第 5 章

静态流水线

前 4 章分别介绍了计算机系统结构的基本概念、二进制和逻辑电路以及指令系统结构。有了这些基础，这一章以一个简单的 CPU 为例介绍 CPU 的流水线设计，后面 2 章再介绍比较复杂的流水线和多发射结构。

我们从 MIPS 指令集拣选部分代表性的指令作为简单 CPU 需要实现的指令集，其中指令及其编码列举在表 5.1 中，指令的具体含义及指令集的其他定义请参看本书的第 4 章。

表 5.1　简单 CPU 指令和指令编码

	31 30 29 28 27 26	25 24 23 22 21	20 19 18 17 16	15 14 13 12 11	10 09 08 07 06	05 04 03 02 01 00
ADDU	000000	rs	rt	rd	00000	100001
SUBU	000000	rs	rt	rd	00000	100010
SLT	000000	rs	rt	rd	00000	101010
SLTU	000000	rs	rt	rd	00000	101011
AND	000000	rs	rt	rd	00000	100100
OR	000000	rs	rt	rd	00000	100101
XOR	000000	rs	rt	rd	00000	100110
NOR	000000	rs	rt	rd	00000	100111
SLLV	000000	rs	rt	rd	00000	000100
SRLV	000000	rs	rt	rd	00000	000110
SRAV	000000	rs	rt	rd	00000	000111
ADDIU	001001	rs	rt	immediate		
LW	100011	base	rt	offset		
SW	101011	base	rt	offset		
BEQ	000100	rs	rt	offset		
BNE	000101	rs	rt	offset		
BLEZ	000110	rs	00000	offset		
BGTZ	000111	rs	00000	offset		

5.1 数据通路设计

基于指令系统的定义,先设计这个简单 CPU 的数据通路,其主要模块包括一个指令存储器、一个数据存储器、一个通用寄存器堆、一个指令寄存器(IR)和一个程序计数器(PC),如图 5.1 所示。

