



IMAGINATION大學計劃

# RVfpga實驗12

算術/邏輯指令：add指令

## 1. 簡介

在本實驗中，我們將分析SweRV EH1管線各個階段的算術和邏輯指令流。圖1顯示了EH1微架構的高階檢視，我們在本實驗中分析的階段以紅色強調顯示：I0管道的解碼、EX1、EX2、EX3、提交（有時稱為EX4）和寫回階段。（I1管道幾乎與I0管道相同，但我們將其深入分析推遲到研究超標量處理的實驗17。我們還會在實驗11和實驗16中分析擷取和對齊階段。）

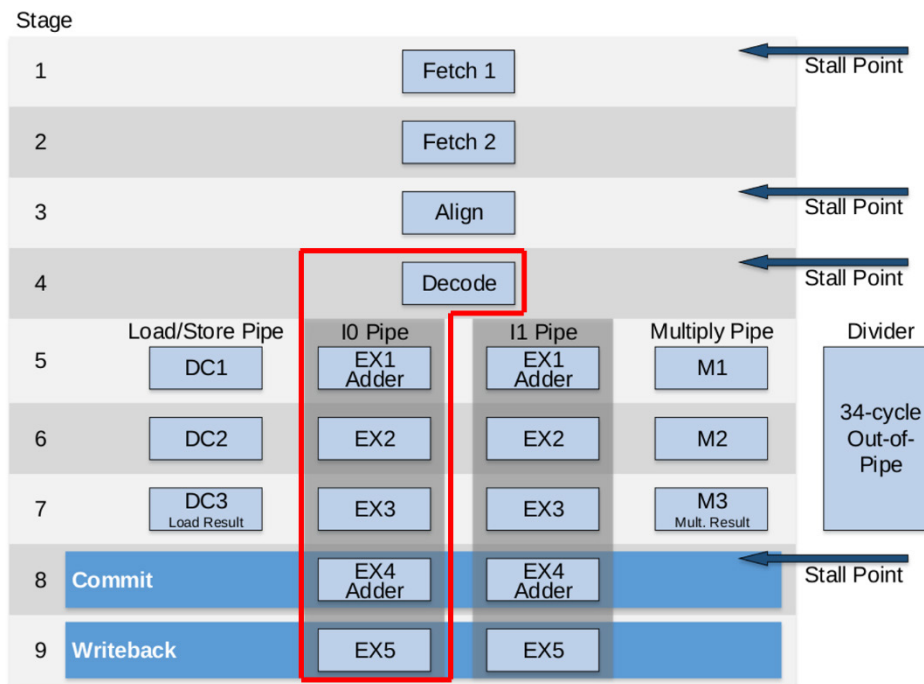


圖1. SweRV EH1管道：強調顯示了add指令的第4-9階段

在第2部分中，我們將分析add指令的解碼到寫回階段（此時它將結果寫入暫存器檔案）。在說明過程中，我們會交錯模擬add指令，您應當在自己的電腦上重複此模擬過程。在第3部分中，我們將提供相關練習以按照與add指令所述步驟類似的過程分析其他算術邏輯指令。

## 2. SweRV EH1核心add指令的分析

在本部分中，我們將使用圖2顯示的範例，該範例執行無限迴圈中包含的add指令。資料夾 [\[RVfpgaPath\]/RVfpga/Labs/Lab12/ADD\\_Instruction](#) 提供了PlatformIO專案，這樣便可根據需要分析、模擬和變更程式。如SweRVref文件的第2部分所述，為簡單起見，本專案中禁止使用壓縮指令。此外，為方便起見，我們會在無限迴圈中插入add指令，這樣一來，如果我們避免分析迴圈的第一次迭代，則可以在沒有指令快取（I\$）未命中的情況下進行檢查。這樣做還能輕鬆地在模擬中找到關注的區域。最後，正如我們在相關實驗包含的範例中執行的操作一樣，add指令（在圖2中以紅色強調顯示）前後存在幾條nop（無操作）指令，以便將其與屬於迴圈的其他迭代的前方/後方add指令隔離。

```
.globl main
main:

li t3, 0x4          # t3 = 4
li t4, 0x1          # t4 = 1

REPEAT:
    INSERT_NOPS_10
    add t3, t3, t4    # t3 = t3 + t4
    INSERT_NOPS_10
    beq zero, zero, REPEAT # Repeat the loop

.end
```

**圖2. add指令範例**

如果在PlatformIO中開啟、編譯專案，然後開啟反組譯檔案（位於 *[RVfpgaPath]/RVfpga/Labs/Lab12/ADD\_Instruction/.pio/build/swervolf\_nexys/firmware.dis* 中），則會發現add指令（0x01de0e33）位於該程式中的位址0x00000108處。

```
0x00000108:  01de0e33      add    t3,t3,t4
```

**任務：**驗證這些32位元（0x01de0e33）是否對應於RISC-V架構中的指令add t3,t3,t4。

## A. add指令的基本分析

圖3顯示了圖2中程式的Verilator模擬，其中顯示了圖2中的程式在迴圈的第四次迭代中執行add指令。這張圖包括與訊號解碼、EX1和寫回（Writeback，WB）階段相關的一些訊號。以紅色強調顯示的值對應於add指令，因為該指令透過I0管道遍歷這三個階段。請注意，圖中所示的訊號對應於I0管道。

