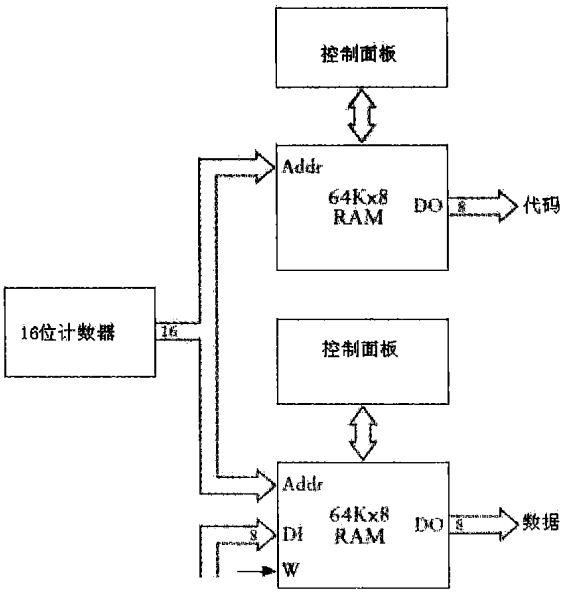
“代码”（第二个 RAM 阵列）。其结构如下图所示。



我们已经清楚地认识到新的自动加法器能够把数据求和的结果写入到第一个 RAM 阵列（标记为“数据”），而新的 RAM 阵列（标记为“代码”）则只能通过控制面板写入。

我们需要四个代码来标记新的自动加法器需要做的四个操作，这些代码可以任意指定。如下所示的是一种方案。

操作码	代码
Load（加载）	10h
Store（保存）	11h
Add（加法）	20h
Halt（停止）	FFh

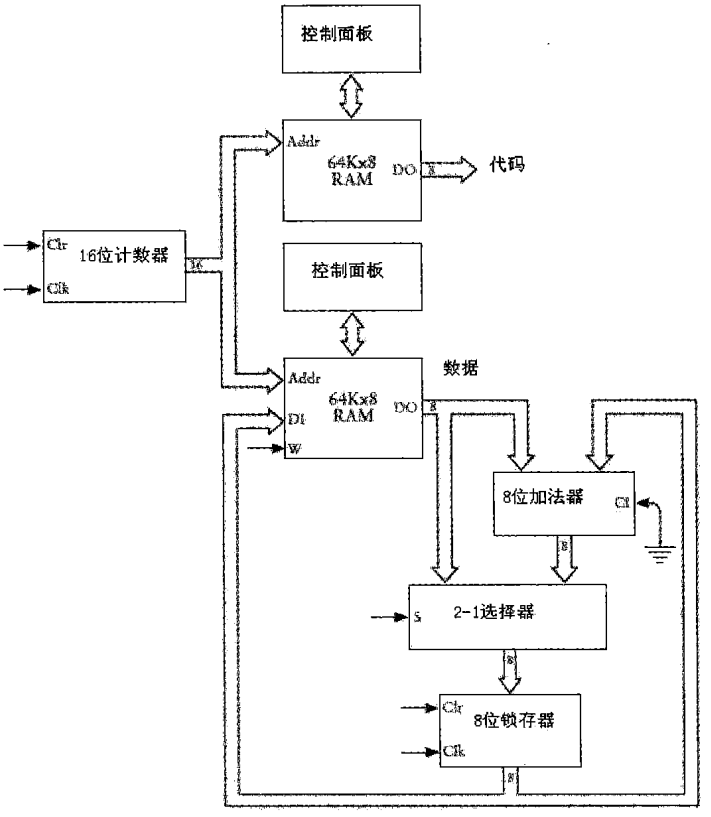
为了使上面讨论的三组加法得以正常执行，你需要通过控制面板把如下值存入代码 RAM 阵列。

比较一下该 RAM 阵列与存放累加数据的 RAM 阵列中的内容，你会发现，代码 RAM 阵列中存放的每一个代码都对应着数据 RAM 中要被加载或者加到累加器中的数，或者对应需要存回到数据 RAM 中的某个数。以这种方式使用的数字代码常常被称为指令码（instruction code）或操作码（operation code，opcode）。它们指示电路要执行的某种操作。

0000h:	10h	Load
	20h	Add
	20h	Add
	11h	Store
0004h:	10h	Load
	20h	Add
	11h	Store
0007h:	10h	Load
	20h	Add
	20h	Add
	11h	Store
000Bh:	FFh	Halt

