# 第1章 前言

计算机硬件系统由中央处理器（CPU）、存储器、输入/输出系统等部件构成，掌握计算机硬件基础知识，理解计算机系统各部件的基本组成、内部运行机制以及计算机系统运行的基本原理，是对计算机专业学生的基本要求。在此基础上，能具备一定的硬件系统设计能力，对今后从事硬件设计相关工作，或者是进行更为深入的学习和研究，都是至关重要的。

计算机专业课程体系中，硬件课程主要包括数字逻辑、计算机组成原理、计算机接口技术、嵌入式系统等。这些课程，从基本的门电路开始，系统讲解从逻辑门电路到计算机的基本组成部件的组成和功能，以及这些部件如何构成整机系统，执行计算机指令和程序的过程。其中，计算机组成原理（Computer Organization）是本科计算机专业的核心基础课程，在课程体系中占用重要的位置。它以程序设计语言及数字逻辑电路为先导课程，重点解析计算机硬件系统的基本组成、运行原理和协同工作机制，分析计算机组成对系统性能的影响，讲授计算机系统的基本设计方法，帮助学生建立计算机整机系统的概念，为学习系统结构、操作系统等课程提供扎实基础。

计算机硬件课程工程性和实践性都很强，教学实验是教学过程中不可缺少的环节，对同学们掌握理论知识，培养大家的设计能力都十分重要，教学实验设计也是一项挑战性工作。在实验内容上，不但要有能体现计算机各部件的基本硬件组成和内部运行原理的基础实验，也需要有能体现计算机内部协同工作机制、尤其是软硬件协同工作的系统实验，甚至还需要一些能结合如编译系统、操作系统课程内容的综合性实验供一些学有余力的同学选做；在实验方法上，既要满足硬件系列课程基础性的要求，也要采用当前主流的硬件系统设计和实现方法，以提高同学们的硬件系统设计和调试能力；在实验平台设计上，要注重体现系统观点，提供软硬件系统框架，以承载实验内容，并满足实验方法上的要求。

THINPAD（TsingHua mINi PAD）教学计算机是清华大学计算机系计算机组成原理教学组推出的系列教学计算机的新版本。它继承了原来TEC-2000系列教学机的面向系统、硬软件配置齐全、支持的实验可全面覆盖课程大纲要求的知识点等特点，同时，根据教学大纲将指令流水作为基本教学要求的实际情况，增强了对指令流水、高速缓存等实验内容的设置，并进一步扩展了外部接口，使THINPAD实验计算机更为适应新时代计算机组成原理课程教学的需要，同时，也能满足其他计算机硬件课程教学实验的要求。

具体来说，THINPAD实验计算机具有以下突出的特点：

1. 以计算机系统为实验设计目标，配置了基本齐全的软、硬件系统，使教学计算机在组成和系统结构上与实际的计算机基本相同，只是在功能上有所简化，性能上有降低；
2. 实验机字长16位，降低了成本，也缩小了实验的规模，能满足教学实验的要求；
3. 实验机指令系统全面采用MIPS指令格式（基本来自于MIPS 16e），与课堂教学采用的MIPS指令系统设计理念相同但细节上有所区别，可方便同学们在实验中加深对教学内容的理解，巩固课堂教学的效果；
4. 可支持丰富的教学实验，基本可覆盖计算机组成原理课程的教学重点，并能完成数字逻辑、计算机接口技术、系统结构课程的部分实验；
5. 利用硬件描述语言编程完成实验，使学生可在实验中掌握基本的硬件设计方法，培养硬件系统设计和调试能力；
6. 教学机配备了多种常用的外部设备接口，方便同学们在实验中选用。

图1.1对照计算机的层次结构，给出了THINPAD教学计算机系统的层次图。

图1.1 THINPAD实验计算机系统层次结构图

本教程作为计算机硬件系统实验教程，将以THINPAD教学计算机为实验平台，介绍它的硬件和软件系统的组成和功能，为方便同学们学习，粗略介绍了硬件描述语言VHDL的基本知识和编程方法，然后，设计了一系列教学实验，以指导同学们在学习计算机硬件课程，尤其是计算机组成原理课程时进行选做。全书共8章，各章主要内容分别为：

1. 前言，介绍教程的编写目的，实验教学计算机的主要特点，各章主要内容，教学实验安排的建议等。
2. 计算机硬件平台。介绍THINPAD教学计算机硬件的组成及连接方式，说明其在教学计算机中的作用。
3. 计算机软件平台。介绍THINPAD教学计算机使用的THCO MIPS指令系统，组成教学计算机软件系统的各程序的功能及使用方法，包括高级语言编译器、汇编程序、监控程序、指令模拟程序等。
4. VHDL硬件描述语言。本章简要介绍VHDL语言的基本知识，
5. ISE开发环境。本章主要介绍教学计算机核心芯片Xilinx，及其开发环境，通过示例帮助读者掌握硬件设计的基本流程。
6. 验证性实验。本章给出了计算机组成原理课程的一些验证性实验的要求，主要是计算机中各重要组成逻辑器件的基本功能和内部实现。验证性实验对掌握课堂教学内容有重要的帮助，同时，也是后续设计性实验的基础。
7. 设计性实验。本章给出了一些设计性实验的要求。设计性实验要求同学们在掌握课堂教学内容的基础上，运用所学知识设计计算机硬件的组成部件并进行调试，以帮助大家加深对知识点的理解。
8. 综合性实验。综合型实验要求同学在完成计算机组成课程学习的基础上，全面运用所学知识，设计和实现一台完整的计算机系统，通过实践检验课程的学习成果，培养综合运用知识解决实际问题的能力。

本教程中设计了3个层次的多个教学实验。这些实验内容十分经典，完成这些实验对同学们学习计算机组成原理知识肯定是大有裨益。但是，限于教学实验的课时，全部完成这些教学实验基本不太可能。我们建议大家根据各自的教学目标，从下面方案中选择一个完成。

1. 研究型大学计算机专业。研究型大学计算机专业本科生的目标是培养计算机设计人才，掌握计算机各组成部件的基本运行原理和运行机制是计算机组成原理课程的教学目的。教学实验应以完成综合性实验为教学目标，在此之前，可先完成如ALU实验、I/O实验等验证性实验，以及控制器设计等设计性实验，为完成综合性实验打好基础。
2. 应用型大学计算机专业。应用型大学计算机专业本科生的目标是培养计算机应用人才，相关硬件课程的教学目标是为同学们今后应用计算机技术打好基础，希望他们能掌握计算机的基本组成以及基本的运行机制。教学实验以完成设计性实验为目标，不需要完成综合型实验。
3. 非计算机专业。对这些同学的教学要求是能了解计算机的基本组成以及基本的运行原理即可，实验要求以挖成验证性实验为目标。

教学计算机系统历经多年研制，在此，作者十分感谢王诚教授对计算机组成原理课程教学所做出的贡献，他总结的教学方法和教学理念给了我们有力的指导，王老师还开创了对教学计算机的研制工作，对THINPAD的研制也给出了具体的建议。另外，清华大学计算机系**袁长俊、周源、陈勇、金昊衠、刘聪、张腾、刘亚楠、曹建勋、刘列、徐展凯、蒋生健、高学成等都对教学计算机的研制做出了贡献。**

本教程第1章至第 章由刘卫东编写， 由李山山编写，各部分实验均已在THINPAD教学计算机上进行了实现。限于时间和作者的水平和能力有限，书中可能还存在不足甚至错误，恳请广大读者批评指正。

# 第2章 THINPAD教学计算机硬件平台

## 2.1 概述

THINPAD教学计算机采用了目前流行的大规模可编程芯片FPGA为主要的实验芯片，存储器使用了大容量的SRAM和Flash存储芯片，数据总线采用指令和数据独立的存储结构，再加上丰富的外围接口，为计算机组成原理课程提供了更加灵活方便的实验方式。图2.1为THINPAD教学计算机硬件实验板。

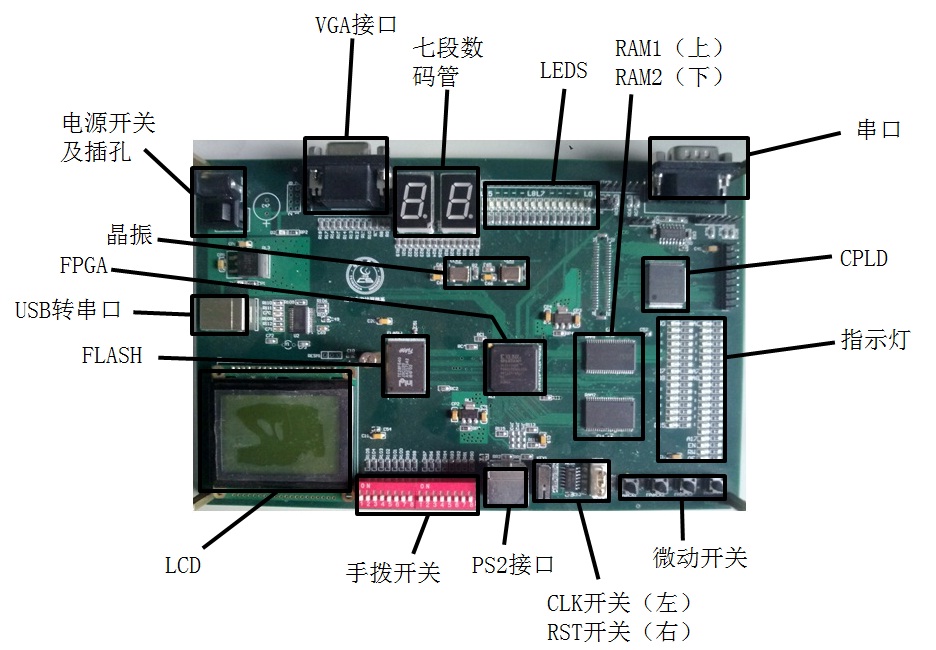


图2.1 THINPAD教学计算机硬件实验板

以往的计算机组成原理实验系统由于其实验体系结构相对固定，采用了现成的运算器或者控制器，导致实验过程中可完成的实验的灵活性和创新性大大减少，限制了学生在实验中的可发挥空间，同时实验内容很难更新和升级，使得实验对学生的吸引力逐年下降。为了解决这一问题，THINPAD教学计算机的主实验芯片采用了大规模可编程芯片，这样就使计算机硬件系统实验的灵活性大大增加，适合于各种不同的指令系统、总线构架、组成结构等，不仅仅可以完成以往的实验内容，更可以让学生在上面充分发挥，设计出各种具有创新性的实验，从而大大提高了实验的开放性和灵活性。采用可编程逻辑芯片这种方式使得实验内容不再由于教学计算机的体系结构的固定而造成实验的重复，教师也可以不断地改进和更新实验内容，以提高学生对实验的兴趣。再有一点就是，采用可编程逻辑器件使得实验系统可以兼顾到数字逻辑实验内容，实验者可以在上面完成很多数字逻辑和EDA的实验内容，实验平台上提供了数码管等接口设备，可以通过这些实验接口进行各种数字逻辑实验的设计和验证。

在以往的计算机组成原理实验系统中，如TEC2008，其提供的存储器容量都比较小，无法加载大规模的测试程序，并且所用的非易失性存储器为老式双列直插的EEPROM芯片，需要将其拔下来才能对其进行烧写，造成其损坏率很高而且很不方便。THINPAD教学计算机采用了两片大容量的SRAM存储器，配合上大容量的Flash存储器作为非易失的存储介质，并提供了在线对Flash进行读写的上位机软件，提高了实验的效率。同时THINPAD教学计算机继承了TEC系列的总线结构，为两片SRAM存储器提供了独立的地址和数据总线，实验过程中可以采用哈佛结构将指令存储器和数据存储器分开，也可以对两片SRAM统一编址作为一个16位的存储器使用，还可以通过位扩展的方式将存储器作为一个32位的存储器使用，从而支持32的实验。

THINPAD教学计算机提供了丰富的外围接口，包括数码管、输入开关、串口、VGA接口和键盘鼠标接口等。这些接口可以作为计算机组成原理课程实验中的实验接口使用，比如实验所要用到的串口与上位机进行通讯。还有实验硬件板上提供了很多数码管和输入开关，有了这些可以帮助实验者更好的进行调试，将可编程芯片内部的信号输出到数码管上进行显示，方便实验者查错。再者，有了VGA接口和鼠标键盘接口可以使计算机组成原理实验的模型机有了更好的可显示度，使得教学机可脱离上位机的控制，使用自身独立的输入输出系统，实现一个真正意义上的独立的计算机系统。同时，这些外围实验接口也可以作为数字逻辑及EDA实验用的接口，实验者可以通过控制这些接口来完成各种数字逻辑和EDA实验内容。

THINPAD教学计算机继承了TEC系列计算机的优点，采用了最新的实验方式和手段，经过了多年的实验教学验证和反复的系统改进，从而形成了目前较为完整的计算机硬件系统实验体系。

## 2.2可编程逻辑器件简述

### 2.2.1 可编程逻辑器件

我们一般使用的逻辑器件可分类两大类：固定逻辑器件（ASIC）和可编程逻辑器件(PLD)。

固定逻辑器件中的电路顾名思义就固定不可更改的，它们被设定用来完成一种或一组功能，从出厂后就确定无法改变，用户根据需要可以选择不同功能的器件，用来完成自己的设计。对于固定逻辑器件，其电路和功能都是生产厂家设定的，根据器件复杂性的不同，生产厂家会花费数月到几年的时间来设计、验证直到生产这些芯片，而且芯片设计制造过程中需要投入大量的人力和物力成本，包括工程资源、昂贵的软件设计工具、用来制造芯片不同金属层的昂贵光刻掩模组，以及初始原型器件的生产成本等，这些都导致芯片的设计生产是一个非常昂贵而且费时的过程。在这过程中如果出现设计失误，或者功能不能满足需要，这就需要重新设计和生产芯片,造成很大的损失。因此，厂家一般不会改变固定逻辑器件的逻辑电路，而普通使用者也很难去设计和生产自己需要的器件，这样就导致了后来可编程逻辑器件的产生。

可编程逻辑器件的出现解决了固定芯片的一些问题，它能够很方便的改变自身的逻辑功能，为使用者提供了更加方便的设计方式。可编程逻辑器件的英文全称为programmable logic device，一般缩写为PLD。它也是厂家提供的一种标准逻辑芯片，厂家也投入了巨大的成本来生产这些芯片，但是这些芯片的内部逻辑并没有被固定化，它可以支持用户修改内部逻辑，就能够为用户提供范围广泛的多种逻辑能力、特性、速度和电压特性，而且这些器件可在随时改变逻辑，从而完成许多种不同的功能，从而大大方便了逻辑设计。一般的PLD的集成度很高，足以满足设计一般的数字系统的需要，用户只要对器件编程来确定其逻辑功能，这样就可以由用户自行编程而把一个数字系统下载在一片PLD上，而不必去花昂贵的费用请芯片制造厂商设计和制作专用的集成电路芯片了。

可编程逻辑器件的生产厂家都会提供相应的开发软件，设计人员可利用价格低廉的软件工具快速开发、仿真和测试其设计，并可以很方便的将设计下载到器件中，在实际运行的电路中对设计进行测试。采用可编程器件的另一个主要优点就是在设计阶段中用户可根据需要修改电路，直到对设计达到功能需求。

早期的可编程逻辑器件只有可编程只读存贮器(PROM)、紫外线可擦除只读存贮器(EPROM)和电可擦除只读存贮器(EEPROM)三种。由于结构的限制，它们只能完成简单的数字逻辑功能。后来，出现了结构上稍复杂的可编程芯片，即可编程逻辑器件，它能够完成各种数字逻辑功能。典型的PLD由一个“与”门和一个“或”门阵列组成，而任意一个组合逻辑都可以用“与一或”表达式来描述，所以，PLD能以乘积和的形式完成大量的组合逻辑功能，例如我们常见的GAL。这些早期的PLD器件的一个共同特点是可以实现速度特性较好的逻辑功能，但其过于简单的结构也使它们只能实现规模较小的电路。为了弥补这一缺陷，20世纪80年代中期Altera和Xilinx分别推出了类似于PAL结构的扩展型CPLD和与标准门阵列类似的FPGA，它们都具有体系结构和逻辑单元灵活、集成度高以及适用范围宽等特点。这两种器件兼容了PLD和通用门阵列的优点，可实现较大规模的电路，编程也很灵活。

可编程逻辑器件按集成度划分可分为高低两类，早先出现的PROM、PAL、可重复编程的GAL都属于低集成度芯片，可重构使用的逻辑门数量很少，也被称为简单PLD。现在大量使用的CPLD、FPGA器件是高集成度芯片，也被称为复杂PLD。如果按照内部结构划分，可分成乘积项结构器件和查找表结构器件两大类。其基本结构为“与-或”阵列的器件是乘积项结构器件，大部分简单PLD和CPLD都属于这个范畴。由简单的查找表组成可编程门，再构成阵列形式的器件是查找表结构器件，大多数FPGA是属于此类器件。

THINPAD教学计算机上CPLD和FPGA都使用了，后面会详细介绍用到的这两类芯片。

### 2.2.2 FPGA工作原理及内部结构

一般的FPGA是基于查找表结构的，下面简单介绍一下查找表的原理。根据数字逻辑电路的基本知识可以知道，对于一个n输入的逻辑运算，不管是与或非运算还是异或运算等等，输出最多只可能存在2种结果。所以如果事先将所有的输出可能都存放于一个存储器中，根据输入的不同将相应的结果从存储器中取出作为输出，整个过程就像查找对数表一样，这样就相当于实现了与非门电路的功能。FPGA的工作原理就是如此，它通过软件生成存储器的内容然后将内容烧写到查找表存储器中，从而在相同的电路情况下实现了不同的逻辑功能。

查找表（Look-Up-Table）简称为LUT，LUT本质上就是一个RAM。目前FPGA中多使用四输入的LUT，所以每一个LUT可以看成一个有四位地址线的RAM。当用户通过原理图或HDL语言描述了一个逻辑电路以后，FPGA开发软件会自动计算逻辑电路的所有可能结果，并把真值表（即结果）事先写入RAM，这样，每输入一组信号进行逻辑运算就等于输入一个地址进行查表，找出地址对应的内容，然后输出即可。

下面以一个4与门电路的例子来说明LUT实现逻辑功能的原理。表2.1是一个使用LUT实现4输入与门电路的真值表。

表2.1 输入与门的真值表

|  |  |  |  |
| --- | --- | --- | --- |
| 逻辑真值表 | | LUT | |
| 输入 | 输出 | RAM地址 | 内容 |
| 0000 | 0 | 0000 | 0 |
| 0001 | 0 | 0001 | 0 |
| …… | …… | …… | …… |
| 1111 | 1 | 1111 | 1 |

从表中可以看到，LUT具有和逻辑电路相同的功能。对于更加复杂的逻辑，在实际中会将很多个LUT连接起来，一些LUT的输出作为其他LUT的输入，这样就能实现更大规模的逻辑，由于基于LUT的FPGA具有很高的集成度，其器件密度从数万门到数千万门不等，可以完成极其复杂的时序逻辑与组合逻辑电路功能，所以适用于高速、高密度的高端数字逻辑电路设计领域。

目前流行的FPGA除了基于查找表技术之外还整合了一些常用的功能（如RAM、时钟管理等）的硬核（ASIC型）模块。如图2.2为FPGA的一般结构，主要由五部分组成：可编程输入输出单元、基本可编程逻辑单元、时钟管理、嵌入式块RAM、布线资源。

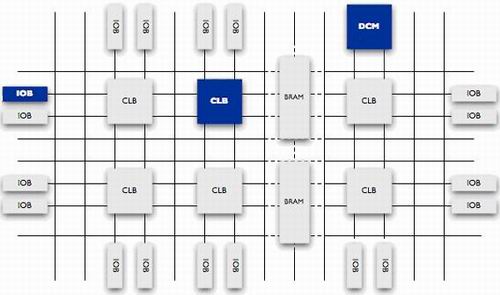


图2.2 FPGA芯片的内部结构

#### 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，完成不同电气特性下对输入/输出信号的驱动与匹配要求，其示意结构如图2.3所示。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，可以调整驱动电流的大小，可以改变上、下拉电阻。

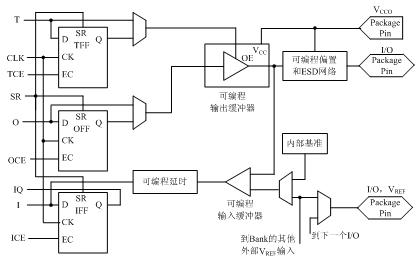


图2.3 典型的IOB内部结构示意图

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。为了便于管理和适应多种电器标准，FPGA的IOB被划分为若干个组（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。

#### 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个输入、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个或2个）相同的Slice和附加逻辑构成，如图2.4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。

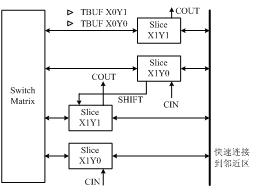


图2.5 典型的CLB结构示意图

Slice是Xilinx公司定义的基本逻辑单位，其内部结构如图2.6所示，一个Slice由两个4输入的函数、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实现两位全加操作，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；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。

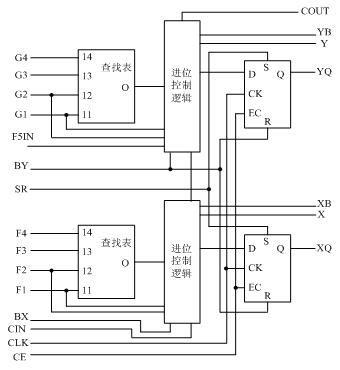


图2.6 典型的4输入Slice结构示意图

#### 数字时钟管理模块（DCM）

业内大多数FPGA均提供数字时钟管理（Xilinx的全部FPGA均具有这种特性）。Xilinx推出最先进的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。

#### 嵌入式块RAM（BRAM）

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。除了块RAM，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。单片块RAM的容量为18k比特，即位宽为18比特、深度为1024，可以根据需要改变其位宽和深度。

#### 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类不同的类别。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。

在实际中设计者不需要直接选择布线资源，布局布线器可自动地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。

### 2.2.3 CPLD工作原理及内部结构

一般的CPLD都是采用基于乘积项（Product-Term)的PLD结构，如图2.7所示是一款CPLD的总体结构：

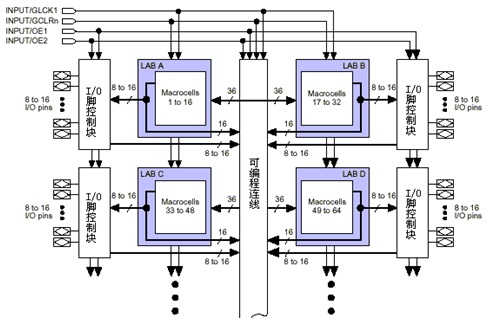


图2.7 基于乘积项的结构

这种PLD可分为三块结构：宏单元（Marocell)，可编程连线（PIA)和I/O控制块。宏单元是PLD的基本结构，由它来实现基本的逻辑功能。图2.7中深色部分是多个宏单元的集合（因为宏单元较多，没有一一画出）。可编程连线负责信号传递，连接所有的宏单元。I/O控制块负责输入输出的电气特性控制，比如可以设定集电极开路输出，三态输出等。INPUT/GCLK1，INPUT/GCLRn,INPUT/OE1,INPUT/OE2是全局时钟，清零和输出使能信号，这几个信号有专用连线与PLD中每个宏单元相连，信号到每个宏单元的延时相同并且延时最短。

宏单元的具体结构见下图：

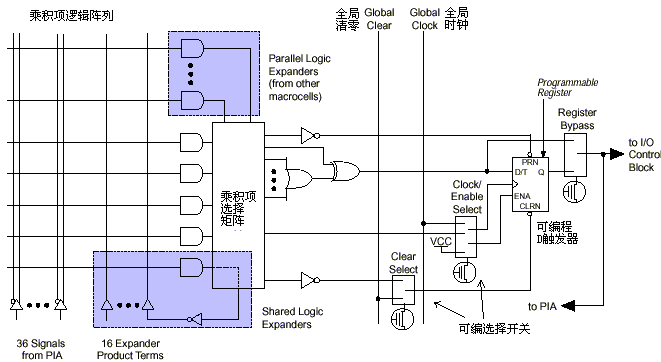


图2.8 宏单元结构

左侧是乘积项阵列，实际就是一个与或阵列，每一个交叉点都是一个可编程熔丝，如果导通就是实现“与”逻辑。后面的乘积项选择矩阵是一个“或”阵列。两者一起完成组合逻辑。图右侧是一个可编程D触发器，它的时钟，清零输入都可以编程选择，可以使用专用的全局清零和全局时钟，也可以使用内部逻辑（乘积项阵列）产生的时钟和清零。如果不需要触发器，也可以将此触发器旁路，信号直接输给PIA或输出到I/O脚。

下面我们以一个简单的电路为例,具体说明PLD是如何利用以上结构实现逻辑的，电路如下图：

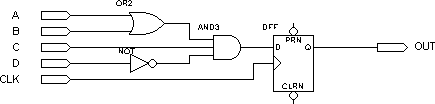


图2.9

假设组合逻辑的输出(AND3的输出)为f，则f=(A+B)\*C\*(!D)=A\*C\*!D + B\*C\*!D

PLD将以下面的方式来实现组合逻辑f:

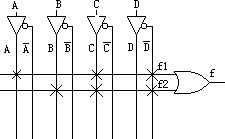


图2.10

A,B,C,D由PLD芯片的管脚输入后进入可编程连线阵列（PIA)，在内部会产生A,A反,B,B反,C,C反,D,D反8个输出。图中每一个叉表示相连（可编程熔丝导通），所以得到：f= f1 + f2 = (A\*C\*!D) + (B\*C\*!D) 。这样组合逻辑就实现了。 图2.9电路中D触发器的实现比较简单，直接利用宏单元中的可编程D触发器来实现。时钟信号CLK由I/O脚输入后进入芯片内部的全局时钟专用通道，直接连接到可编程触发器的时钟端。可编程触发器的输出与I/O脚相连，把结果输出到芯片管脚。这样PLD就完成了图2.9所示电路的功能。（以上这些步骤都是由软件自动完成的，不需要人为干预）

图2.9的电路是一个很简单的例子，只需要一个宏单元就可以完成。但对于一个复杂的电路，一个宏单元是不能实现的，这时就需要通过并联扩展项和共享扩展项将多个宏单元相连，宏单元的输出也可以连接到可编程连线阵列，再作为另一个宏单元的输入。这样PLD就可以实现更复杂逻辑。这种基于乘积项的PLD基本都是由EEPROM和Flash工艺制造的，一上电就可以工作，无需其他芯片配合。

## 2.3 教学机总体结构

THINPAD教学计算机的总体硬件结构如图2.11所示。整个硬件以大规模可编程逻辑器件为中心，通过总线连接SRAM存储器和Flash存储器，再配合以外围各种接口。



图2.11 THINPAD实验计算机硬件结构图

THINPAD教学计算机包括实验芯片、扩展芯片、存储器、手拨开关、指示灯、RS232通信接口、USB转串口芯片、FLASH存储器、下载电路以及各类实验用接口，其中：

* 实验芯片为一片可编程逻辑器件，是一片FPGA，每次上电都需要重新配置。实验的主要的内容在实验芯片上面完成，实验者在PC上完成设计并生成可可下载文件后就可以将该文件配置到实验芯片中，然后就可以在实验板上进行实验和调试了。实验芯片通过基本数据总线、基本地址线、基本内存读写线、扩展信号线和扩展芯片相连；通过基本数据总线、基本地址线、基本内存读写线和基本存储器相连；通过扩展数据总线、扩展地址线、扩展内存读写线和扩展存储器相连；通过基本数据总线和手拨开关相连；通过基本数据总线、基本地址线、基本内存读写线和指示灯相连；通过Flash数据线、Flash地址线、Flash控制线和FLASH存储器相连。
* 扩展芯片也是一片可编程逻辑器件，不过它是非易失的，掉电后里面的配置内容还存在，不需要每次上电后重新配置。扩展芯片主要用于RS232串口通信的控制，并提供其他扩展功能，包括电平匹配等。扩展芯片直接连接到了基本地址和数据线上，它可以直接与实验芯片通信，也可以用来直接访问基本存储器。扩展芯片连接了RS232串行通讯接口，扩展芯片可以作为该串口得控制器，完成数据的收发。同时扩展芯片上连接了一个鼠标键盘的PS2接口，可以把扩展芯片配置成PS2的控制器，也可以将其作为一个数据通路直接连通到实验芯片上，并保证其电平的匹配。
* 教学计算机的存储器分为2个存储芯片，分别为基本内存和扩展内存，是一样的SRAM存储器，二者具有独立的数据总线和地址总线，可以用来分别存储指令以及数据，也可以统一编址存储或者作位扩展使用。
* 在基本数据总线上还连接了手拨开关，它可以用于向基本数据总线拨入数据。在计算机组成原理实验中，如果需要直接设置内存总线上的内容，就可以直接从手拨开关输入；在数字逻辑实验中手拨开关也可以作为输入使用。
* 在基本数据总线上也连接了指示灯，用于观察基本总线的内容，在实验过程中往往需要很直观的方式来查看总线的内容，这个指示灯可以很清晰的显示当前数据线及地址线的内容，方便实验者观察指令或数据的变化。同时实验芯片也直接连接了16个指示灯，还有2个7段数码管，实验者可以使用这些扩展指示灯来观察实验芯片内部的信号，或者作为数字逻辑实验的显示手段。
* 在以往的TEC系列的实验计算机中RS232串行通信接口是用来连接实验计算机和上位机软件的重要手段，在THINPAD上依然保留了串行接口，不过控制器使用了可编程器件代替了以前的8251。通过该串行接口可以与上位机软件进行通讯，上位机软件也可以通过串行接口访问SRAM存储器和Flash存储器。
* FLASH存储器是用来替代以往的EEPROM存储器，用来存储启动代码或者一些需要断电保存的数据。
* 下载电路用于通过JTAG方式配置实验芯片和扩展芯片。
* 各类实验用接口，包括鼠标键盘接口、VGA接口、点阵以及USB串口等，这些接口可以直接作为计算机组成原理实验，也可以用于进行各种数字逻辑实验。
* 实验板上提供了两种时钟频率的晶振，分别为50MHz和11.0592MHz，其中50MHz的为实验用主时钟，11.0592MHz的为串口控制器所使用。

THINPAD教学计算机在实验开展上参考了如下的实验流程：

* 针对部件实验及数字逻辑实验，实验者在PC上完成代码设计及生成可下载文件，然后通过下载电路将实验的硬件代码配置到实验芯片中，并根据需要通过手拨开关将实验数据发送给实验芯片，然后通过观察指示灯或者各类实验用接口就能够观察到实验结果并进行具体的实验操作及调试；
* 针对计算机组成原理模型机设计实验，实验者在PC上完成代码设计及仿真并生成可下载硬件文件，然后将RS232通信控制程序通过下载电路配置到扩展芯片中，并使用上位机软件通过RS232通信接口、扩展芯片和实验芯片将实验测试程序及实验数据下载到FLASH存储器中，接着通过下载电路将实验硬件代码配置到实验芯片中，该代码能够将FLASH存储器中的实验测试程序及实验数据根据需要分别下载到SRAM存储器中，之后实验芯片就能够从SRAM存储器读出实验测试程序及实验数据并运行，运行结果会写入SRAM存储器中、显示到指示灯以及通过RS232通信接口、扩展芯片发送给上位机软件，如果上位机的监控程序能够正确运行则表示实验成功。

## 2.3 实验芯片

THINPAD教学计算机使用了大规模可编程逻辑芯片为主要的实验载体，有了可编程芯片，实验的灵活性和开放性得到了很大的提高。实验板上安排了两片可编程逻辑芯片，一片是FPGA，用来实现主要的实验内容，另外一片为扩展芯片，是一片CPLD，用来提供扩展功能。一般来说，只要使用FPGA就可以完成所需的实验内容。

### 2.3.1 ＦＰＧＡ芯片

THINPAD教学计算机上的主实验芯片是一片FPGA，是由Xilinx公司生产的Spartan-3E系列的XC3S1200EFGG320芯片，320个管脚，120万门,FBGA封装。这一系列的FPGA是专门为大容量、低成本的电子应用而设计的，Spartan-3E系列共有5款不同容量的FPGA，其密度从10万到160万门不等，Spartan-3E系列相对于以前的Spartan-IIE系列，增加了逻辑资源、内部RAM容量、I/O引脚数量、时钟管理功能以及总体性能，这些增强的功能都来自于Virtex-II的技术。这些结合了先进处理技术的改进，使得Spartan-3E的性价比超出以前所能达到的水平，也为可编程逻辑器件提供了新的标准。由于价格低廉，Spartan-3E可广泛地应用于各种电子设计，包括军工航天、宽带接入、家庭网络、投影电视、数字电视等。

Spartan-3E系列FPGA产品的主要技术参数指标如表2.2所示。

表2.2 Spartan-3E系列FPGA技术参数

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 器件名称 | 逻辑单元 | 系统门数 | CLB阵列 | CLB总数 | 最大用户I/O | BlockRAM容量 bit |
| XC3S100E | 2160 | 100k | 22×16 | 240 | 108 | 72k |
| XC3S250E | 5508 | 250k | 34×26 | 612 | 172 | 216k |
| XC3S500E | 10476 | 500k | 46×34 | 1164 | 232 | 360k |
| XC3S1200E | 19512 | 1200k | 60×46 | 2168 | 304 | 504k |
| XC3S1600E | 33192 | 1600k | 76×58 | 3688 | 376 | 648k |

Spartan-3E系列FPGA结构包括5个基本的可编程功能单元：

（1）可配置逻辑块（CLB）。该模块包括基于RAM的查找表(LUT)。除了作为存储器外，CLB还能通过编程实现很多的逻辑功能。

（2）输入/输出模块（IOB）。该模块控制I/O引脚和内部逻辑单元之间的数据流动，每一个IOB支持双向三态的数据流动，支持23种差分信号标准（其中有6种高性能差分标准）。输入/输出模块还包括了双数据速率寄存器（Double Data-Rate,DDR），数控电阻（Digitally controlled Impedance, DCI）提供自动的片内终端，大大简化了电路的设计。