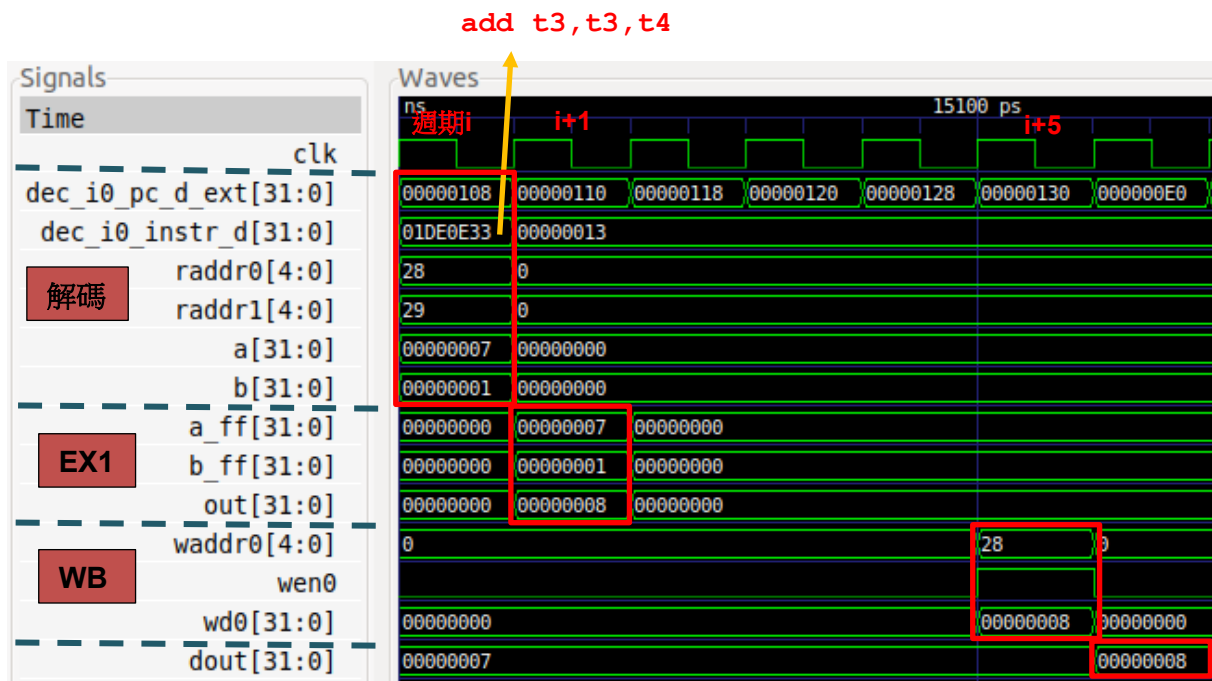


圖3. 圖2中範例程式的Verilator模擬

圖4顯示了透過I/O管道在迴圈的第四次迭代期間執行add指令的SweRV EH1管線簡化圖（參見圖2中的程式）。請注意圖中合併了處理器在不同時鐘週期的狀態：

- 週期i：                   **解碼**：對指令進行解碼並讀取暫存器檔案。透過I/O管道傳送Add指令。
- 週期i+1：               **EX1**：透過ALU計算加法。
- 週期i+5：               **寫回**：使用寫入連接埠0將加法結果寫入暫存器檔案。

