

# 计算机组成原理 课程设计指导书

主编 张磊

北京科技大学  
计算机实验室  
2016 年 2 月

# 前 言

本书从计算机科学与技术专业教学指导委员会发布的计算机组成原理课程设计的大纲出发，首先介绍了模型机设计的 EDA 环境安装配置。针对往届学生设计过程中遇到的典型问题，对 QuartusII 软件的高级用法等展开介绍。由于硬件描述语言是往年学生在模型机设计过程中的大碍，因此利用一个章节的篇幅应用性的介绍了硬件描述语言。之后重点介绍了模型机最小系统-单周期 MIPS 处理器模型机的设计思路与实现步骤；并用 Verilog 语言实现了简易教学用 RISC\_CPU，希望起到抛砖引玉的作用。学生可在此基础上发挥自己的潜能，自行设计指令集完成计算机组成原理课程设计。

作者希望本书内容能引起对 CPU 和复杂数字逻辑系统设计有兴趣的电子工程师们的注意，加入我国集成电路的设计队伍，提高我国电子产品的档次。

在编写过程中，郑榕老师、田军峰老师参与了部分章节的编写，同时得到了刘宏岚老师和张晓彤老师的大力支持，并提供了很多宝贵的意见。

由于作者的经验与学识有限，不足之处敬请读者批评、指正。

编者  
2016 年 2 月

# 目 录

前 言.....	2
第 0 章 计算机组成原理课程设计介绍.....	1
第 1 章 EDA 平台软件安装使用介绍.....	3
1.1 EDA 平台软件安装.....	3
1.1.1 QuartusII 软件安装破解.....	3
1.1.2 Modelsim 软件安装破解.....	5
1.1.3 QuartusII 与 Modelsim 关联设置.....	5
1.2 软件使用介绍.....	6
1.2.1 计数器宏模块调用.....	7
1.2.2 寄存器与锁存器的调用.....	10
1.2.3 ROM/RAM 宏模块的调用与测试.....	12
1.3 硬件平台介绍.....	17
1.3.1 可编程逻辑器件与 FPGA 简介.....	17
1.3.2 FPGA 的特点.....	17
1.3.3 课程设计的实现平台.....	18
第 2 章 HDL 语言基础.....	19
2.1 VerilogHDL 基本程序结构.....	19
2.2 Verilog HDL 语言的数据类型和运算符.....	19
2.3 Verilog HDL 语言的描述语句.....	29
2.4 Verilog 代码书写规范.....	41
2.5 小结.....	43
第 3 章 单周期 MIPS 处理器设计.....	44
3.1 指令系统设计.....	47
3.2 数据通路设计.....	49
3.2.1 选择合适的组件.....	49
3.2.2 建立数据通路.....	51
3.3 控制信号生成.....	55
3.4 控制信号的集成.....	63
拓展练习：.....	65
第 4 章 硬件描述语言实现教学用 RISC CPU 设计.....	66
4.1 设计背景介绍.....	66
4.2 什么是 CPU.....	66
4.3 RISC_CPU 寻址方式和指令系统.....	67
4.4 RISC_CPU 结构.....	68
4.4.1 时钟发生器.....	69
4.4.2 指令寄存器.....	71
4.4.3 累加器.....	72
4.4.4 算术运算器.....	73
4.4.5 数据控制器.....	74
4.4.6 地址多路器.....	74
4.4.7 程序计数器.....	75

4.4.8 状态控制器.....	76
4.4.9 外围模块.....	81
4.5 RISC_CPU 操作和时序.....	82
4.5.1 系统的复位和启动操作.....	82
4.5.2 总线读操作.....	83
4.5.3 总线写操作.....	83
4.6 RISC_CPU 模块的调试.....	84
4.6.1 RISC_CPU 模块的前仿真.....	84
4.6.2 RISC_CPU 模块的综合.....	97
4.6.3 RISC_CPU 模块的优化和布局布线.....	98
4.7 小结 .....	104
拓展练习: .....	104
参考文献.....	104



## 第0章 计算机组成原理课程设计介绍

计算机组成原理课程设计为计算机专业的一门实践类专业必修课程，总学时为 16 学时，占 1 学分。一般安排在秋季小学期开课。根据教育部高等学校计算机科学与技术教学指导委员会编制的高等学校计算机科学与技术专业实践教学体系与规范，该门课程要求学生综合运用计算机组成、数字逻辑和汇编语言等课程知识，设计、调试和实现一台能运行的计算机。使学生更进一步地理解计算机的原理和组成，掌握计算机部件的设计方法，培养学生实际操作能力和分析、解决问题的能力。该课程的课堂授课包括 VerilogHDL 语言、实验相关的设计工具介绍、体系结构设计等内容。

该课程的目的是通过一个完整 N（N 为 8、16 或 32）位计算机系统的设计和实现，进一步加深理解计算机原理系列课程相关内容，掌握处理器设计的基本方法、培养学生解决实际问题的能力、科学作风及协作能力等。本课程设计要求了解和熟悉计算机系统的组成原理；掌握计算机简单功能部件的工作原理和设计方法；了解控制器部件的工作原理，掌握其设计方法；掌握指令集设计的一般方法；熟悉和掌握 HDL 语言，熟悉相关 EDA 设计工具，能够发现、分析和解决各种设计问题；注重团队成员协作。建议 3 人一组进行，每个小组自行设计计算机系统结构和指令系统。

计算机原理课程设计的处理器需要在可编程器件上进行功能验证，这得益于 FPGA 器件和相关的 EDA 工具。随着 FPGA 设计技术的兴起，高性能 EDA 综合开发工具（平台）也得到发展，而且其自动化和智能化程度不断提高，为复杂数字电路设计提供了易于学习和方便使用的集成开发环境，该环境具有集设计输入、综合、布局、布线、编译、模拟、验证和器件编程等一体化的功能，已经得到广泛应用。鉴于计算机逻辑实现的复杂性，计算机原理课程设计采用 HDL 语言进行设计，主要的设计和模拟测试过程均在 PC 机上使用 EDA 工具进行。相关的 EDA 软件包括设计输入与综合、模拟仿真、设计结果最终生成硬核配置文件，下载至 FPGA 芯片中进行功能验证。主要任务是设计出完整的 CPU 部件并能够正确运行，还需要设计其指令集以及编写各种测试程序。硬件环境包括 PC 机和综合实验平台，软件环境包括 ISE, Modelsim, QuartusII 等和实验平台的控制软件。在实验平台上，经配置后的 FPGA 作为 CPU，外围电路提供片外程序存储器和实时节拍，其中存储器为 256 字节，要求将设计后的 CPU 硬核下载到 FPGA 中，将测试程序加载到 RAM 中，单步调试，并能连续运行。通过微机查看存储器内的数据结果。

主要任务和步骤如下：

- (1) 计算机系统结构设计
- (2) 指令系统设计
- (3) 计算机各功能部件的设计
- (4) 综合、仿真
- (5) 编写和运行测试程序，硬件验证
- (6) 完成实验报告

基于综合实验平台和 VerilogHDL 设计和实现一个 N 位字长的计算机，该计算机的组成包括以下部件：N 位字长的运算器、指令系统和控制器、程序存储器/数据存储器、通用寄存器组和其他必要部件。该计算机应能进行加/减、无符号乘法等多种算术运算。要求设计的计算机系统能完成指定的功能，功能较强且又简洁。设计计算机总体结构，画出计算机总体框图，规定各功能部件的功能和各功能部件之间数据通路的走向。

设计的指令系统必须能够完成规定的各种运算需求，并且应该考虑指令的效率，要求必须包括如下指令和寻址方式：

- (1) MOV 类型：包括取数、存数和寄存器之间传送等指令。取数、存数指令应具

有立即数、存储器立即寻址、寄存器立即寻址等方式。

(2) 运算类型：包括加法、减法、加 1（或减 1）、移位、取反等基本算术和逻辑运算指令。

(3) 转移类型：无条件转移指令、条件转移指令。

(4) 停机指令。

利用 EDA 工具模拟和测试各个功能部件和整机系统，并且能够获得正确的测试结果。利用实验平台对设计后的计算机进行功能验证，能够正确执行程序，并且获得正确的执行和运算结果。计算机原理课程设计的实验步骤如下：

- (1) 用 Top Down 设计方法，画出计算机总体结构框图。
- (2) 确定每一个功能部件的功能，以及与外部的连接端口。
- (3) 用 VerilogHDL 完成各功能模块设计。
- (4) 对每个模块分别进行模拟测试。
- (5) 将各功能模块互连实现整个计算机系统的设计。
- (6) 编写测试程序，对计算机系统进行功能模拟和测试。
- (7) 综合、设计实现，产生下载文件，下载至实验平台。
- (8) 利用设计的指令系统编写各种测试程序，下载至实验平台上的存储器，运行 CPU，查看运算结果。

重点是计算机系统结构、指令系统、控制器，其中指令系统和控制器设计是实验的重点和难点内容，两者是紧密联系的。指令系统和控制器的设计工作主要包括以下内容：

- (1) 确定指令系统及指令编码。
- (2) 分析和确定执行每条指令所需的微操作数及节拍数。
- (3) 确定系统时钟节拍数。
- (4) 确定每个节拍对应的微操作。
- (5) 根据上述分析，设计实现状态机。
- (6) 编写各个控制信号的产生逻辑。

在完成 HDL 设计并通过仿真测试后，经过综合实践，生成配置文件下载至实验平台进行功能验证和调试，其硬件验证步骤如下：

- (1) 连接好下载电缆，打开实验平台电源，将设计文件下载至 FPGA。
- (2) 用通信电缆将 PC 机与实验平台连接。
- (3) 在 PC 机上将编写的各种算术程序编译成机器代码。
- (4) 使用平台控制软件将机器代码下载到实验平台上的存储器中。
- (5) 复位并运行 CPU，观察运行过程。
- (6) 程序运行完成后在 PC 机上读取存储器数据，检查运算结果。

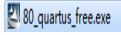
根据该大纲要求，本书下面分为四个章节来介绍。首先是对于实验平台和开发工具的介绍，然后应用性地介绍硬件描述语言，最后两章给出了用电路图和硬件描述语言两种方式实现模型机的分析方法和具体步骤，均采用硬布线逻辑实现控制器。

## 第 1 章 EDA 平台软件安装使用介绍

本书中的模型机设计是基于 QuartusII8.0 和 Modelsim SE 10.2c 版本软件进行的。硬件实验平台上 FPGA 型号为 Altera 公司生产的 CycloneII 系列的 EP2C5Q208C8N。

### 1.1 EDA 平台软件安装

#### 1.1.1 QuartusII 软件安装破解

在 32 位 Win7 操作系统下安装 QuartusII 8.0，双击图标 ，点击 setup，点下一步，根据提示步骤安装完成。接下来开始破解：在路径 C:\altera\80\quartus\bin 下，用记事本打开 License.DAT，把 license.dat 里的 XXXXXXXXXXXX 用本计算机的网卡号替换。本机网卡号通过如下方法得到：点击“开始”->“程序”->“附件”->“命令提示符”，输入 ipconfig /all 命令出现如图 1-1 内容。00E05C008C7F 为本机网卡号。

```
E:\Documents and Settings\wutao1>ipconfig/all

Windows IP Configuration

    Host Name . . . . . : wutao
    Primary Dns Suffix . . . . . : 
    Node Type . . . . . : Unknown
    IP Routing Enabled. . . . . : No
    WINS Proxy Enabled. . . . . : No

Ethernet adapter 本地连接:

    Connection-specific DNS Suffix  . : 
    Description . . . . . : Realtek RTL8139 Family PCI Ethernet NIC
    Physical Address. . . . . : 00-E0-5C-00-8C-7F
    Dhcp Enabled. . . . . : No
    IP Address. . . . . : 192.168.0.18
    Subnet Mask . . . . . : 255.255.255.0
    Default Gateway . . . . . : 192.168.0.1
```

图 1-1

在桌面打开安装好的 Quartus 8.0，如图 1-2，

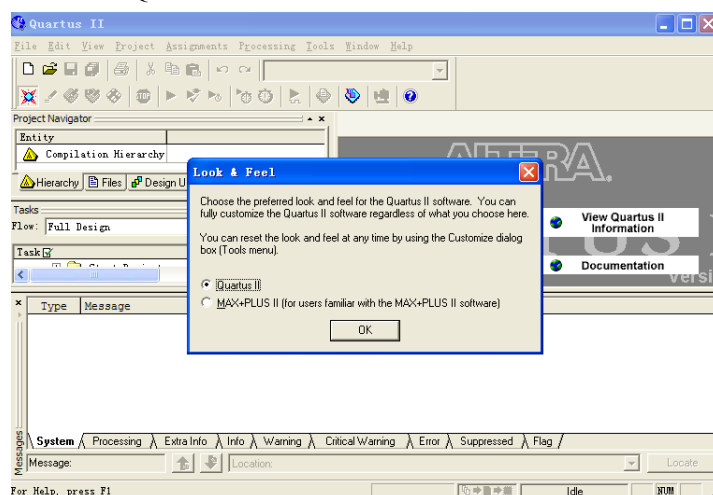


图 1-2

选 Quartus II，点 OK，出现图 1-3。



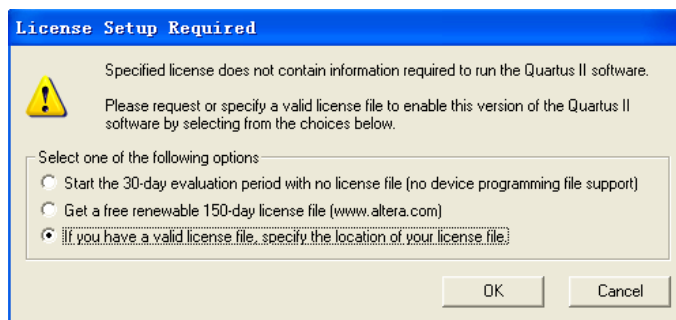


图 1-3

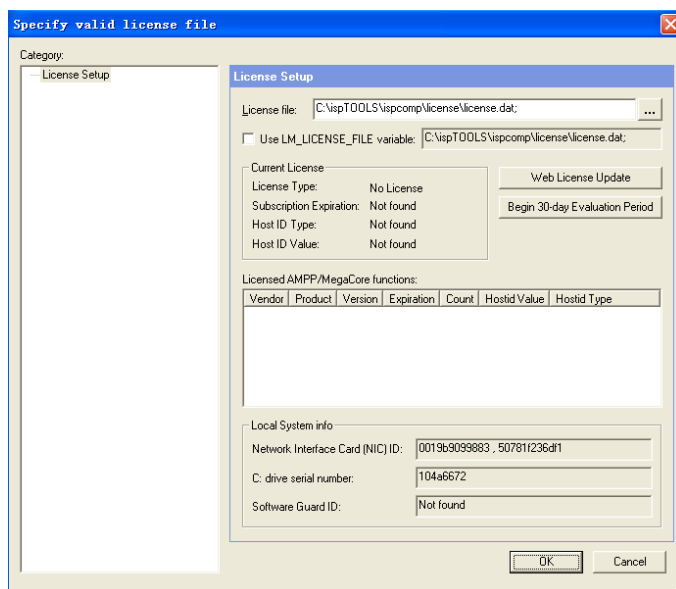


图 1-4

在 License file 路径里，给出破解的 License.Dat 路径，图 1-4。

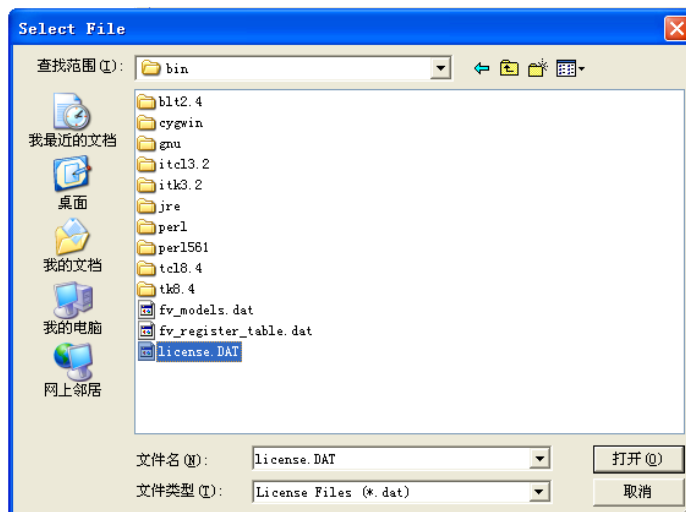


图 1-5

点打开，如图 1-5。

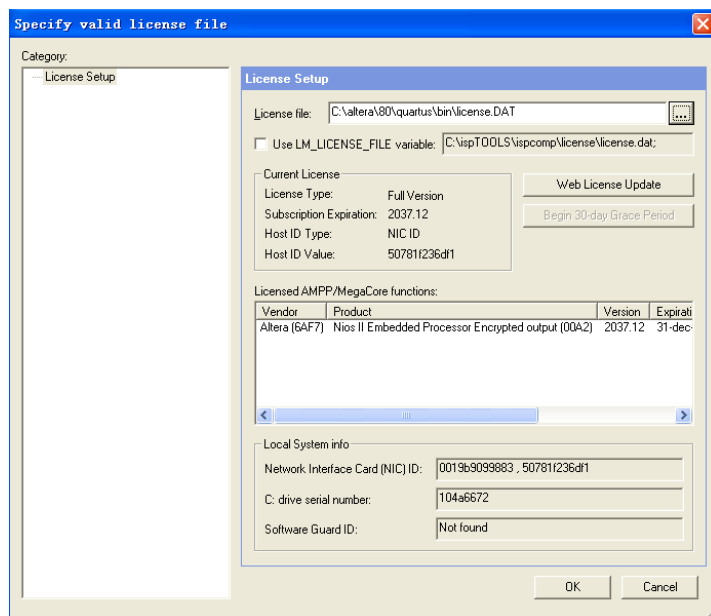



图 1-6

点击 OK，如图 1-6 所示，即安装破解全部完成。

### 1.1.2 Modelsim 软件安装破解

在 32 位操作系统下安装 Modelsim SE 10.2c，双击图标  modelsim-win32-10.2c-se.exe，根据提示步骤安装完成。

下面是破解步骤：首先将破解文件中的 MentorKG.exe 和 patch\_dll.bat 拷贝至安装目录的 win32 下（如：C:\modeltech\_10.2c\win32），如果不在安装目录下运行 patch\_dll.bat 则会找不到 mgls.dll 文件。

然后，“windows 键 +R”，输入 cmd，打开 CMD，然后输入“D:”“cd D:\modeltech\_10.2c\win32”（如果安装在 D 盘），进入到 win32 目录下，输入 patch\_dll.bat，点击运行，产生 license 后，放到任意英文目录下，建议放在“D:\modeltech\_10.2c\win32”（注意一定要使用 cmd 来进行此项操作）；

最后右键点击桌面上的我的电脑---属性---高级，进入环境变量设置，新增用户环境变量 MGLS\_LICENSE\_FILE 指向 LICENSE.dat 的位置，如：D:\modeltech\_10.2c\win32\LICENSE.dat，这里需要着重强调的是，新增环境变量名一定要是 MGLS\_LICENSE\_FILE。破解完成。

### 1.1.3 QuartusII 与 Modelsim 关联设置

双击打开 QuartusII8.0 软件，然后打开所建立的工程以激活菜单栏中的项目。点击 Assignments->EDA Tool Settings...，在弹出框中点击 Simulation，Tool name 选择 ModelSim，Format for output netlist 选择 Verilog。如图 1-7 所示，点击 OK。

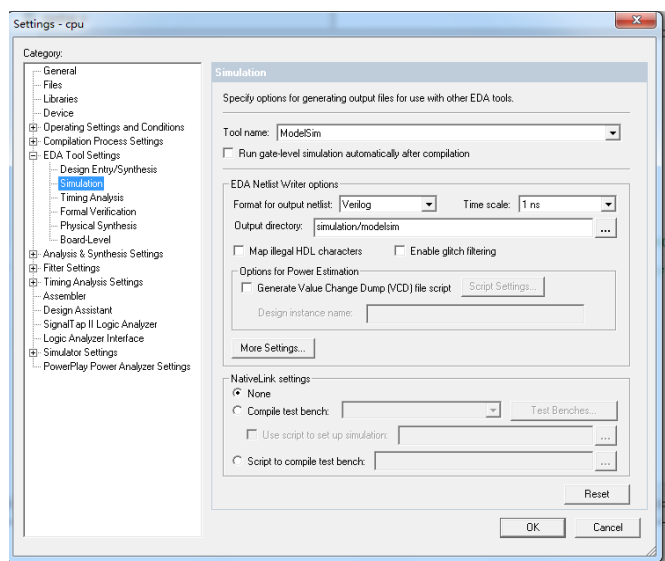


图 1-7

在 Tools->Options...单击，然后单击 General->EDA Tool Options，在右侧窗口中设置 ModelSim 的安装路径。如图 1-8 所示。

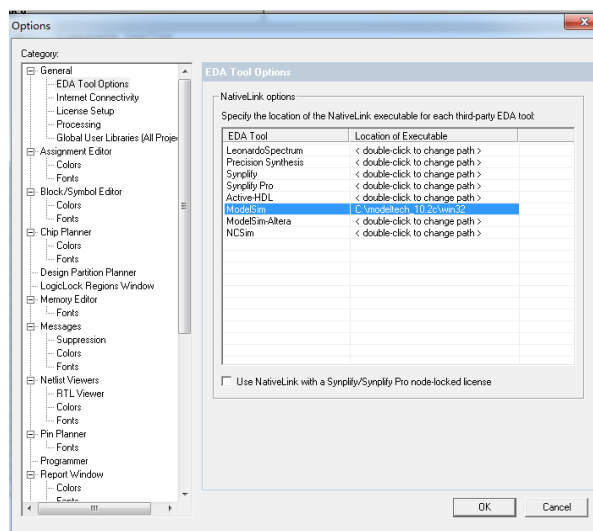


图 1-8

然后通过单击 Tools->Run EDA Simulation Tool->EDA RTL Simulation，如图 1-9 所示。

可自动打开 ModelSim 软件进行仿真。关联完成。

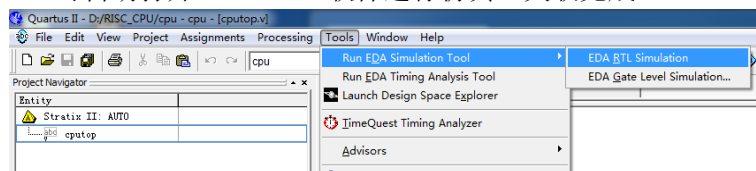


图 1-9

## 1.2 软件使用介绍

ModelSim 软件的使用在数电课中已掌握，在此不做过多介绍。如有需要学生可自行查阅资料学习。

Quartus II 软件是 Altera 公司推出的 FPGA 集成开发环境，是上一代开发环境

MAX+plusII 的更新换代产品, 提供了完全集成且与电路结构无关的设计环境, 使设计者能够方便地完成设计输入、综合、布局布线、仿真测试和器件编程等所有工作, 能够完成可编程片上系统的设计与开发。

QuartusII 软件工具支持原理图、VHDL、Verilog、AHDL (Altera HDL) 以及混合输入方式的设计流程, 可以使用内嵌的综合器或者如 Modelsim 这种第三方仿真器对设计进行功能和时序仿真; QuartusII 的一些高级特征, 如可以在系统设计中使用宏模块、允许第三方 EDIF 文件输入等, 为使用 FPGA 芯片上的各种异构资源, 如存储器、DSP 模块、LVDS 模块等提供了便利; 此外, QuartusII 还可以与 Matlab 等工具结合使用, 进行复杂数字系统的设计与开发。使用 QuartusII 软件能够完成设计流程的所有阶段, 它是一个全面易用的独立解决方案。

Quartus II 软件提供了基于原理图的多层次设计功能, 同时具有适用于各种需要的元件库, 包括基本逻辑元件库 (如基本逻辑门电路、触发器、IO 管脚等)、宏功能元件 (包含几乎所有的 74 系列的器件) 和参数可设置的宏功能模块 LPM 库等。使用原理图设计能够在不具备硬件描述语言等编程技术的条件下进行任意层次的数字系统设计, 能够对任意层次系统的功能进行精确的仿真, 迅速确定电路系统的错误所在并能随时纠正, 能够方便地修正设计方案, 并且能够使用 FPGA 芯片进行任意次系统的硬件测试验证, 为现代数字电路系统提供了良好的设计环境。

基于原理图的 Quartus II 设计使用包括建立工作目录、创建原理图文件、建立工程、工程编译、仿真、引脚绑定和下载; 基于 Verilog HDL 的设计流程; 基于 VerilogHDL 与原理图的混合设计步骤均详见《计算机组成原理实验指导书 2012 版》。下面介绍 QuartusII 软件的高级特征: 宏功能模块调用方法。

在 CPU 结构中, 有许多通用的功能模块, 如算术模块、数据缓冲寄存器、移位寄存器、指令寄存器、地址寄存器、程序计数器、微指令和指令存储器、数据存储器等, 可以直接调用 LPM 库中的宏功能模块实现它们的功能。这极大地方便了 CPU 逻辑系统的构建和设计, 也提高了对计算机原理的认识。本章将针对 CPU 中的一些通用功能模块, 重点介绍 LPM 宏功能模块的调用、参数设置、性能测试、使用方法及相关的测试工具的用法。LPM 是 Library of Parameterized Modules (参数可设置模块库) 的缩写, QuartusII 中提供了不同类型的性能优良的可参数设置的宏功能模块。

事实上, 现在在许多实用设计和开发中, 必须利用宏功能模块才可以使用一些 FPGA 器件中特定模块的硬件功能。例如各类片上存储器、高速硬件乘法器、LVDS 驱动器、嵌入式锁相环 PLL 模块等。这些可以以电路原理图图形或 HDL 硬件描述语言模块形式方面调用的宏功能模块, 使得基于 EDA 技术的电子设计的效率和系统性能有了很大的提高。设计者可以根据实际电路的设计要求, 选择 LPM 库中的适当模块, 并为其设定适当的参数, 就能满足自己的设计需要, 从而在自己的项目中十分方便地调用优秀的电子工程技术人员的硬件设计成果。

### 1.2.1 计数器宏模块调用

LPM 计数器的接口和功能特点是可以通过其参数设置来实现的, 它很容易构成 CPU 中常用的程序计数器等逻辑模块。本书将通过介绍 LPM 计数器 LPN\_COUNTER 的调用和测试的流程, 给出 Quartus II 的 Mega Wizard Plug-In Manager 管理器对同类宏模块的一般使用方法。此流程对后面的 LPM 模块调用具有示范意义, 因此对于之后介绍的其他模块则主要介绍调用方法上的不同之处和不同特性的仿真测试方法。

#### 1. 调用 LPM 计数器及参数设置

流程如下:

(1) 建立原理图为顶层设计的工程。首先创建一个以空原理图文件为顶层设计的工程。原理图文件名可取为 CNT8B, 工程名也取为 CNT8B。为了方便说明, 目标器件选择 Altera 公司生产的 CycloneII 系列的 EP2C5Q208C8N, 并将此工程及空文件存于文件夹 D:\LPM\_MD 中。

(2) 新建一个空的原理图文件, 在空白部分双击, 打开宏功能块调用管理器, 进入图 1-10 所示的元素调用对话框, 单击按钮 MegaWizard Plug-In Manager, 打开如图 1-11 所示的对话框, 选中 Create a new custom megafuction variation 单选按钮, 即定制一个新的模块。

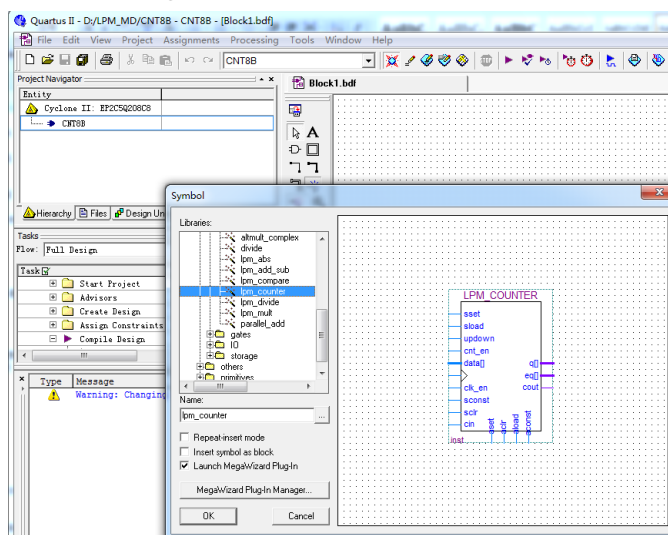


图 1-10 新建一个空的原理图文件

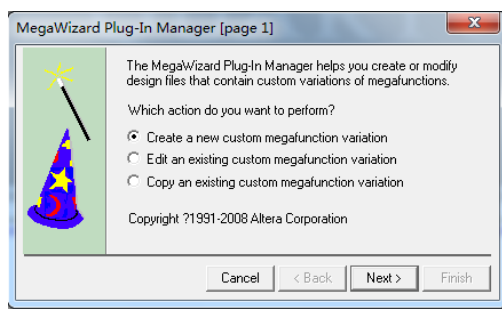


图 1-11 定制新的宏功能模块

单击 Next 按钮后, 将进入图 1-12 所示的对话框, 可以看到左栏中有各类功能的 LPM 模块选项目录。当单击算术项 Arithmetic 后, 立即展示许多 LPM 算术模块选项, 选择计数器 LPM\_COUNTER。再于右上选择 CycloneII 器件系列和 VerilogHDL 语言方式, 最后键入此模块文件名: D:\LPM\_MD\PCNT。

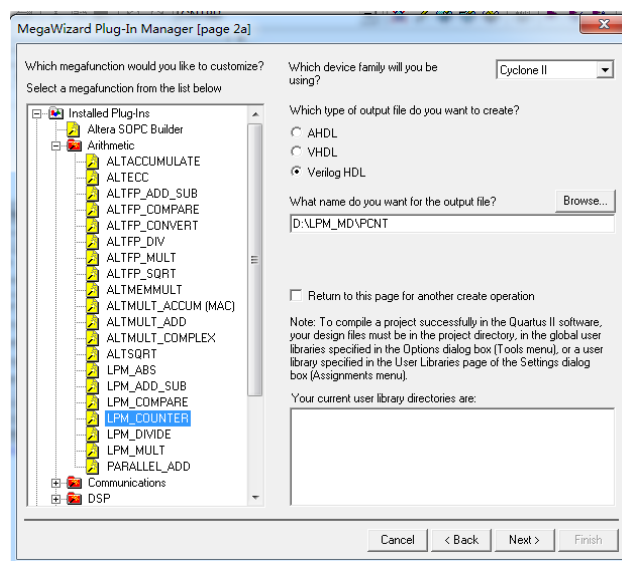


图 1-12 LPM 宏功能模块设定

(3) 单击 Next 按钮后，打开如图 1-13 所示的对话框。在对话框中选择 8 位计数器，再选择“Create an updown input...”使计数器有加减控制功能。

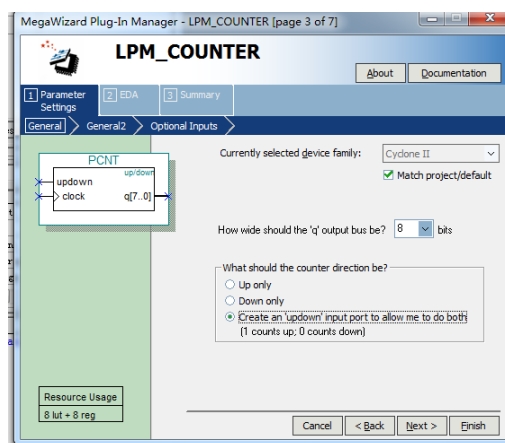


图 1-13 设置 8 位可加减计数器

(4) 再单击 Next 按钮，默认选择 Plain binary，表示是普通二进制计数器，然后选择进位输出 Carry-out。

(5) 再单击 Next 按钮，打开如图 1-14 所示的对话框，在此选择 8 位数据同步加载控制 sload 和异步清 0 控制 aclr。最后结束设置。

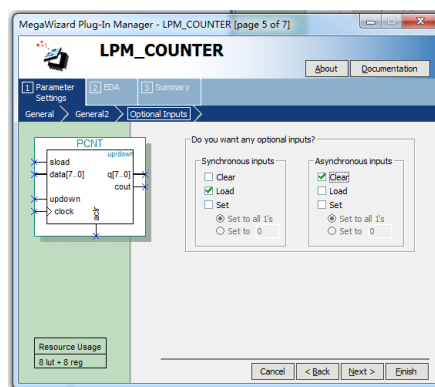


图 1-14 加入 8 位并行数据同步预置功能

以上的流程设置同步生成了 LMP 计数器的 Verilog 文件 PCNT.v, 可被高一层次的 Verilog 程序例化调用。

对于选择 VHDL 代码形式, 整个流程也一样。

## 2. 对计数器进行仿真测试

最后单击图 1-10 右侧对话框左下角的 OK 按钮, 即能将当前设定的计数器模块加入 CNT8B 工程的原理图编辑窗中。为此计数器模块添加了必要的输入输出端口后的电路即如图 1-15 所示。图 1-16 即为此计数器的仿真波形。注意第 1 个加载信号 LOAD 并没能将 d 的数据 36H 加载进计数器; 此时在此信号的有效时间内, 没有 CLK 的上升沿。而第 2 个 LOAD 脉冲恰遇时钟有效边沿, 故将 d 端数据 F9 加载进计数器。显然这是因为在设置时选择 LOAD 为同步控制信号。在计数到 FF 时, cout 出现一个进位信号。注意清 0 信号 RST 是异步控制的, 故在 CLK 非上升沿时也能起到清 0 作用。

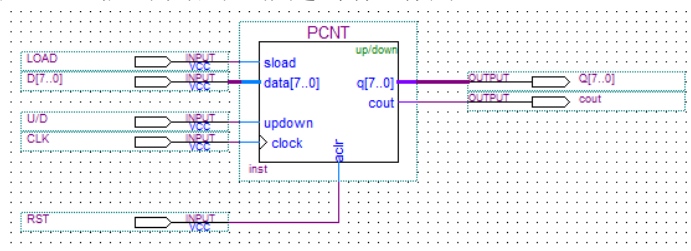


图 1-15 计数器 PCNT 原理图

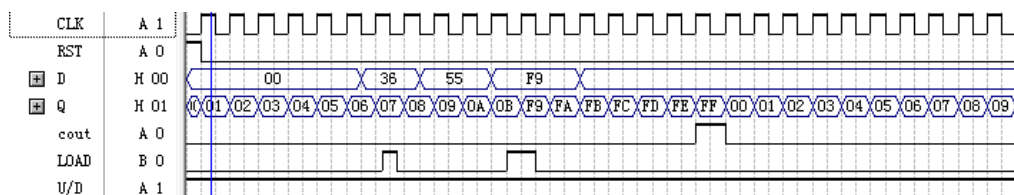


图 1-16 计数器 PCNT 的仿真波形

## 1.2.2 寄存器与锁存器的调用

计算机系统中寄存器与锁存器的应用十分普遍。在计算机系统中直接调用 LPM 库的寄存器或锁存器模块会使设计变得十分便利和高效。

习惯上将时钟边沿控制数据存储的基本时序模块称为触发器, 如 D 触发器; 而将时钟或门控信号的电平控制数据存储的基本时序模块称为锁存器。而以触发器或锁存器构建的数据暂存单元, 多称为寄存器或缓存器, 如地址寄存器、指令寄存器、数据缓存器等。然而由于触发控制的方式和时序特点不同, 这两类寄存器适用场合也不同, 同样, 它们的结构和调用方式也稍有不同。

### 1. 基于 D 触发器的寄存器的调用

基于 D 触发器构建的寄存器最为常用, 其调用和测试步骤如下:

(1) 进入 LPM 库。在进入 LPM 库调用此寄存器之前, 为了便于时序仿真, 验证其功能, 前期工作最好仍然按照 1.2.1 节的流程, 即创建一个原理图工程。然后进入图 1-10 的对话框。在此对话框中选择最下第二排的 Storage 项。在 Storage 的展开菜单中选择触发器类寄存器模块 LPM\_FF。再于右上选择 CycloneII 器件系列和 Verilog HDL 语言方式。最后键入此模块文件名: D:\LPM\_MD\PREG。

(2) 设置参数。单击 Next 按钮后, 进入如图 1-17 所示的对话框。在选择数据位宽为 8, D 触发器类型, 以及选择 Asynchronous inputs 项下的 Clear, 即异步清 0 控制。

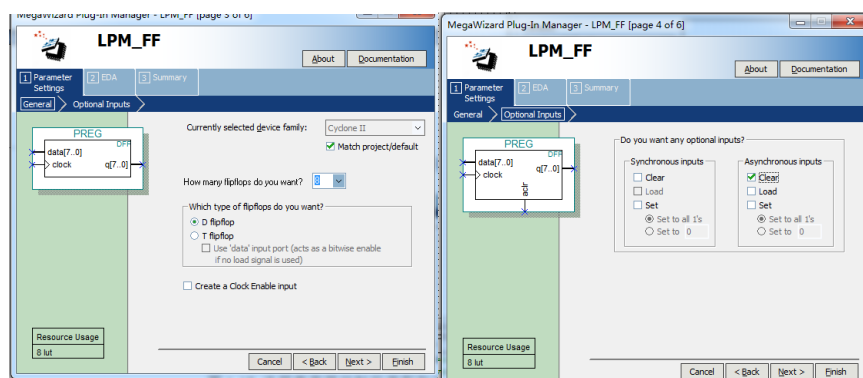


图 1-17 选择寄存器的触发类型和位宽

(3) 功能测试。最后将调入原理图编辑界面的 8 位寄存器 PREG 进行编译和仿真。其仿真波形如图 1-18 所示，波形显示出 PREG 清晰的时钟边沿控制特性，即仅在时钟的上升沿处，输出数据 q 才发生变化，以及 aclr 信号的异步清 0 功能，即无需时钟上升沿也能实现清 0。

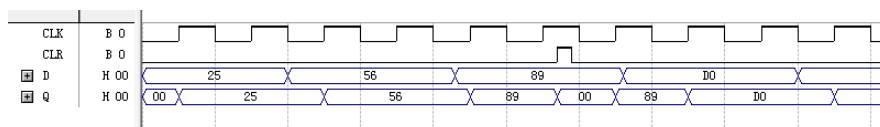


图 1-18 8 位触发器型寄存器的工作时序

## 2. 基于锁存器的寄存器的调用

基于锁存器构建的寄存器的调用流程与以上给出的流程基本相同。取名为 PLATCH 的 8 位寄存器的参数设置对话框及其工作时序波形分别如图 1-19 和 1-20 所示。

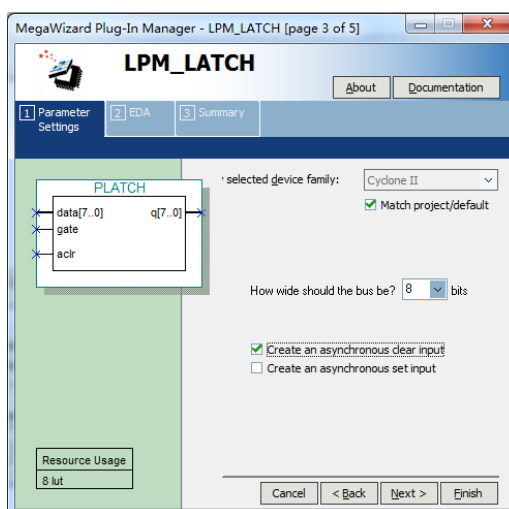


图 1-19 选择寄存器的位宽为 8

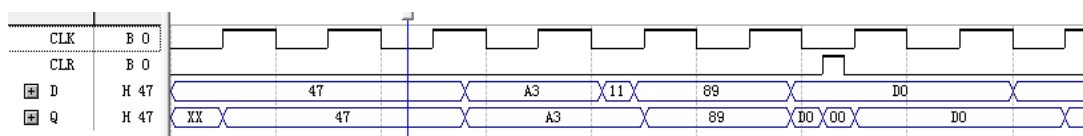


图 1-20 8 位锁存器寄存器的工作时序

图 1-18 和 1-20 的激励信号基本相同。图 1-20 显示，只有在 gate(CLK)为低电平时，输出的数据 q 才不随输入数据 D 的变化而变化。而在 gate(CLK)为高电平时，输入输出端的数据将同步变化。此外，异步复位 CLR 在 CLK 为 0 时也能对输出端 q 清 0。



### 1.2.3 ROM/RAM 宏模块的调用与测试

即使在最初级的计算机系统构建中，ROM 和 RAM 的应用也同样十分广泛。例如，它们可用作数据存储、数据缓冲器、显示缓冲器、程序存储器、微程序微指令存储器等。由于 LPM 存储器是由嵌入于 FPGA 内的高速可编辑存储单元构建的，而且它们与 FPGA 的 JTAG 口有独立的通信通道，这使得 ROM 和 RAM 等存储器在 EDA 设计开发中，调用 LPM 模块类存储器变得十分方便、经济和高效，而性能也最容易得到满足。与传统计算机系统存储器相比，LPM 模块类 ROM 或 RAM 有其独特的优势：

- 容易构成 SOC 单片系统。这是由于 LPM 存储器是嵌入在 FPGA 中的，从而使这个系统可以集成于一个芯片内，而传统计算机系统的存储器都是外挂的。

- 即使对于已定义于 FPGA 中的 LPM\_ROM，也能利用 EDA 软件通过 FPGA 的 JTAG 口对其内容进行读写，实时改变其中的内容。这在系统设计中，无论对于计算机硬件系统的实时在系统控制还是软件程序的调试都十分方便。

- 由于存储器单元处于 FPGA 内部，所以其工作的可靠性和速度都很高。

本节主要介绍 Quartus II 调用 LPM\_ROM/RAM 的方法和相关技术，包括初始化配置文件生成、参数设置、仿真测试，以及存储器内容的在系统编辑方法等。

#### 1. 存储器初始化文件

所谓存储器的初始化文件就是可配置于 RAM 或 ROM 中的数据或程序文件代码，这些代码可以是普通数据，也可以是计算机程序或微程序。而与传统 RAM 不同的是，LPM\_RAM 也可以像 ROM 一样在启动系统后，为 RAM 加载数据文件。这些数据文件统称为初始化文件。从而 LPM\_RAM 可以同时兼任 RAM 和 ROM 的功能。

在 EDA 设计中，通过 EDA 工具设计或特定的存储器中的代码文件必须由 EDA 软件在统一编译的时候自动调入。所以此类代码文件，即初始化文件的格式必须满足一定的要求。以下介绍两种格式的初始化文件及生成方法，其中 Memory Initialization File(.mif)格式和 Hexadecimal(Inter-Format) File (.hex) 格式是 Quartus II 能直接调用的两种初始化文件的格式，而更具一般性的.dat 格式文件可通过 Verilog / VHDL 语言直接调用。

##### (1) .mif 格式文件

生成.mif 格式的文件有以下多种方法：

① 直接编辑法。首先在 Quartus II 中打开 mif 文件编辑窗，即选择 File→New 命令，并在 New 窗中选择 Memory File 栏的 Memory Initialization File 项，单击 OK 按钮后产生 mif 数据文件大小选择窗口。在此根据存储器的地址和数据宽度选择参数。如果对应地址线为 8 位，选 Number 为 256；对应数据宽为 8 位，选择 Word size 为 8 位。按 OK 按钮，即出现如图 1-21 所示的 mif 数据表格。然后可以在此键入数据。表格中的数据格式可通过右击窗口边缘的地址数据所弹出的窗口中选择，此表中任一数据对应的地址为左列与顶行数之和。填完此表后，选择 File→Save As 命令，保存此数据文件，如取名为 data8X8.mif。

Addr	+0	+1	+2	+3	+4	+5	+6	+7
00	80	83	89	86	8C	8F	92	95
08	00	00	00	00	00	00	00	00
10	00	00	00	00	00	00	00	00
18	00	00	00	00	00	00	00	00
20	00	00	00	00	00	00	00	00
28	00	00	00	00	00	00	00	00
30	00	00	00	00	00	00	00	00
38	00	00	00	00	00	00	00	00
40	00	00	00	00	00	00	00	00
48	00	00	00	00	00	00	00	00
50	00	00	00	00	00	00	00	00
58	00	00	00	00	00	00	00	00
60	00	00	00	00	00	00	00	00
68	00	00	00	00	00	00	00	00
70	00	00	00	00	00	00	00	00
78	00	00	00	00	00	00	00	00
80	00	00	00	00	00	00	00	00
88	00	00	00	00	00	00	00	00

图 1-21 mif 文件编辑器

② 文件直接编辑法。即使用 Quartus II 以外的编辑器编辑 mif 文件，其格式如例 1-1 所示。其中地址和数据都为十六进制，冒号左边是地址值，右边是对应的数据，并以分号结尾。存盘以 mif 为后缀，如取名 data8X8.mif。

```
[例 1-1]
DEPTH=256;           数据深度，即存储的数据个数
WIDTH=8;             : 输出数据宽度
ADDRESS_RADIX = HEX; : 地址数据类型
DATA_RADIX = HEX;    : 存储数据类型
CONTENT              : 此为关键词
BEGIN                : 此为关键词
0000      :   0080
0001      :   0083
0002      :   0086
...(数据略去)
00FE      :   0079
00FF      :   007C;
END;
```

## (2) .hex 格式文件

建立 .hex 格式文件也有多种方法，例如也可类似以上介绍的那样，在 New 窗口中选择 Hexadecimal(Intel-Format) File 选项，加入必要的数后，以 .hex 格式文件存盘即可。或是用诸如单片机编译器来产生，方法是利用汇编程序编辑器将数据编辑于汇编程序中，然后用汇编编译器生成 .hex 格式文件。这里提到的 .hex 格式文件生成的第二种方法很容易应用到 8086 或 51 单片机 SOC 系统设计中。

## 2. ROM 宏模块的调用

基本流程与以上的计数器调用相同。为测试方便，首先仍打开一个原理图编辑器，再存盘，文件取名设为 ROMMD，并将其创建成工程。在此工程的原理图编辑窗，进入图 1-10 所示的 Symbol 对话框。单击左下的 Mega Wizard Plug-In Manager 管理器按钮，进入图 1-12 所示的 LPM 模块编辑调用窗。在这里的左栏选择 Memory Compiler 项下的单口 ROM 模块“ROM: 1-PORT”。文件名取为 ROM1P.v，设存在 D:\LPM\_MD 中。

单击 Next 按钮后打开如图 1-22 所示的对话框。选择数据位 8 和数据深度 256，即 8 位数据线。对应 CycloneII，存储器构建方式选择 M4K，及选择默认的单时钟方式。

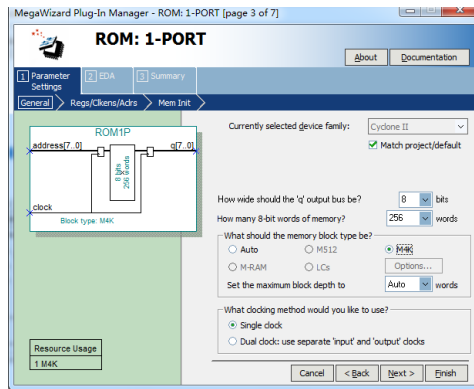


图 1-22 调用单口 LPM\_RAM

单击 Next 按钮，直至进入如图 1-23 所示的对话框。在此对话框的 Do you want to specify the initial content of the memory 栏中选中“Yes, use this file for the memory content date”，并单

击 Browse 按钮，选择指定路径上的文件初始化文件 DATA8X8.mif。

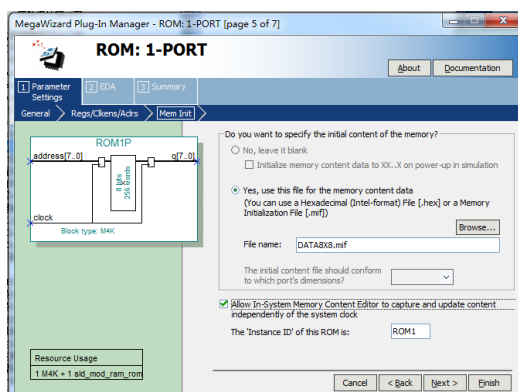


图 1-23 为 LPM\_ROM 设置初始化文件

此文件假设就是图 1-21 所示的文件。在图 1-23 下面若选中“Allow In-System Memory...”复选框，并在“The Instance ID of this ROM is”文本框中输入 ROM1，作为此 ROM 的 ID 名称。通过这个设置，可以允许 Quartus II 通过 JTAG 口对下载于 FPGA 中的此 ROM 进行“在系统”测试和读写。如果需要读写多个不同的 LPM\_ROM，则此 ID 号 ROM1 即作为此 ROM 的识别名称，而在“在系统”读写编辑中作辨别。

### 3. ROM 宏模块的测试

最后单击 Finish 按钮后完成了 ROM 的定制。调入顶层原理图后，连接好的端口引脚如图 1-24 所示。可以有两种方法来测试此 LPM\_ROM 的功能，一种是基于软件的波形仿真，另一种是基于 FPGA 硬件的“在系统”数据读写编辑测试（本书不作介绍）。

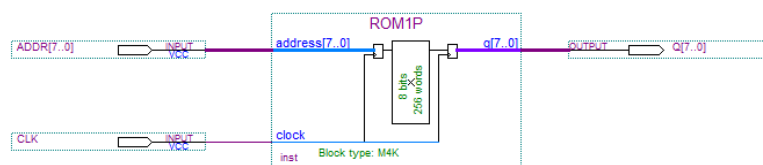


图 1-24 LPM\_ROM 的测试原理图

图 1-25 所示的波形就是此 LPM\_ROM 的仿真波形。从波形可以看出，随着时钟脉冲的出现和地址数据的递增，被读出的数据在每一时钟上升沿后出现在输出端口 q 上。对应地址 00,01,02, ... 的输出数据分别是 80,83,86, ...。这些数据显然与图 1-21 显示的 mif 文件的数据吻合。这说明，在编译后，Quartus II 已成功将指定的初始化文件加载于 ROM 中了。需要注意的是，如果设置不当，这种正确的加载并非总能发生，这也是进行仿真测试的目的之一。

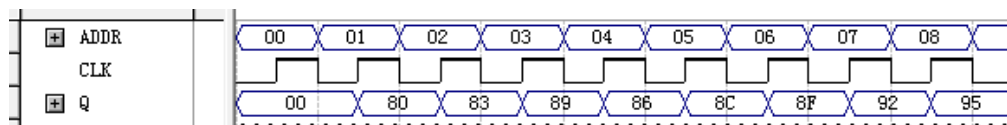


图 1-25 LPM\_ROM 的仿真波形图

### 4. RAM 宏模块的调用

按照第 2 节的流程，调用 LPM\_RAM。进入图 1-12 所示对话框，在其左栏选择 Memory Compiler 项下的单口 RAM 模块“RAM: 1-PORT”。文件名取为 RAM1P.v，设存在 D:\LPM\_MD 中。再单击 Next 按钮后打开如图 1-26 所示的对话框，进行参数设置。仍选择数据位为 8 和数据深度为 256；存储器构建方式选择 M4K。对于 RAM 来说要特别注意控制时钟方式的设置。在这里选择图下方的双时钟控制方式：Dual clock。

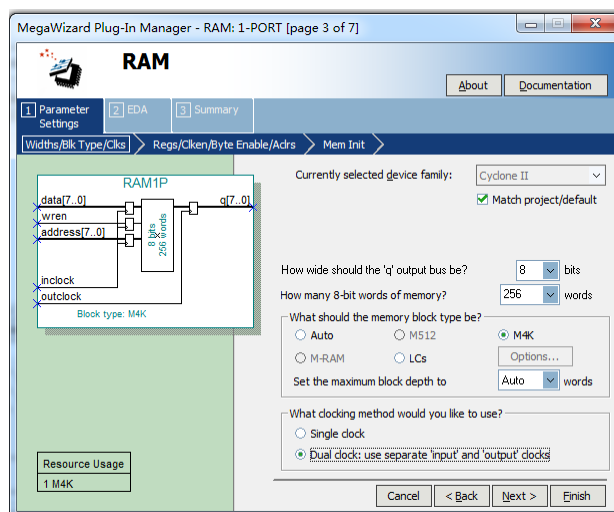


图 1-26 设置单口 LPM\_RAM 的结构参数

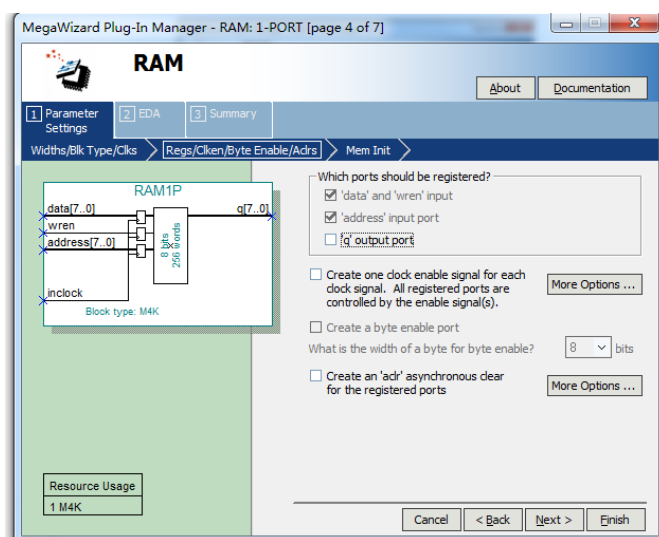
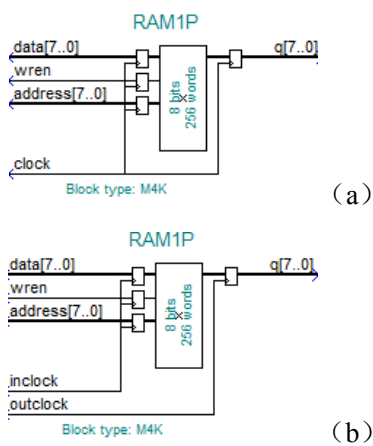


图 1-27 设定 RAM 仅输入时钟控制



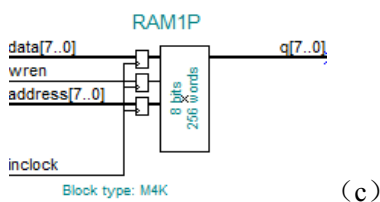


图 1-28 不同方式的端口控制时钟

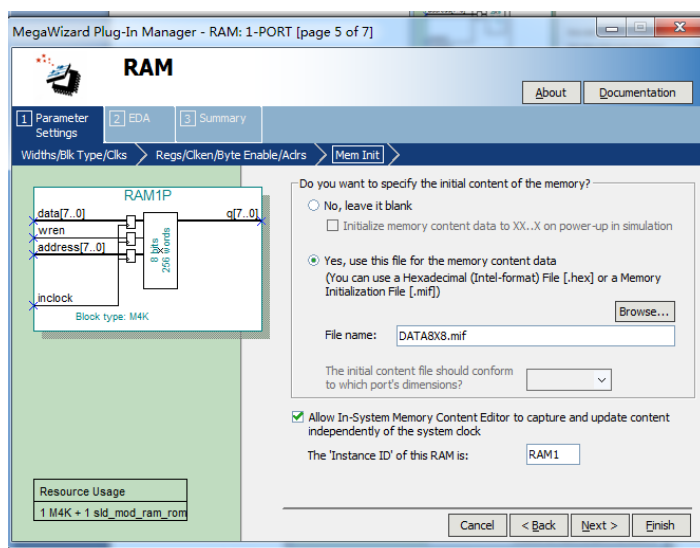


图 1-29 设定初始化文件和允许在系统编辑

单击 Next 按钮后，进入如图 1-27 的对话框。消去“qoutput port”选项，即消除了输出端口的锁存器，使得此模块只有输入信号的控制时钟。

其实通过图 1-26 和图 1-27 的不同选择，可以使 RAM 的输入输出端口有三种不同的时钟控制方式（如图 1-28 所示）。与图 1-28（c）的电路相比，（a）、（b）两个图中的 RAM 的输出口都含时钟同步控制的锁存器，所以都会导致输出数据比前者至少延迟一个时钟周期。

接下去就是进入如图 1-29 所示的对话框。在此对话框同样可以像配置 ROM 那样选择加入初始化文件，即在 Do you want to specify the initial content of the memory 栏中选中“**Yes, use this file for the memory content data**”，并单击 Browse 按钮，选择指定路径上的文件初始化文件 DATA8X8.mif。其实，对于 RAM 来说，不一定加初始化文件，但在许多情况下，这样的选择会对某些功能需求带来诸多便利。在此，如果选择调入初始化文件，则系统于每次开启后，将像对待 ROM 一样自动向此 LPM\_RAM 加载该 mif 文件。

至于对图 1-29 对话框以下的 Allow In-System Memory 选择，其功能已于第 2 节介绍了。设在此对话框设置 RAM 的 ID 名称为 RAM1。

## 5. RAM 宏模块的测试

最后单击 Finish 按钮后即完成了 RAM 的定制。调入顶层原理图后，连接好端口引脚后的电路如图 1-30 所示。接下去的任务就是对图 1-30 所示的 RAM 模块进行测试。

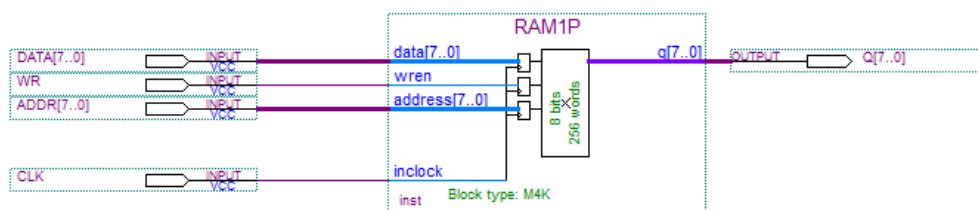


图 1-30 在原理图上连接好的 RAM 模块，以待测试

图 1-31 是此模块的仿真波形图。在 RAM 写允许控制 WREN 的高电平和低电平对应的两个不同电平段，分别安排的地址信号 ADR 都是从 0 开始递增。这样有利于了解随着地址的变化，数据的读写情况。

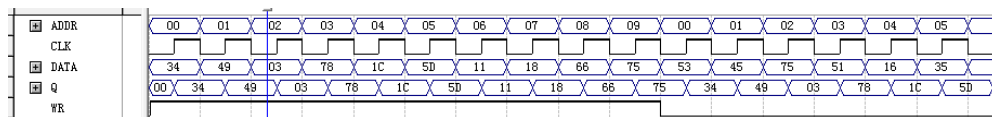


图 1-31 图 1-30 电路的仿真波形

由图 1-31 可见，在写允许 WREN=1 时段，随着地址的递增和时钟上升沿的出现，将输入数据 DATA 端口给出的一系列数据 34、49、03、78 等，同步写入此时地址的对应单元；而就在同时，输出口 q 出现的数据，恰好是新写入的数据。

而在写允许 WREN=0 时段，由于地址 ADR 的输入仍旧从 0 开始，随着地址的递增和时钟上升沿的出现，从 RAM 中读出的数据 34、49、03、78 等，完全与写入的数据相同。显然，此 RAM 的各项功能符合要求。

图 1-32 的时序波形来自图 1-28 (a)、(b) 中 RAM 模块的仿真。与波形图 1-31 相比，其输出数据要落后一个时钟周期，这显然是由于输出口多了一个 8 位锁存器所致。

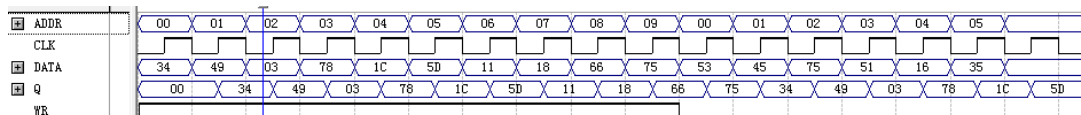


图 1-32 图 1-28 左侧电路的仿真波形，数据输出延迟一个时钟周期

## 1.3 硬件平台介绍

### 1.3.1 可编程逻辑器件与 FPGA 简介

FPGA (Field Programmable Gate Array)，即现场可编程门阵列，它是在可编程阵列逻辑 PAL(Programmable Array Logic)、门阵列逻辑 GAL(Gate Array Logic)、可编程逻辑器件 PLD(Programmable Logic Device)等可编程器件的基础什么是 FPGA 上进一步发展的产物。作为专用集成电路 ASIC (Application Specific Integrated Circuit) 领域中的一种半定制电路出现，它既解决了定制电路的不足，又克服了原有可编程器件门电路数有限的缺点。FPGA 能完成任何数字器件的功能，上至高性能 CPU,下至简单的 74 系列电路，都可以用 FPGA 来实现。FPGA 如同一张白纸或是一堆积木，工程师可以通过传统的原理图什么是输入法，或是硬件描述语言自由设计一个数字系统。通过软件仿真，我们可以事先验证设计的正确性。在 PCB 完成以后，还可以利用 FPGA 的在线修改能力，随时修改设计而不必改动硬件电路。使用 FPGA 来开发数字电路，可以大大缩短设计时间，减少 PCB 面积，提高系统的可靠性。

### 1.3.2 FPGA 的特点

FPGA 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。它兼容了 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。与门阵列等其它 ASIC 相比，它又具有设计周期短、设计制造成本低、开发工具先进、标准产品无需测试、质量稳定以及可实时在线检验等优点，因此被广泛应用于产品的原

型设计和产品生产(一般在 10,000 件以下)之中。几乎所有应用门阵列、PLD 和中小规模通用数字集成电路的场合均可应用 FPGA。

### 1.3.3 课程设计的实现平台

由于 CPU 的设计复杂度高,方案修改频繁,且需要测试验证其可靠性,因此课程设计选择 FPGA 作为实验样机的平台,示例的仿真器件为 CycloneII 系列 EP2C5Q208C8N 芯片,通过了功能和时序仿真,开发环境采用 Altera 公司的 Quatrus II 8.0 集成开发环境。

硬件逻辑描述可以采用多种方法,主要有原理图输入、VHDL 描述语言、VerilogHDL 硬件描述语言等。最后两章采用了原理图设计和 VerilogHDL 硬件描述语言设计两种方式实现了示例模型机。



## 第2章 HDL 语言基础

本章重点介绍了 HDL 硬件描述语言的语法，包括语言结构、数据类型、过程描述语句以及代码书写规范等。

### 2.1 VerilogHDL 基本程序结构

用 VerilogHDL 描述的电路就是该电路的 VerilogHDL 模型。Verilog 模块是 Verilog 的基本描述单位。模块描述某个设计的功能或结构以及与其他模块通信的外部接口，一般来说一个文件就是一个模块，但也可以将多个模块放入一个文件中。模块是并行运行的，通常需要一个高层模块通过调用其他模块的实例来定义一个封闭的系统，包括测试数据和硬件描述。

一般的模块结构如下：

```
module <模块名> (<端口列表>)
```

```
<说明部分>
```

```
<语句>
```

```
Endmodule
```

其中，说明部分用来指定数据对象为寄存器型、存储器型、线型等，可以分散于模块的任何地方，但是变量、寄存器、线网和参数等的说明必须在使用前出现。语句用于定义设计的功能和结构，可以是 initial 语句、always 语句、连续赋值语句或模块实例。

下面给出一个简单的 Verilog 模块，实现了一个二选一选择器。

**[例 2-1]** 二选一选择器（见图 2-1）

```
module muxtwo(a, b, s1, out);
```

```
    input a, b, s1;
```

```
    output out;
```

```
    reg out;
```

```
    always @(s1 or a or b)
```

```
        if(!s1) out = a;
```

```
        else out = b;
```

```
endmodule
```

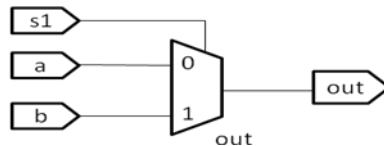


图 2-1 二选一选择器

模块的名字是 muxtwo，模块有 4 个端口，3 个输入端口 a、b 和 s1，一个输出端口 out。由于没有定义端口的位数，所有端口大小都默认为 1 位；由于没有定义端口 a、b、s1 的数据类型，这 3 个端口都默认为线网型数据类型。输出端口 out 定义为 reg 类型。如果没有明确的说明，则端口都是线网型的，且输入端口只能是线网类型。

这里特别指出，模块端口是指模块与外界交互信息的接口，包括以下 3 种类型：

- (1) input: 模块从外界读取数据的接口，在模块内不可写。
- (2) output: 模块往外界送出数据的接口，在模块内不可读。
- (3) inout: 可读取数据也可以送出数据，数据可双向流动。

这 3 种端口类型在后续章节中均有使用，请读者留意其使用方法，尤其是 inout 端口的使用方法。

### 2.2 Verilog HDL 语言的数据类型和运算符

#### 1. 标识符



标识符可以是一组字母、数字、\_下划线和\$符号的组合，且标识符的第一个字符必须是字母或者下划线。另外，标识符是可以区别大小写的。下面给出标识符的几个例子。

**traffic\_state**

**\_rst**

**clk\_10kHz**

**MODULE**

**P\_1\_02**

需要注意的是，Verilog HDL 定义了一系列保留字，叫作关键字。通常关键字都是由小写字母构成的，因此在实际应用中，建议将不确定是否是保留字的标识符首字母大写。例如：标识符 if（关键字）与标识符 If 是不同的。

## 2. 数据类型

数据类型用来表示数字电路硬件中的数据存储和传送元素。Verilog HDL 中总共有约 20 种数据类型，本章只介绍 3 个常用的数据类型：wire 型、reg 型和 memory 型。

### (1) wire 型

wire 型数据常用来表示以 assign 关键字指定的组合逻辑信号。Verilog 程序模块中输入、输出信号类型默认为 wire 型。Wire 型信号可以用作方程式的输入，也可以用作 assign 语句或者实例元件的输出。

wire 型信号的定义格式如下：

**wire [n-1 : 0] 数据名 1, 数据名 2, ..., 数据名 N;**

这里，总共定义了 N 条线，每条线的位宽为 n。下面给出几个例子。

**wire [7 : 0]a, b, c;           // a、b、c 都是位宽为 8 的 wire 型信号**

**wire d;                       //d 是位宽为 1 的 wire 型信号**

### (2) reg 型

reg 是寄存器数据类型的关键字。寄存器是数据存储单元的抽象，通过赋值语句可以改变寄存器存储的值，其作用相当于改变触发器存储器的值。reg 型数据常用来表示 always 模块内的指定信号，代表触发器。通常在设计中要由 always 模块通过使用行为描述语句来表达逻辑关系。在 always 模块内被赋值的每一个信号都必须定义为 reg 型，即赋值操作符的右端变量必须是 reg 型。

reg 型信号的定义格式如下：

**reg [n-1 : 0] 数据名 1, 数据名 2, ..., 数据名 N;**

这里，总共定义了 N 个寄存器变量，每条线的位宽为 n。下面给出几个例子。

**reg [7 : 0]a, b, c;           // a、b、c 都是位宽为 7 的 reg 型信号**

**reg d;                       //d 是位宽为 1 的 reg 型信号**

reg 型数据的默认值是未知的。Reg 数据可以为正值或负值。但当一个 reg 型数据是一个表达式中的操作数时，它的值被当作无符号值，即正值。如果一个 4 位的 reg 型数据被写入-1，在表达式中运算时，其值被认为是+15。

reg 型和 wire 型的区别在于：reg 型保持最后一次的赋值，而 wire 型则需要持续的驱动。

### (3) memory 型

Verilog 通过对 reg 型变量建立数组来对存储器建模，可以描述 RAM、ROM 和寄存器数组。数组中的每一个单元通过一个整数索引进行寻址。memory 型通过扩展 reg 型数据的地址范围来达到二维数组的效果，其定义的格式如下：

**reg [n-1 : 0]存储器名 [m-1 : 0];**

其中, `reg [n-1 : 0]` 定义了存储器中每一个存储单元的大小, 即该存储器单元是一个  $n$  位位宽的寄存器; 存储器后面的 `[m-1 : 0]` 则定义了存储器的大小, 即该存储器中有多少个这样的寄存器。例如:

```
reg [17 : 0] ROMA [1023 : 0];
```

这个例子定义了一个存储位宽为 18 位, 存储深度为 1024 的一个寄存器。该寄存器的地址范围是 0 到 1024。

**需要注意的是: 对存储器进行地址索引的表达式必须是常数表达式。**

尽管 memory 型和 reg 型数据的定义比较接近, 但二者还是有很大区别的。例如, 一个由  $n$  个 1 位寄存器构成的存储器是不同于一个  $n$  位寄存器的。

```
reg [n-1 : 0] rega;      //一个 n 位的寄存器
```

```
reg memb [n-1 : 0];    //一个由 n 个 1 位寄存器构成的存储器组
```

一个  $n$  位的寄存器可以在一条赋值语句中直接进行赋值, 而一个完整的存储器则不行。

```
rega=0;                //合法赋值
```

```
memb=0;                //非法赋值
```

如果要对 memory 型存储单元进行读写必须要指定地址。例如:

```
memb[0]=1;              //将 memb 中的第 0 个单元赋值为 1
```

```
reg[3 : 0] ROMB[1 : 4];  //将 ROMB 的 4 个单元分别进行赋值
```

```
ROMB[1]=4'h0;
```

```
ROMB[2]=4'h7;
```

```
ROMB[3]=4'h9;
```

```
ROMB[4]=4'he;
```

### 3. 常量

Verilog HDL 有下列 4 种基本的数值。

0: 逻辑 0 或“假”。

1: 逻辑 1 或“真”。

x: 未知。

z: 高阻。

其中,  $x$ 、 $z$  是不区分大小写的。Verilog HDL 中的数字由这 4 类基本数值表示。

Verilog HDL 中的常量分为 3 类: 整数型、实数型以及字符串型。下划线符号“`_`”可以随意用在整数和实数中, 没有实际意义, 只是为了提高可读性。例如: 56 等效于 5\_6。通常用 parameter 来定义变量。

#### (1) 整数

整数型可以按如下两种方式书写: 简单的十进制数格式以及基数格式。

① 简单的十进制格式。简单的十进制数格式的整数定义为带有一个“+”或“-”操作符的数字序列。下面是这种简易十进制形式整数的例子。

100 十进制数 100。

-100 十进制数-100。

简单的十进制数格式的整数值代表一个有符号的数, 其中负数可以使用两种补码形式表示。例如, 32 在 6 位二进制形式中表示为 100000, 在 7 位二进制格式中为 0100000, 这里最高位 0 表示符号位; -15 在 5 位二进制中的形式为 10001, 最高位 1 表示符号位, 在 6 位二进制中为 110001, 最高位 1 位符号扩展位。

