



IMAGINATION大學計劃

RVfpga實驗15

資料冒險

1. 簡介

在本實驗中，我們將處理**資料冒險**。正如Hennessy和Patterson在其《電腦體系結構：量化研究方法》[HePa]一書的第6版中所述：當管線變更運算元讀/寫存取順序致使此順序與在非管線處理器上順序執行指令的順序不同時，將發生資料冒險。假設指令*i*在程式中位於指令*j*之前並且兩條指令均使用暫存器*x*。*i*和*j*之間可能發生三種類型的資料冒險：

- **寫後讀 (Read After Write, RAW) 資料冒險**：這是最常見類型的冒險。當指令*j*先讀取暫存器*x*，指令*i*後寫入暫存器*x*時，將發生RAW冒險。因此，指令*j*將使用錯誤的*x*值。
- **讀後寫 (Write After Read, WAR) 資料冒險**：當指令*j*寫入*x*、指令*i*讀取*x*且指令*j*順序調整為在*i*之前發生時，將發生WAR冒險。因此，指令*j*將讀取錯誤的*x*值。這種冒險只在調整指令順序時發生（在SweRV EH1中很少發生）；具體來說，SweRV EH1中從未發生過WAR冒險。
- **寫後寫 (Write After Write, WAW) 資料冒險**：當調整指令順序致使指令*j*先寫入*x*，指令*i*後寫入*x*時，將發生WAW冒險。這種冒險只在調整指令順序時發生（在SweRV EH1中很少發生）；但是，在非阻塞載入的情況下，也可能會發生WAW冒險，我們將在本實驗的後面部分進行分析。

在以下部分中，我們將分析SweRV EH1處理器中如何解除RAW資料冒險，然後介紹與RAW冒險相關的任務和練習，還將透過一個練習分析發生WAW冒險時的情況。

附註：在分析SweRV EH1資料冒險邏輯之前，我們建議閱讀DDCARV中的第7.5節，瞭解如何在管線處理器中解除冒險。資料冒險具體在第7.5.3節中進行分析。盡管本書中展示的管線處理器比SweRV EH1簡單，但兩種處理器中解除資料冒險的方法類似。

2. 在解碼階段透過轉送解除資料冒險

如DDCARV的第7.5.3節所述，可以透過將結果從後期管線階段執行的指令轉送（也稱為旁路）到早期管線階段執行的相關指令來解除一些RAW資料冒險。這需要在功能單元（ALU、乘法器、計算DC1中有效位址的加法器等）前面新增多路開關，以從暫存器檔案或後續階段選擇其運算元。

圖1使用旁路值延伸了實驗11的圖4中所示的解碼階段。轉送邏輯為每個通路中的兩個來源運算元中的每一個產生旁路（即轉送）：

- **通路0：**
 - 第一個輸入運算元：i0_rs1_bypass_data_d[31:0]
 - 第二個輸入運算元：i0_rs2_bypass_data_d[31:0]

- 通路1：

- 第一個輸入運算元：i1_rs1_bypass_data_d[31:0]
- 第二個輸入運算元：i1_rs2_bypass_data_d[31:0]

這四個輸入分配到3:1和4:1多路開關，這兩個多路開關確定每個執行階段管線路徑的輸入運算元。為清楚起見，圖1中的訊號按名稱連接。轉送邏輯的輸入是更後期管線中先前程式指令產生的結果，如下所示。

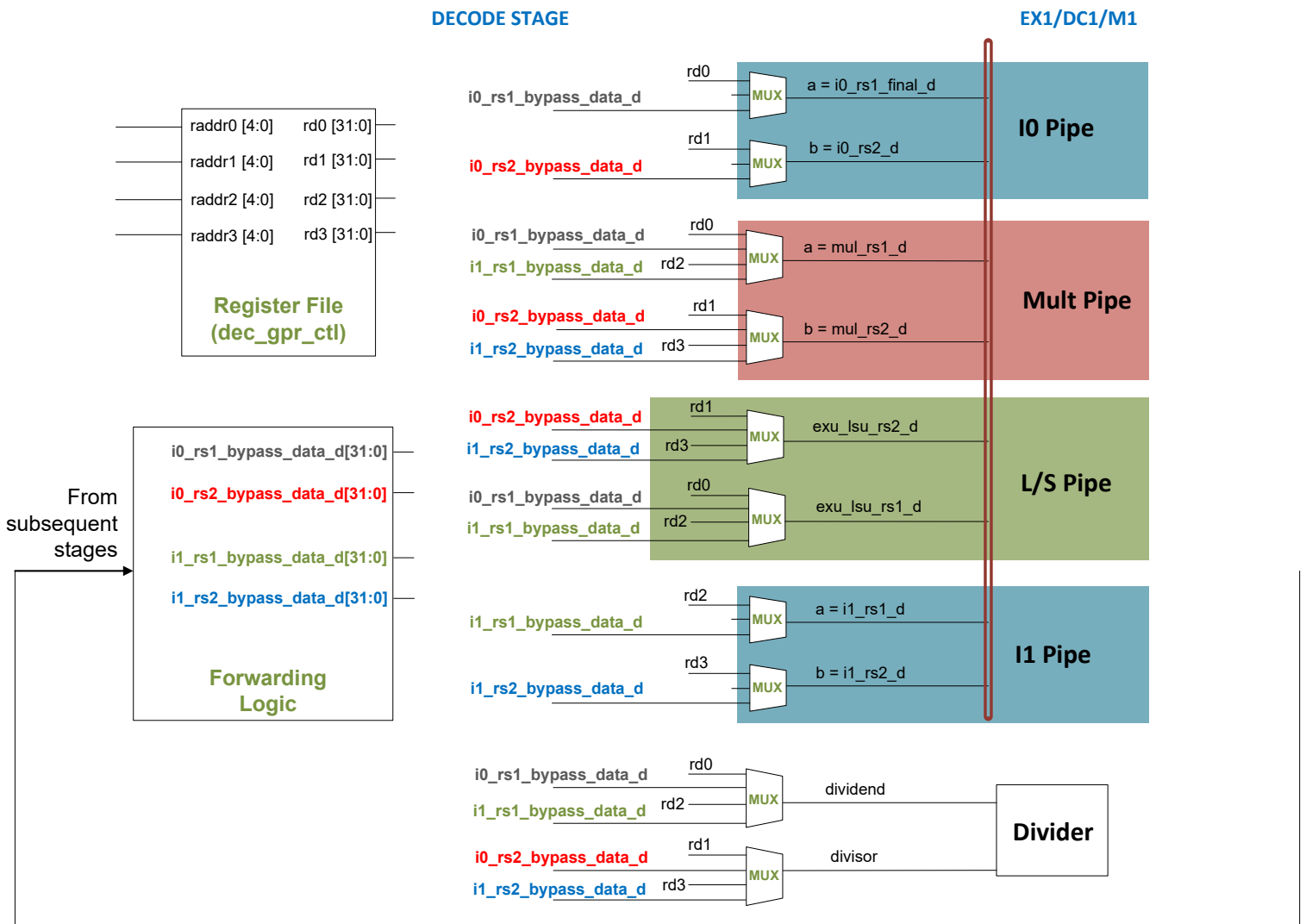


圖1. 功能單元的旁路輸入

SweRV EH1處理器中存在許多轉送路徑。在本部分中，我們將重點關注特定路徑並加以詳細分析。然後在任務和練習中檢查其他情況。我們將分析兩條相關A-L指令同時執行的情況以及如何解除RAW資料冒險。正如在實驗12和13中一樣，我們將先從基礎研究（第2.A部分）開始，然後再進行進階分析（第2.B部分）。可以選擇僅完成基本部分，也可選擇完成兩個部分。

我們將使用圖2顯示的範例，該範例執行重複0xFFFF次迭代的迴圈中包含的兩條add指令。第一條add指令將一個值寫入t4，第二條add指令使用t4作為其第二個輸入運算元。這兩條add指令之間將插入一條獨立add指令（add t6, t6, -1，用於更新迴圈索引的指令）以強制相關add指令使用處理器的相同通路。

```
.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
```

圖2. 兩條add指令之間的RAW資料冒險

資料夾[RVfpgaPath]/RVfpga/Labs/Lab15/DataHazards_AL-AL提供PlatformIO專案，以便可以根據需要分析、模擬和修改程式。在PlatformIO中開啟、編譯專案，然後開啟反組譯檔案（位於[RVfpgaPath]/RVfpga/Labs/Lab15/DataHazards_AL-AL/.pio/build/swervolf_nexys/firmware.dis中），則會發現所分析的兩條add指令位於位址0x000001A0和0x000001A8處：

