让我们来看看这种处理异常的方法是怎样解决刚才提到的那些细节问题的。当流水线中有一个或多个阶段出现异常时，信息只是简单地存放在流水线寄存器的状态字段中。异常事件不会对流水线中的指令流有任何影响，除了会禁止流水线中后面的指令更新程序员可见的状态(条件码寄存器和内存)，直到异常指令到达最后的流水线阶段。因为指令到达写回阶段的顺序与它们在非流水线化的处理器中执行的顺序相同，所以我们可以保证第一条遇到异常的指令会第一个到达写回阶段，此时程序执行会停止，流水线寄存器 W 中的状态码会被记录为程序状态。如果取出了某条指令，过后又取消了，那么所有关于这条指令的异常状态信息也都会被取消。所有导致异常的指令后面的指令都不能改变程序员可见的状态。携带指令的异常状态以及所有其他信息通过流水线的简单原则是处理异常的简单而可靠的机制。

4.5.7　PIPE 各阶段的实现

现在我们已经创建了 PIPE 的整体结构，PIPE 是我们使用了转发技术的流水线化的 Y86-64 处理器。它使用了一组与前面顺序设计相同的硬件单元，另外增加了一些流水线寄存器、一些重新配置了的逻辑块，以及增加的流水线控制逻辑。在本节中，我们将浏览各个逻辑块的设计，而将流水线控制逻辑的设计放到下一节中介绍。许多逻辑块与 SEQ 和 SEQ+中相应部件完全相同，除了我们必须从来自不同流水线寄存器(用大写的流水线寄存器的名字作为前缀)或来自各个阶段计算(用小写的阶段名字的第一个字母作为前缀)的信号中选择适当的值。

作为一个示例，比较一下 SEQ 中产生 srcA 信号的逻辑的 HCL 代码与 PIPE 中相应的代码：

```
    # Code from SEQ

word srcA = [
        icode in { IRRMOVQ, IRMMOVQ, IOPQ, IPUSHQ  } : rA;
        icode in { IPOPQ, IRET } : RRSP;
        1 : RNONE; # Don't need register
];

    # Code from PIPE

word d_srcA = [
        D_icode in { IRRMOVQ, IRMMOVQ, IOPQ, IPUSHQ  } : D_rA;
        D_icode in { IPOPQ, IRET } : RRSP;
        1 : RNONE; # Don't need register
];
```

它们的不同之处只在于 PIPE 信号都加上了前缀：“D_”表示源值，以表明信号是来自流水线寄存器 D，而“d_”表示结果值，以表明它是在译码阶段中产生的。为了避免重复，我们在此就不列出那些与 SEQ 中代码只有名字前缀不同的块的 HCL 代码。网络旁注 ARCH:HCL 中列出了完整的 PIPE 的 HCL 代码。

1. PC 选择和取指阶段

图 4-57 提供了 PIPE 取指阶段逻辑的一个详细描述。像前面讨论过的那样，这个阶段必须选择程序计数器的当前值，并且预测下一个 PC 值。用于从内存中读取指令和抽取不同指令字段的硬件单元与 SEQ 中考虑的那些一样(参见 4.3.4 节中的取指阶段)。