

微程序设计入门

[美] H. 卡赞 著

科学出版社

73.87221
170

微程序设计入门

〔美〕H. 卡赞 著

吴时霖 张根度 陈炳从 译

JS101/16

科学出版社



内 容 简 介

这是一本关于微程序设计入门书。作者在书中简明地介绍了微程序设计的概念和方法以及仿真技术的基本概念和方法。全书共分十章：微程序设计的概念；计算机基础；微程序计算机的组织；微程序操作；仿真入门；翻译程序；模拟程序；仿真原理 I：算术运算；仿真原理 II：解释程序的设计；微程序设计的其他问题。每章后都有讨论题和习题，以供读者进一步理解和掌握基本概念与方法。

本书适宜于计算机专业学习使用。可供计算机设计人员、微型机工作人员、研究微程序设计的软、硬件工作者及大学有关专业高年级学生学习参考。

Harry Katzan Jr.
MICROPROGRAMMING PRIMER
McGraw-Hill, Inc. 1977

微 程 序 设 计 入 门

[美] H. 卡赞 著

吴时霖 张根度 陈炳从 译

责任编辑 那莉莉 李 立

科 学 出 版 社 出 版

北京朝阳门内大街 137 号

中国科学院印刷厂印刷

新华书店北京发行所发行 各地新华书店经售

*

1983 年 7 月 第 一 版	开本：850×1168 1/32
1983 年 7 月 第 一 次 印 刷	印张：9
印数：0001—18,650	字数：229,000

统一书号：15031·508

本社书号：3155·15—8

定 价：1.70 元

译 者 序

本书是 H. 卡赞教授的专著。

迄今为止，各种不同型号的计算机已大量投入应用，在应用中积累了许多丰富的软件资料。面对这种实际情况，人们在研制新的计算机系统时，就不能只是简单地淘汰老系统、老设备，装备新系统、新设备，而往往要求新系统也能运行老系统的应用程序。这就是说，往往要求新、老系统之间具有“程序兼容性”。十多年来的实践已经证实：微程序设计是解决这种程序兼容性的主要手段。这种手段的描述性术语就是微程序仿真。Burroughs 公司著名的 D 机器实际上已是一种仿真机，具有一定程度的通用仿真的能力。D 机器采用了毫微程序设计概念，具有较高的微程序设计技巧。H. 卡赞教授以 D 机器的微程序设计方法为实际背景，针对初学微程序设计和仿真技术的读者，写了这本书，想达到如下三个目标：

1. 使读者初步掌握微程序设计的基本概念；
- 2 教会读者书写微程序；
3. 使读者初步掌握微程序仿真的基本方法。

译者认为，以上三个目标对我国计算机界的许多读者可能也是适用的。如果这本译著能在上述三方面取得效果，就是译者的希望所在了。

中国科学院计算所的惠毓明同志审阅了本书的译稿，在此表示衷心的感谢。

译者 于北京

• • •

序 言

本书的目的是简明地介绍微程序设计和仿真技术，并说明程序、计算机和实现方法之间的关系。我们假定大多数读者通过学习计算机课程或工作实践，对计算机、程序设计语言、操作系统、语言编译程序和程序设计等基本概念已有所了解。显然，对于大多数学过计算机结构这门课程的读者来说阅读本书将更容易，虽然这门课程的知识并不是必要的。

我们打算向读者介绍三方面的知识：(1)介绍微程序设计概念；(2)掌握编制微程序的实际方法；(3)了解仿真原理。这些知识将通过引导性的材料、实际例子以及翻译程序和模拟程序系统介绍给读者，而这些系统则使学生可以在任何一台配备有FORTRAN语言的计算机上编写并执行微程序。

本书的组织将使得编写微程序过程中所必须解决的问题按先后次序罗列出来了，其中也吸收了一些较好的学术观点和与此有关的实际经验。我们将首先向读者介绍微程序设计概念以及它与现代计算机系统结构的关系；其次，介绍仿真概念，微程序设计语言和可编微程序计算机。从而使学生能利用这些工具掌握编写微程序的方法；最后，介绍仿真原理，使学生能运用这些方法去设计和研制实际的系统。

这里所用的翻译程序和模拟程序是用标准 FORTRAN IV 写成的，这些程序能在任何一台配备有 FORTRAN IV 的计算机上执行。根据实际经验，编写这样的程序用不了一天的时间。它们已收录在 McGraw-Hill 出版社出版的教师手册中了。

非常感谢 E. W. Reigel 的帮助，是他引进了D机器的概念。也要感谢 J. T. Lynch 的支持。他们二位都是 Burroughs 公司的成员。Pratt 学院毕业的助手 Sheldon Orloff 和 Myron Sagall 在翻

译程序和模拟程序方面作了辛勤的努力。Calldata 系统的 Hall Robins 和 P.S. Young III 给予了密切的配合和技术支援，而整个工作都得到我的妻子 Margaret 的帮助。

H. 卡赞

目 录

序言

第一章 微程序设计的概念	1
1-1 现代化技术	1
1-1-1 黑箱概念	1
1-1-2 常规计算机的设计及其进展	1
1-1-3 现代计算机的设计及其进展	2
1-2 微程序设计的概念	2
1-2-1 微程序控制	3
1-2-2 微程序存储器	4
1-2-3 基本硬件的功能	4
1-2-4 微程序设计的简要历史	5
1-3 微程序系统举例	6
1-3-1 假想机器	6
1-3-2 微程序功能	8
1-3-3 机器操作	9
1-3-4 注意事项	10
1-4 微程序设计的应用	10
1-4-1 通用计算机	11
1-4-2 控制设备	13
1-4-3 专门应用	13
词汇	13
提问	14
习题	15
第二章 计算机基础	16
2-1 系统构造	16
2-1-1 计算机组织与计算机结构	16
2-1-2 计算机系统概貌	17
2-1-3 数据的表示	18
2-1-4 字与字节	19
2-1-5 字符、逻辑值与数值	20

2-1-6 计算机指令	23
2-2 处理机	25
2-2-1 寄存器	25
2-2-2 机器寄存器的实现方法	30
2-2-3 有效地址	31
2-2-4 处理机的操作	32
2-3 表示方法和术语	34
2-3-1 描述性的表示法	35
2-3-2 微程序设计术语	37
2-4 主存贮器	38
2-4-1 存取宽度	38
2-4-2 主存贮器的操作	39
2-4-3 存贮器变换	40
2-4-4 对界	41
词汇	42
提问	43
习题	44
第三章 微程序计算机的组织	45
3-1 引言	45
3-1-1 实现方法	45
3-1-2 垂直型与水平型微程序设计	45
3-2 系统构造	46
3-2-1 D 机器的组织	46
3-2-2 控制存贮器	48
3-3 微指令	48
3-3-1 分割指令的概念	48
3-3-2 机器周期	49
3-3-3 指令	50
3-3-4 关于 D 机器说明的注释	53
3-4 逻辑部件	53
3-4-1 加法器	54
3-4-2 A 寄存器	56
3-4-3 B 寄存器	57
3-4-4 环移开关	60
3-4-5 存贮器信息寄存器	60
3-4-6 关于逻辑部件操作的注释	61

3-5 存储器控制部件	61
3-5-1 微程序的寻址	61
3-5-2 S 存储器的寻址	62
3-5-3 计数寄存器	65
3-5-4 文字寄存器	67
3-6 控制器	67
3-6-1 移位系统	68
3-6-2 条件系统	70
3-6-3 命令系统	73
3-7 结束语	75
词汇	75
提问	77
习题	78
第四章 微程序操作	79
4-1 概述	79
4-1-1 程序结构	79
4-1-2 语句标号	81
4-1-3 效率	85
4-1-4 求补操作与逻辑“非”	85
4-2 逻辑部件的操作	86
4-2-1 逻辑操作	86
4-2-2 算术操作	87
4-2-3 条件	91
4-2-4 移位操作	94
4-2-5 目标描述	95
4-2-6 下一条命令控制	96
4-3 微程序设计方法	96
4-3-1 常数值的生产	96
4-3-2 交换寄存器中的数据	101
4-3-3 比较操作	102
4-3-4 循环	103
4-3-5 移位	105
4-3-6 其他技术	109
词汇	110
提问	111
习题	111

第五章 仿真入门	113
5-1 任务	113
5-2 有关仿真的微程序设计技术	113
5-2-1 基址寄存器	113
5-2-2 S 存储器的读和写操作	115
5-2-3 文字表	118
5-3 一个简单的仿真程序的设计	122
5-3-1 S 机器的特性	122
5-3-2 S 机器映象	122
5-3-3 仿真程序的微程序	123
5-4 仿真程序的微程序的分析	125
5-4-1 目录表	125
5-4-2 取指令子程序	126
5-4-3 取数子程序	128
5-4-4 存数子程序	128
5-4-5 转移子程序	128
5-4-6 测试零子程序	129
5-4-7 测试最低位子程序	129
5-4-8 减量和增量子程序	130
5-4-9 移位子程序	131
词汇	131
提问	132
习题	132
第六章 翻译程序	133
6-1 概述	133
6-2 微程序设计语言	134
6-2-1 Barkus Naur 范式	134
6-2-2 参考语言	135
6-2-3 TRANSLANG 语言	138
6-3 执行翻译程序	144
6-3-1 处理说明	144
6-3-2 运行实例	145
6-4 十六进制微编码	148
6-4-1 文字赋值语句的刻划	148
6-4-2 毫微指令语句的刻划	150
词汇	154

提问	154
习题	155
第七章 模拟程序	156
7-1 概述	156
7-2 运行模拟程序	156
7-2-1 微程序执行的例子	158
7-2-2 文件名	159
7-2-3 输出格式	161
7-2-4 S 存储器的输入	161
7-2-5 起始地址	162
7-2-6 模拟的最大时钟数	162
7-2-7 两个输出点之间的时钟数	162
7-2-8 输出行	163
7-2-9 输出说明表	163
7-2-10 S 存储器的装入	164
7-2-11 S 存储器的输出	165
7-3 在微程序执行期间的打印输出	165
7-3-1 时钟的一般描述	166
7-3-2 程序控制行	167
7-3-3 逻辑部件寄存器行	171
7-3-4 第一控制寄存器行	172
7-3-5 第二控制寄存器行	173
7-3-6 条件行	174
词汇	175
提问	176
习题	176
第八章 仿真原理 I: 算术运算	179
8-1 概述	179
8-2 定点运算——补码运算	179
8-2-1 加法和减法	179
8-2-2 乘法	181
8-2-3 除法	189
8-3 定点算术运算——原码表示法	194
8-3-1 加法和减法	194
8-3-2 乘法和除法	199
8-4 浮点运算	199

8-4-1	表示法	200
8-4-2	基本浮点算法	202
8-4-3	加法微程序	205
8-4-4	乘法微程序	210
词汇	215
提问	215
习题	216
问题	216
第九章	仿真原理 II: 解释程序的设计	218
9-1	概述	218
9-2	设计考虑	218
9-2-1	计算机的操作原理	218
9-2-2	机器寄存器	219
9-2-3	S 指令格式	219
9-2-4	S 机器操作	220
9-3	堆栈机器的设计和实现	220
9-3-1	波兰表示法	221
9-3-2	指令格式和机器操作	222
9-3-3	机器寄存器的实现	223
9-3-4	操作上的约定	223
9-3-5	堆栈机器的解释程序	223
9-3-6	运行实例	232
9-4	多寄存器计算机的设计和实现	237
9-4-1	指令格式和机器操作	237
9-4-2	机器寄存器的实现	238
9-4-3	操作上的约定	238
9-4-4	多寄存器机器的解释程序	239
9-4-5	运行实例	247
词汇	249
提问	250
习题	250
问题	251
第十章	微程序设计的其他问题	252
10-1	概述	252
10-2	定时	252

10-2-1 时钟脉冲	252
10-2-2 指令相位的描述	253
10-2-3 定时的例子	255
10-3 微程序设计的一些特殊问题	258
10-3-1 动态地址变换	259
10-3-2 表操作	261
10-3-3 矩阵操作	263
10-3-4 向量运算	264
10-3-5 图灵机器	265
参考文献	268
关键字词汇表	269

第一章 微程序设计的概念

1-1 现代化技术

现代社会正经历着一场电子革命，这场革命实际上已影响到计算技术、信息科学和通信科学的所有领域和日常生活的许多方面。所以我们有诸如计算器、精致的收音机、电视机、录音机、监视器与监听器以及品种多样化的通信与控制设备；也正是因为这个原因，电子设备在商业、教育与政府管理部门中得到了大量的使用。

1-1-1 黑箱*概念

如同开汽车一样，对许多其他电子设备或产品来说，为了有效地使用它们，并不一定需要有专门的电子学知识，而这一事实本身把我们引向所谓功能块概念。在功能块中，人们关心的主要是系统的输入、输出以及系统所完成的功能。也就是说，对于仅仅使用功能块的系统设计师、系统工程师或系统分析师来说可以不需要知道功能块内部的具体结构。

1-1-2 常规计算机的设计及其进展

对于计算机的用户来说，计算机可以看作是一个功能块，其理由很简单。因为大多数计算机用户既不是计算机工程师，也不是计算机科学工作者，他们往往是其他领域中的专业人员。事实上，计算机之所以能得到广泛的应用，最重要的一个因素就是没有专门电子学知识的人，也能有效地使用计算机。

* 黑箱即通常所说的功能块。——译者注

引入功能块概念以后,使得一台计算机可通过它的指令系统、指令和数据格式、数据通路的特性以及系统结构来描述。对于常规计算机,一般认为:当将计算机中各部件组织好以后,计算机的体系结构也就确定了。第一代和第二代计算机如果不是全部、至少也是大多数是用这种方式组织起来的,而近来大部分微型机、小型机以及超高速计算机也都是如此。总之,当一台计算机组装好以后,其功能结构就由那些相互连接起来的硬件确定了;除非修改硬件,否则功能是不会改变的。

1-1-3 现代计算机的设计及其进展

对现代大、中、小型计算机的设计,要求比常规硬连部件的计算机具有更大的灵活性。无论是对计算机厂家还是用户,在考虑到经济上与技术上的原因后都希望能有相同体系结构的计算机系列,以便为用户提供不断增长的能力而又无需重编程序。也就是说,在更新计算机时,要求原来的程序能在新机器上运行。而微程序设计技术恰恰能提供这种灵活性。

因此,微程序设计的主要优点是:它使一个指令系统能在几种不同型号的计算机上实现^{*)},而在一台计算机上又能实现多个指令系统。IBM360/370 计算机系列具有前一种优点。具有后一种优点的系统则是通过在许多现代计算机中所用的兼容性特点达到的。

1-2 微程序设计的概念

通常,一个计算机系统中的处理部件可以看作由两类线路组成:即数据通路和控制线路。数据通路包括代码线与存贮代码的元件数字信号在这些代码线与存贮元件中传送和保存。一般说来,数据通过算术逻辑部件后,它的值就改变了。控制线路用来解释

^{*)} 例如,由于微程序设计的优越性,使得同一个系列中的大、中、小型机都能实现同一个庞大的指令系统。——译者注

计算机指令,并在数据通路网络中规定适当的数据传送线路。

1-2-1 微程序控制

在处理部件中,控制线路是一步一步进行操作的,它很像一般计算机程序中的指令。对计算机系统中控制器进行程序设计的过程,就是所谓微程序设计。由于在机器语言的程序设计¹⁾与微程序设计之间存在某种相似性,因此,机器语言程序与微程序也有某种相似性。从概念上说,它们之间的关系如图 1-1 所示。算法是作为面向过程语言或汇编语言中的源程序描述的,然后通过汇编编译程序或其他编译程序将源程序翻译成机器语言程序²⁾。程序中机器指令的执行是通过执行微程序来实现的,这个微程序执行由该指令所规定的操作功能。

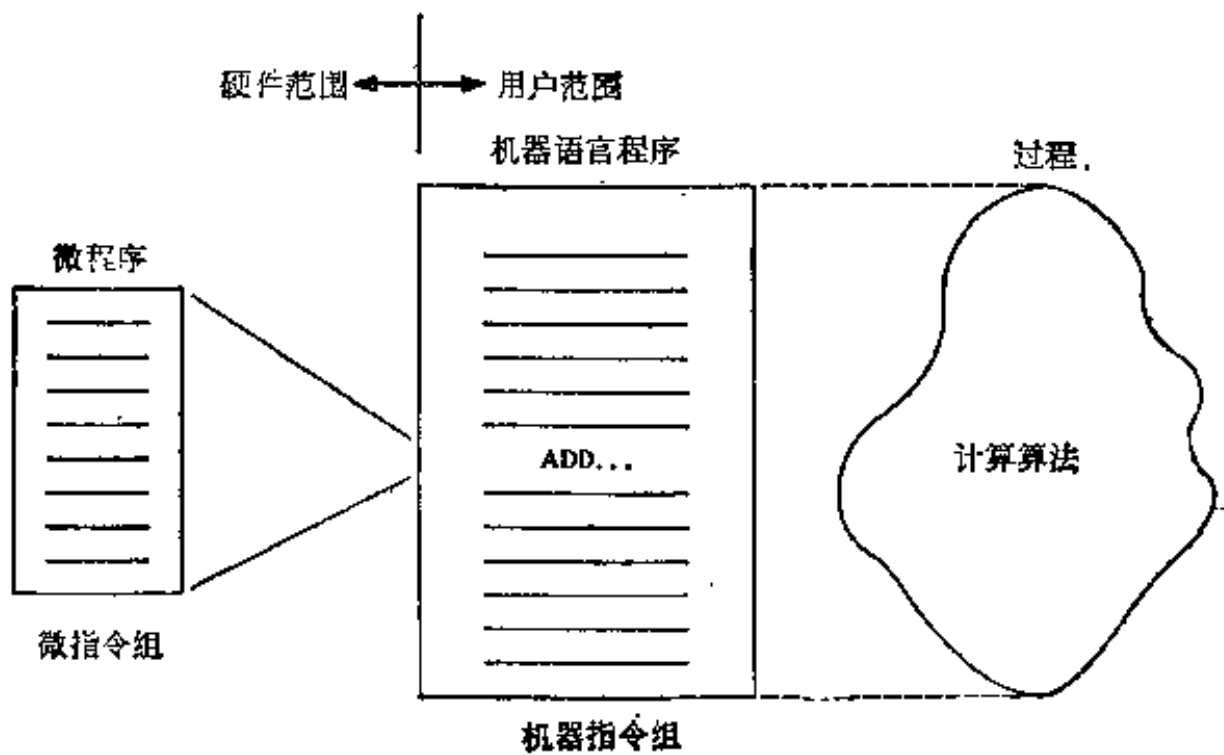


图 1-1 机器语言指令与微指令之间相互关系的概念图

1) 汇编语言程序设计与机器语言程序设计这两个术语时常用作同一意思，它们都与机器级程序有关，汇编语言是机器语言的符号形式。

*2) 原文误为“机器语言指令”。——译者注

然而,这种相似性是有局限性的。因为对于每条指令,并不是都有一组微指令与之对应^{*}。在把面向过程语言中的一个语句翻译成一组机器指令时,这些指令是在语句对语句的基础上产生的。而在微程序级,微程序是通过译出每条指令,进而对其进行解释来执行该机器指令的操作功能的。因此,在操作上,微程序类似于语言解释程序。

1-2-2 微程序存贮器

在执行微程序期间,微程序存放在高速的控制存贮器中,这种控制存贮器通常有两种存取方式,即只读型与可写型。对一台计算机来说,只读存贮器(ROS)的内容是固定的,它不能在微程序控制下进行修改。修改只读存贮器的内容相当于修改计算机,虽然,修改它要比修改硬连控制部件更容易些。对一台计算机来说,可写控制存贮器(WCS)的内容也是固定的,但是它可以在微程序控制下进行修改。因此,有了可写控制存贮器,就可以动态地改变计算机的功能结构。而在微程序控制下进行操作的计算机中,常常既包含有只读型也包含有可写型控制存贮器。

通常,一台计算机系统的微程序是由计算机厂家研制的,用户不能改变它。当把可写控制存贮器加进计算机系统后,由于它允许改变操作方式而提高了系统的操作能力。研制微程序的厂家,通过指令和数据格式以及指令系统从用户角度给出了计算机的外特性,而不规定实现措施,这就使用户也能修改通用计算机系统微程序。

市场上出售的、用户可编微程序计算机一般都是微型机或小型机,它们是专用的或教育上用的可编微程序计算机。

1-2-3 基本硬件的功能

一台可编微程序计算机由一些基本的硬件组成。这些硬件包

^{*} 因为机器语言程序中相同的指令,而相同的指令则公用一组微指令。——译者注

括开关、加法器、寄存器和控制线路。微程序的目的就是利用这些基本设备去仿真通常的机器指令。因此，标准的微程序可以执行下述几种功能：

1. 清除加法器；
2. 把加法器的内容送到累加器；
3. 把一个数据字从缓冲寄存器送到累加器；
4. 把现行地址寄存器加 1。

如上所述，为了执行所需要的机器操作，由计算机执行的每条指令，都需要有相应的微程序进行解释，而这个微程序由若干条微指令组成。例如，执行一条加法指令就需要有能够完成下列步骤的相应的微指令。

1. 从主存中取出指令(即加法指令)；
2. 该指令进行译码；
3. 计算有效地址；
4. 取操作数；
5. 执行加法操作；
6. 将结果送入累加器。

通常，采用公用的微子程序来执行取指令和产生有效地址，而用不同的微子程序来执行不同的计算机操作。

1-2-4 微程序设计的简要历史

早在五十年代初，剑桥大学数学实验室的 M. V. Wilkes 教授就认为，计算机系统的控制线路是按照一系列离散步骤进行操作的，这些步骤就像通常的计算机程序一样。Wilkes 教授提出了微程序设计这个概念。根据这个概念为了按所规定的方式使用系统的其它部件，可以把计算机的控制器程序化。这一概念的提出使计算机研制发生了大变革，但是直到 ROS 被广泛使用的六十年代中期，微程序设计并没有被大规模使用。最初，使用微程序仅意味着计算机的控制器可以用 ROS 形式的微程序来代替。制做好的 ROS (通常是由计算机厂家提供的)不能由计算机程序来访问，并且，对

于既定的 ROS 设备来说,它是固定的。后来随着 WCS 的发展,允许操作人员从控制台或使用计算机指令从主存把微程序写入到 WCS 中去。

从传统的意义上讲,微程序既不是硬件也不是软件。已故的 Ascher Opler 提出了下面的看法:

“为了更好地理解微程序设计的本质……,我认为,有必要引入一个新词——固件。我使用这个术语来表示计算机控制存贮器中的微程序,控制存贮器是针对特殊用途而进行特殊逻辑设计的,例如,它可作为另一台计算机的仿真器。我想固件将会有有一个惊人的发展——很明显这种发展,既与硬件有关又与软件¹⁾有关。”

在所引用的同一篇文章中,Opler 进一步刻画了硬件、软件和固体之间的相互关系:

“以浮点加法为例,在第一代计算机中,它是用普通的子程序来实现的;在第二代计算机中,它是用电路来实现的;在第三代微程序计算机中,它是用存贮在 ROS 中的微程序来实现的²⁾。”

1-3 微程序系统举例

微程序设计是一个复杂的过程,它既要求有软件的知识又要求有基本硬件功能的知识。下面我们通过一台假想计算机及其微程序操作的实例,来对微程序设计进行一般介绍。

1-3-1 假想机器

图 1-2 给出了一台假想的微程序处理机。它是一台单地址计算机,有一个算术运算的累加器。图 1-2 所示的功能部件说明如下:

1) A. Opler, "Fourth-Generation Software", *Datamation*, January 1967. p. 22.

2) A. Opler, "Fourth-Generation Software", *Datamation*, January 1967, p. 22.

指令地址寄存器 (IAR) 保存下一条要执行的指令地址;

存储器的地址寄存器 (SAR) 保存对主存执行读或写操作的存储器地址;

增量器 (I) 将 IAR 加 1;

加法器 (ADDER) 执行加法操作;

指令寄存器 (IR) 在译码期间用来保存指令;

累加器 (AC) 保存计算过程中的结果;

存储器缓冲寄存器 (SBR) 在主存执行读或写操作期间用来保存数据;

主存 (MS) 以字为单位存放指令和数据;

控制器 (C) 使微程序转到指定的微子程序。

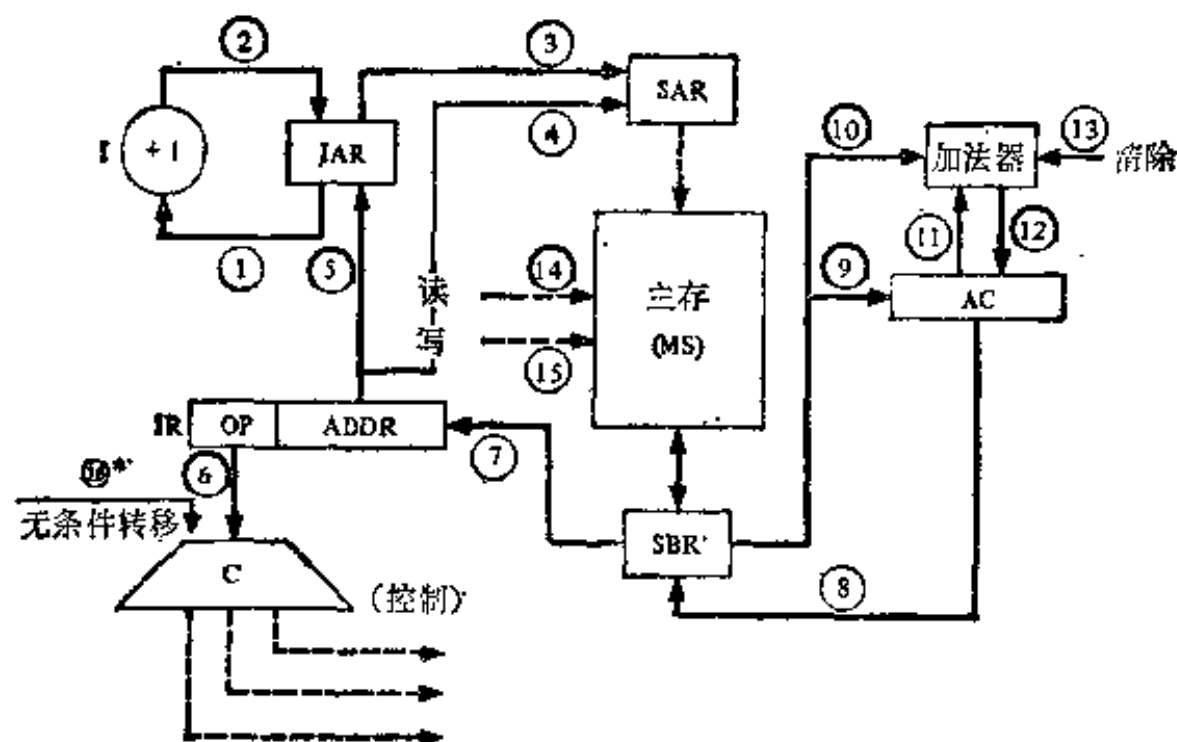


图 1-2 假想微程序处理机

控制器、增量器和加法器等都能完成特定的操作。当给控制器提供一个操作码或者一个微地址时，控制器就使微程序转到相应的微子程序。为此控制器里将包含一个保存在控制存储器中的转换表，当然，也可以使控制器通过硬连线路连到计算机中的各部件。当信息从处理机的一个部件传送到增量器或加法器时，则完

成相应的操作。例如, $AC \longrightarrow ADDER$ (下一节将介绍), 它将累加器的内容送入加法器以进行加法运算。

因此, 微程序设计实际包含着对那些使数据从一个部件传送到另一个部件的微指令的详细说明。

1-3-2 微程序功能

微程序功能就是由执行一条微指令所完成的微程序操作。下面定义的 16 种微程序功能, 其编号对应于图 1-2 中箭头所指出的数据通路。

- ① $IAR \longrightarrow I$;
- ② $I \longrightarrow IAR$;
- ③ $IAR \longrightarrow SAR$;
- ④ $IR(addr) \longrightarrow SAR$;
- ⑤ $IR(addr) \longrightarrow IAR$;
- ⑥ $IR(op) \longrightarrow C$;
- ⑦ $SBR \longrightarrow IR$;
- ⑧ $AC \longrightarrow SBR$;
- ⑨ $SBR \longrightarrow AC$;
- ⑩ $SBR \longrightarrow ADDER$;
- ⑪ $AC \longrightarrow ADDER$;
- ⑫ $ADDER \longrightarrow AC$;
- ⑬ $0 \longrightarrow ADDER$;
- ⑭ $Read[MS(SAR) \longrightarrow SBR]$;
- ⑮ $Write[SBR \longrightarrow MS(SAR)]$;
- ⑯ $(Unconditional\ branch) \longrightarrow C$.

例如, $IAR \longrightarrow SAR$ 功能规定: 把数据从指令地址寄存器传送到存储器地址寄存器。类似地, $IR(addr) \longrightarrow SAR$ 功能规定: 把指令寄存器地址字段的内容传送到存储器地址寄存器。

读和写操作要使用 SAR 和 SBR, 读操作是把由 SAR 中的地址所指定的主存单元的内容传送到 SBR, 写操作是把 SBR 的内容送

到由 SAR 中的地址所指定的主存单元中。

1-3-3 机器操作

目标计算机中的每一个机器操作都必须通过设计一组微程序来实现,这是因为,微程序功能一般不便于单独表示算法过程,也不直接对应于典型的机器指令。为了列举在一台假想计算机中微程序设计的方法,下面给出对应于四种机器操作的微程序。

1-3-3-1 取指令

取指令微子程序用来从主存中取出一条指令,将它送到指令寄存器;将指令地址寄存器加 1;并把操作码送到控制器中去。为了执行由操作码所规定的操作,控制器要使微程序转到相应的微子程序。而 FETCH 微子程序如下。

```
FETCH      IAR $\longrightarrow$ SAR  
           Read[MS(SAR) $\longrightarrow$ SBR]  
           SBR $\longrightarrow$ IR  
           IAR $\longrightarrow$ I  
           I $\longrightarrow$ IAR  
           IR(op) $\longrightarrow$ C
```

1-3-3-2 取数指令

取数指令的微子程序将指令地址字所指出的主存单元的内容传送到存储器缓冲寄存器,然后再传送到累加器。最后,把控制转到控制器,以准备执行下一条指令。相应的取数微子程序如下:

```
LOAD      IR(addr) $\longrightarrow$ SAR  
           Read[MS(SAR) $\longrightarrow$ SBR]  
           SBR $\longrightarrow$ AC  
           (FETCH) $\longrightarrow$ C
```

1-3-3-3 加法指令

加法微子程序首先清除加法器,然后,把累加器的内容和(由

指令地址字段所指定的) 主存单元的内容送到加法器进行加法运算, 并将其结果送到累加器。最后, 把控制转给控制器。相应的加法微子程序如下:

```
ADD      IR(addr) → SAR
          Read[MS(SAR) → SBR]
          0 → ADDER
          AC → ADDER
          SBR → ADDER
          ADDER → AC
          (FETCH) → C
```

1-3-3-4 存数指令

存数微子程序把累加器的内容写入由指令地址字所指定的主存单元中, 相应的存贮微子程序如下:

```
STORE    IR(addr) → SAR
          AC → SBR
          Write [SBR → MS(SAR)]
          (FETCH) → C
```

1-3-4 注意事项

必须注意的是, 作为假想计算机, 上面所给出的几个微子程序的例子是十分简单的。然而, 通过它, 我们可以进一步理解微程序设计的概念。每条计算机指令被“分解”成一些上述提到的基本微功能, 它比通常的机器语言低一级。

1-4 微程序设计的应用

微程序设计的应用分为三种类型。(1) 通用计算机的设计与实现;(2) 设备控制器的研制;(3) 专用计算机的应用。其中, 大多数应用蕴含着使用类似的微程序设计概念, 但最终的目标是明显

不同的。

1-4-1 通用计算机

采用微程序设计的计算机系统一般都必须包括下述几个组成部分：

1. 微程序化的处理机；
2. 主存；
3. 控制存储器；
4. 微子程序组；
5. 外围设备。

微程序加上其他部件就构成了计算机的体系结构。

1-4-1-1 微程序的实现

实现微程序的关键是将所有的计算机都看作解释器。这就是说，处理机被设计成对机器指令所规定的操作进行解释，然后使用给定的操作数执行操作，这种概念有下面两个优点：

1. 一个指令系统可以在几种型号的计算机上实现；
2. 一台计算机可以实现几个指令系统。

在第一种情况下，研制这样的计算机系列是可行的。因为这种系列计算机具有不同的计算能力，但是指令系统却是相同的，这就使得指令系统与计算机的工程设计无关。因此，程序可以在型号不同的计算机上互换，这就为用户提供了扩充性能的途径而不必重编程序。例如 370 系列中的计算机包含有 10 种不同型号^{*)}，每个型号代表一种不同的设计，且都不使用硬连方式的控制器结构。这 10 种型号的机器都具有共同的 370 体系结构，而且，为了按 370 系列方式进行操作，每个型号都是用微程序设计的。

1-4-1-2 仿真

微程序设计的第二个优点是，它使为原来的计算机编制的程

^{*)} S/370 中应有 10 种标准型号，原文写为 9 种。——译者注

序可以不经修改就能在新的计算机¹⁾上运行。所谓仿真,就是用一组微子程序去执行被仿真的计算机的指令。

1-4-1-3 指令模拟

仿真的效率取决于目标机与宿主机兼容的程度。当两台计算机互不兼容时,为了使那些用微编码不能实现或用它实现起来无效的操作功能能够实现,常常要使用软件子程序。因此,应当尽可能采用微程序设计,而只对那些不能仿真的操作,才用软件进行指令模拟。

1-4-1-4 兼容与结合型仿真

兼容性是指在宿主机上运行目标机程序的能力,它常通过微程序设计和专用硬件来实现。

当控制存贮器是只读存贮器时,则宿主机与目标机或者全部兼容或全部不兼容。而正常工作方式与兼容工作方式之间的转换,则可通过暂停计算机操作并置某一选择开关来实现。这样,使得计算机可以任何一种方式运行,但为了改变运行方式,需要人工进行干预。

当控制存贮器为全部或部分可写,即为可写控制存贮器时,则可通过计算机指令来动态地改变其运行方式而无需操作员的干预。这种能力就称作结合型仿真。在输入作业流中,它允许正常方式作业与兼容方式作业混合进行。用户可以直接地或者隐含地用控制卡片指出所需要的方式,而操作系统则读出相应的微程序送到可写控制存贮器中。

1-4-1-5 可变微程序逻辑

众所周知,由于机器语言的结构,使得在一台特定计算机中,用某一程序设计语言编制的程序的效率高于用另一种程序设计语

1) 运行程序的计算机,严格地讲,应称为宿主机。

言编制的同一个程序。因此,有效地运行 FORTRAN 程序的计算机可能不能有效地运行 COBOL 程序。在这种情况下,最好对每种程序设计语言都确立一种有效的机器语言,而对每一种机器语言配备相应的语言编译程序,由它将源程序翻译成不同的机器语言。当目标程序装入主存贮器并准备执行时,操作系统读出用以解释该机器语言的微程序,并送入可写控制存贮器。这种技术称为可变微逻辑,它使得能在机器上有效地执行范围广泛的程序设计语言。

1-4-2 控制设备

在计算机系统中,有些设备的功能实际上就像一台小型硬连方式的计算机。例如通道和设备控制器就是这样。更确切地说,这种设备是包含有一定数量寄存器(但没有主存贮器)的专用处理机。在现代计算机系统中,控制设备经常作为用微程序控制的小处理机来设计。用这种方法来设计与研制控制设备时有许多灵活性。

1-4-3 专门应用

在许多使用小型计算机控制实时处理过程、监督设备操作以及收集试验或实验数据的系统中,程序设计是在机器语言级上进行的,因此,这很复杂。因为,在功能上,机器语言指令不能直接描述处理过程。近来,为这类应用发展了一类微处理机,它是用微程序控制的,且可以根据这类应用的特定要求编制微程序。按照这种方法使用微程序设计时,它所涉及的计算机设备代价是小的,并且,它还提供了一种操作效率更高的系统设计。

词 汇

读者应当熟悉本章中所使用的下列术语:

黑箱(功能块)概念 (Black box concept)

控制线路 (Control circuits)

数据通路 (Data flow circuits)
微程序设计 (Microprogramming)
控制存储器 (Control storage)
机器语言 (Machine language)
微程序 (Microprogram)
只读存储器 (Read-only storage) (ROS)
可写控制存储器 (Writable control storage) (WCS)
固件 (Firmware)
微程序功能 (Microprogram function)
解释程序 (Interpreter)
微程序实现方法 (Microprogrammed implementation)
计算机系列 (Family of computers)
仿真 (Emulation)
指令模拟 (Instruction simulation)
宿主机 (Host computer)
兼容性特性 (Compatibility feature)
结合型仿真 (Integrated emulation)
可变微程序逻辑 (Variable micrologic)

提 问

下面一些问题打算用来测验你对主题内容的理解。所有问题都可以直接从课文或者通过对所提出的主题进行逻辑推理而得到答案。有些问题适合于在讨论班上加以研究。

1. 试指出日常生活中通常认为是黑箱的六种左右的设备名称。
2. 从现代微型电路的发展使得一个完整的处理机可装在一个单片上的事实来看,微程序设计的概念完全可行吗?
3. 微程序设计与机器语言程序设计有什么相似之处?
4. 在现代计算机中,控制存储器应看作是处理机的一部分还是主存储器的一部分? 为了回答这个问题,一个好的方法是考虑这样的做法: 在现代计算机中,既配备控制存储器也配备主存储器。然后你可以用某些现代计算机作为模型来考察(教授能帮助你回答这个问题)。
5. 微程序在哪些方面类似于语言解释程序?
6. “改变只读存储器的内容相当于改变计算机。”这句话如何理解?

7. 假设有一台通用计算机(如 370 系列的机器),它只能运行 FORTRAN 程序,为了进行数值计算.你是否认为:可以给系统重编微程序,使得在同一个硬件基础上,它能更快地执行各种作业?(提示:关于调试的问题可问教授)
8. 试解释**固件**这个术语的必要性和应用。
9. 为了更容易理解模型,本章所讨论的假想计算机已经省略了一些部件和功能.试指出你所知道的哪些明显的省略。
10. 外部设备是计算机系统的重要部分,它用来存贮数据、源程序和目标程序.试问外部设备是否可运用微程序?如果可行,那么应该怎样运用而目的又是什么呢?
11. 在结合型仿真与可变微程序逻辑之间,概念上有相似之处吗?
12. 在微程序设计的计算机中,语言翻译程序把源程序翻译成目标程序.在将目标程序装入并准备执行时,微程序则解释机器语言.一个有趣的问题是:为什么语言翻译程序不把源程序直接翻成微编码?根据一般常识和你的基本计算机原理知识指出这样做的优点和缺点。

习 题

1. 在计算机中,转移指令是这样实现的:把转移地址送入指令地址寄存器,并从该地址继续执行指令.用本章中所给定的假想微程序计算机写出实现转移指令的微程序。
2. 假定你正在考虑进行微程序设计并选择下列题目之一:
 - (a) 对给定的一个符号查找连接表;
 - (b) 内部分类程序;
 - (c) 虚拟存贮器中的动态地址转换;
 - (d) 相联存贮器检索;
 - (e) 在字符串操作中查找与替换符号;
 - (f) 求平方根。

首先,给实现上述每一过程画一个简略的流程图,然后,用机器语言指令给该功能编写程序,最后,假定你正要为该功能设计一条机器指令,为该指令编制实际微编码,并比较这二种实现方法的效率,在这里,基本要求是:考虑微程序设计及它能做些什么.做这个练习时不要过份涉及到具体的细节:数量的概念与顺序的考虑.教授可以帮助你,并可添加一些题目到表目中。

第二章 计算机基础

2-1 系统构造

虽然,本书的大部分读者都熟悉有关计算的基本概念,但是,计算机的迅速发展与广泛使用已使读者可以不必精确地熟悉这些概念.例如,一般认为:在某种意义上,计算机能够扩展人们的智力,正如工具能够扩展人们的体力一样.众所周知,计算机是一台自动机,它必须用程序控制其操作.另一方面,也可以从硬件或算法的观点把计算机组织的概念介绍给读者.所以,重要的是简明而全面地提出适合于微程序设计计算机组织的概念,以此来达到“全面论述计算机组织”的目的.

2-1-1 计算机组织与计算机结构

计算机组织 (Computer organization) 与计算机结构 (Computer architecture) 这两个术语时常被混用.其理由很简单:它们都没有被明确定义,而又经常使用.在通常情况下,二者几乎不需要加以区别.这是因为,计算机构造 (Computer structure) 这个词用在讨论这个概念时便于强调别的总体属性.然而现实世界中的计算机的实际功能表明,必须“约束”基本概念,使得计算机能够从硬件的角度加以综合,而通过固件来构造.

计算机组织这个名词用来描述计算机系统中各部件的特性以及如何把它们实际组织起来.因此,当论述计算机组织时,一般是指“大的”部件,例如处理机、存贮器和设备.在通常的意义下,部件的属性既明显地支配着如何组成系统,也蕴含地决定了系统的功能.计算机组织所描述的是系统的实际结构而不是逻辑结构.

计算机结构是描述计算机系统的逻辑,特别强调系统的内部

特性。因此，它所描述的内部实体（例如，一个通用寄存器或变址寄存器）可以是实际不存在的（它可以是一种逻辑概念，例如，它指出主存中的一个字是用来保存数据的）。因此，计算机的结构，可以以一组硬件的形式存在，这些硬件是用微程序设计的，且具有特定的逻辑结构；或者、更明确地说，它可以以硬件和微程序化结构相结合的形式存在。当一台计算机完全采用硬接线方式时，则对于它来说，除了强调的重点不一样外，计算机组织和计算机结构是相同的。

当用户将一台计算机概念化时，计算机的体系结构就突出起来了。例如，可以对 370 系列编制和发展它的程序系统而不考虑它的特定机型。因此，具有这种体系结构的计算机系统，可以结合考虑经济与性能等因素来加以组织和发展。

2-1-2 计算机系统概貌

一个系统是由若干个有相互连接关系的部件组成的。当将这种系统组装起来后，它就构成计算机系统，或者说它是一台计算机。

计算机系统部件有：(1)处理机，(2)主存贮器，(3)输入/输出子系统。而相互连接关系则使部件之间可通过控制信号与数据传输进行操作。计算机系统的构造如图 2-1 所示。

处理机将执行和控制计算机系统的操作，它由控制部件与算术/逻辑部件组成。控制部件控制计算机自动地进行操作：从主存贮器读出一条指令，对这条指令进行译码，并发出为执行该指令所必需的控制信号。当执行完一条指令后，控制部件读出下一条指令并重复上述过程。算术/逻辑部件完成计算机中的算术和逻辑运算。主存贮器是一个高速存贮器，在处理机操作时，用它来保存程序和数据。因此，在执行程序以前，包含在该程序中的指令必须先写入主存贮器，在程序使用操作数据之前，操作数据也必须预先写入主存贮器。在计算过程中，主存贮器还用来保存中间结果。但并不用它长期保存程序和数据。

输入/输出子系统由输入/输出设备、远程访问设备、大容量存贮器及相应的支持设备组成。

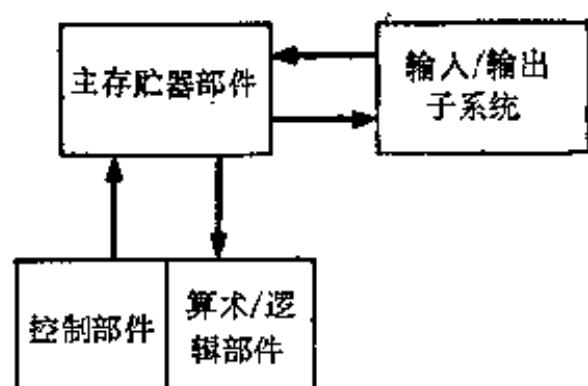


图 2-1 计算机系统概貌

输入和输出设备与记录介质有关，例如有：穿孔卡片机、穿孔纸带机或磁带机、行印机或直接访问存贮器。虽然，输入/输出子系统是计算机系统中的一个重要部件，但是，广泛讨论这个题目已超出本书范围。事实上，许多输入输出部件是微程序化的，读者可参阅附在本书之后的选读资料。

2-1-3 数据的表示

计算机是一台高效率的符号处理机。简单地说，计算机是对数、真值、表达式、字符串或者表示事件的符号进行处理的机器，所以，在计算机内部如何有效的和经济的表示数据是很重要的。已经证明，二进制符号是记录信息的最有效的方法，并且，采用二进制的设备成本低，技术上也容易实现。

单个二进制器件有两种状态，可以表示两个符号。因此，将若干个二进制器件组合起来就可以表示更多的符号。例如，两个二进制器件可组合成下述四种不同符号：

符 号	第一个器件的状态	第二个器件的状态	
1	0	0	用 00 表示的符号
2	0	1	用 01 表示的符号
3	1	0	用 10 表示的符号
4	1	1	用 11 表示的符号

在这种情况下，二进制数字“1”表示“开”状态，而二进制数字“0”则表示“关”状态。每一个二进制数字称为一位。所以，用 n 位

二进制数字可以表示 2^n 个不同的符号或状态。

由于二进制开关器件在工程实现上比较简单,因此,所有现代计算机都采用二进制表示法。尽管,在结构级上是非二进制的计算机,实际在硬件(或组织)级上仍然是二进制机器。

2-1-4 字与字节

主存贮器用来保存处理机要用的指令和数据,并且是根据地址存取信息的。因此,主存贮器可以看成是由许多单元(有时称为存贮单位)组成的,每一个单元都赋予一个地址,并通过地址来存取该单元的内容。

在计算机设计中,一个关键的问题是:一个单元能保存多少信息?这可以从计算机组织和计算机结构的观点来回答。

在计算机组织(硬件)的范围内,一个单元可以保存 8 到 60 二进制位。微型或超小型计算机通常采用 8,16 或 24 位字长。这是因为,这类计算机常常作控制用,在这种应用中,一般不需要高精度的数。而小型、中型、大型和超大型计算机的字长一般在 24 位到 60 位之间。选择字长要考虑性能和成本。对一台计算机来说,字长通常是固定的,因此,每次访问主存贮器,通常是存取一个字的信息。例如,假定一台计算机字长为 32 位(当按字节编址时,它相当于 4 个字节),若处理机需要 64 位(或 8 个字节),那么,为了获得所需要的信息,则需要两次访问存贮器。类似地,若处理机需要 16 位(或 2 个字节),则一次访问存贮器为 32 位,而处理机可从中选取 16 位。

在计算机结构的范围内,可以把一个存贮单元抽象化,使它包含任何适当的位数,以满足计算机的设计目的。这有两种解释:在面向字的计算机中,一个单元中包含若干位(例如 36 位),称它为一个字;在面向字节的计算机中,一个单元包含的位数可少一些(例如 8 位),并称它为一个字节。这样,在面向字的计算机中,一个单元能保存一条指令、一个数据项,或若干个字符信息;在面向字节的计算机中,一个字节单元可保存一个字符信息,而用若干个

连续的字节单元来存贮指令和数据。

在用微程序设计的计算机组织中，既能包含有面向字的计算机结构，也能包含有面向字节的计算机结构。因此，可编微程序计算机一般包含有最小限度的一组指令和寄存器，而具有很高的内部操作速度。例如，一个典型的“基本”可编微程序计算机可以只包含一条定点加法指令，而其他定点算术指令和所有浮点指令都可以通过适当的微程序实现。因为基本的机器是以很高的速度操作的，所以执行一条指令的时间仍不大于使用硬连部件的指令执行时间，考虑到大规模集成技术，实现微程序的成本是低的。

2-1-5 字符、逻辑值与数值

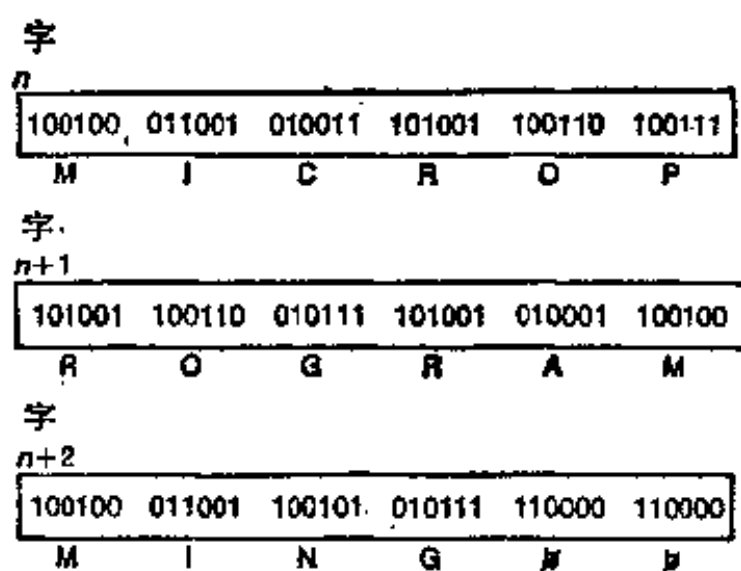
在计算机系统信息是按照设计时的规定来存贮和解释的。当然也还取决于：在执行程序期间，何时使用以及使用该信息的方式。对于一个信息通常可以有四种不同的解释：

1. 作为字符；
2. 逻辑值；
3. 数据；
4. 指令。

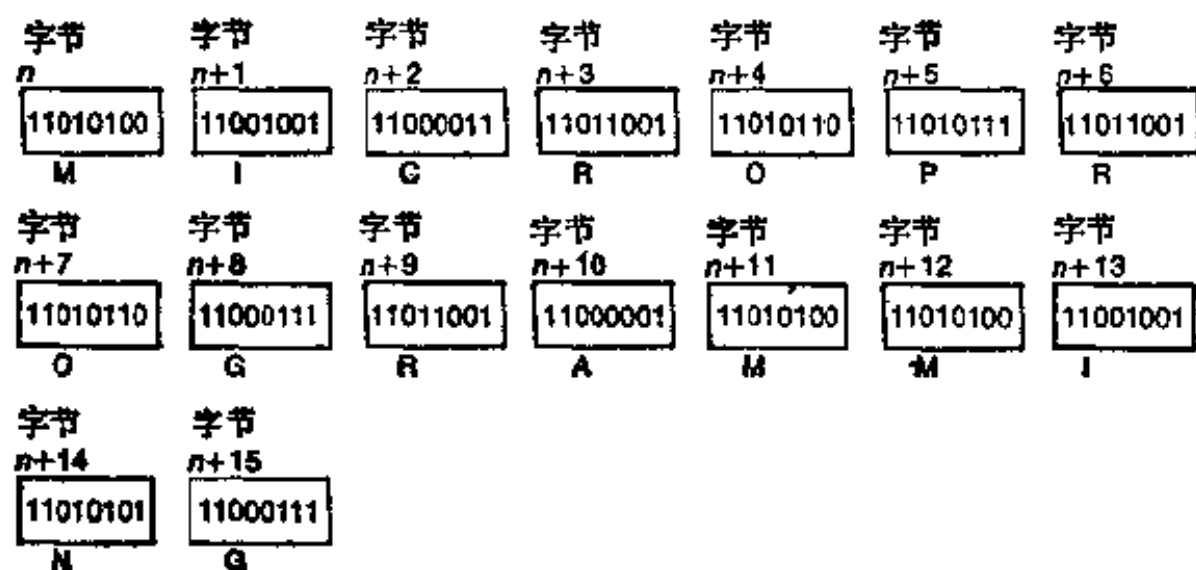
本节简要地讨论字符、逻辑值和数据，在下一节将介绍指令。

2-1-5-1 字符

字符的意义与编码有关，在字母表中的每个字符都用一个既定长度的二进位串来表示。在设计外部设备、研制计算机软件和应用程序时，都要考虑每一个二进制位组的特定意义。图 2-2 给出用 6 位和 8 位编码的字符串的例子，它们分别用在面向字和面向字节的计算机中。关于典型的 6 位和 8 位字符的编码可以在计算机或微据处理的入门书籍中找到。此处所给出的一种编码称为二进制编码的十进制码 (BCD)。



(a) 在面向字的计算机中的 6 位字符



(b) 在面向字节的计算机中的 8 位字符

图 2-2 6 位和 8 位字符表示法

2-1-5-2 逻辑值

逻辑值通常是用 0 或 1 表示，下面给出二种表示法

条件	一位表示法	字节表示法
真(开)*)	1	00001101 (至少有一位置 1)
假(关)**)	0	00000000 (所有位都为 0)

*) 一般指“1”。——译者注

***) 一般指“0”。——译者注

这个概念可以扩充到计算机中的字上去,并且,一般地,逻辑值的表示法与计算机的编码无关。

2-1-5-3 数值

可以利用不同的数制来表示数值,这样的数系可以取两个基数之一:二进制形式或者二进制编码形式。例如,在基数为 10 的数系中,数 123 可按下式求值:

$$1 \times 10^2 + 2 \times 10^1 + 3 \times 10^0$$

在二进制数系中, $(123)_{10}$ 等价于 $(1111011)_2$, 它可按下式求值:

$$1 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$$

二进制是当前计算机中存贮数值数据用的一种主要的基。另一种形式是二进制编码的十进制数,它又有两种形式:(1)数字字符;(2)压缩的十进制数。采用数字字符的数据,其数字信息以 BCD 格式存贮,并能直接显示(即它无需通过输出转换就能打印出来)。然而,在大多数计算机中,数字字符数据在没有被转换成二进制形式或者压缩的十进制形式前,是不能进行算术运算的,很清楚,通过微程序设计可以实现一种特定类型的操作(例如由数字字符数据所规定的算术运算),只要计算机设计者把这类操作特性注入计算机之中即可。对于压缩的十进制数据,每个十进制数字用 4 位二进制表示。这种格式等价于去掉标号后的字符编码。表 2-1 给出数字 0 到 9 的一组 4 位编码,于是,十进制数 6,183 可表示为:

0110	0001	1000	0011
------	------	------	------

所以,压缩十进制数表示法是介于二进制和字符之间的一种“折衷的表示方法”,而且,针对压缩的十进制操作数所规定的算术操作常常通过微程序设计来实现。

表 2-1 4 位压缩的十进制码

数	编 码
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

关于数值的另外一种考虑是与操作数的范围有关的，一般有三种类型：

1. 定点操作数；
2. 浮点操作数；
3. 可变长度操作数。

对每种操作的细节，以后将作为微程序实现方法的例子来介绍。定点和浮点操作是对固定长度的二进制操作数（通常是计算机字或者寄存器）进行的。一般允许使用可变长度操作数的唯一的算术操作是对压缩十进制数的操作，此时操作数保存在主存贮器中。

对存贮器中的数值操作数，其基数与采用面向字节还是采用面向字的计算机结构无关。在面向字节的计算机中，字是由字节组成的；而在面向字的计算机中，一个单字可包含若干个字符。于是，执行算术运算的方式取决于微程序，并且，在某些情况下，还与硬连部件有关，但它与数的表示方法无关。

2-1-6 计算机指令

计算机指令的表示法在概念上是和字符、逻辑值或者数值的表示法相同的，它也是表示成二进位串。所以，重要的是：在取指

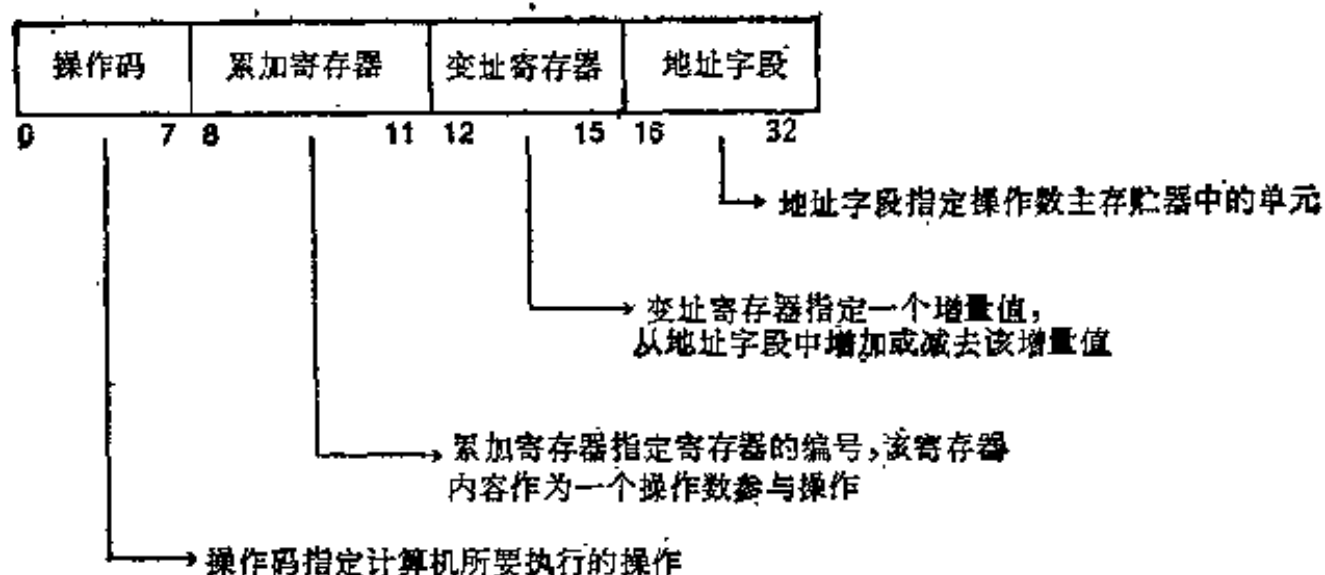


图 2-3 假想计算机的指令

令周期,使处理机去找一条实际的指令;在取操作数周期,使处理机去找正确类型的数据(各种类型的周期在以后再讨论)。一般地,在适当的时刻使用适当的信息,这是程序设计的一个重要特点。

在一条计算机指令中的各种不同字段反映了计算机系统的基本结构。例如,如果处理机包含有多个累加器,那么,指令格式中必须有一个字段来指示参与操作的寄存器号。在图 2-3 中给出了假想计算机的指令。指令的主要字段列出如下:

1. 操作码;
2. 操作数寄存器;
3. 变址寄存器;
4. 地址字段。

每个字段的基本功能如图 2-3 所示。然而,每个字段(除操作码以外)都能指出某一特定操作的辅助操作,例如指定移位操作中被移的位数,或者指定输入/输出操作中的设备地址。

指令中每个字段的长度也实际限制了计算机结构。例如,如果操作码字段有 n 位,那么,就可以表示 2^n 种不同的操作。类似地;如果变址器字段是 3 位,那么、在一条指令中,就可以指定八个不同的变址寄存器。而其中,地址字段的长度也许是最重要的,这

是因为， k 位地址字段允许编址 2^k 个存贮单元。当用字节编址时，地址数目就迅速增加，这使得所能访问的实际存贮器比采用字编址时要少。在某些计算机结构中采用基址寄存器来弥补这种损失。这时，基址寄存器用来指点一个程序段的开始，而地址字段则指出操作数从基址开始的位移量。基址编址工作方式在下面还要讨论。

在研究计算机结构时，注意各种概念的相互影响是有趣的。例如，面向字节的存贮器通常要求采用某种基址/位移量寻址方式，这种寻址方式也需要使用多个通用累加寄存器，而通用累加器又要求在指令中设置一个字段，以指定参与指令执行的累加寄存器号。微程序设计的优点是，计算机结构的设计不受硬件的约束，而且可以很容易地调整计算机结构来适应计算机的发展。

2-2 处 理 机

处理机的主要任务是：取出主存贮器中的指令，译操作码并执行该指令。对不同的计算机，它们的实现方法是不相同的。例如，有些计算机可以预取指令和操作数；另一些计算机则具有多个能并行操作的算术/逻辑部件。然而，在大多数计算机中，指令是顺序执行的，并且，指令和操作数是按正常的次序取出的。因此，本书也采用简单的实现方法。

2-2-1 寄存器

在执行程序时，寄存器是处理机所使用的高速存贮区。累加器、变址寄存器、通用寄存器，或者一个现行地址寄存器都是这种类型的寄存器。有些寄存器（例如通用寄存器）可在程序中指出来，并在执行到该指令时访问它，但是另一些寄存器（例如现行地址寄存器）却不能在指令中直接指出来，而是供处理机操作过程中使用的。

一个正在执行的程序不能访问处理机控制部件所使用的寄存

器，这样的寄存器主要有两种：指令寄存器和现行地址寄存器。执行的程序可以访问算术/逻辑部件所使用的寄存器，这类寄存器主要有三种：算术/逻辑寄存器，变址寄存器，以及寻址寄存器。

2-2-1-1 指令寄存器

指令寄存器中的指令是由处理机中的控制部件进行译码和解释的。从主存贮器取出指令以后，就将它送到指令寄存器。在指令执行期间，指令寄存器中的操作码，寄存器字段，以及地址字段是独立使用的。

2-2-1-2 现行地址寄存器

处理机控制部件中的现行地址寄存器用来保存现行指令的主存地址。当控制部件需要取出一条指令时，它就根据现行地址寄存器的内容访问主存中的指令单元。

在处理机执行了一条指令以后，它就根据被执行的指令长度来修改现行地址寄存器的内容，使得这个寄存器中的内容始终保存下一条指令的地址。这也说明指令在主存贮器中一般是顺序存放的。

2-2-1-3 算术/逻辑寄存器

关于算术/逻辑寄存器，一般采用两种方式配置：单累加寄存器或多个运算寄存器。

当单累加寄存器用来进行算术/逻辑操作时，除了逻辑操作以外，定点和浮点操作都是在该寄存器中执行的。但使用单累加器必须频繁地存贮中间结果。例如在下面的例子中：

高级语言
 $A = (B + C) * (D - E)$

汇编语言
LOAD B
ADD C
STORE TEMP

LOAD	D
SUB	E
MULT	TEMP
STORE	A

单累加器实际上是不够用的,这是因为,定点乘法运算产生双倍长的积,而定点除法运算则要用双倍长的被除数。因此,单累加器系统要对累加器加以扩充,如下例:



或是另一种如:



在具有多个运算寄存器的系统中,常常把定点寄存器(或者,通常称为“通用寄存器”)与浮点寄存器区别开来。这种区别的优点是:作为辅助手段,通用寄存器能灵活地用作变址寄存器或者基址寄存器;浮点寄存器能有较长的宽度,以提高精度。而使用多累加器减少了把信息暂存到主存贮器的次数。这可用下例说明:

高级语言	汇编语言
$A \leftarrow (B + C) * (D - E)$	LE 2,B
	AE 2,C
	LE 4,D
	SE 4,E
	MER 2,4
	STE 2,A

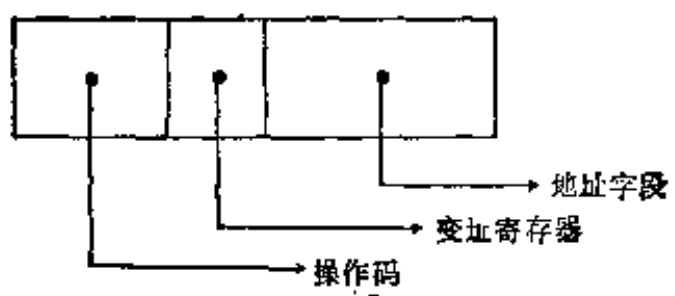
在通用计算机中,累加器的设计本质上是成本、效率与灵活性三因素在工程上的一种折衷。

2-2-1-4 变址寄存器

在对数组进行运算操作时,能在不改变实际指令中的地址部

分的前提下去有效地修改指令中的地址是很必要的. 这便于使用一条指令可用于数组中不同的元素.

在高级语言中,数组元素用下标表示,如 $X(I)$. 假定 $X(I)$ 的内容已送入累加器中(在单累加器的计算机中),并且计算机具有如下的指令格式:



若在语言中使用从 1 开始变址的方式,并且数组已存贮在从 4567 号单元开始的连续单元中,则下列计算机指令是合适的:

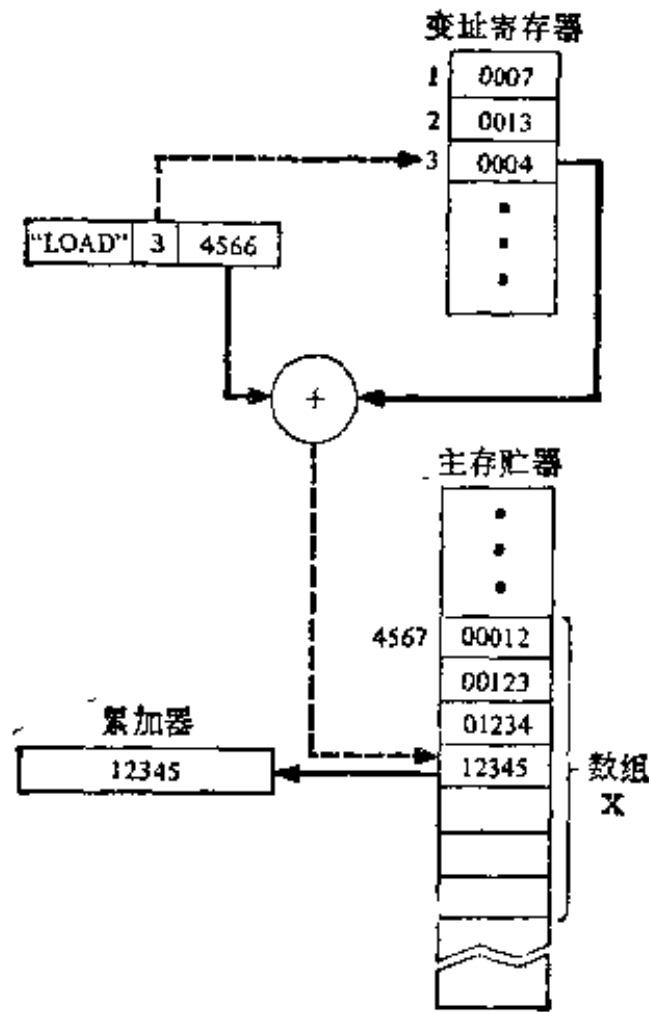


图 2-4 变址的一个例子

汇编语言(符号操作数)

LDX 3,1
LOAD X - 1,3

汇编语言(数值操作数)

LDX 3,6669
LOAD 4566,3

假定 1 已保存在 6669 号单元, 而 3 号变址寄存器保存着下标。LDX 指令把变址值送入 3 号变址寄存器, 而 LOAD 指令则把所需要的操作数送入累加器。借助如下方式实现变址: 处理机把指定的变址寄存器的内容与指令中的地址字段相加, 计算出被取操作数的地址。从概念上看, 变址过程可如图 2-4 那样描述。在图中 3 号变址寄存器的内容是 4, 它指定数组 X 中的第四个元素。

在多累加器的计算机中, 一般把一个或多个通用寄存器用于变址, 而且不需要专门的变址寄存器。

2-2-1-5 寻址寄存器

它是用于寻址的寄存器, 它弥补了面向字节计算机中指令地址字段较短的不足。当程序装入主存贮器后, 通用寄存器之一作

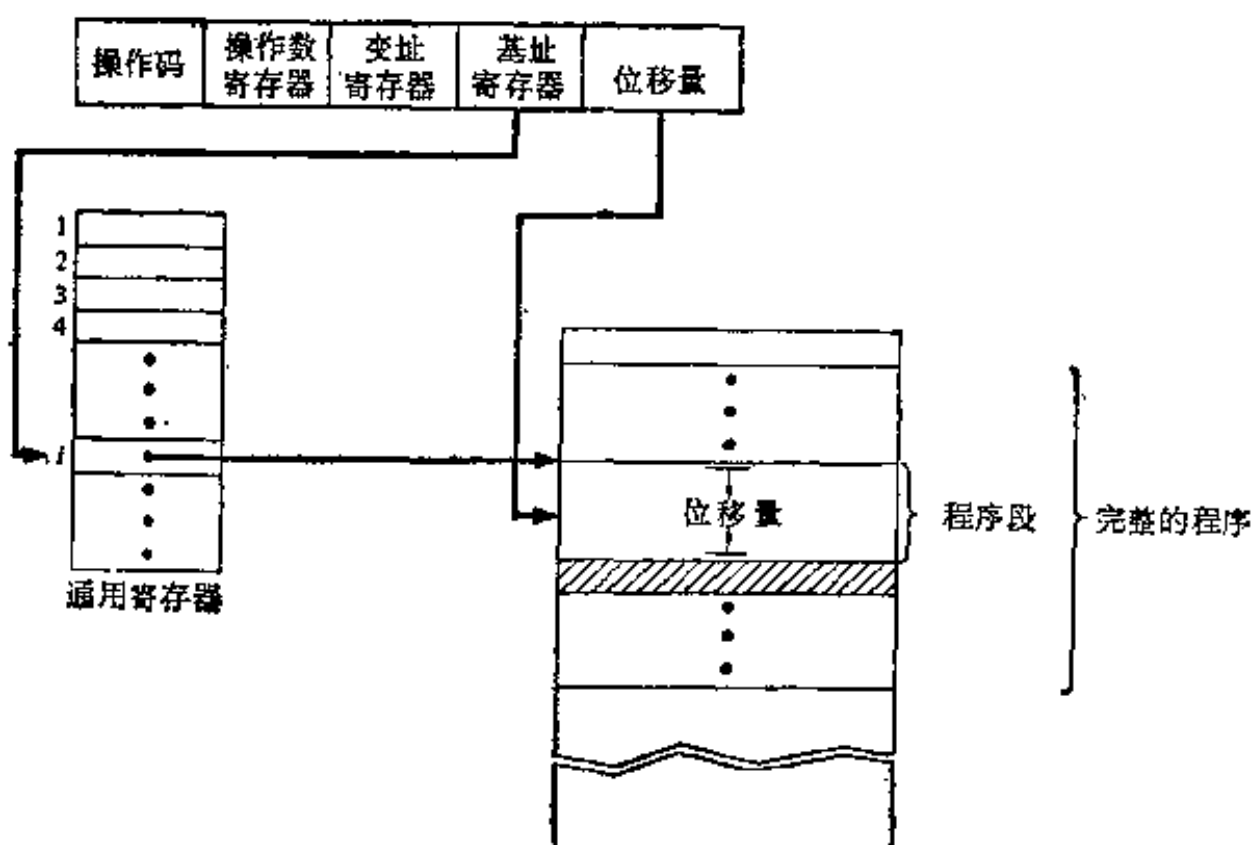


图 2-5 寻址(基址)寄存器的用法

为基址寄存器,用它来指点一段程序的开始;而指令地址字段则作为从基址开始的位移量.寻址(基址)寄存器用法如图 2-5 所示,以后还要讨论.