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

A. A-L指令之間RAW資料冒險的基本分析

在我們分析的範例中，第二條add指令（add t3, t3, t4）需要使用第一條add指令（add t4, t4, t5）的結果作為其第二個輸入運算元。此結果在EX1階段提供，可以從此階段旁路到解碼階段，供第二條add指令使用。在我們的範例（圖2）中，所有迭代都是相等的，t4最初為2，第一個加法後為3。最後這個值（3）是第二個加法必須用作其第二個輸入運算元的值，而不是從暫存器檔案讀取的值（在第一條add指令到達寫回階段並進行更新之前為2）。

圖3所示為圖2範例中迴圈的任意一次迭代期間通過SweRV EH1管線的指令流。在週期i中，在I0管道的EX1階段計算的值必須轉送到處於通路0解碼階段的指令，因為所分析的兩條add指令之間存在RAW資料冒險。

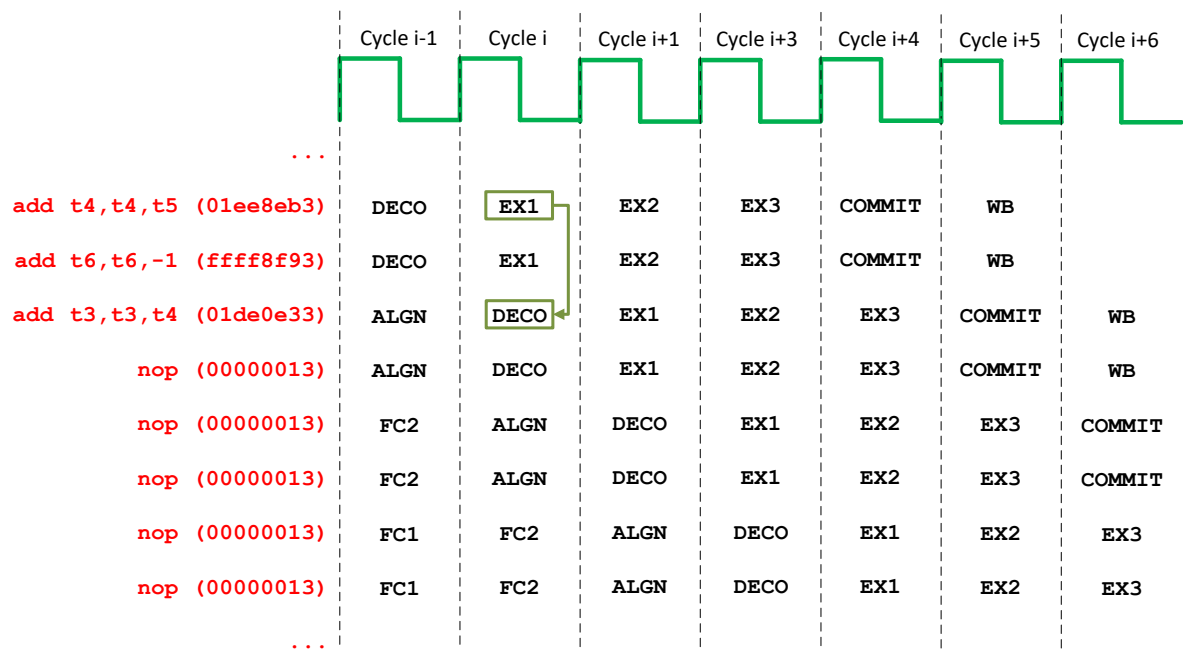


圖3. 圖2範例程式碼的執行。轉送在週期*i*中執行。

圖4 所示為圖3的週期*i*期間SweRV EH1通路0的解碼和EX1階段。在此週期中，第一條add指令（add t4,t4,t5）處於EX1階段，第二條add指令（add t3,t3,t4）處於解碼階段。如圖所示，第一條add指令的結果旁路到解碼階段，它由轉送邏輯（我們將在下一部分詳細分析）選擇，然後用作第二條add指令的第二個輸入運算元。



圖4. 結果從通路0的EX1階段轉送到解碼階段（第二個運算元）

最後，圖5顯示了圖2中的程式在圖3的週期和*i+1*期間的模擬情況。



圖5. 圖2範例程式碼的模擬

任務：在自己的電腦上重複圖5中的模擬過程。可以使用以下位置提供的.tcl檔案：
[RVfpgaPath]/RVfpga/Labs/Lab15/DataHazards_AL-AL/test_Basic.tcl。

同時分析圖5中的模擬以及圖4中的圖。

- 指令 `add t4,t4,t5 (0x01ee8eb3)` :
 - o 在週期*i*中，此指令處於I0管道的EX1階段 (`i0_inst_e1 = 0x01ee8eb3`)。它在ALU中計算以下加法：

$$a_ff(2) + b_ff(1) = out(3)$$
 加法的結果在解碼階段作為輸入提供給轉送邏輯，如圖4所示。
- 指令 `add t3,t3,t4 (0x01de0e33)` :
 - o 在週期*i*中，此指令處於通路0的解碼階段 (`dec_i0_instr_d = 0x01de0e33`)。轉送邏輯將 `i0_result_e1` 與 `i0_rs2_bypass_data_d` 連接。兩個3:1多路開關為下一個週期 (週期*i+1*) 在I0管道的EX1階段計算的加法選擇輸入運算元；具體如下：

$$a = 3 \text{ (來自暫存器檔案)}$$

$$b = 3 \text{ (來自I0管道EX1階段的ALU輸出，經過轉送邏輯的訊號 } i0_rs2_bypass_data_d)$$
 - o 在週期*i+1*中，此指令處於I0管道的EX1階段 (`i0_inst_e1 = 0x01de0e33`)。它在ALU中計算以下加法：

$$a_ff(3) + b_ff(3) = out(6)$$

任務：刪除圖2範例中的所有nop指令。對於迴圈的兩次連續迭代，繪製與圖3類似的圖，然後透過將其與Verilator模擬進行比較來分析並確認此圖是否正確，最後在板上執行程式時使用效能計數器計算IPC。

任務：在圖2的範例中，刪除所有nop指令並將 `add t6,t6,-1` 指令移至 `add t3,t3,t4` 指令之後，然後重新檢查模擬中以及板上的程式。在這個調整順序的程式中，兩條相關add指令 (`add t4,t4,t5` 和 `add t3,t3,t4`) 在同一週期到達解碼階段，這將影響效能。透過模擬和板上執行來說明這些變化的影響。

測試將相關add指令替換為其他相關指令時的類似情況，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指令之間RAW資料冒險的進階分析

i. 理論說明

圖6透過新增一個10:1多路開關（在圖6中用藍框包住）對圖1和圖4中的圖進行了延伸，此多路開關產生訊號*i0_rs2_bypass_data_d*，此訊號在圖2的範例中為第二條add指令（add t3, t3, **t4**）提供第二個輸入運算元。此10:1多路開關在轉送邏輯方塊內實作，如圖1和圖4所示。

這張圖還顯示了此10:1多路開關的輸入連接。旁路的值可來自透過通路0或通路1執行的指令。因此，每路需要五個轉送路徑。具體來說，10:1多路開關的輸入可來自通路0或通路1的任何後續階段（EX1、EX2、EX3、提交和寫回）。為簡單起見，我們用線連接來自通路0的5個輸入，而來自通路1的5個輸入按名稱連接。

轉送邏輯內部的三個附加10:1多路開關計算其他三個來源運算元：訊號

i0_rs1_bypass_data_d、*i1_rs1_bypass_data_d*和*i1_rs2_bypass_data_d*。但是，我們沒有在圖中顯示這三個訊號，因為它們並未在本部分分析的範例中使用（圖2）。全部四個多路開關均可在模組**dec_decode_ctl**的第2429-2473行中找到。

