



IMAGINATION大學計劃

RVfpga實驗11

SweRV EH1組態、結構和效能監視

1. 簡介

在前10個RVfpga實驗（實驗1-10）中，我們介紹了RISC-V架構以及如何使用各種周邊設備與SweRV EH1核新通訊。在接下來的10個實驗（實驗11-20）中，我們將深入到微架構等級並分析SweRV EH1處理器的內部運作方式以及快取/記憶體階層的工作方式。

SIGASI STUDIO：在這些實驗中，我們將處理擴展Verilog專案：SweRV EH1 Core RTL。分析各種模組和訊號的一種方法是使用Sublime Text等典型編輯器（<https://www.sublimetext.com/>），這類編輯器提供了強大的功能，支援在專案中導覽、檢查檔案、尋找字串等。不過，還有更適合的特定替代方案，例如我們強烈建議的**Sigasi Studio**（<https://www.sigasi.com/>）。補充文件**SweRVref.docx**展示了如何安裝和使用Sigasi Studio（SweRVref文件的第1部分）。

正如RVfpga入門指南（Getting Started Guide，GSG）中所述，SweRV EH1是32位元2路超標量9級管線順序處理器。圖1顯示了SweRV EH1微架構的高階檢視。SweRV EH1支援RISC-V的整數指令（I）、壓縮指令（C）和整數乘法指令（M）擴展。它擁有極高的每MHz效能（4.9 CM/MHz），這得益於其包含多種微架構技術，從管線和指令快取等最基本、最常見的技術到超標量執行、非阻塞載入和除法、兩個可在出現資料冒險的必要情況下重複算術邏輯指令的輔助ALU（有關詳細資訊，請參見實驗15）、未對齊的載入和儲存、指令和資料的暫存記憶體以及進階分支預測等其他更具體、更先進的技術。上述所有技術都將在這些實驗中進行廣泛分析。

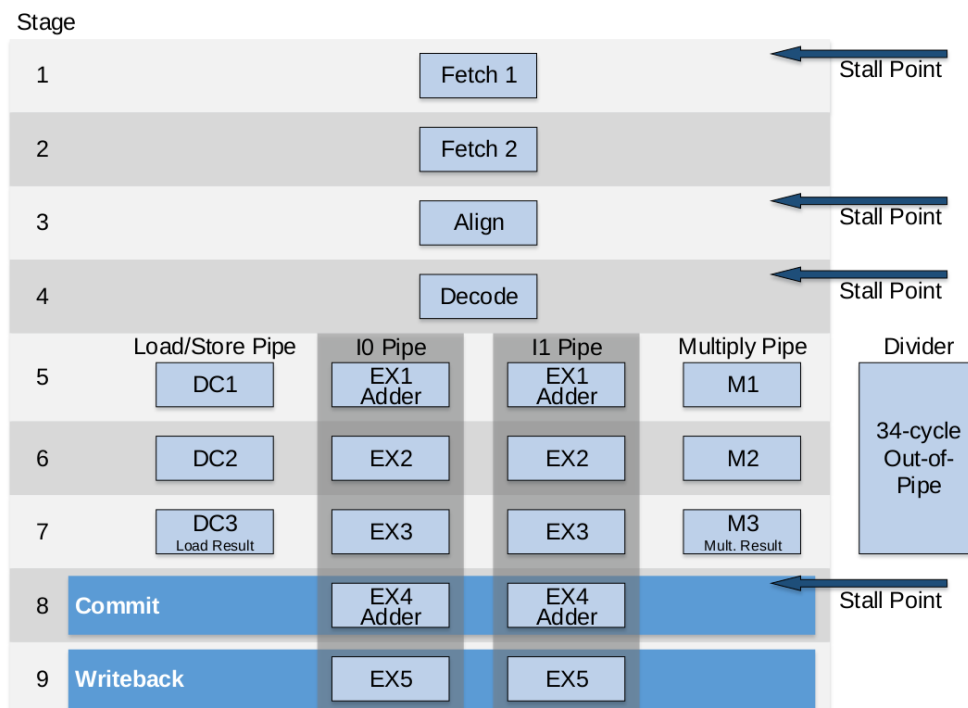


圖1. SweRV EH1核心微架構

（圖來自https://github.com/chipsalliance/Cores-SweRV/blob/master/docs/RISC-V_SweRV_EH1_PRM.pdf）

附註：在開始這組實驗之前，我們建議您仔細閱讀教材《數位設計和電腦體系結構：RISC-V 版本》的第7章和第8章，作者為S. Harris和D. Harris（Morgan Kaufmann © 2021）。這些實驗的一些內容受這本書的啟發。我們將這本書稱為DDCARV。

大多數實驗分為兩部分：分別是基礎部分和進階部分。此外，鑒於實際處理器（例如SweRV EH1）某些部分的複雜度較高，一些詳細資訊已移至指定實驗的附錄中。這樣，使用者便可選擇只完成基礎部分、完成基礎部分和進階部分，甚至深入研究附錄以瞭解處理器更複雜的部分。

實驗11-20從概念的理論說明開始，隨後使用圖和範例程式的Verilator模擬來說明概念。這些範例程式僅用於說明概念。我們還提供相關練習以加深對所描述概念的理解和體驗。

根據課程的目標和深度，學員有可能只完成實驗的一小部分。以下實驗涵蓋管線、記憶體構成和進階微架構/記憶體階層的概念：

- **管線：**實驗11、12、14、15和實驗16的第一部分（分支指令）
- **記憶體：**實驗11、13和19
- **進階微架構和記憶體層級：**實驗17、18、20和實驗16的第二部分（分支預測器）

在本實驗（實驗11）中，我們開始分析SweRV EH1處理器。具體來說：

- **第2部分**介紹Verilog RTL結構和每個管線階段的詳細資訊。
- **第3部分**展示如何使用效能計數器來分析處理器效能。

補充文件（**SweRVref.docx**）介紹以下內容：

- **第1部分：**Sigasi Studio的應用。
- **第2部分：**SweRV EH1處理器的組態。
- **第3部分：**模組的RVfpga系統階層及其最相關的訊號
- **第4部分：**用於對控制位元進行分組的結構/類型
- **第5部分：**RISC-V壓縮指令
- **第6部分：**實際基準

在此初始方法後，我們會在實驗12-20中將此分析擴展到不同處理器單元。具體來說：

- **實驗12**重點關注**算術邏輯**指令，實驗期間將更深入地瞭解解碼、EX1/EX2/EX3和寫回階段。
- **實驗13**介紹**記憶體存取指令**（載入和儲存），實驗期間重點關注DC1/DC2/DC3階段。
- **實驗14**探討了**結構冒險**，實驗期間重點關注3週期管線乘法指令和與非阻塞載入相關的特定情況。該實驗還將分析附錄中的34週期非管線除法指令。
- **實驗15**分析**資料冒險**，實驗期間將介紹處理器的旁路路徑。

- **實驗16**介紹控制冒險、分支指令和分支預測器，將重點關注SweRV EH1處理器的擷取1和擷取2兩個階段。
- 在之前的實驗中，大多數情況下只使用1路處理器，但**實驗17**介紹2路超標量處理器，例如SweRV EH1。
- **實驗18**是一個實踐實驗，期間將向SweRV EH1核心新增新指令和硬體計數器。
- **實驗19**和**20**重點關注處理器中的各個低延遲記憶體：指令快取（I\$）以及緊密耦合的指令和資料記憶體（ICCM和DCCM）。

2. SweRV EH1微架構的初始近似

DDCARV中介紹的處理器包含5個管線階段，分別稱為擷取、解碼、執行、記憶體存取和寫回階段。相比之下，SweRV EH1管線分為9級（圖1）：擷取1、擷取2、對齊、解碼、EX1/DC1/M1、EX2/DC2/M2、EX3/DC3/M3、提交和寫回級。比較兩個處理器時，有些階段是等效的，例如解碼和寫回階段。但SweRV EH1新增了平行路徑（載入/儲存、整數與乘法管道），將一些階段分成多個階段（擷取為2個階段，執行為3個階段），並新增了一些階段（提交和對齊階段）。

本節的其餘部分介紹Verilog RTL結構和每個管線階段的詳細資訊。A部分介紹SweRV EH1 Verilog模組的層級。B部分和C部分逐階段討論SweRV EH1的微架構。最後，D部分提供B部分和C部分中提出的理論說明的實際範例。

SWERV EH1處理器的組態： SweRV EH1處理器的許多結構和功能都可以配定或啟用/停用。補充文件SweRVref.docx在第2部分中解釋了這些不同的選項，實驗12-20中將經常使用這些選項。

A. SweRV EH1 Verilog模組的階層

圖2顯示了構成SweRV EH1處理器的主要Verilog模組（某些模組未包含在圖中）的階層。該圖擴展了GSG的圖29，後者顯示了構成RVfpga系統的Verilog模組的階層。這些模組位於同名檔案中，位置如下：`[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex`目錄。

mem模組對構成SweRV EH1處理器記憶體階層的結構進行實例化：ICCM、DCCM和I\$。
swerv模組為整體CPU；其對構成SweRV EH1處理器的模組進行實例化：指令擷取單元（ifu）、解碼單元（dec）、執行單元（exu）、載入/儲存單元（lsu）...

