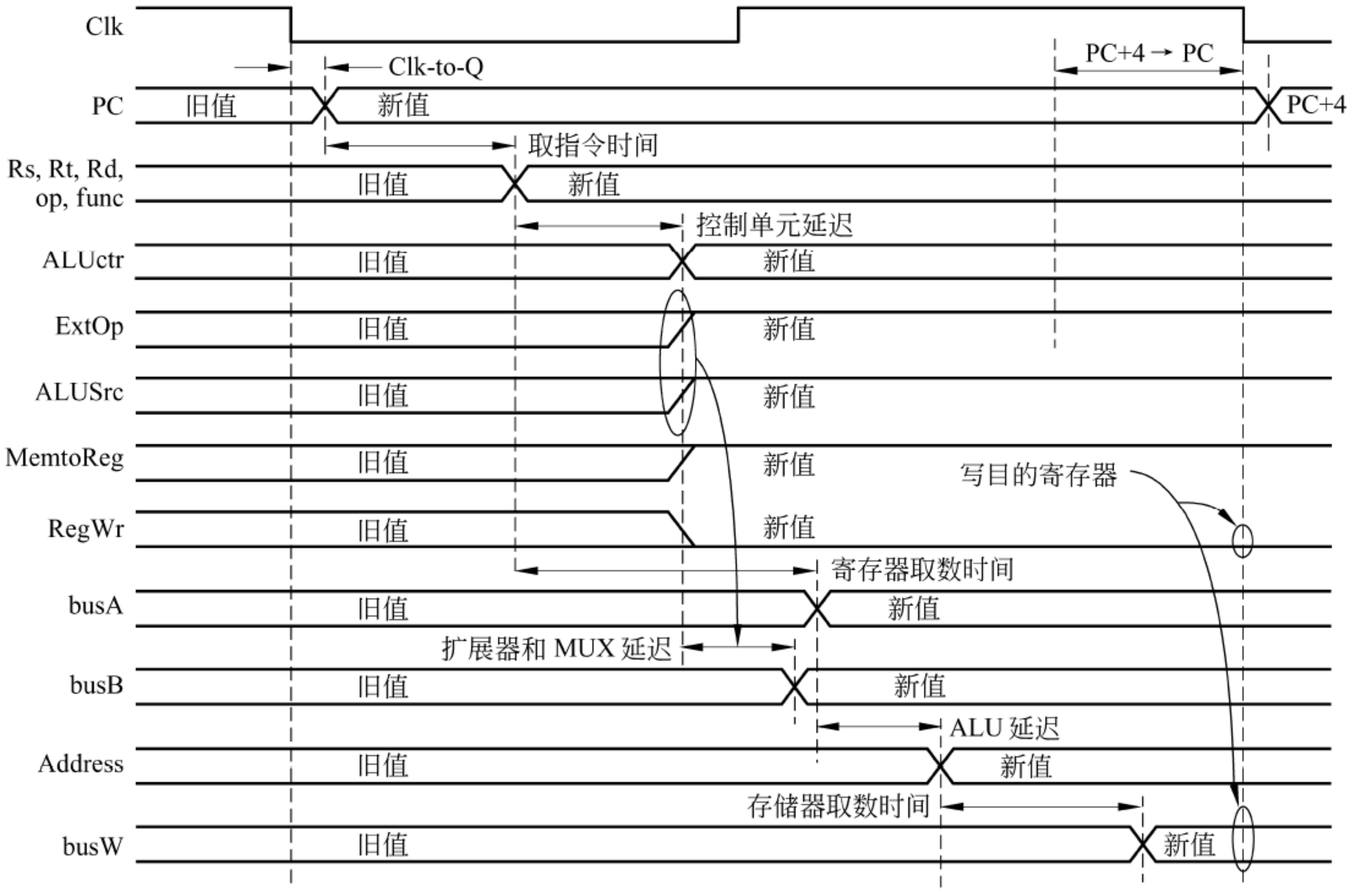


图 5.30　Load 指令的执行定时

5.3　多周期处理器设计

单周期处理器时钟周期取最复杂指令所用指令周期，因而远远大于许多指令实际所需执行时间。受时钟周期宽度的影响，单周期处理器的效率低下、性能极差，实际上，现在很少用单周期方式设计 CPU。本书介绍单周期 CPU 的设计实现，只是为了有助于理解实际的多周期执行和流水线执行两种方式。

*5.3.1　信号竞争问题

多周期处理器的基本思想为：把每条指令的执行分成多个大致相等的阶段，每个阶段在一个时钟周期内完成；各阶段内最多完成一次访存或一次寄存器读写或一次 ALU 操作；各阶段的执行结果在下一个时钟到来时保存到相应存储单元或稳定地保持在组合电路中；时钟周期的宽度以最复杂阶段所用时间为准，通常取一次存储器读写的时间。

在介绍单周期处理器时，存储器被简化为理想情况，即假定每次写操作都由时钟控制，并且在每次时钟到来时，地址、数据和写使能信号都已稳定一段时间。事实上，存储器的实际写操作不是由时钟边沿触发，而是一个组合逻辑电路。其写操作的过程为：当“写使能”信号有效，并且写入数据和地址已稳定，则经过一个写操作时间后，数据被写入。这里，重要的一点是：地址和数据必须在“写使能”信号有效前先稳定在各自的输入端。实际的存储器在单周期数据通路中不能可靠工作，这是因为：不能保证地址和数据能在“写使能”信号有效前稳定，即地址、数据和“写使能”之间存在竞争(race)问题。

竞争问题有时会导致机器意外出错。在多周期处理器中,可通过以下方式来解决竞争问题:首先确认地址和数据在第 n 周期结束时已稳定,然后,使"写使能"信号在一个周期后(即第 $n+1$ 周期)有效,并使地址和数据在"写使能"信号无效前不改变其值。

*5.3.2 指令执行状态分析

多周期处理器中,每条指令分多个阶段执行,每个阶段占一个时钟周期,称为一个状态。因此,一条指令的执行过程由多个状态组成。在指令被译码之前,每条指令所完成的操作是一样的,指令译码后不同的指令有不同的执行过程。

图 5.31 是加了控制信号的多周期数据通路示意图。因为 PC、IR、分支目标地址寄存器和寄存器堆只能在需要时写入新值,其他情况下不能写入,所以它们都需要由"写使能"信号来控制;而寄存器 A 和 B 是临时寄存器,每来一个时钟都可改变它们的值,因而它们无须"写使能"控制信号。

