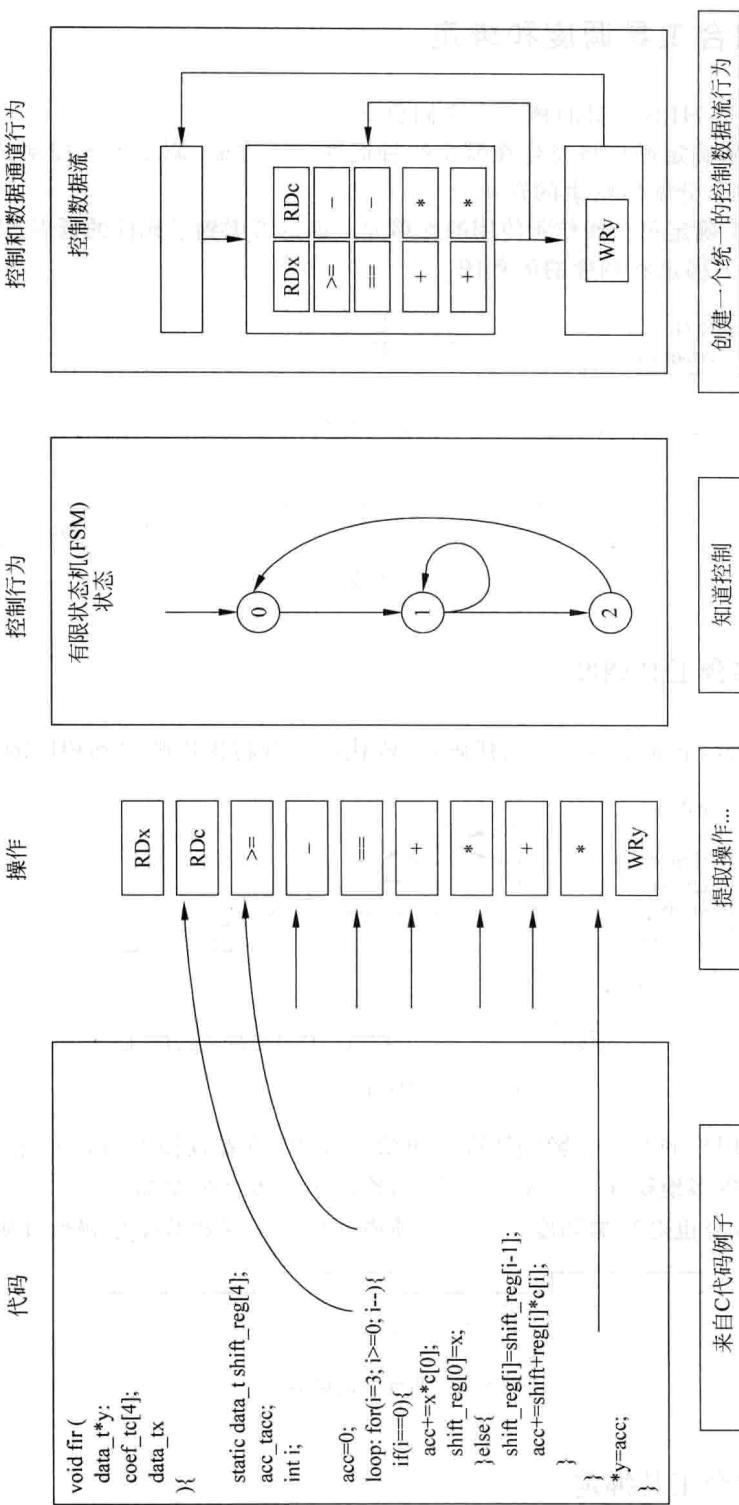


图 5.6 HLS 提取控制和数据流

## 5.2 高级综合工具调度和绑定

调度和绑定是 HLS 工具的核心。它们负责：

(1) 调度用来确定操作将发生在哪个时钟周期。一方面，调度考虑控制、数据流和用户指令；另一方面，分配所约束的资源。

(2) 绑定用于确定每个操作所使用的库单元。绑定考虑到了元件的延迟，用户的命令。

图 5.7 给出了绑定和调度的流程图。

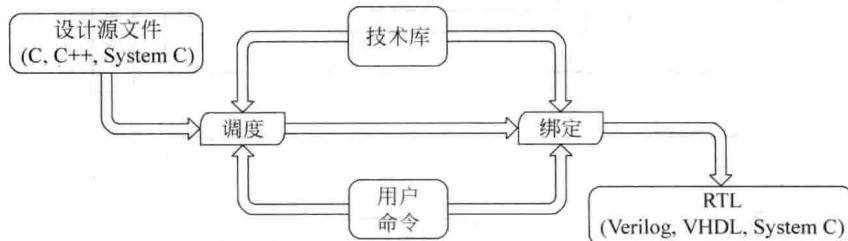


图 5.7 绑定和调度的流程图

### 5.2.1 高级综合工具调度

如图 5.8 所示，下面给出的一段代码，该段代码实现将操作映射到相应的时钟周期。

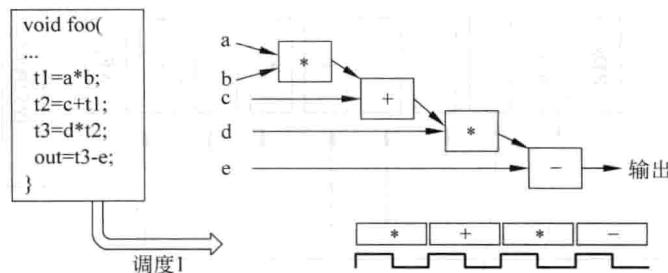


图 5.8 调度 1 的策略

技术方面和用户的约束会影响调度。更快的技术（或者较慢的时钟），允许在一个相同的时钟内实现更多地操作。如图 5.9 所示，给出了调度 2 的策略。

此外，代码本身也会影响调度。因此必须遵守代码含义和数据依赖性规则。

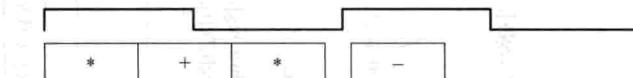


图 5.9 调度 2 的策略

### 5.2.2 高级综合工具绑定

绑定是指将操作映射到来自硬件库的核（操作符号映射到核）。下面给出绑定的两

种分割方式：

(1) 绑定决策：去共享。

图 5.10 给出了共享的绑定分割方法。

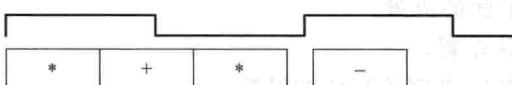


图 5.10 绑定分割方法(1)

① 上面的绑定过程需要两个乘法器，这是由于在一个周期内进行了两个乘法运算；

② 能确定使用一个加法器、减法器或者一个加/减法器。

(2) 绑定决策：或者非共享。

图 5.11 给出了非共享的绑定分割方法。



图 5.11 绑定分割方法(2)

① 由于在不同的周期使用乘法器，所以绑定可能决定共享乘法器；

② 或者它决定共享(复用)的代价。由于共享可能会影响时序，所以决定不共享乘法器；

③ 在上面的第一个例子中也可能作出相同的决策。

### 5.3 Vivado HLS 工具的优势

如图 5.12 所示，给出了对于一个图像处理的性能指标。从图中可以看出，HLS 在 FPGA 设计方面的巨大优势。

Video 设计例子			
输入	C 仿真时间	RTL 仿真时间	改善
10帧 1280×720	10s	~2天 (ModelSim)	1200倍

图 5.12 视频设计的例子

在验证方面，HLS 可以实现功能级和结构级的验证。在提取方面，HLS 可以提取出数据类型、接口和类。

HLS 的可移植性主要体现在：

(1) 处理器和 FPGA；

(2) 技术的转移；

(3) 减少开销；

(4) 降低功耗。

此外，HLS 工具可以实现设计重用和 IP 重用。

Vivado HLS 工具用于：

- (1) 决定在哪个周期进行操作(调度)；
- (2) 决定每个操作所使用的逻辑硬件单元(绑定)。

Vivado HLS 执行下面的决策：

- (1) 遵守内建的默认设置；
- (2) 使用用户命令和约束覆盖默认的设置；
- (3) 使用指定的技术/器件,计算延迟和面积。

Vivado HLS 命令的优先级体现在：

- (1) 首先,满足性能(时钟和吞吐量)：
  - ① 如果要求满足吞吐量,HLS 允许一个本地时钟路径的失败；
  - ② 当逻辑综合后,能满足常用的可能时序。
- (2) 然后,降低延迟。
- (3) 最后,减少面积。

## 5.4 C 代码的关键属性

如图 5.13 所示,给出一段 C 代码。通过这段代码,分析 C 代码的关键属性:

```
void fir (
    data_t * y,
    coef_t c[4],
    data_t x
) {
    static data_t shift_reg[4];
    acc_t acc;
    int i;
    acc = 0;
    loop: for (i = 3; i >= 0; i--) {
        if (i == 0) {
            acc += x * c[0];
            shift_reg[0] = x;
        } else {
            shift_reg[i] = shift_reg[i - 1];
            acc += shift_reg[i] * c[i];
        }
        *y = acc;
    }
}
```

图 5.13 描述 FIR 滤波器 C 语言代码

(1) 函数：所有的代码由函数组成。函数用于表示设计的层次,这对于硬件逻辑也一样。

(2) 参数：顶层函数的参数,决定了硬件 RTL 接口的端口。

(3) 类型：所有变量都是有所定义的类型。不同的类型对面积和性能有不同的影响。

(4) 循环：典型地，函数包含循环。处理循环的不同方法，对面积和性能都有很大的影响。

(5) 数组：在 C 代码中，经常使用数组。它们影响设备的 IO 端口。并且，可能会变成性能的瓶颈。

(6) 操作符：在 C 代码中的操作符，可能要求共享。通过共享，控制面积或者指定的硬件实现，以满足性能的要求。

### 5.4.1 函数

在 HLS 中，每个函数都被翻译成一个 RTL 模块，用于表示 Verilog 模块或者 VHDL 实体。如图 5.14 所示，给出了函数翻译成 RTL 模块的描述。其中：

(1) 默认地，使用一个公共的模块(实例)实现每个函数。

(2) 可以通过内联函数，来解析它们之间的层次。对于较小的函数，自动地进行内联。图 5.15 给出了从 C 代码中提取操作的过程。

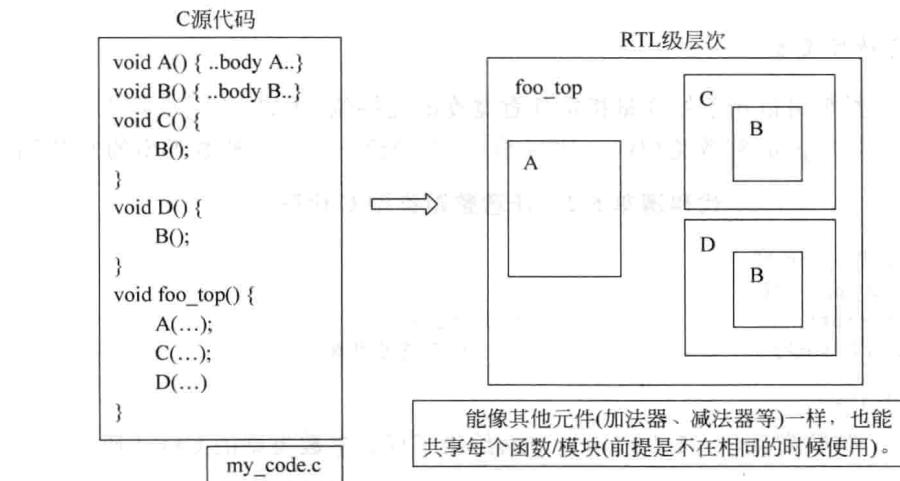


图 5.14 函数翻译成 RTL 模块的描述

### 5.4.2 类型

数据类型包括标准 C 整数类型、任意精度类型和浮点类型三种

#### 1. 标准的 C 整数类型

长整型(64 位)、短整型(16 位)、无符号类型、整型(32 位)、字符型(8 位)

注：对于单精度和双精度浮点，在库中必须绑定和映射到一个浮点处理单元。否则，HLS 不能对设计综合。

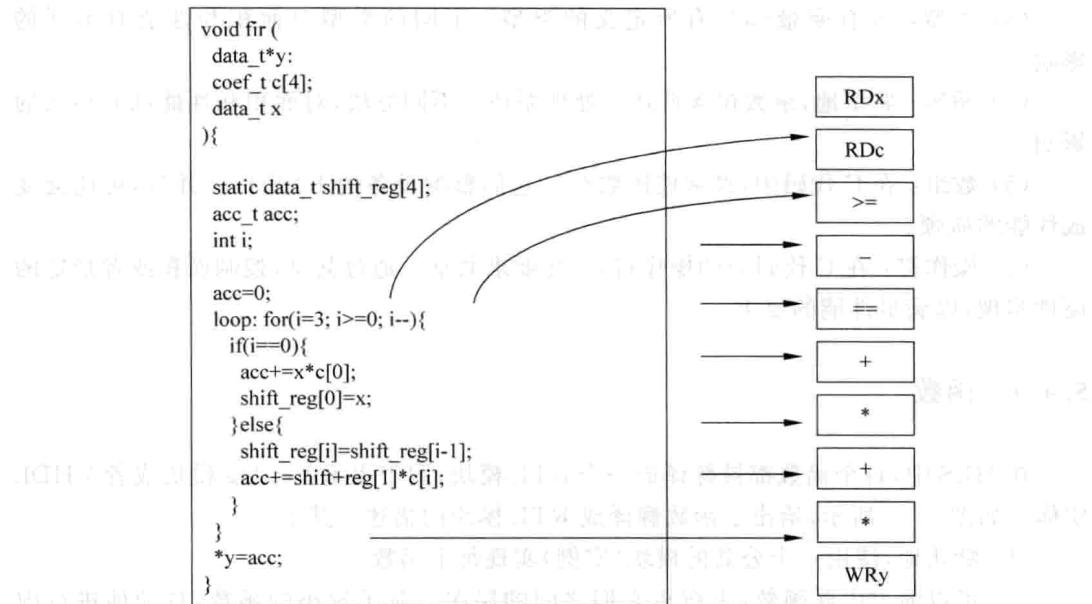


图 5.15 从 C 代码中提取操作的过程

## 2. 任意精度类型

任意精度类型可以用于给变量指定任意宽度的比特位(例如: 17 位、47 位等)。

(1) 对于 C: ap(u)整数类型(1~1024 位)。下面给出了任意整数类型的 C 代码。

### 代码清单 5-2 任意整数类型 C 代码

```

#include "ap_int.h"
void foo_top(...){
    int9 var1;           // 9 个整数位
    uint10 var2;          // 10 位无符号整数
  
```

(2) 对于 C++:

① ap\_(u)整数类型(1~1024 位)。下面给出了任意整数类型的 C++ 代码。

### 代码清单 5-3 任意整数类型 C++ 代码

```

#include "ap_int.h"
void foo_top(...){
    ap_int<9> var1;      // 9 个整数位
    ap_uint<10> var2;      // 10 位无符号整数
  
```

② ap\_fixed 类型。定义格式为: ap\_[u]fixed<W,I,Q,O,N>。表 5.1 给出了定点数的定义格式。下面给出了定点数类型的 C++ 代码。

**注:** 总能用于 C++, 而不能用于 C'。

### 代码清单 5-4 定点数类型的 C++ 代码

```

#include "ap_fixed.h"
void foo_top(...){
  
```

```
ap_fixed<9,5,AP_RND_CONV,AP_SAT> var1; //9位,5个整数位
ap_ufixed<10,7,AP_RND_CONV,AP_SAT> var2; //10位无符号,7个整数位
```

③ 对于 C++/System C:

sc\_(u)整数类型(1~1024 位);  
sc\_fixed 类型。

表 5.1 定点数的定义格式

	描    述	
W	总共的位宽	
I	整数位的个数	
Q	量化模式。确定当产生更大的精度范围时,所实现的行为	
	定点模式	描述
	AP_RND	四舍五入到正无穷(正的最大)
	AP_RND_ZERO	四舍五入到零。例如: (1) 正数 01.01(1.25)四舍五入到 01.0(1); (2) 负数 10.11(-1.25)四舍五入到 11.0(-1)
	AP_RND_NIN_INF	四舍五入到负无穷(负的最小)
	AP_RND_INF	四舍五入到无穷
	AP_RND_CONV	四舍五入到最近的值,取决于 LSB。 (1) 如果设置了 LSB,四舍五入到正无穷; (2) 否则,四舍五入到负无穷。 例如: ① 正数 00.11(0.75),四舍五入到 01.0(1.0); ② 负数 10.11(-1.25),四舍五入到 11.0(-1)
	AP_TRN	截断到负无穷,去除冗余位。例如: 正数 01.01(1.25),四舍五入到 01.0(1)
	AP_TRN_ZERO	截断到零(默认) ① 对于正数,和 AP_TRN 一样。比如: 01.01(1.25)四舍五入到 01.0(1)。 ② 对于负数,四舍五入到零。 10.11(-1.25)四舍五入到 11.0(-1)
O	溢出模式,当要求更多的位时,所实现的行为	
	定点模式	描述
	AP_SAT	饱和。 ① 对于上溢,将指定的值转换到 MAX; ② 对于下溢,将指定的值转换到 MIN。 注:由可用的位确定 MAX 和 MIN
	AP_SAT_ZERO	饱和到零。当结果超过范围的时候,将结果设置为零
	AP_SAT_SYM	对称饱和。在二进制的补码表示中,负数比正数要多。如果希望 MIN 和 MAX 的绝对值在零周围对称,则使用这种方式。正的向上溢出将产生 MAX,负的溢出将产生 MIN。例如: ① 0110(6)四舍五入到 011(3); ② 1011(-5)四舍五入到 101(-3)
	AP_WRAP	回卷的四舍五入

续表

	描    述
AP_WRAP_SM	<p>符号幅度回卷四舍五入。</p> <p>(1) <math>N=0</math>:</p> <ul style="list-style-type: none"> <li>① 这个模式使用符号幅度回卷;</li> <li>② 符号位设置为被删除的最低有效位;</li> <li>③ 如果最高保留有效位不同于最初的 MSB, 将所有的其他位取反。否则, 复制过来。</li> </ul> <p>例如: 0100(4):</p> <ul style="list-style-type: none"> <li>① 删除冗余的 MSB, 结果为 100(-4);</li> <li>② 新的符号位是所删除的最低有效位 0;</li> <li>③ 最高有效位和最初的不一样, 所以取反(011)。</li> </ul> <p>(2) <math>N&gt;0</math>:</p> <ul style="list-style-type: none"> <li>① 使用符号幅度饱和;</li> <li>② 这里的 MSB 将饱和到 1;</li> <li>③ 除了正数和负数保留外, 行为类似于 <math>N=0</math> 的情况</li> </ul>
N	在回卷模式下的饱和位数

### 3. 浮点类型

尽管 FPGA 实现定点运算比浮点运算要快, 且面积更高效。但是, 往往有时也需要浮点来实现。这是因为定点数有限的数据动态范围, 需要深入地分析才能决定整个设计中间数据位宽变化的模式, 为了达到优化的 QoR, 需要引入很多不同类型的定点中间变量。而浮点数具有更大的数据动态范围, 从而在很多算法中具有只需要一种数据类型的优势。

Xilinx Vivado HLS 工具支持 C/C++ IEEE-54 标准单精度及双精度浮点数据类型, 可以比较容易、快速地将 C/C++ 浮点算法转成 RTL 代码。与此同时, 为了达到设计者所期望的 FPGA 资源与性能, 当使用 Vivado HLS 命令时, 需要注意 C/C++ 编码风格与技巧相结合。

#### 1) 单双精度浮点数学函数

对于下面的代码:

#### 代码清单 5-5 对数源点运算代码

```
# include <math.h>
float example(float var)
{
    return log(var);
}
```

对于这个例子来说, 在 C 设计中, Vivado HLS 生成的 RTL 实现将输入转换成双精度浮点。基于双精度浮点计算自然对数, 并且将双精度浮点输出转换成单精度浮点。

而对于下面的代码:

### 代码清单 5-6 单精度自然对数代码

```
#include <math.h>
float example(float var)
{
    return logf(var);
}
```

在 C 设计中,logf 才是单精度自然对数。在这个例子中,Vivado HLS 生成的 RTL 实现将基于单精度浮点计算自然对数,并且没有输入输出单双精度的互相转换。

#### 2) 浮点运算优化

对于下面给出的代码,从代数上看三个的描述方法差不多。但是,在 Vivado HLS 中综合出来的三个结果截然不同。

### 代码清单 5-7 浮点运算(1)

```
void example(float *m0, float *m1, float *m2, float var)
{
    *m0 = 0.2 * var;           // 双精度浮点乘法,单双精度类型转换
    *m1 = 0.2f * var;          // 单精度浮点乘法
    *m2 = var / 5.0f;          // 单精度浮点除法
}
```

Vivado HLS 将 m0、m1、m2 综合成不同的 RTL 实现。由于 0.2 是一个不能精确表示的双精度数字,所以 Vivado HLS 将 m0 运算综合成一个双精度浮点乘法。并且,将 var 转换成双精度,然后将双精度乘法输出 m0 转换成单精度。

**注:** 如果希望 Vivado HLS 综合出单精度常数,需要在常数后面加 f,如 0.2f。这样,将 m1 综合成一个单精度乘法的输出。同理,m2 将被 Vivado HLS 综合成单精度除法的输出。

对于下面给出的代码:

### 代码清单 5-8 浮点运算描述(2)

```
void example(float *m0, float *m1, float var)
{
    *m0 = 0.2f * 5.0f * var;      // *m0 = var;优化常数乘法
    *m1 = 0.2f * var * 5.0f;      // 两个双精度浮点乘法
}
```

例如下面给出的代码:

### 代码清单 5-9 浮点运算描述(3)

```
void example(float *m0, float *m1, float var)
{
    *m0 = 0.5 * var;
    *m1 = var/2;                // var/2 可以得到半精度,单精度除法
}
```

Vivado HLS 将 m0 运算综合成一个双精度浮点乘法,并且将 var 转换成双精度,然

后将双精度乘法输出 m0 转换成单精度。

Vivado HLS 将 m1 运算综合成简单的右移运算。所以如果用户希望实现对 var 除以 2, 就写成 m1 这种表达式, 而不是写成 m0 开线的表达式。

### 3) 并行度与资源复用

与整数和定点运算相比, 浮点运算消耗更多的资源。因此, Vivado HLS 会尽量用更有效的资源来实现浮点运算。当在数据相关性及约束许可的情况下, Vivado HLS 会尽量复用一些浮点运算单元。为了说明这个方法, 下面给出简单的三个浮点加法例子, Vivado HLS 复用一个浮点加法器来串行实现三个浮点加法。代码及 HLS 综合结果如下:

**代码清单 5-10 浮点运算描述(4)**

```
void example(float * r, float a, float b, float c, float d)
{
    *r = a + b + c + d;    /* 需要复用一个单精度加法器 */
}
```

如果希望并行三个浮点加法来实现, 可以加上 pipeline 命令, 由于浮点运算的精度与运算顺序有很大的关系, 因此 HLS 工具不会改变用户代码的计算顺序, 这样只是个级联结构。代码及 HLS 综合结果如下, 延迟是 35。

**代码清单 5-11 浮点运算描述(5)**

```
void example(float * r, float a, float b, float c, float d)
{
    #pragma HLS PIPELINE
    *r = a + b + c + d;
}
```

如果希望并行三个浮点加法来实现, 同时降低延迟, 可以在加上 pipeline 命令的同时将代码简单修改成加法树结构。这样的代码及 HLS 综合结果如下, 延迟从 35 降低到 23。

**代码清单 5-12 浮点运算描述(6)**

```
void example (float * r, float a, float b, float c, float d)
{
    #pragma HLS PIPELINE
    float e, f;
    e = a + b;
    f = c + d;
    *r = e + f;
}
```

有时设计需要更高的吞吐量及更低的延迟。这时, 就需要提高设计的并行度。以下面例子来说明, Vivado HLS 就需要对 for 循环加 pipeline 与 unroll 的命令。同时需要通过设置 a、b、r0 为 FIFO 结构, 并对其重排以提高两倍的 I/O 带宽。这样, Vivado HLS 就会综合出两个浮点加法单元来并行实现算法, 这是因为每个加法器中的计算完全独立。

### 代码清单 5-13 浮点运算描述(7)

```
void example(float r0[32], float a[32], float b[32])
{
    #pragma HLS interface ap_fifo port = a,b,r0
    #pragma HLS array_reshape cyclic factor = 2 variable = a,b,r0
    for (int i = 0; i < 32; i++)
    {
        #pragma HLS pipeline
        #pragma HLS unroll factor = 2
        r0[i] = a[i] + b[i];
    }
}
```

然而,如果有更复杂的运算,可能会导致浮点运算失去独立性。在这种情况下,Vivado HLS不能重新排列这些运算的顺序。这样,会导致更低的、非所期望的复用。下面举例来说明如何提高带有反馈浮点运算的性能。

这个例子的累加会导致反复,并且通常浮点加法的延迟大于一个时钟周期,而所添加的 pipeline 命令并不能达到一个时钟周期,完成一次累加的 throughput。

### 代码清单 5-14 浮点运算描述(8)

```
float example(float x[32])
{
    #pragma HLS interface ap_fifo port = x
    float acc = 0;
    for (int i = 0; i < 32; i++)
    {
        #pragma HLS pipeline
        acc += x[i];
    }
    return acc;
}
```

为了对上面例子并行展开,可以对代码做如下较小的改动,也就是拆分为先部分累加,再最后累加。当然也需要对输入数据进行简单地重新排列,以获得相应的 I/O 带宽,从而达到所期望的并行度。

### 代码清单 5-15 浮点运算描述(9)

```
float top(float x[32])
{
    #pragma HLS interface ap_fifo port = x
    float acc_part[4] = {0.0f, 0.0f, 0.0f, 0.0f};

    for (int i = 0; i < 32; i += 4) { // 手动 unroll by 4
        for (int j = 0; j < 4; j++) { // 部分累加
            #pragma HLS pipeline
            acc_part[j] += x[i + j];
        }
        for (int i = 1; i < 4; i++) { // 最后累加
    }
```

```
# pragma HLS unroll
acc_part[0] += acc_part[i];
}
return acc_part[0];
}
```

### 5.4.3 循环

如图 5.16 所示,给出了一段运算 C 代码。主要表现在以下:

- (1) 展开循环会导致使用更多的元件。但是,有更大的操作符机动性。
- (2) 从图中可以看出,可以从开始到迭代 4 的任何地方,对数据端口 X 进行读操作。对于 RDx 的唯一要求是,它发生在最后的乘法操作之前。

- Vivado HLS 为这个操作提供了很多的灵活性,表现在:
- (1) 它一直等待直到要求读操作为止,节省了一个寄存器。
  - (2) 任何较早的读操作,都没有优势(除非想让它寄存)。
  - (3) 可以选择将读输入进行寄存。

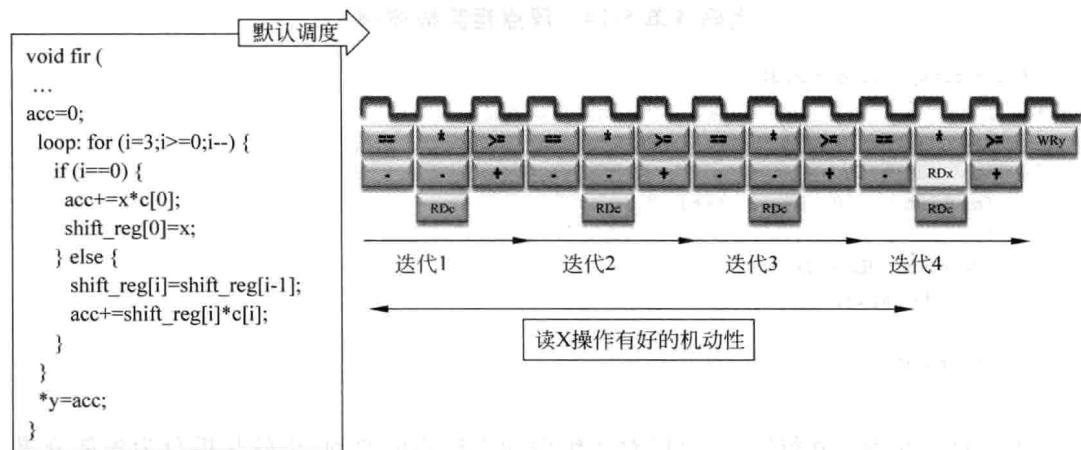


图 5.16 数据依赖性: 好的读操作机动性

如图 5.17 所示,对于最后的乘法操作,有很多约束,这样,导致其机动性较差。体现在:

- (1) 在读和最后的加法运算前,必须产生最后的乘法操作。如果时序允许的话,可以在相同的周期内产生。
- (2) 默认地,不展开循环。主要表现在:
  - ① 不能开始一个迭代,直到完成前面的迭代为止;
  - ② 最后的乘法操作(在迭代 4)必须等待完成前面的迭代。
- (3) 代码的结构迫使产生一个特殊的调度。
- 对于大多数的操作,几乎没有什幺机动性。
- (4) 允许展开循环的优化,提供了更大的自由度。

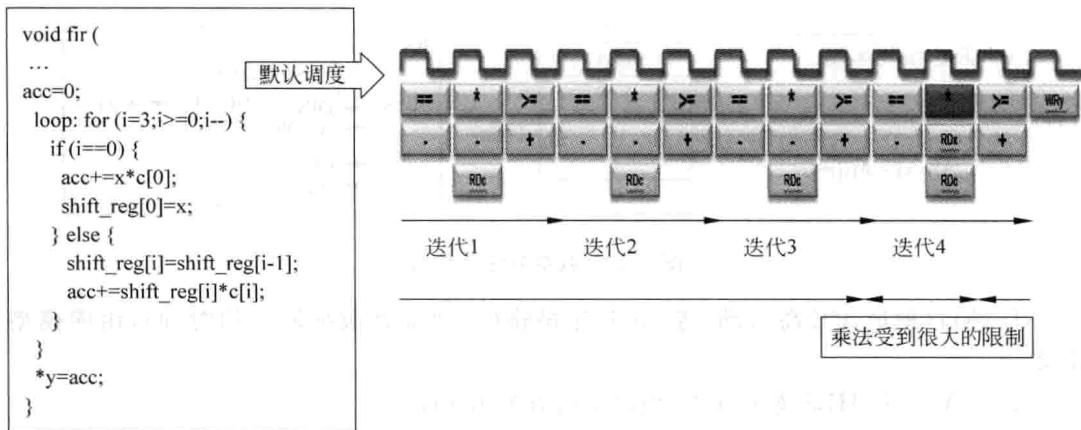


图 5.17 数据依赖性：差

如图 5.18 所示,说明在对循环优化之后的调度情况。主要体现在:

- (1) 消除了循环迭代之间的依赖性。
- (2) 可以并行地产生操作。
  - ① 如果数据依赖性允许;
  - ② 如果操作符时序允许。
- (3) 更快地完成设计,但是使用了更多的操作符。即: 2 个乘法器和 2 个加法器。

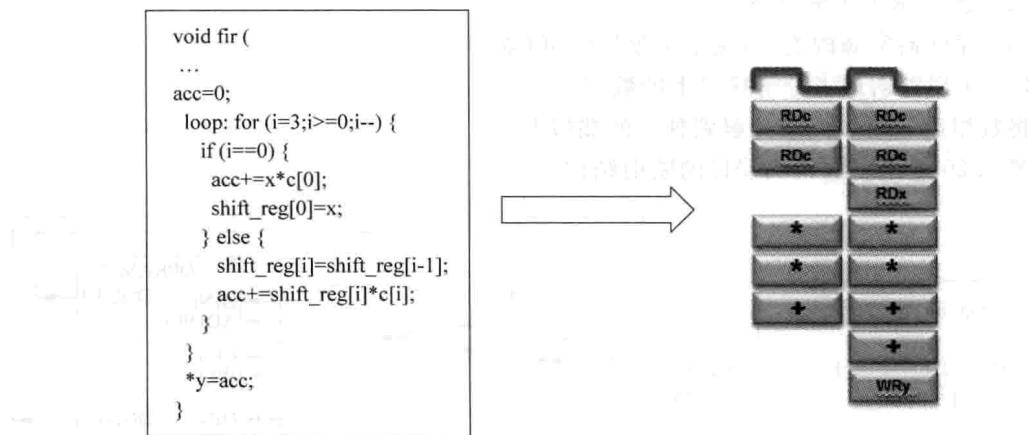


图 5.18 对循环优化之后的调度情况

#### 5.4.4 数组

如图 5.19 所示,给出了通过在 RTL 内的存储器实现 C 代码中的数组。主要体现在以下几个方面:

- (1) 默认地,数组用 RAM 实现(可选 FIFO)。
- (2) 实现数组的目标可以是库里的任何存储器资源:

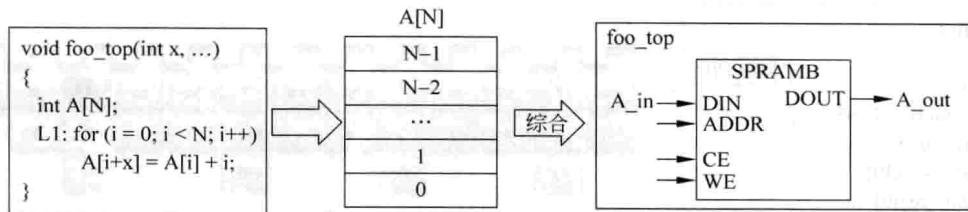


图 5.19 数组的综合结构

- ① 端口(地址,CE 高活动,等)和时序操作(从地址到数据输出的时钟),由库模型定义。
- ② 在 Vivado HLS 库向导中,列出了所有的 RAM。
- (3) 数组可以和其他数组合并。并且,可以进行重新配置。可以使用相同或者不同宽度和大小的存储器实现它们。
- (4) 数组能分解单个元素中,它们使用更小的 RAM 或者寄存器实现。

#### 5.4.5 端口

所有顶层函数的参数都有一个默认的硬件端口类型。当数组是顶层函数的一个参数时:

- (1) 数组/RAM 是“片外”;
- (2) 存储器资源的类型,决定了顶层的 IO 端口;
- (3) 可以映射或者分解接口上的数组。

将数组内的每个元素分解到独立的端口上。

图 5.20 给出了数组到端口的映射结构。

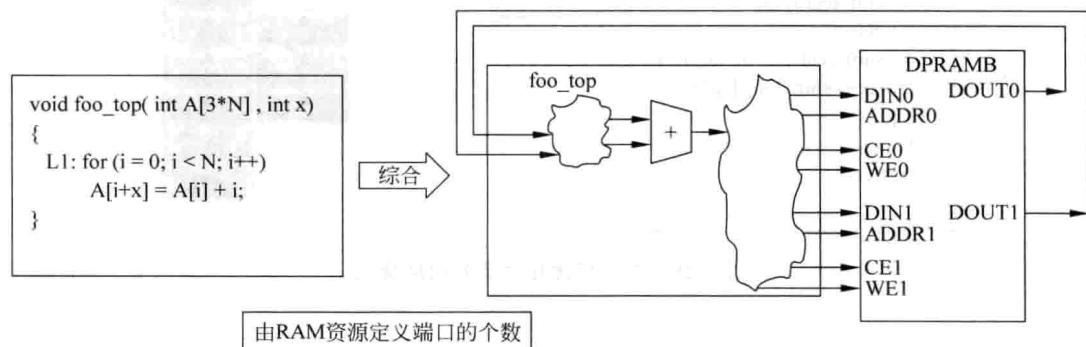


图 5.20 数组到端口的映射结构

使用双端口 RAM 提高性能。否则,使用单端口。

如图 5.21 所示,给出优化完数组后的调度。其带有已存在代码和默认设置,即:

- ① 端口 C 是双端口 RAM;
- ② 在每个时钟周期,允许 2 个读操作。

从该例子中可以看出,行为影响性能。

注:在最初的设计中执行两个读。但是,由于为了展开循环,迫使每个周期都存在一个读操作,所以这个结构并没有优势。

当C端口分解为4个独立的端口后:

- (1)可以在一个周期内,产生所有的读和乘操作。
- (2)如果时序允许,则
  - ①可以在相同的周期内,产生加法操作;
  - ②可以在相同的周期内,进行写操作;
  - ③可以选择对端口读和写操作进行寄存。

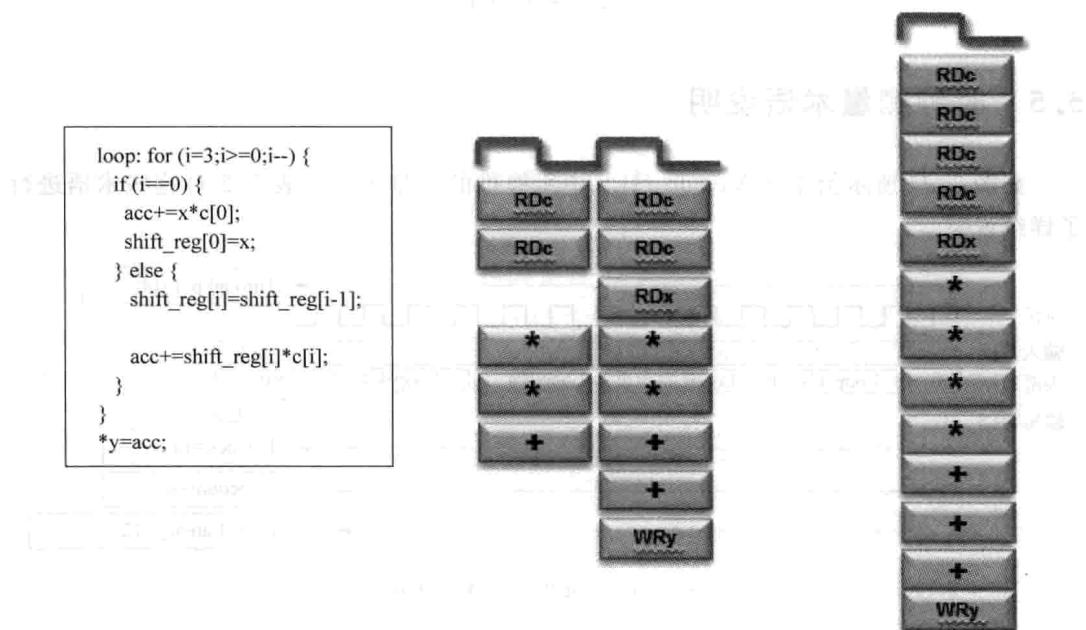


图 5.21 当优化完数组后的调度

#### 5.4.6 操作符

由类型定义操作符的大小。Vivado HLS 工具尽量将操作符的个数降到最低。默认地,在满足约束条件后,HLS 寻求最小化面积,即使用最少的逻辑资源。用户可以为所使用的资源设置指定的约束和目标,例如:

(1) 对分配进行控制:能对设计中操作符的个数设置更高的限制。该限制将强迫共享。如图 5.22 所示,将乘法器的个数限制为 1,将使得 HLS 共享乘法器资源。

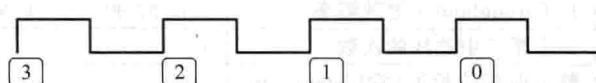


图 5.22 操作时序(1)

如图 5.22 所示,该资源使用 1 个乘法器实现,但是消耗了 4 个周期。即使可以使用 4 个乘法器在 1 个周期内实现乘法运算,但是由于乘法器个数的限制,也无法实现。

(2) 指定资源:可以指定用于实现每一个操作符的核。如图 5.23 所示,使用 2 级的流水线核(硬件)实现每个乘法器。通过使用 2 个流水线的乘法器,就可以执行相同的 4 个乘法操作(分配限制乘法器为 2)。

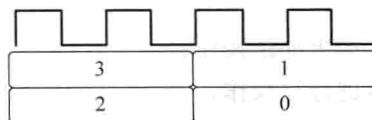


图 5.23 操作时序(2)

## 5.5 时钟测量术语说明

如图 5.24 所示给出了 Vivado HLS 中所提供的测量术语。表 5.2 对这些术语进行了详细说明。

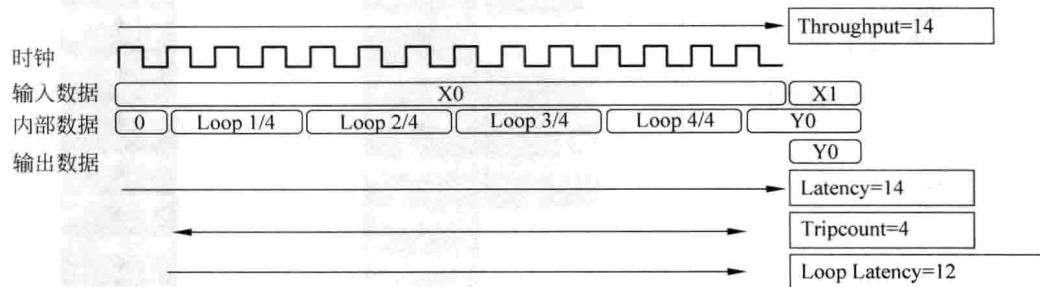


图 5.24 测量时序的术语表示

表 5.2 术语说明

术 语	描 述	数 量
Latency	从输入到输出的周期个数(一个数组最终的写输出)	14 个周期
Throughput	在一个新的采样之间的周期个数(这个例子中,在读一个新的输入前,它必须等待所有的操作完成)	14 个周期
Initiation Interval(II)	在新的输入到一个流水线之间的时钟周期的个数(和吞吐量一样,但是这个术语带有流水线)	在这个例子中没有显示出来
Data Rate	$1/\text{Throughput} * \text{时钟频率}$	$10\text{ns 时钟} = >7.14\text{MHz}((1/10\text{e}9) * 14)$
Trip count	一个循环中迭代的次数	4
Loop Latency	整个循环的延迟(除以 tripcount 得到每个循环迭代的延迟)	12 个周期

## 5.6 HLS 关键优化策略

HLS 关键优化策略包括延迟和吞吐量、循环处理、数组处理、#PARGRAM。

### 5.6.1 延迟和吞吐量

#### 1. 延迟和吞吐量概念

(1) 设计延迟：从输入到产生输出结果的周期数。如图 5.25 所示，该延迟为 10 个周期。

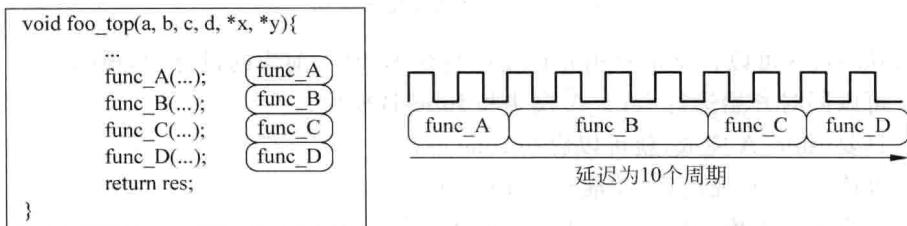


图 5.25 延迟表示

(2) 设计吞吐量：在两个新输入数据之间的周期数。如图 5.26 所示，该吞吐量为 10 个周期。并且：

- ① 默认地，在没有并发的情况下，吞吐量和延迟相同；
- ② 当该交易结束时，开始下一个交易；
- ③ 吞吐量和延迟之间存在一定的关系。

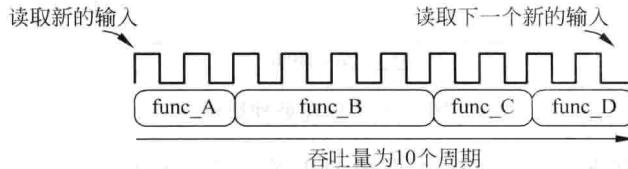


图 5.26 吞吐量表示(1)

Vivado HLS 中为函数和循环提供了流水线(pipeline)的功能，以提高吞吐量。如图 5.27 所示，经过流水线处理后，虽然延迟仍然是 10 个周期，但是吞吐量却减少到 4 个周期。

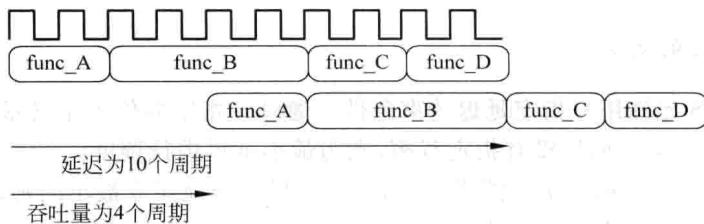


图 5.27 吞吐量表示(2)

默认地,Vivado HLS 将延迟降低到最小。在 Vivado HLS 中,吞吐量的优先级高于延迟。在 Vivado HLS 中并没有提供优化吞吐量的相关策略。如图 5.28 所示,在该设计描述中,根据数据之间的关联性,对模块进行了相关的连接。在该设计中,func\_B 模块所消耗的时间要多于其他模块所消耗的时间。

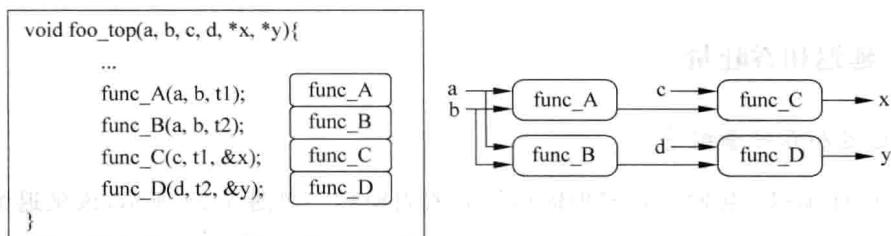


图 5.28 根据设计连接模块

Vivado HLS 可以自动地利用并行度进行处理,对于前面的设计中,很明显:

- (1) 可以同时开始运行 `func_A` 模块和 `func_B` 模块;
- (2) 只要 `func_A` 完成,就可以启动 `func_C`;
- (3) 等待 `func_B` 完成后,才能开始 `func_D`;
- (4) 当完成所有模块后,就可以产生输出结果。

如图 5.29 所示,将前面的结构进行更进一步改进,就可以降低设计延迟。很明显,延迟减少为 6 个周期。

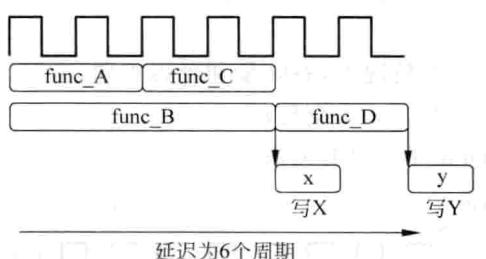


图 5.29 优化处理结构

在默认优化策略下,Vivado HLS 降低延迟的方法主要有:

- (1) 函数:通过并行运行函数,以减少操作。上面的例子说明了该实现过程。
- (2) 循环:默认地,Vivado HLS 并不会调度循环,使其并行执行。要想并行执行循环,则必须使用数据流(dataflow)优化或者循环展开(unroll)策略。
- (3) 操作:Vivado HLS 允许在函数和循环内并行执行操作,用来将延迟降低到最小。

## 2. 用户延迟的定义

Vivado HLS 允许用户指定延迟约束条件。通过在指定的位置定义最小和最大延迟来实现用户延迟约束。如果没有指定范围,则为最小延迟进行调度。

- (1) 如果指定的延迟小于所能实现的延迟,则尽可能地建立最小的延迟。
- (2) 如果指定的延迟大于所能实现的延迟,则指定的延迟将不起任何作用。

延迟指令可以用于函数、循环和区域。使用区域可以为指定的区域使用延迟约束条件。一个区域由{}括起来名字标识。例如：

```
int write_data(int buf, int putput){
    if(x < y){
        return (x + y);
    }else{
        My_Region:{
            return(y - x) * (y + x);
        }
    }
}
```

这样，就允许为指定区域添加延迟约束条件。在该例子中，允许为 else 条件添加用户延迟命令。

### 3. 数据流优化

通过数据流策略可以改善吞吐量。可以在顶层函数中使用数据流(Dataflow)优化。这样，允许并发操作代码块。这些代码块可以是函数或者循环。如图 5.30 所示，给出了数据流结构图。从图中可以看出，在模块间设置通道用于维持数据率。

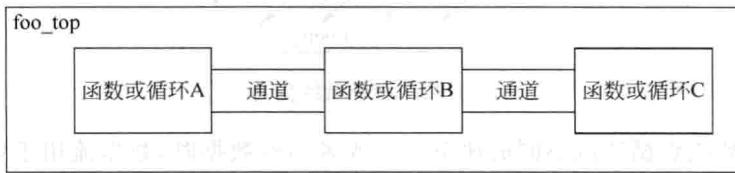


图 5.30 数据流结构

**注：**

- (1) 对于数组，通道包含存储器元素，它们用于对采样进行缓冲；
- (2) 对于标量，通过一个带有握手信号的寄存器用于对数据进行同步。

数据流优化是以开销面积(逻辑资源)为代价的，即：在设计中添加额外的存储器块。

下面给出配置数据流存储器的方法：

(1) 默认地，Vivado HLS 在函数之间使用“乒乓”存储器缓冲区。存储器的大小由生产者或者消费者元件确定。

(2) 在循环之间，Vivado HLS 将确定是否使用 FIFO 来代替乒乓缓冲区。

① 当默认是乒乓缓冲区时，选择一个数组，并且使用命令 STREAM 将其标记为 Streaming，将数组作为 FIFO。

② 当默认是 FIFO 时，选择一个数组，使用命令 STREAM 将其标记为 Streaming。选项设置为 off 时，将数组作为乒乓缓冲区。

(3) 使用数据流配置，可以将存储器指定为 FIFO。

① 菜单：Solution→Solution Settings→config\_dataflow。

② 设计者可以覆盖默认的 FIFO 大小值。如果 FIFO 的值太小，将导致 RTL 验证失败。

数据流非常适合于流数组和多速率处理的实现。默认地，将数组作为单个实体进行传递。图 5.31 给出了描述和时序。

```
int a[N], b[N], c[N];

Loop_1: for(i=0; i<=N-1; i++){
    b[i]=a[i]+in1;
}

Loop_1: for(i=0; i<=N-1; i++){
    c[i]=b[i]*in2;
}
```

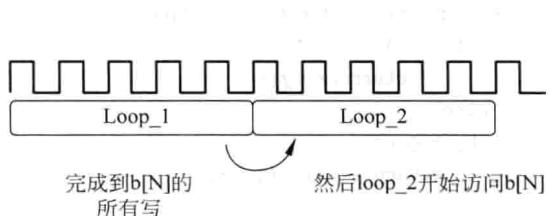


图 5.31 默认的数组处理方式

如图 5.32 所示，当数据准备好后，数据流流水线允许开始 loop2。这样，大大改善了系统的吞吐量。如果允许数据依赖性的话，将在并行方式下运行循环。

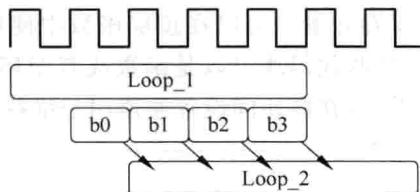


图 5.32 吞吐量的提高

当一个函数或者循环以不同的速率产生或者消费数据时，数据流用于缓冲数据。在流设计中为了充分利用数据流，数据通道的两端的 IO 接口都应该是 stream/handshake 类型(ap\_hs/ap\_fifo)。

#### 4. 流水线优化

数据流优化用于顶层粗粒度的流水。而在操作符级上是细粒度的流水处理。这样，允许并行执行函数或者循环内的操作。下面通过函数流水线和循环流水线，说明它们对吞吐量的改善。

##### 1) 函数流水线

对于下面的代码：

```
void foo( ... ){
    op_Read;           // RD
    op_Compute;        // CMP
    op_Write;          // WR
}
```

如图 5.33(a)所示，如果没有使用流水线，在发生下一个 RD(读)操作时，吞吐量为 3 个周期。当写第一个输出结果时，延迟 3 个周期。如图 5.33(b)所示，当使用流水线时，延迟还是 3 个周期，但是吞吐量明显提高，即减为一个周期。

##### 2) 循环流水线

对于下面的代码：

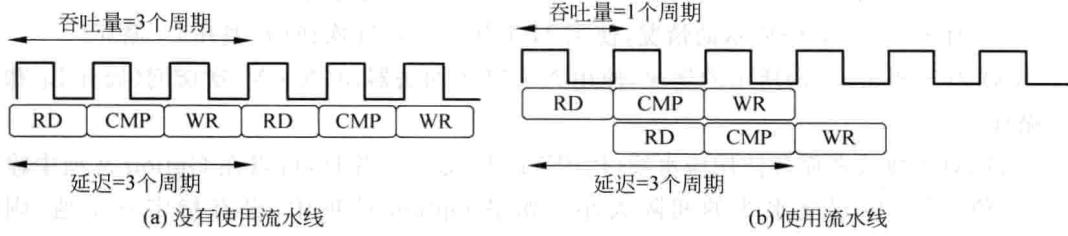


图 5.33 函数的流水处理

```
Loop: for(i = 1;i<3;i++){
    op_Read; //RD
    op_Compute; //CMP
    op_Write; //WR
}
```

如图 5.34(a)所示,对于每个循环来说,在开始下一个 RD 前,需要三个周期时钟。所以,吞吐量是 3 个周期。在写一个结果前,需要 3 个周期。因此,延迟是 3 个周期。对于这个循环来说,一共是 6 个周期。如图 5.34(b)所示,对于每一个循环,延迟相同。但是,整个循环的延迟减少到 4 个周期。

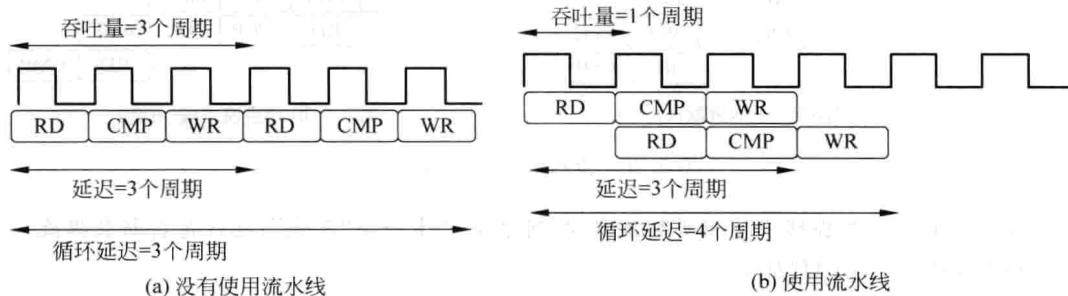


图 5.34 循环的流水处理

当在多重循环的不同位置使用流水线策略时,也会产生不同的优化结果。图 5.35 给出不同结果的描述。

<pre>void foo( in1[ ][ ], in2[ ][ ], ... ) {     ...     L1:for(i = 1;i&lt;N;i++) {         L2:for(j = 0;j&lt;M;j++) {             # pragma AP PIPELINE             out[i][j] = in1[i][j] + in2[i][j];         }     } }</pre>	<pre>void foo( in1[ ][ ], in2[ ][ ], ... ) {     ...     L1:for(i = 1;i&lt;N;i++) {         # pragma AP PIPELINE         L2:for(j = 0;j&lt;M;j++) {             out[i][j] = in1[i][j] +             in2[i][j];         }     } }</pre>	<pre>void foo( in1[ ][ ], in2[ ][ ], ... ) {     ...     # pragma AP PIPELINE     L1:for(i = 1;i&lt;N;i++) {         L2:for(j = 0;j&lt;M;j++) {             out[i][j] = in1[i][j] +             in2[i][j];         }     } }</pre>
--	--	--

(a) 最内侧循环流水

(b) 中间循环流水

(c) 最外循环流水

图 5.35 循环的不同流水处理

- (1) 对于图 5.35(a)所示的情况, 使用 1 个加法器, 3 次访问;
- (2) 对于图 5.35(b)所示的情况, 使用  $M$  个加法器,  $3M$  次访问(展开  $L_2$  循环);
- (3) 对于图 5.35(c)所示的情况, 使用  $N * M$  个加法器,  $3(N * M)$  次访问(展开  $L_1$  和  $L_2$  循环)。

可以对函数或者循环使用流水线(PIPELINE)命令。并且,可以在 Option 选项中输入一个值,用于控制流水线的间隔大小。如果 Option 选项中,没有指定一个值,则 Vivado HLS 将自动实现流水线,以便实现最快的设计。如果 Option 的值越大,则允许更多的资源共享,即:所要求较少的逻辑设计资源实现设计。

此外,设计者可以选择清空流水线。如图 5.36(b)所示,如果选择清空流水线时,输入使能(Data Valid)为低(没有更多的数据),将清空所有已经存在的结果。输入使能信号可以来自一个输入接口或者来自设计中的其他模块。如图 5.36(a)所示,默认地,停止流水线内的所有已经存在的值。

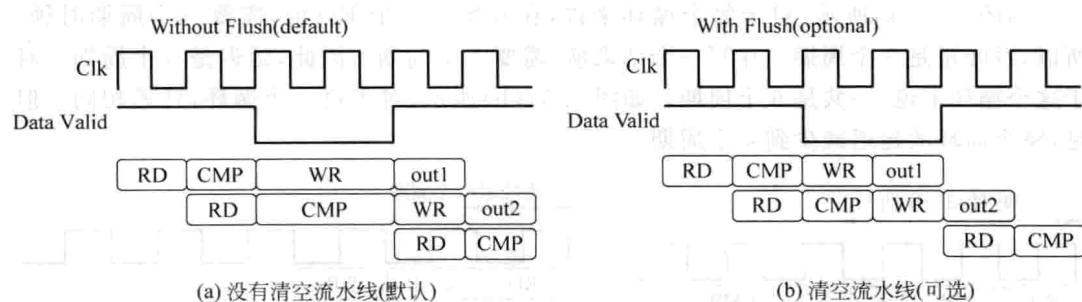


图 5.36 流水线清空/不清空

**注:** 如果在顶层循环中使用循环流水时,可能会产生一个“气泡”,也就是打断数据流。例如,对于下面的代码:

```
void foo_top(in1, in2, ...){  
    static accum = 0;  
    ...  
    L1: for(i = 1; i < N; i++){  
        accum = accum + in1 + in2;  
        ...  
        out1_data = accum;  
    }  
}
```

该函数用于处理一个数据流。如图 5.37 所示,当下次调用函数时,仍然需要执行初始化(init)操作。因此,导致打断数据流。

为了避免产生气泡,在命令选项中,选择 enable loop rewinding(使能循环回卷)。表示:

- (1) 立即重新执行顶层循环;
- (2) 操作回卷到循环的开始。

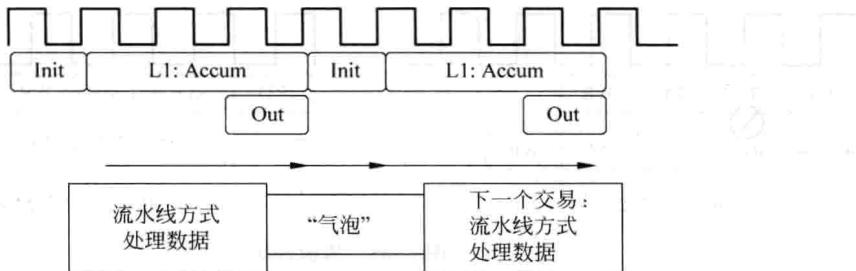


图 5.37 顶层循环流水时序

该选项在循环开始前，忽略任何初始化语句。该选项只影响顶层循环。

**注：**

- ① 确保在重新执行函数时，不会重新执行循环前的操作。
- ② 对函数使用流水线，将展开所有的循环。
- ③ 边界不能是变量，应该是常数。否则，将影响对循环的展开。
- ④ 在代码中，如果存在反馈将阻止/限制流水线。下面给出带有反馈的代码：

```
void foo_top(a,b,c,d){  
...  
    while(a!= b){  
        if(a> b)  
            a -= b;  
        else  
            b -= a;  
        ...  
    }  
}
```

在减法操作前，不能执行比较操作。所以，限制了流水线。

- ⑤ 资源冲突可能阻止流水线。典型地，对于下面的流水线的间隔设置不同值时，将产生不同的结果。

对于下面的代码：

```
void foo(m[2] ...) {  
    op_Read_m[0];           //RD  
    op_Read_m[1];           //RD  
    op_Compute;             //CMP  
    op_Write;               //WR  
}
```

如图 5.38(a)所示，当 II=1 时，由于在同一时刻，发生读操作的重叠，所以产生资源冲突。如图 5.38(b)所示，当 II=2 时，读操作发生在不同的时刻，所以不会发生读操作重叠，不产生资源冲突。

**注：**即使约束导致冲突时，Vivado HLS 总是尝试创建一个设计。此时，Vivado HLS 将自动地增加 II 的值。

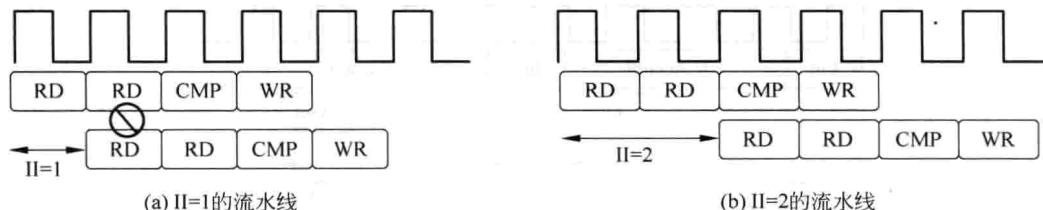


图 5.38 资源冲突

## 5.6.2 循环的处理

本节介绍对循环的处理,主要有循环展开、循环平坦化和循环合并。

### 1. 循环展开

默认地,不展开循环,即:

- (1) 在一个状态内,实现对每个 C 循环的迭代操作;
- (2) 在相同的资源内,实现对每个 C 循环的迭代操作。

对于下面的代码。默认时,将综合出图 5.39 所示的结构。

```
void foo_top( ... ){
    ...
    Add: for(i = 3;i >= 0;i --){
        b = a[i] + b;
        ...
    }
}
```

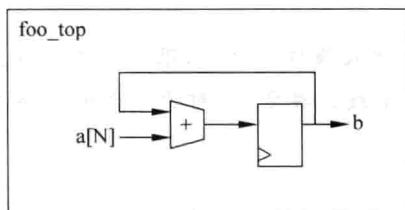


图 5.39 未展开循环的结构

在为循环添加用户约束命令时,要求使用标号对循环进行标识。即使在描述 C 模型时没有给出标号,在添加用户约束命令时也会自动地添加标号。

当且仅当循环迭代的次数为常数时,Vivado HLS 允许展开循环。

对于图 5.39 所示的结构,如果加法器的延迟小于时钟频率时,如图 5.40 所示,不管使用何种优化类型,这个未展开的循环不小于 4 个周期。最小的延迟是循环迭代次数的函数。

如图 5.41 所示,当把循环全部展开时,消耗了 4 个加法器。如图 5.42 所示,但是延迟大大降低。很明显,如果将循环全部展开,则会消耗更多的硬件资源。

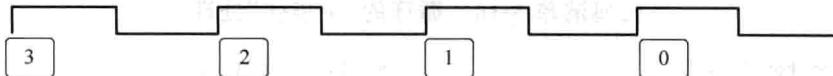


图 5.40 未展开循环迭代的时序

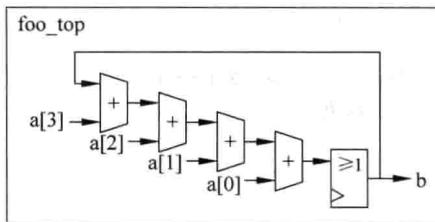


图 5.41 展开循环迭代的结构

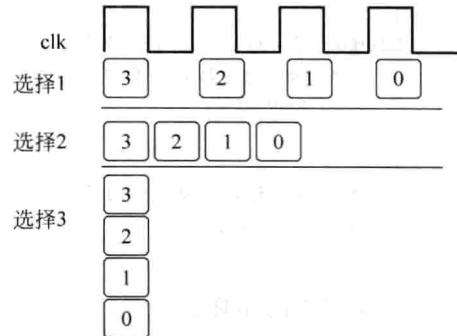


图 5.42 为展开循环迭代的时序

从上面可以看出，将循环全部展开会消耗大量的硬件资源。因此，可以根据设计条件，将循环部分地展开。

对于下面的代码：

```
Add: for(int i = 0; i < N; i++) {
    a[i] = b[i] + c[i];
}
```

可以进行改写，将循环迭代的次数变成  $N/2$ 。

```
Add: for(int i = 0; i < N; i += 2) {
    a[i] = b[i] + c[i];
    if(i + 1 >= N) break;
    a[i + 1] = b[i + 1] + c[i + 1];
}
```

## 2. 循环展开

Vivado HLS 可以自动地将嵌套的循环“平坦化”处理。与手工修改代码相比，这是一种更快的方法。在最里面的循环中，可以指定“平坦化”。

**注：**可以关闭对循环嵌套的“平坦化”处理。下面给出了对嵌套循环的平坦化处理。很明显，L2 嵌套循环被“平坦化”为一级 16 次迭代的循环。

是否能对嵌套循环进行“平坦化”处理，这和代码的书写也有关系。只有下面的情况才可以对代码进行“平坦化”处理：

(1) 完美的循环：

- ① 只有最里面的循环有循环体(内容)；
- ② 在循环描述中没有指定的逻辑；
- ③ 循环边界是常数。

## 代码清单 5-16 循环的“平坦化”处理

```

void foo_top ( ... ) {
    ...
L1: for ( i = 3; i >= 0; i -- ) {
    [loop body 11 ]
}
L2: for ( i = 3; i >= 0; i -- ) {
    L3: for ( j = 3; j >= 0; j -- ) {   └─→
        [loop body 13 ]
    }
}
L4: for ( i = 3; i >= 0; i -- ) {
    [loop body 14 ]
}
}

void foo_top ( ... ) {
    ...
L1: for ( i = 3; i >= 0; i -- ) {
    [loop body 11 ]
}
L2: for ( k = 15, k >= 0; k -- ) {
    [loop body 13 ]
}
L4: for ( i = 3; i >= 0; i -- ) {
    [loop body 11 ]
}
}

```

(2) 半完美的循环：

- ① 只有最里面的循环有循环体(内容)；
- ② 在循环描述中没有指定的逻辑；
- ③ 循环边界为变量。

(3) 其他类型：

应该将其他类型转换为完美的或者半完美的循环嵌套。

## 代码清单 5-17 循环嵌套的不同表示

```

Loop_outer: for ( i = 3; i >= 0; i -- ) {
    Loop_inner: for ( j = 3; j >= 0; j -- ) {
        [loop body]
    }
}
//完美循环

```

```

Loop_outer: for ( i = 3; i > N; i -- ) {
    Loop_inner: for ( j = 3; j >= 0; j -- ) {
        [loop body]
    }
}
//半完美循环

```

```

Loop_outer: for ( i = 3; i > N; i -- ) {
    [loop body]
    Loop_inner: for ( j = 3; j >= M; j -- ) {
        [loop body]
    }
}
//其他循环

```

## 3. 循环合并

Vivado HLS 可以自动地合并循环。与手工修改代码相比，这是一种比较快的方法。通过将循环进行合并，允许探索更加高效的实现结构。

注：FIFO 读要求严格的顺序，这样将阻止循环合并。可以在使用 MERGE 命令时，在 Option 中，选择 force 选项。这样做的结果是，用户负责管理读 FIFO 的正确性。

下面代码说明了循环的合并。

### 代码清单 5-18 循环合并

```

void foo_top ( ... ) {
    ...
L1: for ( i = 3; i >= 0; i -- ) {
    [loop body 11]
}
L2: for ( i = 3; i >= 0; i -- ) {
    L3: for ( j = 3; j >= 0; j -- ) { →
        [loop body 13]
    }
}
L4: for ( i = 3; i >= 0; i -- ) {
    [loop body 14]
}
}

void foo_top ( ... ) {
    ...
L123: for ( l = 16, l >= 0; l -- ) {
    if (cond1)
        [loop body 11]
    [loop body 13]
    if (cond4)
        [loop body 14]
}
}

```

下面给出合并循环的规则：

- (1) 如果循环边界都是变量，则边界的值应该相同；
- (2) 如果循环边界是常数，最大的边界值用作合并循环的边界；
- (3) 不能合并带有可变边界和常数边界的循环；
- (4) 对于所合并的循环之间的代码，不能有“副作用”，即不能破坏代码的正确执行；
- (5) 总是顺序执行对 FIFO 或者 FIFO 接口的读取操作。

### 5.6.3 数组的处理

数组是一种直观且有用的软件结构。它让设计者能够很容易地捕获和理解 C 算法。对数组的访问经常变成性能上的瓶颈。默认地，数组对应到 RAM。对于下面的算法：

```

void foo_top( ... ){
    ...
    for(i = 2; i < N; i++)
        mem[ i ] = mem[ i - 1 ] + mem[ i - 2 ];
}

```

如图 5.43 所示，给出了存储器读写时序。对于这个操作，甚至通过一个双端口存储器，也不能执行所有读和写操作。这样，会造成性能瓶颈。

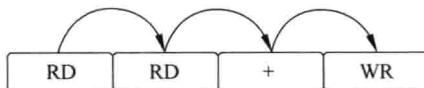


图 5.43 存储器读写时序

Vivado HLS 允许对数组进行“分割”(partition)和“重组”(reshape)。这样：

- (1) 允许对存储器进行更加优化的配置；
- (2) 提供了对存储器资源的更好实现方式。

对于下面的代码，经过综合后，得到如图 5.44 所示的结构。如图 5.45 所示，对于这个结构，Vivado HLS 可以使用库中所提供的任何存储器资源。在库模型中，定义了端口和时序操作。

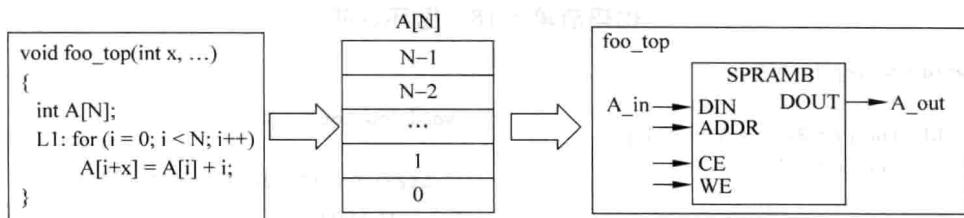


图 5.44 综合后的存储器结构

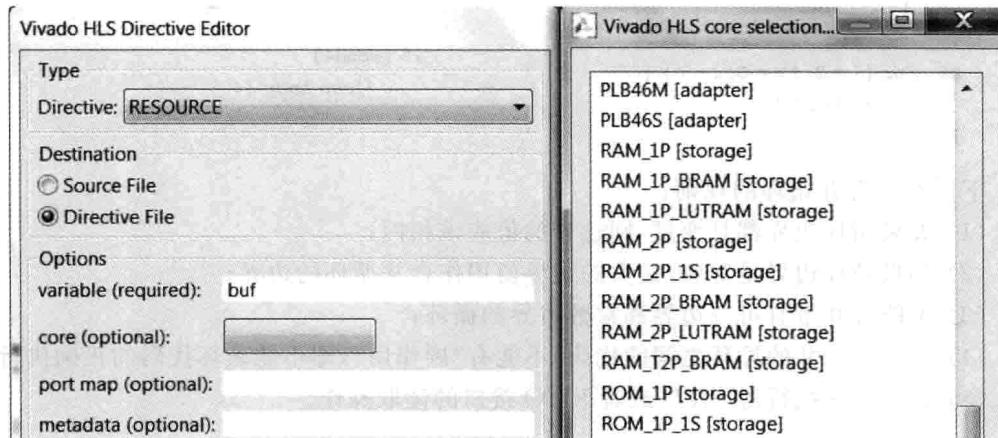


图 5.45 库中的存储器的结构

如果没有选择 RAM 资源, Vivado HLS 将推断所使用的 RAM。即:

- (1) 如果提高吞吐量, 使用双端口存储器;
- (2) 否则, 使用单端口存储器。

对于选择 BRAM 还是 LUTRAM, 即:

- (1) 如果没有指定, 例如使用 RAM\_1P。则 RTL 综合将确定是否 RAM 使用 BRAM 或者 LUTRAM;
- (2) 如果用户指定了 RAM 目标(例如 RAM\_IP\_BRAM 或者 RAM\_IP\_LUTRAM), Vivado HLS 将遵守这个目标。如果选择 LUTRAM, Vivado HLS 将报告寄存器而不是 BRAM。

### 1. 数组分割

数组分割(Array Partitioning)是将一个数组分割成更小的元素。如图 5.46 所示, 其中:

- (1) 如果 factor 不是整数, 最终的数组有很多的元素。

(2) 数组可以分割成任何的维度。

- ① 如果在 Option 没有指定维数(dimension), 则将其假定为 0。
- ② dimension=0, 表示所有的维数。

如图 5.47 所示, 对于下面数组:

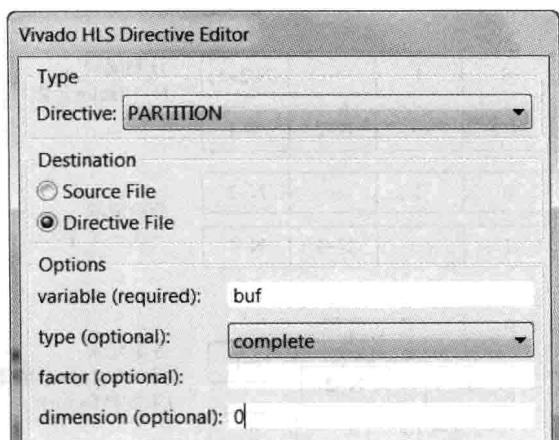


图 5.46 数组分割选项

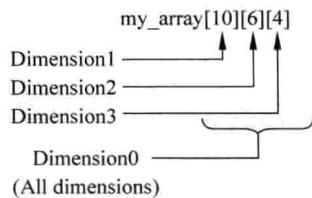


图 5.47 数组维数的表示

① 分割维数 3, 结果是:

```
my_array_0[10][6]
my_array_1[10][6]
my_array_2[10][6]
my_array_3[10][6]
```

② 分割维数 1, 结果是:

```
my_array_0[6][4]
my_array_1[6][4]
my_array_2[6][4]
my_array_3[6][4]
my_array_4[6][4]
my_array_5[6][4]
my_array_6[6][4]
my_array_7[6][4]
my_array_8[6][4]
my_array_9[6][4]
```

③ 分隔维数 0, 结果是生成  $10 \times 6 \times 4 = 240$  个单独的元素。

(3) 所有分割也使用相同的资源目标。

如图 5.48 所示, 给出了 PARTITION 的类型分别为 block、cyclic 和 complete 时, 对数组处理产生的不同结构。

Vivado HLS 可以自动地将数组进行分割, 以提高设计吞吐量。如图 5.49 所示, 通过数组配置命令进行控制。

(1) 在此选中 throughput\_driven, 使能吞吐量驱动。

(2) 带有常数索引的自动分区。

① 数组索引不是变量;

② 低于门限的数组, 被自动分区。

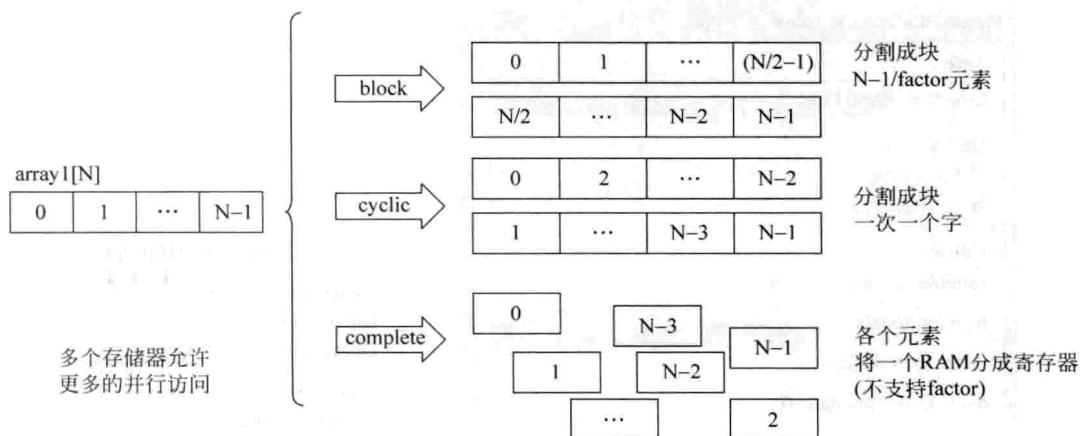


图 5.48 数组的不同分割方式

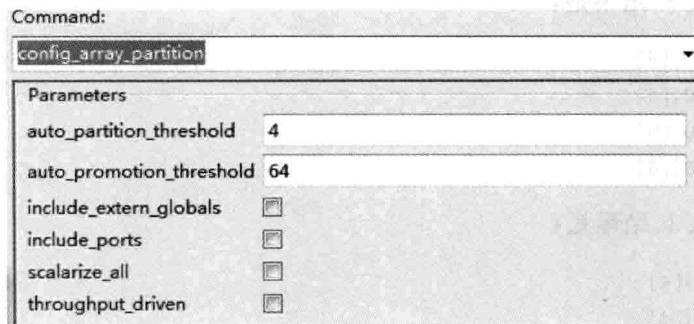


图 5.49 数组配置界面

(3) 使用选项 elem\_count\_limit, 设置门限。

(4) 通过使用选项 scalarize\_all, 对设计内的所有数组进行分割。

(5) 包含所有的数组：

① include\_ports 选项：当执行分割的时候，在 I/O 接口上，包含任何的数组。这是因为对数组进行分割，会导致出现多个端口，并且会改变接口。然而，会提高设计的吞吐量。

② include\_extern\_globals 选项：任何定义为全局的数组，都将包含在分区中。

## 2. 数组重组

数组重组将分割的数组重新组合为一个单个的数组。如图 5.50 所示，数组重组的命令选项和数组分隔完全一样。但是，数组重组将自动地进行重组，将这些部分组合成一个单个元素。新组成的数组有相同的名字。

## 3. 数据打包

结构体类似于数组，结构体是一个通用的编码结构。默认地，将结构分隔成结构体

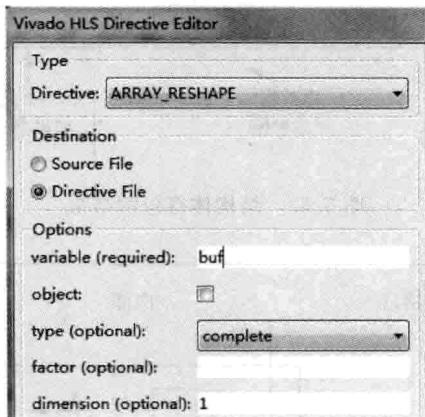


图 5.50 数组重组设置命令界面

内包含的各自的元素。例如对于下面的结构体，默认地，生成图 5.51 所示的结构。如图 5.52 所示，给出了结构和内部连接的关系：

```
type struct{
    unsigned char A;
    unsigned char B[4];
    unsigned char C;
}
void foo(my_data * a);
```

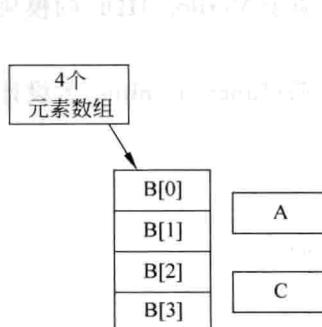


图 5.51 结构体的结构

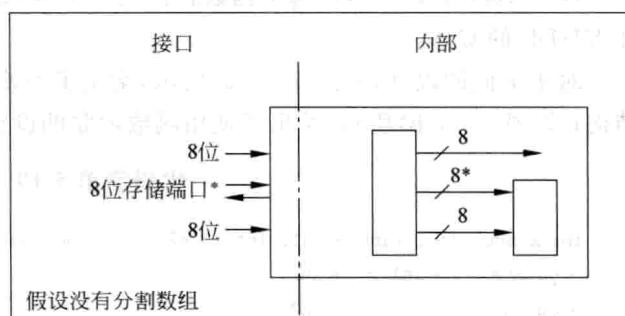


图 5.52 结构体生成的接口

- (1) 接口上是单独的接口；
- (2) 独立的总线和连线；
- (3) 独立的控制逻辑。

这样可能更加复杂、速度变慢，并且增加了延迟。

如图 5.53 所示，数据打包将结构体内的元素构成一组，将其放置在 I/O 接口上。对于上面的结构体，当数据打包后，如图 5.54 所示，生成了一个接口，通过该接口与内部进行连接。

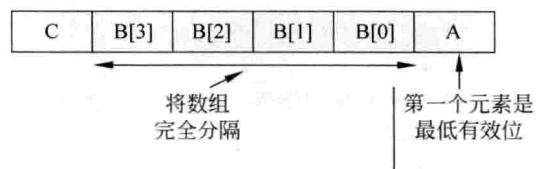


图 5.53 结构体打包的结构

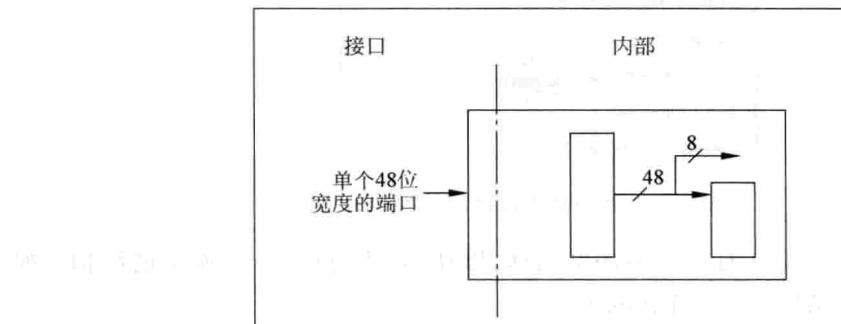


图 5.54 结构体打包生成的接口

#### 5.6.4 函数内联

在 Vivado HLS 中, 将每个函数转换成一个 RTL 块, 其对应于 Verilog HDL 的模块和 VHDL 的实体。

对于下面的设计, 如图 5.55(a)所示, 给出了不使用函数内联(function inline)的设计结构; 如图 5.55(b)所示, 给出了使用函数内联的设计结构。

##### 代码清单 5-19

```

int sumsub_func ( int * in1, int * in2, int * outSum, int * outSub) {
    * outSum = * in1 + * in2;
    * outSub = * in1 - * in2;
}
int shift_func ( int * in1, int * in2, int * outA, int * outB) {
    * outA = * in1 >> 1;
    * outB = * in2 >> 2;
}
void add_sub_pass(int A, int B, int * C, int * D) {
    int apb, amb;
    int a2, b2;

    sumsub_func(&A, &B, &apb, &amb);
    sumsub_func(&apb, &amb, &a2, &b2);
    shift_func(&a2, &b2, C, D);
}

```

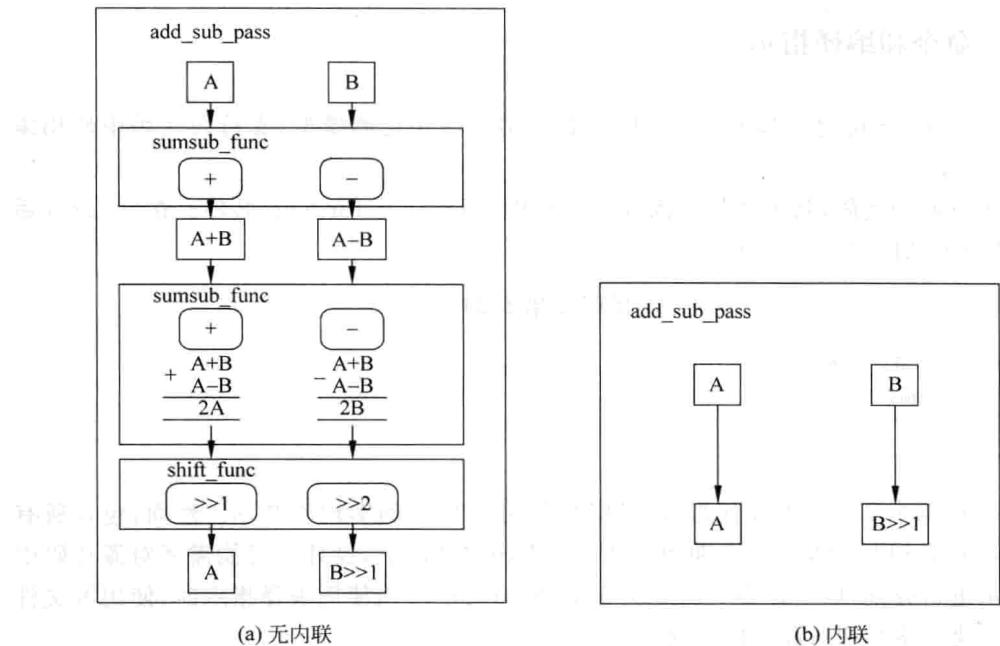


图 5.55 函数的不同处理方式

从图 5.55(b)中可以看出,函数内联允许跨越函数层次进行优化。但是,过多的内联将创建大量的逻辑,并且会降低运行速度。

使用函数内联(function inline)，可以去掉函数之间的层次。Vivado HLS 自动执行某些内联。如图 5.56 所示，设计者可以通过 INLINE 命令，使用函数内联策略：

- (1) recursive: 当选中该选项时,按层次向下递归内联所有的函数。
  - (2) region: 当选中该选项时,内联指定区域内的所有函数。

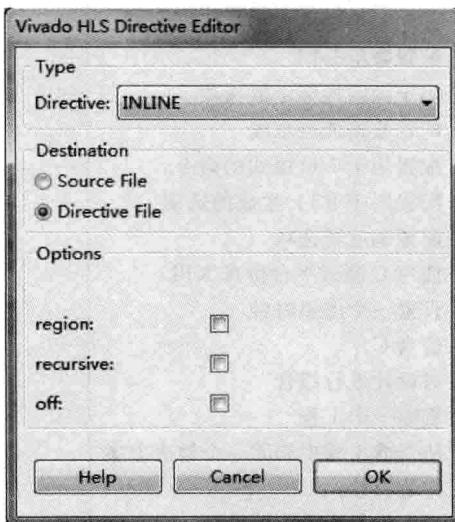


图 5.56 INLINE 命令配置

### 5.6.5 命令和编译指示

Vivado HLS 优化是基于代码中的位置。用于待优化的模型，支持在代码中使用编译指示(pragma)。

用于下面的代码，将编译指示插入到代码中，用于展开 for-loop 循环。在 pragma 后面，总是指定 AP，然后是指示。

**代码清单 5-20**

```
for( int i = 0; i < 64; ++i){  
    pragma HLS UNROLL  
    ...  
}
```

在 C 代码中所输入的任何指示，可以被使用 C 规范的实现所共享。然而，包含所有优化指示的 C 规范可能是一个期望的特征。在很多情况下，设计者可能需要对源代码中编译指示进行分割，以创建不同的解决方案(Solution)。当使用编译指示时，使用源文件的每个解决方案都执行相同的优化。

在上面的例子中，命令 set\_directive 用于命令行接口(提供有一个命名称号的循环)。可选择的另一种方式是在 Directive 标签下，将指示直接添加到源代码中。

注：pragma 编译指示不能使用 define 所定义的值。

表 5.3 给出了用于优化指示的相关 pragma。

**表 5.3 用于优化指示的相关 pragma**

TCL 命令	描述	# pragma
add_files	添加文件到当前的工程中	—
close_project	关闭目前正在打开的工程	—
close_solution	关闭一个解决方案	—
config_array_partition	配置数组分割	—
config_bind	用于微结构绑定的选项	—
config_dataflow	配置数据流流水线	—
config_interface	配置用于 I/O 模式的命令	—
config_rtl	配置用于 RTL 生成的选项	—
config_schedule	配置调度器选项	—
cosim_design	使用 C 测试平台仿真 RTL	—
create_clock	设置一个虚拟时钟	—
csim_design	仿真 C	—
csynth_design	对设计进行综合	—
delete_project	删除一个工程	—
delete_solution	从当前工程中删除一个解决方案	—
export_design	封装设计	—
help	打印一条命令的手册页	—

续表

TCL 命令	描述	# pragma
list_core	列出源库中的核	
list_part	列出所支持的元件	
open_project	打开或者创建一个工程	
open_solution	打开或者创建一个解决方案	
set_clock_uncertainty	设置时钟的不确定性	
set_directive_allocation	命令 ALLOCATION	# pragma HLS allocation instances = < Instance Name List > limit = < Integer Value > < operation, function >
set_directive_array_map	命令 ARRAY_MAP	# pragma HLS array_map variable = < variable > \ instance = < instance > < horizontal, vertical > \ offset = < int >
set_directive_array_partition	命令-ARRAY_PARTITION	# pragma HLS array_partition variable = < variable > < block, cyclic, complete > \ factor = < int > dim = < int >
set_directive_array_reshape	命令-ARRAY_RESHAPE	# pragma HLS array_reshape variable = < variable > \ < block, cyclic, complete > factor = < int > dim = < int >
set_directive_clock	命令-CLOCK	# pragma HLS clock domain = < string >
set_directive_data_pack	命令-DATA_PACK	# pragma HLS data_pack variable = < variable > instance = < string >
set_directive_dataflow	命令-DATAFLOW	# pragma HLS dataflow
set_directive_dependence	命令-DEPENDENCE	# pragma HLS dependence variable = < variable > \ < array, pointer > < inter, intra > < RAW, WAR, WAW > distance = < int > < false, true >
set_directive_expression_balance	命令-EXPRESSION_BALANCE	# pragma HLS expression_balance < off >
set_directive_function_instantiate	命令-FUNCTION_INSTANTIATE	# pragma HLS function_instantiate variable = < variable >
set_directive_inline	命令-INLINE	# pragma HLS inline < region   recursive   off >
set_directive_interface	命令-INTERFACE	# pragma HLS interface < mode > register port = < string >
set_directive_latency	命令-LATENCY	# pragma HLS latency min = < int > max = < int >
set_directive_loop_flatten	命令-LOOP_FLATTEN	# pragma HLS loop_flatten off

续表

TCL 命令	描 述	# pragma
set_directive_loop_merge	命令-LOOP_MERGE	# pragma HLS loop_merge force
set_directive_loop_tripcount	命令-LOOP_TRIPCOUNT	# pragma HLS loop_tripcount \min=<int> \max=<int> avg=<int>
set_directive_occurrence	命令-OCCURRENCE	# pragma HLS occurrence cycle = <int>
set_directive_pipeline	命令-PIPELINE	# pragma HLS pipeline II=<int> enable_flush \rewind
set_directive_protocol	命令-PROTOCOL	# pragma HLS protocol <floating, fixed>
set_directive_reset	命令-RESET	# pragma HLS reset variable=a off
set_directive_resource	命令-RESOURCE	# pragma HLS resource variable = <variable> \core=<core>
set_directive_stream	命令-STREAM	# pragma HLS streamvariable = <variable> \off depth=<int>
set_directive_top	命令-TOP	# pragma HLS top name = <string>
set_directive_unroll	命令-UNROLL	# pragma HLS unroll skip_exit_check factor=<int>\region
set_part	设置目标器件	—
set_top	设置顶层模型	—

## 5.7 基于 HLS 的数字系统实现

为了说明 HLS 工具在数字系统设计中的应用,下面通过组合逻辑、时序逻辑、矩阵相乘这三个有代表性的应用来说明高级综合工具 HLS 在数字系统实现中的应用。同时,也帮助读者进一步地理解和掌握 HLS 的设计流程,以及 FPGA 在高性能信号处理中的应用。

### 5.7.1 基于 HLS 实现组合逻辑

本部分内容包含:设计组合逻辑模型、设计综合、设计优化、查看生成的数据处理图、运行 C 仿真和验证功能、运行 RTL 级仿真等内容。

#### 1. 设计组合逻辑模型

下面将在 Vivado HLS 中创建一个完成 6 种逻辑运算的模型。这 6 种逻辑运算包含:逻辑与、逻辑与非、逻辑或、逻辑或非、逻辑异或、逻辑异或非操作。在 HLS 中实现 6 种逻辑运算描述的步骤主要包括:

(1) 打开 Vivado HLS 软件设计工具,下面给出两种方法:

注:

① 在使用 Vivado HLS 软件设计工具前,必须得到 Vivado HLS License 许可文件;

- ② 参考 Xilinx 提供的关于 Vivado HLS 工具的许可授权说明；
- ③ 本书所给的 HLS 的设计实例是在 Vivado HLS 2013.3 集成开发环境下完成的；
- ④ 本书所给的 HLS 的设计实例是在 Xilinx 大学计划开发平台 Nexys4 上完成的。
- ① 在 Windows 7 系统主界面下,选择开始→Xilinx Design Tools→Vivado 2013.3→Vivado HLS 2013.3；

- ② 在 Windows 7 系统桌面上,选择并双击如图 5.57 所示的图标。
- (2) 出现 Vivado HLS 设计主界面。如图 5.58 所示,在主界面内单击 Create New Project 条目,进入创建新工程设计向导界面。



图 5.57 Vivado HLS 图标



Create New Project

New Project Wizard will guide you through the process of selecting design sources and a target device for a new project.

图 5.58 创建新工程设计向导

- (3) 如图 5.59 所示,出现 New Vivado HLS Project-Project Configuration(新 Vivado HLS 工程-工程配置)向导界面。在该界面内按如下设置参数:

- ① Project name: gate.prj;
- ② Location: E:\vivado\_example\hls\_basic\logic\_gate;
- ③ 单击 Next 按钮。

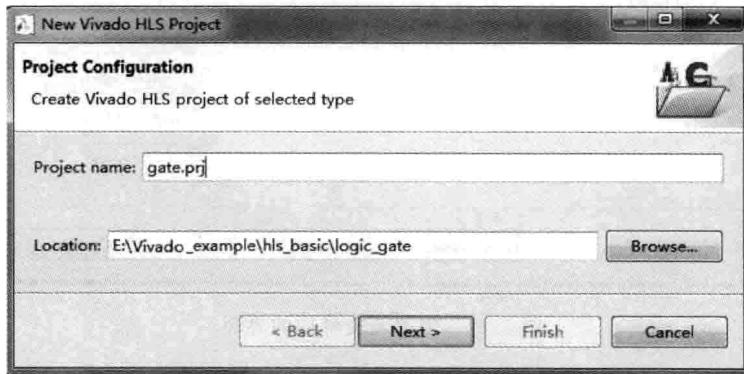


图 5.59 工程配置界面

- (4) 如图 5.60 所示,出现 New Vivado HLS Project-Add/Remove Files(新 Vivado HLS 工程-添加/删除源文件)界面,在该界面下按如下设置参数:

- ① 单击 New File 按钮。按下面步骤操作:

出现另存为文件对话框界面,输入文件名 gate.c,单击保存按钮。

- ② 在 Top Function 右侧输入顶层函数的名字 gate。

如图 5.61 所示,给出了创建新文件后的对话框界面。

- ③ 单击 Next 按钮。

- (5) 出现 Add/Remove Files-Add/remove C-based source files(design specification)(添加/删除文件-添加/删除基于 C 的测试平台文件(设计测试))界面,不设置任何参数,

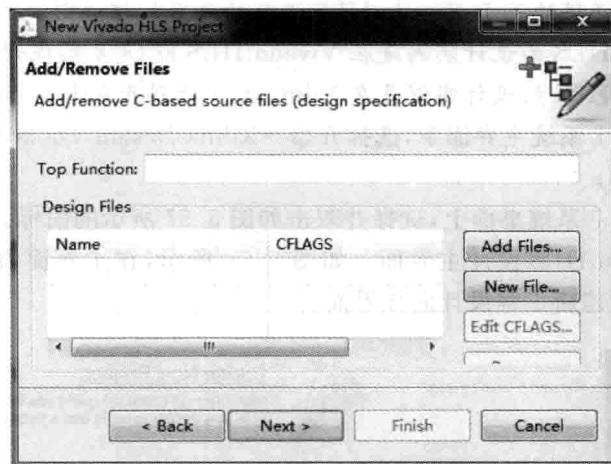


图 5.60 添加/删除文件界面

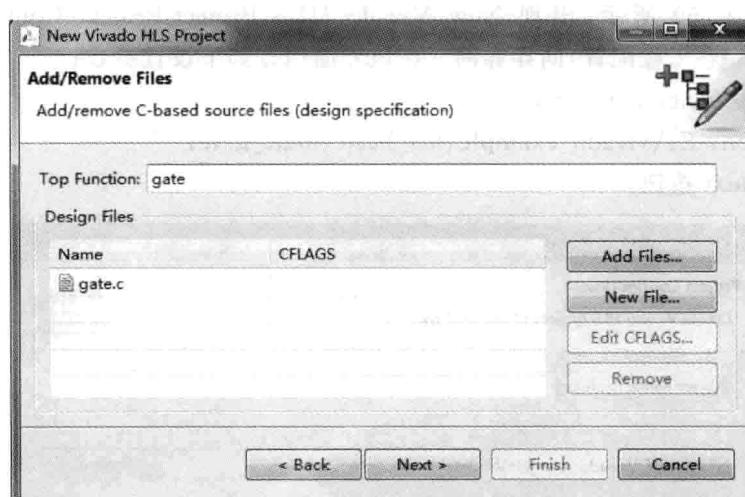


图 5.61 创建新文件后的界面

单击 Next 按钮。

(6) 如图 5.62 所示, 出现 Solution Configuration-Create Vivado HLS solution for selected technology(解决方案配置-创建用于所选择技术的 Vivado HLS 解决方案)对话框界面。在该对话框中, 单击 **Next >** 按钮。按下面步骤操作:

① 如图 5.63 所示, 出现 Device Selection Dialog(器件选择对话框)界面。在该界面中, 选择 xc7a100tcsg324-1(器件)。

② 然后单击 OK 按钮。

**注:** 为了加快搜索的速度, 在 Filter 标题栏下面不同的选项中, 通过下拉框, 选择器件过滤条目:

① Family: artix7;

② Sub-Family: artix7;

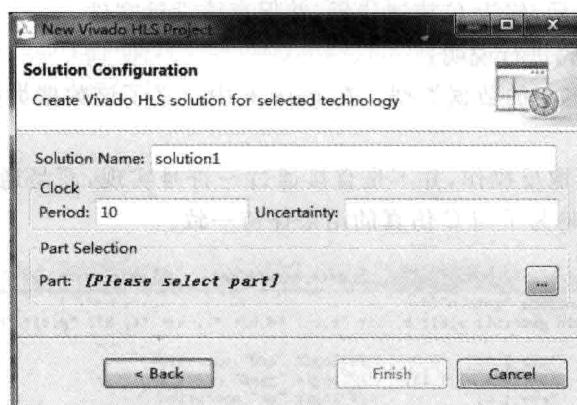


图 5.62 选择解决配置方案

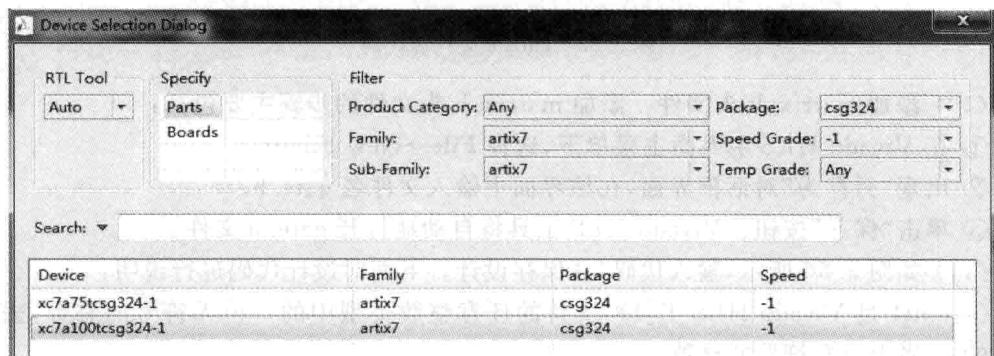


图 5.63 器件选择界面

- ③ Package: csg324;
- ④ Speed Grade: -1;
- ⑤ Temperature Grade: Any。
- ⑥ 退回到图 5.62 所示的界面。
- (7) 如图 5.62 所示,单击 Finish 按钮。
- (8) 如图 5.64 所示,在 Vivado HLS 主界面左侧的 Explorer 窗口下,出现工程设计文件目录列表。在该窗口下,展开 Source, 在展开项中,选择 gate.c 文件。双击该文件名,打开 gate.c 文件。

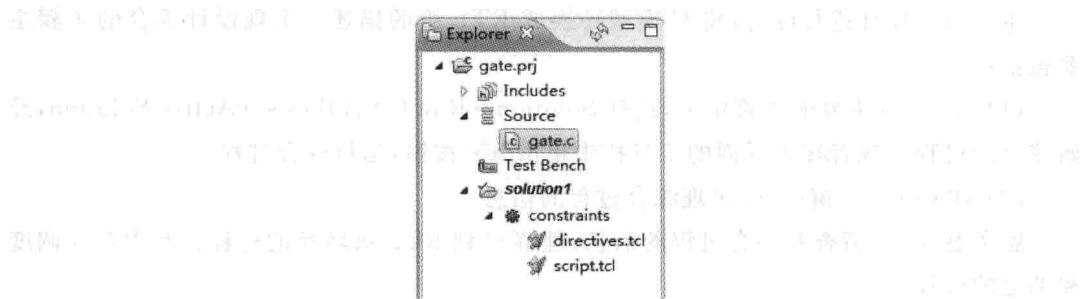


图 5.64 工程设计文件列表

(9) 如图 5.65 所示,输入 C 描述代码,并保存该设计文件。

下面对设计源代码进行说明:

① bit 为作者自定义的数据类型。在 gate.h 中定义了该数据类型,表示一位无符号的整数。

② 设计中的按位取反操作,并不是直接通过 $\sim$ 符号实现,而是通过和 1 进行逻辑异或操作得到。这样做是为了与 C 仿真的结果保持一致。

```

1 #include "gate.h"
2 void gate(bit a,bit b, bit *c,bit *d,bit *e, bit *f, bit *g,bit *h)
3{
4     *c=a & b;           // logic "and" operation
5     *d=((a & b) ^ 1); // logic "nand" operation
6     *e=a | b;          // logic "or" operation
7     *f=((a | b) ^ 1); // logic "nor" operation
8     *g=a ^ b;          // logic "xor" operation
9     *h=((a ^ b) ^ 1); // logic "xnor" operation
10}
11

```

图 5.65 gate.c 文件源代码

(10) 添加 matrix.h 头文件。添加 matrix.h 头文件的步骤主要包括:

① 在 Vivado HLS 主界面主菜单下,选择 File→New File…。

② 出现“另存为”对话框界面,在该界面中输入文件名 gate.h。

③ 单击“保存”按钮。Vivado HLS 工具将自动地打开 gate.h 文件。

(11) 如图 5.66 所示,输入代码,并保存设计。下面对设计代码进行说明:

① uint1 是 Vivado HLS 工具所支持的任意整数类型中的一位无符号的整数,去掉前面的 u,将表示有符号的整数。

② 在使用任意整数类型时,需要包含头文件 ap\_cint.h。

```

1 ifndef _GATE_
2 define _GATE_
3 include "ap_cint.h"
4 typetypedef uint1 bit;
5 endif
6

```

图 5.66 gate.h 文件源代码

## 2. 设计综合

本节将对设计进行综合,将 C 模型转换成 RTL 级的描述。实现设计综合的步骤主要包括:

(1) 在 HLS 主界面主菜单下,选择 Solution→Run C Synthesis→Active Solution,开始综合的过程。或者在主界面的工具栏中单击 按钮,运行综合过程。

(2) 在 Console 窗口下,出现综合过程的信息。

思考题 5.1: 请查看综合过程的信息,理解 C 到 RTL 级转换的过程。其中存在调度和绑定的过程。

(3) 综合完成后,自动打开综合后的报告 gate\_csynth.rpt 文件。下面对报告进行

分析：

- ① 如图 5.67 所示,给出了一般信息。

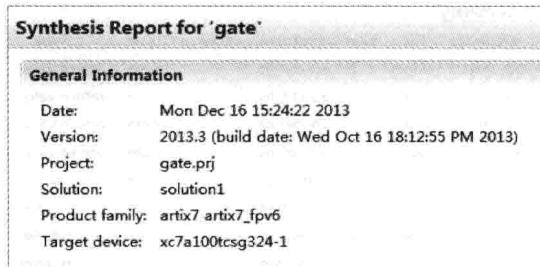


图 5.67 综合后报告给出一般信息

- ② 如图 5.68 所示,给出了性能信息。

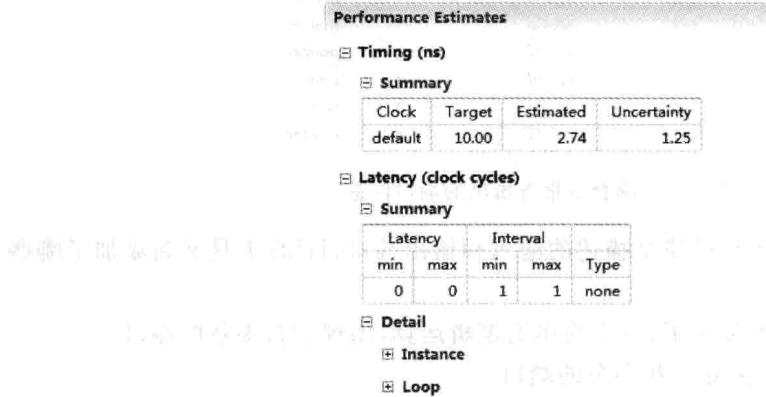


图 5.68 综合后报告给出的性能信息

思考题 5.2：请给出 Latency(延迟)和 Interval(间隔)的值,这些值给出什么性能信息。

- ③ 如图 5.69 所示,给出了器件利用率信息。

思考题 5.3：请给出该设计中所消耗 FPGA 的逻辑资源的个数。

**Utilization Estimates**

**Summary**

Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	-	0	12
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	-	-
ShiftMemory	-	-	-	-
Total	0	0	0	12
Available	270	240	126800	63400
Utilization (%)	0	0	0	~0

图 5.69 综合后报告给出的器件利用率信息

④ 如图 5.70 所示,给出了综合后该设计所提供的端口。

Interface						
Summary						
RTL Ports	Dir	Bits	Protocol	Source Object	C Type	
ap_start	in	1	ap_ctrl_hs	gate	return value	
ap_done	out	1	ap_ctrl_hs	gate	return value	
ap_idle	out	1	ap_ctrl_hs	gate	return value	
ap_ready	out	1	ap_ctrl_hs	gate	return value	
a	in	1	ap_none		a	scalar
b	in	1	ap_none		b	scalar
c	out	1	ap_vld		c	pointer
c_ap_vld	out	1	ap_vld		c	pointer
d	out	1	ap_vld		d	pointer
d_ap_vld	out	1	ap_vld		d	pointer
e	out	1	ap_vld		e	pointer
e_ap_vld	out	1	ap_vld		e	pointer
f	out	1	ap_vld		f	pointer
f_ap_vld	out	1	ap_vld		f	pointer
g	out	1	ap_vld		g	pointer
g_ap_vld	out	1	ap_vld		g	pointer
h	out	1	ap_vld		h	pointer
h_ap_vld	out	1	ap_vld		h	pointer

图 5.70 综合后报告给出的端口信息

思考题 5.4: 请查看除了 C 模型描述的输入和输出端口,HLS 工具又新添加了哪些新的端口。

结论: 所生成的端口太复杂了,一个简单的逻辑运算,出现这么多新的端口。

下面将使用用户策略,去除一些多余的端口。

### 3. 设计优化

本节将通过添加指令对 HLS 综合过程进行优化。添加用户命令的步骤主要包括:

- (1) 打开 gate.c 文件。
- (2) 在打开 gate.c 文件的右侧窗口内,选择 Directive 标签。
- (3) 如图 5.71 所示,在 Directive 标签窗口下,选择 gate,单击右键出现浮动菜单,选择 Insert Directive... 选项。
- (4) 打开如图 5.72 所示的界面,在该界面窗口下,按下面设置参数:
  - ① 在 Directive 右侧的下拉框中选择 Interface;
  - ② 在 mode(optional)右侧下拉框中选择 ap\_ctrl\_none;
  - ③ 单击 OK 按钮。
- (5) 在 HLS 主界面主菜单下选择 Solution→Run C Synthesis→Active Solution,开始综合的过程;或者在主界面的工具栏中单击  按钮,启动综合的过程。

思考题 5.5: 如图 5.73 所示,再次查看设计报告中的生成端口信号,减少了所生成的端口的个数。

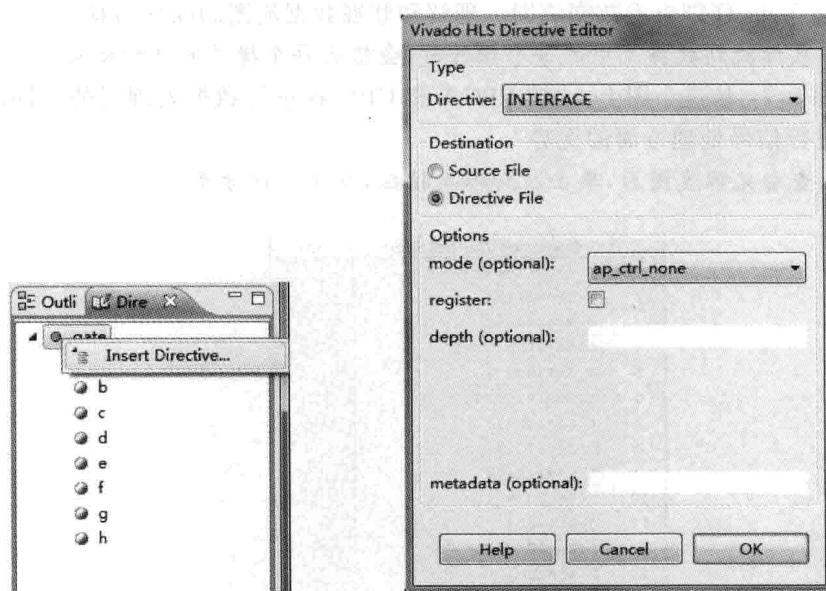


图 5.71 打开插入命令窗口

图 5.72 Vivado HLS 命令窗口界面

Interface						
Summary						
RTL Ports	Dir	Bits	Protocol	Source Object	C Type	
a	in	1	ap_none	a	scalar	
b	in	1	ap_none	b	scalar	
c	out	1	ap_vld	c	pointer	
c_ap_vld	out	1	ap_vld	c	pointer	
d	out	1	ap_vld	d	pointer	
d_ap_vld	out	1	ap_vld	d	pointer	
e	out	1	ap_vld	e	pointer	
e_ap_vld	out	1	ap_vld	e	pointer	
f	out	1	ap_vld	f	pointer	
f_ap_vld	out	1	ap_vld	f	pointer	
g	out	1	ap_vld	g	pointer	
g_ap_vld	out	1	ap_vld	g	pointer	
h	out	1	ap_vld	h	pointer	
h_ap_vld	out	1	ap_vld	h	pointer	

图 5.73 优化策略后生成的端口

#### 4. 查看生成的数据处理图

本节将查看生成的数据处理图,了解 HLS 工具如何把 C 模型转换成 FPGA 上的操作数据流。查看所生成数据处理图的步骤主要包括:

- (1) 在 Vivado HLS 主界面主菜单下,选择 Solution→Open Analysis Perspective,或者单击 **Analysis** 按钮。
- (2) 如图 5.74 所示,打开数据调度图。在 C0 周期内,共进行了 14 个操作,包括读取 a 和 b,以及逻辑结果的分配。

思考题 5.6：仔细查看数据流图。理解和掌握数据流图的操作信息。

注：将鼠标放到数据流中的每个操作时，会给出每个操作的详细信息。

思考题 5.7：从这个图上，比较 FPGA 和 CPU 在进行数据处理时的不同之处，以及 FPGA 在进行信号处理方面的优势。

注：在查看完调度图后，单击 **Synthesis** 按钮，准备进行仿真。

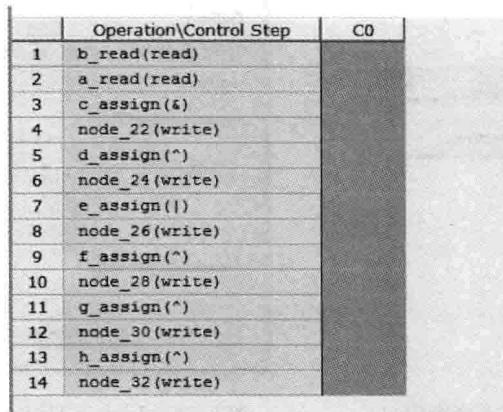


图 5.74 数据操作调度图

## 5. 运行 C 仿真和验证功能

本节将运行仿真，并对 C 模型进行验证。实现这个过程的步骤主要包括：

(1) 为了对 C 模型进行测试，需要构建基于 C 语言的测试平台文件。构建基于 C 语言测试平台文件的步骤包括：

- ① 如图 5.75 所示，在 Vivado HLS 主界面左侧的 Explorer 窗口下，找到并选择 TestBench 条目。单击右键，出现浮动菜单。在浮动菜单内，选择 New File... 选项。
- ② 出现“另存为”对话框界面，在该界面中输入文件名 gate\_test.c。
- ③ 单击“保存”按钮。HLS 工具将自动打开 gate\_test.c 文件。



图 5.75 添加测试平台文件

(2) 如图 5.76 所示，输入测试代码，并保存该测试文件。下面对该测试文件进行说明：

- ① 给 a 和 b 四组不同的测试向量，a 和 b 测试向量依次按照“00”、“01”、“10”和“11”

变化；

- ② 依次调用 gate 函数；
- ③ 然后把结果打印出来。

```

1 #include "gate.h"
2 main()
3 {
4     bit a=0;
5     bit b=0;
6     bit c,d,e,f,g,h;
7     gate(a,b,&c,&d,&e,&f,&g,&h);
8     printf("a=%d, b=%d, c=%d, d=%d, e=%d, f=%d, g=%d, h=%d\n",a,b,c,d,e,f,g,h);
9     a=0;
10    b=1;
11    gate(a,b,&c,&d,&e,&f,&g,&h);
12    printf("a=%d, b=%d, c=%d, d=%d, e=%d, f=%d, g=%d, h=%d\n",a,b,c,d,e,f,g,h);
13    a=1;
14    b=0;
15    gate(a,b,&c,&d,&e,&f,&g,&h);
16    printf("a=%d, b=%d, c=%d, d=%d, e=%d, f=%d, g=%d, h=%d\n",a,b,c,d,e,f,g,h);
17    a=1;
18    b=1;
19    gate(a,b,&c,&d,&e,&f,&g,&h);
20    printf("a=%d, b=%d, c=%d, d=%d, e=%d, f=%d, g=%d, h=%d\n",a,b,c,d,e,f,g,h);
21
22 }

```

图 5.76 添加测试平台文件代码

(3) 如图 5.77 所示,在 Vivado HLS 主界面的工具栏内,单击黑框内的 Run C Simulation 按钮;或者在主界面主菜单下,选择 Solution→Run C/RTL Cosimulation。

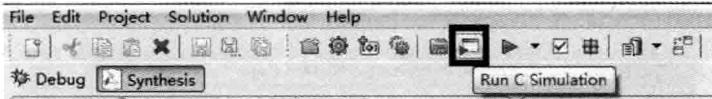


图 5.77 点击 Run C Simulation 按钮

(4) 出现 Warning(警告)对话框界面。单击 OK 按钮。

(5) 出现如图 5.78 所示的界面,按下面设置参数:

- ① Verilog/VHDL Simulator Selection: Auto;
- ② RTL Selection: SystemC;
- ③ 单击 OK 按钮。

下面对图 5.78 中的选项进行说明:

① Setup Only: 这将创建用于运行仿真的所有文件(封装器、适配器、脚本等)。但是,并不执行仿真过程。之后,可以在命令窗口内运行仿真。

② Dump Trace: 这使得将一个 VCD 文件写到目录: \solution\sim\systemc 下。

③ Optimizing Compile: 当编译 C 测试平台文件时,这个选项使用更高级的优化。

(6) 如图 5.79 所示,看到在 Vivado HLS 的控制台界面下,给出了仿真的信息。很明显基于 C 模型的仿真结果,达到了设计的要求。

## 6. 运行 RTL 级仿真

本节将在 Vivado 2013.3 环境下实现对设计的仿真。在 Vivado 2013.3 集成环境下对设计进行 RTL 级仿真的步骤主要包括:

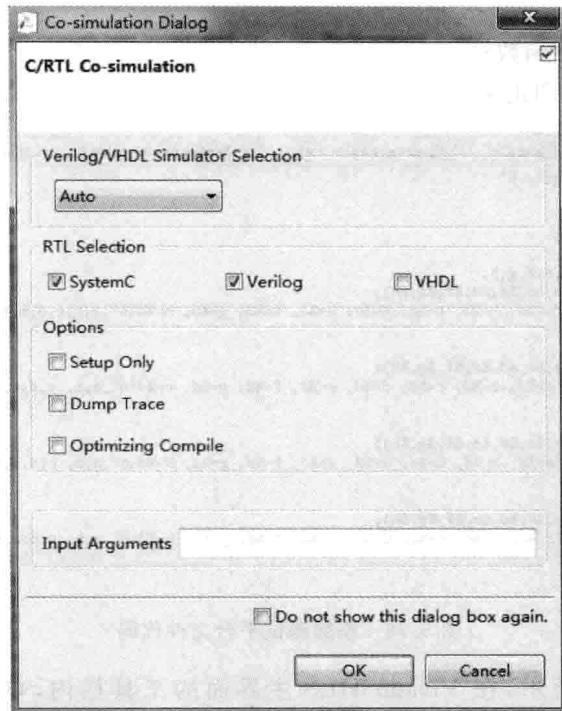


图 5.78 Co-simulation Dialog 界面

```

Console X Errors Warnings Man Page
Vivado HLS Console
clang: warning: argument unused during compilation: '-fno-builtin-isnan'
gcc.exe: warning: d:/Xilinx/Vivado_HLS/2013.3/bin/unwrapped/win32.o/apcc
@I [APCC-1] APCC is done.
@I [LIC-101] Checked in feature [VIVADO_HLS]
    Generating cosim.tv.exe
@I [SIM-302] Generating test vectors ...
a=0, b=0, c=0, d=1, e=0, f=1, g=0, h=1
a=0, b=1, c=0, d=1, e=1, f=0, g=1, h=0
a=1, b=0, c=0, d=1, e=1, f=0, g=1, h=0
a=1, b=1, c=1, d=0, e=1, f=0, g=0, h=1
@I [SIM-333] Generating C post check test bench ...
@I [SIM-12] Generating RTL test bench ...
Build using "D:/Xilinx/Vivado_HLS/2013.3/msys/bin/g++.exe"
Compiling gate.autob.cpp
Compiling gate.cpp
Generating cosim.sc.exe
@I [SIM-11] Starting SystemC simulation ...

SystemC 2.3.0-ASI --- Jul 4 2013 12:03:34
Copyright (c) 1996-2012 by all Contributors,
ALL RIGHTS RESERVED

Note: VCD trace timescale unit is set by user to 1.000000e-012 sec.

Info: /OSCI/SystemC: Simulation stopped by user.
@I [SIM-316] Starting C post checking ...
a=0, b=0, c=0, d=1, e=0, f=1, g=0, h=1
a=0, b=1, c=0, d=1, e=1, f=0, g=1, h=0
a=1, b=0, c=0, d=1, e=1, f=0, g=1, h=0
a=1, b=1, c=1, d=0, e=1, f=0, g=0, h=1
@I [SIM-1000] -> C/RTL co-simulation finished: PASS ***
@I [LIC-101] Checked in feature [VIVADO_HLS]

```

图 5.79 C 仿真结果图

(1) 在 Windows 7 操作系统主界面下,选择开始→所有程序→Xilinx Design Tools →Vivado 2013. 3→Vivado 2013. 3,或者在桌面系统上双击 Vivado 2013. 3 图标,打开 Vivado 集成设计工具。

(2) 在 Vivado 主界面主菜单下,选择 File→New Project…。

(3) 出现 New Project-Create a New Vivado Project(新工程-创建一个新 Vivado 工程)对话框界面。

(4) 单击 Next 按钮。

(5) 如图 5.80 所示,出现 New Project-Project Name(新工程-工程名字)对话框界面,按图中设置参数(用户可以根据自己的情况确定)。

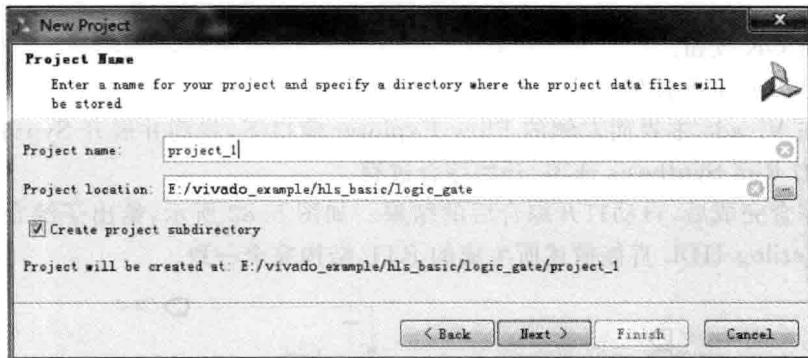


图 5.80 定位工程路径和工程名字

(6) 出现 New Project-Project Type(新工程-工程类型)对话框界面,不修改任何参数。

(7) 单击 Next 按钮。

(8) 出现 New Project-Default Part(新工程-默认器件)对话框界面。在该界面中选择 xc7a100tcsg324-1 器件。

(9) 单击 Next 按钮。

(10) 出现 New Project-New Project Summary(新工程-新工程总结)对话框界面,单击 Finish 按钮。

(11) 如图 5.81 所示,在 Project Manager 界面内的 Sources 窗口下,找到 Design Sources 并单击右键,出现浮动菜单。在浮动菜单内,选择 Add Sources…选项。

(12) 出现 Add Sources(添加源文件)对话框界面,在该界面中,选择 Add or Create Design Sources(添加或者创建设计源文件)选项。

(13) 单击 Next 按钮。

(14) 出现 Add Sources-Add or Create Design Sources(添加源文件-添加或者创建源文件)对话框界面。在该对话框界面中,按下面步骤操作:

① 单击 Add Files 按钮。

② 出现 Add Source Files(添加源文件)对话框界面。将路径定位到

E:\vivado\_example\hls\_basic\logic\_gate\gate.prj\solution\syn\verilog\

在该路径下找到并打开 gate.v 文件。

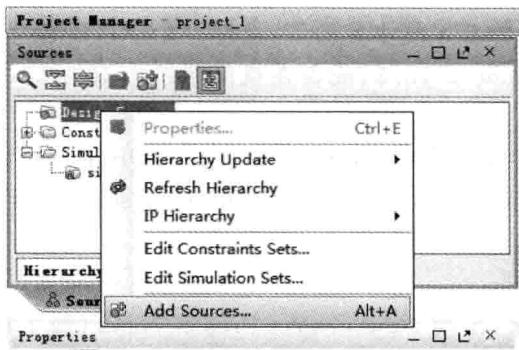


图 5.81 添加新文件入口界面

③ 单击 OK 按钮。

(15) 单击 Finish 按钮。

(16) 在 Vivado 主界面左侧的 Flow Explorer 窗口下, 找到并展开 Synthesis。在展开项中, 单击 Run Synthesis 选项, 开始综合过程。

(17) 综合完成后, 自动打开综合后的结果。如图 5.82 所示, 给出了综合后的结构, 该结构和 Verilog HDL 直接描述所生成的 RTL 结构完全一致。

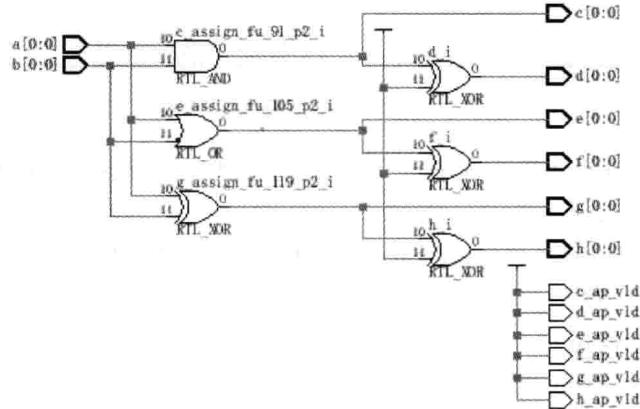


图 5.82 综合后的结构

(18) 如图 5.83 所示, 在 Project Manager 界面的 Sources 窗口下, 找到并选中 Simulation Sources 条目。单击右键, 出现浮动菜单。在浮动菜单内, 选择 Add Sources... 选项。

(19) 出现 Add Sources(添加源文件)窗口对话框界面。在该界面中, 选择 Add or Create Simulation Sources(添加或者创建仿真源文件)选项。

(20) 单击 Next 按钮。

(21) 出现 Add or Create Simulation Sources(添加或者创建仿真文件)对话框界面。在该界面中, 单击 **Create File...** 按钮。

(22) 如图 5.84 所示, 出现 Create Source File(创建源文件)对话框界面。在该界面中, 按下面参数设置:

① File Type: Verilog;

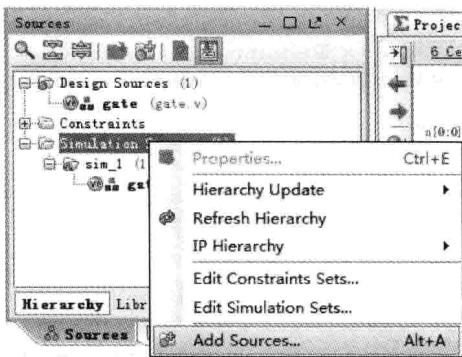


图 5.83 添加新仿真文件入口

- ② File name: test\_gate;
- ③ File location: Local to Project;
- ④ 单击 OK 按钮。

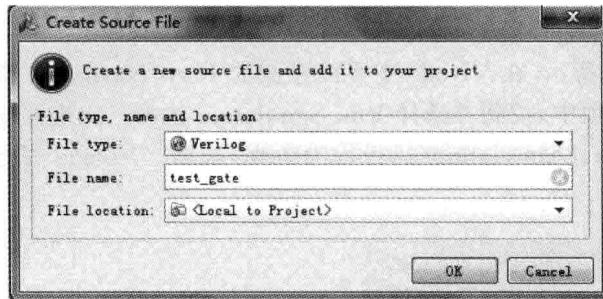


图 5.84 创建新仿真文件入口

- (23) 返回到添加或者创建仿真文件对话框界面，在该界面中单击 Finish 按钮。
- (24) 出现 Define Module(定义模块)对话框界面。单击 OK 按钮。
- (25) 如图 5.85 所示，在 Sources 窗口下，展开 Simulation Sources 选项。在展开项中，找到并展开 Sim\_1。在展开项中，找到并双击 test\_gate.v 文件。

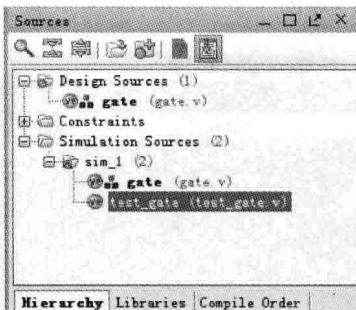


图 5.85 test\_gate.v 文件入口

- (26) 如图 5.86 所示，在 test\_gate.v 文件中输入下面的代码。
- (27) 保存 test\_gate.v 文件。

```

22
23 module test_gate;
24 reg a;
25 reg b;
26 gate Inst_gate(a, b, c, c_sp_vld, d, d_sp_vld, e, e_sp_vld, f, f_sp_vld, g, g_sp_vld, h, h_sp_vld);
27 always
28 begin
29   a=0;
30   b=0;
31   #100;
32   a=1;
33   b=1;
34   #100;
35   a=1;
36   b=0;
37   #100;
38   a=1;
39   b=1;
40   #100;
41 end

```

图 5.86 在 test\_gate.v 文件中添加代码

(28) 如图 5.87 所示,在 Vivado 主界面左侧的 Flow Navigator 窗口下,选择并展开 Simulation。在展开项中,找到并选择 Run Simulation 选项。出现浮动菜单,在浮动菜单中选择 Run Behavioral Simulation(运行行为仿真)选项。Vivado 开始运行对 gate 的行为仿真过程。

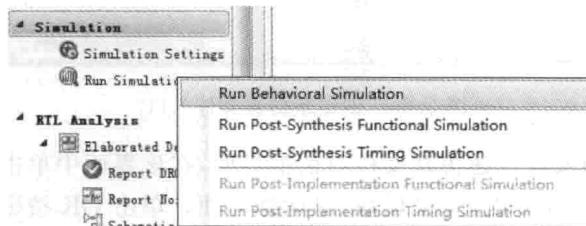


图 5.87 启动行为仿真入口

(29) 等待行为仿真结束后,如图 5.88 所示,给出仿真后的波形图。

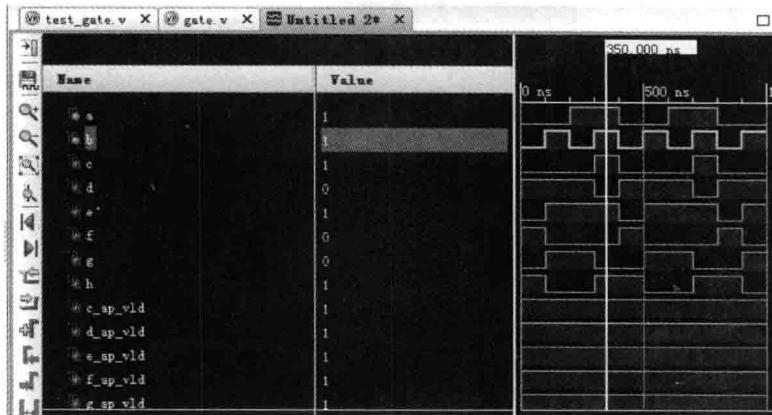


图 5.88 行为仿真后的波形界面

### 5.7.2 基于 HLS 实现时序逻辑

本部分内容包含：设计时序逻辑模型、设计综合、查看生成的数据处理图、运行 RTL 级仿真等内容。

#### 1. 设计时序逻辑模型

下面将在 Vivado HLS 中创建一个 8 位循环左移移位寄存器的模型。在 HLS 中实现 8 位循环左移移位寄存器的步骤主要包括：

- (1) 按前面的方法打开 Vivado HLS。
- (2) 出现 Vivado HLS 设计主界面。在主界面主菜单下，选择 File→New Project…，进入创建新工程设计向导界面。
- (3) 出现 New Vivado HLS Project-Project Configuration(新 Vivado HLS 工程-工程配置)向导界面。在该界面内按如下设置参数：
  - ① Project name: shifter.prj;
  - ② Location: E:\vivado\_example\hls\_basic\shifter8；
  - ③ 单击 Next 按钮。
- (4) 出现 New Vivado HLS Project-Add/Remove Files(新 Vivado HLS 工程-添加/删除源文件)界面，在该界面下按如下参数设置：
  - ① 单击 New File…按钮。按下面步骤操作：  
出现另存为文件对话框界面，输入文件名 shifter.c，单击保存按钮。
  - ② 在 Top Function 右侧输入顶层函数的名字 shifter。
  - ③ 单击 Next 按钮。
- (5) 出现 Add/Remove Files-Add/remove C-based testbench files(design test)(添加/删除文件-添加/删除基于 C 的测试平台文件(设计测试))界面，不修改任何参数设置，单击 Next 按钮。
- (6) 出现 Solution Configuration-Create Vivado HLS solution for selected technology(解决方案配置-创建用于所选择技术的 Vivado HLS 解决方案)对话框界面，在该对话框中，单击  按钮。按下面步骤操作：
  - ① 出现选择器件界面，在该界面中选择 xc7a100tcsg324-1；
  - ② 单击 OK 按钮。
  - ③ 单击 Finish 按钮。
- (8) 在 Vivado HLS 主界面左侧的 Explorer 窗口下，出现工程设计文件目录列表。在该窗口下，展开 Souce。在展开项中，选择 shifter.c 文件。双击文件名，打开 shifter.c 文件。
- (9) 如图 5.89 所示，输入 C 描述代码，并保存该设计文件。
- (10) 添加 shifter.h 头文件。添加 shifter.h 头文件的步骤主要包括：
  - ① 在 Vivado HLS 主界面主菜单下，选择 File→New File…；
  - ② 出现“另存为”对话框界面，在该界面中输入文件名 shifter.h；
  - ③ 单击“保存”按钮。HLS 工具将打开 shifter.h 文件。

```

1 #include "shifter.h"
2 void shifter(bit *x0,bit *x1,bit *x2,bit *x3,bit *x4,bit *x5,bit *x6,bit *x7)
3 {
4     int i;
5     bit x[8]={0,0,0,0,0,0,1};      //setup initial value of x[8]
6     bit tmp1=0,tmp2=0;
7     *x0=0,*x1=0,*x2=0,*x3=0,*x4=0,*x5=0,*x6=0,*x7=1;    //setup initial value of output port
8
9 while(1)
10 {
11     tmp1=x[7];
12     shifter_label0:for(i=7;i>0;i--)      // finish shift operation
13         x[i]=x[i-1];
14     x[0]=tmp1;
15     *x0=x[0];
16     *x1=x[1];
17     *x2=x[2];
18     *x3=x[3];
19     *x4=x[4];
20     *x5=x[5];
21     *x6=x[6];
22     *x7=x[7];
23 }
24
25 }

```

图 5.89 shifter.c 文件源代码

(11) 如图 5.90 所示,输入设计代码,并保存设计代码。

## 2. 设计综合

本节将通过添加用户命令对 HLS 综合过程进行优化。添加用户命令的步骤主要包括:

- (1) 打开 shifter.c 文件。
- (2) 在打开 shifter.c 文件的右侧窗口内,选择 Directive 标签。
- (3) 如图 5.91 所示,在 Directive 界面下,选择 shifter\_label0。单击右键,出现浮动菜单。在浮动菜单内,选择 Insert Directive... 选项。

```

#ifndef _SHIFTER_
#define _SHIFTER_
#include "ap_cint.h"
typedef int1 bit;
#endif

```

图 5.90 gate.h 文件源代码

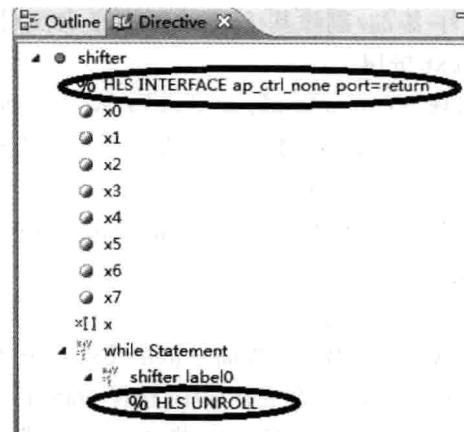


图 5.91 用户实现策略的设置

- (4) 打开 Vivado HLS Directive Editor(Vivado HLS 命令编辑器)界面,在该界面窗口下,按下面设置参数:

① Directive 右侧的下拉框中选择 UNROLL；

② 单击 OK 按钮。

(5) 如图 5.91 所示,在 Directive 界面下,选择 shifter,单击右键出现浮动菜单,选择 Insert Directive...选项。打开 Vivado HLS Directive Editor 界面,在该界面窗口下,按下设置参数:

① 在 Directive 右侧的下拉框中选择 INTERFACE;

② 在 mode(optional): ap\_ctrl\_none;

③ 单击 OK 按钮。

(6) 在 HLS 主界面主菜单下选择 Solution→Run C Synthesis→Active Solution,开始综合的过程;或者在主界面的工具栏中单击 按钮,启动综合过程。

(7) 综合完成后,自动打开综合报告。

思考题 5.8: 根据图 5.92 给出的性能报告,分析为什么在该报告的 Latency 项中出现“?”?

思考题 5.9: 根据图 5.93 给出的信息,说明该设计的资源使用情况。

The screenshot shows two parts of the synthesis report. On the left is the 'Latency (clock cycles)' section, which includes a summary table for various components like Expression, FIFO, Instance, Memory, Multiplexer, Register, and ShiftMemory. The 'Register' row shows a utilization of 9. Below this is a detailed latency table for each component, with all values marked as '?'. On the right is the 'Utilization Estimates' section, which provides a summary of resource usage across BRAM\_18K, DSP48E, FF, and LUT categories. The total utilization for each category is 0.

Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	-	-	-
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	9	-
ShiftMemory	-	-	-	-
<b>Total</b>	<b>0</b>	<b>0</b>	<b>9</b>	<b>0</b>
<b>Available</b>	<b>270</b>	<b>240</b>	<b>126800</b>	<b>63400</b>
<b>Utilization (%)</b>	<b>0</b>	<b>0</b>	<b>~0</b>	<b>0</b>

图 5.92 综合后报告中的 Latency 选项

图 5.93 综合后报告中的资源使用情况

思考题 5.10: 根据图 5.94 所示,查看该设计所生成的端口。

### 3. 查看生成的数据处理图

本节将查看生成的数据处理图,了解 HLS 工具如何把 C 模型转换成 FPGA 上的操作数据流。查看生成数据处理图的步骤主要包括:

(1) 在 Vivado HLS 主界面主菜单下,选择 Solution→Open Analysis Perspective,或者单击 按钮。

(2) 打开如图 5.95 所示的数据调度图。在 C0 周期内,共进行了 16 个操作。

思考题 5.11: 仔细查看数据流图,理解和掌握数据流图的操作信息。

注:

① 将鼠标放到数据流中的每个操作时,会给出每个操作的详细信息;

② 在查看完调度图后,单击 按钮,准备进行仿真。

Interface						
Summary						
RTL Ports	Dir	Bits	Protocol	Source Object	C Type	
ap_clk	in	1	ap_ctrl_none	shifter	return value	
ap_rst	in	1	ap_ctrl_none	shifter	return value	
x0	out	1	ap_vld	x0	pointer	
x0_ap_vld	out	1	ap_vld	x0	pointer	
x1	out	1	ap_vld	x1	pointer	
x1_ap_vld	out	1	ap_vld	x1	pointer	
x2	out	1	ap_vld	x2	pointer	
x2_ap_vld	out	1	ap_vld	x2	pointer	
x3	out	1	ap_vld	x3	pointer	
x3_ap_vld	out	1	ap_vld	x3	pointer	
x4	out	1	ap_vld	x4	pointer	
x4_ap_vld	out	1	ap_vld	x4	pointer	
x5	out	1	ap_vld	x5	pointer	
x5_ap_vld	out	1	ap_vld	x5	pointer	
x6	out	1	ap_vld	x6	pointer	
x6_ap_vld	out	1	ap_vld	x6	pointer	
x7	out	1	ap_vld	x7	pointer	
x7_ap_vld	out	1	ap_vld	x7	pointer	

图 5.94 综合后报告中的端口使用情况

Current Module : shifter			
	Operation\Control Step	C0	C1
1	node_19(write)		
2	node_20(write)		
3	node_21(write)		
4	node_22(write)		
5	node_23(write)		
6	node_24(write)		
7	node_25(write)		
8	Loop 1		
9	node_37(write)		
10	node_38(write)		
11	node_39(write)		
12	node_40(write)		
13	node_41(write)		
14	node_42(write)		
15	node_43(write)		
16	node_44(write)		

图 5.95 数据操作调度图

#### 4. 运行 RTL 级仿真

本节将在 Vivado 2013.3 环境下, 实现对设计的仿真。在 Vivado 2013.3 集成环境下, 对设计进行 RTL 级仿真的步骤主要包括:

- (1) 在 Windows 7 操作系统主界面下, 选择开始→所有程序→Xilinx Design Tools

→Vivado 2013.3→Vivado 2013.3,或者在桌面系统上双击 Vivado 2013.3 图标打开 Vivado 集成设计工具。

(2) 在 Vivado 主界面主菜单下,选择 File→New Project…。

(3) 出现 New Project-Create a New Vivado Project(新工程-创建一个新 Vivado 工程)对话框界面。

(4) 单击 Next 按钮。

(5) 出现 New Project-Project Name(新工程-工程名字)对话框界面,按下面设置参数:

① Project name: project\_1;

② Project location: E:\vivado\_example\hls\_basic\shifter8。

注: 用户可以根据自己的情况确定。

(6) 出现 New Project-Project Type(新工程-工程类型)对话框界面,不修改任何参数。

(7) 单击 Next 按钮。

(8) 出现 New Project-Default Part(新工程-默认器件)对话框界面。在该界面中选择 xc7a100tcsg324-1 器件。

(9) 单击 Next 按钮。

(10) 出现 New Project-New Project Summary(新工程-新工程总结)对话框界面,单击 Finish 按钮。

(11) 在 Project Manager 界面的 Sources 窗口下,找到并选中 Design Sources。单击右键,出现浮动菜单。在浮动菜单内,选择 Add Sources…选项。

(12) 出现 Add Sources(添加源文件)对话框界面,在该对话框界面中,选择 Add or Create Design Sources 选项。

(13) 单击 Next 按钮。

(14) 出现 Add Sources-Add or Create Design Sources(添加源文件-添加或者创建源文件)对话框界面。在该界面中按下面步骤操作:

① 单击 Add Files 按钮。

② 出现 Add Source Files(添加源文件)对话框界面。将路径定位到

E:\vivado\_example\hls\_basic\shifter8\shifter.prj\solution\syn\verilog\

在该路径下找到并打开 shifter.v 文件。

③ 单击 OK 按钮。

(15) 单击 Finish 按钮。

(16) 在 Vivado 主界面左侧的 Flow Navigator 窗口下,找到并展开 Synthesis。在展开项中,单击 Run Synthesis 选项,开始综合过程。

(17) 综合完成后,自动打开综合后的结果。

思考题 5.12: 请分析综合后生成的 RTL 结构是否和 Verilog HDL 直接描述的结构一致。

(18) 在 Project Manager 界面的 Sources 标签窗口下,找到并选中 Simulation Sources。单击右键,出现浮动菜单。在浮动菜单内,选择 Add Sources…选项。

(19) 出现 Add Sources(添加源文件)窗口对话框界面。在该对话框界面中,选择 Add or Create Simulation Sources(添加或者创建仿真源文件)选项。

(20) 单击 Next 按钮。

(21) 出现 Add or Create Simulation Sources(添加或者创建仿真文件)对话框界面。在该界面中,单击 **Create File...** 按钮。

(22) 出现 Create Source File(创建源文件)对话框界面。在该界面中,按下面设置参数:

① File Type: Verilog;

② File name: test\_shifter;

③ File location: Local to Project;

④ 单击 OK 按钮。

(23) 返回到添加或者创建仿真文件对话框界面,在该界面中单击 Finish 按钮。

(24) 出现 Define Module(定义模块)对话框界面。在该界面内,单击 OK 按钮。

(25) 在 Sources 窗口下,展开 Simulation Sources 选项。在展开项中,找到并展开 Sim\_1。在展开项中,找到并双击 test\_shifter.v 文件,打开 test\_shifter.v 文件。

(26) 如图 5.96 所示,输入下面的代码。

```

Project Summary × Device × shifter.v × Schematic × Schematic (2) × t
E:/fpga_dsp_example/hls_basic/shifter8/project_1/project_1.srcs/sim_1/new/test_shifter.v
21
22
23 module test_shifter;
24 reg ap_clk;
25 reg ap_rst;
26 shifter Inst_shifter(ap_clk,ap_rst,x0,x0_ap_vld,x1,x1_ap_vld,x2,x2_ap_vld,x3,x3_ap_vld,
27                                     x4,x4_ap_vld,x5,x5_ap_vld,x6,x6_ap_vld,x7,x7_ap_vld);
28 initial
29 begin
30 ap_rst=1;
31 #100;
32 ap_rst=0;
33 end
34 always
35 begin
36 ap_clk=1;
37 #10;
38 ap_clk=0;
39 #10;
40 end
41
42 endmodule

```

图 5.96 在 test\_shifter.v 文件中添加代码

(27) 保存 test\_gate.v 文件。

(28) 在 Vivado 主界面左侧的 Flow Navigator 窗口下,选择并展开 Simulation。在展开项中,找到并选择 Run Simulation 选项。出现浮动菜单,在浮动菜单中选择 Run Behavioral Simulation(运行行为仿真)选项。Vivado 开始对 shifter 进行行为仿真。

(29) 等待行为仿真结束后,如图 5.97 所示,出现仿真结果的波形。

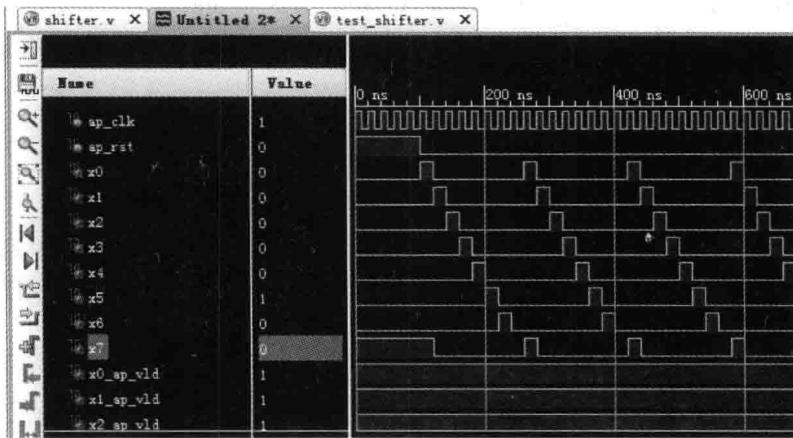


图 5.97 行为仿真后的波形界面

### 5.7.3 基于 HLS 实现矩阵相乘

本部分内容包含：设计矩阵相乘模型、添加 C 测试文件、运行和调试 C 工程、设计综合查看生成的数据处理图、设计优化、对优化前的设计运行 RTL 级仿真、对优化后的设计运行 RTL 级仿真等内容。

#### 1. 设计矩阵相乘模型

下面将在 Vivado HLS 中创建一个模型，该模型实现  $A_{3 \times 3}$  和  $B_{3 \times 3}$  相乘运算。在 HLS 中实现两个矩阵运算的步骤主要包括：

- (1) 打开 Vivado HLS 软件设计工具。
- (2) 出现 Vivado HLS 设计主界面。在主界面主菜单下，选择 File→New Project…，进入创建新工程设计向导界面。
- (3) 出现 New Vivado HLS Project-Project Configuration(新 Vivado HLS 工程-工程配置)向导界面。在该界面内按如下设置参数：

- ① Project name: matrx. prj;
- ② Location: E:\vivado\_example\hls\_basic\matrix;
- ③ 单击 Next 按钮。

(4) 出现 New Vivado HLS Project-Add/Remove Files(新 Vivado HLS 工程-添加/删除源文件)界面，在该界面下按如下参数设置：

- ① 单击 New File…按钮。按下面步骤操作：

出现另存为文件对话框界面。在该对话框界面中，输入文件名 matrix. c，单击保存按钮。

- ② 在 Top Function 右侧输入顶层函数的名字 matrix。

- ③ 单击 Next 按钮。

(5) 出现 Add/Remove Files-Add/remove C-based testbench files(design test)(添加/删除文件-添加/删除基于 C 的测试平台文件(设计测试))界面，不修改任何参数设置，单击 Next 按钮。

(6) 出现 Solution Configuration—Create Vivado HLS solution for selected technology(解决方案配置-创建用于所选择技术的 Vivado HLS 解决方案)对话框界面,在该对话框中,单击  按钮。

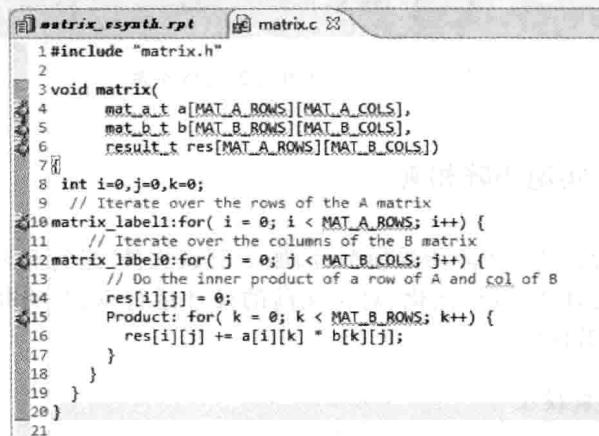
① 出现选择器件界面。在该界面中,选择 xc7a100tcsg324-1;

② 单击 OK 按钮。

(7) 单击 Finish 按钮。

(8) 在 Vivado HLS 主界面左侧的 Explorer 窗口下,出现工程设计文件目录列表。在该窗口中,展开 Sources。在展开项中,选择 matrix.c 文件。双击文件名,打开 matrix.c 文件。

(9) 如图 5.98 所示,输入 C 描述代码,并保存该设计文件。



```

1 #include "matrix.h"
2
3 void matrix(
4     mat_a_t a[MAT_A_ROWS][MAT_A_COLS],
5     mat_b_t b[MAT_B_ROWS][MAT_B_COLS],
6     result_t res[MAT_A_ROWS][MAT_B_COLS])
7 {
8     int i=0,j=0,k=0;
9     // Iterate over the rows of the A matrix
10    matrix_label1:for( i = 0; i < MAT_A_ROWS; i++) {
11        // Iterate over the columns of the B matrix
12        matrix_label2:for( j = 0; j < MAT_B_COLS; j++) {
13            // Do the inner product of a row of A and col of B
14            res[i][j] = 0;
15            Product: for( k = 0; k < MAT_B_ROWS; k++) {
16                res[i][j] += a[i][k] * b[k][j];
17            }
18        }
19    }
20}
21

```

图 5.98 matrix.c 文件源代码

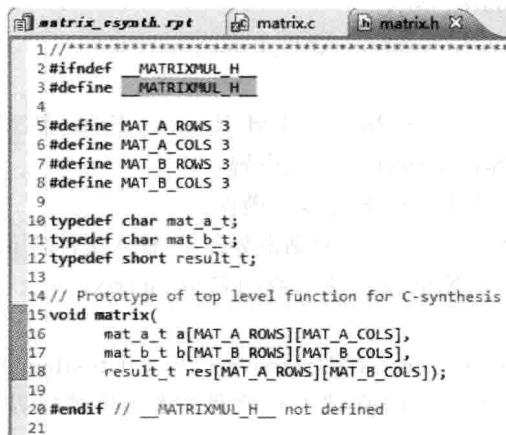
(10) 添加 matrix.h 头文件。添加 matrix.h 头文件的步骤主要包括:

① 在 Vivado HLS 主界面主菜单下,选择 File→New File…。

② 出现“另存为”对话框界面,在该界面中输入文件名 matrix.h。

③ 单击“保存”按钮。HLS 工具将打开 matrix.h 文件。

(11) 如图 5.99 所示,输入代码,并保存设计代码。



```

1 /**
2 #ifndef __MATRIXMUL_H__
3 #define __MATRIXMUL_H__
4
5 #define MAT_A_ROWS 3
6 #define MAT_A_COLS 3
7 #define MAT_B_ROWS 3
8 #define MAT_B_COLS 3
9
10 typedef char mat_a_t;
11 typedef char mat_b_t;
12 typedef short result_t;
13
14 // Prototype of top level function for C-synthesis
15 void matrix(
16     mat_a_t a[MAT_A_ROWS][MAT_A_COLS],
17     mat_b_t b[MAT_B_ROWS][MAT_B_COLS],
18     result_t res[MAT_A_ROWS][MAT_B_COLS]);
19
20#endif // __MATRIXMUL_H__ not defined
21

```

图 5.99 matrix.h 文件源代码

## 2. 添加 C 测试文件

本节将添加 C 仿真文件。添加 C 仿真文件的步骤主要包括：

- (1) 在 Vivado HLS 主界面左侧的 Explorer 窗口下,找到并选择 TestBench 条目,单击右键出现浮动菜单。在浮动菜单内,选择 New File...选项。
- (2) 出现“另存为”对话框界面。在该界面中,输入文件名 test\_matrix.c。
- (3) 单击“保存”按钮。HLS 工具将自动打开 test\_matrix.c 文件。
- (4) 如图 5.100 所示,添加下面的测试代码。
- (5) 保存该测试文件。



```

matrix.c      matrix.h      test_matrix.c
1 int main()
2 {
3
4     int i=0,j=0;
5     char in_mat_a[3][3] = {
6         {11, 12, 13},
7         {14, 15, 16},
8         {17, 18 ,19}
9     };
10    char in_mat_b[3][3] = {
11        {21, 22, 23},
12        {24, 25, 26},
13        {27, 28, 29}
14    };
15    short hw_result[3][3];
16
17    matrix(in_mat_a, in_mat_b, hw_result);
18    for (i = 0; i < 3; i++)
19    {
20        for (j = 0; j < 3; j++)
21            printf("%d ",hw_result[i][j]);
22            printf("\n");
23    }
24}
25

```

图 5.100 test\_matrix.c 源代码

## 3. 运行和调试 C 工程

本节将运行 C 和调试工程。运行调试 C 工程的步骤主要包括：

- (1) 在 HLS 主界面主菜单下,选择 Project→Run C Simulation。
- (2) 如图 5.101 所示,打开 C Simulation Dialog(C 仿真对话框)界面。在该界面的 Options 标题栏下选中 Debug、Build Only、Clean Build 前面的复选框。

下面对这些选项进行说明：

- ① Debug: 这个在默认模式下,编译 C。然后,打开调试界面。
- ② Build Only: 将编译 C 代码,但不运行仿真。
- ③ Clean Build: 删除当前工程下所有存在的可执行文件和目标文件。
- ④ Optimized Compile: 在 Release 模式下,编译 C(没有带任何调试信息)。在编译 C 的过程中,使用更高级的优化。
- (3) 单击 OK 按钮。
- (4) 如图 5.102 所示,单击 test\_matrix.c 标签,打开该文件。在该文件窗口下,分别

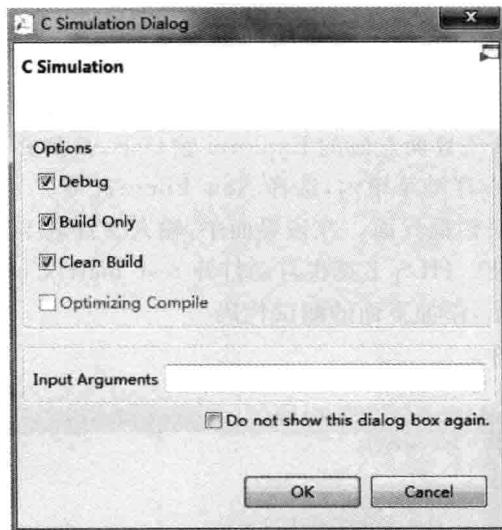


图 5.101 C 仿真对话框界面

在标号为 18 和 24 的行前的空白处双击,可以看到出现两个小蓝点,表示为这两行分别设置了断点,用于对该程序进行调试。

```

matrix.c [matrix.h] [test_matrix.c] [matrix_cosine.rj]
13     {27, 28, 29}
14 };
15 short hw_result[3][3];
16
17 matrix(in_mat_a, in_mat_b, hw_result);
18 for (i = 0; i < 3; i++)
19 {
20     for (j = 0; j < 3; j++)
21         printf("%d ",hw_result[i][j]);
22     printf("\n");
23 }
24

```

图 5.102 设置断点

(5) 如图 5.103 所示,在调试窗口的工具栏内,单击 Run C Simulation 按钮。启动运行 C 仿真。

(6) 再次出现 C 仿真对话框界面。

(7) 单击 OK 按钮。

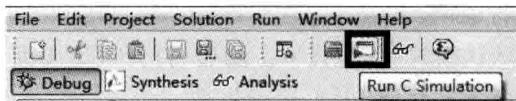


图 5.103 运行调试入口

(8) 如图 5.104 所示。单击图中的 ▶ 按钮,使程序运行到设置第二个断点的行。

(9) 如图 5.105 所示,在调试界面右上侧的窗口中,展开 hw\_result。然后,在展开项中分别展开 hw\_result[0]、hw\_result[1] 和 hw\_result[2]。可以看到矩阵相乘的运行结果。



图 5.104 控制程序的运行

Name	Type	Value
in_mat_b	char [3][3]	0x22fef6
hw_result	short int [3][3]	0x22fee4
hw_result[0]	short int [3]	0x22fee4
(0): hw_result[0][0]	short int	870
(0): hw_result[0][1]	short int	906
(0): hw_result[0][2]	short int	942
hw_result[1]	short int [3]	0x22fea
(0): hw_result[1][0]	short int	1086
(0): hw_result[1][1]	short int	1131
(0): hw_result[1][2]	short int	1176
hw_result[2]	short int [3]	0x22fef0
(0): hw_result[2][0]	short int	1302
(0): hw_result[2][1]	short int	1356
(0): hw_result[2][2]	short int	1410

图 5.105 矩阵相乘的运行结果

- (10) 单击停止按钮 ，退出调试器界面。  
(11) 单击 Synthesis 按钮，重新进入 matrix.c 文件界面窗口。

#### 4. 设计综合

本节将对设计进行综合，将 C 模型转换成 RTL 级的描述。实现设计综合的步骤主要包括：

- (1) 在 HLS 主界面主菜单下，选择 Solution→Run C Synthesis→Active Solution，开始综合的过程。或者在主界面的工具栏中单击 按钮，启动综合的过程。
- (2) 在 Console 窗口下，出现综合过程的信息。
- (3) 综合完成后，自动打开综合后的报告 matrix\_csynth.rpt 文件。下面对报告进行分析：

① 如图 5.106 所示，给出了性能信息。

思考题 5.13：如图 5.106 所示，根据给出的 Latency(延迟)和 Interval(间隔)值，说明这些值所给出的性能信息。

Performance Estimates				
Timing (ns)				
Summary				
Clock	Target	Estimated	Uncertainty	
default	10.00	7.18	1.25	

Latency (clock cycles)				
Summary				
Latency		Interval		
min	max	min	max	Type
106	106	107	107	none

图 5.106 综合后报告给出的性能信息

② 如图 5.107 所示,给出了该设计的器件利用率信息。

思考题 5.14: 请给出该设计中所消耗 FPGA 的逻辑资源的个数。

Utilization Estimates				
Summary				
Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	1	0	34
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	-	-	-	-
Multiplexer	-	-	-	22
Register	-	-	60	-
ShiftMemory	-	-	-	-
Total	0	1	60	56
Available	270	240	126800	63400
Utilization (%)	0	~0	~0	~0

图 5.107 综合后报告给出的器件利用率信息

③ 如图 5.108 所示,给出了综合后该设计给出的端口。

思考题 5.15: 请说明除了 C 模型描述的输入和输出端口外, HLS 工具又新添加了哪些新的端口。

Interface						
Summary						
RTL Ports	Dir	Bits	Protocol	Source Object	C Type	
ap_clk	in	1	ap_ctrl_hs	matrix	return value	
ap_rst	in	1	ap_ctrl_hs	matrix	return value	
ap_start	in	1	ap_ctrl_hs	matrix	return value	
ap_done	out	1	ap_ctrl_hs	matrix	return value	
ap_idle	out	1	ap_ctrl_hs	matrix	return value	
ap_ready	out	1	ap_ctrl_hs	matrix	return value	
a_address0	out	4	ap_memory	a	array	
a_ce0	out	1	ap_memory	a	array	
a_q0	in	8	ap_memory	a	array	
b_address0	out	4	ap_memory	b	array	
b_ce0	out	1	ap_memory	b	array	
b_q0	in	8	ap_memory	b	array	
res_address0	out	4	ap_memory	res	array	
res_ce0	out	1	ap_memory	res	array	
res_we0	out	1	ap_memory	res	array	
res_d0	out	16	ap_memory	res	array	

图 5.108 综合后报告给出的端口信息

## 5. 查看生成的数据处理图

本节将查看生成的数据处理图,了解 HLS 工具如何把 C 模型转换成 FPGA 上的操作数据流。查看生成的数据处理图的步骤主要包括:

(1) 在 Vivado HLS 主界面主菜单下,选择 Solution→Open Analysis Perspective; 或者单击 **Analysis** 按钮。

(2) 打开如图 5.109 所示的数据调度图。

思考题 5.16：仔细查看数据流图。理解和掌握数据流图的操作信息。

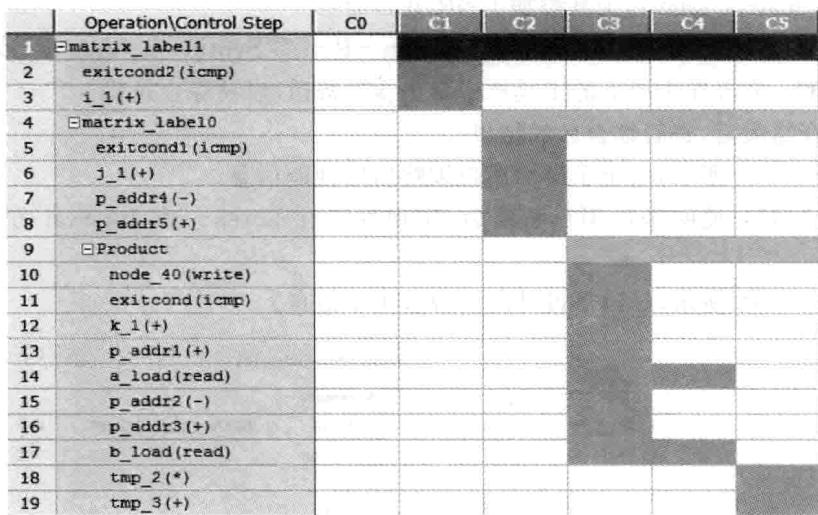


图 5.109 数据操作调度图

## 6. 设计优化

本节将通过添加用户指令对 HLS 综合过程进行优化。添加用户命令的步骤主要包括：

- (1) 在 HLS 主界面主菜单下,选择 Project→New Solution…。
- (2) 出现 Solution Wizard(解决向导)对话框界面,在该界面内:
  - ① Part Selection(器件选择)下,按前面的方法选择器件: xc7a100tcsg324-1。
  - ② 其他按默认参数设置。
  - ③ 单击 Finish 按钮。
- (3) 打开 matrix.c 文件。
- (4) 在打开 matrix.c 文件的右侧窗口内,选择 Directive 标签。
- (5) 如图 5.110 所示,分别添加下面的用户策略:
  - ① 选择 Product,为其添加 UNROLL 命令;

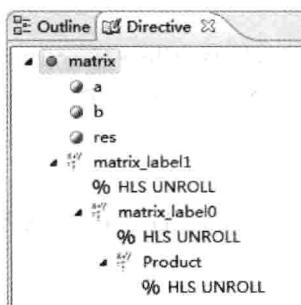


图 5.110 添加用户命令后的界面

② 选择 matrix\_label0, 为其添加 UNROLL 命令；

③ 选择 matrix\_label1, 为其添加 UNROLL 命令。

(6) 在 HLS 主界面主菜单下选择 Solution→Run C Synthesis→Active Solution, 开始综合的过程；或者在主界面的工具栏中单击 按钮, 启动综合过程。

(7) 综合结束后, 查看综合后的结果。

① 如图 5.111 所示, 给出了添加用户策略后的性能信息。

思考题 5.17: 说明添加用户策略后, 实现性能在 Latency 和 Interval 方面的改善情况。

② 如图 5.112 所示, 给出了设计的资源占用率信息。

The screenshot shows the 'Utilization Estimates' report. It includes sections for 'Summary' and 'Latency (clock cycles)'. The 'Summary' section provides a detailed breakdown of resources used, including BRAM\_18K, DSP48E, FF, and LUT counts for various components like Expression, FIFO, Instance, Memory, Multiplexer, Register, and ShiftMemory. The 'Latency (clock cycles)' section shows latency and interval values for different components, with a table for each component's min, max, and type.

Utilization Estimates					
Summary					
Name	BRAM_18K	DSP48E	FF	LUT	
Expression	-	18	0	0	
FIFO	-	-	-	-	
Instance	-	9	-	-	
Memory	-	-	-	-	
Multiplexer	-	-	-	72	
Register	-	-	532	-	
ShiftMemory	-	-	-	-	
<b>Total</b>	<b>0</b>	<b>27</b>	<b>532</b>	<b>72</b>	
Available	270	240	126800	63400	
Utilization (%)	0	11	-0	-0	

图 5.111 添加用户策略后的实现性能

图 5.112 添加用户策略后的资源占用率

③ 如图 5.113 所示, 查看设计报告中的生成端口信号。

思考题 5.18: 请分析在添加策略后, 端口有哪些变化。

## 7. 对优化前的设计运行 RTL 级仿真

本节将在 Vivado 2013.3 环境下, 对优化前的设计进行仿真。在 Vivado 2013.3 集成环境下, 对设计进行 RTL 级仿真的步骤主要包括:

(1) 在 Windows 7 操作系统主界面下, 选择开始→所有程序→Xilinx Design Tools →Vivado 2013.3→Vivado 2013.3; 或者在桌面系统上双击 Vivado 2013.3 图标, 打开 Vivado 集成设计工具。

(2) 在 Vivado 主界面主菜单下, 选择 File→New Project…。

(3) 出现 New Project-Create a New Vivado Project(新工程-创建一个新 Vivado 工程)对话框界面。

(4) 单击 Next 按钮。

(5) 出现 New Project-Project Name(新工程-工程名字)对话框界面按下面设置参数:

① Project name: project\_1;

② Project location: E:\vivado\_example\hls\_basic\matrix。

注: 用户可以根据自己的情况确定。

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_hs	matrix	return value
ap_rst	in	1	ap_ctrl_hs	matrix	return value
ap_start	in	1	ap_ctrl_hs	matrix	return value
ap_done	out	1	ap_ctrl_hs	matrix	return value
ap_idle	out	1	ap_ctrl_hs	matrix	return value
ap_ready	out	1	ap_ctrl_hs	matrix	return value
a_address0	out	4	ap_memory	a	array
a_ce0	out	1	ap_memory	a	array
a_q0	in	8	ap_memory	a	array
a_address1	out	4	ap_memory	a	array
a_ce1	out	1	ap_memory	a	array
a_q1	in	8	ap_memory	a	array
b_address0	out	4	ap_memory	b	array
b_ce0	out	1	ap_memory	b	array
b_q0	in	8	ap_memory	b	array
b_address1	out	4	ap_memory	b	array
b_ce1	out	1	ap_memory	b	array
b_q1	in	8	ap_memory	b	array
res_address0	out	4	ap_memory	res	array
res_ce0	out	1	ap_memory	res	array
res_we0	out	1	ap_memory	res	array
res_d0	out	16	ap_memory	res	array
res_address1	out	4	ap_memory	res	array
res_ce1	out	1	ap_memory	res	array
res_we1	out	1	ap_memory	res	array
res_d1	out	16	ap_memory	res	array

图 5.113 添加用户策略后的端口的变化

(6) 出现 New Project-Project Type(新工程-工程类型)对话框界面,不修改任何参数。

(7) 单击 Next 按钮。

(8) 出现 New Project-Default Part(新工程-默认器件)对话框界面。在该界面中,选择 xc7a100tcsg324-1 器件。

(9) 单击 Next 按钮。

(10) 出现 New Project-New Project Summary(新工程-新工程总结)对话框界面。在该界面下,单击 Finish 按钮。

(11) 在 Project Manager 界面的 Sources 窗口下,找到并选中 Design Sources 条目。并单击右键,出现浮动菜单。在浮动菜单内,选择 Add Sources...选项。

(12) 出现 Add Sources(添加源文件)对话框界面。在该界面中,选择 Add or Create Design Sources 选项。

(13) 单击 Next 按钮。

(14) 出现 Add Sources-Add or Create Design Sources(添加源文件-添加或者创建源文件)对话框界面。在该界面中按下面步骤操作:

① 单击 Add Files 按钮。

② 出现 Add Source Files(添加源文件)对话框界面。将路径定位到

E:\Vivado\_example\hls\_basic\matrix\matrix.prj\solution\syn\vhdl\

在该路径下找到并打开文件：matrix.vhd。

③ 单击 OK 按钮。

(15) 单击 Finish 按钮。

(16) 选择 matrix.vhd 文件，并在 Vivado 主界面左侧的 Flow Navigator 窗口下，找到并展开 Synthesis。在展开项中，单击 Run Synthesis 选项，开始综合过程。

(17) 综合完成后，自动打开综合后的结果。

(18) 在 Project Manager 界面的 Sources 窗口下，找到并选中 Simulation Sources。单击右键，出现浮动菜单。在浮动菜单内，选择 Add Sources... 选项。

(19) 出现 Add Sources(添加源文件)窗口对话框界面。在该界面中，选择 Add or Create Simulation Sources(添加或者创建仿真源文件)选项。

(20) 单击 Next 按钮。

(21) 出现 Add or Create Simulation Sources(添加或者创建仿真文件)对话框界面。在该界面中，单击 **Create File...** 按钮。

(22) 出现 Create Source File(创建源文件)对话框界面。在该界面中，按下面设置参数：

① File Type: VHDL；

② File name: matrixmul\_testbench；

③ File location: Local to Project；

④ 单击 OK 按钮。

(23) 返回到添加或者创建仿真文件对话框界面，在该界面中单击 Finish 按钮。

(24) 出现 Define Module(定义模块)对话框界面。在该界面内，单击 OK 按钮。

(25) 在 Sources 窗口下，展开 Simulation Sources 选项。在展开项中，找到并展开 Sim\_1。在展开项中，找到并双击 matrixmul\_testbench.vhd 文件，打开该文件。

(26) 输入下面的测试代码。

#### 代码清单 5-21

```

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY matrixmul_testbench IS
END matrixmul_testbench;

ARCHITECTURE behavior OF matrixmul_testbench IS
COMPONENT matrix
PORT(
    ap_clk : IN    std_logic;
    ap_rst : IN    std_logic;
    ap_start : IN    std_logic;
    ap_done : OUT   std_logic;
    ap_idle : OUT   std_logic;
    a_address0 : OUT  std_logic_vector(3 downto 0);
    a_ce0 : OUT    std_logic;
)
END COMPONENT;

```

```
a_q0 : IN      std_logic_vector(7 downto 0);
b_address0 : OUT     std_logic_vector(3 downto 0);
b_ce0 : OUT      std_logic;
b_q0 : IN      std_logic_vector(7 downto 0);
res_address0 : OUT     std_logic_vector(3 downto 0);
res_ce0 : OUT      std_logic;
res_we0 : OUT      std_logic;
res_d0 : OUT      std_logic_vector(15 downto 0)
);
END COMPONENT;
-- Inputs
signal ap_clk : std_logic := '0';
signal ap_rst : std_logic := '0';
signal ap_start : std_logic := '0';
signal a_q0 : std_logic_vector(7 downto 0) := (others => '0');
signal b_q0 : std_logic_vector(7 downto 0) := (others => '0');
-- Outputs
signal ap_done : std_logic;
signal ap_idle : std_logic;
signal a_address0 : std_logic_vector(3 downto 0);
signal a_ce0 : std_logic;
signal b_address0 : std_logic_vector(3 downto 0);
signal b_ce0 : std_logic;
signal res_address0 : std_logic_vector(3 downto 0);
signal res_ce0 : std_logic;
signal res_we0 : std_logic;
signal res_d0 : std_logic_vector(15 downto 0);
-- Clock period definitions
constant ap_clk_period : time := 10 ns;
BEGIN
    -- Instantiate the Unit Under Test (UUT)
    uut: matrix PORT MAP (
        ap_clk => ap_clk,
        ap_rst => ap_rst,
        ap_start => ap_start,
        ap_done => ap_done,
        ap_idle => ap_idle,
        a_address0 => a_address0,
        a_ce0 => a_ce0,
        a_q0 => a_q0,
        b_address0 => b_address0,
        b_ce0 => b_ce0,
        b_q0 => b_q0,
        res_address0 => res_address0,
        res_ce0 => res_ce0,
        res_we0 => res_we0,
        res_d0 => res_d0
    );

```

```

);
-- Clock process definitions
ap_clk_process :process
begin
    ap_clk <= '0';
    wait for ap_clk_period/2;
    ap_clk <= '1';
    wait for ap_clk_period/2;
end process;

-- Stimulus process
stim_proc: process
begin
    -- hold reset state for 100 ns.
    wait for 100 ns;
    ap_rst <= '1';
    wait for ap_clk_period * 10;
    ap_rst <= '0';
    -- insert stimulus here
    wait for ap_clk_period;

-- row 1
--     a_q0 <= x"0b";
--     b_q0 <= x"15";
--     ap_start <= '1';
--     wait for ap_clk_period;
--     ap_start <= '0';
--     a_q0 <= x"0b";
--     b_q0 <= x"15";
--     wait for ap_clk_period * 5;
--     a_q0 <= x"0c";
--     b_q0 <= x"18";
--     wait for ap_clk_period * 3;
--     a_q0 <= x"0d";
--     b_q0 <= x"1b";
--     wait for ap_clk_period * 3;
--     wait for ap_clk_period; -- wait for result to come out
if (res_d0 = x"0366") then
    report "PASS: result[1][1] = 870";
else
    report "FAIL at result[1][1]";
end if;
a_q0 <= x"0b";
b_q0 <= x"16";
wait for ap_clk_period * 4;
a_q0 <= x"0c";
b_q0 <= x"19";
wait for ap_clk_period * 3;
a_q0 <= x"0d";
b_q0 <= x"1c";
wait for ap_clk_period * 3;
wait for ap_clk_period; -- wait for result to come out

```

```
if (res_d0 = x"038a") then
    report "PASS: result[1][2] = 906";
else
    report "FAIL at result[1][2]";
end if;
a_q0 <= x"0b";
b_q0 <= x"17";
wait for ap_clk_period * 4;
a_q0 <= x"0c";
b_q0 <= x"1a";
wait for ap_clk_period * 3;
a_q0 <= x"0d";
b_q0 <= x"1d";
wait for ap_clk_period * 3;
wait for ap_clk_period; -- wait for result to come out
if (res_d0 = x"03ae") then
    report "PASS: result[1][3] = 942";
else
    report "FAIL at result[1][3]";
end if;
wait for ap_clk_period * 2;
-- row 2
a_q0 <= x"0e";
b_q0 <= x"15";
wait for ap_clk_period * 4;
a_q0 <= x"0f";
b_q0 <= x"18";
wait for ap_clk_period * 3;
a_q0 <= x"10";
b_q0 <= x"1b";
wait for ap_clk_period * 3;
wait for ap_clk_period; -- wait for result to come out
if (res_d0 = x"043e") then
    report "PASS: result[2][1] = 1086";
else
    report "FAIL at result[2][1]";
end if;
a_q0 <= x"0e";
b_q0 <= x"16";
wait for ap_clk_period * 4;
a_q0 <= x"0f";
b_q0 <= x"19";
wait for ap_clk_period * 3;
a_q0 <= x"10";
b_q0 <= x"1c";
wait for ap_clk_period * 3;
wait for ap_clk_period; -- wait for result to come out
if (res_d0 = x"046b") then
    report "PASS: result[2][2] = 1131";
else
    report "FAIL at result[2][2];
```

```

    end if;
    a_q0 <= x"0e";
    b_q0 <= x"17";
    wait for ap_clk_period * 4;
    a_q0 <= x"0f";
    b_q0 <= x"1a";
    wait for ap_clk_period * 3;
    a_q0 <= x"10";
    b_q0 <= x"1d";
    wait for ap_clk_period * 3;
    wait for ap_clk_period; -- wait for result to come out
    if (res_d0 = x"0498") then
        report "PASS: result[2][3] = 1176";
    else
        report "FAIL at result[2][3]";
    end if;
    wait for ap_clk_period * 2;
-- row 3
    a_q0 <= x"11";
    b_q0 <= x"15";
    wait for ap_clk_period * 4;
    a_q0 <= x"12";
    b_q0 <= x"18";
    wait for ap_clk_period * 3;
    a_q0 <= x"13";
    b_q0 <= x"1b";
    wait for ap_clk_period * 3;
    wait for ap_clk_period; -- wait for result to come out
    if (res_d0 = x"0516") then
        report "PASS: result[3][1] = 1302";
    else
        report "FAIL at result[3][1]";
    end if;
    a_q0 <= x"11";
    b_q0 <= x"16";
    wait for ap_clk_period * 4;
    a_q0 <= x"12";
    b_q0 <= x"19";
    wait for ap_clk_period * 3;
    a_q0 <= x"13";
    b_q0 <= x"1c";
    wait for ap_clk_period * 3;
    a_q0 <= x"11";
    b_q0 <= x"17";
    wait for ap_clk_period; -- wait for result to come out
    if (res_d0 = x"054c") then
        report "PASS: result[2][2] = 1356";
    else
        report "FAIL at result[3][2]";
    end if;

```

```

    wait for ap_clk_period * 4;
    a_q0 <= x"12";
    b_q0 <= x"1a";
    wait for ap_clk_period * 3;
    a_q0 <= x"13";
    b_q0 <= x"1d";
    wait for ap_clk_period * 3;
    wait for ap_clk_period; -- wait for result to come out
    if (res_d0 = x"0582") then
        report "PASS: result[3][3] = 1410";
    else
        report "FAIL at result[3][3]";
    end if;
    wait for ap_clk_period * 5;

    wait;
end process;

END;

```

(27) 保存 matrixmul\_testbench.vhd 文件。

(28) 选中 matrixmul\_testbench.vhd 文件，并且在 Vivado 主界面左侧的 Flow Navigator 窗口下，选择并展开 Simulation。在展开项中，找到并选择 Run Simulation 选项。出现浮动菜单，在浮动菜单中选择 Run Behavioral Simulation(运行行为仿真)选项。Vivado 开始对 matrix 进行行为仿真。

(29) 等待行为仿真结束后，如图 5.114 所示，出现仿真结果的波形。

思考题 5.19：仔细分析图 5.114 所示的行为仿真结果，理解和掌握其时序之间的关系。

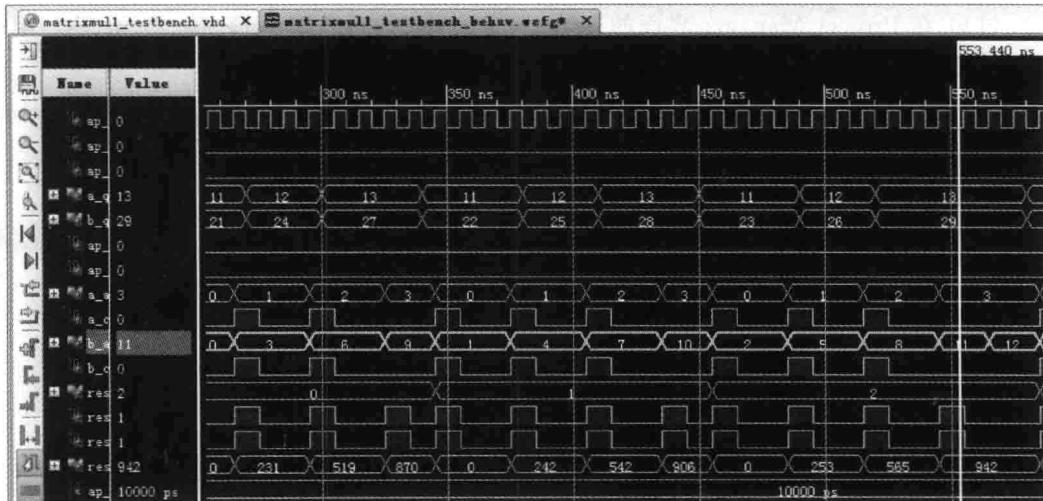


图 5.114 行为仿真后的波形界面

## 8. 对优化后的设计运行 RTL 级仿真

本节将在 Vivado 2013.3 环境下,对优化后的设计进行仿真。在 Vivado 2013.3 集成环境下,对设计进行 RTL 级仿真的步骤主要包括:

(1) 在 Windows 7 操作系统主界面下,选择开始→所有程序→Xilinx Design Tools →Vivado 2013.3→Vivado 2013.3; 或者在桌面系统上双击 Vivado 2013.3 图标,打开 Vivado 集成设计工具。

(2) 在 Vivado 主界面主菜单下,选择 File→New Project…。

(3) 出现 New Project-Create a New Vivado Project(新工程-创建一个新 Vivado 工程)对话框界面。

(4) 单击 Next 按钮。

(5) 出现 New Project-Project Name(新工程-工程名字)对话框界面。在该界面下,按下面设置参数:

① Project name: project\_2;

② Project location: E:\vivado\_example\hls\_basic\matrix;

注: 用户可以根据自己的情况确定。

(6) 出现 New Project-Project Type(新工程-工程类型)对话框界面,不修改任何参数。

(7) 单击 Next 按钮。

(8) 出现 New Project-Default Part(新工程-默认器件)对话框界面。在该界面中,选择 xc7a100tcsg324-1 器件。

(9) 单击 Next 按钮。

(10) 出现 New Project-New Project Summary(新工程-新工程总结)对话框界面。在该界面下,单击 Finish 按钮。

(11) 在 Project Manager 界面的 Sources 窗口下,找到并选中 Design Sources。单击右键,出现浮动菜单。在浮动菜单内,选择 Add Sources…选项。

(12) 出现 Add Sources(添加源文件)对话框界面。在该界面中,选择 Add or Create Design Sources 选项。

(13) 单击 Next 按钮。

(14) 出现 Add Sources-Add or Create Design Sources(添加源文件-添加或者创建源文件)对话框界面。在该界面中按下面步骤操作:

① 单击 Add Files 按钮。

② 出现 Add Source Files(添加源文件)对话框界面。将路径定位到

E:\vivado\_example\hls\_basic\matrix\matrix.prj\solution\syn\verilog\

在该路径下找到并打开下面的文件:

matrix.v;

matrix\_mul\_8s\_8s\_16\_3.v。

③ 单击 OK 按钮。

- (15) 单击 Finsih 按钮。
- (16) 选择 matrix.v 文件，并在 Vivado 主界面左侧的 Flow Navigator 窗口下，找到并展开 Synthesis。在展开项中，单击 Run Synthesis 选项，开始综合过程。
- (17) 综合完成后，自动打开综合后的结果。
- (18) 在 Project Manager 界面的 Sources 窗口下，找到并选中 Simulation Sources。单击右键，出现浮动菜单。在浮动菜单内，选择 Add Sources... 选项。
- (19) 出现 Add Sources(添加源文件)窗口对话框界面。在该界面中，选择 Add or Create Simulation Sources(添加或者创建仿真源文件)选项。
- (20) 单击 Next 按钮。
- (21) 出现 Add or Create Simulation Sources(添加或者创建仿真文件)对话框界面。在该界面中，单击 Create File... 按钮。
- (22) 出现 Create Source File(创建源文件)对话框界面。在该界面中，按下面设置参数：
- ① File Type: Verilog;
  - ② File name: test\_matrix;
  - ③ File location: Local to Project;
  - ④ 单击 OK 按钮。
- (23) 返回到添加或者创建仿真文件对话框界面，在该界面中单击 Finish 按钮。
- (24) 出现 Define Module(定义模块)对话框界面。单击 OK 按钮。
- (25) 在 Sources 窗口下，展开 Simulation Sources 选项。在展开项中，找到并展开 Sim\_1。在展开项中，找到并双击 test\_gate.v 文件，打开该文件。
- (26) 输入下面的代码。

