

# RVfpga

## 了解计算机架构的完整课程

# 致谢

## 作者

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

## 顾问

David Patterson教授

## 贡献者

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

## 合作者

José Ignacio Gómez教授  
Christian Tenllado教授  
Daniel León教授  
Katzalin Olcoz教授  
Alberto del Barrio教授  
Fernando Castro教授  
Manuel Prieto教授

Francisco Tirado教授  
Román Hermida教授  
Ataur Patwary教授  
Cathal McCabe  
Dan Hugo  
Braden Harwood  
David Burnett教授

Gage Elerding  
Brian Cruickshank教授  
Deepen Parmar  
Thong Doan  
Oliver Rew  
Niko Nikolay  
Guanyang He

## 赞助商与支持者



# 简介

---

- RISC-V FPGA (**RVfpga**) 是一套教学包，其中包含一套指令、工具和实验，用于展示如何：
  - 确定商用RISC-V片上系统（SoC）的目标**FPGA**
  - 对RISC-V SoC进行编程
  - 向RISC-V SoC添加更多功能
  - 分析和修改RISC-V内核与存储器层级
- 该教学包由**Imagination Technologies**与其学术和行业合作伙伴共同开发。
- RVfpga系统围绕Chips Alliance的**SweRVolf SoC**（基于Western Digital的RISC-V **SweRV EH1**内核）构建。

# RVfpga概述

- **RVfpga教学包提供:**
  - 一门内容全面且免费发布的完整**RISC-V**课程
  - 一种容易上手的动手实验方法来了解RISC-V处理器和RISC-V生态系统
  - 一款以低成本**FPGA**（在许多大学和公司都比较常用）为目标的**RISC-V**系统。
- 完成RVfpga课程后，用户将了解并掌握如何使用和修改**RISC-V**处理器、**SoC**和生态系统。

# 第二门课程： RVfpga-SoC

- **RVfpga-SoC**课程是第二门课程：
  - 介绍如何构建**RISC-V SoC**、安装**Zephyr RTOS**（实时操作系统）并在其上运行程序（包括简单的**Tensorflow**程序）。
  - 提供5个实验。
- 两门课程（RVfpga和RVfpga-SoC）均可在以下网址单独下载（注册后免费）：  
<https://university.imgtec.com/rvfgpa/>
- 余下的幻灯片重点关注**RVfpga**课程。

# RVfpga课程內容

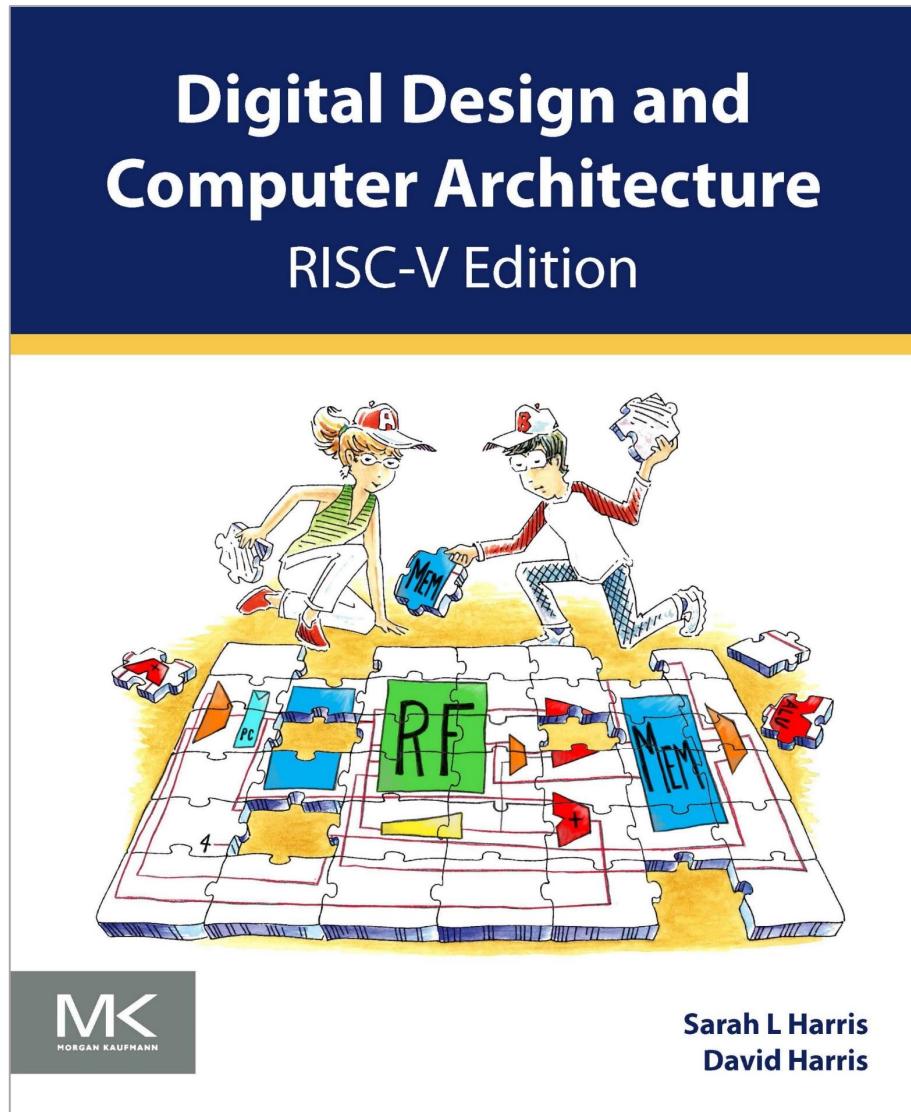
# RVfpga内容

- 入门指南
  - 快速入门指南
  - RISC-V架构和RVfpga概述
  - 安装工具（VSCode、PlatformIO、Vivado、Verilator和Whisper）
  - 通过硬件和仿真运行RVfpga系统
- 实验
  - **1-10:** 在Vivado中编译RVfpga系统，编程RVfpga系统，通过添加外设扩展RVfpga系统（2020年11月发布）
  - **11-20:** 分析和修改RVfpga的RISC-V内核和存储器系统（计划于2021年第4季度发布）

- 2-3学期课程
  - 本科（实验1-10）
  - 硕士/本科高年级（实验11-20）
- 知识储备要求
  - 基本了解数字设计、高级编程（最好是C语言）、指令集架构和汇编语言设计、处理器微架构和存储器系统（相关信息参见《数字设计和计算机体系结构》RISC-V版本（作者：Harris & Harris）© Elsevier，预计出版时间：2021年夏季）
  - 在整个RVfpga课程中，将通过动手实验来扩展和巩固上述主题

# 教材

在开始RVfpga课程之前建议了解以下教材：《数字设计和计算机体系结构》，**RISC-V**版本（作者：Harris & Harris）© Elsevier，2021年。



# 如何获取RVfpga



Imagination大学计划网站

- 通过以下网址注册Imagination大学计划(IUP) - 面向全球教师、研究人员和学生：  
<https://university.imgtec.com>
  - 接收发布更新和通知
  - 申请并下载资料
  - 支持论坛：PowerVR、RVfpga和AI论坛；IUP论坛（面向课程/教学讨论）
- 社交媒体：
  - IUP主管Robert Owen: @UniPgm
  - Imagination Technologies: @ImaginationTech
  - 微信和微博: ImaginationTech

# RVfpga所需的软件和硬件

## 软件

Xilinx Vivado 2019.2 WebPACK

PlatformIO – Microsoft Visual Studio Code的扩展 – 支持Chips Alliance平台，其中包括：RISC-V工具链、OpenOCD、Verilator HDL仿真器、WD Whisper指令集仿真器（ISS）

除FPGA开发板外，所有软硬件均可免费使用（FPGA开发板的官方价为265美元，学术优惠价为199美元）

## 硬件\*

Digilent的Nexys A7/Nexys 4 DDR FPGA开发板

\*可选：所有实验只需通过仿真即可完成，因此该硬件只是推荐使用，并非必须使用。

## RISC-V内核和SOC

内核：Western Digital的SweRV EH1\*\*

SOC：Chips Alliance的SweRVolf\*\*

\*\*开源 - 在RVfpga软件包中提供。

# 支持的平台

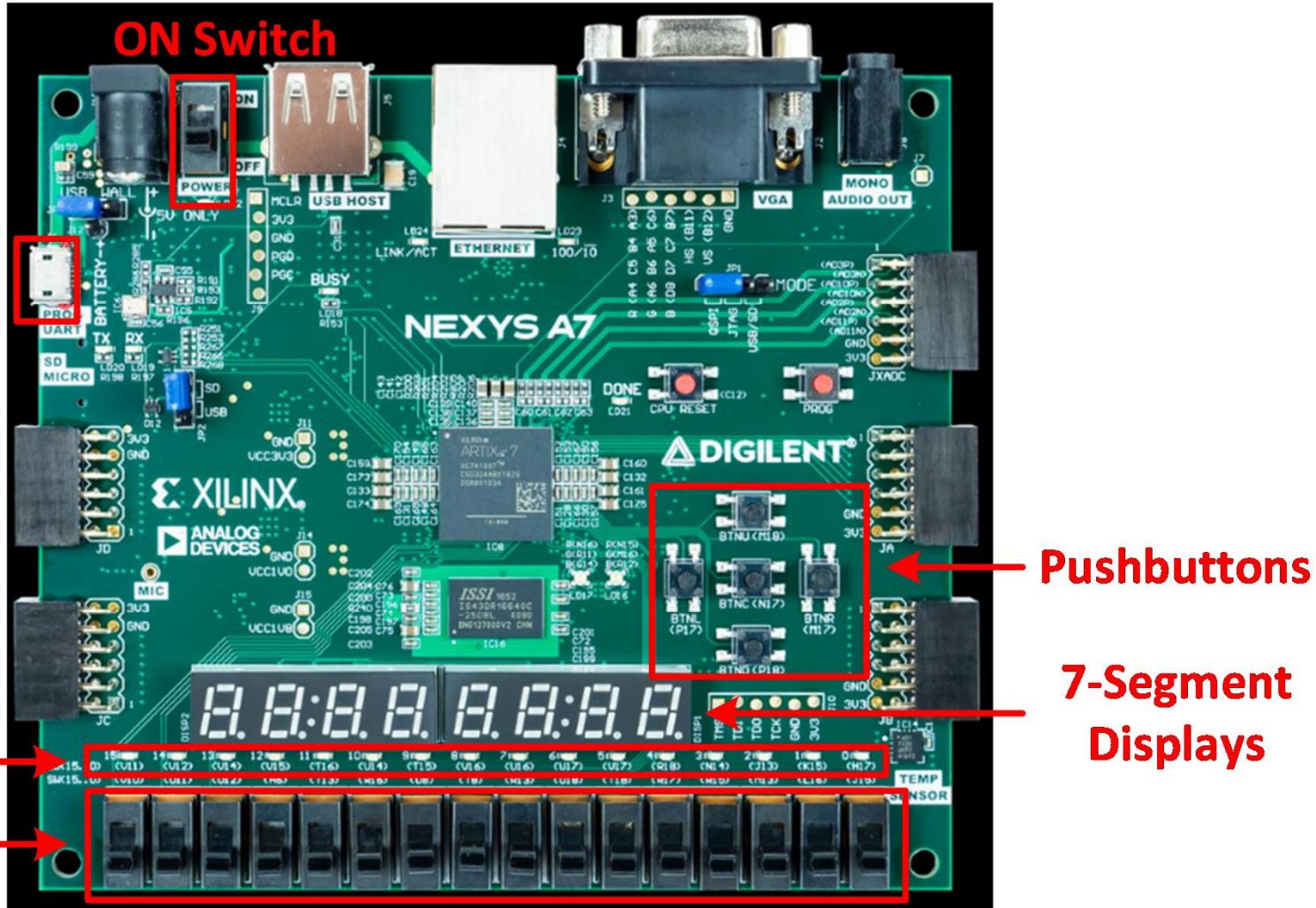
---

- 操作系统
  - **Ubuntu 18.04** (更高版本也可能适用)
  - **Windows 10**
  - **macOS**

# RVfpga软件工具

- **Xilinx的Vivado IDE**
  - 查看RVfpga源文件（Verilog/SystemVerilog）和层级
  - 为以Nexys A7开发板为目标的RVfpga创建bit文件（FPGA配置文件）
- **PlatformIO – Visual Studio Code (VSCode) 的扩展**
  - 将RVfpga系统下载到Nexys A7开发板
  - 在RVfpga系统上编译、下载、运行和调试C程序和汇编程序
- **Verilator – 一款HDL（硬件描述语言）仿真器**
  - 在HDL（低）级别仿真RVfpga系统，以分析其内部信号

# Nexys A7-100T FPGA开发板

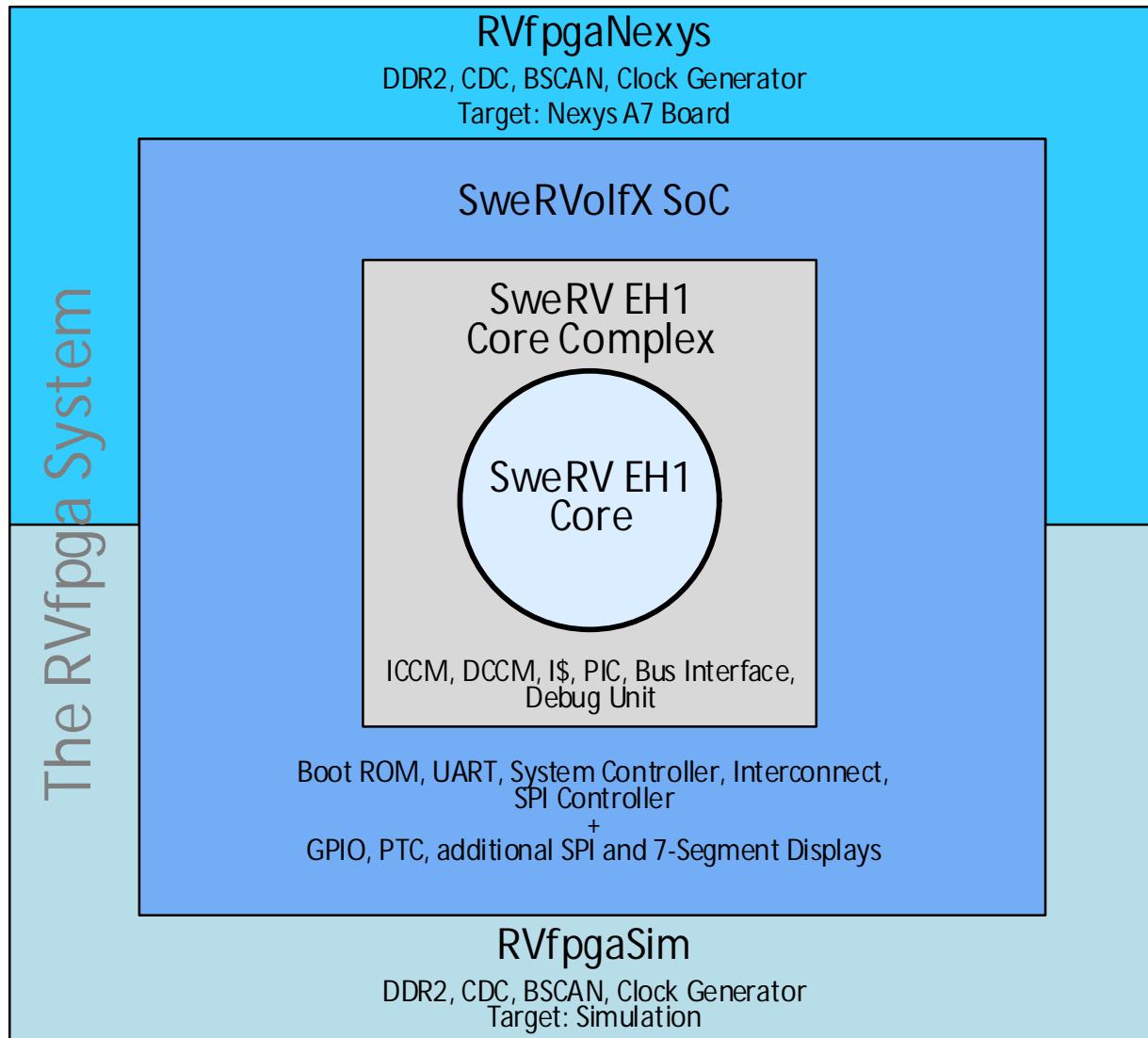


开发板图片来源: <https://reference.digilentinc.com/>

- 包括Artix-7现场可编程门阵列（FPGA）
- 包括外设（即LED、开关、按钮、7段显示屏、加速计、温度传感器和麦克风等）
- 可从digilentinc.com和其他供应商处购买

