第 1 章

FPGA基本情况介绍

FPGA（Field Programmable Gate Array，现场可编程门阵列）属于可编程逻辑器件的一种，在 20 世纪 90 年代获得突飞猛进的发展，经过将近 30 年的发展，到目前已成为实现数字系统的主流平台之一。本章主要介绍 FPGA 的基本历史、基本结构、应用领域及其最新进展，使初学者对 FPGA 能够有基本的了解。

1.1　FPGA 简史

本节从 FPGA 与 ASIC、CPLD 的区别，Altera 与 Xilinx 的区别，Verilog 与 VHDL 等方面对 FPGA 进行简要介绍。

1.1.1　FPGA 与 ASIC

1. FPGA

FPGA 是在 PAL、GAL、CPLD 等可编程器件的基础上进一步发展的产物。它是作为专用集成电路（Application Specific Integrated Circuit，ASIC）领域中的一种半定制电路而出现的，既解决了定制电路的不足，又克服了原有可编程器件门电路数有限的缺点。它是当今数字系统设计的主要硬件平台，主要特点是完全由用户通过软件进行配置和编程，从而完成某种特定的功能，且可反复擦写。在修改和升级时，不需要额外改变 PCB 电路板，只是在计算机上修改和更新程序，使硬件设计工作成为软件开发工作，缩短系统设计的周期，提高实现的灵活性并降低成本。

FPGA 的特点：加电时，FPGA 芯片将 EPROM 中的数据读入片内编程 RAM 中，配置完成后，FPGA 进入工作状态。掉电后，FPGA 恢复成白片，内部逻辑关系消失，因此，FPGA 能够反复使用。

理论上，FPGA 允许无限次的编程。

FPGA 的编程无须专用的 FPGA 编程器，只需用通用的 EPROM、PROM 编程器即可。FPGA 内部有丰富的触发器和 I / O 引脚，能够快速成品，不需要用户介入芯片的布局布线和工艺问题，

而且可以随时改变逻辑功能，使用灵活。

2. ASIC

ASIC 是应特定用户要求和特定电子系统的需要而设计、制造的集成电路。用一句话总结就是市场上买不到的芯片。苹果的 A 系列处理器就是典型的 ASIC。

ASIC 是定制的，具体分为全定制和半定制。

- 全定制设计可以实现最小面积、最佳布线布局、最优功耗速度积，得到最好的电特性。全定制设计的特点是精工细作，设计要求高、成本高、周期长。
- 半定制设计方法又分为基于标准单元的设计方法 CBIC（Cell Based IC）和基于门阵列的设计方法。半定制主要适合开发周期短、低开发成本、投资风险小的小批量数字电路设计。

ASIC 的特点：面向特定用户的需求，量身定制，执行速度较快。ASIC 在批量生产时与通用集成电路相比具有体积小、功耗低、可靠性高、性能高、保密性强、成本低等优点。ASIC 需要较长的开发周期，风险较大，一旦有问题，就会导致成片全部作废，所以小公司已经玩不起了。ASIC 的基本结构如图 1.1 所示。

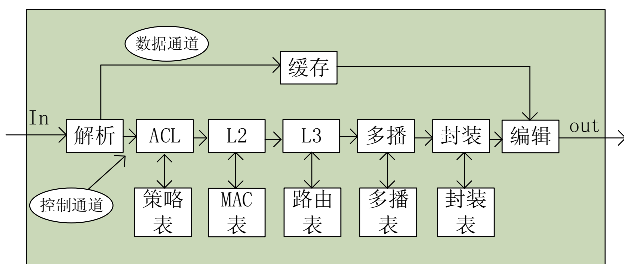

图 1.1　ASIC 基本架构

近年来人工智能受到的关注越来越多，许多公司正在积极开发能实现移动端人工智能的硬件，尤其是能够结合未来的物联网应用。移动端人工智能的实现方法有两大流派：FPGA 流派和 ASIC 流派。FPGA 流派的代表有 Xilinx 主推的 Zynq 平台，而 ASIC 流派的代表有 Movidius。下面来分析 FPGA 与 ASIC 的具体区别在哪里。

3. FPGA 和 ASIC 的区别

（1）设计流程

图 1.2 所示为 FPGA 和 ASIC 设计流程。

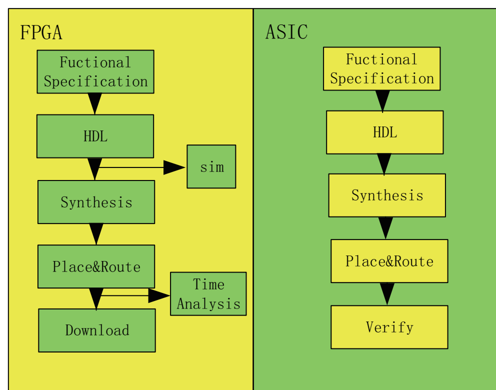

图 1.2　FPGA 和 ASIC 设计流程

- **FPGA：** 完整的 FPGA 设计流程包括功能描述、电路设计与输入、功能仿真、综合优化、综合后仿真、实现与布局布线、时序仿真、板级仿真与验证、调试与加载配置。
- **ASIC：** ASIC 的设计流程（数字芯片）包括功能描述、模块划分、模块编码输入、模块级仿真验证、系统集成和系统

仿真验证、综合、STA（静态时序分析）、形式验证。

说　明

在 ASIC 设计过程中，往往要用 FPGA 进行原型验证。FPGA 验证是进行 ASIC 设计的重要环节，其后还需要引入 ASIC 版本源码，插入 IO PAD、DFT，进行功耗估计和其他后端流程。完成 FPGA 验证可以就完成了 ASIC 整套流程的 50%～80%。

从设计成本来考虑：小批量时，FPGA 占优；大批量时，ASIC 占优。

FPGA 本身就是一个芯片，只是我们可以通过编程的方式修改内部逻辑连接和配置实现自己想要的功能。实现 ASIC 就如从一张白纸开始，必须得有代码，之后综合，然后布局、布线，最后得到 GDSII（一种时序提供格式）后去流片。

（2）速度

相同的工艺和设计，在 FPGA 上的速度应该比 ASIC 慢。因为 FPGA 内部基于通用的结构，也就是 LUT（Look Up Table），可以实现加法器、组合逻辑等。而 ASIC，一般加法器就是加法器，比较器就是比较器，FPGA 结构上的通用性必然导致冗余。另外，FPGA 的基本单元是 LUT（LUT 组成 Slice，Slice 组成 CLB，这是 Xilinx 的结构），为此，大的设计假如一个 LUT 实现不了，就得用两个 LUT，一个 Slice 实现不了，就要用 CLB，不同结构处于特定的位置，信号之间的互联导致的 wire delay 是不可忽略的一部分。而对于 ASIC 来说没有结构上的限制，而且对于特定的线路布局来说可以在空间上靠得很近，相对来说 wire delay 和 cell delay 都应该比 FPGA 小。当然，LUT 中也有 DFF，作为高速的设计，一般都会在一个简单的组合逻辑操作之后打一拍，再做下一步的处理。

提　示

delay（延迟）分为 cell delay 和 wire delay。cell delay 是指元器件内部的 delay，wire delay 是器件互连 Pin-to-Pin 的 delay。

（3）体积

如果结构完全相同，那么 FPGA 会被 ASIC 远远踢飞。FPGA 要规模大得多才能实现 ASIC 相同的功能，主频还只有几分之一。

（4）功耗

在相同工艺条件下，FPGA 的功耗要大于 ASIC。尤其是基于占用大量硅面积的、每个单元 6 个晶体管的静态存储器（SRAM）的查寻表（LUT）和配置元件技术的 FPGA，其功耗要比对等的 ASIC 大得多。

（5）成本

FPGA 贵在单片，开发工具费用和硬件损耗风险基本不存在。ASIC 贵在流片的费用和开发工具。NRE（Non-Recurring Engineering，一次性工程）费用随着工艺的提高变得相当贵，除非芯片一次成功后即可量产，否则单片费用将奇贵无比！

（6）其他方面

ASIC 用于大型项目，而对于需要快速投放市场且支持远程升级的小型项目，FPGA 则更为适合。FPGA 技术的主要优势仍是产品投放市场的时间较短。

ASIC 的优势在于加电后可立即运行，就单位逻辑大小而言，封装选择更多，还可以包括某些模拟逻辑。与此相对比，FPGA 加载配置进入存储器需要时间，因此不能立即工作。此外，FPGA 的封装也较复杂。

FPGA 内部还包括接口 I/O。I/O 分为普通 I/O 和高速 I/O。高速 I/O 支持高速的 SERDES 等，用于实现 XAUI、PCI-e 等高速接口，这些接口动辄几 Gbps。此外，种类多种多样的硬核 IP 也是各 FPGA 厂商差异化竞争的利器，例如 POWERPC、ARM 等硬核 IP，从而构成 CPU+FPGA 于一体的集可编程性和可重构的处理平台。

（7）两者的定位

FPGA 和 ASIC 产品的使用要根据产品的定位和设计需要来选择。ASIC 产品适用于设计规模特别大（如 CPU、DSP 或多层交换芯片等）或者应用于技术非常成熟且利润率非常低的产品（如家用电器和其他消费类电器）以及大量应用的通用器件（如 RAM、PHY 等）。FPGA 产品适用于设计规模适中、产品要求快速占领市场或产品需要灵活变动的特性设计等方面的产品，如 PDH、2.5GB 以下的 SDH 设备和大部分的接口转换芯片等。当然，具体选择哪种产品来设计还要设计者充分考虑自己的产品定位来决定。

（8）两者在互相融合

最明显的莫过于处理器中开始集成 FPGA，而可编程的 ASIC 也开始兴起。随着 SoC（系统级芯片或系统）成为主流，两者的边界也就不那么明显了。

（9）总结

总的来说，就如同 GPU 和 CPU 一样：GPU 可以非常快速地处理图像，但是要处理其他的东西，GPU 则有些困难。CPU 能处理很多的运算，也能处理图像，只是慢而已。一旦你是冲着某个目的去的（ASIC），最快速的实现方式就可以。如果想要多方面兼顾（FPGA），就不可能在每一个方面都做到最好。在使用时，你必须权衡利弊。

1.1.2 FPGA 与 CPLD

CPLD（Complex Programmable Logic Device，复杂可编程逻辑器件）是从 PAL 和 GAL 器件发展出来的器件，相对而言规模大、结构复杂，属于大规模集成电路范围，是一种用户根据各自需要而自行构造逻辑功能的数字集成电路。其基本设计方法是借助集成开发软件平台，用原理图、硬件描述语言等方法生成相应的目标文件，通过下载电缆（“在系统”编程）将代码传送到目标芯片中来实现设计的数字系统。

FPGA 和 CPLD 的区别如下：

（1）CPLD 更适合完成各种算法和组合逻辑，FPGA 更适合完成时序逻辑。换句话说，FPGA 更适合触发器丰富的结构，而 CPLD 更适合触发器有限而乘积项丰富的结构。

（2）CPLD 的连续式布线结构决定了它的时序延迟是均匀和可预测的，而 FPGA 的分段式布线结构决定了其延迟的不可预测性。

（3）在编程上，FPGA 比 CPLD 具有更大的灵活性。CPLD 通过修改具有固定内连电路的逻辑功能来编程，FPGA 主要通过改变内部连线的布线来编程；FPGA 可在逻辑门下编程，而 CPLD 是在逻辑块下编程的。

（4）FPGA 的集成度比 CPLD 高，具有更复杂的布线结构和逻辑实现。

（5）CPLD 比 FPGA 使用起来更方便。CPLD 的编程采用 E2PROM 或 FastFlash 技术，无须外部存储器芯片，使用简单。而 FPGA 的编程信息需存放在外部存储器上，使用方法复杂。

（6）CPLD 的速度比 FPGA 快，并且具有较大的时间可预测性。这是由于 FPGA 是门级编程，并且 CLB 之间采用分布式互联，而 CPLD 是逻辑块级编程，并且其逻辑块之间的互联是集总式的。

（7）在编程方式上，CPLD 主要是基于 E2PROM 或 FastFlash 存储器编程的，编程次数可达 1 万次，优点是系统断电时编程信息也不丢失。CPLD 又可分为在编程器上编程和在系统编程两类。FPGA 大部分是基于 SRAM 编程的，编程信息在系统断电时丢失，每次上电时，需从器件外部将编程数据重新写入 SRAM 中。其优点是可以编程任意次，可在工作中快速编程，从而实现板级和系统级的动态配置。

（8）CPLD 保密性好，FPGA 保密性差。

（9）一般情况下，CPLD 的功耗要比 FPGA 大，且集成度越高越明显。

随着复杂可编程逻辑器件（CPLD）密度的提高，数字器件设计人员在进行大型设计时，既灵活又容易，而且产品可以很快进入市场。许多设计人员已经感受到 CPLD 有容易使用、时序可预测和速度高等优点。然而，过去由于受到 CPLD 密度的限制，他们只好转向 FPGA 和 ASIC。现在，设计人员可以体会到密度高达数十万门的 CPLD 所带来的好处。

1.1.3 Altera 与 Xilinx

要比较 Xilinx 和 Altera 公司的 FPGA，就要清楚两个大厂 FPGA 的结构，由于设计不同，两家的 FPGA 结构、参数也各不相同，但可以统一到 LUT 查找表上。

1. Altera 芯片

（1）逻辑阵列模块（LAB）

逻辑阵列模块的主要结构是 8 个适应逻辑模块（ALM），还包括一些进位链和控制逻辑等结构。每个 ALM 中都包含了两个可编程的寄存器、两个专用全加器、一个进位链、一个共享算术链和一个寄存器链。

（2）存储器模块（RAM）

StratixII 器件具有 TriMatrix 存储结构，包括 3 种大小的嵌入式 RAM 块。TriMatrix 存储器包括 512 位的 M512 块、4KB 的 M4K 块和 512KB 的 M-RAM 块，每个都可以配置支持各种特性。

（3）数字信号处理模块（DSP）

DSP 块结构是为实现多种最大性能和最小逻辑资源利用率的 DSP 功能而优化的。

（4）锁相环（PLL）

StratixII 器件具有 12 个锁相环和 48 个独立系统时钟，可以作为中央时钟管理器以满足系统时序需求。

2. Xilinx 芯片

Xilinx 的 FPGA 主要由可配置逻辑块（CLB）、时钟管理模块（CMT）、存储器（RAM/FIFO）、

数字信号处理模块（DSP）和一些专用模块组成。

（1）可配置逻辑块（CLB）

可配置逻辑块是 Xilinx 的基本逻辑单元。s7 系列的每个 CLB 包含两个 Slice，每个 Slice 由 4 个（A、B、C、D）6 输入 LUT 和 8 个寄存器 REG 组成。

（2）时钟管理模块（CMT）

时钟管理模块用于产生高质量的时钟。以 Virtex-5 系列器件为例，CMT 包括两个数字时钟管理单元（DCM）和一个锁相环电路（PLL）。

（3）存储器（RAM/FIFO）

现代 Xilinx 的 FPGA 都有内部的存储器块。以 Virtex-5 为例，内部包含若干块 RAM，每一块 36KB，并且 RAM 的大小可以灵活配置。Virtex-5 内的 RAM 是同步的双口 RAM，并且可以配置为多速率的 FIFO（First In First Out，先进先出）存储器，极大地提高了设计的灵活性。

（4）数字信号处理模块（DSP）

大多数的 FPGA 产品都提供了 DSP。

（5）一些专用模块

除了上述模块外，在现代的 Xilinx 的 FPGA 产品中还有一些专用模块，例如 Rocket IO 千兆位级收发器、PCI-e 端点模块和三态以太网 MAC 模块等。

Xilinx 与 Altera 的 FPGA 结构最大的不同是逻辑单元部分：Xilinx 的逻辑单元基本组成为可配置逻辑模块（CLB），而 Altera 的为 LAB，更深一层讲，CLB 和 LAB 里面也都是由 LUT、触发器等构成的。两个公司的 FPGA 组成各有特点，这也决定了它们的 FPGA 产品在功能上各有特点。

现在的 Altera 已经成被 Intel 收购，Xilinx 和 Intel 正积极开拓通信军工以外的市场，目前已经在机器学习领域占据一席之地，无论面向服务器领域做 Training（训练）的 high-end（可理解为高端）芯片，还是面对终端领域（如汽车电子），IoT（Internet of Things，物联网）设备的低功耗芯片都有不错的应用前景。FPGA 也跟随摩尔定律在规模上不断发展，不论是逻辑还是片内存储，现在的 FPGA 也不再是仅仅擅长定点运算，大量浮点 DSP 已经开始使用，加上原有 IO 方面的优势，相信可以和 GPU 一较高下。

最后，两家 FPGA 都在向异构架构发展，Xilinx 主推 ARM+FPGA 的架构，新的 UltraScale MPSOC 更是配有 4 核 A53、Mail400 GPU、cortex-R5 实时处理器；Altera 也出过 ARM 的 SOC，但是现在被 Intel 收购了，会不会投入 x86 的怀抱还是未知数，目前其发展路线图并不如 Xilinx 那么清晰。

1.1.4 Verilog 与 VHDL

Verilog HDL 是业界普遍采用的一种硬件描述语言，用于从算法级、门级到开关级的多种抽象设计层次的数字系统建模过程。被建模的数字系统对象的复杂性可以介于简单的门和完整的电子数字系统之间。数字系统能够按层次描述，并可在相同描述中显式地进行时序建模。

Verilog HDL 从 C 编程语言中继承了多种操作符和结构，其语法规则与 C 语言非常相似，尽管二者有着本质上的区别。考虑到绝大多数的数字设计工程师都应该熟悉 C 语言，因此 Verilog 语言的入门相比 VHDL 语言更为简单。

Verilog HDL 语言不仅定义了语法，还对每个语法结构定义了清晰的模拟、仿真语义。因此，用这种语言编写的模型能够使用 Verilog 仿真器进行验证。Verilog HDL 提供了扩展的建模能力，其中许多扩展最初很难理解。但是，Verilog HDL 语言的核心子集非常易于学习和使用，这对大多数建模应用来说已经足够。当然，完整的硬件描述语言足以对从最复杂的芯片到完整的电子系统进行描述。

VHDL 具有以下特点：

（1）功能强大，设计灵活

VHDL 具有功能强大的语言结构，可以用简洁明确的源代码来描述复杂的逻辑控制。它具有多层次的设计描述功能，层层细化，最后可直接生成电路级描述。VHDL 支持同步电路、异步电路和随机电路的设计，这是其他硬件描述语言所不能比拟的。VHDL 还支持各种设计方法，既支持自底向上的设计，又支持自顶向下的设计；既支持模块化设计，又支持层次化设计。

（2）支持广泛，易于修改

由于 VHDL 已经成为 IEEE 标准所规范的硬件描述语言，目前大多数 EDA（Electronics Design Automation，电子设计自动化）工具几乎都支持 VHDL，这为 VHDL 的进一步推广和广泛应用奠定了基础。

（3）强大的系统硬件描述能力

VHDL 具有多层次的设计描述功能，既可以描述系统级电路，又可以描述门级电路。描述既可以采用行为描述、寄存器传输描述或结构描述，也可以采用三者混合的混合级描述。另外，VHDL 支持惯性延迟和传输延迟，可以准确地建立硬件电路模型。VHDL 支持预定义和自定义的数据类型，给硬件描述带来较大的自由度，使设计人员能够方便地创建高层次的系统模型。

（4）独立于器件的设计，与工艺无关

设计人员用 VHDL 进行设计时，不需要首先考虑选择完成设计的器件，就可以集中精力进行设计的优化。当设计描述完成后，可以用多种不同的器件结构来实现其功能。

（5）很强的移植能力

VHDL 是一种标准化的硬件描述语言，同一个设计描述可以被不同的工具所支持，使得设计描述的移植成为可能。

（6）易于共享和复用

VHDL 采用基于库（Library）的设计方法，可以建立各种可再次利用的模块。这些模块可以预先设计或使用以前设计中的存档模块，将这些模块存放到库中，就可以在以后的设计中进行复用，使设计成果在设计人员之间进行交流和共享，减少硬件电路设计。现在，VHDL 和 Verilog 作为 IEEE 的工业标准硬件描述语言，又得到众多 EDA 公司的支持，在电子工程领域已成为事实上的通用硬件描述语言。有专家认为，在新的世纪中，VHDL 与 Verilog 语言将承担起大部分的数字系统设计任务。

提　示
具体采用 Verilog 还是 VHDL 作为设计语言并不重要。

其实作为一个成熟的数字设计工程师，Verilog 和 VHDL 都应该是熟悉的，最低的要求应该是能够读懂一种，熟练掌握另一种进行设计。硬件描述语言只是数字系统的设计工具，虽然掌握工具对于成功进行数字系统设计很重要，但是更为重要的是对于数字设计的基本原理和理论

的学习，只有深入掌握了数字系统设计的基本原理和理论，才能设计出符合实际需求的数字系统，只有在这样的前提下，学习设计工具才是有意义的。

1.2 FPGA 芯片（Xilinx）介绍

本节主要谈谈 Xilinx 的 FPGA 芯片的结构。目前，FPGA 芯片仍是基于查找表技术的，但其概念和性能已经远远超出查找表技术的限制，并且整合了常用功能的硬核模块（如块 RAM、时钟管理单元 MMCM、DSP 硬核乘加器等）。图 1.3 所示为 Xilinx 公司推出的最新系列 FPGA，采用的是 ASMBL（Advanced Silicon Modular Block）架构。在 ASMBL 架构中，每类资源以列形式存在。

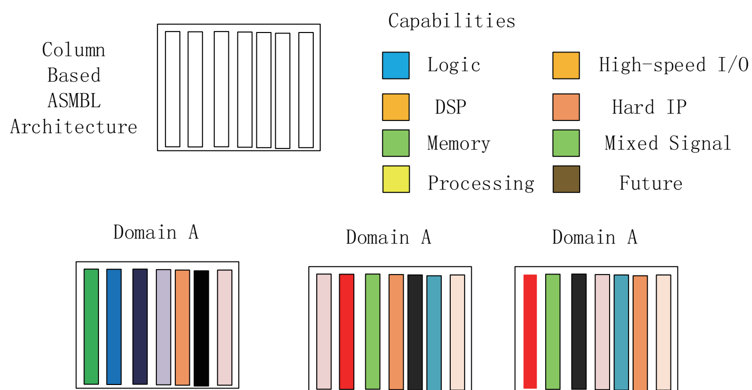

图 1.3 ASMBL 架构

1.2.1 FPGA 的基本结构

图 1.4 所示为 Xilinx 公司 Spartan-2 系列 FPGA 的内部结构图，从中可以看出 FPGA 芯片主要由 7 部分组成：可编程输入输出单元、基本可编程逻辑单元、完整的时钟管理、嵌入块式 RAM、丰富的布线资源、内核的底层功能单元和内嵌专用硬件模块。

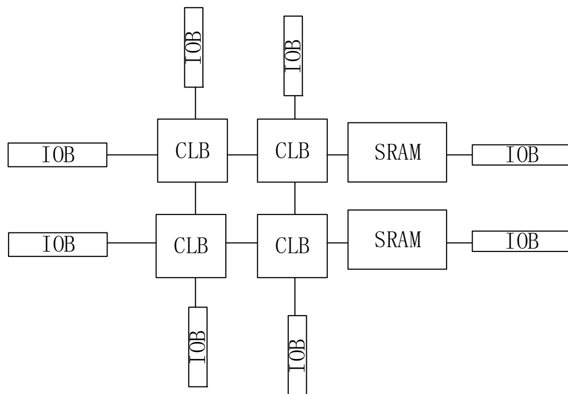

图 1.4 FPGA 芯片的内部结构

注 意
Virtex 5 后续的 Virtex 系列、Spartan 6 后续的 Spartan 系列均为 6 输入的 LUT，之前的 FPGA 系列均为 4 输入的 LUT。

每个模块的功能如下。

1. 可编程输入输出单元（IOB）

可编程输入/输出单元简称 I/O 单元，是芯片与外界电路的接口部分，完成不同于电气特性下对输入/输出信号的驱动与匹配要求，其典型结构如图 1.5 所示。为了便于管理和适应多种电气标准，FPGA 的 IOB 被划分为若干个 bank，每个 bank 的接口标准由其接口电压 VCCO 决定，一个 bank 只能有一种 VCCO，但不同的 bank 的 VCCO 可以不同。只有相同电气标准的端口才能连接在一起，VCCO 电压相同是接口标准的基本条件。

FPGA 内的 I/O 按组分类，每组都能够独立地支持不同的 I/O 标准。通过软件的灵活配置，可适配不同的电气标准与 I/O 物理特性、调整驱动电流的大小、改变上拉电阻与下拉电阻。目前，I/O 的频率越来越高，一些高端的 FPGA 通过 DDR 寄存器技术可以支持高达 2Gb/s 的数据速率。

外部输入信号可以通过 IOB 模块的存储单元输入到 FPGA 的内部，也可以直接输入 FPGA 内部。当外部输入信号经过 IOB 模块的存储单元输入到 FPGA 内部时，其保持时间（hold time）的要求可以降低，通常默认为 0。IO 块内部结构如图 1.5 所示。

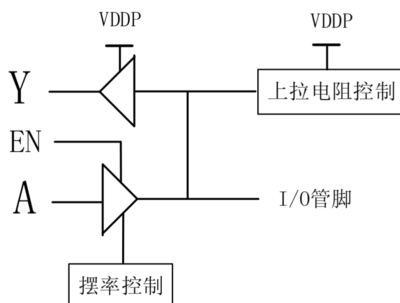

图 1.5　IO 块内部结构示意图

2. 可配置逻辑块（CLB）

CLB 是 FPGA 内的基本逻辑单元。CLB 的实际数量和特性会依据器件的不同而不同，但是每个 CLB 都包含一个可配置开关矩阵，此矩阵由 4 或 6 输入、一些选型电路（多路复用器等）和触发器组成。开关矩阵是高度灵活的，可以对其进行配置，以便处理组合逻辑、移位寄存器或 RAM。在 Xilinx 公司的 FPGA 器件中，CLB 由多个（一般为 4 或 2 个）相同的 Slice 和附加逻辑构成，如图 1-6 所示。每个 CLB 模块不仅可以用于实现组合逻辑、时序逻辑，还可以配置为分布式 RAM 和分布式 ROM。典型的 CLB 结构如图 1.6 所示。

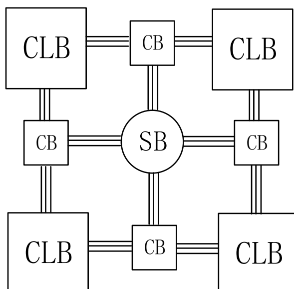

图 1.6　典型的 CLB 结构示意图

Slice 是 Xilinx 公司定义的基本逻辑单位。基于 4 输入 LUT 的传统 Slice 内部结构如图 1.7 所示。一个 Slice 由两个 4/6 输入的查找表函数、进位逻辑、算术逻辑、存储逻辑和函数复用器组成。算术逻辑包括一个异或门（XORG）和一个专用与门（MULTAND），一个异或门可以使一个 Slice 实现 2bit 全加操作，专用与门用于提高乘法器的效率；进位逻辑由专用进位信号和函数复用器（MUXC）组成，用于实现快速的算术加减法操作；4 输入函数发生器用于实现 4 输入 LUT、分布式 RAM 或 16 比特移位寄存器。典型的 4 输入 Slice 结构如图 1.7 所示。

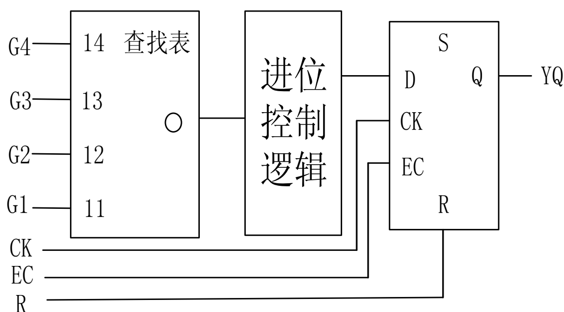

图 1.7　典型的 4 输入 Slice 结构示意图

目前的 FPGA 一般都采用 6 输入查找表，可以实现 6 输入 LUT 或 64 比特移位寄存器，进位逻辑包括两条快速进位链，用于提高 CLB 模块的处理速度。

基于 6 输入 LUT 的 Slice 内部结构包括 4 个 6 输入的 LUT 和 8 个寄存器，因此新一代的 6 输入 Slice，其逻辑能力从资源上讲是“老”器件的 4 倍（$2^6/2^4=4$）。

3. 时钟管理模块

业内大多数 FPGA 均提供数字时钟管理（赛灵思公司的全部 FPGA 均具有这种特性）。赛灵思公司推出先进的 FPGA 提供数字时钟管理和相位环路锁定。相位环路锁定能够提供精确的时钟综合，且能够降低抖动，并实现过滤功能。PLL 原理图如图 1.8 所示。

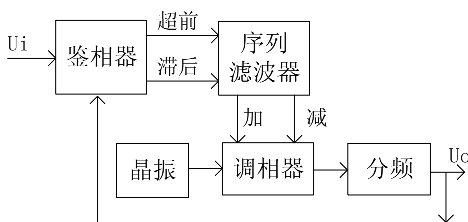

图 1.8　PLL 原理图

4．嵌入式块 RAM（BRAM）

嵌入式 RAM 模块概况如图 1.9 所示。

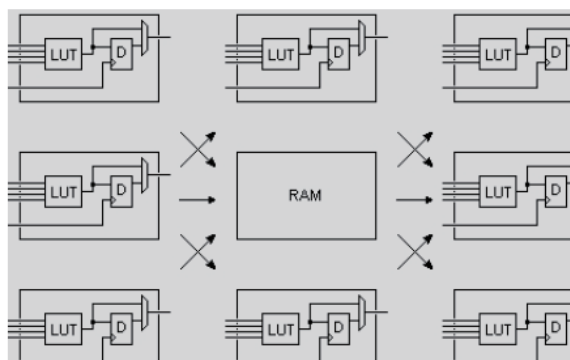

图 1.9　内嵌的块 RAM

大多数 FPGA 都具有内嵌的块 RAM，这大大拓展了 FPGA 的应用范围和灵活性。块 RAM 可被配置为单端口 RAM、双端口 RAM、内容地址存储器（CAM）以及 FIFO 等常用存储结构。RAM、FIFO 是比较普及的概念，在此就不冗述了。CAM 存储器在其内部的每个存储单元中都有一个比较逻辑，写入 CAM 中的数据会和内部的每一个数据进行比较，并返回与端口数据相同的所有数据的地址，因而在路由的地址交换器中有广泛的应用。除了块 RAM 外，还可以将 FPGA 中的 LUT 灵活地配置成 RAM、ROM 和 FIFO 等结构。在实际应用中，芯片内部块 RAM 的数量也是选择芯片的一个重要因素。

目前，6 输入 LUT 器件中的单片块 RAM 的容量为 36K 比特，即位宽为 36 比特，深度为 1024 个，可以根据需要改变其位宽和深度，但要满足两个原则：

- 首先，修改后的容量不能大于 36K 比特。
- 其次，位宽最大不能超过 72 比特。

当然，可以将多片块 RAM 级联起来形成更大的 RAM，此时只受限于芯片内块 RAM 的数量，而不再受上面两个原则的约束。

5. 丰富的布线资源

布线资源连通 FPGA 内部的所有单元，而连线的长度和工艺决定着信号在连线上的驱动能力和传输速度。FPGA 芯片内部有着丰富的布线资源，根据工艺、长度、宽度和分布位置的不同而划分为 4 个不同的类别。