如前所述，最初的自动加法器的 8 位锁存器的输出要作为数据 RAM 阵列的输入，这就是 Save 指令的功能。还需要做另一个改变：以前 8 位加法器的输出是 8 位锁存器的输入，但现在为了执行 Load 指令，数据 RAM 阵列的输出有时也要作为 8 位锁存器的输入，这种新的变化需要一个 2-1 选择器来实现。改进后的自动加法器如下图所示。

图中略去了一些组件，但是仍然清晰地描述了各个组件之间的 8 位数据通路。16 位的计数器为两个 RAM 阵列提供地址输入。通常，数据 RAM 阵列的输出传入到 8 位加法器执行加操作。8 位锁存器的输入可以是数据 RAM 阵列的输出（当执行 Load 指令时），也可以是加法器的输出（当执行 Add 指令时），这种情况下就需要 2-1 选择器。通常，锁存器电路的输出又流回到加法器中，但是当执行 Save 指令时，它就成为了数据 RAM 阵列的输入数据。

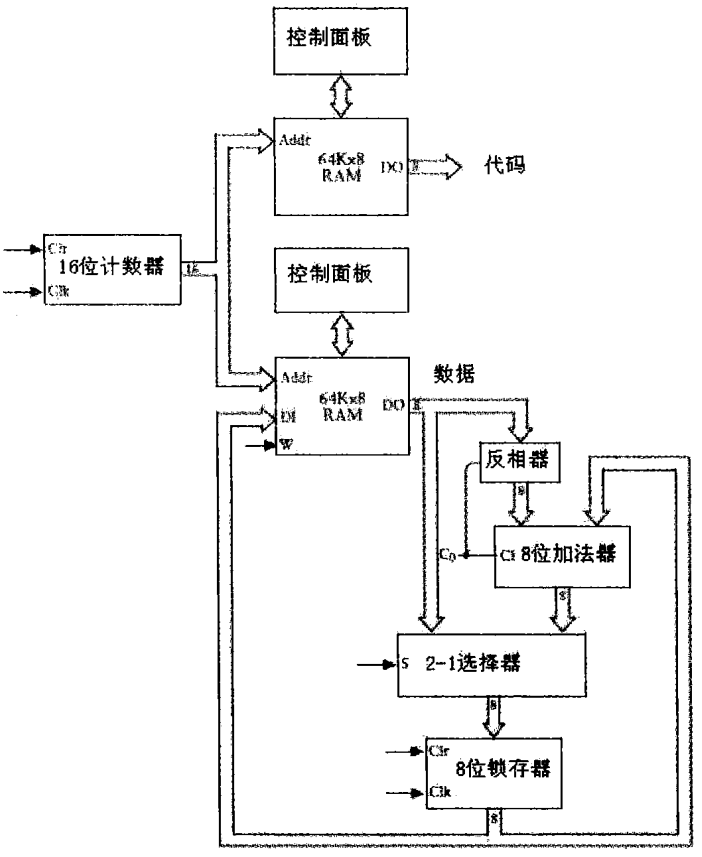


上图缺少的是控制所有这些组件的信号，它们统称为控制信号，包括 16 位计数器的“时钟”输入和“清零”输入，8 位锁存器的“时钟”输入和“清零”输入，数据 RAM 阵列的“写”（W）输入，2-1 选择器的“选择”（S）输入。其中的一些信号很明显是基于代码 RAM 阵列的输出，例如，如果代码 RAM 阵列输出是 Load 指令，那么 2-1 选择器的“选择”输入必须是 0（即选择数据 RAM 的输出）。只有当操作码是指令 Store 时，数据 RAM 阵列的“写”（W）输入必须是 1。这些控制信号可以通过逻辑门的各种组合来实现。

利用最少的附加硬件和一些新增的操作码，可以让这个电路从累加器中减去一个数。第 1 步是向操作码表增加一些代码。

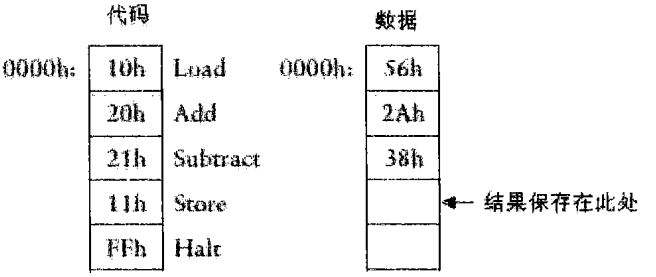
操作码	代码
Load	10h
Store	11h
Add	20h
Subtract（减法）	21h
Halt	FFh