# RISC-V内核和SoC

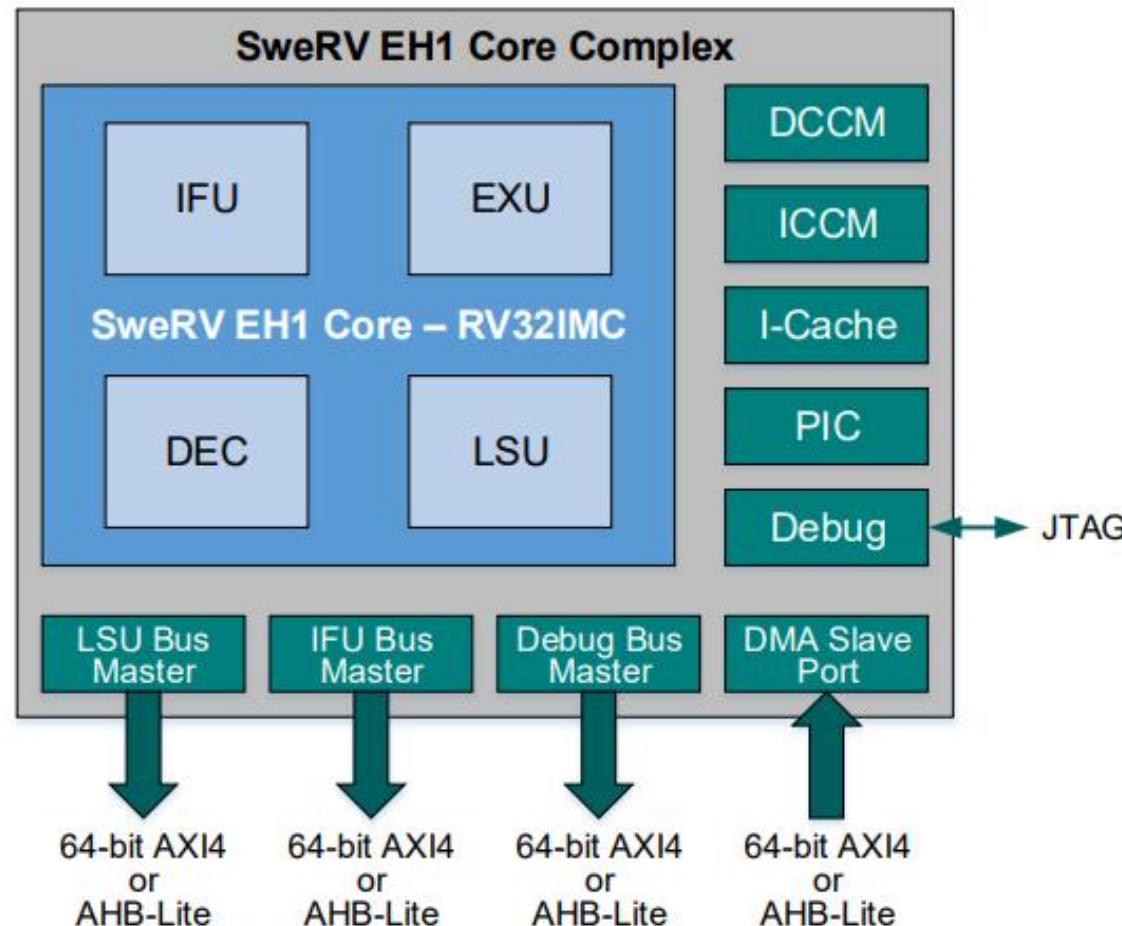
# RVfpga层级



# RVfpga层级

名称	说明
SweRV EH1内核	由Western Digital开发的开源商用RISC-V内核 ( <a href="https://github.com/chipsalliance/Cores-SweRV">https://github.com/chipsalliance/Cores-SweRV</a> )。
SweRV EH1内核组合	一种增加了存储器（ICCM、DCCM和指令高速缓存）、可编程中断控制器（Programmable Interrupt Controller, PIC）、总线接口和调试单元的SweRV EH1内核 ( <a href="https://github.com/chipsalliance/Cores-SweRV">https://github.com/chipsalliance/Cores-SweRV</a> )。
SweRVolfX (扩展SweRVolf)	我们在RVfpga课程中使用的片上系统。它是SweRVolf的扩展。 <u>SweRVolf</u> ( <a href="https://github.com/chipsalliance/Cores-SweRVolf">https://github.com/chipsalliance/Cores-SweRVolf</a> )：一种围绕SweRV EH1内核组合构建的开源SoC。它增加了引导ROM、UART接口、系统控制器、互连（AXI互连、Wishbone互连和AXI转Wishbone桥）以及SPI控制器。 <u>SweRVolfX</u> ：与SweRVolf相比增加了4个新外设：GPIO、PTC、一个额外的SPI以及用于8位7段显示屏的控制器。
RVfpgaNexys	以Nexys A7开发板及其外设为目标的SweRVolfX SoC。它增加了DDR2接口、CDC（时钟域交叉）单元、BSCAN逻辑（用于JTAG接口）和时钟发生器。 RVfpgaNexys与SweRVolf Nexys基本相同 ( <a href="https://github.com/chipsalliance/Cores-SweRVolf">https://github.com/chipsalliance/Cores-SweRVolf</a> )，只是后者基于SweRVolf。
RVfpgaSim	SweRVolfX SoC具有配套的测试平台和用于仿真的AXI内存。 RVfpgaSim与SweRVolf Sim基本相同 ( <a href="https://github.com/chipsalliance/Cores-SweRVolf">https://github.com/chipsalliance/Cores-SweRVolf</a> )，只是后者基于SweRVolf。

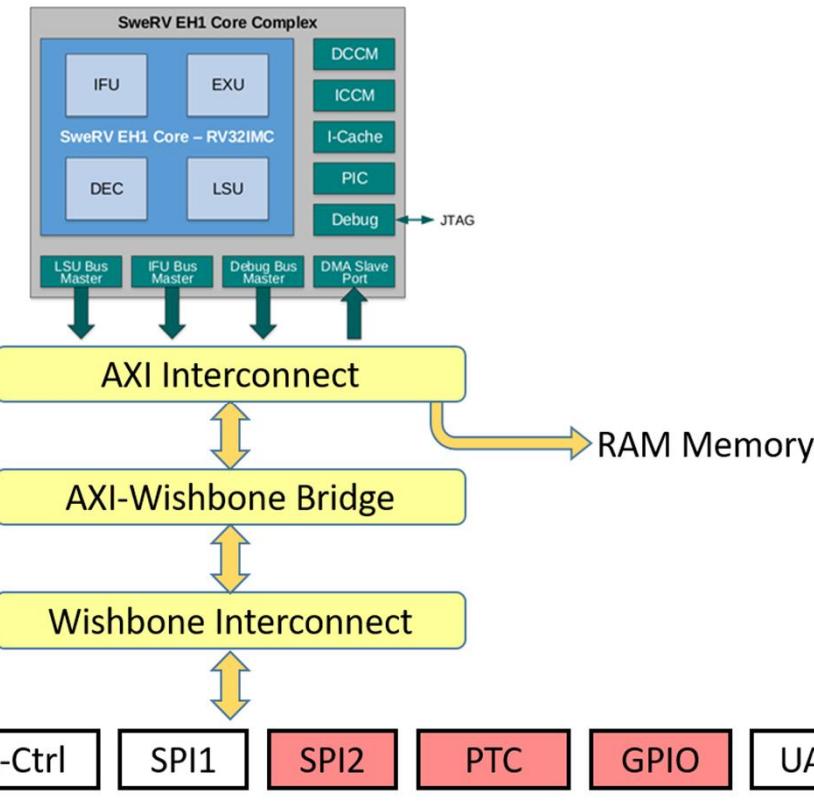
# SweRV EH1内核和SweRV EH1内核组合



- Western Digital的开源内核
- 32位（RV32ICM）超标量内核，具有双发射9级流水线
- 独立的指令和数据存储器（ICCM和DCCM），与内核紧密耦合
- 4路组相连指令缓存，具有奇偶校验或ECC保护
- 可编程中断控制器
- 符合RISC-V调试规范的内核调试单元
- 系统总线：AXI4或AHB-Lite

图片来源：[https://github.com/chipsalliance/Cores-SweRV/blob/master/docs/RISC-V\\_SweRV\\_EH1\\_PRM.pdf](https://github.com/chipsalliance/Cores-SweRV/blob/master/docs/RISC-V_SweRV_EH1_PRM.pdf)

# SweRVolfX SoC

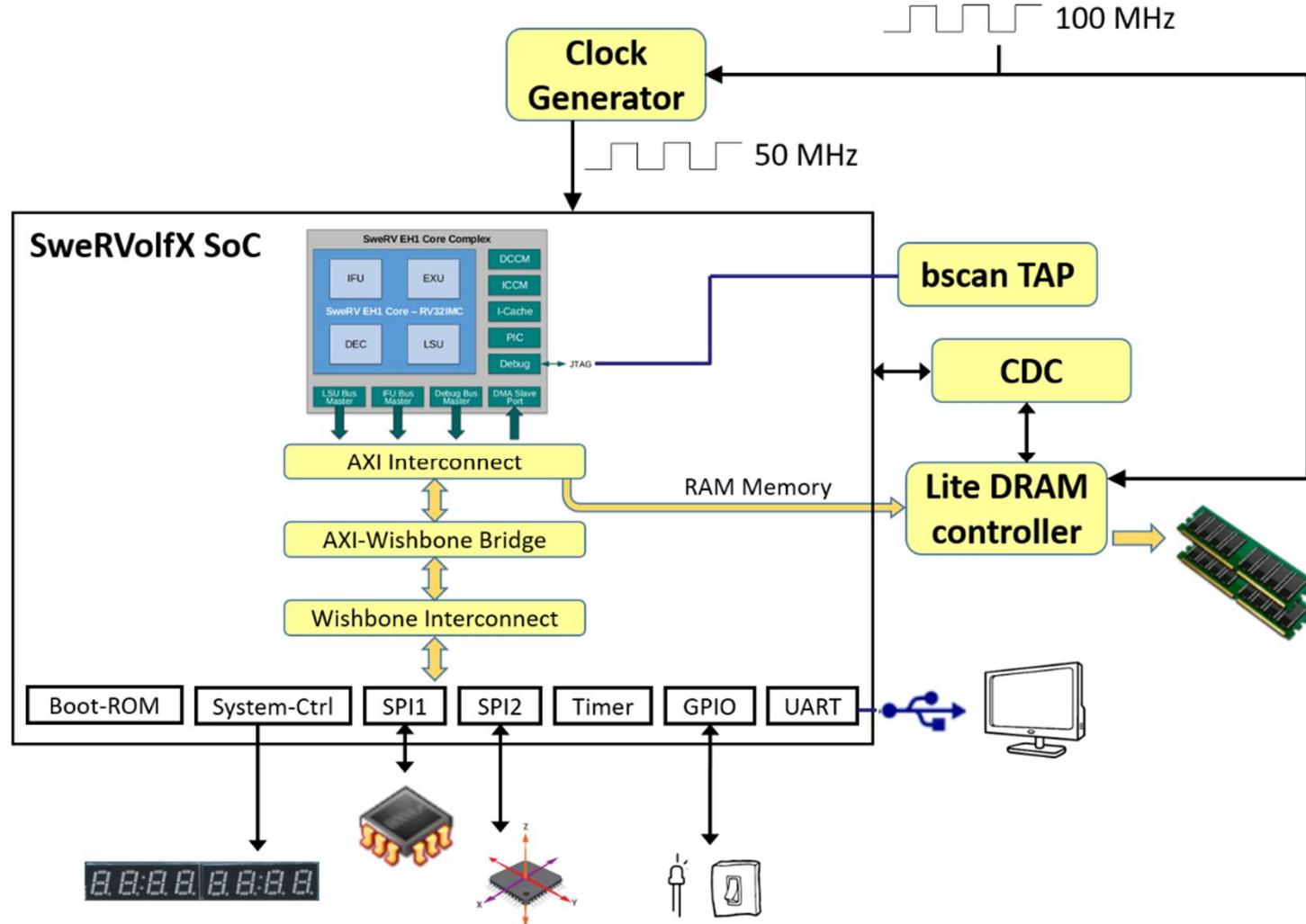


SweRVolfX存储器映射

系统	地址
引导ROM	0x80000000 - 0x80000FFF
系统控制器	0x80001000 - 0x8000103F
SPI1	0x80001040 - 0x8000107F
SPI2	0x80001100 - 0x8000113F
定时器	0x80001200 - 0x8000123F
GPIO	0x80001400 - 0x8000143F
UART	0x80002000 - 0x80002FFF

- Chips Alliance的开源片上系统（SoC）
- SweRVolf使用SweRV EH1内核。SweRVolf包括引导ROM、UART、系统控制器和SPI控制器（SPI1）
- SweRVolfX通过添加另一个SPI控制器（SPI2）、GPIO（通用输入/输出）、8位7段显示屏和PTC（以红色显示）扩展SweRVolf。
- SweRV EH1内核使用AXI总线，而外设使用Wishbone总线，因此SoC还具有AXI转Wishbone桥

# RVfpgaNexys



- **RVfpgaNexys:** 以Nexys A7 FPGA开发板为目标的SweRVolfX SoC（添加若干外设）：
  - 内核和系统：
    - SweRVolfX SoC
    - Lite DRAM控制器
    - 时钟发生器、时钟域和JTAG端口的BSCAN逻辑
  - Nexys A7 FPGA开发板上使用的外设：
    - DDR2存储器
    - 采用USB连接的UART
    - SPI闪存
    - 16个LED和16个开关
    - SPI加速计
    - 8位7段显示屏

- SweRVolfX SoC还可以包括Verilog测试程序以允许其进行仿真。
- **RVfpgaSim**是包装在HDL仿真器所用测试平台中的SweRVolfX SoC。

# RVfpga系统扩展

- RVfpga系统在实验6-10中进一步扩展：
  - 添加**GPIO**控制器，与板上Nexys A7按钮连接
  - 修改**7段显示屏**控制器
  - 新增定时器模块，以便使用板上三色**LED**
  - 新增外部中断源

# RVfpga实验概述

# RVfpga实验概述

## 第1部分：实验1-10

- Vivado项目和编程
- I/O系统

## 第2部分：实验11-20

- RISC-V内核
- RISC-V存储器系统
- RISC-V基准测试

所有实验均包括使用和修改RVfpga系统的练习，通过实际操作来加深理解。

# 实验内容

## 第1部分

编号	标题
1	创建Vivado项目
2	C语言编程
3	RISC-V汇编语言
4	函数调用
5	图像处理
6	I/O简介
7	7段显示屏
8	定时器
9	中断驱动I/O
10	串行总线

## 第2部分

编号	标题
11	SweRV EH1配置和性能监视
12	算术/逻辑指令：add指令
13	访存指令：lw和sw指令
14	结构冒险
15	数据冒险
16	控制冒险：分支指令
17	超标量执行
18	添加新功能（指令和计数器）
19	存储器层级：指令高速缓存
20	ICCM、DCCM和基准测试

# RVfpga实验1-10

展示如何查看RVfpga系统源代码（Verilog/SV）并确定其目标FPGA（实验1）、如何编写C程序和汇编程序（实验2-5）以及如何修改RVfpga系统以添加外设（实验6-10）。

- 实验0: RVfpga实验概述
- 实验1: 创建Vivado项目
- 实验2: C语言编程
- 实验3: RISC-V汇编语言
- 实验4: 函数调用
- 实验5: 图像处理: C语言和汇编语言
- 实验6: I/O简介
- 实验7: 7段显示屏
- 实验8: 定时器
- 实验9: 中断驱动I/O
- 实验10: 串行总线

编程

I/O系统

# RVfpga实验1-5：Vivado项目和编程

- **实验1：创建Vivado项目：**构建Vivado项目并确定RVfpgaNexys的目标FPGA开发板，然后在Verilator中仿真RVfpgaSim。
- **实验2：C语言编程：**在PlatformIO中编写一个C程序，然后在RVfpgaNexys/RVfpgaSim/Whisper上运行/调试。此外，还介绍了Western Digital的开发板支持包和平台支持包（BSP和PSP），用于支持终端打印等操作。
- **实验3：RISC-V汇编语言：**在PlatformIO中编写一个RISC-V汇编程序，然后在RVfpgaNexys/RVfpgaSim/Whisper上运行/调试。
- **实验4：函数调用：**函数调用、C库和RISC-V调用约定简介。
- **实验5：图像处理：C语言和汇编语言：**将C代码嵌入汇编代码。

# RVfpga实验6-10：I/O和外设

- **实验6：I/O简介：**存储器映射I/O和RVfpga系统开源GPIO模块简介。
- **实验7：7段显示屏：**构建一个7段显示屏译码器并将其集成到RVfpga系统中。
- **实验8：定时器：**了解和使用定时器与定时器控制器。
- **实验9：中断驱动I/O：**RVfpga系统中断支持和中断驱动I/O使用简介。
- **实验10：串行总线：**串行接口（SPI、I2C和UART）简介。介绍如何使用采用SPI接口的板上加速计。

# RVfpga实验11-20：RISC-V内核

- **实验11：**了解SweRV EH1配置、内核结构和性能监视。
- **实验12、13和16：**检查流水线中的指令流（算术/逻辑、存储器、跳转和分支）。
- **实验14-16：**了解冒险及其处理方法
- **实验16：**了解和修改分支预测器
- **实验17：**探究超标量执行。
- **实验18：**添加新指令和硬件计数器。
- **实验19：**了解存储器层级和I\$。
- **实验20：**使能ICCM和DCCM（指令和数据紧密耦合存储器）并使用基准测试来比较性能。

# RVfpga受众和过往成绩

- 目标受众
  - 电气工程、计算机科学或计算机工程专业的本科生
  - 有兴趣学习RISC-V架构的学者和行业专业人士
- **Imagination大学计划 (IUP)** 过往成绩：开展了MIPSfpga计划：
  - 2015年4月推出
  - 800所大学参与
  - 荣获2015年欧洲Elektra最佳教育支持奖

# RVfpga

# 快速入门指南

# 快速入门指南概述

- 安装VSCode和PlatformIO
- 在**RVfpgaNexys**上运行示例程序

# 安装PlatformIO和VSCode

- 安装**VSCode**
  - <https://code.visualstudio.com/Download>
  - 对于Ubuntu和macOS，需安装**Python**（Windows不需要该步骤）
- 在VSCode中安装**PlatformIO**扩展功能
- 安装Nexys A7开发板驱动程序（见RVfpga入门指南说明）

# 将RVfpgaNexys下载到开发板并运行程序

- 在**PlatformIO**中：
  - 打开示例程序（将开关的值写入LED）。程序位置：  
*[RVfpgaPath]\RVfpga\examples\LedsSwitches\_C-Lang*
  - 更新PlatformIO初始化文件（platformio.ini）中**RVfpgaNexys bit**文件的目录位置
    - 即，将以下行添加到platformio.ini中： *board\_build.bitstream\_file = [RVfpgaPath]\RVfpga\src\rvfpga.bit*
  - 将**RVfpgaNexys**下载到Nexys A7开发板（“ Project Tasks → env:swervolf\_nexys → Platform → Upload Bitstream”（项目任务 → env:swervolf\_nexys → 平台 → 上传比特流））
  - 通过按下“ Run/Debug ”（运行/调试）按钮在**RVfpgaNexys**上编译、下载和运行程序 