- 第一类是全局布线资源，用于芯片内部全局时钟和全局复位/置位的布线。
- 第二类是长线资源，用以完成芯片 Bank 间的高速信号和第二全局时钟信号的布线。
- 第三类是短线资源，用于完成基本逻辑单元之间的逻辑互连和布线。
- 第四类是分布式的布线资源，用于专有时钟、复位等控制信号线。

FPGA 芯片的性能差异主要就是布线资源数量的差异，高端 FPGA 具有最好的布线资源，能够实现更高速的设计，更容易达到时序收敛。

在实际设计中，设计者不需要直接选择布线资源，布局布线器可自动根据输入逻辑网表的

拓扑结构和约束条件选择布线资源来连通各个模块单元。从本质上讲，布线资源的使用方法和设计的结果有密切、直接的关系。

6. 底层内嵌功能单元

内嵌功能模块主要是指 DLL（Delay Locked Loop）、PLL（Phase Locked Loop）、DSP 等软处理核（Soft Core）。现在越来越丰富的内嵌功能单元使得单片 FPGA 成为系统级的设计工具，使其具备了软硬件联合设计的能力，逐步向 SOC 平台过渡。

DLL 和 PLL 具有类似的功能，可以完成时钟高精度、低抖动的倍频和分频，以及占空比调整和移相等功能。Xilinx 公司生产的芯片上集成了 PLL 和 DLL，Altera 公司的芯片上集成了 PLL，Lattice 公司的新型芯片上同时集成了 PLL 和 DLL。PLL 和 DLL 可以通过 IP 核生成的工具方便地进行管理和配置。DLL 的结构如图 1.10 所示。

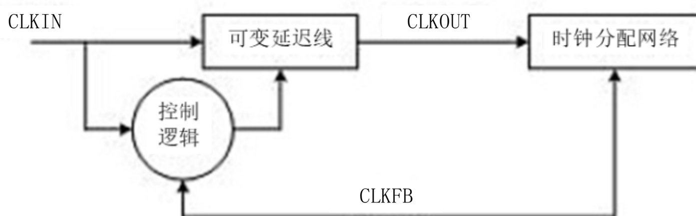

图 1.10　典型的 DLL 模块示意图

7. 内嵌专用硬核

内嵌专用硬核是相对底层嵌入的软核而言的，指 FPGA 处理能力强大的硬核（Hard Core），等效于 ASIC 电路。为了提高 FPGA 性能，芯片生产商在芯片内部集成了一些专用的硬核。例如，为了提高 FPGA 的乘法速度，主流的 FPGA 中都集成了专用乘法器；为了适用通信总线与接口标准，很多高端的 FPGA 内部都集成了串并收发器（SERDES），可以达到数十吉字节每秒的收发速度。

赛灵思公司的高端产品不仅集成了 Power PC 系列 CPU，还内嵌了 DSP Core 模块，相应的系统级设计工具是 EDK 和 Platform Studio，并依此提出了片上系统（System on Chip）的概念。通过 PowerPC、Miroblaze、Picoblaze 等平台，能够开发标准的 DSP 处理器及其相关应用，达到 SOC 的开发目的。

此外，新推出的赛灵思的 FPGA 系列（如 Virtex-5 LXT）还内建了 PCI-e 和三态以太网 MAC 硬核（TEMAC）。与软核实现方式相比，硬核可以把功耗降低 5~10 倍，节约将近 90% 的逻辑资源。

Xilinx 三态以太网 MAC 核是一个可参数化的核，非常适合在网络设备中使用，例如开关和路由器等。可定制的 TEMAC 核使系统设计者能够实现宽范围的集成式以太网设计，从低成本 10/100Mbps 以太网到性能更高的 1GB 端口。TEMAC 核设计符合 IEEE 802.3 规范的要求，并且可以在 1000Mbps、100 Mbps 和 10 Mbps 模式下运行。另外，它还支持半双工和全双工操作。TEMAC 核通过 Xilinx CORE Generator 工具提供，是 Xilinx 全套以太网解决方案的一部分。

1.2.2　软核、硬核及固核

IP（Intelligent Property）核是具有知识产权核的集成电路芯核的总称，是经过反复验证过的、

具有特定功能的宏模块，与芯片制造工艺无关，可以移植到不同的半导体工艺中。到了 SOC 阶段，IP 核设计已成为 ASIC 电路设计公司和 FPGA 提供商的重要任务，也是其实力体现。对于 FPGA 开发软件，其提供的 IP 核越丰富，用户的设计就越方便，其市场占用率就越高。目前，IP 核已经变成系统设计的基本单元，并作为独立设计成果被交换、转让和销售。

从 IP 核的提供方式上，通常将其分为软核、固核和硬核 3 类。从完成 IP 核所花费的成本来讲，硬核代价最大；从使用灵活性来讲，软核的可复用性最高。

1. 软核

软核在 EDA 设计领域指的是综合之前的寄存器传输级（RTL）模型；具体在 FPGA 设计中指的是对电路的硬件语言描述，包括逻辑描述、网表和帮助文档等。软核只经过功能仿真，需要经过综合以及布局布线才能使用。其优点是灵活性高、可移植性强，允许用户自配置；缺点是对模块的预测性较低，在后续设计中存在发生错误的可能性，有一定的设计风险。软核是 IP 核应用最广泛的形式。

2. 硬核

硬核在 EDA 设计领域指经过验证的设计版图；在 FPGA 设计中指布局和工艺固定、经过前端和后端验证的设计，设计人员不能对其进行修改。不能修改的原因有两个：首先是系统设计对各个模块的时序要求很严格，不允许打乱已有的物理版图；其次是保护知识产权的要求，不允许设计人员对其有任何改动。IP 硬核不许修改的特点使其复用有一定的困难，因此只能用于某些特定应用，使用范围较窄。

目前，所有的 Xilinx FPGA 内部都集成了 CDM（时钟管理单元）、DSP 单元及 BRAM 等硬核资源，高端芯片中还包括 SERDES、PCI-e、DDR3 控制器、ARM/Power PC 以及 XADC 等硬核资源。

3. 固核

固核在 EDA 设计领域指的是带有平面规划信息的网表；具体在 FPGA 设计中可以看作带有布局规划的软核，通常以 RTL 代码和对应具体工艺网表的混合形式提供。将 RTL 描述结合具体标准单元库进行综合优化设计，形成门级网表，再通过布局布线工具即可使用。和软核相比，固核的设计灵活性稍差，但在可靠性上有较大提高。目前，固核也是 IP 核的主流形式之一。

1.2.3 7 系列 FPGA 简介

在 7 系列芯片以前，Xilinx 的两个重要产品分别是面向高性能应用场景下的 Virtex 系列和面向低成本低功耗应用场景下的 Spartan 系列，不过这两个系列从芯片内部的布局布线、时钟管理单元到内部的 BRAM 等硬件模块都有显著的不同。开发人员在不同平台间切换时，因为硬件模块的定义不同，往往需要做一定的代码修改，从而减慢了项目开发的速度。因此，在最新的 7 系列芯片中，Xilinx 采用了统一的架构，FPGA 内硬件例化模块都采用统一的定义，以帮助用户快速完成设计的迁移。

7 系列 FPGA 采用了最新的 28nm 工艺，应用范围更是涵盖了所有的系统要求，从低功耗、小型化、成本敏感、大批量应用到超高连接带宽、逻辑能力强和高性能，各种应用需要的信号

处理能力它都具备。Xilinx 7 系列 FPGA 包括以下几种。

- Artix-7 系列：为最低的成本和功耗做了优化，为大批量应用的小型化封装设计。
- Kintex-7 系列：为最大的性价比做了优化，和前一代相比，提高一倍的性能。
- Virtex-7 系列：为最高的系统性能做了优化，通过硅片堆叠技术（SSI）提供高性能、高容量的芯片。

根据以上的系列分布，可以看到两个显著的变化：

其一，低成本、低功耗的产品名为 Artix，而不是原来的 Spartan。

其二，在原来高性能和低成本中间的产品线中增加了实现最大性价比的 Kintex 产品线，以往使用 Virtex 系列的产品都可以尝试先在 Kintex 器件上做实验。同时，因为芯片内部硬件架构相同，所以三个产品线的读音也都类似。

7 系列 FPGA 有以下几个主要的特征：

- 基于 6 输入查找表（LUT）技术的先进的高性能 FPGA 逻辑，并且可配置为分布式存储器。
- 带有内建 FIFO 的 36Kb 双端口 BRAM，可作为片内数据缓存。
- 支持速度高达 1866Mb/s 的 DDR3 接口的高性能 Select IOTM 技术。
- 内建多个吉比特（Gb）收发器的高速串行连接器，速度从 600Mb/s 到最高 28.05Gb/s，提供低功耗模式，优化了芯片到芯片的接口。
- 用户可配置的模拟接口（XADC）包括两个 12 位 1MSPS 的 ADC 和片上温度、电压传感器。
- DPS Slice 包括 25×18 的乘法器、48 位累加器和预加法器，可用于高性能滤波，其中包括优化的对称滤波。
- 强大的时钟管理模块 CMT，连接相位锁相环 PLL 和混合模式时钟管理器 MMCM，可用于高精度和低抖动的应用。
- 集成了用于 x8 Gen3 端点和根端口设计的 PCI-e 硬核。
- 丰富的可配置选项，包括支持商用存储器，带有 HMAC/SHA-256 授权的 256 位 AES 加密及内建的 SEU 检测和纠错单元。
- 高性能低功耗的 28nm 工艺。HKMG、HPL 处理，1.0V 核心电压处理技术和 0.9V 核心电压选项用于更低的功耗应用。

7 系列 FPGA 和 Zynq AP SoC 平台具有千丝万缕的联系。Zynq 内部的 FPGA 部分就是基于 7 系列 FPGA 的。两个面向低端应用的 Zynq 芯片 Zynq-7010 和 Zynq-7020 使用 Artix-7，两个面向高端应用的 Zynq 芯片 Zynq-7030 和 Zynq-7045 使用 Kintex-7。

说　明
本书并不详细介绍 7 系列 FPGA 内部的结构，如果读者对 FPGA 的结构、资源和设计方法感兴趣，请到 Xilinx 官网参考官方的文档，或者阅读其他关于 FPGA 内部组织结构的图书。

下面我们会简单介绍 FPGA 的组成单元和内部的资源，主要包括 CLB、Slices、LUTs、BRAM、DSP Slices、PCI-e 接口、XADC。（先列一张资源清单，让大家有了解一下。）7 系列 FPGA 资源如图 1.11 所示。

Max. Capability	Spartan-7	Artix-7	Kintex-7	Virtex-7
Logic Cells	102K	215K	478K	1,955K
Block RAM(1)	4.2 Mb	13 Mb	34 Mb	68 Mb
DSP Slices	160	740	1,920	3,600
DSP Performance(2)	176 GMAC/s	929 GMAC/s	2,845 GMAC/s	5,335 GMAC/s
Transceivers	–	16	32	96
Transceiver Speed	–	6.6 Gb/s	12.5 Gb/s	28.05 Gb/s
Serial Bandwidth	–	211 Gb/s	800 Gb/s	2,784 Gb/s
PCIe Interface	–	x4 Gen2	x8 Gen2	x8 Gen3
Memory Interface	800 Mb/s	1,066 Mb/s	1,866 Mb/s	1,866 Mb/s
I/O Pins	400	500	500	1,200
I/O Voltage	1.2V–3.3V	1.2V–3.3V	1.2V–3.3V	1.2V–3.3V
Package Options	Low-Cost, Wire-Bond	Low-Cost, Wire-Bond, Lidless Flip-Chip	Lidless Flip-Chip and High-Performance Flip-Chip	Highest Performance Flip-Chip

图 1.11　7 系列 FPGA 资源

图 1.11 比较清楚地列出了 7 系列 FPGA 的资源，下面我们简单介绍 7 系列的关键技术。

（1）硅片堆叠（SSI）技术

要设计出高容量的 FPGA，是有很多挑战的，Xilinx 的解决方案是使用 SSI 技术。SSI 技术允许使用多个超级逻辑区域（SLR）组成被动式内插层（Passive Interposer Layer），使用经过行业领导者验证的制造和装配技术。在单 FPGA 上集成了多达 10 000 个 SLR 连接，提供具有低延迟和低功耗的超高带宽。

（2）逻辑资源（CLB、Slice 和 LUT）

7 系列 FPGA 的 LUT 可以被配置为一个 6 输入 1 输出的 LUT，或者两个 5 输入具有相同地址或逻辑输入的独立输出的 LUT。每一个 LUT 的输出可以有选择性地在 FF（Flip-Flop）寄存。4 个这样的 LUT 和与之相关的 8 个 FF 再加上一些多路选择器和算术进位逻辑组成了 Slice，两个 Slice 组成了 CLB。25%～50%的 Slice 可以用 LUT 作为分布式 64 位 RAM 或者 32 位移位寄存器。CLB 的一些关键特性如下：

- 真正 6 输入 LUT。
- LUT 具有存储功能。
- 具有寄存器和移位寄存器功能。

说　明

详细的 CLB 配置情况可从官网上下载 UG474“7 Series FPGAs Configurable Logic Block User Guide”，以供参考。

（3）时钟管理

7 系列 FPGA 拥有多达 24 个时钟管理通道（CMT），每个 CMT 都由一个混合模式时钟管理器（MMCM）和一个锁相环（PLL）组成。7 系列 FPGA 提供 6 种不同类型的时钟线（BUFG、BUFR、BUFIO、BUFH、BUFMR 和高性能时钟）来解决不同的需求，包括高扇出、短传输延迟和低抖动等需求。这一部分是 FPGA 的关键部分，可参考官方的 UG472“7 Series FPGAs Clocking Resources User Guide”了解详情。

（4）BRAM（块存储器）

大多数的 FPGA 都具有内嵌的 BRAM，这加强了 FPGA 的灵活性。BRAM 可用于片内数据

缓冲、FIFO 缓冲等。7 系列的 FPGA 提供了 30~1880 个双端口 BRAM，每一个存储 36Kb 的数据。每个 BRAM 都有两个只共享数据的独立端口。BRAM 的特性包括：

- 双端口 36Kb 的 BRAM，最大位宽是 72 位。
- 可编程的 FIFO 逻辑。
- 内建可选的纠错电路。

（5）DSP Slice

7 系列的 FPGA 集成了专用的、充分定制的、低功耗的 DSP Slice。其增强的功能主要包括：

- 25×18 的补码乘法器/累加器，高分辨率为 48 位的信号处理器。
- 用于对称滤波器应用的低功耗预加法器。
- 一些高级特性（可选的流水线、可选的 ALU 和用于级联的专用总线）。

（6）输入/输出（I/O）块

7 系列的 FPGA 的 I/O 块使用不同的封装，满足物理和逻辑级别上不同的要求。物理级上，I/O 块支持一个范围内的驱动电压和驱动强度，以及接收功能接口的不同 I/O 标准。7 系列的 FPGA 有高性能 HP 和高范围 HR 两种类型 I/O。逻辑级上，所有的 I/O 都能被配置为组合或者寄存方式，都支持 DDR 模式。

（7）低功耗吉比特收发器

7 系列的 FGPA 提供的吉比特收发器具有以下特征：

- 提供高达 6.6Gb/s（GTP）、12.5Gb/s（GTX）、13.1Gb/s（GTH）或 28.05Gb/s（GTZ）线速率，是业内第一个实现 400Gb/s 数据吞吐的单芯片。
- 支持芯片到芯片接口的低功耗模式。
- 支持高级预发送、后加重、接收器线性（CTLE）和判决反馈均衡（DFE），包括用于额外余量的自适应均衡。

（8）集成 PCI-e 模块

7 系列的 FPGA 集成的 PCI-e 模块包括如下重要的特性：

- 兼容 PCI-e 2.1 或 3.0 基本规范，提供端点和根端口的能力。
- 根据芯片类别，支持 Gen1（2.5Gb/s）、Gen2（5Gb/s）和 Gen3（8Gb/s）。
- 支持高级配置选项、高级错误报告（AER）（包括端到端的（ECRC）高级错误报告）和 ECRC 特性。
- 具有多重功能和单个启动 I/O 虚拟化（SR-IOV）支持。

（9）XADC 模块

全部的 Xilinx 7 系列 FGPA 都集成了新的灵活模拟接口 XADC。XADC 结合了 7 系列 FPGA 的可编程能力，满足板级数据采集和监控的需求。这种独特的组合被称为敏捷的混合信号的模拟和可编程逻辑。欲了解更多信息，可访问 http://www.xilinx.com/ams。XADC 模块的主要特性包括：

- 双 12 位 1MSPS 模拟到数字转换（ADC）。
- 高达 17 个灵活的用户可配置逻辑输入。

- 可选片内或者片外参考电压。
- 片内温度（最大误差±4℃）和电压（最大误差±1%）传感器。

1.3 FPGA 的应用领域

FPGA 从最初的逻辑黏合、接口互联的角色逐渐成为大规模计算硬件平台之一。FPGA 未来的发力点分别在机器学习领域、5G 无线通信领域、嵌入式视觉领域、工业物联网领域、云计算平台领域五大方面。

1.3.1 机器学习

要想明白“深度学习”“机器学习”需要什么样的硬件，必须了解深度学习的工作原理。首先在表层上，我们有一个巨大的数据集，并选定了一种深度学习模型。每个模型都有一些内部参数需要调整，以便学习数据。而这种参数调整实际上可以归结为优化问题，在调整这些参数时，就相当于在优化特定的约束条件。

百度的硅谷人工智能实验室（SVAIL）已经为深度学习硬件提出了 Deep Bench 基准。这一基准着重衡量的是基本计算的硬件性能，而不是学习模型的表现。这种方法旨在找到使计算变慢或低效的瓶颈。因此，重点在于设计一个对于深层神经网络训练的基本操作执行效果最佳的架构。那么基本操作有哪些呢？现在的深度学习算法主要包括卷积神经网络（CNN）和循环神经网络（RNN）。基于这些算法，Deep Bench 提出以下 4 种基本运算：

- 矩阵相乘（Matrix Multiplication） 几乎所有的深度学习模型都包含这一运算，它的计算十分密集。
- 卷积（Convolution） 这是另一个常用的运算，占用了模型中大部分的每秒浮点运算（浮点／秒）。
- 循环层（Recurrent Layers） 模型中的反馈层，并且基本上是前两个运算的组合。
- All Reduce 这是一个在优化前对学习到的参数进行传递或解析的运算序列。在跨硬件分布的深度学习网络上执行同步优化时（如 AlphaGo 的例子），这一操作尤其有效。

除此之外，深度学习的硬件加速器需要具备数据级别和流程化的并行性、多线程和高内存带宽等特性。另外，由于数据的训练时间很长，硬件架构必须低功耗，因此效能功耗比（Performance per Watt）是硬件架构的评估标准之一。

1. GPU VS CPU

在如今的深度学习平台上，CPU 面临着一个很尴尬的处境：它很重要又不是太重要。它很重要，是因为它依旧是主流深度学习平台的重要组成部分，机器学习领域的专家吴恩达曾利用 16000 颗 CPU 搭建了当时世界上最大的人工神经网络 Google Brain 并利用深度学习算法识别出了“猫”，又比如名震一时的 AlphaGo 就配置了多达 1920 颗 CPU。

但是它又不是太重要，相比于其他硬件加速工具，传统的 CPU 在架构上就有着先天的弱势。图 1.12 所示为 CPU 和 GPU 的区别。

图 1.12 CPU VS GPU

图 1.12 是 CPU 与 GPU 内部结构的对比。总体上来说，二者都是由控制器（Control）、寄存器（Cache、DRAM）和逻辑单元（Arithmetic Logic Unit，ALU）构成的，但是三者的比例却有很大的不同。在 CPU 中，控制器和寄存器占据了结构中很大一部分；与之相反，在 GPU 中，逻辑单元的规模则远远超过其他二者之和。

说 明
这种不同的构架就决定了 CPU 在指令的处理/执行、函数的调用上有着很好的发挥，但由于逻辑单元所占比重较小，相对于 GPU 而言，在数据处理方面（算术运算或者逻辑运算）的能力就弱了很多。

GPU 进行数据处理的过程可以描述成：GPU 从 CPU 处得到数据处理的指令，把大规模、无结构化的数据分解成很多独立的部分，然后分配给各个流处理器集群。每个流处理器集群再次把数据分解，分配给调度器所控制的多个计算核心，同时执行数据的计算和处理。如果一个核心的计算算作一个线程，那么在这颗 GPU 中就有 32×4×16=2048 个线程同时进行数据的处理。

尽管每个线程/Core 的计算性能、效率与 CPU 中的 Core 相比低了不少，但是当所有线程都并行计算时，累加之后它的计算能力又远远高于 CPU。对于基于神经网络的深度学习来说，它的硬件计算精度要求远远没有对其并行处理能力的要求来的迫切。而这种并行计算能力转化为对于硬件的要求就是尽可能大的逻辑单元规模。通常我们使用每秒钟进行的浮点运算（Flops/s）来量化参数。不难看出，对于单精度浮点运算，GPU 的执行效率远远高于 CPU。

除了计算核心的增加外，GPU 另一个比较重要的优势就是它的内存结构。

（1）共享内存。在 NVIDIA 披露的性能参数中，每个流处理器集群末端设有共享内存。相比于 CPU 每次操作数据都要返回内存再进行调用，GPU 线程之间的数据通信不需要访问全局内存，而在共享内存中就可以直接访问。这种设置带来最大的好处就是线程间通信速度的提高。

（2）高速的全局内存（显存）。目前 GPU 上普遍采用 GDDR5 的显存颗粒，不仅具有更高的工作频率，从而带来更快的数据读取/写入速度，还具有更大的显存带宽。我们认为在数据处理中，速度往往最终取决于处理器从内存中提取数据以及流入和通过处理器要花多少时间。

在传统的 CPU 构架中，尽管有高速缓存（Cache）的存在，但是由于其容量较小，大量的数据只能存放在内存（RAM）中。进行数据处理时，数据要从内存中读取，然后在 CPU 中运算，最后返回内存中。由于构架的原因，二者之间的通信带宽通常在 60Gb/s 徘徊。与之相比，大显存带宽的 GPU 具有更大的数据吞吐量。在大规模深度神经网络的训练中，必然带来更大的优势。

目前而言，越来越多的深度学习标准库支持基于 GPU 的深度学习加速，通俗点描述就是深

度学习的编程框架会自动根据 GPU 所具有的线程/Core 数去自动分配数据的处理策略，从而达到优化深度学习的时间，而这些软件上的全面支持也是其他计算结构所欠缺的。

2. FPGA 的潜在优势

“CPU+GPU”或者“MIC”的计算模型被广泛地应用于各种深度学习中。CPU 与 GPU 都是利用现有的成熟技术去提供一种通用级的解决方法来满足深度学习的要求，尽管 Intel 与 NVIDIA 不断推出了 KNL 和 Pascal 等系列加速芯片来助阵深度学习，但这仅仅是大公司对于深度学习的一种妥协，并不是一种针对性的专业解决方案。

目前，在深度学习模型的训练领域使用的基本是 SIMD（Single Instruction Multiple Data，单指令多数据流架构）计算，即只需要一条指令就可以平行处理大批量数据。但是，在平台完成训练之后，它还需要进行推理环节的计算。这部分的计算更多的是 MISD（Multiple Instruction Single Data，多指令流单数据流）。比如讯飞语音输入法在同一时间要对数以百万计的用户的语音输入进行识别并转化为文字输出。

在这个阶段，它们的作用远不如训练阶段那么得心应手。未来，至少 95%的深度学习都用于推断，尤其是在移动端，只有不到 5%的深度学习用于模型训练。因此，寻找低功耗、高性能、低延时的加速硬件成了当务之急。在这种情况下，人们把目光投向了 FPGA 与 ASIC。

与 GPU/CPU 相比，FPGA 与 ASIC 拥有良好的运行能效比，在实现相同性能的深度学习算法中，GPU 所需的功耗远远大于 FGPA 与 ASIC。浪潮与 Intel 于 2016 年底推出了 FPGA 加速卡 F10A，是目前最高性能的加速卡，单芯片峰值运算能力达到 1.5TFlops，功耗才 35W，每瓦特功率为 42GFlops，是 GPU 的数倍之高。其次，对于 SIMD 计算，GPU/CPU 尽管具有很多逻辑核心，但是受限于冯诺依曼结构，无法发挥其并行计算的特点。FPGA 与 ASIC 不仅可以做到并行计算，还能实现流水处理。这大大减小了输入与输出的延时比。

针对移动端的深度学习，FPGA 或者 ASIC 更多的会以 SOC 形式出现，以更好地优化神经网络结构，从而提升效率。

1.3.2 5G 无线

图 1.13 展示了 Xilinx 5G 应用及对应解决方案示意图，FPGA 在推动 5G 发展中有很大的作用。

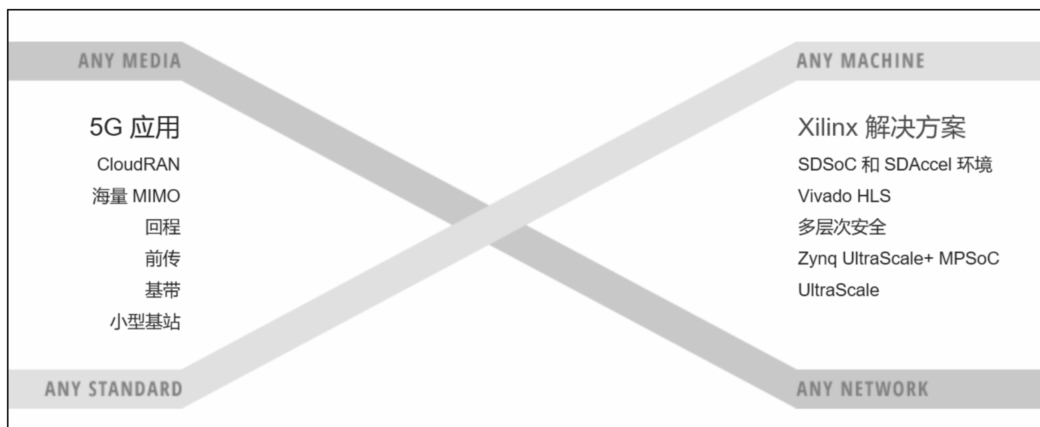

图 1.13 Xilinx 5G 应用及对应解决方案示意图

到 2020 年，预计将有超过 500 亿的互联设备。因此，5G 网络必须具有更高的可扩展性、智能性和异构性。分布式小型基站、支持数百个天线的海量 MIMO 以及通过 CloudRAN 进行的集中式基带处理等技术将显著增大覆盖范围与数据吞吐量。网络将需要通过回程及光前传来进行安全连接，以完成处理。

对于 5G 而言，FPGA 的技术正在帮助解决容量、连接以及性能挑战，并灵活支持多标准、多频带和多子网络，实现多样物联网驱动的 5G 应用。

只有 FPGA 能够提供这样一个灵活的、基于标准的解决方案，融软件可编程能力、多标准、多频带硬件优化为一体，并将任意（any-to-any）连接以及保密性等功能紧密结合在一起，以满足 5G 网络的需求。客户可在 Zynq MPSoC 和 UltraScale FPGA 芯片平台上使用 Vivado HLS、SDSoC 和 SDAccel 软件定义环境快速开发其应用。Xilinx 推出业界首款带有 RF 级模拟技术的 RF SoC 器件，可实现高性能 ADC/DAC 和 SD-FEC 与 SoC 的突破性集成。

1.3.3 嵌入式视觉

图 1.14 所示为 Xilinx 嵌入式视觉应用及对应解决方案的内容。

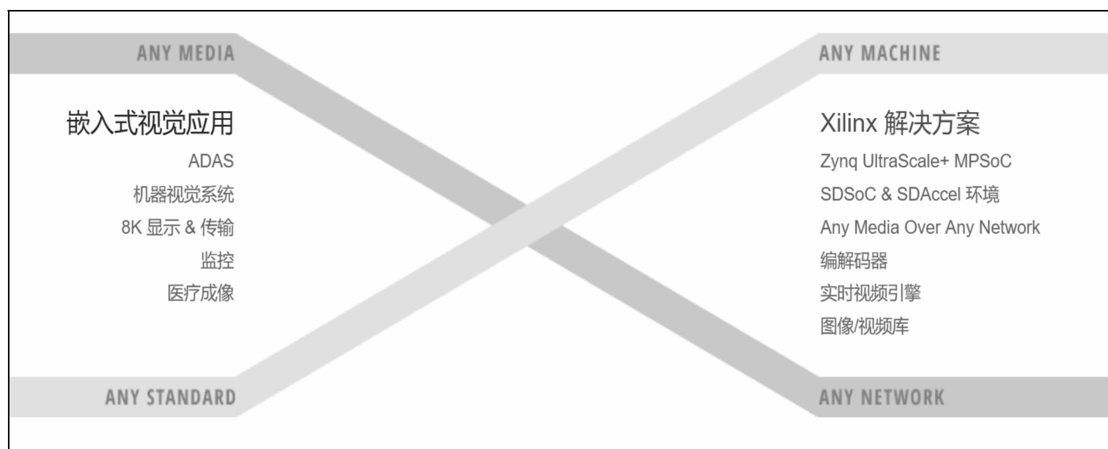

图 1.14　Xilinx 嵌入式视觉应用及对应解决方案示意图

1. 嵌入式的发展前景

当前嵌入式设计发展迅猛，有很多需求无法被现有的产品满足。现有的产品包括单个处理器、单个 ASIC、ASSP 或者单纯的 FPGA 方案，甚至这些方案的组合也都无法满足需求。根据市场调查显示，全新的嵌入式应用（比如汽车驾驶员辅助系统、智能视频监视、先进的工业控制、先进的工业遥测、先进的工业制导、企业级的工作站、广播级的摄像机、多功能打印机、航空航天电子等应用）设计随市场需求的旺盛，将成为新的利润增长点。

深入了解下一代嵌入式处理器的需要，我们会发现提高性能、减少成本、降低功耗、缩小外形、增加灵活性是主要需要。现有方案的局限性也是相当明显的，例如现有的微处理器缺乏足够的信号处理能力，需要多芯片方案来搭建下一代的处理器系统。多芯片方案成本较高，功耗大，占用更多的空间，不利于缩小外形，同时如果采用现有的 ASIC 或 ASSP 方案，更新换代的速度或者说差异化的能力都会受限，不能适应快速变化的需求，也很难提供差异化的竞争优

势，这些局限性使我们看到下一代处理器需要解决和应对的问题。在市场需求推动下，为了满足下一代应用处理的需求，Xilinx 定义了 Zynq-7000 系列产品，并首先推出了 4 款（7010、7020、7030 和 7045）。它们都是高性能和低功耗的处理器平台，是灵活和可扩展的解决方案。

下面我们一起来初步了解 Zynq-7000 AP SoC 平台的 ARM+FPGA 体系结构。此体系结构的左上角方块就是 ARM 部分，在 Zynq-7000 AP SoC 平台里称为 Processing System（PS）。外面包着它的是 FPGA 部分，称为 Programmable Logic（PL）。FPGA 部分就是我们上一节介绍的 7 系列 FPGA，内部的资源和结构与 7 系列 FPGA 一样。