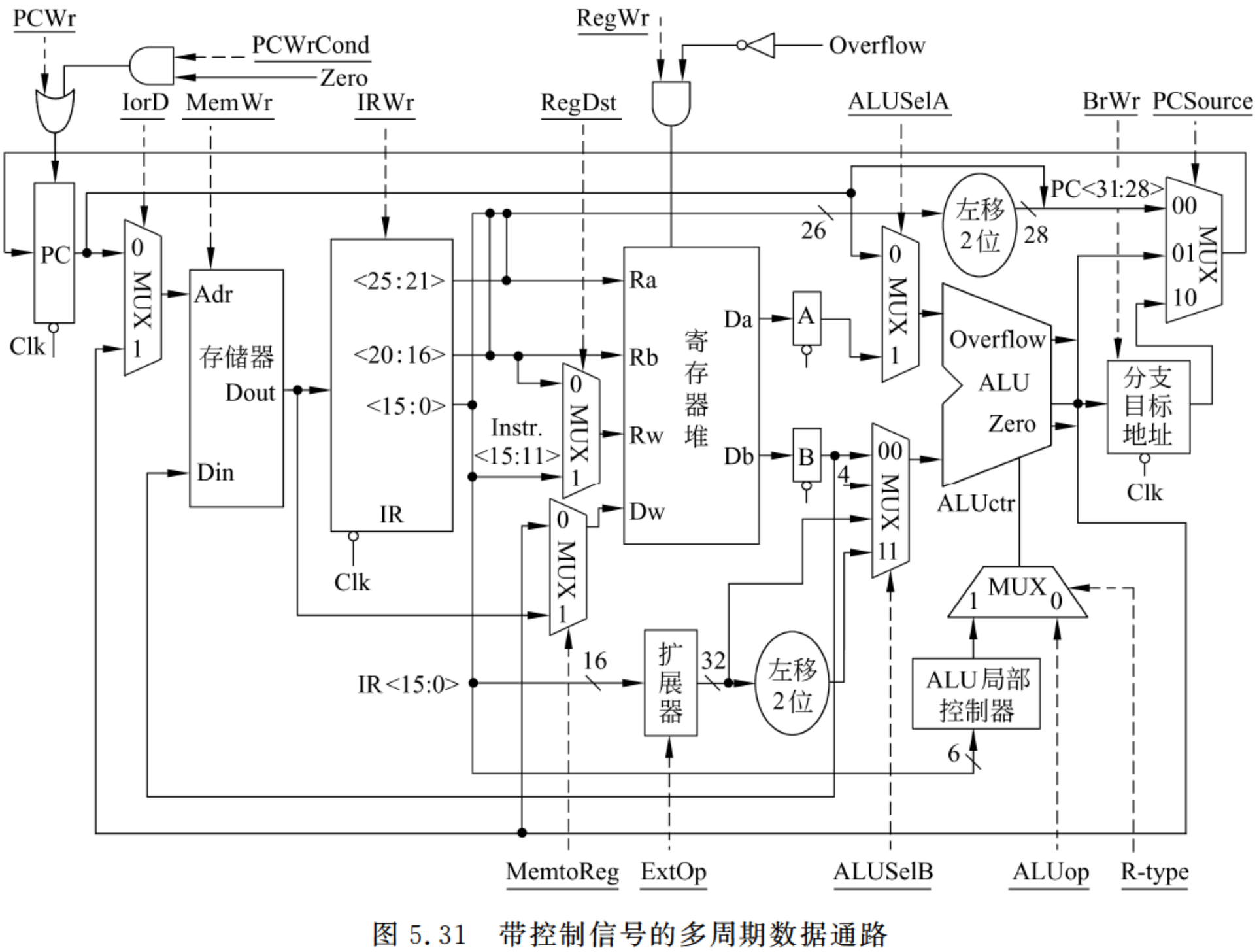


图 5.31 带控制信号的多周期数据通路

PC 的来源可以是 PC+4、分支目标地址和无条件跳转目标地址,由 PCSource 信号控制。分支目标地址保存在专门的分支目标地址寄存器中,无条件跳转目标地址由 PC 高 4 位和指令低 26 位直接拼接后在低位添 00 得到。同单周期数据通路一样,对于需要判断溢出的算术运算类指令,当发生溢出时,禁止写结果到寄存器堆。

1. 取指令、指令译码/取数阶段

取指令、指令译码/取数是所有指令译码之前的公共操作,与指令译码结果没有关系。

1) 取指令状态

取指令阶段的功能是 IR←M[PC],PC←PC+4。因此,需要将 PC 的值作为地址来读

存储器，并将读出指令送 IR 输入端，使得下一个时钟到来时，读出的指令送 IR，同时，PC 将送 ALU 的 A 口，并选择 4 送 ALU 的 B 口，控制 ALU 做不判溢出的加法操作，得到 PC+4，送 PC 输入端，使得下一个时钟到来时，PC+4 送 PC。

该状态名记为 IFetch，控制信号取值为：IorD＝0，ALUSelA＝0，ALUSelB＝01，ALUOp＝addu，PCSource＝01，PCWr＝IRWr＝1，MemWr＝RegWr＝BrWr＝R-type＝0，其余任意。

2）译码/取数状态

译码/取数阶段的功能是 CU(译码)←IR<31:26>，A←R[IR<25:21>]，B←R[IR<20:16>]。该阶段 ALU 是空闲的，所以可以利用 ALU“投机”计算分支目标地址。如果当前指令是分支指令，则可节省一个时钟周期；若不是分支指令，也不会有任何影响。

分支目标地址计算方法为 PC+4+(SignExt(imm16)×4)，因为取指令阶段结束时已经把 PC+4 送 PC 输入端，所以，该阶段只要计算 PC+(SignExt(imm16)×4)。

状态名记为 RFetch/ID，控制信号值为：ExtOp＝1，ALUSelA＝0，ALUSelB＝11，ALUOp＝addu，BrWr＝1，PCWr＝PCWrCond＝IRWr＝MemWr＝RegWr＝R-type＝0，其余任意。

2. R-型指令运算阶段

R-型指令运算执行阶段功能为 R[IR<15:11]>]←A op B，即 A、B 内容分别送 ALU 的 A 口和 B 口，进行相应运算后，写入到寄存器堆中。对于 R-型指令，ALU 操作控制信号 ALUctr 由局部 ALU 控制器根据 func 字段产生，而主控制器生成的 ALUOp 不起作用。考虑到寄存器堆写入时的“写使能”信号和写入数据、地址信号之间的竞争问题，该阶段要用两个状态来完成，第一个状态先送数据、地址，第二个状态再使“写使能”信号有效。

R-型指令执行状态(记为 RExec)的控制信号取值为：ALUSelA＝1，ALUSelB＝00，RegDst＝R-type＝1，RegWr＝PCWr＝PCWrCond＝IRWr＝MemWr＝BrWr＝MemtoReg＝0，其余任意。

结束状态(记为 RFinish)的控制信号取值为：除 RegWr＝1 外，其余同执行状态。

3. I-型指令立即数运算阶段

I-型运算指令的执行阶段功能为 R[IR<20:16>]←A op Ext([IR<15:0>])。算术运算指令(如 addiu)对立即数进行的是符号扩展，而逻辑运算指令(如 ori)进行的是零扩展。

每条 I-型运算指令与上述 R-型指令的运算阶段一样，也分执行和结束两个状态。执行状态中有两个信号(ALUOp 和 ExtOp)的值会随指令不同而不同：算术运算指令的 ExtOp 为 1，逻辑运算指令的 ExtOp 为 0；ALUOp 的取值由指令的运算类型确定，例如，ori 指令的 ALUOp 为 or，andi 指令的 ALUOp 为 and，addiu 指令的 ALUOp 为 addu，addi 指令的 ALUOp 为 add 等。其他信号取值如下：ALUSelA＝1，ALUSelB＝10，MemtoReg＝RegDst＝RegWr＝PCWr＝PCWrCond＝IRWr＝MemWr＝BrWr＝R-type＝0，其余任意。

结束状态控制信号取值除 RegWr＝1 外，其余信号的取值同执行状态。

例如，ori 指令执行状态(记为 oriExec)的控制信号取值为：ALUSelA＝1，ALUSelB＝10，ALUOp＝or，ExtOp＝MemtoReg＝RegDst＝RegWr＝PCWr＝PCWrCond＝IRWr＝MemWr＝BrWr＝R-type＝0，其余任意；ori 指令结束状态(记为 oriFinish)的各控制信号的取值为：RegWr＝1，ALUSelA＝1，ALUSelB＝10，ALUOp＝or，ExtOp＝MemtoReg＝

RegDst=PCWr=PCWrCond=IRWr=MemWr=BrWr=R-type=0,其余任意。

4. lw 指令执行阶段

lw 指令执行阶段功能为 R[IR<20:16>]←M[A+SignExt([IR<15:0>])],由以下3个状态组成:

(1) 访存地址计算状态(记为 MemAdr):ALUSelA=1,ALUSelB=10,ALUOp=addu,ExtOp=IorD=1,RegWr=PCWr=PCWrCond=IRWr=MemWr=BrWr=R-type=0,其余任意。

(2) 存储器取数状态(记为 MemFetch):ExtOp、ALUSelA、ALUSelB、ALUOp、IorD、R-type 和上一个状态一样,以继续保持访存地址信号的稳定;MemtoReg=1,使数据尽早稳定在寄存器堆的 Dw 输入端;RegDst=0,使地址尽早稳定在寄存器堆的 Rw 输入端;RegWr=PCWr=PCWrCond=IRWr=MemWr=BrWr=0,使所有寄存器和存储器不做任何更新;其余任意。

(3) 结果写回寄存器状态(记为 lwFinish):RegWr=1,使数据写入寄存器,其余信号取值同上一个状态,以继续保持寄存器堆的 Dw 和 Rw 输入端稳定不变。

5. sw 指令执行阶段

sw 指令执行阶段功能为 M[A+SignExt([IR<15:0>])]←B,由访存地址计算和存储器存数(记为 swFinish)两个状态组成。第一个状态同 lw 指令第一个状态,第二个状态只要使 MemWr=1,其余控制信号不变,这样保证存储器的写入地址和写入数据在本状态中稳定不变。

6. 分支指令执行阶段

分支指令 beq 执行阶段的功能为 if(A−B=0) then PC←分支目标地址。因此只要一个状态即可,该状态名记为 BrFinish。其控制信号取值为:ALUSelA=1,ALUSelB=00,ALUOp=subu,PCWrCond=1,PCSource=10,RegWr=PCWr=IRWr=MemWr=BrWr=R-type=0,其余任意。

