



Imagination大学计划

# RVfpga实验15

## 数据冒险

## 1. 简介

在本实验中，我们将处理**数据冒险**。正如Hennessy和Patterson在其《计算机体系结构：量化研究方法》[HePa]一书的第6版中所述：当流水线更改操作数读/写访问顺序致使此顺序与在非流水线处理器上顺序执行指令的顺序不同时，将发生数据冒险。假设指令*i*在程序中位于指令*j*之前并且两条指令均使用寄存器*x*。*i*和*j*之间可能发生三种类型的数据冒险：

- **写后读（Read After Write, RAW）数据冒险**：这是最常见类型的冒险。当指令*j*先读取寄存器*x*，指令*i*后写入寄存器*x*时，将发生RAW冒险。因此，指令*j*将使用错误的*x*值。
- **读后写（Write After Read, WAR）数据冒险**：当指令*j*写入*x*、指令*i*读取*x*且指令*j*顺序调整为在*i*之前发生时，将发生WAR冒险。因此，指令*i*将读取错误的*x*值。这种冒险只在调整指令顺序时发生（在SweRV EH1中很少发生）；具体来说，SweRV EH1中从未发生过WAR冒险。
- **写后写（Write After Write, WAW）数据冒险**：当调整指令顺序致使指令*j*先写入*x*，指令*i*后写入*x*时，将发生WAW冒险。这种冒险只在调整指令顺序时发生（在SweRV EH1中很少发生）；但是，在非阻塞加载的情况下，也可能会发生WAW冒险，我们将在本实验的后面部分进行分析。

在以下部分中，我们将分析SweRV EH1处理器中如何解除RAW数据冒险，然后介绍与RAW冒险相关的任务和练习，还将通过一个练习分析发生WAW冒险时的情况。

**注：**在分析SweRV EH1数据冒险逻辑之前，我们建议阅读DDCARV中的第7.5节，了解如何在流水线处理器中解除冒险。数据冒险具体在第7.5.3节中进行分析。尽管本书中展示的流水线处理器比SweRV EH1简单，但两种处理器中解除数据冒险的方法类似。

## 2. 在译码阶段通过转发解除数据冒险

如DDCARV的第7.5.3节所述，可以通过将结果从后期流水线阶段执行的指令转发（也称为旁路）到早期流水线阶段执行的相关指令来解除一些RAW数据冒险。这需要在功能单元（ALU、乘法器、计算DC1中有效地址的加法器等）前面添加多路开关，以从寄存器文件或后续阶段选择其操作数。

图1使用旁路值扩展了实验11的图4中所示的译码阶段。转发逻辑为每个通路中的两个源操作数中的每一个生成旁路（即转发）：

- **通路0：**
  - 第一个输入操作数：i0\_rs1\_bypass\_data\_d[31:0]
  - 第二个输入操作数：i0\_rs2\_bypass\_data\_d[31:0]

- **通路1:**

- 第一个输入操作数: `i1_rs1_bypass_data_d[31:0]`
- 第二个输入操作数: `i1_rs2_bypass_data_d[31:0]`

这四个输入分配到**3:1**和**4:1**多路开关，这两个多路开关确定每个执行阶段流水线路径的输入操作数。为清楚起见，图1中的信号按名称连接。转发逻辑的输入是更后期流水线中先前程序指令生成的结果，如下所示。

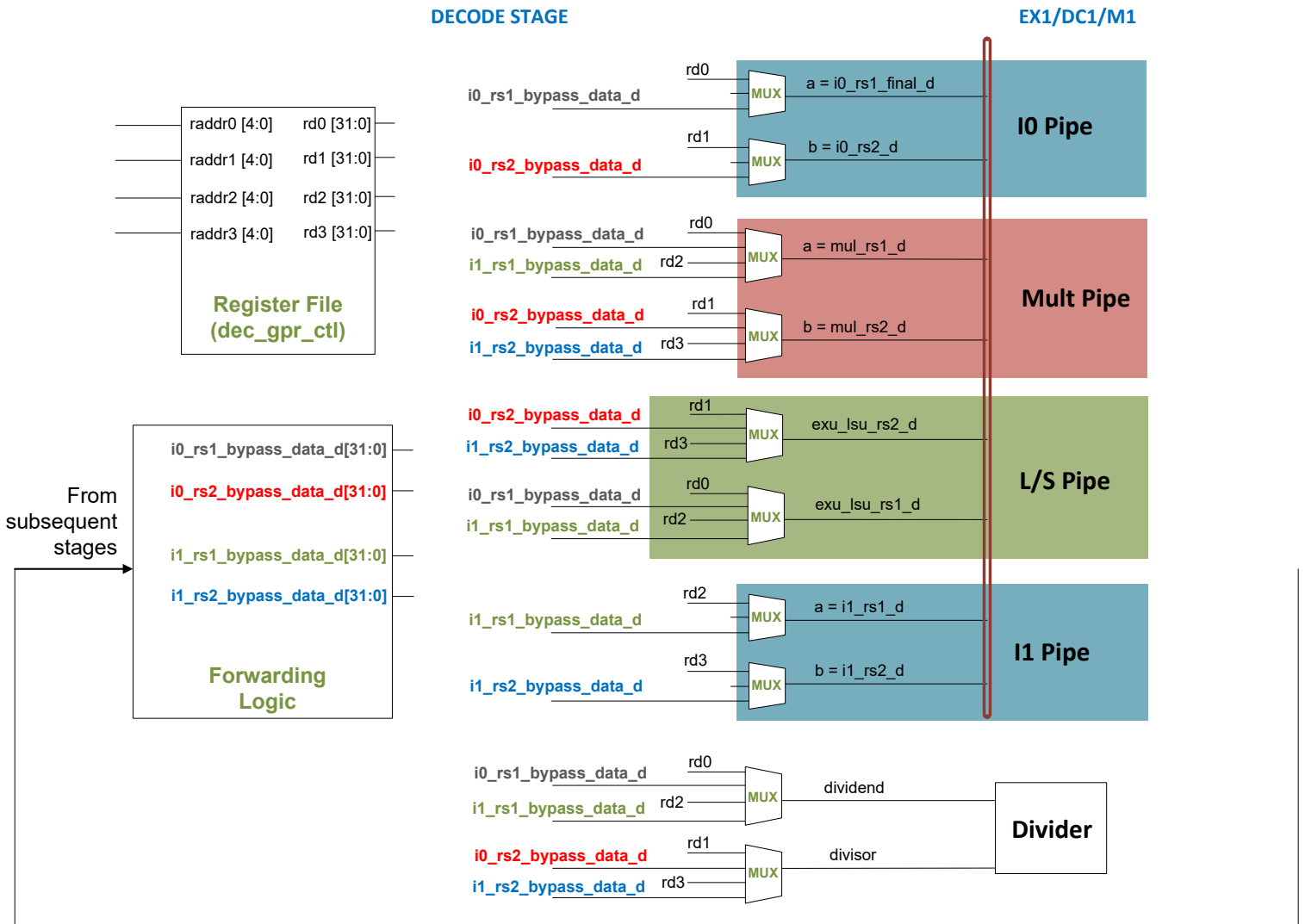


图1. 功能单元的旁路输入

SweRV EH1处理器中存在许多转发路径。在本部分中，我们将重点关注特定路径并加以详细分析。然后在任务和练习中检查其他情况。我们将分析两条相关**A-L**指令同时执行的情况以及如何解除**RAW**数据冒险。正如在实验12和13中一样，我们将先从基础研究（第2.A部分）开始，然后再进行高级分析（第2.B部分）。可以选择仅完成基本部分，也可选择完成两个部分。

我们将使用图2给出的示例，该示例执行重复0xFFFF次迭代的循环中包含的两条add指令。第一条add指令将一个值写入t4，第二条add指令使用t4作为其第二个输入操作数。这两条add指令之间将插入一条独立add指令（add t6, t6, -1，用于更新循环索引的指令）以强制相关add指令使用处理器的相同通路。

```
.globl Test_Assembly

.text
Test_Assembly:

li t3, 0x3
li t4, 0x2
li t5, 0x1
li t6, 0xFFFF

REPEAT:
    INSERT_NOPS_8
    add t4, t4, t5          # t4 = t4 + t5 (t4 = 2 + 1)
    add t6, t6, -1
    add t3, t3, t4          # t3 = t3 + t4 (t3 = 3 + 3)
    INSERT_NOPS_9
    li t3, 0x3
    li t4, 0x2
    li t5, 0x1
    bne t6, zero, REPEAT    # Repeat the loop

.end
```

图2. 两条add指令之间的RAW数据冒险

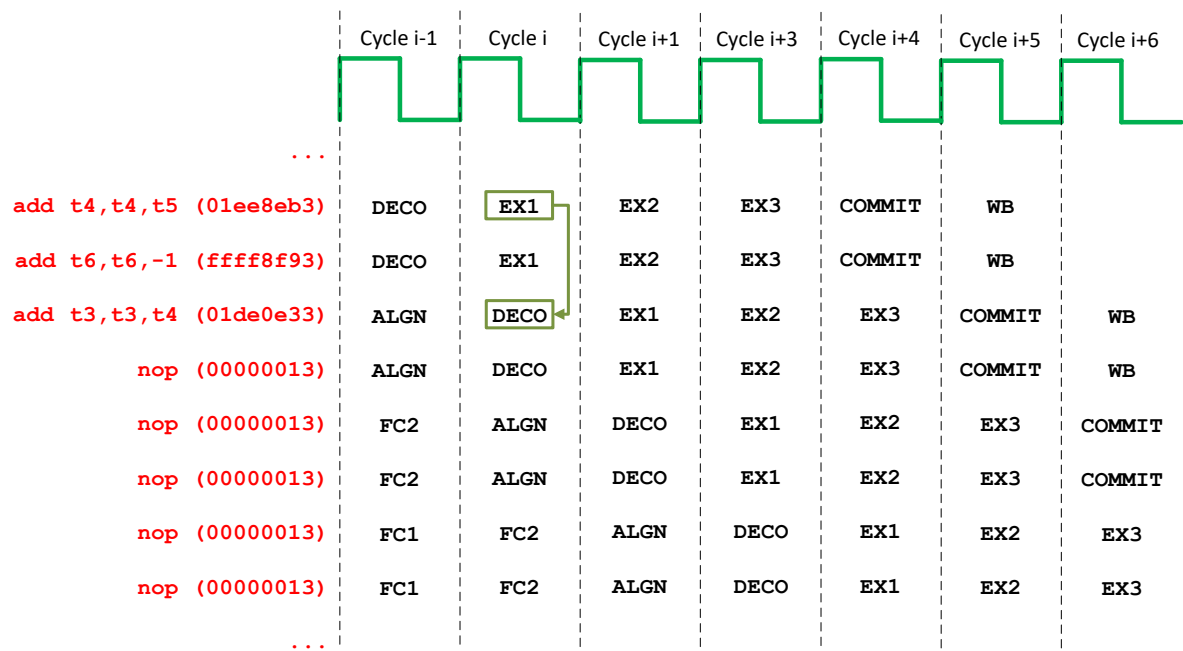
文件夹[RVfpgaPath]/RVfpga/Labs/Lab15/DataHazards\_AL-AL提供PlatformIO项目，以便可以根据需要分析、仿真和修改程序。在PlatformIO中打开、编译项目，然后打开反汇编文件（位于[RVfpgaPath]/RVfpga/Labs/Lab15/DataHazards\_AL-AL/.pio/build/swervolf\_nexys/firmware.dis中），则会发现所分析的两条add指令位于地址0x000001A0和0x000001A8处：