#### 代码清单 5-22 for 循环

```
module test_matrix;
reg ap_clk;
reg ap_rst;
reg ap_start;
reg [7:0] a_q0;
reg [7:0] b_q0;
reg [7:0] a_q1;
reg [7:0] b_q1;
wire [15:0] res_d0;
wire [15:0] res_d1;
wire [3:0] a_address0;
wire [3:0] a_address1;
wire [3:0] b_address0;
wire [3:0] b_address1;
wire [3:0] res_address0;
wire [3:0] res_address1;
parameter clk = 10;
matrix Inst_matrix( ap_clk,ap_rst,ap_start,ap_done,ap_idle,ap_ready,
                    a_address0,a_ce0,a_q0,a_address1,a_ce1,a_q1,
                    b_address0,b_ce0,b_q0,b_address1,b_ce1,b_q1,
                    res_address0,res_ce0,res_we0,res_d0,
```

```

        res_address1,res_ce1,res_we1,res_d1);

// Clock process definitions
always
begin
    ap_clk <= 1'b0;
#(clk/2);
    ap_clk <= 1'b1;
#(clk/2);
end
always
begin
    # 100;
    ap_rst <= 1'b1;
    #(clk * 10);
    ap_rst <= 1'b0;
    # 10000;
end
always
begin
    ap_start <= 1'b0;
    #(clk * 21);
    ap_start <= 1'b1;
    #(clk);
    ap_start <= 1'b0;
    # 100000;
end
always
begin
    # 220;
    a_q0 <= 8'h0d;           //a2
    a_q1 <= 8'h10;           //a5
    #(clk);
    a_q0 <= 8'h13;           //a8
    #(clk);
    a_q0 <= 8'h0c;           //a1

    #(clk);
    a_q0 <= 8'h0b;           //a0
    a_q1 <= 8'h0f;           //a4
    #(clk);
    a_q0 <= 8'h0e;           //a3
    a_q1 <= 8'h12; //a7
    #(clk);
    a_q0 <= 8'h11;           //a6
    #(clk);
end

always
begin
    # 220;

```

```

    b_q0 <= 8'h1b;           //b6
    b_q1 <= 8'h1c;           //b7
    # clk;
    b_q0 <= 8'h1d;           //b8

    # clk;
    b_q0 <= 8'h18;           //b3

    # clk;
    b_q0 <= 8'h15;           //b0
    b_q1 <= 8'h19;           //b4
    # clk;
    b_q0 <= 8'h16;           //b1
    b_q1 <= 8'h1a;           //b5
    # clk;
    b_q0 <= 8'h17;           //b2
    # clk;
end
endmodule

```

(27) 保存 test\_matrix.v 文件。

(28) 在 Vivado 主界面左侧的 Flow Navigator 窗口下,选择并展开 Simulation。在展开项中,找到并选择 Run Simulation 选项。出现浮动菜单,在浮动菜单中选择 Run Behavioral Simulation(运行行为仿真)选项。Vivado 对 matrix 进行行为仿真。

(29) 等待行为仿真结束后,如图 5.115 所示,出现仿真结果的波形图。

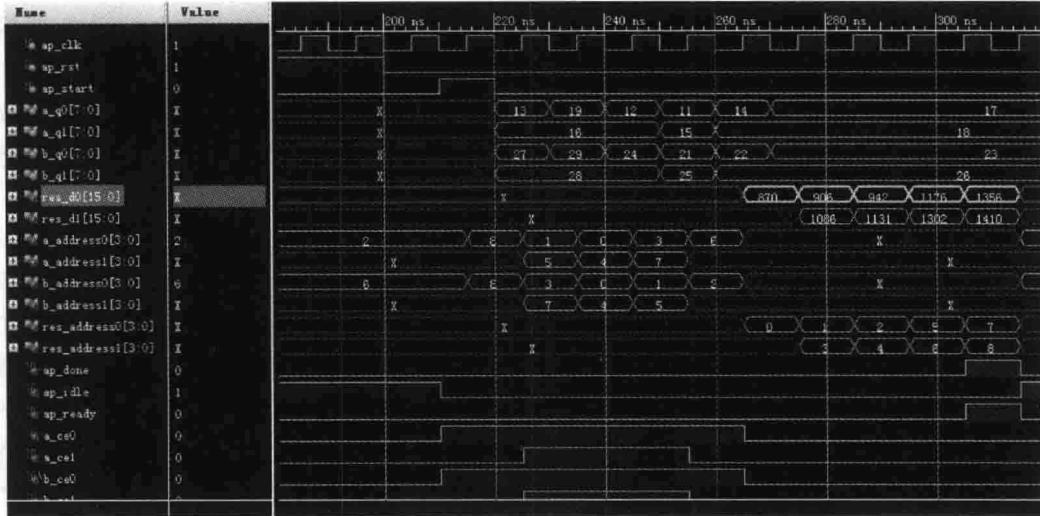


图 5.115 行为仿真后的波形界面

# 第6章 System Generator设计流程

基于 FPGA 实现数字信号处理引起越来越多的重视,这是由于 FPGA 和数字信号处理器相比,具有更高的处理能力。随着高级建模工具 System Generator 的出现,使得越来越多的高性能信号处理可以很容易地在 FPGA 上实现。

本章首先简要介绍了 Vivado 集成开发环境下 System Generator 工具的功能,然后通过一些具体的例子说明 System Generator 的设计流程。对于更详细的基于模型的数字信号处理的实现算法,可以参考《Xilinx FPGA 数字信号处理权威指南》(清华大学出版社,2014)。

注:本章内容需要 Vivado2013.3 和 MATLAB R2013a 集成开发环境。

## 6.1 FPGA 信号处理方法

在实现数字信号处理方面,使用 FPGA 比使用数字信号处理器 (Digital Signal Processor, DSP) 具有更多的优势,集中表现在速度和性能方面。FPGA 是高性能的数据处理器件,其数字信号处理性能取决于处理数据的并行结构。而微处理器或数字信号处理器的处理性能与处理器所运行的频率密切相关。下面通过一个例子来说明这个问题。

如图 6.1(a) 所示,传统的微处理器和数字信号处理器在实现 256 阶 FIR 滤波器时,采用了顺序的处理结构。这样,就限制了数据的吞吐量。这个结构基于共享时间的 MAC 单元,通过高频率的驱动造成系统竞争力的困难。在这个结构中,每个数据采样执行 256 个乘法累加 (Multiply Accumulate, MAC) 操作。每 256 个时钟周期才能输出一个结果。如图 6.1(b) 所示,当采用 FPGA 的并行处理结构时,得到最大的数据吞吐量。这种结构支持任何的并行度,可以在性能和成本之间进行很好的权衡。在这个结构中,每个数据采样同时执行 256 个乘法累加 (MAC) 操作,在每个时钟周期都可以输出一个结果。

但是,以前使用 FPGA 进行数字信号处理比较困难,这是由于使用 HDL 语言描述数字信号处理模型非常麻烦。但是,随着高级建模工具 System Generator 的出现,使得为 FPGA 建立数字信号处理数学模型变得非常简单。综上所述,越来越多的高性能的信号处理都采用

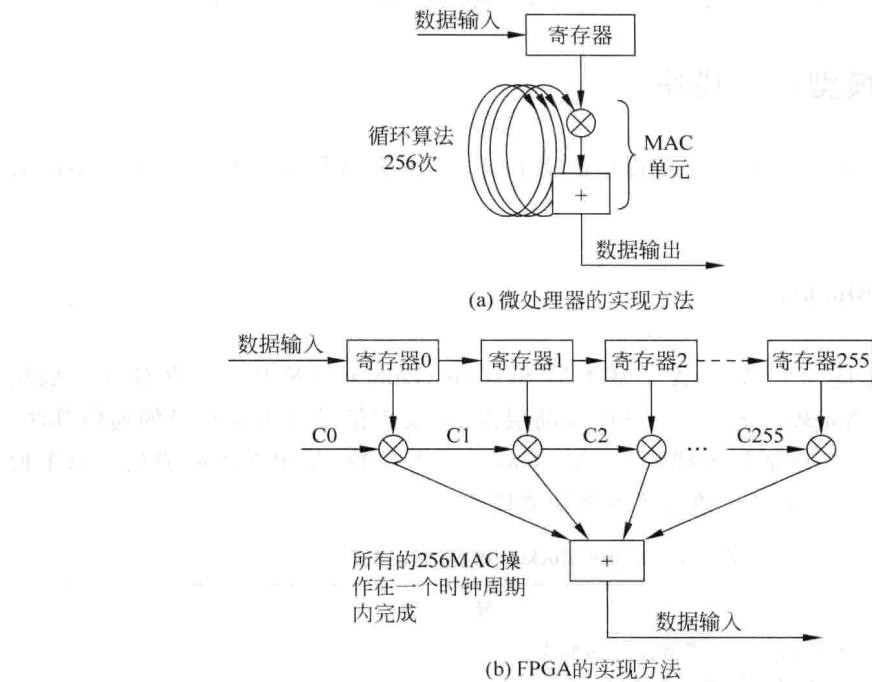


图 6.1 256 阶的 FIR 滤波器的实现

FPGA 实现,而不使用数字信号处理器实现。

System Generator 运行在 MATLAB 软件的 Simulink 工具中,该工具是基于模型的设计方法,通过使用标准的 Simulink 块集(Simulink block set)来创建一个可执行的规范。该规范可以使用浮点数值精度进行设计,而无须知道详细的硬件细节。一旦定义了功能和基本的数据流,System Generator 就可以用来为 Xilinx 的器件指定硬件的实现细节。System Generator 使用了用于 Simulink 的 Xilinx DSP 块集,并且自动调用 Xilinx 的核生成器为所构建的 DSP 模块生成经过优化的网表。System Generator 能执行所有下游的实现工具,为编程 FPGA 生成比特流文件。可以选择通过使用来自 Simulink 环境所提取出来的测试向量创建测试平台,这个测试平台可以用于 ModelSim 或者 ISim 仿真器。

System Generator 并不是用来取代基于 HDL 的设计,但是可以使设计的注意力只集中在关键部分。类似地,绝大多数的 DSP 程序员都不会专门在汇编器中编程,他们都在高级语言开始编程,比如 C。只有必须满足性能要求时,才编写汇编代码。

给出一个拇指规则:在设计中必须管理内部硬件时钟的部分,应该使用 HDL 实现。设计中少量的关键部分使用 System Generator 实现,然后将 HDL 和 System Generator 设计连接起来。通常,除了外部接口以外,一个信号处理系统的绝大部分不需要这个级别的控制。System Generator 提供了机制,用于将 HDL 代码导入到设计中,这对于 HDL 设计者来说,令人非常感兴趣。

另一个令设计者感兴趣的方面是 System Generator 能自动地生成 HDL 测试平台,包括测试向量。

硬件协同仿真处理接口,允许设计者在 Simulink 的控制下,在硬件上运行设计。这

样,可以充分发挥 MATLAB 和 Simulink 用于数据分析和可视化的功能。

## 6.2 FPGA 模型设计模块

本节介绍 System Generator 中提供的 Xilinx Block 和 Xilinx Reference Blockset 设计模块。

### 6.2.1 Xilinx Blockset

Xilinx 的块集是一个库,包含了基本的 System Generator 模块。一些模块是底层的,提供了对器件指定硬件的访问;其他是高层次的,实现信号处理和高级的通信算法。为了方便起见,具有广泛适用的模块(比如:Gateway I/O 块)是很多库的成员。每个模块包含在 Index 库中。表 6.1 给出了库及功能描述。

表 6.1 Xilinx Blockset 库及功能描述

库	描述
AXI4	带有符合 AXI4 规范接口的模块
Basic Elements	用于数字逻辑的标准的建立模块
Communication	前向错误纠错和调制器模块,通常用于数字通信系统中
Control Logic	用于控制电路和状态机的模块
DSP	数字信号处理模块
Data Types	用于进行数据类型转换的模块(包括 Gateways)
Float-Point	支持浮点类型的模块
Index	在 Xilinx Blockset 中的每个模块
Math	实现算术功能的模块
Memory	实现和访问存储器的模块
Shared Memory	实现和访问 Xilinx 共享存储器的模块
Tools	Utility 工具,代码生成(System Generator token),资源评估,HDL 协同仿真,等

### 6.2.2 Xilinx Reference Blockset

Xilinx Reference Blockset 包含了 System Generator 的复合体,用于实现范围更广的功能。表 6.2 给出了 Xilinx Reference Blockset 的库及功能描述。

表 6.2 Xilinx Reference Blockset 库及功能描述

库	描述
Communication	用于数字通信系统中的块
Control Logic	用于控制电路和状态机的逻辑块
DSP	数字信号处理模块
Imaging	图像处理模块
Math	实现算术功能的块

## 6.3 System Generator 运行环境的配置

在使用 System Generator 工具前,必须安装 MathWorks 公司的 MATLAB 软件工具。在本书中,使用 Vivado 2013.3 和 MATLAB R2013a 集成开发环境。

配置 System Generator 运行环境的步骤主要包括:

(1) 在正确安装 MATLAB R2013a 工具和 Vivado 2013.3 设计套件后,在 Windows 环境的桌面下,选择开始→所有程序→Xilinx Design Tools→Vivado 2013.3→System Generator→System Generator 2013.3 MATLAB Configurator,打开配置界面。

(2) 如图 6.2 所示,选中 R2013a 前面的复选框。

(3) 单击 OK 按钮。

(4) 出现 Info 对话框界面。

(5) 单击 OK 按钮,退出配置界面。

注: 必须严格对应 MATLAB 的版本号,否则在运行设计时会报错。

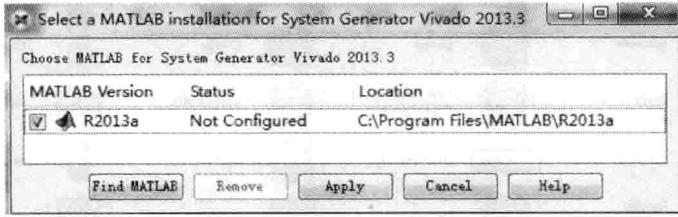


图 6.2 MATLAB 配置界面

## 6.4 信号模型的构建和实现

本节将介绍信号仿真模型的构建和实现过程。

### 6.4.1 信号模型的构建

该设计将实现下面的浮点离散数字信号处理模型:

$$y(n) = x(n) + 3.0 \times x(n-1) + 4.0 \times x(n-2) + 1.5 \times x(n-3)$$

通过 System Generator 工具实现该模型的步骤包括:

(1) 首先通过 Z 变换将其转换到 Z 域进行描述。

$$y(n) = x(n) + 3.0 \times x(n-1) + 4.0 \times x(n-2) + 1.5 \times x(n-3)$$

$$\Rightarrow Y(z) = X(z) + 3.0 \times z^{-1} \times X(z) + 4.0 \times z^{-2} \times X(z) + 1.5 \times z^{-3} \times X(z)$$

$$\Rightarrow Y(z) = X(z) \times (1 + 3.0 \times z^{-1} + 4.0 \times z^{-2} + 1.5 \times z^{-3})$$

(2) 在 Windows 7 主界面下,选择: 开始→所有程序→Xilinx Design Tools→Vivado 2013.3→System Generator→System Generator 2013.3,打开 MATLAB R2013a 开发环境。

(3) 在 MATLAB 主界面 Home 标签窗口下,单击如图 6.3 所示的 Simulink Library 按钮。

(4) 出现 Simulink Library Browser 窗口界面。在该界面主菜单下,选择 File→New→Model,出现一个空白的设计界面。

(5) 在 Simulink Library Browser 主界面的 Libraries 窗口下,找到并展开 Xilinx Blockset。如图 6.4 所示,在展开项中找到并双击 Floating-Point 图标。



图 6.3 Simulink Library 按钮



Floating-Point

图 6.4 Floating-Point 图标

(6) 如图 6.5 所示,在窗口右侧的 Library: Xilinx Blockset/Floating-Point 标签窗口下,出现用于浮点信号处理的各种 Xilinx IP 核元件符号。

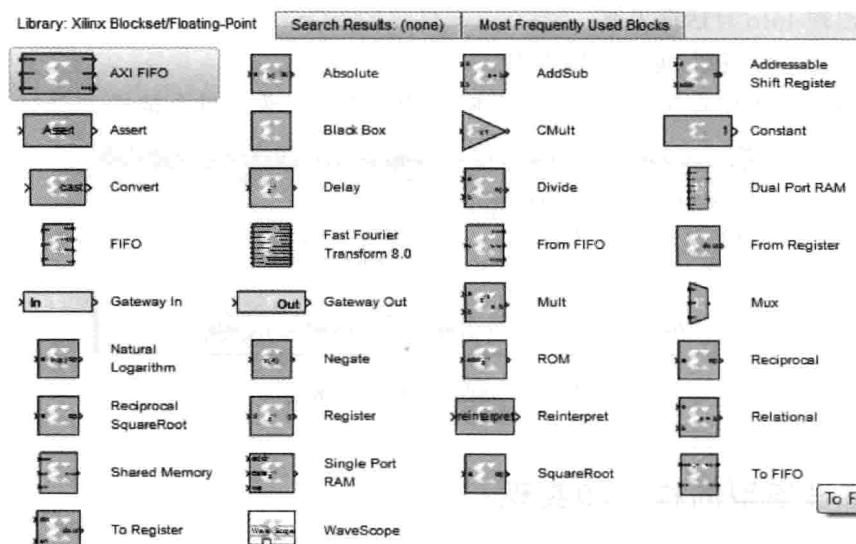
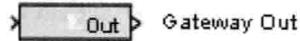


图 6.5 浮点处理元件

(7) 在如图 6.5 所示的界面下,找到图 6.6 所示的 Gateway In 元件符号和 Gateway Out 元件符号,将其拖入到如图 6.7 所示的空白界面中。



(a) Gateway In元件



(b) Gateway Out元件

图 6.6 Gateway In 和 Gateway Out 元件

**注:** Gateway In 元件支持:

- ① 从浮点数转换为 N 位 boolean 类型,有符号(二进制补码)或者无符号定点精度。
- ② 在转换期间,提供选项管理额外的位。
- ③ 在 System Generator 生成的 HDL 设计中,定义了顶层输入端口的名字。
- ④ 当在 System Generator 块中,选中 Create Testbench 复选框时,定义了测试平台的激励源。

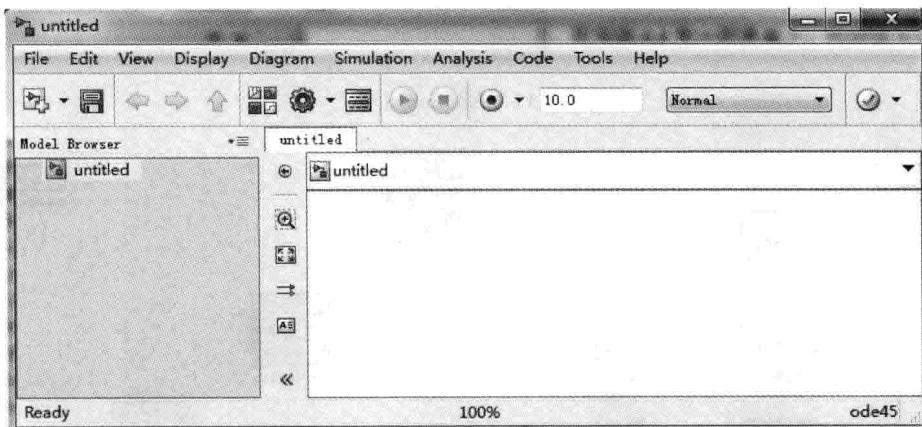


图 6.7 空白设计界面

Gateway Out 支持：

- ① 将 System Generator 产生的定点数转换为 Simulink 的双精度数。
- ② 在 System Generator 生成的 HDL 设计中, 定义了顶层输出端口的名字。
- (8) 在同样的窗口下, 找到图 6.8 所示的 Delay 元件符号, 拖入三个 Delay 元件到图 6.10 所示的设计界面中。
- (9) 在同样的窗口下, 找到图 6.9 所示的 CMult 元件符号, 拖入三个 CMult 元件到图 6.10 所示的设计界面中。



图 6.8 Delay 元件

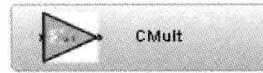


图 6.9 CMult 元件

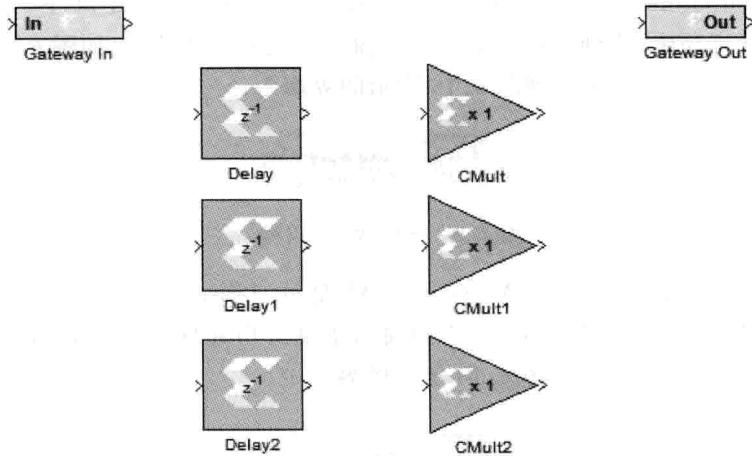


图 6.10 放入 Delay 和 CMult 元件后的界面

- (10) 在同样的窗口下, 找到图 6.11 所示的 AddSub 元件符号, 拖入三个 AddSub 元件到图 6.12 所示的设计界面中。



图 6.11 AddSub 元件

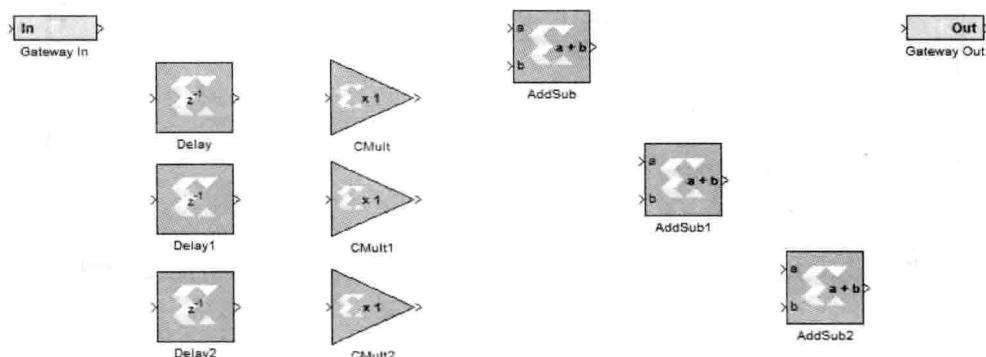


图 6.12 放入 AddSub 元件后的界面

(11) 在 Simulink Library Browser 主界面的 Libraries 窗口下, 找到并展开 Xilinx Blockset。在展开项中, 找到 Basic Elements。在右边窗口出现基本元件符号, 如图 6.13 所示, 找到 System Generator Token 符号。将其拖入到图 6.16 所示的界面中。

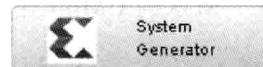


图 6.13 System Generator

注: 这个符号必须出现在所有的 System Generator 设计中, 否则在运行设计时会报错。

(12) 在 Simulink Library Browser 主界面的 Libraries 窗口下, 找到并展开 Simulink。在展开项中, 找到 Sources, 在右边窗口出现源元件符号, 如图 6.14 所示, 找到 Sine Wave 元件符号, 将其拖入到图 6.16 所示的界面中。

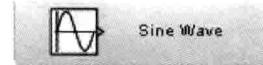


图 6.14 Sine Wave 符号

(13) 在 Simulink Library Browser 主界面的 Libraries 窗口下, 找到并展开 Simulink。在展开项中, 找到 Sinks, 在右边窗口出现分析元件符号。如图 6.15 所示, 找到 Scope 元件符号, 将其拖入到图 6.16 所示的界面中。

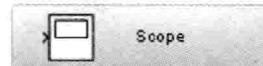


图 6.15 Scope 元件

注: 打开 Scope 元件符号, 将 Number of axes 设置为 2。



System Generator

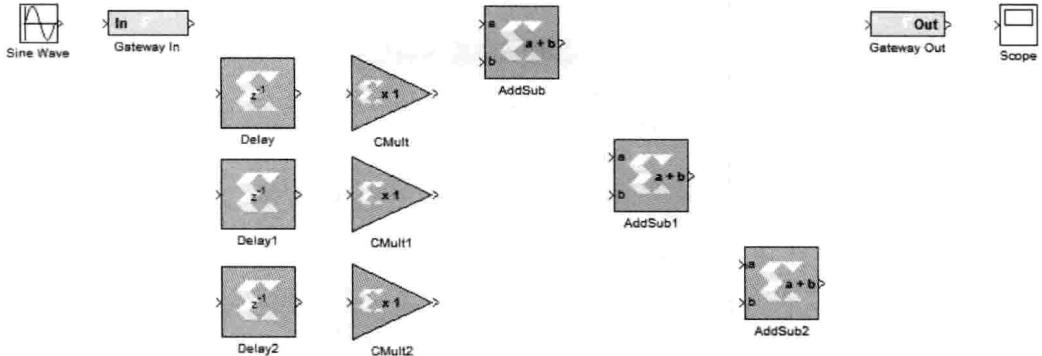


图 6.16 放置 Scope 元件后的界面

(14) 按如图 6.17 所示的界面,将这些元件进行连接在一起。



System Generator

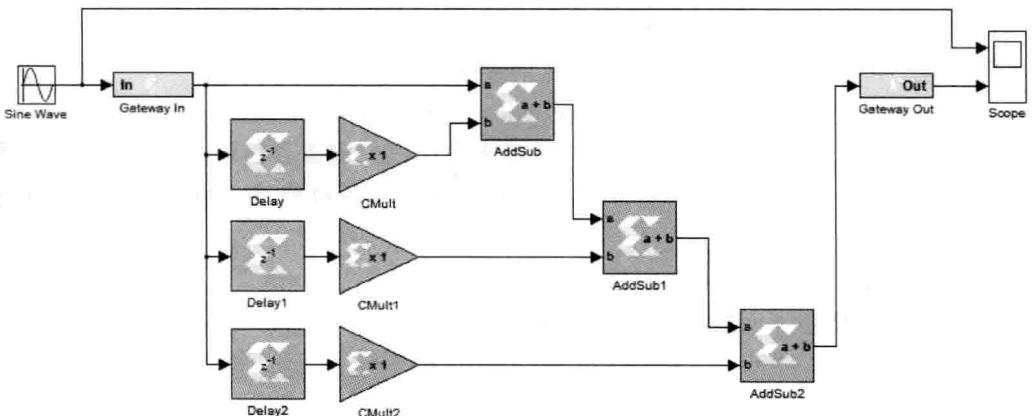


图 6.17 连接元件后的模型界面

(15) 保存模型。在该设计中,将其保存到 e:\vivado\_example\model\_design\basic\_model\basic\_model.slx。

## 6.4.2 模型参数的设置

下面将为该模型设置合适的参数。其参数配置的步骤主要包含：

(1) 双击图 6.17 内的 Sine Wave 符号, 如图 6.18 所示, 打开正弦信号参数配置界面。首先介绍该配置界面内一些参数的含义:

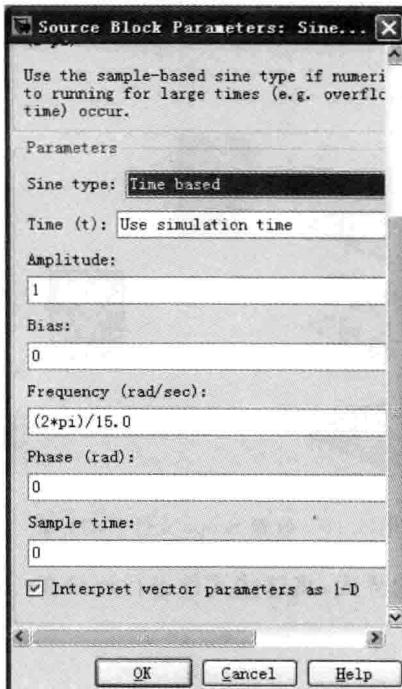


图 6.18 Sine Wave 参数设置界面

### ① 基于时间(Time-Based)的模式:

在该模式下, 使用下面的公式计算正弦信号的输出:

$$y = \text{amplitude} \times \sin(\text{frequency} \times \text{time} + \text{phase}) + \text{bias}$$

在该模式下有两个子模式: 连续模式或者离散模式。配置界面中的 Sample time 参数的值确定子模式。当该参数取值为 0 时, 该模块运行在连续模式。当该参数的值大于 0 时, 该模式运行在离散模式。

### ② 基于采样(Sample-Based)的模式:

在该模式下, 使用下面的公式计算正弦信号的输出:

$$y = A \sin(2\pi(k+o)/p) + b$$

其中:

- ①  $A$  是正弦信号的幅度;
- ②  $p$  是每个正弦周期的采样个数;
- ③  $k$  是重复的整数值, 其范围为  $0 \sim p-1$ ;
- ④  $o$  是信号的偏置(相位移动);
- ⑤  $b$  是信号的直流偏置。

在该参数配置界面中, 按如下设置参数:

- ① Sine type: Time based;
- ② Time(t): Use simulation time;

- ③ Amplitude: 1;
- ④ Bias: 0;
- ⑤ Frequency(rad/sec):  $(2 * \pi) / 15.0$ ;
- ⑥ Phase(rad): 0;
- ⑦ Sample time: 0(连续时间)。

(2) 单击 OK 按钮,退出 Sine Wave 参数设置界面。

(3) 在如图 6.17 所示的界面中,双击 Gateway In 符号,打开参数设置界面,按如下设置参数:

- ① Output Type: Float-point;
- ② Floating-point Precision: Single;
- ③ 其他按默认参数设置。

(4) 单击 OK 按钮,退出参数设置界面。

(5) 在如图 6.17 所示的界面,双击 Delay1 符号,打开参数设置界面,按如下设置参数:在 Basic 标签下,设置 Latency 为 2(延迟为 2)。

(6) 单击 OK 按钮,退出参数设置界面。

(7) 在如图 6.17 所示的界面,双击 Delay2 符号,打开参数设置界面,按如下设置参数:在 Basic 标签下,设置 Latency: 3(延迟为 3)。

(8) 单击 OK 按钮,退出参数设置界面。

(9) 在如图 6.17 所示的界面,双击 CMult 符号,打开参数设置界面,按如下设置参数:

- ① Constant value: 3;
- ② Constant Type: Floating-point;
- ③ Floating-point Precision: Single。

(10) 单击 OK 按钮,退出参数设置界面。

(11) 在如图 6.17 所示的界面,双击 CMult1 符号,打开参数设置界面,按如下设置参数:

- ① Constant value: 4;
- ② Constant Type: Floating-point;
- ③ Floating-point Precision: Single。

(12) 单击 OK 按钮,退出参数设置界面。

(13) 在如图 6.17 所示的界面,双击 CMult2 符号,打开参数设置界面,按如下设置参数:

- ① Constant value: 1.5;
- ② Constant Type: Floating-point;
- ③ Floating-point Precision: Single。

- (14) 单击 OK 按钮,退出参数设置界面。
  - (15) 在如图 6.17 所示的界面,双击 Addsub 符号,打开参数设置界面,按如下设置参数:在 Basic 标签下,设置 Latency 为 0(无延迟)。
  - (16) 单击 OK 按钮,退出参数设置界面。
  - (17) 在如图 6.17 所示的界面,双击 Addsub1 符号,打开参数设置界面,按如下设置参数:在 Basic 标签下,设置 Latency 为 0(无延迟)。
  - (18) 单击 OK 按钮,退出参数设置界面。
  - (19) 在如图 6.17 所示的界面,双击 Addsub2 符号,打开参数设置界面,按如下设置参数:在 Basic 标签下,设置 Latency 为 0(无延迟)。
  - (20) 单击 OK 按钮,退出参数设置界面。
  - (21) 在如图 6.17 所示的界面,双击 Scope 符号,打开 Scope 主界面。在其主界面的工具栏下,单击按钮,打开图 6.19 所示的显示界面。
- 在 General 标签栏下,将 Number of axes 设置为 2。表示有两个参数将显示在 Scope 窗口。

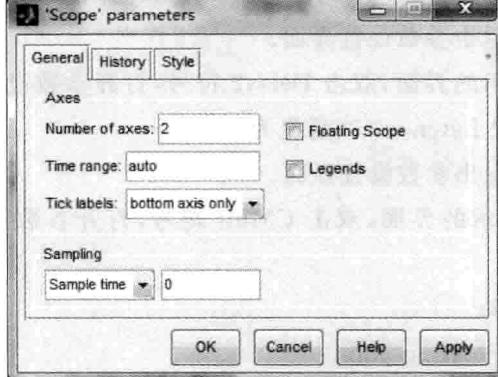


图 6.19 参数设置界面

- (22) 单击 OK 按钮,退出参数设置界面。
- (23) 在 Scope 主界面下,单击 **X** 退出 Scope 界面。
- (24) 保存该设计。

### 6.4.3 信号处理模型的仿真

本节对该信号处理模型进行仿真。实现仿真的步骤主要包括:

- (1) 在设计界面工具栏内的输入框中,输入 45.0;
- (2) 在设计界面工具栏下,单击 **Run** 按钮,开始仿真;
- (3) 仿真结束后,在图 6.17 所示界面中,单击 Scope 符号;
- (4) 打开如图 6.20 所示的仿真结果显示窗口;
- (5) 退出 Scope 窗口。

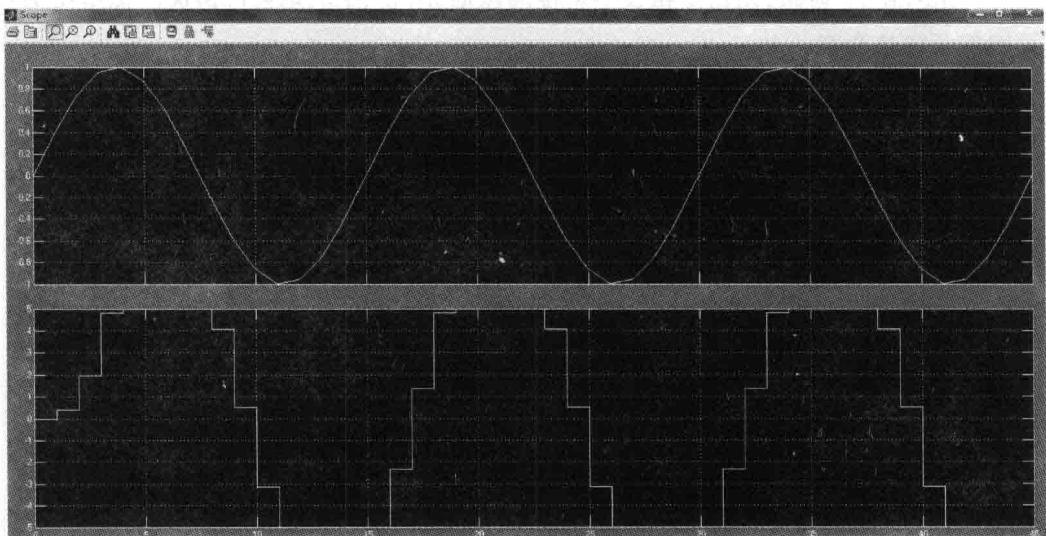


图 6.20 仿真结果观察窗口

#### 6.4.4 生成模型子系统

本节将生成子系统。实现子系统的步骤主要包括：

- (1) 如图 6.21 所示,用鼠标选中黑框内的区域。

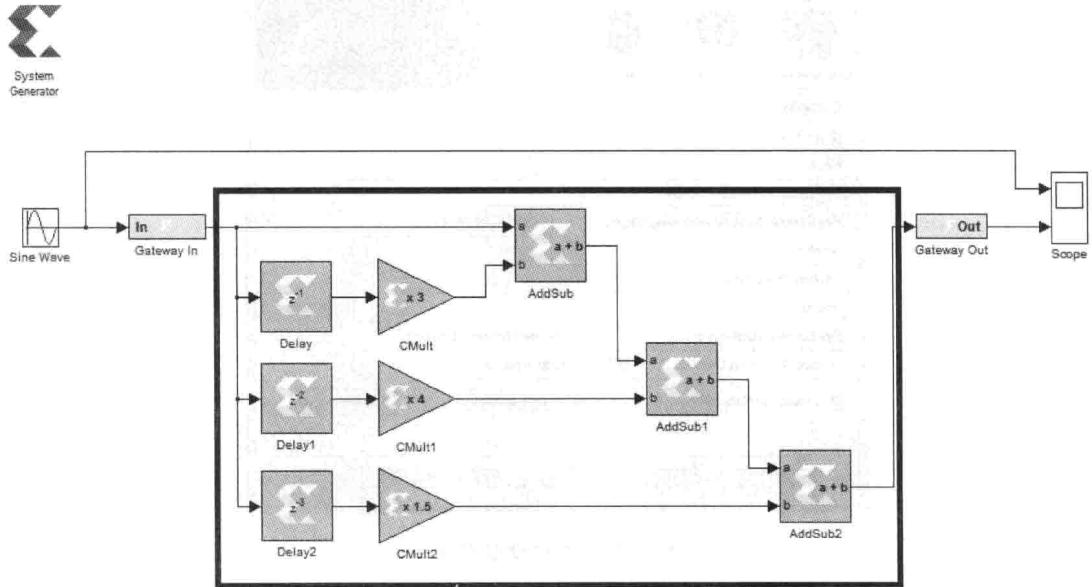


图 6.21 选择黑框内的区域

- (2) 单击鼠标右键,出现浮动菜单。在浮动菜单内,选择 Create Subsystem From Selection 选项。

(3) 如图 6.22 所示,给出了包含子系统的模型,图 6.21 的黑框区域内的模块作为整个系统的子模块存在。

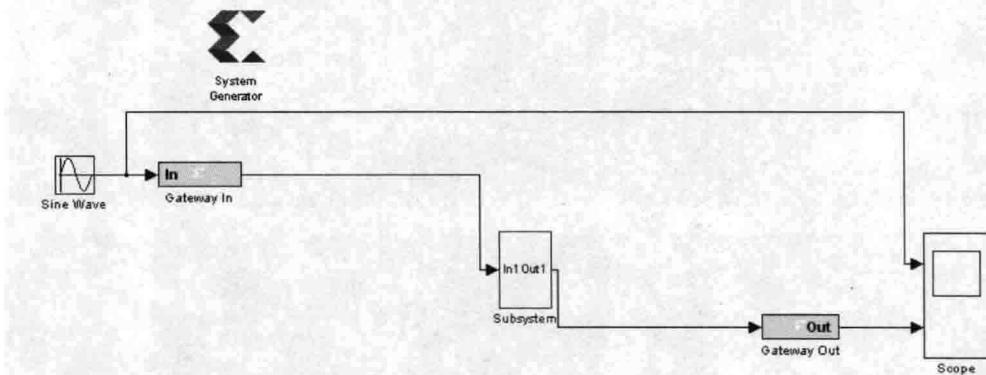


图 6.22 包含子系统的模型

#### 6.4.5 模型 HDL 代码的生成

生成模型 HDL 代码的步骤主要包含:

(1) 在图 6.22 所示的界面中,双击 System Generator 符号。如图 6.23 所示打开 System Generator: basic\_model 对话框界面。按下面设置参数:



图 6.23 参数设置界面

- ① Compilation: HDL Netlist;
- ② Part: Zynq xc7z020-1clg484;
- ③ Hardware description language: Verilog;
- ④ Target directoty: ./netlist(在当前工程目录下的子目录 netlist);

- ⑤ Synthesis strategy: Vivado Synthesis Defaults;
- ⑥ Implementation strategy: Vivado Implementation Defaults;
- ⑦ 选中 Create interface document 前面的复选框;
- ⑧ 选中 Create testbench 前面的复选框。
- (2) 单击 Generate 按钮,生成该模型的 HDL 描述和测试平台。
- (3) 综合完成后,出现 Compilation status 对话框界面。在该界面中,提示 Generation Completed 信息,表示成功完成生成过程。
- (4) 单击 OK 按钮。

#### 6.4.6 打开生成设计文件并仿真

下面将使用 Vivado 集成开发环境,打开 System Generator 工具生成的 Verilog HDL 代码和测试平台。打开 Verilog HDL 代码的步骤主要包含:

- (1) 在保存 slx 文件的目录下,找到 netlist 子目录。在该子目录下,找到 hdl\_netlist 子目录。在该子目录下,双击 basic\_model.xpr。如图 6.24 所示,给出该设计模型的文件结构。

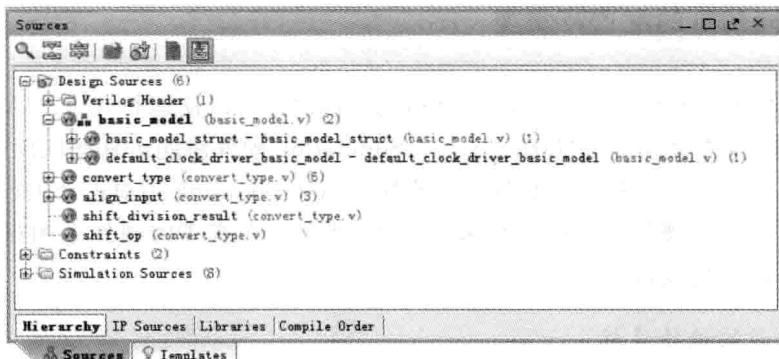


图 6.24 模型的 HDL 描述

- (2) 如图 6.25 所示,在 Source 窗口下,找到并展开 Simulation Sources。在展开项中,找到并展开 Sim\_1。在展开项中,选中 basic\_model\_tb。

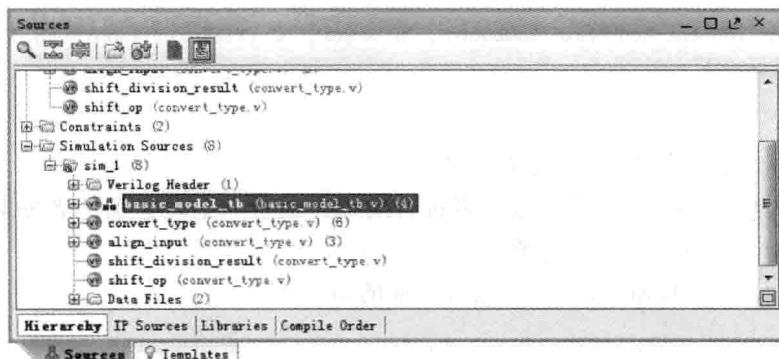


图 6.25 测试平台

(3) 在 Vivado 主界面左侧 Flow Navigator 窗口,找到并展开 Simulation。在展开项中,找到 Run Simulation。然后,选择 Run Behavioral Simulation,开始对设计进行行为级仿真。

(4) 如图 6.26 所示,出现行为仿真波形界面。

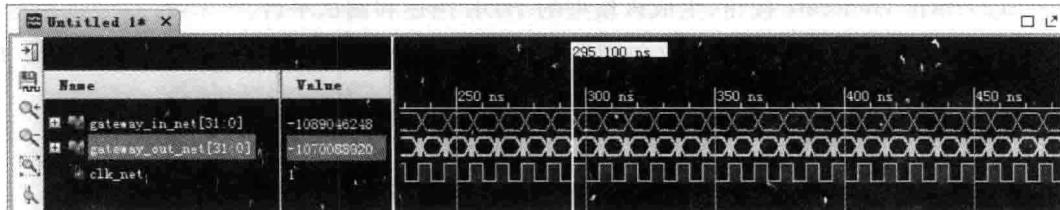


图 6.26 仿真波形界面

(5) 退出 Vivado 集成开发环境。

注: 将该设计保存到 e:\Vivado\_example\model\_design\basic\_model\_hw\basic\_model.slx。

#### 6.4.7 协同仿真的配置及实现

System Generator 提供了硬件协同仿真能力,这样可以使运行在 FPGA 上的设计导入到 Simulink 仿真环境中。Hardware Co-Simulation 硬件协同仿真选项用于编译目标,并且自动地生成比特流,将其关联到一个模块。当在 Simulink 中对设计进行硬件协同仿真后,就可以知道在硬件上的运行结果。这样,就允许在真正的硬件上测试被编译的部分,并且动态加速仿真过程。

##### 1. 添加新的编译目标

在进行协同仿真前,需要配置仿真环境参数。该仿真所对应的目标平台为 Xilinx XUP 提供的 Zedboard 开发板,该开发板上使用了 Xilinx 的 Zynq-7000 器件。配置协同仿真环境的参数配置步骤主要包括:

(1) 在 MATLAB 命令窗口下,输入下面的命令:

```
xilinx.environment.addBoard('Zedboard', 'U:\demo', 'C:\Xilinx\Vivado\2013.3\data\boards\zynq\ZED\revD')
```

其中:

'U:\demo' 为设计者任意指定的路径。

该命令提供了一个模板类供设计者进行编辑。最后一个域根据读者安装 Vivado 所包含 board.xml 文件的路径进行修改。

(2) 在 MATLAB 命令窗口下,输入下面的命令:

```
xilinx.environment.rehashCompilationTarget
```

这将确保新的编译目标出现在 System Generator 符号中。

- (3) 重新打开 System Generator 符号。
- (4) 如图 6.27 所示,在 Compilation 下拉框中,添加了 Zedboard。

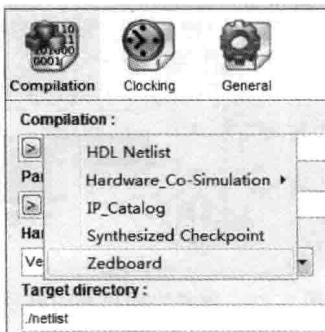


图 6.27 添加了新的编译目标

## 2. 生成协同仿真模块

下面介绍生成协同仿真模块的主要步骤。

- (1) 在 Simulink 设计环境中,定位到路径:

```
E:\vivado_example\model_design\basic_model_hw
```

重新打开 basic\_model.slx 文件。

- (2) 如图 6.28 所示,在 Compilation 下拉框中,选择 Zedboard。

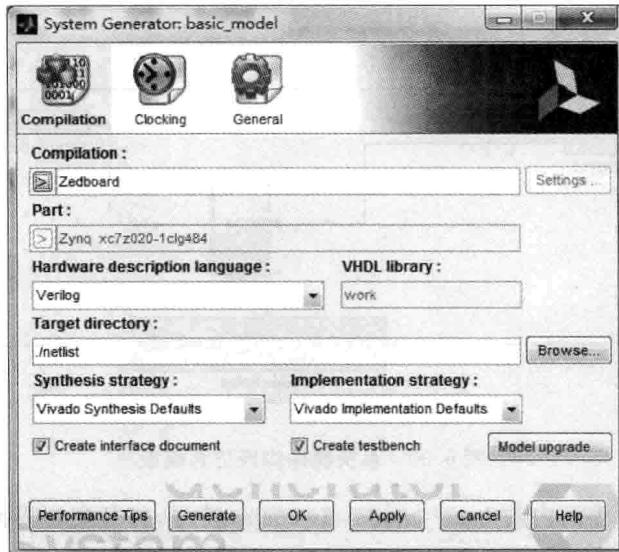


图 6.28 选择新的编译目标

- (3) 单击 Generate 按钮。对所选的目标平台进行编译。
- (4) 当编译完成后,出现 Compilation status 对话框界面。该界面中,提示 Generation Completed 信息,表示编译成功。

(5) 单击 OK 按钮。

(6) 如图 6.29 所示,生成 basic\_model hwcosim\_lib 协同仿真符号。

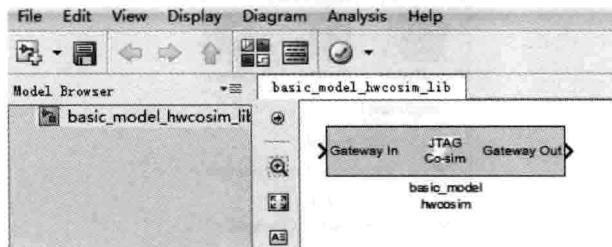


图 6.29 硬件协同仿真的符号

### 3. 协同仿真的实现

下面介绍实现硬件协同仿真的主要步骤。

(1) 将图 6.29 内的 basic\_model hwcosim 符号复制粘贴到如图 6.30 所示的界面中。

(2) 如图 6.30 所示,将硬件协同仿真符号连接到数字信号处理模型系统中,完成硬件系统仿真的整体结构设计。

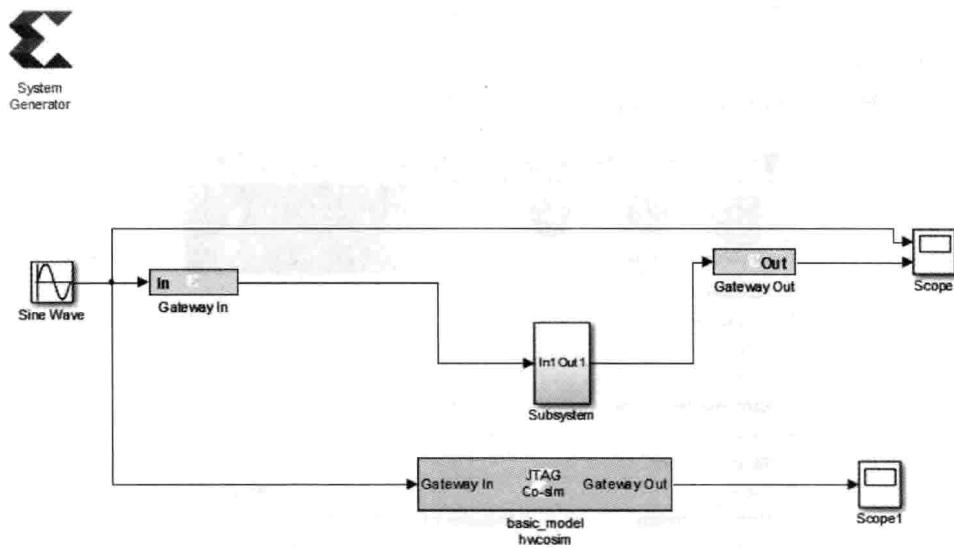


图 6.30 系统硬件协同仿真模型

(3) 双击 6.30 界面内的 basic\_model hwcosim 符号,打开配置界面。

(4) 如图 6.31 所示,单击 Cable 标签,进入 Cable 设置界面。在 Cable Settings 下面的 Type 右侧下拉框中选择 Digilent USB JTAG Cable。

(5) 单击 OK 按钮。

(6) 在主界面的工具栏下单击 ▶ 按钮,启动硬件协同仿真。

(7) 等待仿真结束后,双击 Scope1 符号。如图 6.32 所示,打开仿真结果界面,观察协同仿真的结果。

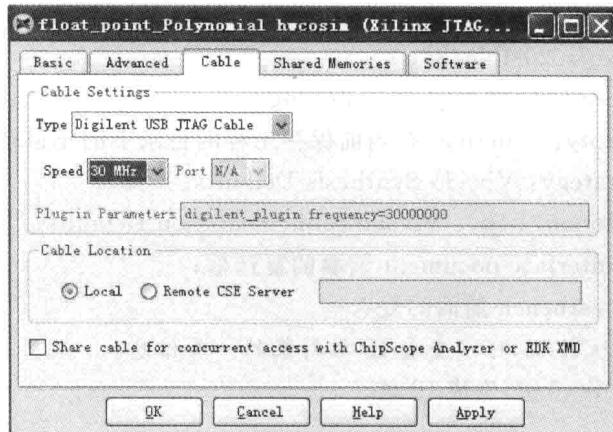


图 6.31 下载电缆配置界面

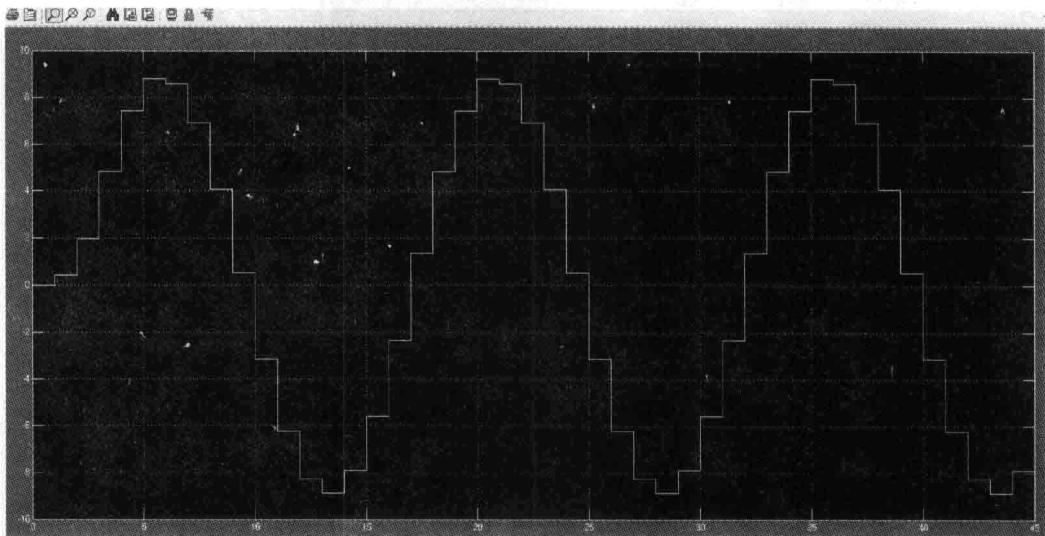


图 6.32 仿真结果界面

#### 6.4.8 生成 IP 核

System Generator 提供了 IP 封装器编译目标, 允许设计者将 System Generator 生成的设计封装到一个 IP。这样, 就可以将其包含在 Vivado IP 目录中。通过例化语句将其包含在其他设计中, 作为其他设计的一部分。生成 IP 核的步骤主要包括:

- (1) 在 Simulink 设计环境中, 定位到路径:

```
E:\vivado_example\model_design\basic_model_ip
```

重新打开 basic\_model.slx 文件。

- (2) 双击 System Generator 图标, 如图 6.33 所示, 打开 System Generator 对话框界面, 按下面参数配置:

- ① Compilation: IP Catalog;
- ② Part: Zynq xc7z020-1clg484;
- ③ Hardware description language: Verilog;
- ④ Target directory: ./netlist(在当前保存工程的目录下的子目录 netlist);
- ⑤ Synthesis strategy: Vivado Synthesis Defaults;
- ⑥ Implementation strategy: Vivado Implementation Defaults;
- ⑦ 选中 Create interface document 前面的复选框;
- ⑧ 选中 Create testbench 前面的复选框。

注：单击 Settings 按钮，可以查看 IP 核封装的一些信息。

(3) 单击 Generate 按钮，生成 IP 核。



图 6.33 IP 生成界面

## 6.5 编译 MATLAB 到 FPGA

本节将介绍编译 MATLAB 到 FPGA 的方法和设计流程。

### 6.5.1 模型的设计原理

System Generator 通过使用 MCode 模块提供了对 MATLAB 的直接支持。MCode 模块将输入值应用到 M-函数，用于对 Xilinx 定点数据类型进行评估，且在每个采样的周期进行评估。模块通过使用永久的状态变量来保持内部的状态。模块的输入端口由 M-函数指定的输入变量所决定，输出端口由 M-函数输出变量所决定。这个模块为构建有限状态机、控制逻辑等，提供了一个便捷的方法。

本节将通过 MCode 构建两个滤波器，并将两个模块的计算结果进行比较。

## 1. simple\_fir 函数

**代码清单 6-1 simple\_fir.m**

```
function y = simple_fir(x, lat, coefs, len, c_nbBits, c_binpt, o_nbBits, o_binpt)
    coef_prec = {xlSigned, c_nbBits, c_binpt, xlRound, xlWrap};
    out_prec = {xlSigned, o_nbBits, o_binpt};
    coefs_xfix = xfix(coef_prec, coefs);
    persistent coef_vec, coef_vec = xl_state(coefs_xfix, coef_prec);
    persistent x_line, x_line = xl_state(zeros(1, len-1), x);
    persistent p, p = xl_state(zeros(1, lat), out_prec, lat);
    sum = x * coef_vec(0);
    for idx = 1:len-1
        sum = sum + x_line(idx-1) * coef_vec(idx);
        sum = xfix(out_prec, sum);
    end
    y = p.back;
    p.push_front_pop_back(sum);
    x_line.push_front_pop_back(x);
```

## 2. fir\_transpose 函数

**代码清单 6-2 fir\_transpose.m 函数**

```
function y = fir_transpose(x, lat, coefs, len, c_nbBits, c_binpt, o_nbBits, o_binpt)
    coef_prec = {xlSigned, c_nbBits, c_binpt, xlRound, xlWrap};
    out_prec = {xlSigned, o_nbBits, o_binpt};
    coefs_xfix = xfix(coef_prec, coefs);
    persistent coef_vec, coef_vec = xl_state(coefs_xfix, coef_prec);
    persistent reg_line, reg_line = xl_state(zeros(1, len), out_prec);
    if lat <= 0
        error('latency must be at least 1');
    end
    lat = lat - 1;
    persistent dly,
    if lat <= 0
        y = reg_line.back;
    else
        dly = xl_state(zeros(1, lat), out_prec, lat);
        y = dly.back;
        dly.push_front_pop_back(reg_line.back);
    end
    for idx = len-1:-1:1
        reg_line(idx) = reg_line(idx - 1) + coef_vec(len - idx - 1) * x;
    end
    reg_line(0) = coef_vec(len - 1) * x;
```

注：预先设计这两个文件，并将其保存到：e:\vivado\_example\model\_design\mcode\_model 目录下。

### 6.5.2 系统模型的建立

本节将建立系统模型。建立系统模型的步骤主要包含：

- (1) 在 Windows 7 主界面下,选择:开始→所有程序→Xilinx Design Tools→Vivado 2013.3→System Generator→System Generator 2013.3,打开 System Generator 开发工具。
- (2) 在 MATLAB 主界面 Home 标签窗口下,单击如图 6.34 所示的按钮,打开 Simlink 工具箱。
- (3) 在 Simulink 主界面主菜单下,选择 File→New→Model,建立一个新的模型。
- (4) 在 Simulink Library Browser 窗口下,找到并展开 Xilinx Blockset。在展开项中,找到并单击 Math,在窗口右边出现数学运算元件。
- (5) 右侧窗口中,找到图 6.35 所示的 MCode 元件符号,将其拖入到新模型设计界面中(读者可以参考已经完成设计的图 6.36)。



图 6.34 Simulink 入口

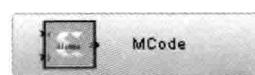


图 6.35 MCode 模块符号

- (6) 类似的,在该界面下,找到图 6.35 所示的 MCode 元件符号,再次将其拖入到新模型设计界面中(读者可以参考已经完成设计的图 6.36)。



System Generator

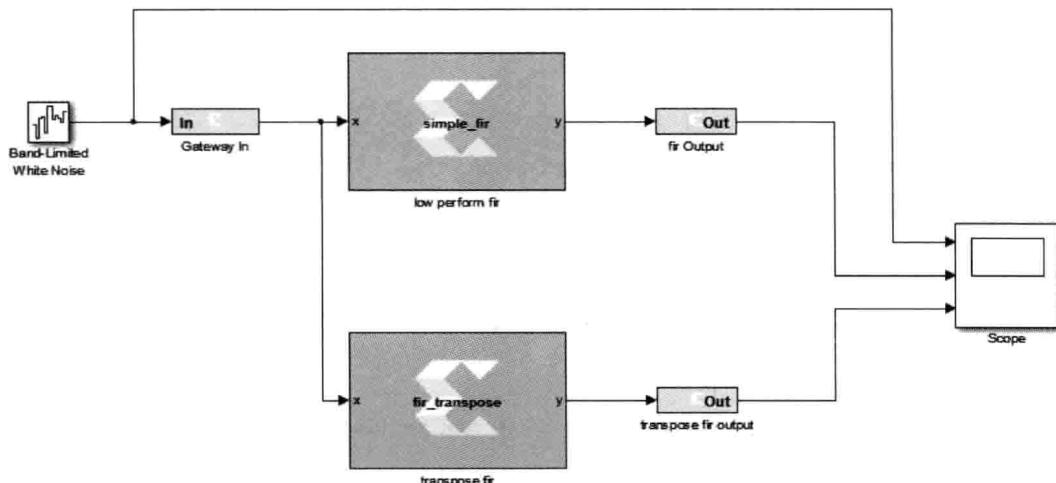


图 6.36 完成后的系统结构

(7) 在 Simulink Library Browser 窗口下,找到并展开 Xilinx Blockset。在展开项中,找到并单击 Basic Elements。在右侧窗口中找到 Gateway In 元件符号,将其拖入到模型设计界面中(读者可以参考已经完成设计的图 6.36)。

(8) 在 Simulink Library Browser 窗口下,找到并展开 Xilinx Blockset。在展开项中,找到并单击 Basic Elements。在右侧窗口中找到 Gateway Out 元件符号,将其分两次拖入到模型设计界面中(读者可以参考已经完成设计的图 6.36)。

(9) 在 Simulink Library Browser 窗口下,找到并展开 Simulink。在展开项中,找到 Sources。在右边窗口找到 Band-Limited White Noise 元件,将其拖入到图 6.36 所示的界面中。

(10) 在 Simulink Library Browser 窗口下,找到并展开 Simulink。在展开项中,找到 Sinks。在右边窗口找到 Scope 元件符号,将其拖入到图 6.36 所示的界面中。

