对于所有指令用一个固定时钟周期的计算机，时钟周期长度将由最长的指令决定，在此为600 ps。（这只是一种近似的计时，因为计时模型很简单。实际上，现代数字系统的计时很复杂，通常可以从一个时钟周期中借一点时间给下一个时钟周期使用。）

对时钟周期长度可变的计算机，其长度将在200 ps至600 ps之间变化。根据上面的信息和指令频度分布情况，可得出这种计算机的平均时钟周期长度。

于是，一条指令的平均执行时间为：

CPU时钟周期 $= 600 \times 25\% + 550 \times 10\% + 400 \times 45\% + 350 \times 15\% + 200 \times 5\% = 447.5$ ps

由于可变时钟周期的实现方法中平均时钟周期较短，它当然较快。来计算一下性能比：

$$\frac{\text{CPU 性能}_{\text{可变时钟}}}{\text{CPU 性能}_{\text{固定时钟}}} = \frac{\text{CPU 执行时间}_{\text{固定时钟}}}{\text{CPU 执行时间}_{\text{可变时钟}}}$$

$$= \left(\frac{\text{IC} \times \text{CPU 时钟周期}_{\text{固定时钟}}}{\text{IC} \times \text{CPU 时钟周期}_{\text{可变时钟}}} = \frac{\text{CPU 时钟周期}_{\text{固定时钟}}}{\text{CPU 时钟周期}_{\text{可变时钟}}} \right)$$

$$= \frac{600}{447.5} = 1.34$$

可见可变时钟周期的实现方式快了1.34倍。遗憾的是，为每个指令类型实现一个可变的时钟周期非常困难，而且所导致的额外开销可能得不偿失。在下一节中，将介绍另一种方法，它通过在更短的时钟周期做较少的工作，改变每种指令类型所用的时钟周期数。 ■

使用固定时钟周期的单周期设计的代价是很大的，但对于小指令集可能可以接受。历史上看，具有非常简单的指令集的早期计算机的确使用了这项技术。可是若要实现浮点单元和包含更复杂指令的指令集，这样的单周期设计根本不能胜任。下面给出CD中■ **For More Practice** 练习5.4中的一个例子。

因为必须假定时钟周期等于所有指令中最坏情况下的延迟，因此在不改进最坏时钟周期时间情况下，不能使用减少一般操作延迟的技术。所以，单周期的实现方法违反了关键的设计原则，即加速完成常用操作。另外，在单周期实现中，一个时钟周期内每个功能单元只能使用一次。因此，某些功能单元必须有副本，这就增加了实现的代价。从性能和硬件成本来说单周期设计都是低效的！

为避免这些困难，可以使用具有更短时钟周期的实现技术——时钟周期由基本功能单元延时决定——并且每条指令需要多个时钟周期完成。下一节就将讨论这种替代实现方案。在第6章中，将要介绍另一种实现技术，称为流水线，流水线使用类似单周期数据通路的通路，但是更有效率。通过重叠执行多条指令，流水线获得更高的效率，提高了硬件利用率并且提高了系统性能。只需对处理器使用的高层概念感兴趣的读者，掌握本节的内容足以阅读第6章的概述小节，并且可以理解流水线处理器的基本功能。要是想要理解硬件实际如何实现控制，还要继续学习下面的章节！

自测

参考图5-22中的控制信号。图中任何控制信号都能被另一个的非替换吗？如果可以，是否可以在不添加非门的情况下，使用一个信号作为另一个控制信号。

5.5 多周期实现方案

在前面的一个例子中，曾根据所需进行的功能单元的操作，将一条指令的执行分解为一系列步骤。可以用这些步骤来生成一个**多周期实现**[⊖]方案。在这个方案中，指令的每一步将占用一个时钟周期。一个功能单元可以在一条指令的执行过程中使用多次，只要是在不同周期中。这种共享可减少所需硬件数量。允许指令的执行占用不同周期数和功能单元可在单指令执行过程中共享，是多周期设计的主要优点。图5-25给出了多周期数据通路的大致轮廓。和图5-11的单周期数据通路相比，可发现以下区别：

⊖ 多周期实现(multicycle implementation) 也称为多时钟周期实现，是指一条指令的执行要经过多个时钟周期。

- 指令和数据使用同一个存储器单元。
- 只有一个 ALU，而不是一个 ALU 和两个加法器。
- 每个主要的功能单元都加上了一个或多个寄存器存储输出值，以便在后面的时钟周期中使用。

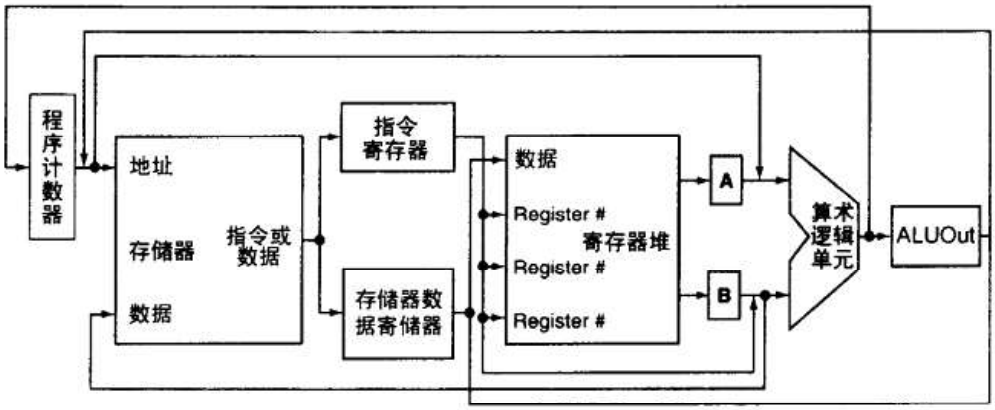

图 5-25 多周期数据通路的高层视图

[这张图画出了数据通路的关键部件：共享的存储单元、一个各指令共享的 ALU 和各共享单元之间的联系。共享功能单元的使用要求增加或扩展多路复用器，以及用于在同一条指令不同时钟周期间存储数据的临时寄存器。附加的寄存器有指令寄存器(IR)、存储器数据寄存器(MDR)、A、B 和 ALUOut]

一个时钟周期结束时，在以后的周期中将要用到的所有数据都必须存储在状态单元中。以后时钟周期中随后的指令将要用到的数据被存入一个程序员可见的状态单元中(即寄存器堆、PC 或存储器)。相反，在以后的周期中同一指令要用到的数据必须存入这些附加寄存器中。

这样，附加寄存器究竟放在哪里将由两个因素决定：哪些组合单元适合在单个周期中使用，以及哪些数据将在指令执行过程中后面的各个周期中使用。在这个多周期设计中，假设一个时钟周期内最多只能完成下列操作之一：一次访存、一次寄存堆访问(2 次读或 1 次写)或一个 ALU 操作。于是这三个功能单元的任何一个(存储器、寄存器堆或 ALU)产生的数据必须存储在一个临时寄存器中，以供后面的周期使用。如果没有被缓存，则可能出现周期竞争，导致使用不正确的值。

为满足这些要求，加入下面的临时寄存器：

- 指令寄存器(IR)和存储器数据寄存器(MDR)分别用于存储从存储器读取的指令和数据的输出。使用两个独立的寄存器，是因为在同一周期中两者的值都要被用到，稍后会进一步说明。
- 寄存器 A、B，用于存储从寄存器堆中读出的寄存器操作数。
- 寄存器 ALUOut，用于存储 ALU 的输出。

除 IR 外，所有寄存器只在相邻两时钟周期之间存储数据，所以它们不需要写控制信号。IR 需保存指令至其执行结束，所以需要一个写控制信号。这个区别将在研究了每条指令的各个时钟周期后更加清楚。

由于一些功能单元要为不同目的所共享，所以需要添加多路复用器，并扩展已有的多路复用器。比如，由于一个存储器既用于存储指令也用于存储数据，就需要一个多路复用器来为一个存储地址选择数据来源，即 PC(用于存取指令)还是 ALUOut(用于存取数据)。

用一个 ALU 代替单周期数据通路中的三个 ALU，意味着这一个 ALU 必须能接收原来三个 ALU 的所有输入。处理这些输入使数据通路有以下变化：

1) 给 ALU 的第一个输入添加了一个多路复用器，它在寄存器 A 和 PC 间进行选择。

2) ALU 第二个输入的多路复用器由 2 路改为 4 路，其新加的两个输入为常数 4(用于 PC 增值)和符号扩展及移位后的偏移量字段(用于计算分支地址)。

图 5-26 详细画出了加上了这些多路复用器后的数据通路。通过引入一些寄存器和多路复用器得以将存储单元数目从 2 减为 1，并取消了 2 个加法器。由于寄存器和多路复用器都很小，这样做可以大大降低硬件成本。

