

RVfpga

瞭解電腦架構的完整課程

致謝

作者

Sarah Harris教授
Daniel Chaver教授
Zubair Kakakhel
M. Hamza Liaqat

顧問

David Patterson教授

貢獻者

Robert Owen
Olof Kindgren
Luis Piñuel教授
Ivan Kravets
Valerii Koval
Ted Marena
Roy Kravitz教授

聯合作者

José Ignacio Gómez教授	Francisco Tirado教授	Gage Elerding
Christian Tenllado教授	Román Hermida教授	Brian Cruickshank教授
Daniel León教授	Ataur Patwary教授	Deepen Parmar
Katzalin Olcoz教授	Cathal McCabe	Thong Doan
Alberto del Barrio教授	Dan Hugo	Oliver Rew
Fernando Castro教授	Braden Harwood	Niko Nikolay
Manuel Prieto教授	David Burnett教授	Guanyang He

贊助商與支持者

Western Digital®

Imagination

CHIPS
ALLIANCE

RISC-V®

DIGILENT®
A National Instruments Company

XILINX.
| UNIVERSITY PROGRAM

Digi-Key®
ELECTRONICS

Esperanto
TECHNOLOGIES

codasip®

硬禾学堂

ANDES
TECHNOLOGY

PLATFORMIO.ORG

簡介

- RISC-V FPGA (**RVfpga**) 是一套教學套件，其中包含一套指令、工具和實驗，用於展示如何：
 - 確定商用RISC-V晶片上系統 (SoC) 的目標**FPGA**
 - 設計RISC-V SoC程式
 - 向RISC-V SoC新增更多功能
 - 分析和修改RISC-V核心與記憶體階層
- 該教學包由**Imagination Technologies**與其學術和行業合作伙伴共同開發。
- RVfpga系統圍繞Chips Alliance的**SweRVolf SoC**（基於Western Digital的RISC-V **SweRV EH1**核心）建立。

RVfpga概述

- **RVfpga教學套件提供：**
 - 一門內容全面且免費發佈的完整**RISC-V**課程
 - 一種容易上手的**動手實驗**方法來瞭解**RISC-V**處理器和**RISC-V**生態系統
 - 一款以低成本**FPGA**（在許多大學和公司都比較常用）為目標的**RISC-V**系統。
- 完成**RVfpga**課程後，使用者將瞭解並掌握如何使用和修改**RISC-V**處理器、**SoC**和生態系統。

第二門課程：RVfpga-SoC

- **RVfpga-SoC**課程是第二門課程：
 - 介紹如何**建立RISC-V SoC**、安裝**Zephyr RTOS**（即時作業系統）並在其上執行程式（包括簡單的**Tensorflow**程式）。
 - 提供**5**個實驗。
- 兩門課程（**RVfpga**和**RVfpga-SoC**）均可在以下網址單獨下載（註冊後免費）：
<https://university.imgtec.com/rvfpga/>
- 剩餘的幻燈片 重點關注**RVfpga**課程。

RVfpga課程內容



RVfpga內容

- 入門指南

- 快速入門指南
- RISC-V架構和RVfpga概述
- 安裝工具（VSCode、PlatformIO、Vivado、Verilator和Whisper）
- 透過硬體和模擬執行RVfpga系統

- 實驗

- **1-10**：在Vivado中編譯RVfpga系統，編寫RVfpga系統，透過新增周邊設備延伸RVfpga系統（2020年11月發佈）
- **11-20**：分析和修改RVfpga的RISC-V核心和記憶體系統（計劃於2021年第4季度發佈）

- **2-3學期課程**

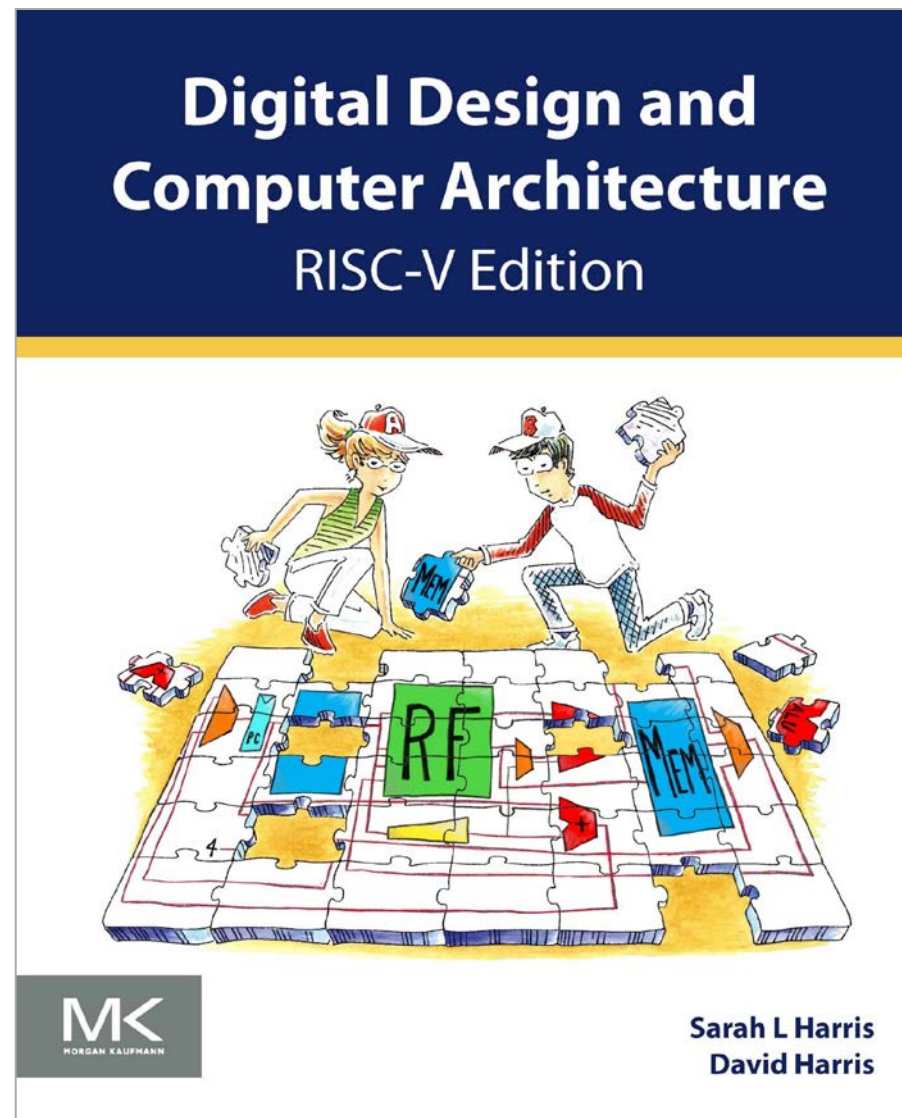
- 本科（實驗1-10）
- 碩士/本科高年級（實驗11-20）

- **知識儲備要求**

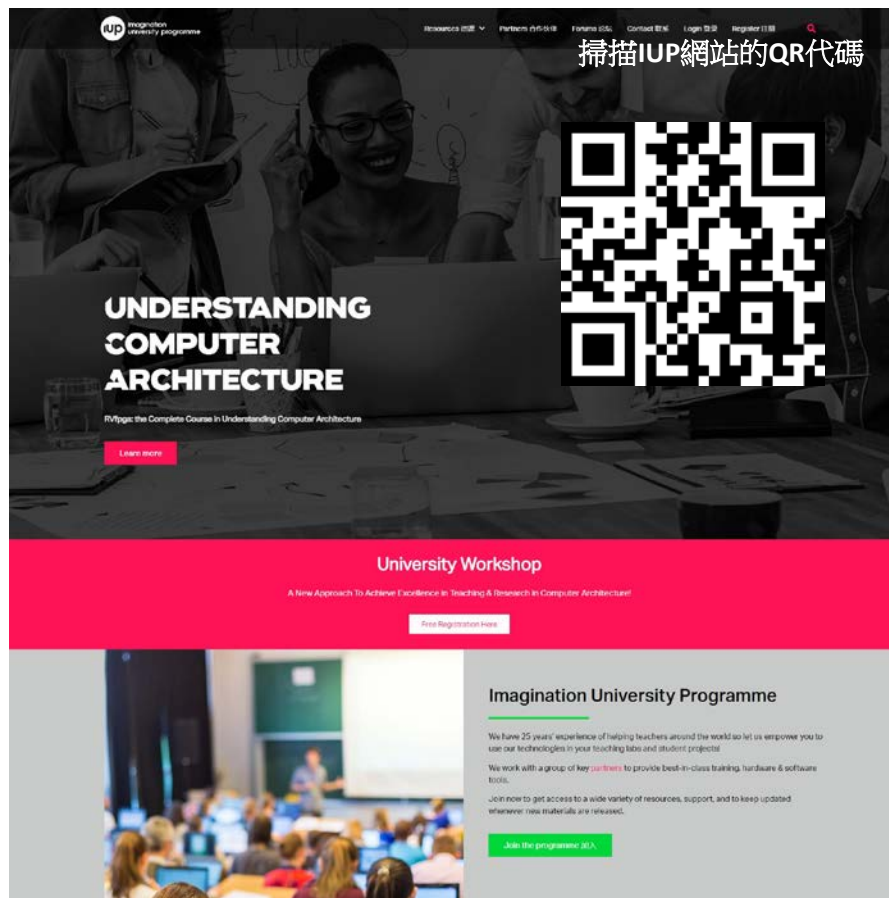
- 基本瞭解數位設計、高階程式設計（最好是C語言）、指令集架構和組合語言程式設計、處理器微架構和記憶體系統（相關資訊請參閱《數位設計和電腦體系結構》*RISC-V*版本（作者：Harris & Harris）© Elsevier（預計出版時間：2021年夏季））。
- 在整個RVfpga課程中，將通過動手實驗來擴展和鞏固上述主題

教材

在開始RVfpga課程之前建議
瞭解以下教材：《數位設計
和電腦結構》：**RISC-V**版本
（作者：Harris & Harris）
© Elsevier，2021年。



如何取得RVfpga



Imagination大學計劃網站

- 通過以下網址註冊**Imagination**大學計劃（**IUP**）－面向全球教師、研究人員和學生：

<https://university.imgtec.com>

- 接收發佈更新和通知
- 申請並下載資料
- 支援論壇：PowerVR、RVfpga和AI論壇；IUP論壇（面向課程/教學討論）
- 社交媒體：
 - **IUP主管Robert Owen**：@UniPgm
 - **Imagination Technologies**：@ImaginationTech
 - 微信和微博：ImaginationTech

RVfpga所需的軟體和硬體

軟體

Xilinx **Vivado** 2019.2 WebPACK

PlatformIO – Microsoft **Visual Studio Code** – 的延伸模組 – 支援Chips Alliance平台，其中包括：RISC-V工具鏈、OpenOCD、Verilator HDL模擬器、WD Whisper指令集模擬器（ISS）

硬體*

Digilent的**Nexys A7**/Nexys 4 DDR FPGA開發板

*選用：所有實驗只需通過模擬即可完成，因此該硬體只是推薦使用，並非必須使用。

RISC-V核心和 SOC

核心：Western Digital的**SweRV EH1****

SoC：Chips Alliance的**SweRVolf****

**開放原始碼 - 在RVfpga軟體套件中提供。

除FPGA開發板外，所有軟硬體均可免費使用（FPGA開發板的官方價為265美元，學術優惠價為199美元）

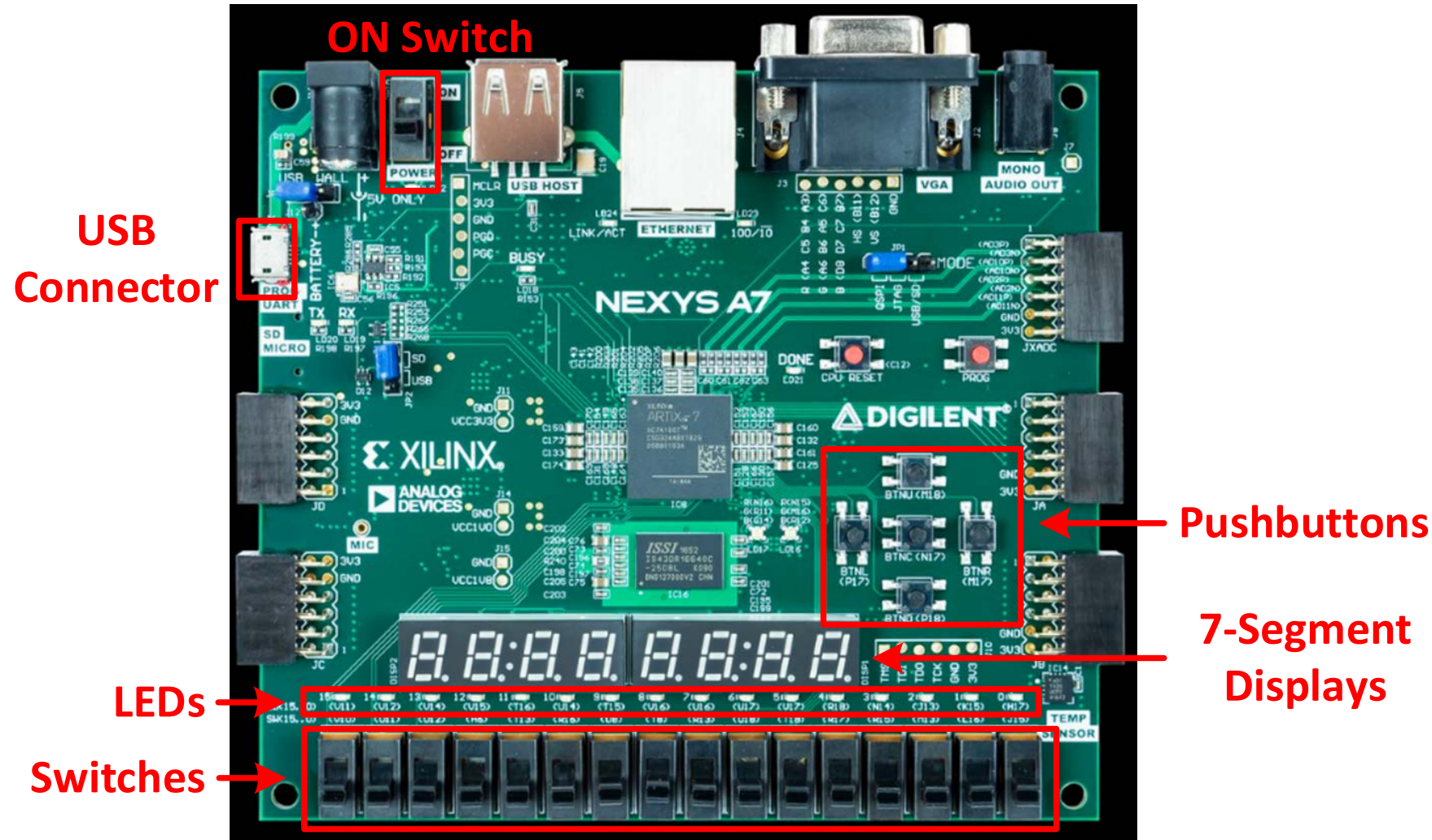
支援的平台

- 作業系統
 - **Ubuntu 18.04**（更高版本也可能適用）
 - **Windows 10**
 - **macOS**

RVfpga軟體工具

- **Xilinx的Vivado IDE**
 - 檢視RVfpga原始程式檔（Verilog/SystemVerilog）和階層
 - 為以Nexys A7開發板為目標的RVfpga建立bit檔（FPGA配置檔）
- **PlatformIO – Visual Studio Code（VSCode）的延伸模組**
 - 將RVfpga系統下載到Nexys A7開發板
 - 在RVfpga系統上編譯、下載、執行和除錯C程式和組合語言程式
- **Verilator – 一款HDL（硬體描述語言）模擬器**
 - 在HDL（低）層級模擬RVfpga系統，以分析其內部訊號

Nexys A7-100T FPGA開發板



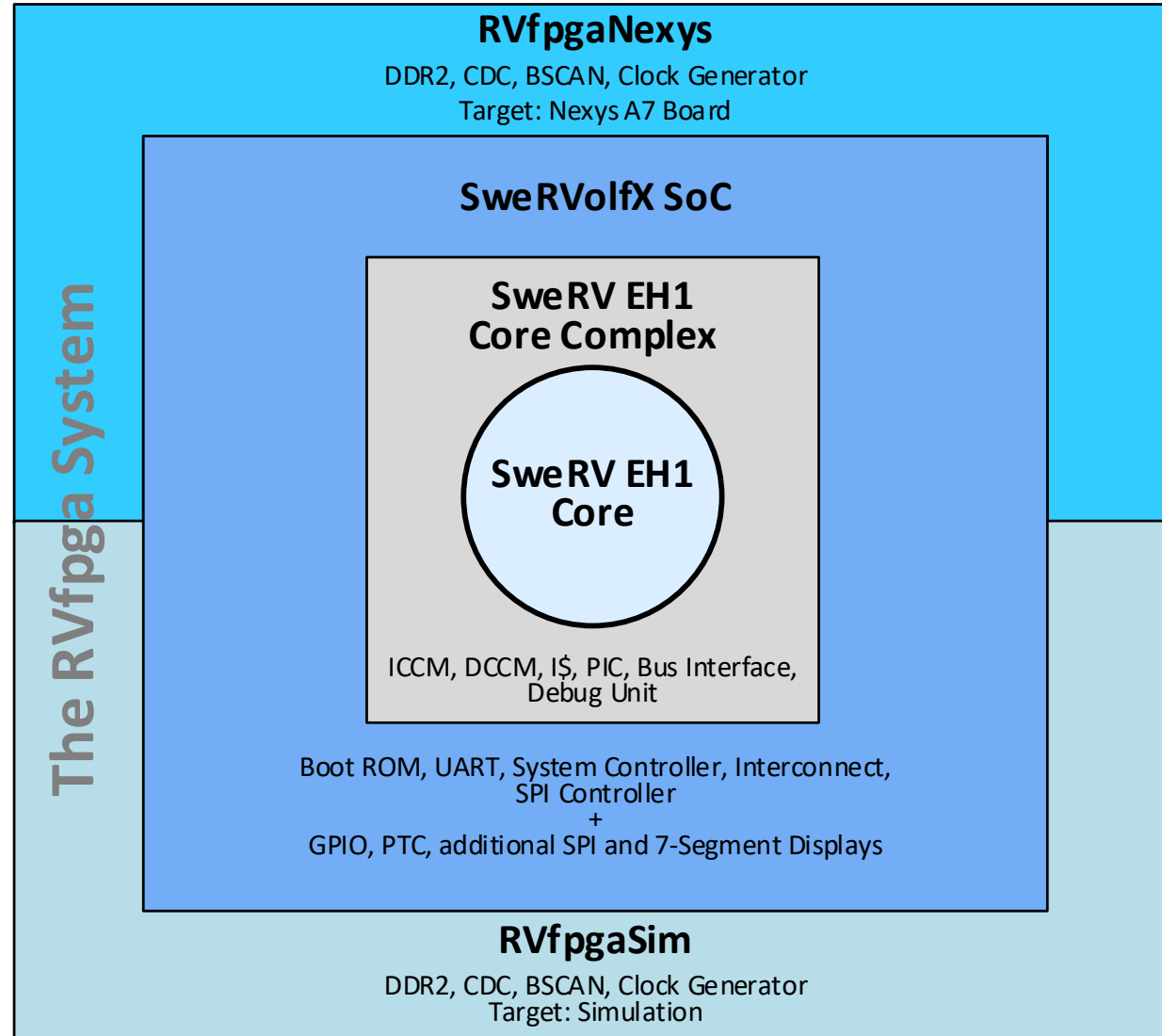
開發板圖片來源：<https://reference.digilentinc.com/>

- 包括**Artix-7**可現場程式化閘道陣列（FPGA）
- 包括週邊設備（即LED、開關、按鈕、7段顯示器、加速計、溫度感應器和麥克風等）
- 可從**digilentinc.com**和其他供應商處購買

RISC-V核心和SoC



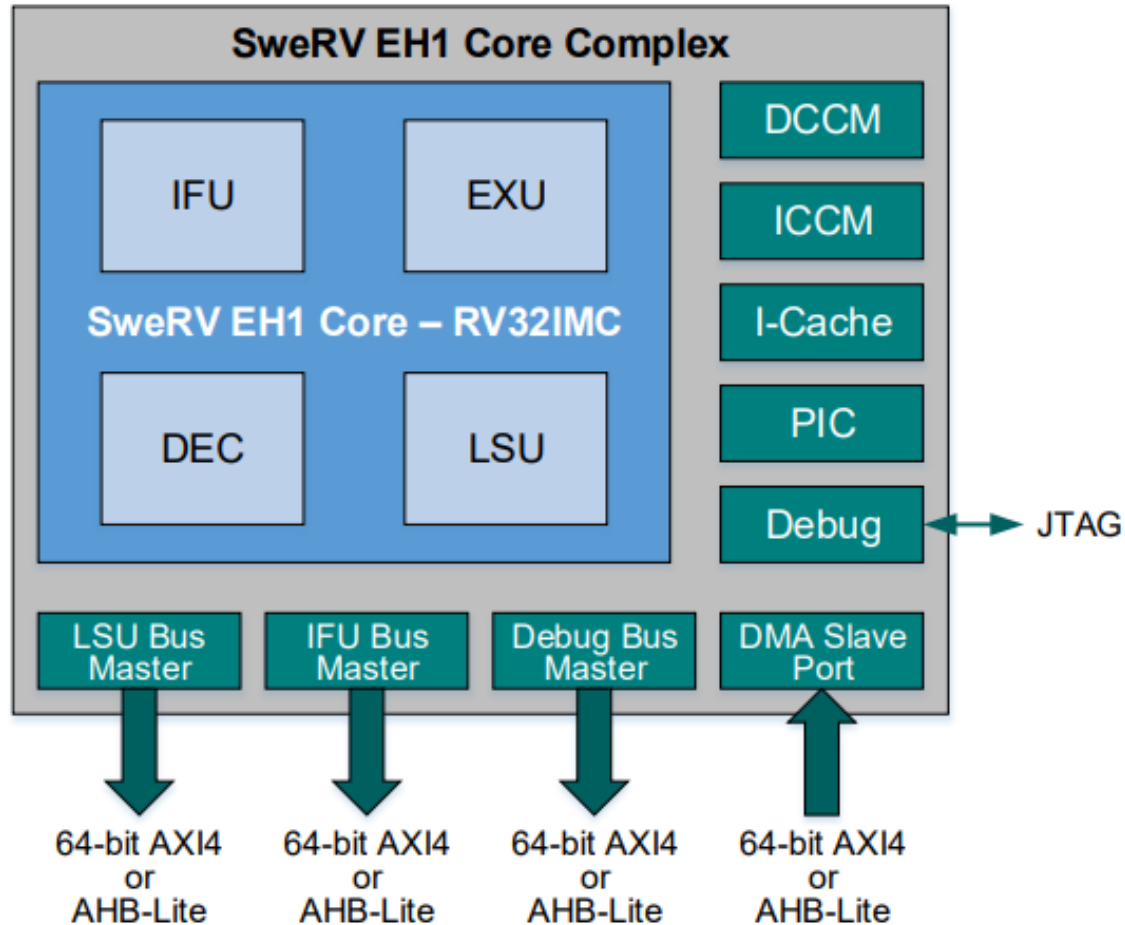
RVfpga層級



RVfpga層級

名稱	說明
SweRV EH1核心	由Western Digital開發的開放原始碼商用RISC-V核心（ https://github.com/chipsalliance/Cores-SweRV ）。
SweRV EH1核心組合	一種新增了記憶體（ICCM、DCCM和指令快取）、可程式設定中斷控制器（Programmable Interrupt Controller，PIC）、匯流排介面和除錯單元的SweRV EH1核心（ https://github.com/chipsalliance/Cores-SweRV ）。
SweRVolfX （延伸SweRVolf）	<p>我們在RVfpga課程中使用的晶片上系統。它是SweRVolf的延伸。</p> <p><u>SweRVolf</u>（https://github.com/chipsalliance/Cores-SweRVolf）：一種圍繞SweRV EH1核心組合建構的開放原始碼SoC。它新增了啟動ROM、UART介面、系統控制器、互連（AXI互連、Wishbone互連和AXI轉Wishbone橋接）以及SPI控制器。</p> <p><u>SweRVolfX</u>：與SweRVolf相比新增了4個新周邊設備：GPIO、PTC、另一個SPI以及用於8位元7段顯示器的控制器。</p>
RVfpgaNexys	<p>以Nexys A7開發板及其周邊設備為目標的SweRVolfX SoC。它新增了DDR2介面、CDC（跨時脈變換）單元、BSCAN邏輯（用於JTAG介面）和時鐘產生器。</p> <p>RVfpgaNexys與SweRVolf Nexys基本相同（https://github.com/chipsalliance/Cores-SweRVolf），只是後者基於SweRVolf。</p>
RVfpgaSim	<p>一種具有測試平台包裝函式和AXI記憶體的SweRVolfX SoC，用於模擬。</p> <p>RVfpgaSim與SweRVolf Sim基本相同（https://github.com/chipsalliance/Cores-SweRVolf），只是後者基於SweRVolf。</p>

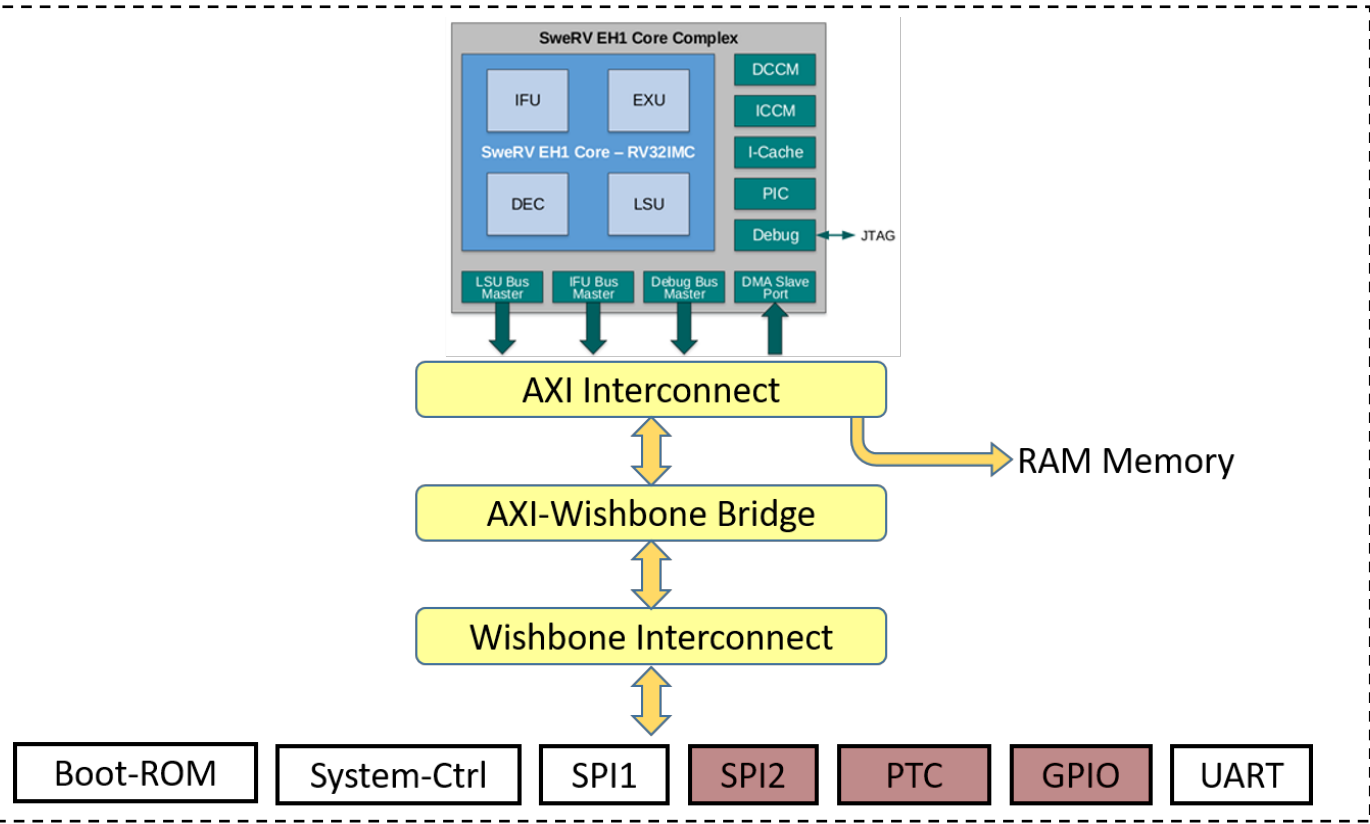
SweRV EH1核心和SweRV EH1核心組合



- Western Digital的開放原始碼核心
- 32位元（RV32ICM）超標量核心，具有雙發射9階段管線
- 獨立的指令和資料記憶體（ICCM和DCCM），與核心緊密耦合
- 4向集關聯指令緩存，具有同位檢查或ECC保護
- 可程式化中斷控制器
- 符合RISC-V偵錯規格的核心偵錯單元
- 系統匯流排：AXI4或AHB-Lite

圖片來源：https://github.com/chipsalliance/Cores-SweRV/blob/master/docs/RISC-V_SweRV_EH1_PRM.pdf

SweRVolfX SoC

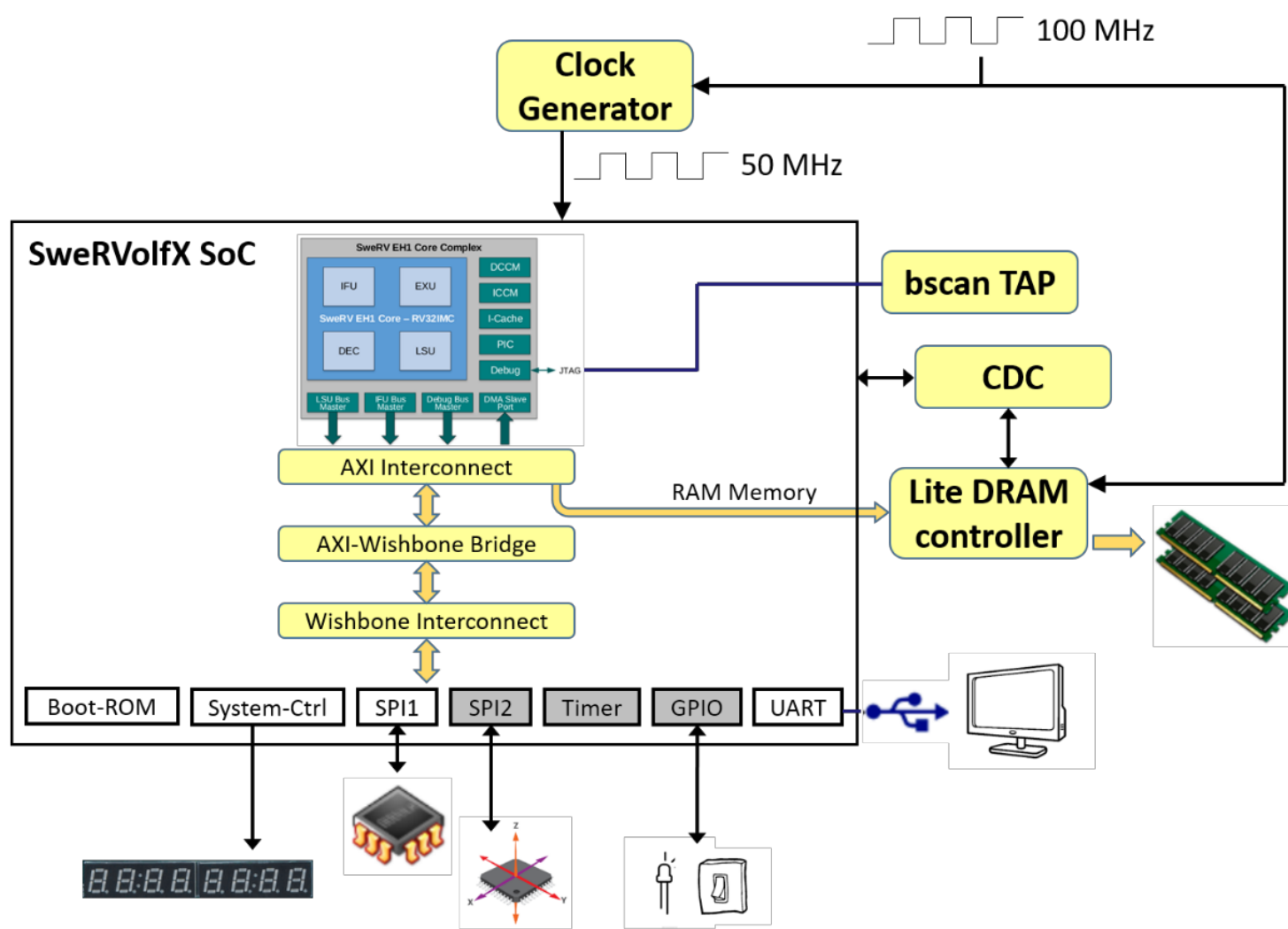


- Chips Alliance的開放原始碼晶片上系統（SoC）
- SweRVolf使用SweRV EH1核心。SweRVolf包括開機ROM、UART、系統控制器和SPI控制器（SPI1）
- SweRVolfX透過新增另一個SPI控制器（SPI2）、GPIO（通用輸入/輸出）、8位元7段顯示器和PTC（以紅色顯示）延伸SweRVolf。
- SweRV EH1核心使用AXI匯流排，而週邊設備使用Wishbone匯流排，因此SoC還具有AXI轉Wishbone橋接器

SweRVolfX記憶體對映

系統	位址
開機ROM	0x80000000 - 0x80000FFF
系統控制器	0x80001000 - 0x8000103F
SPI1	0x80001040 - 0x8000107F
SPI2	0x80001100 - 0x8000113F
計時器	0x80001200 - 0x8000123F
GPIO	0x80001400 - 0x8000143F
UART	0x80002000 - 0x80002FFF

RVfpgaNexys



- **RVfpgaNexys**：以Nexys A7 FPGA開發板為目標的 SweRVolfX SoC（新增若干周邊設備）：

- 核心和系統：

- SweRVolfX SoC
- Lite DRAM控制器
- 時脈產生器、鐘域和JTAG 連接埠的BSCAN邏輯

- Nexys A7 FPGA開發板上使用的 週邊設備：

- DDR2記憶體
- 採用USB連接的UART
- SPI快閃記憶體
- 16個LED和16個開關
- SPI加速計
- 8位7段顯示器

- SweRVolfX SoC還可以包括Verilog包裝函式以啟用模擬。
- **RVfpgaSim**是包裝在HDL模擬器所用測試平台中的SweRVolfX SoC。

RVfpga系統延伸

- RVfpga系統在實驗6-10中進一步延伸：
 - 新增**GPIO**控制器，與板上Nexys A7按鈕連接
 - 修改**7段顯示器**控制器
 - 新增**計時器**模組，以便使用板上三色**LED**
 - 新增外部中斷源

RVfpga實驗概述



RVfpga實驗概述

第1部分：實驗1-10

- Vivado專案和程式設計
- I/O系統

第2部分：實驗11-20

- RISC-V核心
- RISC-V記憶體系統
- RISC-V基準測試

所有實驗均包括使用和/或修改RVfpga系統的**練習**，透過動手設計來加深理解。

實驗內容

第1部分

編號	標題
1	建立Vivado專案
2	C語言程式設計
3	RISC-V組合語言
4	函數呼叫
5	影像處理：
6	I/O簡介
7	7段顯示器
8	計時器
9	中斷驅動I/O
10	序列匯流排

第2部分

編號	標題
11	SweRV EH1配置和效能監視
12	算術/邏輯指令：add指令
13	記憶體存取指令：lw和sw指令
14	結構冒險
15	資料冒險
16	控制冒險：分支指令
17	超標量執行
18	新增新功能（指令和計數器）
19	記憶體階層：指令快取
20	ICCM、DCCM和基準測試

RVfpga實驗1-10

顯示如何查看RVfpga系統原始程式碼（Verilog/SV）並確定其目標FPGA（實驗1）、如何編寫C程式和組合語言程式（實驗2-5）以及如何修改RVfpga系統以新增周邊設備（實驗6-10）。

- 實驗0：RVfpga實驗概述
- 實驗1：建立Vivado專案
- 實驗2：C語言程式設計
- 實驗3：RISC-V組合語言
- 實驗4：函數呼叫
- 實驗5：影像處理：C語言和組合語言
- 實驗6：I/O簡介
- 實驗7：7段顯示器
- 實驗8：計時器
- 實驗9：中斷驅動I/O
- 實驗10：序列匯流排

程式設計

I/O系統

RVfpga實驗1-5：Vivado專案和程式設計

- **實驗1：建立Vivado專案**：建立Vivado專案並確定RVfpgaNexys的目標FPGA開發板，然後在Verilator中模擬RVfpgaSim。
- **實驗2：C語言程式設計**：在PlatformIO中編寫一個C程式，然後在RVfpgaNexys/RVfpgaSim/Whisper上執行/除錯。此外，還介紹了Western Digital的開發板支援套件和平台支援套件（BSP和PSP），用於支援終端機列印等操作。
- **實驗3：RISC-V組合語言**：在PlatformIO中編寫一個RISC-V組合語言程式，然後在RVfpgaNexys/RVfpgaSim/Whisper上執行/除錯。
- **實驗4：函數呼叫**：函數呼叫、C函數庫和RISC-V呼叫慣例
- **實驗5：影像處理：C語言和組合語言**：將C程式碼嵌入組合語言程式碼

RVfpga實驗6-10：I/O和周邊設備

- **實驗6：I/O簡介**：記憶體對映I/O和RVfpga系統開放原始碼GPIO模組簡介。
- **實驗7：7段顯示器**：構建一個7段顯示器解碼器並將其整合到RVfpga系統中
- **實驗8：計時器**：瞭解和使用計時器與計時器控制器。
- **實驗9：中斷驅動I/O**：RVfpga系統中斷支援和中斷驅動I/O使用簡介。
- **實驗10：序列匯流排**：序列介面（SPI、I2C和UART）簡介。介紹如何使用採用SPI介面的板上加速計。

RVfpga實驗11-20：RISC-V核心

- **實驗11**：瞭解SweRV EH1組態、核心結構和效能監視。
- **實驗12、13和16**：檢查管線中的指令流動（算術/邏輯、記憶體、跳轉和分支）。
- **實驗14-16**：瞭解冒險及其處理方法
- **實驗16**：瞭解和修改分支預測器
- **實驗17**：探究超標量執行。
- **實驗18**：新增新指令和硬體計數器。
- **實驗19**：瞭解記憶體階層和I\$。
- **實驗20**：啟用ICCM和DCCM（指令和資料緊密耦合記憶體）並使用基準測試來比較效能。

RVfpga受眾和過往成績

- 目標受眾
 - 電氣工程、電腦科學或電腦工程專業的本科生
 - 有興趣學習RISC-V架構的學者和行業專業人士
- **Imagination大學計劃（IUP）** 過往成績：開展了MIPSfpga計劃：
 - 2015年4月推出
 - 800所大學參與
 - 榮獲2015年歐洲Elektra最佳教育支持獎

RVfpga 快速入門指南

快速入門指南概述

- 安裝VSCode和**PlatformIO**
- 在**RVfpgaNexys**上執行範例程式

安裝PlatformIO和VSCode

- 安裝**VSCode**
 - <https://code.visualstudio.com/Download>
 - 對於Ubuntu和macOS，需安裝**Python**（Windows不需要該步驟）
- 在VSCode中安裝**PlatformIO**延伸模組
- 安裝Nexys A7開發板驅動程式（請參閱RVfpga入門指南說明）

將RVfpgaNexys下載到開發板並執行程式

- 在**PlatformIO**中：

- 開啟程式範例（將開關的值寫入LED）。程式位置：
[RVfpgaPath]\RVfpga\examples\LedsSwitches_C-Lang
- 更新PlatformIO初始化檔案（platformio.ini）中**RVfpgaNexys bit**檔案的目錄位置 – 即，將以下行新增到platformio.ini中：
board_build.bitstream_file = [RVfpgaPath]/RVfpga/src/rvfpga.bit
- 將**RVfpgaNexys**下載到Nexys A7開發板（「Project Tasks → env:swervolf_nexys → Platform → Upload Bitstream」（專案任務 → env:swervolf_nexys → 平台 → 上傳位元流））
- 透過按下「Run/Debug」（執行/除錯）按鈕在**RVfpgaNexys**上編譯、下載和執行程式：

[RVfpgaPath]是電腦上**RVfpga**資料夾的位置。該資料夾隨Imagination大學計劃的RVfpga套件一起提供。

Rvfpga實驗 說明

實驗1： Vivado專案



RVfpga實驗1：RVfpga Vivado專案

- **Vivado**是一款Xilinx工具，用於查看、修改和合成RVfpga系統的原始（Verilog）程式碼。
- RVfpga系統的原始程式碼位置：
[RVfpgaPath]/RVfpga/src
- 建立一個包含RVfpga系統原始程式碼的**Vivado專案**。合成以Nexys A7開發板為目標的RVfpgaNexys，並建立一個包含將FPGA配置為RVfpgaNexys的資訊的**bit檔**（也稱為位元串流檔）。
- 也可以使用**Verilator**（一款HDL模擬器）來模擬RVfpga系統的原始程式碼和檢查內部訊號（有關如何使用Verilator的說明，請參閱RVfpga入門指南）。
- RVfpga實驗6-10中將廣泛使用Vivado和Verilator來修改和模擬RVfpga系統。

實驗2： C語言程式設計



RVfpga實驗2：C語言程式設計

- 建立**PlatformIO**專案
- 將**C**程式範例新增到專案
- 將**RVfpgaNexys**下載到Nexys A7開發板上
- 將**C**程式下載到RVfpgaNexys中並執行/除錯該程式
- 在實驗結束時完成部分或全部練習
- 請記住，也可以在Verilator（使用**RVfpgaSim**）或**Whisper**中模擬該程式。

RVfpga實驗2：C程式範例

```
// memory-mapped I/O addresses
#define GPIO_SWs      0x80001400
#define GPIO_LEDs     0x80001404
#define GPIO_INOUT    0x80001408

#define READ_GPIO(dir) (*(volatile unsigned *)dir)
#define WRITE_GPIO(dir, value) { (*(volatile unsigned *)dir) = (value); }

int main ( void )
{
    int En_Value=0xFFFF, switches_value;

    WRITE_GPIO(GPIO_INOUT, En_Value);

    while (1) {
        switches_value = READ_GPIO(GPIO_SWs);    // read value on switches
        switches_value = switches_value >> 16;   // shift into lower 16 bits
        WRITE_GPIO(GPIO_LEDs, switches_value);   // display switch value on LEDs
    }

    return(0);
}
```

該程式將開關的值寫入LED。

RVfpga實驗2：記憶體映射I/O位址

裝置	記憶體映射I/O位址
開關（Nexys A7開發板上為16個）	0x80001400（高16位元）
LED（Nexys A7開發板上為16個）	0x80001404（低16位元）
GPIO的輸入/輸出（1 = 輸出，0 = 輸入）	0x80001408

RVfpga實驗2：Western Digital的BSP和PSP

- Western Digital提供：
 - **PSP**：處理器支援套件
 - **BSP**：開發板支援套件
- 這兩個支援套件為給定的處理器（**SweRV EH1**核心）和開發板（**Nexys A7 FPGA**開發板）提供了常用的函數。
 - **範例**：`printfNexys`（類似C語言的`printf`函數）

RVfpga實驗2：使用UART列印到終端機

```
#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>
#define DELAY 10000000

int main(void) {
    int i, j = 0;

    // Initialize UART
    uartInit();
    while (1) {
        printfNexys("Hello RVfpga users! Iteration: %d\n", j);
        for (i=0; i < DELAY; i++) ; // delay between printf's
        j++;
    }
}
```