（3）Block RAM模块。该模块提供了18Kbit的双口数据存储。

（4）乘法器模块。该模块提供了18位的二进制数据乘法。

（5）数字时钟管理模块(Digital Clock Manager,DCM)。该模块提供了自校准、全数字的解决方案，可以提供分布式的、延时的、合成频率的、分频的以及移相的时钟信号。

实验芯片上连接了存储器和各种的接口，实验者只要完成自己的设计，并分配相应的管脚，实用软件编译并配置到实验FPGA中就可以进行实验了。具体的管脚分配表如表2.3所示。

表2.3 FPGA管脚分配表

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **信号** | **管脚** | **属性** | **说明** | **信号** | **管脚** | **属性** | **说明** |
| FLASH\_A0 | T2 | OUT | Flash地址线 | FLASH\_D0 | J5 | INOUT | Flash数据线 |
| FLASH\_A1 | T1 | OUT | FLASH\_D1 | J4 | INOUT |
| FLASH\_A2 | R3 | OUT | FLASH\_D2 | J2 | INOUT |
| FLASH\_A3 | R2 | OUT | FLASH\_D3 | J1 | INOUT |
| FLASH\_A4 | P4 | OUT | FLASH\_D4 | H6 | INOUT |
| FLASH\_A5 | P3 | OUT | FLASH\_D5 | H5 | INOUT |
| FLASH\_A6 | P2 | OUT | FLASH\_D6 | H4 | INOUT |
| FLASH\_A7 | P1 | OUT | FLASH\_D7 | H3 | INOUT |
| FLASH\_A8 | N5 | OUT | FLASH\_D8 | H2 | INOUT |
| FLASH\_A9 | N4 | OUT | FLASH\_D9 | H1 | INOUT |
| FLASH\_A10 | M6 | OUT | FLASH\_D10 | G6 | INOUT |
| FLASH\_A11 | M5 | OUT | FLASH\_D11 | G5 | INOUT |
| FLASH\_A12 | M4 | OUT | FLASH\_D12 | G4 | INOUT |
| FLASH\_A13 | M3 | OUT | FLASH\_D13 | G3 | INOUT |
| FLASH\_A14 | L6 | OUT | FLASH\_D14 | F2 | INOUT |
| FLASH\_A15 | L5 | OUT | FLASH\_D15 | F1 | INOUT |
| FLASH\_A16 | L4 | OUT | FPGA\_KEY0 | U11 | IN | 微动开关 |
| FLASH\_A17 | L3 | OUT | FPGA\_KEY1 | U14 | IN |
| FLASH\_A18 | L2 | OUT | FPGA\_KEY2 | V14 | IN |
| FLASH\_A19 | L1 | OUT | FPGA\_KEY3 | V16 | IN |
| FLASH\_A20 | K6 | OUT | CLK0 | B8 | IN | 11.0592M时钟晶振 |
| FLASH\_A21 | K5 | OUT |
| FLASH\_A22 | K4 | OUT | CLK1 | B9 | IN | 50M时钟晶振 |
| LCD\_CS1 | P6 | OUT | LCD点阵信号 | SW\_DIP0 | J6 | IN | 左侧手拨开关信号 |
| LCD\_CS2 | R8 | OUT | SW\_DIP1 | J7 | IN |
| LCD\_DB0 | T5 | INOUT | SW\_DIP2 | K2 | IN |
| LCD\_DB1 | T4 | INOUT | SW\_DIP3 | K7 | IN |
| LCD\_DB2 | N8 | INOUT | SW\_DIP4 | M1 | IN |
| LCD\_DB3 | R6 | INOUT | SW\_DIP5 | N1 | IN |
| LCD\_DB4 | R5 | INOUT | SW\_DIP6 | N2 | IN |
| LCD\_DB5 | P9 | INOUT | SW\_DIP7 | R1 | IN |
| LCD\_DB6 | P8 | INOUT | SW\_DIP8 | R4 | IN | 右侧手拨开关信号 |
| LCD\_DB7 | P7 | INOUT | SW\_DIP9 | U1 | IN |
| LCD\_E | T8 | OUT | SW\_DIP10 | V2 | IN |
| LCD\_RESET | N7 | OUT | SW\_DIP11 | V3 | IN |
| LCD\_RS | U3 | OUT | SW\_DIP12 | V4 | IN |
| LCD\_RW | U4 | OUT | SW\_DIP13 | U8 | IN |
| VGA\_B0 | D5 | OUT | VAG接口信号 | SW\_DIP14 | R7 | IN |
| VGA\_B1 | A6 | OUT | SW\_DIP15 | T7 | IN |
| VGA\_B2 | B6 | OUT | FLASH\_BYTE# | C1 | OUT | Flash控制信号 |
| VGA\_G0 | B4 | OUT | FLASH\_CE | E4 | OUT |
| VGA\_G1 | C4 | OUT | FLASH\_CE1 | E3 | OUT |
| VGA\_G2 | C5 | OUT | FLASH\_CE2 | E2 | OUT |
| VGA\_HHYNC | E6 | OUT | FLASH\_OE | E1 | OUT |
| VGA\_R0 | A7 | OUT | FLASH\_RP# | D1 | OUT |
| VGA\_R1 | C3 | OUT | FLASH\_STS | D2 | OUT |
| VGA\_R2 | A4 | OUT | FLASH\_VPEN | C2 | OUT |
| VGA\_VHYNC | D6 | OUT | FLASH\_WE | D4 | OUT |
| PS2KB\_CLOCK | V7 | OUT | PS2信号 | U\_RXD | V5 | OUT | 串口信号 |
| PS2KB\_DATA | V6 | IN | U\_TXD | K3 | OUT |
| RAM1DATA0 | J12 | INOUT | RAM1数据线 | RAM2DATA0 | U13 | INOUT | RAM2数据线 |
| RAM1DATA1 | J13 | INOUT | RAM2DATA1 | T16 | INOUT |
| RAM1DATA2 | J14 | INOUT | RAM2DATA2 | T15 | INOUT |
| RAM1DATA3 | J15 | INOUT | RAM2DATA3 | T14 | INOUT |
| RAM1DATA4 | J16 | INOUT | RAM2DATA4 | T12 | INOUT |
| RAM1DATA5 | J17 | INOUT | RAM2DATA5 | R13 | INOUT |
| RAM1DATA6 | K12 | INOUT | RAM2DATA6 | R14 | INOUT |
| RAM1DATA7 | K13 | INOUT | RAM2DATA7 | R12 | INOUT |
| RAM1DATA8 | K14 | INOUT | RAM2DATA8 | R11 | INOUT |
| RAM1DATA9 | K15 | INOUT | RAM2DATA9 | R10 | INOUT |
| RAM1DATA10 | L15 | INOUT | RAM2DATA10 | P13 | INOUT |
| RAM1DATA11 | L16 | INOUT | RAM2DATA11 | P12 | INOUT |
| RAM1DATA12 | L17 | INOUT | RAM2DATA12 | P11 | INOUT |
| RAM1DATA13 | L18 | INOUT | RAM2DATA13 | P10 | INOUT |
| RAM1DATA14 | M13 | INOUT | RAM2DATA14 | N12 | INOUT |
| RAM1DATA15 | M14 | INOUT | RAM2DATA15 | N11 | INOUT |
| RAM1\_EN | M15 | OUT | RAM1控制信号 | RAM2\_EN | N10 | OUT | RAM2控制信号 |
| RAM1\_OE | M16 | OUT | RAM2\_OE | R9 | OUT |
| RAM1\_RW | M18 | OUT | RAM2\_RW | M9 | OUT |
| FPGA\_LED8 | A12 | OUT | 右侧LED数码管信号 | FPGA\_LED0 | D10 | OUT | 左侧LED数码管信号 |
| FPGA\_LED9 | E12 | OUT | FPGA\_LED1 | E10 | OUT |
| FPGA\_LED10 | F12 | OUT | FPGA\_LED2 | A11 | OUT |
| FPGA\_LED11 | A13 | OUT | FPGA\_LED3 | B11 | OUT |
| FPGA\_LED12 | B13 | OUT | FPGA\_LED4 | C11 | OUT |
| FPGA\_LED13 | D13 | OUT | FPGA\_LED5 | D11 | OUT |
| FPGA\_LED14 | E13 | OUT | FPGA\_LED6 | E11 | OUT |
| FPGA\_LED15 | A14 | OUT | FPGA\_LED7 | F11 | OUT |
| RAM1ADDR0 | C17 | OUT | RAM1地址线 | RAM2ADDR0 | N14 | OUT | RAM2地址线 |
| RAM1ADDR1 | C18 | OUT | RAM2ADDR1 | N15 | OUT |
| RAM1ADDR2 | D16 | OUT | RAM2ADDR2 | N18 | OUT |
| RAM1ADDR3 | D17 | OUT | RAM2ADDR3 | P16 | OUT |
| RAM1ADDR4 | E15 | OUT | RAM2ADDR4 | P17 | OUT |
| RAM1ADDR5 | E16 | OUT | RAM2ADDR5 | P18 | OUT |
| RAM1ADDR6 | F14 | OUT | RAM2ADDR6 | R15 | OUT |
| RAM1ADDR7 | F15 | OUT | RAM2ADDR7 | R16 | OUT |
| RAM1ADDR8 | F17 | OUT | RAM2ADDR8 | R18 | OUT |
| RAM1ADDR9 | F18 | OUT | RAM2ADDR9 | T17 | OUT |
| RAM1ADDR10 | G13 | OUT | RAM2ADDR10 | T18 | OUT |
| RAM1ADDR11 | G14 | OUT | RAM2ADDR11 | U18 | OUT |
| RAM1ADDR12 | G15 | OUT | RAM2ADDR12 | V15 | OUT |
| RAM1ADDR13 | G16 | OUT | RAM2ADDR13 | V13 | OUT |
| RAM1ADDR14 | H14 | OUT | RAM2ADDR14 | V12 | OUT |
| RAM1ADDR15 | H15 | OUT | RAM2ADDR15 | V9 | OUT |
| RAM1ADDR16 | H16 | OUT | RAM2ADDR16 | U16 | OUT |
| RAM1ADDR17 | H17 | OUT | RAM2ADDR17 | U15 | OUT |
| DYP0A | C7 | OUT | 左侧七段数码管信号 | DYP1A | C9 | OUT | 右侧七段数码管信号 |
| DYP0B | D7 | OUT | DYP1B | D9 | OUT |
| DYP0C | E7 | OUT | DYP1C | E9 | OUT |
| DYP0D | F7 | OUT | DYP1D | F9 | OUT |
| DYP0E | A8 | OUT | DYP1E | G9 | OUT |
| DYP0F | E8 | OUT | DYP1F | A10 | OUT |
| DYP0G | F8 | OUT | DYP1G | B10 | OUT |
| CLK\_FROM\_KEY | U9 | IN | 单步时钟源 | RESET | U10 | IN | 硬件复位 |
| InterConn0 | A16 | INOUT | 与CPLD通讯信号 |  |  |  |  |
| InterConn1 | B14 | INOUT |  |  |  |  |
| InterConn2 | B16 | INOUT |  |  |  |  |
| InterConn3 | C14 | INOUT |  |  |  |  |
| InterConn4 | D14 | INOUT |  |  |  |  |
| InterConn5 | N9 | INOUT |  |  |  |  |
| InterConn6 | U5 | INOUT |  |  |  |  |
| InterConn7 | U6 | INOUT |  |  |  |  |
| InterConn8 | V6 | INOUT |  |  |  |  |
| InterConn9 | V7 | INOUT |  |  |  |  |

### 2.3.2 ＣＰＬＤ芯片

THINPAD教学计算机上的扩展芯片是一片CPLD，是由Xilinx公司生产的XC9500系列的XC95144XL-7TQ100芯片，100个管脚，144个宏单元,TQFP封装。此款芯片使用3.3V电压，适用于高性能、低电压需用的通信和计算机系统,它由8个54V18功能模块组成，可提供3200个5ns延迟可用门。

XC9500XL系列被广泛地应用于通信、网络和计算机等产品中。该系列器件采用快闪存储技术（FastFlash），比EECMOS工艺的速度更快，功耗更低。目前，Xilinx公司XC9500XL系列宏单元数达到288个，系统时钟可达到200MHz。器件支持PCI总线规范和JTAG边界扫描测试功能，具有在系统可编程功能。该系列的芯片，兼容5V、3.3V电平，因此可以用来进行5V和3.3V电压的转换及匹配，比如实验板上的PS2接口就是通过CPLD转换后接到FPGA上的。

扩展芯片通过基本数据线和实验FPGA及基本存储器RAM1连接到一块，扩展芯片上也连接了串口、PS2接口，而且扩展芯片和实验芯片之间也有扩展互联线，作为扩展使用。在实验过程中扩展芯片可以辅助完成一些任务，比如当做串口控制器（8251），PS2的电平转换等，还可以充当ROM供实验芯片使用。表2.4是扩展芯片的管脚分配表。

表2.4 CPLD的管脚分配

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **信号** | **管脚** | **属性** | **说明** | **信号** | **管脚** | **属性** | **说明** |
| Ram1Data0 | p6 | INOUT | 数据总线 | Ram1\_EN | p96 | OUT | RAM1控制信号 |
| Ram1Data1 | p7 | INOUT | Ram1\_OE | p95 | OUT |
| Ram1Data2 | p8 | INOUT | Ram1\_RW | p94 | OUT |
| Ram1Data3 | p9 | INOUT | CLK0 | p1 | IN | 11.0592M时钟晶振 |
| Ram1Data4 | p10 | INOUT |
| Ram1Data5 | p11 | INOUT | CLK\_FROM\_KEY | p3 | IN | 单步时钟源 |
| Ram1Data6 | p12 | INOUT | CLK1 | p2 | IN | 50M时钟晶振 |
| Ram1Data7 | p13 | INOUT | Ram1Addr0 | p23 | OUT | 地址总线 |
| Ram1Data8 | p14 | INOUT | Ram1Addr1 | p24 | OUT |
| Ram1Data9 | p15 | INOUT | Ram1Addr2 | p25 | OUT |
| Ram1Data10 | p16 | INOUT | Ram1Addr3 | p27 | OUT |
| Ram1Data11 | p17 | INOUT | Ram1Addr4 | p28 | OUT |
| Ram1Data12 | p18 | INOUT | Ram1Addr5 | p29 | OUT |
| Ram1Data13 | p19 | INOUT | Ram1Addr6 | p30 | OUT |
| Ram1Data14 | p20 | INOUT | Ram1Addr7 | p32 | OUT |
| Ram1Data15 | p22 | INOUT | Ram1Addr8 | p33 | OUT |
| InterConn9 | p36 | INOUT | 与FPGA通讯信号 | Ram1Addr9 | p34 | OUT |
| InterConn8 | p37 | INOUT | Ram1Addr10 | p35 | OUT |
| InterConn7 | p39 | INOUT | PS2KB\_CLOCK | p71 | OUT | PS2信号 |
| InterConn6 | p40 | INOUT | PS2KB\_DATA | p70 | INOUT |
| InterConn5 | p41 | INOUT | RXD1 | p58 | OUT | 串口1控制信号 |
| InterConn4 | p42 | INOUT | TXD1 | p54 | INOUT |
| InterConn3 | p43 | INOUT | RXD2 | p56 | INOUT | 串口2控制信号 |
| InterConn2 | p46 | INOUT | TXD2 | p55 | INOUT |
| InterConn1 | p49 | INOUT | Reset | p4 | IN | 硬件复位 |
| InterConn0 | p50 | INOUT |  |  |  |  |

在完成计算机组成原理实验的时候，扩展CPLD会充当串口控制器的角色，完成串行数据传输的功能，它会被配置成一个UART(Universal Asynchronous Receiver Transmitter)即通用异步收发器，是被广泛使用的串行数据传输装置。UART主要由控制逻辑、发送器、接收器和波特率发生器组成，通过VHDL语言实现这些模块，将生成的bit文件下载到CPLD，从而得到功能完备的UART，其原理图如图2.12。

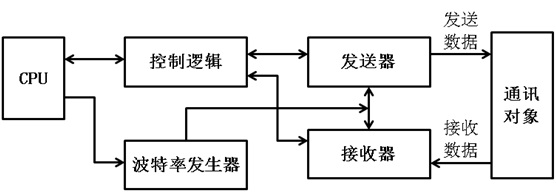


图2.12 UART原理图

UART的帧格式（如图2.13所示）：包括1位起始位、5-8位数据位、校验位、停止位和空闲位，其中数据位、校验位和停止位可根据数据传输具体情况来设定，本文涉及到数据格式为8位数据位、1位校验位和1位停止位，波特率为9600bit/s。

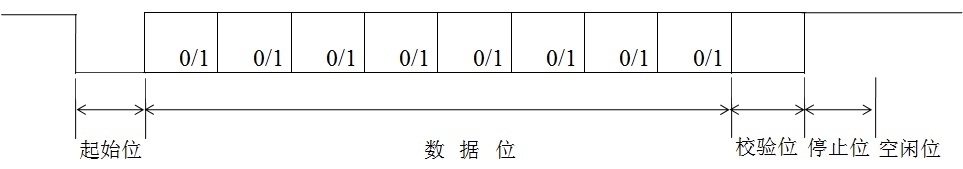


图2.13 UART的帧格式

UART的设计中包括的信号如表2.5，其中分频器是考虑到实验平台配置的时钟频率过高而进行的分频设计。分频器的工作时序是：clk接收实验平台的时钟信号，通过分频后，由clk0输出给接收器和发送器使用；接收器的工作时序是：由rxd接收来的数据通过对起始位的捕捉开始接收数据，并移位到寄存器中，接收完成之后输出到dout，并由data\_ready给出标志信号；发送器的工作时序是：CPU通过对tsre和tbre两个标志信号判断决定是否可以发送数据到din,如果可以则通过wrn信号控制din经过发送缓存器和移位寄存器将并行数据转为串行数据发送到sdo,同时通过tsre和tbre两个标志信号告知CPU可以继续发送数据。

表2.5 UART信号表

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **分频器信号** | | **接收器端口信号** | | **发送器端口信号** | |
| rst | 复位信号 | rst | 复位信号 | rst | 复位信号 |
| clk | 输入时钟 | Clk16 | 输入时钟 | clk16 | 输入时钟 |
| clk0 | 输出时钟 | rxd | 串行数据输入 | wrn | 写锁存信号 |
|  |  | rdn | 读锁存信号 | din[7downto0] | 数据输入 |
|  |  | dout[7downto0] | 数据输出 | tsre | 发送器标志 |
|  |  | data\_ready | 数据接收标志 | tbre | 接收缓存器标志 |
|  |  | framing\_error | 帧错误信号 | sdo | 串行数据输出 |
|  |  | parity\_error | 校验错误信号 |  |  |

由于串口的传输速度有限，而且考虑到CPLD在内存有限的情况下没有加入缓存，这就要求CPU和CPLD实现的UART模块能够在频率方面很好的匹配，可是在实际设计过程中会对CPU的频率有不同的需求，所以对UART进行了进一步的改进，修改之后data\_ready只会被一个输入数据置‘1’一次，CPU修改wrn后，cpld可以立刻响应并且将tbre置‘0’，使得data\_ready=‘1’时CPU可以读数据，只要将rdn置‘0’再置‘1’即可（同时数据线写‘Z’）；当tbre=‘1’ and tsre=‘1’时CPU可以写数据，只要将wrn置‘0’再置’1’即可，wrn可以立刻响应。UART设计的管脚设置见表2.6。

表2.6 UART管脚分配

|  |  |  |  |
| --- | --- | --- | --- |
| **信号** | **属性** | **管脚** | **备注** |
| Clk | in | P1 | 时钟输入信号 |
| data<0> | inout | P6 | 数据位 |
| data<1> | inout | P7 |
| data<2> | inout | P8 |
| data<3> | inout | P9 |
| data<4> | inout | P10 |
| data<5> | inout | P11 |
| data<6> | inout | P12 |
| data<7> | inout | P13 |
| data\_ready | out | P50 | 数据接收完毕，InterConn0 |
| framing\_error | out | P49 | 帧错误信号，InterConn1 |
| parity\_error | out | P46 | 校验错误信号，InterConn2 |
| Rdn | in | P43 | 读锁存信号，InterConn3 |
| Rst | in | P4 | 复位信号 |
| Rxd | in | P58 | 接收串行输入数据，RXD1 |
| Sdo | out | P54 | 串行数据输出，TXD1 |
| Tbre | out | P42 | 接收缓存器标志，InterConn4 |
| Tsre | out | P41 | 发送器标志，InterConn5 |
| Wrn | in | P40 | 写锁存，InterConn6 |

在实际实验过程中，实验资料中会提供CPLD中的UART。

## 2.4 内存

THINPAD教学计算机的实验板上提供了2种存储器，一种是静态存储器SRAM，其内容是掉电不保存的，另外一种是Flash存储器，可以永久保存数据的，二者在实验板上的作用不同。

### 2.4.1 SRAM

THINPAD教学计算机的实验板使用了2片SRAM（Static Random Access Memory）作为主要存储器，RAM1为基本内存，RAM2为扩展内存，二者具有独立的数据总线和地址总线，可以用来分别存储实验测试程序以及数据，也可以统一编址存储测试程序和数据，其中RAM1与CPLD共用基本数据总线和地址总线。

SRAM是一种具有静止存取功能的内存，不需要刷新电路即能保存它内部存储的数据，其特点是访问速度快，访问操作、管理方式简单方便，不需要额外的内存刷新电路。SRAM芯片采用的是ISSI公司生产的异步高速CMOS SRAM，型号为IS61LV25616-10TI，电压需求为3.3V，具有低耗、高性能的特点。此芯片配置16位的数据线（D），17位的地址线（A），以及内存使能信号（EN）、内存读写信号（WE）、内存输出使能信号（OE），其读写时序如图4.2。

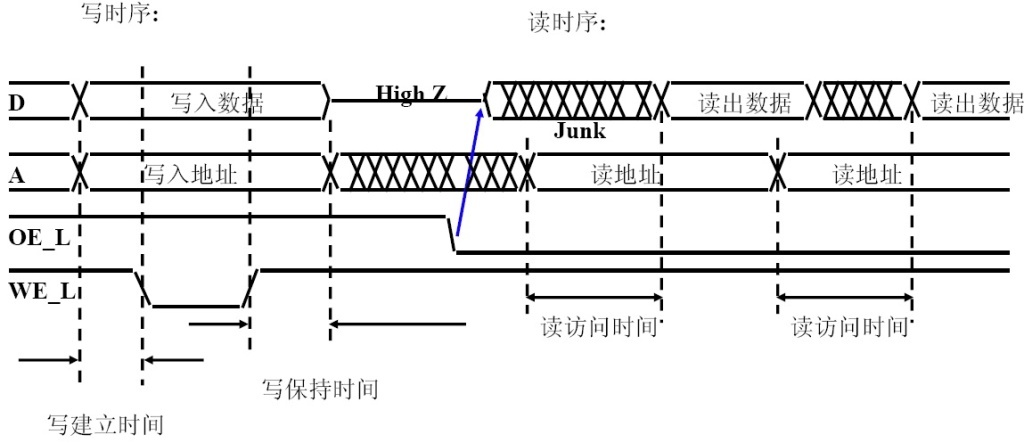


图2.14 RAM读写时序

### 2.4.2 Flash

THINPAD教学计算机的实验板还使用了Flash存储器存储实验系统数据，全名叫Flash EEPROM Memory。Flash存储器又称为闪存，它兼具了ROM和RAM的长处，不仅可以擦出及修改里面的内容，还可以快速读取，同时它还具有数据不会因为断电而丢失的特点，它还有功耗低、集成度高等特点，因此被广泛使用与各种领域。

实验平台使用的Flash存储器是一片MT28F640J3，其数据线为16位，地址线为23位，可寻址空间为8MB。

在利用实验平台进行CPU设计实验时，由于RAM存储器的内容断电后不能保存，因此需要一个非易失的存储器来保存启动程序、测试程序以及数据等，当实验装置断电时Flash存储器中的内容并不会丢失，当实验装置上电时，CPU需要将实验程序数据等从Flash存储器中读出并写入RAM存储器中，并开始执行代码。

Flash是按照块进行存储访问的，其访问操作比较复杂，需要通过特定的步骤来修改里面的数据，具体的Flash操作方法请见后面的实验内容。

## 2.5 总线

总线（Bus）是计算机各种功能部件之间传送信息的公共通信干线，它是由信号线组成的传输线束，按照计算机所传输的信息种类，计算机的总线可以划分为数据总线（DataBus）、地址总线(AddrBus)和控制总线(ControlBus)，分别用来传输数据、数据地址和控制信号。当总线上有多个器件时，如果当前总线空闲（其他器件都以高阻态形式连接在总线上）且一个器件要与目的器件通信时，发起通信的器件驱动总线，发出地址和数据。其他以高阻态形式连接在总线上的器件如果收到（或能够收到）与自己相符的地址信息后，即接收总线上的数据。发送器件完成通信，将总线让出（输出变为高阻态）。

THINPAD教学计算机上共有三条总线，分别是基本总线、扩展总线和Flash总线，它们都分别有着自身的数据、地址和控制线。其中扩展总线只连接了实验FPGA和扩展内存RAM2，Flash总线也只连接了Flash存储器和实验FPGA，这两条总线是专门供实验FPGA访问扩展内存和Flash存储器使用的。而基本总线上就连接了多个器件，包括实验FPGA、基本内存RAM1、扩展CPLD和指示灯。这几个器件之中，实验FPGA来负责总线的访问，当FPGA要访问基本内存或者扩展CPLD时就需要让另外一个器件保持高阻态输出，具体操作可以使用基本内存的OE信号，对于扩展CPLD则可以通过配置成根据地址访问等方式来使得器件不占用总线。当然指示灯没有数据向总线的输入，因此可以不需要控制，它可以用来观察当前总线的内容。基本总线结构如图2.15所示。



图2.15 基本总线

虽然手拨开关没有直接连接到基本总线上，而是直接连接到了实验FPGA上，但是考虑到实验的完整及方便，在整体结构上也将手拨开关算作是连接到基本总线上，只不过是由FPGA内部来控制数据向基本数据总线的发送。如果需要直接设置内存总线上的内容，就可以直接通过手拨开关拨入FPGA，然后由FPGA发送到数据总线上。

## 2.6 外部接口

为了方便进行计算机组成原理和数字逻辑实验，THINPAD教学计算机上提供了多种常用接口，如七段数码管、开关、VGA、串口、PS2、LCD点阵等。他们大多直接连接到实验FPGA上，具体的管脚分配请参考表2.3。

七段数码管、发光二极管LEDS都是直接接到实验FPGA上，FPGA可以直接驱动他们，完成实验，或者协助调试。

开关包括手拨开关、微动开关和复位开关，手拨开关可以作为信号输入，也可以作为总线输入；微动开关可以提供脉冲，不过需要在FPGA中加入去抖动电路；复位开关包括手动时钟和复位开关两个，这两个信号已经加入了去抖动电路，是实验调试过程中不可或缺的两个信号。

实验板上的串口包括两个普通串口和一个USB转串口电路。两个普通串口是直接连接在扩展CPLD上（RXD1/TXD1、RXD2/TXD2），使用CPLD作为控制器，实验FPGA通过访问扩展CPLD来操作串口。还有一个USB转串口电路(U\_RXD/U\_TXD)，这个串口是完全由实验FPGA来完成的，不过为了方便，在实验板上加入了一个USB转串口的电路，他可以直接将串口转换成USB口，使用PC上的USB口就能与实验板进行串口通讯了，当然需要在PC端安装驱动程序。

为了配合完成计算机组成原理实验，实验板上专门加入了PS2和VGA接口，实验者可以使用它们作为教学计算机的输入输出，使得教学计算机独立成为一个完成的计算机系统，这里有一点需要注意的是，为了匹配电平，PS2接口被连接到扩展CPLD上，实验者将CPLD仅仅作为一个通路将PS2接口连接到实验FPGA上，可以以在扩展CPLD上设计一个PS2控制器。VGA接口直接连接在实验FPGA上，使用的时候可以将FPGA的片内Ram作为显存，也可以使用片外的SRAM作为显存。

在实验板上还设置了一块128\*64的单色点阵LCD，也是直接连接到实验FPGA上。它是带驱动的液晶，是由两块单独的液晶屏组成，每块为64\*64 单色点阵，通过片选来控制选通，这样就组成了一个128\*64的单色液晶屏。每块液晶屏的控制部分都内置了64×64=4096位显示RAM，RAM中每位数据对应LCD屏上一个点的亮、暗状态，通过改写RAM 中的内容就可以修改LCD上的显示内容。具体信号说明与操作请参考相应器件说明。

# 第3章 THINPAD教学计算机软件平台

## 3.1 概述

THINPAD教学计算机系统配备有必要的软件系统，包括有：

1. 监控程序。运行在教学机硬件平台上，作为用户使用教学计算机的界面和工具，相当于商业计算机的操作系统。可接收用户输入的命令并完成相应的功能，如运行用户程序、将用户输入的汇编语句翻译成机器指令并保存在教学机存储器中、反汇编机器指令为汇编语句、单步执行机器指令等。
2. 模拟器。运行在PC机的Windows操作系统下，主要功能是在PC机上模拟教学计算机指令系统编写的汇编程序的执行情况。在教学实验中，可用来熟悉教学机指令系统、作为汇编语言程序的实验平台等，也可在硬件调试过程中，作为参照系统使用。
3. 汇编器。虽然监控程序中可完成一些汇编语句的汇编工作，但监控程序功能还是比较薄弱，比较适合一些功能简单、需要的语句少的汇编程序的编辑和汇编工作，对于一些功能复杂，规模比较大的汇编程序，还是希望能通过PC机上进行编辑，然后直接汇编成教学机机器语言程序后，再放置到教学机硬件上调试运行。汇编器就是用来完成此项工作。
4. 编译器。尽管教学试验平台主要是为“计算机组成原理”课程设计的，但考虑到教学机本身系统的完备性，尽量给同学们提供一台更为真实的计算机系统，我们也为教学机配备了高级语言编译器。它所使用的高级语言是一种C语言的简化版本，编译器能将该语言的高级语言程序编译成THCO MIPS指令系统的机器语言程序，然后在教学计算机硬件上运行。
5. 终端程序。终端程序是我们为教学计算机专门设计的一个软件，运行于通过串口和教学机连接的PC计算机上，这样，PC机可作为教学机的输入／输出设备。当然，由于教学计算机监控程序实现的功能还比较薄弱，终端程序还需要辅助监控程序完成一些功能。
6. 数据通讯程序。教学计算机的许多实验，需要在其存储器中预先加载程序或者数据，数据通讯程序是为这个目的而设计的。它可完成PC机中文件和教学机存储器之间的数据交换。

以上这些系统软件构成了基本完备的实验计算机软件体系，实现了如图1.1所示的完整的6层次计算机结构。

## 3.2指令系统

指令系统是计算机系统的灵魂和核心，它定义了计算机软件和硬件的接口。设计一个计算机系统的首要任务就是设计指令系统。完成指令系统设计后，硬件系统设计师可根据指令系统确定的功能和指令格式，完成指令系统的硬件实现。而软件设计师也可根据指令集，实现相关的系统软件，如操作系统、编译系统等，并支持应用软件的开发。

计算机指令系统应满足完备性、规整性、高效性和兼容性要求，也就是说，指令功能应该完备，指令格式应尽量规整，指令执行应该比较高效，并与以前的同系列计算机兼容。

教学计算机指令系统主要应满足教学的需要，对指令系统的完备性、兼容性方面要求不是很高。从教学角度考虑，指令系统应基本完备，满足运行简单监控程序的需要；指令格式和寻址方式尽量简单，可在硬件上简洁实现。根据这些特点和要求，我们借用MIPS16e指令系统中的部分指令的格式和功能，进行必要的调整后，形成了THINPAD教学机指令系统。

该指令系统由43条指令构成，能满足一般程序实现的需要。指令格式和寻址方式简单，保留了MIPS指令系统的风格，容易在硬件上实现。由于MIPS指令系统的特点，该指令系统实现指令流水十分方便，能很好地体现教学内容，满足教学要求。

### 3.2.1 教学计算机指令格式

教学机指令系统全部指令字长均为16位，其中操作码字段固定为指令的高5位。对于一些相同类型的指令，另外安排一些辅助位区分不同的操作。按照指令格式可将指令划分成寄存器（R）型、立即数（I）型、转移（B）型和长跳转（J）型4种类型。同样由于字长较短，不同指令中立即数所占用的位数不相同，这是需要特别注意的一点。图3.1列出了教学机指令系统的指令格式。

R型：

I型：

B型：

J型：

图3.1 教学计算机指令格式

属于R型指令格式的指令有：SLL、SRL、SRA、SLLV、SRLV、SRAV、MTSP、MOVE、ADDU、SUBU、MFPC、SLT、SLTU、CMP、NEG、AND、OR、XOR、NOT、MFIH、MTIH 。

属于I型指令格式的指令有：SW\_RS、SW\_SP、SW、LW\_SP、LW、ADDIU3、ADDSP3、ADDSP、ADDIU、LI、SLTI、SLTUI、CMPI、INT 。

属于B型指令格式的指令有：B、BEQZ、BNEZ、BTEQZ、BTNEZ 。

属于J型指令格式的指令有：JR、JRRA、JALR 。

教学机指令系统寻址方式可分为寄存器寻址、立即数寻址以及基址加偏移量的访存寻址3种方式。

### 3.2.2 教学计算机指令及功能

教学计算机指令系统共有43条指令。下面按指令操作码的字母顺序给出每条指令的格式和功能说明。表中使用的主要符号含义如下：

表 3.1 指令功能描述中符号与含义对照表

|  |  |
| --- | --- |
| ***含***符号 | ***含***含义 |
| **rx**  **ry**  **rz**  **R[X]**  **MEM[X]**  **PC**  **SP**  **IH**  **RA**  **T**  **X = Y**  **X < Y**  **X >= Y**  **X←Y**  **X + Y**  **X - Y**  **X & Y**  **X | Y**  **X Y**  **~X**  **X<<Y**  **X>>Y**  **immediate**  **Zero\_extend(X)**  **Sign\_extend(X)**  **(logical)**  **(arithmetic)**  **if () then …** | 通用寄存器，编号为x  通用寄存器，编号为y  通用寄存器，编号为z  编号为X的通用寄存器  内存中地址为X的字(X为字节地址)  PC(程序计数器)值  SP(栈顶指针寄存期)值  IH(中断寄存器)值  RA(返回值寄存器)值  T标志寄存器的值  X等于Y  X小于Y  X大于或等于Y  将Y的值赋给X  结果为X与Y相加后的值  结果为X与Y相减后的值  结果为X与Y作按位与后的值  结果为X与Y作按位或后的值  结果为X与Y作按位异或后的值  结果为X取非后的值  结果为X左移Y位后的值  结果为X右移Y位后的值  立即数  结果为X零扩展为16位数后的值  结果为X符号扩展为16位数后的值  逻辑移位  算数移位  如果满足括号内的条件，则执行then后的操作 |