对于 Add 和 Subtract 的代码，其区别仅在于最低有效位，我们称该位为 C_0。如果操作码为 21h，除了数据 RAM 阵列的数据传入加法器之前要取反，并且加法器进位输入置 1 之外，电路所做的操作与执行 Add 指令所做的操作相同。在这个增加了一个反相器的改进电路中，C_0 信号可以完成这两项任务。改进后的电路结构图如下。



假设现在要把 56h 和 2Ah 相加，然后再从中减去 38h，可以按照下图中两个 RAM 阵

列中的代码（操作码）和数据（操作数）完成该运算。



Load 操作完成之后，累加器中的值更新为 56h，加法操作完成后累加器中的值为 56h 与 2Ah 的和，即 80h。Subtract 操作使数据 RAM 阵列的下一个值（38h）按位取反，得到 C7h。当加法器的进位输入置 1 时，取反得到 C7h，然后使其与 80h 相加：

$$\begin{array}{r} \text{C7h} \\ +\ \text{80h} \\ +\ \ \text{1h} \\ \hline \text{48h} \end{array}$$

最后的结果是 48h。（在十进制中，86 加 42 再减去 56 等于 72）

还有一个一直没有找到合适的解决办法的问题：加法器及连接到它的所有设备的宽度只有 8 位。以前提出过的一个解决办法是把两个 8 位加法器（其他的大部分设备也用两个）连在一起，构成一个 16 位的设备。

但还有代价更小的解决办法，假如你想把两个 16 位的数相加，比如：

$$\begin{array}{r} \text{76ABh} \\ +\ \text{232Ch} \\ \hline \end{array}$$

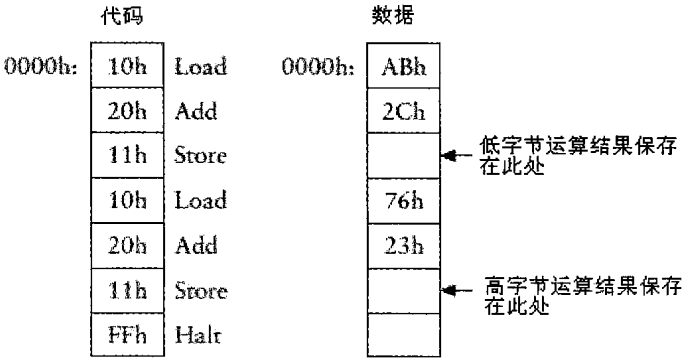
这种 16 位的加法先单独处理最右边的字节（通常称之为低字节）：

$$\begin{array}{r} \text{ABh} \\ +\text{2Ch} \\ \hline \text{D7h} \end{array}$$

然后再计算最左边的字节，即高字节的和：

$$\begin{array}{r} 76h \\ +23h \\ \hline 99h \end{array}$$

得到相同的结果 99D7h。因此，如果我们把两个 16 位的数用这种方式保存在存储器中，就像下面这样：



运算结果 D7h 将被保存到地址 0002h，而结果 99h 将被保存到地址 0005h。

当然并非所有的情况都是这样处理，只是上面的例子中用到了这种方法。如果要把 76ABh 和 236Ch 这两个 16 位的数相加该怎么做呢？在这个例子中，对两个数的低字节求和时将会产生一个进位：

$$\begin{array}{r} ABh \\ +6Ch \\ \hline 117h \end{array}$$

产生的这个进位必须与两个数的高字节的和再相加：

$$\begin{array}{r} 1h \\ +76h \\ +23h \\ \hline 9Ah \end{array}$$

最后的计算结果为 9A17h。

我们能够改进自动加法器的电路，使它可以正确地进行 16 位数的加法操作吗？答案是肯定的，我们需要做的仅仅是在第一步运算时保存低字节数运算的进位输出，并把它

作为下一步高字节数运算的进位输入。如何保存 1 位呢？1 位锁存器就是最好的选择了，该锁存器应该被称为进位锁存器（Carry latch）。

为了使用进位锁存器，还需要另一个操作码，我们称之为“进位加法”（Add with Carry）。当进行 8 位数加法时，使用的是常规的 Add 指令。加法器的进位输入是 0，它的进位输出将会保存到进位锁存器（尽管它根本不会被用到）。

