

述。然后将这个描述编译成一种比等效RTL描述快1000倍以上的形式，以此验证算法的正确性。

在第9章介绍的基于HDL的设计流程中，这种表示算法的C/C++描述随后被手工翻译成VHDL或者Verilog形式的RTL代码。C/C++描述通常还继续被作为一个黄金参考模型，它可以与RTL仿真器联系起来，并与RTL仿真同步运行。随后将C/C++描述的输出结果与RTL模型的输出结果进行比较，以确保它们在功能上完全一样。

另外，也可以用基于SystemC设计流程的一种变体，即在原始的C/C++模型中增加一些时序信息、并行特征以及引脚定义，等等，以便将这个模型转换到一个新的层次上，并保持与基于SystemC的RTL或者行为综合的一致性。

另一个基于SystemC设计流程的变体中，可以先利用系统、算法或者事务级结构输入设计，这些结构可用于高抽象级的验证。随后再将这种描述进行修改，以将其降低到与基于SystemC的RTL或者行为综合相一致的层次上。

不管人们通过何种途径达到这一目的，我们总是作如下假设：用SystemC描述的设计可以进行SystemC的行为或者RTL综合。这种情况下，主要有两个可选的设计流程，它们是（1）自动将SystemC翻译成RTL级的VHDL或Verilog代码，然后再使用传统的RTL综合技术；（2）使用SystemC的综合技术直接生成实现级的网表。

对此存在两种不同的看法。一种认为直接将SystemC综合为实现级网表是最干净、最快速、最有效的方法。但是另一种看法则认为，先将SystemC翻译成RTL级的VHDL或Verilog更可取，因为对于设计工程师而言，RTL才是他们真正的思考方式；而且RTL是集成多方提供的设计模块（包含第三方IP）的一个自然分界点；另外Verilog/VHDL综合技术是非常成熟而且强大的（与SystemC综合技术相比）。

不过我们有些跑题了。这些流程既适用于ASIC设计又适用于FPGA设计，如图11-5所示。

第1个SystemC综合应用软件主要面向ASIC设计流程，所以在推测针对FPGA的实体时效果不是很好，如嵌入式RAM、嵌入式乘法器等。后来出现的这种软件在这方面表现更好，但是不同工具表现出来的复杂程度也有很大差异，所以这些工具厂商强烈建议用户在决定购买工具前一定要深入地进行性能评估。

注意，图11-5说明了用针对实现的SystemC来驱动ASIC和FPGA设计流程。一旦你开始在RTL级编写代码并增加一些时序特性，不管是用VHDL、Verilog还是SystemC，要实现一个优化的设计总是需要带着专门目标架构的想法来编写代码。

再说一次，同样的SystemC既可以用于ASIC设计流程也可以用于FPGA设计流程，但是通常要付出一定的代价。如果SystemC代码最初是以FPGA为目标器件设计的，而随后被用在了ASIC设计流程中，那么最终的ASIC一般要比专门针对ASIC架构设计的SystemC代码占用更大的面积和消耗更多的功率。同样，如果代码最初是为ASIC设计的，随后被用在FPGA中，最后的FPGA实现一般要比专门针

202

203

对FPGA设计的SystemC代码具有更低的性能。这主要是由源代码中的微观结构用硬编码方式定义造成的。

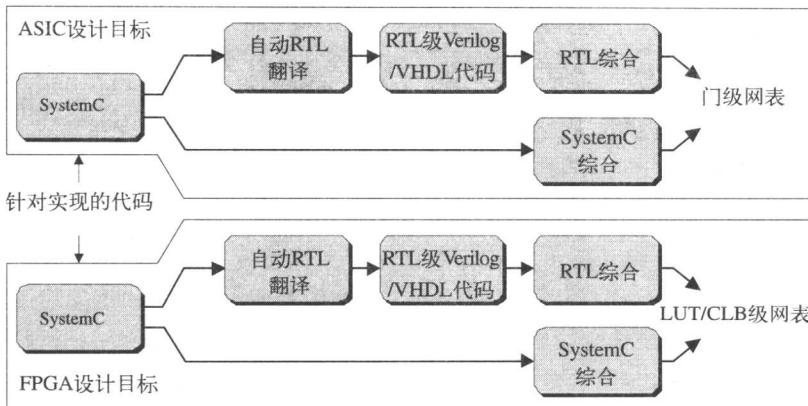


图11-5 可选SystemC设计流程

11.3.6 要么喜爱它，要么讨厌它

与不同的人交谈，你就会发现有的喜欢SystemC，而有的则讨厌。大多数人都会同意这样一个观点：SystemC 2.0是很有前途的，没有其他语言能够提供与之相同的功能（其中的一些功能正在加入到SystemVerilog中，但不是全部）。

遗憾的是，许多设计工程师都对C语言编程非常精通，但是他们中大多数对面向对象的C++却不熟悉。所以要他们使用SystemC，就等于给了他们强大的“武器”，推动他们进入一个自己不喜欢或者不了解的世界里。SystemC在验证以及高层次系统建模中可能非常有用，不过在有些方面，相对实际实现流程中的工具而言还相当不成熟。

有一种看法认为，尽管用SystemC写代码难度很大而且也很难综合，导致人们认为它是一种笨拙的规范语言，但是它却为各种语言和各种抽象级别的仿真提供了一个强大的框架。

写作本书时，美国有许多公司已经在过去几年内成长为SystemC的坚定支持者。另一方面，在欧洲和亚洲地区，SystemC也正在成长。未来会是什么样的？几年以后我会高兴地告诉你！

204

11.4 基于增强型C/C++的设计流程

11.4.1 什么是增强型C/C++

有两种方法可以对标准C/C++进行增强，从而扩展其功能和描述范围。第一种

是在纯C/C++代码中增加一些专门的注释，有人将其称为注释指令或编译指令(pragma)。这些注释随后由语法分析器、预编译器和编译器以及其他工具进行识别和解释，并据此在代码中增加一些特定的结构或者修改该工具的处理方式^①。这种方法有一个重大的缺陷就是仿真需要使用专门的C/C++编译器，而不能用现有的标准编译器。这就限制了用户的选择，而且如果标准开发出来是为了平衡各家EDA厂商，这种方法就是唯一可行的。

另一种方法是在语言中增加专门的关键字和语句。这是一种非常流行的技术，语言中这类变化非常广泛，每一种都针对不同的应用领域。这种方法的缺点是，也需要专门的C/C++编译器；另外，仿真器等工具还没有增强功能支持这些新的关键字和语句。**205**解决这个问题的一个普遍做法是在新的关键字和语句附近增加标准的#define指令，以使预编译器在必要的时候能够忽略它们（这听起来有些不雅，不过确实能够工作）。

在描述ASIC和FPGA设计的功能时，有必要使用一些专门的语句增强标准的C/C++，以支持某些概念，例如时钟(clock)、引脚(pin)、并发(concurrency)、同步(synchronization)和资源共享(resource sharing)^②。

如果原始的设计模型是用纯C/C++编写的，那么第一步就是增加一些时钟语句，再加上一些定义输入和输出引脚的接口语句。接着利用适当的综合工具产生出一个具体的实现（下面将会讨论）。但是，由于C/C++本质上是顺序执行的，因此如果综合工具不能找到可能存在的并行性并加以利用，最终产生的硬件将会非常慢，没有任何效率。

例如，假定设计中有如下的C/C++语句：

```
a = 6; /* Standard C/C++ statement */
b = 2; /* Standard C/C++ statement */
c = 9; /* Standard C/C++ statement */
d = a + b; /* Standard C/C++ statement */
:
etc
```

206默认情况下，综合工具假定每一个赋值符号(=)都代表一个时钟周期。这样，如果以上代码就保持原样，增强型的C/C++综合工具将会产生如下的硬件实现，在第1个时钟周期将数值6装入变量(寄存器)a中，第2个时钟周期将2装入b中，第3个时钟周期则把9装入c中，依此类推。所以，按照硬件的标准来衡量，这是非常慢的。

① 这种类型的C/C++增强的一个例子就是0-In设计自动化公司(www.0in.com)提供的基于断言的验证技术(assertion-based verification, ABV)。另一个特别合适的例子是Future设计自动化公司(www.future-da.com)，该公司在其C/C++到RTL的综合引擎中使用了这种增强技术。

② 用于ASIC和FPGA设计输入、仿真和综合的C/C++增强方案中，Celoxica公司(www.celoxica.com)及其Handel-C语言是一个很主要的参与者。

当然，大多数综合工具能够在上面的例子中找到可能存在的并行性操作并加以利用，但是这些工具还不能处理更加复杂的情况，如需要人参与时。为了本章讨论的方便，我们继续用这个简单的测试用例来说明。增强型C/C++语言有一些新的关键字，例如parallel（或par）和sequential（或seq），这些关键字可以指导下游的综合软件，哪些语句应该并行执行、哪些应该顺序执行，等等。例如：

```
parallel; /* Augmented C/C++ statement */
a = 6; /* Standard C/C++ statement */
b = 2; /* Standard C/C++ statement */
c = 9; /* Standard C/C++ statement */
sequential; /* Augmented C/C++ statement */
d = a + b; /* Standard C/C++ statement */
:
etc
```

正如前面所提到的，仿真器等工具还没有增强到可以理解这些新的关键字和语句，因此当这些工具处理这些描述的时候会出现问题。一种解决方案是用标准的#define指令将新的关键字和语句封装起来，这样预编译器在必要的时候就可以忽略它们。但是，这就意味着，仿真器和综合工具所处理的设计有一些差异，这可不是个好主意。另一个解决方案是使用专用仿真器，但是这种仿真器可能不具备现有仿真技术的能力或容量。

这种情况下，并行声明会告知综合工具，后面的语句需要以并行方式实现，而顺序声明则意味着各个操作必须按一定的顺序执行。当然，这些并行和顺序语句在必要的时候可以嵌套使用。

当出现循环的时候，事情就变得更加复杂了，具体情况与设计者希望是部分拆开还是全部拆开这些循环有关。在这里，我们给出一个循环的例子，仅仅作为参考，如`for i= 1 to 10 in increments of 1 do xxxx, yyyy, and zzzz`。有些情况下，简单地将并行或顺序语句做成循环结构是可行的，但是如果需要技巧性更强些，设计者就必须完全重写这些结构。

如果需要进行资源共享，也可能需要增加share声明，另外，channel语句可以实现在各个表达式之间共享信号。

207

11.4.2 可选择的增强型C/C++设计流程

要达到某个目的，通常有很多种方法。正如前面提到的，一个设计通常总是从一个算法开始的，而算法大多数情况下都用C/C++来描述。在随后的验证中，还要对这个C/C++模型加以修改，增加一些描述时钟、引脚、并发性、同步和资源共享等特征的语句，以使这个模型适合于综合工具处理。或者，设计可能用

C/C++语言。

不管设计的具体实现方式是什么，我们总是假定用增强型C/C++描述的设计都是可以综合的。再重申一次，主要有两种设计流程可供选择，它们是（1）将增强型C/C++代码自动翻译成RTL级的Verilog或VHDL代码，然后使用传统的RTL综合技术；（2）使用适当的增强型C/C++综合工具。

有一种看法是，直接将增强型C/C++描述综合成实现级的网表是最干净、最快速、最高效的。另一种看法是，RTL级的Verilog/VHDL代码才是设计集成的最本质阶段，而且今天的RTL综合技术已经非常成熟和强大。

在ASIC和FPGA为目标的设计中，这两种流程都可以使用，如图11-6所示。起初的增强型C/C++综合软件主要是面向ASIC流程。这就是说，这些早期的软件在推断FPGA专用结构时的表现不是很让人满意，例如嵌入式RAM、嵌入式乘法器等。稍后的一些软件表现有所改善，但是，我们一般强烈建议用户在打算花掉辛苦挣来的钱之前做深入的评估。

208

需要注意的是，图11-6说明了使用针对专门实现的代码来驱动ASIC和FPGA设计流程，因为要实现一个最优化的设计需要带着专门的目标结构的想法编写代码。实际上，ASIC和FPGA设计流程可以使用同样的代码，只不过通常会有一些性能方面的损失（见有关SystemC的讨论）。

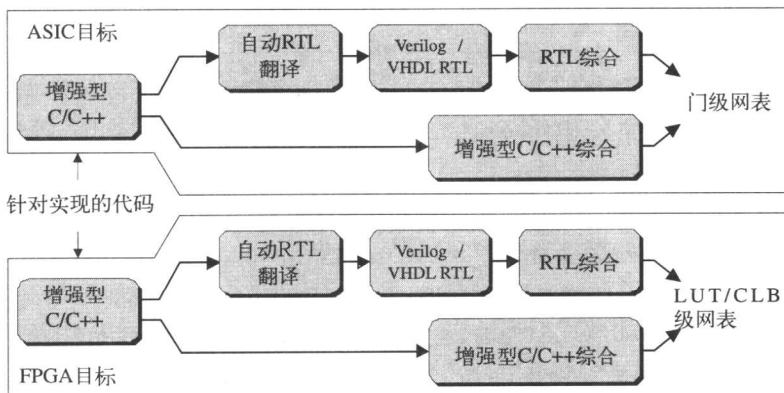


图11-6 可选择的增强型C/C++设计流程

11.5 基于纯C/C++的设计流程

最后，我们来介绍一下基于纯C/C++的设计流程^①。实际上，所谓的纯C/C++

^① 写本书时，纯C/C++流程中最好的例子可能是Mentor公司 (www.mentor.com) 的Precision C Synthesis软件。还有一个是SPARK C-to-VHDL综合工具，由美国加州大学嵌入式计算机系统中心的San Diego和Irvine(www.cecs.uci.edu/~spark) 开发。

是指工业标准的C/C++再用SystemC的数据类型进行最低限度的增强，以使变量和常量具备专门的位宽描述。

1904年，英国，John Ambrose Fleming发明了真空二极管整流器。

尽管基于纯C/C++的设计流程还是一个比较新的概念，但是，与其他基于C的设计流程和传统的Verilog/VHDL设计流程相比，这种设计流程有许多优点。

- **创建纯C/C++，速度快、效率高：**与等价的SystemC和增强型C/C++描述（它们比RTL描述要紧凑，代码量只相当于RTL的1/10到1%）相比，无时序特性的纯C/C++描述更加紧凑，创建和理解也更加容易。[209]
- **C/C++的验证快速高效：**无时序特性的纯C/C++描述仿真速度要比有时序特性的SystemC或增强型C/C++模型快得多，更要比RTL描述快100~10 000倍。其实，算法和系统验证的设计师已经在广泛使用纯C/C++流程了。
- **评估备选实现方案更快更高效：**修改和重新验证纯C/C++的设计，以便对多种备选的微观结构进行假设评估，这一过程很快。这就使设计团队能够快速找到最优的微观结构方案。相应地，就可以得到比传统RTL流程更小更快的设计结果。
- **可以容易地实现规范修订：**如果在工程进行过程中需要修订规范，那么在纯C/C++流程中实现和评估这些改变相当容易，因此可以很快将这些改变反映到实现结果中。

此外，如前所述，现有SystemC和增强型C/C++设计流程中最严重的问题之一是，与设计相关的引导指令部分必须被硬编码成模型，这样就使得实现是有针对性的。

纯C/C++流程的一个关键特性是，提交给综合引擎的代码可以由任何人编写，即使他或她不熟悉硬件设计或者目标器件的结构也没关系。这就是说，系统设计师写的C/C++代码就是这种综合系统的理想输入。使用纯C/C++代码进行综合时需要做的唯一改变通常是在源代码中增加一行专门的注释，来指出设计中的顶层功能模块（这个模块以上的任何部分都属于testbench的范畴）。[210]

与在源代码中增加引导指令（这样就会将设计锁定到某种目标实现上）不同的是，用户提供的所有引导指令控制和引导着综合引擎本身，如图11-7所示。

一旦综合引擎解析完源代码，用户就可以交替进行各种微观结构的验证，并且评估这些结构在速度和面积上的效果。综合引擎会对代码进行分析，并识别出各种结构和操作符，以及相应的数据和存储位置，如果可能的话，还会自动推断出并行结构。综合引擎也提供一个图形化接口，可以让用户指定各种不同元素被处理的方式。例如，用户可以通过接口将端口与寄存器或者块RAM连接起来；可以识别出循环体等结构，并允许用户指定这些结构是应该完全拆开、部分拆开，[211]

还是保持独立；用户可以指定循环体和其他结构是否进行流水线处理；用户还可以对某些实体进行资源共享；等等。

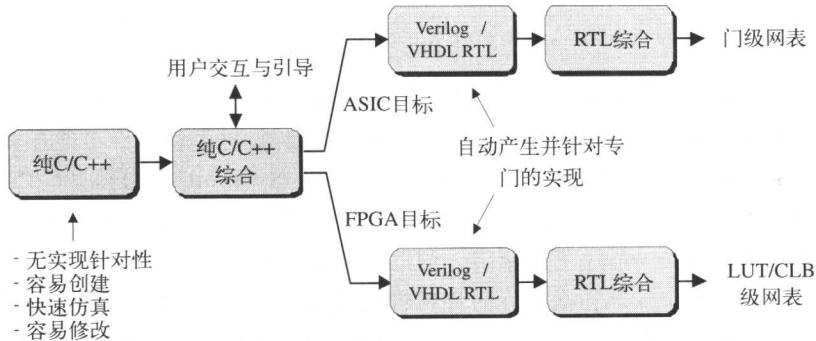


图11-7 纯C/C++流程

这些评估可以在空闲时间做，综合引擎会输出报告，说明总的规模/面积以及时钟周期的潜伏期和I/O延迟（或者在流水线设计中的有效运行周期）。每一次假设分析用的用户配置文件都可以命名并保存，方便以后重用（在传统的手工编码的RTL设计流程中，要想快速进行这些交替变换的评估几乎是不可能的）。

综合引擎所使用的无时序特性的纯C/C++源代码不必包含任何实现引导指令，所有的引导指令都是控制综合引擎本身的。这就是说，同样的源代码可以很容易地重定位到另一种结构和不同的实现技术上。

一旦用户完成评估，只要单击一下Go按钮，综合引擎就会生成相应的RTL级的VHDL代码。随后，传统的逻辑综合或者物理综合工具就使用这些代码产生网表，以便驱动下游的实现工具（如布局布线等）。

一般来说，将无时序特性的纯C/C++代码直接综合成门级网表是有可能的（这种方案在图11-7中并没有示出）。但是，生成中间形式的RTL代码更加符合工程师们的习惯，因为他们可以检查C/C++向RTL的转换是否满足要求。

212

此外，生成中间形式的RTL代码在其他方面也是有用的，因为在这个抽象级别上，硬件工程师通常可以将各种功能模块组合在一起构成他们自己的设计。今天，每个设计的大部分都是以RTL描述的IP核的形式存在。这就意味着，图11-7中所示的生成中间形式RTL的步骤在集成和验证整个硬件系统中是非常有用的。设计工程师可以完全利用现有的RTL综合技术，因为这些技术成熟、健壮，而且容易理解。

11.6 综合的不同抽象级别

本章所讨论的几种基于C/C++的设计流程，其根本差异在于每一种流程支持的

综合抽象级别不同。例如，尽管SystemC具有强大的系统级、算法级和事务级建模能力，但是它的可综合子集却是在一个相当低的抽象级别上。同样，虽然增强型C/C++要比SystemC更加接近纯C/C++（意味着仿真速度更快），但是它们的可综合子集所处的抽象级别仍然比理想情况低得多。

综合抽象级别的不足使得有时序特性的SystemC和增强型C/C++描述与具体实现有关。因此，这也使得它们很难创建和修改，严重地削弱了在假设性评估和重新定位不同实现技术上的灵活性，如图11-8所示。

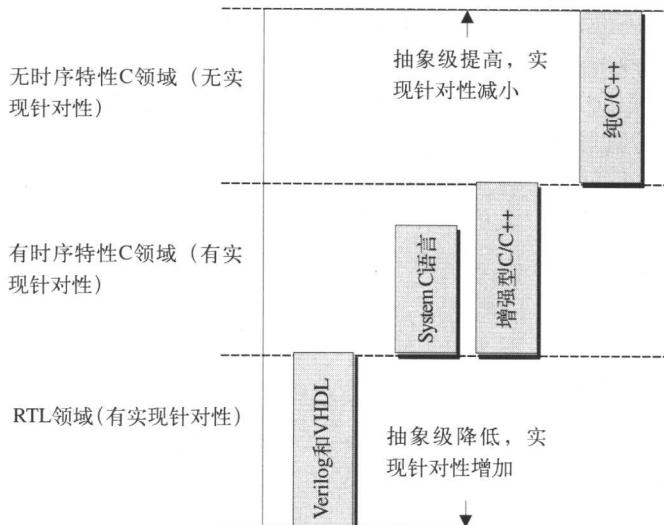


图11-8 C/C++综合的不同抽象级别

或许有人就此认为作者是反SystemC的，并打算进行批评。在此之前，应该重申一下，这里所有的讨论都是针对FPGA设计流程中各种C/C++变体的使用的。

这种情况下，从SystemC描述到实际的实现需要很多的工具，而这些工具还相当不成熟，也很简单。

但是，当谈到系统级建模和验证应用时，SystemC则是极其高效的（许多人将SystemC和SystemVerilog结合起来使用，先用SystemC进行最初的系统级设计描述，然后再用SystemVerilog进行详细的实现级设计）。

同样，对于那些具有软件设计背景，或者具有嵌入式软件和软硬件协同仿真及验证经验的设计人员来讲，很多人都认为SystemC是非常好的，好像“the bees knees”。

比较起来，最新的无时序纯C/C++综合技术则可以支持很高的综合抽象级别。无时序针对性的C/C++模型非常紧凑，而且创建和修改起来都很容易和快速。用户通过综合引擎本身可以快速而容易地进行假设性评估和重新定位到新的实现技术。结果就是，与其他C/C++设计流程相比，纯C/C++设计流程可以极大地加速实现过程，提高设计灵活性。

213

11.7 混合语言设计和验证环境

最后，我们应该注意到，许多EDA厂商都提供混合层次设计和验证环境，这可以支持多种抽象级别的模型进行协同仿真。

有些情况下，这可以简化一些过程，例如通过编程语言接口（PLI）将C/C++模型与Verilog仿真器连接起来，或者通过外部语言接口（FLI）与VHDL仿真器连接起来。另外，也可能在SystemC环境中具有接受Verilog或VHDL模块的能力。

后来就出现了非常复杂的设计环境，可以在一个图形化模块的编辑器中将设计中主要的功能单元显示出来，每一个模块的内容可以使用以下语言进行描述：

- VHDL
- Verilog
- SystemVerilog
- SystemC
- Handel-C
- Pure C/C++

214

顶层设计可能用传统的HDL编写，其中再调用各种HDL或者C/C++的各种变体编写的子模块。另外，顶层设计也可能用C/C++的一种变体编写，其中再调用其他语言编写的子模块。

在这种环境中，VHDL、Verilog和SystemVerilog描述通常由一个单核仿真引擎来处理。然后这个仿真引擎再与各种C/C++变体的仿真引擎进行联合仿真。此外，这种环境还集成了源代码调试器，支持各种C/C++变体；支持用任何语言创建testbench；支持图形化波形显示工具，可以显示任何语言描述的模块中的信号与变量^①。

还有一点可能并没有真正考虑到，当你用这里讨论的某一种C/C++变体来描述设计，那么你也常常会用同一语言来编写testbench。

^① 这种混合语言仿真与验证环境主要面向FPGA领域，在ASIC领域中较少出现，其中一个很好的例子就是Aldec公司（www.aldec.com）提供的解决方案。另一个例子是Mentor公司的ModelSim[®]，具有本地SystemC支持，因此可以使用单核进行VHDL、Verilog和SystemC的联合仿真。

这种testbench所采用的语言结构通常无法被后续的工具理解，如C/C++到RTL的翻译工具。所以，在过去就必须靠手工将testbench从C/C++描述翻译成等效的RTL描述，以便在VHDL/Verilog仿真器中使用。

混合语言设计和验证环境的优点之一是，用户可以继续使用原始的C/C++编写的testbench来驱动后续的VHDL/Verilog和门级设计（可能只需要修改很少一部分内容，但这要比完全重写所有代码强得多）。

实际上，各种混合语言设计和验证环境的解决方案几乎每周都有所变化，所以在讨论这些环境前，最好先了解一下最新的进展。

第12章 基于DSP的设计流程

12.1 DSP简介

DSP（即数字信号处理）是电子学的一个分支，主要包括对数字信号的描述和处理。这里所说的处理包括压缩、解压缩、调制、错误纠正、滤波以及其他处理方式，如对于电信、雷达和图像处理（包括医学图像）应用中的音频信号（声音、音乐等）、视频信号还有类似信号的处理。

这些情况下，对数据的处理是从模拟信号开始的。对模拟信号进行周期性的采样，然后利用模数转换器（A/D）将每个采样值转换为等值的数字信号，如图12-1所示。

模数转换器（A/D）也可称为ADC。

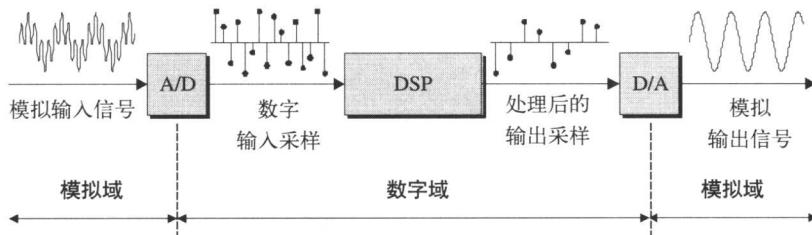


图12-1 什么是DSP

217

随后就在数字域内对这些采样值进行各种处理。之后再通过数模转换器（D/A）将处理后的数字采样信号转换为等价的模拟信号。

数模转换器（D/A）也可称为DAC。

DSP几乎出现在所有的领域内，如蜂窝电话系统，CD、DVD和MP3播放器，无线设备和医学设备，电子视觉系统，等等，新的应用还在不断增加中。这就是说，整个DSP的市场是巨大的。

DSP领域内的专业人员常常挂在嘴边的一个词汇是CODEC。

有时候这个词代表COmpressor/DECompressor，即压缩/解压缩器；也就是对数据进行压缩和解压缩的某种装置。

但是在电信领域内，这个词一般表示CODEC，即编码/解码器；也就是对信号进行编码和解码的某种装置。

CODEC可以用软件实现，也可以用硬件实现，还可以用软硬件混合实现。

12.2 可选择的DSP实现方案

12.2.1 随便选一个器件，不过不要让我看到是哪种器件

通常情况下，任何事情都不简单，因为可以用很多不同的方法来实现数字信号处理任务。

- **通用微处理器（μP）：**也可以称为中央处理器（CPU）或者微处理器（MPU）。通过在处理器中运行适当的DSP算法可以执行DSP任务。
- **数字信号处理器（DSP）：**这是一种特殊的微处理器芯片（或者内核，稍后会讨论），经过了专门的设计，执行DSP任务时要比通用微处理器更快更高效。
- **专用ASIC硬件：**考虑到本节中的讨论，我们假定专用ASIC硬件是指一种客户化的执行DSP任务的硬件实现。然而，我们也应该注意到，在ASIC中嵌入微处理器或者DSP内核之后，就可以用软件来实现DSP任务了。
- **专用FPGA硬件：**考虑到本节中的讨论，我们仍然假定专用FPGA硬件是指一种客户化的执行DSP任务的硬件实现。但是，也应该注意到，如果在FPGA中嵌入一个微处理器内核（在写作本书时，用于FPGA的专用DSP硬核还没有出现），就可以用软件来实现DSP功能了。[218]

12.2.2 系统级评估和算法验证

先不考虑最后的实现技术（μP、DSP、ASIC、FPGA），如果要开发一个基于一种新的DSP算法的产品，一般的做法是，首先利用适当的环境（在本章后面详细讨论）来进行系统级评估和算法验证。

虽然本书尽量避免集中讨论某些公司及其产品，但是如果不说也不太合适。在写作本书时，DSP算法验证中事实上的工业标准是MATLAB[®]^①，由MathWorks公司出品（www.mathworks.com）^②。

因此，为了讨论的方便，我们认为使用MATLAB是有必要的。但应该了解的是，对于DSP开发人员而言，其他功能强大的工具和环境也有很多。例如，

^① MATLAB和Simulink是MathWorks公司的注册商标。

^② 应该注意到，MATLAB和Simulink的应用范围非常广泛，包括控制系统设计与分析、图像处理、财务建模，等等。

MathWorks公司的Simulink[®]；CoWare公司^① (www.coware.com) 的SPW (Signal Processing Worksystem) 非常流行，尤其是在电信领域；Elanix公司 (www.elanix.com) 的工具也有很多工程师在使用。

12.2.3 在DSP内核中运行的软件

假定我们将要用微处理器或者DSP芯片（内核）来实现新DSP算法，那么，设计流程可能如图12-2所示。

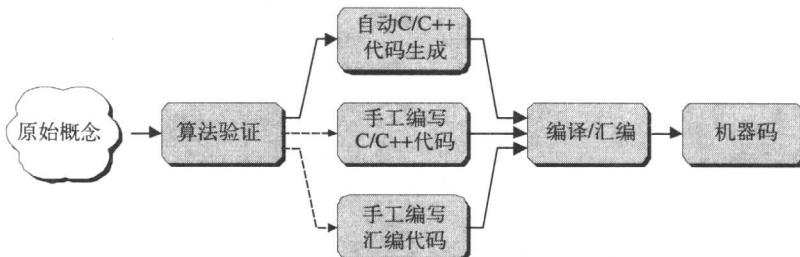


图12-2 软件DSP实现的简单设计流程

开始先有人提出一个或者一组新的算法。这一概念一般要经过上面提到的工具如MATLAB等的验证。有些情况下，也有可能直接从概念跳到编写C/C++或汇编语言代码。

算法经过验证后，必须用C/C++或者汇编语言重新生成。MATLAB可以自动为目标DSP内核生成调整后的C/C++代码，但是有时候，设计团队更喜欢手动来做这种转换，因为他们相信自己能够实现更加优化的结果。另外有一种做法，可以首先在算法验证环境中自动生成C/C++代码，通过对这些代码进行分析和概括，找出性能瓶颈，然后对代码中最关键的部分进行手工调整（这是一个很好的能说明古老的80：20规则的例子，即80%的时间用在了设计中最关键的20%上）。

一旦用C/C++或者汇编语言描述完算法，就可以将其编译或汇编为机器码，机器码最终将运行在微处理器或者DSP内核中。

这种实现方式非常灵活，如果想做什么改变，只要修改源代码并重新编译一下就可以又快又容易地实现。但是，这也会导致DSP算法以最差的性能实现，因为微处理器和DSP芯片都属于图灵机类型，即它们的主要作用是处理指令，包括以下这些操作：

- 取指令
- 指令译码

^① EDA的变化非常快。在我开始构思本书时，SPW还由Cadence公司控制，但是当我写到一半时，它却到了CoWare公司的名下！

- 取数据
- 对数据进行操作
- 存储计算结果
- :
- 取另一条指令并重复以上操作

当然，在以上所述的情况下，DSP算法实际上是运行在微处理器或者DSP内核这样的硬件上，但我们认为这是一种软件实现，因为算法的实际（物理）表现是在芯片上运行的程序。

1937年，性情古怪的英国天才阿兰·图灵还是一名大学生的时候，就写了一篇轰动世界的论文《论可计算数及其在判定问题上的应用》。由于图灵并没有接触过真正的计算机（那时候还没有计算机），所以他的发明完全是一种抽象的“论文练习”。这种理论模型称为图灵机，后来激发出许多的“思想火花”。

12.2.4 专用DSP硬件

有无数种方法可以在ASIC或者FPGA中实现DSP算法，本章集中讨论后一种情况。但是，在开始讨论以前，我们首先考虑一个问题，即各种不同的架构对最终的实现速度和面积（指硅芯片面积）有何影响。

DSP算法一般需要大量的乘法运算和加法运算。举一个简单的例子，一个新的DSP算法包含如下的表达式：

$$Y = (A * B) + (C * D) + (E * F) + (G * H);$$

如果你是非技术类读者，这里有这种解释：这个等式中的每一个变量都代表一组二进制信号。而且，两个同样宽度的二进制数相乘后，其结果的宽度将加倍（例如，A和B都是16位宽度，它们的乘积就是32位的）。

通常，这是很普通的语法，并没有与任何HDL有特别的联系，只是用在这里的讨论。当然，这个表达式只是一个极其复杂的算法中的非常微不足道的一个元素。但是，说到底，再复杂的DSP算法也都是由许多这种类型的元素组成的。

关键是，我们可以利用硬件具有的并行特性来执行DSP算法，这要比在DSP内核上运行软件快得多。例如，假设所有的乘法并行执行，然后有两级加法，如图12-3所示。

请记住，乘法器相当大而且很复杂，加法器也比较大，所以这种实现方式虽然非常快但却占用了比较大的芯片面积。

另一种方法，我们还可以用资源共享（在乘法操作之间共享一些乘法器和加法器）技术得到一个并行与串行混合的方案，如图12-4所示。

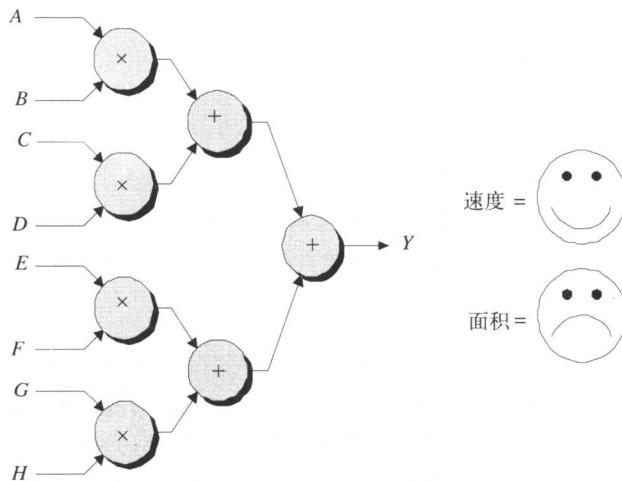


图12-3 该功能的并行实现方案

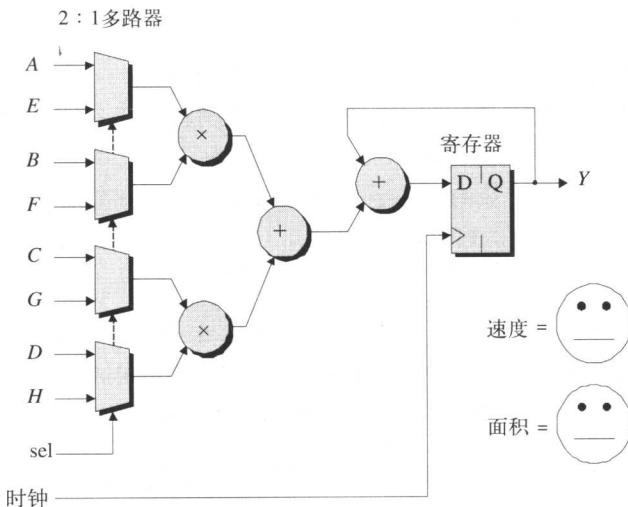


图12-4 该功能的一种折中实现方案

这种方案需要增加4个2:1多路器和1个寄存器（注意，增加的这些元件与其各自的数据路径具有相同的宽度）。但是，多路器和寄存器占用的面积要比2个乘法器和1个加法器占用的面积小得多，而与原来的方案相比，本方案已经不再需要这2个乘法器和1个加法器了。

这种实现方案的缺点是速度比较慢，因为我们必须先执行 $(A \times B)$ 和 $(C \times D)$ 这2个乘法，两个乘积相加后再与寄存器中的原值（初始值为0）相加，并将结果存入寄存器中。接下来，还要执行 $(E \times F)$ 和 $(G \times H)$ ，两个乘积相加后再与寄

存器中的原值（当前值是前一次乘法和加法的结果）相加，然后存入寄存器中。

这种交替使用不同数据路径和控制逻辑的实现方式一般称为微观结构探测（也可参考第11章中对于这一点的更详细的讨论）。

还有一种方案，可以使用完全串行的方法来实现，如图12-5所示。

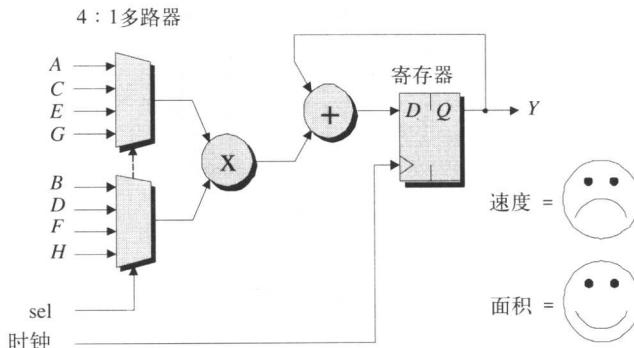


图12-5 该功能的串行实现方案

这种实现方案在面积方面非常高效，因为只需要1个乘法器和1个加法器即可。但这也是最慢的一种实现方式，因为我们先要执行 $(A \times B)$ ，结果与寄存器中的原值（初始值为0）相加后再存入寄存器中。接下来，要执行 $(C \times D)$ ，结果与寄存器中的原值相加后再存入寄存器中。剩余的乘法操作依此类推。（注意，这里所说的“最慢的一种实现方式”是对硬件而言的，但是即使最慢的硬件实现也要比在微处理器或者DSP上运行的软件实现快得多。）

223

12.2.5 与DSP相关的嵌入式FPGA资源

在第4章中已经讨论过，一些如乘法器的功能模块，如果用FPGA中的可编程逻辑块来实现，其速度根本不会快。由于大量的应用都需要这些功能模块，所以许多FPGA中都集成了专门的硬线乘法器模块（这些模块通常都被放在靠近嵌入式RAM块的位置，因为它们常常结合在一起使用）。

同样，有些FPGA还提供专用的加法器模块。在DSP类型的应用中有一种非常普遍的操作称为乘累加。正如其名字一样，这一功能完成两个数的乘法，并将乘积与累加器（寄存器）中的数据进行累加。因此，也常常被称为MAC，代表乘、加、累加（如图12-6所示）。

224

需要注意的是，图12-5所示的串行实现方式中的乘法器、加法器和寄存器就是一个经典的MAC例子。如果你使用的FPGA只提供嵌入式乘法器，那你就必须用一定量的可编程逻辑块构成加法器并与嵌入式乘法器结合，结果要存入一个块RAM中或者许多的分布式RAM中。如果FPGA也提供嵌入式加法器，事情就会变得更容易些。另外，有些FPGA提供完整的嵌入式MAC模块。

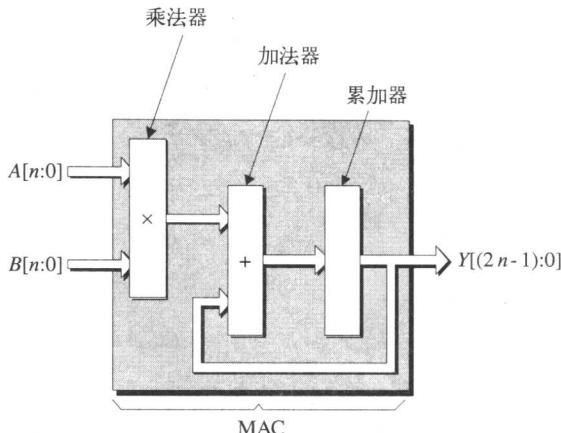


图12-6 构成MAC的功能模块

12.3 针对DSP的以FPGA为中心的设计流程

啊哈！我在写下这些话时，激动得有些颤抖（也是期待）。因为在写作本书时，
[225] 用FPGA来执行DSP运算的想法还是相当新鲜。因此，实际上并没有什么权威性的设计流程和方法，每一个人都可能有自己的独一无二的做法，而且不管你选择哪一个，几乎总会开辟出一片新天地。

12.3.1 专用领域语言

随着时间的推移，电子设计的规模和复杂性也在不断增加。在维持或者更通常地说是提高生产力的时候，为了解决这个问题，有必要提高抽象层次来描述和验证设计的功能。

由于这个原因，第8章讨论的门级原理图就被第9章中讨论的用VHDL和Verilog进行RTL描述的方法所代替。同样，第11章中介绍的基于C语言的设计流程也被一种渴望所推动，即更快地实现复杂的概念，更容易地简化结构分析和探测。

在如DSP的某些专门的领域，系统架构师和设计工程师通过使用专用领域语言（DSL）可以显著提高效率，描述某些专门的任务时，使用DSL要比用一些通用语言如C/C++和SystemC更加简单。

MATLAB就是这样一种语言，DSP设计工程师通过它仅仅使用一行代码^①就可以描述信号变换，例如FFT：

```
y = fft(x);
```

实际上，MATLAB既是一种语言，又是一个算法级的仿真环境。为了避免混淆，

^① 注意，本例中MATLAB语句后面的分号是可选的。如果有分号，它可以禁止显示输出信息。

一般说成M代码（即“MATLAB代码”）和M文件（MATLAB代码组成的文件）。也有些工程师偶尔将其称为“M语言”，但是MathWorks公司的人并不喜欢这种称呼。

MATLAB仿真的输入激励可能来自于一个或者更多的数学函数，例如正弦波发生器，也有可能是真实世界中的数据，例如音频或视频数据文件。

M文件可能包含脚本（要执行的动作）或者变换，也有可能是两者的混合。另外，M文件还可以调用其他M文件。

主（顶层）M文件一般包含了一个定义仿真运行的脚本。这个脚本可能会提示用户输入一些信息，例如滤波器系数的值、输入激励的文件名，等等，然后调用其他M文件并把这些用户定义的值传递给这些M文件。

除了一些复杂的变换，如上文中的FFT，还有些更简单的变换，如加法、减法、乘法、逻辑运算、矩阵运算等。必要的话，还可以用这些基本的运算构成像FFT这样的复杂运算。每一个变换的输出可以作为一个或者多个下游变换的输入，如此继续下去，直到在这一抽象层次上描述了整个系统。