**[RVfpgaPath]**是计算机上**RVfpga**文件夹的位置。该文件夹随Imagination大学计划的RVfpga教学包一起提供。

# Rvfpga

# 实验说明

# 实验1： Vivado项目

# RVfpga实验1：RVfpga Vivado项目

- **Vivado**是一款Xilinx工具，用于查看、修改和合成RVfpga系统的源（Verilog）代码。
- RVfpga系统的源代码位置：  
*[RVfpgaPath]/RVfpga/src*
- 创建一个包含RVfpga系统源代码的**Vivado**项目。合成以Nexys A7开发板为目标的**RVfpgaNexys**，并创建一个包含将FPGA配置为**RVfpgaNexys**的信息的**bit**文件（也称为比特流文件）。
- 也可以使用**Verilator**（一款HDL仿真器）来仿真RVfpga系统的源代码和检查内部信号（有关如何使用Verilator的说明，请参见RVfpga入门指南）。
- RVfpga实验6-10中将广泛使用Vivado和Verilator来修改和仿真RVfpga系统。

# 实验2： C语言编程

# RVfpga实验2：C语言编程

- 创建**PlatformIO**项目
- 将示例**C**程序添加到项目中
- 将**RVfpgaNexys**下载到Nexys A7开发板上
- 将**C**程序下载到**RVfpgaNexys**中并运行/调试该程序
- 在实验结束时完成部分或全部练习
- 请记住，也可以在Verilator（使用**RVfpgaSim**）或**Whisper**中仿真该程序。

# RVfpga实验2：示例C程序

```
// 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);
}
```

该程序将开关的值写入LED。

# RVfpga实验2：存储器映射I/O地址

器件	存储器映射I/O地址
开关（Nexys A7开发板上为16个）	0x80001400（高16位）
LED（Nexys A7开发板上为16个）	0x80001404（低16位）
GPIO的输入/输出（1 = 输出， 0 = 输入）	0x80001408

# RVfpga实验2：Western Digital的BSP和PSP

- Western Digital提供：
  - **PSP**: 处理器支持包
  - **BSP**: 开发板支持包
- 这两个支持包为给定的处理器（Swerv EH1内核）和开发板（Nexys A7 FPGA开发板）提供了常用的函数。
  - **示例**: printfNexys (类似C语言的printf函数)

# RVfpga实验2：使用UART打印到终端

```
#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++;
    }
}
```

- 将以下行添加到platform.ini文件中：  
monitor\_speed = 115200
- 程序开始运行后，通过按下窗口底部的以下按钮打开PlatformIO终端：



# 实验3： RISC-V汇编语言

# RVfpga实验3：RISC-V汇编语言

- RISC-V汇编语言概述
- 创建**PlatformIO**项目
- 将示例**RISC-V**汇编程序添加到项目中
- 将**RVfpgaNexys**下载到Nexys A7开发板上
- 将**RISC-V**汇编程序下载到RVfpga中并运行/调试该程序
- 在实验结束时完成部分或全部练习
- 请记住，也可以在Verilator（使用**RVfpgaSim**）或**Whisper**中仿真该程序。

# RVfpga实验3：RISC-V汇编指令

## 常用的RISC-V汇编指令/伪指令

RISC-V汇编语言	说明	运算
add s0, s1, s2	加法	$s0 = s1 + s2$
sub s0, s1, s2	减法	$s0 = s1 - s2$
addi t3, t1, -10	加立即数	$t3 = t1 - 10$
mul t0, t2, t3	32位乘法	$t0 = t2 * t3$
div s9, t5, t6	除法	$t9 = t5 / t6$
rem s4, s1, s2	求余	$s4 = s1 \% s2$
and t0, t1, t2	按位与	$t0 = t1 \& t2$
or t0, t1, t5	按位或	$t0 = t1   t5$
xor s3, s4, s5	按位异或	$s3 = s4 ^ s5$
andi t1, t2, 0xFFB	按位与（立即数）	$t1 = t2 \& 0xFFFFFB$
ori t0, t1, 0x2C	按位或（立即数）	$t0 = t1   0x2C$
xori s3, s4, 0xABCD	按位异或（立即数）	$s3 = s4 ^ 0xFFFFABCD$
sll t0, t1, t2	逻辑左移	$t0 = t1 << t2$
srl t0, t1, t5	逻辑右移	$t0 = t1 >> t5$
sra s3, s4, s5	算术右移	$s3 = s4 >>> s5$
slli t1, t2, 30	逻辑左移（立即数）	$t1 = t2 << 30$
srlti t0, t1, 5	逻辑右移（立即数）	$t0 = t1 >> 5$
srai s3, s4, 31	算术右移（立即数）	$s3 = s4 >>> 31$

# RVfpga实验3：RISC-V汇编指令

## 常用的RISC-V汇编指令/伪指令（续）

RISC-V汇编语言	说明	运算
lw s7, 0x2C(t1)	装载字	$s7 = \text{memory}[t1+0x2C]$
lh s5, 0x5A(s3)	装载半字	$s5 = \text{SignExt}(\text{memory}[s3+0x5A]_{15:0})$
lb s1, -3(t4)	装载字节	$s1 = \text{SignExt}(\text{memory}[t4-3]_{7:0})$
sw t2, 0x7C(t1)	存储字	$\text{memory}[t1+0x7C] = t2$
sh t3, 22(s3)	存储半字	$\text{memory}[s3+22]_{15:0} = t3_{15:0}$
sb t4, 5(s4)	存储字节	$\text{memory}[s4+5]_{7:0} = t4_{7:0}$
beq s1, s2, L1	如相等则分支跳转	$\text{if } (s1 == s2), \text{PC} = L1$
bne t3, t4, Loop	如不相等则分支跳转	$\text{if } (s1 != s2), \text{PC} = \text{Loop}$
blt t4, t5, L3	如小于则分支跳转	$\text{if } (t4 < t5), \text{PC} = L3$
bge s8, s9, Done	如不相等则分支跳转	$\text{if } (s8 >= s9), \text{PC} = \text{Done}$
li s1, 0xABCD12	加载立即数	$s1 = 0xABCD12$
la s1, A	加载地址	$s1 = \text{存储变量A的存储器地址}$
nop	无操作	无操作
mv s3, s7	移动	$s3 = s7$
not t1, t2	非（求反）	$t1 = \sim t2$
neg s1, s3	求补	$s1 = -s3$
j Label	跳转	$\text{PC} = \text{标签}$
jal L7	跳转并链接	$\text{PC} = L7; ra = \text{PC} + 4$
jr s1	跳转寄存器	$\text{PC} = s1$

# RVfpga实验3：RISC-V寄存器

32个32位寄存器

名称	寄存器编号	用途
zero	x0	常数值0
ra	x1	返回地址
sp	x2	堆栈指针
gp	x3	全局指针
tp	x4	线程指针
t0-2	x5-7	临时变量
s0/fp	x8	保存变量/帧指针
s1	x9	保存变量
a0-1	x10-11	函数参数/返回值
a2-7	x12-17	函数参数
s2-11	x18-27	保存变量
t3-6	x28-31	临时变量

# RVfpga实验3：示例RISC-V汇编程序

```
• // 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
•     srlt 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
```

该程序将开关的值写入LED。

# 实验4： 函数调用

# RVfpga实验4：函数调用

- 编写具有函数调用的C程序
  - 函数也称为程序
- 使用**C**库
- RISC-V（程序）调用约定

# RVfpga实验4：具有函数的示例程序

```
// 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实验4：具有函数的示例程序

```
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实验4：C库

- 库
  - 常用函数的集合
  - 旨在方便使用常用函数（节省编程时间）
- 示例C库：
  - **math.h**（数学库）：包括诸如**sqrt**（平方根）和**cos**（余弦）等函数。
  - **stdio.h**（标准I/O库）：包括用于将值打印到屏幕（**printf**）和从用户读取值（**scanf**）在内的函数。
  - **stdlib.h**（标准库）：包括用于生成随机数（**rand**）的函数。
  - 其他...（google C库）

# RVfpga实验4：使用C库的示例程序

```
#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);
}
```

该程序将0到65535之间的随机数写入LED。

# RVfpga实验4：RISC-V调用约定

- 调用函数

jal function\_label

- 从函数返回

jr ra

- 参数

- 置于寄存器a0-a7中

- 返回值

- 置于寄存器a0中

# RVfpga实验4：RISC-V调用约定示例

## C代码

```
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汇编语言

```
# 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实验4：堆栈

- 存储器中用于保存寄存器值的暂存空间
- 堆栈指针（sp）用于保存栈顶的地址
- 存储器中的堆栈自上而下存储。因此，如果要在堆栈上留出4字（16字节）空间，应使用以下代码：

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

- 两类寄存器：
  - 保留寄存器：函数调用过程中必须保留寄存器内容（即，函数调用前后寄存器值保持不变）
  - 非保留寄存器：函数调用过程中不得保留寄存器内容（即，函数调用前后寄存器值无需保持不变）
  - 保存寄存器（s0-s11）、返回地址寄存器（ra）和堆栈指针（sp）均为保留寄存器。所有其他寄存器均为非保留寄存器。

# RVfpga实验4：保留/非保留寄存器

名称	寄存器编号	用途	保留
<b>zero</b>	x0	常数值0	-
<b>ra</b>	x1	返回地址	是
<b>sp</b>	x2	堆栈指针	是
<b>gp</b>	x3	全局指针	-
<b>tp</b>	x4	线程指针	-
<b>t0-2</b>	x5-7	临时变量	否
<b>s0/fp</b>	x8	保存变量/帧指针	是
<b>s1</b>	x9	保存变量	是
<b>a0-1</b>	x10-11	函数参数/返回值	否
<b>a2-7</b>	x12-17	函数参数	否
<b>s2-11</b>	x18-27	保存变量	是
<b>t3-6</b>	x28-31	临时变量	否

# RVfpga实验4：堆栈 - 修改后的汇编代码

## C代码

```
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汇编语言

```
# 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
```

# 实验5： C语言和汇编语言

# RVfpga实验5：结合使用C语言和汇编语言

- 示例：图像处理程序
- 有些函数用C语言编写，有些则用汇编语言编写

# RVfpga实验5：图像处理程序

- 将彩色图转换为灰度图



# RVfpga实验5：图像处理程序

- 每个像素存储为三种8位颜色: **R** = 红色, **G** = 绿色, **B** = 蓝色
- 可以通过更改**R**、**G**和**B**值来创建任何颜色
- 要将图像转换为8位灰度图 (**grey**) , 每个像素应进行如下变换:

```
grey = (306*R + 601*G + 117*B) >> 10
```

- RGB权重加起来为1024 ( $306 + 601 + 117 = 1024$ ) , 因此若要返回8位范围 (0-255) , 结果应除以1024 (即右移10位: `>> 10`)
- 有关算法的更多详细信息, 请访问:

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

# RVfpga实验5：汇编函数

```
.globl ColourToGrey_Pixel ← .globl使ColourToGrey_Pixel函数对项目中的所有文件可见
.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
```

grey = ( 306\*R + 601\*G + 117\*B ) >> 10

# RVfpga实验5：结构和数组

```
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实验5：主函数

```
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;

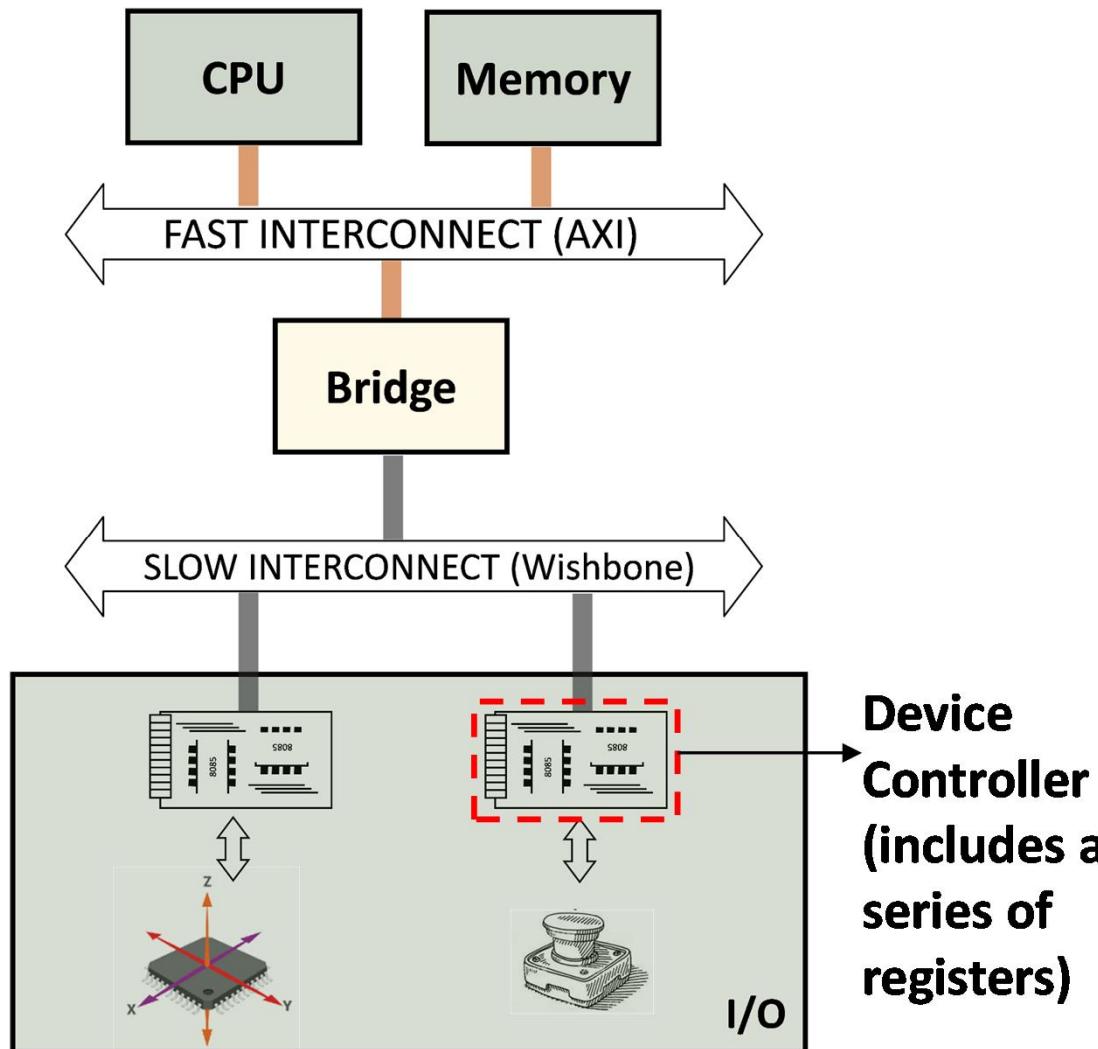
    for (i=0; i<N; i++)
        for (j=0; j<M; j++)
            Grey[i][j] = ColourToGrey_Pixel(Colour[i][j].R, Colour[i][j].G,
                                             Colour[i][j].B);
}
```