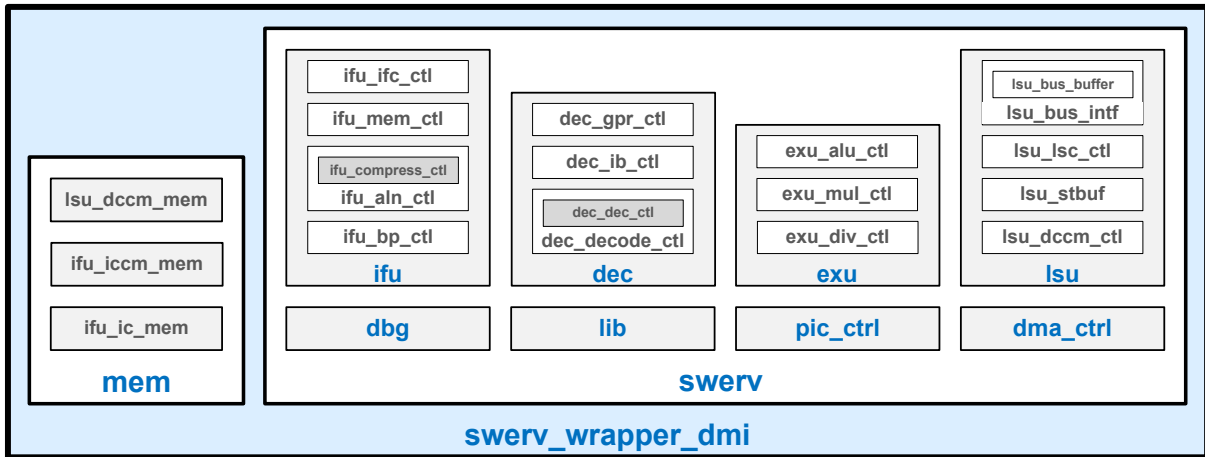


圖2. SweRV EH1主要模組

SweRV EH1核心的主要訊號：補充文件SweRVref.docx在第3部分中提供SweRV EH1處理器各模組的主要輸入/輸出訊號。進行實驗11-20時，可將其作為參考。

B. 擷取 (FC1和FC2) 和對齊階段

在本部分中，我們將分析管線的前三個階段：**SweRV EH1**管線的兩個擷取階段（**FC1**和**FC2**）和對齊階段。圖3提供了這些階段的高度簡化檢視。

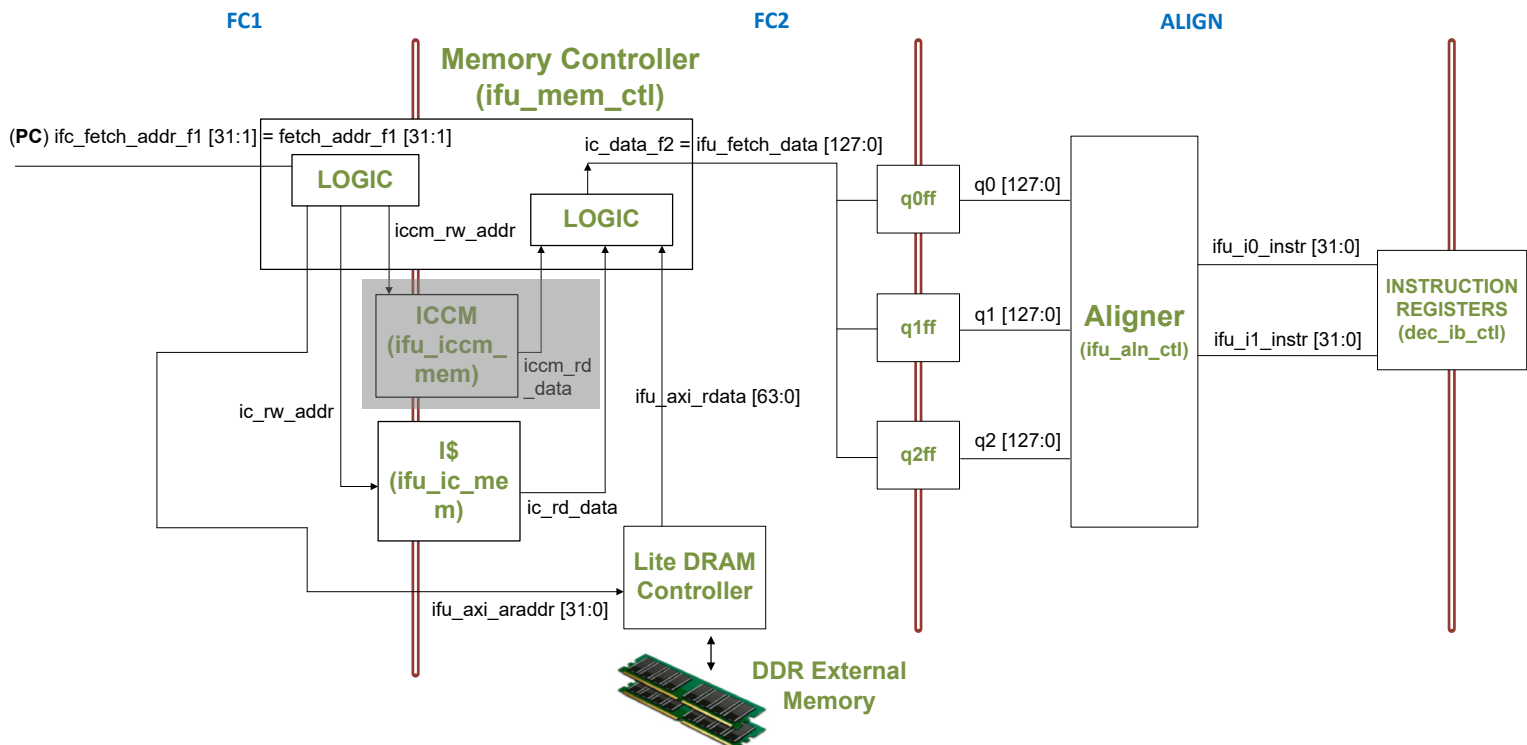


圖3. FC1、FC2和對齊階段的簡化檢視。請注意，ICCM帶有陰影，表示它在RVfpga系統中被停用。

i. 擷取階段 (FC1和FC2)

在每個週期中，擷取階段負責從指令記憶體中讀取指令。在我們的組態中，指令記憶體由ICCM（在模組`ifu_iccm_mem`中實作）、指令快取（`IS`，在模組`ifu_ic_mem`中實作）和DDR外部記憶體組成。`IS`和ICCM都由一個統一的記憶體控制器（`ifu_mem_ctl`）控制，而外部記憶體由Lite DRAM控制器控制。在預設RVfpga系統中，ICCM被停用，但可以按照實驗20中的說明輕鬆將其包含在內。

如圖3所示，指令位址（稱為擷取位址`ifc_fetch_addr_f1`）在第一個擷取階段（**FC1**）計算，實驗16中將對其進行進一步討論。該位址提供給指令記憶體控制器（在模組`ifu_mem_ctl`中實作）：`fetch_addr_f1 = ifu_fetch_addr_f1`。

訊號通常具有與其所屬單元相對應的前綴。例如，「`ifu`」代表擷取單元。訊號將附加與其關聯的階段。例如，「`f1`」表示**FC1**階段。

指令在第二個擷取階段（**FC2**）從主記憶體（即DDR外部記憶體）或ICCM讀取。如果指令位址處於主記憶體位址範圍內，則`IS`提供指令。在`IS`未命中時，管線必須暫停，直到外部記憶體透過AXI匯流排提供指令，這需要幾個週期。如果指令位址處於ICCM位址範圍內，則ICCM透過`ifu_ic_mem`模組內部實作的多路開關提供低延遲指令。

RVfpga系統的指令記憶體設定如下（此組態可以修改，我們將在以後的實驗中展示）：

- 16 KiB指令快取
- 512 KiB ICCM（停用）：位址範圍：0xEE000000 – 0xEE07FFFF
- 128 MiB外部記憶體：位址範圍：0x00000000 – 0x07FFFFFF

如果程式沒有暫停（即沒有控制、資料或結構冒險，沒有`IS`未命中等），則每兩個週期讀取4條32位元指令（總共128位元）：請參見訊號`ifu_fetch_data[127:0]`。這足以使2路超標量管線以每週期2條指令的最大吞吐量工作。三個緩衝區（`q0ff`、`q1ff`和`q2ff`）最多可以儲存三個此類128位元指令束。

ii. 對齊階段

對齊階段在兩個擷取階段之後（參見圖3），在模組`ifu_aln_ctl`中實作。對齊階段負責執行兩個主要任務：

- 每個週期向解碼階段提供兩條32位元指令：對齊階段每個週期從指令記憶體提供的128位元指令束中擷取兩條指令，它們暫時儲存在緩衝區`q0ff`、`q1ff`和`q2ff`中。這兩條指令透過訊號`ifu_i0_instr[31:0]`（通路0）和`ifu_i1_instr[31:0]`（通路1）指定給SweRV EH1中的每一路（共兩路），然後儲存在模組`dec_ib_ctl`中實作的兩個指令暫存器（Instruction Register，IR）中。