- 將下行新增到**platform.ini**檔：
monitor_speed = 115200
- 程式開始執行後，通過按下視窗底部的以下按鈕開啟**PlatformIO終端機**：



實驗3： RISC-V組合語言



RVfpga實驗3：RISC-V組合語言

- RISC-V組合語言概述
- 建立**PlatformIO**專案
- 將**RISC-V**組合語言程式範例新增到專案
- 將**RVfpgaNexys**下載到Nexys A7開發板上
- 將**RISC-V**組合語言程式下載到RVfpga中並執行/偵錯該程式
- 在實驗結束時完成部分或全部練習
- 請記住，也可以在Verilator（使用**RVfpgaSim**）或**Whisper**中模擬該程式。

RVfpga實驗3：RISC-V組合語言指令

常用的RISC-V組合語言指令/虛擬指令

RISC-V組合語言	說明	操作
add s0, s1, s2	加法	$s0 = s1 + s2$
sub s0, s1, s2	減法	$s0 = s1 - s2$
addi t3, t1, -10	加立即數	$t3 = t1 - 10$
mul t0, t2, t3	32位元乘法	$t0 = t2 * t3$
div s9, t5, t6	除法	$t9 = t5 / t6$
rem s4, s1, s2	求餘數	$s4 = s1 \% s2$
and t0, t1, t2	按位元AND	$t0 = t1 \& t2$
or t0, t1, t5	按位元OR	$t0 = t1 t5$
xor s3, s4, s5	按位元XOR	$s3 = s4 \wedge s5$
andi t1, t2, 0xFFB	按位元AND（立即數）	$t1 = t2 \& 0xFFFFFBB$
ori t0, t1, 0x2C	按位元OR（立即數）	$t0 = t1 0x2C$
xori s3, s4, 0xABC	按位元XOR（立即數）	$s3 = s4 \wedge 0xFFFFFABC$
sll t0, t1, t2	邏輯左移	$t0 = t1 \ll t2$
srl t0, t1, t5	邏輯右移	$t0 = t1 \gg t5$
sra s3, s4, s5	算術右移	$s3 = s4 \ggg s5$
slli t1, t2, 30	邏輯左移（立即數）	$t1 = t2 \ll 30$
srlt t0, t1, 5	邏輯右移（立即數）	$t0 = t1 \gg 5$
srair s3, s4, 31	算術右移（立即數）	$s3 = s4 \ggg 31$

RVfpga實驗3：RISC-V組合語言指令

常用的RISC-V組合語言指令/虛擬指令（續）

RISC-V組合語言	說明	操作
lw s7, 0x2C(t1)	載入字	$s7 = \text{memory}[t1+0x2C]$
lh s5, 0x5A(s3)	載入半字	$s5 = \text{SignExt}(\text{memory}[s3+0x5A]_{15:0})$
lb s1, -3(t4)	載入位元組	$s1 = \text{SignExt}(\text{memory}[t4-3]_{7:0})$
sw t2, 0x7C(t1)	儲存字	$\text{memory}[t1+0x7C] = t2$
sh t3, 22(s3)	儲存半字	$\text{memory}[s3+22]_{15:0} = t3_{15:0}$
sb t4, 5(s4)	儲存位元組	$\text{memory}[s4+5]_{7:0} = t4_{7:0}$
beq s1, s2, L1	如相等則分支	if ($s1 == s2$), PC = L1
bne t3, t4, Loop	如不相等則分支	if ($s1 \neq s2$), PC = Loop
blt t4, t5, L3	如小於則分支	if ($t4 < t5$), PC = L3
bge s8, s9, Done	如不相等則分支	if ($s8 \geq s9$), PC = Done
li s1, 0xABCDEF12	載入立即數	$s1 = 0xABCDEF12$
la s1, A	載入位址	$s1 = \text{儲存變數A的記憶體位址}$
nop	無操作	無操作
mv s3, s7	移動	$s3 = s7$
not t1, t2	非（求反）	$t1 = \sim t2$
neg s1, s3	求補	$s1 = -s3$
j Label	跳轉	PC = 標籤
jal L7	跳轉和連結	PC = L7 ; ra = PC + 4
jr s1	跳轉暫存器	PC = s1

RVfpga實驗3：RISC-V暫存器

32個32位元暫存器

名稱	暫存器編號	用途
zero	x0	常數值0
ra	x1	返回位址
sp	x2	堆疊指標
gp	x3	全域指標
tp	x4	執行緒指標
t0-2	x5-7	臨時變數
s0/fp	x8	儲存變數/架構指標
s1	x9	儲存變數
a0-1	x10-11	函數引數/傳回值
a2-7	x12-17	函數引數
s2-11	x18-27	儲存變數
t3-6	x28-31	臨時變數



RVfpga實驗3：RISC-V組合語言程式範例

```
• // memory-mapped I/O addresses
• # GPIO_SWs      = 0x80001400
• # GPIO_LEDs     = 0x80001404
• # GPIO_INOUT    = 0x80001408
•
• .globl main
• main:
•
• main:
•     li t0, 0x80001400    # base address of GPIO memory-mapped registers
•     li t1, 0xFFFF       # set direction of GPIOs
•                          # upper half = switches (inputs)    (=0)
•                          # lower half = outputs (LEDs)       (=1)
•     sw t1, 8(t0)         # GPIO_INOUT = 0xFFFF
•
• repeat:
•     lw  t1, 0(t0)        # read switches: t1 = GPIO_SWs
•     srli t1, t1, 16       # shift val to the right by 16 bits
•     sw  t1, 4(t0)        # write value to LEDs: GPIO_LEDs = t1
•     j   repeat           # repeat loop
```

該程式將開關的值寫入LED。



實驗4： 函數呼叫



RVfpga實驗4：函數呼叫

- 編寫具有**函數呼叫**的C程式
 - 函數也稱為**程序**
- 使用**C函數庫**
- RISC-V（程序）呼叫慣例

RVfpga實驗4：具有函數的程式範例

```
// memory-mapped I/O addresses
#define GPIO_SWs      0x80001400
#define GPIO_LEDs     0x80001404
#define GPIO_INOUT    0x80001408
#define READ_GPIO(dir) (*(volatile unsigned *)dir)
#define WRITE_GPIO(dir, value) { (*(volatile unsigned *)dir) = (value); }

void IOsetup();
unsigned int getSwitchVal();
void writeValtoLEDs(unsigned int val);

int main ( void ) {
    unsigned int switches_val;

    IOsetup();
    while (1) {
        switches_val = getSwitchVal();
        writeValtoLEDs(switches_val);
    }

    return(0);
}
```

RVfpga實驗4：具有函數的程式範例

```
void IOsetup()
{
    int En_Value=0xFFFF;
    WRITE_GPIO(GPIO_INOUT, En_Value);
}

unsigned int getSwitchVal()
{
    unsigned int val;

    val = READ_GPIO(GPIO_SWs);    // read value on switches
    val = val >> 16;    // shift into lower 16 bits

    return val;
}

void writeValtoLEDs(unsigned int val)
{
    WRITE_GPIO(GPIO_LEDs, val);    // display val on LEDs
}
```

RVfpga實驗4：C函數庫

- 函數庫
 - 常用函數的集合
 - 旨在方便使用常用函數（節省程式設計時間）
- C函數庫範例：
 - **math.h**（數學函數庫）：包括諸如**sqrt**（平方根）和**cos**（餘弦）等函數。
 - **stdio.h**（標準I/O函數庫）：包括將值列印到螢幕（**printf**）、從使用者讀取值（**scanf**）等函數。
 - **stdlib.h**（標準函數庫）：包括用於產生隨機數（**rand**）的函數。
 - 其他...（google C函數庫）

RVfpga實驗4：使用C函數庫的程式範例

```
#include <stdlib.h>
```

```
...
```

```
int main(void) {  
    unsigned int val;  
    volatile unsigned int i;  
  
    IOsetup();  
    while (1) {  
        val = rand() % 65536;  
        writeValtoLEDs(val);  
        for (i = 0; i < DELAY; i++)  
            ;  
    }  
    return(0);  
}
```

此程式將0到65535之間的隨機數寫入LED。

RVfpga實驗4：RISC-V呼叫慣例

- 呼叫函數

`jal function_label`

- 從函數返回

`jr ra`

- 引數

- 置於暫存器a0-a7中

- 傳回值

- 置於暫存器a0中

RVfpga實驗4：RISC-V呼叫慣例範例

C程式碼

```
int main() {  
    ...  
    int y = y + func1(1, 2, 3)  
    y++;  
    ...  
}  
  
int func1(int a, int b, int c) {  
    int sum;  
    sum = a + b + c;  
    return sum;  
}
```

RISC-V組合語言

```
# y is in s0  
main:  
    ...  
    addi a0, zero, 1    # put values in argument registers  
    addi a1, zero, 2  
    addi a2, zero, 3  
    jal  func1          # call function func1  
    add  s0, s0, a0      # y = y + return value  
    addi s0, s0, 1      # y = y++  
    ...  
  
# sum is in s0  
func1:  
    add s0, a0, a1      # sum = a + b  
    add s0, s0, a2      # sum = a + b + c  
    addi a0, s0, 0      # return value = sum  
    jr   ra             # return
```

RVfpga實驗4：堆疊

- 記憶體中用於儲存暫存器值的**暫存空間**
- 堆疊指標（`sp`）用於儲存堆疊頂端的位址
- 記憶體中的**堆疊自上而下儲存**。因此，如果要在堆疊上留出**4個字**（**16位元組**）空間，應使用以下程式碼：

```
addi sp, sp, -16
```

- **兩類暫存器：**
 - **保留暫存器**：函數呼叫過程中必須**保留**暫存器內容（即，函數呼叫前後暫存器值保持不變）
 - **非保留暫存器**：函數呼叫過程中不得**保留**暫存器內容（即，函數呼叫前後暫存器值無需保持不變）
 - 儲存暫存器（`s0-s11`）、返回位址暫存器（`ra`）和堆疊指標（`sp`）均為**保留暫存器**。所有其他暫存器均為非保留暫存器。



RVfpga實驗4：保留/非保留暫存器

名稱	暫存器編號	用途	保留
zero	x0	常數值0	-
ra	x1	返回位址	是
sp	x2	堆疊指標	是
gp	x3	全域指標	-
tp	x4	執行緒指標	-
t0-2	x5-7	臨時變數	否
s0/fp	x8	儲存變數/架構指標	是
s1	x9	儲存變數	是
a0-1	x10-11	函數引數/傳回值	否
a2-7	x12-17	函數引數	否
s2-11	x18-27	儲存變數	是
t3-6	x28-31	臨時變數	否



RVfpga實驗4：堆疊 - 修改後的組合語言程式碼

C程式碼

```
int main() {  
    ...  
    int y = y + func1(1, 2, 3)  
    y++;  
    ...  
}  
  
int func1(int a, int b, int c) {  
    int sum;  
  
    sum = a + b + c;  
    return sum;  
}
```

RISC-V組合語言

```
# y is in s0  
main: ...  
    addi a0, zero, 1 # put values in argument registers  
    addi a1, zero, 2  
    addi a2, zero, 3  
    jal  func1        # call function func1  
    add  s0, s0, a0    # y = y + return value  
    addi s0, s0, 1     # y = y++  
    ...  
  
# sum is in s0  
func1: addi sp, sp, -4 # make room on stack  
       sw  s0, 0(sp) # save s0 on stack  
    add s0, a0, a1 # sum = a + b  
    add s0, s0, a2 # sum = a + b + c  
    addi a0, s0, 0 # return value = sum  
    lw  s0, 0(sp) # restore s0 from stack  
    addi sp, sp, 4 # restore stack pointer  
    jr  ra        # return
```

實驗5： C語言和組合語言

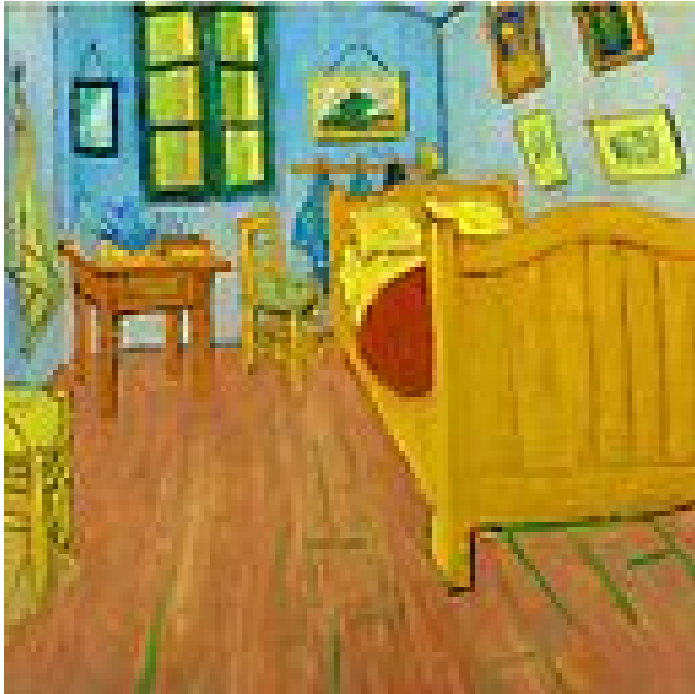


RVfpga實驗5：結合使用C語言和組合語言

- 範例：影像處理程式
- 有些函數用C語言編寫，有些則用組合語言編寫

RVfpga實驗5：影像處理程式

- 將彩色圖轉換為灰階



RVfpga實驗5：影像處理程式

- 每個像素儲存為三種8位位顏色：**R** = 紅色，**G** = 綠色，**B** = 藍色
- 可以通過變更**R**、**G**和**B**值來建立任何顏色
- 要將圖像轉換為8位元灰階圖（grey），每個像素應進行如下變換：

$$\text{grey} = (306 * \text{R} + 601 * \text{G} + 117 * \text{B}) \gg 10$$

- RGB權重加起來為1024（ $306 + 601 + 117 = 1024$ ），因此若要返回8位元範圍（0-255），結果應除以1024（即右移10位： $\gg 10$ ）
- 有關演算法的詳細資訊，請參閱：

<https://www.mathworks.com/help/matlab/ref/rgb2gray.html>

RVfpga實驗5：組合語言函數

`.globl ColourToGrey_Pixel` ← `.globl`使ColourToGrey_Pixel函數對專案中的所有檔案可見
`.text`

ColourToGrey_Pixel:

```
li x28, 306          # a0 = R * 306
mul a0, a0, x28
li x28, 601          # a1 = G * 601
mul a1, a1, x28
li x28, 117          # a2 = B * 117
mul a2, a2, x28
add a0, a0, a1        # grey = a0 + a1 + a2
add a0, a0, a2
srl a0, a0, 10        # grey = grey / 1024
ret                  # return
.end
```

$$\text{grey} = (306 * \text{R} + 601 * \text{G} + 117 * \text{B}) \gg 10$$

RVfpga實驗5：結構和陣列

```
typedef struct {
    unsigned char R;
    unsigned char G;
    unsigned char B;
} RGB;

extern unsigned char VanGogh_128x128[]; // 1D array of individual RGB values
RGB ColourImage[N][M];                // 2D array of RGB struct (colour image)
unsigned char GreyImage[N][M];        // 2D array of greyscale image

// VanGogh_128.c
unsigned char VanGogh_128x128[] = {
    157, // R (pixel [0][0])
    182, // G (pixel [0][0])
    161, // B (pixel [0][0])
    171, // R (pixel [0][1])
    195, // G (pixel [0][1])
    173, // B (pixel [0][1])
    173, // R (pixel [0][2])
    ...
}
```

RVfpga實驗5：主函數

```
int main(void) {
    // Create an N x M matrix using the input image
    initColourImage(ColourImage);

    // Transform Colour Image to Grey Image
    ColourToGrey(ColourImage, GreyImage);
    ...
}

void ColourToGrey(RGB Colour[N][M], unsigned char Grey[N][M]) {
    int i, j;

    for (i=0; i<N; i++)
        for (j=0; j<M; j++)
            Grey[i][j] = ColourToGrey_Pixel(Colour[i][j].R, Colour[i][j].G,
                                                Colour[i][j].B);
}
```

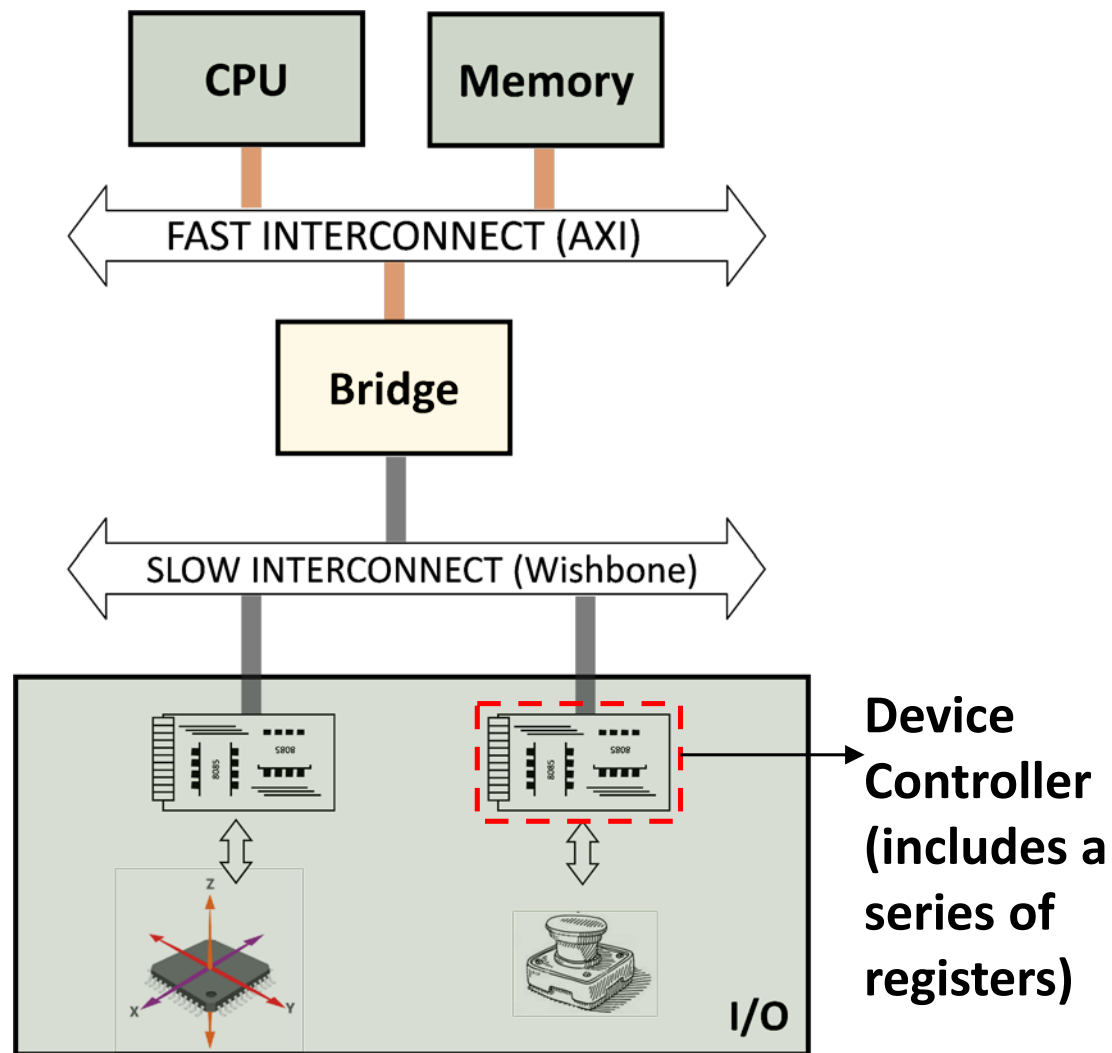
實驗6： I/O簡介



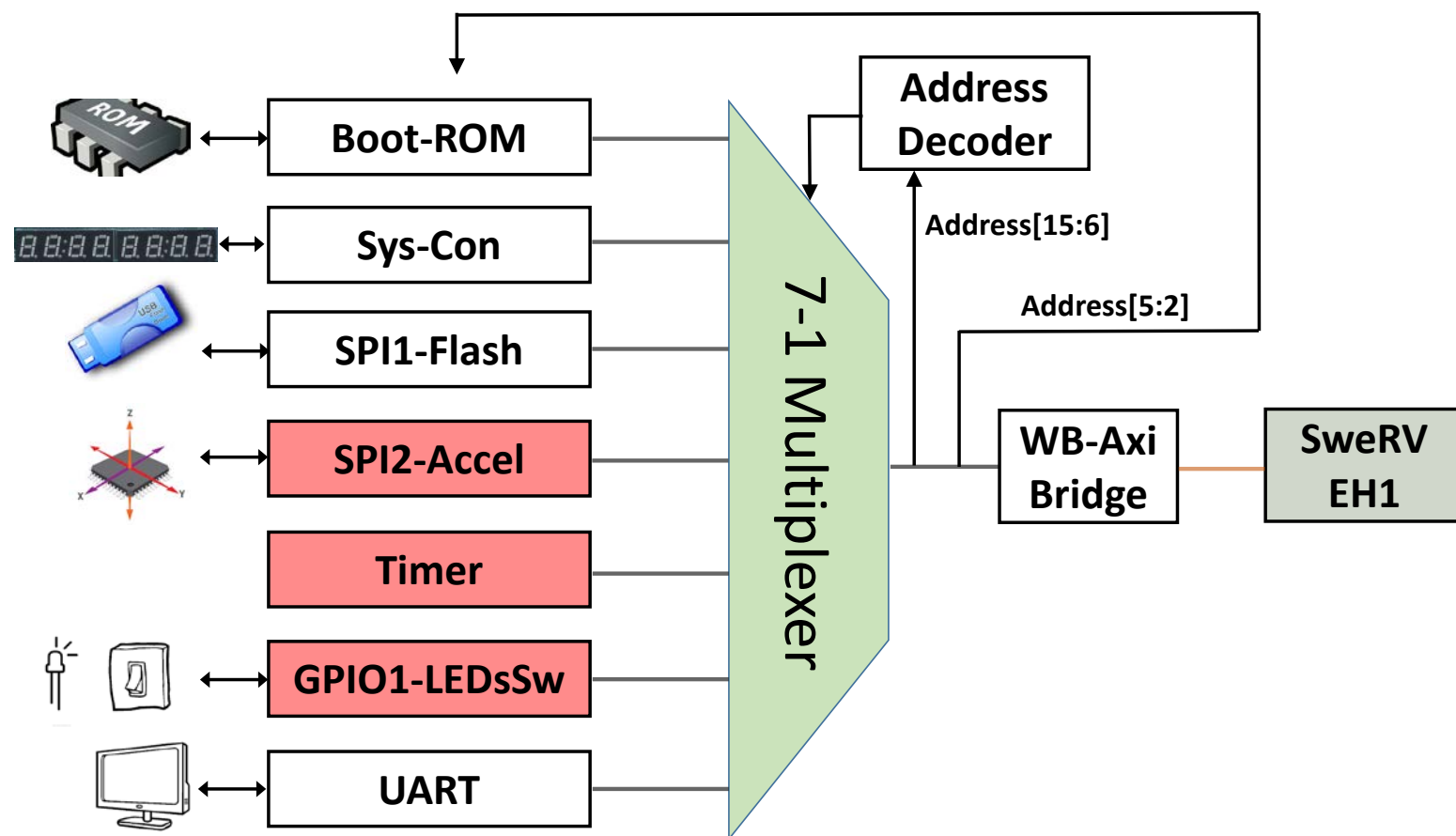
RVfpga實驗6：I/O簡介

- 輸入/輸出（I/O）系統 - 也稱為週邊設備
- 通用I/O（GPIO）
- GPIO控制器

RVfpga實驗6：具有I/O的通用處理器



RVfpga實驗6：具有I/O的處理器



週邊設備

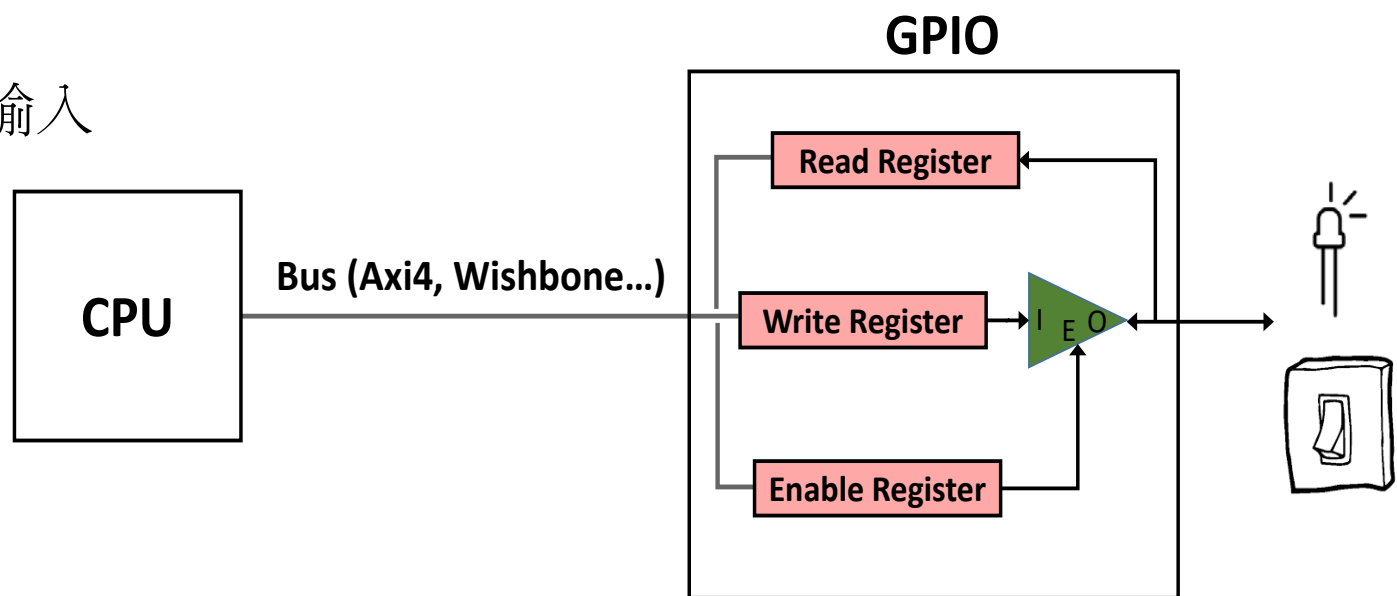
- **SweRVolfX** 周邊設備：

- 開機ROM
- 系統控制器
- SPI1快閃記憶體
- UART
- GPIO LED和開關
- 計時器
- SPI2加速計
- 7段顯示器（在系統控制器內：Sys-Con）

RVfpga實驗6：通用I/O（GPIO）

- 通用I/O：
 - 允許處理器讀取/寫入連接到週邊設備（例如，開關和LED）的引腳
 - 每個引腳均可配置為輸入或輸出（使用三態）
- 三個記憶體映射暫存器：
 - 讀取暫存器：從針腳讀取的值
 - 寫入暫存器：寫入針腳的值
 - 啟用暫存器：1 = 輸出，0 = 輸入

週邊設備



RVfpga實驗6：記憶體映射暫存器

暫存器	記憶體映射位址
讀取暫存器	0x80001400
寫入暫存器	0x80001404
啟用暫存器	0x80001408

- 將**GPIO**的位元**15:0**配置為輸出，**bit 31:16**配置為輸入：

```
li t0, 0x80001400    # t0 = 0x80001400
li t1, 0xFFFF        # 1 = output, 0 = input
sw t1, 8(t0)         # [15:0] = outputs, [31:16] = inputs
```

- 讀取**I/O**：

```
lw t2, 0(t0)         # t2 = value of GPIO pins
```

- 寫入**I/O**：

```
sw t3, 4(t0)         # GPIO pins = t3
```

RVfpga實驗6：SweRVofX GPIO模組

- **OpenCores的GPIO模組**

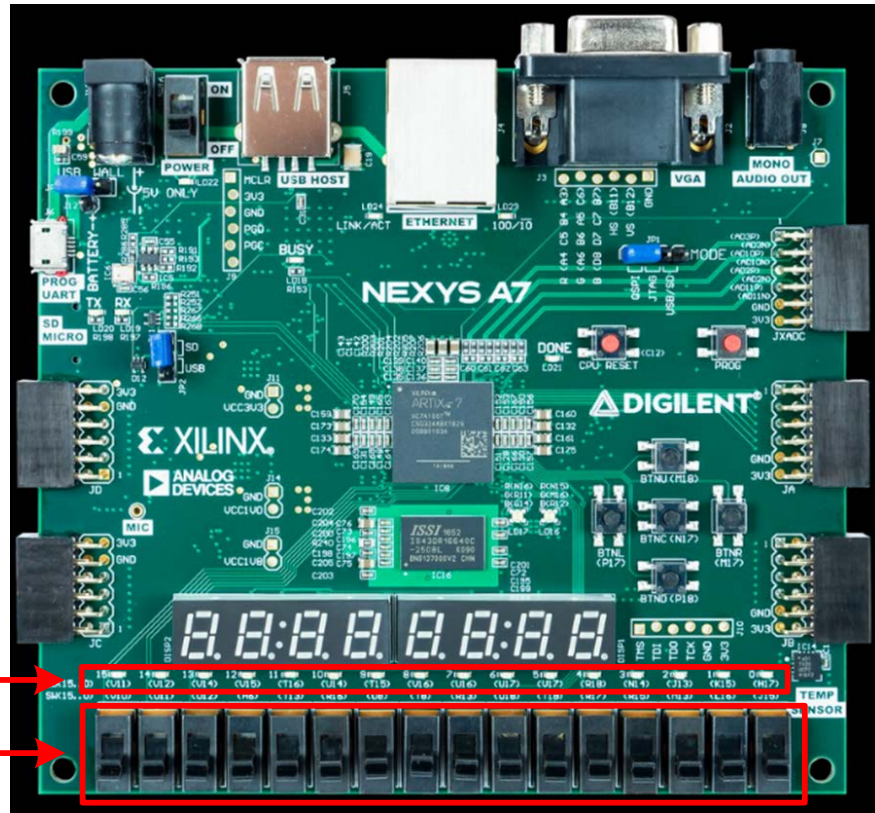
<https://opencores.org/projects/gpio>

- **最多允許32個GPIO引腳**

- 所有引腳均可單獨配置為輸入（啟用 = 0）或輸出（啟用 = 1）
- 整個程式的組態都可以變更

暫存器	記憶體映射位址
讀取暫存器	0x80001400
寫入暫存器	0x80001404
啟用暫存器	0x80001408

RVfpga實驗6：記憶體映射暫存器



開發板圖片來源：<https://reference.digilentinc.com/>

將LED和開關映射到GPIO引腳：

- LED：針腳[15:0]（處理器的輸出）
- 開關：針腳[31:16]（處理器的輸入）

配置GPIO：

- 啟用暫存器 = **0x0000FFFF**（1 = 輸出，0 = 輸入）

```
li t0, 0x80001400
li t1, 0xFFFF
sw t1, 8(t0) # Enable Register = 0xFFFF
```

寫入LED：

- 將[15:0]中的值寫入位址0x80001404

```
sw t3, 4(t0) # LEDs = [t3]15:0
```

讀取開關：

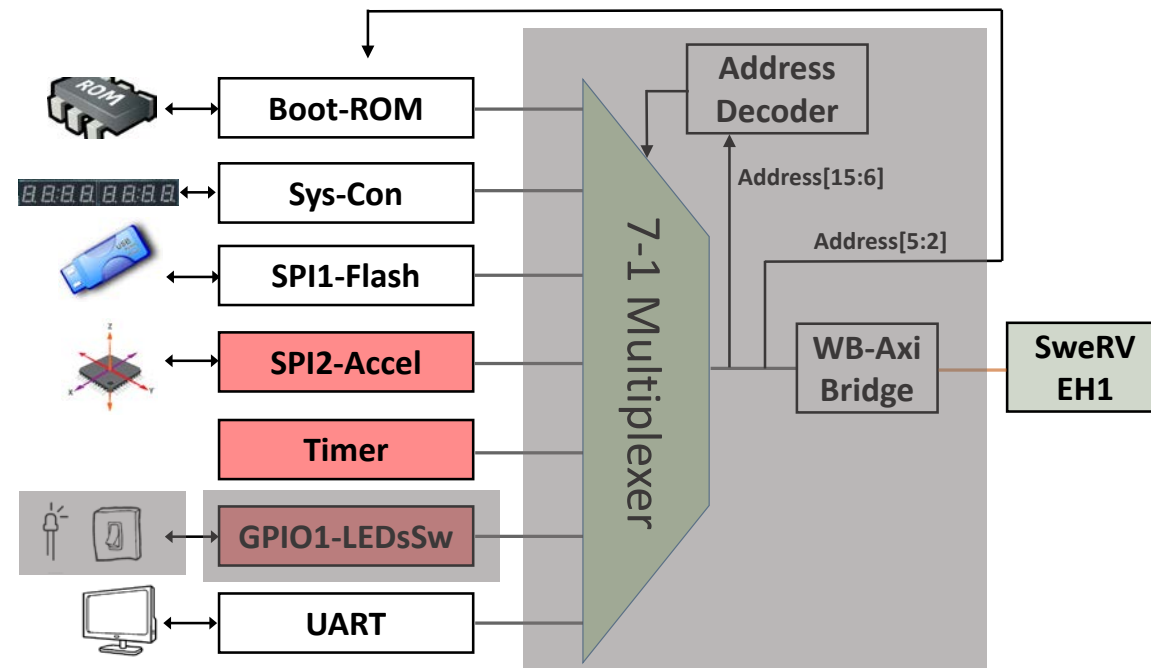
- 從位址0x80001400讀取位元 [31:16]中的開關值
- 右移16位元，將開關值置於低16位元中

```
lw t5, 0(t0) # [t5]31:16 = switch values
srli t5, t5, 16 # [t5]15:0 = switch values
```

RVfpga實驗6：GPIO低階實作

- 主要分為3個部分

- RVfpgaNexys與板上LED/開關的外部連接（左側陰影區域）
- SweRVolfX中整合的GPIO模組（中間陰影區域）
- GPIO和SweRV EH1之間的連接（右側陰影區域）



RVfpga實驗6：外部連接

檔案rvfpganexys.xdc：定義i_sw[15:0]與開發板上開關的連接，以及o_led[15:0]與開發板上LED的連接

```
26 set_property -dict { PACKAGE_PIN J15 IOSTANDARD LVCMOS33 } [get_ports { i_sw[0] }]
27 set_property -dict { PACKAGE_PIN L16 IOSTANDARD LVCMOS33 } [get_ports { i_sw[1] }]
28 set_property -dict { PACKAGE_PIN M13 IOSTANDARD LVCMOS33 } [get_ports { i_sw[2] }]
29 set_property -dict { PACKAGE_PIN R15 IOSTANDARD LVCMOS33 } [get_ports { i_sw[3] }]
30 set_property -dict { PACKAGE_PIN R17 IOSTANDARD LVCMOS33 } [get_ports { i_sw[4] }]
31 set_property -dict { PACKAGE_PIN T18 IOSTANDARD LVCMOS33 } [get_ports { i_sw[5] }]
32 set_property -dict { PACKAGE_PIN U18 IOSTANDARD LVCMOS33 } [get_ports { i_sw[6] }]
33 set_property -dict { PACKAGE_PIN R13 IOSTANDARD LVCMOS33 } [get_ports { i_sw[7] }]
34 set_property -dict { PACKAGE_PIN T8 IOSTANDARD LVCMOS18 } [get_ports { i_sw[8] }]
35 set_property -dict { PACKAGE_PIN U8 IOSTANDARD LVCMOS18 } [get_ports { i_sw[9] }]
36 set_property -dict { PACKAGE_PIN R16 IOSTANDARD LVCMOS33 } [get_ports { i_sw[10] }]
37 set_property -dict { PACKAGE_PIN T13 IOSTANDARD LVCMOS33 } [get_ports { i_sw[11] }]
38 set_property -dict { PACKAGE_PIN H6 IOSTANDARD LVCMOS33 } [get_ports { i_sw[12] }]
39 set_property -dict { PACKAGE_PIN U12 IOSTANDARD LVCMOS33 } [get_ports { i_sw[13] }]
40 set_property -dict { PACKAGE_PIN U11 IOSTANDARD LVCMOS33 } [get_ports { i_sw[14] }]
41 set_property -dict { PACKAGE_PIN V10 IOSTANDARD LVCMOS33 } [get_ports { i_sw[15] }]
42
43 set_property -dict { PACKAGE_PIN H17 IOSTANDARD LVCMOS33 } [get_ports { o_led[0] }]
44 set_property -dict { PACKAGE_PIN K15 IOSTANDARD LVCMOS33 } [get_ports { o_led[1] }]
45 set_property -dict { PACKAGE_PIN J13 IOSTANDARD LVCMOS33 } [get_ports { o_led[2] }]
46 set_property -dict { PACKAGE_PIN N14 IOSTANDARD LVCMOS33 } [get_ports { o_led[3] }]
47 set_property -dict { PACKAGE_PIN R18 IOSTANDARD LVCMOS33 } [get_ports { o_led[4] }]
48 set_property -dict { PACKAGE_PIN V17 IOSTANDARD LVCMOS33 } [get_ports { o_led[5] }]
49 set_property -dict { PACKAGE_PIN U17 IOSTANDARD LVCMOS33 } [get_ports { o_led[6] }]
50 set_property -dict { PACKAGE_PIN U16 IOSTANDARD LVCMOS33 } [get_ports { o_led[7] }]
51 set_property -dict { PACKAGE_PIN V16 IOSTANDARD LVCMOS33 } [get_ports { o_led[8] }]
52 set_property -dict { PACKAGE_PIN T15 IOSTANDARD LVCMOS33 } [get_ports { o_led[9] }]
53 set_property -dict { PACKAGE_PIN U14 IOSTANDARD LVCMOS33 } [get_ports { o_led[10] }]
54 set_property -dict { PACKAGE_PIN T16 IOSTANDARD LVCMOS33 } [get_ports { o_led[11] }]
55 set_property -dict { PACKAGE_PIN V15 IOSTANDARD LVCMOS33 } [get_ports { o_led[12] }]
56 set_property -dict { PACKAGE_PIN V14 IOSTANDARD LVCMOS33 } [get_ports { o_led[13] }]
57 set_property -dict { PACKAGE_PIN V12 IOSTANDARD LVCMOS33 } [get_ports { o_led[14] }]
58 set_property -dict { PACKAGE_PIN V11 IOSTANDARD LVCMOS33 } [get_ports { o_led[15] }]
```



RVfpga實驗6：RVfpga整合

swervolf_core.v檔：三態緩衝區和GPIO模組實例化

```
bidirec gpio0 (.oe(en_gpio[0]), .inp(o_gpio[0]), .outp(i_gpio[0]), .bidir(io_data[0]));
bidirec gpio1 (.oe(en_gpio[1]), .inp(o_gpio[1]), .outp(i_gpio[1]), .bidir(io_data[1]));
bidirec gpio2 (.oe(en_gpio[2]), .inp(o_gpio[2]), .outp(i_gpio[2]), .bidir(io_data[2]));
bidirec gpio3 (.oe(en_gpio[3]), .inp(o_gpio[3]), .outp(i_gpio[3]), .bidir(io_data[3]));
bidirec gpio4 (.oe(en_gpio[4]), .inp(o_gpio[4]), .outp(i_gpio[4]), .bidir(io_data[4]));
bidirec gpio5 (.oe(en_gpio[5]), .inp(o_gpio[5]), .outp(i_gpio[5]), .bidir(io_data[5]));
bidirec gpio6 (.oe(en_gpio[6]), .inp(o_gpio[6]), .outp(i_gpio[6]), .bidir(io_data[6]));
bidirec gpio7 (.oe(en_gpio[7]), .inp(o_gpio[7]), .outp(i_gpio[7]), .bidir(io_data[7]));
bidirec gpio8 (.oe(en_gpio[8]), .inp(o_gpio[8]), .outp(i_gpio[8]), .bidir(io_data[8]));
bidirec gpio9 (.oe(en_gpio[9]), .inp(o_gpio[9]), .outp(i_gpio[9]), .bidir(io_data[9]));
bidirec gpio10 (.oe(en_gpio[10]), .inp(o_gpio[10]), .outp(i_gpio[10]), .bidir(io_data[10]));
bidirec gpio11 (.oe(en_gpio[11]), .inp(o_gpio[11]), .outp(i_gpio[11]), .bidir(io_data[11]));
bidirec gpio12 (.oe(en_gpio[12]), .inp(o_gpio[12]), .outp(i_gpio[12]), .bidir(io_data[12]));
bidirec gpio13 (.oe(en_gpio[13]), .inp(o_gpio[13]), .outp(i_gpio[13]), .bidir(io_data[13]));
bidirec gpio14 (.oe(en_gpio[14]), .inp(o_gpio[14]), .outp(i_gpio[14]), .bidir(io_data[14]));
bidirec gpio15 (.oe(en_gpio[15]), .inp(o_gpio[15]), .outp(i_gpio[15]), .bidir(io_data[15]));
bidirec gpio16 (.oe(en_gpio[16]), .inp(o_gpio[16]), .outp(i_gpio[16]), .bidir(io_data[16]));
bidirec gpio17 (.oe(en_gpio[17]), .inp(o_gpio[17]), .outp(i_gpio[17]), .bidir(io_data[17]));
bidirec gpio18 (.oe(en_gpio[18]), .inp(o_gpio[18]), .outp(i_gpio[18]), .bidir(io_data[18]));
bidirec gpio19 (.oe(en_gpio[19]), .inp(o_gpio[19]), .outp(i_gpio[19]), .bidir(io_data[19]));
bidirec gpio20 (.oe(en_gpio[20]), .inp(o_gpio[20]), .outp(i_gpio[20]), .bidir(io_data[20]));
bidirec gpio21 (.oe(en_gpio[21]), .inp(o_gpio[21]), .outp(i_gpio[21]), .bidir(io_data[21]));
bidirec gpio22 (.oe(en_gpio[22]), .inp(o_gpio[22]), .outp(i_gpio[22]), .bidir(io_data[22]));
bidirec gpio23 (.oe(en_gpio[23]), .inp(o_gpio[23]), .outp(i_gpio[23]), .bidir(io_data[23]));
bidirec gpio24 (.oe(en_gpio[24]), .inp(o_gpio[24]), .outp(i_gpio[24]), .bidir(io_data[24]));
bidirec gpio25 (.oe(en_gpio[25]), .inp(o_gpio[25]), .outp(i_gpio[25]), .bidir(io_data[25]));
bidirec gpio26 (.oe(en_gpio[26]), .inp(o_gpio[26]), .outp(i_gpio[26]), .bidir(io_data[26]));
bidirec gpio27 (.oe(en_gpio[27]), .inp(o_gpio[27]), .outp(i_gpio[27]), .bidir(io_data[27]));
bidirec gpio28 (.oe(en_gpio[28]), .inp(o_gpio[28]), .outp(i_gpio[28]), .bidir(io_data[28]));
bidirec gpio29 (.oe(en_gpio[29]), .inp(o_gpio[29]), .outp(i_gpio[29]), .bidir(io_data[29]));
bidirec gpio30 (.oe(en_gpio[30]), .inp(o_gpio[30]), .outp(i_gpio[30]), .bidir(io_data[30]));
bidirec gpio31 (.oe(en_gpio[31]), .inp(o_gpio[31]), .outp(i_gpio[31]), .bidir(io_data[31]));
```

```
gpio_top gpio_module(
    .wb_clk_i      (clk),
    .wb_rst_i      (wb_rst),
    .wb_cyc_i      (wb_m2s_gpio_cyc),
    .wb_adr_i      ({2'b0,wb_m2s_gpio_adr[5:2],2'b0}),
    .wb_dat_i      (wb_m2s_gpio_dat),
    .wb_sel_i      (4'b1111),
    .wb_we_i       (wb_m2s_gpio_we),
    .wb_stb_i      (wb_m2s_gpio_stb),
    .wb_dat_o      (wb_s2m_gpio_dat),
    .wb_ack_o      (wb_s2m_gpio_ack),
    .wb_err_o      (wb_s2m_gpio_err),
    .wb_inta_o     (gpio_irq),
    // External GPIO Interface
    .ext_pad_i     (i_gpio[31:0]),
    .ext_pad_o     (o_gpio[31:0]),
    .ext_padoe_o   (en_gpio));
```