7. 无条件跳转指令执行阶段

跳转指令 jump 执行阶段的功能为 PC←跳转目标地址。因此只要一个状态即可,该状态名记为 jumpFinish。其控制信号取值为:PCSource=00,PCWr=1,RegWr=IRWr=MemWr=BrWr=0,其余任意。

根据上述对每条指令执行过程的分析,得到一个状态转换图。图 5.32 是一个支持 R-型指令、I-型运算类指令 ori、lw/sw、beq 和 jump 指令执行的状态转换示意图,图中每个状态用一个状态号和状态名标识,例如,0:IFetch 表示第 0 状态,执行取指令(IFetch)操作,圆圈中示意性地给出了该状态下部分控制信号相应的取值。

程序在图 5.31 所示的多周期数据通路中执行的过程就是图 5.32 所示的状态转换过程。每来一个时钟,进入下一个状态。从图 5.32 可看出,R-型指令、I-型运算类指令和 sw 指令的 CPI 都为 4;lw 指令的 CPI 最大,其 CPI 为 5;分支指令和跳转指令的 CPI 为 3。如果不在译码/取数阶段“投机”计算分支目标地址,则分支指令的 CPI 为 4。

*5.3.3 硬连线路控制器设计

由于多周期数据通路中每个指令的执行有多个周期,每个周期的控制信号取值不同,所

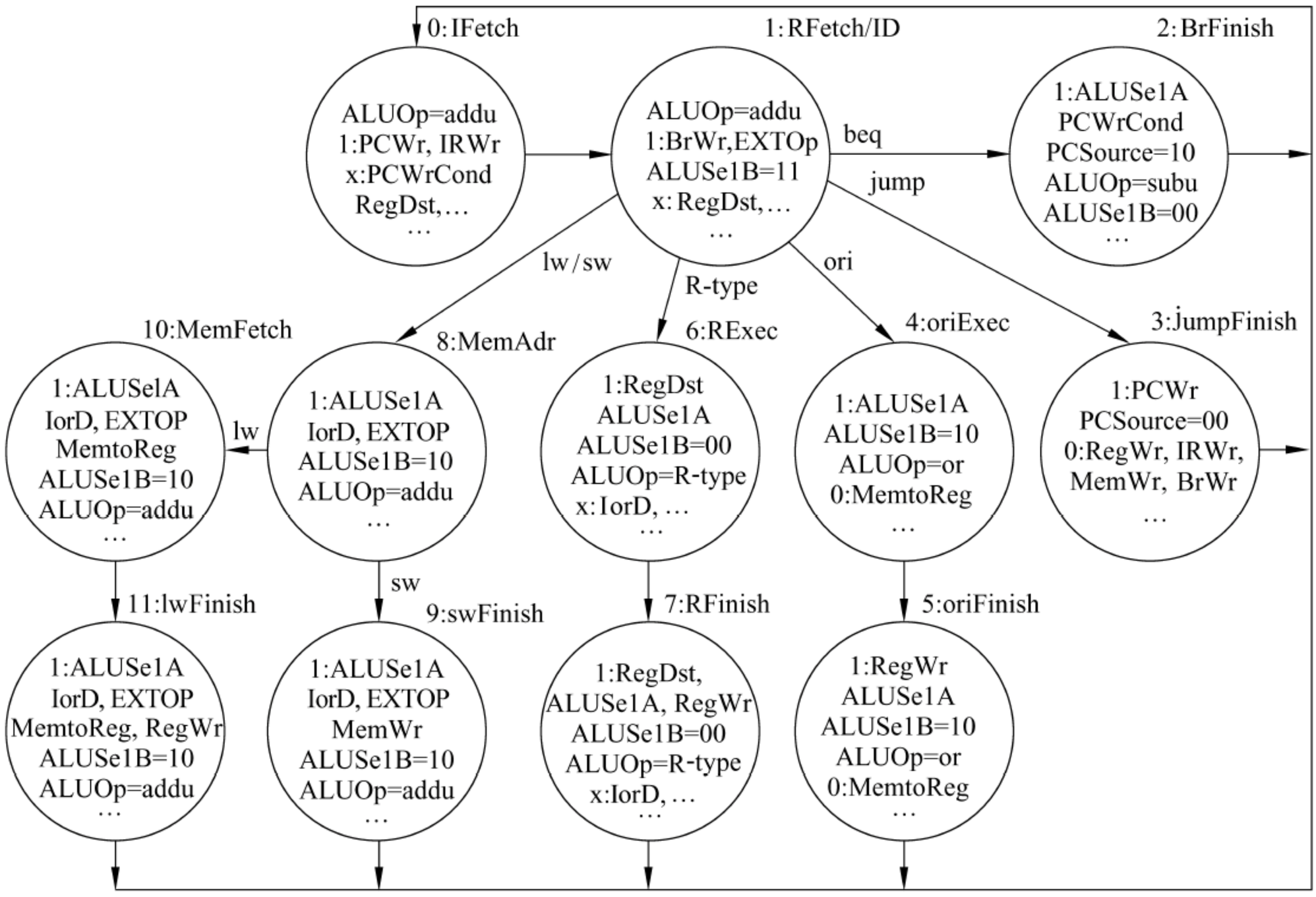


图 5.32　指令执行状态转换图

以，不能像设计单周期控制器那样用简单的真值表描述的方式。多周期控制器通常采用基于有限状态机描述和微程序描述两种方式来实现。

有限状态机描述方式实现的控制器称为有限状态机控制器，其基本思想为：用一个有限状态机描述指令执行过程，由当前状态和操作码确定下一状态，每来一个时钟发生一次状态改变，不同状态输出不同的控制信号值，然后送到数据通路来控制指令的执行。

图 5.33 描述了采用这种方式实现的控制器结构，它由两部分组成：一个组合逻辑控制单元和一个状态寄存器。通常用 PLA 电路实现组合逻辑控制单元。所以，这种控制器也称为组合逻辑控制器、PLA 控制器、硬连线路控制器。

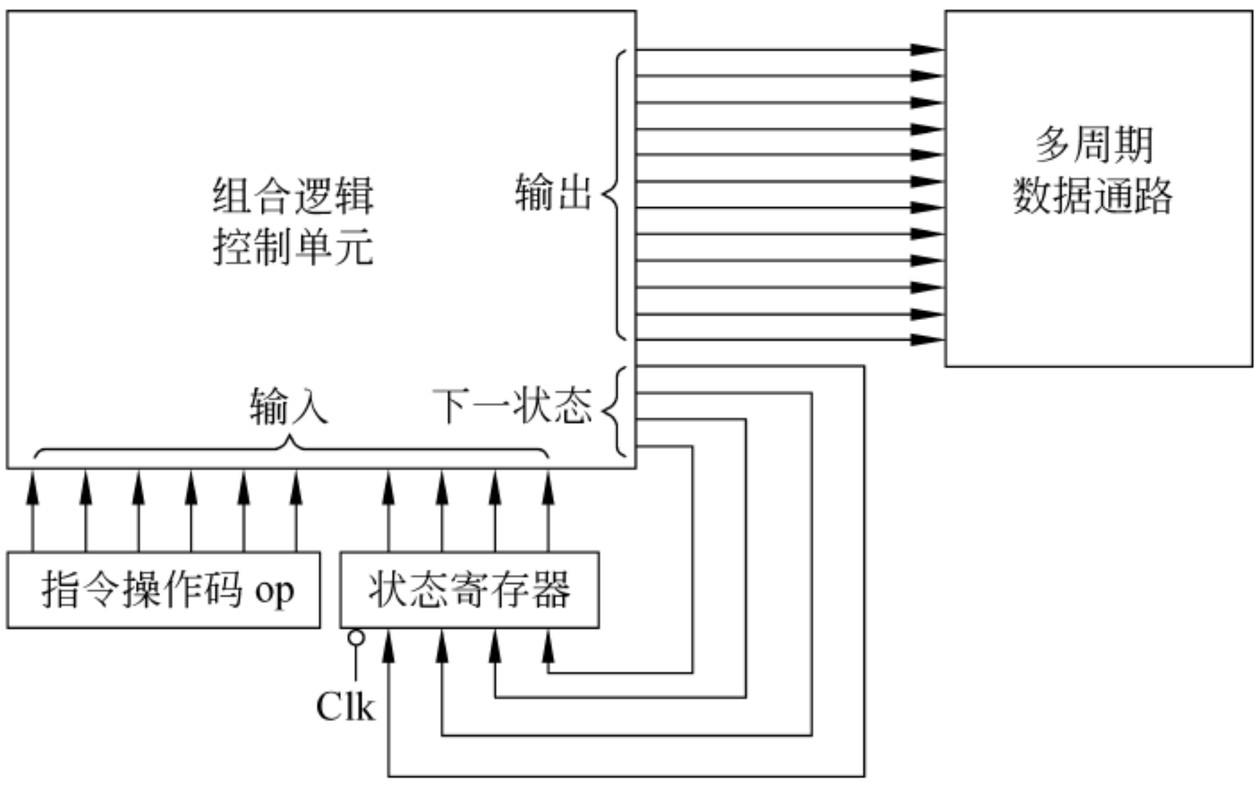


图 5.33　有限状态机控制器结构

对于图 5.33 所示的有限状态机,假定每个状态号如图 5.32 中所设,分别为 0～11,共 12 个状态,因此,状态变量要用 4 位,设分别为 $S_3S_2S_1S_0$。考察每个状态前面的状态和指令操作码,得到状态转换表 5.7。