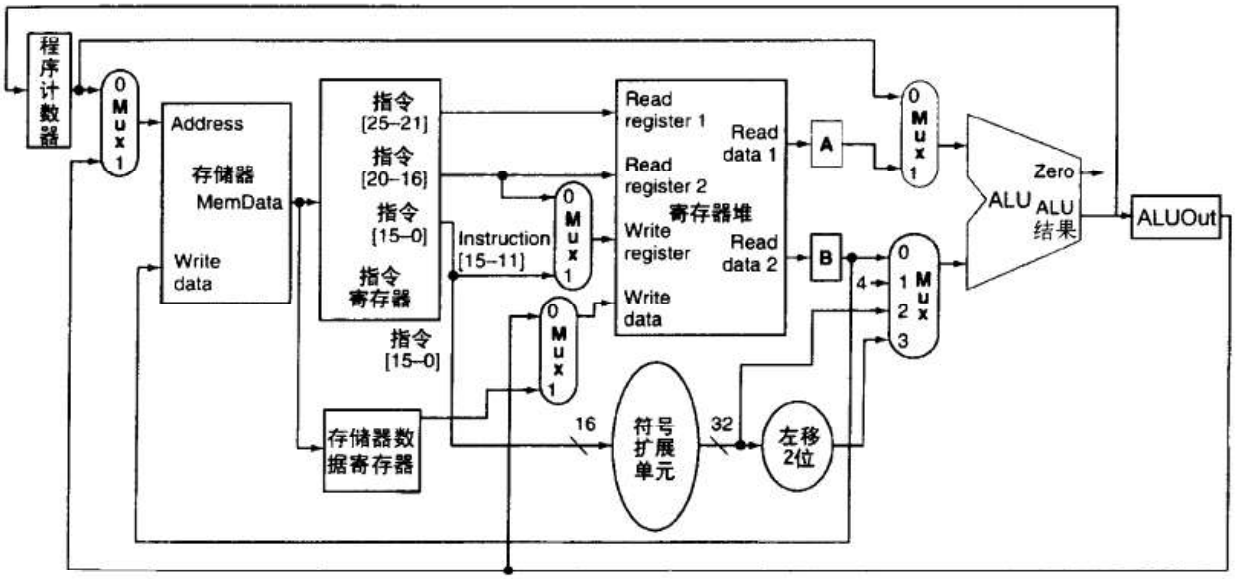

图 5-26 MIPS 的多周期数据通路处理基本指令

[虽然这条数据通路支持 PC 的普通增值，为了分支和跳转指令还需要增加一些连接和一个多路复用器；将在后面很快加入这些部件。比起单周期的数据通路，增加了一些寄存器(IR、MDR、A、B、ALUOut)，一个用于存储地址的多路复用器，一个用于 ALU 最高输入的多路复用器以及将 ALU 底部输入的多路复用器扩展为 4 个输入。这些小的增加可以去掉两个加法器和一个存储单元]

图 5-26 的数据通路用多个时钟周期完成一条指令，所以需要一套不同的控制信号。程序员可见的状态单元(PC、存储器和寄存器)和 IR 需要写控制信号。存储器还需一个读信号。单周期数据通路的 ALU 控制单元(参见图 5-13 和■附录 C)也可在此用于控制 ALU。最后，每个 2 输入的多路复用器都需要一根控制线，4 输入的多路复用器需要 2 根控制线。图 5-27 给出了图 5-26 中的数据通路加上这些控制线后的情况。

为支持分支和跳转指令，多周期数据通路还要添加部件；完成这些添加后，再看看指令的执行顺序如何确定，以及如何生成数据通路的控制。

对于跳转和分支指令，PC 有三种可能来源：

1) ALU 的输出，取指时为 PC+4 。这个数应直接写入 PC。

2) 寄存器 ALUOut，存储的是计算出的分支目标地址。

3) 指令寄存器(IR)的低 26 位左移 2 位与 PC 增值后的高 4 位相连，这是跳转指令时 PC 的来源。

在单周期控制的实现中可以看出，PC 的写入可以是条件的或无条件的。在普通的增值和跳转时，PC 的写入是无条件的。对于条件分支指令，只有在两个指定寄存器相等时 ALUOut 的值才会取代增值的 PC。所以，控制部分需两条独立的 PC 写控制信号，称为“PCWrite”和“PCWrite-Cond”。

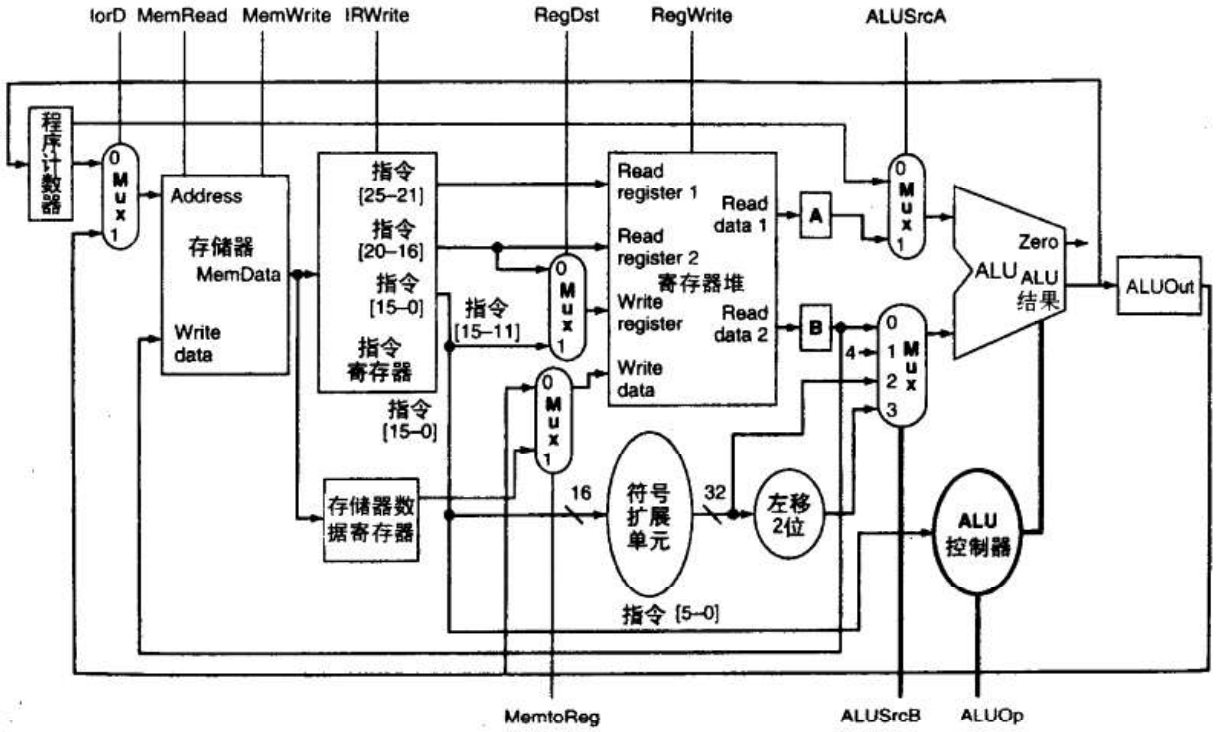

图 5-27 图 5-26 的数据通路加上控制线路

[ALUOp 和 ALUSrcB 信号为 2 位控制信号，其他控制信号为 1 位。寄存器 A 和 B 都不需要写控制信号，因为它们的内容只在写入的下一个周期读出。加入了存储数据寄存器，以便使取数时存储从存储器读出的数据。从存储器读出的数据不能直接写入寄存器堆，因为单时钟周期内不够访问存储器外加寄存器堆写。MemRead 信号被移到存储器单元的顶部以简化本图。分支指令的全部数据通路和控制线路将很快在后面加入]

这两个控制信号需要与 PC 写控制相连。和单周期数据通路一样，我们将使用一些门电路，从 PCWrite、PCWriteCond 和用于检验 beq 的两个寄存器操作数是否相等的 ALU 零输出信号生成 PC 写控制信号。为了决定在条件分支中 PC 是否被写入，将 ALU 的零输出信号与 PCWriteCond 相与，然后将与门的输出结果与非条件的 PC 写信号(PCWrite)相或。或门的输出直接作为 PC 的写控制信号。

图 5-28 为完整的多周期数据通路和控制单元，包括为实现更新 PC 而添加的控制信号和多路复用器。

在讨论各指令的执行步骤前，让我们先大致看看所有控制信号的作用(就像在图 5-16 一样)。图 5-29 说明了各个控制信号被设置为有效或无效状态时所起的作用。

细节：为了减少连接功能单元用的信号线，设计者可以使用*共享总线*。共享总线是连接多个单元的一组线；一般情况下，它有多个向总线传输数据的源，也有多个总线数据值的读者。就像减少数据通路的功能单元一样，可以通过共享总线来减少连接多个单元的总线数目。比如，ALU 有 6 个输入来源；然而任一时刻只需要其中两个。这时就可以用一对总线来传输送给 ALU 的数据值。设计者可以使用一根共享总线并保证任意时刻总线只有一个驱动源，而无须在 ALU 前面加一个大的多路复用器。虽然这样可以节省信号线，却需要同样数目的控制线路以控制总线传输什么数据。使用这样的总线结构的主要缺点是可能会对性能产生影响，因为总线不会像点对点连接那么快。