# 实验6： I/O简介

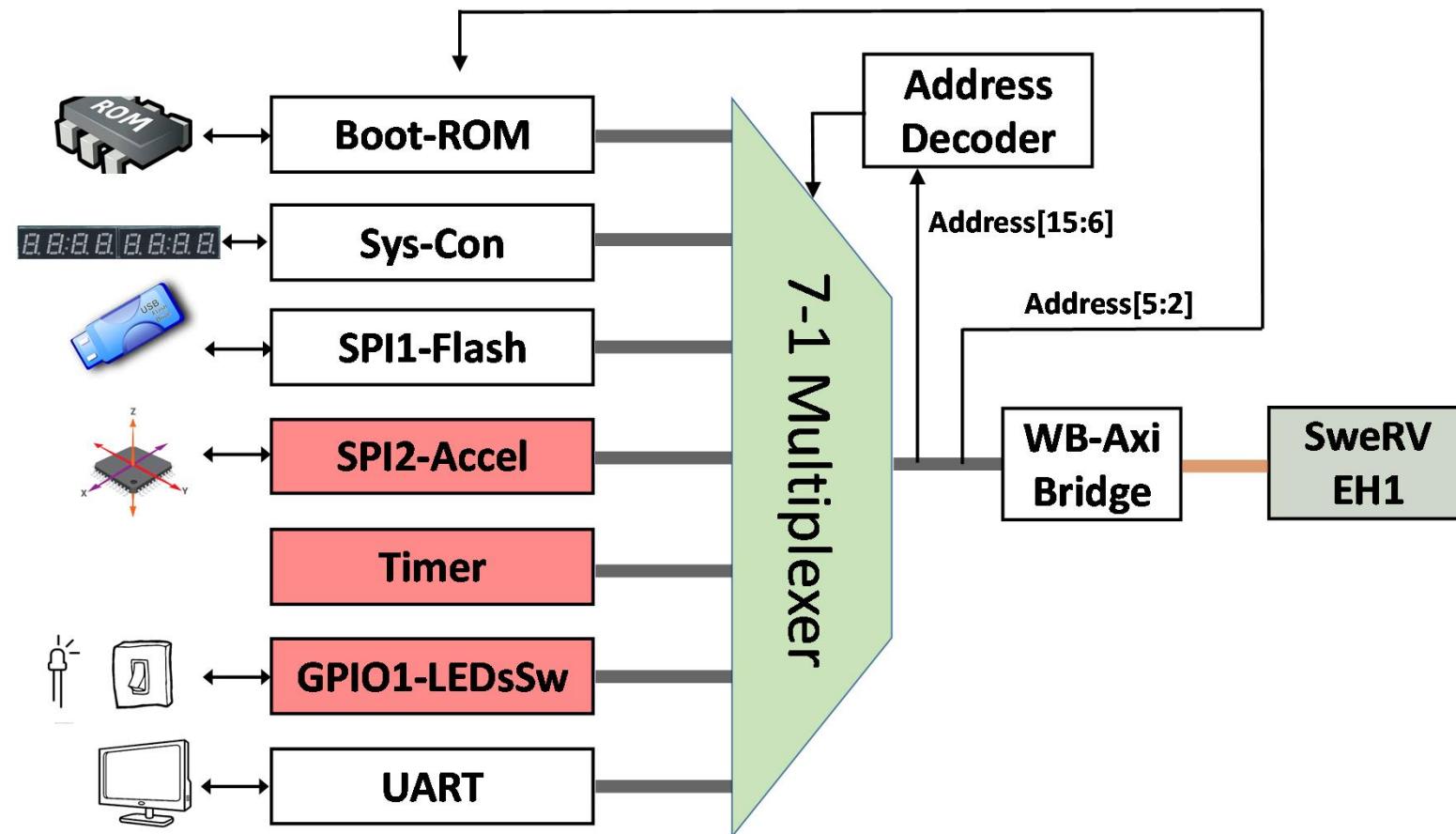
# RVfpga实验6：I/O简介

- 输入/输出（I/O）系统 - 也称为外设
- 通用I/O（GPIO）
- GPIO控制器

# RVfpga实验6：具有I/O的通用处理器



# RVfpga实验6：具有I/O的处理器



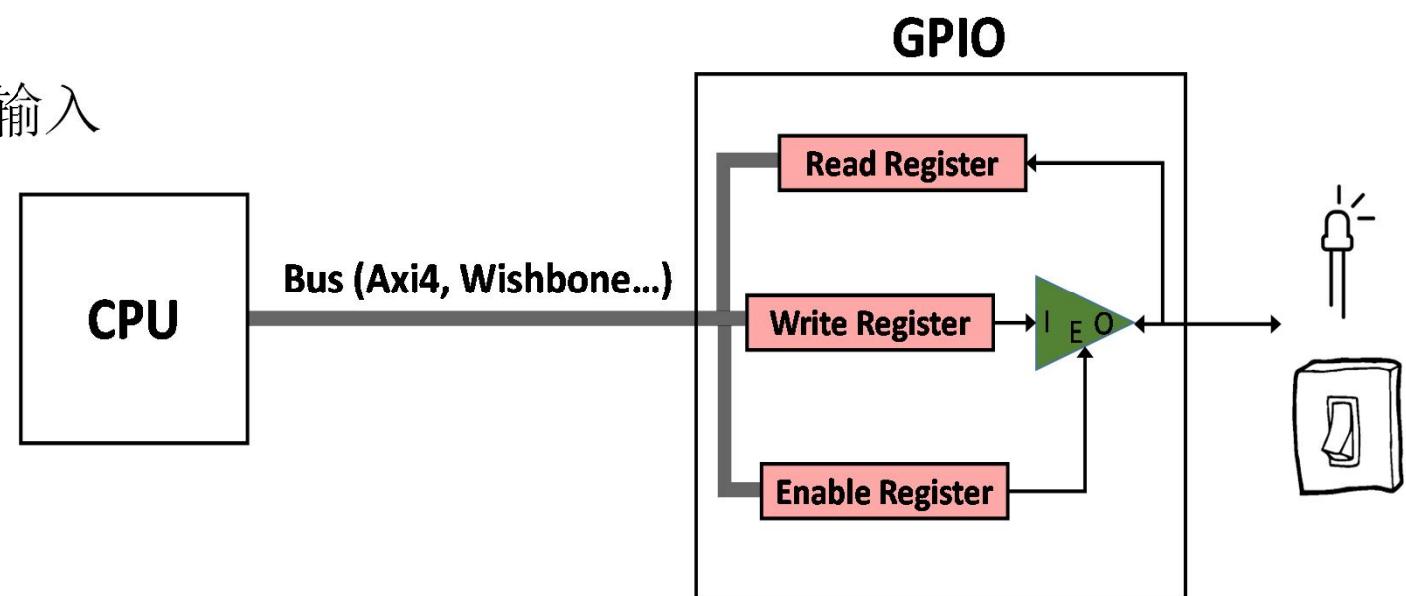
外设

- **SweRVolfX外设：**
  - 引导ROM
  - 系统控制器
  - SPI1闪存
  - UART
  - GPIO LED和开关
  - 定时器
  - SPI2加速计
  - 7段显示屏（在系统控制器内： Sys-Con）

# RVfpga实验6：通用I/O（GPIO）

- 通用I/O：
  - 允许处理器读取/写入连接到外设（例如，开关和LED）的引脚
  - 每个引脚均可配置为输入或输出（使用三态）
- 三个存储器映射寄存器：
  - 读取寄存器：从引脚读取的值
  - 写入寄存器：写入引脚的值
  - 使能寄存器：1 = 输出， 0 = 输入

外设



# RVfpga实验6：存储器映射寄存器

寄存器	存储器映射地址
读取寄存器	0x80001400
写入寄存器	0x80001404
使能寄存器	0x80001408

- 将GPIO的bit 15:0配置为输出， bit 31:16配置为输入：

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

- 读取I/O：

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

- 写入I/O：

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

# RVfpga实验6：SweRVolfX GPIO模块

- **OpenCores的GPIO模块**

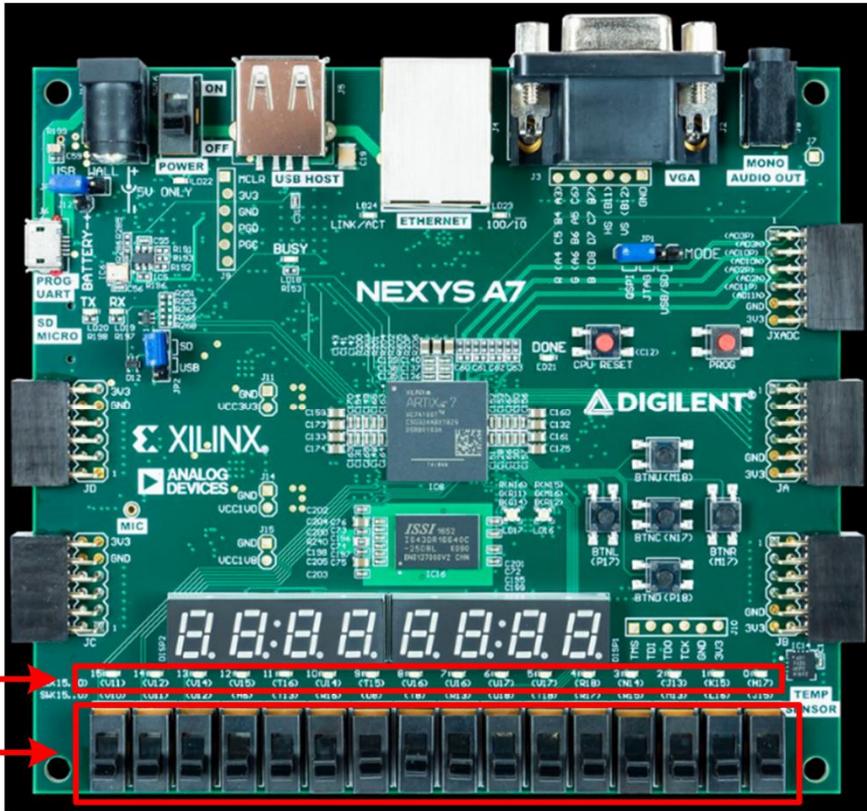
<https://opencores.org/projects/gpio>

- **最多允许32个GPIO引脚**

- 所有引脚均可单独配置为输入（使能 = 0）或输出（使能 = 1）
- 整个程序的配置都可以更改

寄存器	存储器映射地址
读取寄存器	0x80001400
写入寄存器	0x80001404
使能寄存器	0x80001408

# RVfpga实验6：存储器映射寄存器



开发板图片来源: <https://reference.digilentinc.com/>

将LED和开关映射到GPIO引脚:

- LED: 引脚[15:0] (处理器的输出)
- 开关: 引脚[31:16] (处理器的输入)

配置GPIO:

- 使能寄存器 = 0x0000FFFF (1 = 输出, 0 = 输入)  
li t0, 0x80001400  
li t1, 0xFFFF  
sw t1, 8(t0) # Enable Register = 0xFFFF

写入LED:

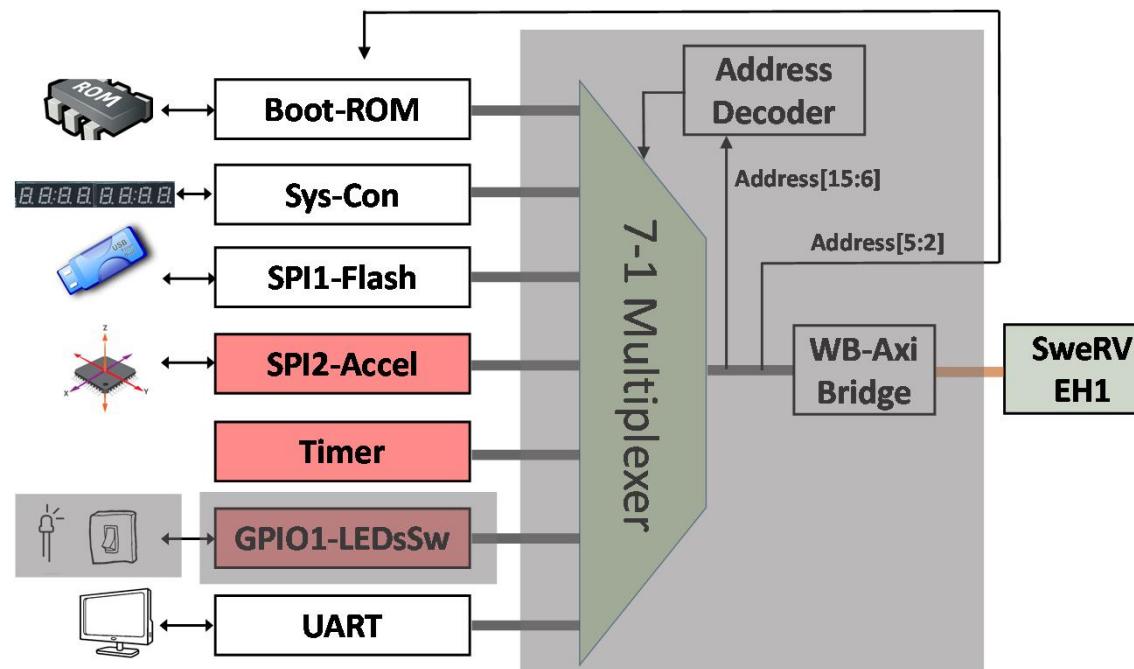
- 将[15:0]中的值写入地址0x80001404  
sw t3, 4(t0) # LEDs = [t3]<sub>15:0</sub>

读取开关:

- 从地址0x80001400读取bit [31:16]中的开关值
- 右移16位, 将开关值置于低16位中  
lw t5, 0(t0) # [t5]<sub>31:16</sub> = switch values  
srl li t5, t5, 16 # [t5]<sub>15:0</sub> = switch values

# RVfpga实验6：GPIO底层实现

- 主要分为3个部分
  - RVfpgaNexys与板上LED/开关的外部连接（左侧阴影区域）
  - SweRVolfX中集成的GPIO模块（中间阴影区域）
  - GPIO和SweRV EH1之间的连接（右侧阴影区域）



# RVfpga实验6：外部连接

文件**rvfganexys.xdc**： 定义*i\_sw[15:0]*与板上开关的连接以及*o\_led[15:0]*与板上LED的连接

```
26 set_property -dict { PACKAGE_PIN J15 IOSTANDARD LVCMS33 } [get_ports { i_sw[0] }]
27 set_property -dict { PACKAGE_PIN L16 IOSTANDARD LVCMS33 } [get_ports { i_sw[1] }]
28 set_property -dict { PACKAGE_PIN M13 IOSTANDARD LVCMS33 } [get_ports { i_sw[2] }]
29 set_property -dict { PACKAGE_PIN R15 IOSTANDARD LVCMS33 } [get_ports { i_sw[3] }]
30 set_property -dict { PACKAGE_PIN R17 IOSTANDARD LVCMS33 } [get_ports { i_sw[4] }]
31 set_property -dict { PACKAGE_PIN T18 IOSTANDARD LVCMS33 } [get_ports { i_sw[5] }]
32 set_property -dict { PACKAGE_PIN U18 IOSTANDARD LVCMS33 } [get_ports { i_sw[6] }]
33 set_property -dict { PACKAGE_PIN R13 IOSTANDARD LVCMS33 } [get_ports { i_sw[7] }]
34 set_property -dict { PACKAGE_PIN T8 IOSTANDARD LVCMS18 } [get_ports { i_sw[8] }]
35 set_property -dict { PACKAGE_PIN U8 IOSTANDARD LVCMS18 } [get_ports { i_sw[9] }]
36 set_property -dict { PACKAGE_PIN R16 IOSTANDARD LVCMS33 } [get_ports { i_sw[10] }]
37 set_property -dict { PACKAGE_PIN T13 IOSTANDARD LVCMS33 } [get_ports { i_sw[11] }]
38 set_property -dict { PACKAGE_PIN H6 IOSTANDARD LVCMS33 } [get_ports { i_sw[12] }]
39 set_property -dict { PACKAGE_PIN U12 IOSTANDARD LVCMS33 } [get_ports { i_sw[13] }]
40 set_property -dict { PACKAGE_PIN U11 IOSTANDARD LVCMS33 } [get_ports { i_sw[14] }]
41 set_property -dict { PACKAGE_PIN V10 IOSTANDARD LVCMS33 } [get_ports { i_sw[15] }]
42
43 set_property -dict { PACKAGE_PIN H17 IOSTANDARD LVCMS33 } [get_ports { o_led[0] }]
44 set_property -dict { PACKAGE_PIN K15 IOSTANDARD LVCMS33 } [get_ports { o_led[1] }]
45 set_property -dict { PACKAGE_PIN J13 IOSTANDARD LVCMS33 } [get_ports { o_led[2] }]
46 set_property -dict { PACKAGE_PIN N14 IOSTANDARD LVCMS33 } [get_ports { o_led[3] }]
47 set_property -dict { PACKAGE_PIN R18 IOSTANDARD LVCMS33 } [get_ports { o_led[4] }]
48 set_property -dict { PACKAGE_PIN V17 IOSTANDARD LVCMS33 } [get_ports { o_led[5] }]
49 set_property -dict { PACKAGE_PIN U17 IOSTANDARD LVCMS33 } [get_ports { o_led[6] }]
50 set_property -dict { PACKAGE_PIN U16 IOSTANDARD LVCMS33 } [get_ports { o_led[7] }]
51 set_property -dict { PACKAGE_PIN V16 IOSTANDARD LVCMS33 } [get_ports { o_led[8] }]
52 set_property -dict { PACKAGE_PIN T15 IOSTANDARD LVCMS33 } [get_ports { o_led[9] }]
53 set_property -dict { PACKAGE_PIN U14 IOSTANDARD LVCMS33 } [get_ports { o_led[10] }]
54 set_property -dict { PACKAGE_PIN T16 IOSTANDARD LVCMS33 } [get_ports { o_led[11] }]
55 set_property -dict { PACKAGE_PIN V15 IOSTANDARD LVCMS33 } [get_ports { o_led[12] }]
56 set_property -dict { PACKAGE_PIN V14 IOSTANDARD LVCMS33 } [get_ports { o_led[13] }]
57 set_property -dict { PACKAGE_PIN V12 IOSTANDARD LVCMS33 } [get_ports { o_led[14] }]
58 set_property -dict { PACKAGE_PIN V11 IOSTANDARD LVCMS33 } [get_ports { o_led[15] }]
```

# RVfpga实验6： RVfpga集成

文件swervolf\_core.v: 三态缓冲器和GPIO模块实例

```
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实验6：与SweRV EH1的连接

文件wb\_intercon.v: 7-1多路开关实现

```
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'hfffff000, 32'hfffffc0, 32'hfffffc0, 32'hfffffc0, 32'hfffffc0, 32'hfffffc0, 32'hfffff000}))
112 wb_mux io
113 (.wb_clk_i (wb_clk_i),
114 .wb_rst_i (wb_rst_i),
115 .wbm_adr_i (wbm_io_adr_i),
116 .wbm_dat_i (wbm_io_dat_i),
117 .wbm_sel_i (wbm_io_sel_i),
118 .wbm_we_i (wbm_io_we_i),
119 .wbm_cyc_i (wbm_io_cyc_i),
120 .wbm_stb_i (wbm_io_stb_i),
121 .wbm_cti_i (wbm_io_cti_i),
122 .wbm_bte_i (wbm_io_bte_i),
123 .wbs_dat_o (wbm_dat_o),
124 .wbs_ack_o (wbm_ack_o),
125 .wbs_err_o (wbm_err_o),
126 .wbs_rty_o (wbm_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