如果要对两个 16 位的数进行加法运算，我们仍然使用常规的 Add 指令对两个低字节数进行加法运算。加法器的进位输入是 0，而其进位输出被锁存到进位锁存器中。当把两个高字节数相加时，要使用新的 Add with Carry 指令。在这种情况下，两个数相加时要用进位锁存器的输出作为加法器的进位输入。因此，如果第一步低字节数的加法运算有进位，则该进位将用于第二步高字节数的加法运算；如果没有进位，则进位锁存器的输出是 0。

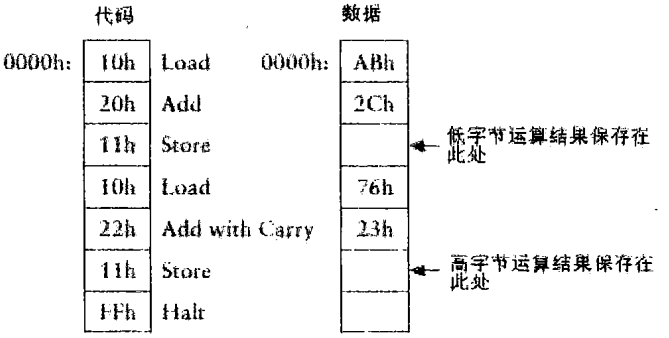
如果要进行 16 位数的减法运算，则还需要一个新的指令，称为“借位减法”（Subtract and Borrow）。通常，Subtract 指令需要将减数取反并且把加法器的进位输入置 1。进位输出通常不是 1，因此应该被忽略。但对 16 位数进行减法运算时，进位输出应该保存在进位锁存器中。在进行第二步的高字节减法运算时，锁存器保存的结果应该作为加法器的进位输入。

在加入了 Add with Carry 和 Subtract and Borrow 之后，目前我们已经有了 7 个操作码，如下表所示。

操作码	代码
Load	10h
Store	11h
Add	20h
Subtract	21h
Add with Carry（进位加法）	22h
Subtract with Borrow（借位减法）	23h
Halt	FFh

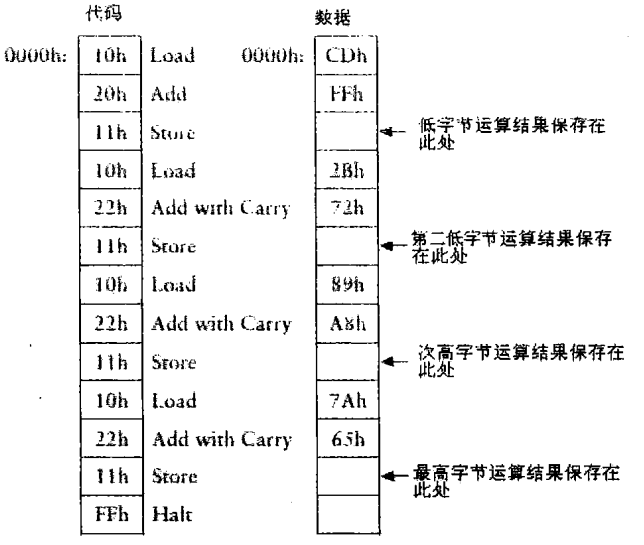
在执行减法或借位减法运算时，送入加法器中的操作数需要进行取反预处理。加法器的进位输出是进位锁存器的数据输入。无论何时，执行加法、减法、进位加法或借位减法中的任一种运算，进位锁存器都是同步的。当执行减法运算，或进位锁存器的数据输入为 1 且正在执行进位加法或者借位减法运算时，8 位加法器的进位输入都是置 1 的。

需要记住的是，只有当前一次的加法或者进位加法操作使加法器产生进位输出时，Add with Carry 指令才会使 8 位加法器的进位输入置 1。因此，只要进行多字节数加法运算，不管实际是否需要，都应该使用 Add with Carry 指令。为了保证编码的正确，使前面提到 16 位加法正常进行，可用如下方法。



不论操作数是什么，该方法都可以正确执行。

增加了两个新的操作码之后，我们已经极大地扩展了加法器的功能，它不再局限于 8 位数的加法运算。通过执行进位加法操作，可以对 16 位数、24 位数、32 位数、40 位数，甚至更多位的数进行加法运算。假如要进行两个 32 位数 7A892BCDh 和 65A872FFh 的加法运算，我们仅需要 1 条 Add 指令和 3 条 Add with Carry 指令，如下图所示。



当然，把这些数依次输入存储器并不是最好的做法。因为你不但要使用开关来输入这些数，而且保存这些数的存储单元的地址也不是连续的。例如，7A892BCDh 从最低字节开始，每个字节依次保存在 0000h，0003h，0006h，0009h 中。而为了得到最后的结果，还需要检查 0002h，0005h，0008h，000Bh 这几个地址中的数。