1. ADDIU

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 指令编码 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 0 | 1 | 0 | 0 | 1 | rx | | | Immediate | | | | | | | |
| 指令格式 | ADDIU rx immediate | | | | | | | | | | | | | | | |
| 指令功能 | R[x]←R[x]+Sign\_extend(immediate) | | | | | | | | | | | | | | | |
| 功能说明 | 对立即数immediate进行符号扩展后与寄存器rx的值求和，结果保存到寄存器rx中。 | | | | | | | | | | | | | | | |

1. ADDIU3

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 指令编码 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 0 | 1 | 0 | 0 | 0 | rx | | | ry | | | 0 | Immediate | | | |
| 指令格式 | ADDIU3 rx ry immediate | | | | | | | | | | | | | | | |
| 指令功能 | R[y]←R[x]+ Sign\_extend(immediate) | | | | | | | | | | | | | | | |
| 功能说明 | 对立即数immediate进行符号扩展后与寄存器rx的值求和，结果保存到寄存器ry中。 | | | | | | | | | | | | | | | |

1. ADDSP3

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 指令编码 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 0 | 0 | 0 | 0 | 0 | rx | | | Immediate | | | | | | | |
| 指令格式 | ADDSP3 rx immediate | | | | | | | | | | | | | | | |
| 指令功能 | R[x]←SP+Sign\_extend(immediate) | | | | | | | | | | | | | | | |
| 功能说明 | 对立即数immediate进行符号扩展后与寄存器SP的值求和，结果保存到寄存器rx中。 | | | | | | | | | | | | | | | |

1. ADDSP

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 指令编码 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | immediate | | | | | | | |
| 指令格式 | ADDSP immediate | | | | | | | | | | | | | | | |
| 指令功能 | SP←SP+Sign\_extend(immediate) | | | | | | | | | | | | | | | |
| 功能说明 | 对立即数immediate进行符号扩展后与寄存器SP的值求和，结果保存到寄存器SP中。 | | | | | | | | | | | | | | | |

1. ADDU

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 指令编码 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1 | 1 | 1 | 0 | 0 | rx | | | ry | | | rz | | | 0 | 1 |
| 指令格式 | ADDU rx ry rz | | | | | | | | | | | | | | | |
| 指令功能 | R[z]←R[x]+R[y] | | | | | | | | | | | | | | | |
| 功能说明 | 将寄存器rx与寄存器ry的值求和，结果保存到寄存器rz中。 | | | | | | | | | | | | | | | |

1. AND

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 指令编码 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1 | 1 | 1 | 0 | 1 | rx | | | ry | | | 0 | 1 | 1 | 0 | 0 |
| 指令格式 | AND rx ry | | | | | | | | | | | | | | | |
| 指令功能 | R[x]←R[x] & R[y] | | | | | | | | | | | | | | | |
| 功能说明 | 对寄存器rx与寄存器ry的值逐位进行逻辑与，结果保存到寄存器rx中。 | | | | | | | | | | | | | | | |

1. B

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 指令编码 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 0 | 0 | 0 | 1 | 0 | Immediate | | | | | | | | | | |
| 指令格式 | B immediate | | | | | | | | | | | | | | | |
| 指令功能 | PC←PC+Sign\_extend(immediate) | | | | | | | | | | | | | | | |
| 功能说明 | 将寄存器PC与立即数immediate符号扩展后的值求和，结果保存到寄存器PC中，即程序无条件跳转到目的地址执行。 | | | | | | | | | | | | | | | |

1. BEQZ

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 指令编码 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 0 | 0 | 1 | 0 | 0 | rx | | | immediate | | | | | | | |
| 指令格式 | BEQZ rx immediate | | | | | | | | | | | | | | | |
| 指令功能 | if (R[x] = 0) then PC←PC+Sign\_extend(immediate) | | | | | | | | | | | | | | | |
| 功能说明 | 若寄存器rx的值为0，则跳转到目的地址执行；否则顺序执行下一条指令。 | | | | | | | | | | | | | | | |

1. BNEZ

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 指令编码 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 0 | 0 | 1 | 0 | 1 | rx | | | immediate | | | | | | | |
| 指令格式 | BNEZ rx immediate | | | | | | | | | | | | | | | |
| 指令功能 | if (R[x] != 0) then PC←PC+Sign\_extend(immediate) | | | | | | | | | | | | | | | |
| 功能说明 | 若寄存器rx的值不为0，则跳转到目的地址执行；否则顺序执行下一条指令。 | | | | | | | | | | | | | | | |

1. BTEQZ

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 指令编码 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | immediate | | | | | | | |
| 指令格式 | BTEQZ immediate | | | | | | | | | | | | | | | |
| 指令功能 | if (T = 0) then PC←PC+Sign\_extend(immediate) | | | | | | | | | | | | | | | |
| 功能说明 | 若标志寄存器T的值为0，则跳转到目的地址执行；否则顺序执行下一条指令。 | | | | | | | | | | | | | | | |

1. BTNEZ

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 指令编码 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | immediate | | | | | | | |
| 指令格式 | BTNEZ immediate | | | | | | | | | | | | | | | |
| 指令功能 | if (T != 0) then PC←PC+Sign\_extend(immediate) | | | | | | | | | | | | | | | |
| 功能说明 | 若标志寄存器T的值不为0，则跳转到目的地址执行；否则顺序执行下一条指令。 | | | | | | | | | | | | | | | |

1. CMP

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 指令编码 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1 | 1 | 1 | 0 | 1 | rx | | | ry | | | 0 | 1 | 0 | 1 | 0 |
| 指令格式 | CMP rx ry | | | | | | | | | | | | | | | |
| 指令功能 | if (R[x] = R[y]) then T←0  if (R[x] != R[y]) then T←1 | | | | | | | | | | | | | | | |
| 功能说明 | 比较rx和ry的值，若相等，则给标志寄存器T置0；否则置1。 | | | | | | | | | | | | | | | |

1. CMPI

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 指令编码 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 0 | 1 | 1 | 1 | 0 | rx | | | Immediate | | | | | | | |
| 指令格式 | CMPI rx immediate | | | | | | | | | | | | | | | |
| 指令功能 | if (R[x] = Sign\_extend (immediate)) then T←0  if (R[x] != Sign\_extend (immediate)) then T←1 | | | | | | | | | | | | | | | |
| 功能说明 | 将立即数immediate进行符号扩展后，与rx的值比较。若相等，则给标志寄存器T置0；否则置1。 | | | | | | | | | | | | | | | |

1. INT

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 指令编码 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | immediate | | | |
| 指令格式 | INT immediate | | | | | | | | | | | | | | | |
| 指令功能 | 软件中断，immediate（unsigned）为中断号 | | | | | | | | | | | | | | | |

1. JALR

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 指令编码 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1 | 1 | 1 | 0 | 1 | Rx | | | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 指令格式 | JALR rx | | | | | | | | | | | | | | | |
| 指令功能 | PC←R[x]，RA←RPC | | | | | | | | | | | | | | | |
| 功能说明 | 与JRRA配合用于子程序调用，RPC是子程序返回后应该执行的指令地址，即延时槽后一条指令的地址 | | | | | | | | | | | | | | | |

1. JR

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 指令编码 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1 | 1 | 1 | 0 | 1 | rx | | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 指令格式 | JR rx | | | | | | | | | | | | | | | |
| 指令功能 | PC←R[x] | | | | | | | | | | | | | | | |
| 功能说明 | 程序无条件跳转到rx地址单元执行，用于长地址跳转 | | | | | | | | | | | | | | | |

1. JRRA

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 指令编码 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 指令格式 | JRRA | | | | | | | | | | | | | | | |
| 指令功能 | PC←RA | | | | | | | | | | | | | | | |
| 功能说明 | 与JALR指令配合，完成从子程序返回功能 | | | | | | | | | | | | | | | |

1. LI

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 指令编码 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 | rx | | | immediate | | | | | | | |
| 指令格式 | LI rx immediate | | | | | | | | | | | | | | | |
| 指令功能 | R[x]←Zero\_Extend (immediate) | | | | | | | | | | | | | | | |
| 功能说明 | 将立即数immediate的值（进行零扩展后）存储到寄存器rx中。 | | | | | | | | | | | | | | | |

1. LW

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 指令编码 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1 | 0 | 0 | 1 | 1 | rx | | | ry | | | Immediate | | | | |
| 指令格式 | LW rx ry immediate | | | | | | | | | | | | | | | |
| 指令功能 | R[y]←MEM[R[x] + Sign\_extend (immediate)] | | | | | | | | | | | | | | | |
| 功能说明 | 从内存中读取数据到寄存器ry中，内存地址为寄存器rx的内容与立即数（进行符号扩展后）immediate之和。 | | | | | | | | | | | | | | | |

1. LW\_SP

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 指令编码 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 | Rx | | | immediate | | | | | | | |
| 指令格式 | LW\_SP rx immediate | | | | | | | | | | | | | | | |
| 指令功能 | R[x]←MEM[SP + Sign\_extend (immediate)] | | | | | | | | | | | | | | | |
| 功能说明 | 从内存中读取数据到寄存器rx中，内存地址为寄存器SP的内容与立即数（进行符号扩展后）immediate之和。 | | | | | | | | | | | | | | | |

1. MFIH

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 指令编码 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1 | 1 | 1 | 1 | 0 | Rx | | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 指令格式 | MFIH rx | | | | | | | | | | | | | | | |
| 指令功能 | R[x]←IH | | | | | | | | | | | | | | | |
| 功能说明 | 将寄存器IH的值赋给寄存器rx。 | | | | | | | | | | | | | | | |

1. MFPC

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 指令编码 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1 | 1 | 1 | 0 | 1 | Rx | | | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 指令格式 | MFPC rx | | | | | | | | | | | | | | | |
| 指令功能 | R[x]←PC | | | | | | | | | | | | | | | |
| 功能说明 | 将寄存器PC的值赋给寄存器rx。 | | | | | | | | | | | | | | | |

1. MOVE

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 指令编码 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 0 | 1 | 1 | 1 | 1 | Rx | | | Ry | | | 0 | 0 | 0 | 0 | 0 |
| 指令格式 | MOVE rx ry | | | | | | | | | | | | | | | |
| 指令功能 | R[x]←R[y] | | | | | | | | | | | | | | | |
| 功能说明 | 将寄存器ry的值赋给寄存器rx。 | | | | | | | | | | | | | | | |

1. MTIH

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 指令编码 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1 | 1 | 1 | 1 | 0 | Rx | | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 指令格式 | MTIH rx | | | | | | | | | | | | | | | |
| 指令功能 | IH←R[x] | | | | | | | | | | | | | | | |
| 功能说明 | 将寄存器rx的值赋给寄存器IH。 | | | | | | | | | | | | | | | |

1. MTSP

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 指令编码 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | Rx | | | 0 | 0 | 0 | 0 | 0 |
| 指令格式 | MTSP rx | | | | | | | | | | | | | | | |
| 指令功能 | SP←R[x] | | | | | | | | | | | | | | | |
| 功能说明 | 将寄存器rx的值赋给寄存器SP。 | | | | | | | | | | | | | | | |

1. NEG

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 指令编码 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1 | 1 | 1 | 0 | 1 | Rx | | | Ry | | | 0 | 1 | 0 | 1 | 1 |
| 指令格式 | NEG rx ry | | | | | | | | | | | | | | | |
| 指令功能 | R[x]←0 – R[y] | | | | | | | | | | | | | | | |
| 功能说明 | 将寄存器ry的值求相反数后赋给寄存器rx。 | | | | | | | | | | | | | | | |

1. NOT

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 指令编码 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1 | 1 | 1 | 0 | 1 | rx | | | Ry | | | 0 | 1 | 1 | 1 | 1 |
| 指令格式 | NOT rx ry | | | | | | | | | | | | | | | |
| 指令功能 | R[x]←~ R[y] | | | | | | | | | | | | | | | |
| 功能说明 | 将寄存器ry的值求逻辑反后赋给寄存器rx。 | | | | | | | | | | | | | | | |

1. NOP

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 指令编码 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | | 8 | | 7 | 6 | 5 | | 4 | 3 | 2 | 1 | 0 |
| 0 | 0 | 0 | 0 | 1 | 0 | | 0 | | 0 | 0 | 0 | | 0 | 0 | 0 | 0 | 0 | 0 |
| 指令格式 | NOP | | | | | | | | | | | | | | | | | | |
| 指令功能 | 空操作 | | | | | | | | | | | | | | | | | | |
| 功能说明 | 什么也不做。 | | | | | | | | | | | | | | | | | | |

1. OR

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 指令编码 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1 | 1 | 1 | 0 | 1 | rx | | | Ry | | | 0 | 1 | 1 | 0 | 1 |
| 指令格式 | OR rx ry | | | | | | | | | | | | | | | |
| 指令功能 | R[x]←R[x] | R[y] | | | | | | | | | | | | | | | |
| 功能说明 | 将寄存器rx和寄存器ry的值求逻辑或后赋给寄存器rx。 | | | | | | | | | | | | | | | |

1. SLL

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 指令编码 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 0 | 0 | 1 | 1 | 0 | rx | | | Ry | | | immediate | | | 0 | 0 |
| 指令格式 | SLL rx ry immediate | | | | | | | | | | | | | | | |
| 指令功能 | if (immediate = 0) then R[x]←R[y]<<8  If (immediate != 0) then R[x]←R[y]<<immediate(unsigned) | | | | | | | | | | | | | | | |
| 功能说明 | 对寄存器ry的值逻辑左移immediate位后赋值给寄存器rx。（注意：immediate为无符号数，当其为0时，左移8位） | | | | | | | | | | | | | | | |

1. SLLV

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 指令编码 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1 | 1 | 1 | 0 | 1 | rx | | | Ry | | | 0 | 0 | 1 | 0 | 0 |
| 指令格式 | SLLV rx ry | | | | | | | | | | | | | | | |
| 指令功能 | R[y]←R[y]<<R[x] | | | | | | | | | | | | | | | |
| 功能说明 | 对寄存器ry的值逻辑左移R[x]位后赋值给寄存器ry。 | | | | | | | | | | | | | | | |

1. SLT

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 指令编码 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1 | 1 | 1 | 0 | 1 | rx | | | Ry | | | 0 | 0 | 0 | 1 | 0 |
| 指令格式 | SLT rx ry | | | | | | | | | | | | | | | |
| 指令功能 | if (R[x] < R[y]) then T←1 (有符号比较)  if (R[x] >= R[y]) then T←0 (有符号比较) | | | | | | | | | | | | | | | |
| 功能说明 | 比较寄存器rx和ry的值 并根据结果对标志寄存器T赋值。 | | | | | | | | | | | | | | | |

1. SLTI

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 指令编码 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 | rx | | | Immediate | | | | | | | |
| 指令格式 | SLTI rx immediate | | | | | | | | | | | | | | | |
| 指令功能 | if (R[x] < Sign\_extend(immediate)) then T←1 (有符号比较)  if (R[x] >= Sign\_extend(immediate)) then T←0 (有符号比较) | | | | | | | | | | | | | | | |
| 功能说明 | 比较寄存器rx 和立即数immediate 的值并根据结果对标志寄存器T赋值。 | | | | | | | | | | | | | | | |

1. SLTU

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 指令编码 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1 | 1 | 1 | 0 | 1 | rx | | | Ry | | | 0 | 0 | 0 | 1 | 1 |
| 指令格式 | SLTU rx ry | | | | | | | | | | | | | | | |
| 指令功能 | if (R[x] < R[y]) then T←1 (无符号比较)  if (R[x] >= R[y]) then T←0 (无符号比较) | | | | | | | | | | | | | | | |
| 功能说明 | 同SLT指令，但将参与比较的数据作为无符号数。 | | | | | | | | | | | | | | | |

1. SLTUI

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 指令编码 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 0 | 1 | 0 | 1 | 1 | rx | | | immediate | | | | | | | |
| 指令格式 | SLTUI rx immediate | | | | | | | | | | | | | | | |
| 指令功能 | if (R[x] < Zero\_extend(immediate)) then T←1 (无符号比较)  if (R[x] >= Zero\_extend(immediate)) then T←0 (无符号比较) | | | | | | | | | | | | | | | |
| 功能说明 | 同SLTI指令，但将参与比较的数据作为无符号数。 | | | | | | | | | | | | | | | |

1. SRA

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 指令编码 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 0 | 0 | 1 | 1 | 0 | rx | | | Ry | | | immediate | | | 1 | 1 |
| 指令格式 | SRA rx ry immediate | | | | | | | | | | | | | | | |
| 指令功能 | if (immediate = 0) then R[x]←R[y]>>8 (arithmetic)  if (immediate != 0) then R[x]←R[y]>>immediate (arithmetic)  Immediate:unsigned | | | | | | | | | | | | | | | |
| 功能说明 | 对寄存器ry的值算术右移immediate位后赋值给寄存器rx。（注意：immediate为无符号数，当其为0时，右移8位） | | | | | | | | | | | | | | | |

1. SRAV

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 指令编码 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1 | 1 | 1 | 0 | 1 | rx | | | Ry | | | 0 | 0 | 1 | 1 | 1 |
| 指令格式 | SRAV rx ry | | | | | | | | | | | | | | | |
| 指令功能 | R[y]←R[y]>>R[x] (arithmetic) | | | | | | | | | | | | | | | |
| 功能说明 | 对寄存器ry的值算术右移R[x]位后赋值给寄存器ry。 | | | | | | | | | | | | | | | |

1. SRL

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 指令编码 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 0 | 0 | 1 | 1 | 0 | rx | | | Ry | | | immediate | | | 1 | 0 |
| 指令格式 | SRA rx ry immediate | | | | | | | | | | | | | | | |
| 指令功能 | if (immediate = 0) then R[x]←R[y]>>8 (logical)  if (immediate != 0) then R[x]←R[y]>>immediate (logical)  Immediate:unsigned | | | | | | | | | | | | | | | |
| 功能说明 | 对寄存器ry的值逻辑右移immediate位后赋值给寄存器rx。（注意：immediate为无符号数，当其为0时，右移8位） | | | | | | | | | | | | | | | |

1. SRLV

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 指令编码 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1 | 1 | 1 | 0 | 1 | Rx | | | Ry | | | 0 | 0 | 1 | 1 | 0 |
| 指令格式 | SRLV rx ry | | | | | | | | | | | | | | | |
| 指令功能 | R[y]←R[y]>>R[x] (logical) | | | | | | | | | | | | | | | |
| 功能说明 | 对寄存器ry的值逻辑右移R[x]位后赋值给寄存器ry。 | | | | | | | | | | | | | | | |

1. SUBU

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 指令编码 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1 | 1 | 1 | 0 | 0 | rx | | | Ry | | | rz | | | 1 | 1 |
| 指令格式 | SUBU rx ry rz | | | | | | | | | | | | | | | |
| 指令功能 | R[z]←R[x] - R[y] | | | | | | | | | | | | | | | |
| 功能说明 | 用寄存器rx的值减寄存器ry的值，结果保存到寄存器rz中。 | | | | | | | | | | | | | | | |

1. SW

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 指令编码 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1 | 1 | 0 | 1 | 1 | Rx | | | Ry | | | immediate | | | | |
| 指令格式 | SW rx ry immediate | | | | | | | | | | | | | | | |
| 指令功能 | MEM[R[x] + Sign\_extend (immediate)]←R[y] | | | | | | | | | | | | | | | |
| 功能说明 | 将寄存器ry的值写入到内存中，内存地址为寄存器rx的内容与立即数（进行符号扩展后）immediate之和。 | | | | | | | | | | | | | | | |

1. SW\_RS

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 指令编码 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | Immediate | | | | | | | |
| 指令格式 | SW\_RS immediate | | | | | | | | | | | | | | | |
| 指令功能 | MEM[SP + Sign\_extend (immediate)]←RA | | | | | | | | | | | | | | | |
| 功能说明 | 将寄存器RA的值写入到内存中，内存地址为寄存器SP的内容与立即数（进行符号扩展后）immediate之和。 | | | | | | | | | | | | | | | |

1. SW\_SP

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 指令编码 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 | rx | | | immediate | | | | | | | |
| 指令格式 | SW\_SP rx immediate | | | | | | | | | | | | | | | |
| 指令功能 | MEM[SP + Sign\_extend (immediate)]←R[x] | | | | | | | | | | | | | | | |
| 功能说明 | 将寄存器rx的值写入到内存中，内存地址为寄存器SP的内容与立即数（进行符号扩展后）immediate之和。 | | | | | | | | | | | | | | | |

1. XOR

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 指令编码 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1 | 1 | 1 | 0 | 1 | rx | | | ry | | | 0 | 1 | 1 | 1 | 0 |
| 指令格式 | XOR rx ry | | | | | | | | | | | | | | | |
| 指令功能 | R[x]←R[x] R[y] | | | | | | | | | | | | | | | |
| 功能说明 | 将寄存器rx和ry的值进行逻辑异或，结果保存到寄存器rx。 | | | | | | | | | | | | | | | |

## 3.3 监控程序

### 3.3.1 监控程序简介

现代计算机均配备有操作系统，为用户使用计算机提供便利，并帮助管理计算机的资源。教学计算机提供监控程序，可执行简单的用户命令，诸如输入汇编语句并进行汇编、运行汇编程序、显示寄存器的内容等。

监控程序使用3.2节的指令系统编写，在教学机硬件上运行。教学计算机上设置监控程序的主要目的有两个，一是方便同学们理解、掌握指令的功能和基本执行过程，二是作为检测同学们完成的计算机系统硬件设计的一个统一的测试标准。

监控程序用来作为同学们在教学机上编程的界面，支持编辑并汇编（A）、反汇编（U）、连续运行用户程序（C）、查看内存单元内容（D）、查看寄存器值（R）等5条基本指令，还可支持中断。

由于指令系统中没有设计专门的输入/输出指令，而是将输入/输出功能由访存指令LW和SW通过指定的端口地址来实现，因此，教学计算机监控程序将存储空间和输入/输出端口统一编址，形成统一的地址空间，并将其划分为系统程序区、用户程序区、系统数据区及用户数据区4个部分，表3.2 给出了各区地址段及接口地址。

表3.2 监控程序划分的地址空间段

|  |  |  |
| --- | --- | --- |
| 功能区 | 地址段 | 说明 |
| 系统程序区 | 0x0000~0x3FFF | 存放监控程序 |
| 用户程序区 | 0x4000~0x7FFF | 存放用户程序 |
| 系统数据区 | 0x8000~0xBEFF | 监控程序使用的数据区 |
| Com1数据端口/命令端口 | 0xBF00~0xBF01 | 串口的端口 |
| Com2数据端口/命令端口 | 0xBF02~0xBF03 | 第2个串口的端口 |
| 预留给其他接口 | 0xBF04~0xBF0F | 保留 |
| 系统堆栈区 | 0xBF10~0xBFFF | 用于系统堆栈 |
| 用户数据区 | 0xC000~0xFFFF | 用户程序使用的数据区 |

另外，监控程序保留使用R6和R7两个通用寄存器，分别用作调用地址和返回地址，建议用户程序中不使用这两个寄存器。

### 3.3.2 监控程序框架

根据上述要求，监控程序主要由表3.3中的模块组成。

表3.3 监控程序的主要模块

|  |  |
| --- | --- |
| *模块* | *功能* |
| START | 程序初始化 |
| BEGIN | 主程序 |
| TESTR | 测试串口1是否可读 |
| TESTW | 测试串口1是否可写 |
| SHOWREGS | 显示用户程序寄存器的值 |
| SHOWMEM | 显示内存中内容 |
| ASM | 汇编语句并保存 |
| UASM | 反汇编 |
| COMPILE | 执行用户程序 |
| DELINT | 中断处理 |

START模块：本模块主要实现各项初始化操作，为一些主要寄存器及内存单元赋值。主要包括：

中断寄存器IH=0x0007

堆栈地址SP=0xBF10

内存单元BF10~BF05置0，这些单元分别用来保存用户程序中通用寄存器R0~R5的值。

串口COM1和COM2初始化，即往8251数据端口BF01和BF03发送0x4E、 0x37两个字符，使其可与终端进行通信。

向通讯终端发出“OK”的欢迎字样

跳到主程序BEGIN模块

BEGIN模块：本模块循环等待用户命令，接到命令后检测是否为“R”、“D”、“G”、“Ａ”和“Ｕ”５个合法命令之一。如果是，则跳转到相应模块操作，如果不是，则继续等待下一个命令。

监控整体流程如图3.2所示：

Cin

R

G



D



A



U



else

SHOWREGS

SHOWMEM

COMPILE

ASM

UASM

START

开始

BEGIN

图3.2 监控程序流程图

### 3.3.3 监控程序支持的命令

目前监控程序支持５条命令，各命令的功能、输入格式和实现说明如下。

（１）R

**格式**：R

**功能**：查看寄存器R0-R5 的值

**实现**：监控程序主程序收到R命令后，跳到SHOWREGS模块。该模块依次从BF10~BF15中取出R0~R5的值，按照先低位，后高位的顺序发给终端。

（２）D

**格式**：D [addr] [num]

**功能**：查看由地址addr开始的num个内存的值。缺省值为addr=0x4000，num=0x0A。需要注意的是，如果num有值，则addr不可缺省。

**实现**：由SHOWMEM模块实现。按照先低8位，后高8位的顺序，依次接收addr和num。从地址addr开始，向终端发出连续num个地址单元内的值。

（３）A

**格式**：A [addr]

**功能**：修改地址addr处的值。可以是汇编指令，也可以是立即数，数字统一使用十六进制。缺省值为addr=0x4000。编辑完后，如果合法，则自动编辑下一地址，否则，重新编辑。如果输入为空，则返回到主程序。需要注意，最后两条指令必须是JR R7和NOP。

**实现**：由ASM模块完成，只需要先后接收地址和立即数，再将立即数写入相应地址即可。其中，对语句的汇编工作由终端程序“term”完成。

（４）U

**格式**：U [addr] [num]

**功能**：反汇编地址addr开始的num个内存的值。缺省值为addr=0x4000，num=0x0A。需要注意的是，如果num有值，则addr不可缺省。

**实现**：处理模块是UASM。先接收addr和num，再取出addr开始连续num个地址内的值，发给终端程序，进行反汇编处理。

（５）G

**格式**：G [addr]

**功能**：从地址addr处开始连续执行用户程序，缺省值为addr=0x4000。

**实现**：由COMPILE模块完成。首先接收地址addr；取出BF10-BF15中的值给R0-R5，恢复用户程序现场；接着中断寄存器高位赋1，中断开启；跳到addr执行程序，返回地址保存到R7中。执行完后，关中断，保存现场，跳到主程序BEGIN 。具体流程如图３．３所示。

JR R7

int end int



G



关中断

保存R0-R5

用户程序

DELINT

恢复R0-R5

开中断

JR R6

BEGIN

R6=addr

图3.３　监控程序COMPILE模块流程图

### 3.3.4 对中断的支持

中断是计算机系统的重要概念，它一般指CPU在运行过程中，由于接收到外部设备（外中断）或者CPU本身（内中断，也称异常）的请求信号，转而进行服务响应的过程。中断的实现是一个较为复杂的过程，需要硬件和软件共同完成。中断的响应要求CPU暂时停止当前程序的执行，转而执行中断服务程序，因此，需要一个保留中断断点及中断现场的过程。

中断断点包括被中断程序的当前执行指令的地址，也就是PC寄存器，以及当前的程序状态字的内容，这些需要在响应中断时由硬件来进行保存。而中断现场一般指相关通用寄存器的值，硬件并不清楚需要保存哪些通用寄存器，因此，它们一般由软件来保存。因此，中段服务程序首先应该是保存中断现场，再完成中断功能，最后，再返回到原程序的断点继续执行。

除了保存断点外，是否响应中断、中断优先级的判定及最终跳转到中断服务程序的入口也是硬件需要完成的功能。

监控程序是一个软件程序，因此，它对中断的支持就是提供了一个简单的中断服务程序。而且，这个中断服务程序设计的十分简单，只是输出中断请求编号来区分响应的不同的中断请求，它首先保存中断现场（通用寄存器R0~R3），然后输出中断号，返回前再恢复保留的现场。当然，如果同学们希望改写中断服务程序，丰富中断服务程序的功能，可以自行设计自己的中断服务程序并添加到监控程序中。

### 3.3.5 监控程序使用

在教学计算机FPGA芯片完成正确的CPU设计和实现，并将监控程序执行代码装入到内存从地址0开始的单元后，计算机可执行监控程序，在终端上输出命令提示符“〉〉”。这时，用户可输入“R”、“G”、“A”、“D”、“U”中的一个命令，计算机将完成相应的功能。如图3.4所示。

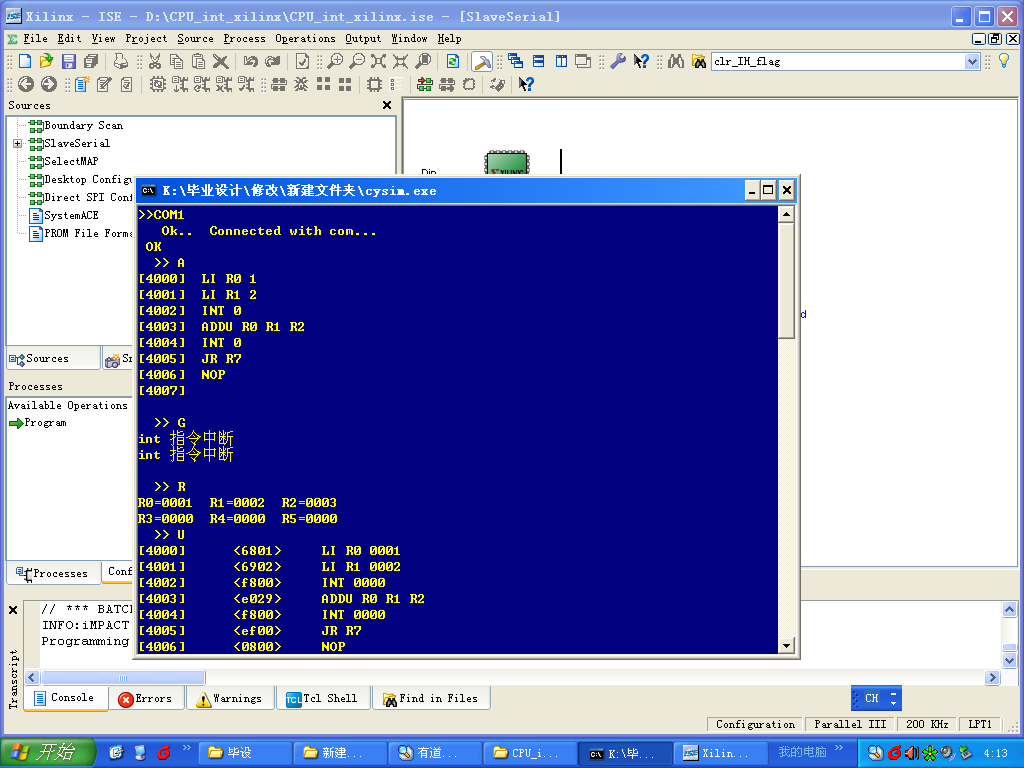


图3.4 监控程序运行界面

## 3.4模拟器

### 3.4.1 模拟器简介

熟悉和理解教学计算机指令系统各指令的功能是进行硬件实现的基础，然而，设计之初我们并没有真正的硬件系统可支持指令系统的运行，因此，用PC计算机，编写一个软件程序来模拟我们将要设计的计算机硬件的功能就十分必要了。教学计算机模拟器就是应此需要而设计的。事实上，计算机模拟的方法在很多研究和应用领域里已经十分常见，许多大型的科学实验，如核爆炸的效果、微观世界的运行规律等需要用计算机模拟来验证，而飞行训练器等模拟训练的方式在实际生活中也屡见不鲜了。

教学计算机模拟器可完成教学计算机指令系统的功能模拟，也就是说，可以用模拟器直接来编辑、汇编和运行3.2节中所有指令编写的汇编语言程序，以方便大家熟悉指令的功能和格式；同时，也可以通过装入教学计算机监控程序，来实际模拟教学计算机硬件的功能，在这种运行模式下，模拟器就成为一台“实际”的THINPAD教学机，方便大家在调试真正的硬件时进行对照。

### 3.4.2 模拟器使用

THINPAD模拟器可运行在两种模式下，一种是模拟指令系统功能，另一种是模拟教学计算机硬件系统。两种模式的适用方式略有不同，分别介绍如下：

1. 指令系统功能模拟

指令系统功能模拟主要通过在模拟器下运行汇编语言程序来实现，以帮助大家了解和熟悉指令系统的功能。为此，模拟器实现了汇编、反汇编、查看和设置寄存器值、单步/连续运行以及对断点的操作等基本功能。这些极大的方便了简单程序的调试工作。

需要指出的是，在模拟器上运行用户编写的汇编程序，采用的是通过模拟器主程序调用用户程序的方式进行的，因此，用户程序结束时，应有一条指令来返回到模拟器主程序。为实现这一功能，系统专门设计了一条RET指令来完成这一功能（THINPAD上监控程序用“G”命令调用用户程序，用户程序通过“JR R7”命令返回，因为调用时将返回地址存放在R7寄存器中）。RET指令是专门为模拟器设计的指令，提示模拟器用户程序到此结束，因此，前节中指令系统中没有这条指令的说明。当然，用户程序中也可以不用这条指令，这时，模拟器会把后续的地址空间中的内容当做指令继续执行，直到出错或者是到了最后的地址才结束。