2-2-2 机器寄存器的实现方法

不管一个计算机系统是硬连的还是微程序设计的,设计时都有一个如何实现这些寄存器的问题.应当强调指出,寄存器的设计与计算机结构有关,而不是与计算机的组织有关.所以,在处理机中,寄存器未必需要作为单独的实体存在.

寄存器通常有三种实现方式:

1. 它是处理机中专门设计的寄存器;
2. 采用处理机中不是专门设计的寄存器(它们是固定的寄存器或是暂存器);
3. 利用计算机系统的主存储器来存放寄存器.

处理机中控制部件所使用的寄存器(即现行地址寄存器和指令寄存器),通常采用第 1 种实现方式或第 2 种实现方式,且用硬件实现.处理机的算术/逻辑部件所使用的寄存器(即累加器、变址寄存器和寻址寄存器),在硬连计算机中通常采用硬件寄存器,而在微程序设计的计算机中或者采用硬件寄存器,或者利用主存储器.利用主存储器的一部分作为寄存器,是为了降低硬件成本.然而,它将使操作时间拉长.

所有按微程序设计的计算机都包含有为了正常执行程序所必需的内部寄存器;但是,在大多数情况下,这些内部寄存器并不就是计算机结构中寄存器.所以必须按下列方式来指定各个寄存器:

1. 指定微程序处理机中的内部寄存器作为计算机结构寄存器;而且这些内部寄存器的作用是一对一的.
2. 修改(或设计)微程序处理机中的硬件,使它包括计算机结构所需要的寄存器.这是专门研制的机器寄存器.
3. 当用到结构寄存器时,则使用微程序处理机中的内部寄存

器暂时作为结构寄存器。当不用结构寄存器时，则将它保存在主存贮器中。

利用主存贮器保存结构寄存器时，每次访问寄存器都必须从主存贮器中取，这需要花费处理机的时间。同样，信息还必须存入主存贮器，这也要花费处理机的时间。

在上述三种实现计算机结构寄存器的方式中，没有一个方式会比另外二个来得好，所以，设计时应该考虑满足经济上与性能上的要求。

2-2-3 有效地址

处理机操作特性的一个重要方面是对主存贮器操作数地址的计算。这个地址称为有效地址。事实上，有效地址概念的含义是

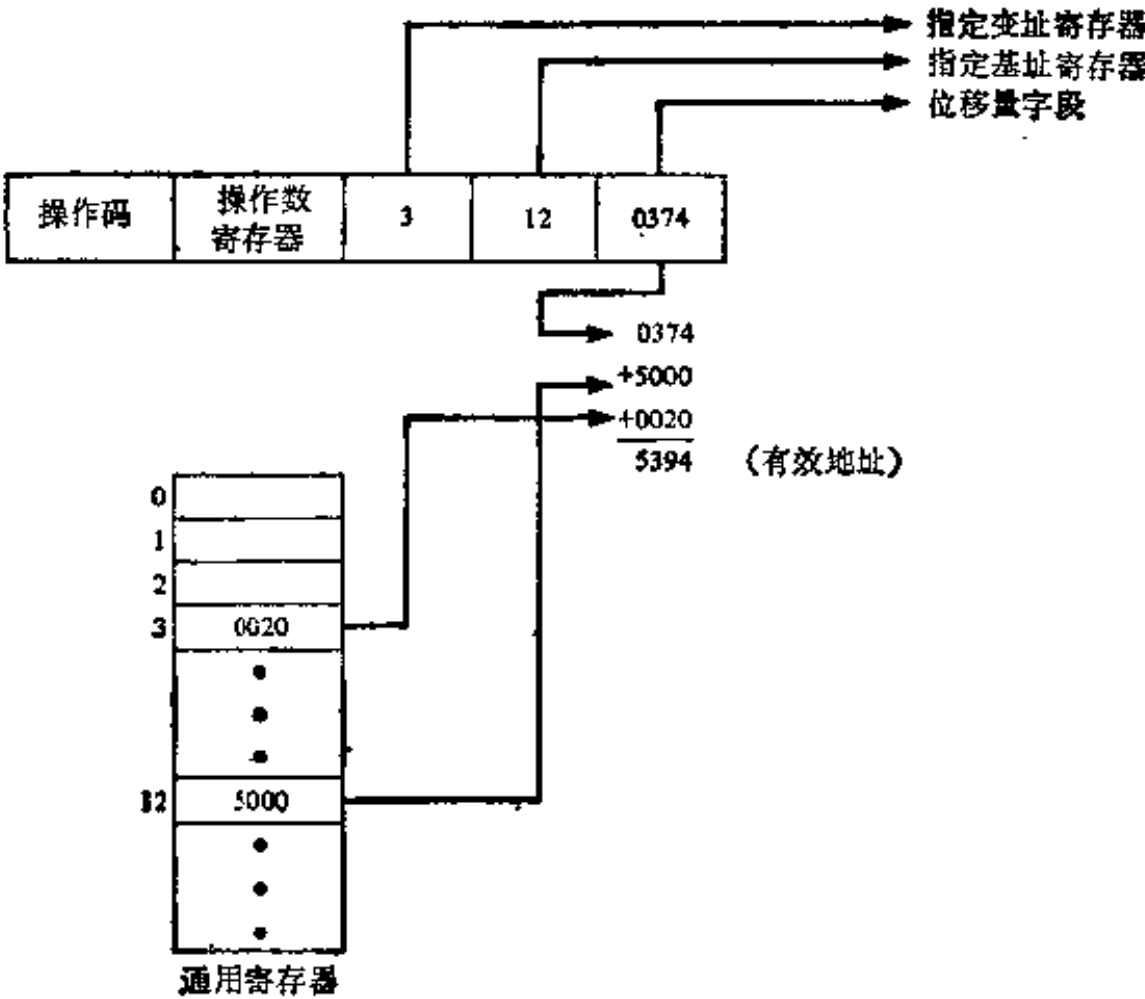


图 2-6 在使用变址/基址寄存器/位移量寻址方式的计算机中形成有效地址的例子

固有的：它必须由处理机进行计算，并且，在微程序实现的情况下，需要用微指令去完成这个计算。

一个有效地址是这样计算的：把变址寄存器与/或基址寄存器的内容相加后再加上指令中地址的位移量部分，从而形成访问主存贮器的地址。涉及变址、有效地址计算的一个例子已经给在图 2-4 中。对于具有变址/基址/位移量寻址的字节编址的计算机而言，图 2-6 说明了有效地址形成的过程。

有效地址是在取出指令并且由控制部件译码以后，在算术/逻辑部件使用操作数之前，由处理机进行计算的。

2-2-4 处理机的操作

计算机中的内部操作都在特定的时间间隔内进行，该时间间隔受到从一个电子时钟发出的脉冲信号的控制，而电子时钟的频率可高达每秒十亿个脉冲信号的数量级。一定个数的脉冲间隔组成一个机器周期。在一个机器周期中，计算机能完成一个或多个特定的微操作，这些微操作与其他一些微操作组合起来执行一条计算机指令。在一条指令中，确切的微操作数是可变的，它与特定的指令和操作数的特性有关。处理机按指定的顺序操作，以实现指令的取出、解释和执行。应当区分两种主要周期：指令周期和执行周期。这两种周期合起来确定了处理机的操作。

2-2-4-1 指令周期

指令处理的第一个周期就是指令周期，常称为“*I* 周期”。在“*I* 周期”内，处理机的控制部件完成下列操作：

1. 从现行地址寄存器取下一条指令的地址；
2. 从主存贮器取出指令并送入指令寄存器；
3. 检查操作码，以确定要执行的功能；
4. 识别基址寄存器、变址寄存器，操作数寄存器；
5. 计算有效地址；
6. 按照正在处理的指令的长度，更新现行地址寄存器的内容

(即按一个字或者按给定的字节个数增量)，使它指向下条顺序执行的指令地址。

从概念上看，1 周期可概括成图 2-7。控制子程序利用操作码的编码，把微程序转到相应的微子程序去。有效地址通常存放在一个内部寄存器(常常称为地址寄存器)中，以供某些微子程序用作操作数地址，或者作为指令的辅助操作位——如移位位数或输入/输出地址。

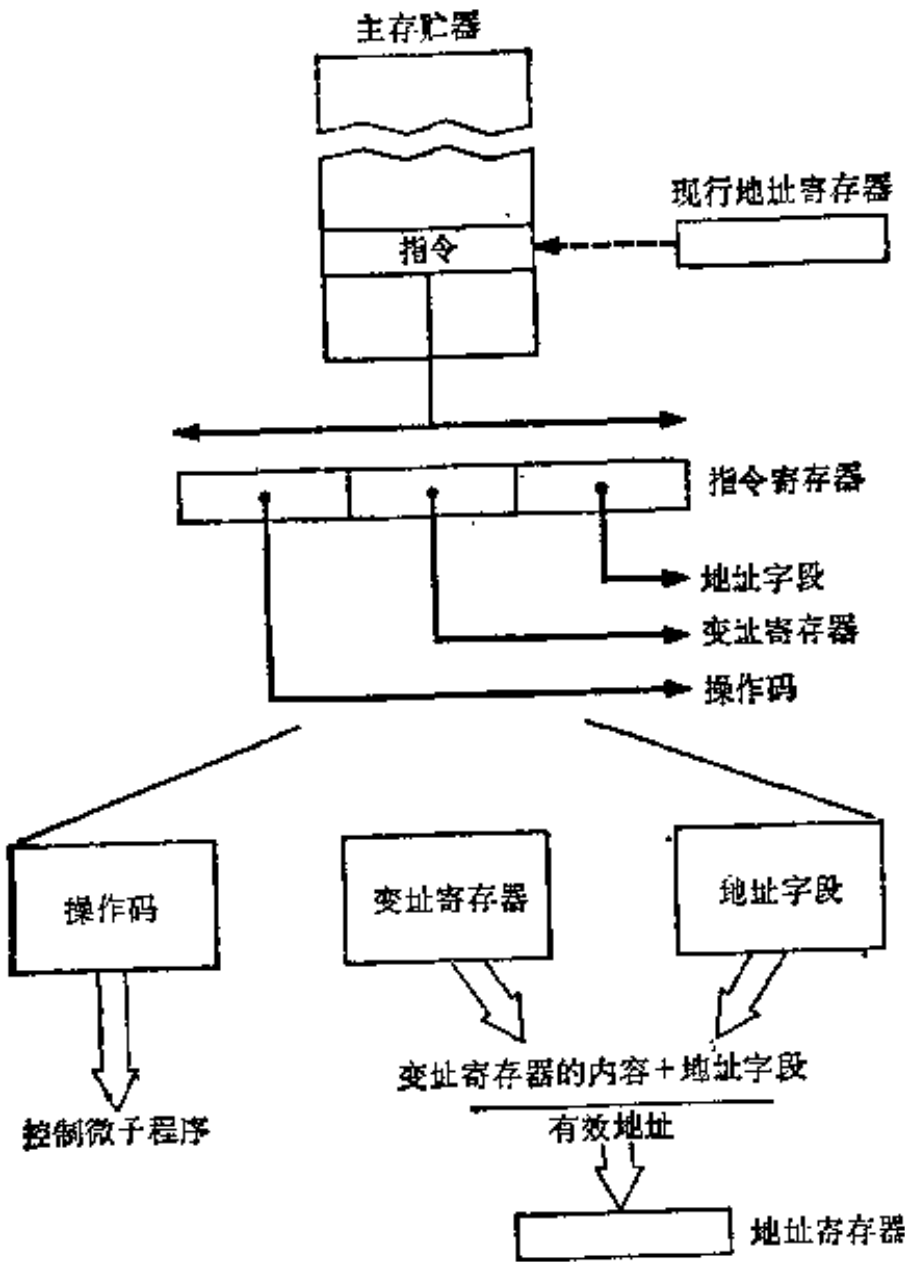


图 2-7 采用变址/寻址方式的计算机的 I 周期的概念图

尽管对指令周期,不同计算机有不同的考虑,但是一般都取固定长的时间,且用机器的内部周期来度量。这是因为 I 周期中,所有指令都公用同一个取指令、译码指令以及形成有效地址的子程序。事实上,在 I 周期内,控制部件一般辨别不出、也不需要知道具有不同操作码的指令之间的差别。

2-2-4-2 执行周期

指令周期的后面总是接着执行周期(称作“E 周期”),在执行周期中,利用内部硬件执行指定的操作。并按照下列方法使用上述有效地址。

1. 用作操作数地址。例如取数指令中的取数地址或加法指令中的加数、被加数地址;
2. 用作写数地址。例如存贮指令中的送数地址;
3. 用作下条指令的地址。例如转移指令中的转移目标地址;
4. 用作指令的修改位。例如移位操作的移位长度或输入/输出操作中的输入/输出地址。

显然,所执行的操作与要完成的计算特性有关,也随着完成这种计算所需要的机器周期数而变化。执行时间可以从几个机器周期(对于转移指令)变到很多机器周期(对于浮点乘法指令)。

E 周期中微子程序的长度,也与所执行的操作有关。因此,为了节省控制存贮器的容量,微程序常常以模块方式构成,所以,微程序设计很类似于常规的计算机程序设计。在常规计算机程序设计中的“时间与空间”问题,也实用于微程序设计。

2-3 表示方法和术语

因为基本的机器概念对于描述和使用都是既麻烦效率又低。所以,为了便于讲授,就采用描述性的表示方法和有关的术语。这种表示方法和术语使我们能以更加确切的方法来论述最基本的概念。

2-3-1 描述性的表示法

在讨论主存贮器单元时,我们一般对两个问题有兴趣:

1. 单元的地址;
2. 单元的内容.

在机器语言中,借助数值地址来识别单元;而在汇编语言中,则借助符号地址来识别它. 例如,在下面的指令中:

```
LOAD      321
STORE     RES
```

经常会遇到用数值或符号加以识别的存贮单元的内容,这些单元内容用括号把单元地址括起来表示. 例如,456号单元的内容用(456)表示,符号单元MARK的内容用(MARK)表示. 这种表示方法也适用于寄存器,因此,(reg#2)表示寄存器2的内容.

在所述的计算机中,正常访问主存贮器是以一个计算机字为单位的. 如果访问按存贮器的另一种单位进行时,那么,这种访问可以用下面的规定来表示.

一个字的一个部分(例如指令的地址字段和操作码字段,或者浮点数的阶码字段)可用下标表示. 例如,(IR)_{addr}表示指令寄存器内容的地址字段. 类似地,(SAREA)₀₋₇表示存贮器单元SAREA内容的第0位到第7位,而(9989)₁₂₋₁₅表示存贮器单元9989的内容的第12位到第15位. 为了表示多重寻址,括号可以再套括号. 如果指令寄存器的12位到第15位表示基址寄存器,那么((IR)₁₂₋₁₅)就表示该基址寄存器的内容. 对这个表示式解释如下:

1. (IR)表示计算机指令,它包含在指令寄存器中;
2. (IR)₁₂₋₁₅表示基址寄存器的地址字段;
3. ((IR)₁₂₋₁₅)表示基址寄存器的内容.

算术和逻辑操作是用通常的方式表示的,只是括号不再用作分组号. 例如,表达式

(MARK) - (reg#2)

表示 MARK 的内容减去 2 号寄存器的内容,而表示式

$$(SARE) + 4$$

是指符号单元 SARE 的内容加 4. 朝右指的箭头 (→) 表示替换, 于是

$$0 \rightarrow (MARK)$$

是指值 0 替换单元 MARK 的内容. 表达式是从左到右执行的, 所以

$$(reg\#3) + 1 \rightarrow (reg\#2)$$

是指“3 号寄存器内容加 1 后替换 2 号寄存器的内容”. 在表 2-2 中给出了其他一些例子.

表 2-2 描述性表示法的例子

表 示 式	意 义
$KTABLE \rightarrow (reg\#12)$	KTABLE 的地址替换 12 号寄存器的内容
$(A) * (B) \rightarrow (C)$	A 的内容乘以 B 的内容, 替换 C 的内容
$(reg\#7) + 4 \rightarrow (reg\#7)$	7 号寄存器的内容加 4, 替换 7 号寄存器原来的内容
$(reg\#1) + (FACT) \rightarrow (reg\#1)$	1 号寄存器的内容加上 FACT 的内容, 替换原来 1 号寄存器的内容
$(VAL1) \rightarrow (reg\#3)$	VAL1 的内容替换 3 号寄存器的内容
$(reg\#3) \rightarrow (VAL2)$	3 号寄存器的内容替换 VAL2 的内容

为了说明可以用描述性的表示法描述机器指令, 考虑下面的指令:

$$ADD \quad 3, INCRE$$

它的意义是: “符号单元 INCRE 的内容加到 3 号累加寄存器的内容上去”. 在描述性表示法中, 该指令可描述如下:

$$(reg\#3) + (INCRE) \rightarrow (reg\#3)$$

类似地, 指令

$$LOAD \quad 6, BIG$$

和

STORE 6, LITTLE

可以用符号分别描述为

和 $(BIG) \longrightarrow (reg\#6)$
 $(reg\#6) \longrightarrow (LITTLE)$

为了计算有效地址,使用描述性表示法是很方便的. 例如,在采用变址/地址的寻址方式中,对每个操作数的有效地址可用下面的方法计算:

$(\text{变址器}) + \text{地址} = \text{有效地址}$

类似地,在采用基址/变址/位移量的寻址方式中,有效地址的计算可描述如下:

$(\text{基址}) + (\text{变址}) + \text{位移量} = \text{有效地址}$

描述性表示法最经常的用法是,以精确和方便的形式来描述计算机的指令.

2-3-2 微程序设计术语

当人们介绍微程序设计概念时,出于下面一些有关原因,有时会产生混淆:

1. 涉及到多种程序设计语言;
2. 可用多种类型的指令;
3. 使用多种类型的存贮器.

因此,例如我们必须知道所说的是主存贮器还是控制存贮器. 下面的术语已被大多数微程序设计文献所采用,也为本书所采用.

术语	意 义
H 语言	高级语言. 例如 ALGOL, COBOL, FORTRAN 或 PL/1.
S 语言	存在于普通计算机中的机器语言 (例如 360/370 中二进制表示的机器语言,但不是汇编语言).
S 指令	用 S 语言表示的一条指令.
S 程序	由 S 指令组成的程序.

S 存贮器	在执行期间保存 S 指令的主存贮器。
M 语言	微程序机器语言,它常常以二进制形式出现。
M 指令	一条微指令。
M 程序	由 M 指令组成的微程序。
M 存贮器	在执行期间保存微程序的控制存贮器。

对上述术语作如下一些说明,程序员用 H 语言(如 FORTRAN 或 COBOL)书写程序。这种程序被翻译成由 S 指令组成的 S 程序,并被送入 S 存贮器以待执行。每一条 S 指令都由 M 程序中的一些 M 指令加以解释和执行。M 程序保存在 M 存贮器中,或者,按通常的术语说,M 程序保存在控制存贮器中。

2-4 主 存 贮 器

在微程序计算机系统中,主存贮器(即 S 存贮器)一般被认为是处理机以外的部件,当系统操作需要使用 S 指令和数据时,微程序化的处理机就从主存贮器取出 S 指令和数据。当需要从主存贮器取出信息时,处理机必须执行用以获得该信息的一组微指令(即一组 M 指令)。当处理机或者正在执行的计算机程序的操作需要将信息存入主存贮器时,处理机也必须执行一组微指令。访问主存贮器的方式取决于它的操作特性。

对我们来说,主存贮器与 S 存贮器是同一个东西。随时都可能使用主存贮器这一术语,这是因为,它是更加通用的术语,而且在那些非程序化的系统中全部使用这个术语。

2-4-1 存取宽度

存取宽度是指: 在访问主存贮器的一个周期中,主存贮器与主存贮器外的一个部件之间所传送的信息量。此处,我们涉及到计算机组织,并且,对于每一次访问,被存取的信息量都是相同的。访问主存贮器的典型部件是处理机和输入/输出子系统。

从概念上讲,存取宽度是指在主存贮器与处理机之间的信息

通道的数目(即连线的数目),若连线数目较多(例如是 32 或 64 根线),则信息位可以以 32 位或 64 位长度并行传送,且所需要的时间相对地短,这时计算机的操作速度快、成本高。若连线数目较少(例如 8 根或 16 根线),则并行传送的数目就减少了,这使得传送一定量的信息(例如,一个 64 位的字)所需要的时间也相对地长了,这时计算机操作速度变慢,但成本降低。这个概念说明:为什么一台机器周期短的计算机的有效操作速度会比另一台周期相同的计算机慢。

程序员是看不到存取宽度的,他只能使用通过微程序实现的计算机结构。

2-4-2 主存贮器的操作

通常,当执行程序中的一条取数指令时,可以把操作看成为数据从源(通常是主存贮器或寄存器)到目标(通常是寄存器)的简单传送。这种思想也适用于用到主存贮器操作数的指令,也适用于送数指令。然而,与主存贮器操作的实际过程并非那样简单,这已在第一章微程序控制的介绍中说明了。

由存贮器控制部件来执行访问主存贮器的操作,而存贮器控制部件本身在逻辑上又可以看作主存贮器的一部分。信息从主存贮器传送到处理机或输入/输出子系统的操作过程如下:

1. 由处理机或者输入/输出子系统把要读取的数据字的地址送到存贮器地址寄存器中(注意:要读取的数据可以是被取出的数据的一部分;也可以是使被取出的数据只是所要访问的数据的一部分。而由微程序来确定被访数据的范围);
2. 在完成第一步后,存贮器控制部件启动读操作;
3. 主存贮器进行读操作,并将读出的数据送到存贮器缓冲寄存器,而处理机或输入/输出系统都可以访问该缓冲寄存器。

把信息从处理机或输入/输出系统传送到主存贮器的过程,则与上述过程相反,其操作如下:

1. 要存贮的数据的地址送入存贮器地址寄存器;

2. 要存贮的数据送入存贮器缓冲寄存器；
3. 存贮器控制部件启动写操作，以完成存数操作。

在执行一条 S 指令期间，要存贮的数据总数可以大于或小于信息的存取宽度。并且在许多计算机中都包含有可变长度的 S 指令，它能有效地把信息从主存贮器的一个区域传送到另一个区域，而传送的总长度可以大于存取宽度。为了成功地执行一条 S 指令，读和写操作的正确顺序必须受相应的微子程序的管理和控制，该微子程序保存在控制存贮器中，以供微程序设计的处理机使用。

2-4-3 存贮器变换

在进行微程序设计时的一个设计问题是如何将计算机结构级上定义的 S 存贮器变换到实主存贮器。当结构级上所用的指令和数据的字长与主存贮器存取宽度相等时，则没有问题。在这种情况下，S 程序的取或存正好与主存贮器的取或存对应。同样，在面向字的计算机结构中，若其字长是存取宽度的整数倍时，那么变换也是简捷的。例如，若 S 存贮器字长是 32 位，而主存贮器字长是 16 位，则对 S 程序每取或存一次，都要求对主存贮器取或存两次。如果在 S 存贮器和主存贮器中，字地址都是连续排列的，则可采用下面的地址变换：

第一个半字： $2 \times [\text{S 存贮器地址}] \longrightarrow [\text{主存贮器地址}]$

第二个半字： $2 \times [\text{S 存贮器地址}] + 1 \longrightarrow [\text{主存贮器地址}]$

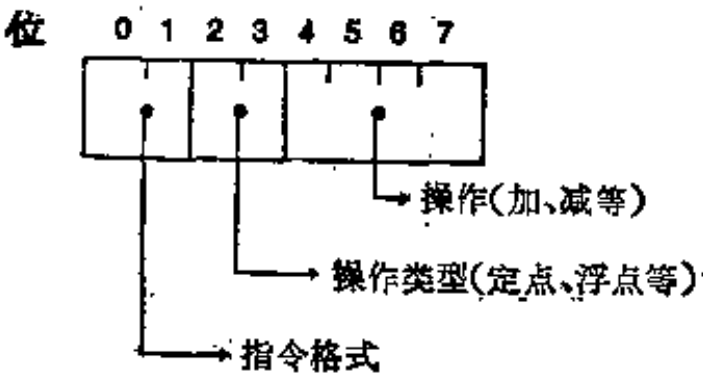
在完成 S 存贮器有效地址的计算后，必须由取数微子程序来计算主存贮器地址。当 S 存贮器字长小于存取宽度时，则地址变换后还要给出访问主存单元的地址，而在该单元中包含有所需要的字。例如，若 S 存贮器的字长为 16 位，而主存贮器字长为 48 位，则可以采用如下的地址变换，这种变换把 S 存贮器的三个字变换到主存贮器的一个字：

S 存贮器字 $\text{FLOOR}([\text{S 存贮器地址}] \div 3)$
 $\longrightarrow [\text{主存贮器地址}]$

其中，函数 $\text{FLOOR}(X)$ 定义为不超过自变量 X 的最大整数。很

清楚,当 S 存贮器与主存贮器不匹配时,则存贮器地址变换将导致降低主存贮器的使用效率,或者导致需要增加额外的存贮器访问。

把 S 存贮器的字节变换到主存贮器的字,这种存贮器变换是上面给出的后一种变换方法的特殊情况。但是,由于指令和数据常常超过主存贮器的字长,因此,对于字节编址,重要的是能识别长度特征,一般,微程序是能识别这种特征的。操作数的长度特征一般蕴含在 S 指令的操作码中,或者作为一个操作数长度字段明确地包含在 S 指令中。S 指令本身的长度特征通常蕴含在操作码字段中,因此,在必要时,可以进行附加的取操作。例如,在 360/370 计算机系统中,指令的操作码字段是 8 位,分割如下:



指令格式字段指出指令字的长度和构成,其定义如下:

0 位和 1 位	指令格式	指令长度(按字节计算)
0 0	寄存器 → 寄存器型	2
0 1	寄存器 → 变址寄存器型	4
1 0	存贮器直接型	4
1 1	存贮器 → 存贮器型	6

有些计算机使用固定长度的指令,但也有少数例外。例如当现行指令的下一个顺序字是用来增补该指令时,就是这种例外情况。

2-4-4 对界

在面向字节的计算机中,容易产生一种情况,即: 由于信息在存贮器中安排不当,使得多半由 4 或 8 个字节组成的一个 S 存贮

器字,会跨越两个或者更多的主存贮器字. 当出现这种情况时,为了存取信息就必须增加访问存贮器的次数. 缓和这种情况的一种方法是,要求把固定长的一些信息单位(如半字、全字以及双倍字)安排在对于该信息单位为整数的边界上. 所谓 S 存贮器中一个给定字是按整边界存放的,是指要求它的存贮器地址是以字节为单位的字长的倍数. 例如,可以这样安排一个 4 字节的字,使其地址是 4 的倍数. 这样安排信息的方法称为对界,它与存取宽度相配合时,就能减少访问存贮器的次数.

另一种对界是字节对界,此时,存贮器中所有信息单位都安排在字节边界上. 对界和字节对界这两种方法都用于微程序设计的计算机中.

词 汇

读者应当熟悉本章所使用的下列术语:

计算机组织 (Computer organization)

计算机结构 (Computer architecture)

系统 (System)

主存贮器 (Main storage unit)

输入输出子系统 (Input/output subsystem)

控制部件 (Control component)

算术/逻辑部件 (Arithmetic/logic component)

二进制符号 (Binary symbol)

位 (Bit)

字 (Word)

字节 (Byte)

存贮单元 (Storage location)

二进制编码的十进制 (Binary coded decimal)

逻辑数据 (Logical data)

数值数据 (Numeric data)

压缩的十进制 (Packed decimal)

数值字符数据 (Numeric character data)

字符数据 (Character data)

字节编址 (Byte addressing)
字编址 (Word addressing)
寄存器 (Register)
现行地址寄存器 (Current-address register)
指令寄存器 (Instruction register)
变址寄存器 (Index register)
算术/逻辑寄存器 (Arithmetic/logic register)
基址寄存器 (Base register)
有效地址 (Effective address)
机器周期 (Machine cycle)
指令周期 (Instruction cycle)
I 周期 (I-cycle)
执行周期 (Execution cycle)
E 周期 (E-cycle)
描述性表示法 (Descriptive notation)
H 语言 (H-language)
S 语言 (S-language)
S 指令 (S-instruction)
S 程序 (S-program)
S 存储器 (S-memory)
M 语言 (M-language)
M 指令 (M-instruction)
M 程序 (M-program)
M 存储器 (M-memory)
存取宽度 (Access width)
存储器地址寄存器 (Storage address register)
存储器缓冲寄存器 (Storage buffer register)
存储器地址变换 (Storage mapping)
对界 (Boundary alignment)
字节对界 (Byte boundary alignment)

提 问

下面一些问题打算用来测验你对本章内容的理解,所有问题都可以直接

从课文或者通过对所提出的问题进行逻辑推理而得到答案。有些问题适合于在讨论班上研究。

1. 研究或者简单地考察你持有的计算工具——计算机。你有在结构级上或者在组织级上的机器概念吗？
2. 计算机系统有哪些主要部件？每一个主要部件又由若干主要子部件构成，等等。如此继续向下，分成更多的级，并指出主要部件中的主要子部件。
3. 术语位（bit）是什么的缩写？
4. 一个单元中保存有多少信息？
5. BCD是什么？
6. 压缩十进制的优点是什么？
7. 何时使用基址寄存器？
8. 在计算机结构中对操作码的长度通常有什么限制？
9. 寄存器单元与存储器单元有什么差别？
10. 现行地址寄存器为什么是重要的？
11. 是否能够设计一台没有算术/逻辑寄存器的计算机？在什么级上考虑，是结构级还是组织级？
12. 有效地址是在处理机的那个周期中进行计算的？
13. 存储器地址寄存器有什么作用？存储器缓冲寄存器呢？
14. 为何字节对界效率低？
15. 至少说出在微程序设计的计算机中所使用的三种类型的存储器的名称。

习 题

1. 到目前为止，我们已经把一台计算机概念化为物理的或逻辑的部件的集合。但是，我们都知道：许多用户都从简单程序设计语言的观点来看待整个计算过程。试用你自己选择的一种程序设计语言来把计算机概念化。
2. 假设逻辑数据的形式是：真、假和任意。试作出存贮这种数据的办法。
3. ASCII 代码是什么？如何使它适合于图解？
4. 在什么方式下压缩十进制的效率低？详细说明之。
5. 说明：怎样才能做到不要现行地址寄存器？
6. 作一些调查并鉴别利用主存储器存放寄存器的计算机。
7. 用符号描述性表示法描述下面的语言

$$A(I) = B(I + 1) + 4$$

第三章 微程序计算机的组织

3-1 引言

本书的目的是介绍微程序设计的概念和有关技术。从概念上看,微程序设计类似于常规的计算机程序设计,只不过它是在更基本的机器级上、针对不同的目标实现的。然而,研究微程序设计并不就是研究微程序计算机。在这个意义上,它也可以和常规计算机程序设计相比拟。关于微程序计算机的设计,可参看有关计算机设计方面的工程书刊。

3-1-1 实现方法

为了进行微程序设计,需要有一台微程序计算机,本书中利用了一台称为D机器的基本机器。对于D机器,可利用一个完全的逐位模拟程序,并把它作为适当微程序设计语言的翻译程序。在本书的“教师指南”中给出了用标准 FORTRAN IV 书写的模拟程序和翻译程序。

本书其他部分的内容是要说明微程序的设计过程,其中包括仿真技术以及为支持这些技术所需要的计算机方法。而对硬件的讨论将放在次要地位。然而,为了提供一种真实结构和说明微程序设计的细节,D机器是足够复杂的了。

3-1-2 垂直型与水平型微程序设计

在实现微程序设计时,常用的有两种方法:垂直型微程序设计与水平型微程序设计。对于垂直型微程序设计来说,每条M指令都单独指定一个微操作,并且微子程序是由比较长的、顺序执行的M指令序列组成的。对于水平型微程序设计来说,每条M指令

都指定若干个可以同时并行执行的微操作。

每种方法都有优点和缺点，垂直型微程序设计的优、缺点如下：

优 点

微程序容易编制微指令中各位都得到了充分利用

缺 点

执行微子程序要用许多机器周期

不能利用硬件固有的并行性

优点

水平型微程序设计的优、缺点如下：

优 点

执行微子程序所需要的微指令相对地少

可以利用硬件能并行执行的

优点

缺 点

编制微程序较难

除非算法排得“紧凑”，否则

指令位就不能充分利用

两种方法都有可取的优点。如同下面一些章节所述的那样，D 机器运用了垂直型和水平型的特性，因而是研究微程序设计的理想工具。

3-2 系 统 构 造

D 机器的介绍分几步进行。这一章包括系统的总构造、D 机器中各部件所完成的功能以及系统操作的方式。以后各章讨论模拟程序的用法、翻译程序的用法以及微程序操作。

3-2-1 D 机器的组织

D 机器的组织如图 3-1 所示。系统由三个主要的硬件子系统和两个控制存储器组成，它们是：

1. 逻辑部件；
2. 控制部件；
3. 存储器控制部件；

4. 微程序存贮器;
5. 毫微程序存贮器。

这一节介绍逻辑部件、控制部件和存贮器控制部件,下一节讨论存贮器。

逻辑部件执行由控制部件指出的 D 机器中的微指令。逻辑部件完成的功能是: 算术和逻辑操作、移位操作以及对 S 存贮器和外部设备的数据输入和输出操作 (D 机器的控制存贮器只能存放微程序, 而不能存放 S 程序和数据)。控制部件通过命令、移位位数以及操作中的条件来控制逻辑部件的操作。存贮器控制部件提供微程序存贮器、S 存贮器和外部设备的寻址方式, 随时修改现行

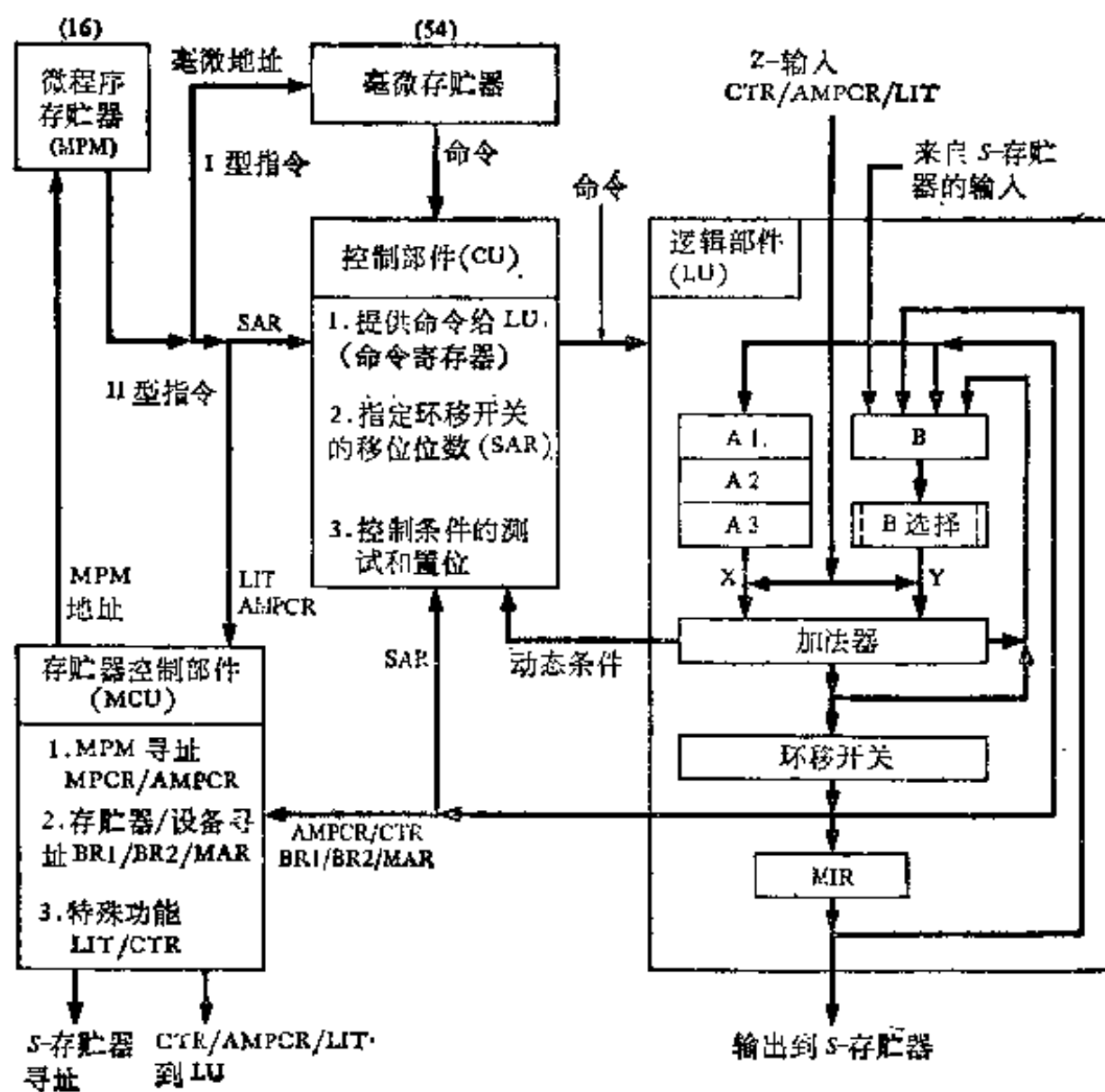


图 3-1 D 机器的组成

地址寄存器以及保存一个文字值和一个计数器。

3-2-2 控制存储器

D 机器中的控制存储器被分成微程序存储器和毫微程序存储器。微程序存储器字长短，它包含有与每条指令对应的微程序入口。微程序存储器的用法与垂直型微程序设计过程相类似，因为它包含了相当多的“小”指令。存放在微程序存储器中的某些 M 指令本身是完备的和独立的，而另一些 M 指令(尤其是那些使用逻辑部件的 M 指令)则必须增加某些指令位，才能有效地控制逻辑部件的操作。毫微程序存储器就是用来存储所需要的附加指令位。毫微程序存储器中的每条微指令的较长，因而使逻辑部件能够并行操作。因此，毫微程序存储器的用法与前面所说的水平型微程序设计有些类似。只有需要使用毫微程序存储器的那些 M 指令才指向它。所以，控制存储器不会浪费。况且，若两条 M 指令需要同样的毫微程序指令(称为 N 指令)，则每条 M 指令能指向同一条 N 指令。

通常，微程序存储器简写成 MPM；毫微程序存储器简写成 nano。

3-3 微 指 令

D 机器的微指令分两种类型：I 型与 II 型。I 型指令利用毫微存储器，而 II 型指令不用毫微存储器。这两类指令都源发于微程序存储器中的一条指令。

3-3-1 分割指令的概念

图 3-2 给出分割指令概念的图解。由 D 机器执行的每一组操作都开始于一条 M 指令。这里讲的“一组”操作，可以只包含单个微操作，也可以包含一系列能并行执行的微操作。但是，使用逻辑部件的所有操作，都需要一条存放在毫微程序存储器中的 N 指令。

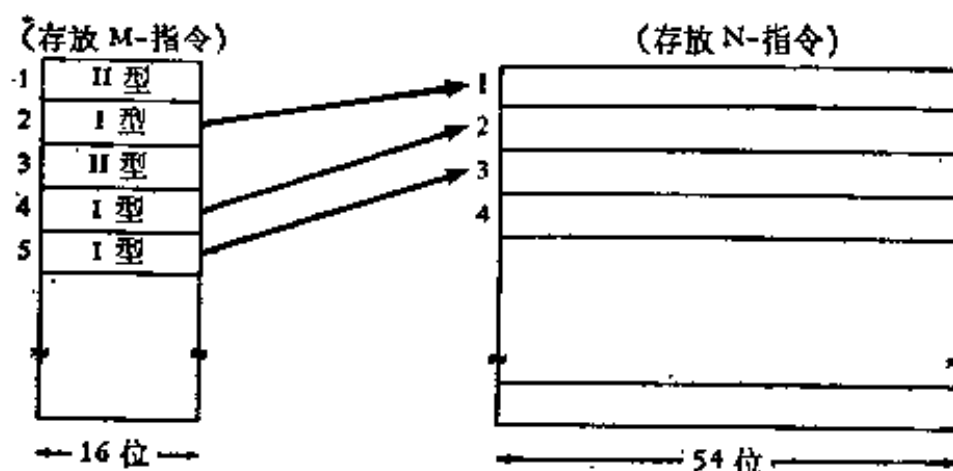


图 3-2 分割指令概念的图解

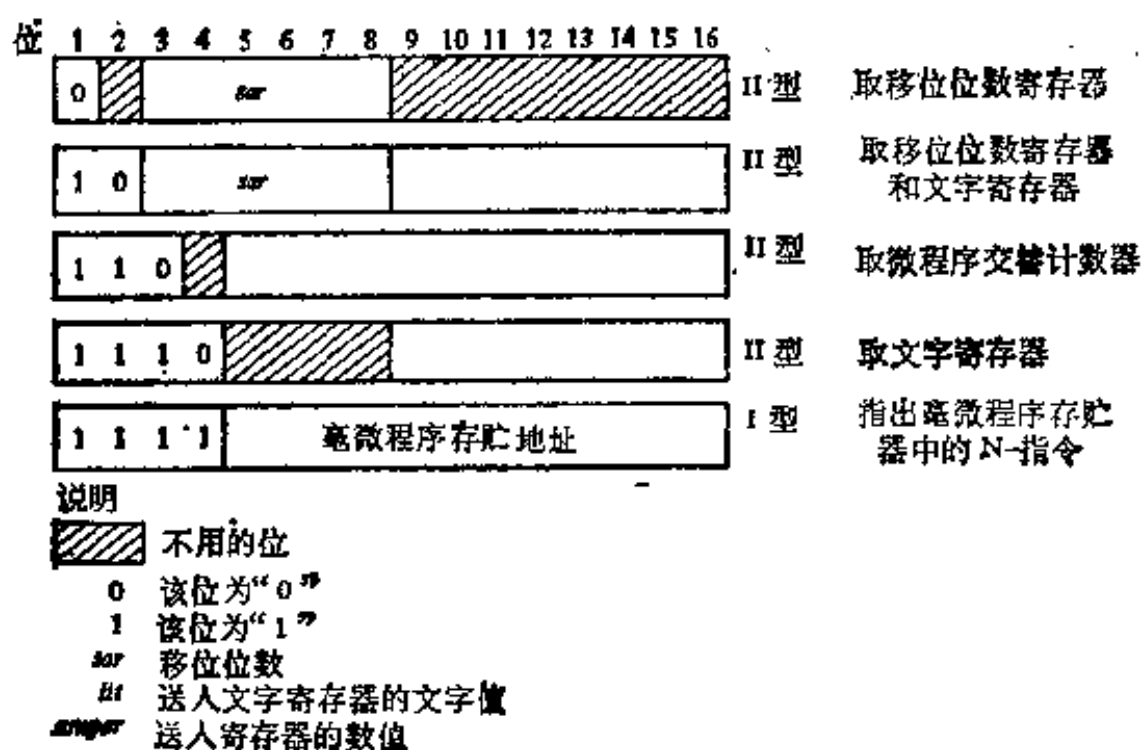


图 3-3 M 指令的格式

每条 M 指令 16 位长。我们已经定义了五种不同的 M 指令，这些指令及其格式如图 3-3 所示。其中有四种指令属 II 型，它们用来取 D 机器中的寄存器。第五种指令属 I 型，它指出毫微指令在毫微程序存储器中的地址。

3-3-2 机器周期

D 机器按照固定时间间隔(称为“时钟”)进行操作。在每一个

机器周期内,时钟信号将传送到整个系统,用它来选通各种门和开关以及改变信息状态。

在每个时钟周期(即在每个机器周期)内,从微程序存储器中读出一条 M 指令。如上所述,指令的前几位指出指令的类型。如果它是 II 型指令,则操作就在该时钟周期内完成,因为所有的操作数都包含在 M 指令中。如果它是 I 型指令,则在同一个周期内还要访问毫微存储器,并执行相应的毫微指令,为此还需增加一个第二周期。也即执行 I 型微指令时需要两个时钟周期;而 II 型微指令却只需一个时钟周期。第一个时钟周期称为执行微指令的第 1 相^{*}),而第二个时钟周期(如果它存在的话)则称为第 3 相^{**})。

I 型微指令中的第 3 相是与下一条微指令中的第 1 相重叠的,从而缩短了微指令的有效执行时间,使它实际所占时间仍只相当于一个时钟周期。关于第 1 相和第 3 相周期内, D 机器所完成的具体操作,将在补充介绍材料之后再叙述。

3-3-3 指令

在 D 机器操作时, I 型或 II 型指令是利用设置在 M 指令中的特征位的状态来区别的。这些指令是用微程序设计参考语言中的符号表示的。在进行微程序设计时,先将微程序写成 TRANSLANG 语言中的符号形式,然后用常规语言处理程序把它翻译成二进制编码的形式。在本书中给出的例子是使用参考语言,在计算机上运行的例子则使用 TRANSLANG 语言。由于 II 型指令容易描述,并且它能揭示参考语言的本质,所以首先介绍 II 型指令。

3-3-3-1 II 型指令

一条 II 型指令能加载 D 机器中三个不同的寄存器之一,并可取下列四种形式之一,

$$k \longrightarrow \text{SAR}$$

^{*}) “第 1 相”有时也称为“相位 1”。——译者注

^{**}) “第 3 相”有时也称为“相位 3”。——译者注

$$k \longrightarrow \text{LIT}$$
$$k \longrightarrow \text{AMPCR}$$
$$k_1 \longrightarrow \text{SAR}, k_2 \longrightarrow \text{LIT}$$

其中, k 是一个十进制正常数, 其数值大小适合于被加载的寄存器的宽度. 例如, 语句

$$5 \longrightarrow \text{LIT}$$
$$1 \longrightarrow \text{SAR}$$
$$100 \longrightarrow \text{AMPCR}$$

分别把数值 5, 1 和 100 送入文字寄存器 (LIT), 移位位数寄存器 (SAR) 和另一个微程序计数寄存器 (AMPCR). 在这些语句中, 关键字 LIT, SAR 和 AMPCR 是 D 机器中寄存器的名称, 而不是程序设计语言中的变量. 可以用单独一个 II 型语句同时将各种数据传送到 LIT 寄存器和 SAR 寄存器, 例如:

$$\text{COMP } 0 \longrightarrow \text{LIT}, 8 \longrightarrow \text{SAR}$$

这个语句把 0 的反码 (即 225) 送入 LIT 寄存器, 同时把数值 8 送入 SAR 寄存器. 而它前面的语句及对它的解释则说明: LIT 寄存器的宽度是 8 位 (这是正确的假定). 一般说来, 为了有效地利用处理机, 微程序员必须知道每个寄存器的宽度.

3-3-3-2 I 型

I 型指令控制 D 机器中逻辑部件、控制部件和存贮器控制部件的操作, 它总是指向毫微程序存贮器中的一条 54 位的 N 指令. 每条 N 指令能同时指出下列类型的操作:

检测条件

存贮器-设备的操作与条件的修正 (它可以是有条件的或无条件的)

逻辑部件、控制部件或存贮器控制部件的操作 (有条件的或无条件的)

例如, 语句

$$A1 + B \quad L \longrightarrow A2$$

是把 A1 和 B 寄存器的内容相加，把和数左移，并把结果送入 A2 寄存器。当执行该语句时，移位位数是受 SAR 中的内容控制的，再如语句

MW2, IF MST THEN A3 + B + 1 → A1 BEX, INC, SKIP ELSE JUMP

该语句指定完成下列操作：

1. 无条件执行对 S 存贮器的写操作 (MW2)，即将 D 机器中存贮器信息寄存器 (MIR) 的内容写入 S 存贮器的单元中；

2. 检测前一条指令中加法器输出的最高位 (MST)。如果该位为 1，则完成下列操作：

A3 + B + 1 → A1 A3 的内容加 B 的内容再加 1，替换 A1 原来的内容。

BEX 把一个字从外部数据总线送到 B 寄存器。

INC 存贮器控制部件中的计数器值加 1。

SKIP 后继指令(即下次要执行的指令)是现行指令地址加 2。因此，将跳过下一条指令。

若 MST 位为 0，则微程序转到 AMPCR 所指定的地址加 1，这是由关键字 JUMP 指示的。D 机器中的每个设备将在下面几节中详细叙述。前面所述的 II 型指令仅仅给出它在 D 机器中完成的操作类型的“特征”，而没有说明使用水平型毫微指令后所能得到的并行性。这里所说的并行性是指：能同时执行的基本的机器操作。显然，为了实现这种并行性将使微程序设计更加困难，但同时，它却使计算机的功能更强。一般，用一条毫微指令可以完成下列操作：

1. 检测一个条件；
2. 置位或者清除一个开关；
3. 启动一个外部读/写操作；
4. 在加法器中执行一次操作；
5. 对加法器操作的结果进行移位；

6. 把加法器操作与/或移位操作的结果存入一个或多个寄存器中去；
7. 计数器增量；
8. 在寄存器之间传送信息；
9. 对移位位数寄存器的内容取补；
10. 为现行指令选择后继指令。

在介绍了系统的各部件以后，D 机器中的各类操作将变得更有意义。

3-3-4 关于 D 机器说明的注释

D 机器并不是一台假想的计算机，它是 Burroughs 公司的产品。尽管这类机器具有若干种不同型号，但是本书所用的 D 机器的模拟程序却使用机器的一种特殊形式。例如，逻辑部件中操作寄存器的宽度为 32 位，移位位数寄存器 (SAR) 的宽度为 6 位，其他寄存器同样也有固定的宽度。作为可以在市场上出售的产品，D 机器逻辑部件的宽度可以是 64, 56, 48, 40, 32, 24, 16 或 8 位。实际的 D 机器还有其他一些特点，这里就不再叙述了，因为它已超出本书的范围。然而，在模拟程序和翻译程序中，已包括了与微程序设计过程最直接有关的 D 机器的主要特性。事实上，模拟程序和翻译程序包括了本书所提到的、但是未曾正式介绍过的那些方面。虽然，D 机器中那些未曾述及的硬件特征还将在后面的章节中列出，但不能把本书和有关的软件看作是对完整的 D 机器的精确描述。

3-4 逻辑部件

逻辑部件主要包含：加法器，三个 A 寄存器，B 寄存器，环移开关以及存贮器信息寄存器 (MIR)。逻辑部件中各设备之间的关系列在图 3-1 中。图中的箭头表示信息流，两个设备之间的箭头则表示：在 D 机器中可使信息按箭头所指的路径流动。如前所述，在

介绍D机器的某些部件之前,就可能要用到这些设备,这可参看逻辑部件图。但是,D机器中的主要功能和设备则将在本章介绍。

3-4-1 加法器

加法器字长32位,它有两个输入: X选择和Y选择。对一定的X输入和Y输入,加法器能够完成下列操作:

X

Y

$X + Y$

$X + Y + 1$

$X - Y$

$X - Y - 1$

NOT X

$X \oplus Y$

其中, \oplus 表示一个异或操作,它与AND, OR, NAND或者NOR一样都是逻辑操作。X输入与Y输入可以是输入加法器的数值,寄存器的信息或用其他方法构造的信息。

3-4-1-1 X和Y输入

加法器的输入值可分为如下几类:

A输入: A1、A2和A3寄存器。

B输入: 具有选择门的B寄存器(以后讨论)。

Z输入: AMPCR, CTR和LIT寄存器。

微程序设计的一个目标就是要选择对应于X和Y操作数的输入。

对X选择的输入可以取下面内容:

0

A1寄存器

A2寄存器

A3寄存器

CTR 寄存器

LIT 寄存器

或者, X 选择是空的, Y 选择的输入可以取下面内容:

0

1

B 寄存器

CTR 寄存器

AMPCR

另外,还可以在 Y 选择之前预置一个算符 NOT, 它表示在逻辑部件操作之前, Y 选择应该变成反码. 加法器操作的一些例子是:

$A2 + 1 \longrightarrow$

$A1 \text{ AND } LIT \longrightarrow$

$A3 - B - 1 \longrightarrow$

$\text{NOT } CTR \longrightarrow$

$AMPCR \longrightarrow$

$A2 \text{ EQV } B \longrightarrow$

$LIT + \text{NOT } CTR + 1 \longrightarrow$

应当注意: 这些加法器操作的例子是不完整的, 因为尚未指出加法器的结果输出到哪里去.

3-4-1-2 加法器输入

加法器的 A 类输入和 B 类输入必须分别使用 X 选择和 Y 选择. 每个寄存器的宽度都和加法器的宽度相同, 并且寄存器各位至加法器的输入都是一样的, 没有例外; B 寄存器的输出受门控制, 这在以后再讨论.

加法器的 Z 类输入则送到加法器的特定位上:

1. CTR 寄存器的输出送到加法器的最高 8 位, 其余各位皆为 0;
2. LIT 寄存器的输出送到加法器的最低 8 位, 其余各位皆为 0;

3. AMPCR 寄存器的输出送到加法器的最低 12 位, 其余各位皆为 0。

Z 类输入是逻辑部件的外输入信号。逻辑部件的外输入包括 Z 输入寄存器以及在控制部件和存储器控制部件中的若干其他寄存器。

3-4-1-3 加法器输出

加法器的输出可以送到环移开关(用于移位操作), 并且在加法器的输出到达环移开关以前, 也可以把它送到 B 寄存器。此时加法器到 B 寄存器的输出, 就作为 B 寄存器的输入。

3-4-1-4 加法器的空操作

在每个时钟周期内都可以完成一次加法器操作。如果在一条毫微指令中没有指定逻辑部件的操作, 则此时加法器完成下述操作:

$$0 + 0 \longrightarrow$$

这是一次没有指定要将结果送到那里去的“0 + 0”的加法器操作。因此, 任何一个与逻辑部件的操作有关的后继操作都可以成功地执行。

3-4-2 A 寄存器

D 机器有三个寄存器, 分别称为 A1, A2 和 A3。在微程序执行期间, 它们作为暂存器使用。它们也可作为加法器的初始输入。这三个寄存器中的任何一个都可作为加法器的 X 输入; 但是对于逻辑部件来说, 每执行一个操作只能取其中一个 A 寄存器作为输入。

取 A 寄存器的唯一的方法是: 把环移开关的输出作为 A 寄存器的输入, 如下面的例子:

$$A3 + 1 \quad L \longrightarrow A2$$

$$AMPCR \longrightarrow A1$$

在上述两个例子中,都用 A 寄存器来存放逻辑部件操作的结果。

后一例(即 $AMPCR \rightarrow A1$) 说明了微程序设计的一个特点,也说明了它与常规计算机程序设计的区别。在常规计算机程序设计中(只要知道一个寄存器的内容被送入另一个寄存器就够了)。在微程序设计中,微程序员还必须知道处理机各部件的信息通路。这也是为确定所完成的操作,以及微程序的优化、同步和有效地使用后继微指令所必须了解的信息通路。

3-4-3 B 寄存器

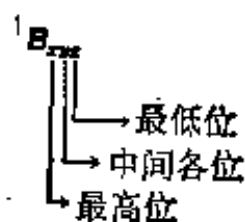
B 寄存器可以通过 Y 选择输入送到加法器,它也可以用作外输入的接口。B 寄存器还可以存放逻辑部件的操作结果。B 寄存器的宽度是 32 位,它与 A 寄存器和加法器的宽度是相同的。

3-4-3-1 B 寄存器的输出

B 寄存器的输出作为加法器的 Y 选择输入。但是, B 寄存器与 A 寄存器不同,它具有真/假/求补的选择门。这些选择门控制最高位(即最左的位)、最低位(即最右位)以及其余的中间位(即最高位与最低位之间的那些位)。当不指定选择门时, B 寄存器的内容就按真形式使用,有如下面的例子:

$$A1 + B \rightarrow AMPCR$$

门的选择是通过下标来指定的,下标被翻译成相应的毫微指令中的某些位。在 B 寄存器的三部分中,每一部分都可独立地选择,寄存器的说明采用下述形式:



B 寄存器控制门的选择是:

T 原内容

F	原内容的反码
0	0 输出
1	1 输出

例如,下表说明了某些 B 选择操作:

B 寄存器说明	意 义
B_{T00}	最高位是原内容,其余各位为 0
B_{001}	最低位是 1,其余各位为 0
B_{FIT}	最高位取反,其余各位是原内容
B_{000}	所有各位都为 0
B_{111}	所有各位都为 1
B_{FFF}	所有各位都取反
B_{TTT}	所有各位都是原内容,相当于 B 没有下标

B 寄存器的选择门的最大用处是生成常数、生成屏蔽值以及在 B 寄存器范围内选择特定的字段。

3-4-3-2 B 寄存器的输入

B 寄存器的输入可以来自外界,也可以来自逻辑部件本身.外界的输入是指来自 S 存贮器的读出信息,或者来自某一外部设备.但在本书中,不使用外部设备,因此,所有外界输入都来自 S 存贮器。

3-4-3-2-1 外界输入 此处只简要讨论来自 S 存贮器的外界输入,在下面的章节中还将更详细地讨论它。为了执行一次 S 存贮器的读出,必须完成下面几步操作:

1. 把 S 存贮器的地址送到存贮器控制部件;
2. 在逻辑部件中启动一次从 S 存贮器取一个字的读操作;
3. 当完成读操作后,把数据通过互锁开关传送到 B 寄存器。

作为例子,假定 A1 寄存器存放了 S 存贮器中一个字的地址。则下列 D 机器语句将把该字读到 B 寄存器:

A1 → MAR1 (1)

MR1 (2)

IF RDC THEN BEX ELSE WAIT (3)

语句(1)把 S 存贮器的地址送到存贮器地址寄存器, 语句(2)启动 S 存贮器的读操作, 而语句(3)检验读操作是否完成(即 RDC). 当读操作完成时(即 RDC 为“真”), 则 BEX 操作把来自 S 存贮器的字从互锁开关送到 B 寄存器. 对于 RDC 为“真”的情况, 后继指令就是下一条 M 指令. 对于 RDC 为“假”的情况, 后继指令就是语句(3)的重复. 后一过程一直持续到完成读操作为止.

3-4-3-2-2 来自逻辑部件的输入 B 寄存器可以按下列方式之一取逻辑部件的内容:

1. 来自环移开关的输出, 例如:

$A1 \text{ AND LIT } R \longrightarrow B$

2. 来自加法器的输出, 这是在它进入环移开关之前引出来的, 例如:

$A1 + \text{LIT } R \longrightarrow A2, \text{BAD}$

3. 来自存贮器信息寄存器 (MIR) 的输出, 例如:

$A1 + \text{LIT } R \longrightarrow A2, \text{BMI}$

4. 来自加法器、互锁开关, 或者 MIR 中某一个的输出与环移开关的逻辑“或”, 例如:

$A1 + \text{LIT } R \longrightarrow A2, \text{BBA} [\text{BSW} \vee \text{adder}]$

$A1 + \text{LIT } R \longrightarrow A2, \text{BBE} [\text{BSW} \vee \text{external}]$

$A1 + \text{LIT } R \longrightarrow A2, \text{BBI} [\text{BSW} \vee \text{MIR}]$

在所有这些情况下, 当 B 寄存器作为逻辑部件时钟周期的结果而被加载时, 它就是一个目标寄存器. 在第 1 种情况下, B 寄存器作为环移开关的输出目标, 直接参与加法器/环移开关(BSW)的操作. 第 2、3 和第 4 种情况中, B-寄存器的是单独被加载, 但可能与并列执行的加法器/BSW 的操作有关. B 寄存器既可以参与加法器的操作, 又可以是一个寄存器. 如

$A1 + B \longrightarrow A2, \text{BMI}$

在这个例子中, 加法器的操作使用了 B 的原内容. 当执行了上述

语句后, MIR 寄存器的原内容取代了 B 寄存器的内容。

3-4-4 环移开关

环移开关提供了把一个字进行右移或者左移的能力, 移位的位数由移位位数寄存器 (SAR) 指定。环移开关的输入来自加法器。有三种移位方式:

- L 左移
- R 右移
- C 循环右移

环移开关的输出可以送到下列任何一个寄存器: A1, A2, A3, B, MIR, BR1, BR2, MAR, CTR, SAR 以及 AMPCR。A1, A2, A3, B 以及 MIR 寄存器都接收环移开关的全部输出。BR1 和 BR2 寄存器只接收环移开关的次最低字节, 而这时 MAR, CTR 和 SAR 接收最低字节。AMPCR 接收环移开关输出的最低 12 位。

不管是否执行移位操作, 加法器的输出总是通过环移开关送到指定的目标寄存器。若语句中规定不移位, 则翻译程序自动地把一个无移位型的指示符填入毫微指令中。

3-4-5 存贮器信息寄存器

存贮器信息寄存器 (MIR) 作为写入 S 存贮器中或外部设备中的字的缓冲器。MIR 与第二章所述的存贮器缓冲寄存器的功能相同。

MIR 总是作为环移开关的目标寄存器, 如下例:

A1 → MIR

写入 S 存贮器的过程包括四个步骤:

1. 把要写入的数据传送到 MIR;
2. 在存贮器控制部件中设置 S 存贮器的地址;
3. 启动写操作;
4. 检验互锁开关是否已接收了信息。

假定寄存器 A3 存放着 S 存储器的地址,而寄存器 A1 存放着要写入该地址的一个字. 那么,下列语句将完成所指定的写操作:

A3 \longrightarrow MAR1 (1)

A1 \longrightarrow MIR (2)

MW2 (3)

IF SAI THEN STEP ELSE WAIT (4)

语句(1)和(2)分别设置 S 存储器的地址和把所要写入的字送入 MIR. 语句(3)启动写操作,在“开关接收信息”(SAI)的指示器置位前,语句(4)使 D 机器处于等待状态. 如前所述,也可以通过使用关键字 BMI 来把 MIR 的内容送入 B 寄存器.

3-4-6 关于逻辑部件操作的注释

逻辑部件操作的语法和语义将在下面的章节中给出. 然而,在进行详细说明之前,必须介绍 D 机器的其他主要部件,给出毫微指令格式的各个字段,给出微程序设计的有意义的例子以及介绍如何使用翻译程序和模拟程序.

3-5 存储器控制部件

存储器控制部件提供微程序的寻址、S 存储器和外部设备的寻址,并为逻辑部件中的加法器提供 Z 类输入. 在后一种情况下, Z 类输入包括 CTR 和 LIT 寄存器. 存储器控制部件和控制部件的简图如图 3-4 所示.

3-5-1 微程序的寻址

存储器控制部件有两个用于微程序存储器(MPM)寻址的寄存器.(显然,毫微程序存储器是通过 I 型指令来访问的.)微程序计数寄存器(MPCR)作为现行地址寄存器. 在执行 M 指令期间, MPCR 根据现行指令的后继指令的位置执行加 0、加 1 或者加 2 的操作.