RVfpga實驗6：與SweRV EH1的连接

wb_intercon.v檔：7-1 多工器實作

```
108 wb_mux
109     #(.num_slaves (7),
110       .MATCH_ADDR ({32'h00000000, 32'h00001000, 32'h00001040, 32'h00001100, 32'h00001200, 32'h00001400, 32'h00002000}),
111       .MATCH_MASK ({32'hffff000, 32'hfffffc0, 32'hfffffc0, 32'hfffffc0, 32'hfffffc0, 32'hfffffc0, 32'hffff000})),
112     wb_mux_io
113     (.wb_clk_i (wb_clk_i),
114      .wb_rst_i (wb_rst_i),
115      .wbm_adr_i (wb_io_adr_i),
116      .wbm_dat_i (wb_io_dat_i),
117      .wbm_sel_i (wb_io_sel_i),
118      .wbm_we_i (wb_io_we_i),
119      .wbm_cyc_i (wb_io_cyc_i),
120      .wbm_stb_i (wb_io_stb_i),
121      .wbm_cti_i (wb_io_cti_i),
122      .wbm_bte_i (wb_io_bte_i),
123      .wbm_dat_o (wb_io_dat_o),
124      .wbm_ack_o (wb_io_ack_o),
125      .wbm_err_o (wb_io_err_o),
126      .wbm_rty_o (wb_io_rty_o),
127      .wbs_adr_o ({wb_rom_adr_o, wb_sys_adr_o, wb_spi_flash_adr_o, wb_spi_accel_adr_o, wb_ptc_adr_o, wb_gpio_adr_o, wb_uart_adr_o}),
128      .wbs_dat_o ({wb_rom_dat_o, wb_sys_dat_o, wb_spi_flash_dat_o, wb_spi_accel_dat_o, wb_ptc_dat_o, wb_gpio_dat_o, wb_uart_dat_o}),
129      .wbs_sel_o ({wb_rom_sel_o, wb_sys_sel_o, wb_spi_flash_sel_o, wb_spi_accel_sel_o, wb_ptc_sel_o, wb_gpio_sel_o, wb_uart_sel_o}),
130      .wbs_we_o ({wb_rom_we_o, wb_sys_we_o, wb_spi_flash_we_o, wb_spi_accel_we_o, wb_ptc_we_o, wb_gpio_we_o, wb_uart_we_o}),
131      .wbs_cyc_o ({wb_rom_cyc_o, wb_sys_cyc_o, wb_spi_flash_cyc_o, wb_spi_accel_cyc_o, wb_ptc_cyc_o, wb_gpio_cyc_o, wb_uart_cyc_o}),
132      .wbs_stb_o ({wb_rom_stb_o, wb_sys_stb_o, wb_spi_flash_stb_o, wb_spi_accel_stb_o, wb_ptc_stb_o, wb_gpio_stb_o, wb_uart_stb_o}),
133      .wbs_cti_o ({wb_rom_cti_o, wb_sys_cti_o, wb_spi_flash_cti_o, wb_spi_accel_cti_o, wb_ptc_cti_o, wb_gpio_cti_o, wb_uart_cti_o}),
134      .wbs_bte_o ({wb_rom_bte_o, wb_sys_bte_o, wb_spi_flash_bte_o, wb_spi_accel_bte_o, wb_ptc_bte_o, wb_gpio_bte_o, wb_uart_bte_o}),
135      .wbs_dat_i ({wb_rom_dat_i, wb_sys_dat_i, wb_spi_flash_dat_i, wb_spi_accel_dat_i, wb_ptc_dat_i, wb_gpio_dat_i, wb_uart_dat_i}),
136      .wbs_ack_i ({wb_rom_ack_i, wb_sys_ack_i, wb_spi_flash_ack_i, wb_spi_accel_ack_i, wb_ptc_ack_i, wb_gpio_ack_i, wb_uart_ack_i}),
137      .wbs_err_i ({wb_rom_err_i, wb_sys_err_i, wb_spi_flash_err_i, wb_spi_accel_err_i, wb_ptc_err_i, wb_gpio_err_i, wb_uart_err_i}),
138      .wbs_rty_i ({wb_rom_rty_i, wb_sys_rty_i, wb_spi_flash_rty_i, wb_spi_accel_rty_i, wb_ptc_rty_i, wb_gpio_rty_i, wb_uart_rty_i}));
139
140 endmodule
```

CPU/Controller Signals

Peripheral Signals

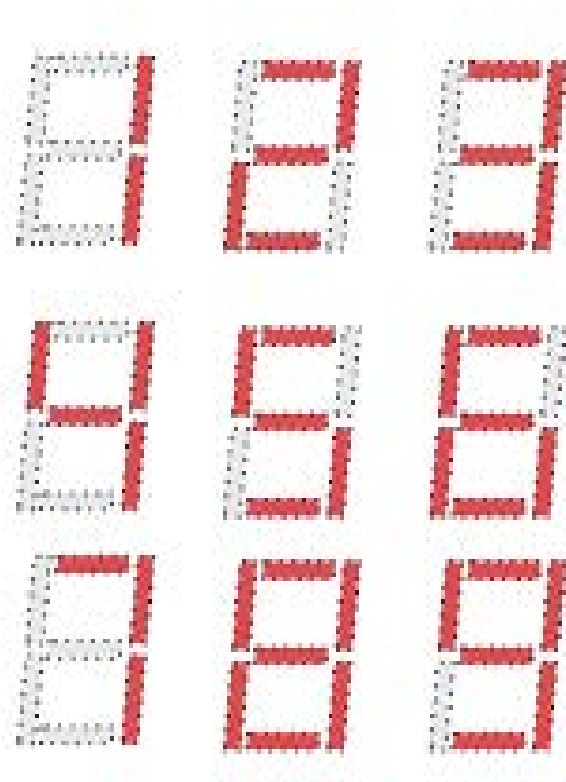
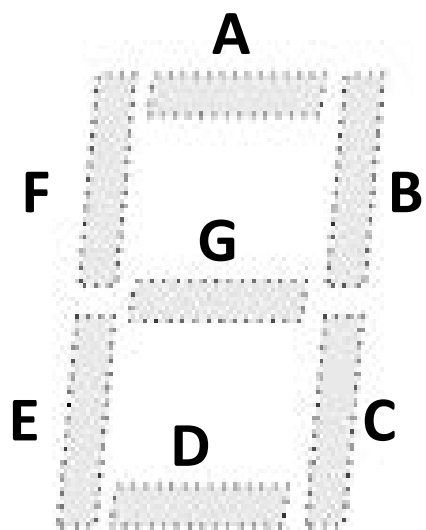
實驗7： 7段顯示器



RVfpga實驗7：7段顯示器

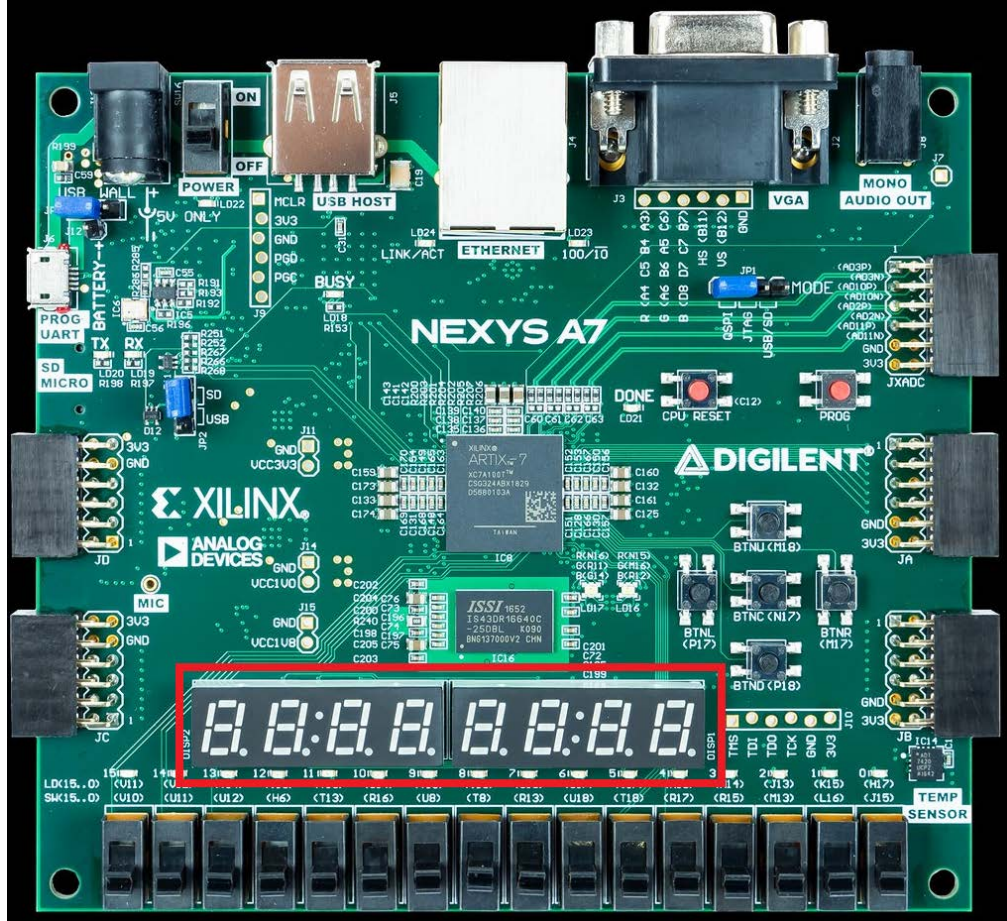
- 7段顯示器概述
- 7段顯示器硬體

RVfpga實驗7：7段顯示器概述



- 7個**LED**段：A-G
- 點亮相應段可形成特定數位
 - **1**：B段和C段
 - **2**：A段、B段、D段、E段和G段
 - **3**：A段、B段、C段、D段和G段
 - 依此類推

RVfpga實驗7：Nexys A7上的7段顯示器



開發板圖片來源：<https://reference.digilentinc.com/>

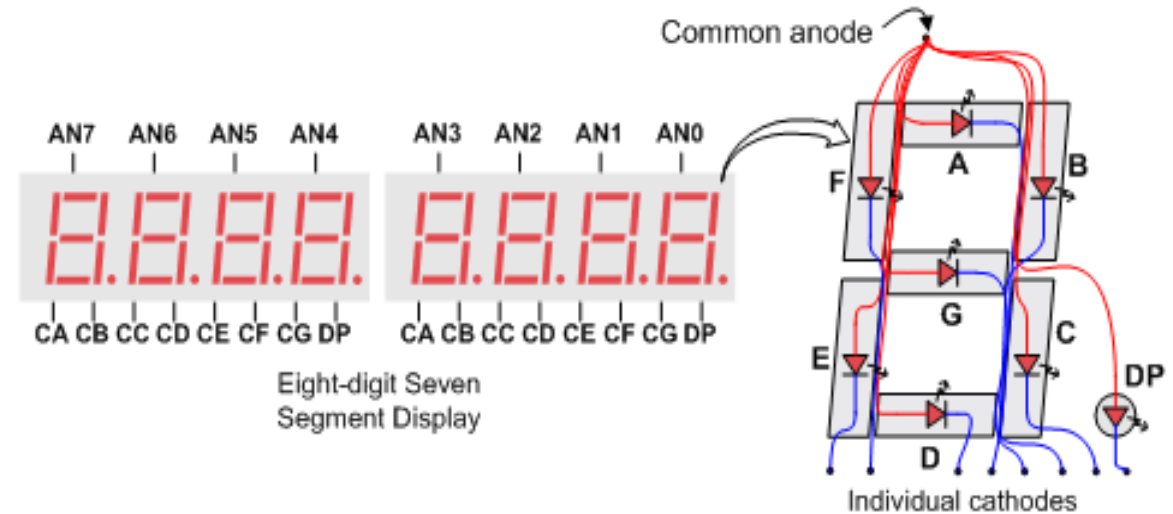
- 8位7段顯示器
- 記憶體映射存取：
 - **Enables_Reg** : 0x80001038
 - **Digits_Reg** : 0x8000103C
- 啟用訊號低電平有效
- 範例：最右邊的兩位顯示71：
 - **Enables_Reg** = 0xFC (0b11111100：啟用最右邊的兩位)
 - **Digits_Reg** = 0x71
 - 組合語言：

```
li t0, 0x80001038
li t1, 0xFC
li t2, 0x71
sw t1, 0(t0)
sw t2, 4(t0)
```

RVfpga實驗7：7段顯示器硬體

- 每一位都是**共陽極**（每一位所有**LED**的陽極連接在一起）
 - 陽極訊號充當**啟用訊號**（AN0 - AN7）
 - 驅動為**低電平**以啟用相應位（AN0-AN7經逆變器（未顯示）饋入**LED**）
- 所有數字的各個**段**均連接在一起
 - 各個段驅動為**低電平**時即可點亮
 - 憑借**AN0 - AN7**訊號的**時分復用**，可以在每一位上顯示惟一值
 - 一個位的**AN**訊號（AN0 - AN7）必須每**1-16 ms**變為低電平才能點亮

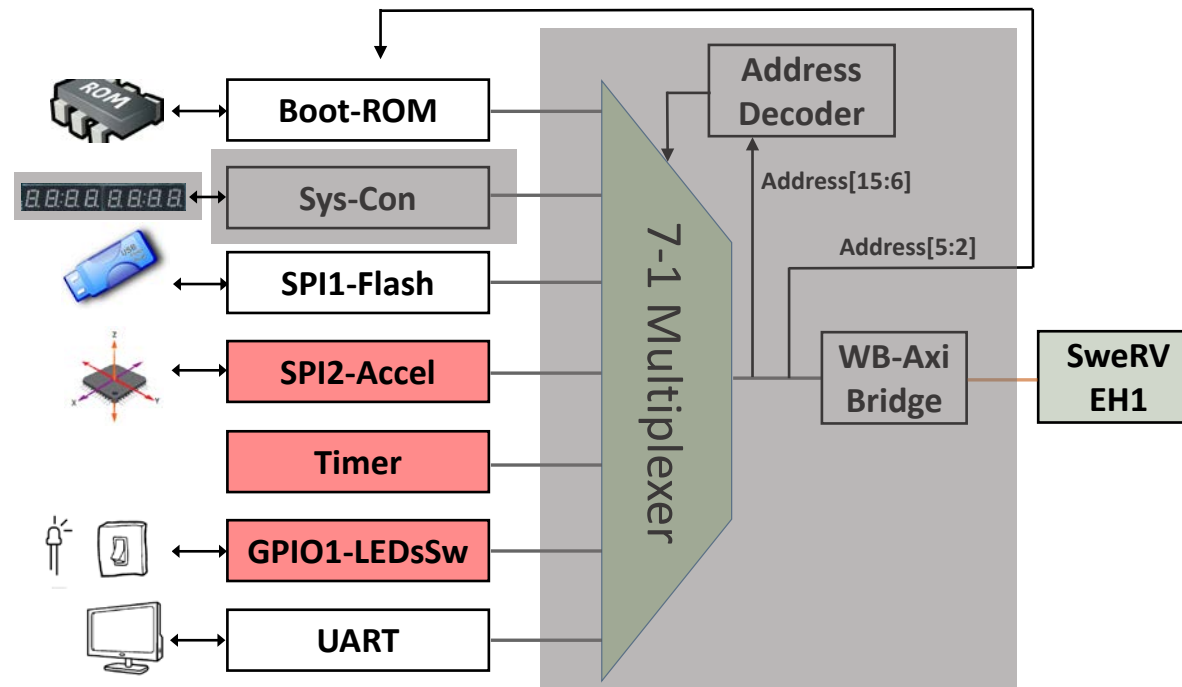
8位7段顯示器



RVfpga實驗7：7段顯示器低階實作

- 主要分為3個部分

- RvfpgaNexys與板上7段顯示器的外部連接（左側陰影區域）
- SweRVolfX中整合的7段顯示器模組（中間陰影區域）
- 7段顯示器和SweRV EH1之間的連接（右側陰影區域）



RVfpga實驗7：外部連接

檔案rvfpganexys.xdc：定義使用開發板上7段顯示器時CA-CG（在SoC中稱為Digits_Bits[i]）和AN[i]的連接

```
60 ##7 segment display
61 set_property -dict { PACKAGE_PIN T10 IOSTANDARD LVCMOS33 } [get_ports { CA }]; #IO_L24N_T3_A00_D16_14 Sch=ca
62 set_property -dict { PACKAGE_PIN R10 IOSTANDARD LVCMOS33 } [get_ports { CB }]; #IO_25_14 Sch=cb
63 set_property -dict { PACKAGE_PIN K16 IOSTANDARD LVCMOS33 } [get_ports { CC }]; #IO_25_15 Sch=cc
64 set_property -dict { PACKAGE_PIN K13 IOSTANDARD LVCMOS33 } [get_ports { CD }]; #IO_L17P_T2_A26_15 Sch=cd
65 set_property -dict { PACKAGE_PIN P15 IOSTANDARD LVCMOS33 } [get_ports { CE }]; #IO_L13P_T2_MRCC_14 Sch=ce
66 set_property -dict { PACKAGE_PIN T11 IOSTANDARD LVCMOS33 } [get_ports { CF }]; #IO_L19P_T3_A10_D26_14 Sch=cf
67 set_property -dict { PACKAGE_PIN L18 IOSTANDARD LVCMOS33 } [get_ports { CG }]; #IO_L4P_T0_D04_14 Sch=cg
68 #set_property -dict { PACKAGE_PIN H15 IOSTANDARD LVCMOS33 } [get_ports { DP }]; #IO_L19N_T3_A21_VREF_15 Sch=dp
69 set_property -dict { PACKAGE_PIN J17 IOSTANDARD LVCMOS33 } [get_ports { AN[0] }]; #IO_L23P_T3_F0E_B_15 Sch=an[0]
70 set_property -dict { PACKAGE_PIN J18 IOSTANDARD LVCMOS33 } [get_ports { AN[1] }]; #IO_L23N_T3_FWE_B_15 Sch=an[1]
71 set_property -dict { PACKAGE_PIN T9 IOSTANDARD LVCMOS33 } [get_ports { AN[2] }]; #IO_L24P_T3_A01_D17_14 Sch=an[2]
72 set_property -dict { PACKAGE_PIN J14 IOSTANDARD LVCMOS33 } [get_ports { AN[3] }]; #IO_L19P_T3_A22_15 Sch=an[3]
73 set_property -dict { PACKAGE_PIN P14 IOSTANDARD LVCMOS33 } [get_ports { AN[4] }]; #IO_L8N_T1_D12_14 Sch=an[4]
74 set_property -dict { PACKAGE_PIN T14 IOSTANDARD LVCMOS33 } [get_ports { AN[5] }]; #IO_L14P_T2_SRCC_14 Sch=an[5]
75 set_property -dict { PACKAGE_PIN K2 IOSTANDARD LVCMOS33 } [get_ports { AN[6] }]; #IO_L23P_T3_35 Sch=an[6]
76 set_property -dict { PACKAGE_PIN U13 IOSTANDARD LVCMOS33 } [get_ports { AN[7] }]; #IO_L23N_T3_A02_D18_14 Sch=an[7]
77
```



RVfpga實驗7：SweRVolfX整合

- **swervolf_syscon.v**檔：7段顯示器控制器實例化。除了時脈訊號（**i_clk**）和重設訊號（**i_rst**）外，該模組還接收兩個輸入訊號（**Enables_Reg**和**Digits_Reg**），即前文所述的兩個暫存器映射控制暫存器。該模組輸出兩個訊號（**AN**和**Digits_Bits**），這兩個訊號連接到開發板上7段顯示器。

```
// Eight-Digit 7 Segment Displays

reg [ 7:0] Enables_Reg;
reg [31:0] Digits_Reg;

SevSegDisplays_Controller SegDispl_Ctr(
    .clk          (i_clk),
    .rst_n        (i_rst),
    .Enables_Reg  (Enables_Reg),
    .Digits_Reg   (Digits_Reg),
    .AN           (AN),
    .Digits_Bits  (Digits_Bits)
);
```

RVfpga實驗7：SweRVolfX整合

- 該檔案中還實作了 **SevSegDisplays_Controller**，其中包含以下子單元：
 - 兩個多工器（模組 **SevSegMux**），用於選擇每2 ms傳送到AN和 Digits_Bits訊號的值。
 - 計數器（模組 **counter**），用於產生2 ms週期。
 - 解碼器（模組 **SevenSegDecoder**），用於輸出給定4位元十六進位值的段值。

實驗8： 計時器



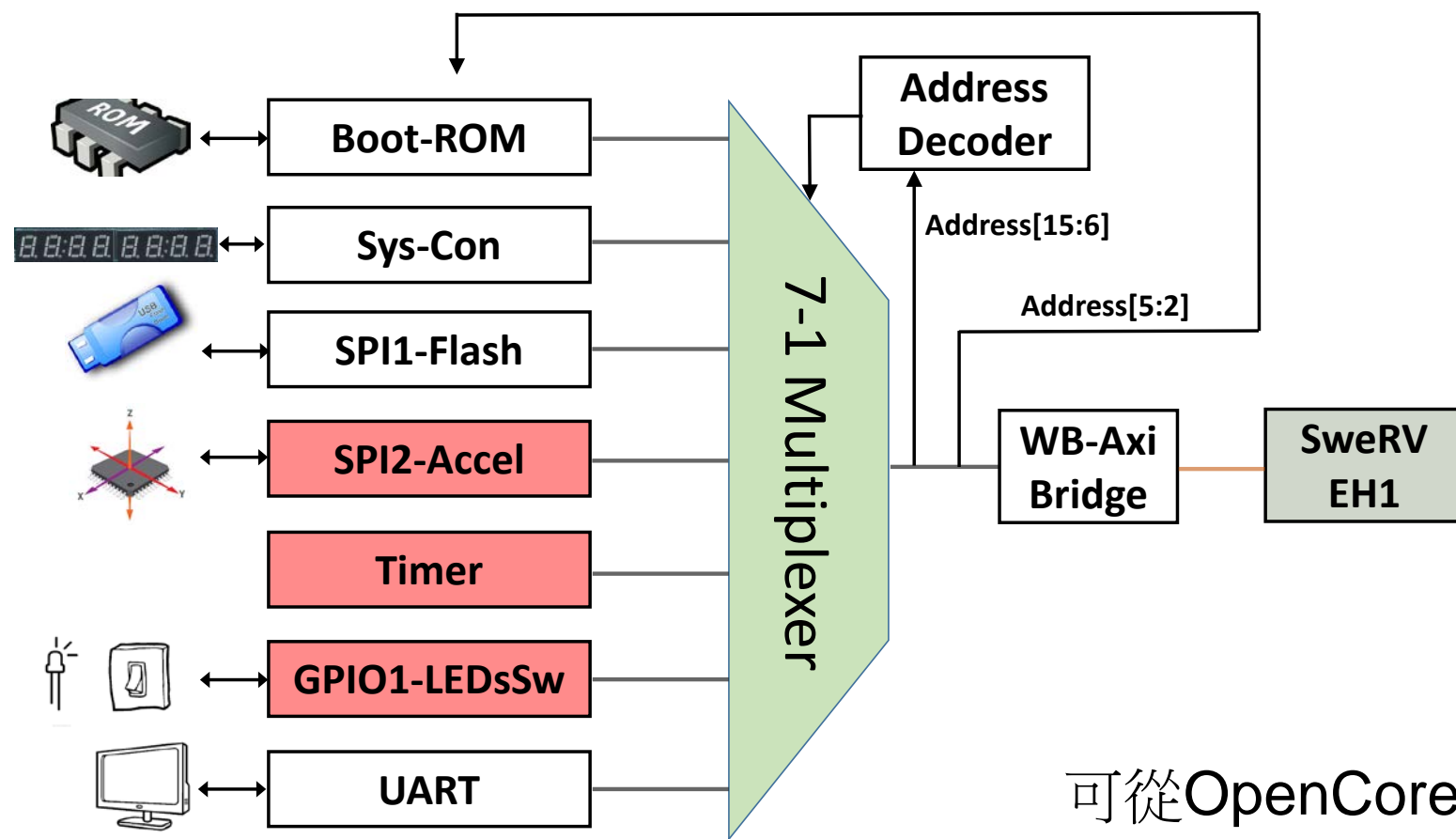
RVfpga實驗8：計時器

- 計時器概述
- 計時器暫存器
- 計時器範例

RVfpga實驗8：計時器

- 計時器以固定的頻率遞增或遞減計數器
- 常見於微控制器和SoC中
- 用於產生精確時序

RVfpga實驗8：計時器

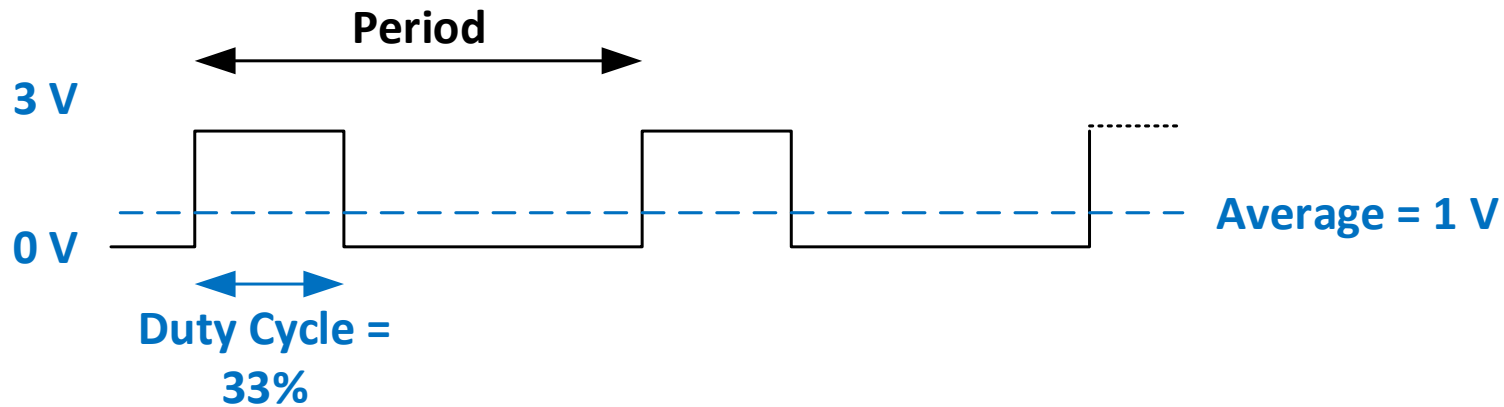


週邊設備

可從OpenCores
(<https://opencores.org/projects/ptc>)
取得使用的計時器模組。

RVfpga實驗8：計時器（PTC）模組

- 計時器模組（也稱為**PTC**模組）用於：
 - 計時器/計數器：對時脈邊緣（或另一個訊號的邊緣，也稱為事件）進行計數
 - 脈寬調變（PWM）：
 - 輸出變為高電平的持續時間（稱為工作週期）
 - 用於以數位方式計算模擬電壓近似值
- PWM範例**：33%工作週期（訊號在1/3的時間內為高電平）如果高電平為3 V，則模擬電壓（訊號的平均電壓）為 $3\text{ V} * 0.33 = 1\text{ V}$



RVfpga實驗8：計時器（PTC）暫存器

名稱	位址	寬度	存取	說明
RPTC_CNTR	0x80001200	1-32	R/W	主PTC計數器
RPTC_HRC	0x80001204	1-32	R/W	PTC高電平參考/擷取暫存器
RPTC_LRC	0x80001208	1-32	R/W	PTC低電平參考/擷取暫存器
RPTC_CTRL	0x8000120C	9	R/W	控制暫存器

- **RPTC_CNTR**：計數器（計數器的值）
- **RPTC_HRC**：高電平參考擷取 - 指示PWM模式下輸出應變為高電平的週期數（重設後）
- **RPTC_LRC**：低電平參考擷取 - 指示計數器/計時器模式下計數完成時的週期數（重設後）；指示PWM模式下輸出應變為低電平的時脈週期數（重設後）。
- **RPTC_CTRL**：控制暫存器

RVfpga實驗8：計時器（PTC）控制暫存器

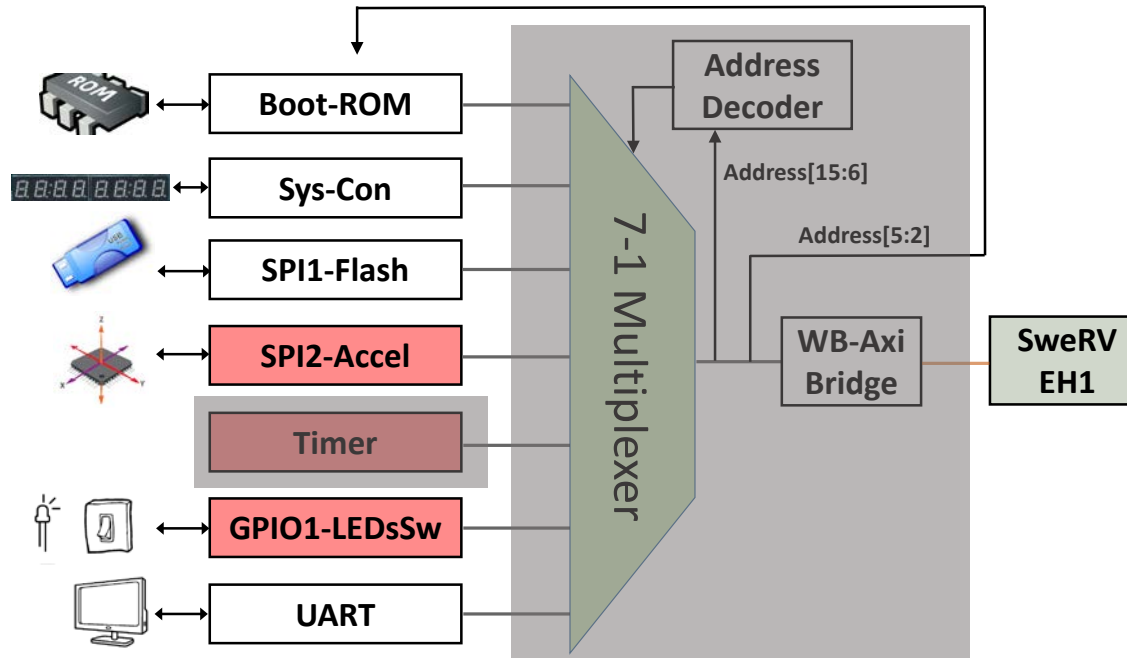
位元	存取	重設	名稱和說明
0	R/W	0	EN ：設為1時，RPTC_CNTR遞增。
1	R/W	0	ECLK ：選擇時鐘訊號：外部時鐘（通過<g>ptc_ecgt</g>）（1）或系統時鐘（0）。
2	R/W	0	NEC ：用於選擇外部時脈（ptc_ecgt）的負邊緣/正邊緣和低電平/高電平週期。
3	R/W	0	OE ：啟用PWM輸出驅動器。
4	R/W	0	SINGLE ：設為1時，RPTC_CNTR在達到RPTC_LRC值後不遞增。清零時，RPTC_CNTR在達到RPTC_LCR暫存器中的值後重新啟動。
5	R/W	0	INTE ：設為1時，當RPTC_CNTR值等於RPTC_LRC或RPTC_HRC的值時，PTC會將中斷置為有效。清除訊號時，中斷將被屏蔽。
6	R/W	0	INT ：讀取時，該位元表示待處理的中斷。設為1時，表示有一個中斷待處理。當該位元寫入1時，中斷請求將被清除。
7	R/W	0	CNTRRST ：設為1時，將重設RPTC_CNTR。清零時，計數器將正常工作。
8	R/W	0	CAPTE ：設為1時，RPTC_CNTR將被擷取到RPTC_LRC或RPTC_HRC暫存器中。清零時，擷取功能將被屏蔽。

RVfpga實驗8：計時器範例

- 將**RPTC_LRC**設定為要計數的週期數
- 設定控制位元（**RPTC_CTRL**）以配置計時器：
 - 重設計數器並清除中斷：**RPTC_CTRL = 0xC0**（0b011000000）：CNTRRST（bit 7）= 1：重設計數器（RPTC_CNTR = 0）；INT（bit 6）= 1：清除中斷請求。
 - 啟用計數器並允許中斷：**RPTC_CTRL = 0x21**（0b000100001）：EN（bit 0）= 1：啟用計數器，從而使RPTC_CNTR遞增；INTE（bit 5）= 1：當RPTC_CNTR == RPTC_LRC時，PTC會將中斷視為有效。
- 程式讀取控制暫存器中的中斷位元（**INT**是**RPTC_CTRL**的**bit 6**），直到其為1（表示RPTC_CNTR == RPTC_LRC）。
- 該演算法不使用中斷，但它會讀取中斷位元（INT，RPTC_CTRL的bit 6）以確定何時達到正確的時脈週期數。我們將在實驗9中展示如何使用中斷。

RVfpga實驗8：計時器低階實作

- 主要分為2個部分
 - （無外部連接）
 - SweRVolfX中整合的計時器模組（左側陰影區域）
 - 計時器和SweRV EH1之間的連接（右側陰影區域）



RVfpga實驗8：SweRVolfX整合

swervolf_core.v檔：PTC模組實例化

```
// PTC
wire          ptc_irq;

ptc_top timer_ptc(
    .wb_clk_i      (clk),
    .wb_rst_i      (wb_rst),
    .wb_cyc_i      (wb_m2s_ptc_cyc),
    .wb_adr_i      ({2'b0,wb_m2s_ptc_adr[5:2],2'b0}),
    .wb_dat_i      (wb_m2s_ptc_dat),
    .wb_sel_i      (4'b1111),
    .wb_we_i      (wb_m2s_ptc_we),
    .wb_stb_i      (wb_m2s_ptc_stb),
    .wb_dat_o      (wb_s2m_ptc_dat),
    .wb_ack_o      (wb_s2m_ptc_ack),
    .wb_err_o      (wb_s2m_ptc_err),
    .wb_inta_o     (ptc_irq),
    // External PTC Interface
    .gate_clk_pad_i (),
    .capt_pad_i  (),
    .pwm_pad_o  (),
    .oen_padoen_o ()
);
```



實驗9： 中斷驅動I/O



RVfpga實驗9：中斷驅動I/O

- 中斷驅動I/O與程式設計I/O
- Rvfpga系統的中斷控制器
- 如何使用Western Digital的週邊設備支援套件和開發板支援套件（PSP和BSP）來配置中斷
- 中斷範例

RVfpga實驗9：中斷驅動I/O簡介

- 程式設計I/O：

- 程式會連續輪詢一個值（例如開關），直到獲得所需值為止。
- 舉例來說，該方法曾在之前的實驗中用於讀取開關。
- 該方法會佔用處理器，不斷存取一個值（而無法執行其他有用的工作）。

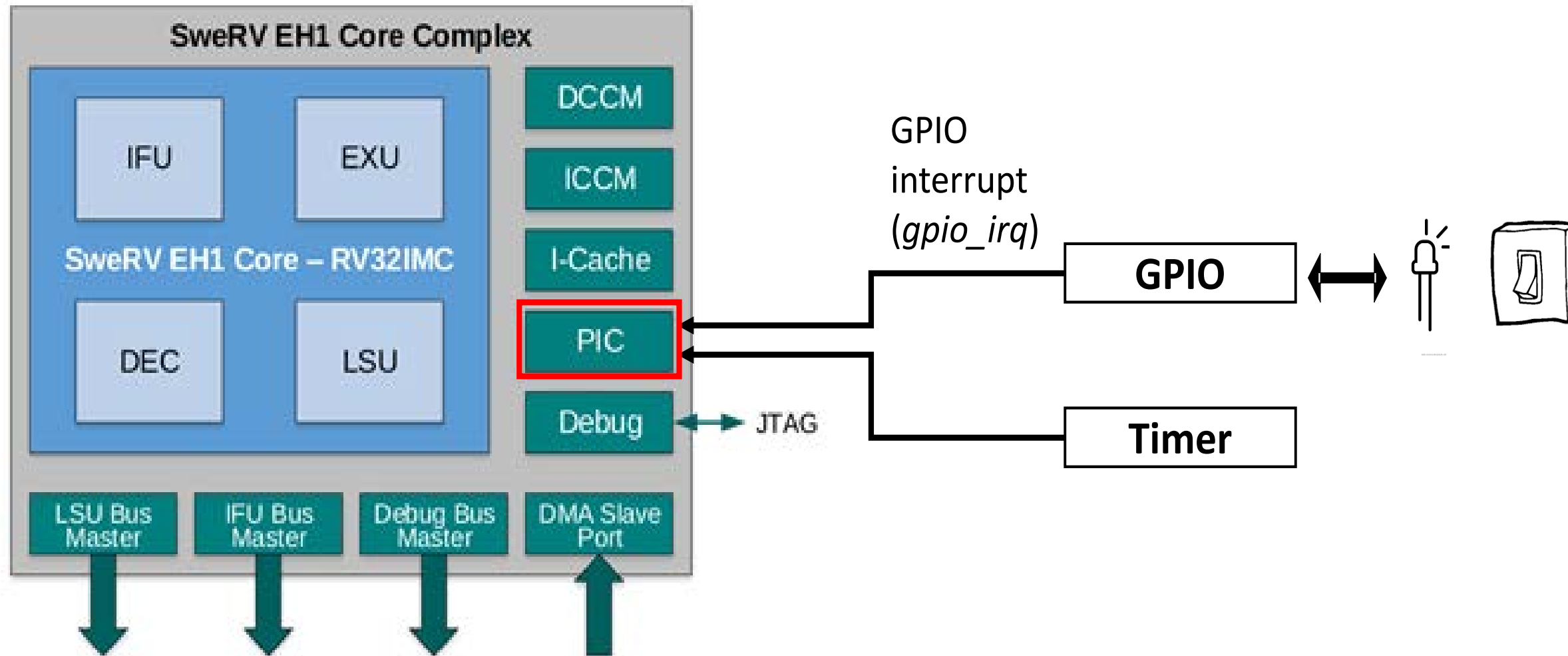
- 中斷驅動I/O：

- 事件（例如，引腳視為有效）使處理器跳轉到中斷服務常式（**ISR**，也稱為中斷處理常式），這類似於未調度的函數呼叫。**ISR**處理中斷（例如，讀取開關的值），然後返回到常規程式。
- 在該事件發生之前，處理器可以繼續執行有用的工作。

RVfpga實驗9：處理中斷

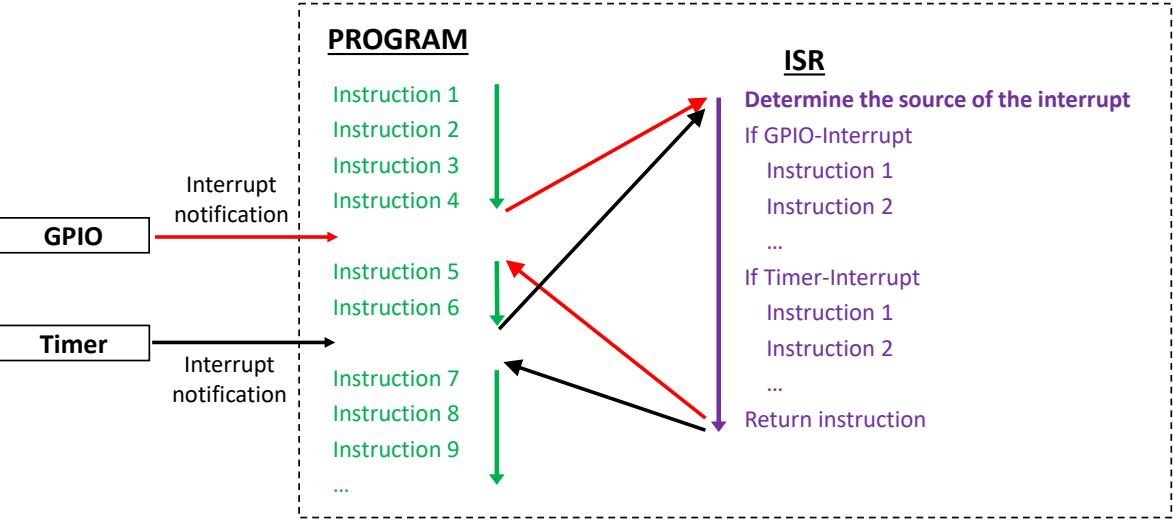
- 中斷可能是由**硬體**或**軟體**引起的
- 在本實驗中，我們重點關注的是**硬體中斷**
- **SweRV EH1**核心按照**RISC-V**的**PLIC**（平台級中斷控制器）規格處理中斷。該核心被稱為可程式化中斷控制器（**PIC**）。它具有：
 - 255個中斷源
 - 15個優先級

RVfpga實驗9：中斷硬體

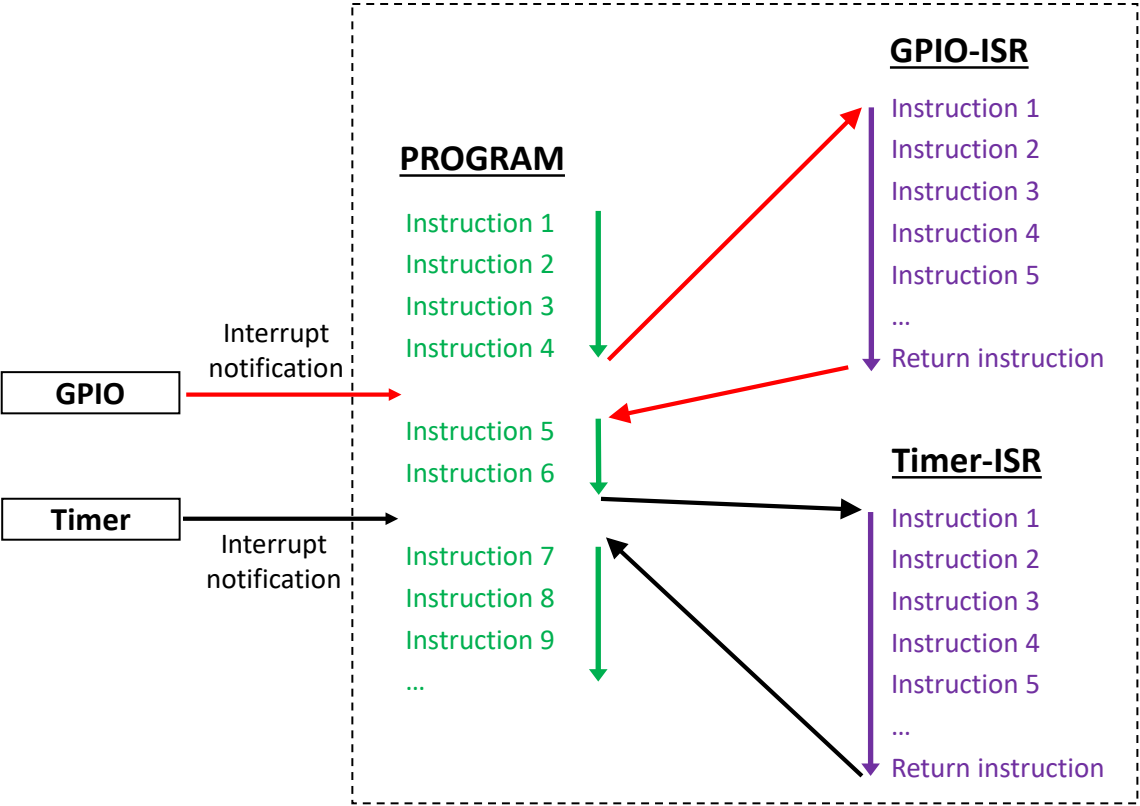


RVfpga實驗9：單向量模式與多向量模式

單向量模式範例：



多向量模式範例：



RVfpga實驗9：處理中斷

- 使用WD的PSP/BSP：
 - 使用WD的PSP/BSP初始化中斷
 - 初始化255個中斷中的一或多個中斷，並提供ISR的名稱
 - 將應觸發中斷的週邊設備訊號與中斷引腳相連。
 - 允許所有中斷
 - 允許外部中斷