在Windows操作系统下，运行模拟器程序Simulator，将出现如下界面：

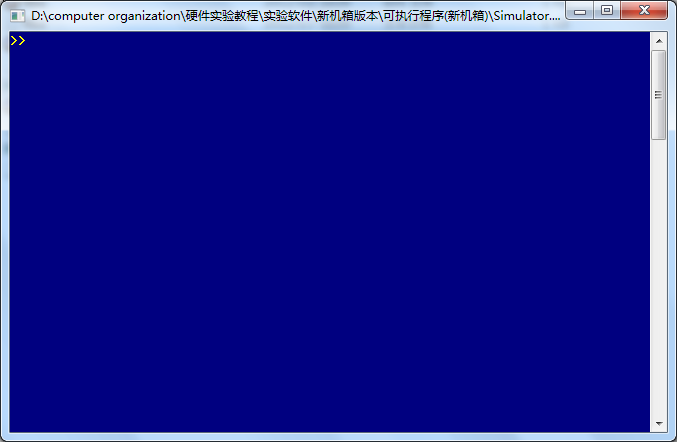


图3.5 教学计算机模拟器

在界面中“〉〉”提示符后面，可以分别输入如下命令，完成相应的功能：

* u命令

命令格式：u［地址］［指令数目］[[1]](#footnote-1)

该命令可以从指定的位置(缺省值为PC位置)开始进行反汇编(缺省为10条)。模拟器支持的内存地址范围是0~0x27FF 和0x4000~0x6003，其它位置的内存单元是不可以访问的也是不可以编辑的，当被用户程序访问时会自动终止程序并报错。

* ea命令

命令格式：ea［地址］

该命令可以从指定的位置(缺省值为PC位置)开始进行汇编。若输入的指令合法，则地址会自动跳转到下一条，否则停留在当前地址上等待输入。输入空串则编辑结束。

* em命令

命令格式：em［地址］

该命令可以从指定的位置(缺省值为PC位置)开始编辑二进制代码。与ea不同的是ea只能进行指令的编辑，而em实现了机器代码或数据段的编辑。在该编辑模式下只能够输入数字0、1。回车键表示当前行编辑结束保存并转到下一行，而输入m表示不保存当前行并退出编辑。

* v命令

命令格式：v［地址/SP］［指令条数］

该命令可以从指定的位置查看内存单元的二进制和十六进制数据(缺省为10条)。与u不同的是v指令查看的是数值而不是反汇编结果。为了方便查看程序堆栈的内容，v指令的第二个参数可以是栈寄存器SP。

* r命令

命令格式：r

该命令可以查看到全部寄存器的值。

* sr命令

命令格式：sr <寄存器名> <数值>[[2]](#footnote-2)

该命令可以修改指定寄存器的数值。其中指令参数<寄存器名>只能够是通用寄存器R0~R7和栈寄存器SP。

* b命令

命令格式：b <地址>

该命令可以向模拟器中添加一个断点。断点的功能比较简单，一旦当前PC的数值出现在断点队列里时连续执行的用户程序会自动中断。

* lb命令

命令格式：lb

该命令可以查看当前全部断点。断点队列里的断点是按照地址从小到大 的顺序排列的。

* db命令

命令格式：db <断点编号>

该命令可以删除断点队列里指定编号的断点。连续使用lb时值得注意的是，删除某一个断点之后其他全部编号较大的断点将会向前移动。

* c命令

命令格式：c［int］

该命令可以从当前PC指向的内存单元开始连续执行用户程序。使用时应当注意的是当PC的值出现在断点队列中的时候，用户程序会自动中断并切换到模拟器模式下。而如果用户希望手动停止当前连续运行的程序并切换到模拟器模式，可以输入Ctrl+G。int参数是可选参数，当用户选择了该参数时，模拟器会打开时钟中断[[3]](#footnote-3)。

* s命令

命令格式：s

该命令可以从当前PC开始进行单步操作。与c命令不同的是，如果当前PC出现在断点队列中，模拟器仍然会执行该条指令即将断点屏蔽掉。因此，如果使用c命令遇到断点，并且用户希望能够继续使用c命令，则应先使用一条s命令然后再使用c命令。

* goto 命令

命令格式：goto <地址>

该命令可以修改当前PC的值。PC指向的是下一条将执行的内存单元。

* reset命令

命令格式：reset

该命令可以进行寄存器的重新初始化。建议重新运行用户程序前使用reset而不是使用goto命令，因为后者不对除PC以外的寄存器进行初始化。(注：PC初始化值是0x0001，因此用户程序的代码段应从0x0001开始，以后不在赘述)

* restart命令

命令格式：restart

该命令可以重新运行用户程序。这条指令集成了reset和c两条指令的功能，使用时该指令先对寄存器进行初始化，然后以连续执行的方式执行用户程序。

* q命令

命令格式：q

使用该命令可以退出模拟器。

例：用模拟器求Fibonacci数的前10个数，逐个保存到地址0X8000开始的内存空间中。

解：首先，启动模拟器程序，用“ea”命令，逐条将汇编语言程序输入，同时完成了汇编，如图3.6所示。

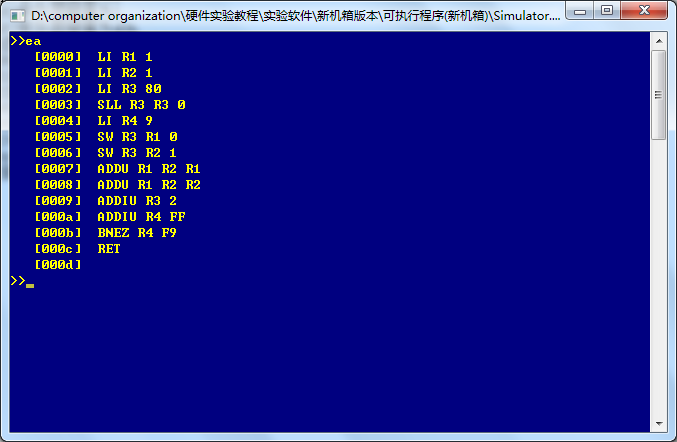


图3.6 编辑命令ea

其次，用“u”命令检查输入的程序是否正确。如有错误，可用“ea [地址]”修改对应的指令。如图3.7所示。

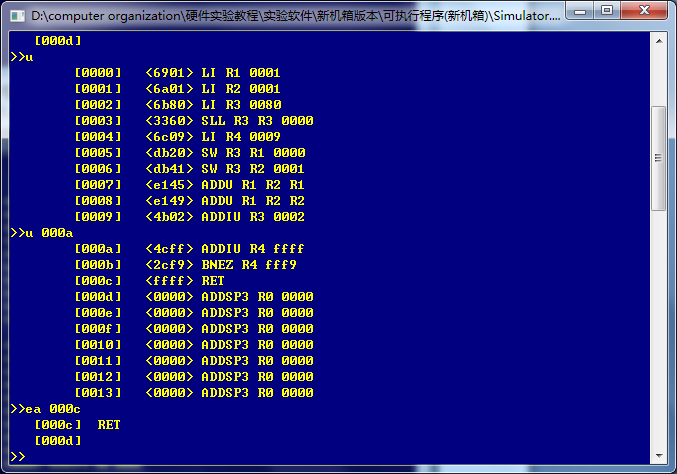


图3.7 反汇编命令u

第三，用“restart”命令运行程序，用“v”命令查看地址8000处的内容，程序运行正确。如图3.8所示。

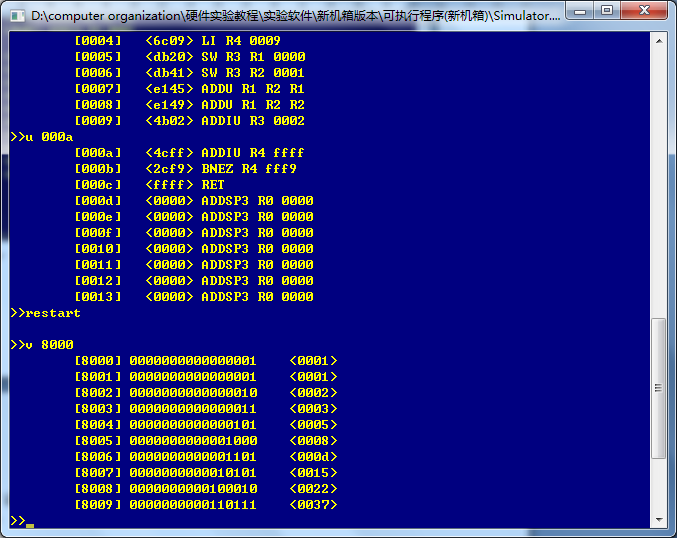


图3.8 运行程序及结果显示

大家也可尝试使用其它命令，体会模拟器的功能。

通过指令功能的模拟，可直接在模拟器下运行汇编语言程序，对大家了解指令系统中各指令的功能和格式是大有益处的。

1. 硬件系统功能模拟

在THINPAD教学计算机硬件上运行程序时，一般是通过教学计算机串口连接PC机，PC机上运行仿真终端程序TERM，利用PC机的键盘和显示器作为输入/输出设备，来进行输入/输出操作。

为了方便硬件调试，模拟器也可运行在这种模式下，由它来仿真THINPAD教学计算机硬件，TERM程序通过Socket来和模拟器通讯，就像和真正的硬件通讯一样。

模拟器主要由以下几个命令来实现上述功能。

* load命令

命令格式：load <文件名>

该命令能够将指定的二进制文件读入到内存中。

* save命令

命令格式：save［文件名］

该命令能够将当前模拟器内存中的全部数据保存到磁盘中。缺省的文件名是rom.dat。

* saveFormat命令

命令格式：saveFormat [文件名]

该命令能够将当前模拟器内存中从0000开始到PC处的指令数据以Intel Hex File Format文件的形式保存到磁盘中，这种文件格式可以被用来在Modelsim仿真软件中模拟内存指令使用。使用该命令时请注意PC的值，以免造成错误。

其中，load命令装入的机器语言程序，可以由汇编器对汇编语言程序生成，这样，还可以避免用模拟器输入长程序时，由于其编辑功能较弱容易造成输入错误的不足。Save和 saveformat命令实现了将内存中的数据或程序卸出的功能，可方便进行调试。

当然，THINPAD教学计算机提供了监控程序，模拟器也完全可以模拟监控程序的运行。为此，专门设计了一条指令server。需要提醒的是，运行server指令前，请确保监控程序的机器码kernel.bin已经存放在当前目录下。图3.9 是server命令启动后的界面。

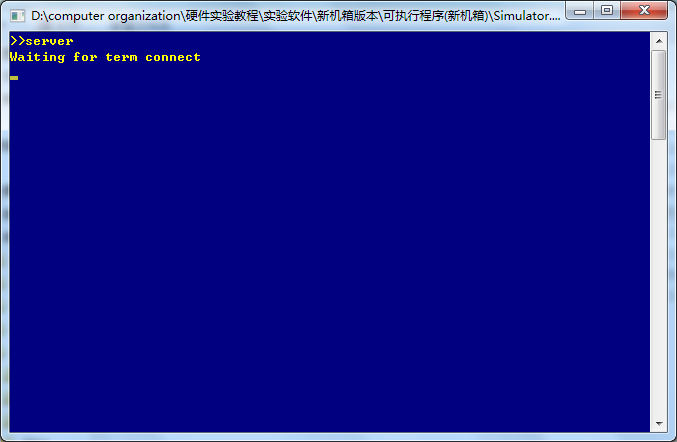


图3.9 模拟器模拟监控程序的运行

使用server后，模拟器会将当前目录下的监控程序二进制文件load到内存中去。操作成功后，模拟器会打开一个socket用以和终端通信。这时除了Ctrl+G以外的一切用户输入都将无效。Ctrl+G键的功能和前面描述的一样，当用户输入该键时模拟器会关闭与串口的通讯并切换到通常的状态。［int］是server的可选参数，如果选择该参数，模拟器在执行过程中会支持时钟中断。同时在int模式下，我们虚拟了教学机上的硬中断。用户可以在模拟器上长按(由于时钟中断产生的较为频繁，如果按键时间过短可能导致模拟器无法响应)Ctrl+U，此时模拟器会产生中断并将操作系统激活。

此时，可启动term程序运行，并在term中输入sim命令，要求term去连接模拟器的socket，而不是连接真正的串口。见到如图3.10 所示界面后，可使用监控程序的A、U、G、D、R等命令。

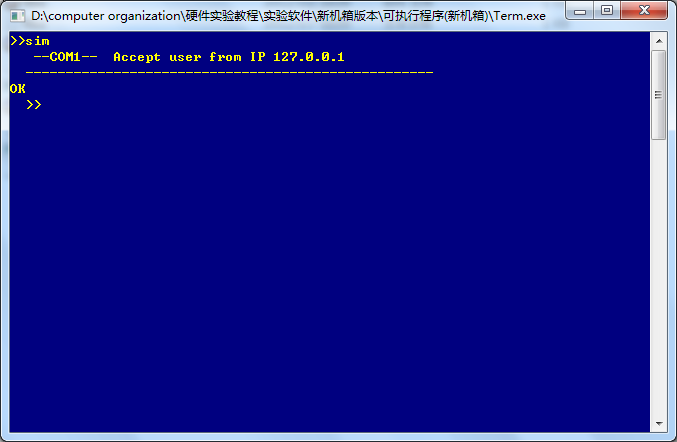


图3.10 教学计算机终端界面（使用模拟器连接）

## 3.5 汇编器

教学计算机汇编器用来对汇编语言程序进行汇编，以生成机器语言代码，可直接装入到模拟器运行，也可通过3.7节介绍的数据通信程序，装入到教学计算机硬件中进行运行。

尽管教学计算机监控程序提供了汇编命令“A”，能实现对单条汇编语句的汇编并保存相应的机器代码，但是，限于监控程序的设计目的之一是为底层的计算机设计提供一个统一的测试平台，监控程序本身不能设计得太复杂，加之教学计算机硬件功能、性能指标均受到限制，对于规模比较大的汇编程序用“A”命令来汇编就显得不太方便。为完成大规模汇编语言程序的汇编工作，我们在PC机上设计了汇编器，由它来生成THINPAD教学计算机的机器语言程序。

汇编程序的使用十分简单，在命令行下直接输入“assembleler.exe input-file output-file”命令即可。其中，input-file是汇编语言源程序的文件名，output-file是输出的执行代码的文件名。汇编完成后，汇编输出的执行代码可装入到模拟器中执行，也可通过数据通信程序装入到硬件平台上执行。

## 3.6 终端程序Term

终端程序term是为教学计算机THINPAD配套的一个程序，主要完成教学计算机硬件的输入和输出功能。它也可和模拟器配合使用，具体方法见3.4节。

终端程序TERM运行在PC机的Windows环境下，通过串口和教学计算机THINPAD进行通信。由于现在的PC机已经不再有通常所说的串口，THINPAD专门配备了一条串口转USB接口的通信线，方便教学机和PC机的USB接口连接。

终端程序Term为教学计算机提供了输入/输出功能，弥补了教学机自身硬件的不足。从具体实现上说，Term是与教学机上运行的监控程序配合使用的，为监控程序提供输入和输出。同时，为简化监控程序的实现，Term也为监控命令的实现提供了支持，使监控程序只需要完成与硬件直接相关的初始化、监控命令的接收和判断、输出字符的发送等功能，而使用汇编程序编写比较困难的监控命令的功能的实现过程则由Term程序自身实现。

Term程序的实现流程如图3.11所示。

失败

成功

打开串口

输出” Ok.. Connected with com...”

输出”Can not open COM1”

接收Kernel发送的”OK”并输出

等待用户输入命令

A命令

U命令

D命令

G命令

R命令

命令执行完毕

图3.11 Term程序框图

从图中可以看出，监控程序5个命令（A、U、D、G和R）的具体实现均是在Term中完成的，而监控程序本身仅仅完成了命令的接收和数据发送的任务。更为详细的实现过程，请同学们阅读本教材所附光盘中的Term的源代码。

## 3.7 数据通信

教学计算机THINPAD与同学们日常使用的PC机不同，它的硬件是一个开放的平台，本身并没有保存任何计算机程序，也就是说，实验中要使用的程序必须先通过某种方式加载到THINPAD的存储器，也就是RAM或者是FLASH中。为方便同学们进行实验，我们为教学计算机配套开发了数据通信程序FlashAndRam，它运行在PC上，通过PC机的串口和教学机串口进行通信，把PC机的文件（程序）加载到教学机中，或者，把教学机存储器中的程序或数据下载到PC机来，供大家分析或后续处理。

显然，要和PC机进行串口通信，首先要准备好教学机的串口，也就是要先把教学机的CPLD芯片中烧入UART，使其完成一个串口的功能。

串口准备好以后，要为PC机和教学机建立连接。首先，要将串口线和JTAG线连接好后，给教学机加电。在PC机上运行FlashAndRam程序，选择好正确的串口，并按“确定”按钮。如不是很确定串口线的USB接口连接在PC机的哪个串口上，可通过Windows下设备管理器中查看USB串口对应的串口号来确定该选哪个串口。

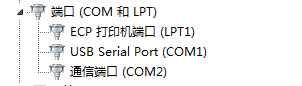


图3.12 串口的选择  
例如图3.12的情况下应该选COM1。

然后，点击界面上的“下载主控程序”按钮，将flashram.bit文件下载到FPGA芯片中，使它能帮助我们完成通信功能。下载完毕后按一下新板上的Reset按钮。此时，教学机和PC机已经建立好通过串口通信的连接，可以互相传送文件了。

为进行PC机和教学机之间的文件交互，FlashAndRam程序在内存中分别为Flash、RAM1和RAM2开辟了缓冲区，两边文件通过对应的缓冲区进行传送。图3.13是FlashAndRam程序的主界面。我们以将文件从ＰＣ机写入Ram2为例来说明文件通信的过程。

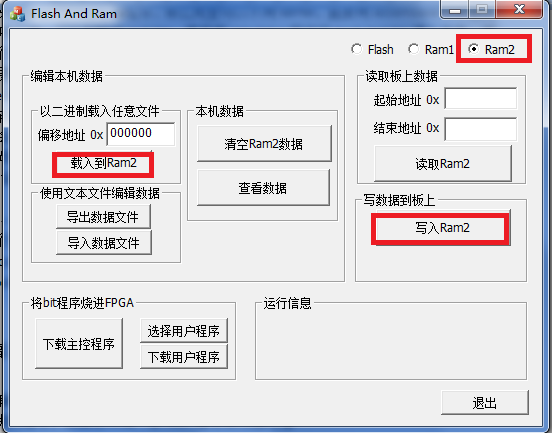


图3.13 FlashAndRam程序的主界面

FlashAndRam程序能将两种格式的文件装入到教学机中。一是二进制文件；另外一种格式是以可见字符表示的二进制文件，这样可以更好地方便大家对二进制文件进行编辑。

对于二进制文件，程序可以将其完整的写入到教学机Ram2存储器指定的起始地址开始的单元中。需要注意的是，由于教学机内存是按字（16位）进行编址，程序将以字为单位写入数据，最后不足部分补“FF”。

对于可见字符表示的二进制文件，用的是“地址=数据”对来表示。它表示了教学机存储器的每个地址单元的内容。例如，用“0000-2323=1B2D”表示从地址0~地址2323的内容均为“1B2D”；“002425=0000”表示2425地址中存放“0”。

数据写入的操作过程比较简单，首先确定要写入的二进制文件的格式，如为第一种格式，则选择点击“载入到Ram2”按钮，将文件写入到程序的缓冲区中，并可以用“查看数据”按钮进行检查，指定好导入的目的地址后，按“写入Ram2”按钮即可。

如果是第二种格式，则选择“导入数据文件”按钮，后续步骤同第一种格式。

写入Flash、Ram1的过程与此类似，不再赘述。

将教学机存储器内容导出的过程是写入的逆过程，大家可自行体验。

# 第4章 VHDL硬件描述语言

## 4.1概述

硬件描述语言HDL(Hardware Description Language)，顾名思义就是可以表述硬件的语言，它可以用文本语言的形式描述硬件电路的功能，信号连接关系以及时序关系等，它的出现为硬件设计提供了一种更加快捷精确的描述方法，成为了连接硬件设计者和电子设计自动化(EDA)软件工具之间的桥梁。相对于以往的图形输入，硬件描述语言虽然没有那么直观，但是其描述能力更强，可描述的层次也更高，因而可以进行大规模的数字系统的设计。

随着电子技术的不断发展和硬件电路规模的不断扩大，图形描述的方法已经远远不能描述现有的电路设计，其一是电路的规模越来越大，采用电路图的门级描述方式就有产生异常大的描述规模，不光难以管理，查错也会异常困难；其二是目前硬件设计主要采用自顶向下的设计方法，对于高层次的电路抽象描述，图形方式很难表述的非常清晰。因此，更加灵活、能力更强、层次更高的硬件描述语言逐渐替代了以往的图形描述方式。

目前硬件描述语言就像软件设计领域中的c和c++一样，已经在硬件设计领域中占据了主导地位，它具有以下特点：

* 可以在抽象层次上对设计进行精确而简练的描述；
* 可以很方便的在不同层次上对设计进行模拟和验证；
* 可以进行软硬件协同设计，大大减小了软件设计和硬件设计上的时间差；
* 设计易于修改，可理解性强；
* 有很多的可用的辅助软件；
* 易于生成设计文档资料。

目前最常用的HDL有VHDL和Verilog HDL。VHDL诞生于1982年，在1987年底，VHDL被IEEE和美国国防部确认为标准硬件描述语言。自IEEE公布了VHDL的标准版本，IEEE- 1076（简称87版)之后，各EDA公司相继推出了自己的VHDL设计环境，或宣布自己的设计工具可以和VHDL接口。此后VHDL在电子设计领域得到了广泛的接受，并逐步取代了原有的非标准的硬件描述语言。同时，Verilog HDL是在用途最广泛的C语言的基础上发展起来的一种硬件描述语言，它是由GDA(Gateway Design Automation)公司的PhilMoorby在1983年末首创的，最初只设计了一个仿真与验证工具，之后又陆续开发了相关的故障模拟与时序分析工具。1985年Moorby推出它的第三个商用仿真器Verilog-XL,获得了巨大的成功，从而使得Verilog HDL迅速得到推广应用。1989年CADENCE公司收购了GDA公司，使得VerilogHDL成为了该公司的独家专利。1990年CADENCE公司公开发表了Verilog HDL,并成立LVI组织以促进Verilog HDL成为IEEE标准，即IEEE Standard 1364-1995。这两种语言各有千秋，都有各自的支持者，语言的选择主要还是取决于设计本身和所用工具的支持。当然，随着技术的不断进步，又出现了更高层次的硬件设计语言，比如SystemC、System Verilog等，这些都可以支持系统级建模，但是他们还没有被广泛使用。

VHDL(Very-High-Speed Integrated Circuit HardwareDescription Language)是本书采用的硬件设计语言。VHDL主要用于描述数字系统的结构，行为，功能和接口，与其他硬件描述语言相比，VHDL具有以下特点：

* 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电路。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。

* 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。在硬件电路设计过程中，主要的设计文件是用VHDL编写的源代码，因为VHDL易读和结构化，所以易于修改设计。

* 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。而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，还可以准确地建立硬件电路模型。VHDL支持预定义的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。

* 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。

* 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。

* 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，可以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。

本书中的VHDL部分知识简单的介绍了一些VHDL的基本语法，读者如果想要学习一些深入的内容，请参考其他一些专门的语言类书籍。

## 4.2 程序结构

### 4.2.1基本结构

一个完整的VHDL程序就是一个电路的抽象，他可以代表一个复杂的电路系统、一块电路板、一片芯片、一个电路模块或者一个门电路，这样的VHDL程序被称作一个设计单元，他抽象了一个实际电路。一个VHDL程序的结构如图4.1所示。

图4.1 VHDL程序组成

一个设计单元有以下几部分组成：实体（Entity）、结构体（Architecture）、配置（Configuration）、包集合（Package）、库（Library）。下例为一个2输入与门的VHDL程序。

例 一个2输入的与门的逻辑描述

LIBRARY ieee; --库说明语句

USE ieee.std\_logic\_1164.ALL; --程序包说明语句

ENTITY and2 IS

PORT(a,b : IN STD\_LOGIC;

实体

y : OUT STD\_LOGIC);

END and2;

ARCHITECTURE and2x OF and2 IS

BEGIN

结构体

y<=a AND b;

END and2x;

从上面的例子可以看出，实体部分描述了设计系统的外部接口信号（即输入/输出信号），它从整体上表示了设计单元的外部特性；结构体则描述了描述设计单元的内部电路，它显示了设计单元的行为、数据的流程或组织结构形式；配置在例子中没有体现，它用于从库中选取所需元件安装到设计单元的实体中，来组成设计的不同规格的不同版本，使设计单元的功能发生变化；包集合存放了各设计模块能共享的数据类型、常数、子程序等；库里存放了已编译的实体、结构体、包集合和配置。

### 4.2.2实体（ENTITY）

实体是VHDL中描述元件外部特性的部分，即元件的输入和输出（I/O）的端口等信息，它并不描述元件的具体功能，在电路原理图上实体相当于元件符号。在层次化系统设计中，顶层的实体可以是整个系统模块或整个单元模块的输入、输出（I/O）描述；在一个器件级的设计中（底层），实体可以是一个元件或芯片的输入、输出（I/O）描述；实体在VHDL程序设计中描述一个元件或一个模块与设计系统的其余部分（其余元件、模块）之间的连接关系，可以看作是一个电路符号。如下面的例子就描述了一个二选一的多路选择器的实体部分，对应于元件符号就是如图4.2所示。

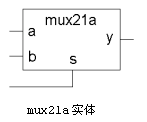


图4.2 元件符号

ENTITY mux21a IS

PORT( a, b : IN BIT ;

s : IN BIT;

y : OUT BIT ) ;

END ENTITY mux21a ;

一个实体由实体名、类属表、端口表、实体说明部分和实体语句部分组成，VHDL中实体的语法格式如下所示：

ENTITY 实体名 IS

[GENERIC（类属表）；]

[PORT ( 端口表）;]

END [ENTITY] 实体名;

在这里ENTITY、GENERIC、PORT、IS、BEGIN、END是VHDL的关键字（保留字），实体名、端口名等均应为符合VHDL命名规则的标识符。实体中的每一个I/O信号被称为端口，其功能对应于电路图符号的一个引脚。端口说明则是对一个实体的一组端口的定义，即对基本设计实体与外部接口的描述。端口是设计实体和外部环境动态通信的通道。

类属参数说明是可选部分，用来指定该设计单元的类属参数（如延时、功耗等），类属参数说明必须放在端口说明之前。参数在该模块被调用时从外部传入参数值，参数值为本实体所属的所有结构体使用，传入的类属参数作为常量使用，在使用时不能修改。类属特性的设计为多个高层设计实体使用同一个元件提供了静态参数上的区别。例如，一个通用的计数器元件，在高层使用时通过类属参数的指定，可以设定为8位、16位或者24位的计数器。

类属参数说明的语法格式为：

GENERIC（端口名{，端口名}：[IN] 子类型 [：=初始值]

{；端口名{，端口名}： [IN] 子类型 [：=初始值]} ）；

例如指定上升沿时间为3ns的说明如下：

GENERIC（rising\_edge: TIME:=3ns）

还可能会有实体说明部分，它是实体接口中的公共信息，放在端口说明之后，比如定义一些子类型说明等。

端口为实体提供了与外部通信的通道，每个端口就相当于一个元件I/O，对应于电路图符号的一个引脚。端口说明中的每一个I/O信号被称作一个端口,一个端口就是一个数据对象，必须有端口名、端口模式和数据类型。

端口模式用来说明数据、信号传输通过该端口的方向。端口模式有以下几类：

* IN（输入）：定义的端口为单向只读模式，数据只能通过该端口被读入实体中，仅允许数据流进入端口，主要用于时钟输入、控制输入、单向数据输入。
* OUT（输出）：定义的端口为单向输出模式，数据只能通过该端口从实体向外流出，或者说可以将实体中的数据向此端口赋值。该模式通常用于终端计数一类的输出，不能用于反馈。
* INOUT（双向）：定义的端口为双向输入输出模式，允许数据流入或流出该实体。这个端口相当于一个输入端口和一个输出端口的组合。从端口内部看，可以对此端口进行赋值，也可以通过此端口读入外部的数据信息；而从端口的外部看，信号既可以从此端口流出，也可以向此端口输入信号，如RAM的数据端口。该模式允许用于内部反馈。
* BUFFER（缓冲）：功能与INOUT类似，但与其的区别是当需要输入数据时，只允许回读内部信号，即允许反馈，该回读信号不是由外部输入，而是由内部产生。如计数器设计，可将计数器输出的计数信号回读，以作为下一计数值的初值。该模式允许数据流出该实体和作为内部反馈时用，但不允许作为双向端口使用。

如果端口模式没有指定，则该端口处于缺省模式为：IN 。

端口的数据类型原则上可以是任何标准的数据类型和用户自定义类型，这个根据需要自由选择，但是要注意当端口连接的时候数据类型的匹配。

图4.3以一个多端口的电路图。

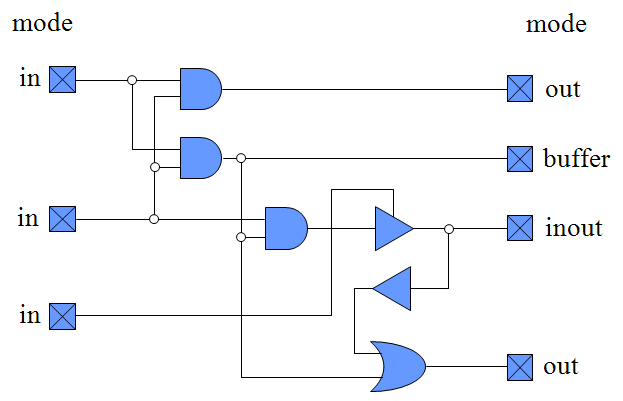


图4.3 端口说明

这个电路图对应的实体有三个输入端口，两个输出端口，一个双向端口和一个缓冲端口，其对应的实体如下：

ENTITY ent IS

PORT( a, b , c : IN BIT;

e,f : OUT BIT;

g :BUFFER BIT;

d : INOUT BIT ) ;

END ENTITY ent;

### 4.2.3结构体（ARCHITECTURE ）

结构体是元件的功能部分，具体说明了该设计实体的行为，定义了该设计实体的功能，规定了该实体的数据流程，指定了实体中内部元件的连接关系。若把设计实体抽象为一个功能方块图，结构体则描述这个功能方块图的内部实现细节。对于一个实体来说，具体的电路实现可能有多种，因此一个电路系统程序设计的一个实体，可以对应多个结构体，以对应该元件的不同设计方案，通过配置来选择不同的实现结构体。当然，一个独立的VHDL文件一般由一个实体说明和一个结构体组成。例如图4.2中的二选一多路选择器的实体，其可能的结构体图如下图4.4所示。

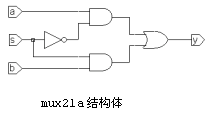


图4.4 结构体

这个图示所对应的结构体如下所示：

ARCHITECTURE one OF mux21a IS

BEGIN

y <= (b and s) or (a and (not s));

END ARCHITECTURE one;

从上面的例子也可以看成，使用语言来描述这个元件比使用原理图要方便很多。

VHDL中结构体语法结构如下：

ARCHITECTURE 结构体名 OF 实体名 IS

[结构体说明部分];

BEGIN

[并发处理语句]；

END 结构体名；

结构体是对应于实体存在的，因此他必须有一个存在的实体，并且显式的表示出来。

结构体说明语句是对结构体的功能描述语句中将要用到的信号（SIGNAL）、数据类型（TYPE）、常数（CONSTANT）、函数（FUNCTION）和过程（PROCEDURE）等加以说明的语句。结构体中说明和定义的数据类型、常数、元件、函数和过程只能用于这个结构体中，若希望其能用于其他的实体或结构体中，则需要将其作为程序包来处理。

并发处理语句位于BEGIN和END之间，这些语句具体地描述了构造体的行为。并发处理语句是功能描述的核心部分，也是变化最丰富的部分。并发处理语句可以使用赋值语句、进程语句、元件例化语句、块语句以及子程序等。需要注意的是，这些语句都是并发（同时）执行的，与排列顺序无关。

### 4.2.4配置（CONFIGURATION）

当一个实体多个结构体时，就需要使用配置来确定元件使用哪个结构体来构成，配置描述了实体与结构之间的连接关系。在实体与结构体之间连接关系的配置中，设计者可以利用配置语句为实体选择不同的结构体，通过仿真进行性能对比，从而获得最佳的设计目标。

配置指定语言是在元件例化的时候，用来指定该元件使用的是实体的哪一个结构体，具体的语法如下：

for 例化元件标号表 : 元件模型 use entity 实体名(结构体名)

[generic map (参数关联表)]

[port map (端口关联表)]；

比如实体Half\_Adder有两个结构体A1，A2，分别使用不同的实现方式，如果我想要使用其中的结构体A1作为元件ADD1例化过程中使用的结构体，可以使用以下语句：

for ADD1: Hadd use entity Half\_Adder(A1);

配置还有很多其他的语法使用，这里就不一一详解了。

### 4.2.5包集合（PACKAGE）

对于一个复杂的系统，可能很多个实体需要共享一些数据类型、常量与子程序等，这些可以在一个实体说明部分和结构体部分加以说明，在其相应的结构体中是可见的，即可以被使用的，但是，在另外一个实体的说明部分与结构体部分中定义的数据类型、常量及子程序对于同一个工程中其他设计实体是不可见的。为了使一组类型说明、常量说明和子程序说明等对同一工程中多个设计实体都成为共享的，VHDL提供了包集合。

包集合是一种使其中的数据类型、常量、子程序和其它说明对其它设计实体可见的设计单元。它将常用的相关说明（类型说明、常量说明、子程序说明、元件说明、属性说明等）搜集在一起，一起编译后存入库中，如果其他设计实体想要使用包里的内容，只需要通过USE语句调用即可。包集合包括包头和包体两部分。包头为包集合定义了接口，用来声明包中的类型、元件、函数和子程序；而包体规定集合包的实际功能，用来存放说明中的函数和子程序，未含有子程序和函数的包集合不需要包体。