- **解壓縮指令：**RISC-V的壓縮指令擴展（RVC）透過減少控制、立即數和暫存器欄位的大小並利用冗餘或隱含暫存器將常見整數和浮點指令的大小減小到16位元。指令大小減小後可降低成本、功耗和所需的儲存空間（參見DDCARV的第6.6.5節）。對齊階段將在必要時解壓縮這些16位元指令，然後將其傳送到僅解碼32位元指令的解碼階段。此過程由if_compress_ctl模組執行，該模組在對齊器（模組ifu_aln_ctl）內部實例化。

壓縮指令：補充文件SweRVref.docx在第5部分中解釋了SweRV EH1中壓縮指令的執行，並提出了一些新任務。

C. 解碼、執行、提交和寫回階段

在本部分中，我們將分析SweRV EH1管線的解碼、執行、提交和寫回階段。圖4提供了這些階段的簡化檢視，我們將在以後的實驗中進行擴展。

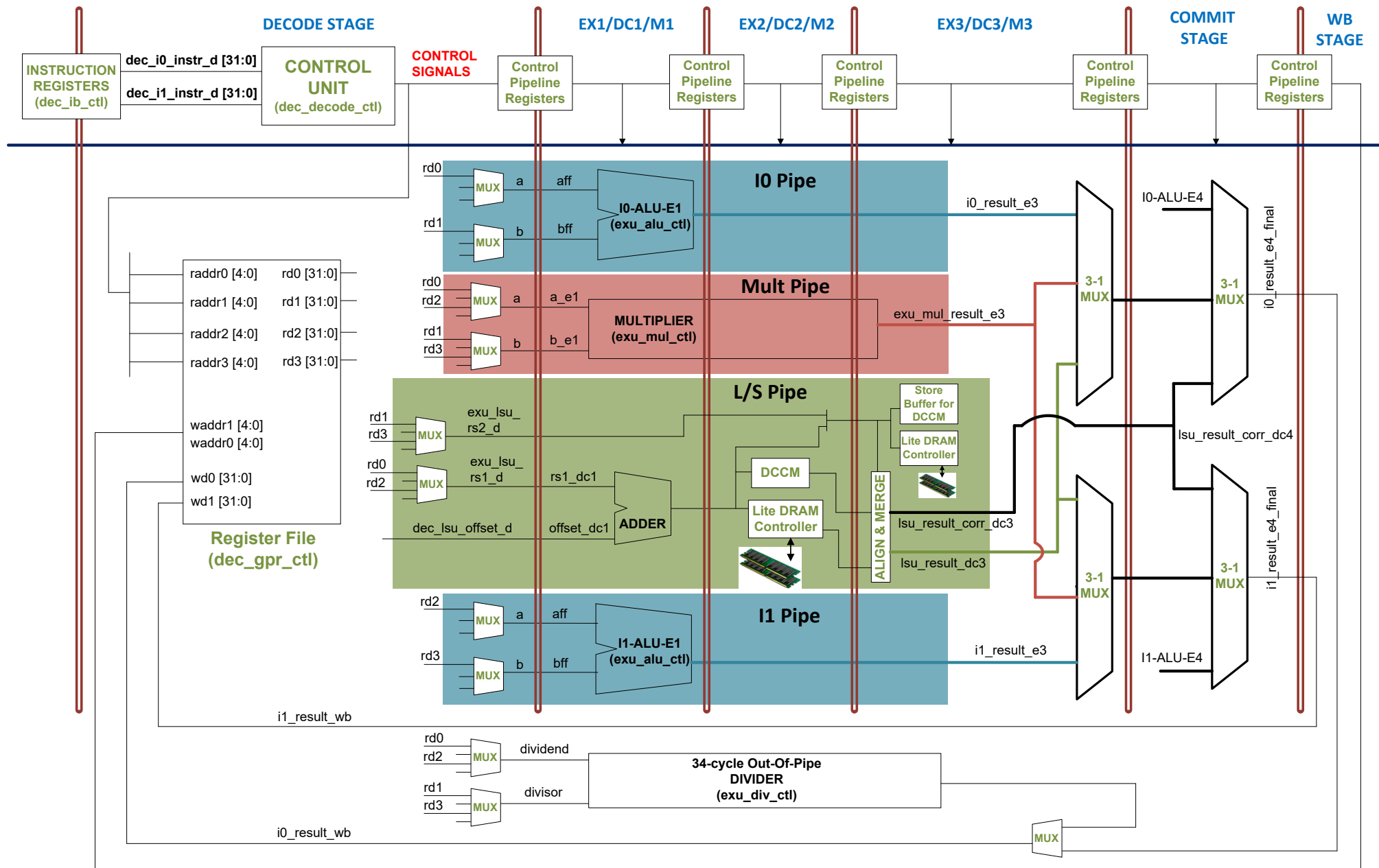


圖4. 解碼、執行、提交和寫回階段的簡化檢視

i. 解碼階段

此階段的Verilog模組位於資料夾

[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/dec中。在每個週期中，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：控制訊號分為幾種類型，具體在 [RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/include/swerv_types.sv 中定義。每個結構/類型都與指定的單元相關：ALU (alu_pkt_t)、乘法單元 (mul_pkt_t)、除法單元 (div_pkt_t) 和暫存器 (reg_pkt_t) 等。

用於控制位元的結構：補充文件SweRVref.docx在第4部分中擴展了SweRV EH1處理器中用於對控制訊號進行分組的主要結構/類型的說明，並提出了一些新任務。在後面的實驗中，我們將重點關注與所討論單元相關的類型。

模組dec_decode_ctl中實現的控制單元接收在前幾個階段擷取、解壓縮、對齊及指定到每一路的兩條32位元指令（通路0為訊號dec_i0_instr_d[31:0]，通路1為訊號dec_i1_instr_d[31:0]）並對其進行解碼，為每條指令產生控制訊號。圖5顯示了控制單元（模組dec_decode_ctl）的高階檢視，控制單元分兩個階段產生控制訊號：前兩個模組（i0_dec和i1_dec）使用指令（i0和i1）產生整體控制訊號（i0_dp和i1_dp，其類型均為dec_pkt_t），然後第二個單元（**解碼**）使用這些訊號為每個管線路徑產生控制訊號，也稱為「管道」（i0_ap、i1_ap、lsu_p和mul_p等）。

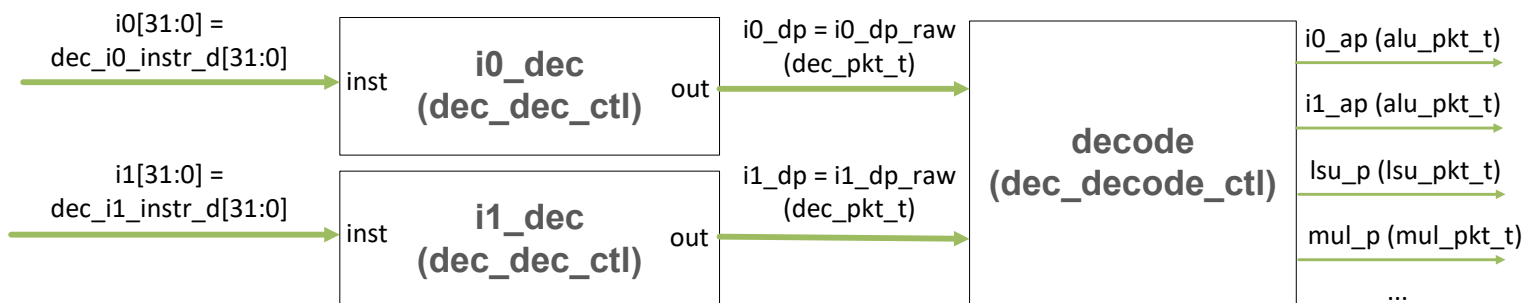


圖5. 控制單元

控制單元使用管線暫存器（在圖4中標記為**控制管線暫存器**）將這些控制訊號傳播到後續管線階段，這些管線暫存器處於各管線階段之間。

- 將指令分發到適當的管道並提供運算元：如圖4所示，SweRV EH1包括兩個整數管道（I0和I1）、一個乘法管道和一個載入/儲存管道（L/S），此外，還包括一個位於管線外部的34週期除法器。每條指令解碼完成後，處理器將其傳送到四個單獨的管線之一：
 - 算術邏輯和分支指令在I0/I1管道中執行。

- 載入和儲存在L/S管道中執行。
- 乘法指令透過乘法管道執行。
- 將執行的指令劃分到除法管道。

鑒於每個週期最多對兩條指令進行解碼（一條在通路0中，另一條在通路1中），兩條均在可能時調度執行。例如，一些可能的組合包括：

- 兩條獨立的算術邏輯指令傳送到I0和I1管道。
- 一條算術邏輯指令和一條乘法（mul）指令分別傳送到I0（或I1）和乘法管道。
- 記憶體存取（載入或儲存）指令在L/S管道中執行，乘法指令在乘法管道中執行。