② 基数表示格式。基数格式的整数格式如下。

[长度]'基数数值

长度为常量的位长，基数可以是二进制、八进制、十进制、十六进制之一。数值是基于基数的数字序列，且数值不能为负数。下面是一些具体实例。

**6'b9** 6 位二进制数 9。

**10'o9** 10 位八进制数 9。

**16'd9** 16 位十进制数 9。

(2) 实数

实数可以用下列两种形式定义。

③ 十进制计数法

**3.0。**

**1234.567。**

④ 科学计数法

**345.12e2** 其值为 34512。

**9E-3** 其值为 0.009。

实数的科学计数法中，e 与 E 相同，实数通常用于仿真。

(3) 字符串

字符串是双引号内的字符序列。字符串不能分成多行书写。例如：

**“counter”**

用 8 位 ASCII 值表示的字符可看作是无符号整数，因此字符串是 8 位 ASCII 值的序列。为存储字符串“counter”，变量需要 56 位。

**reg[1 : 8\* 7] Char;**

**Char=”counter”;**

(4) parameter 型

在 Verilog HDL 中用 parameter 来定义常量，即用 parameter 来定义一个标识符表示一个常数。采用该类型可以提高程序的可读性和可维护性。

parameter 型信号的定义格式如下。

**parameter 参数名 1=数据名 1;**

下面给出一些例子。

**parameter s1=0;**

**parameter s0=2'b00, s1=2'b01, s2=2'b10, s3=2'b11;**

#### 4. 运算符和表达式

在 Verilog HDL 语言中运算符所带的操作数是不同的，按其所带操作数的个数可以分为以下 3 种。

(1) 单目运算符：带一个操作数，且放在运算符的右边。

(2) 双目运算符：带两个操作数，且放在运算符的两边。

(3) 三目运算符：带三个操作数，且被运算符间隔开。

Verilog HDL 语言参考了 C 语言中大多数算符的语法和句义，运算范围很广，其运算符按其功能分为下列 9 类：

1) 基本算术运算符

在 Verilog HDL 中，算术运算符又称为二进制运算符，有下列 5 种。

(1) + 加法运算符或正值运算符，例如：s1+s2; +5。

(2) -减法运算符或负值运算符，例如：s1-s2; -5。

(3) \*乘法运算符，例如：s1\*5。

(4) / 除法运算符，例如：s1/8。

(5) %模运算符，例如：s1%8。

在进行整数除法时，结果值要略去小数部分。在取模运算时，结果的符号位和模运算第一个操作数的符号位保持一致。例如：

运算表达式	结果	说明
12.5/3	4	结果为 4，小数部分省去
12%4	0	整除，余数为 0
-15%2	-1	结果取第一个数的符号，所以余数为-1
13/-3	1	结果取第一个数的符号，所以余数为 1

**注意：**在进行基本算术运算时，如果某一操作数有不确定的值 **X**，则运算结果也是不确定值 **X**。

## 2) 赋值运算符

赋值运算分为连续赋值和过程赋值两种。

(1) 连续赋值。连续赋值语句只能用来对线网型变量进行赋值，而不能对寄存器变量进行赋值，其基本的语法格式如下：

**线网型变量类型 [线网型变量位宽] 线网型变量名；**

**assign # (延时量) 线网型变量名=赋值表达式；**

**例如：**

**wire a,b,c;**

**assign a=1'b1;**

**assign a=b+c;**

一个线网型变量一旦被连续赋值语句赋值之后，赋值语句右端赋值表达式的值将持续对被赋值变量产生连续驱动。只要右端表达式任一个操作数的值发生变化，就会立即触发对被赋值变量的更新操作。

在实际使用中，连续赋值语句有下列几种应用。

①对标量线网型赋值。

**wire a,b;**

**assign a=b;**

②对矢量线网型赋值。

**wire [ 7: 0] a,b;**

**assign a=b;**

③对矢量线网型中的某一位赋值。

**wire [7 : 0] a,b;**

**assign a[3]=b[1];**

④对矢量线网型中的某几位赋值。

**wire [7:0] a,b;**

**assign a[3: 0]=b[7: 4];**

⑤对任意拼接的线网型赋值。

**wire a,b;**

**wire [1:0] c;**

**assign c= {a ,b};**

(2) 过程赋值。过程赋值主要用于两种结构化模块（initial 模块和 always 模块）中的赋值语句。在过程块中只能使用过程赋值语句，不能在过程块中出现连续赋值语句，同时过程赋值语句也只能用在过程赋值模块中。

过程赋值语句的基本格式为

**<被赋值变量><赋值操作符><赋值表达式>**

其中，<赋值操作符>是“=”或“<=”，它分别代表了阻塞赋值和非阻塞赋值类型。

下面通过 5 个例题来说明两种赋值方式的不同。这 5 个例题的设计目标都是实现 3 位移寄存器，它们分别采用了阻塞赋值方式和非阻塞赋值方式。

**[例 2-2]** 阻塞赋值方式描述的移位寄存器 1。

```
module block1(
    input clk,
    input D,
    output Q0,
    output Q1,
    output Q2
);
    reg Q0, Q1, Q2;
    always @(posedge clk)
    begin
        Q2=Q1; //注意赋值语句的顺序
        Q1=Q0;
        Q0=D;
    end
endmodule
```

综合结果如图 2-2 所示。

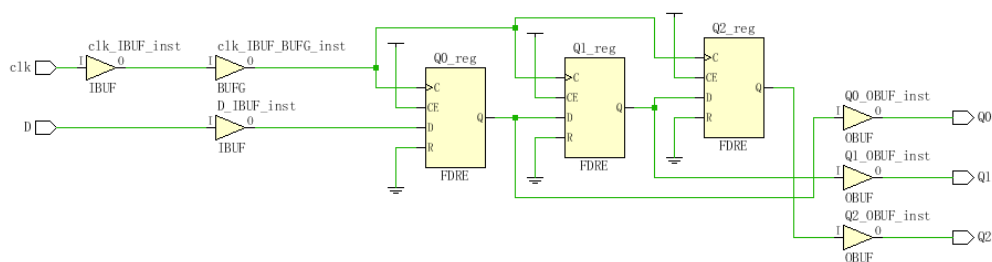


图 2-2

**[例 2-3]** 阻塞赋值方式描述移位寄存器 2。

```
module block2(
    input clk,
    input D,
    output Q0,
    output Q1,
    output Q2
);
```

```

reg Q0, Q1, Q2;
always @(posedge clk)
begin
    Q1=Q0; //该句与下句的顺序与例 2-2 颠倒
    Q2=Q1;
    Q0=D;
end
endmodule
    
```

综合结果如图 2-3 所示。

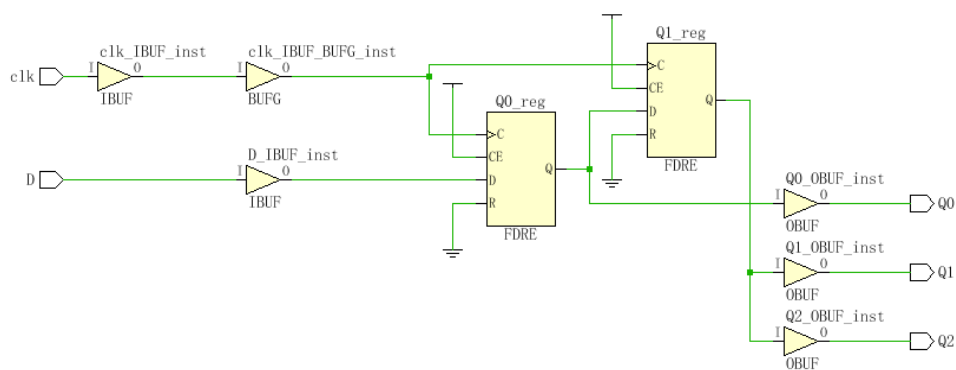


图 2-3

**[例 2-4]** 阻塞赋值方式描述的移位寄存器 3。

```

module block3(
    input clk,
    input D,
    output Q0,
    output Q1,
    output Q2
);
reg Q0, Q1, Q2;
always @(posedge clk)
begin
    Q0=D; //3 条赋值语句的顺序与例 2-2 完全颠倒
    Q1=Q0;
    Q2=Q1;
end
endmodule
    
```

综合结果如图 2-4 所示。

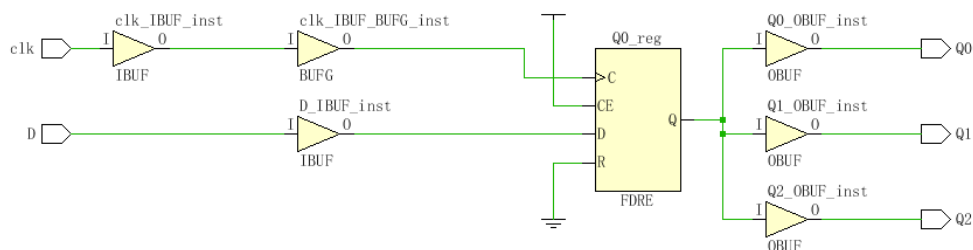


图 2-4

**[例 2-5]** 非阻塞赋值方式描述的移位寄存器 1。

```
module non_block1(
    input clk,
    input D,
    output Q0,
    output Q1,
    output Q2
);
    reg Q0, Q1, Q2;
    always @(posedge clk)
    begin
        Q1<=Q0;
        Q2<=Q1;
        Q0<=D;
    end
endmodule
```

**[例 2-6]** 非阻塞赋值方式描述的移位寄存器 2。

```
module non_block2(
    input clk,
    input D,
    output Q0,
    output Q1,
    output Q2
);
    reg Q0, Q1, Q2;
    always @(posedge clk)
    begin
        Q2<=Q1;    //3 条赋值语句的顺序与例 2-5 完全颠倒
        Q0<=D;
        Q1<=Q0;
    end
endmodule
```

例 2-5 和例 2-6 综合结果是一样的，如图 2-5 所示。

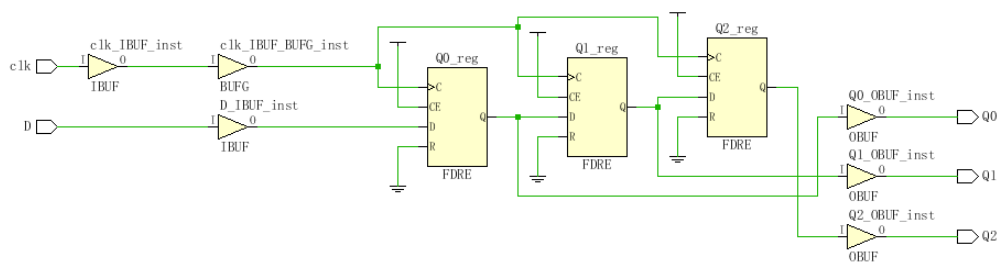


图 2-5

例 2-2~例 2-6 的程序说明如下。

① 5 个例题的设计目标均是实现 3 位移位寄存器，但从综合结果可以看出例 2-3 和例 2-4 没有最终实现设计目标。

②  $Q2=Q1$ ；这种赋值方式称为阻塞赋值， $Q2$  的值在赋值语句执行完成后立刻就改变，而且随后的语句必须在赋值语句执行完成后才能继续执行。所以对于第 3 个例题中的 3 条语句  $Q0=D$ ;  $Q1=Q0$ ;  $Q2=Q1$ ；执行完成后， $Q0$ 、 $Q1$ 、 $Q2$  的值都变化为  $D$  的值，也就是说， $D$  的值同时赋给了  $Q0$ 、 $Q1$ 、 $Q2$ ，参照其综合结果能更清晰地看到这一点。例 2-2、例 2-3 可通过同样的分析得出与综合结果一致的结论。

③  $Q2<=Q1$ ；这种赋值方式称为非阻塞赋值。 $Q2$  的值在赋值语句执行完成后并不会立刻就改变，而是等到整个 `always` 语句块结束后才完成赋值操作。所以对于例 2-6 中的 3 条语句  $Q0<=D$ ;  $Q2<=Q1$ ;  $Q1<=Q0$ ；执行完成后， $Q0$ 、 $Q1$ 、 $Q2$  的值并没有立刻更新，而是保持了原来的值，直到 `always` 语句块结束后才同时进行赋值，因此  $Q0$  的值变为了  $D$  的值， $Q2$  的值变为了原来  $Q1$  的值， $Q1$  的值变为了原来  $Q0$  的值（而不是刚刚更新的  $Q0$  的值  $D$ ），参照其综合结果能更清晰地看到这一点。例 2-5 可通过同样的分析得出与综合结果一致的结论。

④ 例 2-2~例 2-4 采用的是阻塞赋值方式，可以看出阻塞赋值语句在 `always` 块语句中的位置对其结果有影响；例 2-5 和例 2-6 采用的是非阻塞赋值方式，可以看出非阻塞赋值语句在 `always` 块语句中的位置对其结果没有影响。因此，在使用赋值语句时要注意两者的区别与联系。

过程赋值语句只能对寄存器类型的变量（`reg`、`integer`、`real` 和 `time`）进行操作，经过赋值后，上面这些变量的取值将保持不变，直到另一条赋值语句对变量重新赋值为止。过程赋值操作的具体目标如下：

- ① `reg`、`integer`、`real` 和 `time` 型变量（矢量和标量）
- ② 上述变量的一位或几位。
- ③ 上述变量用 `{}` 操作符所组成的矢量。
- ④ 存储器类型，只能对指定地址单元的整个字进行赋值，不能对其中某些位单独赋值。

### 3) 关系运算符

关系运算符总共有 8 种： $>$ （大于）、 $>=$ （大于或等于）、 $<$ （小于）、 $<=$ （小于或等于）、 $==$ （逻辑相等）、 $!=$ （逻辑不相等）、 $===$ （全等）、 $!==$ （不全等）。

在进行关系运算时，如果操作数之间的关系成立，返回值为 1；关系不成立，则返回值为 0；若某一个操作数的值不定，则关系是模糊的，返回的是不定值  $X$ 。

算子“ $==$ ”和“ $!=$ ”可以在比较含有  $X$  和  $Z$  的操作数，在模块的功能仿真中有着广泛的应用。所有的关系运算符有着相同优先级，但低于算术运算符的优先级。



#### 4) 逻辑运算符

Verilog HDL 中有 3 类逻辑运算符：&&（逻辑与）、||（逻辑或）、!（逻辑非）。

其中，“&&”和“||”是二目运算符，要求有两个操作数；而“!”是单目运算符，只要求一个操作数。“&&”和“||”的优先级高于算术运算符。逻辑运算符的真值表如表 2-1 所示。

表 2-1 逻辑运算符的真值表

a	b	!a	!b	a&& b	a  b
1	1	0	0	1	1
1	0	0	1	0	1
0	1	1	0	0	1
0	0	1	1	0	0

#### 5) 条件运算符

条件运算符的格式如下：

**y=x? a:b ;**

条件运算符有 3 个操作数，若第一个操作数 y=x 是 True，算子返回第二个操作数 a，否则返回第三个操作数 b。例如：

**wire y;**

**assign y= (s1==1)? a:b ;**

嵌套的条件运算符可以实现多路选择。例如：

**wire[1 :0 ] s;**

**assign s=(a>=2) ? 1 ((a<0) ? 2: 0);**

当 a>=2 时，s=1；当 a<0 时，s=2；在其余情况，s=0。

#### 6) 位运算符

作为一种针对数字电路的硬件描述语言，Verilog HDL 用位运算来描述电路信号中的与、或以及非操作，总共有以下 7 种位逻辑运算符：~（非）、&（与）、|（或）、^（异或）、^^（同或）、~&（与非）、~|（或非）。位运算符中除了“~”，都是二目运算符。位运算对其自变量的每一位进行操作，例如：s1&s2 的含义就是 s1 和 s2 的对应位相与。如果两个操作数的长度不相等的话，将会对较短的数高位补零，然后进行对应位运算，使输出结果的长度与位宽较长的操作数长度保持一致。例如：

**reg[ 3:0 ] s1, v1, v2, var;**

**s1=~s1;**

**var=v1&v2;**

#### 7) 移位运算符

移位运算符只有两种：“<<”（左移）和“>>”（右移），左移一位相当于乘 2，右移一位相当于除 2。其使用格式如下：

**s1<<N; 或 s1>>N;**

其含义是将第一个操作数 s1 向左（右）移位，所移动的位数由第二个操作数 N 来决定，且都用 0 来填补移出的空位。

在实际运算中，经常通过不同移位数的组合来计算简单的乘法和除法。例如 s1\*21，因为 21=16+4+1，所以可以通过 s1<<4+s1<<2+s1 来实现；s1/8 可以通过 s1>>3 来实现。

#### 8) 拼接运算符

拼接运算符可以将两个或更多个信号的某些位拼接起来进行运算操作。其使用格式如下。

**{s1,s2,...,sn}**

将某些信号的某些位详细地列出来，中间用逗号隔开，最后用一个大括号表示一个整体信号。在工程实际中，拼接运算受到了广泛应用，特别是在描述移位寄存器时。例如：

```
reg[ 15:0 ]shiftreg ;
always @(posedge clk)
    shiftreg [15: 0] <= {shiftreg [14:0] , data_in} ;
```

#### 9) 一元约简运算符

一元约简运算符是单目运算符，其运算规则类似于位运算符中的与、或、非，但其运算过程不同，约简运算符对单个操作数进行运算，最后返回一位数，其运算过程为：首先将操作数的第一位和第二位进行与、或、非运算；然后再将运算结果和第三位进行与、或、非运算，以此类推直至最后一位。

常用的约简运算符的关键字和位操作符关键字一样，仅仅由单目运算和双目运算来区别。例如：

```
reg[3:0] s1;
reg s2;
s2=&s1;    //&即为一元约简运算符“与”
```

#### 10) 各种运算符的优先级别

如果不使用小括号将表达式的各个部分分开，则 Verilog 将根据运算符之间的优先级对表达式进行计算。如图 2-6 列出了常用的几种运算符的优先级别。

优先级别	
<pre> !  - *  /  % +  - &lt;&lt;  &gt;&gt; &lt;  &lt;=  &gt;  &gt;= ==  != &amp;  ~&amp; ^  ^~    ~  &amp;&amp;    ?:                     </pre>	<p>高优先级别</p> <p>↓</p> <p>低优先级别</p>

图 2-6 运算符的优先级

## 2.3 Verilog HDL 语言的描述语句

Verilog HDL 代码设计中常用 3 类描述语句：结构描述、数据流描述和行为描述。下面分别进行说明。

## 1. 描述形式

通过实例进行描述的方法，将 Verilog HDL 预先定义的基本单元实例嵌入到代码中。Verilog HDL 中定义了 20 多个有关门级的关键字，比较常用的有 8 个。在实际工程中，简单的逻辑电路由逻辑门和开关组成，通过门元语可以直观地描述其结构。

基本的门类型关键字包括：and、nand、nor、or、xor、xnor、buf、not。

Verilog HDL 支持的基本逻辑部件是由该基本逻辑器件的原语提供的。其调用格式如下：

**门类型<实例名>（输出，输入 1，输入 2，...，输入 N）**

例如，nand na01 (na\_out,a,b,c); 表示一个名字为 na01 的与非门，输出为 na\_out，输入为 a、b、c。

例 2-7 一个简单的全加器例子。

**[例 2-7] 全加器。**

```
module ADD(
    input A,
    input B,
    input Cin,
    output Sum,
    output Cout
);
//声明变量
wire S1, C1, C2, C3;
xor Xor1(S1, A, B),
    Xor2(Sum, S1, Cin);
and And1(C3, A, B),
    And2(C2, B, Cin),
    And3(C1, A, Cin);
or Or1(Cout, C1, C2, C3);
endmodule
```

在这一实例中，模块包含门的实例语句，也就是包含内置门 xor、and 和 or 的实例语句。门实例由线网型变量 S1、C1、C2 和 C3 互连。由于未指定顺序，门实例语句可以以任何顺序出现。

门级描述本质上也是一种结构网表。在实际应用中的使用方式为：先使用门逻辑构成常用的触发器、选择器、加法器等模块，再利用已经设计的模块构成更高一层的模块，依次重复几次，便可以构成一些结构复杂的电路。其缺点是：不易管理，难度较大且需要一定的资料积累。

## 2. 数据流描述形式

数据流型描述一般都采用 assign 连续赋值语句来实现，主要用于实现组合功能。连续赋值语句右边所有的变量受持续监控，只要这些变量有一个发生变化，整个表达式被重新赋值给左端。

**[例 2-8] 一个利用数据流描述的移位器。**

```
module mlshift2(a, b);
    input a;
    output b;
    assign b = a << 2;
endmodule
```

在上述模块中，只要 a 的值发生变化，b 就会被重新赋值，所赋值为 a 左移两位后的值。

### 3. 行为描述形式

行为型描述主要包括过程结构、语句块、时序控制、流控制 4 个方面，主要用于时序逻辑功能的实现。

#### 1) 过程结构

过程结构采用 4 种过程模块来实现，具有强的通用型和有效型。这 4 种模块包括：initial 模块、always 模块、task（任务）模块、function（函数）模块。

一个程序可以有多个 initial 模块、always 模块、task 模块和 function 模块。initial 模块和 always 模块都是同时并行执行的，区别在于 initial 模块只执行一次，而 always 模块则是不断重复地执行。另外，task 模块和 function 模块能被多次调用。

(1) initial 模块。在进行仿真时，一个 initial 模块从模拟 0 时刻开始执行，且在仿真过程中只执行一次，在执行完一次后，该 initial 模块就被挂起，不再执行。如果仿真中有两个 initial 模块，则同时从 0 时刻开始并行执行。

Initial 模块是**面向仿真的，是不可综合的**，通常被用来描述测试模块的初始化、监视、波形生成等功能。其格式为

```
initial begin/fork
    块内变量说明
    时序控制 1 行为语句 1;
    :
    :
    时序控制 n 行为语句 n;
```

end/join

其中，begin...end 块中的语句是串行执行的，而 fork...join 块中的语句是并行执行的。当块内只有一条语句且不需要定义局部变量时，可以省略 begin...end/ fork...join。

**[例 2-9]** 下面给出了 2 个 initial 模块的实例，分别用来设定时钟和复位信号。

//初始化复位信号

```
initial begin
    rst = 0;
    #100;
    rst = 1;
```

end

//初始化时钟信号，设置成周期为 20ns 的时钟信号

```
initial begin
    clk = 0;
    forever
        #10 clk = ~clk;
```

end

(2) always 模块和 initial 模块不同，always 模块是一直重复执行的，并且**可被综合**。always 过程块由 always 过程语句和语句块组成，其格式如下：

```
always @（敏感事件列表）begin
    块内变量说明
    时序控制 1 行为语句 1;
    :
```

:  
 时序控制 n 行为语句 n;

end

其中, `begin...end` 的使用方法和 `initial` 模块中的一样。敏感事件列表是可选项, 但在实际工程中却很常用, 而且是比较容易出错的地方。敏感事件表的目的是触发 `always` 模块的运行, 而 **`initial` 后面是不允许有敏感事件表的**。

敏感事件表由一个或多个事件表达式构成, 事件表达式就是模块启动的条件。当存在多个事件表达式时, 要使用关键词 `or` 将多个触发条件结合起来。Verilog HDL 的语法规则: 对于这些表达式所代表的多个触发条件, 只要有一个成立, 就可以启动块内语句的执行。例如, 在语句

```
always @(a or b or c) begin
```

```
...
```

```
end
```

中, `always` 过程块的多个事件表达式所代表的触发条件是: 只要 `a`、`b`、`c` 信号的电平有任意一个发生变化, `begin...end` 语句就会被触发。

`always` 模块主要是对硬件功能的行为进行描述, 可以实现锁存器和触发器, 也可以用来实现组合逻辑。利用 `always` 实现组合逻辑时, 要将所有的信号放进敏感列表, 而实现时序逻辑时却不一定要将所有的结果放进敏感信号列表。敏感信号列表未包含所有输入的情况称为不完整事件说明, 有时可能会引起综合器的误解, 产生许多意想不到的结果。

**[例 2-10]** 下面给出敏感事件未包含所有输入信号的情况。

```
module and3(f, a, b, c);
```

```
input a, b, c;
```

```
output f;
```

```
reg f;
```

```
always @(a or b) begin
```

```
    f = a&b&c;
```

```
end
```

```
endmodule
```

其中, 由于 `c` 不在敏感变量列表中, 所以当 `c` 值变化时, 不会重新计算 `f` 值。所以上面的程序并不能实现 3 输入的与门功能行为。正确的 3 输入与门应当采用下面的表述形式。

```
module and3 (f, a, b, c);
```

```
input a, b, c;
```

```
output f;
```

```
reg f;
```

```
always @(a or b or c) begin
```

```
    f=a&b &c;
```

```
end
```

```
endmodule
```

(3) `task` (任务) 模块和 `function` (函数) 模块。`task` 和 `function` 说明语句分别用来定义任务和函数。利用任务和函数可以把一个很大的程序模块分解成许多较小的任务和函数, 这样便于理解和调试。输入、输出和总线信号的值可以传入、传出任务和函数。任务和函数往往还是大的程序模块中在不同地点多次用到的相同的程序段。学会使用 `task` 和 `function` 语句可以简化程序的结构, 使程序明白易懂, 是编写较大型模块的基本功。

Verilog HDL 函数和任务在综合时被理解成具有独立运算功能的电路, 每调用一次函数

和任务相当于改变这部分电路的输入以得到相应的计算结果。

下面分别通过任务和函数来实现对输入数进行按位逆序后输出。

**[例 2-11]** 用任务实现输入数据按位逆序后输出的功能。

```
module task_ex(clk, D, Q);
input clk;
input [MAX_BITS:1]D;
output reg[MAX_BITS:1] Q;
parameter MAX_BITS = 8;
task reverse_bits;
    input [MAX_BITS:1] data;
    output [MAX_BITS:1] result;
    integer K;
    for(K = 0; K<MAX_BITS; K = K+1)
        result[MAX_BITS-K]=data[K+1];
endtask
always @(posedge clk)
    reverse_bits(D, Q);
endmodule
```

程序说明：

① 本例说明了怎样定义任务和调用任务。起始于 `task` 而结束于 `endtask` 的部分定义了一个任务。任务的定义语法如下。

```
task <任务名>;
    <端口及数据类型声明语句>
    <语句 1>
    <语句 2>
    :
    :
    <语句 n>
endtask
```

这些声明语句的语法与模块定义中的对应声明语句的语法是一致的。

② `reverse_bits(D,Q);` 的功能是调用任务并传递输入输出变量给任务。调用任务并传递输入输出变量的声明语句的语法如下：

**<任务名> (端口 1, 端口 2, ..., 端口 n) ;**

本例中，任务调用变量 (D,Q) 和任务定义的 I/O 变量 (data, result) 之间是一一对应的。当任务启动时，由 D 传入的变量赋给了 data，而当任务完成从后的输出又通过 result 赋给了 Q。

③ 如果传给任务的变量值和任务完成后接收结果的变量已定义，就可以用一条语句启动任务。任务完成以后控制就传回启动过程。

使用任务完成的综合模块，同样也可以由函数实现。例 2-12 使用函数对例 2-11 进行了重新改写。

**[例 2-12]** 用函数实现输入数据位逆序后输出的功能。

```
module function_ex(clk, D, Q);
input clk;
```

```

input [MAX_BITS:1] D;
output reg [MAX_BITS:1] Q;
parameter MAX_BITS = 8;
function [MAX_BITS:1] reverse_bits;
    input [MAX_BITS:1] data;
    integer K;
    for(K = 0; K < MAX_BITS; K = K + 1)
        reverse_bits[MAX_BITS - K] = data[K + 1];
endfunction
always @(posedge clk)
    Q <= reverse_bits(D);
endmodule
    
```

程序说明：

① 本例说明了怎样定义函数和调用函数。起始于 **function** 而结束于 **endfunction** 的部分定义了一个函数。函数的定义语法如下。

```

function <返回值的类型或范围> (函数名);
    <端口说明语句>
    <变量类型说明语句>
begin
    <语句>
    :
    :
end
endfunction
    
```

**注意：** <返回值的类型或范围>这一项是可选项，如省略则返回值为 1 位寄存器类型数据。

这些声明语句的语法与模块定义中的对应声明语句的语法是一致的。

② **Q <= reverse\_bits(D);** 的功能是调用函数并传递输入变量给函数，函数的调用是通过将函数作为表达式中的操作数来实现的。在函数中，**reverse\_bits** 被赋予的值就是函数的返回值。

函数的定义蕴含声明了与函数同名的、函数内部的寄存器。如在函数的声明语句中 <返回值的类型或范围> 省略，则这个寄存器是一位的，否则是与函数定义中 <返回值的类型或范围> 一致的寄存器。函数的定义把函数返回值所赋值寄存器的名称初始化为与函数同名的内部变量。

调用函数并传递输入输出变量的声明语句的语法如下。

<函数名> (<表达式>, <<表达式>> \*)

其中函数名作为确认符。

本例中，函数调用变量 (**D, Q**) 和函数定义的 I/O 变量 (**data, reverse\_bits**) 之间是一一对应的。当函数启动时，由 **D** 传入的变量赋给了 **data**，而当函数完成后的输出又通过 **reverse\_bits** 赋给了 **Q**。

③ 如果传给函数的变量值和函数完成后接收结果的变量已定义，就可以用一条语句启动函数。函数完成以后控制就传回启动过程。

例 2-11 和例 2-12 两个例子的仿真波形如图 2-7 所示。

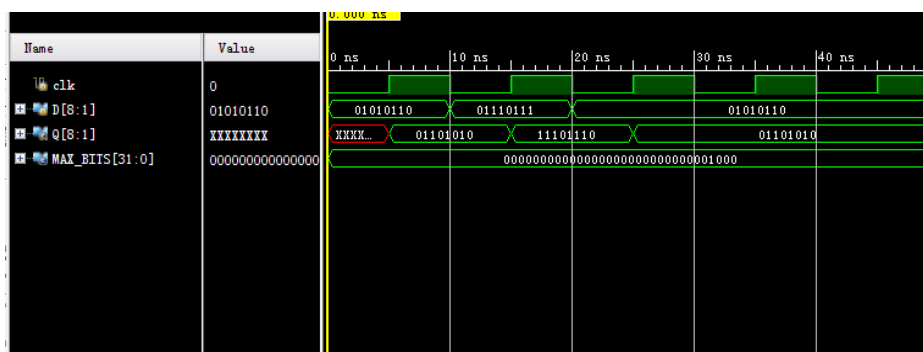


图 2-7

从以上仿真波形可以看出，两个例子均实现了位逆序的功能。

关于任务和函数的进一步说明如下。

① 任务和函数是有区别的。函数只能与主模块共用一个仿真时间单位，而任务可以定义自己的仿真时间单位；函数至少要有有一个输入变量，而任务可以没有或有多个任何类型的变量。

② 任务可以启动其他的任务，其他任务又可以启动别的任务，可以启动的任务数是没有限制的。不管有多少任务启动，只有当所有的启动任务完成以后，控制才能传回启动过程。另外，**任务能调用其他函数，而函数不能调用任务**。

③ 函数的目的是通过返回一个值来响应输入信号的值。任务却能支持多种目的，能计算多个结果值，而这些结果值只能通过被调用的任务的输出或总线端口送出。Verilog HDL 模块使用函数时是把它当作表达式中的操作符，这个操作符的结果值就是这个函数的返回值。也就是说，函数返回一个值，而任务则不返回值。

例如，定义一任务或函数对一个 16 位的字进行操作，让高字节与低字节互换，把它变为另一个字（假定这个任务或函数名为：switch\_bytes）。

任务返回的新字是通过输出端口的变量，因此 16 位字字节互换任务的调用方法为：switch\_bytes( old\_word , new\_word );。任务 switch\_bytes 把输入 old\_word 的高、低字节互换放入 new\_word 端口输出。

而函数返回的新字是通过函数本身的返回值，因此 16 位字高、低字节互换函数的调用方法为：new\_word= switch\_bytes ( old\_word );。

④ 与任务相比较函数的使用有较多的约束。例如：函数的定义不能包含有任何的时间控制语句，即任何用#、@或 wait 来标识的语句；函数不能启动任务；定义函数时至少要有有一个输入参量；在函数的定义中必须有一条赋值语句给函数中的一个内部变量赋以函数的结果值，该内部变量具有和函数名相同的名字。

## 2) 语句块

语句块就是在 initial 或 always 模块中位于 begin...end / fork...join 块定义语句之间的一组行为语句。语句块可以有个名字，写在块定义语句的第一个关键字之后，即 begin 或 fork 之后，可以唯一地标识出某一语句块。如果有了块名字，则该语句块被称为一个有名块。在有名块内部可以定义内部寄存器变量，且可以使用 disable 中断语句中断。块名提供了唯一标识寄存器的一种方法。

**[例 2-13]** 语句块使用的例子。

```
always @(a or b)
```

```
begin: adder
```

```
    C = a + b;
```



end

定义了一个名为 `adder` 的语句块，实现输入数据的相加。

按照界定不同分为以下 3 种。

(1) `begin...end`，用来组合需要顺序执行的语句，被称为串行块。

**[例 2-14]** 串行块的使用例子。

```
parameter d = 50;
reg[7:0] r;
begin                                //由一系列延迟产生的波形
    #d r=8'h35;                      //语句 1
    #d r=8'hE2;                      //语句 2
    #d r=8'hFA;                      //语句 3
    #d r=8'h96;                      //语句 4
    #d->end_wave;                    //语句 5，触发事件 end_wave
end
```

串行块的执行特点如下。

串行块的各条语句是按它们在块内的语句逐次逐条顺序执行的，当前一条执行完之后，才能执行下一条。如例 2-14 中语句 1 至语句 5 是顺序执行的。

块内每一条语句中的延时控制都是相对于前一条语句结束时刻的延时控制。如例 2-14 中语句 2 的时延为 2d。

在进行仿真时，整个语句块总的执行时间等于所有语句执行时间之和。如例 2-14 中语句块中总的执行时间为 5d。

(2) `fork...join`，用来组合需要并行执行的语句，被称为并行块。

**[例 2-15]** 并行块的使用例子。

```
parameter d = 50;
reg[7:0] r;
fork                                //由一系列延迟产生的波形
    #d r=8'h35;                      //语句 1
    #2d r=8'hE2;                     //语句 2
    #3d r=8'hFA;                     //语句 3
    #4d r=8'h96;                     //语句 4
    #5d->end_wave;                   //语句 5，触发事件 end_wave
join
```

并行块的执行特点如下。

并行语句块内各条语句是各自独立地同时开始执行的，各条语句的起始执行时间都等于程序流程进入该语句块的时间。如例 2-15 中语句 2 并不需要等语句 1 执行完才开始执行，它与语句 1 是同时开始的。

块内每一条语句中的延时控制都是相对于程序流程进入该语句块的时间而言的。如例 2-15 中语句 2 的延时为 2d。

在进行仿真时，整个语句块总的执行时间等于执行时间最长的那条语句所需要的执行时间，如例 2-15 中整个语句块的执行时间为 5d。

(3) 混合使用。在分别对串行块和并行块进行了介绍之后，还需要讨论一下二者的混合使用。混合使用可以分为下面两种情况。

串行块和并行块分别属于不同的过程块时，串行块和并行块是并行执行的。例如一个串

行块和并行块分别存在于两个 `initial` 过程块中，由于这两个过程是并行执行的，所以其中所包含的串行语句和并行语句也是同时并行执行的。在串行块内部，其语句是串行执行的；在并行块内部，其语句是并行执行的。

当串行块和并行块嵌套在同一过程中时，内层语句可以看作是外层语句块中的一条普通语句，内层语句块什么时候得到执行是由外层语句块的规则决定的，而在内层语句块开始执行时，其内部语句怎么执行就要遵守内层语句块的规则。

### 3) 时序控制

Verilog HDL 提供了两种类型的时序控制，一种是延迟控制，在这种类型的时序控制中通过表达式定义开始遇到这一语句和真正执行这一语句之间的延迟时间。另外一种事件控制，这种时序控制是通过表达式来实现的，只有当某一时间发生时才允许语句继续向下执行。

(1) 延时控制。延时控制的语法如下：

#延时数表达式;

延时控制表示在语句执行前的“等待时延”，下面给出一个例子。

```
initial
begin
    #10 clk = ~clk;
end
```

**延时控制只能在仿真中使用，是不可综合的。**在综合时，所有的延时控制都会被忽略。

(2) 事件控制。事件控制分为两种：边沿触发事件控制和电平触发事件控制。

边沿触发事件是指指定信号的边沿信号跳变时发生指定的行为，分为信号的上升沿和下降沿控制。上升沿用 `posedge` 关键字来描述，下降沿用 `negedge` 关键字描述。边沿触发事件控制的语法格式如下。