包集合的语法结构如下：

PACKAGE ＜包名＞ IS

［外部子程序说明］

［外部常量说明］

［外部元件模板］

［外部类型说明］

［属性说明］

［属性指定］

END ［＜包名＞］

PACKAGE BODY ＜包名＞ IS

［外部子程序体］

［内部子程序说明］

［内部子程序体］

［内部常量说明］

［内部类型说明］

END ［＜包名＞］

一个包头只能有惟一的一个包体与之对应，二者使用相同的名字。包体的内容是基本说明和子程序体说明。但要注意，若包集合中含有子程序说明时，必须将子程序放在对应的包体中。包体中的子程序及其相应的说明是专用的，不能被其它VHDL单元所引用；而包集合中的说明是公用的，它可以独立地编译并插入设计库中。包集合体是次级设计单元，只有在其对应的主设计单元编译并插入设计库之后，才可独立地编译并插入到设计库中。

下面是一个包的实现示例：

PACKAGE Logic IS

TYPE three\_level\_loglc IS（‘0’，‘1’，‘Z’）；

CONSTANT unknown\_value： three\_level\_logic：＝‘0’；

FUNCTION invert（input：three\_level\_loglc）

RETURN three\_level\_logic；

END logic；

PACKAGE BODY logic IS

FUNCTION invert（input：three\_level\_logic）

RETURN three\_level\_logic IS

BEGIN

CASE input IS

WHEN ‘0’＝＞ RETURN ‘1’；

WHEN ‘1’＝＞ RETURN ‘0’；

WHEN ‘ Z’＝＞ RETURN ‘ Z’；

END CASE；

END invert；

END logic；

一个包集合所定义的内容如果想要在某个VHDL设计实体中使用，需要在实体说明之前加上USE语句，则可以使得包集合说明中的内容在该设计实体中使用。

如果某个设计实体想要使用上述包集合logic中的类型three\_level\_logic和函数invert，则需要USE语句引用他们，具体实例如下：

USE logic.three\_level\_logic，logic.invert；

ENTITY inverter IS

PORT （ x：IN three\_level\_logic；

y：OUT three\_level\_logic）；

END inverter；

ARCHITECTURE inverter\_body OF inverter IS

BEGIN

PROCESS

BEGIN

y <= invert(x)；--一个函数调用

END PROCESS；

END inverter\_body；

对于结构体来说，它会继承实体说明部分的内容，所以不必再使用USE语句。USE语句后跟保留字ALL，表示使用库或程序包中的所有定义。

### 4.2.6库（Library）

包集合、实体、结构体编译之后可以存放在库里，通过其目录可查询和调用库里的内容。库的功能类似于操作系统中的目录，存放设计的数据，使设计者可以共享已经编译过的设计结果。库的引用说明总是放在设计实体的最前面，其语法结构如下：

Library 库名；

Use ＜库名＞．＜包名＞．a11；

引用后，在设计实体内的语句就可以使用库中的内容了，在VHDL语言中可以存在多个不同的库，但库与库之间是独立的，不能互相嵌套。VHDL语言中的库分为设计库和资源库两类，如图4.5所示：

图4.5 VHDL库

（1）设计库

设计库是由VHDL标准规定的，对所有项目是默认可见的，不需要用Library、USE子句声明的库。STD库和WORK库是设计库。

STD库：VHDL的标准库，预定义了STANDARD和TEXTIO两个包集合，这两个包集合是使用VHDL语言时最常用到的内容。Standard的包集合不需要引用，其中所有预定义的数据类型和函数都可以使用，为所有设计单元所共享、隐含定义、默认和可见。若要使用TEXTIO包集合中的内容时，应先说明库和包集合名，然后才可使用该包集合中的内容。

WORK库：是VHDL语言的工作库，用户在工程中已经设计好的，正在校验、未仿真的中间件等都放在工作库中。实际上WORK是一个临时的仓库，用来存放成品或半成品。WORK库对所有设计都是隐含可见的，因此在使用该库时无需进行任何说明。

也就是说，在每段VHDL程序中都隐含有下面的不可见的代码：

Library work；

Library std；

Use std.standard.all；

（2）资源库

除了STD和WORK库以外，其他的库都为资源库，这些库中的内容若想使用都必须加以预先声明引用。资源库用来存放编译好的常规元件和标准模块，在有些库中，存放的元件、函数都是IEEE标准化组织认可的，称为IEEE库，如STD\_LOGIC\_1164、STD\_LOGIC\_ARITH（算术运算库）、STD\_LOGIC\_UNSIGNED等。通常，VHDL工具厂商与EDA工具专业公司也有自己的资源库，比如ASIC矢量库，为了进行门级仿真，各公司可提供面向ASIC的逻辑门库，在该库中存放着与逻辑门一一对应的实体。当然，用户也可以有自己定义的库，简称用户库，是由用户自己创建并定义的库。设计者可以把自己经常使用的非标准（一般是自己开发的）包集合和实体等汇集成在一起定义成一个库，作为对VHDL标准库的补充。

常用的IEEE资源库包含的程序包如下：

STD\_LOGIC\_1164

NUMERIC\_BIT

MATH\_REAL

库中包集合的使用如下例所示：

LIBRARY IEEE；

USE IEEE.STD\_LOGIC\_1164.ALL；

## 4.2.7 结构体子结构

在规模较大的电路设计中，整个电路将被分成若干个相对独立的模块来描述。这样，一个结构体可以用几个子结构，即相对独立的几个模块来构成。VHDL语言有以下3种形式的子结构描述语句：块BLOCK语句结构、进程PROCESS语句结构和子程序SUBPROGRAMS结构。

1、块BLOCK

一般一个大规模的电路通常可以分成多个子模块，以便于设计和开发。同样，在VHDL程序中，结构体内部也可以由多个BLOCK块组成，每一个BLOCK块则对应一个子模块，其原理就如同电路图被分成若干各部分，每个子原理图对应于一个VHDL程序中的BLOCK块。

BLOCK的语法格式如下：

块标号：BLOCK [(块保护条件)]

[说明语句]；

BEGIN

[并发处理语句]；

END BLOCK 标号名；

其中保护条件是可选项，它是一个布尔表达式，只有当其为真时，该块中的语句才被启动执行；否则，该块中的语句不被执行，如果有保护条件，则该条件应用圆括号括起来，放在BLOCK之后。BLOCK语句中所描述的各个语句是可以并行执行的，它和书写顺序无关。下面是一个二选一选择器的VHDL实现，其中就是用了块语句。

ENTITY mux2\_1 IS

PORT(d0, d1, sel : IN STD\_LOGIC;

q : OUT STD\_LOGIC);

END mux2\_1;

ARCHITECTURE amux OF mux2\_1 IS

SIGNAL tmp1,tmp2,tmp3 : STD\_LOGIC;

BEGIN

B1: BLOCK

BEGIN

tmp1<=d0 AND sel;

tmp2<=d1 AND (not sel);

END BLOCK B1;

B2:BLOCK

BEGIN

tmp3<=tmp1 OR tmp2;

q<=tmp3;

END BLOCK B2;

END amux;

2、进程PROCESS

PROCESS语句是一种并发处理语句，在一个构造体中多个PROCESS语句可以同时并发运行。因此，PROCESS语句是VHDL中描述硬件系统并发行为的最常用、最基本的语句。我们会在后面详细介绍其用法。

3、子程序

VHDL的子程序有两种类型：过程（PROCEDURE）和函数（FUNCTION）。他们可以在结构体或包集合的任何位置被调用，而且可以反复调用。子程序是一个非重入的程序，即子程序返回后才能再被调用，不能递归调用。在调用时子程序首先要进行初始化，执行结束后子程序终止；再调用时要再进行初始化。因此，子程序内部的值不能保持。VHDL的子程序具有可重载性，即允许有许多重名的子程序，但这些子程序的参数类型和返回数值类型是不同的。

过程（PROCEDURE）定义的语法结构如下：

PROCEDURE 过程名（参数1；参数2；… ）IS

[定义语句]；

BEGIN

[顺序处理语句]；

END 过程名；

调用过程语句的时候只要加上参数表就可以了，下面是一个求最大值的过程定义和调用示例：

PROCEDURE max(a, b: IN INTEGER;

y: OUT INTEGER) IS

BEGIN

IF (a<b) THEN

y<=b;

ELSE

y<=a;

END IF;

END max;

过程的调用：max ( x, y, maxout ) ;

函数（FUNCTION）的用法同过程类似，但是函数比过程多了返回值，其语法结构如下：

FUNCTION 函数名（输入参数表）RETUEN 数据类型 IS

[定义语句]；

BEGIN

[顺序处理语句]；

RETUEN [返回变量名]；

END [函数名]；

函数定义的时候有返回的类型，函数体中也需要有RETURN语句来返回结果。

## 语言元素

### 4.3.1标识符

VHDL语言是不区分大小写的，但所使用的名字，如信号名、实体名、结构体名、变量名、各种进程标记、块标记等这些标示符进行命名时，应遵守如下规则：

* 第一个字符必须是字母；
* 构成名字的字符只能用英文字母、数字和下划线；
* 不能连续使用下划线，标示符的最后一个字符也不能用下划线。
* VHDL的保留字（关键字）不能用于标识符；

如下标识符是合法的：

tx\_clk

three\_state\_Enable

sel7D

HIT\_1124

如下标识符是非法的：

\_tx\_clk 标识符必须起始于字母

8B10B

large#number 只能是字母、数字、下划线

link\_\_bar 不能有连续两个下划线

select 关键字（保留字）不能用于标识符

rx\_clk\_ 最后字符不能是下划线

在编写VHDL程序的时候，为了使程序结构清晰，具有更好的可读性，建议采用以下的书写规则：

* 对VHDL语言的保留字，习惯上用大写，其他应小写。但有一种情况需要注意，代表不定状态的“X”和高阻态的“Z”要求必须大写。
* 单词、信号名的含义要明确，以免造成混乱。
* 要求段落分明、含义确切，嵌套关系一目了然。
* 应辅以适量的程序注释。VHDL语言中使用的注释符是“--”，从注释符号“--”开始到该行末尾结束。所注释的文字不作为语句来处理，不产生硬件电路结构，不描述电路硬件行为。

### 4.3.2数据对象

在VHDL语言中，数据对象是可以赋予一个值的客体，即可以存储值的容器，常用的数据对象有四类：常量、变量、信号和文件。信号和变量可以连续地被赋值，而常量只能在它被说明时赋值。文件内容是不可以通过赋值来更新的，文件可以作为参数向子程序传递，通过子程序对文件进行读和写操作，文件参数没有类型。

#### 常量（CONSTANT）

常量是一个固定的值，在设计描述中不会变化，一般是全局量，通常可以看做电路中不变的那些值，比如电源、地等常数。在常量说明的过程中就对其赋予一个固定的值，通常赋值在程序开始前进行，该值的数据类型则在说明语句中指明。常量说明的一般语法格式如下：

CONSTANT 常量名{，常量名}：数据类型 ：= 表达式；

例如：

CONSTANT VCC：REAL：= 5.0；

CONSTANT DELAY：TIME：=100 ns；

常量一旦被赋值就不能再改变。上面VCC被赋值为 5.0V，那么在所在的VHDL语言程序中VCC的值就固定为5.0V，不像后面所提到的信号和变量那样，可以任意赋值不同的数值。另外，常量所赋的值应和定义的数据类型一致。

#### 变量（VARIABLE）

变量是暂存数据的局部量，它在电路中没有实际对应的对象，只能在进程语句、函数语句和过程语句结构中使用。在仿真过程中，它不像信号那样，到了规定的仿真时间才进行赋值，变量的赋值是立即生效的。变量说明的语法格式为：

VARIABLE 变量名{，变量名}：数据类型 [：=初始值]；

变量声明例如：

VARIABLE x，y： INTEGER；

VARIABLE count: INTEGER RANGE 0 TO 99 : =0；

由于变量是一个局部量，它必须在进程或子程序的说明区域中加以说明。

变量赋值的时候表达式必须与目标变量具有相同的数据类型，赋值语法格式为：

目标变量名 ：= 表达式；

变量赋值是直接的、非预设的，它在某一时刻仅包含一个值。变量的赋值立即生效，不存在延时行为。变量常用在实现某种运算的赋值语句中。变量在赋值时不能产生附加延时。例如，tmp3是变量，那么下式产生延时的方式是不合法的：

tmp3 ：= tmpl ＋ tmp2 AFTER 10 us ；

#### 信号（SIGNAL）

信号是硬件设计语言特有的一种对象，是电子电路内部硬件之间相互的连线的抽象表示，除没有方向外，与“端口”概念相似。其实，端口说明中的量都是信号，由于是默认，因此不加SIGNAL标示。信号是是全局量，通常在构造体、程序包和实体中说明，信号不能在进程中说明（但可以在进程中使用）。信号说明的语法格式为：

SIGNAL 信号名{，信号名} ：数据类型 [：=初始值]；

例如：SIGNAL sysclk： STD\_LOGIC；

在程序中，信号值的赋值采用“<=”赋值符，而不是像变量赋值时用“：=”符，语法格式为：

目标信号名 ＜= 表达式；

赋值语句中的表达式必须与目标信号具有相同的数据类型。信号一般包括I/O引脚信号以及IC内部缓冲信号，有硬件电路与之对应，因此信号之间的传递（赋值）有实际的附加延时。例如，s1和s2都是信号，且s2的值经10 ns延时以后才被赋值s1。此时信号传送语句可书写为：

s1 <= s2 AFTER 10 ns；

可以想象出来，硬件中的连线总是同时工作的，对应于信号，他也是同时在各个模块中流动，这就是硬件电路的并发性。硬件设计语言HDL体现了实际电路中信号“同时”流动的这种基本特性。

#### 信号和变量的区别

信号是硬件中连线的抽象描述，有实际的物理意义；变量在硬件中没有类似的对应关系，它们用于硬件特性的高层次建模所需的计算中，用作进程中暂存数据的单元。变量是一个局部量，只能用于进程或子程序中；信号是一个全局量，它可以用来进行进程之间的通信。

因为电流流动一定有时间延迟，信号赋值因此至少有δ延迟（如果不指定延迟，则延迟接近于0），而变量因为不是实际物理量，因此变量的赋值就不允许有延迟。进程（PROCESS）语句只对信号敏感而不对变量敏感。信号和变量值的赋值不仅形式不同，而且其操作过程也不相同。在变量的赋值语句中，变量的赋值符为“：=”，该语句一旦被执行，其值立即被赋予变量。在执行下一条语句时，该变量的值就为上一句新赋的值。信号赋值语句采用“<=”赋值符，该语句即使被执行也不会使信号立即发生赋值。下一条语句执行时，仍使用原来的信号值。由于信号赋值语句是同时进行处理的，因此，实际赋值过程和赋值语句的处理是分开进行的。

信号赋值可以出现在进程中，也可以直接出现在结构体中，但它们的运行含义不同：前者属顺序信号赋值，此时的赋值操作要视进程是否已被启动；后者属并行信号赋值，其赋值操作是各自独立并行发生的。

下面以一个例子来详细介绍一下进程中信号赋值与变量赋值的区别，由于信号a发生变化使进程语句开始启动执行。这样一来，仿真器对进程中的各语句自上至下地进行处理。

例a：

PROCESS（ a，b，c，d）

BEGIN

d <= a；

x <= b + d；

d <= c；

y <= b + d；

END PRO0CESS；

结果

x <= b + c；

y <= b + c；

当进程运行时，信号赋值将自上而下顺序执行，但第一项赋值操作并不会发生，因为信号赋值是在进程结束时才起作用。因为在进程结束更新时，d的最后一个赋值为c ，因此执行结果为x，y均为b+c。

例b：

PROCESS（a，b，c）

VARIABLE d：STD\_LOGIC\_VECTOR（3 DOWNTO 0）；

BEGIN

d := a；

x <= b + d；

d := c；

y <= b + d；

END PROCESS；

结果

x <= b + a；

y <= b + c；

在这个进程中，d是变量。在执行“d ：=a”语句以后，a的值就被赋给d，所以x为b + a。此后又执行“d ：= c”，从而使y为b + c。

从上面的例子可以看出，信号量的值将进程语句最后所赋值的值作为最终赋值值，而变量的值一经赋值就变成新的，这就是变量赋值和信号赋值在操作上的区别。这一点还会在信号赋值语句中详细介绍。

### 4.3.3数据类型

VHDL语言是一种强数据类型语言，任一常量、信号、变量、函数和参数在声明时必须声明类型，使用时必须保持数据类型的一致性，不同类型之间的数据不能直接赋值，即使数据类型相同，而位长不同时也不能直接赋值，这种特性可以帮助设计者在设计前期发现错误。VHDL提供了多种标准的数据类型，另外，为使用户设计方便，还可以由用户自定义数据类型。这样使语言的描述能力及自由度更进一步提高，从而为系统高层次的仿真提供了必要手段。

#### 标准数据类型

VHDL提供了10种标准数据类型，如表4.1所示：

表4.1 VHDL的标准数据类型

|  |  |
| --- | --- |
| 数据类型 | 含 义 |
| 整数 | 整数32位，取值范围：－(－1) ～ (－1) |
| 实数 | 浮点数，取值范围：-1.0e+38 ～ 1.0e+38 |
| 位 | 逻辑’0’或’1’ |
| 位矢量 | 位矢量，用双引号括起来的一组数据 |
| 布尔量 | 逻辑“真” 或“假”，用TRUE和FALSE标记 |
| 字符 | ASCII字符 |
| 字符串 | 字符矢量 |
| 时间 | 时间单位fs，ps，ns，μs，ms，sec，min，hr |
| 自然数、正整数 | 整数的子集：自然数取值范围为0 ～（－1）；正整数是大于0的整数 |
| 错误等级 | Note，warning，error，failure |

（1）整数（integer）

整数与数学中的整数的定义相同。整数的表示范围从一（－l）到（一1）。整数的例子如 136，12456，-457。一个整数类型的信号或变量，要被综合进逻辑时在其范围上应有约束。例如：variable a: integer range –255 to 255;

尽管整数值在电子系统中可能是用一系列二进制位来表示的，但是整数不能看作是位矢量，也不能按位来进行运算，对整数不能用逻辑操作符。当需要进行位操作时，可以用转换函数，将整数转换成位矢量。

（2）实数（real）

实数的定义值范围为-1.0E＋38 ～ 1.0E＋38,和整数一样，实数能被约束。实数有正负数，书写时一定要有小数点。例如：

-1.0，2.5，-1.0E38

有些数可以用整数表示也可以用实数表示。例如，数字1的整数表示为1，而用实数表示则为1.0。两个数的值是一样的，但数据类型却不一样。由于实数运算需要大量的资源，因此综合工具常常并不支持实数类型。

（3）位（bit）

在数字系统中，信号值通常用一个位来表示。位值的表示方法是，用字符‘0’或者‘1’表示之。位与整数中的1和0不同，‘0’和‘1’仅表示一个位的两种取值。另外，位不能用来描述三态信号。

（4）位矢量（bit\_vector）

位矢量是用双引号括起来的一组位数据,每位只有两种取值：0和1，在其前面可加以数制标记，如X（16进制）、B（2进制、默认）、O（8进制）等，例如：“001100”，X“00BB”。用位矢量数据表示总线状态最形象也最方便。

（5）布尔量（boolean）

又称逻辑量，一个布尔量具有两种状态，“真”或者“假”，分别用TRUE和FALSE标记，用于关系运算和逻辑运算。虽然布尔量也是二值枚举量，但它和位不同，布尔量没有数值的含义，也不能进行算术运算。例如，它可以在IF语句中被测试，测试结果产生一个布尔量TRUE或者FALSE。

一个布尔量常用来表示信号的状态或者总线上的情况。如果某个信号或者变量被定义为布尔量，那么在仿真中将自动地对其赋值进行核查。一般这一类型的数据的初始值总为 FALSE。

（6）字符（character）

字符是用单引号括起来的一个字母、数字、空格或一些特殊字符（如$、@、%等），如‘A’。一般情况下VHDL对大小写不加以区分，但是对字符中的大、小写字符则认为是不一样的。例如，‘A’不同于‘a’。程序包STANDARD中给出了预定义的128个ASCII码字符类型，不能打印的用标识符给出。

（7）字符串（string）

字符串是由双引号括起来的一个字符序列，它也称字符矢量或字符串数组，区分大、小写字母，常用于程序的提示和结果说明等。例如：

“Integer range”

（8）时间（time）

时间是一个物理量数据，取值范围从－（－1）～（－1）。完整的时间量数据应包含整数和单位两部分，例如，55sec，2 min等。在程序包STANDARD中给出了时间的预定义，其单位为fs，ps，ns，µs，ms，sec，min，hr。下面是时间数据的例子：

在系统仿真时，常用于指定时间延时和标记仿真时刻从而使模型系统能更逼近实际系统的运行环境。时间在逻辑综合中不起作用。

（9）Natural（自然数）和Positive（正整数）

自然数和正整数是整数类型的子类型。自然数取值范围为0 ～（－1）；正整数是大于0的整数。

（10）错误等级

错误等级类型数据用来表征系统的状态，它共有4种：NOTE（注意），WARNING（警告），ERROR（出错），FAILURE（失败）。在系统仿真过程中可以用这4种状态来提示系统当前的工作情况。这样可以使操作人员随时了解当前系统工作的情况，并根据系统的不同状态采取相应的对策。

上面的十种数据类型是VHDL种标准的数据类型，在编程时可以直接引用。如果用户需使用其它的数据类型，则需要调用一些库，或者自定义一些需要的数据类型。

在IEEE库STD\_LOGIC\_1164程序包中有两个我们做常用的数据类型，分别是STD\_LOGIC类型和STD\_LOGIC\_VECTOR类型。STD\_LOGIC类型的数据可以具有九种取值，其含义如下：

’U’：初始值

’X’：不定态

’0’： 强制0

’1’： 强制1

’Z’： 高阻态

’W’：弱信号不定态

’L’：弱信号0

’H’：弱信号1

’\_’：不可能情况（可忽略值）

其中，“X”方便了系统仿真，“Z”方便了双向总线的描述。

std\_logic\_vector是由 std\_logic 构成的数组。定义如下：

type std\_logic\_vector is array(natural range<>) of std\_logic;

在赋值的时候也需要保证相同数据类型，并且数据宽度相同。

#### 用户自定义数据类型

为了用户使用方便，VHDL允许用户自己定义数据类型，其定义语法结构如：

TYPE 数据类型名 {，数据类型名} IS [数据类型定义];

常用的用户自定义数据类型主要包括枚举类型、整数类型、实数类型、数组类型、记录类型。存取类型、文件类型等。

（1）枚举类型（enumerated type）

通过列举某类变量所有可能的取值来加以定义。对这些取值，一般使用自然语言中有相应含义的单词或字符序列来代表，以便于阅读和理解。语法格式如下：

TYPE 数据类型名 IS （元素1，元素2，…）；

例如，在程序包STD\_LOGIC\_1164中对STD\_LOGIC的定义为

TYPE STD\_LOGIC IS ( 'U' ,'X' ,'0' ,'1' ,'Z' ,'W' ,'L' ,'H' ,'\_' );

该数据类型括号中的值自左向右按升序排列，中间用逗号分隔。综合器自动实现枚举类型元素的编码，一般将第一个枚举量（最左边）编码为0，以后的依次加1。编码用位矢量表示 ，位矢量的长度将取所需表达的所有枚举元素的最小值。这为“属性”提供了一个访问位置编号的机制。

（2）整数类型、实数类型

由于标准的整数或实数类型范围太大，可以通过限定范围来约束整数或实数，这样做一般是为了电路设计的特殊要求。语法格式如下：

TYPE 数据类型名 IS 数据类型约束范围；

例如：TYPE subInt IS integer range 0 to 9；

（3）数组类型（array type）

又称为向量，是同一类型的数据组织在一起而形成的新的数据类型，它可以是二维的和多维的。定义：把两个以上的称为数组。语法格式如下：

TYPE 数据类型名 IS ARRAY （范围） OF 元素类型名；

数组中的元素可以是VHDL语言的任何一种数据类型；可以是一维或多维数组，多维数组要用多个范围来描述。

例如：TYPE d IS ARRAY（INTEGER range 0 to 3）of STD\_LOGIC\_VECTOR（7 downto 0）；

多维数组不能生成逻辑电路，EDA工具不能用多维数组进行设计综合，只能用于设计仿真和系统建模。范围一项规定数组下标的类型和范围，默认的下标类型是整型，但也可以使用其它数据类型，这就需要在范围中标明下标的类型。例如：

TYPE count1 IS ARRAY ( STD\_LOGIC '\_' DOWNTO 'U' ) OF INTEGER;

（4）记录类型（record type）

当有多个不同类型的数据对象需要组织在一起，就需要使用记录来创建新的数据类型。语法格式如下：

TYPE 数据类型名 IS RECORD

元素名： 数据类型名；

元素名： 数据类型名；

……

END RECORD；

记录中的每个元素通过字段名访问；记录中的元素可以是相同或不同的类型；适用于仿真，不能用于综合。如要访问记录类型的对象时，可在记录类型对象名后加点“．”，然后再加赋值元素的元素名。

例如：TYPE date IS RECORD

year：INTEGER range 2000 to 2009;

month：INTEGER range 1 to 12；

date：INTERGER range 1 to 31；

END RECORD；

SIGNAL d：date；

BEGIN

d.year <= 2004；

d.month <= 2；

d.date <= 15；

END；

（5）子类型（SYBTYPE）

子类型是已定义的类型或子类型的一个子集，通常用来定义具有一定限制条件的基本类型的数据对象。语法格式如下：

SYBTYPE 子类型名 IS 数据类型名[约束范围]；

例如：bit\_vector 类型定义如下：

type bit\_vector is array (natural range <>) of bit；

如设计中只用到16位，可定义如下子类型：

subtype my\_vector is bit\_vector(0 to 15)；

其中子类型与基（父）类型具有相同的操作符和子程序，可以直接进行赋值操作，但是子类型不能超出范围。

#### 数据类型的转换

由于VHDL是一种强类型语言，不同类型的数据对象必须经过类型转换，才能相互操作。类型转换的方法有类型标记法和函数转换法。

（1）类型标记法

对相互间非常关联的数据类型（如整型、浮点型），可进行直接类型转换，一般只适用于标量类型。使用方法就是直接再要转换的对象前加上类型名称即可，例如：

variable a, b : real;

variable c, d : integer;

……

a:= real（c）;

d:= integer（b）;

（2）函数转换法

通过调用类型转换函数，使相互操作的数据对象的类型一致，从而完成相互操作，用户可以根据需要自己编写一些函数转换数据类型。VHDL中也提供了多种转换函数，帮助转换一些常用类型的数据，以实现正确的赋值操作，在使用时必须首先打开库和相应的程序包。常用的类型转换函数有：

* CONV\_INTEGER ( )

将STD\_LOGIC\_VECTOR类型转换成INTEGER类型，存在于STD\_LOGIC\_UNSIGNED；

* CONV\_STD\_LOGIC\_VECTOR( )

将INTEGER类型、UNSIGNED类型或 SIGNED类型转换成STD\_LOGIC\_VECTOR类型；存在于STD\_LOGIC\_ARITH；

* TO\_BIT ( )

将STD\_LOGIC类型转换成BIT类型，存在于STD\_LOGIC\_1164；

* TO\_BIT\_VECTOR( )

将STD\_LOGIC\_VECTOR类型转换BIT\_VECTOR类型，存在于STD\_LOGIC\_1164。

* TO\_STD\_LOGIC( )

将BIT类型转换成STD\_LOGIC类型，存在于STD\_LOGIC\_1164。

* TO\_STD\_LOGIC\_VECTOR( )

将BIT\_VECTOR类型转换成 STD\_LOGIC\_VECTOR类型，存在于STD\_LOGIC\_1164。

### 4.3.4运算符与操作符

在VHDL语言中共有4类操作符，可以分别进行逻辑运算（logical）、关系运算（relational）、算术运算（Arithmetic）和并置运算（Concatenation）。操作数的类型应该和操作符所要求的类型相一致。运算操作符是有优先级的，例如逻辑运算符NOT，在所有操作符中其优先级最高。表6-3列出了所有操作符的优先次序。

表4.2 操作符的优先级



1． 逻辑运算符

在VHDL语言中逻辑运算符共有6种，包括一元逻辑运算符和二元逻辑运算符。一元逻辑运算符包括：NOT，二元逻辑运算符包括：AND、OR、NAND、NOR、XOR。这6种逻辑运算符可以对“STD\_LOGIC”和“STD\_LOGIC\_VECTOR”等数据进行逻辑运算。必须注意，运算符的左边和右边，以及赋值的信号的数据类型必须是相同的。

当一个语句中存在两个以上的逻辑表达式时，在C语言中运算有自左至右的优先级顺序的规定，而在VHDL语言中，左右没有优先级差别。例如，在下例中，如去掉式中的括号，那么从语法上来说是错误的：

x <=（a AND b）OR（NOT c AND d）；

当然，如果一个逻辑表达式中只有“AND”，“OR”，“XOR”运算符，那么改变运算顺序将不会导致逻辑的改变。此时，括号是可以省略的。例如：

a <= b AND c AND d AND e；

a <= b OR c OR d OR e；

在所有逻辑运算符中NOT的优先级最高。

2．算术运算符

VHDL语言有10种算术运算符，包括一元算术运算符和二元算术运算符。一元算术运算符包括：+（正号）、—（负号）、ABS（求绝对值）；二元算术运算符包括：+ 、— 、\* 、/ 、MOD（求模）、REM（求余）、\*\*（指数运算）。在算术运算中，对于一元运算的操作数（正、负）可以为任何数值类型（整数、实数、物理量）。乘除法的操作数可以同为整数和实数。物理量可以被整数或实数相乘或相除，其结果仍为一个物理量。物理量除以同一类型的物理量即可得到一个整数量。求模和取余的操作数必须是同一整数类型数据。一个指数的运算符的左操作数可以是任意整数或实数，而右操作数应为一整数。

若对“STD\_LOGIC\_VECTOR”进行“＋”（加）、“一”（减）运算时，两边的操作数和赋值的变量位长如不同，则会产生语法错误。另外，“\*”运算符两边的位长相加后的值和要赋值的变量的位长不相同时，同样也会出现语法错误。

3．关系运算符

VHDL语言中有6种关系运算符，包括：= 、/= 、＜、＜= 、＞、＞=。不同的关系运算符对两边的操作数的数据类型有不同的要求，其中等号“＝”和不等号“／＝”可以适用所有类型的数据，其它关系运算符则可使用于整数（INTEGER）和实数（REAL）、位（STD\_LOGIC）等校举类型以及位矢量（STD\_LOGIC\_VECTOR）等数组类型的关系运算。在进行关系运算时，左右两边的操作数的数据类型必须相同，但是位长度不一定相同。当然也有例外的情况，在利用关系运算符对位矢量数据进行比较时，比较过程是从最左边的位开始。自左至右按位进行比较的。在位长不同的情况下，只能按自左至右的比较结果作为关系运算的结果。如下例所示对3位和4位的位矢量进行比较：

SIGNAL a：STD\_LOGIC\_VECTOR（3 DOWNTO 0）；

SIGNAL b：STD\_LOGIC\_VECTOR（2 DOWNTO 0）；

a <= “1010”； －－10

b <= “111”； －－7

IF（ a ＞ b） THEN

┇

ELSE

┇

上例中a的值为10，而b的值为7，a应该比b大。但是，由于位矢量是从左至右按位比较的，当比较到次高位时，a的次高位为“0”而b的次高位为“1”，故比较结果b比a大。这样的比较结果显然是不符合实际情况的。

为了能使位矢量进行关系运算，在程序包“STD\_LOGIC\_UNSIGNED”中对“STD\_LOGIC\_VECTOR”关系运算重新作了定义，使其可以正确的进行关系运算。注意在用时必须首先说明调用该程序包。

4．并置运算符

并置运算符“＆”（连接），用于将多个位连接成为位矢量。例如，将4个位用并置运算符”＆”连接起来就可以构成一个具有4位长度的位矢量。两个4位的位矢量用并置运算符“＆”连接起来就可以构成8位长度的位矢量。例如：DBUS＜=D0&D1&D2&D3，即DBUS＜=（D0,D1,D2,D3）。

### 4.3.5属性

VHDL语言为用户提供了多种能反映和影响硬件行为的属性，包括设计实体、结构体、类型、信号等项目的指定特性，其实属性相当于为获取硬件设计中的一些有关信息（各类项目特性）而定义的内部函数。属性提供了描述特定对象的多个侧面值的手段，信号属性在检测信号变化和建立详细的时域模型时非常重要。VHDL的属性可分为数值类、函数类、信号类、类型类和范围类等属性。语法格式如下：

Object ′Attributes

1．数值类属性

数值类属性用于返回数组、块或一般数据的有关值，如边界、数组长度等。例如：返回数组长度、数据类型的上下界等。

常用数据类型的数值类属性：

Object‘left：返回一个数据类型或子类型最左边的值。

Object ‘right：返回一个数据类型或子类型最右边的值。

Object ‘high：返回一个数据类型或子类型的最大值。

Object ‘low：返回一个数据类型或子类型的最小值。

对于数组的数值属性，返回数组的长度值。其格式为Object’length。

2、函数类属性

函数类属性是指属性以函数的形式返回有关数据类型、数组、信号的相关信息，一般使用时以函数表达式的形式出现，属性根据输入的值去执行函数，返回一个相应的值。该返回值可能是数组区间的某一个值，或是枚举数据的位置序号等。函数类属性分三类：数据类型属性函数、数组类型属性函数和信号属性函数。

数据类型的属性函数利用数组属性获得数组的信息，主要用来得到数据类型的各种相关信息，有如下几个主要属性：

Object’SUCC（X）； 获取X的下一个值

Object’PRED（X）； 获取X的前一个值

Object’LEFTOF（X）； 获取X的左边值

Object’RIGHTOF（X）； 获取X的右边值

其中，“Object”为数据类型名，X为其中的一个元素。

例如定义一个类型TYPE Week IS (sun, mon, tue, wed, thu, fri, sat);

其中一些属性结果如下：

Week’SUCC(mon) --得tue

Week’PRED(mon) --得sun

Week’LEFTOF(mon) --得sun

Week’RIGHTOF(mon) --得tue

数组的属性函数主要用来得到数组的信息，共有4种属性：

Object’LEFT（n）； --获取索引号为n的区间左端边界值

Object’RIGHT（n）； --获取索引号为n的区间右端边界值

Object’HIGH（n）； --获取索引号为n的区间高端边界值

Object’LOW（n）； --获取索引号为n的区间低端边界值

其中，“Object”为数组名；n为多维数组中所定义的多维区间的序号，默认值n=1，表示对一维空间进行操作。例如一个数组定义如下：

TYPE matrix IS ARRAY(0 TO 7, 15 DOWNTO 0) OF std\_logic；

其属性结果如下：

i <= matrix ’ left (1); -- i = 0;

i <= matrix ’ right (1); -- i = 7;

i <= matrix ’ low (1); -- i = 0;

i <= matrix ’ high (2); -- i = 15;

i <= matrix ’ low (2); -- i = 0;