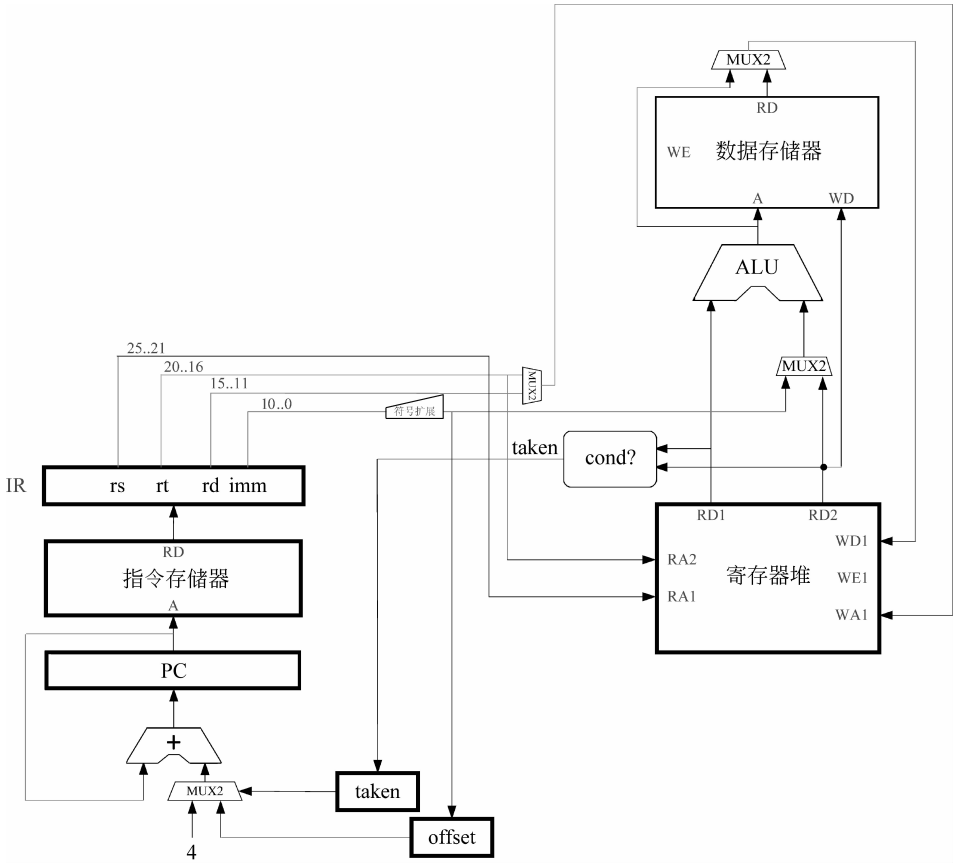

图 5.1 主要数据通路

CPU 工作时,首先用 PC 作为地址去指令存储器中取指令。PC 的值是怎么来的呢?有两种情况,第一种是执行完一条指令顺序执行时,下一条指令的 PC(Next PC,NPC)的值是 PC+4,因为指令占 4 个字节;第二种是执行转移指令时 NPC 值是延迟槽 PC+offset。因为延迟槽指令总是需要执行的,所以当前指令是跳转的转移指令时并不能立即修改 PC 为跳转目标,只能是延迟槽指令在 CPU 里时才能修改。这样,生成 NPC 的部分有一个 2 选 1 逻辑根据转移指令跳转是否成功来选择 offset 值和 4,选择之后再由一个加法器跟 PC 的值相加,并送到 PC 中。然后,根据这个 PC 的值到指令存储器取指,指令取出来以后放到指令寄存器 IR 中。IR 中的指令包含操作码(op)和功能码(func),目标寄存器号(rd),两个源寄存器号(rs、rt),还有立即数/偏移量(imm),其中立即数/偏移量有 16 位,与 rd 和 func 域

有部分重叠。

通用寄存器堆、运算部件和存储器的通路由 IR 中的域统一控制。通用寄存器的内部电路结构如图 5.2 所示，其读地址 RA1 通过控制一个 32 选 1 逻辑从 32 组寄存器中选出一组将其值输出至 RD1，同样的 RA2 控制另一个 32 选 1 逻辑从 32 组寄存器中选出另一组将其值输出至 RD2；当发生写操作时，写地址 WA1 通过译码器得到各组的选择信号再与上全局写使能 WE1 形成每一组寄存器的写使能，用来控制将写入数据 WD1 写入到相应的寄存器组中。IR 的 rs 域连接到通用寄存器堆的读端口 1 的地址输入，从中选出一个将其值送到 ALU 的其中一端；IR 的 rt 域连接到通用寄存器堆的读端口 2 的地址输入，从中也选出一个值来，并和符号扩展后的立即数/偏移量 2 选 1 后送到 ALU 的另外一端。这是因为 ADDIU、LW 和 SW 指令不用寄存器读出的值作为第二个源操作数进行运算，而是用指令中的立即数/偏移量进行运算。转移指令也用到立即数/偏移量，但仅在计算 NPC 时使用，这里我们使用独立的加法器进行 NPC 的计算。ALU 完成计算操作之后要把算术运算或逻辑运算的结果写回到通用寄存器堆里去，具体写回到哪个寄存器由指令中的 rd 或 rt 域来控制，目标连接到通用寄存器堆的写端口 1 的地址输入，进而选中一个寄存器并打开其写使能。对于 LW 指令来说，其目标寄存器号来自于指令的 rt 域而非其他指令的 rd 域，所以需要通过一个 2 选 1 逻辑选择出目标寄存器号。访存指令 LW 和 SW 把 ALU 的运算结果作为访存地址。LW 从数据存储器中把值取出，然后写回到目标寄存器去，所以写入通用寄存器堆的数据也需要通过一个 2 选 1 逻辑从 ALU 运算结果和数据存储器读出结果之间选择。SW 将寄存器堆中读出的值写入到数据存储器中。