遺憾的是，當一條或兩條指令必須暫停時，存在某些情況（例如冒險，我們將在實驗14-16中分析這些情況）。這些情況也會在解碼階段確定。例如：

- 如果兩條mul指令在同一個週期內解碼，則透過將第二條mul指令延遲一個週期來消除結構冒險（實驗14中將對此進行詳細分析）。
- 如果兩個相關的算術邏輯（Arithmetic-Logic, A-L）指令在同一週期內進行解碼，則透過將第二條A-L指令延遲一個週期來消除寫後讀資料冒險（實驗15中將對此進行詳細分析）。

除了調度指令外，還必須為管道提供相應的運算元。為此，幾個3:1和4:1多路開關（參見圖4）會在可能的運算元中進行選擇，並使用管線暫存器將其傳播到下幾個階段。這些多路開關在模組exu的第279-328行中實作（即使多路開關位於exu模組內部，它們也在解碼階段工作）。其輸入運算元的來源可能如下：

- **旁路邏輯**：大多數資料相關性在解碼階段透過旁路來避免，我們將在實驗15中進行分析。來自旁路邏輯的輸入未在圖4的3:1和4:1多路開關中進行標記，為簡單起見，僅顯示空白線。
- **立即**：一些RISC-V指令使用立即定址模式，其中運算元直接從指令位元提供。來自立即定址的輸入未在圖4的3:1和4:1多路開關中顯示，— 僅顯示空白輸入線）。
- **暫存器檔案**：SweRV EH1處理器中可用的暫存器檔案（圖6）有4個讀取連接埠和3個寫入連接埠（請注意，第三個寫入連接埠在圖4中包含的暫存器檔案中被忽略，因為它僅用於特定情況，我們將在以後的實驗中進行分析）。這些讀/寫連接埠可在每個週期執行兩條指令。來自暫存器檔案的輸入僅使用訊號的名稱在圖4的3:1和4:1多路開關中顯示。為簡單起見，未顯示與暫存器檔案的連接。

每個讀/寫連接埠都有一個5位元位址（raddr0 ... raddr3和waddr0 ... waddr2）以及一個1位元啟用訊號（rden0 ... rden3和wen0 ... wen2），後者未在圖4中顯示。寫入連接埠也有一個32位元寫入資料輸入（wd0 ... wd2），讀取連接埠有一個32位元讀取資料輸出（0 ... rd3）。暫存器檔案包含32個32位元暫存器（稱為x0-x31），其中x0硬接線到0。

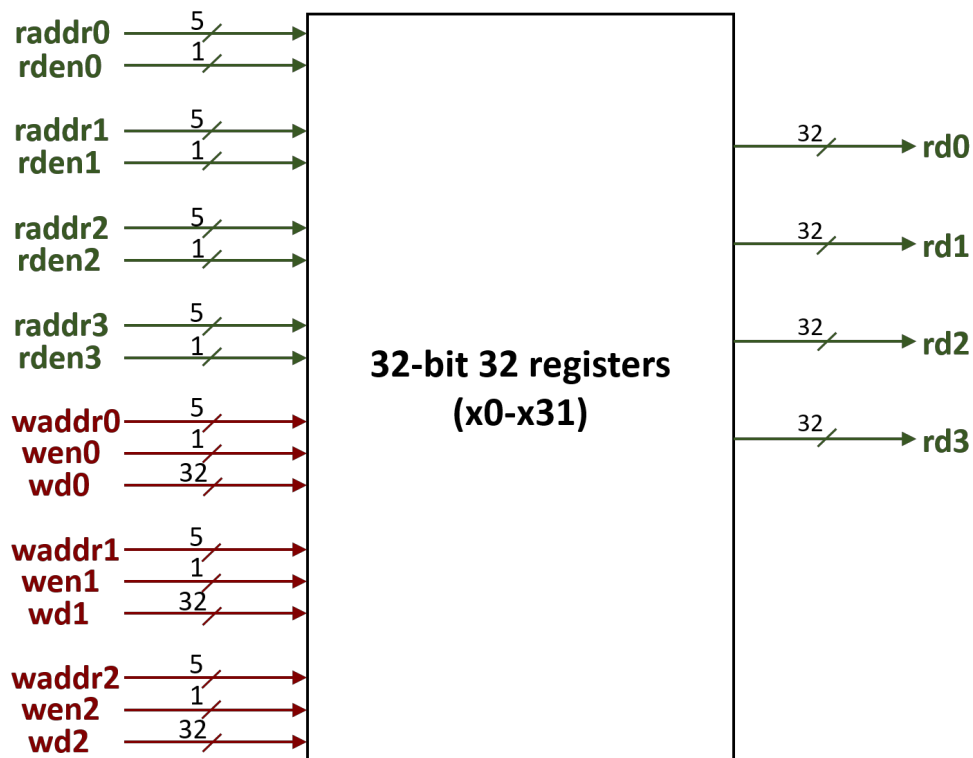


圖6. SweRV EH1中可用的暫存器檔案

任務：在模組**dec_gpr_ctl**中實作暫存器檔案，並在模組**dec**中將其實例化（參見圖7）。分析模組**dec_gpr_ctl**的Verilog程式碼及主要訊號的模擬（位於檔案 `[RVfpgaPath]/RVfpga/src/SweRVofSoC/SweRVEh1CoreComplex/dec/dec_gpr_ctl.sv`中），以瞭解其工作方式。請注意，SweRV EH1處理器允許包含多個暫存器檔案，但RVfpga系統中使用的組態僅使用一個暫存器檔案（參見檔案**dec.sv**的第402行：`localparam GPR_BANKS = 1;`）。

```

525     dec_gpr_ctl #(GPR_BANKS(GPR_BANKS),
526                  .GPR_BANKS_LOG2(GPR_BANKS_LOG2)) arf (.,
527                  // inputs
528                  .raddr0(dec_i0_rs1_d[4:0]), .rden0(dec_i0_rs1_en_d),
529                  .raddr1(dec_i0_rs2_d[4:0]), .rden1(dec_i0_rs2_en_d),
530                  .raddr2(dec_i1_rs1_d[4:0]), .rden2(dec_i1_rs1_en_d),
531                  .raddr3(dec_i1_rs2_d[4:0]), .rden3(dec_i1_rs2_en_d),
532
533                  .waddr0(dec_i0_waddr_wb[4:0]), .wen0(dec_i0_wen_wb), .wd0(dec_i0_wdata_wb[31:0]),
534                  .waddr1(dec_i1_waddr_wb[4:0]), .wen1(dec_i1_wen_wb), .wd1(dec_i1_wdata_wb[31:0]),
535                  .waddr2(dec_nonblock_load_waddr[4:0]), .wen2(dec_nonblock_load_wen), .wd2(lsu_nonblock_load_data[31:0]),
536
537                  // outputs
538                  .rd0(gpr_i0_rs1_d[31:0]), .rd1(gpr_i0_rs2_d[31:0]),
539                  .rd2(gpr_i1_rs1_d[31:0]), .rd3(gpr_i1_rs2_d[31:0])
540                  );

```

圖7. 模組**dec**內部的暫存器檔案實例化

ii. 執行階段

在本小節中，我們將分析SweRV EH1中可用管道的簡化版本：兩個**整數管道**（**I0管道**和**I1管道**）、一個**乘法管道**、一個**載入/儲存管道**和一個非管線34週期除法器。

I0/I1管道：兩個整數管道在圖4中以藍色顯示。它們分為三個階段，分別稱為**EX1**、**EX2**和**EX3**。這兩個管道的**EX1**階段均包含一個1週期延遲的ALU，能夠執行算術運算（例如**加法**或**減法**）以及邏輯運算（例如**邏輯與**或**邏輯或**）。**EX2**和**EX3**階段執行少量任務，但必須透過這兩個階段將A-L指令與其他需要三個週期來計算運算的指令類型（例如載入、儲存和乘法等）進行同步。在實驗12中，我們將更詳細地分析I0/I1管道。

乘法管道：乘法管道在圖4中以紅色顯示。它分為三個階段：**M1**、**M2**和**M3**。該管道包括一個能夠執行整數乘法的3週期乘法器。在實驗14中，我們將更詳細地分析乘法管道。