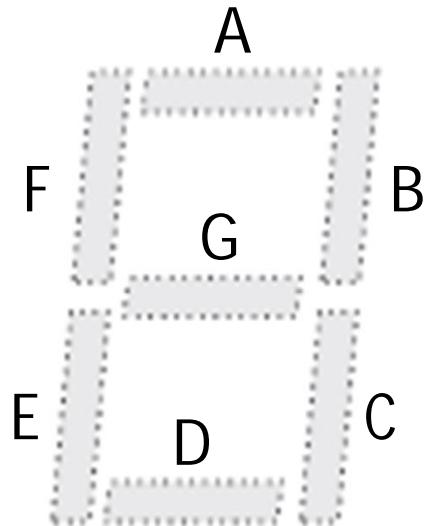
Peripheral Signals

# 实验7： 7段显示屏

# RVfpga实验7：7段显示屏

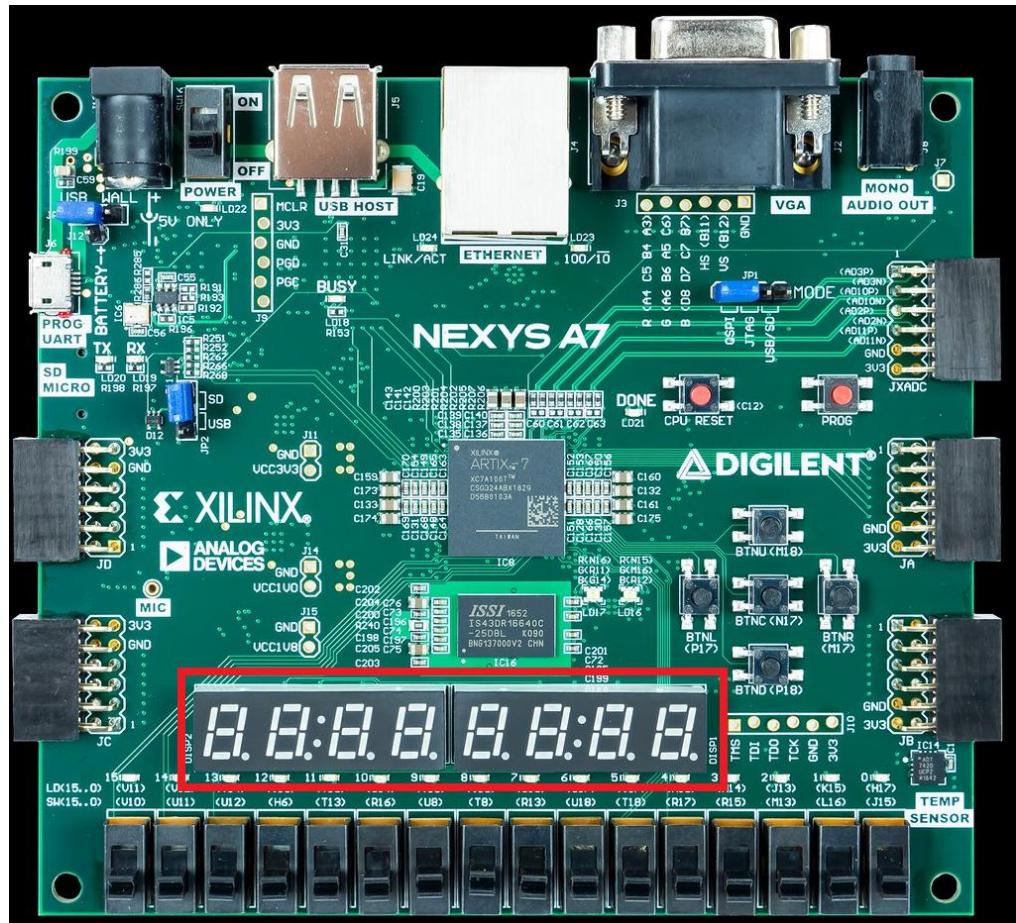
- 7段显示屏概述
- 7段显示屏硬件

# RVfpga实验7：7段显示屏概述



- 7个LED段：A-G
- 点亮相应段可形成特定数字
  - 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
  - 以此类推

# RVfpga实验7：Nexys A7上的7段显示屏



开发板图片来源：<https://reference.digilentinc.com/>

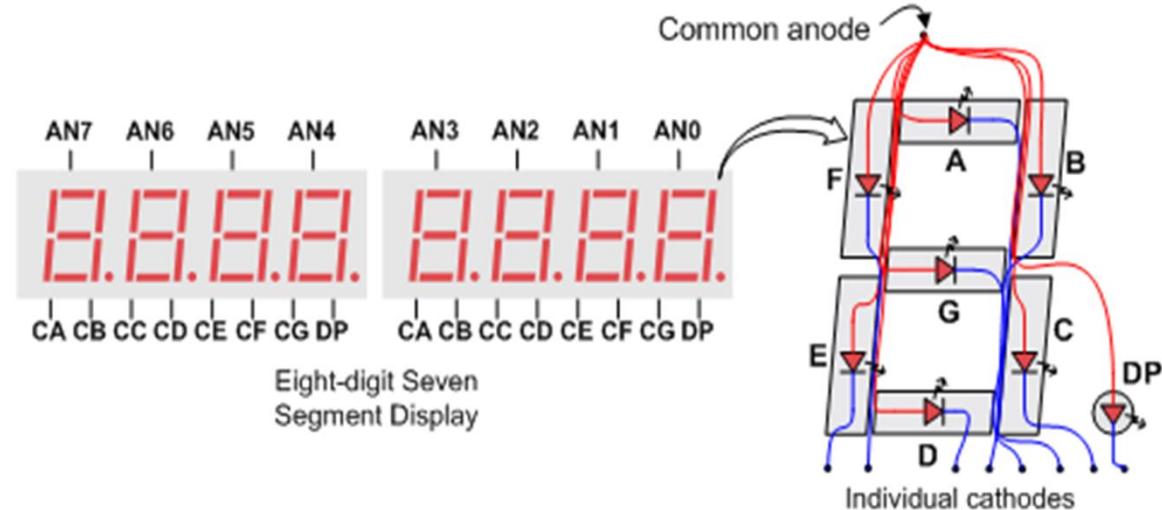
- 8位7段显示屏
- 存储器映射访问：
  - **Enables\_Reg:** 0x80001038
  - **Digits\_Reg:** 0x8000103C
- 使能信号低电平有效
- 示例：最右边的两位显示71：
  - **Enables\_Reg** = 0xFC (0b11111100: 使能最右边的两位)
  - **Digits\_Reg** = 0x71
  - 汇编语言：

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

# RVfpga实验7：7段显示屏硬件

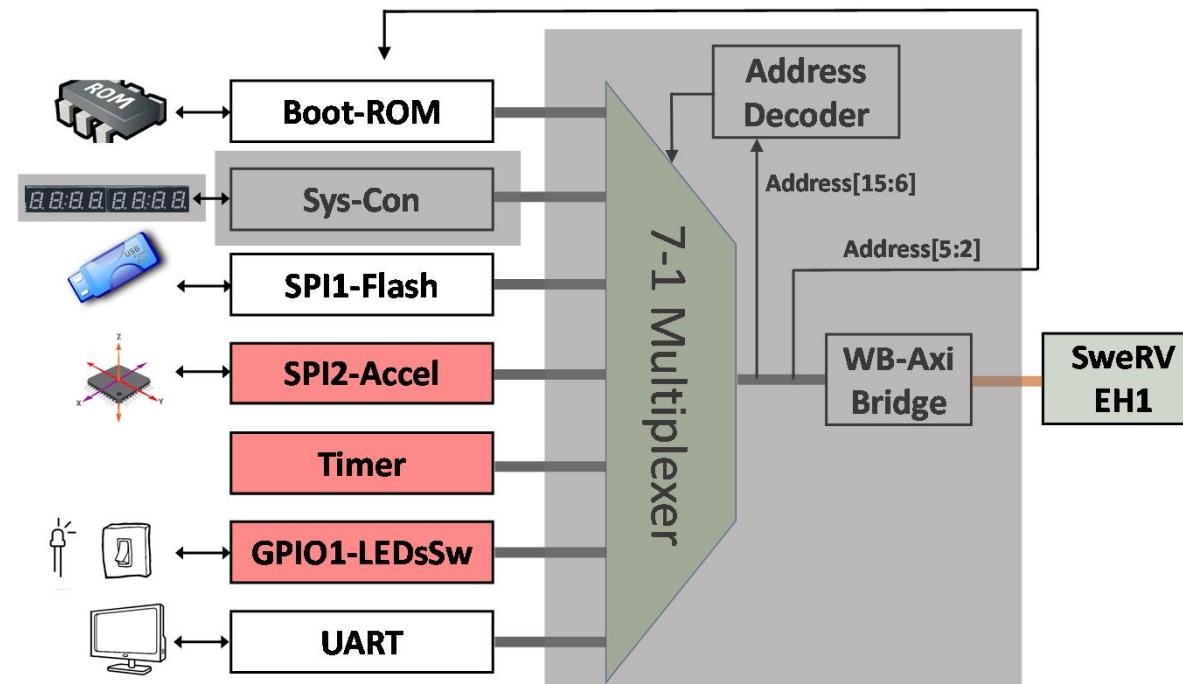
- 每一位都是共阳极（每一位所有LED的阳极连接在一起）
  - 阳极信号充当使能信号（AN0 - AN7）
  - 驱动为低电平以使能相应位（AN0-AN7经逆变器（未显示）馈入LED）
- 所有位的各个段均连接在一起
  - 各个段驱动为低电平时即可点亮
  - 凭借AN0 - AN7信号的时分复用，可以在每一位上显示惟一值
  - 一个位的AN信号（AN0 - AN7）必须每**1-16 ms**变为低电平才能点亮

8位7段显示屏



# RVfpga实验7：7段显示屏底层实现

- 主要分为3个部分
  - RVfpgaNexys与板上7段显示屏的外部连接（左侧阴影区域）
  - SweRVolfX中集成的7段显示屏模块（中间阴影区域）
  - 7段显示屏和SweRV EH1之间的连接（右侧阴影区域）



# RVfpga实验7：外部连接

文件**rvfganexys.xdc**：定义使用板上7段显示屏时CA-CG（在SoC中称为Digits\_Bits[i]）和AN[i]的连接

```
60  ##7 segment display
61  set_property -dict { PACKAGE_PIN T10    IOSTANDARD LVCMS33 } [get_ports { CA }]; #IO_L24N_T3_A00_D16_14 Sch=ca
62  set_property -dict { PACKAGE_PIN R10    IOSTANDARD LVCMS33 } [get_ports { CB }]; #IO_25_14 Sch=cb
63  set_property -dict { PACKAGE_PIN K16    IOSTANDARD LVCMS33 } [get_ports { CC }]; #IO_25_15 Sch=cc
64  set_property -dict { PACKAGE_PIN K13    IOSTANDARD LVCMS33 } [get_ports { CD }]; #IO_L17P_T2_A26_15 Sch=cd
65  set_property -dict { PACKAGE_PIN P15    IOSTANDARD LVCMS33 } [get_ports { CE }]; #IO_L13P_T2_MRCC_14 Sch=ce
66  set_property -dict { PACKAGE_PIN T11    IOSTANDARD LVCMS33 } [get_ports { CF }]; #IO_L19P_T3_A10_D26_14 Sch=cf
67  set_property -dict { PACKAGE_PIN L18    IOSTANDARD LVCMS33 } [get_ports { CG }]; #IO_L4P_T0_D04_14 Sch=cg
68  #set_property -dict { PACKAGE_PIN H15    IOSTANDARD LVCMS33 } [get_ports { DP }]; #IO_L19N_T3_A21_VREF_15 Sch=dp
69  set_property -dict { PACKAGE_PIN J17    IOSTANDARD LVCMS33 } [get_ports { AN[0] }]; #IO_L23P_T3_FOE_B_15 Sch=an[0]
70  set_property -dict { PACKAGE_PIN J18    IOSTANDARD LVCMS33 } [get_ports { AN[1] }]; #IO_L23N_T3_FWE_B_15 Sch=an[1]
71  set_property -dict { PACKAGE_PIN T9     IOSTANDARD LVCMS33 } [get_ports { AN[2] }]; #IO_L24P_T3_A01_D17_14 Sch=an[2]
72  set_property -dict { PACKAGE_PIN J14    IOSTANDARD LVCMS33 } [get_ports { AN[3] }]; #IO_L19P_T3_A22_15 Sch=an[3]
73  set_property -dict { PACKAGE_PIN P14    IOSTANDARD LVCMS33 } [get_ports { AN[4] }]; #IO_L8N_T1_D12_14 Sch=an[4]
74  set_property -dict { PACKAGE_PIN T14    IOSTANDARD LVCMS33 } [get_ports { AN[5] }]; #IO_L14P_T2_SRCC_14 Sch=an[5]
75  set_property -dict { PACKAGE_PIN K2     IOSTANDARD LVCMS33 } [get_ports { AN[6] }]; #IO_L23P_T3_35 Sch=an[6]
76  set_property -dict { PACKAGE_PIN U13    IOSTANDARD LVCMS33 } [get_ports { AN[7] }]; #IO_L23N_T3_A02_D18_14 Sch=an[7]
```