微程序交替计数寄存器 (AMPCR) 在程序执行控制转移时与 MPCR 交替使用。AMPCR 可以输入到加法器, 也可以用作环移开关的输出目标寄存器。

可以对现行指令提供下列后继指令:

WAIT	重复现行指令。
STEP	顺序地执行下一条M指令。
SKIP	跳过下一条M指令, 继续执行。
SAVE	把现行地址 MPCR 保存到 AMPCR 中, 并且步进到下一条M指令。
CALL	使微程序转移到微程序存储器中地址为 AMPCR 加 1 (即 $(AMPCR) + 1$) 的M指令。而现行地址则保存到 AMPCR 中。
EXEC	将微程序跳到存储器中地址为 AMPCR 加 1 (即 $(AMPCR) + 1$) 的一条指令执行, 然后仍接着现行指令继续执行下去。
RETN	使控制转移到微程序存储器中地址为 AMPCR 加 2 (即 $(AMPCR) + 2$) 的M指令, 并从那条指令起继续执行。
JUMP	使控制转到微程序存储器中地址为 AMPCR 加 1 (即 $(AMPCR) + 1$) 的指令, 并从那条指令起继续执行。

一般地, 不指定真、假条件的后继指令就是 STEP。并且, 除 EXEC 外都把指示后继指令的新的微程序存储器地址放在 MPCR 中, II 型指令不影响指令的顺序, 并且它的后继指令就是 STEP。

3-5-2 S 存储器的寻址

S 存储器的寻址要用到存储控制部件中的三个寄存器 BR1, BR2 和 MAR。BR1 和 BR2 用作基址寄存器, 并且通常存有 S 存储器地址的次低字节。存储器地址寄存器 (MAR) 用作基址寄存器中的地址值的位移量, 并且存有 S 存储器地址的最低字节。

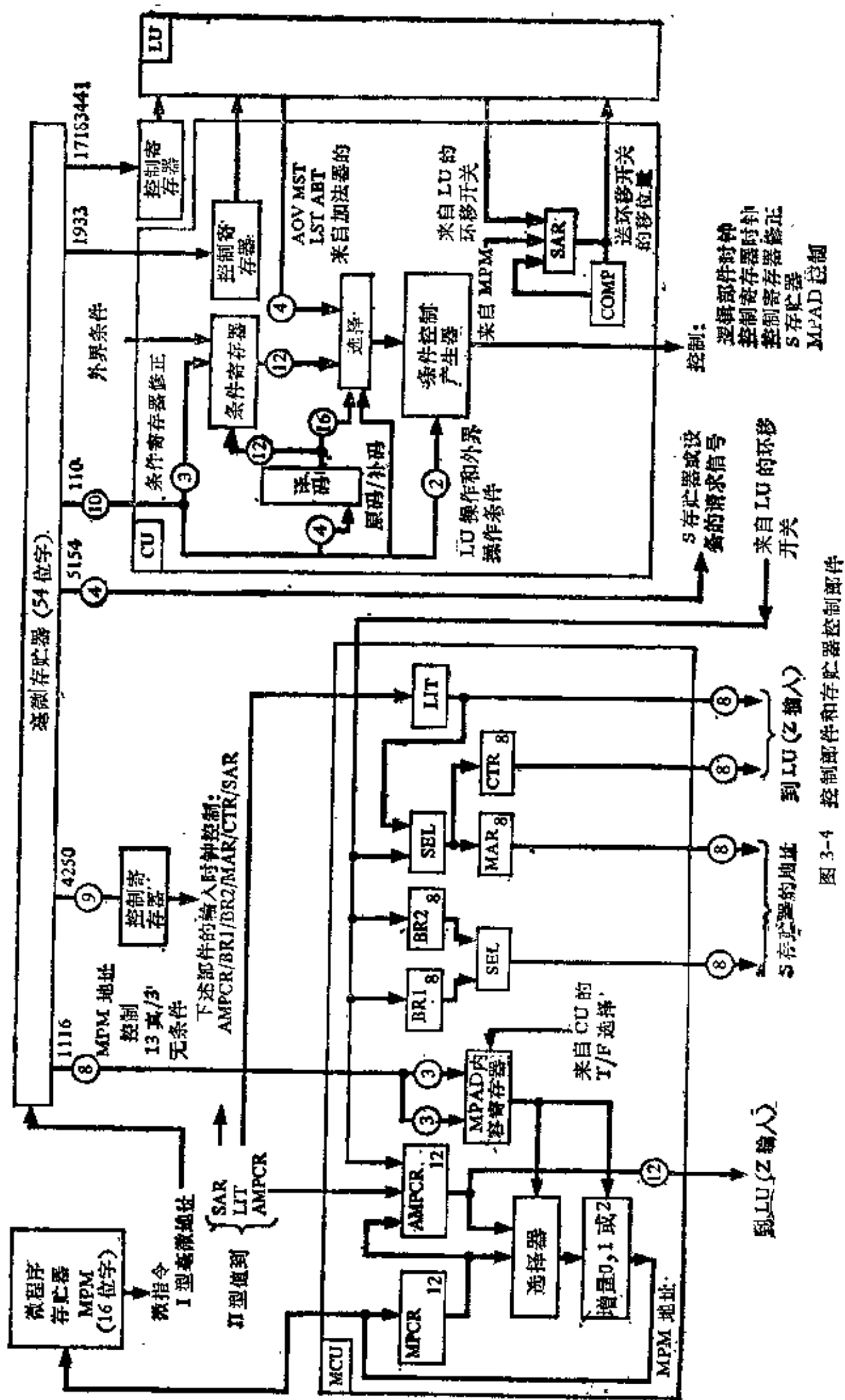
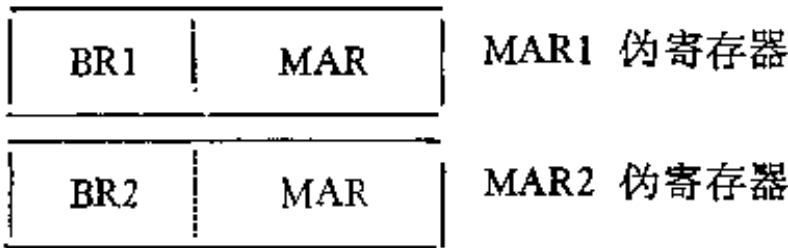


图 3-4 控制器部件和存储器部件

BR1, BR2 和 MAR 可分别从环移开关取信息, 或者通过按如下定义的伪寄存器 MAR1, MAR2 连起来取信息.



当 MAR1 和 MAR2 用作目标寄存器时, 整个 16 位寄存器(即把 BR1 或 BR2 与 MAR 连起来)全部从环移开关取信息, 则该寄存器包含有环移开关输出的最低 16 位. 当 BR1, BR2 和 MAR 分别从环移开关取信息时, BR1 和 BR2 接收环移开关输出的次低字节, 而 MAR 则接收环移开关输出的最低字节. S 存贮器的读、写操作按如下方式使用伪寄存器 MAR1 和 MAR2:

寄存器	S 存贮器读命令	S 存贮器写命令
MAR1	MR1	MW1
MAR2	* MR2	MW2

作为如何使用上述概念的例子, 考虑下面的问题:

假设 S 机器的寄存器保存在 S 存贮器中. BR1 指出 16 位地址寄存器中的高阶字节. A1 含有寄存器的编号, 它们可以是 0, 1, 2 等. 现要把寄存器 A3 的值加到由 A1 指定的 S 机器的寄存器中, 并将其结果重新写回 S 存贮器.

完成上述操作的一段简短的微程序如下:

```
A1→MAR
MR1
IF RDC THEN BEX ELSE WAIT
A3 + B→MIR
MW1
IF SAI THEN STEP ELSE WAIT
```

在这个例子中, BR1 应该预先放入指定的其地址值, 这通常是用一个初始子程序实现的. 另一方面, 假定寄存器 A2 含有一个有效地址, 要求把 A3 寄存器的内容与地址存贮单元的内容相

加,并把和数送回该单元。此时,相应的微程序如下:

```
A2→MAR1  
MR1  
IF RDC THEN BEX ELSE WAIT  
A3 + B→MIR  
MW1  
IF SAI THEN STEP ELSE WAIT
```

上述这两段微程序本质上是相同的,只是,前一种情况将地址送到 MAR,后一种情况将地址送到 MAR1.

存贮器地址寄存器 (MAR) 也可以通过 LMAR 命令直接从 LIT 寄存器取地址信息,而无需加法器操作.例如下面所示的微程序:

```
A1 + B→A3, LMAR  
3→LIT
```

这里,将 3 送到 MAR. 这是用 II 型指令送信息到寄存器的第一个例子,并且这个 II 型指令用在一条 I 型指令之后.由于 D 机器的时钟同步特性以及由于前面提到的相位 1 的重叠,在执行 LMAR 命令之前,已把 3 送到 LIT 寄存器了. 这是一个微程序设计的范例. 若将 3→LIT 放在 LMAR 命令之前,即

```
3→LIT  
A2 + B→A3, LMAR
```

则完成 $A2 + B \rightarrow A3$ 和 LMAR 命令将需要三个时钟周期. 如果将 3→LIT 放在 LMAR 命令之后,即

```
A2 + B→A3, LMAR  
3→LIT
```

则全部操作就只要二个时钟周期.

3-5-3 计数寄存器

计数寄存器 (CTR) 是存贮器控制部件中的一个 8 位计数器. CTR 有两种方式取信息: 作为环移开关的输出,如

$A3 \longrightarrow \text{CTR}$

或者,通过使用 LCTR 命令,直接从 LIT 寄存器取信息,而无需加法器操作。如下面所示的微程序:

$A1 + B \longrightarrow A3, \text{LTCR}$

$3 \longrightarrow \text{LIT}$

此时,把 3 的反码(即二进制值 11111100)送到 CTR 中。事实上,不管赋值的方法如何,CTR 总是取源信息字段的反码。

CTR 不能直接用 II 型指令取信息。

3-5-3-1 增量

CTR 可按下述方式用于加法器操作:

$A3 + \text{NOT CTR} \longrightarrow B$

每当需要 CTR 的真值时,必须使用 NOT CTR。通过 INC 命令,在不使用加法器时也可以使 CTR 增量,例如:

$A1 + B \longrightarrow A3, \text{INC}$

INC 命令总是使反码形式的计数器的值加真值 1 的。因此,在执行下面的语句

$A1 + B \longrightarrow A3, \text{LTCR}$

$3 \longrightarrow \text{LIT}$

INC

之后,CTR 将包含的二进制值为 11111101。

3-5-3-2 溢出

计数寄存器从全“1”(即十进制 255)变到全“0”就是溢出,这时,计数器的溢出指示器被置位。可以利用计数器溢出(COV)指示器来测试 CTR,如

INC

⋮

IF COV THEN...

作为 CTR 怎样操作的一个例子,考虑

$A1 + B \longrightarrow A3, LTR$
 $3 \longrightarrow LIT$
 \dots
 $INC \dots$
 $IF\ COV\ THEN \dots$

最初,把 3 的反码送到 CTR, CTR 逐次增量后的内容如下:

	CTR 值
初始值:	11111100
第一次增量后:	11111101
第二次增量后:	11111110
第三次增量后:	11111111
第四次增量后:	00000000 “溢出”

计数器溢出 (COV) 条件的测试,可以在计数器增量 (INC) 命令之后进行,也可以同该 INC 命令同时进行. 溢出条件指出计数器已经溢出了,或者正处在溢出的过程中.

3-5-4 文字寄存器

文字寄存器 (LIT) 是存储器控制部件中的一个 8 位寄存器. 文字寄存器只能用 II 型指令取信息,它可以给微程序提供常数. 例如:

$9 \longrightarrow LIT$
 $COMP\ 0 \longrightarrow LIT, 15 \longrightarrow SAR$

LIT 寄存器可以作为加法器的输入,也可以作为 LCTR 和 LMAR 命令的源寄存器.

3-6 控 制 器

控制器包括四部分,它们协同控制 D 机器的操作. 这四部分是: 移位系统,条件系统,命令系统以及时钟系统. 控制器的主要部分如图 3.4 所示. 时钟系统已简要地介绍过,在下面章节中还将

更详细地讨论它。

3-6-1 移位系统

移位位数寄存器 (SAR) 和环移开关合起来操作, 可以提供算术移位或循环移位的能力。对于右移操作, 则把移位的位数送

表 3.1 32 位机器的移位位数及其补码 (×位为硬件所忽略)

SAR 的内容		
移位 位数	右 移 (原码位数)	左 移 (补码位数)
0	000X00	000X00
1	01	111X11
2	10	10
3	11	01
4	001X00	00
5	01	110X11
6	10	10
7	11	01
8	010X00	00
9	01	101X11
10	10	10
11	11	01
12	011X00	00
13	01	100X11
14	10	10
15	11	01
16	100X00	00
17	01	011X11
18	10	10
19	11	01
20	101X00	00
21	01	010X11
22	10	10
23	11	01
24	110X00	00
25	01	001X11
26	10	10
27	11	01
28	111X00	00
29	01	000X11
30	10	10
31	11	01

入 SAR 中。对于左移操作，则把移位位数的补码送入 SAR 中。
II 型指令产生的移位位数的补码是由翻译程序给出的，如：

II 型指令	所完成的动作
3—→SAR	原码值 3，用于控制右移和循环移位
COMP 5—→SAR	5 的补码，用于控制左移

然而，当把 SAR 用作目标寄存器时，就要求更仔细地检测 SAR 的内容。

3-6-1-1 移位位数及其补码

正如本章前面所述，D 机器是针对不同宽度的逻辑部件设计的，这一灵活性反映在移位位数寄存器 (SAR) 中。对于一台 32 位的机器而言，移位位数及其补码如表 3-1 所示。例如，右移 6 位要求把下面的值送入 SAR 中：

$$001 \times 10$$

其中，×位为硬件所忽略。类似地，左移 3 位要求把下面的值送入 SAR 中：

$$111 \times 01$$

其中，×位为硬件所忽略。在前一种情况中，微程序员写语句：
6→SAR，而语言处理程序则产生修正过的 SAR 值。在后一种情况中，微程序员写语句：COMP 3—→SAR，而语言处理程序则计算 3 的补码，并产生修正过的 SAR 值。

当 SAR 从环移开关取信息，并把它作为目标操作符时，移位位数并未自动修正。此时，如果在移位位数送入 SAR 之前，不把 ×位插入到移位位数中去的话，则会得到错误的结果。

3-6-1-2 SLIT

如果要把 LIT 的内容传送到 SAR，那么，借助于使用关键字 SLIT，就可以避免“将 ×位插到移位位数中去”的动作。例如，若要求右移 5 位，则下面列出相应的正确的和错误的程序段：

错误的程序

正确的程序

5—→LIT

...

LIT—→SAR

5—→SLIT

...

LIT—→SAR

关键字 SLIT 表示：把移位位数记入 LIT 寄存器，并把正确的移位值置于文字赋值指令中。

3-6-1-3 CSAR

CSAR 命令提供了又一种能对 SAR 的内容求补无需用加法和环移开关的手段。例如，用 CSAR 控制目标寄存器时：

A1 + B—→A2, CSAR

这里，有必要给求补作个注释。对于 SAR 和 SLIT 寄存器的求补操作符，例如

COMP 3—→SAR

COMP 7—→SLIT

总是执行求补。而对于 AMPCR 和 LIT 寄存器的求补操作符，例如，

COMP—→AMPCR

COMP 0—→LIT

则总是执行求反。

3-6-2 条件系统

条件系统根据条件执行命令，在同一语句中，可以包含条件操作和无条件操作，并且一个语句的条件部分既可以使用 IF，也可以使用 WHEN。条件可分为静态条件和动态条件两类。

3-6-2-1 静态条件

静态条件是在机器操作的过程中建立的，而且通过测试来清除该条件。静态条件包括：

SAI 互锁开关接收了信息。这个条件与写操作一起使用时表示：在逻辑上该操作已经完成了。

RDC	读完成。这个条件与读操作一起使用时这个条件表示：
COV	已经可以用 BEX 命令把读出数据送入 B 寄存器。计数器溢出。当 CTR 由全“1”变成全“0”而溢出时，就建立这个条件。
LC1	局部条件 1 } 局部条件1到3提供布尔条件，它们 分别用命令 SET LC 1, SET LC 2, SET LC 3 来建立。
LC2	
LC3	

SAI, RDC 和 COV 条件是作为执行其他命令的结果而隐含地建立的。LC1, LC2 和 LC3 条件是用 SET 命令直接建立的。一旦建立了静态条件，不管这中间插入多少语句，它一直保持着，直到它被测试为止。

3-6-2-2 动态条件

动态条件是由使用逻辑部件的前一条指令根据相位 3 的加法器输出而动态地建立的。动态条件只保持到下一次逻辑操作它们被动态改变为止。动态条件包括：

AOV	加法器溢出。这个条件是在加法器的操作中，用其最高位的进位建立的。
LST	最低位。当加法器的最低位输出是“1”时，就建立这个条件。若最低位是“0”，则不建立这个条件。
MST	最高位。加法器最高位输出为“1”时，建立这个条件。最高位为“0”时，则不建立这个条件。
ABT	所有位皆 1。当加法器的所有输出位都是“1”时，就建立这个条件。若至少有一位输出为“0”，则不建立这个条件。

一个语句中建立的动态条件必须在执行下个 I 型语句中检测。

在所有静态或动态条件之前都可冠以 NOT，它表示：在语句的条件部分被执行之前，要把条件取反。

3-6-2-3 IF 语句

IF 语句的一般格式为:

IF 条件 THEN 对应于真的后继指令, ELSE 对应于假
 的后继指令

其中,“条件”是前面所说的条件之一. 下面举出几个例子:

IF COV THEN $A1 + B \longrightarrow A2$, BEX, SET LC1, LCTR ELSE JUMP

IF NOT ABT THEN SKIP ELSE STEP

IF LC1 THEN $A1 \longrightarrow$ MIR, LMAR ELSE RETN

在以后要详细讨论的另一个课题中, 将涉及到信息在目标寄存器中的实际放法. 在一个 I 型语句中, 例如

$A2 + 1 \longrightarrow A2$

则在遇到下一个 I 型语句中的逻辑部件操作之前, 加法器/环移开关操作的结果实际上并没有放到寄存器 A2. 让我们来考察下面的一段微程序, 其中对 AOV 条件测试的结果为“假”:

$B_{000} \longrightarrow A2$ (1)

$A2 + 1 \longrightarrow A2$ (2)

IF AOV THEN $A2 \longrightarrow$ MIR, INC, STEP ELSE STEP (3)

$A2 + 1 \longrightarrow A3$ (4)

STEP (5)

在语句(1)中把值 0 赋给 A2. 在语句(2)中, 1 加 A2 的内容, 但在下一次逻辑部件操作之前, A2 不变. 所以, 在语句(3)中, MIR 的内容不变, CTR 也未增量. 然而在执行语句(4)之前, 语句(2)中的目标操作符 A2 修改了. 因为语句(4)隐含指出的是一个假条件, 所以将导致执行一个逻辑部件操作: 1 加上 A2 的内容, 并把和送入 A3. 语句(5)是一个“空操作”语句, 它使语句(4)得以完成.

静态条件是借助于测试条件来清除的, 可是在一个没有 THEN 子句和 ELSE 子句的 IF 子句中却能在没有指定条件操作时去清除条件. 例如, 在语句

$A1 + B \longrightarrow A2, \text{IF } LC1$

中, 加法器/环移开关的操作是无条件完成的, $LC1$ 被清除, 并隐含地指定后继指令是 $STEP$ 。这种方式提供了一种在无需使用单独语句的情况下去清除一个条件的方法。

3-6-2-4 WHEN 语句

WHEN 语句是 IF 语句的一种简化形式, 考察下述形式的一个 IF 语句:

IF 条件 THEN 命令 STEP ELSE WAIT

这种特殊形式的语句经常结合存贮器的读和写操作来使用, 并且可以用下述简化形式的语句来代替:

WHEN 条件 THEN 命令

这两个语句在操作上是等价的。当条件成立(即“真”)时, 后继指令是 $STEP$; 当条件不成立(即“假”)时, 后继指令是 $WAIT$ 。下面的语句说明 WHEN 语句的用法:

WHEN RDC THEN BEX, INC

在一个语句中, 关键字 WHEN 可以随意加在无条件的一个或多个命令之前。

3-6-3 命令系统

在把一条 I 型指令翻译成一条 54 位的毫微指令时, 每个命令是借助于把毫微指令中的一位或多位置成 1 的方式来表示的。特殊位的置 1 及其意义将在以后讨论。进行微程序设计并不需要知道毫微指令的格式, 但是, 为了完全理解微程序设计的过程, 还是应当了解这种格式。

I 型指令的一般格式如图 3-5 所示, 这个图的目的需要说明: 可以无条件地或者有条件地执行 I 型指令。在图 3-5 中, 建立条件和完成外界操作的命令列为 A 组, 而逻辑部件, 存贮器控制部件以及控制部件的命令列为 B 组。由于在毫微指令中某些附加位非常重要, 而条件/无条件执行的命令或者适合于集中到 A 组, 或者

适合于集中到B组。因此,例如不可以写:

$A1 + B \longrightarrow A2$, IF ABT THEN INC

这是因为, $A1 + B \longrightarrow A2$ 与 INC 命令两者都是B组命令。然而,可以这样写:

$A1 + B \longrightarrow A2$, IF ABT THEN MR1 ELSE STEP

这是因为, $A1 + B \longrightarrow A2$ 是一个B组的命令,而 MR1 是一个外界操作。

来自毫微指令(它代表要完成的命令)的控制信号存贮在控制部件的一个36位的控制寄存器中。所以,当由于时钟同步而需要重复执行一条指令时,就很容易利用这种控制信息。

3-7 结 束 语

本章叙述了有关微程序设计和D-机器的技术资料。显然,我们希望并且需要有一组恰当的例子(这安排在下一章)。这样组织主题资料,使读者可以把本章用于查阅,而下一章则可作为例子来应用。

词 汇

读者应当熟悉在本章中用过的下列术语和缩写词:

术语

垂直型微程序设计 (Vertical microprogramming)

水平型微程序设计 (Horizontal microprogramming)

D 机器 (D machine)

逻辑部件 (Logic unit)

控制部件 (Control unit)

存贮控制部件 (Memory control unit)

微程序存贮器 (Microprogram memory)

毫微程序存贮器 (Nanoprogram memory)

S 存贮器 (S Memory)

控制存贮器 (Control storage)

M指令 (M instruction)
N指令 (N instruction)
时钟 (Clock)
相位 1 (Phase 1)
相位 3 (Phase 3)
I 型指令 (Type I instruction)
II 型指令 (Type II instruction)
条件 (Condition)
外界操作 (External operation)
加法器 (Adder)
X 选择输入 (X select input)
Y 选择输入 (Y select input)
A 输入 (A input)
B 输入 (B input)
Z 输入 (Z input)
环移开关 (Barrel switch)
前缀操作符 (Prefix Operator)
求补操作符 (Complement Operator)
反码 (One's complement)
补码 (Two's complement)
后继指令 (Successor)
目标 (Destination)
修正条件操作 (Condition-adjust operation)
门控制(或选择门) (Gating)
移位位数寄存器 (Shift amount register)
计数寄存器 (Counter register)
存储器信息寄存器 (Memory information register)
存储器地址寄存器 (Memory address register)
文字寄存器 (Literal register)
计数器溢出 (Counter overflow)
最低位 (Least significant bit)
最高位 (Most significant bit)
静态条件 (Static condition)

动态条件 (Dynamic condition)

命令 (Command)

提 问

1. 就你所知的关于计算机和模拟的知识, 把使用模拟程序的计算机与使用一台实际可编微程序设计的计算机作个比较. 说明模拟程序具有什么优点?
2. 第一章中介绍过的可微程序设计的假想计算机是水平型微程序机器还是垂直型微程序机器?
3. 说出 D 机器中两个控制存贮器的名称.
4. 说出 D 机器中三种主要硬件子系统的名称. 在一般情况下, 每个子系统完成什么功能?
5. M 指令的宽度(以位计)有多长? N 指令呢?
6. 下面这类 II 指令 “5 → CTR” 有什么错误?
7. 在 D 机器中, 微指令的重叠执行取什么方式?
8. 衡量毫微指令效率的一种有效方法是什么?
9. 试列出逻辑部件和存贮器控制部件中的寄存器.
10. 使用下列加法器语句有什么错误?

$B + A1 \rightarrow MIR$

$1 + B \rightarrow A1$

11. 说明下述指令操作的结果

$CTR + 1 \rightarrow A1$

12. 给出等价于下述寄存器的数值

$B_{002}, B_{005}, B_{111}, B_{115}, B_{100}$

13. 说明 MPCR 寄存器与 AMPCR 寄存器之间的差别.
14. 用什么条件检验 MR1 的操作? 检验 MW1 操作的条件又是什么?
15. 可以放入寄存器的最大文字值是何数量?
16. 用什么方法可以确定 D 机器的 MPM 有多大? 它究竟有多大?
17. MAR2 寄存器是由哪些寄存器连结而成的?
18. 说明目标与后继指令之间的区别?
19. 试列出把一个字写入 S-存贮器所需要的各个步骤.
20. 在执行了下述 II 型指令以后

$5 \rightarrow SAR$

SAR 的内容将变成什么?

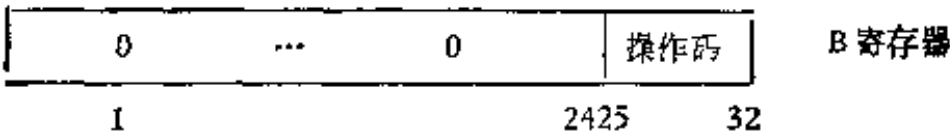
- 21. 说明静态与动态条件之间的区别。
- 22. 给出与下面 WHEW 语句等价的 IF 语句

$$\text{WHEN RDC THEN BEX}$$
- 23. 在什么情况下, SKIP 与 RETN 的继指令是相同的?
- 24. 下述语句完成什么样的简单操作?

$$A1 + \text{NOT}0 \longrightarrow A1$$
- 25. 给出与加法器输出有关的条件的名称?

习 题

- 1. 写出完成下列操作的语句:
 - (a) 把 5 送到 LIT 寄存器, 12 送到 SAR 寄存器。
 - (b) 检验加法器输出的最高位, 如果该位是“1”, 则执行下一语句; 否则转向 AMPCR 的内容加 1。
 - (c) 把 A1 寄存器的内容求反, 并把结果送入 B 寄存器。
- 2. 写出完成下列操作的微程序段:
 - (a) A1 寄存器与 B 寄存器的值互换。
 - (b) 把 S 存储器的 123 号单元的内容传送到 321 号单元。
 - (c) 寄存器 BR1 指示 S 存储器中的 16 个定点寄存器。把寄存器 3 的内容与寄存器 2 的内容相加, 并把结果送入寄存器 2。
- 3. 寄存器 A3 存放着一条 S 机器指令, 其中指令的前 8 位是操作码。要求把操作码移入 B 寄存器的右边, 即:



- 试写出完成这一数据处理的一个语句或一段程序。
- 4. 试写出交换 B 寄存器与 MIR 寄存器的值的一个语句或一段程序。
 - 5. 试写出用以把 A1 寄存器的内容与 A2 寄存器的内容相加, 并把结果送入 A3 寄存器的一个语句或一段程序。

第四章 微程序操作

4-1 概 述

第三章的任务是介绍微程序计算机,并且讨论了微指令的结构.本章介绍微程序中通常所包含的微程序操作类型以及毫微指令编码的表示方法.本章还介绍了若干个微程序设计方法的例子.

4-1-1 程序结构

微程序是由一系列语句组成的,这些语句规定了一组基本的机器操作.在微程序中,唯一必不可少的语句是 END 语句,它是翻译程序结束处理操作所需要的.翻译程序为 END 语句产生一条“4000”(十六进制)的 M 指令,这条 M 指令被模拟程序认作一个“停止”命令.

下面的微程序把数值 1234 写入 S 存储器中的 1234 号单元:

```
AMPCR→MAR1, MIR  
1234→AMPCR  
MW1, IF SAI  
WHEN SAI THEN STEP  
END
```

尽管这个微程序并没有实现特殊的功能,它却是一个完整的微程序.这个程序依靠翻译程序/模拟程序而运行,其打印的输出结果如图 4-1 所示.虽然尚未讨论过翻译程序和模拟程序,但很明显它是一种有用的工具.【参考语言和机器可识别语法之间的最大差别是,用一个等号(=)代替表示替换的箭头(→),并且每个语句必须用一个美元符号(\$)表示结束.】

a 翻译

>>>> TRANSLANG D MACHINE MICROTRANSLATOR

ENTER FILENAME FOR HEX? F41HEX
SUPPRESS BIT PATTERNS?(1=SUPPRESS)? 1
SUPPRESS HEX LISTING?(1=SUPPRESS)? 1
IF INPUT IN FILE ENTER 17 31
ENTER SOURCE FILENAME? FIG41

0000 AMPCR = MAR1, MIR 5
0001 1234 = AMPCR 5
0002 MW1, IF SAI 5
0003 WHEN SAI THEN STEP 3
0004 END 5

THE TOTAL NUMBER OF ERRORS = 0

EXECUTE (Y/N)? N

b 执行

DNACH1 (COMPILED) 05 JAN 76 16:29

ENTER SAME FILENAME FOR HEX? F41HEX
OUTPUT REGISTERS AND S MEMORY IN INTEGER(1) OR OCTAL(2)? 1
INPUT S MEMORY IN INTEGER(1) OR OCTAL IN OII FORMAT(2)? 1
STARTING ADDRESS =? 0
MAXIMUM NUMBER OF CLOCKS TO SIMULATE=? 10
NUMBER OF CLOCKS BETWEEN OUTPUT POINTS=? 1

ENTER OUTPUT LINES DESIRED 1-ADDRESSES AND CLOCK
2- A1,A2,A3,B 3- MIR,SAR,LIT,CTR,AMPCR 4- BR1,BR2,MAR,BMAR,GC1,GC2
5- CONDITIONS
ENTER NUMBER OF OUTPUT LINES DESIRED? 5

BEGIN OUTPUT AT MPM ADDRESS=? 4
END OUTPUT AT MPM ADDRESS=? 4
ENTER 1 FOR S MEMORY DUMP WHEN PROGRAM TERMINATES? 1

ENTER S MEMORY VALUES IN CONSECUTIVE BLOCKS
ENTER 9999 FOR STARTING ADDRESS WHEN FINISHED

STARTING S MEMORY ADDRESS=? 1234
FINAL S MEMORY ADDRESS FOR THIS BLOCK=? 1234
SMEM(1234)=? 0

STARTING S MEMORY ADDRESS=? 9999

END OF SIMULATION - REGISTERS CONTAIN

P(1) ADDR. = 3 P(3) ADDR. = 3 CLOCK = 6
A1 = 0 A2 = 0 A3 = 0 B = 0
MIR = 1234 SAR = 0 LIT = 0 CTR = 0 AMPCR = 1234
BR1 = 4 BR2 = 0 MAR = 210 BMAR = 1234 GC1=0 GC2=0
LC1=0 LC2=0 MS1=0 LS1=0 A&T=0 A&V=0 C&V=0 SAI=1 RDC=0 INT=0

```

MEMORY DUMP REQUESTED

ENTER VALUES AS DONE IN MEMORY INPUT (9999 FOR STARTING ADDRESS WHEN
FINISHED)

STARTING S MEMORY ADDRESS=? 1234
FINAL S MEMORY ADDRESS FOR THIS BLOCK=? 1234

S MEMORY(1234) TO S MEMORY(1234) =

1234
STARTING S MEMORY ADDRESS=? 9999

```

图 4-1 利用翻译程序和模拟程序执行一个完整微程序的例子

4-1-2 语句标号

使用语句标号可以访问 II 型指令中的 MPM 地址。在 II 型指令中，语句标号用来取 AMPCR，以供后面的 I 型指令（它把 AMPCR 作为下一个 MPM 地址的源）使用。在引用 AMPCR 以前，它必须先放进内容，例如：

标号 1-1 → AMPCR
 :
 :
 ————, JUMP

JUMP 命令使微程序控制转移到 AMPCR 的内容加 1，用符号表示，就是

$$(\text{AMPCR}) + 1$$

例如，下面的微程序把全“1”写入 S 存贮器的前 10 个单元中：

- | | |
|------------------------|-----|
| 0 → A1, LCTR | (1) |
| 9 → LIT | (2) |
| RPT - 1 → AMPCR | (3) |
| NOT 0 → MIR, INC | (4) |
| RPT: A1 + 1 → A1, MAR1 | (5) |
| MW1, IF SAI | (6) |
| WHEN SAI THEN STEP | (7) |

a 翻译

>>>> TRANSLANG D MACHINE MICROTRANSLATOR

ENTER FILENAME FOR HEX? F40HEX
SUPPRESS BIT PATTERNS?(1=SUPPRESS)? 1
SUPPRESS HEX LISTING?(1=SUPPRESS)? 1
IF INPUT IN FILE ENTER 1? 31
ENTER SOURCE FILENAME? FIG42

0000 0 = A1, LCTR \$
0001 9 = LIT \$
0002 RPT - 1 = AMPCR \$
0003 NOT 0 = MIR, INC \$
0004 RPT. A1 + 1 = A1, MAR1 \$
0005 MW1, IF SAI \$
0006 WHEN SAI THEN STEP \$
0007 IF NOT C0V THEN INC. JUMP ELSE STEP \$
0008 END \$

THE TOTAL NUMBER OF ERRORS = 0

EXECUTE (Y/N)? N

b 执行

B) DNACHI (COMPILED) 05 JAN 76 17:10

ENTER SAME FILENAME FOR HEX? F40HEX
OUTPUT REGISTERS AND S MEMORY IN INTEGER(1) OR OCTAL(2)? 2
INPUT S MEMORY IN INTEGER(1) OR OCTAL IN 011 FORMAT(2)? 2
STARTING ADDRESS =? 0
MAXIMUM NUMBER OF CLOCKS TO SIMULATE=? 75
NUMBER OF CLOCKS BETWEEN OUTPUT POINTS=? 1

ENTER OUTPUT LINES DESIRED 1-ADDRESSES AND CLOCK
2- A1,A2,A3,B 3- MIR,SAR,LIT,CTR,AMPCR 4- BR1,BR2,MAR,BMAR,GC1,GC2
5- CONDITIONS

ENTER NUMBER OF OUTPUT LINES DESIRED? 5

BEGIN OUTPUT AT MPH ADDRESS=? 8
END OUTPUT AT MPH ADDRESS=? 8
ENTER 1 FOR S MEMORY DUMP WHEN PROGRAM TERMINATES? 1

ENTER S MEMORY VALUES IN CONSECUTIVE BLOCKS
ENTER 9999 FOR STARTING ADDRESS WHEN FINISHED

STARTING S MEMORY ADDRESS=? 9999

END OF SIMULATION - REGISTERS CONTAIN

P(1) ADDR. = 7 P(2) ADDR. = 6 CLOCK = 55
A1=000000000012 A2=000000000000 A3=000000000000 B =000000000000
MIR =3177777777 SAR = 0 LIT = 9 CTR = 0 AMPCR = 3
BR1 = 0 BR2 = 0 MAR = 10 BMAR = 10 GC1=0 GC2=0
LC1=0 LC2=0 MST=0 LST=0 ABT=0 ABV=0 C0V=0 SAI=1 RDC=0 INT=0

```

MEMORY DUMP REQUESTED

ENTER VALUES AS DONE IN MEMORY INPUT (9999 FOR STARTING ADDRESS WHEN
FINISHED)

STARTING S MEMORY ADDRESS=? 1
FINAL S MEMORY ADDRESS FOR THIS BLOCK=? 15

S MEMORY( 1) TO S MEMORY( 15) =

3777777777 3777777777 3777777777 3777777777 3777777777
3777777777 3777777777 3777777777 3777777777 3777777777
0000000000 0000000000 0000000000 0000000000 0000000000

STARTING S MEMORY ADDRESS=? 9999

```

图 4-2 描述循环和语句标号用法的微程序

```

IF NOT COV THEN INC, JUMP ELSE STEP (8)
END (9)

```

寄存器 A1 用来保存 S 存储器地址中的当前值，而计数寄存器 (CTR) 则用来记录循环的重复次数。语句 (1) 完成置初值的功能：把值“0”送入 A1，并把经语句 (2) 置初值的 LIT 寄存器的内容送入 CTR。这里要记住：送入 CTR 的，是 LIT 的反码。语句 (3) 把标号 RPT 的地址值减 1 后送入 AMPCR。于是，转移地址将是 RPT。语句 (4) 把全 1 送入 MIR，并使 CTR 的值增加 1。语句 (5) 使 A1 加 1，并把结果值送入 MAR1 寄存器，它用作 S 存储器的地址。这里要记住：MAR1 是寄存器 BR1 和 MAR 的组合。语句 (6) 用 MW1 命令启动存储器写操作，并通过测试来清除 SAI。语句 (7) 在存储器写操作被互锁开关接受信息之前，将一直等待着，在它写入信息后则步进到下一条指令。应当注意：MW1 命令规定了 MAR1 将用作 S 存储器的地址。如果是用 MW2 命令，则规定 MAR2 将用作 S 存储器的地址，MAR2 是 BR2 和 MAR 的组合。语句 (8) 测试 CTR 是否溢出。如果 CTR 不溢出，则 CTR 的值加 1，并转到 RPT。如果 CTR 溢出，则控制转到语句 (9)，结束该微程序。

初编的微程序	在作了某些改进以后	最后的形式
<pre> 0→A1 LIT→CTR 9→LIT RPT: A1 + 1→A1 INC FIN - 1→AMPCR IF COV THEN JUMP ELSE STEP A1→MARI MW1, IF SAI RPT - 1→AMPCR WHEN SAI THEN JUMP FIN: STEP END </pre>	<pre> 0→A1 LIT→CTR 9→LIT RPT - 1→AMPCR NOT 0→MIR INC RPT: A1 + 1→A1, MARI MW1, IF SAI WHEN SAI THEN STEP IF COV THEN SKIP ELSE STEP INC, JUMP END </pre>	<pre> 0→A1, LCTR 9→LIT RPT - 1→AMPCR NOT 0→MIR, INC RPT: A1 + 1→A1, MARI MW1, IF SAI WHEN SAI THEN STEP IF NOT COV THEN INC, JUMP ELSE STEP END </pre>

图 4-3 为改善微程序的效率而不断改进的过程

语句标号是一串字母和数字，其第一个符号必须是字母。语句标号与语句本身是用冒号(:)隔开的。

上述微程序是依靠翻译程序和模拟程序而运行的，其打印结果如图 4-2 所示。在计算机输出中，用句点来代替通常用以隔开语句标号和语句本身的冒号。

4-1-3 效率

在微程序设计中，效率(它用执行一个微程序所需要的时钟数目来度量)是极其重要的。一般由于缺乏经验，一开始就要写出高效率的微程序是困难的。甚至连专业微程序员也不能简单地“随即写出”高效率的微程序。通常，一个微程序要进行不断改进，才能使所需要的时钟数目减为最少。重要的是一开始就要把所需要的功能包括在微程序中，然后，才能改善效率。

作为例子，回顾前面列举的作为循环和语句标号用法的例子的微程序。这个微程序一开始可以写成图 4-3 所示的初始形式。在它作了不断改进后，微程序就可以有另外一些形式(在同一个图中)。

4-1-4 求补操作与逻辑“非”

求补操作用于 II 型指令，逻辑“非”用于 I 型指令。在 II 型指令中，求补的操作方式取决于目标寄存器的类型。对于寄存器 LIT 和 AMPCR 而言，求补操作的就是

$$\text{COMP } n \longrightarrow \text{LIT}$$

$$\text{COMP } n \longrightarrow \text{AMPCR}$$

其中 n 是一个非负整数，先取 n 的反码，然后取所需要的位数送入寄存器(LIT 取 8 位，AMPCR 取 12 位)。对于寄存器 SAR 和 SLIT 而言，求补操作的就是

$$\text{COMP } n \longrightarrow \text{SAR}$$

$$\text{COMP } n \longrightarrow \text{SLIT}$$

即取 n 的补码，分别送入寄存器 SAR 和 LIT。(这里要记住：使用

SLIT 意味着 LIT 寄存器要取信息, 且需要插入第三章提到的附加位, 使得接着进行的 LIT \rightarrow SAR 的操作是有意义的.)

逻辑“非”(即 NOT)是用在 I 型逻辑表达式中的一个算符, 它表示逐位求反, 即把所有那些为 1 的位都变成 0, 把所有那些为 0 的位都变成 1. 在前例中, I 型语句

NOT 0 \rightarrow MIR

指 0 的反码, 它是一个全 1 的字. 显然, B111 \rightarrow MIR 与上述语句等价.

在 I 型语句中有两个目标寄存器可进行求补操作. LCTR 规定把 LIT 中内容的反码送入计数器, 而 CSAR 规定将 SAR 中的内容求补.

4-2 逻辑部件的操作

逻辑部件的操作可分为六类: 逻辑操作、算术操作、条件操作、移位操作、目标描述以及后继控制.

4-2-1 逻辑操作

表 4-1 概括了 D 机器的逻辑操作, 并通过对图 4-4 中的毫微指令的适当位加以置位的方式来规定这些操作. 例如, 逻辑表达式

A1 AND LIT \rightarrow A3

在毫微指令中是用下列位来表示的(见图 4-4):

A1:	17 18 19	加法器 X 输入
	1 0 1	
LIT:	20 21 22 23 24 25 26	加法器 Y 输入
	0 0 0 0 1 0 1	
AND:	28 29 30 31	加法器操作
	1 0 1 1	

A3:

34	35	36
0	0	1

 从 BSW (环移开关) 送到某个

A 寄存器

如果指令写成:

A1 AND LIT \longrightarrow A3, CTR

则下列各位应被置成:

CTR:

46	47	48
1	0	1

翻译程序负责把毫微指令中的适当位加以置位。事实上, 这些位不可能靠人工置位。微程序员书写微程序所具有的唯一手段是用 TRANSLANG 微程序来设计语言。

表 4-1 逻辑操作

(位操作按如下方式计算: $R_i \leftarrow X_i \oplus Y_i$, 其中 R_i 是结果位, 而 \oplus 是所执行的逻辑操作)

逻辑操作	符号操作	逻辑定义	等价的位操作
或非(不是“或”)	X NOR Y	$\bar{X} \wedge \bar{Y}$	$X \nabla Y$
非反蕴含	X NRI Y	$\bar{X} \wedge Y$	$X < Y$
与	X AND Y	$X \wedge Y$	$X \wedge Y$
非蕴含	X NIM Y	$X \wedge \bar{Y}$	$X > Y$
异或	X NOR Y	$(X \wedge \bar{Y}) \vee (\bar{X} \wedge Y)$	$X \neq Y$
等价(“同或”)	X EQV Y	$(X \wedge Y) \vee (\bar{X} \wedge \bar{Y})$	$X = Y$
蕴含	X IMP Y	$\bar{X} \vee Y$	$X \leq Y$
与非(不是“与”)	X NAN Y	$\bar{X} \vee \bar{Y}$	$X \nabla Y$
反蕴含	X RIM Y	$X \vee \bar{Y}$	$X \geq Y$
或	X OR Y	$X \vee Y$	$X \vee Y$
非	NOT X	\bar{X}	
非	NOT Y	\bar{Y}	

4-2-2 算术操作

D机器的算术操作概括在表 4-2 中, 并通过在毫微指令中对适当的位加以置位来控制这些操作。这里再次使用图 4-4 给出的毫微指令一览表。

微控制

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	
0	0	*	SAR						0	0	0	0	0	0	0	0
1	0	SAR						LIT								
1	1	0	0	*	AMPCR											
1	1	1	0	0	0	0	0	LIT								
1	1	1	1	*	毫微地址											

0 不用 (的位)

* 较短的字段 右边对齐

毫微控制

括号中括出的, 是尚未提供的可选的词汇

1 2 3 4 条件测试结果是布尔形式的结果

0	0	0	0	GC1
0	0	0	1	GC2
0	0	1	0	LC1
0	0	1	1	LC2
0	1	0	0	MST
0	1	0	1	LST
0	1	1	0	ABT
0	1	1	1	AOV
1	0	0	0	COV
1	0	0	1	SAI
1	0	1	0	RDC
1	0	1	1	LC3
1	1	0	0	EX1
1	1	0	1	INT
1	1	1	0	EX2
1	1	1	1	EX3

5 FT 条件值

0	NOT end=SC
1	end=SC

6 逻辑部件条件

0	无条件执行
1	按 SC 条件执行

7 外部操作 (MDOP/CAJ) 条件

0	无条件执行
1	按 SC 条件执行

8 9 10 条件修正-CAJ

0	0	0	-
0	0	1	置 LC1
0	1	0	置 C2
0	1	1	清除 GC
1	0	0	置 INT
1	0	1	置 LC3
1	1	0	置 GC1
1	1	1	置 LC1

11 12 13 后继地址

THEN 部分
(当 SC = 1 时使用)

0	0	0	WAIT
0	0	1	(STEP)
0	1	0	SAVE
0	1	1	SKIP
1	0	0	JUMP
1	0	1	EXEC
1	1	0	CALL
1	1	1	RETN

14 15 16 Else 部分

(当 SC = 0 时使用)

0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

17 18 19 加法器的 X 输入

0	0	0	(0)
0	0	1	LIT
0	1	0	ZEXT
0	1	1	CTR
1	0	0	Z
1	0	1	A1
1	1	0	A2
1	1	1	A3

20 21 22 23 24 25 26 加法器的 Y 输入

0	0	-	-	-	-	B0--
0	1	-	-	-	-	BT--
1	0	-	-	-	-	BF--
1	1	-	-	-	-	B1--
-	-	0	0	0	-	B-0-
-	-	1	0	0	-	B-T-
-	-	-	-	0	0	B-0
-	-	-	-	0	1	B-T
-	-	-	-	1	0	B-f
-	-	-	-	1	1	B-1
Comp	1	0	0	Comp		B-F*
Comp	0	0	0	Comp		B-1*
0	0	0	0	1	0	LIT
0	0	0	1	0	0	ZEXT
0	1	0	1	1	0	CTR
0	1	1	0	1	0	Z
0	0	1	1	0	0	AMPCR

* 加法器操作使用 Y 的反码

27				禁止字节间的进位					
0	--	允许							
1	IC	禁止							
28 29 30 31				加法器操作		逻辑			
0	0	0	0	X	+	Y			
0	0	0	1	X	NOR	Y	$X \vee Y$		
0	0	1	0	X	NRI	Y	$X \vee Y$		
0	0	1	1	X	+	Y	+1		
0	1	0	0	X	NAN	Y	$X \vee Y$		
0	1	0	1	X	OAD	Y	$X + (X \vee Y)$		
0	1	1	0	X	XOR	Y	$X \vee Y$		
0	1	1	1	X	NIM	Y	$X \vee Y$		
1	0	0	0	X	IMP	Y	$X \vee Y$		
1	0	0	1	X	EQV	Y	$X \vee Y$		
1	0	1	0	X	AAD	Y	$X + (XY)$		
1	0	1	1	X	AND	Y	XY		
1	1	0	0	X	-	Y	$X + Y$		
1	1	0	1	X	RIM	Y	$X \vee Y$		
1	1	1	0	X	OR	Y	$X \vee Y$		
1	1	1	1	X	-	Y	$X + Y + 1$		
32 33				选择 BSW 的移位方式					
0	0	--	不移位						
0	1	R	右移						
1	0	L	左移						
1	1	C	循环右移						
34 35 36				从 BSW 输入 A 寄存器					
0	0	0	--	不变					
1	-	-	A1						
-	1	-	A2						
-	-	1	A3						
37 38 39 40				B 寄存器的输入选择					
0	0	0	0	--	不变				
0	0	0	1	BC4	四位进位取反				
1	0	0	0	BAD	加法器				
1	0	0	1	BC8	八位进位取反				
1	0	1	0	SDA	BSW \vee 加法器				
1	0	1	1	B	BSW				
1	1	0	0	BEX	外部输入				
1	1	0	1	BMI	MIR				
1	1	1	0	BSE	BSW \vee 外部输入				
1	1	1	1	BBI	BSW \vee MIR				
41				从 BSW 输入 MIR					
0	--	不变							
1	MIR								
42				从 BSW 输入 AMPCR					
0	--	不变							
1	AMPCR								
43 44 45 46				存储器设备地址输入					
0	0	0	-	--	不变				
-	-	1	0	LMAR	来自 LIT				
-	-	1	1	MAR	来自 BSW				
-	1	0	-	BR2	来自 BSW				
-	1	1	1	MAR2	来自 BSW				
1	-	0	-	BR1	来自 BSW				
1	-	1	1	MAR1	来自 BSW				
46 47 48				计数器输入					
-	0	0	--	不变					
0	0	1	LCTR	来自 LIT*					
1	0	1	CTR	来自 BSW*					
-	1	0	INC	+1					
				* 反码					
49 50				SAR 输入					
0	0	--	不变						
0	1	CSAR	取补						
1	0	SAR	来自 BSW						
51 52 53 54				存储器设备操作-MDOP					
0	0	0	0	--	不变				
0	0	1	0	MR1					
0	0	1	1	MR2					
0	1	1	0	MW1					
0	1	1	1	MW2					
1	0	0	0	DL1					
1	0	0	1	DL2					
1	0	1	0	DE1					
1	0	1	1	DE2					
1	1	0	0	DU1					
1	1	0	1	DU2					
1	1	1	0	DW1					
1	1	1	1	DW2					

* 原文为“1011”——译者注

图 4-4 毫微指令——指令位的设置

例如,算术表达式

$$A2 + AMPCR \longrightarrow B$$

在毫微指令中用下列位来表示(见图 4-4)

A2:	<table><tr><td>17</td><td>18</td><td>19</td></tr></table>	17	18	19	加法器的 X 输入				
17	18	19							
	1 1 0								
AMPCR:	<table><tr><td>20</td><td>21</td><td>22</td><td>23</td><td>24</td><td>25</td><td>26</td></tr></table>	20	21	22	23	24	25	26	加法器的 Y 输入
20	21	22	23	24	25	26			
	0 0 1 1 0 0 1								
+	<table><tr><td>28</td><td>29</td><td>30</td><td>31</td></tr></table>	28	29	30	31	加法器操作			
28	29	30	31						
	0 0 0 0								
B:	<table><tr><td>37</td><td>38</td><td>39</td><td>40</td></tr></table>	37	38	39	40	B 寄存器的输入选择			
37	38	39	40						
	1 0 1 1								

同样,表达式

$$LIT - B \longrightarrow AMPCR$$

在毫微指令中被指定为

LIT:	<table><tr><td>17</td><td>18</td><td>19</td></tr></table>	17	18	19	加法器的X输入				
17	18	19							
	0 0 1								
B:	<table><tr><td>20</td><td>21</td><td>22</td><td>23</td><td>24</td><td>25</td><td>26</td></tr></table>	20	21	22	23	24	25	26	加法器的Y输入
20	21	22	23	24	25	26			
	0 1 1 0 0 0 1								
—:	<table><tr><td>28</td><td>29</td><td>30</td><td>31</td></tr></table>	28	29	30	31	加法器操作			
28	29	30	31						
	1 1 1 1								
AMPCR:	<table><tr><td>42</td></tr></table>	42	来自 BSW (环移开关)的 AMPCR 输入						
42									
	1								

有些指令要由翻译程序加以替换。例如,表达式 $A1 + 1 \longrightarrow MIR$ 等价于 $A1 + B_{001} \longrightarrow MIR$, 并且,它对应于下述毫微指令位的配置:

A1:	<table border="1"><tr><td>17</td><td>18</td><td>19</td></tr></table>	17	18	19	加法器的 X 输入
17	18	19			
	1 0 1				

表4-2 算术操作

加 法 操 作	符 号 操 作	逻 辑 等 价
加 法	$X + Y$	
减 法	$X - Y$	$X + \bar{Y} + 1$
或 加	$X \text{ OAD } Y$	$X + (X \vee Y)$
与 加	$X \text{ ADD } Y$	$X + (X \wedge Y)$
	$X + Y + 1$	
	$X - Y - 1$	

B₀₀₁:

20	21	22	23	24	25	26
0	0	0	0	0	1	1

 加法器的 Y 输入

+:

28	29	30	31
0	0	0	0

 加法器操作

MIR:

41

 来自 BSW 的 MIR 输入
1

若没有加法器操作时, 翻译程序产生空操作 $0 + 0 \rightarrow$ 。如前所述, 在每个有加法器操作的时钟周期内, 时钟信号把加法器或环移开关的输出送到指定的目标(若有的话), 并选择一条后继指令。微程序员的任务就是把毫微指令中的适当的位加以置位, 使得机器能完成所需要的操作序列。

4-2-3 条件

一个条件可以有条件地或者无条件地起作用——这取决于毫微指令中适当位的状态。D 机器中的三种操作(逻辑操作、外部操作以及后继指令的选择)可以有条件地或无条件地执行。

在执行一条毫微指令时, 首先要测试条件。根据条件的真假值来执行后面的 D 机器的操作。例如, 在语句

IF COV THEN A1 \rightarrow MIR STEP ELSE SKIP 中, 置如下的毫微指令位(使用图 4-4):

IF COV:

1	2	3	4
1	0	0	0

 条件测试;结果是布尔值

5

 条件值
1 (“真”条件)

6

 逻辑部件的条件
1 (有条件地执行)

STEP ELSE SKIP:

真

11	12	13
0	0	1

 后继指令

14	15	16
0	1	1

 假

A1:

17	18	19
1	0	1

 加法器的X输入

20	21	22	23	24	25	26
0	0	0	0	0	0	0

 加法器的Y输入

28	29	30	31
0	0	0	0

 加法器操作

MIR:

41
1

 来自环移开关 BSW 的 MIR 输入

作为另一个例子,考虑语句

LIT + AMPCR → AMPCR IF ABT THEN MW1 STEP ELSE RETN

在这个语句中,加法器/环移开关的操作是无条件执行的。但是,外部操作(MW1)是有条件执行的,这反映在下述毫微指令位的配置上:

IF ABT:

1	2	3	4
0	1	1	0

 条件测试

5

 条件值
1 (“真”条件)

6 逻辑部件条件
0 (无条件地执行)

7 外部操作条件
1 (有条件地执行)

STEP ELSE RETN:

真 11 12 13 后继指令 14 15 16 假
0 0 1 1 1 1

LIT: 17 18 19 加法器的X输入
0 0 1

AMPCR: 20 21 22 23 24 25 26 加法器的Y输入
0 0 1 1 0 0 1

+: 28 29 30 31 加法器操作
0 0 0 0

AMPCR: 42 从环移开关 BSW 输入到 AMPCR
1

MW1: 51 52 53 54 存储器设备操作
0 1 1 0

在没有条件测试时，由翻译程序产生一个 NOT MST 的空操作，并且置某些位，用以无条件地执行逻辑部件的操作和外部操作。

修正条件说明通常是置位一个指示器，如下面的语句中：

SET LC3

这个语句的毫微指令位的配置是：

SET LC3: 8 9 10
1 0 1

修正条件的操作是由外部操作条件来控制的。

4-2-4 移位操作

移位操作是对加法器的输出在环移开关中执行移位。在执行该操作时，移位的位数总是由移位位数寄存器 (SAR) 中的数值规定的。当要求右移时，就把一个原值送入 SAR，如：

A1 R → B

3 → SAR

当要求左移时，就把一个补码值送入 SAR，如：

A1 L → B

COMP 3 → SAR

循环移位总是向右移位，并把一个原值送入 SAR。

移位操作是通过对毫微指令中的相应位置位来规定的。例如，在语句

A1 R → B

中，毫微指令位的配置如下(见图 4-4)：

A1:	<table><tr><td>17</td><td>18</td><td>19</td></tr></table>	17	18	19	加法器的X输入				
17	18	19							
	1 0 1								
	<table><tr><td>20</td><td>21</td><td>22</td><td>23</td><td>24</td><td>25</td><td>26</td></tr></table>	20	21	22	23	24	25	26	加法器的Y输入
20	21	22	23	24	25	26			
	0 0 0 0 0 0 0								
	<table><tr><td>28</td><td>29</td><td>30</td><td>31</td></tr></table>	28	29	30	31	加法器操作			
28	29	30	31						
	0 0 0 0								
R:	<table><tr><td>32</td><td>33</td></tr></table>	32	33	移位类型选择					
32	33								
	0 1								
B:	<table><tr><td>37</td><td>38</td><td>39</td><td>40</td></tr></table>	37	38	39	40				
37	38	39	40						
	1 0 1 1								

移位操作属于逻辑部件的操作。当在语句中不规定移位时，翻译程序就选择“不移位”，并把毫微指令的 32 位和 33 位都置成“0”。

4-2-5 目标描述

目标描述可以表示下列各组的输入：A 寄存器、B 寄存器、MIR、AMPCR、存储器设备地址、CTR 以及 SAR。每一组都可用一条指令来指示，但是，除 A 的输入以外，每一组内的各输入项都是互斥的。例如下面这条指令

A2 + LIT → A1, A3, BEX, MIR, AMPCR, MAR1, INC, CSAR

可以指示几个“目标描述”。事实上，这条指令的毫微指令位的配置是：

加法器操作

A2:

17	18	19
----	----	----

 加法器的 X 输入
1 1 0

LIT:

20	21	22	23	24	25	26
----	----	----	----	----	----	----

 加法器的 Y 输入
0 0 0 0 1 0 1

+:

28	29	30	31
----	----	----	----

 加法器操作

目标描述

A1 与 A3:

34	35	36
----	----	----

 从环移开关 BSW 输入到 A 寄存器
1 0 1

BEX:

37	38	39	40
----	----	----	----

 选择 B 寄存器的输入
1 1 0 0

MIR:

41

 从环移开关 BSW 输入到 MIR
1

AMPCR:

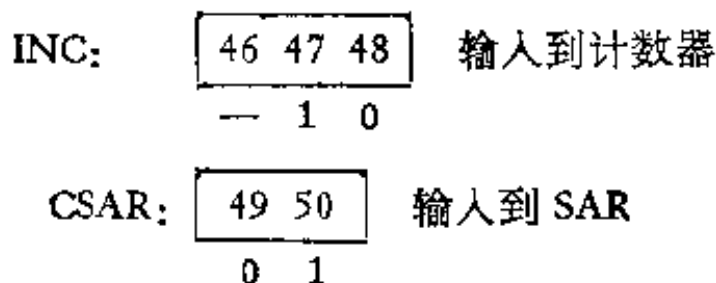
42

 环移开关 BSW 输入到 AMPCR
1

MAR1:

43	44	45	46
----	----	----	----

 输入到存储器设备地址
1 0 1 1



正如图 4-4 中所示,毫微指令的第 46 位为存贮器设备地址输入和计数器输入所共有,因此,必须作如下限制:

1. 若计数器的输入是 CTR, 则存贮器-设备地址的输入不能是 MAR、MAR1 或 MAR2;
2. 若存贮器-设备地址的输入是 LMAR, 则计数器的输入不能是 CTR.

目标描述中的空操作是通过在适当位中置“0”来指示的,它表示“不改变”操作.

4-2-6 下一条命令控制

选择一个语句的下一条命令是通过条件测试控制的,它允许有对应一个“真”条件的下一条命令和对应一个“假”条件的下一条命令.唯一的例外是 SAVE 命令,它隐含着总是指定 STEP 为下一条命令.表 4-3 摘要地列出了对下一条命令控制的各种选择.

4-3 微程序设计方法

本节将列举若干微程序设计方法,力图指出怎样利用 D 机器的全部优点.通过运用这些方法,读者将能理解微程序设计的过程.

4-3-1 常数值的生产

在微程序中,可以用三种方法来产生常数:

1. 来自 S 存贮器;
2. 通过文字赋值;

3. 作为加法器操作的结果。

当常数值来自 S 存贮器时,是通过读命令来获得该常数的,这在后面讨论。

表 4-3 下一条命令控制选择一览表

下一条命令	下一个 MPM 地址	MPCR 的下一个内容	AMPCR 的下一个内容
WAIT	(MPCR)	(MPCR)	——
STEP	(MPCR) + 1	(MPCR) + 1	——
SKIP	(MPCR) + 2	(MPCR) + 2	——
SAVE	(MPCR) + 1	(MPCR) + 1	(MPCR)
CALL	(AMPCR) + 1	(AMPCR) + 1	(MPCR)
EXEC	(AMPCR) + 1	(MPCR)	——
JUMP	(AMPCR) + 1	(AMPCR) + 1	——
RETN	(AMPCR) + 2	(AMPCR) + 2	——

注意: ——表示不变。

(X) 表示 X 的内容。

4-3-1-1 文字赋值

通过文字赋值而产生的常数,来源于 MPM 存贮器,并用下述 II 型指令来赋值。

$n \longrightarrow \text{LIT}$

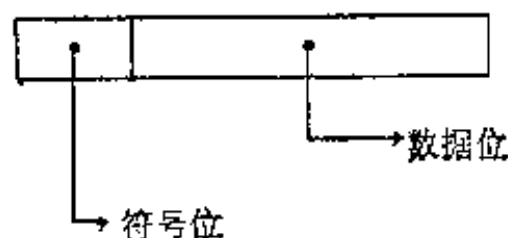
$m \longrightarrow \text{AMPCR}$

通过文字赋值而产生的常数在寄存器中是从最右边开始的,其左边则补零。

4-3-1-2 带符号的数值表示法与补码表示法

在 S 机器的定义中,可以用符号的数值或补码来表示定点数。通过微程序设计可以选择一种表示法。

采用带符号的数值表示法时,一个数值表示成原码形式,符号位固定在前面,如:



在二进制数中,正数的符号位通常是“0”,负数的符号位是“1”。例如,采用8位字长表示一个字时,+6和-6的带符号的数值表示法给出如下:

0	0	0	0	0	1	1	0
---	---	---	---	---	---	---	---

+6 的表示

1	0	0	0	0	1	1	0
---	---	---	---	---	---	---	---

-6 的表示

因此,针对这种数值进行操作的微程序必须把符号与数的表示法结合起来。用带符号的数值形式表示数的加、减的算法如下:

加 法

1. 若被相加的两个数的符号相同,则把两数的值相加,而和数的符号就是它们共同的符号。

2. 若被相加的两个数的符号不同,则计算两数的值之差,且以数值大者的符号为结果符号。

减 法

1. 改变减数的符号。

2. 将被减数与改变过符号的减数相加。

一般地,带符号的数值表示法比补码表示法需要的微程序更为复杂。然而,以原码形式存贮的数值便于输入/输出的变换,从而简化了S程序。当累加寄存器不用于变址和寻址时,最常用的是带符号数值表示法。

采用补码表示法时,正数以原码形式存贮,而负数则以补码形式存贮,如下列的+6与-6的8位二进制表示法为:

0	0	0	0	0	1	1	0
---	---	---	---	---	---	---	---

+6 的表示

1	1	1	1	1	0	1	0
---	---	---	---	---	---	---	---

-6 的表示

因为产生一个二进制数的补码比较容易¹⁾,且对补码进行算术运算可以不考虑符号,因此,用补码进行加法和减法运算是简单而直接的。加法是简单的二进制加运算,如下列的例子所示:

1) 补码是这样产生的: 把每个为1的位变为0,每个为0的位变为1,再把结果加1。

$\begin{array}{r} 0000101 \quad (5) \\ + 00001011 \quad (11) \\ \hline 00010000 \quad (16) \end{array}$	$\begin{array}{r} 11111011 \quad (-5) \\ + 00001011 \quad (11) \\ \hline 10000110 \quad (6) \end{array}$	$\begin{array}{r} 0000101 \quad (5) \\ + 11110101 \quad (-11) \\ \hline 11111010 \quad (-6) \end{array}$	$\begin{array}{r} 11111011 \quad (-5) \\ + 11110101 \quad (-11) \\ \hline 11110000 \quad (-16) \end{array}$

减法可通过把减数取补加上被减数来实现。如下列的例子所示：

$\begin{array}{r} 00001011 \quad (11) \\ - 0000101 \quad (5) \\ \hline 00001011 \quad (11) \\ + 11111011 \quad (-5) \\ \hline 0000110 \quad (6) \end{array}$	$\begin{array}{r} 00001011 \quad (11) \\ - 11111011 \quad (-5) \\ \hline 00001011 \quad (11) \\ + 0000101 \quad (5) \\ \hline 00010000 \quad (16) \end{array}$	$\begin{array}{r} 11110101 \quad (-11) \\ - 0000101 \quad (5) \\ \hline 11110101 \quad (-11) \\ + 11111011 \quad (-5) \\ \hline 11110000 \quad (-16) \end{array}$	$\begin{array}{r} 11110101 \quad (-11) \\ - 11111011 \quad (-5) \\ \hline 11110101 \quad (-11) \\ + 0000101 \quad (5) \\ \hline 11111010 \quad (-6) \end{array}$

一般地，补码表示法所需要的微程序比带符号数值表示法所需要的微程序更简单，但是用补码表示的负数会使输入/输出的变换复杂化。当把通用寄存器用于变址、寻址以及定点运算时，就用补码表示法。

4-3-1-3 加法器产生的常数

把 B 寄存器用作加法器的 Y 选择输入，就可以产生若干个常数和操作数的函数。下面是一组有用的常数和函数。

常数或函数	补码表示法的 Y 选择	带符号数值表示法的 Y 选择	例子
-2	-1 - 1	$+B_{101} + 1$	$A1 - 1 - 1$ 或 $A1 + B_{101} + 1 \rightarrow$
-1	-1	B_{101}	$A1 - 1 \rightarrow$ 或 $A1 + B_{101} \rightarrow$
0	0	0	$0 \rightarrow$
1	1	1	$1 \rightarrow$

2	+ 1 + 1	+ 1 + 1	$A1 + 1 + 1 \longrightarrow$
$-B$	$-B$	B_{FTT}	$-B \longrightarrow$ 或 $B_{FTT} \longrightarrow$
notB	NOTB		NOT B \longrightarrow
sign B		B_{T00}	$A1 \vee B_{T00} \longrightarrow$
abs B		B_{0TT}	$A1 + B_{0TT} \longrightarrow$
neg B		B_{1TT}	$A1 + B_{1TT} \longrightarrow$

用加法器产生常数一般比文字赋值或者从 S 存储器读出常数更快。

4-3-1-4 通过环移开关产生的常数

常数也可通过文字赋值和移位操作的共同作用来产生。这种方法如下：

$LIT\ C \longrightarrow A1$
 $n \longrightarrow SAR, \nu \longrightarrow LIT$

其中 n 是移位的位数，而 ν 是文字值。例如，要把值 237 送入寄存器 A1 的高位字节时，可以用下列语句：

$LIT\ C \longrightarrow A1$
 $8 \longrightarrow SAR, 237 \longrightarrow LIT$

这个操作要到下一次逻辑部件操作时才完成。

形如 2^n 的特殊常数可用下列格式产生：

$1\ C \longrightarrow \text{目标}$
 $COMP\ n \longrightarrow SAR$

例如，在 B 寄存器中，可按如下方法产生数 $2^4 = 16$ ：

$1\ C \longrightarrow B$
 $COMP\ 4 \longrightarrow SAR$

另一方面，可用下列格式把从最左位算起的第 n 位置 1：

$1\ C \longrightarrow \text{目标}$
 $n \longrightarrow SAR$

例如，语句

$1\ C \longrightarrow MIR$

15 \longrightarrow SAR

把 MIR 中从最左位算起的第 15 位置 1。

4-3-2 交换寄存器中的数据

在一台可编程序处理机中，能够用于逻辑部件操作的寄存器的数目一般是有限的，再加上对加法器输入端的限制，为此常常需要交换寄存器中的数据。除不能对专用于交换操作的寄存器进行数据交换外，一般要实现这种交换操作是比较简单的和直接的。

4-3-2-1 不用中间寄存器的交换方法

按下列方法运用异或 (XOR) 逻辑操作可以交换两个寄存器的数值。

reg #1 XOR reg #2 \longrightarrow reg #1

reg #1 XOR reg #2 \longrightarrow reg #2

reg #1 XOR reg #2 \longrightarrow reg #1

例如，语句

A1 XOR B \longrightarrow A1

A1 XOR B \longrightarrow B

A1 XOR B \longrightarrow A1

交换了 A1 与 B 中的数据。

4-3-2-2 交换 B 寄存器与 MIR 中的数据

B 寄存器与 MIR 中的数据可用如下方法交换：

B \longrightarrow MIR, BMI

在 D 机器的微程序设计中，命令 BMI 特别有用，因为它是把 MIR 中的一个字传送到逻辑部件中其他寄存器的唯一方法。

4-3-2-3 利用 MIR 交换 A 寄存器与 B 寄存器中的数据

因为 MIR 并不作为加法器的输入而参与逻辑部件操作，所以它可用作 A 寄存器与 B 寄存器交换数据时的暂存器。下列语句交

换 A 寄存器与 B 寄存器中的数据:

$$A1 \longrightarrow \text{MIR}$$

$$B \longrightarrow A1, \text{BMI}$$

利用 MIR 寄存器胜过使用前面的 XOR 操作, 因为它只要两条指令而不需要三条指令。

4-3-3 比较操作

以 A1 寄存器和 B 寄存器为例, 使用下列关系, 可把通常的操作微程序化:

比较	加法器操作	“真”条件
$A1 < B$	$A1 - B$	NOT AOV
$A1 \leq B$	$A1 - B - 1$	NOT AOV
$A1 \neq B$	$A1 \text{ EQV } B$	NOT ABT
$A1 = B$	$A1 \text{ EQV } B$	ABT
$A1 \geq B$	$A1 - B$	AOV
$A1 > B$	$A1 - B - 1$	AOV

通常, 假定数据是以补码形式存贮的。

应该注意, 如果不考虑 D 机器完成算术运算的方法, 那么“真”条件看起来可能是错误的。然而, 当 D 机器采用补码运算时, 这种“真”条件是正确的, 并且同样也存在着下面的关系:

$$-B \equiv +B_{\text{FFF}} + 1$$

$$-B - 1 \equiv +B_{\text{FFF}}$$

例如, 假定在 8 位的寄存器中, 有下列数值。

$$A1 \quad \boxed{0 \ 0 \ 0 \ 0 \ 1 \ 1 \ 0 \ 1}$$

$$B \quad \boxed{0 \ 0 \ 0 \ 0 \ 0 \ 1 \ 1 \ 0}$$

并且测试条件是: $A1 \geq B$ 。则比较操作按下述方法计算:

$$A1 - B \equiv A1 + B_{\text{FFF}} + 1$$

其执行过程为:

$$\boxed{0 \ 0 \ 0 \ 0 \ 1 \ 1 \ 0 \ 1} \quad (A1)$$

$$\begin{array}{r}
 \boxed{1\ 1\ 1\ 1\ 1\ 0\ 0\ 1} \quad (B_{FFF}) \\
 + \quad \boxed{0\ 0\ 0\ 0\ 0\ 0\ 0\ 1} \quad (1) \\
 \hline
 \begin{array}{c}
 1 \\
 \uparrow
 \end{array}
 \boxed{0\ 0\ 0\ 0\ 0\ 1\ 1\ 1}
 \end{array}$$

溢出

因此,对加法器是否溢出进行测试就是正确的了. 如果 A1 内容小于 B 内容,加法器就不会产生溢出.