Zynq-7000 系列的亮点在于它包含完整的 ARM 处理子系统，每一颗 Zynq-7000 系列的处理器都包含双核的 CortexTM-A9 处理器，整个处理器的搭建都以处理器为中心，而且处理器子系统中集成了内存控制器和大量的外设，使 CortexTM-A9 的核在 Zynq-7000 中完全独立于可编程逻辑单元，也就是说，如果暂时没有用到 FPGA 的部分，ARM 处理器的子系统也可以独立工作，这与以前的 FPGA 处理器有本质区别，其是以处理器为中心的。

另外，在可编程逻辑部分紧密地与 ARM 的处理单元相结合。FPGA 的部分用于扩展子系统，有丰富的扩展能力，有 3000 多个内部互连，连接资源非常丰富，可提供 100Gb/s 以上的内部带宽。此外，在 I/O 接口方面，FPGA 的优点是 I/O 可以充分自定义，并在 FPGA 部分集成高速串行口（Multi Gigabit Transceiver）。同时，在 FPGA 内也继承了模数转换器（XADC）。

2. Xilinx 嵌入式视觉解决方案

Xilinx 可为嵌入式视觉开发人员提供一系列支持软硬件设计的技术。Xilinx 器件包括 FPGA、SoC 以及 MPSoC。

Xilinx 的 Vivado HLx 设计环境可帮助硬件及平台开发商开发最新嵌入式视觉硬件。这些工具支持业界最新高带宽传感器接口。Xilinx 包括 SDSoC 在内的 SDx 工具有助于软件及算法开发人员在基于 Eclipse 的熟悉环境中采用 C、C++和 OpenCL 等计算机语言进行开发。

Xilinx reVISION 堆栈建立在 SDx 概念基础之上，支持 OpenCV 和机器学习推断，从而支持 AlexNet、GoogLeNet、SqueezeNet、SSD 和 FCN 等最普及的神经网络以及构建定制神经网络（CNN/DNN）所需的功能元件。同时，该堆栈还允许设计团队将预定义的优化 CNN 实现方案用于网络层。这得到了加速型 OpenCV 函数的有力补充，支持计算机视觉处理。

1.3.4 工业物联网

图 1.15 所示为 Xilinx 工业物联网应用及对应解决方案示意图。

工业物联网（IIoT）正在推动第 4 轮工业革命。它极大地改变着制造、能源、交通运输、城市、医疗以及其他工业行业。大部分专家认为 IIoT 时代已悄然来到，带来了实实在在可测量的业务影响。IIoT 可帮助企业从传感器收集、聚集和分析数据，从而最大限度地提高机器效率以及整个工作的吞吐量。应用包括运动控制、机器与机器通信、预测性维护、智能能源/电网、大数据分析以及智能互联医疗系统等。

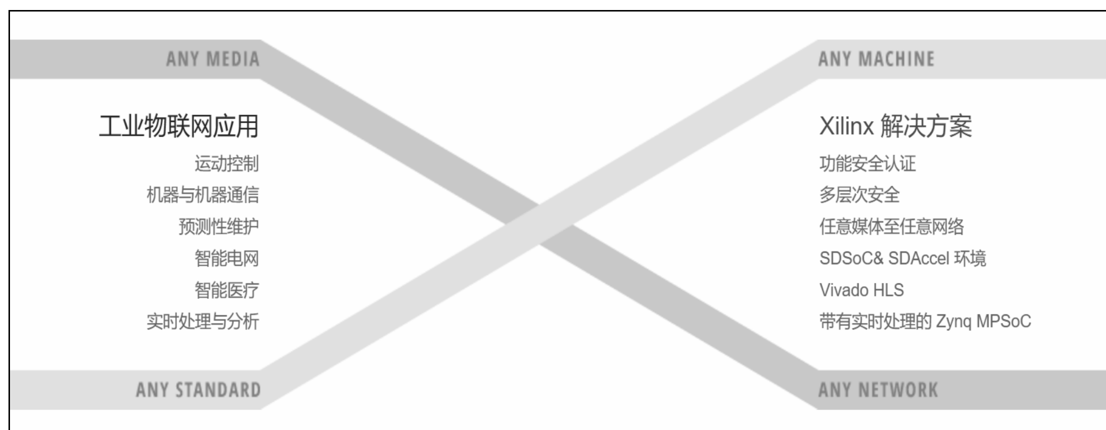

图 1.15　Xilinx 工业物联网应用及对应解决方案示意图

Xilinx 提供了面向工业物联网的灵活标准化解决方案，包括全可编程、实时处理、硬件优化和针对保密和安全的“任意”互联。Xilinx SDAccel、SDSoC 和 Vivado HLS 可帮助客户快速开发更智能的互连差异化应用。

1.3.5　云计算

图 1.16 所示为 Xilinx 云计算应用及对应解决方案示意图。

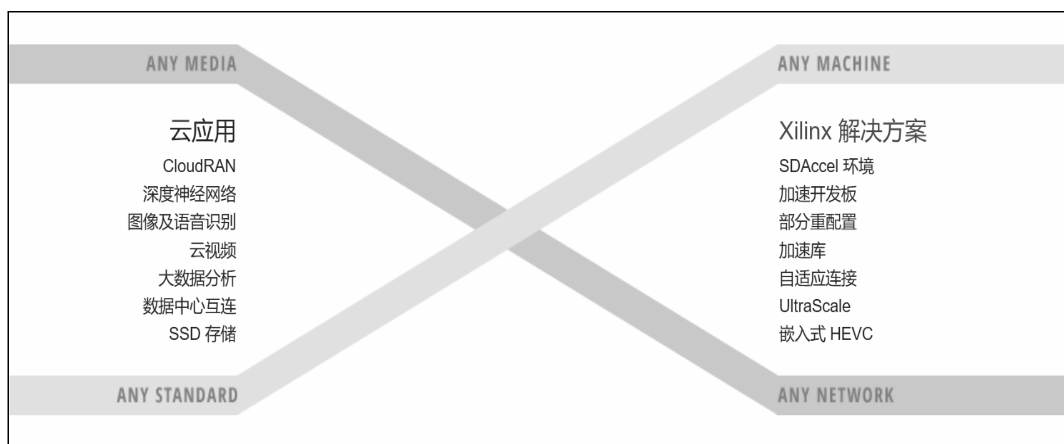

图 1.16　Xilinx 云计算应用及对应解决方案示意图

数据中心需要工作负载优化，以迅速适应来自广泛虚拟化软件应用不断变化的吞吐量、延迟和功率需求。这些应用包括机器学习、视频转码、图像和语音识别、CloudRAN 和大数据分析，以及存储与网络加速、高度灵活的高性能连接。

Xilinx 可使服务器通过工作负载优化，以类似 CPU 和 GPU 方案的 1/10 时延及功耗实现 10 倍吞吐量。应用通过 OpenCL、C 和 C++等语言编写。灵活、基于标准的解决方案融软件可编程能力、负载优化和高性能数据中心互联为一体，并与保密性紧密结合在一起，以满足下一代云计算的需求。

Xilinx 产品系列包含 UltraScale 技术，其可作为可扩展、可重新配置的加速平台，针对任何

工作负载按需进行优化。Xilinx 软件定义开发环境 SDAccel 可帮助客户使用 OpenCL、C 和 C++ 的任意组合来开发其独特应用。SDAccel 部署独有的架构优化型编译器和部分重新配置技术，这两者的结合可通过灵活的运行时功能提供最高质量的结果。

1.3.6 FPGA 硬件加速平台

1. 算法加速

卷积神经网络（Convolutional Neural Network，CNN）是一种典型的多层神经网络，是首个真正意义上成功训练多个层次网络的结构模型。权值共享的网络结构使之在图像处理、语音识别等领域有着重要的应用价值。CNN 算法通常是在 CPU 或 GPU 上以软件编程的方式进行实现的，方法虽简单却无法发挥 CNN 并行性的特点、训练速度慢。FPGA 含有丰富的计算资源，基于 SRAM 结构的 FPGA 能够在运行过程中对片上的资源进行重新配置，实现系统逻辑功能的切换、提高系统的灵活性和资源利用率。

对生物大分子进行结构预测是生物信息学研究的重要领域，具有重大的理论和应用价值。随着基因测序技术的不断发展，RNA 和蛋白质序列数据库的规模急剧膨胀，现有结构测定技术无法满足不断增长的序列数据库的发展需求，基于序列信息对生物大分子的空间结构进行预测成为生物信息学研究的热点。

生物大分子的结构预测算法计算复杂性高，对数据的处理能力提出了更高的要求，迫切需要高性能计算的支持。基于 FPGA 平台的算法加速器是高性能计算研究的重要方向之一，能较好地适应生物信息学算法特征多样性的特点，在提高生物信息学算法性能方面具有广阔的应用前景。

2. 数据库加速

FPGA 领域研发需要软件与硬件两方面的支持。其中，硬件部分包括 FPGA 开发板与相应下载器，软件部分包括 FPGA 设计过程中对应的 EDA 工具。FPGA 芯片数据库是实现软、硬件协同工作的纽带，实现一种简明、精准的芯片数据库描述方法是 FPGA 高效开发的一项重要内容，同时对所描述的芯片数据库信息实现简洁、高效管理是至关重要的。

针对现存芯片数据库中存在的描述结构单一、描述信息管理效率低的问题，研究人员提出了一种新型芯片结构描述与管理方法，提高了 FPGA 芯片数据库信息的准确性，并实现了芯片数据库信息的高效率智能化管理。

1.4 总　　结

本章就此结束，主要对 FPGA 的基本结构、用途等进行了详细的解释，初学者可通过此部分对 FPGA 有初步的认识。下一章对 FPGA 软件安装和语法等基础知识进行介绍。

第 2 章

FPGA设计基础知识

本章将对 FPGA 的基础知识进行介绍，包括软件的下载安装、相关软件进行关联，以便更好地使用，还会涉及 Verilog 语言特性的介绍和使用等。通过本章的学习，初学者可以顺利安装 FPGA 相关软件，使这些软件成为学习 FPGA 的尖兵利器。同时，Verilog 语言的介绍会为第 3、4、5 章 FPGA 的设计打下基础。

2.1 软件下载及安装

本节主要对 FPGA 相关软件的安装方法进行介绍。初学者根据此安装步骤即可顺利安装软件。

2.1.1 ISE 下载及安装

步骤 01 可以在 ISE 官网上下载软件，下载后单击 xsetup 安装文件，直接进行安装，如图 2.1 所示。

图 2.1 单击 xsetup 进行安装

步骤 02 安装程序开始运行后，一直单击 Next 按钮，如图 2.2 所示，直到出现如图 2.3 所示的界面。

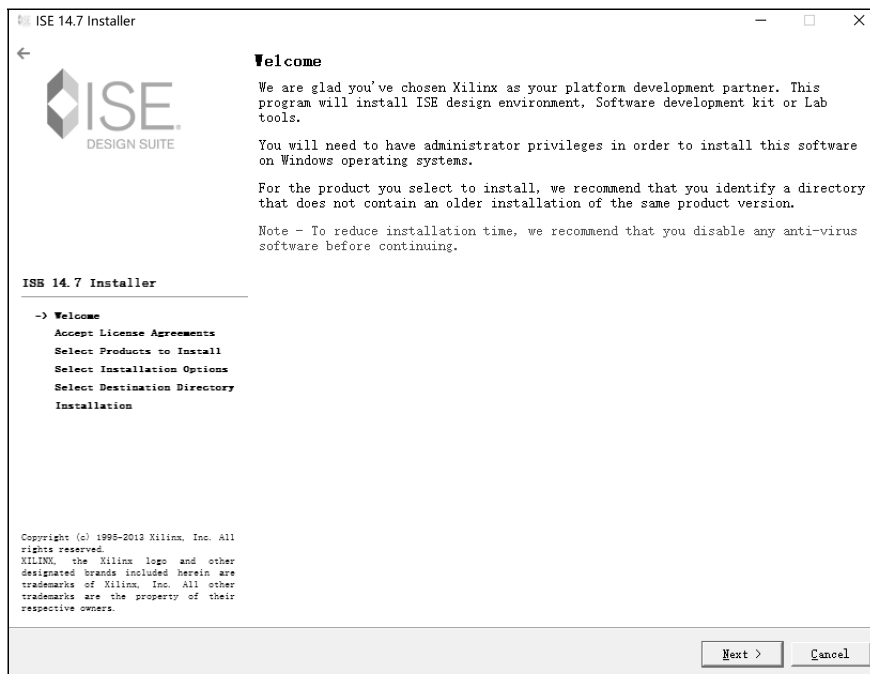

图 2.2 单击 Next 按钮

步骤 03 勾选图 2.3 中的两个选项，单击 Next 按钮。

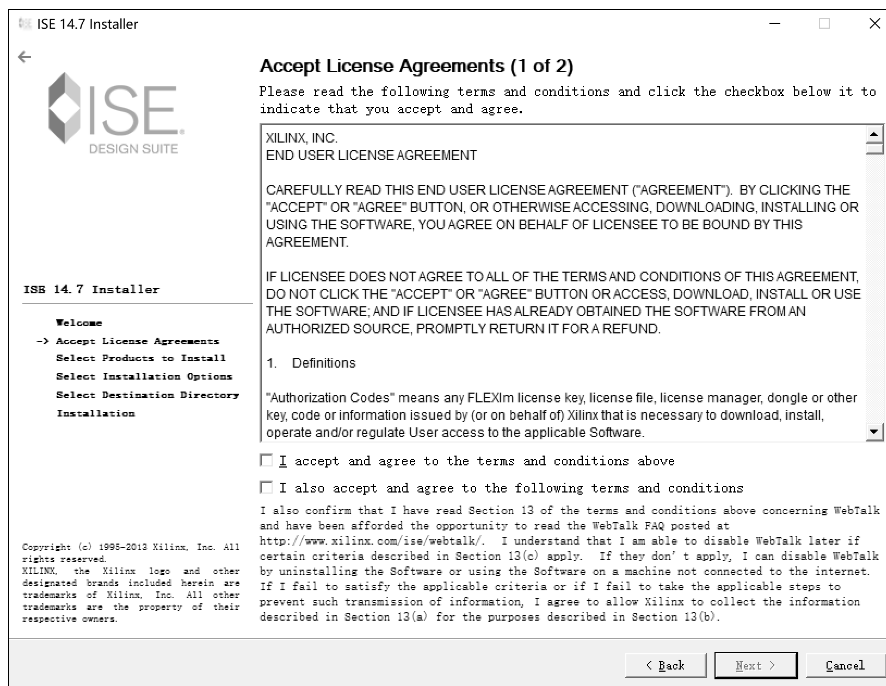

图 2.3 将两个选项勾选并单击 Next 按钮

步骤04 出现如图 2.4 所示的界面时，选择 ISE Design Suite System Edition 选项，单击 Next 按钮。

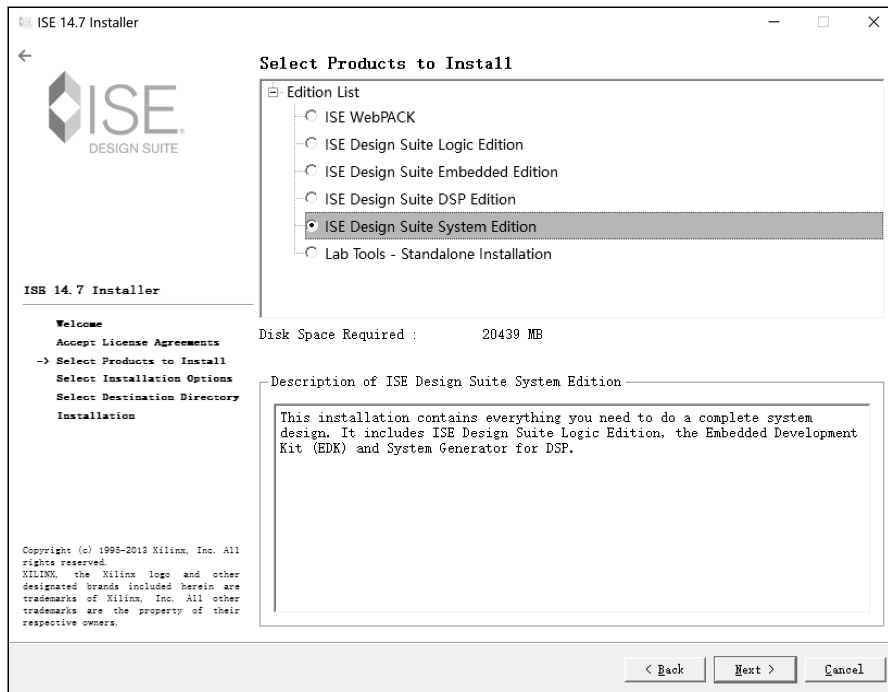

图 2.4 选择 ISE Design Suite System Edition 选项

步骤05 出现如图 2.5 所示的界面时，勾选所有选项，然后单击 Next 按钮。

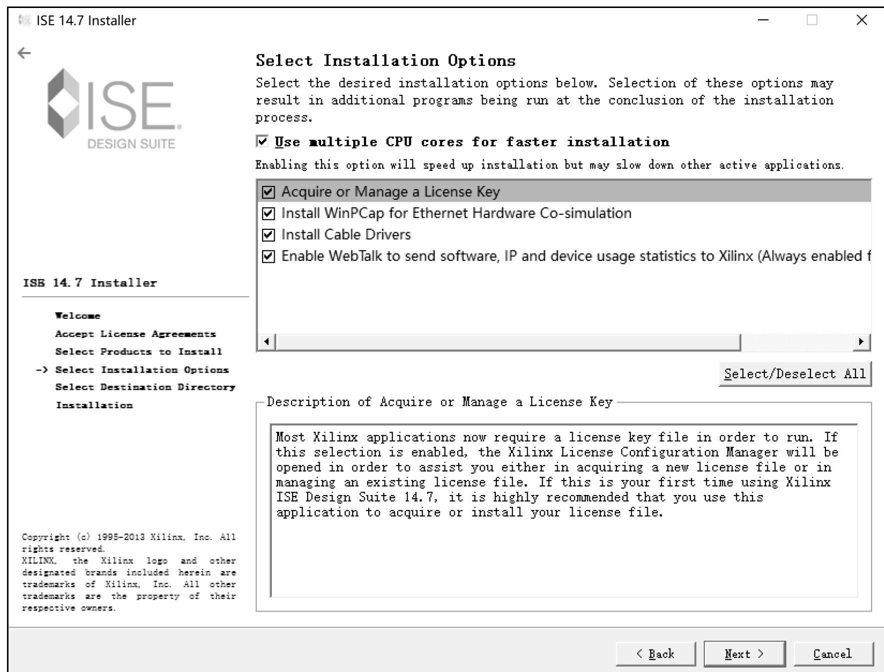

图 2.5 勾选所有选项

步骤 06 出现如图 2.6 所示的界面时选择创建目录等情况。此时最好选择在 C 盘中安装，并选中所有工具。

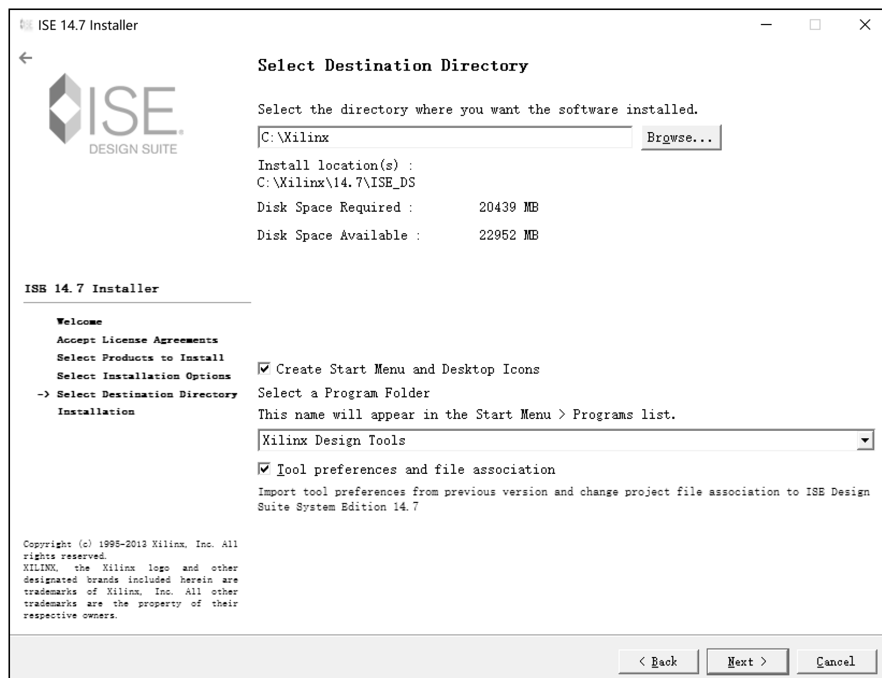

图 2.6 选择创建目录等情况

步骤 07 配置完环境后，出现如图 2.7 所示的安装界面，此时单击 Install 按钮即可进行安装。

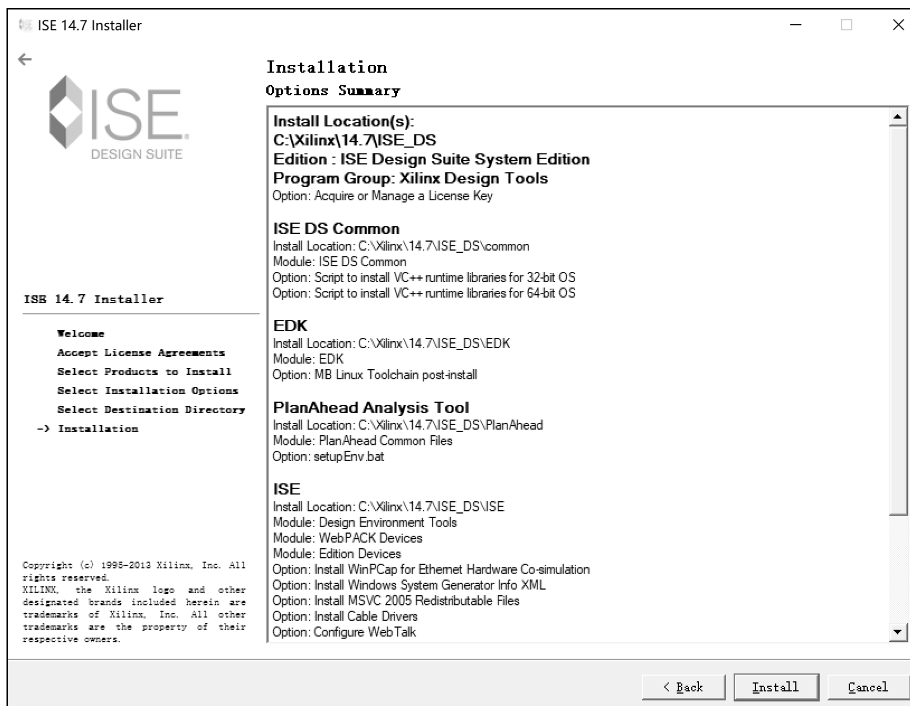

图 2.7 单击 Install 按钮进行安装

步骤08 安装完成后，桌面上出现如图 2.8 所示的图标。

图 2.8　安装完成

2.1.2　ModelSim 下载及安装

步骤01 下载 ModelSim 10.5，如图 2.9 所示，自行选择 32 位或 64 位。（本节以 64 位为例进行介绍。）

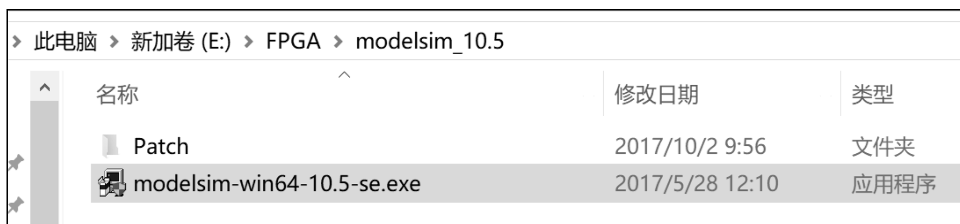

图 2.9　选择 32 位或者 64 位软件

步骤02 选择安装路径，之后一直单击 Next 按钮直至安装结束。选择默认安装路径 C:\modeltech64_10.5，如图 2.10 所示。

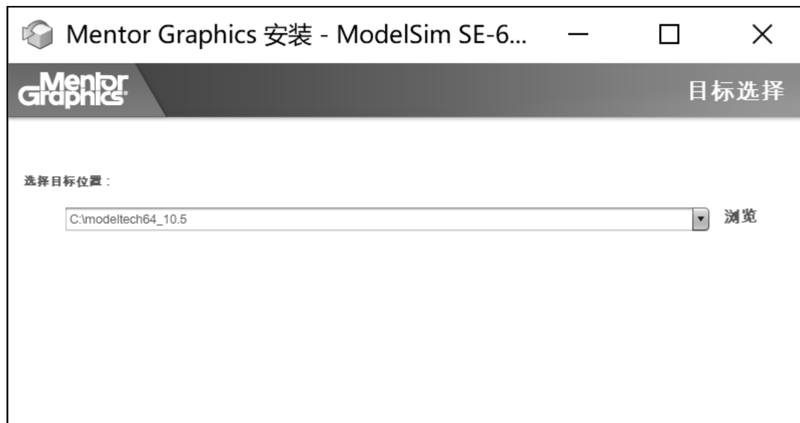

图 2.10　安装到 C 盘

提　示

建议安装至默认路径，下面都以默认路径操作。若没有选择默认路径，则本文中所有 C:\modeltech64_10.5 都应该代替为自己选择的路径。

步骤03 最后安装成功。

2.1.3 Vivado 下载及安装

1.Vivado 2016 安装

步骤 01 从官网下载 Vivado 2016，解压执行 xsetup.exe，开始安装，如图 2.11 所示。

图 2.11 解压执行 xsetup.exe

步骤 02 一直单击 Next 按钮，安装过程中出现对话框，提示要不要更新到最新版，选择不要，因为最新版的系统可能不稳定，如图 2.12 所示。

图 2.12 不要安装最新版

步骤 03 单击 Continue 按钮。勾选所有的 I Agree 复选框，然后单击 Next 按钮，如图 2.13 所示。

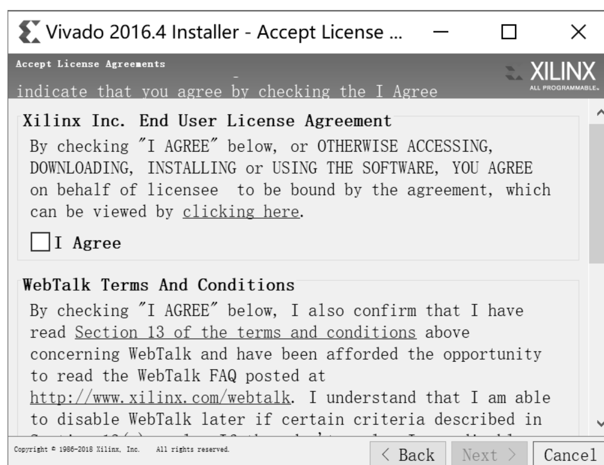

图 2.13 选择所有选项

步骤04 单击 Vivado HL Design Edition，然后单击 Next 按钮，如图 2.14 所示。

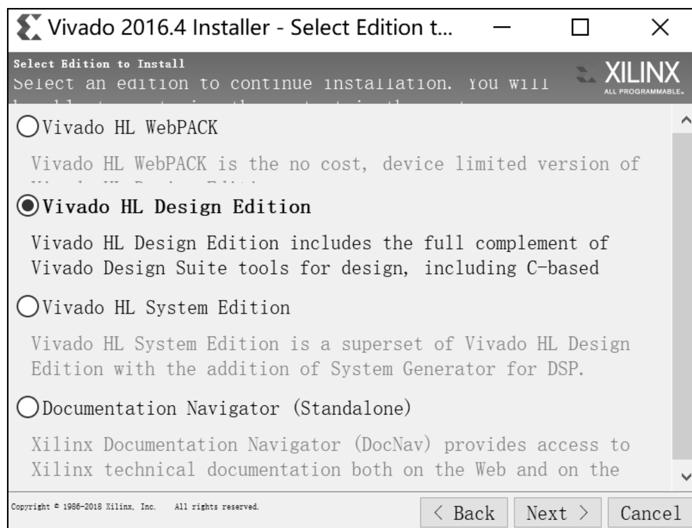

图 2.14 勾选第三个选项

步骤05 在图 2.15 所示的对话框中选中方框内的 Software Development Kit（在以后做 SOC 功能时需要），如果此时未选，那么之后将无法使用。然后单击 Next 按钮。

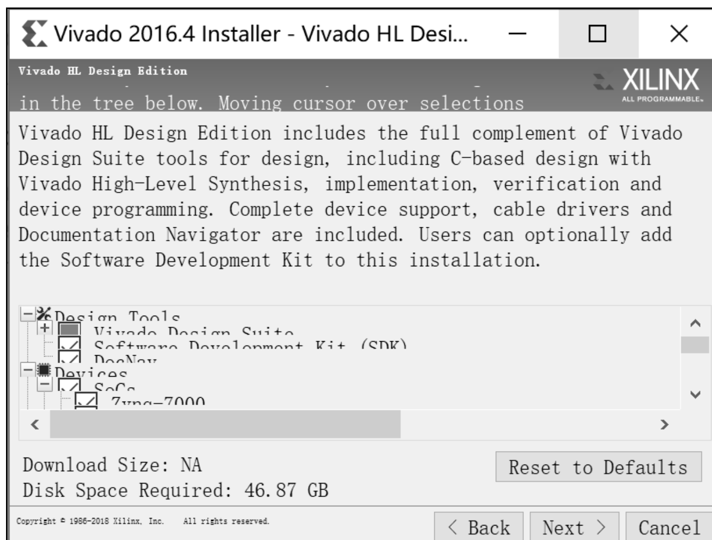

图 2.15 勾选 SDK 安装环境

步骤06 选择自己所要安装的路径，然后单击 Next 按钮。根据之前勾选的软件进行安装。该软件占用磁盘空间较大，若磁盘空间不足，则会出现红色字体，如图 2.16 所示。