(11) 在 Simulink Library Browser 窗口下,找到并展开 Xilinx Blockset。在展开项中,找到 Basic Elements。在右边窗口找到 System Generator Token 符号,将其拖入到图 6.36 所示的界面中。

(12) 双击图 6.36 中上面的 MCode 模块,打开参数配置对话框界面。如图 6.37 所示,单击 Basic 标签。在该标签窗口下,单击 Browse... 按钮。定位到路径:

```
e:\vivado_example\model_design\mcode_model\simple_fir.m
```

这样,就可以在 MATLAB function 标题下的窗口中,看到 simple\_fir。

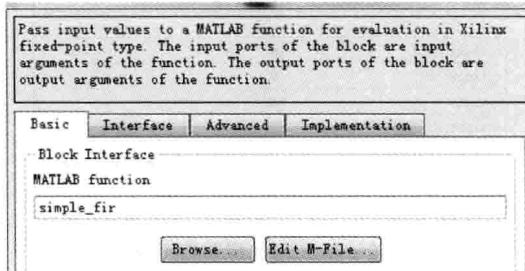


图 6.37 Basic 标签配置界面(1)

(13) 单击 Interface 标签,如图 6.38 所示。在该标签窗口下,按图中设置参数。

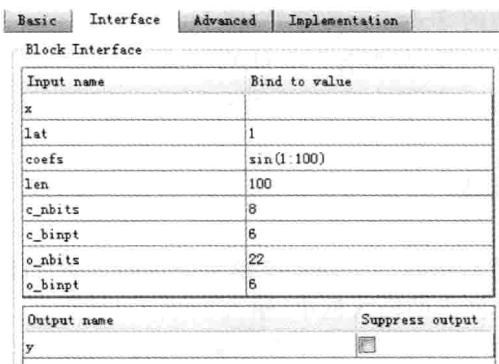


图 6.38 Interface 标签配置界面(1)

(14) 单击 Advanced 标签,在该标签窗口下,选中 Enable printing with disp 前面的复选框。

(15) 单击 OK 按钮。

(16) 双击图 6.36 中下面的 MCode 元件符号。打开参数配置对话框界面。如图 6.39 所示,单击 Basic 标签。在该标签窗口下,单击 Browse... 按钮。定位到路径:

e:\vivado\_example\model\_design\mcode\_model\fir transpose.m

这样,就可以在 MATLAB function 标题栏下的窗口中,看到 fir transpose。

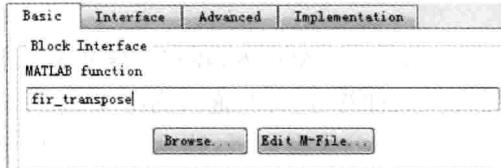


图 6.39 Basic 标签配置界面(2)

(17) 如图 6.40 所示,单击 Interface 标签。在该标签界面下,按图中设置参数。

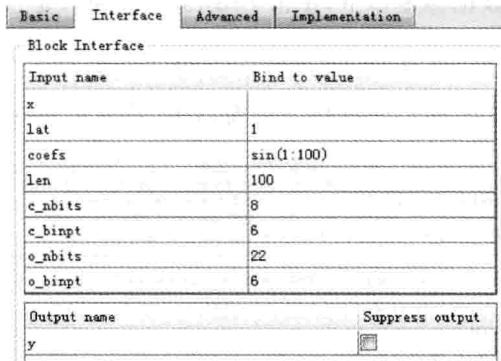


图 6.40 Interface 标签配置界面

(18) 单击 Advanced 标签,在该标签窗口下,选中 Enable printing with disp 前面的复选框。

(19) 单击 OK 按钮。

(20) 双击图 6.36 内的 Scope 元件符号,将 Number of axis 设置为 3。

(21) 如图 6.36 所示,完成模型各个元件的连接。

注:读者可以按照图 6.36 修改各个元件的名字。

### 6.5.3 系统模型的仿真

下面对该设计模型进行仿真。实现系统模型仿真的步骤主要包括:

(1) 在设计界面工具栏内的输入框中,输入 100;

(2) 在 Simulink 主界面的工具栏内,单击 按钮,开始仿真;

(3) 在如图 6.36 所示的界面中,单击 Scope 符号。如图 6.41 所示,打开仿真观察窗

口,观察仿真结果;

(4) 退出 Scope 窗口。

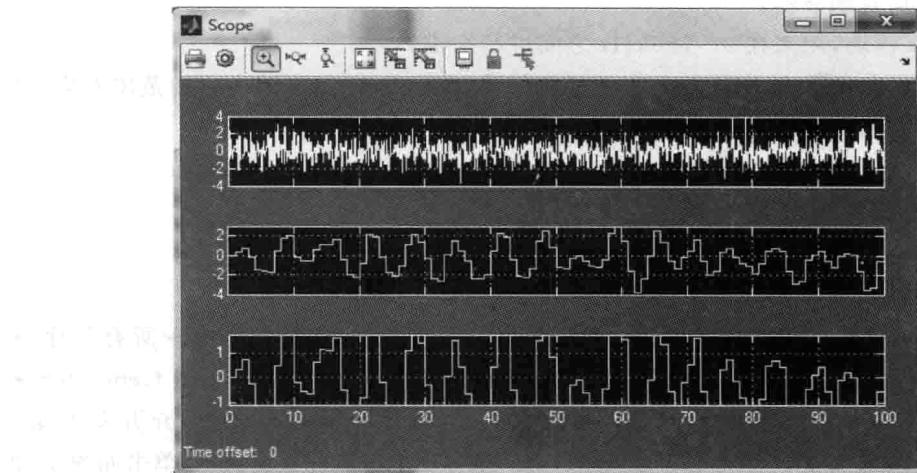


图 6.41 仿真波形界面

## 6.6 FIR 滤波器的设计与实现

本节将介绍使用 System Generator 所提供的 FIR 和 FDA 工具模块实现 FIR 滤波器。FDA Tool 模块用来定义滤波器的阶数和系数; FIR 模块用于 Simulink 仿真和在 FPGA 内通过 Vivado 实现设计。并且,在真正的 FPGA 上运行这个设计来验证其功能。

### 6.6.1 FIR 滤波器设计原理

读者可能是公司里的一个数字信号处理应用工程师。你所在的公司正在调研使用数字滤波器来代替模拟滤波器的可行性,这个滤波器用于它的安全标签监测器。这样做的目的是尝试改善其性能,以及降低系统的整体成本。这就使得公司能进一步地渗透到日益增长的安全市场空间。下面给出这个单通道、单数据率滤波器的设计指标:

- (1) 采样频率  $F_s = 1.5\text{MHz}$ ;
- (2)  $F_{stop1} = 270\text{kHz}$ ;
- (3)  $F_{pass1} = 300\text{kHz}$ ;
- (4)  $F_{pass2} = 450\text{kHz}$ ;
- (5)  $F_{stop2} = 480\text{kHz}$ ;
- (6) 在通带双边带的衰减 = 54dB;
- (7) 通带纹波 = 1。

由于 FPGA 的灵活性、上市时间和性能优势超过了数字信号处理器,所以选择 FPGA 实现数字滤波器作为一个设计工程师,你的 HDL 设计经验是有限的,但是如果读者很熟悉 MathWorks 的 MATLAB,使用 System Generator 是一个优秀的解决方案,这

个方案用来帮助读者在 FPGA 内实现滤波器。

该设计在 Xilinx XUP 提供的 Nexys4 开发平台上实现设计原型。该设计使用了两种不同的源来仿真滤波器：

- (1) Chirp 模块, 用来在 0~750kHz 之间进行扫描;
- (2) 随机源生成器, 输出均匀分布的随机信号, 范围是 0~1。因为它的范围有限, 所以均匀分布是用来驱动定点滤波器的最好选择。

### 6.6.2 生成 FIR 滤波器系数

生成 FIR 滤波器系数的步骤主要包括：



图 6.42 Simulink 入口

- (1) 在 Windows 7 主界面下, 选择: 开始→所有程序→Xilinx Design Tools→Vivado 2013.3→System Generator→System Generator 2013.3, 打开 System Generator 开发工具。

- (2) 在 MATLAB 主界面 Home 标签窗口下, 单击如图 6.42 所示的按钮, 打开 Simlink 工具箱。

- (3) 在 Simulink 主界面主菜单下, 选择 File→New→Model, 建立一个新的名字为 bandpass\_filter.slx 的模型。

注: 其保存路径为 E:\vivado\_exampe\model\_design。

- (4) 在 Simulink Library Brower 的 Libraries 窗口下, 找到并展开 Xilinx Blockset。在展开项中, 选择 DSP。

- (5) 在右侧窗口中, 找到 FDA Tools 元件符号, 将其添加到空白设计界面中。
- (6) 双击 FDA Tools 符号。如图 6.43 所示, 打开滤波器参数配置界面, 按照设计指标输入滤波器参数。

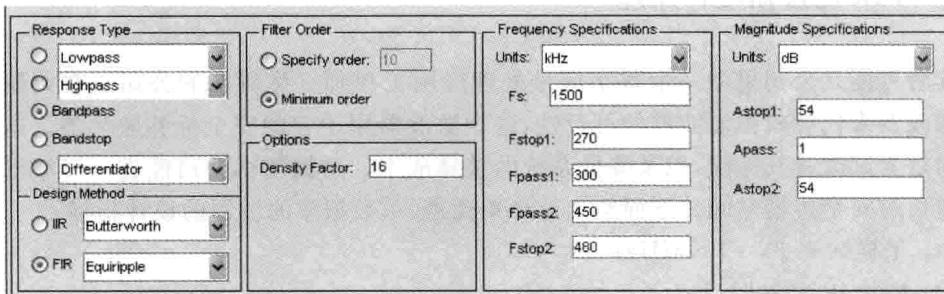


图 6.43 滤波器参数配置界面

- (7) 单击该界面内的 Design Filter 按钮。如图 6.44 所示, 出现所设计滤波器幅频响应特性等。

- (8) 在 MATLAB 主界面主菜单下, 选择 File→Export。如图 6.45 所示, 出现 Export 对话框界面, 该界面用于将名字为 Num 的滤波器系数导入到工作空间。

注: 将在 MATLAB 工作空间内添加 Num 变量。对于一个 FIR 滤波器, Num 表示滤波器所使用的系数, 这是一个可选的步骤。

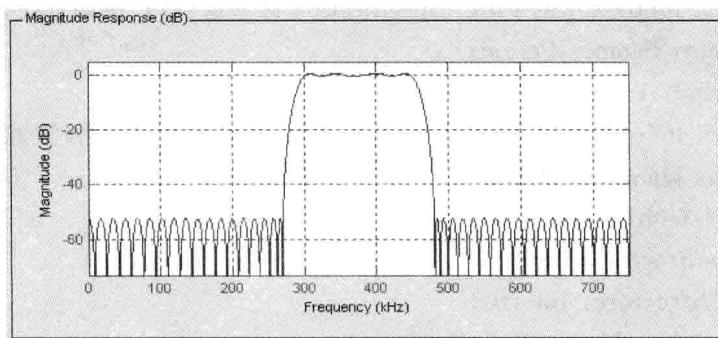


图 6.44 滤波器的频谱图



图 6.45 导出滤波器系数

- (9) 单击 Export 按钮,导出滤波器系数数据。
- (10) 在 MATLAB 主界面命令窗口中输入 Num,查看生成的滤波器系数。
- (11) 输入 Max(Num)和 Min(Num),确定滤波器最大的系数值和最小的系数值,这些数值决定了滤波器系数的宽度和二进制的小数点。

### 6.6.3 建模 FIR 滤波器模型

本节将建模 FIR 滤波器模型,建模 FIR 滤波器的步骤主要包含:

(1) 在 Simulink Library Brower 的 Libraries 窗口下,找到并展开 Xilinx Blockset。在展开项中,选择并单击 DSP。

(2) 在右侧窗口找到 FIR Compiler 7.1 元件符号,并将其添加到设计界面中。

(3) 双击 FIR 元件符号,打开参数配置界面:

① 在如图 6.46(a)所示的 Filter Specification 标签界面内,按如下设置参数:

Coefficient Vector: xlfda\_numerator('FDATool');

Number of Coeffiicent Sets: 1;

Filter type: Single\_rate;

Select format: Sample\_Period。

② 在如图 6.46(b)所示的 Filter Specification 标签窗口内,按如下设置参数:

Select format: Sample\_Period;

Sample period: 1.

③ 在如图 6.46(c)所示的 Implementation 标签窗口内,按如下设置参数:

Quatization: Quantize\_Only;

Coefficients Width: 12;

Coefficients Fractional Bits: 12;

Coefficient Structure: Inferred;

Output Rounding Mode: Full\_Precision。

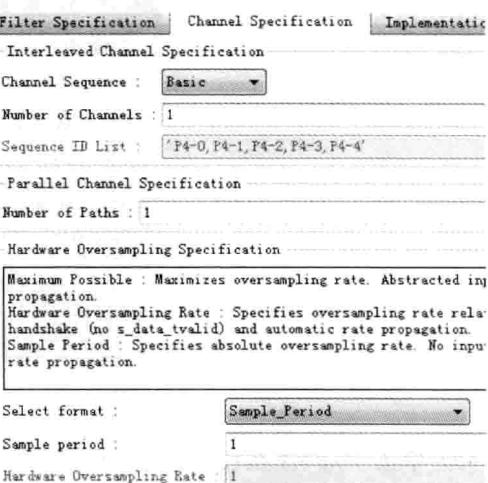
④ 在 Advanced 标签界面内,选中 Display shortened port names 前面的复选框。

⑤ 其余按默认参数设置。

(a) 滤波器设计界面(1)



(b) 滤波器设计界面(2)



(c) 滤波器设计界面(3)

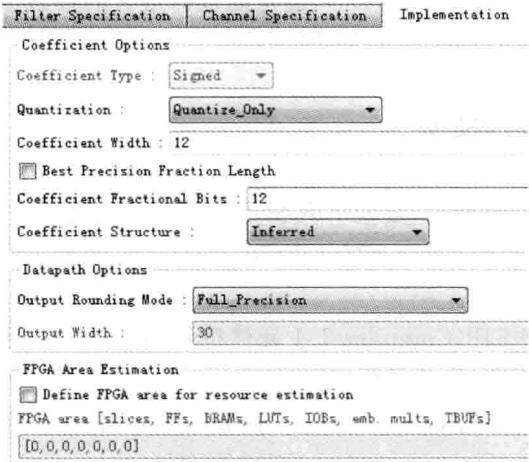


图 6.46 FIR 滤波器参数配置界面

(4) 单击 OK 按钮,退出滤波器配置界面。

(5) 在 Simulink Library Browser 窗口下,找到并展开 Xilinx Blockset。在展开项中,找到并单击 Basic Elements。在右侧窗口中找到 Gateway In 元件符号,将其拖入到模型设计界面中(读者可以参考已经完成设计的图 6.47)。

(6) 双击图 6.47 界面内的 Gateway In 元件符号,按下面参数设置:

- ① Output Type: Fixed-point;
- ② Number of bits: 8。

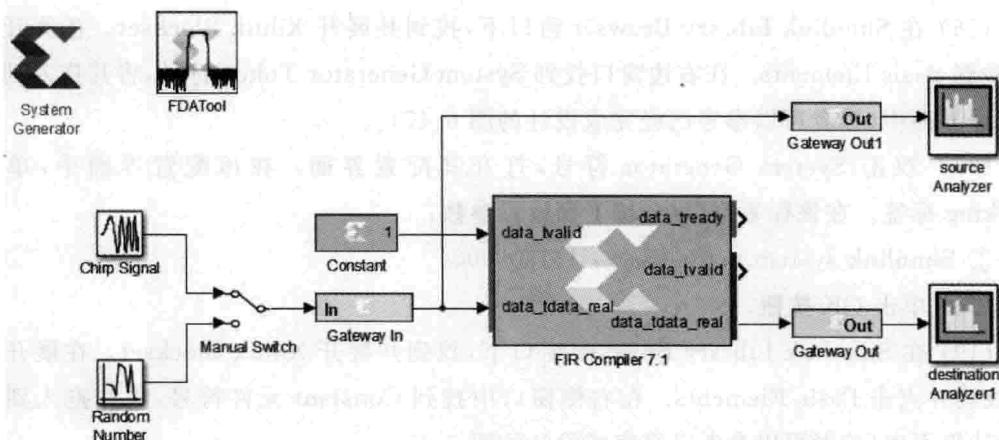


图 6.47 系统设计模型

(7) 在 Simulink Library Browser 窗口下,找到并展开 Xilinx Blockset。在展开项中,找到并单击 Basic Elements。在右侧窗口中找到 Gateway Out 元件符号,将其分两次拖入到模型设计界面中(读者可以参考已经完成设计的图 6.47)。

(8) 在 Simulink Library Browser 窗口下,找到并展开 Simulink。在展开项中,找到 Sources。在右边窗口找到 Chirp Signal 元件符号,将其拖入模型设计界面中(读者可以参考已经完成设计的图 6.47)。

(9) 双击 Chrp Signal 元件符号,打开其参数配置界面,按下面设置参数:

- ① Initial frequency(Hz): 0.1;
- ② Target time(secs): 100;
- ③ Frequency at target time(Hz): 750000

(10) 单击 OK 按钮。

(11) 在 Simulink Library Browser 窗口下,找到并展开 Simulink。在展开项中,找到 Sources。在右边窗口找到 Random Number 元件符号,将其拖入模型设计界面中(读者可以参考已经完成设计的图 6.47)。

(12) 双击 Random Number 元件符号,打开其参数配置界面,按下面设置参数

- ① Mean: 0;
- ② Variance: 1;
- ③ Sample time: 1/1500000。

(13) 单击 OK 按钮。

(14) 在 Simulink Library Browser 窗口下,找到并展开 Simulink。在展开项中,找到 Signal Routing。在右边窗口找到 Manual Switch 元件符号,将其拖入模型设计界面中(读者可以参考已经完成设计的图 6.47)。

(15) 在 Simulink Library Browser 窗口下,找到并展开 DSP System Toolbox。在展开项中,找到 Sinks。在右边窗口找到 Spectrum Analyzer 元件符号,将其分两次拖入模型设计界面中(读者可以参考已经完成设计的图 6.47)。

(16) 在 Simulink Library Browser 窗口下,找到并展开 Xilinx Blockset。在展开项中,找到 Basic Elements。在右边窗口找到 System Generator Token 符号,将其拖入到模型设计界面中(读者可以参考已经完成设计的图 6.47)。

(17) 双击 System Generator 符号,打开其配置界面。在该配置界面中,单击 Clocking 标签。在该标签窗口中,按下面设置参数:

① Simulink system period(sec): 1/1500000。

(18) 单击 OK 按钮。

(19) 在 Simulink Library Browser 窗口下,找到并展开 Xilinx Blockset。在展开项中,找到并点击 Basic Elements。在右侧窗口中找到 Constant 元件符号,将其拖入到模型设计界面中(读者可以参考已经完成设计的图 6.47)。

(20) 双击 Constant 元件符号,打开其参数配置界面。在 Basic 标签界面下,将 Output Type 设置为 Boolean。

(21) 单击 OK 按钮,退出参数配置界面。

(22) 按照图 6.47 所示,完成系统中所有元件的连接。

#### 6.6.4 仿真 FIR 滤波器模型

本节将对 FIR 滤波器模型进行仿真。仿真 FIR 滤波器模型的步骤包括:

(1) 双击图 6.47 内的 Manual Switch,将开关切换到 Chirp Signal;

(2) 在设计界面工具栏内的输入框中,输入 100;

(3) 单击设计界面工具栏内的按钮,开始进行仿真;

(4) 如图 6.48 和图 6.49 所示,给出源信号和通过滤波器后输出信号的频谱图,验证在 FIR 滤波器通带外的信号被衰减;

(5) 双击图 6.47 内的 Manual Switch,将开关切换到 Random Number;

(6) 如图 6.50 和图 6.51 所示,给出源信号和通过滤波器后输出信号的频谱图,验证在 FIR 滤波器通带外的信号被衰减;

(7) 单击设计界面内的按钮,停止仿真过程;

**注:** 将设计模型另保存为 bandpass\_filter\_conv.slx 用于后续设计。

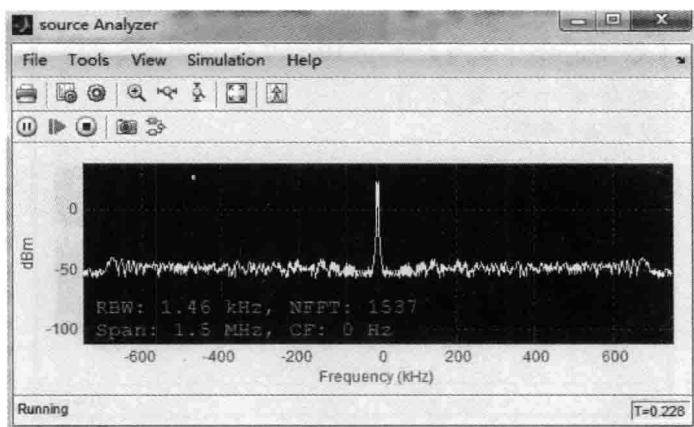


图 6.48 Chirp 信号

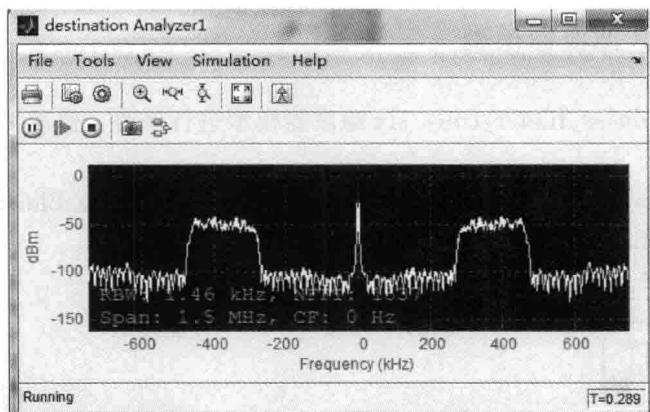


图 6.49 Chirp 信号通过滤波器后的波形

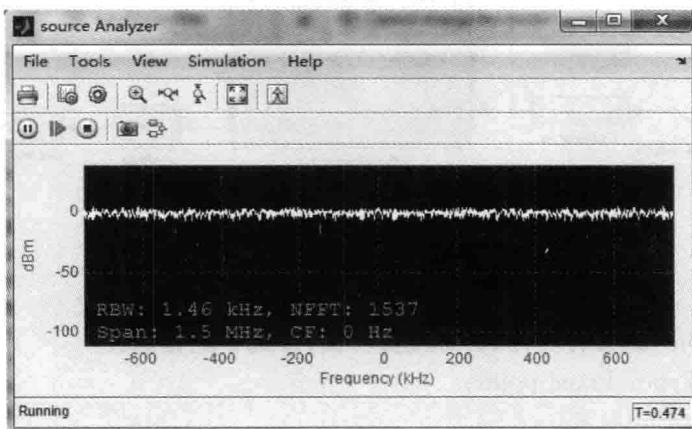


图 6.50 Random Number 信号

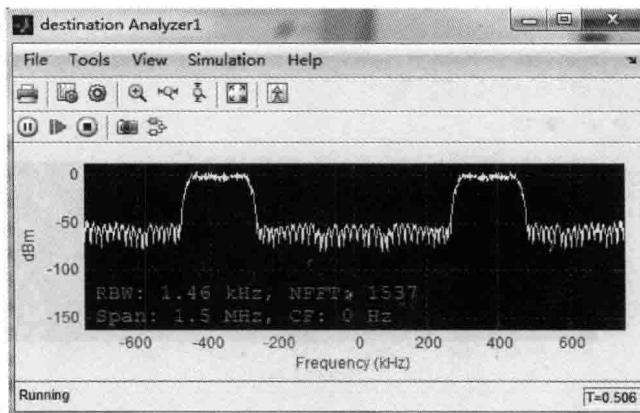


图 6.51 Random Number 信号通过滤波器后的信号

### 6.6.5 修改 FIR 滤波器模型

本节将对 bandpass\_fileter\_conv.slx 滤波器模型进行修改。修改 FIR 滤波器模型的步骤主要包括：

- (1) 在 Simulink Library Browser 窗口下, 找到并展开 Xilinx Blockset。在右侧窗口中, 找到 Convert 元件, 将其添加到设计中。
- (2) 如图 6.52 所示, 完成与 FIR 滤波器的输出的连接。

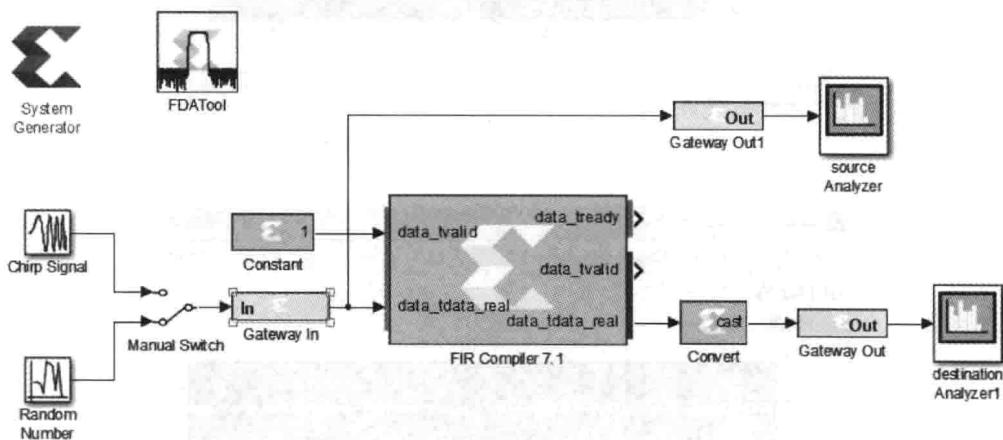


图 6.52 修改设计模型

- (3) 双击 Convert 元件符号, 打开配置界面, 按如下配置参数:

- ① Output Type: Fixed-point;
- ② Number of bits: 8;
- ③ Binary point: 6;
- ④ Quantization: Truncate;
- ⑤ Overflow: Wrap;

- ⑥ 其他按默认设置。  
 (4) 单击 OK 按钮。  
 (5) 保存修改后设计模型。

### 6.6.6 仿真修改后 FIR 滤波器模型

本节将对修改后的 FIR 滤波器模型进行仿真。仿真修改后的 FIR 滤波器模型的步骤包括：

- (1) 双击图 6.52 内的 Manual Switch, 将开关切换到 Chirp Signal。
- (2) 在设计界面工具栏内的输入框中, 输入 100。
- (3) 单击设计界面工具栏内的 按钮, 开始进行仿真。
- (4) 如图 6.53 所示, 给出通过滤波器后的输出信号的频谱图, 验证在 FIR 滤波器带外的信号被衰减。
- (5) 双击图 6.52 内的 Manual Switch, 将开关切换到 Random Number。
- (6) 如图 6.54 所示, 给出了通过滤波器后输出信号的频谱图。验证在 FIR 滤波器通带外的信号被衰减。
- (7) 单击设计界面内的 按钮, 停止仿真过程。

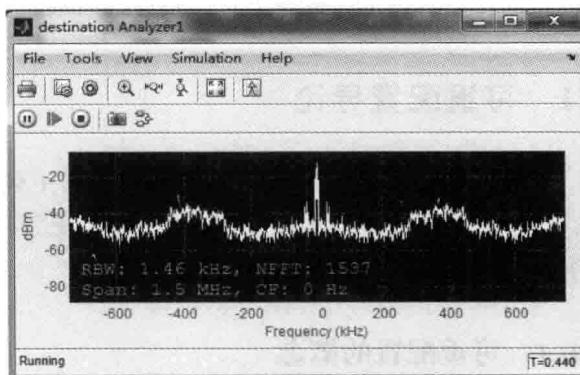


图 6.53 Chirp 信号通过修改后的滤波器的波形

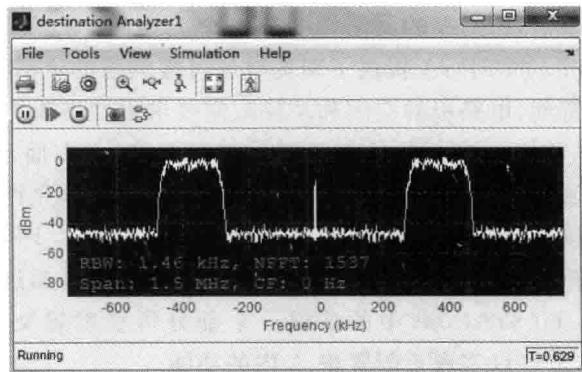


图 6.54 Random Number 信号通过修改后的滤波器的波形

# 第7章 Vivado 部分可重配置设计流程

可重配置技术是 Xilinx 公司提供的用来高效利用 FPGA 设计资源,实现 FPGA 资源可重复利用的最新 FPGA 设计技术。这种技术的发展为 FPGA 应用提供了更加广阔前景。本章重点介绍了对数字逻辑系统的可重配置技术,内容包括:可重配置导论、部分可重配置实现。

本章所介绍的设计例子,综合了 Vivado 2013.3 软件的复杂应用。这些设计例子,也反映了 Xilinx FPGA 高级设计技术及特点。

对于部分可重配置更详细的一些设计方法,可以参考《Xilinx FPGA 设计权威指南》(清华大学出版社,2012)。

## 7.1 可重配置导论

本节介绍可重配置的一些基础知识。其中包括:可重配置的概念、可重配置的应用、可重配置的特点、可重配置术语解释、可重配的要求、可重配置的标准、可重配置的流程。

### 7.1.1 可重配置的概念

当系统变得越来越复杂时,设计者要求用尽可能少的成本做更多的事情,FPGA 的灵活性变成一个关键的“财富”。为什么这么说呢?Xilinx 的 FPGA 提供了现场设备可重编程的灵活性。当今,成本越来越苛刻,电路板的空间和功耗限制要求更加高效的设计策略。

Xilinx 所提供的部分可重配置功能,一方面,允许对 FPGA 指定区域使用新的功能进行重新配置;另一方面,允许在器件的剩余空间继续运行当前的设计,这样,就进一步扩展了 FPGA 固有的灵活性。如图 7.1 所示给出了部分可重配置的原理图,通过将 A1.bit、A2.bit、A3.bit 和 A4.bit 中的任何一个部分可重配置文件下载到配置块 A 中,就可以实现重配置块 A 内的功能。

FPGA 内的逻辑分割成两种不同的类型:可重配置的逻辑和静态逻辑。FPGA 块内灰颜色的区域表示静态逻辑,标识可重配置块 A 的

区域为可重配置逻辑。静态逻辑所保留的功能不会由于加载一个部分比特文件受到影响。通过替换部分比特文件的内容,改变可重配置逻辑功能。

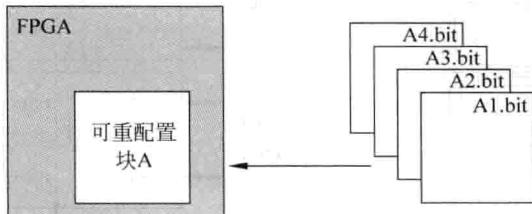


图 7.1 部分可重配置结构

### 7.1.2 可重配置的应用

#### 1. 网络多端口接口

为了更好地说明可重配置的优势,下面以网络多端口接口的设计为例。如图 7.2 所示,开关的端口可能支持多个接口协议。然而,在配置 FPGA 前,系统不可能预测使用哪个协议。为了 FPGA 设备不会被重新配置,以及由此所造成的禁止所有端口的情况,每个端口上都要实现各种可能的接口协议。

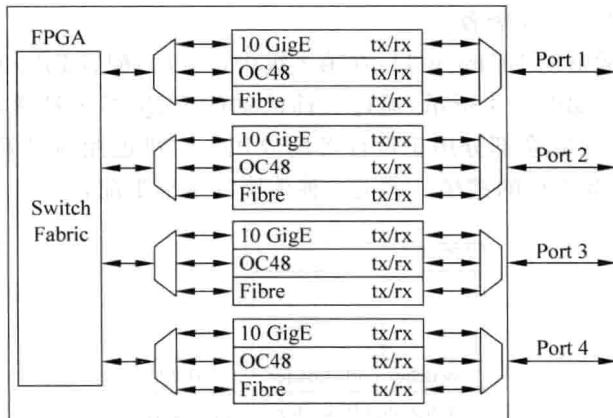


图 7.2 不带有重配置的网络开关

从图中可以看出,该设计的效率很低,这是因为每个端口只使用一种协议。如图 7.3 所示,部分可重配置通过使每个端口接口成为可重配置的模块能实现更高效的设计。这样,就消除了用于将多个协议引擎连接到一个端口上所需要的 MUX 元件。

此外,通过可重配置结构,在任何时候都可以支持新的协议,而不会影响静态逻辑和开关逻辑。当端口加载一个新的协议时,其他端口并不受到影响。可以创建额外的标准,并添加到配置存储器库中,而不要求完全地重新设计。这样,就可以允许系统有更大的灵活性和可靠性。可以创建一个调试模块,当一个端口遇到错误时,一个没有使用的端口能加载分析/修正逻辑来实时地处理问题。

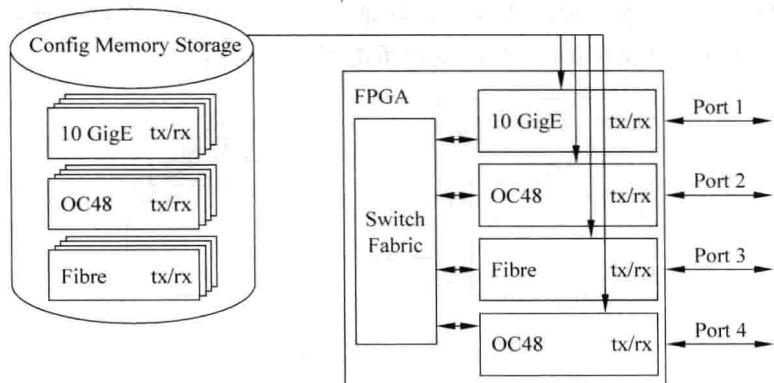


图 7.3 带有部分可重配置的网络开关

## 2. 通过 PCIe 接口的可重配置应用

当供电稳定后的 100ms, PCIe 规范要求端口准备用于连接训练。由于新一代 FPGA 容量的不断增加, 确定配置存储器大小将变成一个困难的任务。解决这个困难的一个创新的方法是, 利用部分重配置将在一个大规模 FPGA 内的、基于 PCIe 规范的整个配置分割成两个顺序步骤:

- (1) 初始化 PCIe 系统连接配置;
- (2) 随后用户使用可重配置。

当基本复位释放前的 100ms 窗口, 在第 1 级内, 可以只配置 FPGA 内用于 PCIe 和相关逻辑的集成模块。如图 7.4 所示, 通过当前活动的 PCIe 系统链接, 使用部分重配置。然后, 主机配置 FPGA 剩余部分用于设计者的应用。主机也能通过 PCIe 总线更新设计者的应用, 而不失去和主机的链接。这是一种传统的全部重配置。

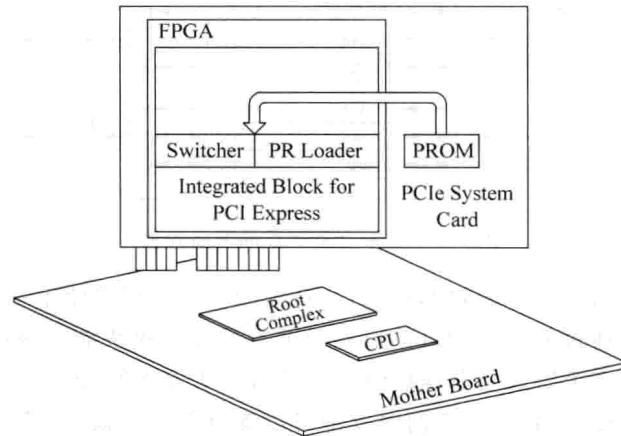


图 7.4 使用部分重配置快速配置 PCIe 系统

这种方法一方面提供了用于快速 PCIe 系统配置的解决方案; 另一方面, 由于比特流只能由主机访问, 并且能提供更好的加密, 因此也扩展了用户应用的安全性。这种方法

通过减少外部配置元件的成本和板子空间,也帮助降低系统成本。

### 3. 动态可重配置包处理器

包处理器可以使用部分重配置功能,根据接收到的包类型来快速地改变它的处理功能。如图 7.5 所示,包含着头部,头部包含部分比特文件,或者包含着部分比特文件的一个特殊的包。当处理完部分比特文件后,用于在 FPGA 内重新配置一个协处理器。

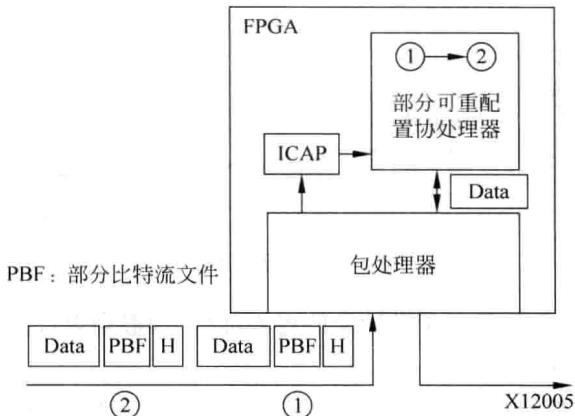


图 7.5 动态可重配置包处理器

### 4. 非对称密钥加密

对于一些新的应用如果没有部分可重配置技术是不可能实现的。通过使用部分重配置和非对称加密,实现用于保护 FPGA 配置文件非常安全的方法。

如图 7.6 所示,黑盒内的所有功能在 FPGA 的物理封装内实现。明文信息和私钥永远不会离开保护容器。

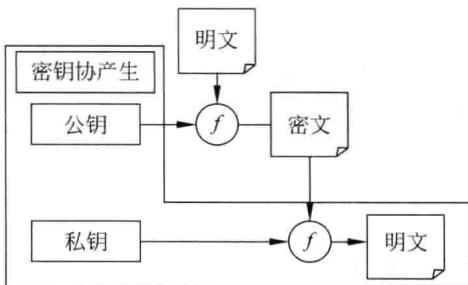


图 7.6 非对称密钥加密

在真正实现这个设计时,初始比特文件是一个未加密的设计,它不包含任何专有的信息。初始设计只包含产生公钥和私钥对的算法,以及主机、FPGA 和 ICAP 之间接口连接。

当加载初始文件后,FPGA 产生公钥和私钥对。公钥发送给主机用来加密一个部分比特文件。如图 7.7 所示,加密的部分比特文件下载到 FPGA 中,在这里被加密,然后送

到 ICAP 用于部分重配置 FPGA。

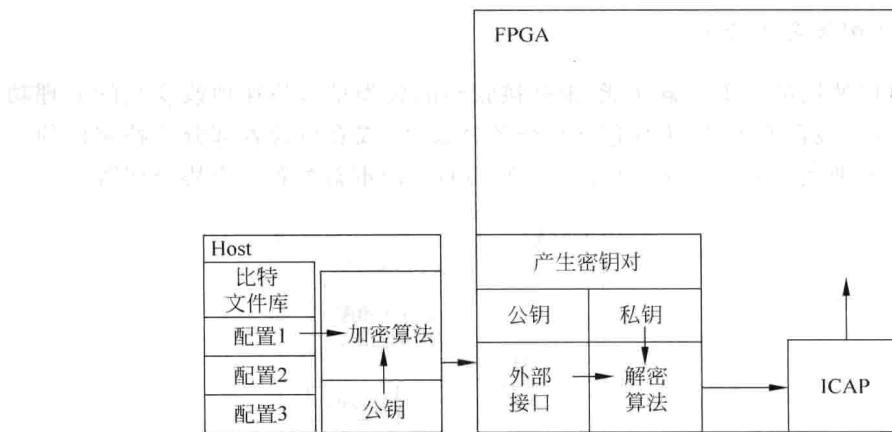


图 7.7 加载一个加密的部分比特文件

部分比特文件可能是 FPGA 设计中的绝大部分,而静态设计只消耗了很少的 FPGA 资源。这个方法有下面几个优势:

- (1) 能在任何时候重新产生公钥和私钥对。如果从主机下载一个新的配置,则可以用不同的公钥加密。如果 FPGA 使用相同的部分比特文件配置,例如从上电复位开始,即使对于一个相同的比特文件也使用一个不同的公钥对。
- (2) 私钥保存在 SRAM 中。如果 FPGA 掉电,则私钥不再存在。
- (3) 即使系统被偷盗,FPGA 仍处于上电状态,找到私钥是一件非常困难的事情,这是因为它保存在 FPGA 的通用结构中。它并不是保存在特殊的寄存器中。设计者能在在一个物理上不相关的区域,手工确定保存私钥的每个寄存器比特位。

### 7.1.3 可重配置的特点

部分可重配置解决了下面三种基本的需求:

- (1) 降低成本和/或者电路板的空间;
- (2) 改变某一区域的设计;
- (3) 降低功耗。

#### 1. 降低成本和电路板的空间

通过重配置技术解决两个用户普遍存在的问题:

- (1) 在一个已经使用的器件中,适配更多的逻辑;
- (2) 在一个更小的、更便宜的器件中,适配一个设计。

历史上,设计者们花费数天时间,如果不是几个星期,尝试新的实现开关,返工设计代码,并重新设计解决方案,将设计装入尽可能小的 FPGA 器件中。通过部分重新配置,设计者就可以动态地时分复用部分可用硬件资源,以减少其设计规模。根据需要加载逻辑功能的能力也减少了空闲逻辑,从而节省了额外的空间。

这一策略的一个例子是在软件无线电(Software Defined Radio, SDR)系统中使用部分可重配置。在SDR中,用户根据要求上传一个新的波形,用于和一个新的通道建立通信。为了使单一的硬件平台可以支持任意数量的波形,要求只有唯一的一部分比特流能用于这些波形。由于部分可重配置的在飞行(on-the-fly)特性,其他已经建立链接的通道不会被上传到另外的通道所破坏。

## 2. 增加已部署系统的灵活性

在过去,改变现场的一个设计,要求设计新的布局和布线,以及交付整个的配置文件。当进行变化时,设计人员不得不将系统断电。与之相比较,使用部分可重配置,设计人员只需要布局和布线所修改的功能,然后将这个新的部分映像交付到系统中。

并且,当系统启动并运行时,设计者可以动态地插入新的功能,改善系统的性能。这样,可以在相同的空间内插入互斥的功能,而不需要重新设计系统或者将设计移动到一个更大的器件。

体现部分可重配置好处的另一个例子是在光纤传输网络中的应用。类似于SDR,支持不同的协议来创建一个更高效的硬件系统,即在FPGA中只加载用于当前在任何一点特定通道的协议。一个已经部署的系统不仅通过使用最小的资源处理更多不同类型的流量,而且能用最新的协议更新系统,并且不需要进行全部的重新设计。

## 3. 降低功耗

对于当前的设计者来说,功耗是最为关心的问题。但是,随着FPGA设计规模和复杂度的增加,功耗也相应增加。而带有设计技巧的综合和实现工具,能帮助降低功耗。通过使用部分重配置实现,更能更加降低静态和动态功耗。

一种降低静态功率的方法是使用较小规模的器件。使用部分可重配置,设计者能将FPGA进行时间片分割,并且能独立地运行设计的一部分。由于不是设计的每个部分都需要100%的时间,因此,可以要求更小的器件或者更少数量的器件来实现设计。

部分可重配置也潜在地降低了操作功耗和静态功耗。例如很多设计必须能高速运行,而最高的性能可能只需要一个很少的时间段。为了降低功耗,设计者能使用部分重配置来暂时移除一个高性能的设计,这个设计是同一设计的低功耗版本,而不使用排他性设计用于最大性能。当系统要求时,设计者可以重新切换到高性能的设计。

这个原理也应用到I/O标准,特别是不要求100%时间的高功耗接口。低电压差分传输(Low Voltage Differential Signaling,LVDS)是一个高功耗的接口(不考虑活动性,这是由于要求高DC电流为接口提供能量)。当不要求最高性能时,设计者能使用部分可重配置将I/O从LVDS变成低功耗接口,例如LVCMOS;而当要求高速传输时,再重新切换到LVDS。

## 4. 其他优点

在一个FPGA内动态地时分复用时间复用硬件的能力,提供了其他方面的优势。部

分重配置提供的其他优势包括：

- (1) 在选择用于一个应用的算法或者协议时,提供了实时的灵活性;
- (2) 在设计一个安全性时,可以使用新的技术;
- (3) 改善 FPGA 的故障容错能力;
- (4) 促进可配置计算;
- (5) 降低存储比特流文件的要求。

#### 7.1.4 可重配置术语解释

为了后续章节的描述方便,下面对可重配置技术中所涉及到的一些术语进行说明:

(1) Top-down synthesis(自顶向下的综合,不用于部分可重配置):

- ① 一个综合工程,该工程综合平面设计用于优化;
- ② 经常称之为平面综合;
- ③ 不支持层次化实现。

(2) Bottom-up synthesis(自底向上综合):

- ① 独立综合工程,产生多个网表;
- ② 自底向上综合要求为每个分区分配一个独立的网表;
- ③ 没有跨越边界的优化。这样,可以对设计的每个分区进行独立地综合;
- ④ 对带有用于分区黑盒的顶层逻辑进行综合。

(3) Configuration(配置):

一个配置是一个完整的设计,包括:

- ① 静态逻辑;
- ② 用于每个可配置分区的一个可配置的模块。

在一个部分重配置 FPGA 工程中,可以有多个配置。每个配置生成一个充分的比特流文件,以及用于每个可配置模块的一个部分比特流文件。

(4) Configuration Frame(配置帧):

配置帧是 FPGA 配置存储空间中最小可寻址的段。通过这些分散的最底层元件,构建配置帧。在 7 系列器件中,基本的可重配置的帧的大小为一个元件(CLB、BRAM、DSP)宽度和一个时钟区域高度。

(5) Internal Configuration Access Port(内部配置访问端口):

内部配置访问端口是 SelectMAP 接口的一个内部版本。

(6) Partial Reconfiguration(部分重配置):

部分重配置是正在修改当前运行的一个 FPGA 设计,通过下载一个部分比特流实现这个修改。

(7) Partition(分区):

一个分区是设计的一个逻辑部分,由设计者在一个层次中进行定义。这个定义用于设计重用。一个分区是一个新的实现或者是先前保留的一个实现。一个被保护的分区

其功能和实现都是一致的。

(8) Partition Pins(分区引脚):

一个分区引脚是在静态和可重配置逻辑之间的逻辑和物理的连接。

(9) Static Logic(静态逻辑):

所有设计内的逻辑是不可重配置的,它不是可重配置分区的一部分。当对可重配置分区进行重新配置时,静态逻辑总是活动的。

(10) Static Design(静态设计):

静态设计是设计的一部分,它在部分重配置的过程中不会变化。静态设计包括顶层和没有定义为可重配置的所有模块。静态设计由静态逻辑和静态布线所建立。

(11) Reconfigurable Module(可重配置模块,RM):

在可重配置分区内,一个 RM 是网表或者 HDL 描述的实现。每个可配置分区内可以有多个可重配置的模块。

(12) Reconfigurable Partition(可重配置分区,RP):

RP 是在一个例化中的属性设置,定义了该实例是可重配置的。PR 是设计层次中的一个层次,在这个层次上可以使用不同的 RM 进行实现。TCL 命令,例如: opt\_design、place\_design 和 route\_design,用于检测实例上的 HD.RECONFIGURABLE 属性,并且正确地处理它。

### 7.1.5 可重配置的要求

可重配置的要求包含以下:

- (1) 可重配置要求使用 Vivado 2013.3 或者更新的版本。
- (2) Vivado 2013.3 可重配置支持 Kintex-7、Virtex-7 和 XT(包含 7V2000T 和 7VX1140T),以及四个 ZYNQ-7000 器件,即: 7Z045、7Z030、7Z020 和 7Z010。将在以后的版本中支持其他 7 系类器件。
- (3) 只通过 Tcl 或者命令行支持 PR。目前,并不支持工程模式。
- (4) 要求为每个元素类型进行布局规划,用于定义可重配置的区域。
- (5) 设计者负责自底向上的综合,以及管理 RM 网表文件。
  - ① 禁止 I/O 插入,用于创建可重配置模块网表;
  - ② Vivado 综合使用脱离上下文的模块综合流程,用于对 RM 的综合。
- (6) 支持标准的时序约束和额外的时序规划能力。
- (7) 建立一个唯一的 DRC 集,用于引导设计者在一个成功的布线路径上完成设计。
- (8) 一个 PR 设计必须考虑 PR 的初始化和传递部分比特流文件,或者在 FPGA 内,或者作为系统设计的一部分。

### 7.1.6 可重配置的标准

可重配置的标准如下:

- (1) 一些元件类型可以重配置,但是一些不可以重配置:

① 可重配置的资源包括：CLB、BRAM 和 DSP 元件类型，以及布线资源；  
 ② 时钟和时钟正在修改的逻辑（包括 BUFG、BUFR、MMCM、PLL 和类似的元件）不可以重配置，必须驻留在静态区域内。

(2) 下面的元件不可以重配置，必须驻留在静态区域内：

- ① I/O 和 I/O 相关的元件(ISERDES、OSERDES、IDELAYCTRL 等)；
- ② 串行收发器(MGT)和相关的元件；
- ③ 单个的结构特性元件(例如 BSCAN、STARTUP、XADC 等)，必须保留在设计的静态区域。

(3) 当使用元件实现 IP 时，IP 的使用也会受到限制：

- ① Vivado 调试集线器(BSCAN 和 BUFG)；
- ② 带有嵌入式全局缓冲区或者 IP 的 IP 模块；
- ③ MIG 控制器(MMCM)。

(4) 必须对可重配置模块进行初始化，以保证重配置后的初始条件。通过一个本地复位或者一个专用的 GSR 事件(通过选择 RESET\_AFTER\_RECONFIG 特性)，设计者手工实现该要求。

(5) 推荐去耦合逻辑。这样，在部分重配置期间，将设计中的静态部分和可重配置区域的连接断开。可以将可重配置模块的时钟和其他输入去耦合，用于阻止在重配置期间对存储器虚假的写操作。

(6) 必须对一个可重配置区域进行布局规划。这样，模块必须是一个由 Pblock 保留的模块，并且满足时序的要求。如果完成该模块，推荐通过使用非 PR 流程运行这个设计，以得到一个对布局、布线和时序结果的初始评估。如果设计是通过非 PR 流程发布，则在移动到 PR 流程前，必须解决这些问题。

在一个 RP 中的每个引脚，有一个分区引脚。这是一个布线点，用于将静态逻辑连接到 RP 上。如果一个设计有过多的分区引脚用于可用的布线资源，可能会发生布线阻塞。

(7) Virtex-7 SSI 器件(7V2000T, 7VX1140T)有两个基本的要求。当可用时，这些要求将扩展到 Virtex-7 HT 器件。

① 在一个单个的 SLR 中，必须充分包含可重配置的区域。这确保全局复位事件和 RM 内的所有元素同步，所有的超长连线(Super Long Line, SLL)包含在设计的静态部分。SLL 是不可重配置的。

② 如果使用 ICAP 发送部分比特流，必须位于主 SLR，它是这些器件的 SLR1。在 ICAP 上应用位置约束，将其约束到 ICAP\_X0Y2 或者 ICAP\_X0Y3。比特流的格式是这样的，即：贯穿四个 SLR 的标准菊花链即使可重配置区域位于这个地方，也不要在其他 SLR 上使用一个 ICAP。

(8) 在将来发布的 Vivado 中，自然支持用于对 7 系列器件的部分比特流的专用加密支持。

(9) 在将来发布的 Vivado 中，7 系列器件将使用每帧 CRC 检查机制，使能通过 write\_bitstream，确保在下载前每帧都有效。

### 7.1.7 可重配置的流程

Vivado 部分重配置设计流程类似于一个标准的设计流程,但是有一些显著的特点。实现软件自动地管理底层的细节,用于满足硅片的要求。设计者必须提供引导,用于定义设计结构和规划布局。处理 PR 设计的步骤如下:

- (1) 独立地综合静态模块和可重配置的模块;
- (2) 创建物理约束(Pblock),用于定义可重配置的区域;
- (3) 在每个可重配置的分区上,设置 HD.RECONFIGUTABLE 属性;
- (4) 实现一个完整的设计(静态设计和每个 RP 有一个 RM);
- (5) 保存一个设计检查点,用于充分地布线设计;
- (6) 从这个设计中,去除 RM,只保存静态设计检查点;
- (7) 锁定静态布局和布线;
- (8) 添加新 RM 到静态设计,实现这个新的设计;
- (9) 重复第(8)步,直到实现所有的可重配置模块;
- (10) 对所有的配置,运行验证工具(pr\_verify);
- (11) 为每个配置创建比特流。

## 7.2 可重配置的实现

本节将使用 led\_shift\_count 设计来说明可重配置的实现。该设计使用 Xilinx 提供的 KC705 开发板,其器件使用的是 xc7k325t。该设计比较简单,用于帮助读者快速掌握可重配置的基本流程。取决于 Vivado 的版本,读者可以将该设计移植到其他 7 系列器件上。

该设计在本书所提供设计资料的 vivado\_example\led\_shift\_count 目录下。读者可以进行参考。

### 7.2.1 查看脚本

看到在 led\_shift\_count 目录下,有两个 Tcl 脚本文件: design.tcl 和 design\_complete.tcl。这两个文件包含相同的信息,但是 design.tcl 的参数设置中,只有综合运行,而 design\_complete.tcl 为两个配置运行整个的设计流程。

在文本编辑器中,打开 design.tcl 文件。这是主脚本,定义了设计参数、设计源文件和设计结构。这是读者可以修改的唯一文件,它用于编译一个完整的部分重配置设计。下面对该文件的一些细节进行说明:

- (1) 使用 set\_param hd.visual 1(第 6 行,在 run.tcl 中调用 set\_param 命令),要求脚本可见。这个命令在根目录下创建脚本,在设计后面用于标识部分比特流中所包含的帧。
- (2) 在 flow control 下,读者可以控制运行综合和实现的不同阶段。在该设计中,脚

本只运行综合,实现、验证和通过交互式方式运行生成比特流。如果通过脚本运行这些额外的步骤,则设置流变量(例如 run, prImpl)为1。

(3) Output Directories 和 Input Directories 部分,为设计源文件和结果文件设置所希望的文件结构。读者可以修改相关的设置。

(4) Top Definition 和 RP Module Definitions 部分,允许设计者为设计的每个部分引用所有的源文件。Top Definition 覆盖了用于静态设计所需要的所有源文件,包含约束和 IP。RP Module Definitions 用于可重配置分区所需要的所有源文件。为每个 RP 完成一个部分,并且列出用于每个 RP 的所有可配置模块的变量。

**注:** 该设计有两个可重配置的分区(inst\_shift 和 inst\_count),每个 RP 有两个模块变量。

(5) Configuration Definition 部分,定义了构成一个配置的静态和可重配置模块集合。

**注:** 该设计有两个配置 Config\_shift\_right\_count\_up 和 Config\_shift\_left\_count\_down。读者可以通过添加 RM 或者组合已经存在的 RM,来创建更多的配置。

此外,在 led\_shift\_count\Tcl 子目录下,还存在着所支持的一些 Tcl。通过 design.tcl,调用这些脚本。这些脚本用于管理部分重配置流程中指定的细节。下面对这些脚本进行了说明:

**注:** 读者不要修改这些脚本。

- ① step.tcl: 通过监视检查点,管理设计的当前状态;
- ② synth.tcl: 管理综合阶段的所有细节;
- ③ impl.tcl: 管理实现阶段的所有细节;
- ④ pr\_impl.tcl: 管理一个 PR 设计顶层实现的所有细节;
- ⑤ run.tcl: 启动综合和实现真正的运行;
- ⑥ log.tcl: 在流程处理期间,管理在关键点上所创建的报告文件。

剩余的脚本在这些脚本中(例如 \*\_utils.tcl)提供了细节,或者管理其他层次设计流程(例如 ooc\_impl.tcl)。

## 7.2.2 综合设计

design.tcl 自动地综合这个设计。在这个过程中,调用五个迭代过程,一个用于静态的顶层设计,剩下的四个用于每个可重配置的模块。综合设计的步骤主要包括:

- (1) 在 Window 7 操作系统主界面下,选择开始→所有程序→Xilinx Design Tools→Vivado 2013.3→Vivado 2013.3 Tcl Shell。
- (2) 在 Tcl Shell 窗口下,将路径修改为 led\_shift\_count。
- (3) 输入下面的命令,运行 design.tcl。

```
source design.tcl -notrace
```

当使用 Vivado 综合实现这五步后,仍然打开 Vivado Tcl Shell。此外,在 Synth 子目录下,位于每个命名的文件夹内,读者可以看到每个模块的日志文件和报告文件,以及最

终的检查点。

**注：**在当前文件夹下，创建了多个日志文件。其中：

- ① run.log 列出了在 Tcl Shell 窗口下给出的总结；
- ② command.log 回应了脚本运行的每个步骤；
- ③ critical.log 在运行的过程中，报告所有严重的警告。

### 7.2.3 实现第一个配置

现在用于顶层和每个模块的综合后的检查点可用。设计者可以对设计进行组装。由于没有提供针对部分重配置流程的工程支持，所以不能使用 IDE 工程。

读者可以从 Tcl 窗口下运行所有的流程步骤。但是，也可以使用 IDE 内的特性（例如 floorplanning 工具）交互事件。

#### 1. 实现设计

实现设计的步骤主要包括：

(1) 通过下面两种方式，打开 Vivado 集成开发环境。

- ① 在 Tcl Shell 窗口中，输入 start\_gui 命令；
- ② 在命令行，输入 vivado-mode gui 命令。

(2) 定位到 vivado\_example\led\_shift\_count\_completed 文件夹。如果不知道当前路径，可以在 Vivado 主界面的 Tcl Console 窗口下，输入 pwd 命令，查看当前路径。

(3) 在 Vivado 主界面的 Tcl Console 窗口下，输入下面的命令，该命令用于加载静态设计：

```
open_checkpoint Synth/Static/top_synth.dcp
```

如图 7.8 所示，在 Vivado 主界面的 Netlist 标签窗口下，可以看到设计结构。但是，注意到 inst\_shift 和 inst\_count 模块是两个黑盒。

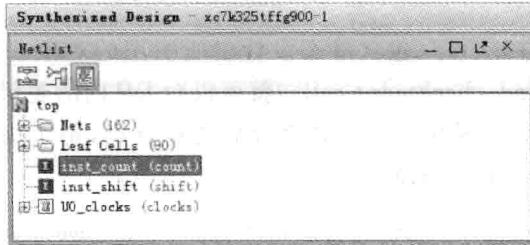


图 7.8 综合后的设计

**注：**

- ① 在 Vivado 主界面左侧没有出现 Flow Navigator 窗口。此时，工作在非工程模式下。
- ② 如图 7.9 所示，出现两个严重警告信息，提示没有匹配的实例。这些实例是可重配置的模块，但是可以加载。因此，可以放心地忽略这些警告信息。

```

Tcl Console
INFO: [Memdata 29-14] Successfully populated the EEPROM INIT strings from the following elf files:
CRITICAL WARNING: [Project 1-086] Could not resolve non-primitive block box cell 'shift' instantiated as 'inst_shift' [E:/vivado_Example/Led_Shift_Count_Completed/Sources/Edk1/top/top.v:92]
CRITICAL WARNING: [Project 1-086] Could not resolve non-primitive block box cell 'count' instantiated as 'inst_count' [E:/vivado_Example/Led_Shift_Count_Completed/Sources/Edk1/top/top.v:100]
INFO: [Project 1-111] Unalias transformation Summary:
No Unalias elements were transformed

```

图 7.9 严重敬告信息

(4) 在 Vivado 主界面的 Tcl Console 窗口下,输入下面的命令:

```
read_xdc Sources/xdc/top.xdc
```

该命令用于加载顶层约束文件。该文件设置器件的引脚和顶层时序约束,以及为每个可配置的分区创建 Pblock。如图 7.10 所示,在 Device 视图窗口下,为两个可重配置的分区创建了两个 Pblock。不能从 Vivado 集成开发环境访问 XDC 文件,因为它不能作为设计源文件显示出来。

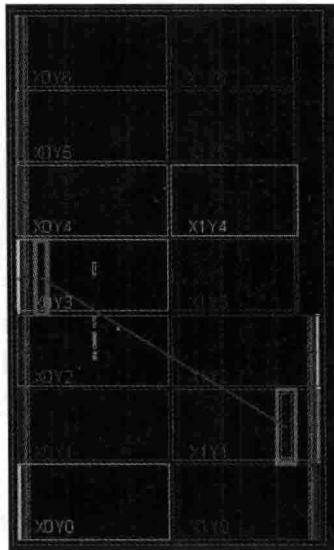


图 7.10 top.xdc 初始布局规划

这个顶层 XDC 文件,应该只包含静态设计中所引用的对象。一旦将 RM 加载到设计中(在下一步,运行 read\_checkpoint-cell),就可以对 RP 内的逻辑或者网络应用约束。

(5) 为第一个可配置的模块变量加载综合后的检查点,用于每个可配置的分区。在 Tcl Console 窗口下,输入下面的命令:

```
read_checkpoint - cell inst_shift Synth/shift_right/shift_synth.dcp
read_checkpoint - cell inst_count Synth/count_up/count_synth.dcp
```