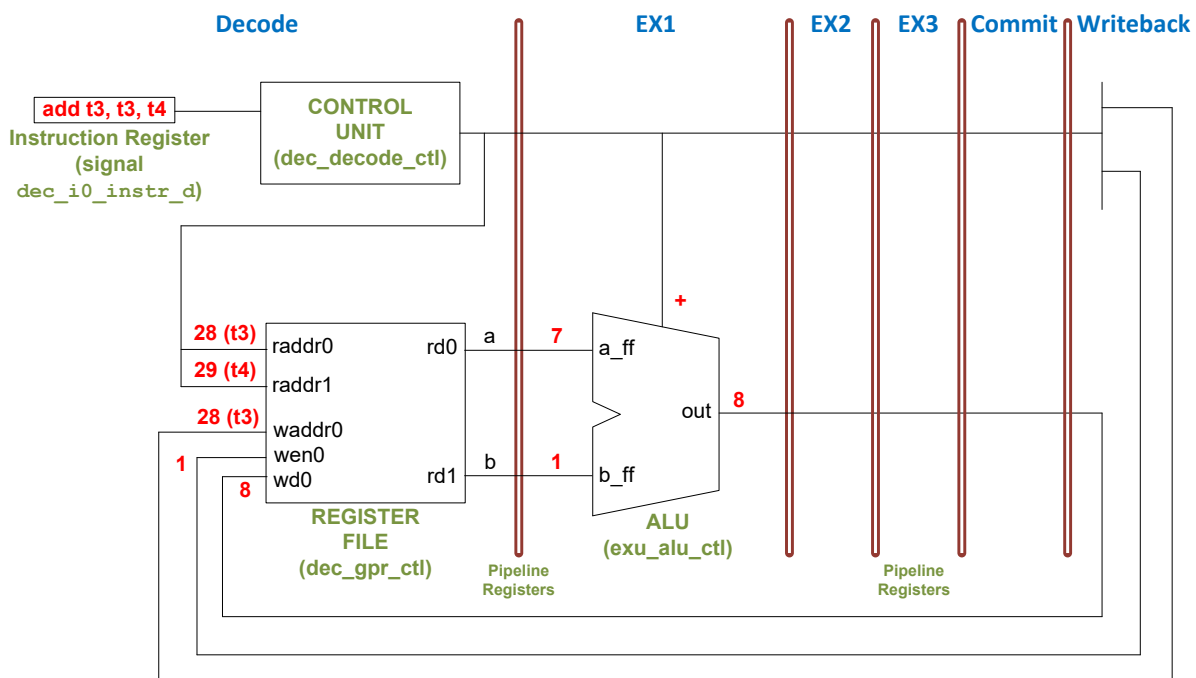



圖4. 執行add指令的SweRV EH1管線

**任務：**在自己的電腦上重複圖3中的模擬過程。為此，請按照以下步驟操作（在GSG的第7部分中詳述）：

- 必要時產生模擬二進位檔案（*Vrvfpgasim*）。
- 在PlatformIO中，開啟在以下位置提供的專案：  
*[RVfpgaPath]/RVfpga/Labs/Lab12/ADD\_Instruction*。
- 在檔案*platformio.ini*中建立到RVfpga模擬二進位檔案（*Vrvfpgasim*）的正確路徑。
- 使用Verilator產生模擬軌跡（產生軌跡）。
- 在GTKWave上開啟軌跡。
- 使用檔案*test\_1.tcl*（在*[RVfpgaPath]/RVfpga/Labs/Lab12/ADD\_Instruction/*中提供）開啟與圖3所示訊號相同的訊號。為此，在GTKWave上，按一下「*File – Read Tcl Script File*」（檔案 – 讀取Tcl指令碼檔案）並選擇*test\_1.tcl*檔案。
- 按幾次「*Zoom In*」（放大）（）移動至15000 ps。

我們將同時分析圖3中的波形和圖4中的圖，以此瞭解管線中的add指令，具體如下所述。

- **週期i：** **解碼：**訊號dec\_i0\_instr\_d包含32位元機器指令0x01DE0E33。在RISC-V中，add指令的操作碼如下（參見[DDCARV]的附錄B）：

```
00      | rs1 | 000 | rd | 0110011
```

可以輕鬆驗證0x01DE0E33是否對應於：add t3, t3, t4（請記住，t3=x28且t4=x29）。

在這一階段，將產生**控制訊號**並**讀取暫存器檔案**。在下一階段（EX1），運算元將透過IO管道傳送到ALU。訊號raddr0和raddr1（圖中以十進位表示）包含add指令的兩個來源暫存器編號，訊號a和b包含將在下一個（EX1）階段傳送到ALU的值。本例中的a和b是從

暫存器檔案中讀取的值。對於其他指令，a和b可能是不同的值；例如，b可能是立即數。我們將在後面的實驗中分析其他指令。

- 週期i+1： **EX1：執行**add指令。訊號a\_ff和b\_ff包含ALU的輸入（本例中分別為7和1），而訊號out包含加法的結果（8）。
- 週期i+5： **寫回**：最後，在4個週期後，加法的結果透過訊號wd0 = 0x8寫回到暫存器檔案中，其中包含要寫入的資料。鑒於本週期內wen0 = 1（寫入啟用），加法結果會在週期結束時寫入暫存器x28（以十進位表示，waddr0 = 28）。可以發現，在接下來的週期（圖中最後一個週期）中，暫存器x28已更新為新值（dout = 8）。

請記住，可透過GTKwave輕鬆更改訊號的資料格式。為此，將游標置於訊號上，按一下鼠標右鍵，然後選擇所需的「資料格式」。例如，waddr0採用十進位格式（28）而不是十六進位格式（0x1C）將更方便查看，如圖5所示。



圖5. 以十進位格式顯示的訊號waddr0

## B. add指令的進階分析

本部分將比A部分更詳細地分析add指令遍歷的各個階段（從解碼到寫回），逐漸向圖3中的模擬新增更多訊號。

圖6顯示了add指令透過I0管道執行期間遍歷的主要元素的詳細圖。實驗11的圖4已提供相關圖示（建議將兩個圖進行比較），但現在只關注I0管道並提供與add指令相關的詳細資訊。可能需要將圖放大才能看到詳細資訊。控制訊號的名稱顯示為紅色，而資料訊號的名稱顯示為黑色。這些名稱是SweRV EH1 Verilog模組中使用的實際名稱。等號（=）表示Verilog程式碼中的訊號指定。

