# 任務

**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。

**0x01de0e33 🡪 0000000 11101 11100 000 11100 0110011**

**funct7 = 0000000**

**rs2 = 11101 = x29 (t4)**

**rs1 = 11100 = x28 (t3)**

**funct3 = 000**

**rd = 11100 = x28 (t3)**

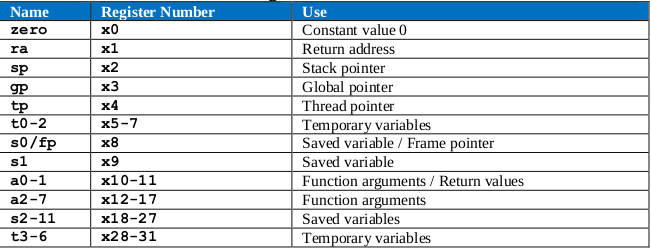
**op = 0110011**

來自DDCARV的附錄B：









**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。

解答請參見實驗12的主文件。

**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。

* 模組**dec\_decode\_ctl**中的控制單元
* 暫存器檔案：
  + 模組**dec**第525行中的實例化。
  + 模組**dec\_gpr\_ctl**中的實作。
* 解碼階段的3:1多工器：模組**exu**的第279行。
* 控制訊號的管線暫存器：分布在多個模組中。
* 暫存器aff和bff：模組**exu\_alu\_ctl**的第90行和第92行。
* EX1階段的I0 ALU：
  + 模組**exu**第401行中的實例化。
  + 模組**exu\_alu\_ctl**中的實作。
* 包含運算結果的管線暫存器（i0e2resultff、i0e3resultff、i0e4resultff和i0wbresultff）：模組**dec\_decode\_ctl**的第2260-2283行。
* EX3階段的3:1多工器：模組**dec\_decode\_ctl**的第2268行。
* EX4階段的3:1多工器：模組**dec\_decode\_ctl**的第2277行。
* 寫回階段的2:1多工器：模組**dec\_decode\_ctl**的第2286行。

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

* 暫存器識別碼從通路0中的32位元指令取得：訊號i0[31:0] = dec\_i0\_instr\_d[31:0]。

在R型指令中，它們位於以下欄位中：

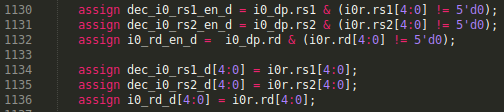


在模組**dec\_decode\_ctl**中：

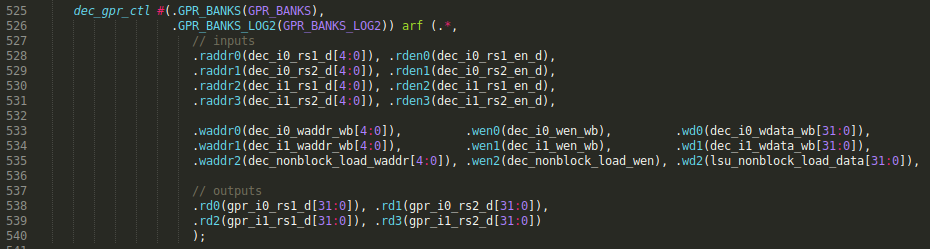


* 暫存器識別碼和讀取啟用訊號分配給dec\_i0\_rs1\_d/dec\_i0\_rs2\_d和dec\_i0\_rs1\_en\_d/dec\_i0\_rs2\_en\_d。

這些訊號從模組**dec**傳送到模組**dec\_decode\_ctl**。在模組**dec\_decode\_ctl**中：

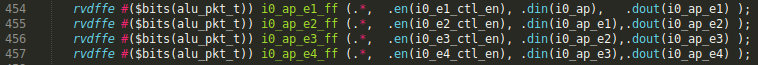


* 暫存器識別碼和讀取啟用訊號提供給暫存器檔案，該檔案在模組**dec**中實例化。在模組**dec**中：



**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階段。此外，還需尋找dd控制訊號周遊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。

訊號i0\_ap在模組**dec\_decode\_ctl**中取得。它提供給模組exu，並從中傳播到EX1、EX2、EX3和提交（EX4）階段。在模組**exu**中：



訊號dd在模組**dec\_decode\_ctl**中取得，並傳播到EX1、EX2、EX3、提交（EX4）和WB（EX5）階段。在模組**dec\_decode\_ctl**中：