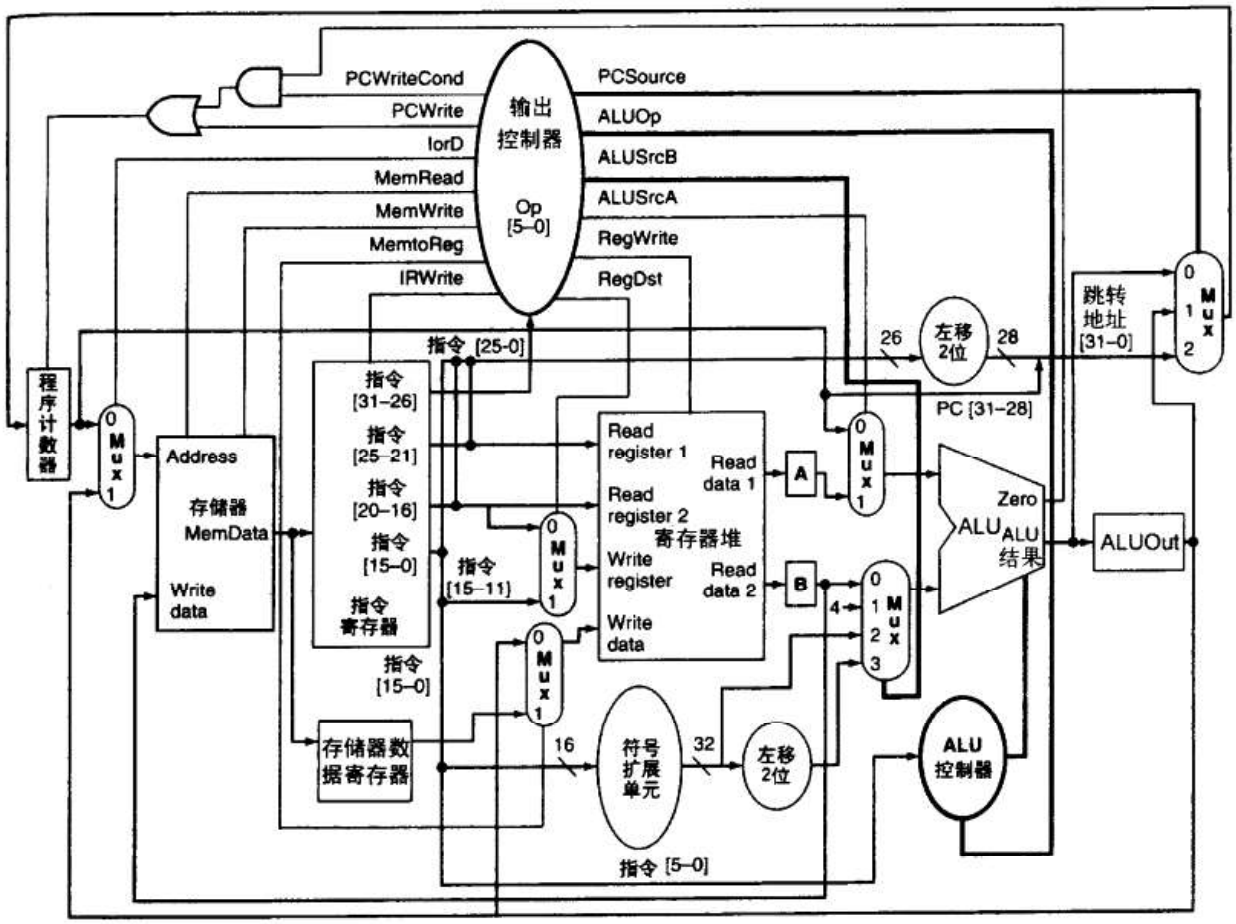

图 5-28 多周期实现的完整的数据通路和必要的控制线路

[图 5-27 的控制线路被加到控制单元，并且包含了改变 PC 所需的控制和数据通路单元。相对于图 5-27 而言，主要增加的包括一个用于选择新 PC 值来源的多路复用器(右上角)；用于合并 PC 写信号(左上角)的两个门；和控制信号 PCSource、PCWrite、PCWriteCond。PCWriteCond 信号用来判定条件分支是否成立。这也包括对跳转指令的支持]

1 位控制信号动作

信号名	无效时作用	有效时作用
RegDst	写寄存器在寄存器堆中的目的编号来自于 rt 字段	写寄存器在寄存器堆中的目的编号来自于 rd 字段
RegWrite	无	写数据输入的值写入到被写寄存器号选择的通用寄存器
ALUSrcA	ALU 第一个操作数是 PC	ALU 第一个操作数来自寄存器 A
MemRead	无	将地址输入指定位置的存储器内容放到存储器数据输出上
MemWrite	无	将地址输入指定位置的存储器内容替换为写数据输入的值
MemtoReg	送往寄存器堆写数据输入的值来自于 ALUOut	送往寄存器堆写数据输入的值来自于 MDR
IorD	PC 提供地址给内存单元	ALUOut 提供地址给存储器单元
IRWrite	无	存储器的输出写入 IR
PCWrite	无	写 PC；源由 PCSource 控制
PCWriteCond	无	如果 ALU 的 Zero 输出也被激活，写 PC

图 5-29 设置图 5-28 的控制信号所产生的动作

2位控制信号动作		
信号名	值(二进制)	作 用
ALUOp	00	ALU执行加操作
	01	ALU执行减操作
	10	指令的功能字段决定ALU操作
ALUSrcB	00	ALU的第二个输入来自寄存器B
	01	ALU的第二个输入是常数4
	10	ALU的第二个输入是经过符号扩展的IR的低16位
	11	ALU的第二个输入是经过符号扩展的IR的低16位左移2位的值
PCSource	00	ALU的输出(PC+4)送去写入PC
	01	ALUOut的内容(分支目标地址)送去写入PC
	10	跳转目标地址(IR[25: 0]左移2位后与PC+4[31: 28]连接)送去写入PC

图 5-29 (续)

[上表描述1位控制信号，下表描述2位控制信号。只有那些影响多路复用器的控制线路设为无效时有动作。这类似于图5-16的单周期数据通路，但增加了一些新的控制线路(IRWrite、PCWrite、PCWriteCond、ALUSrcB和PCSource)，去掉了一些不用的或被取代的线路(PCSrc、分支和跳转)]

5.5.1 将指令的执行分到各个时钟周期

图5-28给出了数据通路，现在应该看看多周期指令执行在每个时钟周期内的动作，因为这决定了可能需要附加什么控制信号，以及它们的设置。将指令的执行分解到各个时钟周期，目的是最大化性能。可以先把任意指令的执行分解为一系列步骤，每步一个时钟周期，基本上长度相当。比如，规定每一步最多只能包含一次ALU操作，或一次寄存器堆访问，或一次存储器访问。在此规定下，时钟周期应当至少长于这些操作中最长的操作所花费的时间。

前面讲过，每个时钟周期结束时，要在后面的周期中用到的所有数据必须存入一个寄存器，可以是一个主状态单元(如PC、寄存器堆或存储器)，一个在每个时钟周期写入的临时寄存器(如A、B、MDR或ALUOut)或一个有写控制的临时寄存器(如IR)。并且，由于我们的设计基于边缘触发型，可以继续读取寄存器的当前值；新的值直到下一个时钟周期到来时才可能出现。

在单周期数据通路中，每条指令的执行使用一套数据通路单元。许多数据单元顺序操作，以某个其他单元的输出作为自己的输入。一些数据通路单元并行操作；比如，PC的增值和指令的读取同时进行。多周期数据通路的情况类似。同一步内列出的所有操作在一个时钟周期内并行完成，随后各步的操作在不同周期内顺序完成。而一次ALU操作、一次存储器访问和一次寄存堆访问的限制也决定了一步内最多可以完成的内容。

注意：需要把读写PC或一个独立的寄存器与读写寄存器堆区分开来。前者的读写只占一个时钟周期的一部分，而寄存器堆的读写额外需要一个周期。此区别的原因是相对于单独的寄存器而言，寄存器堆有额外的控制和访问开销。为了缩短时钟周期，访问寄存器堆需要多个时钟周期完成。

可能的指令执行步骤及动作如下。每条MIPS指令需要其中3到5步：

步骤1：取指

从存储器中取出指令并计算下条指令的地址：

```
IR <= Memory[PC];
PC <= PC + 4;
```

操作：将 PC 作为地址传送给存储器，进行读取，将指令写入并存储在指令寄存器(IR)。同时，将 PC 加 4。我们使用 Verilog 符号"<=",此符号表明计算右边的式子然后赋值，它有效说明了时钟周期期间硬件执行的情况。

为实现这一步，需要将控制信号 MemRead 和 IRWrite 设为有效，将 IorD 置 0 以选择 PC 作为地址来源。为实现 PC 加 4，要将 ALUSrcA 信号置 0(将 PC 送往 ALU)，ALUSrcB 信号置 01(将 4 送往 ALU)，ALUOp 置 00(使 ALU 进行加法)。最后，还要将 PC 源设置为 00 并且设置 PCWrite，以把增值后的指令地址存入 PC。PC 的增值和指令存储器的访问可以并行进行。新 PC 值直到下个时钟周期才可见。(增值后的 PC 也要存入 ALUOut，但这个动作没有什么影响。)

步骤 2：指令译码和读取寄存器

在上一步和这一步，还不知道指令的具体内容是什么，所以只能进行那些针对所有指令都需要进行的操作(如第一步中的取指)或没有坏影响的操作。于是，在这一步中可以读取指令字段 rs 和 rt 指定的寄存器的内容，因为即使不必要这样做也没有什么害处。从寄存器堆读出的数据值可能在后面的步骤中要用到，所以我们将它们以寄存器堆中读出并将它们存入临时寄存器 A 和 B 中。

还要用 ALU 计算分支目标地址，这样做也没有什么害处。因为若发现指令不是分支指令则忽略计算结果即可。计算出的分支目标地址存于 ALUOut 中。

尽早执行这些"良性"操作的好处是减少指令执行所用的时钟周期数。指令格式的规整便于这些"良性"操作的尽早完成。比如，若指令有两个寄存器输入，则它们总是位于 rs 和 rt 字段；若指令是分支指令，则其偏移量总是低 16 位：

```
A <= Reg[IR[25:21]];
B <= Reg[IR[20:16]];
ALUOut <= PC + (sign-extend (IR[15-0]) << 2);
```

操作：访问寄存器堆，读取寄存器 rs 和 rt 的内容并将结果存入寄存器 A 和 B。由于寄存器 A 和 B 在每个时钟周期都被重新写入，所以可以在每个时钟周期读出寄存器堆的内容存入寄存器 A、B。这一步中还计算分支目标地址并存入 ALUOut，若指令为分支指令则下一个时钟周期将用到此 ALUOut。这个操作要求给 ALUSrcA 置 0(以将 PC 送入 ALU)，ALUSrcB 置 11(以将符号扩展并移位后的偏移量送入 ALU)，ALUOp 置 00 (以使 ALU 作加法)。读取寄存器堆和计算分支目标地址并行执行。

在这个时钟周期以后，就可以根据指令内容来决定要进行的具体操作了。

步骤 3：指令的执行，存储地址的计算或分支的完成

这是数据通路操作中由指令类型决定的第一个时钟周期。在所有情况下，ALU 对前面准备好的操作数进行操作，根据指令类型执行四种功能之一种。根据指令的类型说明要进行的操作：

存储器访问：

```
ALUOut <= A + sign-extend (IR[15:0]);
```

操作：ALU 将操作数相加，得到存储地址。这要求将 ALUSrcA 置 1(使 ALU 的第一个输入为寄存器 A)，ALUSrcB 置 10(使符号扩展单元的输出作为 ALU 的第二个输入)。需将 ALUOp 信号置为 00(以使 ALU 作加法运算)。

算术逻辑指令(R 型)：

```
ALUOut <= A op B;
```

操作：ALU 对前面周期中从寄存器堆读出的数据进行功能码所指定的操作。这要求将 ALUSrcA 置 1，ALUSrcB 置 00(使寄存器 A 和 B 作为 ALU 的输入)。ALUOp 信号需置为 10(以由