**任務：**在SweRV EH1處理器的Verilog檔案中找到圖6中的主要結構和訊號。

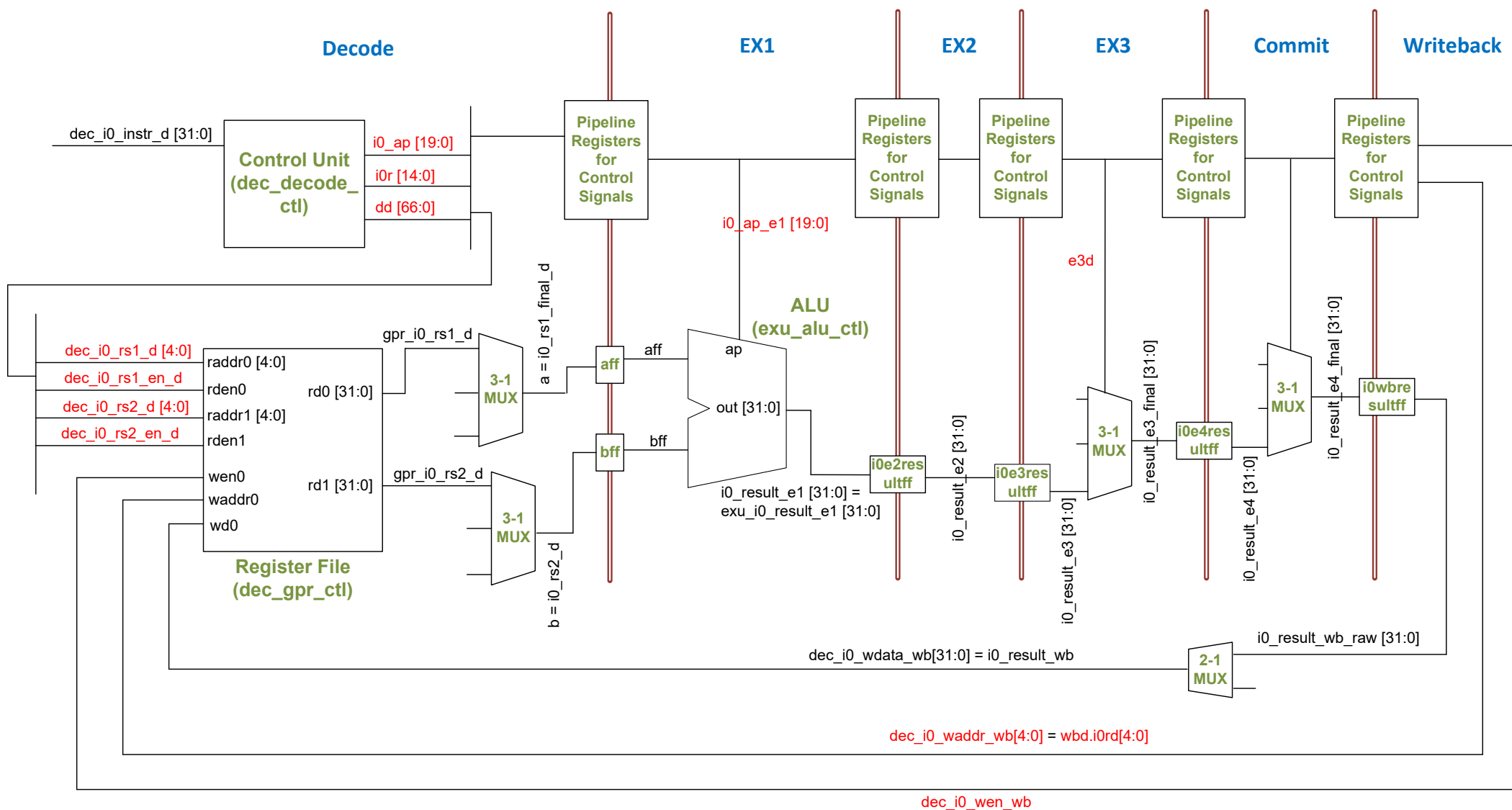


圖6. 經過I0管道的算術邏輯指令使用的主要單元

## i. 解碼階段

如實驗11中所述，解碼階段負責兩個主要任務：

- 對指令進行解碼並產生控制訊號。
- 讀取或組合來源運算元並將指令傳送到適當的管道。

接下來將針對add指令分析上述每項任務並將一些相關訊號新增到模擬中。

### 對指令進行解碼並產生控制訊號：

如實驗11的第2.C.i部分所述，

[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/include/swerv\_types.sv中定義了多個結構來對控制位元進行分組。其中三個結構與算術邏輯（Arithmetic-Logic，A-L）指令直接相關：

- **alu\_pkt\_t**：這是A-L指令的主要結構：

```
190 typedef struct packed {
191     logic valid;
192     logic land;
193     logic lor;
194     logic lxor;
195     logic sll;
196     logic srl;
197     logic sra;
198     logic beq;
199     logic bne;
200     logic blt;
201     logic bge;
202     logic add;
203     logic sub;
204     logic slt;
205     logic unsign;
206     logic jal;
207     logic predict_t;
208     logic predict_nt;
209     logic csr_write;
210     logic csr_imm;
211 } alu_pkt_t;
```

該類型的兩個訊號i0\_ap（通路0）和i1\_ap（通路1）於解碼階段在模組dec\_decode\_ctl內部定義和指定，並在後續執行（EX1-4）階段傳播（通路0為訊號i0\_ap\_e1、i0\_ap\_e2、i0\_ap\_e3和i0\_ap\_e4，通路1為訊號i1\_ap\_e1、i1\_ap\_e2、i1\_ap\_e3和i1\_ap\_e4）。它們包含用於向ALU告知其必須執行的操作的控制訊號。當執行add指令時，i0\_ap/i1\_ap除了下面兩位元控制訊號外的所有位元均設定為0：

- o valid：表示這是一條有效的ALU指令
- o add：表示這是一條add指令

當通路0/1中的指令不是A-L指令時，訊號i0\_ap/i1\_ap的所有位元均為0（具體來說，valid = 0），這樣一來I0/I1 ALU根本不起作用。

- **reg\_pkt\_t**：該類型的訊號i0r（通路0）和i1r（通路1）在模組dec\_decode\_ctl內部定義、指定和使用。它們包含兩個來源暫存器（欄位rs1和rs2）的編號和目標暫存器（欄位rd）的編號：



```

183 typedef struct packed {
184     logic [4:0] rs1;
185     logic [4:0] rs2;
186     logic [4:0] rd;
187 } reg_pkt_t;

```

- **dest\_pkt\_t**：該類型的訊號在解碼階段在模組**dec\_decode\_ctl**內部定義和指定，並在所有剩餘階段（訊號e1d、e2d、e3d、e4d和wbd）傳播。它包含多個欄位，例如通路0和通路1中指令的目標暫存器：分別為i0rd[4:0]和i1rd[4:0]。