0x000001a0:	01ee8eb3	add t4,t4,t5
0x000001a4:	ffff8f93	addi t6,t6,-1
0x000001a8:	01de0e33	add t3,t3,t4

## A. A-L指令之间RAW数据冒险的基本分析

在我们分析的示例中，第二条add指令（add t3,t3,t4）需要使用第一条add指令（add t4,t4,t5）的结果作为其第二个输入操作数。此结果在EX1阶段提供，可以从此阶段旁路到译码阶段，供第二条add指令使用。在我们的示例（图2）中，所有迭代都是相等的，t4最初为2，第一个加法后为3。最后这个值（3）是第二个加法必须用作其第二个输入操作数的值，而不是从寄存器文件读取的值（在第一条add指令到达回写阶段并进行更新之前为2）。

图3所示为图2示例中循环的任意一次迭代期间通过SweRV EH1流水线的指令流。在周期中，在I0管道的EX1阶段计算的值必须转发到处于通路0译码阶段的指令，因为所分析的两条add指令之间存在RAW数据冒险。



**图3. 图2示例代码的执行。转发在周期*i*中执行。**

图4 所示为图3的周期*i*期间SweRV EH1通路0的译码和EX1阶段。在此周期中，第一条add指令（add t4, t4, t5）处于EX1阶段，第二条add指令（add t3, t3, t4）处于译码阶段。如图所示，第一条add指令的结果旁路到译码阶段，它由转发逻辑（我们将在下一部分详细分析）选择，然后用作第二条add指令的第二个输入操作数。

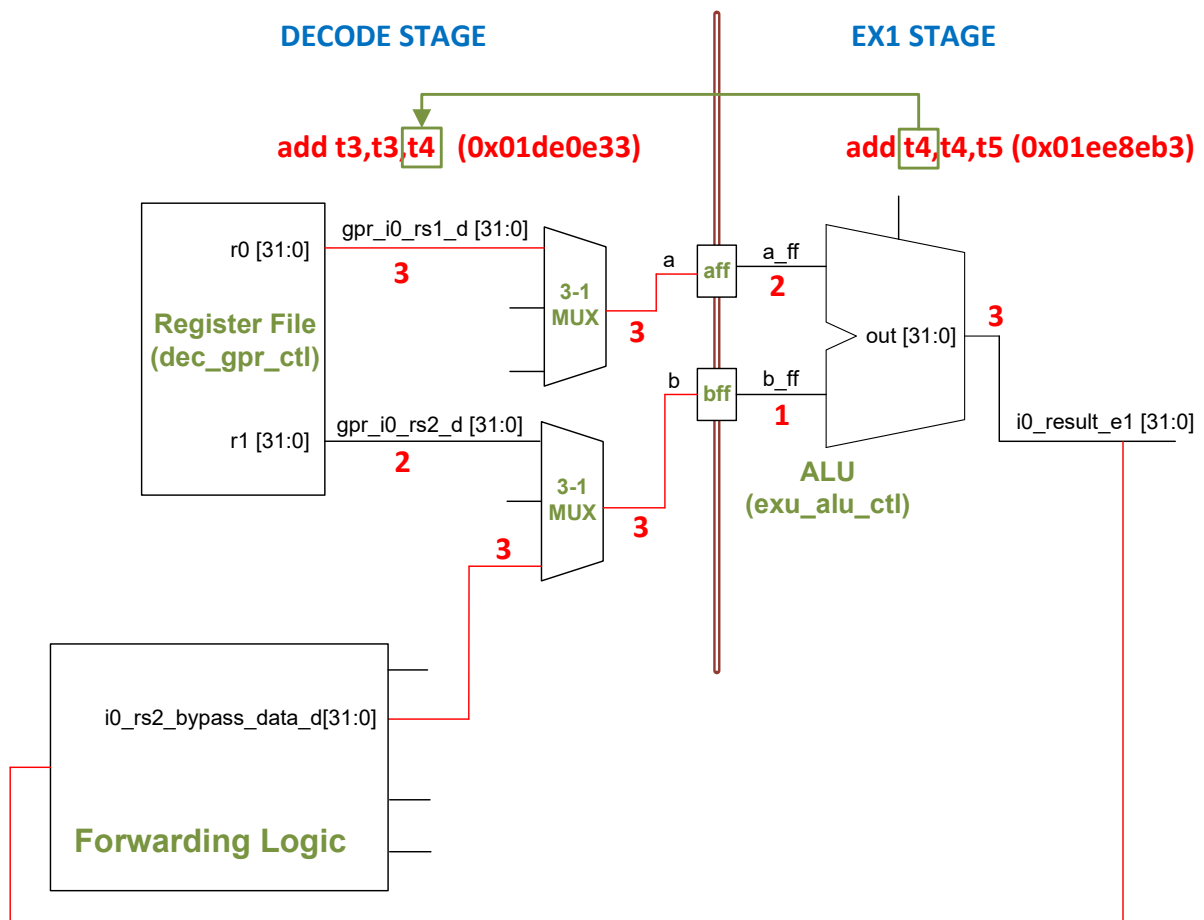


图4. 结果从通路0的EX1阶段转发到译码阶段（第二个操作数）

最后，图5给出了图2中的程序在图3的周期*i*和*i+1*期间的仿真情况。

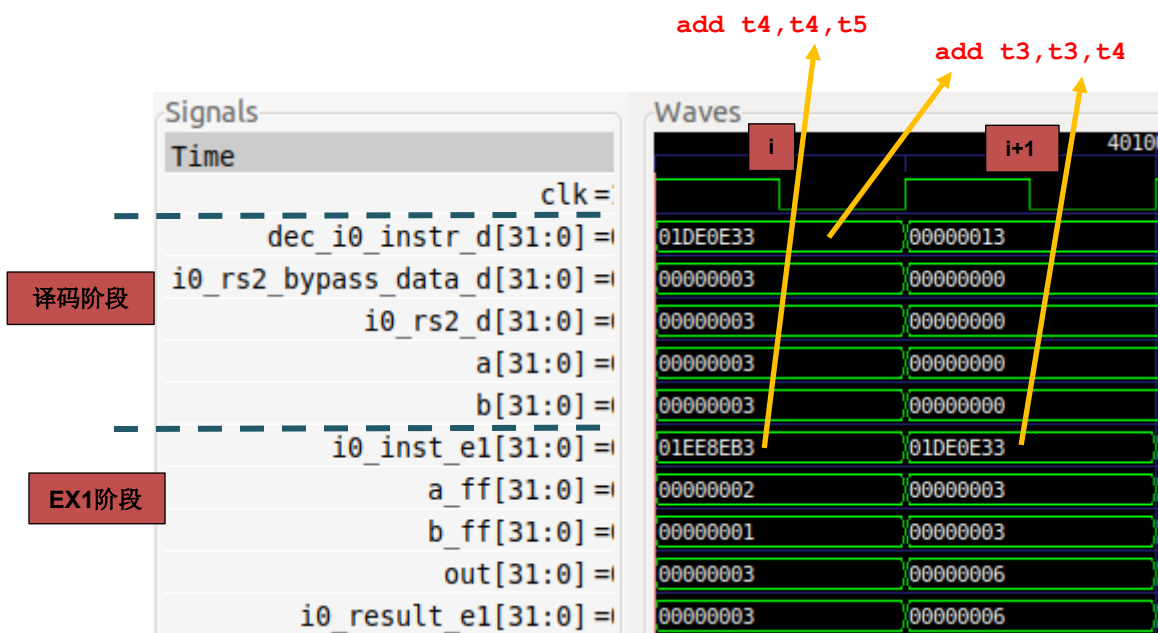


图5. 图2示例代码的仿真

**任务：** 在自己的计算机上重复图5中的仿真过程。可以使用以下位置提供的.tcl文件：  
[RVfpgaPath]/RVfpga/Labs/Lab15/DataHazards\_AL-AL/test\_Basic.tcl。

同时分析图5中的仿真以及图4中的图。

- 指令 `add t4,t4,t5 (0x01ee8eb3)` :
  - o 在周期*i*中，此指令处于I0管道的EX1阶段 (`i0_inst_e1 = 0x01ee8eb3`)。它在ALU中计算以下加法：
 
$$a\_ff(2) + b\_ff(1) = out(3)$$
 加法的结果在译码阶段作为输入提供给转发逻辑，如图4所示。
- 指令 `add t3,t3,t4 (0x01de0e33)` :
  - o 在周期*i*中，此指令处于通路0的译码阶段 (`dec_i0_instr_d = 0x01de0e33`)。转发逻辑将 `i0_result_e1` 与 `i0_rs2_bypass_data_d` 连接。两个3:1多路开关为下一个周期（周期*i+1*）在I0管道的EX1阶段计算的加法选择输入操作数；具体如下：
 
$$a = 3 \text{ (来自寄存器文件)}$$

$$b = 3 \text{ (来自I0管道EX1阶段的ALU输出, 经过转发逻辑的信号)}$$

$$i0\_rs2\_bypass\_data\_d$$
  - o 在周期*i+1*中，此指令处于I0管道的EX1阶段 (`i0_inst_e1 = 0x01de0e33`)。它在ALU中计算以下加法：
 
$$a\_ff(3) + b\_ff(3) = out(6)$$

**任务：** 删除图2示例中的所有nop指令。对于循环的两次连续迭代，绘制与图3类似的图，然后通过将其与Verilator仿真进行比较来分析并确认此图是否正确，最后在板上执行程序时使用性能计数器计算IPC。

**任务：** 在图2的示例中，删除所有nop指令并将 `add t6,t6,-1` 指令移至 `add t3,t3,t4` 指令之后，然后重新检查仿真中以及板上的程序。在这个调整顺序的程序中，两条相关add指令 (`add t4,t4,t5` 和 `add t3,t3,t4`) 在同一周期到达译码阶段，这将影响性能。通过仿真和板上执行来说明这些变化的影响。