4-3-4 循环

因为 S 机器的许多操作是逐位完成的,所以循环是微程序设计的一个重要方式. 下面介绍几个控制循环执行的方法.

4-3-4-1 提前判定法

在执行循环之前,提前判定法测试某些控制参数,并使用下列格式:

```

TEST - 1 → AMPCR
..., CALL
LOOP: ...
TEST: IF 条件 THEN..., JUMP

```

其中,省略号(...)表示一条或多条 1 型指令. CALL 命令使控制转移到条件语句并为后面的叠代建立循环地址. 下面列出一个微程序,它借助重复加的方式来实现两个正整数的相乘,并且使用了提前判定法. 这种判定法也适合于乘数为零的情况.

```

INIT: LIT R → A1, BAD    % LOAD 10 TO B AND
      10 → LIT, 1 → SAR  % 5 TO A1
COMMENT LOOP STARTS HERE
      TEST - 1 → AMPCR
      0 → A2, CALL - % BRANCH TO TEST
LOOP: A2 + B → A2    % PRODUCT IN A2

```

TEST: A1 - 1 → A1

IF AOV THEN JUMP % BACK TO LOOP

在这个微程序中, A1 存放被乘数, B 存放乘数, 而 A2 存放乘积。这段微程序已借助翻译程序/模拟程序运行过, 图 4-5 中给出了打印输出。用参考语言写的上述微程序与用 TRANSLANG 写的同一个程序都说明了注解的用法, 注解被安排在同一行上, 或者作为单独的一个语句。而且, 当在一个 IF 语句中省略了 ELSE 时, 则认为省略了下一条命令 ELSE STEP。

4-3-4-2 固定叠代

下列格式可以使一个循环正好执行 n 次¹⁾:

$n \rightarrow \text{CTR}$

TEST - 1 → AMPCR

..., INC, CALL

LOOP: ...

...

TEST: IF NOT COV THEN ..., INC, JUMP

其中, 有关省略号与 CALL 语句的约定与前面的格式相同。下列微程序把 S 存贮器中 65 到 69 号单元的五个数相加, 其和送到 S 存贮器的 70 号单元:

B₀₀₁ L → A2, LCTR

5 → LIT, COMP 6 → SAR

TEST - 1 → AMPCR

0 → A1, INC, CALL

LOOP: MR2

WHEN ROC THEN BEX

A1 + B → A1, MIR

TEST: IF NOT COV THEN A2 + 1 → A2, MAR2, INC, JUMP

1) 应当承认: 除非 n 是特殊值, 否则语句 $n \rightarrow \text{CTR}$ 并非是一个正确的语句。该语句用以表示把一个数值取到 CTR。下一个例子则是正确的。

WRITE: $A2 + 1 \rightarrow MAR2$

MW2, IF SAI

WHEN SAI THEN STEP

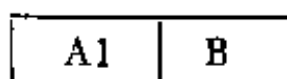
在这个微程序中，A2 存放访问 S 存储器单元地址的当前值，A1 存放和数。这段微程序已借助翻译程序/模拟程序运行过，在图 4-6 中给出了打印的输出结果。

4-3-5 移位

在仿真程序中，为了执行双倍长寄存器的定点操作以及在指令译码时将各字段正确的分隔开来，移位操作是必要的而且是经常使用的。因此，提高移位的效率将对提高整个系统的效率有重要作用。

4-3-5-1 双倍长的移位

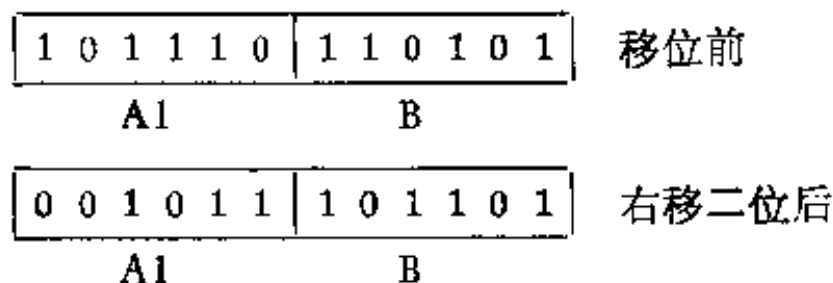
假定 A1 与 B 寄存器在逻辑上构成一个双倍长的寄存器，如：



则双倍长的右移操作可定义如下：

1. A1 的最右位移移入 B 中；
2. A1 的左边补“0”；
3. B 的最右位被移掉。

为简单起见，假定 A1 与 B 寄存器都是 6 位，现要求将它们右移二位则有：



因此，由 SAR 给出移位位数而使寄存器进行右移的微程序格式如下：

B R \rightarrow MIR

a 翻译

>>>> TRANSLANG D MACHINE MICROTRANSLATOR

ENTER FILENAME FOR HEX? F45HEX
 SUPPRESS BIT PATTERNS?(1=SUPPRESS)? 1
 SUPPRESS HEX LISTING?(1=SUPPRESS)? 1
 IF INPUT IN FILE ENTER 1? 31
 ENTER SOURCE FILENAME? FIG45

```
0000  INIT. LIT R = A1, BAD $ LOAD 10 TO B AND
0001      10 = LIT, 1 = SAR $ 5 TO A1
0002      COMMENT LOOP STARTS HERE $
0002      TEST - 1 = AMPCR $
0003      0 = A2, CALL $ BRANCH TO TEST
0004  LOOP. A2 + B = A2 $ PRODUCT IN A2
0005      TEST. A1 - 1 = A1 $
0006      IF ADV THEN JUMP $ BACK TO LOOP
0007      END $
```

THE TOTAL NUMBER OF ERRORS = 0

EXECUTE (Y/N)? N

b 执行

2) DMACH (COMPILED) 07 JAN 76 17:40

ENTER SAME FILENAME FOR HEX? F45HEX
 OUTPUT REGISTERS AND \$ MEMORY IN INTEGER(1) OR OCTAL(2)? 1
 INPUT \$ MEMORY IN INTEGER(1) OR OCTAL IN Q11 FORMAT(2)? 1
 STARTING ADDRESS =? 0
 MAXIMUM NUMBER OF CLOCKS TO SIMULATE=? 30
 NUMBER OF CLOCKS BETWEEN OUTPUT POINTS=? 1

ENTER OUTPUT LINES DESIRED 1-ADDRESSES AND CLOCK
 2- A1, A2, A3, B 3- MIR, SAR, LIT, CTR, AMPCR 4- BR1, BR2, MAR, BMAR, GC1, GC2
 5- CONDITIONS
 ENTER NUMBER OF OUTPUT LINES DESIRED? 5

BEGIN OUTPUT AT MPH ADDRESS=? 7
 END OUTPUT AT MPH ADDRESS=? 7
 ENTER 1 FOR \$ MEMORY DUMP WHEN PROGRAM TERMINATES?

ENTER \$ MEMORY VALUES IN CONSECUTIVE BLOCKS
 ENTER 9999 FOR STARTING ADDRESS WHEN FINISHED

STARTING \$ MEMORY ADDRESS=? 9999

END OF SIMULATION - REGISTERS CONTAIN

```
P(1) ADDR. = 6 P(3) ADDR. = 6 CLOCK = 22
A1 = 4294967295 A2 = 50 A3 = 0 B = 10
MIR = 0 SAR = 1 LIT = 10 CTR = 0 AMPCR = 3
BR1 = 0 BR2 = 0 MAR = 0 BMAR = 0 GC1 = 0 GC2 = 0
LC1 = 0 LC2 = 0 MST = 1 LST = 1 ABT = 1 ADV = 0 CAV = 0 SA1 = 0 RDC = 0 INT = 0
```

图 4-5 借助重复加的方式来实现整数乘法的微程序。A1 存放被乘数(开始时), B 存放乘数, A2 存放乘积(执行完成后)

a 翻译

>>>> TRANSLANG D MACHINE MICROTRANSLATOR

ENTER FILENAME FOR HEX? F46HEX
 SUPPRESS BIT PATTERNS?(1=SUPPRESS)? 1
 SUPPRESS HEX LISTING?(1=SUPPRESS)? 1
 IE INPUT IN FILE ENTER 1? 3!
 ENTER SOURCE FILENAME? F1046

```
0000      8001 L = A2, LCTR 3 FORM 64 IN A2
0001      S = LIT, COMP 6 = SAR 5
0002      TEST - 1 = AMPCR 5
0003      0 = A1, INC, CALL 5
0004      LOOP, MR2 5
0005      WHEN RDC THEN BEK 5
0006      A1 + B = A1, MIR 5
0007      TEST, IF NOT COV THEN A2 + 1 = A2, MAR2, INC, JUMP 5
0008      WRITE, A2 + 1 = MAR2 5
0009      MW2, IF SAI 5
0010      WHEN SAI THEN STEP 5
0011      END 5
```

THE TOTAL NUMBER OF ERRORS = 0

EXECUTE (Y/N)? N

b 执行

2) DMACH1 (COMPILED) 07 JAN 76 17:52

ENTER SAME FILENAME FOR HEX? F46HEX
 OUTPUT REGISTERS AND S MEMORY IN INTEGER(1) OR OCTAL(2)? 1
 INPUT S MEMORY IN INTEGER(1) OR OCTAL IN 011 FORMAT(2)? 1
 STARTING ADDRESS =? 0
 MAXIMUM NUMBER OF CLOCKS TO SIMULATE=? 50
 NUMBER OF CLOCKS BETWEEN OUTPUT POINTS=? 1

ENTER OUTPUT LINES DESIRED 1-ADDRESSES AND CLOCK
 2- A1, A2, A3, B 3- MIR, SAR, LIT, CTR, AMPCR 4- BR1, BR2, MAR, BMAR, GC1, GC2
 5- CONDITIONS
 ENTER NUMBER OF OUTPUT LINES DESIRED? 5

BEGIN OUTPUT AT MPM ADDRESS=? 1!
 END OUTPUT AT MPM ADDRESS=? 1!
 ENTER 1 FOR S MEMORY DUMP WHEN PROGRAM TERMINATES? 1

ENTER S MEMORY VALUES IN CONSECUTIVE BLOCKS
 ENTER 9999 FOR STARTING ADDRESS WHEN FINISHED

STARTING S MEMORY ADDRESS=? 65
 FINAL S MEMORY ADDRESS FOR THIS BLOCK=? 70
 SMEM(65)=? 4
 SMEM(66)=? 2
 SMEM(67)=? 6
 SMEM(68)=? 5
 SMEM(69)=? 9
 SMEM(70)=? 0

STARTING S MEMORY ADDRESS=? 9999

图 4-6 对 S 存储器的 65 到 69 号单元的 5 个数值求和,并把和数送入 S 存储器的 70 号单元。A2 存放 S 存储器单元地址的当前值,而 A1 则存放和数

A1 L→BBI

A1 R→A1

目标命令 BBI 把环移开关的输出与 MIR 进行逻辑“或”，结果送入 B 寄存器。读者可在 SAR 的值为 2 的条件下对上述 A1 与 B 的内容验证这种微程序格式的正确性。

对于双倍长度左移，假定 A1 与 B 寄存器在逻辑上作成如下形式：

B	A1
---	----

则双倍长的左移操作定义如下：

1. A1 的最左位移入 B；
2. A1 的右边补“0”；
3. B 的最左位被移掉。

为了简单起见，再次假定 A1、B 两个寄存器都是 6 位，要求将它们左移二位，则有：

1 0 1 1 1 0	1 1 0 1 0 1	移位前
B	A1	

1 1 1 0 1 1	0 1 0 1 0 0	左移二位后
B	A1	

那么，由 SAR 给出移位位数的补码而且使寄存器左移的微程序格式给出如下：

B L→MIR

A1 R→BBI

A1 L→A1

因此，为了左移二位，所需要的移位位数为

COMP 2→SAR

读者可用上述 A1 与 B 的内容再次验证这个微程序格式的正确性。

4-3-5-2 字段的分隔

为了把一个字中的某个字段的某些位分离出来，通常采用移

位与屏蔽的方法。下面给出这种要求的一般提法：

输入字，A1

m 位	要分离的字段	n 位
-------	--------	-------

要求的结果，A2

$m + n$ 个位“0”	被分离的字段
---------------	--------

最简单的方法是先左移，使左端移掉 m 位，接着再右移，使右端移出 $m + n$ 位，如：

A1 L \longrightarrow A2
 COMP $m \longrightarrow$ SAR
 A2 R \longrightarrow A2
 $m + n \longrightarrow$ SAR

一种可望有更高效率的字段分隔方法是采用屏蔽与移位相结合的方法，如：

NOT 0 R \longrightarrow B
 $m \longrightarrow$ SAR
 A1 AND B R \longrightarrow A2
 $n \longrightarrow$ SAR

这个方法之所以能提高效率乃在于：前面两条指令仅用来构成屏蔽，这样的屏蔽可以利用包括文字赋值在内的某些方法实现。还可以使用后面跟有屏蔽操作的循环移位，如：

A1 C \longrightarrow A2
 $n \longrightarrow$ SAR, $k \longrightarrow$ LIT
 A2 AND LIT \longrightarrow A2

其中， $k = 2^w - 1$ ， w 是被分离的字段的宽度。若 k 超过 8 位，则可用 AMPCR 代替 LIT。

4-3-6 其他技术

有效地使用逻辑操作，有助于研制高效率的微程序。例如，把 A 寄存器的值加倍的常用方法如下：

A1 \longrightarrow B
 A1 + B \longrightarrow A1

更有效的方法是用或加 (OAD)指令, 如:

A1 OAD 0 \longrightarrow A1

使用这同一个逻辑操作, 下一次能够对寄存器的次最高位进行测试, 如:

A1 OAD 0 \longrightarrow

IF MST THEN...

其他各种技术, 将在本书的后面在讨论微程序设计的专题时再介绍。

词 汇

读者应当熟悉本章所使用的下列术语:

语句标号 (Statement label)

微程序效率 (Microprogram efficiency)

求补 (Complement)

逻辑“非” (Logical negation)

逻辑操作 (Logical operation)

算术操作 (Arithmetic operation)

条件 (Condition)

修改条件说明 (Condition adjust specification)

移位操作 (Shift operation)

目标说明 (Destination specification)

后继命令 (Successor)

常数 (Constant)

带符号的数值表示法 (Signed magnitude representation)

补码表示法 (Two's complement representation)

交换 (Exchange)

循环 (Loop)

提前判定法 (Method of leading decisions)

双倍长的移位 (Double length shift)

字段分隔 (Field isolation)

提 问

下列问题打算用来测验你对主题内容的理解,所有问题都可以直接从课文或者通过对所提出的主题进行逻辑推理而得到答案。有些问题适合于在讨论班上进行研究。

1. 给出纪录微程序循环中叠代次数的两种方法。
2. COMP, 何时取补?何时取反?
3. 逻辑算符 NOT 是取反还是取补?
4. 一个高效率的微程序“首要的”或者根本的要素是什么?
5. 能够有条件或无条件完成的三种基本微程序功能是什么?
6. 毫微指令中第 28 位到 31 位能够定义什么?
7. 对于右移操作,放在 SAR 中的移位量是原码还是补码?
8. 毫微指令中,哪一位是共享的?它共用什么功能?
9. 不写出条件的后继命令是什么?
10. 产生常数的三种方法是什么?
11. 假定存放在 B 寄存器中的数值是用带符号的数值表示法表示的,则 B₀TT 完成什么数学功能?
12. 如何把 MIR 的信息传送到其他机器寄存器?
13. 提前判定法的主要优点是什么?

习 题

1. 写出用以把 S 存储器前 n 个单元求和的微程序,其中 n 可以是 0 到 225 中的任何一个数值。
2. 使用假想的 8 位寄存器,针对下列各种操作,试按二进制形式给出例子:

X NRI Y

X NIM Y

X EQU Y

X IMP Y

X RIM Y

3. 针对下列语句,指出毫微指令位的配置:

A1 + 1 → MIR, BEX

IF ABT THEN INC, JUMP ELSE SKIP

MRI, IF COV THEN B → MIR, RETN ELSE JUMP

A1 AND B \rightarrow A2, AMPCR, MAR, B, MIR

LCTR, INC,

4. 设 S 存贮器的 100 号和 101 号单元分别存放 H 和 I, 写出计算

$$H + 2 * I$$

并将其结果存入 S 存贮器的 102 号单元的微程序。

5. 写出分离寄存器 A1 的第 20 位到第 27 位字段并把结果置于寄存器 A2 右边的微程序。

6. 寄存器 A1 和 B 存放的是用带符号数值表示法表示的整数值。写出一个微程序来比较 A1 与 B 的内容, 并且, 若 $A1 \geq B$ 时, 就转移到符号单元 REPEAT。

第五章 仿真入门

5-1 任 务

本章的任务是说明一个简单的仿真程序的特性。实际上，本章内容与前面讨论过的许多概念都是联系在一起的，并且也作为介绍微程序设计的参考。

虽然微程序设计已经广泛地应用在设备/控制器和现代实验室设备方面，但是，仿真仍然是微程序设计的主要应用。为了支持仿真，需要采用特殊的微程序设计技术，介绍这些技术则是本章的第二个任务。

5-2 有关仿真的微程序设计技术

仿真通常包括：基址寄存器在 MPM 和 S 存贮器寻址中的用法；作“跳转表”用的文字赋值表；有效地使用 S 存贮器操作以及在这种方式下完成操作的方法。一般来说，这些概念和方法与特定的微程序计算机无关。

5-2-1 基址寄存器

我们已经说过，寄存器 BR_i 和 MAR 可以连在一起形成一个 16 位的地址，用 BR1 或者 BR2 可以指点 S 存贮器中一个 256 字的字组。因此，在基址/变址/位移量寻址方式中，BR1 或 BR2 作为基址寄存器可用于 S 存贮器的寻址，或者，作为基址寄存器可用作指点保存在 S 存贮器中的累加寄存器。

5-2-1-1 S 存储器的访问

对 S 存储器寻址是基于下面两个典型的要求:

1. 为了访问 S 机器中的某些寄存器,即通用寄存器,浮点寄存器等;
2. 为了访问 S 存储器中的数据(即操作数)和指令(即 S 机器的指令).

为了给 BR1 和 BR2 寄存器指定一种特定的访问类型,需要建立一种任意的、但又是有用的约定. 这种约定为:

BR1 指点 S 存储器中的 S 机器的寄存器;

BR2 指点 S 存储器中的指令和数据.

所以, BR1 总是通过特定的操作接收地址信息. 如下述语句中:

$$A1 \longrightarrow BR1$$

相应地, MAR 也用类似方法赋值. 另一方面, BR2 则总是隐含在别的操作中接收地址信息的(作为 S 存储器地址的高位部分), 如下述语句中:

$$A2 + 1 \longrightarrow MAR2$$

其中, BR2 和 MAR 同时接收信息.

5-2-1-2 取信息到存储器地址寄存器

通常可用两种方法取信息到 MAR, 以供 S 机器的寄存器操作使用. 当已知 S 机器的寄存器与 S 程序无关时, 则可用下面的格式:

$$LMAR, \dots$$
$$S \text{ 机器寄存器号} - \# \longrightarrow LIT$$

如下例中:

$$A1 + 1 \longrightarrow A2, LMAR$$
$$4 \longrightarrow LIT$$

当 S 机器寄存器不能肯定是否与 S 程序无关时, 它可以是 D 机器

寄存器中被分隔的字段，并通过使用下列格式的 I 型指令来给 MAR 赋值：

S 机器寄存器 \longrightarrow MAR

如语句

B \longrightarrow MAR

当使用 LMAR 命令，把 LIT 的内容传送到 MAR 时，它无需通过加法器或环移开关。

BR1 或者 BR2 与 MAR 组合起来后称为 BMAR。

5-2-2 S 存贮器的读和写操作

前面几章中给出的对 S 存贮器的读和写操作过程虽被简化了，但都是完全正确的。实际的读和写操作的过程要更复杂些。

5-2-2-1 一般说明

首先要考虑的是，在 S 存贮器的操作中包括两类信息：地址和数据。考虑下述格式（这种格式以前已经用过，今后将经常使用）：

S 存贮器地址 \longrightarrow MAR2

MR2,

...

WHEN RDC THEN BEX, ...

（显然，在 MR2 与 WHEN 语句之间，可以插入若干条指令，并且，在实时操作中，这种格式可提供重叠操作）。事实上读操作包含两个步骤：

1. 通过互锁开关从 BMAR 接收存贮器地址；
2. 把读出数据置于外部输入总线上¹⁾，以便能用 BEX 命令把它读入 B 寄存器。

写操作只包含一个步骤，即通过互锁开关来接受 BMAR 和

1) 这条总线称为外部数据接口或 XDI。

MIR 中的地址和数据。

5-2-2-2 读操作的扩充

在一次读操作期间,通过使用下列格式,可以在读操作完成以前使用 BMAR 地址:

格	式	注	解
S 存储器地址	→MAR2		
MR2, ..., IF SAI		把 SAI 清“0”	
...		至少插入一条指令	
WHEN SAI THEN ...		现在可以改变地址	
...			

WHEN RDC THEN BEX, ... 数据已经可以使用

然而,为多处理机系统设计的重叠操作功能,在 D 机器的模拟程序中却很少(或者很少需要)使用。

5-2-2-3 写操作的扩充

S 存储器写操作的正确格式如下:

格	式	注	解
S 存储器地址	→MAR2		
数据	→MIR		
MW2, ...,IF SAI		启动写并清除 SAI	
...		至少插入一条指令	
WHEN SAI THEN ...		可以改变 MIR 和 MAR	

在一次读或写操作之后,要花费一条指令的时间来置位 SAI,否则,直接跟在读或写指令后面的 WHEN SAI 语句将被延迟执行(顺便说一下,在图4-1中就有这种情况,其中 WHEN SAI 直接跟在命令 MW1 之后*)。

*) 在启动 S 存储器读或写操作时,用 IF SAI 把 SAI 清成“0”。在完成读或写操作后,由 S 存储器置位 SAI。因此,直接跟在读或写指令后面的 WHEN SAI 语句将被延迟一条指令的时间,等待 S 存储器完成读或写操作后置位 SAI。
——译者注

可以用一条 I 型指令送信息到 MIR 和 MAR, 如下面的格式所示:

逻辑部件的操作 \longrightarrow MIR, LMAR

S 机器寄存器号 \longrightarrow LIT

MW1, ..., IF SAI

不能把 BR1, BR2 和 MAR 的内容送回逻辑部件, 但可用命令 BM1 和 BBI 把 MIR 的内容送到 B 寄存器.

5-2-2-4 定时

因为没有设备会与 S 存贮器的操作冲突, 所以可用下面格式来执行读出-传送操作:

MR1, ..., BEX

WHEN RDC THEN...

使得当 D 机器遇到一个“真”条件时, 就完成 BEX 命令.

一般地, 当用一条 I 型指令来指定一个目标寄存器时, 要等到接收到下一次的“真”条件时, 才能更换目标寄存器. 可是, 逻辑部件和存贮器-设备的操作是在正常的相位 3 周期中执行的. 所以, 在前面的例子中, MR1 立即被执行, 而 BEX 是在 RDC 条件为“真”时才被执行.

5-2-2-5 对同一数据的多次读

在 S 存贮器读操作之后, 只要重复 BEX 命令而不必再插入读命令, 就能重复访问来自外部设备接口 (XDI) 上的同一个数据. 例如在下述格式中:

S 存贮器地址 \longrightarrow MAR2

MR2

...

BEX

...

BEX

每一个 BEX 都把同一个数据传送到 B 寄存器。反之，在下述格式中：

S 存储器地址-1 → MAR2
MR2
...
BEX
...
S 存储器地址-2 → MAR2
MR2
...
BEX

每一个 BEX 命令都取到一个不同的数据。

5-2-3 文字表

因为象表目之类的常数不能存贮在微程序中，所以，每次使用的一个常数，不论它表示的是数据还是地址，都必须用一条 I 型或 II 型指令把它传送到一个寄存器中。因此，S 程序中的一个表，在微程序设计中通常是作为文字赋值语句表来实现的。在 S 机器的程序设计中，用取数指令来访问一个表目。在微程序设计中，则用 EXEC 指令来把一个表目送入一个机器寄存器中。在 S 机器程序设计和微程序设计中，表的访问方法如图 5-1 所示。

5-2-3-1 跳转表

跳转表的形式取为：

标号 A - 1 → AMPCR
标号 B - 1 → AMPCR
⋮
⋮
⋮
标号 K - 1 → AMPCR

其中标号是微程序中的语句标号，而-1（即标号-1）是为了补偿

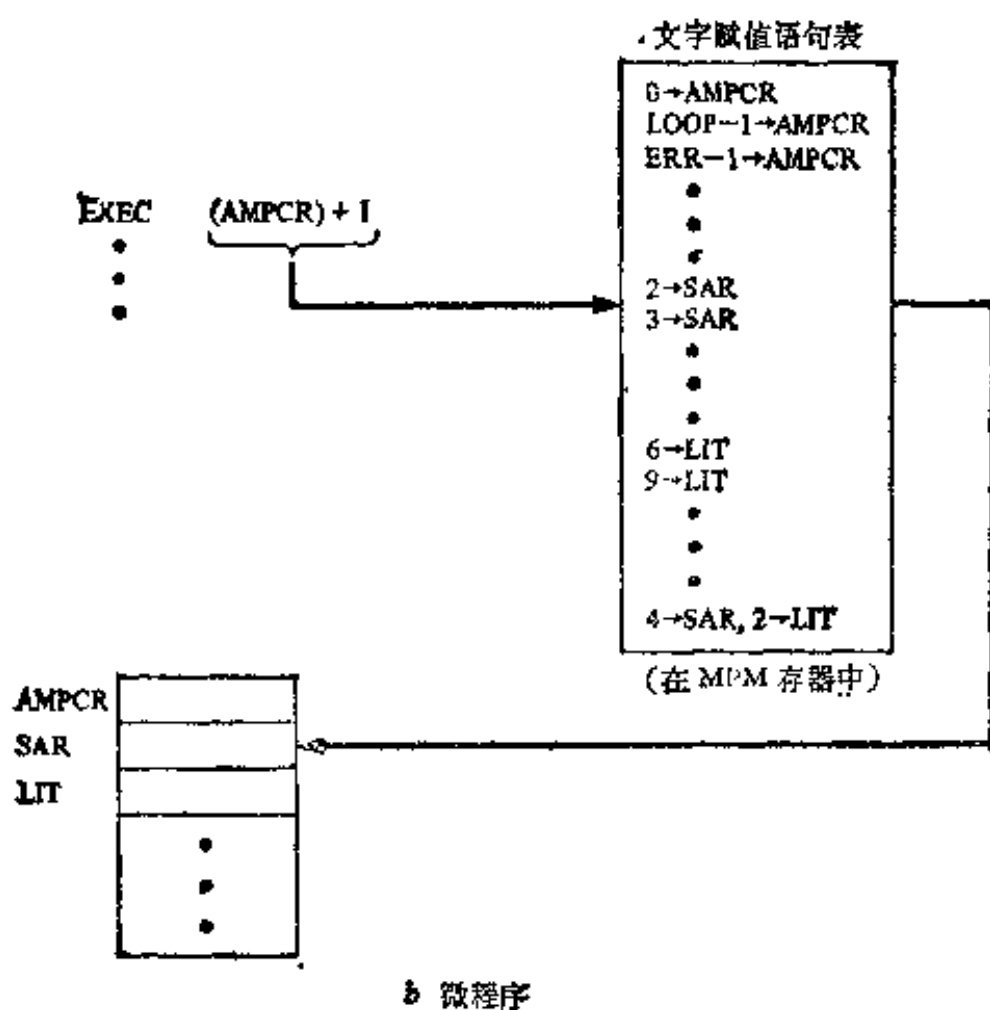
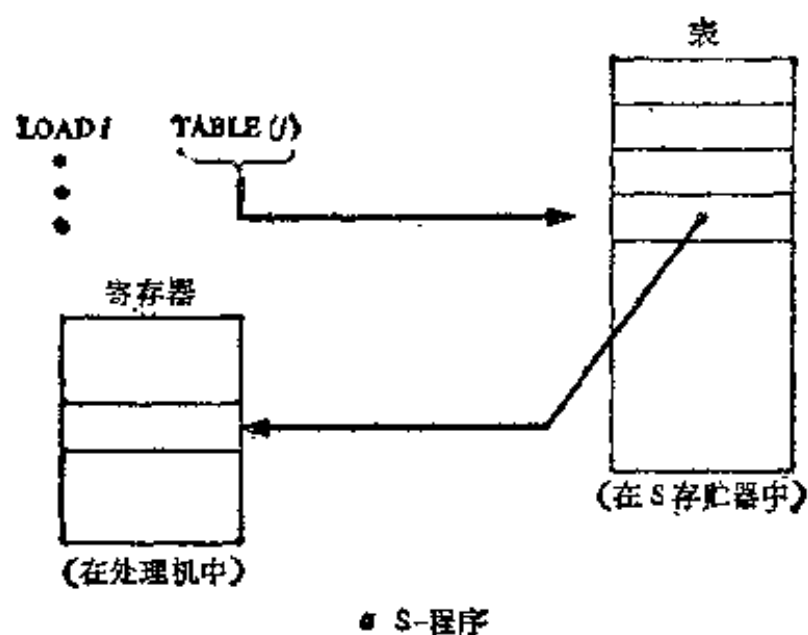


图 5-1 在 S 程序和微程序中表的访问方法

D 机器中的转移地址总是 AMPCR 的内容加 1 (即 $(AMPCR) + 1$) 而设置的。为了访问这个表, 我们使用如下一组语句序列:

$A1 + AMPCR \longrightarrow AMPCR$

基址 $-1 \longrightarrow AMPCR^{*)}$

EXEC

JUMP

其中, $A1$ 保存表的位移量. 而语句“基址 $-1 \longrightarrow AMPCR$ ”把表的基址送入 $AMPCR$, 而语句“ $A1 + AMPCR \longrightarrow AMPCR$ ”把所需要的文字赋值指令的地址送入 $AMPCR$. 下一条 EXEC 指令则执行位于跳转表中 $(AMPCR) + 1$ 处的指令, 也就是用插入方式把程序控制转移到所希望的 MPM 单元. 然后, 继续执行

微 程 序		注 解
TBL: ABLE $-1 \longrightarrow AMPCR$	}	跳转表
BAKER $-1 \longrightarrow AMPCR$		
CHARLY $-1 \longrightarrow AMPCR$		
...		
$1 \longrightarrow A2$		位移量 1 送到 $A2$
$A2 + AMPCR \longrightarrow AMPCR$		计算文字单元
TBL $-1 \longrightarrow AMPCR$		
EXEC		把跳转地址送入 $AMPCR$
JUMP		把控制转移到 $(AMPCR) + 1$
ABLE: —		
—		
—		
—		
BAKER: —		控制转到这里
—		
—		
—		
CHARLY: —		
—		
—		
—		

图 5-2 使用跳转表的例子

^{*)} $A1 + AMPCR$ 是 I 型指令, 基址 $-1 \longrightarrow AMPCR$ 是 II 型指令, 由于 II 型指令与 I 型指令的操作重叠, 结果在 $AMPCR$ 中得到“基址 + $A1 - 1$ ”。——译者注

EXEC 后面的指令。最后, JUMP 命令把程序控制转移到跳转表中由 A1 中的位移量所确定的语句标号处。若在转移出去后还希望能返回, 那么, 应该用 CALL 代替 JUMP。在图 5-2 中, 给出了使用跳转表的一个例子。

5-2-3-2 EXEC 命令

在表 4-3 所列的下一条命令选择一览表中, 对于 EXEC 命令中“MPCR 的下一个内容”是 (MPCR), 就如同 WAIT 命令一样。而 STEP 命令则有入口 (MPCR) + 1。这说明: 应首先执行 EXEC 命令, 然后再执行语句的其余部分。这种设计允许用一条指令从文字表中选取该指令要用的屏蔽值、移位值和操作数值。例如, 考虑下述微程序段:

1 SAR, 3 \longrightarrow LIT (1)

2 SAR, 7 \longrightarrow LIT (2)

3 SAR, 15 \longrightarrow LIT (3)

...

表索引 \longrightarrow AMPCR (4)

...

A3 AND LIT R \longrightarrow ..., EXC (5)

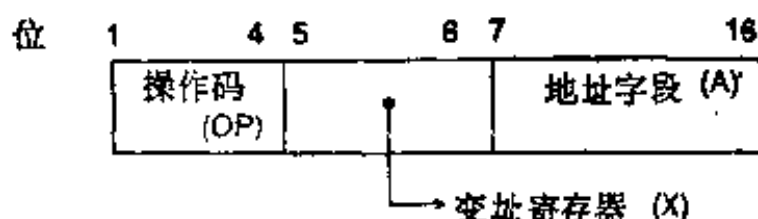
语句 (1), (2) 和 (3) 表示一个文字表。语句 (4) 把表目的 MPM 地址送入 AMPCR。语句 (5) 执行插入的文字赋值语句, 分别把屏蔽值和移位值送入 LIT 和 SAR, 然后用新确定的操作数来执行 AND 操作。把 LIT 寄存器用作屏蔽值只能控制一次加法器操作的最右边 8 位, 而最左边 24 位皆为 0。如果在利用 EXEC 命令的灵活性的同时, 要求有更大的屏蔽值, 则可在一个文字赋值语句和逻辑操作中使用 AMPCR。这样就提供了一个使用 12 位屏蔽的方法。

5-3 一个简单的仿真程序的设计

本书介绍一个简单的仿真程序以及相应的微子程序组¹⁾。对这组微子程序的完整分析将在下一节给出。为了简单起见,在假定D机器的逻辑部件为16位的情况下,虽然所给出的微程序是正确的,但它不能借助我们已按32位的机器编制的翻译/模拟程序来运行。在第九章将把这个仿真程序扩充为32位,它具有多个寄存器。

5-3-1 S机器的特性

S机器是一台单累加器、单地址的计算机。每条指令的字长都是16位,与累加器的字长相同。指令的格式是:



因此,操作码字段是4位,变址寄存器字段是2位,地址字段是10位。在表5-1中,定义了八种不同的机器操作。

5-3-2 S机器映象

当把S机器映象到微程序计算机上时,为了表示S机器寄存器,必须建立某些约定。显然,仿真程序效率的高低取决于微程序处理机与S机器相匹配的程度。

在仿真程序中,使用下面的规定:

1. 把D机器中的A1用作现行地址寄存器;
2. 把D机器中的A2用作指令寄存器,并在指令处理过程中用作暂存器;

1) 这个仿真程序是由Burroughs公司的Earl Reigel博士设计和编制微程序的。

表 5-1 一台单地址计算机的仿真程序的机器操作

名称	操作码	记忆符	说 明	符号表示法
取数	0001	LD	将 S 存储器中地址为 $A + (X)$ 的单元的内容取到累加器 (ACCUM) 中	$(A + (X)) \rightarrow (ACCUM)$
存取	0010	ST	把累加器的内容存入地址为 $A + (X)$ 的 S 存储器单元中	$(ACCUM) \rightarrow (A + (X))$
转移	0011	BR	转移到 S 存储器中地址为 $A + (X)$ 的指令	$\rightarrow A + (X)$
测试零	0100	TZ	若累加器的内容为全“0”，则转移到 S 存储器中地址为 $A + (X)$ 的指令	$\rightarrow A + (X)$ 若 $ACCUM = 0$
测试最低位	0101	TL	若累加器的最低位 (即最小位或最右位) 是“1”，则转移到 S 存储器中地址为 $A + (X)$ 的指令	$\rightarrow A + (X)$ 若 $ACCUM_{16} = 1$
减量	0110	DEC	S 存储器地址为 $A + (X)$ 的单元的内容减“1”	$(A + (X)) - 1 \rightarrow (A + (X))$
增量	0111	INC	S 存储器地址为 $A + (X)$ 的单元的内容加“1”	$(A + (X)) + 1 \rightarrow (A + (X))$
移位	1000	SHF	累加器的内容循环右移 1 位	

3. 把 D 机器中的 A3 用作 S 机器的累加器；

4. BR1 指示 S 存储器的变址寄存器。

在这个例子中，假定两台机器的字长是相同的。所有其他的考虑都服从于能够清楚和简单地表达这个例子的目的。

5-3-3 仿真程序的微程序

图 5-3 给出了仿真程序的微程序。假定 S 程序已被存入 S 存储器中，并且，寄存器 A1 的内容开始时指向要执行的第一条指令。类似地，BR1 由初始微子程序置位，以指向编号为 00、01、10 和 11 的四个变址寄存器组。为了简单起见，还进一步假定：寄存器 00 是确实存在的，并且，它可用于变址。

DRTRY:	ERR-1→AMPCR	(1)
	ERR-1→AMPCR	(2)
	LD-1→AMPCR	(3)
	ST-1→AMPCR	(4)
	BR-1→AMPCR	(5)
	TZ-1→AMPCR	(6)
	TL-1→AMPCR	(7)
	DEC-1→AMPCR	(8)
	INC-1→AMPCR	(9)
	SHF-1→AMPCR	(10)
IFETCH:	A1→MAR2	(11)
	MR2, A1+1→A1, BEX	(12)
	WHEN RDC THEN B C→A2	(13)
	10→SAR, 3→LIT	(14)
	A2 AND LIT→MAR	(15)
	MR1, B R→AMPCR	(16)
	12→SAR	(17)
	A2 R→A2	(18)
	6→SAR	(19)
	BEX, EXEC	(20)
	WHEN RDC THEN A2+B→A2, MAR2, JUMP	(21)
LD:	MR2, BEX	(22)
	IFETCH-1→AMPCR	(23)
	WHEN RDC THEN B→A3, JUMP	(24)
ST:	A3→MR	(25)
	MW2, IF SAI	(26)
	IFETCH-1→AMPCR	(27)
	WHEN SAI THEN JUMP	(28)
BR:	IFETCH-1→AMPCR	(29)
	A2→A1, RETN	(30)
TZ:	NOT A3→	(31)
	IFETCH-1→AMPCR	(32)
	IF ABT THEN A2→A1 RETN ELSE JUMP	(33)
TL:	A3→	(34)
	IFETCH-1→AMPCR	(35)
	IF LST THEN A2→A1 RETN ELSE JUMP	(36)
DEC:	MR2, B ₁₁₁ →A2, BEX	(37)
	IFETCH-1→AMPCR	(38)
	WHEN RDC THEN A2+B→MR	(39)
	MW2, IF SAI	(40)
	WHEN SAI THEN JUMP	(41)
INC:	MR2, BEX	(42)
	IFETCH-1→AMPCR	(43)
	WHEN RDC THEN 0+B+1→MR	(44)
	MW2, IF SAI	(45)
	WHEN SAI THEN JUMP	(46)
SHF:	IFETCH-1→AMPCR	(47)
	1→SAR	(48)
	A3 C→A3, JUMP	(49)

图 5-3 本章所描述的 S 机器仿真程序的微程序

最后，文字赋值的目录表(即跳转表)涉及一个其符号地址为 ERR 的错误微子程序。但在微程序中并没有介绍它。

微程序将在下节中讨论。

5-4 仿真程序的微程序的分析

总的来看，仿真程序的微程序(见图 5-3)的构造非常类似于用符号机器语言编制的普通的 S 程序。仿真程序由目录表、取指令子程序，以及各种指令的执行周期子程序所组成。S 机器的控制总是从取指令子程序开始并返回到该取指令子程序。

5-4-1 目录表

目录表列出如下：

DRTRY: ERR — 1 → AMPCR (1)

ERR — 1 → AMPCR (2)

LD — 1 → AMPCR (3)

ST — 1 → AMPCR (4)

BR — 1 → AMPCR (5)

TZ — 1 → AMPCR (6)

TL — 1 → AMPCR (7)

DEC — 1 → AMPCR (8)

INC — 1 → AMPCR (9)

SHF — 1 → AMPCR (10)

这种目录表是作为文字赋值语句表而存在的，并且，在仿真程序中，它用作执行周期微子程序的一种跳转表。每一个文字赋值语句在翻译成十六进制的 D 机器指令时，都要计算地址。因此，例如表达式 LD-1 给出 LD 的 MPM 地址减 1。若 LD 的 MPM 地址被赋以 21，则执行语句

LD — 1 → AMPCR

后，将把数值 20 送入 AMPCR。

应当强调,为了使文字赋值语句能把数据送入 AMPCR 中,这些语句必须直接由 D 机器执行。

我们已经知道,每条 I 型或者 II 型指令都是用微程序存储器(MPM)中的一条 M 指令来表示的。全部 II 型指令都存放在微程序存储器中,而 I 型指令还要求有一条相应的毫微指令。所以,仿真程序中的每个语句都对应着一条 M 指令。MPM 地址从 0 开始,向上递增。仿真程序中有 49 个语句,所以相应的 M 指令被安排在 MPM 从 0 号到 48 号的地址中。

由于特定的原因,就把目录表安排在微程序存储器的起始部分。试看仿真程序的语句(16)和(20)。数字操作码被用作进入跳转表的变址量。语句(16)分离出操作码字段,并且,由于目录表位于微程序存储器的起始部分,因此,不需要加基地址就能取得另一条微指令。当执行了语句(20)时,EXEC 命令就执行一个插入语句,其 MPM 地址为操作码加 1。在操作码为 1 的取数指令的场合,EXEC 语句执行微程序存储器中 2 号单元的文字赋值语句,即 LD - 1 → AMPCR 语句。

5-4-2 取指令子程序

取指令子程序列出如下:

IFETCH: A1 → MAR2 (11)

MR2, A1 + 1 → A1, BEX (12)

WHEN RDC THEN B C → A2 (13)

10 → SAR, 3 → LIT (14)

A2 AND LIT → MAR (15)

MR1, B R → AMPCR (16)

12 → SAR (17)

A2 R → A2 (18)

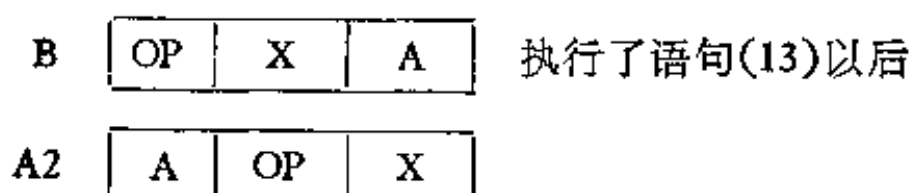
6 → SAR (19)

BEX, EXEC (20)

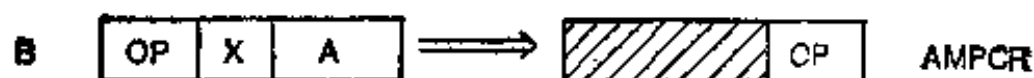
WHEN RDC THEN A2 + B → A2, MAR2, JUMP (21)

这个子程序用来读出下一条指令,修改现行地址寄存器,译码指令并分离它的各个字段,计算有效地址以及转移到所需要的执行周期子程序。

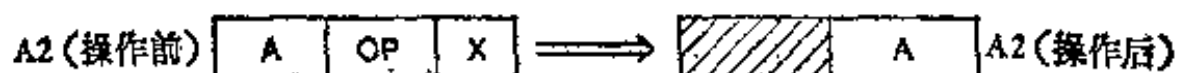
语句(11)为取下一条指令作准备:把下一条指令的地址(来自现行地址寄存器)送入 MAR2 寄存器以读 S 存储器。语句(12)启动读操作,并修改现行地址寄存器。当读操作完成时,语句(13)把 B 寄存器中的指令进行循环右移,以分离变址寄存器字段,于是, B 寄存器和 A2 寄存器为:



语句(15)把地址字段(A)和操作码字段(OP)屏蔽掉,并把变址寄存器编号送入 MAR。因此, MAR 的内容是 X,或者,用符号表示为 $(MAR) = X$ 。为了取得变址寄存器的内容,语句(16)启动 S 存储器的读操作(其地址为 $\langle BR1, MAR \rangle$),并把操作码字段分离送入 AMPCR 中,如下所示:



语句(18)分离出地址字段并送入 A2 中,如下所示:



在语句(20)中,变址寄存器的内容从外部设备接口送到了 B 寄存器,并且,用 EXEC 命令把与现行指令相应的微子程序的(地址-1)送入 AMPCR。根据 D 机器的定时关系,语句(20)的 BEX 要到语句(21)的 RDC 条件为“真”时才能完成。在产生这种情况时, B 寄存器仍存放着变址寄存器的内容,或者用符号表示为 $(B) = (X)$ 。回想到 A2 寄存器存放着指令的地址字段,为了形成有效地址,语句(21)把 A2 与 B 的内容相加,并把结果送入 A2 (作暂时存贮用) 和 MAR2 (用于下一次的取指令操作和写操作)。最后,

语句(21)转移到执行周期的微子程序。

5-4-3 取数子程序

取数子程序列出如下:

LD: MR2, BEX (22)

IFETCH - 1 \longrightarrow AMPCR (23)

WHEN RDC THEN B \longrightarrow A3, JUMP (24)

它取出操作数,并把操作数送入累加器,然后转移到取指令子程序。这个操作用符号表示为 $(A + (X)) \longrightarrow (\text{ACCUM})$ 。语句(22)完成 S 存贮器的读操作,其有效地址已由取指令子程序送入 MAR2 中。语句(24)把操作数送入累加器(即寄存器 A3),并且把控制转移到取指令子程序,以取出下一条指令,进而继续执行。

5-4-4 存数子程序

存数子程序列出如下:

ST: A3 \longrightarrow MIR (25)

MW2, IF SAI (26)

IFETCH - 1 \longrightarrow AMPCR (27)

WHEN SAI THEN JUMP (28)

它把累加器的内容存入有效地址单元中,即 $(\text{ACCUM}) \longrightarrow (A + (X))$,并转移到取指令子程序。语句(25)把累加器的内容送入 MIR,以便为 S 存贮器的写操作做准备。语句(26)把 SAI 条件复位,并启动写操作。当语句(28)的写操作完成时,控制转移到取指令子程序。

5-4-5 转移子程序

转移子程序列出如下:

BR: IFETCH - 1 \longrightarrow AMPCR (29)

A2 \longrightarrow A1, RETN (30)

它把有效地址送入现行地址寄存器,并返回到取指令子程序。在

语句(30)中,寄存器 A2 中的有效地址被送入现行地址寄存器 A1。为了跳过取指令子程序的第一个语句,就利用 RETN 命令返回到取指令子程序。这种方法省了一条指令的执行时间,因为有效地址(现在是新的现行地址)已由取指令子程序的最后一个语句送入 MAR2 中了。

5-4-6 测试零子程序

测试零子程序列出如下:

TZ: NOT A3 \longrightarrow (31)

IFETCH - 1 \longrightarrow AMPCR (32)

IF ABT THEN A2 \longrightarrow A1 RETN ELSE JUMP (33)

如果累加器(即 D 机器的寄存器 A3)中所有位都为 0,则上述测试零子程序就转移到有效地址。语句(31)把累加器的内容取反码送入加法器。取反是用“非”算符 NOT 完成的。如果加法器的各位是全 1,则意味着累加器的各位是全 0。在语句(33)中,测试全 1 条件(ABT),若为全 1,则执行如转移指令一样的过程。否则,用 JUMP 命令使控制转回到取指令子程序,并使 S 程序继续顺序执行在这条 TZ 指令后面的一条 S 机器指令。

5-4-7 测试最低位子程序

测试最低位子程序列出如下:

TL: A3 \longrightarrow (34)

IFETCH - 1 \longrightarrow AMPCR (35)

IF LST THEN A2 \longrightarrow A1 RETN ELSE JUMP (36)

如果累加器的最右位是 1,则上述测试最低位子程序就转移到有效地址。语句(34)把累加器(即 D 机器的寄存器 A3)的内容送入加法器。在语句(36)中,用 LST 条件来测试最低位。若该条件为“真”,就如同执行转移指令那样来处理转移。否则,则用 JUMP 命令使控制转回到取指令子程序,并使 S 程序继续顺序执行在这条 TL 指令后面的一条 S 机器指令。

5-4-8 减量和增量子程序

减量子程序列出如下:

DEC: MR2, B_{in} → A2, BEX (37)

IFETCH - 1 → AMPCR (38)

WHEN RDC THEN A2 + B → MIR (39)

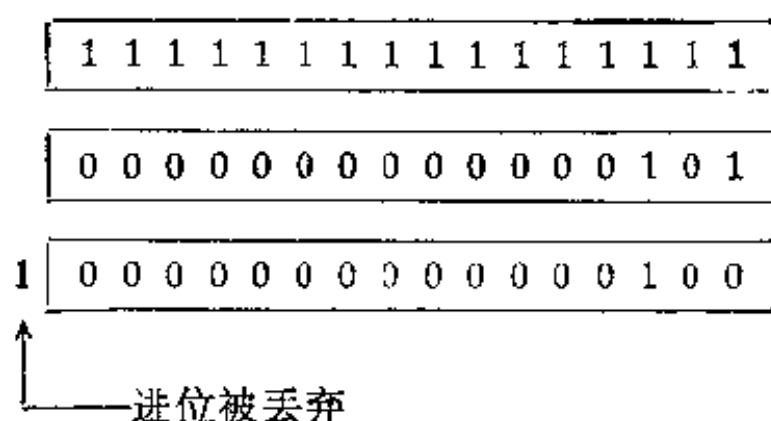
MW2, IF SAI (40)

WHEN SAI THEN JUMP (41)

它取出有效地址的内容, 把该内容减 1, 然后把结果存回 S 存贮器中的同一个地址单元。语句(37)启动读操作, 并且把全 1 的一个字送入 A2。在语句(39)中, 当完成读操作时, 就利用下述恒等式

$$B - 1 = B + (2^{16} - 1)$$

使 B 的内容减 1。例如, 若寄存器 B 的内容是 5, 则使用下列计算:



对于用语句(40)启动 S 存贮器的写操作而言, 加法器的操作结果被送入了 MIR。语句(40)中的 MW2 命令是无条件执行的, 并且, IF SAI 子句使 SAI 条件复位, 这个 SAI 条件可以是上次 S 存贮器的读操作或写操作留下的。当写操作完成时, 语句(41)使控制返回到取指令子程序。

增量子程序列出如下:

INC: MR2, BEX (42)

IFETCH - 1 → AMPCR (43)

WHEN RDC THEN 0 + B + 1 → MIR (44)

MW2, IF SAI (45)

WHEN SAI THEN JUMP (46)

上述增量子程序类似于减量子程序,不同的只是:在语句(44)中,B寄存器的内容被增量(即加1),而不是被减量(即减1)。之所以在语句(44)中使用了不常用的表达式 $0 + B + 1$,是因为需要一个X选择的操作数。

5-4-9 移位子程序

移位子程序列出如下:

SHF: IFETCH - 1 \longrightarrow AMPCR (47)

1 \longrightarrow SAR (48)

A3 C \longrightarrow A3, JUMP (49)

它把S机器的累加器(即D机器的寄存器A3)的内容循环右移1位。用语句(49)完成移位操作,然后微程序控制转回到取指令子程序。在移位操作前,用语句(48)将移位位数送到SAR,所以,移位微子程序的功能是完整的。这种情况提出了一个重要问题。若在语句(49)中的循环右移完成之前,SAR的内容已被改变了,则结果也将被改变。所以最好避免在微子程序的第一个语句中传送信息到SAR和LIT,而在开始先放一条I型指令。

词 汇

读者应当熟悉本章中所使用的下列术语:

基址寄存器 (Base register)

有效地址 (Effective address)

多次读 (Multiple read)

文字表 (Literal table)

跳转表 (Jump table)

文字赋值 (Literal assignment)

累加器 (Accumulator)

现行地址寄存器 (Current-address register)

指令寄存器 (Instruction register)

目录表 (Directory)

提 问

下面的一些问题打算用来测验你对主题内容的理解。所有的问题都可以直接从课文或者通过对所提出的主题进行逻辑推理而得到答案。有些问题适合于在讨论班上进行研究。

1. 试给出访问 S 存贮器的两点理由。
2. 试给出能够用作暂时存贮器的两种类型 (除 A1、A2、A3 和 B 寄存器外)。
3. 为什么必须执行跳转表中的一个表目?
4. 如果仅有一个 BR_i 寄存器, 编写仿真程序可能吗?
5. 在一次输入操作中, MR_i 与 BEX 命令完成什么功能?
6. 为什么要执行 EXEC 命令?
7. “两台机器的字长相同”的约定有什么意义?
8. 为什么说把 I 型指令作为微子程序的第一条指令是一种好办法?

习 题

1. 在仿真程序中添加一种可变长度的右移操作, 其中有效地址就是移位置。
2. 在仿真程序中添加一条加法指令 ADD, 其操作用符号表示如下:
 $(\text{ACCUM}) + (A + (X)) \longrightarrow (\text{ACCUM})$
3. 修改仿真程序, 使其目录表位于微程序的末尾。
4. 假定一台 S 机器的传送指令 MOVE 具有下列格式:

位 1	4 5	10 11	16
操作码 OP	地址 1 (A_1)	地址 2 (A_2)	

MOVE 指令的操作码是 0110, 这条指令用符号描述为: $(A_1) \longrightarrow (A_2)$ 。
试写出用以执行这条指令的仿真程序。

第六章 翻 译 程 序

6-1 概 述

本章简述如何使用 TRANSLANG 翻译程序，并介绍有关参考语言、符号硬件语言的语法的基础知识以及关于十六进制微编码的说明。

图 6-1 描述了翻译过程的概况。可以采用两种操作方式：

1. 符号微程序和控制信息一起从终端或卡片读入器输入；
2. 预先存贮好符号微程序，只有控制信息从终端或卡片读入器输入。

翻译程序可以在成批处理操作方式或者分时操作方式中使用。

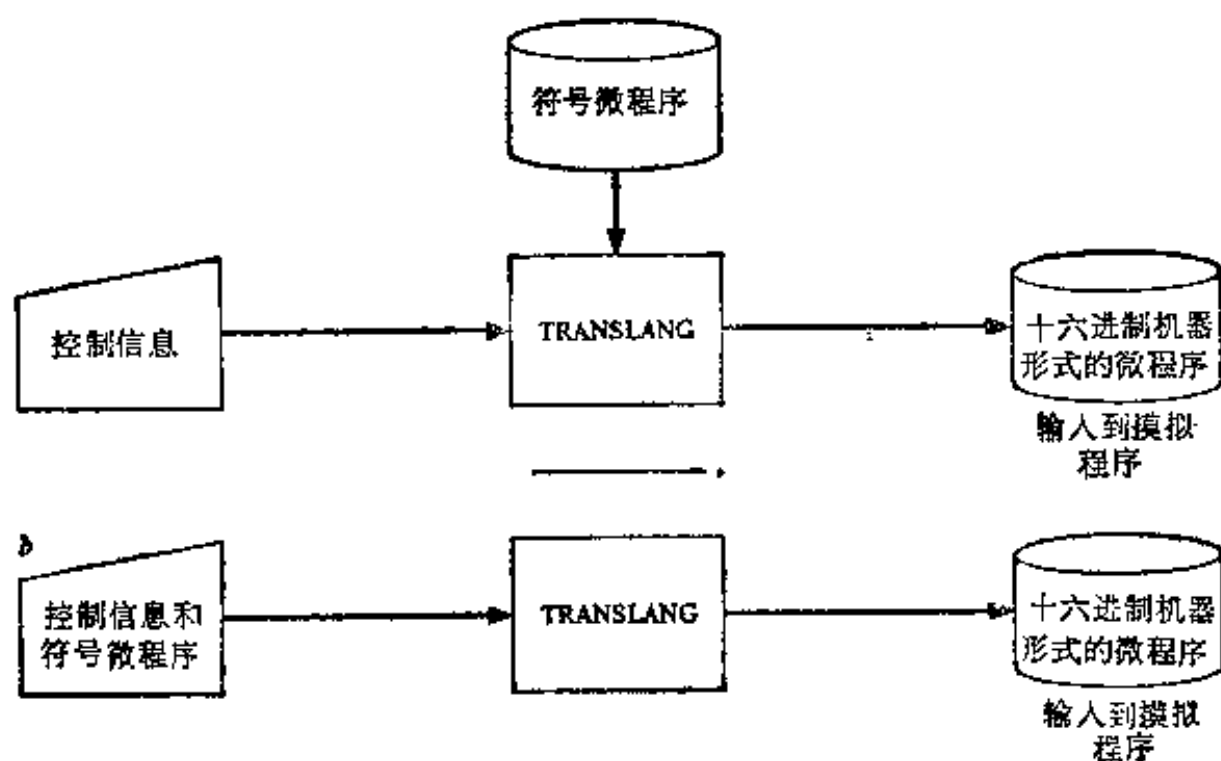


图 6-1 TRANSLANG 翻译过程的概况。

- (a) 预先存放的符号微程序，
(b) 符号微程序不预先存放

用。可是,为了说明用户与翻译程序之间的交互运用方式,列举的所有例子都是在分时操作方式中运行的。

短的微程序从终端输入,长的微程序要预先存放在分时系统中。而 TRANSLANG 系统能够支持这两种输入微程序的方式。

6-2 微程序设计语言

在前面几章中,已非正式地介绍了微程序设计的参考语言。本章将按 Backus Naur 范式 (BNF) 来叙述它。TRANSLANG 语言也只是简略地介绍过,这里也将更详细地加以介绍,使得用户在输入微程序时尽可能减少词法错误。

6-2-1 Backus Naur 范式

在讨论计算机语言时,常常需要区别语法上正确的语句与语法上不正确的语句。用来描写另一种语言的语言称为元语言。在计算机领域内,为了用计算机语言写程序,程序员在编写程序时,系统程序员在研制处理机时,都要使用元语言。

第一个众所周知的元语言是为了描述 ALGOL 60 而研制的 Backus Naur 范式 (BNF), 它至今还在被广泛地使用它。BNF 引用了三个元符号,并用这三个符号来构造语言的定义:

符号	意 义
::=	“被定义为”
	读作“或”
< >	尖括号, 它指示某些事件的名称。括号内的所有其他的符号都是文字, 并只代表它们自己。

通过使用这些元符号和某些操作约定,就可以描述一种语言。

作为一个例子,考察通常计算机中数的定义。计算机中典型的数是 45, 621.4, -57, 1.23E - 4, -.0003 和 .96E + 17。而在

BNF 中,一个数的定义是从描述它的基本组成部分开始的,并由此发展一种结构化的定义方法。在 BNF 中使用下列语句来描述一个数:

$\langle \text{数字} \rangle ::= 0|1|2|3|4|5|6|7|8|9$

$\langle \text{无符号整数} \rangle ::= \langle \text{数字} \rangle | \langle \text{无符号整数} \rangle \langle \text{数字} \rangle$

$\langle \text{整数} \rangle ::= \langle \text{无符号整数} \rangle | + \langle \text{无符号整数} \rangle | - \langle \text{无符号整数} \rangle$

$\langle \text{十进制小数} \rangle ::= . \langle \text{无符号整数} \rangle$

$\langle \text{指数部分} \rangle ::= E \langle \text{整数} \rangle$

$\langle \text{十进制数} \rangle ::= \langle \text{无符号整数} \rangle | \langle \text{十进制小数} \rangle | \langle \text{无符号整数} \rangle \langle \text{十进制小数} \rangle$

$\langle \text{无符号数} \rangle ::= \langle \text{十进制数} \rangle | \langle \text{十进制数} \rangle \langle \text{指数部分} \rangle$

$\langle \text{数} \rangle ::= \langle \text{无符号数} \rangle | + \langle \text{无符号数} \rangle | - \langle \text{无符号数} \rangle$

这种定义方法使用了左递归技术,如在无符号整数的定义中所说明的那样,现将其重复如下: $\langle \text{无符号整数} \rangle ::= \langle \text{数字} \rangle | \langle \text{无符号整数} \rangle \langle \text{数字} \rangle$ 。这表明,无符号整数是一个数字或一串数字。把连续列出的两个组成部分看成是并列的。

6-2-2 参考语言

微程序设计参考语言的描述将分成三部分: 高级部分、操作部分和低级部分。定义的方式是从高到低。这就是说,在定义其组成部分之前先提出该定义。

被定义的语句将按序编号,以供参考。

6-2-2-1 高级部分

一个微程序由一系列 I 型或 II 型指令所组成,指令的后面跟着一个 END 语句。

$\langle \text{程序} \rangle ::= \langle \text{程序本体} \rangle \langle \text{结束行} \rangle \quad (1)$

$\langle \text{程序本体} \rangle ::= \langle \text{程序行} \rangle | \langle \text{程序本体} \rangle \langle \text{程序行} \rangle \quad (2)$

$\langle \text{程序行} \rangle ::= \langle \text{注解行} \rangle | \langle \text{指令} \rangle \quad (3)$

$\langle \text{注解行} \rangle ::= \text{COMMENT} \langle \text{注解字符串} \rangle \quad (4)$

〈指令〉::=〈标号部分〉〈文字赋值〉|〈标号部分〉〈毫微指令〉 (5)

〈标号部分〉::=〈语句标号〉:|〈空格字符串〉 (6)

〈结束行〉::=END (7)

6-2-2-2 操作部分

操作部分包括文字赋值、毫微指令及其组成部分的定义。

6-2-2-2-1 文字赋值语句 文字赋值语句属于Ⅱ型,它在一个时钟间隔内执行,并将文字赋值送到被选定的D机器寄存器。

〈文字赋值〉::=〈文字结构〉|〈文字结构〉%〈注解字符串〉 (8)

〈文字结构〉::=〈文字〉→AMPCR|〈文字〉→SAR|〈文字〉→
〈文字寄存器〉|〈文字〉→SAR,〈文字〉→〈文
字寄存器〉|〈文字〉→〈文字寄存器〉,〈文字〉
→SAR (9)

〈文字〉::=〈整数〉|COMP〈整数〉|〈语句标号〉|〈语句标号〉-1
(10)

〈文字寄存器〉::=LIT|SLIT (11)

6-2-2-2-2 毫微指令语句 在两个或多个时钟间隔内执行的毫微指令是Ⅰ型指令,它被用来规定D机器的操作功能。

〈毫微指令〉::=〈无条件部分〉〈条件部分〉|〈无条件部分〉〈条件部
分〉%〈注解字符串〉 (12)

〈无条件部分〉::=〈组成元素表〉 (13)

〈组成元素表〉::=〈组成元素〉|〈组成元素表〉,〈组成元素〉|〈空〉
(14)

〈组成元素〉::=〈外部操作〉|〈逻辑操作〉|〈后继命令〉 (15)

〈外部操作〉::=〈存贮器设备操作〉|〈置位操作〉|〈存贮器设备操
作〉,〈置位操作〉|〈置位操作〉,〈存贮器设备操
作〉|〈空〉 (16)

〈存贮器设备操作〉::=MR1|MR2|MW1|MW2 (17)

〈置位操作〉::=SET〈条件修正位〉 (18)

〈条件修正位〉::=LC1|LC2|LC3 (19)

- $\langle \text{逻辑操作} \rangle ::= \langle \text{加法器操作} \rangle \langle \text{移位操作} \rangle \langle \text{目标表} \rangle \quad (20)$
- $\langle \text{加法器操作} \rangle ::= 0 | 1 | \langle \text{一元操作}^* \rangle | \langle \text{二进制操作} \rangle | \langle \text{空} \rangle \quad (21)$
- $\langle \text{一元操作} \rangle ::= \langle \text{非} \rangle \langle \text{X 选择} \rangle | \langle \text{非 Y 选择} \rangle \quad (22)$
- $\langle \text{非} \rangle ::= \text{NOT} | \langle \text{空} \rangle \quad (23)$
- $\langle \text{X 选择} \rangle ::= 0 | A1 | A2 | A3 | \text{CTR} | \text{LIT} | \langle \text{空} \rangle \quad (24)$
- $\langle \text{非 Y 选择} \rangle ::= \langle \text{非} \rangle \langle \text{Y 选择} \rangle \quad (25)$
- $\langle \text{Y 选择} \rangle ::= 0 | 1 | B | B \langle m \rangle \langle c \rangle \langle i \rangle | \text{CTR} | \text{LIT} | \text{AMPCR} \quad (26)$
- $\langle m \rangle ::= 0 | 1 | T | F \quad (27)$
- $\langle c \rangle ::= 0 | 1 | T | F \quad (28)$
- $\langle i \rangle ::= 0 | 1 | T | F \quad (29)$
- $\langle \text{二进制操作} \rangle ::= \langle \text{X 选择} \rangle \langle \text{操作符} \rangle \langle \text{非 Y 选择} \rangle | \langle \text{X 选择} \rangle + \langle \text{非 Y 选择} \rangle + 1 | \langle \text{X 选择} \rangle - \langle \text{非 Y 选择} \rangle - 1 \quad (30)$
- $\langle \text{操作符} \rangle ::= \text{AND} | \text{OR} | \text{NAN} | \text{NOT} | \text{IMP} | \text{NIM} | \text{RIM} | \text{NRI} | \text{XOR} | \text{EQV} | + | - | \text{OAD} | \text{ADD} \quad (31)$
- $\langle \text{移位操作} \rangle ::= R | L | C | \langle \text{空} \rangle \quad (32)$
- $\langle \text{目标表} \rangle ::= \rightarrow | \rightarrow \langle \text{目标} \rangle | \langle \text{目标} \rangle | \langle \text{目标表} \rangle, \langle \text{目标} \rangle \quad (33)$
- $\langle \text{目标} \rangle ::= A1 | A2 | A3 | \text{MIR} | \text{BR1} | \text{BR2} | \text{AMPCR} | \langle \text{输入 B} \rangle | \langle \text{输入 CTR} \rangle | \langle \text{输入 MAR} \rangle | \langle \text{输入 SAR} \rangle \quad (34)$
- $\langle \text{输入 B} \rangle ::= B | \text{BEX} | \text{BAD} | \text{BMI} | \text{BBE} | \text{BBA} | \text{BBI} \quad (35)$
- $\langle \text{输入 CTR} \rangle ::= \text{CTR} | \text{LCTR} | \text{INC} \quad (36)$
- $\langle \text{输入 MAR} \rangle ::= \text{MAR} | \text{MAR1} | \text{MAR2} | \text{LMAR} \quad (37)$
- $\langle \text{输入 SAR} \rangle ::= \text{SAR} | \text{CSAR} \quad (38)$
- $\langle \text{后继命令} \rangle ::= \text{WAIT} | \text{STEP} | \text{SKIP} | \text{SAVE} | \text{CALL} | \text{EXEC} | \text{JUMP} | \text{RETN} \quad (39)$
- $\langle \text{条件部分} \rangle ::= \langle \text{IF 子句} \rangle \langle \text{THEN 子句} \rangle \langle \text{ELSE 子句} \rangle | \langle \text{IF 子句} \rangle | \langle \text{WHEN 子句} \rangle \langle \text{THEN 子句} \rangle | \langle \text{空} \rangle \quad (40)$
- $\langle \text{IF 子句} \rangle ::= \text{IF} \langle \text{条件} \rangle \quad (41)$
- $\langle \text{条件} \rangle ::= \langle \text{非} \rangle \langle \text{基本条件} \rangle \quad (42)$

*1 一元操作指只用一个操作数的操作。——译者注

〈基本条件〉:: = LST|MST|AOV|ABT|COV|SAI|RDC| 〈条件修正位〉 (43)

〈THEN 子句〉:: = THEN〈组成元素表〉 (44)

〈ELSE 子句〉:: = ELSE 〈后继命令〉|〈空〉 (45)

〈WHEN 子句〉:: = WHEN 〈条件〉 (46)

6-2-2-3 低级部分

低级部分由组成微程序的其他结构组成,如标号、常数和注解等.