其中一些訊號用於解碼階段，不會透過控制管線暫存器傳播到後面的階段。i0r.rs1/i1r.rs1和i0r.rs2/i1r.rs2就屬於這種情況，它們在解碼階段直接提供給暫存器檔案以讀取兩個輸入運算元（訊號raddr0、raddr1、raddr2和raddr3）。

**任務：**在Verilog程式碼（模組**dec\_decode\_ctl**）中尋找如何使用i0r控制訊號在解碼階段讀取暫存器檔案。

不過，其他控制訊號必須傳播到後面的階段。i0\_ap/i1\_ap就屬於這種情況，**ALU**使用它來瞭解必須執行的操作（本例中為加法），dd也屬於這種情況，暫存器檔案使用它來寫入兩個結果。

**任務：**在Verilog程式碼（模組**exu**）中尋找i0\_ap和dd控制訊號如何從解碼階段傳播到執行（**EX1**）階段。此外，還需要尋找dd控制訊號遍歷解碼到寫回的所有階段之後，如何在寫回階段被暫存器檔案使用。

### 讀取或組合來源運算元並將指令傳送到適當的管道：

如實驗11中所述，**SweRV EH1**處理器包含多個用於執行指令的管道。在解碼階段，指令一旦被解碼，就必須透過適當的管道進行調度。具體來說，如果**A-L**指令在通路0上，則必須盡可能傳送到**I0**管道；類似地，如果**A-L**指令在通路1上，則必須盡可能傳送到**I1**管道。在本實驗分析的程式（圖2）中，一旦處理器在通路0上對add指令進行解碼（即，其「已知」這是一條**A-L**指令，因此必須將其傳送到**I0**管道），則必須檢查是否滿足透過**I0**管道執行的所有條件：有效解碼？2個輸入運算元可用？管線未阻塞？...在本例中，此項檢查的結果透過兩個狀態訊號傳送到**I0**管道，這兩個訊號在**dec\_decode\_ctl**模組中計算，供**exu**模組中的**ALU**使用（在下一小節中，我們將更詳細地說明**ALU**）。這兩個狀態訊號為：

- **i0\_e1\_ctl\_en**（在**ALU**內部重新命名為enable）：該訊號取決於dec\_i0\_ctl\_en[4:1]，它在解碼時確定通路0的每個執行階段（**EX1-3**）以及提交階段必須啟用（1）還是停用（0）。請注意，根據不同情況（分支預測錯誤、除法計算錯誤等）的影響，該指令可能是非法的，或者管線可能被阻塞和清除等，這些都會停用管線。

- `dec_i0_alu_decode_d` (在ALU內部重新命名為`valid`)：如果通路0處的算術/邏輯指令已合法解碼並且不使用輔助ALU（我們將在實驗15中說明該結構），則該訊號為1。

兩個訊號均必須為1，ALU才能在下一階段（EX1階段）執行add操作。

**任務：**這兩個訊號（`i0_e1_ctl_en`和`dec_i0_alu_decode_d`）的產生過程相當複雜，這裡不做詳細說明，但您可自行在模組`dec_decode_ctl`和`exu`中進一步分析。

同樣如實驗11中所述，輸入運算元透過解碼階段實現的兩個3:1多路開關提供給I0管道（`i0_rs1_final_d`和`i0_rs2_d`）（參見圖6）。在本範例的add指令中，兩個輸入運算元均從暫存器檔案中直接取得：

- 第一個輸入運算元：`i0_rs1_final_d[31:0] = gpr_i0_rs1_d[31:0]`
- 第二個輸入運算元：`i0_rs2_d[31:0] = gpr_i0_rs2_d[31:0]`

**任務：**在Verilog程式碼（模組`exu`）中尋找底部的3:1多路開關（第二個輸入運算元）並嘗試找到其輸入的來源（圖6中僅顯示來自暫存器檔案的輸入）。不需要太仔細地查看輸入，因為它們將在第3部分和後續實驗提供的練習中進行分析。

圖7透過新增上文所述訊號延伸圖3中的Verilator模擬：

- `i0_ap[19:0]`
- `i0_ap.valid`（透過下述`tcl`指令碼中的別名在圖中命名為`i0_ap_valid`，指令碼位於[RVfpgaPath]/RVfpga/Labs/Lab12/ADD\_Instruction/test\_2.tcl）
- `i0_ap.add`（透過下述`tcl`指令碼中的別名在圖中命名為`i0_ap_add`）
- `i0r[14:0]`
- `raddr0`
- `raddr1`
- `i0_e1_ctl_en`（在ALU內部重新命名為`enable`）
- `dec_i0_alu_decode_d`（在ALU內部重新命名為`valid`）
- `gpr_i0_rs1_d[31:0]`
- `gpr_i0_rs2_d[31:0]`