RVfpga實驗9：中斷範例

- 使用中斷讀取Switch[0]的值 - 僅在上升邊緣（0跳變為→1）

名稱	位址	寬度	存取	說明
RGPIO_IN	0x80001400	1-32	R	GPIO輸入資料
RGPIO_OUT	0x80001404	1-32	R/W	GPIO輸出資料
RGPIO_OE	0x80001408	1-32	R/W	GPIO輸出驅動器啟用
RGPIO_INTE	0x8000140C	1-32	R/W	中斷啟用
RGPIO_PTRIG	0x80001410	1-32	R/W	觸發中斷的事件類型
RGPIO_AUX	0x80001414	1-32	R/W	多工處理輔助輸入與GPIO輸出
RGPIO_CTRL	0x80001418	2	R/W	控制暫存器
RGPIO_INTS	0x8000141C	1-32	R/W	中斷狀態
RGPIO_ECLK	0x80001420	1-32	R/W	啟用gpio_eclk以鎖存RGPIO_IN
RGPIO_NEC	0x80001424	1-32	R/W	選擇gpio_eclk的有效邊緣

有關完整代碼，請參見：[\[RVfpgaPath\]/RVfpga/Labs/Lab9/LED-Switch_7SegDispl_Interrupts_C-Lang.c](#)

RVfpga實驗9：中斷範例

- 設定用於中斷的GPIO暫存器：
 - `RGPIO_INTE = 0x10000`（允許Switch[0]的中斷）
 - `RGPIO_PTRIG = 0x10000`（在Switch[0]的上升邊緣觸發中斷）
 - `RGPIO_INTS = 0x0`（清除所有中斷）
 - `RGPIO_CTRL = 0x1`（允許GPIO中斷）

RVfpga實驗9：中斷範例

- GPIO ISR：

```
void GPIO_ISR(void) {
    unsigned int i;

    /* Invert LED value */
    i = M_PSP_READ_REGISTER_32(GPIO_LEDS);      /* RGPIO_OUT */
    i = !i;                                       /* Invert the LEDs */
    i = i & 0x1;                                  /* Only keep right-most LED */
    M_PSP_WRITE_REGISTER_32(GPIO_LEDS, i)        /* RGPIO_OUT */

    /* Clear GPIO interrupt */
    M_PSP_WRITE_REGISTER_32(RGPIO_INTS, 0x0); /* RGPIO_INTS */

    /* Stop the generation of this interrupt (IRQ4) */
    bspClearExtInterrupt(4);
}
```

RVfpga實驗9：中斷範例

- 將中斷4（IRQ4）與開關的中斷相連，並將中斷服務常式設定為GPIO_ISR
- 記憶體對映暫存器 $0x80001018 = 0x1$ ：將GPIO中斷與IRQ4相連
- 啟用全域中斷

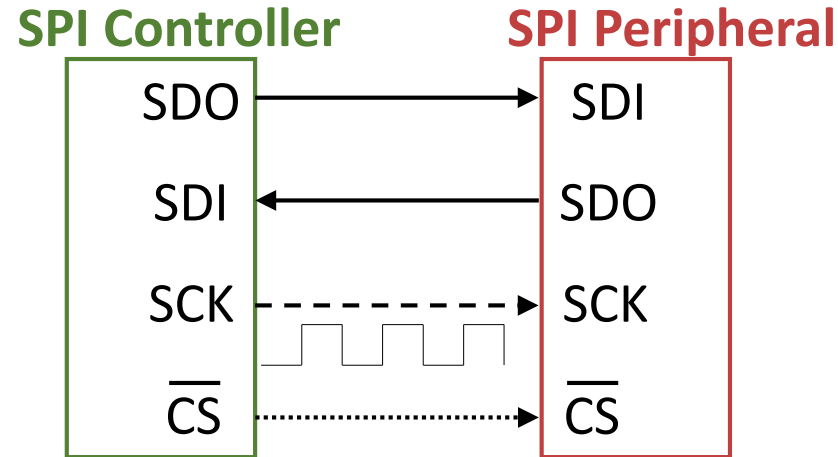
實驗10： 序列匯流排



RVfpga實驗10：序列匯流排

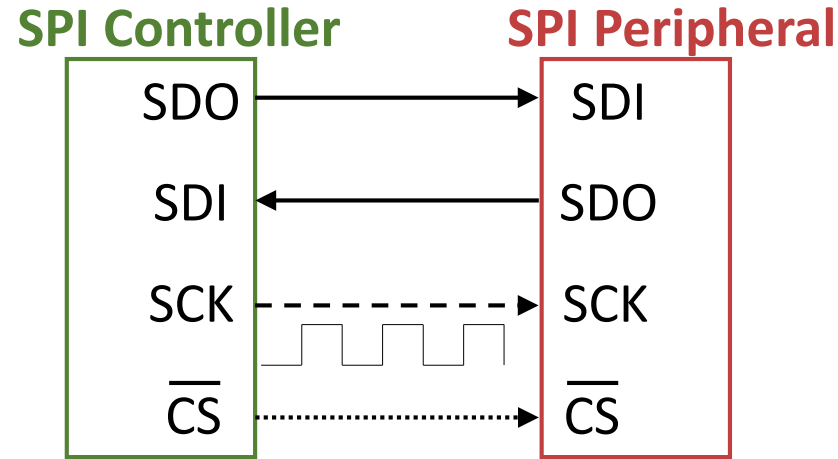
- 序列匯流排一次傳送一位元
 - 相比之下，並行匯流排則一次傳送多個位元
- 通用序列匯流排
 - **UART**（通用異步收發器）
 - **SPI**（序列週邊設備介面）
 - **I2C**（積體電路間通訊協定）
- 在本實驗中，我們重點關注的是**SPI**

RVfpga實驗10：序列匯流排

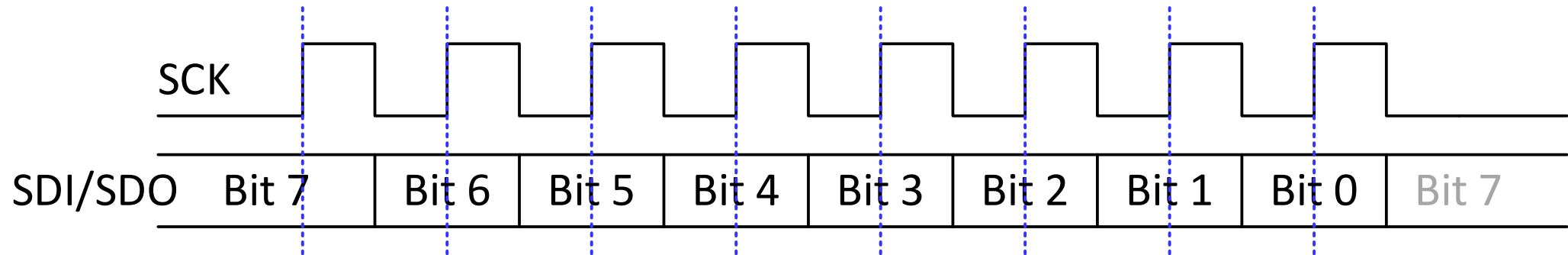


- 控制器：傳送時脈，傳送和接收資料
- 周邊設備：接收時脈，傳送和接收資料
- 訊號：
 - **SDO**：序列資料輸出
 - **SDI**：序列資料輸入
 - **SCK**：SPI時鐘
 - **CSbar**：低電平有效晶片選擇

RVfpga實驗10：序列匯流排



- **SCK閒置**
- 當控制器傳送**SCK邊緣**時，控制器和周邊設備將採樣並傳送資料。資料在下降邊緣變更（傳送），在上升邊緣採樣（但這是可配置的）



RVfpga實驗10：RVfpga系統的SPI模組

- RVfpga系統的SPI模組來自OpenCores

https://opencores.org/projects/simple_spi

- 4項目讀寫緩衝區
- SPI暫存器：

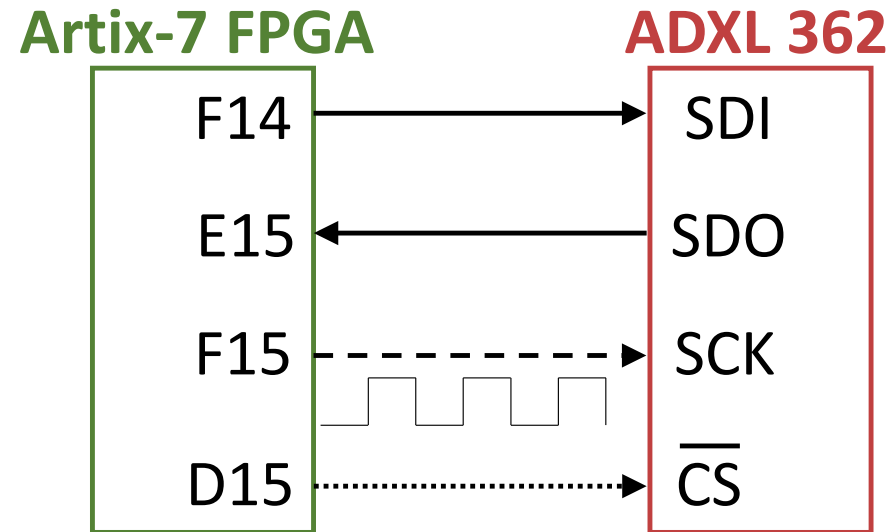
名稱	位址	寬度	存取	說明
SPCR	0x80001100	8	R/W	控制暫存器
SPSR	0x80001108	8	R/W	狀態暫存器
SPDR	0x80001110	8	R/W	資料暫存器
SPER	0x80001118	8	R/W	擴展暫存器
SPCS	0x80001120	8	R/W	CS暫存器



RVfpga實驗10：ADXL362加速計

- Nexys A7開發板包括一個類比裝置ADXL362加速計。有關完整資訊，請造訪：

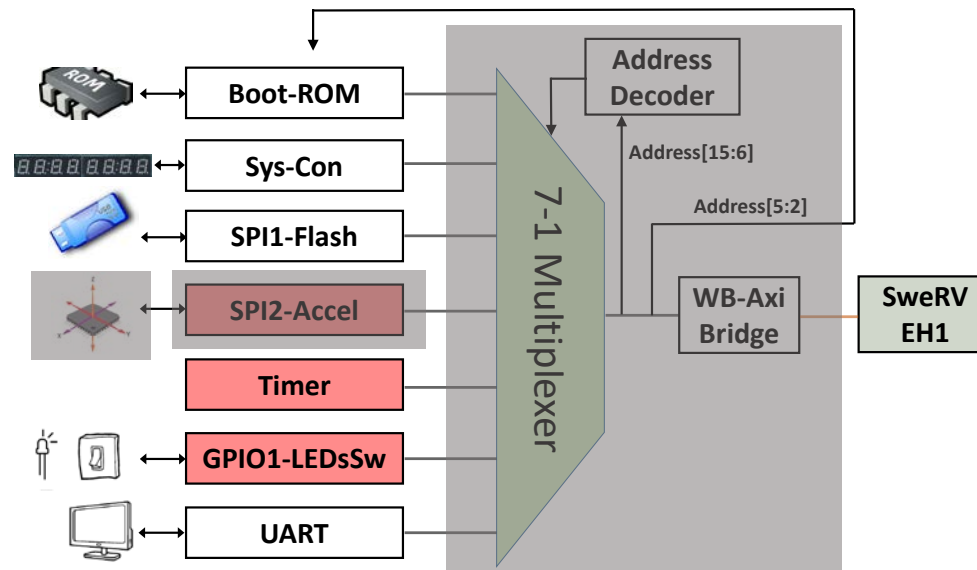
<https://www.analog.com/media/en/technical-documentation/data-sheets/ADXL362.pdf>



RVfpga實驗10：加速計低階實作

- 主要分為3個部分

- RVfpgaNexys與板上加速計的外部連接（左側陰影區域）
- SweRVolfX中整合的全新SPI模組（中間陰影區域）
- 加速計和SweRV EH1之間的連接（右側陰影區域）



RVfpga實驗10：外部連接

檔案**rvfpganexys.xdc**：定義SoC中使用的SPI訊號與相應開發板上加速計引腳的連接

```
78 ##Accelerometer
79 set_property -dict { PACKAGE_PIN E15   IOSTANDARD LVCMOS33 } [get_ports { i_accel_miso }]; #IO_L11P_T1_SRCC_15 Sch=acl_miso
80 set_property -dict { PACKAGE_PIN F14   IOSTANDARD LVCMOS33 } [get_ports { o_accel_mosi }]; #IO_L5N_T0_AD9N_15 Sch=acl_mosi
81 set_property -dict { PACKAGE_PIN F15   IOSTANDARD LVCMOS33 } [get_ports { accel_sclk }]; #IO_L14P_T2_SRCC_15 Sch=acl_sclk
82 set_property -dict { PACKAGE_PIN D15   IOSTANDARD LVCMOS33 } [get_ports { o_accel_cs_n }];
```

RVfpga實驗10：SweRVolfX整合

swervolf_core.v檔：三態緩衝區和GPIO模組實例化

```
simple_spi spi2
// Wishbone slave interface
.clk_i  (clk),
.rst_i  (wb_rst),
.adr_i  (wb_m2s_spi_accel_adr[2] ? 3'd0 : wb_m2s_spi_accel_adr[5:3]),
.dat_i  (wb_m2s_spi_accel_dat[7:0]),
.we_i   (wb_m2s_spi_accel_we),
.cyc_i  (wb_m2s_spi_accel_cyc),
.stb_i  (wb_m2s_spi_accel_stb),
.dat_o  (spi2_rdt),
.ack_o  (wb_s2m_spi_accel_ack),
.inta_o (spi2_irq),
// SPI interface
.sck_o  (o_accel_sclk),
.ss_o   (o_accel_cs_n),
.mosi_o (o_accel_mosi),
.miso_i (i_accel_miso));
```



實驗11-20 瞭解核心和 記憶體系統



RVfpga實驗11-20概述

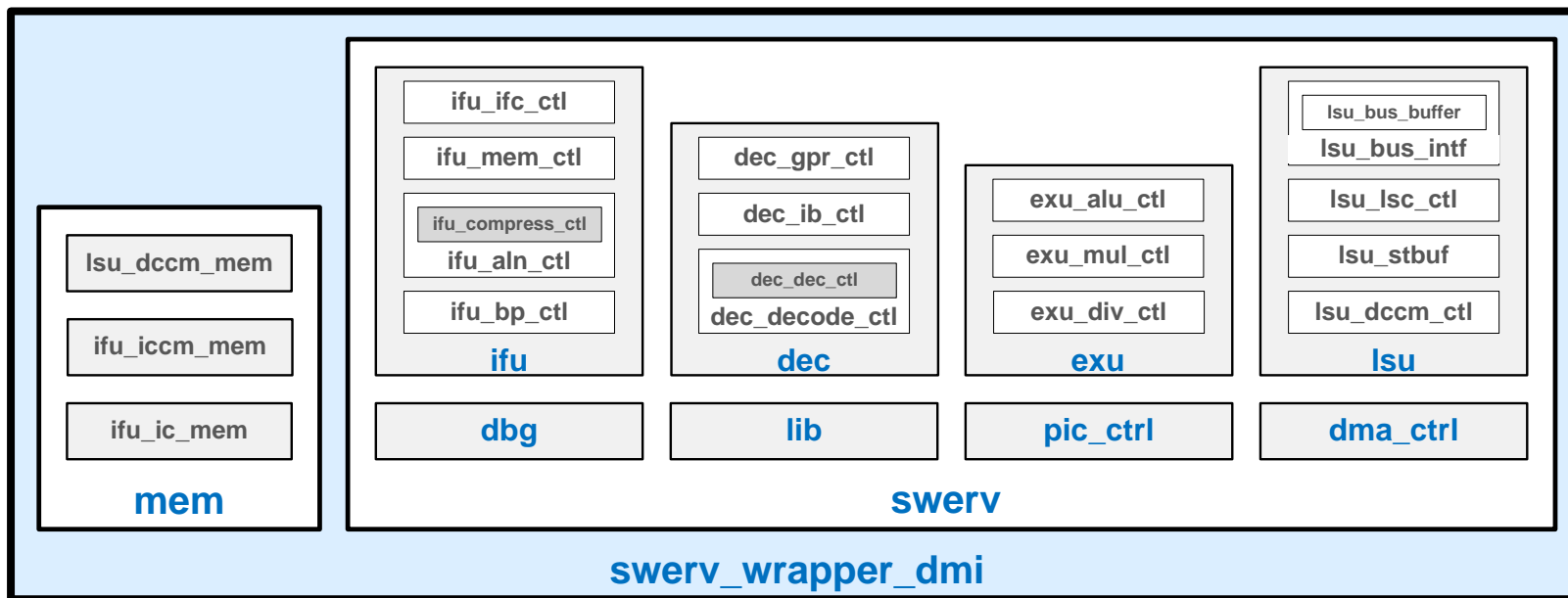
- 實驗11-20將深入到微架構等級並分析SweRV EH1處理器以及快取/記憶體階層的工作方式。
 - 每個實驗分為兩部分：
 - 概念的理論說明
 - 概念的圖示說明（使用圖和範例程式的Verilator模擬來說明概念）。
- 我們還提供相關練習以加深對所描述概念的理解和體驗。

實驗11： SweRV EH1組態、 結構和效能監視



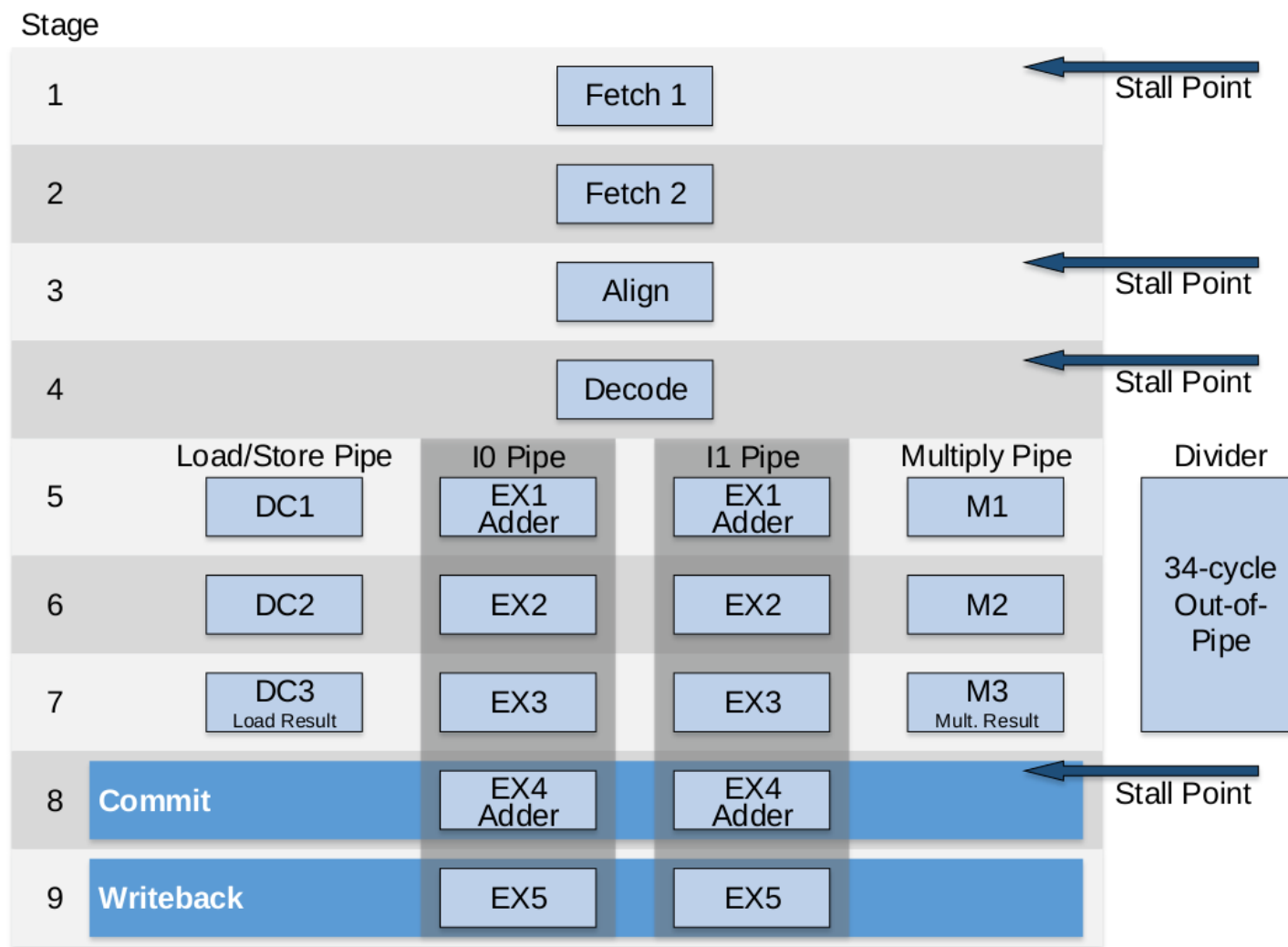
RVfpga實驗11：SweRV EH1 Verilog模組的階層

- 下圖顯示了主要Verilog模組的階層。
- **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。



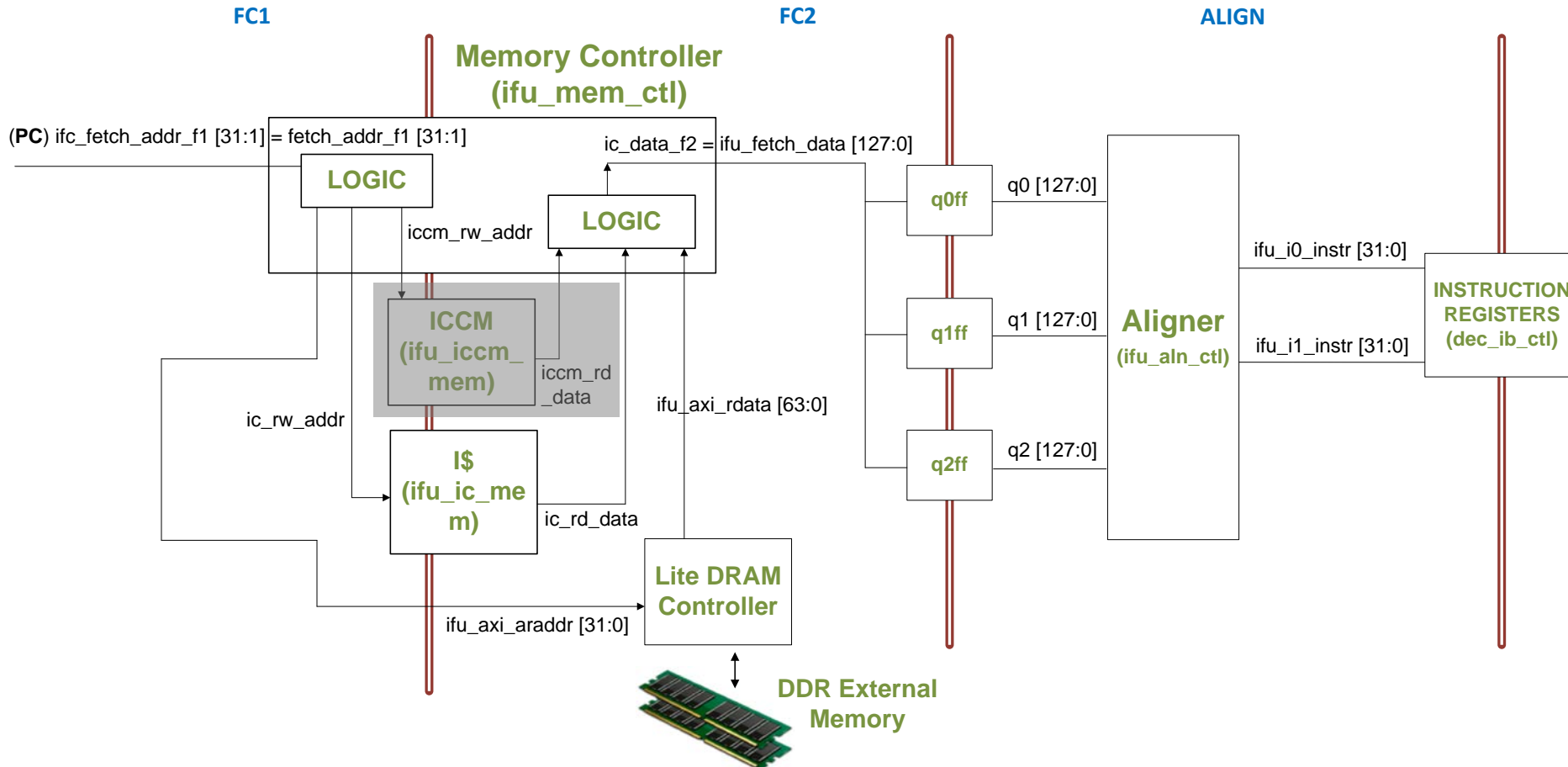
RVfpga實驗11：SweRV EH1管線

- 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級管線有序處理器。
- 右圖顯示了處理器微架構的高階檢視，實驗11-20將對此進行詳細分析。



RVfpga實驗11：擷取（FC1和FC2）和對齊階段

- SweRV EH1管線的前三個階段是：兩個擷取階段（FC1和FC2）和一個對齊階段



RVfpga實驗11：擷取（FC1和FC2）階段

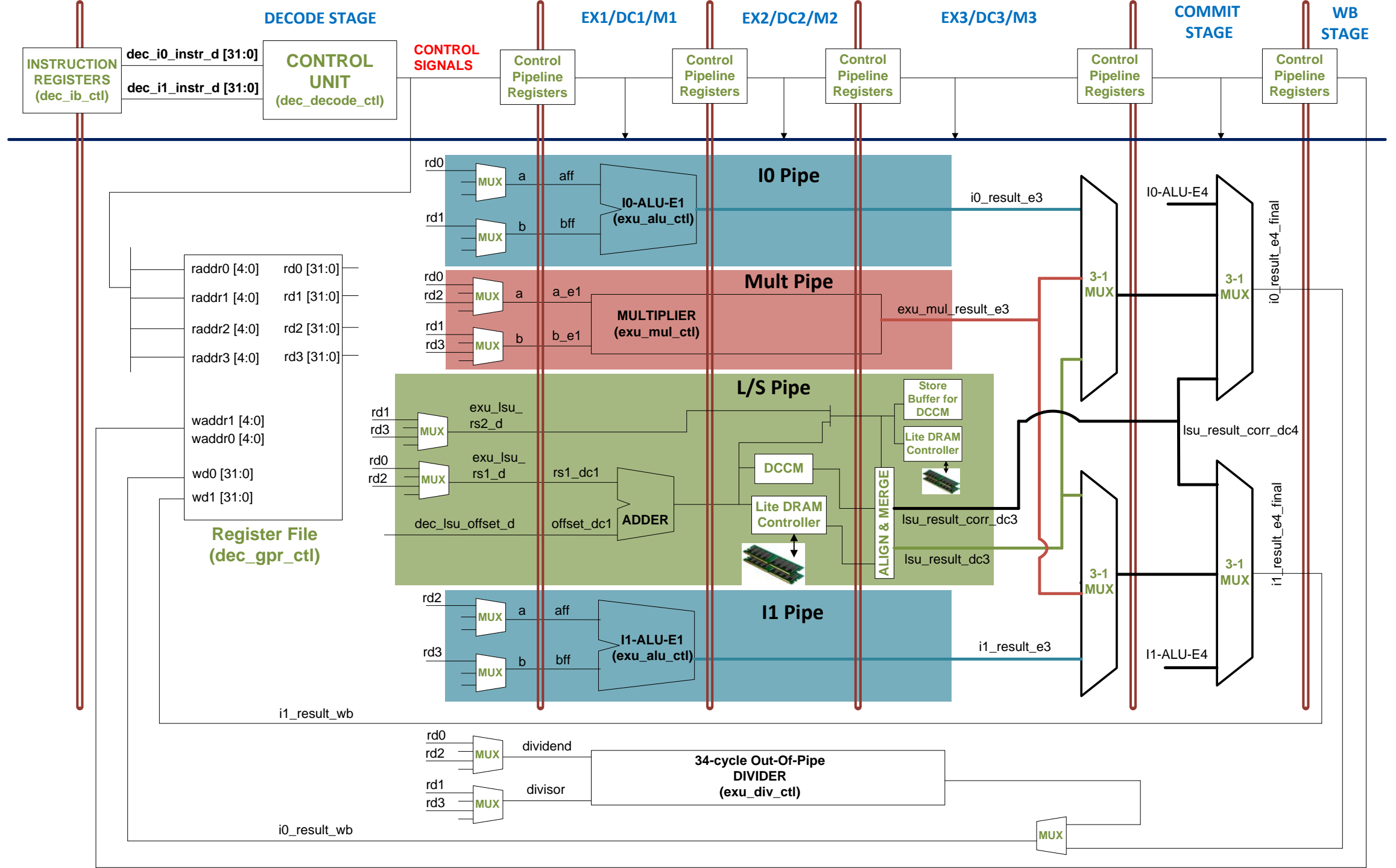
- 擷取階段：從指令記憶體讀取指令
- 在RVfpga中，指令記憶體包括：
 - 512 KiB ICCM（在預設RVfpga組態中停用）
 - 16 KiB指令快取
 - 128 MiB DDR外部記憶體
- **FC1**：計算指令位址（`ifc_fetch_addr_f1`）
- **FC2**：從I\$、DDR外部記憶體或ICCM讀取指令。（I\$僅快取主記憶體位址範圍內的記憶體。）

RVfpga實驗11：對齊階段

- 對齊階段執行兩個主要任務：
 - 每個週期向解碼階段提供兩條**32位元指令**：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，並將它們指定給**SweRV EH1**中的每個通路（共兩個通路）。
 - **解壓縮指令**：對齊階段將**16位元指令**解壓縮為**32位元指令**。

RVfpga實驗11：解碼、EX1/2/3、提交和WB階段

- 下一張幻燈片上的圖顯示了管線的最後六個階段：解碼階段、三個執行階段、提交階段和寫回（Writeback，WB）階段。



RVfpga實驗11：解碼階段

- 解碼階段執行兩個主要任務：
 - 對指令進行解碼並產生控制訊號（由控制單元執行）
 - 將指令和運算元分發到適當的管道：
 - 管道：
 - 兩個整數管道：I0和I1
 - 乘法管道
 - 載入/儲存管道（L/S）
 - 管外34週期除法器
 - 幾個多路開關在可能的運算元中進行選擇

RVfpga實驗11：執行階段 – 3個管道和一個除法器

- I0/I1管道：兩個整數管道具有三個階段（EX1、EX2和EX3）。EX1執行ALU運算。
- 乘法管道：乘法管道包含一個使用三個階段（M1、M2和M3）的3週期整數乘法器。
- 載入/儲存（L/S）管道：L/S管道執行載入和儲存指令。
- 除法器：除法器是一個非管線式34週期整數除法器。

在第三個執行階段（EX3/DC3/M3）結束時，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。

RVfpga實驗11：提交和寫回階段

- 提交階段：選擇要寫回到暫存器檔案的結果。
- 寫回階段：使用寫入連接埠0和1將結果寫入暫存器檔案。控制管線暫存器提供暫存器識別碼和啟用訊號（在解碼階段產生）。

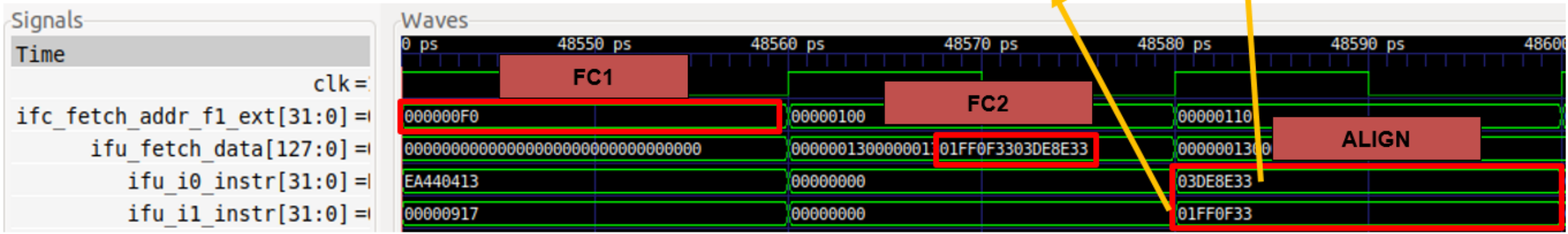
RVfpga實驗11：範例程式

```
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, ...)
add x30, x30, x31      # x30 = 4+1 = 5 (later iterations: 5+1=6, ...)
INSERT_NOPS_10
add x29, x29, 1        # x29 = x29 + 1
INSERT_NOPS_10
beq zero, zero, REPEAT # Repeat the loop
```

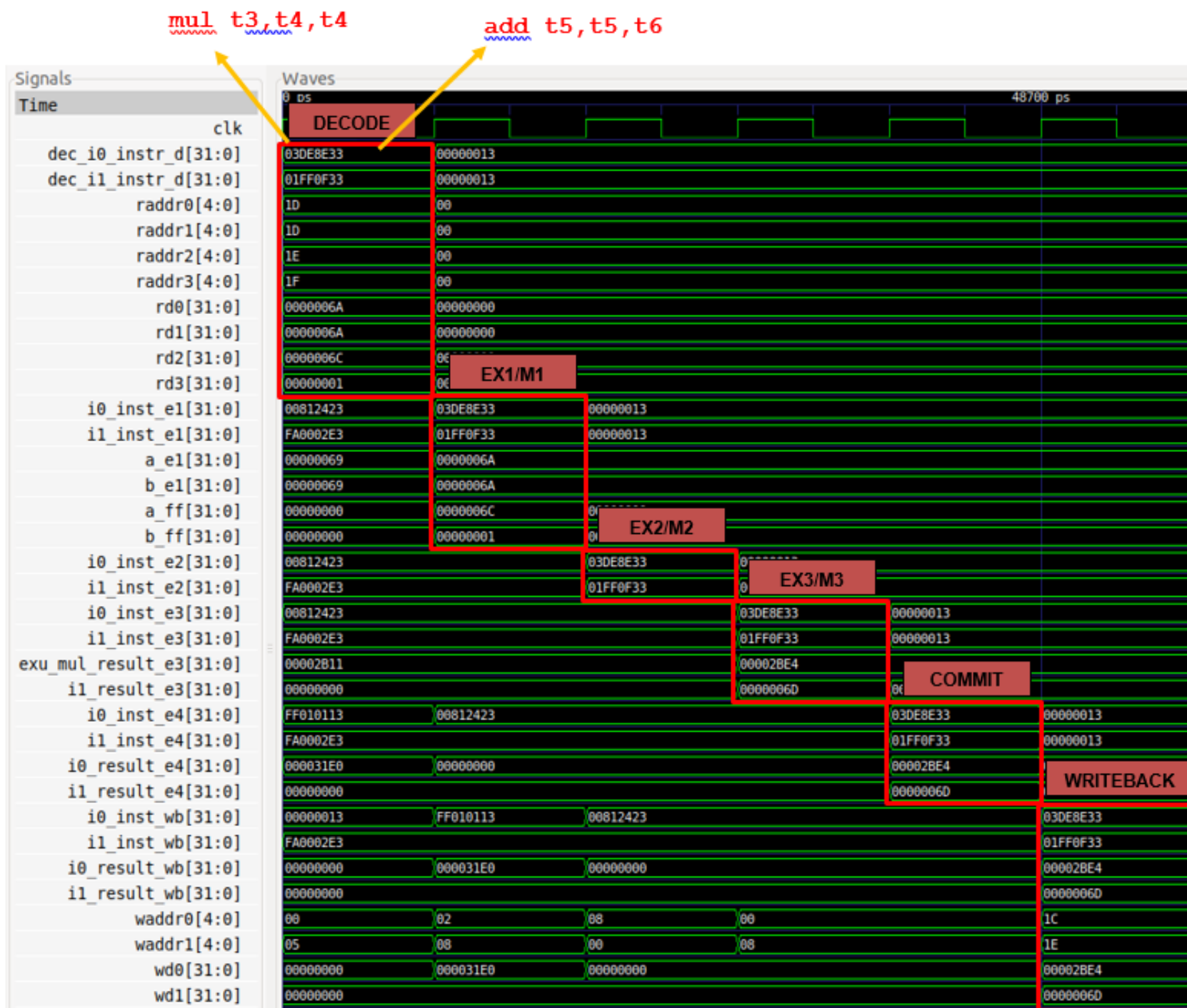
RVfpga實驗11：FC1、FC2和對齊階段模擬



RVfpga實驗11：模擬分析

- **FC1**：計算mul指令的位址：
 - `ifc_fetch_addr_f1_ext = 0x000000F0`
- **FC2**：從指令記憶體中的128位元指令束中擷取兩條指令（以紅色顯示）：
 - `ifu_fetch_data = 0x000000130000001301FF0F3303DE8E33`
- **對齊**：擷取兩條指令並分發到SweRV EH1的兩個通路。
 - 通路0：`ifu_i0_instr = 0x03DE8E33`（mul指令）
 - 通路1：`ifu_i1_instr = 0x01FF0F33`（add指令）

RVfpga實驗11：解碼、EX1/2/3、提交和WB階段模擬



RVfpga實驗11：模擬分析

- **解碼**：從暫存器檔案中讀取指令的運算元，然後提供給乘法管道和I1管道。
- **EX1/2/3和提交**：計算加法和乘法。
 - $i0_result_e4 = exu_mul_result_e3 = 0x6A * 0x6A = 0x2BE4$
 - $i1_result_e4 = i1_result_e3 = 0x6C + 0x01 = 0x6D$
- **寫回**：將結果寫回到暫存器檔案中。
 - $waddr0 = 0x1C \quad wd0 = 0x2BE4$
 - $waddr1 = 0x1E \quad wd1 = 0x6D$

RVfpga實驗11：硬體計數器

- 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，用於記錄表中所示的指標。

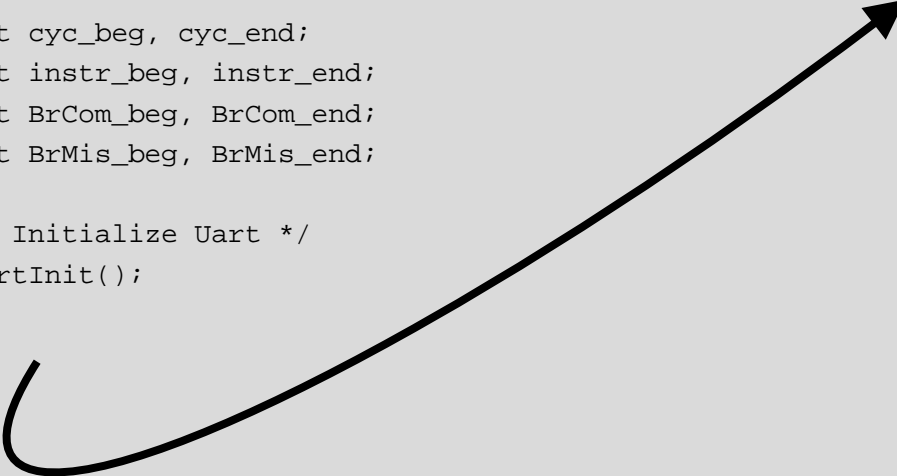
0	Reserved	17	CSR read/write	34	Cycles SB/WB stalled
1	Cycles clock active	18	CSR write rd==0	35	Cycles DMA DCCM transaction stalled
2	I-Cache hits	19	Ebreak	36	Cycles DMA ICCM transaction stalled
3	I-Cache misses	20	Ecall	37	Exceptions taken
4	Instrs committed	21	Fence	38	Timer interrupts taken
5	Instrs committed 16-b	22	Fence.i	39	External interrupts taken
6	Instrs committed 32-b	23	Mret	40	TLU flushes
7	Instrs aligned	24	Branches committed	41	Branch error flushes
8	Instrs decoded	25	Branches mispredicted	42	I-bus transactions – instr
9	Muls committed	26	Branches taken	43	D-bus transactions – ld/st
10	Divs committed	27	Unpredictable branches	44	D-bus transactions misaligned
11	Loads committed	28	Cycles fetch stalled	45	I-bus errors
12	Stores committed	29	Cycles aligner stalled	46	D-bus errors
13	Misaligned loads	30	Cycles decode stalled	47	Cycles stalled due to I-bus busy
14	Misaligned stores	31	Cycles postsync stalled	48	Cycles stalled due to D-bus busy
15	Alus committed	32	Cycles presync stalled	49	Cycles interrupts disabled
16	CSR read	33	Cycles frozen	50	Cycles interrupts stalled while disabled

RVfpga實驗11：透過Western Digital的PSP使用效能計數器

```
#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);
}
```

實驗12： 算術/邏輯指令： add指令



RVfpga實驗12：簡介

- 本實驗將分析SweRV EH1管線中的算術和邏輯指令流，重點關注add指令。
- 分為兩部分：
 - add指令的基本分析
 - add指令的進階分析
- 這兩種分析使用相同的範例程式，如下一張幻燈片所示。