信号的属性函数主要用来得到信号的各种行为功能信息：包括信号值的变化、信号变化后经过的时间、变化前的信号值等。共有5种属性：

Object’EVENT； 信号的值是否变化，如是，则返回为“真”

Object’ACTIVE； 信号是否活跃，如是，则返回为“真”

Object’LAST\_EVENT； 从最近一次事件到现在经过的时间，返回一个时间值

Objectt’LAST\_VALUE； 信号变化前的取值，并将该历史值返回

Object’LAST\_ACTIVE； 从最近一次活跃到现在经过的时间，返回一个时间值

其中信号的事件（Event）和活跃（Active）是这两个属性是经常用到的，两个不同的概念，必须严格区分。信号的活跃定义为信号值的任何变化，信号值由1变为0是一个活跃，而从1变为1也是一个活跃，唯一的准则是发生了事情，这种情况被称为一个事项处理（Transaction）。然而，信号的事件则要求信号值发生变化。信号值从1变为0是一个事件，但从1变为1虽是一个活跃却不是一个事件。因此事件EVENT可以用来检测脉冲信号的正跳变或负跳变边沿，也可以检查信号是否刚发生变化并且正处于某一个电平值。

比如监测一个正边沿可以使用如下语句：

(clk=’1’) AND (clk’EVENT)

3、信号类属性

信号类属性的作用对象是信号，根据这个信号去建立一个新的信号，因此其返回值也是一个信号。共有四种信号类属性，分别是：

DELAYED (time)：即延时，time为时间表达式，将产生一个特别的延迟信号，与主信号类型相同，该信号使主信号按time确定的时间产生附加的延迟。如：a’DELAYED (5ns)即信号a延时5ns。

STABLE (time)：用于监测信号在规定时间内的稳定性。当所加属性的信号在时间time内没有发生变化，则该属性的结果为“TRUE”。该属性中当t=0时可以得到与属性’EVENT相反的值。

QUIET（time）：用于监测信号在规定时间内的是否没有事件。若受它作用的信号在time所规定的时间内没有发生事情或事件（Active或Event），则返回true，否则返回false。典型应用是用来对中断优先处理机制进行建模。

TRANSACTION：用于检测信号的Active 或Event。当Active或Event发生时，该属性的值也将发生改变。

## 基本描述语句

在用VHDL语言描述系统硬件行为时，按语句实际执行顺序对其进行分类，可以将VHDL语句分为顺序（Sequential）描述语句和并行（Concurrent）描述语句。顺序描述语句是指语句是按照语句书写的顺序自上至下，一个语句一个语句地执行的，IF语句、LOOP语句等都属于此类顺序描述语句，他们一般只会出现在进程和子程序中；并行描述语句的执行顺序与书写顺序无关，他们是并行执行的，例如在一个结构体内可以有几个进程语句同时存在，各进程语句是并行执行的。使用VHDL语言描述实际的硬件系统的行为，这两类描述语句都是必须的。

### 4.4.1顺序（Sequential）描述语句

顺序描述语句顾名思义就是顺序执行的语句，每一条语句的执行（指仿真执行）都是按语句排列的次序执行的。顺序描述语句只能出现在进程或子程序中，语句中所涉及到的系统行为有时序流控制、条件和迭代等；语句的功能操作有算术、逻辑运算、信号和变量的赋值、子程序调用等。顺序描述语句像在一般高级语言中一样，其语句是按出现的次序加以执行的。在VHDL语言中顺序描述语句有以下几种：信号代入语句、变量赋值语句、WAIT语句、IF语句、CASE语句、LOOP语句、NEXT语句、EXIT语句、RETURN语句和NULL语句。

#### 信号代入语句

信号带入语句就是信号赋值语句，其语法格式为：

信号量 <= 波形{，波形}；

波形::= 值表达式 [**after** 时间表达式]；

该语句的意义是将右边表达式的值赋予左边信号量。例如a＜=b就是将b的当前值赋予目的信号量a。在这里代入语句的符号“＜=”和关系运算的小于等于符“＜=”相同，应根据上下文的含义和说明正确判别其意义，并且信号代入语句符号两边的类型和长度应该是一致的。

信号代入也可以是有时间含义的一个波形，例如：

B <= A after 5 ns;

Clock <= ‘0’,

‘1’ after 5 ns,

‘0’ after 10 ns,

‘1’ after 50 ns;

前面在介绍信号的时候就说过信号的赋值是有延时的，就算不写明延时时间，其延时值为无穷小量，近似为0。这样就使得信号赋值语句即使在顺序执行的过程中也是最后统一赋值的，如下例所示的赋值语句：

A <= B after 5ns;

B <= A after 5ns;

信号A的当前值5ns后会被赋给信号B，同样信号B的当前值5ns后会被赋给信号A，这样5ns后信号A、B的值会互换，如下图4.6所示：

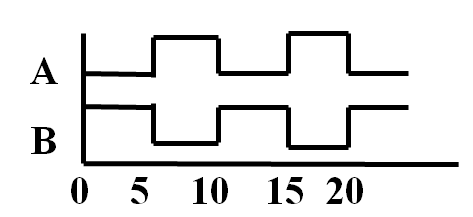


图4.6 信号赋值

当信号代入语句中没有指定延迟的时候，如下例：

B <= A;

A <= B;

由于信号赋值是有延时的，这个语句相当于延时了δfs，δ是一个无穷小量，则该语句相当于：

B <= A after δfs;

A <= B after δfs;

这样过了δfs后，A、B的值就会互换，而跟语句书写的顺序没有关系了，当然如果A、B不是信号而是变量后，执行上面的语句会导致A、B的值相等都等于最开始B的值，这样就跟书写顺序有关了。

#### 变量赋值语句

变量赋值语句就是对变量进行赋值，其语法格式为：

变量 := 表达式；

变量的值将由表达式所表达的新值替代，但两者的类型必须相同。变量只存在于进程或子程序中，无法传递在进程外使用。变量的类型、范围及初值应事先说明，右边的表达式可以是变量、信号或字符组成。例如：

a := 2；

b := c+d；

变量的赋值是立即发生的，没有延迟，其执行顺序就是书写顺序。

#### WAIT语句

WAIT语句是进程的同步语句，在进程内是顺序执行的，当执行到WAIT等待语句时，进程将被挂起，直到满足此语句的结束挂起条件后，将重新开始执行进程中的程序。WAIT语句有以下四种形式：

* WAIT

无限等待，表示永远挂起。

* WAIT ON 敏感信号表；

敏感信号量变化，进程启动

敏感信号等待语句，即敏感信号一旦发生变化将结束挂起，再次启动进程。VHDL规定，已列出敏感信号表的进程中不能使用任何形式的WAIT语句。一般， WAIT语句可用于进程中的任何地方。

* WAIT UNTIL 条件表达式；

条件等待，表达式成立时，进程将会启动。被此语句挂起的进程需满足如下两个条件才能结束挂起状态，重新启动：

* + 条件表达式中所含信号发生了变化；
  + 此信号改变后，且满足WAIT语句所设的条件。

两条件缺一不可，且必须依照以上顺序完成。

* WAIT FOR 时间表达式；

延时等待，延迟时间表达式所规定的时间到后，进程再次，此语句不可综合。

WAIT ON语句相当于进程的敏感信号表，如下例是一个与门电路，分别使用WAIT语句和进程敏感信号表，二者是等价的。

PROCESS

BEGIN

Y <= a AND b;

WAIT ON a,b;

END PROCESS；

END example；

PROCESS（a,b）

BEGIN

Y <= a AND b;

END PROCESS；

#### IF语句

IF语句是根据所指定的条件来确定执行哪些语句的。IF语句至少有一个条件句，条件句必须由布尔表达式构成；根据条件句产生的判断结果为true或false，有条件地选择执行其后的顺序语句；条件表达式中只能使用关系运算操作（=、/=、<、>、 <=、>= ）及逻辑运算操作的组合表达式。其书写格式通常可以分成3种类型。

* IF语句的门闩控制，其语法格式为：

IF条件 THEN

顺序处理语句；

END IF；

当程序执行到该IF语句时，就要判断IF语句所指定的条件是否成立。如果条件成立，则IF语句所包含的顺序处理语句将被执行；如果条件不成立，程序将跳过IF语句所包含的顺序处理语句，而向下执行IF语句后继的语句。这里的条件起到门闩的控制作用，如下例所示。

IF（a=’1’）THEN

c<=b;

END IF;

当信号a =’1’时，信号b任何值的变化都将被赋给信号c，也就是说，此时c值与b值是永远相等的；当a≠’1’时，c<= b语句不被执行，c将维持原值不变，而不管b值发生什么变化。如果进程中只有这一个语句，这种描述经逻辑综合，实际上可以生成一个D触发器。如下例所示：

PROCESS(a)

BEGIN

IF（a’EVENT AND a=’1’）THEN

c<= b;

END IF;

END PROCESS;

* IF语句的二选择控制

这种语句为IF语句提供了两种结果选择，其语法格式为：

IF条件 THEN

顺序处理语句；

ELSE

顺序处理语句；

END IF；

当IF语句的条件满足时，将执行THEN和ELSE之间的顺序处理语句；当条件不满足时，将执行ELSE和END IF之间的顺序处理语句，也就是说，用条件来选择两段不同程序的执行路径。

这种描述的典型逻辑电路实例是二选一电路。例如，二选一电路的输入为a和b，选择控制端为sel，输出端为c。那么该电路行为的程序如下例所示

PROCESS（a，b，sel）

BEGIN

IF（sel=’1’） THEN

c <= a；

ELSE

c <= b；

END IF；

END PROCESS；

这是一个组合逻辑电路，可以看出如果IF语句中没有ELSE后面的部分，则隐含着要生成触发器或者锁存器；如果IF语句带有ELSE部分，则隐含指明生成组合逻辑。

* IF语句的多选择控制

IF语句的多选择控制的语法格式为：

IF条件 THEN

顺序处理语句；

ELSIF 条件 THEN

顺序处理语句；

……

ELSIF 条件 THEN

顺序处理语句；

ELSE

顺序处理语句；

END IF；

在这种多选择控制的IF语句中，设置了多个条件，当满足所设置的多个条件之一时，就执行该条件后跟的顺序处理语句。如果所有设置的条件都不满足，则执行ELSE和END IF之间的顺序处理语句。这种描述的典型逻辑电路实例是多选一电路，如下例就是一个4选1选择器电路：

PROCESS（input，sel）

BEGIN

IF（sel=”00”）THEN

y <= input（0）;

ELSIF（sel=”01”）THEN

y <= input（1）；

ELSIF（sel=”10”）THEN

y <= input（2）;

ELSE

Y <= input（3）;

END IF;

END PROCESS;

IF语句不仅可以用于选择器的设计，而且还可以用于比较器、译码器等凡是可以进行条件控制的逻辑电路设计。

#### CASE语句

CASE语句主要用来描述总线或编码、译码的行为，从许多不同语句的序列中选择其中之一执行之。虽然IF语句也有类似的功能，但是CASE语句的可读性比IF语句要强得多，程序的读者很容易找出条件式和动作的对应关系。CASE语句的语法格式如下所示：

CASE表达式 IS

WHEN条件表达式 => 顺序处理语句；

WHEN条件表达式 => 顺序处理语句；

……

END CASE；

上面的条件表达式可以有如下4种不同的表示形式：

WHEN值 => 顺序处理语句；

WHEN值∣值∣值∣…∣值 => 顺序处理语句；

WHEN值TO值 => 顺序处理语句；

WHEN OTHERS => 顺序处理语句；

当CASE和IS之间的表达式的取值满足指定的条件表达式的值时，程序将执行后卖面的由符号 => 所指的顺序处理语句。条件表达式的值可以是一个值；或者是多个值的“或”关系；或者是一个取值范围；或者表示其它所有的缺省值。CASE语句是无序的，所有表达式的值都是并行处理的；所有表达式的值都必须穷举，且不能重复，不能穷尽的值用 OTHERS 表示 ；至少要包含一个条件语句；不支持对任意项输入的条件表达式，即条件表达式的值不能含有‘X’。

下例是一个用CASE语句描述的4选1选择器：

PROCESS（a，b，ic，if，iZ，i3）

BEGIN

sel <= ‘0’;

IF（a＝’1’）THEN

sel <= sel + 1;

END IF;

IF（b＝’1’）THEN

sel <= sel + 2;

END IF;

CASE sel IS

WHEN 0 => q <= i0;

WHEN 1 => q <= i1;

WHEN 2 => q <= i2;

WHEN others=> q <= i3;

END CASE；

END PROCESS；

从上面的例子可以看出，选择器的行为描述不仅可以用IF语句，也可以用CASE语句。两者的区别在于：在IF语句中，先处理最起始的条件，如果不满足，再处理下一个条件；而在CASE语句中，没有值的顺序号，所有值是并行处理的，因此条件值不能重复使用。通常在CASE语句中，WHEN语句可以颠倒次序而不至于发生错误，而在IF语句中，颠倒条件判别的次序往往会使综合的逻辑功能发生变化。

#### LOOP语句

LOOP语句使程序能进行有规则的循环，循环次数受迭代算法控制。常用来描述迭代电路的行为，通常使用在循环语句中。一个LOOP语句中所包含的一组顺序语句要重复执行若干次。LOOP语句有两种重复方式：FOR模式、WHILE模式。

* FOR模式

语法格式如下：

［标号］：FOR 循环变量 IN 范围 LOOP

顺序处理语句

END LOOP［标号］；

循环变量的值在每次循环中都会发生变化，是一个临时变量，属于局部变量；循环变量在信号说明、变量说明中不能使用，只能作为赋值源，不能被赋值，它由LOOP语句自动定义；范围表示循环变量在循环过程中的取值范围；只要循环变量还在范围内，循环将一直继续下去；循环变量是一个整数变量，不用事先说明。下面是一个奇偶校验电路的示例：

PROCESS(a)

VARIABLE tmp : STD\_LOGIC;

BEGIN

tmp:='0';

FOR i IN 0 to 7 LOOP

tmp:=tmp XOR a(i);

END LOOP;

y<=tmp;

END PROCESS;

* WHILE模式

语法格式如下：

［标号］：WHILE 条件 LOOP

顺序处理语句

END LOOP［标号］；

当条件为“真”（true）时，执行紧跟着的顺序语句；反之，如果条件为“假”（false）就结束循环。下面是用WHILE描述奇偶校验电路。

PROCESS(a)

VARIABLE tmp : STD\_LOGIC;

VARIABLE i : INTEGER;

BEGIN

tmp:='0';

i:=0;

WHILE(i<8) LOOP

tmp:=tmp XOR a(i);

i:=i+1;

END LOOP;

y<=tmp;

END PROCESS;

#### NEXT语句

使用在LOOP语句中，用来跳出本次循环，其语法格式为：

NEXT［标号］［WHEN条件］；

NEXT语句执行时将停止本次循环，而转入下一次新的循环。NEXT后跟的“标号”表明下一次循环的起始位置，而“WHEN”后面是NEXT语句执行的条件。如果NEXT语句后面既无“标号”也无“WHEN”说明，那么只要执行到该语句就立即无条件地跳出本次循环，从 LOOP语句的起始位置进入下一次循环。如下例。

L1: WHILE i<10 LOOP

L2: WHILE j<10 LOOP

NEXT L1 WHEN i = j;

END LOOP L2;

END LOOP L1;

上例中，当 i = j 时， NEXT语句被执行，程序跳出内循环，下一次从外循环开始执行。

#### EXIT语句

在LOOP语句中，用EXIT语句跳出并结束整个循环状态（而不是仅跳出本次循环），继续执行LOOP语句后继的语句。EXIT语句的语法格式为：

NEXT［标号］［WHEN条件］；

当 “WHEN条件”为真时，跳出LOOP至程序标号处。如果EXIT后面没有跟“标号”和“ WHEN条件”，则程序执行到该语句时就无条件地从LOOP语句中跳出，结束循环状态，继续执行LOOP语句后继的语句。

EXIT语句主要用于控制循环，它提供了一个处理保护、出错和警告等状态的简便方法。

#### RETURN语句

RETURN语句是一段子程序结束后，返回主程序的控制语句，语法格式为：

RETURN [条件表达式]；

RETURN用于函数和过程体内，用来结束函数或过程体的执行。

用于过程的RETURN语句只能结束过程，并不返回任何值。用于函数中的RETURN语句必须有条件表达式，并且必须返回一个值。每一个函数必须至少包含一个返回语句，也可以拥有多个返回语句，但在函数调用时只有其中一个返回语句可以将值带出。

#### NULL语句

NULL语句表示没有操作，即空操作，语法格式为：

NULL；

NULL语句不完成任何操作，其作用只是使程序运行流程跨入下一步语句的执行。NULL语句常用于CASE语句中，为满足所有可能的条件，利用NULL来表示所余的不用条件下的操作行为。

### 4.4.2并行（Concurrent）描述语句

由于硬件描述语言所描述的实际系统，其许多操作是并行的，所以在对系统进行仿真时，这些系统中的元件在定义的仿真时刻应该是并行工作的。并行语句就是用来表示这种并行行为的，并发语句结构是最具VHDL特色的。

在VHDL中，并发语句有多种语句格式，它们在结构体中的执行是同步进行的，或者说是并行运行的；其执行方式与书写顺序无关。在执行中，并发语句之间可以有信息往来；也可以是互为独立、互不相关、异步运行（如多时钟情况）。但每一并发语句内部的语句运行方式可以不同，即有并行执行方式（如块语句）和顺序执行方式（如进程语句）。

并发描述语句主要包括进程语句、BLOCK语句、并发代入语句、条件代入语句、选择信号赋值语句、元件例化语句等。

#### 进程语句

PROCESS语句是VHDL中最基本的一种并发处理语句，在一个结构体中多个PROCESS语句可以同时并发运行。因此，PROCESS语句是VHDL中描述硬件系统并发行为的最常用、最基本的语句。进程可以与其它进程并发执行，并可存取结构体或实体所定义的信号；进程中的所有语句都是顺序执行的；进程中必须包含一个显式的敏感信号表或者包含一个WAIT语句；进程之间的通信是通过信号传递来实现的。

并行信号赋值语句、条件信号赋值语句等并行执行语句，实质上是一种进程的缩写形式，它们仍可以归属于进程语句。

进程（PROCESS）语句的语法格式如下：

[进程名：] PROCESS [（敏感信号1，敏感信号2，…）]

[变量说明语句；]

BEGIN

顺序执行语句；

END PROCESS；

在VHDL程序仿真的过程中，进程PROCESS只有两种可能的状态：激活与挂起。进程一般带有几个敏感信号（若没有的话，进程内必须有wait语句），他们是PROCESS的输入信号，控制着进程的状态。当程序开始时(初始化阶段)，所有的进程均被激活并执行，当遇到wait语句进程会挂起；当等待语句条件满足时，进程再次被激活，并从等待语句处接着运行直至最后。若没有wait语句，则进程会一直运行到最后一句。执行完最后一条语句后，返回第一条语句，若没有敏感信号，则从头开始执行；若存在敏感信号，进程就会挂起，直到敏感信号发生变化，进程就会重新激活，并从头开始执行。当进程存在敏感信号，则进程内部不允许有wait语句。

下图4.7是一个进程执行的过程：

图4.7 进程执行过程

下面以一个例子来说明敏感信号对进程执行的影响。

Pro1:PROCESS（A,B）

BEGIN

q:= A AND B；

END PROCESS；

进程pro1中，敏感信号为A、B，则当A或B发生变化时，q都会重新被计算，如图4.8所示：

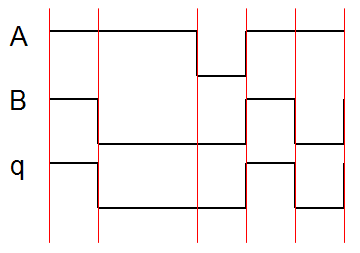


图4.8 敏感信号A B

例外一个进程Pro2如下：

Pro1:PROCESS（A,B）

BEGIN

q:= A AND B；

END PROCESS；

进程pro1中，敏感信号只有为A，只有当A发生变化时，q才会重新被计算，B的变化对q没有影响，如图4.9所示：

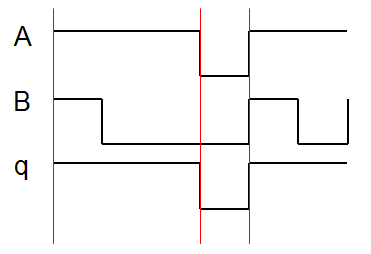


图4.9 敏感信号A

#### 并发代入语句

并发代入语句就是信号赋值语句，它可以在进程内部使用，此时它作为顺序语句形式出现；它也可以在结构体的进程之外使用，此时它作为并行语句形式出现。一个并行信号赋值语句实际上是一个进程的编写。例如：

ARCHITECTURE behav OF ent IS

BEGIN

Output <= a AND b；

END behav；

它等效于

ARCHITECTURE behave OF ent IS

BEGIN

PROCESS（a，b）

BEGIN

Output <= a AND b；

END PROCESS；

END behav；

当赋值符号“<=”右边的信号值发生任何变化时，赋值操作就会被激活，新的值将开始赋予赋值符号“<=”左边的信号。等效成进程语句描述来看，在PROCESS语句的敏感信号量表a、b，任何一个敏感信号量发生变化，进程将得到启动，赋值语句将被执行。因此并行信号赋值语句和进程语句在这种情况下确实是等效的。

并发代入语句在仿真时刻同时运行，它表征了各个独立器件的各自的独立操作。例如：

a <= b + c；

d <= e \* f；

第一个语句描述了一个加法器的行为，而第二个语句描述了一个乘法器的行为。在实际硬件系统中，加法器和乘法器是独立并行工作的。现在第一个语句和第二个语句都是并行信号赋值语句，在仿真时刻，两个语句是并行处理的，从而真实地模拟了实际硬件系统中的加法器和乘法器的工作。

#### 条件信号赋值语句

条件信号赋值语句它可以根据不同条件将不同的多个表达式之一的值赋值信号量，其语法格式为：

目标信号 <= 表达式1 WHEN 条件1 ELSE

表达式2 WHEN 条件2 ELSE

……

表达式n；

条件信号代入语句与进程中的IF语句相同，具有顺序性，但ELSE不能省略。在每个表达式后面都跟有用“WHEN”所指定的条件，如果满足该条件，则该表达式的值赋值目的信号量；如果不满足条件，则再判别下一个表达式所指定的条件。最后一个表达式可以不跟条件表达式。执行该语句时，每一赋值条件是按书写的先后关系逐项测定的，一旦发现条件为“true”，立即将表达式的值赋予目标信号。由于条件测试的顺序性，条件信号代入语句的赋值具有优先级别，其中第1子句优先级别最高，依次类推。

#### 选择信号赋值语句

选择信号赋值语句类似于CASE语句，它对表达式进行测试，当表达式取值不同时，将使不同的值赋值目的信号量。选择信号赋值语句的语法格式如下：

WITH 条件表达式 SELECT

目标信号 <= 表达式1 WHEN 条件1，

表达式2 WHEN 条件2，

……

表达式n WHEN 条件n；

选择信号语句不能在进程中应用，每当选择表达式的值发生变化，便启动该语句对各子句的选择值（条件）进行测试对比，当发现有满足条件的子句时，就将此子句表达式的值赋予目标信号。与CASE语句相类似，该语句对子句条件选择值具有同等地位，没有顺序性，因此不允许有条件重叠现象，也不允许存在条件涵盖不全的情况。

#### 元件例化语句

元件例化语句使我们能够将原来设计好的VHDL功能模块当作元件一样用在其它VHDL文件中。也就是引入一种连接关系，利用特定的语句将元件与当前的设计实体中的指定端口相连接，从而成为当前设计实体的一个新的低一级的设计层次。有了元件例化，使得VHDL设计实体构成能够形成自上而下层次化的设计。例化的过程中需要元件的说明和例化两个部分。元件说明语句的语法格式为：

COMPONENT 元件名

[GENERIC 说明；]

PORT 说明；

END COMPONENT

元件例化语句的语法格式为：

元件名 PORT MAP（信号，…）；

例化的过程中需要建立端口的信号映射关系，主要的方法有两种：

* 位置映射方法 ：即在元件例化语句中，实例元件的PORT MAP（ ）中的实际信号书写顺序位置应与元件说明语句中的端口说明中的信号书写顺序位置一一对应。
* 名称映射方法 ：这种方法与信号的书写顺序位置无关，使用”=>”来说明影射关系。

PORT MAP（形参 => 实参）；

其中，实参是设计中连接到端口的实际信号；形参是指元件的对外接口信号。例如：

u0: xnor2 PORT MAP (in1=>a(0), in2=>b(0), out=>s(0))；

下面举一个全加器的实例，如图4.10所示：

图4.10 全加器

由上图可知，一个全加器由两个半加器和一个或门组成，半加器的实现如下：

ENTITY h\_adder IS

PORT(a, b : IN STD\_LOGIC;

s, c : OUT STD\_LOGIC);

END h\_adder;

ARCHITECTURE rtl OF h\_adder IS

BEGIN

c<=a AND b;

s<=a XOR b;

END rtl;

使用元件例化上面的半加器组成全加器的实现如下：

ENTITY f\_adder IS

PORT(a, b,ci : IN STD\_LOGIC;

s, co : OUT STD\_LOGIC);

END f\_adder;

ARCHITECTURE arc OF f\_adder IS

COMPONENT h\_adder

PORT(a, b : IN STD\_LOGIC;

s, c : OUT STD\_LOGIC);

END COMPONENT;

SIGNAL s1,c1,c2:STD\_LOGIC;

BEGIN

u1:h\_adder PORT MAP(a,b,s1,c1);

u2:h\_adder PORT MAP(s1,ci,s,c2);

co<=c1 OR c2;

END arc;

## 描述方式

VHDL的结构体描述方式主要有三种：行为描述方式、寄存器传输（RTL）描述方式、结构化描述方式。这三种描述方式体现了对硬件的不同理解方式。行为描述描述该设计单元的功能，即该硬件能做什么；数据流（寄存器传输RTL）描述描述了数据的传输和变换过程；结构化描述描述了该设计单元的硬件结构，即该硬件是如何构成的。

### 4.5.1 行为描述

行为描述主要使用函数、过程、进程语句，以算法形式描述数据的变换和传送。他是对系统数学模型的抽象描述，其抽象程度比寄存器传输描述方式和结构化描述方式的更高。因此，行为描述有时被称为高级描述。在行为描述方式的程序中，大量采用了算术运算、关系运算、惯性延时、传输延时等难于进行逻辑综合和不能进行逻辑综合的VHDL语句，一般来说，采用行为描述方式的VHDL程序主要用于系统层数学模型的仿真或者系统工作原理的仿真。

下面以一个一位比较器为例说明其描述方式，图4.11是一个一位比较器的示意图：

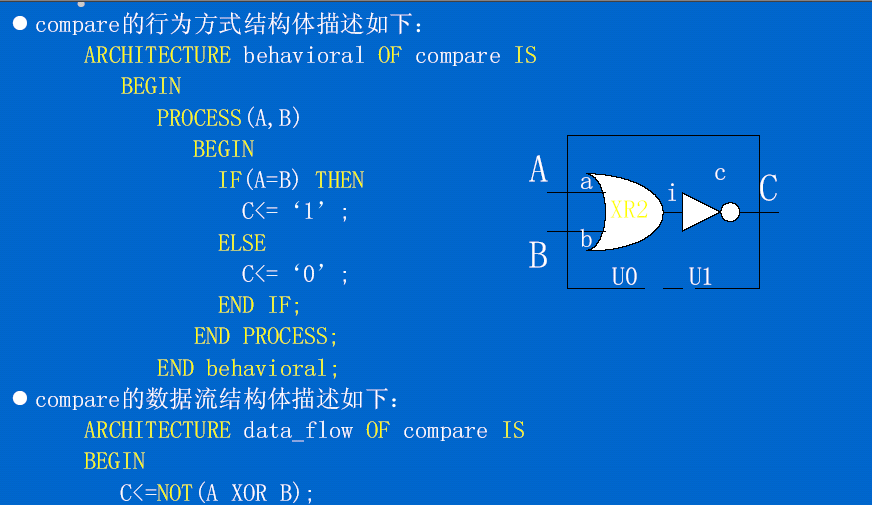


图4.11 与非门

使用行为描述如下：

ARCHTECTURE behavioral OF compare IS

BEGIN

PROCESS(A,B)

BEGIN

IF(A=B) THEN

C<='1';

ELSE

C<='0';

END IF;

END PROCESS;

END behavioral;

从上面的描述可以看出，它描述了一位比较器的数学算法，即他的行为方式。

行为描述只描述设计电路的功能或电路的行为，而没有指明或实现这些行为的硬件结构；或者说行为级描述只表示输入输出之间的转换行为，它不包含任何结构信息。行为级描述通常指顺序语句描述，即含有进程的非结构化的逻辑描述。

行为描述是VHDL最重要的描述方式，它使得硬件的描述不再依赖于其结构而描述。

### 4.5.2 寄存器传输（RTL）描述方式

数据流（寄存器传输RTL）描述主要使用并行的信号赋值语句，即显示了该设计单元的行为，也隐含表示了该设计单元的结构。它是一种以规定设计的各种寄存器形式为特征的描述方法，可以采用寄存器硬件一一对应的直接描述，或者采用寄存器之间的功能描述。

RTL描述方式是建立在并行信号赋值语句描述的基础上，描述了数据流的运动路径、运动方向和运动结果，故又称为数据流描述。它是真正可以进行逻辑综合的描述方式。

一位比较器使用这种描述方式的描述如下：

ARCHTECTURE data\_flow OF compare IS

BEGIN

C<=NOT(A XOR B);

END data\_flow;

从上面的描述可以看出，数据A、B是如何流动到C的，同时也可以看出一位比较器由一个异或门接着一个非门组成。

由于这种描述方式是依据硬件存在的，当硬件无法实现时，有些描述是不合理的，因此它有一些限制：

* 禁止在一个进程中存在两个边沿检测的寄存器描述

如下例中对clk1和clk2都检测边沿是不允许的：

PROCESS(clk1,clk2)

BEGIN

IF (clk1 'EVENT AND clk1='1') THEN

y<=a;

END IF;

IF (clk2 'EVENT AND clk2='1') THEN

z<=b;

END IF;

END PROCESS;

* 禁止使用检测边沿的IF语句中的ELSE项

如下例ELSE是不允许的，实际中不存在这样的硬件电路：

PROCESS(clk)

BEGIN

IF (clk'EVENT AND clk='1') THEN

y<=a;

ELSE

y<=b;

END IF;

END PROCESS;

### 4.5.3 结构化描述

结构化描述主要使用配置指定语句及元件例化语句来描述元件的类型及元件的互联关系。它描述了设计单元的硬件结构，即该硬件是如何构成的。结构描述方式通常采用元件例化语句编写程序。

一位比较器使用这种描述方式的描述如下：

ARCHTECTURE structral OF compare IS

SIGNAL I : BIT;

COMPONENT xor

PORT(A,B: IN BIT;

I : OUT BIT);

END COMPONENT;

COMPONENT inv

PORT(I: IN BIT;

C : OUT BIT);

END COMPONENT;

BEGIN

U0: xor PORT MAP(A,B,I);

U1: inv PORT MAP(I,C);

END structral;

在多层次的设计中，常采用结构描述方式在高层次的设计模块中调用低层次的设计模块，或者直接用门电路设计单元构造一个复杂的逻辑电路。编写结构描述程序的直观方法，可模仿逻辑图的绘制方法：即用框图来表示当前设计单元的组成和内部联结关系，而后对照该框图编制出所需的VHDL程序。

# 第5章 开发环境

# 5.1 概述

一般生产FPGA的厂商都会提供相应的开发软件，Xilinx公司也提供了自己的开发工具，由早期的Foundation系列逐步发展到目前的ISE系列，集成了很多工具软件，这里我们只介绍FPGA开发需要的功能。

ISE具有界面友好、操作简单的特点，再加上Xilinx的FPGA芯片占有很大的市场，使其成为非常通用的FPGA工具软件。ISE作为高效的EDA设计工具集合，与第三方软件扬长补短，使软件功能越来越强大，为用户提供了更加丰富的Xilinx平台。ISE的主要功能包括设计输入、综合、仿真、实现和下载，涵盖了FPGA开发的全过程，从功能上讲，其工作流程无需借助任何第三方EDA软件。图5.1为FGPA开发的一般过程：

图5.1 FPGA开发过程

图5.1是一个比较详细的过程，整体来看，整个过程可以分为以下几个部分：

* 设计输入：选择好设计方案后，使用硬件描述语言或者原理图方式创建设计输入；ISE提供的设计输入工具包括用于HDL代码输入和查看报告的ISE文本编辑器（The ISE Text Editor），用于原理图编辑的工具ECS（The Engineering Capture System），用于生成IP Core的Core Generator，用于状态机设计的StateCAD以及用于约束文件编辑的Constraint Editor等。
* 综合：将设计输入转换成使用基本的逻辑单元组成的逻辑网络结构。ISE的综合工具不但包含了Xilinx自身提供的综合工具XST，同时还可以内嵌Mentor Graphics公司的LeonardoSpectrum和Synplicity公司的Synplify，实现无缝链接。
* 仿真：是对设计进行软件的模拟仿真，分为前仿真和后仿真，前者只是对功能进行验证，而后者会将时序信息加入。ISE本身自带了一个具有图形化波形编辑功能的仿真工具HDL Bencher，同时又提供了使用Model Tech公司的Modelsim进行仿真的接口。
* 实现：将设计映射到器件结构上，进行布局布线，此功能包括了翻译、映射、布局布线等，还具备时序分析、管脚指定以及增量设计等高级功能。
* 配置：又称为下载，将布局布线后的设计文件转换为位流文件，并将程序烧写到FPGA芯片中去。