测试将相关add指令替换为其他相关指令时的类似情况，例如：

- `add t4,t4,t5`  
`mul t3,t3,t4`
- `add t4,t4,t5`  
`div t3,t3,t4`
- `add t4,t4,t5`  
`lw t3, 0(t4)`

## B. A-L指令之间RAW数据冒险的高级分析

### i. 理论说明

图6通过添加一个10:1多路开关（在图6中用蓝框包围）对图1和图4中的图进行了扩展，此多路开关生成信号*i0\_rs2\_bypass\_data\_d*，此信号在图2的示例中为第二条add指令（add t3,t3,t4）提供第二个输入操作数。此10:1多路开关在转发逻辑框内实现，如图1和图4所示。

这张图还给出了此10:1多路开关的输入连接。旁路的值可来自通过通路0或通路1执行的指令。因此，每路需要五个转发路径。具体来说，10:1多路开关的输入可来自通路0或通路1的任何后续阶段（EX1、EX2、EX3、提交和回写）。为简单起见，我们用线连接来自通路0的5个输入，而来自通路1的5个输入按名称连接。

转发逻辑内部的三个附加10:1多路开关计算其他三个源操作数：信号*i0\_rs1\_bypass\_data\_d*、*i1\_rs1\_bypass\_data\_d*和*i1\_rs2\_bypass\_data\_d*。但是，我们没有在图中显示这三个信号，因为它们并未在本部分分析的示例中使用（图2）。全部四个多路开关均可在模块**dec\_decode\_ctl**的第2429-2473行中找到。



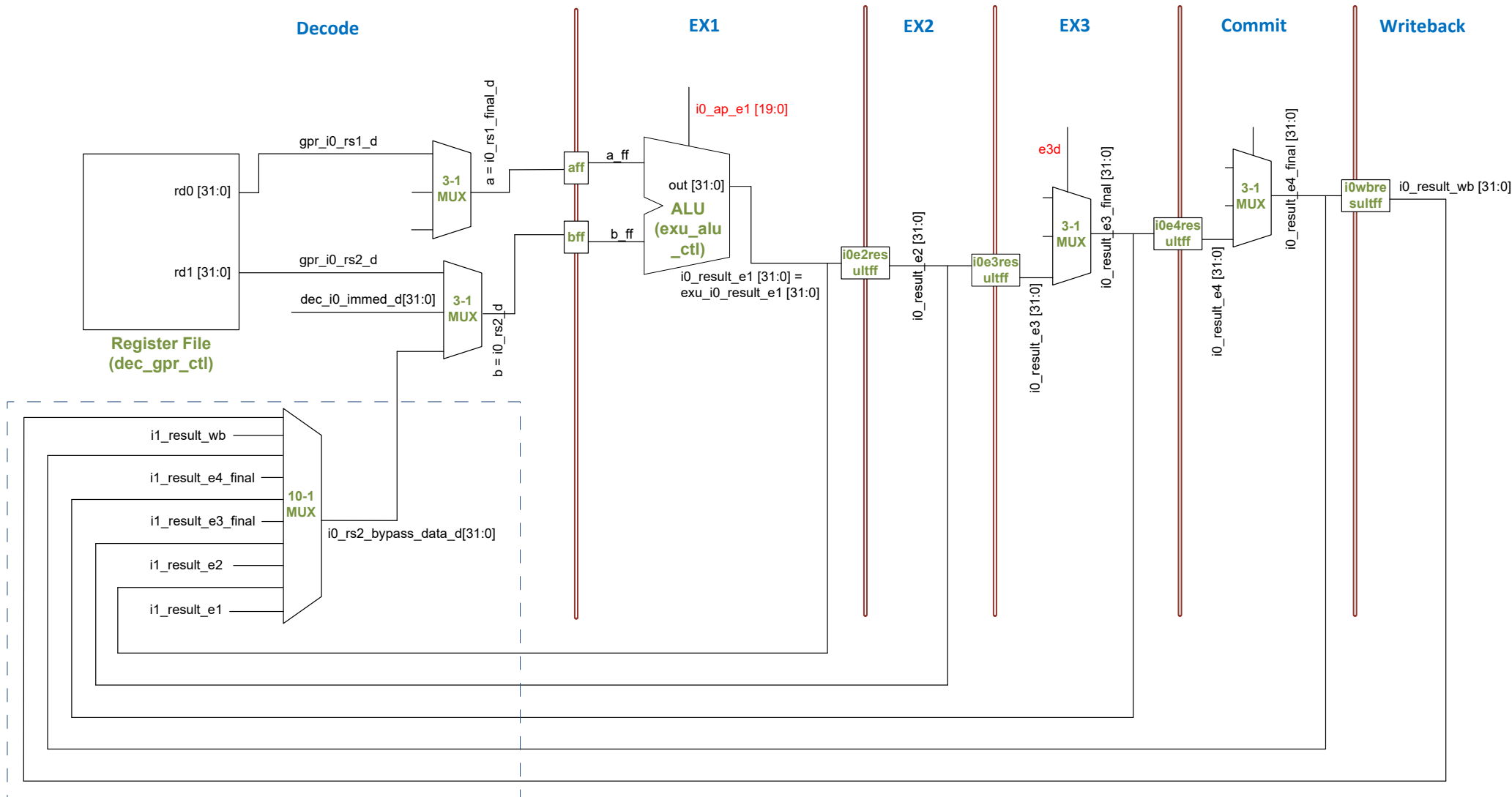


图6. 包含ALU第二个输入源所用转发逻辑的I/O管道

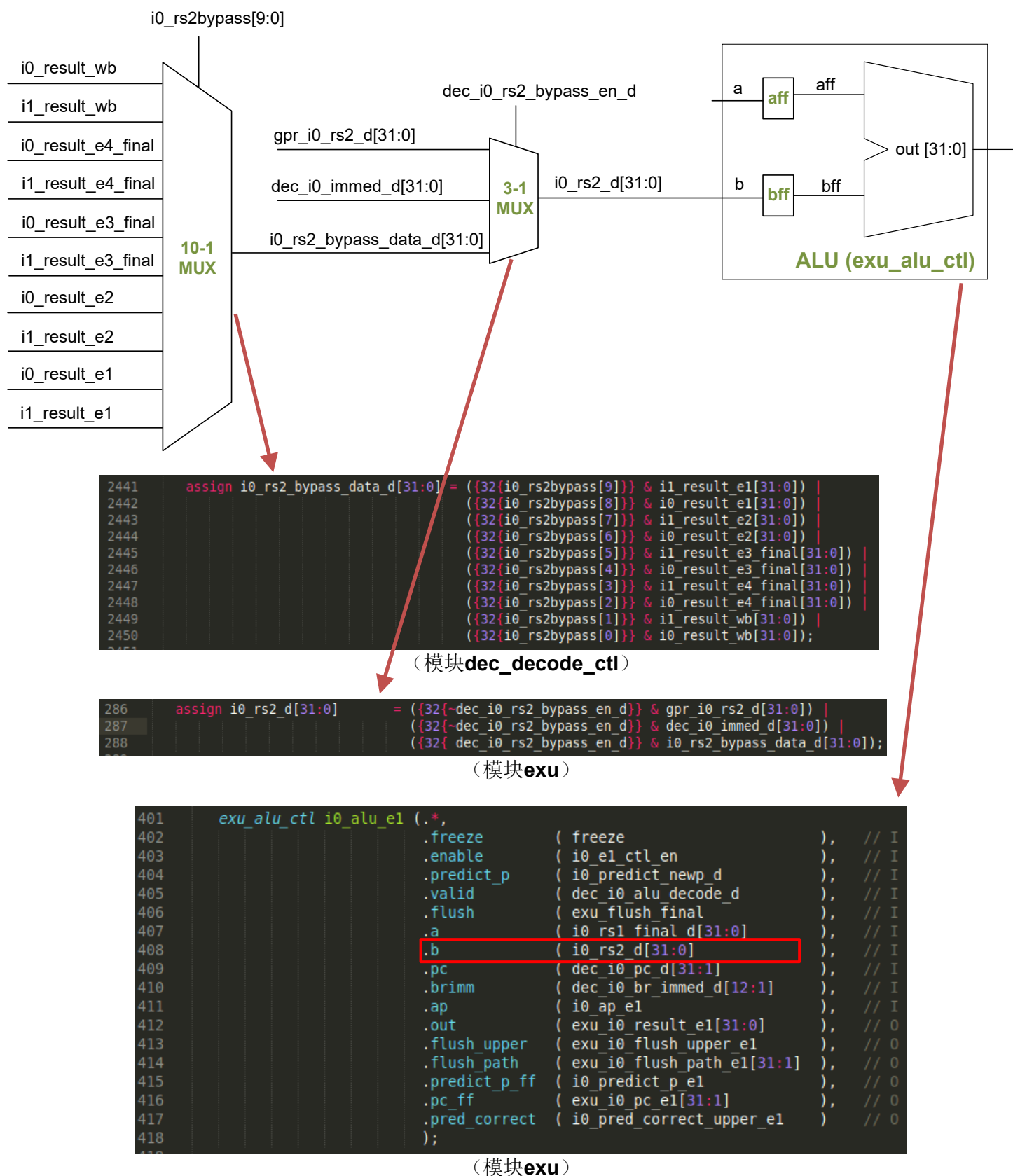


图7. 图6中突出显示的10:1和3:1多路开关

图7对图6的两个多路开关（10:1和3:1多路开关）进行了放大，这两个多路开关计算I0管道ALU的第二个输入操作数（b）。图7给出了在模块dec\_decode\_ctl和exu中实现这两个多路开关的框图和Verilog代码。

**注：**DDCARV所述的处理器中也包含图7中的两个多路开关。数据转发到此处理器中的执行阶段，并且转发路径较少，因为此处理器不是超标量处理器并且流水线较短。可以分析DDCARV的图7.55中的转发路径。

接下来分析图7所示的两个多路开关的输入、输出和控制信号。

### **10:1多路开关：**

**输出：**10:1多路开关的输出为i0\_rs2\_bypass\_data\_d[31:0]。此信号包含必须转发（旁路）到译码阶段指令的值。

**输入：**10:1多路开关的输入是程序中在后面阶段（EX1、EX2、EX3、提交或回写）执行的先前指令的结果。其中五个信号来自I0管道（如图6所示），其他五个信号来自I1管道（图6中未显示），因为译码阶段的指令可能与两个通路中任一通路中执行的指令相关。