**注:** 如图 7.11 所示,逻辑资源已经填充了 inst\_shift 和 inst\_count 模块。

(6) 通过设置 HD.RECONFIGURABLE 属性,将每个子模块定义为部分可重配。在 Vivado Tcl Console 窗口下,输入下面的命令:

```
set_property HD.RECONFIGURABLE 1 [get_cells inst_shift]
set_property HD.RECONFIGURABLE 1 [get_cells inst_count]
```

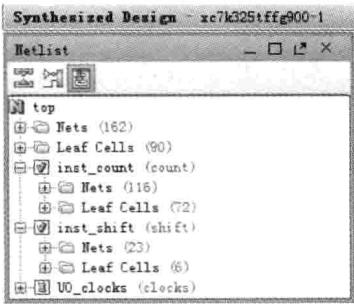


图 7.11 逻辑资源已经填充了黑盒

注：

- ① 当输入命令后, Vivado 会检查部分重配置的 license 文件。如果 license 有效, 则看到下面的信息:

```
Feature available: PartialReconfiguration
```

如果没有有效的 PartialReconfiguration 的 license 文件, 则可以通过 Xilinx 官网申请 30 天免费的评估版本的 license 文件。

② 在该点, 加载了所有的设计, 并且为 PR 设置了所有属性。下面可以实现设计。

(7) 运行设计规则检查(Design Rule Checks, DRC), 确保已经满足了部分重配置的要求。在 Vivado 主界面主菜单下, 选择 Tools→Report→Report DRC 打开可用的 DRC 设置。如图 7.12 所示, 读者可以先单击 Clear All 按钮, 不选中所有选项前面的复选框。然后, 再选择 Partial Reconfiguration 前面的复选框。

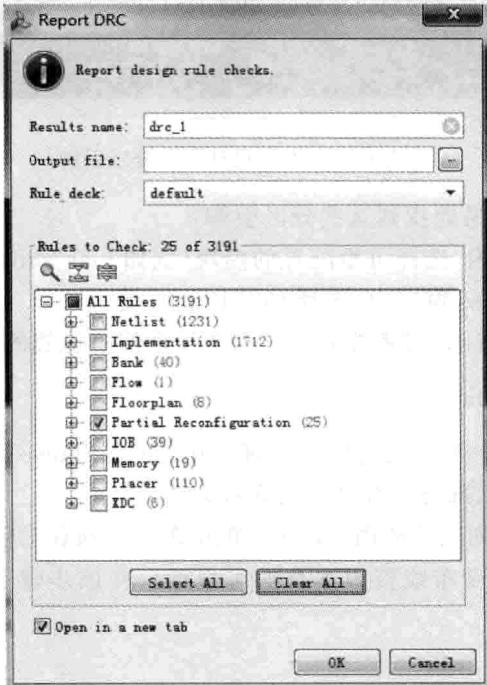


图 7.12 报告 DRC 对话框界面

```

Tcl Console
Finished Parsing IDC File [e:/vivado_example/led_shift_count_completed/ Xil/Vivado-18512-hebin/PC/dcp/top.xdc]
INFO [Mendate 2S-14] Successfully populated the SRAM LHM strings from the following elf files:
CRITICAL WARNING [Project 1-486] Could not resolve non-primitive black box cell 'shift' instantiated as 'inst_shift' [E:/vivado_example/led_shift_count_completed/Sources/hdl/top.v#0];
CRITICAL WARNING [Project 1-486] Could not resolve non-primitive black box cell 'count' instantiated as 'inst_count' [E:/vivado_example/led_shift_count_completed/Sources/hdl/top.v#0];
INFO [Project 1-111] Unisim Transformation Summary:
No Unisim elements were transformed

```

图 7.9 严重敬告信息

(4) 在 Vivado 主界面的 Tcl Console 窗口下,输入下面的命令:

```
read_xdc Sources/xdc/top.xdc
```

该命令用于加载顶层约束文件。该文件设置器件的引脚和顶层时序约束,以及为每个可配置的分区创建 Pblock。如图 7.10 所示,在 Device 视图窗口下,为两个可重配置的分区创建了两个 Pblock。不能从 Vivado 集成开发环境访问 XDC 文件,因为它不能作为设计源文件显示出来。

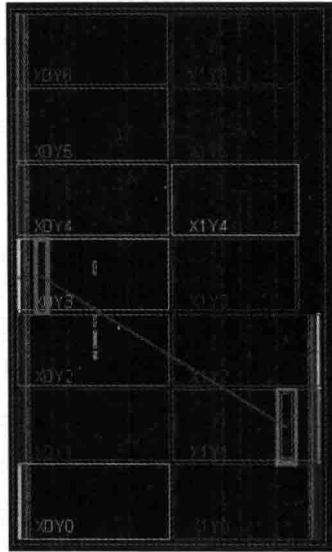


图 7.10 top.xdc 初始布局规划

这个顶层 XDC 文件,应该只包含静态设计中所引用的对象。一旦将 RM 加载到设计中(在下一步,运行 read\_checkpoint-cell),就可以对 RP 内的逻辑或者网络应用约束。

(5) 为第一个可配置的模块变量加载综合后的检查点,用于每个可配置的分区。在 Tcl Console 窗口下,输入下面的命令:

```
read_checkpoint - cell inst_shift Synth/shift_right/shift_synth.dcp
read_checkpoint - cell inst_count Synth/count_up/count_synth.dcp
```

**注:** 如图 7.11 所示,逻辑资源已经填充了 inst\_shift 和 inst\_count 模块。

(6) 通过设置 HD.RECONFIGURABLE 属性,将每个子模块定义为部分可重配。在 Vivado Tcl Console 窗口下,输入下面的命令:

```
set_property HD.RECONFIGURABLE 1 [get_cells inst_shift]
set_property HD.RECONFIGURABLE 1 [get_cells inst_count]
```

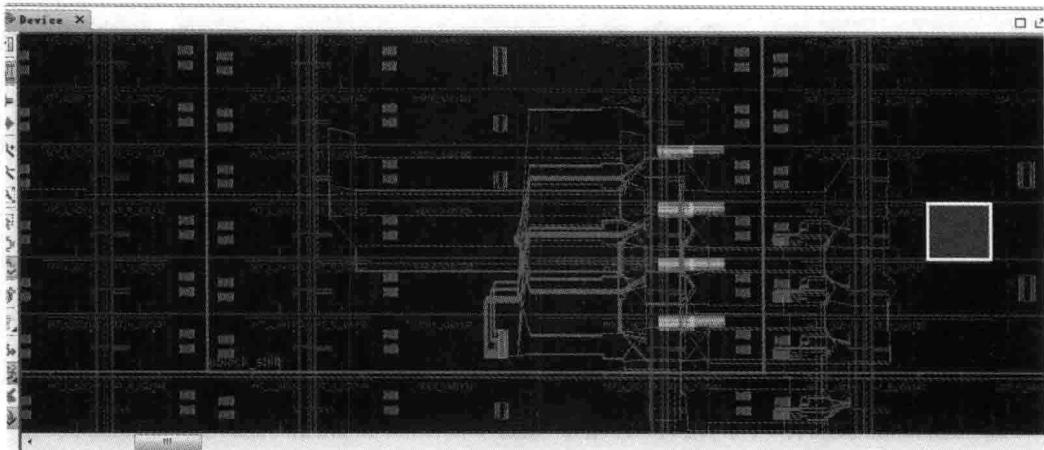


图 7.14 第一个配置的布线视图

## 2. 保存设计

本节将保存设计。保存设计的步骤主要包括：

(1) 在 Tcl Console 窗口下,输入下面的命令：

```
write_checkpoint -force Implement/Config_shift_right_count_up/top_route_design.dcp
report_utilization -file Implement/Config_shift_right_count_up/top_utilization.rpt
report_timing_summary -file Implement/Config_shift_right_count_up/top_timing_summary.rpt
```

(2) 输入下面的命令,保存每个可配置模块的检查点(可选的)：

```
write_checkpoint -force -cell inst_shift Checkpoint/shift_right_route_design.dcp
write_checkpoint -force -cell inst_count Checkpoint/count_up_route_design.dcp
```

**注：**当运行 design\_complete.tcl 时,以批处理方式处理整个设计,在流程的每一步创建设计检查点、日志文件和报告文件。

在该点,读者已经创建了一个充分实现的部分重配置设计。从该点,读者可以创建充分的和部分的比特流。这个配置的静态部分用于所有随后的配置。为了和静态设计相隔离,必须去掉当前可配置的模块。

(3) 使用下面的 Tcl 命令,创建时序和利用率报告。

```
report_utilization -file Implement/Config_shift_right_count_up/top_utilization.rpt
```

(4) 创建一个在 IDE 环境中可以使用的交互时序总结(可选的)。输入下面的命令,将在 IDE 中创建一个时序报告,该报告能交叉探测到 Device 和 Netlist 窗口。

```
report_timing_summary -name timing_1
```

如图 7.15 所示,自动打开时序总结对话框界面。

(5) 确认使能布线资源,放大到带有分区引脚的一个互连单元。

(6) 输入下面的 Tcl 命令,清除可重配置模块逻辑：

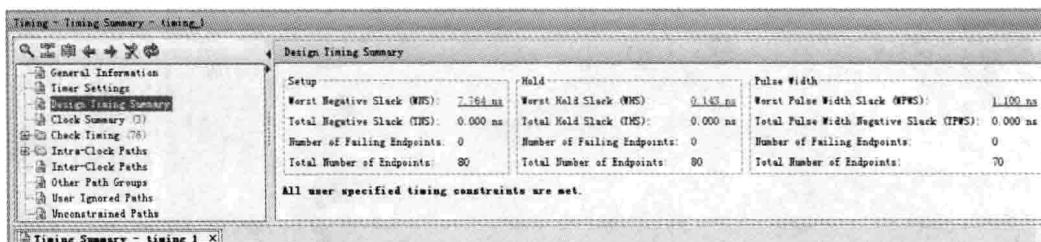


图 7.15 时序总结对话框界面

```
update_design -cell inst_shift -black_box
update_design -cell inst_count -black_box
```

输入这些命令后,清空出现在 Netlist 视图中的 inst\_shift 和 inst\_count 模块。

(7) 输入下面的 Tcl 命令,写入到只包含静态的检查点:

```
write_checkpoint -force Checkpoint/static_route_design.dcp
```

这个只包含静态的检查点用于将来的配置。但是,在该设计中,读者只是简单地在存储器中保持这个设计打开。

## 7.2.4 实现第二个配置

已经建立了静态设计结果,读者将它作为上下文使用,用于更深入地实现可重配置的模块。读者必须锁定这些结果,以确保在实现新的 RM 时,不会对该模块有任何的修改。

### 1. 实现设计

本节将实现设计。实现设计的步骤主要包括:

(1) 由于在存储器中已经加载布线后的静态设计,所以,输入下面的 Tcl 命令,用于锁定所有的布局和布线。

```
lock_design -level routing
```

由于 lock\_design 命令没有标识单元,将影响存储器中的整个设计(当前由带有黑盒的静态设计构成)。如图 7.16 所示,现在显示了所有布线后的网络,用虚线表示。充分布线的网络用绿色显示,部分的布线用黄色显示。

(2) 读取用于其他两个重配置模块的综合后的检查点:

```
read_checkpoint -cell inst_shift Synth/shift_left/shift_synth.dcp
read_checkpoint -cell inst_count Synth/count_down/count_synth.dcp
```

(3) 输入下面的 Tcl 命令,在静态逻辑的上下文中,优化、布局和布线新的 RM。

```
opt_design
place_design
route_design
```

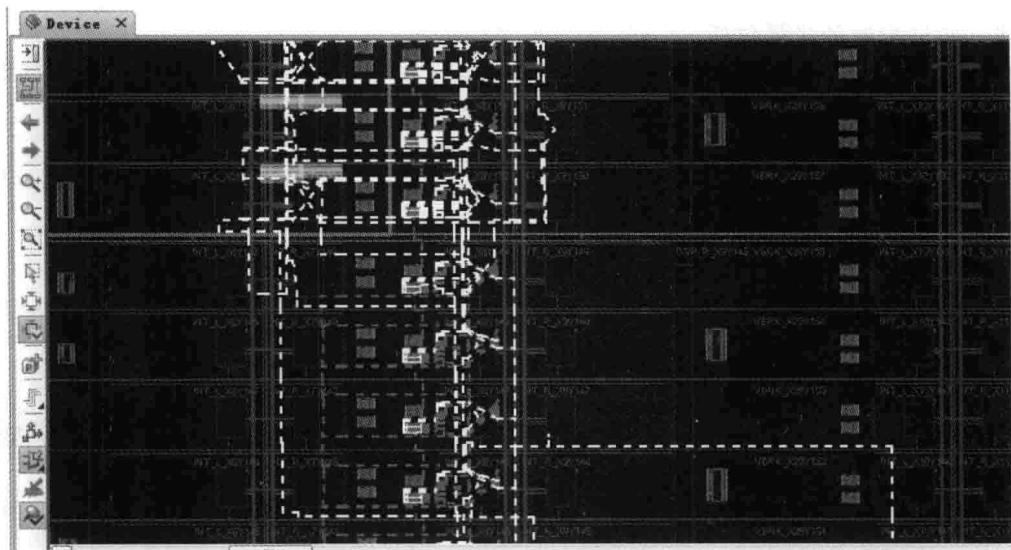


图 7.16 锁定布局后的静态设计

这样,又再次充分地实现了设计,现在带有新的重配置模块变量。所有的布线变成绿色,是虚线(锁定)和实线(新的)布线段的混合。

## 2. 保存设计

本节将保存设计。保存设计的步骤主要包括:

(1) 在 Tcl Console 窗口下,输入下面的命令:

```
write_checkpoint - force Implement/Config_shift_left_count_down/top_route_design.dcp
report_utilization - file Implement/Config_shift_left_count_down/top_utilization.rpt
report_timing_summary - file Implement/Config_shift_left_count_down/top_timing_summary.rpt
```

(2) 输入下面的命令,保存每个可配置模块的检查点(可选的):

```
write_checkpoint - force - cell inst_shift Checkpoint/shift_left_route_design.dcp
write_checkpoint - force - cell inst_count Checkpoint/count_down_route_design.dcp
```

在该点,读者已经实现了静态逻辑和所有可重配置模块变量。

(3) 在 Vivado 主界面下的 Tcl Console 窗口下,输入:

```
close_project
```

关闭当前的设计工程。

## 7.2.5 验证配置

在生成比特流之前,对所有的配置进行验证,以保证每个配置的静态部分有相同的匹配。这样,可以在 FPGA 上正确地使用这些配置。PR 验证特性检查完整的静态设计,其中也包括分区引脚,以确认它们的一致性。但是,并不检查可重配置模块内的布局和

布线。验证配置的步骤包括：

在 Tcl Console 窗口下,输入下面的命令：

```
pr_verify Implement/Config_shift_right_count_up/top_route_design.dcp
Implement/Config_shift_left_count_down/top_route_design.dcp
```

如果成功的话,这个命令返回下面的信息：

```
INFO: [Project 1 - 484] Checkpoint was created with build 329390
INFO: [Vivado 12 - 3253] PR_VERIFY: check points
Implement/Config_shift_right_count_up/top_route_design.dcp and
Implement/Config_shift_left_count_down/top_route_design.dcp are compatible
```

默认地,如果存在任何一个不匹配时,只报告第一个不匹配。使用-full\_check 查看所有的不匹配信息。

## 7.2.6 生成比特流

本节将生成比特流,随后将这些比特流下载到 KC705 目标板上。生成比特流的步骤主要包括：

(1) 在 Tcl Console 窗口下面,输入下面的 Tcl 命令：

```
open_checkpoint Implement/Config_shift_right_count_up/top_route_design.dcp
```

将第一个配置读入存储器。

(2) 在 Tcl Console 窗口下,输入下面的 Tcl 命令：

```
write_bitstream -file Bitstreams/Config_RightUp.bit
```

为该设计生成充分的和部分的比特流。

**注：**保证比特流文件存放在一个合适的目录下。

(3) 在 Tcl Console 窗口下,输入下面的 Tcl 命令：

```
close_project
```

关闭当前的工程。

**注：**已经生成了三个比特流文件：

- ① Config\_RightUp.bit：这是一个上电后,充分设计比特流文件；
- ② Config\_RightUp\_pblcok\_shift\_partial.bit：这是用于 shift\_right 模块的一个部分比特流文件；
- ③ Config\_RightUp\_pblcok\_count\_partial.bit：这是用于 count\_up 模块的一个部分比特流文件。

当前的比特流文件的名字不反映可重配置模块变量的名字,只是用于区分所加载的映像文件。当前的解决方案使用了-file 选项给出的名字,并且附加了可重配置单元 Pblock 的名字。关键是要提供名字的足够描述,用于清楚地区分可重配置比特流文件。所有的部分比特流文件都带有\_partial 后缀。

(4) 在 Tcl Console 窗口下, 输入下面的 Tcl 命令:

```
open_checkpoint Implement/Config_shift_left_count_down/top_route_design.dcp
```

将第二个配置读入存储器。

(5) 在 Tcl Console 窗口下, 输入下面的 Tcl 命令:

```
write_bitstream -file Bitstreams/Config_LeftDown.bit
```

为该设计生成充分的和部分的比特流。

**注:** 保证比特流文件存放在一个合适的目录下。类似地,也生成了三个比特流文件。

(6) 在 Tcl Console 窗口下, 输入下面的 Tcl 命令:

```
close_project
```

关闭当前的工程。

(7) 生成一个带有黑盒的充分比特流, 为可重配置模块分配了空白的比特流。空白的比特流用于擦除已存在的配置, 降低系统功耗。在 Tcl Console 窗口下, 输入下面的 Tcl 命令:

```
open_checkpoint Checkpoint/static_route_design.dcp
```

将第三个配置读入存储器。

(8) 在 Tcl Console 窗口下, 输入下面的 Tcl 命令:

```
write_bitstream -file Bitstreams/blanking.bit
```

**注:** 类似地, 也生成了三个比特流文件。

(9) 在 Tcl Console 窗口下, 输入下面的 Tcl 命令:

```
close_project
```

关闭当前的工程。

### 7.2.7 部分重配置 FPGA

#### 1. 使用充分映像配置器件

本节将使用充分映像配置器件。使用充分映像配置器件的步骤主要包括:

(1) 通过 USB 电缆, 将 KC705 目标系统和调试主机电脑连接。

(2) 给 KC705 目标系统上电。

(3) 在 Vivado 集成开发环境主界面主菜单下, 选择 Flow → Open Hardware Manager。

(4) 在 Hard Manager 窗口下, 单击 Open a new hardware target。按照前面的方法建立目标板的 JTAG 链路。

(5) 选择 XC7K325T\_0, 单击鼠标右键, 定位到生成比特流的文件夹下, 选择 Config\_RightUp.bit 文件。

(6) 单击 OK 按钮, 对 FPGA 进行编程。

此时,可以看到 GPIO LED 组执行两个任务。其中,四个 LED 执行向上计数功能(最左是 MSB),而其他四个 LED 执行向右移操作。

注:请注意观察配置器件所用的时间。

## 2. 使用部分映像配置器件

本节将使用前面创建的任何比特流文件对 FGPA 进行编程。使用部分映像配置器件的步骤主要包括:

(1) 浏览到生成比特流的文件夹下,选择 Config\_LeftDown\_pblock\_shift\_partial.bit 文件,对 FPGA 编程。

注:LED 的移位部分改变了方向,但是计数器仍然向上计数,重配置并没有影响它。并且注意到配置时间很短。

(2) 浏览到生成比特流的文件夹下,选择 Config\_LeftDown\_pblock\_count\_partial.bit 文件,对 FPGA 编程。

注:LED 的计数部分开始向下计数,而移位部分没有发生任何变化,重配置并没有影响它。并且注意到配置时间很短。

# 第8章 Vivado 高级设计技术

本章将介绍 Vivado 集成开发环境中的高级设计技术。内容包括：Vivado 支持的属性、增量设计、修改布线和逻辑、布局约束、查看和分析时序报告，以创建时序约束。

本章内容介绍了 Vivado 所提供的各种高级设计方法。通过本章内容的学习，读者将进一步掌握 Vivado 集成开发环境的高级设计技术。

## 8.1 Vivado 支持的属性

本节将详细介绍 Vivado 支持的各种属性。

### 8.1.1 ASYNC\_REG

该选项用于通知 Vivado，D 输入引脚能够接收相对于时钟源的异步数据，或者说寄存器是一个同步链上正在同步的寄存器。这个属性可以放置在任何寄存器上，其值为 FALSE 或者 TRUE。

(1) ASYNC\_REG Verilog 例子：

```
(* ASYNC_REG = "TRUE" *) reg [2:0] sync_regs;
```

(2) ASYNC\_REG VHDL 例子：

```
attribute ASYNC_REG : string;
attribute ASYNC_REG of sync_regs : signal is "TRUE";
```

### 8.1.2 BLACK\_BOX

该属性是一个非常有用的调试属性，它用于关闭层次上某一级，使能综合工具可以为该模块或者实体创建黑盒。该属性可以放置在一个模块、实体或者元件上。由于该属性影响综合编译器，所以只能在 RTL 级上设置该属性。

(1) BLACK\_BOX Verilog 例子：

```
(* black_box *) module test(in1, in2, clk, out1);
```

注：不需要值。

(2) BLACK\_BOX VHDL 例子：

```
attribute black_box : string;
attribute black_box of beh : architecture is "yes";
```

### 8.1.3 BUFFER\_TYPE

该属性用于输入，确定输入所使用缓冲区的类型。默认地，对于时钟来说，Vivado 综合工具使用 IBUF/BUFG；而对于其他输入则使用 IBUF。其值可以是 ibuf、ibufg 和 none。它放置在顶层端口，只能在 RTL 级描述中使用。

(1) BUFFER\_TYPE Verilog 例子：

```
(* buffer_type = "none" *) input in1; //this will result in no buffers
(* buffer_type = "ibuf" *) input clk1; //this will result in a clock with no bufg
BUFFER_TYPE
```

(2) VHDL 例子：

```
entity test is port(
    in1 : std_logic_vector (8 downto 0);
    clk : std_logic;
    out1 : std_logic_vector(8 downto 0));
    attribute buffer_type : string;
    attribute buffer_type of in1 : signal is "none";
end test;
```

### 8.1.4 DONT\_TOUCH

使用该属性，用于替换 KEEP 或者 KEEP\_HIERARCHY 属性。其原理和这两个属性一样。然而，不像这两个属性，DONT\_TOUCH 属性是向前注解到布局和布线，以阻止逻辑优化。其取值为 TRUE/FALSE 或者 yes/no。该属性可以放置在信号、模块、实体或者元件上。

(1) DONT\_TOUCH Verilog 例子：

① Wire 例子：

```
(* dont_touch = "true" *) wire sig1;
assign sig1 = in1 & in2;
assign out1 = sig1 & in2;
```

② Module 例子：

```
(* DONT_TOUCH = "true|yes" *)
module example_dt_ver(clk, In1, In2, out1);
```

③ Instance 例子：

```
(* DONT_TOUCH = "true|yes" *) example_dt_ver U0 (.clk(clk), .in1(a), .in2(b), out1(c));
```

## (2) DONT\_TOUCH VHDL 例子：

## ① Signal 例子：

```
signal sig1 : std_logic;
attribute dont_touch : string;
attribute dont_touch of sig1 : signal is "true";
...
...
sig1 <= in1 and in2;
out1 <= sig1 and in3;
```

## ② Entity 例子：

```
entity example_dt_vhd is
port (
    clk : in std_logic;
    In1 : in std_logic;
    In2 : in std_logic;
    out1 : out std_logic
);
attribute dont_touch : string;
attribute dont_touch of example_dt_vhd : entity is "true|yes";
end example_dt_vhd;
architecture rtl of top is
attribute dont_touch : string;
attribute dont_touch of rtl : architecture is "true|yes";
begin
    ...
    ...
    ...
end rtl;
```

**8.1.5 FSM\_ENCODING**

该属性用于控制状态机的编码。典型地，基于启发式的方法，Vivado 工具为状态机选择一个编码协议。在某些指定的编码类型下，这些设计类型可以工作得更好。该属性放置在状态寄存器上。该属性有效的值为 one\_hot、sequential、johnson、gray 和 auto，可以在 RTL 级或者 XDC 中设置该属性。

## (1) FSM\_ENCODING Verilog 例子：

```
(* fsm_encoding = "one_hot" *) reg [7:0] my_state;
```

## (2) FSM\_ENCODING VHDL 例子：

```
type count_state is (zero, one, two, three, four, five, six, seven);
signal my_state : count_state;
attribute fsm_encoding : string;
attribute fsm_encoding of my_state : signal is "sequential";
```

### 8.1.6 FSM\_SAFE\_STATE

当检测到状态机中有非法状态时,该属性告诉综合工具在状态机中插入逻辑。当下一个时钟周期到来时,将其置为一个已知状态。例如,如果 onehot 编码,而进入“0101”状态(对于 onehot 编码,该状态编码无效),则应该能够恢复状态机。该属性放在状态寄存器上,其有效的取值为 reset\_state 或者 power\_on\_state,该属性只能用于 RTL 级。

(1) FSM\_SAFE\_STATE Verilog 例子:

```
(* fsm_safe_state = "reset_state" *) reg [7:0] my_state;
```

(2) FSM\_SAFE\_STATE VHDL 例子:

```
type count_state is (zero, one, two, three, four, five, six, seven);
signal my_state : count_state;
attribute fsm_safe_state : string;
attribute fsm_safe_state of my_state : signal is "power_on_state";
```

### 8.1.7 FULL\_CASE (Verilog Only)

该属性表示在 case、casex 或者 casez 描述中,给出了所有可能情况的取值。如果指定了 case 的值,Vivado 综合工具不能创建额外的逻辑,该逻辑用于 case 值。该属性放置在 case 描述中,由于该属性影响编译器,因此可以改变设计的逻辑行为。所以,只能在 RTL 中设置该属性。

```
(* full_case *)
case select
  3'b100 : sig = val1;
  3'b010 : sig = val2;
  3'b001 : sig = val3;
endcase
```

### 8.1.8 GATED\_CLOCK

Vivado 允许门控时钟的转换。有两项用于执行这个转换:

(1) Vivado GUI 的开关,用于指导工具尝试转换。在 Vivado 主界面左侧 Flow Navigator 窗口下,找到 Synthesis Settings。在选项设置中,将-gated\_clock\_conversion 选项设置为: off、on 或 auto。当选择 auto 时,如果发生下面事件,则进行转换:

- ① gated\_clock 属性设置为 true。
- ② Vivado 综合工具检测到门,并且设置了有效的时钟约束。

该选项让工具做出决策。

(2) 在门控逻辑中,RTL 属性指导工具,帮助确认哪个信号是时钟。该属性放置在时钟信号或者端口上。

(1) GATED\_CLOCK Verilog 例子：

```
(* gated_clock = "true" *) input clk;
```

(2) GATED\_CLOCK VHDL 例子：

```
entity test is port (
    in1, in2 : in std_logic_vector(9 downto 0);
    en : in std_logic;
    clk : in std_logic;
    out1 : out std_logic_vector( 9 downto 0));
    attribute gated_clock : string;
    attribute gated_clock of clk : signal is "true";
end test;
```

### 8.1.9 IOB

IOB 不是综合属性，下游的实现工具可以使用 IOB。这个属性指定是否将寄存器合并入到 I/O 缓冲区中。其值为 true 或者 false，将这个属性放置在需要寄存器的 I/O 上。可以在 RTL 级或者 XDC 中设置该属性。

(1) IOB Verilog 例子：

```
(* IOB = "true" *) reg sig1;
```

(2) IOB VHDL 例子：

```
signal sig1: std_logic;
attribute IOB: string;
attribute IOB of sig1 : signal is "true";
```

### 8.1.10 KEEP

该属性用于阻止优化，即：信号被优化或者被吸收进逻辑块中。该属性告诉 Vivado 综合工具保持所放置的信号，则该信号将出现在网表中。其取值为 true 或者 false。该属性可以放置在信号、寄存器或者 wire 上。Xilinx 推荐在 RTL 中设置该属性。

**注：**该属性不强迫布局和布线工具保持该信号。在这种情况下，使用 DONT\_TOUCH 属性。

(1) KEEP Verilog 例子：

```
(* keep = "true" *) wire sig1;
assign sig1 = in1 & in2;
assign out1 = sig1 & in2;
```

(2) KEEP VHDL 例子：

```
signal sig1 : std_logic;
attribute keep : string;
```

```

attribute keep of sig1 : signal is "true";
...
...
sig1 <= in1 and in2;
out1 <= sig1 and in3;

```

### 8.1.11 KEEP\_HIERARCHY

该属性用于阻止在层次边界的优化,Vivado综合工具尝试保持在 RTL 级所定义的层次,但是由于 QoR 的原因,它会展开它们。如果在实例上放置了该属性,综合工具将保持静态级的逻辑层次。它不能用于那些描述控制三态输出和 I/O 缓冲区的模块。该属性可以放置在实例的模块或者结构级上。只能在 RTL 级使用该属性。

(1) KEEP\_HIERARCHY Verilog 例子:

① On Module:

```
(* keep_hierarchy = "yes" *) module bottom (in1, in2, in3, in4, out1, out2);
```

② On Instance:

```
(* keep_hierarchy = "yes" *)bottom u0 (.in1(in1), .in2(in2), .out1(temp1));
```

(2) KEEP\_HIERARCHY VHDL 例子:

① On Module:

```
attribute keep_hierarchy : string;
attribute keep_hierarchy of beh : architecture is "yes";
```

② On Instance:

```
attribute keep_hierarchy : string;
attribute keep_hierarchy of u0 : label is "yes";
```

### 8.1.12 MAX\_FANOUT

该属性告诉综合工具,限制寄存器和信号的扇出。可以在 RTL 中指定该属性,将其作为工程的一部分,其值为整数。该属性只能用于寄存器和组合信号。可以在 RTL 或者 XDC 中设置该属性。该属性可以覆盖在综合属性设置中-fanout\_limit 的值。

(1) MAX\_FANOUT Verilog 例子:

```
(* max_fanout = 50 *) reg sig1;
```

(2) MAX\_FANOUT VHDL 例子:

```
signal sig1 : std_logic;
attribute max_fanout : integer;
attribute max_fanout : signal is 50;
```

### 8.1.13 PARALLEL\_CASE (Verilog Only)

该属性用来指定以并行结构构建 case 描述。该属性只能用于 RTL 描述中。

```
( * parallel_case * ) case select
  3'b100 : sig = val1;
  3'b010 : sig = val2;
  3'b001 : sig = val3;
endcase
```

### 8.1.14 RAM\_STYLE

该属性用于帮助 Vivado 综合工具推断存储器的实现方式。可用的值为 block 或者 distributed。可以在 RTL 或者 XDC 中设置该属性。

(1) RAM\_STYLE Verilog 例子：

```
( * ram_style = "distributed" * ) reg [data_size - 1:0] myram [2 * * addr_size - 1:0];
```

(2) RAM\_STYLE VHDL 例子：

```
attribute ram_style : string;
attribute ram_style of myram : signal is "distributed";
```

### 8.1.15 ROM\_STYLE

该属性用于帮助 Vivado 综合工具推断 ROM 存储器的实现方式。可用的值为 block 或者 distributed。可以在 RTL 或者 XDC 中设置该属性。

(1) ROM\_STYLE Verilog 例子：

```
( * rom_style = "distributed" * ) reg [data_size - 1:0] myrom [2 * * addr_size - 1:0];
```

(2) ROM\_STYLE VHDL 例子：

```
attribute rom_style : string;
attribute rom_style of myrom : signal is "distributed";
```

### 8.1.16 SHREG\_EXTRACT

该属性用于帮助 Vivado 工具是否推断 SRL 结构,可接受的值是 yes 或者 no。该属性放在用于 SRL 的信号上或者带有 SR(置位/复位)的模块或者实体上。可以在 RTL 或者 XDC 中设置该属性。

(1) SHREG\_EXTRACT Verilog 例子：

```
( * shreg_extract = "no" * ) reg [16:0] my_srl;
```

## (2) SHREG\_EXTRACT VHDL 例子：

```
attribute shreg_extract : string;
attribute shreg_extract of my_srl : signal is "no";
```

**8.1.17 SRL\_STYLE**

该属性告诉综合工具,如何推断在设计中发现的 SR(置位/复位)。可用的值为: register、srl、srl\_reg、reg\_srl 和 reg\_srl\_reg。将该属性放置在声明为 SRL 的信号上。只能在 RTL 中设置该属性。此外,该属性只能用于静态 SRL。

## (1) SRL\_STYLE Verilog 例子：

```
(* srl_style = "register" *) reg [16:0] my_srl;
```

## (2) SRL\_STYLE VHDL 例子：

```
attribute srl_style : string;
attribute srl_style of my_srl : signal is "reg_srl_reg";
```

**8.1.18 TRANSLATE\_OFF/TRANSLATE\_ON**

该属性告诉 Vivado 综合工具忽略代码块。在 RTL 中的注释中给出该属性。注释用下面的关键字开始: synthesis、synopsys、pragma。只能在 RTL 中设置该属性。

## (1) TRANSLATE\_OFF/TRANSLATE\_ON Verilog 例子：

```
// synthesis translate_off
  code...
// synthesis translate_on
```

## (2) TRANSLATE\_OFF/TRANSLATE\_ON VHDL 例子：

```
-- synthesis translate_off
  code...
-- synthesis translate_on
```

**8.1.19 USE\_DSP48**

该属性告诉 Vivado 综合工具如何处理综合算术指令。默认地,乘法、乘法-加法、乘法-减法、乘法-累加使用 DSP48 块。加法器、减法器和累加器可以使用这些块,但是默认使用分布逻辑实现。该属性强迫使用 DSP48 块。可用的值是 yes 和 no。该属性可以放置在 RTL 内的信号、结构和元件、实体和模块。可以在 RTL 和 XDC 中设置该属性。

## (1) USE\_DSP48 Verilog 例子：

```
(* use_dsp48 = "yes" *) module test(clk, in1, in2, out1);
```

## (2) USE\_DSP48 VHDL 例子：

```
attribute use_dsp48 : string;
```

```
attribute use_dsp48 of P_reg : signal is "no"
```

### 8.1.20 在 XDC 文件中使用属性

使用下面的语法，在 XDC 中指定属性：

```
set_property <attribute> <value> <target>
```

例如：

```
set_property MAX_FANOUT 15 [get_cells in1_int_reg]
```

## 8.2 增量编译

增量编译是一个高级的设计流程，用于探索设计中的时序收敛。当重新综合小的设计后：

- (1) 加速布局和布线的运行时间；
- (2) 通过重新使用以前的布局和布线，保护时序收敛。

### 8.2.1 增量编译流程

当综合后结果的改变导致和参考设计有 95% 的相似之处时，这个流程非常有用。如图 8.1 所示，给出了增量编译的流程。

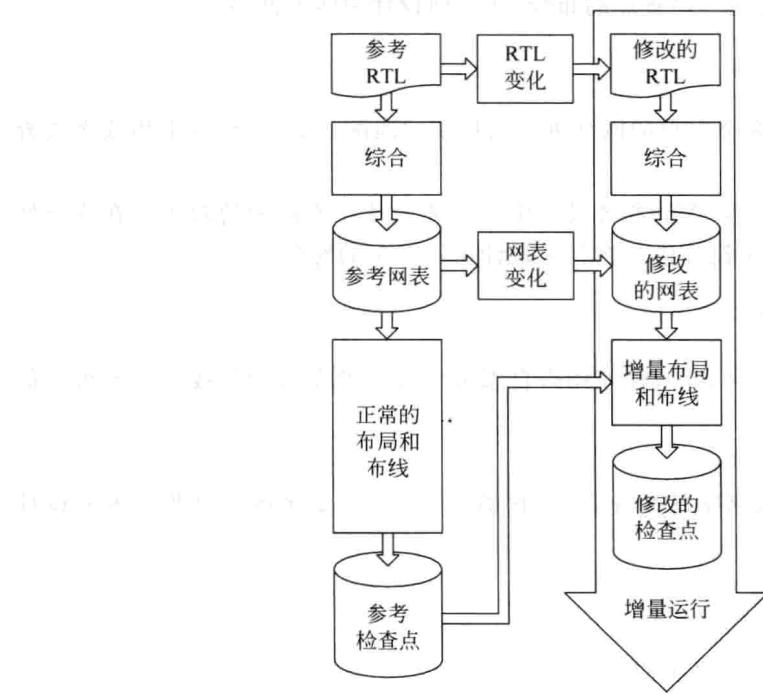


图 8.1 增量设计流程

在增量编译中,涉及两个不同的设计:

(1) 参考设计:第一个增量编译流程的设计是参考设计。参考设计通常是当前设计的早期迭代或者是改变,这个早期的设计已经被综合、布局和布线。

参考设计检查点(Design Checkpoint, DCP)可能是包含用于收敛时序的代码变化、布局规划和修改约束的多次设计迭代的结果。

当加载当前设计后,使用 `read_checkpoint -incremental<dcp>` 命令来加载参考设计检查点。使用`-incremental` 选项,表示随后的布局和布线操作中使用增量编译设计流程。

(2) 当前设计:第二个增量编译流程的设计称为当前设计。当前设计和来自参考设计的微小设计改变/变化合并。这些改变/变化可以包括:

- ① RTL 改变;
- ② 网表变化;
- ③ RTL 改变和网表变化。

## 8.2.2 运行增量布局和布线

首先,将更新后的当前设计加载到存储器中。然后,增量地加载参考设计检查点。增量编译流程中,关键的元件是增量布局和布线。参考设计检查点包括:网表、约束和物理数据(布局和布线)。

- (1) 当前设计和参考设计进行比较,用于识别匹配的单元和网络;
- (2) 重新使用来自参考设计检查点的布局,在当前设计中放置匹配的单元;
- (3) 重新使用来自参考设计检查点的布线,在当前设计中匹配网络。

### 1. 带有多个扇出的网络

Vivado 布线器为带有多个扇出的网络进行细粒度的匹配。如果合适,重用或者放弃每个布线段。

在增量布局完成后,将重新布局参考设计中与当前设计中不匹配的对象。在重新使用布线后,将重新布线在参考设计中与在当前设计中不匹配的对象。

### 2. 有效地使用布局和布线

根据两个设计之间的差异,有效地使用来自参考设计中的布局和布线。甚至很小的差异都会导致很大的影响。

- (1) 微小 RTL 变化的影响。

虽然综合尝试降低网表名字的变化,有时候微小的 RTL 变化将导致非常大的设计变化:

- ① 推断存储器大小的增加;
- ② 内部总线宽度变化;
- ③ 将数据类型从无符号改成有符号。

- (2) 改变约束和综合选项的影响。

类似地,改变下面的约束和综合选项,将对增量布局和布线有相当大的影响:

- ① 改变时序约束和重新综合;
- ② 保留或者平坦化逻辑层次;
- ③ 使能寄存器重定时。

### 3. 检查参考设计和当前设计的相似度。

运行 report\_incremental\_reuse 命令将检查并报告参考设计检查点文件和当前设计的相似度。该命令用于比较在存储器中的当前设计和参考设计检查点,报告匹配的单元、网络和端口的百分比。

相似度越高,越可以有效地重用来自参考设计的布局和布线。如图 8.2 所示,给出了一个报告。在该报告中,比较在存储器内的综合后的设计和布局后设计检查点。

Type(类型)	Count(个数)	Total(总计)	Percentage(百分比)
Reused Cells(重用的单元)	3591	3821	93.98
Reused Ports(重用的端口)	71	71	100.00
Reused Nets(重用的网络)	6142	6564	93.57
Reused Cells(重用的单元)	3591		
Non-Reused Cells(非重用的单元)	230		
New(新的)	230		
Fully reused nets(充分重用的网络)	5383		
Partially reused nets(部分重用的网络)	759		
New nets(新的网络)	422		

图 8.2 比较结果的报告

当单元、网络和端口相似度达到 95% 时,与普通的布局和布线运行时间相比,Vivado 的增量布局和布线大约有三倍的改善。当相似度降低到 80% 以下的时候,使用增量布局和布线将几乎不会带来任何改善。

其他影响改善运行时间的因素包括:

- (1) 在关键时序区域进行改变的个数。
- (2) 当使用增量流程或者标准流程时,phys\_opt\_design 命令执行时序驱动的逻辑转换,它可能产生不同的结果。如果使用 phys\_opt\_design 造成设计的变化,并且集中在关键时序区域,则布线重用不会提供所期望的好处。
- (3) 布局和布线运行时的初始化部分。对于较短的布局和布线运行时间,Vivado 布局和布线器的初始化开销可能抵消来自增量布局和布线过程所带来的利益。对于较长运行时间的设计,初始化只占用很少量的运行时间。

### 8.2.3 使用增量编译

在工程模式和非工程模式下,当使用命令:

```
read_checkpoint - incremental <dcp_file>
```

加载参考设计检查点时,进入增量布局和布线模式。

其中:

- (1) dcp\_file 指定参考设计检查点的路径和名字;
- (2) 使用-incremental 选项,在后面的布局和布线操作使用增量设计流程;
- (3) 在非工程模式下,该命令将用在 opt\_design 后面,但是在 place\_design 的前面。

使用下面的方法,可以退出增量编译模式:

- (1) 不使用-incremental 选项;
- (2) 如果使用工程,去掉增量编译检查点设置。

制定一个设计检查点文件 DCP 作为参考文件,运行增量布局:

```
link_design ;                                     #加载当前设计
opt_design
read_checkpoint - incremental <dcp_file>
place_design
```

注:在运行增量编译流程时,限制使用 opt\_design - resynth\_area。

在工程模式下,读者可以在 Design Run 窗口下,设置增量编译选项。下面给出设置步骤:

- (1) 如图 8.3 所示,在 Vivado 主界面的 Design Runs 窗口下,选中一个实现 impl\_1。单击右键,出现浮动菜单。在浮动菜单内,选择 Set Incremental Compile... 选项。

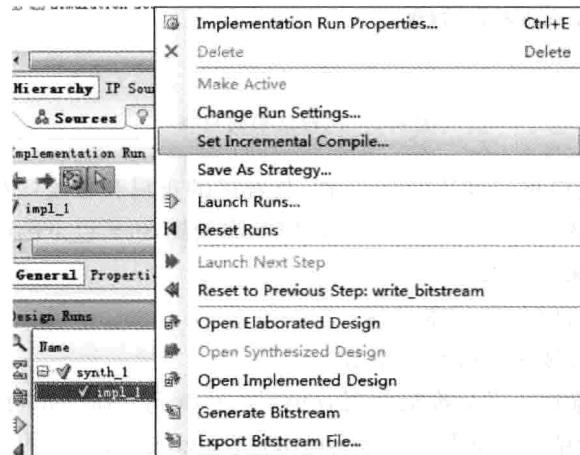


图 8.3 设置增量编译入口

- (2) 如图 8.4 所示,出现 Set Incremental Compile 对话框界面。在该界面中,设置参考设计的检查点。

(3) 单击 OK 按钮。

注:

- ① 如果对设计运行进行了复位,则删除了参考设计检查点文件。如果要使用它,则将它单独保存到一个目录中。
- ② 如果不想使用增量编译,则在如图 8.5 所示的 Project settings 窗口下,将 Incremental compile 后面变为空,即:不使用参考设计检查点。

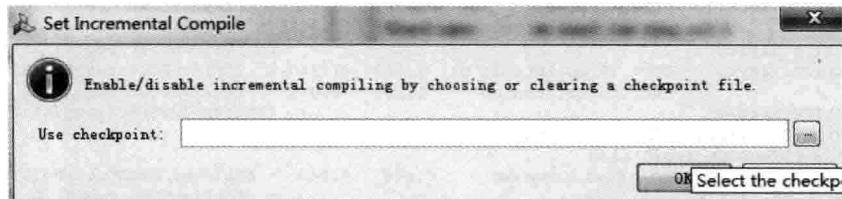


图 8.4 设置检查点入口

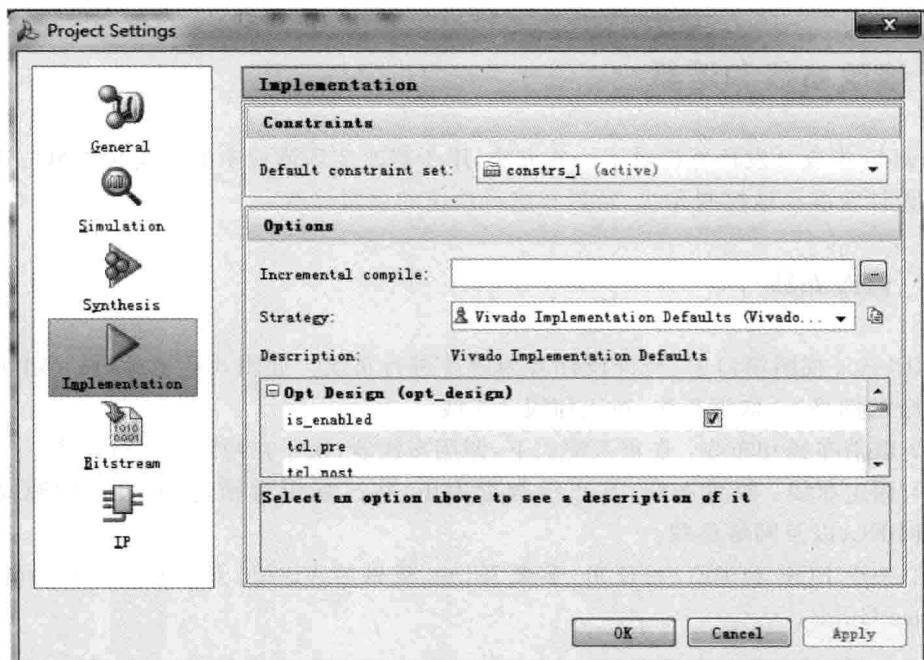


图 8.5 工程设置窗口

### 8.2.4 增量编译高级分析

在完成增量布局和布线后,设计者可以分析带有单元和重用细节的时序。时序报告中的对象,给出了物理数据的重用。这样,识别更新的设计是否影响关键路径。

为了在时序报告中,看到增量流程的细节,可以使用 report\_timing - label\_reused 选项。这个选项将生成一个报告,该报告中将详细地展示出在输入和输出引脚上的重用标号。这些标号用于表示重用于引脚的单元和网络的物理数据的个数。图 8.6 给出了一个一个高级分析的例子,其中:

- (1) (R): 重用的所有单元布局和网络布线;
- (2) (NR): 没有重用的单元布局和网络布线;
- (3) (PNR): 重用单元布局,但是没有重用网络布线;
- (4) (N): 引脚、单元或者网络是一个新的设计对象,没有出现在参考设计中。

(NR) SLICE_X46Y42	FDRE (Prop_fdre_C_Q)	0.259	-1.862	r	fftI/fifoSel_reg[5]/Q
	net (fo=8, estimated)	0.479	-1.383		fftI/n_fifoSel_reg[5]
(R) SLICE_X46Y43	LUT4 (Prop_lut4_I1_O)	0.043	-1.340	r	fftI/wbDOut_reg[31]i5/I1
(R) SLICE_X46Y43	net (fo=32, routed)	1.325	-0.014		fftI/wbDOut_reg[31]i5/O
(R) SLICE_X44Y39				r	fftI/wbci/wbDOut_reg[0]i1/S
(PNR) SLICE_X44Y39	MUXF7 (Prop_muxf7_S_O)	0.154	0.140	r	fftI/wbci/wbDOut_reg[0]i1/O
fftI/wbci/wbDOut_reg[0]i1/O	net (fo=1, routed)	0.000	0.140		fftI/wbci/wbDOut_reg[0]i1/D
(PNR) SLICE_X44Y39				r	fftI/wbDOut_reg[0]D

图 8.6 增量编译高级分析报告

## 8.3 修改布线和逻辑

Vivado 集成开发环境提供了一些方法,用于修改实现流程中的布线和逻辑。这些方法允许设计者准确地控制布线、延迟和进行快速的逻辑修改。

### 8.3.1 修改布线

在 Device 视图窗口中,允许设计者为设计进行布线。如图 8.7 所示,设计者可以在任何单个的网络上,取消布线、布线和锁定布线。

- (1) 取消布线和布线: 在重入模式下,调用布线器,用于在网络上执行操作。
- (2) 固定布线: 放置布线,在布线数据库中,将其标记为锁定,并且锁定驱动器的 LOC 和 BEL,以及网络负载。

注: 关于 LOC 和 BEL 的使用,参考 Xilinx 提供的 Vivado Design Swite Properties Reference Guide。

#### 1. 手工布线

手工布线允许设计者为网络选定特定的布线资源。这样,允许设计者完全控制信号的布线路径。手工布线时,不调用 route\_design。在布线数据库中,直接更新布线。

当设计者需要准确地控制网络延迟时,就需要使用手工布线。需要特别注意的是,手工布线必须清楚地知道器件的内部结构。这种方法,最好用于有限的信号和短距离的连接。

当进行手工布线时,需要遵循下面的规则:

- (1) 驱动器和负载要求 LOC 约束和 BEL 约束。
- (2) 在手工布线时,不允许分支。但是,设计者可以从分支点开始一个新的手工布线。这样,就可以实现分支。
- (3) 必须锁定负载的引脚。
- (4) 设计者必须布线连接到负载,而这些负载并没有连接到一个驱动器。
- (5) 只允许完整的连接,不允许出现天线。
- (6) 允许带有非锁定布线网络的重叠。在手工布线后,运行 route\_design 解决由于重叠网络所引起的冲突。

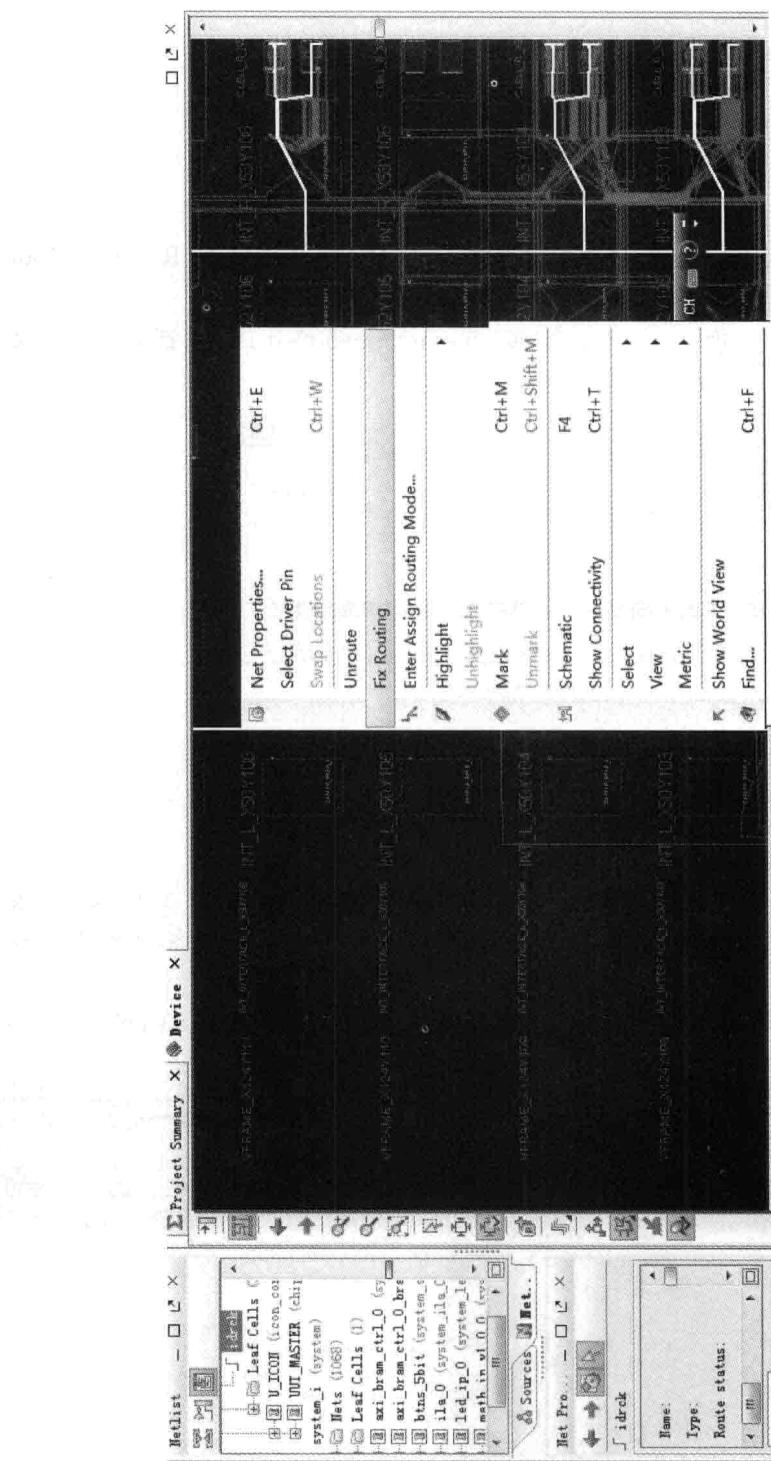


图 8.7 修改布线界面

## 2. 进入分配布线模式

进入分配布线模式的步骤主要包括：

- (1) 进入 Device 设计视图。
- (2) 选择需要布线的网络。
- ① 没有布线的网络由红色的飞线表示。
- ② 部分布线网络由黄色高亮显示。
- (3) 单击右键, 出现浮动菜单。在浮动菜单内, 选择 Enter Assign Routing Mode... 选项。
- (4) 如图 8.8 所示, 出现 Assign Routing: Target Load Cell Pin 对话框界面。读者可以选择想要布线的一个负载单元引脚(可选的)。

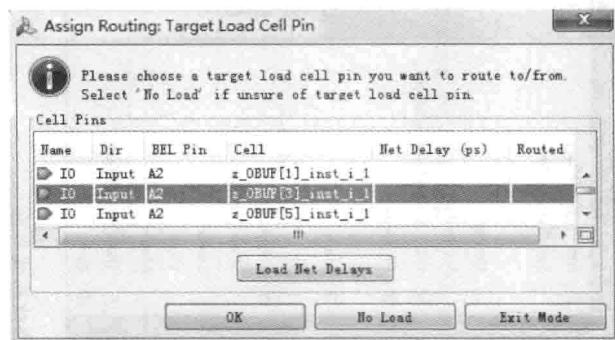


图 8.8 分配布线对话框界面

- (5) 单击 OK 按钮。

**注:** 读者可以打开本书提供的设计资料的 vivado\_example\gate\_verilog\_assign\_route。由于这个设计比较简单, 读者可以删除已经完成的某个布线网络, 进行手工布线练习。

- (6) 如图 8.9 所示, 布线分配网络分为两个部分: Assigned Nodes 和 Neighbor Nodes。

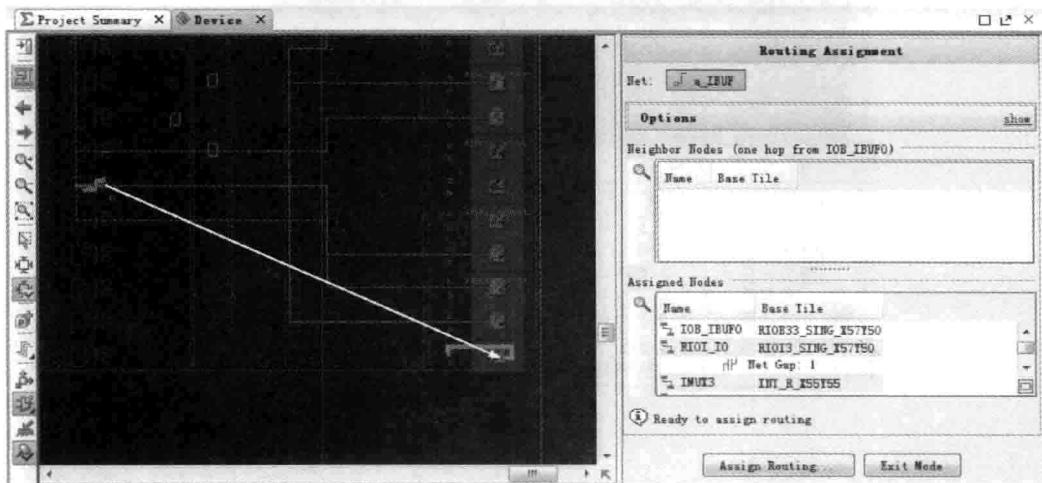


图 8.9 布线分配对话框界面

在 Assigned Nodes 窗口显示了已经分配了布线的节点。每个已经分配的节点作为单独的一行显示。

在 Device 视图中,带有分配布线的节点,以橙色高亮方式显示。在 Assigned Nodes 窗口中,在分配节点之间的任何空白点(gap),都以 GAP 行的形式显示。

要分配下一个布线段,选择空白点之前的节点和之后的节点,或者在 Assigned Nodes 部分的最后一个分配的节点。

在该视图的 Neighbor Nodes 部分,显示了允许的相邻节点。如图 8.10 所示,以白色高亮的形式,显示当前选择的节点;以白色虚线的形式,显示相邻的节点。

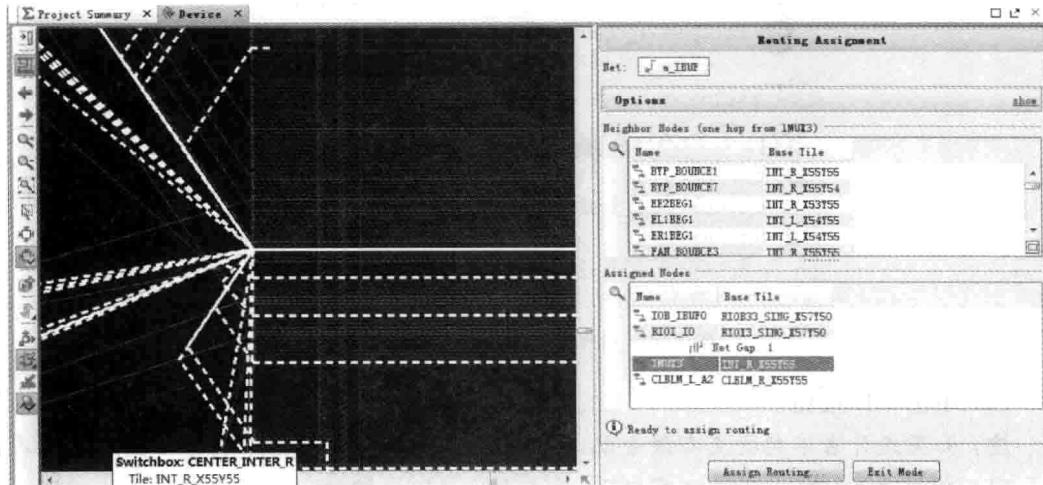


图 8.10 分配下一个布线段

### 3. 分配布线节点

一旦确定哪一个相邻的节点将用于下一个布线段,设计者就可以:

(1) 在 Neighbor Node 部分,选中该相邻节点。单击右键,出现浮动菜单。在浮动菜单内,选择 Assign Node 选项。

(2) 在 Neigbor Nodes 部分,双击节点。

(3) 在 Device 视图中,单击节点。

一旦给相邻节点分配了布线路径,则在 Assigned node 部分显示该节点,并且在 Device 视图中以橙色高亮方式显示。

连续分配节点,直到达到负载为止;或者已经准备使用空白点分配路径为止。

### 4. 取消分配布线节点

取消分配布线节点的步骤主要包括:

(1) 在 Routing Assignment 窗口下,进入 Assigned Nodes 子窗口;

(2) 选择将要取消分配布线的节点;

(3) 单击右键,出现浮动菜单。在浮动菜单内,选择 Remove 选项。

这样,从分配窗口中删除该节点。

## 5. 完成并退出分配布线模式

完成并退出分配布线模式的步骤主要包括：

- (1) 在 Device 视图中,选择一个网络。
- (2) 在 Routing Assignment 窗口下,单击 Assign Routing...按钮。
- (3) 如图 8.11 所示,出现 Assign Routing 对话框界面。该界面用于在提交布线前,对分配的节点进行验证。

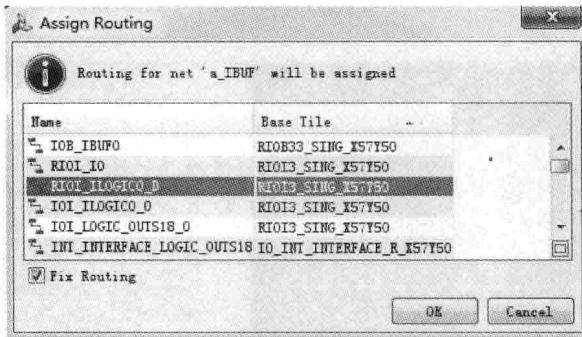


图 8.11 分配布线确认对话框界面

- (4) 单击 OK 按钮。

**注:** 如果设计者不想提交布线分配,可以通过下面的方法取消并退出分配布线模式:

- (1) 在 Routing Assignment 窗口下,单击 Exit Mode 按钮。
- (2) 在 Device 视图中,单击右键,出现浮动菜单。在浮动菜单内,选择 Exit Assign Routing Mode...。