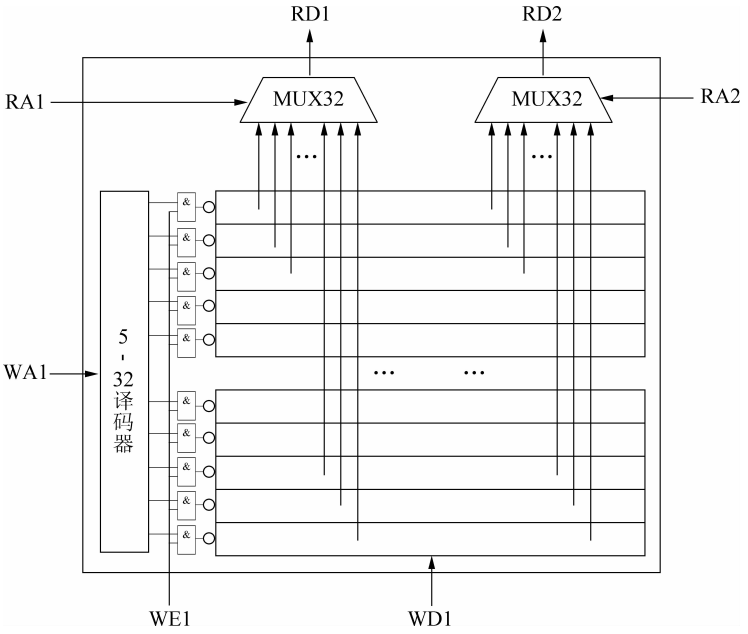

图 5.2　寄存器堆电路结构

上述描述实现了这个 CPU 中的主要数据通路，并涵盖了指令系统中定义的所有指令，但没有描述这个通路的控制逻辑部分。下面我们一步一步地往里加东西。

5.2 控制逻辑设计

实现了 CPU 的数据通路之后,下面先添加 CPU 的控制逻辑。控制逻辑根据指令的要求控制数据在数据通路中流动。

从上述数据通路可以看出,为了让数据根据指令的要求在数据通路中正确地流动,需要对以下通路进行控制:计算 PC 的加法器是否需要看转移跳转情况决定是加 4 还是加 offset(C1);是选择寄存器的值还是选择立即数作为 ALU 的第二个源操作数(C2);ALU 做什么运算(ALUOp);运算结果是把 ALU 的运算结果写回,还是把从数据存储器读出来的结果写回(C3);目的寄存器号是来自指令的 rd 域还是 rt 域(C4);什么情况下使能通用寄存器堆的写使能(C5),因为有一些指令是不写寄存器的,例如 SW 指令和转移指令;什么情况下使能数据存储的写使能(C6)。

根据指令的功能和数据通路的情况,表 5.2 给出了 CPU 中控制逻辑的真值表,其中 X 表示是 0 或 1 无所谓。

表 5.2 控制逻辑真值表

指 令	op 域	func 域	C1	C2	C3	C4	C5	C6	ALUOp
ADDU	000000	100001	0	0	0	0	1	0	0000
SUBU	000000	100010	0	0	0	0	1	0	0001
SLT	000000	101010	0	0	0	0	1	0	0010
SLTU	000000	101011	0	0	0	0	1	0	0011
AND	000000	100100	0	0	0	0	1	0	0100
OR	000000	100101	0	0	0	0	1	0	0101
XOR	000000	100110	0	0	0	0	1	0	0110
NOR	000000	100111	0	0	0	0	1	0	0111
SLLV	000000	000100	0	0	0	0	1	0	1000
SRLV	000000	000110	0	0	0	0	1	0	1010
SRAV	000000	000111	0	0	0	0	1	0	1011
ADDIU	001001	XXXXXX	0	1	1	0	1	0	0000
LW	100011	XXXXXX	0	1	1	1	1	0	0000
SW	101011	XXXXXX	0	1	0①	X	0	1	0000
BEQ	000100	XXXXXX	1	X	0①	X	0	0	XXXX
BNE	000101	XXXXXX	1	X	0①	X	0	0	XXXX
BLEZ	000110	XXXXXX	1	X	0①	X	0	0	XXXX
BGTZ	000111	XXXXXX	1	X	0①	X	0	0	XXXX

① 单就区分目的寄存器时来自 rd 域还是 rt 域,SW、BEQ、BNE、BLEZ 和 BGTZ 指令的 C4 控制信号可以是 0 或 1,但此处设为 0 是为了 5.5 节中生成流水线阻塞控制逻辑的简洁。