重要的一点是，这种系统级的描述开始并没有暗示是用硬件实现还是用软件实现。例如，如果有DSP内核，可以用本章前面介绍的方法使用软件来实现整个功能。另一种选择，系统架构师还可以对设计进行划分，一部分用软件实现，而另一些性能较为关键的任务则用专门的ASIC或者FPGA等硬件来实现。这种情况下，一般需要有一个硬件与软件混合设计的环境（见第13章）。但是，为了方便讨论，我们假定完全用硬件实现。

12.3.2 系统级设计和仿真环境

系统级设计与仿真环境从概念上讲要比DSL更高级。这种环境中的一个著名的例子就是MathWorks公司的Simulink。对于一般人而言，Simulink给人的感觉只不过是MATLAB的一个图形化用户界面。但实际上，它是一个与MATLAB协同工作的独立的动态建模应用程序。

如果你正在使用Simulink，设计的开始阶段一般是创建一个图形化的方块图，其中含有一些功能模块以及它们之间的连接关系。这些模块有可能是用户定义的，也有可能来自Simulink提供的一些库（这些库包括DSP库、通信模块库以及控制功能模块库等）。使用用户定义的模块时，可以“进入”模块内部用一个新的图形化方块图描述其内容。你也可以创建一些包含MATLAB函数、M代码、C/C++代码、FORTRAN代码等功能模块。

FORTRAN语言（其名称来源于最初用途：公式转换）于1962年开发成功，是世界上最早的高级编程语言之一。

226

227

一旦你输入了设计的意图，就可以用Simulink来仿真和验证其功能了。由于使用了MATLAB，Simulink的输入激励可能来自于一个或者多个数学函数，例如正弦波产生器，也有可能是真实世界中的一些数据，例如音频或者视频文件。许多情况下，输入激励是两者的混合，例如真实的数据中可能混合了一些由Simulink模块产生的伪随机噪声。

问题的关键是并没有什么必须遵守的规则。有些DSP设计者喜欢用MATLAB作为开始，而有些则喜欢用Simulink（后者在实际设计中很少见）。有人说这一偏爱取决于用户的背景（DSP软件开发还是ASIC/FPGA设计），但是也有人说这是一堆废话。事实上这并不算是什么问题，因为，如果所言属实的话，与随后的担心相比，背后隐藏的原因显得毫无意义。

12.3.3 浮点与定点表示

不管在开始是使用Simulink还是使用MATLAB（或者使用其他厂商的类似环境），系统的初次通过模块几乎总是用浮点表示法来描述的。在十进制数字系统中，浮点数就是类似 1.235×10^3 这样的数（即一个小数乘以10的n次幂）。在MATLAB这样的应用软件中，使用IEEE标准定义的双精度浮点数来表示计算机内部的等效二进制数值。

228

浮点数的优点是，能够表示非常大的范围内极其精确的数值。但是，用FPGA或者ASIC实现浮点运算时需要非常多的资源，硬件速度也非常慢。因此，在某个阶段，设计还需要移植到定点描述，所谓定点描述是指数字的整数部分和小数部分具有固定的位数。这一过程通常称为量化。

这一转换过程完全取决于系统和算法，可能需要很多次试验才能找到一个平衡点，以便能用最少的位数表示数据（这样就可以减少对硅芯片资源的需求并提高运算速度）同时又能保持足够的精度来执行相应的任务。也有人将这种平衡看作，对于给定的数据位数，设计者可以接受多大的噪声。有时候，设计人员可能会花上几天的时间来决定“应该用14、15还是16位来表示这种特殊的数据”。另外，在系统和算法中的不同部分分别用不同的位数来表示数据可能是最好的。

229

事情开始变得有趣了，因为从浮点表示到定点表示的转换可能发生在上游的系统/算法设计和验证环境中，也有可能发生在下游的C/C++代码中。后面12.3.6节中将对此进行详细说明。其实这么说就足够了：如果在MATLAB环境中工作，只要给称为量化器的专门转换函数传递浮点信号，它们就可以执行这些转换。另外一种情况，如果用Simulink环境，可以将浮点信号通过专门的定点模块来执行转换。

12.3.4 系统/算法级向RTL的转换（手工转换）

在写作本书时，许多DSP设计团队开始先在MATLAB（或者等效的环境）中

用浮点表示法进行系统级评估和算法验证（为了进行快速的仿真和验证，通常会有一个中间步骤，即先创建一个定点C/C++模型）。随后，许多设计团队就直接开始用VHDL或者Verilog编写等价的定点RTL设计版本，如图12-7a所示。另外，他们也可能先在系统/算法级将浮点表示转换为相应的定点表示，然后再用VHDL或Verilog编写RTL代码，如图12-7b所示。

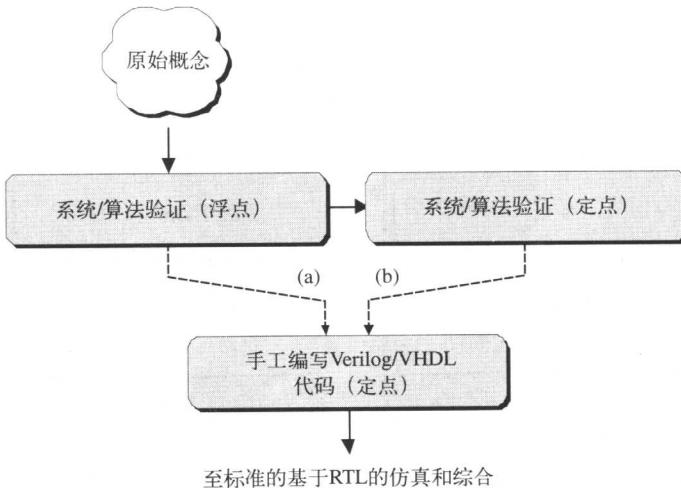


图12-7 手工生成RTL

当然，这种流程存在很多的问题，至少在系统/算法级的系统架构师和用VHDL或Verilog描述RTL的硬件设计师之间存在重大的概念性和描述方法上的分歧。230

由于系统/算法域与RTL域差异非常大，因此在两者之间的手工转换非常耗时间而且容易发生错误。还有一个事实，最后得到的RTL与实现技术有关，因为在FPGA中优化的设计所要求的编码类型与ASIC要求的不同。

另一个需要考虑的问题是，手工修改和验证RTL以便对多种可选微观结构执行一系列的假设性评估是非常耗时的 [这种评估可能包括分别以并行和串行方式执行某种操作，流水线技术与非流水线技术对比，资源共享（如两次操作共用一个乘法器）与使用专用资源的对比，等等]。

同样，如果在设计过程中原始规范有了任何的变化，在系统/算法级实现和评估这些变化相当容易，但是随后将这些变化反映到RTL中却是非常耗时间的。

当然，设计的RTL描述一旦完成，我们就可以使用第9章中介绍的基于逻辑综合的流程了。

12.3.5 系统/算法级向RTL的转换（自动生成）

前一节中讲过，手工进行系统/算法级向RTL的转换是非常耗时间而且容易发

生错误的。但是，还有许多其他的选择，因为有些系统/算法级设计环境提供了直接生成VHDL或Verilog代码的能力，如图12-8所示。

通常，系统/算法级设计开始会使用浮点表示法。在其中的一种流程中，系统/算法环境将这些浮点描述转换为定点描述，然后自动用VHDL或Verilog产生对应的RTL描述，如图12-8a所示^①。

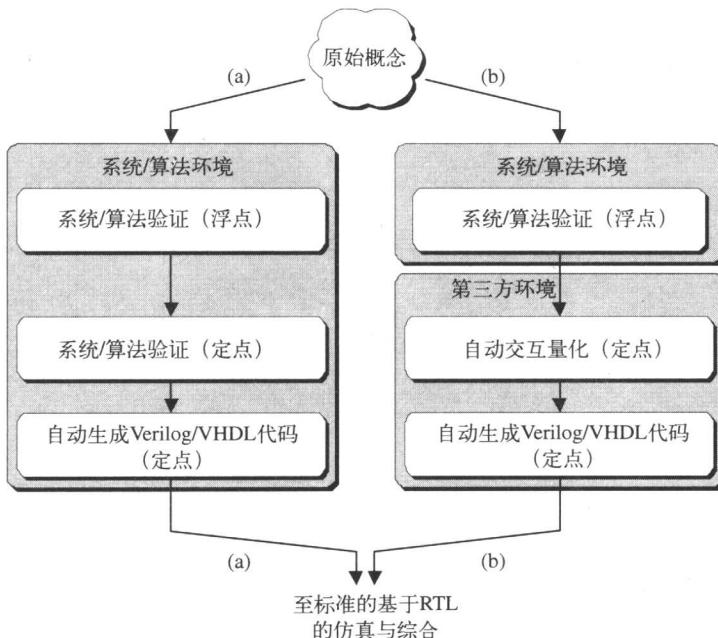


图12-8 直接生成RTL

另外，还可以用第三方的环境将浮点表示的系统/算法级描述自动交互地量化为定点形式，然后自动用VHDL或Verilog生成RTL代码，如图12-8b所示^②。

如前所述，设计的RTL描述创建完成后，我们就可以使用第9章中介绍的基于逻辑综合的流程了。

12.3.6 系统/算法级向C/C++的转换

由于在RTL级检查设计存在很多的问题，因此在中间使用一个“跳板”的方法有了渐增的趋势。这包括将系统/算法级描述转换为C/C++描述的过程，随后再转换为RTL描述。这样做了一个原因是，大多数DSP设计团队已经有一个C/C++模

^① Elanix公司 (www.elanix.com) 提供的环境就是这种类型的一个好例子。

^② AccellChip公司 (www.accelchip.com) 提供的环境就是一个很好的例子，该环境可以接受浮点 MATLAB的M文件作为输入，输出则是等效的用于验证的定点描述，然后再利用这些新的M文件自动产生RTL。

型作为黄金（参考）模型了，对于下游的RTL设计工程师而言，这种做法不会对他们产生任何影响。

要通过这些流程来验证各种备选方案的效果还是有些困难的。一般而言，可以做到如下几点：

- (1) 从MATLAB到C/C++的手工转换比较容易，需要的时间大约在数小时到几天之间（自动转换一般只用在仿真中或者根据性能要求产生DSP代码时）。
- (2) 手工检查量化效果也比较容易，尤其是对有经验的系统设计人员而言（自动交互量化用得较少）。而且，许多设计人员在这一过程中还需要噪声分析作为指导。
- (3) 从MATLAB或C/C++手工转换为RTL就比较困难了，需要的时间在几个星期到几个月。这一领域的自动化转换会给出大量的选择，从中还是有可能找到质量比较高的转换结果。
- (4) 基于MATLAB或Simulink的自动化流程依赖于IP核的产生，一般不适用于包含有实质性的原始内容的设计。

当然，首先要决定的是应该在什么时候、在设计流程中的哪个阶段将浮点描述转换为定点描述，如图12-9所示。

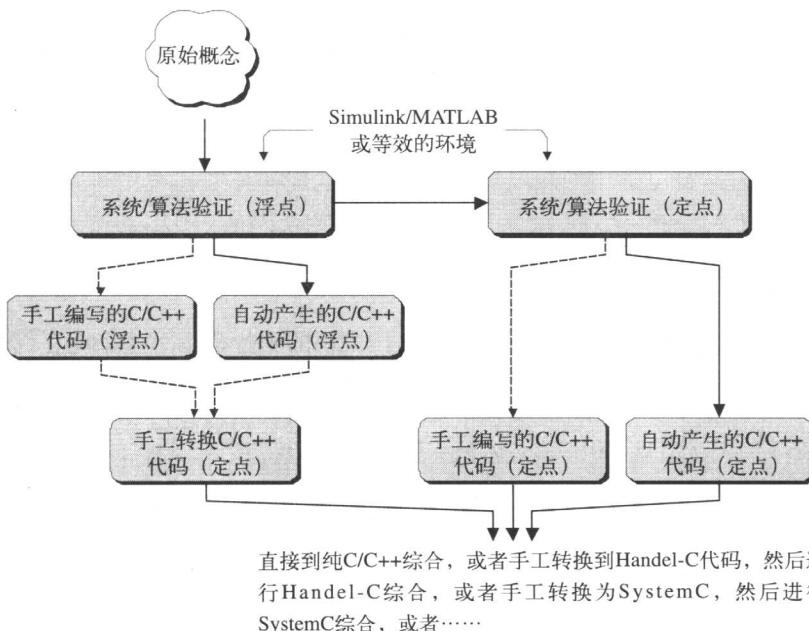


图12-9 从浮点到定点的转换

图12-9只是显示了各种可能流程的一个子集而已，这听起来也许让人有些害

怕。例如，在手工编写代码的选项中，既可以先编写C/C++代码，然后逐渐转换为Handel-C或者SystemC代码，又可以直接用Handel-C或者SystemC来编写代码。

不过，需要记住的主要问题是，一旦有了用C/C++的变种语言之一编写的定点描述后，接下来就可以用第11章中介绍的C/C++流程来工作了（这一领域内有一个特别有意思的流程，就是Mentor公司的Precision C语言使用的纯粹无时序的C/C++方法）。

12.3.7 模块级IP环境

在这个世界上，任何事情都不简单，因为总是有不止一种方法能够实现它。例如，既可以在系统/算法级创建一个DSP功能模块库，又可以用VHDL或者Verilog在RTL级创建一个一对一的等效模块库。

这里的想法是，你可以使用一个在系统/算法级指定的层次化功能模块来输入和验证设计。一旦你对设计感到满意了，然后就可以产生一个例化RTL级模块的结构化网表，并且用这个网表来驱动下游的仿真和综合工具（这些模块必须在所有抽象级上都是参数化的，以便让用户对某些参数进行调整，例如总线宽度等）。

作为一种选择，大型的FPGA厂商通常提供IP核产生器（在这里，核是指能完成某种专门逻辑功能的一个模块，而不是指微处理器或者DSP核）。在某些情况下，这些内核产生器还被集成到了系统/算法级环境中。这就是说，你可以直接在系统/算法级环境中利用这些模块搭建一个设计，指定一些与这些模块相关的参数，并进行系统/算法级的验证工作。

然后，当你准备进一步完善设计时，内核产生器将自动为每一个模块生成硬件模型^①（系统/算法级模型和内核产生器给出的硬件模型具有相同的位数和相同的周期精确性）。有些情况下，为硬件模块产生的代码是VHDL或Verilog形式的可综合RTL。另外，也有可能生成LUT/CLB级的固核，以便最大限度地利用目标FPGA器件的内部资源。

这种方法的一个比较大的缺点是，从IP模块的本性而言，它们是基于硬编码的微观结构的。这就意味着，使用IP模块为某些专门的设计创建高度调优的实现的能力被降低了。造成的结果是，基于IP的设计流程可能会以较少的风险实现一个速度更快的设计，但是与客户化硬件实现相比，这样的实现可能在面积、性能和功耗等方面优化得不够。

12.3.8 别忘了测试平台

DSP设计工具的销售人员常常忘记提到的一点是测试平台。例如，假设你的

^① Xilinx公司(www.xilinx.com)的System Generator工具与Simulink相结合就是一个很好的例子。

流程是系统/算法级设计并在随后手工转换成了RTL。这时，你还必须用测试平台再做一次同样的验证。许多情况下，这也不是一个轻松的任务，可能需要几天甚至几个星期的时间！

或者换句话说，你的流程是基于浮点表示的系统/算法级设计并手工转换成了浮点C/C++描述，此时你就希望对这种新的描述方式进行验证。然后，你可能又将浮点C/C++描述转换成了定点C/C++，这时你也希望对它进行验证。随后你可能有希望将定点C/C++描述自动综合为一个等效的RTL描述……但在这里我要送你一个礼物^①。

问题是，在每一个阶段都必须用测试平台做同样的验证工作^②，除非像下一节（也是最后一节）讨论的那样采取些技巧。

235

12.4 DSP与VHDL/Verilog混合设计环境

前一章中我们提到过，许多EDA公司可以提供混合级别的设计和验证环境，这种环境能够支持多种抽象级模型的混合仿真。例如，开始可以用一个图形化的模块编辑器来大致画出设计的主要功能单元，其中每一个模块的内容可以用以下语言描述：

- VHDL
- Verilog
- SystemVerilog
- SystemC
- Handel-C
- Pure C/C++

顶层设计可能是用传统HDL描述，其中又调用了用不同HDL或者C/C++的不同变种描述的子模块。另外，顶层设计也有可能使用C/C++的变种之一进行描述，然后又调用其他语言描述的子模块。

再后来，结合了系统/算法级和实现级的设计环境开始出现。这种环境的工作方式取决于用户是谁和用户要做什么。例如，在系统/算法级（如MATLAB）工作的系统架构师可能想用等效的VHDL或Verilog描述的RTL级模块来代替原来的一些模块。另外，工作在VHDL或Verilog的RTL级设计工程师则有可能想调用一些系统/算法级模块。

236

这两种情况下都需要系统/算法级环境和VHDL/Verilog环境的混合仿真，主要的不同是哪个调用哪个。当然，如果说得很快，这听起来也很容易，但是其中

^① 不要笑我，因为我确实知道有一个巨型的系统设计室用这种方式工作！

^② 对于从C/C++到RTL的转换，即使有C/C++到RTL的综合引擎，测试平台中通常也包含一些无法综合的语言结构，这就需要手工来做这部分工作。

有大量的问题需要解决，例如两个域之间时间概念的同步，信号在两个域之间相互传递时不同的信号类型怎样转换，等等。

实际上是这样一种情况，即以怀疑的眼光看待现有的示例。如果你正打算这么做，那就有必要坐下来同厂商的工程师一起，从头至尾调试你的例子并使其工作。如果你愿意，不妨称我为一个老的愤世嫉俗者，不过我的建议是让厂商的工程师来指导你，同时你的手不要离开键盘和鼠标^①。（有一个古老的问题：“天哪！你看见窗外刚刚飞过了什么吗？”，当你扭头去看的这短短一瞬间就会发生很多事情；你知道的话一定会非常惊奇。）

[237]

^① 不要漏掉厂商工程师的每一个动作，因此才有后面括号中的话。——译者注

第13章 基于嵌入式处理器的设计流程

13.1 引言

鉴于本书的目的，我们在这里只关心印制电路板（PCB）上有一个或者多个FPGA的电子系统。这些系统中，绝大多数使用通用微处理器（μP）来执行各种控制和数据处理应用^①。这种通用微处理器常被称为中央处理单元（CPU）或者微处理器单元（MPU）。

直到最近，CPU和一些外围设备还是以分立芯片的形式出现在电路板上。对于CPU与内存的连接，几乎有无限种可能的方式，但是这里只介绍两种主要的，如图13-1所示。

239

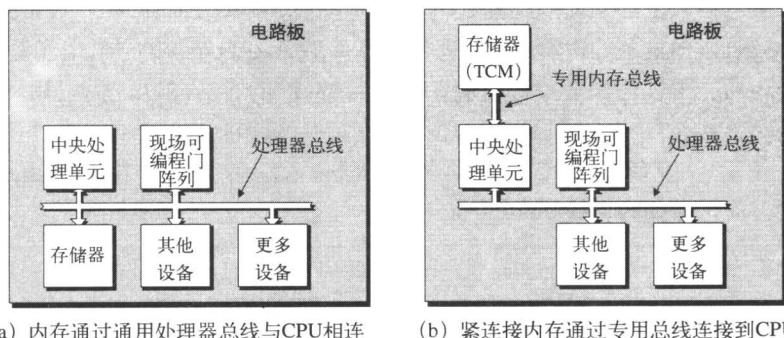


图13-1 电路板级的两种连接方式

图中所示的两种情况下，CPU都是通过通用处理器总线与FPGA和其他设备相连的（这里的其他设备是指一些外围设备，如计数定时器、中断控制器、通信设备，等等。）

有些情况下，内存（MEM）也可以通过处理器总线与CPU相连，如图13-1a所示（实际上，这种连接是通过一种专门的内存控制器实现的，图中并没有画出，以免问题变得复杂）。也可以用另一种方法，通过专用内存总线将内存与CPU连接起来，如图13-1b所示。

关键的问题是，电路板上如果分别使用专门的CPU芯片和各种外围设备芯片

^① 作为另一种选择，也可以使用微控制器（μC），这种器件中结合了CPU内核和一些外围设备，以及一些专门的输入输出。

就会增加成本，并且占用相当的面积。另外还会对电路板的可靠性产生不利影响，因为每一个焊点都有可能是一个潜在的故障来源。

240 一种替代方案是将CPU以及一部分外围设备嵌入到FPGA中，如图13-2所示。^①

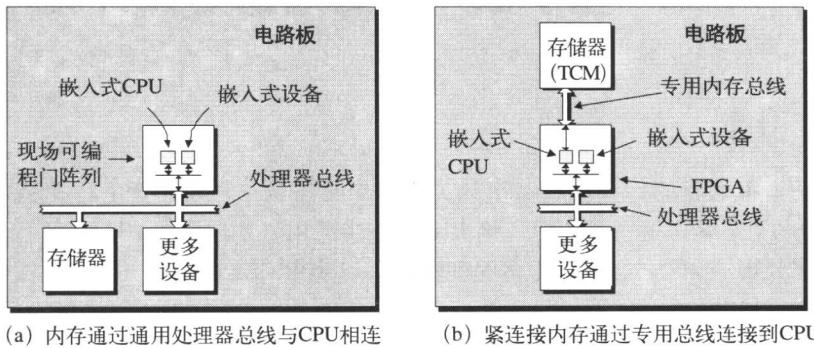


图13-2 FPGA级的两种连接方式

通常情况下，CPU使用的内存中只有很少一部分可以嵌入到FPGA内部。不过，在写作本书时，将全部CPU内存嵌入FPGA中还是很罕见的。

创建一个这种类型的FPGA设计也带来了一些全新的问题。首先，系统架构必须决定哪些功能由软件实现（即需要CPU执行的指令），哪些功能需要用硬件实现（使用FPGA结构）。其次，设计环境还必须支持混合验证，即把系统中的硬件和嵌入式软件合在一起进行验证，以保证整个系统能够正常工作。本章后面的部分将对这些问题进行详细的讨论。

除了支持微处理器内核外，每一个FPGA厂商也同时支持相关的处理器总线。例如，Altera和QuickLogic都支持ARM公司的AMBA总线（这是一个开放的规范，可以从ARM公司的网站www.arm.com上免费下载）。

比较起来，Xilinx公司的嵌入式处理器内核则使用了IBM公司的CoreConnect总线架构。

CoreConnect总线有两个变体。64位总线称为处理器局部总线（Processor Local Bus, PLB）。可以和一个或者多个32位的片上外围设备总线（On-chip peripheral bus, OPB）相连接。

13.2 硬核与软核

13.2.1 硬核

微处理器硬核是一种用专用的预定义好的硬线逻辑块实现的内核（这些内核

^① 另一种方法是在ASIC中嵌入微处理器，但是这只能在另一本书阐述了！

只在某些器件族中可用)。每一个主流的FPGA厂商都会选择一个特定的处理器类型来实现其硬核。例如, Altera公司提供嵌入式ARM处理器, QuickLogic公司则选用了基于MIPS处理器的方案, 而Xilinx公司提供PowerPC的内核。

当然, 每一家厂商都会很高兴地解释, 为什么自己的实现方式要比竞争对手的强很多(其实, 到底哪种好受很多因素影响, 因为不同的处理器适合处理不同的任务)。

在第4章中介绍过, 主要有两种方法可以将这些处理器内核集成到FPGA中。第一个是在FPGA主逻辑结构的旁边开辟一个条形区域来放置处理器内核, 如图13-3所示。

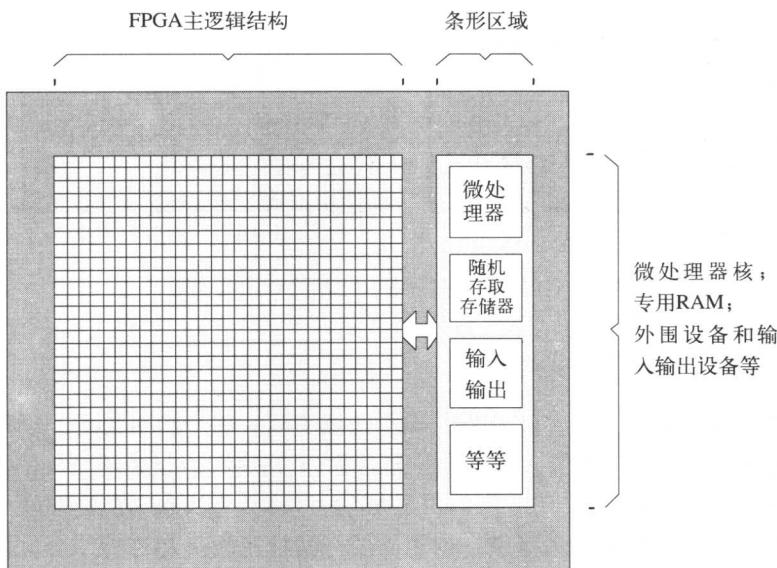


图13-3 在FPGA主逻辑结构外嵌入硬核的芯片俯视图

这种方式下, 一般是将所有元件集成在同一个硅芯片中, 不过也可以放在两个芯片中并封装成多芯片组件(MCM)。

这种实现方式的一个优点是, FPGA的主要逻辑结构在嵌入微处理器内核前后能够保持不变, 这样就使工程师使用的设计工具能够更容易地处理设计。另一个优点是, FPGA厂商可以在条形区域中增加其他附属功能以作为微处理器内核的补充, 例如存储器和一些外围设备^①。

另一种方案是直接在FPGA的主要逻辑结构中嵌入一个或多个微处理器内核。在写这本书时, 已经可以将1个、2个, 甚至4个内核嵌入到FPGA中, 如图13-4所示。

^① Altera公司 (www.altera.com) 和QuickLogic公司 (www.quicklogic.com) 喜欢用这种方式。

这种情况下，设计工具就必须考虑到主逻辑结构中这些内核模块的存在；内核使用的存储器由嵌入式块RAM实现，而外围设备则用通用可编程逻辑块构成。

242 这种方案的支持者认为，由于微处理器内核非常靠近FPGA的主逻辑结构，因此具有内在的速度优势^①。

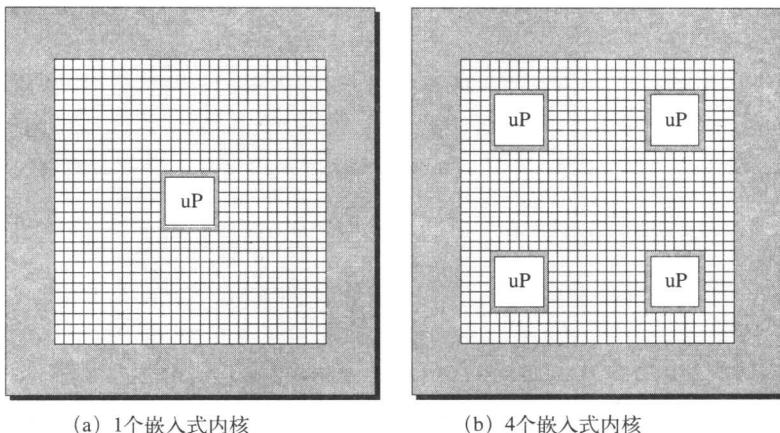


图13-4 在主逻辑结构中嵌入内核的芯片俯视图

13.2.2 微处理器软核

与在芯片内部嵌入一个实际的微处理器内核不同，将一组可编程逻辑块配置成一个微处理器也是有可能的。这种内核通常称为“软核”，但是更精确的划分应该是软核或者固核，这取决于微处理器的功能通过什么方式映射到逻辑块中。例如，如果内核以RTL网表形式提供，由于需要与其他逻辑一起进行综合，因此这实际上是一种软核。如果内核的形式是布局布线后的LUT/CLB模块，那么这通常被看作是一种固核。

在软核领域内，一个有趣的工具就是CoWare公司 (www.coware.com) 的LisaTek。你可以使用一种专门的语言来定义指令集以及与期望得到的微处理器有关的微结构（资源、流水线、时钟周期）。LisaTek根据这个定义生成相应的软核RTL代码，以及相关的软件工具，如C编译器、汇编器、连接器和指令集仿真器（ISS）。

以上这些情况下，所有的外围设备，如计数定时器、中断控制器、内存控制器、通信功能模块等也使用软核或者固核的形式来实现（FPGA厂商一般会提供一个很大的这类内核的库）。

Nios是一个基于使用寄存器窗概念的SPARC架构的软核，而MicroBlaze则是基于经典的RISC架构的软核。

^① Xilinx公司 (www.xilinx.com) 喜欢用这种方式，同时也以软IP核的形式提供大量的外围设备。

软核要比对应的硬核慢一些，但更简单（当然，在一般人看来，它们仍然快得令人无法相信）。然而，除了几乎免费外，这些软核还有其他优点——你可以只实现一个软核，也可以需要多少就实例化多少，直到用完所有的可编程逻辑资源为止。

再说明一次，主流FPGA厂商一般都选择一些特定类型的处理器作为软核。例如，Altera公司的Nios，Xilinx公司的MicroBlaze。Nios有16位和32位两种架构，分别支持16位和32位数据宽度（两种架构共享同一种16位宽度的指令集）。比较下来，MicroBlaze是一个真正的32位处理器（即指令和数据都是32位的）。还有，每一个FPGA厂商都会很高兴地告诉你它的软核为何会在市场上占主导地位，而竞争对手的软核又是怎样的不入流。

QuickLogic提供一种9位的微控制器，其名字很容易记——Q90C1xx。（9位的数据字在某些通信应用中是有用的）。

Xilinx公司提供的集成开发环境（IDE）有一个很酷的特点，它将PowerPC硬核与MicroBlaze软核同等对待。这其中包括，两种处理器都基于同样的CoreConnect处理器总线，并且可以共享外围设备的软IP核。这一切使得从一种处理器换到另一种处理器时非常容易。

还有一个有趣的事情，Xilinx公司提供一种小型的8位处理器软核，称为*PicoBlaze*，这种软核只需要使用150个逻辑单元（很少的数量）。相比而言，MicroBlaze则需要大约1000多个逻辑单元^①（对于32位的处理器实现而言，这仍是非常合理的，尤其是当使用的FPGA包含70000^②个甚至更多的逻辑单元时，这不算多）。

244

一些愤世嫉俗者说，设计中那些容易理解的部分用硬件来实现，而那些在设计开始阶段还有些不明确的部分常常交由软件来处理。（这样说的理由是，软件在设计过程中一直都可以修改。）

13.3 将设计划分为硬件和软件部分

第4章中提到过，电子设计中几乎任何部分都可以用硬件（逻辑门和寄存器等）或者软件（需要由微处理器执行的指令）来实现。划分软硬件的主要标准之一是，各种功能模块在执行任务时的快慢。

^①出于这些讨论的目的，可以假定一个逻辑单元包含一个四输入查找表（LUT）、一个寄存器和其他一些零散逻辑，如多路选择器和快速进位逻辑。

^②这里的70 000在我今天吃早饭时还是正确的，但是在你读这本书的时候，这个数值无疑已经增加了。

245

- **皮秒和纳秒级逻辑：**这种逻辑必须以极快的速度运行，这就要求用硬件实现（FPGA的主逻辑结构）。
- **微秒级逻辑：**这种逻辑以相当快的速度运行，既可以用硬件实现也可以用软件实现（这种逻辑正是需要花大量的时间来考虑怎样实现的地方）。
- **毫秒级逻辑：**这种逻辑一般用来实现一些接口，例如读取开关的数据、点亮LED等。将硬件的速度减慢以实现这种功能是一件很痛苦的事情，例如要使用很大的计数器来产生延迟。这样，用软件来实现这些功能更合适（因为与专用的硬件比较起来，处理器的速度很慢，但是实现的功能却可以非常复杂）。

划分的诀窍是，以一种最划算的方式解决每一个问题。某些功能本质上要求使用硬件，其他的则要求用软件实现，也有些功能两种方法都可以用，这取决于怎样才能最好最有效地利用资源（既包括芯片级资源，也包括硬件和软件工程师资源）。

设想有一个“理想”的电子系统级（ESL）环境，在这个环境中系统架构师先通过一个图形界面输入设计，将各种功能模块连接在一起。随后，将其中每一个模块都用系统级或算法级的SystemC来描述。这样，在决定哪些部分用硬件实现，哪些又用软件实现以前，就可以对整个设计进行验证。这完全有可能的。

当真正要划分的时候，我们可能会梦想有一种工具，只要用鼠标单击每一个图形模块并选择一个用硬件或软件实现的选项即可。随后，我们要做的所有工作就是单击一下Go按钮，上面提到的环境就会对硬件部分进行综合，对软件部分进行编译，最后将它们集成在一起。

以上只是设想，当我们回到现实世界时，会受到沉重的打击。实际上，许多下一代的设计环境都承诺，新的工具和技术几乎每天都会出现。但是，在写作本书时，系统架构师手动将设计划分为硬件和软件部分，然后将这些顶层功能模块分配给适当的工程师，并希望得到最好的结果，这种做法仍然非常普遍。

实时系统是指这样的系统，该系统中的计算或行为的正确性不仅仅依赖于它怎样执行，而且还取决于它什么时候执行。

说到设计中的软件部分，它可能简单得就像一个控制人类层次接口的状态机（读开关的状态，控制显示设备）。尽管状态机本身有可能非常复杂，但是这一层次的软件却肯定不是用于火箭科技。话又说回来了，也有可能会有非常复杂的软件需求，包括：

- 系统初始化程序和硬件抽象层（HAL）
- 硬件诊断测试组件
- 实时操作系统（RTOS）
- RTOS设备驱动

246

□ 嵌入式程序代码

这种代码一般用C/C++编写，然后编译成可以运行在微处理器内核上的机器指令（在极端的情况下，有可能对设计的性能进行优化，某些程序可能还需要在汇编代码中进行人工优化）。

与此同时，硬件设计工程师一般使用VHDL或者Verilog（或SystemVerilog）语言在RTL级输入他们负责的部分。

今天的设计通常非常复杂，硬件和软件部分必须一起进行验证。不幸的是，这种混合验证的选择太多而且又非常复杂，因此考虑这些事情可能足以使一个成年人（哈哈，实际上是我）崩溃。

13.4 硬件和软件的世界观

对设计的硬件部分和软件部分进行混合验证时要克服的最大问题之一是，这两者的创建者具有完全不同的世界观。

硬件设计人员通常会将整个设计中由他们负责的部分看作一些RTL模块，例如寄存器、逻辑功能块以及它们之间的连线。当硬件设计工程师调试设计时，他们关心的是一个能够显示RTL源代码的编辑器、一个逻辑仿真器以及一个图形化的波形显示工具，可以显示出信号在某些时刻的变化。典型的硬件设计环境中，在波形显示区域内单击某个事件就会自动定位到引起这次事件的RTL代码行。

相比而言，软件设计人员则考虑的是C/C++源代码，CPU（以及外围设备）中的寄存器，还有存储器中各个地址的内容。当他们调试一个程序时，常常希望单步运行程序并能观察各个寄存器中的值。或者他们可能希望在程序中设置一个或多个断点（也就是说在代码中某些位置放置一些标志），当程序运行到其中一个断点时就会暂停，然后他们看一看运行的情况。另外，他们也可能希望设定某些条件，例如一个寄存器包含了某个特定的值，然后程序一直运行到这个条件满足，暂停后去查看出现的情况。

当一个软件开发人员编写一个游戏时，他或她都非常相信硬件（如一台家用计算机）具有相当强的处理能力，而且没有bug。但是，当软件工程师正在编写一个要运行在某个硬件平台上的嵌入式软件，而这个硬件平台也同时处于设计过程中时，事情就完全不同了。当出现问题的时候，要确定是软件的错误还是硬件出现了故障，这是需要极高的智慧的。在软件和硬件两大阵营之间有一个经典的玩笑。

软件工程师说：“我认为，我的嵌入式软件在运行时遇到了硬件问题。”

硬件工程师说：“什么时候发生错误？能给我一个测试用例来找到问题吗？”

软件工程师说：“错误发生在今天上午9:30，测试用例就是我写的程序！”

若使用今天的艺术级的混合验证环境，硬件和软件就会紧密地结合在一起了。这就意味着，如果软件工程师发现了一个潜在的硬件bug，那么只要找出正在执行

的代码行，就可以让硬件工程师直接在波形显示界面中找到相应的仿真时间点。

[248] 同样，如果硬件工程师发现软件有一个潜在的bug（例如代码请求一次非法的硬件处理），那么他们就可以利用他们的接口引导软件团队找到对应的源代码行。不幸的是，这种验证环境非常昂贵，所以有时候你不得不选择一个更简单的解决方案。

13.5 利用FPGA作为自身的开发环境

或许，最简单的开始点就是用FPGA作为自身的开发环境。基本思想是在一块开发板上安装一个基于SRAM的内含嵌入式处理器（硬核或软核）的FPGA，然后与计算机相连。除了FPGA外，这块开发板上还要有一个存储器件，以便存储要在嵌入式CPU上运行的软件程序，如图13-5所示。



图13-5 利用FPGA作为自身的开发环境

一旦系统架构师决定了设计中哪些部分用软件实现，哪些部分用硬件实现，硬件工程师就可以开始设计各个功能模块的RTL代码了，然后综合为LUT/CLB级网表。同时，软件工程师也可以开始编写C/C++程序并编译成机器码。最后，通过配置文件将LUT/CLB级网表下载到FPGA中，与之对应的机器码映像文件则被

[249] 下载到存储器件中，然后系统就可以自由地运行了，如图13-6所示。

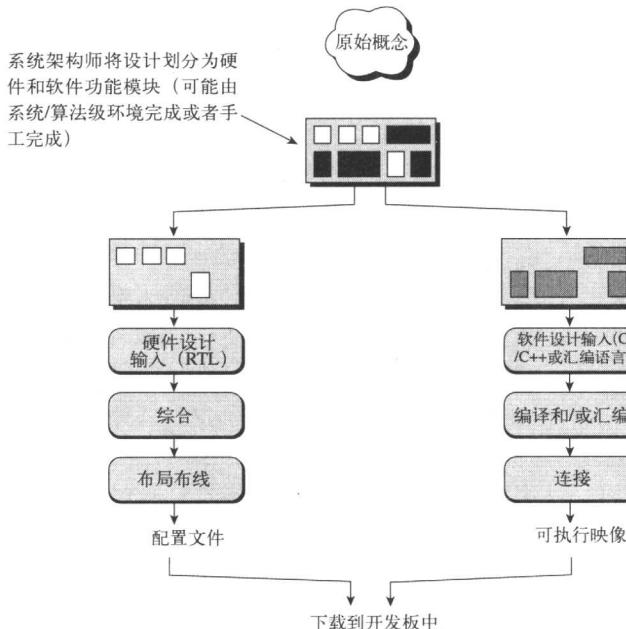


图13-6 一个非常简单的设计流程

任何将要嵌入到FPGA片内块RAM中的机器码实际上也是通过配置文件下载的。

13.6 增强设计的可见性

前一节讨论的方案中存在一个主要问题——设计的硬件部分缺乏可见性。缓解这一问题的一个方法是使用虚拟逻辑分析仪来观察硬件中的情况（这将在第16章中详细讨论）。

如果要使用软件来观察内部情况，事情可能变得稍微复杂一点。有一点需要记住，如第5章讨论的那样，嵌入式CPU内核有自己的专用JTAG边界扫描链，如图13-7所示。

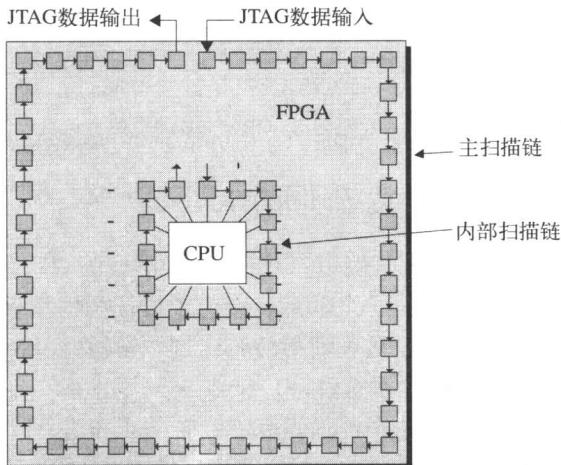


图13-7 嵌入式处理器的JTAG边界扫描链

在硬核和更复杂的软核中，这都是事实。这种情况下，混合验证环境可以使用扫描链来监控总线以及连接CPU与其他部分的控制信号上的动作。也可以通过JTAG端口来访问CPU内部的寄存器，这样就允许外部调试器来控制器件、单步执行指令、设置断点，等等。

13.7 其他一些混合验证方法

如果你真想看一看硬件内部发生了什么，有一个方法就是使用逻辑仿真器。这种情况下，系统中有一大部分都要用VHDL、Verilog或SystemVerilog在RTL级进行建模和仿真。但是，对于CPU核却有很多种方法来描述，如图13-8所示。

不管用什么样的模型来描述CPU，设计中的嵌入式软件（机器码）部分都将被下载到某种形式的存储器中，要么是FPGA中的嵌入式存储器，要么是外部存储

器件，然后CPU模型就会执行机器码指令。

注意，图13-8中只是对FPGA内部进行了较高抽象级的描述。如果机器码存储在外部器件中，那么这些外部器件也必须作为仿真的一部分。实际上，通常情况下，只要是与软件有关联的部分，都必须作为混合验证环境中的一部分。

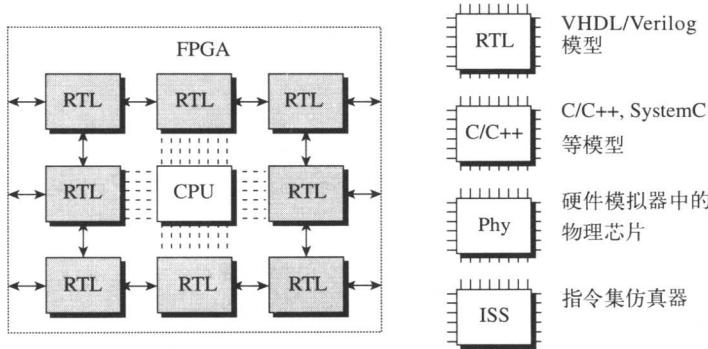


图13-8 CPU的多种描述方法

13.7.1 RTL (VHDL或Verilog)

或许最简单的选项就是使用CPU的RTL模型，此时所有的行为都发生在逻辑仿真器中。但是这种方法有一个缺点，即为了执行哪怕是最简单的任务，CPU都要进行极其多的内部操作，这就使得仿真速度非常缓慢（幸运的话，每秒钟可以仿真实际运行时的10~20个系统时钟）。