除此之外，当前设计的自动加法器不允许在随后的计算中重复使用前面的计算结果。假设我们要对三个 8 位数求和，然后再从中减去一个 8 位数并保存结果。这可能需要一条 Load 指令，两条 Add 指令，一条 Subtract 指令以及一条 Store 指令。但如果想从原来的求和结果（3 个 8 位数的和）中减去另一个数该怎么做呢？这个求和结果已经不能被访问了，每次我们使用它的时候都必须重新计算。

产生上述情况的原因就在于我们构造的自动加法器具有如下的特性：它的代码存储器和数据存储器是同步的、顺序的，并且都从 0000h 开始寻址。代码存储器中的每一条指令对应数据存储器中相同地址的存储单元。一旦执行了一条 Store 指令，相应的，就会有一个数被保存到数据存储器中，而这个数将不能重新加载到累加器。

要解决这个难题，需要对自动加法器的设计做一个根本性的且程度极大的修改。这个想法实现起来似乎非常困难，但是很快你就会发现（我希望是这样）改进后的加法器具有更高的灵活性。

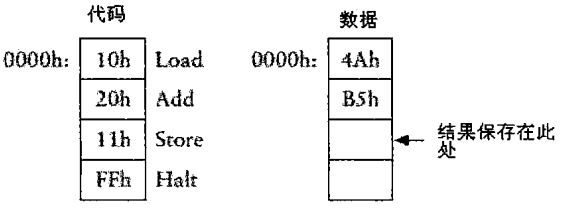
现在让我们立刻开始吧，目前已经有了 7 个操作码，如下所示。

操作码	代码
Load	10h
Store	11h
Add	20h
Subtract	21h
Add with Carry（进位加法）	22h
Subtrac t with Borrow（借位减法）	23h
Halt	FFh

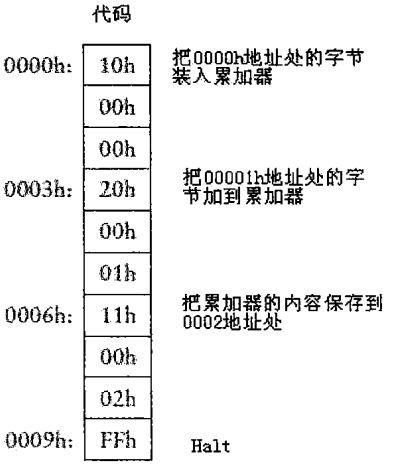
每一个操作码在存储器中占 1 个字节。现在除了 Halt 操作码外，我希望每一个指令在存储器中仅占据 3 个字节的空间，其中第一个字节为代码本身，另外的两个字节用来存放 1 个 16 位存储器单元地址。对于 Load 指令来说，后两个字节保存的地址用来指明数据 RAM 阵列的一个存储单元，该单元存放的是需要被加载到累加器中的字节。对于

Add，Subtract，Add with Carry，Subtract with Borrow 指令来说，该地址指明的存储单元所保存的是要从累加器中加上或减去的字节。对于 Store 指令来说，该地址指明的是累加器中的内容将要保存到的存储单元地址。

例如，当前加法器所能进行的最简单的运算就是对两个数求和。为了执行这个操作，需要按下面的方式设置代码 RAM 阵列和数据 RAM 阵列。



在改进的自动加法器中，每条指令（除了 Halt 指令）需要 3 个字节。

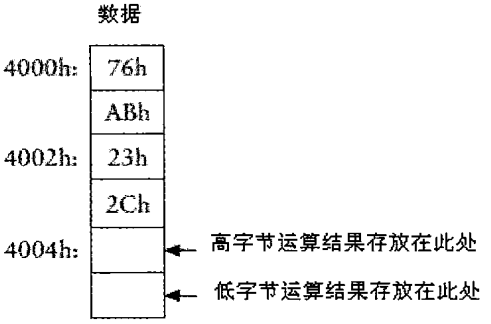


每一条指令的代码（除了 Halt 指令）后跟两个字节，用来指明数据 RAM 阵列中 16 位的存储地址。这三个地址恰巧是 0000h，0001h 和 0002h，但它们可以是任何其他可用的地址。