## 5.2 ISE用户界面

ISE12.3的界面如图5.26所示，由上到下主要分为标题栏、菜单栏、工具栏、管理区、编辑区、信息显示等几部分。

标题栏

菜单栏

工具栏

管理区

编辑区

信息显示

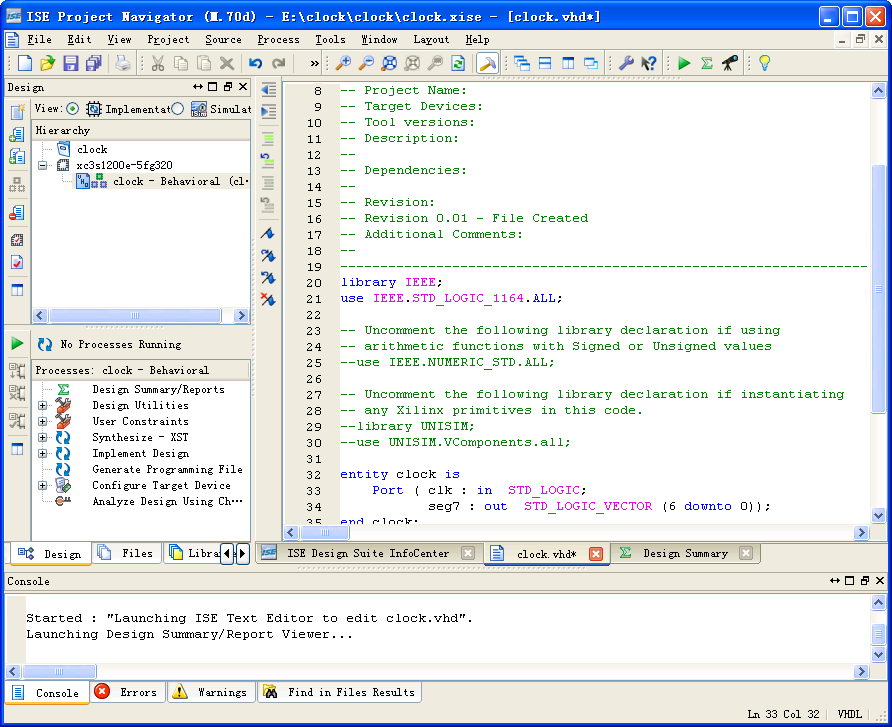


图5.2 ISE的用户界面

标题栏：主要显示当前工程的名称和当前打开的文件名称。

菜单栏：主要包括文件（File）、编辑（Edit）、视图（View）、工程（Project）、源文件（Source）、操作（Process）、窗口（Window）、布局（Layout）和帮助（Help）等8个下拉菜单。其使用方法和常用的Windows软件类似。

工具栏：主要包含了常用命令的快捷按钮。灵活运用工具栏可以极大地方便用户在ISE中的操作。在工程管理中，此工具栏的运用极为频繁。

管理区：提供了工程以及其相关文件的显示和管理功能，主要包括开始（Start），设计（Design），文件（File）和库（Library）四个视图。其中设计视图比较常用，显示了工程名、指定的芯片和设计相关文档等，并显示了设计的层次关系。在设计视图的每一个文件都有一个相关的图标，这个图标显示的是文件的类型（HDL文件、原理图、IP核和文本文件）。‘+’表示该设计文件包含了更低层次的设计模块。设计视图有两个分视图，分别是实现（Implementation）和模拟（Simulation）,选择不同的设计层次和视图时，下方的操作区会出现不同的操作选项，方便用户操作。相关操作和FPGA设计流程紧密相关，包括设计输入、综合、仿真、实现和生成配置文件等。对某个文件进行了相应的处理后，在处理步骤的前面会出现一个图标来表示该步骤的状态。库视图则显示了工程中用户产生的库的内容。

编辑区：编辑区设计总结、文本编辑器、ISE仿真器/波形编辑器、原理图编辑器功能。设计总结提供了关于该设计工程的更高级信息，包括信息概况、芯片资源利用报告、与布局布线相关性能数据、约束信息和总结信息等。源文件和其它文本文件可以通过设计人员指定的编辑工具打开。编辑工具的选择由Edit->Preference属性决定，默认ISE的文本编辑器，通过该编辑器可以编辑源文件和用户文档，也可以访问语言模板。

信息显示：显示ISE中的处理信息，如操作步骤信息、警告信息和错误信息等。信息显示区的下脚有四个标签，分别对应控制台信息区（Console）、错误（Errors）、警告（Warnings）和文件查找区（Find in Files）。如果设计出现了警告和错误，双击信息显示区的警告和错误标志，就能自动切换到源代码出错的地方。

## 5.3 秒表设计实例

下面以一个秒表的设计为例，说明如何使用ISE创建工程并进行调试，以及如何在THINPAD硬件平台上进行配置。

### 5.3.1 创建空白工程

首先打开ISE软件，每次启动时ISE都会默认恢复到最近使用过的工程界面。当第一次使用时，由于此时还没有过去的工程记录。如图5.3所示：

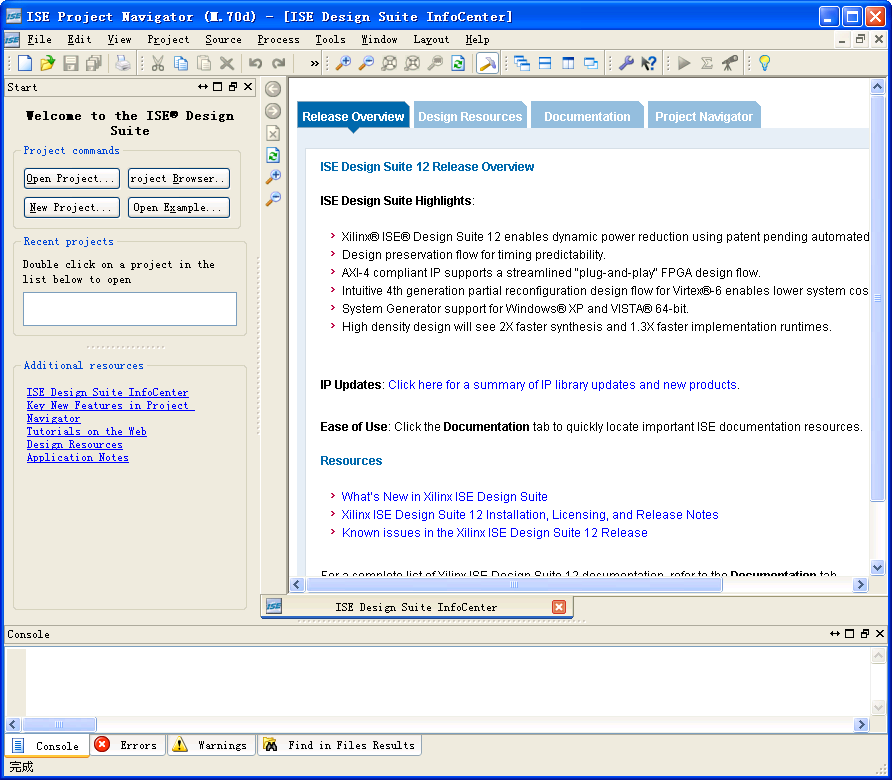


图5.3 ISE空白界面

选择左边的New Project…按钮或者通过菜单File｜New Project选项，在弹出的新建工程对话框中的工程名称中输入“clock”。在工程路径中单击Browse按键，当工程放到指定目录，并选择顶层源文件类型为HDL，如果顶层文件为原理图或其他的可以选择其它类型，如图5.4所示：

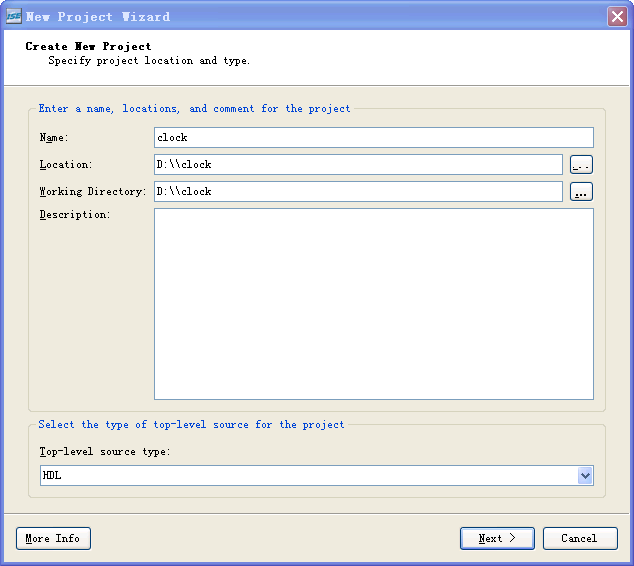


图5.4 设定工程类型和位置

点击Next，出现工程设定对话框，如图5.5所示，在这里面可以指定所选用的芯片，综合仿真工具，使用的语言类型等。其中前五项为芯片类型，主要包括芯片系列、型号、封装以及速度等级等，这些内容在所选用的芯片上都有标注，根据芯片选择即可，这里选择如图所示的Spartan3E系列的XC3S1200E，封装为FG320，速度等级是4ns。Synthesis Tool对应的是综合工具，Simulator对应的是仿真工具，这里都取默认值，即选用ISE自带的工具。Preferred Language对应语言类型，这里是VHDL。

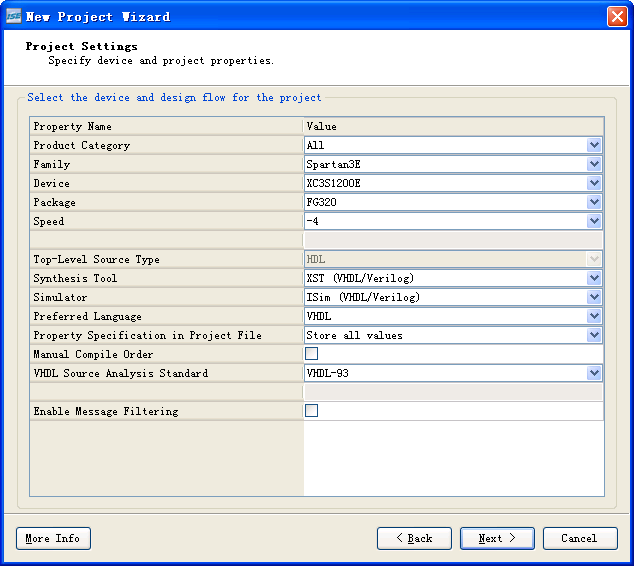


图5.5 工程设定对话框

点击Next，出现工程汇总对话框，里面显示了所创建工程的一些信息，如图5.6所示：

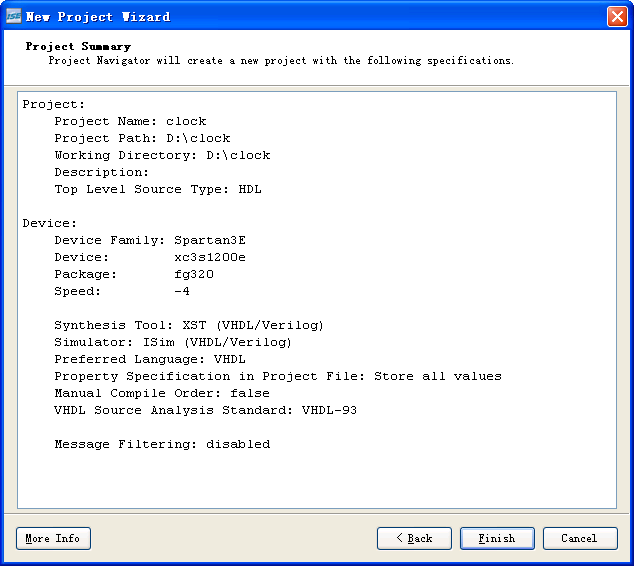


图5.6 工程汇总对话框

点击Finish完成创建空白工程，在管理区内可以看到新创建的工程clock，如图5.7所示。

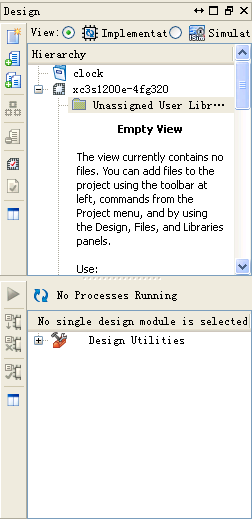


图5.7 空白工程

### 5.3.2 添加源文件

下面开始给空白工程创建并添加源文件，在管理区内右键单击工程clock，选择New Source…，出现创建源文件向导的选择源文件类型对话框，如图5.8所示，在此选择VHDL module,并设定文件名为clock，下面有个复选框选择将文件添加到工程中。

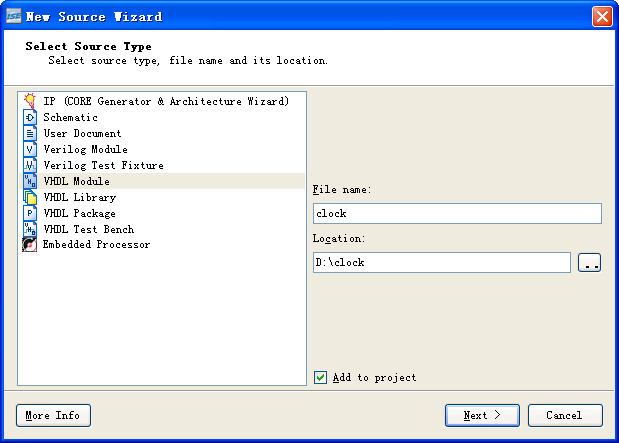
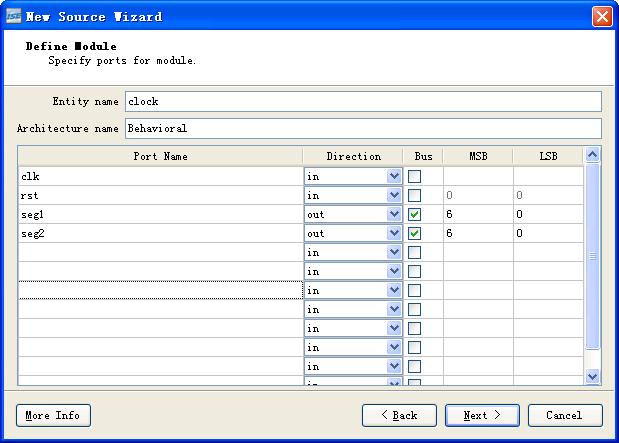


图5.8 选择源文件类型对话框

点击Next，出现模块定义对话框，如图5.9所示，这里我们要实现一个两位十进制的秒表，输入为50M时钟，复位和两个7段数码管的输出：



如图5.9 模块定义对话框

点击Next，出现新文件总结对话框，如图5.10所示：

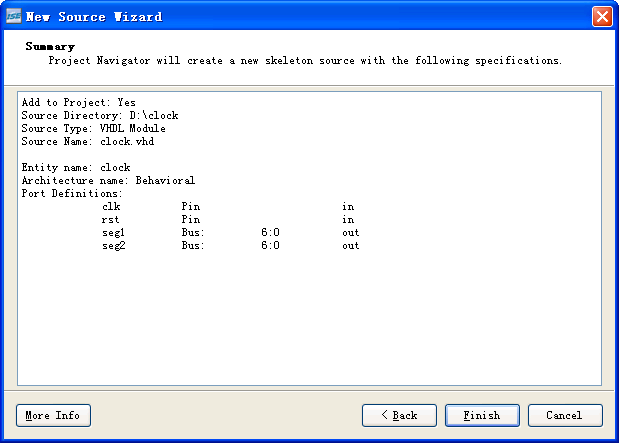


图5.10 新文件总结对话框

点击Finish完成新文件创建，如图5.11所示，在管理区能看到新文件的名字，在编辑区内是VHDL代码，但只包含了框架部分。

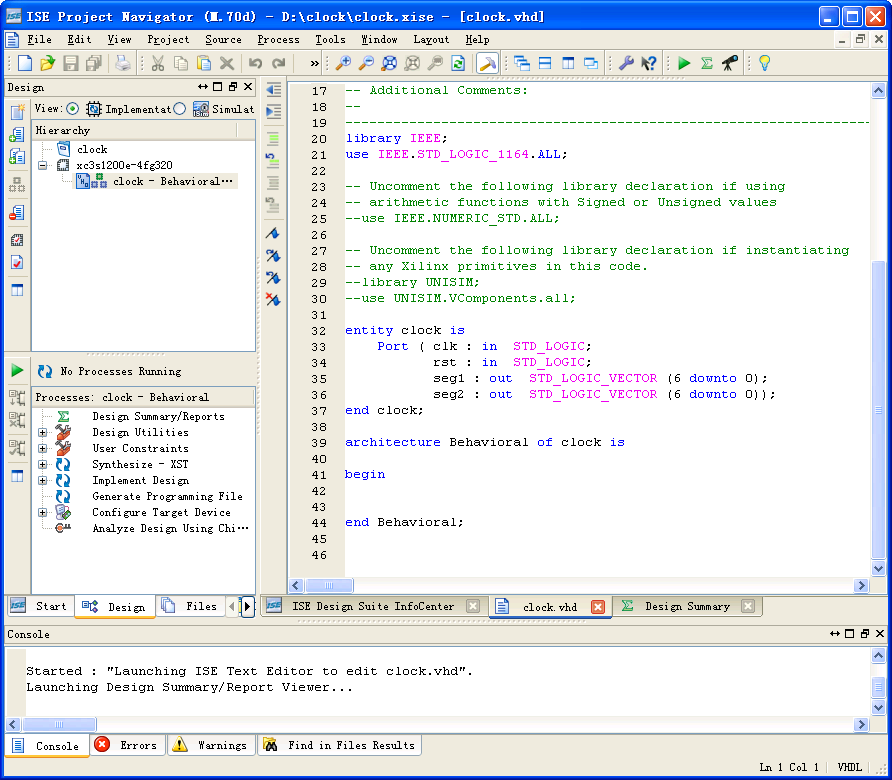


图5.11 新文件添加完成

在编辑区内修改VHDL源代码，具体代码如下：

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

use IEEE.STD\_LOGIC\_ARITH.ALL;

use IEEE.STD\_LOGIC\_UNSIGNED.ALL;

-- Uncomment the following library declaration if using

-- arithmetic functions with Signed or Unsigned values

--use IEEE.NUMERIC\_STD.ALL;

-- Uncomment the following library declaration if instantiating

-- any Xilinx primitives in this code.

--library UNISIM;

--use UNISIM.VComponents.all;

entity clock is

Port ( clk : in STD\_LOGIC;

rst : in STD\_LOGIC;

seg1 : out STD\_LOGIC\_VECTOR (6 downto 0);

seg2 : out STD\_LOGIC\_VECTOR (6 downto 0));

end clock;

architecture Behavioral of clock is

signal clk\_out : std\_logic:='0';

signal cnt: std\_logic\_vector(25 downto 0):="00000000000000000000000000";

signal cnt\_H:std\_logic\_vector(3 downto 0):="0000";

signal cnt\_L:std\_logic\_vector(3 downto 0):="0000";

begin

process(clk) --时钟

begin

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

cnt <= cnt + '1';

if(cnt = "00000000000000000000000000") then

clk\_out <= '0';

end if;

if(cnt = "01011011100011011000000000") then

cnt <= "00000000000000000000000000";

clk\_out <= '1';

end if;

end if;

end process;

process(clk\_out,rst)

variable tmp\_H, tmp\_L:std\_logic\_vector(3 downto 0):="0000";

begin

if(rst = '0')then

tmp\_H := "0000";

tmp\_L := "0000";

else

if(clk\_out'event and clk\_out='1') then

tmp\_L := cnt\_L + '1';

if(tmp\_L > "1001") then

tmp\_L := "0000";

tmp\_H := cnt\_H + '1';

if(tmp\_H > "1001") then

tmp\_H := "0000";

end if;

end if;

end if;

end if;

cnt\_L <= tmp\_L;

cnt\_H <= tmp\_H;

end process;

process(cnt\_L)

begin

case cnt\_L is

when "0000"=>seg1<=not "1000000";

when "0001"=>seg1<=not "1111001";

when "0010"=>seg1<=not "0100100";

when "0011"=>seg1<=not "0110000";

when "0100"=>seg1<=not "0011001";

when "0101"=>seg1<=not "0010010";

when "0110"=>seg1<=not "0000010";

when "0111"=>seg1<=not "1111000";

when "1000"=>seg1<=not "0000000";

when "1001"=>seg1<=not "0010000";

when others=>seg1<=not "1111111";

end case;

end process;

process(cnt\_H)

begin

case cnt\_H is

when "0000"=>seg2<=not "1000000";

when "0001"=>seg2<=not "1111001";

when "0010"=>seg2<=not "0100100";

when "0011"=>seg2<=not "0110000";

when "0100"=>seg2<=not "0011001";

when "0101"=>seg2<=not "0010010";

when "0110"=>seg2<=not "0000010";

when "0111"=>seg2<=not "1111000";

when "1000"=>seg2<=not "0000000";

when "1001"=>seg2<=not "0010000";

when others=>seg2<=not "1111111";

end case;

end process;

end Behavioral;

这是一个秒表，每秒计数一次，99秒后回到0开始重新计数。

### 5.3.3 综合与功能仿真

在集成电路设计领域，综合是指设计人员使用高级设计语言对系统逻辑功能的描述，在一个包含众多结构、功能、性能均已知的逻辑元件的逻辑单元库的支持下，将其转换成使用这些基本的逻辑单元组成的逻辑网络结构实现。这个过程一方面是在保证系统逻辑功能的情况下进行高级设计语言到逻辑网表的转换，另一方面是根据约束条件对逻辑网表进行时序和面积的优化。综合过程中，主要执行以下三个步骤：

* 语法检查过程，检查设计文件语法是否有错误
* 编译过程，翻译和优化HDL代码，将其转换为综合工具可以识别的元件序列
* 映射过程，将这些可识别的元件序列转换为可识别的目标技术的基本元件

ISE中的综合操作如下图5.12所示，只要双击Synthesize就开始综合操作：

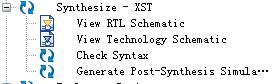


图5.12 综合操作

综合工具可以完成下面的任务：查看RTL原理图（View RTL schematic）、查看技术原理图（View Technology Schematic)、源代码语法检查(Check Syntax)、生成综合后仿真模块及报告。

当综合成功后，综合工具前面或出现√，如图5.13所示，若存在错误可根据错误说明修改，还有一种情况是存在警告，一般需要根据情况处理。

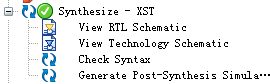


图5.13 综合成功

综合完成后，可以进行功能仿真，若只要看到下载到实验板上的结果，这一步可跳过。首先在View栏选择simulation，如图5.14所示：

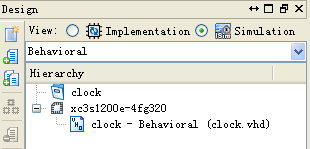


图5.14 选择simulation

选择要仿真的vhd文件，这里为clock.vhd，右键单击选择new source…，出现新文件对话框，如图5.15所示，在其中选择VHDL Test Bench文件类型，文件命名为test。

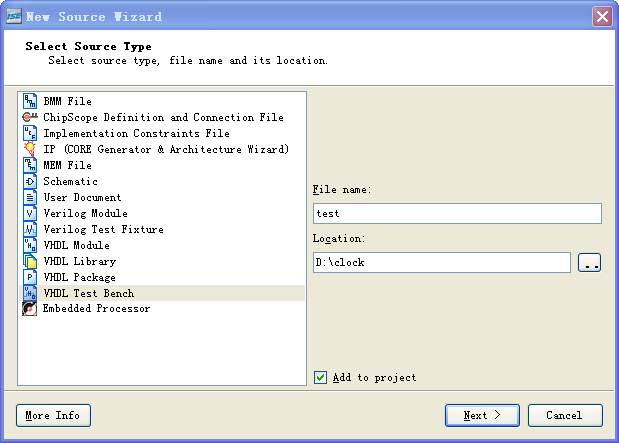


图5.15 创建test bench文件

点击Next，出现关联源文件对话框，如图5.16所示，这里将test bench关联到clock上，即对clock进行仿真：

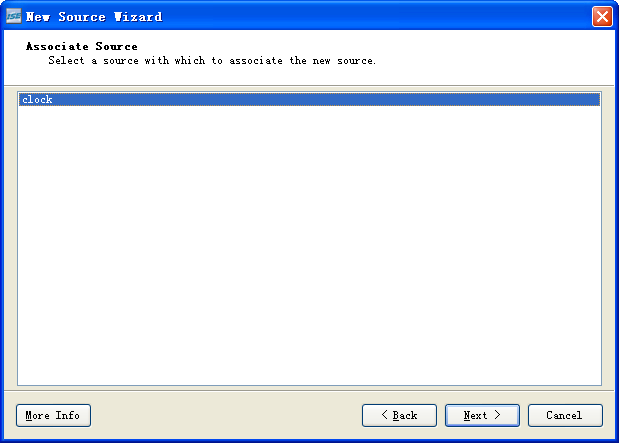


图5.16 关联源文件对话框

点击Next，出现总结对话框，如图5.17所示

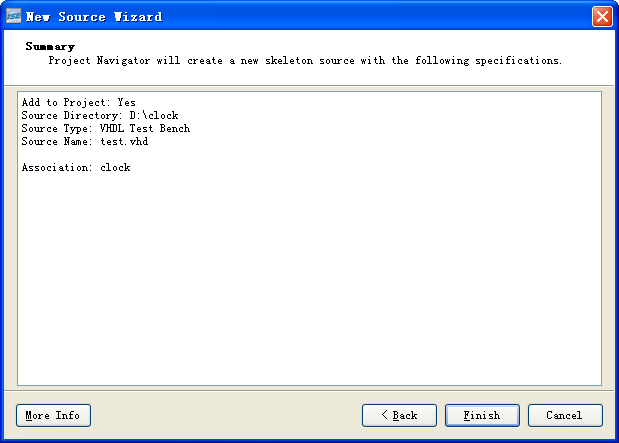


图5.16 总结对话框

点击finish完成仿真平台创建，此时在管理区内可以看到仿真平台test.vhd，以及被测试文件clock.vhd，如图5.17所示。

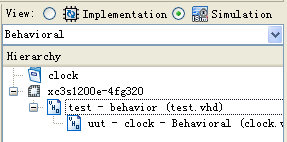


图5.17 仿真平台

此时在编辑区也出现了test.vhd的源代码，可以修改其中的代码进行仿真测试，在其中需要修改一下代码，如图5.18所示中的阴影部分：

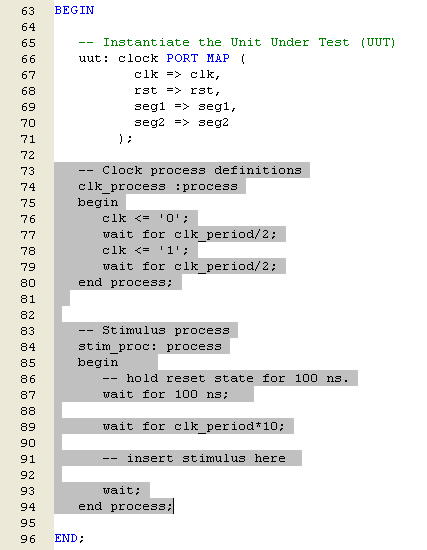


图5.18 需要修改的代码

将上述代码修改为如下代码：

-- Clock process definitions

clk\_process :process

begin

clk <= '0';

wait for 10ns;

clk <= '1';

wait for 10ns;

end process;

-- Stimulus process

stim\_proc: process

begin

-- hold reset state for 100 ns.

rst <= '0';

wait for 100ns;

rst <= '1';

wait;

end process;

上面的代码就是创建了一个50MHz的时钟并加入了一个复位信号。

在管理区下面的操作框中展开ISim Simulation，如图5.19所示：



图5.19 仿真工具栏

双击Simulation Behavioral Model开始仿真（请注意这时管理区中要选择test.vhd文件），会出现仿真窗口，如图5.20所示：

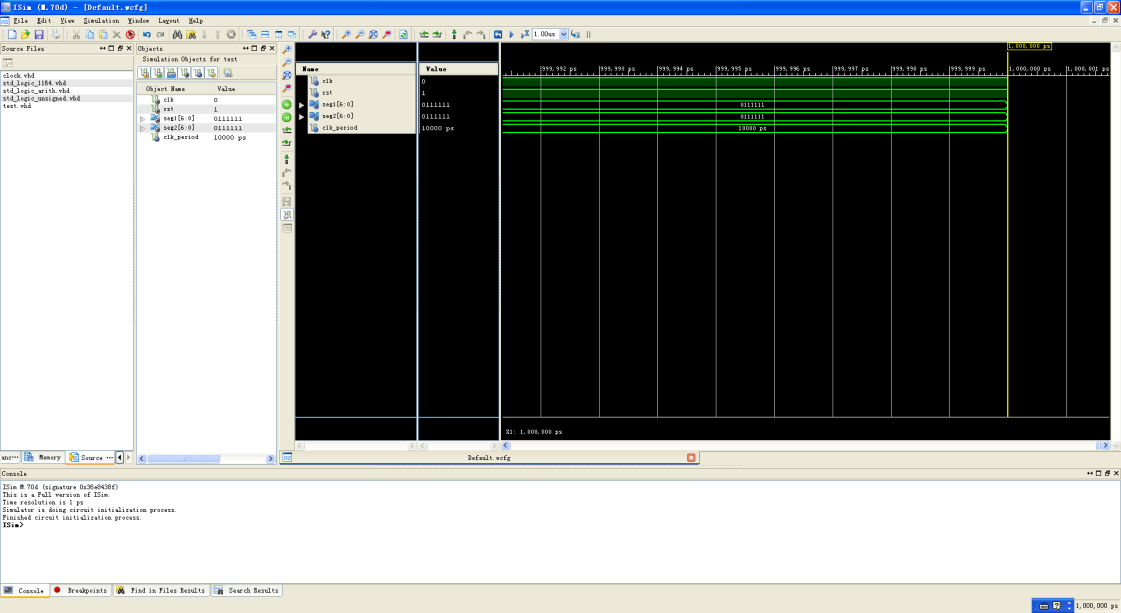


图5.20 仿真窗口

在此仿真窗口中进行仿真，由于这个设计是一个秒表，每隔一秒输出才会发生变化，因此这里每次仿真间隔为1s，如图所示，仿真后就能看到运行结果，如图5.21所示：

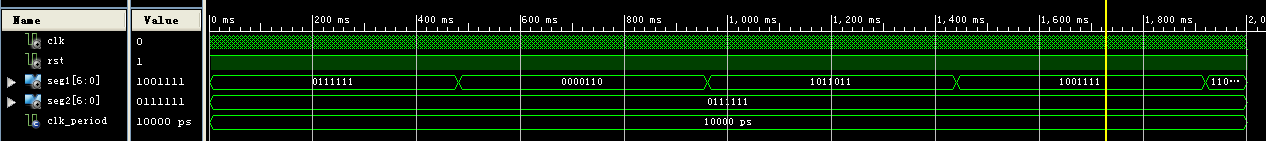


图5.21 仿真结果

由于是秒表程序，每仿真一秒才会出现变化，因此仿真过程比较慢。从上面的结果可以看出输出seg1是每秒变化一次，也是按照七段数码管的数值对应变化的，比如0111111对应就是数字0的显示。

上面的仿真都是行为仿真，也成为前仿真，这种仿真没有将信号延时加入，因此离实际的硬件还有一定的差距。功能仿真可以用来验证设计的功能，但不能用来检查严格的时序，而时序仿真（后仿真）则是将延时加入，因此更接近实际硬件运行。

### 5.3.4 添加用户约束

约束是指对设计进行一些设定，让实现工具能够根据用户想要的目标进行实现，约束可以指定管脚、限制时延、限定布线范围等。通过附加约束可以控制逻辑的综合、映射、布局和布线，以 减小逻辑和布线延时，从而提高工作频率，还可以指定I/O引脚所支持的接口标准和其他电气特性等。本节只介绍如何对设计进行管脚约束。

在管理区的View栏中选择Implementation项，选择要约束的源文件，这里是clock.vhd，在下面的操作窗口中展开User Constraints，会出现用户约束设定的选项，如图5.22所示：

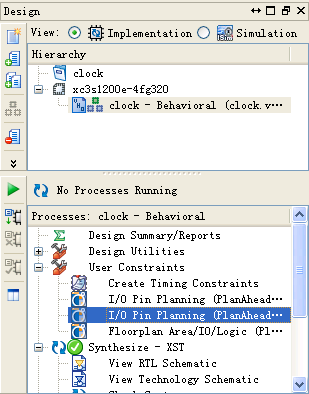


图5.22 用户约束操作

双击I/O Pin Planing(PlanAhead)-Post-Synthesis操作，会出现添加用户约束文件对话框，如图5.23所示。

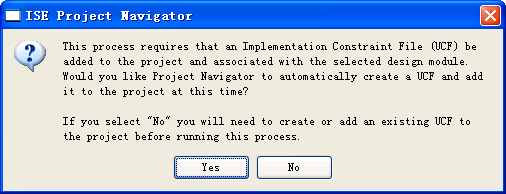


图5.23 添加用户约束文件对话框

单击Yes，确认添加用户约束文件，将打开管脚约束窗口，如图5.24所示。

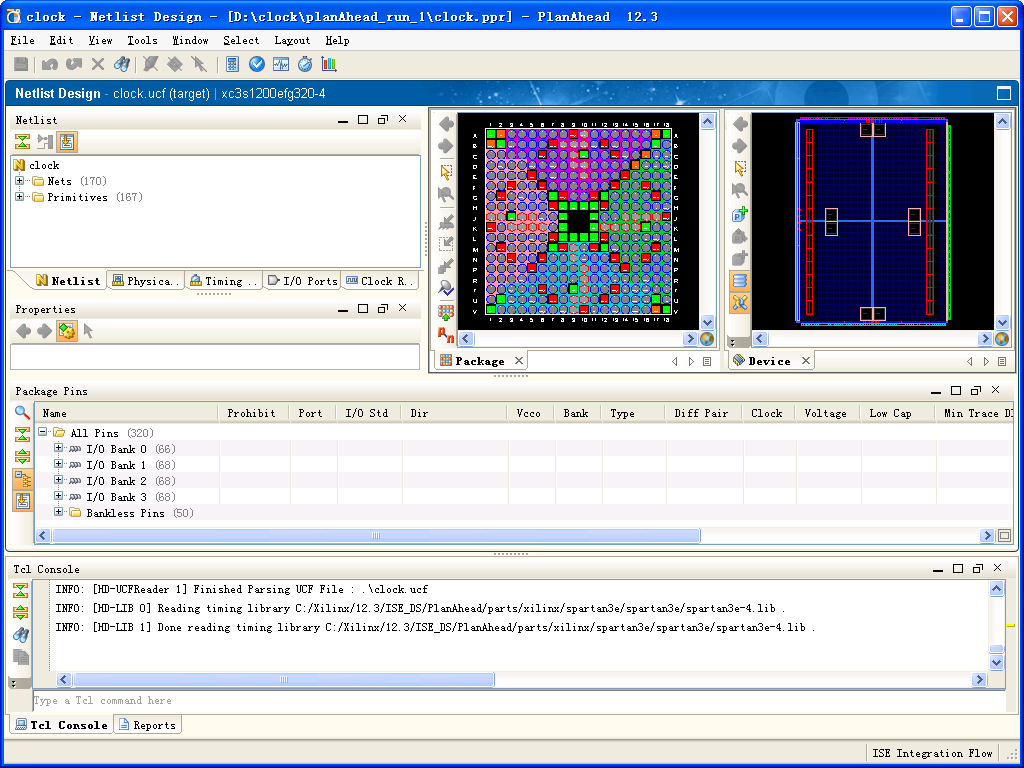


图5.24 管脚约束窗口

在其中可以指定设计的管脚，如图5.25所示，在左侧的窗口中选择I/O Port标签栏，会出现设计中所有的I/O管脚，展开后在Site列中就可以指定对应的管脚号，直接点击输入即可，管脚号可以参考表2.3。