# RVfpga实验7：SweRVolfX集成

- 文件**swervolf\_syscon.v**: 7段显示屏控制器实例。除了时钟信号（**i\_clk**）和复位信号（**i\_rst**）外，该模块还接收两个输入信号（**Enables\_Reg**和**Digits\_Reg**），即前文所述的两个寄存器映射控制寄存器。该模块输出两个信号（**AN**和**Digits\_Bits**），这两个信号连接到板上7段显示屏。

```
// 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实验7：SweRVolfX集成

- 该文件中还实现了**SevSegDisplays\_Controller**，其中包含以下子单元：
  - 两个多路开关（模块**SevSegMux**），用于选择每2 ms发送到AN和**Digits\_Bits**信号的值。
  - 计数器（模块**counter**），用于产生2 ms周期。
  - 解码器（模块**SevenSegDecoder**），用于输出给定4位十六进制值的段值。

# 实验8： 定时器

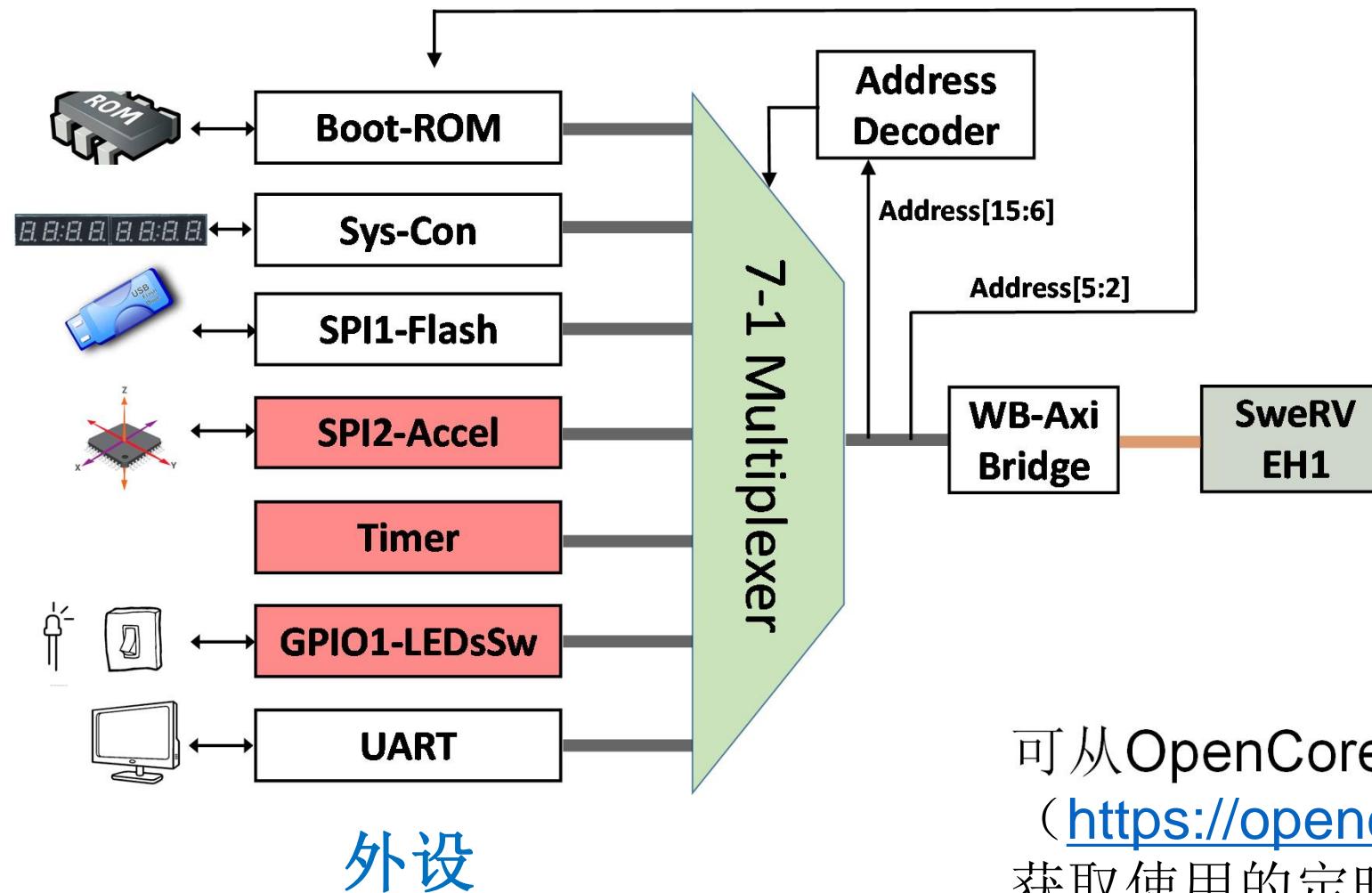
# RVfpga实验8：定时器

- 定时器概述
- 定时器寄存器
- 定时器示例

# RVfpga实验8：定时器

- 定时器以固定的频率递增或递减计数器
- 常见于单片机和SoC中
- 用于生成精确时序

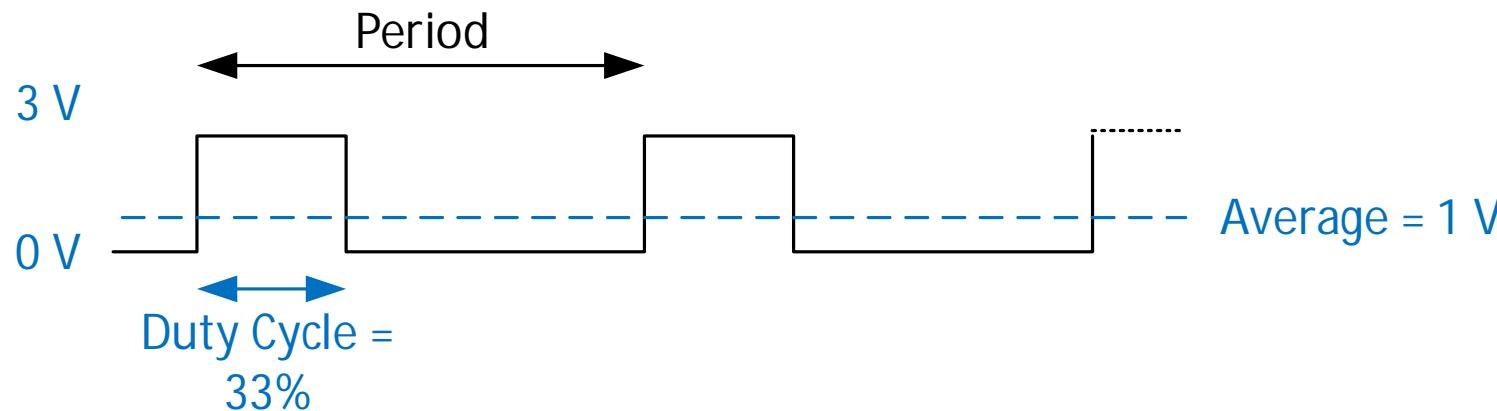
# RVfpga实验8：定时器



可从OpenCores  
(<https://opencores.org/projects/ptc>)  
获取使用的定时器模块。

# RVfpga实验8：定时器（PTC）模块

- 定时器模块（也称为**PTC**模块）用于：
  - 定时器/计数器：对时钟边沿（或另一个信号的边沿，也称为事件）进行计数
  - 脉宽调制（Pulse-Width Modulation, PWM）：
    - 输出变为高电平的持续时间（称为占空比）
    - 用于以数字方式近似计算模拟电压
- **PWM示例：**33%占空比（信号在1/3的时间内为高电平）如果高电平为3 V，则模拟电压（信号的平均电压）为 $3 \text{ V} * 0.33 = 1 \text{ V}$



# RVfpga实验8：定时器（PTC）寄存器

名称	地址	宽度	访问	说明
RPTC_CNTR	0x80001200	1-32	R/W	主PTC计数器
RPTC_HRC	0x80001204	1-32	R/W	PTC高电平参考/捕捉寄存器
RPTC_LRC	0x80001208	1-32	R/W	PTC低电平参考/捕捉寄存器
RPTC_CTRL	0x8000120C	9	R/W	控制寄存器

- **RPTC\_CNTR:** 计数器（计数器的值）
- **RPTC\_HRC:** 高电平参考捕捉 - 指示PWM模式下输出应变为高电平的周期数（复位后）
- **RPTC\_LRC:** 低电平参考捕捉 - 指示计数器/定时器模式下计数完成时的周期数（复位后）；指示PWM模式下输出应变为低电平的时钟周期数（复位后）。
- **RPTC\_CTRL:** 控制寄存器

# RVfpga实验8：定时器（PTC）控制寄存器

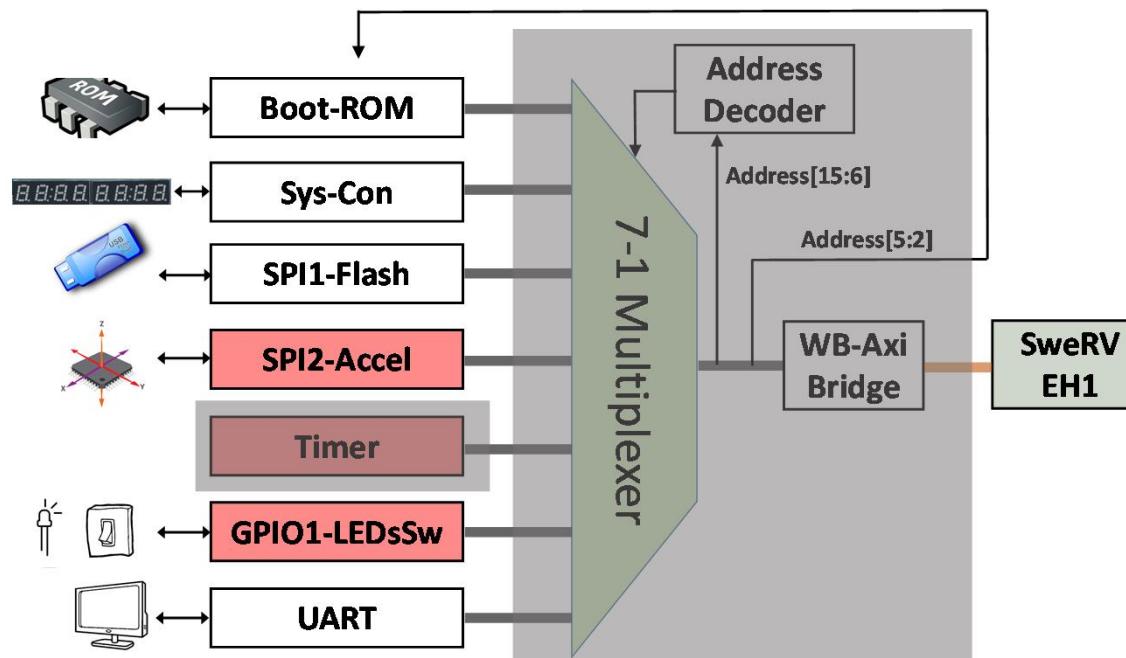
位	访问	复位	名称和说明
0	R/W	0	EN: 置1时, RPTC_CNTR递增。
1	R/W	0	ECLK: 选择时钟信号: 外部时钟(通过ptc_ecgt) (1) 或系统时钟 (0)。
2	R/W	0	NEC: 用于选择外部时钟(ptc_ecgt) 的下降沿/上升沿和低电平/高电平周期。
3	R/W	0	OE: 使能PWM输出驱动器。
4	R/W	0	SINGLE: 置1时, RPTC_CNTR在达到RPTC_LRC值后不递增。清零时, RPTC_CNTR在达到RPTC_LCR寄存器中的值后重新启动。
5	R/W	0	INTE: 置1时, 当RPTC_CNTR值等于RPTC_LRC或RPTC_HRC的值时, PTC会将中断置为有效。信号清零时, 中断将被屏蔽。
6	R/W	0	INT: 读取时, 该位表示待处理的中断。置1时, 表示有一个中断待处理。当该位写入1时, 中断请求将被清除。
7	R/W	0	CNTRRST: 置1时, 将复位RPTC_CNTR。清零时, 计数器将正常工作。
8	R/W	0	CAPTE: 置1时, RPTC_CNTR将被捕捉到RPTC_LRC或RPTC_HRC寄存器中。清零时, 捕捉功能将被屏蔽。

# RVfpga实验8：定时器示例

- 将**RPTC\_LRC**设置为要计数的周期数
- 设置控制位（**RPTC\_CTRL**）以配置定时器：
  - 复位计数器并清除中断： **RPTC\_CTRL = 0xC0** (0b011000000) : CNTRRST (bit 7) = 1: 复位计数器 (**RPTC\_CNTR = 0**) ; INT (bit 6) = 1: 清除中断请求。
  - 使能计数器并允许中断： **RPTC\_CTRL = 0x21** (0b000100001) : EN (bit 0) = 1: 使能计数器，从而使**RPTC\_CNTR**递增; INTE (bit 5) = 1: 当**RPTC\_CNTR == RPTC\_LRC**时，PTC会将中断置为有效。
- 程序读取控制寄存器中的中断位（**INT**是**RPTC\_CTRL**的**bit 6**），直到其为1（表示**RPTC\_CNTR == RPTC\_LRC**）。
- 该算法不使用中断，但它会读取中断位（**INT**, **RPTC\_CTRL**的bit 6）以确定何时达到正确的时钟周期数。我们将在实验9中展示如何使用中断。

# RVfpga实验8：定时器底层实现

- 主要分为2个部分
  - (无外部连接)
  - SweRVolfX中集成的定时器模块 (左侧阴影区域)
  - 定时器和SweRV EH1之间的连接 (右侧阴影区域)



# RVfpga实验8：SweRVolfX集成

文件**swervolf\_core.v**: PTC模块实例