第一种：@(<边沿触发事件>) 行为语句。

第二种：@(<边沿触发事件 1> or <边沿触发事件 2> or...or<边沿触发事件 n>) 行为语句。

**[例 2-16]** 边沿触发事件计数器。

```
reg [3:0] cnt;
always @(posedge clk)
begin
    if(reset)
        cnt<=0;
    else
        cnt<=cnt+1;
end
```

这个例子表明：只要 `clk` 信号有上升沿，那么 `cnt` 信号就会加 1，完成计数的功能。这种边沿计数器在同步分频电路中有着广泛的应用。

电平敏感事件是指指定信号的电平发生变化时发生指定的行为。下面是电平触发事件控制的语法和实例。

第一种：@(<电平触发事件>) 行为语句。

第二种：@(<电平触发事件 1> or <电平触发事件 2> or...or<电平触发事件 n>) 行为语句。

**[例 2-17]** 电平触发计数器。

```
reg [3:0] cnt;
always @(clk)
begin
    if(reset)
        cnt<=0;
    else
        cnt<=cnt+1;
end
```

这个例子表明：只要 clk 信号的电平有变化，包括上升沿和下降沿这两种情况，信号 cnt 的值就会加 1，这可以用于记录 cnt 变化的次数。注意与例 2-16 的区别，例 2-17 的计数值应该比例 2-16 的计数值多 1 倍。

#### 4) 流控制

流控制语句包括 3 类，即跳转、分支和循环语句。

(1) if 语句。if 语句的语法如下。

```
if (条件 1)
    语句块 1
else if (条件 2)
    语句块 2
:
:
else
    语句块 n
```

如果条件 1 的表达式为真（或非 0 值），那么语句块 1 被执行，否则语句块不被执行，然后依次判断条件 2 至条件 n 是否满足，如果满足就执行相应的语句块，最后跳出 if 语句，整个模块结束。如果所有的条件都不满足，则执行最后一个 else 分支。在应用中，else if 分支的语句数目由实际情况决定，else 分支也可以忽略，但会产生一些不可预料的结果，生成本不期望的锁存器。

**[例 2-18]** 下面给出一个 if 语句的例子，并说明省略 else 分支所产生的一些结果。

```
always@(cond or d)
begin
    if(cond) q<=d;
end
```

if 语句只能保证当 cond=1 时，q 才取 d 的值，但程序没有给出 cond=0 时的结果。因此在缺少 else 语句的情况下，即使 cond=0 时，q 的值也会保持 cond=1 的原值，这就综合成了一个锁存器。

如果希望 cond=0 时，q 的值为 0 或者其他值，那么 **else 分支是必不可少的**。下面给出 cond=0，q=0 的设计方法。

```
always @(cond or d)
begin
    if (cond) q<=d;
    else q<=0;
end
```

(2) case 语句。case 语句是一个多路条件分支形式，其用法和 C 语言的 case 语句是一样的。

下面给出一个 case 语句的例子。

```
reg[2:0] cnt;
case (cnt)
    3'b000:q=8'b00000001;
    3'b001:q=8'b00000010;
    3'b010:q=8'b00000100;
    3'b011:q=8'b00001000;
    3'b100:q=8'b00010000;
    3'b101:q=8'b00100000;
    3'b110:q=8'b01000000;
    3'b111:q=8'b10000000;
    default: q<=0;
```

endcase

需要指出的是，case 语句的 default 分支虽然可以忽略，但是一般不要省略，否则会和 if 语句中缺少 else 分支一样，生成锁存器。

**[例 2-19]** 给出 case 语句的 Verilog 实例。

```
always@(cond[1:0] or d)
begin
    case(cond)
        2'b00: q<=d;
        2'b01:q<=d+1;
    end
```

这样就会生成锁存器。一般为了使 case 语句可控，都需要加上 default 选项。

```
always@(cond[1:0] or d)
begin
    case(cond)
        2'b00: q<=d;
        2'b01:q<=d+1;
        default: q<=0;
    end
```

在实际开发中，要避免生成锁存器的错误。如果用 if 语句，最好写上 else 选项；如果用 case 语句，最好写上 default 项。遵循上面两条原则，就可以避免发生这种错误，使设计者更加明确设计目标，同时也增加了 Verilog 程序的可读性。

此外，还需要解释在硬件语言中使用 if 语句和 case 语句的区别，在实际中如果有分支情况，尽量选择 case 语句。这是因为 case 语句的分支是并行执行的，各个分支没有优先级的区别，而 if 语句的选择分支是串行执行的，是按照书写的顺序逐次判断的。如果设计没有这种优先级的考虑，if 语句和 case 语句相比，需要占用额外的硬件资源。

(3) 循环语句。Verilog HDL 中提供了 4 种循环语句：for 循环语句、while 循环语句、forever 循环语句和 repeat 循环语句。其语法和用途与 C 语言很相似。

for 循环语句按照指定的次数重复执行过程赋值语句。for 循环语句的语法如下：

**for (表达式 1; 表达式 2; 表达式 3) 语句**

for 循环语句最简单的应用形式是很容易理解的，其形式如下。

for（循环变量赋初值；循环结束条件；循环变量增值）

**[例 2-20]** for 语句的应用实例。

```
parameter WIDTH = 32;
reg [WIDTH-1:0] DATA;
integer I;
initial
    for(i = 0; i < WIDTH; i=i+1)
        DATA[i] = 1'b0;
```

例 2-20 将 32 位的变量逐位清 0。

while 循环语句执行过程赋值语句直到指定的条件为假。如果表达式条件在开始不为真（包括假、x 以及 z），那么过程语句将永远不会被执行。while 循环语句的语法如下。

```
while（条件表达式）begin
```

```
...
```

```
end
```

**[例 2-21]** while 循环语句的应用实例。

```
initial begin
    while(EMPTY == 1'b0) begin
        @(posedge CLK);
        #1 read_fifo = 1'b1;
    end
```

在例 2-21 中当 FIFO 不空时，在时钟上升沿之后的 1 个时间单位时刻读取 FIFO。

forever 循环语句连续执行过程语句。为跳出这样的循环，中止语句可以与过程语句共同使用。同时，在过程语句中必须使用某种形式的时序控制，否则 forever 循环语句将永远循环下去。forever 循环语句必须写在 initial 模块中，用于产生周期性波形。forever 循环语句的语法如下。

```
forever begin
```

```
...
```

```
end
```

**[例 2-22]** forever 语句的应用实例。

```
initial begin
    forever begin
        CLK = 1'b0;
        #5 CLK = 1'b1;
        #5;
    end
```

```
end
```

例 2-22 将产生一个周期为 10 的 CLK 信号。

repeat 循环语句执行指定循环数，如果循环计数表达式的值不确定，即为 x 或 z 时，那么循环次数按 0 处理。repeat 循环语句的语法如下。

```
repeat（表达式）
```

```
begin
```

```
...
```

```
end
```

**[例 2-23]** repeat 语句的应用实例。

```

initial begin
    repeat (30) begin
        @(posedge CLK);
        #1 DATA_IN=$random;
    end
end

```

在例 2-23 中，在每次时钟上升沿之后 1 个时间单位时随机产生一个数据并赋给 DATA\_IN，这样的操作重复 30 次。

上面介绍了 Verilog HDL 代码设计中常用的 3 类描述语句。需要说明的是，在实际应用中，结构描述、数据流描述和行为描述可以自由混合。也就是说，模块描述中可以包含实际化的门、模块实例化语句、连续赋值语句以及行为描述语句的混合。下面给出一个混合设计方式的实例。

**[例 2-24]** 用结构和行为实体描述了一个 4 位全加器。

```

module adder4(in1, in2, sum, flag);
    input [3:0] in1;
    input [3:0] in2;
    output[4:0] sum;
    output flag;
    wire c0, c1, c2;
    reg [4:0] sum_o;
    fulladd u1(in1[0], in2[0], 0, sum[0], c0);
    fulladd u1(in1[1], in2[1], c0, sum[1], c1);
    fulladd u1(in1[2], in2[2], c1, sum[2], c2);
    fulladd u1(in1[3], in2[3], c2, sum[3], sum[4]);
    assign flag = sum[4];
    always @(in1, in2)
        sum_o = in1+in2;
endmodule

```

在这个例子中，混合使用了结构描述、数据流描述和行为描述。

## 2.4 Verilog 代码书写规范

代码书写规范就是通过建立起一种通用的约定和模式，在写代码的时候遵循，以此帮助打造健壮的软件。

使用编码规范有很多好处。

- (1) 保持编码风格，注释风格一致，应用设计模式一致。
- (2) 新程序员通过熟悉相关的编码规范，可以更容易、更快速地掌握已有的程序库。
- (3) 降低代码中漏洞（bug）出现的可能性。

代码书写规范包括的内容很多，例如信号命名规范、模块命名规范、代码格式规范、模块调用规范等，本节就代码格式规范和模块调用规范作一些说明。

### 1. 代码格式规范

#### (1) 分节书写格式

各节之间加一到多行空格。如每个 always、initial 语句都是一节。每节基本上完成一个特定的功能，即用于描述某几个信号的产生。在每节之前有几行注释对该节代码加以描述，

至少列出本节中所描述信号的含义。

行首不要使用空格来对齐，而是用 Tab 键，Tab 键的宽度设为 4 个字符宽度。行尾不要有多余的空格。

## (2) 注释的规范

使用//进行的注释行以分号结束；使用/\* \*/进行的注释，/\* 和\*/各占用一行，并且顶头。例如：

```
//Edge detector used to synchronize the input signal;
对于函数，应该从“功能”、“参数”、“返回值”、“主要思路”、“调用方法”、“日期”6 个
方面如下格式注释：
//模块说明开始
//=====//
//功能：完成两个输入数的相加。
//参数：a, b, sum
//输入参数：a, b
//输出参数：sum
//主要思路：本算法主要采用一位加法器级联的方法完成多位相加
//日期：2015/11/20
//版本：
//代码编写人员：
//=====//
//模块说明结束
```

此外，在注释说明中，需要注意以下细节：

① 在注释中应该详细说明模块的主要实现思路，特别要注明自己的一些想法，如果有必要则应该写明想法产生的来由。

② 在注释中详细注明模块的适用性，强调使用时可能出错的地方。

③ 对模块注释开始到模块命名之间应该有一组用来标识的特殊字符串。如果算法比较复杂，或算法中的变量定义与位置有关，则要求对变量的定义进行图解。对难以理解的算法能图解尽量图解。

## (3) 空格的使用

不同变量，以及变量与符号、变量与括号之间都应当保留一个空格。Verilog 关键字与其他任何字符串之间都应当保留一个空格。例如：

```
always @(...)
```

使用大括号和小括号时，前括号的后边和后括号的前边应当留有一个空格。逻辑运算符、算术运算符、比较运算符等运算符的两侧各留一个空格，与变量分离开；单操作数运算符例外，直接位于操作数前，不使用空格。使用“//”进行的注释，在“//”后应当有一个空格；注释行的末尾不要有多余的空格。例如：

```
assign SramAddrBus = { AddrBus[31:24], AddrBus[7:0] };
assign DivCnt[3:0] = DevCnt[3:0] + 4'b0001;
assign Result = ~Operand;
```

## (4) begin...end 的书写规范

同一个层次的所有语句左端对齐；initial、always 等语句块的 begin 关键词跟在本行的末尾，相应的 end 关键词与 initial、always 对齐。这样做的好处是避免因 begin 独占一行而造成行数太多。例如：

```
always @(posedge SysClk or negedge SysRst) begin
```

```
if ( !SysRst ) DataOut <= 4'b0000;  
else if ( LdEn ) begin  
    DataOut <= DataIn;  
end  
else  
    DataOut <= DataOut + 4'b0001;  
end
```

不同层次之间的语句使用 Tab 键进行缩进，每加深一层缩进一个 Tab。在 endmodule、endtask、endcase 等标记一个代码块结束的关键词后面要加上一行注释说明这个代码块的名称。

## 2. 模块调用规范

在 Verilog 中，有两种模块调用的方法，一种是位置映射法，严格按照模块定义的端口顺序来连接，不用注明原模块定义时规定的端口名，其语法如下：

模块名 (连接端口 1 信号名, 连接端口 2 信号名, 连接端口 3 信号名, ...);

另一种为信号映射法，即利用“.”符号，表明原模块定义时的端口名，其语法如下：

模块名 (.端口 1 信号名 (连接端口 1 信号名),  
 .端口 2 信号名 (连接端口 2 信号名),  
 .端口 3 信号名 (连接端口 3 信号名), ...);

显然，信号映射法同时将信号名和被引用端口名列出来，不必严格遵守端口顺序，不仅降低了代码易错性，还提高了程序的可读性和可移植性。因此，在良好的代码中，尽量避免使用位置调用法，建议全部采用信号映射法。

## 2.5 小结

本章应用性地介绍了 Verilog HDL 硬件描述语言的语法，包括以下几方面内容：

- (1) 语言结构
- (2) 数据类型
- (3) 过程描述语句
- (4) 代码书写规范



## 第3章 单周期 MIPS 处理器设计

计算机的工作目标是严格按照输入设备提供的指令和数据利用指令流操纵数据集，并将执行的结果以某种形式表达出来。只要满足了若干基础指令，它便是一个可以精确表达运算逻辑的万能计算系统。CPU（即中央处理器）在计算机中担当计算和处理的重任。它本身不关心数据的存储及结果的显示等等，而需要根据指令对数据集进行运算或指令流控制，更新内部的核心寄存器的值，并提供输出。CPU 是一个被动工作的数字电路。它的工作节拍通过外部时钟和锁相环的倍频所确定，它的指令和数据由存储器所提供，它的工作逻辑由指令所确定。如果我们预知它的输入，那么我们可以精确推导它的将来状态。如果把指令地址作为状态变量，那么它是一个可控可测的大型的数字电路状态机。数字集成电路技术的飞跃发展以及芯片工艺能力的提高使得我们能够容易地将数十亿个晶体管组成我们的 CPU 系统，使得它愈加神秘。然而，从功能实现的角度，它只不过是一个复杂一些的用于将机器指令码转化为电子的规则运转的数字逻辑。

CPU 自从诞生之日起，就牵扯到速度和效率的问题。因此几十年来人们在不断改进它的工艺和设计思想，用以制造出更快更高效的 CPU。当然，使 CPU 工作得更快的方法有很多，无论是工艺的改善还是结构的改进都能够有所效用，然而，最大的修改莫过于指令集的改进，CPU 根据指令集的不同，主要分为 CISC 和 RISC 两大类。RISC 作为 CISC 的改进，是计算机体系结构上的里程碑，因而 RISC 精简指令集架构在结构和效率上都较 CISC 更高效。

### CISC 处理器与传统处理器设计思想

CISC（Complex Instruction Set Computer），中文译名为“复杂指令计算机”，它是传统计算机的代表。在计算机发展的初期，由于 VLSI 工艺水平较低，存储介质较差等原因，这类计算机处理器在设计理念上具有如下特点：

#### 1. 注重代码长度和存储效率，大量使用存储器-存储器操作指令

由于当时的存储器容量小，因此，希望设计更加紧凑的代码，采用存储器操作的指令可以有效的减少指令长度。为了充分利用内存，还采用了变长指令，即复杂指令可以有多个字节，虽节省了存储空间，但大大增加了解码难度。而由于当时主存速度与一个微码的存储周期大致相同，所以大量采用存储器操作是很划算的。如果一个加法操作采用存储器-存储器方式，只需要一条指令，但如果采用寄存器-寄存器方式，则还需要两条 load 指令和一条 store 指令。因此，为了代码长度和存储效率，当时的流行的方法是采用丰富的存储器操作。

#### 2. 指令系统丰富，功能强大

当时认为愈加丰富的指令功能，可以大大减轻编译系统的工作，亦可以减轻软件危机。由于软件成本的上升和硬件成本的下降，计算机设计者向尽可能的把计算机执行高级语言的功能转移到硬件上去。这也无疑增大了结构的复杂性，降低了执行效率。

#### 3. 大量采用微码技术

为了满足指令兼容和不断扩充的复杂指令要求，单单依靠增大结构复杂度很难解决问题，因此设计了一个相对固定的微处理核，要扩展指令只需扩充微程序存储器即可。即采用一个只能执行少量操作的核通过多个周期的微操作来完成一条复杂指令。简单讲是处理器中嵌入处理器，由于当时的工艺技术所限，当时传统计算机的机器周期一般相当于主存周期，正好等于 5-10 个微存储周期。因此将一个操作分为 5-10 个微操作阶段可以和主存相匹配。因此采用微码技术是合理的。

## CISC 处理器的瓶颈

随着时代的发展，尤其是 VLSI 工艺技术取得的惊人的进步，上述的传统计算机结构设计思想已经不能符合新的工艺技术要求，传统处理器遇到了下述若干问题：

(1) 存储效率显著提高，使得主存速度已经可以和微程序存储相比，不再比后者慢 5-10 倍了，同时，存储容量也有了很大的提高，因此，所谓存储效率已不再是体系结构设计时要考虑的重要标准了。

(2) 现有的一些著名计算机指令系统过于复杂，有些微码多达 400kb，使得微程序设计相当困难而难以调试。

(3) 在微码型计算机中很难做到一条指令的执行接近于一个微周期，平均来说需要 3-4 个微周期，而很多程序中的一些简单指令实际上只与一条微周期操作相当，若不采用微周期，反而能大大改进效率。

(4) 根据 20%-80%定律，一个指令系统中大约 20%的指令是程序中经常反复使用的，其使用量大约占到整个程序的 80%，而该指令系统中大约 80%的指令时很少用到的，其使用量只占整个程序的 20%。

(5) VLSI 进展带来的影响使得需要重新考虑系统设计硬件与软件的折中，试验表明，采用精简的指令集在不显著降低执行效率的情况下（不多于 20%），可以简化设计规模 5-10 倍。

## RISC 微处理器

RISC (Reduced Instruction Set Computer) 即精简指令系统计算机的中心思想是简化硬件设计硬件只执行很有限的最常用的那部分指令，大部分复杂的操作则使用成熟的编译技术由简单指令合成。RISC 出现的结果是用相对少的晶体管可设计出极快的微处理器。

RISC微处理器具有以下几个特征：

### 1. 简化的指令集

(1) 大多数指令是单周期完成的，指令系统中的绝大部分指令只执行一些简单和基本的功能，这些指令可以较快的在单周期内执行完毕，并使指令的译码和解释开销减少。

(2) 采用硬布线控制逻辑，可以使大多数指令在单周期执行完毕，并减少微码技术中的指令解释开销。以往，微码控制部件往往占去VLSI芯片面积的50%-60%，节省微码器件的空间可以用于制作较大的寄存器堆。典型的RISC中都采用大量的寄存器，使大部分指令操作都在寄存器之间进行，从而提高了处理速度。

(3) 较少的指令数和寻址方式，从而有利于控制单元的简化和执行速度的加快指令格式尽量简单规范，使指令的译码逻辑电路简化，从而也使控制部件速度加快。

### 2. 面向寄存器堆的结构

过去传统的设计思想中，从提高“存储效率”出发，设置很多存储器-存储器操作指令，然而，存储器与CPU之间需要进行板级通信，较之CPU内部寄存器间的芯片级通信，其速度要低得多，因此，面向存储器意味着绝大多数运算只在寄存器之间完成，与外界存储器的通信只保留Load/Store两组指令，达到了凡是ALU执行部件中所用的操作数都是已经放在寄存器中的寄存器操作数的目的，从而有效地减少存储器的访问时间。

### 3. 充分提高流水线的效率

虽然流水线不是RISC的专利，RISC处理器也并不一定是流水的。但RISC处理器要提高性能，基本指令要做到一个机器周期内完成，必须采用流水线技术。这样才

可以充分利用CPU内部器件的并行性。而传统的流水线技术面临着指令长度不同，执行周期不一，资源争用问题以及转移跳转难以控制等困难，RISC流水线则有所不同。RISC的指令格式长度统一，使得更容易用简单的流水线结构来处理。分支延迟槽的引入使得分支跳转更加容易控制。当然，它也同样面临着相邻指令间结构相关性的问题，不可避免地影响执行的流畅性。

#### 4.注重编译优化

用编译时间换取运行时间的高效率。RISC指令集的简化虽可以使硬件复杂性降低，但也导致了编译后代码的长度较长。RISC技术强调编译优化技术，即编译初步生成的代码要重新加以组合，调度指令的执行次序，尽量少的存储器访问操作，以及转移时插入与转移无关的语句，发挥其流水并行化的特点，从而使执行效率提高。

一般来说RISC处理器比同等的CISC处理器要快50%至75% 同时由于RISC处理器结构的简单，使得更容易设计和纠错。

#### MIPS 处理器

MIPS(Microprocessor without interlocked piped stages)是高效的 RISC 体系结构中最优雅的一种体系结构，其中文意思为“无内部互锁流水级的微处理器”，其机制是尽量利用软件办法避免流水线中的数据相关问题。它最早是在 80 年代初期由斯坦福(Stanford)大学 Hennessy 教授领导的研究小组研制出来的。MIPS 公司的 R 系列就是在此基础上开发的 RISC 工业产品的微处理器。这些系列产品为很多计算机公司采用构成各种工作站和计算机系统。

MIPS 总能在每代处理器设计时保持最简洁的设计，同时获得最快的速度。1986 年推出 R2000 处理器，1988 年推出 R3000 处理器，1991 年推出第一款 64 位商用微处理器 R4000。之后，又陆续推出 R8000(于 1994 年)、R10000(于 1996 年)和 R12000(于 1997 年)等型号。1999 年，MIPS 公司发布 MIPS 32 和 MIPS 64 架构标准。2000 年，MIPS 公司发布了针对 MIPS 32 4Kc 的新版本以及未来 64 位 MIPS 64 20Kc 处理器内核。

在 MIPS 芯片的发展过程中，SGI 公司在 1992 年收购了 MIPS 计算机公司，1998 年，MIPS 公司又脱离了 SGI，成为 MIPS 技术公司；MIPS32 4KcTM 处理器是采用 MIPS 技术特定为片上系统(System-On-a-Chip)而设计的高性能、低电压 32 位 MIPS RISC 内核。采用 MIPS32TM 体系结构，并且具有 R4000 存储器管理单元(MMU)以及扩展的优先级模式，使得这个处理器与目前嵌入式领域广泛应用的 R3000 和 R4000 系列(32 位)微处理器完全兼容。

新的 64 位 MIPS 处理器是 RM9000x2，从“x2”这个标记判断，它包含了不是一个而是两个均具有集成二级高速缓存的 64 位处理器。RM9000x2 主要针对网络基础设施市场，具有集成的 DDR 内存控制器和超高速的 HyperTransport I/O 链接。

处理器、内存和 I/O 均通过分组交叉连接起来的，可实现高性能、全面高速缓存的统一芯片系统。除通过并行处理提高系统性能外，RM9000x2 还通过将超标量与超流水线技术相结合来提高单个处理器的性能。

64 位处理器 MIPS 64 20Kc 的浮点能力强，可以组成不同的系统，从一个处理器的 Octane 工作站到 64 个处理器的 Origin 2000 服务器；这种 CPU 更适合图形工作站使用。MIPS 最新的 R12000 芯片已经在 SGI 的服务器中得到应用，目前其主频最大可达 400MHz。

MIPS 处理器是八十年代中期 RISC CPU 设计的一大热点。MIPS 是卖的最好的 RISC CPU，可以从任何地方，如 Sony，Nintendo 的游戏机，Cisco 的路由器和 SGI

超级计算机, 看见 MIPS 产品在销售。目前随着 RISC 体系结构遭到 x86 芯片的竞争, MIPS 有可能是起初 RISC CPU 设计中唯一的一个在本世纪盈利的。和英特尔相比, MIPS 的授权费用比较低, 也就为除英特尔外的大多数芯片厂商所采用。

MIPS 的系统结构及设计理念比较先进, 其指令系统经过通用处理器指令体系 MIPS I、MIPS II、MIPS III、MIPS IV 到 MIPS V, 嵌入式指令体系 MIPS16、MIPS32 到 MIPS64 的发展已经十分成熟。在设计理念上 MIPS 强调软硬件协同提高性能, 同时简化硬件设计。

我国的龙芯 2E 和前代产品采用的都是 64 位 MIPS 指令架构, 索尼 PS2 游戏机所用的“Emotion Engine”也采用 MIPS 指令, 这些 MIPS 处理器的性能都非常强劲, 而龙芯 2E 也属于这个阵营, 在软件方面与上述产品完全兼容。

下面介绍单周期 MIPS 处理器的分析设计过程。

处理器或者说 CPU 是现在计算机非常复杂的部件, 设计一个简单但是能工作的处理器也不是那么神秘。下面来探索处理器如何设计出来的。

设计处理器可分为以下几个步骤:

- (1) 首先分析指令系统, 指令系统是在处理器设计之前由软硬件设计人员共同协商决定的。通过分析指令系统, 可以得出指令所要操作的数据要经过怎样的电路结构, 这就是要得出对数据通路的要求。
- (2) 为数据通路选择合适的组件, 比方说加法器、减法器、寄存器等。
- (3) 选好合适的组件后, 按照最初分析的要求连接组件, 建立完整的数据通路。
- (4) 仅有数据通路是不够的, 还要控制数据通路如何工作。因此分析每条指令的实现, 以确定控制数据通路工作的控制信号。
- (5) 集成控制信号, 形成完整的控制逻辑。也称为控制器。

### 3.1 指令系统设计

指令集结构包括指令长度、寻址方式等。设计的指令集采用 32-bit 编码, 因为指令长度相同, 能够降低译码的复杂度, 减少译码延迟, 并能很容易的使用流水线来提高处理器的效率。设计的嵌入式处理器只有 load 和 store 指令对存储系统进行访问。简单的指令寻址方式简化了嵌入式微处理器控制器和数据通路的设计。

处理器采用通用寄存器组结构, 在这种结构中, 指令可以从寄存器堆读出操作数, 并将执行的结果回写到寄存器堆中, 其中 R 类型和 I 类型都可能回写寄存器堆。

MIPS 指令集可以分为以下几类, 如图 3-1 和 3-2 所示:

- (1) 计算指令, 对操作数进行计算, 可能是 R 型或 I 型指令
- (2) 存取指令, 用于存取存储器, 指令格式与 I 型指令相同, 意义不同。
- (3) 分支和跳转指令, 它可能改变程序的执行过程, 可能是 I 型, 由当前地址加当前 PC 得到新地址, 也可能是 J 型, 绝对地址由指令低 26 位给出。
- (4) 协处理器及其 CPO 指令, 和专用指令, 未予实现。

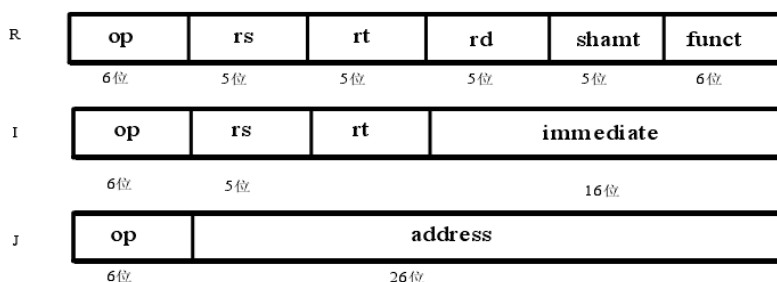


图 3-1 Mips 三种不同的指令类型

字段名	字段意义说明
op	指令操作码
rs	源操作数 1
rt	源操作数 2 或目的操作数
rd	目的操作数
shamt	位移量，在移位运算中存储位移量
funct	功能码，当指令为 R 类型时，根据功能码判断指令的操作类型
immediate	当其为 I 型指令时为立即数，分支指令时，为跳转偏移量
address	跳转绝对地址，为指令 J 的指令格式

图 3-2 指令字段说明

根据分析的嵌入式处理器指令类型，至少需要实现三种寻址方式：寄存器寻址、基址加偏移寻址，PC 相对寻址。

**寄存器寻址：**源操作数，或目的操作数为寄存器的指令，主要有 R 型指令，I 型指令，通过指明寄存器号来读取操作数。

**基址加偏移寻址：**数据传送类型指令 LW，SW，目的地址的产生通过寄存器与偏移量相加得到。

**PC 相对寻址：**目的地址通过 PC 与地址偏移相加得到，主要有分支指令，原理如图 3-3：

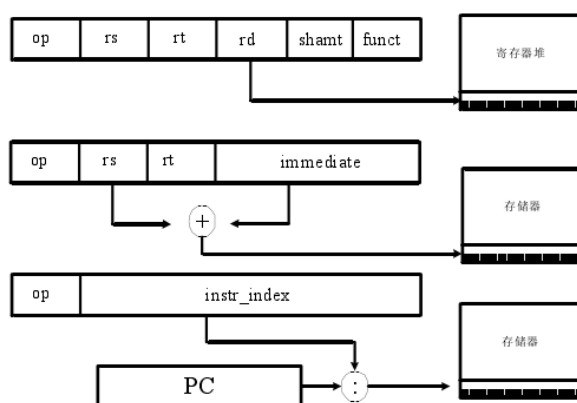


图 3-3 指令寻址方式

寄存器结构采用标准的 32 位寄存器堆，共 32 个寄存器，标号为 0-31。其中第 0 寄存器永远为全 0，第 31 寄存器是跳转链接地址寄存器。它在链接型跳转指令下

会自动存入返回地址值。对于其它寄存器，可由软件自由控制。在 MIPS 的规范使用方法中，各寄存器的含义规定见表 3-1。

表 3-1 寄存器堆使用规范

寄存器编号	助记符	用途
\$0	zero	常数 0
\$1	at	汇编暂存寄存器
\$2 \$3	v0,v1	表达式结果或子程序返回值
\$4-\$7	a0-a3	过程调用的前几个参数
\$8-\$15	t0-t7	临时变量，过程调用时不需要恢复
\$16-\$23	s0-s7	临时变量，过程调用时需要恢复
\$24 \$25	t8 t9	临时变量，过程调用时不需要恢复
\$26 \$27	k0 k1	保留给操作系统，通常被中断或例外用来保存参数
\$28	gp	全局指针
\$29	sp	堆栈指针
\$30	s8/fp	第 9 寄存器变量，过程调用时作为帧指针
\$31	ra	过程返回地址

以 MIPS 指令系统的简化版本为例，在简化版本中，实现如下指令：无符号加法和减法指令各一条：addu rd, rs, rt; subu rd, rs, rt; 都是 R 型指令。一个操作数为立即数的逻辑运算指令：ori rt,rs,imm16; 属于 I 型指令。两类指令的操作数要么是存储器，要么是立即数，因此还需要访问存储器的指令：lw rt, imm16(rs); sw rt, imm16(rs); 可在存储器和寄存器之间传送操作数。最后是一条分支指令，条件分支指令：beq rs,rt, imm16。

## 3.2 数据通路设计

### 3.2.1 选择合适的组件

确定指令集后，需要分析这样的指令系统对于数据通路有怎样的需求。首先通过对指令的各个位域进行分解，来看这些指令的含义。对于 R 型指令，参照图 3-1，一共分为六个位域，最高的 6-bit 称为操作码，后面连着 3 个位域都是 5-bit，各自标明了寄存器编号。然后 5 个 bit 在完整 MIPS 指令集中用于标记移位数量，最后 6 个 bit 是功能位域。

因此当取到一条 R 型 MIPS 指令时，可将其分为这样的六组控制信号。与之类似的，I 型指令 32 个二进制位被分为 4 组信号。这些指令编码都是从存储器中取得的，因此需要一个存放指令的指令存储器，不需支持写入的功能，可读即可。希望这个存储器具有如下功能：外界给出 32 位的地址输入，存储器给出 32 位数据输出。32 位地址需要有一个存放指令地址的 32 位寄存器，称为 PC（程序计数器）。有了这两个组件，就可以取得想要的指令了。

接下来从指令的操作来分析其它的需求。首先来看加法和减法指令。这两条指令的主体功能都是选择两个不同的寄存器，对它进行加法或减法的运算，然后将结果存到另外一个寄存器中。因此我们需要有一组存放数据的通用寄存器，每个寄存器都是 32 位的，而且我们可以约定这一组寄存器可以有 32 个。这样一组通用的寄存器我们称为寄存器堆。从加法和减法指令的操作还可以看出，在运算时，还需要同时读取两个寄存器的内容，这两个寄存器分别由寄存器位域中 rs 和 rt 这两个位域所指定。在运算完成后，我们还需要写入另一个寄存器。这个寄存器的编号由 rd 或者 rt 来指定。对于加法和减法运算都是由 rd 指定的；对于

立即数的计算，运算结果需要改写的寄存器需要由 *rt* 这个位域来指定。立即数运算指令的操作数除了 *rs* 寄存器外，另一个操作数是一个立即数。其中 16 位是直接填写在指令位域中的，但是我们的运算需要是 32 位的，因此还需要一个功能：将 16 位的立即数扩展到 32 位。对于这个运算我们做的是零扩展，也就是将高 16 位都填 0，从而构成一个 32 位的数。这三条都是运算指令，因此还需要支持不同的运算类型，在此我们需要提供具有加法、减法、逻辑或三种功能的运算器，这个运算器的操作数可以是两个寄存器，也可以是一个寄存器加一个扩展后的立即数，这些就是运算指令的需求了。

然后我们再看访存指令，对于 Load 指令，需要从存储器中读出一个字，这个字所在的存储单元的地址是由一个存储器的内容加上一个立即数来决定的。取出这个字之后，会把它存放在寄存器堆中由 *rt* 所指定的寄存器里，与之相对的还有 store 操作，该操作是将 *rt* 所指定的寄存器的内容传送到内存的指定地址的存储单元中，对于这两条访存指令我们又有什么需求呢？首先需要有一个能够存放数据的存储器，这个存储器既要可读又要可写，它的地址输入以及输入和输出的数据都应该是 32 位的，从地址的计算方法可以看出，也需要将 16 位的立即数扩展到 32 位，但扩展方法是符号扩展。也就是将立即数作为低 16 位，并将最高位复制到高 16 位当中，从而形成一个 32 位的立即数。这就是访存指令的主要需求。

最后是分支指令。对于分支指令首先要判断两个寄存器中的内容是否相等，如果相等，就将指令位域中立即数的部分经过一个简单的变化加到 PC 上从而得到新的 PC，如果比较的结果不相等，那就直接 PC 加 4，从而产生新的 PC。因此分支指令的需求首先能够比较两个寄存器的内容，判断是否相等，然后还需要 PC 寄存器支持两种自增的方式，一种是加 4，一种是加一个指定的立即数，对于 PC 加 4 这个需求对于前面介绍的其它指令也都是需要的。

下面再把指令系统的需求整理一下：

(1) 首先是一个算术逻辑单元 (ALU)，要支持加法、减法、逻辑或和比较相等这样的操作。有 2 个 32 位的输入，可以来自寄存器或扩展后的立即数。

(2) 然后还需要一个立即数扩展部件，可以将一个 16 位的立即数扩展为 32 位，扩展方式可以是零扩展也可以符号扩展。

(3) 还需要一个程序计数器 (PC)，这是一个 32 位的寄存器，由时钟控制，而且还需要支持两种加法运算，加 4 或加一个立即数。这样的需求可以用 ALU 实现，也可以配上简单的加法器。

(4) 除此之外还需要两个带有存储功能的组件：寄存器堆和存储器。对于寄存器堆，我们一共需要 32 个 32 位的寄存器，需要支持同时读出两个寄存器，和写入一个寄存器，这样的寄存器堆称为两读一写寄存器堆。对于存储器，我们需要一个只读的指令存储器，地址和数据都是 32 位，还需要一个可读写的数据存储单元，地址和数据也都是 32 位的，在处理器内部往往会设计高速缓存，也就是 Cache 这样的部件，用来保存内存中的一部分数据，高速缓存这个结构会分成指令和数据两个部分，因此这里我们选择分开的结构。**要注意这两个存储器实际对应了 CPU 中的指令和数据高速缓存，而不是整个计算机中的内存。**