RVfpga實驗12：範例程式

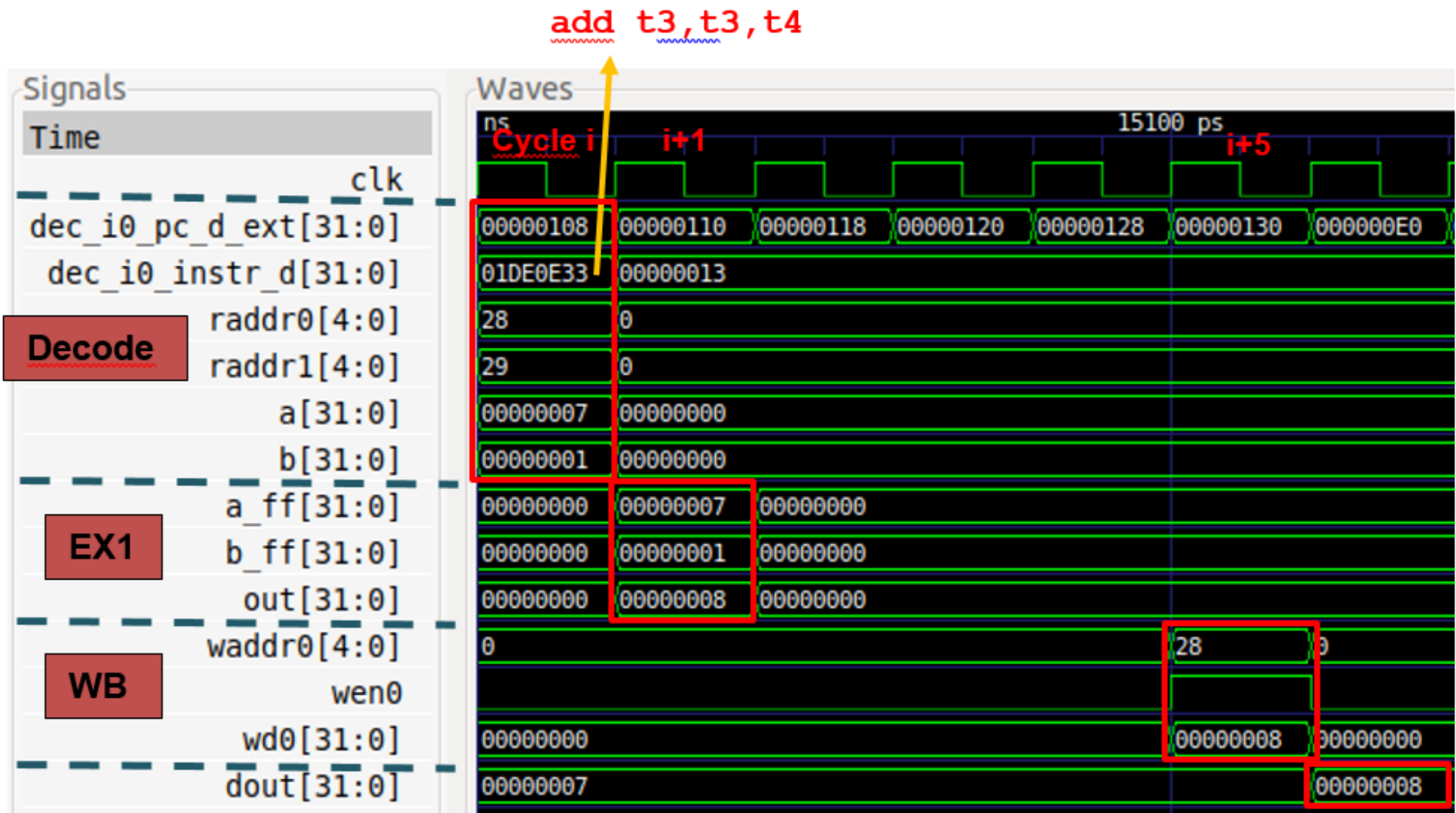
```
.globl main
main:

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

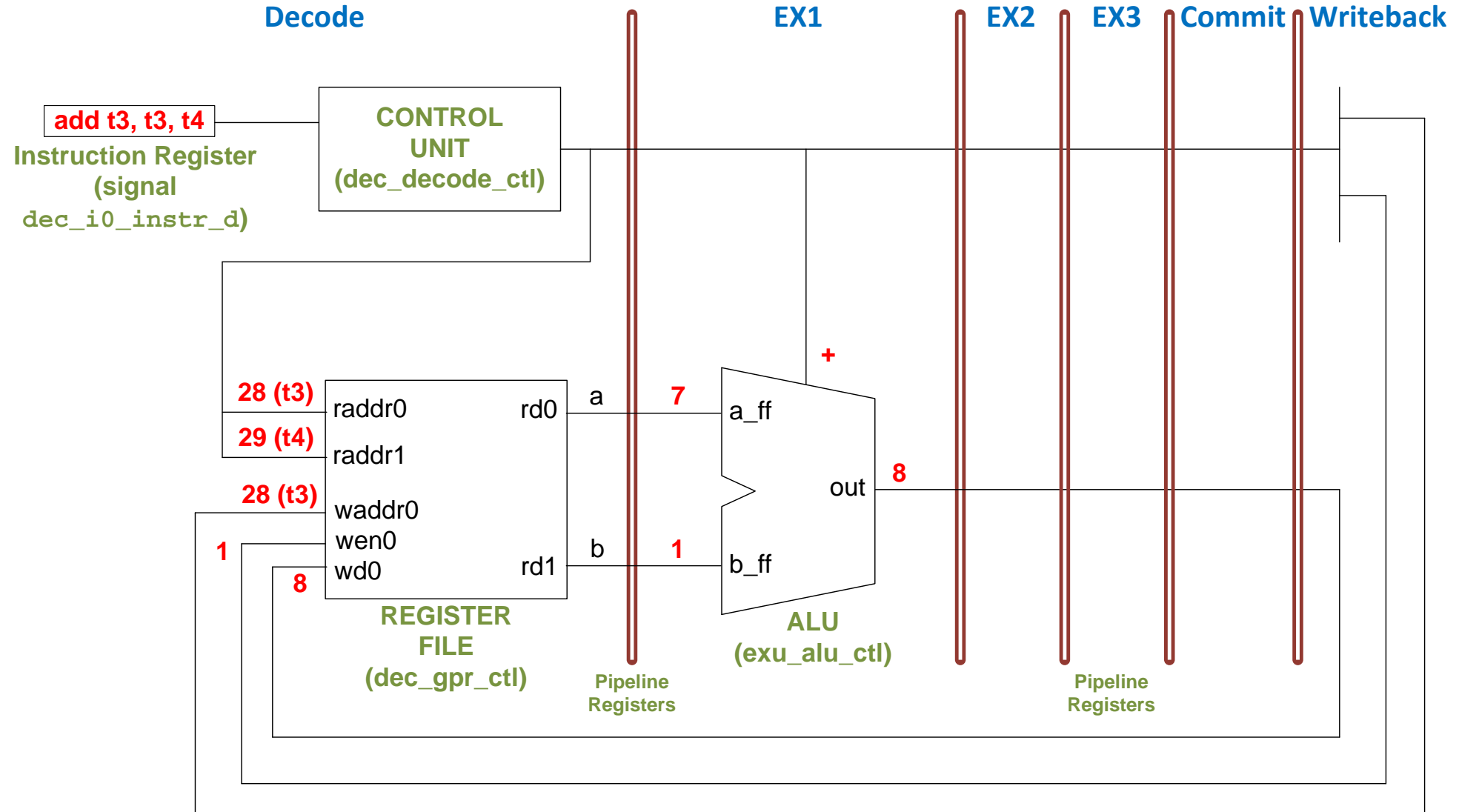
REPEAT:
    INSERT_NOPS_10
    add t3, t3, t4     # t3 = t3 + t4
    INSERT_NOPS_10
    beq zero, zero, REPEAT # Repeat the loop

.end
```

RVfpga實驗12：基本分析 - 模擬



RVfpga實驗12：基本分析 – SweRV EH1管線



RVfpga實驗12：基本分析 - 模擬

- **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指令的欄位為：
00 | rs1 | 000
| rd | 0110011
在這一階段，將產生控制訊號並讀取暫存器檔案。此外，運算元將傳播到IO管道。
- **週期i+1：** **EX1：**執行add指令。將加法的結果作為ALU的輸出（訊號out = 8）。
- **週期i+5：** 寫回：將加法的結果寫回到暫存器檔案中：wd0 = 0x8、wen0 = 1且waddr0 = 0x28

RVfpga實驗12：進階分析

- 下一頁的圖為add指令遍歷IO管道的詳細圖。

Decode

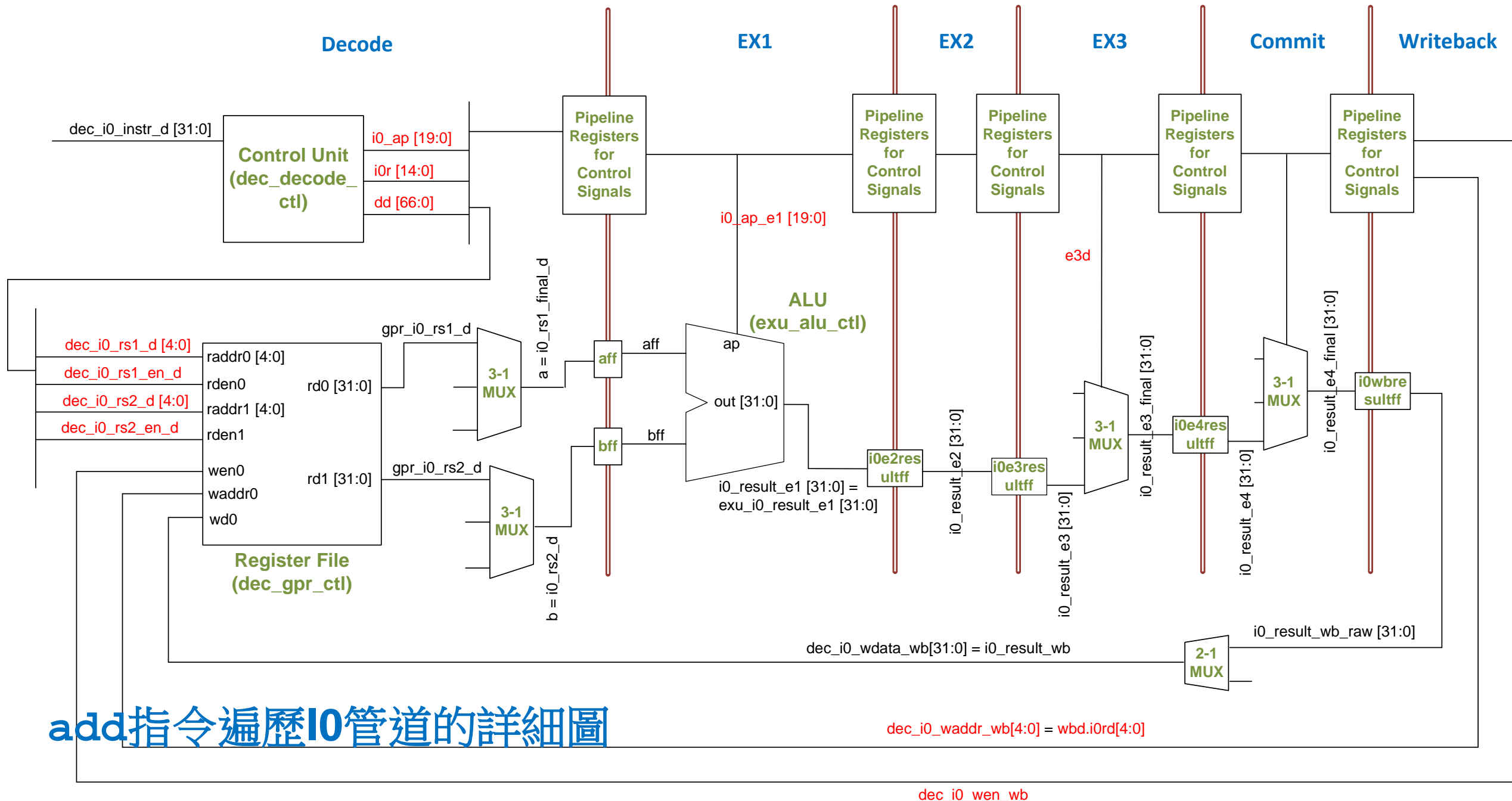
EX1

EX2

EX3

Commit

Writeback



add指令遍历IO管道的详细图

實驗13： 記憶體存取指令： lw和sw指令



RVfpga實驗13：簡介

- 實驗13將分析記憶體讀寫。
- 分為三部分：
 - 低延遲載入：檢查讀取低延遲DCCM（不暫停處理器）時的載入/儲存管道。
 - 低延遲儲存：檢查DCCM的儲存。
 - 高延遲載入和儲存：在讀/寫Nexys A7開發板上的DDR主記憶體時重複之前的分析。

RVfpga實驗13：載入－範例程式

```
.globl main
```

```
.section .midccm
```

```
A: .space 8
```

```
.text
```

```
main:
```

```
# Register t3 = x28 (register 28)
```

```
la t0, A          # t0 = addr(A)
```

```
li t1, 0x2        # t1 = 2
```

```
sw t1, (t0)       # A[0] = 2
```

```
add t1, t1, 6     # t1 = 8
```

```
sw t1, 4(t0)      # A[1] = 8
```

```
INSERT_NOPS_9
```

```
REPEAT:
```

```
INSERT_NOPS_1
```

```
lw t1, (t0)
```

```
INSERT_NOPS_9
```

```
INSERT_NOPS_4
```

```
lw t1, 4(t0)
```

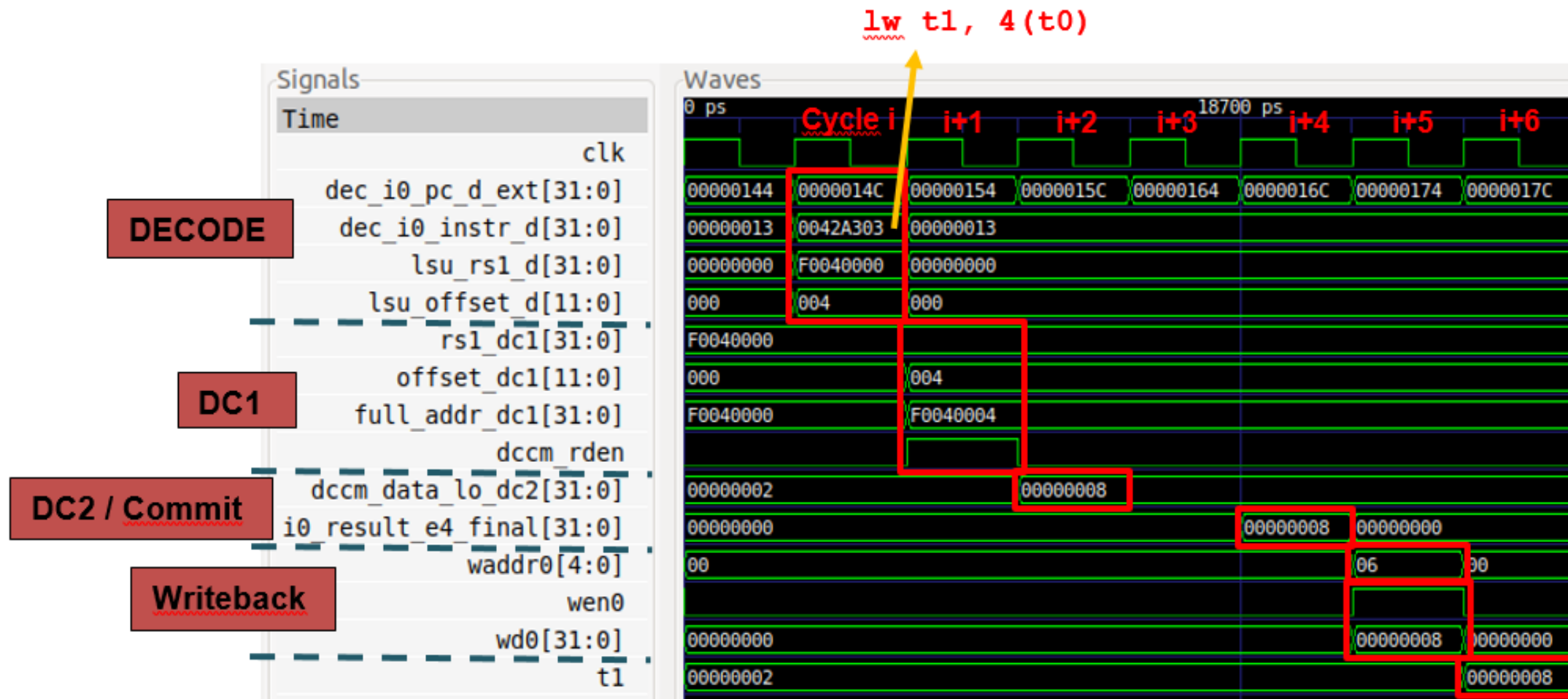
```
INSERT_NOPS_10
```

```
INSERT_NOPS_4
```

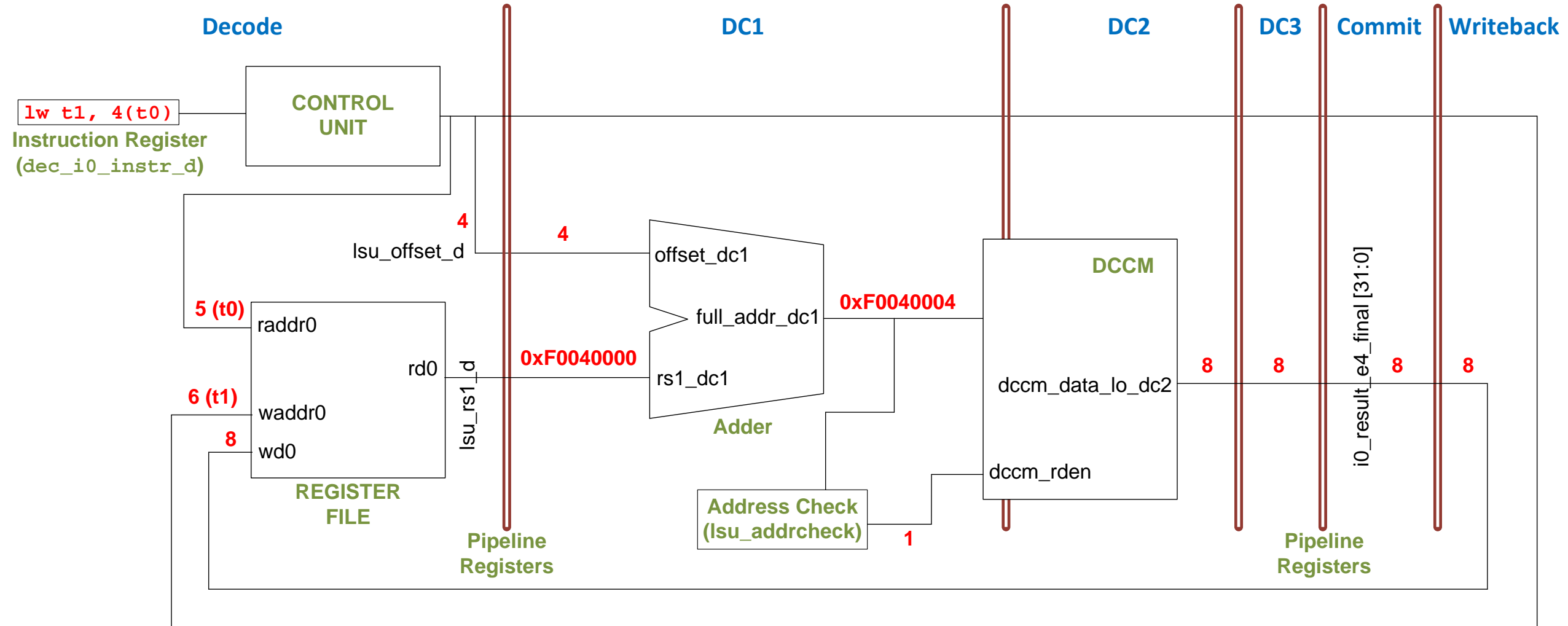
```
beq zero, zero, REPEAT # Repeat the loop
```

```
.end
```

RVfpga實驗13：低延遲載入－模擬



RVfpga實驗13：低延遲載入 – SweRV EH1管線

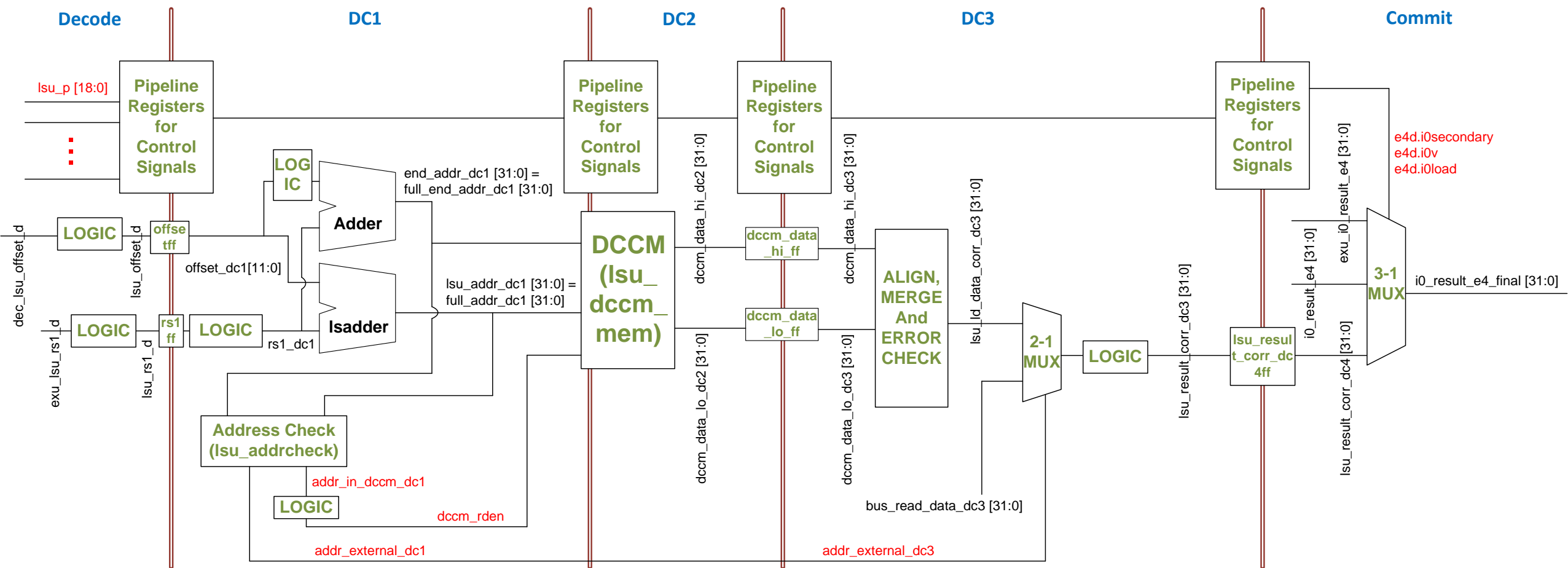


RVfpga實驗13：低延遲載入－分析

- **週期i：** 解碼：產生控制訊號並讀取運算元：
 - $t0 = 0xF0040000$
 - $Offset = 0x004$
- **週期i+1：** **DC1：**
 - 計算位址： $full_addr_dc1 = 0xF0040004$
 - 尋找存取的記憶體區域 → $dccm_rden$ 置為有效
- **週期i+2：** **DC2：** 讀取DCCM → $dccm_data_lo_dc2 = 0x8$
- **週期i+5：** 寫回：將從記憶體讀取的值寫回到暫存器檔案：
 - $wd0 = 0x8$
 - $wen0 = 1$
 - $waddr0 = 0x6$

RVfpga實驗13：高延遲載入分析

- 下一張幻燈片上的圖為1w指令透過IO管道執行期間遍歷的主要元素的詳細圖。
- 實驗11已提供相關說明，但下面的新圖只關注LSU管道，提供與1w指令相關的詳細資訊。
- 實驗13的文件提供下圖中1w指令執行期間每個階段的深入說明（此處不包含）。



1w 遍歷 IO 管道的詳細圖

RVfpga實驗13：儲存－範例程式

```
.globl main
```

```
.section .midccm
```

```
A: .space 4000
```

```
.text
```

```
main:
```

```
la t0, A                # t0 = addr(A)
```

```
li t1, 0x2              # t1 = 2
```

```
li t2, 1000             # t2 = 1000
```

```
INSERT_NOPS_2
```

```
REPEAT:
```

```
sw t1, (t0)
```

```
INSERT_NOPS_10
```

```
INSERT_NOPS_4
```

```
lw t1, (t0)
```

```
INSERT_NOPS_10
```

```
add t1,t1,t1
```

```
add t0,t0,0x04
```

```
add t2,t2,-1
```

```
INSERT_NOPS_10
```

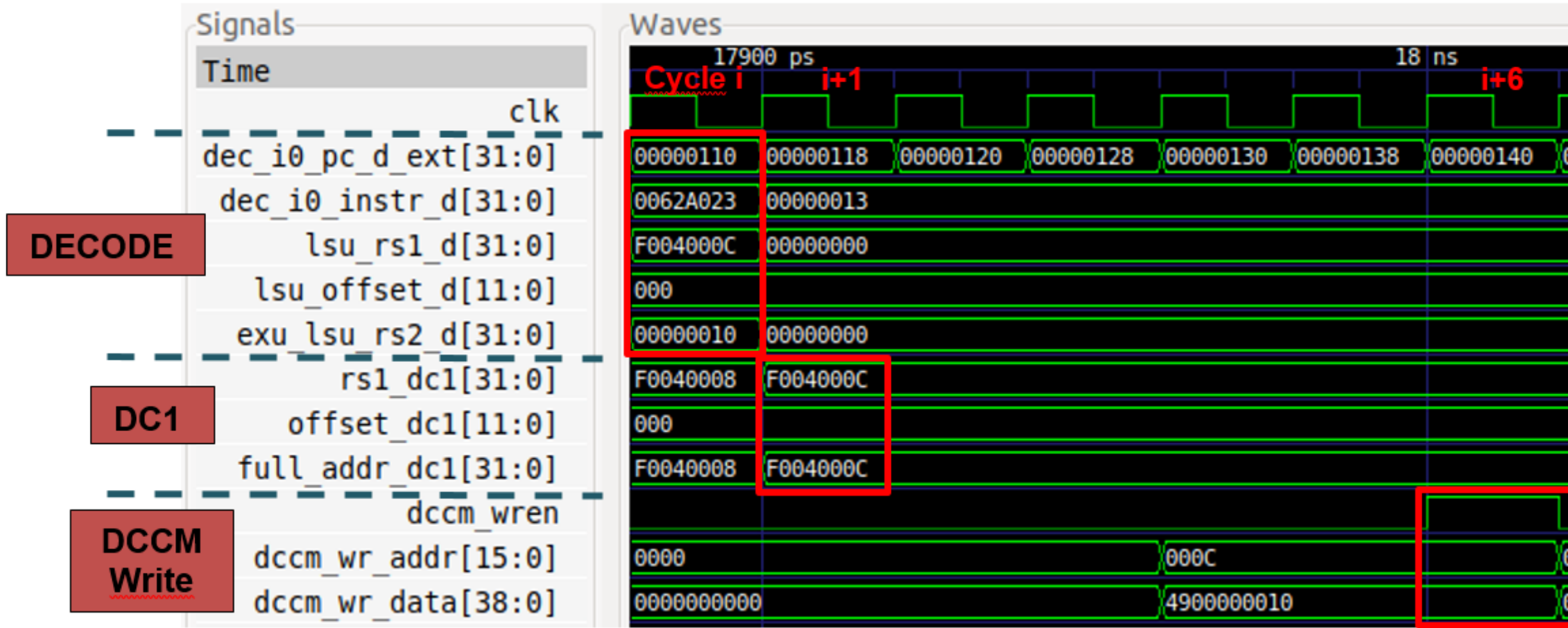
```
bne t2, zero, REPEAT    # Repeat the loop
```

```
nop
```

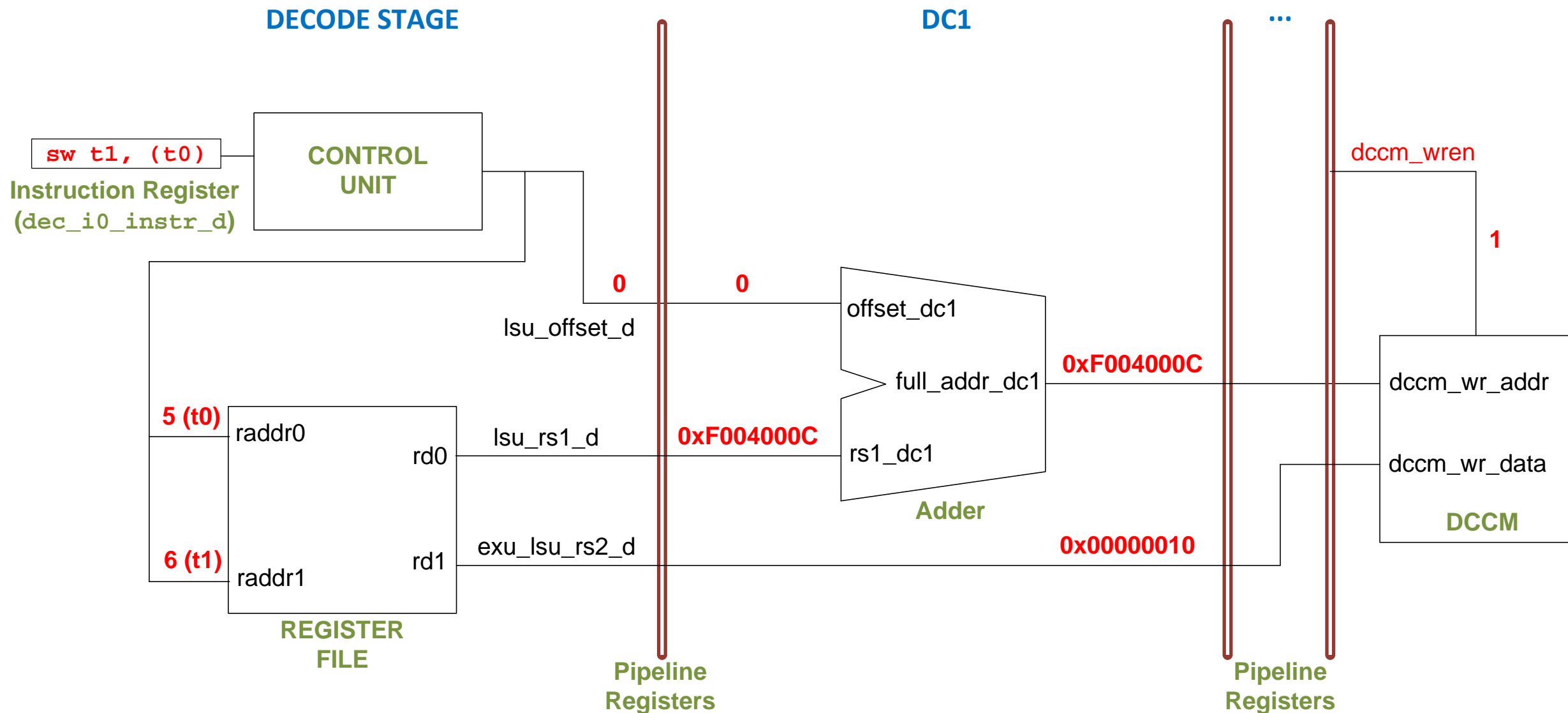
```
nop
```

```
.end
```

RVfpga實驗13：低延遲儲存－模擬



RVfpga實驗13：低延遲儲存 – SweRV EH1管線



RVfpga實驗13：儲存基本分析 - 模擬

- **週期i：解碼：產生控制訊號並讀取運算元：**

- $t0 = 0xF004000C$
- $Offset = 0x000$
- $t1 = 0x10$

- **週期i+1：DC1：**

- 計算位址： $full_addr_dc1 = 0xF004000C$

- **週期i+6：DCCM寫入：**

- $dccm_wr_addr = 0x000C$
- $dccm_wr_data = 0x10$

RVfpga實驗13：外部記憶體載入

- 下一張幻燈片上的圖顯示了lw指令為讀取主記憶體而遍歷的主要路徑。
- 處理器必須暫停來等待來自外部記憶體的資料。
- 外部記憶體透過AXI匯流排存取，該匯流排為Lite DRAM控制器提供位址，然後在幾個週期後將要求的資料對齊並傳送到DC3階段。
- DC3階段的2選1多路開關選擇來自外部記憶體的資料，而不是來自DCCM的資料。

DC1 STAGE

Delay due to
accessing External
Memory

DC3 STAGE

COMMIT STAGE

Pipeline
Registers
for
Control
Signals

Pipeline
Registers
for
Control
Signals

Pipeline
Registers
for
Control
Signals

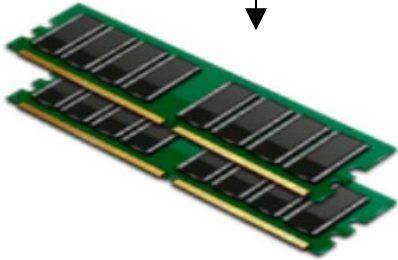
end_addr_dc1 [31:0] =
full_end_addr_dc1 [31:0]

lsu_addr_dc1 [31:0] =
full_addr_dc1 [31:0]

External Memory
accessed through AXI
Bus
(lsu_bus_intf)

Lite DRAM
Controller

addr_external_dc1



bus_read_data_dc3 [31:0]

lsu_ld_data_corr_dc3 [31:0]

2-1
MUX

LOGIC

lsu_result_corr_dc3 [31:0]

lsu_resul
t_corr_dc
4ff

i0_result_e4 [31:0]

exu_i0_result_e4 [31:0]

lsu_result_corr_dc4 [31:0]

3-1
MUX

e4d.i0secondary
e4d.i0v
e4d.i0load

i0_result_e4_final [31:0]

RVfpga實驗13：外部記憶體－範例程式

```
.globl main
```

```
.data
```

```
D: .word 3,5,6,8,7,10,12,2,1,4,11,9
```

```
.text
```

```
main:
```

```
li t2, 0x020
```

```
csrrs t1, 0x7F9, t2
```

```
la t4, D
```

```
li t5, 12
```

```
li t6, 0x0
```

```
INSERT_NOPS_1
```

```
REPEAT:
```

```
lw t3, (t4)
```

```
add t5, t5, -1
```

```
INSERT_NOPS_10
```

```
add t6, t3, t6
```

```
add t4, t4, 4
```

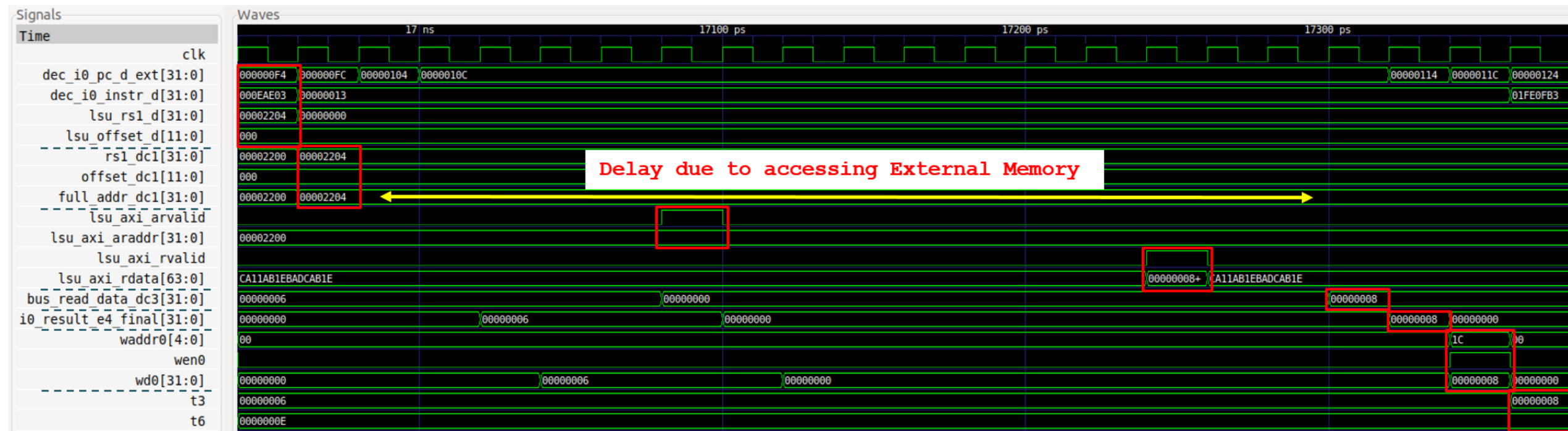
```
INSERT_NOPS_9
```

```
bne t5, zero, REPEAT # Repeat the loop
```

```
INSERT_NOPS_4
```

```
.end
```

RVfpga實驗13：外部記憶體－模擬



RVfpga實驗13：外部記憶體－分析

- 在解碼階段，範例的第四次迭代中的位址為0x00002204。
- 然後，位址透過AXI匯流排傳送到外部記憶體：
 - lsu_axi_arvalid = 1
 - lsu_axi_araddr = 0x00002200
- 幾個週期後，外部記憶體傳回透過AXI匯流排讀取的64位元資料
 - lsu_axi_rdata = 0x0000000800000006
 - lsu_axi_rvalid = 1
- 最後，從64位元資料中擷取所要求的32位元資料，插入到主管線路徑中，並寫入暫存器檔案。

實驗14： 結構冒險



RVfpga實驗14：簡介

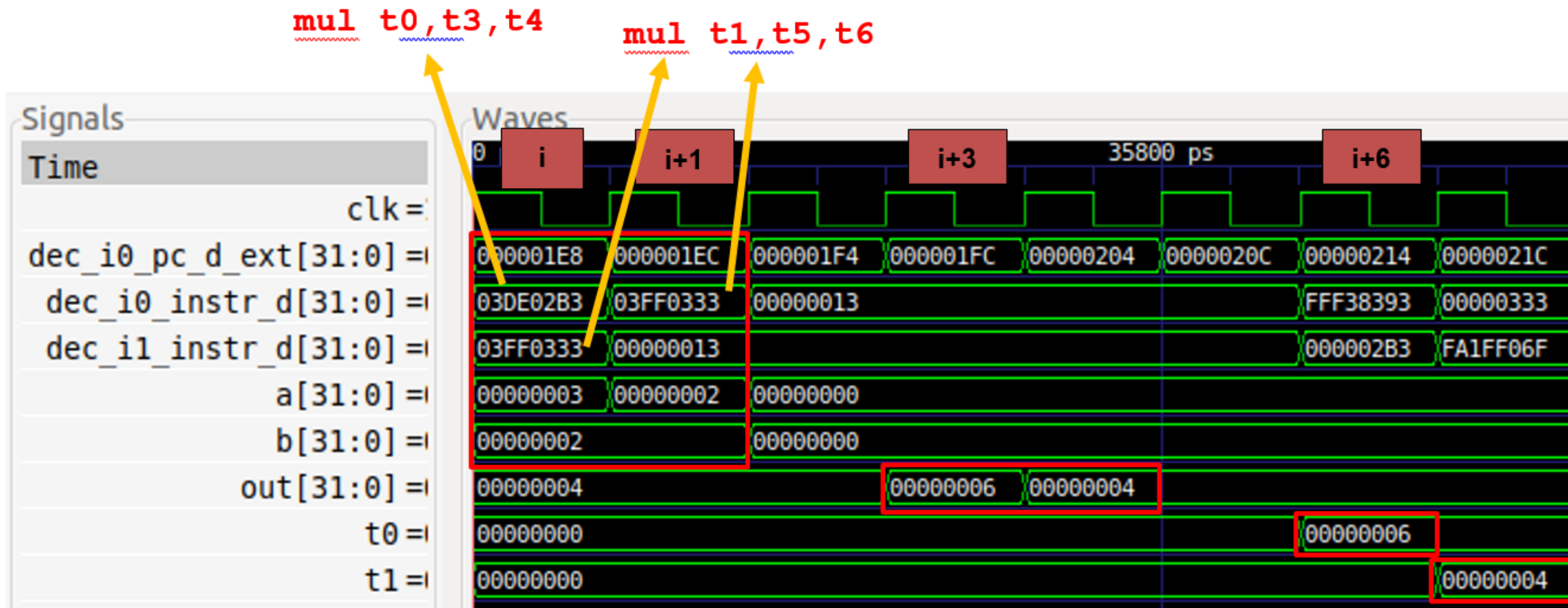
- 實驗14將說明兩種結構冒險（它們具有不同的效能-成本權衡）。
 - 單元衝突：二條mul指令在同一週期到達解碼階段。乘法器是管線的，因此第二條mul指令僅延遲一個週期。硬體成本和效能下降（僅一個週期）較低。
 - 暫存器檔案寫入連接埠衝突：三條指令在同一週期內到達寫回階段，其中之一是在幾個週期之前執行的非阻塞載入。SweRV EH1具有三個（而不是兩個）寫入連接埠。可避免結構冒險（不會導致效能損失），但由於額外的暫存器檔案連接埠，其硬體成本較高。
 - 請注意，div指令也可能導致冒險（在實驗的附錄中進行討論）。

RVfpga實驗14：2條mul指令 - 範例程式

```
.globl Test_Assembly
Test_Assembly:
li t2, 0xFFFF
li t3, 0x3
li t4, 0x2
li t5, 0x2
li t6, 0x2
REPEAT:
    beq t2, zero, OUT    # Stay in the loop?
    INSERT_NOPS_9
    mul t0, t3, t4        # t0 = t3 * t4
    mul t1, t5, t6        # t1 = t5 * t6
    INSERT_NOPS_9
    add t2, t2, -1
    add t0, zero, zero
    add t1, zero, zero
    j REPEAT
OUT:
.end
```



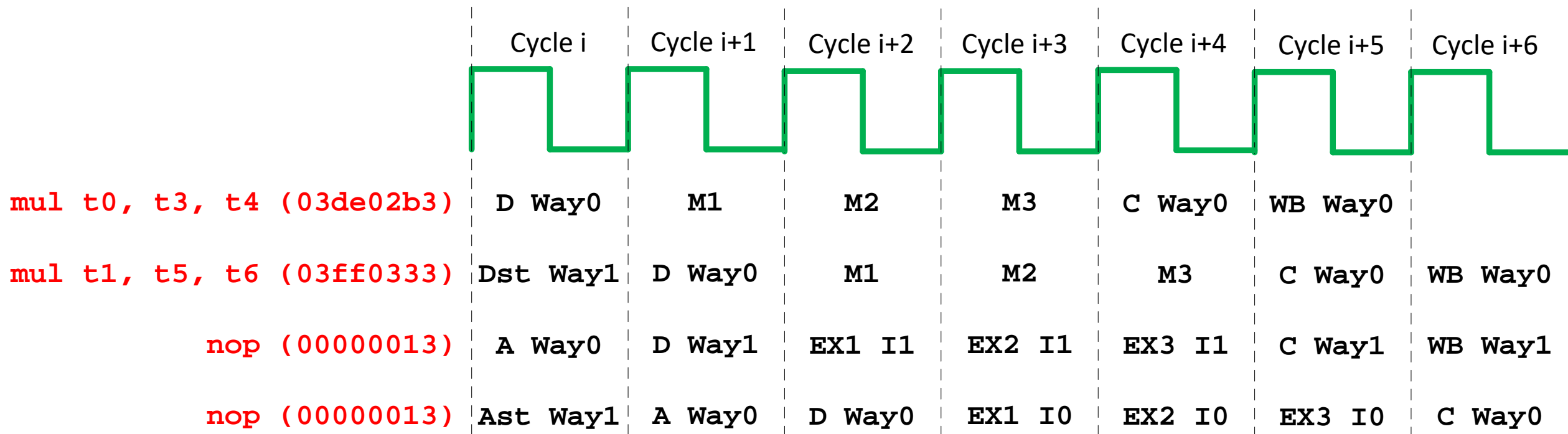
RVfpga實驗14：2條mul指令 - 模擬



RVfpga實驗14：2條mul指令 - 分析

- **週期i**：兩條mul指令在同一週期到達解碼階段。結構冒險將阻止第二條mul指令進行。
- **週期i+1**：第一條mul指令在管線乘法器的第一階段（M1）執行，而第二條mul指令在解碼階段等待。
- **週期i+2**：第一條mul指令在管線乘法器的第二階段（M2）執行，而第二條mul指令在第一階段（M1）執行。
- **週期i+3**：第一條mul指令取得結果：`out = 0x6`。
- **週期i+4**：第二條mul指令取得結果：`out = 0x4`。
- **週期i+6**：暫存器檔案使用第一條mul指令的結果（`t0 = 0x6`）更新。
- **週期i+7**：暫存器檔案使用第二條mul指令的結果（`t1 = 0x4`）更新。