252

另一个缺点是，无法在源代码级看到软件正在进行什么操作。能做的所有事情就是观察线路和内部寄存器逻辑值的变化。

然而，事实却总是这样的：无论提供CPU的是哪个厂商，总是不希望让人们知道CPU的内部工作细节，因为这家厂商可能使用了一些非常巧妙的方法并希望保护他们的知识产权。这样一来，你就会发现，要得到一个CPU的RTL模型是非常困难的。

回顾过去，Logic Modeling公司（LMC）——后来被Synopsys公司收购——定义了一种接口，可以将硬件的行为模型与逻辑仿真器连接起来。这种接口称为SWIFT接口，任何模型（例如CPU）只要符合这个接口规范就可以叫作SWIFT模型。

13.7.2 C/C++、SystemC等

与使用RTL模型不同的是，CPU的C/C++模型被普遍使用（SystemC的支持者

有一种观点，认为CPU以及主要的外围设备都用SystemC建模，并作为这种设计环境中的标准应用)。

编译后的CPU模型可以通过一些接口连接到仿真过程中，如Verilog仿真器中的编程语言接口(PLI)，或者VHDL仿真器的外部语言接口(FLI)。

这种模型的优点是，运行速度比对应的RTL模型要快得多；另外还可以使用编译后的形式发放，因此也保护了私有的IP；还有一点，至少是在FPGA电路中，这种模型通常是免费的(FPGA厂商卖的是器件，而不是模型)。

不过这种方法也有一个缺点，即C/C++模型可能不能提供百分之百周期精确的CPU描述，如果使用过程中不小心，就有可能会产生问题。但话又说回来了，这种模型的主要缺点是，模型的唯一目的是提供一个引擎来执行机器码程序，因此你无法在源代码级看到软件正在进行什么操作。能做的所有事情就是观察线路和内部寄存器逻辑值的变化。

253

13.7.3 硬件模拟器中的物理芯片

另一种可能是使用一个物理器件来代表CPU内核。例如，如果你正在使用Xilinx FPGA中的PowerPC内核，就可以很容易地找到一个真正的PowerPC芯片来使用。将这个芯片安装在一个叫做硬件模拟器(hardware modeler)的盒子中，然后连接到逻辑仿真系统。

这种方法的优点是，物理模型(即实际的芯片)在功能上非常接近所用的硬核。其缺点是硬件模拟器并不便宜，而且很难使用。

大多数基于硬件模拟器的方案都不支持源代码级调试，因此，你无法在源代码级看到软件正在做的操作^①，只能观察线路和内部寄存器逻辑值的变化。

13.7.4 指令集仿真器

前面提到过，某些情况下，设计中软件部分的作用可能比较有限。例如，软件可能只是作为一个状态机来控制一些接口。另外，也有可能用软件对硬件的某些部分进行初始化，然后软件就空闲下来，监视着硬件的动作。如果属于这种情况，那么有一个C/C++模型或者物理模型可能就足够了，至少对硬件设计工程师来说是这样。

另一个极端是，设计的硬件部分可能主要作为与外界的一个接口。例如，硬件可能从外界读取一个数据包并将其存放在FPGA的存储器内，然后再由CPU对这些数据进行大量的非常复杂的处理。这些情况下，对于软件工程师来说，拥有强大的源代码调试能力是非常必要的。这就需要使用指令集仿真器(ISS)，这种仿

254

^①实际上，有些硬件模拟器提供一定的源代码级调试能力，例如，Simpod公司(www.simpod.com)就有一种有趣的方案。

真器提供了一个虚拟的CPU。

尽管指令集仿真器几乎肯定是用C/C++创建的，但是在结构上与前面讨论的CPU的C/C++模型还是有很大的差异。这是因为指令集仿真器是在很高的抽象层次上创建的，其中处理的事务类似于：“将内存地址x中的数据字传给我”；它不会考虑细节问题，例如信号在真实世界中的行为是怎样的。要解释指令集仿真器的工作原理，最容易的方法就是使用如图13-9所示的例子。

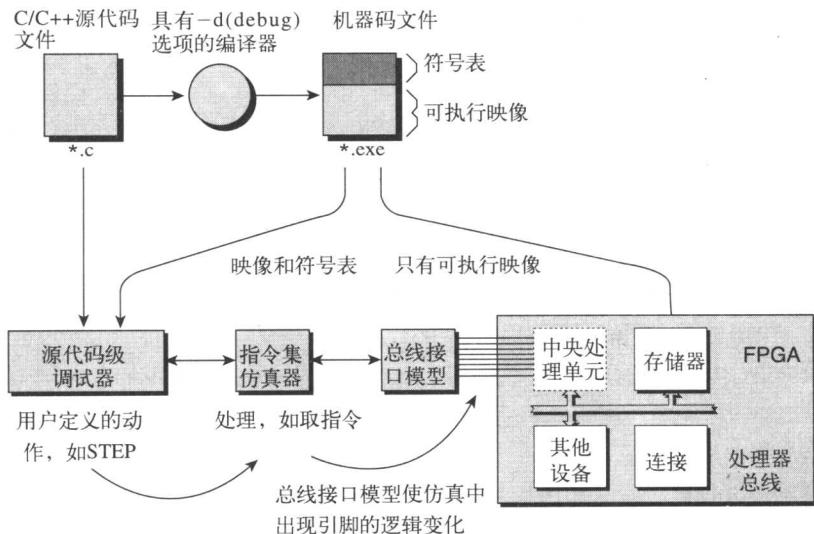


图13-9 如何将指令集仿真器用图形表示

首先，软件工程师用C/C++编写源代码，然后使用选项-d(debug)编译，产生一个符号表和其他调试专用信息，以及可执行的机器码映像。

当我们在做混合验证时，会遇到很多困难。一方面，我们要有源代码级调试器，软件工程师利用其接口与验证环境交互。另一方面，我们又要有逻辑仿真器，用来模拟存储器、外围设备、通用逻辑等（为了简单起见，此例子中假定所有程序存储器都在FPGA内部）。

然而，对于CPU而言，逻辑仿真器基本上看到的是功能模块的接口。更准确地说，仿真器实际上看到的是CPU的输入和输出。这些输入输出连接到总线接口模型（BIM）上，这个模型扮演着逻辑仿真器和指令集仿真器之间的变换桥梁。

源代码和可执行映像（以及符号表和其他调试信息）都要载入源代码级调试器中。同时，可执行映像还要载入到存储器中。当用户需要源代码级调试器执行一个动作，如单步运行一行代码，那么调试器会发出一些命令给指令集仿真器。接着，指令集仿真器将会执行一些高层次的处理，如一个取指令动作，或者存储器读写动作，或者一个I/O命令。这些处理传递给BIM后，BIM就会使适当的引脚

在仿真中发生逻辑变化。

同样，当FPGA中某些连接到处理器总线上的设备要和CPU进行交互时，将会引起驱动总线接口模型的某些引脚发生逻辑变化。然后，总线接口模型将会把这些底层动作翻译成高层次的处理，以传递给指令集仿真器，接着指令集仿真器会通知源代码级调试器所发生的事情。然后，源代码级调试器将显示出程序中相关变量、CPU寄存器的状态以及其他信息。[256]

在EDA市场上，这种非常高级（通常也极其昂贵）的验证环境有很多种^①。每一种都有自己巧妙的方法和能力，有些适合于ASIC设计，有些则适合于FPGA设计。然而，目标通常是变化的，所以在买这些工具前一定要仔细了解一下各个EDA厂商都在做什么。

13.8 一个相当巧妙的设计环境

本书尽可能（在有意义的范围内）避免讨论某些专门的公司及其产品。但是，每种规则都有例外。这里我们就要谈一个例外，因为一家名为Altium（www.altium.com）的公司提出一种很巧妙的FPGA设计环境，称为Nexar，值得我们关注。

现在已经很难搞清楚这一环境是从哪里开始的了，所以我们就干脆撇开这个问题，谈一谈那个价值7995美元^②的完整的FPGA软硬件混合设计与混合验证环境。这个环境的目标用户是那些为家用电器，如洗衣机等设计简单控制器的工程师，并且是你现在可以用大约20美元^③买到含有100万以上系统门的FPGA器件。

Nexar包含一块硬件开发板，可以插入到PC机中。这个开发板还配备了两块子板：一块含有Xilinx的FPGA，另一块含有Altera的FPGA。Nexar还具有许多微处理器软核的特征，具备了工业标准的8位器件的全部功能，如8051、Z80以及PIC微控制器（未来计划扩展到16位和32位处理器和DSP内核）。另外，还包含了一个外围设备库、一个大约有1500个元件模块的库（从简单逻辑门到计数器等复杂的器件，不一而足）和一个小型的实时操作系统（RTOS）。[257]

通过一个电路图输入界面，用户可以放置一些分别代表处理器、外围设备和其他逻辑功能的模块，并用导线将它们连接起来。Nexar提供的所有模块都是免费的。这些模块已经预综合过，所以当你有些动摇时，这些模块还可以直接下载到开发板上的FPGA中（如果有必要，你也可以创建自己的模块，并用RTL输入其内容，随后通过Nexar附带的综合工具进行处理）。

^① 例如，Mentor公司（www.mentor.com）的Seamless、Cadence公司（www.cadence.com）的Incisive，以及Axis Systems公司（www.axissystems.com）的XoC。

^② 这是2003年11月前后的价格。

^③ 这是2003年11月前后的逻辑门数和价格。

单击处理器模块，你还可以输入与该处理器相关的C/C++程序。然后可以通过Nexar附带的编译器进行处理。

基本思路是将与设计相关的所有东西（不管是硬件还是软件）都下载到开发板上的FPGA中。为了观察硬件中的情况，你可以在电路图中添加各种虚拟仪器模块，如逻辑分析仪、频率计数器、频率发生器等。对于软件，Nexar还提供了一个源代码级调试器，通过它你可以执行所有常见的任务，如设置断点、指定监视表达式、单步执行、跳出和跳入等。

我能说什么呢？我真的见过这样一个工具具有如此的魔力，给我留下了深刻的印象。我非常喜欢这种全面的解决方案，体积很小却能容纳所有东西，而且不需要昂贵的附件支持。对于它所面向的设计领域，我个人认为，Nexar在不久的将来必定会扮演一个十分重要的角色。

第14章 模块化设计和增量设计

14.1 将设计作为一个大的模块进行处理

为了使本章的讨论有一个基础，我们姑且认为FPGA由一系列的列状结构组成，每一列都包含大量的可编程逻辑块，还有一些RAM块和其他硬线元件，例如乘法器和乘累加单元（MAC）等，如图14-1所示。

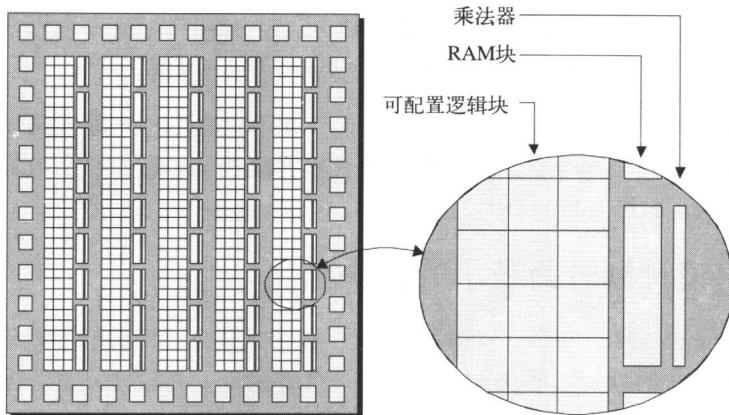


图14-1 基于列的结构

当然，这个例子是经过简化的，因为现代FPGA器件包含的列结构数量远比我们想象的要多，而且每一列又包含非常多的可编程逻辑以及其他功能模块。

我们在第5章中讨论SRAM工艺FPGA的编程问题时就曾经指出，可以非常形象地将所有SRAM配置单元看作一个很长的移位寄存器。例如，图14-2中所示的芯片表面鸟瞰图显示了I/O引脚/焊盘以及SRAM配置单元。

我们可以认为，SRAM配置单元是一系列的列状结构，每个列状结构都对应于图14-1所示的可编程逻辑中的一列。这也是一种经过简化的描述方法，因为一个FPGA器件可能包含几千万个配置单元，但是在这里这种简化的描述方法还是能够满足我们的要求。图中寄存器链的两端可以被外界访问，访问的方式由编程模式决定（这个问题与本章中的讨论无关）。

在FPGA设计的早期，即在1980年代中晚期，器件的逻辑容量非常有限。造成的一个副产品就是，一个设计工程师通常要为器件创建全部RTL代码。随后对这

些RTL代码进行综合，产生的网表交由布局布线软件处理，而这需要布局布线软件满负荷运行才能完成。

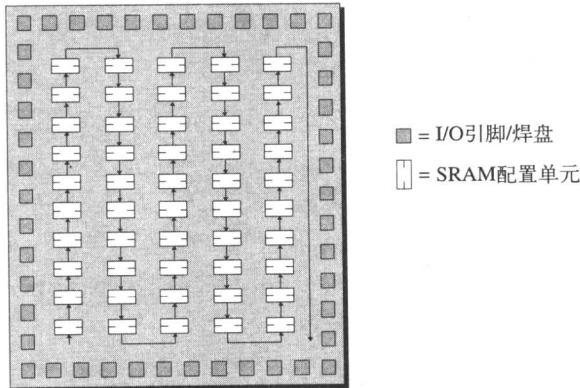


图14-2 SRAM配置单元构成一个寄存器链

结果就产生了一个单片电路的配置文件，其中定义了整个器件的功能，并被作为一个整体下载到器件中。很显然，当我们将配置单元看作一个很长的寄存器链时，这样做可以满足每一个人的需要。

14.2 将设计计划分为更小的模块

随着时间的推移，FPGA变得越来越大、越来越复杂，而设计的规模和复杂度也出现了跳跃式的增加。解决这个问题的一个办法是，将设计划分为多个功能模块，每一个模块由一名或多名设计工程师负责。

这些模块中，每一个都可以单独进行综合。但是，在最终要将设计交由布局布线软件处理前，所有模块的网表必须集成在一起。再强调一遍，布局布线软件通常需要满负荷运行才能完成这一工作，如果这是一个数百万门级的设计，那么可能需要运行一个通宵。

在2002年前后，一些FPGA厂商开始提供更大的FPGA器件，其中的SRAM配置单元构成多个比较短的寄存器链，如图14-3所示。

使用多个寄存器链构成器件的想法可能在模块化设计和增量设计等概念中已经被提出过。也有可能是在普通的硬件设计中产生的，随后会闪现一些智慧的火花：“稍等一下，现在我们有了这些寄存器链，如果我们开始支持模块化设计和增量设计会出现什么情况呢？”

如果我是一个赌徒，我可能会将钱压在后者上，不过我们干脆大方一点，假定某人在某处实际上了解了他/她正在做的事情（嘿，这肯定会发生）。但是，当这种情况出现时，这一结构及其相关的软件应用程序带来的最终结果就是，可以

支持模块化设计和增量设计。

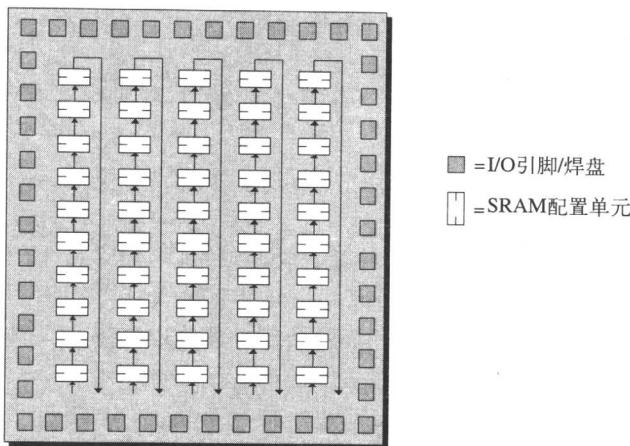


图14-3 SRAM配置单元构成多个比较短的寄存器链

14.2.1 模块化设计

有人将模块化设计称为团队设计，就是将一个大型的设计计划分为多个功能模块，并为每一个模块及其相关的时序约束指定一名设计工程师或者一个设计小组。每一个模型的RTL代码都是独立输入并综合的，而且最终的物理实现网表将交给系统集成工程师。

术语“基于模块 (block-based)” 和 “自底向上 (bottom-up)” 也可能与模块化设计有一定的关系。

最后，每一个模块（或者几个模块构成的小组）将被分配到器件中专门的区域。系统集成工程师负责将所有这些区域“缝合”在一起。这种方法在一定程度上与将一个设计分割到多个FPGA中类似，只不过这里是在同一个器件中。

这种方案的主要优点是，每一个区域的网表可以独立地交给布局布线软件（这些工具会根据给定的约束将这些网表限制在一些特定的预先定义好的区域）。这就意味着，每一位团队成员都可以完成各自的设计并能够保证在实现后完全满足时序要求，而不仅仅是在综合后满足时序要求。

14.2.2 增量设计

这个概念是指，只要你保持各模块/列块之间的接口不变，你就可以修改某个模块的RTL代码，然后重新综合这个模块，再单独对这个模块进行布局布线。这与重新运行整个设计的布局布线相比要快得多。

实际上，前一个段落所用的术语单独（isolation）”可能会令人产生一点误解。换成以下这种说法可能更合适一点，即增量设计工具“冻结（freeze）”了所有没有变化的模块，仅仅对整个设计中那些变化了的模块进行重新实现。这就为其他模块还不存在的模块化设计提供了优势（当然，模块化设计和增量设计技术也可以联合起来使用）。

14.2.3 存在的问题

这里提到的技术存在一个问题，即可能造成资源的较大浪费。因为在写作本书时FPGA中最精细的结构是整个列，所以如果某个功能模块只占用了某列75%的逻辑，那么其余25%的逻辑就不能被使用而浪费了（支持这些结构的FPGA厂商正在讨论，在将来提供一些机制来支持更精细的结构）。

这里描述的方法存在的另外一个问题是，每一个功能模块几乎肯定会产生“细长型”的实现，因为本质上你是将模块限制在一个或者多个垂直的列结构上。很明显，对于那些更适于“短粗型”实现（即跨多个列结构，而每个列中只使用很小的一部分逻辑）的功能模块来说，这是很痛苦的。

或许，早期的设计工具和设计流程利用这些结构支持模块化设计和增量设计
[263] 时存在的最重要的问题是，有人（例如系统集成人员）必须手工绘制一份布局图。这种蹩脚的方法还要求定义和放置一些专门的接口模块，称为总线宏单元，用来将总线和各模块间的独立信号连接起来，如图14-4所示。

1919年，人们开始使用拨号方式打电话。

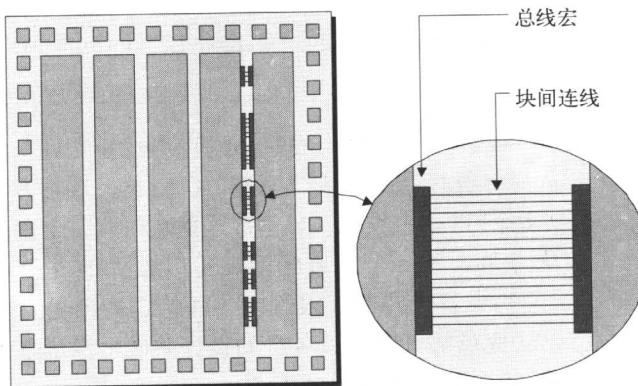


图14-4 放置总线宏单元

这些工具的最初版本使得创建布局图、定义和放置总线宏变得很困难。大街上总有这样的传言，即软件在近期的更新将会极大地简化这一过程（其实，真正的意思是不会让这一过程变得更难）。

14.3 总有其他办法

回顾第10章，我们介绍了FPGA为中心的硅虚拟原型，即SVP。那时我们提到，有些EDA厂商已经开始通过将布局图和布局布线前的时序分析结合起来的方法为用户提供能够支持FPGA SVP概念的工具。这与前面提到的对单个设计模块进行布局布线的能力结合起来，就可以极大地加快实现的过程^①。264

关键是，如果你回过头来再读一遍第10章，你就会发现那里描述的FPGA SVP的实现完全支持模块化设计和增量设计的概念，而且没有任何与本章所提到的技术相关的问题。唯一的问题是，由于更加复杂，EDA厂商提供的工具肯定要比FPGA厂商的工具昂贵得多。总之，还是那句老话：“钱在你手里，怎么花由你自己决定”。265

^① 写作本书时，FPGA SVP（本书描写的形式）的主要支持者之一是Hier Design公司www.hierdesign.com)。

第 15 章 高速设计与其他PCB设计注意事项

15.1 开始之前

如果你非要在本章中寻找含有吉比特串行I/O收发器的FPGA的有关信息，那你就找错地方了，这些内容将在第21章中介绍。

15.2 我们都很年轻，因此

十多年前（从1990年开始），对于FPGA设计工程师来说，生活在许多方面都是很简单的。在那些平静的日子里，没有人对那些蹩脚的PCB布局工程师有太多的想法。

这里我们来看看事情发展的经过。首先，在当时，即使最高端的FPGA也仅仅只有大约200个引脚（pin），用今天的标准来衡量，这相对少一些。如果采用PGA^①封装，则引脚间距（pin pitch）大约1/10英寸，即2.54mm；同今天的标准比，显然是非常大的。其次，通过类似FPGA的器件的信号延迟也比通过电路板走线的信号延迟要大得多。所有这些因素导致了一个相当简单的设计流程。

大约在20世纪90年代，以PGA形式封装的FPGA主要应用在军事领域，商业应用主要采用塑料四角扁平封装，即PQFP（plastic quad flat package），引脚从器件的四周引出。

设计过程总是从系统架构师手工绘制电路板的粗略平面图开始，这一工作通常在白板或者一小片纸上完成。实际上，对于现在我们所讨论的内容而言，“平面图”这个术语可能太大了，草图更适合表达电路各主要元件及其元件间的主要连接关系（如图15-1所示）。

以这一平面图为基础，系统架构师会挥动双手，再做出全局性的有依据的推测，最终得到了假想中的FPGA器件的输入输出定时约束。

有了这些定时约束和FPGA的功能规范之后，设计工程师（请记住，每个器件一般只有一个工程师负责）就开始埋头实现他们的计划。

一般，设计工程师很少担心FPGA的引脚配置问题，很大程度上，他们会让布

^① PGA读作“P-G-A”。PGA封装是指将引脚（pin）按阵列形式放置在器件的底面。电路板上则制作与之相对应的通孔（via）。安装器件时，只要将每个引脚插入电路板上对应的通孔就可以了。

局布线软件毫无约束地运行，无论软件最终给出什么样的引脚配置结果，他们都会接受。

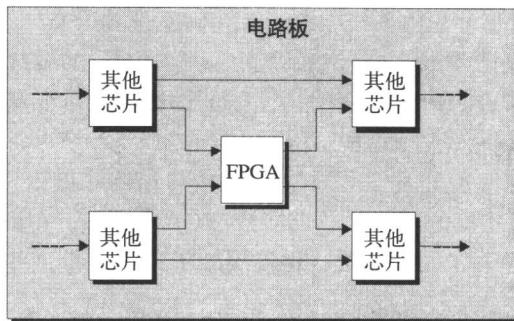


图15-1 系统架构师绘制的粗略平面图

FPGA设计（包括引脚配置）一旦完成，为了在电路原理图中能使用它，就要为其创建一个图形符号，同时为了在电路板布局中使用这个设计，还必须创建一个器件物理丝印（footprint）的图形表示。这些符号包含有关信号的详细信息，例如信号名以及与之对应的物理引脚（在布局表示中，还有引脚的物理位置信息）。

与此同时，电路板设计人员也已经对除FPGA之外的其他器件进行了布局，并尽最大可能进行布线。只有在完成FPGA设计并创建了相关的符号后，FPGA才能完全集成到电路板环境中，布线才能完成。这就意味着，到设计的最后阶段，所有工作都将集中在电路板设计师身上。[268]

但是，当我们说FPGA设计已经完成的时候，其实我们只不过是希望这个设计接近结束。而实际上，一个几乎不变的事实是，电路板设计师刚刚完成最后的布线，FPGA工程师往往就会想起一个必须修改的问题。要完成这次修改，常常最终要调整引脚配置，这多少会让电路板设计师感到不快（其实，我们都知道，接下来会使用比“不快”更加强烈的词语）。

15.3 变革的时代

看起来可能令人忧虑，几乎在整个20世纪90年代都流行上面讨论的简单设计流程，但今天的FPGA，无论是规模还是复杂程度，都决定了这种简单流程无法继续存在下去。

在本书写作期间，我们所讨论的高端FPGA已经包含多达1700个引脚，以BGA（ball grid array）^①形式封装，引脚间距只有1mm。此外，今天的IC（包含

^① BGA读作“B-G-A”。BGA封装是指将焊垫（pad）按阵列形式放置在器件的底面。电路板上则制作与之相对应的焊垫。FPGA底面的每个焊垫上都有一个很小的锡球。安装器件时，只要将其放在PCB的对应位置上并将底面的小锡球熔化以便与PCB上的焊垫形成良好的连接就可以了。

FPGA) 与其先辈比较起来就像闪电一样快, 这就使电路板走线的信号延迟变得更加突出。

归根结底, 以前那种先由系统架构师相当随意地为FPGA设定定时约束, 然后将问题交给电路板设计师来解决, 直到电路板能够工作的做法已经不再被人接受了, 这样做根本行不通。取而代之的是, 整个设计过程需要从板级开始, 其中的FPGA应该看作一个黑盒子 (如图15-2所示)。

这种情况下, 电路板布局工程师在PCB预布局的基础上进行板级定时分析, 得到的信息用来计算实际的约束, 以反馈给FPGA设计师。在现代设计中, 这种定时约束会有成百上千个, 如果不做这种板级分析, 要产生这些约束并区分它们的优先级是不可能的。

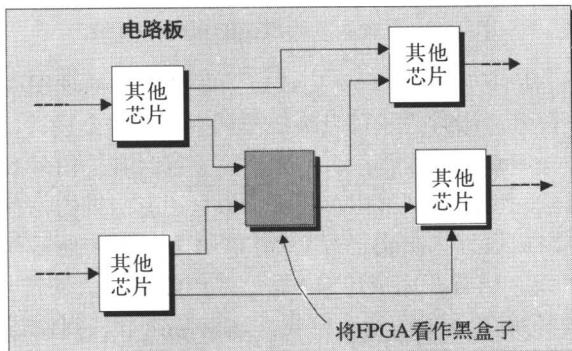


图15-2 电路板工程师进行PCB预布局

1919年, Walter Schottky发明了四极管 (tetrode), 这是第一个多栅格真空管。

但是别急, 更进一步可以看到, 为了能够对FPGA成功布线, 现在板级工程师一开始就要将信号分配到FPGA的I/O引脚上。板级工程师可以利用一些新的工具做到这一点, 这些工具提供了代表器件物理覆盖区的图形符号以及一个交互式接口, 这个接口允许用户声明信号名并将其与器件的引脚相连^①。

这些工具也可以自动产生原理图符号。如果器件的引脚超过1000个, 这些工具会将符号分成几个部分。按钮选项可以在FPGA的I/O组基础上产生这些分块, 但是也可以由用户按引脚来定义分块。

一旦电路板工程师完成这些前端工作, 就有必要将引脚配置信息通过某种机制传递给FPGA设计工程师, 再由他们将这些信息作为物理约束来引导FPGA布局布线应用软件的运行。在实际应用中, 如果FPGA工程师发现有必要修改原始引脚

^① 写作本书时, Mentor公司 (www.mentor.com) 的BoardLink Pro应用软件就是一个很好的当前工艺状况的例子。

配置，那么前述的过程可能会多次反复，但与本章开始部分介绍的流程相比，这种反复是相当少的。

FPGA Xchange

直到最近，板级工程师和FPGA工程师之间的数据交换仍然会涉及大量的手工处理过程，幸好这种情况即将发生变化，因为Mentor公司正在联合Altera、Xilinx和其他主要FPGA厂商一起定义一种新的ASCII文件格式，称为FPGA Xchange。

使用这种格式的文件，电路板工具软件和FPGA工具软件能够共享器件的某些公共定义，例如信号名是怎样分配给器件的物理引脚的。这就使得板级工程师和FPGA工程师能够快速方便地交换数据。

例如，板级工程师可以创建原始的引脚配置信息，并使用FPGA Xchange文件将其传给FPGA工程师，作为布局布线工具软件的约束，然后板级工程师可以继续PCB的布局工作。

同时，FPGA工程师可能会发现有必要修改一些引脚配置，那么这些改变将被合并到原始的FPGA Xchange文件中，随后板级Layout软件使用这个文件分离出与有变化的引脚相关联的所有走线，并对这些走线自动重新布线或者进行交互式的重新布线。

271

15.4 其他注意事项

15.4.1 高速设计

一个普遍的错误观念是，高速设计（high-speed design）意味着有一个快速的系统时钟。实际上，所谓的高速是和信号的边缘速度有关系的，这里的边缘速度是指信号从逻辑0转换到逻辑1的速度，或者做相反转换的速度，边缘转换得越快，信号完整性（SI）问题（例如噪声，串扰等）就越严重，现在可以肯定的是，随着系统时钟频率的提高，边缘速度也不得不提高，如果信号的边缘速度很高，即使系统时钟频率只有1MHz，设计人员也有可能会遇到高速设计问题。（现在的设计中，绝大多数信号的边缘速度都很快。）

15.4.2 信号完整性分析

FPGA的好处之一是，厂商已经在其芯片内部解决了绝大多数的信号完整性（SI）问题；然而在板级做信号完整性分析正在变得越来越重要。最好的工具都很昂贵，使用它们可以保证生产出来的电路板能够工作。因此，是进行信号完整性分析，还是随便处理一下看看会发生什么，你必须在两者中做出选择。

15.4.3 SPICE与IBIS

使用SPICE^①模型做板级信号完整性分析可能会消耗时间。20世纪90年代初期，Intel公司创建并提出了IBIS（input/output buffer information specification），即输入/输出缓冲信息规范，这是一种模型格式，用来描述驱动方和接收方的模拟特征。

Intel公司提出这种模型的原因，是不想将详细的SPICE模型提供给客户，因为这些SPICE模型是晶体管—电容—电阻级的，含有大量的不想让竞争对手知道的元件信息。

[272] IBIS模型从本质上讲是一种行为级模型，任何有关制程的信息都被隐藏起来了。然而，这些模型只在达到很高的频率时才是精确的，这个频率范围从500MHz~1GHz，取决于所谈论的对象。此外则必须使用更精确的模型，如SPICE模型。

另一个问题是，为了包容新的技术，语言必须具有可扩展性。例如，IBIS就没有对预加重效应（见第21章）建模的机制。然而，IBIS的语法本质上是不可扩展的，而要通过各种开放式论坛委员会对语言进行增强将是一个冗长的过程。（等这些增强有了一些成果时，你已经完成了一些工作，又有了新的问题需要考虑。）

2002年后期，一个增强的IBIS标准提议被提出，称为BIRD75 [这里的BIRD是指缓冲信息解决文档（Buffer Information Resolution Document）]。这一提议允许在端口定义（或者引脚连接层次）的基础上对外部模块进行调用。如果这一提议被采纳，IBIS就会成为可扩展的标准，因为外部模块可能是由多种不同的语言（例如SPICE、VHDL-AMS、Verilog-A等）描述的。

15.4.4 起动功率

有些FPGA有较高的瞬态起动电流，因而对电源的要求较为苛刻。板级工程师需要和FPGA设计团队一起来决定对电源的要求，以保证电路板能提供足够的电源，避免产生问题。

15.4.5 使用内部末端阻抗

几乎所有现代高速I/O标准都要求电路板上的走线有规定的阻抗和与之相连的末端电阻，这些电阻的阻值必须符合要求以消除反射和振铃效应，否则将破坏信号完整性（SI）并影响系统性能。

[273] 在器件外部使用末端电阻可能需要电路板中有额外的层，而这会增加成本，延长开发时间。如果FPGA有成百上千个引脚，那么要在离器件较近的合理距离内

① SPICE是Simulation Program with Integrated Circuit Emphasis的缩写，这个仿真程序由加州大学伯克利分校开发，从20世纪70年代初期开始获得广泛应用。

(到引脚的距离大于1cm时可能会有问题) 放置这些末端电阻几乎是不可能的。由于这些原因,一些FPGA包含了数控阻抗 (Digitally Controlled Impedance, DCI) 的功能。

数控阻抗在输入和输出端都可以使用,另外还可以配置成并行或串行方式。这些片内电阻的阻值完全由用户定义,这种技术的数字化实现意味着阻值不会随着温度或电源电压的改变而改变。

一个简单的标准是,对于上升/下降时间不大于500ps的信号,使用外部末端电阻将引起信号的不连续,这时应该使用片内的末端电阻。

15.4.6 串行或并行处理数据

电子系统中普遍将多个位 (bit) 分组并行处理,这个组称为字 (word),字的宽度与系统有关,例如8位微处理器或微控制器的字宽就是8位。

以前,IC的封装中每增加一个引脚,都会给制造厂商带来很多麻烦,因此那时的芯片中都包含了UART (Universal Asynchronous Receiver Transmitter),即通用异步收发器。这里假定字宽为8位,如果芯片要给外部发送信息,那么UART会将内部总线上的8位数据转换为一系列脉冲用来发送。与此类似,若芯片要接收外部信息,UART会接收一系列脉冲,并将它们转换为一个8位的数据,然后再把这个数据放在内部总线上。这样,使用这一技术的芯片只需要两个引脚就可以读写数据:一个是TXD,用来发送数据;另一个是RXD,用来接收数据。

随着封装技术的进步,增加引脚数量不再成为负担,因而,将整个字一起传送就越来越普遍。在8位电子系统中,需要电路板上有8条走线,每个芯片要有8个引脚与总线相连。274

随着时间的推移,越来越有必要让系统处理更多的信息并提高处理速度,这样,总线宽度就增加到16位、32位、64位等,同时时钟频率也从几个MHz增加到数十MHz,数百MHz,甚至上千MHz、这里的1000MHz就是1GHz。

随着系统时钟频率的增加,在电路板上对宽的总线进行布线并且希望信号能按时到达预期的地点而不遇到任何信号完整性问题(如噪声、串扰)时,面对的困难和问题也越来越多^①,所以,对于最高带宽的应用,设计人员正在转回到串行数据收发上,但所用的工具是Gbit串行收发器。这些内容在第21章有更详细的介绍。275

^①当然这是一个很长的句子,不过在这里是合适的,因为这个问题出现的时间已经很长了。

第16章 观察FPGA的内部节点

16.1 缺乏可见性

调试芯片时，无论是ASIC还是FPGA，都会遇到一个问题，那就是发生在芯片内部的情况缺乏可见性。为了讨论这些问题，这里假设有一个很简单的流水线设计，包含少量的寄存器和逻辑门，如图16-1所示。

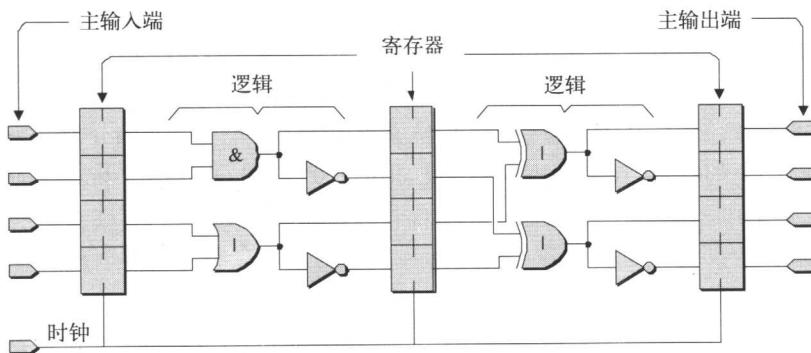


图16-1 一个非常简单的流水线电路

很明显，这是一个毫无用处的电路，不过却能满足我们现在的要求。

问题是，我们只能访问芯片的主输入端和主输出端，也就是芯片的外部端口，因此芯片内部所发生的一切我们都无从得知。如果设计已经完成并经过了验证，
277 这个问题并非特别严重，然而，如果是在芯片的调试阶段，要测定芯片为什么没有实现预期的功能，就成了一件令人痛苦的事情。

一个容易理解的解决方法是，将内部节点连接到器件的主输出引脚上，使其可见，如图16-2所示。

这种方法的不足之处在于，大多数设计都是“I/O有限”的，也就是说，瓶颈是器件封装的可用主I/O引脚数量有限。实际上，即使不使用任何I/O引脚来访问内部节点，许多FPGA设计也会有许多内部资源无法使用，因为没有足够可用的I/O引脚，这些引脚可将必需的控制信号和数据信号从器件中引出或放入器件。

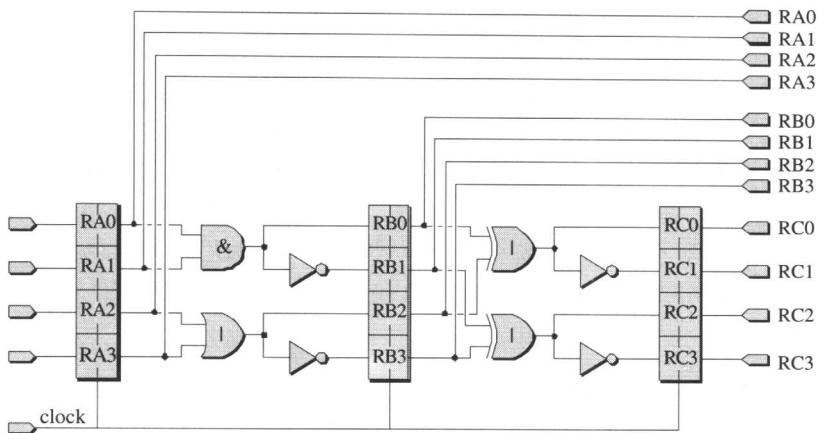


图16-2 将内部节点连接到主输出端

16.2 使用多路复用技术

另一种简单的方法是，将主要的输出和内部信号进行复用，也就是使用同一组输出引脚输出这两组信号，如图16-3所示。

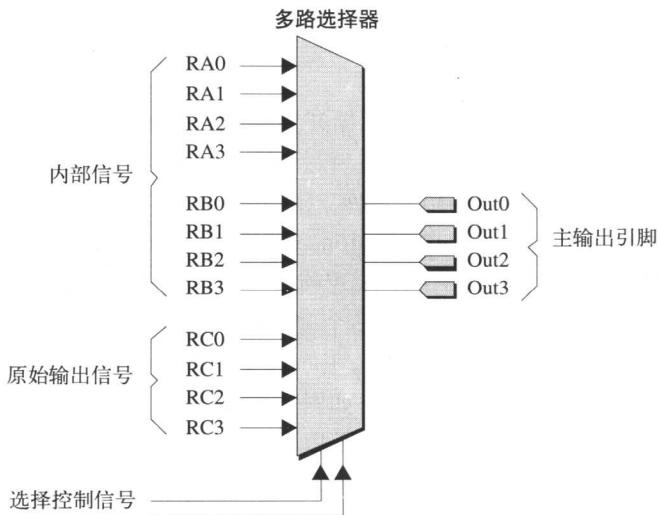


图16-3 多路信号进行复用

当然，为了复用而必须使用的选择控制信号还需要使用一些主I/O引脚。图16-3所示的例子中，最简单的情况是将两个选择控制信号直接连接到外部，因此需要

能控制多路器选择一组不同的信号，也可以参看本章后面有关虚拟线路的讨论。

这种方法的主要优点是，具有良好的可见性而且速度相对较快；主要的缺点是不够灵活，而且实现起来消耗时间，因为如果想更改正被监控的内部引脚，就必须修改设计的源代码然后重新进行综合，同样，如果想改变状态机使用的触发条件以便弄清楚某个特定的时刻多路器选择了哪一组信号，必须再一次修改设计的源代码。

还有一点需要考虑，一旦整个设计调试完毕，如果将那些测试结构从源代码中删除，有可能给设计带来新的问题，这些问题可能不止是改变了布线，还有相应的延时。
[279]

16.3 专用调试电路

有些FPGA含有专用的调试电路，可以方便用户观察内部信号节点，例如，Actel公司的FPGA就有两个专门的引脚PRA（Probe A）和PRB（Probe B）。通过内嵌的调试电路，再结合专门的调试工具软件^①，任何内部信号都可以连接到这两个引脚的任意一个上，这样就可以对该节点进行观察和分析。

这种方案最大的优点就是不必修改源代码，缺点是只有两个探测引脚可供使用，如果设计中有数百个甚至上千个内部信号需要分析，它们就有点不够了。

16.4 虚拟逻辑分析仪

尽管以上所讨论的方案都有用，但开发人员往往更愿意使用逻辑分析仪来跟踪和调试一组内嵌的信号，还可以使用逻辑分析仪将待分析信号和其他相关信号一起进行分析，或者设定一些触发条件。

片内仪器（On-Chip Instrumentation, OCI）是一种方便逻辑调试的分析方法，可以让用户将诊断IP模块。（例如虚拟逻辑分析仪）嵌入到设计中。这一想法是利用FPGA中的一部分资源实现一个或多个虚拟逻辑分析仪以便捕获被选信号的活动状态。所得到的数据将被存储在FPGA中的嵌入式RAM块中，外部逻辑分析仪软件可以通过JTAG端口访问这些数据，如图16-4所示。

[280] 虚拟逻辑分析仪中有一部分将用于检测特定信号的触发条件，通过这些条件可以启动和停止被监控信号的数据捕获过程。

是否需要修改设计源代码以包含这种功能，取决于使用什么样的虚拟逻辑分析仪。这种方案最大的优点是，即使必须在源代码中包含某种特定的宏单元，也可以比较容易地在设计中实现非常复杂的调试功能。

^① 以前的名称为Actionprobe[®]，新的增强版本称为Silicon Explorer，本书写作时，已经发展成为Silicon Explorer II，以后还会继续发展。