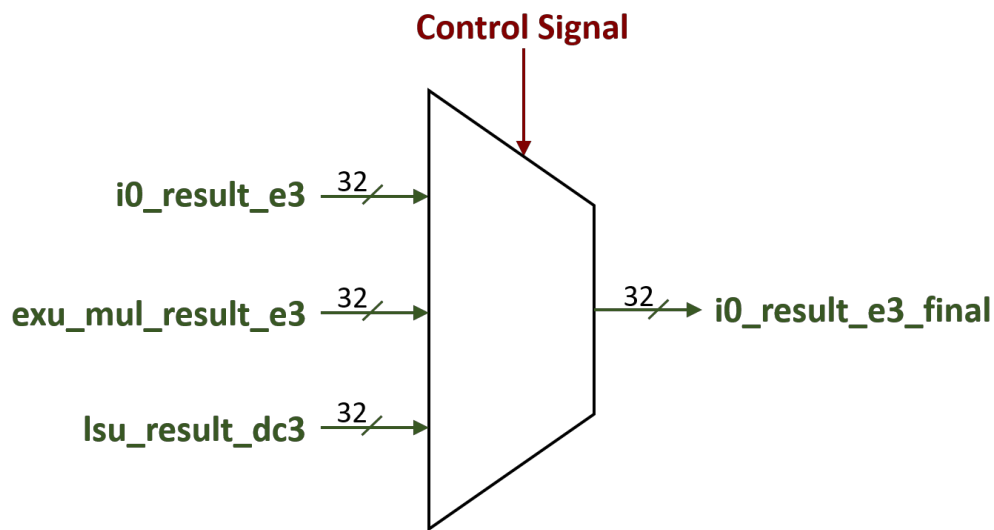
載入/儲存（L/S）管道：L/S管道在圖4中以綠色顯示。在實驗13中，我們將深入探索這條管線路徑。載入和儲存指令均透過L/S管道執行。它包括3個階段：

- **DC1**：在第一階段，加法器單元透過將暫存器基本位址和立即偏移量相加來計算位址。
- **DC2**：在第二階段，載入指令使用DC1中計算的位址讀取記憶體。如果位址對映到DCCM，則存取延遲僅為1個週期，管線將繼續執行，不會暫停。但是，如果存取對映到主記憶體，則管線可能需要暫停幾個週期，具體取決於阻塞/非阻塞載入如何使用以及相關性是否存在，我們將在後面的實驗中進一步分析。
- **DC3**：在第三階段，資料進行對齊和合併（例如，如果之前儲存到同一位址的儲存操作仍在執行，則可能需要將來自該儲存操作的資料轉送到載入階段）。在此階段，儲存指令開始寫入記憶體，過程持續幾個週期。如果寫入操作對映到DCCM，則資料和位址在傳送到DCCM之前均在儲存緩衝區中緩衝（正如我們在實驗13中分析的那樣）；如果寫入操作對映到主記憶體，則資料和位址均透過AXI匯流排傳送到外部記憶體（Lite DRAM控制器管理對該記憶體的存取）。

除法器：除法器在圖4中以白色顯示。它是一個非管線單元，需要最多34個週期來計算其結果。實驗14將更詳細地分析除法器。

兩個3:1多路開關：在第三個執行階段（EX3/DC3/M3）結束時（如圖4所示），使用兩個3:1多路開關（每路一個）從正確的管道（I0/I1、MUL或L/S）中選擇指令的結果。這兩個多路開關位於**dec_decode_ctl**模組中。與通路0相關聯的上方多路開關如圖8所示。該多路開關的三個輸入包括：

1. **I0管道結果**：i0_result_e3。實驗12將分析該路徑。
2. **L/S管道結果**：lsu_result_dc3。實驗13將分析該路徑。
3. **乘法管道結果**：exu_mul_result_e3。實驗14將分析該路徑。



```
2268 assign i0_result_e3_final[31:0] = (e3d.i0v & e3d.i0load) ? lsu_result_dc3[31:0] : (e3d.i0v & e3d.i0mul) ? exu_mul_result_e3[31:0] : i0_result_e3[31:0];
```

圖8. 透過3:1多路開關選擇EX3結果：圖和Verilog

任務：根據圖8分析多路開關的控制位元。請注意，控制位元在訊號e3d中，該訊號由訊號dd經管線處理得到，後一個訊號由控制單元在解碼階段產生（有關控制位元的說明，請參見SweRVref.docx）。

iii. 提交階段

在提交階段，兩個3:1多路開關（每路一個）選擇要寫回到暫存器檔案的結果（參見圖4）。與通路0相關聯的上方多路開關如圖9所示。它有三個輸入：

1. **EX3結果：**i0_result_e4。（EX3的3:1多路開關的輸出）。
2. **更正後的讀取資料：**lsu_result_corr_dc4。實驗13將分析該路徑。
3. **輔助ALU結果：**exu_i0_result_e4。為了簡單起見，這些ALU未在圖4中顯示。如前文所述，這些ALU可在出現資料冒險的必要情況下重複算術邏輯指令（有關詳細資訊，請參見實驗15）。

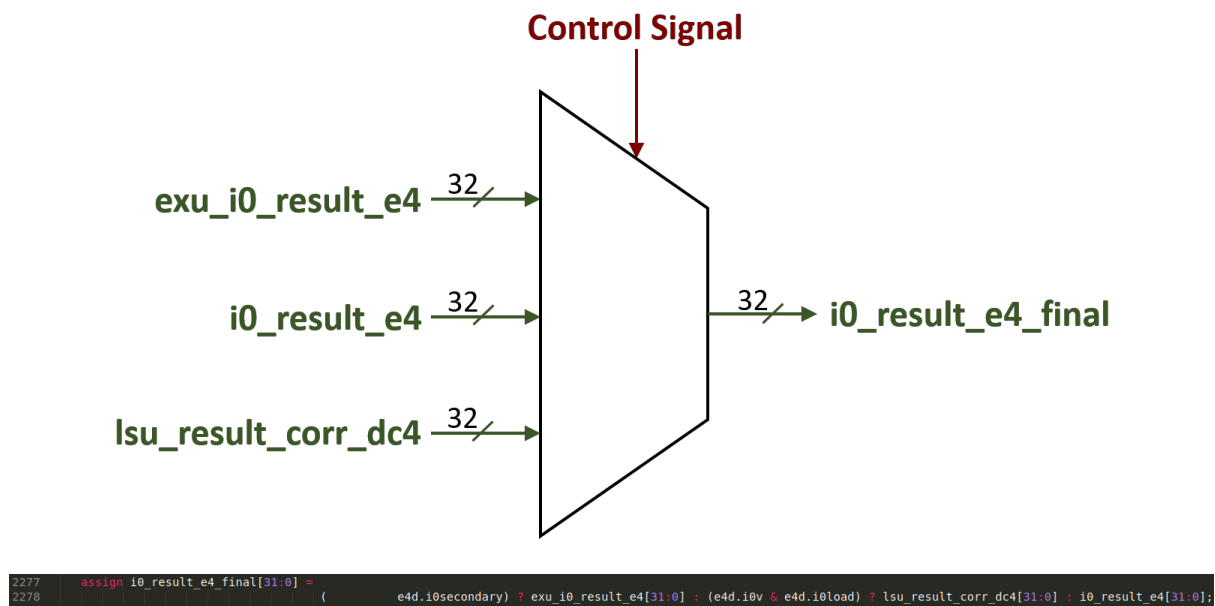


圖9. 透過3:1多路開關選擇最終結果：圖和Verilog


任務：根據圖9分析多路開關的控制位元，這些控制位元位於模組dec_decode_ctl中。

iv. 寫回階段

最後一個階段（寫回階段）使用如圖6所示的前兩個寫入連接埠（0和1）將結果寫入暫存器檔案（在實驗14中，我們將看到何時使用第三個寫入連接埠（2））。並非所有週期均寫入兩個結果：有些指令不寫入暫存器（即分支指令、儲存指令...），也並非所有週期均執行兩條指令。暫存器識別碼和啟用訊號在解碼階段產生並由控制管線暫存器提供。

D. Verilator中的模擬範例

在本部分中，我們將展示在SweRV EH1管線中平行執行的兩條指令的模擬過程，展示前幾部分中介紹的訊號。後面的實驗還將使用Verilator模擬來視覺化處理器的內部訊號並透過圖示闡述理論說明。

接下來將執行圖10所示的範例程式碼，重點關注mul和add指令（以紅色強調顯示），它們是無限迴圈的一部分。資料夾[RVfpgaPath]/RVfpga/Labs/Lab11/ExampleProgram提供了PlatformIO專案，這樣便可根據需要分析、模擬和變更程式。在PlatformIO中開啟專案並進行編譯（根據《入門指南》，可以透過按一下VSCode底部的按鈕  編譯專案）。反組譯檔案（位於

[RVfpgaPath]/RVfpga/Labs/Lab11/ExampleProgram/.pio/build/swervolf_nexys/firmware.dis)
會顯示位址和機器碼。請注意，兩條指令分別位於位址0x000000F0和0x000000F4處：

0x000000f0:	03de8e33	mul	t3,t4,t4
0x000000f4:	01ff0f33	add	t5,t5,t6

這兩條指令前後存在幾條nop（無操作）指令以將其與其他指令隔離，這樣便能更好地進行分析。nop指令不會變更系統的狀態。在RISC-V中，nop轉換為addi x0,x0,0，後者編碼為值為0x00000013的32位元機器指令。這段程式碼中定義了幾個巨集以將一些nop指令（1到10）插入其中（為簡單起見，巨集定義不包含在圖10中，但可以在PlatformIO專案中看到）。