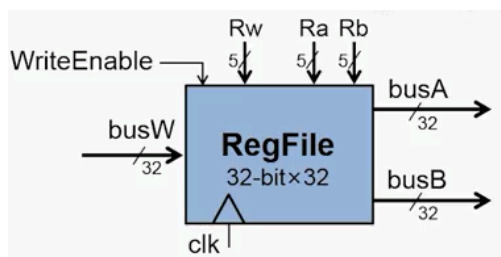


图 3-4 寄存器堆的结构示意图

图 3-4 是寄存器堆的结构示意图，内部有 32 个 32 位的寄存器，有 3 组数据接口，其中



busA 和 busB 是两个 32 位的数据输出接口，busW 是 32 位的数据输入接口，如何对寄存器堆进行读写呢，需要读写控制信号，首先是 Ra，这是一个 5 位的信号，5 位的信号正好可以选择编号 0-31 的寄存器，因此寄存器堆会根据 Ra 的输入，选择对应编号的寄存器，将其内容放在 busA 信号上，同样根据 Rb 选中对应编号寄存器，将其内容放到 busB 信号上。这样外界将两个编号分别为 Ra 和 Rb 信号放在寄存器堆输入端，寄存器堆就会将寄存器的内容分别放在 busA 和 busB 上。这就完成了同时读取两个寄存器的功能。写寄存器操作稍微复杂一些，首先将要写的寄存器的编号通过 Rw 信号输入，在时钟信号的上升沿如果写使能信号有效（WriteEnable==1），寄存器堆就会将 busW 的内容存入 Rw 信号指定的寄存器。寄存器堆的读操作不受时钟控制，即在任何时候 Ra 或者 Rb 发生了变化，对应的 busA 或者 busB 的信号就会发生变化。

然后我们来看存储器，存储器有两组数据接口信号，有 32 位的数据输入和 32 位的数据输出。存储器的读写和寄存器堆有些类似，只要给出一个地址，存储器就会将对应地址单元中的数送到数据输出信号上，而与寄存器堆不同在于只需给出一组输入信号，而不是寄存器堆的两组数据输入，从这个角度来说，存储器是一个一读一写的存储器。另一个方面，我们给出的地址信号是 32 位的，所以理论上存储器可以达到 2 的 32 次方那么大，当然这只是理想化的情况，对于存储器的写操作，也提供了一个写使能信号，在时钟上升沿到来时，如果写使能为 1，存储器就会将数据输入信号的内容存入地址信号所指定的存储单元中。同样需要注意，存储器的写操作是在时钟上升沿发生的，而存储器的读操作不受时钟信号的控制。只要输入的地址信号发生变化，需要很短的时间输出的数据信号随之发生变化。现在完成了指令系统的分析，得出对数据通路的需求，为数据通路选择了合适的组件，接下来开始建立数据通路的工作。

### 3.2.2 建立数据通路

现在有了指令系统的总体需求，准备好了关键的组件，我们看如何将这些关键组件拼接起来构造出一个完整的数据通路。

要建立一个数据通路，基本原则是分析指令系统中每一条指令需求，并根据指令的需求连接已经选好的组件。指令的需求又分为两大类，有些需求是所有指令共同需要的，不同的指令也有一些各自特殊的需求。

首先来看所有指令的共同需求：要执行指令，首先需要取指令，指令是放在指令存储器中的，要访问指令存储器取得指令，就得有一个地址，这个地址是存放在 PC 寄存器中的，我们已经有了一个 32 位的 PC 寄存器，把 PC 寄存器的输出作为指令地址连接到指令存储器的输入端，指令存储器会根据地址的输入选中对应的地址单元，并将其内容输出。这样我们就得到了所需的那条指令的二进制编码。除了取当前指令外，还要为取下一条指令做好准备。这就需要更新 PC 寄存器，分成两种情况：

(1) 大多数情况指令是顺序执行的，PC 只需要加上当前这条指令的长度就可以得到下一条指令的地址，在 MIPS 指令中，每条指令都是 4 个字节的，所以只需要简单的进行 PC 加 4 的运算，增加一个加法器，PC 的输出连接到加法器的一个输入端，另一端为常数 4，PC 的输出一方面送到指令寄存器输入端，另一方面送到加法器计算出 PC 加 4 的值，在下一个时钟上升沿到来时，PC 寄存器中就会将 PC 加 4 的值存入其中，然后再将这个更新后的内容同时送到指令存储器和加法器，如此周而复始就完成了每个时钟上升沿更新 PC 寄存器的内容，然后指令存储器就会送出新一条指令的二进制编码。



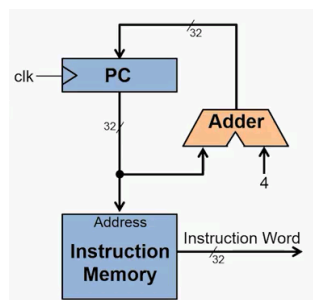


图 3-5 PC 加 4 的数据通路

(2) 如果遇到分支情况，下一条指令地址就不是 PC 加 4，而是由分支指令进行指定。因此需要修改图 3-5 的结构，增加一个二选一的多选器，如图 3-6 所示，在顺序执行时，选择 0 输入端输入，在分支执行时，选择 1 输入端的输入。在下一个时钟上升沿到来时，PC 寄存器就会采样多选器的输出，并将其保存起来，这样的结构，就完成了不断取回指令的功能，称为取指部件，简称 IFU，IFU 作为一个整体，如图 3-7，从外界只有一个时钟信号的输入和一个多选器选择信号的输入，并且提供指令编码的输出。我们只要在启动时给 PC 寄存器一个合适的初始值，并在指令存储器中存放好我们需要运行的指令，然后在运行过程中给出合适的多选器选择信号  $nPC\_sel$ ，这个 IFU 就可以在时钟信号的驱动下连续的工作起来了。这些就是所有指令的共同需求。

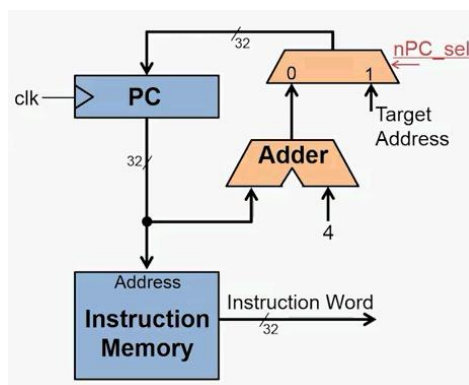


图 3-6 遇到分支情况的数据通路

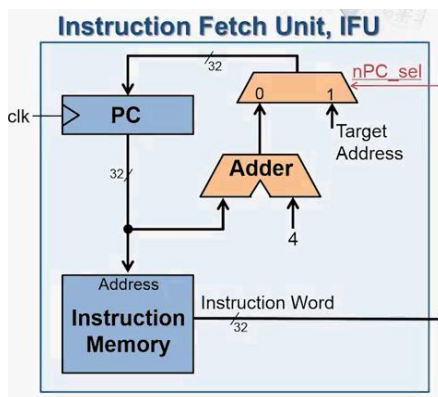


图 3-7 取指部件

然后根据指令的不同类别，分析其各自需求。首先来看加法和减法指令，这两条指令都是 R 型指令，它的主体操作是用  $rs$  和  $rt$  两个信号访问寄存器堆，取得对应两个寄存器的内容，并对它们执行相应的运算。这些运算目前我们提供了加法、减法两种，将计算的结果放到另一个五位信号  $rd$  指定的寄存器中，因此需要用到寄存器堆这个组件，这是一个两读一

写的寄存器堆，两个需要读出的寄存器编号是由指令编码中 *rs* 和 *rt* 这两个位域指定，而要写入的寄存器的编号由 *rd* 这个位域指定的。所以只需要将这三个位域信号对应的值连接到这三个输入上，这样在寄存器堆的输出端，*busA* 就会输出 *rs* 指定的寄存器内容，*busB* 输出 *rt* 指定的寄存器内容，然后将 *busA* 和 *busB* 连到 ALU 的输入端，并且根据指令编码中的操作码和功能位域就会知道当前指令是加法还是减法指令。我们要通过一个控制信号来选择当前 ALU 能够提供的运算的类型，然后将 ALU 的输出连接到 ALU 的输入端，也就是 *busW* 信号，在下一个时钟上升沿到来时，如果寄存器堆的写使能信号有效，寄存器堆就会采样 *busW* 信号的内容，并将其存入到 *rd* 信号所指定的寄存器中，这样就在一个时钟周期完成了一个加法或者减法的指令。红色标明的两个信号都称为控制信号。

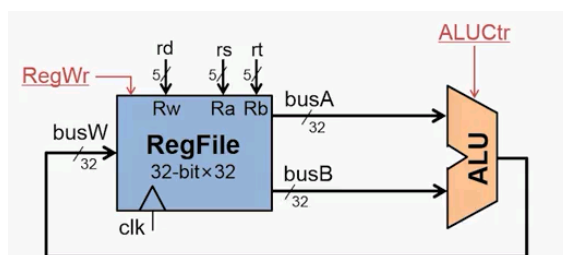


图 3-8 加减法指令的部分数据通路

然后我们来看逻辑运算指令的要求，在简化版本中提供的逻辑运算就是对立即数进行或操作的指令。这条指令是一条 I 型指令，刚才建立的数据通路是否能满足逻辑运算指令的要求呢，经过分析，可以发现几个问题：（1）逻辑运算指令目的寄存器是 *rt* 而不是 *rd*，而在图 3-8 中寄存器堆的输入端连接的是 *rd* 这个位域，这样没法满足这条指令将运算结果写入的 *rt* 这个寄存器中的目的。（2）这条指令的操作数是一个立即数，而我们现在 ALU 的两个输入都是来自寄存器堆，对这一点也无法支持。（3）在指令中只提供了一个 16 位的立即数，ALU 的输入都是 32 位的，所以还需要对这个 16 位的立即数进行 0 扩展，针对这三个问题需要对现有的数据通路进行改造。针对问题（1）增加一个二选一的多选器，当执行之前的加法或减法指令时，让多选器的选择信号为 1，这时就会和刚才一样，将 *rd* 信号的内容传递到 *Rw* 的信号端，而在执行现在这条逻辑运算指令时，可以选择多选器的 0 号输入端，将 *rt* 信号的值传送到 *Rw* 的信号端。这就为写入 *rt* 所指定的寄存器提供了支持。针对第（2）个问题，在 ALU 的输入端增加了一个多选器，对于之前的功能，通过多选器的 0 输入端将寄存器堆的 *busB* 的信号和 ALU 的输入端相连，而对于这条逻辑运算提出的新的需求，可通过多选器的 1 号输入端将立即数与 ALU 相连，这个 16 位的立即数需要经过一个 0 扩展部件扩展成 32 位再接到多选器的输入。这样通过增加两个多选器和一个 0 扩展部件满足了逻辑运算指令提出的新需求，如图 3-9 所示。在满足新需求的同时需要保证原来的需求还是依然得到满足的。

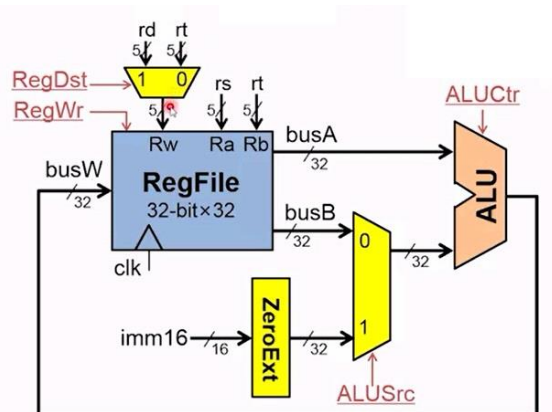


图 3-9 根据逻辑运算指令需求改进后的数据通路

访存指令的需求。访存指令也是 I 型指令。Load 指令是要对数据存储器进行访问。访存的地址由 rs 指定的寄存器内容加上立即数进行符号扩展后的值，而且访存得到的数据也要保存在 rt 所指向的寄存器中，现在的数据通路已经支持了写入 rt 所指向的寄存器了，而访存地址的计算是一个寄存器加上立即数，现在的数据通路也大体支持这样的功能，但仍然是有问题的：首先还不支持符号扩展，其次 ALU 运算的结果应该作为地址去访问存储器，从而获得数据而不是直接连到寄存器堆的写入端，因此还需要对通路进行改造。针对符号扩展的要求，将原来 0 扩展的部件改为多功能扩展部件，通过控制信号选择进行 0 扩展或者进行符号扩展。对于这条 Load 指令可以选择将 16 位的立即数符号扩展为 32 位的数，并通过多选器连接到 ALU 输入端，ALU 的另一个输入端则是 rs 所指定的寄存器的内容，执行加法运算后获得了存储器的地址，在这里需要新增一个数据存储器，这个存储器根据地址就可以得到对应的存储单元的数据，因为需要将这个数据写入到寄存器堆，因此还需要增加一个多选器，将数据存储器中输出的内容传输到寄存器堆的输入端，这就是 Load 指令提出的需求。我们的解决方案是将原来 0 扩展部件增加符号扩展的功能，并增加一个数据存储器，当然还包括相应的多选器。

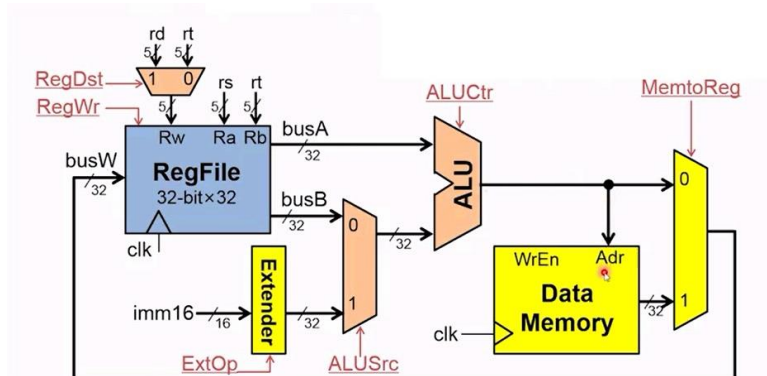


图 3-10 根据 Load 指令需求改进后的数据通路

还有另一个访存指令 Store 指令，其地址运算方式和 Load 一样。不同的是 store 指令是将 rt 指定的寄存器的内容存入到数据存储器中。再看数据通路，首先需要 rt 所指定的寄存器的内容，rt 所指定的内容会从 busB 信号上出来，需要把这个信号连接到数据存储器的数据输入端，但是对于除了 store 外的其他指令都不希望 busB 的信号写入到数据存储器中，所以需要给数据存储器连接一个控制信号，只有控制信号有效的时候才将信号输入，这样就满足了 store 指令的需求，如图 3-11 所示。

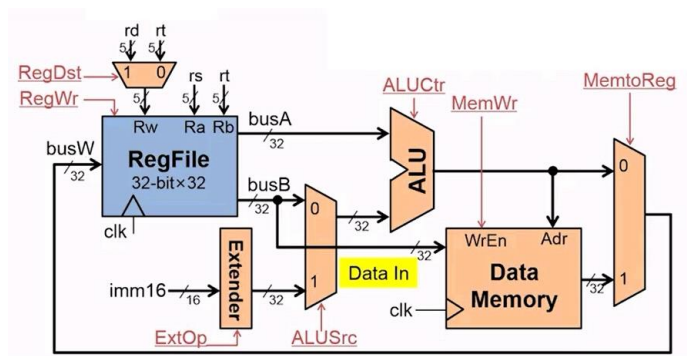


图 3-11 根据 Store 指令需求改进后的数据通路

现在除了比较特殊的分支指令的要求，我们已经连接了组件。现在在处理器的设计步骤中，我们完成了第三步。不过数据通路是否完整了？是否可以一步步正常工作了？这些需要通过一条条指令进行更细致的检查和确认。

### 3.3 控制信号生成

通常在一个指令系统中，运算指令是最为重要的，从另一个方面看，运算指令的执行过程又是最为简单明了的。下面先来看运算指令所需的控制信号是怎样构成的。

MIPS 的指令系统从功能上划分可以分为运算指令、访存指令和分支指令。而从指令格式上划分，分为 R 型 I 型 J 型指令。在本章的示例中，经过简化后只有 6 条指令如图 3-12 所示。

运算指令	addu rd,rs,rt subu rd,rs,rt	ori rt,rs,imm16	
访存指令		lw rt,imm16(rs) sw rt,imm16(rs)	
分支指令		beq rs,rt,imm16	
	R型指令	I型指令	J型指令

图 3-12 简化后的指令分布

R 型的运算指令，加法和减法指令操作非常相似，下面以加法指令为例进行分析说明。加法指令的执行分为三步，首先是从指令存储器中取出指令；然后将 rs 寄存器和 rt 寄存器的内容进行相加，运算结果保存在 rd 指定的寄存器中；最后是计算下一条指令的地址：执行完加法指令，下一条指令是顺序执行的，因此将 PC 加 4 更新到 PC 中。依次来看这三个步骤

(1) 首先是从指令存储器中取回指令，这个不是加法指令所特有的，对于所有的指令都需要有这个步骤。这个步骤是在取指部件 IFU 中完成的。假设系统中的某个时刻，选取  $t_0$  上升沿后很短的时间，这时 PC 寄存器的输出已经稳定，它的输出就是当前要取回的指令的地址，这个信号会连接到指令存储器的地址输入端，经过一小段访问时间后，指令存储器就会将这个地址指定的存储单元的内容放到其输出端，也就是要取回的指令的二进制编码。如图 3-13。



图 3-13

(2) 然后进入第二步执行加法运算，但需要先取得操作数，操作数所在的寄存器的编



号是放在指令编码中的，现在 IFU 已经送出了指令编码，可以从 IFU 送出的 32 位信号选取对应的信号部分，如图 3-14 所示，第 25 到 21 位组成一组信号，记为 rs，第 20 位到 16 位记为 rt，第 15 位到 11 位记为 rd。这三个信号就被连接到寄存器堆的输入端。看寄存器堆的输入信号：rs 被连接到寄存器堆 Ra 的输入端，rt、rd 这些信号也都连接到寄存器堆标了 rt 和 rd 的地方。根据寄存器堆的设计特性，随时会根据 Ra、Rb 这两个端口的输入信号找到对应编号的寄存器并放到输出端，因此一旦 IFU 取回了这条加法指令，rs 所对应的寄存器内容就放到 busA 信号线上，rt 所对应的寄存器内容就放到 busB 信号线上，接下来对这些数做哪些操作需要由控制信号来决定了。首先看给 PC 寄存器的这个控制信号，因为这是一个加法指令，给出下一个 PC 的选择信号应该是选取 PC 加 4 的通道，用“+4”来表示。看一下寄存器堆的两个输出，busA 直接连到了 ALU 的一个输入端，busB 连到了一个多路器，这个多路器负责为 ALU 的另一个输入端选择输入信号，对于这条加法指令，正是需要从寄存器堆输出的 busB 信号，所以多路器对应的选择信号应该是 0；这样这个多路器的 1 号输入的通道信号就没有意义了。而产生这个信号的是立即数的扩展部件也有一个控制信号，决定它是 0 扩展还是符号扩展，现在这个控制信号设置成什么都可以，用 X 来标记。现在 ALU 的两个输入端都已经连接上了两个正确的信号，还需要给 ALU 一个控制信号，以便 ALU 执行加法操作，这个控制信号以“ADD”标记。经过一段运算时间后 ALU 就会输出这两个数运算的结果，这个结果会被送到两个地方，其中一个数据寄存器的地址输入端，通过分析这条加法指令是不需要读写数据寄存器的，因此把数据寄存器的写使能信号置为无效，否则在下一个时钟上升沿，数据存储器会采样它的数据输入，从而改变其中的内容，这不是我们希望达到的结果，但即使把写使能信号置为 0，数据存储器依然会根据地址的输入选择对应存储单元的内容送到输出信号线上，因此还需要正确的设置最后这个多路器的控制信号，把它设置成 0，这样多路器就会将 0 输入端的输入放到多路器的输出上，这个信号最终会被连到寄存器堆的输入端。我们希望在下一个时钟上升沿到来时寄存器堆会采样其输入端的信号，并保存到 rd 所指定的寄存器中，因此首先需要将寄存器堆的写使能信号置为有效，而要写入的寄存器的编号需要在 rd 和 rt 中选一个，根据这条指令的需要应该选择 rd，所以这个多路器的控制信号应该设为 1，这样所有的控制信号都已经设置完成了。时间也过去了一小段，只要所有控制信号都设置正确，从寄存器堆输出的两个寄存器的值就会经过 ALU 完成加法。然后经过多路器最终送到寄存器堆的输入端。经过一小段时间后输出信号都会变的稳定。在下一个时钟上升沿到来时寄存器堆就会采样 busW 的信号，选中 rd 所指定的寄存器，将采样的信号保存在其中，这样就完成了这条加法指令这一步操作。

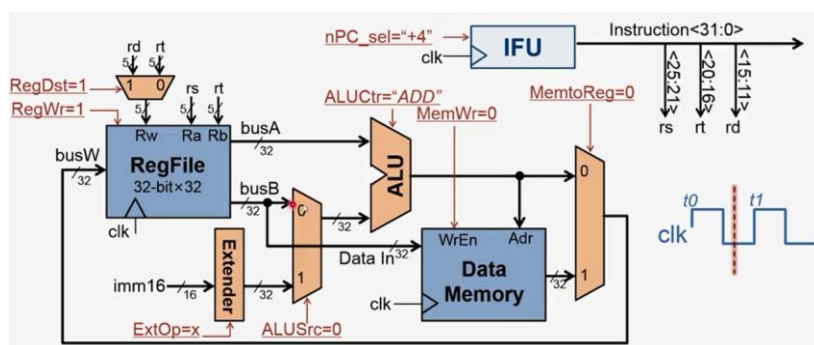


图 3-14 加法指令控制信号

(3) 不过这条指令还没有完全完成，还有第三步：更新 PC 寄存器的内容。除了分支指令，其它指令都应该是顺序执行的，因此都应该用 PC 寄存器的值加 4 作为下一条指令的地址。注意时钟信号的波形，虽然这是第三步，但是和第二步完成的时间是一样的。根据 IFU 的结构图，在第一步提到的 PC 将输出送到指令存储器的地址输入端，与此同时，PC

的输出也会送到加法器的输入端，加法器的另一个输入端固定连了常数 4，这个加法器的输出就会计算出 PC 加 4，当指令取出后，经过一小段时间就会产生对应的控制信号，对于这条加法指令，这条  $nPC\_sel$  信号就会被设为 0 也就是刚才标记的“+4”，所控制的多选器就会选通 0 号通道，将 PC 加 4 送到输出，并最终连到了 PC 寄存器的输入端，在下一个时钟上升沿到来时，也就是图中标记的  $t_1$  时刻，PC 寄存器就会采样它的输入，从而把 PC 加 4 存到寄存器中，再过一小段时间，也就是到了  $t_1$  上升沿之后的一小段时间，PC 寄存器的输出就会稳定，从而把递增了 4 后的地址再次送到了指令存储器地址输入端，取出了下一条指令，这样取指并执行再取指再执行这个过程就会自动进行下去，而且每一个指令都是在一个时钟周期之内完成的。

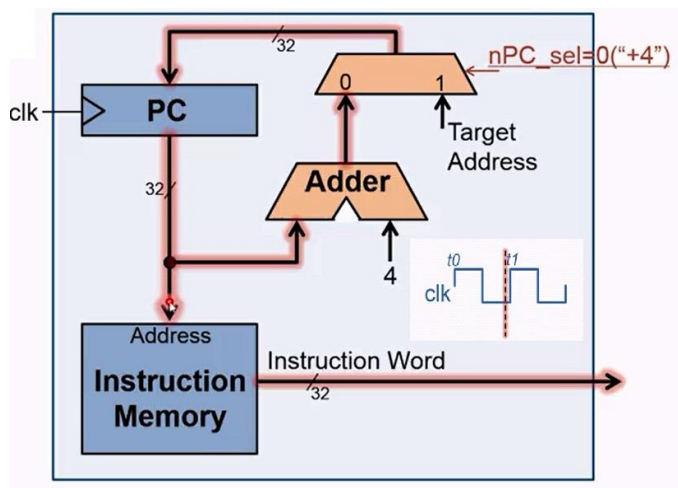


图 3-15 更新 PC 寄存器的控制信号

接下来再来看另一类指令，也就是运算指令中的 I 型格式的指令，在本章的简化指令集中，这类指令只有一条，下面分析这条指令的操作步骤。同样分为 3 步：第一步从指令存储器中取回指令；第二步进行逻辑或运算，这个运算的一个操作数是由指令编码中  $rs$  域所指定的一个寄存器，另一个操作数则是指令编码中 16 位的立即数，经过 0 扩展后得到的 32 位数，这个或运算的结果最终会保存到指令编码  $rt$  位域所指定的寄存器中；第三步是将 PC 寄存器的值加 4 再更新到 PC 中。第一步和第三步和加法指令一样，只有第二步不同，结合数据通路来看，刚才从 IFU 中选出了三组信号，现在需要再多一组信号，低 16 位组成 16 位立即数信号，这个信号会被连接到数据通路上的扩展功能部件，如图 3-16。

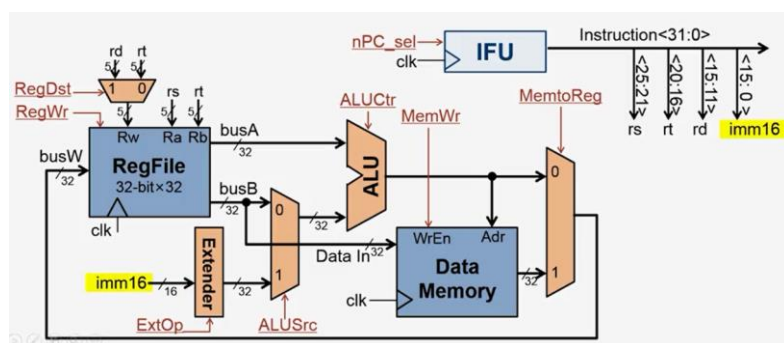


图 3-16 低 16 位立即数信号连接

对于这条指令，控制信号该如何设置呢？首先来看 IFU，对于 `ori` 指令，它的下条指令应该依然顺序执行的。所以  $nPC\_sel$  信号依然被设置为“+4”，和加法指令是一样的，这条指令与加法指令最大的不同是 ALU 的第二个操作数是一个立即数，而不是来自于寄存器堆，所以 ALU 第二个输入端选择信号来源的多选器的控制信号应该选择 1，立即数作为来源，

这 32 位的立即数是由指令编码中 16 位的立即数扩展而得的。根据指令要求，需要 0 扩展，因此扩展部件的控制信号应该被设置为 0 扩展。然后看 ALU 的运算功能选择信号，应该控制 ALU 进行或运算。数据存储器的写使能信号由于这条指令不需要写数据存储器，所以应该被设为 0，然后看最右边的多路器，由于 ALU 的输出也就是或运算的输出送到寄存器堆，所以多路器应该选择 0 号通道，这样 ALU 的输出结果就会追踪连接到了寄存器堆的输入端。与此同时还要设置寄存器堆的写使能信号有效，而且从指令要求可以看出，目的寄存器是 *rt* 这个位域指定的，所以对于寄存器堆写入寄存器的编号应该选择 *rt* 这个来源，因此需要将这个多路器的选择信号设为 0，这样所有的控制信号都设置好了，如图 3-17 所示。

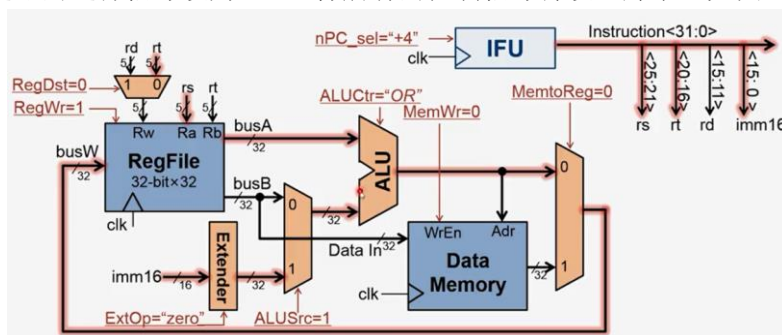


图 3-17 ori 指令的控制信号

上面我们已经看到运算指令的控制信号是如何产生的。只有运算指令是不够的，运算指令不能直接访问存储器的。因此需要设计单独的访存指令来完成寄存器和存储器的数据传输。下面分析访存指令的控制信号如何生成的。

对于要实现的存储器，访存指令都是属于 I 型指令。对于 Load 指令，首先也是取指；然后根据运算出的地址访问数据存储器，并将读出的内容写入 *rt* 指定的寄存器中；最后是更新 PC 寄存器。第一步和第三步与运算指令都是一样的，只需要设计第二步相关的操作。

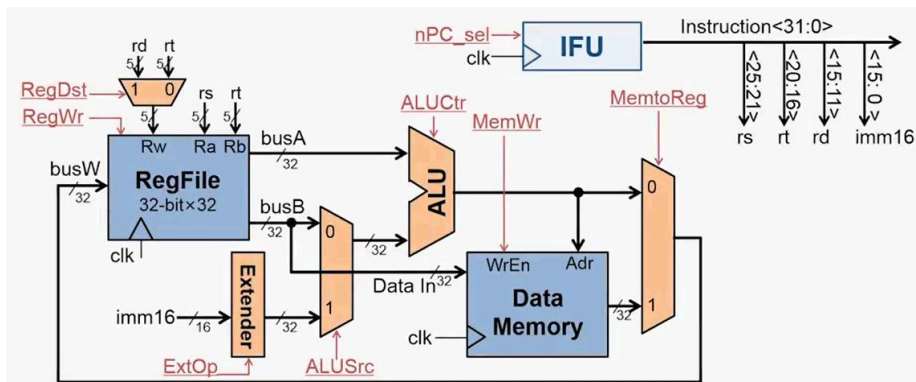


图 3-18 数据通路图

如图 3-18 是目前的数据通路，如果当前 IFU 取回的是一条 Load 指令，这些控制信号将如何设置呢？首先下一条指令的地址还是采用 PC 加 4 的方式，*busA* 和 *busB* 信号线上分别是由 *rs* 和 *rt* 所指定的寄存器的内容。但要注意这条指令运算的是 *rs* 寄存器中的内容和立即数的符号扩展进行加法，因此 ALU 另一个操作数的来源通过多路器选择通道 1，并且设置扩展部件为符号扩展的功能，这样指令位域中的 16 位立即数就会经过符号扩展连接到 ALU 的另一个输入端，而发送到 ALU 的控制信号则是要设置为加法运算，这样 ALU 就会完成访存地址的运算，并将地址信号送到数据存储器的输入端。同时 *busB* 的信号也会被连接到数据存储器的数据输入端。虽然我们并不需要它，但是这条信号线仍然会把 *rt* 寄存器的内容送过来，所以需要设置数据存储器得的写使能信号为 0，保证数据存储器内容不会发生改变。然后来看最后这个多路器的输入，一个是 ALU 的运算结果即访存的地址，另一个是从



数据存储器中读出的数据，对于这条指令，需要把后者送到寄存器堆中去，因此对于这个多选器来说需要设置选择信号为 1，把数据存储器的输出信号送到寄存器堆的输入端，而且还要写入到 *rt* 所指定的寄存器，需要设置寄存器堆的写使能信号为有效，并把写入寄存器编号来源设置为 *rt*，这样在下一个时钟上升沿到来时，数据存储器内容就被写入到寄存器中去了，而且在同样的时钟上升沿，PC 寄存器的内容也会被更新为 PC 加 4。这就是 *load* 指令所完成的操作。

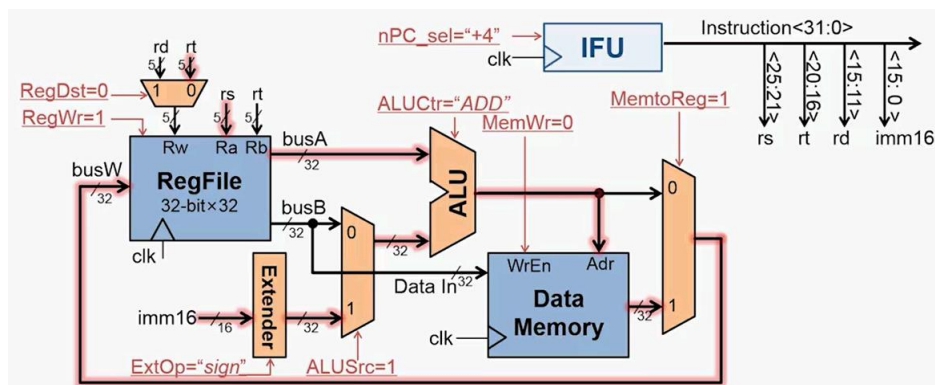


图 3-19 Load 指令的控制信号

然后我们来看 *store* 指令，该指令操作也是分为三步，同样只看其中的第二步。对于 *store* 指令，也需要将 *rs* 寄存器内容与立即数符号扩展进行加法运算，并以此作为地址访问数据存储器，不过不是读出，而是写入。因此对应到数据通路上直接标出了对于 *store* 指令有效的这些信号，这些控制信号如何设置呢？对于 IFU，需要选择下一条指令的地址，仍然是 PC 加 4。选择 ALU 数据的输入，现在选的是立即数的符号扩展，对于 ALU 依然要选择加法的运算类型，ALU 的运算结果依然是地址，会被送到数据存储器上，但是与之前不同的是这条指令要完成向数据存储器的写操作，因此把数据存储器的写使能信号 *MemWr* 置为有效，这样在下一个时钟的上升沿，数据存储器就会采样数据存储器输入端的信号，输入端信号是 *rt* 这个位域指定的寄存器内容。因此在下一个时钟上升沿到来的时候，数据存储器会根据 ALU 运算出来的地址选中对应的存储单元并采样来自 *rt* 寄存器中的内容，将它存到对应的存储单元中。该指令的操作完成了，但是需要将剩余的控制信号设置完整。对于后面的多选器来说，控制信号可以是 0 也可以是 1，用 *x* 表示，设置寄存器堆的写使能信号 *RegWr* 为无效，在时钟上升沿到来时，寄存器堆的内容就不会发生改变，也正因为如此，这个要写入的寄存器编号设置成 *rt* 或者 *rd* 也就无所谓了，因此 *RegDst* 为设置为 *x*。这就是对 *store* 指令的控制信号的设置方法。如图 3-20 所示。

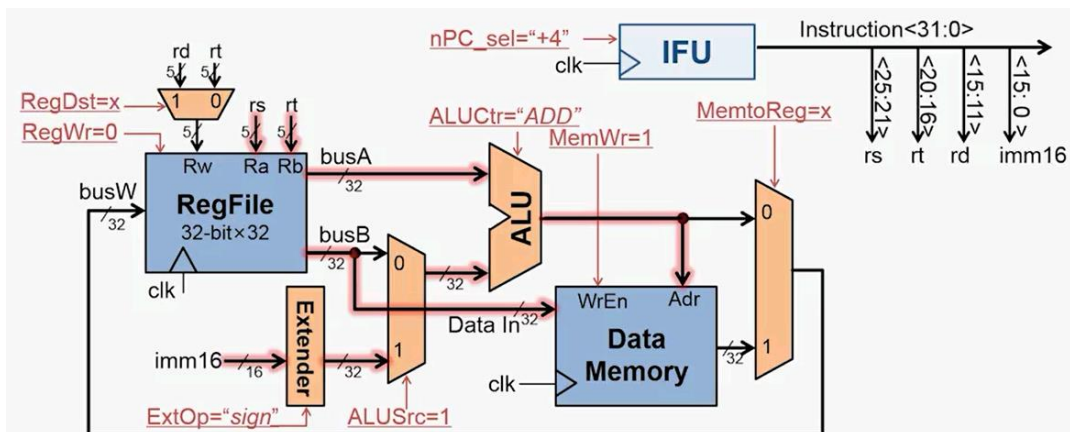


图 3-20 Store 指令的控制信号



现在我们已经分析了访存指令是如何生成控制信号的,再加上前面的运算指令我们就可以编写程序让计算机完成一定的任务了。不过仅有这两种指令是不够的,还需要让计算机改变流向的分支指令。