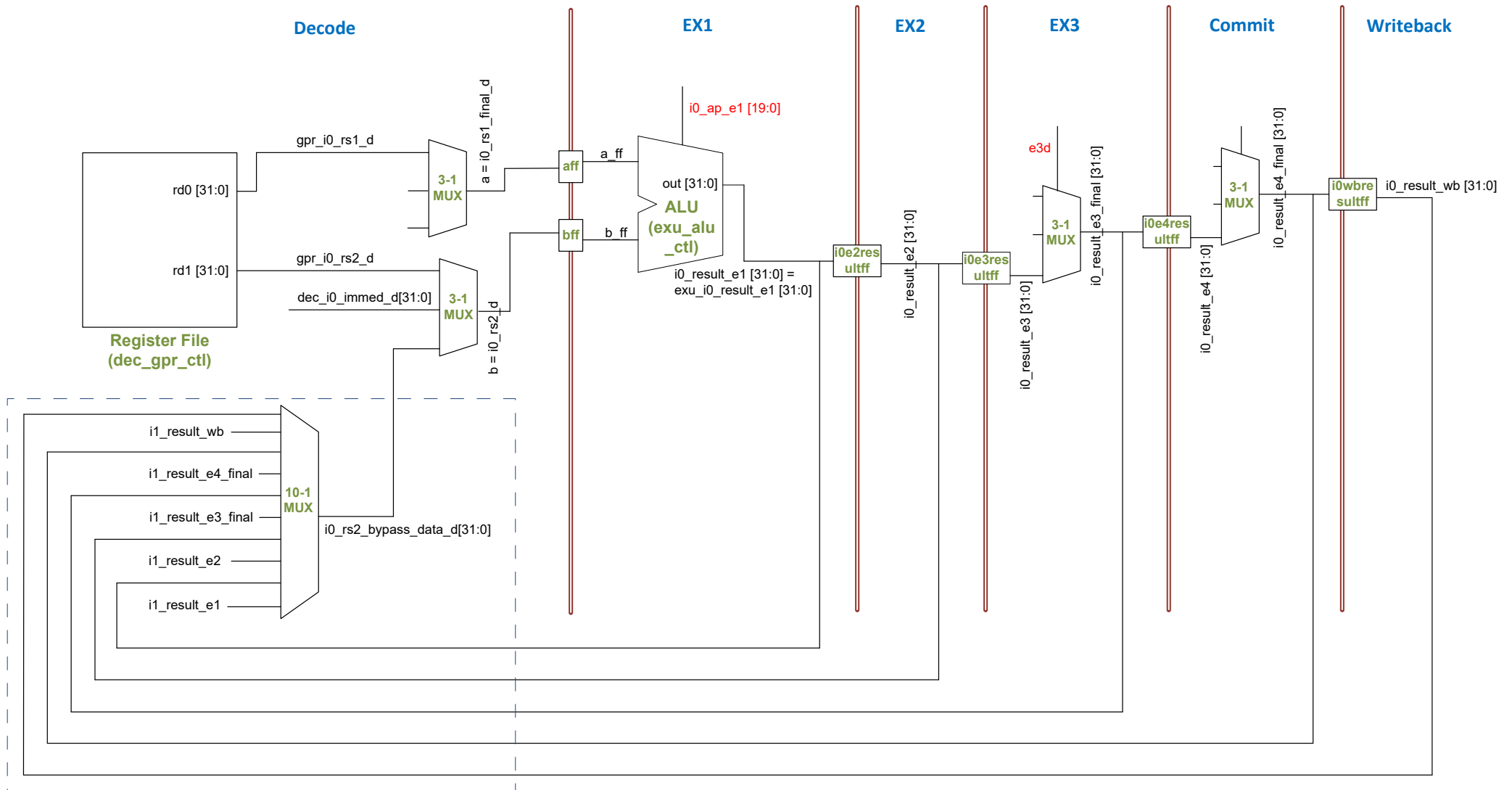


圖6. 包含ALU第二個輸入來源所用轉送邏輯的I0管道

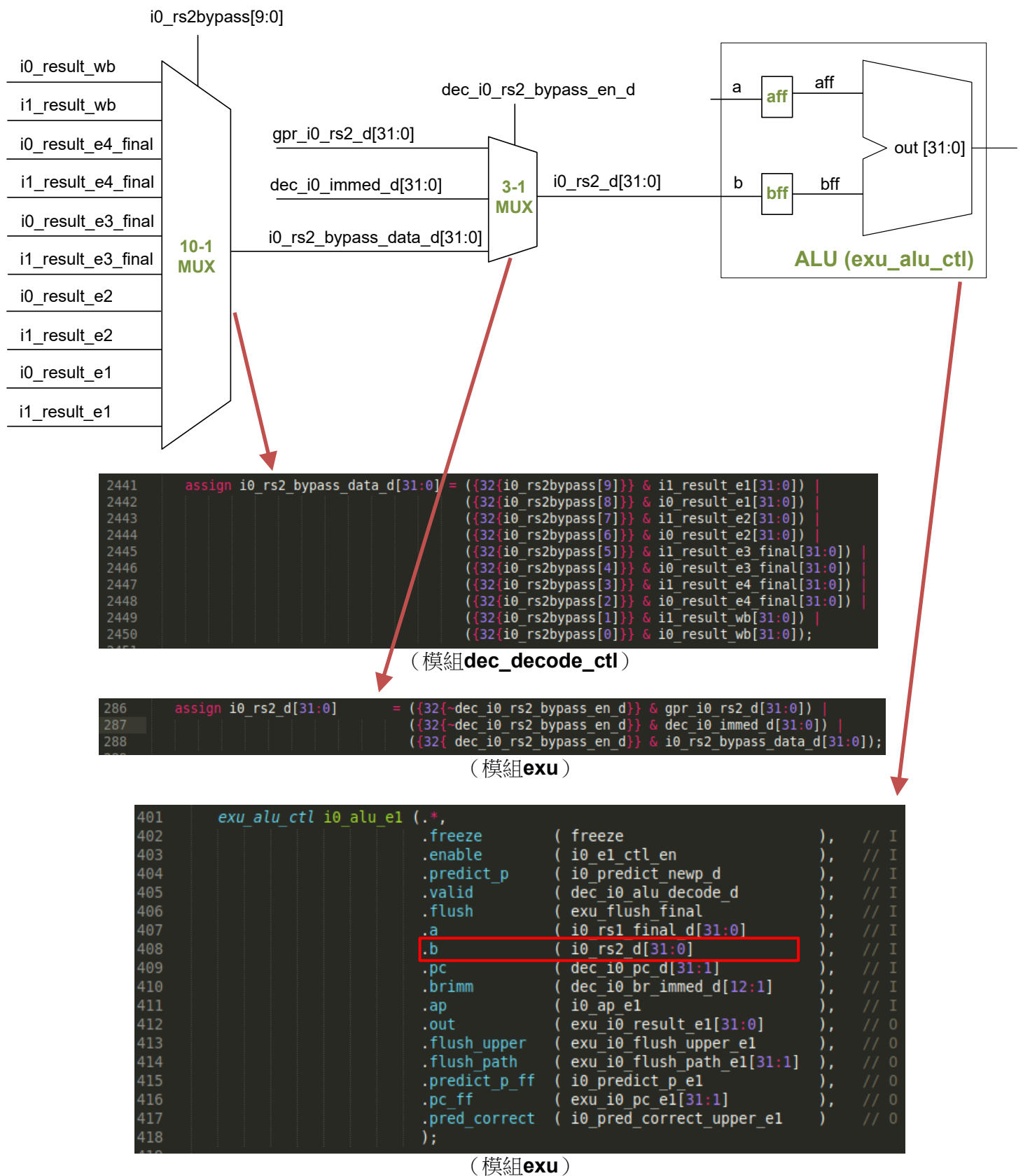


圖7. 圖6中強調顯示的10:1和3:1多路開關

圖7對圖6的兩個多路開關（10:1和3:1多路開關）進行了放大，這兩個多路開關計算I0管道ALU的第二個輸入運算元（b）。圖7顯示了在模組`dec_decode_ctl`和`exu`中實作這兩個多路開關的區塊圖和Verilog程式碼。

附註：DDCARV所述的處理器中也包含圖7中的兩個多路開關。資料轉送到此處理器中的執行階段，並且轉送路徑較少，因為此處理器不是超標量處理器並且管線較短。可以分析DDCARV的圖7.55中的轉送路徑。

接下來分析圖7所示的兩個多路開關的輸入、輸出和控制訊號。

10:1多路開關：

輸出：10:1多路開關的輸出為`i0_rs2_bypass_data_d[31:0]`。此訊號包含必須轉送（旁路）到解碼階段指令的值。

輸入：10:1多路開關的輸入是程式中在後面階段（EX1、EX2、EX3、提交或寫回）執行的先前指令的結果。其中五個訊號來自I0管道（如圖6所示），其他五個訊號來自I1管道（圖6中未顯示），因為解碼階段的指令可能與兩個通路中任一通路中執行的指令相關。

控制訊號：控制訊號（`i0_rs2bypass[9:0]`）選擇與多路開關的輸出連接的輸入。它由10位元組成，同一時間最多有一位元為高電平（如果沒有資料冒險，它們均可以為0）。多路開關的工作方式如下：