有些情况下，FPGA厂商将会提供这种功能，这类调试工具中比较优秀的是Xilinx公司（www.xilinx.com）的Chipscope™ Pro和Altera（www.altera.com）公司的SignalTap® II。如果所用的FPGA器件没有这种功能，还可以使用第三方的工具，如First Silicon Solutions公司（www.fs2.com），此公司专门研究片内仪器（OCI）和针对FPGA逻辑及嵌入式处理器的调试方法。

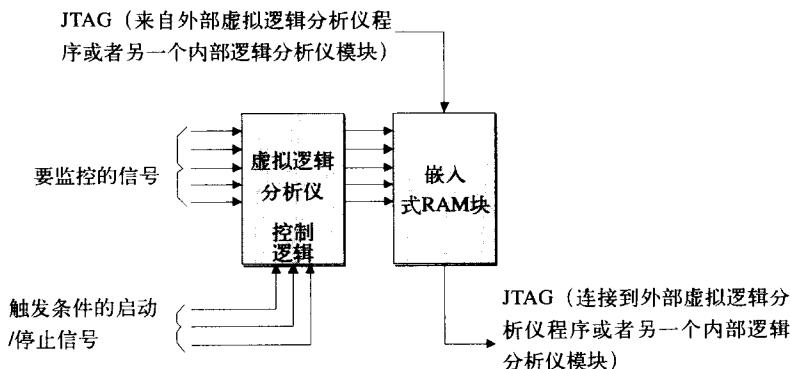


图16-4 虚拟逻辑分析仪

关于对FPGA内部信号进行跟踪、分析和调试时所用的虚拟逻辑分析仪，要提到First Silicon Solutions公司的可配置逻辑分析仪模块（Configurable Logic Analyzer Module, CLAM），其中包含一个OCI块（有Verilog和VHDL两种版本），可以重新配置并综合到设计中，这个OCI块（如果使用多个，可称为块组）可以和主机上的控制、检测和显示软件建立联系。[281]

16.5 虚拟线路

大约在20世纪90年代早期，一家名为Virtual Machine Works的公司提出了一种技术，他们称之为VirtualWires™，即虚拟线路。最初，虚拟线路是为了实现大规模的多FPGA系统，为各种基于FPGA的仿真系统提供实现的基础。这里提到虚拟线路的一个原因是它与本章前面所提到的多路技术有些相似，另外，这项技术本身确实是一个很棒的想法。

16.5.1 问题描述

虚拟线路技术的出发点是一个规模巨大的设计根本无法在单个FPGA芯片中实现，需要分解到多个器件中。举一个简单的例子，假定一个设计等效于一定数量的系统门，可用的最大的FPGA芯片只能提供一半的系统门，因此，潜意识里的想法几乎肯定就是将设计分解到两个器件中，如图16-5所示。

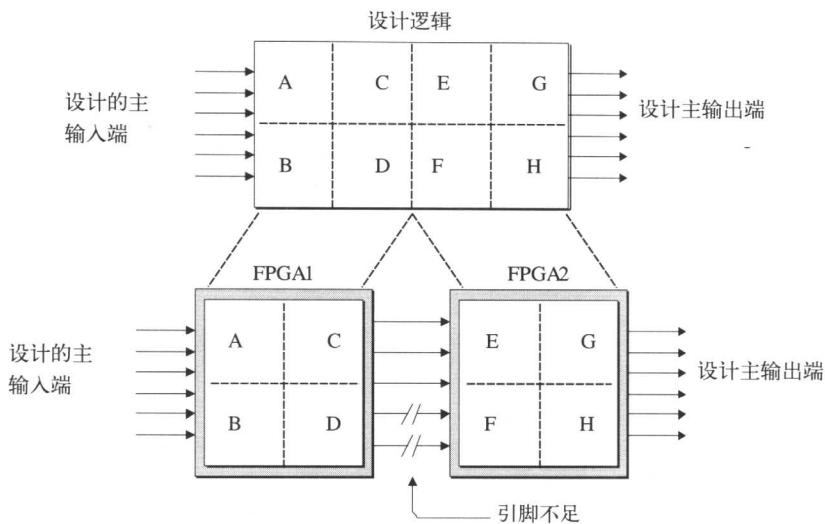


图16-5 将设计分解到两个器件中时，引脚不足

需要注意的是，图中所示的逻辑由A到H共8个子块组成，这样做只是为了更清楚地说明逻辑被分解到多个器件的过程。

问题是，芯片通常没有足够的I/O引脚来满足主输入/输出以及两个分块之间的连接。在虚拟线路或任何与之相似的概念提出之前，唯一的解决办法就是将设计再进一步细分到更多的器件中，如图16-6所示。

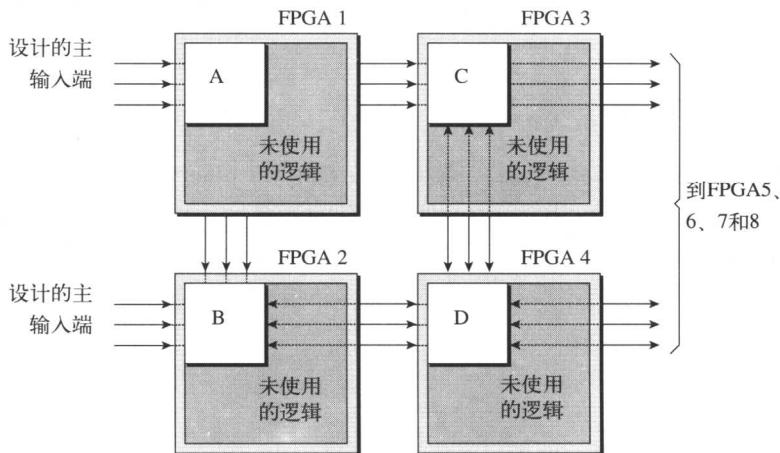


图16-6 将设计分成多块造成FPGA逻辑资源的极大浪费

不过这样做又带来一个新的问题，即每个FPGA中的逻辑资源被严重浪费，而

16.5.2 虚拟线路解决方案

为了了解虚拟线路技术是如何解决问题的，这里先假定一种极端的情况，即所用的FPGA非常独特，只有三个引脚，其中一个输出，两个是输入，并假定其中一个输入是时钟。这种情况下，我们可能只能使用每个FPGA中的很少一部分逻辑资源，如图16-7所示。

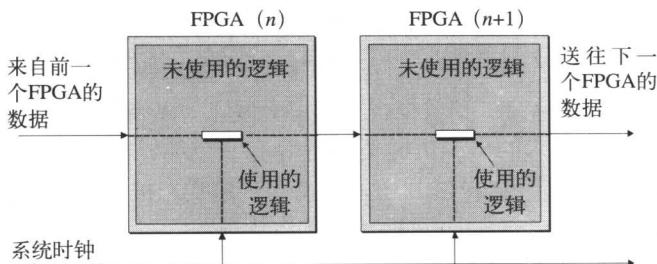


图16-7 一种极端情况：FPGA只有三个引脚

虚拟线路技术的思想是，既然每个器件的内部资源有很大一部分被浪费了，那么可以利用这些资源实现一些专门的电路，通过这些电路可以将单一的输入数据锁存到许多寄存器中，这些寄存器都可以驱动各自的逻辑块。同样，每个逻辑块的输出可以用多路技术转换为一路输出并进行寄存器化处理，如图16-8所示。

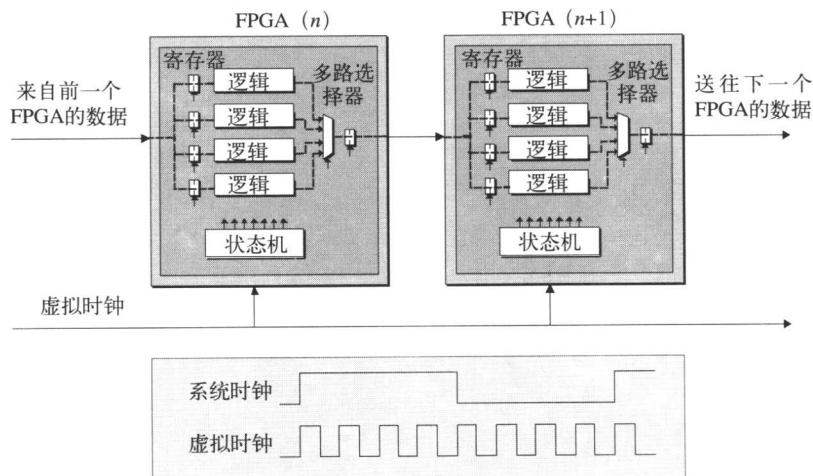


图16-8 虚拟线路的简单例子

需要注意，原始的系统时钟已经被虚拟时钟所代替，虚拟时钟的频率要比系统时钟高许多倍。还要注意，每个FPGA内部都设计了一个状态机，所有这些状态机都用来对各个寄存器进行使能控制，也包括控制多路器等。（当然，同实现实际

逻辑相比，状态机和其他虚拟线路结构实际上只在每个器件中使用很少的逻辑资源，而图16-8中并没有显示出这一点。)

在虚拟时钟的每一个节拍，各个FPGA内的状态机都使能一个寄存器驱动一个逻辑模块，这样从输入引脚来的数据就被存储到对应的那个寄存器中。同时，状态机会控制多路器选择一个逻辑块的输出，并将这个数据存储到与输出引脚相连的一个寄存器中，然后依次驱动下一个FPGA的输入。

当然，在实际应用中，FPGA的引脚都有数百甚至上千个，每一个输入可能驱动几个逻辑模块，每一个输出由各自的虚拟线路多路器驱动，多路器则从许多逻辑块中选择数据。简而言之，不管情况变得多么复杂，基本原理都是相同的。

最后，虚拟线路技术的一个关键要素是要有一个编译器，它能将门级网表形式的原始设计分解到多个FPGA芯片中，并自动创建状态机及其他与虚拟线路有关的结构，然后产生要下载到各个FPGA中的配置文件。

第17章 IP

17.1 IP的来源

现在的FPGA设计，规模巨大而且功能复杂，因此设计的每一部分都从头开始是不切实际的。一种解决方法是：对于较为通用的部分可以重用现有的功能模块，而把主要的时间和资源用在设计中的那些全新的、独特的部分。因为这些部分体现了自己的“独创性”，同时也可和竞争对手的设计区分开来。

任何现有的功能模块一般都称作IP，IP的来源主要有三个：（1）来自前一个设计的内部创建模块；（2）FPGA厂商；（3）第三方IP厂商。为了方便，我们将集中讨论后两种类型的IP。

17.2 人工优化的IP

一种情况是IP厂商从RTL级描述开始对IP进行人工优化（也有可能使用下文介绍的IP模块/核生成软件）。这种情况下，终端用户可以通过几个途径购买和使用这样一个IP模块，如图17-1所示。

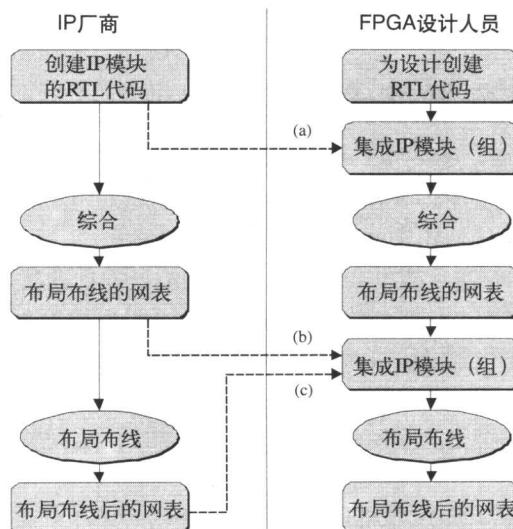


图17-1 可供选择的IP获取点

17.2.1 未加密的RTL级IP

某种情况下，FPGA设计人员可以购买源代码未加密的RTL级IP模块，然后将这些IP模块集成到设计的RTL代码中，如图17-1a所示。应当注意，IP厂商在交付 RTL代码前已经对IP模块进行了仿真、综合和验证。

一般而言，这是一个很昂贵的选择，因为IP厂商通常不想让任何人看到他们的RTL级源代码。当然，尽管FPGA厂商不情愿，但一般都会提供未加密的RTL源代码，因为他们不希望用户转而使用竞争对手的器件。所以，如果真想购买这种未加密的RTL级IP，那么不管提供IP的是谁，他都将束缚住用户的手脚，用户必须签署全部许可文件和保密协议（nondisclosure agreements, NDA）。

假定用户打算购买未加密的RTL级IP，那么一个优点就是可以修改源代码以便移除不需要的功能或者自己增加一些新的功能。另一个优点是，假如不是从FPGA 厂商处而是从第三方购买IP，就可以快速而容易地改变目标器件和FPGA厂商，但是这样做有一个最大的缺点，那就是与下面即将讨论的优化后的网表级IP相比，在资源需求和性能方面的效率会比较低。

17.2.2 加密的RTL级IP

很遗憾，本书写作期间，还没有一个被流行工具普遍支持的RTL加密技术工业标准，这就使得像Altera和Xilinx这样的公司开发了自己的加密算法和工具。结果是由某一家FPGA厂商的工具加密后的RTL代码只能由这家公司的综合工具进行处理，有时候也可以由第三方综合工具处理，但此工具一定是该FPGA厂商的 OEM厂商开发的。

17.2.3 未经布局布线的网表级IP

对于FPGA设计人员来说，或许最普遍的方式就是购买未经布局布线的LUT/CLB网表级IP，如图17-1b所示。这种网表通常以加密形式提供，可能使用加密的EDIF格式，也可能使用某些FPGA厂商的专用格式。

这种情况下，IP厂商可能也会提供一个已经编译好周期精确的C/C++模型，以便进行功能验证，因为这种模型的仿真速度要比LUT/CLB网表级模型快得多。

这种方案的主要优点是，IP厂商已经做了大量的工作来调整综合引擎，并对某些功能进行了人工优化，以便在资源利用和性能方面达到最优。缺点一个是FPGA设计人员无法移除不需要的功能，另一个是IP模块同某家FPGA厂商和具体的器件族紧密相连。

17.2.4 布局布线后的网表级IP

某些情况下，FPGA设计人员可能会购买布局布线后的LUT/CLB网表级IP，如

[289]

图17-1c所示。这种网表通常也以加密形式提供，可能使用加密的EDIF格式，也可能使用某些FPGA厂商的专用格式。

之所以使用布局布线后的IP，其原因是为了得到最高的性能。一些情况下，布局是有相关性的，也就是说所有LUT、CLB和其他构成IP的元素，彼此之间的位置是相对固定的，但是整个IP模块作为一个整体可以放在FPGA中的任意位置。而对于通信或者总线协议方面的IP模块，它们都有专门的I/O引脚要求，这时构成IP模块的元素的位置就绝对不能任意改变。

同样，IP厂商也会提供一个已经编译好的周期精确的C/C++模型，以便进行功能验证，因为这种模型的仿真速度要比LUT/CLB网表级模型快得多。

17.3 IP核生成器

FPGA厂商的另一个非常普遍的做法是提供一个专门的工具作为IP模块/核生成器，有时候EDA厂商、IP厂商甚至一些小型的独立设计工作室也这么做。这些生成器软件几乎总是参数化的，可以由用户指定总线和功能单元的宽度和深度等参数。

第一步，从IP模块/核列表中选择自己需要的一个IP核，然后设定相关的参数；第二步，对于某些IP模块/核，生成器软件可能要求用户从功能列表中选择是否包含某些功能。例如，对于通信模块，就有可能要求用户选择是否包含某种错误检测逻辑；对于CPU核，就有可能省略某些指令或寻址模式等。通过这些设定，生成器软件就可以生成在资源需求和性能方面最具效率的IP模块/核。

根据生成器软件的起源不同或者所签署的许可文件要求不同，生成器的输出可能是加密或未加密的RTL源代码，也有可能是未经布局布线的网表或布局布线后的网表文件。有些情况下，也可能会输出周期精确的C/C++模型用于仿真，如图17-2所示。

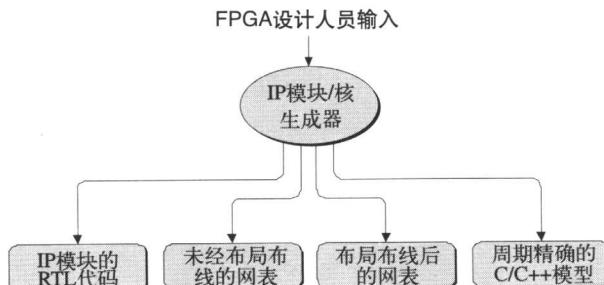


图17-2 IP模块/核生成器

17.4 综合资料

当前，主要的FPGA厂商正在推动一种专门的系统生成工具，这种工具本质上

[290]

是一种IP核集成器，可以很快让用户利用各个FPGA厂商提供的众多IP核搭建起非常复杂的设计。

这些系统生成工具主要以较为抽象的形式为系统设计输出网表（这里的抽象是相对详细的终端用户RTL代码而言的）。这些工具的目标是要在标准的基于RTL的设计流程之上提供一个系统级设计范例，并以此来改变FPGA的设计模型。对这个概念特别感兴趣的应该是那些不愿意写RTL代码，或者那些想在更高的抽象层次上完成工作的设计人员（也可参考第12章）。

为了提供系统生成工具，FPGA厂商也会努力简化IP的使用，一般是将基于IP的设计流程集成到各自的独立开发环境（Independent Development Environment, IDE）中。
[291]

IDE中的D有时候代表“design”，有时候代表“development”，取决于说话对象。

最后，有些过去被认为是软IP的，现在也正在变成硬IP，例如，目前最流行的FPGA中就含有硬处理器、时钟管理器、以太网模块和G比特I/O模块等。这样就将高端ASIC的功能带进了标准FPGA中，随着时间的推移，其他附加功能也有可能被集成到FPGA器件中。
[292]

第18章 ASIC设计与FPGA设计之间的移植

18.1 可供选择的设计方法

创建一个FPGA设计，其实有很多种可能的设计方法，具体选择哪一种取决于用户最终要干什么，如图18-1所示。

	现有设计	新设计	最终实现
只做FPGA设计	N/A	FPGA	FPGA
FPGA到FPGA的转换	FPGA	FPGA	FPGA
FPGA到ASIC的转换	N/A	FPGA	ASIC
ASIC到FPGA的转换	ASIC	FPGA	FPGA

图18-1 可供选择的设计方法

18.1.1 只做FPGA设计

这指的是一个设计只做FPGA实现，此时用户可以使用本书介绍的任何设计流程和工具。

18.1.2 FPGA之间的转换

这指的是将现有的FPGA设计移植到新的FPGA技术中，这里的新技术通常是指同一个FPGA厂商新的器件系列，当然用户也可以转到新的FPGA厂商的器件。

使用这种方法，比较少见的情况是简单地进行一对一的移植，即把现有的整个设计直接移植到一个新的器件中，更常见的情况是，将原来由多个FPGA实现的功能移植到一个单一的FPGA芯片中，或者也可能是将一个或多个现有的FPGA设计，再加上周边的一些离散逻辑，一起放进一个新的FPGA芯片中实现。

这些情况下，一般的做法是将全部原有的RTL级代码和离散逻辑合并到一个新的设计中。设计源代码可能需要作些修改以便能利用目标器件的新特性，然后重新合成。

18.1.3 FPGA到ASIC的转换

这指利用一个或多个FPGA作为ASIC设计的原型。一个较大的问题是，除非用户的ASIC设计是中小规模的，否则常常需要将ASIC设计分解到多个FPGA芯片中。一些EDA厂商和FPGA厂商具有（或曾有）这类软件^①，可以自动对设计作划分，但是这类工具变化太快，而且这些工具的特性、功能和划分结果的质量几乎每周都会有所变化（这是作者婉转的表达方法，读者应该自己去了解最新的情况）。

在本书即将出版的时候，Synopsys公司 (www.synopsys.com) 发表了一个相当令人感兴趣的通告，以该公司著名的Design Compiler[®] ASIC综合引擎为基础，他们已经开发出了一个FPGA优化版本，称为Design Compiler FPGA。除了其他方面外，DC FPGA还使用了一些创新的看起来十分有趣的自适应优化技术 (Adaptive Optimization Technology)。不过重要的是，DC ASIC和DC FPGA在对同一个设计进行ASIC和FPGA实现时可以使用同样的RTL源代码、约束等等，这两个综合引擎都可以通过指令控制来使用不同的微观结构，例如资源共享和流水线级数等。而且，DC FPGA还可以对RTL中ASIC设计的门控时钟进行自动转换。所有这些都使得用FPGA做最终ASIC设计时，将DC FPGA和DC ASIC结合变得十分有趣。

另一个需要考虑的问题是，在使用FPGA内部的嵌入式RAM块实现FIFO存储器或者双口RAM这样的配置时都做了特殊的处理，这些处理方式与ASIC中实现同样的功能所采用的方式一般是不相同的，不注意这点可能会引起一些问题。一个解决方法是创建自己的RTL级ASIC功能库，其中包含乘法器、比较器、存储器以及类似的模块，它们都应该与相应的FPGA实现存在一一对应的关系。但这就要求在RTL源代码中对这些模块进行例化，而不是使用通用RTL描述让综合工具处理一切。所以，和其他工程领域相同，这也是一种平衡。
[294]

正如第7章所讨论的，针对FPGA实现的设计中，各级寄存器之间的逻辑层次要比纯ASIC设计中的少。某些情况下，最好带着用最终的ASIC设计来实现的想法来编写RTL代码，当然，这样做会对FPGA原型设计的性能造成一定的影响。

另外也可以设计两套RTL代码，一套用于FPGA原型开发，另一套用于最终的ASIC设计。但这通常被认为是一种可怕的方法，因为这两套代码很容易失去同步而最终走向两个完全不同的方向。

一个解决方法是使用第11章介绍的纯粹的基于C/C++的工具，这里的意思是，不要故意在RTL源代码中增加智能型的描述（否则会将设计锁定在某种固定的实

^① Synplicity (www.synplicity.com) 公司出品的Certify[®]软件就是具备这种功能的一个好例子。

现上)，而是将所有智能型描述交由控制和引导C/C++综合引擎本身来实现（如图18-2所示）。

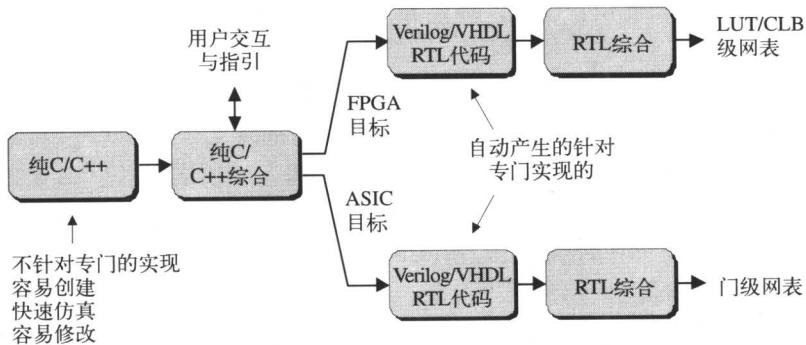


图18-2 纯粹的基于C/C++的设计流程

一旦综合引擎解析完C/C++源代码，就可以用它来权衡一下微观结构并在面积和速度方面评估它们的效果。与每个假设方案相关的用户配置都可以保存下来以备重用，这样，用户可以先创建一个配置文件用于FPGA原型设计，验证后就可以创建第二个配置文件用于最终的ASIC实现，关键是两个设计流程使用同样的C/C++源代码。

1927年，Harold Stephen Black提出一种负反馈的概念，从而使得高保真(Hi-Fi)放大器成为可能。

还应考虑一点，现代ASIC设计所包含的时钟域和子域数量相当多（这里讨论的是数百个域/子域），比较起来，FPGA中的主时钟域就非常有限（与10是同一个数量级）。这就意味着，如果使用一个或者多个FPGA进行ASIC原型设计，就必须花大量的精力考虑如何处理时钟的问题。

最后补充一点，有一个很有意思的专利号为EP0437491 (B1)的欧洲专利，读完后就会发现情况很无聊，专利中所描述的思想就是利用多个可编程器件（如FPGA）临时搭建一个设计，目的是为最终的ASIC设计做准备。实际上，本书作者认为这个专利可能是想利用FPGA建立一个逻辑仿真器，但是所描述的方法将会阻止别人使用两个或者多个FPGA进行ASIC原型开发。

18.1.4 ASIC到FPGA的转换

这是指将现有的ASIC设计移植到FPGA中，原因多种多样，但大都涉及这样一个要求，即想要对现有ASIC的功能进行调整而不花费太多的钱。原有ASIC的制造工艺可能已经被废弃，但是该器件仍然需要继续交货以履行合同（军事项目中

[296] 常有这种情况)。有趣的一点是近来FPGA器件发展非常迅速,几年前的ASIC设计可以完全放进一个单片FPGA中,即使必须将ASIC设计划分到多个FPGA中,如今也有一些工具可以帮助用户完成这项任务,上一节FPGA到ASIC的转换中讨论过这些内容。

首先,需要对原有的RTL代码进行一番精细的梳理,以便移除,至少是评估一下电路中存在的异步逻辑、组合电路反馈环、延迟链以及类似的问题(见第7章),对于既有置位端又有复位端的触发器,需要重新编码,只使用其中之一(见第7章)。另外也需要寻找所有的锁存器,然后使用寄存器取代它们,重新设计电路。还应该检查类似于*if-then-else*的结构有没有else分支,没有这个分支会导致综合工具推断出锁存器(见第9章)。

至于时钟,还必须保证所使用的FPGA器件能提供足够的时钟域以满足原始ASIC设计对于时钟的要求,否则就必须重新设计时钟电路。此外,如果原始ASIC设计使用了门控时钟技术,就必须用具有时钟使能端的等效电路来代替(见第7章)。一些FPGA厂商和EDA厂商提供了具有这种功能的综合工具,可以自动将使用了门控时钟技术的ASIC转换为支持时钟的等效FPGA设计^①。

对于像记忆模块这样的复杂的功能单元(例如FIFO和双口RAM),可能有必要调整RTL代码以使它们适用于FPGA的结构。有些情况下,会调用专门的子电路或者FPGA单元来代替原设计中的一些普通RTL描述(将由综合来处理)。

[297] 最后,原始的流水线型ASIC设计中,各级寄存器之间的逻辑层次要比相应的FPGA设计多,为了保持原有的性能,在转换时需要进行一定的调整。现在大多数逻辑综合工具和物理综合工具都提供时序重调(retiming)的功能,就是将各级流水线寄存器之间的逻辑电路在各级之间进行移动以便平衡分配,从而获得更好的时序性能(物理综合工具在这一点上有更好的表现,见第19章)。

[298] 也可能存在这种情况,现代的FPGA采用比较先进的制造工艺,比如130nm,而原始ASIC设计却用250nm。这就给了FPGA一个内在的速度优势,从而弥补了其固有的线路延时缺陷,然而,在设计的最后阶段,有可能需要手工修改代码以增加更多的流水线平台。

^① Synplicity (www.synplicity.com) 公司出品的Amplify[®]就是具有这种功能的好例子。

第19章 仿真、综合、验证等设计工具

19.1 引言

设计工程师在进行设计输入、验证、综合与实现时通常需要使用大量的设计工具。即使只介绍这些工具就可以写一本书，因此本章只集中讲解与FPGA设计相关的一些较为重要的设计工具，也包含作者本人感兴趣的两个工具：

- 仿真（基于周期、事件驱动等）
- 综合（逻辑/HDL综合与物理综合）
- 时序分析（静态与动态）
- 一般验证
- 形式验证
- 混合设计

19.2 仿真（基于周期、事件驱动等）

19.2.1 什么是事件驱动逻辑仿真器

逻辑仿真目前是设计工程师（或验证工程师）的工具库中最主要的验证工具之一。逻辑仿真最普遍的形式是事件驱动，因为这些工具将事物看成是一系列离散的事件。举一个例子，一个或门（OR gate）驱动一个BUF（buffer）和两个非门（NOT gate），如图19-1所示。

299

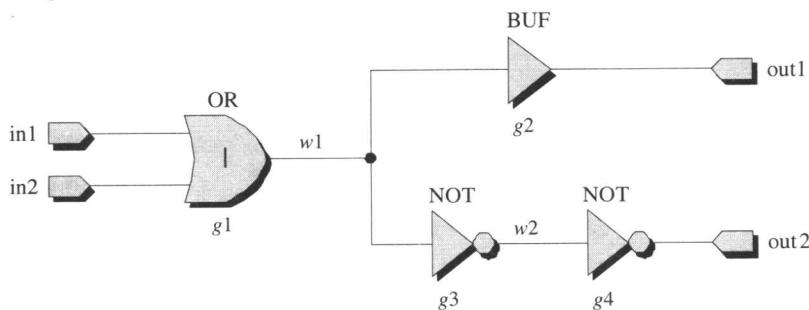


图19-1 电路举例

为了让事情尽量简单，这里假定非门的延迟为5ps，BUF的延迟为10ps，或门

的延迟为15ps，在此基础之上，考虑当输入信号之一发生变化时会出现什么情况（如图19-2所示）。

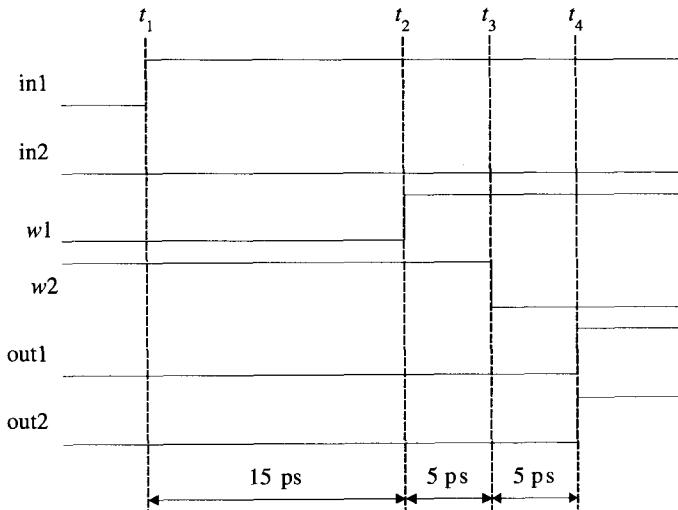


图19-2 事件驱动仿真的结果

300 仿真器在内部维护着某些信息，称为事件轮（event wheel），其中记录着在未来某些时刻要“被激活”的一些事件。当输入信号in1发生第一个事件时，标记为 t_1 ，仿真器检查这个输入连接到了何处，也就是或门，由于或门的延迟为15ps，因此仿真器会在 t_1 时刻后15ps的 t_2 时刻为或门的输出端安排一次事件，即w1上的信号由逻辑0翻转为逻辑1。

然后仿真器检查当前时刻（ t_1 ）是否还有其他动作需要执行，也就是要检查事件轮上下一个事件是什么。在本例中，下一个事件将发生在 t_2 时刻，也就是w1上的上升沿。仿真器在执行这个动作的同时也会检查w1连接到了哪里，这里是指BUF门g2和非门g3。

由于非门g3的延迟为5ps，仿真器会在 t_2 时刻后5ps的 t_3 时刻为g3的输出w2安排一次下降跳变。同样，BUF门g2的延迟为10ps，所以仿真器会在 t_2 时刻后10ps的 t_4 时刻为g2的输出out1安排一次上升跳变。这一过程一直持续下去，直到由输入端in1上的首次变化触发的所有事件都被执行完毕才结束。

这种事件驱动方式的优点是，使用这种技术的仿真器几乎可以处理任何形式的设计，包括同步和异步电路、组合反馈环等。这些仿真器具有极好的设计可见性，方便调试，也可以评估由延迟造成的短脉冲对设计产生的影响，使用其他技术就很难发现这些问题（见下一节中对延迟的讨论）。这些仿真器的最大缺点是计算量非常大，因此速度非常慢。

19.2.2 事件驱动逻辑仿真器发展过程简述

正如第8章所讨论的一样，大约在20世纪60年代末和70年代初出现的第一代事件驱动数字逻辑仿真器是基于仿真基本元件（simulation primitive）这一概念的。[301]这些基本元件至少包括缓冲门（BUF）、非门（NOT）、与门（AND）、与非门（NAND）、或门（OR）、或非门（NOR）、异或门（XOR）和同或门（XNOR）等逻辑门，以及一些三态缓冲门。有些仿真器也提供一些寄存器和锁存器作为基本元件，而其他一些仿真器就需要用户自己创建这些功能，要将几个更基本的逻辑门组成子电路。

在当时，一般使用标准的文本编辑器在门级网表级别上输入设计，同样，也可以采用文本（表式）激励文件作为测试平台（testbench）。仿真器可以接受网表和测试平台，以及一些控制文件和命令行指令，它使用网表在计算机内存中建立一个电路模型，然后将测试平台中的测试激励应用到这个模型上，并将结果输出到一个文本（表式）文件中（如图19-3所示）。

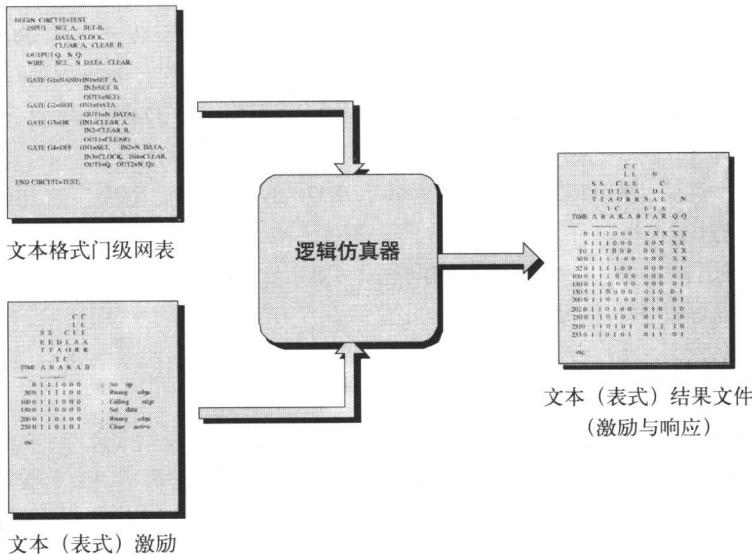


图19-3 运行逻辑仿真器

随后，事情开始变得复杂，首先，出现了原理图输入包，用来输入设计和生成门级网表，接下来，使用专门的显示工具软件读取存储仿真结果的文本文件并以图形化的波形方式显示出来。有时，这些波形显示工具也可以用图形方式输入测试平台，并产生对应的表式激励文件。[302]

再后来，数字仿真器的开发者们开始尝试使用更复杂的语言，可以在更高抽象层上描述逻辑功能，例如在寄存器传输级，即RTL，System HILO仿真器使用的

GHDL (GenRad Hardware Description Language) 语言就是一个很好的例子。

测试平台语言同样也有所发展，例如GWDL (GenRad Waveform Description Language) 语言。这类语言支持循环等复杂的结构，甚至能访问电路的当前状态并相应地改变测试，例如，判断某个输出是否为逻辑0，是则跳转到Test B，否则跳转到Test C。

这些早期的语言在某些方面是超越了那个时代的。例如，GWDL语言具有一个十分有用的特性，除了能指定输入激励（如“input-A = 0”），它还可以指定期望的输出响应（如“output-Y == 1”），一个等号表示对一个输入信号赋值，两个等号则表示期望的输出响应。如果使用了STROBE语句，仿真器就会检查电路的实际响应是否与波形中指定的期望响应相一致，不一致时就产生警告。

几年以后，出现了硬件描述语言（HDL）的工业标准，例如Verilog和VHDL。这些语言的优点是，同一种语言既可以描述电路的功能，又可以写testbench^①，也可参考本章19.5节中对专用验证语言e语言的讨论。

另外，一些记录仿真结果的标准文件格式也开始出现，例如值变转储格式（Value Change Dump, VCD）。这推动了第三方EDA公司开发更复杂的波形显示和分析工具，这些工具可以利用多种仿真器的输出数据工作。后来出现的一种格式是Novas 软件公司（www.novas.com）开发的Fast Signal DataBase™ (FSDB) 格式，这种文件比VCD文件更小，却具有非常快的信息检索速度。

同样，类似于标准延迟格式（Standard Delay Format, SDF）这样的规范也方便了第三方EDA公司开发复杂的时序分析工具，这些工具可以对电路进行评估，产生时序报告，显示出可能存在的问题，并输出SDF文件，用于进行更准确的时序仿真（也可参考其他延迟格式的讨论）。

19.2.3 逻辑值与不同逻辑值系统

当今的数字电子系统中，绝大多数都是基于二进制逻辑的，其中的数字称为位，也就是说，逻辑门使用两种不同的电压来表示二进制数0和1或者布尔逻辑值真（True）和假（False）。也有人作过一些三值逻辑的实验，其技术基础是三进制数，使用三个不同的逻辑电平。然而，到目前为止这种技术还没有得到商业上的应用，谢天谢地，这样一来也让我省事了。

书归正传，要表示二进制逻辑门的操作，至少需要两个逻辑值：0和1。下一步就是要有能力表示未知值，通常用字母X表示，这些未知值可以表示多种状态，例如未初始化的寄存器中存储的内容或者两个门用相反的逻辑值驱动同一线路造成的冲突。另外通常用字母Z来表示由三态门输出驱动的高阻状态。

^① Verilog语言主要的创造者Phil Moorby也是HILO语言和仿真器的设计者之一。

303

304

与使用字母“X”表示“未知”不同，数据手册上通常用“X”表示“无关项”。硬件描述语言倾向于使用“?”或“-”表示“无关项”，而且，不能将“无关项”作为驱动状态赋值给输出；相反，它们通常用来指定模型的输入响应不同的信号组合的情况。

数字仿真逻辑评估系统（如向量积与离散值逼近）和未知 X 值的各个方面在本书作者的“Designus Maximus Unleashed”（美国亚拉巴马州禁止出版）一书中有详细介绍，书号为 ISBN 0-7506-9089-5。

然而，0、1、X和Z这四个状态仅仅只是一小部分。更复杂的逻辑仿真器还能将不同的驱动强度与不同逻辑门的输出联系起来，这样结合起来可以解决多个逻辑门用强度不同而逻辑值也不同的信号驱动同一线路所产生的问题。当然，VHDL和Verilog在处理这类问题时使用的方法有所不同。

19.2.4 混合语言仿真

由于有两种成为工业标准的硬件描述语言（Verilog和VHDL），这就产生一个问题，同一个设计的不同部分可能使用不同的语言。在设计中使用哪种语言，很明显取决于所在公司的喜好，然而，如果想要重新使用其他语言书写的代码时就会产生问题。同样，购买第三方的IP核时，该IP核也可能只有与当前所用语言不同的另一种语言书写的版本。还有一种情况，你的公司与其他公司合并或者合作开发一个项目，而这两个公司原来的设计流程使用不同的语言，这些都会产生问题。

由此就产生了混合语言仿真的概念，其中还带有一些历史的痕迹，早期使用的一项技术是将“外语”（你没有使用的那种语言）翻译成你正在使用的语言，这是很痛苦的，因为不同的语言支持不同的逻辑状态和语言结构（即使同样的语言描述也有不同的语义）。最终的结果是，当你对翻译过来的设计进行仿真时，该设计所表现出来的行为很少能与期望的相一致，所以今天已经很少使用这种方法了。

另一项技术是使用一个VHDL仿真器和一个Verilog仿真器共同仿真两个内核，但这样会严重降低仿真性能，因为每一个内核在等待另一个内核完成操作时永远都处于停止状态。因此，这种方法在今天也很少被使用。[305]

最好的解决方案是使用一个支持VHDL和Verilog语言混合设计的单核仿真器^①。所有在EDA领域时间稍长的人都有自己的这类工具，其中有些工具比较先进，因为它们可以支持多种语言，例如Verilog、SystemVerilog、VHDL、SystemC和PSL（本章19.6节将对PSL进行更详细的介绍）。

^① Mentor Graphics公司（www.mentor.com）的Modelsim[®]就是这种单核仿真器的很好例子。

19.2.5 其他延迟格式

如何在所创建的模型中表示延迟以便让事件驱动仿真器使用取决于两个因素：
 (a) 仿真器本身的延迟建模能力；(b) 在设计流程的哪个阶段，使用什么工具执行时序分析。

一个常用的做法是，在仿真以外单独进行静态时序分析（static timing analysis, STA），这一概念将在本章后面的部分详细讨论。这样，逻辑门或更复杂的描述可能以零（0个时基单位）延迟或单位（1个时基单位）延迟来建模，这里的时基单位是指仿真器认可的最短时间片段。

另外也可以将更复杂的延迟信息与逻辑门（或更复杂的描述）联系起来，以便在仿真中使用。首先要做的是将逻辑门（或更复杂的描述）输出端上的上升延迟和下降延迟分开。由于历史原因，上升（0到1）延迟常被称为LH，代表低变到高，下降（1到0）延迟相应地也被称为HL，即高变到低。例如，考虑图19-4中的BUF，它的延迟属性为LH=5ps，HL=8ps，如果在它的输入端施加一个12ps的正脉冲，那么会发生什么情况呢？

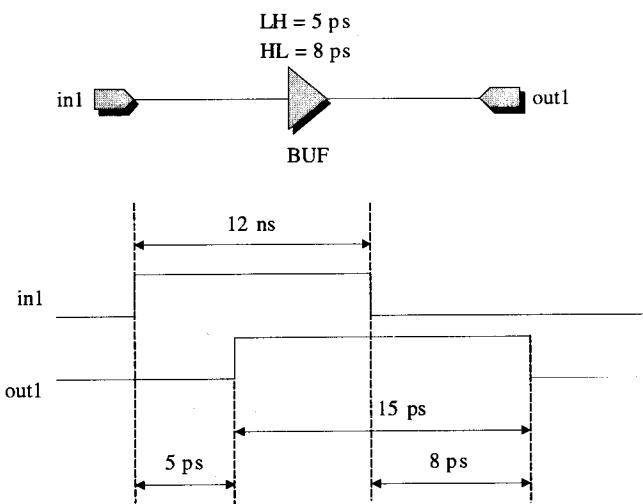


图19-4 分开上升延迟和下降延迟

毫无疑问，BUF的输出在输入端出现上升沿5ps后开始上升，在输入端出现下降沿8ps后开始下降。真正令人感兴趣的一点是，由于上升延迟和下降延迟不平衡，本来12ps的脉冲在输出端被扩展成了15ps，这额外的3ps反映了LH和HL之间的差异。同样，如果输入端的脉冲是一个12ps的负脉冲，那输出端相应的脉冲将缩减到只有9ps（读者可以试着在纸上画出这种情况）。