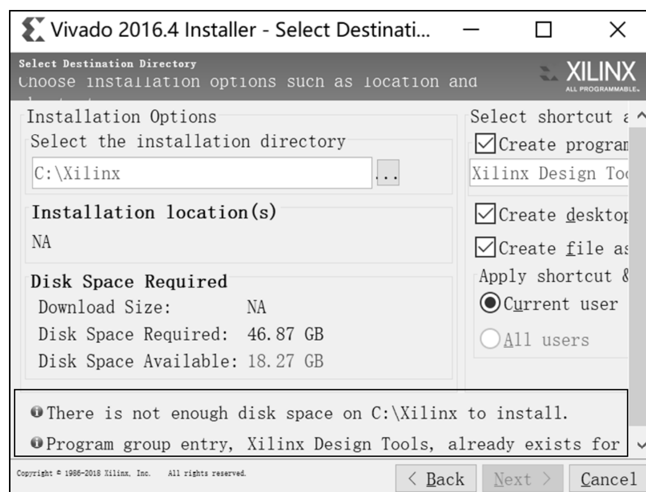

图 2.16　磁盘空间不足（见下载资源）

当磁盘空间足够时，红色字体将会全部消失，如图 2.17 所示。

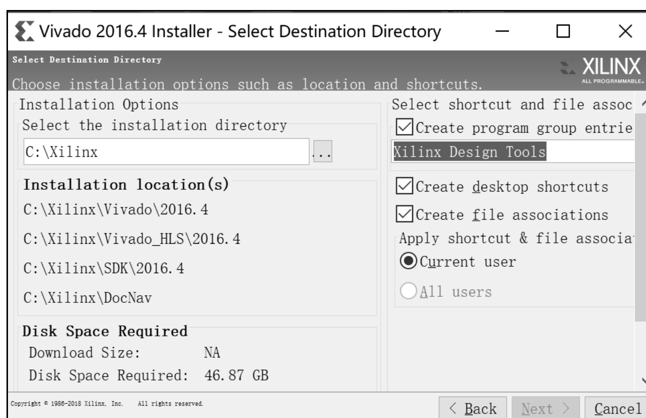

图 2.17　磁盘空间足够，红色字体消失（见下载资源）

步骤 07 若第一次安装 Xilinx，则会弹出是否要创建 Xilinx 文件夹的对话框，此时单击 Yes 按钮即可，如图 2.18 所示。

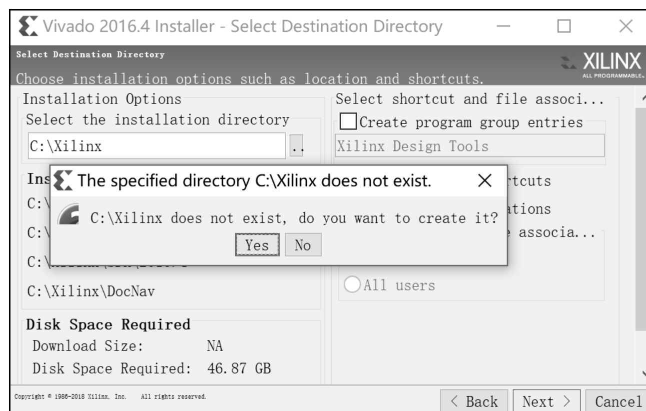

图 2.18　创建 Xilinx 文件夹

步骤 08 然后单击 Install 按钮，如图 2.19 所示。

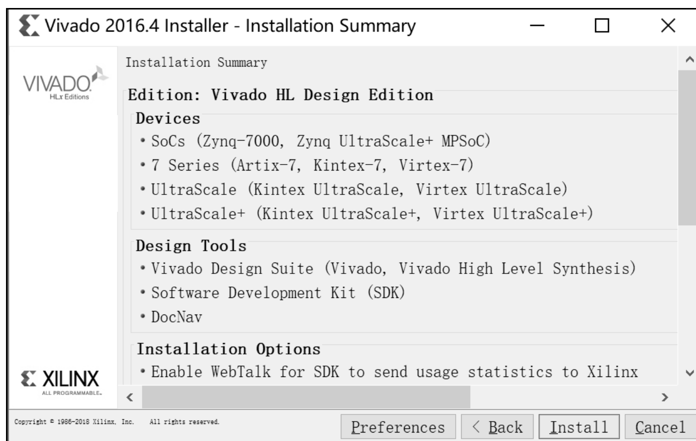

图 2.19　进行安装

步骤 09 正在安装软件，差不多要半个小时到一个小时，请耐心等待，如图 2.20 所示。

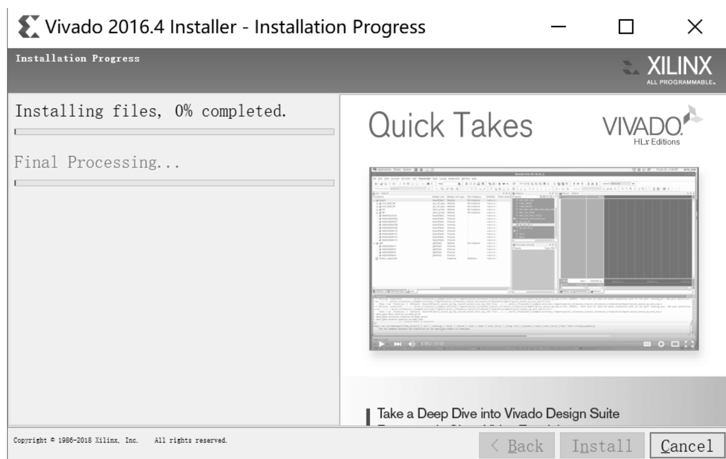

图 2.20　安装提示

2.1.4　ISE 关联 ModelSim

步骤 01 按 Windows 键，然后选择“所有程序→ISE Design Suite”，在 ISE Design Tools 里找到 Simulation Library Compilation Wizard，如图 2.21 所示。如果 ModelSim 是 64 位的就要选择 64 位文件夹下的，32 位的就选择 32 位文件夹下的。

步骤 02 找到 ModelSim 的安装目录，把 modelsim.ini 文件的只读属性去掉，如图 2.22 所示。

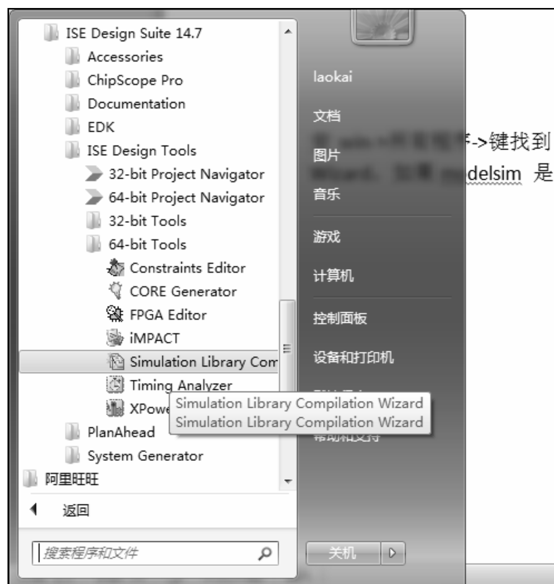

图 2.21　找到 Modelsim

图 2.22　去掉 modelsim.ini 文件的只读属性

步骤 03　ISE 仿真库编译工具选择 ModelSim SE。

步骤 04　设置 ModelSim 仿真工具，启动工具路径，如图 2.23 所示。

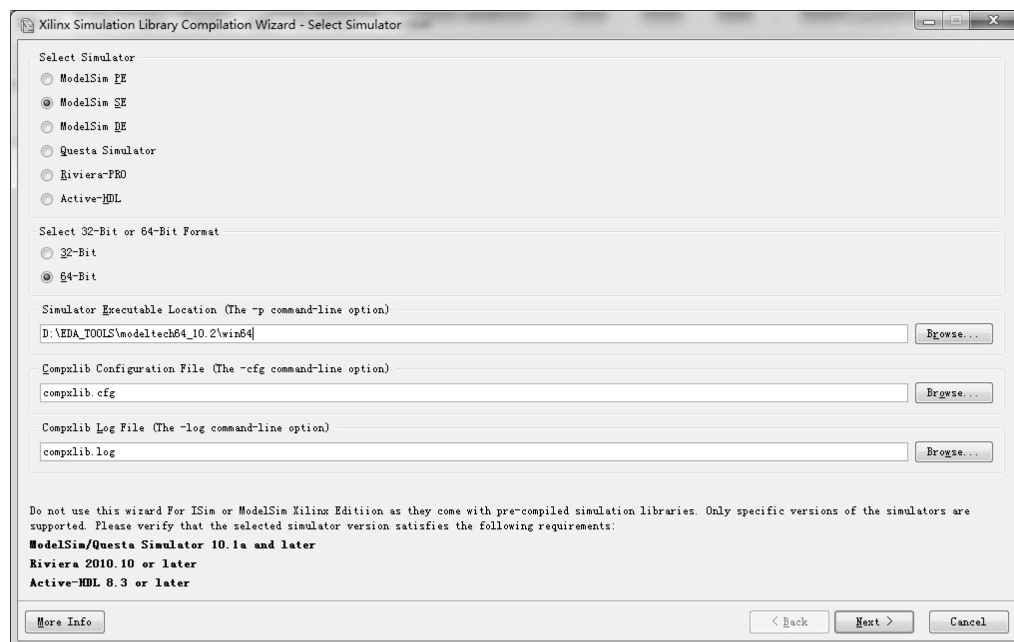

图 2.23　设置 ModelSim 仿真工具，启动工具路径

步骤 05　语言选择 Verilog or vhdl or both。选择器件库，如图 2.24 所示。

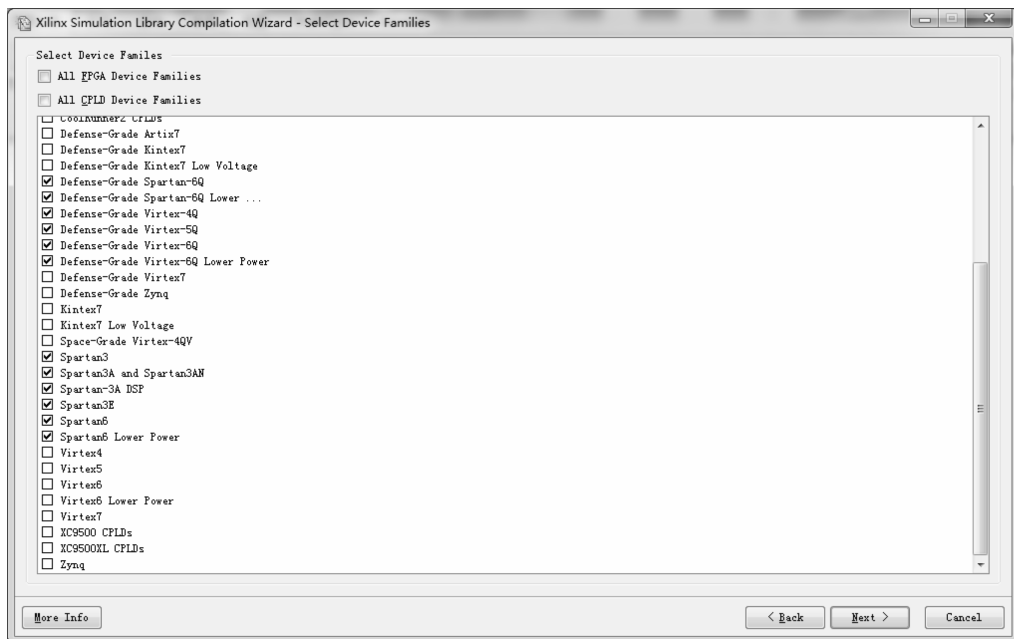

图 2.24 选择器件库

步骤 06 选择默认 Library 存放路径，如图 2.25 所示。

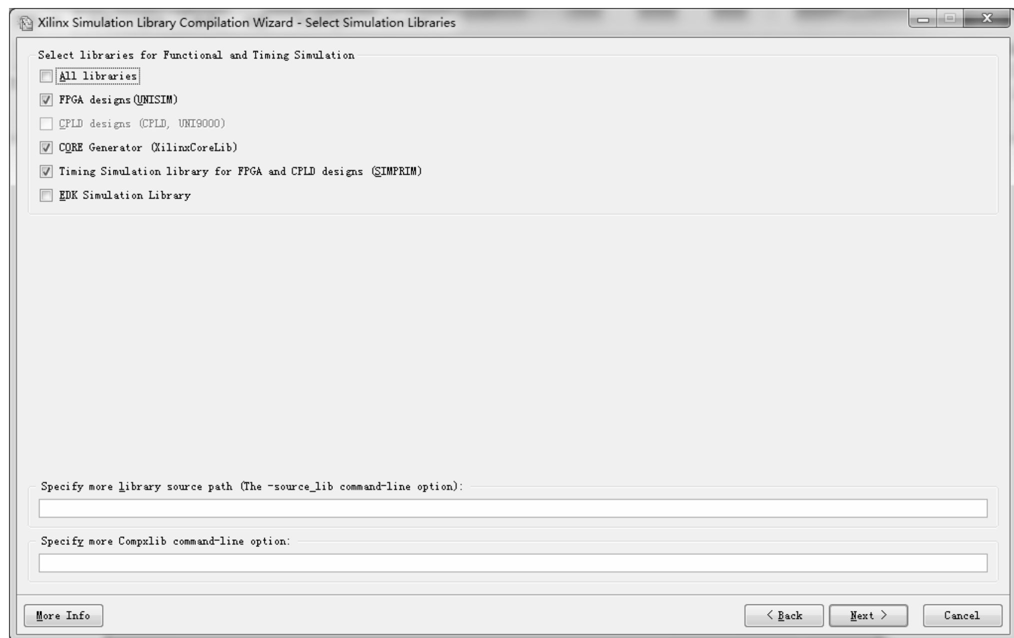

图 2.25 选择默认 Library 存放路径

步骤 07 一直单击 Next 按钮，直到出现编译按钮。单击编译按钮进行编译。

步骤 08 生成编译库后，在 Xilinx 的默认路径找到 modelsim.ini 文件并打开，如图 2.26 所示。

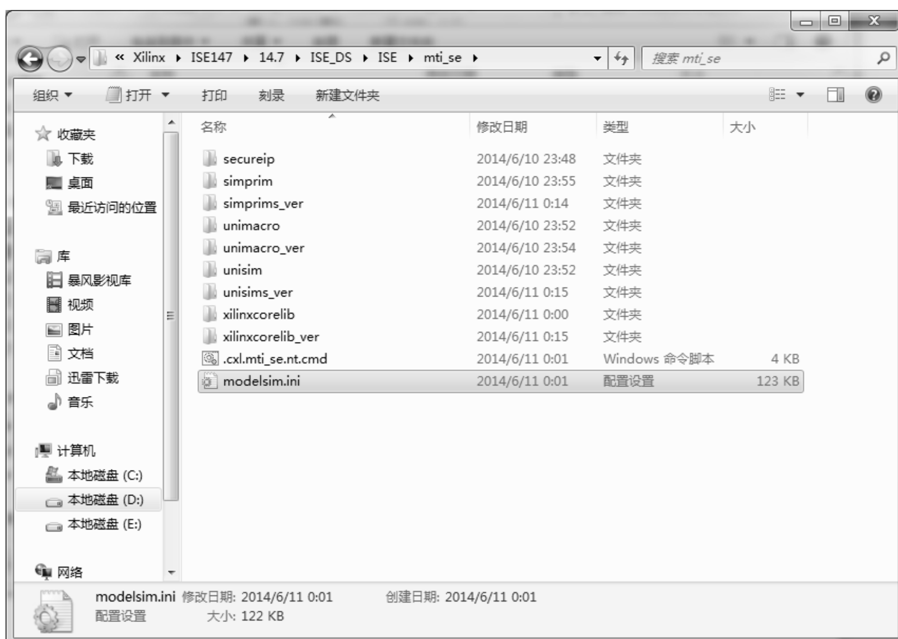

图 2.26 找到 modelsim.ini 文件

把选中部分复制到 ModelSim 安装路径下的 modelsim.ini 文件中，也就是之前把写只读属性去掉的那个文件，如图 2.27 所示。

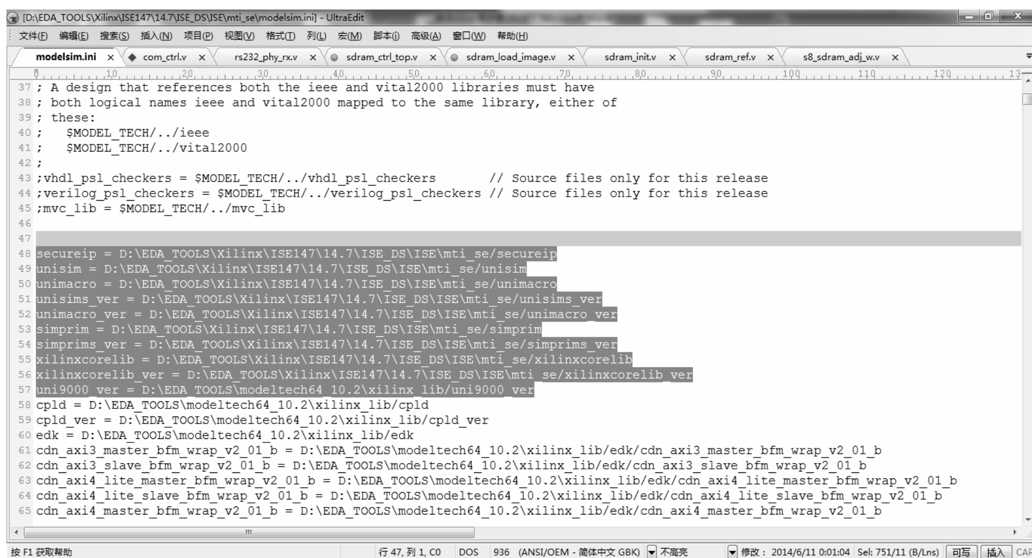

图 2.27 选取 modelsim.ini 文件部分内容

复制完成后，关闭 modelsim.ini 文件，选中只读属性，如图 2.28 所示。

注　意
一旦编译开始，就千万不要中断，如果中断，只能重新装系统才能编译库。如果第一次设置不正确，但是单击了编译库，那么让程序编译完成之后再去更改设置。

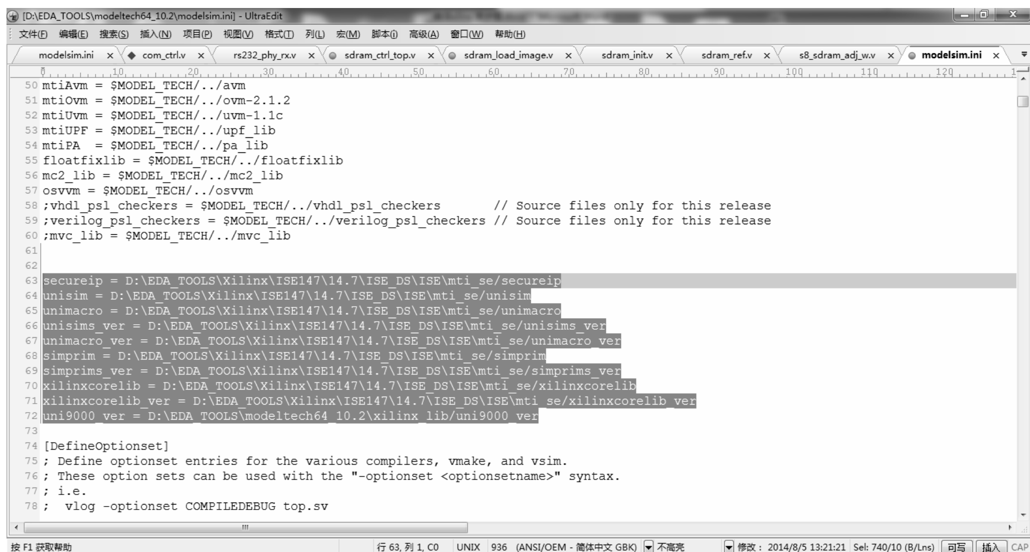

图 2.28　将选中的内容复制到安装路径下的 ini 文件中

2.1.5　Vivado 关联 ModelSim

Vivado 仿真编译步骤如下。

步骤 01　打开 Vivado 软件，单击 Tools→Compile Simulation Libraries…菜单，出现如图 2.29 所示的对话框。寻找仿真路径，选择 ModelSim 安装路径下的 win64 文件夹。

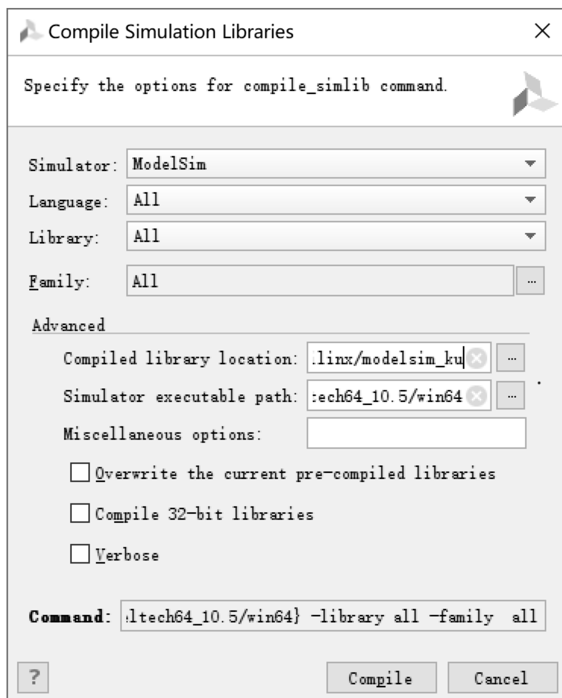

图 2.29　设置编译库路径

在自己想要的位置新建一个文件夹，用于存放编译好的内容，以便日后使用，如图 2.30 所示。

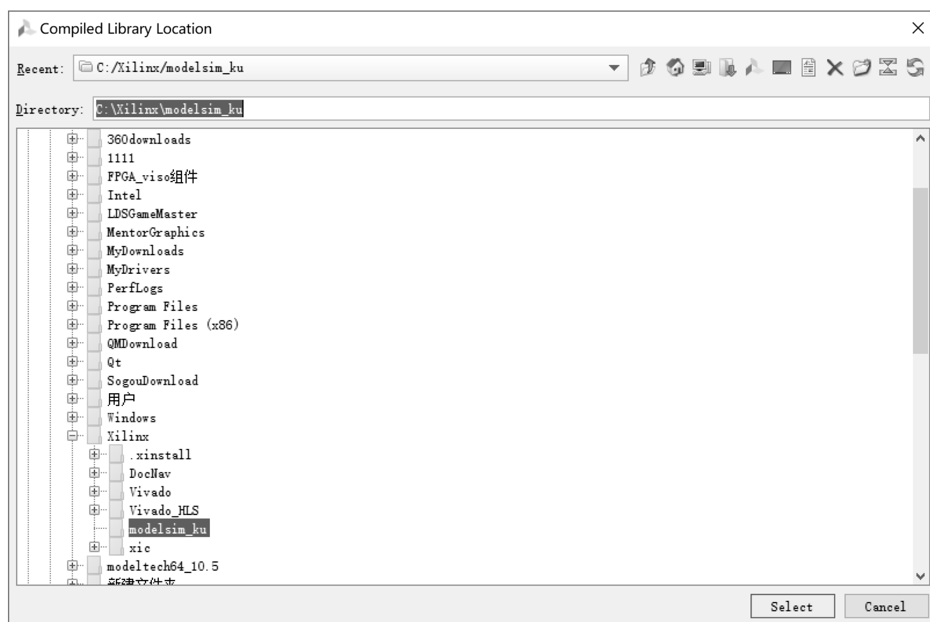

图 2.30 选择自己新建的文件夹

步骤 02 单击 Compile（综合）按钮，如图 2.31 所示。

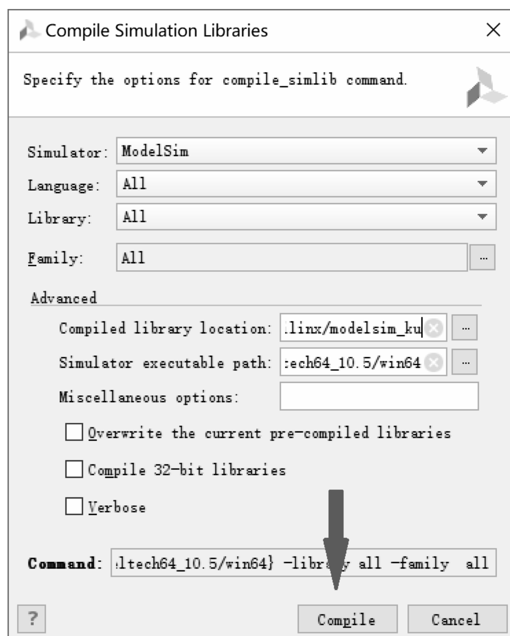

图 2.31 开始进行编译

步骤 03 编译过程如图 2.32 所示，若单击 Background 按钮，则会将编译进度隐藏。

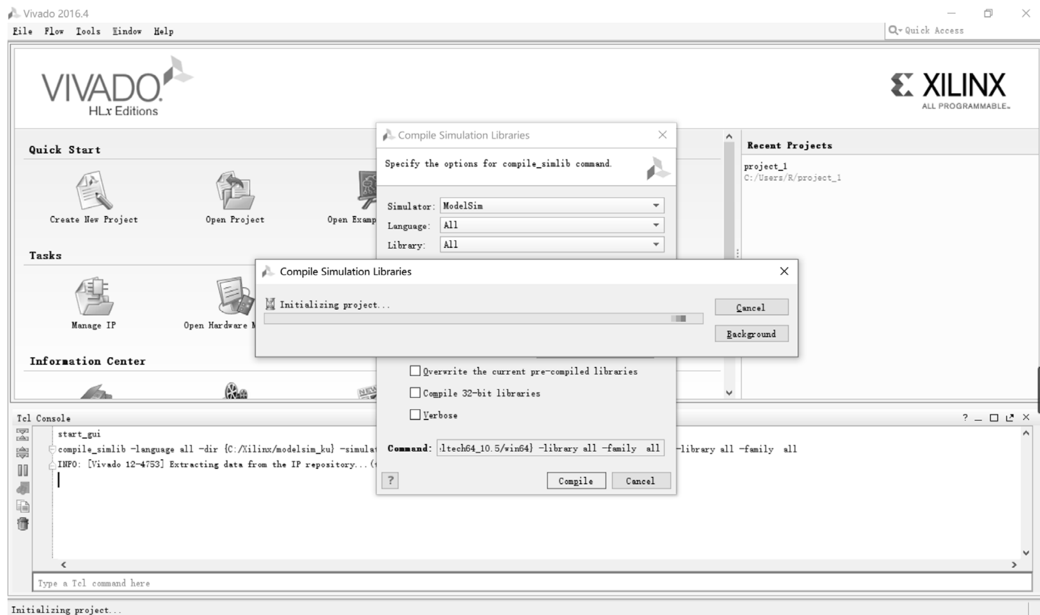

图 2.32　编译过程

步骤 04 选择仿真工具，单击 Simulation → Simulation Settings，如图 2.33 所示。出现如图 2.34 所示的界面，选择 ModelSim Simulator，然后单击 OK 按钮。