根据上述真值表，可以给出相应控制信号的逻辑表达式：

```
wire [5:0] op    = inst[31:26];
wire [5:0] func  = inst[5:0];
wire inst_addu  = (op==6'b0) && (func==6'b100001);
wire inst_subu  = (op==6'b0) && (func==6'b100010);
wire inst_slt   = (op==6'b0) && (func==6'b101010);
wire inst_sltu  = (op==6'b0) && (func==6'b101011);
wire inst_and   = (op==6'b0) && (func==6'b100100);
wire inst_or    = (op==6'b0) && (func==6'b100101);
wire inst_xor   = (op==6'b0) && (func==6'b100110);
wire inst_nor   = (op==6'b0) && (func==6'b100111);
wire inst_sllv  = (op==6'b0) && (func==6'b000100);
wire inst_srlv  = (op==6'b0) && (func==6'b000110);
wire inst_srav  = (op==6'b0) && (func==6'b000111);
wire inst_addiu = (op==6'b001001);
wire inst_lw    = (op==6'b100011);
wire inst_sw    = (op==6'b101011);
wire inst_beq   = (op==6'b000100);
wire inst_bne   = (op==6'b000101);
wire inst_blez  = (op==6'b000110);
wire inst_bgtz  = (op==6'b000111);
wire c1         = inst_beq | inst_bne | inst_blez | inst_bgtz;
wire c2         = inst_addiu | inst_lw | inst_sw;
wire c3         = inst_addiu | inst_lw;
wire c4         = inst_lw;
wire c5         = ~(inst_sw | c1);
wire c6         = inst_sw;
wire [3:0] aluop;
assign aluop[0] = inst_subu | inst_sltu | inst_or | inst_nor | inst_srav;
assign aluop[1] = inst_slt | inst_sltu | inst_xor | inst_nor | inst_srlv | inst_srav;
assign aluop[2] = inst_and | inst_or | inst_xor | inst_nor;
assign aluop[3] = inst_sllv | inst_srlv | inst_srav;
```

上述控制逻辑加在数据通路图的基础上，如图 5.3 所示。图中虚线部分是新加上去的控制逻辑。由 6 位操作码 op 和 6 位功能码 func 形成控制信号，C1 与转移条件判断的结果 cond 相“与”后控制 NPC 加法器第二个操作数的 2 选 1，C2 控制 ALU 第二个操作数的 2 选 1，ALUOp 控制运算部件 ALU，C3 控制结果写回的 2 选 1，C4 控制目的寄存器的 2 选 1，C5 控制通用寄存器的写使能，C6 控制数据存储器的写使能。这些控制信号都是组合逻辑，这个过程称为译码，即把指令操作码翻译成 CPU 主要数据通路所需要的控制信号。

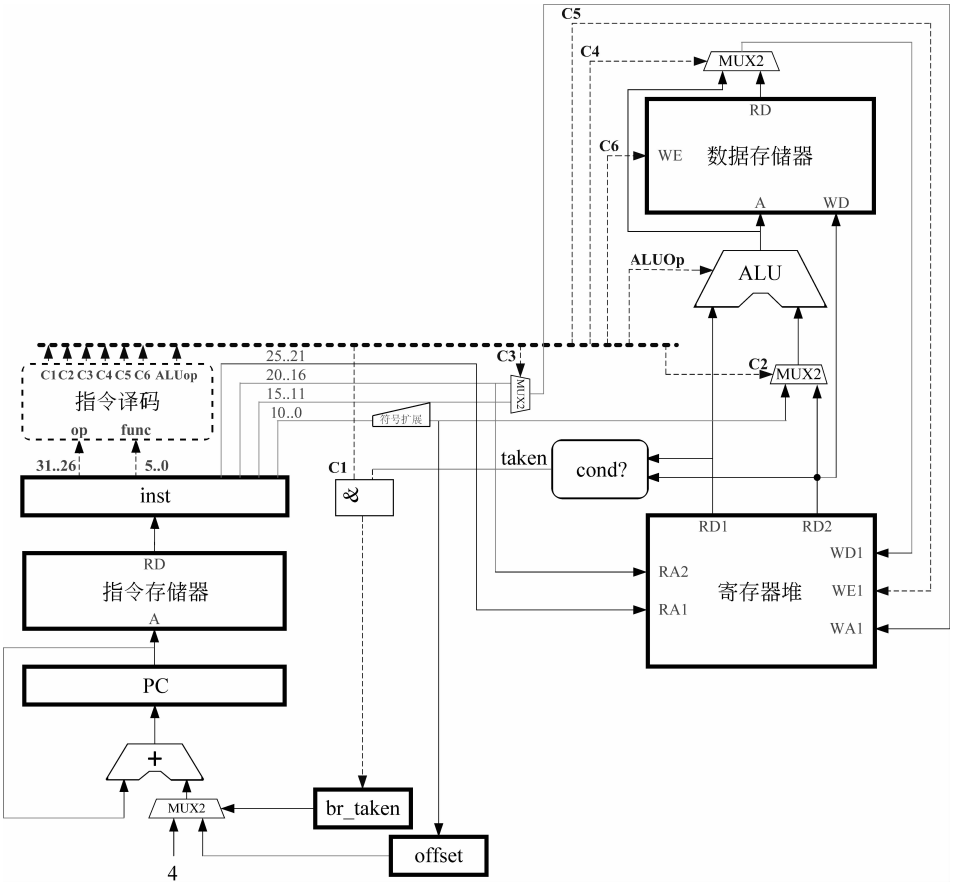

图 5.3 主要控制逻辑

5.3 时序

到目前为止,基本实现了 CPU 的主要数据通路和控制通路,但是这个 CPU 还是不能工作,还差什么呢? 还差时钟部分。时钟就是 CPU 的驱动,就像人的心跳。第 3 章介绍过,时序电路是需要由时钟驱动的。