功能字段决定 ALU 控制信号的设置)。

分支:

```
if (A == B) PC <= ALUOut;
```

操作:ALU 将前面步骤中从寄存器堆读出的两寄存器进行相等比较。ALU 的零信号输出用来决定是否分支。这要求将 ALUSrcA 置 1,ALUSrcB 置 00(使寄存器堆的输出成为 ALU 的输入)。ALUOp 信号需置为 01(使 ALU 做减法)以进行等值的测试。若 ALU 的零输出信号有效,则需将 PCWriteCond 信号置有效,以修改 PC。通过将 PCSource 置 01,写入 PC 的内容将来自 ALUOut,而 ALUOut 存储的正是前面周期中计算出来的分支目标地址。对于条件分支指令,实际上有两次写 PC:一次来自 ALU 的输出(在指令译码/读取寄存器时),一次来自 ALUOut(在分支完成时)。最后写入 PC 的内容用于下一次取指。

跳转:

```
# {x, y}是字段 x 和 y 连接的 Verilog 标记
PC <= {PC [31:28], (IR[25:0]],2'b00)};
```

操作:PC 由跳转目的地址取代。PCSource 设置为指示将跳转地址直接作为 PC 的输入,PCWrite 置为有效,以将跳转地址写入 PC。

步骤 4:存储器的访问或 R 型指令的完成

在这一步中,存或取指令访问存储器,算术逻辑指令写结果。当数据值从存储器中取出后,被存入存储器的数据寄存器 (MDR),它将用于下一个时钟周期。

存储访问指令:

```
MDR <= Memory [ALUOut];
```

或

```
Memory [ALUOut] <= B;
```

操作:若为取数指令,从存储器中读出一个数据字并写入 MDR。若为存数指令,则将数据写回存储器。在这两种情况下,所用的存储器地址都是在前面的步骤中计算出并存于 ALUOut 中的地址。对于存数指令,源操作数存于寄存器 B 中。(实际上 B 被读取了两次,一次在第 2 步,一次在第 3 步。幸好,两次读取的数据值相同,因为寄存器号——存于 IR 中,用于读取寄存器堆——没有变。)信号 MemRead(用于取数)或 MemWrite(用于存数)将置为有效。另外,对于存取指令,信号 IorD 被置为 1,以强迫存储器地址来自 ALU 而非 PC。由于 MDR 在每个时钟周期被写入,不需设置另外的控制信号。

算术逻辑指令(R 型):

```
Reg[IR[15:11]] <= ALUOut;
```

操作:将 ALUOut 的内容,即前面周期中的 ALU 的操作结果,写入结果寄存器。信号 RegDst 必须置为 1,以使 rd 字段(15: 11 位)在寄存器堆中用于选择要写入的寄存器。RegWrite 必须有效,同时 MemtoReg 必须为 0,以使 ALU 的输出而非存储器的数据输出被写入。

步骤 5:读取存储器完成

在这一步中,写入来自存储器的值,完成取数指令。

取数指令:

```
Reg[IR[20:16]]<= MDR;
```

操作:将在前面周期中存入 MDR 的数据写入寄存器堆,为此,将 MemtoReg 置 1(以写入来自存储器的结果),使 RegWrite 有效(以引起写操作),将 RegDst 置 0 以选择 rt(20:16 位)字段为

寄存器号。

以上五步概括如图 5-30。这五步可以决定每个时钟周期里控制信号该做些什么。

步 骤 名	R 型指令动作	存储器访问指令动作	分 支 动 作	跳 转 动作
取指	IR <= Memory[PC] PC <= PC + 4			
指令译码/取寄存器	A <= Reg[IR[25:21]] B <= Reg[IR[20:16]] ALUOut <= PC + (sign-extend(IR[15:0]) << 2)			
执行、地址计算、分支/跳转完成	ALUOut <= A op B	ALUOut <= A + sign-extend (IR[15:0])	if (A==B) PC <= ALUOut	PC <= {PC[31:28], (IR[25:0],2'b00)}
存储器访问或 R 型指令完成	Reg[IR[15:11]]<= ALUOut	Load: MDR <= Memory [ALUOut] or Store: Memory[ALUOut] <= B		
存储器读操作完成		Load: Reg[IR[20:16]] <= MDR		

图 5-30 所有类型指令执行步骤的总结

[指令分 3 到 5 步执行。前两步与指令类型无关。之后，指令根据类型不同，还需要 1～3 步完成。存储器访问或存储器读完成步骤为空的项说明有些指令需要较少的周期完成。在一个多周期实现中，当前指令一完成下一条指令就马上开始，所以这些周期不会空闲或浪费。如前所述，实际上每个周期都读寄存器堆，但是只要 IR 不变，从寄存器堆读出的内容就是一样的。特别地，对于分支或 R 型指令，在指令解码阶段读入寄存器 B 的值，与在执行阶段写入 B 的值，以及随后存字指令中存储器访问阶段使用的值都是一样的]

5.5.2 控制单元的定义

现在已经确定了控制信号的内容及设置的时间，就可以实现控制单元了。为单周期数据通路设计控制单元，使用了一组真值表，根据指令类型确定控制信号的设置。而多周期数据通路的控制更为复杂，因为其指令是分一系列步骤执行的。多周期数据通路的控制部分必须确定每一步及下一步要设置的信号。

在本小节及■ 5.7 节中，将研究两种不同的控制技术。第一种技术基于有限状态机，通常以图形表示。第二种技术称为**微程序**[⊖]，通过编程实现控制。这两种描述控制的形式都允许其具体实现——门电路、ROM、PLA 的使用由 CAD 系统合成。在本章中，将集中讨论这两种方式下控制的设计及描述。

■ 5.8 节给出了如何使用硬件设计语言设计现代处理器，并且包括多周期数据通路和有限状态控制的例子。在现代数字系统设计中，将硬件描述变为实际门电路的最后步骤由逻辑和数据通路合成工具处理。附录 C 中通过将多周期控制单元转化为详细硬件实现过程显示了处理器如何工作。无需阅读■ 5.8 节和■附录 C，就可以通过本节掌握控制的关键思想。但是，如果想要进行实际的硬件设计，5.9 节是有用的，并且■附录 C 给出了门电路级的具体实现。

根据这个实现方案和每个状态需要一个时钟周期的知识，可以计算出一套典型混合指令的 CPI。

例题 **多周期 CPU 的 CPI**

使用图 3-26 所示的 SPECINT2000 混合指令，假设每个多周期 CPU 状态需一个时钟周期，CPI 为多少？

⊖ 微程序(microprogram) 将控制用符号化形式表示为一组微指令，并在一个简单的微机器上执行。

解 混合指令为25%取数据字(1%取字节+24%取字)，10%存数据字(1%存字节+9%存字)，11%分支(6%beq，5%bne)，2%跳转(1%jal+1%jr)，52%ALU(假定所有剩下的混合指令都是ALU指令)。从图5-30可得，各指令类型所需时钟周期数为：

- 取数据字：5
- 存数据字：4
- ALU指令：4
- 分支：3
- 跳转：3

CPI计算如下：

$$\text{CPI} = \frac{\text{CPU 时钟周期数}}{\text{指令数}} = \frac{\sum(\text{指令数}_i \times \text{CPI}_i)}{\text{指令数}}$$

$$= \sum \frac{\text{指令数}_i}{\text{指令数}} \times \text{CPI}_i$$

比值

$$\frac{\text{指令数}_i}{\text{指令数}}$$

即为指令类型 i 的指令频度。于是可得

$$\text{CPI} = 0.25 \times 5 + 0.10 \times 4 + 0.52 \times 4 + 0.11 \times 3 + 0.02 \times 3 = 4.12$$

这个CPI要优于所有指令使用相同周期数5.0时的最坏CPI。当然，设计开销可能会降低或增加这种差异。多周期设计可能也有更好的性价比，因为在数据通路中需要更少的独立组件。■

描述多周期控制的第一种方法是**有限状态机**[⊖]。一个有限状态机由一组状态及状态间的转换规则组成。转换规则由一个**后继状态函数**[⊖]确定，它将现有状态和输入映射到一个新的状态。当用一个有限状态机描述控制时，每个状态还表示一组输出，当机器在此状态时，这些输出有效。有限状态机的实现方式一般假设所有的输出都不会被明确地设为有效或无效。同样地，当一个信号没有被明确地设为有效时，就是无效状态，而非无关状态，这是数据通路正常运作的前提。比如，仅当寄存器堆中某一项要被写入时，RegWrite信号才应是有效的；当它未被明确设为有效时，一定是无效的。

多路复用器的控制稍有不同，因为它们选择一个为0或为1的输入。这样，在有限状态机中，经常要确定好所关心的多路复用器的控制。当逻辑实现有限状态机时，可将控制置0设为默认值，这样它们不需任何门电路。在■附录B中有一个简单的有限状态机的例子，如果你对有限状态机的概念不甚熟悉，可在继续学习之前仔细研读■附录B。

有限状态机的控制基本上对应于5.5.1节的五个执行步骤；有限状态机的每一个状态占用一个时钟周期。有限状态机由几个部分组成。由于所有指令的前两个执行步骤都一样，所有指令的有限状态机的前两个状态也都一样。第3～5步不一样，具体由操作码决定。在某类型指令的最后一步执行完后，有限状态机将回到原始状态，开始取下一条指令。

图5-31抽象地表示了一个有限状态机。为了进一步研究有限状态机的细节，首先要扩充取指令和解码部分，然后介绍不同指令类型各自对应的状态(和动作)。

⊖ 有限状态机(finite state machine) 一个逻辑函数序列由下列元素构成：一组输入，输出，将当前状态和输入映射到一个新状态的下一状态函数，以及将当前状态和可能的输入映射为一组有效输出的输出函数。

⊖ 后继状态函数(next-state function) 一个组合函数：给定输入和当前状态，决定有限状态机的下一个状态。

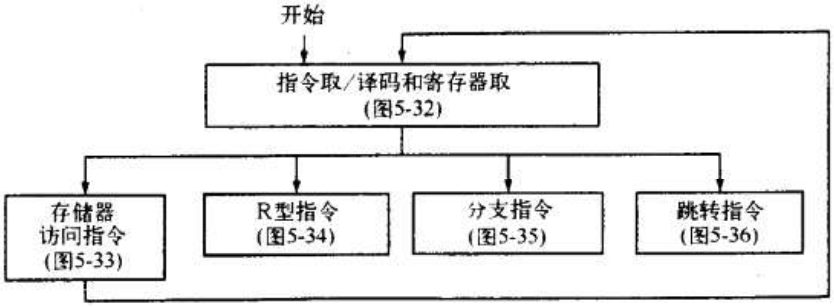

图 5-31 有限状态机控制的高层视图

[前面的步骤与指令类型无关；然后进行一系列由指令的操作码决定的操作，以完成各种类型指令的执行。完成该类型指令所需操作后，控制转回取新的指令。这张图中的每一个方框表示一个或多个状态。标有开始的弧线指出要取第一条指令开始时的状态]