根据表 5.7 可画出用 PLA 电路实现的状态转换电路以及控制信号生成电路,从而实现组合逻辑控制单元,如图 5.34 所示。该图实现的有限状态机称为“摩尔机”,其特点是控制信号的输出仅依赖于当前的状态,而与其他输入没关系。因此,“摩尔机”方式实现的组合逻辑控制单元被分为两部分:由操作码和当前状态确定下一状态的电路部分和由当前状态确定控制信号的电路部分(图 5.34 中由右下角虚线区域标出的部分)。

表 5.7　多周期控制器状态转换表

当 前 状 态 $S_3\ S_2\ S_1\ S_0$	指令操作码 $op_5\ op_4\ op_3\ op_2\ op_1\ op_0$	下 一 状 态 $NS_3\ NS_2\ NS_1\ NS_0$
2、3、5、7、9、11		0 0 0 0
0		0 0 0 1
1	000100 (beq)	0 0 1 0
1	000010 (jump)	0 0 1 1
1	001101 (ori)	0 1 0 0
4		0 1 0 1
1	000000 (R-type)	0 1 1 0
6		0 1 1 1
1	100011 (lw)	1 0 0 0
1	101011 (sw)	1 0 0 0
8	101011 (sw)	1 0 0 1
8	100011 (lw)	1 0 1 0
10		1 0 1 1

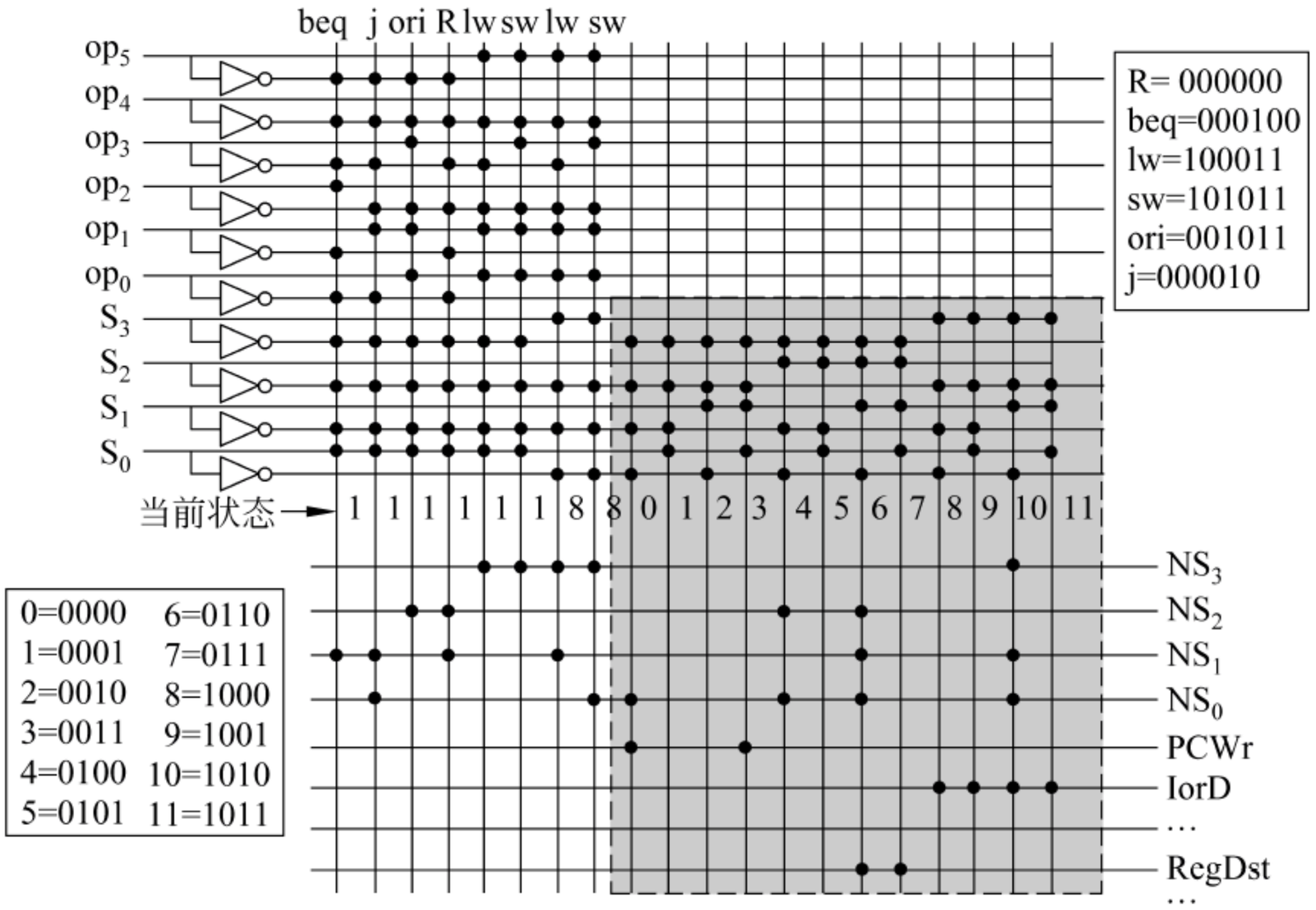


图 5.34　用 PLA 电路实现的组合逻辑控制单元

例 5.1 假定多周期 CPU 采用图 5.31 所示的数据通路和图 5.34 所示的控制单元。若程序中各类指令所占比例为：Load—22%，Store—11%，R-型和 I-型运算—49%，Branch—16%，Jump—2%，则多周期 CPU 比单周期 CPU 大约快多少倍？

解：由图 5.32 知，图 5.31 所示多周期 CPU 中各指令时钟周期数为：Load—5，Store—4，R-型和 I-型运算—4，Branch—3，Jump—3，故 CPI＝0.22×5＋0.11×4＋0.49×4＋0.16×3＋0.02×3＝4.04。

单周期 CPU 的 CPI 为 1，但时钟周期为最长的 Load 指令执行时间，约为多周期时钟宽度的 5 倍。假设单周期时钟宽度为 1，则多周期时钟宽度约为单周期的 1/5，因此，多周期的总体时间约为 4.04×1/5＝0.81；而单周期总体时间为 1×1＝1，故多周期 CPU 的指令执行速度大约是单周期 CPU 的 1/0.81＝1.23 倍。

5.4 微程序控制器设计

硬连线路控制器速度快，适合于简单或规整的指令系统，例如 MIPS 指令集。但是，由于它是一个多输入/多输出的巨大逻辑网络，对于复杂指令系统来说，则对应的硬连线路控制器结构庞杂，实现困难，维护不易，扩充和修改指令相当困难。如果指令系统太复杂，甚至无法用有限状态机描述。所以，对于复杂指令系统或其中的复杂指令，大多采用微程序方式来设计控制器。

微程序控制器是 M. V. Wilkes 最先在 1951 年提出的。用微程序方式实现的控制器称为微程序控制器，其基本思想为：仿照程序设计方法，将每条指令的执行过程用一个微程序来表示，每个微程序由若干条微指令组成，每条微指令相当于有限状态机中的一个状态。所有指令对应的微程序都存放在一个 ROM 中，这个 ROM 称为控制存储器(Control Storage，CS)，简称控存。

在微程序控制器控制下执行指令时，CPU 从控存中取出每条指令对应的微程序，在时钟的控制下，按照一定的顺序执行微程序中的每条微指令。通常一个时钟周期执行一条微指令。

5.4.1 微程序控制器的结构

一条指令的功能通过执行一系列基本操作来完成，这些基本操作称为微操作。每个微操作在相应控制信号的控制下执行，这些控制信号在微程序设计中称为微命令。例如，前面提到的控制信号 PCWr 就是一个微命令，可以控制将信息写入 PC。

