这个内存有一个地址输入，一个写的数据输入，以及一个读的数据输出。同寄存器文件一样，从内存中读的操作方式类似于组合逻辑：如果我们在输入 address 上提供一个地址，并将 write 控制信号设置为 0，那么在经过一些延迟之后，存储在那个地址上的值会出现在输出 data 上。如果地址超出了范围，error 信号会设置为 1，否则就设置为 0。写内存是由时钟控制的：我们将 address 设置为期望的地址，将 data in 设置为期望的值，而 write 设置为 1。然后当我们控制时钟时，只要地址是合法的，就会更新内存中指定的位置。对于读操作来说，如果地址是不合法的，error 信号会被设置为 1。这个信号是由组合逻辑产生的，因为所需要的边界检查纯粹就是地址输入的函数，不涉及保存任何状态。

旁注　现实的存储器设计

真实微处理器中的存储器系统比我们在设计中假想的这个简单的存储器要复杂得多。它是由几种形式的硬件存储器组成的，包括几种随机访问存储器和磁盘，以及管理这些设备的各种硬件和软件机制。存储器系统的设计和特点在第 6 章中描述。

不过，我们简单的存储器设计可以用于较小的系统，它提供了更复杂系统的处理器和存储器之间接口的抽象。

我们的处理器还包括另外一个只读存储器，用来读指令。在大多数实际系统中，这两个存储器被合并为一个具有双端口的存储器：一个用来读指令，另一个用来读或者写数据。

4.3　Y86-64 的顺序实现

现在已经有了实现 Y86-64 处理器所需要的部件。首先，我们描述一个称为 SEQ（“sequential” 顺序的）的处理器。每个时钟周期上，SEQ 执行处理一条完整指令所需的所有步骤。不过，这需要一个很长的时钟周期时间，因此时钟周期频率会低到不可接受。我们开发 SEQ 的目标就是提供实现最终目的的第一步，我们的最终目的是实现一个高效的、流水线化的处理器。

4.3.1　将处理组织成阶段

通常，处理一条指令包括很多操作。将它们组织成某个特殊的阶段序列，即使指令的动作差异很大，但所有的指令都遵循统一的序列。每一步的具体处理取决于正在执行的指令。创建这样一个框架，我们就能够设计一个充分利用硬件的处理器。下面是关于各个阶段以及各阶段内执行操作的简略描述：

- **取指**(fetch)：取指阶段从内存读取指令字节，地址为程序计数器(PC)的值。从指令中抽取出指令指示符字节的两个四位部分，称为 icode(指令代码)和 ifun(指令功能)。它可能取出一个寄存器指示符字节，指明一个或两个寄存器操作数指示符 rA 和 rB。它还可能取出一个四字节常数字 valC。它按顺序方式计算当前指令的下一条指令的地址 valP。也就是说，valP 等于 PC 的值加上已取出指令的长度。
- **译码**(decode)：译码阶段从寄存器文件读入最多两个操作数，得到值 valA 和/或 valB。通常，它读入指令 rA 和 rB 字段指明的寄存器，不过有些指令是读寄存器%rsp 的。
- **执行**(execute)：在执行阶段，算术/逻辑单元(ALU)要么执行指令指明的操作(根据 ifun 的值)，计算内存引用的有效地址，要么增加或减少栈指针。得到的值我们称为 valE。在此，也可能设置条件码。对一条条件传送指令来说，这个阶段会检验条件码和传送条件(由 ifun 给出)，如果条件成立，则更新目标寄存器。同样，