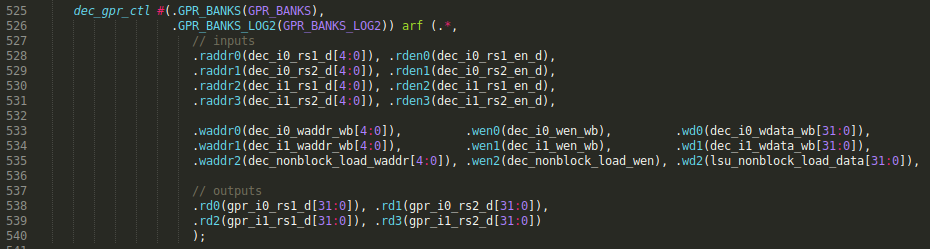
請注意，在進入下一個暫存器之前，每個暫存器的輸出都會稍作修改（並因此重新命名）。如果要查看詳細資訊，可以查看Verilog程式碼。

輸出運算元的暫存器識別碼在解碼階段分配：



訊號dd從解碼階段傳播到寫回階段（如上所示）：dd 🡪 e1d 🡪 e2d 🡪 e3d 🡪 e4d 🡪 wbd。隨後，目標暫存器在寫回階段提供給暫存器檔案：





**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。

不提供解答。

**任務：**在Verilog程式碼（模組**exu**）中尋找底部的3:1多工器（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。



這些3:1多工器接收3個輸入：

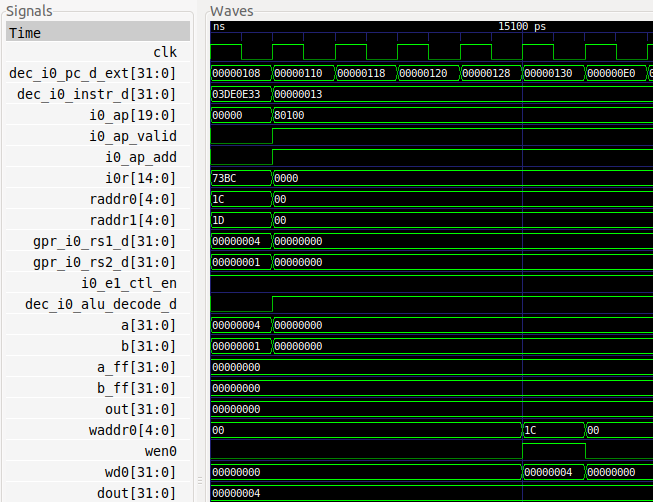
* 一個來自暫存器檔案（gpr\_i0\_rs2\_d）
* 一個來自32位元指令暫存器，它構成立即數（dec\_i0\_immed\_d）
* 一個來自旁路邏輯，我們將在實驗15中分析（i0\_rs2\_bypass\_data\_d）