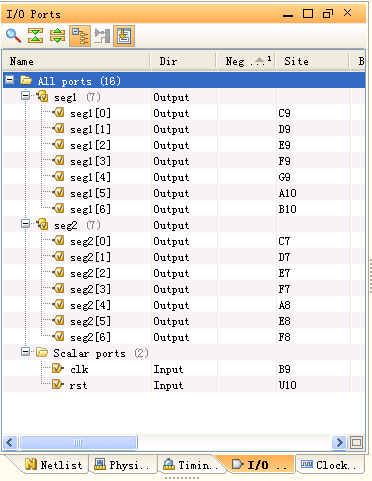


图5.25 指定管脚

管脚指定完成后保存退出窗口，在管理区的clock.vhd文件下会出现约束文件clock.ucf，如图5.26所示，用户也可以直接编辑该文件进行约束设定。

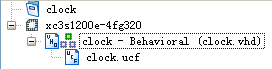


图5.26 约束文件

### 5.3.5 实现

实现就是将逻辑网表翻译成底层模块与硬件原语，将设计映射到器件结构上，进行布局布线分为翻译(Translate)，映射(Map)，布局布线(Place&Route)三步：

* 翻译：把多个设计文件合并成一个单独的网表文件。
* 映射：把网表中的门级逻辑映射到物理器件资源上。
* 布局布线：把Map中的物理器件资源在器件上布局，并用布线资源连接起来，把时序数据写入到时序报告中。

在管理区中选择clock.vhd，在操作栏中选择Implement Design，双击开始实现，如图5.27所示。

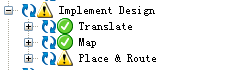


图5.27 实现

实现成功后，设计就会有延时信息，这就可以进行时序仿真了，这里就不详细介绍了。

### 5.3.6 配置

配置就是将设计下载到实际的FPGA芯片中，然后就可以开始实际测试了，首先在配置之前需要生成配置文件，并连接到下载电路。

双击操作栏中的Generate Programming File，生成可配置文件，如图5.28所示：



图5.28生成可配置文件

将USB下载线分别和计算机USB接口及THINPAD电路板上的JTAG1口相连，计算机会自动安装驱动程序，然后给电路板上电。

双击操作栏中的Configure Target Device，回弹出询问是否打开iMPACT的对话框，如图5.29所示：

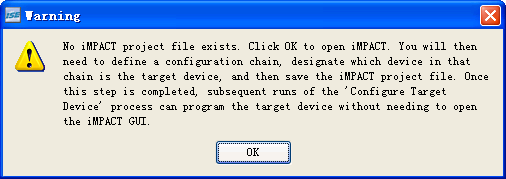


图5.29 询问是否打开iMPACT

选择OK，将打开iMPACT窗口，如图5.30所示：

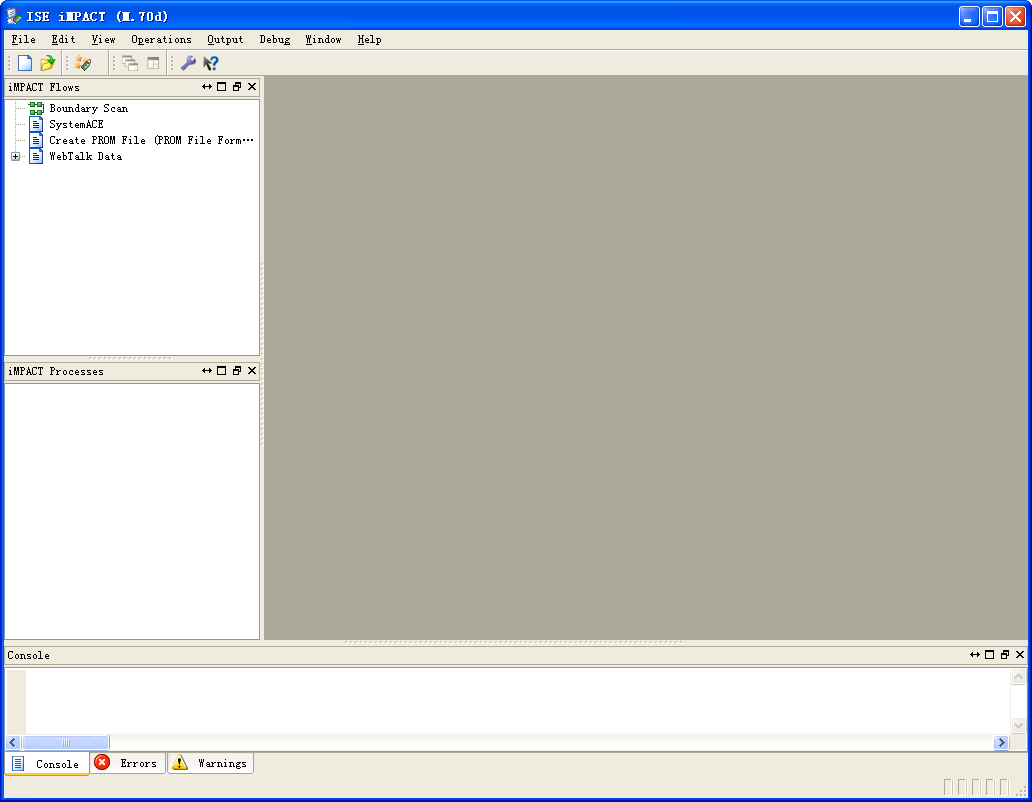


图5.30 iMPACT窗口

双击左边窗口中的Boundary Scan，在右边出现的空白窗口中右键单击选择Initialize Chain，如图5.31所示：

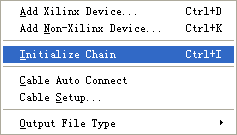


图5.31 Initialize Chain

iMPACT会识别出电路板上的FPGA芯片，显示出一个FPGA芯片图标，并弹出是否指定配置文件对话框，如图5.32所示：

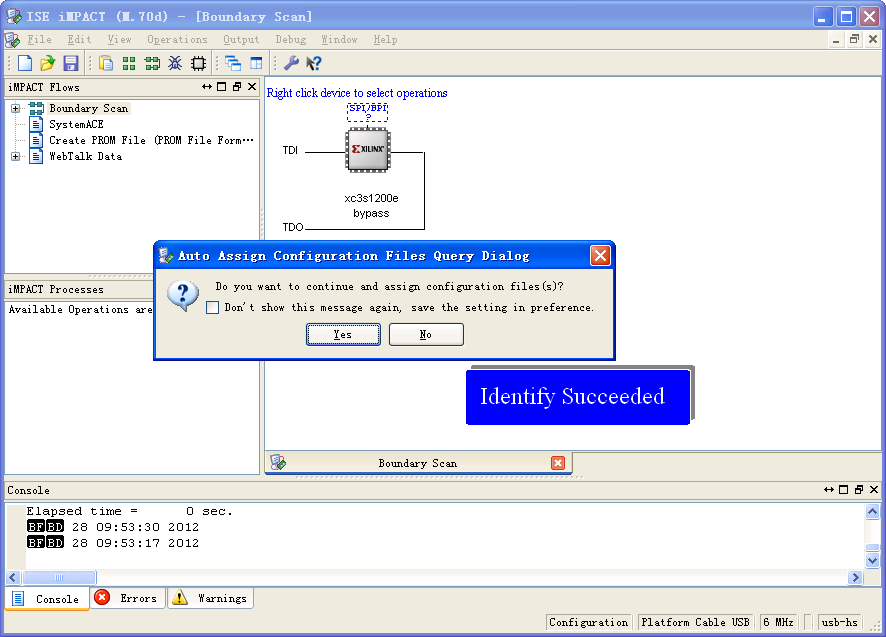


图5.32 是否指定配置文件对话框

选择Yes，弹出指定配置文件对话框，在其中选择刚才生成的配置文件clock.bit，如图5.33所示：

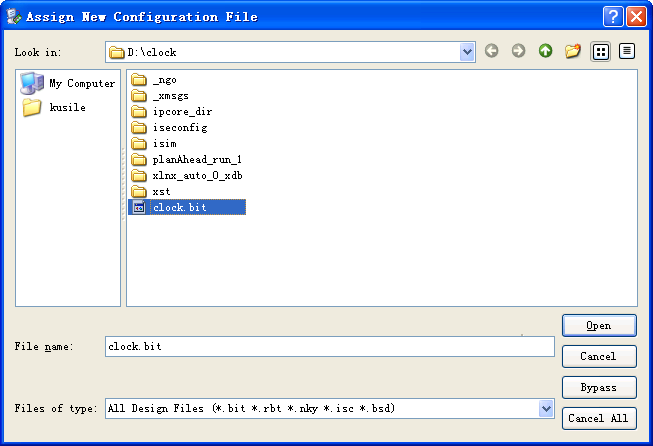


图5.33 配置文件对话框

选择Open按钮，弹出选择选择附加Flash对话框，如图5.34所示：



图5.34 选择附加Flash对话框

Spartan3E支持商用的并行Flash，此处不需要使用它，所以选择“No”按钮，弹出下载属性对话框，如图5.35所示：

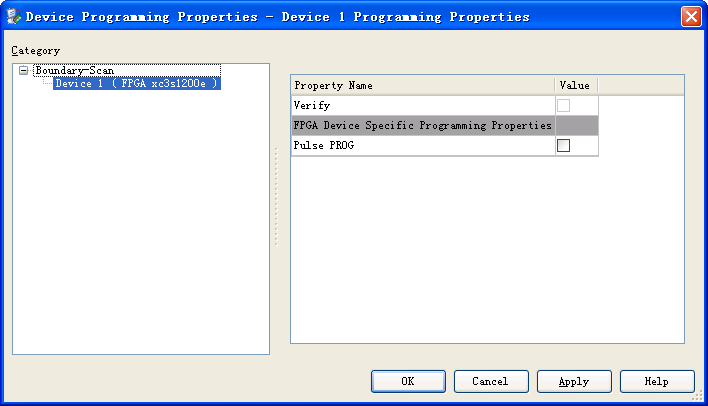


图5.35 下载属性对话框

此处选择默认设置，然后选择OK，这样FPGA芯片就指定好了配置文件clock.bit，在芯片图标上右键单击，出现选择菜单，如图5.36所示：

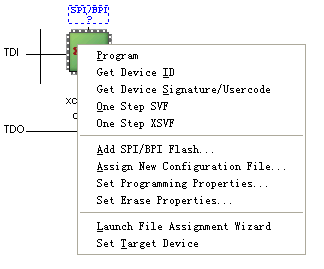


图5.36 右键菜单

在菜单中选择Program，就会开始配置FPGA，完成后出现Program Succeeded字样表示配置成功，如图5.37所示：

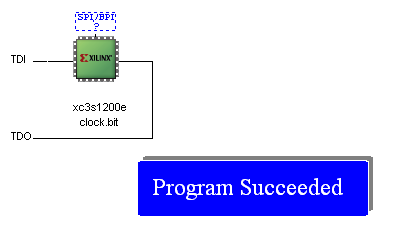


图5.37置成功

配置成功后，可以看到THINPAD上的两个七段数码管显示出数字，从0到99循环显示，如图5.38示：

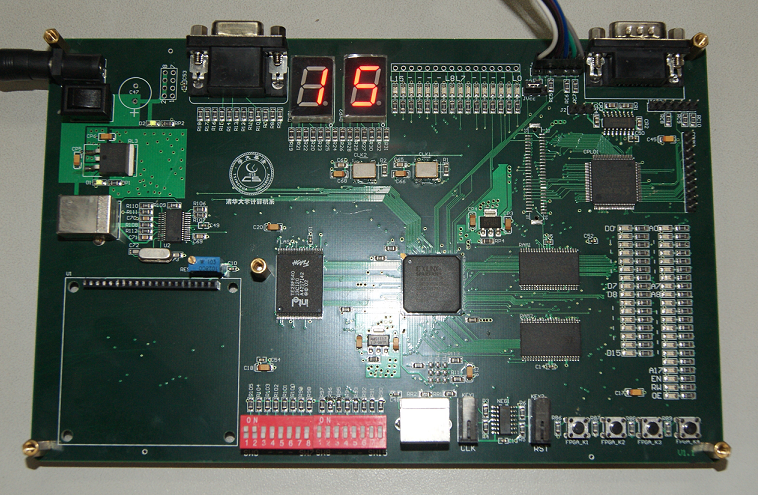


图5.38秒表示例

# 第6章 验证性实验

## 6.1 THCO MIPS指令系统实验

THCO MIPS指令系统是THINPAD教学计算机建议的指令系统，熟悉指令系统中各指令的功能及格式后，可为在后续实验中实现该指令系统打好基础。

### 6.1.1 实验目的

熟悉教学计算机THINPAD指令系统，包括指令功能、指令格式和寻址方式等；

了解教学计算机模拟器的功能，并能熟练使用；

掌握THINPAD教学计算机汇编语言程序的基本编程方法，能编写简单程序。

### 6.1.2 实验环境

硬件环境：PC计算机，WIndows7或以上操作系统；

软件环境：THINPAD教学计算机模拟器、汇编器。

### 6.1.3实验内容

编写汇编语言程序，求前10个Fibonacci数，将结果保存到起始地址为0x8500的连续10个字中；

编写汇编语言程序，将ASCII字符表在终端上输出；

自行编写一个能完成一定功能的汇编语言程序并完成调试。

### 6.1.4 实验原理

本实验通过模拟器的使用，帮助同学们学习和掌握THCO MIPS指令系统的指令功能和指令格式，并体会计算机机器语言程序的执行过程。实验提供的模拟器可直接模拟汇编语言程序的执行，也可模拟THINPAD教学计算机硬件运行监控程序的过程，在监控程序的管理下运行用户程序。

### 6.1.5 主要实验步骤

本实验所有的3个汇编语言程序均在模拟器的两种工作模式下运行。

1. 指令系统功能模拟

指令系统功能模拟模式下，启动模拟器以后，使用模拟器提供的命令，输入汇编语句并逐句完成汇编，然后再执行用户程序。具体步骤如下：

1. 启动模拟器程序；
2. 用ea命令，逐句输入编写好的程序并汇编，注意程序的起始地址；
3. 用v命令观察输入的程序的二进制代码，分析指令格式，指明操作码、操作数和寻址方式；
4. 用c命令运行程序，并用v命令查看运行结果并记录。请注意程序的首地址，必要时可先用goto命令保证程序的首地址的正确性；
5. 使用模拟器中其他命令，体会各命令的功能。
6. 按以上步骤完成3个程序的调试，并记录过程和结果。

1. 硬件系统功能模拟

硬件系统功能模拟是指用模拟器来充当THINPAD教学计算机的硬件，再用终端程序TERM来做模拟器的输入/输出设备，连接模拟器运行。具体步骤如下：

1. 启动模拟器，并确保监控程序kernel.bin在当前目录下；
2. 在模拟器下运行server指令，启动模拟THINPAD硬件执行；
3. 启动Term程序，并运行SIM命令，与模拟器模拟的硬件连接，这样，Term程序将作为模拟器的终端，与连接真正的硬件运行时完全一样；
4. 在监控程序的命令提示符下，输入“A”命令，然后输入编写好的汇编程序，输入完成后，退出编辑和汇编状态；
5. 用“G”命令运行用户程序，注意程序的首地址；
6. 用U命令逐条指令查看指令格式，并指出其寻址方式；
7. 按以上步骤完成3个程序的调试，并记录过程和结果。

### 6.1.6 思考题

① 根据程序中每条指令的机器代码，给出其指令格式及寻址方式。

② 指出程序中寻址方式为立即数寻址的指令中，立即数的机器表示和实际数值各是多少？

③如果要求程序计算前50个Fibonacci数，应如何修改程序？

④读懂监控程序和TERM程序的代码，并画出两个程序的框图。

## 6.2 监控程序扩展实验

监控程序直接运行于THINPAD硬件平台之上，相当于教学计算机的简单的操作系统，也被后续实验作为验证硬件设计是否正确的一个测试软件。了解监控程序功能及实现过程，掌握其使用方法，一方面可继续熟悉指令系统，另一方面可为后续实验的调试打好基础。

### 6.2.1实验目的

熟悉教学计算机监控程序，了解监控程序功能的具体实现；

熟悉教学计算机终端程序TERM，了解监控程序与TERM的相互关系；

将监控程序支持的指令集扩充，为后续实验打好基础。

### 6.2.2 实验环境

硬件环境：PC计算机，WIndows7或以上操作系统；

软件环境：THINPAD教学计算机模拟器、终端程序TERM、监控程序Kernel。

### 6.2.3 实验内容

阅读监控程序和终端程序TERM的源代码，并给出程序框图；

标准的TERM和监控程序只能汇编基本指令集中的指令，请将其扩充为可汇编和反汇编包含你们要实现的指令集的全部指令。

### 6.2.4 实验原理

本实验通过对相关程序的源代码的分析，要求同学们了解后续实验中需要运行的软件的结构，进一步熟悉指令系统和实验环境。

教学计算机中的监控程序功能相对简单，仅完成了最基本的字符输入和输出功能，而对这些字符进行分析和正确显示是通过高级语言编写的TERM来完成的。实验中请注重了解两个程序的配合关系。

### 6.2.5 主要实验步骤

本实验主要是对程序的阅读和分析，具体操作中仅仅是对完成修改后的程序进行验证。

根据本组所要完成的指令系统，用“A”命令汇编基本指令和全部扩展指令，并用“U”命令进行反汇编，用“D”命令检查结果是否正确。

### 6.2.6 思考题

1. 读懂监控程序和TERM程序的代码，并画出两个程序的框图。

② 监控程序是否也能完成汇编和反汇编功能？如何让监控程序连接标准的终端程序实现目前的功能?请简要分析实现的难点。

## 6.3 算逻部件ALU实验

算术逻辑部件ALU是计算机的核心功能部件，完成数据的算术和逻辑运算，功能和结构相对简单，实现也不太复杂，但却是所有计算机的基础。

### 6.3.1 实验目的

1. 熟悉硬件描述语言及开发环境，了解硬件系统开发的基本过程；
2. 掌握简单运算器的数据传送通路；
3. 验证运算器的功能。

### 6.3.2实验环境

1. 硬件环境：PC计算机，win7及以上操作系统；THINPAD开发板；
2. 软件环境：FPGA开发工具软件Xilinx ISE 12.3。

### 6.3.3实验内容

1. 根据实验原理中的要求，用VHDL语言实现一个简单的ALU；
2. 在教学计算机THINPAD上验证实现的ALU的功能。

### 6.3.4实验原理

本实验通过设计一个简单的ALU帮助同学们更好的理解数据通路和ALU的工作原理，并通过该实验使同学们熟悉VHDL硬件描述语言，为接下来的实验打好基础。

算术逻辑部件的主要功能是对二进制数据进行定点算术运算、逻辑运算和各种移位操作。算术运算包括定点加减乘除运算；逻辑运算主要有逻辑与、逻辑或、逻辑异或和逻辑非操作。ALU通常有两个数据输入端A和B，一个数据输出端Y以及标志位等。本实验通过实现一个状态机，根据状态机状态的变化实现不同的运算，将结果和标志位呈现出来。实验中的ALU可以实现基本的算术运算、逻辑运算、移位运算等，功能如下表所示：

|  |  |  |
| --- | --- | --- |
| 操作码 | 功能 | 描述 |
| ADD | A + B | 加法 |
| SUB | A – B | 减法 |
| AND | A and B | 与 |
| OR | A or B | 或 |
| XOR | A xor B | 与或 |
| NOT | not A | 取非 |
| SLL | A sll B | 逻辑左移B位 |
| SLA | A sla B | 算术左移B位 |
| ROL | A rol B | 循环左移B位 |

要求：ALU为16位，算术运算时数据用补码表示。

### 6.3.5主要实验步骤

本实验通过VHDL语言实现一个比较简单的ALU模块，并且利用THINPAD教学计算机完成ALU功能的验证，具体步骤如下：

1. 用VHDL语言编写ALU功能代码，并用状态机对其进行控制，使其完成实验要求的操作。操作码和操作数的输入可用微型开关SW0~SW15，计算结果的输出可用教学机上的LED灯来显示。
2. 将代码下载到教学机的FPGA芯片中，并调试完成。
3. 在THINPAD教学机上运行时，reset和时钟均用手动开关或按钮，便于演示。操作码和操作数在开关SW0——SW15上输入；为便于观察和调试，每次ALU得到操作码和操作数，最好可以在led上显示一下。最后的运算结果在L0——L15上显示，标志位可自行选择显示方法。
4. 记录实验结果。

注意：

教学机上SW0——SW15是ON端表示0，相反表示1；CLK、RST、K1——K4均是按下为0，放开为1。

### 6.3.6思考题

* 1. ALU进行算术逻辑运算所使用的电路是组合逻辑电路还是时序逻辑电路？
  2. 如果给定了A和B的初值，且每次运算完后结果都写入到B中，再进行下次运算。这样一个带暂存功能的ALU要增加一些什么电路来实现？

## 6.4 内存储器系统实验

存储器系统是计算机中存放程序和数据的场所，分为内存和外存。THINPAD教学计算机设置了两片RAM作为内存，使用FLASH存储器作为外存。本实验主要完成内存储器的访问。

### 6.4.1 实验目的

1. 熟悉THINPAD教学计算机内存储器的配置及与总线的连接方式；
2. 掌握教学机内存（RAM）的访问时序和方法；
3. 理解总线数据传输的基本原理。

### 6.4.2实验环境

1. 硬件环境：PC计算机，win7及以上操作系统；THINPAD教学计算机；
2. 软件环境：FPGA开发工具软件Xilinx ISE 12.3。

### 6.4.3实验内容

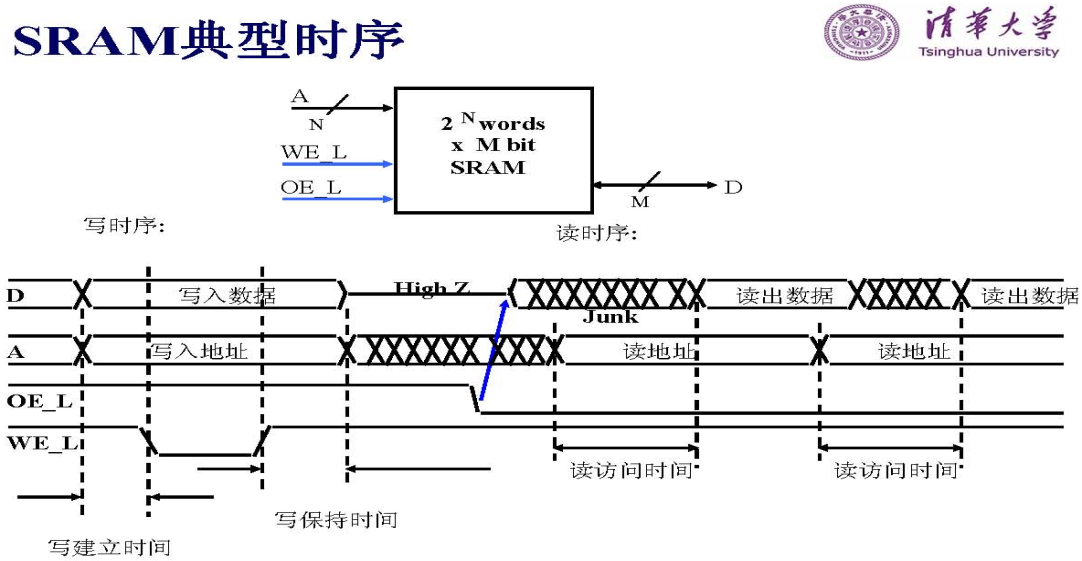
使用教学计算机上的FPGA芯片，完成对存储器RAM的访问。具体要求如下：

1. 写RAM1。将手拨开关上的数据，写入到RAM1的相应存储单元中。首先，在手拨开关上拨入写入地址单元的地址，按CLK键后，再在手拨开关上拨入要写入该单元的数据，再按CLK键后，数据应写入到RAM1的对应单元中。继续按CLK键，则地址和数据各加1后写入，共写10个数据。过程中，应在LED灯上分别显示地址和数据。
2. 读RAM1。按Reset键，转化到读RAM1的模式。然后，按CLK键，逐个 将刚写入的10个数据从存储单元中读出，送到LED上显示。
3. 写RAM2。将前面RAM1中已保存的10个数据，各减1后写入到RAM2的同样的地址单元内，写入过程中要求在LED灯上输出地址和被写入的数据。
4. 读RAM2。要求同读RAM1。

### 6.4.4实验原理

对于存储器芯片的访问，首先要熟悉存储器芯片的访问时序，并了解THINPAD教学机上存储器芯片与FPGA芯片的具体连接方式，也就是说，如何在THINPAD教学机上具体实现对RAM芯片的访问时序。

下图是SRAM的访问时序：



从时序要求上看，要完成实验，就要给内存芯片提供地址、数据和控制信号。而初始地址和数据均来自手拨开关，需要有寄存器接收并保存好，然后分别送到地址总线和数据总线。控制信号有3个，以RAM1为例，分别是RAM1\_EN、RAM1\_OE和RAM1\_RW，对应连接到RAM芯片的/CE、/OE和/WE管脚，需要根据对存储芯片的访问要求来正确设置。需要指出的是，由于RAM1和串口共同连接在一条总线上，可用/CE信号来控制存储芯片工作还是串口工作。

### 6.4.5主要实验步骤

本实验的完成需要用FPGA芯片来控制存储芯片的读写，根据实验要求，需要在FPGA中实现地址寄存器、数据寄存器，并实现一个简单的状态机来控制对存储芯片的访问。具体步骤如下：

1. 用VHDL语言编写存储芯片访问的控制代码，设计状态机对其进行控制，使其完成实验要求的操作。初始地址和数据均来自手拨开关SW0~SW15，地址和数据的输出可用教学机上的LED灯来显示。
2. 将代码下载到教学机的FPGA芯片中，并调试完成。
3. 注意观察LED上地址和数据的变化，是否符合实验的设计要求
4. 记录实验结果。

注意：

教学机上SW0——SW15是ON端表示0，相反表示1；CLK、RST、K1——K4均是按下为0，放开为1。

### 6.4.6思考题

1. 静态存储器的读和写各有什么特点？
2. 什么是RAM芯片输出的高阻态？它的作用是什么？
3. 本实验完成的是将RAM1和RAM2作为独立的存储器单独进行访问的功能。如果希望将RAM1和RAM2作为一个统一的32位数据的存储器进行访问，该如何进行？

## 6.5 串行接口实验

计算机内部数据通常以字或字节为单位进行并行访问。但与外部设备进行数据交互或与其他计算机进行通信时，也经常需要串行数据通信。串行接口是一类使用相对简单的接口，THINPAD教学计算机上也配置了基本的串口，作为连接终端设备的接口。本实验主要完成串行接口的访问。

### 6.5.1 实验目的

1. 熟悉THINPAD教学计算机串行接口的配置及与总线的连接方式；
2. 掌握教学机串口UART的访问时序和方法；
3. 理解总线数据传输的基本原理。

### 6.5.2实验环境

1. 硬件环境：PC计算机，win7及以上操作系统；THINPAD教学计算机；
2. 软件环境：FPGA开发工具软件Xilinx ISE 12.3；串口调试精灵。

### 6.5.3实验内容

使用教学计算机上的FPGA芯片编写代码，完成对教学机CPLD芯片中UART的访问，实现和PC机上运行的串口调试精灵程序的通信。要求分别实现如下功能：

1. 能接收串口调试精灵发送的数据（十六进制00~FF），并显示在THINPAD教学机的LED灯上；
2. 能通过UART往串口精灵发送数据（十六进制00~FF），并由串口精灵程序显示出来；
3. 接收串口精灵发过来的数据，加1后再发送回串口精灵并显示。

### 6.5.4实验原理

THINPAD教学计算机的FPGA芯片通过基本总线连接了存储器芯片RAM1以及被配置为UART的扩展芯片CPLD。UART连接RS-232接口，作为串口与其他设备连接。本实验中，该串口将连接PC机的串口，与PC机上运行的“串口精灵”程序通信。

由于UART和RAM1共享基本总线，因此，需要有控制信号来区分在一个总线周期内是RAM1工作还是UART工作，是进行读操作还是写操作。在THINPAD教学计算机中，给出了3位控制信号RAM1\_EN、RAM1\_OE、RAM1\_RW来实现这个功能。

和普通的串口芯片8251不同，教学机上的UART没有设计缓冲，也不具备可编程的性质（它本身就是由CPLD编程实现的）。其基本组成如图2.12所示，功能和管脚说明也已在第2章中叙述。我们在此简单说明如下。

异步接收/发送器（UART），可完成并行数据和串行数据之间的相互转换，还能检测串行通信在传送过程中可能发生的错误。UART主要由数据总线接口、控制逻辑、波特率发生器、发送部分和接收部分等组成。本实验主要涉及UART中最重要的发送部分和接收部分， 其功能包括发送缓冲器（tbr）、发送移位寄存器（tsr）、帧产生、奇偶校验、并转串、数据接收缓冲器（rbr）、接收移位寄存器（rsr）、帧产生、奇偶校验、串转并。

数据的发送由UART中的微处理器控制，微处理器给出wrn信号，发送器根据此信号将并行数据din[7..0]锁存进发送缓冲器tbr[7..0]，并通过发送移位寄存器tsr[7..0]发送串行数据至串行数据输出端sdo。在数据发送过程中用输出信号tbre、tsre作为标志信号，当一帧数据由发送缓冲器tbr[7..0]送到发送移位寄存器tsr[7..0]时，tbre信号为1，而数据由发送移位寄存器tsr[7..0]串行发送完毕时，tsre信号为1，通知CPU在下个时钟装入新数据。

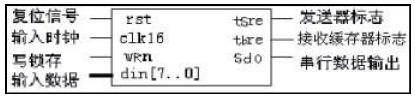
[](http://www.dzkf.cn/upimg/allimg/0705/1_08131455.JPG)

图6. UART发送器结构

串行数据帧和接收时钟是异步的，发送来的数据由逻辑1变为逻辑0可以视为一个数据帧的开始。接收器先要捕捉起始位，确定rxd输入由1到0，逻辑0要8个CLK16时钟周期，才是正常的起始位，然后在每隔16个CLK16时钟周期采样接收数据，移位输入接收移位寄存器rsr，最后输出数据dout。还要输出一个数据接收标志信号标志数据接收完。

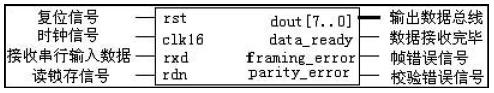
[](http://www.dzkf.cn/upimg/allimg/0705/1_08131746.JPG)

图6. UART接收器结构

以上是UART工作的基本原理说明。请同学们事先把串口的工作原理掌握清楚后，尤其是访问的时序后，再进行实验代码的编写。CPLD和FPGA芯片的管脚分配请参阅第2章相关内容。

另外需要注意的是，UART和RAM1都是通过基本总线和FPGA相连，本实验仅仅是完成UART的访问，请一定通过控制信号让RAM1不工作（不往总线上发送数据）。

### 6.5.5主要实验步骤

本实验的操作比较复杂，首先要将教学机和PC机通过串口连接好，并正确配置串口精灵的参数（端口、波特率等），然后，在编写VHDL代码时，要正确理解UART的工作原理，设计好UART发送和接收数据的状态转换关系，以及各状态与对应的信号之间的关系。在此基础上，设计好实现的代码，再将代码下载到FPGA中，进行调试。具体步骤如下：

1. 用VHDL语言编写往串口发送数据的代码，将手拨开关上的十六进制数据发送到串口，并在串口精灵中显示。

（提示：可设计4个状态。第1个状态为初始化，将UART设置为写串口状态，为发送数据做好准备；第2个状态为将待发送数据送入到发送器tbr[7..0]并锁存；第3个状态等待被发送数据进入移位寄存器tsr[7..0]；最后1个状态等待数据发送完毕tsre信号。）

1. 用VHDL语言编写读串口数据的代码，将从串口精灵中发送过来的数据读出，并显示到LED灯上。
2. 综合上面两个操作，将从串口精灵中发送过来的数据，加1后送回到串口精灵中显示。
3. 记录实验结果。

### 6.5.6思考题

1. 请总结教学机上的UART和普通的串口芯片8251的异同点？
2. 如果要求将串口精灵中发送过来的数据存入到RAM1的某个单元，然后从该单元中读出，再加1送回到串口精灵，则代码需要进行怎样的修改？有兴趣可以试着做一下。

1. 号中的参数表示可以由用户输入，也可以不输入，对于后者，系统将采用默认值。 [↑](#footnote-ref-1)
2. 格式中尖括号中的参数表示必须由用户输入，否则视为格式错误。 [↑](#footnote-ref-2)
3. 模拟器实现每执行65536条指令会产生一个时钟中断。 [↑](#footnote-ref-3)