图 5-32 以传统的图形方式表示了有限状态机的前两个状态。为便于说明，给各状态随机标号。0 状态对应第一步，是有限状态机的初始状态。

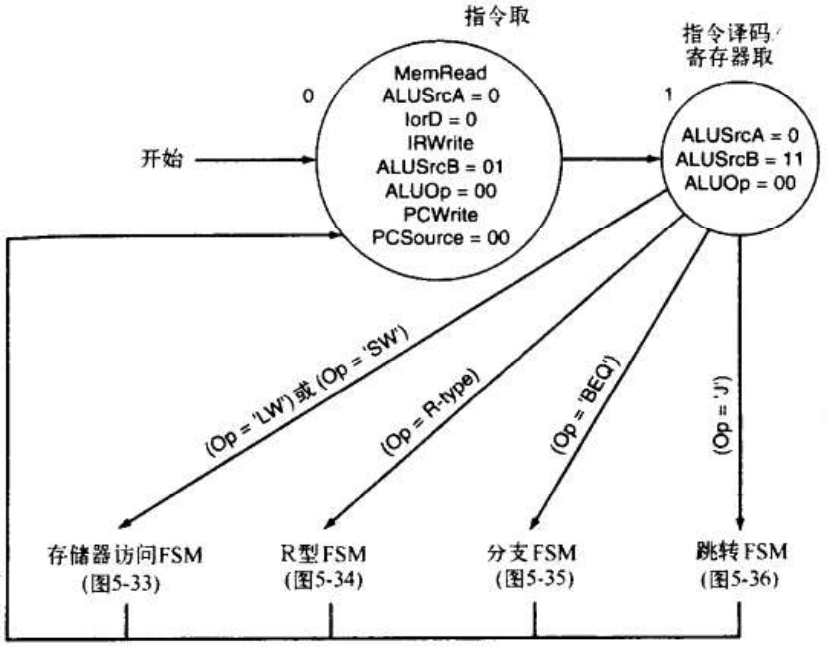

图 5-32 所有指令的取指和译码部分都是相同的

[这些状态对应于图 5-31 的概括的抽象有限状态机的上面那个方框。在第一个状态，激活两个信号(MemRead 和 IRWrite)使存储器读一条指令并将它写入指令寄存器，并设置 IorD 为 0 以选择 PC 为地址来源。设置信号 ALUSrcA、ALUSrcB、ALUOp、PCWrite、PCSource 以计算 PC + 4 并存入 PC。(它也会存入 ALUOut，但不会从那里被使用。)在下一状态，将 ALUSrcB 设为 11(使低 16 位符号扩展并移位后的 IR 送入 ALU)，ALUSrcA 设为 0，ALUOp 设为 00，以计算分支目标地址；将结果存于每个周期都写入的 ALUOut 寄存器中。根据在当前状态中得知的指令类型，在 4 个后续状态中进行选择。控制单元的输入称为 Op，用于决定选择哪条弧线。请记住，所有不是显式标明为有效的信号都是无效的；这点对于控制写操作的信号特别重要。对于多路复用器控制，没有特别设置的表示无关多路复用器的设置]

各状态中有效的信号在表示该状态的圆圈中写出。状态之间的弧线指向下一个状态，当有多个可能的下一状态时，弧线上标注了选择此下一状态的条件。在状态 1 之后，有效的信号取决于指令类型。所以，有限状态机从状态 1 伸出 4 条弧线，对应于 4 个指令类型：存储访问、R 型、等

值分支和跳转。这个根据指令向不同状态分支的过程称为解码，因为后继状态的选择，即下面的动作，由指令类型决定。

图 5-33 给出了有限状态机用于实现存储访问指令的部分。对于这种指令，取指或读取寄存器后的第一个状态计算存储地址(状态 2)。为计算存储地址，ALU 的输入多路复用器需设置为接受寄存器 A 为第一个输入，符号扩展后的偏移量字段为第二个输入；其结果写入 ALUOut 寄存器中。存储地址计算之后，应读/写存储器；这需要两个不同的状态。若指令的操作码为 lw，则状态 3(对应于存储访问步骤)进行存储器读(MemRead 被激活)。存储器的输出通常写入 MDR。若是 sw，状态 5 进行存储器写(MemWrite 被激活)。在状态 3 和 5 中，IorD 信号被设为 1 以保证存储器地址来自 ALU(存数指令并不需要这样，因为写地址使用存储器的不同输入)。写存储后，sw 指令执行完毕，后继状态为状态 0。若是取数指令，则需要另一状态(状态 4)将存储器的结果写入寄存器堆。设置多路复用器控制信号 MemtoReg = 1 和 RegDst = 0 将使从 MDR 中取出的数值写入寄存器堆，并以 rt 作为寄存器号。这一步对应于存储器读完成步骤，其下一状态为状态 0。

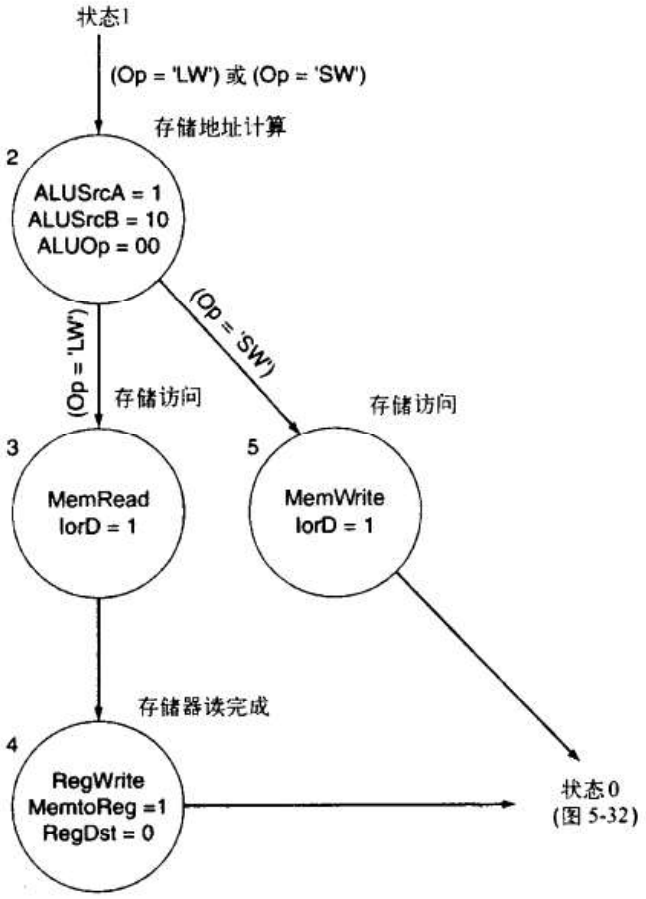

图 5-33 控制存储访问指令的有限状态机有四个状态

[这些状态对应于图 5-31 中标有"存储访问指令"的方框。计算存储地址之后，存和取指令分别需要一个序列。信号 ALUSrcA、ALUSrcB 和 ALUOp 的设置是为了在状态 2 进行存储地址的计算。取指令需要多一个状态以将来自 MDR 的结果(结果在状态 3 时写入)写入寄存器堆]

为实现 R 型指令，需要对应于步骤 3(执行)和步骤 4(R 型指令完成)的两个状态。图 5-34 给出了有限状态机的两个状态部分。状态 6 激活 ALUSrcA 并置 ALUSrcB 为 00；这使得从寄存器堆中读取的两个寄存器作为 ALU 的输入。置 ALUOp 为 10 使 ALU 控制单元根据指令的功能字段

设置 ALU 的控制信号。在状态 7 中，RegWrite 被激活以写寄存器堆，RegDst 被激活以使 rd 字段作为目的寄存器号，MemtoReg 无效以选择 ALUOut 作为源值写入寄存器堆。

对于分支指令，只需多加一个状态，因为指令执行在第 3 步就完成了。在这一状态中，必须设置控制信号使 ALU 对寄存器 A 和 B 的内容做比较，也要设置信号使 ALUOut 寄存器中的内容有条件地写入 PC。为进行比较，需激活 ALUSrcA 并置 ALUSrcB 为 00，置 ALUOp 为 01(强制进行减法)。(只使用 ALU 的零输出信号，而不用减法的结果。)为控制 PC 的写入，需激活 PCWrite-Cond 并置 PCSource=01，这样若 ALU 的零输出信号有效，ALUOut 寄存器的内容(为状态 1 计算出的分支目标地址，见图 5-32)将写入 PC。图 5-35 给出这一状态。

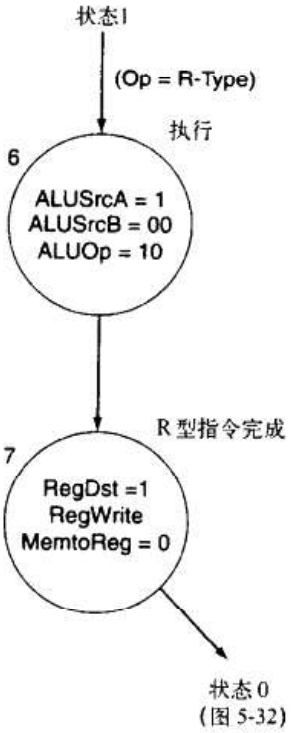

图 5-34 R 型指令可以由简单的两个状态的有限状态机实现

[这些状态对应于图 5-31 中标有"R 型指令"的方框。第一个状态使 ALU 进行操作，第二个状态使 ALU 的结果(在 ALUOut 中)写入寄存器堆。在状态 7 激活的三个信号使 ALUOut 的内容写入寄存器堆中由指令寄存器的 rd 字段标识的寄存器中]