除了LH和HL延迟，仿真器还支持两种延时每一种的三种值，即最小值：典型

值:最大值，一般缩写为min:typ:max。例如，考虑图19-5中的BUF，在输入端施加一个16 ps的正脉冲，其上升延迟和下降延迟分别为6:8:10 ps 和 7:9:11 ps。

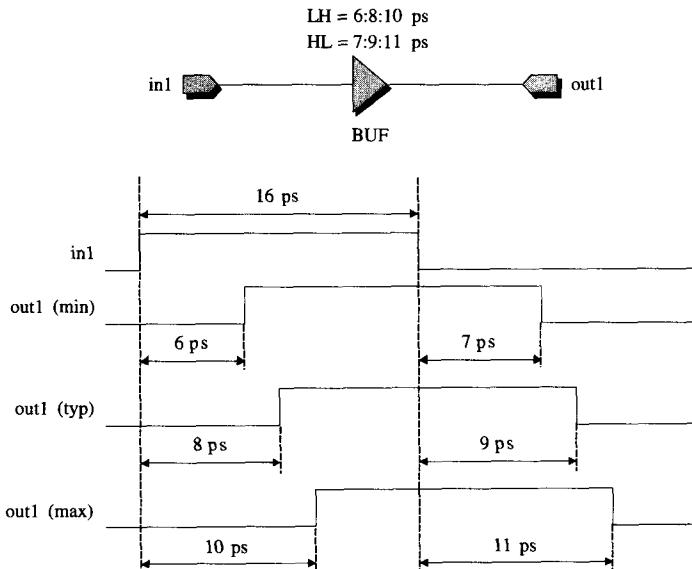


图19-5 支持最小值:典型值:最大值 (min:typ:max) 延迟

TTL即晶体管—晶体管逻辑，是指双极结型晶体管（BJT）通过某种方式连接在一起。

这种表示方法是为了涵盖操作环境的各种变化，如温度和电压等，另外也涵盖了制造过程中的各种变化，因为同一类型的芯片，有些就比另外一些快或慢一点。同样，芯片（ASIC或FPGA）中某个区域的逻辑门也可能比另一个区域的同种逻辑门具有更快或更慢的转换速度。（也可参考本章后面讨论的时序分析部分，尤其是动态时序分析）。

原来，一个多输入逻辑门（或更复杂的描述）中，所有输入到输出的延迟都是相同的，例如，考虑一个三输入与门，输出端为y，输入端为a、b、c，那么a到y、b到y和c到y三条路径的LH和HL延迟都是相同的。开始，这并没有引起任何问题，因为它符合数据手册中指定延迟的方式，但后来的数据手册开始单独指定各个路径输入到输出的延迟，因此仿真器也就必须升级以支持这种能力。

ECL即发射极耦合逻辑，是指按另一种与TTL不同的方式将双极结型晶体管连接在一起。用ECL方式实现的逻辑门比TTL逻辑门有更快的开关速度，但是功耗也更大，因此会散发更多的热量。

另一点也需要考虑，当在逻辑门的输入端施加一个很窄的脉冲时会出现什么情况，这里所说的“很窄”是指脉冲宽度小于门的传输延迟。第一代逻辑仿真器主要是针对电路板级使用的基于晶体管—晶体管逻辑（TTL）技术的简单集成电路，这些芯片通常不会让窄脉冲通过，这也正是仿真器的做法，后来被称为惰性延迟模型。举一个简单的例子，在一个buffer的输入端施加两个正脉冲，脉冲宽度分别为8ps和4ps，buffer的上升延迟和下降延迟都设为6:6:6ps（如图19-6所示）。

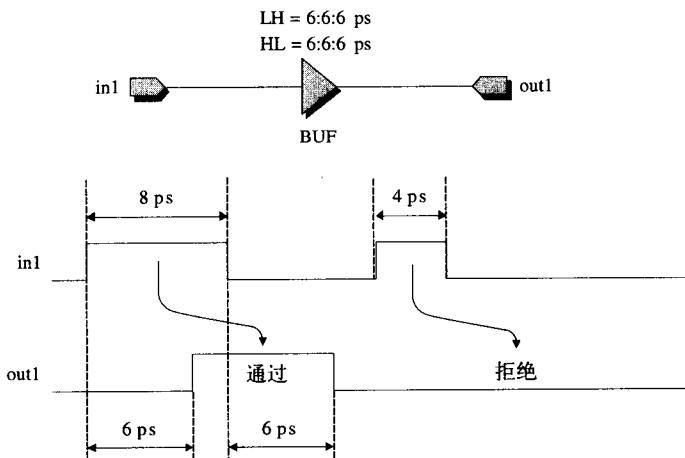


图19-6 惰性延迟模型拒绝任何小于门传输延迟的脉冲通过

309

作为对比，再来看一看用后来的技术实现的逻辑门，例如ECL（发射极耦合逻辑），这种逻辑门允许小于门传输延迟的脉冲通过。为了适应这种情况，有些仿真器配备了一种模型，称为传输延迟模型。再考虑图19-7中的例子，条件同图19-6相同。

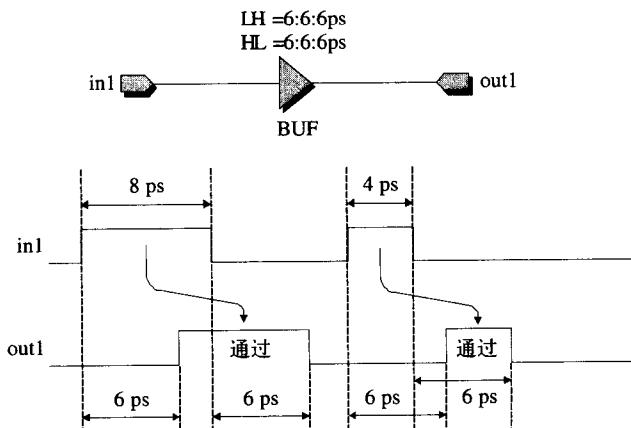


图19-7 传输延迟模型会传播任何脉冲，而不管其宽度

无论惰性延迟模型，还是传输延迟模型，都有一个问题，它们只为极端情况而设，因此一些仿真器的开发者就开始试验更复杂的窄脉冲处理技术，例如三段式延迟模型（three-band delay model）^①。这种情况下，每一个延迟可能被限制为两种值，即r代表拒绝，p代表通过，还要指定各自在整个延迟中所占的百分比，例如，假定一个buffer的延迟为6:6:6 ps，r和p分别为33%和66%（如图19-8所示）。

输入端的任何脉冲，只要宽度大于或等于p值对应的时间就能通过；而那些宽度小于r值对应时间的脉冲就被完全阻挡；宽度介于这两者之间的脉冲则被作为未知值（即X）传输过去，表明这些值是不确定的，因为无法确切地知道它们是否能通过实际应用中的逻辑门。（将r和p都设为100%相当于惰性延迟模型，设为0则相当于传输延迟模型。）

310

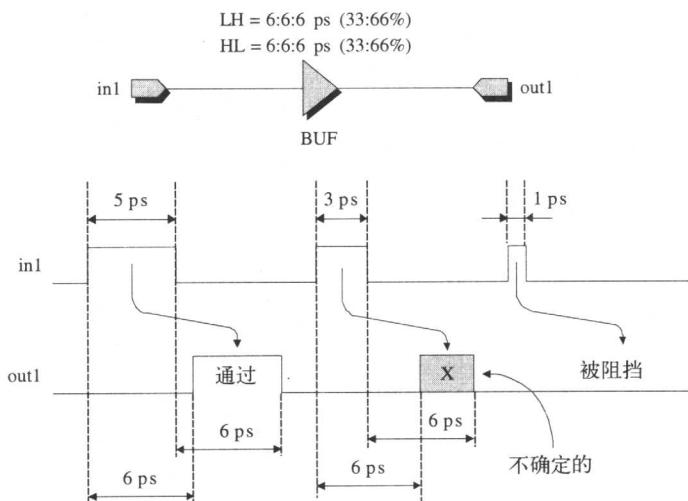


图19-8 三段式延迟模型

19.2.6 基于周期的仿真器

事件驱动方法的一种替代方案是使用基于周期的仿真技术。这比较适合于流水线设计，因为在流水线设计中，组合逻辑像小岛一样夹在各寄存器模块之间（如图19-9所示）。

这种情况下，基于周期的仿真器会忽略组合逻辑中所有逻辑门的时序信息，并把这些逻辑转换为一系列布尔运算，然后直接使用CPU的逻辑运算指令来实现。

对于运行适当的电路，基于周期的仿真器可能比事件驱动的仿真器有更好的运行时（run-time）优势。然而，说到底，这两种仿真器只是使用了逻辑0和逻辑1。

311

^① GenRad公司的System HILO仿真器在其消失前不久就开始使用三段式延迟模型。

1这两个值，并没有使用X、Z，也无法表示驱动强度，而且基于周期的仿真器也不能处理异步逻辑和组合反馈环。

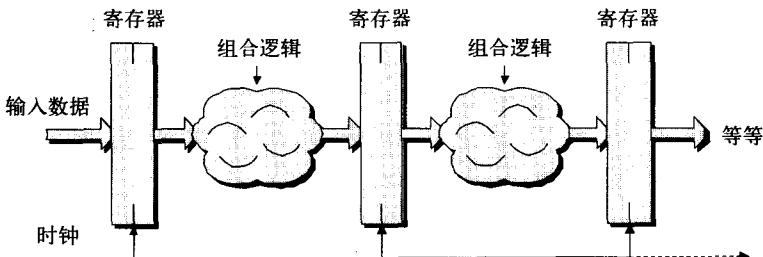


图19-9 一个简单的流水线设计

现在很少能看见有人使用纯粹的基于周期的仿真器，而是增加几个事件驱动仿真器，使具备混合仿真能力。这样，如果要让仿真器以高性能为目标（与时序精确相对），那么仿真器将自动使用事件驱动方法来处理电路中相应的部分，而对电路其他部分则使用基于周期的技术来处理。

19.2.7 选择世界上最好的逻辑仿真器

与工程中其他事情一样，选择逻辑仿真器也是一个寻求平衡的过程。例如，如果启动资金少，而成本是压倒一切的衡量标准，那么就应该跳到第25章关于创建一个基于开源的设计流程的讨论。

首先需要考虑是否需要混合语言仿真的性能，如果启动资金少，可以只使用单一语言仿真器，但是要记住，要购买的任何IP可能并不是这种语言写成的。能与VHDL、Verilog和SystemVerilog协同工作将是一个良好的开端，如果还能处理SystemC以及一种或者几种形式验证语言，也将为你带来极大的好处，节省大量的时间。

312 一般而言，性能是大多数人考虑的首要标准，问题是如何测定一个仿真器的性能而不被它欺骗，唯一的方法就是要有一个自己的基准设计，并能在多个仿真器上运行。创建一个良好的基准设计是一项非常重要的工作，但是这比使用EDA厂商提供的设计要好，因为EDA厂商提供的这些设计都作了专门的调整以支持其解决方案，同时也会给竞争对手的设计工具一些隐喻性的批评。

然而，除了性能以外，还有更多的方面需要考虑，要寻找一个具有良好交互性的调试环境，这样当发现一个问题时，就可以让仿真器停下来，然后在设计中查找错误。不同的仿真器可能是由水平不同的设计团队开发出来的，因此不同的工具所具有的功能级别也有所不同，这样即使仿真器做到了你想要做的事情，也可能需要费些周折才能达到目的。所以，诀窍是，先运行基准设计，再拿一个已知bug的同样电路做验证，看一下需要花多长时间，是否能很容易地找到问题的所在。

实际上，有些仿真器已经做了这一乏味的工作，但必须使用第三方的仿真后分析工具^①。

另外需要考虑的是仿真器的容量。大型EDA厂商提供的工具本质上是没有容量限制的，但是一些小型厂商的仿真器则不然，深入研究就会发现它们可能是基于有限的32位代码开发的。当然，如果只做一些50万门以下的小型设计，使用FPGA厂商提供的仿真器已经足够了，这些仿真器一般是大型EDA厂商所提供工具的精简版本。

当然，除了上面提到的几项外，你也会有自己的评判标准，例如代码覆盖的质量以及不同工具提供的性能分析等。这些功能过去都属于专门的第三方工具的领域，不过现在大多数较大的仿真器都在自身的仿真环境中集成了代码覆盖率检查和性能分析的功能，但不同的仿真器提供的特征集有所不同（也可参考本章后面“混合设计”一节中讨论的代码覆盖率和性能分析的内容）。

313

19.3 综合（逻辑/HDL综合与物理综合）

19.3.1 逻辑/HDL综合技术

传统的逻辑综合技术大约是在20世纪80年代早期到中期出现的，现在这些技术常被称为HDL综合技术。

最初的逻辑/HDL综合工具的作用是将ASIC设计的RTL描述和一组时序约束进行综合，生成相应的门级网表。在此期间，综合软件会执行一系列的精简和优化（包括面积和时序优化）。

大约在90年代中期，综合工具有了一些增强，可以解释FPGA的体系结构概念。这些具有结构化概念的工具能够输出LUT/CLB级网表，随后将这些网表传给FPGA厂商的布局布线软件来处理（如图19-10所示）。

实际应用中，由具有结构化概念的综合工具综合出来的FPGA设计要比传统门级综合工具综合出来的结果快15%~20%。

19.3.2 物理综合技术

传统的逻辑/HDL综合技术存在的一个问题是在开发这项技术时，逻辑门的延迟在时序路径中占有极大的比例，而轨道延迟比较起来就显得无关紧要，这就意味着综合工具可以使用简单的线路负载模型来估算轨道延迟的影响（这些简单模型是这样计算负载的，即一个负载门相当于 x pF的电容，两个负载门相当于 y pF的电容等）。这样综合工具就可以将每条轨道延迟作为其负载的函数来进行估算，

314

^① Novas软件公司（www.novas.com）凭借其Debussy[®]和VerdiTM工具在业界处于领先地位。

并估算门的驱动强度。

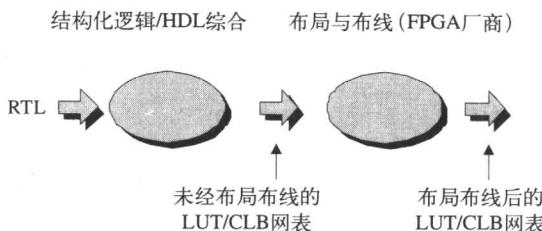


图19-10 传统的逻辑/HDL综合

这一技术对于当时的设计来说是足够的，因为当时的设计采用几微米的制造工艺，所含的逻辑门数量同今天的标准相比少很多，现代设计可能包含几千万逻辑门，而且深亚微米的特征尺寸意味着轨道延迟有可能占到总延迟的80%以上。使用传统的逻辑/HDL综合技术综合这一类设计时，综合工具所估算出来的时序很少能与实际相符，因此想实现时序收敛几乎是不可能的。

由于这个原因，ASIC设计流程在1996年左右开始使用物理综合技术，FPGA设计流程也大约于2000年或2001年开始采用类似的技术。当然，关于物理综合的意思有许多种定义，核心的概念是：在综合过程的早期就要利用物理信息，但具体是指什么呢？举个例子，有些公司在其综合引擎前端增加了交互式的布局能力，并称之为物理综合。但是，对于大多数人而言，物理综合就意味着将实际的布局信息与设计中的各种逻辑元素联系起来，并利用这些信息估算精确的线路延迟，再利用这些延迟信息对布局进行微调和其他优化。有趣的是，物理综合是以使用相对传统的逻辑/HDL综合引擎先运行开始（如图19-11所示）。

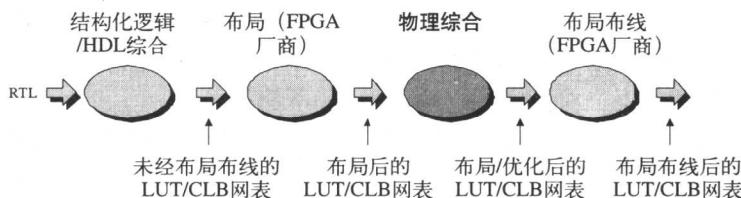


图19-11 物理综合

1933年，Edwin Howard Armstrong构思出一种新的无线电通信系统——多频率调制系统，即wideband frequency's modulation (FM)。

19.3.3 时序重调、复制及二次综合

在物理综合的相关文章中有许多术语，包括时序重调（retiming）、复制

(replication) 及二次综合 (resynthesis)^①。首先，时序重调是基于这样一种概念，即对整个设计的正负slack进行平衡，这里，正slack是指一条路径的可用延迟时间没有被用完，还有富余，而负slack是指一条路径的实际延迟时间大于其可用延迟时间。

例如，假定一个流水线设计的时钟频率使得寄存器到寄存器的延迟为 15 ps，再假定如图19-12a所示的情况：第一个组合逻辑块中最长的时序路径为 10 ps（也就是它的正slack是 5 ps），而第二个组合逻辑块中最长的时序路径为 20 ps（也就是它的负slack是 5 ps）。

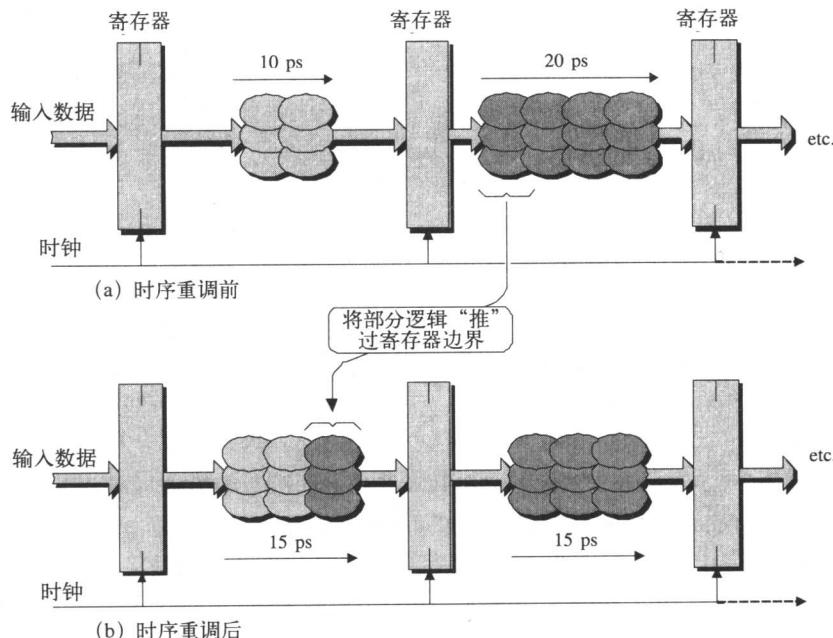


图19-12 时序重调

一旦计算完原始的路径时序，包括布线延迟，就可以将组合逻辑进行移动，越过寄存器的边界（反之亦然，取决于你的想法），等于从slack为正的路径中借来时序补偿给slack为负的路径（如图19-12b所示）。时序重调在FPGA的物理设计流程中用得非常普遍，因为FPGA内部有大量的寄存器资源。

复制与时序重调相似，但是它的重点是打断过长的内部互连。例如，假定有一个寄存器，其输入端有4ps的正slack，再假设这个寄存器驱动三条路径，各个路径上都是负的slack（如图19-13a所示）。

通过复制寄存器并让这些复制的寄存器离每一个负载最近，就可以重新分配

^① 传统的逻辑/HDL综合也有可能使用这些概念，但用于物理综合时十分有效。

317 slack以使所有的时序路径都能工作（如图19-13b所示）。

最后，二次综合的概念是基于这样一个事实，即实现同一个功能可能有许多种不同的方法。二次综合就是利用物理布局信息对关键路径进行局部优化，所用的方法包括逻辑重构、重新分组、置换，可能还会移除某些逻辑门和线路。

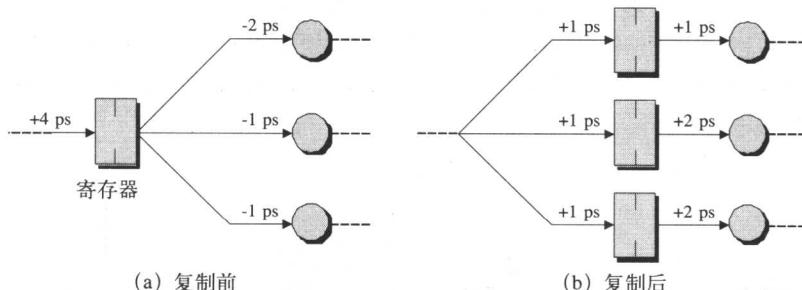


图19-13 复制

19.3.4 选择世界上最好的综合工具

认真一点，难道你真的不想在这里找到答案吗？实际上，各种综合引擎的性能以及相关特征（例如自动交互式布局）几乎每天都在变化，各个综合工具厂商之间也在相互竞争，交替前进。

也存在这样一个事实：不同的综合引擎对于不同的FPGA体系结构进行综合时，表现有好有坏。需要探究的一点是综合引擎进行自动推断的能力（或缺乏），如在源代码或者约束文件中没有明确地定义的时钟元素和嵌入式功能单元等，而综合工具能自动推断出这些单元。但工具评估并罗列出各种推断的结果时，最终还得靠你自己来做决定，如果愿意，请发电子邮件到max@techbites.com，告诉我你是如何做的。

318

19.4 时序分析（静态与动态）

19.4.1 静态时序分析

现在使用最普遍的一种时序验证形式就是静态时序分析，即STA，这从概念上讲是相当简单的，但实际上通常要比给人的第一印象复杂得多。

时序分析工具本质上是将每一条路径上的所有门延迟和轨道延迟加在一起构成这些路径从输入到输出的总延迟（在流水线设计中，分析工具计算的是从一级寄存器到下一级寄存器的延迟）。

布局布线之前，分析工具会对轨道延迟作一些估计，在接下来的布局布线中，

分析工具将利用从物理线路上提取出来的寄生参数（电阻和电容）计算出更精确的结果。任何不符合原始时序约束的路径都会由分析工具报告出来，同时也会对寄存器或者锁存器输入端的信号可能出现的时序问题（例如建立和保持时间相违背等）给出警告。

静态时序分析（STA）特别适合于经典的同步设计和流水线结构，其主要的优点是速度很快，不需要testbench，能完全测试每一条路径；另一方面，静态时序分析工具探测在设计的正常操作中永远不会出现的虚假路径时会出现很少一些不可靠的情况，这些工具在处理含有锁存器、异步电路和组合反馈环的设计时也存在不足。

19.4.2 统计静态时序分析

静态时序分析是现代ASIC设计和FPGA设计流程中的支柱，但在处理最近的制造工艺节点时开始遇到一些问题。写作本书时，90纳米节点正在逐渐成为主流，而45纳米节点有望在2007年前后成为主流。

正如前面所讨论的，现代的硅芯片中，内连线延迟要比逻辑延迟大得多，尤其是在FPGA结构中。内连线延迟的大小取决于寄生电容、寄生电阻和寄生电感的值，而这些寄生效应又与内部连线的拓扑结构和剖面形状有很大的关系。[319]

在最新的制造工艺条件下，光刻制程存在一个问题，已经不能产生足够精确的形状，这样一来，本来要刻成正方形和矩形，现在却刻成了圆形和椭圆形。像线宽这样的特征尺寸现在已经非常小了，以至于在蚀刻制程中很微小的变化都可能引起偏差，尽管这些偏差都很微弱，但与特征尺寸比起来就显得相当严重了（高频设计中，这些不规则的现象更加严重，出现了所谓的集肤效应或表面效应，即高频信号只在导体的外表面流动）。此外，化学机械抛光（CMP）这类制程也会引起线路横截面的变化。

这些因素综合起来造成的后果是：精确地估计线路延迟越来越困难。当然，可以像传统工程学那样预留足够的余量（最差情况下的估计值），但过分保守的设计会导致性能远远低于芯片的全部潜力，这在今天高度竞争的市场条件下是极其不引人注意的。事实上，几何形状改变的效应正在引起延迟随机分配变得越来越广泛，以至于最差情况下估算出来的延迟数据甚至比早期制造工艺下的数据还要大。

一个可能的解决方案是统计静态时序分析（statistical static timing analyzer, SSTA）。这一概念的基础是要为每条线路上每一段的每个信号延迟生成一个概率函数，然后再估算出信号通过整条路径时的总延迟概率函数。写本书时，还没有可交付使用的商业SSTA产品，但是有许多EDA界人士和学术机构正在研究这项技术。[320]

19.4.3 动态时序分析

时序验证的另一种形式叫做动态时序分析（dynamic timing analysis, DTA），

现在确实不常见，不过这里提到它是因为兴趣。这种验证形式需要使用事件驱动仿真器，而且必须使用testbench。标准的事件驱动仿真器和动态时序分析器之间的主要差别是，前者只能在每条路径上使用单一的三值延迟，即最小值：典型值：最大值；而后者可以使用延迟对，即min:typ, typ:max, min:max等。例如，图19-14中显示了两种仿真器对一个简单的buffer进行延迟估算的情况。

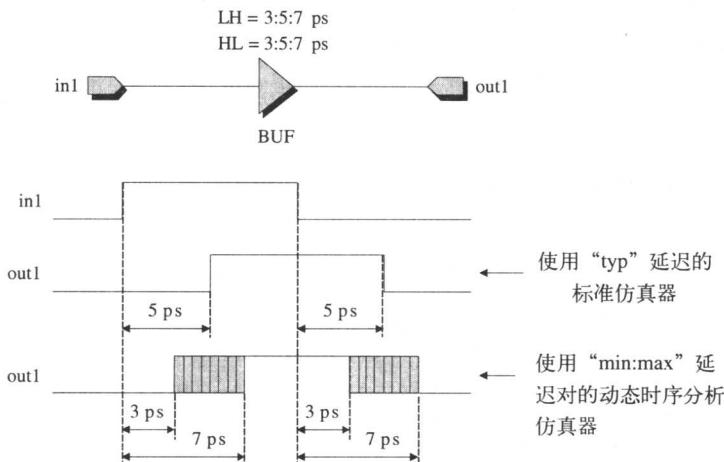


图19-14 标准的事件驱动仿真器和动态时序分析器

使用标准仿真器时，逻辑门输入端上的一个信号变化将会引起仿真器在将来某个时刻安排一个事件；比较起来，使用动态时序分析器时，假设用的是min:max延迟对，逻辑门的输出将会在最小的延迟后开始转换，但是这种转换直到最大延迟时才会结束，也就是说，从最小延迟到最大延迟这段时间，转换都可能发生。

[321]

这两者之间的不确定状态与未知状态X是不同的，因为我们确切地知道0到1的转换或者1到0转换将要发生，只是不知道是在什么时候。由于这个原因，我们引入两个新的状态，即“必定变为高电平，但时间未知”和“必定变为低电平，但时间未知”。

动态时序分析可以检测到一些细微的、潜在的问题，使用其他形式的时序分析基本不可能发现这些问题，但是，这些工具需要极大的计算量，因此在短期内还无法看到它们，但将来会怎么样无人知道。

19.5 一般验证

19.5.1 验证IP

当设计变得越来越复杂，相应的功能验证也就消耗越来越多的时间和资源。

这种验证包括：建立一个验证环境、创建一个测试平台（testbench）执行逻辑仿真、分析仿真结果以检测和定位问题等等。实际上，对于今天的高端ASIC、SoC或FPGA设计来说，从最初的概念到最终实现的全部开发过程中，至少有70%的时间花在了验证上。

缓解这一问题的一个方法是利用验证IP。思路是一个设计，在验证时称为被测器件，即device under test（DUT），通常使用标准接口和协议与外界通信。此外，与DUT通信的器件一般是微处理器、外围设备、仲裁器等。

进行功能验证时最普遍的技术就是使用工业标准的事件驱动逻辑仿真器。测试DUT的一个方法是创建testbench，在比特级上精确地描述输入端上的信号和输出端上期望的输出。然而，现在的各种接口和总线的协议是非常复杂的，以这种方式创建一个测试环境不太可能。

另一项技术是使用所有外部器件的RTL模型来构成系统的其余部分，但是，这些器件中许多是私有的，不可能很容易地得到RTL模型。另外，完全使用所有微处理器和I/O设备的功能模型对整个系统进行仿真，在时间和计算量需求上将是极其昂贵的。

解决方法是使用验证IP，用总线功能模型（bus functional models, BFMs）来表示处理器和I/O设备以构成被测系统^①（如图19-15所示）。

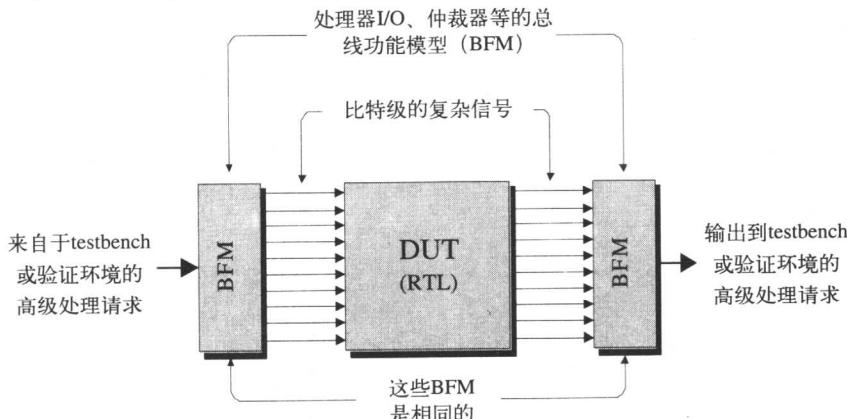


图19-15 使用BFM形式的验证IP

BFM不会复制它所描述的设备的全部功能，而是通过产生和接收一些事务处理（transaction）来模仿设备在总线接口上的行为，这里的transaction是指一种高级总线事件，例如执行一个读或写周期。验证环境（或testbench）可以指示BFM执行一个专门的处理，如写存储器，然后BFM会在总线上产生一些复杂的底层

^① 复杂验证IP的一个发起者是TransEDA PLC公司 (www.transeda.com)。

322

323

“bit-twiddling”信号交互以驱动DUT对用户的接口。

同样，当DUT（某个设计）以复杂的信号模式作为响应时，另一个BFM（也可能是原来的BFM）就能解释这些信号并将它们翻译成相应的高级处理。也可参考下面对验证环境和创建testbench的讨论。

应该注意，尽管这些BFM比它们代表的设备要简单得多也小得多（因此而使仿真更快），但它们却非常重要。例如，复杂的BFM常以周期精确、位精确的C/C++模型来创建，其中可能还包含了内部cache以及对cache初始化的功能，内部缓存、可配置寄存器、序列写回等功能。而且，BFM可以提供大量的参数以便对底层行为进行控制，例如地址时序、侦测时序，针对不同存储设备的数据等待状态等。

19.5.2 验证环境和创建testbench

作者年轻的时候就开始接触仿真，创建了仿真中用的测试向量（激励和响应），并存成只包含逻辑0和1的ASCII码文本文件（幸运的话包含十六进制数值）。那时要测试的设计同今天的巨型设计相比简直简单得让人难以置信，翻译成自然语言就像是一些流水账记录，例如：在时刻1 000使reset信号处于激活状态；在时刻2 000再让reset信号进入非激活状态；在时刻2 500检查8位数据总线上的数据是否为00000000等。

后来，设计变得越发复杂起来，而且随着一些可指定测试激励和响应的高级语言的出现，验证这些设计的方法也变得越来越复杂。这些高级语言具有多种特征，例如循环结构以及能随着输出状态的不同改变测试的能力（如果状态总线上的值为010，那么跳转到测试xyz中），后来人们开始将这些测试称作testbench，即测试平台^①。

现在的情况是，许多设计都太复杂，因此手工创建一个合适的testbench几乎是不可能的，这就为那些高级验证环境和语言的出现铺平了道路。这些语言被人们称为硬件验证语言（hardware verification language, HVL），其中最先进的要算Verisity Design公司（www.verisity.com）^②推出的面向对象的硬件验证语言e语言了。

实际上，e语言现在并不能代表一切，但当初它是为了反映“类英语”的想法而设计的，因为这一语言看起来有些像自然语言。如果愿意可以用e语言作定向测试，但是人们通常只是在某些特殊情况下才这么做。e语言混合了C、Verilog和Pascal的一些元素，它的本质概念是声明输入信号值的有效范围和序列（连同对应

^① 从书面意义上讲，testbench实际上指的是支持测试运行过程的基础结构。

^② 工业界基本倾向于对公司的私有语言持怀疑态度，所以Verisity公司同IEEE组织一起努力将e语言发展成为一个工业标准语言。本书写作期间，IEEE的P1647工作组已经成立，e语言参考手册（LRM）也已经出版。

325

的无效范围和序列) 以及高层次验证策略, 验证环境使用*e*语言的描述来引导仿真过程。

第一个(在写作本书时也是唯一的一个)完全利用了*e*语言能力的验证环境是Verisity公司的出品的Specman Elite[®]。我们可以认为Specman是编译器和事件驱动仿真器之间的交叉, 因为它连接和控制已经使用的标准HDL事件驱动仿真器。Specman利用*e*程序来产生测试激励并通过HDL仿真器将激励加到设计上, 它也可以监控仿真结果和仿真的功能覆盖率, 并且依据监控到的结果动态地调整后面的测试激励以便填补任何残余的覆盖率漏洞。

19.5.3 分析仿真结果

几乎每一个仿真器都有一个图形界面的波形浏览器, 在仿真器运行的时候, 利用它可以交互地显示仿真结果, 或者从仿真后产生的值改变堆存(VCD)文件中读取波形数据并显示仿真结果。

然而, 美中不足的是, 这些工具在真正要分析数据和跟踪问题时往往没有想象中有效, 这时, 可以使用第三方工具^①。

19.6 形式验证

尽管一些大型的计算机和芯片公司(如IBM、Intel和Motorola)大约从20世纪80年代中期开始就已经在公司内部开发和使用形式工具, 但对于大多数人来说, 整个形式验证(FV)领域都是相当新鲜的。在FPGA设计领域尤其是这样, 形式验证在FPGA设计中的应用要落后于在ASIC设计领域中的应用, 它是一个非常强大的工具, 越来越多的人已经开始认真地使用它了。

一个较大的问题是: 对于主流应用来说形式验证仍然是一个全新的概念, 以至于许多从业人员都在各种令人眼花缭乱的方向上乐此不疲地为自己充电。这不是因为没有标准, 而是如今的选择太多, 人们难于决定。几乎每一个人都被这么多的选择弄得晕头转向, 使得混乱状况更加严重。例如, 如果请20个EDA厂商定义断言(assertion)和属性(property), 你的脑髓可能会从耳朵里漏出来^②。

326

形式验证工具最初是大型计算机公司和芯片公司为了内部使用而开发的。首批被广泛接受的商业形式验证工具之一是名为“Design VERIFYer[®]”的等效性检查器, 该工具由Chrysalis Symbolic Design公司于1993年开发。

^① 在经典的波形分析、调试和显示工具中, 工业界公认的领导者之一是拥有Debussey[®]工具的Novas软件公司(www.novas.com)。该公司另一个非常值得一提的工具是VerdiTM, 这个工具提供了一种强大且极具创新性的方法, 可以对设计中穿越多时钟周期的瞬时行为进行提取、观察、分析、探测和调试。

^② 关于这一点, 作者是有着痛苦的经历的。

模型检查工具原先也是大型公司为了内部使用而开发的，由Chrysalis公司于1996年开发的Design inSIGHT®标志着模型检查技术的首次商业应用。

要想解开这团乱麻，说得最轻也是一件令人害怕的事情。不过，正如我那亲爱的老父亲过去常说的，除了害怕本身以外，没有什么可害怕的，所以让我们拨开迷雾用一种大家都能够理解的方式来描述形式验证技术。

19.6.1 形式验证的不同种类

不久之前，大多数设计工程师都认为形式验证（formal verification）和等效性检查（equivalency checking）是同义词。这里，等效性检查器是一个工具，它使用严格的数学技术对同一个设计的两种不同表述（即RTL描述和门级网表描述）进行比较，检查两者在输入输出功能上是否一致。

实际上，可以将等效性检查看作是形式验证的一个子集，称为模型检查（model checking），这种技术可以用来探测一个系统的状态空间以便测试某种属性（通常以断言形式指定）是否为真。327 属性和断言的定义将在本节后面的部分介绍。

为了方便这里的讨论，我们先认为形式验证就是模型检查。不过，应该注意的是形式验证还有另一个类别，叫做自动推理（automated reasoning），就是用逻辑推理来证明具体实现和相关的规范一致，这更像一个数学形式的论据。

19.6.2 形式验证究竟是什么

为了给我们的讨论提供一个开始点，假定有一个设计包含许多子模块，而当前正在使用其中的一个模块，这个模块的作用是执行一些特定的功能。除了为这个模块定义功能的HDL表述外，还可以将几个断言/属性同这个模块联系起来，与这些断言/属性相联系的既可以是模块接口上的信号，也可以是模块内部信号和寄存器。

一个非常简单的断言/属性可能是这样的：信号A和信号B永远不能同时活动或变为低电平。不过这些声明也可以扩展为极其复杂的事务级结构，例如：当收到一个PCI写命令时，某种类型的存储器写命令必须在5到36个时钟周期内发出。

这样，断言/属性就允许用户以一种正式且严格的方式来描述时基系统的行为，这种方式为设计意图提供了一种明确而通用的表述（试着说得快点）。此外，断言/属性还可以用来描述期望的行为和禁止的行为。

断言/属性既可以人工读，又可以机读，这就使得它成为理想的可执行规范的设计工具，但它的功能还不止这些，我们再回到前面提到的那个简单的例子，即信号A和信号B永远不能同时活动或变为低电平。我们经常听到的一个术语是基于断言的验证（Assertion-Based Verification, ABV），它来源于几个方面：仿真、静

态形式验证和动态形式验证。静态形式验证中有一个合适的工具读取设计的功能描述（通常在RTL级），然后尽可能对逻辑进行分析以保证这种情况不会发生；比较起来，在动态形式验证中，增强的逻辑仿真器将概括到某一点上，然后暂停并自动调用相关的形式验证工具（这一内容将在下面详细讨论）。

当然，断言/属性可以在任何级别与设计联系起来，从单独的模块到模块的接口，甚至到整个系统，这就产生了一个非常重要的概念，即验证重用（verification reuse）。在形式验证出现以前，很少有办法能实现验证重用。例如，购买一个IP核时，通常会配备一个相关的testbench，这个testbench集中测试IP核的I/O信号，这样可以对IP核进行独立验证，但是一旦将这个IP核集成到设计中，附带的testbench基本上就没有用了。

现在来考虑购买一个IP核，并附带一套预定义断言/属性，例如，在信号B激活后的三个周期内，信号A不应该出现上升沿。这些断言/属性为建立IP核开发方和下游用户之间的沟通接口提供了极好的机制。此外，即使这个IP核被集成到了设计中，这些断言/属性仍然可用，并且可由验证环境评估。

至于和系统的主要输入输出有关的断言/属性，验证环境可能会用它们自动创建测试激励以驱动设计。另外，还可以用断言/属性贯穿整个设计，增强代码和功能覆盖率分析（也可参考“混合设计”一节）以确保特定的动作或状态序列已被执行。

329

19.6.3 术语及定义

既然我们已经全面介绍了形式验证中模型检查方面的概念，我们就能够通过这些术语和定义更好地继续“向前跋涉”了，不过公平地说，这是一片相当陌生的“水域”（里面可能会有危险动物）。以下部分收集自与许多人的谈话，并尽可能对这些谈话之间的差异进行了合理化处理。

□ 断言/属性 (Assertions/properties)：属性这一术语来源于模型检查领域，指的是要验证的设计所具有的某种功能行为，例如：请求后，希望在10个时钟周期内得到应答；断言来源于仿真领域，指的是仿真期间要监控的设计所具有的某种功能行为，如果断言“触发”则标识一次违例。

目前，可以在统一的环境和方法中使用形式化工具和仿真工具，因而属性和断言可交替使用，即属性就是断言，反之亦然。一般而言，我们把断言/属性理解为关于某个与设计有关的具体特性的一种声明，并且期望这种特性在逻辑上为真，这样，断言/属性就可以被用作检查器/监控器或者作为形式验证证据的目标，它们通常可以鉴别或捕获一些不希望出现的异常行为。

□ 约束 (Constraints)：在形式验证中，这一术语来自模型检查领域。形式验

330

证工具在执行其功能时会考虑所有可能的输入组合，因此有必要将输入限制在合理的行为上，否则对于那些在实际设计中一般不会发生的属性违例也会报错。

与属性相似，约束既可简单也可复杂。有些情况下，约束可能被解释为要证明的属性，例如，某个模块的输入约束可能也是驱动这个输入的另一个模块的输出属性，这样，属性和约束在本质上可能是等同的。另外，约束也可以用在“受约束的随机仿真”领域，这时候的约束通常用来指定驱动总线的数值范围。

□ 事件 (Event)：和断言/属性类似，一般而言，事件可能被看作是断言/属性的一个子集。但是，它们也有差别，断言/属性一般用来捕获异常行为，而事件则用来指定期望的正常行为以便进行功能覆盖率分析。有些情况下，断言/属性可能由一系列事件组成，事件也可以用来指定一个窗口，在这个窗口内可以对断言/属性进行测试，例如：a, b, c之后，我们希望d为真，一直到e发生。这里的a, b, c和e都是事件，而d是将要验证的行为。

对事件和断言/属性的发生进行测量会产生一定量的数据，根据这些数据可以对设计的边界条件和其他属性进行验证。关于事件和断言/属性的统计资料也可以用来为设计产生功能覆盖率指标。

□ 进程 (Procedural)：进程是指在一个执行过程的上下文中或者一组连续语句中（例如VHDL语言中的process或者Verilog语言中的“always”块）描述的断言/属性/事件/约束（所以有时候也称为“上下文中的断言/属性”）。这种情况下，断言/属性内建在设计逻辑中，通过一组连续语句中的路径才能对其进行计算。