- `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`

為瞭解此10位元控制訊號如何計算，我們將說明訊號`i0_rs2bypass[8]`的計算過程，它是圖2的範例中由於add-add旁路而變為高電平的訊號。

- 如果`i0_rs2bypass[8]`為1，則選擇的旁路值為`i0_result_e1`，即I0管道EX1階段執行的指令的結果（參見圖6）。
- 要使EX1將資料轉送到解碼階段（均在I0管道中），必須滿足以下條件（有關控制訊號的資訊，請參見SweRVref文件的第4部分）：

- 解碼階段指令的第二個輸入運算元從暫存器檔案中讀取，而不是從暫存器0中讀取。在SweRV EH1控制單元中，當dec_i0_rs2_en_d為1時，將出現這種情況。相應的Verilog程式碼如下：

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

- I0管道EX1階段的指令有效：
e1d.i0v == 1
- I0管道EX1階段指令的目標暫存器和通路0解碼階段指令的第二個來源暫存器相同：
e1d.i0rd[4:0] == i0r.rs2[4:0]

- I0管道EX1階段的指令為ALU操作：
i0_rs2_class_d.alu == 1

- 考慮到上述所有情況，可以得出以下結論：

```
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                      ;
```

任務：將前面的公式與DDCARV中管線處理器部分所述的公式進行比較。

任務：分析Verilog程式碼，說明前一個公式如何執行計算。必須檢查模組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_e1c, 4'd1 } :
1735                                                    (i0_rs2_depend_i0_e1) ? { i0_e1c, 4'd2 } :
1736                                                    (i0_rs2_depend_i1_e2) ? { i1_e2c, 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);
```

任務：寫出i0_rs2bypass[9:0]、i0_rs1bypass[9:0]、i1_rs2bypass[9:0]和i1_rs1bypass[9:0]的其他控制位元的公式（與上述公式類似）。

3:1多路開關：

輸出：3:1多路開關的輸出為`i0_rs2_d[31:0]`。此訊號傳送到通路0中ALU的第二個輸入(b)。

輸入：3:1多路開關的輸入為：

- 從暫存器檔案讀取的值 (`gpr_i0_rs2_d`)。
- 從指令中取得的立即數 (`dec_i0_immed_d`)。
- 從上述10:1多路開關取得的後面階段的旁路值 (`i0_rs2_bypass_data_d`)。

控制訊號：3:1多路開關的控制訊號 (`dec_i0_rs2_bypass_en_d`) 選擇：

- 後面階段的旁路值 (`i0_rs2_bypass_data_d`) (`dec_i0_rs2_bypass_en_d == 1`時)
- 或者來自暫存器檔案或立即數的值 (分別為`gpr_i0_rs2_d`和`dec_i0_immed_d`) (`dec_i0_rs2_bypass_en_d == 0`時)。同一個訊號選擇兩個輸入可能看起來很奇怪；但是，不得選擇的訊號 (`gpr_i0_rs2_d`或`dec_i0_immed_d`) 在Verilog程式碼中被強制為0。

3:1多路開關的選擇訊號 (`dec_i0_rs2_bypass_en_d`) 簡單計算為10:1多路開關的10位元控制訊號的邏輯或運算：

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

因此，只要指令的第二個輸入運算元與仍在執行的先前指令的結果相關 (即訊號 `i0_rs2bypass[9:0]` 的10位元中的任何一位元為1)，則`dec_i0_rs2_bypass_en_d == 1`且運算元透過轉送取得。相反的，如果它不與任何先前指令相關，則`dec_i0_rs2_bypass_en_d == 0`且運算元來自暫存器檔案或立即數。

ii. 實驗

圖8顯示了圖2程式中迴圈的任意一次迭代的模擬結果。圖3中的週期顯示在圖的頂部。

頂部的訊號 (追蹤訊號) 可幫助追蹤通過管線的指令。請注意，這些訊號已在之前的實驗中使用過。通路0中各個訊號的含義如下 (同樣適用於通路1，只需將訊號名稱中的i0替換為i1)：

- `Dec_i0_instr_d` → 解碼階段的指令
- `i0_inst_e1` → EX1階段的指令
- `i0_inst_e2` → EX2階段的指令

- i0_inst_e3 → EX3階段的指令
- i0_inst_e4 → 提交階段的指令
- i0_inst_wb → 寫回階段的指令

追蹤訊號下方為上面分析的每個多路開關的主要訊號。每個多路開關被兩條藍線包住，而每個多路開關的控制訊號、輸入和輸出由紅線分隔。



圖9所示為圖2程式在週期*i*（如圖8所定義）中執行期間的解碼和EX1階段。