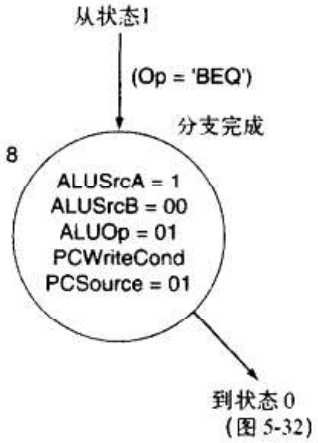

图 5-35 分支指令只需要一个状态

[被激活的前三个输出(ALUSrcA、ALUSrcB 和 ALUOp)使 ALU 对寄存器进行比较，若分支条件为真，PCSource 和 PCWriteCond 信号进行条件写。注意不使用写入 ALUOut 的值；而只用 ALU 的零输出信号。分支目标地址从 ALU-Out 得到，它在第一个状态结束时存入 ALUOut 中]

最后一类是跳转指令；同分支指令一样，它也只需一个状态(如图 5-36 所示)完成其执行。在这一状态中，PCWrite 信号被激活以写 PC。通过置 PCSource 为 10，所写的内容将为指令寄存器的低 26 位加上最低两位 0 和高 4 位 PC 值。

现在，我们可以将有限状态机的这些部分合在一起，组成控制单元如图 5-38 所示。在每一状态中写出了被激活的信号。后继状态取决于指令的操作码位，所以弧线上标出相应的指令操作码。

一个有限状态机可由一个包含当前状态的临时寄存器和一个决定数据通路信号设置和下一状态的组合逻辑模块实现。图 5-37 给出了这种实现可能的样子。■附录 C 详细描述了用这种结

构实现的有限状态机。在■C.3 节，图 5-38 的有限状态机的组合控制逻辑分别由一个 ROM(只读存储器)和一个 PLA(可编程逻辑阵列)实现。(■附录 B 中有对这些逻辑单元的描述。)在本章的下一部分，将以另外的方式描述控制。两种技术只是同样控制信息的不同表示方式。

第 6 章中主要内容流水线几乎总被用来加速指令的执行。对于简单指令，流水线能够比多周期设计获得更高时钟频率和比单周期设计获得更高单周期 CPI。但是，大多数流水线处理器中，某些指令的执行比单个时钟周期长，需要多时钟周期控制。浮点指令就是普遍的例子。在 IA-32 体系结构中有许多需要使用多周期控制的例子。

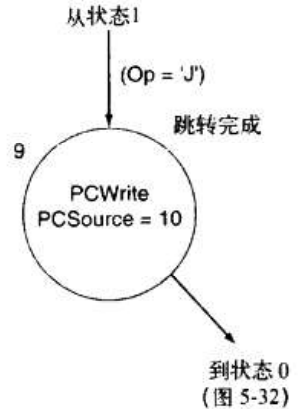

图 5-36 跳转指令只需要一个状态，它激活两个控制信号，以将指令寄存器的低 26 位左移 2 位并与本指令 PC 的高 4 位相连后写入 PC

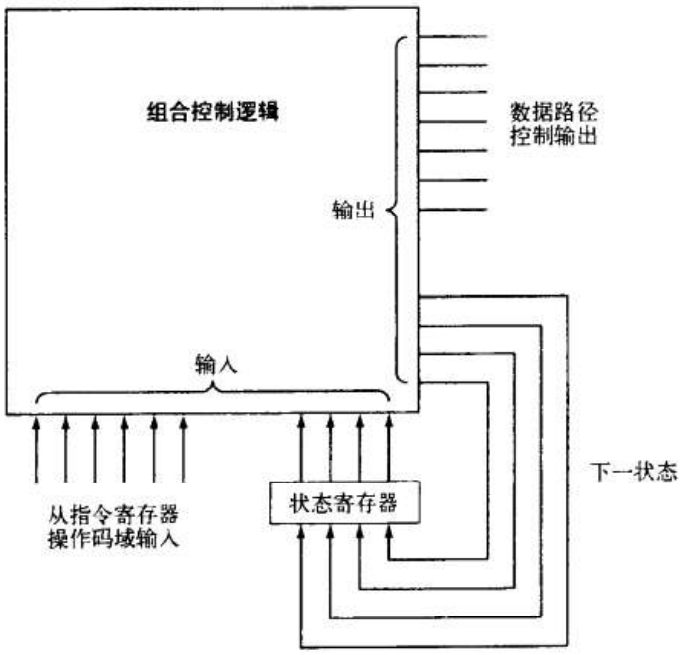

图 5-37 有限状态机的控制器通常由一个组合逻辑模块和一个保存当前状态的寄存器组成

[组合逻辑的输出为下一状态号和在当前状态中应该激活的控制信号。组合逻辑的输入为当前状态和所有可能决定后继状态的输入信号，在当前情况下，输入即为指令寄存器的操作码位。注意在本章使用的有限状态机中，输出仅取决于当前状态，而不取决于输入信号。下面的细节部分对此进行了更详细的解释]

细节： 图 5-37 的有限状态机称为 Moore 机，以 Edward Moore 命名。其显著特点是其输出只依赖于当前状态。对一个 Moore 机，标有组合控制逻辑的模块可分为两部分。一部分有控制输出和只有状态输入，另一部分只有后继状态输出。

另一种有限状态机称为 Mealy 机，以 George Mealy 命名。Mealy 机允许输入和当前状态共同决定输出。Moore 机在速度和控制单元大小方面有潜在的实现优势。速度优势是因为在时钟周期早期需要的控制输出与输入无关，只取决于当前状态。在■附录 C

中，当有限状态机的实现具体到逻辑门一级时，大小优势便显而易见。Moore 机的潜在缺陷是它可能需要额外的状态。比如，当两个状态序列间只有一个状态不同时，Mealy 机可通过使输出取决于输入而将状态合并。

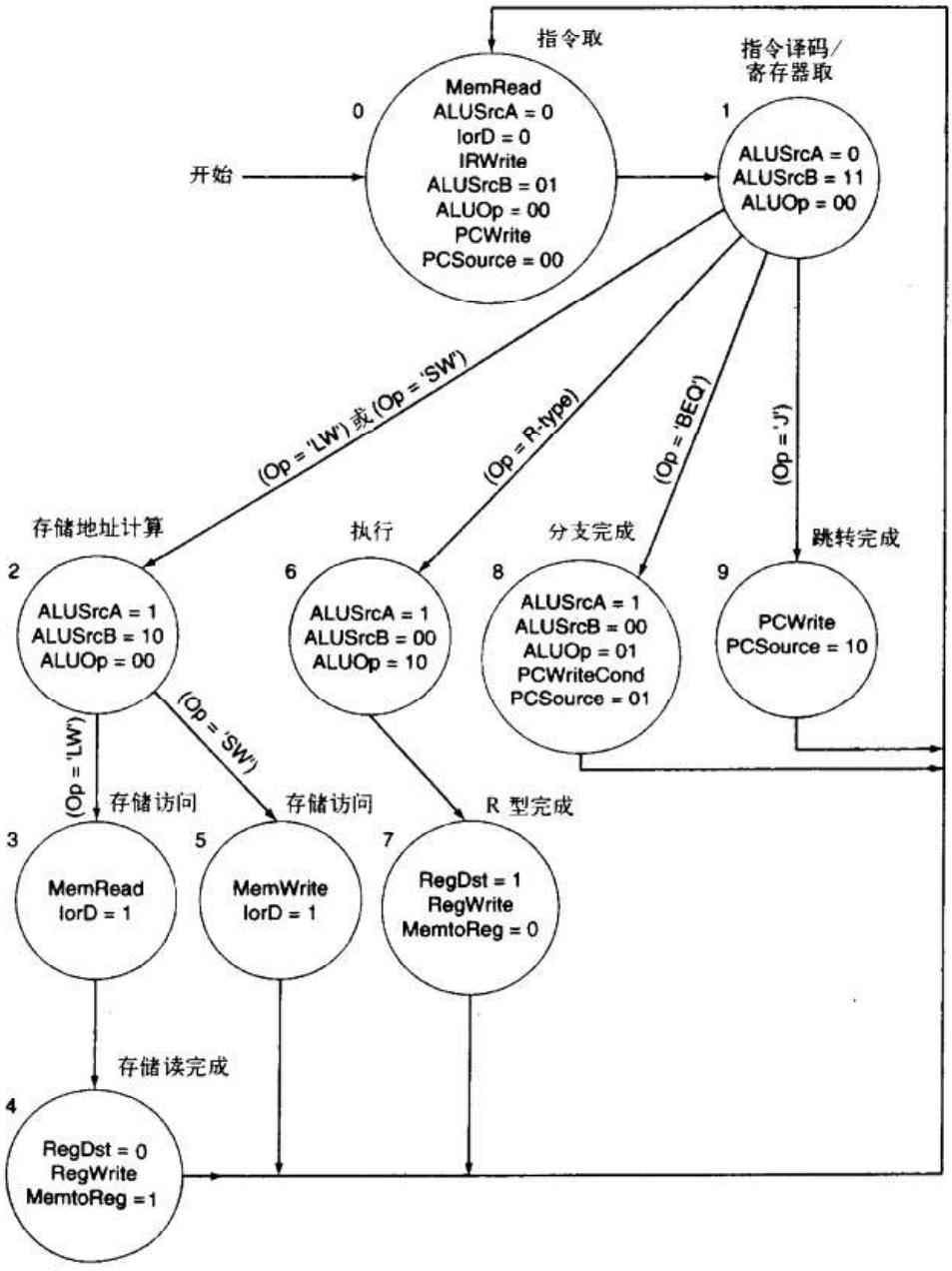

图 5-38 图 5-28 中数据通路的完整的有限状态机控制

[弧线上的标注是在确定后继状态时进行测试的条件；当后继状态是无条件的时候没有标注。结点中的标注为该状态中激活的输出信号；如果正确的操作要求，通常就要指明多路复用器控制信号的设置。这样，在某些状态中一个多路复用器的控制可能设为 0]