RVfpga實驗14：2條mu1指令 - 圖



RVfpga實驗14：3個同步寫入操作－範例程式

REPEAT:

lw x28, (x29)

add x30, x30, -1

add x1, x1, 1

add x31, x31, 1

add x3, x3, 1

add x4, x4, 1

add x5, x5, 1

add x6, x6, 1

add x7, x7, 1

add x8, x8, 1

add x9, x9, 1

add x10, x10, 1

add x11, x11, 1

add x12, x12, 1

add x13, x13, 1

add x14, x14, 1

add x15, x15, 1

add x16, x16, 1

add x17, x17, 1

add x18, x18, 1

add x19, x19, 1

add x20, x20, 1

add x21, x21, 1

add x22, x22, 1

add x23, x23, 1

add x24, x24, 1

add x25, x25, 1

add x26, x26, 1

add x27, x27, 1

add x31, x31, 1

add x3, x3, 1

add x4, x4, 1

add x5, x5, 1

add x6, x6, 1

add x25, x25, 1

add x26, x26, 1

add x27, x27, 1

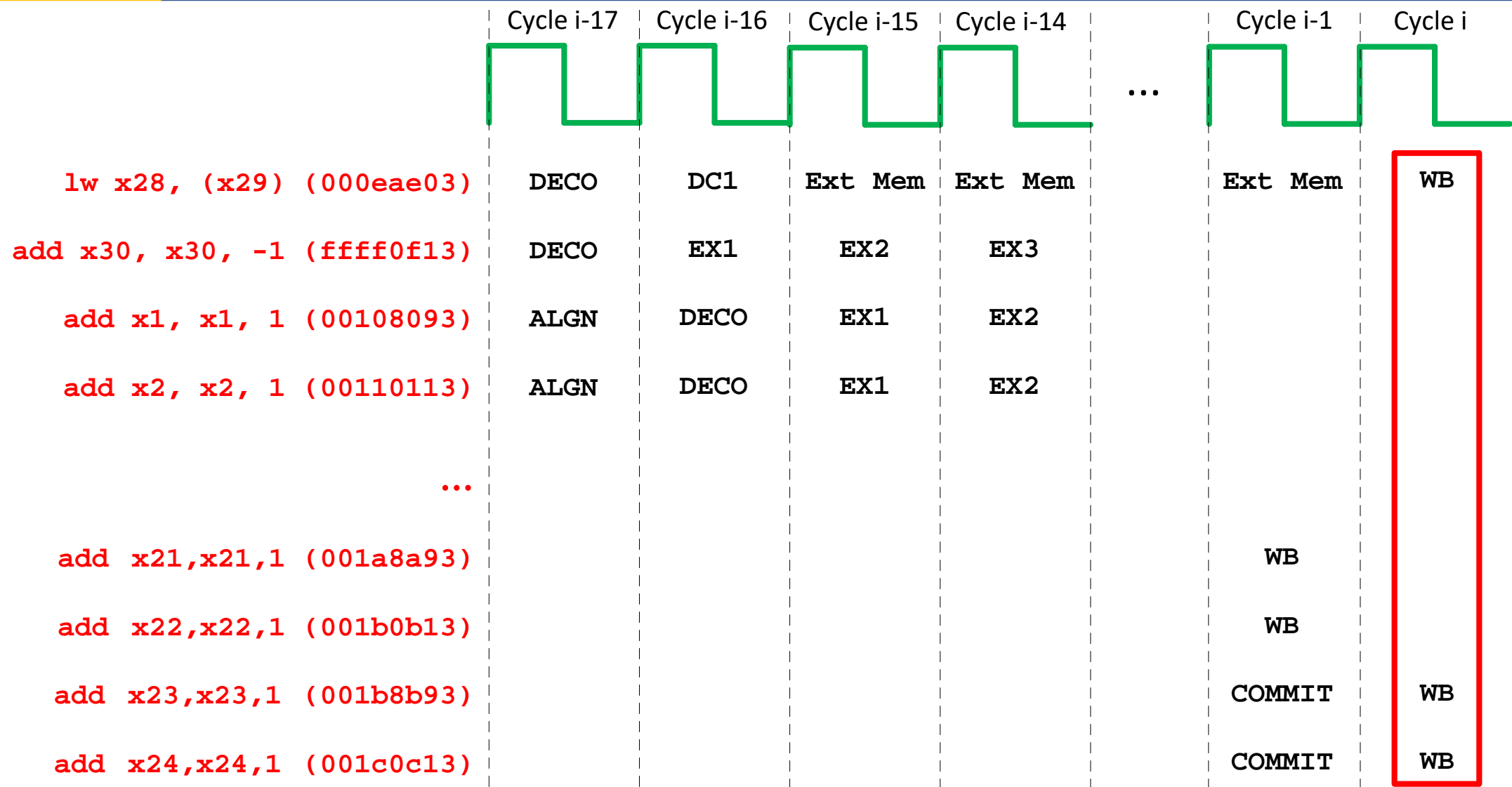
bne x30, zero, REPEAT



RVfpga實驗14：3個同步寫入操作 – 分析

- **週期i-17**：lw指令處於解碼階段。
- **週期i-16**：計算有效記憶體位址並透過AXI匯流排傳送到外部記憶體。lw指令等待幾個週期，以便外部記憶體提供資料。
- **週期i-5**：對兩條衝突的add指令進行解碼。
- **週期i**：lw指令和兩條衝突的add指令進入寫回階段（寫入暫存器檔案），由於暫存器檔案具有三個寫入連接埠，因此可以實現。

RVfpga實驗14：3個同步寫入操作 - 圖



實驗15： 資料冒險



RVfpga實驗15：簡介

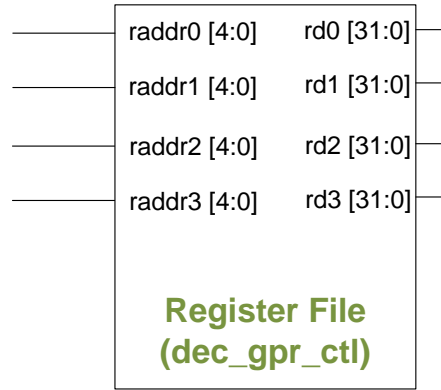
- 實驗15將分析如何解除**RAW**資料冒險。
- 可透過以下方式解除**RAW**資料冒險：**暫停**處理器或**轉送**（也稱為旁路）來自後面階段執行的指令的值。
- 兩種情況分析：
 - 透過**轉送**到解碼階段（使用幾個新的多路開關）解除**RAW**資料冒險
 - 使用兩個額外的**ALU**在提交階段解除**RAW**資料冒險

RVfpga實驗15：透過轉送解除資料冒險

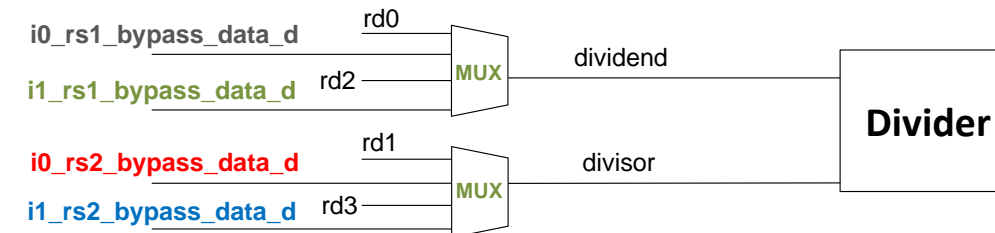
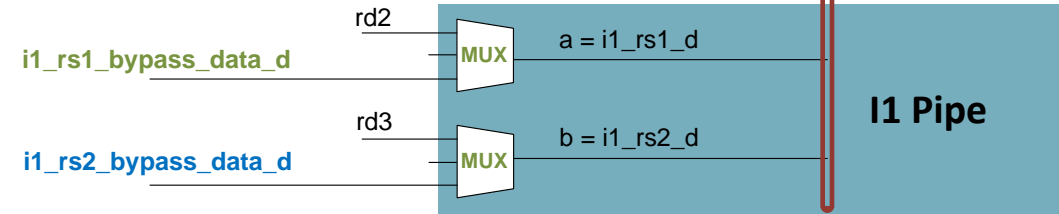
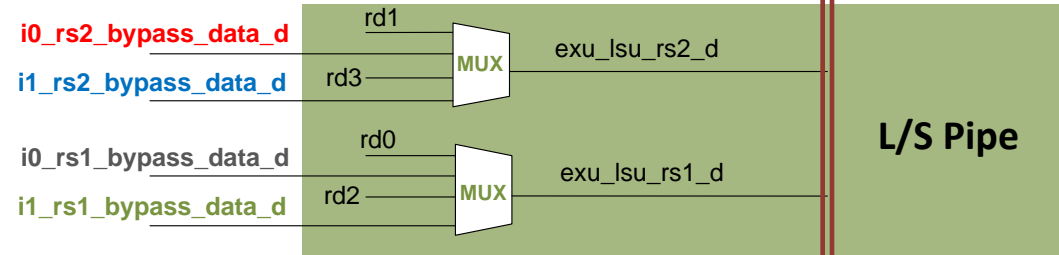
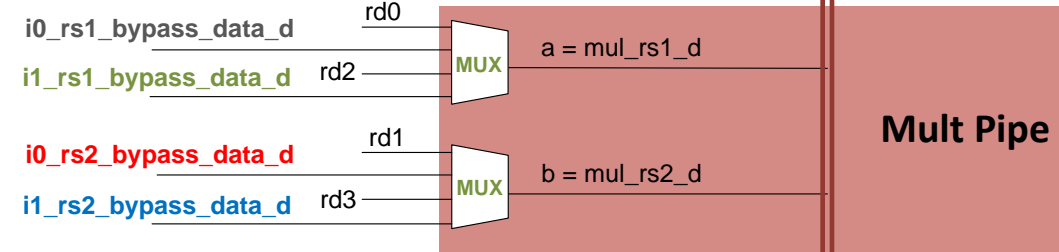
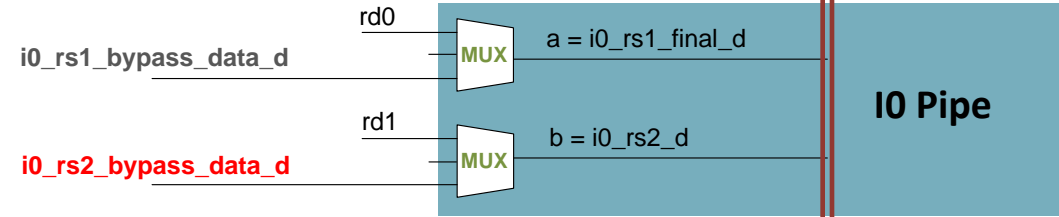
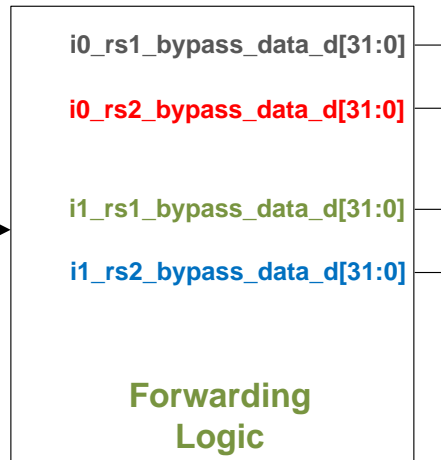
- 轉送到解碼階段需要在功能單元（ALU、乘法器、計算DC1中的有效位址的加法器等）前面新增多路開關，以從暫存器檔案或後續階段選擇運算元。
- 下一張幻燈片上的圖顯示了解碼階段的轉送值。轉送邏輯為每個通路中的兩個來源運算元中的每一個產生旁路

DECODE STAGE

EX1/DC1/M1



From subsequent stages



RVfpga實驗15：透過轉送解除資料冒險－範例

```
.globl Test_Assembly
.text

Test_Assembly:

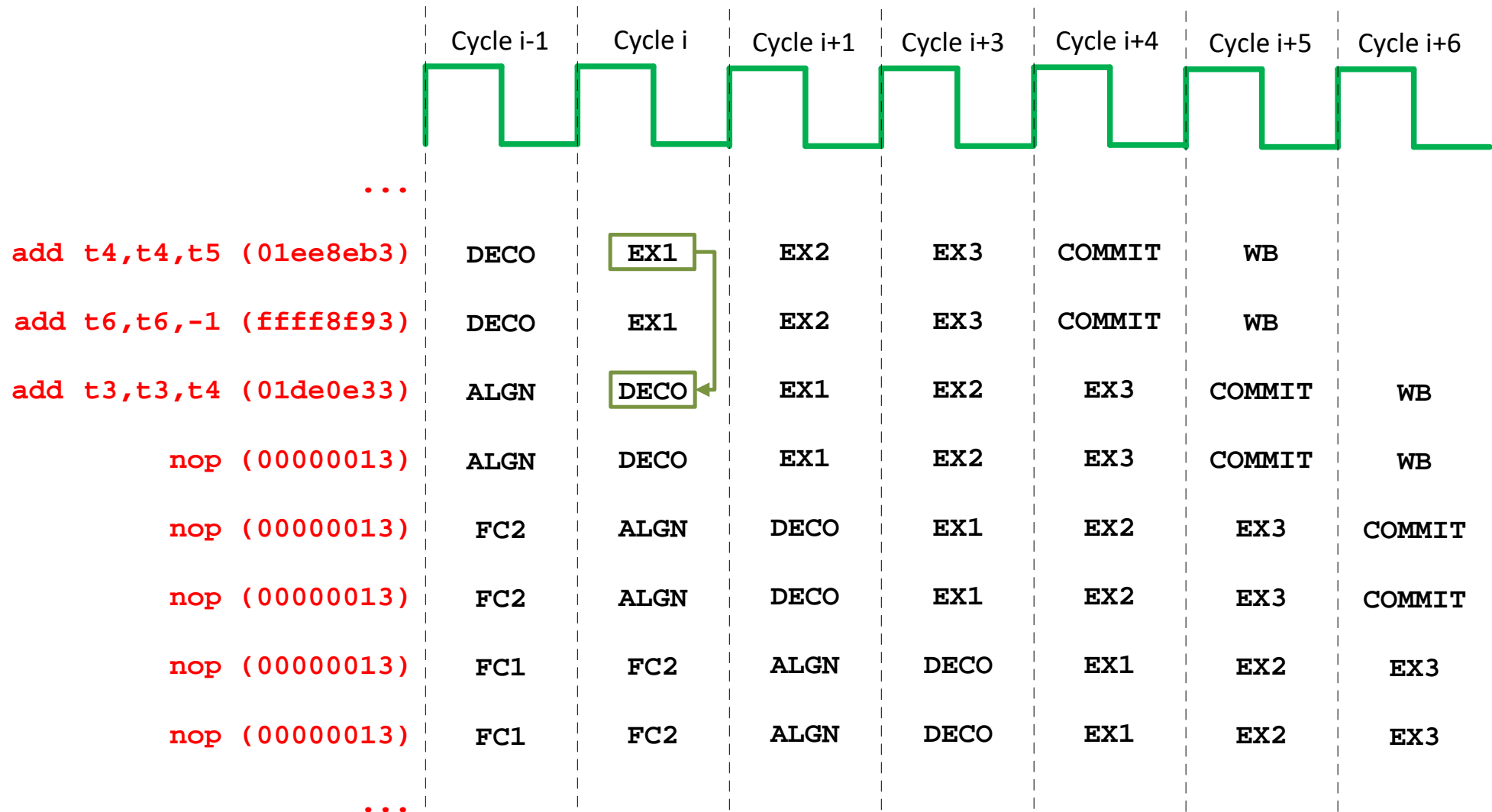
li t3, 0x3
li t4, 0x2
li t5, 0x1
li t6, 0xFFFF

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

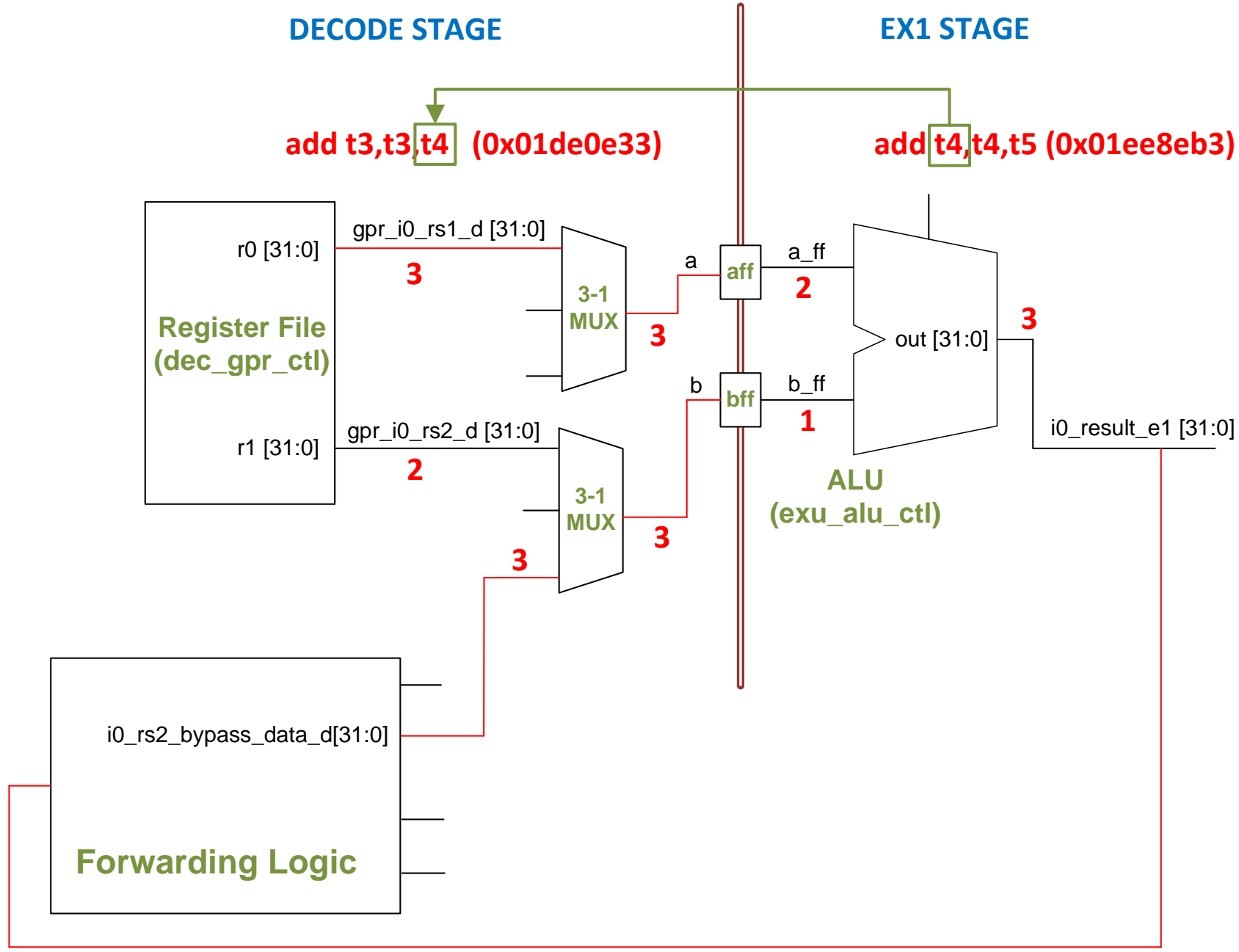
.end
```



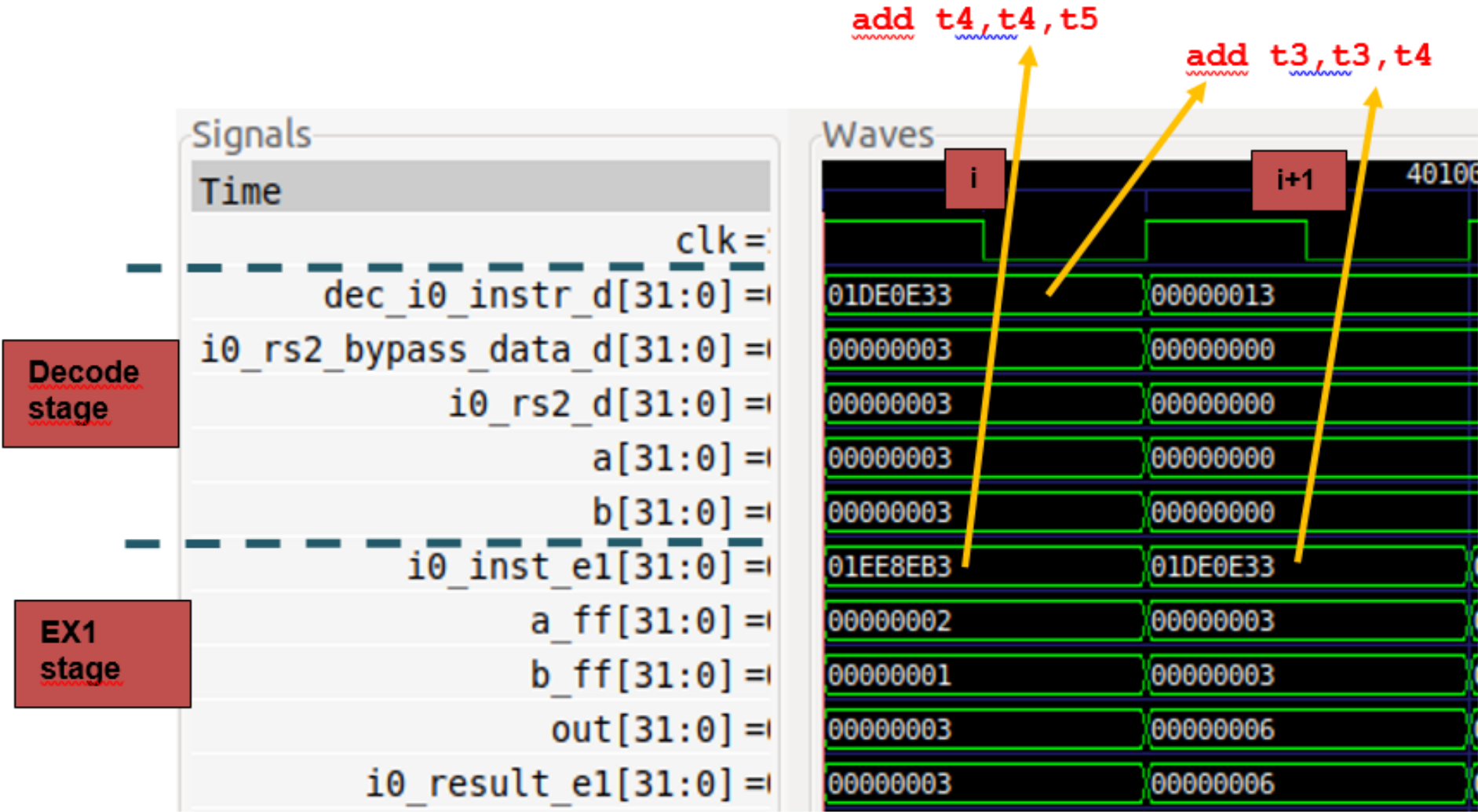
RVfpga實驗15：透過轉送解除資料冒險－圖



RVfpga實驗15：透過轉送解除資料冒險－管線

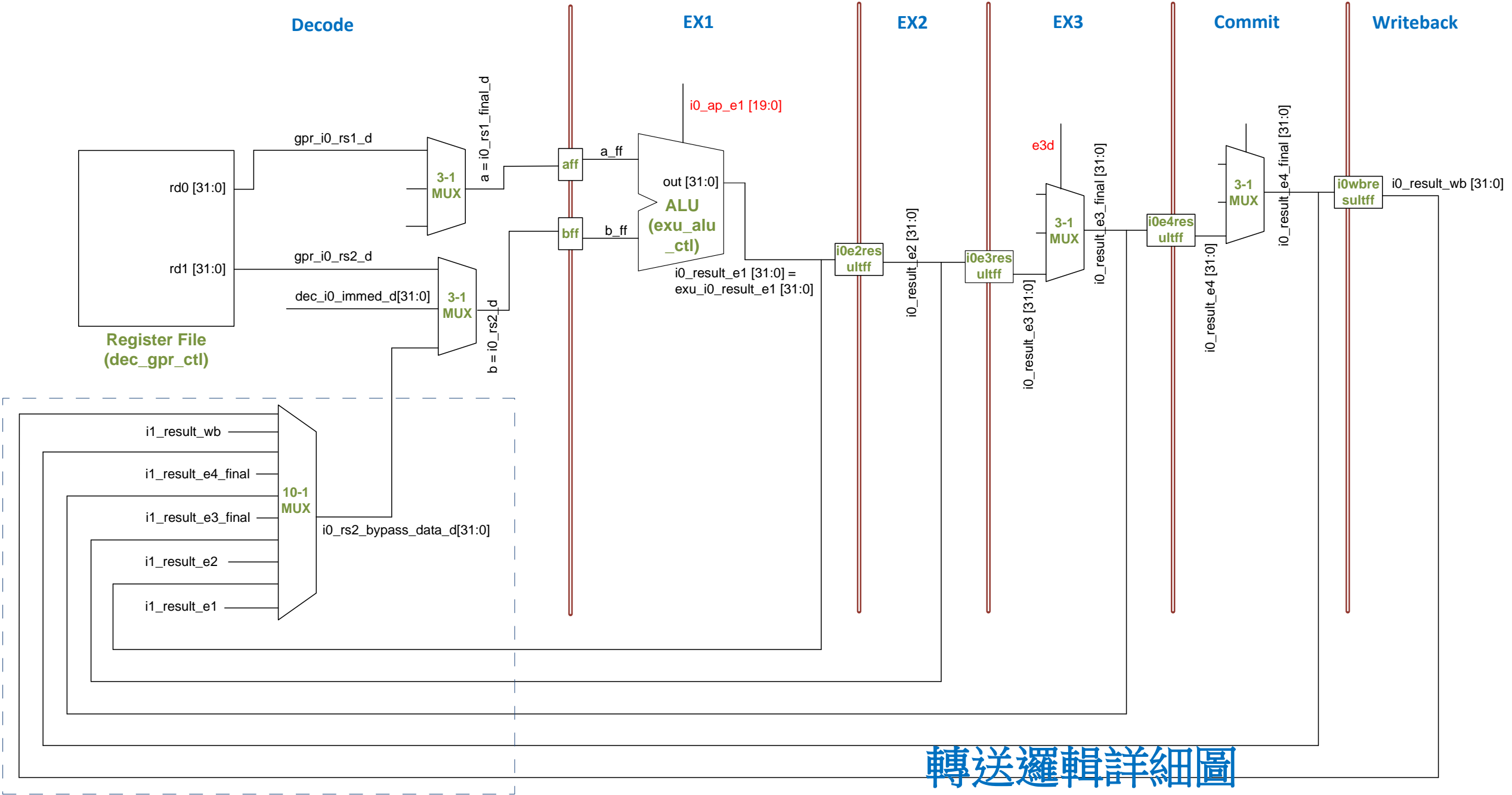


RVfpga實驗15：透過轉送解除資料冒險－模擬



RVfpga實驗15：透過轉送解除資料冒險－分析

- 指令 `add t4, t4, t5 (0x01ee8eb3)` :
 - **週期*i***：此add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：
 - $a_ff(2) + b_ff(1) = out(3)$
 - 結果在解碼階段傳送到轉送邏輯。
- 指令 `add t3, t3, t4 (0x01de0e33)` :
 - **週期*i***：此add指令處於通路0的解碼階段 (`dec_i0_instr_d = 0x01de0e33`)。轉送邏輯將EX1階段的結果 (`i0_result_e1`) 轉送到解碼階段 (`i0_rs2_bypass_data_d`)。兩個3選1多路開關產生運算元，具體如下：
 - 運算元 $a = 3$ (來自暫存器檔案)
 - 運算元 $b = 3$ (來自I0管道EX1階段的ALU輸出，經過轉送邏輯)
 - **週期*i+1***：此add指令處於I0管道的EX1階段 (`i0_inst_e1 = 0x01de0e33`)。它在ALU中計算正確的加法：
 - $a_ff(3) + b_ff(3) = out(6)$

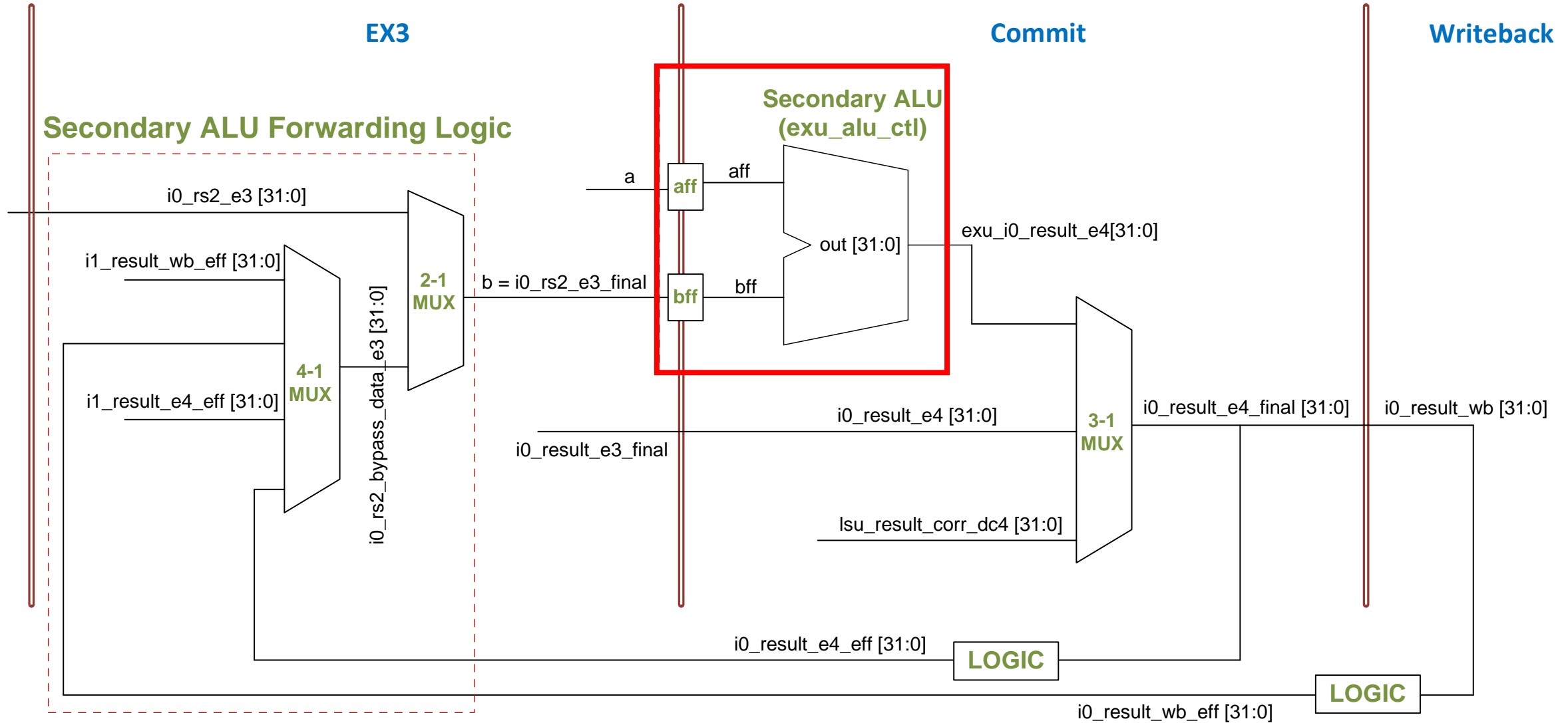


轉送邏輯詳細圖

RVfpga實驗15：透過在提交階段轉送解除資料冒險

- 需要多個週期才能取得結果的指令（即多週期操作，例如lw、mul和div）無法轉送到解碼階段。
- 但SweRV EH1在每路的提交階段都新增一個額外的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。

RVfpga實驗15：透過在提交階段轉送解除資料冒險－管線



RVfpga實驗15：透過在提交階段轉送解除資料冒險－範例

```
.globl Test_Assembly
```

```
.section .midccm
```

```
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
```

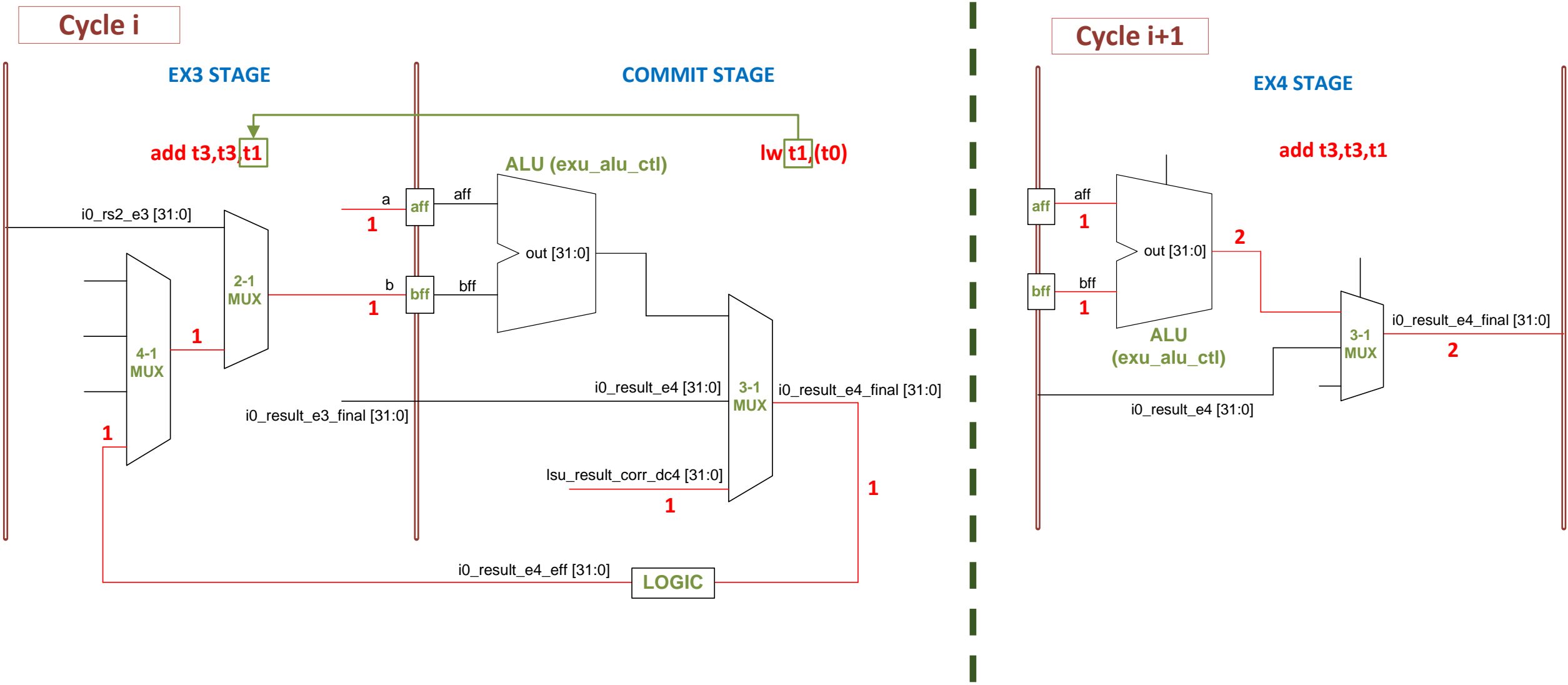
```
    add t5, t5, 0x1
```

```
j REPEAT
```

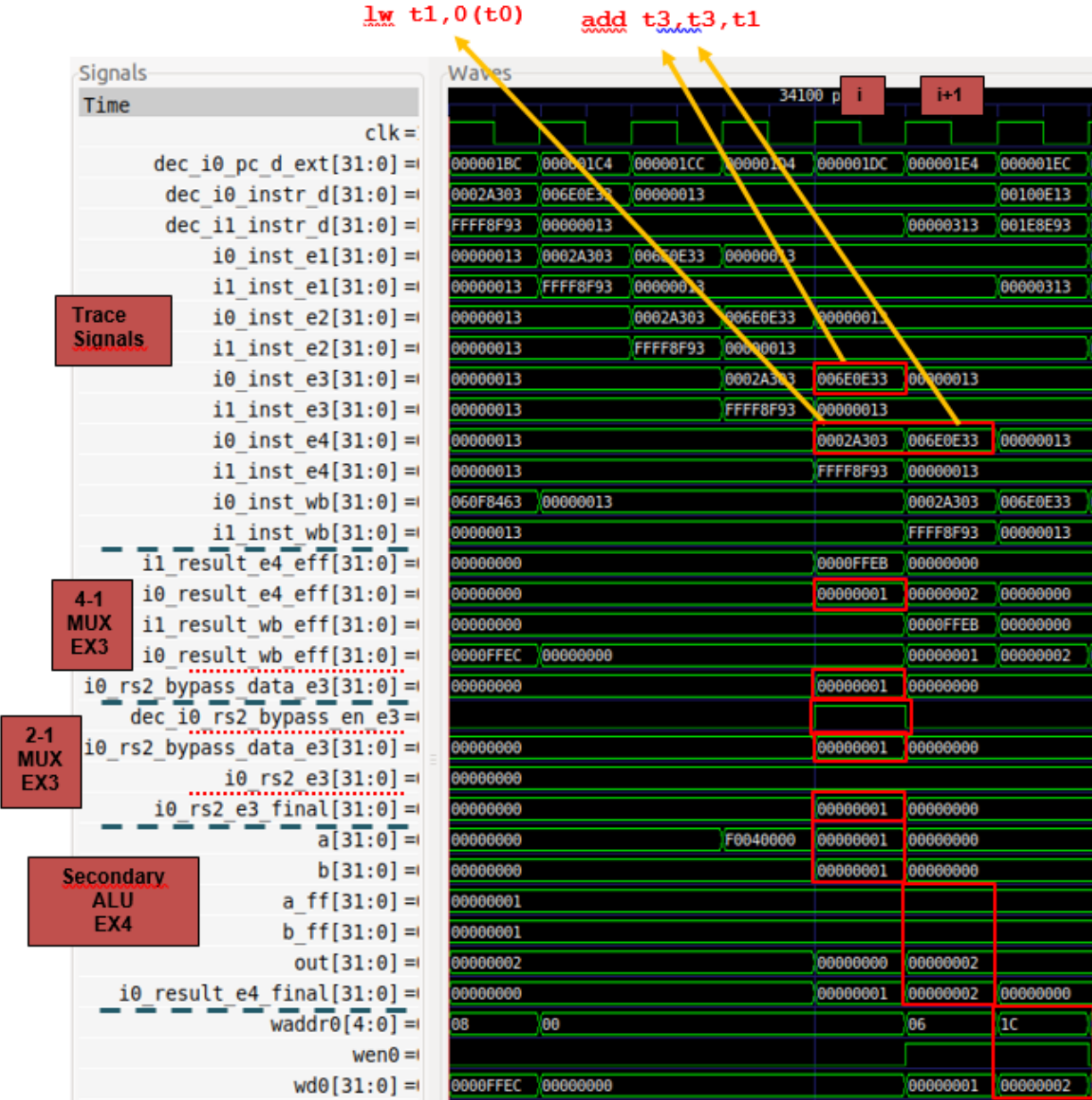
```
OUT:
```

```
.end
```

RVfpga實驗15：透過在提交階段轉送解除資料冒險－管線



RVfpga實驗15：透過在提交階段轉送解除資料冒險－模擬



RVfpga實驗15：透過在提交階段轉送解除資料冒險－模擬

- 追蹤訊號

- 週期*i*：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`）。
- 週期*i+1*：add指令處於通路0的提交階段（`i0_inst_e4 = 0x006E0E33`）。

- 4選1多路開關

- 週期*i*：選擇lw指令的結果（在提交階段）：
`i0_rs2_bypass_data_e3 = i0_result_e4_eff = 0x00000001`

- 2選1多路開關

- 週期*i*：由於lw和add之間的相關性，選擇旁路值：
`i0_rs2_e3_final = i0_rs2_bypass_data_e3 = 0x00000001`

- 提交階段ALU

- 週期*i+1*：使用正確的值重新計算add運算：
`out = a_ff + b_ff = 0x00000001 + 0x00000001 = 0x00000002`

- 3選1多路開關

- 週期*i+1*：選擇輔助ALU的輸出（`exu_i0_result_e4`）。（當不存在相關性時，選擇*i0_result_e4*。）

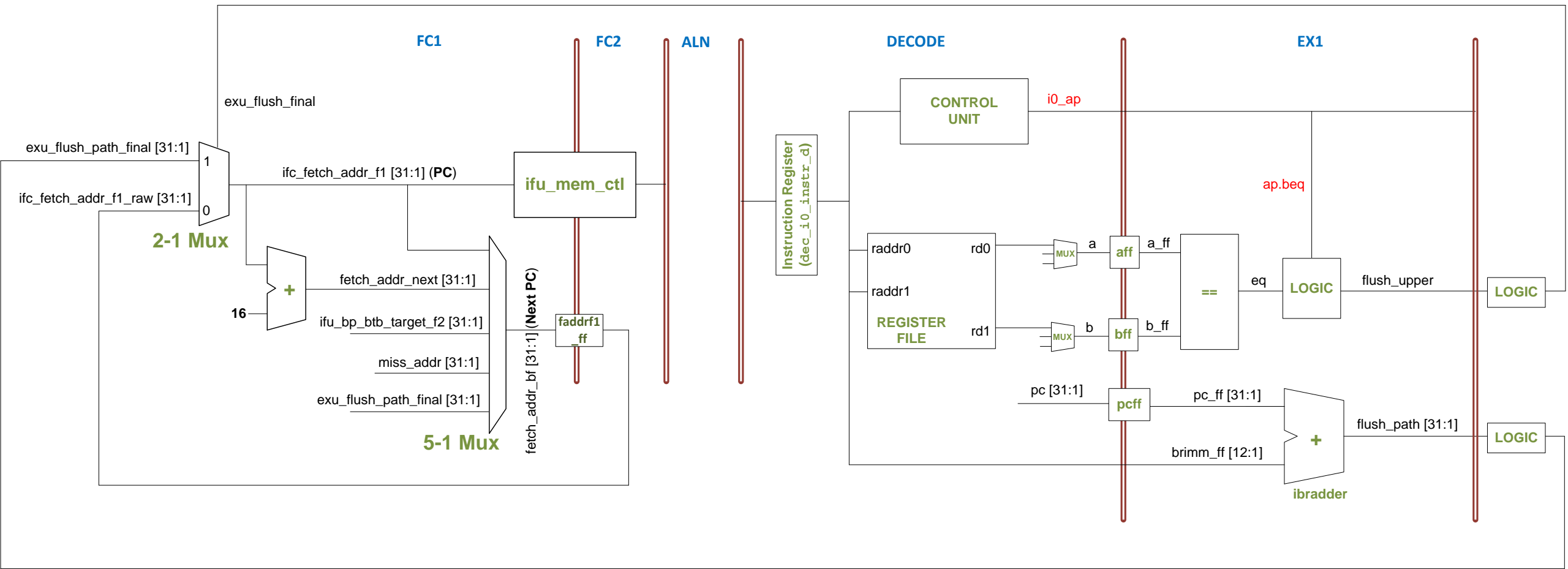
實驗16： 控制冒險和 分支指令



RVfpga實驗16：簡介

- 實驗16將重點關注分支和跳轉指令引起的**控制冒險**。
- 這些指令必須計算下一條指令的位址（對於非跳轉/分支指令，下一條指令的位址是 $PC + 4$ ）。
- 控制冒險可能：
 - 暫停管線，直到計算出下一條指令的位址，或
 - 預測是否會發生分支，並從預測的路徑中擷取。分支確定後，如果與預測相反，處理器可以清除擷取的指令，如果與預測相同，則繼續執行擷取的指令。
- SweRV EH1有兩個可能的分支預測器（Branch Predictor，BP）
 - 簡單分支預測器：總是預測為不發生分支。效能較差，但沒有硬體成本。
 - Gshare分支預測器：效能較高，需要花費額外的硬體成本。
- 本實驗將使用naïve和Gshare BP分析beq指令的執行。