**控制信号：**控制信号（i0\_rs2bypass[9:0]）选择与多路开关的输出连接的输入。它由10位组成，同一时间最多有一位为高电平（如果没有数据冒险，它们均可以为0）。多路开关的工作方式如下：

- If i0\_rs2bypass[9] == 1 → i0\_rs2\_bypass\_data\_d = i1\_result\_e1
- If i0\_rs2bypass[8] == 1 → i0\_rs2\_bypass\_data\_d = i0\_result\_e1
- If i0\_rs2bypass[7] == 1 → i0\_rs2\_bypass\_data\_d = i1\_result\_e2
- If i0\_rs2bypass[6] == 1 → i0\_rs2\_bypass\_data\_d = i0\_result\_e2
- If i0\_rs2bypass[5] == 1 → i0\_rs2\_bypass\_data\_d = i1\_result\_e3\_final
- If i0\_rs2bypass[4] == 1 → i0\_rs2\_bypass\_data\_d = i0\_result\_e3\_final
- If i0\_rs2bypass[3] == 1 → i0\_rs2\_bypass\_data\_d = i1\_result\_e4\_final
- If i0\_rs2bypass[2] == 1 → i0\_rs2\_bypass\_data\_d = i0\_result\_e4\_final
- If i0\_rs2bypass[1] == 1 → i0\_rs2\_bypass\_data\_d = i1\_result\_wb
- If i0\_rs2bypass[0] == 1 → i0\_rs2\_bypass\_data\_d = i0\_result\_wb

为了解此10位控制信号如何计算，我们将说明信号i0\_rs2bypass[8]的计算过程，它是图2的示例中由于add-add旁路而变为高电平的信号。

- 如果i0\_rs2bypass[8]为1，则选择的旁路值为i0\_result\_e1，即I0管道EX1阶段执行的指令的结果（参见图6）。
- 要使EX1将数据转发到译码阶段（均在I0管道中），必须满足以下条件（有关控制信号的信息，请参见SweRVref文档的第4部分）：

- 译码阶段指令的第二个输入操作数从寄存器文件中读取，而不是从寄存器0中读取。在SweRV EH1控制单元中，当dec\_i0\_rs2\_en\_d为1时，将出现这种情况。

相应的Verilog代码如下：

```
dec_i0_rs2_en_d = i0_dp.rs2 & (i0r.rs2[4:0] != 5'd0);
```

- I0管道EX1阶段的指令有效：

```
e1d.i0v == 1
```

- I0管道EX1阶段指令的目标寄存器和通路0译码阶段指令的第二个源寄存器相同：

```
e1d.i0rd[4:0] == i0r.rs2[4:0]
```

- I0管道EX1阶段的指令为ALU操作：

```
i0_rs2_class_d.alu == 1
```

- 考虑到上述所有情况，可以得出以下结论：

```
i0_rs2bypass[8] =
    (i0_dp.rs2 & (i0r.rs2[4:0] != 5'd0))    &
    e1d.i0v                                  &
    (e1d.i0rd[4:0] == i0r.rs2[4:0])          &
    i0_rs2_class_d.alu                      ;
```

**任务：**将前面的公式与DDCARV中流水线处理器部分所述的公式进行比较。

**任务：**分析Verilog代码，说明前一个公式如何执行计算。必须检查模块dec\_decode\_ctl的以下行。

```
2384    assign i0_rs2bypass[9:0] = { i0_rs2_depth_d[3:0] == 4'd1 & i0_rs2_class_d.alu,
2385                                i0_rs2_depth_d[3:0] == 4'd2 & i0_rs2_class_d.alu,
2386                                i0_rs2_depth_d[3:0] == 4'd3 & i0_rs2_class_d.alu,
```

```
1733    assign {i0_rs2_class_d, i0_rs2_depth_d[3:0]} =
1734                                                    (i0_rs2_depend_i1_e1) ? { i1_elc, 4'd1 } :
1735                                                    (i0_rs2_depend_i0_e1) ? { i0_elc, 4'd2 } :
1736                                                    (i0_rs2_depend_i1_e2) ? { i1_elc, 4'd3 } :
```

```
1509    assign i0_rs2_depend_i0_e1 = dec_i0_rs2_en_d & e1d.i0v & (e1d.i0rd[4:0] == i0r.rs2[4:0]);
```

```
1131    assign dec_i0_rs2_en_d = i0_dp.rs2 & (i0r.rs2[4:0] != 5'd0);
```

**任务：**写出i0\_rs2bypass[9:0]、i0\_rs1bypass[9:0]、i1\_rs2bypass[9:0]和i1\_rs1bypass[9:0]的其他控制位的公式（与上述公式类似）。

### 3:1多路开关:

**输出:** 3:1多路开关的输出为`i0_rs2_d[31:0]`。此信号发送到通路0中ALU的第二个输入(b)。

**输入:** 3:1多路开关的输入为:

- 从寄存器文件读取的值 (`gpr_i0_rs2_d`)。
- 从指令中获取的立即数 (`dec_i0_immed_d`)。
- 从上述10:1多路开关获取的后面阶段的旁路值 (`i0_rs2_bypass_data_d`)。

**控制信号:** 3:1多路开关的控制信号 (`dec_i0_rs2_bypass_en_d`) 选择:

- 后面阶段的旁路值 (`i0_rs2_bypass_data_d`) (`dec_i0_rs2_bypass_en_d == 1`时)
- 或者来自寄存器文件或立即数的值 (分别为`gpr_i0_rs2_d`和`dec_i0_immed_d`) (`dec_i0_rs2_bypass_en_d == 0`时)。同一个信号选择两个输入可能看起来很奇怪;但是,不得选择的信号 (`gpr_i0_rs2_d`或`dec_i0_immed_d`) 在Verilog代码中被强制为0。

3:1多路开关的选择信号 (`dec_i0_rs2_bypass_en_d`) 简单计算为10:1多路开关的10位控制信号的逻辑或运算:

```
assign dec_i0_rs2_bypass_en_d = |i0_rs2bypass[9:0];
```

因此,只要指令的第二个输入操作数与仍在执行的先前指令的结果相关(即信号`i0_rs2bypass[9:0]`的10位中的任何一位为1),则`dec_i0_rs2_bypass_en_d == 1`且操作数通过转发获得。相反,如果它不与任何先前指令相关,则`dec_i0_rs2_bypass_en_d == 0`且操作数来自寄存器文件或立即数。

## ii. 实验

图8给出了图2程序中循环的任意一次迭代的仿真结果。图3中的周期*i*显示在图的顶部。

顶部的信号(跟踪信号)可帮助跟踪通过流水线的指令。请注意,这些信号已在之前的实验中使用过。通路0中各个信号的含义如下(同样适用于通路1,只需将信号名称中的*i0*替换为*i1*):