為清楚起見，我們將按照SweRVref文件的第2部分所述的過程停用分支預測器和壓縮指令。

```
li x28, 0x1
li x29, 0x2
li x30, 0x4
li x31, 0x1

REPEAT:
    mul x28, x29, x29    # x28 = 2 * 2 = 4 (later iterations: 3*3=9, 4*4=16, ...)
    add x30, x30, x31    # x30 = 4 + 1 = 5 (later iterations: 5+1=6, 6+1=7, ...)
    INSERT_NOPS_10
    add x29, x29, 1      # x29 = x29 + 1
    INSERT_NOPS_10
    beq zero, zero, REPEAT # Repeat the loop
```

圖10. 迴圈中包含mul和add指令的範例程式

圖11和12所示為執行圖10中的程式時處理器訊號的Verilator波形。圖11所示為來自前三個管線階段（FC1、FC2和對齊 – 參見圖3）的訊號。圖12所示為來自其餘階段的訊號（參見圖4）。為了與圖3和圖4保持一致，我們將結果拆分為兩個圖，但請記住，這兩條指令來自對齊階段（圖11的右側）到解碼階段（圖12的左側）。

圖中包含以下訊號，用於在指令通過管線時追蹤指令（ifu追蹤對齊階段的指令，dec追蹤解碼階段的指令，ex追蹤X（X = 第一、第二和第三）執行階段的指令，e4追蹤提交階段的指令，wb追蹤寫回階段的指令）以及瞭解為指令指定的通路（為i0指定通路0，為i1指定通路1）。

- | | |
|---------------------------------|-----------|
| • ifu_i0_instr和ifu_i1_instr | →對齊階段的指令 |
| • dec_i0_instr_d和dec_i1_instr_d | →解碼階段的指令 |
| • i0_inst_e1和i1_inst_e1 | →EX1階段的指令 |
| • i0_inst_e2和i1_inst_e2 | →EX2階段的指令 |
| • i0_inst_e3和i1_inst_e3 | →EX3階段的指令 |
| • i0_inst_e4和i1_inst_e4 | →提交階段的指令 |
| • i0_inst_wb和i1_inst_wb | →寫回階段的指令 |

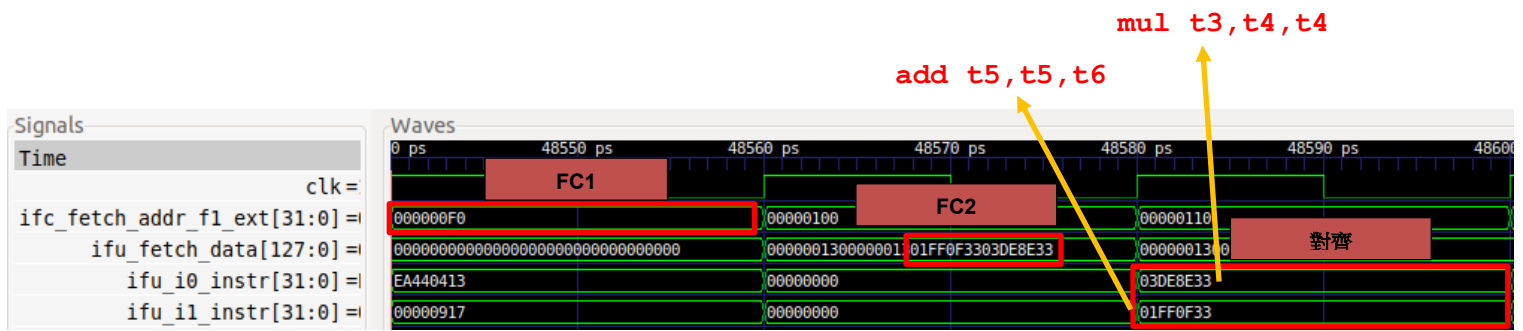


圖11. 前三個管線階段的模擬：FC1、FC2和對齊

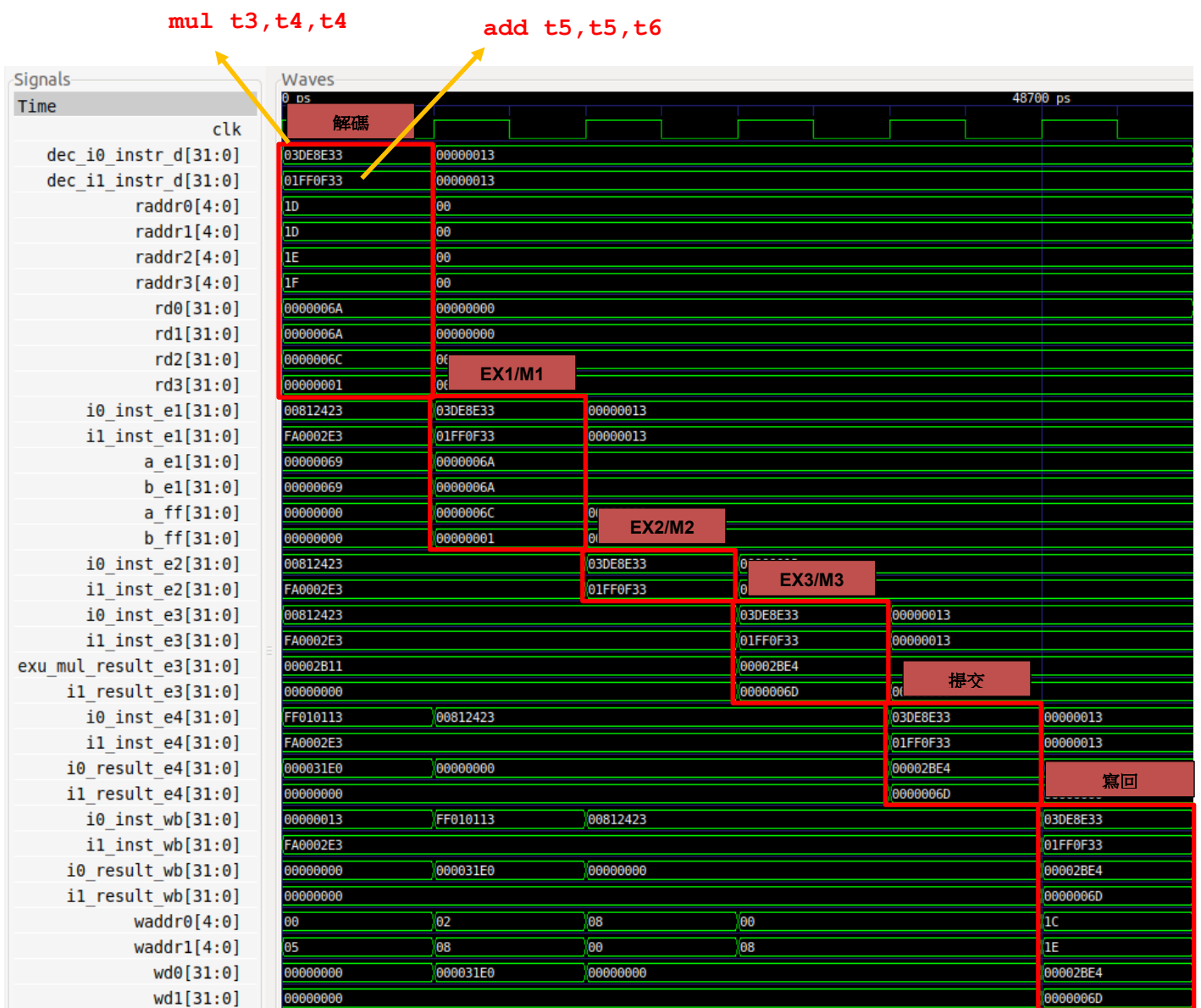



圖12. 最後階段的模擬：解碼、EX1/M1、EX2/M2、EX3/M3、提交和寫回

任務：按照以下步驟（如GSG第7部分中詳述）在自己的電腦上重複圖11和圖12中的模擬過程：

- 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/Lab11/ExampleProgram。
- 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）。
- 使用GTKWave開啟軌跡。
- 使用檔案*test_1.tcl*和*test_2.tcl*（在*[RVfpgaPath]/RVfpga/Labs/Lab11/ExampleProgram*中提供）開啟與圖11和圖12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），然後選擇*test_1.tcl*或*test_2.tcl*檔案。
- 按幾次「*Zoom In*」（放大）（）移動至48600 ps（或迴圈的任何其他迭代，第一次迭代除外）。

同時分析圖11和圖12中的波形以及圖3和圖4中的圖。圖中包括與每個管線階段相關聯的一些訊號。以紅色強調顯示的值對應於兩條經過管線的指令（mul和add）。

- **FC1：**在圖11的第一個週期中，訊號*ifc_fetch_addr_f1_ext[31:0]*（程式計數器，提供給指令記憶體）包含mul指令的位址（即，*指向*）（*ifc_fetch_addr_f1_ext* = 0x000000F0）。
- **FC2：**在圖11的第二個週期中，指令記憶體提供一個新的128位元訊號，其中包括我們在範例中分析的兩條指令（mul以綠色顯示，add以紅色顯示）：

```
ifu_fetch_data = 0x000000130000001301FF0F3303DE8E33
```