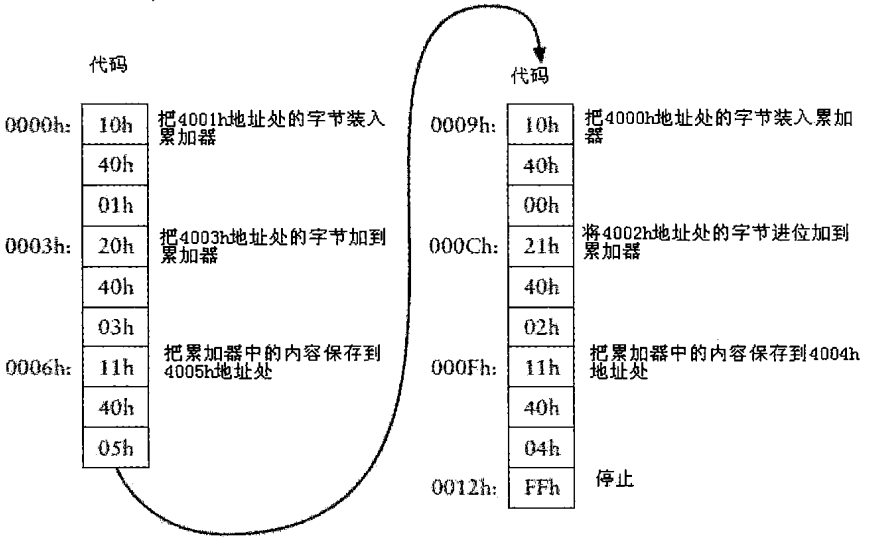
前面讲到了如何用 Add 或 Add with Carry 指令来对两个 16 位数——比如 76ABh 和 232Ch 求和。必须把两个数的低字节保存到存储器的 0000h 和 0001h 地址，把其高字节保存到 0003h 和 0004h 地址，运算的结果分别保存在 0002h 和 0005h。

通过这种变化，我们可以用一种更合理的方式来保存这两个操作数及其运算结果，

可能会把它们保存到我们从未用到过的存储区域。



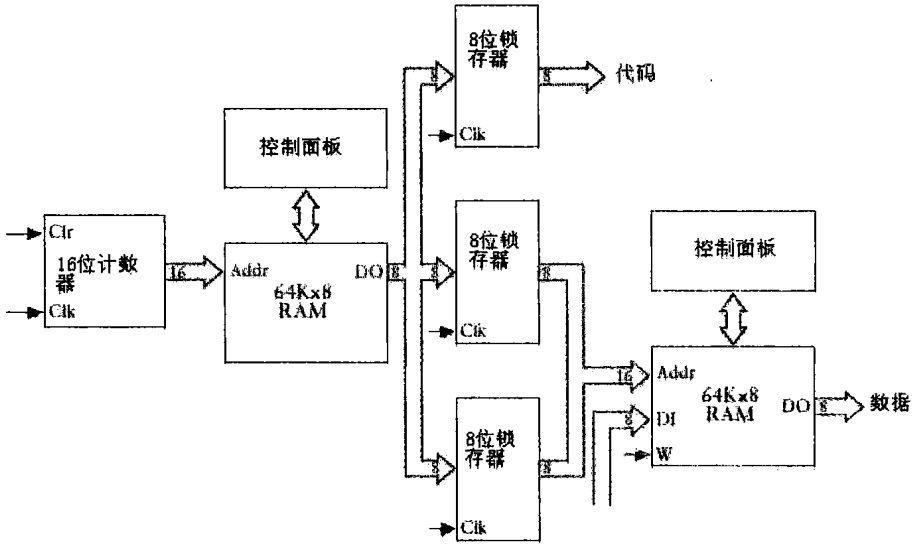
这 6 个存储单元不必像上图中这样全都连在一起，它们可以分散在整个 64 KB 数据 RAM 阵列的任意位置。为了把这些地址中的数相加，代码 RAM 阵列中的指令必须用以下方式设置。



可以看到，保存在地址 4001h 和 4003h 处的两个低字节数先执行加法，其结果保存在 4005h 地址处。两个高字节数（分别保存在 4000h 和 4002h 处）通过 Add with Carry 指令相加，其结果保存在地址 4004h 处。如果去掉 Halt 指令并向代码 RAM 中加入更多指令，随后的计算可以通过引用地址很方便地使用原来的那些操作数及其结果。

实现该设计的关键是把代码 RAM 阵列的数据输出到 3 个 8 位锁存器中。每个锁存器

保存该 3 字节指令的一个字节。第一个锁存器保存指令代码本身，第二个锁存器保存地址的高字节，第三个锁存器保存地址的低字节。第二个和第三个锁存器的输出构成了数据 RAM 阵列的 16 位地址。