- `Dec_i0_instr_d` → 译码阶段的指令
- `i0_inst_e1` → EX1阶段的指令
- `i0_inst_e2` → EX2阶段的指令

- `i0_inst_e3` → EX3阶段的指令
- `i0_inst_e4` → 提交阶段的指令
- `i0_inst_wb` → 回写阶段的指令

跟踪信号下方为上面分析的每个多路开关的主要信号。每个多路开关被两条蓝线包围，而每个多路开关的控制信号、输入和输出由红线分隔。

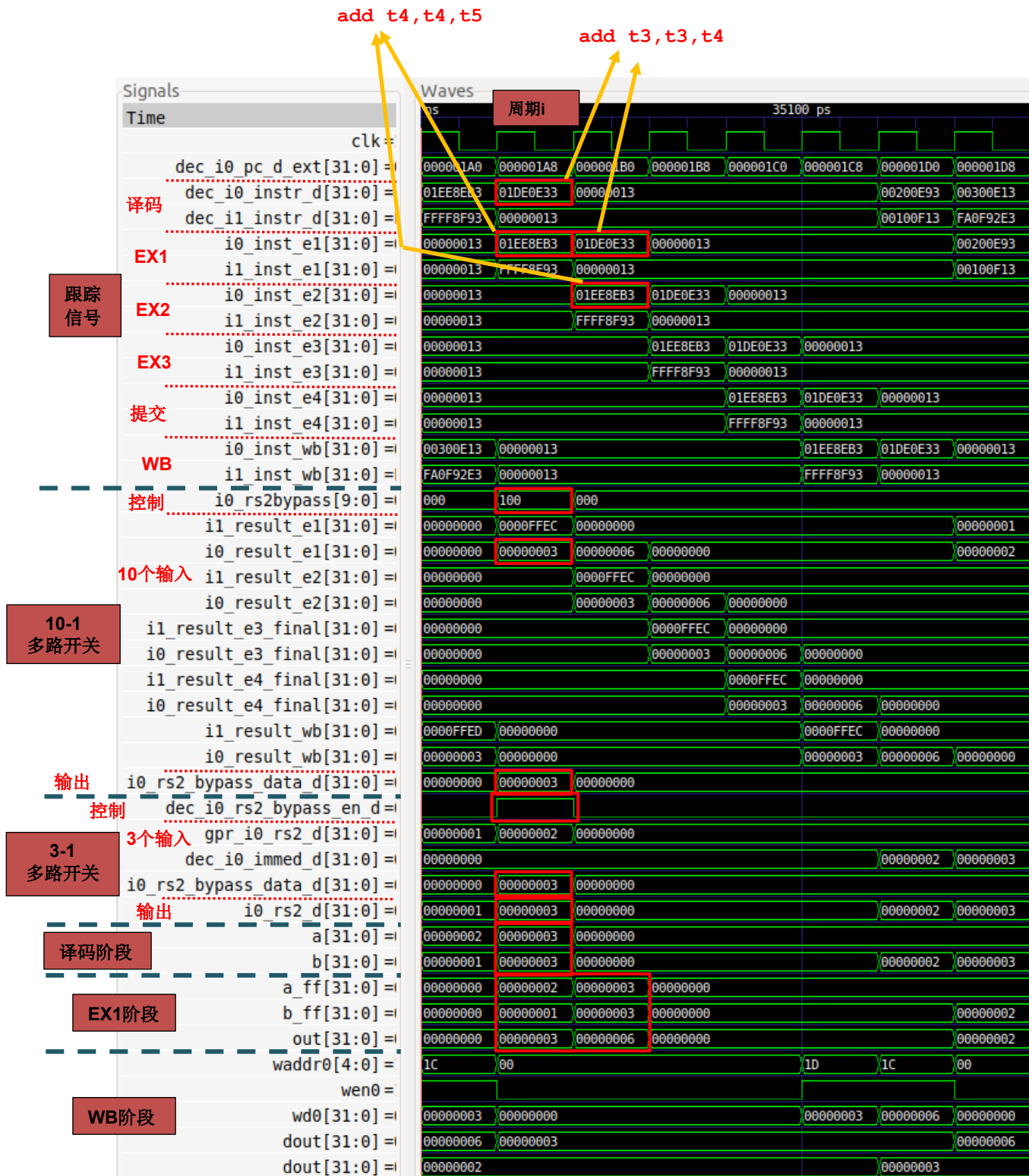


图8. 图2程序中循环的任意一次迭代的仿真结果

图9所示为图2程序在周期*i*（如图8所定义）中执行期间的译码和EX1阶段。

## Cycle i

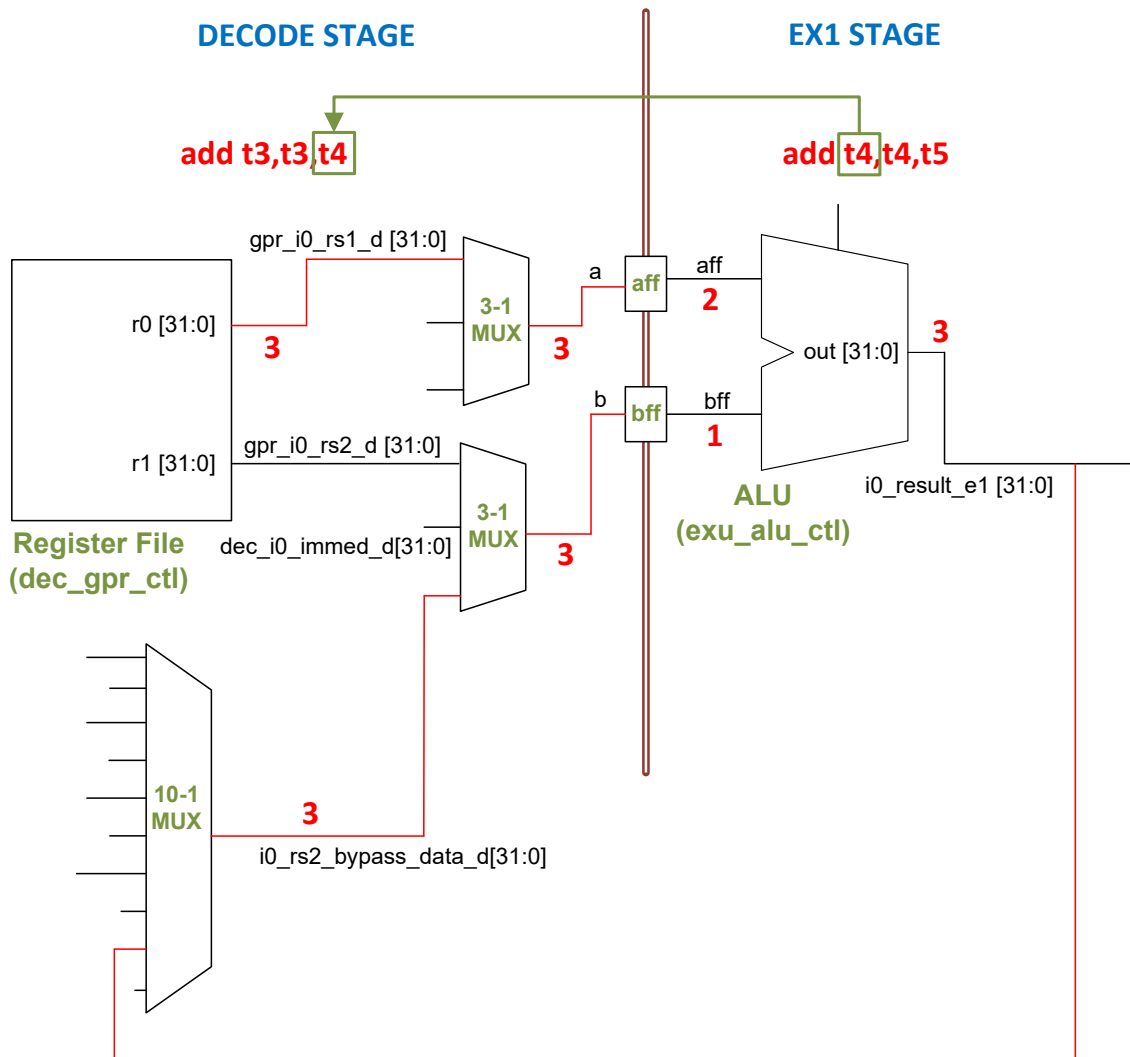


图9. 图2示例程序在周期*i*（如图8所定义）中执行期间的译码和EX1阶段

**任务:** 在自己的计算机上重复图8中的仿真过程。可以使用以下位置提供的.tcl文件：  
[RVfpgaPath]/RVfpga/Labs/Lab15/DataHazards\_AL-AL/test\_Advanced.tcl。

同时分析图8中的仿真以及图9中的图。

### - 图8所示的跟踪信号:

- 在周期*i*中，第二条add指令在通路0的译码阶段执行（dec\_i0\_instr\_d = 0x01DE0E33），第一条add指令在I0管道的EX1阶段执行（i0\_inst\_e1 = 0x01EE8EB3）。
- 在周期*i+1*中，第二条add指令进入I0管道的EX1阶段（i0\_inst\_e1 = 0x01DE0E33），第一条add指令进入I0管道的EX2阶段（i0\_inst\_e2 = 0x01EE8EB3）。



- **10:1多路开关:** 在周期*i*中, 信号*i0\_rs2bypass*[9:0] = 0x100 (即 *i0\_rs2bypass*[8] = 1), 因此输出与来自I0管道EX1阶段的值相连 (参见图9):  

$$i0\_rs2\_bypass\_data\_d = i0\_result\_e1 = 0x00000003$$
- **3:1多路开关:** 在周期*i*中, 信号*dec\_i0\_rs2\_bypass\_en\_d* = 1, 因此输出与来自旁路逻辑的值相连 (参见图9):  

$$i0\_rs2\_d = i0\_rs2\_bypass\_data\_d = 0x00000003$$
- 图8所示的EX1阶段:
  - o 在周期*i*中, 第一条add指令计算I0管道ALU中的加法:  

$$a\_ff(2) + b\_ff(1) = out(3)$$
  - o 在周期*i+1*中, 第二条add指令计算I0管道ALU中的加法:  

$$a\_ff(3) + b\_ff(3) = out(6)$$

**任务:** 对于图2中的程序, 针对两条相互依赖的指令彼此之间距离不同的情况执行与图8中相同的分析。可以通过更改两条相关add指令之间的nop数量来控制距离。

此外, 需创建第一个输入操作数接收转发数据的其他示例。

还可以创建两条add指令通过I1管道执行的其他示例, 但需确认行为相同。

最后, 将相关add指令 (add t3, t3, t4) 替换为通过其他管道执行的其他相关指令并分析仿真结果。例如, 可以包含以下指令之一来代替第二条add指令:

```
- lw t3, (t4) (强制读取值来自DCCM, 如实验13所述)
- mul t3, t3, t4
- div t3, t3, t4
```

### 3. 在提交阶段通过转发解除数据冒险

当一条指令与需要几个周期才能获得结果的前一条指令 (即多周期操作, 例如lw、mul和div指令等) 相关时, 会出现一种更微妙的情况。在本部分中, 我们将分析在执行lw指令和相关add指令时可能发生的特定情况, 我们将其他指令和情况的分析留作练习。

如实验13所述, 当使用低延时DCCM存储器时, lw指令需要三个周期 (DC1、DC2和DC3阶段) 才能获得其结果。这是本部分中使用的场景。(正如我们在实验13和14中分析的那样, 使用外部DDR2存储器时会产生更大的延时 – 这种更大的存储器延时对数据冒险的影响留作练习。)

如果lw指令在相关add指令之前三个或三个以上周期执行，则按照第2部分所述解除冒险。在这种情况下，相应部分所述的相同10:1和3:1多路开关用于将装载指令读取的数据转发到依赖于它的后续指令。

**附录A:** 文档末尾的附录包含按照第2部分所述处理的lw-add RAW数据冒险的示例。

但是，如果装载指令执行的时间与相关add指令执行的时间间隔较近，则按照与第2部分所述不同的方式解除冒险。现在的问题是，当add指令到达EX1阶段时，lw指令读取的值尚不可用。

在DDCARV所述的流水线处理器中，这种情况下会引入气泡（bubble），这样相关指令便可等待至读取值可用时才加以使用。这几乎不需要添加硬件，但会影响性能。因此，SweRV EH1允许相关指令继续通过流水线，然后在提交阶段重新计算运算（由于数据相关性而需要时）。

具体来说，SweRV EH1在每路的提交阶段添加一个额外的ALU（辅助ALU）。必要时，此ALU使用适当的输入重新计算算术逻辑运算。因此，不会因暂停而丢失周期 – 但代价是添加了两个额外的ALU（每路一个）以及控制信号和逻辑。图10给出了此辅助ALU在通路0提交阶段的实现（ALU用蓝框包围）以及在EX3阶段为第二个输入操作数添加的转发逻辑（此逻辑由红框包围）。（在实验11的图4中，为简单起见，没有包含这两个额外的ALU和转发路径。）