理解程序性能

对于给定时钟频率的处理器，两个代码段的相对性能由 CPI 的结果和执行每个代码段的指令

数决定。正如这里所见，即使简单的处理器，不同指令也可能有不同的 CPI。在随后两章中，引入流水线和高速缓存后不同指令的 CPI 可能会更不同。虽然硬件设计者可以控制许多影响 CPI 的因素，但是程序员、编译器和软件系统决定了最终执行哪些指令，从而决定了程序的有效 CPI 是多少。程序员要想提高性能必须理解 CPI 的作用和影响 CPI 的因素。

自测

1. 对错与否：由于跳转指令不依赖寄存器的值或正在计算的分支目标地址，因此可以在第二个状态间完成分支，而不是等到第三个状态。

2. 对错与否或者是否可能：能否使用 PCSource[0]取代 PCWriteCond 控制信号。

5.6 异常

控制设计是处理器设计中最具挑战性的一个方面：它最难达到正确，也最难以提高速度。控制中最难的部分之一是实现**异常**[⊖]和**中断**[⊖]——除分支、跳转以外改变正常指令执行顺序的事件。一个异常是一个来自处理器内部的意外事件；算术溢出就是一个异常的实例。一个中断也是一件导致控制流意外改变的事件，但它来自处理器外部。在第 8 章中将会看到，I/O 部件通过中断与处理器进行通信。

许多体系结构和著作者不区分中断和异常，通常用老术语中断概括这两种事件。遵循 MIPS 的习惯，术语异常指控制流中任何意外的改变，而无论其产生原因是来自处理器内部还是外部；术语中断则只用于由外部引起的事件。Intel IA-32 体系用中断指代这两种意外事件。

中断本来是为处理如算术溢出之类的意外事件和指示 I/O 部件请求服务而创建的。同样的基本机制被扩展，也用于处理内部产生的异常。这里有一些例子，说明某些情况是处理器内部还是外部产生的：

事件类型	来 源	MIPS 术语
I/O 部件的请求	外部	中断
从用户程序调用操作系统	内部	异常
算术溢出	内部	异常
使用未定义的指令	内部	异常
硬件错误	两者都可能	异常或中断

许多对异常支持的要求都来自导致意外发生的特殊情况。在第 7 章讨论存储级别和第 8 章讨论 I/O 时，将回到这个话题，以能更清楚地理解要求额外的异常机制的动机。在这一节中讨论了检测两种异常的控制实现，这两种异常由我们前面已经涉及的指令集部分和其实现方式所产生。

检测异常条件并采取适当举措，这经常处于一个机器的关键时间路径上，这条路径决定了时钟周期的长度，即机器性能。如果在控制单元的设计中没有对异常的充分考虑，那么在复杂的实现中加入异常支持可能导致性能的明显下降，并且使正确的设计变得更为困难。

5.6.1 异常是怎样处理的

这里的实现中可能产生的两种异常是未定义指令的执行和算术溢出。异常发生时机器必须

⊖ 异常(exception) 也称中断。是一个打断程序运行的突发事件；它被用来检测溢出等。

⊖ 中断(interrupt) 来自处理器外的异常。(有些体系结构中也用中断一词表示所有的异常现象。)

进行的基本操作是：在异常程序计数器(EPC)中保存出错指令的地址，并把控制权转交给某特定地址处的操作系统。

操作系统可采取适当的行动，如给用户程序提供一些服务，对溢出情况进行事先定义好的操作，或者终止程序的执行并报告错误。在完成处理异常所需动作后，操作系统可以终止程序，也可以继续程序的执行，此时由 EPC 决定重新开始执行的地方。在第 7 章将更详细讨论重新开始执行的问题。

为处理异常，操作系统必须知道引起异常的是哪条指令，和异常的原因。主要有两种方法用于表示产生异常的原因。MIPS 体系使用的方法是设置一个状态寄存器(称为*原因寄存器*(cause register))，其中有一个字段用于记录异常产生的原因。

另一种方法是使用**向量中断**[⊖]。在向量中断中，控制权被转移到由异常原因决定的地址处。例如，为处理前面的两种异常，可如下定义两种异常向量地址：

异常类型	异常向量地址(十六进制)
未定义的指令	$C000\ 0000_{hex}$
算术溢出	$C000\ 0020_{hex}$

操作系统通过初始地址可以知道产生异常的原因。地址由 32 个字节即 8 条指令表示，操作系统必须记录异常起因，并可以在这个指令序列之内进行一些有限的处理。当异常不是向量型的，所有异常使用同一入口点，操作系统通过解码状态寄存器以寻找异常的原因。

通过给基本实现加上一些额外的寄存器和控制信号，再稍微扩充一下有限状态机，就可以进行对异常的处理。如果要实现的是 MIPS 体系结构使用的异常系统。(实现异常向量也不难。)这需要给数据通路加上两个附加寄存器：

- EPC：一个 32 位寄存器，用于存储受影响的指令地址。(向量式异常也需要这样一个寄存器。)
- Cause：一个记录异常原因的寄存器。MIPS 体系结构中这个寄存器是 32 位的，虽然其中一些位现在还没有用。假定用寄存器的低位对前面提到的两种异常的可能原因进行编码：未定义指令 = 0，算术溢出 = 1。

需要加入两个控制信号称为 EPCWrite 和 CauseWrite，使 EPC 和 Cause 寄存器被写入。另外，需要一位控制信号 IntCause 对 Cause 寄存器的低位进行正确的设置；最后，需要将异常地址，即操作系统处理异常的入口点，写入 PC；在 MIPS 体系中地址为 $8000\ 0180_{hex}$(MIPS 的 SPIM 模拟器使用 $8000\ 0080_{hex}$)。现在，在 PC 的入口接有一个由 PCSource 信号(参见图 5-28)控制的三路复用器。也可以改用一个四路复用器，另一个输入恒为 $8000\ 0180_{hex}$。当 PCSource 为 11_{two}时就选择了这个值写入 PC。

由于在每条指令的第一个周期 PC 增值，所以不能将 PC 值写入 EPC，这时 PC 值为指令地址加 4。然而可以用 ALU 将 PC 减 4，再把 ALU 的输出写入 EPC。这并不需要额外的控制信号或通路，因为可以用 ALU 做减法，且常数 4 已经是 ALU 的一个可选的输入。所以 EPC 的写数据端口与 ALU 的输出相连。图 5-39 画出了加上实现异常处理后的多周期数据通路。

用图 5-39 所示数据通路，处理各种异常所需的操作各占一个状态。对每种情况，该状态设置 Cause(原因)寄存器，计算并存储原始 PC 的值至 EPC 寄存器，而将处理异常的地址写入 PC。所

⊖ 向量中断(vectored interrupt) 用来进行控制转换的中断地址根据例外的原因决定。

以，为处理这里考虑的两种异常，只需加入两个状态。在加入这两个状态前必须决定如何检查异常，因为这些检查控制连到新状态的弧。

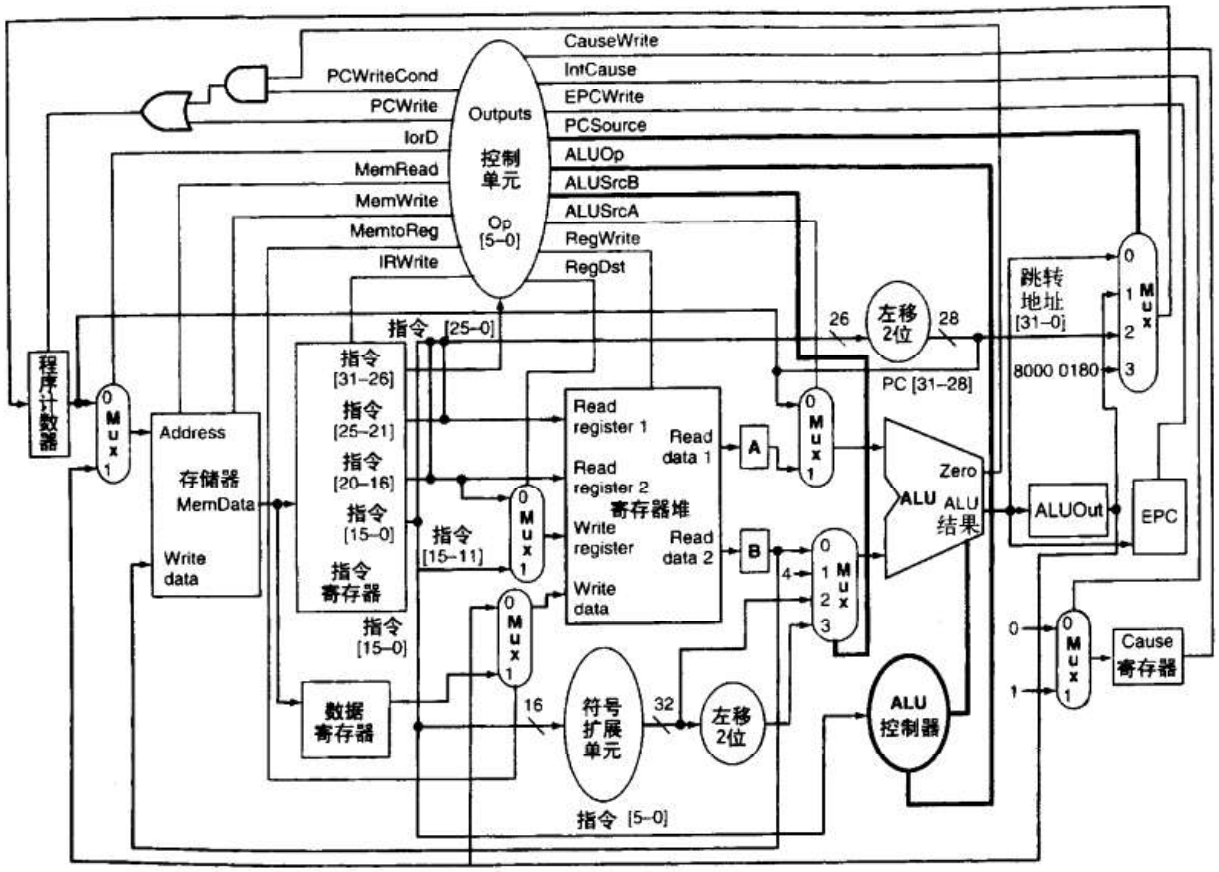

图 5-39 加入了异常处理的多周期数据通路

[所作的增加包括 Cause 和 EPC 寄存器，一个控制 Cause 寄存器输入的多路复用器，一个控制写入 PC 的值的多路复用器的扩展，和用于所加的多路复用器和寄存器的控制线路。为了简化起见，本图中没有显示 ALU 溢出信号，这个信号需要存储在寄存器中的一位，并且作为控制单元的额外输入(图 5-40 中显示如何使用这个信号)]