可以简单直观地把一条指令的执行分成 3 步:①计算 PC 值;②根据这个 PC 值把指令从指令存储器里面取出来放到指令寄存器里;③根据指令寄存器的内容来控制这个指令的执行,并把结果写到寄存器或数据存储器。上述 3 步工作可以分别由 3 个时钟来驱动,这 3 个时钟可以通过对一个统一的时钟 CLK 进行分频得到。第一个时钟把 NPC 值送到 PC 里面去;等到根据 PC 寄存器的值完成取指后,第二个时钟把取出的指令送到 IR 里面去;等到指令执行完后,第三个时钟把指令的运算结果送入到通用寄存器或数据存储器。一条指令执行完以后,再执行下一条,CPU 就能循环往复地工作了,如图 5.4 所示。

图 5.5 描述了上述指令执行的 3 个阶段。从图中可以看出,上述指令执行的 3 个阶段是顺序进行的。

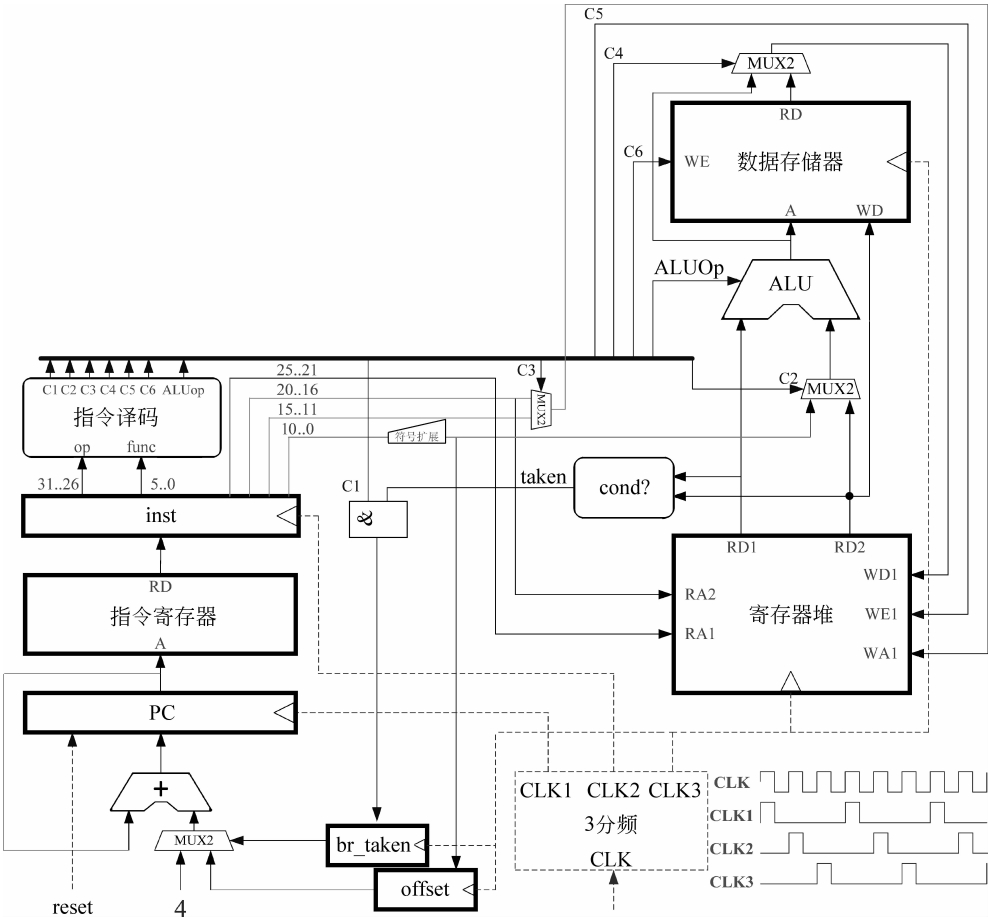

图 5.4　带时序的控制逻辑

图 5.5　指令执行的 3 个阶段

5.4　流水线技术

现在这个 CPU 一拍一拍、一条一条地执行指令。就像在一个超市里面，第一个人进去了，第二个人就不许进，等他买完了东西，从出口出来了以后，才放第二个人进去。当然，从功能上考虑这是不会出错的，但是它不够高效，所以需要改进。