微程序是一个微指令序列，对应一条机器指令的功能。每条微指令是一个 0/1 序列，其中包含若干个微命令，它完成一个基本运算或传送功能。有时也将微指令字称作控制字(Control Word，CW)。

图 5.35 给出了微程序控制器的基本结构。其输入是指令和条件码，输出是微命令。图 5.35 中使用了一个微程序计数器 μPC，用来指出微指令在控存中的地址。每次把新指令装入 IR 时，“起始和转移地址发生器”将根据指令内容，生成微程序入口地址放入 μPC 中，以后每来一个时钟，μPC 自动增值(＋“1”)，这样，依次从控存中读出一条条微指令执行。μIR 为微指令寄存器，存放从控存取出的微指令，每条微指令被译码后，产生一系列微

命令,送到数据通路中。机器指令的执行过程常常与条件码(即标志)有关,因此微程序中也引入了条件转移概念。微指令中的“转移控制”部分被送到转移地址发生器,根据条件码及相应微命令产生新的微指令地址送入 μPC。

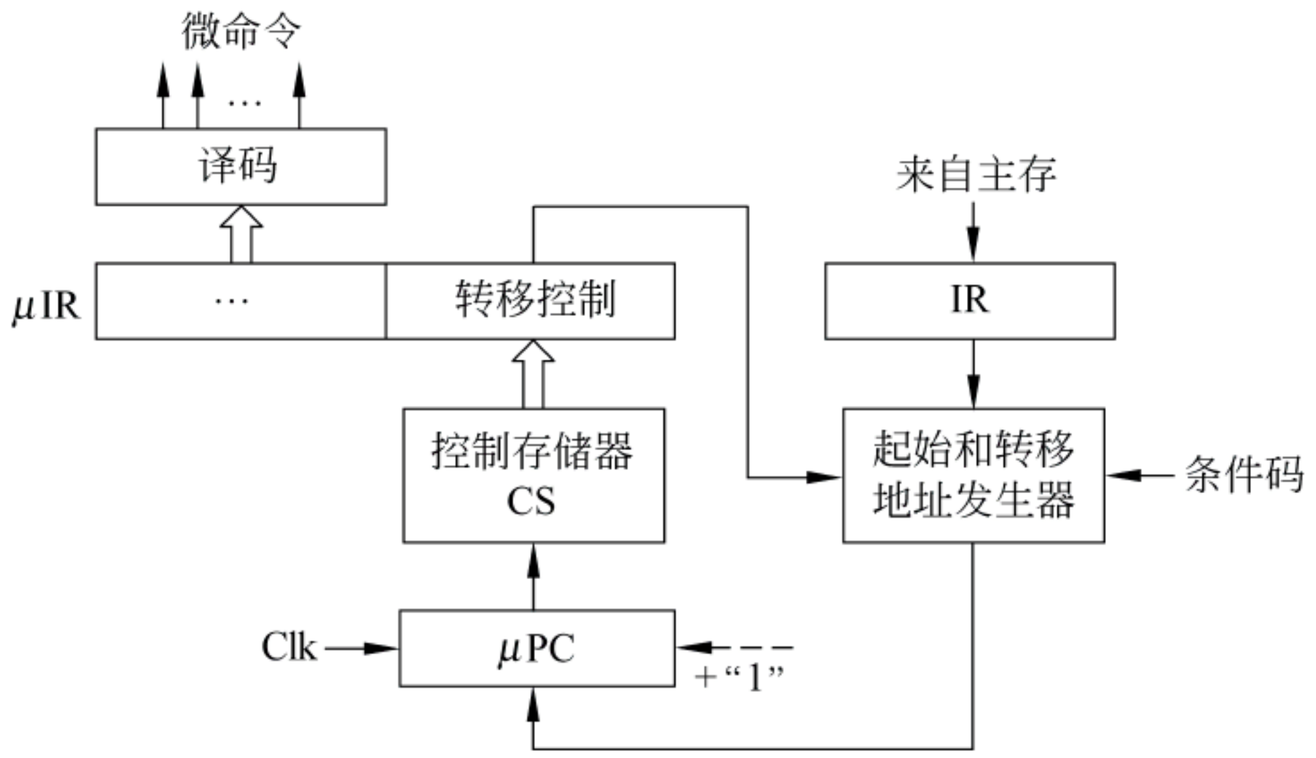


图 5.35　微程序控制器基本结构

取指令过程是每条指令的公共操作,可以专门用一个取指令微程序来实现。因此,微程序控制器的工作流程就是不断地执行取指令微程序和执行相应指令功能对应的微程序的过程。

微程序由微指令组成,微程序的执行要解决与程序的执行类似的两个问题:①微指令格式和微命令编码问题;②下一条微指令地址确定问题。

*5.4.2　微命令编码

为了加快指令执行速度,通常采用定长微指令字格式。与指令由操作码和地址码组成类似,微指令由微操作码和微地址码两部分组成。微操作码格式设计主要由微命令编码方式决定。微命令编码方式主要有不译法(直接控制法)和字段直接编码法两种,早期还有字段间接编码法和最小(最短、垂直)编码法,现在基本不用了。

1. 直接控制法

一位对应一个微命令,微操作码的长度与所有微命令的个数相当,无须译码,因此,也称为不译法。显然,直接控制法的并行控制能力强,不必译码,控制电路简单、速度快。但是,由于一台机器的微命令个数很多,通常达几百个,因此微指令字可能多达几百位,实现起来非常困难。此外,因为一条微指令中往往只包含很少几个微操作,所以只有很少几位对应的微命令为 1,使得几百位的微指令字中只有少数几位为 1,因而编码空间利用率低。

2. 字段直接编码法

数据通路中的微操作之间存在两种关系:相容和互斥。相容微操作是指在数据通路中能同时进行的微操作,对应的微命令称为相容微命令;互斥微操作是指不能同时进行的微操作,对应的微命令称为互斥微命令。

字段直接编码法的基本思想是:将微指令分成若干字段,每个字段包含若干微命令。把互斥微操作组合在同一字段,相容微操作组合在不同字段,对每一字段内的微操作进行编码。因此,一条微指令中最多可同时发出的微操作数就是微命令字段的个数。

例 5.2　对于图 5.7 和图 5.8 所示的单总线数据通路,假定有 4 个通用寄存器 R0、R1、

R2 和 R3，ALU 操作类型共 16 种，内存和 CPU 之间采用“异步”方式通信，存取操作由 Read 和 Write 两种信号控制。每条指令执行结束时，都要执行一个公共操作，由控制信号 End 控制进行指令结束处理，查询是否有外部中断请求等。请分别写出采用直接控制法和字段直接编码法的微操作码格式。

解：虽然在图 5.7 所示的单总线数据通路中没有标出控制信号(微命令)，但是，不难看出其控制信号包括以下几类。

(1) 根据图 5.8 所示，控制在寄存器和内总线之间进行传送的控制信号有 16 个：R0in、R0out、R1in、R1out、R2in、R2out、R3in、R3out、Yin、Zout、MARin、MDRin、MDRout、PCin、PCout、IRin。

(2) ALU 操作类型有 16 种：add、sub、or、and、xor、…、mov，因此控制信号占 4 位。

(3) ALU 进位控制信号有 1 个：1→C0。

(4) 暂存器 Y 清零控制信号有 1 个：ClearY。

(5) 内存读写控制信号有 3 个：Read、Write、WMFC。

(6) 结束控制信号有 1 个：End。

如果采用直接控制法，则微指令字中微操作码部分的长度就是控制信号的总个数。本例中共有控制信号(微命令)数为 16＋4＋1＋1＋3＋1＝26 个，因此，微操作码共 26 位，若某位为 1，则对应的微命令有效，否则，对应微命令无效。需要说明的是，对于 ALU 操作控制信号，其控制信号的个数并不是 16，而是 4，这是由 ALU 的结构和功能确定的(参见 3.2.4 节)。