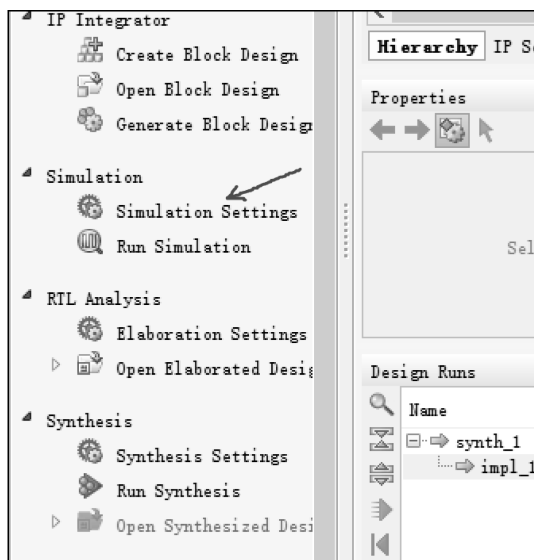

图 2.33　进行仿真设置

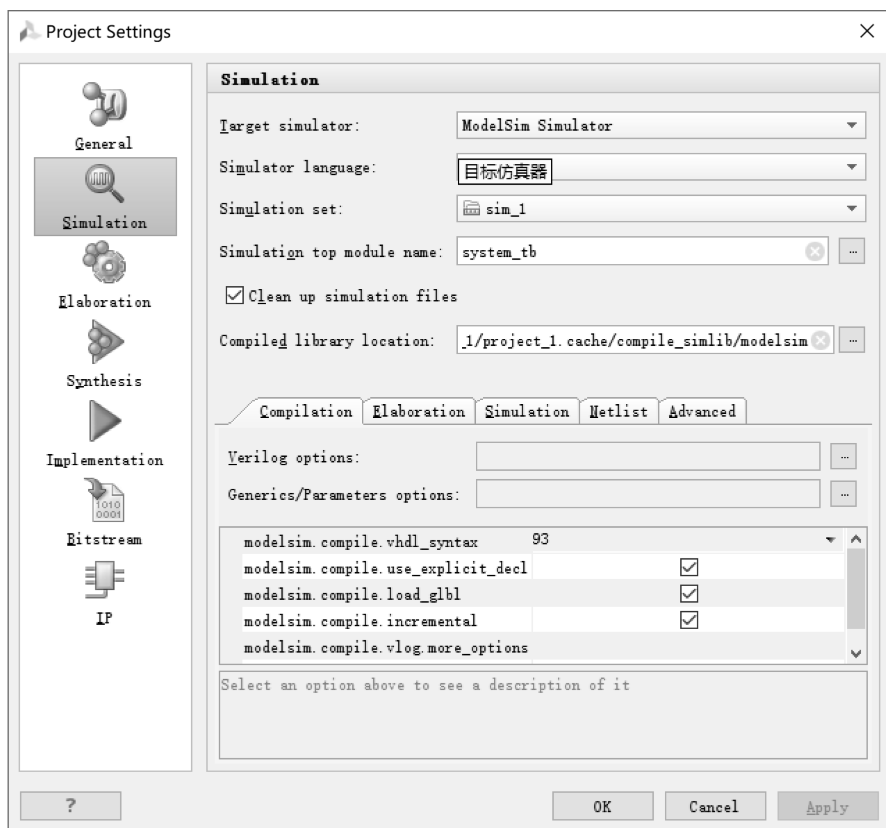

图 2.34　选择 ModelSim Simulator

步骤 05　单击开始仿真，如图 2.35 所示。

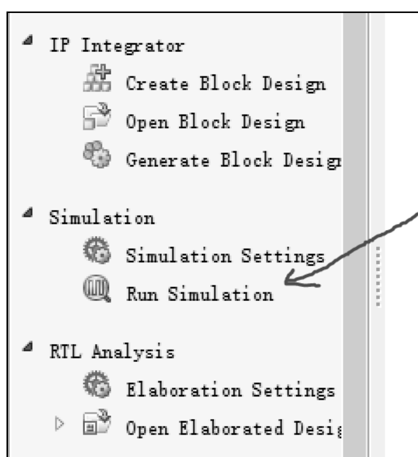

图 2.35　开始仿真

能够正常打开 ModelSim 就关联成功了。

2.1.6 UE（UltraEdit）的安装和配置

注　意

安装时，安装包不能放在中文文件夹路径下，安装步骤和普通软件一样，如图 2.36 所示。

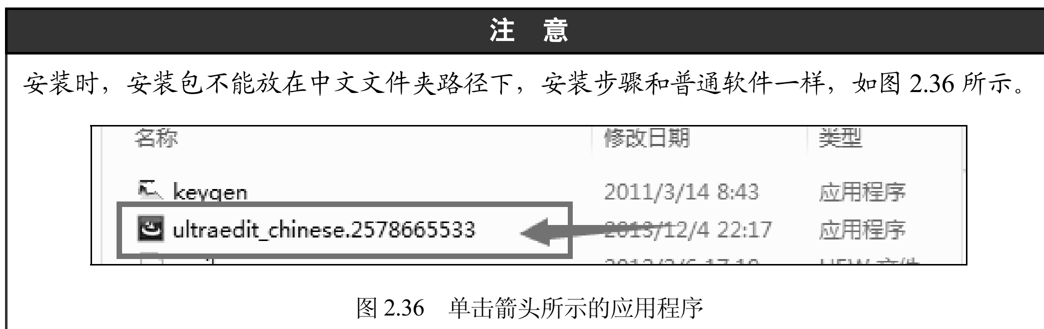

图 2.36　单击箭头所示的应用程序

1. UE 安装

步骤 01　单击“立即安装”按钮开始安装，如图 2.37 所示。

图 2.37　开始安装

步骤 02　之后等待软件运行即可，一直单击“下一步”按钮，如图 2.38 所示。

图 2.38　单击“下一步”按钮

步骤 03　遇到接受协议的地方，选择“我接受此协议”，单击“下一步”按钮，如图 2.39 所示。

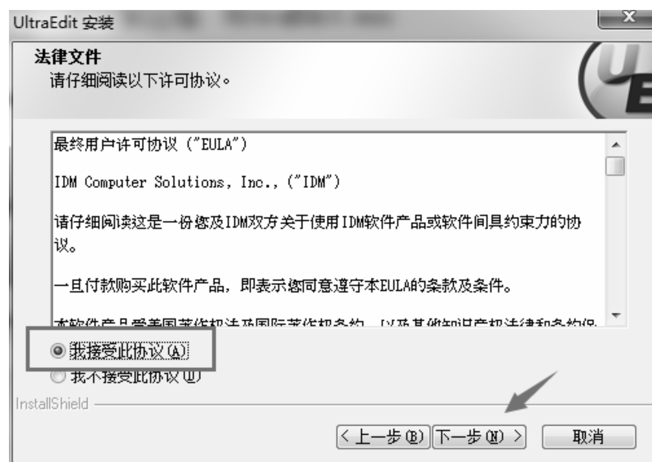

图 2.39　接受协议

步骤 04 在选择完成还是定制的时候，选择定制即可，如图 2.40 所示。

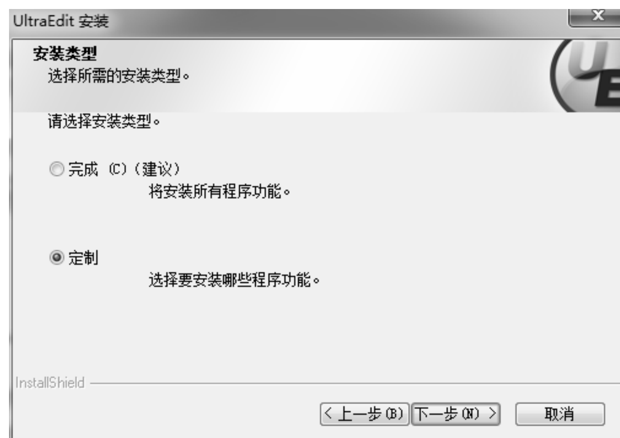

图 2.40　选择定制

步骤 05 选择安装路径，如图 2.41 所示。

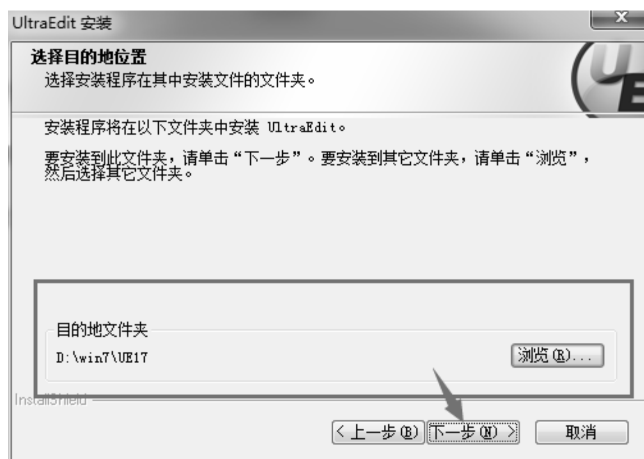

图 2.41　选择安装路径

步骤 06 一直单击“下一步”按钮，直到安装完成，如图 2.42 所示。

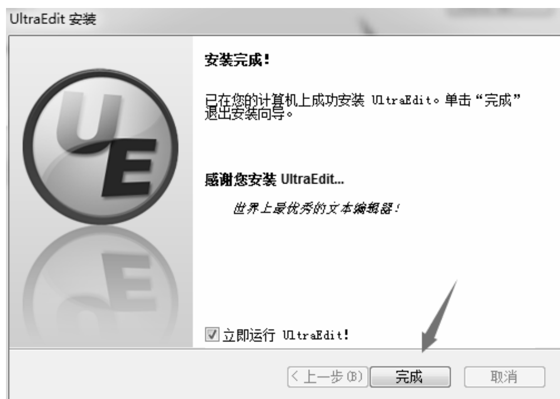

图 2.42 安装完成

2. 设置 Verilog 程序高亮

步骤 01 复制文件 verilog.uew 到 UE 注册表的 wordfiles 文件夹内，如图 2.43 所示。

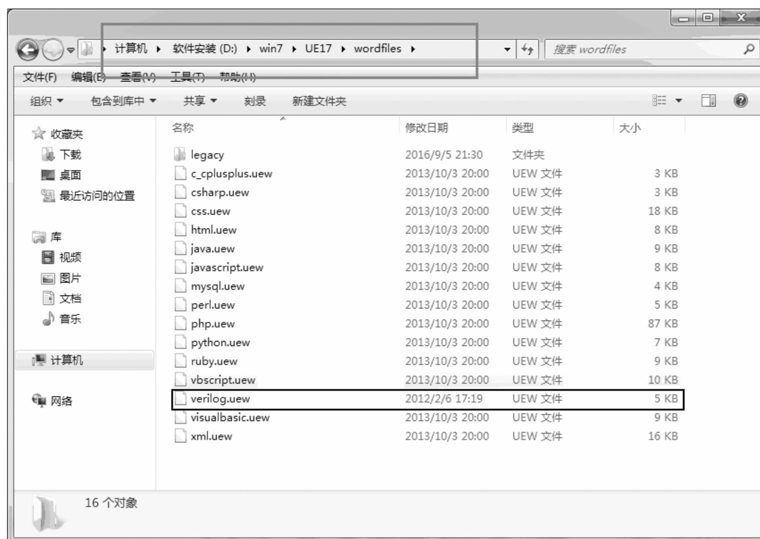

图 2.43 复制 verilog.uew 到此处

步骤 02 打开 UE 高级配置编辑，如图 2.44 所示。

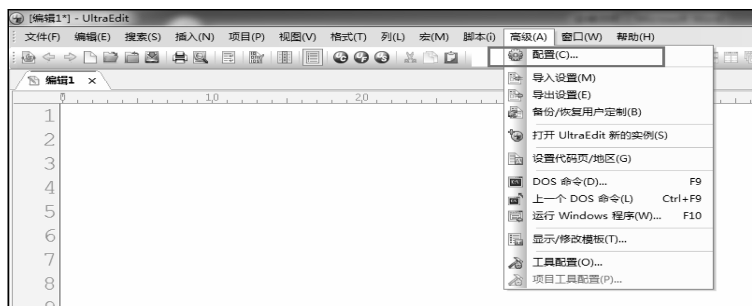

图 2.44 单击“配置”

步骤 03 单击“编辑器显示”列表项，如图 2.45 所示。

图 2.45 单击“编辑器显示”列表项

步骤 04 单击“语法高亮”选项，如图 2.46 所示。

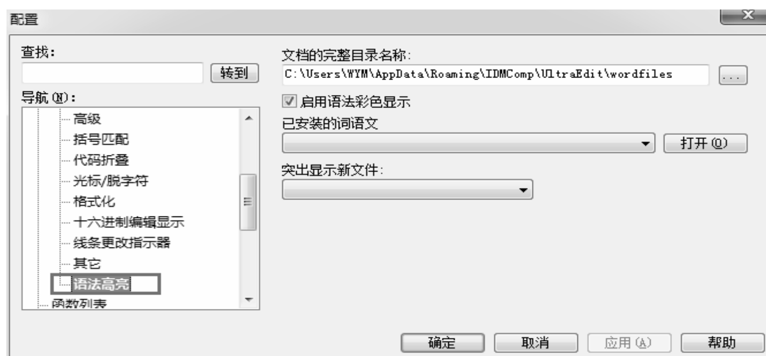

图 2.46 单击“语法高亮”选项

步骤 05 选择刚才存放 verilog.uew 文件的路径，如图 2.47 所示。

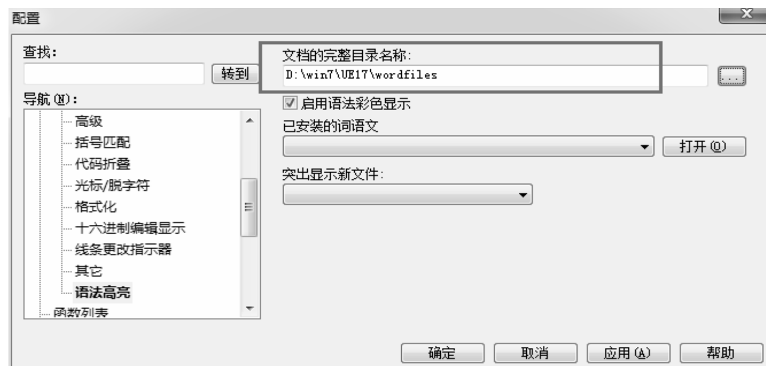

图 2.47 选择存放 verilog.uew 的路径

至此，UE 的配置全部完成。

2.2 Verilog 基本语法介绍

Verilog HDL 是一种硬件描述语言（Hardware Description Language，HDL），是以文本形式来描述数字系统硬件的结构和行为的语言，既可以表示逻辑电路图、逻辑表达式，也可以表示数字逻辑系统所完成的逻辑功能。Verilog HDL 和 VHDL 是世界上最流行的两种硬件描述语言，都是在 20 世纪 80 年代中期开发出来的。前者由 Gateway Design Automation 公司（该公司于 1990 年被 Cadence 公司收购）开发。两种 HDL 均为 IEEE 标准。

2.2.1 发展历史

Verilog 是由 Gateway 设计自动化公司的工程师于 1983 年末创立的。当时 Gateway 设计自动化公司还叫自动集成设计系统（Automated Integrated Design Systems），1985 年公司将名字改成了前者。该公司的菲尔·莫比（Phil Moorby）完成了 Verilog 的主要设计工作。1990 年，Gateway 设计自动化被 Cadence 公司收购。

1990 年代初，开放 Verilog 国际（Open Verilog International，OVI）组织（现在的 Accellera）成立，Verilog 面向公有领域开放。1992 年，该组织寻求将 Verilog 纳入电气电子工程师学会标准。最终，Verilog 成为电气电子工程师学会 1364—1995 标准，即通常所说的 Verilog-95。

设计人员在使用这个版本的 Verilog 的过程中发现了一些可改进之处。为了解决用户在使用此版本 Verilog 过程中反映的问题，Verilog 进行了修正和扩展，这部分内容后来再次被提交给电气电子工程师学会。这个扩展后的版本后来成为电气电子工程师学会 1364—2001 标准，即通常所说的 Verilog-2001。Verilog-2001 是对 Verilog-95 的一个重大改进版本，具备一些新的实用功能，例如敏感列表、多维数组、生成语句块、命名端口连接等。

2005 年，Verilog 再次进行了更新，即电气电子工程师学会 1364—2005 标准。该版本只对上一版本进行了细微修正。这个版本还包括一个相对独立的新部分，即 Verilog-AMS。这个扩展使得传统的 Verilog 可以对集成的模拟和混合信号系统进行建模。容易与电气电子工程师学会 1364—2005 标准混淆的是加强硬件验证语言特性的 SystemVerilog（电气电子工程师学会 1800—2005 标准），它是 Verilog-2005 的一个超集，是硬件描述语言、硬件验证语言（针对验证的需求，特别加强了面向对象特性）的一个集成。

2009 年，IEEE 1364—2005 和 IEEE 1800—2005 两个部分合并为 IEEE 1800—2009，成为一个新的、统一的 SystemVerilog 硬件描述验证语言（Hardware Description and Verification Language，HDVL）。

任何新生事物的产生都有它的历史沿革，早期的硬件描述语言是以一种高级语言为基础、加上一些特殊的约定而产生的，目的是实现 RTL（Register Transfer Leve）级仿真，以验证设计的正确性，而不必像在传统的手工设计过程中那样必须等到完成样机后才能进行实测和调试。

2.2.2 语言设计思路

描述复杂的硬件电路时，设计人员总是将复杂的功能划分为简单的功能，模块是提供每个简单功能的基本结构。设计人员可以采取“自顶向下”的思路，将复杂的功能模块划分为低层次的模块。这一步通常是由系统级的总设计师完成的，而低层次的模块则由下一级的设计人员完成。自顶向下的设计方式有利于系统级别层次的划分和管理，并提高了效率，降低了成本。“自底向上”方式是“自顶向下”方式的逆过程。

使用 Verilog 描述硬件的基本设计单元是模块（module）。构建复杂的电子电路主要是通过模块的相互连接调用来实现的。模块被包含在关键字 module、endmodule 之内。Verilog 中的模块类似 C 语言中的函数，能够提供输入、输出端口，可以实例调用其他模块，也可以被其他模块实例调用。模块中可以包括组合逻辑部分、过程时序部分。例如，四选一的多路选择器就可以用模块进行描述。它具有两个位选输入信号、四个数据输入和一个输出端，在 Verilog 中可以表示为：

```
module mux (out, select, in0, in1, in2, in3);
output out;
input [1:0] select;
input in0, in1, in2, in3;      //具体的寄存器传输级代码
endmodule
```

设计人员可以使用一个顶层模块，通过实例调用上面这个模块的方式来进行测试。这个顶层模块常被称为“测试平台（Testbench）”。为了对电路的逻辑进行最大程度的功能验证，测试代码需要尽可能多地覆盖系统所涉及的语句、分支、条件、路径、触发、状态机状态，验证人员需要在测试平台里创建足够多的输入激励，并连接到被测模块的输入端，然后检测其输出端的表现是否符合预期（诸如 SystemVerilog 的硬件验证语言能够提供针对验证专门优化的数据结构，以随机测试的方式进行验证，这对于高度复杂的集成电路设计验证可以起到关键作用）。实例调用模块时，需要将端口的连接情况按照这个模块声明时的顺序排列。这个顶层模块由于不需要再被外界调用，因此没有输入输出端口：

```
module tester;
reg [1:0] SELECT;
reg IN0, IN1, IN2, IN3;
wire OUT;
mux my_mux (OUT, SELECT, IN0, IN1, IN2, IN3);    //实例调用 mux 模块，这个实例被命名为
                                                  //my_muxinitial
endmodule
```

在这个测试平台模块里，设计人员可以设定仿真时的输入信号以及信号监视程序，然后观察仿真时的输出情况是否符合要求，这样就可以了解设计是否达到了预期。

示例中对模块进行实例引用时，按照原模块声明时的顺序罗列了输入变量。除此之外，还可以使用或者采用命名端口连接的方式。使用这种方式，端口的排列顺序可以与原模块声明时不同，甚至可以不连接某些端口：

```
mux my_mux (
.out(OUT),
.select(SELECT),
.in0(IN0), .in1(IN1),
.in2(IN2),
.in3(IN3));        //使用命名端口连接，括号外面是模块声明时的端口，括号内是实际的端口连接，
                   括号外相当于C语言的形式参数，括号内相当于实际参数
```

上面所述的情况是，测试平台顶层模块的测试变量直接连接了所设计的功能模块。测试平台还可以是另一种形式，即测试平台并不直接连接所设计的功能模块，而是在这个测试平台之下将激励模块和功能模块以相同的抽象级别通过线网相互连接。这两种形式的测试平台都可以完成对功能模块的测试。大型的电路系统正是由各个层次不同模块之间的连接、调用来实现复杂功能的。

2.2.3 语言要素

Verilog的设计初衷是成为一种基本语法与C语言相近的硬件描述语言。这是因为C语言在Verilog设计之初已经在许多领域得到广泛应用，C语言的语言要素已经被许多人习惯。一种与C语言相似的硬件描述语言可以让电路设计人员更容易学习和接受。不过，Verilog与C语言还是存在许多差别的。另外，作为一种与普通计算机编程语言不同的硬件描述语言，它还具有一些独特的语言要素，例如向量形式的线网和寄存器、过程中的非阻塞赋值等。总的来说，具备C语言的设计人员将能够很快掌握Verilog硬件描述语言。

1. 空白符

空白符是指代码中的空格（对应的转义标识符为\b）、制表符（\t）和换行符（\n）。如果这些空白符出现在字符串里，就可以忽略。除此之外，代码中的其他空白符在编译的时候都将被视为分隔标识符，即使用2个空格或者1个空格并无影响。不过，在代码中使用合适的空格可以让上下行代码的外观一致（例如使赋值运算符位于同一个竖直列），从而提高代码的可读性。

2. 注释

为了方便代码的修改或其他人的阅读，设计人员通常会在代码中加入注释。与C语言一样，有两种书写注释的方式。第一种为多行注释，即注释从/*开始，到*/结束；另一种为单行注释，从//开始，到这一行末尾的内容都会被系统识别为注释。

某些电子设计自动化工具会识别出代码中以特殊格式书写、含有某些预先约定关键词的注释，并从这些注释中提取有用的信息。这些注释不是供人阅读的，而是向第三方工具提供有关设计项目的额外信息。例如，某些逻辑综合工具可以从注释中读取综合的约束信息。

3. 大小写敏感性

Verilog是一种大小写敏感的硬件描述语言。其中，它的所有系统关键字都是小写的。

4. 标识符及保留字

Verilog 代码中用来定义语言结构名称的字符称为标识符，包括变量名、端口名、模块名等。标识符可以由字母、数字、下画线以及美元符（$）来表示。但是标识符的第一个字符只能是字母、数字或者下画线，不能为美元符，这是因为以美元符开始的标识符和系统任务的保留字冲突。

和其他许多编程语言类似，Verilog 也有许多保留字（或称为关键字），用户定义的标识符不能够和保留字相同。Verilog 的保留字均为小写的。变量类型中的 wire、reg、integer 等表示过程的 initial、always 等，以及所有其他的系统任务、编译指令都是关键字。可以查阅官方文献中完整的关键字的列表。

5. 转义标识符

转义标识符（又称转义字符）是由\开始的、以空白符结束的一种特殊编程语言结构。这种结构可以用来表示那些容易与系统语言结构相同的内容（例如，"在系统中被用来表示字符串，如果字符串本身的内容包含一个与之形式相同的双引号，就必须使用转义标识符）。常用的转义标识符有\n（换行）、\t（制表位）、\b（空格）、\\（反斜杠）和\"（英文的双引号）等。除此之外，在反斜杠之后也可以加上字符的 ASCII 码，这种转义标识符相当于一个字符。

2.2.4 数据类型

1. 线网与寄存器

Verilog 用到的所有变量都属于两个基本的类型：线网类型和寄存器类型。

线网与我们实际使用的电线类似，它的数值一般只能通过连续赋值（continuous assignment），由赋值符右侧连接的驱动源决定。线网在初始化之前的值为 x（trireg 类型的线网是一个例外，相当于能够储存电荷的电容器）。若未连接驱动源，则该线网变量的当前数值为 z，即高阻态。线网类型的变量有 wire、tri、wor、trior、wand、triand、tri0、tri1、supply0、supply1、trireg，其中 wire 作为一般的电路连线使用最为普遍，而其他几种用于构建总线，即多个驱动源连接到一条线网的情况，或搭建电源、接地等。

当进行模块的端口声明时，如果没有明确指出其类型，那么这个端口就会被隐含地声明为 wire 类型。因此，在声明输出端口时，应该注意是否有必要加上 reg 关键字。以下面的代码片段为例：

```
module my_moule (out1, out2, in1, in2);      //该模块具有两个输出端口
output reg out1;           //out1 端口被声明为 reg 类型，可以保存当前值
output out2;               //out2 端口被隐含地声明为 wire 类型，它的数值必须依赖连续赋值语句维持
endmodule
```

寄存器与之不同，可以保存当前的数值，直到另一个数值被赋值给它。在保持当前数值的过程中，不需要驱动源对它进行作用。如果未对寄存器变量赋值，那么初始值为 x。Verilog 中所说的寄存器类型变量与真实的硬件寄存器是不同的，是指一个储存数值的变量。如果要在一个过程（initial 过程或 always 过程）里对变量赋值，那么这个变量必须是寄存器类型的。寄存器

类型的变量有 reg（普通寄存器）、integer（整数）、time（时间）、real（实数），其中 reg 作为一般的寄存器使用最为普遍。利用寄存器变量的数组还可以对 ROM 进行建模。

关于选择线网类型还是寄存器类型，需要符合一定的规定。模块的输入端口可以与外界的线网或寄存器类型的变量连接，但是这个模块输出端口只能连接到外界的线网。再简单点，就是在两个模块的信号连接点，提供信号的一方可以是寄存器或者线网，但是接收信号的一方只能是线网。此外，在 initial、always 过程代码块中，赋值的变量必须是寄存器类型的，而连续赋值的对象只能是线网类型的变量。

2. 数字的表示

在 Verilog 里，当一个变量的类型确定，即已经知道它是寄存器类型或者是线网类型，当把具体的数值赋给它时，需要利用下面所述的数字表示方法。数字表示的基本语法结构为：

```
<位宽>'<数制的符号><数值>
```

其中，位宽是与数据大小相等的对应二进制数的位数加上占位所用 0 的位数，需要使用十进制来表示。位宽是可选项，若没有指明位宽，则默认的数据位宽与仿真器有关（最小 32 位）；数制需要用字母来表示，h 对应十六进制，d 对应十进制，o 对应八进制，b 对应二进制。若没有指明数制，则默认数据为十进制数，例如：

- 12'h123：十六进制数 123（使用 12 位）。
- 20'd44：十进制数 44（使用 20 位，高位自动使用 0 填充）。
- 4'b1010：二进制数 1010（使用 4 位）。
- 6'o77：八进制数 77（使用 6 位）。

如果某个数的最高位为 x 或 z，那么系统会自动使用 x 或 z 来填充没有占据的更高位。如果最高位为其他情况，系统会自动使用 0 来填充没有占据的更高位。

另外，如果需要使用 reg 表示负数，可以在位宽之前添加一个负号，但是需要注意后面的数值为所需负数的二进制补码。为了防止出错，可以直接使用整数 integer 或实数 real，二者都是带符号数，再利用省略位宽和数制的十进制数来表示负数。

3. 向量

向量形式的数据是 Verilog 相对 C 语言较为特殊的一种数据，但是这种数据在硬件描述语言中十分重要。在 Verilog 中，标量的意思是只具有一个二进制位的变量，而向量表示具有多个二进制位的变量。如果没有特别指明位宽，系统默认它为标量。

在真实的数字电路中，例如将两个 4 位二进制数相加的进位加法器中，我们可以发现，其中一个数是通过四条电线（每条线表示 4 位中的某一位）连接到加法器上的。我们可以用一个向量来表示这个多位数，分别用这个向量的各个分量来表示“四条电线”，即四位中的某一位。这样做的好处是，可以方便地在 Verilog 代码的其他地方选择其中的一位（位选）或多位（域选）。当然，若没有进行位选或域选，则这个多位数整体被选择。

向量的表示需要使用方括号，方括号里的第一个数字为向量第一个分量的序号，第二个数字为向量最后一个分量的序号，中间用冒号隔开。向量分量的序号不像 C 语言的数组一样必须从 0 开始，不过为了和数字电路里二进制数高低位的表示方法一致，我们常常让最低位为 0（对

于 4 位二进制数，其最高位为第 3 位、次高位为第 2 位、次低位为第 1 位、最低位为第 0 位），当然这只是一种习惯。例如，上面提到的四位二进制数用向量表示为：

```
wire [3:0] input_add;      //声明名为 input_add 的 4 位 wire 型向量
wire [4:1] input_add1;     //也是 4 位 wire 型向量，但是分量序号从 4 到 1
wire [0:3] input_add2;     //也是 4 位 wire 型向量，但是分量序号从 0 到 3
```

上面的向量声明之后，我们就可以方便地选择其中的某几个分量进行操作。请注意用于域选的方括号的位置在向量名称之后，方括号内的数字为所需的位数。例如，我们可以进行以下操作：

```
input_add [3] = 1'b1;       //将 1 赋值给 input_add 向量的第 3 位（最高位）
input_add [1:0] = 2'b01;    //将 0 和 1 分别赋值给 input_add 向量的第 1、0 位（最低两位）
```

当对向量进行赋值时，若右边的数值位宽大于左边的变量，则多出来的位被丢弃；若右边的数值位宽小于左边的变量，则不够的位用 0 填补。

4. 数组

Verilog 中的几种寄存器类型的数据（包括 reg、integer、time、real 以及由这几种数据构成的向量）都可以构成数组。声明数组时，方括号位于数组名的后面，括号内的第一个数字为第 1 个元素的序号，第 2 个数字为最后一个元素的序号，中间用冒号隔开。若数组是由向量构成的，则其中某个元素是向量。同样，出于习惯考虑，我们一般让数组第一个元素的序号为 0，后面元素的序号依次递增。此外，和 C 语言类似，用户可以声明多维数组，例如：

```
integer number [0:100];          //声明一个有 101 个元素的整数数组
number [25] = 1234;              //将 1234 赋值给 25 号（第 26 个）元素
reg [7:0] my_input [65535:0];    //声明一个有 65536 个元素的 8 位向量寄存器
my_input [97] = 8'b10110101;     //将 10110101 分别赋值给 97 号（第 2 个）元素的 7 至 0 位
reg my_reg [0:3][0:4];           //声明一个具有 20 个元素的二维寄存器数组
my_reg [1][2] = 1'b1;            //将 1 赋值给上述二维数组的第 2 行、第 3 列元素
```

由于数组和向量的表示都使用了方括号，因此使用时需要注意这个变量或向量的名称在最初被声明为何种类型的数据。上面第三行的例子是 65536 个 8 位向量组成的向量数组，它可以描述一个 64KB 的存储器。

表示数组某个元素时，允许使用变量来表示元素的索引（如 number [i] = 1234;），但是表示一个向量的一位或者几位时，只允许使用数字来表示位的索引；此外，使用数组时，一次只能对一个元素进行操作，而不能像向量那样同时对连续的几个位进行操作，例如：

```
my_input [65535][7:4] = 4'b1010; //将一个 4 位二进制数赋值给第 65536 个元素的高 4 位
```

5. 参数

可以通过 parameter 关键字声明参数。参数与常数的意义类似，不能够通过赋值运算改变它的数值。在模块进行实例化时，可以通过 defparam（参数重载语句块）来改变模块实例的参数。另一种方法是在模块实例化时使用#()将所需的实例参数覆盖模块的默认参数。局部参数可以用 localparam 关键字声明，不能够进行参数重载。

在设计中使用参数可以使得模块代码在不同条件下被重复利用，例如四位数全加器和十六位数全加器可以通过参数实例化同一个通用全加器模块。

6. 字符串

Verilog 中的字符串总体来说与 C 语言中的字符串较为类似，其中每个字符以 ASCII 表示，占 8 位。字符串存储在位宽足够的向量寄存器中。字符串中的空格、换行等特殊内容以转义标识符（参见前面提到过的转义标识符）的形式表示。

2.2.5 流程控制

为了使设计人员方便地使用寄存器传输级描述，Verilog 提供了多种流程控制结构，包括 if、if...else、if...else if...else 等形式的条件结构，case 分支结构，for、while 循环结构。这些流程控制结构与 C 语言有着相似的用法。不同的循环结构可能造成不同的逻辑综合结果。

Verilog 也提供了一些 C 语言中没有的流程控制结构以适应硬件描述语言的需要。例如，在 casex、casez 两种选择结构中，前者可将条件数值中的 x、z 均作为无关值，后者仅将 z 作为无关值。此外，还提供了 forever、repeat 两种循环结构，分别用于无限循环和指定次数循环。数字电路的逻辑功能描述常常使用到这些流程控制结构，例如 case 结构可以清晰地描述一个数据选择器。

1. 运算符

Verilog 的许多运算符和 C 语言类似，但是有一部分运算符是特有的，例如拼接运算符、缩减运算符、带有无关位的相等运算符等。

2. 按位运算

- 按位取反（~）：1 个多位操作数按位取反。例如，a=4'b1011，则~a 的结果为 4'b0100。
- 按位与（&）：2 个多位操作数按位进行与运算，各位的结果按顺序组成一个新的多位数。例如，a=2'b10，b=2'b11，则 a&b 的结果为 2'b10。
- 按位或（|）：2 个多位操作数按位进行或运算，各位的结果按顺序组成一个新的多位数。例如，a=2'b10，b=2'b11，则 a|b 的结果为 2'b11。
- 按位异或（^）：2 个多位操作数按位进行异或运算，各位的结果按顺序组成一个新的多位数。例如，a=2'b10，b=2'b11，则 a^b 的结果为 2'b01。
- 按位同或（~^或^~）：2 个多位操作数按位进行同或运算，各位的结果按顺序组成一个新的多位数。例如，a=2'b10，b=2'b11，则 a~^b 的结果为 2'b10。

3. 逻辑

- 逻辑取反（!）：对 1 个操作数进行逻辑取反，若这个操作数为 0，则结果为 1；若这个操作数不为 0，则结果为 0。
- 逻辑与（&&）：对 2 个操作数进行逻辑与，若二者都为 0 或都不为 0，则结果为 1，否则为 0。例如，3 && 0 的结果为 0。
- 逻辑或（||）：对 2 个操作数进行逻辑或，若二者中至少有一个不为 0，则结果为 1，否则为 0。例如，3||0 的结果为 1。

4. 缩减

- 缩减与（&）：对一个多位操作数进行缩减与操作，先将最高位与次高位进行与操作，其结果再与第二次高位进行与操作，直到最低位。例如，&(4'b1011)的结果为 0。
- 缩减与非（~&）：对一个多位操作数进行缩减与非操作，先将最高位与次高位进行与非操作，其结果再与第二次高位进行与非操作，直到最低位。例如，~&(4'b1011)的结果为 1。
- 缩减或（|）：对一个多位操作数进行缩减或操作，先将最高位与次高位进行或操作，其结果再与第二次高位进行或操作，直到最低位。例如，|(4'b1011)的结果为 1。
- 缩减或非（~|）：对一个多位操作数进行缩减或非操作，先将最高位与次高位进行或非操作，其结果再与第二次高位进行或非操作，直到最低位。例如，|(4'b1011)的结果为 0。
- 缩减异或（^）：对一个多位操作数进行缩减异或操作，先将最高位与次高位进行异或操作，其结果再与第二次高位进行异或操作，直到最低位。例如，^(4'b1011)的结果为 1。
- 缩减同或（~^or^~）：对一个多位操作数进行缩减同或操作，先将最高位与次高位进行同或操作，其结果再与第二次高位进行同或操作，直到最低位。例如，~^(4'b1011)的结果为 0。

5. 算术

- 加（+）：2 个操作数相加。
- 减（-）：2 个操作数相减或取 1 个操作数的负数（二进制补码表示）。
- 乘（*）：2 个操作数相乘。
- 除（/）：2 个操作数相除。
- 求幂（**）：2 个操作数求幂，前一个操作数为底数，后一个操作数为指数。