Cycle i

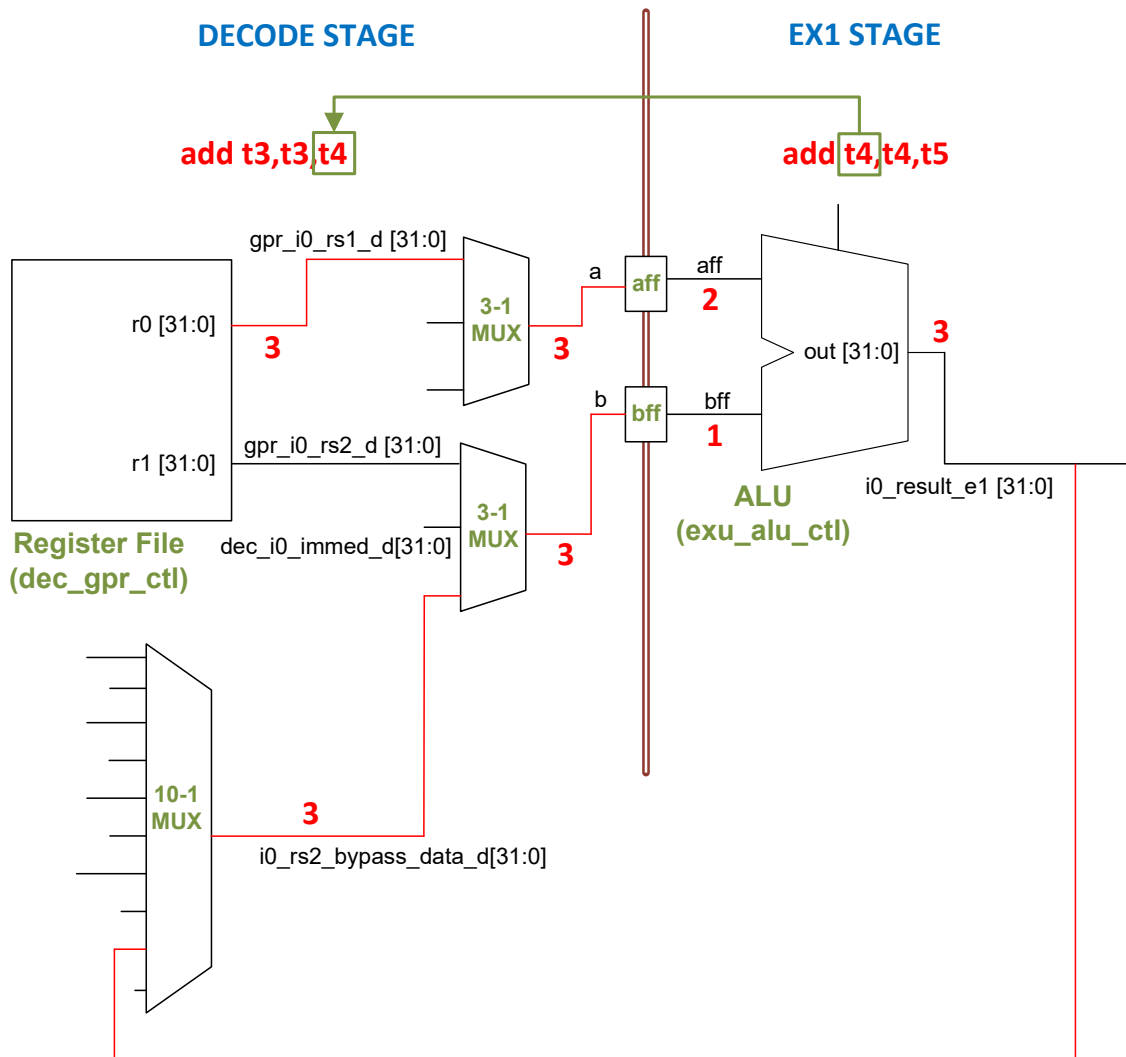


圖9. 圖2範例程式在週期*i*（如圖8所定義）中執行期間的解碼和EX1階段

任務： 在自己的電腦上重複圖8中的模擬過程。可以使用以下位置提供的.tcl檔案：
[RVfpgaPath]/RVfpga/Labs/Lab15/DataHazards_AL-AL/test_Advanced.tcl。

同時分析圖8中的模擬以及圖9中的圖。

- 圖8所示的追蹤訊號：

- 在週期*i*中，第二條add指令在通路0的解碼階段執行（dec_i0_instr_d = 0x01DE0E33），第一條add指令在I0管道的EX1階段執行（i0_inst_e1 = 0x01EE8EB3）。
- 在週期*i+1*中，第二條add指令進入I0管道的EX1階段（i0_inst_e1 = 0x01DE0E33），第一條add指令進入I0管道的EX2階段（i0_inst_e2 = 0x01EE8EB3）。

- **10:1多路開關**：在週期*i*中，訊號*i0_rs2bypass*[9:0] = 0x100（即 *i0_rs2bypass*[8] = 1），因此輸出與來自I0管道EX1階段ex1的值相連（參見圖9）：

$$i0_rs2_bypass_data_d = i0_result_e1 = 0x00000003$$
- **3:1多路開關**：在週期*i*中，訊號*dec_i0_rs2_bypass_en_d* = 1，因此輸出與來自旁路邏輯的值相連（參見圖9）：

$$i0_rs2_d = i0_rs2_bypass_data_d = 0x00000003$$
- 圖8所示的EX1階段：
 - o 在週期*i*中，第一條add指令計算I0管道ALU中的加法：

$$a_ff(2) + b_ff(1) = out(3)$$
 - o 在週期*i+1*中，第二條add指令計算I0管道ALU中的加法：

$$a_ff(3) + b_ff(3) = out(6)$$

任務：對於圖2中的程式，針對兩條相互依賴的指令彼此之間距離不同的情況執行與圖8中相同的分析。可以透過變更兩條相關add指令之間的nop數量來控制距離。

此外，需建立第一個輸入運算元接收轉送資料的其他範例。

還可以建立兩條add指令透過I1管道執行的其他範例，但需確認行為相同。

最後，將相關add指令（add t3, t3, t4）替換為透過其他管道執行的其他相關指令並分析模擬結果。例如，可以包含以下指令之一來代替第二條add指令：

- lw t3, (t4)（強制讀取值來自DCCM，如實驗13所述）
- mul t3, t3, t4
- div t3, t3, t4

3. 在提交階段透過轉送解除資料冒險

當一條指令與需要幾個週期才能取得結果的前一條指令（即多週期操作，例如lw、mul和div指令等）相關時，會出現一種更微妙的情況。在本部分中，我們將分析在執行lw指令和相關add指令時可能發生的特定情況，我們將其他指令和情況的分析留作練習。

如實驗13所述，當使用低延遲DCCM記憶體時，lw指令需要三個週期（DC1、DC2和DC3階段）才能取得其結果。這是本部分中使用的場景。（正如我們在實驗13和14中分析的那樣，使用外部DDR2記憶體時會產生更大的延遲 – 這種更大的記憶體延遲對資料冒險的影響留作練習。）

如果lw指令在相關add指令之前三個或三個以上週期執行，則按照第2部分所述解除冒險。在這種情況下，相應部分所述的相同10:1和3:1多路開關用於將載入指令讀取的資料轉送到依賴於它的後續指令。

附錄A：文件末尾的附錄包含按照第2部分所述處理的lw-add RAW資料冒險的範例。

但是，如果載入指令執行的時間與相關add指令執行的時間間隔較近，則按照與第2部分所述不同的方式解除冒險。現在的問題是，當add指令到達EX1階段時，lw指令讀取的值尚不可用。

在DDCARV所述的管線處理器中，這種情況下會引入氣泡（bubble），這樣相關指令便可等待至讀取值可用時才加以使用。這幾乎不需要新增硬體，但會影響效能。因此，SweRV EH1允許相關指令繼續通過管線，然後在提交階段重新計算運算（由於資料相關性而需要時）。

具體來說，SweRV EH1在每路的提交階段新增一個額外的ALU（輔助ALU）。必要時，此ALU使用適當的輸入重新計算算術邏輯運算。因此，不會因暫停而遺失週期 – 但代價是新增了兩個額外的ALU（每路一個）以及控制訊號和邏輯。圖10描繪了此輔助ALU在通路0提交階段的實作（ALU用藍框包住）以及在EX3階段為第二個輸入運算元新增的轉送邏輯（此邏輯由紅框包住）。（在實驗11的圖4中，為簡單起見，沒有包含這兩個額外的ALU和轉送路徑。）

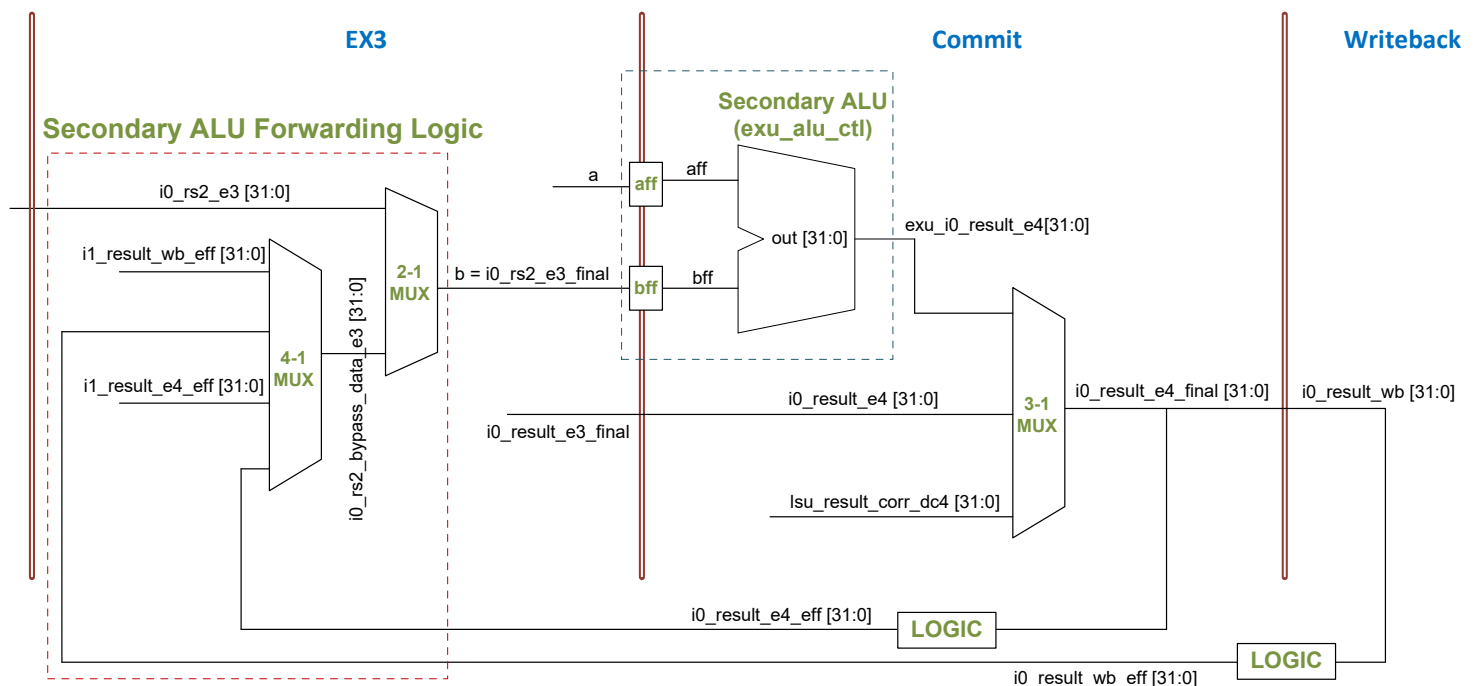


圖10. 通路0提交階段中的輔助ALU

任務：向圖10新增相應邏輯以產生IO管道中的輔助ALU的第一個輸入運算元（a）。

圖11顯示了本部分中使用的範例程式碼。它在執行一條lw指令後緊接著執行一條獨立的add指令（add t6, t6, -1：計算迴圈索引），然後執行一條與載入相關的add指令。獨立add指令用於強制lw指令和相關add指令透過通路0執行。因此，與附錄中程式的唯一區別在於lw和add指令現在更接近；不過，正如我們剛剛所說，程式中的這種微小差異會導致執行方式存在巨大差異，接下來將具體闡述。

