和除法需要更多的专用资源。我们看到存储操作要两个功能单元——一个计算存储地址，一个实际保存数据。5.12 节将讨论存储(和加载)操作的机制。

我们可以看出功能单元的这种组合具有同时执行多个同类型操作的潜力。它有 4 个功能单元可以执行整数操作，2 个单元能执行加载操作，2 个单元能执行浮点乘法。稍后我们将看到这些资源对程序获得最大性能所带来的影响。

在 ICU 中，*退役单元*(retirement unit)记录正在进行的处理，并确保它遵守机器级程序的顺序语义。我们的图中展示了一个*寄存器文件*，它包含整数、浮点数和最近的 SSE 和 AVX 寄存器，是退役单元的一部分，因为退役单元控制这些寄存器的更新。指令译码时，关于指令的信息被放置在一个先进先出的队列中。这个信息会一直保持在队列中，直到发生以下两个结果中的一个。首先，一旦一条指令的操作完成了，而且所有引起这条指令的分支点也都被确认为预测正确，那么这条指令就可以*退役*(retired)了，所有对程序寄存器的更新都可以被实际执行了。另一方面，如果引起该指令的某个分支点预测错误，这条指令会被*清空*(flushed)，丢弃所有计算出来的结果。通过这种方法，预测错误就不会改变程序的状态了。

正如我们已经描述的那样，任何对程序寄存器的更新都只会在指令退役时才会发生，只有在处理器能够确信导致这条指令的所有分支都预测正确了，才会这样做。为了加速一条指令到另一条指令的结果的传送，许多此类信息是在执行单元之间交换的，即图中的“操作结果”。如图中的箭头所示，执行单元可以直接将结果发送给彼此。这是 4.5.5 节中简单处理器设计中采用的数据转发技术的更复杂精细版本。

控制操作数在执行单元间传送的最常见的机制称为*寄存器重命名*(register renaming)。当一条更新寄存器 r 的指令译码时，产生标记 t，得到一个指向该操作结果的唯一的标识符。条目(r, t)被加入到一张表中，该表维护着每个程序寄存器 r 与会更新该寄存器的操作的标记 t 之间的关联。当随后以寄存器 r 作为操作数的指令译码时，发送到执行单元的操作会包含 t 作为操作数源的值。当某个执行单元完成第一个操作时，会生成一个结果(v, t)，指明标记为 t 的操作产生值 v。所有等待 t 作为源的操作都能使用 v 作为源值，这就是一种形式的数据转发。通过这种机制，值可以从一个操作直接转发到另一个操作，而不是写到寄存器文件再读出来，使得第二个操作能够在第一个操作完成后尽快开始。重命名表只包含关于有未进行写操作的寄存器条目。当一条被译码的指令需要寄存器 r，而又没有标记与这个寄存器相关联，那么可以直接从寄存器文件中获取这个操作数。有了寄存器重命名，即使只有在处理器确定了分支结果之后才能更新寄存器，也可以预测着执行操作的整个序列。

旁注　乱序处理的历史

乱序处理最早是在 1964 年 Control Data Corporation 的 6600 处理器中实现的。指令由十个不同的功能单元处理，每个单元都能独立地运行。在那个时候，这种时钟频率为 10Mhz 的机器被认为是科学计算最好的机器。

在 1966 年，IBM 首先是在 IBM 360/91 上实现了乱序处理，但只是用来执行浮点指令。在大约 25 年的时间里，乱序处理都被认为是一项异乎寻常的技术，只在追求尽可能高性能的机器中使用，直到 1990 年 IBM 在 RS/6000 系列工作站中重新引入了这项技术。这种设计成为了 IBM/Motorola PowerPC 系列的基础，1993 年引入的型号 601，它成为第一个使用乱序处理的单芯片微处理器。Intel 在 1995 年的 PentiumPro 型号中引入了乱序处理，PentiumPro 的底层微体系结构类似于我们的参考机。