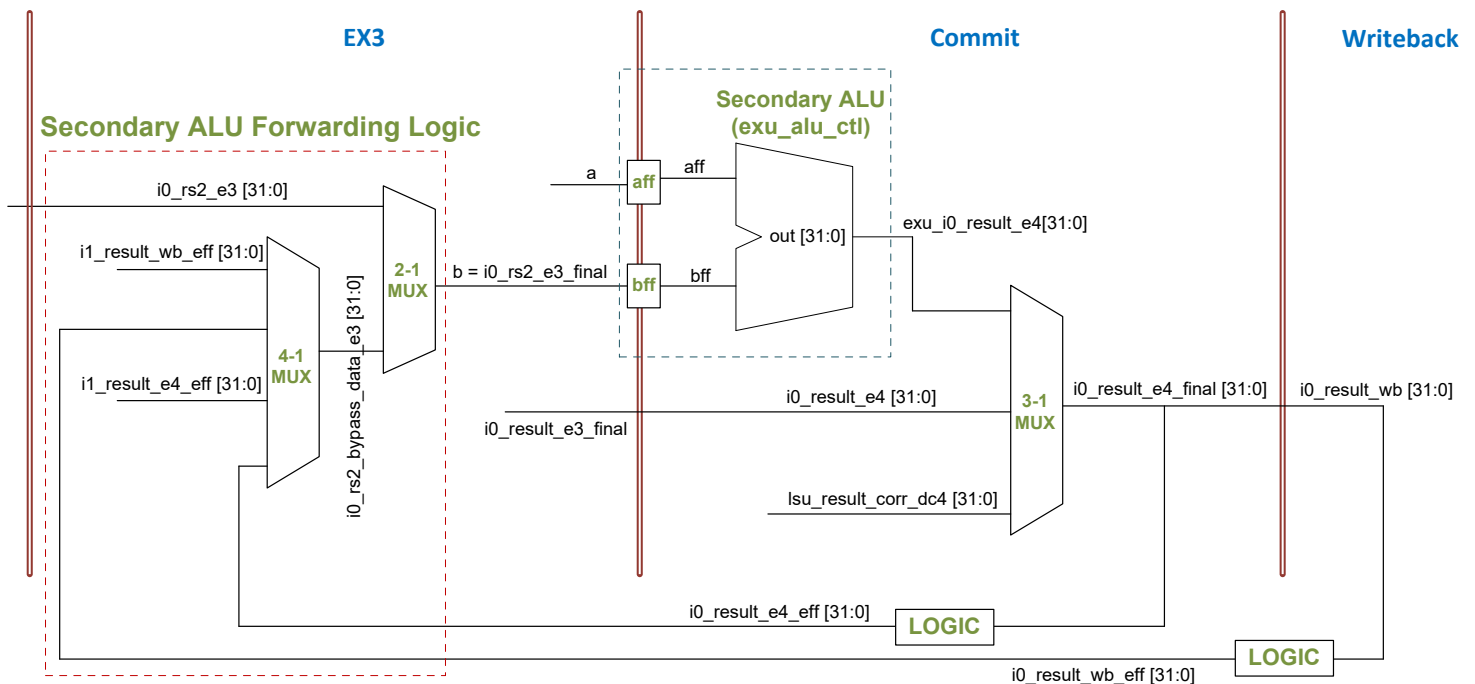


图10. 通路0提交阶段中的辅助ALU

**任务：** 向图10添加相应逻辑以生成IO管道中的辅助ALU的第一个输入操作数（a）。

图11给出了本部分中使用的示例代码。它在执行一条lw指令后紧接着执行一条独立的add指令（add t6, t6, -1：计算循环索引），然后执行一条与装载相关的add指令。独立add指令用于强制lw指令和相关add指令通过通路0执行。因此，与附录中程序的唯一区别在于lw和add指令现在更接近；不过，正如我们刚刚所说，程序中的这种微小差异会导致执行方式存在巨大差异，接下来将具体阐述。

```
.globl Test_Assembly

.section .midccm
#.data
A: .space 4

.text

Test_Assembly:

la t0, A                                # t0 = addr(A)
li t1, 0x1                              # t1 = 1
sw t1, (t0)                             # A[0] = 1
li t1, 0x0
li t3, 0x1
li t6, 0xFFFF

REPEAT:
    beq t6, zero, OUT                  # Stay in the loop?
    INSERT_NOPS_9
    lw t1, (t0)
    add t6, t6, -1
    add t3, t3, t1                    # t3 = t3 + t1
    INSERT_NOPS_8
    li t1, 0x0
    li t3, 0x1
    add t4, t4, 0x1
    add t5, t5, 0x1
    j REPEAT
OUT:

.end
```

图11. 执行lw指令、独立add指令和相关add指令的程序

通常，文件夹[RVfpgaPath]/RVfpga/Labs/Lab15/DataHazards\_Close-LW-AL提供PlatformIO项目，以便可以根据需要分析、仿真和修改程序。在PlatformIO中打开、编译项目，然后打开反汇编文件（位于[RVfpgaPath]/RVfpga/Labs/Lab15/DataHazards\_Close-LW-AL/pio/build/swervolf\_nexys/firmware.dis）。请注意，lw和add指令位于地址0x000001bc和0x000001c4处。

0x000001bc:	0002a303	lw t1, 0(t0)
0x000001c0:	ffff8f93	addi t6, t6, -1
0x000001c4:	006e0e33	add t3, t3, t1

图12给出了图11程序中循环的任意一次迭代的仿真结果。同样，由于指令高速缓存未命中，除了第一次迭代之外的任何迭代均有效，应尽量避免这种情况。与前一部分的示例一样，顶部的信号（跟踪信号）可帮助跟踪通过流水线的指令。跟踪信号下方为4:1和2:1多路开关的主要信号以及图11中的新ALU。

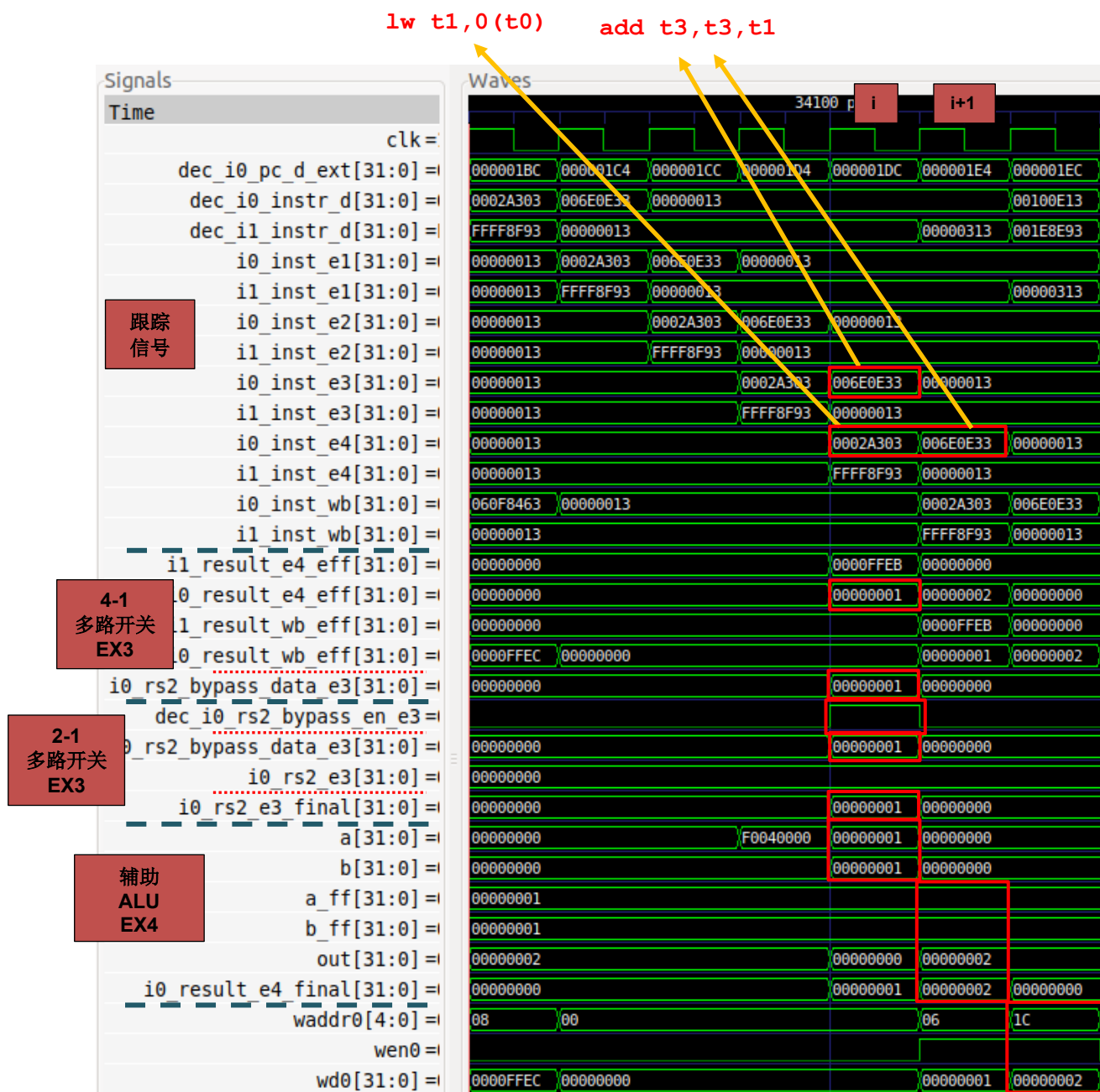


图12. 图11程序中循环的第三次迭代的仿真结果

图13给出了图11程序中循环的第七次迭代在图12所示的周期*i*（add指令处于EX3阶段，lw指令处于提交阶段）和周期*i+1*（add指令处于提交（即EX4）阶段并重新计算适当输入的运算）中的执行图。