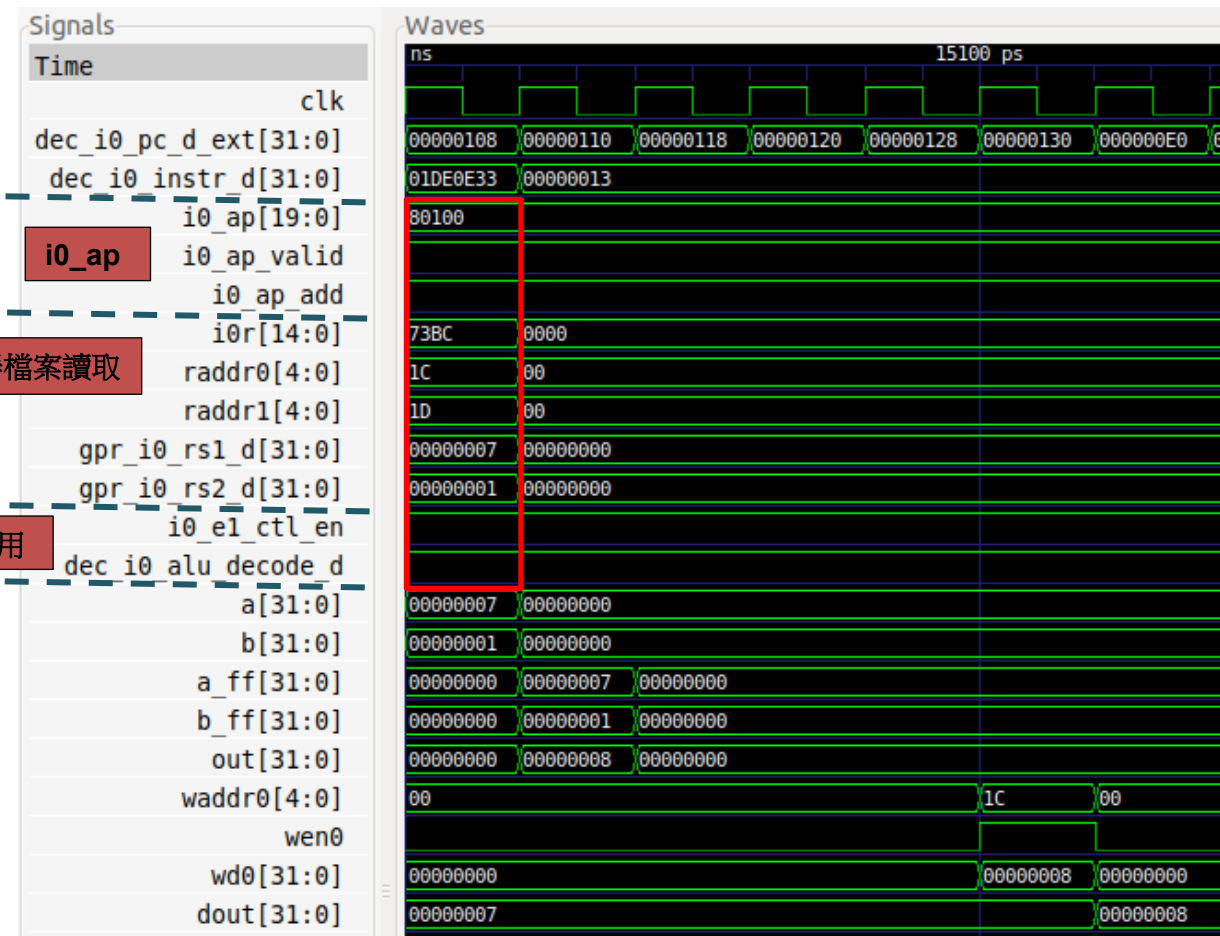


圖7. 圖2範例程式的Verilator模擬，包括控制訊號和暫存器檔案讀取連接埠

**任務：**在自己的電腦上重複圖7中的模擬過程。可以使用以下位置提供的.tcl指令碼：  
[RVfpgaPath]/RVfpga/Labs/Lab12/ADD\_Instruction/test\_2.tcl。請注意，該.tcl檔案中為一些控制位元使用了別名。

分析圖7中的波形。如上文所述，除valid和add位元以外，i0\_ap的所有位元均為0。此外，訊號i0r包含add指令的兩個來源暫存器和一個目標暫存器的識別碼。使用i0r.rs1和i0r.rs2存取暫存器檔案（參見訊號raddr0和raddr1）並將讀取的值提供給I0管道：gpr\_i0\_rs1\_d = a = 0x7且gpr\_i0\_rs2\_d = b = 0x1。最後，可以看到valid和enable訊號均為1，因此I0管道ALU將在下一個週期中使用。

**任務：**在圖2的範例中，將add指令替換為非A-L指令（例如mul指令）。驗證i0\_ap訊號的所有欄位是否均等於0，等於0時I0 ALU不起作用（對於該指令，EX1階段I0管道的訊號a\_ff和b\_ff將保持不變）。可以使用與圖7範例所用檔案相同的test\_2.tcl檔案。