```
// 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_addr[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 ()
);
```

# 实验9： 中断驱动I/O

# RVfpga实验9：中断驱动I/O

- 中断驱动I/O与编程I/O
- RVfpga系统的中断控制器
- 如何使用Western Digital的外设支持包和开发板支持包（PSP和BSP）配置中断
- 中断示例

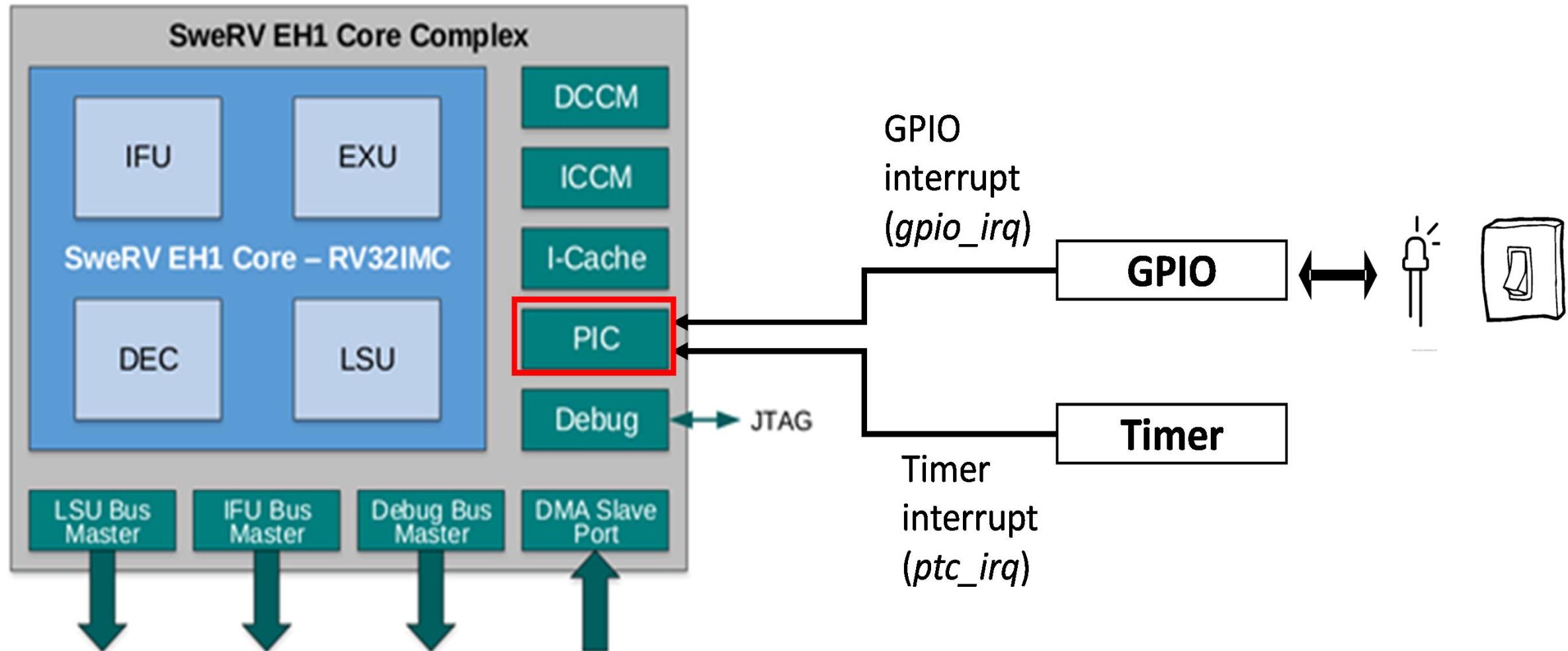
# RVfpga实验9：中断驱动I/O简介

- 编程I/O：
  - 程序会连续轮询一个值（例如开关），直到获得所需值为止。
  - 举例来说，该方法曾在之前的实验中用于读取开关。
  - 该方法会占用处理器，不断访问一个值（而不是能够执行其他有用的工作）。
- 中断驱动I/O：
  - 事件（例如，引脚置为有效）使处理器跳转到中断服务程序（ISR，也称为中断处理程序），这类似于未调度的函数调用。ISR处理中断（例如，读取开关的值），然后返回到常规程序。
  - 在该事件发生之前，处理器可以继续执行有用的工作。

# RVfpga实验9：处理中断

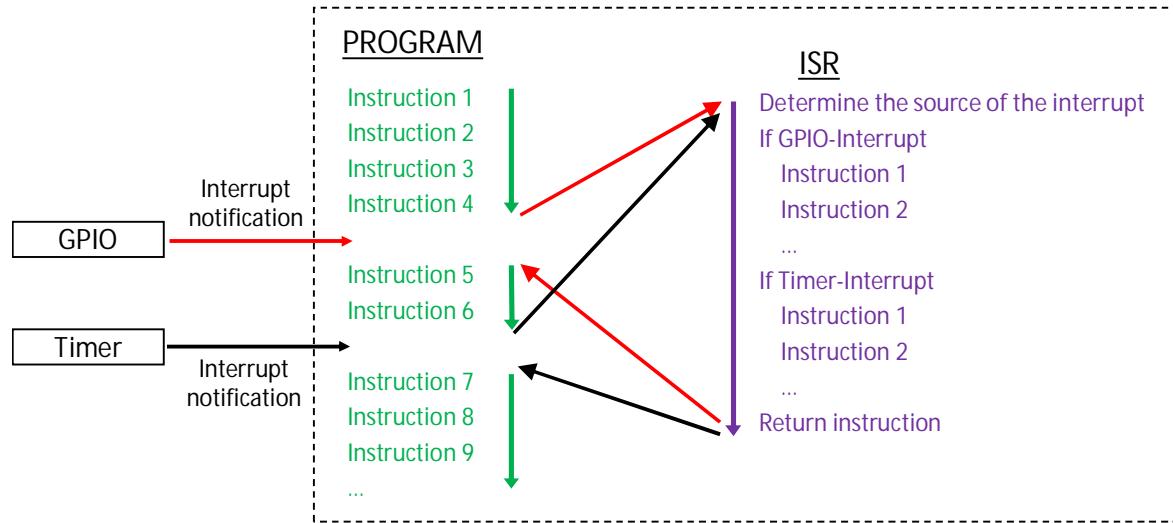
- 中断可能是由硬件或软件引起的
- 在本实验中，我们重点关注的是硬件中断
- SweRV EH1内核按照RISC-V的PLIC（平台级中断控制器）规范处理中断。该内核被称为可编程中断控制器（PIC）。  
它具有：
  - 255个中断源
  - 15个优先级

# RVfpga实验9：中断硬件

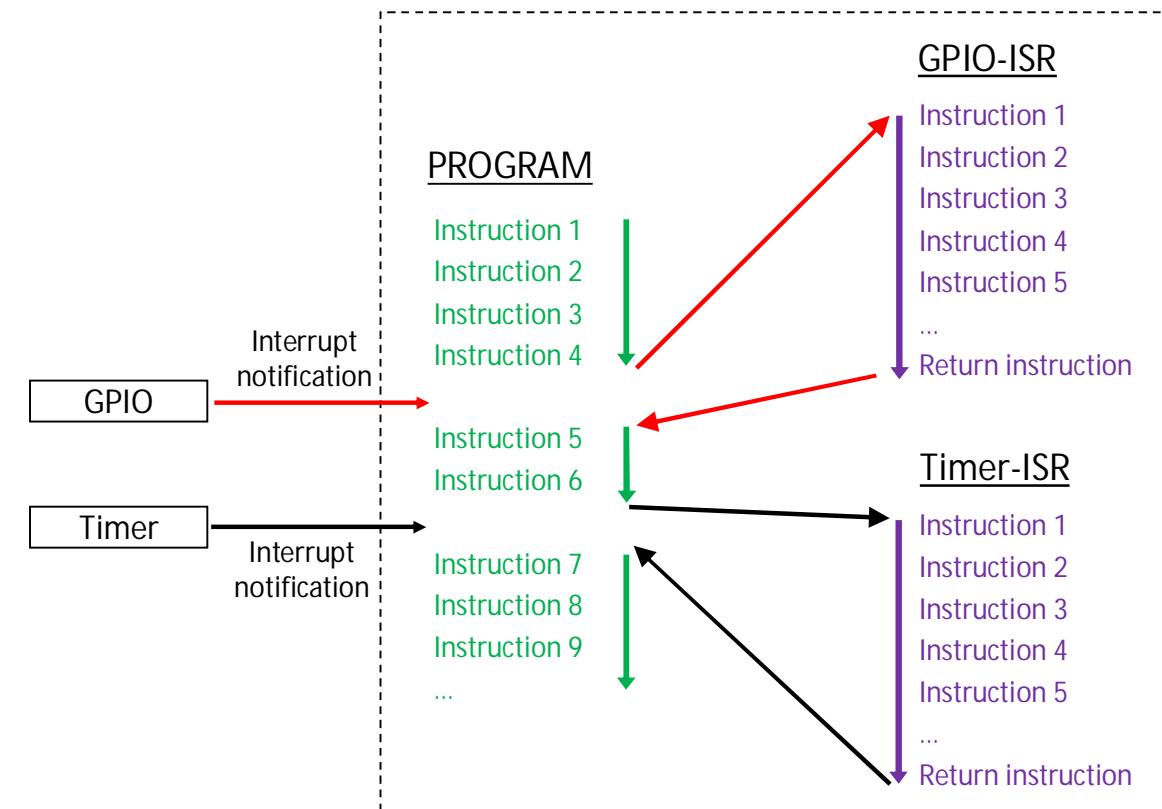


# RVfpga实验9：单向量模式与多向量模式

单向量模式示例：



多向量模式示例：



# RVfpga实验9：处理中断

- 使用WD的PSP/BSP：
  - 使用WD的PSP/BSP初始化中断
  - 初始化255个中断中的一个或多个并提供ISR的名称
  - 将应触发中断的外设信号与中断引脚相连。
  - 允许所有中断
  - 允许外部中断

# RVfpga实验9：中断示例

- 使用中断读取Switch[0]的值 - 仅在上升沿（0跳变为→1）

名称	地址	宽度	访问	说明
GPIO_IN	0x80001400	1-32	R	GPIO输入数据
GPIO_OUT	0x80001404	1-32	R/W	GPIO输出数据
GPIO_OE	0x80001408	1-32	R/W	GPIO输出驱动器使能
GPIO_INTE	0x8000140C	1-32	R/W	中断允许
GPIO_PTRIG	0x80001410	1-32	R/W	触发中断的事件类型
GPIO_AUX	0x80001414	1-32	R/W	将辅助输入与GPIO输出复用
GPIO_CTRL	0x80001418	2	R/W	控制寄存器
GPIO_INTS	0x8000141C	1-32	R/W	中断状态
GPIO_ECLK	0x80001420	1-32	R/W	使能gpio_eclk以锁存GPIO_IN
GPIO_NECK	0x80001424	1-32	R/W	选择gpio_eclk的有效边沿

有关完整代码，请参见： [RVfpgaPath]/RVfpga/Labs/Lab9/LED-Switch\_7SegDispl Interrupts\_C-Lang.c

# RVfpga实验9：中断示例

- 设置用于中断的GPIO寄存器：
  - GPIO\_INTE = 0x10000 (允许Switch[0]的中断)
  - GPIO\_PTRIG = 0x10000 (在Switch[0]的上升沿触发中断)
  - GPIO\_INTS = 0x0 (清除所有中断)
  - GPIO\_CTRL = 0x1 (允许GPIO中断)

# RVfpga实验9：中断示例

- GPIO ISR:

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

    /* Invert LED value */
    i = M_PSP_READ_REGISTER_32(GPIO_LEDs);      /* RGPI0_OUT */
    i = !i;                                       /* Invert the LEDs */
    i = i & 0x1;                                   /* Only keep right-most LED */
    M_PSP_WRITE_REGISTER_32(GPIO_LEDs, i)          /* RGPI0_OUT */

    /* Clear GPIO interrupt */
    M_PSP_WRITE_REGISTER_32(RGPIO_INTS, 0x0); /* RGPI0_INTS */

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

# RVfpga实验9：中断示例

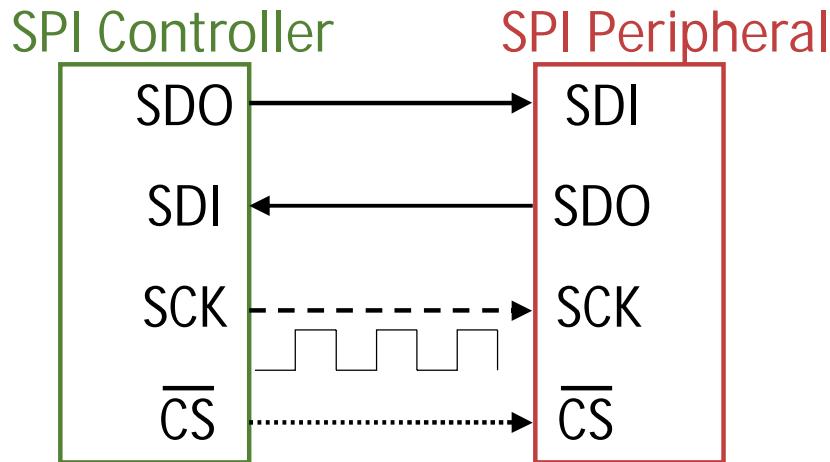
- 将中断4 (IRQ4) 与开关的中断相连，并将中断服务程序设置为 **GPIO\_ISR**
- 存储器映射寄存器 $0x80001018 = 0x1$ : 将GPIO中断与IRQ4相连
- 允许全局中断

# 实验10： 串行总线

# RVfpga实验10：串行总线

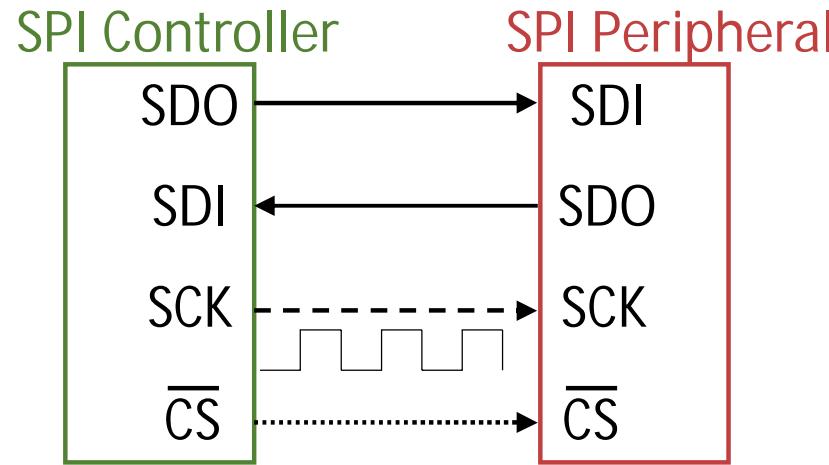
- 串行总线一次发送一位
  - 相比之下，并行总线则一次发送多位
- 通用串行总线
  - **UART**（通用异步收发器）
  - **SPI**（串行外设接口）
  - **I2C**（集成电路间协议）
- 在本实验中，我们重点关注的是**SPI**

# RVfpga实验10：串行总线

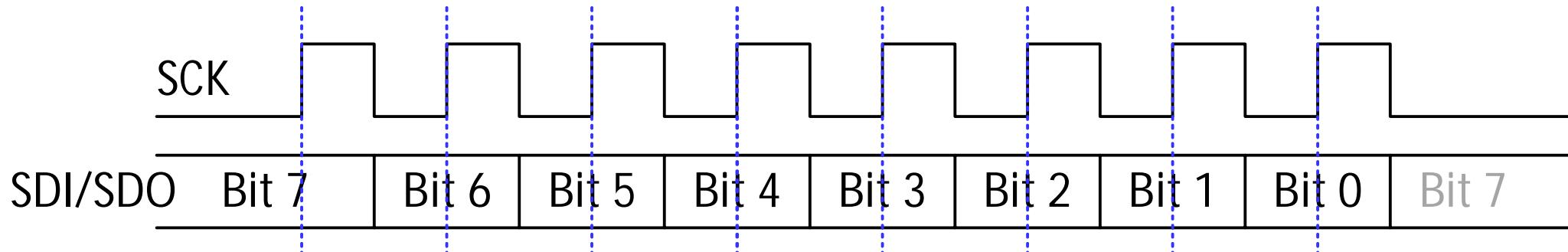


- **控制器:** 发送时钟，发送和接收数据
- **外设:** 接收时钟，发送和接收数据
- **信号:**
  - **SDO:** 串行数据输出
  - **SDI:** 串行数据输入
  - **SCK:** SPI时钟
  - **CSbar:** 低电平有效片选

# RVfpga实验10：串行总线



- **SCK空闲**
- 当控制器发送**SCK**边沿时，控制器和外设将采样并发送数据。数据在下降沿更改（发送），在上升沿采样（但这是可配置的）



# RVfpga实验10：RVfpga系统的SPI模块

- RVfpga系统的SPI模块来自OpenCores

[https://opencores.org/projects/simple\\_spi](https://opencores.org/projects/simple_spi)

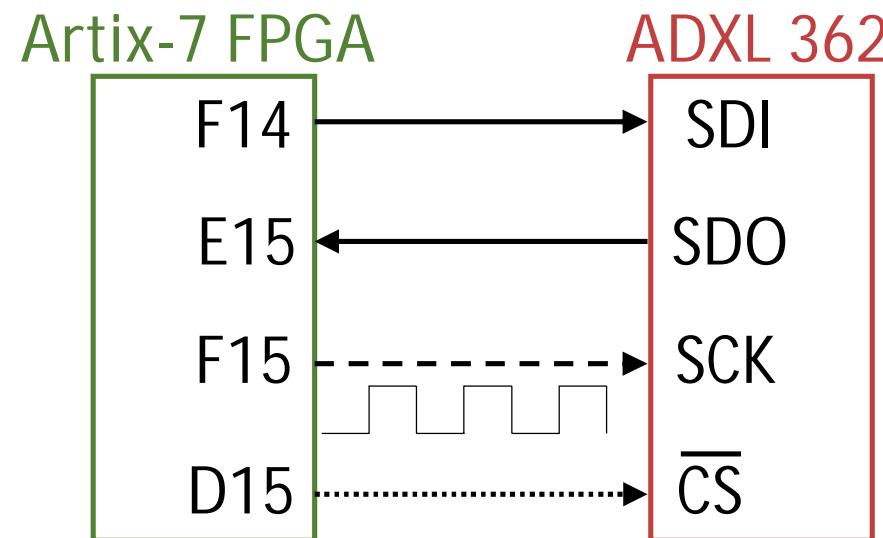
- 4条目读写缓冲区
- SPI寄存器：

名称	地址	宽度	访问	说明
SPCR	0x80001100	8	R/W	控制寄存器
SPSR	0x80001108	8	R/W	状态寄存器
SPDR	0x80001110	8	R/W	数据寄存器
SPER	0x80001118	8	R/W	扩展寄存器
SPCS	0x80001120	8	R/W	CS寄存器

# RVfpga实验10：ADXL362加速计

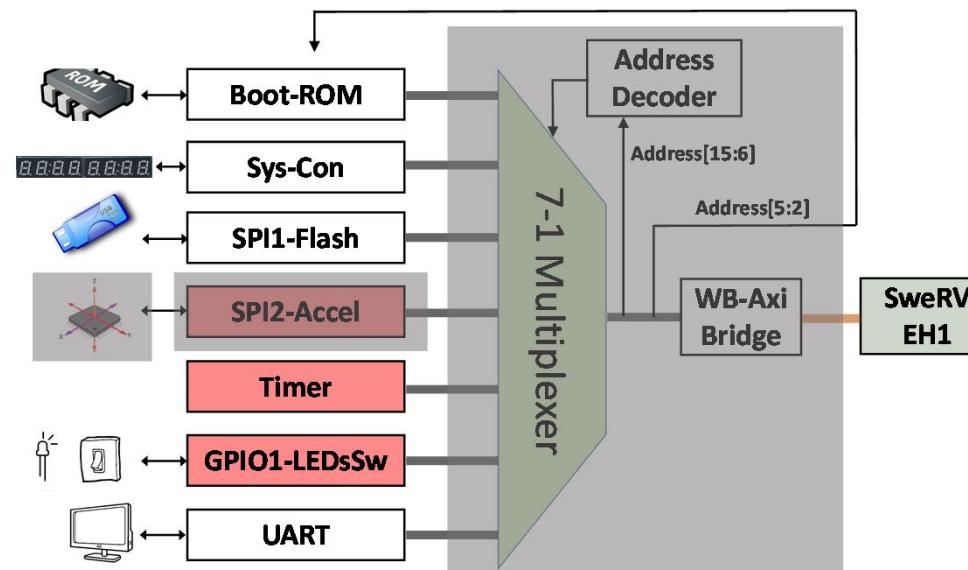
- Nexys A7开发板包括一个模拟器件ADXL362加速计。有关完整的信息，请访问：

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



# RVfpga实验10：加速计底层实现

- 主要分为3个部分
  - RVfpgaNexys与板上加速计的外部连接（左侧阴影区域）
  - SweRVolfX中集成的全新SPI模块（中间阴影区域）
  - 加速计和SweRV EH1之间的连接（右侧阴影区域）



# RVfpga实验10：外部连接

文件**rvfganexys.xdc**： 定义SoC中使用的SPI信号与相应板上加速计引脚的连接

```
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实验10: SweRVolfX集成

文件**swervolf\_core.v**: 三态缓冲器和GPIO模块实例

```
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));
```

# 实验11-20

## 了解内核和 存储器系统

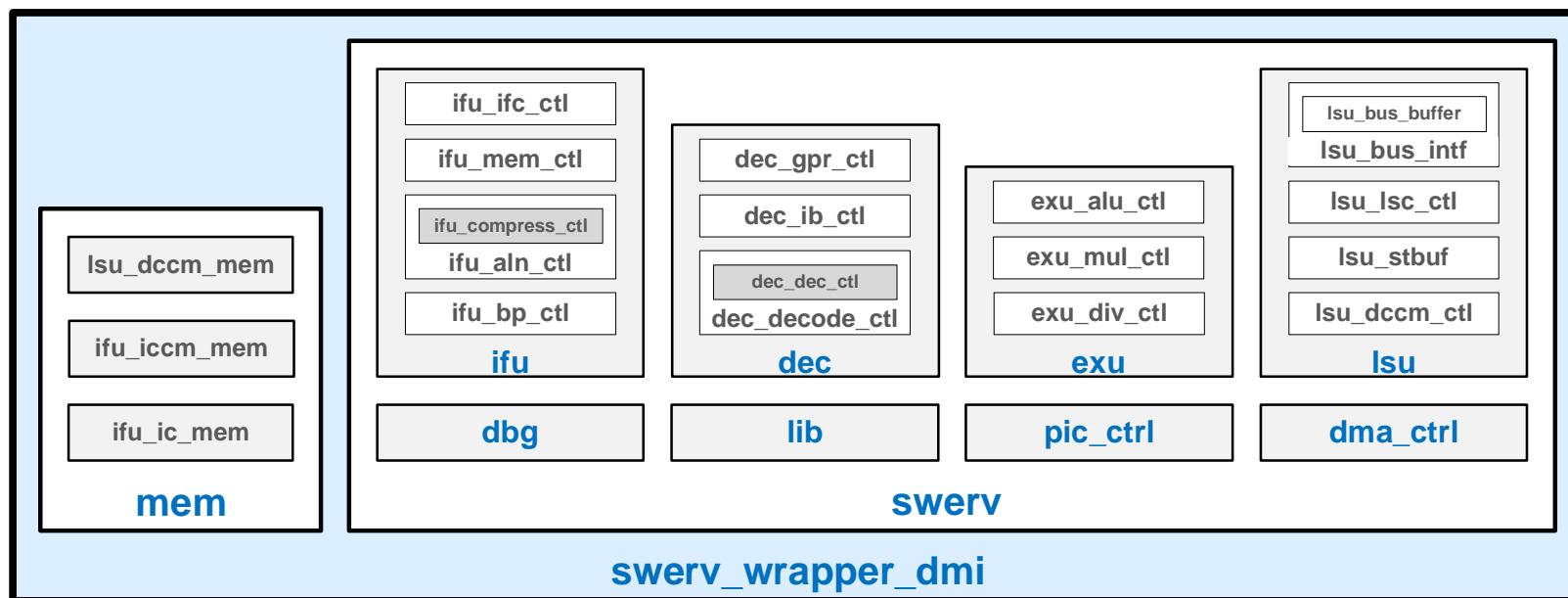
# RVfpga实验11-20概述

- 实验11-20将深入到微架构级别并分析SweRV EH1处理器以及高速缓存/存储器层级的工作方式。
  - 每个实验分为两部分：
    - 概念的理论说明
    - 概念的图示说明（使用图和示例程序的Verilator仿真来说明概念）。
- 我们还提供相关练习以加深对所描述概念的理解和体验。

# 实验11： SweRV EH1配置、 结构和性能监视

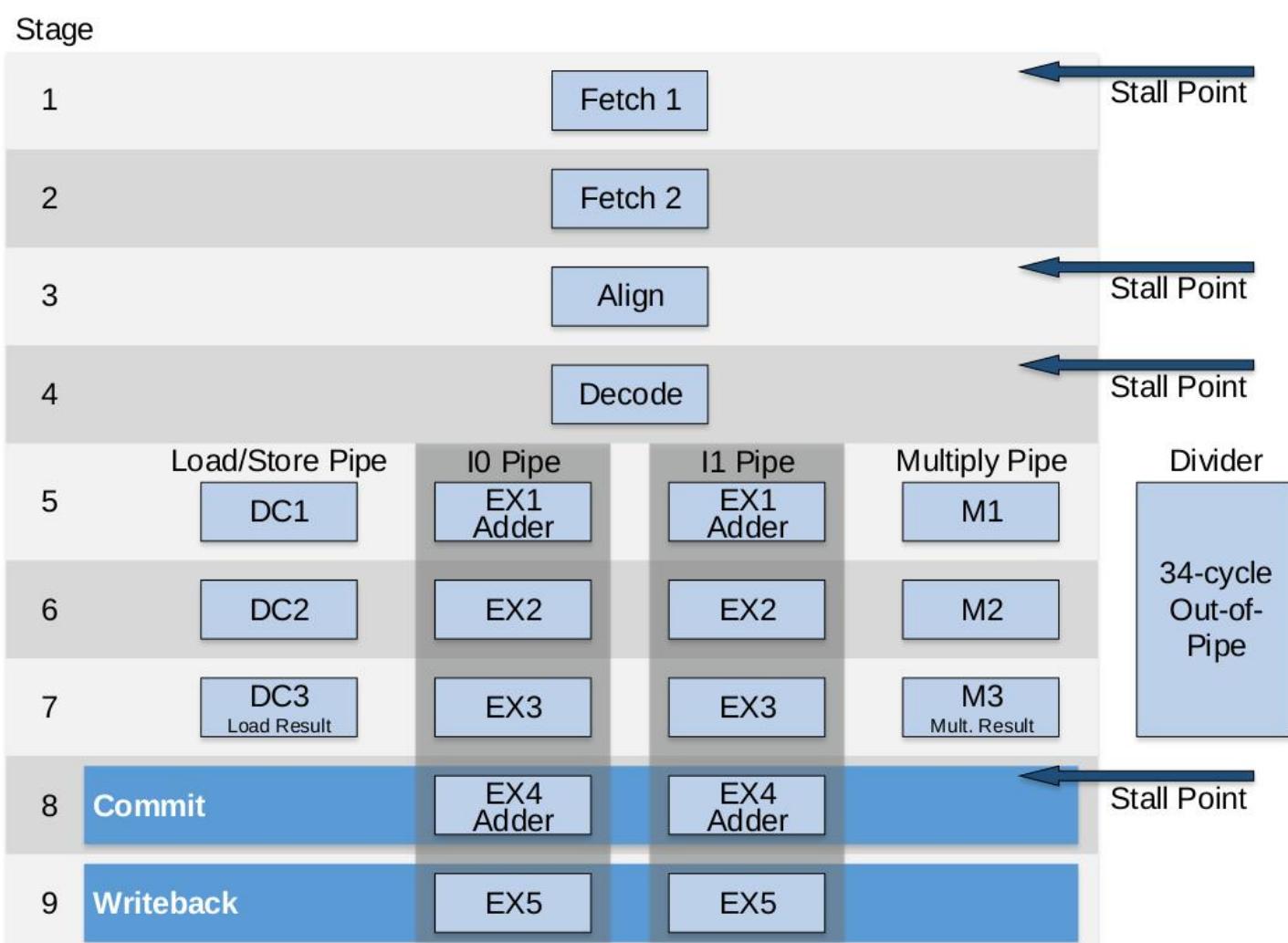
# RVfpga实验11：SweRV EH1 Verilog模块的层级

- 下图给出了主要Verilog模块的层级。
- mem**模块对构成SweRV EH1处理器存储器层级的结构进行实例化：ICCM、DCCM和I\$。
- swerv**模块为整体CPU；其对构成SweRV EH1处理器的模块进行实例化。