5.6.2 控制部件如何检测异常

现在要设计一种方法去检测异常，并将控制权转交给异常状态中适当的一个。图 5-40 给出了两种新状态(10 和 11)，以及这两个状态与其他有限状态控制的连接。对两种可能的异常分别进行检测：

- 未定义指令：当操作码值的状态 1 没有符合定义的后继状态时，就认为发现了这种异常。为处理这种异常，将除 lw、sw、0(R 型)、j 和 beq 以外的操作码值的后继状态定义为 10。给不符合从状态 1 射出至新状态 10 的弧线上所标的任何操作码的操作码字段标符号 *other*，以表示这种异常情况。
- 算术溢出：■附录 B 讲述了 ALU 中检测溢出的逻辑，并提供了一个称为 Overflow 的信号作为 ALU 的输出。在图 5-40 中，修改过的有限状态机用这个信号指明状态 7 的一个增加的可能的后继状态(11)。

图 5-40 完整地描述了这个有两种异常的 MIPS 子集的控制。请牢记，实际机器的控制设计的挑战性在于处理指令与其他引起异常的事件之间的多种相互影响，同时要求控制逻辑又小又快。可能存在的复杂的相互影响使控制单元在硬件设计中最具挑战性。

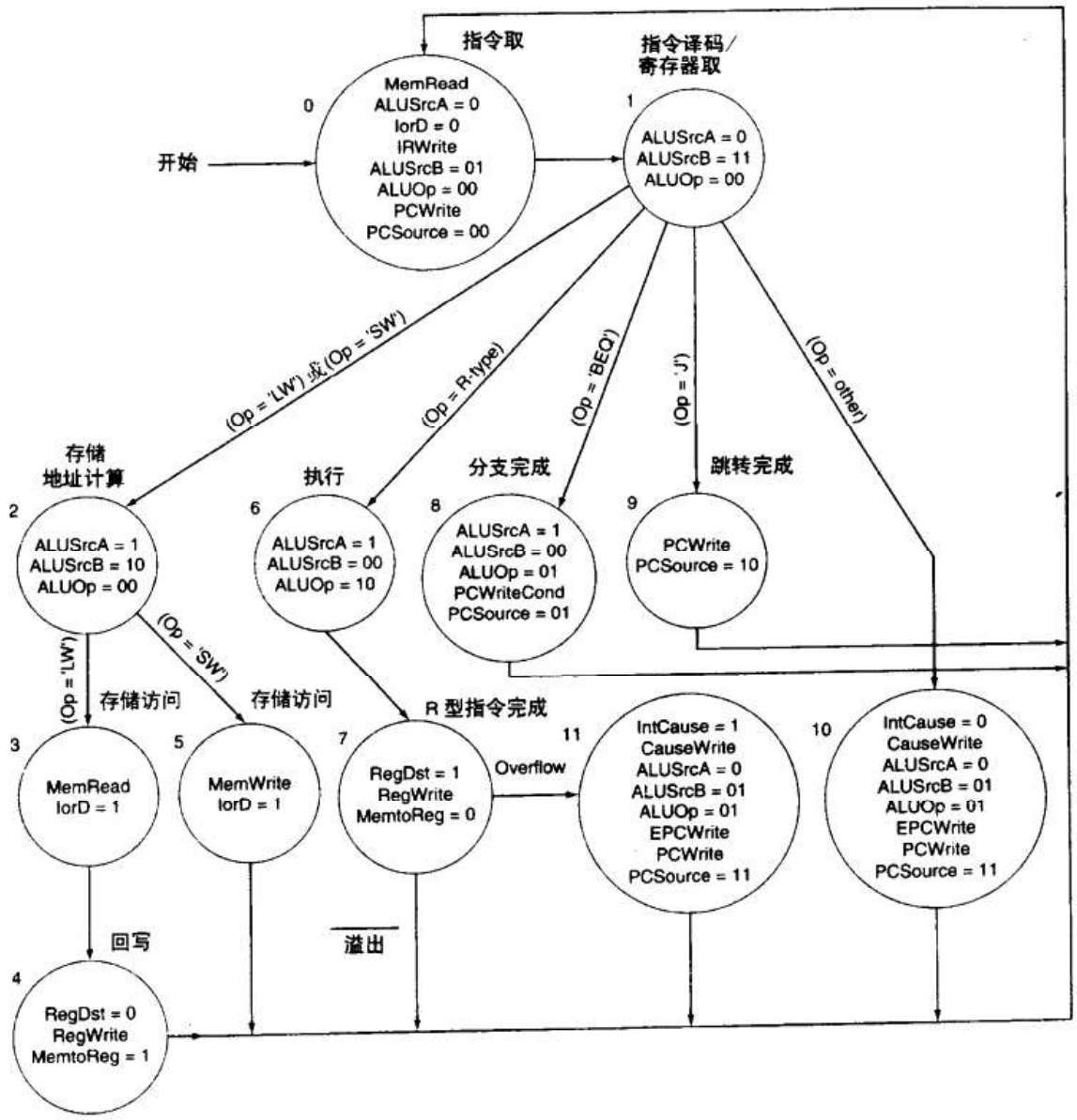

图 5-40 这是加入了处理检测异常部分后的有限状态机

[状态 10 和 11 来是对异常生成相应控制的新的状态。出自状态 1 的标有(Op = other)的弧线指明了当输入与操作码 lw、sw、0(R 型)、j 或 beq 都不匹配时的后继状态。出自状态 7 的标有溢出的弧线指明了当 ALU 出现溢出时应采取的动作]

细节：如果仔细观察图 5-40 的有限状态机，你会发现处理异常的方式可能出现一些问题。例如，对算术溢出，引起溢出的指令仍完成结果的写入，因为溢出分支是在写完成状态中的。不过，系统可以将引起异常的指令定义为无效；MIPS 指令集系统就是这样做的。在第 7 章中，将看到某些类型的异常要求禁止指令改变机器状态，这样处理异常就变得复杂并且限制了机器的性能。

自测

5.5.2 节自测中提到的有关 PCSource 的优化对于图 5-40 中异常的扩展控制是否有效？为什么？

5.7 微程序设计：简化控制设计

微程序设计是设计复杂控制单元的一种技术。它使用非常简单的硬件机构，然后编程实现更复杂的指令集。当前，微程序设计用来实现复杂指令集的某些部分，例如 Pentium 和其他特殊用途处理器。本节收录在光盘中，其中解释了微程序的基本概念，显示了如何用微程序来实现 MIPS 的多周期控制。

5.8 使用硬件描述语言进行数字设计概述

现代数字设计使用硬件描述语言和现代计算机辅助合成工具完成，可以使用库和逻辑合成器将设计描述转化成详细的硬件设计。全书都是关于这些语言和它们在数字设计中的运用。本节收录在光盘中，给出了简要介绍并且阐述了如何使用 Verilog 硬件设计语言从行为角度(适合通过硬件合成的形式)来描述 MIPS 多周期控制。

5.9 实例：近期的 Pentium 处理器的实现结构

本章讲述的数据通路和控制单元的构造技术对每个计算机都至为重要。然而，所有的近期计算机都不仅应用本章介绍的技术还使用流水线。流水线是下一章的主要内容，它通过重叠多条指令的执行来改善机器的性能，使吞吐率达到几乎每周期一条指令(就像单周期实现)，其时钟周期的长度则由单个功能单元的延迟，而非一条指令的整个执行通路(如多周期设计)决定。最后的不采用流水线的 Intel IA-32 处理器是 1985 年生产的 80 386；第一个 MIPS 处理器 R2000 也是在 1985 年生产的，它便采用了流水技术。

最近的 Intel IA-32 处理器 (奔腾 II、III 和 4)采用了越来越复杂的流水线方法。然而，这些处理器仍然面临着为复杂的 IA-32 指令集(参见第 2 章)实现控制电路这一挑战。这些现代处理器的基本功能单元和数据通路，虽然比本章所描述的复杂得多，但具有与本章相同的基本功能和相似的控制信号类型。所以其控制单元的设计基于与本章所用到的相同的原理。

5.9.1 实现更复杂体系结构的挑战

与 MIPS 架构不同，IA-32 体系结构的指令非常复杂，可能用到几十个、甚至上百个时钟周期来执行。例如，串传送指令(MOVS)要求计算并修改两个不同的存储地址，并且存取一个字节串。IA-32 系统繁多而复杂的寻址模式会使得即使像 MIPS 一样简单的指令结构的实现也变得复杂。幸运的是，多周期数据通路的结构，可适应 IA-32 内在指令要进行的多样的工作。这种适应性来自两种能力：

1) 一个多周期数据通路允许指令占用不定数目的时钟周期。IA-32 中类似 MIPS 系统的简单指令可占 3、4 个时钟周期，而复杂的指令可占用几十个时钟周期。

2) 一个多周期数据通路可在一条指令内多次使用数据通路部件。这点对于处理 IA-32 体系结构更复杂的寻址方式和实现 IA-32 更复杂的操作很关键。若没有这种能力，数据通路必须进行扩展，以在不重用部件的前提下处理复杂指令的需求，而这几乎是不可能实现的。例如，一个不重用部件的单周期数据通路若用于 IA-32 ，则需要多个数据存储器和非常多的 ALU。

使用多周期数据通路和**微程序控制器**[⊖]为实现 IA-32 指令集提供了一个框架。然而，挑战性的任务是构造高性能的实现，这要求处理来自不同指令的多样化要求。简而言之，一个高性能的实现要求保证简单指令的快速执行，而复杂指令系统的负担主要由复杂的、不频繁使用的指令承担。

⊖ 微程序控制(microprogrammed control) 使用微代码而不是有限状态机来表示控制的方法。