如果采用字段直接编码法，则需根据微操作之间的相容和互斥关系进行分组。假设 ALU 操作控制信号为 ALUOp＜3:0＞，则分组和编码情况如表 5.8 所示。

表 5.8　单总线数据通路微命令分组情况及其编码表

组　　名	微操作(编码)	微命令取值	说　　明
寄存器送总线控制(Rout)	No action (0000) R0→Bus (0001) R1→Bus (0010) R2→Bus (0011) R3→Bus (0100) Z→Bus (0101) MDR→Bus (0111) PC→Bus (1000)	组内所有都为 0 $R0_{out}=1$，其余为 0 $R1_{out}=1$，其余为 0 $R2_{out}=1$，其余为 0 $R3_{out}=1$，其余为 0 $Z_{out}=1$，其余为 0 $MDR_{out}=1$，其余为 0 $PC_{out}=1$，其余为 0	(1) 有些微指令中，不需要有寄存器内容打到总线上，故“No action”。 (2) 某一时刻只能有一个寄存器内容可以打到总线，此时，相应寄存器的输出微命令取值为 1，组内其余寄存器输出微命令的取值都为 0
寄存器输入控制(Rin)	No action (000) Bus→R0 (001) Bus→R1 (010) Bus→R2 (011) Bus→R3 (100) Bus→Y (101) Bus→PC (110) Bus→IR (111)	组内所有都为 0 $R0_{in}=1$，其余为 0 $R1_{in}=1$，其余为 0 $R2_{in}=1$，其余为 0 $R3_{in}=1$，其余为 0 $Y_{in}=1$，其余为 0 $PC_{in}=1$，其余为 0 $IR_{in}=1$，其余为 0	某一时刻，可以同时将总线内容送多个寄存器，所以，组内这些微操作并不互斥，但基本上不会同时发生，为了缩短微操作码的长度，将它们分在同一组
MAR 和 MDR 输入控制(MRin)	No action (00) MARin (01) MDRin (10)	$MAR_{in}=0$，$MDR_{in}=0$ $MAR_{in}=1$，$MDR_{in}=0$ $MAR_{in}=0$，$MDR_{in}=1$	可能和 Rout 组操作同时发生，故分在不同组

续表

组　　名	微操作(编码)	微命令取值	说　　明
ALU操作控制 (ALUOp)	add　(0000) sub　(0001) and　(0010) or　(0011) xor　(0100) … mov　(1111)	ALUOp=0000 ALUOp=0001 ALUOp=0010 ALUOp=0011 ALUOp=0100 … ALUOp=1111	ALU的操作由ALU的操作控制端ALUOp控制
ALU进位控制 (Cin)	0→Cin　(0) 1→Cin　(1)	1→C_0=0 1→C_0=1	
暂存器清零控制 (ClearY)	No action　(0) ClearY　(1)	ClearY=0 ClearY=1	
内存读写控制 (MEMOp)	No action　(00) Read　(01) Write　(10)	Read=0,Write=0 Read=1,Write=0 Read=0,Write=1	
等待MFC控制 (WMFC)	No action　(0) 采样MFC信号　(1)	WMFC=0 WMFC=1	WMFC=1,则启动CPU对MFC信号进行采样,若有效,则说明内存完成读写,否则,CPU继续等待
指令结束控制 (END)	No action　(0) 启动中断查询等(1)	End=0 End=1	End=1,则启动CPU对中断请求信号采样,若有效,则进入中断响应,否则,CPU继续执行下一条指令

从表6.8可看出,采用字段直接编码方式,共有9组,其微操作码的位数从直接控制方式的26位减少到了4+3+2+4+1+1+2+1+1=19位。

例5.3　对于图5.31所示的多周期数据通路,假定采用字段直接编码法,则微指令字可划分为几个微命令字段?请给出一种微命令编码方案,并根据该方案写出微程序。

解:图5.31所示的多周期数据通路中共有15种控制信号(共19位):ExtOp,ALUSelA,ALUSelB,ALUOp,IorD,RegWr,PCWr,PCWrCond,IRWr,MemWr,BrWr,MemtoReg,PCSource,RegDst,R-type。涉及的微操作分为9组,每组内微操作互斥,分组情况以及其中的一种编码方案如表5.9所示。表中×表示任意,可以用0或1代替。

表5.9中有6组是一个微命令控制一个微操作的情况,所以,这些字段的微命令编码很简单,就用微命令的取值作为编码;还有3组是多个微命令控制一个微操作,所以,需要进行编(译)码,编码后可缩短微命令字段长度。例如,内存访问控制字段(MemOp)从3位缩短到两位,写PC控制字段(PCWrOp)从4位缩短到两位。但是,寄存器操作控制字段(RegOp)包含的微操作有3个,需要用3位编码,而本身包含的微命令也只有3个,不编码的话也只要3位,而且执行时不需要译码,所以,这个字段还是不组合为好。按表5.9表示的微指令字(仅包含微命令字段)的长度为16位,实际上也仅比未编码的直接控制方式少3位,并没有节省多少空间。因此,到底采用直接控制法还是采用编码法需要很好地权衡。

每种微指令字组合相当于图5.32中的一个状态,可将图5.32的有限状态机用微程序来表示,结果如表5.10所示。这样每个状态下的0/1序列就构成了一条微指令,共有12条微指令。

表 5.9　多周期数据通路微操作分组情况及其编码表

组　　名	微操作(编码)		微命令取值	说　　明
ALU 操作控制(ALUOp)	addu	(000)	ALUOp=000	lw/sw 指令计算访存地址或 addiu 执行时,进行 addu 操作
	subu	(001)	ALUOp=001	beq 指令比较大小时,进行 subu 操作
	or	(010)	ALUOp=010	ori 指令时 ALU 进行 or 操作
	R-Type	(001)	ALUOp=001	由 func 确定 ALU 操作类型
ALU 的 A 端口输入控制(ALUSelA)	PC→A 口	(0)	ALUSelA=0	PC 作为 ALU 第一个操作数
	A→A 口	(1)	ALUSelA=1	A 作为 ALU 第一个操作数
ALU 的 B 端口输入控制(ALUSelB)	B→B 口	(00)	ALUSelB=00	B 作为 ALU 第二个操作数
	4→B 口	(01)	ALUSelB=01	4 作为 ALU 第二个操作数
	Ext→B 口	(10)	ALUSelB=10	扩展器输出作为 ALU 第二个操作数
	Ext<<2→B 口	(11)	ALUSelB=11	扩展器内容乘 4 作为 ALU 第二个操作数
寄存器操作控制(RegOp)	ori-Ready	(000)	RegWr=0,RegDst=0,MemtoReg=0	ori 指令结果写寄存器堆前先送数据和地址
	lw-Ready	(001)	RegWr=0,RegDst=0,MemtoReg=1	lw 指令结果写寄存器堆前先送数据和地址
	R-Ready	(010)	RegWr=0,RegDst=1,MemtoReg=0	R-型指令结果写寄存器堆前先送数据和地址
	Read	(011)	RegWr=0,RegDst=×,MemtoReg=×	将 IR<25:21>、IR<20:16>分别作为寄存器读地址,读出后分别送 A 和 B
	Write ALU-R	(100)	RegWr=1,RegDst=1,MemtoReg=0	R-型指令结果写寄存器堆
	Write ALU-ori	(101)	RegWr=1,RegDst=0,MemtoReg=0	ori 指令结果写寄存器堆
	Write Dout	(110)	RegWr=1,RegDst=0,MemtoReg=1	lw 指令取出内存单元内容写寄存器堆
内存访问控制(MemOp)	Read Instruction	(00)	MemWr=0,IorD=0,IRWr=1	PC 为地址,读出指令送 IR
	Read Data	(01)	MemWr=0,IorD=1,IRWr=0	ALU 结果作为地址,读内存
	Write Data	(10)	MemWr=1,IorD=1,IRWr=0	ALU 结果作为地址,写内存

续表

组　　名	微操作(编码)	微命令取值	说　　明
扩展器操作控制(ExtOp)	SignExt (1)	ExtOp=1	计算访存地址或分支目标地址时需要进行符号扩展
	ZeroExt (0)	ExtOp=0	ori 指令需要进行零扩展
R-型指令 ALU 操作控制(RType)	R-type (1)	R-type=1	由 func 确定 ALU 操作类型
	NR-type (0)	R-type=0	由 op 确定 ALU 操作类型
分支目标地址写入控制(BrWr)	Keep Branch-Target the same (0)	BrWr=0	在分支目标地址读出之前,不能改变 Branch-Target 的值
	Write Branch-Target (1)	BrWr=1	事先计算出的分支目标地址需要保存到 Branch-Target 中,以便在需要时读出送 PC
写 PC 控制(PCWrOp)	Keep PC the same (00)	PCSource=××, PCWr=0, PCWrCond=0	在译码/取数、非转移指令执行等阶段不能改变 PC 的值
	PC+4 (01)	PCSource=01, PCWr=1, PCWrCond=×	顺序执行下一条指令
	PC+4+Ext≪2 (10)	PCSource=10, PCWr=0, PCWrCond=1	分支条件满足时,指令转移到分支目标地址执行
	PC[31−28]‖IR[25−0]‖00 (11)	PCSource=00, PCWr=1, PCWrCond=×	无条件转移目标地址送 PC