```
.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
```

圖11. 執行lw指令、獨立add指令和相關add指令的程式

通常，資料夾[RVfpgaPath]/RVfpga/Labs/Lab15/DataHazards_Close-LW-AL提供PlatformIO專案，以便可以根據需要分析、模擬和修改程式。在PlatformIO中開啟、編譯專案，然後開啟反組譯檔案（位於[RVfpgaPath]/RVfpga/Labs/Lab15/DataHazards_Close-LW-AL/.pio/build/swervolf_nexys/firmware.dis）。請注意，lw和add指令位於位址0x000001bc和0x000001c4處。

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

圖12顯示了圖11程式中迴圈的任意一次迭代的模擬結果。同樣，由於指令快取未命中，除了第一次迭代之外的任何迭代均有效，應盡量避免這種情況。與前一部分的範例一樣，頂部的訊號（追蹤訊號）可幫助追蹤通過管線的指令。追蹤訊號下方為4:1和2:1多路開關的主要訊號以及圖11中的新ALU。

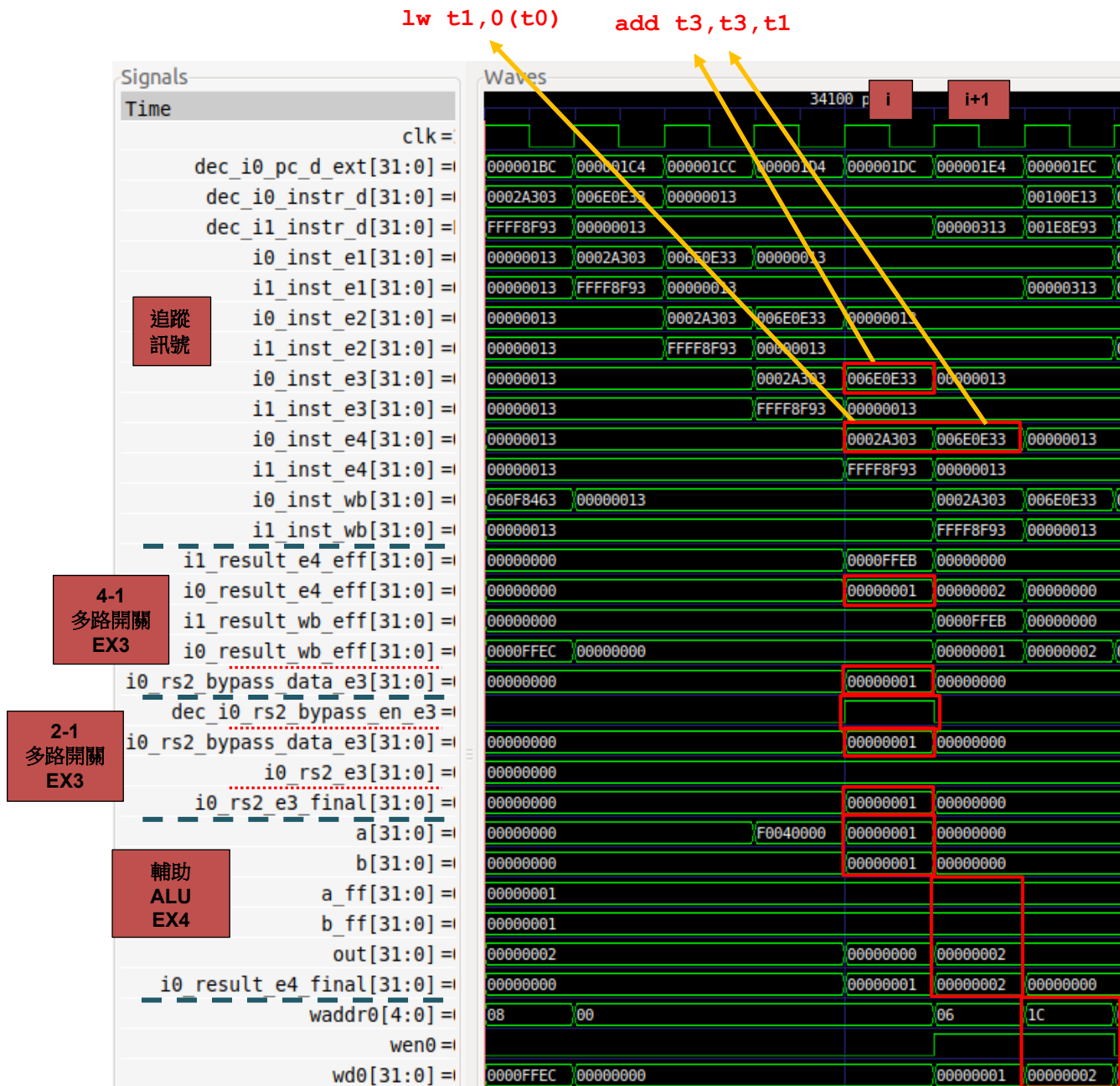


圖12. 圖11程式中迴圈的第三次迭代的模擬結果

圖13顯示了圖11程式中迴圈的第七次迭代在圖12所示的週期*i*（`add`指令處於EX3階段，`lw`指令處於提交階段）和週期*i+1*（`add`指令處於提交（即EX4）階段並重新計算適當輸入的運算）中的執行圖。

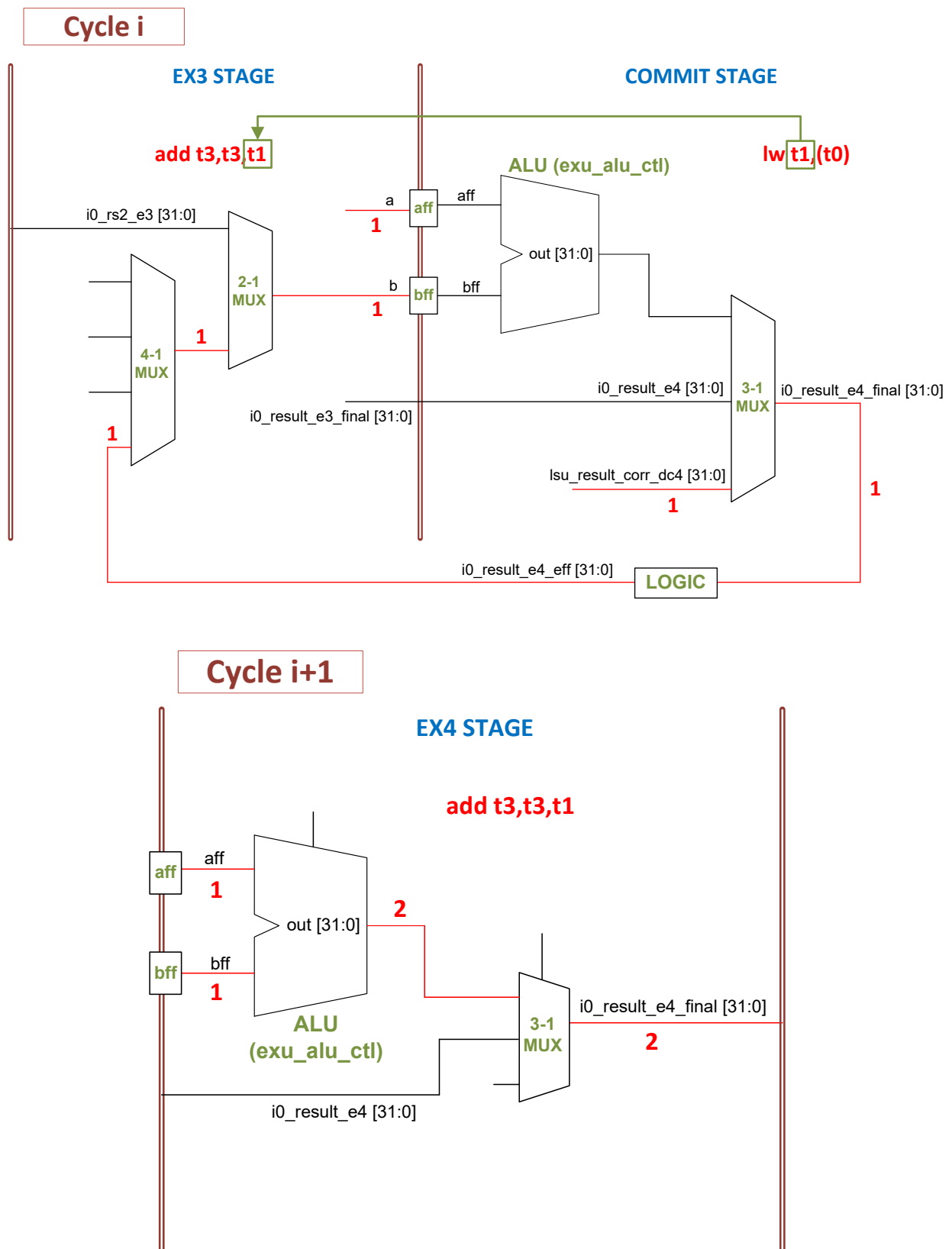


圖13. 圖11程式中迴圈的第七次迭代在圖12的週期*i*和*i+1*中的執行圖

任務：在自己的電腦上重複圖12中的模擬過程。可以使用以下位置提供的.tcl檔案：
[RVfpgaPath]/RVfpga/Labs/Lab15/DataHazards_Close-LW-AL/scriptLoad.tcl

任務：為圖11的範例繪製與圖3類似的圖。

同時分析圖12中的波形以及圖13中的圖。

- **圖12所示的追蹤訊號：**
 - o 在週期*i*中，add指令處於通路0的EX3階段 ($i0_inst_e3 = 0x006E0E33$)，lw指令處於I0管道的提交階段 ($i0_inst_e4 = 0x0002A303$)。
 - o 在週期*i+1*中，add指令處於通路0的提交階段 ($i0_inst_e4 = 0x006E0E33$)。
- **4-1多路開關：**在週期*i*中，選擇載入指令（在本週期中處於提交階段）讀取的值：
 $i0_rs2_bypass_data_e3 = i0_result_e4_eff = 0x00000001$
- **2-1多路開關：**在週期*i*中，由於載入和加法之間的相關性，選擇旁路值 ($dec_i0_rs2_bypass_en_e3 = 1$)。因此：
 $i0_rs2_e3_final = i0_rs2_bypass_data_e3 = 0x00000001$
- **提交階段ALU：**在週期*i+1*中，使用正確的值重新計算加法：
 $out = a_ff + b_ff = 0x00000001 + 0x00000001 = 0x00000002$
然後，在3:1多路開關中，選擇ALU輸出 ($exu_i0_result_e4$)。請注意，如果不存在相關性，則選擇訊號*i0_result_e4*中的值。

任務：在前一個範例中，分析如何取得add t3, t3, t1指令的第一個運算元 (t3)。可以使用以下位置提供的.tcl檔案：
[RVfpgaPath]/RVfpga/Labs/Lab15/DataHazards_Close-LW-AL/scriptLoad_FirstOperand.tcl

任務：刪除圖11的範例中的nop指令並使用硬體計數器取得IPC。

任務：停用實驗11所述的輔助ALU，並透過Verilator模擬和板上執行方式分析圖11中的範例。

任務：在圖11的範例中，將add t6, t6, -1指令移至add t3, t3, t1指令之後，然後重新檢查模擬中以及板上的程式。

4. 練習

- 1) 透過新增與add指令結果相關的額外算術-邏輯指令，修改第3部分中使用的程式。例如，可以將圖11中的迴圈替換為以下程式碼，其中包含一條新的AND指令（**and t3, t4, t3**），並透過前移指令**add t5, t5, 0x1**對程式碼順序稍作調整：