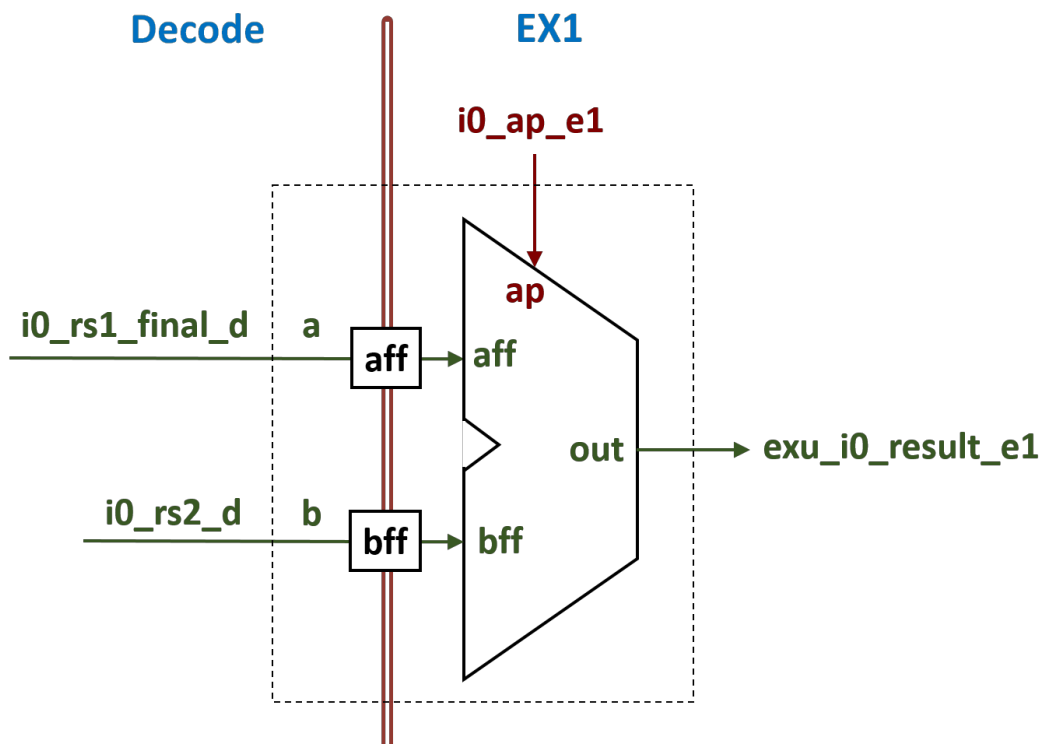
## ii. 執行階段

如實驗11中所述，SweRV EH1包含四個執行管道（參見實驗11中的圖4）：I0/I1、乘法和L/S管道。此外，它還包含一個非管線化除法器。每個管道分為3個階段：**EX1/EX2/EX3**（I0/I1管道）、**M1/M2/M3**（乘法管道）和**DC1/DC2/DC3**（L/S管道）。在本實驗中，我們將重點關注I0管道，其中會執行add指令。Add指令的I0管道的主要任務是在ALU中計算加法並將其傳播到提交階段。

### i. EX1階段

該階段執行ALU操作 – 本例中為加法。SweRV EH1的算術邏輯單元（Arithmetic-Logical Unit, ALU）在模組**exu\_alu\_ctl**（位於 `[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/exu/exu_alu_ctl.sv`）中實現，並在模組**exu**（位於 `[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/exu.sv`）中實例化。

圖8顯示了I0管道EX1階段中包含的ALU的實例化過程，以及ALU及其一些輸入/輸出連接埠的簡化圖。請注意，大多數輸入/輸出訊號在ALU中均已重新命名。



```

401     exu_alu_ctl i0_alu_e1 (.*,
402         .freeze      ( freeze                ), // I
403         .enable      ( i0_e1_ctl_en          ), // I
404         .predict_p    ( i0_predict_newp_d     ), // I
405         .valid       ( dec_i0_alu_decode_d    ), // I
406         .flush       ( exu_flush_final        ), // I
407         .a           ( i0_rs1_final_d[31:0]    ), // I
408         .b           ( i0_rs2_d[31:0]         ), // I
409         .pc          ( dec_i0_pc_d[31:1]      ), // I
410         .brimm       ( dec_i0_br_immed_d[12:1] ), // I
411         .ap          ( i0_ap_e1               ), // I
412         .out         ( exu_i0_result_e1[31:0] ), // 0
413         .flush_upper ( exu_i0_flush_upper_e1  ), // 0
414         .flush_path  ( exu_i0_flush_path_e1[31:1] ), // 0
415         .predict_p_ff ( i0_predict_p_e1       ), // 0
416         .pc_ff       ( exu_i0_pc_e1[31:1]     ), // 0
417         .pred_correct ( i0_pred_correct_upper_e1 ), // 0
418     );

```

圖8. I0的ALU（exu\_alu\_ctl模組）：高階視圖和Verilog程式碼

**ALU輸入：**ALU輸入（a和b）在解碼階段由圖6中所示的兩個3:1多路開關選擇（如上一部分所述）。在exu\_alu\_ctl模組內部，當valid和enable訊號均為1時，兩個暫存器（aff和bff）將運算元從解碼階段傳播到EX1階段。

**ALU控制訊號：**ALU由訊號i0\_ap在解碼階段產生的控制位元支配（請記住，這是一種alu\_pkt\_t類型結構）。該訊號透過管線暫存器傳播（如上一部分所述）。在EX1中，該訊號稱為i0\_ap\_e1，它在ALU內部重新命名為ap（參見圖8）。

**ALU輸出：**ALU輸出在訊號exu\_i0\_result\_e1中取得（參見圖8）。該訊號使用新的管線暫存器傳播到EX2（參見圖6），它位於模組dec\_decode\_ctl中（訊號首先指定給i0\_result\_e1）：

```

2256     assign i0_result_e1[31:0] = exu_i0_result_e1[31:0];
2260     rvdffe #(32) i0e2resultff (.*, .en(i0_e2_data_en), .din(i0_result_e1[31:0]), .dout(i0_result_e2[31:0]));

```

**任務：**將本部分中分析的新訊號新增到圖7的模擬中。

**任務：**對sub指令執行與圖7中的模擬類似的模擬。請記住，可以透過.tcl檔案將新訊號新增到模擬中。

**任務：**分析模組exu\_alu\_ctl中實作的加法器/減法器的Verilog實作。圖9透過顯示與加法和減法運算直接相關的邏輯來提供一些幫助。可以使用Verilator模擬作為幫助。

```

90     rvdffe #(32) aff (.*, .en(enable & valid), .din(a[31:0]), .dout(a_ff[31:0]));
91
92     rvdffe #(32) bff (.*, .en(enable & valid), .din(b[31:0]), .dout(b_ff[31:0]));