RVfpga實驗16：beq指令執行和PC計算

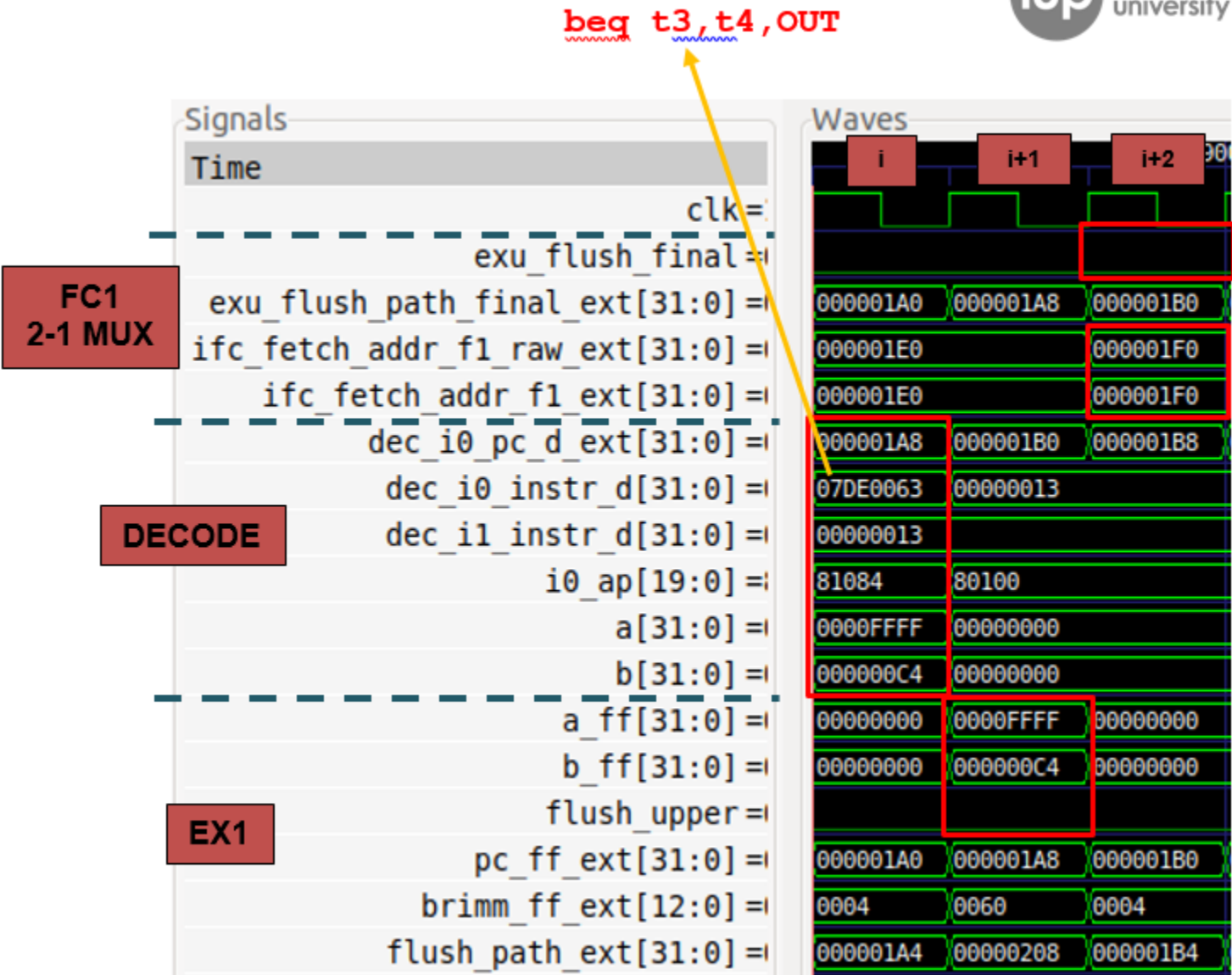


RVfpga實驗16：beq指令執行和PC計算－範例

```
Test_Assembly:
li t2, 0x008                # Disable Branch Predictor
csrrs t1, 0x7F9, t2
li t3, 0xFFFF
li t4, 0x1
li t5, 0x0
li t6, 0x0
LOOP:
    add t5, t5, 1
    INSERT_NOPS_7
    beq t3, t4, OUT
    INSERT_NOPS_7
    add t4, t4, 1
    INSERT_NOPS_7
    beq t3, t3, LOOP
    INSERT_NOPS_7
OUT:
INSERT_NOPS_8
.end
```



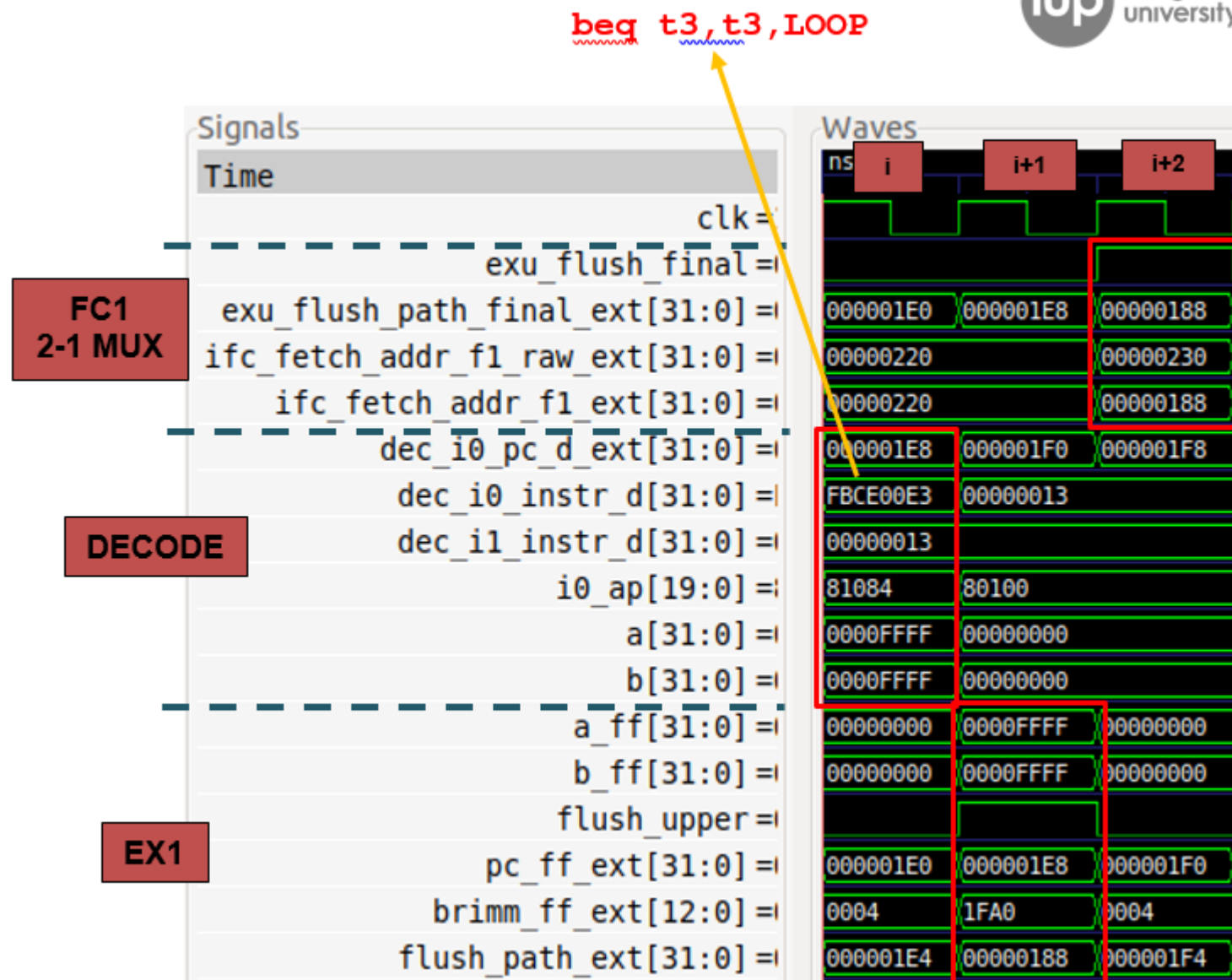
RVfpga實驗16：第一條beq指令的執行－模擬



RVfpga實驗16：第一條beq指令的執行 - 分析

- **週期*i* - beq指令的解碼階段**：在通路0中對第一條beq指令（0x07DE0063）進行解碼。產生控制訊號、讀取暫存器檔案並將分支指令傳送至I0管道。訊號a和b（本例中分別為0xFFFF和0xC4）中包含下一階段所使用的比較器的輸入。
- **週期*i+1* - beq指令的EX1階段**：執行beq指令。比較訊號a_ff和b_ff。兩個數（0xFFFF和0xC4）不同，因此不發生分支。在本例中，Gshare預測器被停用，因此所有分支的預測結果均為不發生（i0_ap.predict_nt = 1）。因此，分支預測結果正確，管線不排清（flush_upper = 0）。
- **週期*i+2* - FC1階段**：假設已預測分支並確定不發生分支，則正常按順序擷取。應注意exu_flush_final = 0，ifc_fetch_addr_f1_ext[31:0] = ifc_fetch_addr_f1_raw_ext[31:0] = 0x000001F0。該位址指向下一個連續的128位元指令束。

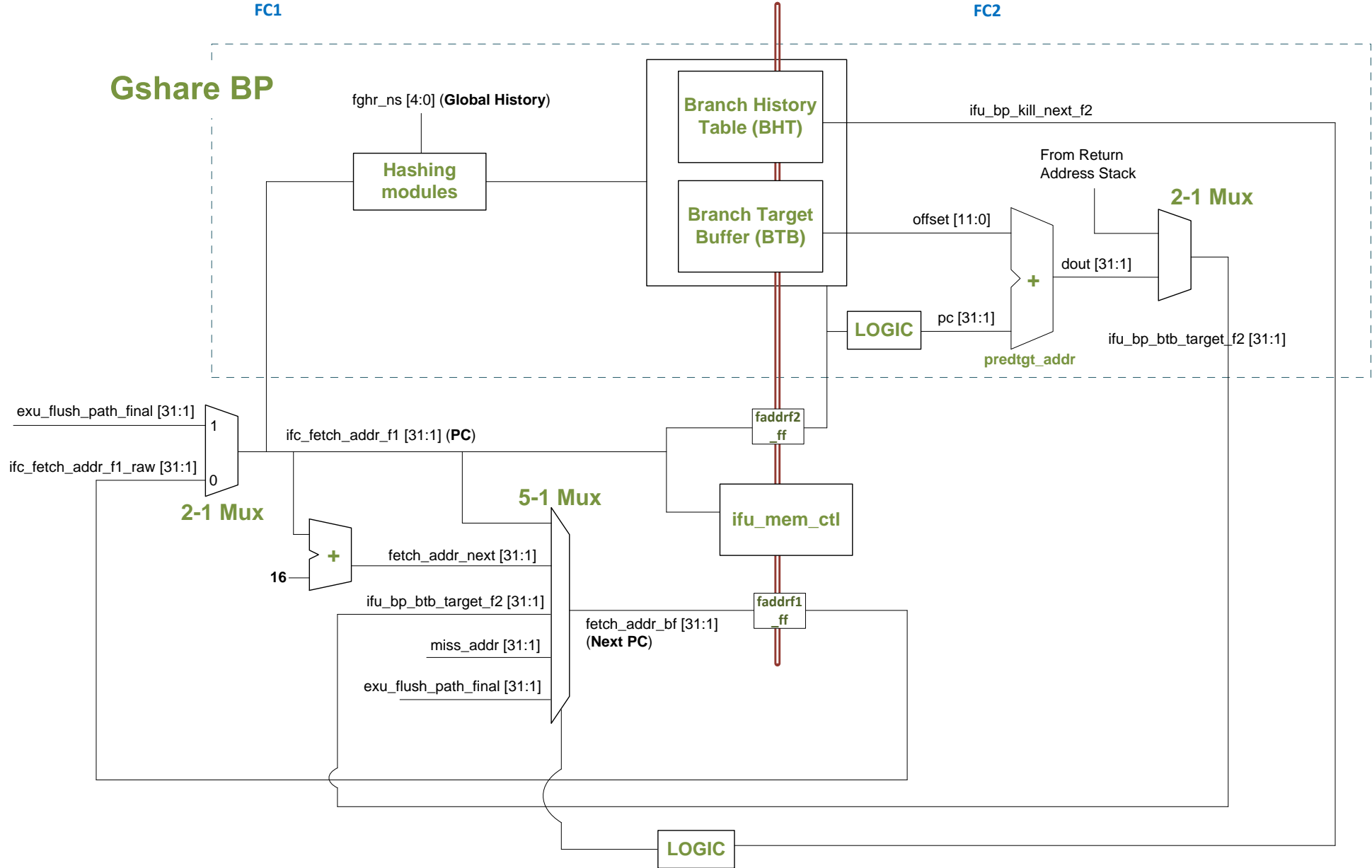
RVfpga實驗16：第二條beq指令的執行－模擬



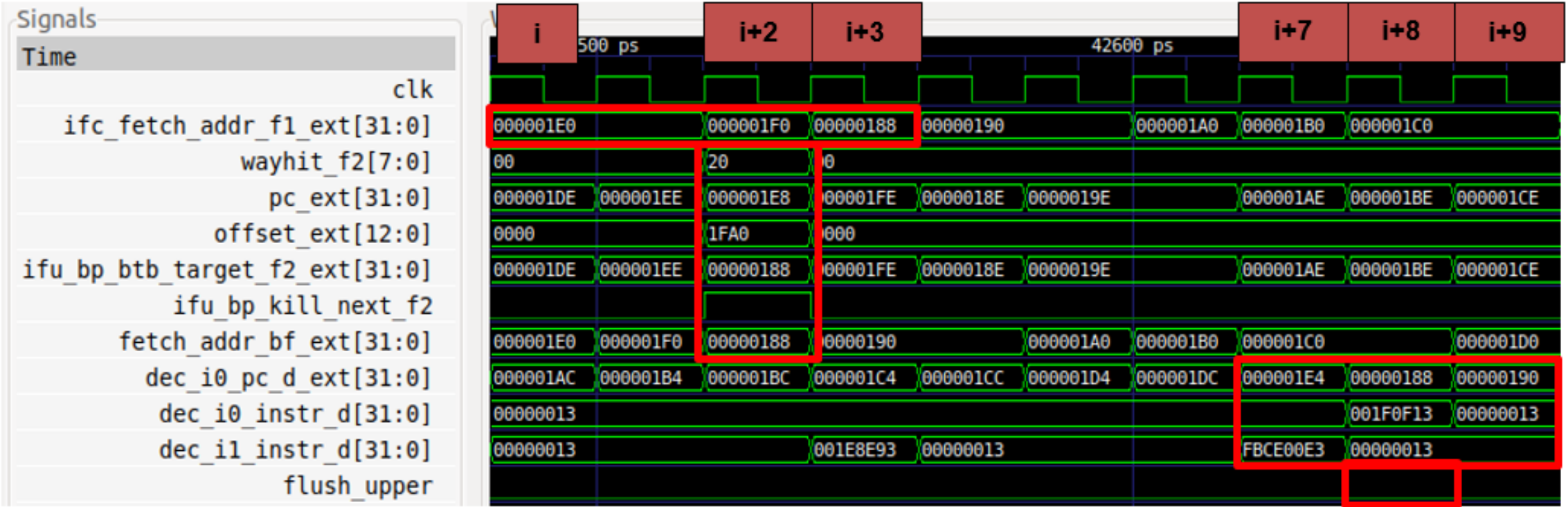
RVfpga實驗16：第二條beq指令的執行 - 分析

- **週期*i* - beq指令的解碼階段**：在通路0中對第二條beq指令（0xFBCE00E3）進行解碼。產生管線控制訊號、讀取暫存器檔案並將分支指令傳送至IO管道。訊號a和b（本例中均為0xFFFF）中包含下一階段所使用的比較器的輸入。
- **週期*i+1* - beq指令的EX1階段**：執行beq指令。比較訊號a_ff和b_ff。兩個數相等，因此發生分支。但是，簡單BP將所有分支預測為不發生（i0_ap.predict_nt = 1）。因此，分支預測結果錯誤，必須清除擷取的指令（flush_upper = 1）。
- **週期*i+2* - FC1階段**：指令必須在分支目標位址處繼續執行。exu_flush_final = 1 and ifc_fetch_addr_f1_ext = exu_flush_path_final_ext = 0x00000188。該位址對應於分支目標位址，即迴圈的第一條指令的位址。

RVfpga實驗16：Gshare分支預測器



RVfpga實驗16：第二條beq指令的Gshare分支預測器



RVfpga實驗16：第二條beq指令的Gshare分支預測器

- **週期*i***：包含第二條分支的指令末的位址將提供給指令快取：`ifc_fetch_addr_f1_ext = 0x000001E0`。系統使用該位址讀取分支目標緩衝區（BTB）。
- **週期*i+2***：在BTB中發生了命中：`wayhit_f2 = 0x20`。將分支位址（`pc_ext = 0x000001E8`）與BTB提供的偏移量（`offset_ext = 0x1FA0`，該值為負值）相加，即可得到預測目標位址（`ifu_bp_btb_target_f2_ext = 0x00000188`）。如果BHT預測將發生分支（`ifu_bp_kill_next_f2 = 1`），則預測目標位址將用作下次擷取PC（`fetch_addr_bf_ext = 0x00000188`）。
- **週期*i+3***：擷取位址為上一週期中計算的分支預測目標位址：`ifc_fetch_addr_f1_ext = 0x00000188`。
- **週期*i+7***：在通路1中對分支進行解碼（`dec_i1_instr_d = 0xFBCE00E3`）。
- **週期*i+8***：執行分支。預測正確，因此無需觸發排清操作（`flush_upper = 0`）。
- **週期*i+9***：如果預測正確，將透過分支目標位址正常繼續執行分支。

實驗17： 超標量執行



RVfpga實驗17：簡介

- Western Digital的SweRV EH1處理器是32位元核心，採用9級管線和雙路超標量設計。
- 超標量處理器包含資料路徑硬體的多個副本，可同時執行多條指令。
- 執行單條指令的延遲與標量處理器相同，但處理器可以在每個週期執行和提交更多的指令，進而提高吞吐量。

RVfpga實驗17：簡介

- SweRV EH1是一個**雙路超標量**處理器，
 - 每個週期最多可以擷取、執行和提交兩條指令。
 - 多連接埠暫存器檔案最多讀取四個來源運算元，並在每個週期中寫回兩個值（加上一個來自非阻塞載入的值，如實驗15的分析所示）。
 - 每個通路包含**獨立管道**：兩條整數管道、一條乘法管道、一條載入-儲存管道和一個非管線除法器。
- 理想情況下，雙路超標量處理器的吞吐量（**IPC**）應為單指令處理器的兩倍。遺憾的是，在實際的程式中，從單路處理器升級為雙路處理器時，效能會提高到**1.3至1.5倍**；但新增第二條通路需要使用更多的硬體。
- 在本實驗中，我們將分析兩個簡單的程式，比較使用**SweRV EH1**單指令和雙指令組態時的行為。

RVfpga實驗17：四條獨立的A-L指令 – 範例 – 單指令

```
.globl Test_Assembly
```

```
.text
```

```
Test_Assembly:
```

```
li t2, 0x400          # Disable Dual-Issue Execution  
csrrs t1, 0x7F9, t2
```

```
li t0, 0x0
```

```
li t1, 0x1
```

```
li t2, 0x1
```

```
li t3, 0x3
```

```
li t4, 0x4
```

```
li t5, 0x5
```

```
li t6, 0x6
```

```
lui t2, 0xF4
```

```
add t2, t2, 0x240
```

```
REPEAT:
```

```
add t0, t0, 1
```

```
INSERT_NOPS_10
```

```
INSERT_NOPS_4
```

```
add t3, t3, t1
```

```
sub t4, t4, t1
```

```
or t5, t5, t1
```

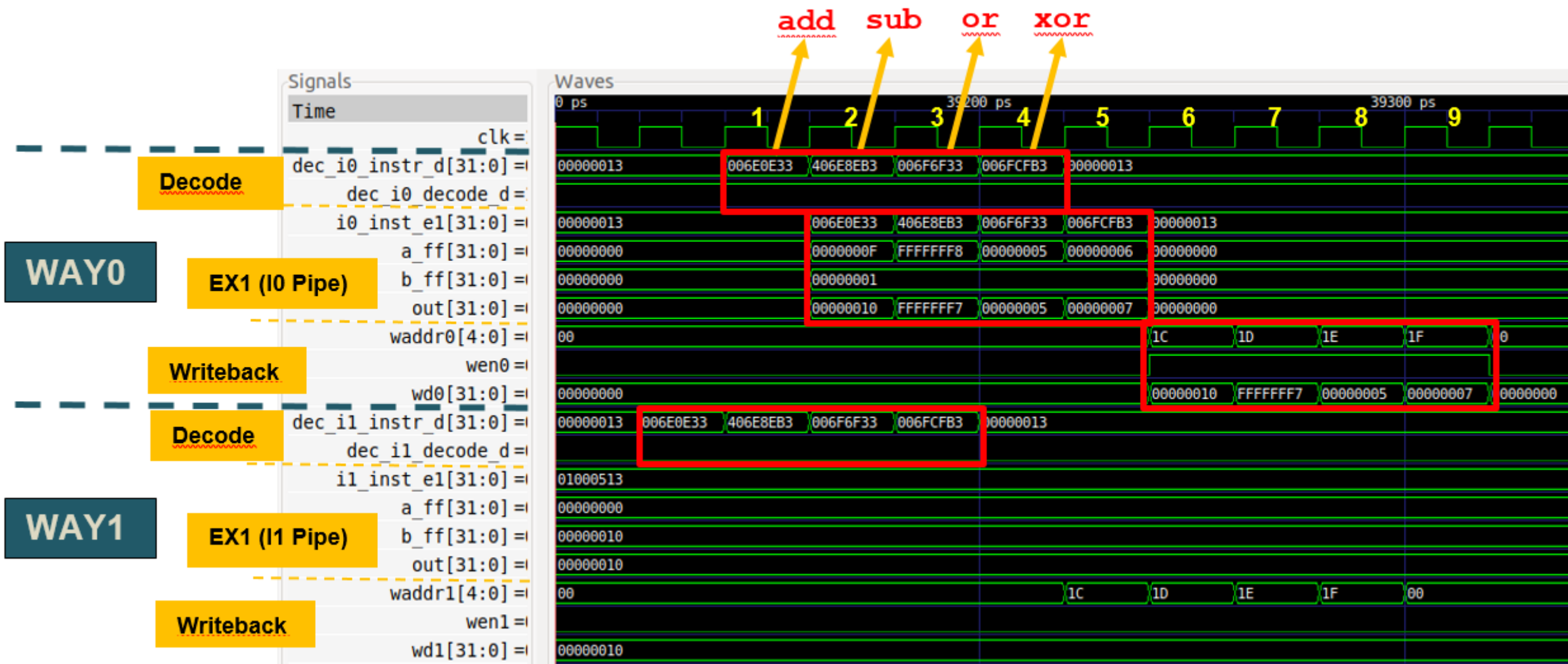
```
xor t6, t6, t1
```

```
INSERT_NOPS_10
```

```
INSERT_NOPS_3
```

```
bne t0, t2, REPEAT # Repeat the loop
```

RVfpga實驗17：四條獨立的A-L指令 – 模擬 – 單指令



RVfpga實驗17：四條獨立的A-L指令 – 模擬 – 單指令

- 解碼時，兩條通路都會接收指令，但僅通路0會將指令傳送至執行階段，因為通路1的傳送功能已停用。
 - 通路0：
 - 在我們的範例中，訊號dec_i0_decode_d始終為1；具體地說，對於我們所分析的四條AL指令，其值均為1。
 - 解碼階段的指令會傳播到I0管道（i0_inst_e1[31:0]）。
 - 通路1：
 - 在我們的範例中，訊號dec_i1_decode_d始終為0；具體地說，對於我們所分析的四條AL指令，其值均為0。
 - 解碼階段的指令不會傳播（i1_inst_e1[31:0]）到執行階段。
- 因此，系統僅使用來自I0管道的ALU（參見兩條通路中的訊號aff、bff和out），並且僅使用暫存器檔案的寫入連接埠0（參見兩條通路中的訊號waddr、wen和wd）。

RVfpga實驗17：四條獨立的A-L指令 – 圖 – 單指令

	Cyc 1	Cyc 2	Cyc 3	Cyc 4	Cyc 5	Cyc 6	Cyc 7	Cyc 8	Cyc 9
Decode	add	sub	or	xor					
EX1		add	sub	or	xor				
EX2			add	sub	or	xor			
EX3				add	sub	or	xor		
Commit					add	sub	or	xor	
Writeback						add	sub	or	xor



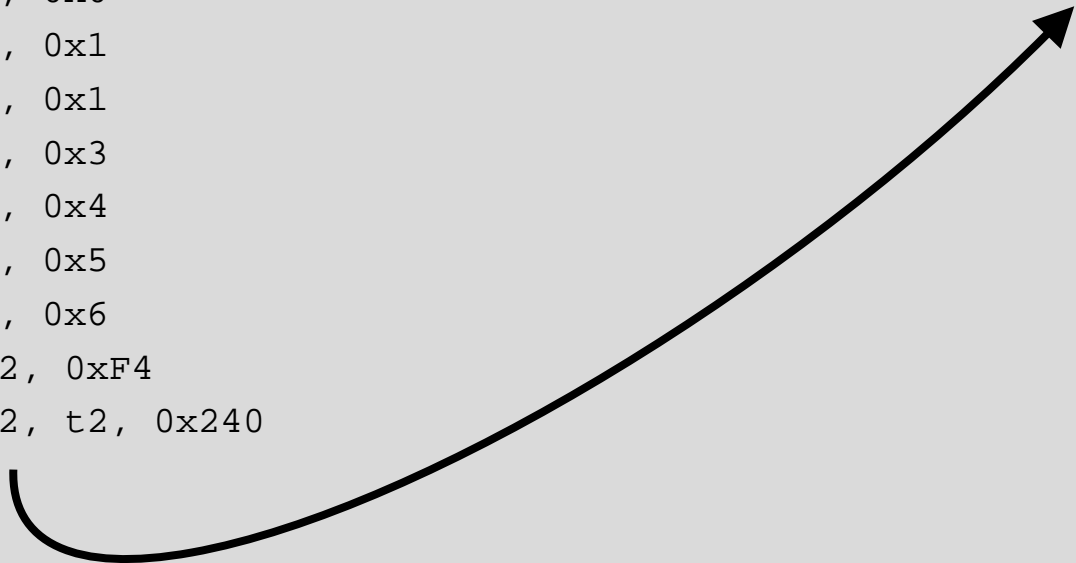
RVfpga實驗17：四條獨立的A-L指令 – 範例 – 雙指令

```
.globl Test_Assembly

.text
Test_Assembly:

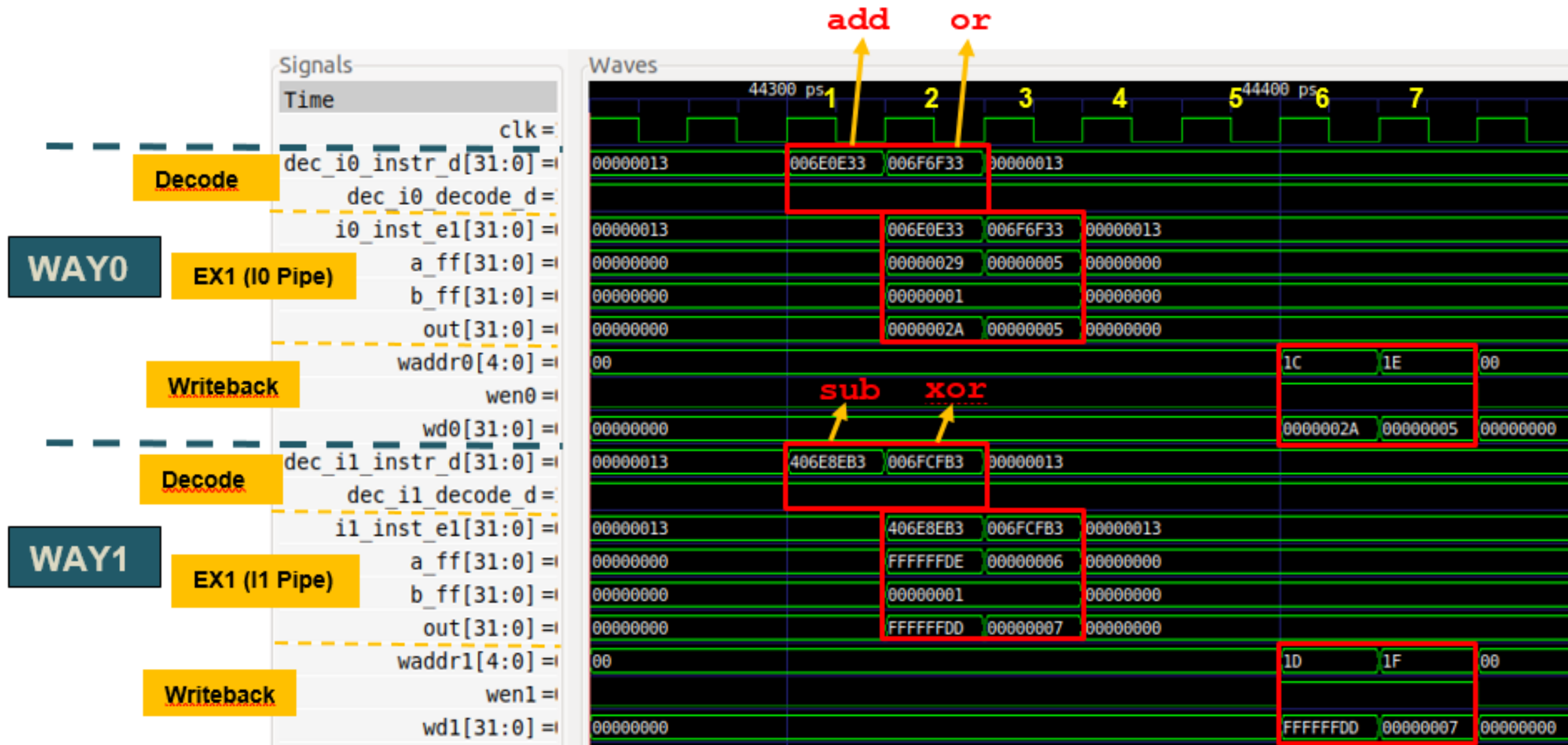
# li t2, 0x400          # Disable Dual-Issue Execution
# csrrs t1, 0x7F9, t2

li t0, 0x0
li t1, 0x1
li t2, 0x1
li t3, 0x3
li t4, 0x4
li t5, 0x5
li t6, 0x6
lui t2, 0xF4
add t2, t2, 0x240
```



```
REPEAT:
    add t0, t0, 1
    INSERT_NOPS_10
    INSERT_NOPS_4
    add t3, t3, t1
    sub t4, t4, t1
    or  t5, t5, t1
    xor t6, t6, t1
    INSERT_NOPS_10
    INSERT_NOPS_3
    bne t0, t2, REPEAT # Repeat the loop
```

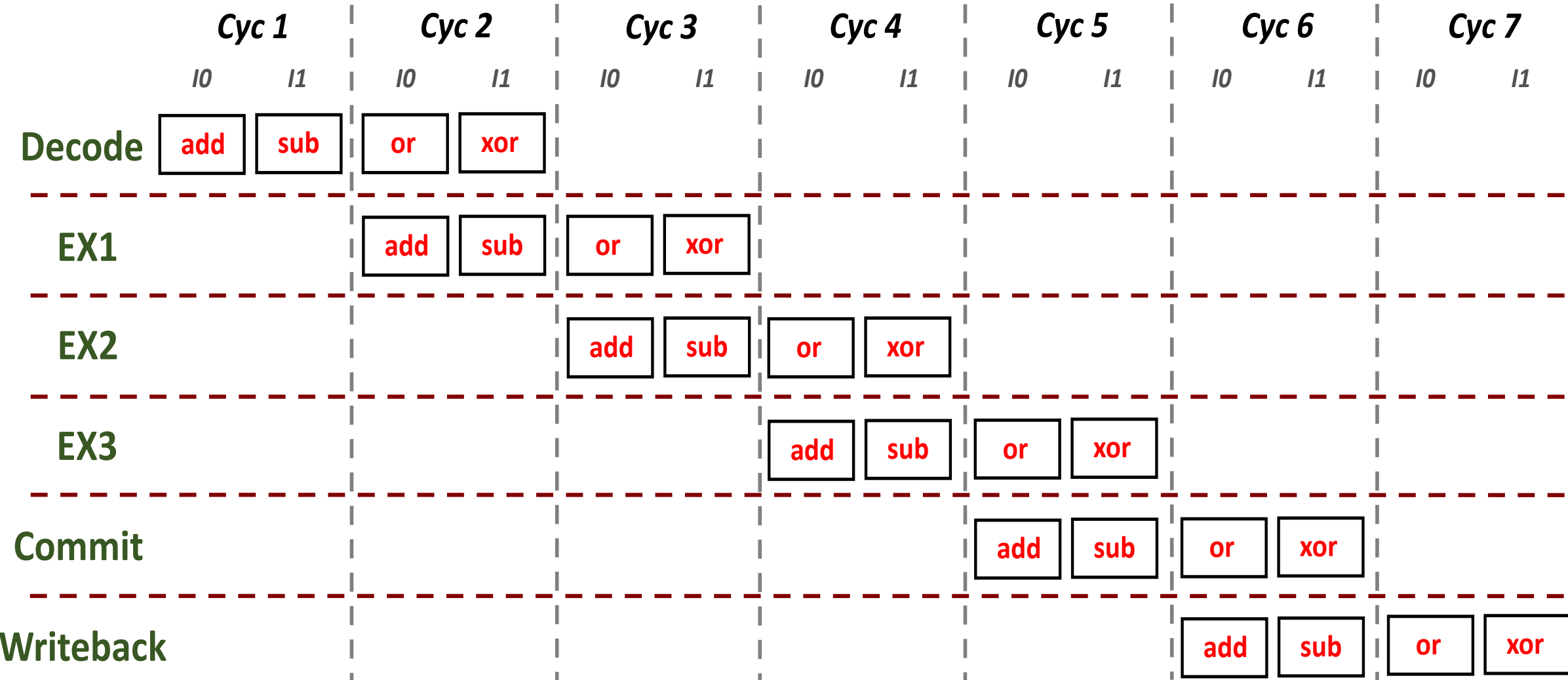

RVfpga實驗17：四條獨立的A-L指令 – 模擬 – 雙指令



RVfpga實驗17：四條獨立的A-L指令 – 分析 – 雙指令

- 在每個週期中，對兩條指令進行解碼（每個通路一條），並將兩條指令傳送到執行階段（一條通過I0管道，另一條通過I1管道）。
 - 通路0：
 - 對於範例中的四條A-L指令中的兩條，訊號dec_i0_decode_d始終為1（另外兩條在通路1中進行解碼）。
 - 解碼階段的指令會傳播到I0管道（i0_inst_e1[31:0]）。
 - 通路1：
 - 對於範例中的四條A-L指令中的兩條，訊號dec_i1_decode_d始終為1（另外兩條在通路0中進行解碼）。
 - 解碼階段的指令會傳播到I1管道（i1_inst_e1[31:0]）。
- 因此，系統會使用兩條管道（I0和I1）中的ALU（參見兩條通路中的訊號aff、bff和out），並且會使用兩個暫存器檔案寫入連接埠（參見兩條通路中的訊號waddr、wen和wd）。

RVfpga實驗17：四條獨立的A-L指令 – 圖 – 雙指令



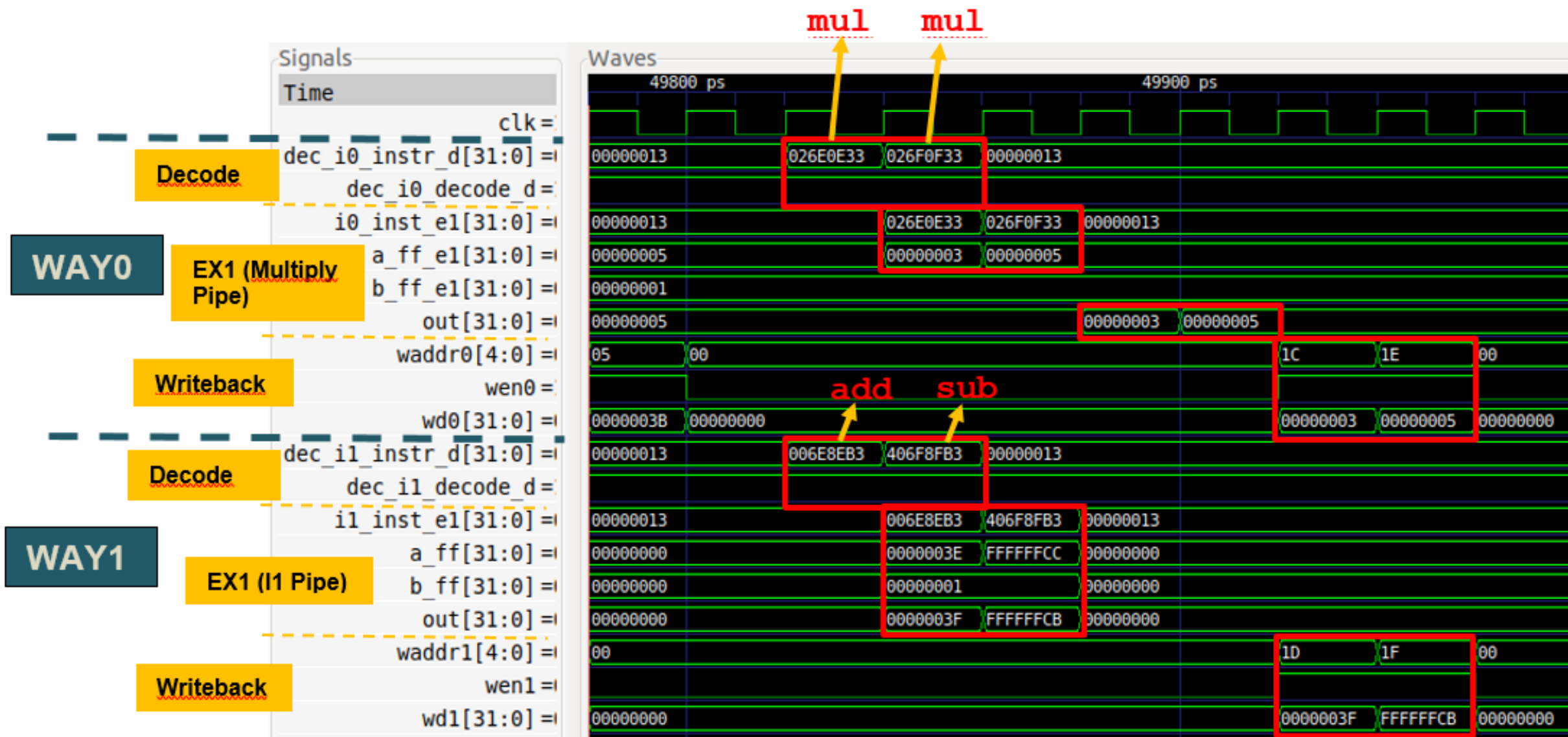
RVfpga實驗17：互相交織的兩條mul指令和兩條A-L指令 – 範例 – 雙指令

```
.globl Test_Assembly
.text
Test_Assembly:
# li t2, 0x400      # Disable Dual-Issue Execution
# csrrs t1, 0x7F9, t2

li t3, 0x3
li t4, 0x4
li t5, 0x5
li t6, 0x6
li t0, 0x0
lui t1, 0xF4
add t1, t1, 0x240
```

```
REPEAT:
    add t0, t0, 1
    INSERT_NOPS_10
    INSERT_NOPS_4
    mul t3, t3, t1
    add t4, t4, t1
    mul t5, t5, t1
    sub t6, t6, t1
    INSERT_NOPS_10
    INSERT_NOPS_3
    bne t0, t1, REPEAT # Repeat the loop
.end
```

RVfpga實驗17：互相交織的兩條mul指令和兩條A-L指令 – 模擬 – 雙指令



RVfpga實驗17：互相交織的兩條mul指令和兩條A-L指令 – 分析 – 雙指令

- 解碼時，兩條通路都會接收指令並將指令傳送至執行階段。
 - 通路0：
 - 對於範例中所分析的四條指令中的兩條，訊號dec_i0_decode_d始終為1（另外兩條在通路1中進行解碼）。
 - 解碼階段的指令會傳送到乘法管道（i0_inst_e1[31:0]）。
 - 通路1：
 - 對於範例中所分析的四條指令中的兩條，訊號dec_i1_decode_d始終為1（另外兩條在通路1中進行解碼）。
 - 解碼階段的指令（dec_i1_instr_d[31:0]）會傳播到I1管道（i1_inst_e1[31:0]）
- 因此，系統會使用I1管道和乘法器中的ALU（參見訊號a_ff_e1、b_ff_e1和out，以及訊號a_ff、b_ff和out），並且會使用兩個暫存器檔案寫入連接埠（參見兩條通路中的訊號waddr、wen和wd）。

實驗18： 新增新功能： 指令和計數器



RVfpga實驗18：簡介

- 本實驗將應用在先前實驗中獲得的知識來修改SweRV EH1處理器，向其新增以下新功能：
 - **新增A-L指令**：透過RISC-V架構中提供的全新位元操作延伸功能來新增算術邏輯指令。
 - **新增浮點指令**：新增三條浮點指令：加、乘、除。然後使用這些指令進行二分法計算。
 - **新增硬體計數器**：新增一個新的硬體計數器，用於計算執行的I型指令數量。
- 在一些練習中，我們將指導您完成修改核心的過程，在其他練習中，您必須自行確定所需的操作。您可以在實驗文件中找到所有詳細資訊。

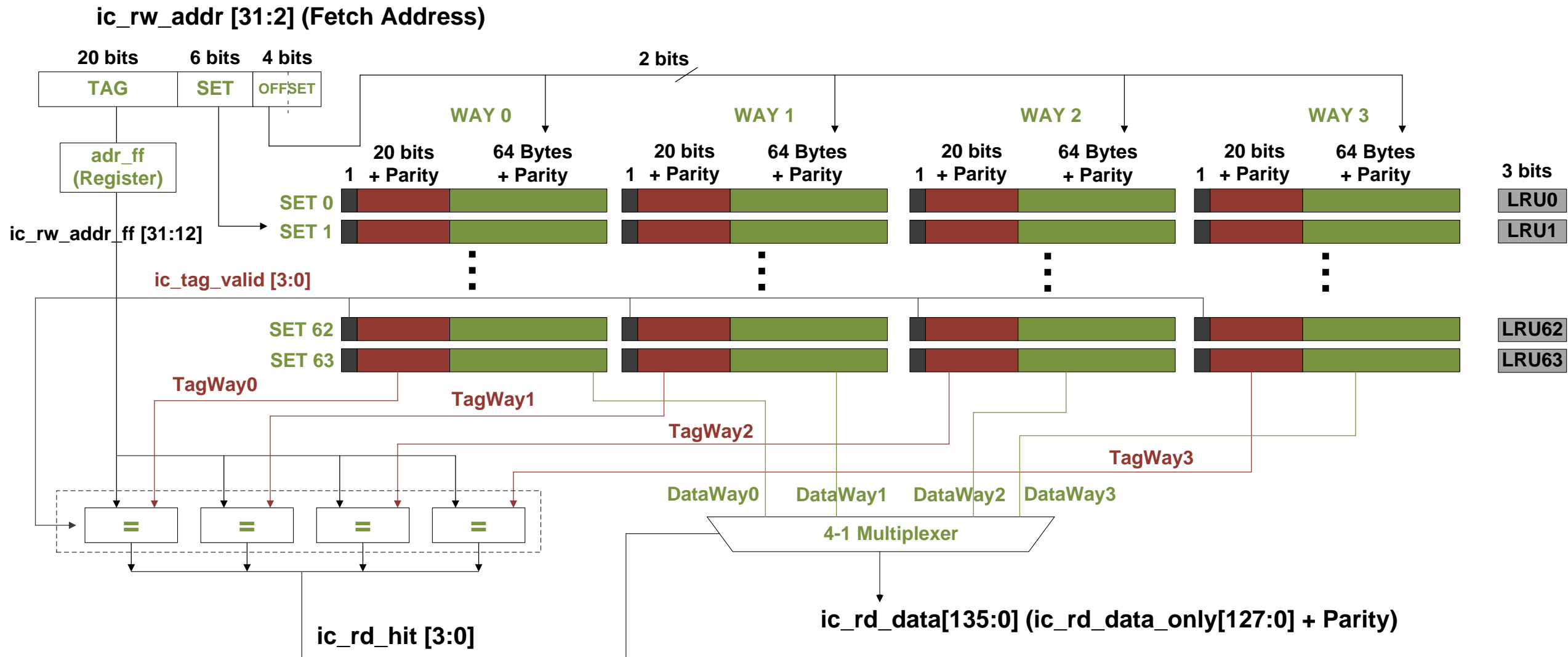
實驗19： 指令快取



RVfpga實驗19：簡介

- 本實驗將介紹並探究RVfpga系統的記憶體系統。Rvfpga的記憶體系統具有以下元素：
 - 外部DDR主記憶體
 - 指令快取（I\$）
 - 兩個暫存記憶體（也稱為緊密耦合記憶體），一個用於儲存資料（DCCM），另一個用於儲存指令（ICCM）。預設系統中會停用ICCM。
- 本實驗主要分析指令快取（I\$）的操作。本實驗將介紹I\$的配置方式，研究快取失效和命中的處理方式，並分析I\$替換策略。