```
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模擬並說明如何為新的A-L指令處理資料冒險。然後刪除所有nop指令並分析硬體計數器提供的結果。

- 2) 分析與第3部分所述相同的情況：mul指令後跟使用乘法結果的add指令。在圖11的程式中，只需將lw替換為寫入暫存器t1的mul。

- 3) 分析lw指令後跟與載入讀取的值相關的mul指令這種情況。在圖11的程式中，只需將相關add指令替換為mul指令。

- 4) （以下練習基於[PaHe]的練習4.18、4.19、4.20和4.26。）

假設在不處理資料冒險的SweRV EH1處理器版本上執行了以下程式碼（即，程式設計師負責透過在必要時插入nop來處理資料冒險）。向程式碼中新增nop以使其正確執行。

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


然後組成包含至少三個組合語言程式碼片段的序列，這些片段顯示不同類型的RAW資料冒險。RAW資料相關性的類型由產生結果的階段和使用結果的下一條指令標識。

對於每個序列，要使程式碼在沒有轉送或冒險偵測的SweRV EH1處理器上正確執行，需要在何處插入多少條nop？如果我們使用SweRV EH1中提供的轉送而不插入nop，那麼CPI是多少？

5) 在實驗14第2.C部分的程式（位於 `[RVfpgaPath]/RVfpga/Labs/Lab14/LW_Instruction_ExtMemory`）中，將指令 `add x1, x1, 1` 替換為 `add x28, x1, 1`。這將在修改後的add指令和迴圈開頭的非阻塞載入（`lw x28, (x29)`）之間引入WAW冒險。利用模擬分析SweRV EH1中如何處理此冒險，可以在暫存器檔案中查看訊號wen2的值。嘗試瞭解控制單元（模組dec）中如何計算此訊號。

6) 在實驗14第2.C部分的程式（位於 `[RVfpgaPath]/RVfpga/Labs/Lab14/LW_Instruction_ExtMemory`）中，將指令 `add x1, x1, 1` 替換為 `add x1, x28, 1`。這將在修改後的add指令和迴圈開頭的非阻塞載入（`lw x28, (x29)`）之間引入RAW冒險。透過模擬分析SweRV EH1中如何處理此冒險。

7) 在實驗14第2.C部分的程式（位於 `[RVfpgaPath]/RVfpga/Labs/Lab14/LW_Instruction_ExtMemory`）中，將指令 `add x1, x1, 1` 替換為 `add x1, x28, 1`，將指令 `add x7, x7, 1` 替換為 `add x28, x7, 1`。這將導致發生RAW和WAW冒險。利用模擬分析SweRV EH1中如何處理這兩種冒險。

8) 儲存-載入轉送

這是一種非常值得關注的情況，我們沒有在本實驗中分析，但您可在本練習中進行分析。當儲存以及隨後的載入存取相同位址時，可以在核心內將資料從儲存轉送到載入，無需讀取DDR外部記憶體，進而節省時間和功耗。

實作這種轉送的邏輯包含在LSU中，具體來說是包含在模組lsu_bus_intf和lsu_bus_buffer中，必須在本練習中進行檢查。

`[RVfpgaPath]/RVfpga/Labs/Lab15/Sw-Lw-Forwarding`中的PlatformIO專案用於說明儲存-載入轉送。此資料夾中提供了.tcl指令碼，您可以使用它來分析迴圈的任意一次迭代以及瞭解SweRV EH1中如何執行儲存-載入轉送。

附錄A

本附錄包含按照第2部分所述處理的lw-add RAW資料冒險的範例。圖14顯示了本附錄中使用的範例程式碼。它在執行1條lw指令後接著執行5條nop指令，然後執行1條與載入相關的add指令。中間的nop指令用於將兩條相關指令分開。

```
.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
```

圖14. 執行lw、5條nop和1條相關add指令的程式

通常，資料夾RVfpgaPath]/RVfpga/Labs/Lab15/DataHazards_FarAway-LW-AL提供PlatformIO專案，以便可以根據需要分析、模擬和修改程式。開啟、編譯專案，然後開啟反組譯檔案（位於RVfpgaPath]/RVfpga/Labs/Lab15/DataHazards_FarAway-LW-AL/.pio/build/swervolf_nexys/firmware.dis）。請注意，lw和add指令位於位址0x000001b0和0x000001c8處。

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

圖15顯示了圖14程式中迴圈的第三次迭代的模擬結果。同樣，由於指令快取未命中，除了第一次迭代之外的任何迭代均有效，應盡量避免這種情況。與主要實驗的範例一樣，頂部的訊號（追蹤訊號）可幫助追蹤通過管線的指令。追蹤訊號下方為每個多路開關的主要訊號。每個多路開關的訊號被藍色虛線包住。圖中描繪了每個多路開關的控制訊號、輸入和輸出（與主要實驗中一樣）。