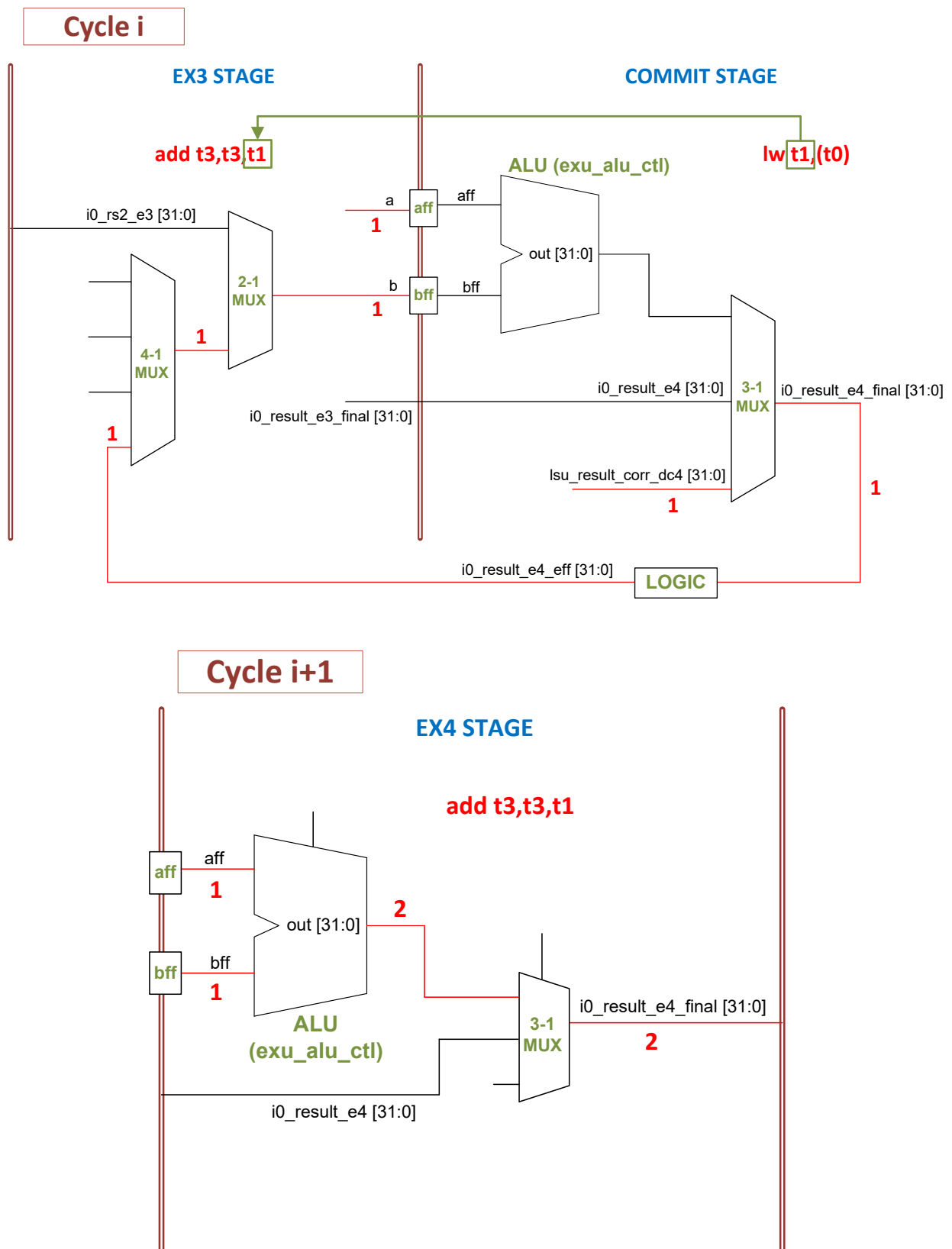


图13. 图11程序中循环的第七次迭代在图12的周期*i*和*i+1*中的执行图

**任务：** 在自己的计算机上重复图12中的仿真过程。可以使用以下位置提供的.tcl文件：  
[RVfpgaPath]/RVfpga/Labs/Lab15/DataHazards\_Close-LW-AL/scriptLoad.tcl

**任务：** 为图11的示例绘制与图3类似的图。

同时分析图12中的波形以及图13中的图。

- **图12所示的跟踪信号：**
  - o 在周期*i*中，add指令处于通路0的EX3阶段 ( $i0\_inst\_e3 = 0x006E0E33$ )，lw指令处于I0管道的提交阶段 ( $i0\_inst\_e4 = 0x0002A303$ )。
  - o 在周期*i+1*中，add指令处于通路0的提交阶段 ( $i0\_inst\_e4 = 0x006E0E33$ )。
- **4-1多路开关：** 在周期*i*中，选择装载指令（在本周期中处于提交阶段）读取的值：  
 $i0\_rs2\_bypass\_data\_e3 = i0\_result\_e4\_eff = 0x00000001$
- **2-1多路开关：** 在周期*i*中，由于装载和加法之间的相关性，选择旁路值  
( $dec\_i0\_rs2\_bypass\_en\_e3 = 1$ )。因此：  
 $i0\_rs2\_e3\_final = i0\_rs2\_bypass\_data\_e3 = 0x00000001$
- **提交阶段ALU：** 在周期*i+1*中，使用正确的值重新计算加法：  
 $out = a\_ff + b\_ff = 0x00000001 + 0x00000001 = 0x00000002$   
然后，在3:1多路开关中，选择ALU输出 ( $exu\_i0\_result\_e4$ )。请注意，如果不存在相关性，则选择信号*i0\_result\_e4*中的值。

**任务：** 在前一个示例中，分析如何获取add t3, t3, t1指令的第一个操作数 (t3)。可以使用以下位置提供的.tcl文件：  
[RVfpgaPath]/RVfpga/Labs/Lab15/DataHazards\_Close-LW-AL/scriptLoad\_FirstOperand.tcl

**任务：** 删除图11的示例中的nop指令并使用硬件计数器获取IPC。

**任务：** 禁止实验11所述的辅助ALU，并通过Verilator仿真和板上执行的方式分析图11中的示例。

**任务：** 在图11的示例中，将add t6, t6, -1指令移至add t3, t3, t1指令之后，然后重新检查仿真中以及板上的程序。

## 4. 练习

- 1) 通过添加与add指令结果相关的额外算术-逻辑指令，修改第3部分中使用的程序。例如，可以将图11中的循环替换为以下代码，其中包含一条新的AND指令（**and t3, t4, t3**），并通过前移指令**add t5, t5, 0x1**对代码顺序稍作调整：

```

REPEAT:
    beq t6, zero, OUT
    INSERT_NOPS_9
    lw t1, (t0)
    add t6, t6, -1
    add t3, t3, t1
    add t5, t5, 0x1
    and t3, t4, t3
    INSERT_NOPS_8
    li t1, 0x0
    li t3, 0x1
    add t4, t4, 0x1
    j REPEAT
OUT:

```

分析Verilator仿真并说明如何为新的A-L指令处理数据冒险。然后删除所有nop指令并分析硬件计数器提供的结果。

- 2) 分析与第3部分所述相同的情况：mul指令后跟使用乘法结果的add指令。在图11的程序中，只需将lw替换为写入寄存器t1的mul。

- 3) 分析lw指令后跟与装载读取的值相关的mul指令这种情况。在图11的程序中，只需将相关add指令替换为mul指令。

- 4) （以下练习基于[PaHe]的练习4.18、4.19、4.20和4.26。）

假设在不处理数据冒险的SweRV EH1处理器版本上执行了以下代码（即，编程器负责通过在必要时插入nop来处理数据冒险）。向代码中添加nop以使其正确运行。

```

    addi x11, x12, 5
    add x13, x11, x12
    addi x14, x11, 15
    add x15, x13, x12

```



然后组成包含至少三个汇编代码片段的序列，这些片段显示不同类型的RAW数据冒险。RAW数据相关性的类型由产生结果的阶段和使用结果的下一条指令标识。

对于每个序列，要使代码在没有转发或冒险检测的SweRV EH1处理器上正确运行，需要在何处插入多少条nop？如果我们使用SweRV EH1中提供的转发而不插入nop，那么CPI是多少？

5) 在实验14第2.C部分的程序（位于 `[RVfpgaPath]/RVfpga/Labs/Lab14/LW_Instruction_ExtMemory`）中，将指令 `add x1, x1, 1` 替换为 `add x28, x1, 1`。这将在修改后的add指令和循环开头的非阻塞装载（`lw x28, (x29)`）之间引入WAW冒险。利用仿真分析SweRV EH1中如何处理此冒险，可以在寄存器文件中查看信号wen2的值。尝试了解控制单元（模块dec）中如何计算此信号。

6) 在实验14第2.C部分的程序（位于 `[RVfpgaPath]/RVfpga/Labs/Lab14/LW_Instruction_ExtMemory`）中，将指令 `add x1, x1, 1` 替换为 `add x1, x28, 1`。这将在修改后的add指令和循环开头的非阻塞装载（`lw x28, (x29)`）之间引入RAW冒险。通过仿真分析SweRV EH1中如何处理此冒险。

7) 在实验14第2.C部分的程序（位于 `[RVfpgaPath]/RVfpga/Labs/Lab14/LW_Instruction_ExtMemory`）中，将指令 `add x1, x1, 1` 替换为 `add x1, x28, 1`，将指令 `add x7, x7, 1` 替换为 `add x28, x7, 1`。这将导致发生RAW和WAW冒险。利用仿真分析SweRV EH1中如何处理这两种冒险。

## 8) 存储-装载转发

这是一种非常值得关注的情况，我们没有在本实验中分析，但您可在本练习中进行分析。当存储以及随后的装载访问相同地址时，可以在内核内将数据从存储转发到装载，无需读取DDR外部存储器，从而节省时间和功耗。

实现这种转发的逻辑包含在LSU中，具体来说是在模块 `lsu_bus_intf` 和 `lsu_bus_buffer` 中，必须在本练习中进行检查。

`[RVfpgaPath]/RVfpga/Labs/Lab15/Sw-Lw-Forwarding` 中的PlatformIO项目用于说明存储-装载转发。此文件夹中提供了 `.tcl` 脚本，您可以使用它来分析循环的任意一次迭代以及了解SweRV EH1中如何执行存储-装载转发。



## 附录A

本附录包含按照第2部分所述处理的lw-add RAW数据冒险的示例。图14给出了本附录中使用的示例代码。它在执行1条lw指令后接着执行5条nop指令，然后执行1条与装载相关的add指令。中间的nop指令用于将两条相关指令分开。

```
.globl Test_Assembly

.section .midccm
#.data
A: .space 4

.text
Test_Assembly:

# Register t3 is also called register 28 (x28)
la t0, A                                # t0 = addr(A)
li t1, 0x1                              # t1 = 1
sw t1, (t0)                             # A[0] = 1
li t1, 0x0
li t3, 0x1
li t6, 0xFFFF

REPEAT:
    beq t6, zero, OUT                  # Stay in the loop?
    INSERT_NOPS_8
    lw t1, (t0)
    INSERT_NOPS_5
    add t3, t3, t1                    # t3 = t3 + t1
    INSERT_NOPS_8
    li t1, 0x0
    li t3, 0x1
    add t6, t6, -1
    j REPEAT
OUT:

.end
```

图14. 执行lw、5条nop和1条相关add指令的程序

通常，文件夹RVfpgaPath]/RVfpga/Labs/Lab15/DataHazards\_FarAway-LW-AL提供PlatformIO项目，以便可以根据需要分析、仿真和修改程序。打开、编译项目，然后打开反汇编文件（位于[RVfpgaPath]/RVfpga/Labs/Lab15/DataHazards\_FarAway-LW-AL/.pio/build/swervolf\_nexys/firmware.dis）。请注意，lw和add指令位于地址0x000001b0和0x000001c8处。

