

合并信号 `valA` 和 `valP` 的依据是, 只有 `call` 和跳转指令在后面的阶段中需要 `valP` 的值, 而这些指令并不需要从寄存器文件 `A` 端口中读出的值。这个选择是由该阶段的 `icode` 信号来控制的。当信号 `D_icode` 与 `call` 或 `jXX` 的指令代码相匹配时, 这个块就会选择 `D_valP` 作为它的输出。

4.5.5 节中提到有 5 个不同的转发源, 每个都有一个数据字和一个目的寄存器 ID:


数据字	寄存器 ID	源描述
<code>e_valE</code>	<code>e_dstE</code>	ALU 输出
<code>m_valM</code>	<code>M_dstM</code>	内存输出
<code>M_valE</code>	<code>M_dstE</code>	访存阶段中对端口 <code>E</code> 未进行的写
<code>W_valM</code>	<code>W_dstM</code>	写回阶段中对端口 <code>M</code> 未进行的写
<code>W_valE</code>	<code>W_dstE</code>	写回阶段中对端口 <code>E</code> 未进行的写

如果不满足任何转发条件, 这个块就应该选择 `d_rvalA` 作为它的输出, 也就是从寄存器端口 `A` 中读出的值。

综上所述, 我们得到以下流水线寄存器 `E` 的 `valA` 新值的 HCL 描述:

```
word d_valA = [
    D_icode in { ICALL, IJXX } : D_valP; # Use incremented PC
    d_srcA == e_dstE : e_valE;          # Forward valE from execute
    d_srcA == M_dstM : m_valM;          # Forward valM from memory
    d_srcA == M_dstE : M_valE;          # Forward valE from memory
    d_srcA == W_dstM : W_valM;          # Forward valM from write back
    d_srcA == W_dstE : W_valE;          # Forward valE from write back
    1 : d_rvalA; # Use value read from register file
];
```

上述 HCL 代码中赋予这 5 个转发源的优先级是非常重要的。这种优先级是由 HCL 代码中检测 5 个目的寄存器 ID 的顺序来确定的。如果选择了其他任何顺序, 对某些程序来说, 流水线就会出错。图 4-59 给出了一个程序示例, 要求对执行和访存阶段中的转发源设置正确的优先级。在这个程序中, 前两条指令写寄存器 `%rdx`, 而第三条指令用这个寄存器作为它的源操作数。当指令 `rrmovq` 在周期 4 到达译码阶段时, 转发逻辑必须在两个都以该源寄存器为目的的值中选择一个。它应该选择哪一个呢? 为了设定优先级, 我们必须考虑当一次执行一条指令时, 机器语言程序的行为。第一条 `irmovq` 指令会将寄存器 `%rdx` 设为 10, 第二条 `irmovq` 指令会将之设为 3, 然后 `rrmovq` 指令会从 `%rdx` 中读出 3。为了模拟这种行为, 流水线化的实现应该总是给处于最早流水线阶段中的转发源以较高的优先级, 因为它保持着程序序列中设置该寄存器的最近的指令。因此, 上述 HCL 代码中的逻辑首先会检测执行阶段中的转发源, 然后是访存阶段, 最后才是写回阶段。只有指令 `popq %rsp` 会关心在访存或写回阶段中的两个源之间的转发优先级, 因为只有这条指令能同时写两个寄存器。

 **练习题 4.32** 假设 `d_valA` 的 HCL 代码中第三和第四种情况(来自访存阶段的两个转发源)的顺序是反过来的。请描述下列程序中 `rrmovq` 指令(第 5 行)造成的行为:

```
1    irmovq $5, %rdx
2    irmovq $0x100, %rsp
3    rrmovq %rdx, 0(%rsp)
4    popq %rsp
5    rrmovq %rsp, %rax
```