6. 关系

- 大于（>）：比较 2 个操作数，如果前者大于后者，结果为真。
- 小于（<）：比较 2 个操作数，如果前者小于后者，结果为真。
- 大于或等于（>=）：比较 2 个操作数，如果前者大于或等于后者，结果为真。
- 小于或等于（<=）：比较 2 个操作数，如果前者小于或等于后者，结果为真。
- 逻辑相等（==）：2 个操作数比较，如果各位均相等，结果为真。
- 逻辑不等（!=）：2 个操作数比较，如果各位不完全相等，结果为真。
- case 相等（===）：2 个操作数比较，如果各位（包括 x 和 z 位）均相等，结果为真。
- case 不等（!==）：2 个操作数比较，如果各位（包括 x 和 z 位）不完全相等，结果为真。

7. 移位

- 逻辑右移（>>）：1 个操作数向右移位，产生的空位用 0 填充。
- 逻辑左移（<<）：1 个操作数向左移位，产生的空位用 0 填充。
- 算术右移（>>>）：1 个操作数向右移位。若是无符号数，则产生的空位用 0 填充；若是有符号数，则用其符号位填充。
- 算术左移（<<<）：1 个操作数向左移位，产生的空位用 0 填充。
- 拼接（{,}）：2 个操作数分别作为高低位进行拼接，例如{2'b10,2'b11}的结果是 a'b1011。

- 重复（{n{m}}）：将操作数 m 重复 n 次，拼接成一个多位的数。例如，A=2'b01，则{2{A}}的结果是 4'b0101。

8. 条件运算符（?:）

根据？前的表达式是否为真，选择执行后面位于：左右的两个语句。例如，(a>b)?(a=a-1):(b=b-2)，若 a 大于 b<code>，则将 a-1 的值赋给 a，否则将 b-2 的值赋给 b。

9. 系统任务

系统任务可以被用来执行一些系统设计所需的输入、输出、时序检查、仿真控制操作。所有的系统任务名称前都带有美元符号$，使之与用户定义的任务和函数相区分。

例如，$display 用于显示指定的字符串，然后自动换行（用法类似 C 语言中的 printf 函数）；$monitor 用于监视变量，一旦被监视的变量发生变化，就会显示指定的字符串；而$time 可以提取当前的仿真时间。完整的列表请查阅参考工具、Verilog 手册或标准文档。

10. 编译指令

Verilog 具有一些编译指令，它们的基本格式为`<keyword>，注意第一个符号不是单引号，而是键盘上数字 1 左边那个键对应的撇号。常用的编译指令有文本宏预定义`define、`include，它们的功能与 C 语言中的类似，分别提供文本替换、文件包含的功能。Verilog 还提供了`ifdef、`ifndef 等一系列条件编译指令，设计人员可以使得代码在满足一定条件的情况下才进行编译。此外，`timescale 指令可以对时间单位进行定义。详细的编译指令清单请参阅相关参考书籍。

2.2.6 语言描述方法

1. 声明的两种过程

在 Verilog 中，可以声明两种不同的过程：always 过程和 initial 过程。过程可以是包含时序的过程描述，而不包含时序的过程还可以表达组合逻辑。always 过程从关键字 always 开始，可以连续多次运行，当过程的最后一行代码执行完成后，再次从第一行代码开始执行。如果没有使用系统任务$finish，always 过程将不断循环执行。initial 过程从关键字 initial 开始，它只能执行一次。

一个模块中可以包含多个过程，各个过程相互之间是并发执行的。不过，过程不能够嵌套使用。若过程中有多个语句，则需要使用关键字 begin、end 或 fork、join 将它们组成一个代码块。这两种关键字组合代表着顺序代码块和并行代码块，后面的部分会讲述这两种结构。

例如，利用 always 过程循环执行的特点，可以为模块提供一个时间脉冲（注意第一个 initial 过程为时钟的初始化，这个过程只需要进行一次）：

```
initial a = 1'b0;
always #1 a=~a;
```

虽然 always 代码块和 while 语句、forever 语句都能提供循环功能，但是 always 代码块的循环更侧重过程的循环执行，而后二者更侧重代码的循环执行。因此，为了使代码更具条理，过程的循环应当用 always 语句描述。当然，在实际使用过程中，强制使用其中的某一种在功能实

现上都是可行的。

2. 寄存器变量的过程赋值

在 Verilog 中，有两种赋值运算：

- 阻塞赋值（blocking assignment），其运算符为=。
- 非阻塞赋值（non-blocking assignment），其运算符为<=。

在顺序代码块中使用阻塞赋值语句，如果这一句没有执行完成，那么后面的语句不会执行；如果在顺序代码块中使用非阻塞赋值，那么执行这一句的同时，并不会阻碍下一句代码的执行。而且，如果后一个语句涉及前面一个非阻塞赋值语句中的变量，由于这两个语句“同时”执行，因此后一个语句所用到的是前面一个语句执行前变化的数值。非阻塞赋值是 Verilog 作为硬件描述语言与普通编程语言的一个重大区别。

带有两个触发器输出端的简单示例如下：

```
always @ (posedge reset or posedge clock)
begin
a <= b;
 b <= a;
 e n d
endmodule
```

上面的例子如果没有使用非阻塞赋值，而使用阻塞赋值，那么 flop1 和 flop2 的数值就不能被交换。flop1 和 flop2 在执行完毕后，其数值都与之前 flop2 的数值相同。在传统的编程语言中，可能需要一个临时的变量或者指针才能够达到交换两个变量的目的。这里使用了非阻塞赋值，相当于引入了一个隐含的临时变量。第二个非阻塞赋值语句右边的 a 是第一个赋值语句之前的数值，变量交换的目的得以实现。信号边缘敏感的过程语句块内常使用非阻塞赋值，使语句块的诸赋值语句同时进行，虽然功能上似乎可以用阻塞赋值实现，但是仿真时会产生不正常的结果。

通常过程赋值语句只有在触发或循环等情况，即赋值语句被执行到的时候，才会使左边的寄存器变量改变一次；而线网变量的连续赋值则一直“监视”右边表达式的变化，一旦其结果发生变化，立即会将左边的线网变量更新为此结果。如果需要对寄存器变量进行过程连续赋值，那么可以使用 Verilog 提供的 assign 或 force 关键字“强制地”将赋值运算符右边表达式的结果连续不断地施加在左边的寄存器变量上。

3. 线网变量的连续赋值

对线网类型变量的连续赋值是数字电路数据流建模的重要步骤，数字系统不含时序的组合逻辑部分可以使用线网的连续赋值描述。线网不能够像寄存器那样储存当前数值，需要驱动源提供信号，由于这种驱动是连续不断的，因此线网变量的赋值称为连续赋值。这与寄存器变量在过程中的单次赋值不同，而且所用的运算符也有区别。在 Verilog 里，线网连续赋值的关键字为 assign，下面举一个例子：

```
module andwire out;
```

```
wire in1, in2;
assign out = in1 & in2;
```

在这个例子中，线网变量 out 在系统运行过程中为两个输入线网变量 in1 和 in2 逻辑与的结果。

线网的连续赋值可以在关键字 assgin 附加延迟信息，例如上面的代码可以改为：

```
assign #5 out = in1 & in2;          //in1 和 in2 逻辑与的结果在 5 个时间周期后才施加在 out 上
```

4. 时序控制

Verilog 能够描述过程中的时序特性，这也是硬件描述语言与普通计算机编程语言的重要差别之一。过程的时序控制可以通过三种方式实现：延迟时序控制、事件时序控制以及电平敏感时序控制。

过程中的时序控制可以控制代码的执行时间。在 Verilog 中，除了过程中的时序控制外，还可以定义元件、路径的延迟。

5. 延迟时序控制

在代码中使用关键字#和延迟的时间，就可以通过延迟来进行时序控制。延迟的时间可以是数字、变量或者表达式。延迟时序控制又分为两种：常规延迟和内嵌延迟。

常规延迟在赋值语句的左边，系统执行到这一行代码时先进行延迟，再计算表达式，并将结果赋值给左边的变量。内嵌延迟在赋值语句的右边，系统执行到这一行代码时，先立即计算表达式，再进行延迟，最后把表达式的结果赋值给左边的变量。在上述两种延迟方式中，设计人员需要注意表达式的自变量在延迟过程中可能发生变化。常规延迟是先延迟再计算表达式，这时表达式的自变量可能已经发生了变化；而内嵌延迟在延迟前就已经进行了计算，表达式的自变量在延迟过程中发生的变化对已经计算的表达式结果没有影响，延迟只是指这个结果需要等待一段时间再赋给左边的变量。

下面的代码片段分别展示了常规延迟和内嵌延迟：

```
parameter latency = 8;
initialbegin x = 1;
y = 2;
#5 x = 3;              //使用常规延迟：等待 5 个系统周期后对 x 赋值
#latency y = 4;        //使用变量进行常规延迟，再等待 8 个系统周期后对 y 赋值
z = #10 (x+y);         //使用内嵌延迟：先用当前时刻的 x、y 数值计算(x+y)，再等待 10 个系统周期后对 z 赋值
end                    //z 的最终数值为 3
```

在顺序语句块（begin...end）中，语句是从上到下一行一行执行的，所有常规延迟时间都是实际执行时间相对于这一句本来应该开始执行的时间（也是上一句执行完成之时）的延迟值。因此，在上面的代码示例中，对变量 y 的赋值时间相对于上一句结束延迟了 8 个系统周期，而上一句相对系统零时刻已经延迟了 5 个系统周期，因此对 y 的赋值发生在第 13 个系统周期。不过，如果顺序语句块中存在非阻塞赋值，由于这个结构有着类似并行语句块的特点，因此需要特别考虑。

在并行语句块（fork...join）中，所有语句都是并发执行的，而所有常规延迟时间都是实际

执行时间相对于这一句本来应该开始执行的时间（也是上一句执行完成之时）的延迟值，因此各个常规延迟所指的时间都是相对于系统零时刻的。

6. 事件时序控制

事件时序控制的意思是，如果指定的事件发生，代码就被触发执行。它的关键字为@，后面可以加变量或者事件名称。参见下面的例子：

```
@(clk) x = 1;              //当变量 clk 发生变化时将 1 赋给 x
@(posedge clk) y = 2;      //在变量 clk 的上升沿将 2 赋给 y
z = @(negedge clk) (x+y);  //先计算表达式(x+y)，然后在变量 clk 下降沿将表达式的结果赋给 z
```

上面@后面括号里的是常规事件。Verilog 允许设计人员通过关键字 event 和触发符号→定义自己所需要的命名事件触发：

```
event bigger_than_two;
always @(posedge clock)
begin
if(a > 2) →bigger_than_two;            //若 a 大于 2，则事件 bigger_than_two 被触发
endalways @(bigger_than_two)           //当 bigger_than_two 被触发时，执行下面的过程
begin
//过程的代码
end
```

一种经典的用法结构如下，可以理解为“在整个仿真过程中，一旦某变量发生变化，就执行某操作”：

```
always @(a)
begin
x = x+1;
end
```

另一种用法称为 OR 事件时序控制，其代码结构为@(a or b)或@(a, b)，即当 a 或 b 其中任意一个变量发生变化时，代码或代码块才被触发执行。若监视的变量有 3 个，则其代码结构变为@(a or b or c)或@(a, b, c)，以此类推。若需要监视的变量很多，则可以使用@*或@(*)，表示对之后代码块中的所有输入变量敏感。此外，敏感列表中除了变量外，还可以是前面所提到过的常规事件、命名事件。

7. 电平敏感时序控制

Verilog 中还有一种电平敏感时序控制方式，即使用 wait(a)，当变量 a 为真时，执行后面的代码块。

8. 顺序代码块与并行代码块

begin、end 组合代表这个代码块的各行代码是顺序执行的，这种代码块称为顺序代码块；后面的 fork、join 代表这个代码块的各行代码是并发执行的，这种代码块称为并行代码块。与模块、过程不同，两种代码块是可以嵌套的，即顺序代码块中可以包含并行代码块。下面的例子

展示了这两种代码块嵌套使用的效果：

```
Initial fork
x = 1;
y = 2;
begin
z = 3;
w = 4;
end
join
```

由于这个 initial 过程使用了关键字 fork、join，其中 x、y 的赋值同时于系统零时刻发生，而 z 和 w 位于一个顺序代码块中，因此 w 的赋值在 z 的赋值后才进行。

在使用并行代码块的时候，有可能引起代码的竞争，例如两个语句对一个变量同时进行赋值。虽然理论上两个语句同时执行，但是具体的情况是必然有一句先执行，但这与顺序语句块的“先后”有本质区别。实际的先后顺序取决于所用的仿真系统。这并不是 Verilog 硬件描述语言本身的缺陷，并行语句块是一种人为设定的功能，这可以让设计人员更容易地描述某些过程，当然他们必须认真考虑竞争带来的潜在问题。

9. 任务和函数

如果某部分代码需要在不同的地方多次使用，可以在模块中定义任务或函数。

任务通过关键字 task 来声明。任务可以有零个或者多个输入变量，但是没有输出返回值。调用任务时，将按照任务内指定的方式处理这些变量。由于它相当于一个子过程，因此任务中赋值的变量只能是寄存器类型的，而且只能使用过程赋值语句。任务可以具有时序结构，例如延迟、非阻塞赋值等。任务中可以调用任务和函数。与模块的声明不同，任务的声明没有类似模块端口列表的输入变量列表。尽管如此，调用任务的时候，还是需要在括号里按照任务声明时的顺序罗列输入变量。在某种程度上，任务和 C 语言中没有返回值的函数有些类似。

函数通过关键字 function 来声明。任务不仅有输入变量，还有一个返回值作为输出变量，这个返回值的名称与函数的名称相同。函数与任务不同，它是一个只有逻辑功能的部分，不能包含时序结构。函数中只能调用函数。Verilog 中的函数与 C 语言中有返回值的函数有些类似。通常将函数放在赋值运算符的右边，返回值被赋给左边的变量。

若任务或函数同时在多个地方被调用，则需要使用 automatic 关键字声明，这样系统可以为不同地方的调用分配独立的内存空间。

2.2.7　逻辑门级描述

逻辑门级描述的抽象级别较低，仅次于晶体管级。实际的硬件电路往往都是以逻辑门级网表作为基础构建的，而设计人员常常会进行更高抽象级别的设计。尽管如此，逻辑门级的设计还是更接近真实电路形式。Verilog 提供了一系列逻辑门原语（Primitive）供用户使用。例如，非（not）、与门（and）、或门（or）、与非门（nand）、或非（nor）、异或（xor）、同或（xnor）。逻辑门原语和模块类似，可以通过实例引用的方式使用。

2.2.8 晶体管级描述

Verilog 能够在低抽象级别对电路进行描述是它的一个重要特点。Verilog 中提供了多种晶体管级（也称开关级）元件类型，包括 N 型金属氧化物半导体场效应管（关键字为 nmos）、P 型金属氧化物半导体场效应管（关键字为 pmos）、互补式金属氧化物半导体（关键字为 cmos）、带阻抗的互补式金属氧化物半导体（关键字为 rcmos）、电源单元（关键字为 supply1）、接地单元（关键字为 supply0）等。所有的晶体管都可以设置延迟属性，设计人员可以利用这些低抽象级元件构建所需要的逻辑门或直接构成其他高级组件。下面来介绍延迟编辑。

1. 逻辑门和晶体管的延迟

真实的硬件电路不可避免地都存在延迟现象。在 Verilog 中，可以对逻辑门、晶体管这些元件的延迟信息进行描述。可以为元件的延迟指定一个时间，上升、下降、关断的延迟都使用这个时间；也可以按照先后顺序分别指定上升延迟、下降延迟，而关断延迟取二者的较小值；当然也可以为上升、下降、关断各指定一个时间。例如，下面的代码为与门实例添加了三个延迟时间，分别对应上升、下降、关断：

```
and #(1, 2, 3) my_and (out, in1, in2);
```

逻辑门和晶体管的延迟属于“惯性延迟”。它的意思是，逻辑门和晶体管获得外部输入之后，延迟指定的时间后才会将结果呈现在输出端上。在延迟期间，如果输入改变，但是这个信号的持续时间小于指定延迟的时间，就不会影响逻辑门和晶体管的输出；若这个信号的持续时间大于指定延迟的时间，则之前的结果将不会呈现在输出端，改变输入信号后的结果将经过延迟后呈现在输出端。

Verilog 还允许设计人员为每个延迟时间设置最大值、典型值、最小值，在编译阶段可以通过编译代码选择其中一个。

2. 线网延迟

在声明线网或对线网进行连续赋值的时候，可以为线网添加延迟信息。这样，所有连续赋值给线网的表达式都会立即计算出结果，但是这个结果在延迟时间后才会赋给线网。如果在这段延迟时间内，右侧表达式的结果发生变化，则用于赋值的表达式结果取变化后的值。另外，如果输入变量变化的脉冲宽度小于延迟的时间，其变化不会对输出造成影响。这种延迟被称为“惯性延迟”，逻辑门和晶体管的延迟也是这种情况。

3. 过程延迟

过程延迟在前面的延迟时序控制部分讲述过。过程赋值语句中的延迟主要分为常规延迟（又称为外部延迟）和内嵌延迟（又称为内部延迟）两种。其中，前者先延迟再计算表达式，最后赋值给左边的变量；而后者则先计算表达式，经过延迟后再将结果赋给左边的变量。

4. 路径延迟

设计人员可以在模块中的关键字 specify、endspecify 之间对路径延迟进行描述。与元件的延迟不同，路径延迟是指信号在某两个寄存器类型或线网类型变量之间传递所需的延迟时间。在

specify 代码块中可以使用条件结构来根据情况选择所需的延迟时间值。与元件延迟相同的是，延迟的时间值可以指定上升、下降、关断的情况，同时也可以包含最大值、典型值、最小值。

2.2.9 逻辑综合编辑

设计人员编写的 Verilog 代码通常是在较高抽象级别的，例如寄存器传输级。这一抽象级别包含对电路信号在寄存器之间传输情况的描述，但是逻辑门级的网表（逻辑门的相互连接形式）才最接近真实的硬件电路。这一形式与寄存器传输级的描述在功能上是等效的。为了给后续硬件制造人员提供这种低抽象级别的描述，需要将高抽象级别的 Verilog 代码转换为低抽象级别的逻辑门级网表。这一过程称为逻辑综合（Logic Synthesis）。

在自动化逻辑综合工具出现之前，尽管人们可以用硬件描述语言进行设计，但是还是需要人工进行逻辑综合。例如，电路模块只有少数几个输入端，我们可以使用类似卡诺图的方法来对逻辑函数进行化简。随着电路规模不断增加，人工逻辑综合容易出错、耗费大量时间的缺点逐渐凸显。

同时，在某种特殊器件工艺下，最优化的综合结果不一定在另一种工艺下还合适，如果需要采用另外的工艺，设计人员需要花费很长时间重新进行逻辑综合。随着自动化逻辑综合工具的出现，硬件描述语言、所需器件工艺信息（工艺库）可以直接被逻辑综合工具读取，通过其内部的自动综合算法输出符合设计约束（通常包括时序、功耗、面积的约束）的逻辑门级网表。借助自动综合工具，设计人员可以将更多的精力放在高抽象级别的硬件描述语言设计中。

2.2.10 可综合代码

逻辑综合工具不能接受所有的 Verilog 代码。设计人员需要确保硬件描述语言代码是周期到周期的寄存器传输级描述。诸如 while 的循环结构必须通过信号边缘的形式（如@(posedge clock)）提供终止条件，initial 结构可能也不能被转换。

如果不指明数字的位宽，那么系统可能默认为一个较大的值（如 32 位），这就可能产生规模非常庞大的逻辑门级网表，其中一部分是不必要的，这将造成资源的浪费。与未知逻辑 x、高阻态 z 有关的运算符不能被转换，例如===、!==；此外，条件结构如果在只有 if 而没有 else 的情况下设计，或者选择结构缺少默认情况 default，很可能产生预期之外的锁存器。由于需要使用与工艺相关的逻辑门，因此用户自定义的原语很可能不能被转换。

设计人员需要采取良好的代码风格，以获得更优化的逻辑综合结果。为了适应符合可重用设计思想的系统芯片、IP 核设计，设计人员还应该遵循更严格的编码规范。

2.2.11 不可综合结构类型

除了之前介绍的可以综合的结构类型，还有一些类型是不可以综合的，只能在仿真文件中使用或者提供限制说明。下面对这些结构类型进行简单说明。

- Initial：只用于仿真测试文件。
- Events：Events 对于同步测试文件的各个组件比较有意义。

- Real：Real 数据类型不可综合。
- Time：Time 数据类型不可综合。
- Force 和 Release：Force 和 Release 不可综合。
- assign 和 deassign: reg 类型的 assign 和 deassign 操作不可综合，但是 wire 类型的 assign 操作可以综合。
- fork join：使用非阻塞赋值可以获得同样的效果。
- Primitive：只有门级的原语（Primitives）可综合。
- Table：用户自定义原语（UDP）及 Table 不可综合。
- #1：延迟只用于仿真，综合器一般直接忽略延迟。

2.2.12 高级功能编辑

1. 自定义原语

除了系统提供的 26 种逻辑门、晶体管原语外，Verilog 也提供用户自定义原语（User Defined Primitive，UDP）。原语与模块的层次结构类似，但是原语的输入输出关系完全是通过查表实现的。组合逻辑的用户自定义原语的核心是真值表，时序逻辑的用户自定义原语的核心是激励表。设计人员需要在状态表中罗列可能出现的输入和输出情况。若在实际使用过程中遇到状态表中没有定义的情况，则输出不确定值 x。使用自定义原语很直观，但是如果输入变量较多，状态表就会变得很复杂。在很多情况下，用户自定义原语并不能被逻辑综合工具转换。

2. 编程语言接口

编程语言接口（Program Language Interface，PLI）提供了通过 C 语言函数对 Verilog 数据结构进行存储、读取操作的途径。

Verilog 编程语言接口的发展先后经过了三代，其中第一代为任务或函数子程序，可以在 C 程序和 Verilog 设计之间传递数据；第二代为存取子程序，可以在用户自定义 C 程序和 Verilog 的内部数据表示的接口上被使用；第三代为 Verilog 过程接口，进一步扩展了前两代编程语言接口的功能。

通过使用编程语言接口，设计人员可以自定义接口的功能，然后通过类似调用系统任务的方式调用这些自定义功能。这样，设计人员可以很大程度地扩展所能使用的功能，例如监视、激励、调试功能，或者用来提取设计信息、显示输出等。

3. 相关工具编辑

Verilog 作为业界使用最广泛的硬件描述语言之一，有大量的电子设计自动化工具对它予以支持。通过使用集成开发环境，设计人员可以在常见的 Windows 或其他图形化系统中进行设计、仿真、验证，例如 Cadence 和 Synopsys 等公司提供的集成电路计算机辅助设计系统。

Verilog HDL 的最大特点是易学易用，如果有 C 语言的编程经验，可以在较短的时间内很快地学会和掌握，因而可以把 Verilog HDL 内容安排在与 ASIC 设计等相关的课程内部讲授。HDL 语言本身是专门面向硬件与系统设计的，这样的安排可以使学习者同时获得设计实际电路的经验。与之相比，VHDL 的学习要困难一些，但 Verilog HDL 较自由的语法也容易使初学者犯一些错误，这一点要注意。

2.3 FPGA 开发流程

FPGA 是可编程芯片，设计方法包括硬件设计和软件设计两部分。硬件包括 FPGA 芯片电路、存储器、输入输出接口电路以及其他设备，软件即相应的 HDL 程序以及最新才流行的嵌入式 C 程序。

2.3.1 设计流程

目前微电子技术已经发展到 SOC 阶段，即集成系统（Integrated System）阶段，相对于集成电路（IC）的设计思想有着革命性的变化。SOC 是一个复杂的系统，将一个完整产品的功能集成在一个芯片上，包括核心处理器、存储单元、硬件加速单元以及众多的外部设备接口等，具有设计周期长、实现成本高等特点，因此其设计方法必然是自顶向下、从系统级到功能模块的软、硬件协同设计，达到软、硬件的无缝结合。

这么庞大的工作量显然超出了单个工程师的能力，因此需要按照层次化、结构化的设计方法来实施。首先由总设计师将整个软件开发任务划分为若干个可操作的模块，并对其接口和资源进行评估，编制出相应的行为或结构模型，再将其分配给下一层的设计师。这就允许多个设计者同时设计一个硬件系统中的不同模块，并为自己所设计的模块负责；然后由上层设计师对下层模块进行功能验证。

自顶向下的设计流程从系统级设计开始，划分为若干个二级单元，然后把各个二级单元划分为下一层次的基本单元，一直下去，直到能够使用基本模块或者 IP 核直接实现为止。流行的 FPGA 开发工具都提供了层次化管理，可以有效地梳理错综复杂的层次，能够方便地查看某一层次模块的源代码以修改错误。

在工程实践中，还存在软件编译时长的问题。由于大型设计包含多个复杂的功能模块，其时序收敛与仿真验证复杂度很高，为了满足时序指标的要求，往往需要反复修改源文件，再对所修改的新版本进行重新编译，直到满足要求为止。这里面存在两个问题：

- 首先，软件编译一次需要长达数小时甚至数周的时间，这是开发所不能容忍的。
- 其次，重新编译和布局布线后的结果差异很大，会将已满足时序的电路破坏。因此，必须提出一种有效提高设计性能、继承已有结果、便于团队化设计的软件工具。

FPGA 厂商意识到这类需求，由此开发出了相应的逻辑锁定和增量设计的软件工具。例如，赛灵思公司的解决方案就是 PlanAhead。

PlanAhead 允许高层设计者为不同的模块划分相应的 FPGA 芯片区域，并允许底层设计者在所给定的区域内独立地进行设计、实现和优化，等各个模块都正确后再进行设计整合。如果在设计整合中出现错误，单独修改即可，不会影响到其他模块。PlanAhead 将结构化设计方法、团队化合作设计方法以及重用继承设计方法完美地结合在一起，有效地提高了设计效率、缩短了设计周期。

从其描述可以看出，新型的设计方法对系统顶层设计师有很高的要求。在设计初期，他们不仅要评估每个子模块所消耗的资源，还需要给出相应的时序关系；在设计后期，需要根据底层模块的实现情况完成相应的修订。

2.3.2 典型 FPGA 开发流程与注意事项

FPGA 的设计流程就是利用 EDA 开发软件和编程工具对 FPGA 芯片进行开发的过程。典型 FPGA 的开发流程一般包括功能定义/器件选型、设计输入、功能仿真、综合优化、综合后仿真、实现与布局布线、时序仿真、板级仿真与验证以及芯片编程与调试等主要步骤。

1．功能定义/器件选型

在 FPGA 设计项目开始之前，必须有系统功能的定义和模块的划分，另外就是要根据任务要求，如系统的功能和复杂度，对工作速度和器件本身的资源、成本以及连线的可布性等方面进行权衡，选择合适的设计方案和合适的器件类型。一般都采用自顶向下的设计方法，把系统分成若干个基本单元，再把每个基本单元划分为下一层次的基本单元，一直这样做下去，直到可以直接使用 EDA 元件库为止。

2．设计输入

设计输入是将所设计的系统或电路以开发软件要求的某种形式表示出来，并输入给 EDA 工具的过程。常用的方法有硬件描述语言（HDL）和原理图输入等。原理图输入方式是一种最直接的描述方式，在可编程芯片发展的早期应用得比较广泛。它将所需的器件从元件库中调出来，画出原理图。这种方法虽然直观并易于仿真，但效率很低，且不易维护，不利于模块构造和重用。更主要的缺点是可移植性差，当芯片升级后，所有的原理图都需要做一定的改动。目前，在实际开发中应用最广的就是 HDL 语言输入法，利用文本描述设计，可以分为普通 HDL 和行为 HDL。

普通 HDL 有 ABEL、CUR 等，支持逻辑方程、真值表和状态机等表达方式，主要用于简单的小型设计。在中大型工程中，主要使用行为 HDL，其主流语言是 Verilog HDL 和 VHDL。除了 IEEE 标准语言外，还有厂商自己的语言。也可以用 HDL 为主、原理图为辅的混合设计方式，以发挥两者各自的特色。

3．功能仿真

功能仿真也称为前仿真，是在编译之前对用户所设计的电路进行的逻辑功能验证，此时的仿真没有延迟信息，仅对初步的功能进行检测。仿真前，要先利用波形编辑器和 HDL 等建立波形文件和测试向量（将所关心的输入信号组合成序列），仿真结果将会生成报告文件和输出信号波形，从中便可以观察各个节点信号的变化。若发现错误，则返回修改逻辑设计。常用的工具有 Model Tech 公司的 ModelSim、Sysnopsys 公司的 VCS 和 Cadence 公司的 NC-Verilog 以及 NC-VHDL 等软件。

4．综合优化

所谓综合优化（Synthesis），就是将较高级抽象层次的描述转化成较低层次的描述。综合优化根据目标与要求优化所生成的逻辑连接使层次设计平面化，供 FPGA 布局布线软件进行实现。就目前的层次来看，综合优化是指将设计输入编译成由与门、或门、非门、RAM、触发器等基本逻辑单元组成的逻辑连接网表，并非真实的门级电路。

真实具体的门级电路需要利用 FPGA 制造商的布局布线功能，根据综合后生成的标准门级

结构网表来产生。为了能转换成标准的门级结构网表，HDL 程序的编写必须符合特定综合器所要求的风格。门级结构、RTL 级的 HDL 程序的综合是很成熟的技术，所有的综合器都可以支持这一级别的综合。常用的综合工具有 Synplicity 公司的 Synplify/Synplify Pro 软件以及各个 FPGA 厂家自己推出的综合开发工具。

5．综合后仿真

综合后仿真检查综合结果是否和原设计一致。在仿真时，把综合生成的标准延时文件反标注到综合仿真模型中去，可估计门延时带来的影响。但这一步不能估计线延时，因此和布线后的实际情况还有一定的差距，并不十分准确。目前的综合工具较为成熟，对于一般的设计可以省略这一步，如果在布局布线后发现电路结构和设计意图不符，就需要回溯到综合后仿真来确认问题所在。在功能仿真中介绍的软件工具一般都支持综合后仿真，如图 2.48 所示。