□ 声明 (Declarative)：声明是指设计的结构化上下文中存在的断言/属性/事件/约束，连同设计中其他结构化元素（例如采用结构化实例形式的模块）一起被评估。换句话说，声明后的断言/属性总是“打开/活动”(on/active)的，而进程中对应的概念只在执行HDL代码中某个特殊的路径时才是“打开/活动”(on/active)的。

□ 编译指令 (Pragma)：Pragma是“pragmatic information”的缩写，是指插入到源代码（包括C/C++和HDL代码）中的某种专门的伪注释，它们可以由编译器解释，也可以被其他工具使用（这是一个通用术语，除了形式验证技术外，其他许多工具都在使用基于编译指令的技术）。

331

19.6.4 其他可选的断言/属性规范技术

真正有意思的内容从这里开始，因为有很多种方法可以实现断言/属性，概括如下。

- 专门的语言：是指使用一种正式的断言/属性语言，这种语言是为了最大效率地指定断言/属性而专门构造的。Sugar、PSL和OVA就是这种语言的好例子，它们在创建复杂、规则和暂时性的表达式时具有非常强大的能力，而且还能用很少的代码描述复杂的行为（Sugar、PSL和OVA在本章后面有详细介绍）。

这些语言常用来在附属文件中定义断言/属性，附属文件的维护是在HDL主设计之外进行的。可以在解析/编译期间对这些附属文件进行存取，实现时则用声明的方式。另一种选择是，一些增强功能的解析器/编译器/仿真器允许在HDL代码中直接嵌入用这些专门语言写的语句，可以是代码形式，也可以是编译指令形式。在这两种情况下，可以使用声明和/或进程方式实现这些语句的描述。[332]

1936年，发明荧光灯。

- HDL语言中专门的描述语句（special statements in the HDL itself）：从一开始，VHDL语言中就有一种简单的assert语句，可以检查布尔表达式的值，如果表达式的值为假（false）还可以按用户指定格式显示文本信息。Verilog语言刚开始并没有这种语句，但是SystemVerilog语言已经对此作了增强，具有了这种功能。

这一技术的优点是，综合工具会忽略这些语句，所以不用采取专门的措施来防止这些语句被综合成逻辑门；缺点是和专门的断言/属性语言相比过于简单，无法用来指定复杂的临时性序列（尽管SystemVerilog语言在这方面比VHDL语言稍好一点）。

- 用HDL写的模型和HDL调用的模型：这个概念是指访问内部或外部的模型库。这些模型用标准的HDL语句描述了断言/属性，也可以像其他模块一样在设计中被实例化，但是，设计中会用编译指令对这些实例进行封装，以保证不对它们进行物理实现。此方法的一个很好的例子是Accellera标准委员会（www.accellera.org）的开放验证库（open verification library, OVL），下一节将讨论这一内容。[333]

1936年，电视转播慕尼黑奥运会。

- HDL写的模型并通过编译指令访问：这与前一种方法在概念上相似，因为也涉及一个用标准HDL语言描述断言/属性的模型库。但是，与前面在主设计代码中直接对这些模型进行实例化的方法不同的是，这里使用编译指令来指定模型。0-In Design Automation（www.0-In.com）公司的Checker-Ware®模型库就是这种技术的一个很好的例子，例如，设计中可能包含以下

Verilog代码：

```
reg [5:0] STATE_VAR; // 0in one_hot
```

左边声明了一个6位的寄存器STATE_VAR，假定这个寄存器要保存一个有限状态机的状态变量。同时，右边的“0in one-hot”是一个编译指令(pragma)。大多数工具会简单地将这个编译指令(pragma)作为注释而忽略掉，但是0-In公司的工具将就此从CheckerWare模型库中调用相应的“one-hot”断言/属性模型。0-In的这种实现方式意味着你不必为断言指定变量、时钟和位宽，这些信息将被自动指定。另外，依据编译指令在代码中的位置，也可以用声明或者进程的方式实现。

19.6.5 静态形式验证和动态形式验证

这一节的内容稍微有一点复杂，让我们一步一步地来理解。首先，可以在仿真环境中使用断言/属性，这种情况下，如果设置了如下的断言/属性，即“信号A和B永远都不能同时活动或为低”，那么若在仿真期间发生了这种异常行为，仿真器将会发出一个警告并把发生的情况记录下来。

仿真器可以覆盖很大的范围，但是需要能够动态产生测试激励的testbench或验证环境。另一个要考虑的问题是，设计中有些部分可能很难通过仿真来验证，因为它们深深地隐藏在设计中，很难从主输入端控制、可能还有些部分同其他状态机或外部的逻辑有大量复杂的交互行为，这也很难控制。

另一方面是静态形式验证。这些工具非常精确，不用任何仿真就可以检查所有的状态空间；缺点是它们通常只能用于设计中的一些小的部分，因为随着具有复杂属性的状态空间的数量指数式地增加，很快就会出现“状态空间爆炸”。比较起来，可以测试断言的逻辑仿真器，覆盖范围很大，但其需要测试激励，并非能覆盖到每一种可能的情况。

为了解决这些问题，出现了一些将两种技术结合起来的方案。例如，可以利用仿真达到一个边界条件，然后自动暂停仿真器并调用静态形式验证工具尽最大可能来评估边界条件（这里，边界条件的一般定义是指在设计中很难遇到或很难达到的功能性条件）。一旦完成对边界条件的评估，控制权将返回给仿真器，继续原来的仿真。这种将仿真和传统静态形式验证相结合的方法称为动态形式验证。

举一个简单的例子，一个FIFO存储器，它的“Full”和“Empty”状态就可以认为是边界条件。达到“Full”状态需要很多时钟周期，最好利用仿真来实现，但是要最大可能地评估与这一边界条件相关的属性，例如当FIFO处于“Full”状态时不应该再写入数据，最好使用静态形式验证技术。

关于OVA，其原始版本利用了Synopsys公司在仿真技术中的实力。

Synopsys公司的一些人后来想扩展OVA以支持形式属性验证，于是他们与Intel公司的一些人合作，凭借这些人利用内部开发的规范断言语言进行形式验证的经验，研究结果即OVA 2.0，包括强大的静态和动态形式验证结构。

0-In公司的工具就是这种动态形式验证技术的又一个好例子，该公司在其CheckerWare模型库中明确地定义了边界条件。仿真期间达到一个边界条件时，仿真器立刻暂停，然后调用静态形式验证工具对边界条件进行详细的分析。

19.6.6 各种语言的总结

如果不细心，有些事情确实会变得很混乱（所以我们应该细心）。首先从Vera[®]语言开始，这一语言最早起源于20世纪90年代初期Sun Microsystems公司做的工作，后来于90年代中期交给了Systems Science公司，1998年该公司被Synopsys公司收购。

Vera本质上是一个完整的验证环境，与本章前面介绍的*e*语言验证环境相似但可能没有那么复杂。Vera语言将testbench特征和基于断言的能力封装起来，Synopsys公司将其提升为一个独立的产品，也可以和Synopsys逻辑仿真器集成在一起。后来，由于形势的要求，Synopsys公司开放了Vera，提出OpenVeraTM和OpenVera Assertions（OVA）的概念，从此第三方工具也可以使用这一语言。

在此期间，SystemVerilog语言中首次增加了assert语句。同时，由于人们对形式验证技术的兴趣日渐增大，Accellera标准委员会开始寻找一种可作为工业标准的形式验证语言，评估了许多语言之后（包括OVA），委员会于2002年最终选择了IBM公司的Sugar语言。竞争更加有趣，Synopsys公司后来将OVA语言捐赠给了负责SystemVerilog的Accellera委员会，而这个委员会和评估形式属性语言的委员会是不同的。

[336]

1937年，英国大学生阿兰·图灵（Alan Turing）发明了一种理论计算机，即图灵机。

然而，另一个负责开放验证库（OVL）的Accellera委员会宣布解散，开放验证库是指一个断言/属性模型库，其中的验证模型有VHDL和Verilog 2K1两种类型可以利用。

这样，现在有assert语句的语言有VHDL、SystemVerilog、OVL（模型库）、OVA（断言语言）以及属性规范语言（property specification language, PSL），其中的属性规范语言是IBM Sugar语言的Accellera版本（如图19-16所示）^①。PSL语

^① 不要犯了将“PSL/Sugar”看作单一/组合语言的普遍错误。PSL是PSL，而Sugar是Sugar，两者不是一个事物。PSL是Accellera标准，而Sugar是IBM内部使用的语言。

言的优点是，它可以独立被使用，而不依赖于描述设计功能所用的语言；缺点是它看起来不像任何设计工程师熟悉的硬件描述语言，例如VHDL、Verilog和C/C++等等。也有人在讨论将PSL发展为各种变体，例如VHDL PSL、Verilog PSL、SystemC PSL等，这些变体的语法会有所不同以便与目标语言匹配，但是它们的语义却是相同的。

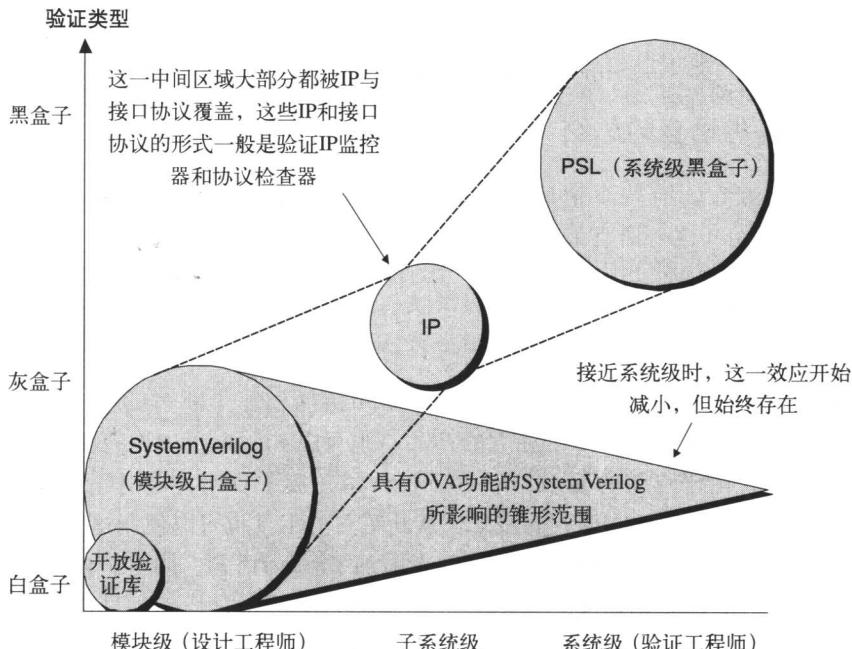


图19-16 各种语言在当前环境和远期环境中的地位

重要的是，图19-16只是反映了一种观点，并不是所有人都同意这种观点。有些人认为这是对极度混乱的状况所做的一次极好的总结；而有些人则认为这只不过是一个粗略的简化，甚至是一堆废话。

19.7 混合设计

19.7.1 HDL语言到C语言的转换

正如第11章所讨论的，在更高抽象级上输入设计（例如C/C++）具有了越来越大的推动力量。高级（行为级或算法级）C/C++模型除了能方便结构探测外，其仿真速度要比相应的HDL/RTL模型快成百上千倍。

前面已经说过，许多设计工程师仍然喜欢在RTL级进行设计，问题是，你对整个SoC进行仿真，而这个SoC中的嵌入式处理器核、存储器、周边设备和其他逻

辑都是RTL级描述，如果能达到几个赫兹的仿真速度也算幸运了，也就是说，从实际时间看，每秒钟只能仿真几个系统主时钟的周期。

为了解决这个问题，一些EDA公司开始提出方法，可以将原来的“RTL黄金模型”翻译成具有更快仿真速度的模型，仿真速度能达到几千赫兹^①。这种速度足可以满足在硬件描述上运行几毫秒实际时间的软件；也可以测试关键的底层软件，例如驱动程序、诊断程序和固件，能方便系统的验证，并且比传统的方法要快得多。

1937年，脉冲编码调制为数字无线电发射指明了道路。

19.7.2 代码覆盖率

几年前，代码覆盖率工具还是由第三方EDA厂商提供的专家级项目。但现在所有大型EDA厂商都认识到了这种能力的重要性，因此已经将代码覆盖率检查功能集成到了各自的验证（仿真）环境中，不过这些环境实现的特征集有所不同。

到目前为止，已经有多种不同的代码覆盖率，以复杂程度为顺序简短概括如下。

- 基本代码覆盖率 (basic code coverage)：只是代码行覆盖率，即源代码中每一行代码执行次数。
- 分支覆盖率 (branch coverage)：条件语句，例如if-then-else，执行then路径共多少次，执行else路径共多少次。
- 条件覆盖率 (condition coverage)：类似于“if (a OR b== TRUE) then”这样的语句。此时，我们感兴趣的是因为变量a为真而导致then路径被执行的次数和因为变量b为真而被执行次数的比较。
- 表达式覆盖率 (expression coverage)：类似于“a = (b AND c) OR !d”的表达式。此时，我们感兴趣的是分析表达式以得到所有可能的输入组合，也要分析出哪种组合触发了输出的一个变化以及哪个变量从来没有被测试到。
- 状态覆盖率 (state coverage)：指分析状态机以得出哪种状态被访问，哪种状态被忽略以及状态之间哪些守护条件和路径被采用，哪些没有被采用等。这类信息也可以从代码行覆盖率中得到，但必须逐行去阅读（这里有意使用双关语）。

^①一个有趣的解决方案是Tenison Technology Ltd. (www.tenison.com) 公司出品的VTOCTM (Verilog-to-C) 翻译器。另一个是来自Carbon Design Systems Inc. (www.carbondesignsystems.com) 公司的SPEEDCompilerTM和DesignPlayerTM。

1938年，美国人Claude E. Shannon以自己在麻省理工学院的硕士论文为基础发表了一篇文章，阐明了怎样利用布尔代数来设计数字电路。

- 功能覆盖率 (functional coverage)：指分析哪些种类的事务级事件和这些事件的哪些组合与排列已经被执行。
- 断言/属性覆盖率 (assertion/property coverage)：一种验证环境，能够收集、组织各种验证引擎给出的结果，并使该结果能用于分析，这些验证引擎包括仿真驱动的、静态的和动态的基于断言/属性的验证引擎。

这些形式的覆盖率实际上可以分为两大阵营：规范级覆盖率和实现级覆盖率。

在本章中，规范级覆盖率是对高层次功能或者宏观结构定义作验证时使用的一个衡量表征，这一概念包括设计中的I/O行为，设计能处理的事务类型（包含不同事务类型之间的关系）以及必需的数据转换等。比较起来，实现级覆盖率则是对具体实现中的微观结构作验证时使用的一个衡量表征，这涉及在RTL代码中插入能产生某些特定边界条件的描述，例如，FIFO的深度以及FIFO处于“几乎满”和“满”时的边界条件。这些具体实现中的细节在规范级几乎是不可见的。

19.7.3 性能分析

现代验证环境中最后一个重要的特征是执行性能分析 (performance analysis) 的能力。这是指一些方法，它们可以精确地分析和报告出仿真器把时间用在了哪些地方，这样用户就可以将注意力集中到设计中活动频繁的区域，对于最终的系统性能来讲，这可能会带来非常大的收益。

1938年，Walter Schottky发现了半导体带结构中存在孔洞，并且解释了金属/半导体界面的整流现象。

第 20 章 选择合适的器件

20.1 丰富的选择

如果我们所面临的选择更少一些，生活中的许多方面都将会更加简单。例如，订购一份看似简单的美式周日早餐，包括鸡蛋、熏肉、炸土豆和烤面包等，可能会花不少的时间，因为有太多的选项可供选择。

首先，女服务员会问你怎样烹制鸡蛋，是向阳面朝上、嫩一点、半生熟还是老一点，是摊成鸡蛋饼、荷包蛋、煮鸡蛋，还是做成煎蛋卷。接着，还会问你要美式烤肉还是加拿大式烤肉；薯饼上要不要加些洋葱、番茄酱、奶酪、火腿、红辣椒，或者加其中的几样进行调味；烤面包片是要用白面包、黑麦面包、全麦面包，还是要用石磨面粉和酵母……

让人害怕的事情是选早午餐（即早餐和午餐合二为一）的复杂，就像选择FPGA一样，因为各个厂商都有许多产品系列可供选择。同一个厂商的不同产品线和系列也有交叠，不同厂商的产品线和系列既有交叠，同时又具有不同的特征和性能，而且情况总是在变化，这种变化几乎每天都会发生。

20.2 要是有选型工具就好了

开始之前，值得说明的一点是，在FPGA设计中，设计的大小并不是关键，真正需要的只是根据设计需求选择合适的FPGA，例如I/O引脚的数量、可用的逻辑资源、是否有某些专门的功能模块等。343

另一个需考虑的问题是，是否已经在使用某个FPGA厂商的产品系列，或者是否是第一次做FPGA设计。如果同某个厂商已经有了一段时间的交往，并且熟悉其元器件、工具和设计流程，那么通常会继续使用原有厂商的工具和器件，除非有极其重要的原因要改变。

但为了方便其余部分讨论，我们假定从零开始，和任何FPGA厂商都没有特别的联系。这时，要为某个设计选择一个合适的器件就成了一项艰巨的任务。

要熟悉不同FPGA厂商的各种产品系列，包括结构、资源和性能等，不是一件容易的事情，需要消耗相当多的时间和精力。在现实世界中，产品上市时间的压力非常大，设计工程师在选定厂商、器件系列和器件前一般只有时间来做高层次评估。这种情况下，选定的FPGA几乎肯定不是对设计最合适的器件，但世界本来

就是这样的。

有一个选择，可以利用某种FPGA选型向导程序（最好基于网络）来选择FPGA，效果非常好。使用这种向导可以选择某个特定的厂商，也可以选择一组厂商，或者在所有厂商中进行搜索。

为了完成一个基本的设计，向导应该提示用户输入一些估计值，例如ASIC等效门或FPGA系统门（假定等效门和系统门的定义已经完备，可参考第4章）。向导还应该提示用户输入一些细节信息，如I/O引脚需求、I/O接口技术和可接受的封装选项等等。

对于更高级的设计，向导还应该提示用户作一些特殊的选择，如吉比特收发器或者嵌入式功能，例如乘法器、加法器、MAC、RAM（包括分布式RAM和RAM块）等。如果需要用嵌入式处理器核（软核或硬核）以及一些相关的外围设备，向导也应该允许用户来指定。

1939年，Messers Bay和Szigeti发明发光二极管。

最后，如果向导能将有关IP的需求提示给用户将会很好（这听起来简直是做梦，既然是梦，那我们就将它做得大一点）。完成选择后，单击一下“Go”按钮，就会生成一份报告，详细地说明了筛选出来的主要器件及其性能和成本。

回到现实世界中，现在还没有这种工具^①，所以我们就必须手工来做所有评估，但这样的效果并不理想……当然，开发这类应用程序不是件容易的事情，而且维护起来也昂贵且耗时，但可以肯定的是，系统设计工程师将会很高兴为这类服务付费，因此人人都应该勇于接受挑战，积极实践设计出这样的工具软件。

20.3 工艺

首先要做的选择之一是使用什么工艺的FPGA，主要有如下几种选择。

- 基于SRAM工艺：尽管这种工艺的FPGA很灵活，但需要一个外部配置器件，系统第一次加电后要经过几秒钟才能配置成功。早期的这些器件有很大的瞬时启动电流，因而对电源有较高的要求，但是新一代的器件已经解决了这个问题。这一选择最主要的优点是使用标准的CMOS工艺，不需要任何复杂的工序，这就意味着基于SRAM工艺的FPGA总是处于主流技术的前沿。
- 基于反熔丝技术：许多人考虑过使用反熔丝技术为IP设计提供最高的安全性保障，除此之外，反熔丝技术还具有低功耗、上电即可用、不需要外部配置器件（这就节约了电路板的成本、空间和重量）的优点。反熔丝器件

^① 有些类似的工具可以帮助选择PLD器件，但它们都是些较为简单的工具。

比其他工艺的器件更能抵抗辐射，因此受到航空类应用的青睐，但是这种技术是一次可编程的（OTP），不适于作原型设计。反熔丝器件一般比最主流技术要落后一代甚至几代，因为这些器件比标准CMOS器件多需要几个额外的工序。

- 基于FLASH工艺：尽管人们认为这类器件比基于SRAM的器件具有更高的安全性，但是在IP设计上比不上反熔丝器件。基于FLASH的FPGA不需要外部配置器件，但是在必要的情况下仍然可以在系统中进行重新配置。与反熔丝器件相同，基于FLASH的器件也具有上电即可用的优点，但一般也要比主流技术落后一代到几代，比标准CMOS器件多需要几个额外的工序。另外，这些器件的逻辑门（或系统门）数量一般比与其对应的基于SRAM的器件要少很多。

20.4 基本资源和封装

一旦决定了要使用何种工艺，就需要了解哪些器件可以满足资源和封装需求。346 在使用内核资源的情况下，大多数设计都会出现引脚不足的现象，但一般只有在含有复杂算法的设计中才会出现逻辑资源不足的情况，例如颜色空间转换。不管是哪种类型的设计，都需要决定使用多少I/O引脚，大约需要多少基本逻辑元件（LUT和寄存器）。

第4章已经讨论过，LUT、寄存器和相关的逻辑组合起来称为逻辑元件（logic element, LE），也可以称为逻辑单元（logic cell, LC）。用这些术语考虑问题，一般要比用那些高级结构（例如slices、可配置逻辑块（CLB）或逻辑阵列块（LAB））更有用，因为这些更复杂结构的定义对于不同的器件族可能有所不同。

接下来，需要判断哪些器件包含足够的时钟域和相应的PLL、DLL或者数字时钟管理单元（DCM）。

最后，如果还有些特殊的封装要求，注意一定要保证所选的FPGA器件族中有可用的封装形式。虽然这一点看起来很明显，但你不能保证以前就没有人在这上面犯过错误。

20.5 通用I/O接口

接下来要考虑的是哪些器件具有可配置通用I/O模块，这些I/O模块要支持必要的信号标准和终端工艺，以便和电路板上其他器件进行接口连接。

我们假设又回到了设计过程的开始阶段，系统结构工程师选择了一个或几个电路板上要使用的I/O标准。理想情况下，会找到一个FPGA能支持这个标准，并且具有所有需要的功能；如果没有找到，还有以下几个选择：

- 最初选择的FPGA不能提供任何必需的功能，这时可以考虑使用其他FPGA

族的器件，也可以使用其他厂商的器件。

- 最初选择的FPGA不能提供部分必需的功能，这时可以考虑使用一些外部的桥接设备，但这样做增加了成本并消耗电路板上的可用面积。另一种方法是，与系统设计团队其他成员一起协商，改变电路板的总体结构，但是如果系统设计已经到了任何很重要的阶段，这种做法是非常昂贵的。

20.6 嵌入式乘法器、RAM等

在设计的某些阶段，往往需要评估一下设计中需要的分布式RAM和嵌入式块RAM的数量（以及块RAM的要求宽度和深度）。

同样，还要考虑需要多少专门的嵌入式功能模块（及其宽度和性能），例如乘法器和加法器。对于以DSP为中心的设计，有些FPGA还包含了嵌入式MAC模块，这些模块对于此类设计特别有用，可能会左右你选择器件的决定。

20.7 嵌入式处理器核

如果想在设计中使用一个嵌入式处理器核，就需要确定软核是不是能够满足设计要求（这种核可以在多种器件族上实现），或者如果选用硬核，就要考虑该硬核是否通用（也可参考第13章中的讨论）。

对于软核，可以使用FPGA厂商提供的选择，这样的话，就可能固定在这个FPGA厂商上，因此在作决定以前需要仔细地评估各种替代方案。另一种方法是使用第三方软核方案，可以使用多个厂商的器件实现^①。

如果要使用硬核，除了固定使用某个厂商的硬核外，很少有其他选择。可能影响决策过程的一个考虑因素是已有的使用各种处理器的经验，例如，设计基于PowerPC的系统时的经验。这种情况下，为了保护在PowerPC设计工具和设计流程中的投入（以及使用这些工具和流程的经验和知识），用户可能会使用Xilinx公司的FPGA，因为这些器件支持PowerPC。如果熟悉ARM或MIPS处理器，则分别选择Altera公司或者QuickLogic公司的FPGA器件。

20.8 吉比特I/O能力

如果系统需要使用吉比特收发器，那么需要考虑的是器件中所含收发器的个数以及系统结构设计人员在电路板级选择的传输标准（可参考第21章）。

20.9 可用的IP

每一个FPGA厂商都有一个IP开发部门，各个FPGA厂商的IP部门之间经常有

^① 这种方案的一个例子是Altium Ltd.公司（www.altium.com）出品的Nexar软件，第13章中有介绍。

些重叠，但是那些比较复杂的IP模块却只有某些厂商提供，这可能会影响对器件的选择。

另一种选择，可以从第三方厂商购买IP。这些IP可以用于不同厂商的多种FPGA，或者只能用于某些厂商的器件（或这些厂商的某些器件）。

还有一点：我们普遍认为IP是为了实现某些硬件功能，但是有些IP却是软件程序^①。例如，通信功能既可以用FPGA中的硬件资源来实现，也可以用嵌入式处理器运行软件来实现。后一种情况下，可以从第三方购买软件程序，这基本上是得到了软件IP。

20.10 速度等级

一旦为设计选定了某种FPGA器件，最后要决定的就是器件的速度等级。FPGA传统的定价模式使器件的性能（速度等级）成为决定器件成本的主要因素。

一般来讲，提高一个速度等级将带来12%到15%的性能提升，但是器件的成本却增加了20%到30%。相反，如果利用设计结构来将性能提升12%到15%（通过增加额外的流水线），那么就可以降低速度等级，从而节约20%到30%的成本。

如果你只是想用一个器件来进行原型设计，这对你可能不是重要的因素；但是，如果你打算购买大量的这类器件，那么就应该认真考虑使用能满足设计要求的最低速度等级。

问题是，要利用结构来提升性能必须对各种可能的实现方案执行一系列的假设性评估，因此需要修改RTL代码并重新进行验证，这是一项困难而且费时的工作。这种评估可能包括使用并行和串行方式分别执行某种操作、设计的流水线部分和非流水线部分、资源共享等等。所以设计团队只能执行有限几次这种评估，最终得到一个接近于最佳的实现方案。

第11章讨论过，一个替换方案是使用完全无时序问题的基于C/C++的设计流程，这种流程有一个C/C++分析和综合引擎，可以让用户对设计中的微观结构进行替换并根据面积和速度性能评估这些结构的效果。这种流程有利于提升设计的性能，需要的话可以用更慢的速度等级来实现设计。

1945年，科幻小说作家Arthur C. Clark构想出地球同步通信卫星。

1946年，美国的John William Mauchly和J. Presper Eckert带领团队建成了
一台通用电子计算机ENIAC。

1946年，车载无线电话连接到了普通电话网络。

1947年，美国物理学家William Shockley、Walter Brattain和John Bardeen于
12月23日制作了世界上第一个点接触锗晶体管。

^① 还有一种验证IP，在第19章讨论过了。

20.11 轻松的注解

一个朋友Tom Dillon说过，用上面这些复杂的东西吓唬完读者后，我应该以一个轻松的注解来结束本章。所以，比较乐观的情况是设计团队一旦选择了一个FPGA厂商并且熟悉了该厂商某个产品系列，那么该团队往往会在相当长的时间内
351 固定使用该系列产品，因为这样可以使后续工程中的器件选择变得更加容易。

第21章 吉比特收发器

21.1 引言

在第4章中已经讨论过，要在同一个电路板上的两个或者更多器件之间传送大量数据，传统的方法是使用总线，也就是传输同类数据并执行同种功能的信号的组合，如图21-1所示。

1975年前后的早期微处理器系统使用8位总线传输数据。随着传输数据量的增加和对传输速度的要求越来越高，总线宽度也相应地增加到了16位、32位、64位等，但是随之而来的问题是，芯片的引脚被大量占用，芯片之间的互连需要大量的线路。随着电路板复杂程度的增加，对这些线路进行布线以便得到相同的长度和阻抗也变成一件越来越痛苦的事情。此外，在处理大量总线线路时，对信号完整性（SI）的管理也越来越困难，例如噪声和串扰效应的敏感性。

1948年，美国使用飞机向九个州转播电视信号。

由于这个原因，今天的高端FPGA包含了专门的吉比特收发器（gigabit transceiver）模块。这些高速串行接口使用一对差分信号来发送数据，即TX，另一对差分信号则用来接收数据，即RX，如图21-2所示。

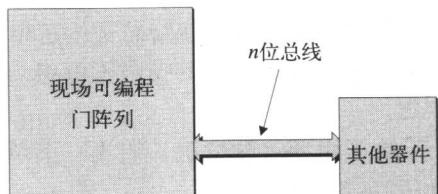


图21-1 器件之间使用总线进行通信

353

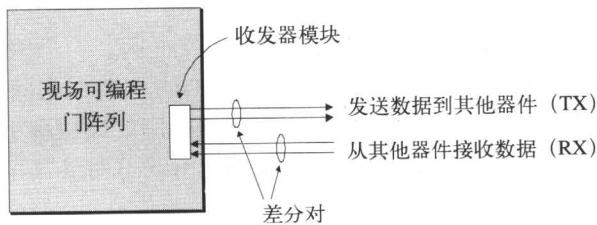


图21-2 各器件之间使用高速收发器进行通信

需要注意的是，传统的数据总线上可以挂接许多设备，而这些高速串行接口却是点对点连接，因此每一个收发器只能与其他设备中的一个收发器通信。

写作本书时，使用这些高速串行接口的设计还非常少（可能只有很小比例的设计刚开始使用），但是这一数字有望在未来几年内急剧增加。使用这些吉比特收发器有一定的难度，不过FPGA厂商都会提供详细的用户指导和对特有技术的应用说明。

这些接口存在的一个问题是，有太多的细节情况需要用户考虑。但是鉴于本书的目的，我们在这里只介绍主要的概念，给出粗略但比较充分的信息，以便人们使用这些接口时更加注意。

21.2 差分对

差分对是指两条线路总是传送相反的逻辑电平，使用差分对的原因是这些信号对外界干扰源产生的噪声不敏感，例如无线电干扰和这些线路附近的其他信号变化。
[354]

为了说明这种情况，假设一条单线和一个差分对都受到了同样的噪声干扰，如图21-3所示。

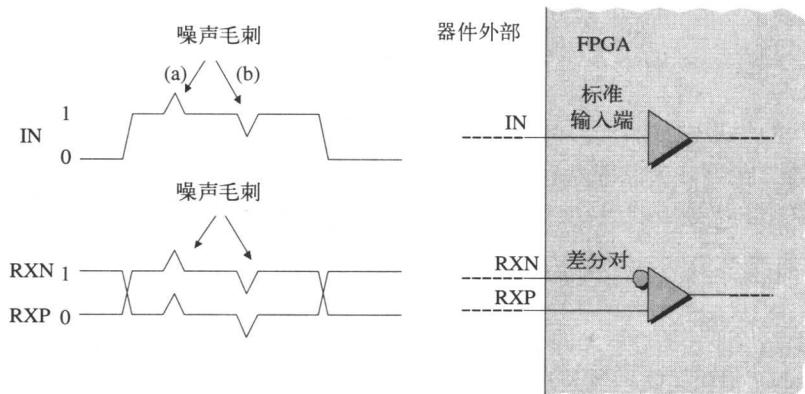


图21-3 各器件之间使用高速收发器进行通信

使用标准输入时，名为IN的引脚连接到一个缓冲门。这个例子中，我们对第一个噪声毛刺(a)不感兴趣，但是第二个毛刺(b)可能引起问题，如果这个毛刺越过了缓冲门的输入转换阈值，那么它将在缓冲门的输出端引起一个尖脉冲，依次进行下去，这个尖脉冲可能在FPGA内部引起一些不期望的行为，例如寄存器载入错误值。

几年前，当逻辑0和逻辑1之间的电压差为5V时，这些问题还不算严重，因为即使噪声毛刺达到1V，也不会引起任何问题。但是，时间永远不会停止，技术时时刻刻都在进步，现在的逻辑0和逻辑1之间的差异依据不同的I/O标准，可能只有1.8V，1.5V甚至更低，这种情况下，即使远小于1V的噪声毛刺也可能是破坏

性的^①。

现在来看看差分对，其信号由发送设备中的特殊类型驱动门产生（图21-4所示）。对于纯化论者，应该说明的是，差分对中的活动（正）信号（图21-3和图21-4中的RXP和TXP）通常画在上面，而反信号（RXN和TXN）则画在下面并在缓冲门符号上加一个小圆圈表示反相。这样画的原因是为了使图21-3中的RXP信号与IN信号相协调，从而使该图更容易理解。

记住，差分对中的两个信号总是传送互补的逻辑值。所以，当图21-3中的RXP为逻辑0时，RXN一定是逻辑1，反之亦然。正如图21-3中所示，关键的一点是，差分对中的两条线路布线时靠得非常近，因此任何噪声对它们的影响都是相同的。接收端的缓冲门基本上只对两个信号之间的差异有所反映，也就是说差分对比单线连接对噪声更不敏感。

结果是，假定电路板经过适当的设计，那么这些收发器就可以工作在极高的速度下。而且，每个FPGA中可能包含许多这样的收发器模块，几个收发器也可以组合在一起提供更高的数据传输速率。

21.3 多种多样的标准

当然，如果没有各种各样的这类标准，电子学也就不是电子学了，每一个标准都定义了一套完整的内容，从高层次协议一直到物理层（PHY）。下面列出一些比较常见的标准：

- Fibre Channel
- InifiniBand®
- PCI Express（Intel公司发起和推动）
- RapidIO™
- SkyRail™（来自Mindspeed Technologies™）
- 10GB Ethernet

对于其中的某些标准，如PCI Express和Sky-Rail，器件厂商可能使用了相同的基本概念，但是这些厂商又各自用自己的名称和术语将这些概念注册了商标，这使得情况更加复杂。而且，实现这些标准也需要使用多个收发器模块（也可参考21.6节）。

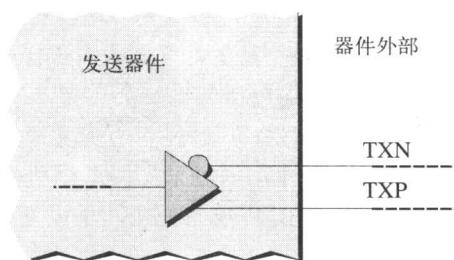


图21-4 产生差分对

① 使用差分对时，有一个标准规定，逻辑0和逻辑1之间的电压差只有0.175V！

假如我们打算设计一个电路板，想使用一些高速串行接口，此时，系统结构设计人员会决定使用哪一个标准。FPGA中的各个吉比特收发器一般可以配置为支持多个标准，但通常不会支持所有标准。这就意味着系统设计人员或者选择一个拟使用的FPGA支持的标准，或者选择一个支持拟采用的接口标准的FPGA。

357 如果系统中包括创建一个或者多个ASIC的任务，我们当然可以从头开始实现选择的接口标准，但更可能的情况是从第三方购买适当的IP核。然而，那些不用定制的标准器件，一般只支持上面所提到的标准中的一个或一个子集，这种情况下，就可以使用FPGA来充当两个或更多标准之间的接口（如图21-5所示）。

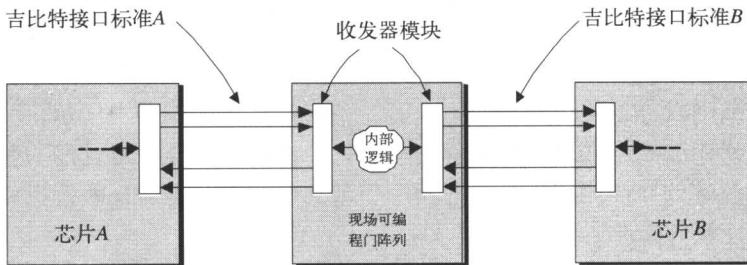


图21-5 用FPGA作为多个标准之间的接口

21.4 8bit/10bit编码等

人们谈论信号传输能达到每秒数吉比特的速率时，却忽略了一个问题，那就是电路板和线路吸收了信号中大量的高频成分，这就导致接收器只能收到被极大地衰减了的信号。

358 但是，人们并没有认识到这一点。来看一个例子。首先，考虑一个理想信号在逻辑0和逻辑1之间交替变化，如图

21-6所示。

有经验的工程师立刻会发现图中的一些错误。例如，发射芯片产生的信号是理想的方波，而在现实世界里，这种方波信号实际上有重要的模拟特征。在实际应用中，信号达到这些频率时，从发射芯片出来的信号很差，进入到接收芯片时会变得更差。而且，接收器收到的信号可能还会相对图21-6所示的情况有些相位偏移，这里我们将两个信号排列整齐是为了看清楚发射端和接收端哪些位彼此对应。

正如图中所示，接收端收到的信号已经严重衰减了，但是仍然能够在中心值附近上下振荡，这样接收器就可以进行判断并从中提取出有用的信息。这里我们

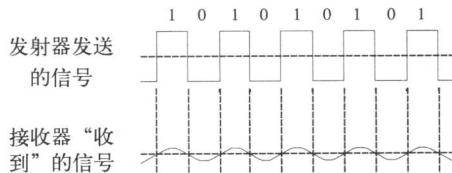


图21-6 一个理想信号

修改一下发射序列，在其开始处发射三个连续的逻辑1，看看会发生什么情况（如图21-7所示）。

这种情况下（记住，这是一个比较极端的方案，纯粹是为了提供一个讨论例子），接收器收到的信号会在前三个bit期间连续上升，这就导致信号超过了中心值，这意味着当发射序列最终回到其原始的010101时，接收器实际上会继续将接收到的信号作为逻辑1对待。

在数据通信领域，单独的二进制数（有时候指二进制数组成的字）被称为符号（symbol）。符号的“拖尾效应”，也就是说一个符号的能量影响了后面的符号，由此造成接收端错误地解释了收到的信号，这种现象称为符号间干扰（intersymbol interference, ISI）。

经常听到的另一个与此有关的术语是连续相同数值（consecutive identical digits, CID），这是指类似于图21-7中所示的连续出现三个逻辑1的情况。正如前面说明的，图21-7中的例子是极其悲观的，实际上，只要能保证不在同一行中发送五个以上相同比特就可以了。这样，高速收发器模块就必须包含某些形式的编码运算，例如8bit/10bit（简写为8b/10b或者8B/10B）标准，这个标准是指每8位的数据块被扩展成10位以保证在同一行中不会发送超过五个逻辑0或逻辑1。此外，这个标准也保证了信号在20位（2个块）以上总是DC平衡的，也就是说中心值上下两部分能量是完全相同的。

除了8B/10B标准外，还有些替代方案，包括64B/66B（或64b/66b）标准和SONET Scrambling标准。后者中的“scrambling”部分来源于如下事实：把数据流中的0和1完全打乱以防止出现长的连续全0或全1的字符串。这一点与这里讨论的所有方案是相同的。

到这里，最后一点也就不值一提了。除了解决图21-7中所示的问题外，使用这些编码方案的主要原因之一是为了减轻从数据流中恢复时钟信号的任务（也可参考21.8节的讨论）。

21.5 深入收发器模块内部

现在来介绍一下8B/10B编码的概念，我们处在一个较好的位置可以近距离观察组成收发器模块的主要元素，如图21-8所示。

图中所示的是一种非常简单的表示方法，忽略了许多的位（bit）和片（piece），但是涵盖了我们比较感兴趣的内容。关于“预加重（preemphasis）”和“均衡化（equalization）”的注释，将在本章后面的部分中介绍。

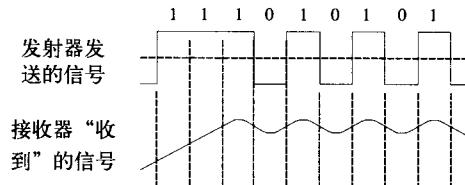


图21-7 发射一系列相同bit值的结果

359

360

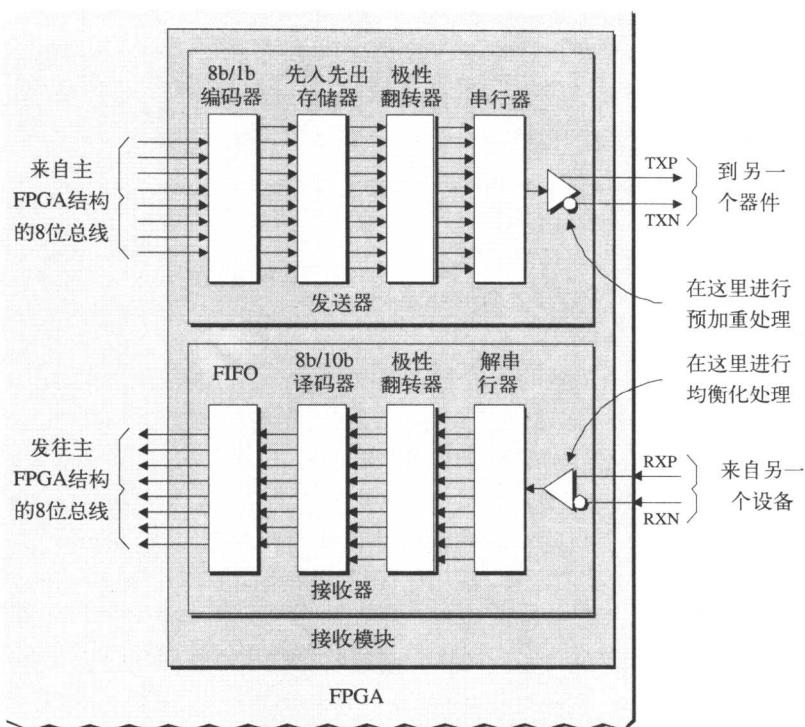


图21-8 组成收发器模块的主要元素

在发射器一侧，数据经由8位总线由FPGA中的用户定义逻辑发送到收发器，

361 然后经过8B/10B编码器后写入FIFO中，这个FIFO主要是为了在连续收到很多数据时临时存储数据。FIFO的输出数据进入一个极性翻转器，这个模块可以将数据原封不动传送过去也可以将数据的每一位进行翻转，即从0变为1，或者相反（只有接收我们发送数据的设备需要翻转数据时才进行极性翻转）。然后，极性翻转器的输出数据进入串行器，即把并行输入数据转换为串行位流，然后由一个专门的输出驱动/缓冲门对这一串行位流进行处理，产生一对差分信号。