**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。

解答請參見實驗12的主文件。

**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中範例所用*test\_2.tcl*檔案相同的檔案。

例如，mul t3, t3, t4的模擬（0x03de0e33）提供以下結果：

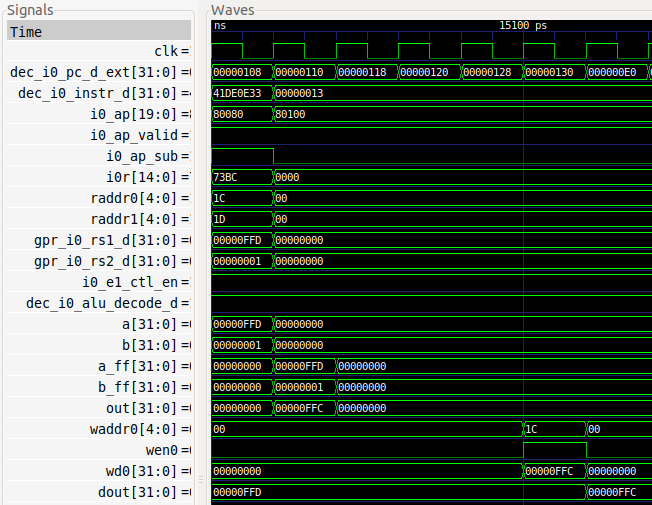


**任務：**將本部分中分析的新訊號包含在圖7的模擬中。

不提供解答。

**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*檔案將新訊號包含在模擬中。

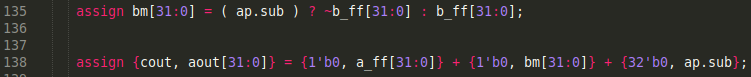
例如，sub t3, t3, t4的模擬（0x41de0e33）提供以下結果：



**任務：**分析模組**exu\_alu\_ctl**中實作的加法器/減法器的Verilog實作。圖8透過顯示與加法和減法運算直接相關的邏輯來提供一些協助。



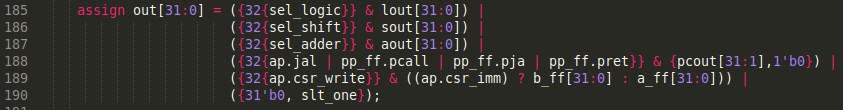
輸入運算元從解碼階段（a和b）傳播到執行階段（a\_ff和b\_ff）。



這是加法器/減法器。

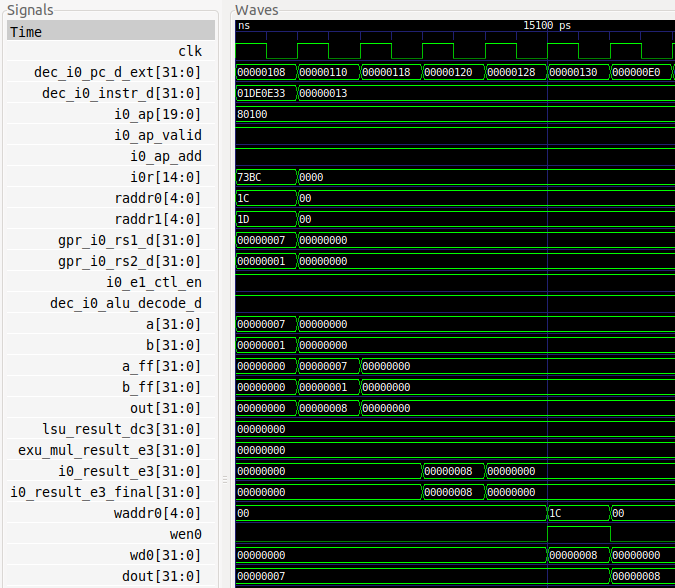
* 如果指令是加法，則aout = a\_ff + b\_ff
* 如果指令是減法，則先計算b\_ff的二進位補碼，然後計算a\_out。



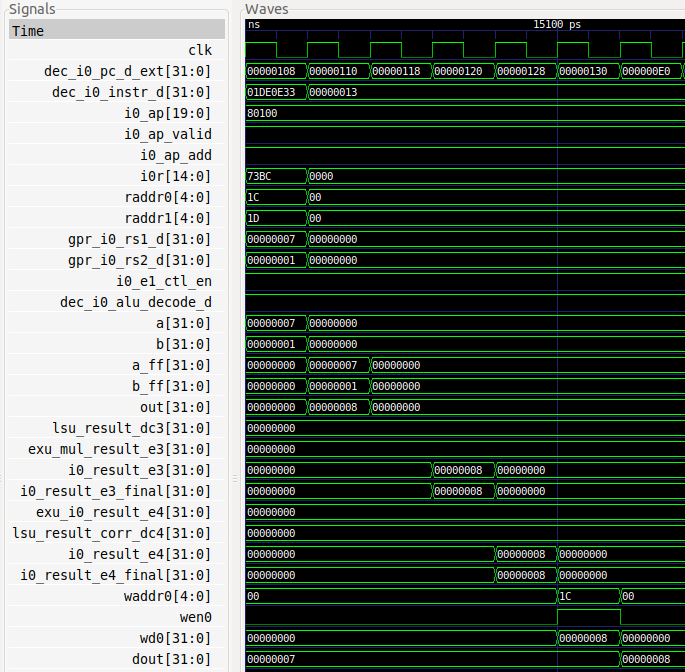


如果指令是加法或減法，則out = aout。

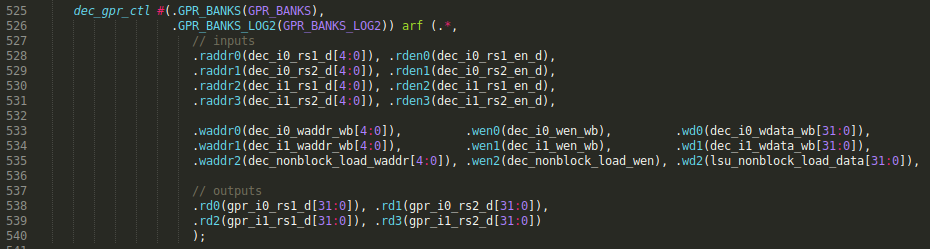
**任務：**對於圖2中的範例，在模擬中驗證該多工器是否從add指令的預期管線中選擇結果。