当前的设计中的 3 个步骤（计算 PC、取指、执行）可以合并成 2 个步骤：计算下一条指令的 PC 和指令执行可以重叠。这样重叠不会出错，因为计算 NPC 跟当前指令执行没有直接关联；PC＋4 或根据转移指令 PC＋offset，也可以理解为是对当前指令的执行。因此，可以在指令执行的同时计算下一个 PC。这样，就可以把 3 步变成 2 步来完成指令的执行，其

中第 2 步把计算 PC 和指令执行重叠。相应地,把原来 3 个时钟变成 2 个时钟就可以了,如图 5.6 所示。

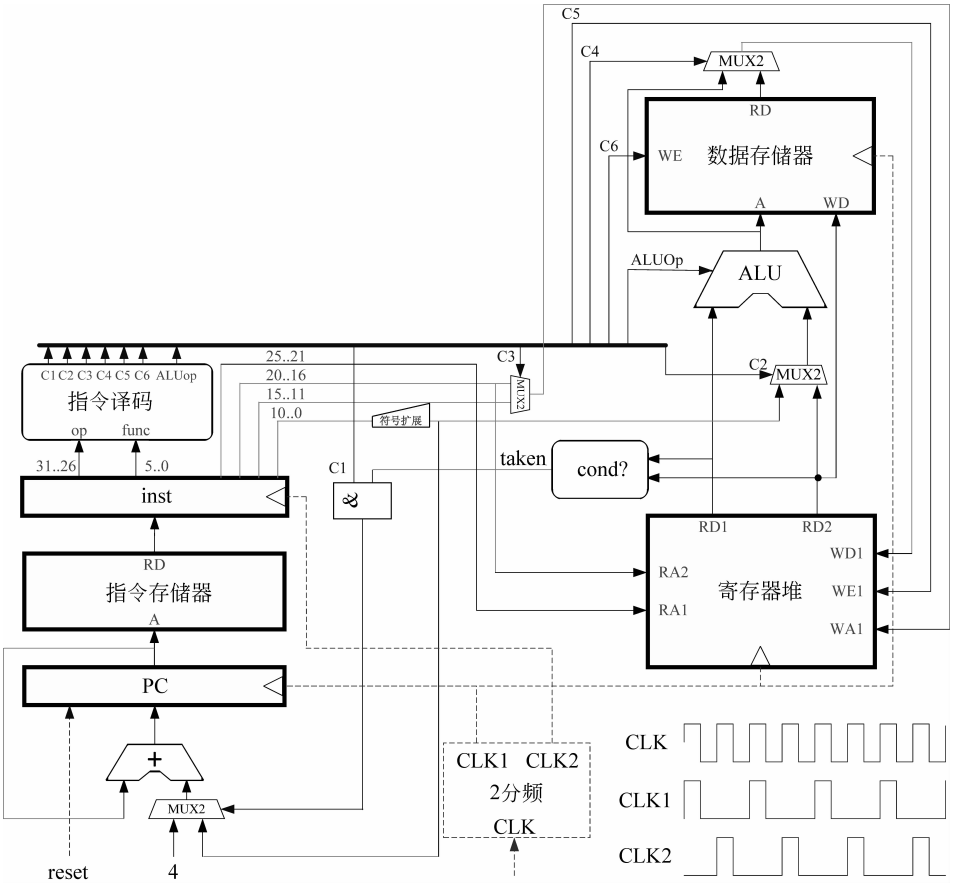

图 5.6 改进后的时序(计算 PC 与执行重叠)

图 5.7(a)描述了把上述流水线中第 N 条指令的执行和第 $N+1$ 条指令计算 PC 重叠后的流水线阶段,图 5.7 (b)描述了把执行和计算 PC 合并的流水线阶段。