从存储器中取出指令的过程称为取指令（instruction fetch）。在我们设计的加法器中，每一条指令的长度是 3 个字节。因为每次从存储器取回一个字节，所以取每条指令需要的时间为 3 个时钟周期。此外，一个完整的指令周期需要 4 个时钟周期。这些变化必然使得控制信号更加复杂。

机器响应指令码做一系列操作的过程称为执行（execute）指令，但这并不能表明机器是一种有生命的东西，因为它不能自行分析机器代码并决定该做什么。每一种机器码用其唯一的方式触发多种控制信号，从而引发机器执行各种操作。

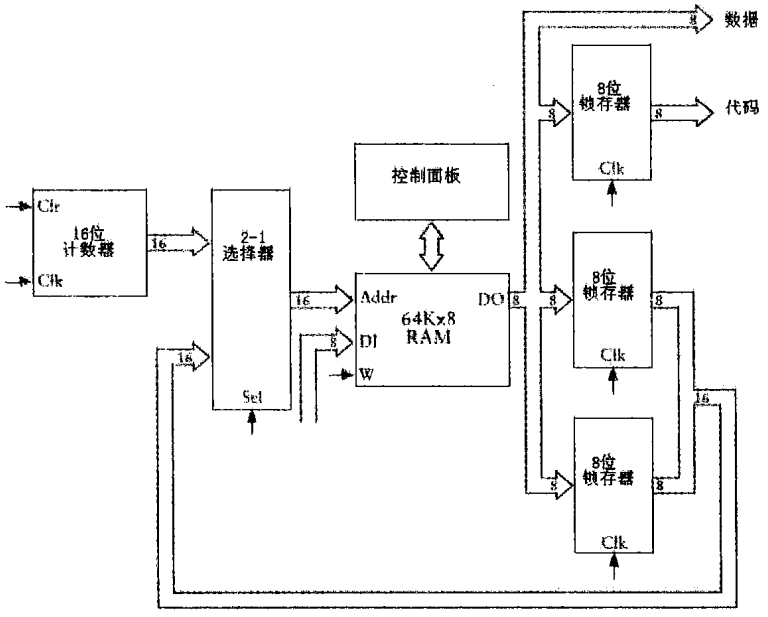
注意，为了让上面的加法器更加有用，我们牺牲了运算速度。使用同样的振荡器，它的运算速度只有本章提到的第一个加法器的 1/4。这验证了一个称为 TANSTAAFL 的工程准则，它的意思是“天下没有免费的午餐”。通常上帝总是很公平的，你改进了机器的某个方面，则其他方面就会受到损失，有得就有失。

如果不使用继电器构造这个电路的话，很显然，两个 64 KB 的 RAM 阵列构成了电路中最主要的部分。事实上早就应该放弃这些部件了，甚至从一开始就应该决定只需要

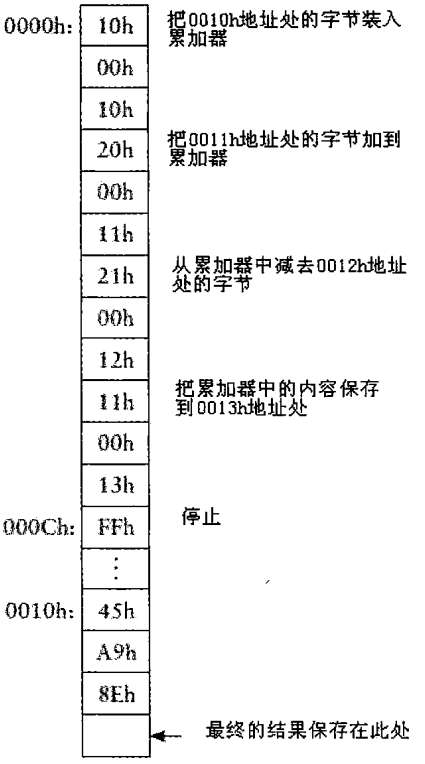
一个 1 KB 的存储器。如果能确定所有的数据都可以存放在 0000h～03FFh 的地址空间中，那么使用一个小于 64 KB 的存储器加法器也可以很好地工作。

但是你现在仍然可能不在意使用了 2 个 RAM 阵列。事实上，确实不必要在意。前面介绍了两种 RAM 阵列，一个用来存放指令码，另一个用来存放操作数据——这种设计使得自动加法器的结构非常清晰和易于使用。但现在我们使用 3 字节长的指令格式，第二个和第三个字节用来指明操作数的存储地址，因此就没有必要再使用两个独立的 RAM 阵列。操作码和操作数可以存放在同一个 RAM 阵列。