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



**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。









# 練習

1. 對邏輯指令（and、or和xor）執行與本實驗中提供的分析類似的分析。

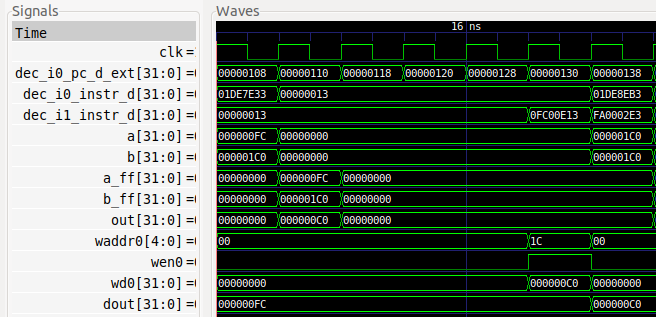
以下範例（在*[RVfpgaPath]/RVfpga/Labs/RVfpgaLabsSolutions/Programs\_Solutions/Lab12/AND\_Instruction*中提供）說明了無限迴圈中包含的and指令的執行情況。與add指令的範例中一樣，and指令（以紅色強調顯示）前後有幾條nop指令。迴圈末尾包含兩條指令，用於修改儲存在t3和t4中的值。

|  |
| --- |
| #define INSERT\_NOPS\_1 nop;  #define INSERT\_NOPS\_2 nop; INSERT\_NOPS\_1  #define INSERT\_NOPS\_3 nop; INSERT\_NOPS\_2  #define INSERT\_NOPS\_4 nop; INSERT\_NOPS\_3  #define INSERT\_NOPS\_5 nop; INSERT\_NOPS\_4  #define INSERT\_NOPS\_6 nop; INSERT\_NOPS\_5  #define INSERT\_NOPS\_7 nop; INSERT\_NOPS\_6  #define INSERT\_NOPS\_8 nop; INSERT\_NOPS\_7  #define INSERT\_NOPS\_9 nop; INSERT\_NOPS\_8  #define INSERT\_NOPS\_10 nop; INSERT\_NOPS\_9  .globl main  main:  li t3, 0xFC # t3 = 0xFC  li t4, 0x7 # t4 = 0x7    REPEAT:  INSERT\_NOPS\_10  **and t3, t3, t4** # t3 = t3 & t4  INSERT\_NOPS\_10  li t3, 0xFC # t3 = 0xFC  add t4, t4, t4  beq zero, zero, REPEAT # Repeat the loop  .end |

如果在PlatformIO中開啟、編譯項目，然後開啟反組譯檔案（位於*[RVfpgaPath]/RVfpga/Labs/RVfpgaLabsSolutions/Programs\_Solutions/Lab12/AND\_Instruction/.pio/build/swervolf\_nexys/firmware.dis*中），可以看到and指令位於位址0x00000108處，還可以看到指令的機器程式碼（0x01de7e33）：

**0x00000108: 01de7e33 and t3,t3,t4**

接下來，我們在Verilator中模擬程式，然後在GTKWave上開啟模擬器產生的追蹤檔案。移至迴圈的任何一次迭代（第一次除外）。



分析波形（以紅色強調顯示的值對應於and指令）。在本實驗中，我們將跳過擷取和對齊階段，這兩個階段將在後面的實驗中說明。

* **解碼**階段：訊號dec\_i0\_pc\_d\_ext包含指令的位址（在教材中，該訊號通常稱為程式計數器），and的位址為0x00000108，訊號dec\_i0\_instr\_d包含32位元機器指令0x01DE7E33（在教材中，該訊號通常稱為指令暫存器）。

在RISC-V中，and指令的操作碼如下（參見[Harris&Harris]的附錄B）：

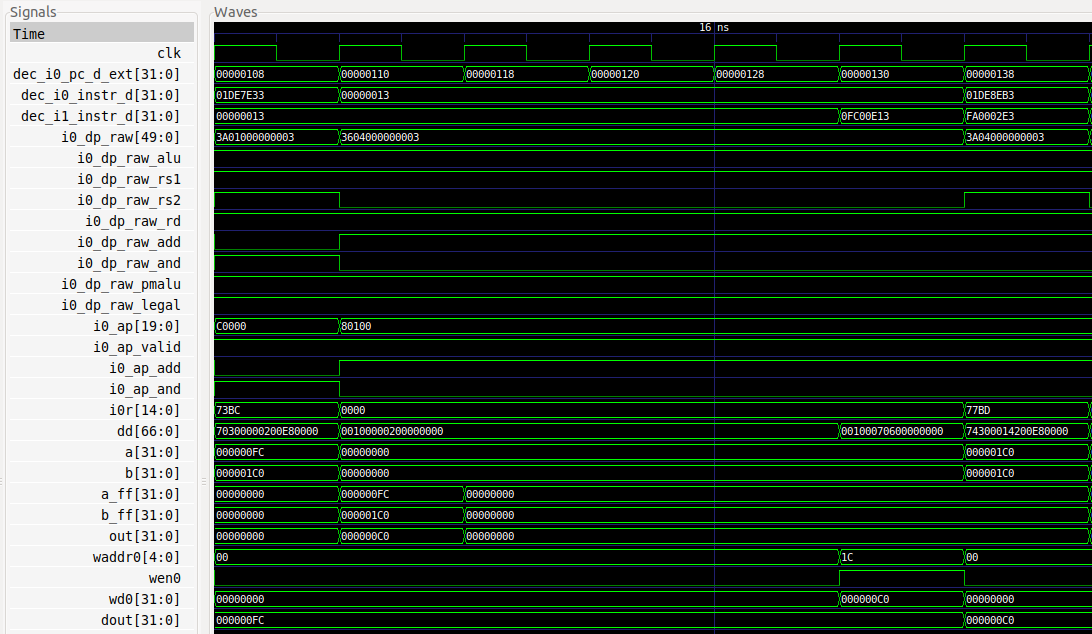
0000000 | rs2 | rs1 | 111 | rd | 0110011