在此基础上,还可以进一步对该 CPU 结构进行改进,看能不能把取指和执行也重叠。在大多数的情况下,取指和执行是可以重叠的。例如,当第 $N+1$ 条指令执行的时候,第 N 条的指令已经执行完,它已经把所运算的结果写回到寄存器了,第 $N+1$ 条指令可以使用第 N 条指令的结果。但是有一种特殊情况,如果第 $N+1$ 条指令的取指(即 PC 的计算)也要用到第 N 条指令的结果,那么第 $N+1$ 条指令的取指必须等到第 N 条指令执行结束了才能执行。什么情况下第 $N+1$ 条指令的取指会用到第 N 条指令的结果呢? 如果第 N 条指令是转移指令就会出现这种情况。在这种情况下,第 $N+1$ 条在取指的时候,需要知道转移指令成功不成功,往哪里跳转,这个时候就不能把第 $N+1$ 条指令的取指和第 N 条指令的运算叠在一起了。这时就能看出定义延迟槽指令的作用了。所谓延迟槽指令,是指紧挨着转移指令后面的那条指令,不论转移是否成功都执行,这样即使第 N 条指令是转移指令,第 $N+1$ 条指令的取指也不会用到第 N 条指令的结果了。经过上述改进,整个 CPU 的所有部