为了实现这个设计，我们需要一个 2-1 选择器来确定如何对 RAM 阵列寻址。通常，和前面的方式相同，我们用一个 16 位的计数器来计算地址。数据 RAM 阵列的输出仍然连接到 3 个锁存器，分别用来保存指令代码及其对应操作数的 16 位地址，其 16 位的地址输出是 2-1 选择器的第二种输入。地址被锁存后，可以通过选择器将其作为 RAM 阵列的地址输入。



我们已经对原电路做了不少改进，现在可以把操作指令和操作数据保存在同一个 RAM 阵列中。例如，下图演示了如何把两个 8 位数相加，然后从结果中再减去一个 8 位数。

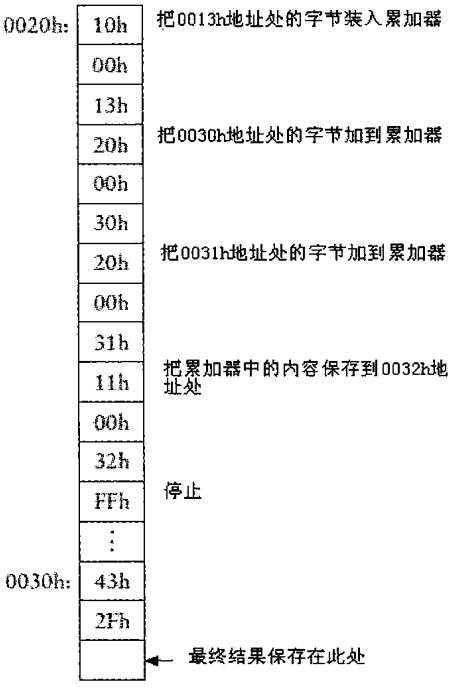


通常，指令从 0000h 开始存放，这是因为当计数器复位后从该位置访问 RAM 阵列。最后的 Halt 指令存放在 000Ch 地址。我们可以把这 3 个操作数及它们的运算结果保存在 RAM 阵列的任何地址（当然这不包括最开始的 13 个字节，因为它们已经用来存放操作指令），所以我们选择在从 0010h 地址开始保存操作数。

假设现在你发现需要在原来的结果中再加两个数，你可以向存储器中输入一些新的指令以替换原来所有的指令，但是你可能不愿意这么做。或许你更倾向于在原指令的地址后增加一些新的指令。第一步要做的就是把 000Ch 地址处的 Halt 指令替换为一个 Load 指令。但你仍然需要增加两条 Add 指令，一条 Store 指令，以及一条新的 Halt 指令。唯一的问题是，现在 0010h 地址已经保存了一些数据，因此需要把这些数据转移到较高的地址空间中，然后还需要修改那些指向这些地址空间的指令。

试想一下，把操作码和操作数存放在同一个 RAM 阵列并不是一个急于解决的问题。但可以肯定的是，这是一个迟早要解决的问题，不如现在就找一个解决办法吧。在当前

的例子中，也许你更愿意从 0020h 地址开始存放新的指令，并从 0030h 处开始存放新的操作数据。



注意，第一条 Load 指令所指向的地址为 0013h，这个位置保存着第一次运算的结果。

现在，两部分指令的位置分别起始于地址 0000h 和 0020h，而两部分操作数据的地址分别起始于 0010h 和 0030h。我们希望自动加法器从 0000h 开始执行所有指令完成计算任务。

我们必须移除 000Ch 处的 Halt 指令，这里的移除是指用其他代码替换它。但仅仅如此是不够的，问题在于不论我们用什么来替换 Halt 指令，保存在地址 000Ch 的字节都会被当做指令代码。更糟糕的是，从这个位置开始，存储器中每隔 3 个字节的地址：000Fh，0012h，0015h，0018h，001Bh 以及 001Eh，这些地址保存的字节也会被当做指令代码来处理。如果其中的一个字节恰好为 11h(这是 Store 指令的代码)将会发生什么？如果 Store 指令后面的 2 个字节的地址所指向的位置刚好为 0023h，那又会发生什么呢？它导致的结果是，加法器将累加器中的内容写入这个地址，而这个地址中已经保存了重要的数据。即使这些情况都没有发生，加法器从存储器的 001Eh 地址之后取到的下一条指令的位置