因此可以輕鬆驗證0x01DEFE33是否對應於：and t3, t3, t4（請記住，t3=x28且t4=x29）。

在此階段**將產生管線控制訊號**（我們將在下一部分中詳細介紹）。此外，在此階段還將**讀取暫存器檔案**。訊號a和b包含ALU的輸入，本例中與從暫存器檔案讀取的值一致（對於後續實驗中將分析的其他範例，情況並非如此）。

* **EX1**階段：在下一週期中，將**執行**and指令。訊號a\_ff和b\_ff包含ALU的輸入（分別為0xFC和0x1C0），而out包含加法的結果（0xC0）。
* **EX5**階段（也稱為**寫回**）：最後，在4個週期後，加法結果透過訊號wd0=0xC0**寫回**到暫存器檔案中，其中包含要寫入的資料。鑒於wen0=1（寫入啟用），邏輯與運算結果在相應週期結束時寫入暫存器x28（暫存器索引，waddr0=0x1C）。可以發現，在接下來的週期（圖中最後一個週期）中，暫存器x28包含新值（dout=0xC0）。

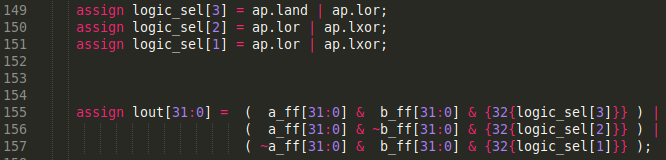
接下來，我們將控制訊號新增到之前的模擬中：



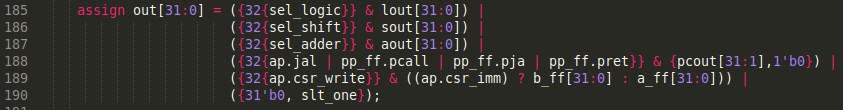
可以看到，在第一個週期中，and指令的控制位元為1。

以下Verilog片段顯示了SweRV EH1的邏輯單元。









當and控制位為1時，選擇邏輯與運算的結果：

logic\_sel[3]=1且logic\_sel[2]=logic\_sel[1]=0 🡪 lout = a\_ff & b\_ff

1. （*以下練習基於《電腦組織結構和設計》（RISC-V版本，作者Patterson & Hennessy（[HePa]））中的練習4.1。*）

請看下面的指令：and rd, rs1, rs2

* 1. SweRV EH1為該指令產生的控制訊號的值是多少？
  2. 哪些資源（區塊）對該指令執行有用的功能？
  3. 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？

不提供解答。

1. 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。

#define INSERT\_NOPS\_0

#define INSERT\_NOPS\_1 nop; INSERT\_NOPS\_0

#define INSERT\_NOPS\_2 nop; INSERT\_NOPS\_1

#define INSERT\_NOPS\_3 nop; INSERT\_NOPS\_2

#define INSERT\_NOPS\_4 nop; INSERT\_NOPS\_3

#define INSERT\_NOPS\_5 nop; INSERT\_NOPS\_4

#define INSERT\_NOPS\_6 nop; INSERT\_NOPS\_5

#define INSERT\_NOPS\_7 nop; INSERT\_NOPS\_6

#define INSERT\_NOPS\_8 nop; INSERT\_NOPS\_7

#define INSERT\_NOPS\_9 nop; INSERT\_NOPS\_8

#define INSERT\_NOPS\_10 nop; INSERT\_NOPS\_9

.globl main

main:

li t3, 0xEEEEEEEE

li t4, 0x1

REPEAT:

srl t0, t3, t4

INSERT\_NOPS\_7

sra t1, t3, t4

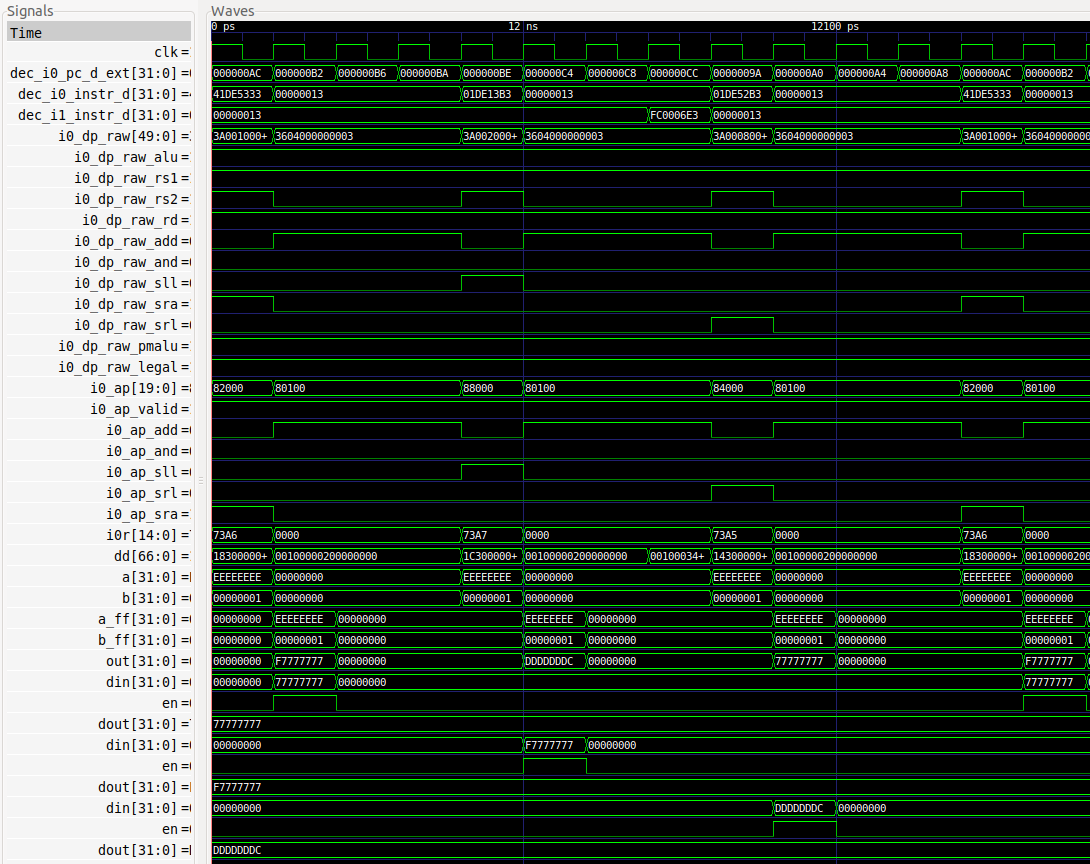
INSERT\_NOPS\_7

sll t2, t3, t4

INSERT\_NOPS\_6

beq zero, zero, REPEAT # Repeat the loop

.end

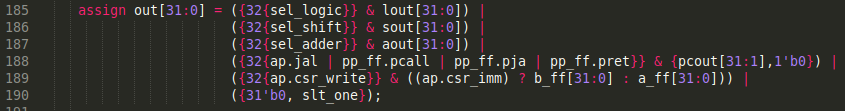


以下Verilog片段顯示了SweRV EH1的移位單元。









1. 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。

#define INSERT\_NOPS\_0

#define INSERT\_NOPS\_1 nop; INSERT\_NOPS\_0

#define INSERT\_NOPS\_2 nop; INSERT\_NOPS\_1

#define INSERT\_NOPS\_3 nop; INSERT\_NOPS\_2

#define INSERT\_NOPS\_4 nop; INSERT\_NOPS\_3

#define INSERT\_NOPS\_5 nop; INSERT\_NOPS\_4

#define INSERT\_NOPS\_6 nop; INSERT\_NOPS\_5

#define INSERT\_NOPS\_7 nop; INSERT\_NOPS\_6

#define INSERT\_NOPS\_8 nop; INSERT\_NOPS\_7

#define INSERT\_NOPS\_9 nop; INSERT\_NOPS\_8

#define INSERT\_NOPS\_10 nop; INSERT\_NOPS\_9

.globl main

main:

li t3, 0x80000007

li t4, 0x6

REPEAT:

slt t0, t3, t4

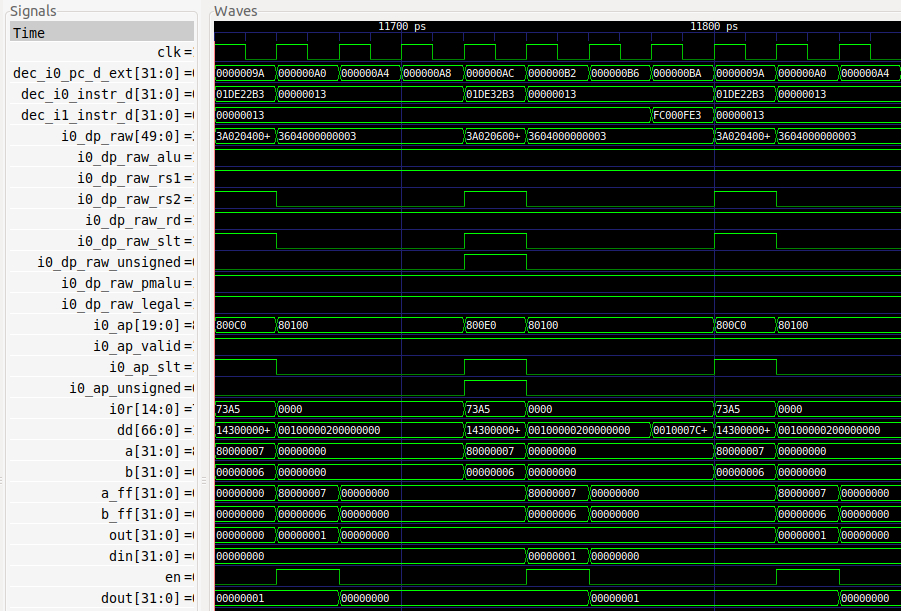
INSERT\_NOPS\_7

sltu t0, t3, t4

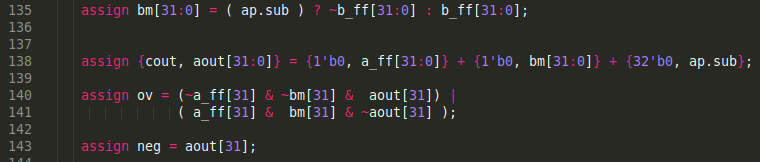
INSERT\_NOPS\_6

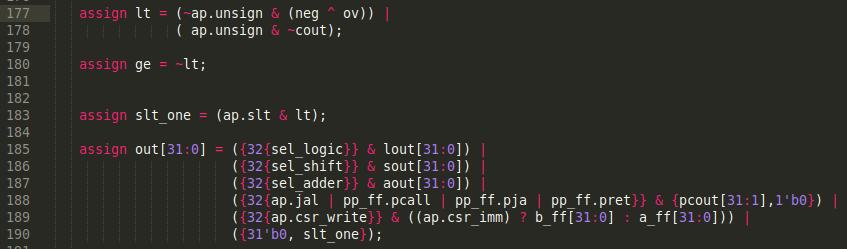
beq zero, zero, REPEAT # Repeat the loop

.end



以下Verilog片段顯示了在SweRV EH1中執行這些運算的邏輯。





1. 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。

#define INSERT\_NOPS\_0

#define INSERT\_NOPS\_1 nop; INSERT\_NOPS\_0

#define INSERT\_NOPS\_2 nop; INSERT\_NOPS\_1

#define INSERT\_NOPS\_3 nop; INSERT\_NOPS\_2

#define INSERT\_NOPS\_4 nop; INSERT\_NOPS\_3

#define INSERT\_NOPS\_5 nop; INSERT\_NOPS\_4

#define INSERT\_NOPS\_6 nop; INSERT\_NOPS\_5

#define INSERT\_NOPS\_7 nop; INSERT\_NOPS\_6

#define INSERT\_NOPS\_8 nop; INSERT\_NOPS\_7

#define INSERT\_NOPS\_9 nop; INSERT\_NOPS\_8

#define INSERT\_NOPS\_10 nop; INSERT\_NOPS\_9

.globl main

main:

li t3, 0x4 # t3 = 4

INSERT\_NOPS\_1

REPEAT:

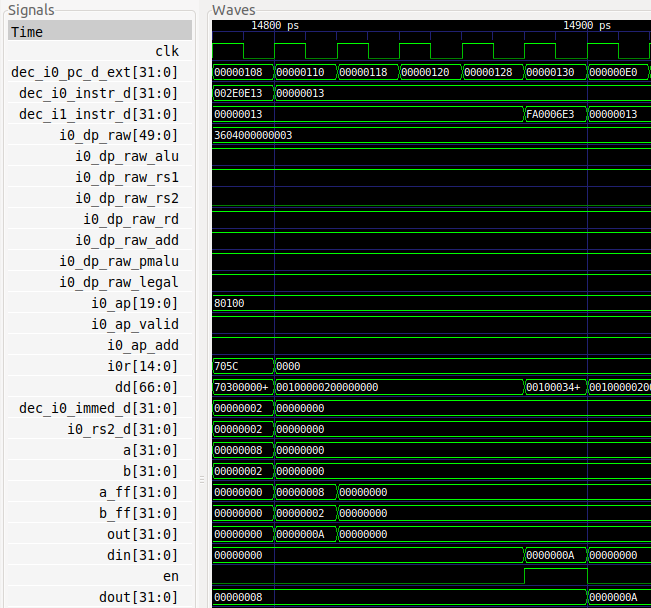
INSERT\_NOPS\_10

addi t3, t3, 2 # t3 = t3 + t4

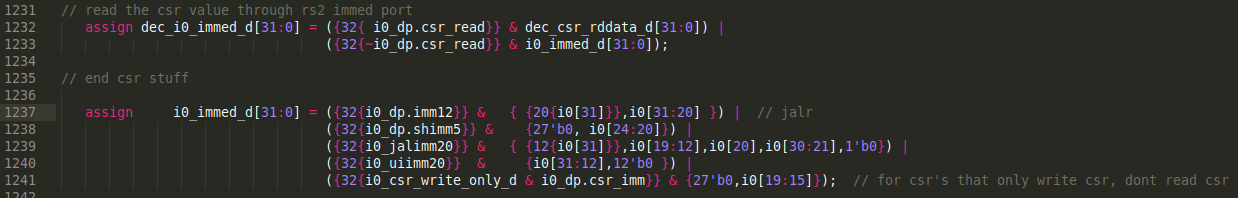
INSERT\_NOPS\_10

beq zero, zero, REPEAT # Repeat the loop

.end



在模組**dec\_decode\_ctl**中，計算32位立即數。



在模組**exu**中，選擇正確的*rs2*源。本例中我們使用*dec\_i0\_immed\_d*。



在模組**dec\_gpr\_ctl**中，使能訊號*rden1*確定是否訪問暫存器檔案來獲取第二個操作數。如果指令使用立即數操作數：i0\_dp.rs2=0 🡪 rden1=0 🡪 rd1[31:0]=0x00000000 🡪 gpr\_i0\_rs2\_d[31:0]=0x00000000。



1. （*以下練習基於[HePa]的練習4.4以及S. Harris和D. Harris所編教材《數字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傳送「固定為0」故障的影響。

結構類型在檔案swerv\_types.sv中定義：

typedef struct packed {

logic valid;

logic land;

logic lor;

logic lxor;

logic sll;

logic srl;

logic sra;

logic beq;

logic bne;

logic blt;

logic bge;

logic add;

logic sub;

logic slt;

logic unsign;

logic jal;

logic predict\_t;

logic predict\_nt;

logic csr\_write;

logic csr\_imm;

} alu\_pkt\_t;

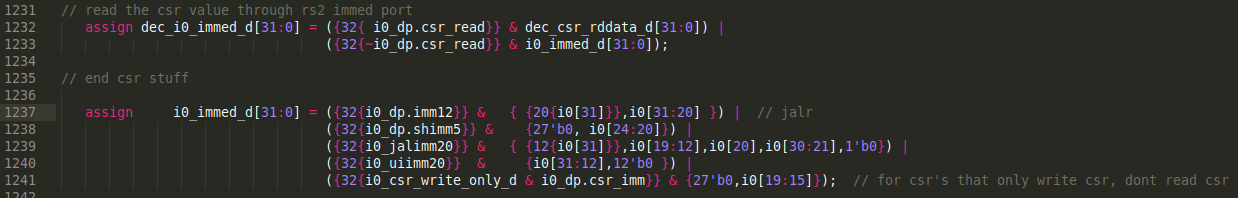
* 訊號valid固定為0：無法執行任何A-L指令，因為任何A-L指令都將被視為無效。
* 訊號land、lor、lxor、sll、srl、sra、beq、bne、blt、bge、add、sub、slt和jal固定為0：對於上述每一位元，都無法執行相應的A-L指令；例如，如果land固定為0，將無法執行and指令。
* 訊號unsign固定為0：無法向處理器傳達運算必須為無符號運算的訊息。
* 訊號predict\_t和predict\_nt：無法向處理器傳達預測採用或不採用分支的訊息。
* 訊號csr\_write和csr\_imm：無法在CSR暫存器中寫入或使用立即數進行運算。

1. （*以下練習基於[HePa]的練習4.6。*）

圖5不討論I型指令，如addi或andi。

* 1. 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5並說明其用途。
  2. 列出addi的控制單元產生的訊號的值。

解碼階段兩個3-1多工器的輸入之一來自訊號dec\_i0\_immed\_d[31:0]中的立即數。立即數是一個32位元訊號，根據執行的I型指令進行不同的計算。它是組成指令的一個子集（32位元），相應位元的選擇和符號延伸過程如下：



addi的控制訊號的值位於練習5的模擬中。