表 5.10　图 5.32 中有限状态机的各状态对应的微指令

状态号	ALUOp	ALUSelA	ALUSelB	RegOp			MemOp	ExtOp	R-type	BrWr	PCWrOp
				RegWr	RegDst	MemtoReg					
0	000	0	01	0	×	×	00	×	0	0	01
1	000	0	11	0	×	×	01	1	0	1	00
2	100	1	00	0	×	×	01	×	0	0	10
3	×××	×	××	0	×	×	01	×	0	0	11
4	010	1	10	0	0	0	01	0	0	0	00
5	010	1	10	1	0	0	01	0	0	0	00
6	001	1	00	0	1	0	01	×	1	0	00
7	001	1	00	1	1	0	01	×	1	0	00
8	000	1	10	0	×	×	01	1	0	0	00
9	000	1	10	0	×	×	10	1	0	0	00
10	000	1	10	0	×	×	01	1	0	0	00
11	000	1	10	1	×	×	01	1	0	0	00

字段直接编码方式采用相容微操作分在不同字段，因此，能最大限度地并行执行微操作，具有较高的并行控制能力，速度较快。此外，由于对同字段的微操作进行了编码，缩短了微命令字段的长度，因而微指令字较短，节省了控存容量。但是，对于多个微命令组合的字段，由于进行了编码，所以执行时需要相应的译码线路，这样，与直接控制法相比，增加了译码线路，加大了一些成本，并多出了一部分译码时间，不过分段后各字段位数少，所以译码对微指令的执行速度影响不大。

3. 字段间接编码法

在字段直接编码法基础上，可通过字段间接编码的方式进一步压缩微指令长度。字段间接编码是指：某一字段可以表示多个微命令组，到底代表哪组微命令，则由另一个专门的字段确定。

虽然这种方式可进一步缩短微指令字的长度，节省控存容量，但意义不大，而且译码线路复杂，时间开销大。通常极少使用这种方式。

4. 最少(最短、垂直)编码法

其基本思想是：采用类似指令编码的思想，即，整个微操作码部分作为一个字段，每次只产生一个微操作。采用这种方式得到的微操作码位数最少，长度最短，所以有时称为最少(最短)编码法；用这种格式的微指令编写的微程序较长，所以也称为垂直型微指令。

*5.4.3　微指令地址的确定

当前微指令执行结束后，必须确定下一条执行的微指令。若当前执行的是某条机器指令对应微程序的最后一条微指令，则下一条微指令就是取指令微程序的第一条微指令；若执行完取指令微程序的最后一条微指令，则下一条微指令就是当前指令对应微程序的首条微指令；若在某个微程序执行过程中，则可能按顺序取出下一条微指令执行，或者无条件转到另一处微指令执行，或根据条件码或指令操作码选择不同分支处的微指令执行。

可以通过在微指令中显式或隐含地指定下一条微指令在控存中的地址(简称下址)来解决下一条微指令的确定问题。下一条微指令地址的确定方式有两种：计数器法(即增量法)和断定法(即下址字段法)。

1. 计数器法

计数器法的主要思想是：使用一个专门的微程序计数器 μPC,将下条微指令地址隐含地存放在 μPC 中,因此,这种方法称为计数器法。顺序执行时,根据 μPC+1→μPC,得到下一条微指令地址;转移执行时,在当前微指令后添加一条转移微指令,并在微指令中添加专门的转移控制字段,将转移微指令或转移控制字段中的控制信息送到微指令地址发生器,与相应的指令操作码以及条件码等组合,生成转移地址送 μPC。图 5.35 所示的就是采用计数器法的微程序控制器基本结构。当转移分支很多时,相应的微地址生成逻辑电路很复杂。为简化微地址生成逻辑,通常采用 PLA 或 ROM 来实现。

2. 断定法

计数器法的缺点是必须在不连续执行的微指令之间加入转移微指令,这样,在增加微指令条数的同时,还严重影响指令执行速度。如果在微指令中直接明确指定下一条微指令地址,这样,相当于每条都是转移微指令,即使不连续执行也没有关系。这种方法称为断定法,也称为下址字段法。断定法虽然加快了指令执行速度,但因为增加了微指令的长度,从而影响控制存储器的有效利用。例如,假定一个采用微程序控制器的处理器的控存容量为 4KB,共有 500 条左右的微指令,这意味着下址字段至少有 9 位,微指令长度为 64 位左右,其中除了下址字段以外,还有一些其他如控制多分支转移的条件测试和转移控制字段等,也都用于控制微指令的寻址,因此,大约有五分之一的控存空间用于微指令寻址,真正用来存放微命令的空间只有五分之四左右。

此方法最明显的优点是消除了专门的转移微指令,而且在给微指令分配地址时不需要考虑如何排列,也不需要对 μPC 增量,而用一个简单的微指令地址寄存器(μAR)来存放当前微地址。采用断定法的微程序控制器结构如图 5.36 所示,其中,微地址修改逻辑根据当前指令、状态条件、下址字段和转移控制字段来确定微程序的执行顺序。