当提交布线后,将固定驱动器和负载 BEL 和 LOC。

在 Device 视图中,以绿色线高亮显示已经分配的布线;以黄色虚线显示部分已经分配的布线。

## 6. 锁定 LUT 负载上的单元输入

设计者必须确保连接到 LUT 负载的布线,不会被其他 LUT 的输入取代。实现这个目标的步骤包括:

- (1) 打开 Device 视图窗口;
- (2) 在该窗口下,选择负载 LUT;
- (3) 单击右键,出现浮动菜单。在浮动菜单内,选择 Lock Cell Input Pins。

## 7. 分支布线

当给一个带有多个负载的网络分配布线时,通过下面的步骤,实现对该网络的布线:

- (1) 按前面的方法,进入手工布线模式;
- (2) 为网络的所有分支分配布线。

图 8.12 给出了网络 a\_buf 已经分配了一部分的布线。

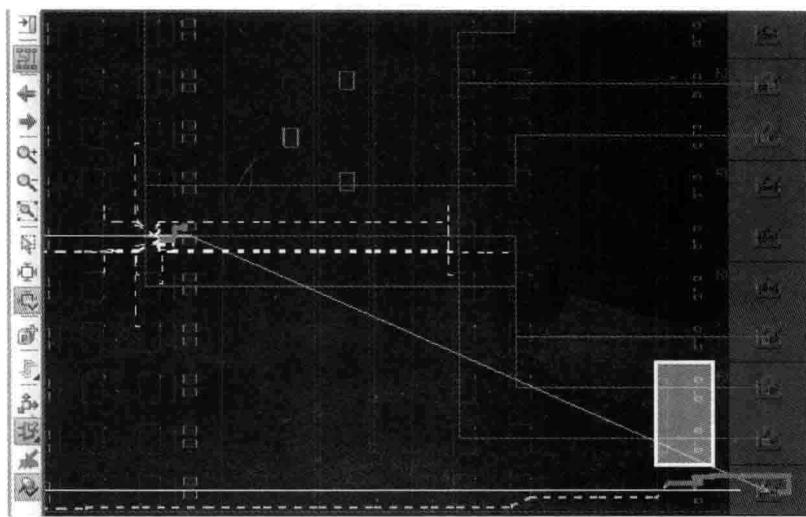


图 8.12 分配分支布线确认对话框界面

为分支分配布线的步骤主要包括：

- (1) 进入 Device 视图界面。
- (2) 选择将要布线的网络，在本例中选择 a\_ibuf。
- (3) 单击右键，出现浮动菜单。在浮动菜单内，选择 Enter Assign Routing Mode... 选项。
- (4) 如图 8.13 所示，出现对话框界面，该对话框界面列出了 a\_ibuf 网络所有的负载。

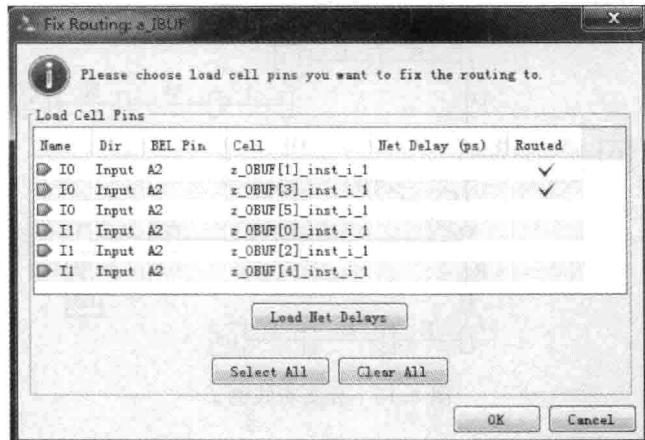


图 8.13 目标负载单元引脚对话框界面

注：如果为负载已经分配了布线，则在 Routed 所对应的列中，使用标记。

- (5) 单击 OK 按钮。
- (6) 如图 8.14 所示，出现 Assign Routing : Branch Start 对话框界面。在该对话框界面中，选择开始布线驱动器一侧的节点。
- (7) 单击 OK 按钮。并按前面的方法分配布线。

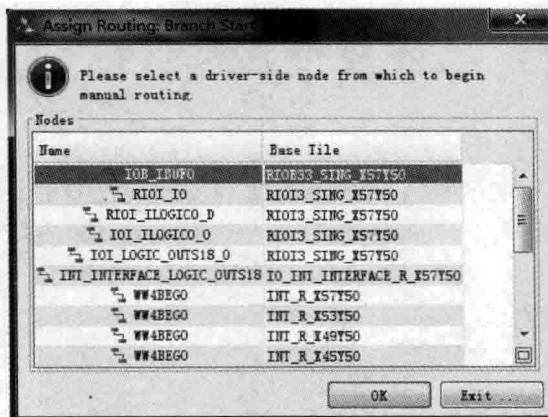


图 8.14 为分支分配布线对话框界面

## 8. 直接约束布线

在布线数据库中,以直接的布线字符串方式,保存分配的固定布线。在一个直接的布线字符串中,由嵌套的{}表示分支。

如图 8.15 所示,给出了一个分支布线的例子。在这个简化的布线图上,给出了不同的元件。其中:

- (1) 方框表示驱动器和负载;
- (2) 矩形框表示开关;
- (3) 连线表示节点。

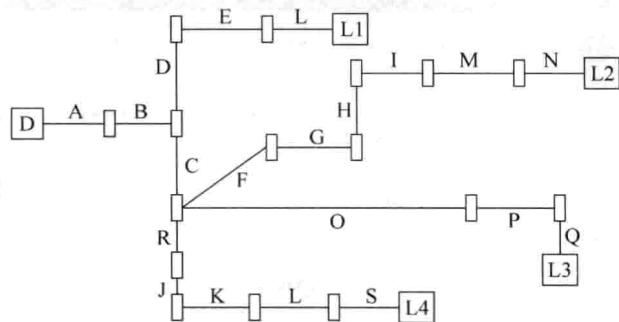


图 8.15 分支布线例子

一个简化直接布线字符串如下:

```
{A B {D E L} C {F G H I M N} {O P Q} R J K L S}
```

### 8.3.2 修改逻辑

在实现后,使用 Vivado 集成开发环境用户接口或者 Tcl 脚本命令,可以修改逻辑对象的属性。修改逻辑对象属性的步骤主要包括:

(1) 如图 8.16 所示,在 Device 视图窗口下,选择一个逻辑对象。在本设计中,选择一个 LUT。

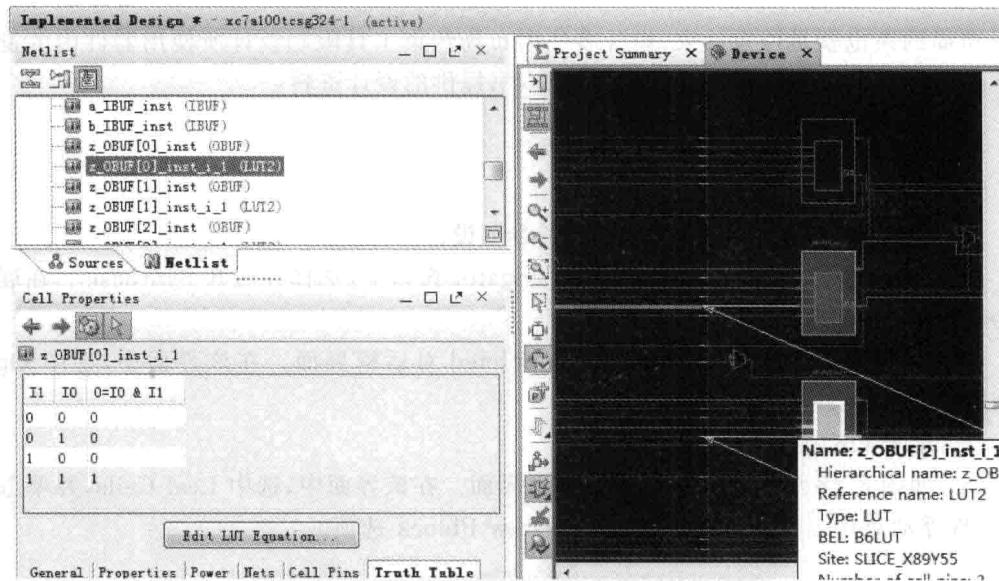


图 8.16 修改逻辑对象属性入口

- (2) 在该界面左侧的 Cell Properties 窗口下,选择 Truth Table 标签。
- (3) 在 Truth Table 标签窗口下,单击 Exit LUT Equation 按钮。
- (4) 如图 8.17 所示,出现 Specify LUT Equation 对话框界面。在该界面的 LUT Equation 右侧输入逻辑表达式。

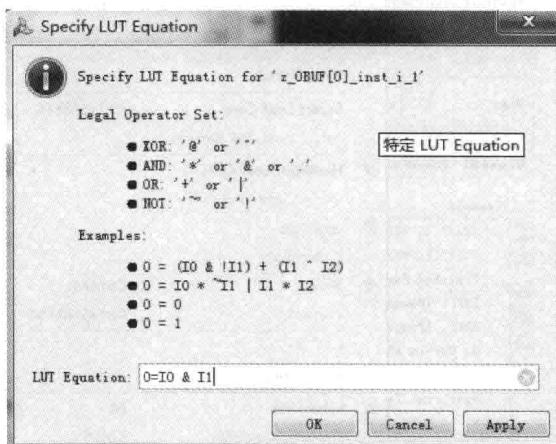


图 8.17 修改 LUT 属性

- (5) 单击 OK 按钮。

**注:** 可修改的属性包括从修改 BRAM 的 INIT 到修改 MMCM 的时钟属性。

- (6) 在 Vivado 主界面主菜单下,选择 File→Save Constraints。保存约束文件。

## 8.4 布局约束

布局约束也就是位置约束,设计者使用布局约束工具限制设计所使用器件内的逻辑资源。为了帮助读者掌握布局约束,使用本书提供的设计资料:

Vivado\_example\gate\_verilog\_assign\_place

布局约束的步骤主要包括:

- (1) 在 Vivado 集成开发环境中,打开参考设计。
- (2) 在 Vivado 主界面左侧的 Flow Navigator 窗口下,选择并展开 Synthesis。在展开项中,单击 Run Synthesis 选项。
- (3) 综合完成后,出现 Synthesis Completed 对话框界面。在该界面中,选中 Open Synthesized Design 前面的复选框。
- (4) 单击 OK 按钮。
- (5) 如图 8.18 所示,出现 Netlist 窗口界面。在该界面中,选中 Leaf Cells,并单击右键,出现浮动菜单。在浮动菜单内,选择 Draw Pblock 选项。

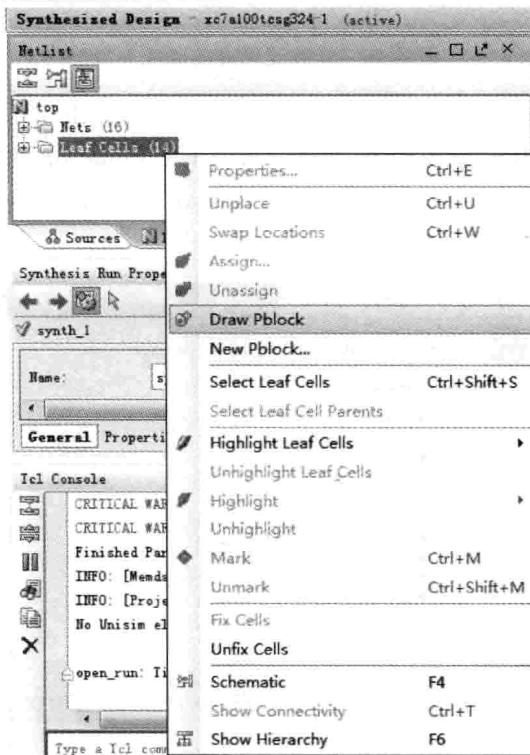


图 8.18 布局约束入口

- (6) 如图 8.19 所示,按照设计者的设计要求,在 Device 中绘制一个矩形框。
- (7) 如图 8.20 所示,出现 New Pblock 对话框界面。在 Name 后输入: pblock\_1。
- (8) 单击 OK 按钮。

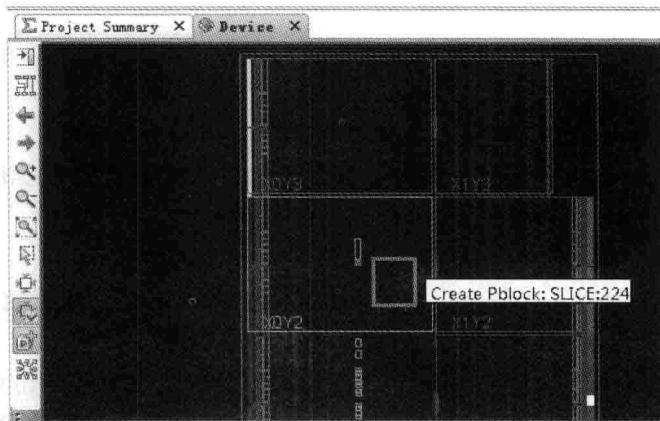


图 8.19 绘制 Pblock 块

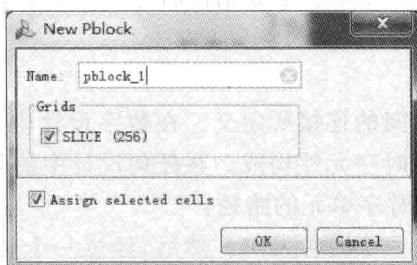


图 8.20 绘制 Pblock 块

- (9) 在 Vivado 主界面主菜单下,选择 File→Save Constraints,保存约束。
- (10) 在 Vivado 主界面源文件窗口下,找到并打开 top.xdc 文件。如图 8.21 所示,看到该设计的位置约束条件。

```

38 create_pblock pblock_1[Leaf_Cells_top]
39 create_pblock pblock_1
40 create_pblock pblock_2
41 add_cells_to_pblock [get_pblocks pblock_2] [get_cells -quiet [list b_IBUF_inst {z_OBUF[3]_inst_i_1} {z_OBUF[2]_inst_i_1} {z_OBUF[5]
42 resize_pblock [get_pblocks pblock_2] -add [SLICE_X32Y111 SLICE_X47Y126]

```

图 8.21 位置约束

## 8.5 查看和分析时序报告

时序报告是进行时序分析的前提。本节介绍时序检查基础、生成时序报告。分析时序报告。

注：生成时序报告，需要完成实现(Implement)过程。

### 8.5.1 时序检查基础

本节将介绍时序约束的一些基本概念,以帮助读者更好地使用时序约束提高设计

性能。

## 1. 基本术语

- (1) 发送沿(launch edge): 是指发送数据的源时钟的活动边沿。
- (2) 捕获边沿(capture edge): 是指捕获数据的目的时钟的活动边沿。
- (3) 源时钟(source clock): 是指发送数据的时钟。
- (4) 目的时钟(destination clock): 是指捕获数据的时钟。
- (5) 建立要求(setup requirement): 是指定义了最苛刻建立约束的发送沿和捕获沿之间的关系。
- (6) 建立关系(setup relationship): 是指由时序分析工具验证的建立检查。
- (7) 保持要求(hold requirement): 是指定义了最苛刻保持约束的发送沿和捕获沿之间的关系。
- (8) 保持关系(hold relationship): 是指由时序分析工具验证的保持检查。

## 2. 时序路径

时序路径由设计实例之间的连接所定义。在数字设计中,时序路径是由相同时钟或者两个不同时钟控制的一对时序元件构成。在任何设计中的公共时序路径是指:

- (1) 从输入端口到内部时序单元的路径:  
数据由板上的器件外的一个时钟发出。然后,经过一个延迟(称之为输入延迟(SDC文件定义))后到达设备端口。在到达一个由目的时钟所控制的一个时序单元之前,穿过器件内部的逻辑传播。
- (2) 从时序单元到时序单元的内部路径:  
由器件内部的时序单元发出数据,该时序单元由源时钟驱动。在到达由目的时钟所控制的一个时序单元之前,穿过一些内部的逻辑传播。
- (3) 从内部时序元件到输出端口的路径:  
由器件内部的时序单元发出数据,该时序单元由源时钟驱动。在到达输出端口前,穿过一些内部的逻辑传播。在额外的延迟(称为输出延迟)后,由PCB板上的时钟捕获。
- (4) 从输入端口到输出端口的路径:  
在从输入到输出端口的路径中,没有锁存数据,数据直接穿过器件。这种类型的路径通常称为输入到输出路径。用于输入和输出延迟的参考时钟可以是虚拟时钟或者一个设计时钟。

如图8.22所示,给出了一个时钟路径,图中对上面的介绍的时序路径概念进行了充分的说明。在该例子中,设计时钟CLK0可用于对DIN和DOUT的延迟进行约束。

如图8.23所示,每个时序路径由下面三部分构成:

### (1) 源时钟路径:

源时钟路径从来自源点(典型的,一个输入端口)的源时钟,一直到发送时序单元的时钟引脚。对于起始于输入端口的时序路径,不存在源时钟路径。

### (2) 数据路径:

数据路径是时序路径的一部分,其位于路径的开始点和路径的结束点。

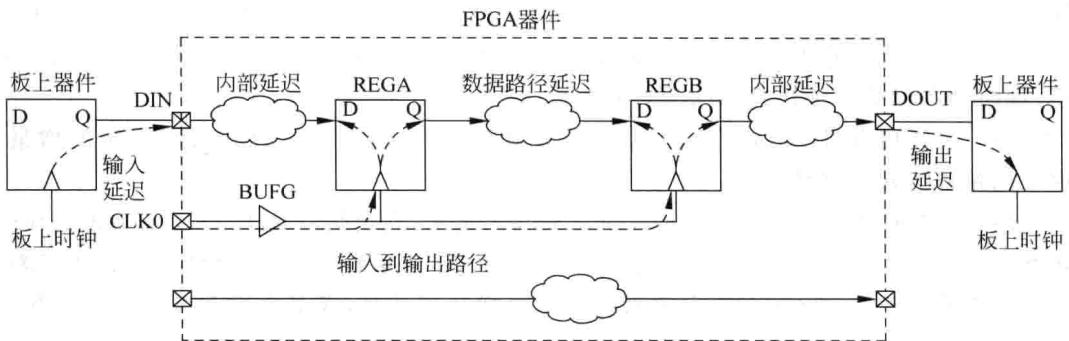


图 8.22 时序路径的例子

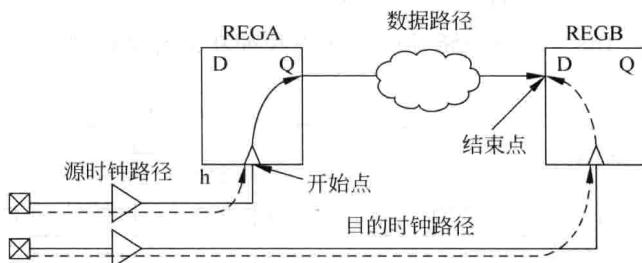


图 8.23 典型的时序路径

- ① 一个路径开始点是一个时序单元的时钟引脚或者是一个数据输入端口；
- ② 一个路径结束点是一个时序单元的数据输入引脚或者一个数据输出端口。
- (3) 目的时钟路径：

目的时钟路径来自源点(典型的,一个输入端口)的目的时钟,一直到正在捕获时序单元的时钟引脚。对于终止于一个输出端口的时序路径,不存在目的时钟路径。

### 3. 发送沿和捕获沿

当在时序单元或者端口之间传输数据时,数据由称之为发送沿的源时钟的一个边沿发出,由一个称之为捕获沿的目的时钟的一个沿捕获。

在一个典型的时序中,数据在一个时钟周期内的两个时序单元中传输。在 0ns 出现发送沿,而在一个周期后出现捕获沿。下面将进一步地定义用于时序分析的建立和保持关系。

### 4. 建立和保持分析

在 Vivado 集成开发环境中,需要分析和报告松弛(slack)。松弛是指在路径端点,数据所要求时间和数据到达时间之间的差。如果在一个路径上,成功地验证了两个寄存器之间的建立和保持关系,则认为数据传输是安全的。换句话说,如果建立和保持松弛的值都是正的,从时序上来说,认为该路径工作正常。

#### 1) 建立检查

只对存在最“悲观”建立关系的两个时钟进行检查。默认地,这对应于发出沿和捕获

沿之间的最小正的增量。为了表示这个关系,以及计算相应的路径要求,时序引擎执行:

(1) 确定源时钟和目的时钟之间的公共周期。公共周期是指这个时间,即:在 0ns 后,源时钟和目的时钟还有相同的对齐相位。

(2) 在公共周期内的任意两个活动的捕获沿和发出沿之间,确定最小的正的增量。这个增量称为建立路径要求。

**注:**如果超过 1000 个周期,都没有找到公共周期,则使用在这 1000 个周期内最坏的建立关系。对于这样的情况,则称这两个时钟是不可扩展的,或者说时钟没有公共周期。设计者查看这些时钟之间的路径来访问它们之间的有效性,或者决定是否把它们当作异步路径。

如图 8.24 所示,给出了建立路径要求的例子。考虑两个寄存器之间的一个路径,这两个寄存器对它们各自的时钟的上升沿敏感。从图 8.24 中可以看出,clk0 的周期为 6ns。clk1 的周期为 4ns。假设该路径的发送沿和捕获沿都是时钟的上升沿。

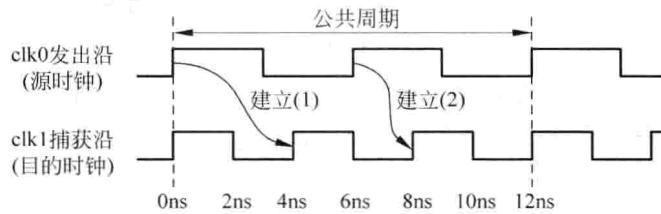


图 8.24 建立路径要求

从 clk0 到 clk1 的最小正的增量是 2ns,对应于建立(2)。

源时钟发送沿时间:  $0\text{ns} + 1 * T(\text{clk0}) = 6\text{ns}$ ;

目的时钟捕获沿时间:  $0\text{ns} + 2 * T(\text{clk1}) = 8\text{ns}$ ;

建立路径要求=捕获沿时间-发送沿时间=2ns;

在计算路径要求时,基于两个重要的考虑:

(1) 时钟沿是理想的,即:没有考虑时钟树插入延迟。

(2) 默认地,在 0 时刻对齐时钟相位。异步时钟没有已知的相位关系。当分析它们之间的路径时,时序引擎使用默认的假设。

用于建立分析的数据要求时间是指,数据必须稳定以前的时间,这个时间确保该数据能被目的单元安全地捕获。这个值基于:

(1) 目的时钟捕获沿时间;

(2) 目的时钟延迟;

(3) 源和目的时钟的不确定性;

(4) 端点的建立时间。

用于建立分析的数据达到时间是指,在路径端点,源时钟发出数据后,数据稳定的时间。这个值基于:

(1) 源时钟发出沿;

(2) 源时钟延迟;

(3) 数据路径延迟。

数据延迟包含从开始点到端点的所有单元和网络延迟。在 Vivado 工具中,将建立

时间作为数据路径的一部分。用下面等式表示：

$$\text{数据要求时间(建立)} = \text{捕获沿时间} + \text{目的时钟路径延迟} - \text{时钟不确定性} - \text{建立时间}$$

数据到达时间(建立)=发送沿时间+源时钟路径延迟+数据路径延迟  
建立松弛是指要求时间和达到时间的差值。

$$\text{松弛(建立)} = \text{数据要求时间} - \text{数据到达时间}$$

当一个寄存器输入数据引脚松弛的值为负时,表示寄存器可能潜在地会锁存一个不期望的值,或者进入到一个亚稳定状态。

## 2) 保持检查

保持检查(也称为保持关系)直接连接到建立关系。当建立分析验证在最“悲观”的情况下仍可以安全捕获数据后,保持关系确认:

- (1) 在建立捕获沿前,建立发送沿不能发送可以被活动沿锁存的数据;
- (2) 在建立发送沿之后的下一个活动的源时钟沿不能发送可以被建立捕获沿锁存的数据。

在保持分析期间,时序引擎只报告最“悲观”保持关系的任何两个时钟。最悲观的保持时间并不是总是和最坏的建立关系有关。时序分析必须检查所有可能的建立关系和它们相应的保持关系,以识别最“悲观”的保持关系。

对于每个建立关系,有两个保持要求用于定义下面的要求:

- (1) 要求(a): 前面的捕获沿减去发送沿;
- (2) 要求(b): 捕获沿减去下一个发送沿。

最终在要求(a)和要求(b)中,将最大值作为保持要求,相对应的时钟沿用在保持分析报告中。

如图 8.25 所示,给出了建立检查沿和它们相关的保持检查。考虑前面建立路径要求的例子。对于建立分析来说,有两个可能的沿组合:

- (1) 建立路径要求( $S1$ )= $1 * T(\text{clk1}) - 0 * T(\text{clk0}) = 4\text{ns}$ ;
- (2) 建立路径要求( $S2$ )= $2 * T(\text{clk1}) - 1 * T(\text{clk0}) = 2\text{ns}$ 。

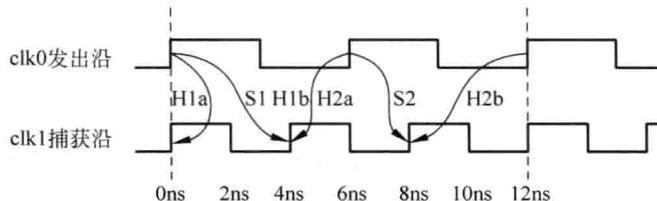


图 8.25 保持路径要求

其相对应的保持要求如下:

(1) 对于建立  $S1$ :

$$\text{保持路径要求}(H1a) = (1 - 1) * T(\text{clk1}) - 0 * T(\text{clk0}) = 0\text{ns};$$

$$\text{保持路径要求}(H1b) = 1 * T(\text{clk1}) - (0 + 1) * T(\text{clk0}) = -2\text{ns};$$

(2) 对于建立  $S2$ :

$$\text{保持路径要求}(H2a) = (2 - 1) * T(\text{clk1}) - 1 * T(\text{clk0}) = -2\text{ns};$$

保持路径要求( $H_{2b}$ ) $=2 * T(clk1) - (1+1) * T(clk0) = -4\text{ns}$ ;

最大的保持要求是 $0\text{ns}$ ,其对应于源时钟和目的时钟的第一个上升沿。在这个例子中,没有从最紧的建立时间中导出最终的保持要求。这是因为,考虑了所有可能的建立沿,用于找到最有挑战性的保持要求。

类似于建立分析一样,基于下面计算数据要求时间和数据到达时间:

- (1) 源时钟发送沿时间;
- (2) 目的时钟捕获沿时间;
- (3) 源和目的时钟延迟;
- (4) 时钟不确定性;
- (5) 数据路径延迟;
- (6) 端点保持时间。

因此,可以得到下面的等式:

$$\begin{aligned} \text{数据要求时间(保持)} &= \text{目的时钟捕获沿时间} + \text{目的时钟路径延迟} \\ &\quad + \text{时钟不确定性} + \text{保持时间}; \end{aligned}$$

$\text{数据到达时间(保持)} = \text{源时钟发送沿时间} + \text{源时钟路径延迟} + \text{数据路径延迟}$   
保持松弛是指要求时间和到达时间的差值:

$$\text{松弛(保持)} = \text{数据到达时间} - \text{数据要求时间}$$

正的保持松弛意味着,在对“悲观”的条件下,数据不能被错误的时钟沿捕获。一个负的保持松弛表示捕获一个不期望的数据,或者一个寄存器进入亚稳定状态。

## 5. 恢复和去除分析

恢复和去除时序检查类似于建立和保持检查,但是它们应用于异步引脚,比如置位和复位。

对于带有异步复位的一个寄存器来说:

(1) 恢复时间是指在异步复位信号切换到不活动状态后,下一个活动时钟沿之前的最长时间。这个最小的时间要求用于安全地锁存一个新的数据。

(2) 去除时间是在一个活动时钟沿后,异步复位信号可以安全切换到不活动状态前的最长时间。

### 1) 恢复检查

下面给出了计算恢复松弛的等式:

$$\begin{aligned} \text{要求时间(恢复)} &= \text{目的时钟沿起始时间} + \text{目的时钟路径延迟} \\ &\quad - \text{时钟不确定性} - \text{恢复时间} \end{aligned}$$

$\text{到达时间(恢复)} = \text{源时钟沿开始时间} + \text{源时钟路径延迟} + \text{数据路径延迟}$   
由此可以得到:

$$\text{松弛(恢复)} = \text{要求时间} - \text{到达时间}$$

### 2) 去除检查

下面给出了计算去除松弛的等式:

$$\begin{aligned} \text{要求时间(去除)} &= \text{目的时钟沿起始时间} + \text{目的时钟路径延迟} \\ &\quad + \text{时钟不确定性} + \text{去除时间} \end{aligned}$$

到达时间(去除)=源时钟沿起始时间+源时钟路径延迟+数据路径延迟  
由此可以得到：

$$\text{松弛(去除)} = \text{到达时间} - \text{要求时间}$$

### 8.5.2 生成时序报告

本节将介绍生成时序报告的流程。生成时序报告的步骤主要包括：

(1) 在 Vivado 主界面左侧的 Flow Navigator 窗口下,找到并展开 Implementation。在展开项中,找到并展开 Implemented Design。在展开项中,选择并单击 Report Timing Summary。

(2) 如图 8.26 所示,出现 Reoprt Timing Summary 对话框界面。在 Options 标签窗口下,可修改的路径包括:

① Path delay type: 设置运行分析的类型。对于综合后设计,默认只执行最大延迟分析(建立/恢复);对于实现后的设计,默认可以执行最小和最大延迟分析(建立/保持,恢复/去除)。如果只运行最小延迟分析(保持/去除),选择延迟类型 min。

② Report unconstrained paths: 针对没有时序要求的路径生成信息。默认地,在 Vivado 集成开发环境选中这个命令,在 Tcl 命令中,关闭该命令。等效的 Tcl 选项:  
-report\_unconstrained。

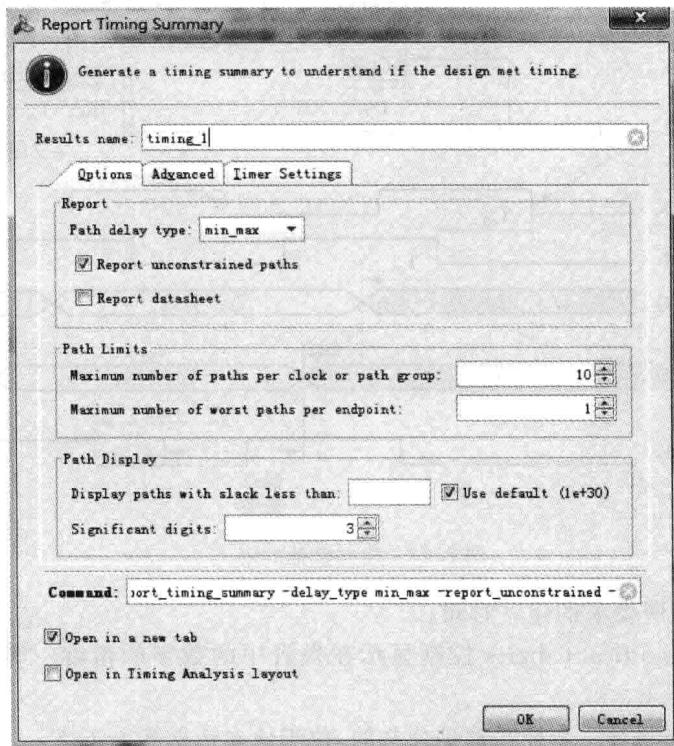


图 8.26 时序报告-Options 标签

③ report datasheet: 生成设计的数据手册。等效的 Tcl 选项: -datasheet。

④ Path Limits 部分:

Maximum number of paths per clock or path group: 控制报告每个时钟对或者路径组路径的最大个数。等效的 Tcl 选项: -max\_paths。

Maximum number of worst paths per endpoint: 控制报告在每个路径端点最坏路径的最大个数。等效的 Tcl 选项: -nworst。

⑤ Path Display 部分:

Display paths with slack less than: 基于它们松弛的值,过滤所要报告的路径。这个选项不影响总结表中的内容。等效的 Tcl 选项: -slack\_lesser\_than。

注: Slack 英文本身的意思是松弛,若建立时间/保持时间 slack 的值为正,表示目前满足建立/保持时间的要求,而且还有多余的时间;若 slack 为负值,表示目前已经不满足建立/保持时间要求,且表示缺少多长时间。

如图 8.27 所示,建立松弛由下式表示:

$$\text{建立松弛} = \text{数据要求时间} - \text{数据到达时间}$$

当该值为正时,表示满足时序要求,否则不满足建立时间要求。如图 8.28 所示,保持松弛由下式表示:

$$\text{保持松弛} = \text{数据到达时间} - \text{数据要求时间}$$

若该值为正,表示满足保持时间,否则不满足保持时间。

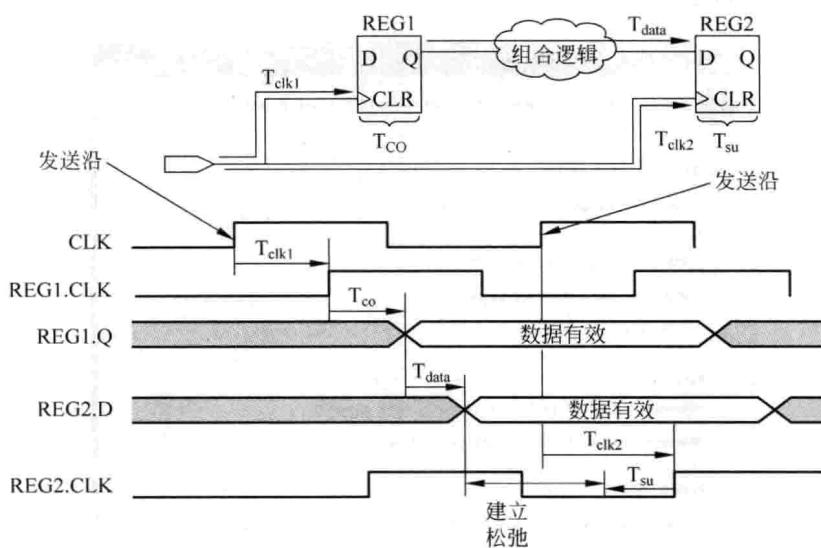


图 8.27 建立松弛的定义

图中: Tsu 为所要求的建立时间。

Number of significant digits 控制显示在报告中的数字的精度。等效的 Tcl 选项: -significant\_digits。

下面的控制选项对三个标签都是公共的,位于该对话框界面下方:

(1) Command: 在 Report Timing Summary 下,显示等效于这种选项的 Tcl 命令;

(2) Open in a New Tab: 在一个新的标签中,打开结果;或者代替最后一个由结果

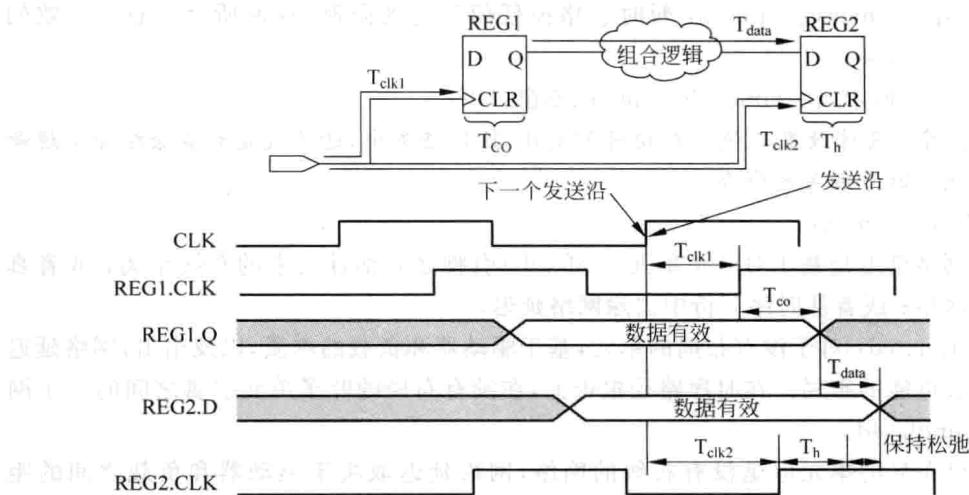


图 8.28 保持松弛的定义

窗口打开的标签；

注： $T_h$  为所要求的保持时间。

(3) Open in Timing Analysis Layout: 将当前视图布局复位到时序分析视图布局。

如图 8.29 所示，在 Advanced 标签窗口下：

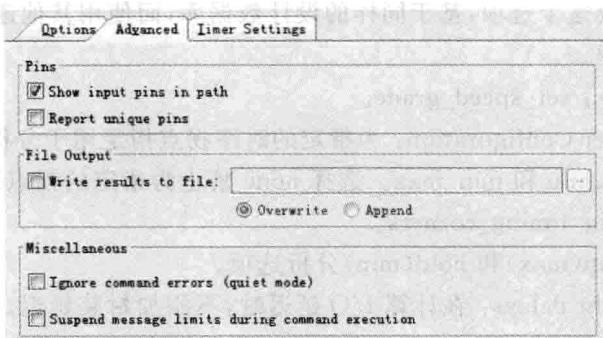


图 8.29 时序报告-Advanced 标签

(1) Show input pins in path: 推荐始终选中该选项前面的复选框。等效的 Tcl 选项：-input\_pins。

(2) File Output:

① Write results to file: 写结果到指定的文件名字。默认地，结果写到 Vivado 集成开发环境的 Timing 窗口。等效的 Tcl 选项：-file。

② Overwrite/Append: 当结果写到文件时，确定是否覆盖指定的文件，或者是否将新的信息添加到一个已经存在的报告中。等效的 Tcl 选项：-append。

(3) Miscellaneous:

① Ignore command errors: 执行命令，忽略任何命令行错误，不返回信息。命令也返回 TCL\_OK，不考虑在执行过程中所遇到的任何错误。等效的 Tcl 选项：-quiet。

② Suspend message limits: 暂时忽略掉任何信息的限制, 返回所有信息。等效的 Tcl 选项: -verbose。

如图 8.30 所示, 在 Timer Settings 标签窗口下:

**注:** 推荐不要修改默认值。在相同 Vivado IDE 任务中, 这些设置影响除综合实现命令外的其他与时序相关的命令。

(1) Interconnect:

控制网络延迟是基于对叶子单元(leaf cell)引脚之间估计出来的布线距离; 或者真正的布线网络; 或者从时序分析中去除网络延迟。

① Estimated: 对于没有布局的单元, 基于驱动器和负载的本质, 以及扇出, 网络延迟值对应于最可能的布局。在时序路径报告上, 在没有布局的叶子单元引脚之间的一个网络标记为 unplaced。

对于已布局的单元但是没有布线的网络, 网络延迟取决于驱动器和负载之间的距离, 以及扇出。在时序路径报告上, 这个网络标记为 estimated。

② Actual: 对于布线的网络, 网络延迟对应于布线互连的真实的硬件延迟。在时序路径报告上, 这个网络标记为 routed。

③ None: 在时序报告中, 不考虑互连延迟。将网络延迟置为 0。

(2) Speed Grade: 设置器件速度等级。默认地, 该选项基于创建工程或者打开一个设计检查点时, 所选择的器件。

设计者可以修改这个选项, 基于同样的设计数据库, 而使用其他速度等级, 并且不需要重新运行实现流程。

等效的 Tcl 命令: set\_speed\_grade。

(3) Multi-Corner Configuration: 为指定的时序拐点指定用于分析的路径延迟类型。有效值为 none、max、min 和 min\_max。选择 none 禁止为指定的拐点进行时序分析。等效的 Tcl 命令: config\_timing\_corners。

推荐: 保持 setup(max) 和 hold(min) 分析选项。

(4) Disable flight delays: 在计算 I/O 延迟时, 不添加封装延迟。等效的 Tcl 命令: config\_timing\_analysis。

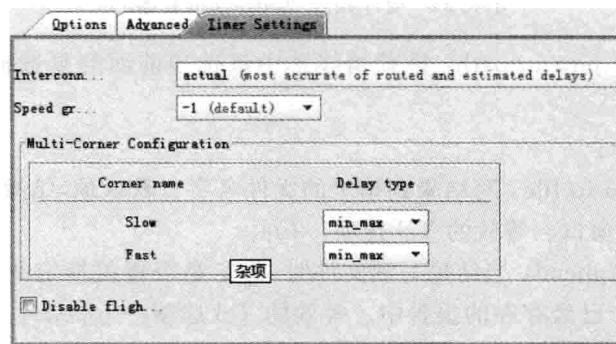


图 8.30 时序报告-Timer Settings 标签

### 8.5.3 分析时序报告

如图 8.31 所示,给出了生成的时序报告,时序报告包含以下几个部分:

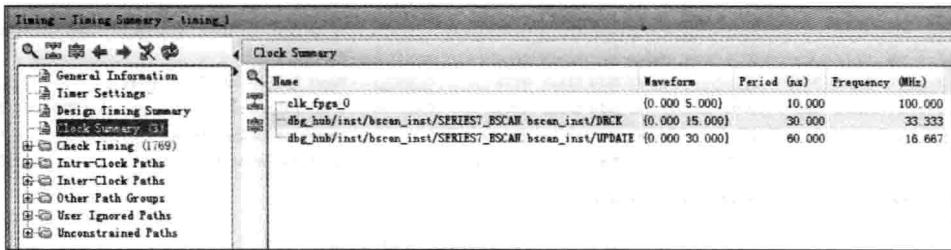


图 8.31 时序报告的构成

#### (1) General Information:

给出了下面的相关信息:

- ① Design: 设计名字;
- ② Part: 选择的器件、封装和速度等级;
- ③ Version: Vivado 集成开发环境的版本;
- ④ Date: 当前日期;
- ⑤ Command: 用于生成该报告的等效 Tcl 命令。

#### (2) Timer Settings:

包含 Vivado 集成开发环境时序分析引擎的设置,这些设置用于在报告中生成时序信息。如图 8.32 所示,给出了 Timer Settings。

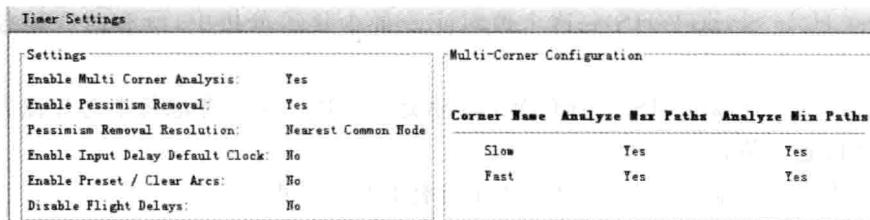


图 8.32 Timet Setting 内容

① Enable Multi Corner Analysis: 使能该选项,用于每个拐点(Multi-Corner 配置);

② Enable Pessimism Removal (and Pessimism Removal Resolution): 确保所报告中,每个路径的源和目的时钟,在它们公共节点上没有抖动。

**注:** 总是使能该设置。

③ Enable Input Delay Default Clock: 在输入端口上,创建一个默认的,不带有用户约束的空输入延迟约束。默认地,禁止该设置。

④ Enable Preset/Clear Arcs: 使能通过异步引脚的时序路径传播。它不影响恢复/去除前面的复选框设置。默认地,禁止该设置。

⑤ Disable Flight Delays: 禁止封装延迟用于计算 I/O 延迟。

## (3) Design Timing Summary:

如图 8.33 所示,该部分给出了设计的时序总结。

**注:** 查看设计时序总结,验证在布局后,满足所有的时序约束,理解设计流程中任意一点的设计状态。

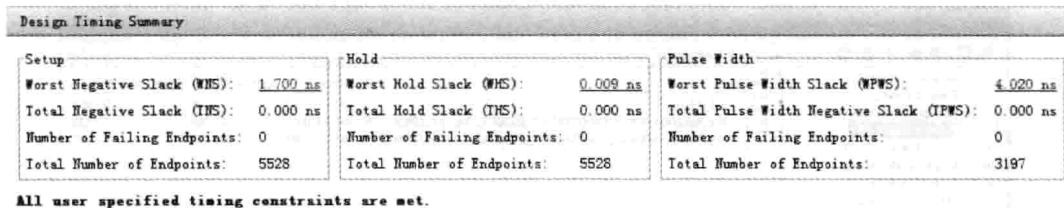


图 8.33 设计时序总结界面

设计时序由下面三部分构成:

## ① Setup(建立,最大延迟分析):

Worst Negative Slack(WNS): 这个值对应于最大延迟分析中,所有时序路径最坏的松弛。

Total Negative Slack(TNS): 所有 WNS 冲突的个数。当只考虑每个时序路径端点最坏的冲突时,它的值是:

0——表示对于最大延迟分析中,满足所有的时序约束;

负值——当出现一些冲突时。

Number of Failing Endpoints: 带有冲突的总的端点的个数(WNS<0ns)。

Total Number of Endpoints: 所分析端点的总的个数。

## ② Hold(保持,最小延迟分析):

Worst Hold Slack(WHS): 这个值对应于最小延迟分析中,所有时序路径最坏的松弛。

Total Hold Slack(THS): 所有 WHS 冲突的个数。当只考虑每个时序路径端点最坏的冲突时,它的值是:

0——表示对于最小延迟分析中,满足所有的时序约束;

负值——当出现一些冲突时。

Number of Failing Endpoints: 带有冲突的总的端点的个数(WHS<0ns)。

Total Number of Endpoints: 所分析端点的总的个数。

## ③ Pulse Width(脉冲宽度,引脚开关限制):

Worst Pulse Width Slack(WPWS): 这个值对应于最大和最小延迟分析中,所列出的所有时序检查中最坏的松弛。

Total Pulse Width Negative Slack(TPWS): 所有 WPWS 冲突的个数。当只考虑设计中每个引脚最坏的冲突时,它的值是:

0——满足所有相关的时序约束;

负值——当出现一些冲突时。

Number of Failing Endpoints: 带有冲突的总的端点的数量(WPWS<0ns)。

Total Number of Endpoints: 所分析端点的总的数量。

#### (4) Clock Summary

如图 8.34 所示,给出了时钟总结界面,其类似于 report\_clock 生成的总结。图中给出:

- ① 设计中所有的时钟(不管是由 create\_clock、create\_generated\_clock 创建,还是由工具自动创建)。

- ② 每个时钟的属性,例如名字、周期、波形、类型和目标频率。

Name	Waveform	Period (ns)	Frequency (MHz)
clk_fpga_0	{0.000 5.000}	10.000	100.000
dbg_hub/inst/bscan_inst/SER...	{0.000 15.000}	30.000	33.333
dbg_hub/inst/bscan_inst/SER...	{0.000 30.000}	60.000	16.667

图 8.34 时钟总结界面

注:名字的缩进反映了主时钟和生成时钟的关系。

#### (5) Check Timing:

如图 8.35 所示,给出了检查时序界面。其中:包含丢失时序约束,或者带有约束问题的路径。对于完整的时序签证(timing signoff)来说,必须约束所有的路径端点。

Timing Check	Count
unconstrained_internal_endpoints	1,195
no_clock	553
no_input_delay	13
no_output_delay	8
multiple_clock	0
generated_clocks	0
loops	0
partial_input_delay	0
partial_output_delay	0
unexpansible_clocks	0
latch_loops	0

图 8.35 检查时序界面

注:时序签证是两个标准的组合:

- ① 充分地约束设计;
- ② 设计满足时序。

注:在 Vivado 主界面主菜单下,选择 Tools→Timing→CheckTiming,或者运行 Tcl 命令: Check\_timing,均可生成单独的检查时序报告。

- ① no\_input\_delay: 少于一个输入延迟约束的输入端口的个数。
- ② no\_output\_delay: 少于一个输出延迟约束的输出端口的个数。
- ③ unconstrained\_internal\_endpoints: 排除不带有时序要求的输出端口后,路径端点的个数。这个数量直接和丢失时钟定义相关联,也在 no\_clock 检查中报告。
- ④ no\_clock: 一个定义时序的时钟没有到达时钟引脚的个数,也报告常量时钟引脚。
- ⑤ multi\_clock: 多个时序时钟所到达的时钟引脚个数。这通常发生在一个时钟树存在一个时钟多路复用器。
- ⑥ generated\_clocks: 参考主时钟源,所生成时钟的个数,它不是相同时钟树的一部分。
- ⑦ loops: 在设计中,发现的组合环路的个数。Vivado 集成开发环境时序引擎,自动

断开环路以报告时序。

⑧ partial\_input\_delay: 只有一个最小输入延迟或者最大输入延迟约束的输入端口个数。在建立和保持分析中, 均不报告这些端口。

⑨ partial\_output\_delay: 只有一个最小输出延迟或者最大输出延迟约束的输出端口个数。在建立和保持分析中, 均不报告这些端口。

⑩ unexpandable\_clocks: 对于时钟对, Vivado 集成开发环境时序引擎在超过 1000 个时钟周期后, 不能找到一个公共周期的多路复用器。在这些时钟对之间的路径不能被安全地合拍, 将它们当作异步时钟。

⑪ latch\_loops: 检查和警告设计中的时序反馈环路。

#### (6) Intra-Clock Paths(内部时钟路径):

如图 8.36 所示, 给出带有相同时钟源和目的时钟的时序路径中, 最坏的松弛和冲突的总数。

Intra-Clock Paths														
Clock	Edges	WNS	TNS	Failing Endpoints	Total Endpoints	Edges	WNS	TNS	Failing Endpoints	Total Endpoints	WNS	TNS	Failing Endpoints	Total Endpoints
	(WNS)	(ns)	(TNS)	(IPWS)	(TNS)		(WNS)	(ns)	(TNS)	(TNS)	(ns)	(TNS)	(IPWS)	(TNS)
clk_ipgen_0	rise - rise	1.700	0.000	0	5129	rise -	0.000	0.000	0	5129	4.920	0.000	0	3008
dbg_hub/inst..._0	rise - rise	22.833	0.000	0	332	rise -	0.132	0.000	0	332	14.900	0.000	0	188
dbg_hub/inst..._1	rise - rise	56.921	0.000	0	1	rise -	0.280	0.000	0	1	29.500	0.000	0	1

图 8.36 内部时钟路径界面

注: 在左侧索引窗口下, 在 Intra-Clock Paths 下单击名字, 就可以看到细节。

#### (7) Inter-Clock Paths(互时钟路径):

如图 8.37 所示, 给出带有不同时钟源和目的时钟的时序路径的最坏松弛和冲突的总数。

Inter-Clock Paths													
From Clock	To Clock	Edges	WNS	TNS	Failing Endpoints	Total Endpoints	Edges	WNS	TNS	Failing Endpoints	Total Endpoints		
		(WNS)	(ns)	(TNS)	(IPWS)	(TNS)		(WNS)	(ns)	(TNS)	(IPWS)	(TNS)	
dbg_hub/inst/bsc..._0	dbg_hub/in...	rise - rise	60.397	0.000	0	18	rise - rise	26.890	0.000	0	18		

图 8.37 互时钟路径界面

#### (8) Other Path Group:

定义了其他路径组的信息。

#### (9) User-Ignored Paths Section:

报告在时序分析过程中, 由于 set\_clock\_group 和 set\_false\_path 约束而忽略的路径。

#### (10) Unconstrained Path

报告由于缺少时序约束, 没有合拍的逻辑路径。这些路径按照源和目的时钟对进行分组。如图 8.38 所示, 当没有时钟和路径起始点和端点关联时, 显示为空(或者 NONE)。

此外, 读者可以在 Vivado 主界面左侧的 Flow Navigator 窗口下, 找到并展开 Implementation。在展开项中, 找到并展开 Implemented Design。在展开项中, 选择并单击 Report Clock Network 选项, 报告设计中的时钟网络。如图 8.39 所示, 给出了一个设计的时钟网络。

Unconstrained Paths		
Path Group	From Clock	To Clock
(none)		
(none)	clk_fpga_0	
(none)		clk_fpga_0
(none)		dbg_hub/inst/bscan_inst/SERIES7_BSCAN.bscan_inst/DRCK
(none)		dbg_hub/inst/bscan_inst/SERIES7_BSCAN.bscan_inst/UPDATE

图 8.38 无约束路径界面

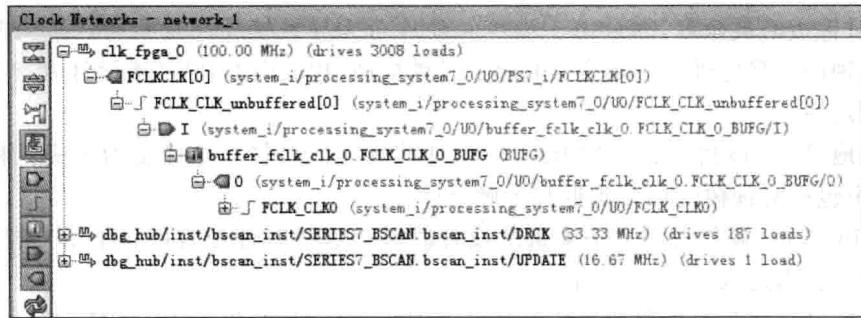


图 8.39 时钟网络界面

## 8.6 时序约束

本节将介绍创建时序约束的方法。包括时钟定义、时钟组、I/O 延迟约束、时序例外和时序约束界面。

### 8.6.1 时钟定义

准确地定义时钟,以最好的准确性来确定覆盖的最大时序路径。对于一个时钟来说,时钟定义包含下面的特性:

- (1) 定义一个驱动器的引脚或者它树根的端口,称为源点。
- (2) 通过周期和波形的属性,描述时钟的边沿。
- (3) 以纳秒(ns)指定时钟的周期。它对应于重复波形所需要的时间。
- (4) 波形是时钟周期内,以 ns 描述的上升沿和下降沿绝对时间列表。列表必须包含偶数个值。第一个值对应于第一个上升沿。除非有其他指定,默认的占空比为 50%,相移为 0ns。

如图 8.40 所示,clk0 时钟的周期为 10ns,50% 的占空比,以及 0ns 的相位。clk1 时钟的周期为 8ns,75% 的占空比,以及 2ns 的相移。

使用下面的方法描述时钟:

```
clk0: period = 10, waveform = {0 5}
clk2: period = 8, waveform = {2 8}
```

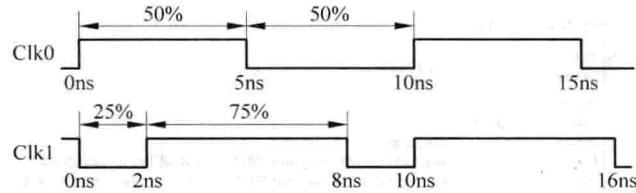


图 8.40 时钟波形

周期和波形属性表示了一个时钟的理想特性。当进入 FPGA 器件或者通过时钟树传播时,时钟边沿被延迟,并且由于噪声和硬件行为导致变化。这些特性称为时钟网络延迟和时钟的不确定性。不确定性包括:时钟抖动、相位误差,以及由设计者指定的任何额外不确定性。

默认地,Vivado 将时钟作为传播时钟,即非理想的时钟。这样是为了提供精确的松弛值,其中包括时钟树的插入延迟和不确定性。

在 FPGA 内,硬件资源支持大量的设计时钟。这些时钟经常由板上的元件提供,然后通过输入引脚进入 FPGA 器件。

可以通过称之为时钟修改模块的特殊原语来生成时钟,例如 MMCM、PLL、BUFR。

### 1. 基本时钟

一个基本时钟是指,通过下面方式进入到设计中的 PCB 板上时钟:

- (1) 一个输入端口;
- (2) 一个吉比特收发器输出引脚(例如一个恢复时钟)。

通过 create\_clock 命令定义一个基本时钟。在网表对象中,必须添加一个基本的时钟,通过这个时钟将派生出设计所用的时钟树。换句话说,当计算时钟延迟和不确定性时,基本时钟的源点为 Vivado 定义了时序的零点。

如图 8.41 所示,给出了一个基本时钟的例子。PCB 板上的时钟通过端口 sysclk 输入到器件中,通过输入缓冲区和一个时钟缓冲区,最后到达路径上的寄存器。该时钟周期为 10ns,占空比为 50%,没有相位移动。

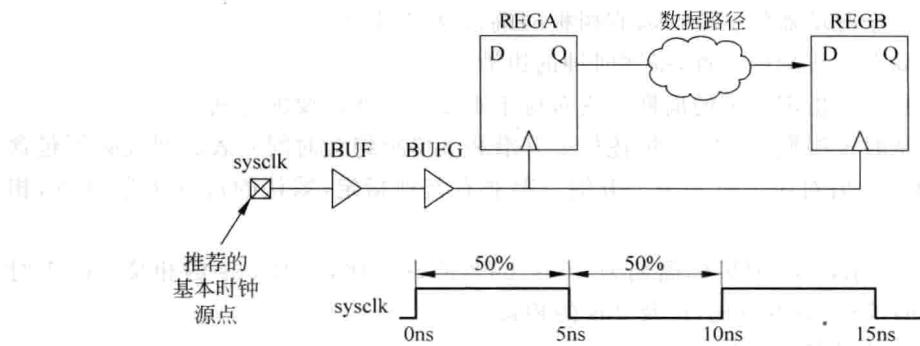


图 8.41 基本时钟例子

在 Xilinx 设计约束(Xilinx Design Constraints,XDC)中,描述为:

```
create_clock - period 10 [get_ports sysclk]
```

类似地,PCB 板上一个名字为 devclk 的时钟通过端口 ClkIn 输入到 FPGA 内。该时钟周期为 10ns, 占空比为 25%, 相移 90 度。在 XDC 中描述为:

```
create_clock - name devclk - period 10 - waveform{2.5 5} [get_ports Clkin]
```

如图 8.42 所示,给出了 GT 基本时钟的例子。一个收发器 gt0,从 PCB 板的一个高速链路中恢复时钟 rxclk。其周期为 3.33ns, 占空比为 50%,连接到 MMCM。该模块生成一些补偿后的时钟,用于设计。

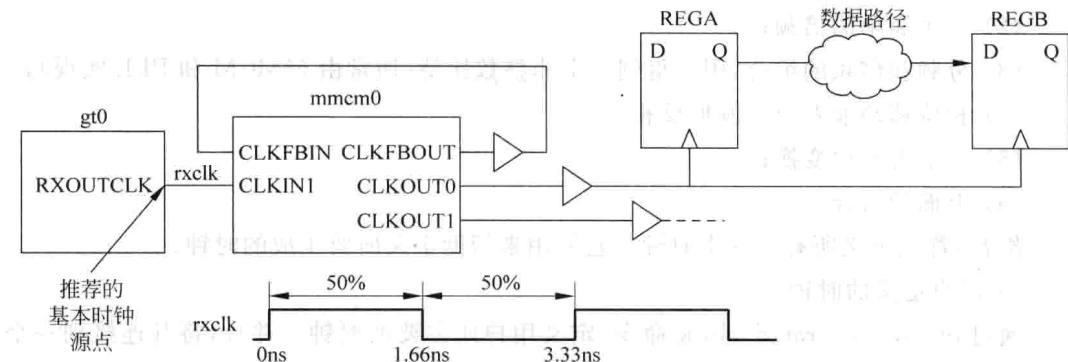


图 8.42 GT 基本时钟例子

当在 GT0 的输出驱动器引脚上定义了 rxclk, MMCM 产生的所有时钟都有一个公共的源点,该源点为 gt0/RXOUTCLK。在它们之间计算松弛,使用了正确的时钟延迟和不确定值。在 XDC 中,表示为:

```
create_clock - name rxclk - period 3.33 [get_pins gt0/RXOUTCLK]
```

## 2. 虚拟时钟

虚拟时钟是指没有物理连接到设计中任何网表元件的时钟。通过 create\_clock 定义虚拟时钟,并没有指明一个源对象。

虚拟时钟经常用在下面的条件中,指明输入和输出延迟约束:

- (1) 外部设备 I/O 参考时钟不是设计中的一个时钟。
- (2) FPGA I/O 路径和一个内部生成时钟相关。但是,不能正确地对该生成时钟(该时钟由来自板上时钟得到)进行定时。这种情况发生在两个周期的比值不是整数的情况下。这样,就导致一个非常紧张和不可靠的时序路径要求。
- (3) 设计者只想为和 I/O 延迟约束相关的时钟指明不同的抖动和延迟,但是不修改内部时钟特性。例如,在 XDC 中可以声明一个虚拟时钟,表示为:

```
create_clock - name clk_virt - period 10
```

注: 在输入和输出延迟约束使用虚拟时钟之前,必须首先定义该时钟。

### 3. 生成时钟

生成时钟由设计内特殊的称之为时钟修改模块(例如 MMCM)的单元或者一些用户逻辑生成。生成的时钟和一个基本时钟或者其他生成的时钟相关。可以从它们的主时钟,直接得到生成时钟的属性。取代指定生成时钟的周期和波形,设计者描述正在修改的电路对主时钟进行变换的方法。

主时钟和变换时钟之间的关系可能是:

- (1) 一个简单的分频;
- (2) 一个简单的倍频;
- (3) 分频和倍频的组合,用于得到一个非整数比值(通常由 MMCM 和 PLL 实现);
- (4) 相位移动或者一个波形反相;
- (5) 一个占空比变换;
- (6) 上面的组合。