同样，在接收端，从差分对上过来的串行数据流通过一个专门的输入缓冲门进入解串行器，就是将串行数据转换为10位的字。这些字再进入极性翻转器，与发送端的极性翻转器相同，也可以原样传送数据或者按位进行翻转（但是只有当接收到的数据是翻转数据时才进行翻转）。极性翻转器的输出由8B/10B解码器进行解码，得到8位数据并通过FIFO进入FPGA中，然后就可以由设计工程师实现的任何逻辑来处理。

需要注意的是，根据使用的FPGA工艺不同，有些收发器模块可能支持多种编码标准，例如8B/10B、64B/66B、SONET Scrambling等。有些则可能只支持一种

8B/10B标准，不过在这种情况下可以不管这些模块，必要的话还可以在FPGA内设计自己的编码方案。

21.6 组合多个收发器

波特率是指通信连接中的信号每秒钟（可以）变化的次数。依据所用编码技术的不同，通信连接每波特可以发送一个数据位，也可以发送更少或更多的数据位。[362]

写本书时，每个收发器当前的运行状态可以达到以每秒3.125 Gbit (Gbps)^①的速度收发8B/10B编码或者类似编码的数据。如果忽略8B/10B编码额外增加的位数，实际的速率相当于2.5 Gbps（换句话说，就是将3.125 Gbps除以10，再乘以8，就等于实际的速率2.5 Gbps）。

按照定义，像10GB以太网这样的标准就应该要求传输速率达到10Gbps。所以还有些附加的标准，例如10Gbit attachment unit interface (XAUI)，这一标准定义了怎样在每个方向上用四个差分对实现10 Gbps的传输速率，如图21-9所示。[363]

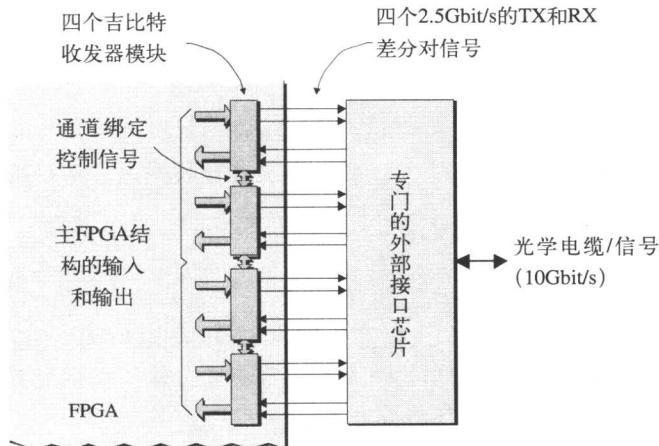


图21-9 组合多个收发器模块

1950年，美国物理学家William Shockley发明了第一个双极结型晶体管。

这种情况下，四个收发器模块被专门的通道绑定控制信号连接在一起，这样每个模块才能明确要做什么及什么时候做。

在将来的某个阶段（主要是看电路板级高速串行接口技术采用的速率），现在

^① 检查3.175Gbit/s的波特率时，8B/10B编码相关的数据将会变得过高，这样我们只能选择其他编码，如64B/66B。

这些由外部接口芯片实现的功能很可能被集成到FPGA芯片内，那时FPGA就有能力直接发送和接收光学信号了（参考第26章）。

21.7 可配置资源

FPGA内部的嵌入式Gbit收发器模块一般都有很多可配置特征。不同厂商和器件族可能支持不同的特征组合，其中一些主要的特征说明如下。

21.7.1 逗号检测

8B/10B编码和其他一些编码方式都包含某种专门的逗号字符。这是一些空字符，目的是为了让连接保持“活动”状态，或者通过告知接收器即将有数据要发送，要接收器做好接收准备，从而初始化一次数据传输。

另外，这些高速串行接口本质上都是异步的，因此数据信号中都嵌入了时钟信息（可参考21.8节）。因此，当一个收发器模块要初始化一次传输时，首先要发送一系列逗号字符（有几百位），以便线路另一端的接收器能取得同步。对多个位流进行校准时也可能使用逗号字符。

364

1950年，Maurice Karnaugh发明卡诺图，并很快成为逻辑设计人员最主要的工具之一。

重要的是，有些收发器模块允许要发送或者接收的逗号字符被配置成任意的10位数据，这样收发器就可以支持多种通信协议。

21.7.2 差分输出摆幅

差分输出摆幅是指逻辑0和逻辑1之间的峰—峰值电压差，不同的标准支持不同的差分输出摆幅。因此收发器模块一般允许将差分输出摆幅配置为某个范围的值，以便同各种串行系统的电压标准兼容。

21.7.3 片内末端电阻

高速串行接口支持的高速传输速率意味着使用外部末端电阻可能会引起信号的不连续，因此一般推荐使用FPGA器件提供的片内末端电阻。这些片内末端电阻的阻值一般是可以配置的（通常设定为 50Ω 或 75Ω ），以便支持各种不同的接口标准和电路板环境。

21.7.4 预加重

正如图21-6中所示的情况，信号在穿过高速串行接口后，到达接收器时已经

严重失真（衰减），因为电路板和线路吸收了信号的大部分高频成分，只留下一些低频成分（变化较慢的频率）。

预加重技术可以减轻这种效应，这种技术的原理是，将传送的0位流中的第一个0的电压稍微降低，而将1位流中的第一个1的电压稍微抬高（这里所说的“位流”是指一个或多个bit）。使用这种方法，实际上相当于人为地为信号加上了和电路板失真相反的失真（如图21-10所示）。

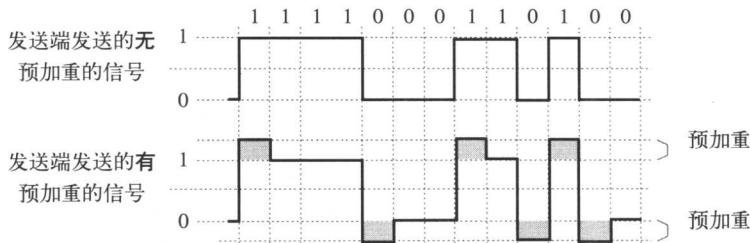


图21-10 应用信号预加重

再说明一下，这个例子显示的是发送芯片产生的理想信号（边缘很陡峭），但在实际应用中，这些信号带有很强的模拟特征。

预加重的量一般是可配置的，以便能适应不同的电路板环境。某种特定高速连接需要的预加重的量是FPGA与其他元件的相对位置（等同于线路长度）、各种电路板属性和所用的高速传输标准的函数。预加重的量可以通过仿真来决定，也可以根据经验确定。

21.7.5 均衡化

这个概念与前面讨论的预加重有一定的联系，只是均衡化是发生在高速接口的接收器一端，如图21-11所示。

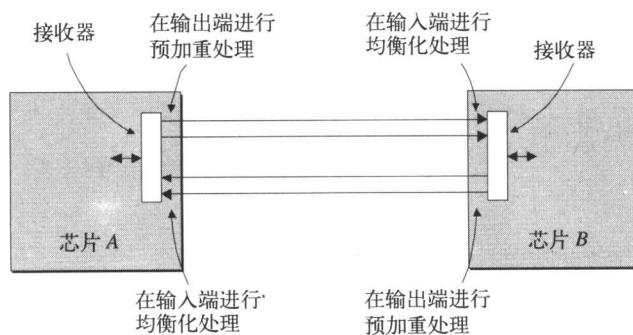


图21-11 应用信号均衡化

均衡化是指一个专门的放大过程，但是对信号的高频成分的放大程度要比对

低频成分大。与预加重相似，这也相当于人为地为信号加上了和电路板失真相反的失真。

[366] 均衡化的量一般也是可配置的，以便能适应不同的电路板环境。对于不同的设计，可以使用预加重或者均衡化，也可以混合使用两种技术。

1952年，美国的John William Mauchly、J. Presper Eckert及他们的团队制作了一台通用电子计算机EDVAC。

1952年，英国雷达专家G.W.A. Dummer首次公开讨论了集成电路的概念。

值得注意的一点是，如果电路板上高速接口的线路确实很长（例如超过40英寸），那么使用外部均衡器件要比内部均衡更可行，因为使用专门的模拟均衡器件比在FPGA内部均衡效果更好。不过，FPGA也在逐步提高这种能力，各个FPGA厂商在技术上彼此超越，因此像片内均衡质量这类因素也可能影响用户对器件的选择。

21.8 时钟恢复、抖动和眼图

21.8.1 时钟恢复

从根本上说，高速串行接口是异步的，数据信号中嵌入了时钟信号。所以，收发器的接收器部分包含了时钟和数据恢复（clock and data recovery, CDR）电路，这一电路可以分开识别输入信号的上升沿和下降沿，并自动提取出代表输入数据速率的时钟。可以想像，如果输入信号在逻辑0和逻辑1之间来回翻转，这一功能就不算什么了，这时的时钟和数据是相同的（如图21-12a所示）。

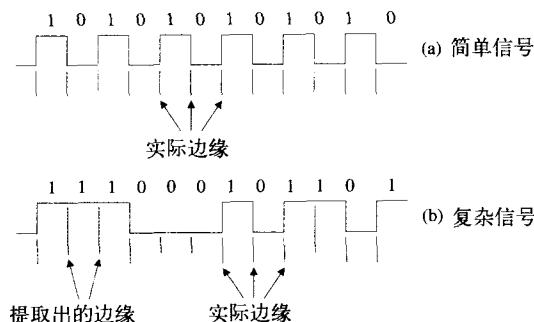


图21-12 恢复时钟信号

但是当信号变得更复杂时，事情就不是这么简单了（如图21-12b所示）。例如，如果输入信号以三个1，后面紧跟着三个0开始，那么时钟恢复电路将作一个初始

的猜测，得到的时钟频率只有真实值的三分之一，但不能因此就认为时钟恢复电路不能正常工作。实际上，当更多的数据（确切地说是转换）到达时，时钟恢复电路将会逐步细化猜测，直到提取出正确的时钟频率。

一旦接收器锁定了时钟，它将利用这一信息对输入数据流进行采样，采样点在每一位的中间，由此可以判断该位是逻辑0还是逻辑1，如图21-13所示。

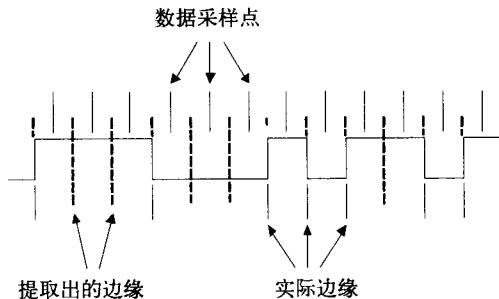


图21-13 对输入信号采样

正如前面提到的，这就是为什么用几百个逗号字符来开始一次数据传输，实际上是为了让接收器锁定到时钟上并准备接收数据。

时钟恢复电路会连续监控信号边缘，不断地修正时钟频率以便适应外界环境引起轻微时钟偏移，这些外界环境包括温度和电压等的变化。

368

21.8.2 抖动和眼图

1954年，制造出第一只硅晶体管。

1956年，美国的John Backus和在IBM的团队提出了第一个被广泛使用的计算机语言FORTRAN。

抖动是指信号实际的转换时间点与理想转换时间点之间的差异。例如，如果我们想获取一个在逻辑0和逻辑1之间振荡的输入信号（如图21-14a、图21-14b所示）并且每一个时钟周期对应的数据与前一个时钟周期的数据有交叠，这时就会出现一些毛刺（如图21-14c至f所示）。

这种毛刺是由多种因素引起的，包括发送器件中的时钟偏移以及前面提到的“符号间干扰”效应（如图21-7所示）。

事实上，我们还可以再深入一步，在概念上将每一个时钟周期对半折叠起来，这样前半个周期的正向0-1-0脉冲就和后半个周期的负向1-0-1脉冲重叠了，如图21-14g所示。

再指出一点，图21-14中所示的波形实际上是不存在的，因为它们的边缘非常

陡峭。在现实世界中，所有的信号都具有模拟的特征，如果看重叠的真实波形，其形状则如图21-15所示。

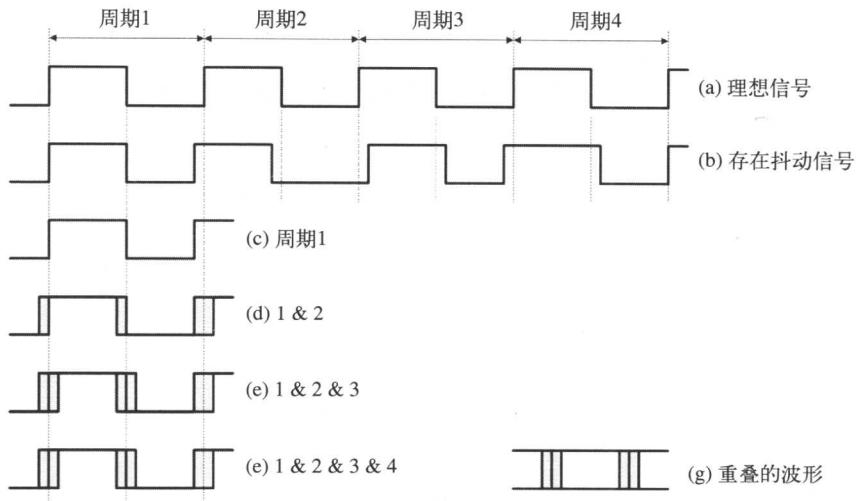


图21-14 抖动

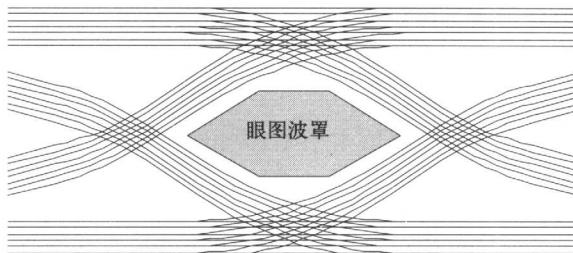


图21-15 眼图和眼图波罩

这是一个中心像人眼睛的图形，所以人们将这个图称为眼图就不足为怪了。随着抖动、衰减和其他失真的增加，眼图中间的区域变得越来越小，因此，有许多规范都定义了一个几何形状，称为眼图波罩（eye mask）。这个“波罩”可能是矩形或六边形，代表数据有效窗口，只要所有的曲线都在“波罩”以外下降，高

速接口就可以工作。

最重要的是，如果打算从这些高速串行通信接口中选择一个来使用，那必须保证你的信号完整性分析工具能够支持眼图这一概念。

1956年，美国的John McCarthy为人工智能应用开发了一种计算机语言LISP。

第 22 章 可重配置计算

22.1 可动态重配置逻辑

基于SRAM的FPGA的出现为电子学领域带来了一种全新的概念：可动态重配置逻辑，也就是说，设计在不脱离上层系统的情况下可以被重新配置。

只是重述一下，FPGA含有大量的可编程逻辑和寄存器，这些资源可以用不同的方式连接在一起实现多种功能，基于SRAM的技术可以让主系统给FPGA器件中下载一些新的配置数据。尽管构成FPGA的所有逻辑门、寄存器和SRAM单元都是在同一块硅衬底的表面上制造出来的，但有时会将FPGA形象地看作两层结构：一层是逻辑门和寄存器；另一层是可编程的SRAM配置单元（如图22-1所示）。

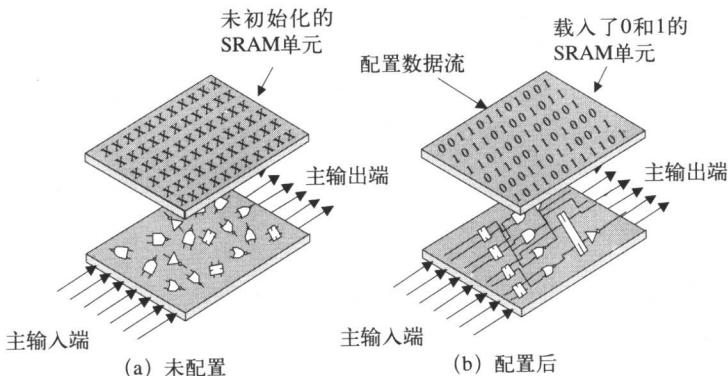


图22-1 可动态重配置逻辑：基于SRAM的FPGA

这些FPGA器件的多功能性开启了通往大量可能性的大门。例如，当系统上电时，可以配置FPGA执行一些系统测试（甚至是自测试）操作，一旦系统测试通过，就可以重新配置FPGA，执行其正常的功能。

22.2 可动态重配置互连线

尽管电路板上单个FPGA的功能能够重新配置这样非常好，但是设计工程师仍有机会去创建可重新配置的板级系统，以便能执行各种各样截然不同的功能。

解决方法是动态地配置器件之间的板级连线，有一类器件专门提供这种功能：现场可编程互连器件（Field-Programmable Interconnect Device, FPID），也可称为

现场可编程互连芯片（Field-Programmable Interconnect Chip, FPIC）^①。这些器件可以将各个逻辑器件连接在一起，其中的连接关系还可以像标准的基于SRAM的FPGA一样动态地进行重新配置。由于每一个FPID可能有1000甚至更多个引脚，因此只有少数这种器件需要放置在电路板上（如图22-2所示）。

有趣的是，这里讨论的概念不仅仅局限于板级实现上，任何技术都有可能用混合方法、多芯片组件和SoC器件来实现。

22.3 可重配置计算

和电子学里的许多概念一样，对可重配置计算（Reconfigurable Computing, RC）这一概念的理解也因人而异。有些人认为，可重配置计算是指专门的微处理器，其指令集可以在空闲时被增强或者修改，但是鉴于本书的目的，我们将可重配置计算理解为一种通用的硬件，例如FPGA，可以配置为执行某种专门的任务，随后也可以按照要求重新配置来执行其他任务。

1958年，美国德州仪器公司的Jack Kilby成功地在一块半导体上制作了多个元器件，这是世界上第一块集成电路。

大多数基于SRAM的FPGA都有一个局限，那就是重新配置时需要较长的时间，因为这些器件在配置时用的是串行数据流或者8位宽的并行数据流。如果是高端器件，其中的SRAM配置单元可能多达数千万之多，对这些单元重新编程需要几秒钟的时间。有些FPGA使用大量的通用I/O引脚充当配置总线来解决这个问题，配置完后再恢复I/O的主功能（见第26章）。另外，一些现场可编程节点阵列（Field-Programmable Node Array, FPNA）也提供了很宽的编程总线（见第23章）。

传统的FPGA架构还有一个局限，想重新配置器件中的某些部分时，必须对整个器件重新编程（第14章中讨论过，一些近期出现的架构允许用户按列对器件重新配置，但这也只能提供粗大粒度的配置）。此外，这些器件在重新配置过程中通常需要停止整个电路板的操作，而且FPGA中所有寄存器的值在配置过程中都无可挽回地丢失了。

1959年，美国提出用于商业应用领域的COBOL计算机语言。

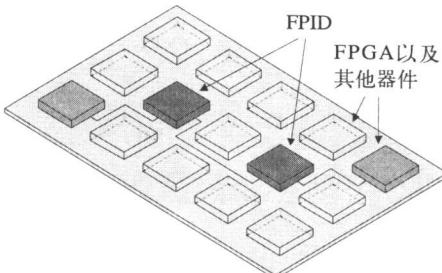


图22-2 可动态重配置互连：
基于SRAM的FPID

374

375

^① FPIC是Aptix公司（www.aptix.com）的注册商标。

为了解决这些问题，Atmel公司（www.atmel.com）于1994年前后提出了一种有趣的FPGA。这种FPGA除了支持对内部逻辑的某些部分进行动态重配置外，还具有以下特征：

- 不破坏器件的输入和输出；
- 不破坏系统时钟；
- 器件中未经过重配置的逻辑部分可以继续正常操作；
- 重配置期间不破坏内部寄存器的值，即使正在重配置的区域也是如此。

后面一点是非常让人们感兴趣的，因为这一特征允许某种功能的实例将数据传递给下一个功能。例如，一组寄存器可能在初始化时被配置成了一个二进制计数器，主系统在某个时刻又可能将这些寄存器配置成线性移位寄存器（linear feedback shift register, LFSR）^①，那么这个线性移位寄存器的初始值就是寄存器在配置前的最终值。

虽然这些器件在技术上是先进的，但其在潜能上则是革命性的。为了表现这种全新的功能，很快就创造出了一些新的词汇，例如“虚拟硬件”（virtual hardware）和“高速缓存逻辑”（cache logic）^②。

虚拟硬件这一术语来源于软件中的虚拟内存，两者都是指实际并不存在的东西。在使用虚拟内存的情况下，计算机操作系统看起来像是使用了比实际可用内存更多的存储器，例如，计算机上运行的一个程序可能需要500MB来存储数据，但是这台计算机可能只有128MB可用内存。为了解决这个问题，无论何时只要这个程序试图访问实际并不存在的内存空间，操作系统都会使用一些技巧将内存中的数据和硬盘上存储的数据交换一下，尽管这种交换技术使程序运行变得较慢，但起码不用去商店现买存储芯片就可以继续执行任务。

376

1959年，美国的Robert Noyce发明了在硅上制作细微铝线的技术，这一技术使现代集成电路得到巨大发展。

同样，高速缓存逻辑的产生源于它与高速缓存存储器的相似性，高速缓存存储器由高速而昂贵的SRAM构成，用来存储活动数据，而其他大部分数据还是存储在慢速低成本的存储器件中，如DRAM。这里的活动数据是指程序当前正在使用的数据和指令或者操作系统认为程序马上就要用到的数据。

事实上，虚拟硬件的原理是比较容易理解的。器件中每一个大的宏功能模块都是由大量更小的微观功能模块组合而成，例如计数器、移位寄存器和多路复用器等。一组宏功能模块被分解成许多微观功能模块时，有两点变得很明显：第一，功能有重叠，像计数器这种元件可能在不同的区域被使用了多次；第二，存在大

^① LFSR，即线性移位寄存器，在附录C中详细介绍。

^② Cache Logic是Atmel公司的注册商标。

377

量的功能潜伏，或者称为功能闲置，即在任何既定的时刻只有一部分微观功能模块是活动的，换句话说，在任何既定的时钟周期只有相当少的微观功能模块在使用。因此，有了这种对虚拟硬件个别部分进行动态重配置的功能，就可以用相当少的逻辑资源来实现各种各样的宏功能。

1959年，瑞士物理学家Jean Hoerni发明了平面制程，可以用平面光刻技术制造晶体管。

跟踪每一个微观结构的使用情况、巩固正在使用的逻辑功能、移除暂时不用的冗余逻辑，通过这些方法，虚拟硬件能够执行的任务要比其看起来具有可用逻辑门执行的任务复杂得多。例如，某个复杂的功能需要100 000等效门实现，但是在任意一个时刻只有10 000门是活动的，这样，通过将其他90 000门实现的功能存储起来或者放在高速缓存内，就可以用一个便宜的只有10 000门的小型器件来代替原来的昂贵的100 000门的器件（如图22-3所示）。

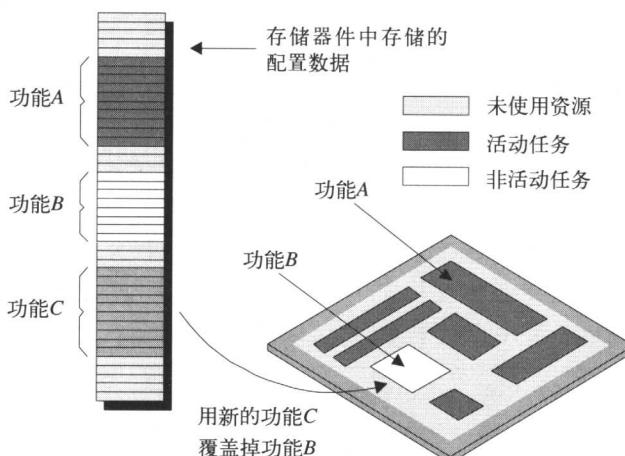


图22-3 虚拟硬件

理论上，实时编译新的设计变更是有可能的，这可以看作是动态地在硬件中创建子程序。

可重配置计算在20世纪90年代后期讨论得比较多，现在仍然有些人在继续打着可重配置计算的旗帜（还穿着T恤），然而，令人难过的是，除了高度专业化的应用外，这种方案并没有真正带来什么结果。核心的问题是传统的FPGA架构太精细化，重新配置一次需要太长的时间（从计算机的角度来看）。为了支持可重配置计算，必须对器件进行每秒成百上千次的重新配置，解决这个问题的答案可能是第23章中介绍的粗粒度的FPNA器件，即现场可编程节点阵列。

378
379

第23章 现场可编程节点阵列

23.1 引言

开始讨论这个题目之前，唯一可以公开说明的是，现场可编程节点阵列（field-programmable node array, FPNA）是作者自己杜撰的一个词汇，至今还不是什么工业标准术语。

精细粒度、中等粒度和粗粒度结构

如果要对各种IC结构进行分类，那么ASIC通常属于精细粒度一类，因为设计工程师可以在单独的逻辑门级别上设定一些功能。比较起来，今天的FPGA可以划归到中等粒度一类，因为这些FPGA由一些处在可编程互连“海洋”（或“小岛”）中的小型可编程逻辑块组成，每一个块都包含许多逻辑门和寄存器（尽管今天的FPGA器件一般都包含了微处理器核、块状存储器和嵌入式功能，如乘法器等，但是根本的结构和上面讲到的相同）。

事实上，许多工程师都将FPGA归入粗粒度一类，但是当我们开始讨论FPNA时，将FPGA划为中等粒度更有意义，因为这些器件才是真正的粗粒度结构。FPNA本质上是由一个节点阵列组成的，每个节点都是一个复杂的处理单元，如图23-1所示。

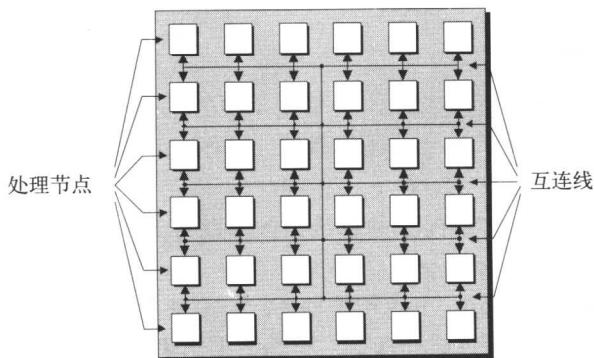


图23-1 FPNA的一般表示法

当然，图中所示的是一种非常简单的FPNA，省略了所有的I/O接口。而且，

[381] 图中只显示了少量相关的处理节点，但实际上这种器件具有很大的潜力，可以包含成百上千的节点。依据厂商的不同，每个节点可能是一个算术逻辑单元（ALU），也可能是一个完整的微处理器（CPU），或者也可能是一个算法处理单元（本章后面将详细讨论）。写本书时，有30到50家公司正在认真地对不同类型的FPNA进行试验，比较有意思的一些典型例子如下所示：

公 司	网 站	产 品
Exilent	www.elixent.com	基于ALU的节点
IPflex	www.ipflex.com	基于操作的节点
Motorola	www.motorola.com	基于处理器的节点
PACT XPP Technologies AG	www.pactxpp.com	基于ALU的节点
picoChip Designs	www.picochip.com	基于处理器的节点
QuickSilver Technology	www.qstech.com	算法元素节点

[382] 鉴于本章所讨论的内容，我们将主要关注其中两个厂商picoChip和QuickSilver，这两家公司提出的概念是相对的（picoChip公司的picoArray器件由处理器阵列组成），它们的主要应用领域是大型的固定设备，例如无线通信网络中的基站，这类应用中功耗不是主要的考虑因素。另外，这些芯片可以不时地重新配置，例如，随着移动电话使用情况的变化，大约每个小时重新配置一次。

1961年，开发出分时处理技术。

比较起来，QuickSilver公司的自适应计算器件（adaptive computing machine, ACM）由许多的算法处理单元节点组成，它们的主要应用领域是小型低功耗的手持设备，例如数码相机、移动电话（在其他很多领域也有应用）。另外，这些芯片可以每秒钟被重新配置（QuickSilver公司更喜欢用“自适应”）成百上千次。

23.2 算法评估

FPNA主要用来执行那些复杂的、计算密集型的算法。因此，在进一步讨论以前，我们应该花一点时间了解一下这些算法，以便对将要讨论的内容有一个大概的认识。

一种是面向字（或者字导向）的算法，例如数字无线传输技术中使用的时分多址（time division multiple access, TDMA）算法，计算量非常大，这一算法的一些变种，如Sirius、XM Radio、EDGE等都是时分多址算法的子集，所以能够处理高端TDMA算法的结构也应该能处理这一算法衍生出来的复杂性更小的其他算法，如图23-2所示。

另一种是面向位（或者位导向）的算法，例如宽带码分多址（wideband code division multiple access, WCDMA）算法及其后续的变种，如CDMA2000、IS-95A

等。WCDMA主要用于互联网、多媒体、视频和其他有容量要求的应用中的宽带数字无线通信。

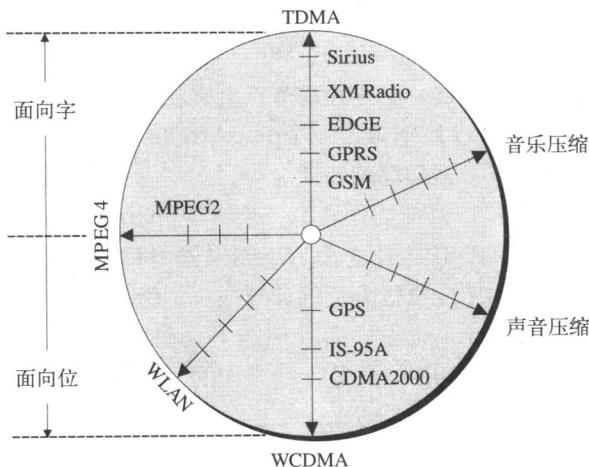


图23-2 算法空间的简单示意图

还有些算法将面向字和面向位结合起来，例如各种MPEG算法、声音和音乐压缩等。

在对这些算法进行评估时，很快就会发现传统的可重配置计算（RC）方法可能在不恰当的级别上处理问题（可重配置计算的概念在第22章中介绍过）。例如，有些可重配置计算（RC）方法处理问题时是在非常微观的级别上，也就是说单个逻辑门或者FPGA模块级别，再加上难度极大的应用编程，这种方法消耗极大，重配置时需要相当长的时间，因此传统的可重配置（RC）方法不适于这些应用。比较起来，其他一些方法处理问题时所在的级别又太宏观，即在整个应用或者算法级上，结果造成资源使用上的低效率。

383

1962年，美国的Steve Hofstein和Fredric Heiman在RCA研究实验室发明了场效应晶体管，即FET。

算法本质上是千差万别的，这或许没有什么奇怪之处，如果你采用了一组不同的算法，它们的构成要素必定具有极大的差异。基于此，一个很简单易懂的解决方案是针对不同的算法而采用不同的结构，但是这样做看起来像什么呢？

23.3 picoChip公司的picoArray技术

为了满足上面所讨论算法的处理要求，picoChip公司提出了一种新器件picoArray。picoArray中各种基于节点的结构具有这样的特征：含有各种不同的

384

RISC处理器，这些16位的器件可以用多种方式进行优化，例如一类处理器可能有大量的存储空间，而另一类可能支持某些专门的算法指令，可以在单时钟周期内执行CDMA无线传输标准中的“spread”和“despread”操作，而用通用处理器需要40个时钟周期。

这些器件的第一代中，每一个处理器节点大概相当于一个控制类型的ARM9或者一个DSP类型的TI C54xx处理器，一个picoArray中就包含几百个这样的节点，可想而知这样的器件具有多么强悍的处理能力。

举一个例子，2002年12月前后作者第一次听说picoArray概念的时候，当时世界上处于绝对领先地位的DSP芯片之一是德州仪器（TI）公司的TMS320C6415，这个芯片能以极其惊人的速度执行大量的计算任务。但是，picoChip公司宣称，一个运行在160MHz的picoArray芯片，其处理能力（在16位ALU MOPS中测量）是600MHz的TMS320C6415的20倍，这是十分强大的。

23.3.1 一个理想的picoArray应用：无线基站

移动电话公司每年都花数十亿美元建设无线基础设施，这些资金中大部分用于开发无线基站中的数字基带处理部件。根据位置的不同，每一个基站必须具有同时处理几十到几百个频道的能力。

385毫无疑问，减小每个频道的实现成本面临着巨大的压力，一个picoArray芯片能够代替许多传统的ASIC、FPGA和DSP，这就极大地降低了每个基站频道的成本。

事实上，使用传统解决方案遇到的问题之一是，至少需要三种设计环境：ASIC/FPGA、DSP和RISC（微处理器类型的功能）。这些使开发和测试变得更加复杂，也延长了基站推向市场的时间，不是一件好事（如图23-3a所示）。相比而言，基于picoArray的方案有一个主要的优点，它将所有的事情都集中到一个单一的设计环境中，如图23-3b所示。

386另外，传统的方案中，尽管ASIC能够提供极高的性能，但是它们的开发费用极其高昂，而且设计周期很长，而且，用ASIC实现的算法完全固定在了硅芯片中无法更改。这是一个主要的问题，因为各种无线传输标准发展非常快，到某个算法的ASIC设计真正实现的时候，它可能已经被废弃不用了（坦白地说，这种情况比想象的更经常发生）。

在基于picoArray的方案中，器件中每个处理器节点都是完全可编程的，所以每个频道都可以很容易地重新配置以便适应实际使用中每小时出现的变化、每周进行的增强与bug修复，以及无线传输协议每月的发展。这样，一台基于picoArray技术的基站在现场应用中的生命周期将会大大延长，从而降低了运行成本。

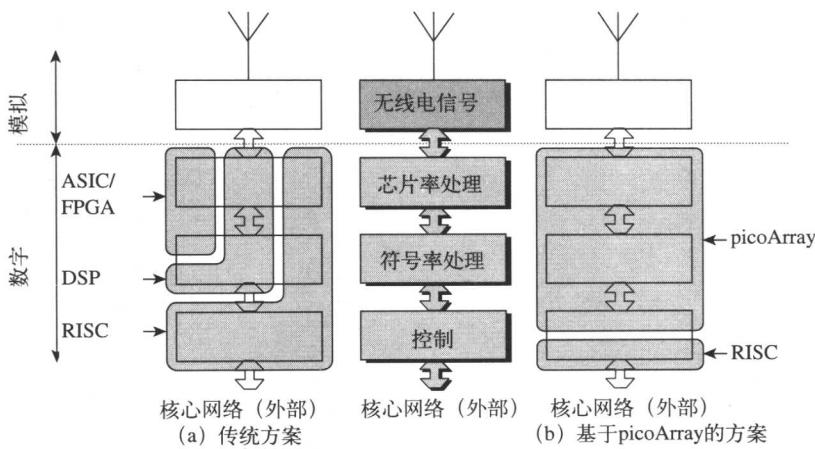


图23-3 传统器件与picoArray

23.3.2 picoArray设计环境

picoArray中每个处理器节点的功能都是用纯C语言或者汇编语言描述的。第11章讨论过，C是一种顺序语言，所以需要使用一些方法来描述并行处理过程，picoArray公司并没有使用第11章提到的那些增强C/C++的技术，而是用VHDL来搭建整个设计的框架结构，其中包含了所有并行处理的过程，然后将设计的各个模块连接起来，在每个模块的内部使用C语言或者汇编语言描述功能。

picoChip公司解决措施中另一个有趣的方面是，他们提供一个完整的编程/配置模块库，将库里的模块相互连接就可以实现一个全功能的基站（用户也可以使用某些独立的模块实现自己的算法，这样就提高了竞争优势）。大约在2003年5月，picoChip公司宣布，他们利用这个库在世界上第一次实现了一个3GPP可兼容负载级基站，而且通过这个基站进行了一次3G通话。从那以后，该公司就连续跳跃式地前进，如果要了解最新的进展，请访问他们的网站www.picochip.com。

387

23.4 QuickSilver公司的ACM技术

多年以来，QuickSilver公司的工程师们一直在利用他们自己的现场可编程节点器件（FPNA）“秘密地”开展各种开发工作（虽然我确信他们肯定会对这一名称有所抱怨）。基于作者所知道的（要比他们认为我知道的更多，至少我自己这么认为），可以公平地说，QuickSilver公司的技术（该公司称为自适应计算机，即Adaptive Computing Machine, ACM）确实是一种革命性的基于不同类节点型的结构和互连结构，如图23-4所示。

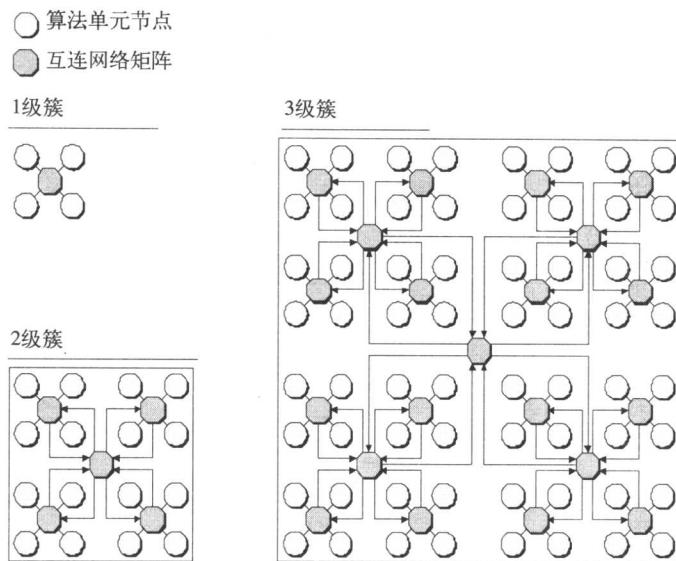


图23-4 自适应计算机（ACM）结构

最低层次是一个算法单元节点，4个这种节点构成一个“方格”（quad），并用一个互连网络矩阵（Matrix Interconnect Network, MIN）连接，最终构成第1级团簇（cluster），四个第1级cluster以同样的方式组成第2级团簇，依此类推便形成更大的团簇。

写作本书时，已经有许多不同类型的算法单元节点（稍后，我们将讨论怎样将这些不同类型的节点映射到方格中）。我们不会在这里深入讨论每种节点的细节，但是理解每个节点完全是在算法级上执行任务还是很重要的，例如，一个算术节点可以执行很多种不同的线性算法，如FIR滤波器、离散余弦变换（DCT）、快速傅立叶变换（FFT）等，这种节点还可以实现各种非线性算法，例如求函数 $(1/\sin A)$ ($1/x$) 的13次幂。

同样，一个位操作节点能够执行各种（不同宽度）位操作功能，例如，线性反馈移位寄存器（Linear Feedback Shift Register, LFSR）、Walsh码产生器、GOLD码产生器、TCP/IP包鉴别器等。

每一个节点都作了外部封装，所以从外界看，所有的节点都是相同的，这层封装负责接收外界传来的信息包（指令、原始数据和配置数据等），然后对信息包拆包并分发给各个节点、完成任务管理、收集计算结果并对结果打包，然后返回给外界。

“封装”是将节点同外界隔离开来，并使所有的节点在外观上一致，当我们把每个节点看作“完整的图灵机”时，这一概念就显得非常有意义。也就是说，可

389

以将任何问题提交给任何节点处理，例如提交一个位操作任务给算法节点，这个节点可以解决问题，只是比起那些专门处理位操作的节点来，效率较低。此外，QuickSilver器件还允许用户创建自己的节点类型，由用户自己定义节点的内核并用QuickSilver的封装将这个内核包装起来。

整日想如何通过这些复杂的东西将工作完成得最好，是件头疼的事。关键的一点是，器件中的任何部分，从个别的节点到整个芯片，都可以很快地被重新配置，多数情况下在一个时钟周期内完成。另一个令人感兴趣的方面是，每个节点中大约有75%是局部存储器，这就要考虑在算法实现的方式上产生根本的改变。与各个功能块之间的数据传递方式不同，节点的功能变化时，节点之间传递的数据还可以继续留在节点内，也就是说，ASIC实现中每一种算法都需要在硅片上开辟自己专用的区域，而自适应计算机（ACM）具有每秒钟重新配置成百上千次的能力，因此在任何一个时刻需要留在器件中执行的只是一个算法的某些部分（也可参考本章后面讨论的SATS），这就极大地减小了硅芯片的面积，降低了功耗。

23.4.1 设计混合节点

对于这个话题，不知从何说起，在这里先大概浏览一遍。前面我们已经知道有很多种不同类型的算法单元节点，还知道每一个团簇由四个节点与一个互联网络矩阵（MIN）组成，所以，你应该想知道这些不同类型的节点是怎样被分配到多个团簇中去的。

重要的是QuickSilver公司并没有真正制造和销售芯片（当然不包括评估与验证用芯片），相反，该公司将ACM技术授权给对这项技术感兴趣的任何人，因此最终用户可以根据自己的特殊应用来决定混合使用哪些类型的节点，然后再根据规范制造出芯片。由于这些节点都有一个封装层，从外界来看它们都是一样的，所以变换各个节点类型非常容易。

23.4.2 系统控制器节点、输入输出节点及其他节点

除了图23-4中所示的结构外，每一个ACM中还包含许多专用节点，例如系统控制器、外部存储控制器、内部存储控制器和输入输出节点。每一个I/O节点可以用来实现一些I/O任务，例如UART或者PCI、USB、Firewire等总线接口（算法节点和I/O节点可以在几个时钟内重新配置）。除此之外，I/O节点还可以用来输入配置数据，因此，每个ACM在必要的时候可以将全部输入引脚用于配置总线。

我们再讨论一下应用程序是怎么被创建并在ACM上立即被执行的。现在几乎每一个用其他技术很难实现的问题都可以用ACM技术来处理，例如，每个ACM都有一个片上操作系统，分布于系统控制器节点和每个算法单元节点封装之上，各个算法单元节点按照进度表执行各自的任务和节点间的通信，这就极大地减轻了