```

```

135     assign bm[31:0] = ( ap.sub ) ? ~b_ff[31:0] : b_ff[31:0];
136
137
138     assign {cout, aout[31:0]} = {1'b0, a_ff[31:0]} + {1'b0, bm[31:0]} + {32'b0, ap.sub};
139
172     assign sel_adder = (ap.add | ap.sub) & ~ap.slt;

185     assign out[31:0] = ({32{sel_logic}} & lout[31:0]) |
186                       ({32{sel_shift}} & sout[31:0]) |
187                       ({32{sel_adder}} & aout[31:0]) |
188                       ({32{ap.jal | pp_ff.pcall | pp_ff.pja | pp_ff.pret}} & {pcout[31:1],1'b0}) |
189                       ({32{ap.csr_write}} & ((ap.csr_imm) ? b_ff[31:0] : a_ff[31:0])) |
190                       ({31'b0, slt_one});
191

```

圖9. exu\_alu\_ctl內部的加法器

## ii. EX2和EX3階段

這些階段在算術-邏輯指令中執行的任務很少；但是，要將這些指令與其他需要三個週期來計算其操作的指令類型（例如載入、儲存、乘法等）同步，這些階段必不可少。請記住，在多週期設計中，每條指令可以有不同數量的階段，但在SweRV EH1等順序管線處理器中，所有指令必須遍歷相同數量的階段。

在本例中，加法的結果透過新的管線暫存器傳播，它位於模組dec\_decode\_ctl中。在EX3的3:1多路開關中，將選擇i0\_result\_e3（參見圖6）。該3:1多路開關也在實驗11的圖4和圖8中顯示。正如該實驗中所述，它從正確的管道中選擇結果，圖2範例中的結果由I0管道提供：（i0\_result\_e3\_final = i0\_result\_e3）。

```

2263     rvdffe #(32) i0e3resultff (.*, .en(i0_e3_data_en), .din(i0_result_e2[31:0]), .dout(i0_result_e3[31:0]));
2274     rvdffe #(32) i0e4resultff (.*, .en(i0_e4_data_en), .din(i0_result_e3_final[31:0]), .dout(i0_result_e4[31:0]));

```

**任務：**對於圖2中的範例，在模擬中驗證該多路開關是否從add指令的預期管道中選擇結果。

## iii. 提交階段

與EX2和EX3類似，該階段對獨立add指令執行少量操作（在實驗15中，我們將分析依賴於前一條指令的add指令必須在輔助ALU中重新計算加法的情況，圖6中未顯示）。在本例中，此階段可用的3:1多路開關選擇輸入i0\_result\_e4。該3:1多路開關也在實驗11的圖4和圖9中顯示。在圖2的範例中，選擇的值仍是I0管道提供的結果（i0\_result\_e4\_final = i0\_result\_e4）。

**任務：**對於圖2範例的add指令，在模擬中驗證該多路開關是否從正確的輸入來源（i0\_result\_e4）選擇結果。

## iv. 寫回階段



在最後一個階段，add指令的結果寫入暫存器檔案，如圖6所示。32位元結果（i0\_result\_wb\_raw[31:0]）在EX1階段計算並傳播到此階段。它在傳遞到暫存器檔案之前遍歷2:1多路開關（此多路開關的另一個輸入來自除法器，我們將在實驗14中進行分析）。暫存器位址（在圖7中，訊號waddr0以十六進位顯示，但也可以如前文所述以十進位顯示）和寫入啟用訊號透過控制管線暫存器提供。

**任務：**在Verilog程式碼中，分析訊號wen0和waddr0如何在解碼階段產生並傳播到寫回階段。

### 3. 練習

- 1) 對以下邏輯指令執行與本實驗中提供的分析類似的分析：and、or和xor。
- 2) （以下練習基於《電腦組織結構和設計》（RISC-V版本，Patterson & Hennessy [PaHe]））中的練習4.1。）  
請看下面的指令：and rd, rs1, rs2
  - a. SweRV EH1為該指令產生的控制訊號的值是多少？
  - b. 哪些資源（區塊）對該指令執行有用的功能？
  - c. 哪些資源（區塊）不為該指令產生輸出？哪些資源產生不使用的輸出？
- 3) 在Verilator模擬中以及直接在Verilog程式碼中分析RV32I基本整數指令集中提供的shift left/right指令：srl、sra和sll。
- 4) 在Verilator模擬中以及直接在Verilog程式碼中分析RV32I基本整數指令集中提供的小於則置位指令：slt和sltu。
- 5) 在Verilator模擬中以及直接在Verilog程式碼中分析RV32I基本整數指令集中提供的immediate指令：addi、andi、ori、xori、srli、srai、slli、slti和sltui。
- 6) （以下練習基於[PaHe]的練習4.6。）  
圖6不討論I型指令，例如addi或andi。
  - a. 需要哪些額外的邏輯區塊（如果有）來支援SweRV EH1中I型指令的執行？將所有必要的邏輯區塊新增到圖6並說明其用途。
  - b. 列出addi的控制單元產生的訊號的值。
- 7) （以下練習基於[PaHe]的練習4.4以及S. Harris和D. Harris所編教材《數位設計和電腦體系結構：RISC-V版本》[DDCARV]第7章的練習1。）  
製造矽晶片時，材料（如矽）中的缺陷和製造錯誤會導致電路有缺陷。一個非常常見的缺陷是一根訊號線「損壞」，邏輯始終為0。這通常稱為「stuck-at-0」（固定為0）故障。確定訊號i0\_ap（alu\_pkt\_t類型）中包含的每個控制位元發生「固定為0」故障的影響。