推荐,首先定义所有的基本时钟。它们用来帮助定义所要生成的时钟。

#### 1) 用户定义的时钟

通过 `create_generated_clock` 命令,定义用户所需要的时钟。并且,将其连接到一个网表对象,最好是时钟树根引脚。

使用`-source` 选项指定主时钟。该选项表示通过设计中的一个引脚或者端口传播主时钟。

**注:** 该选项只接受一个引脚或者端口网表对象。它不接受时钟对象。

如图 8.43 所示,给出了一个生成时钟的例子。在该例子中,基本时钟的周期为 10ns。通过使用 REGA 进行二分频。该分频时钟驱动另一个寄存器的时钟引脚。相应地,将生成的时钟称为 `clkdiv2`。

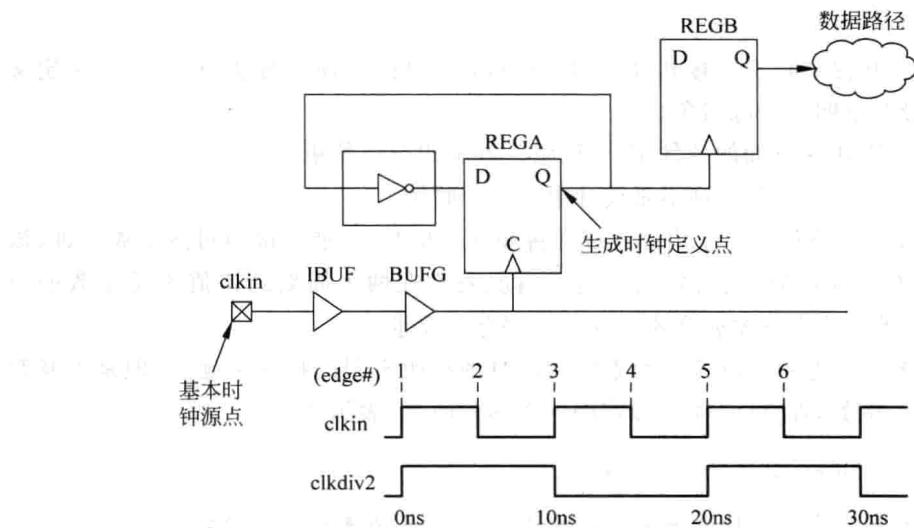


图 8.43 生成时钟例子 1

其等效的约束表示为：

(1) 第一种选择，主时钟源是基本的时钟源点；

```
create_generated_clock - name clkdiv2 - source [get_ports clkin] - divide_by 2 \
[get_pins REGA/Q]
```

(2) 第二种选择，主时钟源是 REGA 时钟引脚。

```
create_generated_clock - name clkdiv2 - source [get_pins REGA/C] - divide_by 2 \
[get_pins REGA/Q]
```

此外，不使用选项，而使用选项，基于主时钟的沿直接描述生成时钟的波形。主时钟边沿的索引是一个列表，用于定义所生成时钟沿的位置，起始于上升沿时钟。下面给出描述的方法。

```
create_generated_clock - name clkdiv2 - source [get_pins REGA/C] - edges {1 3 5} \
[get_pins REGA/Q]
```

通过使用选项，可以对生成时钟波形的每个沿单独地右移或者左移。当需要相移时，才使用这个选项。

当有下面选项时，不能使用选项。即：-divide\_by、-multiply\_by 和-invert。

如图 8.44 所示，给出了生成时钟的例子。该例子中，clkin 是主时钟，周期为 10ns，占空比为 50%。它到达 mmcm0，通过它产生占空比为 25%、相移 90 度的时钟。该生成时钟参照主时钟沿 1、2 和 3 定义。这些边沿发生在 0ns、5ns 和 10ns。为了得到所希望波形，将第一个边沿和第三个边沿移动 2.5ns。

```
create_clock - name clkin - period 10 [get_ports clkin]
create_generated_clock - name clkshift - source [get_pins mmcm0/CLKIN] - edges {1 2 3} \
- edge_shift {2.5 0 2.5} [get_pins mmcm0/CLKOUT]
```

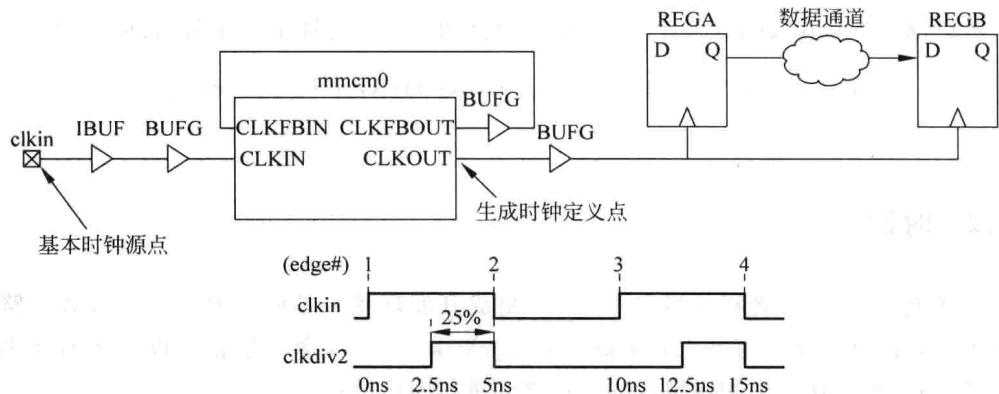


图 8.44 生成时钟例子 2

下面给出同时使用和的例子。这是对标准 Synopsys 设计约束(Synopsys Design Constraints, SDC)的扩展。尽管 Xilinx 推荐让引擎自动地创建这些

约束,但是这对手工定义而由 MMCM 或者 PLL 例化生成的时钟非常方便。

考虑上面的例子,假设对输入主时钟乘 4/3。推荐生成的时钟定义为:

```
create_generated_clock -name clk43 -source [get_pins mmcm0/CLKIN] -multiply_by 4 \
    -divide_by 3 [get_pins mmcm0/CLKOUT]
```

## 2) 自动得到的时钟

自动得到的时钟也称为自动生成的时钟。由 Vivado 集成开发环境在时钟修改模块 (Clock Modifying Blocks, CMB) 的输出引脚自动创建这些约束。CMB 是 MMCM、PLL 或者 BUFR 原语,包含 MIG IP 内的 PHASER\_x。

如果在相同的网表对象上定义了用户(基本或者生成)时钟,则不会创建自动生成的时钟。自动生成时钟的名字取决于与定义点直接相连的网络名字。

图 8.45 给出了自动得到时钟的例子,该例子说明了由 MMCM 生成时钟的方法。在该例子中,主时钟 clkin 驱动 MMCM2 例化(clkip/mmcmm0)的输入 CLKIN。自动生成的时钟名字是 cpuclk,它的定义点是 clkip/mmcmm0/CLKOUT。

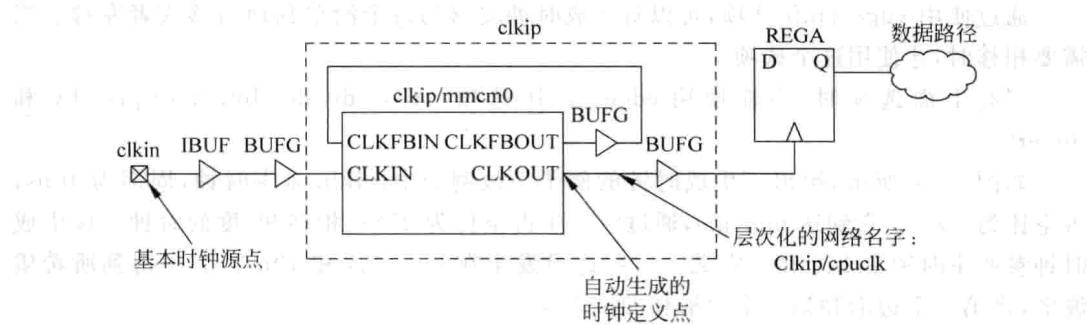


图 8.45 自动得到时钟的例子

可以使用 `get_clocks -of_objects <pin/port/net>` 命令来查询自动生成的时钟,而不需要知道它的名字。

可以通过 `create_generated_clock` 命令为自动得到的时钟重新命名,其格式为:

```
create_generated_clock -name new_name [-source master_pin] [-master_clock master_clk]
source_object
```

## 8.6.2 时钟组

默认地,除非设计者额外指定,Vivado 集成开发环境总是使用时钟组或者假的路径约束来计算设计中所有时钟之间的路径,`set_clock_groups` 命令禁止在设计者所标识的时钟组之间,以及在一个相同组内的时钟之间进行时序分析。

使用原理图查看器或者时钟网络报告,查看时钟树的拓扑结构,以确定哪些时钟不能被放在一起使用。设计者也可以使用时钟交互报告,查看两个时钟之间已经存在的约束,确定它们是否共享相同的基本时钟,即:时钟之间有一个已知的相位关系;或者标识没有公共周期的时钟(不可扩展的)。

**注：**忽略两个时钟之间的时序分析，不意味它们之间的路径可以在硬件中正确地工作。为了防止出现“灾难”，设计者必须验证那些路径，在这些路径上具有正确重同步电路或者异步数据传输协议。

### 1. 时钟范围

#### 1) 同步时钟

当可以预测两个时钟的相对相位时，称这两个时钟同步。当它们的树来自网表相同的根或者它们有一个公共周期时，就属于这种情况。

#### 2) 异步时钟

当不可能确定它们的相对相位时，称这个时钟异步。例如两个时钟由两个独立的振荡器生成，然后它们通过不同输入端口进入 FPGA 器件中。因此，它们就没有已知的相位关系。所以，将它们认为是异步的。

在大多数情况下，将基本时钟看作是异步的。当和它们各自的生成时钟存在关系时，它们形成异步时钟组。

#### 3) 不可扩展时钟

当时序引擎在 1000 个周期后也不能确定它们的公共周期时，两个时钟就是不可扩展的。在这种情况下，在时序分析中，使用这 1000 个周期内最坏的建立关系。

典型地，两个带有循环小数周期比值的时钟就属于这种情况。考虑由共享相同基本时钟的两个 MMCM，生成的时钟 clk0 和 clk1。clk0 有一个 5.125ns 周期，clk1 有一个 6.666ns 周期。

在 1000 个周期内，它们的上升沿没有对齐。时序引擎在两个时钟之间的时序路径上使用一个 0.01ns 建立路径要求。即使这两个时钟在它们的时钟树根上，有一个已知的相位关系，然而它们的波形不允许在它们之间进行安全的时序分析。

**注：**对于不可扩展时钟，经常把它们当作异步时钟。

通过使用 set\_clock\_group 命令，在时序分析中，忽略它们之间的时序路径。

**注：**这个命令比常规的时序例外(后面将译细说明其含义)优先级要高。

### 2. 异步时钟

创建异步时钟组例子的情况：

(1) 在输入端口定义了基本时钟 clk0，然后到达了 MMCM。MMCM 生成 usrclk 和 itfclk 时钟。

(2) 第二个基本时钟 clk1 是在 GTP 例化输出上定义恢复时钟，到达第二个 MMCM。MMCM 生成 gtcclkx 和 gtcclktx 时钟。

使用-asynchronous 选项创建异步组，其格式：

```
set_clock_groups -name async_clk0_clk1 -asynchronous -group {clk0 usrclk itfclk} \
    -group {clk1 gtcclkx gtcclktx}
```

### 3. 独占时钟组

一些设计有几种操作模式，这些操作模式要求不同的时钟。通过一个时钟多路复用

器(例如 BUFGMUXH、BUFGCTRL 或者 LUT),选择不同的时钟。

由于这些单元是可组合的单元,Vivado 将所有进入的时钟传播到输出。在 Vivado 集成开发环境中,在同一时刻,一个时钟树上可以存在几种时序时钟。这样,便于一次报告所有的操作模式。但是,在硬件上是不可能的。

这些时钟称为独占时钟,可以使用 set\_clock\_groups 的选项来约束它们,即- logically \_exclusive 或者-physically\_exclusive 选项。

下面给出独占时钟组例子。一个 MMCM 实例生成 clk0 和 clk1, 它们连接到 BUFGMUX 实例 clk mux。clk mux 的输出驱动设计时钟树。

默认地,即使所有的时钟共享相同的时钟树,它们也不能在同一时刻同时存在。但是,Vivado 集成开发环境可以分析 clk0 和 clk1 之间的路径。

下面的约束用于禁止两个时钟的分析,其格式为:

```
set_clock_groups - name exclusive_clk0_clk1 - physically_exclusive - group clk0 -  
group clk1
```

#### 4. 时钟延迟、抖动和不确定性

除了定义时钟波形外,设计者必须标明与操作条件和环境相关的可预测的和随机的变化。

##### 1) 时钟延迟

当时钟在板上和 FPGA 器件内传播后,经过某个延迟,到达目的地。典型地,它们的延迟由下面表示:

- (1) 源延迟(即时钟源点前的延迟,通常在器件外);
- (2) 网络延迟。

由网络延迟引入的延迟,可能是:

- (1) 自动估计(布线前设计);
- (2) 精确计算(布线后设计)。

很多非 Xilinx 的时序引擎要求 SDC 命令 set\_propagated\_clock,用于触发计算时钟树之间的传播延迟。Vivado 不要求这个命令。默认地,Vivado 计算时钟的传播延迟:

- (1) 将所有时钟认为是传播时钟;
- (2) 一个生成时钟的延迟包括:它的主时钟插入延迟和自身的网络延迟。

对于 Xilinx FPGA 器件,使用 set\_clock\_latency 命令标识器件外的时钟延迟,其格式为:

```
set_clock_latency - source - early 0.2 [get_clocks sysClk]  
Zset_clock_latency - source - late 0.5 [get_clocks sysClk]
```

##### 2) 时钟抖动和不确定性

对于 ASIC 器件,时钟抖动经常用时钟不确定性表示。然而,对于 Xilinx FPGA 来说,抖动属性是可预测的,可以通过时序分析引擎自动计算。

(1) 输入抖动: 输入抖动是连续时钟沿之间的差异(这个差异是与正常或者理想时钟到达时间进行比较而得到的)。使用 set\_input\_jitter 命令,为每个基本时钟指定输入

抖动。设计者不能在生成差异时钟上直接指定输入抖动。Vivado 集成开发环境自动地计算从主时钟继承过来的生成时钟抖动。

① 对于由 MMCM 或者 PLL 驱动的生成时钟, 输入抖动用一个计算出的离散抖动代替。

② 对于由组合或者时序单元创建的生成时钟, 生成时钟的抖动与它主时钟的抖动相同。

(2) 系统抖动: 系统抖动是整体的抖动, 由下列因素引起:

- (1) 供电噪声;
- (2) PCB 板噪声;
- (3) 系统任何额外的抖动。

使用 set\_system\_jitter 命令, 为这个设计只设置一个值。

3) 额外的时钟不确定性

使用 set\_clock\_uncertainty 命令, 为不同的拐点、延迟或者特殊的时钟关系, 定义额外的时钟不确定性。

### 8.6.3 I/O 延迟约束

在设计中, 为了准确地建模外部的时序现场, 设计者必须为输入端口和输出端口给出时序信息。由于 Xilinx Vivado 集成开发环境只在 FPGA 器件的边界内辨识时序, 因此, 设计者必须使用下面的命令, 为这些边界外指定延迟值。

#### 1. 输入延迟

命令 set\_input\_delay, 用于标识在设计接口的输入端口相对于时钟沿的输入路径延迟。当考虑硬件板时, 这个延迟表示下面之间的相位不同:

- (1) 从一个外部芯片通过板子到 FPGA 器件输入封装引脚的数据;
- (2) 相对的参考板时钟。

输入延迟值可以是正的或者负的, 这取决于 FPGA 器件接口的时钟和数据的相对相位。

虽然在 SDC 标准中, -clock 选项是可选的。但是, 在 Vivado 集成开发环境中, 要求 -clock 选项。相对的时钟可以是一个设计时钟, 也可以是一个虚拟时钟。输入延迟命令选项有:

(1) 最小和最大输入延迟命令选项。-min 和-max 选项, 为下面标识不同的值:

- ① 最小延迟分析(保持/去除);
- ② 最大延迟分析(建立/恢复)。

例如:

```
create_clock - name sysClk - period 10 [get_ports CLK0]
set_input_delay - clock sysClk - max 4 [get_ports DIN]
set_input_delay - clock sysClk - min 1 [get_ports DIN]
```

如果没有使用该选项, 则输入延迟值用于最小和最大。

例如：

```
create_clock - name sysClk - period 10 [get_ports CLK0]
set_input_delay - clock sysClk 2 [get_ports DIN]
```

(2) 时钟下降输入延迟命令选项。

-clock\_fall 选项标识输入延迟约束，应用于由相对时钟下降沿发出的时序路径。如果没有该选项，默认地，Vivado 集成开发环境认为相对时钟只有上升沿。

不要将该选项和-rise 和-fall 选项混淆，因为这些选项指向数据沿而不是时钟沿。

(3) 添加延迟输入延迟命令选项。

如果出现下面情况，必须使用-add\_delay 选项：

- ① 存在一个最大(最小)输入延迟约束；
- ② 设计者想在相同端口上，标识第二个最大(最小)输入延迟约束。

这个选项通常用于约束相对于多个时钟沿的一个输入端口(例如 DDR 接口)。

输入延迟选项只能用于输入或者双向端口，而不考虑时钟端口。设计者不可以将输入延迟约束应用于一个内部引脚。

## 2. 输出延迟

命令 set\_output\_delay，用于标识在设计接口的输出端口相对于时钟沿的输出路径延迟。当考虑硬件板时，这个延迟表示下面之间的相位不同：

- (1) 从 FPGA 器件输出封装引脚通过板子到另一个器件的数据。
- (2) 相对的参考板时钟。

输出延迟值可以是正的或者负的，这取决于 FPGA 器件外部的时钟和数据的相对相位。

虽然在 SDC 标准中，-clock 选项是可选的。但是，在 Vivado 集成开发环境中，要求-clock 选项。相对的时钟可以是一个设计时钟，也可以是一个虚拟时钟。输出延迟命令选项有：

(1) 最小和最大输出延迟命令选项。

-min 和-max 选项，为下面标识不同的值：

- ① 最小延迟分析(保持/去除)；
- ② 最大延迟分析(建立/恢复)。

例如：

```
create_clock - name clk_ddr - period 6 [get_ports DDR_CLK_IN]
set_output_delay - clock clk_ddr - max 2.1 [get_ports DDR_OUT]
set_output_delay - clock clk_ddr - max 1.9 [get_ports DDR_OUT] - clock_fall - add_delay
set_output_delay - clock clk_ddr - min 0.9 [get_ports DDR_OUT]
set_output_delay - clock clk_ddr - min 1.1 [get_ports DDR_OUT] - clock_fall - add_delay
```

如果没有使用该选项，则输出延迟值用于最小和最大。

例如：

```
create_clock - name sysClk - period 10 [get_ports CLK0]
```

```
set_output_delay -clock sysClk 6 [get_ports DOUT]
```

(2) 时钟下降输出延迟命令选项。

-clock\_fall 选项标识输出延迟约束,应用于由相对时钟下降沿发出的时序路径。如果没有该选项,默认地,Vivado 集成开发环境认为相对时钟只有上升沿。

不要将该选项和-rise 和-fall 选项混淆,因为这些选项指向数据沿而不是时钟沿。

(3) 添加延迟输出延迟命令选项。

如果出现下面情况,必须使用-add\_delay 选项:

① 存在一个最大(最小)输出延迟约束;

② 设计者想在相同端口上,标识第二个最大(最小)输出延迟约束。

这个选项通常用于约束相对于多个时钟沿的一个输出端口。

输出延迟选项只能用于输出或者双向端口,而不考虑时钟端口。设计者不可以将输出延迟约束应用于一个内部引脚。

#### 8.6.4 时序例外

当以默认的方式不能正确地定时逻辑行为时,需要时序例外(timing exception)。当设计者想以不同的方式处理时序时,就必须使用时序例外命令。表 8.1 给出了 Xilinx Vivado 集成环境中支持的时序例外命令。

表 8.1 时序例外命令

命 令	功 能
set_multicycle_path	指明将数据从路径开始传播到路径结束时,所需要的时钟周期数
set_false_path	指明在设计中不进行分析的路径
set_max_delay	设置最小和最大路径延迟值。它将使用用户指定的最大和最小延迟值覆盖
set_min_delay	默认的建立和保持约束
set_case_analysis	使用逻辑常数或者在端口/引脚上的逻辑跳变执行时序分析,以限制通过设计的信号传播

##### 1. 多周期路径

该约束允许设计者修改建立和保持关系。这种关系是基于设计时钟波形,由定时器确定的。默认的,Vivado 集成开发环境时序引擎执行一个单周期分析。这个分析可能非常严格,对于某些逻辑路径来说,可能不合适。

最常用的例子,例如逻辑路径要求很多时钟周期用于在端点稳定数据。如果路径的起始点和路径的端点的控制电路允许这样,Xilinx 推荐设计者使用 multicycle\_path 约束,放松建立要求。

取决于设计者的意图,保持要求可能仍然维持最初的关系。这帮助时序驱动算法集中在其他较紧要求路径。同时,这也可以帮助减少运行时间。

set\_multicycle\_path 命令用于修改路径要求的乘数(对于建立分析、保持分析或者所有),其相对于源时钟或者目的时钟。语法格式为:

```
set_multicycle_path <path_multiplier> [ -setup | -hold] [ -start | -end]
[ -from <startpoints>] [ -to <endpoints>] [ -through <pins|cells|nets>]
```

设计者必须指定`<path_multiplier>`。定时器使用的默认值是：

- (1) 1, 用于建立分析(或者恢复);
- (2) 0, 用于保持分析(或者去除)。

保持关系和建立关系相关,下面的公式用于计算大多数情况下的保持周期的个数:

$$\text{保持周期} = \frac{\text{建立路径乘子} (\text{setup path multiplier}) - 1}{\text{保持路径乘子} (\text{hold path multiplier})}$$

(1) 默认地,所定义的建立路径乘子是相对于目的时钟的。为了修改相对于源时钟的建立要求,使用`-start`选项。

(2) 类似地,所定义的保持路径乘子是相对于源时钟的。为了修改相对于目的时钟的保持要求,使用`-end`选项。

**注:**如图 8.46 所示,对于每个建立关系,都有两个保持关系。(1)第一个保持关系,保证在活动的捕获沿以前,建立发送沿没有被到达的沿捕获。(2)第二个保持关系,保证在活动的发送沿后的沿没有被活动的捕获沿捕获。时序分析工具计算所有的保持关系,但是在分析和报告中,只保留最“苛刻的”。

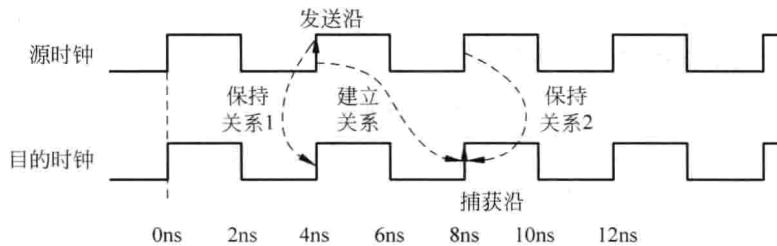


图 8.46 一个路径上建立和保持关系的例子

**注:**当`-start`和`-end`选项用于相同时钟或者两个同一时钟(也就是说,当时钟有相同的相移或者无相移波形)路径的 multicycle path 约束时,没有明显的影响。

表 8.2 说明`-end`和`-start`选项如何影响活动的发送沿和捕获沿。

表 8.2 活动的发送和捕获沿

	源时钟(-start),发送沿移动	目的时钟(-end),捕获沿移动
建立	←→(向后)	→(向前)(默认)
保持	→(向前)(默认)	←→(向后)

**注:**

① `set_multicycle_path` 命令的`-setup`选项,不仅修改建立关系,而且还影响和建立关系相关的保持关系。如果需要将保持关系恢复到最初的位置,需要使用另一条带`-hold`的 `set_multicycle_path` 命令。

② 可以为单个路径、多个路径或者甚至两个时钟之间,设置一个 multicycle 约束。

### 1) 单时钟域上的多周期

如图 8.47 所示,给出了在单时钟域上多周期的约束。如图 8.48 所示,给出了静态

时序分析(Static Timing Analysis, STA)工具给出的默认建立和保持关系。

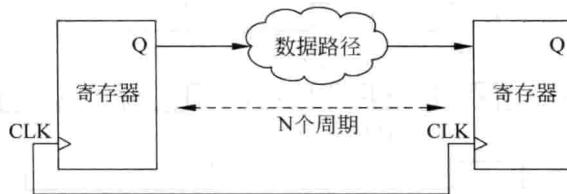


图 8.47 单时钟域上多周期约束

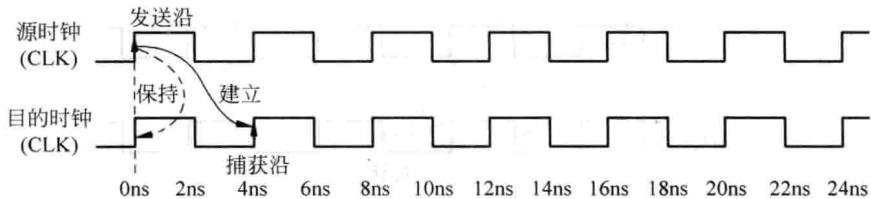


图 8.48 默认的建立和保持关系

建立和保持时序要求：

(1) 建立检查：

$$T_{\text{数据路径(最大)}} = T_{\text{CLK}_{(t=\text{周期})}} - T_{\text{建立}}$$

(2) 保持检查：

$$T_{\text{数据路径(最小)}} = T_{\text{CLK}_{(t=0)}} + T_{\text{保持}}$$

下面的例子说明放松建立，而维持保持。如图 8.49 所示，在两个触发器之间的一个数据路径，该路径每两个周期有效。在这个路径上，定义 multicycle 路径约束，用于：

(1) 目的时钟的第一个沿不是活动的；

(2) 只有目的时钟的第二个边沿捕获新的数据。

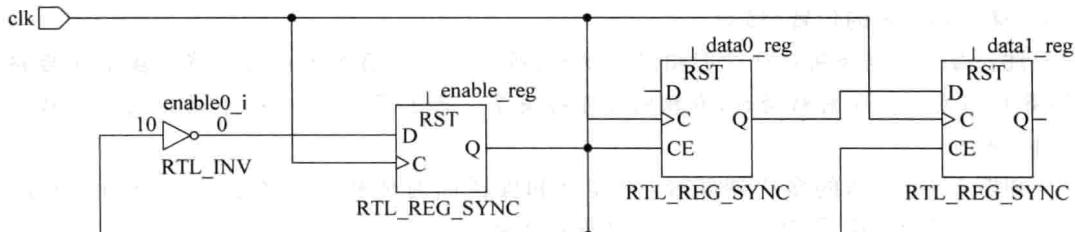


图 8.49 每两个周期使能寄存器

使用下面的约束建立新的建立关系，其格式为：

```
set_multicycle_path 2 - setup - from [get_pins data0_reg/C] - to [get_pins data1_reg/D]
```

当修改建立关系时，为了遵守建立发送沿和捕获沿的变化，自动修改保持关系。

如图 8.50(a)所示，给出了约束前的建立和保持关系；如图 8.50(b)所示，给出了约束后的建立和保持关系。它们表示当多周期路径约束中只有建立路径乘子变化时，建立和保持关系都会发生变化。

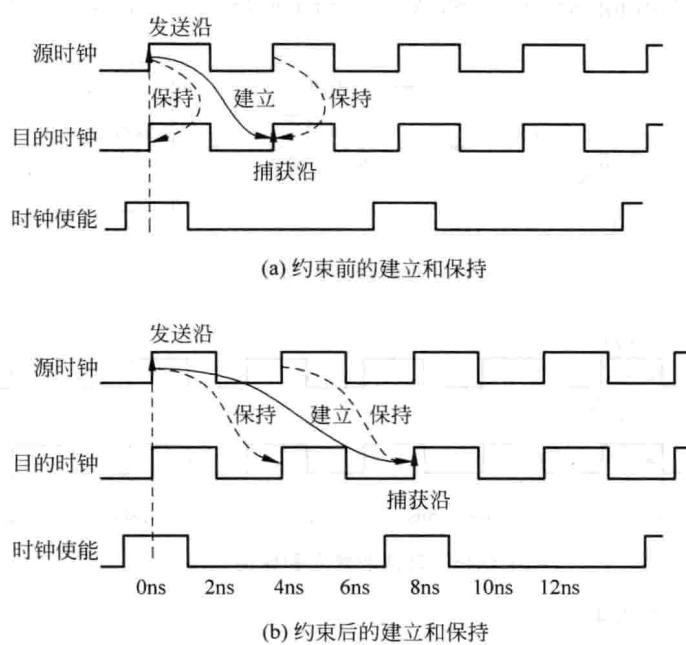


图 8.50 多周期路径：只放松建立

对于这条路径来说,由于存在时钟使能,因此在一个周期内,不需要保持数据 data0\_reg0。在这个情况下,Xilinx 推荐将保持关系恢复到初始状态。添加第二条多周期路径,用于只修改保持检查,其格式为:

```
set_multicycle_path 1 -hold -end -from [get_pins data0_reg/C] \
    -to [get_pins data1_reg/D]
```

在使用 set\_multicycle\_path -hold 命令时,需要-end 选项,这是由于必须要将捕获沿向后(移动到原来的位置)移动。

**注:** 由于发送和捕获时钟有相同的波形,所以-end 选项是可选的。将捕获沿向后移动,导致当发送沿向前移动时,有相同的保持关系。为了简化表达式,下面的例子省略了-end 选项。

如图 8.51 所示的多周期路径,将建立和保持同时放松。该图表示当应用所有的 multicycle 路径约束后,所更新的建立和保持关系。

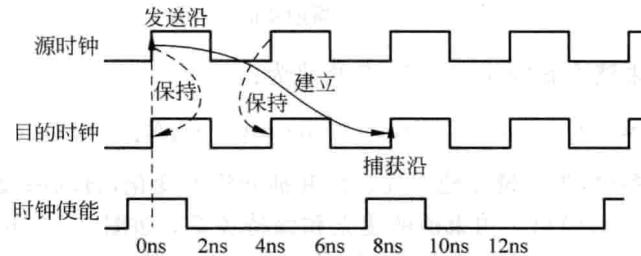


图 8.51 多周期路径：放松建立和保持

总结这个例子,下面的约束是必要的。这些约束用于在 data0\_reg/Q 和 data1\_reg/D 之间,正确定义多通道路径。

```
set_multicycle_path 2 - setup - from [get_pins data0_reg/C] - to [get_pins data1_reg/D]
set_multicycle_path 1 - hold - from [get_pins data0_reg/C] - to [get_pins data1_reg/D]
```

对于带有建立乘子为 4 的多周期约束,其格式为:

```
set_multicycle_path 2 - setup - from [get_pins data0_reg/C] - to [get_pins data1_reg/D]
set_multicycle_path 1 - hold - from [get_pins data0_reg/C] - to [get_pins data1_reg/D]
```

如图 8.52 所示给出了其最终的表示。

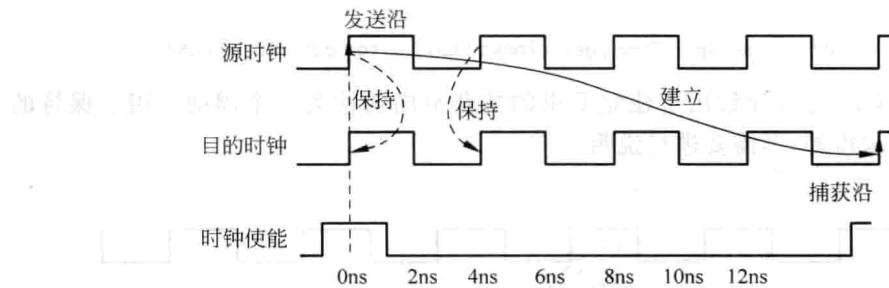


图 8.52 带有建立乘子为 4 的多周期路径

## 2) 多周期路径和时钟相位移动

有时候必须在两个时钟域内定义一个时序约束。这两个时钟有相同的周期,但是它们之间有相位移动。所以,必须了解时序引擎如何使用默认的建立和保持关系。如果不仔细地进行调整,在两个时钟之间的相位移动可能会导致对两个时钟域的逻辑进行过分的约束。如图 8.53 所示给出了多个周期的路径和相位移动。



图 8.53 多周期路径和时钟相位移动

对于一个例子,假设:

- (1) 两个时钟 CLK1 和 CLK2 有相同的波形;
- (2) CLK2 移动 +0.3ns。

通过下面,时序引擎计算建立关系:

- (1) 查看所有波形上的所有边沿;
- (2) 在发送和捕获时钟上,选择导致较严格约束的两个边沿。

由于时钟相移,时序引擎使用的建立和保持关系可能不是期望的那样。如图 8.54 所示给出没有使用多周期路径约束的、默认的情况。

在这个例子中,由于相移 0.3ns,所以有建立约束。这使得最不可能实现时序收敛。另一方面,保持检查是 -3.7ns。

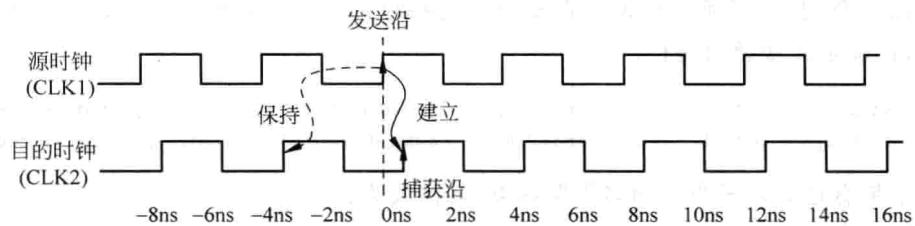


图 8.54 没有多周期路径约束的相移路径

因此,必须调整建立和保持边沿,以满足设计者的目的。添加下面建立乘子为 2 的多周期路径约束,其格式为:

```
set_multicycle_path 2 -setup -from [get_clocks CLK1] -to [get_clocks CLK2]
```

如图 8.55 所示,这将导致用于建立要求的捕获沿向前移动一个周期。用于保持的默认沿由建立要求得到,不需要进行说明。

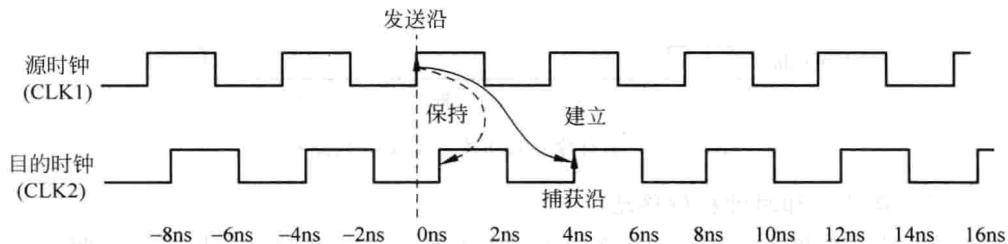


图 8.55 添加多周期路径约束后的路径

如图 8.56 所示,在两个时钟域之间负相移的情况下,用于建立检查和保持检查的发送沿和捕获沿与前面类似(单时钟域,没有相位移动)。

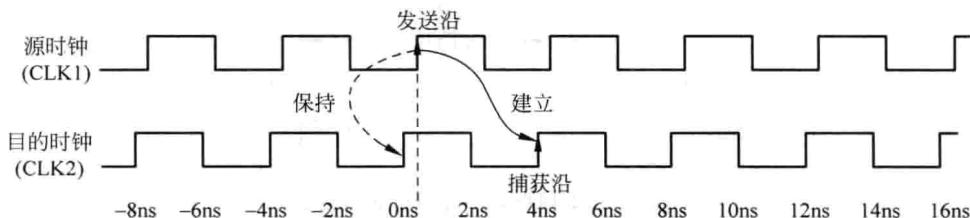


图 8.56 负相移路径的默认关系

### 3) 从慢到快时钟的多周期

如图 8.57 所示,发送时钟 CLK1 是慢时钟,捕获时钟 CLK2 是快时钟。

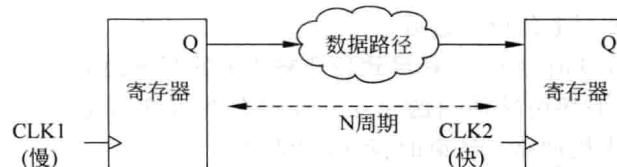


图 8.57 在慢到快时钟多周期

对于一个例子,假设:

- (1) CLK2 的频率是 CLK1 的三倍;
- (2) 在接收寄存器上的一个时钟使能信号,允许在时钟之间设置多周期约束。如图 8.58 所示,给出了在慢到快时钟的多周期。

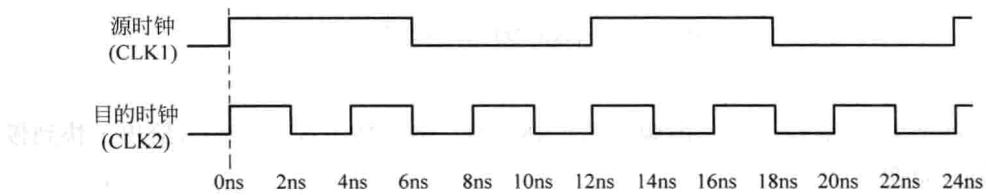


图 8.58 从慢到快时钟多周期

如图 8.59 所示,给出了没有使用多周期约束时,STA 工具解析的建立和保持关系。

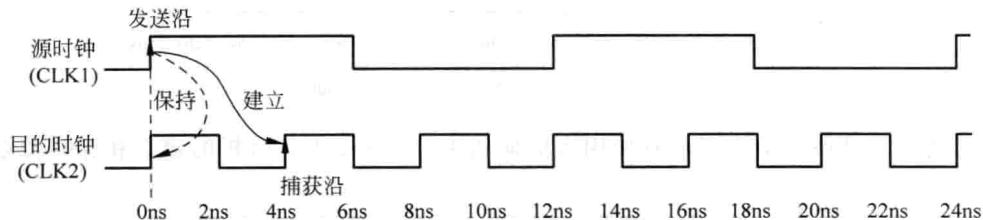


图 8.59 默认的建立和保持关系

下面给出一个例子,该例子定义了一个建立乘子为 3 的多周期路径约束,其格式为:

```
set_multicycle_path 3 - setup - from [get_clocks CLK1] - to [get_clocks CLK2]
```

建立乘子的结果将用于建立检查的捕获时钟向前移动 2 个周期(即: 3-1 周期)。由于没有说明保持乘子,则工具从建立发送和捕获沿得到保持关系。多周期约束没有修改发送时钟活动沿。如图 8.60 所示,给出了 setup=3,保持相应移动的情况。

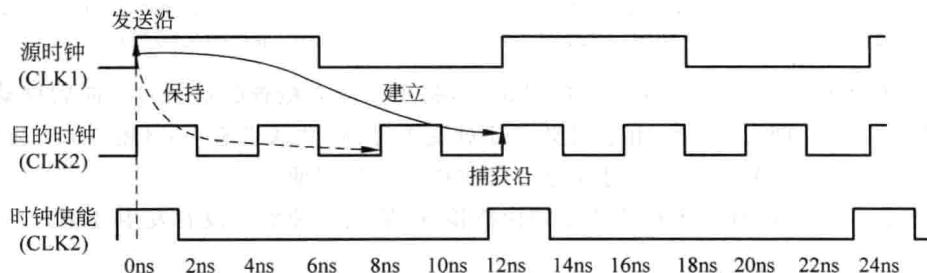


图 8.60 setup=3,保持相应的移动

在这个路径上,没有必要在发送寄存器内保持一个 CLK2 周期的数据。如果这样做,将增加不必要的逻辑,它将增加面积和消耗功率。

由于接收寄存器有一个时钟使能信号,放松保持要求是安全的,并不存在亚稳定状态的风险。

#### 4) 从快到慢时钟的多周期

如图 8.61 所示,发送时钟 CLK1 是快时钟,捕获时钟 CLK2 是慢时钟。

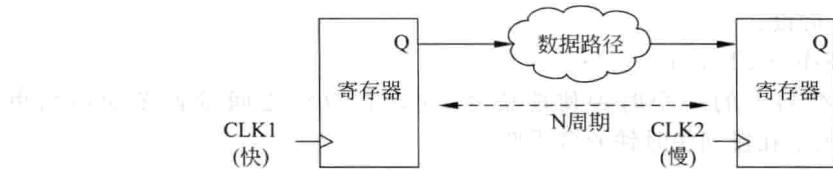


图 8.61 从快到慢时钟多周期

对于这个例子,假设 CLK1 的频率是 CLK2 的三倍。如图 8.62 所示,给出了快到慢时钟之间多周期。

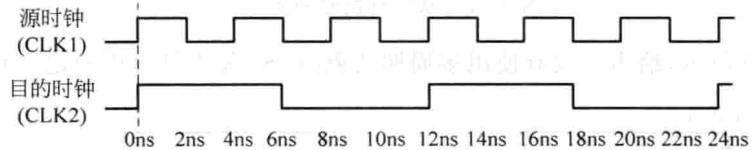


图 8.62 从快到慢时钟之间多周期

如图 8.63 所示,给出了没有使用多周期约束时,STA 工具解析的建立和保持关系。

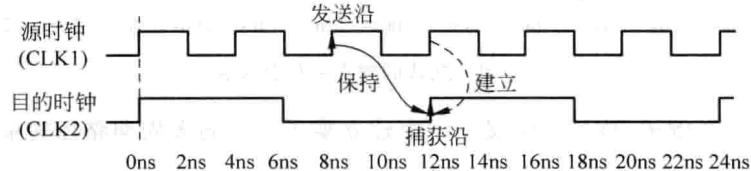


图 8.63 默认的建立和保持关系

下面给出一个例子,该例子针对发送时钟 (-start) 定义了一个建立乘子为 3,保持乘子为 1 的多周期路径约束,其格式为:

```
set_multicycle_path 3 - setup - start - from [get_clocks CLK1] - to [get_clocks CLK2]
set_multicycle_path 2 - hold - from [get_clocks CLK1] - to [get_clocks CLK2]
```

针对发送时钟 (-start) 定义乘子的结果是,将用于建立检查的捕获时钟向后移动 2 个周期(即:3-1 周期)。然而,由于定义了针对发送时钟(默认带有 -hold 的 -start 选项)的保持乘子,用于保持关系的发送时钟边沿向前移动 2 个周期。

如图 8.64 所示,对于所有的建立和保持检查,捕获时钟沿并没有发生变化。

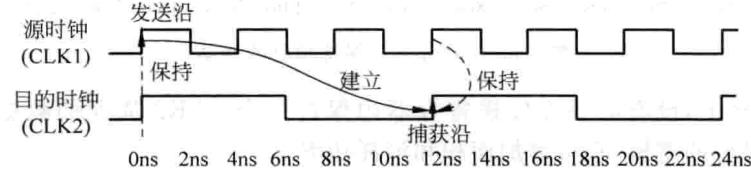


图 8.64 set up=3(-start), hold=2 的多周期路径

注:对于快到慢的时钟域跨越 (Clock Domain Crossing, CDC),针对发送时钟 (-start) 定义一个建立乘子 N,保持乘子为 N-1(大多数通用情况),其格式为:

```
set_multicycle_path N - setup - start - from [get_clocks CLK1] - to [get_clocks CLK2]
set_multicycle_path N-1 - hold - from [get_clocks CLK1] - to [get_clocks CLK2]
```

表 8.3 总结了前面的结果。

表 8.3 定义带有建立 N…的一个多周期路径

情 景	多周期约束
相同时钟域或者两个之间的同步时钟域(相同周期,没有相移)	set_multicycle_path N -setup -from CLK1 -to CLK2 set_multicycle_path N-1 -hold -from CLK1 -to CLK2
在慢到快同步时钟域之间	set_multicycle_path N -setup -from CLK1 -to CLK2 set_multicycle_path N -hold -end -from CLK1 -to CLK2
在快到慢同步时钟域之间	set_multicycle_path N -setup -start -from CLK1 -to CLK2 set_multicycle_path N -hold -from CLK1 -to CLK2

## 2. 假路径

假路径是指设置中的拓扑结构上存在,但是:

- (1) 没有起作用;
- (2) 不需要确定时序。

所以,在时序分析期间,将忽略假路径。假路径的例子包括:

- (1) 跨越时钟域,其中添加了二倍频合成器逻辑;
- (2) 在上电时,可能被写一次的寄存器;
- (3) 复位或者测试逻辑;
- (4) 当可应用的时候,忽略一个分布式 RAM 写和异步读时钟之间的路径。

如图 8.65 所示,给出了一个不起作用路径的例子。图中所有的多路复用器都有相同的选择信号驱动,不存在从 Q 到 D 的路径。所以,应该被定义为假路径。

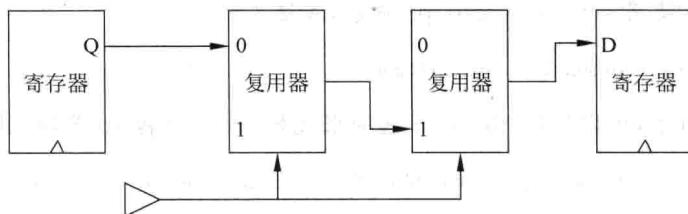


图 8.65 没有起作用的路径

注: 当下面情况时,使用多周期约束代替一个假路径约束:

- (1) 目的只是为了放松同步路径上的时序要求;
- (2) 仍然要求对路径进行定序、验证和优化。

从时序分析中,去掉假路径的原因是:

- (1) 减少运行时间;
- (2) 改善结果的质量。

这是因为对设计的综合、实现和优化,很大程度上受到时序问题的影响。如果不起作用的路径出现时序冲突,则工具试图修复这些路径,而不是用于真正起作用的路径。

使用下面的 XDC 命令定义假路径,其格式为:

```
set_false_path [-setup] [-hold] [-from <node_list>] [-to <node_list>] \
[-through <node_list>]
```

其中:

- (1) 用于-from 选项的节点列表应该是有效起始点的列表,包括:一个时钟对象、一个时序元件的一个时钟引脚、或者一个输入(输出)基本端口。
- (2) 用于-to 选项的节点列表应该是有效起始点的列表,包括:一个时钟对象、一个时序元件输入数据引脚的一个输出(输入)基本端口。
- (3) -through 选项,应该是有效引脚或者端口的列表。

**注:**要谨慎使用-through 选项。并且-through 选项的顺序也非常重要。下面的两个命令不同:

```
set_false_path -through cell1/pin1 -through cell2/pin2
set_false_path -through cell2/pin2 -through cell1/pin1
```

下面的例子将去除从 reset 端口到所有寄存器的时序路径。

```
set_false_path -from [get_port reset] -to [all_registers]
```

下面的例子禁止两个异步时钟域之间的时序路径(例如从 CLKA 到 CLKB)。

```
set_false_path -from [get_clocks CLKA] -to [get_clock CLKB]
```

下面的例子使用两个 set\_false\_path 明令禁止两个时钟域任何方向的所有路径。

```
set_false_path -from [get_clocks CLKA] -to [get_clock CLKB]
set_false_path -from [get_clocks CLKB] -to [get_clock CLKA]
```

**注:**当两个或者多个时钟域是异步的,并且禁止在任何方向上这些时钟域之间的路径时,Xilinx 推荐使用 set\_clock\_groups 命令,其格式为:

```
set_clock_group -group CLK -group CLKB
```

如图 8.66 所示,可以使用-through 选项来代替-from 或者-to 选项,其格式为:

```
set_false_path -through [get_pins MUX1/a0] -through [get_pins MUX2/a1]
```

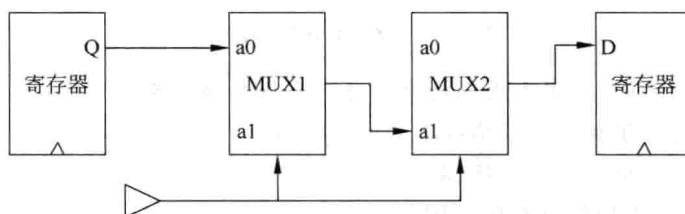


图 8.66 没有起作用路径的例子

另一个常用的例子是在设计中的异步双口 RAM,写操作是同步的,而读操作是异步的。在这种情况下,在写和读时钟之间的时序路径上设置假路径是安全的。下面给出两

种不同的方法：

(1) 从 RAM 前的写寄存器到 RAM 后接收读时钟的寄存器之间，定义假路径，其格式为：

```
set_false_path -from [get_cells <write_registers>] -to [get_cells <read_registers>]
```

(2) 从 RAM 写引脚开始定义假路径，其格式为：

```
set_false_path -from [get_cells -hier -filter {REF_NAME =~ RAM* && IS_SEQUENTIAL && NAME =~ < PATTERN_FOR_DISTRIBUTED_RAMS >}]
```

### 3. 最大/最小延迟

设计者可以使用最大/最小延迟命令来覆盖默认的最大/最小延迟，其格式为：

```
set_max_delay <delay> [-datapath_only] [-from <node_list>]
[-to <node_list>] [-through <node_list>]
set_min_delay <delay> [-from <node_list>]
[-to <node_list>] [-through <node_list>]
```

其中：

(1) 对于-from 选项，有效的起始点是一个时钟、一个输入(或者输出)端口、或者时序元件上的时钟引脚。使用无效的节点，将导致分割路径。

(2) 对于-to 选项，有效的起始点是一个时钟、一个输出(或者输入)端口、或者时序元件上的数据引脚。使用无效的节点，将导致分割路径。

(3) 对于-through 选项，节点列表是有效的引脚、端口或者网络。

(4) 默认地，时序引擎包含松弛计算内的时钟抖动。使用-datapath\_only 选项，可以从松弛计算中去除时钟抖动。该选项只能用于 set\_max\_delay 命令，并且要求-from 选项。

**注：**路径分割将导致不期望的结果。为了避免路径分割，应该谨慎使用约束。

(1) 在一个路径上设置最大延迟和最小延迟约束的结果。

在路径上，设置最大延迟约束时，如果没有使用-datapath\_only 选项，不会修改路径上的最小要求。路径上的保持(去除)检查保持其默认状态。如果使用-datapath\_only 选项，将导致保持要求成为这些路径上的假路径。

类似地，在路径上设置最小延迟约束时，不修改默认的建立(恢复)检查。

例如，如果一个路径只有最大延迟要求，可以使用 set\_max\_delay 和 set\_false\_path 命令的组合。

(2) 约束输入和输出逻辑。

set\_max\_delay 命令和 set\_min\_delay 命令不用于约束输入和输出逻辑。典型地，输入端口和第一级寄存器之间使用 set\_input\_delay 延迟命令。这个命令提供了选项，用于将时钟和输入端口关联。

同样地，对于最后一级寄存器和输出端口之间使用 set\_output\_delay 命令进行约束。

典型地，set\_max\_delay 命令和 set\_min\_delay 命令用于约束基于输入端口和基于输

出端口之间单纯的组合逻辑路径。

### (3) 约束异步信号。

`set_max_delay` 信号可以用于约束异步信号,这些异步信号:

- (1) 没有一个时钟关系;
- (2) 但是要求最大的延迟。

例如,使用 `set_clock_groups` 命令(推荐)或者 `set_false_path`(不推荐),可以禁止两个异步时钟域之间的时序路径。

如果为两个时钟域之间的一些或者所有路径指定最大延迟,则必须使用 `set_maxdelay -datapath_only` 来约束路径。在这种情况下,不能使用 `set_clock_groups` 将两个时钟域定义为异步的,因为在优先级上它可以替代 `set_max_delay` 约束。其他跨时钟域路径必须使用 `set_false_path` 或者 `set_max_delay` 约束的组合进行约束。

## 4. Case 分析

在一些设计中,某些信号在特定模式下是一个常数。例如,在工作模式下,测试信号不会切换,其或者为 VDD 或者为 VSS。这也用于一些信号,这些信号在上电后不会切换。同样的,在当今的设计中,有多个工作模式。在一些工作模式下,一些信号是活动的;而另一些工作模式下,信号是不活动的。

为了帮助减少分析空间、运行时间和存储器开销。让静态时序引擎知道有常数值的信号,这是非常重要的。这样,保证不会报告不起作用的或者不相关的路径。

使用 `set_case_analysis` 命令,声明信号对于时序引擎来说是不活动的。该命令应用于引脚和/或端口,其格式为:

```
set_case_analysis <value> <pins or ports objects>
```

其中:

- (1) `value` 可以是 0、1、rise、rising、fall 或者 falling。

当指定 rise/rising,或者 fall/falling 时,表示对给定的引脚或者端口只能用指定的跳变进行时序分析,禁止其他跳变。

- (2) 可以在一个端口、一个叶子单元或者一个层次模块的引脚上设置一个 case 值。

图 8.67 给出了一个时钟的例子。在这个例子中,在多路复用器的输入引脚上提供了两个时钟。但是,当在选择端引脚 S 设置一个常数值后,只有 `clk_2` 通过输出引脚传播。其约束命令格式为:

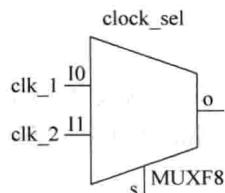


图 8.67 时钟的例子

```

create_clock - name clk_1 - period 10.0 [get_pins clock_sel/I0]
create_clock - name clk_2 - period 15.0 [get_pins clock_sel/I1]
set_case_analysis 1 [get_pins clock_sel/S]

```

### 8.6.5 时序约束实现

为了更好地满足设计时序的要求,设计者可以为设计创建时序约束,时序约束经常用于综合后和实现前。下面给出创建时序约束的示例步骤,其步骤主要包括:

注:具体约束的对象随设计要求会发生变化,本节给出的约束仅供读者参考。

(1) 在 Vivado 主界面左侧 Flow Navigator 窗口下,找到并展开 Synthesis。在展开项中,找到并展开 Synthesized Design。在展开项中,选择并单击 Edit Timing Constraints。

(2) 如图 8.68 所示,出现 Timing Constraints(时序约束)对话框界面。

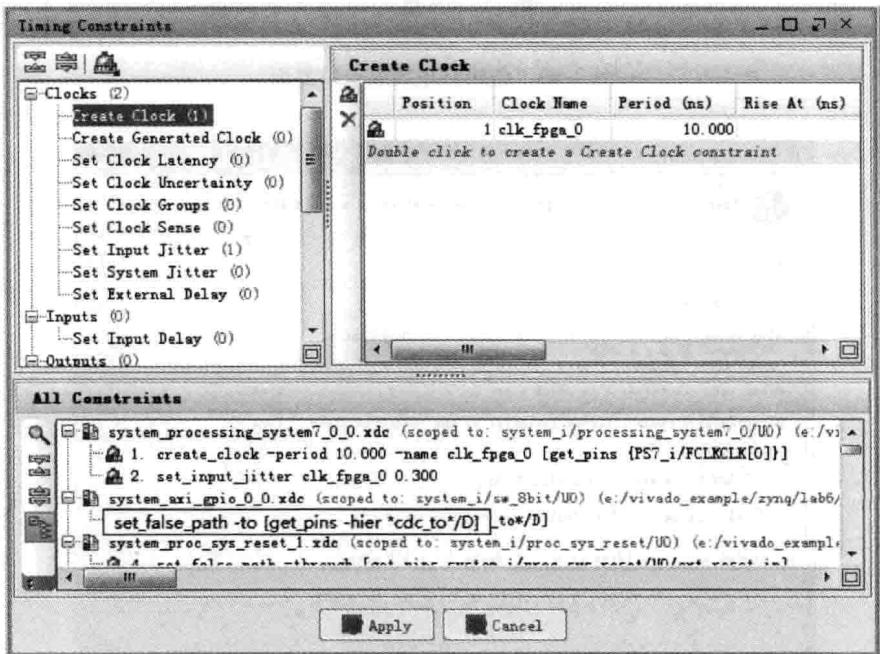


图 8.68 创建时序约束对话框界面

- (3) 如图 8.69 所示,例如,在左侧窗口下,选择 Set Input Delay。
- (4) 在图中右侧 Set Input Delay 窗口下,单击右键,出现浮动菜单。在浮动菜单内,选择 Create Constraint。
- (5) 如图 8.70 所示,出现 Set Input Delay 对话框界面。在该界面中,读者可以按照设计要求,给定相关延迟参数。
- (6) 单击 OK 按钮。

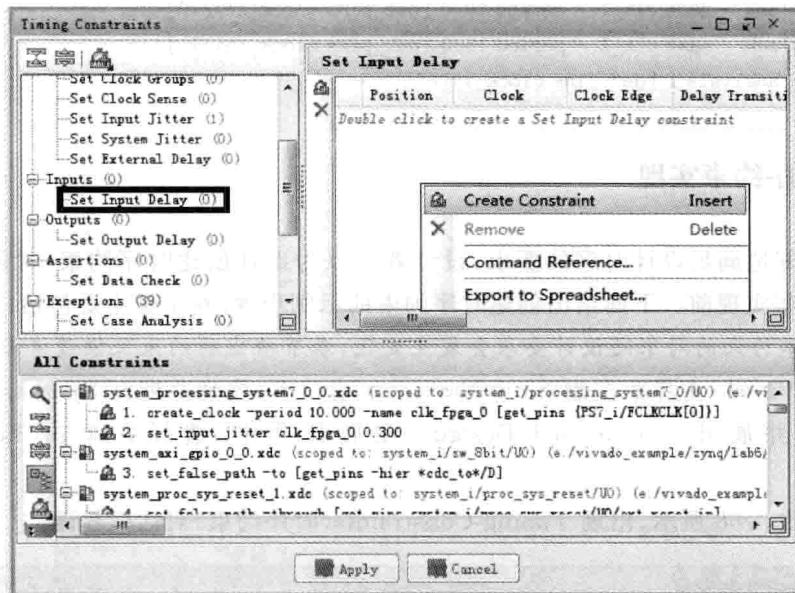


图 8.69 创建时序约束入口界面

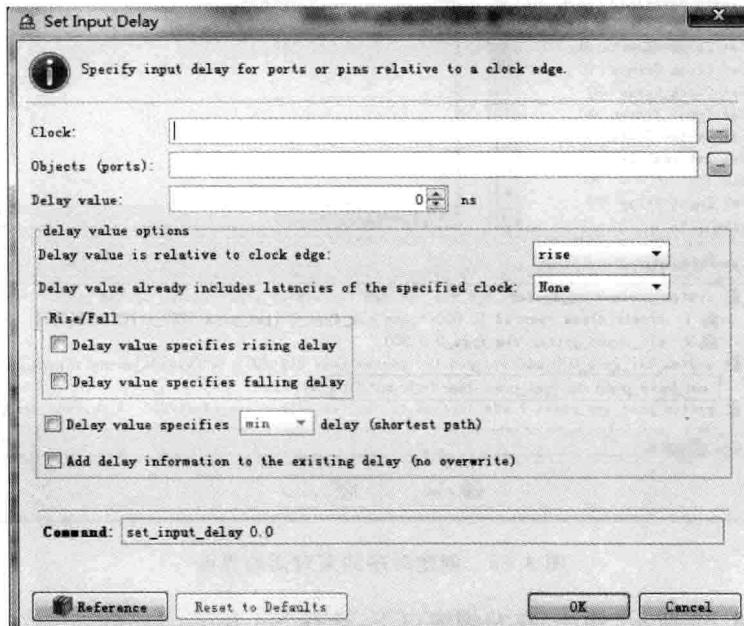


图 8.70 设置输入延迟序约束界面

# 附录 XDC 中有效的命令

## 1. 时序约束命令

```
create_clkok  
create_generated_clock  
group_path  
set_clock_groups  
set_clock_latency  
set_data_check  
set_disable_timing  
set_false_path  
set_input_delay  
set_output_delay  
set_max_delay  
set_min_delay  
set_multicycle_path  
set_case_analysis  
set_clock_sense  
set_clock_uncertainty  
set_input_jitter  
set_max_time_borrow  
set_propagated_clock  
set_system_jitter  
set_external_delay
```

## 2. 物理约束命令

```
add_cells_to_pblock  
create_pblock  
delete_pblock  
remove_cells_from_pblock  
resize_pblock
```

```
create_macro  
delete_macros  
update_macro
```

### 3. 网表约束命令

```
set_load  
set_logic_dc  
set_logic_one  
set_logic_zero  
set_logic_unconnected
```

### 4. 网表对象查询命令

```
all_cpus  
all_dspss  
all_fanin  
all_fanout  
all_hsios  
all_inputs  
all_outputs  
all_rams  
all_registers  
all_ffs  
all_latches  
get_cells  
get_nets  
get_pins  
get_ports  
get_debug_cores  
get_debug_ports
```

### 5. 器件对象查询命令

```
get_iobanks  
get_package_pins  
get_sites  
get_bel_pins  
get_bels  
get_nodes  
get_pips  
get_site_pins
```

get\_site\_pips  
get\_slrs  
get\_tiles  
get\_wires

#### 6. 时序对象查询命令

all\_clocks  
get\_path\_groups  
get\_clocks  
get\_generated\_clocks  
get\_timing\_arcs

#### 7. 布局对象查询命令

get\_pblocks  
get\_macros

#### 8. 通用命令

set  
expr  
list  
filter  
current\_instance  
get\_hierarchy\_separator  
set\_hierarchy\_separator  
get\_property  
set\_property  
set\_units  
endgroup  
startgroup