〈注解字符串〉:: = 〈注解字符〉|〈注解字符串〉〈注解字符〉 (47)

〈语句标号〉:: = 〈字母〉|〈语句标号〉〈字母〉|〈语句标号〉〈数字〉 (48)

〈整数〉:: = 〈数字〉|〈整数〉〈数字*〉 (49)

〈字母〉:: = A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z (50)

〈数字〉:: = 0|1|2|3|4|5|6|7|8|9 (51)

〈注解字符〉:: = 计算机字母表中的任意一个字符 (52)

〈空〉:: = 空的字符串 (53)

前面几章唯一未介绍的结构是注解,它可以是带关键字 COMMENT 的一个单独的输入行,也可以与一条 I 型或 II 型指令列在同一行上(借用百分比符号(%)把指令与注解部分分开).

6-2-3 TRANSLANG 语言

TRANSLANG 语言是一种机器可以识别和可以处理的微程序设计参考语言.这种语言在本书的前面已定义和使用过.本课程的目的是教会你进行微程序设计,而 TRANSLANG 语言则是在计算机上实行微程序的一种方式.

* 原文为 〈integer〉 :: = 〈digit〉|〈integer〉 〈digit〉, 根据定义(51), 这里的“整数”实际上指无符号整数.——译者注

参考语言

```

LST: A-1→AMPCR
BLOCK1-1→AMPCR
FOX→AMPCR
INIT: AMPCR→BR1
3000→AMPCR, 10→LIT
0→A1, LMAR, SAVE
LOOP: A1+1→A1 % BODY OF MPPCG STARTS HERE
MRI, INC, BEX
WHEN RDC THEN B R→A3, MIR
16→SAR
MW1, IF SAI
A3 AND B C→A3
6→SAR 20→LIT
COMMENT CHECK FOR WRITE COMPLETION, CLEANUP, AND RETURN
WHEN SAI THEN LCTR, BMI
A2 EQV B→CSAR
SET LC2, IF ABT THEN NOT A3→A2, JUMP ELSE RETN
END

```

TRANSLANG 语言

```

LST: A = 1 = AMPCR $
BLOCK1 = 1 = AMPCR $
FOX = AMPCR $
INIT: AMPCR = BR1 $
3000 = AMPCR, 10 = LIT $
0 = A1, LMAR, SAVE $
LOOP: A1 + 1 = A1 $ BODY OF MPPCG STARTS HERE
MRI, INC, BEX $
WHEN RDC THEN B R = A3, MIR $
16 = SAR $
MW1, IF SAI $
A3 AND B C = A3 $
6 = SAR, 20 = LIT $
COMMENT CHECK FOR WRITE COMPLETION, CLEANUP, AND RETURN $
WHEN SAI THEN LCTR, BMI $
A2 EQV B =, CSAR $
SET LC1, IF ABT THEN NOT A3 = A2, JUMP ELSE RETN
END $
READY

```

图 6-2 微程序设计参考语言和 TRANSLANG 语言所使用的词法规范比较

6-2-3-1 字符集

运行 TRANSLANG 程序需要有一个最低限度的字符集。除了 26 个字母和 10 个数字外，还需要下列专用字符：+ - = . , \$。

6-2-3-2 词法规范

对于参考语言的每一种结构,都存在着一种 TRANSLANG 词法规范。图 6-2 列出了同一个微程序的参考语言形式和 TRANSLANG 形式。从程序设计的观点来看,虽然这个例子意义不大,但是它代表了在微程序设计中出现的大多数情况(如果不是全部情况的话)。表 6-1 中概括了两种语言之间在印刷上的差别。TRANSLANG 的词法规范概括如下:

- 1.除了不能把空格嵌入到关键字或者语句标号以外,TRANSLANG 处理程序的输入形式是比较随便的。空格有重要意义,如下面所述。
- 2.语句标号(若有的话)与语句之间用句点隔开,句点之后至少要有一个空格。
- 3.等号(=)用作替换符号,它代替了 \longrightarrow ;等号前后至少要各有一个空格。
- 4.所有操作符、关键字或者其他语言符号的前面至少要有一个空格,而后面至少要有一个空格,或者后面必须有一个逗号和至少一个空格。跟有空格的逗号相当于空格。

表 6-1 参考语言和 TRANSLANG 之间在印刷上的差异

参 考 语 言	TRANSLANG	字 符 功 能
:	.	隔开语句标号与语句本身
\longrightarrow	=	指定赋值
%	\$	隔开注解部分与语句本身
COMMENT	COMMNT	注解行的关键字
B _{mc}	BMCL	选通 B 寄存器的门(例如 B _{00.1} B001)

注:在 TRANSLANG 中,所有的语句必须用一个 \$ 表示结束,\$ 的前面有一个空格字符。

- 5.所有语句都必须用一个美元符号(\$)表示结束,美元符号前面至少有一个空格。
- 6.在一个语句行中,注解字符串可以跟在美元符号的后面。
- 7.注解行必须使用关键字 COMMNT。

如果由于微程序员疏忽而违反上述规范之一,则 TRANSLANG 处理程序会产生相应的诊断信息。

6-2-3-3 BNF 的说明

如同参考语言一样, TRANSLANG 的介绍也分成三类: 高级部分, 操作部分和低级部分。参考语言和 TRANSLANG 之间本质上的差别是词法。

6-2-3-3-1 高级部分 用 TRANSLANG 写成的微程序由一系列的 I 型或 II 型指令组成, 最后跟着一个 END 语句。每一个语句都必须用一个美元符号 \$ 表示结束, 而美元符号前面是一个空格。

〈程序〉::=〈程序本体〉〈结束行〉 (1)

〈程序本体〉::=〈程序行〉|〈程序本体〉〈程序行〉 (2)

〈程序行〉::=〈注解行〉|〈指令〉 (3)

〈注解行〉::=COMMNT 〈注解字符串〉 (4)

〈指令〉::=〈标号部分〉〈文字赋值〉|〈标号部分〉〈毫微指令〉 (5)

〈标号部分〉::=〈语句标号〉·|〈空字符串〉 (6)

〈结束行〉::=END 串 (7)

6-2-3-3-2 操作部分 操作部分由文字赋值指令和毫微指令组成。空格字符的用法不作说明, 因为对每个需要空格的位置都作说明将使语法说明不必要地复杂化。关于空格的用法, 读者可以参考前一节中所给出的词法规范。

文字赋值语句

〈文字赋值〉::=〈文字结构〉\$|〈文字结构〉\$〈注解字符串〉 (8)

〈文字结构〉::=〈文字〉= AMPCR |〈文字〉= SAR |〈文字〉=〈文字寄存器〉|〈文字〉= SAR, 〈文字〉=〈文字寄存器〉|〈文字〉=〈文字寄存器〉, 〈文字〉= SAR (9)

〈文字〉::=〈整数〉|COMP 〈整数〉|〈语句标号〉|〈语句标号〉- 1

(10)

〈文字寄存器〉::=LIT|SLIT (11)

毫微指令语句

〈毫微指令〉::=〈无条件部分〉〈条件部分〉\$|〈无条件部分〉〈条件部分〉\$〈注解字符串〉 (12)

〈无条件部分〉::=〈组成元素表〉 (13)

〈组成元素表〉::=〈组成元素〉|〈组成元素表〉,〈组成元素〉|〈空〉 (14)

〈组成元素〉::=〈外部操作〉|〈逻辑操作〉|〈后继命令〉 (15)

〈外部操作〉::=〈存贮器设备操作〉|〈置位操作〉|〈存贮器设备操作〉,〈置位操作〉|〈置位操作〉,〈存贮器设备操作〉|〈空〉 (16)

〈存贮器设备操作〉::=MR1|MR2|MW1|MW2 (17)

〈置位操作〉::=SET〈条件修正位〉 (18)

〈条件修正位〉::=LC1|LC2|LC3 (19)

〈逻辑操作〉::=〈加法器操作〉〈移位操作〉〈目标表〉 (20)

〈加法器操作〉::=0|1|〈一元操作〉|〈二进制操作〉|〈空〉 (21)

〈一元操作〉::=〈非〉〈X选择〉|〈非Y选择〉 (22)

〈非〉::=NOT|〈空〉 (23)

〈X选择〉::=0|A1|A2|A3|CTR|LIT|〈空〉 (24)

〈非Y选择〉::=〈非〉〈Y选择〉 (25)

〈Y选择〉::=0|1|B|B〈m〉〈c〉〈l〉|CTR|LIT|AMPCR (26)

〈m〉::=0|1|T|F (27)

〈c〉::=0|1|T|F (28)

〈l〉::=0|1|T|F (29)

〈二进制操作〉::=〈X选择〉〈操作符〉〈非Y选择〉|〈X选择〉+〈非Y选择〉+1|〈X选择〉-〈非Y选择〉-1 (30)

〈操作符〉::=AND|OR|NAN|NOT|IMP|NIM|RIM|NRJ|XOR|EQV|+|-|OAD|ADD (31)

- 〈移位操作〉::= R|L|C|〈空〉 (32)
- 〈目标表〉::=|=|<目标>|<目标>|<目标表>, <目标> (33)
- 〈目标〉::= A1|A2|A3|MIR|BR1|BR2|AMPCR|〈输入 B〉|〈输入 CTR〉|〈输入 MAR〉|〈输入 SAR〉 (34)
- 〈输入 B〉::= B|BEX|BAD|BMI|BBE|BBA|BBI (35)
- 〈输入 CTR〉::= CTR|LCTR|INC (36)
- 〈输入 MAR〉::= MAR|MAR1|MAR2|LMAR (37)
- 〈输入 SAR〉::= SAR|CSAR (38)
- 〈后继命令〉::= WAIT|STEP|SKIP|SAVE|CALL|EXEC|JUMP|RETN (39)
- 〈条件部分〉::= <IF 子句> <THEN 子句> <ELSE 子句>| <IF 子句>|<WHEN 子句> <THEN 子句>|<空> (40)
- 〈IF 子句〉::= IF <条件> (41)
- 〈条件〉::= <非><基本条件> (42)
- 〈基本条件〉::= LST|MST|AOV|ABT|COV|SAI|ROC|<条件修正位> (43)
- 〈THEN 子句〉::= THEN <组成元素表> (44)
- 〈ELSE 子句〉::= ELSE <后继命令>|<空> (45)
- 〈WHEN 子句〉::= WHEN <条件> (46)
- 6-2-3-3-3 低级部分 低级部分描述形成语句的基本结构.
- 〈注解字符串〉::= <注解字符>|<注解字符串><注解字符> (47)
- 〈语句标号〉::= <字母>|<语句标号><字母>|<语句标号><数字> (48)
- 〈整数〉::= <数字>|<整数><数字> (49)
- 〈字母〉::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z (50)
- 〈数字〉::= 0|1|2|3|4|5|6|7|8|9 (51)
- 〈注解字符〉::= 计算机字母表中(不包括\$)的任何一个字符 (52)
- 〈空〉::= 空的字符串 (53)

在 TRANSLANG 中,语句标号的最大长度是六个字符.

6-3 执行翻译程序

翻译程序是用来产生能被模拟程序执行的十六进制微编码,它的运行过程包括下面两个主要步骤:

1. 符号微程序的准备与处理;
2. 规定要求的处理选择方式.

符号微程序的输入可以通过运行实例来说明,它包括下列微程序(用参考语言表示):

```
0 → A1, SAVE  
LOOP: A1 and B → A2  
2 → SAR  
IF LST THEN MR1, BEX ELSE JUMP  
WHEN RDC THEN JUMP  
END
```

此处,这个微程序只用来说明各种有用的语句类型.

6-3-1 处理说明

对于 TRANSLANG 处理程序可以用四种处理说明:

1. 用十六进制微编码来书写文件名,以便将其加载到 D 机器的模拟程序中去.
2. 在源程序表中,是否应当在每个语句之后产生该语句的二进制模式 (bit pattern).
3. 当完成符号微程序的处理时,是否应当产生十六进制的微编码表.
4. 输入是否从预先存贮的源文件中读出.

上述第一条中,即十六进制的文件名取决于所用的计算机和操作系统,它可以指出从不加标号的磁带到直接存取存贮器上的一个已命名的文件. 实例运行已在 Dartmouth 分时系统中做过,

该系统使用已命名的磁盘文件。第二条中，每个语句的二进制模式是一些枯燥繁琐的二进制数字，它表示翻译过程的结果。除了某些特殊情况外，它很少有实际意义。同样在第三条中，十六进制的微编码表也是枯燥繁琐的，这种表列程序读起来很麻烦。但是，要强调的是，除注解行以外，每一个源语句都被表示成十六进制编码。最后一条说明涉及到符号微程序的来源问题，如果是从卡片输入，那么源程序能够驻留在输入作业流中；如果使用终端输入，那么长的微程序应当预先存贮，因为正确地输入一个长微程序是有困难的。

6-3-2 运行实例

图 6-3 至图 6-6 提供了翻译程序的一些运行实例，这些实例

```

>>>> TRANSLANG D MACHINE MICROTRANSLATOR

ENTER FILENAME FOR HEX? F63HEX
SUPPRESS BIT PATTERNS?(1=SUPPRESS)? 1
SUPPRESS HEX LISTING?(1=SUPPRESS)?
IF INPUT IN FILE ENTER 1?

0000 ? 0 = A1, SAVE S
0001 ? LOOP. A1 AND B R = A2 S
0002 ? 2 = SAR S
0003 ? IF LST THEN MRI, BEX ELSE JUMP S
0004 ? WHEN RDC THEN JUMP S
0005 ? END S

THE TOTAL NUMBER OF ERRORS = 0

HEX TRANSLATION
OUTPUT READY FOR SIMULATOR INPUT

ADDR  NPN      NANG
0      F000 0012 0000 4000 0000
1      F001 0009 AC56 A000 0000
2      0200
3      F002 5E0C 0000 0C00 0800
4      F003 A820 0000 0000 0000
5      4000

EXECUTE (Y/N)? N

```

图 6-3 TRANSLANG 处理程序的运行实例。
规定选择方式：(1)无二进制模式；(2)十六进制表列；(3)源程序未曾预先存贮

TRANSLANG D MACHINE MICROTRANSLATOR

ENTER FILENAME FOR HEX? F64HEX
SUPPRESS BIT PATTERNS?(1=SUPPRESS)?
SUPPRESS HEX LISTING?(1=SUPPRESS)?
IF INPUT IN FILE ENTER 1?

0000 ? 0 = A1. SAVE \$
N=0000 000 000 010 010 000 00000000 0000 00 100 0000 00 0000 0000 0000
MPM=1111000000000000

0001 ? LESSP. A1 AND B R = A2 \$
N=0000 000 000 001 001 101 01100010 1011 01 010 0000 00 0000 0000 0000
MPM=1111000000000001

0002 ? 2 = SAR \$
MPM=0000001000000000

0003 ? IF LST THEN MRI. BEX ELSE JUMP \$
N=0101 111 000 001 100 000 00000000 0000 00 000 1100 00 0000 0000 0010
MPM=1111000000000010

0004 ? WHEN RDC THEN JUMP \$
N=1010 100 000 100 000 000 00000000 0000 00 000 0000 00 0000 0000 0000
MPM=1111000000000011

0005 ? END \$

THE TOTAL NUMBER OF ERRORS = 0

EXECUTE (Y/N)? N

图 6-4 TRANSLANG 处理程序的运行实例。规定选择方式：(1)要求产生二进位模式；(2)不产生十六进制表列；(3)源程序未曾预先存贮

使用了能在 Dartmouth 分时系统 (DTSS)¹⁾ 上运行的 TRANSLANG 处理程序。介绍有关 DTSS 的资料只限于便于理解这些例子。所用的 DTSS 命令如下：

命 令	说 明
OLD 名	把一个旧程序或数据文件送入用户工作区
RUN	执行程序
NEW 名	用来产生新程序或数据文件
SAVE	把文件名(数据文件或程序)存入 DTSS 目录
UNSAVE	删去一个已存入的文件

1) 应当强调，翻译程序和模拟程序是标准的 FORTRAN IV 程序，并能在任何一台具有 FORTRAN IV 编译程序的计算机上运行。

```

NEW F1665
READY

BUILD
SPEAK!

      0 = A1, SAVE $
LOOP. A1 AND B R = A2 $
      2 = SAR $
      IF LST THEN MRI, BEX ELSE JUMP $
      WHEN RDC THEN JUMP $
      END $
READY
LIST
SAVE
READY

```

图 6-5 用 BUID 命令构造一个符号微程序文件(即源文件)。LIST 给出一个文件表列,而 SAVE 则把文件名列入系统目录

```

>>>>> TRANSLANG D MACHINE MICROTRANSLATOR

ENTER FILENAME FOR HEX? F66HEX
SUPPRESS BIT PATTERNS?(1=SUPPRESS)? 1
SUPPRESS HEX LISTING?(1=SUPPRESS)?
IF INPUT IN FILE ENTER 1? 31
ENTER SOURCE FILENAME? F1665

0000      0 = A1, SAVE $
0001  LOOP. A1 AND B R = A2 $
0002      2 = SAR $
0003      IF LST THEN MRI, BEX ELSE JUMP $
0004      WHEN RDC THEN JUMP $
0005      END $

THE TOTAL NUMBER OF ERRORS =      0

HEX TRANSLATION
OUTPUT READY FOR SIMULATOR INPUT

ADDR  MPH      NANO
0      F000 0012 0000 4000 0000
1      F001 0009 AC56 A000 0000
2      0200
3      F002 5E0C 0000 0C00 0800
4      F003 A820 0000 0000 0000
5      4000

EXECUTE (Y/N)? N

```

图 6-6 TRANSLANG 处理程序的运行实例。规定选择方式: (1) 不要求产生二进位模式; (2) 产生十六进制表列; (3) 预先存贮源程序

?	用来提醒用户输入
BUILD	用来构造一个无顺序号的文件
LIST	在终端上列出一个文件

在图 6-3 中, 实例程序从终端输入, 不需要产生二进制模式, 但需要产生十六进制的(微编码)表列。读者应当注意到, 每个 TRANSLANG 语句的前面有一个问号(?), 它提醒用户输入语句。

在图 6-4 中, 输入同一程序, 可是, 在这种情况下, 要求打印二进制模式, 但不要求产生十六进制(微编码)表列。

在图 6-5 中, 源程序文件是用 BUILD 命令来构造的, 并且分别用 LIST 和 SAVE 命令加以列出和存贮。它将是下一张图(图 6-6)用的微程序源文件。

在图 6-6 中, TRANSLANG 处理程序是针对一个预先存贮的符号微程序来运行的。当指定选择一个预先存贮的源文件时, 则 TRANSLANG 处理程序提醒用户引出源文件名, 如图所示。

6-4 十六进制微编码

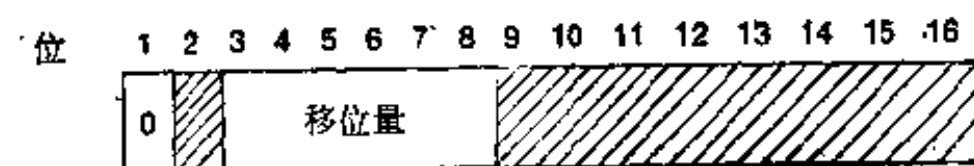
初看起来, 十六进制微编码似乎不大好读, 本节的目的是要通过描述与刻划几个文字赋值语句和毫微指令来简化这种概念。

6-4-1 文字赋值语句的刻划

考察下面的文字赋值语句:

2 → SAR

这个语句的 II 型指令是:



它的二进制模式是

MFM = 0000001000000000

相应的十六进制表示为:

0 2 0 0

同样,对文字赋值语句

COMP 3 \longrightarrow SAR, 5 \longrightarrow LIT

它具有如下的指令格式:

位	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	1	0														

其二进位模式是:

MPM \leftarrow 1 0 1 1 1 1 0 1 0 0 0 0 0 1 0 1

相应的十六进制表示为:

BD05

可是,当使用下列形式的文字赋值语句

<语句标号> - 1 \longrightarrow AMPCR

时,则需要稍作一些处理,为此考察语句

LKR - 1 \longrightarrow AMPCR

在下列微程序中的作用:

MPM 地址	语 句
0	LKR - 1 \longrightarrow AMPCR
1	MR1, BEX
2	LKR: A1 + 1 \longrightarrow A1
3	WHEN RDC JUMP
...	...

该文字赋值语句的形式为:

位	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	1	1	0													

本来指望语句 LKR - 1 \longrightarrow AMPCR 把“1”放入指令的地址字段。可是,指令的二进位模式是

$$\text{MPM} = 1100000000000000$$

这条指令的格式是正确的而地址字段却不正确。产生这种情况的原因是，在第一遍扫描程序时就产生了这样的二进位模式。而要在第一遍扫描完成后，才装配有关的地址字段，并使十六进制微编码构成一条完整的指令。因此，语句 $\text{LKR} - 1 \rightarrow \text{AMPCR}$ 的十六进制微编码是

C001

在 TRANSLANG 中，唯一由程序员定义的结构是语句标号；所有的选择都是在扫描一遍程序的时候判定的。

6-4-2 毫微指令语句的刻划

每一条 I 型指令都导致 TRANSLANG 处理程序产生一个微指令字和一个毫微指令字。下面的微程序(未必是高效率的)把值 1 到 5 分别写入 S 存贮器的 1000 至 1004 号单元：

```
INIT: AMPCR  $\rightarrow$  A1, MAR2 % A1 CONTAINS S MEMORY
      ADDRESS
      1000  $\rightarrow$  AMPCR
      1  $\rightarrow$  A2, MIR LCTR % A2 CONTAINS VALUE
      4  $\rightarrow$  LIT
      SAVE
LOOP: MW2, IF SAI
      A1 + 1  $\rightarrow$  A1, MAR2, INC
      WHEN SAI THEN STEP
      A2 + 1  $\rightarrow$  A2, MIR, IF NOT COV THEN JUMP
      ELSE STEP
      END
```

另外，这个微程序还用来说明毫微指令的十六进制微编码。程序翻译的印出结果示于图 6-7。考察微程序中的第一个语句：

$\text{AMPCR} \rightarrow \text{A1}, \text{MAR2}$

由此产生下面的二进位模式为：

```

>>>> TRANSLANG D MACHINE MICROTRANSLATOR
ENTER FILENAME FOR HEX? F67HEX
SUPPRESS BIT PATTERNS?(1=SUPPRESS)?
SUPPRESS HEX LISTING?(1=SUPPRESS)?
IF INPUT IN FILE ENTER 1? 31
ENTER SOURCE FILENAME? FIG67

0000 INIT. AMPCR = 'A1, MAR2 $ A1 CONTAINS $ MEMORY ADDRESS
N=0000 000 000 001 001 000 00110010 0000 00 100 0000 00 0111 0000 0000
MPM=111100000000000000

0001 1000 = AMPCR $
MPM=1100001111101000

0002 1 = A2, MIR, LCTR $ A2 CONTAINS VALUE
N=0000 000 000 001 001 000 00000110 0000 00 010 0000 10 0000 0100 0000
MPM=11110000000000001

0003 4 = LIT $
MPM=11100000000000100

0004 SAVE $
N=0000 000 000 010 010 000 00000000 0000 00 000 0000 00 0000 0000 0000
MPM=11110000000000010

0005 LOOP. MW2, IF SAI $
N=1001 100 000 001 001 000 00000000 0000 00 000 0000 00 0000 0000 0111
MPM=11110000000000011

0006 A1 + 1 = A1, MAR2, INC $
N=0000 800 000 001 001 101 00000110 0000 00 100 0000 00 0111 1000 0000
MPM=11110000000000100

0007 WHEN SAI THEN STEP $
N=1001 100 000 001 000 000 00000000 0000 00 000 0000 00 0000 0000 0000
MPM=11110000000000101

0008 A2 + 1 = A2, MIR, IF NOT C0V THEN JUMP ELSE STEP $
N=1000 000 000 100 001 110 00000110 0000 00 010 0000 10 0000 0000 0000
MPM=11110000000000110

0009 END $

THE TOTAL NUMBER OF ERRORS = 0

HEX TRANSLATION
OUTPUT READY FOR SIMULATOR INPUT

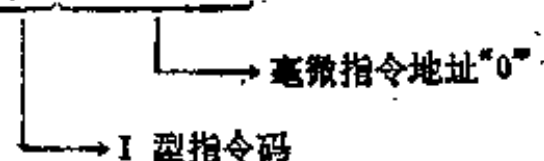
ADDR MPM NAME
0 F000 3009 0640 401C 0000
1 C3E8
2 F001 0009 00C0 2081 0000
3 E004
4 F002 0012 0000 0000 0000
5 F003 9809 0000 0000 1C00
6 F004 0009 A0C0 401E 0000
7 F005 9808 0000 0000 0000
8 F006 8021 C0C0 2080 0000
9 4000

EXECUTE (Y/N)? N

```

图 6-7 说明源微指令的二进制模式和十六进制微编码的微程序的翻译

MPM=1111000000000000



N = 0000 000 000 001 001 000 00110010 0000
00 100 0000 00 0111 0000 0000

相应的十六进制微编码为:

MPM=1111000000000000

F000

N=0000 000 000 001 001 000 00110010 0000 00 100 0000 00 0111 0000 0000

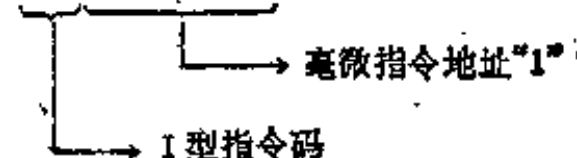
0009 0840 401C 0000 (十六进制)

作为另一个例子,考察语句

1 → A2, MIR, LCTR

由此产生下面的二进制模式:

MPM=1111000000000001



N = 0000 000 000 001 001 000 00000110 0000 00 010
0000 10 0000 0100 0000

相应的十六进制微编码为:

MPM=1111000000000001

F001

N=0000 000 000 001 001 000 00000110 0000 00 010 0000 10 0000 0100 0000

0009 00CD 2081 0000 (十六进制)

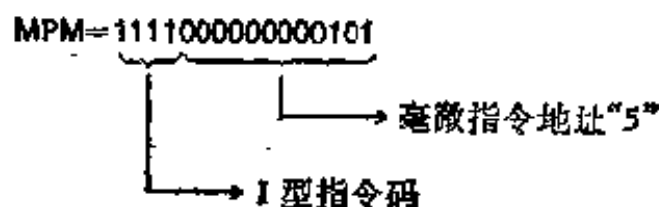
作为第三个例子,考察下面的条件语句:

WHEN SAI THEN STEP

它等价于

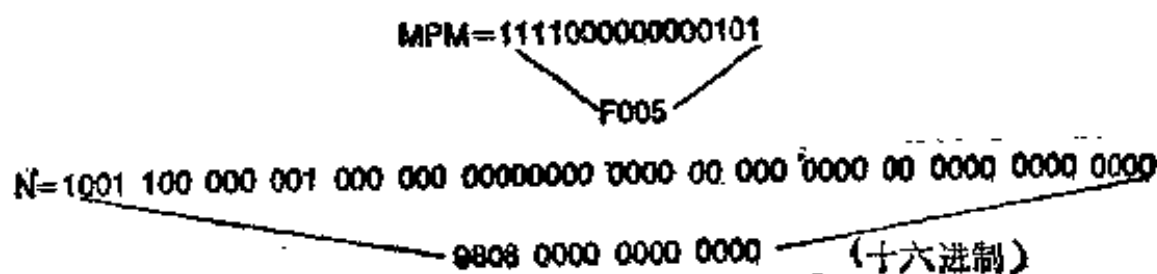
IF SAI THEN STEP ELSE WAIT

经翻译后,针对该语句产生下面的二进制模式:



N = 1001 100 000 001 000 0000 0000 0000 0000 00 000
0000 00 0000 0000 0000

对应的十六进制微编码是:



作为最后一个例子,考察语句

A2 + 1 → A2, MIR, IF NOT COV THEN JOMP ELSE STEP
它说明了无条件成分和条件成分。MPM 指令简单明了且有如下格式:

MPM = 111100000000110

毫微指令需要对下列可选项加以置位:

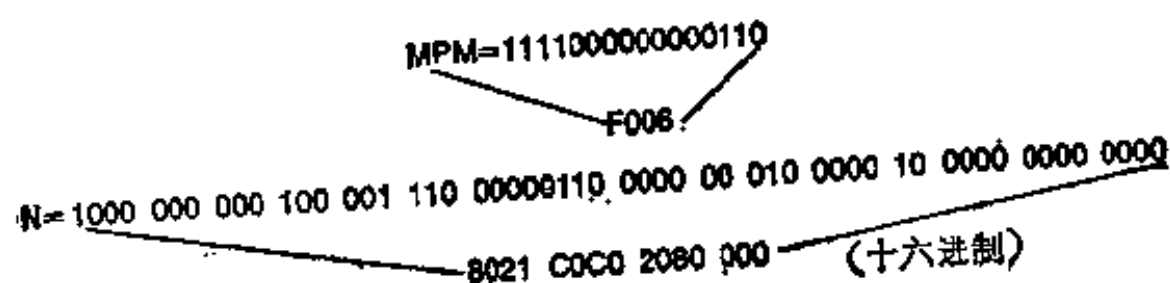
1. NOT 条件;
2. 无条件逻辑部件操作;
3. 真和假的后继命令;
4. 逻辑部件操作。

这些可选项可以用下面的毫微指令来说明:

N = 1000 000 000 100 001 110 00000110 0000 00 010

0000 10 0000 0000 0000

它等价于下面的十六进制微编码:



现在可以更明确地叙述微程序设计的性质。在 MPM 或毫微指令中的每一位都会引起处理机以特定的方式加以反应或动作。微程序设计的目标是把若干条微指令组成一种有意义的序列，而微程序设计语言则简化了这一过程。

词 汇

读者应当熟悉本章中所使用的下列术语

符号微程序 (Symbolic microprogram)

十六进制微编码 (Hexadecimal microcode)

Backus Naur 范式 (Backus Naur form)

元语言 (Metalanguage)

参考语言 (Reference language)

字符集 (Character set)

TRANSLANG (TRANSLANG)

二进位模式 (Bit patterns)

十六进制(微编码)表列 (Hexadecimal Listing)

预先存贮的微程序 (Prestored microprogram)

提 问

1. 计算机语言的语法描述是干什么用的?
2. 微程序设计参考语言和 TRANSLANG 之间有什么差异? 试把它们列出来并具体说明。
3. 用什么方法能够利用二进位模式和十六进制微编码来改善微程序的效率?
4. 在语句 $LOOP - 1 \rightarrow AMPCR$ 中,其二进位模式为什么会不正确?

5. 考虑一条毫微指令与数 54 不是 4 的倍数这个事实, 怎样从毫微指令的二进位模式推出十六进制微编码?

习 题

1. 给出〈组成元素表〉的六种正确的情况。
2. 给出〈文字结构〉的三种正确的情况。
3. 针对在图 6.7 中所给出的下列语句, 描述与刻划其二进位模式和十六进制微编码。

```
SAVE  
MW2, IF SAI  
A1 + 1 → A1, MAR2, INC
```

4. 给出下列语句的二进制位模式

```
9 → SAR  
113 → LIT, COMP14 → SAR  
21 → SLIT
```

5. 给出、描述和刻划下列语句的二进位模式和十六进制微编码。

```
WHEN RDC THEN A2 + B → A2, MAR2, JUMP  
MR2, A1 + 1 → A1, BEX  
MR1, B R → AMPCR  
IF ABT THEN A2 → A1 RETN ELSE JUMP
```


第七章 模拟程序

7-1 概 述

本章叙述如何使用D机器,如何为了执行而装入S存贮器,以及如何理解模拟程序运行的结果。利用D机器的模拟程序执行微程序的过程如图7-1所示。由于D机器的模拟程序是一种模拟程序,因此计算机运行时的打印输出是极重要的。因为微程序中不包含有通常指令中的变量或存贮器访问,因此它的打印输出集中于下面两项:

1. 在微程序执行期间,机器中各寄存器的内容;
2. 执行微程序前后,S存贮器的内容。

大多数处理选择都与这两种打印结果有关。采用打印输出的另一个理由是:由于微程序固有的复杂性,它常常需要调整,而为了这种调整就需要有有关微程序执行的详细资料。

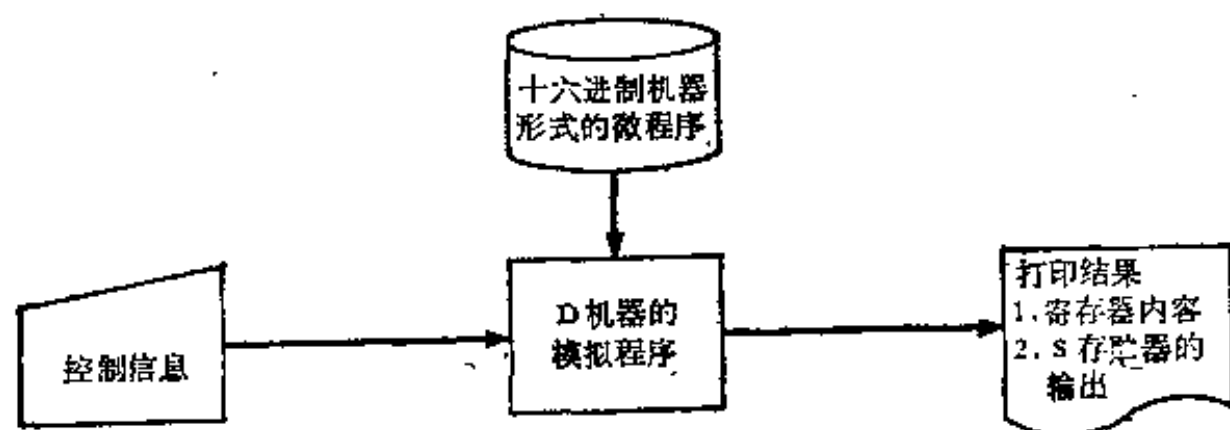


图 7-1 使用D机器的模拟程序执行微程序的过程的概况

7-2 运行模拟程序

D机器模拟程序的输入(见图7-1)是一个文件,它包含有十

六进制的微编码以及控制信息,控制信息从终端或读卡机输入.输入必要的信息是为了回答下列请求:

- (1) ENTER SAME FILENAME FOR HEX?...
(输入同一个用十六进制微编码表示的文件名吗? ...)
- (2) OUTPUT REGISTERS AND S MEMORY IN INTEGER
(1) OR OCTAL(2)?...
(以整数还是以八进制数的形式输出
寄存器和 S 存储器的内容? ...)
- (3) INPUT S MEMORY IN INTEGER(1) OR OCTAL IN 011
FORMAT(2) ?...
(以整数还是以八进制的011格式输入到 S 存储器? ...)
- (4) STARTING ADDRESS = ?...
(起始地址=? ...)
- (5) MAXIMUM NUMBER OF CLOCKS TO SIMULATE = ?...
(模拟的最大时间=? ...)
- (6) NUMBER OF CLOCKS BETWEEN OUTPUT POINTS =
?...
(两个输出点之间的时间=? ...)
- (7) ENTER OUTPUT LINES DESIRED 1-ADDRESSES AND
CLOCK 2-A1, A2, A3, B 3-MIR, SAR, LIT, CTR, AMPCR
4-BR1, BR2, MAR, BMAR, GC1, GC2 5-CONDITIONS
ENTER NUMBER OF OUTPUT LINES DESIRED = ?...
(装入需要输出的行: 1-地址和时钟数; 2-A1, A2, A3, B;
3-MIR, SAR, LIT, CTR, AMPCR; 4-BR1, BR2, MAR,
BMAR, GC1, GC2 5-条件
要求装入的输出行的数目=? ...)
- (8) BEGIN OUTPUT AT MPM ADDRESS = ?...
(开始输出的 MPM 地址=? ...)
END OUTPUT AT MPM ADDRESS = ?...
(结束输出的 MPM 地址=? ...)

ENTER 1 FOR S MEMORY DUMP WHEN PROGRAM
TERMINATES?...

(当程序结束时,要为 S 存储器的输出而记入 1 吗? ...)

- (9) ENTER S MEMORY VALUES IN CONSECUTIVE BLOCKS
ENTER 9999 FOR STARTING ADDRESS WHEN FINISHED
STARTING S MEMORY ADDRESS = ?...

FINAL S MEMORY ADDRESS FOR THIS BLOCK = ?...

S MEM(...)= ?...

(把 S 存储器的值装入到连续的存储块中)(当完成操作时,
装入用作启动地址的 9999)

(S 存储器的(现行)启动地址=?)

(本存储块的 S 存储器的末地址=? ...)

(S MEM(...)= ?...)

- (10) MEMORY DUMP REQUESTED ENTER VALUES AS
DONE IN MEMORY INPUT(9999 FOR STARTING ADD-
RESS WHEN FINISHED)

(存储器输出请求)(装入输入到 S 存储器一样的值(当完成
时 9999 用作启动地址))

STARTING S MEMORY ADDRESS = ?...

(S 存储器的启动地址=? ...)

FINAL S MEMORY ADDRESS FOR THIS BLOCK = ?...

(本存储块的 S 存储器的末地址=? ...)

问句左侧的编号对应于图 7-2 到 7-4 中的打印输出。 在下
面几节中,将一一介绍这些问句。

7-2-1 微程序执行的例子

列在图 7-2 中的实例微程序说明如下:

寄存器 BR1 的内容将指示存放在 S 存储器中的 16 个
定点寄存器。 定点寄存器 2 的内容则指示 S 存储器中具
有三个元素的一个线性数组(通过改变计数器中的值,微

程序对任何大小的数组都可操作。长度 3 用来控制打印输出的长度)。微程序把数组的元素相加，并把结果送入定点寄存器 3。

下面列出用参考语言写的程序：

```
INIT: 0 → BR1, LMAR
      2 → LIT
      MR1, BEX, LCTR
      3 → LIT
      CHK-1 → AMPCR
      WHEN RDC THEN B → A1 MAR2, INC, CALL
ADD:  WHEN RDC THEN BEX
      A3 + B → A3
      A1 + 1 → A1, MAR2
CHK:  IF NOT COV THEN MR2, INC, JUMP
      A3 → MIR, LMAR
      3 → LTT
      MW1, IF SAI
      WHEN SAI THEN STEP
      END
```

寄存器占用 S 存贮器的前 16 个单元，所以，在置初始状态的子程序中，BR1 置成 0。寄存器 A1 用来指示数组的下一个元素，寄存器 A3 保存运行中的和数。

7-2-2 文件名

为了回答下列询问，要装入含有十六进制微编码的文件名：

ENTER SAME FILENAME FOR HEX?...

在微程序执行之前，把它的十六进制微编码从指定的文件装入到微程序存贮器 (MPM) 和毫微存贮器。装入两种控制存贮器的方式都是从 0 单元开始并向后扩展的。

***** TRANSLANG D MACHINE MICROTRANSLATOR

ENTER FILENAME FOR HEX? F72HEX
 SUPPRESS BIT PATTERNS?(1=SUPPRESS)? 1
 SUPPRESS HEX LISTING?(1=SUPPRESS)? 1
 IF INPUT IN FILE ENTER FY 31
 ENTER SOURCE FILENAME? FIG72

```
0000  INIT. 0 = BRT, LMAR 3
0001      2 = LIT 3
0002      MR1, BEX, LCTR 3
0003      3 = LIT 3
0004      CHK = 1 = AMPCR 3
0005      WHEN RDC THEN B = A1, MAR2, INC, CALL 3
0006  ADD, WHEN RDC THEN BEX 3
0007      A3 + 2 = A3 3
0008      A1 + 1 = A1, MAR2 3
0009  CHK, IF NOT CBV THEN MR2, INC, JUMP 3
0010      AS = RTR, LMAR 3
0011      3 = LIT 3
0012      HWT, IF SAI 3
0013      WHEN SAI THEN STEP 3
0014      END 3
```

THE TOTAL NUMBER OF ERRORS = 0

EXECUTE (Y/N)? N

3) DMACH1 (COMPILED) 07 JAN '76 20:08

ENTER SAME FILENAME FOR HEX? F72HEX
 OUTPUT REGISTERS AND S MEMORY IN INTEGER(1) OR OCTAL(2)? 1
 INPUT S MEMORY IN INTEGER(1) OR OCTAL IN 611 FORMAT(2)? 1
 STARTING ADDRESS =? 0
 MAXIMUM NUMBER OF CLOCKS TO SIMULATE=? 30
 NUMBER OF CLOCKS BETWEEN OUTPUT PRINTS=? 1

ENTER OUTPUT LINES DESIRED 1-ADDRESSES AND CLOCK
 2- A1,A2,A3,B 3- MR1,SAR,LIT,CTR,AMPCR 4- BRT,MR2,MAR,BMAR,GC1,GC2
 5- CONDITIONS
 ENTER NUMBER OF OUTPUT LINES DESIRED? 5

BEGIN OUTPUT AT NPM ADDRESS=? 14
 END OUTPUT AT NPM ADDRESS=? 14
 ENTER 1 FOR S MEMORY DUMP WHEN PROGRAM TERMINATES? 1

ENTER S MEMORY VALUES IN CONSECUTIVE BLOCKS
 ENTER 9999 FOR STARTING ADDRESS WHEN FINISHED

STARTING S MEMORY ADDRESS=? 2
 FINAL S MEMORY ADDRESS FOR THIS BLOCK=? 3
 MEM(2)=? 100
 MEM(3)=? 0

STARTING S MEMORY ADDRESS=? 100
 FINAL S MEMORY ADDRESS FOR THIS BLOCK=? 102
 MEM(100)=? 17
 MEM(101)=? 9
 MEM(102)=? 24

STARTING S MEMORY ADDRESS=? 9999

END OF SIMULATION - REGISTERS CONTAIN

P(1) ADDR. = 13 P(3) ADDR. = 13 CLOCK = 25
A1 = 103 A2 = 0 A3 = 50 B = 24
MIR = 50 SAR = 0 LIT = 3 CTR = 0 AMPCR = 5
ERI = 0 BR2 = 0 MAR = 3 BMAR = 3 GC1 = 0 GC2 = 0
LC1 = 0 LC2 = 0 MST = 0 LST = 0 ABT = 0 ADV = 0 CBV = 0 SAI = 1 RDC = 1 INT = 0

MEMORY DUMP REQUESTED

ENTER VALUES AS DONE IN MEMORY INPUT (9999 FOR STARTING ADDRESS WHEN FINISHED)

STARTING S MEMORY ADDRESS = ? 2

FINAL S MEMORY ADDRESS FOR THIS BLOCK = ? 3

S MEMORY(2) TO S MEMORY(3) =

100 50
STARTING S MEMORY ADDRESS = ? 9999

图 7-2 为说明模拟程序的执行而设计的微程序实例

7-2-3 输出格式

在微程序执行期间，可在各种时间间隔打印输出各机器寄存器的内容，以使用户确定微程序执行时的状态。下面的询问使用户能够规定输出格式的类型，这种类型最适于要开始执行的程序：

OUTPUT REGISTERS AND S MEMORY IN INTEGER

(1) OR OCTAL (2)?...

D机器的数据寄存器(即 A1, A2, A3, B 以及 MIR) 和 S 存储器单元，可以用八进制的 011 格式或者用(被删弃了前面诸 0 的)整数格式打印出来。一般地，在处理非数值字时，优先采用的是八进制格式，如指令或屏蔽码。而对于数据值则优先采用整数格式，因为这样就无需为验证计算结果而进行数值变换了。

7-2-4 S 存储器的输入

大多数微程序的执行都涉及到使用 S 存储器和一个用来保存 S 机器指令、S 机器寄存器和数据的存储区。S 存储器总是由模拟程序预置为 0，并且，置 S 存储器的初始状态(如果需要的话)

一般也是由微程序中的一个置初始状态子程序来完成的。然而，用机器语言写的 S 程序或大量数组必须在执行前送入 S 存储器。下面的询问可用来规定输入格式：

INPUT S MEMORY IN INTEGER (1) OR OCTAL 011
FORM AT (2)?...

对这询问的回答仅指定输入格式的类型，S 存储器的输入不在这时进行，而是在回答后一个询问，并在要开始执行微程序前进行。

7-2-5 起始地址

把微程序装入 MPM 和毫微存储器时，都是从 0 号单元开始的。对询问

STARTING ADDRESS = ?...

的回答允许指定第一条 M 指令的 MPM 地址。因此，就象前面已给出的仿真程序的实例那样，可以要求在一个文字赋值表之后开始执行微程序。起始地址是按十进制整数值指定的。

7-2-6 模拟的最大时钟数

用回答下面的询问来规定执行一个微程序的时间范围：

MAXIMUM NUMBER OF CLOCKS TO SIMULATE = ?...

时间范围是按照时钟间隔的个数来定义的。它按十进制整数形式输入并根据每条指令的“平均”时钟数（它的变化范围在 1 到 2 之间）来计算。经验表明，采用每条指令平均为 1.5 个时钟值是较好的。

因为经常会遇到微程序循环，所以“最大时钟数”应根据所执行的算法，仔细地加以计算。

7-2-7 两个输出点之间的时钟数

只有在细调时，才需要在每一个时钟间隔内打印出所有的寄存器的内容；否则，无需产生大量的打印输出。询问

NUMBER OF CLOCKS BETWEEN OUTPUT POINTS = ?...

允许改变两个输出点之间的间隔。这种间隔用十进制整数来规定,并且它仅仅控制打印输出。实际上,模拟程序总是要经过每一个 D 机器周期的,而不管输出规定如何。

7-2-8 输出行

下面的询问将控制每个打印周期的输出量:

ENTER OUTPUT LINES DESIRED

1-ADDRESSES AND CLOCK

2-A1, A2, A3, B

3-MIR, SAR, LIT, CTR, AMPCR

4-BR1, BR2, MAR, BMAR, GC1, GC2

5- CONDITIONS

ENTER NUMBER OF OUTPUT LINES DESIRED?...

回答应当是 1 到 5 之间(包括 1 与 5)的一个十进制数。若装入的是 5,则每个打印周期打印所有的输出,如图 7-2 所示。若装入的是 2 到 4 (包括 2 与 4),则还要作形如下述的辅助询问:

ENTER LINE NUMBERS SEPARATED BY COMMAS = ?...

并且,用户必须指定究竟哪些行应当被打印。

7-2-9 输出说明表

在调试微程序时,常会把错误隔离在一特定的微子程序内,故可以只要求打印该微子程序内的寄存器和条件。在下面的询问中

BEGIN OUTPUT AT MPM ADDRESS = ?...

END OUTPUT AT MPM ADDRESS = ?...

按十进制整数装入的两个值,决定了要产生打印输出的指令——就象通过回答上述两个询问的方式来控制打印一样。在执行上述 MPM 地址以外(包括 MPM 地址)的指令时,将不启动打印输出。

跟在上述询问后面的一个询问为:

ENTER 1 FOR S MEMORY DUMP WHEN PROGRAM
TERMINATES?...

如果回答这个询问时装入的是 1, S 存储器的输出方式已经装入, 并且微程序的执行已经完成, 则意味着用户请求 S 存储器的输出说明表。否则, 执行将自动终止, 并且不进行 S 存储器输出。

7-2-10 S 存储器的装入

S 存储器是通过下面一组交互询问来预置的:

ENTER S MEMORY VALUES IN CONSECUTIVE BLOCKS

ENTER 9999 FOR STARTING ADDRESS WHEN FINISHED

STARTING S MEMORY ADDRESS = ?...

FINAL S MEMORY ADDRESS FOR THIS BLOCK = ?...

在装入上述信息后, 模拟程序以下述形式的询问:

$S\text{MEM}(\text{addr}) = ?$

提醒用户为指定存储块中的每个 S 存储器单元装入信息。例如在下面的例子中:

$S\text{MEM}(1234) = ? \underline{10470000000}$

其中, 下面划线的数字是用户对询问的回答。

当 S 存储器的输入选择规定为整数格式时, 则回答 S 存储器的输入询问是装入一个简单的十进制整数, 例如:

$S\text{MEM}(1637) = ? \underline{163}$

然而, 当选用八进制 (011) 格式时, 则必须装入正好十一位八进制数字。当装入 32 位的二进制数时, 则前面需预加一个 0 位, 使得字长是 3 的倍数, 这样得到的 33 位二进制数将按 3 位分组, 并转换成八进制。因此, 下面的 32 位二进制数

0	11001010011100101110111001010011
└─┘	→ 预加的 0 位

将被装入 75 号单元, 以回答某个适当的询问, 例如下述询问:

$S\text{MEM}(75) = ? \underline{31234567123}$

信息可以按存储块的形式装入, 这样就不必装入“0”。因为

整个 S 存储器已被预置成“0”了。当所有的信息存储块都已经装入时，用户在打字机上打 9999 作为起始地址，模拟程序就自动开始执行。

7-2-11 S 存储器的输出

在由模拟程序启动的微程序开始执行后，一旦执行完了规定的时钟间隔数，微程序就可以正常结束。如果不要求 S 存储器输出，则在打印完寄存器内容和条件状态之后，就立即结束。如果已经要求 S 存储器输出，则产生下面的询问：

MEMORY DUMP REQUESTED

ENTER VALUES AS DONE IN MEMORY INPUT (9999
FOR STARTING ADDRESS WHEN FINISHED)

STARTING S MEMORY ADDRESS — ?...

FINAL S MEMORY ADDRESS FOR THIS BLOCK — ?...

并且按“输出格式”询问中所规定的格式来产生 S 存储器的输出。

一般地，S 存储器的输出是判断微程序正确性的主要方法。正如执行通常的 S 程序那样，由仿真程序把计算结果存入 S 存储器，而 S 存储器的输出则是得以访问这种信息的唯一方法。

7-3 在微程序执行期间的打印输出

在微程序执行期间，由模拟程序所产生的打印输出，在输出类型和数量上是由执行前装入的参数来控制的。图 7-3 给出一组输出行的实例，这些输出行采用整数格式（该输出是从图 4.1 的例子中选出的）。最多可以打印五行，分别称为：

1. 程序控制行。
2. 逻辑部件寄存器行。
3. 第一控制寄存器行。
4. 第二控制寄存器行。
5. 条件行。

下面,我们将分别介绍每一行.

7-3-1 时钟的一般描述

在每个时钟间隔内,一条指令总是先经过相位 1^{*)}. 对于 I 型指令来说,相位 1 执行指令译码和条件测试. 对于 II 型指令来说,则相位 1 还包括执行完整条指令. 并且,在同一个时钟间隔内,另一条 I 型指令可以处于正在完成的过程中.

因此,把完成 I 型指令的相位称为“相位 3”. 相位 3 分成两部分: 逻辑部件操作部分和目标部分. 例如在语句

$$A1 + 1 \longrightarrow B$$

中,运算“ $A1 + 1 \longrightarrow$ ”称为操作部分,它表示“把 A1 的内容和‘1’在加法器中相加”. 把结果送入 B 寄存器则是目标部分. 逻辑部件的操作总是在相位 1 的时钟间隔之后的下一个时钟间隔内完成的. 然而,目标部分直到开始下一次逻辑部件操作时才完成. 相位 2 是保持相位. 当一条 I 型指令由于下一条指令没有逻辑操作而无法完成时才产生相位 2. (顺便提一下,这正是 LIT 寄存器, SAR 或者 AMPCR 为什么可以用 I 型指令使它用这些寄存器后面的指令来装入的原因.) 下面的例子说明没有保持相位的情况.

指令编号	微 程 序	相 位	时钟
1	$A1 + B \longrightarrow A2$	指令 1 的相位 1	1
2	$A2 + 1 \longrightarrow AB$	指令 1 的相位 3 指令 2 的相位 1	2
3	$A3 + B \longrightarrow A1$	指令 2 的相位 3 指令 3 的相位 1	3

但是,在下面的情况中,由于一条插入的 II 型指令导致无逻辑部件操作,使得 I 型指令不能完成. 所以,需要增加保持相位:

指令编号	微 程 序	相 位	时钟
1	$A1 + B R \longrightarrow A2$	指令 1 的相位 1	1

^{*)} 它相应于一个时钟间隔(或时钟周期). ——译者注

2	3 \longrightarrow SAR, 2 \longrightarrow LIT	指令 1 的相位 2	2
		指令 2 的相位 1	
3	A2 \longrightarrow MIR, LMAR	指令 1 的相位 3	3
		指令 3 的相位 1	

在下面的情况中,由于条件不是“真”的,因此也需要保持相位:

指令编号	微 程 序	相 位	时钟
1	A1 + 1 \longrightarrow A2	指令 1 的相位 1	1
2	IF ABT THEN A2 \longrightarrow B SKIP ELSE STEP (假定条件是假的)	指令 1 的相位 2 指令 2 的相位 1	2
3	B _{TFF} \longrightarrow MIR	指令 1 的相位 3 指令 3 的相位 1	3

在这个情况中,第二条指令(即 IF ABT THEN A2 \longrightarrow B) 没有相位 2 或相位 3. 这是因为逻辑部件操作是有条件的,而条件又是假的. 最后,在 IF 语句:

指令编号	微 程 序	相 位	时钟
1	A1 + 1 \longrightarrow A2	指令 1 的相位 1	1
2	A2 \longrightarrow B IF ABT THEN SKIP ELSE STEP(假定条件是“假”)	指令 1 的相位 3 指令 2 的相位 1	2
3	B _{TFF} \longrightarrow MIR	指令 2 的相位 3 指令 3 的相位 1	3

无条件的逻辑部件操作导致完或前面的诸操作,并导致产生 IF 语句,不管 IF 语句的条件是“真”还是“假”,都需要相位 3.

有关定时的另一些说明将在第十章给出.

7-3-2 程序控制行

程序控制行给出正在通过各种执行相位的微指令的 MPM 地址. 它也给出现行时钟的计数值. 在下面的例子中给出各个项目的具体含义:

```

2) DMACHI (COMPILED) 07 JAN 76 19:11
ENTER SAME FILENAME FOR HEX7 F4IHEX
OUTPUT REGISTERS AND S MEMORY IN INTEGER(1) OR OCTAL(2)? 1
INPUT S MEMORY IN INTEGER(1) OR OCTAL IN 011 FORMAT(2)? 1
STARTING ADDRESS=? 0
MAXIMUM NUMBER OF CLOCKS TO SIMULATE=? 10
NUMBER OF CLOCKS BETWEEN OUTPUT POINTS=? 1

ENTER OUTPUT LINES DESIRED 1-ADDRESSES AND CLOCK
2- A1,A2,A3,B 3- MIR,SAR,LIT,CTR,AMPCR 4- BR1,BR2,MAR,BMAR,GC1,GC2
5- CONDITIONS
ENTER NUMBER OF OUTPUT LINES DESIRED? 5

BEGIN OUTPUT AT MPM ADDRESS=? 0
END OUTPUT AT MPM ADDRESS=? 4
ENTER 1 FOR S MEMORY DUMP WHEN PROGRAM TERMINATES? 1

```

```

ENTER S MEMORY VALUES IN CONSECUTIVE BLOCKS
ENTER 9999 FOR STARTING ADDRESS WHEN FINISHED

```

```

STARTING S MEMORY ADDRESS=? 1234
FINAL S MEMORY ADDRESS FOR THIS BLOCK=? 1234
SMENC(1234)=? 0

```

```

STARTING S MEMORY ADDRESS=? 9999

```

```

P(1) ADDR. = 0 P(3) ADDR. = -1 CLOCK = 1
A1 = 0 A2 = 0 A3 = 0 B = 0
MIR = 0 SAR = 0 LIT = 0 CTR = 0 AMPCR = 0
BR1 = 0 BR2 = 0 MAR = 0 BMAR = 0 GC1 = 0 GC2 = 0
LC1 = 0 LC2 = 0 MST = 0 LST = 0 ABT = 0 ABV = 0 CBV = 0 SAI = 0 RDC = 0 INT = 0

```

```

P(1) ADDR. = 1 P(2) ADDR. = 0 CLOCK = 2
A1 = 0 A2 = 0 A3 = 0 B = 0
MIR = 0 SAR = 0 LIT = 0 CTR = 0 AMPCR = 1234
BR1 = 0 BR2 = 0 MAR = 0 BMAR = 0 GC1 = 0 GC2 = 0
LC1 = 0 LC2 = 0 MST = 0 LST = 0 ABT = 0 ABV = 0 CBV = 0 SAI = 0 RDC = 0 INT = 0

```

```

P(1) ADDR. = 2 P(3) ADDR. = 0 CLOCK = 3
A1 = 0 A2 = 0 A3 = 0 B = 0
MIR = 1234 SAR = 0 LIT = 0 CTR = 0 AMPCR = 1234
BR1 = 4 BR2 = 0 MAR = 210 BMAR = 1234 GC1 = 0 GC2 = 0
LC1 = 0 LC2 = 0 MST = 0 LST = 0 ABT = 0 ABV = 0 CBV = 0 SAI = 0 RDC = 0 INT = 0

```

```

P(1) ADDR. = 3 P(3) ADDR. = 2 CLOCK = 4
A1 = 0 A2 = 0 A3 = 0 B = 0
MIR = 1234 SAR = 0 LIT = 0 CTR = 0 AMPCR = 1234
BR1 = 4 BR2 = 0 MAR = 210 BMAR = 1234 GC1 = 0 GC2 = 0
LC1 = 0 LC2 = 0 MST = 0 LST = 0 ABT = 0 ABV = 0 CBV = 0 SAI = 0 RDC = 0 INT = 0

```

```

P(1) ADDR. = 3 P(3) ADDR. = 3 CLOCK = 5
A1 = 0 A2 = 0 A3 = 0 B = 0
MIR = 1234 SAR = 0 LIT = 0 CTR = 0 AMPCR = 1234
BR1 = 4 BR2 = 0 MAR = 210 BMAR = 1234 GC1 = 0 GC2 = 0
LC1 = 0 LC2 = 0 MST = 0 LST = 0 ABT = 0 ABV = 0 CBV = 0 SAI = 1 RDC = 0 INT = 0

```

END OF SIMULATION - REGISTERS CONTAIN

P(1) ADDR. = 3 P(3) ADDR. = 3 CLOCK = 0
A1 = 0 A2 = 0 A3 = 0 B = 0
MIR = 1234 SAR = 0 LIT = 0 CTR = 0 AMPCR = 1234
BR1 = 4 BR2 = 0 MAR = 210 BHAR = 1234 GC1 = 0 GC2 = 0
LC1 = 0 LC2 = 0 MST = 0 LST = 0 ABT = 0 ABV = 0 CBV = 0 SAI = 1 RDC = 0 INT = 0

MEMORY DUMP REQUESTED

ENTER VALUES AS DONE IN MEMORY INPUT (9999 FOR STARTING ADDRESS WHEN FINISHED)

STARTING S MEMORY ADDRESS = 7 1234
FINAL S MEMORY ADDRESS FOR THIS BLOCK = 7 1234

S MEMORY(1234) TO S MEMORY(1234) =

1234
STARTING S MEMORY ADDRESS = 7 9999

图 7-3 在微程序执行期间,按整数格式产生的输出行实例

<u>P(1)ADDR. = 23</u>	<u>P(3)ADDR. = 22</u>	<u>CLOCK = 47</u>
正在通过相位 1 的指令的 MPM 地 址为 23	正在执行指令中的 相位 3 周期. 该指 令的 MPM 地址 为 22	这是本次运行中 所执行的第 47 个 时钟间隔

保持相位是用类似的方法来表示的:

<u>P(1)ADDR. = 75</u>	<u>P(2)ADDR. = 74</u>	<u>CLOCK = 256</u>
正在通过相位 1 的指令的 MPM 地址为 75	正在执行指令中 的相位 2 周期. 该指令的 MPM 地址为 74	这是本次运行 中所执行的第 256 个时钟间 隔

MPM 地址总是按十进制整数给出.

总括起来可以说,在处理机的每个时钟间隔,将执行一条微指令中的相位 1 周期以及另一条微指令中的相位 2 或相位 3 周期.

2) DNACHI (COMPILED) 07 JAN 76 19:19

ENTER SAME FILENAME FOR HEX? F4IHEX
OUTPUT REGISTERS AND S MEMORY IN INTEGER(1) OR OCTAL(2)? 2
INPUT S MEMORY IN INTEGER(1) OR OCTAL IN GII FORMAT(2)? 2
STARTING ADDRESS=? 0
MAXIMUM NUMBER OF CLOCKS TO SIMULATE=? 10
NUMBER OF CLOCKS BETWEEN OUTPUT POINTS=? 1

ENTER OUTPUT LINES DESIRED 1-ADDRESSES AND CLOCK
2- A1,A2,A3,B 3- MIR,SAR,LIT,CTR,AMPCR 4- BR1,BR2,MAR,BMAR,GC1,GC2
5- CONDITIONS
ENTER NUMBER OF OUTPUT LINES DESIRED? 5

BEGIN OUTPUT AT MPM ADDRESS=? 0
END OUTPUT AT MPM ADDRESS=? 4
ENTER 1 FOR S MEMORY DUMP WHEN PROGRAM TERMINATES? 1

ENTER S MEMORY VALUES IN CONSECUTIVE BLOCKS
ENTER 9999 FOR STARTING ADDRESS WHEN FINISHED

STARTING S MEMORY ADDRESS=? 1234
FINAL S MEMORY ADDRESS FOR THIS BLOCK=? 1234
SMEM(1234)=? .00000000000

STARTING S MEMORY ADDRESS=? 9999

P(1) ADDR. = 0 P(3) ADDR. = -1 CLOCK = 1
A1=00000000000 A2=00000000000 A3=00000000000 B=00000000000
MIR=00000000000 SAR=0 LIT=0 CTR=0 AMPCR=0
BR1=0 BR2=0 MAR=0 BMAR=0 GC1=0 GC2=0
LC1=0 LC2=0 MST=0 LST=0 ABT=0 ABV=0 CBV=0 SAI=0 RDC=0 INT=0

P(1) ADDR. = 1 P(2) ADDR. = 0 CLOCK = 2
A1=00000000000 A2=00000000000 A3=00000000000 B=00000000000
MIR=00000000000 SAR=0 LIT=0 CTR=0 AMPCR=1234
BR1=0 BR2=0 MAR=0 BMAR=0 GC1=0 GC2=0
LC1=0 LC2=0 MST=0 LST=0 ABT=0 ABV=0 CBV=0 SAI=0 RDC=0 INT=0

P(1) ADDR. = 2 P(3) ADDR. = 0 CLOCK = 3
A1=00000000000 A2=00000000000 A3=00000000000 B=00000000000
MIR=00000002322 SAR=0 LIT=0 CTR=0 AMPCR=1234
BR1=4 BR2=0 MAR=210 BMAR=1234 GC1=0 GC2=0
LC1=0 LC2=0 MST=0 LST=0 ABT=0 ABV=0 CBV=0 SAI=0 RDC=0 INT=0

P(1) ADDR. = 3 P(3) ADDR. = 2 CLOCK = 4
A1=00000000000 A2=00000000000 A3=00000000000 B=00000000000
MIR=00000002322 SAR=0 LIT=0 CTR=0 AMPCR=1234
BR1=4 BR2=0 MAR=210 BMAR=1234 GC1=0 GC2=0
LC1=0 LC2=0 MST=0 LST=0 ABT=0 ABV=0 CBV=0 SAI=0 RDC=0 INT=0

P(1) ADDR. = 3 P(3) ADDR. = 3 CLOCK = 5
A1=00000000000 A2=00000000000 A3=00000000000 B=00000000000
MIR=00000002322 SAR=0 LIT=0 CTR=0 AMPCR=1234
BR1=4 BR2=0 MAR=210 BMAR=1234 GC1=0 GC2=0
LC1=0 LC2=0 MST=0 LST=0 ABT=0 ABV=0 CBV=0 SAI=1 RDC=0 INT=0

END OF SIMULATION - REGISTERS CONTAIN

P(1) ADDR. = 3 P(3) ADDR. = 3 CLOCK = 6
 A1=00000000000 A2=00000000000 A3=00000000000 B=00000000000
 MIR=00000002322 SAR=0 LIT=0 CTR=0 AMPCR=1234
 BR1=4 BR2=0 MAR=210 BMAR=1234 GC1=0 GC2=0
 LC1=0 LC2=0 MST=0 LST=0 ABT=0 ADV=0 COV=0 SAT=1 RDC=0 INT=0

MEMORY DUMP REQUESTED

ENTER VALUES AS DONE IN MEMORY INPUT (9999 FOR STARTING ADDRESS WHEN FINISHED)

STARTING S MEMORY ADDRESS=? 1234
 FINAL S MEMORY ADDRESS FOR THIS BLOCK=? 1234

S MEMORY(1234) TO S MEMORY(1234) =

00000002322
 STARTING S MEMORY ADDRESS=? 9999

图 7-4 说明以八进制格式打印寄存器内容的一组输出行的实例

7-3-3 逻辑部件寄存器行

逻辑部件寄存器行 在程序控制行所规定的时钟间隔结束时，将给出 A1, A2, A3 和 B 寄存器的内容。如同前面在运行输入部分所规定的那样，本行可用整数格式或者八进制格式打印。若选择整数格式，则寄存器的内容用十进制数打印，前面的 0 被删掉；若选择八进制格式，则寄存器的内容用 11 位八进制数字打印，前面的 0 不删掉。要特别注意，寄存器是 32 位宽。寄存器的最高两位加上其前面的一个 0 位，对应于一个八进制数字。其余的 30 位，按三位一组划分，以形成其余的八进制数字。例如，若寄存器含有下列各位：

10/100/111/001/101/011/110/000/010/111/100

则打印出它的八进制数为：

24715360274

由于是在寄存器的最左位前外加一位二进制“0”，使其位数恰好是 3 的倍数，所以最左边的八进制数字限于 0、1、2 或 3。

图 7-4 给出一组输出行的实例，其中寄存器是以八进制格式打印的。

7-3-4 第一控制寄存器行

第一控制寄存器行给出 MIR, SAR, LIT, CTR 以及 AMPCR 的内容。MIR 的打印格式是与逻辑部件的相同的，即或者用八进制 (011) 格式或者用整数格式。SAR, LIT, CTR 以及 AMPCR 寄存器的内容用十进制格式打印。LIT, CTR 和 AMPCR 寄存器的内容要变换成十进制后再打印。SAR 寄存器的内容则要加以适当的调整，使得能够去掉为补偿寄存器宽度而插入的那一位。

SAR 中的值或者是原码或者是补码。由于 SAR 的宽度为 5 位，因此，若 SAR 的打印输出为：

$$\text{SAR} = 8$$

那么，它可以表示原码 8 (如在语句 $8 \rightarrow \text{SAR}$ 中那样)，也可以表示 32 减 24 (如在语句 $\text{COMP } 24 \rightarrow \text{SAR}$ 中那样)。

LIT 寄存器是 8 位宽，可以保存 0 到 255 范围内的数值。LIT 寄存器的内容或者表示原码，或者表示反码。因此，若 LIT 寄存器的打印输出值为：

$$\text{LIT} = 13$$

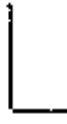
那么，这个值可以表示原码 (如语句 $13 \rightarrow \text{LIT}$ 那样)，也可以表示值 $256 - 242 - 1$ (如语句 $\text{COMP } 242 \rightarrow \text{LIT}$ 那样)。当 SLIT 在文字赋值语句中被用作目标时，对 LIT 寄存器的内容的解释可以有所不同。换言之，用户应当知道 LIT 在微程序中如何使用。若使用语句 $5 \rightarrow \text{SLIT}$ ，则翻译程序将生成一个 II 型语句，以把下面的二进制数值送入 LIT：

$$\begin{array}{c} 00001001 \\ \downarrow \\ \rightarrow \text{附加位} \end{array}$$

并且，寄存器的打印输出将是：

$$\text{LIT} = 9$$

另一方面,若使用语句 $\text{COMP } 5 \longrightarrow \text{SLIT}$, 则翻译程序将生成一个 II 型语句,以把下面的二进制数值送入 LIT:

11110011

 附加位

并且,寄存器的打印输出将是

$$\text{LIT} = 243$$

若把 SLIT 用作 II 型语句中的目标寄存器,那么要求接着执行形如

$$\text{LIT} \longrightarrow \text{SAR}, \dots$$

的 I 型语句.

