在一个典型的 x86 实现中，一条只对寄存器操作的指令，例如

```
addq %rax,%rdx
```

会被转化成一个操作。另一方面，一条包括一个或者多个内存引用的指令，例如

```
addq %rax,8(%rdx)
```

会产生多个操作，把内存引用和算术运算分开。这条指令会被译码成为三个操作：一个操作从内存中加载一个值到处理器中，一个操作将加载进来的值加上寄存器%rax 中的值，而一个操作将结果存回到内存。这种译码逻辑对指令进行分解，允许任务在一组专门的硬件单元之间进行分割。这些单元可以并行地执行多条指令的不同部分。

EU 接收来自取指单元的操作。通常，每个时钟周期会接收多个操作。这些操作会被分派到一组功能单元中，它们会执行实际的操作。这些功能单元专门用来处理不同类型的操作。

读写内存是由加载和存储单元实现的。加载单元处理从内存读数据到处理器的操作。这个单元有一个加法器来完成地址计算。类似，存储单元处理从处理器写数据到内存的操作。它也有一个加法器来完成地址计算。如图中所示，加载和存储单元通过数据高速缓存(data cache)来访问内存。数据高速缓存是一个高速存储器，存放着最近访问的数据值。

使用投机执行技术对操作求值，但是最终结果不会存放在程序寄存器或数据内存中，直到处理器能确定应该实际执行这些指令。分支操作被送到 EU，不是确定分支该往哪里去，而是确定分支预测是否正确。如果预测错误，EU 会丢弃分支点之后计算出来的结果。它还会发信号给分支单元，说预测是错误的，并指出正确的分支目的。在这种情况中，分支单元开始在新的位置取指。如在 3.6.6 节中看到的，这样的预测错误会导致很大的性能开销。在可以取出新指令、译码和发送到执行单元之前，要花费一点时间。

图 5-11 说明不同的功能单元被设计来执行不同的操作。那些标记为执行“算术运算”的单元通常是专门用来执行整数和浮点数操作的不同组合。随着时间的推移，在单个微处理器芯片上能够集成的晶体管数量越来越多，后续的微处理器型号都增加了功能单元的数量以及每个单元能执行的操作组合，还提升了每个单元的性能。由于不同程序间所要求的操作变化很大，因此，算术运算单元被特意设计成能够执行各种不同的操作。比如，有些程序也许会涉及整数操作，而其他则要求许多浮点操作。如果一个功能单元专门执行整数操作，而另一个只能执行浮点操作，那么，这些程序就没有一个能够完全得到多个功能单元带来的好处了。

举个例子，我们的 Intel Core i7 Haswell 参考机有 8 个功能单元，编号为 0～7。下面部分列出了每个单元的功能：

0：整数运算、浮点乘、整数和浮点数除法、分支
1：整数运算、浮点加、整数乘、浮点乘
2：加载、地址计算
3：加载、地址计算
4：存储
5：整数运算
6：整数运算、分支
7：存储、地址计算

在上面的列表中，“整数运算”是指基本的操作，比如加法、位级操作和移位。乘法