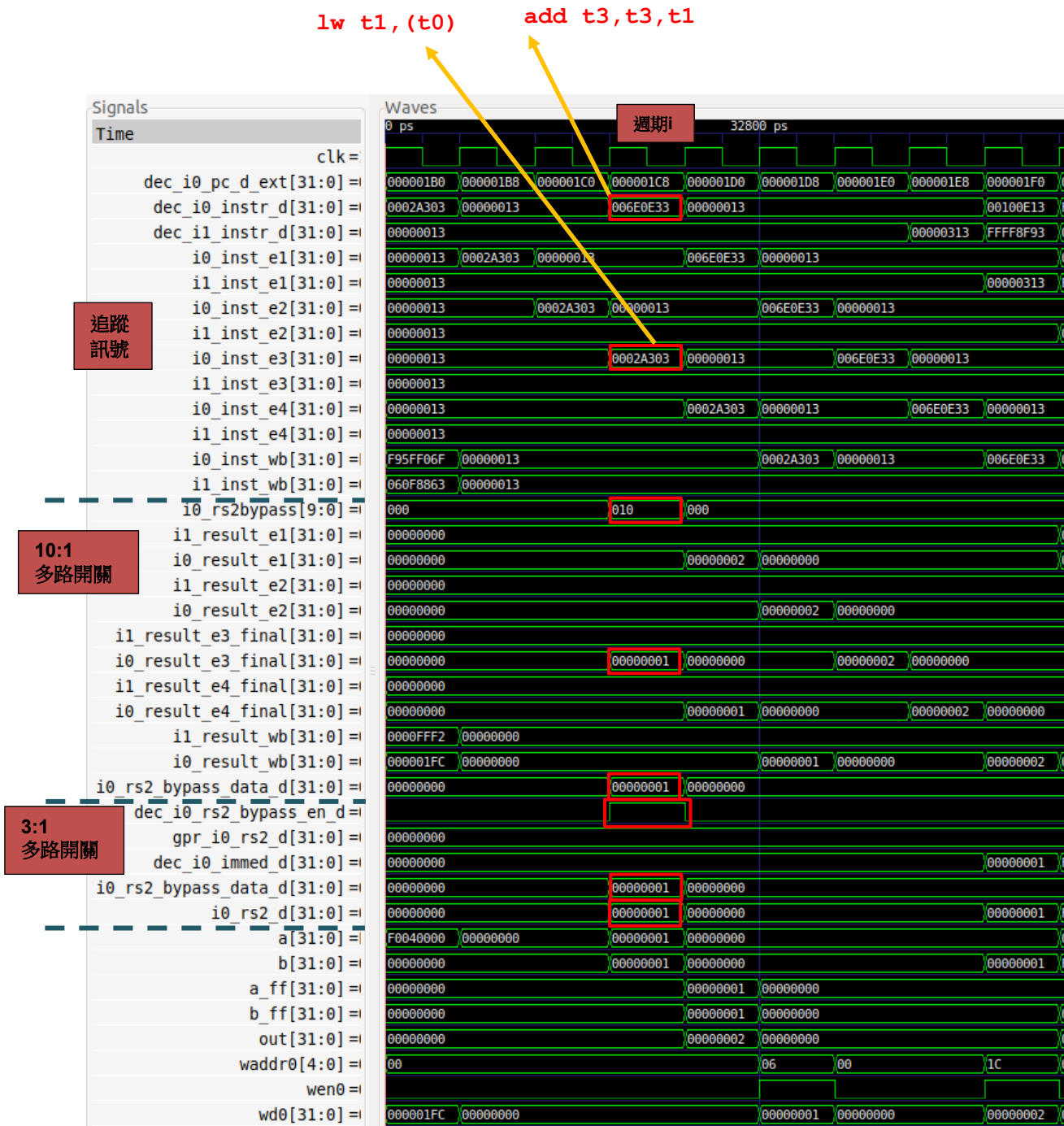


圖15. 圖14程式中迴圈的第三次迭代的模擬結果

圖16顯示了圖14程式在圖15所定義的週期*i*（add指令處於解碼階段，lw指令處於DC3階段）中的執行圖。

Cycle i

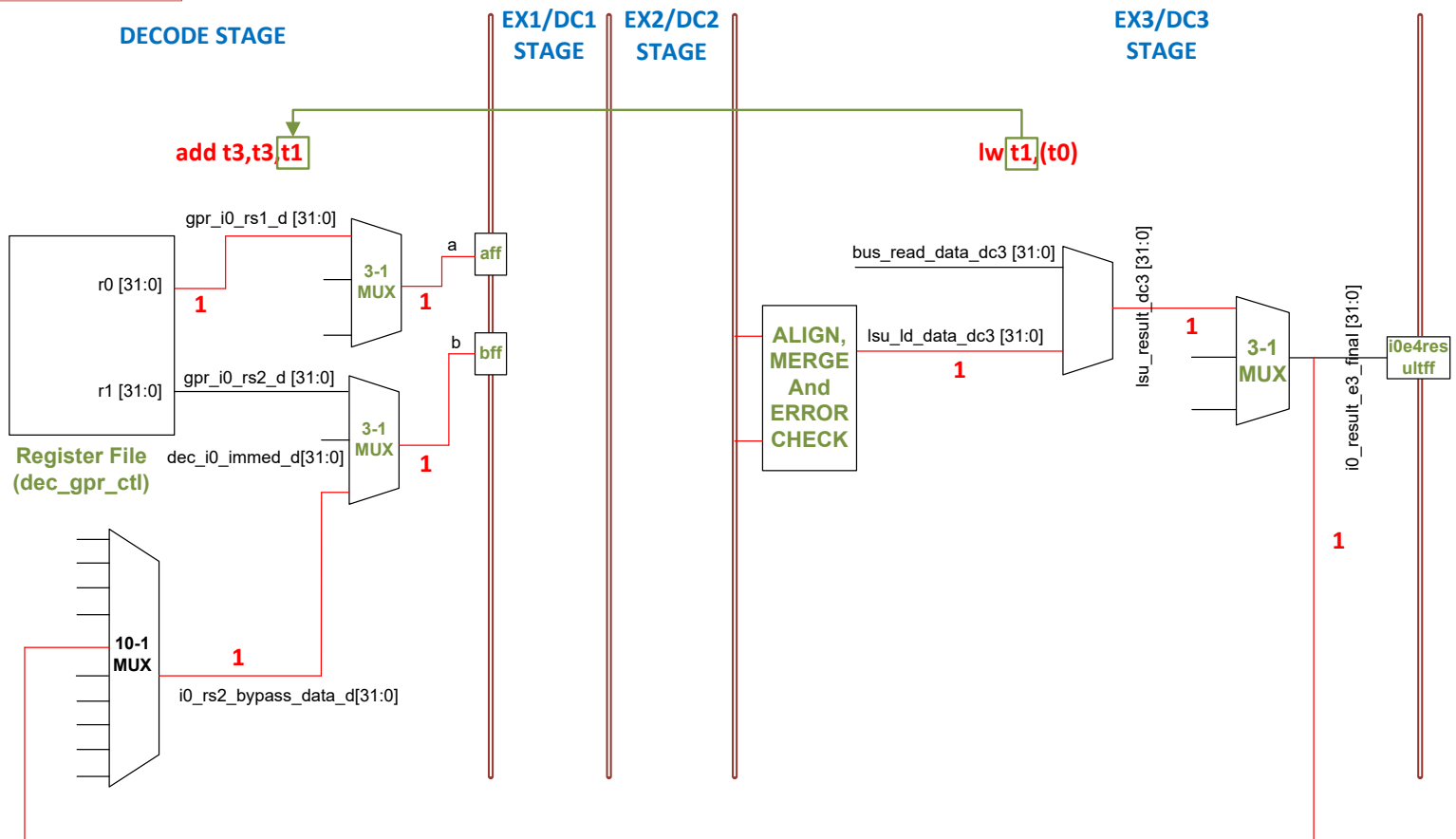


圖16. 圖14程式中迴圈的第三次迭代在圖15顯示的第四個週期中執行期間的硬體

任務： 在自己的電腦上重複圖15中的模擬過程。

同時分析圖15中的波形以及圖16中的圖。

- 圖15所示的追蹤訊號：

- 在週期*i*中，add指令處於通路0的解碼階段（dec_i0_instr_d = 0x006E0E33），lw指令處於I0管道的DC3階段（i0_inst_e3 = 0x0002A303）。
- **10:1多路開關：** 在週期*i*中，訊號i0_rs2bypass[9:0] = 0x010（即 i0_rs2bypass[4] = 1），因此輸出與來自I0管道EX3/DC3階段值相連（參見圖16）：
i0_rs2_bypass_data_d = i0_result_e3_final = 0x00000001
- **3:1多路開關：** 在週期*i*中，訊號dec_i0_rs2_bypass_en_d = 1，因此輸出與來自旁路邏輯的值相連（參見圖9）：
i0_rs2_d = i0_rs2_bypass_data_d = 0x00000001

任務：比較上述場景在SweRV EH1和DDCARV所述管線處理器中的處理方式。

任務：如果仔細比較圖16和實驗13的圖6，將看到實驗13的圖6中lw指令讀入暫存器檔案的值（訊號lsu_ld_data_corr_dc3[31:0]）與圖16中lw轉送的值（訊號lsu_ld_data_dc3[31:0]）不同。兩個值的區別在於前者經過模組lsu_ecc中的ECC邏輯檢查，而後者沒有。說明為什麼lw轉送的值未經過錯誤檢查也沒有問題。

任務：在圖14的範例中，刪除lw之前和add之後的所有nop指令。不要刪除兩條相關指令之間的5個nop。分析模擬，然後透過在開發板上執行程式，用效能計數器計算IPC（在測量IPC時保留nop指令似乎並不適合，因為它們是無用指令；不過，程式本身就是無用的，這裡我們的唯一目的是分析並瞭解資料冒險）。