CTR 中的值应被解释为值的反码. 因此,要执行下面的语句:

$$\begin{aligned} &\text{LCTR} \\ &2 \longrightarrow \text{LIT} \end{aligned}$$

而 CTR 的值将是

11111101

并且,寄存器的打印输出将是

$$\text{CTR} = 253$$

因此,若 CTR 的内容是 247,则相当于 CTR 的值是 8. 要记住:不论什么时候把一个数值送入 CTR,都必须用一个 I 型语句来完成,同时将它变成反码. 因此,若已把数值 n 送入 CTR,那么,重复执行 INC 命令,在 $n + 1$ 次增量操作时,将引起 CTR 溢出.

AMPCR 的值表示原码或者反码. 因此,如果执行下面的语句

$$1234 \longrightarrow \text{AMPCR}$$

则寄存器的打印输出为:

$$\text{AMPCR} = 1234$$

它是第一控制寄存器行的一部分.

7-3-5 第二控制寄存器行

第二控制寄存器行给出 BR1, BR2, MAR, BMAR, GC1 以

及 GC2 寄存器的内容。这行中的值总是用十进制格式打印的。

BR1 和 BR2 寄存器的内容是 16 位外部地址中的高 8 位(即最左边的 8 位)。装入这两个寄存器可以独立于存储器地址寄存器(MAR), 或者也可以用目标操作符 MAR1 或 MAR2 同时装入到这些寄存器和 MAR。类似地, 存储器地址寄存器(MAR)是同一个 16 位外部地址的低 8 位(即右边的), 它可以作为 MAR 而被单独地装入, 或者也可以用操作符 MAR1 和 MAR2 同时装入到 BR1 或 BR2 和 MAR。例如语句

AMPCR → MAR1, MIR

1234 → AMPCR

MW1

将使 16 位的 S 存储器地址 1234 送入 BR1/MAR 寄存器, 如下面所示:

00000100	11010010
BR1	MAR

当把 8 位值转换成十进制时, BR1 和 MAR 寄存器相应的打印输出将是

BR1 = 4 MAR = 210 BMAR = 1234

其中, BMAR 是用于存储器读或写操作的最后的外部 BRi/MAR 地址。

GC1 和 GC2 指示器分别作为总条件 1 和总条件 2。指示器为 1 意味着条件已建立; 指示器为 0 意味着条件未建立。总条件是为多台 D 机器的操作而设计的, 并且不能利用测试这些条件而清除它们。因此, 总条件是用形如 SET GC1 或 SET GC2 这样的命令来建立的, 而用形如 RESET GC1 或 RESET GC2 的命令来清除的。本书不讨论总条件, 但在翻译程序/模拟程序中则要利用它的功能。

7-3-6 条件行

条件行给出 LC1, LC2, MST, LST, ABT, AOV, COV,

SAI, RDC 和 INT 指示器的状态. LC1, LC2, COV, SAI 和 RDC 指示器借助测试这些条件加以清除. MST, LST, ABT 和 AOV 指示器在每一次由 I 型指令所产生的加法器操作期间被建立, 此时, 加法器操作是无条件执行的或在条件为“真”时执行的. 根据加法器操作结果得到的条件可以用下一条 I 型指令来测试.

在所有情况下, 指示器为 1 表示所指定的条件存在, 而指示器为 0 表示所指定的条件不存在. 因此, 例如输出打印下面所示的输出打印表达式:

AOV = 1

表示在上次加法器操作期间产生加法器溢出, 而打印输出打印

LST = 0

表示在完成上一次加法器操作时, 最低有效位的值为“0”. 其他条件也可作出相应的解释.

INT 条件是一种中断机构, 它用于具有多重处理部件的系统. SET INT 命令将中断所有的处理部件, 并使 INT 指示器置位. 这个条件可当作 IF INT THEN... 中的 INT 条件来测试. 如同用总条件一样, 本书不涉及中断功能, 但是翻译程序/模拟程序支持这种功能.

词 汇

读者应当熟悉本章中所使用的下列术语:

文件名 (File name)

整数格式 (Integer format)

八进制格式 (Octal format)

起始地址 (Starting address)

模拟的最大时钟数 (Maximum number of clocks to simulate)

两个输出点之间的时钟数 (Clocks between output points)

输出行 (Output lines)

S 存储器的起始(或曰“启动”)地址 (Starting S-memory address)

S 存储器的终止地址 (Final S-memory address)

存储块 (Block)

相位 1 (Phase 1) (亦称第 1 相)

相位 2 (Phase 2) (亦称第 2 相)

相位 3 (Phase 3) (亦称第 3 相)

提 问

下面一些问题打算用来测验你对主题内容的理解。所有问题都可以直接从课文中或者通过对所提出的主题进行逻辑推理而得到答案。有些问题适合于在讨论班上研究。

1. 询问“文件名”作什么用?
2. 起始地址值可以假定在什么范围内?
3. MPM 存贮器的起始地址是什么数值? 毫微存贮器呢? S 存贮器呢?
4. 什么是“保持相位”?
5. 正在执行的 M 指令给出什么输出行?
6. 为什么 MPM 和毫微存贮器不需要输出? 如何确定这两个存贮器中每一个的内容?
7. BMAR 是什么? 请具体回答。
8. 哪些条件是利用测试它来清除的? 哪些是用其他方法清除的? 为什么?
9. 在语句 $\text{COMP } 5 \longrightarrow \text{SAR}$ 中, 是取反码还是取补码? 在语句 $\text{COMP } 5 \longrightarrow \text{LIT}$ 中, 是取反码还是取补码?
10. 数值 9999 用于什么特殊功能?

习 题

1. 在下面一段微程序中:

$\text{LIT} + 1 \quad \text{R} \longrightarrow \text{B}$

$5 \longrightarrow \text{LIT}, \quad 1 \longrightarrow \text{SAR}$

IF LST THEN...

试问 LST 的测试是“真”还是“假”? 为什么?

2. 已知下面的输出:

$\text{BR2} = 5 \quad \text{MAB} = 326$

试问它表示什么外部地址?

3. 已知下面的输出:

$\text{CTR} = 237$

试问这个值表示什么信息?

4. 已知下面一段微程序:

1 \rightarrow A1

IF LST THEN LIT \rightarrow SAR

9 \rightarrow SAR, 17 \rightarrow SLIT

在这段微程序执行后, 试给出

A1 = _____ LIT = _____ SAR = _____

5. 用人工或用模拟程序执行下面一段微程序, 并给出每个时钟间隔内的各输出。解释其结果。

a AMPCR \rightarrow A1

4032 \rightarrow AMPCR

A1 + LIT C \rightarrow A2

2 \rightarrow LIT, 2 \rightarrow SAR

IF LST THEN A2 + 1 \rightarrow A3 SKIP ELSE STEP

B₁₀₀ \rightarrow A3

STEP

b AMPCR \rightarrow A2

75 \rightarrow AMPCR

A2 - LIT L \rightarrow MIR

3 \rightarrow LIT, COMP 1 \rightarrow SAR

A2 + 1 \rightarrow A2, IF LST THEN SKIP ELSE STEP

B₁₀₀ \rightarrow MIR

STEP

6. 一段微程序把数值 6379 送入 S 存贮器的 238 号单元。试写出、执行该段微程序并说明其结果。用输出 S 存贮器方法指出实际送入 238 号单元的值。

7. 一段微程序读出 S 存贮器 123 号单元的内容, 并把它写入 S 存贮器的 321 号单元。试写出、执行该段微程序并说明其结果。利用 S 存贮器的输入选择, 开始时把值 10450000000 送入 S 存贮器的 123 号单元。

8. 一段微程序产生 10 到 20 之间(包括 10 和 20 在内)的所有偶数并把它们送入 S 存贮器从 1002 开始的连续单元中。试写出、执行该段微程序并说明其结果。

9. 一段微程序产生前 10 个费班纳赛 (Fibonacci) 数*¹⁾ 并把它们送入 S 存贮器

*¹⁾ 所谓“费班纳赛 (Fibonacci) 数”是指, 数列中每个数是其前面两个数之和, 例如 1, 3, 4, 7, 11, 18, ...。——译者注

从 75 号单元开始的连续单元中, 试写出、执行该段微程序并说明其结果。

10. 一段微程序把整数 10、9、8、7、6 和 5 分别送入 S 存贮器的 3150, 3151, 3152, 3153, 3154 和 3155 号单元之中, 试写出、执行该段微程序并说明其结果。这些值并不预先装入存贮器, 而是用微程序来产生。

第八章 仿真原理 I: 算术运算

8-1 概 述

本章将通过实现定点或浮点算术运算的微程序来介绍仿真原理。算术运算操作与取数、存数和输入输出操作不同,它是执行得最频繁的一类计算机操作,务必特别重视。为了得到较好的系统性能,用微程序实现的算术运算效率是关键的因素。因此,相应于它们的微子程序必须反复琢磨,直到把其中每一个多余的时钟都“消除”为止。通过本章提供的算术运算微子程序实例与第九章的仿真程序实例加以比较,就能明显地看到这个事实。实际上,为了阐明概念,仿真程序已被简化了,但算术运算却仍有其更高的效率。然而,另一种考虑是,算术运算总是要处理到一个字中的各个位,这样就更有机会进行优化。就其本质而言,一个仿真程序包含有一定数量的微子程序之间的转移,而控制操作则更难于优化。

8-2 定点运算——补码运算

对补码运算,正数是按原码形式存贮,而负数则按补码形式存贮。尽管补码运算并没有明显地使用一个符号位,但是仍可以把(数的)最高有效位看作符号位: 0 表示正数而 1 表示负数。

8-2-1 加法和减法

补码形式的定点加法和减法是简单明了的。对于加法,就是简单地把两个操作数加起来;减法,就是把减数的补码加到被减数上。这两运算可用下面的例子说明:

SETUP: LIT \rightarrow MAR2


```

2——→LIT
MR2, BEX, LMAR    % READ FIRST OPERAND
3——→LIT
WHEN RDC THEN B——→A1, MR2    % READ
SECOND OPERAND
ADD—1——→AMPCR % READ SECOND OPERAND
WHEN RDC THEN BEX, CALL
A3——→MIR, LMAR    % RESULT IN A3
4——→LIT
MW2, IF SAI    % WRITE SUM TO S-MEM
WHEN SAI THEN STEP
SUB — 1——→AMPCR
CALL    % TO SUB AND RETURN
A3——→MIR,LMAR    % RESULT IN A3
5——→LIT
MW2, IF SAI
FINI — 1——→AMPCR
WHEN SAI THEN JUMP

```

ADD: A1 + B——→A3, JUMP	加法和减法
SUB: A1 + NOT B + 1——→A3, JUMP	微子程序

FINI: END

这个微程序分别从 S 存贮器的 2 号单元和 3 号单元读出第一和第二操作数，并分别把和数以及差数写入 S 存贮器的 4 号及 5 号单元。加法和减法运算用单句微子程序来实现，微子程序中用到的第一操作数（即被加数或被减数）已放入寄存器 A1，第二操作数（即加数或减数）已放入寄存器 B。结果（即和数或差数）送入寄存器 A3。由于这个微子程序比较短，所以这里给出整个微程序。以后，只给出个别运算操作的微子程序。在图 8-1 到 8-3 中，列出了执行加法和减法的翻译程序/模拟程序的打印输出结果。图 8-1

给出翻译程序表,而图 8-2 以及 8-3 分别给出执行 15 和 8 以及 15 和 -8 运算的模拟程序,

定点减法实现下面的表达式:

$$A1 + \text{NOT } B + 1$$

当采用补码运算时,它等价于 $A1 - B$.

```

>>>> TRANSLANG D MACHINE MICROTRANSLATOR.

ENTER FILENAME FOR HEX? BA05BHEX
SUPPRESS BIT PATTERNS?(1=SUPPRESS)? 1
SUPPRESS HEX LISTING?(1=SUPPRESS)? 1
IF INPUT IN FILE ENTER 1? 01
ENTER SOURCE FILENAME? BINADSB

0000  SETUP, LIT = MAR2 S
0001      2 = LIT S
0002  MR2, BEX, LMAR S READ FIRST OPERAND
0003      3 = LIT S
0004  WHEN RDC THEN B = A1, MR2 S READ SECOND OPERAND
0005  ADD - 1 = AMPCR S
0006  WHEN RDC THEN BEX, CALL S TO ADD AND RETURN
0007  A3 = MIR, LMAR S RESULT IN A3
0008      4 = LIT S
0009  MR2, IF SAI S WRITE SUM TO S MEMORY
0010  SUB - 1 = AMPCR S
0011  CALL S TO SUB AND RETURN.
0012  A3 = MIR, LMAR S RESULT IN A3
0013      5 = LIT S
0014  MR2, IF SAI S
0015  FINI - 1 = AMPCR S
0016  WHEN SAI THEN JUMP S
0017  ADD, A1 + B = A3, JUMP S
0018  SUB, A1 + NOT B + 1 = A3, JUMP S
0019  FINI, STEP S
0020      END S

THE TOTAL NUMBER OF ERRORS =      0

EXECUTE (Y/N)? N

```

图 8-1 描述补码形式的定点加法和减法的微程序, 第一和第二操作数存放在 S 存储器的 2 号和 3 号单元; 和数和差数送入 S 存储器的 4 号和 5 号单元(执行部分在图 8-2 和 8-3 中)。

8-2-2 乘法

二进制乘法是通过重复执行加法和移位来实现的, 并产生双倍长度的乘积。下面的例子中, 计算出的是一个双倍长(64 位)的

2) DMACK) (COMPILED) 05 JAN 76 13:43

ENTER SAME FILENAME FOR HEX? BAD5BHEX
OUTPUT REGISTERS AND S MEMORY IN INTEGER(1) OR OCTAL(2)? 1
INPUT S MEMORY IN INTEGER(1) OR OCTAL IN BII FORMAT(2)? 1
STARTING ADDRESS=? 0
MAXIMUM NUMBER OF CLOCKS TO SIMULATE=? 25
NUMBER OF CLOCKS BETWEEN OUTPUT POINTS=? 1

ENTER OUTPUT LINES DESIRED 1-ADDRESSES AND CLOCK
2- A1,A2,A3,B 3- MIR,SAR,LIT,CTR,AMPCR 4- BR1,BR2,MAR,BMAR,GC1,GC2
5- CONDITIONS

ENTER NUMBER OF OUTPUT LINES DESIRED? 5

BEGIN OUTPUT AT MPM ADDRESS=? 20
END OUTPUT AT MPM ADDRESS=? 20
ENTER 1 FOR S MEMORY DUMP WHEN PROGRAM TERMINATES? 1

ENTER S MEMORY VALUES IN CONSECUTIVE BLOCKS
ENTER 9999 FOR STARTING ADDRESS WHEN FINISHED

STARTING S MEMORY ADDRESS=? 2
FINAL S MEMORY ADDRESS FOR THIS BLOCK=? 3
S MEM(2)=? 15
S MEM(3)=? 8

STARTING S MEMORY ADDRESS=? 9999

END OF SIMULATION - REGISTERS CONTAIN

P(1) ADDR. = 19 P(3) ADDR. = 19 CLOCK = 21
A1 = 15 A2 = 0 A3 = 7 B = 8
MIR = 7 SAR = 0 LIT = 5 CTR = 0 AMPCR = 18
BR1 = 0 BR2 = 0 MAR = 5 BMAR = 5 GC1=0 GC2=0
LC1=0 LC2=0 MST=0 LST=0 ABT=0 ABV=0 CBV=0 SAI=1 RDC=1 IN1=0

MEMORY DUMP REQUESTED

ENTER VALUES AS DONE IN MEMORY INPUT (9999 FOR STARTING ADDRESS WHEN FINISHED)

STARTING S MEMORY ADDRESS=? 2
FINAL S MEMORY ADDRESS FOR THIS BLOCK=? 5

S MEMORY(2) TO S MEMORY(5) =

15 8 23 7
STARTING S MEMORY ADDRESS=? 9999

图 8-2 补码形式的定点加法和减法的执行。打印输出说明了 $15+8 \rightarrow 23$ 以及 $15-8 \rightarrow 7$

2) DMACK (COMPILED) 05 JAN 76 13:49

ENTER SAME FILENAME FOR HEX? BADSBHEX
OUTPUT REGISTERS AND S MEMORY IN INTEGER(1) OR OCTAL(2)? 2
INPUT S MEMORY IN INTEGER(1) OR OCTAL IN 011 FORMAT(2)? 2
STARTING ADDRESS=? 0
MAXIMUM NUMBER OF CLOCKS TO SIMULATE=? 25
NUMBER OF CLOCKS BETWEEN OUTPUT POINTS=? 1

ENTER OUTPUT LINES DESIRED 1-ADDRESSES AND CLOCK
2- A1,A2,A3,4 3- MIR,SAR,LIT,CTR,AMPCR 4- BR1,BR2,MAR,BMAR,GC1,GC2
5- CONDITIONS
ENTER NUMBER OF OUTPUT LINES DESIRED? 5

BEGIN OUTPUT AT MPH ADDRESS=? 20
END OUTPUT AT MPH ADDRESS=? 20
ENTER 1 FOR S MEMORY DUMP WHEN PROGRAM TERMINATES? 1

ENTER S MEMORY VALUES IN CONSECUTIVE BLOCKS
ENTER 9999 FOR STARTING ADDRESS WHEN FINISHED

STARTING S MEMORY ADDRESS=? 2
FINAL S MEMORY ADDRESS FOR THIS BLOCK=? 3
SMEM(2)=? 00000000017
SMEM(3)=? 37777777770

STARTING S MEMORY ADDRESS=? 9999

END OF SIMULATION - REGISTERS CONTAIN

P(1) ADDR. = 19 P(3) ADDR. = 19 CLOCK = 21
A1=00000000017 A2=00000000000 A3=00000000027 B=37777777770
MIR=00000000027 SAR=0 LIT=5 CTR=0 AMPCR=16
BR1=0 BR2=0 MAR=5 BMAR=5 GC1=0 GC2=0
LC1=0 LC2=0 MST=0 LST=0 ABT=0 ADV=0 CBV=0 SAI=1 RDC=1 INT=0

MEMORY DUMP REQUESTED

ENTER VALUES AS DONE IN MEMORY INPUT (9999 FOR STARTING ADDRESS WHEN FINISHED)

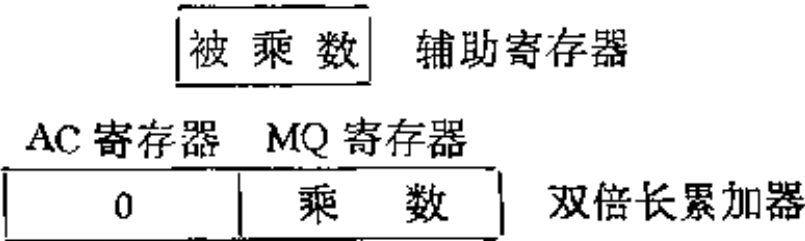
STARTING S MEMORY ADDRESS=? 2
FINAL S MEMORY ADDRESS FOR THIS BLOCK=? 5

S MEMORY(2) TO S MEMORY(5) =

00000000017 37777777770 00000000007 00000000027
STARTING S MEMORY ADDRESS=? 9999

图 8-3 补码形式的定点加法和减法的执行。打印输出说明了 $15+(-8) \rightarrow 7$ 以及 $15-(-8) \rightarrow 23$

乘积,但只保留最右边的 32 位. 执行这种运算要用到一个双倍长的累加器 (AC 和 MQ) 和一个辅助寄存器,如下所示:¹⁾



双倍长累加器是用两个寄存器来连起来实现的. 这两个寄存器按这样的方式使用: 从 AC 寄存器右移出来的二进位将进入 MQ 寄存器,如同这两个寄存器实际上是一个双倍长寄存器. 二进制乘法运算的过程如下:

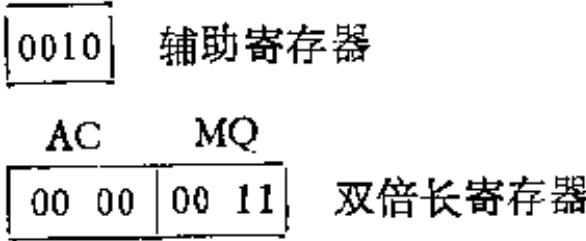
1.把被乘数、乘数以及一个全 0 分别送入辅助寄存器、MQ 寄存器及 AC 寄存器.

2.如果 MQ 寄存器的最右位为 1,则把被乘数加到 AC 寄存器中,反之,不执行加法操作. 但不管是否曾执行过加法操作,寄存器 (AC,MQ) 都要右移 1 位. 并对移位操作的次数进行计数.

3.当移位次数累计到等于乘数的位数时,则本次乘法操作就完成了. 否则,继续进行步骤 2.

4.结果放在寄存器 (AC,MQ) 中. 如果只产生单倍长的积,则它保存在 MQ 寄存器中.

作为本方法的一个例子,假定寄存器为 4 位,乘数为 0011 (二进制)、被乘数为 0010 (二进制). 一开始,被乘数 0010 被送入辅助寄存器,而乘数 0011 被送入 MQ 寄存器. AC 寄存器置为 0,于是,在乘法开始前各个寄存器的内容如下:



在乘法操作期间,辅助寄存器的内容保持不变.

1) MQ 是“乘-商”二字 (Multiplier-Quotient) 英文字头的缩写.

在第一个加/移位的周期内，由于 MQ 寄存器的最右位为 1，所以把被乘数加到 AC 寄存器中去。于是，AC 寄存器和 MQ 寄存器的内容变为：

AC	MQ
00 10	00 11

双倍长寄存器

再将 AC 和 MQ 内容联起来右移 1 位，得到：

AC	MQ
00 01	00 01

双倍长寄存器

这个“0”是——
从左边移入的

这样就完成了第一个加/移位周期。在第二个加/移位周期内，由于 MQ 寄存器的最右位又是 1，所以再次把被乘数加到 AC 寄存器中去。于是 AC 寄存器和 MQ 寄存器的内容变为：

AC	MQ
00 11	00 01

双倍长寄存器

再将 AC 和 MQ 的内容联起来右移 1 位得到：

AC	MQ
00 01	10 00

双倍长寄存器

这样就完成了第二个加/移位周期。在第三和第四个加/移位周期内，由于 MQ 寄存器的最右位都是“0”，所以不进行被乘数与 AC 寄存器的相加运算。在右移两位之后，AC 寄存器和 MQ 寄存器的内容变为：

AC	MQ
00 00	01 10

到此，双倍长的乘积就保留在 AC 寄存器和 MQ 寄存器中。其中，AC 寄存器为高位部分，MQ 寄存器为低位部分。当希望乘积为单倍长时，则可以把高位部分丢弃。

在大多数现代计算机中，都是用微程序来控制定点乘法的。在

下面的例子中，组合寄存器〈A1, A2〉分别用作 AC 寄存器和 MQ 寄存器，这样，重复的加法就加到 A1 寄存器中。乘数存放在 A2 寄存器，而被乘数则存放在 B 寄存器。

在进入乘法微子程序前，预先通过一个装配子程序把两个操作数存放在相应的寄存器中。下面所列的二进制乘法子程序，将把双倍长的乘积送到上述的〈A1, A2〉寄存器：

BMULT: AMPCR → MIR % SAVE RETURN ADDRESS

0 → A1, LCTR

32 → LIT, 1 → SAR

BMTEST - 1 → AMPCR

INC, CALL

主循环

<p>IF NOT LST THEN A1 R → A1, SKIP ELSE STEP</p> <p>A1 + B R → A1</p> <p>IF NOT LST THEN A2 R → A2, SKIP ELSE STEP</p> <p>A2 OR 1 C → A2</p> <p>BMTEST: IF NOT COV THEN A2 → , INC, JUMP</p> <p>ELSE STEP</p>

BMI % RESTORE RETURN ADDRESS

B → AMPCR

JUMP

图 8-4 给出把执行二进制乘法的微程序翻译成十六进制微编码的打印输出，而在模拟程序上运行的一个实例则如图 8-5 所示。乘法子程序由一个装配子程序来调用。在调用之前，该装配子程序从 S 存贮器的 100 号单元读出乘数，并把它放入寄存器 A2；又从 S 存贮器的 101 号单元读出被乘数，并把它放入寄存器 B。在乘法运算之后，装配子程序把乘积存入 S 存贮器的 103 号单元。这个二进制乘法微子程序展示了不同于别的乘法的两种技巧：

1. 提前判定法。
2. 将返回地址保存在 MIR 中。

TRANSLANG D. MACHINE MICRSTRANSULATOR

ENTER FILENAME FOR NEXT BMHEX
 SUPPRESS BIT PATTERNS?(1=SUPPRESS)? 1
 SUPPRESS HEX LISTING?(1=SUPPRESS)? 1
 IF INPUT IN FILE ENTER 17 31
 ENTER SOURCE FILENAME? BINMLT

```

0000  SETUP.  LMAR $
0001      100 = LIT $
0002      MR2 = BEX.  LMAR $
0003      101 = LIT $
0004      WHEN RDC THEN B = A2.  MR2 $
0005  BMULT - 1 = AMPCR $
0006      WHEN RDC THEN BEX.  CALL $
0007      A2 = MR.  LMAR $
0008      102 = LIT $
0009      MR2 = IF SAI $
0010      FINI - 1 = AMPCR $
0011      WHEN SAI THEN JUMP $
0012  BMULT.  AMPCR = MR $ SAVE RETURN ADDRESS
0013      D = A1.  LCTR $
0014      31 = LIT.  1 = SAR $
0015      BMTST - 1 = AMPCR $
0016      INC.  CALL $
0017      IF NOT LST THEN A1 R = A1.  SKIP ELSE STEP $
0018      A1 + B R = A1 $
0019      IF NOT LST THEN A2 R = A2.  SKIP ELSE STEP $
0020      A2 OR 1 C = A2 $
0021      BMTST.  IF NOT COV THEN A2 =, INC.  JUMP ELSE STEP $
0022      BMT $ RESTORE RETURN ADDRESS
0023      B = AMPCR $
0024      JUMP $
0025  FINI.  STEP $
0026      END $
    
```

THE TOTAL NUMBER OF ERRORS = 0

EXECUTE (Y/N)? N

图 8-4 把二进制乘法微程序翻译成十六进制微编码的打印输出

很清楚，当不使用提前判定法而使用寄存器 A3 来保存返回地址时，程序的效率还可以再提高一点，如：

BMULT: AMPCR → A3, LCTR %SAVE RETURN ADDRESS
 31 → LIT, 1 → SAR
 A2 →, SAVE

主循环

```

IF NOT LST THEN A1 R → A1 SKIP ELSE STEP
A1 + B R → A1
IF NOT LST THEN A2 R → A2 SKIP ELSE STEP
A2 OR 1 C → A2, INC
A2 →, IF COV THEN STEP ELSE JUMP
    
```


2) DNACK: (COMPILED) 05 JAN 76 14:08

ENTER SAME FILENAME FOR HEX? BMHEX
OUTPUT REGISTERS AND S MEMORY IN INTEGER(1) OR OCTAL(2)? 1
INPUT S MEMORY IN INTEGER(1) OR OCTAL IN 811 FORMAT(2)? 1
STARTING ADDRESS =? 0
MAXIMUM NUMBER OF CLOCKS TO SIMULATE=? 150
NUMBER OF CLOCKS BETWEEN OUTPUT POINTS=? 1

ENTER OUTPUT LINES DESIRED 1-ADDRESSES AND CLOCK
2- A1,A2,A3,B 3- MIR,SAR,LIT,CTR,AMPCR 4- BR1,BR2,NAR,BMAR,GC1,GC2
5- CONDITIONS
ENTER NUMBER OF OUTPUT LINES DESIRED? 5

BEGIN OUTPUT AT MPM ADDRESS=? 26
END OUTPUT AT MPM ADDRESS=? 26
ENTER 1 FOR S MEMORY DUMP WHEN PROGRAM TERMINATES? 1

ENTER S MEMORY VALUES IN CONSECUTIVE BLOCKS
ENTER 9999 FOR STARTING ADDRESS WHEN FINISHED

STARTING S MEMORY ADDRESS=? 100
FINAL S MEMORY ADDRESS FOR THIS BLOCK=? 101
SMEM(100)=? 37
SMEM(101)=? 15

STARTING S MEMORY ADDRESS=? 9999

END OF SIMULATION - REGISTERS CONTAIN

P(1) ADDR. = 25 P(3) ADDR. = 25 CLOCK = 127
A1 = 0 A2 = 555 A3 = 0 B = 6
MIR = 555 SAR = 1 LIT = 102 CTR = 0 AMPCR = 24
BR1 = 0 BR2 = 0 NAR = 102 BMAR = 102 GC1 = 0 GC2 = 0
LC1 = 0 LC2 = 0 MST = 0 LST = 0 ADT = 0 ASV = 0 CSV = 0 SAI = 1 RDC = 1 INT = 0

MEMORY DUMP REQUESTED

ENTER VALUES AS DONE IN MEMORY INPUT (9999 FOR STARTING ADDRESS WHEN FINISHED)

STARTING S MEMORY ADDRESS=? 100
FINAL S MEMORY ADDRESS FOR THIS BLOCK=? 102

S MEMORY(100) TO S MEMORY(102) =

37 15 555
STARTING S MEMORY ADDRESS=? 9999

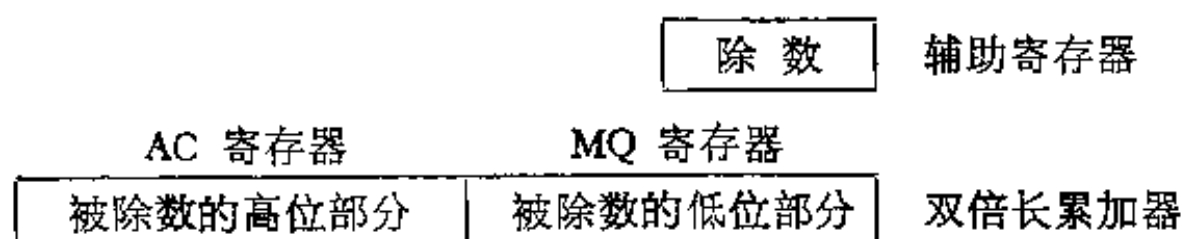
图 8-5 二进制乘法微子程序的执行

A3 → AMPCR % RESTORE RETURN ADDRESS
JUMP

也可以从二进制乘法微子程序返回到微程序中的一个固定单元,例如返回到取指令子程序,这是仿真程序常用的方法。二进制乘法子程序的主循环有五个 I 型语句,对于这类计算,其效率是高的。

8-2-3 除法

二进制除法是通过重复执行减法和移位来实现的,并且,它根据一个双倍长的被除数和一个单倍长的除数来产生一个单倍长的商数。在下述例子中,双倍长的被除数是由单倍长的被除数前加一个 0 而得到的。这个操作使用一个双倍长的累加器和一个辅助寄存器,如下所述:



这个双倍长的累加器是利用两个机器中的寄存器来实现的。这两个寄存器连接起来使用,使得从 MQ 寄存器左移出来的一位能送入 AC 寄存器,这就好象两个寄存器实际是一个双倍长寄存器一样。二进制除法过程如下:

1. 把双倍长被除数送到联合寄存器 (AC, MQ)。把除数送到辅助寄存器。如果 AC 寄存器(见上述)的内容大于或等于除数,则停止除法操作,因为这将产生除法溢出。

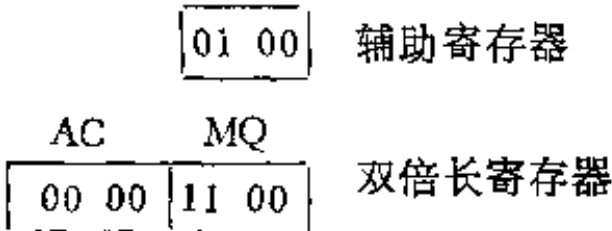
2. 把联合寄存器 (AC, MQ) 左移 1 位,并对移位操作的次数进行计数。

3. 如果 AC 寄存器的内容大于或等于辅助寄存器的内容,则从 AC 寄存器的内容减去辅助寄存器的内容,并令 MQ 寄存器的低位(即最右位)置 1。否则,继续进行步骤 4。

4. 当移位操作的次数等于除数的位数时,除数操作就完成了。

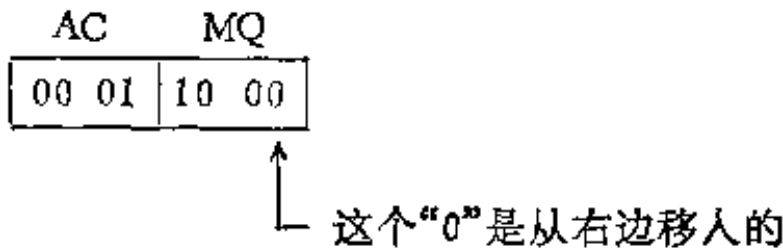
否则,继续进行步骤 2.

5. 商数保留在 MQ 寄存器中,余数保留在 AC 寄存器中,作为这种方法的一个例子,假定寄存器是 4 位,被除数是 00001100 (二进制),除数是 0100 (二进制). 一开始,把被除数和除数分别送入联合寄存器 (AC, MQ) 和辅助寄存器. 例如,

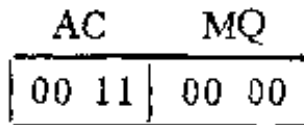


在除法操作期间,辅助寄存器的内容保持不变.

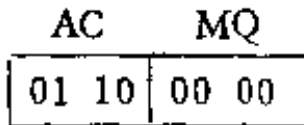
在第一个移位/减法周期内, (AC, MQ) 寄存器被左移 1 位,但不做减法,因为辅助寄存器的内容大于 AC 的内容 (AC, MQ) 寄存器描述如下:



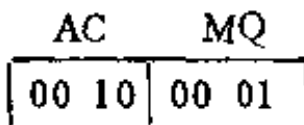
在第二个移位/减法周期内,再次进行左移,但仍不做减法(理由同上); (AC, MQ) 寄存器描述如下:



在第三个移位/减法周期内,要进行一次左移,有如下结果:



执行减法,并令 MQ 寄存器的低位置 1,



在第四亦即最后一个移位/减法周期内,要进行一次左移,有如下结果:

AC	MQ
01 00	00 10

最后一次减法完成了这个除法操作:

AC	MQ
00 00	00 11

商数 0011 (二进制) 在 MQ 寄存器中, 而余数 (本例余数是 0) 则在 AC 寄存器中.

如同定点乘法的情况那样, 在大多数现代计算机中, 定点除法也是用微程序控制的. 在下面的例子中, 把联合寄存器 (A1, A2) 分别用作 AC 寄存器和 MQ 寄存器. 这时, 对 A1 寄存器做减法. 除数存放在 B 寄存器. 当进入除法微子程序之前, 预先通过一个装配子程序把两个操作数存放在它们各自的寄存器中. 下面所列的二进制除法子程序, 把一个单倍长的商数送到寄存器 A2, 而余数则送到寄存器 A1:

```
BDIV: LCTR
      32 → LIT, COMPI → SAR
      BDTEST - 1 → AMPCR
      INC, CALL
```

主循环

```
IF NOT MST THEN A1 L → A1, SKIP ELSE STEP
A1 OR B100C → A1
A1 - B →
IF AOV THEN A1 - B → A1, STEP ELSE SKIP
A2 ORI → A2
BDTEST: IF NOT COV THEN A2 L → A2, INC
        JUMP ELSE STEP
```

```
RET - 1 → AMPCR
JUMP
```

图 8-6 给出把二进制除法微程序翻译成十六进制微编码的打

印输出，而在模拟程序上运行的一个实例则如图 8-7 所示。除法子程序由一个装配子程序来调用，该装配子程序从 S 存贮器的 100 号单元读出被除数，并把它送到寄存器 A2，然后从 S 存贮器的 101 号单元读出乘数，并把它送到寄存器 B。在调用之前，寄存器 A1 被置成 0。在除法运算之后，装配子程序把商数送入 S 存贮器的 102 号单元。

在控制循环和保存返回地址方面，二进制除法子程序使用了与二进制乘法一样的方法。此外二进制除法还需要做如下说明：

1. 为了测试寄存器 A1 的内容是大于还是等于寄存器 B 的内容，使用了形如下述的“大于或等于”测试：

```

>>>>> TRANSLANG D MACHINE MICROTRANSLATOR

ENTER FILENAME FOR HEX? BDHEX
SUPPRESS BIT PATTERNS?(1=SUPPRESS)? 1
SUPPRESS HEX LISTING?(1=SUPPRESS)? 1
IF INPUT IN FILE ENTER 1? 31
ENTER SOURCE FILENAME? BINDIV

0000  SETUP. 0 = A1, LMAR $
0001      100 = LIT $
0002      MR2, BEX, LMAR $
0003      101 = LIT $
0004      WHEN RDC THEN B = A2, MR2 $
0005      BDIV = 1 = AMPCR $
0006      WHEN RDC THEN BEX, JUMP $
0007  RET. A2 = NIR, LMAR $
0008      102 = LIT $
0009      MW2, IF SAI $
0010      FINI = 1 = AMPCR $
0011      WHEN SAI THEN JUMP $
0012  BDIV. LCTR $
0013      32 = LIT, COMP 1 = SAR $
0014      BDTEST = 1 = AMPCR $
0015      INC, CALL $
0016      IF NOT HST THEN A1 L = A1, SKIP ELSE STEP $
0017      A1 SR B100 C = A1 $
0018      A1 - B = $
0019      IF AGV THEN A1 - B = A1, STEP ELSE SKIP $
0020      A2 SR 1 = A2 $
0021  BDTEST. IF NOT CQV THEN A2 L = A2, INC, JUMP ELSE STEP $
0022      RET = 1 = AMPCR $
0023      JUMP $
0024  FINI. STEP $
0025      END $

THE TOTAL NUMBER OF ERRORS = 0

EXECUTE (Y/N)? N

```

图 8.6 把二进制除法微程序翻译成十六进制微编码的打印输出

2) DMACH1 (COMPILED) 05 JAN 76 14:26

Katzan Page 8-12b-

ENTER SAME FILENAME FOR HEX? BDHEX
OUTPUT REGISTERS AND S MEMORY IN INTEGER(1) OR OCTAL(2)? 1
INPUT S MEMORY IN INTEGER(1) OR OCTAL IN #11 FORMAT(2)? 1
STARTING ADDRESS =? 0
MAXIMUM NUMBER BE CLOCKS TO SIMULATE=? 175
NUMBER OF CLOCKS BETWEEN OUTPUT POINTS=? 1

ENTER OUTPUT LINES DESIRED. 1-ADDRESSES AND CLOCK
2- A1,A2,A3,B 3- MIR,SAR,LIT,CTR,AMPCR 4- BR1,BR2,MAR,BMAR,GC1,GC2
5- CONDITIONS
ENTER NUMBER OF OUTPUT LINES DESIRED? 5

BEGIN OUTPUT AT MEM ADDRESS=? 25
END OUTPUT AT MEM ADDRESS=? 25
ENTER 1 FOR S MEMORY DUMP WHEN PROGRAM TERMINATES? 1

ENTER S-MEMORY VALUES IN CONSECUTIVE BLOCKS
ENTER 9999 FOR STARTING ADDRESS WHEN FINISHED

STARTING S MEMORY ADDRESS=? 100
FINAL S MEMORY ADDRESS FOR THIS BLOCK=? 101
SMEM(100)=? 180
SMEM(101)=? 12

STARTING S MEMORY ADDRESS=? 9999

END OF SIMULATION - REGISTERS CONTAIN

P(1) ADDR. = 24 P(3) ADDR. = 24 CLOCK = 157
A1 = 0 A2 = 15 A3 = 0 B = 12
MIR = 15 SAR = 31 LIT = 102 CTR = 0 AMPCR = 23
BR1 = 0 BR2 = 0 MAR = 102 BMAR = 102 GC1 = 0 GC2 = 0
LC1 = 0 LC2 = 0 MST = 0 LST = 0 ABT = 0 ABV = 0 CBV = 0 SAI = 1 ROC = 1 INT = 0

MEMORY DUMP REQUESTED

ENTER VALUES AS DONE IN MEMORY INPUT (9999 FOR STARTING ADDRESS WHEN FINISHED)

STARTING S MEMORY ADDRESS=? 100
FINAL S MEMORY ADDRESS FOR THIS BLOCK=? 102

S MEMORY(100) TO S MEMORY(102) =

180 12 15
STARTING S MEMORY ADDRESS=? 9999

图 8-7 二进制除法微子程序的执行

A1 ← B →
IF AOV

2. “1”的循环左移,即

A1 OR B100 C → A1

此处 SAR 包含 COMP1.

在二进制除法子程序的主循环中是 6 个 I 型语言, 它的效率类似于二进制乘法.

8-3 定点算术运算——原码表示法*)

在第四章中, 已经介绍了定点数的原码表示法, 还给出了加法和减法的过程. 本节将给出这些过程的概要以及一个加法/减法微程序. 还将介绍用原码表示的定点数的乘法和除法, 并给出其过程. 这样, 就可以作为一种练习来编写相应的微子程序.

8-3-1 加法和减法

加法和减法本质上是相同的操作, 只是对减法而言, 在进行加运算之前, 必须改变减数的符号. 这个方法对应于教科书中的减法算法, 它可以说成是“减数改号然后两个数相加”. 因此, 加法和减法子程序通常被当作一个具有两个入口的子程序来实现. 这个算法可用图 8-8 中的流程图来概括.

用原码表示的二进制数的加法和减法数程序要求第一操作数(即被加数或者被减数)置于寄存器 A1 中, 第二操作数(即加数或者减数)置于寄存器 B 中. 运算结果(即和数或者差数)送回寄存器 A3. 子程序的主循环如下:

SUB: B_{FTT} → B

ADD: AMPCR → MIR % SAVE RETURN

A1 XOR B →, IF LCI % RESET OVERFLOWINDICATOR

*) 原文为“带符号的数值表示法”(Signed-magnitude representation), 实指“原码表示法”. ——译者注

SDIF - 1 \rightarrow AMPCR % BRANCH ADDR FOR
DIFF SIGNS

IF NOT MST THEN $A1 + B \rightarrow A3$, STEP ELSE JUMP

IF MST THEN $A3 \text{ AND } B_{011} \rightarrow A3$, SET LC1

$A3 \text{ OR } B_{100} \rightarrow A3$, BMI

$B \rightarrow \text{AMPCR}$ % RESTORE RETURN ADDRESS

JUMP % RETURN-RESULT IN A3

SDIF: $B \rightarrow A3$

IF NOT MST THEN $A1 \rightarrow B$, STEP ELSE SKIP

$A3 \rightarrow A1$ % B ALWAYS CONTAINS NEG

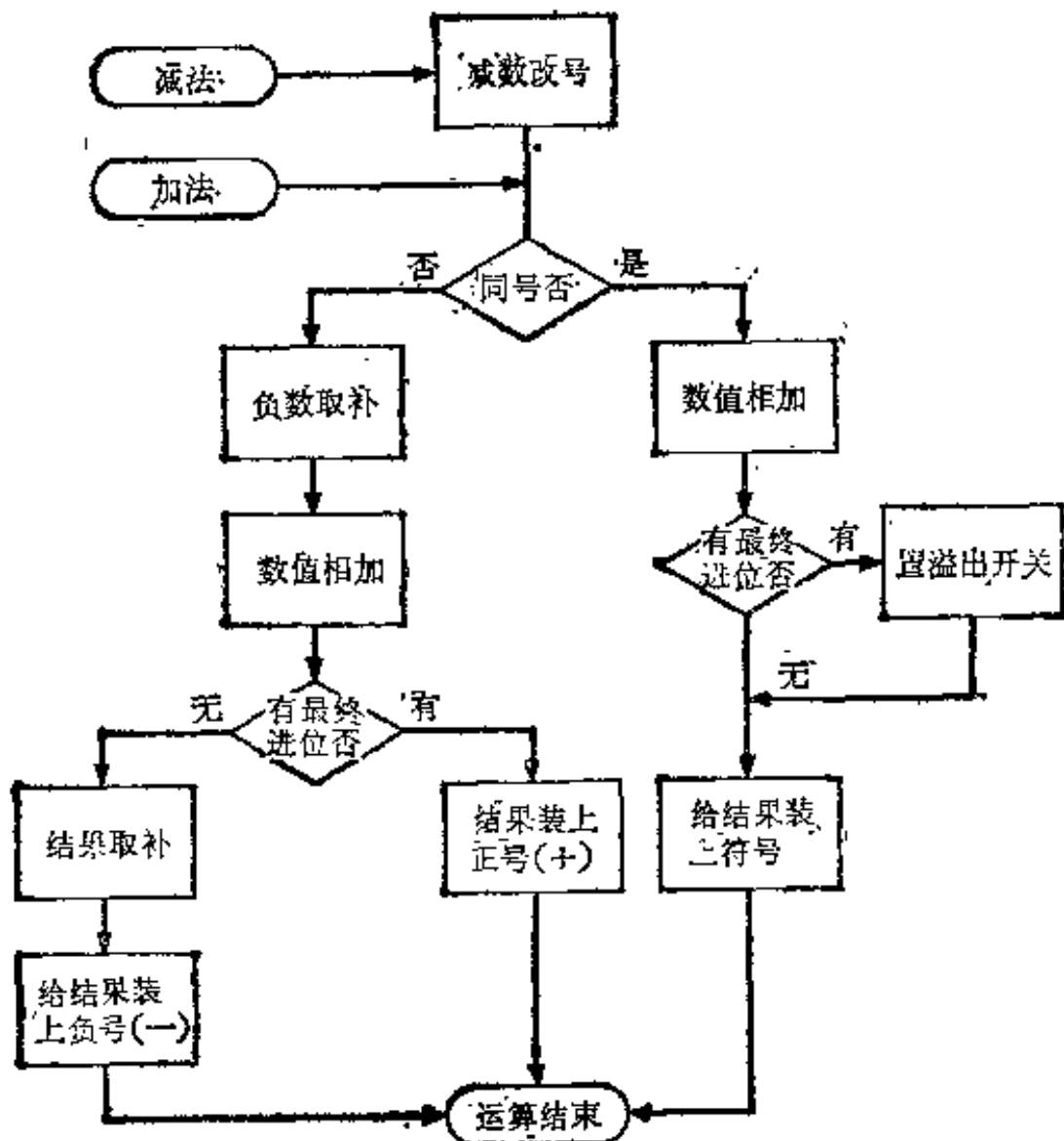


图 8-8 用原码表示的二进制数的
加法和减法算法的流程图


```

>>>> TRANSLANG D MACHINE MICROTRANSLATOR

ENTER FILENAME FOR HEX? SMHEX
SUPPRESS BIT PATTERNS?(1=SUPPRESS)? 1
SUPPRESS HEX LISTING?(1=SUPPRESS)? 1
IF INPUT IN FILE ENTER 17 31
ENTER SOURCE FILENAME? SMAG

0000 SETUP, LMAR $
0001 2 = LIT $
0002 MR2, BEX, LMAR $ READ FIRST OPERAND
0003 3 = LIT $
0004 WHEN RDC THEN B = A1, MR2 $ READ SECOND OPERAND
0005 ADD - 1 = AMPCR $
0006 WHEN RDC THEN BEX, CALL $ TO ADD
0007 A3 = MIR, LMAR $ RETURN SUM IN A3
0008 4 = LIT $
0009 MR2, LMAR, IF SAI $ WRITE SUM TO $ MEM
0010 3 = LIT $
0011 WHEN SAI THEN MR2, STEP $
0012 SUB - 1 = AMPCR $
0013 WHEN RDC THEN BEX, CALL $ TO SUB
0014 A3 = MIR, LMAR $ RETURN DIFF IN A3
0015 5 = LIT $
0016 MR2, IF SAI $ WRITE DIFF TO $ MEM
0017 FINI - 1 = AMPCR $
0018 WHEN SAI THEN JUMP $
0019 SUB, BFTT = B $
0020 ADD, AMPCR = MIR $ SAVE RETURN
0021 A1 XOR B =, IF LCI $ RESET OVRFLW (L01)
0022 SDIF - 1 = AMPCR $
0023 IF NOT MST THEN A1 + B = A3, STEP ELSE JUMP $
0024 IF MST THEN A3 AND B011 = A3, SET LCI $
0025 A3 OR B100 = A3, BMI $
0026 B = AMPCR $
0027 JUMP $ RETURN RESULT IN A3
0028 SDIF, B = A3 $
0029 IF NOT MST THEN A1 = B, STEP ELSE SKIP $
0030 A3 = A1 $ B CONTAINS NEG NUM
0031 BOFF = B $
0032 A1 + B + 1 = B $
0033 OVR - 1 = AMPCR $
0034 IF MST THEN B011 = A3, BMI, JUMP $
0035 BIFF = B $
0036 0 + B + 1 = A3, BMI $
0037 OVR, B = AMPCR $ RESTORE RETURN
0038 JUMP $ RETURN RESULT IN A3
0039 FINI, STEP $
0040 END $

THE TOTAL NUMBER OF ERRORS = 0

EXECUTE (Y/N)? N

```

图 8-9 把二进制原码加、减法微程序翻译成十六进制微编码的打印输出

$B_{OVR} \longrightarrow B$

$A1 + B + 1 \longrightarrow B$

$OVR - 1 \longrightarrow AMPCR$

2) DMACK1 (COMPILED) 05 JAN 76 14:42

ENTER SAME FILENAME FOR NEXT SMHEX

OUTPUT REGISTERS AND S MEMORY IN INTEGER(1) OR OCTAL(2)? 1

INPUT S MEMORY IN INTEGER(1) OR OCTAL IN 011 FORMAT(2)? 1

STARTING ADDRESS =? 0

MAXIMUM NUMBER OF CLOCKS TO SIMULATE=? 50

NUMBER OF CLOCKS BETWEEN OUTPUT POINTS=? 1

ENTER OUTPUT LINES DESIRED 1-ADDRESSES AND CLOCK

2- A1,A2,A3,B 3- MIR,SAR,LIT,CTR,AMPCR 4- BR1,BR2,MAR,BMAR,OC1,OC2

5- CONDITIONS

ENTER NUMBER OF OUTPUT LINES DESIRED? 5

BEGIN OUTPUT AT MPM ADDRESS=? 40

END OUTPUT AT MPM ADDRESS=? 40

ENTER 1 FOR S MEMORY DUMP WHEN PROGRAM TERMINATES? 1

ENTER S MEMORY VALUES IN CONSECUTIVE BLOCKS

ENTER 9999 FOR STARTING ADDRESS WHEN FINISHED

STARTING S MEMORY ADDRESS=? 2

FINAL S MEMORY ADDRESS FOR THIS BLOCK=? 3

SMEM(2)=? 15

SMEM(3)=? 8

STARTING S MEMORY ADDRESS=? 9999

END OF SIMULATION - REGISTERS CONTAIN

P(1) ADDR. = 39 P(3) ADDR. = 39 CLOCK = 42
A1 = 15 A2 = 0 A3 = 7 B = 13
MIR = 7 SAR = 0 LIT = 5 CTR = 0 AMPCR = 36
BR1 = 0 BR2 = 0 MAR = 5 BMAR = 5 OC1=0 OC2=0
LC1=0 LC2=0 MST=0 LST=0 ABT=0 ABV=0 CQV=0 SAT=1 RDC=1 INT=0

MEMORY DUMP REQUESTED

ENTER VALUES AS DONE IN MEMORY INPUT (9999 FOR STARTING ADDRESS WHEN FINISHED)

STARTING S MEMORY ADDRESS=? 2

FINAL S MEMORY ADDRESS FOR THIS BLOCK=? 5

S MEMORY(2) TO S MEMORY(5) =

15 8 23 7
STARTING S MEMORY ADDRESS=? 9999

图 8-10 原码形式的定点加、减法的
执行。打印输出描述了 $15+8=23$ 以
及 $15-8=7$ (输出用整数格式显示)

2) DMACH10 05 JAN 76 15:01

ENTER SAME FILENAME FOR HEX? SMHEX

OUTPUT REGISTERS AND S MEMORY IN INTEGER(1) OR OCTAL(2)? 2

INPUT S MEMORY IN INTEGER(1) OR OCTAL IN 811 FORMAT(2)? 2

STARTING ADDRESS=? 0

MAXIMUM NUMBER OF CLOCKS TO SIMULATE=? 50

NUMBER OF CLOCKS BETWEEN OUTPUT POINTS=? 1

ENTER OUTPUT LINES DESIRED 1-ADDRESSES AND CLOCK

2- A1,A2,A3,B 3- MIR,SAR,LIT,CTR,AMPCR 4- BR1,BR2,MAR,BMAR,GC1,GC2

5- CONDITIONS

ENTER NUMBER OF OUTPUT LINES DESIRED? 5

BEGIN OUTPUT AT MPN ADDRESS=? 40

END OUTPUT AT MPN ADDRESS=? 40

ENTER 1 FOR S MEMORY DUMP WHEN PROGRAM TERMINATES? 1

ENTER S MEMORY VALUES IN CONSECUTIVE BLOCKS

ENTER 9999 FOR STARTING ADDRESS WHEN FINISHED

STARTING S MEMORY ADDRESS=? 2

FINAL S MEMORY ADDRESS FOR THIS BLOCK=? 3

SMEM(2)=? 00000000017

SMEM(3)=? 20000000010

STARTING S MEMORY ADDRESS=? 9999

END OF SIMULATION - REGISTERS CONTAIN

P(1) ADDR. = 39 P(3) ADDR. = 39 CLOCK = 42
A1=00000000017 A2=00000000000 A3=00000000027 B=00000000015
MIR=00000000027 SAR=0 LIT=5 CTR=0 AMPCR=38
BR1=0 BR2=0 MAR=5 BMAR=5 GC1=0 GC2=0
LC1=0 LC2=0 MST=0 LST=0 ABT=0 ASV=0 CSV=0 SAI=1 RDC=1 INT=0

MEMORY DUMP REQUESTED

ENTER VALUES AS DONE IN MEMORY INPUT (9999 FOR STARTING ADDRESS WHEN FINISHED)

STARTING S MEMORY ADDRESS=? 2

FINAL S MEMORY ADDRESS FOR THIS BLOCK=? 5

S MEMORY(2) TO S MEMORY(5) =

00000000017 20000000010 00000000007 00000000027

STARTING S MEMORY ADDRESS=? 9999

图 8-11 原码形式的定点加、减法的执行。打印输出描述了 $15 + (-8) = 7$ 以及 $15 - (-8) = 23$ (输出用八进制格式显示)

IF MST THEN $B_{0TT} \rightarrow A3$, BMI, JUMP

$B_{1FF} \rightarrow B$

$0 + B + 1 \rightarrow A3$, BMI

OVR: $B \rightarrow \text{AMPCR}$ % RESTORE RETURN ADDRESS JUMP
% RETURN-RESULT IN A3

这个子程序事实上屏蔽掉了两个操作数的最高位，然后，分析各个操作数的符号进而执行加法运算。在加法运算之后，结果的最高位(即 MST 位)用来检验运算结果是否有溢出。在这个微子程序中，LC1 条件用来指示溢出条件的状况。

在图 8-9 中，给出翻译原码加法、减法微子程序以及相应的调用子程序的打印输出。运行实例见图 8-10 和 8-11。

8-3-2 乘法和除法

原码二进制乘法和除法需要使用补码运算的微子程序和用于符号控制的直接方法。对于乘法，可以用异或(XOR)条件来查询两个操作数的符号。并且，如果两个符号相同，则置一个开关(可以是 LC1, LC2, 或者 LC3)。然后，二进制乘法按正数值执行。如果两操作数同号，则结果为正；如果两操作数异号，则结果为负。应当注意：在去掉数的符号之后，操作数就少了 1 位。

这种处理符号的方法，也适用于除法，只是还要考虑余数的符号。如果 m 和 n 分别是被除数和除数，则除法可定义如下：

$$m = q \times n + r$$

此处 q 是商数， r 是余数，并且 r 的绝对值小于 n 的绝对值。商数和余数的符号是根据被除数和除数的符号决定的。如果 m 和 n 的代数符号相同，则 q 的符号为“正”；反之，为“负”。而 r 的符号与 m 的符号相同。

8-4 浮点运算

目前在大多数中、大型计算机中，都把浮点运算作为计算机结构的一部分来考虑。而且，浮点运算既可以用硬联方式控制，也可

以用微程序控制实现。在小型或微型计算机中，常常不直接给出进行浮点运算的计算机结构，而是作为 S 机器的子程序来实现。对于这两种情况，浮点算法是相同的，差别只在于实现的方法不同。

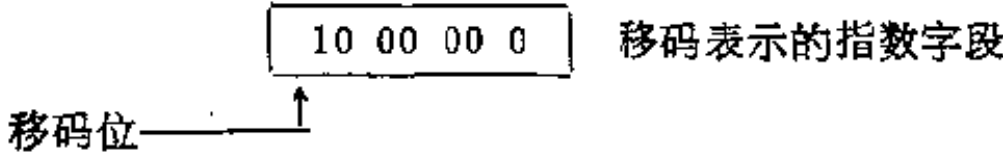
8-4-1 表示法

在执行浮点运算时，由于指数字段(阶部分)和小数字段(分数部分)必须单独处理，所以浮点数的表示法需要有特殊的格式。一般的格式如下：

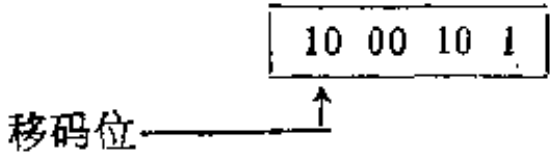
±	指 数	小 数
---	-----	-----

指数字段蕴含地决定了所能表示的数的范围，而小数字段则蕴含地决定了数的精度。与通常的用法相同，最左位被用作数的符号。

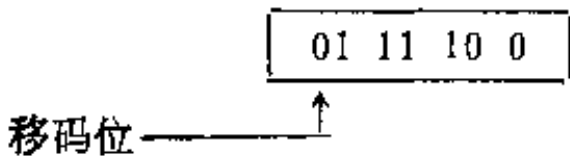
为了能够表示负指数，指数字段用移码表示，最常用的实现方法是把指数的最左位用作为移码 (bias) 位。这样，如果使用 7 位的指数，则“零”指数将呈如下形式：



其中，最左位对应于 64 的移码。二进制指数 +101 的移码将被表示成：



类似地，二进制指数 -100 的移码将被表示成：



二进制指数表示 2 的幂次，就象数学中的表示法那样。

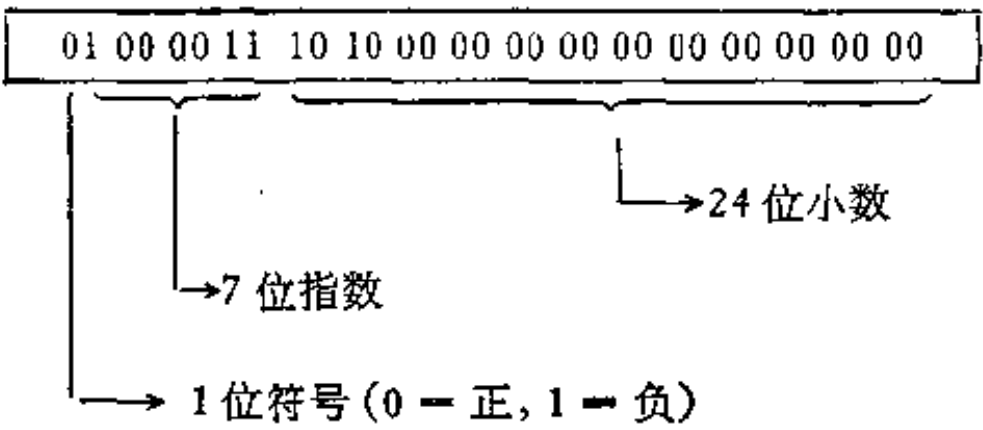
例如，考虑下面的 32 位浮点数的格式(它用在本章的后续部分)：

1	2	8	9	32
S	XXXXXXXX	YYYYYYYYYYYYYYYYYYYYYYYYYYYYYY		

在科学上的十进制和二进制表示法中,十进制数 5 被表示为:

十进制	二进制	
0.5×10^1	$= 101 \times 10^{11}$	
		二进制指数: $(11)_2 = (3)_{10}$
		二进制基数: $(10)_2 = (2)_{10}$
		二进制小数: $(101)_2 = (5)_{10}$

它对应于如下的浮点字:



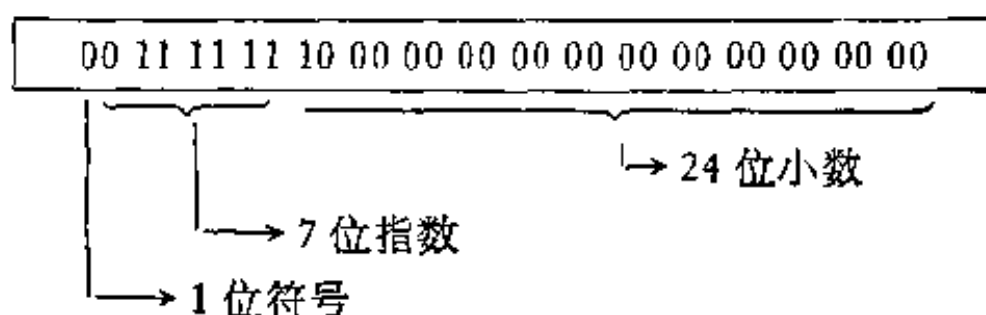
其中,二进制指数 $(11)_2 = (3)_{10}$ 表示: 二进制小数点在右边的第三位之后, 即

$$\cdot 10 \ 10 \ 00 \ 00 \ 00 \ 00 \ 00 \ 00 \ 00 \ 00 \ 00 \ 00 \ 00 \ 00 \ 00$$

这样,二进制小数点当然就正好处于小数的最左位的左边。类似地,在科学上的十进制和二进制表示法中,十进制数 0.25 被表示为:

十进制	二进制	
0.25×10^0	$= .1 \times 10^{-1}$	
		二进制指数: $(-1)_2 = (-1)_{10}$
		二进制基数: $(10)_2 = (2)_{10}$
		二进制小数: $(1)_2 = (1)_{10}$

显然, $(0.25)_{10} = (.01)_2$, 它对应于如下的浮点字:



指出这一点是很重要的：一个二进制指数并不总是表示 2 的幂次。由于指数蕴含地决定了一个浮点数的范围，所以，实际上常常用指数表示 8 的幂次（即每 3 位一组）或者 16 的幂次（即每 4 位一组）。例如，在 IBM360/370 计算机中，移码表示的一个 7 位二进制指数表示 16 的幂次（即十六进制数字的幂次），它给出指数的范围是 -64 到 +63，使得十进制数的范围从 10^{-73} 到 10^{73} 。

因为浮点数的小数部分决定了它的精度，所以，应该尽可能去掉小数部分前面的“0”位，以使有效位数最大。小数部分中最左位不为“0”的浮点数，称为规格化的浮点数，它的这种表示法给出了最大的精度。所以，尽管非规格化的浮点数

0	1 0 0 0 1 1 0	00 11 01 01 00 00 00 00 00 00 00 00
---	---------------	-------------------------------------

等价于下面的规格化数，

0	1 0 0 0 1 0 0	11 01 01 00 00 00 00 00 00 00 00 00
---	---------------	-------------------------------------

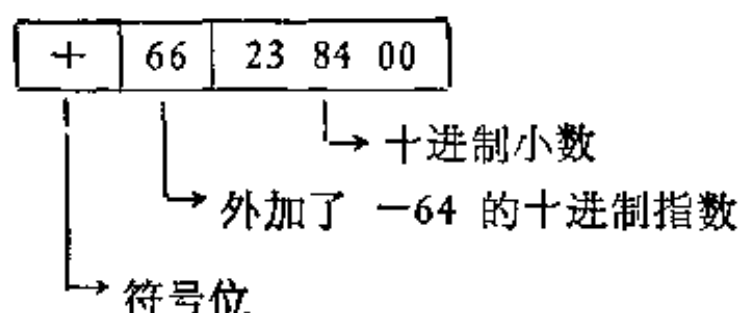
但是，由于上述理由，总是使用后一种形式。对浮点数进行的算术运算，总是产生规格化的结果。

浮点数的指数和小数部分几乎都用原码形式存贮，补码运算用得极少。然而，对于补码运算，除了在做加法运算时对阶可能复杂一些外，看来并没有任何更重要的原因。

8-4-2 基本浮点算法

本节给出实现浮点运算的基本算法。为便于叙述，采用十进制格式。在这种格式中，指数外加 -64，以表示 10 的幂次。这样，

数值 23.84 将按如下形式来存贮:



在这种情况下,指数为 $(66 - 64) = 2$, 它指出十进制小数点位于小数左起第二位之后. 在浮点加、减法运算中,需要知道十进制小数点的准确位置.

8-4-2-1 加法和减法

就浮点加法和减法而言,只当两个操作数的指数相等,即使得两个操作数的小数点位于相同的相对位置上时,才能进行加/减运算. 因此,当必要时,可用如下方法调整浮点数的指数和小数:“当小数部分每右移一位时,就把指数值加 1. 当小数部分每左移一位时,就把指数值减 1”. 利用这个方法,浮点加法和减法运算的执行可概括如下:

1. 比较两个指数;
2. 如果两个指数不等,则指数小的数的小数部分右移,直到两个指数相等;
3. 把两个小数相加或者相减;
4. 通过把小数进行左移,并相应地调整指数,直到小数的最左位为 1 的方式,使结果规格化.

运算结果的代数符号的处理方法,同原码操作数的定点运算方法相同.