RVfpga實驗19：I\$組態和操作



RVfpga實驗19：I\$失效和命中管理 - 範例

Test_Assembly:

```
INSERT_NOPS_3  
INSERT_NOPS_8  
INSERT_NOPS_8  
li t6, 0x10000
```

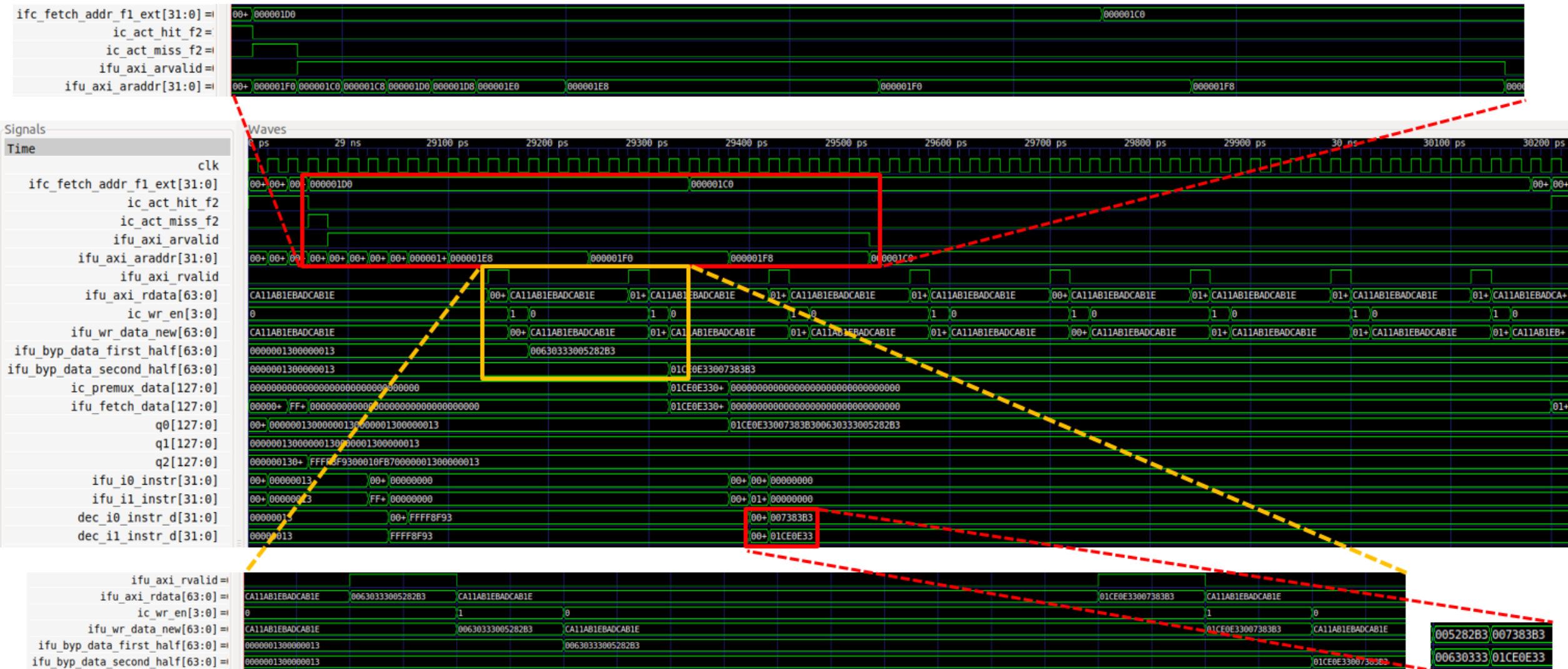
REPEAT:

```
add t6, t6, -1
```

```
add t0, t0, t0  
add t1, t1, t1  
add t2, t2, t2  
add t3, t3, t3  
add t4, t4, t4  
add t5, t5, t5  
add t6, t6, t6  
add a7, a7, a7  
add t0, t0, t0  
add t2, t2, t2  
add t1, t1, t1  
add t3, t3, t3  
add t4, t4, t4  
add t6, t6, t6  
add t5, t5, t5  
add a7, a7, a7
```

```
INSERT_NOPS_8  
INSERT_NOPS_8  
INSERT_NOPS_8  
INSERT_NOPS_8  
INSERT_NOPS_8  
INSERT_NOPS_8  
bne t6, zero, REPEAT  
  
ret
```

RVfpga實驗19：I\$失效管理 - 模擬



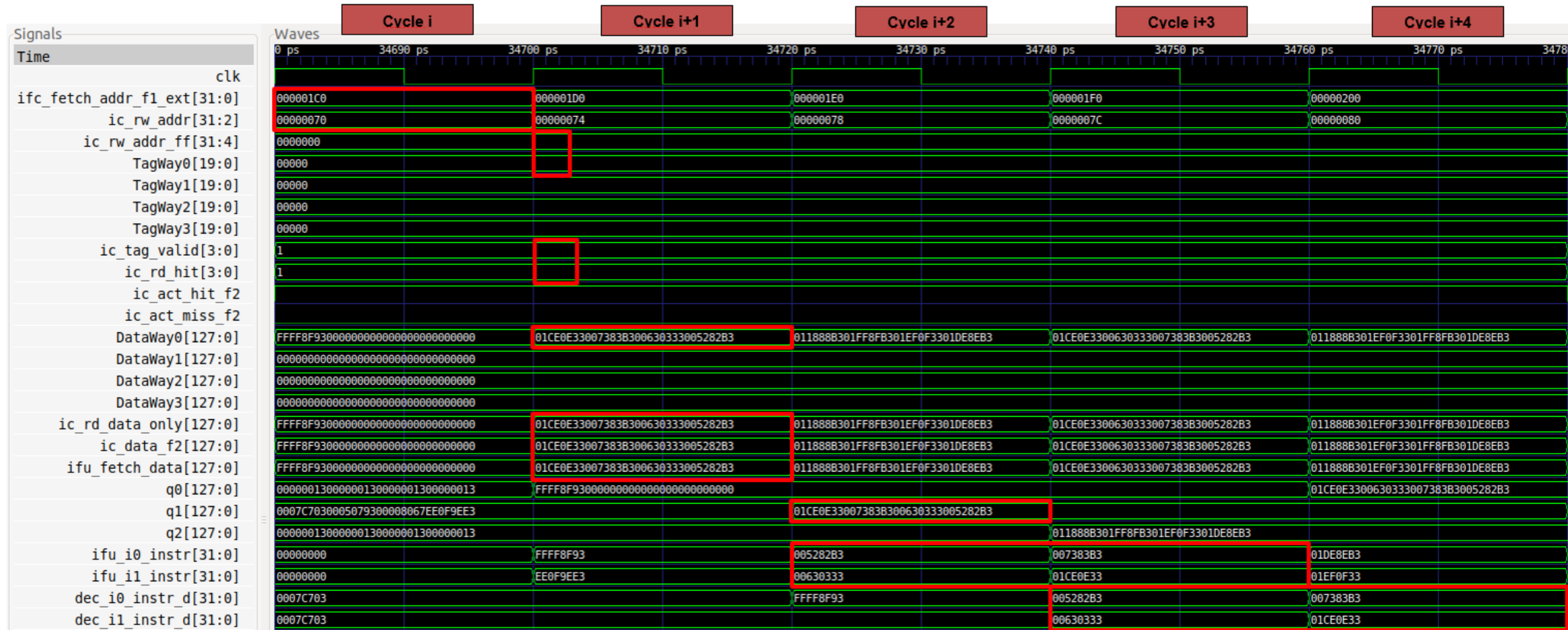
RVfpga實驗19：I\$失效管理 - 分析

- 模擬展示了初次執行16條add指令時的擷取操作。如果這些指令不在I\$中，則I\$中將觸發失效，必須將指令從DDR外部記憶體複製到I\$中。
 - 每隔約29 ns便會出現一次I\$失效（ic_act_miss_f2 = 1），該事件將觸發透過AXI匯流排觸發的區塊要求（ifu_axi_arvalid = 1）。
 - 系統將透過AXI匯流排依次要求構成目標區塊的八個64位元區塊。
 - 訊號ifu_axi_arvalid連續27個週期保持高電平。該訊號表示通道正在傳送有效的讀取位址和控制資訊。
 - 在ifu_axi_arvalid = 1的這27個週期內，訊號ifu_axi_araddr將透過AXI匯流排依次提供8個64位元區塊的初始位址，這8個位址必須從DDR記憶體讀取。

RVfpga實驗19：I\$失效管理 - 分析

- 中間的圖則展示了八個64位元資料區塊透過訊號ifu_axi_rdata中的AXI匯流排依次到達處理器。
 - 訊號ifu_axi_rvalid用於指示通道正在傳送所需的讀取資料，該訊號每經7個週期便會保持一個週期的高電平。
 - 八個64位元區塊（每個區塊包含兩條指令）均由訊號ifu_axi_rdata提供。
- 下面的兩張圖顯示，八個64位元區塊在到達快取控制器後均會立即寫入I\$。
- 最終，可以看到上述四條指令從I\$控制器旁路到管線，以便在I\$失效後盡快重新啟動執行。幾個週期後，四條指令將到達解碼階段。

RVfpga實驗19：I\$命中管理 - 模擬



RVfpga實驗19：I\$命中管理 - 分析

- 在之前的模擬中，可以看到I\$中出現命中。
 - 週期i**：第一條add指令（add t0,t0,t0）的位址由訊號ifc_fetch_addr_f1_ext提供。該訊號將傳遞到I\$，但不需要去掉兩個最低有效位元，因為指令以4個位元組（32位元）邊界對齊。因此，ic_rw_addr = 0x0000070。標籤陣列和資料陣列使用擷取位址的子集。
 - 週期i+1**：四個標籤（每個快取通路一個）位於訊號TagWay0-TagWay3中。這些標籤將與擷取位址的TAG欄位進行比較。在本例中，所有標籤均與TAG欄位相同，但只有一條通路（通路0）有效（ic_tag_valid = 0001），因此將在通路0中發出命中訊息：ic_rd_hit = 0001。四個128位元指令束同樣位於訊號DataWay0-DataWay3中：ic_rd_data_only = 0x01ce0e33007383b300630333005282b3
 - 週期i+2**：在對齊階段，從緩衝區q1中擷取第一條和第二條add指令：ifu_i0_instr = 0x005282b3且ifu_i1_instr = 0x00630333
 - 週期i+3**：在對齊階段擷取第三條和第四條add指令，同時對第一條和第二條add指令進行解碼：ifu_i0_instr = 0x007383b3、ifu_i1_instr = 0x01ce0e33、dec_i0_instr_d = 0x005282b3且dec_i1_instr_d = 0x00630333
 - 週期i+4**：最後，對第三條和第四條add指令進行解碼：dec_i0_instr_d = 0x007383b3且dec_i1_instr_d = 0x01ce0e33

RVfpga實驗19：I\$替換策略

- 大多陣列相連快取採用最近最少使用（Least Recently Used，LRU）的替換策略。然而，追蹤最近最少使用的通路較為複雜，因此一般使用簡化的LRU策略（通常稱為偽LRU），該策略已足以滿足實際需求。
- SweRV EH1使用名為二進位樹偽LRU的簡化策略。
 - 要實現該策略，N路組相連快取中的每組需要有N-1個位元（稱為LRU狀態）。在SweRV EH1的I\$中則為每組3位元。

區塊替換

LRU狀態	替換的通路
x00	通路0
x10	通路1
0x1	通路2
1x1	通路3

LRU狀態更新

寫入的通路	下一LRU狀態
通路0	-11
通路1	-01
通路2	1-0
通路3	0-0

RVfpga實驗19：I\$替換策略 - 範例

- 下面的範例在一個無限迴圈中存取五個不同的I\$區塊，這五個區塊均對映到同一I\$組：SET = 8。
- 無限迴圈包含五條j（跳轉）指令，其中每對j指令由1023個nop分隔。j指令與nop共佔用4 KiB空間，等同於I\$中每條通路的大小。

```
Set8_Block1:    j Set8_Block2          # This j instruction is at address 0x00000200
                INSERT_NOPS_1023
Set8_Block2:    j Set8_Block3          # This j instruction is at address 0x00001200
                INSERT_NOPS_1023
Set8_Block3:    j Set8_Block4          # This j instruction is at address 0x00002200
                INSERT_NOPS_1023
Set8_Block4:    j Set8_Block5          # This j instruction is at address 0x00003200
                INSERT_NOPS_1023
Set8_Block5:    j Set8_Block1          # This j instruction is at address 0x00004200
```

SET 8 after execution of the first j instruction at 0x200

Valid	Tag	Data	
1	00000000000000000000	j Set8_Block2 nop ... nop	WAY 0
0			WAY 1
0			WAY 2
0			WAY 3

LRU STATE = 011

SET 8 after execution of the second j instruction at 0x1200

1	00000000000000000000	j Set8_Block2 nop ... nop	WAY 0
1	00000000000000000001	j Set8_Block3 nop ... nop	WAY 1
0			WAY 2
0			WAY 3

LRU STATE = 001

SET 8 after execution of the third j instruction at 0x2200

1	00000000000000000000	j Set8_Block2 nop ... nop	WAY 0
1	00000000000000000001	j Set8_Block3 nop ... nop	WAY 1
1	00000000000000000010	j Set8_Block4 nop ... nop	WAY 2
0			WAY 3

LRU STATE = 100

SET 8 after execution of the fourth j instruction at 0x3200

1	00000000000000000000	j Set8_Block2 nop ... nop	WAY 0
1	00000000000000000001	j Set8_Block3 nop ... nop	WAY 1
1	00000000000000000010	j Set8_Block4 nop ... nop	WAY 2
1	00000000000000000011	j Set8_Block5 nop ... nop	WAY 3

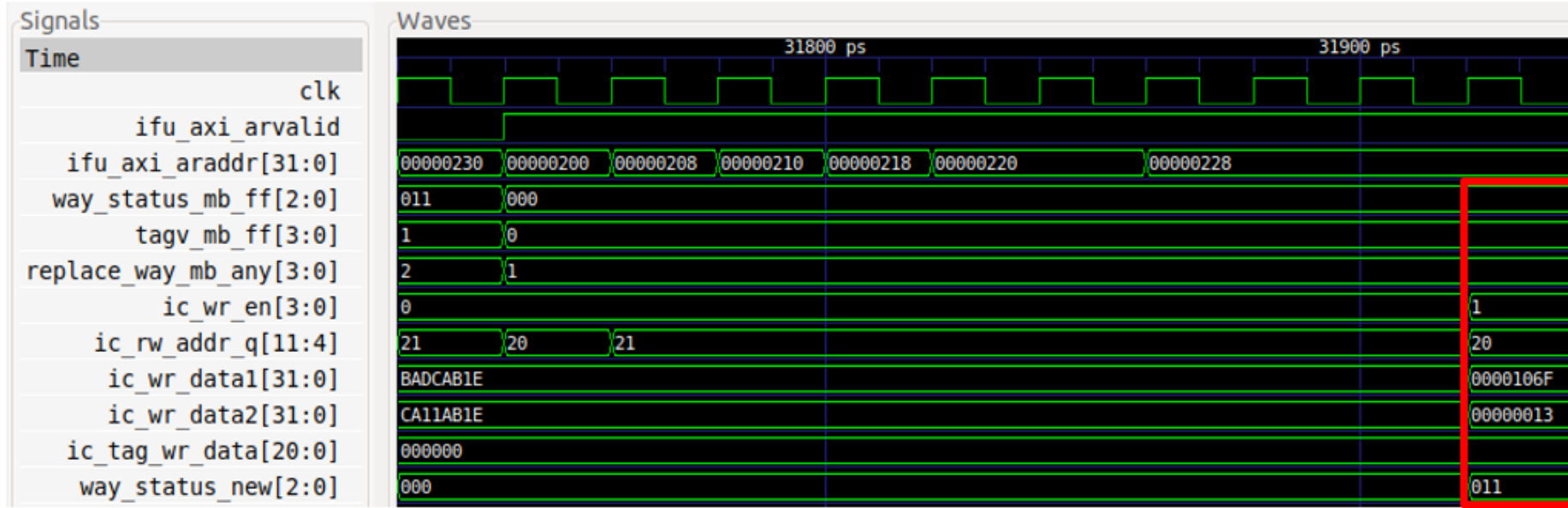
LRU STATE = 000

SET 8 after execution of the fifth j instruction at 0x4200

1	000000000000000000100	j Set8_Block1 nop ... nop	WAY 0
1	000000000000000000001	j Set8_Block3 nop ... nop	WAY 1
1	000000000000000000010	j Set8_Block4 nop ... nop	WAY 2
1	000000000000000000011	j Set8_Block5 nop ... nop	WAY 3

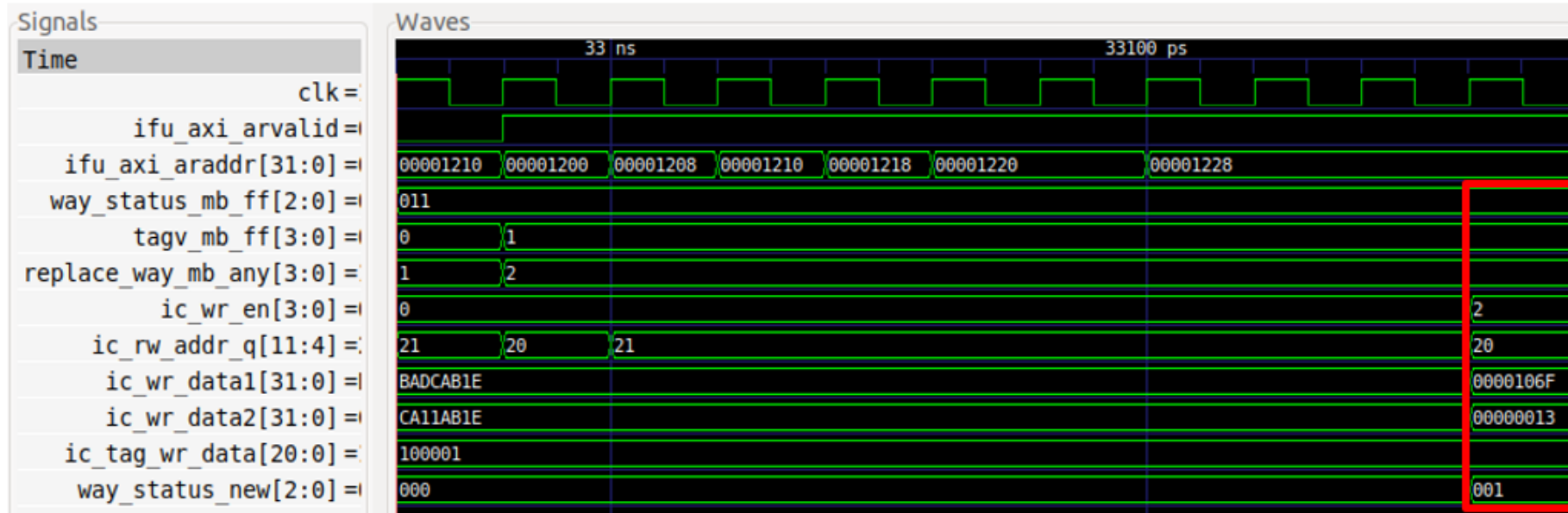
LRU STATE = 011

RVfpga實驗19：I\$替換策略 – 第1次跳轉



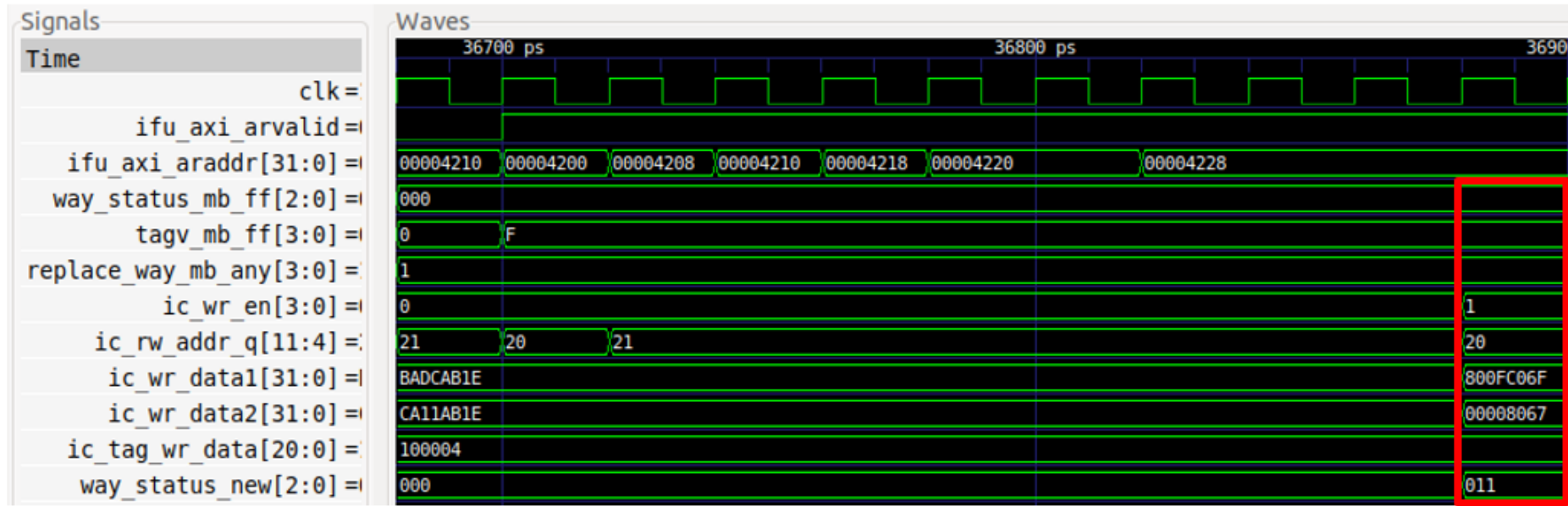
- 第一次跳轉的位址（`0x200`）對映到I\$的組8。該組的初始狀態為空，因此必須將新區塊寫入通路0。
`replace_way_mb_any = ic_wr_en = 0001`。組8的LRU狀態更新如下：`way_status_new = 011`。
- I\$區塊從DDR記憶體中讀取，並以64位元區塊的形式寫入I\$。上圖描繪了將新區塊的標籤和前兩條指令寫入組8的過程：
 - `ic_rw_addr_q[11:4] = 00100000`（組8）
 - `ic_tag_wr_data[19:0] = 0x0`
 - `ic_wr_data1[31:0] = 0x0000106F` (j Set8_Block2)
 - `ic_wr_data2[31:0] = 0x00000013` (nop)

RVfpga實驗19：I\$替換策略 – 第2次跳轉



- 第二次跳轉的位址（0x1200）同樣對映到I\$的組8。在該組中，僅通路0有效：tagv_mb_ff = 0001。因此，必須將新區塊寫入通路1：replace_way_mb_any = ic_wr_en = 0010。組8的LRU狀態更新如下：way_status_new = 001。
- I\$區塊從DDR記憶體中讀取，並以64位元區塊的形式寫入I\$。上圖描繪了將新區塊的標籤和前兩條指令寫入組8的過程：
 - ic_rw_addr_q[11:4] = 00100000（組8）
 - ic_tag_wr_data[19:0] = 0x1
 - ic_wr_data1[31:0] = 0x0000106F (j Set8_Block3)
 - ic_wr_data2[31:0] = 0x00000013 (nop)

RVfpga實驗19：I\$替換策略 – 第5次跳轉



- 第五次跳轉的位址（0x4200）同樣對映到I\$的組8。但是，此時組已滿：tagv_mb_ff = 1111。因此，必須將新區塊寫入通路1：replace_way_mb_any = ic_wr_en = 0001。組8的LRU狀態更新如下：way_status_new = 011。
- I\$區塊從DDR記憶體中讀取，並以64位元區塊的形式寫入I\$。上圖描繪了將新區塊的標籤和前兩條指令寫入組8的過程：
 - ic_rw_addr_q[11:4] = 00100000（組8）
 - ic_tag_wr_data[19:0] = 0x4
 - ic_wr_data1[31:0] = 0x800fc06f (j Set8_Block1)
 - ic_wr_data2[31:0] = 0x00008067 (ret)

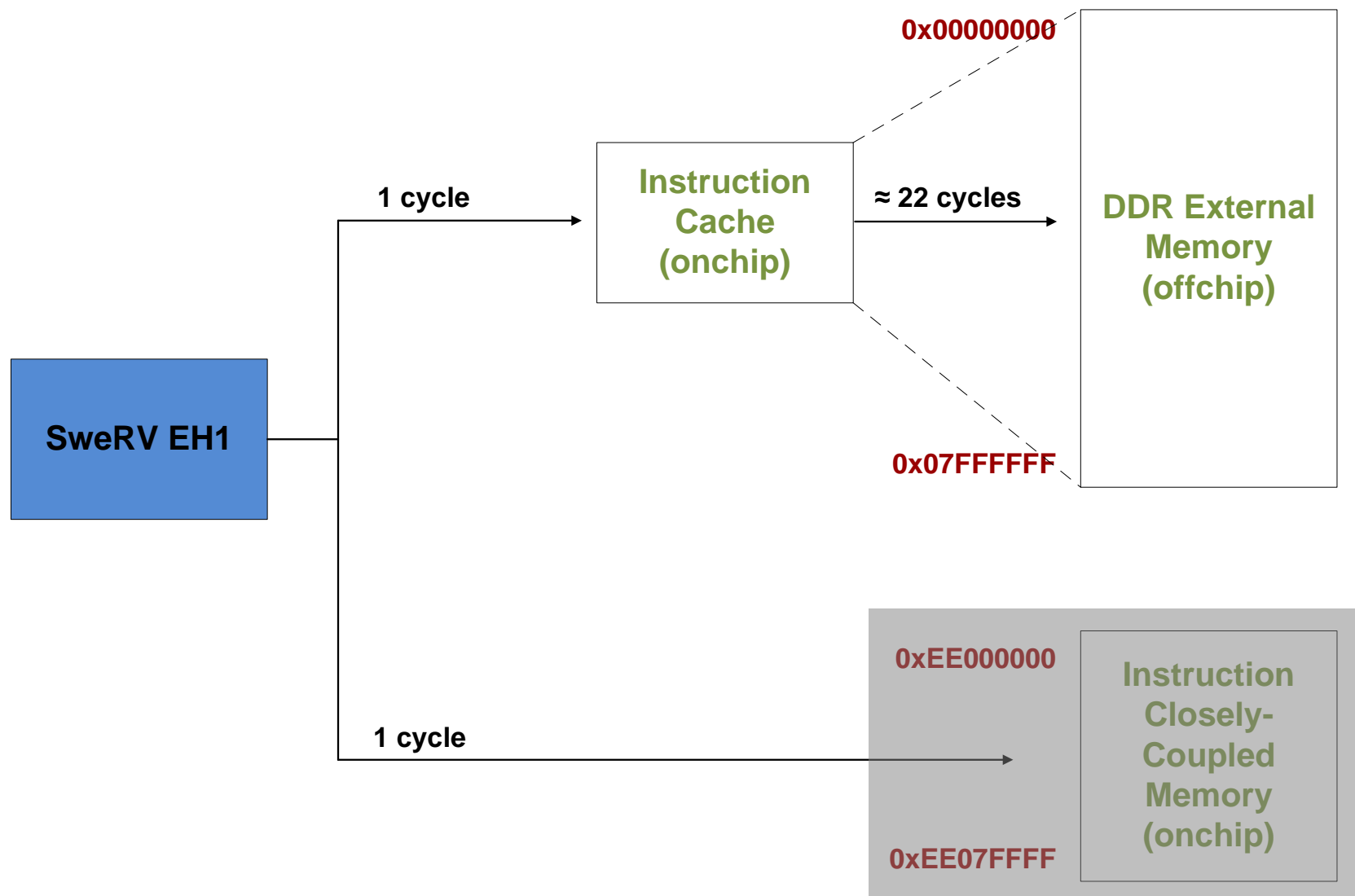
實驗20： ICCM、DCCM 和基準測試



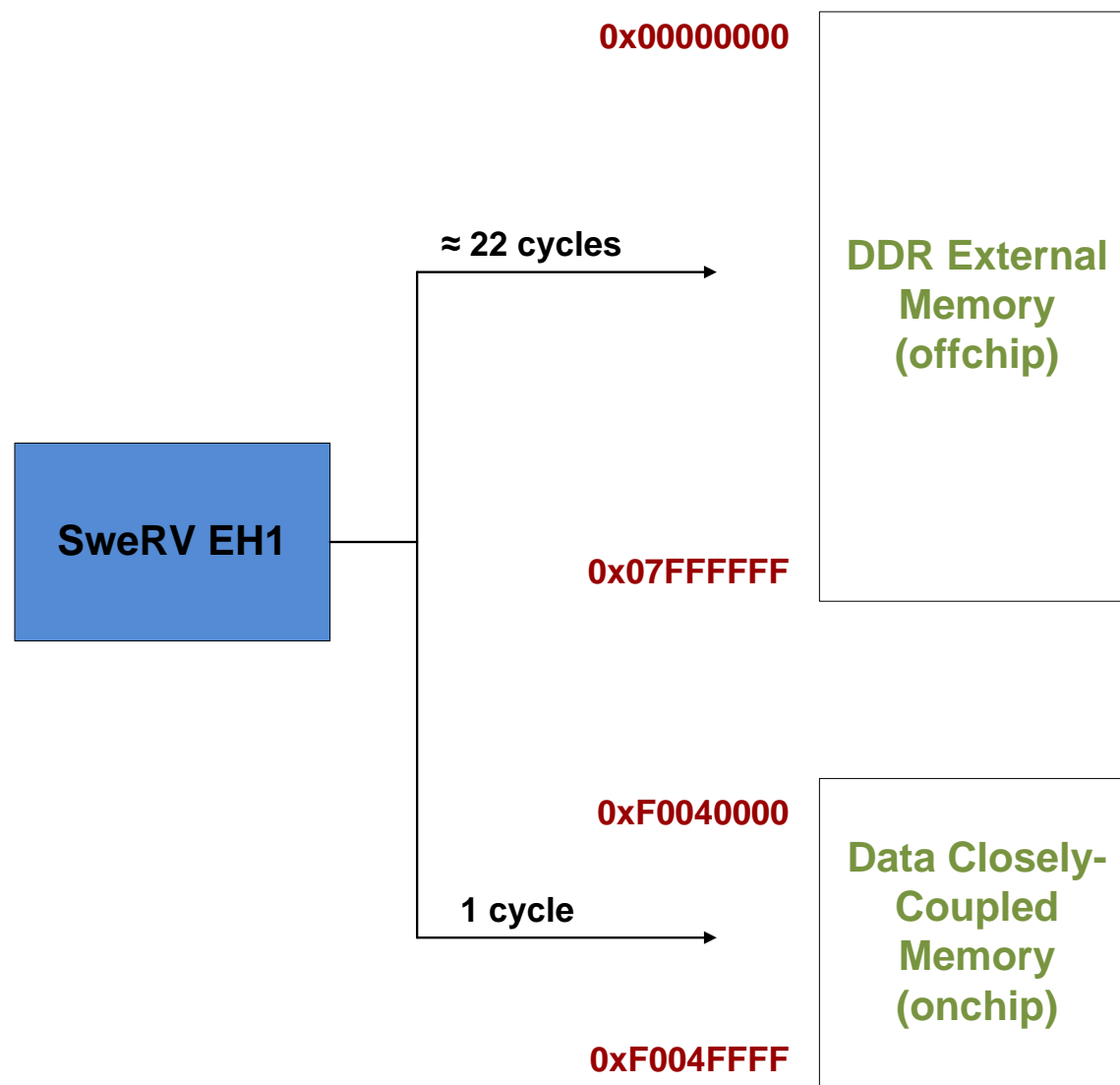
RVfpga實驗20：簡介

- 本實驗將分析SweRV EH1處理器中提供的暫存記憶體（ICCM和DCCM），然後會提供幾個基準測試範例和練習，以展示實驗11-20中的一些概念。
- 回顧一下，RVfpga系統包括兩個暫存記憶體：
 - 一個用於儲存資料，稱為資料緊密耦合記憶體（Data Closely-Coupled Memory，DCCM）
 - 一種用於儲存指令，稱為指令緊密耦合記憶體（Instruction Closely-Coupled Memory，ICCM）

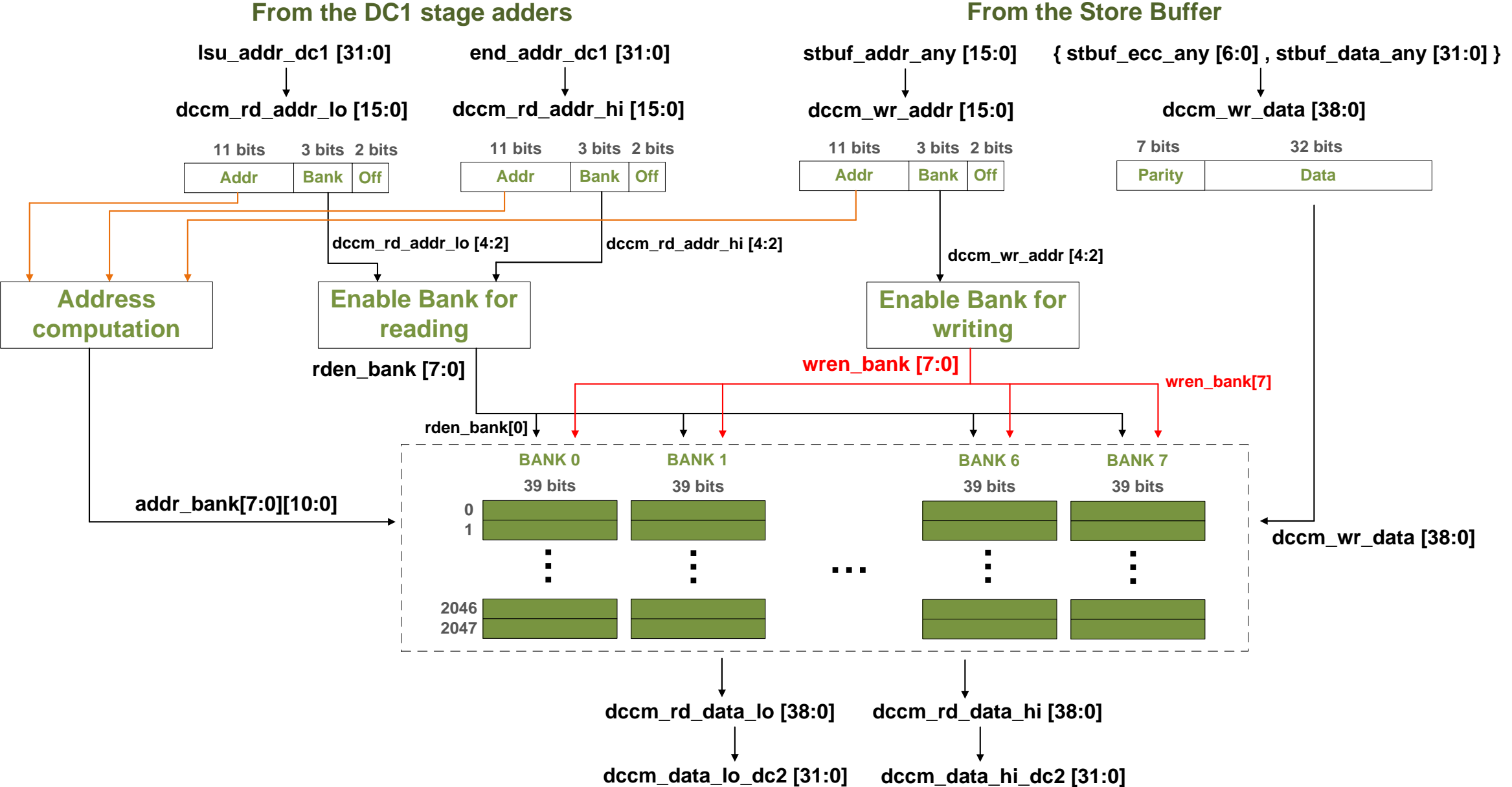
RVfpga實驗20：指令記憶體－位址空間



RVfpga實驗20：資料記憶體－位址空間



RVfpga實驗20：DCCM組態和操作



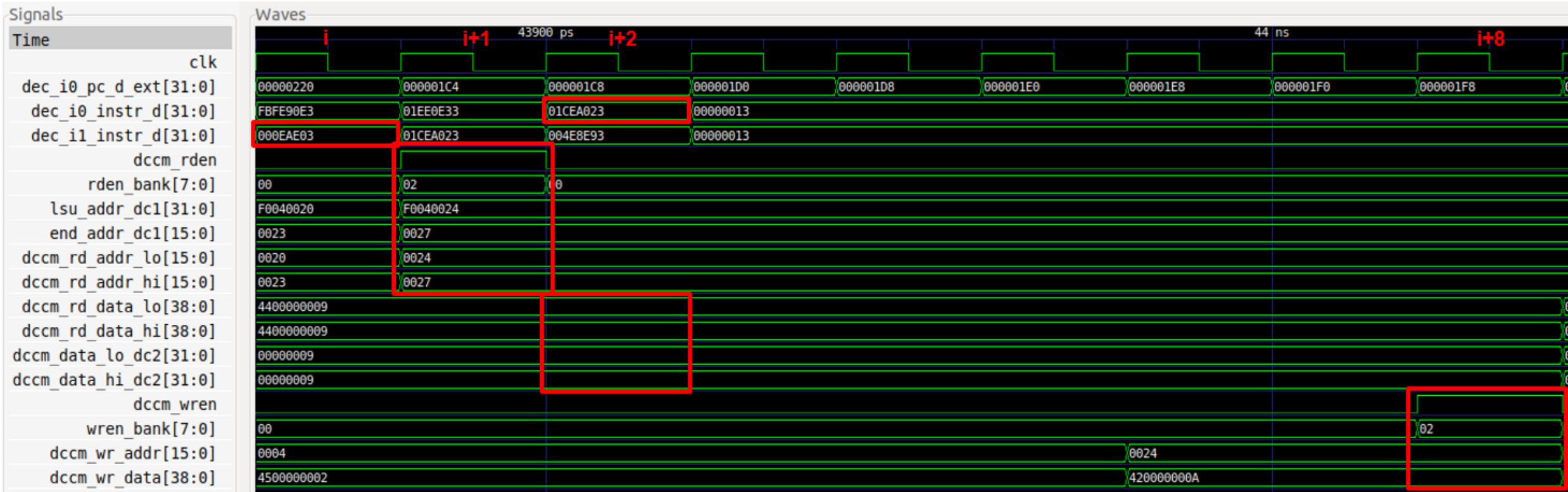
RVfpga實驗20：DCCM – 範例

```
la t4, D
li t5, 50
li t0, 1000
la t6, D
add t6, t6, t0
li t5, 1

REPEAT_Access:
    lw t3, (t4)
    add t3, t3, t5
    sw t3, (t4)
    add t4, t4, 4
    INSERT_NOPS_10
    INSERT_NOPS_10
bne t4, t6, REPEAT_Access    # Repeat the loop
```



RVfpga實驗20：DCCM – 模擬



RVfpga實驗20：DCCM – 分析

- **週期i**：在通路1中對lw指令進行解碼：`dec_i1_instr_d = 0x000eae03`。
- **週期i+1**：在DC1階段產生位址，然後將位址提供給DCCM：
 - `lsu_addr_dc1[31:0] = 0xF0040024` → `dccm_rd_addr_lo[15:0] = 0x0024`
 - `end_addr_dc1[15:0] = 0x0027` → `dccm_rd_addr_hi[15:0] = 0x0027`

完成位址檢查後，將啟用DCCM的讀取操作：`dccm_rden = 1`。由於存取是字對齊的，因此僅讀取第二個儲存區：`rden_bank = 0x02`（二進位值為00000010）。

- **週期i+2**：從DCCM取得讀取資料，並將其提供給核心：
 - `dccm_rd_data_lo = 0x4400000009` → `dccm_data_lo_dc2 = 0x00000009`
 - `dccm_rd_data_hi = 0x4400000009` → `dccm_data_hi_dc2 = 0x00000009`
- **週期i+8**：將讀取值加1的結果（`0x00000009 + 1 = 0x0000000A`）寫入DCCM：
 - `dccm_wren = 1`
 - `wren_bank = 0x02`（二進位值為00000010；即第二個儲存區）
 - `dccm_wr_addr = 0x0024`
 - `dccm_wr_data = 0x420000000A`

RVfpga實驗20：基準測試

- 如需對處理器進行基準測試，應執行程式（或程式集）並測定處理器效能。透過在多個處理器上執行相同的基準，可對這些處理器進行比較。
- 本實驗引入了兩種常用的基準：**CoreMark**和**Dhrystone**。我們使用Chips Alliance提供的原始程式碼（<https://github.com/chipsalliance/Cores-SweRV>）對它們進行了修改，使其能夠適用於RVfpga系統。在練習中，我們還將使用實驗5中的**影像處理**應用程式作為基準。
- 對於任何基準，硬體計數器都將測量各種處理器事件。除了修改基準以便使用RISC-V硬體計數器外，我們還新增了一些對使用DCCM/ICCM和編譯器最佳化的支援。
- 接下來，我們將介紹改變記憶體組態和編譯器最佳化時的**CoreMark**效能。

RVfpga實驗20：指標

- **CoreMark**指標
 - CoreMark執行迴圈的多次迭代。
 - **CoreMark**分數（**CM**）：每秒鐘完成的迭代次數（即，迭代數/秒）。
 - **CM/MHz**：CM除以單位為MHz的時脈頻率（也稱為Iterat/Sec/MHz，即迭代數/秒/MHz）。
- 回顧一下，由於SweRV EH1為雙路超標量處理器，因此其**理想IPC**為**2**。

RVfpga實驗20：各種條件下的CoreMark

	編譯器 = 除錯 外部記憶體	編譯器 = 除錯 DCCM	編譯器 = 最佳化 DCCM
CM/MHz	0.47	1.88	3.47
指令數	約50萬	約50萬	30.9萬
週期數	約200萬	約50萬	28.8萬
IPC（指令數/週期）	0.25	~1	~1
資料匯流排交易	約133,000 （全部轉到外部 記憶體）	0 （由於DCCM）	0 （由於DCCM）
指令匯流排交易	392 （由於I\$）	392 （由於I\$）	392 （由於I\$）