分只要一个时钟就行了，如图 5.8 所示。

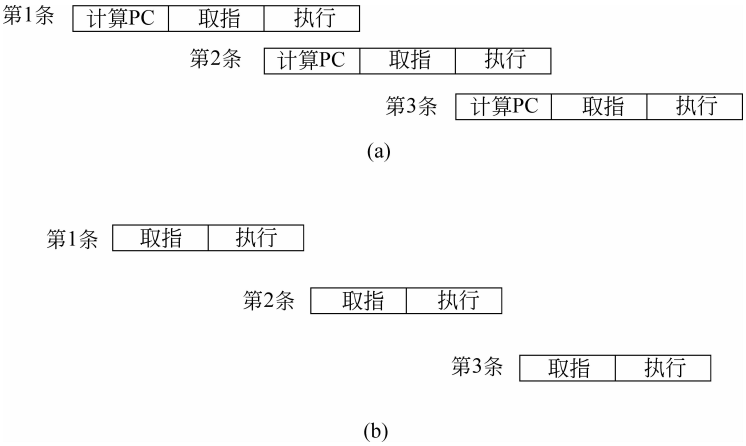

图 5.7　计算 PC 跟执行重叠的流水线阶段

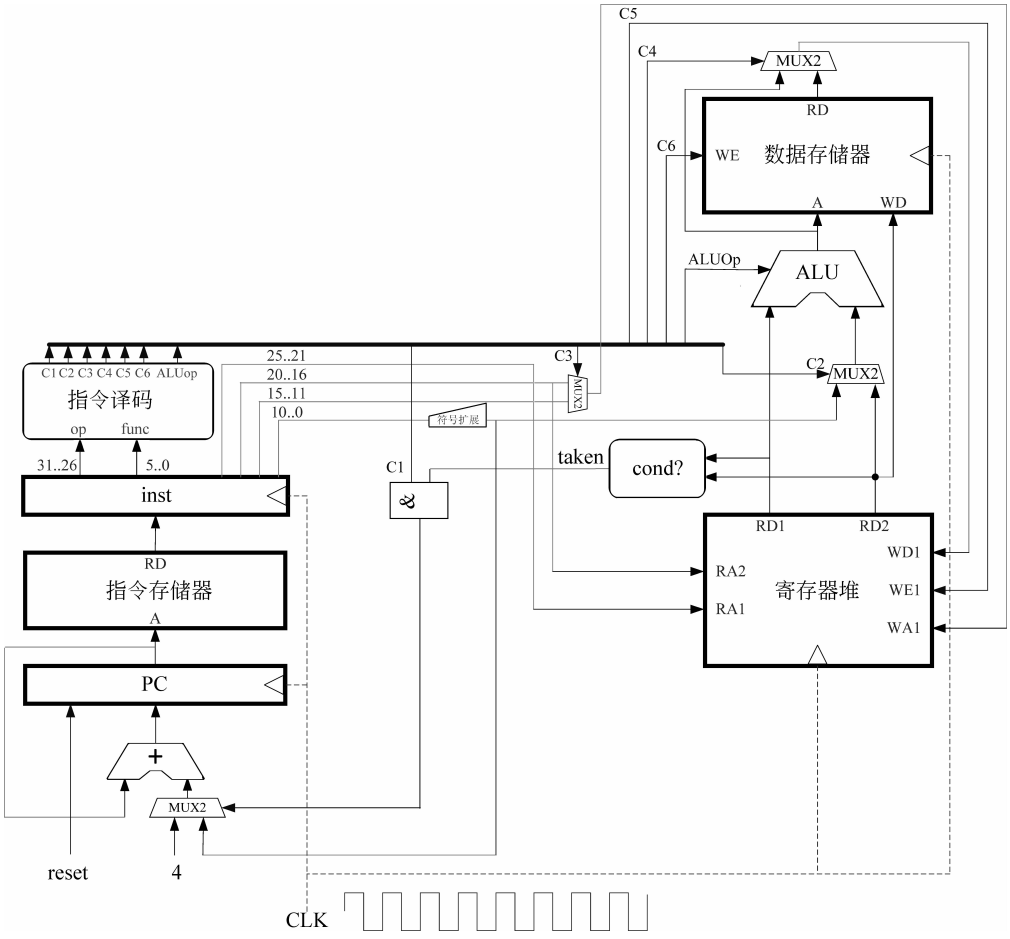

图 5.8　再次改进后的时序

图 5.9 描述了把上述流水线中第 N 条指令的执行和第 $N+1$ 条指令的取指重叠后的流水线阶段。

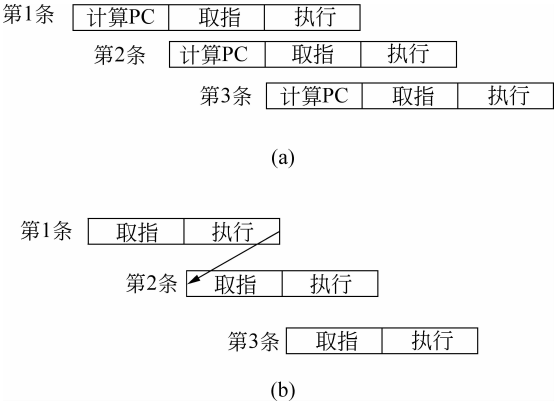

图 5.9 取指与执行重叠的流水线阶段

然而,上述流水线的设计有一个问题,就是在指令执行阶段需要干的事情太多了。根据上述流水级的划分,指令进入指令寄存器 IR 后,要进行译码、读寄存器的值、送到运算器进行运算,有时还要把运算结果当作地址进行访存,最后把结果写回到寄存器。我们在前面学习二进制和逻辑电路的时候,知道任何逻辑运算都是有延迟的,所以指令执行阶段的延迟就非常大。这样设计出来的 CPU 虽然能工作,但主频不高,上一个时钟脉冲把指令送到指令寄存器后要等比较长的时间,才能发出下一个时钟脉冲把结果写到寄存器中。

可以通过进一步细分执行阶段来减少每一拍的工作量,例如把原来执行阶段的工作切分为译码、运算、访存、写回 4 个阶段。这时指令的执行先做译码,然后读寄存器把值读出来并送到 ALU,运算完成后如果是一个访存操作就根据算出来的地址访问数据存储器,最后把运算或访存的结果写回到寄存器堆。流水线的阶段划分如图 5.10 所示。上述流水级的划分在 CPU 执行阶段增加了几组寄存器,用于存放每个阶段的中间结果,即每一阶段结束后通过时钟脉冲把其结果保存在中间寄存器中。上述设计就是传统 RISC CPU 设计中的标准 5 级流水线。在 5 级流水线 CPU 中,同时有 5 条指令在执行,每条指令所处的流水线阶段不一样,所以控制也不一样。不仅要把每个流水阶段的数据存在中间寄存器,还要把每个流水线阶段及其后续阶段要用到的控制信号(ALUOp,C3,C5,C6)和目标寄存器号(dest)存起来跟着指令走,这样就可以在流水线中同时执行 5 条不同阶段的指令。

还把这个 CPU 比作一个超市的话,现在这个超市可以同时有 5 个人在不同的地方选择商品。以后会介绍动态流水线和多发射,这些现代的 CPU“超市”中,可以同时有几十甚至几百个人在选择不同的商品。当然,CPU 比超市复杂得多,因为超市里的人互相之间是没有太多关系的,只要他们不争夺同一件商品就相安无事。但 CPU 的指令之间存在各种关系,例如一条指令需要用到前一条指令的运算结果,或者一条指令是否执行需要等前面的转移指令结束后才能确定。