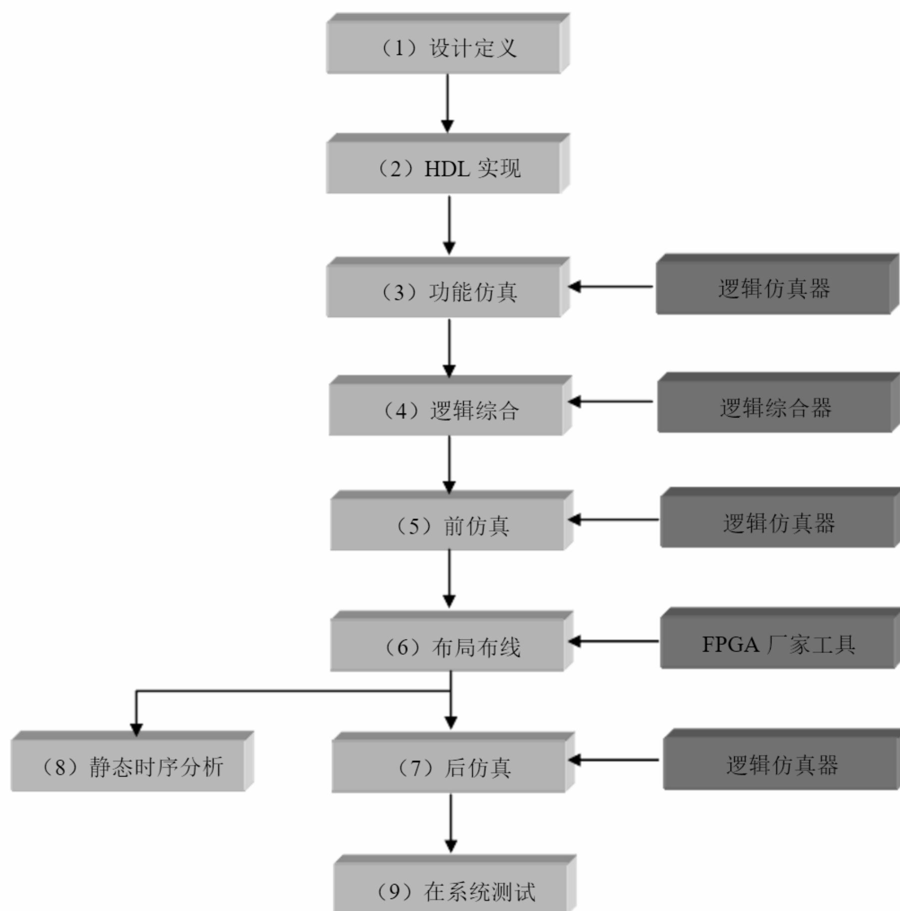

图 2.48　FPGA 设计流程图

6．实现与布局布线

布局布线可理解为利用实现工具把逻辑映射到目标器件结构的资源中，决定逻辑的最佳布局，选择逻辑与输入输出功能连接的布线通道进行连线，并产生相应文件（如配置文件与相关报告），实现是将综合生成的逻辑网表配置到具体的 FPGA 芯片上，布局布线是其中最重要的

过程。布局将逻辑网表中的硬件原语和底层单元合理地配置到芯片内部的固有硬件结构上，并且往往需要在速度最优和面积最优之间做出选择。布线根据布局的拓扑结构，利用芯片内部的各种连线资源合理正确地连接各个元件。

目前，FPGA 的结构非常复杂，特别是在有时序约束条件时，需要利用时序驱动的引擎进行布局布线。布线结束后，软件工具会自动生成报告，提供有关设计中各部分资源的使用情况。由于只有 FPGA 芯片生产商对芯片结构最为了解，因此布局布线必须选择芯片开发商提供的工具。

7. 时序仿真

时序仿真也称为后仿真，是指将布局布线的延时信息反标注到设计网表中来检测有无时序违规（不满足时序约束条件或器件固有的时序规则，如建立时间、保持时间等）现象。时序仿真包含的延迟信息最全，也最精确，能较好地反映芯片的实际工作情况。由于不同芯片的内部延时不一样，不同的布局布线方案也会给延时带来不同的影响。因此，在布局布线后，通过对系统和各个模块进行时序仿真、分析其时序关系、估计系统性能以及检查和消除竞争冒险是非常有必要的。在功能仿真中介绍的软件工具一般都支持综合后仿真。

8. 板级仿真与验证

板级仿真主要应用于高速电路设计中，对高速系统的信号完整性、电磁干扰等特征进行分析，一般都以第三方工具进行仿真和验证。

9. 芯片编程与调试

设计的最后一步就是芯片编程与调试。芯片编程是指产生使用的数据文件（位数据流文件，Bitstream Generation），然后将编程数据下载到 FPGA 芯片中。其中，芯片编程需要满足一定的条件，如编程电压、编程时序和编程算法等方面。逻辑分析仪（Logic Analyzer，LA）是 FPGA 设计的主要调试工具，但需要引出大量的测试管脚，且 LA 价格昂贵。目前，主流的 FPGA 芯片生产商都提供了内嵌的在线逻辑分析仪（如 Xilinx ISE 中的 ChipScope、Altera QuartusII 中的 SignalTapII 以及 SignalProb）来解决上述矛盾，它们只需要占用芯片少量的逻辑资源，具有很高的实用价值。

2.4 总　结

本章至此结束，主要介绍了 FPGA 的软件安装过程、设计流程和相关语法知识。从第 3 章开始将进行 FPGA 项目实战。

第 3 章

FPGA初级设计

第 2 章对 FPGA 进行了基本介绍，从本章开始实战演练。第 3、4、5 章分别属于 FPGA 初级设计、中级设计和高级设计，难度是逐级递增的。本章通过一些小项目使初学者对 FPGA 的设计规范有进一步的了解。

本章主要涉及的项目有：

- 呼吸灯的设计与实现
- 流水灯的设计与实现
- 按键控制 LED 的设计与实现
- 自动售货机的设计与实现

3.1 呼吸灯设计与实现

本节首先介绍该项目的需求，然后根据需求和 FPGA 的特性对该项目进行分析，写出流程图或者原理图，据此画出时序图，分析源码。

3.1.1 需求分析

设计一个周期为 2s 的呼吸灯，从全暗到全亮共需要 2s。FPGA 开发板系统时钟为 50MHz。

3.1.2 流程

将 2s 分为 1000 个 2ms 作为呼吸灯变化周期。将 2ms 分为 1000 个 2μs 作为呼吸灯变化单元。第一个 2ms 内的 1000 个 2μs 为全暗；第二个 2ms 内的第一个 2μs 为亮，其余的 999 个 2μs 为暗；第三个 2ms 内的第一个和第二个 2μs 为亮，其余的 998 个 2μs 为暗，以此类推，如图 3.1 所示。

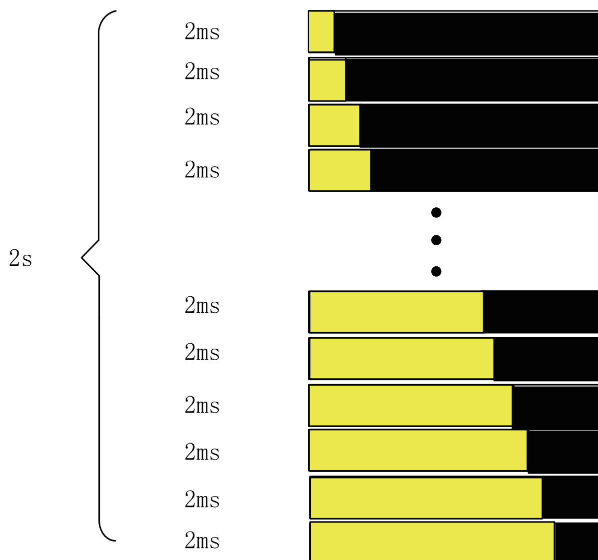

图 3.1 呼吸灯示例图

3.1.3 时序图

由于 FPGA 开发板系统时钟为 50MHz，因此时钟计数 50 000 000 次为 1s；时钟计数 100 000 000 次为 2s；时钟计数 100 000 次为 2ms，作为变化周期；时钟计数 100 次为 2μs，作为变化单元。

设置计数器 cnt_2μs，当系统时钟上升沿到来时计数器加 1，当计数器加到 99 且上升沿到来时归 0。

设置计数器 cnt_2ms，当系统时钟上升沿到来且 cnt_2μs 为 99 时，该计数器加 1，当计数器加到 999 且 cnt_2μs 为 99 且系统时钟上升沿到来时，该计数器归 0。

设置计数器 cnt_2s，当系统时钟上升沿到来且 cnt_2ms 为 999 且 cnt_2μs 为 99 时，该计数器加 1；当系统时钟上升沿到来且 cnt_2ms 为 999、cnt_2μs 为 99、cnt_2s 为 999 时，该计数器归 0，如图 3.2 所示。

在时序图中，本人习惯将输入信号填充为绿色，输出信号填充为红色，中间寄存器信号填充为黄色。本案例中 LED 信号是输出信号，故为红色，其他信号都为黄色皆可。

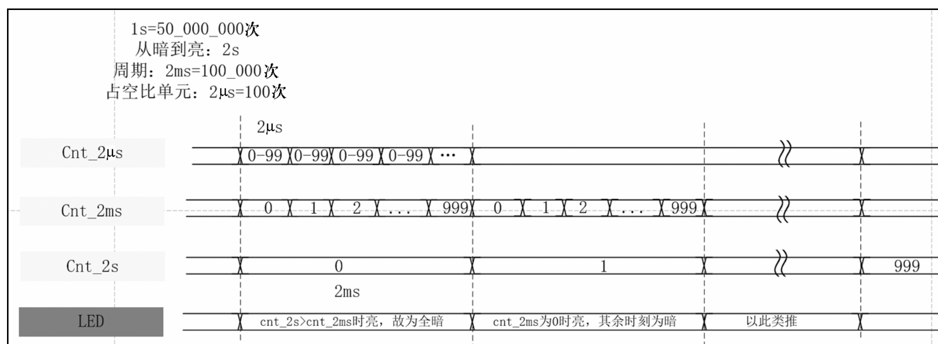

图 3.2 呼吸灯时序图

提　示

黑白印刷的效果不明显，读者可以参考随书资源中的图片。

3.1.4　时序图寄存器分析

本小节的源码是根据 3.1.3 节写出来的。

根据图 3.2 中的 cnt_2μs 可知，寄存器 cnt_2μs 产生从 0 到 99 的循环计数，当加到 99 时，恰好是 2μs 的计时，并且本项目采用的是异步复位，所以该寄存器的代码应为：

```
//cnt_2μs  将 cnt_2μs 进行计数
always@(posedge clk or negedge rst_n)
if(rst_n==0)
         cnt_2μs<=0;
else if(cnt_2μs==99)             //时钟是 50MHz，故每个时钟节拍为 0.02μs，
   cnt_2μs<=0;                   //所以时钟上升沿采集 100 次时为 2μs
else
cnt_2μs<=cnt_2μs+1;
```

寄存器 cnt_2ms 产生 0 到 999 的循环计数，并且当 cnt_2μs 寄存器等于 99 时，cnt_2ms 寄存器才加 1，当加到 999 时，恰好是 2ms 的计时。所以对于该寄存器，代码应为：

```
//cnt_2ms
always@(posedge clk or negedge rst_n)
if(rst_n==0)
cnt_2ms<=0;
/*此处也可以写为(cnt_2ms==999&&cnt_2μs==99)，但容易混淆漏写，不易排查错误，或者因优先级忘记而导致错误，建议加上括号*/
else if((cnt_2ms==999)&&(cnt_2μs==99))
cnt_2ms<=0;
else if((cnt_2μs==99)&&(cnt_2ms<999))
cnt_2ms<=cnt_2ms+1;
```

寄存器 cnt_2s 用于产生 0 到 999 的循环计数，并且当 cnt_2μs 寄存器等于 99 且 cnt_2ms 寄存器等于 999 时，cnt_2s 才加 1，当 cnt_2s 等于 999 时，恰好计时为 2s。所以对于该寄存器，代码应为：

```
//cnt_2s
always@(posedge clk or negedge rst_n)
if(rst_n==0)
cnt_2s<=0;
else if((cnt_2μs==99)&&(cnt_2ms==999)&&(cnt_2s==999))
cnt_2s<=0;
else if((cnt_2μs==99)&&(cnt_2ms==999))
cnt_2s<=cnt_2s+1;
```

寄存器 LED 作为输出控制外部呼吸灯。当寄存器 cnt_2s 大于寄存器 cnt_2ms 时，设置寄存器 LED 为 1，否则为 0。这样在第一个 2ms 内，LED 全为 0；在第二个 2ms 内，第一个 2μs 内 LED 为 1，其余的 2μs 内 LED 为 0，以此类推，以做到呼吸的效果。寄存器 led 的代码如下：

```
//LED
always@(posedge clk or negedge rst_n)
if(rst_n==0)
led<=0;                    //假设开发板中的 LED 为低复位，即 LED 为 0 时灯灭、为 1 时灯亮
else if(cnt_2s>cnt_2ms)    //此处是呼吸灯变化的地方，根据时序图仔细琢磨
led<=1;
else
led<=0;
```

3.1.5 源码展示

这是项目的全部源码，在写代码的时候要注意添加注释、画时序图。

```
module breath(
input wire clk,                 //输入系统时钟
input wire rst_n,               //复位按键
output reg led                  //呼吸灯
);

parameter CNT_2S_END = 999;     //设置 2s 的计数值
parameter CNT_2MS_END = 999;    //设置 2ms 的计数值
parameter CNT_2μs_END = 99;     //设置 2μs 的计数值

reg[9:0] cnt_2s;        //999 转化为二进制为 1111100111，共需要 10 位宽，所以设置为[9:0]
reg[9:0] cnt_2ms;       //999 转化为二进制为 1111100111，共需要 10 位宽，所以设置为[9:0]
reg[6:0] cnt_2μs;       //99 转化为二进制为 1100011，共需要 7 位宽，所以设置为[6:0]

//cnt_2μs  将 cnt_2μs 进行计数
always@(posedge clk or negedge rst_n)
if(rst_n==0)
    cnt_2μs<=0;
else if(cnt_2μs==99)   //时钟是 50MHz，故每个时钟节拍为 0.02μs，所以时钟上升沿采集 100 次时为 2μs
cnt_2μs<=0;
else
cnt_2μs<=cnt_2μs+1;

//cnt_2ms
always@(posedge clk or negedge rst_n)
if(rst_n==0)
cnt_2ms<=0;
```

```
    /*此处也可以写为(cnt_2ms==999&&cnt_2µs==99)，但容易混淆漏写，不易排查错误，或者因优先级忘记而导致错误，建议加上括号*/
    else if((cnt_2ms==999)&&(cnt_2µs==99))
    cnt_2ms<=0;
    else if((cnt_2µs==99)&&(cnt_2ms<999))
    cnt_2ms<=cnt_2ms+1;

    //cnt_2s
    always@(posedge clk or negedge rst_n)
    if(rst_n==0)
    cnt_2s<=0;
    else if((cnt_2µs==99)&&(cnt_2ms==999)&&(cnt_2s==999))
    cnt_2s<=0;
    else if((cnt_2µs==99)&&(cnt_2ms==999))
    cnt_2s<=cnt_2s+1;

    //LED
    always@(posedge clk or negedge rst_n)
    if(rst_n==0)
    led<=0;                    //假设开发板中的LED为低复位，即LED为0时灯灭、为1时灯亮
    else if(cnt_2s>cnt_2ms)    //此处是呼吸灯变化的地方，根据时序图仔细琢磨
    led<=1;
    else
    led<=0;
    endmodule
```

3.1.6 仿真文件

仿真文件对于初学者甚至有一定水平的 FPGA 工程师来说都是必须掌握的，我们在写工程文件的时候并不能保证子模块全部正确，只要有一点小小的错误就可能造成整个工程不能运行。所以我们必须设法把所有的错误都排除，起码能掌控很多不确定因素。仿真文件的存在就是为了保证子模块正确运行。

本项目中模拟开发板条件，在输入部分提供时钟信号和复位信号即可，检测输出部分是否按理想情况进行高低电平变化。因此 tb 文件为：

```
    `timescale 1ns/1ns      //这里设置时间单位为1ns，精度为1ns
    module tb_breath();
    reg clk;                //因为clk要进行赋值变化，所以为reg类型
    reg rst_n;              //和clk同理
    wire led;               //此处要检测LED的信号，所以设置为wire类型
    initial                 //初始化开始
        begin
            clk=0;          //首先设置clk为0
```

```
            rst_n<=0;       //rst_n 为 0
            #100            //延时 100ns，为了便于查看波形，通常会进行一段时间
            rst_n<=1;       //复位按键置 1
        end
    always #10 clk=~clk;    //每过 10ns，clk 取反，这里 clk 时钟周期为 20ns
    breath breath_inst(     //将呼吸灯模块例化，并连线
    .clk(clk),
    .rst_n(rst_n),
    .led(led)
    );
    endmodule
```

3.1.7　仿真结果分析

通常为了使仿真便于进行，可以将参数改小（1s 对于仿真工具来说是很长的时间），只要有相应的效果即可，如图 3.3 所示。

```
//parameter CNT_2S_END = 999;//设置2秒的计数值
//parameter CNT_2MS_END = 999;//设置2ms的计数值
//parameter CNT_2US_END = 99;//设置2μs的计数值

parameter CNT_2S_END = 9;//设置2秒的计数值
parameter CNT_2MS_END = 9;//设置2ms的计数值
parameter CNT_2US_END = 9;//设置2μs的计数值
```

图 3.3　将参数改小以便于仿真

可以将参数都设置为 9，这样只需要 9*9*9=729 个时钟周期就相当于 2s 了。仿真时便于查看，仿真结果如图 3.4 所示。

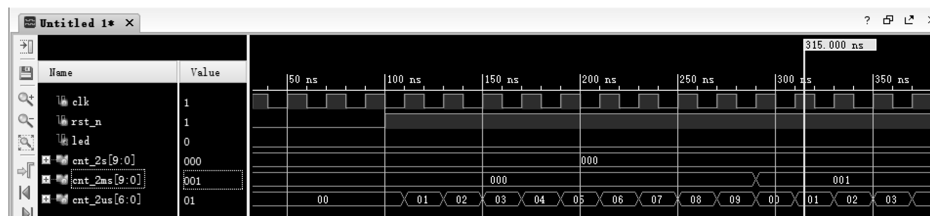

图 3.4　仿真结果（1）

为本项目建立工程，输入工程源码和仿真源码后产生仿真波形。cnt_2μs 在每一个时钟周期后加 1，从 0 到 9 不断循环，当 cnt_2μs 为 9 时，cnt_2ms 加 1，结果和时序图一样，如图 3.5 所示。

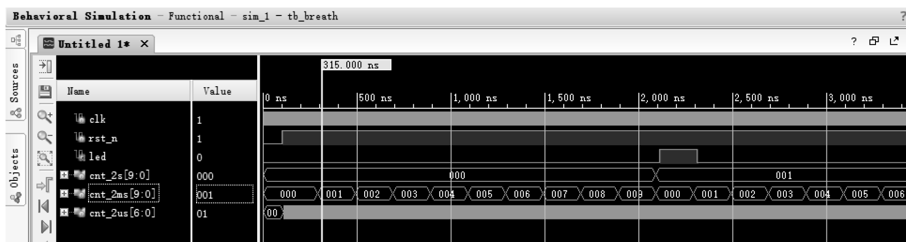

图 3.5　仿真结果（2）

由于界面大小原因，cnt_2μs 已经完全看不清了，但依旧可以看到 cnt_2ms 从 0 到 9 不断循环，而且 cnt_2s 的计数在增加，如图 3.6 所示。

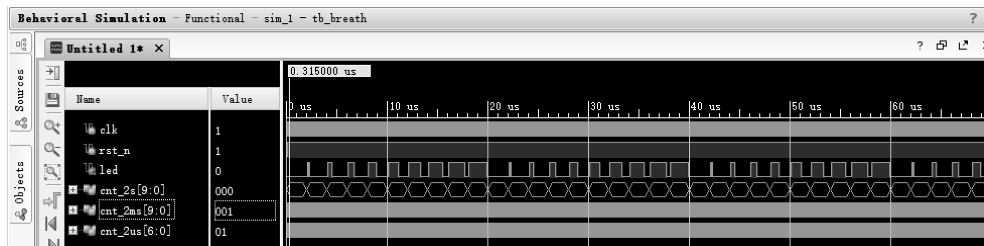

图 3.6　仿真结果（3）

调动滚轮将波形图缩小后，可以看到 LED 波形占空比由小变大不断循环的过程、每个周期都是 2ms，表示呼吸灯仿真成功。

3.1.8　约束文件

为了使项目成功下载到开发板中，还需要将约束文件写入。本项目需要时钟线、复位线和 LED 输出线。所以只需要将这三个端口绑定引脚即可。由于开发板电路连接不同，因此根据不同的开发板绑定不同的引脚，这里不进行详解。

3.1.9　扩展训练

读者可以通过以下几部分对这个应用进行进一步扩展：

（1）示例代码为先从暗到亮再从暗到亮，可以改为先从暗到亮再从亮到暗。
（2）实现多个呼吸灯，每个呼吸灯用不同的控制方法（呼吸频率不同或者暗亮变化不同）。

3.2　流水灯设计与实现

与呼吸灯相比，流水灯设计较为简单。本节主要是为了帮助大家巩固之前所学的内容并为 3.4 节自动售货机中用到的流水灯模块做铺垫。

3.2.1　需求分析

将 4 个 LED 灯每隔一秒从左往右分别依次点亮熄灭，成流水状。按复位键时进行复位，重新开始流水，如图 3.7 所示。

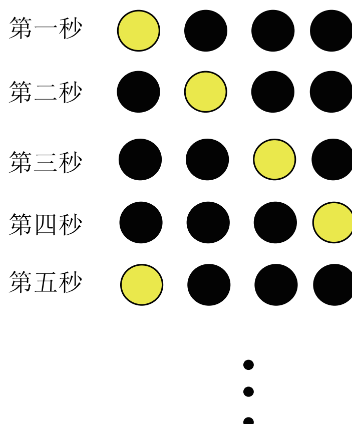

图 3.7　流水灯示例图

3.2.2　流程

由于 FPGA 时钟频率为 50MHz，因此在经历过 50 000 000 个时钟上升沿后计数到 1s，只需设置一个计数器计数到 1s 即可。设置一个 flag 寄存器，当计时到 1s 时产生一个高电压脉冲信号，LED 根据检测到的 flag 信号进行流水操作。这里使用 flag 信号的目的是在仿真和下板的时候便于调试，也是代码书写的良好规范。

3.2.3　时序图

设置 cnt_1s 寄存器，由于本开发板系统时钟为 50MHz，因此当 cnt_1s 寄存器计数 50 000 000 次时恰好为 1s。当其计数到 49 999 999 时将 cnt_1s 归 0，并设置 flag 信号为 1。当 flag 信号为 1 时，LED 进行流水操作，如图 3.8 所示。

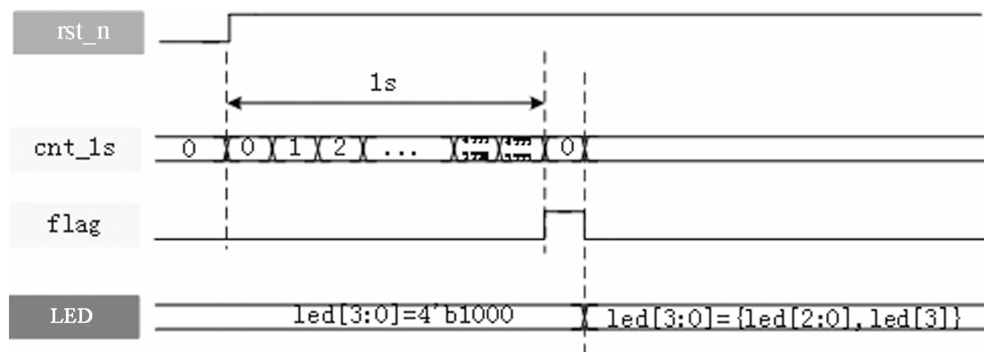

图 3.8　流水灯时序图

这里采用位操作的方法进行流水操作，假设 LED 有 4 位，将 LED 第 4 位放在第 1 位、第 1 位放在第 2 位、第 2 位放在第 3 位、第 3 位放在第 4 位，如图 3.9 所示。

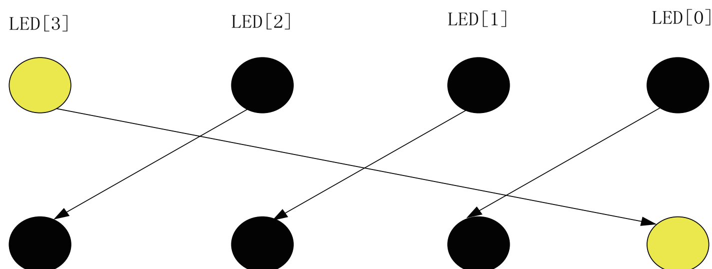

图 3.9 led<={led[2:0],led[3]}效果图

3.2.4 时序图寄存器分析

本小节是对 3.2.3 小节的时序图进行源码分析。由于 FPGA 时钟频率是 50MHz，而本流水灯每过 1s 后执行流水操作，因此寄存器 cnt_1s 计数 50 000 000 次后达到 1s，当 cnt_1s 为 49 999 999 时将其归 0。代码如下：

```
//cnt_1s  将 cnt_1s 进行计数
always@(posedge clk or negedge rst_n)
if(rst_n==0)//使用异步复位
    cnt_1s<=0;
else if(cnt_1s==CNT_1S)
cnt_1s<=0;
else
cnt_1s<=cnt_1s+1'b1;
```

当 cnt_1s 为 49 999 999 时，将 flag 寄存器拉高，否则拉低。在这里不使用 flag 寄存器也能达到效果，但是使用 flag 寄存器无论对仿真方便还是增加代码简洁度都有很大帮助，也便于 FPGA 芯片内部布局，尤其是当项目工程过大的时候很有帮助。代码如下：

```
//flag  进行流水灯变化的标志
always@(posedge clk or negedge rst_n)
if(rst_n == 0)
flag<=0;
else if(cnt_1s == CNT_1S)
flag<=1;
else
flag<=0;
```

经过前面寄存器的层层铺垫，终于可以对 LED 寄存器进行赋值了，在这里只要能检测到 flag 寄存器为高电平，进行流水操作即可。核心代码 led<={led[2:0],led[3]}前面已经讲解过了。

```
//LED
always@(posedge clk or negedge rst_n)
if(rst_n==0)
led<=4'b0001;
```

```
else if(flag == 1)
led<={led[2:0],led[3]};
```

3.2.5 源码展示

这是项目的全部源码，在写代码的时候要注意添加注释、画时序图。

```
module water(
input wire clk,              //输入系统时钟
input wire rst_n,            //复位按键
output reg[3:0] led          //4 盏流水灯
);

parameter CNT_1S = 49999999;   //设置 1s 的计数值

//49999999 转化为二进制为 10111110101111000001111111，共需要 26 位宽，所以设置为[25:0]
reg[25:0] cnt_1s;
reg  flag;
//cnt_1s  将 cnt_1s 进行计数
always@(posedge clk or negedge rst_n)
if(rst_n==0)//使用异步复位
    cnt_1s<=0;
else if(cnt_1s==CNT_1S)
cnt_1s<=0;
else
cnt_1s<=cnt_1s+1'b1;

//flag  进行流水灯变化的标志
always@(posedge clk or negedge rst_n)
if(rst_n == 0)
flag<=0;
else if(cnt_1s == CNT_1S)
flag<=1;
else
flag<=0;

//LED
always@(posedge clk or negedge rst_n)
if(rst_n==0)
led<=4'b0001;
else if(flag == 1)
led<={led[2:0],led[3]};
endmodule
/*
```

```
注：此处不使用flag也可以，但可能会在综合的时候使代码质量变差，故推荐使用flag
*/
```

3.2.6 仿真文件

本仿真文件是为了检验流水灯效果，所以只需要提供复位键和时钟信号即可。把工程文件中的CNT_1S值改小，以便于查看仿真效果。仿真文件tb_water内容如下：

```
`timescale 1ns / 1ps
module tb_water();
reg clk;                    //因为clk要进行赋值变化，所以为reg类型
reg rst_n;                  //和clk同理
wire[3:0] led;              //此处要检测led的信号，所以设置为wire类型
initial                     //初始化开始
    begin
        clk=0;              //首先设置clk为0
        rst_n<=0;           //rst_n为0
        #100                //延时100ns，为了便于查看波形，通常会进行一段时间延时
        rst_n<=1;           //复位按键置1
    end
always #10 clk=~clk;        //每过10ns后，clk取反，这里clk时钟周期为20ns
water water_inst(           //将流水灯模块例化，并连线
.clk(clk),
.rst_n(rst_n),
.led(led)
);
endmodule
```

3.2.7 仿真结果分析

将工程文件中的CNT_1S改为9之后，仿真结果如图3.10所示。从中可以看出，LED每到变化的时候都是从低位到高位依次拉高的。仿真结果和项目要求一致，之后就可以创建约束文件并编译下板了。

图3.10　流水灯仿真图

3.2.8 扩展训练

读者可以通过以下几部分对这个应用进行进一步扩展：

（1）示例代码为从右往左依次移动流水，可以改为从左往右流水或者往两边流水。
（2）将流水灯和呼吸灯结合起来。

3.3 按键控制 LED 设计与实现

本节首先介绍该项目的需求，然后根据需求和 FPGA 特性对该项目进行分析，写出流程图或者原理图，据此画出时序图，分析源码。同时，本节开始介绍实现多个子模块，然后将子模块组成起来的写法。这种写法是提倡的，在大型项目中甚至是必需的。

3.3.1 需求分析

按复位键时点亮最右边的灯，按动按键时灯依次往左移动，首先分几个子模块实现不同的功能，然后将子模块组合实现总功能。

3.3.2 流程

复位的时候点亮最右边的 LED 灯，每次按键的时候 LED 灯依次往左移动，效果如图 3.11 所示。本项目的难点是按键消抖，因为在按下按键的过程中存在按键开关接触与不接触的临界点，这些抖动可能会被 FPGA 捕捉到，造成误操作。

本项目是将按键消抖模块和LED灯流动模块分为两个子模块并例化在顶层模块中，故流程图如图 3.12 所示。

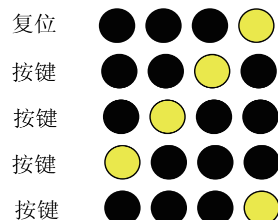

图 3.11 按键示例图

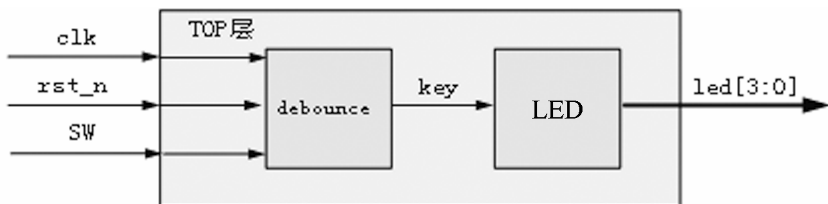

图 3.12 项目流程图

3.3.3 时序图

在画时序图的时候需要将按键抖动模拟出来，以便于理解，同时假设 FPGA 开发板的按键

按下时为低电平、抬起为高电平，如图 3.13 所示。

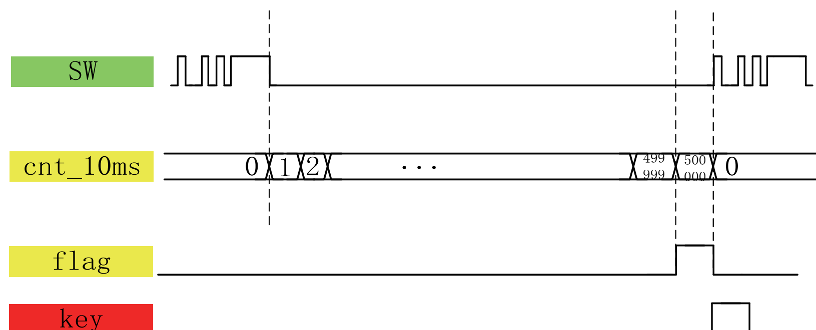

图 3.13　按键消抖时序图

（1）设置 cnt_10ms 的目的是当按下按键保持 10ms 后才认为该按键被按下了，如时序图中 SW 前面和后面的部分都是不稳定的高低电压，中间部分被计时的是 10ms 保持时间。当 cnt_10ms 计数到 499 999 时，表示 SW 被按下且保持了 500 000 个时钟周期（10ms）。当 cnt_10ms 达到 500 000 时，若 SW 依旧被按下，则保持该寄存器为 500 000 不变，这是为了防止按下一次按键后，如果按下的时间超过 20ms，就会产生两个 flag 标志，输出两个信号。