作为本算法的一个例子,考虑用简化的十进制浮点形式表示的下列数值:

变量	浮点数值			
X	+	68	12 34 56	

Y	+	67	78 90 12
---	---	----	----------

计算表达式 $x - y$ 执行如下:

1. 使两个指数相等:

变量	浮点操作数			
X	+	68	12 34 56	
Y	+	68	07 89 01	

2. 执行减法运算:

表达式	浮点结果			
X - Y	+	68	04 45 55	

3. 使结果规格化:

表达式	规格化的浮点结果			
X - Y	+	67	44 55 50	

浮点加法也是按类似的方式执行的。

浮点加法或者减法运算的结果,也可能产生高位溢出。在这种情况下,要使小数部分右移——这同规格化正相反——并相应地增加指数。

8-4-2-2 乘法和除法

浮点数的乘法和除法使用下述数学关系式:

$$(f_1 \times b^{e_1}) \times (f_2 \times b^{e_2}) = (f_1 \times f_2) \times b^{e_1+e_2}$$

$$(f_1 \times b^{e_1}) \div (f_2 \times b^{e_2}) = (f_1 \div f_2) \times b^{e_1-e_2}$$

式中, b 是基数, f_1 和 f_2 是小数, 而 e_1 和 e_2 是指数。这样,这两种运算就分别归结为指数的加法或减法以及二进制小数的乘法或者除法。

作为这两种算法的一个例子,试考虑用简化的十进制浮点形式表示的下列数值:

变量	浮点数值		
A	+	66	25 00 00
B	+	65	30 00 00

计算表达式 $A \times B$ 执行如下:

1. 分离指数和小数

变量	指数	小 数
A	02	25 00 00
B	01	30 00 00

2. 指数相加、小数相乘

表达式	指数	小 数
$A \times B$	03	07 50 00

3. 必要时,使结果规格化

表达式	指数	小 数
$A \times B$	02	75 00 00

4. 使计算过的指数和小数合起来

表达式	规格化的浮点结果		
$A \times B$	+	66	75 00 00

浮点除法也按类似方式执行。

把二进制指数的最左位用作移码位 (bias) 的主要优点之一是, 通过屏蔽来分离指数值的过程比较简捷。另一种显而易见的技术是: 当指数相加时要扣去一个移码位; 当指数相减时要增补一个移码位, 以这种方式来保留移码位并修正所计算的指数。

8-4-3 加法微程序

虽然, 执行一次二进制加法运算只需一条微指令, 但是, 浮点

>>>> TRANSLANG D MACHINE MICROTRANSLATOR

ENTER FILENAME FOR HEX? FAHEX
 SUPPRESS BIT PATTERNS?(1=SUPPRESS)? 1
 SUPPRESS HEX LISTING?(1=SUPPRESS)? 1
 IF INPUT IN FILE ENTER IT 31
 ENTER SOURCE FILENAME? FLTADD

```

0000  SETUP.  LMAR  S
0001      100 = LIT S
0002      MR2, BEX,  LMAR S
0003      101 = LIT S
0004      WHEN RDC THEN B = A1, MR2 S
0005      FLADD - 1 = AMPCR S
0006      WHEN RDC THEN BEX, CALL S
0007      A3 = MIR,  LMAR S SUM IN A3
0008      102 = LIT S
0009      MW2, IF SAI S
0010      FINI + 1 = AMPCR S
0011      WHEN SAI THEN JUMP S
0012  FLADD.  AMPCR = MIR S
0013      B = A2 S OPERAND 2 TO A2
0014      A1 AND B011 R = A3 S EXP OF OPND 1 TO A3
0015      Z4 = SAR S
0016      A2 AND B011 R = B S EXP OF OPND 2 TO B
0017      A1 L = A1 S
0018      COMP B = SAR S
0019      A2 L = A2 S
0020      A1 R = A1 S FRACTION OF OPND 1
0021      B = SAR S
0022      A2 R = A2 S FRACTION OF OPND 2
0023  TEST.  A3 EQV B = S
0024      EQUAL - 1 = AMPCR S
0025      IF ABT THEN JUMP ELSE STEP 9
0026      A3 - B = S
0027      LESS - 1 = AMPCR S
0028      IF NOT ABV THEN JUMP ELSE STEP 3
0029      AR R = A2 S
0030      I = SAR S
0031      TEST - 1 = AMPCR S
0032      0 + B + I = B, JUMP S
0033  LESS.  A1 R = A1 S
0034      I = SAR S
0035      TEST - 1 = AMPCR S
0036      A3 + 1 = A3, JUMP S
0037  EQUAL.  A2 = B S
0038      A1 + B R = A1 S SUM IN A1
0039      I = SAR S
0040      A3 + 1 = A3 S
0041  NORM.  A1 R = A2 S
0042      Z3 = SAR S
0043      OK - 1 = AMPCR S
0044      A2 = S
0045      IF LST THEN JUMP ELSE STEP 3
0046      A1 L = A1 S
0047      COMP I = SAR S
0048      A3 - 1 = A3 S
0049  OK.  A3 L = B S
0050      COMP Z4 = SAR S
0051      A1 OR B = A3, BMI S RESTORE RETURN
0052      B = AMPCR S
0053      JUMP S
0054  FINI.  STEP S
0055      END S
    
```

THE TOTAL NUMBER OF ERRORS = 0
 EXECUTE (Y/N)? N

图 8-12 浮点加法微程序翻译成十六进制微编码的打印输出

操作所需要的一些辅助功能却使实际的加法操作很难理解。本节将介绍一个浮点加法微程序。在图 8-12 中,给出这个微程序的翻译程序表。通过一个装配子程序来完成下列操作:从 S 存储器的 100 号和 101 号单元中读出两个操作数,并分别送入寄存器 A1 和 B,然后调用浮点加微子程序。和数被送回寄存器 A3。为了介绍如何完成加法操作,给微程序加了注解,读者应当把这里给出的语句同图 8-12 联系起来,以得到整个过程的概貌。

微子程序的入口是符号单元 FLADD,而下列语句则准备了操作数:

FLADD: AMPCR	→MIR	A1	<table><tr><td>符号</td><td>指数</td><td>小数</td></tr></table>	符号	指数	小数	OP1
符号	指数	小数					
B	→A2	A2	<table><tr><td>符号</td><td>指数</td><td>小数</td></tr></table>	符号	指数	小数	OP2
符号	指数	小数					
A1 AND B ₂₄	R	→A3	A3	<table><tr><td>0—0</td><td>指数</td></tr></table>	0—0	指数	OP1
0—0	指数						
24	→SAR						
A2 AND B ₂₄	R	→B	B	<table><tr><td>0—0</td><td>指数</td></tr></table>	0—0	指数	OP2
0—0	指数						
A1 L	→A1	A1	<table><tr><td>小数</td><td>0—0</td></tr></table>	小数	0—0	P1	
小数	0—0						
COMP 8	→SAR	A2	<table><tr><td>小数</td><td>0—0</td></tr></table>	小数	0—0	P2	
小数	0—0						
A2 L	→A2						
A1 R	→A1	A1	<table><tr><td>0—0</td><td>小数</td></tr></table>	0—0	小数	OP1	
0—0	小数						
8	→SAR						
A2 R	→A2	A2	<table><tr><td>0—</td><td>小</td></tr></table>	0—	小	P2	
0—	小						

如同前述算法所指示的那样,这些语句只是分离了指数和小数。

在符号单元 TEST,对两个指数进行了比较,所用的比较方法已在前面的章节中给出。对指数和小数要作相应的修正,直到两个指数相等,这就意味着:两个小数点对齐了:

```

TEST: A3 EQV B →
      EQUAL - 1 → AMPCR
      IF ABT THEN JUMP ELSE STEP
      A3 - B →
      LESS - 1 → AMPCR
      IF NOT AOV THEN JUMP ELSE STEP
A2 R → A2
  1 → SAR
  TEST - 1 → AMPCR
  0 + B + 1 → B, JUMP
LESS: A1 R → A1
      1 → SAR
      TEST - 1 → AMPCR
      A3 + 1 → A3, JUMP

```

当通过修正过程使两个指数相等以后,就使两个小数相加,进而右移一位以处理溢出:

```

EQUAL: A2 → B
A1 + B R → A1  A1 

|  |     |
|--|-----|
|  | 小 数 |
|--|-----|

 和数
1 → SAR
A3 + 1 → A3  A3 

|  |     |
|--|-----|
|  | 指 数 |
|--|-----|

 和数

```

然后,通过检查小数的最左位的方式,使结果规格化:

```

NORM: A1 R → A2
      23 → SAR
      OK - 1 → AMPCR
      A2 →
      IF LST THEN JUMP ELSE STEP
      A1 L → A1
      COMP 1 → SAR
      A3 - 1 → A3

```

2) DNACH1 (COMPILED) 02 JAN 76 14:10

ENTER SAME FILENAME FOR HEX? FANEX
OUTPUT REGISTERS AND S MEMORY IN INTEGER(1) OR OCTAL(2)? 2
INPUT S MEMORY IN INTEGER(1) OR OCTAL IN 011 FORMAT(2)? 2
STARTING ADDRESS =? 0
MAXIMUM NUMBER OF CLOCKS TO SIMULATE=? 100
NUMBER OF CLOCKS BETWEEN OUTPUT POINTS=? 1

ENTER OUTPUT LINES DESIRED 1-ADDRESSES AND CLOCK
2- A1,A2,A3,B 3- MIR,SAR,LIT,CTR,AMPCR 4- BR1,BR2,MAR,DMAR,GC1,GC2
5- CONDITIONS
ENTER NUMBER OF OUTPUT LINES DESIRED? 5

BEGIN OUTPUT AT MPM ADDRESS=? 55
END OUTPUT AT MPM ADDRESS=? 55
ENTER 1 FOR S MEMORY DUMP WHEN PROGRAM TERMINATES? 1

ENTER S MEMORY VALUES IN CONSECUTIVE BLOCKS
ENTER 9999 FOR STARTING ADDRESS WHEN FINISHED

STARTING S MEMORY ADDRESS=? 100
FINAL S MEMORY ADDRESS FOR THIS BLOCK=? 101
SMEM(100)=? 10454000000
SMEM(101)=? 10260000000

STARTING S MEMORY ADDRESS=? 9999

END OF SIMULATION - REGISTERS CONTAIN

P(1) ADDR. = 54 P(2) ADDR. = 54 CLOCK = 65
A1=00070000000 A2=00000000000 A3=10470000000 B=00000000006
MIR=10470000000 SAR=8 LIT=102 CTR=0 AMPCR=53
BR1=0 BR2=0 MAR=102 DMAR=102 GC1=0 GC2=0
LC1=0 LC2=0 MST=0 LST=0 ABT=0 AQV=0 CBV=0 SA1=1 RDC=1 INT=0

MEMORY DUMP REQUESTED

ENTER VALUES AS DONE IN MEMORY INPUT (9999 FOR STARTING ADDRESS WHEN FINISHED)

STARTING S MEMORY ADDRESS=? 100
FINAL S MEMORY ADDRESS FOR THIS BLOCK=? 102

S MEMORY(100) TO S MEMORY(102) =

10454000000 10260000000 10470000000
STARTING S MEMORY ADDRESS=? 9999

图 8-13 浮点加法的执行(用八进制
格式显示输入和输出)

规格化之后,把指数和小数合在一起,进而使控制返回到装配子程序:

OK: A3 L \longrightarrow B B

0	指数	0——0
---	----	------

 和数

COMP 24 \longrightarrow SAR

A1 OR B \longrightarrow A3, BMI A3

0	指数	小数
---	----	----

 和数

B \longrightarrow AMPCR

JUMP

在这个微程序中,为了阐明表示方法并强调浮点操作中所包括的那些辅助功能,已经把符号省略了。当考虑两个操作数的符号时,只影响到小数的加法,并且仍然采用适合于原码表示法的方法。

在图 8-13 中,给出浮点加法的一个运行实例。

8-4-4 乘法微程序

浮点乘法微子程序把定点乘法微子程序用作小数相乘的子程序。在图 8-14 中,给出执行浮点乘法微程序的翻译程序表。通过一个装配子程序来完成下列操作: 从 S 存贮器的 100 号和 101 号单元中读出两个操作数,并分别送入寄存器 A1 和 B,然后调用浮点乘法微子程序。乘积被送回寄存器 A3。

微子程序的入口是符号单元 FLMLT,而下列语句则准备了操作数和指数,并调用定点乘法微子程序:

A1

符号	指数	小数
----	----	----

 OP1

FLMLT: B \longrightarrow A2 A2

符号	指数	小数
----	----	----

 OP2

A1 AND B₀₁₁ R \longrightarrow A3 A3

0——0	指数
------	----

 OP1

24 \longrightarrow SAR

A2 AND B₀₁₁ R \longrightarrow B B

0——0	指数
------	----

 OP2

***** TRANSLANG D MACHINE MICROTRANSLATOR

```

ENTER FILENAME FOR HEX? FMHEX
SUPPRESS BIT PATTERNS?(1=SUPPRESS)? 1
SUPPRESS HEX LISTING?(1=SUPPRESS)? 1
IF INPUT IN FILE ENTER 1? 31
ENTER SOURCE FILENAME? FLMLT

0000  SETUP.  LMAR $
0001      100 = LIT $
0002      MR2, BEX, LMAR $
0003      101 = LIT $
0004      WHEN RDC THEN B = A1, MR2 $
0005      FLMLT - 1 = AMPCR $
0006      WHEN RDC THEN BEX, JUMP $
0007  DONE.  A3 = MIR, LMAR $
0008      102 = LIT $
0009      MR2, IF SAI $
0010      FINI - 1 = AMPCR $
0011      WHEN SAI THEN JUMP $
0012  FLMLT. B = A2 $ OPND 2 TO A2 $
0013      A1 AND B011 R = A3 $ EXP OF OPND 1 TO A3
0014      24 = SAR $
0015      A2 AND B011 R = B $ EXP OF OPND 2 TO B
0016      A1 L = A1 $
0017      COMP $ = SAR $
0018      A2 L = A2 $
0019      A1 R = A1 $ FRACTION OF OPND 1
0020      $ = SAR $
0021      A2 R = A2 $ FRACTION OF OPND 2
0022      A3 + B = A3 $
0023      64 = LIT $
0024      A3 - LIT = A3 $ NEW EXPONENT
0025      FXMLT - 1 = AMPCR $
0026      A1 = B, JUMP $
0027  RET.  A2 = B $ PRODUCT IN <A1,A2>
0028      B R = MIR $
0029      24 = SAR $
0030      A1 L = B01 $ FRACTION IN B
0031  NORM.  B R = A1 $
0032      23 = SAR $
0033      BK - 1 = AMPCR $
0034      A1 = $
0035      IF LST THEN JUMP ELSE STEP $
0036      B L = B $
0037      COMP 1 = SAR $
0038      A3 - 1 = A3 $
0039  OK.  A3 L = A3 $
0040      COMP 24 = SAR $
0041      DONE - 1 = AMPCR $
0042      A3 OR B = A3, JUMP $
0043  FXMLT. B = A1, LCTR $
0044      32 = LIT, 1 = SAR $
0045      XTEST - 1 = AMPCR $
0046      INC, CALL $
0047      IF NOT LST THEN A1 R = A1, SKIP ELSE STEP $
0048      A1 + B R = A1 $
0049      IF NOT LST THEN A2 R = A2, SKIP ELSE STEP $
0050      A2 OR 1 C = A2 $
0051  XTEST. IF NOT CBV THEN AB =, INC, JUMP ELSE STEP $
0052      RET - 1 = AMPCR $
0053      JUMP $
0054  FINI. STEP $
0055      END $
THE TOTAL NUMBER OF ERRORS = 0

EXECUTE (Y/N)? N

```

图 8-14 把浮点乘法微程序翻译成十六进制微编码的打印输出

A1 L \longrightarrow A1	A1	<table border="1"><tr><td>小数</td><td>0——0</td></tr></table>	小数	0——0	OP1
小数	0——0				
COMP 8 \longrightarrow SAR					
A2 L \longrightarrow A2	A2	<table border="1"><tr><td>小数</td><td>0——0</td></tr></table>	小数	0——0	OP2
小数	0——0				
A1 R \longrightarrow A1	A1	<table border="1"><tr><td>0——0</td><td>小数</td></tr></table>	0——0	小数	OP1
0——0	小数				
8 \longrightarrow SAR					
A2 R \longrightarrow A2	A2	<table border="1"><tr><td>0——0</td><td>小数</td></tr></table>	0——0	小数	OP2
0——0	小数				
A3 + B \longrightarrow A3					
64 \longrightarrow LIT					
A3 - LIT \longrightarrow A3	A3	<table border="1"><tr><td>0——0</td><td>指数</td></tr></table>	0——0	指数	乘积
0——0	指数				
FXMLT - 1 \longrightarrow AMPCR					
A1 \longrightarrow B, JUMP					

这个定点乘法微子程序 FXMLT 要求把两个操作数存放在寄存器 A2 和 B 中,并把双倍长的乘积送回联合寄存器 (A1, A2)。在这种情况下,乘积是 48 位长:

```

FXMLT: 0  $\longrightarrow$  A1, LCTR
        32  $\longrightarrow$  LIT, 1  $\longrightarrow$  SAR
        XTEST - 1  $\longrightarrow$  AMPCR
        INC, CALL
        IF NOT LST THEN A1 R  $\longrightarrow$  A1, SKIP ELSE STEP
        A1 + B R  $\longrightarrow$  A1
        IF NOT LST THEN A2 R  $\longrightarrow$  A2, SKIP ELSE STEP
        A2 OR 1 C  $\longrightarrow$  A2
XTEST: IF NOT COV THEN A2  $\longrightarrow$ , INC, JUMP ELSE STEP
        RET - 1  $\longrightarrow$  AMPCR
        JUMP

```

在小数部分的乘法已经完成、得到了 48 位乘积之后,只保留高 24 位,而低 24 位被移弃。双倍长寄存器的移位使用前面给出

R) DMACH1 (COMPILED) 02 JAN 76 14:30

ENTER SAME FILENAME FOR NEXT FMHFX

OUTPUT REGISTERS AND S MEMORY IN INTEGER(1) OR OCTAL(2)? 2

INPUT S MEMORY IN INTEGER(1) OR OCTAL IN BII FORMAT(2)? 2

STARTING ADDRESS =? 0

MAXIMUM NUMBER OF CLOCKS TO SIMULATE=? 250

NUMBER OF CLOCKS BETWEEN OUTPUT POINTS=? 1

ENTER OUTPUT LINES DESIRED 1-ADDRESSES AND CLOCK

2- A1,A2,A3,B 3- MIR,SAR,LIT,CTR,AMPCR 4- BR1,BR2,MAR,BMAR,GC1,GC2

5- CONDITIONS

ENTER NUMBER OF OUTPUT LINES DESIRED? 5

BEGIN OUTPUT AT MPM ADDRESS=? 55

END OUTPUT AT MPM ADDRESS=? 55

ENTER 1 FOR S MEMORY DUMP WHEN PROGRAM TERMINATES? 1

ENTER S MEMORY VALUES IN CONSECUTIVE BLOCKS

ENTER 9999 FOR STARTING ADDRESS WHEN FINISHED

STARTING S MEMORY ADDRESS=? 100

FINAL S MEMORY ADDRESS FOR THIS BLOCK=? 101

SMEM(100)=? 10360000000

SMEM(101)=? 10250000000

STARTING S MEMORY ADDRESS=? 9999

END OF SIMULATION - REGISTERS CONTAIN

P(1) ADDR. = 54 P(3) ADDR. = 54 CLOCK = 150

A1=00000000000 A2=00000000000 A3=10474000000 B=00074000000

MIR=10474000000 SAR=8 LIT=102 CTR=0 AMPCR=51

BR1=0 BR2=0 MAR=102 BMAR=102 GC1=0 GC2=0

LC1=0 LC2=0 MST=0 LST=0 ABT=0 ABV=0 CBV=0 SAI=1 RDC=1 INT=0

MEMORY DUMP REQUESTED

ENTER VALUES AS DONE IN MEMORY INPUT (9999 FOR STARTING ADDRESS WHEN FINISHED)

STARTING S MEMORY ADDRESS=? 100

FINAL S MEMORY ADDRESS FOR THIS BLOCK=? 102

S MEMORY(100) TO S MEMORY(102) =

10360000000 10250000000 10474000000

STARTING S MEMORY ADDRESS=? 9999

图 8-15 浮点乘法的执行(用八进制格式显示输入和输出)

的方法(乘积的最右寄存器的内容在 B 寄存器中):

RET A2 \longrightarrow B

B R \longrightarrow MIR

24 \longrightarrow SAR

A1 L \longrightarrow BBI B

0 \longrightarrow 0	小数
-----------------------	----

 乘积

最后,使浮点乘积规格化,把指数和小数部分合起来并返回到其他的调用微子程序(乘积在寄存器 A3 中):

NORM: B R \longrightarrow A1

23 \longrightarrow SAR

OK - 1 \longrightarrow AMPCR

A1 \longrightarrow , IF LST THEN JUMP ELSE STEP

B L \longrightarrow B

COMP 1 \longrightarrow SAR

A3 - 1 \longrightarrow A3

OK: A3 L \longrightarrow A3

COMP 24 \longrightarrow SAR

DONE - 1 \longrightarrow AMPCR

A3 OR B \longrightarrow A3, JUMP

在图 8-15 中,给出浮点乘法微子程序的一个运行实例.

在上述情况中,代数符号仍然被忽略. 然而,把符号考虑进去也是比较简单的. 首先要比较两个符号位,进而当符号位不同时置 LC1 条件,如

A1 XOR B \longrightarrow , IF LC1

IF MST THEN SET LC1

这样,如果乘积为负,就在将要退出浮点乘法微子程序之前,增加下述语句:

IF LC1 THEN A3 OR B₁₀₀ \longrightarrow A3

以此补入一个负号.

词 汇

读者应当熟悉本章中所使用的下列术语:

补码运算 (Complement arithmetic)

符号位 (Sign bit)

被减数 (Minuend)

减数 (Subtrahend)

被加数 (Addend)

加数 (Augend)

被乘数 (Multiplicand)

乘数 (Multiplier)

双倍长累加器 (Double-length accumulator)

辅助寄存器 (Auxiliary register)

原码表示法 (Signed-magnitude representation)

余数 (Remainder)

移码表示的指数 (Biased exponent)

小数 (Fraction)

指数 (Exponent)

规格化数 (Normalized number)

规格化 (Normalization)

提 问

下面一些问题打算用来测验你对主题内容的理解。所有问题都可以直接从课文中或者通过所提出的主题进行逻辑推理而得到答案。有些问题适合于在讨论班上研究。

1. 为什么二进制乘法总是产生双倍长的乘积? 如果考虑有效位数, 那么何时才在实际上需要双倍长的乘积?
2. 在二进制乘法运算前, 为什么要把 0 送入 A1 寄存器?
3. 如果一开始, AC 寄存器的内容就大于辅助寄存器的内容, 为什么二进制除法会产生溢出结果?
4. 语句 $B_{FTT} \rightarrow B$ 是怎样修改减数的符号的? 考虑语句 $B_{LFF} \rightarrow B$, 它取反码还是取补码?
5. 在浮点格式中, 把隐式小数点置于小数部分的最高位的左边, 而不是置于

小数部分的最右位的右边这一方法的主要优点是什么？全都是优点吗？

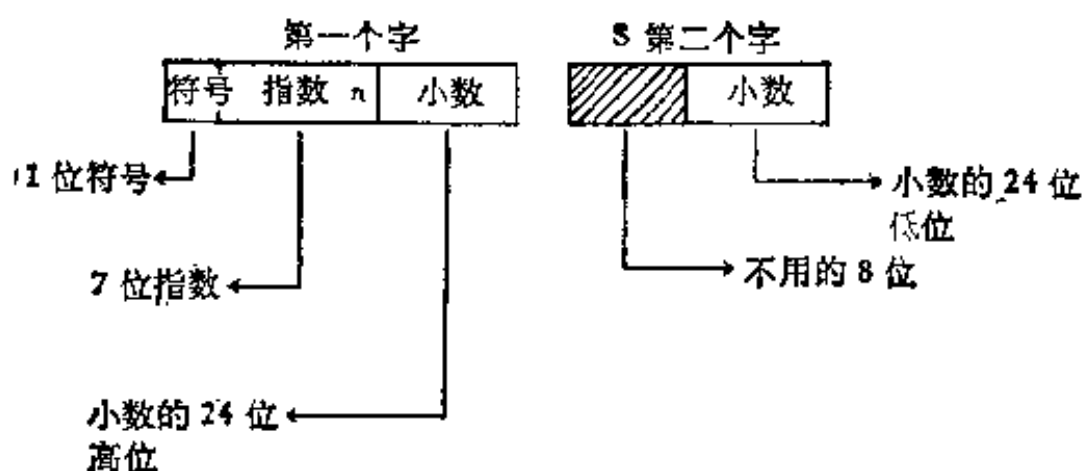
6. 试考虑浮点乘法的微程序。在最右边的 24 位被舍弃之前对小数进行规格化有什么优点？
7. 把定点乘法子程序用于浮点乘法，用什么方式能够提高速度？
8. 在浮点乘法微子程序中，语句 $A3 \leftarrow LIT \rightarrow A3$ 适用于什么基本功能？

习 题

1. 尽可能精练地概述二进制定点补码乘法的执行过程。对于除法也同样概述一遍。
2. 说明为什么 $A1 \leftarrow B$ 等于 $A1 \leftarrow NOT\ B + 1$ 。
3. 在前面的章节中，已经给出了把两个“联合的”寄存器右移 n 位或者左移 n 位的方法。试说明这种方法是如何运用的。
4. 试编写一段微程序，用以屏蔽掉寄存器 B 的最低位，并把结果送入寄存器 A1。
5. 修改浮点加法子程序，使之包括代数符号的处理。
6. 修改浮点乘法子程序，使之包括代数符号的处理。

问 题

1. 当进行定点乘法时，如果乘数的前几位是 0，则微程序要经过一些本无必要的机器周期。试修改二进制乘法微子程序，使得当乘数有多余的 0 位时，就能“跳过”相应的机器周期（提示：除寄存器 A2 之外，又把乘数置于另一个寄存器中，使之与 A2 一起移位，并测试相应的位是否为 0。）
2. 试编写微程序来实现下列运算（采用原码表示法）：
 - (a) 浮点乘法
 - (b) 浮点除法
3. 试编写微程序来实现下列浮点运算：
 - (a) 浮点减法
 - (b) 浮点除法
4. 有些双倍精度的浮点运算使用下列双字长的操作数：
试编写并检验下列运算的双倍精度微子程序：
 - (a) 浮点加法
 - (b) 浮点减法



(c) 浮点乘法

(d) 浮点除法

5. 试编写计算浮点数绝对值的微子程序。这个微子程序应当从寄存器 A2 取得浮点数, 并把 A2 内容的绝对值送回 A2 中。
6. 试编写计算浮点数 x 的整函数的微子程序, 亦即在 BASIC 中是 $\text{INT}(x)$, 或者在 FORTRAN 中是 $\text{AINT}(x)$ 。这个微子程序应当从寄存器 A2 取得浮点数, 并把该浮点数的整数部分送回同一个寄存器 A2 (结果也应当是浮点形式。)
7. 试编写计算浮点数 x 的平方根 (R) 的微子程序, 亦即在 BASIC 中是 $\text{SQR}(x)$, 或者在 FORTRAN 中是 $\text{SQRT}(x)$, 可使用下面的算法:

$E = .01$

$R = 1.0$

2 $R = .5 * (x/R + R)$

$\text{IF}(\text{ABS}(R^2 - x) \geq E) \text{GO TO } 2$

这个微子程序应当从寄存器 A2 取得浮点数, 并把 A2 寄存器内容的平方根送回同一个寄存器 A2。

第九章 仿真原理 II: 解释程序的设计

9-1 概 述

在微程序设计领域中,解释程序(interpreter)是这样一种微程序:它在受控制的情况下,取出 S 指令、译码 S 指令以及执行 S 指令。从另一种不同的观点来看,解释程序是这样一组程序:它占用微程序计算机的控制存贮器,并控制该计算机的操作。解释程序和仿真程序(emulator)是同义语。

在设计解释程序时,必须考虑下列几个因素:

1. 计算机的操作原理;
2. 机器寄存器的实现;
3. S 指令的格式;
4. 可以利用的、S 机器的一组操作。

这里所介绍的方法包括:对每个因素都进行简要的讨论以及在实现一个实际的解释程序中所提出的一些概念性的例子。

9-2 设计考虑

一个解释程序的设计涉及到上述四个因素的相互关系,这实质上意味着计算机设计师必须明确地,或者蕴含地针对每一个因素都作出相应的设计决定,并且,将这些决定汇总起来给出计算机系统的操作特征。

9-2-1 计算机的操作原理

计算机的操作原理要影响指令格式、所需要的寄存器,并且在某种程度上会影响某些可使用的操作。在这里考虑两种基本的设计:堆栈机器以及多寄存器机器。堆栈机器有时也称为“波兰

(Polish) 表示的机器”，它把按堆栈方式工作的机器寄存器用作累加器，并且，它接收形如波兰表示法的 S 机器程序。多寄存器机器使用一组用于算术/逻辑操作和寻址的可编址寄存器，并且，它接收形如常规机器语言的 S 机器程序。

9-2-2 机器寄存器

机器寄存器的设计和实现要考虑三方面：寄存器的数目，如何访问寄存器以及如何实现寄存器。寄存器的访问涉及到是否直接访问寄存器，例如在

LI, XPL

中，它可以被解释成“取出 XPL 的内容送入1号寄存器”；或者涉及到是否间接访问寄存器，例如当寄存器被当作堆栈来实现时，把取出的数值送入堆栈会使堆栈的其他数值下推。寄存器的数量显然涉及到物理元件或者逻辑元件的数量；然而，在堆栈机器中，寄存器系统常常这样设计：当堆栈存满时，就存入 S 存储器。机器寄存器可以用硬件、控制存储器中的单元，或者用 S 存储器中的单元来实现。在主计算机中，一般使用硬件寄存器和控制存储器，而把 S 存储器用作机器寄存器则只能通过微程序设计来实现。这里介绍的解释程序，其 S 存储器是用来实现机器寄存器的。

9-2-3 S 指令格式

S 指令的格式反映了计算机的操作原理，并间接地决定了操作码的范围，操作数寄存器以及寻址结构。由于 S 程序的指令是存放在 S 存储器中的，所以指令的长度也很重要，并且，在研制解释程序之前，就必须规定指令和计算机字之间的相互关系。操作码的长度给出了允许使用的不同操作码数目的上限——除非用一个特殊字段或者一个字来扩大操作码。例如，如果操作码有 n 位，则操作码的最大个数是 $2^n - 1$ 。但正如第二章所指出的，操作码中有时还包括指令的长度和操作类型指示（有如 360/370 系统那样）。因此，操作码除能提供操作说明外，还能提供别的有用信息。

当在执行指令的过程中要用到机器寄存器时，在相应的指令格式中应有特定的字段来指定寄存器号。指定寄存器号的字段长度实际上决定了所能使用的寄存器的个数，而指令的地址字段则同 S 存储器中能访问的存储区有关。当然，访问 S 存储器的地址是由寻址方式决定的，该寻址方式可以是基址/变址/位移量方式、变址/地址方式或者其他类似的方式。当采用基址/变址/位移量寻址方式时，在指令格式中通常包含有一个指定基址寄存器号的字段。但是，也可能在计算有效地址时，不由程序员来指定基址寄存器号，以及由他实现将数据装入到基址寄存器中去。换言之，由用户程序控制处理机时，基址寄存器的内容可由操作系统装入。

因此，有很多因素会影响计算机的操作特性，而计算机体系设计的任务就是考虑各部分因素并将它们有机的结合起来获得有效的操作特性。

9-2-4 S 机器操作

选择 S 机器的操作最终取决于计算机系统的设计目标。“取数”，“存数”等辅助操作总是需要的，但是否要包括浮点运算，这就要视使用要求而定。例如，如果设计一台控制用的小型计算机，那么就没有必要在计算机结构中设置浮点运算，更没有必要设置十进制运算，而使用 S 指令编制的程序来实现这些功能。同样，其他一些因素也可以部分地决定一组 S 机器操作。这包括诸如字长，微程序控制存储器的容量，计算机系统的成本，以及 S 机器和微程序设计的主计算机之间的关系。

9-3 堆栈机器的设计和实现

本节介绍堆栈机器的设计和实现。堆栈机器的含义如下：

1. 机器寄存器用一个堆栈来实现，并且，信息按“后进先出 (LIFO)^{*)}”方式进栈、出栈。

^{*)} 全文为 “last-in-first-out”。——译者注

2. S 指令的设计要适合于使用堆栈,并按波兰后缀表示法来解释 S 指令。

考虑到介绍基本概念比分析效率更为重要。所以,有些例子在设计时还可作进一步的改进。例如,在某些情况下,有些指令格式中包含有无用的字段。如果要设计一台实际的机器,那就要研究消去那些无用字段的方法。然而,总的看来,这个解释程序实际上还是有效的,它能仿真一台堆栈机器。这是基本目标。

9-3-1 波兰表示法

波兰表示法是一种无括号的表示法,它用来按单义方式表示算术/逻辑表达式,本书采用波兰后缀表示法。试看数学表达式 $A + B$,在波兰后缀表示法中,它被写成 $AB+$,两个操作数写在算符的前面。类似地,表达式 $(A + B) * (C - D)$,可以用波兰后缀表示法表示成 $AB + CD - *$,并不需要用括号。

波兰后缀表示法的执行要使用一个操作数堆栈。扫描一个用后缀表示法表示的表达式是从左到右进行的。如果遇到一个操作数,就把它推入堆栈。如果遇到一个算符,就针对栈顶的两个元素执行之。作为一个例子,语句

$$A = (B + C) * (D - E)$$

用后缀表示法表示即为

$$ABC + DE - * =$$

这个后缀字符串将按如下方式执行:

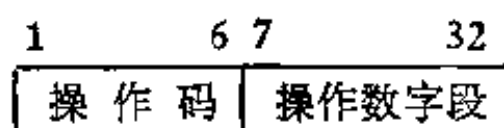
指令	后缀	操 作
1	A	把 A 的地址推入堆栈
2	B	把 B 的值推入堆栈
3	C	把 C 的值推入堆栈
4	+	从栈顶顺序退出两个单元的内容,使之相加,并把和数推入堆栈
5	D	把 D 的值推入堆栈

6	E	把 E 的值推入堆栈
7	—	从栈顶顺序退出两个单元的内容,使之相减,并把差数推入堆栈
8	*	从栈顶顺序退出两个单元的内容,使之相乘,并把乘积推入堆栈
9	=	退出值和地址,把值送入该地址单元中去

基于波兰后缀表示法的 S 指令也可用类似方式解释





9-3-2 指令格式和机器操作

用于堆栈机器的指令格式定义如下:



其中,操作码是个二进制数;操作数字段是一个地址或者一个数值,也可以不用。

定义了下列指令:

指 令	格 式	定 义
名字调用	000001 地址	把地址送入堆栈
值调用	000010 地址	把地址的内容送入堆栈
文字调用	000011 值	把值送入堆栈
存入	000100 	退出栈中的第一个元素(V) 退出栈中的第二个元素(A). (V) → (A)
加法	000101 	退出栈中的第一个元素(X) 退出栈中的第二个元素(Y) $X + Y \rightarrow \text{STACK}(\text{堆栈})$
减法	000110 	退出栈中的第一个元素(X) 退出栈中的第二个元素(Y) $X - Y \rightarrow \text{STACK}(\text{堆栈})$
转移	000111 地址	转移到 S 存储器中由地址所指出的单元
按“零”转移	001000 地址	退出栈中的第一个元素(X). 如果 $X \leq 0$, 则转移到指令中地址所指出的单元
停止	001001 	停止 S 机器的操作

这个指令系统是最基本的,但是很容易扩充它。我们将对这个指令系统的扩充作为一个问题留给读者进行练习。

9-3-3 机器寄存器的实现

堆栈常驻在 S 存储器中(从1号单元开始),且按地址递增的顺序排列。有一个栈顶指点器。当把一个数值推入堆栈时,指点器的值就增加1,然后这个值就被送入 S 存储器中由指点器指定的单元。当从堆栈退出一个数值时,就取出由指点器所指出的位于栈顶的值。然后把指点器减1。简言之是用栈顶上移,来代替把数值向下顺推。

9-3-4 操作上的约定

在实现解释程序时,采用下列约定:

寄存器 A1 指示现行指令地址。

寄存器 A2 指示栈顶。

寄存器 B 保存正被译码的指令。

S 存储器的 1 号单元是栈底。

S 存储器的 64 (+进制)号单元是装入 S 程序的起点。

寄存器 A3 用作累加暂存器。

S 存储器的 50 号单元用作出错指示器:“0”表示成功(即无错);全“1”表示出错。

9-3-5 堆栈机器的解释程序

在图 9-1 中,给出了堆栈机器的一个解释程序表,并在下面几节中介绍其各个微子程序。这个解释程序利用了第五章所给出的仿真技术。

下面列出一个地址目录表,它用作实现 S 机器指令的微程序的跳转表:

DIR^{*)}: PERR - 1 → AMPCR

^{*)} DIR 是目录 (Directory) 的缩写。——译者注

$PERR - 1 \longrightarrow AMPCR$
 $PNAM - 1 \longrightarrow AMPCR$
 $PVAL - 1 \longrightarrow AMPCR$
 $PLIT - 1 \longrightarrow AMPCR$
 $PST - 1 \longrightarrow AMPCR$
 $PADD - 1 \longrightarrow AMPCR$
 $PSUB - 1 \longrightarrow AMPCR$
 $PBR - 1 \longrightarrow AMPCR$
 $PBRZ - 1 \longrightarrow AMPCR$
 $PHLT - 1 \longrightarrow AMPCR$
 $PERR - 1 \longrightarrow AMPCR$

应当仔细地构造这个地址目录表。这是因为，它要被送入微程序存储器 (MPM) 的“0”号单元；而且，在取指令子程序中，要通过 EXEC 命令（带有操作码 AMPCR）来访问它。由于 EXEC 命令执行的是位于 $(AMPCR) + 1$ 处的指令，所以，用于一个操作的文字赋值语句必须置于比操作码大 1 的 MPM 地址单元中。例如，名字调入操作的操作码是 000001，相应的文字赋值语句（即 $PNAM - 1 \longrightarrow AMPCR$ ）就必须置于 MPM 的 2 号地址单元中。

初始化子程序把堆栈指点器置为初始值 0，并把现行地址指点器置为 64(十进制)，如下所述：

$INIT: \quad 0 \longrightarrow A2$
 $\quad \quad LIT \longrightarrow A1$
 $\quad \quad 64 \longrightarrow LIT$

堆栈指点器总是指向栈顶。

取指令子程序读出一条 S 指令，把现行地址寄存器的内容加 1，分离操作码，把相应的微子程序的 MPM 地址送入 AMPCR（用

EXEC 命令), 并把控制转向一个微子程序以执行 S 指令。取指令微子程序如下:

```
FETCH: A1  $\longrightarrow$  MAR1
MR1, A1 + 1  $\longrightarrow$  A1
WHEN RDC THEN BEX
B R  $\longrightarrow$  A3
26  $\longrightarrow$  SAR, 15  $\longrightarrow$  LIT
A3 AND LIT  $\longrightarrow$  AMPCR
EXEC
JUMP
```

这个取指令子程序把 S 指令保存在 B 寄存器中。

寻址子程序完成一种公共操作功能, 用来分离地址字段并按右边对齐的原则送入寄存器 A3。寻址子程序如下:

```
ADDR: B L  $\longrightarrow$  B
COMP 6  $\longrightarrow$  SAR
B R  $\longrightarrow$  A3
6  $\longrightarrow$  SAR
JUMP
```

它总是返回到始于 (AMPCR) + 1 的调用程序。

进栈子程序把堆栈指点器加 1, 并把要推入堆栈的值写入 S 存储器。进栈子程序如下:

```
STACK: A2 + 1  $\longrightarrow$  A2, MAR1
A3  $\longrightarrow$  MIR
MW1, IF SAI
FETCH - 1  $\longrightarrow$  AMPCR
WHEN SAI THEN JUMP
```

它总是返回到取指令子程序, 以便继续执行下一条指令。

退栈子程序退出栈顶的值, 并把它送入寄存器 A3。退栈子程序如下:

```
UNST: NOT A2  $\longrightarrow$ 
```

```

IF ABT THEN EXEC ELSE SKIP
JUMP
A2  $\longrightarrow$  MAR1
MR1, A2 + B111  $\longrightarrow$  A2
WHEN RDC THEN BEX
B  $\longrightarrow$  A3, RETN

```

如果堆栈是空的,则退栈子程序转移到一个错误出口。否则,把栈顶的值送入寄存器 A3,并把堆栈指点器的值减 1。这个子程序返回到始于 (AMPCR) + 2 的调用程序。

名字调用子程序把 S 指令的地址字段内容置入堆栈,并通过进栈子程序返回到取指令子程序:

```

FNAM: ADDR - 1  $\longrightarrow$  AMPCR
CALL
STACK - 1  $\longrightarrow$  AMPCR
JUMP

```

值调用子程序把由 S 指令指定的地址的内容置入堆栈,并经过进栈子程序返回到取指令子程序:

```

PVAL: ADDR - 1  $\longrightarrow$  AMPCR
CALL
A3  $\longrightarrow$  MAR1
MR1
WHEN RDC THEN BEX
STACK - 1  $\longrightarrow$  AMPCR
B  $\longrightarrow$  A3, JUMP

```

文字调用子程序把 S 指令中的值字段的内容置入堆栈,并由进栈子程序返回到取指令子程序,以执行下一条 S 指令。文字调用子程序如下:

```

PLIT: ADDR - 1  $\longrightarrow$  AMPCR
CALL
STACK - 1  $\longrightarrow$  AMPCR

```

JUMP

存入子程序把栈顶的值置入由堆栈的下一个元素所指定的地址单元中:

```
PST: UNST - 1 → AMPCR
      CALL
      PERR - 1 → AMP
          3 → MIR
      UNST - 1 → AMPCR
      CALL
      PERR - 1 → AMPCR
      A3 → MAR1
      MW1, IF SAI
      FETCH - 1 → AMPCR
      WHEN SAI THEN JUMP
```

这个存入子程序直接返回到取指令子程序.

加法子程序控制退出栈顶的两值,使之相加,并把和数送回堆栈. 加法子程序如下:

```
PADD: UNST - 1 → AMPCR
      CALL
      PERR - 1 → AMPCR
      A3 → MIR
      UNST - 1 → AMPCR
      CALL
      PERR - 1 → AMPCR
      BMI
      STACK - 1 → AMPCR
      A3 + B → A3, JUMP
```

这个加法子程序经过进栈子程序返回到取指令子程序,以执行下一条 S 指令.

减法子程序与加法子程序是相同的,只是它完成的是减法运

***** TRANSLANG D MACHINE MICROTRANSLATOR

ENTER FILENAME FOR HEX? PMHEX
 SUPPRESS BIT PATTERNS? (1=SUPPRESS)? 1
 SUPPRESS HEX LISTING? (1=SUPPRESS)? 1
 IF INPUT IN FILE ENTER 1? 01
 ENTER SOURCE FILENAME? PMACH

```

0000  DIR. PERR - 1 = AMPCR $
0001  PERR - 1 = AMPCR $
0002  PNAM - 1 = AMPCR $
0003  PVAL - 1 = AMPCR $
0004  PLIT - 1 = AMPCR $
0005  PST - 1 = AMPCR $
0006  PADD - 1 = AMPCR $
0007  PSUB - 1 = AMPCR $
0008  PBR - 1 = AMPCR $
0009  PBRZ - 1 = AMPCR $
0010  PHLT - 1 = AMPCR $
0011  PERR - 1 = AMPCR $
0012  INIT. 0 = A2 $ STACK PTR
0013  LIT = A1 $ CURRENT ADDR REGISTER
0014  64 = LIT $ PRG STARTS AT LOC 100 $CTAL
0015  FETCH. A1 = MARI $
0016  MRI. A1 + 1 = A1 $
0017  WHEN RDC THEN BEX $
0018  B R = A3 $
0019  26 = SAR. 15 = LIT $
0020  A3 AND LIT = AMPCR $
0021  EXEC $
0022  JUMP $
0023  ADDR. B L = B $
0024  COMP 6 = SAR $
0025  B R = A3 $
0026  6 = SAR $
0027  JUMP $
0028  STACK. A2 + 1 = A2. MARI $
0029  A3 = MIR $
0030  MWI. IF SAI $
0031  FETCH - 1 = AMPCR $
0032  WHEN SAI THEN JUMP $
0033  UNST. NOT A2 = $
0034  IF ABT THEN EXEC ELSE SKIP $
0035  JUMP $
0036  A2 = MARI $
0037  MRI. A2 + 0111 = A2 $
0038  WHEN RDC THEN BEX $
0039  B = A3. RETN $
0040  PNAM. ADDR - 1 = AMPCR $
0041  CALL $
0042  STACK - 1 = AMPCR $
0043  JUMP $
0044  PVAL. ADDR - 1 = AMPCR $
0045  CALL $
0046  A3 = MARI $
0047  MRI $
0048  WHEN RDC THEN BEX $
0049  STACK - 1 = AMPCR $
0050  B = A3. JUMP $
0051  PLIT. ADDR - 1 = AMPCR $
0052  CALL $
0053  STACK - 1 = AMPCR $
0054  JUMP $
0055  PST. UNST - 1 = AMPCR $
0056  CALL $
0057  PERR - 1 = AMPCR $
0058  A3 = MIR $
  
```

```

0059 UNST - 1 = AMPCR $
0060 CALL $
0061 PERR - 1 = AMPCR $
0062 A3 = MAR1 $
0063 MW1, IF SAI $
0064 FETCH - 1 = AMPCR $
0065 WHEN SAI THEN JUMP $
0066 PADD. UNST - 1 = AMPCR $
0067 CALL $
0068 PERR - 1 = AMPCR $
0069 A3 = MIR $
0070 UNST - 1 = AMPCR $
0071 CALL $
0072 PERR - 1 = AMPCR $
0073 BMI $
0074 STACK - 1 = AMPCR $
0075 A3 + B = A3, JUMP $
0076 PSUB. UNST - 1 = AMPCR $
0077 CALL $
0078 PERR - 1 = AMPCR $
0079 A3 = MIR $
0080 UNST - 1 = AMPCR $
0081 CALL $
0082 PERR - 1 = AMPCR $
0083 BMI $
0084 STACK - 1 = AMPCR $
0085 A3 - B = A3, JUMP $
0086 PBR. ADDR - 1 = AMPCR $
0087 CALL $
0088 FETCH - 1 = AMPCR $
0089 A3 = A1, JUMP $
0090 PBR2. ADDR - 1 = AMPCR $
0091 CALL $
0092 A3 = MIR $
0093 UNST - 1 = AMPCR $
0094 CALL $
0095 PERR - 1 = AMPCR $
0096 PBRZ1 - 1 = AMPCR $
0097 NOT A3 = $
0098 IF ABT THEN JUMP ELSE STEP $
0099 NOT A3 = $
0100 IF NOT NST THEN JUMP ELSE STEP $
0101 FETCH - 1 = AMPCR $
0102 JUMP $
0103 PBRZ1. BMI $
0104 FETCH - 1 = AMPCR $
0105 B = A1, JUMP $
0106 PHLT. 0 = MIR, LMAR, SKIP $
0107 PERR. NOT 0 = MIR, LMAR $
0108 50 = LIT $
0109 MW2, IF SAI $
0110 WHEN SAI THEN STEP $
0111 END $

```

THE TOTAL NUMBER OF ERRORS = 0

EXECUTE (Y/N)? N

图 9-1 堆栈机器解释程序的翻译程序表

算而已。减法子程序如下:

```
PSUS: UNST - 1 → AMPCR  
      CALL  
      PERR - 1 → AMPCR  
      A3 → MIR  
      UNST - 1 → AMPCR  
      CALL  
      PERR - 1 → AMPCR  
      BMI  
      STACK - 1 → AMPCR  
      A3 - B → A3, JUMP
```

转移子程序把有效地址置入现行地址寄存器,并返回到取指令子程序:

```
PBR: ADDR - 1 → AMPCR  
      CALL  
      FETCH - 1 → AMPCR  
      A3 → A1, JUMP
```

按 0 转移子程序,在栈顶值小于或等于 0 时,把有效地址置入现行地址寄存器。从堆栈退出的栈顶值被丢失,随后的操作不能使用它。按 0 转移子程序如下:

```
PBRZ: ADDR - 1 → AMPCR  
      CALL  
      A3 → MIR  
      UNST - 1 → AMPCR  
      CALL  
      PERR - 1 → AMPCR  
      PBRZ1 - 1 → AMPCR  
      NOT A3 →  
      IF ABT THEN JUMP ELSE STEP  
      NOT A3 →
```

2) DMACH1 (COMPILED) 02 JAN 76 15:12

ENTER SAME FILENAME FOR HEX? PMHEX
OUTPUT REGISTERS AND S MEMORY IN INTEGER(1) OR OCTAL(2)? 2
INPUT S MEMORY IN INTEGER(1) OR OCTAL IN G11 FORMAT(2)? 2
STARTING ADDRESS=? 12
MAXIMUM NUMBER OF CLOCKS TO SIMULATE=? 100
NUMBER OF CLOCKS BETWEEN OUTPUT POINTS=? 1

ENTER OUTPUT LINES DESIRED 1-ADDRESSES AND CLOCK
2- A1,A2,A3,B 3- MIR,SAR,LIT,CTR,AMPCR 4- BR1,BR2,MAR,BMAR,GC1,GC2
5- CONDITIONS
ENTER NUMBER OF OUTPUT LINES DESIRED? 5

BEGIN OUTPUT AT MPM ADDRESS=? 111
END OUTPUT AT MPM ADDRESS=? 111
ENTER 1 FOR S MEMORY DUMP WHEN PROGRAM TERMINATES? 1

ENTER S MEMORY VALUES IN CONSECUTIVE BLOCKS
ENTER 9999 FOR STARTING ADDRESS WHEN FINISHED

STARTING S MEMORY ADDRESS=? 64
FINAL S MEMORY ADDRESS FOR THIS BLOCK=? 68
SMEM(64)=? 00400000104
SMEM(65)=? 01400000002
SMEM(66)=? 02000000000
SMEM(67)=? 04400000000
SMEM(68)=? 00000000000

STARTING S MEMORY ADDRESS=? 9999

END OF SIMULATION - REGISTERS CONTAIN

P(1) ADDR. = 110 P(3) ADDR. = 110 CLOCK = 93
A1=00000000104 A2=00000000000 A3=00000000011 B=04400000000
MIR=00000000000 SAR=26 LIT=50 CTR=0 AMPCR=105
BR1=0 BR2=0 MAR=50 BMAR=50 GC1=0 GC2=0
LC1=0 LC2=0 MST=0 LST=0 ABT=0 ABV=0 CBV=0 SAI=1 RDC=1 INT=0

MEMORY DUMP REQUESTED

ENTER VALUES AS DONE IN MEMORY INPUT (9999, FOR STARTING ADDRESS WHEN FINISHED)

STARTING S MEMORY ADDRESS=? 50
FINAL S MEMORY ADDRESS FOR THIS BLOCK=? 50

S MEMORY(50) TO S MEMORY(50) =

00000000000

STARTING S MEMORY ADDRESS=? 64
FINAL S MEMORY ADDRESS FOR THIS BLOCK=? 68

S MEMORY(64) TO S MEMORY(68) =

00400000104 01400000002 02000000000 04400000000 00000000002

STARTING S MEMORY ADDRESS=? 9999

图 9-2 堆栈机器执行语言 $A = 2$ (已翻译为 S 语言) 的运行实例

```

IF NOT MOST THEN JUMP ELSE STEP
FETCH - 1 → AMPCR
JUMP

```

```

PBRZ1; BMI
    FETCH - 1 → AMPCR
    B → A1, JUMP

```

这个按 0 转移子程序总是返回到取指令子程序。

停止子程序把 0 置入 S 存贮器的 50 号单元并停止微程序的执行。错误子程序把全 1 置入 S 存贮器的 50 号单元,也停止微程序的执行。这两个子程序一起给出如下:

```

PHLT: 0 → MIR, LMAR, SKIP
PERR: NOT 0 → MIR, LMAR
      50 → LIT
      MW2, IF SAI
      WHEN SAI THEN STEP
      END

```

利用上述微子程序作为模型,就可以给堆栈机器增加其他的指令,有些已作为练习给出了。

9-3-6 运行实例

在准备一个运行实例时,必须先制定一组 S 指令,把它们变换成八进制格式,并在执行前把它们放入 S 存贮器。第一个例子执行“A ← 2”语句,它用后缀表示法表示成“A2←”。一组相应的 S 指令用符号和八进制格式表示如下:

十进制地址单元	八进制地址单元	符号指令	八进制指令
64	100	<i>name A</i>	0 0400000104
65	101	<i>litval 2</i>	0 1400000002
66	102	<i>store</i>	0 2000000000
67	103	<i>halt</i>	0 4400000000

68

104

(A)

0000000000

(存储器单元)

运行实例的打印输出结果如图 9-2 所示,其中, S 存储器的输出表明已成功地完成了替换操作。

第二个例子是执行下列语句:

$$A = 2$$

$$B = 3$$

$$C = A + B - 1$$

这些语句的后缀形式是:

$$A2 = B3 = CAB + 1 - =$$

一组相应的 S 指令用符号和八进制格式表示如下:

十进制地址单元	八进制地址单元	符号指令	八进制指令
64	100	<i>name A</i>	00400000116
65	101	<i>literal 2</i>	01400000002
66	102	<i>store</i>	02000000000
67	103	<i>name B</i>	00400000117
68	104	<i>literal 3</i>	01400000003
69	105	<i>store</i>	02000000000
70	106	<i>name C</i>	00400000120
71	107	<i>value A</i>	01000000116
72	110	<i>value B</i>	01000000117
73	111	<i>add</i>	02400000000
74	112	<i>literal 1</i>	01400000001
75	113	<i>subtract</i>	03000000000
76	114	<i>store</i>	02000000000
77	115	<i>halt</i>	04400000000
78	116	(A)	00000000000
79	117	(B)	00000000000
80	120	(C)	00000000000

运行实例的打印输出结果如图 9-3 所示,其中, S 存储器的输出表示已成功地完成了该计算。

作为最后一个例子,试考虑一个计算 $R = M \times N$ 的模拟乘法运算。这段程序用符号形式写出如下:

```

      M = 5
      N = 4
      T = 0
 $\alpha$  IF N = 0 THEN GO TO  $\beta$ 
      T = T + M
      N = N - 1
      GO TO  $\alpha$ 
 $\beta$   R = T
      HLT
```

组成这段程序的一组符号指令给出如下:

十进制地址单元	符号表示的 S 指令
64	<i>name</i> M
65	<i>literal</i> 5
66	<i>store</i>
67	<i>name</i> N
68	<i>literal</i> 4
69	<i>store</i>
70	<i>name</i> T
71	<i>literal</i> 0
72	<i>store</i>
73	<i>value</i> N
74	<i>bzero</i> 86
75	<i>name</i> T
76	<i>value</i> T
77	<i>value</i> M
78	<i>add</i>

2) DNACHI (COMPILED) 02 JAN 76 15:20

ENTER SAME FILENAME FOR NEXT PMKEX
OUTPUT REGISTERS AND S MEMORY IN INTEGER(1) OR OCTAL(2)? 2
INPUT S MEMORY IN INTEGER(1) OR OCTAL IN Q11 FORMAT(2)? 2
STARTING ADDRESS =? 12
MAXIMUM NUMBER OF CLOCKS TO SIMULATE=? 400
NUMBER OF CLOCKS BETWEEN OUTPUT PRINTS=? 1

ENTER OUTPUT LINES DESIRED 1-ADDRESSES AND CLOCK
2- A1,A2,A3,B 3- MIR,SAR,LIT,CTR,AMPCR 4- BR1,BR2,MAR,DMAR,OC1,OC2
5- CONDITIONS
ENTER NUMBER OF OUTPUT LINES DESIRED? 5

BEGIN OUTPUT AT MPN ADDRESS=? 111
END OUTPUT AT MPN ADDRESS=? 111
ENTER 1 FOR S MEMORY DUMP WHEN PROGRAM TERMINATES? 1

ENTER S MEMORY VALUES IN CONSECUTIVE BLOCKS
ENTER 9999 FOR STARTING ADDRESS WHEN FINISHED

STARTING S MEMORY ADDRESS=? 64
FINAL S MEMORY ADDRESS FOR THIS BLOCK=? 80
SMEM(64)=? 00400000116
SMEM(65)=? 01400000002
SMEM(66)=? 02000000000
SMEM(67)=? 00400000117
SMEM(68)=? 01400000003
SMEM(69)=? 02000000000
SMEM(70)=? 00400000120
SMEM(71)=? 01000000116
SMEM(72)=? 01000000117
SMEM(73)=? 02400000000
SMEM(74)=? 01400000001
SMEM(75)=? 03000000000
SMEM(76)=? 02000000000
SMEM(77)=? 04400000000
SMEM(78)=? 00000000000
SMEM(79)=? 00000000000
SMEM(80)=? 00000000000

STARTING S MEMORY ADDRESS=? 9999

END OF SIMULATION - REGISTERS CONTAIN

PC(1) ADDR. = 110 PC(3) ADDR. = 110 CLOCK = 367
A1=00000000116 A2=00000000000 A3=0000000011 B=04400000000
MIR=00000000000 SAR=26 LIT=50 CTR=0 AMPCR=105
BR1=0 BR2=0 MAR=50 DMAR=50 OC1=0 OC2=0
LC1=0 LC2=0 MST=0 LST=0 ABT=0 AGV=0 CGV=0 SAI=1 RDC=1 INT=0

MEMORY DUMP REQUESTED

ENTER VALUES AS DONE IN MEMORY INPUT (9999 FOR STARTING ADDRESS WHEN FINISHED)

STARTING S MEMORY ADDRESS=? 50
FINAL S MEMORY ADDRESS FOR THIS BLOCK=? 50

S MEMORY(50) TO S MEMORY(50) =


```

0000000000
STARTING S MEMORY ADDRESS=7 64
FINAL S MEMORY ADDRESS FOR THIS BLOCK=7 80

S MEMORY( 64) TO S MEMORY( 80) =

00400000116 01400000002 02000000000 00400000117 01400000003
02000000000 00400000120 01000000116 01000000117 02400000000
01400000001 03000000000 02000000000 04400000000 00000000002
00000000003 00000000004
STARTING S MEMORY ADDRESS=7 9999

```

图 9-3 堆栈机器执行 (语句 $A = 2$, $B = 3$ 以及 $C = A + B - 1$ 的) S 指令的运行实例

79	<i>store</i>
80	<i>name</i> N
81	<i>value</i> N
82	<i>literal</i> 1
83	<i>subtract</i>
84	<i>store</i>
85	<i>branch</i> 73
86	<i>name</i> R
87	<i>value</i> T
88	<i>store</i>
89	<i>halt</i>
90	(M)
91	(N)
92	(T)
93	(R)

把这段 S 程序翻译成八进制格式并借助模拟程序来运行它, 这一工作我们都作为练习留给读者去做。这段程序看来不是高效率的, 至少在 S 指令的数目方面是如此。部分原因是, 只有栈顶寄

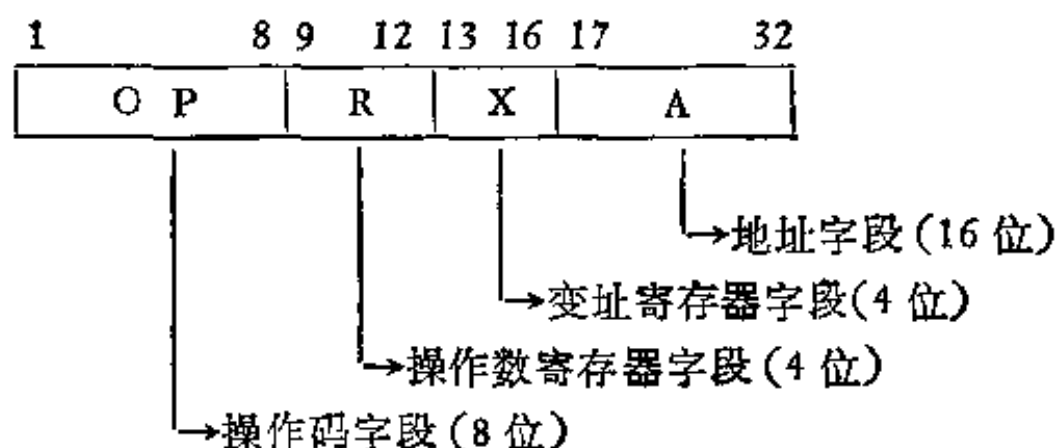
寄存器的存取才是高效率的;而实 S 存储器必须用于中间存贮,这就导致大量的数值调用和写入操作。

9-4 多寄存器计算机的设计和实现

这一节介绍一台多寄存器计算机,它类似于第五章中被仿真的 16 位假想计算机。这台计算机由于使用了一组(共 16 个)通用寄存器而有其特点,这组通用寄存器用于算术/逻辑运算和寻址。这台计算机是常规的取/加法/存方式的计算机,它类似于 360/370 系统或者 PDP11 那种众所周知的计算机。

9-4-1 指令格式和机器操作

这台多寄存器计算机的指令格式定义如下:



8 位的操作码字段可以有 255 个不同的操作码,而每个寄存器字段则允许指定 16 个通用寄存器中的任何一个。计算机中每个寄存器 32 位宽,因此可以产生 32 位的有效地址。16 位的地址字段可以解释为一个地址或者一个位移量。类似地,4 位的变址寄存器字段可以指定一个变址寄存器或者基址寄存器。而 4 位的操作数寄存器字段总是指定一个累加器。

定义下面的指令:

指 令	格	式	定 义
取 数	00000001	R X A	$(A + (X)) \rightarrow (R)$

写 入	<table><tr><td>00000010</td><td>R</td><td>X</td><td>A</td></tr></table>	00000010	R	X	A	$(R) \rightarrow (A + (X))$
00000010	R	X	A			
加 法	<table><tr><td>00000011</td><td>R</td><td>X</td><td>A</td></tr></table>	00000011	R	X	A	$(R) + (A + (X)) \rightarrow (R)$
00000011	R	X	A			
减 法	<table><tr><td>00000100</td><td>R</td><td>X</td><td>A</td></tr></table>	00000100	R	X	A	$(R) - (A + (X)) \rightarrow (R)$
00000100	R	X	A			
转 移	<table><tr><td>00000101</td><td>R</td><td>X</td><td>A</td></tr></table>	00000101	R	X	A	转移到 $A + (X)$
00000101	R	X	A			
按“0”转移	<table><tr><td>00000110</td><td>R</td><td>X</td><td>A</td></tr></table>	00000110	R	X	A	若 $(R) \leq 0$ 则转移到 $A + (X)$
00000110	R	X	A			
停 止	<table><tr><td>00000111</td><td>R</td><td>X</td><td>A</td></tr></table>	00000111	R	X	A	停止 S 机器
00000111	R	X	A			

如同堆栈机器那样,可以很容易地扩充多寄存器机器的指令系统.

9-4-2 机器寄存器的实现

16个通用寄存器保存在从 256 号单元开始的 S 存贮器中. 把寄存器置于一个有 256 位的字块边界的优点是, 二进制地址的最右边的 8 位为全 0 时对应于 0 号寄存器. 类似地, 第 257 号字的二进制地址对应于 1 号寄存器, 该地址的最右位是 1. 16 个通用寄存器的 S 存贮器地址划分如下:

00000001	0000	XXXX
----------	------	------

这样, 当必需访问一个寄存器时, 就可以把左半地址置入 BR1 寄存器, 把右半地址传送到 MAR.

9-4-3 操作上的约定

在实现解释程序时, 采用了下列操作约定:

寄存器 A1 指示现行指令地址.

寄存器 A2 保存正在执行的指令.