- **對齊：**在圖11的最後一個週期中，從新的128位元訊號中擷取兩條指令並分發給SweRV EH1包含的兩個通路。

```
ifu_i0_instr = 0x03DE8E33（通路0）
ifu_i1_instr = 0x01FF0F33（通路1）
```

- **解碼：**在圖12的第一個週期中，對兩條指令進行解碼 – 即從暫存器檔案中讀取指令的暫存器值，並產生控制位元（圖中未顯示，但可以按照SweRVref.docx所述新增其中一些控制位元）。運算元（暫存器值）置於rd0、rd1、rd2和rd3中。

```
rd0 = 0x0000006A
rd1 = 0x0000006A
rd2 = 0x0000006C
rd3 = 0x00000001
```

- **EX1/M1、EX2/M2、EX3/M3和提交：**在圖12接下來的三個週期中，執行加法和乘法。在EX3/M3結束時，使用兩個3:1多路開關選擇結果，然後傳播到提交階段。

```
i0_result_e4 = exu_mul_result_e3 = 0x6A * 0x6A = 0x2BE4
i1_result_e4 = i1_result_e3      = 0x6C + 0x01 = 0x6D
```

- **寫回**：在圖12的最後一個週期中，將結果寫回到暫存器檔案中。

```
waddr0 = 0x1C    wd0 = 0x2BE4
waddr1 = 0x1E    wd1 = 0x6D
```

3. SweRV EH1中的硬體計數器

接下來展示如何使用效能計數器來分析處理器效能。硬體計數器是目前大多數處理器中包含的一組特殊用途暫存器，用於記錄各種指標，例如執行的指令數、執行的週期數、每條指令的平均時鐘週期數（CPI）、指令快取命中/未命中次數、正確/錯誤預測分支數等。

在實驗12-20中，我們將定期使用SweRV EH1中的效能計數器來測量和比較不同的幅值。

實際基準：在資料夾[RVfpgaPath]/RVfpga/Labs/Lab20/RealBenchmarks中，我們提供了三個實際應用（CoreMark、Dhrystone和影像處理），實驗20中將使用它們來測試SweRV EH1處理器的不同功能。補充文件SweRVref.docx在第6部分中簡要描述了這些應用，實驗20將展開描述並提出幾個任務。

A. SweRV EH1中的效能計數器

RISC-V SweRV EH1程式設計師參考手冊（https://github.com/chipsalliance/Cores-SweRV/blob/master/docs/RISC-V_SweRV_EH1_PRM.pdf）介紹了RISC-V處理器的基本硬體效能監視功能。必須實作以下效能計數器，這些計數器也是控制和狀態暫存器（CSR）：

- **mcycle**：自過去任意時間以來，hart（硬體執行緒）已執行的時鐘週期數。
- **minstret**：自過去任意時間以來，hart已淘汰的指令數。
- **mhpmpcounter3–mhpmpcounter31**：29個其他事件計數器。事件選擇器CSR（**mhpmevent3–mhpmevent31**）為WARL（寫入任何值，讀取合法值）暫存器，用於控制哪個事件導致相應計數器遞增。這些事件的含義由平台定義，但事件0保留，表示「無事件」。

並非所有計數器都需要實作。將計數器及其相應的事件選擇器硬接線到0是一種合法實作。具體來說，在SweRV EH1中，只有事件計數器3到6（**mhpmpcounter3–mhpmpcounter6**）及其相應的事件選擇器（**mhpmevent3–mhpmevent6**）正常工作，而事件計數器7到31（**mhpmpcounter7–mhpmpcounter31**）及其相應的事件選擇器（**mhpmevent7–mhpmevent31**）硬接線到0。這些計數器的啟用由**mgpmc**暫存器的位元0（0 = 停用，1 = 啟用）控制。

SweRV EH1程式設計師參考手冊的第7章 (https://github.com/chipsalliance/Cores-SweRV/blob/master/docs/RISC-V_SweRV_EH1_PRM.pdf) 詳細介紹了SweRV EH1中的四個效能計數器的特性和操作：

- 四個標準64位元寬事件計數器
- 每個計數器具有標準單獨事件選擇功能
- 標準選擇性計數啟用/停用控制能力
- 同步計數器啟用/停用控制能力
- 標準週期計數器
- 標準淘汰指令計數器
- 支援基於標準SoC的機器計時器暫存器

該文件中的表7-2列出了SweRV EH1中的50個可計數事件，這些事件在表1中進行了摘要。

表1. SweRV EH1中的可計數事件清單

0	保留	17	CSR讀/寫	34	SB/WB週期暫停
1	時鐘週期有效	18	CSR寫/讀==0	35	DMA DCCM交易週期暫停
2	指令快取命中	19	Ebreak	36	DMA ICCM交易週期暫停
3	指令快取未命中	20	Ecall	37	發生例外狀況
4	指令已提交	21	Fence	38	發生計時器中斷
5	16位元指令已提交	22	Fence.i	39	發生外部中斷
6	32位元指令已提交	23	Mret	40	TLU排清
7	指令已對齊	24	分支已提交	41	分支錯誤排清
8	指令已解碼	25	分支預測錯誤	42	指令匯流排交易 - 指令
9	乘法已提交	26	進行分支	43	資料匯流排交易 - ld/st
10	除法已提交	27	分支不可預測	44	資料匯流排交易未對齊
11	載入已提交	28	擷取週期暫停	45	指令匯流排錯誤
12	儲存已提交	29	對齊器週期暫停	46	資料匯流排錯誤
13	載入未對齊	30	解碼週期暫停	47	由於指令匯流排忙碌，週期暫停
14	儲存未對齊	31	後同步週期暫停	48	由於資料匯流排忙碌，週期暫停
15	ALU已提交	32	預先同步週期暫停	49	中斷週期已停用
16	CSR讀取	33	週期凍結	50	停用時中斷週期暫停

B. 透過Western Digital的處理器支援套件（Processor Support Package，PSP）使用效能計數器

在暫存器等級使用效能監視系統有點複雜；幸運的是，WD PSP

(<https://github.com/westerndigitalcorporation/riscv-fw-infrastructure>) 中的一些函數提供了一種更簡單的效能監視方法。如果已按照GSG中的說明安裝了PlatformIO，Ubuntu系統中應存在以下兩個檔案：

- `~/.platformio/packages/framework-wd-riscv-sdk/psppsp/performance_monitor_eh1.c`
- `~/.platformio/packages/framework-wd-riscv-sdk/psppsp/api/inc/performance_monitor_eh1.h`

Windows： `.platformio`資料夾位於使用者資料夾（`C:\Users\<USER>`）內。請注意，可能需要啟用系統才能檢視隱藏的檔案/資料夾。

macOS： 與在Linux中一樣，`.platformio`資料夾位於主資料夾（`~/.platformio`）內。

憑藉.c檔案（`psppsp/performance_monitor_eh1.c`）實作的函數，可以執行諸如啟用/停用群組效能監視器（`pspEnableAllPerformanceMonitor`）、將計數器與事件配對

(`pspPerformanceCounterSet`) 或獲取計數器值 (`pspPerformanceCounterGet`) 之類操作。

.h檔案 (`psp_performance_monitor_eh1.h`) 在 `typedef enum pspPerformanceMonitorEvents` 中提供表1中的每個事件的名稱。

[*RVfpgaPath*]/RVfpga/Labs/Lab11/HwCounters_Example中提供的以下範例 (圖13) 說明了如何使用SweRV EH1中的四個硬體計數器來測量：週期數、指令數、已提交的分支和預測錯誤的分支。Main函數：

- 初始化UART (`uartInit()`)
- 啟用硬體計數器 (`pspEnableAllPerformanceMonitor(1)`)
- 為每個計數器 (`D_PSP_COUNTER0` - `D_PSP_COUNTER3`) 指定要測量的事件 (週期數、指令數、已提交的分支和預測錯誤的分支)
- 讀取計數器 (`pspPerformanceCounterGet(D_PSP_COUNTER0)`)
- 呼叫一個簡單的組合語言程式 (`Test_Assembly()`) 並再次讀取計數器
- 使用函數 `printfNexys` 輸出每個計數器的值。

完成一些暫存器的初始化之後，`Test_Assembly()` 函數將重複迴圈1,000,000次；該迴圈包含五個算術邏輯 (AL) 指令和一個條件分支。反組譯檔案也顯示在圖13的末尾，以方便您瞭解構成迴圈主體的32位元機器指令的值。