分支指令是一类特殊的指令,可以改变程序的流向。因此要执行分支指令,还需要进行下一步改造。下面是条件分支指令的一个示例:

```
If(i==j)
f = g + h;
else
f = g - h;
编译成 MIPS 汇编语言代码:
beq  $s3, $s4, True   # branch i==j
sub  $s0, $s1, $s2     # f = g - h (false)
j    Next   # goto Next

True: add $s0, $s1, $s2  # f = g+h (true)
Next: ...
```

第一条语句就是一条分支指令,比较  $s3$  和  $s4$  寄存器的值,如果相等就转到 **True** 标号所标明的地址,也就是  $c$  语言代码中  $if$  条件为真时所要执行的语句。执行完这条语句后,程序将顺序执行后面的内容。如果  $if$  的判断条件不成立,对应的这条 **beq** 指令执行时发现  $s3$  和  $s4$  寄存器内容不相等,从而不发生分支转移,而是顺序执行后一条指令。后一条指令是一条减法指令,与刚才的加法指令相对应,它执行的就是  $C$  语言中  $f = g-h$  这条语句。执行完这条指令后,下一条指令是无条件转移指令,直接跳到 **Next**。

下面我们分析 **beq** 这条指令的控制信号是如何生成的。**Beq** 指令的操作同样可分为三步:第一步是取指令;第二步是判断  $rs$  和  $rt$  两个寄存器内容是否相等,可以用减法来判断;第三步是更新  $PC$  寄存器。对于 **beq** 指令来说,所谓的分支就是如何改写  $PC$  寄存器,所以它的重点在于第三步。

首先来看条件不成立的情况,也就是 **else** 对应的  $PC = PC + 4$ ,在条件不成立时,顺序执行后条指令,这就和其它的运算指令访存指令是一样的;如果条件成立时, $PC$  的更新条件相对复杂些,其中也有  $PC$  加 4,然后加上 16 位立即数的符号扩展并乘以 4,也就是说在 **beq** 这条指令种所带的立即数,也就是示例中的目标标号 **True**,实际的数值是转移目标地址和下一条指令地址之间的差值,而且这个差值是以 4 个字节也就是 32 位为一个单位的。这个规则是在制定 **MIPS** 指令系统的时候约定的,现在重点分析如何实现相应的控制信号。

同样,直接来看第二步,对于这一步要做的操作包括从寄存器堆中取出两个寄存器的内容进行减法运算,这个操作和之前的减法指令的需求是一样的。因此现有结构不需要修改就可以实现这个操作。不同的是,减法指令会将经过  $ALU$  后的运算结果通过多选器写回到寄存器堆,而 **beq** 指令只需要判断结果是否为 0,是不需要写回到寄存器堆的。在  $ALU$  中增加判 0 电路,输出是否为 0 的判断信号,命名为 **zero**。如图 3-21 所示,如果运算结果为 0,  $ALU$  会把 **zero** 信号置为 1,否则置为 0。因为运算结果是否为 0,将会影响到  $IFU$  如何去更新  $PC$  寄存器,因此可以把 **zero** 信号连接到  $IFU$  的输入端。这样就可以把第二步操作的描述补充完整。

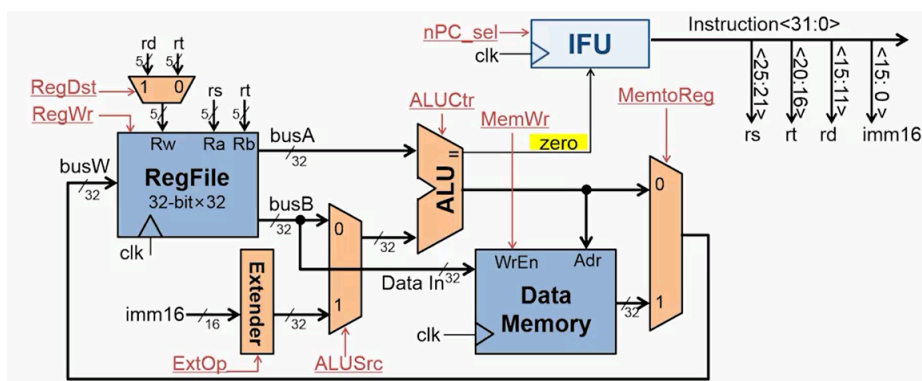


图 3-21 zero 信号的连接

那么控制信号又是如何设置的呢？首先下一个 PC 的设置方式就不能再设置为“+4”了，但是究竟如何更新 IFU 呢？下面先把选择信号标记为“branch”，然后再来看其它的控制信号。ALU 要执行一个减法运算，而且它的两个操作数都应该来自寄存器堆，busB 的多选器就应该选择 0 号通道，扩展部件的控制信号可以任意设置，而 ALU 的功能选择信号需要设置为减法，由于原来 ALU 的输出功能还保持，因此输出结果依然送到数据存储器的输入端和多选器的输入端。为保证数据存储器内容不被改写，还要设置数据存储器的写使能信号为 0，因为该指令不需要改写寄存器堆，因此多选器无论选择哪个通道都是没有意义的。可以把它的选择信号任意设置为 0 或者 1。最后因为不需要写寄存器堆，因此寄存器堆写使能设置为 0，写入寄存器的编号便可以任意设置。这样可以看出 beq 指令执行时真正有效的信号。不过这一步仅完成了判断，下面还需要根据判断结果完成对 PC 寄存器更新，与其它指令的第三步有所不同。

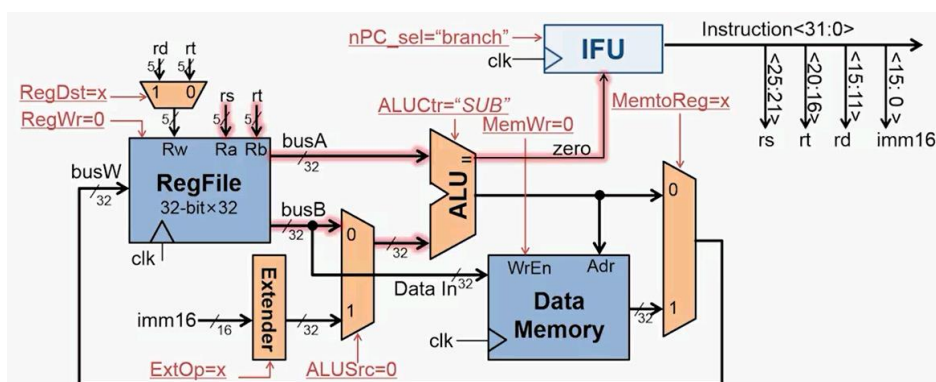


图 3-22 beq 指令的控制信号

对于 IFU 如何更新 PC 寄存器，关键问题是多选器如何选择。它的 0 号通道连接的是 PC 加 4，1 号通道连接的是分支指令的目标地址，现在这个选择信号如何生成呢？

可以把输入列一个表进行观察，如图 3-23，当 nPC\_sel 为 0 时，表示当前指令为运算指令或访存指令，而不是分支指令，这时，无论 zero 信号是 0 还是 1，多选器都应该选择 0 号通道，从而顺序执行下一条指令。当 nPC\_sel 信号为 1 时，说明当前正在执行一条分支指令，但如果此时，zero 信号为 0，表示分支的判断条件不成立，这个多选器仍然应该选择 0 号通道，从而顺序执行下一条指令，只有当 nPC\_sel 信号为 1 说明当前是一条分支指令，而且 zero 信号也为 1，说明当前判断条件成立，这时多选器才可以选择 1 号通道，从而将分支的目标地址更新到 PC 寄存器中。这样在下一个时钟周期，指令存储器就会将分支目标地址指向的指令的编码送出，实现指令流向的改变。通过图 3-23 中的表请思考多选器控制信号的生成方法。

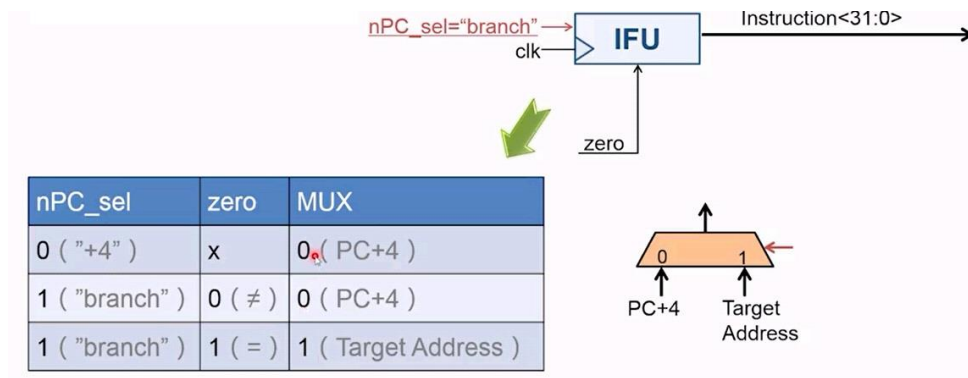


图 3-23

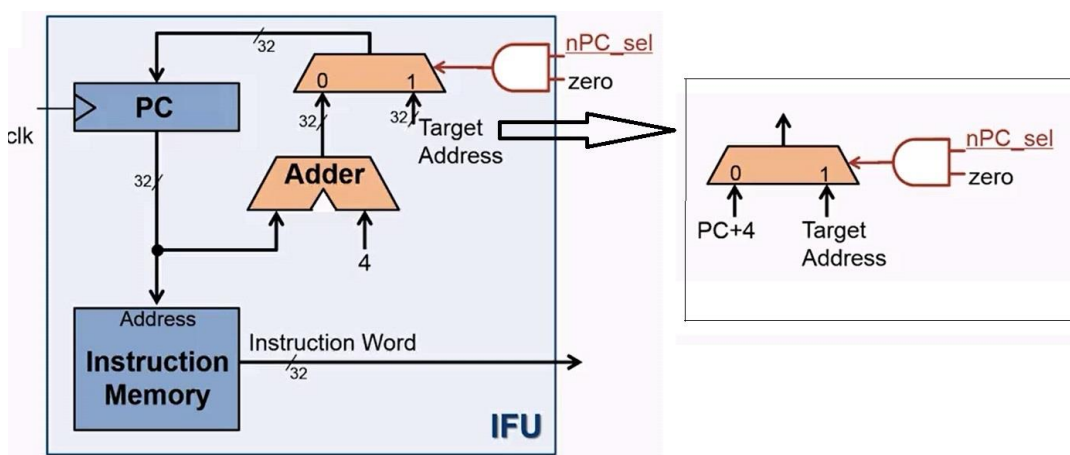


图 3-24

其实，只需要增加一个与门就可以实现表中的逻辑功能，如图 3-24 所示。这样完成对 IFU 模块改造从而支持 Beq 指令的需求。不过这里还有一个问题，分支目标地址究竟是如何生成的，一部分是 PC 加 4，一部分是对立即数进行符号扩展然后乘以 4，这个立即数就是指令编码中的低 16 位。因此把立即数取出来，连接到一个符号扩展的部件上，对于这个小部件再增加一个小功能：左移 2 位，相当于乘以 4，然后再将其与 PC 加 4 的值加到一个新的加法器的输入端，输出端连接到多选器的 1 号通道上，如图 3-25 所示。现在这个 IFU 我们也补充完整了。分析完了指令系统中的最后一条指令了。知道了对于每一条指令每一个控制信号应该赋予什么样的值，但是这些信号的值是如何产生的呢？下节将探讨这个问题。

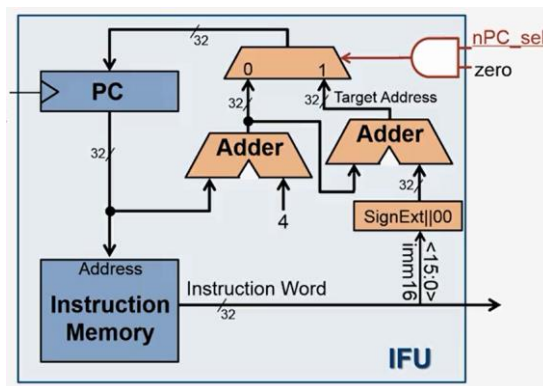


图 3-25 更新后的 IFU

### 3.4 控制信号的集成

自己设计出一台处理器是一个非常有意义的工作，现在我们需要把控制信号集成起来，能够让这个处理器自动的工作。分析完简化指令系统中所有的指令和控制信号，现在考虑如何集成这些控制信号，形成完整的控制逻辑。

首先把数据通路中的实现细节隐藏起来，用一个方框来表示，控制这个通路正常运转的控制信号一共有 8 个，这些控制信号是怎么产生的呢，还是要从来自指令存储器的指令编码开始分析。之前从指令编码中提取了若干的信号，作为数据通路的输入，但是有两个位域没有提及。Opcode 和 func 这两个位域是用来表明指令执行什么样的操作，因此需要用这两个信号来产生下面的控制信号，实现这部分功能的电路被称为控制逻辑，也就是控制器，如图 3-26。

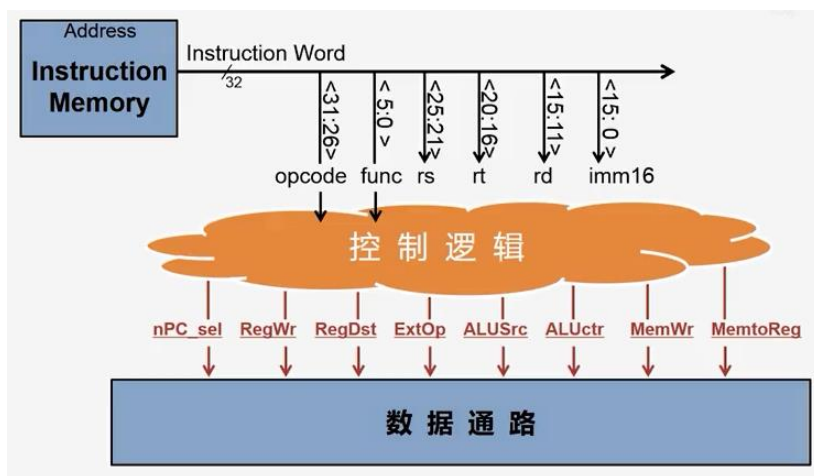


图 3-26 控制逻辑示意图

要想知道控制逻辑是如何实现的，先来看一个例子：以 add 指令为例，取出 add 指令后，所需执行的操作  $R[rd] \leftarrow R[rs] + R[rt]$ ;  $PC \leftarrow PC + 4$ ；对于这条指令所需的控制信号的值我们已经详细分析过，例如 RegDst 信号应该为 1，RegWr 信号也应该为 1，MemtoReg 信号应该为 0，把这些信号的值都摘出来汇总成一张表格，同样方法，我们可以列出其他指令对应的控制信号的值，如图 3-27 所示。

func	100000	100010	/			
opcode (op)	000000	000000	001101	100011	101011	000100
	add	sub	ori	lw	sw	beq
RegDst	1	1	0	0	x	x
ALUSrc	0	0	1	1	1	0
MemtoReg	0	0	0	1	x	x
RegWr	1	1	1	1	0	0
MemWr	0	0	0	0	1	0
nPC_sel	0	0	0	0	0	1
ExtOp	x	x	0	1	1	x
ALUctr<1:0>	00 (ADD)	01 (SUB)	10 (OR)	00 (ADD)	00 (ADD)	01 (SUB)

图 3-27 六条指令的控制信号

如果按行来看，可以得到每个控制信号的逻辑表达式，以第一行为例，只有当前指令为加法或减法指令时，该控制信号才需要为 1，其它情况都为 0 就可以了。写出 RegDst 的逻辑表达式，如果有一个信号标明是加法信号记为 add，另一个信号标明是减法指令记为 sub，这个 RegDst 控制信号等于 add 和 sub 信号进行或操作。下面进一步分析 add 和 sub 信号如

何生成，先看 add 指令的编码，opcode 域为 000000，func 域为 100000，这样的信号组合代表 add 指令。我们用 rtype 代表是 R 型指令， $\text{add} = \text{rtype} \cdot \text{func5} \cdot \sim \text{func4} \cdot \sim \text{func3} \cdot \sim \text{func2} \cdot \sim \text{func1} \cdot \sim \text{func0}$ ， $\text{sub} = \text{rtype} \cdot \text{func5} \cdot \sim \text{func4} \cdot \sim \text{func3} \cdot \sim \text{func2} \cdot \text{func1} \cdot \sim \text{func0}$ 。rtype 信号如何产生呢？因为 opcode 全为 0， $\text{rtype} = \sim \text{op5} \cdot \sim \text{op4} \cdot \sim \text{op3} \cdot \sim \text{op2} \cdot \sim \text{op1} \cdot \sim \text{op0}$ ，这个 rtype 信号就为 1，这样 RegDst 信号的产生表达式就有了，可以用与非门画出电路图，对于这个控制信号的控制逻辑就是确定的了。用同样的方法还可以得到其它信号的逻辑表达式，如图 3-28 所示。

```

RegDst    = add + sub
ALUSrc    = ori + lw + sw
MemtoReg  = lw
RegWr     = add + sub + ori + lw
MemWr     = sw
nPC_sel   = beq
ExtOp     = lw + sw
ALUctr[0] = sub + beq
ALUctr[1] = or

add  = rtype · func5 · ~func4 · ~func3 · ~func2 · ~func1 · ~func0
sub  = rtype · func5 · ~func4 · ~func3 · ~func2 · func1 · ~func0
rtype = ~op5 · ~op4 · ~op3 · ~op2 · ~op1 · ~op0,
ori  = ~op5 · ~op4 · op3 · op2 · ~op1 · op0
lw   = op5 · ~op4 · ~op3 · ~op2 · op1 · op0
sw   = op5 · ~op4 · op3 · ~op2 · op1 · op0
beq  = ~op5 · ~op4 · ~op3 · op2 · ~op1 · ~op0
    
```

图 3-28 控制信号的逻辑表达式

我们画出电路图：输入为 opcode 和 func 域，接到由与门组成的逻辑电路，产生一组中间信号，再接到由或门组成的逻辑电路，最终得到了想要的控制信号。这些电路就是控制逻辑，如图 3-29 所示。

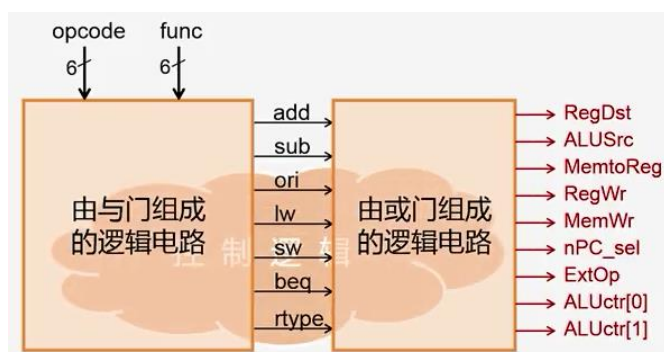


图 3-29 控制器的实现示意图

现在我们完成了控制器的实现。在这样的电路实现下，将指令编码的高六位和低六位连接到控制逻辑的输入端，输出端就会输出控制信号。现在完成了一个处理器结构设计的步骤。虽然这个处理器是单周期的，而且只支持 6 条指令。但是它确实是一个可以工作的处理器了。

单周期 CPU 是其它 CPU 的基础形态。它用最少的器件单元勾画了一个 CPU 的组成。以实现指令的功能为第一要务。在结构上，采用了控制单元对所有控制信号作统一的集中控制。使得所有的控制信号与当前指令的要求相一致。结构上大多采用组合逻辑，使得可以在一个周期内没有时钟上升沿的情况下可以完成功能。又由于 Register 和 DataMemory 两个单元需要有寄存器或 RAM 的写操作，因此只凭靠组合逻辑并不能完成，这里利用时钟下降沿作为这两个部件的时钟信号，以便于写入正确的数据。由于写入数据往往是最后一步操作，因此，一个占空比不对等的输入时钟也许更加有效率。

### 拓展练习：

根据本章所讲述的步骤，参照 1.2 节介绍的 QuartusII 软件的使用，实现支持至少上述六条指令的单周期 MIPS 处理器。（可用原理图或硬件描述语言方式实现）



## 第4章 硬件描述语言实现教学用 RISC CPU 设计

本章将介绍一个经过简化的用于教学目的的精简指令集（RISC）CPU 的构造原理和设计方法。相信您可以通过参考书上的程序和解释，经过自己的努力，可以独立完成该 CPU 核的设计和验证，以此来学习这种新设计方法，并掌握利用 Verilog 硬件描述语言的高层次设计方法。同时，在此基础上也可以设计自己的指令集，完成模型机设计实现。

### 4.1 设计背景介绍

在设计中我们不但关心 CPU 总体设计的合理性，而且还使得构成这个 RISC\_CPU 的每一个模块不仅是可仿真的也都可以综合成门级网表，因而从物理意义上说，这也是一个能真正通过具体电路结构而实现的 CPU。为了能在这个虚拟的 CPU 上运行较为复杂的程序并进行仿真，把寻址空间规定为 8K（即 13 位地址线）字节。

下面让我们一步一步地来设计这样一个 CPU，并进行 RTL 仿真和综合。经过综合、布局布线后，再进行一次仿真从中可以体会到这种设计方法的潜力。本章中的 VerilogHDL 程序都是为教学目的而编写的，全部程序在 QuartusII8.0 和 ModelSim SE 10.2c 工具下仿真验证，针对不同的 FPGA 进行了综合。顺利地通过了 RTL 级仿真、综合后门级逻辑网表仿真以及布线后的门级结构电路模型仿真。这个 CPU 模型只是一个教学模型，设计也不一定很合理，只是从原理上说明了简单的 RISC\_CPU 是如何构成的。本章内容是想达到以下五个目的：

- (1) 学习 RISC\_CPU 的基本结构和原理。
- (2) 了解 VerilogHDL 仿真和综合工具的潜力。
- (3) 展示 Verilog 设计方法对软/硬件联合设计和验证的意义。
- (4) 学习并掌握一些常用的 Verilog 语法和验证方法。
- (5) 能够基于全新指令集设计出自己的模型机。

### 4.2 什么是 CPU

CPU 是中央处理单元的英文缩写，它是计算机的核心部件。计算机进行信息处理可分为两个步骤：

- (1) 将数据和程序（即指令序列）输入到计算机的存储器中。
- (2) 从第一条指令的地址起开始执行该程序，得到所需结果，结束允许。CPU 的作用是协调并控制计算机的各个部件并执行程序的指令序列，使其有条不紊地进行，因此它必须具有以下基本功能：

① 取指令—当程序已在存储器中时，首先根据程序入口地址取出一条程序，为此要发出指令地址及控制信号。

② 分析指令—即指令译码，这是对当前取得的指令进行分析，指出它要求什么操作，并产生相应的操作控制命令。

③ 执行指令—根据分析指令时产生的“操作命令”形成相应的操作控制信号序列，通过运算器、存储器及输入输出设备的执行，实现每条指令的功能，其中包括对运算结果的处理以及下条指令地址的形成。

将 CPU 的功能进一步细化，可概括如下：

- (1) 能对指令进行译码并执行规定的动作；

- (2) 可以进行算术和逻辑运算；
- (3) 能与存储器和外设交换数据；
- (4) 提供整个系统所需要的控制。

尽管各种 CPU 的性能指标和结构细节各不相同，但它们所能完成的基本功能相同。有功能分析，可知任何一种 CPU 内部结构至少应包含下面这些部件：

- (1) 算术逻辑运算部件 (ALU)；
- (2) 累加器；
- (3) 程序计数器；
- (4) 指令寄存器和译码器；
- (5) 时序和控制部件。

RISC 即精简指令集计算机 (Reduced Instruction Set Computer) 的缩写。它是一种 20 世纪 80 年代才出现的 CPU，与一般的 CPU 相比不仅只是简化了指令系统，而且还通过简化指令系统使计算机的结构更加简单合理，从而提高了运算速度。从实现的途径看，RISC\_CPU 与一般 CPU 的不同之处在于：它的时序控制信号形成部件是用硬布线逻辑实现的而不是采用微程序控制的方式。所谓硬布线逻辑也就是用触发器和逻辑门直接连线所构成的状态机和组合逻辑，故产生控制序列的速度比用微程序快的多，因为这样做省去了读取微指令的时间。RISC\_CPU 也包括上述这些部件。下面就详细介绍一个简化的、用于教学目的 RISC\_CPU 的、可综合 Verilog HDL 模型的设计和仿真过程。

## 4.3 RISC\_CPU 寻址方式和指令系统

RISC\_CPU 的指令格式一律为：

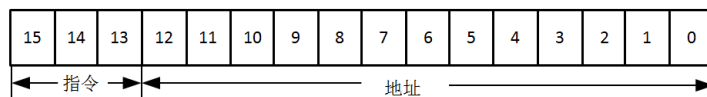


图 4-1 RISC\_CPU 指令格式

由于高三位代表指令，因此它的指令系统只能最多由 8 条指令组成。

- (1) HLT：停机操作。该操作将空一个指令周期，即 8 个时钟周期。
- (2) SKZ：为零跳过下一条语句。该操作先判断当前 alu 中的结果是否为零，若是零就跳过下一条语句，否则继续执行。
- (3) ADD 相加：该操作将累加器中的值与地址所指的存储器或端口的数据相加，结果仍送回累加器中。
- (4) AND 相与：该操作将累加器中的值与地址所指的存储器或端口的数据相与，结果仍送回累加器中。
- (5) XOR 异或：该操作将累加器的值与指令中给出地址的数据异或，结果仍送回累加器中。
- (6) LDA 读数据：该操作将指令中给出地址的数据放入累加器。
- (7) STO 写数据：该操作将累加器的数据放入指令中给出的地址。
- (8) JMP 无条件跳转语句：该操作将跳转至指令给出的目的地址，继续执行。

RISC\_CPU 是 8 位微处理器，一律采用直接寻址方式，即数据总是放在存储器中，寻址单元的地址由指令直接给出，这是最简单的寻址方式。



## 4.4 RISC\_CPU 结构

RISC\_CPU 是一个复杂的数字逻辑电路，但它的基本部件的逻辑并不复杂，可把它分成 8 个基本部件来考虑：

- (1) 时钟发生器；
- (2) 指令寄存器；
- (3) 累加器；
- (4) 算术逻辑运算单元；
- (5) 数据控制器；
- (6) 状态控制器；
- (7) 程序计数器；
- (8) 地址多路器。

各部件的相互连接关系即数据通路图见图 4-2。其中时钟发生器利用外来时钟信号进行分频生成一系列时钟信号，送往其他部件用作时钟信号，各部件之间的相互操作关系则由状态控制器来控制，各部件的具体结构和逻辑关系在下面各节中逐一进行介绍。

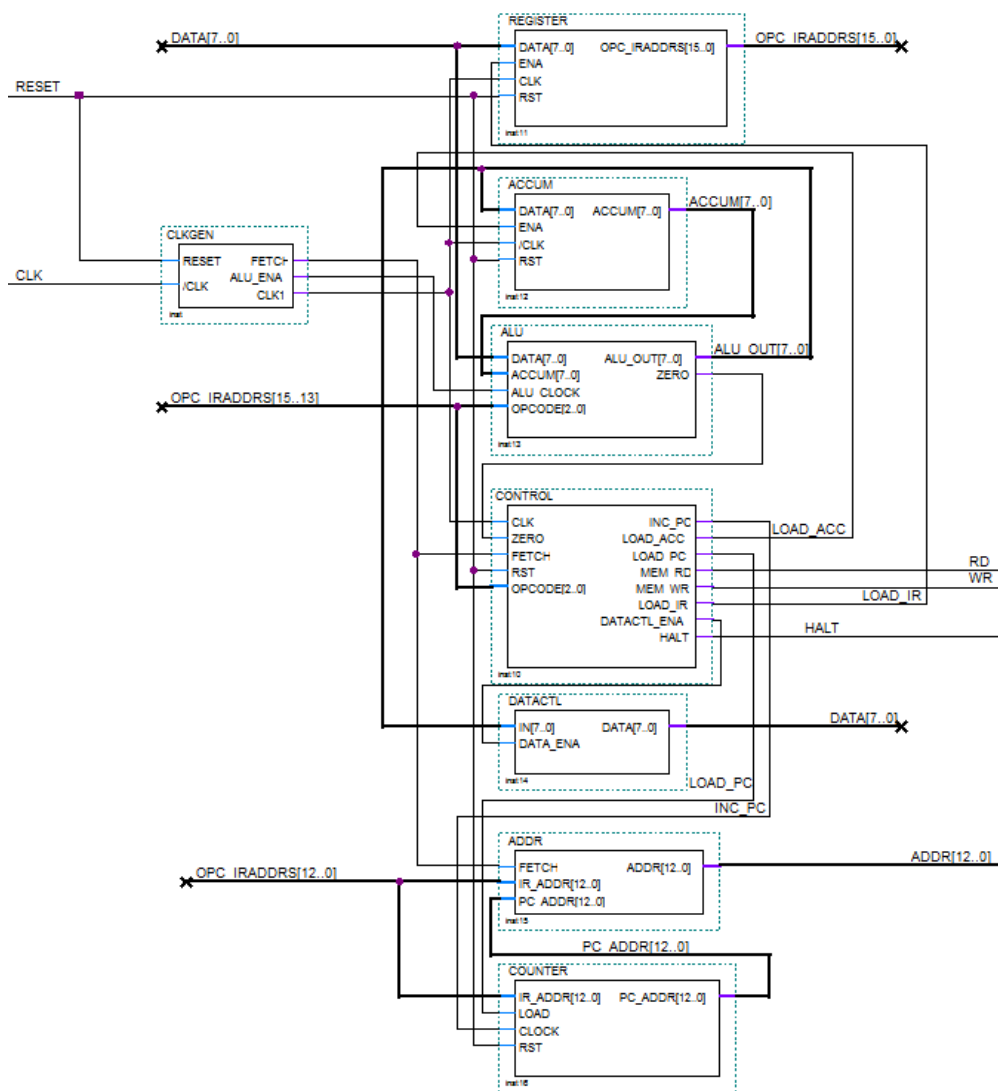


图 4-2 RISC\_CPU 中各部件的相互连接关系

### 4.4.1 时钟发生器

时钟发生器 CLKGEN 利用外来时钟信号 CLK 生成一系列时钟信号 CLK1、FETCH、ALU\_ENA，并送往 CPU 的其他部件。其中，FETCH 是控制信号，CLK 的 8 分频信号。当 FETCH 为高电平时，使 CLK 能触发 CPU 控制器开始执行一条指令；同时 FETCH 信号还将控制地址多路器输出指令地址和数据地址。CLK1 信号用作指令寄存器、累加器、状态控制器的时钟信号。ALU\_ENA 则用于控制算术逻辑运算单元的操作。图 4-3 为时钟发生器原理图。时钟发生器 CLKGEN 的波形如图 4-4 所示。

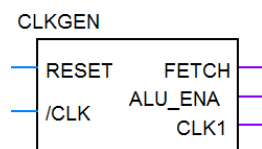


图 4-3 时钟发生器

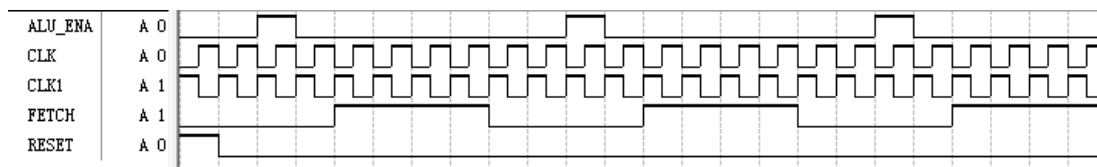


图 4-4 时钟发生器 CLKGEN 的波形

其 VerilogHDL 程序见下面的模块：

```
//-----clk_gen.v 的开始-----
`timescale 1ns/1ns
module clk_gen(clk, reset, fetch, alu_ena, clk1);
input clk, reset;
output fetch, alu_ena, clk1;
wire clk, reset;
reg fetch, alu_ena;
reg[7:0] state;
parameter S1 = 8'b00000001,
           S2 = 8'b00000010,
           S3 = 8'b00000100,
           S4 = 8'b00001000,
           S5 = 8'b00010000,
           S6 = 8'b00100000,
           S7 = 8'b01000000,
           S8 = 8'b10000000,
           idle = 8'b00000000;
assign clk1 = ~clk;
always @(negedge clk)
if(reset)
begin
    fetch <= 0;
    alu_ena <= 0;
    state <= idle;
end
```

```

    end
else
    begin
        case(state)
            S1:
                begin
                    alu_ena <= ~alu_ena;
                    state <= S2;
                end
            S2:
                begin
                    alu_ena <= ~alu_ena;
                    state <= S3;
                end
            S3:
                begin
                    fetch <= 1;
                    state <= S4;
                end
            S4:
                begin
                    state <= S5;
                end
            S5:
                state <= S6;
            S6:
                state <= S7;
            S7:
                begin
                    fetch <= 0;
                    state <= S8;
                end
            S8:
                begin
                    state <= S1;
                end
            idle:
                state <= S1;
            default:
                state <= idle;
        endcase
    end
endmodule
//-----clk_gen.v 的结束-----

```

由于在时钟发生器的设计中采用了同步状态机的设计方法，不但使 `clk_gen` 模块的源程序可以被各种综合器综合，也使得由其生成的 `fetch`, `alu_ena` 在同步性能上有明显的提高，为整个系统的性能提高打下了良好的基础。

## 4.4.2 指令寄存器

顾名思义，指令寄存器用于寄存指令，如图 4-5 所示。

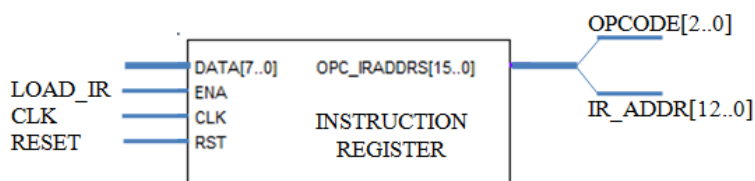


图 4-5 指令寄存器结构

指令寄存器的触发时钟是 `clk1`，在 `clk1` 的正沿触发下，寄存器将数据总线送来的指令存入高 8 位或低 8 位寄存器中。但并不是每个 `clk1` 的上升沿都寄存数据总线的的数据，因为数据总线上有时传送指令，有时传送数据。什么时候寄存，什么时候不寄存由 CPU 状态控制器的 `load_ir` 信号控制。`load_ir` 信号通过 `ena` 口输入到指令寄存器，复位后，指令寄存器被清为零。

每条指令为两个字节，即 16 位。高 3 位是操作码，低 13 位是地址（CPU 的地址总线为 13 位，寻址空间为 8K 字节）。本设计的数据总线为 8 位，所以每条指令需取两次。先取高 8 位，后取低 8 位。而当前取的是高 8 位还是低 8 位，由变量 `state` 记录。`state` 为 0 表示取的是高 8 位，存入高 8 位寄存器，同时将变量置为 1。下次再寄存时，由于 `state` 为 1，可知取的是低 8 位，存入低 8 位寄存器中。