寄存器 BR1 保存通用寄存器中用来指示 S 存贮器的地址

的高 8 位。

S 存贮器的第 1024 号单元(十进制)是装入 S 程序的起始点。

S 存贮器的第 256 号单元是在 S 存贮器中连续存放的 16 个通用寄存器的起始单元。

S 存贮器的 50 号单元用作错误指示器: 0 表示没有错误; 全 1 表示有错误。

寄存器 A3 用作中间累加器。

9-4-4 多寄存器机器的解释程序

图 9-4 中给出了多寄存器机器的解释程序表, 而各子程序则在下一节中介绍。解释程序使用第五章中给出的仿真技术。

下面列出一个目录表, 它用作实现 S 机器指令的微子程序的跳转表:

DIR:	VERR	— 1	→	AMPCR
	VERR	— 1	→	AMPCR
	VLD	— 1	→	AMPCR
	VST	— 1	→	AMPCR
	VADD	— 1	→	AMPCR
	VSUB	— 1	→	AMPCR
	VBR	— 1	→	AMPCR
	VBRZ	— 1	→	AMPCR
	VHLT	— 1	→	AMPCR
	VERR	— 1	→	AMPCR

这个目录表是按照如同堆栈机器目录表一样的方式构成的。对于具有操作码为 n 的指令的跳转表入口必须放在 MPM 的 $n + 1$ 号单元中。

初始化子程序把 BR1 置位到使它指向 S 存贮器中的通用寄存器, 并把现行地址指点器的初值置成 1024 (十进制)。初始化子程序所列:

```

INIT: AMPCR → BR1
      256 → AMPCR
      AMPCR → A1
      1024 → AMPCR

```

取指令子程序的功能是取出下一条 S 指令，把现行地址寄存器的内容加 1，分离出变址寄存器字段并从 S 存储器中取出其内容，分离出操作码字段并把与它相应的微子程序在 MPM 中的地址置于 AMPCR，形成有效地址并把它置入寄存器 A3，使操作数寄存器字段在寄存器 A2 中靠右边对齐，以及把控制转移到所要执行的 S 指令的微子程序。取指令子程序列出如下：

```

FETCH: A1 → MAR2
      MR2, A1 + 1 → A1
      WHEN RDC THEN BEX
      B C → A2
      16 → SAR, 15 → LIT
      A2 AND LIT → MAR
      MR1, B R → AMPCR % READ INDEX, OP
      CODE TO AMPCR
      24 → SAR
      A2 R → A3, EXEC % ADDR FIELD TO A3
      16 → SAR
      WHEN RDC THEN BEX, A2 C → A2
      % RT JUSTIFY IN A2
      4 → SAR
      A3 + B → A3, MAR2, JUMP % EFF ADDR
      IN A3

```

这个取指令子程序把 S 指令保留在寄存器 A2 内，把有效地址保留在寄存器 A3。

取数子程序从 S 存储器中读出操作数，并把它送入 MIR，随后写入到 S 存储器中被指定的操作数寄存器单元。在操作数被写入

>>>> TRANSLANG D MACHINE MICROTRANSLATOR

ENTER FILENAME FOR HEX? MRHEX
 SUPPRESS BIT PATTERNS?(1=SUPPRESS)? 1
 SUPPRESS HEX LISTING?(1=SUPPRESS)? 1
 IF INPUT IN FILE ENTER 1? 31
 ENTER SOURCE FILENAME? MRMACH

```

0000  DIR, VERR - 1 = AMPCR ?
0001  VERR - 1 = AMPCR $
0002  VLD - 1 = AMPCR $
0003  VST - 1 = AMPCR $
0004  VADD - 1 = AMPCR $
0005  VSUB - 1 = AMPCR $
0006  VBR - 1 = AMPCR $
0007  VBRZ - 1 = AMPCR $
0008  VHLT - 1 = AMPCR $
0009  VERR - 1 = AMPCR $
0010  INIT, AMPCR = BR1 $
0011  256 = AMPCR $
0012  AMPCR = A1 $
0013  1024 = AMPCR $
0014  FETCH, A1 = MAR2 $
0015  MR2, A1 + 1 = A1 $
0016  WHEN RDC THEN BEX $
0017  B C = A2 $
0018  16 = SAR, 15 = LIT $
0019  A2 AND LIT = MAR $
0020  MRI, B R = AMPCR $ READ INDEX, OP CODE TO AMPCR
0021  24 = SAR $
0022  A2 R = A3, EXEC $ ADDR FIELD TO A3
0023  15 = SAR $
0024  WHEN RDC THEN A2 C = A2, BEX $ RT JUSTIFY R IN A2
0025  4 = SAR $
0026  A3 + B = A3, MAR2, JUMP $ EFF ADDR IN A3
0027  VLD, MR2 $
0028  WHEN RDC THEN A2 AND LIT = MAR, BEX $
0029  15 = LIT $
0030  B = MIR $
0031  MW1, IF SAI $
0032  FETCH - 1 = AMPCR $
0033  WHEN SAI THEN JUMP $
0034  VST, A2 AND LIT = MAR $
0035  15 = LIT $
0036  MRI $
0037  WHEN RDC THEN A3 = MAR2, BEX $
0038  B = MIR $
0039  MW2, IF SAI $
0040  FETCH - 1 = AMPCR $
0041  WHEN SAI THEN JUMP $
0042  VADD, A2 AND LIT = A2, MAR $
0043  15 = LIT $
0044  MRI $
0045  WHEN RDC THEN A3 = MAR2, BEX $
0046  B = A3, MR2 $
0047  WHEN RDC THEN A2 = MAR, BEX $
0048  A3 + B = MIR $
0049  MW1, IF SAI $
0050  FETCH - 1 = AMPCR $
0051  WHEN SAI THEN JUMP $
0052  VSUB, A2 AND LIT = A2, MAR $
0053  15 = LIT $
0054  MRI $
0055  WHEN RDC THEN A3 = MAR2, BEX $
0056  B = A3, MR2 $
0057  WHEN RDC THEN A2 = MAR, BEX $
0058  A3 - B = MIR $
  
```

```

0059      MW1, IF SAI S
0060      FETCH - 1 = AMPCR S
0061      WHEN SAI THEN JUMP S
0062      VBR. FETCH - 1 = AMPCR S
0063      A3 = A1, RETN S
0064      VBRZ. A2 AND LIT = MAR S
0065      15 = LIT S
0066      MR1 S
0067      WHEN RDC THEN BEX S
0068      FETCH - 1 = AMPCR S
0069      NOT B = S
0070      IF APT THEN A3 = A1, JUMP ELSE STEP S
0071      B = S
0072      IF MST THEN A3 = A1, JUMP ELSE JUMP S
0073      VHLT. 0 = MIR, LMAR, SKIP S
0074      VERR. NOT 0 = MIR, LMAR S
0075      50 = LIT S
0076      MW2, IF SAI S
0077      WHEN SAI THEN STEP S
0078      END S

```

THE TOTAL NUMBER OF ERRORS = 0

EXECUTE (Y/N)? N

图 9-4 多寄存器机器的解释程序的翻译程序表

所指定的通用寄存器之后,取数子程序返回到取指令子程序,取数子程序列出如下:

VLD: MR2

```

      WHEN RDC THEN A2 AND LIT → MAR, BEX
      15 → LIT
      B → MIR
      MW1, IF SAI
      FETCH - 1 → AMPCR
      WHEN SAI THEN JUMP

```

应当注意,因为机器寄存器是包含在 S 存贮器之中,所以,在执行一条指令期间所用到的寄存器的内容,在执行完该指令之后,总是要写到 S 存贮器的。

存入子程序的功能是读出所指定的寄存器的内容并把它放到 MIR. 然后,把它写入由寄存器 A3 中的有效地址所指出的 S 存贮器单元. 而后,存入子程序它返回到取指令子程序,存入子程序列

出如下:

```
VST: A2 AND LIT→MAR
      15→LIT
      MR1
      WHEN RDC THEN BEX, A3→MAR2
      B→MIR
      MW2, IF SAI
      FETCH - 1→AMPCR
      WHEN SAI THEN JUMP
```

加法以及减法子程序的功能是读出有效地址所指出的 S 存贮单元的内容和指定的寄存器的内容,然后,分别执行加法或减法,进而把和数或差数写入 S 存贮器中寄存器单元。最后,加法或减法子程序返回到取指令子程序,列出如下:

```
VADD: A2 AND LIT→A2, MAR
      15→LIT
      MR1
      WHEN RDC THEN BEX, A3→MAR2
      B→A3, MR2
      WHEN RDC THEN BEX, A2→MAR
      A3 + B→MIR
      MW1, IF SAI
      FETCH - 1→AMPCR
      WHEN SAI THEN JUMP
```

```
VSUB: A2 AND LIT→A2, MAR
      15→LIT
      MR1
      WHEN RDC THEN BEX, A3→MAR2
      B→A3, MR2
      WHEN RDC THEN BEX, A2→MAR
      A3 - B→MIR
```



```

MW1, IF SAI
FETCH - 1 → AMPCR
WHEN SAI THEN JUMP

```

加法和减法子程序的简单性表明，采用补码运算的假定是合理的。而我们可以很容易地用这里给出的两个子程序 VADD 和 VSUB 来替换前面讨论过的原码运算的子程序。

转移子程序，

```

VBR:  FETCH - 1 → AMPCR
        A3 → A1, RETN

```

把有效地址传送到现行地址寄存器，并返回到取指令子程序。当取指令子程序从转移子程序接收到控制时，就不执行第一条指令。这是因为取指令子程序的最后一个语句 ($A3 + B \rightarrow A3$, MAR2, JUMP) 也要装入 MAR2。而这是在取指令子程序中专门为转移子程序设置的，在别处并不使用它。可是，由于使用了水平微程序设计的机器，所以，不需要增加操作就能有效地装入寄存器。

如果被指定的寄存器的内容小于或者等于 0，则按 0 转移子程序将把有效地址送到现行地址寄存器。而这个子程序在进行条件测试之前，就读出 S 存储器中寄存器单元的内容，列出如下：

```

VBRZ:  A2 AND LIT → MAR
        15 → LIT
        MR1
        WHEN RDC THEN BEX
        FETCH - 1 → AMPCR
        NOT B →
        IF ABT THEN A3 → A1, JUMP ELSE STEP
        B →
        IF MST THEN A3 → A1, JUMP ELSE JUMP

```

不管小于或等于 0 测试的结果如何，按 0 转移子程序总是要返回到取指令子程序。

停止子程序把 0 置入 S 存储器的 50 号单元，并停止执行微程

2) DMACH1 (COMPILED) 02 JAN 76 16:53

ENTER SAME FILENAME FOR NEXT MRMEX

OUTPUT REGISTERS AND S MEMORY IN INTEGER(1) OR OCTAL(2)? 2

INPUT S MEMORY IN INTEGER(1) OR OCTAL IN OII FORMAT(2)? 2

STARTING ADDRESS =? 10

MAXIMUM NUMBER OF CLOCKS TO SIMULATE=? 100

NUMBER OF CLOCKS BETWEEN OUTPUT POINTS=? 1

ENTER OUTPUT LINES DESIRED 1-ADDRESSES AND CLOCK

2- A1,A2,A3,B 3- MIR,SAR,LIT,CTR,AMPCR 4- BR1,BR2,MAR,BMAR,GC1,GC2

5- CONDITIONS

ENTER NUMBER OF OUTPUT LINES DESIRED? 5

BEGIN OUTPUT AT MPH ADDRESS=? 76

END OUTPUT AT MPH ADDRESS=? 76

ENTER 1 FOR S MEMORY DUMP WHEN PROGRAM TERMINATES? 1

ENTER S MEMORY VALUES IN CONSECUTIVE BLOCKS

ENTER 9999 FOR STARTING ADDRESS WHEN FINISHED

STARTING S MEMORY ADDRESS=? 1024

FINAL S MEMORY ADDRESS FOR THIS BLOCK=? 1028

SMEM(1024)=? 00110002003

SMEM(1025)=? 00210002004

SMEM(1026)=? 00700000000

SMEM(1027)=? 00000000002

SMEM(1028)=? 00000000000

STARTING S MEMORY ADDRESS=? 9999

END OF SIMULATION - REGISTERS CONTAIN

P(1) ADDR. = 77 P(3) ADDR. = 77 CLOCK = 68

A1=00000002003 A2=0000000160 A3=00000000000 B=00000000000

MIR=00000000000 SAR=4 LIT=50 CTR=0 AMPCR=72

BR1=1 BR2=0 MAR=50 BMAR=50 GC1=0 GC2=0

LC1=0 LC2=0 HST=0 LST=0 ABT=0 ASV=0 COV=0 SAI=1 RDC=1 INT=0

MEMORY DUMP REQUESTED

ENTER VALUES AS DONE IN MEMORY INPUT (9999 FOR STARTING ADDRESS WHEN FINISHED)

STARTING S MEMORY ADDRESS=? 50

FINAL S MEMORY ADDRESS FOR THIS BLOCK=? 50

S MEMORY(50) TO S MEMORY(50) =

00000000000

STARTING S MEMORY ADDRESS=? 1024

FINAL S MEMORY ADDRESS FOR THIS BLOCK=? 1028

S MEMORY(1024) TO S MEMORY(1028) =

00110002003 00210002004 00700000000 00000000002 00000000000

STARTING S MEMORY ADDRESS=? 9999

图 9-5 多寄存器机器执行语句 $A = 2$ (已翻译成 S 语言) 的运行实例

2) DMACH1 (COMPILED) 02 JAN 76 17:18

ENTER SAME FILENAME FOR HEX? MRHEX
OUTPUT REGISTERS AND S MEMORY IN INTEGER(1) OR OCTAL(2)? 2
INPUT S MEMORY IN INTEGER(1) OR OCTAL IN 011 FORMAT(2)? 2
STARTING ADDRESS=? 10
MAXIMUM NUMBER OF CLOCKS TO SIMULATE=? 750
NUMBER OF CLOCKS BETWEEN OUTPUT POINTS=? 1

ENTER OUTPUT LINES DESIRED 1-ADDRESSES AND CLOCK
2- A1,A2,A3,B 3- MIR,SAR,LIT,CTR,AMPCR 4- BR1,BR2,MAR,BMAR,GC1,GC2
5- CONDITIONS
ENTER NUMBER OF OUTPUT LINES DESIRED? 5

BEGIN OUTPUT AT MPN ADDRESS=? 78
END OUTPUT AT MPN ADDRESS=? 78
ENTER 1 FOR S MEMORY DUMP WHEN PROGRAM TERMINATES? 1

ENTER S MEMORY VALUES IN CONSECUTIVE BLOCKS
ENTER 9999 FOR STARTING ADDRESS WHEN FINISHED

STARTING S MEMORY ADDRESS=? 1024
FINAL S MEMORY ADDRESS FOR THIS BLOCK=? 1042
SMEM(1024)=? 00110002022
SMEM(1025)=? 00210002014
SMEM(1026)=? 00110002021
SMEM(1027)=? 00210002015
SMEM(1028)=? 00110002017
SMEM(1029)=? 00114002015
SMEM(1030)=? 00614002012
SMEM(1031)=? 00310002014
SMEM(1032)=? 00414002020
SMEM(1033)=? 00500002006
SMEM(1034)=? 00210002016
SMEM(1035)=? 00700000000
SMEM(1036)=? 00000000000
SMEM(1037)=? 00000000000
SMEM(1038)=? 00000000000
SMEM(1039)=? 00000000000
SMEM(1040)=? 00000000001
SMEM(1041)=? 00000000004
SMEM(1042)=? 00000000005

STARTING S MEMORY ADDRESS=? 9999

END OF SIMULATION - REGISTERS CONTAIN

PC(1) ADDR. = 77 PC(3) ADDR. = 77 CLOCK = 540
A1=00000002014 A2=00000000160 A3=00000000000 B=00000000000
MIR=00000000000 SAR=4 LIT=50 CTR=0 AMPCR=72
BR1=1 BR2=0 MAR=50 BMAR=50 GC1=0 GC2=0
LC1=0 LC2=0 MST=0 LST=0 ABT=0 ADV=0 COV=0 SAI=1 RDC=1 INT=0

MEMORY DUMP REQUESTED

ENTER VALUES AS DONE IN MEMORY INPUT (9999 FOR STARTING ADDRESS WHEN FINISHED)

STARTING S MEMORY ADDRESS=? 50
FINAL S MEMORY ADDRESS FOR THIS BLOCK=? 50

```

S MEMORY( 50) TO S MEMORY( 50) =

00000000000
STARTING S MEMORY ADDRESS=? 1024
FINAL S MEMORY ADDRESS FOR THIS BLOCK=? 1042

S MEMORY(1024) TO S MEMORY(1042) =

00110002022 00210002014 00110002021 00210002015 00110002017
00114002015 00614002012 00310002014 00414002020 00500002006
00210002016 00700000000 00000000005 00000000004 00000000024
00000000000 00000000001 00000000004 00000000005
STARTING S MEMORY ADDRESS=? 9999

```

图 9-6 多寄存器机器的解释程序解释两个整数相乘的一串 S 指令的运行实例

序，“错误”子程序把全 1 置入 S 存贮器的 50 号单元，并且也停止执行微程序。这两个子程序合起来列在下面：

```

VHLT: 0 → MIR, LMAR, SKIP
VERR: NOT 0 → MIR, LMAR
      50 → LIT
      MW2, IF SAI
      WHEN SAI THEN STEP
      END

```

解释程序的一个特点是，它几乎全面地综合诸如操作原理，指令格式，机器操作，以及操作约定等因素，以构成一个完整的计算机体系结构，在这个结构中，各种因素互相补充。然而，解释程序的可行性和效率，却明显地取决于这种计算机结构与宿主机中用微程序设计的处理机与主存贮器（亦即 S 存贮器）之间的匹配程序。

9-4-5 运行实例

用于多寄存器机器的解释程序的第一个运行实例是执行语句 $A = 2$ ，这个语句被翻译成符号形式和八进制格式的 S 指令后如下：

十进制 单元号	八进制 单元号	符号指令	八进制指令
1024	2000	LD 2, TWO	00110002003
1025	2001	ST 2, A	00210002004
1026	2002	HLT	00700000000
1027	2003	(TWO)	00000000002
1028	2004	(A)	00000000000

在图 9-5 中,给出这个运行实例的打印输出,其中 S 存贮器的输出表示: 替换操作已完成。

第二个例子是执行前已给出的、形如 $R = M \times N$ 的模拟乘法操作。该程序段用符号形式写出如下:

```

M = 5
N = 4
T = 0
α IF N = 0 THEN GO TO β
  T = T + M
  N = N - 1
  GO TO α
β R = T
HLT

```

这个程序用机器语言编码,而 S 指令则用符号形式和八进制格式给出如下:

十进制 单元号	八进制 单元号	符号指令	八进制指令
1024	2000	LD 2, FIVE	00110002022
1025	2001	ST 2, M	00210002014
1026	2002	LD 2, FOUR	00110002021
1027	2003	ST 2, N	00210002015
1028	2004	LD 2, ZERO	00110002017
1029	2005	LD 3, N	00114002015

1030	2006	BZERO 3,1034	00614002012
1031	2007	A 2, M	00310002014
1032	2010	S 3, ONE	00414002020
1033	2011	BR 1030	00500002006
1034	2012	ST 2, R	00210002016
1035	2013	HLT	00700000000
1036	2014	(M)	00000000000
1037	2015	(N)	00000000000
1038	2016	(R)	00000000000
1039	2017	(ZERO)	00000000000
1040	2020	(ONE)	00000000001
1041	2021	(FOUR)	00000000004
1042	2022	(FIVE)	00000000005

在图 9.6 中, 给出这个运行实例的打印输出, 其中 S 存贮器的输出表示: 已得到乘积 $M \times N$.

就 S 指令的数目而言, S 语言程序的多寄存器机器的翻译效率要比堆栈机器的翻译效率高. 由于可以访问每一个通用寄存器, 所以可以把这些寄存器用作暂时存贮器.

词 汇

读者应当熟悉本章中所使用的下列术语:

解释程序 (Interpreter)

堆栈机 (Stack machine)

波兰后缀表示法 (Polish postfix notation)

指令格式 (Instruction format)

名字调用 (Name call)

值调用 (Value call)

取指令 (Instruction fetch)

(地址)目录 (Directory)

跳转表 (Jump table)

多寄存器机器 (Multiple-register machine)

现行指令地址 (Current-instruction address)

提 问

下面一些问题打算用来测验你对主题内容的理解。所有问题都可以直接从课文或者通过对所提出的主题进行逻辑推理而得到答案。有些问题适合于在讨论班上研究。

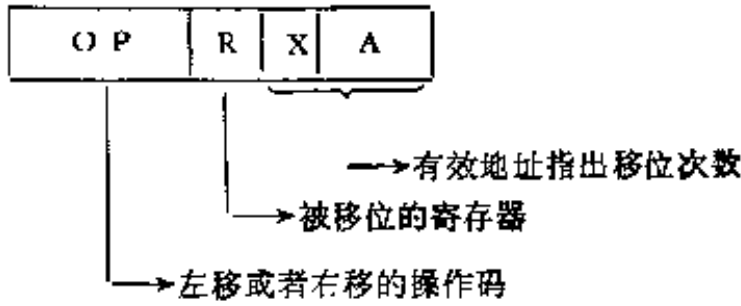
1. 机器寄存器的设计与使用以什么方式影响指令格式的设计?
2. 试举出实现机器寄存器的三种方法。
3. 波兰后缀表示法与操作数堆栈的用法之间有什么关系?
4. 试口头描述 360/370 计算机系统的操作码格式(参看第二章)。
5. 如何消去 S 指令格式中的无用字段?
6. 在堆栈机器的解释程序中, EXEC 语句起什么作用?
7. 名字调用和文字调用指令是用同样方式实现的。在实际的计算机中, 试解释它们如何能用不同的方式实现。
8. 在堆栈机器中, 执行 $A = 2$ 需要多少个时钟周期? 在多寄存器机器中呢? 你能说明其差别吗?
9. 试考虑多寄存器机器的解释程序。为什么转移子程序使用 RETN 而不使用 JUMP?

习 题

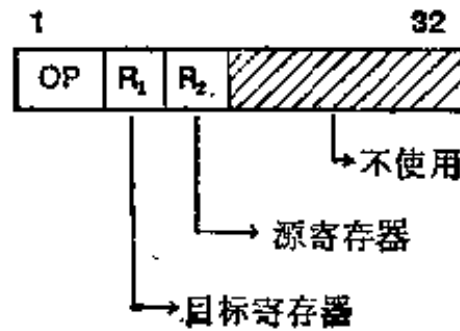
1. 试列出堆栈机器和多寄存器机器操作上的优、缺点。
2. 针对本章所给出的 $R = M \times N$ 的乘法例子, 依靠翻译程序/模拟程序来运行堆栈机器的解释程序。试比较其所需要的时钟数目与多寄存器机器所需要的时钟数目。
3. 在堆栈机器和多寄存器机器上, 为下列语句编出 S 指令并执行之:
 $A = 10$
 $B = 5$
 $C = 2$
 $D = A - B + C - 4$
4. 针对堆栈机器和多寄存器机器, 试把第八章所给出的求平方根的算法编成 S 指令。根据每一种解释程序来计算同一值(例如 25)的平方根。试比较所使用的时钟数目。

问 题

1. 使用如下格式, 为多寄存器机器增加一条移位指令:



2. 为堆栈机器增加浮点运算, 并针对求平方根的算法来测试它。以此完成加、减、乘、除运算。
3. 为多寄存器机器增加带符号的数值运算, 以此完成加法和减法运算。
4. 为多寄存器机器增加定点乘法和除法, 把一对奇/偶寄存器用作为 AC/MQ 寄存器。
5. 为多寄存器机器增加寄存器-寄存器操作。实现下列操作: 取数、加法、减法、以及取正数。使用下面的指令格式:



第十章 微程序设计的其他问题

10-1 概 述

本章是这本书的最后一章,它将提供有关定时的补充资料,并给出一组特殊问题的微程序设计,而这些问题可用于在写出它们的微程序之前,要求详细地分析这些问题,在这个意义上,它们类似于第九章中所给出的解释程序。

10-2 定 时

计算机是根据所发出的时钟脉冲并通过把信息从一个地方传送到另一个地方来进行连续操作的。因为计算机是确定的,所以操作总是按既定的顺序发生的。这样,微程序设计的任务就是编制微指令序列,使之得到所需的结果。

10-2-1 时钟脉冲

使D机器操作概念化的一种方便方法是,把时钟脉冲看作使信息从一处传送到另一处的一种工具。换言之,信息的传送是在一个时钟脉冲周期内进行的。图10-1给出关于微指令定时的概念图。重要的是要注意到:在每个时钟间隔里,一条指令的相位1周期同另一条指令的相位2或者相位3周期都在执行。



图 10-1 微指令的定时

10-2-2 指令相位的描述

对于一条 II 型指令，一个时钟间隔提供了足够的时间来完成文字赋值以及建立后继控制，这个后继控制总是步进到微程序存储器中的下一条指令。

对于一条 I 型指令，在相位 1 和相位 3 以及在这两个相位之后，完成下述基本功能：

在相位 1 期间

选择条件 (AOV, ABT, MST 等)。

确定被测试的是真条件还是假条件。

在相位 1 结束时

决定后继命令(“真”或“假”的后继命令)。有条件地更新命令寄存器。

启动外部操作(例如: MW1)。

修正条件(例如: SET LC1)

在相位 2 期间(保持相位)

上一次逻辑操作可随即用于下一个相位 1 周期的测试。

在相位 3 期间

选择加法器的 X 输入。

选择加法器的 Y 输入。

完成加法器操作或逻辑操作(动态条件可用于与它相重叠的相位 1 中的条件测试)。

移位操作。

在相位 3 结束时

改变目标寄存器

上述过程概括在图 10-2 中，并作如下说明：

1. 除了条件不成立(即“假”条件)且逻辑部件的操作是有条件的以外，I 型指令的相位 1 总是接着相位 3。如果没有指出逻辑部件的操作，则执行形如 $0 + B_{000} \rightarrow$ 这个空操作。

2. 寄存器在一个时钟脉冲间隔内改变状态。这样，更新目标

上为逻辑部件操作的相位 3

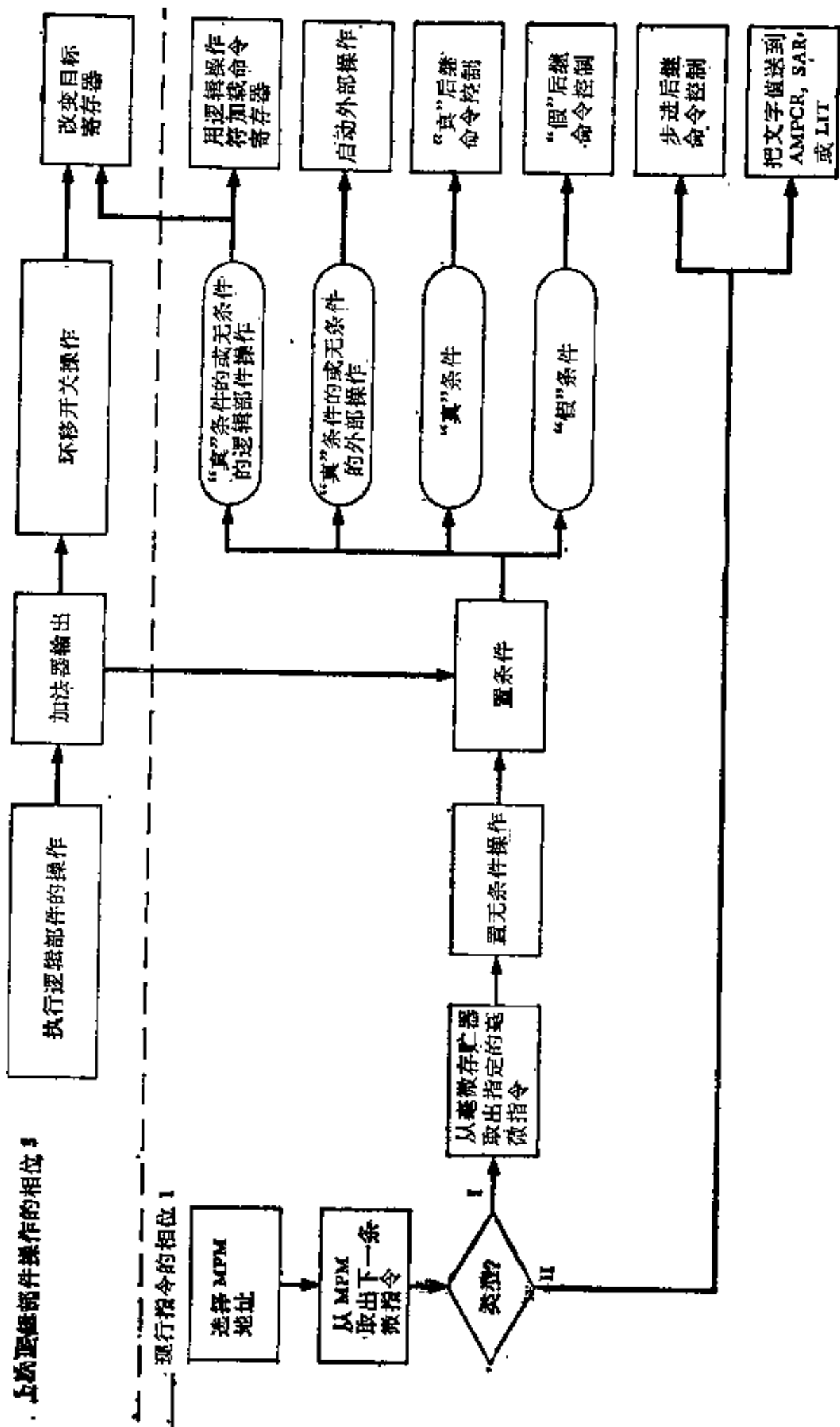


图 10-2 机器操作的同步

寄存器就与加载用于一次新的逻辑部件操作的命令寄存器同时进行。所以,只当启动了一次新的逻辑部件操作时,才能完成目标寄存器的更新。

3. 当一条有逻辑操作的 I 型指令的后面是一条 II 型指令或者是一条 I 型指令(其逻辑部件操作是有条件的,并且条件是不成立的)时,就产生相位 2。在相位 2 周期内,逻辑部件的命令可继续用来测试下一条 I 型指令中相应的条件。

4. 由一条 I 型指令更新的一个寄存器,可以用于紧接着它的下一条 I 型指令。例如,微程序段

$$\begin{aligned}0 &\longrightarrow A1 \\ A1 + 1 &\longrightarrow A1 \\ A1 + 1 &\longrightarrow \text{MIR} \\ \text{MW1, IF SAI}\end{aligned}$$

把数值 2 写入 S 存贮器。

由于定时和操作顺序方面的问题,使得微程序很容易产生错误结果。当若干条 I 型和 II 型指令混用时,尤其是这样。下一节将给出几个例子,用以指出由于疏忽而可能产生的错误。

10-2-3 定时的例子

这里给出三个小型微程序定时的例子。目的是要说明,在执行一段指令序列期间,会产生各种各样的事件。

第一个例子是重复上一节的例子,它由四个无条件连续执行的 I 型指令组成。即:

$$\begin{aligned}0 &\longrightarrow A1 \\ A1 + 1 &\longrightarrow A1 \\ A1 + 1 &\longrightarrow \text{MIR} \\ \text{MW1, IF SAI}\end{aligned}$$

这段微程序的定时图示于图 10-3。在时钟 1,取出指令 1。在时钟 2,把逻辑部件操作“ $0 + B_{A1}$ ”装入命令寄存器(对于指令 1),并且取出指令 2。在时钟 3,对指令 1,寄存器 A1 清除为“0”;对指令

2, 把逻辑部件操作 “ $A1 + B_{001}$ ” 装入命令寄存器, 并取出指令 3. 在时钟 4, 对指令 2, 寄存器 $A1$ 更新为 “1”; 对指令 3, 把逻辑部件操作 “ $A1 + B_{001}$ ” 装入命令寄存器, 并取出指令 4. 在时钟 5, 对指令 3, MIR 更新为 “2”; 把逻辑部件操作 “ $0 + B_{000}$ ” 装入命令寄存器, 并启动存贮器的写操作. 指令 4 中的所有操作都是无条件的, 因而指令 4 具有相位 3. 指令 4 中的 IF SAI 子句用来清除 SAI 指示器.

时 钟	1	2	3	4	5	6
1 $0 \rightarrow A1$	相位 1 相位 3					
2 $A1 + 1 \rightarrow A1$		相位 1 相位 3				
3 $A1 + 1 \rightarrow MIR$			相位 1 相位 3			
4 $MW1, IF SAI$				相位 1 相位 3		

图 10-3 定时的例子

第二个例子给出同时使用 I 型和 II 型指令的情况, 其组成如下:

$A1 \ L \longrightarrow A1$
 $COMP \ 8 \longrightarrow SAR$
 $A2 \ L \longrightarrow A2$
 $A1 \ R \longrightarrow A1$
 $8 \longrightarrow SAR$
 $A2 \ R \longrightarrow A2$

(接着是一条有逻辑部件操作的 I 型指令)

这段微程序定时图示于图 10-4. 在时钟 1, 取出指令 1. 在时钟 2, 把逻辑部件操作 “ $A1 + B_{000}$ ” 和左移操作装入命令寄存器, 并取出指令 2. 在时钟 2 和时钟 3 之间, 执行加法器操作和左移操作. 而左移次数是根据 SAR 中的剩余值进行的. 然而, 在时钟 3, 并没有逻辑部件操作被装入命令寄存器. 因此, 指令 1 的相位 3 就被改变成保持相位 2. 在时钟 3, 把 $COMP \ 8$ 送入 SAR , 并且取出指令 3. 在时钟 3 和 4 之间, 继续使用指令 1 的逻辑部件命令. 在时钟 4, 对于指令 3, 把逻辑部件操作 “ $A2 + B_{000} \ L$ ” 装入命令寄存

时 钟	1	2	3	4	5	6	7	8
1 A1L→A1	相位 1 相位 2 相位 3							
2 COMP 8→SAR	相位 1							
3 A2 L→A2			相位 1 相位 3					
4 A1 R→A1				相位 1 相位 2 相位 3				
5 8→SAR					相位 1			
6 A2 R→A2						相位 1 相位 3		
7 (逻辑操作的 I 型指令)							相位 1 相位 2 或 3	

图 10-4 定时的例子

器,并在环移开关的输出(亦即左移操作的结果)更新了寄存器 A1 的内容时,就完成了指令 1 的相位 3 周期。同样,在时钟 4 也取出指令 4。在时钟 4 和 5 之间,根据 SAR 中的 COMP 来执行指令 3 中的加法器操作和环移开关操作。在时钟 5,完成指令 3,同时对于指令 4,把逻辑部件操作“ $A1 + B_{000}$ ”装入命令寄存器。在时钟 5 还取出指令 5。在时钟 5 和 6 之间,根据 SAR 中的 COMP 8 执行指令 4 中的加法器操作和右移操作(这显然不是所需要的)。然而,在时钟 6,由于指令 5 是 II 型指令,所以没有逻辑部件操作被装入命令寄存器。于是,指令 4 的相位 3 周期就被转换成保持相位 2。在时钟 6 完成的两个操作是: 将 8 送到 SAR,取出指令 6。在时钟 6 和 7 之间,再次执行指令 4 的加法器操作和右移操作;这时 SAR 就含有正确的值 8。在时钟 7,把指令 6 的逻辑部件操作“ $A2 + B_{000}$ ”装入命令寄存器,并完成指令 4 的相位 3 周期;A1 右移 8 位。同样,在时钟 7 还取出指令 7。最后,当把指令 7 的逻辑部件操作装入命令寄存器时,就完成了指令 6 的替换操作。

最后一个例子给出联合使用无条件的和有条件的 I 型指令情况,其内容如下:

$A2 + B \longrightarrow A3$

IF MST THEN $A3 + 1 \longrightarrow A3$, SKIP ELSE STEP

$A3 + B_{111} + 1 \longrightarrow A3$

(紧接着一条具有逻辑操作的 I 型指令)

其中 MST 测试为“假”。这段微程序的定时图示于图 10-5 中。在

时钟 1, 取出指令 1. 在时钟 2, 把逻辑部件操作 “ $A2 + B$ ” 送到命令寄存器, 并取出指令 2. 在时钟 2 和时钟 3 之间, 执行加法器操作 $A2 + B$, 并且, 根据加法器的结果, 可知指令 2 中的 MST 条件测试为“假”. 因此, 不执行指令 2 所指定的逻辑部件操作, 并且把指令 1 的相位 3 周期转换成保持相位 2. 结果是, 指令 2 变成了单相指令. 在时钟 3, 取出指令 3. 在时钟 3 和时钟 4 之间, 继续使用指令 1 的逻辑部件操作 $A2 + B$. 在时钟 4, 把逻辑部件操作 $A3 + B_{111} + 1$ 送到命令寄存器, 并完成指令 1 的相位 3 周期, 亦即用 $A2 + B$ 取代 $A3$. 在时钟 4, 还取出指令 4. 最后, 由于指令含有既定的逻辑部件操作, 所以, 指令 3 就在时钟 5 完成了.

时	钟	1	2	3	4	5	6
1	$A2 + B \rightarrow A3$	相位 1 相位 2 相位 3					
2	IF MST THEN $A3 + 1 \rightarrow A3$ SKIP ELSE STEP	相位 1					
3	$A3 + B_{111} + 1 \rightarrow A3$			相位 1 相位 3			
4	(逻辑操作的 I 型指令)				相位 1 相位 2 或 3		

图 10-5 定时的例子

把尚未完成的相位 3 转换成保持相位 2 的处理过程是动态决定的, 并且, 很容易产生连续的几个相位 2 周期, 如下例所示(其中 AOV 的测试为“假”):

- $A1 + LIT\ R \rightarrow B$ (1)
- 15 LIT, 4 \rightarrow SAR (2)
- IF AOV THEN $B_{0TT} \rightarrow MIR$, SKIP ELSE STEP (3)
- $B \rightarrow MIR$ (4)
- MW1, IF SAI (5)

在这种情况下, 由于指令(2)和(3)都没有相位 3 周期, 所以, 指令(1)的保持相位 2 是为指令(2)和(3)产生的.

10-3 微程序设计的一些特殊问题

微程序设计的过程除了已经指出的与常规计算机的程序设计

的某些相似之处外，还有另外一些相似之处。一旦编写了若干微程序后，则所面临的问题就不在于去编写更多的语句，而在于如何运用微程序设计技术来进行系统设计。朝着这个目标，这里将给出五个“特殊问题”，在考查和实现这些问题时都要求进行创造性的微程序设计，并且也要对主题内容有详尽的理解。

10-3-1 动态地址变换

动态地址变换是一种虚拟存贮器技术，它把虚拟存贮器地址变换成实 S 存贮器地址。动态地址变换可以通过微程序设计来实现或通过一种实现 DAT (dynamic address translation) 功能的硬件来实现。动态地址变换的概貌示于图 10-6。

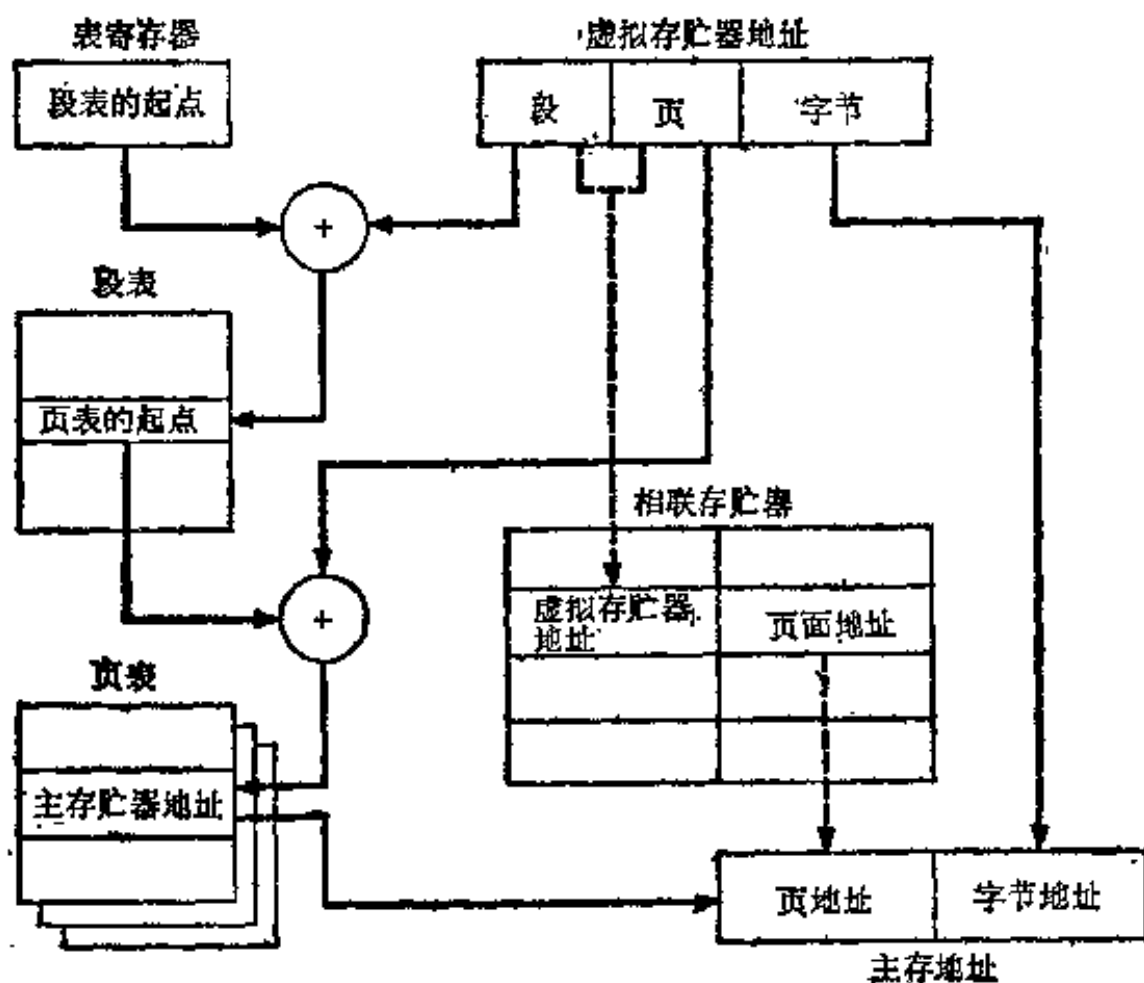
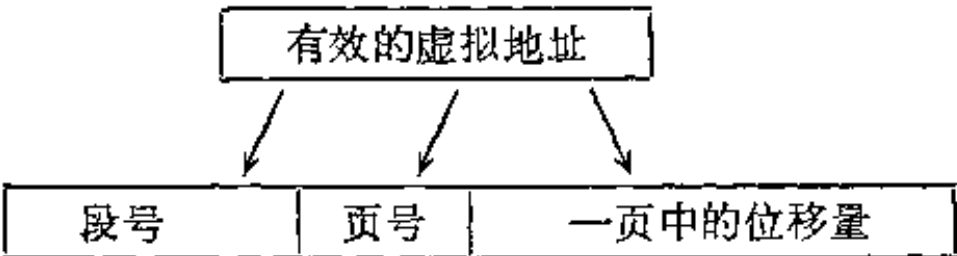


图 10-6 动态地址变换

在虚拟存贮器系统中，通过把虚、实存贮器都分成页面的方式来实现对实存贮器的管理。通常所使用的页面为每页 4096 个字节。作为存贮管理技术的一个方面，将实现页面在实存贮器和直

接存取存贮器之间的传送，并且仅仅是那些在程序执行过程中为支持某一段程序的正常执行所必须的页面才进入主存贮器。通过动态地址变换过程，处理机中形成的每一个有效虚地址将都变换成实存贮器地址，而在执行这种变换过程时使用段表和页面表。每个有效的虚地址都分成段号、页号和位移量三部分，其形式如下：

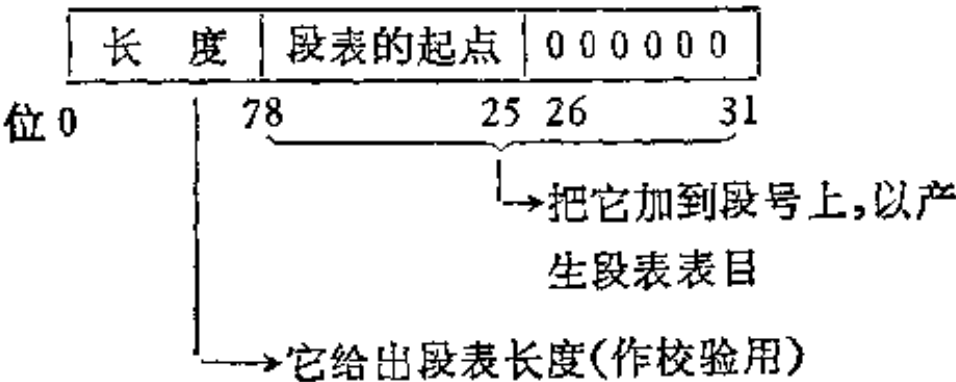


段号作为检索段表的索引，页号作为检索页表的索引。它们典型的长度为：

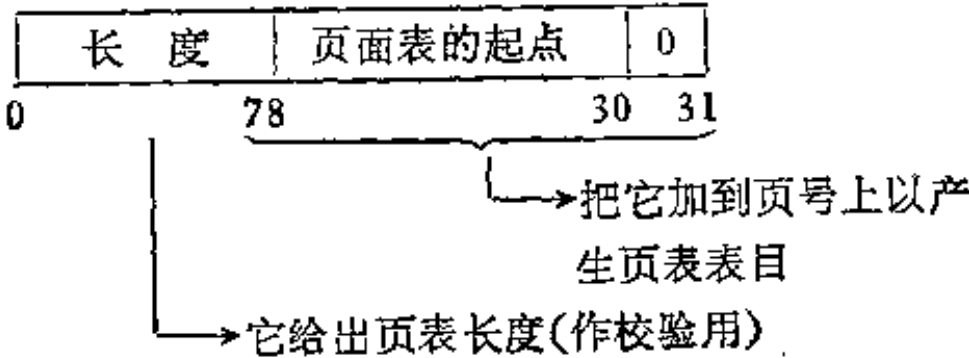
- 段号——4 位
- 页号——8 位
- 位移量——12 位

12 位的位移量可指出一页中的 4096 个字节，因此在一页内无需变换地址——只需要变换页地址。

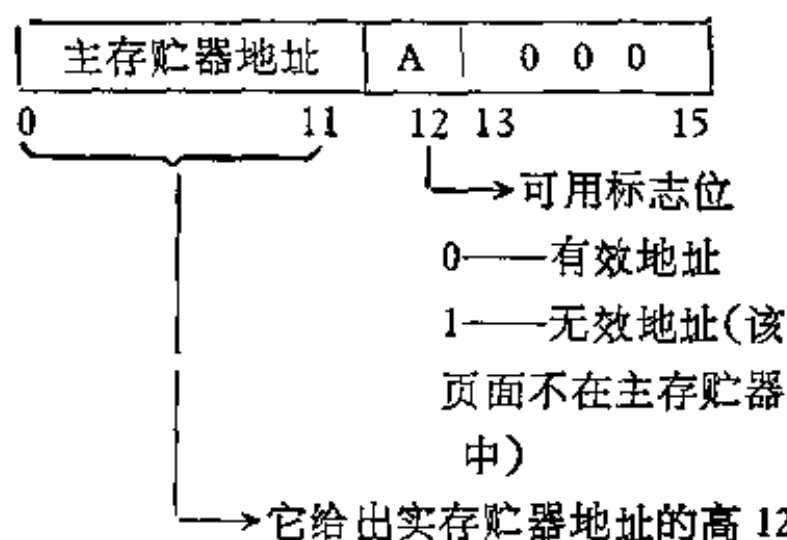
表寄存器指向段表，并具有如下格式：



段表表目指出页表，并且有如下格式：



页表表目给出实存贮器地址,并具有如下格式:



在动态地址变换中,要检测可用标志位,若该位指示的是无效地址(它表示所要的页不在实存贮器),则产生程序中断。通过中断,操作系统把被访问的页调入实存贮器,并把该页的主存贮器地址置入相应的页表表目中。这时,通过动态地址变换,被访问页的实存贮器地址就取代了虚地址中的段号和页号(如图 10-6 所示),从而可以继续指行指令。

动态地址变换需要耗费时间。因此,利用一组保存有最近要使用的虚拟存贮器地址和对应页面地址的相联存贮器,可以提高变换过程的速度。

当然,问题在于如何去设计、实现和调试用以进行动态地址变换的微程序。关于这方面的补充资料请参见脚注中的参考文献¹⁾。

10-3-2 表操作

一个很有意义的计算机功能是根据一个关键值 (Key value)

1) 有关动态地址变换的补充资料,可参看下列参考文献:

P. Freeman, *Software System Principles: A Survey*, Science Research Associates, Chicago, 1975;

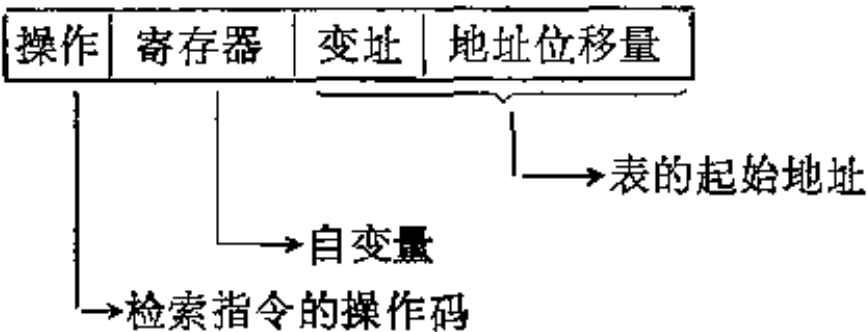
H. Katzan, *Operating Systems: A Pragmatic Approach*, Van Nostrand Reinhold Company, New York, 1973;

S. E. Madnick and J. J. Donovan, *Operating Systems*, McGraw-Hill, New York, 1974.

去检索某个表，常见的例子是形如 (n_i, v_i) 的符号表，其中 n 代表名字， v 代表值。根据自变量名 k 来检索这个表，直到 $k = n_i$ 。在这个过程中，值 v_i 作为检索对象。在语言处理程序中，这种表可以保存符号名和符号地址；在资源应用中，这种表可以包括零件名称和现有的数量。问题在于要构造一条简单的计算机指令来检索这类表格。

10-3-2-1 计算机指令的格式

计算机指令将取如下格式：



这条指令执行如下操作：

1. 计算有效地址（即（变址）+位移量）并且从该地址开始检索。
2. 自变量已事先放在指定的操作数寄存器 R_i 。
3. 如果查出该表中有一个表目的名字与自变量相符合，则检索出相应名字的值并把它送入寄存器 R_{i+1} 中。
4. 检索成功后，就跳过下一条计算机指令。
5. 如果在表的表目中找不到与自变量相符的名字，则检索不成功。在这种情况下，就顺序执行下一条计算机指令。

表目中的信息取决于存贮器的分配方法。下面讨论两种分配方法：连续分配法和链接分配法。

名字	值
名字	值
.	.
.	.
.	.
空白	0

末项

10-3-2-2 连续分配

采用连续分配时，表有如左示的矩阵形式。

信息按行存贮,如右图.

对每个名字和每个值都赋予一个字长标志.

通过执行上述指令来检索这个表,一直检索到相应的名字和值或检索到表的末端.

名字,
值 ₁
名字,
值,
空白
0

10-3-2-3 链接分配

采用链接分配时,每个表目的形式如右下图.

其中“指示器”指出下个表目的地址. 最后一个节点(即表目)的指示器字段是零. 包含在一个节点中的各个量是连续存贮的,并且每个量都有一个字长标志.

通过执行上述指令来检索这种链接表,一直检索到相应的名字和值或检索到最后一个节点.

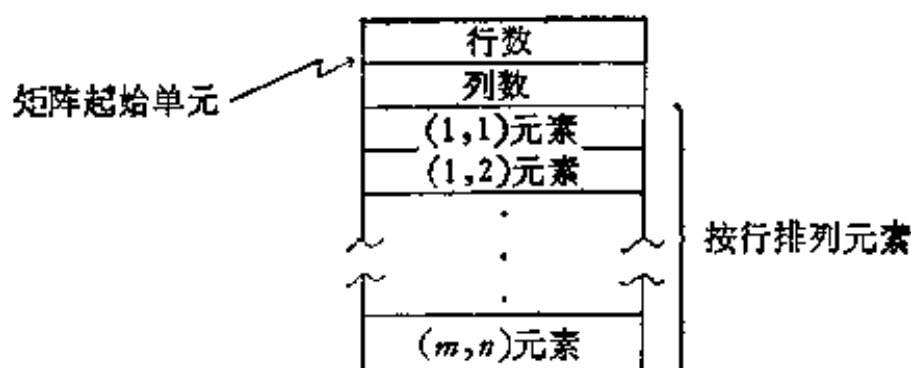
名字
值
指示器

10-3-3 矩阵操作

BASIC 语言¹⁾中包含有如下形式的矩阵操作

$$\text{MAT } C = A \oplus B$$

式中 \oplus 是下列的矩阵运算之一: $+$ (逐个元素加法), $-$ (逐个元素减法), $*$ (矩阵乘法). 且假定矩阵按如下方式存贮.



通过一组微程序来完成每一种矩阵运算. 并且,微程序能控制存放在 S 存贮器中的任何一个矩阵.

1) 关于 BASIC 语言的一本好的参考书是: J. G. Kemeny and T. C. Kurtz, *BASIC Programming*, Wiley, New York, 1971.

另外,它也要求设计一条 S 指令,利用它可以访问微程序,并且在每次矩阵运算中都需要三个操作数.

10-3-4 向量运算

APL 语言包括丰富的向量运算¹⁾. 其中包括:

1. 逐个元素运算. 例如 $+$, $-$, $*$, $/$, \wedge , \vee , \sim , $>$, \geq , $<$, \leq , $=$, \neq , 等等.

2. 凝缩 (Reduction), 例如

$$\oplus/V$$

其中 \oplus 是标量运算, 而 V 是向量, 表达式

$$+/A$$

定义为 $A[1] + A[2] + A[3] + \cdots + A[n]$.

3. 反序 (Reversal), 其形式如下:

$$\phi A$$

它将向量 A 中的诸元素反序排列.

4. 旋转 (Rotation), 其形式如下:

$$n\phi A$$

若 n 是正数, 则把向量 A 循环左移 $|n|$ 个元素位置; 若 n 是负数, 则把向量 A 循环右移 $|n|$ 个元素位置.

5. 抽取 (Take), 其形式如下

$$n \uparrow A$$

若 n 是正数, 则选取 A 的前 $|n|$ 个元素; 若 n 是负数, 则选取 A 的后 $|n|$ 个元素.

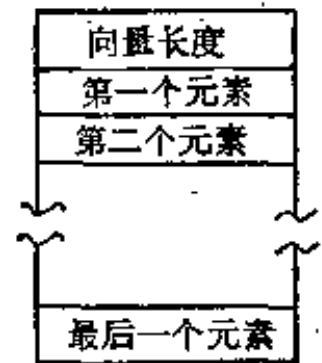
6. 消去 (Drop), 其形式如下:

$$n \downarrow A$$

若 n 是正数, 则消去 A 的前 $|n|$ 个元素; 若 n 是负数, 则消去 A 的后 $|n|$ 个元素.

1) 有关 APL 语言的一本参考书是: H. Katzan, *APL Users Guide*, Van Nostrand Reinhold Company, New York, 1971.

假定向量按右图形式存贮,则问题是要进行微程序设计,用它来实现向量运算。其中,很重要的是要设计若干 S 指令。利用这些指令来访问微程序。



10-3-5 图灵机器

这个特殊问题是要研制图灵机的仿真程序¹⁾。信息记录带和图灵机程序应当存贮在 S 存贮器中,其中每一条带的方块都可以按一个字来存贮,并且可以使用任何一个字母。但在下例中只使用 B (空白), 0 (零) 和 1 (壹)。

10-3-5-1 图灵机简述

图灵机的动作取决于机器的内部状态以及读-写头扫描到的符号。为简单起见,假定机器可完成下列功能之一:

1. 把一个新字符写到读-写头下面的带方块中。
2. 读-写头向右移动一个方块。
3. 读-写头向左移动一个方块。
4. 停止机器操作。

图灵机的操作受一组规则控制,这组规则指定每一步计算所要做的动作。为简单起见,假定操作是由包括下述信息的四元组 (quadruple) 来描述的:

1. 现行状态。
2. 扫描中的符号。
3. 要做的动作。
4. 下一个状态。

允许有下面的四元组

$$1. s_i A_j A_k s_m$$

1) 关于图灵机及相应的算法的一本好的参考书是: M. Davis, *Computability and Unsolvability*, McGraw-Hill, New York, 1958.

2. $s_i A_j R s_m$

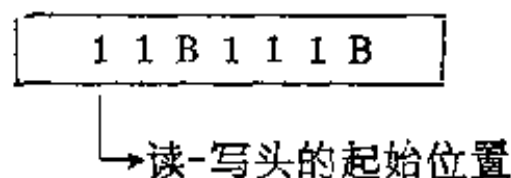
3. $s_i A_j L s_m$

其中, s_i 是现行状态, A_j 是扫描中的符号, A_k 是下一个要写的符号, 而 s_m 是下一个状态. R 表示右移, 而 L 表示左移. 四元组 1 规定用 A_k 替换 A_j 并进入状态 s_m . 四元组 2 规定向右移的一个带块并进入状态 s_m . 四元组 3 规定向左移动一个带块并进入状态 s_m .

图灵机的一个算法(亦即一个图灵机程序)是一群四元组, 其中每两个四元组的前两个符号都是不相同的(它们是现行状态和扫描中的符号). 于是, 现行状态和读-写头下的符号就决定了所要发生的动作, 因而隐含地选择了一个唯一的四元组. 利用四元组的前两个符号来选择下一个操作, 机器就通过从一个状态过渡到另一个状态的方式进行操作. 当现行状态和扫描中的符号同任何一个四元组都不相符合时, 机器就停止操作.

10-3-5-2 算法实例

一个算法实例是, 把两个数加起来, 这两个数都用信息存贮器上的一串 1 表示. 数 n 被表示成 $n + 1$ 个 1, 其后接着一个空白符号. 被相加的两个数用一个空白的符号隔开, 如下述:



把两个数加起来的算法将按下述过程操作:

1. 沿着信息存贮带向右移动, 直到到达一个空白带块为止.
2. 在空白字符位置上写一个 1.
3. 沿着信息存贮带向右移动, 一直到达另一个空白带块为止.
4. 向左移动一个带块.
5. 写一个空白字符.
6. 向左移动一个带块.
7. 写一下空白字符.

8. 停止.

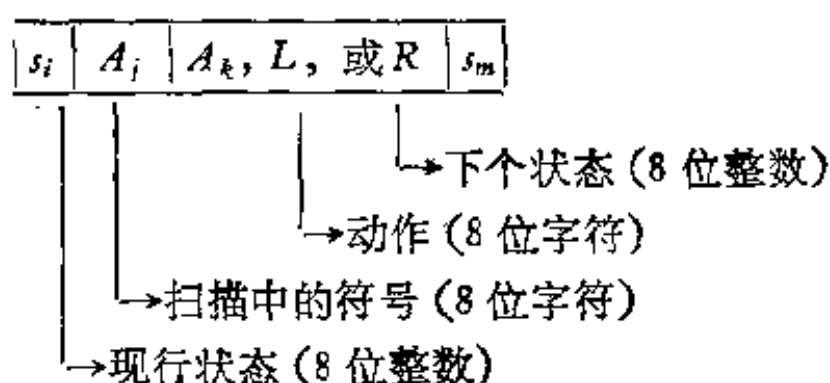
关于这个算法的图灵机的问题就是:

现行状态	被扫描的符号	动作	下个状态
s_1	1	R	s_1
s_1	B	1	s_2
s_2	1	R	s_2
s_2	B	L	s_3
s_3	1	B	s_3
s_3	B	L	s_4
s_4	1	B	s_4

当该机器在 s_4 状态时,它用一个在读-写头下的空白符实现停机.

10-3-5-3 实现

有一种实现方法是把每个四元组都按下述字指令的形式存贮:



这样,仿真程序就需要有一个取指令子程序和一个译码子程序.可是,机器可能进入到编号小的状态,这样,从概念上看,现行状态和扫描中的符号的相配将从一系列四元组的起点开始.

参 考 文 献

- [1] Bingham, H. W., R. L. Davis, U. Faber, D. A. Fisher, J. D. McGonagle, E. W. Reigel, and S. Zucker, *Microprogramming Manual for Interpreter Based Systems*, Burroughs Corporation, Paoli, Pa. 1970, TR 70—8.
- [2] Casaglia, G. F., Nanoprogramming vs. Microprogramming, *Computer*, vol. 9, no. 1, pp. 54—58 (January 1976).
- [3] Chu, Y., *Computer Organization and Microprogramming*, Prentice-Hall, Englewood Cliffs, N. J., 1972.
- [4] Hussen, S., *Microprogramming: Principles and Practices*, Prentice-Hall, Englewood Cliffs, N. J., 1970.
- [5] Jones, L. H., Microinstruction Sequencing and Structured Microprogramming, *Proceedings of the Seventh Annual Workshop on Microprogramming (MICRO-7)*, 1974, pp. 277—289. (Available from the Association for Computing Machinery, located in New York City.)
- [6] Katzan, H., *Computer Organization and the System/370*, Van Nostrand Reinhold Company, New York, 1971.
- [7] Reigel, E. W., U. Faber, and D. A. Fisher, The Interpreter—A Microprogrammable Building Block System, AFIPS vol. 40. *Proceedings of the 1972 Spring Joint Computer Conference*, pp. 705—723.
- [8] Rosin, R. F., Contemporary Concepts of Microprogramming and Emulation *Computing Surveys*, vol. 1, no. 4, pp. 197—212 (December 1969).
- [9] Wilkes, M. V., The Growth of Interest in Microprogramming—A Literature Survey, *Computing Surveys*, vol. 1, no. 3, pp. 139—145 (September 1969).

关键字词汇表

- A1 A1 寄存器 用作X选择的操作数或目标操作符。
- A2 A2 寄存器 用作X选择的操作数或目标操作符。
- A3 A3 寄存器 用作X选择的操作数或目标操作符。
- AAD 与-加逻辑操作符 $X \text{ AAD } Y$ 等价于 $X + (X \wedge Y)$ 。
- ABT 全1条件 (All-bits-true)。它是由前一条 I 型指令在相位 3 期间执行逻辑操作所建立的动态条件。
- AMPCR 微程序交替计数寄存器 在 II 型指令的文字赋值语句执行期间作为 X 选择的操作数，或者作为从环移开关来的目标操作符。从环移开关来的最低 12 个有效位被送到 AM PCR。
- AND 与逻辑操作符 $X \text{ AND } Y$ 等价于 $X \wedge Y$ 。
- AOV 加法器溢出条件 这是在加法器运算结果产生高位溢出时，由前一条 I 型指令在相位 3 期间执行逻辑操作符所建立的动态条件。
- B B 寄存器 用作 Y 选择的操作数或从环移开关来的目标操作符。把 B 用作 Y 选择操作数就等价于 B_{TTT} (在其他命令中，B 寄存器也用作目标寄存器)。
- BAD B 寄存器的内容来源于加法器 用作“把加法器输出送到 B 寄存器”的目标操作符。
- BBA B 寄存器的内容来源于加法器同环移开关的输出之“或” 用作目标操作符。
- BBE B 寄存器的内容来源于外部数据总线同环移开关的输出之“或” 用作目标操作符。
- BBI B 寄存器的内容来源于 MIR 前次内容同环移开关的输出之“或” 用作目标操作符。

- BEX B寄存器来源于外部数据总线 用作目标操作符。
- BMI B寄存器的内容来源于 MIR 前次的内容 用作目标操作符。
- BR1 基址寄存器 1 作为由环移开关输出的目标操作符；取环移开关的次低有效字节。用于外部设备的读和写访问。
- BR2 基址寄存器 2 作为由环移开关输出的目标操作符；取环移开关的次低有效字节。用于外部设备的读和写访问。
- C 循环右移操作 它同逻辑部件一起指定，并在环移开关中进行。
- CALL 选择后继命令 转移到 AMPCR + 1 的内容执行指令，并把现行 MPCR 值置入 AMPCR，以用于此后的返回连接。
- COMMENT 在程序中用于注解行的关键字 (COMMENT 用在 TRANSLANG 中)。
- COMP 用于文字赋值语句的求补算符。
- COV 计数器溢出条件 当 CTR 从全“1”增到“0”时，就置该条件。
- CSAR 目标操作符 用于把 SAR 的内容求补。
- CTR 计数器 用作 X 选择或 Y 选择的操作数；或者用作目标操作符，它取环移开关输出的 8 个低有效位。
- ELSE 分隔符 用来指示当条件为“假”时的选择。
- END 标志一个微程序结束的关键字
- EQV “等价”逻辑算符 $X \text{ EQV } Y$ 等价于 $(X \wedge Y) \vee (\bar{X} \wedge \bar{Y})$ 。
- EXEC 选择后继命令 用于执行 (AMPCR) + 1 处的指令，但继续保持通常的指令执行顺序。
- F 取 B 寄存器的反向输出 用于部分 Y 选择的操作数。
- IF 指示一个语句中条件部分的关键字。
- IMP “蕴含”逻辑算符 $X \text{ IMP } Y$ 等价于 $\bar{X} \vee \bar{Y}$ 。
- INC 目标操作符 它把 CTR 的前次内容增量，并可以由此建立 COV 条件。
- JUMP 选择后继命令 转移到 AMPCR + 1 的内容执行指令，并

把它变成 MPCR 的内容。

- L 左移操作符 指出加法器的输出将在环移开关中执行逻辑左移操作,并在其右边补 0。
- LC1 逻辑条件 1 利用 SET LC1 命令建立这个条件,并通过测试这个条件来清除它。
- LC2 逻辑条件 2 利用 SET LC2 命令建立这个条件,并通过测试这个条件来清除它。
- LC3 逻辑条件 3 利用 SET LC3 命令建立这个条件,并通过测试这个条件来清除它。
- LCTR 目标操作符 将 LIT 内容的反码送到 CTR,并清除 COV 条件。
- LIT 文字寄存器 用作 X 选择或 Y 选择操作数的最低有效字节,并用作 LMAR 和 LCTR 命令中的源操作数。通过文字赋值 I 型指令给它赋值。
- LMAR 目标操作符 把 LIT 的内容送入 MAR。
- LST 最低有效位的条件 它反映出在前一条 I 型指令的相位 3 期间,加法器输出的最右一位的状态。若该位为“1”,则建立 LST 条件;反之,不建立该条件。
- MAR 目标操作符 它把环移开关输出的最低有效字节送到存储器地址寄存器 (MAR)。
- MAR1 目标操作符 把环移开关输出的次低有效字节送到 BR1 寄存器,而环移开关输出的最低有效字节则送到 MAR 寄存器。
- MAR2 目标操作符 把环移开关输出的次低有效字节送到 BR2 寄存器,而环移开关输出的最低有效字节则送到 MAR 寄存器。
- MIR 目标操作符 把环移开关的输出送到存储器信息寄存器 (MIR)。
- MR1 存储器读命令 1 对于由 (BR1, MAR) 所指定的 S 存储器单元启动一次读操作。

- MR2 存储器读命令 2 对于由 $\langle \text{BR2}, \text{MAR} \rangle$ 所指定的 S 存储器单元启动一次读操作。
- MST 最高有效位的条件 它反映出在前一条 I 型指令的相位 3 期间,加法器输出的最高有效位的状态。若该位为“1”,则建立 MST 条件;否则不建立 MST 条件。
- MW1 存储器写命令 1 对由 $\langle \text{BR1}, \text{MAR} \rangle$ 所指定的 S 存储器单元启动一次写操作(写入信息来自 MIR)。
- MW2 存储器写命令 2 对由 $\langle \text{BR2}, \text{MAR} \rangle$ 所指定的 S 存储器单元启动一次写操作(写入信息来自 MIR)。
- NAN “与-非”逻辑算符 $X \text{ NAN } Y$ 等价于 $\bar{X} \vee \bar{Y}$, 也等价于 $\overline{(X \wedge Y)}$ 。
- NIM “非蕴含”逻辑算符 $X \text{ NIM } Y$ 等价于 $X \wedge \bar{Y}$ 。
- NOR “或-非”逻辑算符 $X \text{ NOR } Y$ 等价于 $\bar{X} \wedge \bar{Y}$, 也等价于 $\overline{(X \vee Y)}$ 。
- NOT “求反”逻辑算符 $\text{NOT } Y$ 等价于 \bar{Y} 。
- NRI “非逆蕴含” (Not reverse imply) 逻辑算符 $X \text{ NRI } Y$ 等价于 $\bar{X} \vee Y$ 。
- OAD “或-加”逻辑算符 $X \text{ OAD } Y$ 等价于 $X + (X \vee Y)$ 。
- OR “或”逻辑算符 $X \text{ OR } Y$ 等价于 $X \vee Y$ 。
- R “右移”操作符 指出加法器的输出在环移开关中执行右移操作,并在其左边补 0。
- RDC 读操作完成条件 当输入到 B 寄存器的外部数据已准备好时,就建立这个条件,而通过测试该条件来清除它。
- RETN 选择后继命令 转移到 $\text{AMPCR} + 2$ 的内容,且把它变成 MPCR 的内容(可以用于从子程序的返回连接)。
- RIM “逆蕴含” (Reverse) 逻辑算符 $X \text{ RIM } Y$ 等价于 $X \vee \bar{Y}$ 。
- SAI 开关接收信息的条件 当处理机完成写操作后就建立该条件。这样,操作中所使用的寄存器就可以改变内容。通过测试该条件来清除它。
- SAR 移位次数寄存器 用来规定移位操作的长度(即次数)。

SAR 是通过文字赋值装入的；SAR 也可以作为一个目标操作符，在这种情况下，它接收环移开关输出的最低有效字节。

SAVE 选择后继命令 把 MPCR 的内容置于 AMPCR，并把 $(\text{MPCR}) + 1$ 用作下一条指令的地址。

SET 建立条件的命令 建立所指定的条件指示器。

SKIP 选择后继命令 将 $(\text{MPCR}) + 2$ 用作下一条指令地址，以此跳过顺序执行的下一条指令。

SLIT 文字赋值的目标 用一个值装入到 LIT 寄存器，它随后被转送到 SAR。

STEP 选择后继命令 通过把 $(\text{MPCR}) + 1$ 用作下一条指令地址的方式，步进到下一条指令。

T B 寄存器“真值”输出 用于部分 Y 选择的操作数。

THEN 分隔符 用来指示当一个条件语句为“真”时的选择。

WAIT 选择后继命令 通过把 (MPCR) 用作新指令地址的方式来重复现行指令。

WHEN 利用蕴式的 ELSE WAIT 选择来指定一个条件语句的关键字。

XOR “异或”逻辑算符 $X \text{ XOR } Y$ 等价于 $(X \wedge \bar{Y}) \vee (\bar{X} \wedge Y)$ 。