（2）设置 flag 的作用是当 cnt_10ms 计数到 499 999 并且按键电压依旧为 0 时，可以认为按键被按下，这时产生一个高电压标志。此时 cnt_10ms 已经由 499 999 变为 500 000，所以不会产生两个高电压标志。

（3）设置 key 的作用是 key 为高电压时代表按键被按下一次。

提　示
这里可以用 flag 寄存器代替 key 寄存器，但依旧不建议这么做，良好的写代码习惯很重要。

按键控制 LED 时序图如图 3.14 所示。

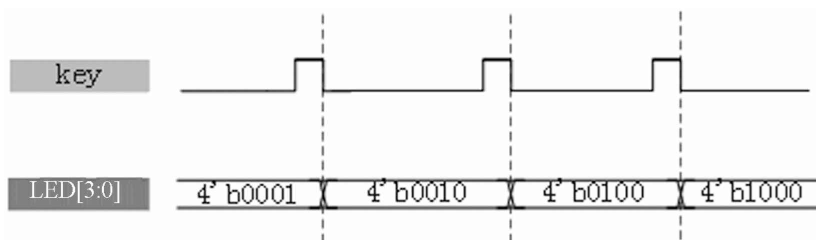

图 3.14　按键控制 LED 时序图

复位键按下时最右边的灯点亮，此时 4 盏 LED 灯为 0001，当按键按下时，LED 灯变为 0010，按键再次按下时，LED 灯变为 0100，以此类推。

3.3.4　时序图寄存器分析

当 SW 为低电平时，cnt_10ms 寄存器开始计时；当 SW 为高电平时，cnt_10ms 寄存器归 0。当 cnt_10ms 寄存器为 499 999 时，表示计时已够 10ms，此时将 flag 寄存器拉高，并将 cnt_10ms

寄存器设为 500 000，保证 flag 只会拉高一次。由于 flag 寄存器和 LED 寄存器在前面已经讲过，因此这里只对 cnt_10ms 寄存器进行分析，代码如下：

```
//cnt_10ms
always @(posedge clk or negedge rst_n)
if(rst_n == 0)
   cnt_10ms <= 0;
else if((cnt_10ms==CNT_10MS+1)&&(sw==1))//CNT_10MS 为 499999
   cnt_10ms<=CNT_10MS+1;
else if(sw==1)
   cnt_10ms<=cnt_10ms+1;
else if(sw == 0)
   cnt_10ms<=0;
```

3.3.5 源码展示

这是项目的全部源码，在写代码的时候要注意添加注释、画时序图。

（1）先写出按键消抖子模块：

```
/*按键消抖模块*/
module debounce(
input wire clk    ,
input wire rst_n  ,
input wire sw,

output reg key
);

//10ms500000 个时钟周期，转为二进制是 1111010000100100000，一共 19 位
reg[18:0]    cnt_10ms;
parameter CNT_10MS = 499999;
reg      flag;

//cnt_10ms
always @(posedge clk or negedge rst_n)
if(rst_n == 0)
cnt_10ms <= 0;
else if((cnt_10ms==CNT_10MS+1)&&(sw==1))//CNT_10MS 为 499999
cnt_10ms<=CNT_10MS+1;
else if(sw==1)
cnt_10ms<=cnt_10ms+1;
else if(sw == 0)
cnt_10ms<=0;
```

```
//flag
always @(posedge clk or negedge rst_n)
if(rst_n == 0)
flag <= 0;
else if((cnt_10ms==CNT_10MS) &&(sw==1))
flag <=1;
else
flag <=0;

//key
always @(posedge clk or negedge rst_n)
if(rst_n == 0)
key <=0;
else if(flag == 1)
key <= 1;
else
key <=0;
endmodule
```

（2）再写 LED 灯子模块：

```
/*LED 灯模块*/
module led(
input wire clk    ,
input wire rst_n  ,
input wire key    ,

output reg[3:0] led
);

//LED
always@(posedge clk or negedge rst_n)
if(rst_n == 0)
         led <= 4'b0001;
else if(key == 1)
         led <= {led[2:0],led[3]};
endmodule
//将两个子模块在顶层模块中例化并连线
/*顶层模块*/
module top(
input    wire clk  ,
input    wire rst_n,
input    wire sw   ,
output   wire[3:0]    led
);
```

```
wire key;
debounce debounce_inst(
.clk(clk),
.rst_n(rst_n),
.sw(sw),
.key(key)
);
led led_inst(
.clk(clk),
.rst_n(rst_n),
.key(key),
.led(led)
);
endmodule
```

3.3.6 仿真文件

本项目设计仿真文件时，要模拟按键按动。在模拟按下的过程中可以用取随机数的方法，当按下的时候赋值给 1 即可；在抬起按键的过程中也可以用取随机数的方法，抬起一段时间后赋值为 0。这里用轮流计数 200 的方法模拟，计数在 0~20 时模拟按下的过程，计数在 21~70 时模拟按下了，计数在 71~100 时模拟抬起的过程，计数在 101~200 时模拟不按按键的过程。计数到 200 时重新从 0 计起。源码如下：

```
`timescale 1ns / 1ps
module tb_top();
reg  clk;                   //因为clk要进行赋值变化，所以为reg类型
reg  rst_n;                 //和clk同理
reg  sw;
reg[8:0] time1;
wire[3:0] led;              //此处要检测LED的信号，所以设置为wire类型
initial                     //初始化开始
    begin
        clk=0;              //首先设置clk为0
        rst_n<=0;           //rst_n为0
        sw<=0;
        time1<=0;
        #100                //延时100ns，为了便于查看波形，通常会进行一段时间
        rst_n<=1;           //复位按键置1
    end
always #10 clk=~clk;        //每过10ns后，clk取反，这里clk时钟周期为20ns
//time1
always #20
    if(time1 == 200)
```

```
            time1 <= 0;
        else
            time1 <= time1 +1;
    //sw
    always #20
        if((time1 < 20)||(time1>70&&time1<100))
            sw <= {$random};
        else if(time1 > 100)
            sw <= 0;
        else
            sw <=1;
    top top_inst(              //将模块例化，并连线
    .clk (clk),
    .rst_n  (rst_n),
    .sw  (sw),
    .led (led)
    );
    endmodule
```

3.3.7 仿真结果分析

将工程文件中的 CNT_10MS 改为 30 后，用 time1 寄存器从 0 到 200 计数，用来模拟按键按下的过程，按键按动的过程如图 3.15 所示。

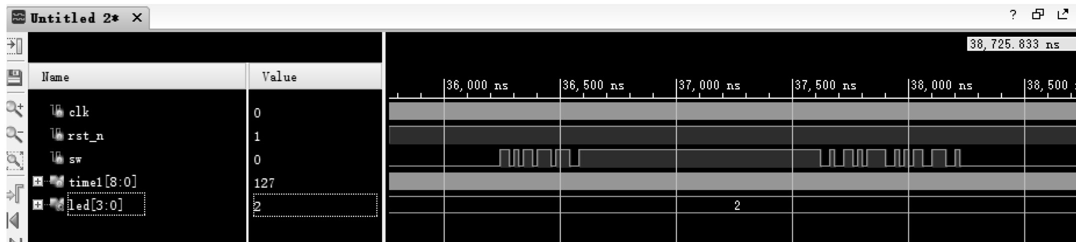

图 3.15　按键仿真图（1）

将图进一步缩放后，可以看到每按动一次按键，LED 灯进行一次变化。仿真结果和项目要求一致，如图 3.16 所示。

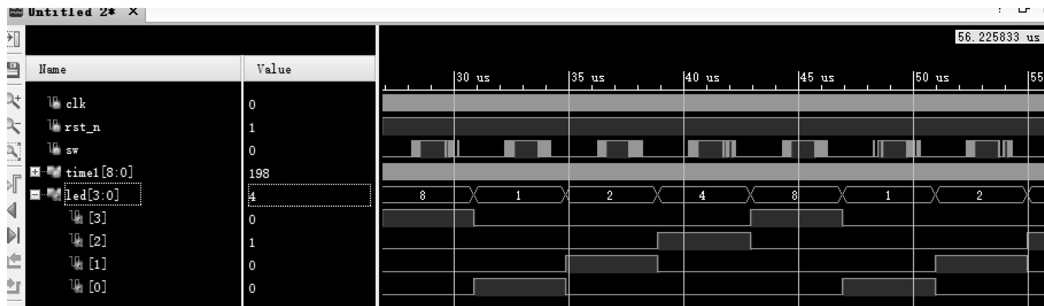

图 3.16　按键仿真图（2）

3.3.8 扩展训练

读者可以通过以下几部分对这个应用进行进一步扩展：

（1）用按键消抖进行呼吸灯和流水灯的切换。

（2）本例按键消抖为按下后触发按键效果，将其改为按下按键后不触发按键效果、抬起按键后触发按键效果。

3.4 自动售货机设计与实现

本节作为 FPGA 初级设计最后一节，将综合前面各节的知识点。本节首先介绍该项目的需求，然后根据需求和 FPGA 的特性对该项目进行分析，写出流程图或者原理图，据此画出时序图，分析源码。

3.4.1 需求分析

本项目是较为实用的项目，模拟自动售货机进行售货。本小节抛砖引玉，假设自动售货机只能接受 1 元和 5 角的硬币，而售货机中只有 2.5 元一瓶的可乐。

有 3 个按键：第一个是复位按键；第二个是 5 角按键，按下表示顾客投入 5 角硬币；第三个是 1 元按键，按下表示顾客投入 1 元硬币。

当顾客投入 5 角硬币的时候，开发板点亮一个灯；当顾客投入 1 元硬币的时候，开发板点亮 2 个灯；当顾客投入 1.5 元硬币的时候，开发板点亮 3 个灯；当顾客投入 2 元硬币的时候，开发板点亮 4 个灯；当顾客投入 2.5 元硬币的时候，开发板 4 个灯进行为时 5 秒的单向流水操作，表示此时正在出货，不找零钱；当顾客投入 3 元硬币的时候，开发板 4 个灯进行为时 5 秒的双向流水操作，表示此时正在出货，也找零钱，如图 3.17 所示。

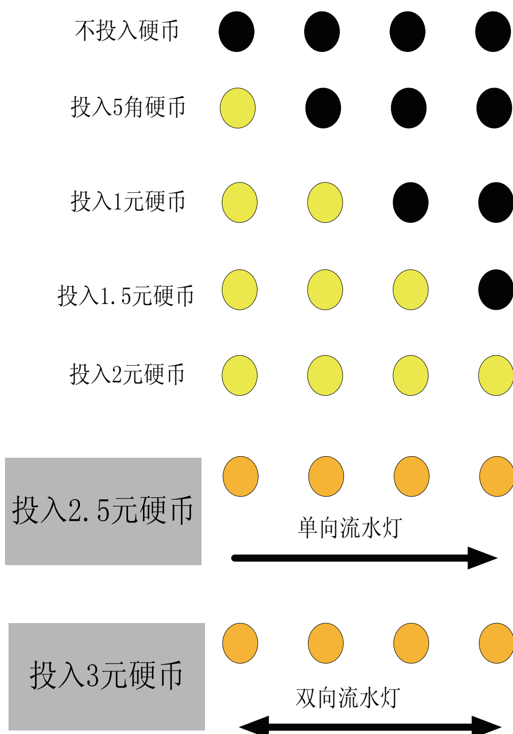

图 3.17　流水灯示意图

3.4.2 流程

根据需求分析可知，本项目是通过按键来控制 LED 灯的，所以肯定会有按键消抖模块、对按键进行处理的模块，之后会有根据按键操作来控制 LED 灯的模块，所以只需要 2 个子模块就

足够了。流程图如图 3.18 所示。

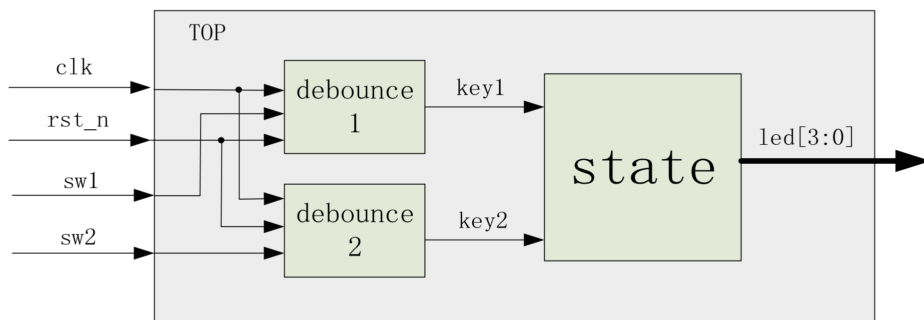

图 3.18　自动售货机流程图

debounce 子模块在前面的项目中已经讲解过了，这里的难点是 state 子模块。本项目在 state 子模块中插入两个 LED 流水灯子模块，当只需要单向流水灯和双向流水灯的时候，给流水灯子模块一个信号，流水灯子模块进行相应操作即可，这样能减少系统的复杂度、便于开发者掌控进程，如图 3.19 所示。

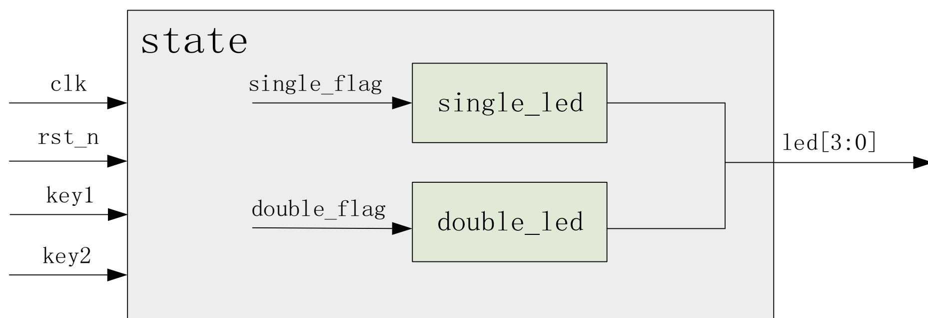

图 3.19　state 子模块图

3.4.3　时序图

说　明
按键消抖时序图请参照 3.3 节，单向 LED 灯时序图请参照 3.2 节，本节不再赘述。

这里给出 state 的状态机图，如图 3.20 所示。

初始状态是 IDLE 状态，当 key1 产生一个高电平时，state 由 IDLE 变为 A1，当 key2 产生一个高电平时，state 由 IDLE 变为 A2，以此类推，直至达到 A5 或者 A6。假设 key1 先产生一个高电平，然后 key2 产生两个高电平，则相关的时序图如图 3.21 所示。

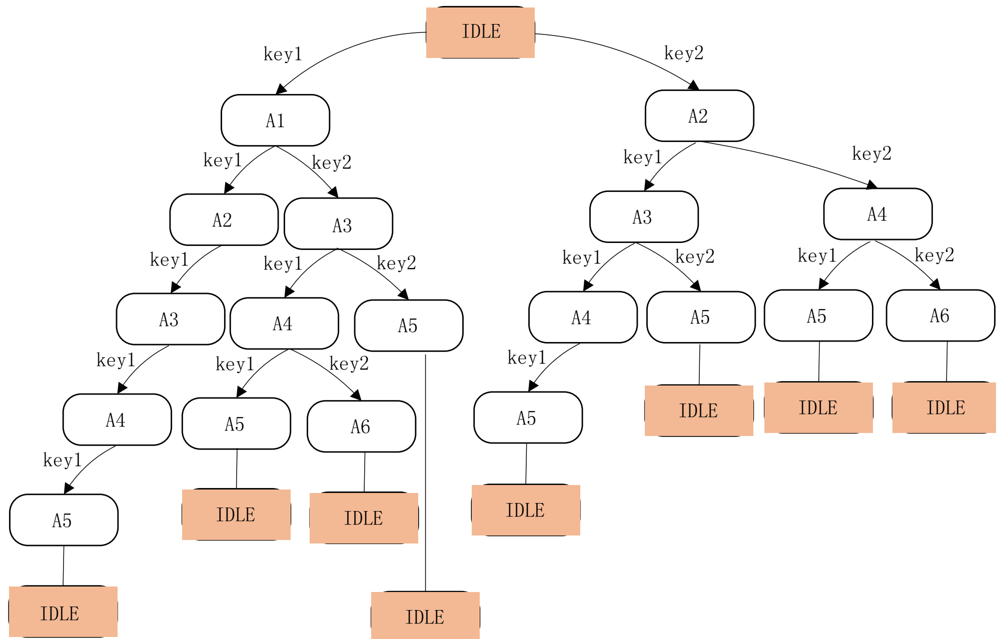

图 3.20 state 状态机图

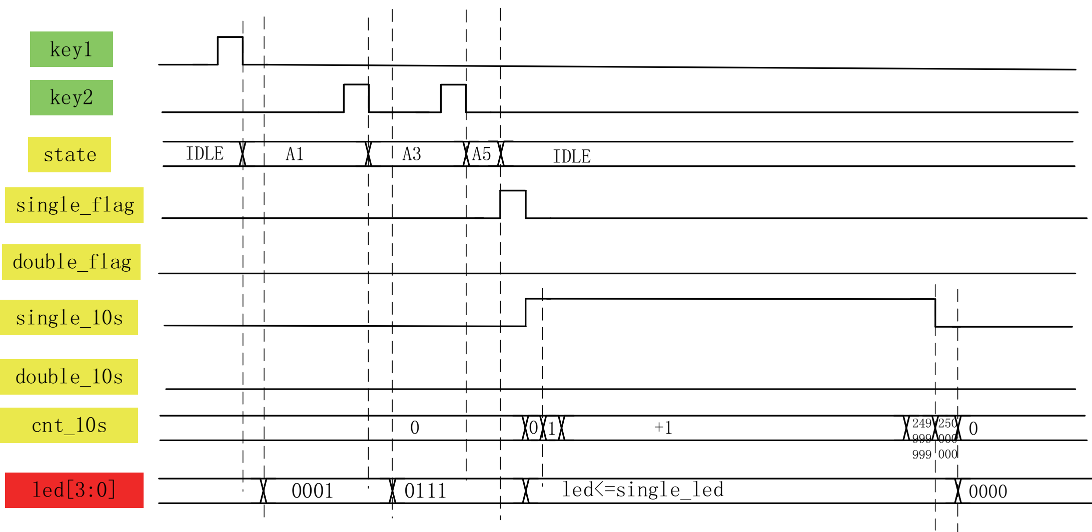

图 3.21 state 时序图（1）

假设 key2 产生三个高电平，相关的时序图如图 3.22 所示。

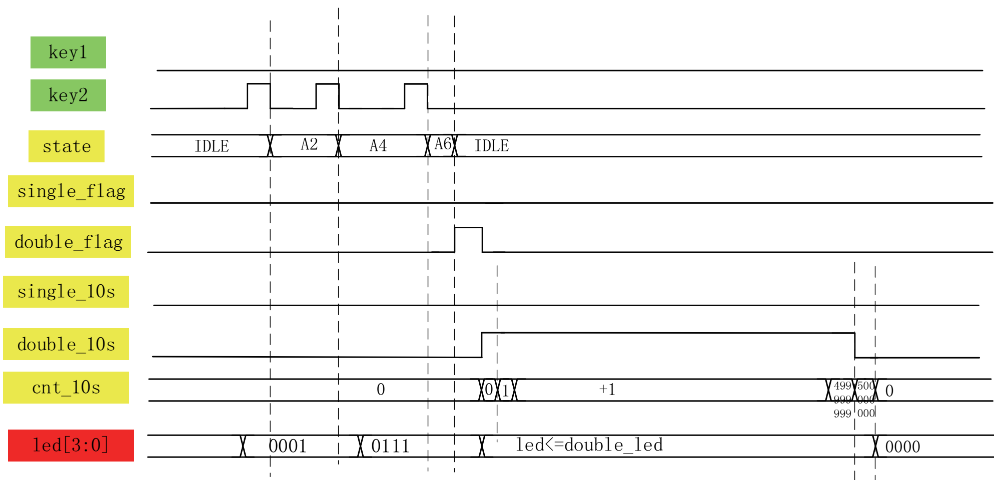

图 3.22 state 时序图（2）

3.4.4 时序图寄存器分析

按键消抖模块和流水灯模块在前面已经讲解过了，这里不再赘述。本小节寄存器分析的主要部分是 state 模块。

首先是 state 寄存器，这里的 state 寄存器是用作状态机的。硬件编程和软件编程的思路完全不同，在软件中设置某个状态通常是产生一个结构体，比如：

```
struct state{
int set_state;
};
```

当需要设置状态时，只需要将里面的成员参数 set_state 设置成相应的值即可。在硬件编程中，设置状态值是根据输入的数据进行设置的。比如，当 key1 产生高电平时，state 由 IDLE 变为 A1；当 key2 产生高电平时，state 由 A2 变为 A4，等等。所以在写代码的时候，只要把这些制约条件描述清楚即可。代码如下：

```
//state
always @ (posedge clk or negedge rst_n)
    if(rst_n == 0)
        state <= IDLE;
    case(state)
    IDLE:
        if(key1 == 1)
            state <= A1;
        else if(key2 == 1)
            state <= A2;
    A1:
        if(key1 == 1)
            state <= A2;
```

```
            else if(key2 == 1)
                state <= A3;
        A2:
            if(key1 == 1)
                state <= A3;
            else if(key2 == 1)
                state <= A4;
        A3:
            if(key1 == 1)
                state <= A4;
            else if(key2 == 1)
                state <= A5;
        A4:
            if(key1 == 1)
                state <= A5;
            else if(key2 == 1)
                state <= A6;
        A5:
            state <= IDLE;
        A6:
            state <= IDLE;
        endcase
```

single_flag 寄存器为高电平时，表示此时状态机的状态为 A5，可以进行 single_led 的 LED 灯变化。这里依旧可以不使用次 single_flag 信号，使用次信号是因为良好的编程习惯和便于调试。double_flag 寄存器也是如此。代码如下：

```
    //single_flag
    always @ (posedge clk or negedge rst_n)
        if(rst_n == 0)
            single_flag <= 0;
        else if(state == A5)
            single_flag <= 1;
        else
            single_flag <= 0;

    //double_flag
    always @ (posedge clk or negedge rst_n)
        if(rst_n == 0)
            double_flag <= 0;
        else if(state == A6)
            double_flag <= 1;
        else
            double_flag <= 0;
```

single_10s 寄存器是用来计时 10s 的，当产生 single_flag 后，LED 将会输出 single_led 的单向流水灯。当其检测到 single_flag 为高电平时，single_10s 寄存器变为高电平时，cnt_10s 开始启动计时，当 cnt_10s 寄存器计够 10s 后，single_10s 寄存器变为低电平。double_10s 同理。

```
//single_10s
always @ (posedge clk or negedge rst_n)
    if(rst_n == 0)
        single_10s <= 0;
    else if(single_flag)
        single_10s <= 1;
    else if(cnt_10s == CNT_10S)
        single_10s <= 0;

//double_10s
always @ (posedge clk or negedge rst_n)
    if(rst_n == 0)
        double_10s <= 0;
    else if(double_flag)
        double_10s <= 1;
    else if(cnt_10s == CNT_10S)
        double_10s <= 0;
```

cnt_10s 寄存器是为 single_10s 和 double_10s 计时的，原因在上面已经讲清楚了。可以分别为 single_10s 寄存器和 double_10s 寄存器设置计时器，但此处只设置一个就已经足够了。

```
//cnt_10s
always @ (posedge clk or negedge rst_n)
    if(rst_n == 0)
        cnt_10s <= 0;
    else if(single_10s || double_10s)
        cnt_10s <= cnt_10s +1;
    else
        cnt_10s <= 0;
```

led[3:0]寄存器一共有 4 位，根据状态机、single_10s 寄存器和 double_10s 寄存器来设置 led[3:0]的值，并且要把 single_10s 寄存器和 double_10s 寄存器的判断条件放在前面，因为如果 single_10s 寄存器和 double_10s 寄存器满足条件了，将不会对之后的条件进行判断，所以要把这两个寄存器的判断条件放在前面。在这里可以用 case 语句写，这样生成的电路是并行的电路。这段代码生成的电路是级联电路，对于后面的判断条件可能会造成时序伪例，当 if 语句较少的时候也可以这么写。

```
//led[3:0]
always @ (posedge clk or negedge rst_n)
    if(rst_n == 0)
        led <= 4'b0000;
```

```
        else if(single_10s == 1)
            led <= single_led;
        else if(double_10s == 1)
            led <= double_led;
        else if(state == A1)
            led <= 4'b0001;
        else if(state == A2)
            led <= 4'b0011;
        else if(state == A3)
            led <= 4'b0111;
        else if(state == A4)
            led <= 4'b1111;
        else if(state == IDLE)
            led <= 4'b0000;
```

3.4.5 源码展示

本项目源码模块和之前的大部分相似，请在随书代码中参阅。

3.4.6 仿真文件

debounce()模块之前已经测试过了，本项目仿真模块只测试 state()模块即可。为了便于读者对整个项目有较深的了解，本项目仿真模块依旧是对 top()模块进行编写的。在仿真文件中设置计数器 time_all，从 0 到 1000 进行计数，由于之前仿真 debounce()模块的时候计数在 0 到 200 即可完成对 SW 的仿真，因此 time_all 中计数 0~200 时进行 SW1 或 SW2 的仿真、201~400 时进行第二个按键的仿真，以此类推，共有 5 次按键。同时设置寄存器 state，用来表示 time_all 在不同时刻按哪个按键，本项目仿真文件中 state 设置按 5 次 sw2（1 元）按键。

本项目仿真文件仿真源码如下：

```
    `timescale 1ns / 1ps
    module tb_top();
    reg  clk;                    //因为clk要进行赋值变化，所以为reg类型
    reg  rst_n;                  //和clk同理
    //reg    sw1;
    reg  sw2;
    reg[8:0] time1;
    reg[8:0] time2;
    reg[18:0]    time_all;
    reg[2:0] state;
    wire[3:0] led;               //此处要检测LED的信号，所以设置为wire类型
    initial                      //初始化开始
        begin
            clk=0;               //首先设置clk为0
```

```
            rst_n<=0;           //rst_n 为 0
            sw1<=0;
            sw2<=0;
            time1<=0;
            time2<=0;
            time_all<=0;
            state<=0;
            #100                //延时 100ns，为了便于查看波形，通常会进行一段时间延时
            rst_n<=1;           //复位按键置 1
        end
    always #10 clk=~clk;        //每过 10ns 后，clk 取反，这里 clk 时钟周期为 20ns
    //state
    always #20
        if(time_all<200)
            state <= 2;         //按 1 元按键
        else if((time_all>200)&&(time_all<400))
            state <= 2;         //按 1 元按键
        else if((time_all>400)&&(time_all<600))
            state <= 2;         //按 1 元按键
        else if((time_all>600)&&(time_all<800))
            state <= 2;         //按 1 元按键
        else if((time_all>800)&&(time_all<999))
            state <= 2;         //按 1 元按键
        else if(time_all == 1000)
            state <= 0;
    //time_all
    always #20
        if(time_all == 1000)
            time_all <= 1000;
        else
            time_all <= time_all +1;
    //time1
    always #20
        if((time1 == 200)&&(state == 1))
            time1 <= 0;
        else if(state == 1)
            time1 <= time1 +1;

    //sw1
    always #20
        if(((time1 < 20)||(time1>70&&time1<100))&&(state == 1))
            sw1 <= {$random};
        else if((time1 > 100)&&(state == 1))
            sw1 <= 0;
```

```
    else if(state == 1)
        sw1 <=1;
    else
        sw1 <= 0;
//time2
always #20
    if((time2 == 200)&&(state ==2))
        time2 <= 0;
    else if(state == 2)
        time2 <= time2 +1;
//sw2
always #20
    if(((time2 < 20)||(time2>70&&time2<100))&&(state == 2))
        sw2 <= {$random};
    else if((time2 > 100)&&(state == 2))
        sw2 <= 0;
    else if(state == 2)
        sw2 <=1;
    else
        sw2 <= 0;
top top_inst(              //将模块例化，并连线
.clk(clk),
.rst_n   (rst_n),
.sw1(sw1),
.sw2(sw2),
.led(led)
);
endmodule
```

3.4.7 仿真结果分析

本次仿真结果如图 3.23 所示，按动一次 sw2 之后，LED 值变为 4’b0011，表示有 1 元硬币投入；再次按动 sw2 之后，LED 值变为 4’b1111，表示有 2 元硬币投入；再次按动 sw2 之后，LED 值变为双流水灯，表示此时有 3 元硬币投入，自动售货机开始出货并找零。

改动仿真模块，变为投入 5 次 5 角硬币。投入第一个硬币时，LED 变为 0001；投入第二个硬币时，LED 变为 0011；投入第三个硬币时，LED 变为 0111；投入第四个硬币时，LED 变为 1111；投入第五个硬币时，LED 变为单向流水灯，结果如图 3.24 所示。

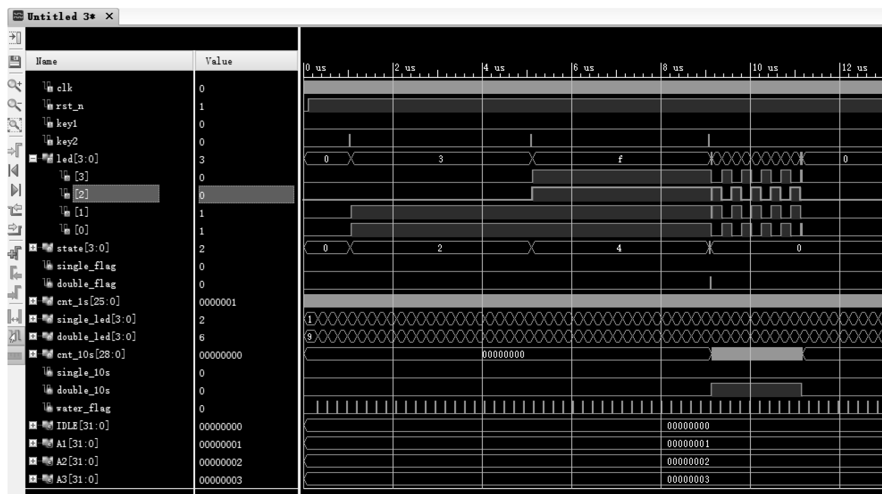

图 3.23　投入 3 元硬币

图 3.24　投入 5 角硬币

3.4.8　扩展训练

（1）读者可以设置投入 10 元、5 元、1 元纸币，根据货物金额进行出货和找零。

（2）可以增加自动售货机中的货物种类，根据不同的选择进行不同的找零。

3.5　总　　结

本章是 FPGA 初级设计部分，对 FPGA 的设计流程做了详细的讲解，方便初学者进行入门学习。读者务必要在完全掌握本章之后再继续学习后面的章节。