Test.C檔案

```
#if defined(D_NEXYS_A7)
#include <bsp_printf.h>
#include <bsp_mem_map.h>
#include <bsp_version.h>
#else
    PRE_COMPILED_MSG("no platform was defined")
#endif

#include <psp_api.h>

extern void Test_Assembly(void);

int main(void)
{
    int cyc_beg, cyc_end;
    int instr_beg, instr_end;
    int BrCom_beg, BrCom_end;
    int BrMis_beg, BrMis_end;

    /* Initialize Uart */

    uartInit();

    pspEnableAllPerformanceMonitor(1);

    pspPerformanceCounterSet(D_PSP_COUNTER0, E_CYCLES_CLOCKS_ACTIVE);
    pspPerformanceCounterSet(D_PSP_COUNTER1, E_INSTR_COMMITTED_ALL);
    pspPerformanceCounterSet(D_PSP_COUNTER2, E_BRANCHES_COMMITTED);
    pspPerformanceCounterSet(D_PSP_COUNTER3, E_BRANCHES_MISPREDICTED);

    cyc_beg = pspPerformanceCounterGet(D_PSP_COUNTER0);
    instr_beg = pspPerformanceCounterGet(D_PSP_COUNTER1);
    BrCom_beg = pspPerformanceCounterGet(D_PSP_COUNTER2);
    BrMis_beg = pspPerformanceCounterGet(D_PSP_COUNTER3);
```

```

Test_Assembly();

cyc_end   = pspPerformanceCounterGet(D_PSP_COUNTER0);
instr_end = pspPerformanceCounterGet(D_PSP_COUNTER1);
BrCom_end = pspPerformanceCounterGet(D_PSP_COUNTER2);
BrMis_end = pspPerformanceCounterGet(D_PSP_COUNTER3);

printfNexys("Cycles = %d", cyc_end-cyc_beg);
printfNexys("Instructions = %d", instr_end-instr_beg);
printfNexys("BrCom = %d", BrCom_end-BrCom_beg);
printfNexys("BrMis = %d", BrMis_end-BrMis_beg);

while(1);
}

```

Test_Assembly.S檔案

```

.global Test_Assembly

.text

Test_Assembly:

li t1, 0x1
li t3, 0x3
li t4, 0x4
li t5, 0x5
li t6, 0x6
li a0, 0x0
lui a1, 0xF4
add a1, a1, 0x240
nop

REPEAT:
    add a0, a0, 1
    add t3, t3, t1
    sub t4, t4, t1
    or  t5, t5, t1
    xor t6, t6, t1
    bne a0, a1, REPEAT # Repeat the loop

.end

```

firmware.dis檔案

```

000001e4 <Test_Assembly>:
1e4: 00100313      li      t1,1
1e8: 00300e13      li      t3,3
1ec: 00400e93      li      t4,4
1f0: 00500f13      li      t5,5
1f4: 00600f93      li      t6,6
1f8: 00000513      li      a0,0
1fc: 000f45b7      lui     a1,0xf4
200: 24058593      addi    a1,a1,576 # f4240 <_sp+0xf0788>
204: 00000013      nop

00000208 <REPEAT>:
208: 00150513      addi    a0,a0,1
20c: 006e0e33      add     t3,t3,t1
210: 406e8eb3      sub     t4,t4,t1

```

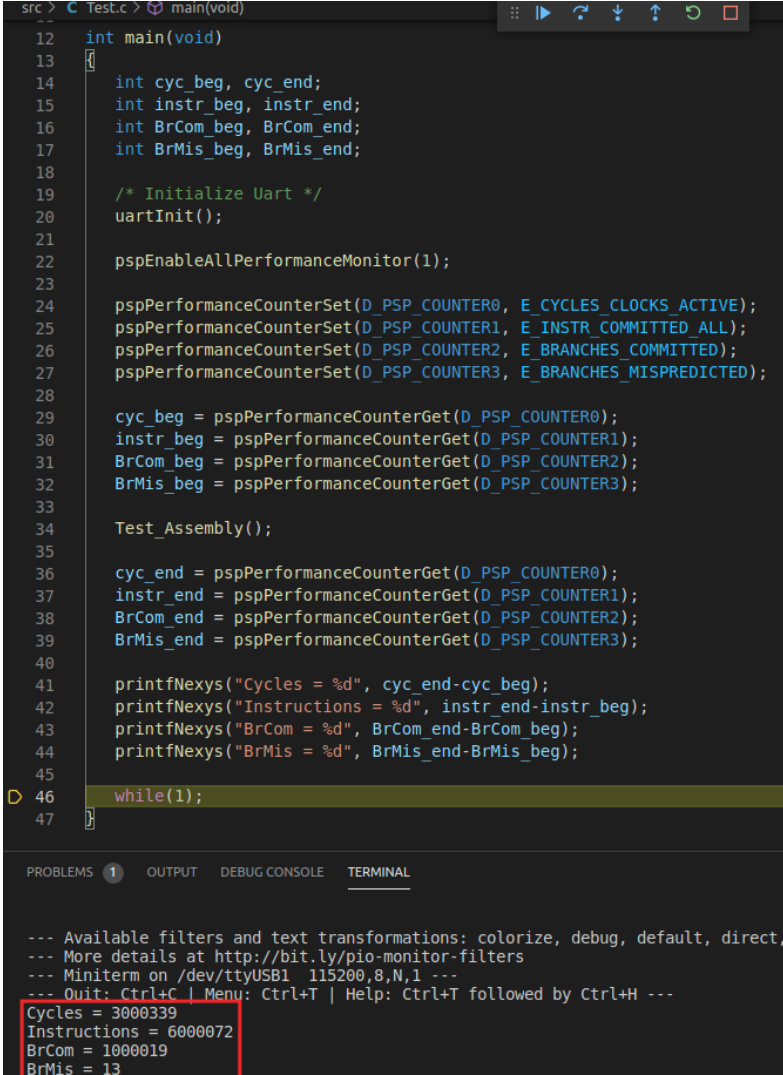
```

214: 006f6f33      or      t5,t5,t1
218: 006fcfb3      xor      t6,t6,t1
21c: feb516e3      bne      a0,a1,208 <REPEAT>

```

圖13. Test.C、Test_Assembly.S和firmware.dis

任務：按照GSG所述在Nexys A7板上執行圖13中的程式。對於測量的四個事件，應得到圖14所示的結果。解釋並證明結果。



```

src > C Test.c > main(void)
12 int main(void)
13 {
14     int cyc_beg, cyc_end;
15     int instr_beg, instr_end;
16     int BrCom_beg, BrCom_end;
17     int BrMis_beg, BrMis_end;
18
19     /* Initialize Uart */
20     uartInit();
21
22     pspEnableAllPerformanceMonitor(1);
23
24     pspPerformanceCounterSet(D_PSP_COUNTER0, E_CYCLES_CLOCKS_ACTIVE);
25     pspPerformanceCounterSet(D_PSP_COUNTER1, E_INSTR_COMMITTED_ALL);
26     pspPerformanceCounterSet(D_PSP_COUNTER2, E_BRANCHES_COMMITTED);
27     pspPerformanceCounterSet(D_PSP_COUNTER3, E_BRANCHES_MISPREDICTED);
28
29     cyc_beg = pspPerformanceCounterGet(D_PSP_COUNTER0);
30     instr_beg = pspPerformanceCounterGet(D_PSP_COUNTER1);
31     BrCom_beg = pspPerformanceCounterGet(D_PSP_COUNTER2);
32     BrMis_beg = pspPerformanceCounterGet(D_PSP_COUNTER3);
33
34     Test_Assembly();
35
36     cyc_end = pspPerformanceCounterGet(D_PSP_COUNTER0);
37     instr_end = pspPerformanceCounterGet(D_PSP_COUNTER1);
38     BrCom_end = pspPerformanceCounterGet(D_PSP_COUNTER2);
39     BrMis_end = pspPerformanceCounterGet(D_PSP_COUNTER3);
40
41     printfNexys("Cycles = %d", cyc_end-cyc_beg);
42     printfNexys("Instructions = %d", instr_end-instr_beg);
43     printfNexys("BrCom = %d", BrCom_end-BrCom_beg);
44     printfNexys("BrMis = %d", BrMis_end-BrMis_beg);
45
46     while(1);
47 }

```

```

--- Available filters and text transformations: colorize, debug, default, direct,
--- More details at http://bit.ly/pio-monitor-filters
--- Miniterm on /dev/ttyUSB1 115200,8,N,1 ---
--- Quit: Ctrl+C | Menu: Ctrl+T | Help: Ctrl+T followed by Ctrl+H ---
Cycles = 3000339
Instructions = 6000072
BrCom = 1000019
BrMis = 13

```

圖14. 執行Test.C

任務：在圖13所示程式的硬體計數器中測量其他事件。為此，必須使用 pspPerformanceCounterSet 函數在 Test.c 檔案中變更待測量事件的組態。請注意，可以使用 WD PSP 檔案中定義的巨集設定不同的事件（如表1所示）：*.platformio/packages/framework-wd-riscv-sdk/psp/api_inc/psp_performance_monitor_eh1.h*。例如，如果要測量 I\$ 未命中數而不是分支未命中數，則必須在檔案中

將 Test.c 行：`pspPerformanceCounterSet(D_PSP_COUNTER3, E_BRANCHES_MISPREDICTED);` 替換為行：`pspPerformanceCounterSet(D_PSP_COUNTER3, E_I_CACHE_MISSES);`

任務：在Test_Assembly函數中提供其他程式並檢查不同的事件是否提供了預期的結果。可以嘗試其他指令，例如載入、儲存、乘法、除法...以及引發管線暫停的冒險。