其 VerilogHDL 程序见下面的模块：

```
//-----register.v 的开始-----
`timescale 1ns/1ns
module register(opc_iraddr, data, ena, clk1, rst);
output[15:0] opc_iraddr;
input[7:0] data;
input ena, clk1, rst;
reg[15:0] opc_iraddr;
reg state;
always @(posedge clk1)
begin
    if(rst)
    begin
        opc_iraddr <= 16'b0000_0000_0000_0000;
        state <= 1'b0;
    end
    else
    begin
        if(ena) // 如果加载指令寄存器信号 load_ir 到来，
        begin // 分两个时钟，每次 8 位加载指令寄存器
```

```

        casex(state) //先高字节, 后低字节
            1'b0: begin
                opc_iraddr[15:8] <= data;
                state <= 1;
            end
            1'b1:begin
                opc_iraddr[7:0] <= data;
                state <= 0;
            end
            default: begin
                opc_iraddr[15:0] <= 16'bxxxxxxxxxxxxxxxx;
                state <= 1'bx;
            end
        endcase
    end
else
    state <= 1'b0;
end
end
endmodule
//-----register.v 的结束-----
    
```

### 4.4.3 累加器

累加器用于存放当前的结果，它也是双目运算中的一个数据来源（见图 4-6）。复位后，累加器的值是零。当累加器通过 `ena` 口收到来自 CPU 状态控制器 `load_acc` 信号时，在 `clk1` 时钟负跳沿时就收到来自于数据总线的数据。

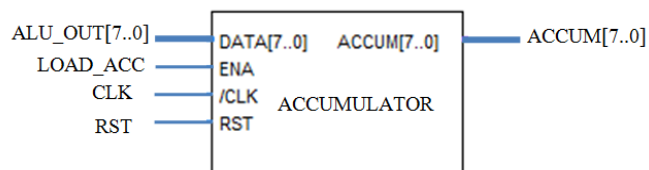


图 4-6 累加器结构

其 VerilogHDL 程序见下面的模块：

```

//-----accum.v 的开始-----
module accum(accum, data, ena, clk1, rst);
    output[7:0] accum;
    input[7:0] data;
    input ena, clk1, rst;
    reg[7:0] accum;
    always @(negedge clk1)
        begin
            if(rst)
                accum <= 8'b0000_0000; //Reset
        end
endmodule
    
```

```

        else
            if(ena)                // CPU 状态控制器发出 load_acc 信号
                accum <= data;    // Accumulate
            end
        endmodule
//-----accum.v 的结束-----
    
```

#### 4.4.4 算术运算器

算术逻辑运算单元如图 4-7 所示。它根据输入的 8 种不同操作码分别实现相应的加、与、异或、跳转等基本操作运算。利用这几种基本运算可以实现很多种其他运算以及逻辑判断等操作。

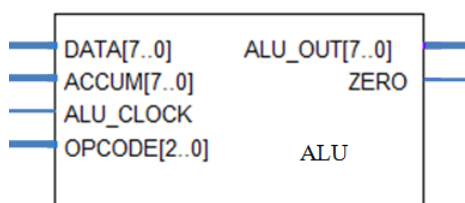


图 4-7 算术运算器结构

其 VerilogHDL 程序见下面的模块：

```

//-----alu.v 的开始-----
`timescale 1ns/1ns
module alu(alu_out, zero, data, accum, alu_ena, opcode);
    output[7:0] alu_out;
    output zero;
    input[7:0] data, accum;
    input[2:0] opcode;
    input alu_ena;
    reg[7:0] alu_out;
    parameter HLT = 3'b000,
              SKZ = 3'b001,
              ADD = 3'b010,
              ANDD = 3'b011,
              XORR = 3'b100,
              LDA = 3'b101,
              STO = 3'b110,
              JMP = 3'b111;
    assign zero = !accum;
    always @(posedge alu_ena)
        begin
            casex(opcode)
                HLT: alu_out <= accum;
                SKZ: alu_out <= accum;
                ADD: alu_out <= data + accum;
            endcase
        end
    
```

```

        ANDD:alu_out <= data & accum;
        XORR:alu_out <= data^accum;
        LDA: alu_out <= data;
        STO: alu_out <= accum;
        JMP: alu_out <= accum;
        default: alu_out <= 8'bxxxx_xxxx;
    endcase
end
endmodule
//----- alu.v 的结束-----

```

#### 4.4.5 数据控制器

数据控制器如图 4-8 所示。其作用是控制累加器的数据输出，由于数据总线是各种操作时传送数据的公共通道，不同情况下传送不同的内容。有时要传送指令，有时要传送 RAM 区或接口的数据。累加器的数据只有在需要往 RAM 区或端口写时才允许输出，否则应呈现高阻态，以允许其他部件使用数据总线。所以任何部件往总线上输出数据时，都需要一控制信号。而此控制信号的启、停则由 CPU 状态控制器输出的各信号控制决定。数据控制器何时输出累加器的数据则由状态控制器输出的控制信号 `datactl_ena` 决定。

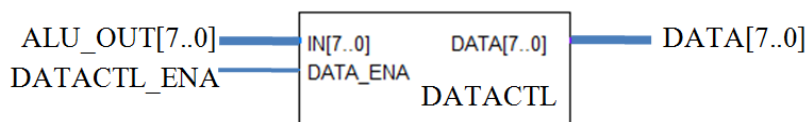


图 4-8 数据控制器结构

其 VerilogHDL 程序见下面的模块：

```

//-----datactl.v 的开始-----
module datactl(data, in, data_ena);
    output [7:0] data;
    input [7:0] in;
    input data_ena;
    assign data = (data_ena)?in:8'bzzzz_zzzz;
endmodule
//-----datactl.v 的结束-----

```

#### 4.4.6 地址多路器

地址多路器如图 4-9 所示。它用于选择输出的地址是 PC（程序计数器）地址还是数据/端口地址。每个指令周期的前 4 个时钟周期用于从 ROM 中读取指令，输出的应是 PC 地址；后 4 个时钟周期用于对 RAM 或端口的读写，该地址由指令给出。地址的选择输出信号由时钟信号的 8 分频信号 `FETCH` 提供。

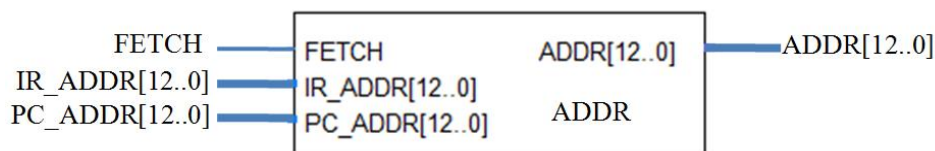


图 4-9 地址多路器结构

其 VerilogHDL 程序见下面的模块：

```
//-----adr.v 的开始-----
module adr(addr, fetch, ir_addr, pc_addr);
output[12:0] addr;
input [12:0] ir_addr, pc_addr;
input fetch;
assign addr = fetch? pc_addr:ir_addr;
endmodule
//-----adr.v 的结束-----
```

#### 4.4.7 程序计数器

程序计数器如图 4-10 所示。它用于提供指令地址，以便读取指令。指令按地址顺序存放在存储器中。有两种途径可形成指令地址；其一是顺序执行的情况，其二是遇到要改变顺序执行程序的情况，例如执行 JMP 指令后，需要形成新的指令地址。下面就来详细说明 PC 地址是如何建立的。

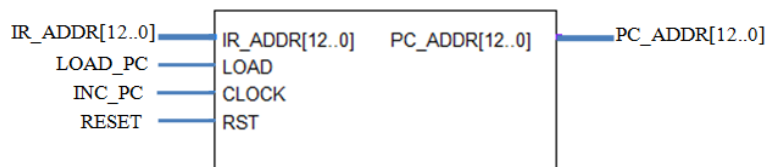


图 4-10 程序计数器结构

复位后，指令指针为零，即每次 CPU 重新启动将从 ROM 的零地址开始读取指令并执行。每条指令执行完需两个时钟，这时 pc\_addr 已被增 2，指向下一条指令（因为每条指令占两个字节）。如果正在执行的指令是跳转语句，这时 CPU 状态控制器将会输出 load\_pc 信号，通过 load 口进入程序计数器。程序计数器（pc\_addr）将装入目标地址（ir\_addr），而不是增 2。

其 VerilogHDL 程序见下面的模块：

```
//-----counter.v 的开始-----
module counter(pc_addr, ir_addr, load, clock, rst);
output[12:0] pc_addr;
input[12:0] ir_addr;
input load, clock, rst;
reg[12:0] pc_addr;
always@(posedge clock or posedge rst)
begin
    if(rst)
        pc_addr <= 13'b0_0000_0000_0000;
end
```



```

    else
        if(load)
            pc_addr <= ir_addr;
        else
            pc_addr <= pc_addr + 1;
        end
    end
endmodule
//-----counter.v 的结束-----

```

#### 4.4.8 状态控制器

状态控制器如图 4-11 所示。它由两部分组成：

- (1) 状态机（图中 machine 部分）；
- (2) 状态机控制器（图中 machinectl 部分）。

状态机控制器接收复位信号 RST，当 RST 有效时，通过信号 ena 使其为 0，输入到状态机中，以停止状态机的工作。

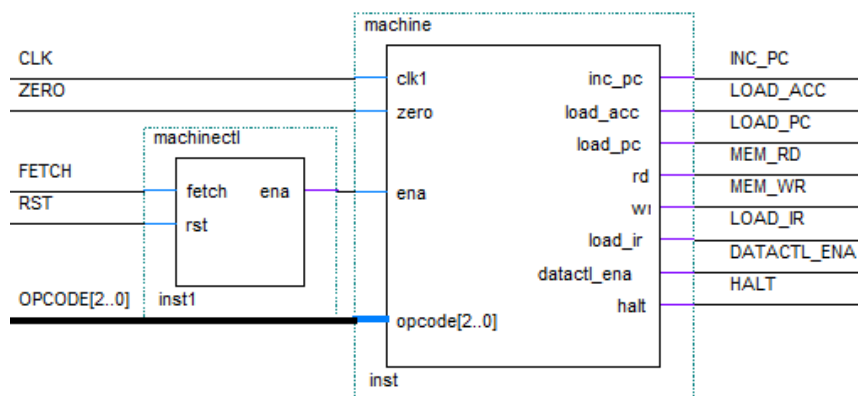


图 4-11 状态控制器

状态机控制器的 VerilogHDL 程序见下面模块：

```

//-----machinectl.v 的开始-----
`timescale 1ns/1ns
module machinectl(ena, fetch, rst);
    input fetch, rst;
    output ena;
    reg ena;
    reg state;
    always@(posedge fetch or posedge rst)
        begin
            if(rst)
                ena <= 0;
            else
                ena <= 1;
            end
        end
endmodule
//-----machinectl.v 的结束-----

```

状态机是 CPU 的控制核心，用于产生一系列的控制信号，启动或停止某些部件。CPU 何时进行读指令来读写 I/O 端口及 RAM 区等操作，都是由状态机来控制的。状态机的当前状态，由变量 state 记录，state 的值就是当前这个指令周期中已经过的时钟数（从零计起）。指令周期是由 8 个时钟周期组成，每个时钟周期都要完成固定的操作，即

(1) 第 0 个时钟，CPU 状态控制器的输出 rd 和 load\_ir 为高电平，其余均为低电平。指令寄存器由 ROM 送来的高 8 位指令代码。

(2) 第 1 个时钟，与上一时钟相比只是 inc\_pc 从 0 变为 1，故 PC 增 1，ROM 送来低 8 位指令代码，指令寄存器寄存该 8 位指令代码。

(3) 第 2 个时钟，空操作。

(4) 第 3 个时钟，PC 增 1，指向下一条指令。若操作符为 HLT，则输出信号 HLT 为高。如果操作符不为 HLT，除了 PC 增 1 外（指向下一条指令），其他各控制线输出为零。

(5) 第 4 个时钟，若操作符为 AND，ADD，XOR 或 LDA，读相应地址的数据；若为 JMP，将目的地址送给程序计数器；若为 STO，输出累加器数据。

(6) 第 5 个时钟，若操作符为 ANDD，ADD 或 XORR，算术运算器就进行相应的运算；若为 LDA，就把数据通过算术运算器送给累加器；若为 SKZ，先判断累加器的值是否为 0，如果为 0，PC 就增 1，否则保持原值；若为 JMP，锁存目的地址；若为 STO，将数据写入地址处。

(7) 第 6 个时钟，空操作。

(8) 第 7 个时钟，若操作符为 SKZ 且累加器值为 0，则 PC 值再增 1，跳过一条指令，否则 PC 无变化。

状态机的 VerilogHDL 程序见下面模块：

```
//-----machine.v 的开始-----
`timescale 1ns/1ns
module machine(inc_pc, load_acc, load_pc, rd, wr, load_ir, datactl_ena, halt, clk1, zero, ena,
opcode);
output inc_pc, load_acc, load_pc, rd, wr, load_ir;
output datactl_ena, halt;
input clk1, zero, ena;
input[2:0] opcode;
reg inc_pc, load_acc, load_pc, rd, wr, load_ir;
reg datactl_ena, halt;
reg[2:0] state;
parameter HLT = 3'b000,
          SKZ = 3'b001,
          ADD = 3'b010,
          ANDD = 3'b011,
          XORR = 3'b100,
          LDA = 3'b101,
          STO = 3'b110,
          JMP = 3'b111;
always@(negedge clk1)
begin
    if(!ena)                //接收到复位信号 RST，进行复位操作
```

```

        begin
            state <= 3'b000;
            {inc_pc, load_acc, load_pc, rd} <= 4'b0000;
            {wr, load_ir, datactl_ena, halt} <= 4'b0000;
        end
    else
        ctl_cycle;
    end
//-----begin of task ctl_cycle-----
task ctl_cycle;
begin
    casex(state)
        3'b000:      //装载指令的高 8 位
            begin
                {inc_pc, load_acc, load_pc, rd} <= 4'b0001;
                {wr, load_ir, datactl_ena, halt} <= 4'b0100;
                state <= 3'b001;
            end
        3'b001:      //PC 增 1, 装载指令低 8 位
            begin
                {inc_pc, load_acc, load_pc, rd} <= 4'b1001;
                {wr, load_ir, datactl_ena, halt} <= 4'b0100;
                state <= 3'b010;
            end
        3'b010:      //idle
            begin
                {inc_pc, load_acc, load_pc, rd} <= 4'b0000;
                {wr, load_ir, datactl_ena, halt} <= 4'b0000;
                state <= 3'b011;
            end
        3'b011:      //下一条指令地址建立, 分析指令从这里开始
            begin
                if(opcode == HLT) //指令为暂停 HLT
                    begin
                        {inc_pc, load_acc, load_pc, rd} <= 4'b1000;
                        {wr, load_ir, datactl_ena, halt} <= 4'b0001;
                    end
                else
                    begin
                        {inc_pc, load_acc, load_pc, rd} <= 4'b1000;
                        {wr, load_ir, datactl_ena, halt} <= 4'b0000;
                    end
                state <= 3'b100;
            end
    endcase
end

```

```

3'b100:      //取操作数
begin
  if(opcode == JMP)
    begin
      {inc_pc, load_acc, load_pc, rd} <= 4'b0010;
      {wr, load_ir, datactl_ena, halt} <= 4'b0000;
    end
  else
    if(opcode == ADD || opcode == ANDD || opcode == XORR || opcode == LDA)
      begin
        {inc_pc, load_acc, load_pc, rd} <= 4'b0001;
        {wr, load_ir, datactl_ena, halt} <= 4'b0000;
      end
    else
      if(opcode == STO)
        begin
          {inc_pc, load_acc, load_pc, rd} <= 4'b0000;
          {wr, load_ir, datactl_ena, halt} <= 4'b0010;
        end
      else
        begin
          {inc_pc, load_acc, load_pc, rd} <= 4'b0000;
          {wr, load_ir, datactl_ena, halt} <= 4'b0000;
        end
      state <= 3'b101;
    end
3'b101:      //执行指令
begin
  if(opcode == ADD || opcode == ANDD || opcode == XORR || opcode == LDA)
    begin //过一个时钟后与累加器的内容进行运算
      {inc_pc, load_acc, load_pc, rd} <= 4'b0101;
      {wr, load_ir, datactl_ena, halt} <= 4'b0000;
    end
  else
    if(opcode == SKZ && zero == 1 )
      begin
        {inc_pc, load_acc, load_pc, rd} <= 4'b1000;
        {wr, load_ir, datactl_ena, halt} <= 4'b0000;
      end
    else
      if(opcode == JMP)
        begin
          {inc_pc, load_acc, load_pc, rd} <= 4'b1010;
          {wr, load_ir, datactl_ena, halt} <= 4'b0000;
        end
      end
end

```

```

        end
    else
        if(opcode == STO)
            begin    //过一个时钟后把 wr 变 1 就可写到 RAM 中
                {inc_pc, load_acc, load_pc, rd} <= 4'b0000;
                {wr, load_ir, datactl_ena, halt} <= 4'b1010;
            end
        else
            begin
                {inc_pc, load_acc, load_pc, rd} <= 4'b0000;
                {wr, load_ir, datactl_ena, halt} <= 4'b0000;
            end
        state <= 3'b110;
    end
3'b110:    //idle
    begin
        if(opcode == STO)
            begin
                {inc_pc, load_acc, load_pc, rd} <= 4'b0000;
                {wr, load_ir, datactl_ena, halt} <= 4'b0010;
            end
        else
            if(opcode == ADD || opcode == ANDD || opcode == XORR || opcode ==
LDA)
                begin
                    {inc_pc, load_acc, load_pc, rd} <= 4'b0001;
                    {wr, load_ir, datactl_ena, halt} <= 4'b0000;
                end
            else
                begin
                    {inc_pc, load_acc, load_pc, rd} <= 4'b0000;
                    {wr, load_ir, datactl_ena, halt} <= 4'b0000;
                end
            state <= 3'b111;
        end
3'b111:
    begin
        if(opcode == SKZ && zero == 1)
            begin
                {inc_pc, load_acc, load_pc, rd} <= 4'b1000;
                {wr, load_ir, datactl_ena, halt} <= 4'b0000;
            end
        else
            begin

```

```

        {inc_pc, load_acc, load_pc, rd} <= 4'b0000;
        {wr, load_ir, datactl_ena, halt} <= 4'b0000;
    end
    state <= 3'b000;
end
default:
    begin
        {inc_pc, load_acc, load_pc, rd} <= 4'b0000;
        {wr, load_ir, datactl_ena, halt} <= 4'b0000;
        state <= 3'b000;
    end
endcase
end
endtask
//-----end of task ctl_cycle-----
endmodule
//-----machine.v 的结束-----

```

状态机和状态机控制器组成了状态控制器，它们之间的连接关系很简单，见图 4-11。

#### 4.4.9 外围模块

为了对 RISC\_CPU 进行测试，需要有存储测试程序的 ROM 和装载数据的 RAM、地址译码器。下面来简单介绍一下：

##### 1. 地址译码器

```

module addr_decode(addr, rom_sel, ram_sel);
    output rom_sel, ram_sel;
    input[12:0] addr;
    reg rom_sel, ram_sel;
    always@(addr)
        begin
            casex(addr)
                13'b1_1xxx_xxxx_xxxx: {rom_sel, ram_sel} <= 2'b01;
                13'b0_xxxx_xxxx_xxxx: {rom_sel, ram_sel} <= 2'b10;
                13'b1_0xxx_xxxx_xxxx: {rom_sel, ram_sel} <= 2'b10;
                default: {rom_sel, ram_sel} <= 2'b00;
            endcase
        end
endmodule

```

地址译码器用于产生选通信号，选通 ROM 或 RAM。

1FFFH-----1800H RAM

17FFH-----0000H ROM

##### 2. RAM 和 ROM

```

module ram(data, addr, ena, read, write);

```

```

inout[7:0] data;
input[9:0] addr;
input ena;
input read, write;
reg [7:0] ram[10'h3ff:0];
assign data = (read && ena)? ram[addr]:8'hzz;
always@(posedge write)
    begin
        ram[addr] <= data;
    end
endmodule

```

```

module rom(data, addr, read, ena);
output[7:0] data;
input[12:0] addr;
input read, ena;
reg[7:0] memory[13'h1fff:0];
wire[7:0] data;
assign data = (read&& ena)? memory[addr]:8'bzzzzzzzz;
endmodule

```

ROM 用于装载测试程序，可读不可写；而 RAM 用于存放数据，可读可写。

## 4.5 RISC\_CPU 操作和时序

一个微机系统为了完成自身的功能，需要 CPU 执行许多操作。以下是 RISC\_CPU 的主要操作：

- (1) 系统的复位和启动操作；
- (2) 总线读操作；
- (3) 总线写操作。

下面详细介绍每个操作，即系统的复位与启动，总线的读与写操作。

### 4.5.1 系统的复位和启动操作

RISC\_CPU 的复位和启动操作是通过 reset 引脚的信号触发执行的。当 reset 信号一进入高电平，RISC\_CPU 就会结束现行操作，并且只要 reset 停留在高电平状态，CPU 就维持在复位状态。在复位状态，CPU 各内部寄存器都被设为初值，全部为零。数据总线为高阻态，地址总线为 0000H，所有控制信号均为无效状态。Reset 回到低电平后，接着到来的第一个 fetch 上升沿将启动 RISC\_CPU 开始工作，从 ROM 的 000 处开始读取指令并执行相应操作。波形如图 4-12RISC\_CPU 的复位和启动操作的波形图，虚线标志处为 RISC\_CPU 启动工作的时刻。

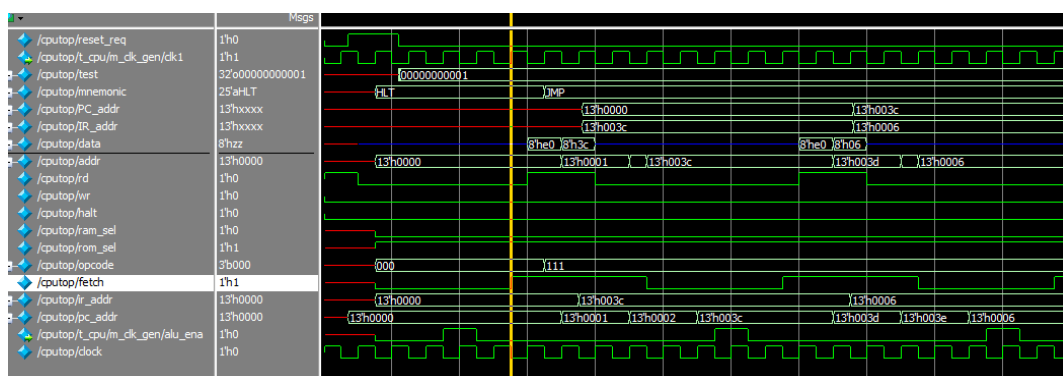


图 4-12 RISC CPU 的复位和启动操作波形

### 4.5.2 总线读操作

每个指令周期的前 0~2 个时钟周期用于读指令，在状态控制器一节中已详细讲述，这里就不再重复；第 3.5 个周期处，存储器或端口地址就输出到地址总线上；第 4~6 个时钟周期，读信号 rd 有效，数据送到数据总线上，以备累加器锁存，或参与算术、逻辑运算；第 7 个时钟周期，读信号无效，第 7.5 个时钟周期，地址总线输出 PC 地址，为下一个指令做好准备。图 4-13 为 CPU 从存储器或端口读取数据的时序。

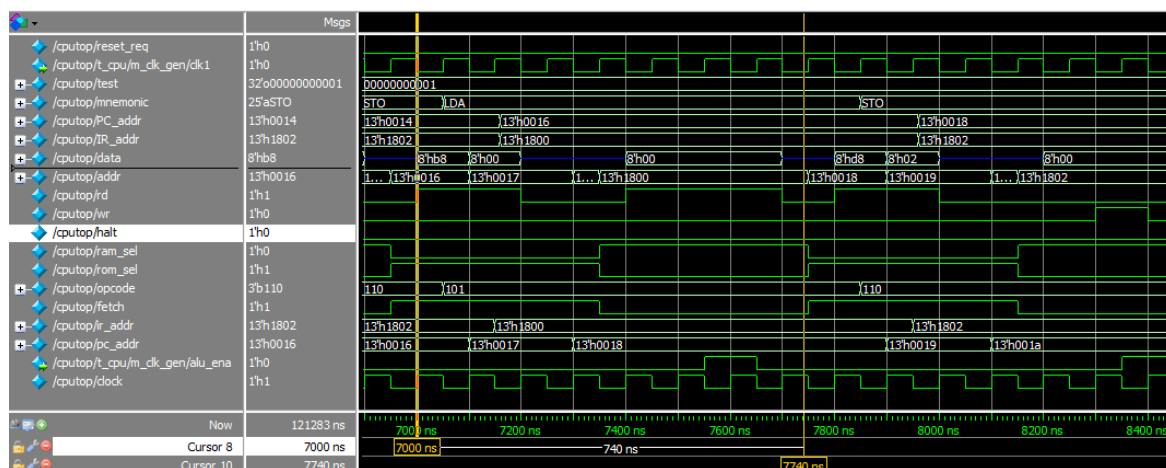


图 4-13 CPU 从存储器或端口读取数据的时序

### 4.5.3 总线写操作

每个指令周期的第 3.5 个时钟周期处，写的地址就建立了；第 4 个时钟周期输出数据；第 5 个时钟周期输出写信号；至第 6 个时钟结束，数据无效；第 7.5 个时钟地址输出为 PC 地址，为下一个指令周期做好准备。图 4-14 为 CPU 对存储器或端口写数据的时序。



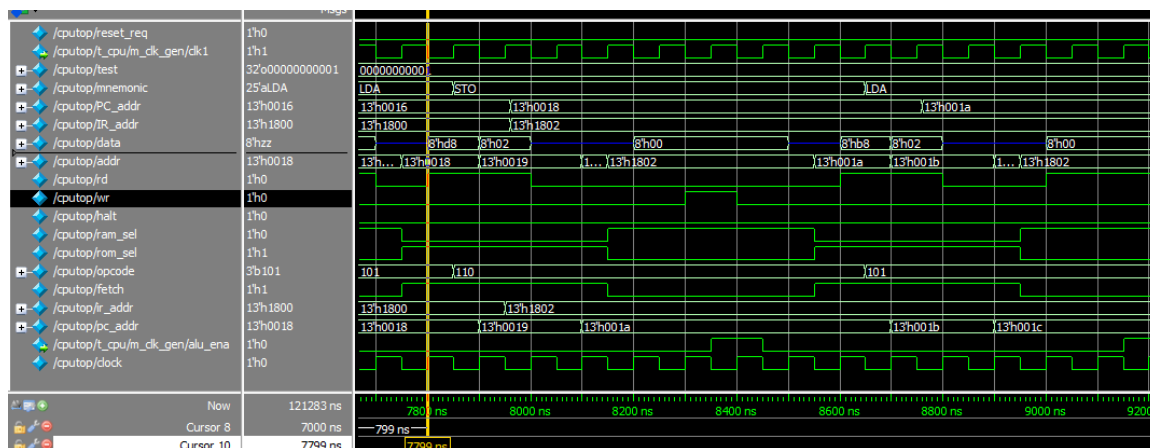


图 4-14 CPU 对存储器或端口写数据的时序

## 4.6 RISC\_CPU 模块的调试

### 4.6.1 RISC\_CPU 模块的前仿真

为了对所涉及的 RISC\_CPU 模块进行验证，需要把 RISC\_CPU 包装在一个模块下，这样其内部连线就隐藏起来，从系统的角度看就显得简洁，见图。。还需要建立一些必要的外围器件模型，例如储存程序用的 ROM 模型，储存数据用的 RAM 和地址译码器等。这些模型都可以用 VerilogHDL 描述。由于不需要综合成具体的电路，只要保证功能和接口信号正确就能用于仿真。也就是说，用虚拟器件来代替真实的器件对所设计的 RISC\_CPU 模块进行验证，检查各条指令是否执行正确，与外围电路的数据交换是否正常。这种模块是很容易编写的，上面一节中的 ROM 和 RAM 模块就是简化的虚拟器件的例子，可在下面的仿真中来代替真实的器件，用于验证 RISC\_CPU 模块是否能正确地运行装入 ROM 和 RAM 的程序。在 RISC\_CPU 的电路图上加上这些外围电路把有关的电路接通，如图 4-15 所示；也可以用 VerilogHDL 模块调用的方法把这些外围电路的模块连接上，这跟用真实的电路器件调试情况很类似。

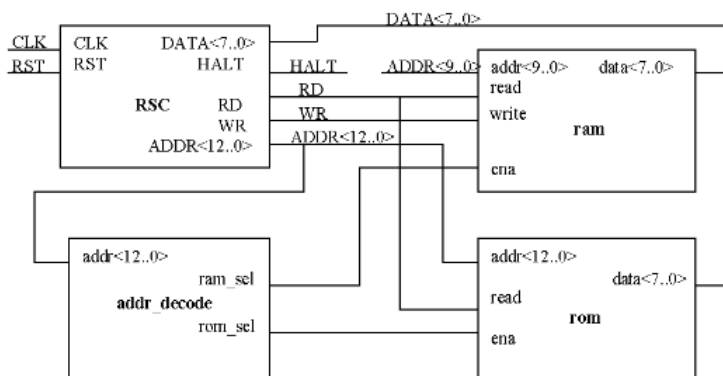


图 4-15RISC\_CPU 及其外围电路

下面介绍的是在 ModelSim SE10.2c 下进行调试的仿真测试程序 cputop.v。可用于对以上所设计的 RISC\_CPU 进行仿真测试。下面是前仿真的测试程序 cputop.v。它的作用是按模块的要求进行仿真，并显示仿真的结果，测试模块 cputop.v 中的 \$display 和 \$monitor 等系统调用能在计算机的显示屏上显示部分测试结果，可以同时用波形观察器观察有关信号的波

形。

```
//-----cputop.v 文件的开始-----
/*****
***模块功能: cputop 模块通过运行 3 个不同的汇编程序对 cpu 模块进行完整的逻辑测试
***           和验证。仿真程序在加载后会自动执行，运行的结果在人机交互式显示屏上
***           显示。本模块是不可综合的行为模块，用于对 cpu 模块进行 RTL 级和门级逻辑
***           功能的全面仿真验证。本测试模块也可以为布线后仿真，只需要将代码的
***           第一句`include “cpu.v”改写为`include “cpu.vo”即可。
*****/
`include “cpu.v” //若改为`include “cpu.vo”便可做布线后仿真
`include “ram.v”
`include “rom.v”
`include “addr_decode.v”
/*****
`timescale 1ns/1ns
`define PERIOD 100 //matches clk_gen.v
module cputop;
reg reset_req, clock;
integer test;
reg[(3*8):0] mnemonic; //array that holds 3 8bit ASCII characters
reg[12:0] PC_addr, IR_addr;
wire[7:0] data;
wire[12:0] addr;
wire rd, wr, halt, ram_sel, rom_sel;
wire[2:0] opcode; //为布线后仿真做的专门添加的 CPU 内部信号线
wire fetch; //为布线后仿真做的专门添加的 CPU 内部信号线
wire[12:0] ir_addr, pc_addr; //为布线后仿真做的专门添加的 CPU 内部信号线
//-----CPU 模块与地址译码器和 ROM、RAM 的连接部分-----
cpu t_cpu(.clk(clock), .reset(reset_req), .halt(halt), .rd(rd), .wr(wr), .addr(addr),
           .data(data), .opcode(opcode), .fetch(fetch), .ir_addr(ir_addr), .pc_addr(pc_addr));
ram t_ram(.addr(addr[9:0]), .read(rd), .write(wr), .ena(ram_sel), .data(data));
rom t_rom(.addr(addr), .read(rd), .ena(rom_sel), .data(data));
addr_decode t_addr_decode(.addr(addr), .ram_sel(ram_sel), .rom_sel(rom_sel));
//-----CPU 模块与地址译码器和 ROM、RAM 的连接部分结束-----
initial
begin
    clock = 1;
    $timeformat(-9, 1, "ns", 12);
    display_debug_message;
    sys_reset;
    test1;
//    $stop;
    test2;
//    $stop;
```

```

        test3;
        $finish;          //simulation is finished here.
    end

task display_debug_message;
    begin
        $display("\n*****");
        $display("*   THE FOLLOWING DEBUG TASK ARE AVAILABLE:   *");
        $display("*   \" test1; \" to load the 1st diagnostic program.   *");
        $display("*   \" test2; \" to load the 2nd diagnostic program.   *");
        $display("*   \" test3; \" to load the Fibonacci program.       *");
        $display("*****\n");
    end
endtask

task test1;
    begin
        test = 0;
        disable MONITOR;
        $readmemb("D:/RISC_CPU/cputop/test1.pro", t_rom.memory);
        $display("rom loaded successfully!");
        $readmemb("D:/RISC_CPU/cputop/test1.dat", t_ram.ram);
        $display("ram loaded successfully!");
        #1 test = 1;
        #14800;
        sys_reset;
    end
endtask

task test2;
    begin
        test = 0;
        disable MONITOR;
        $readmemb("D:/RISC_CPU/cputop/test2.pro", t_rom.memory);
        $display("rom loaded successfully!");
        $readmemb("D:/RISC_CPU/cputop/test2.dat", t_ram.ram);
        $display("ram loaded successfully!");
        #1 test = 2;
        #11600;
        sys_reset;
    end
endtask

task test3;

```

```

begin
    test = 0;
    disable MONITOR;
    $readmemb("D:/RISC_CPU/cputop/test3.pro", t_rom.memory);
    $display("rom loaded successfully!");
    $readmemb("D:/RISC_CPU/cputop/test3.dat", t_ram.ram);
    $display("ram loaded successfully!");
    #1 test = 3;
    #94000;
    sys_reset;
end
endtask

task sys_reset;
begin
    reset_req = 0;
    #(`PERIOD*0.7) reset_req = 1;
    #(1.5*`PERIOD) reset_req = 0;
end
endtask

always @(test)
begin:MONITOR
    case(test)
    1: begin //display results when running test 1
        $display("\n*** RUNNING CPU test1 --- The Basic CPU Diagnostic Program ***
\n");

        $display("\n      TIME      PC   INSTR  ADDR  DATA   ");
        $display("      -----  ---  ---  ---  -----");
        while(test == 1)
            @(t_cpu.pc_addr)
            if((t_cpu.pc_addr%2 == 1)&&(t_cpu.fetch == 1))
                begin
                    #60 PC_addr <= t_cpu.pc_addr - 1;
                    IR_addr <= t_cpu.ir_addr;
                    #340 $strobe("%t   %h   %s   %h   %h", $time, PC_addr,
mnemonic, IR_addr, data);
                end
            end
        2: begin
            $display("\n*** RUNNING CPU test2 --- The Advanced CPU Diagnostic Program
*** \n");

            $display("\n      TIME      PC   INSTR  ADDR  DATA   ");
            $display("      -----  ---  ---  ---  -----");

```

```

        while(test == 2 )
            @(t_cpu.pc_addr)
            if((t_cpu.pc_addr%2 == 1)&&(t_cpu.fetch == 1))
begin
    #60 PC_addr <= t_cpu.pc_addr - 1;
    IR_addr <= t_cpu.ir_addr;
    #340 $strobe("%t  %h  %s  %h  %h", $time, PC_addr, mnemonic, IR_addr,
data);
    end
end
3: begin
    $display("\n*** RUNNING CPU test3 --- An Executable Program *** \n");
    $display("\n* * * This program should calculate the fibonacci * * *");
    $display("  -----  -----");
    while(test == 3)
        begin
            wait(t_cpu.opcode == 3'h1)
            $strobe("%t      %d", $time, t_ram.ram[10'h2]);
            wait(t_cpu.opcode != 3'h1);
        end
    end
endcase
end

always @(posedge halt)      //STOP when HALT instruction decoded
begin
    #500
    $display("\n*****");
    $display("*** A HALT INSTRUCTION WAS PROCESSED !!! ***");
    $display("*****\n");
end

always #(`PERIOD/2) clock <= ~clock;
always @(t_cpu.opcode)//get an ASCII mnemonic for each opcode
case(t_cpu.opcode)
    3'b000: mnemonic = "HLT";
    3'b001: mnemonic = "SKZ";
    3'b010: mnemonic = "ADD";
    3'b011: mnemonic = "AND";
    3'b100: mnemonic = "XOR";
    3'b101: mnemonic = "LDA";
    3'b110: mnemonic = "STO";
    3'b111: mnemonic = "JMP";
    default: mnemonic = "???";