390

系统控制器节点的负担，因为这样一来，系统控制器节点的主要任务只限于确定哪些节点当前是空闲的，然后为它们分配一些新的任务。

从图23-4可知，核心的ACM结构具有非常好的扩展性。如果一块电路板上有多个ACM，处理过程会变得更加智能，因为它们的操作系统连接在一起，对于系统的其他部分而言，它们就像一个独立器件。

23.4.3 空间与时间分割

ACM结构最重要的特征之一是具有每秒钟重配置成百上千次的能力，而消耗的功率却非常少。这一特性使ACM能够支持空间与时间分割（Spatial And Temporal Segmentation, SATS）。

391 许多情况下，不同的算法，甚至同一算法的不同部分是在不同的时间段内执行的。空间与时间分割（SATS）是指动态地重新配置硬件资源，在不同的时间段内使用不同的ACM中的位置（节点）快速执行算法的各个不同的部分。

举一个简单的例子，无线电话中有一些操作称为模式，只有在某些时间段内才需要被执行。三种主要的模式分别为获取、闲置和流量，获取模式是指移动电话定位于最近的基站；在闲置模式，移动电话保持与刚找到的基站的联系并监视页面调度信道，查找信号时发出请求：“唤醒，正在初始化一次通话”；流量模式有两种情况：接收（receiving）与发射（transmitting）。尽管你可能认为打电话时你是同时在说和听，但实际上在一个数字电话上任何时刻都只能做一件事情：要么说，要么听。

1963年，Philips公司开发出了第一个音频盒式录音带。

在基于传统IC技术的无线电话中，这些基带处理功能中的任何一个都需要单独的芯片或者同一个芯片中不同的区域，也就是说，即使某一个功能暂时不用，它仍然会占用一定的芯片面积，因而增加了成本并消耗功率。相比而言，基于ACM技术的电话只需要一个芯片，在需要的时候重新配置便可执行不同的基带处理功能。

但这仅仅是开始，许多情况下，每一个这些主要的功能都由一整套算法组成，而这些算法是在不同的时间段内执行的，例如，图23-5描述了一个高度简化的无线电话接收和处理信号的过程。

输入数据是一系列压缩数据块，每个数据块占用很微小的一段时间，每个数据块都要通过一系列算法处理，每个算法对数据进行一些处理并调整到更低的频率。

392 此过程的一个主要特征是：每级算法都要占用一个时间段。用传统的ASIC实现时，每种功能都要有各自的芯片或者同一个芯片中不同的实际区域，这就造成

可用资源的极大浪费（面积与功耗），因为在任何时刻只有十分有限的几个功能处于执行过程中。

由于ACM具有自适应性，需要的时候可以重新配置执行不同的算法。这种按需生产硬件的概念使得对硬件的使用在成本、尺寸（硅芯片面积）、性能和功耗上都具有最高的效率（ACM方案的性能要比对比方案高出十到一百多倍，而功耗只有其一半到二十分之一。）

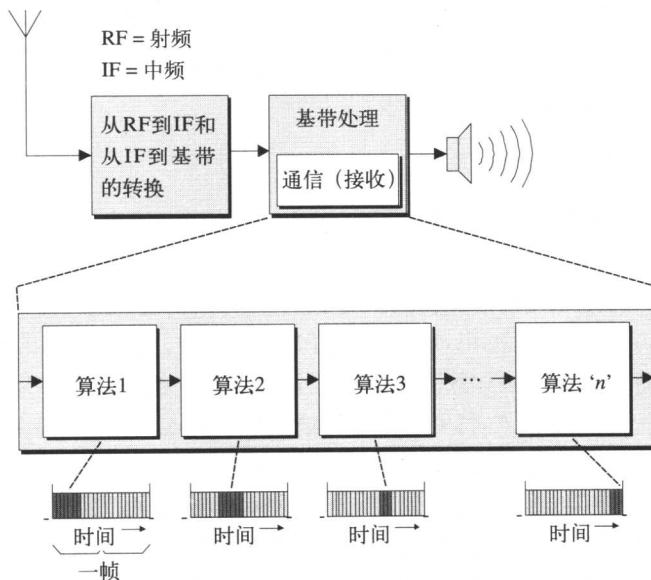


图23-5 高度简化的无线电话接收和处理信号的过程

23.4.4 在ACM上创建和运行程序

下一个大问题是如何为这些ACM创建应用程序。QuickSilver的设计流程是建立在一种基于C语言的系统设计语言上的，该语言称为SilverC，与第11章中介绍的增强型C/C++语言相似。[393]

1967年，美国Fairchild公司开发出了一块集成电路Micromosaic，这是现代ASIC的前身。

SilverC保留了传统C语言的语法和控制结构，使C语言编程人员和DSP设计人员很容易使用，并简化了C语言代码的转换过程。SilverC还包含专用的module、pipe和process等关键字/扩展，方便数据流的描述，并支持并行处理的编程。此外，SilverC还为DSP编程提供了专门的扩展，例如为了有效地使用DAG资源而设置了环形指针、固定宽度整数和定点数据类型，支持饱和和非饱和数据类型（saturated

and nonsaturated types) 等。

SilverC描述比传统ASIC和FPGA设计流程中使用的HDL描述（Verilog和VHDL）具有更快的输入和仿真速度。一旦完成SilverC描述的仿真和验证，就可以将其编译成一个可执行的二进制Silverware应用程序。ACM的片内操作系统在任何时刻只会装载Silverware中的某些需要的部分，而且多个Silverware应用程序可以并行地在一个ACM上运行。

很重要的一点是，创建Silverware应用程序时并不需要知道将要用哪种类型的ACM芯片（包括混合类型节点），也不必知道电路板上究竟有多少个ACM芯片。ACM的片内操作系统会处理所有这类细节问题。

23.4.5 还有更多的内容

在第12章我们讨论了基于DSP的设计流程，介绍了系统级设计和仿真环境的概念，例如MathWorks公司（www.mathworks.com）的Simulink，这个工具拥有很广泛的用户基础，促进了面向数据流的设计，为ACM结构提供了一个极好的映射。

394 QuickSilver公司已经将SilverC集成到了Simulink中。在最简单的层次上，你可以用Simulink描述各种模块和数据流以及它们之间的连接关系，然后Simulink将自动为设计输出一个顶层框架，其中包含了模块实例和将这些实例联系在一起的通路。在此基础上，你可以深入到这个框架内部手工编写SilverC代码，完成处理任务。

另外，QuickSilver公司还开发了一个SilverC模型库，已经映射到了现有的Simulink模块上。这个库中包含了被广泛使用的DSP元件、滤波器、编码器、解码器和比特与字操作模块。这些SilverC模块可以用来作功能和周期精确的仿真，而且编译成Silverware可执行程序后，它们可以直接映射到ACM动态硬件资源上。

23.5 这就是硅，但与我们知道的并不相同

你可能已经猜测到，我对于通用领域有FPNA器件而专用领域有QuickSilver的器件感到很兴奋，这是不是意味ASIC和FPGA就要退出历史舞台了？当然不是。

FPNA器件确实适用于很多应用领域，但是并没有全能型的、放之四海而皆准并能做好所有事情的芯片结构（例如作为一个副产品来美白牙齿）。实际设计中，FPNA只不过为系统结构工程师提供了一种更多的选择。

另一方面，根据以前的经验，在不远的将来出现内嵌FPNA内核的ASIC和FPGA芯片是完全有可能的，这并不奇怪。另外，前面也提到，QuickSilver结构允许用户创建自己的节点类型，然后用QuickSilver的封装层进行封装，所以，另一种选择是使用QuickSilver提供的ACM结构，但是内部的节点使用FPGA结构来实现。

395 如果这些都变成现实，我一定会声势并用地喊：“我告诉过你们这些会发生的！”

第24章 独立的设计工具

24.1 引言

我们谈到逻辑仿真器、综合器等设计工具时，主要关注的是那些大型的全产品线EDA公司，或者是一些较小规模的专注于设计流程中某个专门的领域的EDA公司，或者是FPGA厂商自身。

但是，我们不应该忘记那些在开源领域内辛勤工作的人（见第25章），还有一些小型的FPGA设计咨询公司常会花相当多的时间和精力来开发适当的小工具来协助其内部的开发工作。有时，这些工具也很有用，最终还被产品化，可以在公司以外使用。这一章，我们就简单地介绍这类工具。

24.2 ParaCore Architect

Dillon Engineering (www.dilloneng.com) 公司提供多种多样的客户化设计服务，专注于基于FPGA的DSP算法和高带宽实时数字信号处理以及图像处理应用。

在20世纪90年代末期，该公司的工程师们开始有意识地不断重新创造和实现一些浮点库、卷积核以及FFT（快速傅立叶变换）处理器等，为了让工作更加容易，他们开发了一个工具，称为ParaCore ArchitectTM，这个工具为IP核的设计提供了便利。

开始先要创建一个源文件，使用一种基于Python（第25章对Python语言进行详细介绍）的语言在极高的抽象级上对设计进行高度参数化描述。ParaCore Architect将这个描述与用户指定的参数结合起来，产生一个等效的HDL描述、一个周期精确的用来提高验证速度的C/C++模型和一个testbench，如图24-1所示。

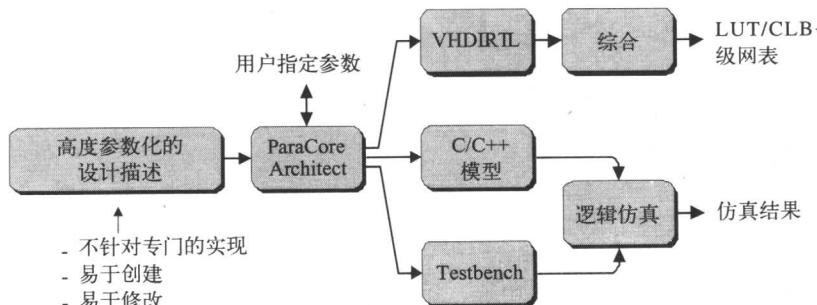


图24-1 ParaCore Architect产生了RTL、C/C++和testbench

这样产生的HDL代码保证可以用于任何仿真和综合环境，因此没有必要运行任何HDL规则检查程序。这种高度参数化的表示方法最大的优点是，可以十分容易地将设计转向新的应用或者其他器件。

24.2.1 产生浮点处理功能模块

现在许多的FPGA厂商提供含有嵌入式微处理器的FPGA器件就是使用ParaCore Architect的简单例子。但是，令人有些失望的是，这些器件一般都没有包含浮点单元（floating-point unit, FPU），这就是说，如果设计人员想要执行浮点表示法的浮点操作，他们要么用软件来实现（极其耗时），要么用硬件实现。后一种情况下，将花费大量的精力，而这些精力本来可以用在开发设计中那些有趣的部分。

398

由于这个原因，ParaCore Architect的设计描述之一可以用来产生相应的浮点内核。使用不同的参数，就可以定义需要什么样的指数和尾数精度、使用多少级的流水线、是否处理IEEE浮点标准中规定的特殊情况，如无穷大（有些应用并不需要这些情况）、使用的微处理器核类型（便于创建适当的接口模块）等。

24.2.2 产生FFT功能模块

产生FFT核所用的设计描述就是一个证明ParaCore Architect强大功能的好例子。其中最小的计算单元称为“蝶形单元”（butterfly），其包含一个复数乘法器、一个复数加法器和一个复数减法器，如图24-2所示。

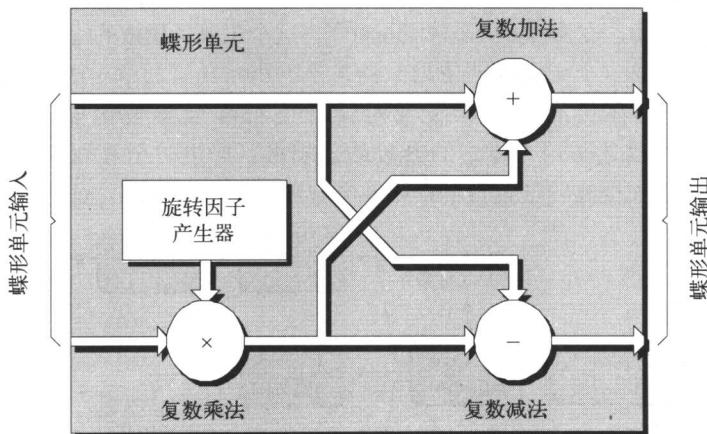


图24-2 蝶形单元是FFT中最小的计算单元

复数乘需要四次简单乘法和两次简单加法，复数加和复数减各需要两次简单

399

加法，这样，每个蝶形单元总共需要四次简单乘法和六次简单加法。

一个实际的图像处理应用需要一个每秒能处理120帧图像（120 fps）的二维 $2K \times 2K$ 点FFT。处理一个含有2 048（2K）个像素的行总共需要11 256个蝶形单元，排成11列，第1列蝶形单元的输出驱动第2列蝶形单元的输入，依此类推，因此，总共需要45 025次简单乘法和67 536次简单加法。为了产生整个 $2K \times 2K$ 画面的FFT，每2 048行就要重复以上的过程，也就是说，为达到每秒处理120帧画面的速度，每一行数据的处理必须在4μs之内完成（相应地，每次简单乘法只有90ps的时间，加法则只有60ps时间）。

我们再来考虑一下实现2 000点FFT需要的11 256个蝶形单元。如果执行时间不是关键因素，有必要考虑使用一个相对较小的FPGA器件（例如Xilinx公司Virtex-II系列的XC2V40，其内部含有四个乘法器模块），创建一个蝶形单元结构（四个乘法和六个加法），然后循环使用这个结构完成全部蝶形运算。这样的结构完成2 000点FFT需要90μs的时间。尽管这是非常了不起的成绩，但离上面讨论的图像处理应用所需要的4μs时间还相去甚远。

为了提高这一算法的速度，最容易的方法是在硬件中增加蝶形单元，以便并行执行更多的运算。若使用Xilinx公司的XC2V6000器件（600万系统门，144个18位乘法器，144个18KB块RAM）完成全部 $2K \times 2K$ 点FFT，实现一个能达到120fps（frame per second）的图像处理系统还是有可能的。

重要的一点是，选定这些不同的目标器件只需要在ParaCoer Architect中设定一个参数，此参数用于指定硬件中要实例化的蝶形单元结构的数量，仅此而已。

再举一个例子，如果要将FFT的长度由2 000点减少到1 000点，只需设置一个参数就可以兼顾所有细节，包括改变用来存储中间结果的RAM的大小。同样，可以使用另一个参数来选择定点格式还是浮点格式（后一种情况下，还有两个参数分别指定指数位和尾数位的大小）。

2002年早期，Dillon Engineering公司利用ParaCore Architect创建了可能是当时世界上最快的FFT处理器。后来这个处理器被用于很多领域，例如SETI（探索外星文明）工程中，该处理器负责处理大量来自探索外星文明的射电望远镜传来的数据。

24.2.3 基于网络的接口

Dillon Engineering公司已经可以让客户通过互联网使用ParaCore Architect。在创建像FFT这类功能模块时，常常要试验各种不同的设置，例如每一点需要存储多少位，现在，Dillon Engineering公司的客户可以访问www.dilloneng.com这个网站，选择感兴趣的内核类型，指定一组参数，然后按一下Go按钮就可以产生等效的HDL描述、C/C++模型和相应的试验台。

24.3 Confluence系统设计语言

与大多数工程师一样，我面对另一种软件编程语言或者硬件设计语言时也会“发抖”，但是Launchbird Design Systems (www.launchbird.com) 公司提出一种新的系统设计语言Confluence和对应的Confluence编译器，这还是非常值得一看的。

一下子要了解Confluence的方方面面是很困难的，但我们总要试一下。首先，
[401] Confluence是一种极其紧凑的语言，既可以描述硬件又可以描述嵌入式软件。描述硬件时，Confluence编译器可将Confluence语言编译成对应的VHDL或Verilog形式的RTL代码，如图24-3所示。

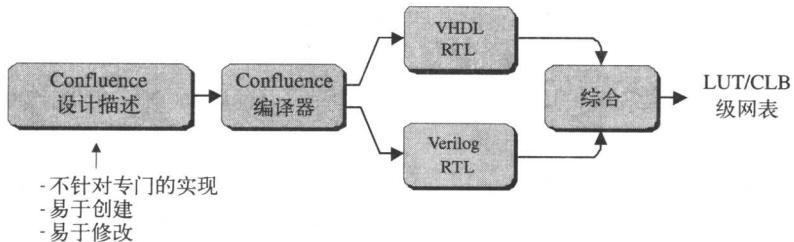


图24-3 从Confluence编译器输出的一种高度简化的表示方法

可以这样来理解这个概念，就是用HDL（VHDL或Verilog）描述的是某种特定的电路，但是用Confluence描述的却是算法，能够产生所有同类电路。关键是可以用非常少的Confluence代码量描述更多的功能（一般能将源代码减少3到10倍，这样可以更快地实现设计，更容易地管理设计，更快地验证设计）。而且，所得到的结果是“绝对纯洁”的RTL，也就是说不存在普遍性的错误和不良的设计方法。

在编程方面，Confluence提供了递归运算、高阶数据类型、词汇作用域 (lexical scoping) 和引用透明性 (referential transparency)，这些特性足以令任何系统工程师兴奋不已了。

24.3.1 一个简单的例子

举一个简单的硬件方面的例子，假如有一个Confluence语言描述的组件，可以级联具有任意级数的单输入单输出元件，描述如下：

```

component Cascade +Stages +SisoComp +Input -Output
  is
    if Stages <= 0
      Output <- Input
    else
      Output <- {Cascade (Stages - 1) SisoComp
                 {SisoComp Input $} $}
  end
endcomponent
  
```

```
    end
end
```

1402

尽管没有程序员开始就认为上面的代码有些许吓人，但也不会总那么糟糕的。it's really not all that bad。第一行声明了一个新的组件，我们决定称之为Cascade，这个组件有四个参数：Stages（要求的级数）、SisoComp（要级联的下级组件的名字）、Input（输入信号名或者总线中的信号）和Output（输出信号名或者总线中的信号）。

1970年，美国Fairchild公司开发出第一个256位的静态RAM，称为4100。

注意，在这一行中，语言本身的关键字只有component和is，Stages、SisoComp、Input和Output都是用户定义的变量名，“+”和“-”表示相关的用户定义的变量是否分别是输入端口和输出端口。

另外，虽然我们在说这个组件级联了任意的单输入单输出元件，但是输入和输出变量其实都可以是多位的总线。事实上，这些信号甚至不一定非得是位向量，它们可以是位向量的列表或者位向量列表的列表（或其他数据类型的列表）。

这里用一个简单的例子说明Cascade组件的使用方法，假如因某些疯狂的原因，我们想把1 024个非门串联在一起（不用问为什么），也就是用第一个非门的输出驱动第二个非门的输入，用第二个非门的输出驱动第三个非门的输入，依此类推，这种情况下，我们只要用一行语句就可以完成这一功能，调用时传递适当的参数即可：

```
{Cascade 1024 ('~') Input Output}
```

本例中，Confluence编译器将符号“~”理解为基本的逻辑非(NOT)功能。

再举一个更有意思的例子，假设我们想级联16个8位寄存器，这样就要用第一个寄存器的输出驱动第二个寄存器的输入，第二个再去驱动第三个，依此类推。首先，我们需要声明一个组件，可以称为Reg8，表示8位寄存器，然后利用Cascade组件将这个寄存器复制16次，代码如下：

```
component Reg8 +A -X is
  {VectorReg 8 A X}
end
```

```
{Cascade 16 Reg8 Input Output}
```

是不是很酷呢？但是还可以更好，在每一级之间用一个流水线寄存器实现四次信号的平方操作又如何呢？我们可以简单地描述如下：

```
component RegisteredPowerOfTwo +A -X is
  {Delay 1 (A '*' A) X}
end
```

403

```
{Cascade 4 RegisteredPowerOfTwo Input Output}
```

可以看到，Cascade组件极好地演示了递归调用和高阶数据类型的用法，这两个编程中常用的主要特征提供了更高级的抽象能力，增强了设计的可重用性。

1970年，美国Intel公司公布了第一个1024位的动态RAM，称为1103。

由于没有限制下级子组件变量SisoComp的输入和输出端口必须具有相同宽度，因此优点还远远不止这些。实际上，这个变量可以关联任何用户定义的功能，甚至能输入一个组件然后输出一个组件，或者输入一个系统（一个例化后的组件）然后输出另一个系统。同样，也没有限制SisoComp只能操作位向量，它也可以操作整数、浮点数、列表、组件、系统或者其他任何Confluence数据类型。

最后一个例子，SisoComp可以将一个位向量与其自身连接起来，从而使位向量中的位数加倍。为了说明这一点，假设我们先创建一个新的组件SelfConcat：

```
component SelfConcat +A -X is
  X = A '++' A
end
```

404

这里的“++”是连接操作符，当SelfConcat和Cascade结合使用时，位向量的宽度在每一级调用后都会变为原来的两倍。例如，开始有一个2位宽的位向量“01”，通过SelfConcat传递到Cascade中：

```
{Cascade 4 SelfConcat '01' Output}
```

这样，输出将是一个32位宽的位向量，其数值为：0101010101010101010101010101010101。

当然，VHDL语言一直就有generate语句，Verilog语言也在其Verilog 2001增强版中添加了这种功能，但是Confluence语言的功能远远超过了这些语句。

24.3.2 还有更多的功能

前面已经说过，一下子要了解Confluence的方方面面是很困难的。或许最好的方法是通过图例总结一下，如图24-4所示。

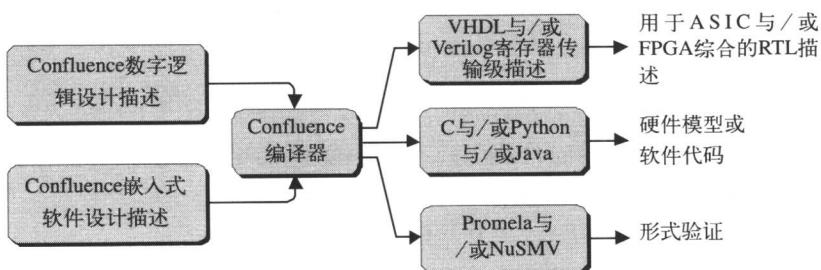


图24-4 Confluence编译器输出的一种更精确的表示方法

在输入一侧，可以使用Confluence语言来创建一个硬件的描述或者一个嵌入式软件。若是一个硬件描述，就可以指示Confluence编译器产生VHDL或Verilog格式的RTL代码，以便仿真和综合工具使用。

也可以使用Confluence编译器输出ANSI C、Python或者Java格式的描述（Python语言将在第25章详细介绍）。如果输入的是硬件描述，那么这些类型的输出可能是周期精确并且位精确的高性能仿真模型，可以连接到客户验证环境中；如果输入了软件描述，那么这些输出可能是在硬件/软件协同验证环境中使用的可执行代码。

最后，还可以指示Confluence编译器产生PROMELA或者NuSMV语言的描述，可以分别用于开源SPIN模型检查器和NuSMV符号模型检查器的形式验证中。形式验证在第19章讨论过，PROMELA、SPIN和NuSMV将在第25章详细讨论。

24.3.3 免费评估版本

如果访问Launchbird公司的网站www.launchbird.com，你将会发现大量的Confluence源代码例子。真正够“cool”的主意是每个人都能免费下载和使用一个单独的无限制的license。若用于商业目的，license将是收费的（用于学术方面是免费的），但是价格总是在变化，所以，必须从Launchbird公司获得这方面的最新消息。

使用这个免费的license，你可以开发任何模型（任意的Confluence源代码模型和任意的VHDL、Verilog、C模型），而且你可以用这些模型做任何想做的事情，包括出售它们，而且无论从哪个方面看，这都是一个很好的交易。

24.4 你是否具有这种工具

在工作或学习过程中，我们应该碰到过小型设计组织（工作室）开发出来的有用的工具，或许你自己已经设计了这种工具，那么请通过电子邮件地址max@techbites.com与作者联系，可能在本书下一版中包含进去，或者可能成为www.eedesign.com网站上作者的双月刊“Max Bytes”专栏中的一篇论文。

第 25 章 创建基于开源的设计流程

25.1 如何白手起家创办一家FPGA设计工作室

这里并非要推荐使用没有良好支持的工具。FPGA厂商提供的低成本工具在成本优先的设计中较受欢迎，而大型专业EDA公司提供的更强大的工具在大规模的复杂设计中则更受欢迎。

但是，如果要在家里创办一个FPGA设计工作室，由于资金有限（或者没有资金），这里所说的开源工具就可能是更好的选择。

许多规模非常小的设计工作室专注于ASIC的开发，这些工作室往往只有两三个人在车库里办公，因此并不多见。其实这并不奇怪，开发这类器件所需要的设计工具极其昂贵，往往高达100 000美元，而且还在继续上涨（当然，实际制造一个芯片的成本高达数百万美元的现实也令人望而却步）。

相比而言，现代FPGA与开源EDA和IP技术的最新发展相结合，已经使建立FPGA设计环境的成本降到几乎为零，这就为那些大学生和有经验的专业人员自主创业铺平了道路。

创办一个成功的FPGA设计工作室，除了要熟悉有关的数字逻辑设计知识外，还要具备以下几个基本条件：

- 一个开发平台
- 一个验证环境
- 形式验证（可选）
- 有公共IP元件可用
- 综合与实现工具
- FPGA开发板（可选）

25.2 开发平台：Linux

瑞典工程师Linus Torvalds（及其朋友）于1990年前后开发了Linux操作系统，此后，Linux很快就成为ASIC和FPGA开发的主流平台。尽管大多数FPGA综合和实现工具开发之初都是基于微软的Windows[®]操作系统，但现在正在或者已经转向Linux平台。

Linux和GNU为硬件和软件开发提供了许多非常有价值的工具。一些公共的Linux工具如下（不分排名先后）。

- **gcc**: C语言仍然是仿真和验证领域内最快的建模语言。如果设计非常大，造成HDL（Verilog或VHDL）仿真无法进行，那么可以考虑创建一个周期精确的C语言模型，使用开源的GNU C编译器（gcc）进行编译。
- **make**: make工具用来使构造（build）过程自动化，在硬件设计中，“build”可以是从仿真、HDL代码生成和逻辑综合到布局布线的任何一个过程。为了让make工具知道要处理哪些文件以及哪些文件之间存在依赖关系，必须在一个名为makefile的文件中定义这些文件和它们之间的联系。
- **gvim**: 来源于“visual interface”，VI是一个经典的UNIX文本编辑器，vim工具是一个VI的增强版本，gvim又是vim的图形用户界面（GUI）版本。gvim对VI进行了扩展，增加了语法高亮功能及各种优秀的宏功能。gvim内建了对Verilog和VHDL语言的支持，是一个超快的设计输入工具，编辑过程中手指几乎不用离开键盘。
- **EMACS**: 许多黑客认为这是一个终极编辑器，EMACS（来自“Editing MACroS”）是一个可编程的文本编辑器，内部集成了一个完整的LISP解释系统。EMACS比VI更强大也更复杂，现在有一些模块可以支持Verilog和VHDL的开发任务。

408

LISP代表“List Processor”，虽然也有些批评者说，它实际上代表“Lots of Irritating, Superfluous Parentheses”。

- **cvs**: Concurrent Versions System（CVS）是一个主流的开源版本控制系统，具有网络透明性，适用于每一个人，从单个的开发人员到大型分布式开发团队。CVS支持分支、多用户和远程协作，负责管理一些目录（文件夹）树和文件，并维护所有对它们变更的历史记录。利用这个历史记录，CVS可以重新产生历史目录和文件状态，并显示出变更时间、原因及变更者。所以，如果不小心弄乱了RTL代码或者决定重新综合三个月前的设计版本，CVS会帮助处理好这一切。
- **PERL**: 脚本语言常常用于一次性编程工作或者创建原型，在电子设计领域，也可以使用脚本语言将设计流程中的许多工具结合在一起，控制这些工具的工作方式并管理工具之间的数据传递。Practical Extraction and Report Language（PERL）是历史上使用非常广泛的脚本语言之一，由Larry Wall开发。PERL也被戏称为UNIX和Linux编程中的“瑞士军刀”，许多硬件设计流程仍然使用PERL脚本将各种工具结合在一起。
- **Python**: Python语言比PERL语言更强大，是一种“全能”的脚本语言，已

经发展为成熟的编程语言。Python语言由Guido Van Rossum于1990年开发，由于Guido非常喜爱Monty Python的飞行马戏团，就以Python来为这种语言命名。Python可以用于很多方面，例如连接设计流程、高级建模和验证、创建客户化的EDA工具（参考本章后面对Python的其他讨论）。

- 409** □ diff：相当简单又非常有用的一个工具，可以快速比较源文件、检查和报告文件之间的差异。

1971年，美国。Ted Hoff设计了第一个单片机4004微处理器，由Intel发布。

- grep：代表globally search for a regular expression and print the lines containing matches to it，即全局搜索正则表达式并打印匹配行，grep用于在一个或一组文件中快速搜索文本字符串或者某种格式。
- OpenSSL：不论公司大小，都会花钱来维护IP的安全性，通过网络或者internet为合作者或客户发送IP时就会面临这方面的问题，这种情况下，在发送前应该考虑将IP加密。一个解决办法是使用开源OpenSSL，这是一个商业级的全特征工具集，实现了安全套接字层(Secure Sockets Layer, SSL)、传输层安全(Transport Layer Security, TLS)协议及一个工业级的通用密码系统库。
- OpenSSH：设计团队成员是不是散落在世界各地？Secure Shell(ssh)工具是一个远程登录程序，允许在远程计算机上执行命令，在不安全网络上的两台无信任主机之间建立安全的加密通信。OpenSSH是ssh套件的开源版本，能够加密所有的通信信息（包括密码），从而有效地消除窃听、连接劫持和其他网络级攻击，它还提供多种安全通道功能和授权方法。
- tar、gzip、bzip2：这些工具不同，但都可以将过程文件压缩和存档。

获得Linux

直到近来，Linux最主要的发行版本仍然是Red Hat (www.redhat.com) 和 MandrakeSoft (www.mandrakesoft.com)^①。但是，Gentoo Linux™ (www.gentoo.org) 正在迅速成为开发者们的最爱。Gentoo有一种独特的包发行系统(package distribution system)，能够自动下载、编译和安装各种软件包，要安装Icarus Verilog 只要键入：

```
$ emerge iverilog
```

几分钟后Icarus就被安装到系统中，可以直接使用。

25.3 验证环境

这一点或许会令人反复争论，但是许多人都会说验证环境是整个设计流程中

^① mandrake已经改名为mandriva。——译者注

最关键的部分。任何人都可以敲击键盘编写HDL代码，但是只有使用验证工具才能为设计者提供反馈信息，保证整个设计朝着正确的实现方向前进。

25.3.1 Icarus Verilog

主要的开源验证工具是Icarus Verilog编译器 (<http://icarus.com/eda/verilog>)，在其基本形式中，Icarus将Verilog设计编译成能够在仿真中运行的可执行程序。实际上，Icarus主要作为一个基于事件的仿真器来使用，但它也能为Xilinx的FPGA做些基本的逻辑综合工作。

Verilog是一种复杂的语言，Icarus的作者Stephen Williams就用Verilog开发工具完成了一项很好的工作。实际上，Icarus Verilog的语言覆盖率和性能超过了某些商业仿真器。

25.3.2 Dinotrace和GTKWave

上面讨论的Icarus Verilog是一个严格的命令行工具，命令行工具在UNIX和Linux环境中都是首选，因为它们可以通过makefile很容易地联系在一起。

Icarus并没有提供GUI来显示仿真结果，但是能够产生工业标准的VCD (Value change dump) 文件，这一文件能被设计流程中的下游工具使用，例如独立的波形显示软件就可以将VCD中的信息显示出来供用户分析。

Dinotrace和GTKWave都是具有图形用户界面 (GUI) 的工具，可以显示VCD格式的仿真结果。这两种波形浏览器能够滚动浏览仿真波形、增加测量线，还能查找模式。Dinotrace (www.veripool.com/dinotrace) 是一个可靠的工具，但是功能有限；相比之下，GTKWave (www.cs.man.ac.uk/apt/tools/gtkwave) 显得有些粗糙，但是最近也有了适当的发展。

411

25.3.3 Covered代码覆盖率工具

验证一个设计时，访问功能覆盖率指标是非常重要的，这样可以保证测试向量能够测试到设计中的边界条件。

Covered (<http://covered.sourceforge.net>) 是一个Verilog代码覆盖率分析工具，能够产生与仿真相关的代码覆盖指标。更确切地说，Covered对Verilog源代码和Icarus Verilog产生的VCD数据进行分析，据此来决定功能覆盖率的级别。

Covered现在可以处理四种类型的覆盖指标：代码行覆盖率 (line coverage)、翻转覆盖率 (toggle coverage)、组合逻辑覆盖率 (combinational coverage) 和有限状态机覆盖率 (finite-state-machine coverage)。

25.3.4 Verilator

近几年，最热门的设计项目就是如何完成SoC设计，这种设计需要将硬件和嵌

入式软件集成在一个单芯片上。许多FPGA中也嵌入了处理器硬核，或者可以使用处理器软核（见第13章）。

一个SoC设计中真正的关键是软件和硬件的协同验证。进入Verilator的网站 www.veripool.com/verilator.html，可以将Verilog代码转换为周期精确的C++模型，这种由RTL源代码自动生成C/C++模型的能力是一个非常强大的验证工具，这样，在仿真时就可以将软件和RTL代码的C/C++版本直接集成起来进行验证。
[412]

另一个有用的工具是Tenison EDA (www.tenison.com) 公司的VTOC。这个工具能够根据RTL源代码产生C++或SystemC模型。

除了用于软硬件协同验证外，Verilator还可以用于通用Verilog仿真，使用周期精确的C语言描述进行仿真比用基于事件的HDL仿真器要快得多，要做的只是用gcc编译器将转换来的C语言代码编译为可执行程序（见25.2节）然后运行^①。

25.3.5 Python

Python (www.python.org) 是一种十分有用的高级脚本和编程语言，其快速实现的能力使其在世界上具有越来越高的知名度。Python正在成为数字设计和验证工程师一个强大的工具，尤其适用于系统建模、testbench搭建和设计管理等。

实际上，许多设计公司也开始发现，用Python模型开始设计过程要比用Verilog或VHDL更加容易和快捷。一旦这些Python模型通过了仿真验证，设计团队就可以一直以这个“Python黄金模型”为参考，并开始进行RTL编码。

MyHDL (www.jandecalwe.com/Tools/MyHDL/Overview.html) 是一个用于高级系统建模的Python框架，它将一些新的特征添加到Python语言（发生器）中，来模拟并发的操作。MyHDL还可以连接到Icarus Verilog中进行Python和Verilog混合仿真。

25.4 形式验证

荷兰数学家、计算机科学的先驱Edsger Wybe Dijkstra教授曾经说过：“程序测试只能找出错误，却不能证明他们不存在。”

尽管硬件仿真仍然是系统测试中主要采用的方法，但也可以通过形式验证来保证系统的正确性（见第19章）。与仿真不同的是，形式验证是从数学上证明一个系统的具体实现满足规范的要求。
[413]

两种主要类型的形式验证是模型检查（model checking）和自动推断（automated reasoning）。模型检查是一种探测系统的状态空间并保证某种属性为真

^① Icarus也具有产生C代码的能力。

的技术，这种属性通常称为“断言”。模型检查有一种形式叫做等效性检查 (equivalence checking)，可以对一个系统的两种不同表述形式进行比较（如RTL和门级网表），并且判断这两种形式是否具有同样的输入和输出功能。自动推断则使用逻辑推理来证明具体实现和相关规范的一致性，这更像是一个数学证明。

25.4.1 开源模型检查

一个主要的开源模型检查工具是SPIN (<http://spinroot.com>)，这个工具已经由贝尔实验室的Gerard J. Holzmann博士开发了将近20年。SPIN是一个非常出色的工具，最近被计算机协会 (Association for Computing Machinery, ACM) 授予软件和系统奖，这不是一个小的荣誉，因为以前的得奖者是UNIX、SmallTalk、TCP/IP和World Wide Web (万维网)。

SPIN接受的输入是一种规范，这种规范是用PROMELA语言描述的系统模型。通过这种语言，用户可以以*never-claims*的形式来创建复杂的断言，也就是定义一系列在系统中永远不会出现的事件。对于一个给定的模型和规范，SPIN会搜索全部状态空间，以便找到违反规范的状态。

SPIN最主要的缺点是，它当初是为异步软件验证而开发的，所以使用了一种显式验证 (explicit verification) 技术，尽管显式验证技术对于软件协议的验证是非常理想的，但是对于大型的硬件设计会存在效率低下的问题。

对于规模适度的硬件设计，符号模型检查器 (symbolic model checker) 是一个不错的选择。和显式验证不同，符号模型检查使用二进制决策图 (binary decision diagrams, BDD) 和命题可满足性算法 (propositional satisfiability algorithms, SAT)^① 来处理问题，尽可能避免进行状态空间探测。此外，还有一个高质量的开源符号模型检查器，称为NuSMV (<http://nusmv.irst.itc.it>)。

414

25.4.2 基于开源的自动推断

上一节讨论的模型检查技术的优点是，它是一个自动化的过程：只要单击一下按钮，等待结果就可以了；缺点是可能会等待很长时间。

虽然NuSMV软件使用的符号表示法对显式模型检查器有很大的帮助，但是状态空间探测仍然是一个很近的威胁，用不了多久，系统规模就会超过模型检查器的实际能力范围。模型检查存在的另一个问题是表达能力有限，不能在模型检查环境中指定一些复杂的断言。另一种方法是自动推断，也称为自动定理证明 (automated theorem proving)。

自动推断不存在模型检查的局限性，例如，与系统规模无关，因为自动推断

^① 缩写SAT来源于satisfiability一词的前三个字母。

不会去搜索状态空间。更重要的是，自动推断支持更高级的表达式，可以对复杂的规范进行精确的建模。

不幸的是，有所得必有所失，尽管名称中有自动，但自动推断却不是一个全自动的过程。实际应用中，验证工程师需要使用一些辅助工具对证明过程进行引导，而且，为了有效地使用这些工具，验证工程师需要精通验证策略、数学逻辑和工具本身的用法，这并非一般的学习过程，但是如果你愿意花费时间和精力，最后必将证明自动推断是最强大的验证方式。
415

1973年，美国Scelbi计算机顾问公司创造出Scelbi-8H计算机，这是一种基于微处理器的个人计算机工具套件。

模型检查的开源工具在努力和商业工具展开竞争，而与其不同的是，自动推断的开源工具本身就处于世界领先地位，这种工具中最流行的三个是：HOL (<http://hol.sourceforge.net>)、TPS (<http://gtps.math.cmu.edu/tps.html>) 和MetaPRL (<http://cvs.metaprl.org:12000/metaprl/default.html>)。

25.4.3 真正的问题是什么

和任何工具一样，形式验证的功能只与使用它的工程师有关。即使处于良好的条件，形式验证也只能回答这样的问题：具体实现是否与规范相符？但是关键的问题仍然是：规范是正确的吗？

对实际设计的评估表明，大多数系统错误从本质上说不是由具体实现中的错误造成的。即使没有使用形式验证，设计正确地实现需求的几率也比没有实现需求大，大多数错误的根本原因通常是需求本身。

打开通路，进行交流与协作是保证规范正确性的最好方法。写作本书时，能够处理这一问题的唯一知名工具是大脑皮层。

25.5 访问公共IP元件

如果你有一间小小的（或者较大的）设计工作室一条很有用的经验就是要避免重复创造已有的东西。随着时间的过去，每一家设计公司都会积累一个经常使用的元件库，这有助于加速设计过程。实际上，有时候可以通过一个设计公司的
416 IP库来判断此公司的实力。

25.5.1 OpenCores

幸运的是，有心的设计人员已经可以通过OpenCores (www.opencores.org) 来访问巨大的IP库了。作为工业界首要的开源硬件IP库，OpenCores聚集了大量的IP，范围极其广泛，涉及算术单元（arithmetic unit）、通信控制器（communication

controller)、协处理器 (coprocessor)、密码处理系统 (cryptography)、数字信号处理 (DSP)、前向误码校正编码 (forward error correction coding) 和嵌入式微处理器 (embedded microprocessor) 等等。此外，OpenCores还负责管理Wishbone，这是SoC中使用的一种总线协议。

25.5.2 OVL

设计人员可能会将设计总开发时间的70%用在验证上，这就产生了访问验证IP库的需求。因此，Accellera (www.accellera.org) 启动了开放验证库的项目，即Open Verification Library (OVL)，用来满足对于公共IP验证元件的访问需求。

25.6 综合与实现工具

综合（逻辑综合与物理综合）是FPGA设计流程中一个主要的步骤，但是目前开源技术还没有完全涉足进来，而且这种情况在短期内不太可能有所改变，因为FPGA综合是一个非常复杂的问题。

写作本书时，Icarus（见前面“验证环境”一节）是唯一能将HDL综合成FPGA基本元件的开源工具。唯一的其他低成本选择是FPGA厂商提供的综合与实现工具（这些工具应该是低成本应用中最主要的选择）。

然而，当一个设计接近了最顶级器件的容量时，FPGA厂商提供的综合工具也会变得不能满足要求，这就意味着，对于大型而尖端的设计，除了花钱购买高端综合工具外别无选择。

417

25.7 FPGA开发板

如果一个设计公司决定涉足物理硬件领域，就必须有FPGA开发板。

OpenCores（见25.5节）提供少量的FPGA开发板项目，但是大多数设计人员更愿意购买专业的FPGA开发板。

好的一面是，花在开发板上的钱可能在其他方面能节省下来。例如，一个聪明的工程师能够将一块小小的FPGA评估板转变成一台强大的逻辑分析仪（这听起来像是一个有潜力的OpenCores项目）。

25.8 综合材料

其他一些可能会让人感兴趣的零星材料总结如下：

- www.easics.be单击“WebTools”链接可以找到一个CRC工具，允许你选择标准或者客户化的多项式并产生相应的Verilog或者VHDL模块。
- www.linuxeda.com，Linux平台上的EDA工具。