0x000001b0:	0002a303	lw	t1, 0(t0)
0x000001b4:	00000013	nop	
0x000001b8:	00000013	nop	
0x000001bc:	00000013	nop	
0x000001c0:	00000013	nop	
0x000001c4:	00000013	nop	
0x000001c8:	006e0e33	add	t3, t3, t1

图15给出了图14程序中循环的第三次迭代的仿真结果。同样，由于指令高速缓存未命中，除了第一次迭代之外的任何迭代均有效，应尽量避免这种情况。与主要实验的示例一样，顶部的信号（跟踪信号）可帮助跟踪通过流水线的指令。跟踪信号下方为每个多路开关的主要信号。每个多路开关的信号被蓝色虚线包围。图中给出了每个多路开关的控制信号、输入和输出（与主要实验中一样）。

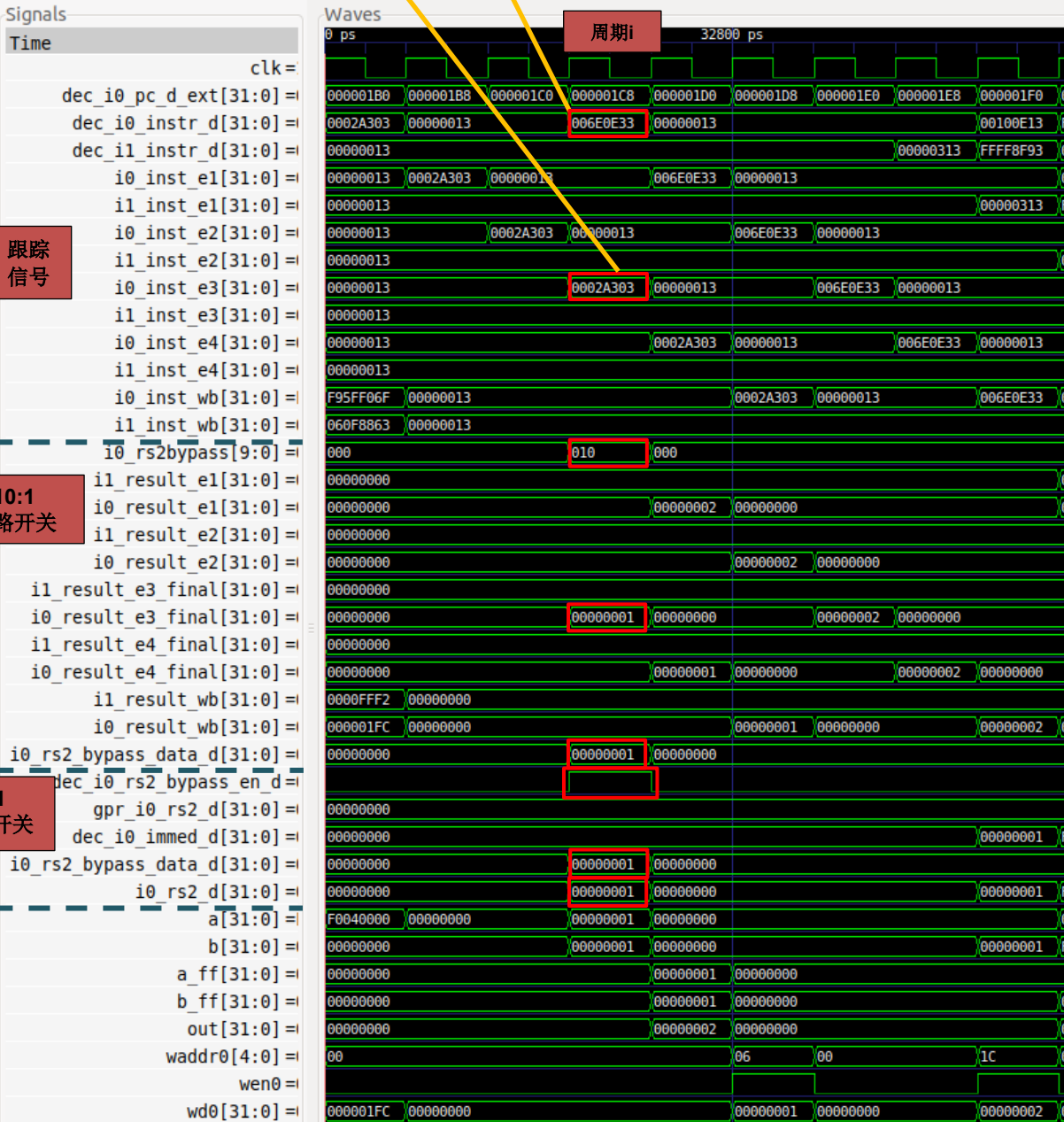


图15. 图14程序中循环的第三次迭代的仿真结果

图16给出了图14程序在图15所定义的周期*i*（add指令处于译码阶段，lw指令处于DC3阶段）中的执行图。

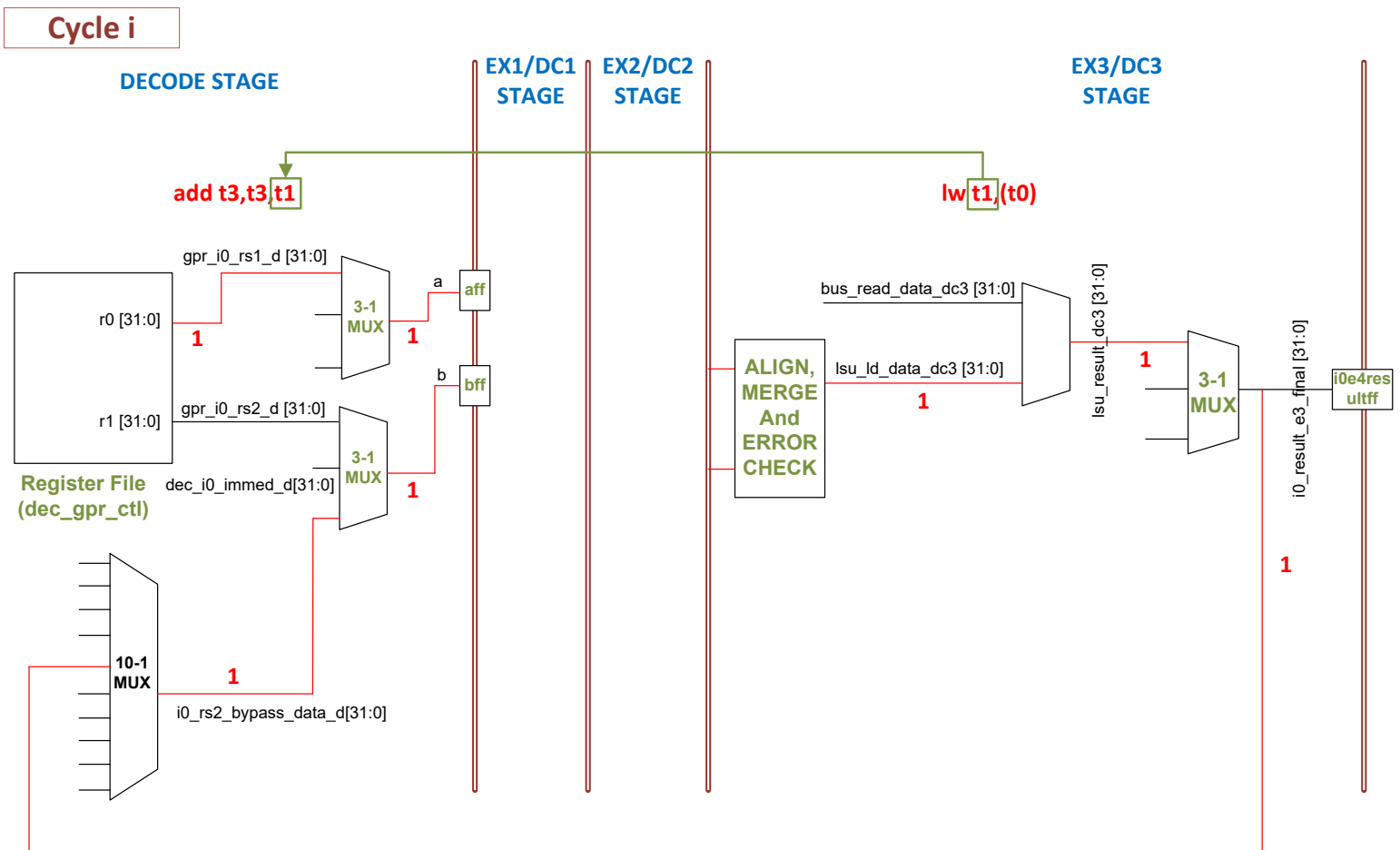


图16. 图14程序中循环的第三次迭代在图15给出的第四个周期中执行期间的硬件

**任务：** 在自己的计算机上重复图15中的仿真过程。

同时分析图15中的波形以及图16中的图。

- 图15所示的跟踪信号：
  - o 在周期*i*中，add指令处于通路0的译码阶段（dec\_i0\_instr\_d = 0x006E0E33），lw指令处于I0管道的DC3阶段（i0\_inst\_e3 = 0x0002A303）。
- **10:1多路开关：** 在周期*i*中，信号i0\_rs2bypass[9:0] = 0x010（即 i0\_rs2bypass[4] = 1），因此输出与来自I0管道EX3/DC3阶段的值相连（参见图16）：
 
$$i0\_rs2\_bypass\_data\_d = i0\_result\_e3\_final = 0x00000001$$
- **3:1多路开关：** 在周期*i*中，信号dec\_i0\_rs2\_bypass\_en\_d = 1，因此输出与来自旁路逻辑的值相连（参见图9）：
 
$$i0\_rs2\_d = i0\_rs2\_bypass\_data\_d = 0x00000001$$

**任务：**比较上述场景在SweRV EH1和DDCARV所述流水线处理器中的处理方式。

**任务：**如果仔细比较图16和实验13的图6，将看到实验13的图6中lw指令读入寄存器文件的值（信号lsu\_ld\_data\_corr\_dc3[31:0]）与图16中lw转发的值（信号lsu\_ld\_data\_dc3[31:0]）不同。两个值的区别在于前者经过模块lsu\_ecc中的ECC逻辑校验，而后者没有。说明为什么lw转发的值未经过错误校验也没有问题。

**任务：**在图14的示例中，删除lw之前和add之后的所有nop指令。不要删除两条相关指令之间的5个nop。分析仿真，然后通过开发板上执行程序，用性能计数器计算IPC（在测量IPC时保留nop指令似乎并不合适，因为它们是无用指令；不过，程序本身就是无用的，这里我们的惟一目的是分析并了解数据冒险）。