```

```

        endcase
    endmodule

//-----cputop.v 文件的结束-----
针对程序做如下说明：测试程序中用宏指令`include”模块文件”名包含了 rom.v, ram.v 和
addr_decode.v3 个外部模块。它们都是检测 RISC_CPU 时必不可少的虚拟部件，却代表了实
际硬件电路中的 RAM，ROM 和地址译码器；对于 RISC_CPU 需要综合成电路的部分，则
已通过 cpu.v 程序将它组合成一个独立的 CPU 模块。具体程序如下：
//-----cpu.v 文件的开始-----
/*****
模块功能：CPU 模块是所设计的 RISC_CPU 的核心，共有 10 个可综合模块的连接组成。它
本身是可综合的模块，已经过门级后仿真验证。
*****/

`include "clk_gen.v"
`include "accum.v"
`include "adr.v"
`include "alu.v"
`include "machine.v"
`include "counter.v"
`include "machinectl.v"
`include "register.v"
`include "datactl.v"
//*****

`timescale 1ns/1ns
module cpu(clk, reset, halt, rd, wr, addr, data, opcode, fetch, ir_addr, pc_addr);
input clk, reset;
output rd, wr, halt;
output[12:0] addr;
output[2:0] opcode;
output fetch;
output[12:0] ir_addr, pc_addr;
inout[7:0] data;
wire clk, reset, halt;
wire[7:0] data;
wire[12:0] addr;
wire rd, wr;
wire fetch, alu_ena;
wire[2:0] opcode;
wire[12:0] ir_addr, pc_addr;
wire[7:0] alu_out, accum;
wire zero, inc_pc, load_acc, load_pc, load_ir, data_ena, contr_ena;
wire clk1;

clk_gen m_clk_gen(.clk(clk), .reset(reset), .fetch(fetch), .alu_ena(alu_ena), .clk1(clk1));
register m_register(.data(data), .ena(load_ir), .rst(reset),

```

```

        .clk1(clk1), .opc_iraddr({opcode, ir_addr}));
accum m_accum(.data(alu_out), .ena(load_acc), .clk1(clk1), .rst(reset), .accum(accum));
alu m_alu(.data(data), .accum(accum), .alu_ena(alu_ena), .opcode(opcode),
        .alu_out(alu_out), .zero(zero));
machinectl m_machinectl(.rst(reset), .fetch(fetch), .ena(contr_ena));
machine m_machine(.inc_pc(inc_pc), .load_acc(load_acc), .load_pc(load_pc), .rd(rd), .wr(wr),
        .load_ir(load_ir), .clk1(clk1), .datactl_ena(data_ena), .halt(halt),
        .zero(zero), .ena(contr_ena), .opcode(opcode));
datactl m_datactl(.in(alu_out), .data_ena(data_ena), .data(data));
adr m_adr(.fetch(fetch), .ir_addr(ir_addr), .pc_addr(pc_addr), .addr(addr));
counter m_counter(.clock(inc_pc), .rst(reset), .ir_addr(ir_addr),
        .load(load_pc), .pc_addr(pc_addr));
    
```

endmodule

//-----cpu.v 文件的结束-----

其中, contr\_ena 用于 machinectl 与 machine 之间的 ena 的连接。cputop.v 中用到下面两条语句需要解释一下:

```
$readmemb("D:/RISC_CPU/cputop/test1.pro", t_rom.memory);
```

```
$readmemb("D:/RISC_CPU/cputop/test1.dat", t_ram.ram);
```

即可把编译好的汇编机器码装入虚拟 ROM, 把需要参加运算的数据装入虚拟 RAM 就可以开始仿真。上面语句中的第一项为打开的文件名, 必须指名所在路径, 后一项为系统层次管理下的 ROM 模块和 RAM 模块中的存储器 memory 和 ram。

下面所列出的是用于测试 RISC\_CPU 基本功能而分别装入虚拟 ROM 和 RAM 的机器码和数据文件, 其文件名分别为 test1.pro, test1.dat, test2.pro, test2.dat, test3.pro, test3.dat。

//-----文件 test1.pro-----

/\*\*\*\*\*\*

\*\*\* Test1 程序是用于验证 RISC\_CPU 逻辑功能的机器代码, 是根据汇编语言由人工编译的。

\*\*\* 本汇编程序用于测试 RISC\_CPU 的基本指令集, 如果 RISC\_CPU 的各条指令执行正确, 它\*\*\* 应在地址为 2E (hex) 处, 在执行 HLT 时停止运行。如果该程序在任何其它地址暂停运行\*\*\* 行, 则必有一条指令运行出错, 可参照注释找到出错的指令。

\*\*\* @符号后的十六进制数表示存储器的地址, 以下的二进制数为机器码。

\*\*\* 每行//符号后表示自己的 RISC\_CPU 设计的汇编程序和程序注释。

\*\*\*\*\*/

//----- test1.pro 的开始-----

机器码	地址	汇编助记符	注释
@00			
111_00000	//00	BEGIN: JMP TST_JMP	
0011_1100			
000_00000	//02	HLT	//JMP did not work at all
0000_0000			
000_00000	//04	HLT	//JMP did not load PC, is skipped
0000_0000			
101_11000	//06	JMP_OK: LDA DATA_1	
0000_0000			

```

001_00000 //08      SKZ
0000_0000
000_00000 //0a      HLT          //SKZ or LDA did not work
0000_0000
101_11000 //0c      LDA  DATA_2
0000_0001
001_00000 //0e      SKZ
0000_0000
111_00000 //10      JMP SKZ_OK
0001_0100
000_00000 //12      HLT
0000_0000
110_11000 //14 SKZ_OK:  STO TEMP
0000_0010
101_11000 //16      LDA DATA_1
0000_0000
110_11000 //18      STO TEMP
0000_0010
101_11000 //1a      LDA TEMP
0000_0010
001_00000 //1c      SKZ
0000_0000
000_00000 //1e      HLT
0000_0000
100_11000 //20      XOR DATA_2
0000_0001
001_00000 //22      SKZ
0000_0000
111_00000 //24      JMP XOR_OK
0010_1000
000_00000 //26      HLT
0000_0000
100_11000 //28 XOR_OK:  XOR DATA_2
0000_0001
001_00000 //2a      SKZ
0000_0000
000_00000 //2c      HLT
0000_0000
000_00000 //2e END:      HLT          //Congratulations ---TEST1 PASSED!
0000_0000
111_00000 //30      JMP BEGIN      //run test again
0000_0000

```

@3c



```

111_00000 //3c TST_JMP:    JMP JMP_OK
0000_0110
000_00000 //3e          HLT          //JMP is broken
//----- test1.pro 的结束-----
/*****

```

下面文件中的数据在仿真时需要用系统任务\$readmemb 读入 RAM，才能被上面的汇编程序 test1.pro 使用。

```

*****/
//----- test1.dat 的开始-----
@00          //address statement at RAM
00000000     //1800 DATA_1
11111111     //1801 DATA_2
10101010     //1802 TEMP
//----- test1.dat 的结束-----
/*****

```

```

** Test2 程序是用于验证 RISC_CPU 的功能，是设计工作的重要环节。
** 本程序测试 RISC_CPU 的高级指令集，如果 RISC_CPU 的各条指令执行正确，
** 它应在地址为 20（hex）处，在执行 HLT 时停止运行。
** 如果该程序在任何其他地址暂停运行，则必有一条指令运行出错。
** 可参照注释找到出错的指令。
** 注意：必须先在 RISC_CPU 上运行 test1 程序成功后，才可运行本程序。

```

```

*****/
//----- test2.pro 的开始-----

```

机器码	地 址	汇编助记符	注释
-----	-----	-------	----

@00			
101_11000	//00	BEGIN:	LDA DATA_2
0000_0001			
011_11000	//02		AND DATA_3
0000_0010			
100_11000	//04		XOR DATA_2
0000_0001			
001_00000	//06		SKZ
0000_0000			
000_00000	//08		HLT
0000_0000			
010_11000	//0a		ADD DATA_1
0000_0000			
001_00000	//0c		SKZ
0000_0000			
111_00000	//0e		JMP ADD_OK
0001_0010			
000_00000	//10		HLT
0000_0000			
100_11000	//12	ADD_OK:	XOR DATA_3

```

0000_0010
010_11000 //14      ADD DATA_1      // FF plus 1 makes -1
0000_0000
110_11000 //16      STO TEMP
0000_0011
101_11000 //18      LDA DATA_1
0000_0000
010_11000 //1a      ADD TEMP      //-1 plus 1 should make zero
0000_0011
001_00000 //1c      SKZ
0000_0000
000_00000 //1e      HLT
0000_0000
000_00000 //20      END: HLT          //Congratulations – TEST2 PASSED!
0000_0000
111_00000 //22      JMP BEGIN      //run test again
0000_0000
//----- test2.pro 的结束-----
/*****

    下面文件中的数据在仿真时需要用系统任务$readmemb 读入 RAM，才能被上面的汇编
    程序 test2.pro 使用。

*****/

//----- test2.dat 的开始-----
@00
00000001      //1800 DATA_1
10101010      //1801 DATA_2
11111111 //1802 DATA_3
00000000      //1803 TEMP
//----- test2.dat 的结束-----

/*****

** Test3 程序是一个计算从 0~144 的 Fibonacci 序列的程序，用于验证 RISC_CPU 的功能。
** 所谓 Fibonacci 序列就是一系列数，其中每一个数都是它前面两个数的和（如：0，1，1，
** 2，3，5，8，13，21，...）。这种序列常用于财务分析。
** 注意：必须在成功地运行前两个测试程序后才运行本程序，否则很难发现问题所在。
*****/

//----- test3.pro 的开始-----
机器码      地址          汇编助记符      注释
@00
101_11000 //00      LOOP:  LDA FN2          //load value in FN2 into accum
0000_0001
110_11000 //02          STO TEMP      //store accumulator in TEMP
0000_0010
010_11000 //04          ADD FN1          //add value in FN1 to accumulator

```

```

0000_0000
110_11000 //06          STO FN2      //store result in FN2
0000_0001
101_11000 //08          VLDA TEMP //load TEMP into the accumulator
0000_0010
110_11000 //0a          STO FN1      //store accumulator in FN1
0000_0000
100_11000 //0c          XOR LIMIT   //compare accumulator to LIMIT
0000_0011
001_00000 //0e          SKZ         //if accum = 0, skip to DONE
0000_0000
111_00000 //10          JMP LOOP    //jump to address of LOOP
0000_0000
000_00000 //12  DONE:  HLT         //end of program
0000_0000
//----- test3.pro 的结束-----
    
```

/\*\*\*\*\*

下面文件中的数据在仿真时需要用系统任务\$readmemb 读入 RAM，才能被上面的汇编程序 test3.pro 使用。

\*\*\*\*\*/

//----- test3.dat 的开始-----

```

@00
00000001 //1800 FN1;      //data storage for 1st Fib. No.
00000000 //1801 FN2;      //data storage for 2nd Fib. No.
00000000 //1802 TEMP;     //temporary data storage
10010000 //1803 LIMIT;    //max value to calculate 144(dec)
    
```


//----- test3.dat 的结束-----

下面介绍前仿真的步骤，首先按照表示各模块之间连线的电路图编制测试文件，即定义 Verilog 的 wire 变量作为连线，连接各功能模块之间的引脚，并将输入信号引入，输出信号引出。如若需要，可加入必要的语句显示提示信息。例如，RISC\_CPU 的测试文件就是 cputop.v。其次，使用仿真软件进行仿真，由于不同的软件使用方法可能有较大差异，以下只简单的介绍 ModelSim10.2 的使用。

(1) 建一个目录存放编写的设计代码文件，注意所有 Verilog 设计代名都必须用扩展名为.v 的文本型文件存档（注意：在计算机环境中将.v 扩展名文件的打开方式设置为 ModelSim）。

(2) 双击 cputop.v 便能自动地启动 modelsim10.2c 进入文件所在的目录。

(3) 单击 File->New->Library 弹出一个配置框，给新的 Library 命名，例如键入 work，单击 OK，在 Transcript 框内出现一些文字告诉操作者所在的目录和处理的文件和新建立的 Library 的名字，这次编译的很多信息将存储在这个由用户起名的 Library 中。

(4) 单击图标，弹出配置框，可设置 Library，并可以把需要编译的文件选中进行编译。

(5) 编译成功后，用户将在 workspace 的框内发现用户命名的 Library 下面出现了编译通过的文件名，如果设计仿真所需要的所有文件已经编译成功，就可以加载仿真代码。至需

要双击 cputop 即可，如果加载成功，ModelSim 自动进入可以仿真的状态，只要配置好需要观察波形的信号，就可以单击开始仿真的图标。由于 cputop.v 编写了很好的输出显示，所以在 Transcript 框内将显示以下信息，这些信息说明仿真工作正确无误。

仿真结果如下：

```
run- all
#
# *****
# *   THE FOLLOWING DEBUG TASK ARE AVAILABLE:           *
# *   " test1; " to load the 1st diagnostic program.      *
# *   " test2; " to load the 2nd diagnostic program.      *
# *   " test3; " to load the Fibonacci program.          *
# *****
#
# rom loaded successfully!
# ram loaded successfully!
#
# *** RUNNING CPU test1 --- The Basic CPU Diagnostic Program ***
#
#
#      TIME      PC      INSTR  ADDR  DATA
#      -----  ---  ---  ---  ---
#      1100.0ns  0000      JMP   003c   zz
#      1900.0ns  003c      JMP   0006   zz
#      2700.0ns  0006      LDA   1800   00
#      3500.0ns  0008      SKZ   0000   zz
#      4300.0ns  000c      LDA   1801   ff
#      5100.0ns  000e      SKZ   0000   zz
#      5900.0ns  0010      JMP   0014   zz
#      6700.0ns  0014      STO   1802   ff
#      7500.0ns  0016      LDA   1800   00
#      8300.0ns  0018      STO   1802   00
#      9100.0ns  001a      LDA   1802   00
#      9900.0ns  001c      SKZ   0000   zz
#     10700.0ns  0020      XOR   1801   ff
#     11500.0ns  0022      SKZ   0000   zz
#     12300.0ns  0024      JMP   0028   zz
#     13100.0ns  0028      XOR   1801   ff
#     13900.0ns  002a      SKZ   0000   zz
#     14700.0ns  002e      HLT   0000   zz
#
# *****
# ** A HALT INSTRUCTION WAS PROCESSED !!! **
# *****
#
```

```
# rom loaded successfully!
# ram loaded successfully!
#
# *** RUNNING CPU test2 --- The Advanced CPU Diagnostic Program ***
#
#
#      TIME      PC      INSTR  ADDR  DATA
#      -----
#      16100.0ns  0000      LDA   1801   aa
#      16900.0ns  0002      AND   1802   ff
#      17700.0ns  0004      XOR   1801   aa
#      18500.0ns  0006      SKZ   0000   zz
#      19300.0ns  000a      ADD   1800   01
#      20100.0ns  000c      SKZ   0000   zz
#      20900.0ns  000e      JMP   0012   zz
#      21700.0ns  0012      XOR   1802   ff
#      22500.0ns  0014      ADD   1800   01
#      23300.0ns  0016      STO   1803   ff
#      24100.0ns  0018      LDA   1800   01
#      24900.0ns  001a      ADD   1803   ff
#      25700.0ns  001c      SKZ   0000   zz
#      26500.0ns  0020      HLT   0000   zz
#
# *****
# ** A HALT INSTRUCTION WAS PROCESSED !!! **
# *****
#
# rom loaded successfully!
# ram loaded successfully!
#
# *** RUNNING CPU test3 --- An Excecutable Program ***
#
#
# * * * This program should calculate the fibonacci * * *
#      -----
#      33150.0ns      0
#      40350.0ns      1
#      47550.0ns      1
#      54750.0ns      2
#      61950.0ns      3
#      69150.0ns      5
#      76350.0ns      8
#      83550.0ns     13
#      90750.0ns     21
```

```
# 97950.0ns      34
# 105150.0ns     55
# 112350.0ns     89
# 119550.0ns    144
#
# *****
# ** A HALT INSTRUCTION WAS PROCESSED !!! **
# *****
#
# ** Note: $finish      : D:/RISC_CPU/cputop/cputop.v(35)
# Time: 121283 ns  Iteration: 0  Instance: /cputop
```

在运行了以上程序后，如仿真程序运行的结果正确，RTL 仿真（即布局布线前的仿真）可告结束。

#### qt\_cpu 工程使用方法：

（1）采用 Verilog HDL 语言顶层文件名为 `cpu.v`，仿真文件名为 `cputop.v`。下面介绍采用硬件描述语言的方法，运用 QuartusII 软件与 ModelSim 联合仿真的步骤。在 QuartusII8.0 下打开工程文件 `cpu.qpf`，将顶层实体名设置为 `cpu`，编译工程。无误后，通过 Tools->Run EDA Simulation Tool->Run RTL Simulation，打开 ModelSim 仿真软件，自动生成 work 目录并将工程中用到的文件重新编译一遍。

通过 File->Open 分别打开 `ram.v`、`rom.v`、`addr_decode.v` 文件进行编译，然后打开仿真文件 `cputop.v`，点击编译按钮，则这些文件均被自动编译进 work 目录下。在 `cputop` 模块上右键单击，在弹出项中点击 `simulate without Optimization`。通过设置观察变量，可观察输出波形或者在 Transcript 窗口观察输出结果。

（2）采用模块化的设计方法，顶层设计文件名为 `qt_cpu.bdf`。可从顶层文件依次进入内部模块中查看其具体结构。

### 4.6.2 RISC\_CPU 模块的综合

在对所设计的 RISC\_CPU 模型进行验证后，如没有发现问题就可开始做下一步的工作即综合。综合工作往往要分阶段来进行，这样便于发现问题。

所谓分阶段是指：

第一阶段：先对构成 RISC\_CPU 模型的各个子模块，如状态控制器模块、指令寄存器模块、算术逻辑运算单元模块等，分别加以综合以检查其可综合性。综合后及时进行后仿真，这样便于即时发现错误，及时改进。

第二阶段：把要综合的模块从仿真测试信号模块和虚拟外围电路模块中分离出来，组成一个独立的模块，其中包括了所有需要综合的模块。然后给这个大模块起一个名字，如本章中的例子。我们要综合的只是 RISC\_CPU，并不包括虚拟外围电路，可以给这一模块起一个名字，例如称它为 `CPUC.v` 模块。见前面测试程序解释时介绍的 `cpu.v` 模块。如用电路图描述的话，还需给它的引脚加上标准的输入输出引脚部件并加标记名字。

第三阶段：把需要综合的模块加载到综合器。可以采用独立的 Synplify 工具，也可以使用 QuartusII 或其他综合工具进行综合。综合器综合的结果会产生一系列的文件，其中有一个文件报告所使用的基本单元，各部件的时间参数以及综合的过程。综合时选定的 FPGA 为 Altera CycloneII EP2C5Q208C8N。在综合结果报告文件中，揭示了综合器对综合过程和结果的分析，有极其重要的意义。它能帮助设计者了解系统运行的最高时钟、关键路径的最

大延迟，使用的逻辑部件的种类和数目。当出现问题时会提示人们：Verilog 源代码的哪个模块第几行有错误或警告。这些资料的熟练应用和分析是很重要的。它能提高设计的工作效率，使综合器综合出更加合理的电路。有关如何利用综合器能处理的综合指令，请有兴趣的读者参考其它相关资料。综合指令和属性是 Verilog 源代码中的一种符合特殊规定的注释行，仿真工具不处理这样的注释行，而综合器却能识别这些符合特殊要求的注释行，根据设计者通过综合指令提出的要求，使综合器综合出更好的符合设计者要求的电路结构。

### 4.6.3 RISC\_CPU 模块的优化和布局布线

选定元件库后就可以对所设计的 RISC\_CPU 模型进行综合，综合工作是把 VerilogHDL 代码通过综合工具，产生一系列由现存元件 逻辑网表组成的文件。在综合工具上通过选择项可以配置生成逻辑网表文件的格式。逻辑网表文件可以是：Verilog Netlist、VHDL Netlist 或者电子设计交换格式（Electronic Design Interchange Format），也就是在电路设计工业界常说的 EDIF 格式文件。在产生了这些文件之后，就可以进行综合后的网表仿真。网表仿真的 Verilog 模型只是对应库逻辑元件的行为模型，并不涉及器件和布局布线的连接线延迟，因此与实际电路的行为还存在着差异，这种仿真模型没有明显的延迟。为了知道实现电路真实的带延迟的行为，还必须进行布局布线操作，以便生成实际电路和连接线带延迟的行为模型。

下面将介绍如何用 Altera QuartusII 进行综合和布局布线，由 RTL 代码产生由对应元件库（Altera CycloneII）Verilog 网表组成的仿真模型以及该网表所提取的延迟参数文件。用 QuartusII 进行综合和布局布线的步骤如下：

- (1) 双击 QuartusII 图标，启动 QuartusII 工具。
- (2) 在 QuartusII 主窗口的工具栏中选择 File->New Project Wizard...，随即弹出对话框，在相应的空格栏选取或者填入工作目录名、项目名和被综合模块组的顶层模块名。
- (3) 单击 Next，随即弹出另外一个对话框，在相应的空格栏中选取或者填入希望被综合的文件名，单击 ADD，添加该文件进入综合环境。
- (4) 单击 Next，随即又弹出一个对话框，在相应的空格栏中选取或者填入实现逻辑的器件型号。
- (5) 单击 Next，随即又弹出一个对话框，选取 EDA 仿真工具，在出现的空格框内选取 ModelSim 和 Verilog 格式。
- (6) 单击 Next，仔细阅读设计者已经配置环境的情况，然后单击 finish，结束综合环境的配置过程。
- (7) 在 QuartusII 主窗口的工具栏中选择 Processing->start compilation，或者直接单击三角形的图标随即开始编译过程。
- (8) 在工作目录中会出现一个新的名为 simulation 的目录打开这个目录，可以看到一个名为 ModelSim 的目录，再打开这个目录，可以看到 3 个文件，分别为 xxx.vo，xxx\_modelsim.xrf 和 xxx\_v.sdo。
- (9) 将 xxx.vo 和 xxx\_v.sdo 复制到工作目录，将 xxx.vo 文件替换原来的 xxx.v 文件再进行一次仿真就能将 xxx\_v.sdo 的延迟信息带入，得到布局布线后的仿真结果。
- (10) 需要注意的是布局布线后的仿真还必须有已经选择库的仿真模型才能进行。这些库究竟在哪里呢？我们可以在 Altera QuartusII 的安装目录里寻找。如果安装在 C 盘上，则在 C:\altera\80\quartus\eda\sim\_lib 目录下可以看到许多型号 FPGA 的元件库的仿真模型，如果用的是 Verilog 模型，只需要在扩展名为.v 的文件中寻找即可，把相应型号的 FPGA 库元件的仿真模型复制到自己的工作目录进行编译就能进行布局布线后的仿真了。

综合和布局布线完成后得到两个文件 cpu.vo，cpu\_v.sdo 和 cpu\_modelsim.xrf。cpu.vo 是

所设计的 RISC\_CPU 的门级结构,即利用 Verilog 语法描述的用 cycloneii 型号 FPGA 库中的基本逻辑电路元件构成的复杂电路连线网络,而 `cpu_v.sdo` 是布局布线的延迟参数文件,`cycloneii_atoms.v` 是 `cpu.vo` 所引用的 Verilog 门级模型的库文件,包含了各种基本逻辑电路的门级模型,它们的参数与真实器件完全一致,包括如延迟等参数。

需要注意的是:必须在布线工具的相关界面上选取生成输出文件的格式为 Verilog 后,`cpu.vo` 和 `cpu_v.sdo` 这两个文件才会产生。将这两个文件和 `cycloneii_atoms.v` 包含在 `cputop.v` 中,来代替原来的 RTL 模块 `cpu.v`。其他外围测试行为模块相同,用仿真器再进行一次仿真,此时称为布局布线后仿真。实际上,后仿真与前仿真的根本区别在于测试文件所包含模型的结构不同。前仿真使用的是 RTL 级模型,如 `cpu.v`,而后仿真使用的是真实的门级结构模型,其中不但有逻辑关系,还包含实际门级电路和布线的延迟,还有驱动能力的问题。仔细观察后仿真波形就会发现与前仿真有一些不同,各信号的变化与时钟沿之间存在着延迟,这些仿真信息在前仿真时并未反映出来,后仿真波形如图 4-17 所示。

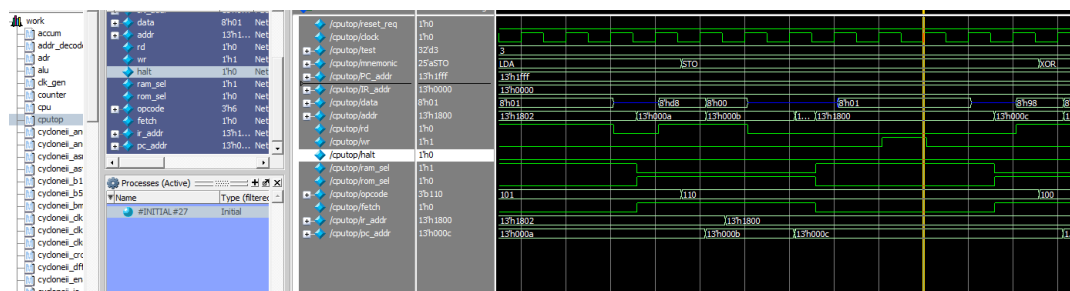


图 4-17 后仿真波形

下面的 Verilog 程序是由布局布线工具生成的,分别命名为 `cpu.vo` 和 `cpu_v.sdo`。由于 `cpu.vo` 是门级描述,共有上千行,而 `cpu_v.sdo` 是延迟参数文件,也有几百行。而 `cycloneii_atoms.v` 是库元件,包含的逻辑元件非常多,无法列出其全部程序,只能从中截取一小段供大家参考。有兴趣的同学可以查看生成的代码并参考 Verilog 语法手册中有关门级描述和用户自定义原语(UDP)来理解这些代码。由于这些代码是 Verilog 的门级模型,又有布线的延迟,所以可以来验证电路结构是否符合设计要求。下面列出了这三个可用于布局布线后仿真的用来代替 RTL 描述的 `cpu.v` 的 Verilog 文件片段供大家参考,帮助同学理解布线后门级仿真原理。

```
//-----cpu.vo 的开始-----
// Copyright (C) 1991-2008 Altera Corporation
// Your use of Altera Corporation's design tools, logic functions
// ...

// VENDOR "Altera"
// PROGRAM "Quartus II"
// VERSION "Version 8.0 Build 215 05/29/2008 SJ Web Edition"

// DATE "02/17/2016 11:24:57"

//
// Device: Altera EP2C8F256C6 Package FBGA256
//
// This Verilog file should be used for ModelSim (Verilog) only
```



```
//
`timescale 1 ps/ 1 ps

module cpu (
    clk,
    reset,
    halt,
    rd,
    wr,
    addr,
    data,
    opcode,
    fetch,
    ir_addr,
    pc_addr);
input    clk;
input    reset;
output   halt;
output   rd;
output   wr;
output   [12:0] addr;
inout    [7:0] data;
output   [2:0] opcode;
output   fetch;
output   [12:0] ir_addr;
output   [12:0] pc_addr;

wire gnd = 1'b0;
wire vcc = 1'b1;

tril develrn;
tril devpor;
tril devoe;
// synopsys translate_off
initial $sdf_annotate("cpu_v.sdo");
// synopsys translate_on//
ooo

// atom is at PIN_F3
cycloneii_io \pc_addr[12]~I (
    .datain(\m_counter|pc_addr [12]),
    .oe(vcc),
    .outclk(gnd),
    .outelkena(vcc),
```

```

.inclk(gnd),
.inclkena(vcc),
.areset(gnd),
.sreset(gnd),
.differentialin(gnd),
.linkin(gnd),
.devclrn(devclrn),
.devpor(devpor),
.devoe(devoe),
.combout(),
.regout(),
.differentialout(),
.linkout(),
.padio(pc_addr[12]));
// synopsys translate_off
defparam \pc_addr[12]~I .input_async_reset = "none";
defparam \pc_addr[12]~I .input_power_up = "low";
defparam \pc_addr[12]~I .input_register_mode = "none";
defparam \pc_addr[12]~I .input_sync_reset = "none";
defparam \pc_addr[12]~I .oe_async_reset = "none";
defparam \pc_addr[12]~I .oe_power_up = "low";
defparam \pc_addr[12]~I .oe_register_mode = "none";
defparam \pc_addr[12]~I .oe_sync_reset = "none";
defparam \pc_addr[12]~I .operation_mode = "output";
defparam \pc_addr[12]~I .output_async_reset = "none";
defparam \pc_addr[12]~I .output_power_up = "low";
defparam \pc_addr[12]~I .output_register_mode = "none";
defparam \pc_addr[12]~I .output_sync_reset = "none";
// synopsys translate_on

endmodule

//----- cpu.vo 的结束-----

//-----cpu_v.sdo 的开始-----
// Copyright (C) 1991-2008 Altera Corporation
// Your use of Altera Corporation's design tools, logic functions
...
// Device: Altera EP2C8F256C6 Package FBGA256

// This SDF file should be used for ModelSim (Verilog) only
(DELAYFILE
(SDFVERSION "2.1")
(DESIGN "cpu")
(DATE "02/17/2016 11:24:57")

```

```

(VENDOR "Altera")
(PROGRAM "Quartus II")
(VERSION "Version 8.0 Build 215 05/29/2008 SJ Web Edition")
(DIVIDER .)
(TIMESCALE 1 ps)
...

(CELL
  (CELLTYPE "cycloneii_asynch_io")
  (INSTANCE pc_addr\[10\]~I.asynch_inst)
  (DELAY
    (ABSOLUTE
      (PORT datain (1198:1198:1198) (1198:1198:1198))
      (IOPATH datain padio (2768:2768:2768) (2768:2768:2768))
    )
  )
)
)
)
(CELL
  (CELLTYPE "cycloneii_asynch_io")
  (INSTANCE pc_addr\[11\]~I.asynch_inst)
  (DELAY
    (ABSOLUTE
      (PORT datain (1349:1349:1349) (1349:1349:1349))
      (IOPATH datain padio (2768:2768:2768) (2768:2768:2768))
    )
  )
)
)
)
(CELL
  (CELLTYPE "cycloneii_asynch_io")
  (INSTANCE pc_addr\[12\]~I.asynch_inst)
  (DELAY
    (ABSOLUTE
      (PORT datain (1504:1504:1504) (1504:1504:1504))
      (IOPATH datain padio (2612:2612:2612) (2612:2612:2612))
    )
  )
)
)
)
)
//----- cpu_v.sdo 的结束-----

//-----cycloneii_atoms.v 的开始-----
// Copyright (C) 1991-2008 Altera Corporation
// Your use of Altera Corporation's design tools, logic functions
.....

```

```
// ***** PRIMITIVE DEFINITIONS *****

`timescale 1 ps/1 ps

// ***** DFFE

primitive CYCLONEII_PRIM_DFFE (Q, ENA, D, CLK, CLRN, PRN, notifier);
    input D;
    input CLRN;
    input PRN;
    input CLK;
    input ENA;
    input notifier;
    output Q; reg Q;

    initial Q = 1'b0;

    table

        //  ENA  D   CLK   CLRN  PRN  notifier  :   Qt   :   Qt+1

        (??) ?   ?       1     1     ?       :   ?   :   -; // pessimism
        x   ?   ?       1     1     ?       :   ?   :   -; // pessimism
        1   1   (01)    1     1     ?       :   ?   :   1; // clocked data
        1   1   (01)    1     x     ?       :   ?   :   1; // pessimism
    ...

    // Optional input registers for dataa,b and signa,b
    always @ (posedge clk_ipd or posedge aclr_ipd or negedge devclrn or negedge devpor)
    begin
        if (devclrn == 0 || devpor == 0 || aclr_ipd == 1)
        begin
            idataout_reg <= 0;
        end
        else if (ena_ipd == 1)
        begin
            idataout_reg <= dataa_ipd;
        end
    end

    // mux input sources from direct inputs or optional registers
    assign dataout_tmp = use_reg == 1 ? idataout_reg : dataa_ipd;

    // accelerate outputs
    buf dataout_buf[dataout_width-1:0] (dataout, dataout_tmp);
```

```
endmodule
```

```
//-----cycloneii_atoms.v 的结束-----
```

从上面提供的带门延迟的布局布线门级网表，可以了解布局布线后产生的带.vo 扩展名的网表的实质。从本质上说，这是一种比综合器产生的更接近实际电路结构的 Verilog 门级和原语基础部件级的源代码。这种代码有自己的仿真行为，但也有确定的电路制造参数与之对应，所以是可以实现的，上面列出的代码明确的说明了这个问题。对于原语基础部件级 Verilog 语法的深入了解是微电子工艺师和电路系统设计师都必须了解和掌握的。

不同的 FPGA 厂家的布局布线工具提供不同的后仿真解决方法，所以很难用一句话作全面的介绍，读者应阅读 FPGA 厂家的布局布线工具的说明书中有关章节，选用正确的 Verilog 门级结构的后仿真解决方案。如后仿真正确无误，就可以把布局布线后生成的一系列文件送 ASIC 厂家或加载到 FPGA 器件的编码工具，使其变为专用的电路芯片。如后仿真中发现有错误，可先降低测试信号模块的主时钟频率，如该问题解决了，则需要找到造成问题的关键路径，下一次在布局布线时应先布关键路径（即在约束文件中注明该路径是关键路径后，再重做自动布局布线）。若还有问题则需检查各模块中是否有个别模块没有按照同步设计的原则。若是，则需改写有关的 VerilogHDL 模块。重复以上工作，直到后仿真正确无误。以上所述的就是用 VerilogHDL 设计一个复杂数字电路系统的步骤，读者可参考以上步骤自己设计一个可在 FPGA 上实现的小型 RISC\_CPU 系统。

## 4.7 小结

复杂的 RISC\_CPU 设计其实是一个从抽象到具体的逐步接近的分析和实现过程。一个大型的设计先从概念出发，用 Verilog 写出抽象的功能块描述，把许多复杂的细节掩盖起来。然后，从行为级分析功能块之间的关系，通过仿真逐步验证，发现问题，改动模块代码使其逐步趋向合理，并最后可以用 RTL 级 Verilog 源代码模块来表示。接下来就可以通过自动综合工具把 RTL 级 Verilog 源代码模块综合成电路网表，再通过布局布线工具让它们更具体化。在基础器件原语级基础上的系统精确仿真结果正确，可以使系统电路芯片的制作有 90% 以上的一次流片成功把握。这就是我们为什么要学习 Verilog 高层次先进设计方法的原因。

### 拓展练习：

改进本章中的 RISC\_CPU 系统，把指令数增至 16，寻址空间降为 4KB，完成课程设计报告，实现两个层次的仿真运行。

## 参考文献

- [1] 潘松，潘明，黄继业. 现代计算机组成原——结构 原理 设计技术与 SOC 实现[M]. 第二版. 北京：科学出版社，2013.
- [2] 贺敬凯. Xilinx FPGA 应用开发[M]. 北京：清华大学出版社，2015.
- [3] 夏雨闻. Verilog 数字系统设计教程[M]. 第 3 版. 北京：北京航空航天大学出版社，2013.
- [4] 张晓彤，张磊，胡玥. 计算机组成原理实验指导书 2012 版. 讲义，2013.