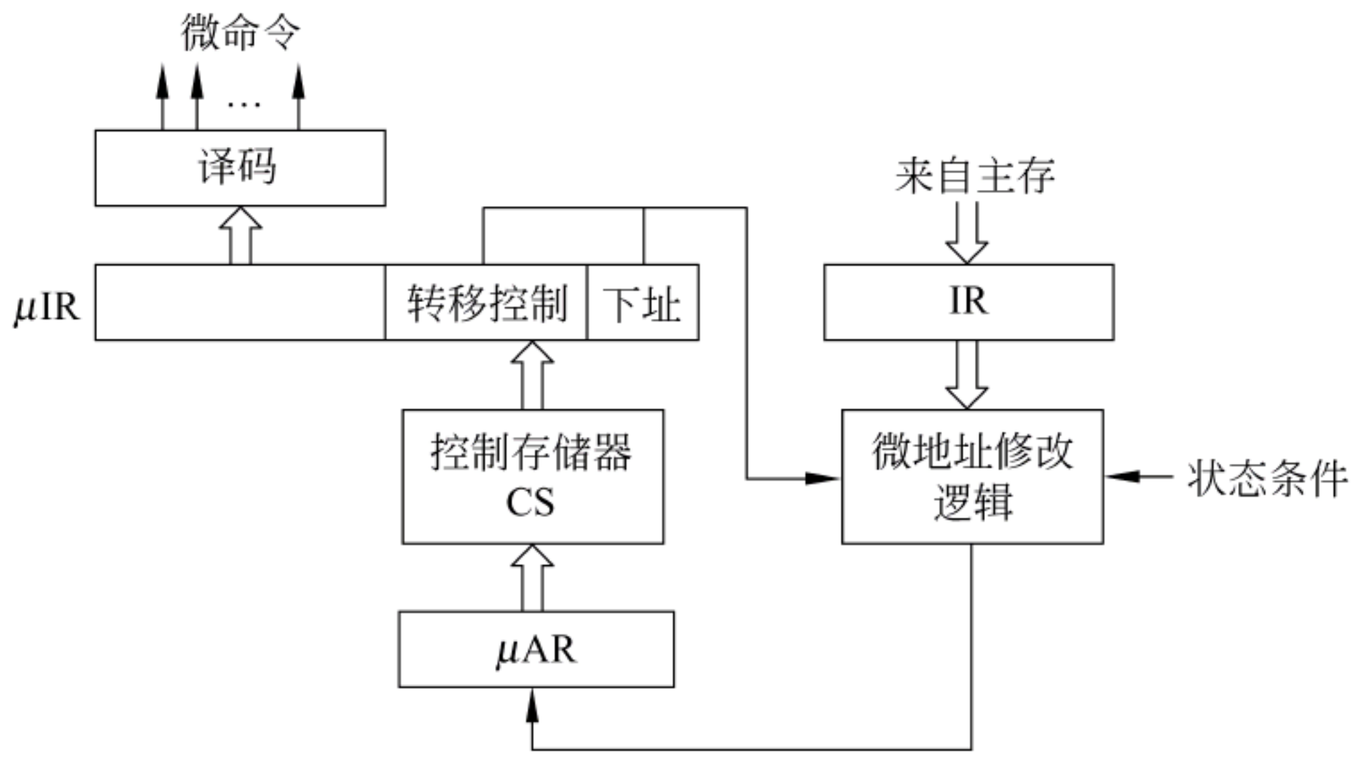


图 5.36　采用断定法的微程序控制器结构

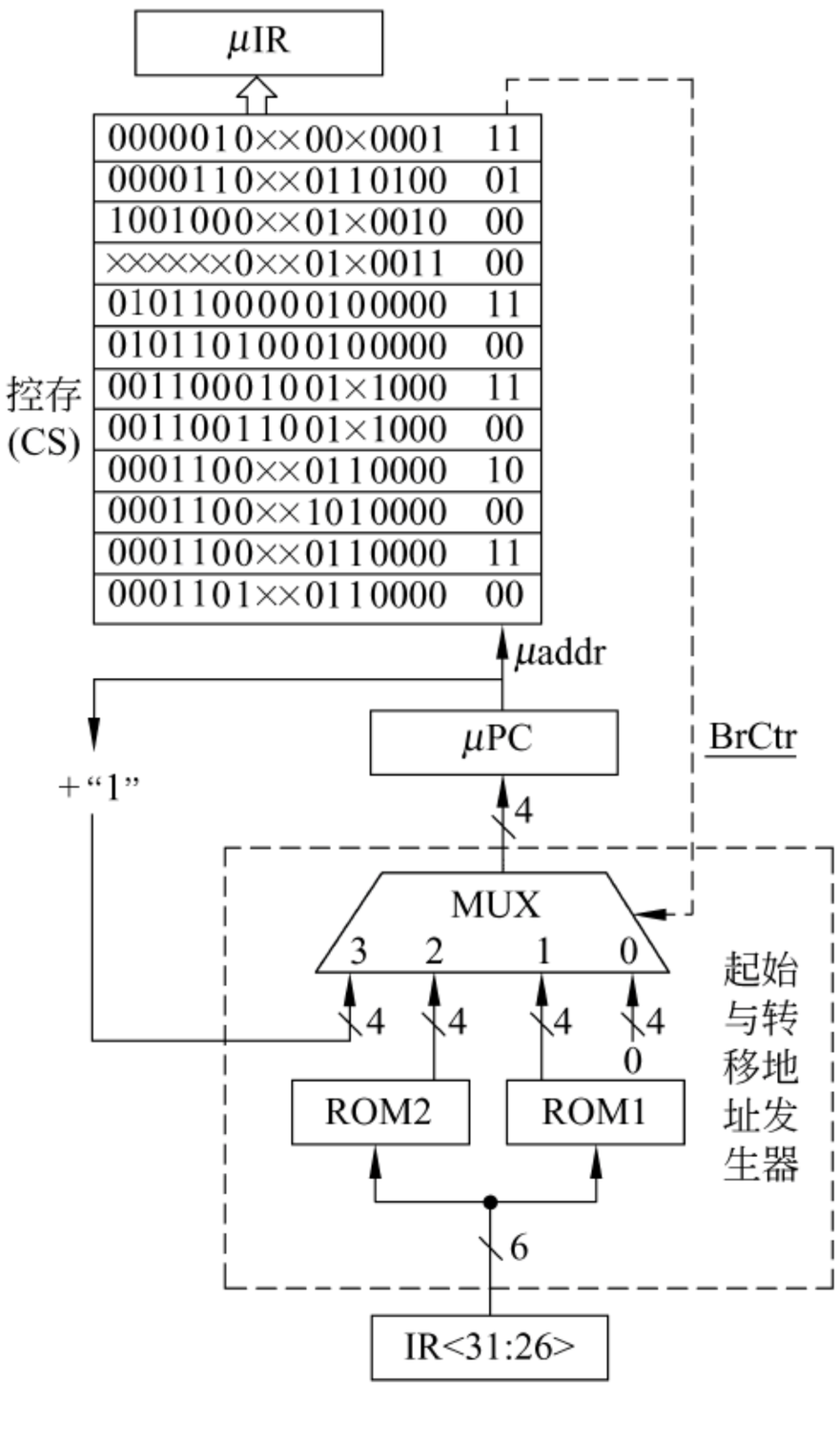
下面通过具体例子来介绍计数器法和断定法的相关实现细节。

例 5.4　假定采用在微指令字中增加转移控制字段的方式来实现分支,并用 ROM 方式

实现机器指令微程序首地址的生成，那么，对于表 5.10 给出的微程序，请分别给出采用计数器法和下址字段法时的微程序控制器结构。

解：表 5.10 的微程序中，将状态号作为微指令地址，所以微地址共 4 位。其中，取指令微程序首址为 0000，地址 0001 处的微指令执行结束后，可能会转到不同指令对应的微程序执行，其分支转移功能由 ROM1 实现，可能转到 beq、jump、ori、R-型、lw/sw 这 5 种指令对应的微程序去执行，beq 和 jump 指令的微程序都只有一条微指令，分别存放在地址 0010 和 0011 中；ori 和 R-型指令的微程序都有两条微指令，其首地址分别为 0100 和 0110；lw/sw 指令的微程序有 4 条微指令，首地址为 1000，微程序中有一个分支转移点，需要根据当前指令是 lw 还是 sw 来决定 1000 处的微指令执行后是转到 1010 处执行还是 1001 处执行。

图 5.37(a)给出的是计数器法微程序控制器的结构。图中 μPC 具有自增功能，每来一个时钟自动加 1。lw/sw 指令微程序中的分支功能由 ROM2 实现。BrCtr 微命令是在微指令字中增加的转移控制字段，用来控制下一条微地址的选择方式。BrCtr 的含义如下。



(a) 计数器法微程序控制器结构　　(b) 断定法微程序控制器结构

图 5.37　表 5.10 中的微程序的两种实现方式

BrCtr＝00：转到取指令微程序首条微指令执行。

BrCtr＝01：转到由 ROM1 的输出所指的微指令执行。

BrCtr＝10：转到由 ROM2 的输出所指的微指令执行。

BrCtr＝11：按顺序执行。

图 5.37(b)给出的是断定法微程序控制器的结构。下一条微指令地址可直接来自当前

微指令字的下址字段,但在某些情况下需要对其进行修改。例如,lw/sw 指令微程序中的分支功能由操作码 IR<31:26>(对应 op5～op0)中的 op3 对微地址修改实现。因为 OP(lw)=100011,OP(sw)=101011,两者的差别仅在 op3。假定 1000 处微指令的下址字段为 1010,则当 op3=0 时不需修改微地址;当 op3=1 时需将 1010 修改为 1001,所以,只要将 4 位地址的后 2 位与 op3 进行"异或"即可。BrCtr 微命令是在微指令字中增加的转移控制字段,用来控制下条微地址的选择方式。BrCtr 的含义如下:

BrCtr=00:转到微指令的下址字段指出的微指令执行。

BrCtr=01:转到由 op3 修改后的微地址所指的微指令执行。

BrCtr=10:转到由 ROM1 的输出所指的微指令执行。

图 5.37 的计数器法和断定法中都用 ROM 方式实现多分支转移,ROM1 中存放的是各指令微程序的首地址;ROM2 中只有两个单元,分别存放 1001 和 1010。断定方式下微指令长度为 22 位,计数器法的微指令长度为 18 位。

微程序设计的思想给计算机控制部件的设计和实现技术带来了巨大的影响。与硬连线设计相比,它大大降低了控制器设计的复杂性,提高了设计的标准化程度。由于机器指令的执行过程用微程序控制,因而提供了很大的灵活性,使得设计的变更、修改以及指令系统的扩充都成为不太困难的事情。它与传统的软件设计有许多类似之处,但是,由于微程序相对固定,且通常不放在主存内,故有可能利用工作速度较高的 ROM 存放微程序,从而缩短微程序的运行时间。这是一种固化了的微程序,称为固件(firmware)。

微程序控制器的主要缺点是比相同或相近指令系统的硬连线控制器慢。因此,RISC 机大都采用硬连线控制器,而 IA-32 则采用了硬连线和微程序相结合的方式来实现控制逻辑。

5.5 异常和中断处理

5.5.1 基本概念

在程序正常执行过程中,CPU 会遇到一些特殊情况而无法继续执行当前程序。这种中断程序正常执行的情况主要有以下两大类。

1. 内部异常

内部异常(exception)是指由处理器内部异常引起的意外事件。根据其发生的原因又分为硬故障中断和程序性异常。硬故障中断是由硬连线路出现异常引起的,如电源掉电、存储器线路错等;程序性异常也称软中断,是由 CPU 执行某个指令而引起的发生在处理器内部的异常事件,也称为例外。如整除 0、溢出、断点、单步跟踪、访问超时、非法操作码、栈溢出、缺页、地址越界等。按发生异常的报告方式和返回方式的不同,内部异常可分为故障(fault)、自陷(trap)和终止(abort)三类。

1) 故障

故障也称为失效,它是在引起故障的指令启动后、执行结束前被检测到的一类异常事件。例如,指令译码时,出现"非法操作码";取指令或数据时,发生"段不存在"、"缺页"或"保护错";执行整数除法指令时,发现"除数为 0"等。显然,"段不存在"、"缺页"等这类异常处