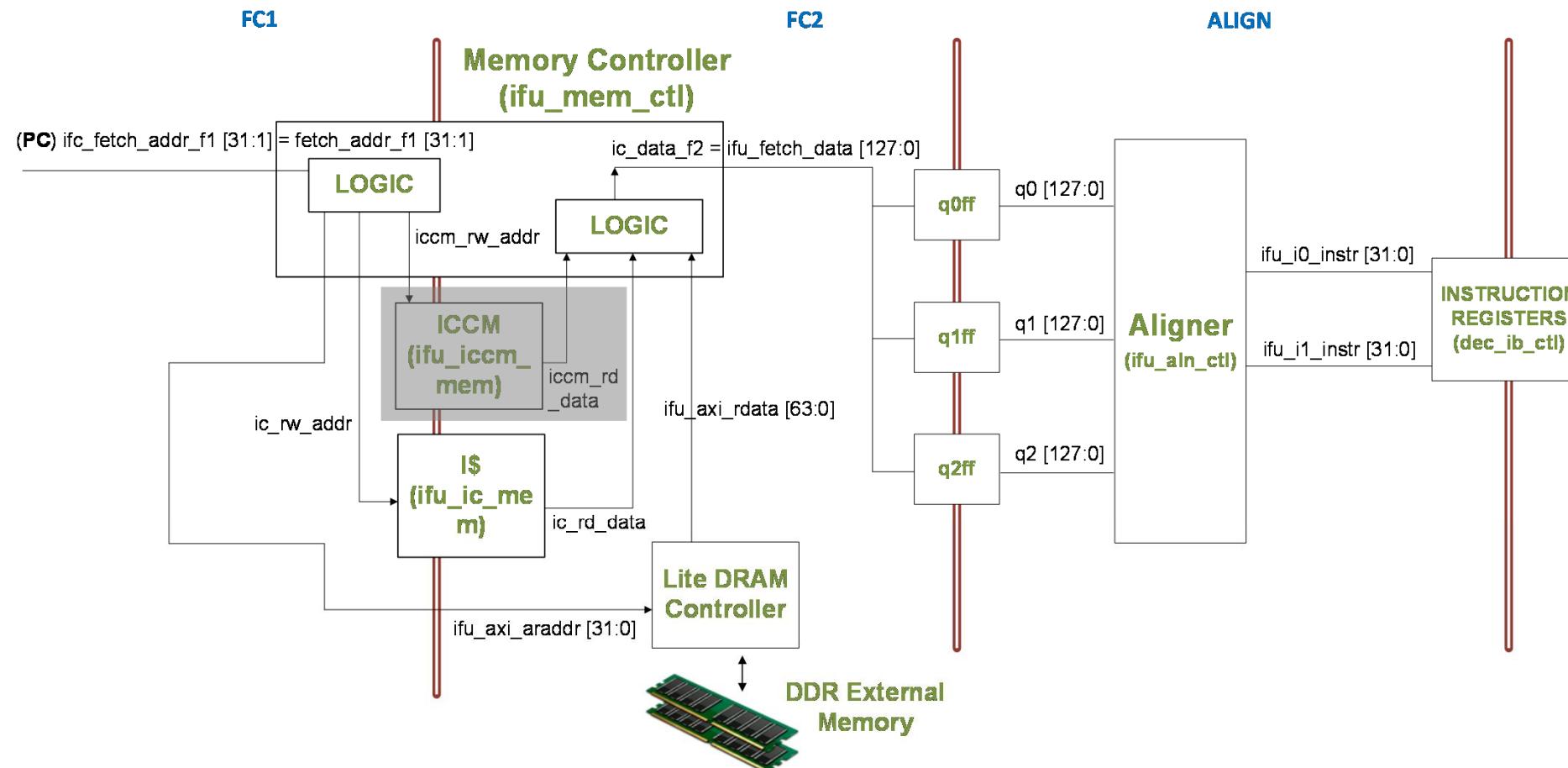
# RVfpga实验11：SweRV EH1流水线

- 正如RVfpga入门指南（Getting Started Guide, GSG）中所述，SweRV EH1是32位2路超标量9级流水线顺序处理器。
- 右图给出了处理器微架构的高级视图，实验11-20将对此进行详细分析。



# RVfpga实验11：取指（FC1和FC2）和对齐阶段

- SweRV EH1流水线的前三个阶段是：两个取指阶段（FC1和FC2）和一个对齐阶段



# RVfpga实验11：取指（FC1和FC2）阶段

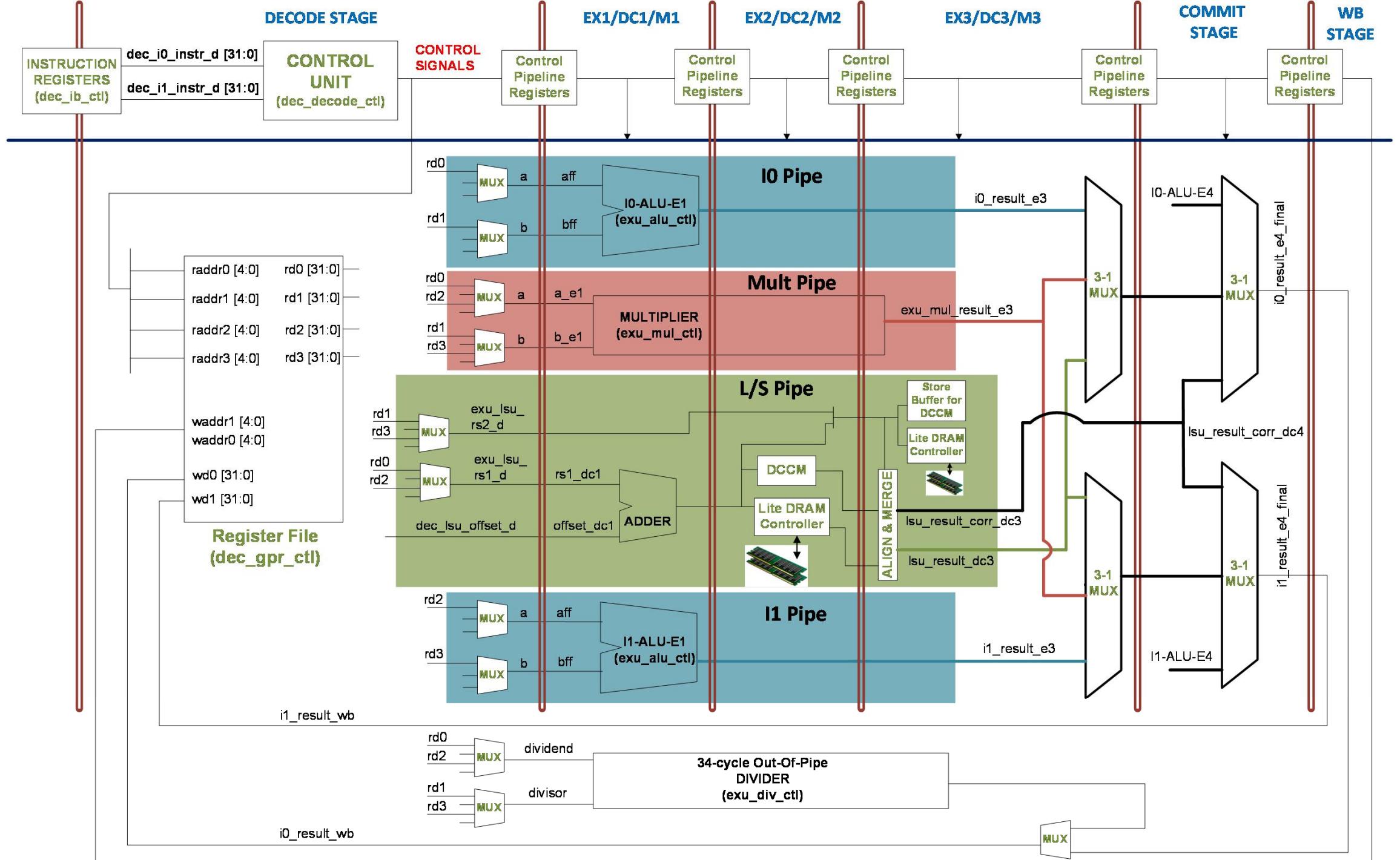
- 取指阶段：从指令存储器读取指令
- 在RVfpga中，指令存储器包括：
  - 512 KiB ICCM（在默认RVfpga配置中禁止）
  - 16 KiB指令高速缓存
  - 128 MiB DDR外部存储器
- **FC1**：计算指令地址（`ifc_fetch_addr_f1`）
- **FC2**：从I\$、DDR外部存储器或ICCM读取指令。（I\$仅缓存主存储器地址范围内的存储器。）

# RVfpga实验11：对齐阶段

- 对齐阶段执行两个主要任务：
  - 每个周期向译码阶段提供两条**32位指令**：每个周期从指令存储器提供的128位指令束中提取两条指令，并将它们分配给SweRV EH1中的每个通路（共两个通路）。
  - 解压指令：对齐阶段将16位指令解压为32位指令。

# RVfpga实验11：译码、EX1/2/3、提交和WB阶段

- 下一张幻灯片上的图显示了流水线的最后六个阶段：译码阶段、三个执行阶段、提交阶段和回写（Writeback, WB）阶段。



# RVfpga实验11：译码阶段

- 译码阶段执行两个主要任务：
  - 对指令进行译码并生成控制信号（由控制单元执行）
  - 将指令和操作数分发到适当的管道：
    - 管道：
      - 两个整数管道：I0和I1
      - 乘法管道
      - 加载/存储管道（L/S）
    - 管外34周期除法器
    - 几个多路开关在可能的操作数中进行选择

# RVfpga实验11：执行阶段 – 3个管道和一个除法器

- **I0/I1管道**: 两个整数管道具有三个阶段（EX1、EX2和EX3）。EX1执行ALU运算。
- **乘法管道**: 乘法管道包含一个使用三个阶段（M1、M2和M3）的3周期整数乘法器。
- **加载/存储（L/S）管道**: L/S管道执行加载和存储指令。
- **除法器**: 除法器是一个非流水线式34周期整数除法器。

在第三个执行阶段（EX3/DC3/M3）结束时，使用两个3:1多路开关（每路一个）从正确的管道（I0/I1、MUL或L/S）中选择指令的结果。

# RVfpga实验11：提交和写回阶段

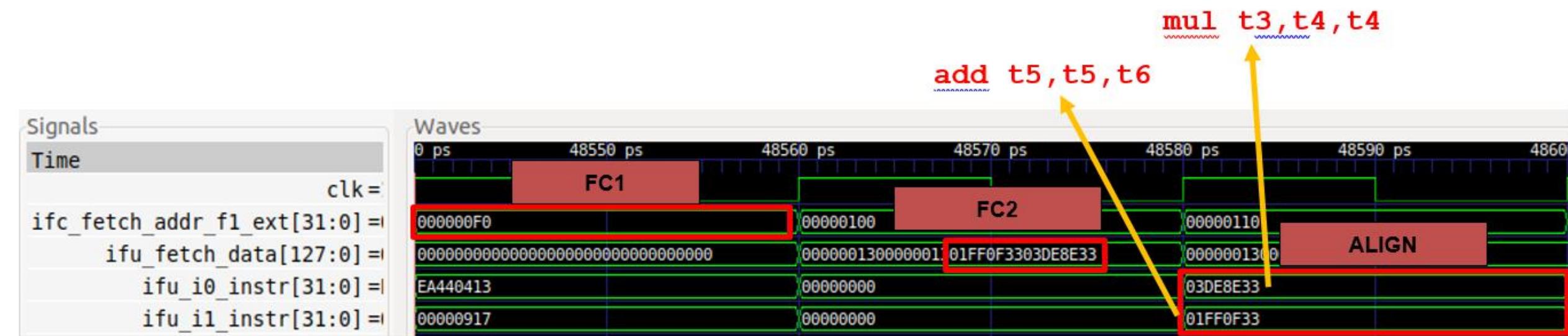
- 提交阶段：选择要写回到寄存器文件的结果。
- 写回阶段：使用写端口0和1将结果写入寄存器文件。控制流水线寄存器提供寄存器标识符和使能信号（在译码阶段生成）。

# RVfpga实验11：示例程序

```
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
```

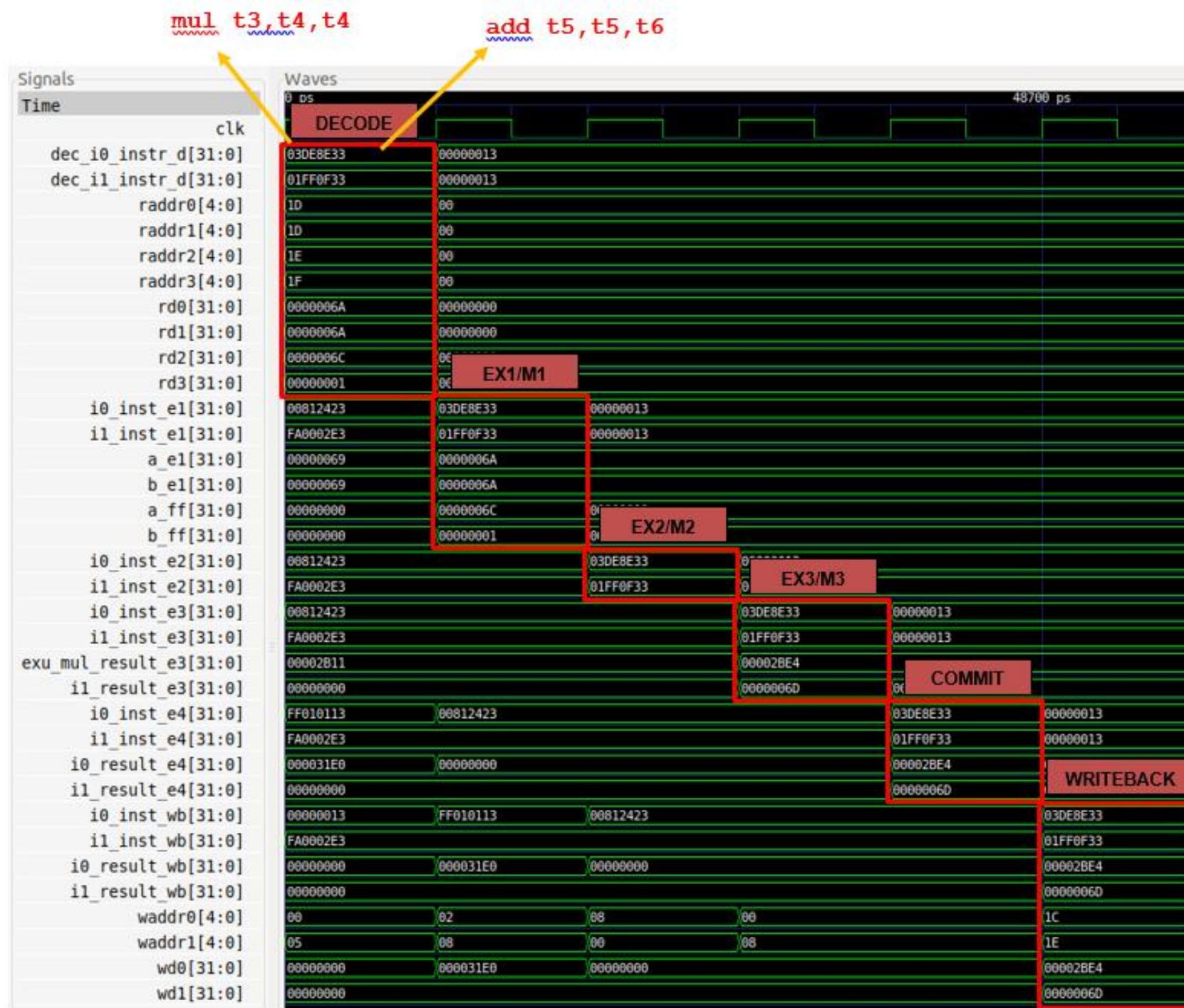
# RVfpga实验11：FC1、FC2和对齐阶段仿真



# RVfpga实验11：仿真分析

- **FC1:** 计算mul指令的地址：
  - ifc\_fetch\_addr\_f1\_ext = 0x000000F0
- **FC2:** 从指令存储器的128位指令束中提取两条指令（以红色显示）：
  - ifu\_fetch\_data = 0x000000130000001301FF0F3303DE8E33
- 对齐：提取两条指令并分发到SweRV EH1的两个通路。
  - 通路0：ifu\_i0\_instr = 0x03DE8E33 (mul指令)
  - 通路1：ifu\_i1\_instr = 0x01FF0F33 (add指令)

## RVfpga实验11：译码、EX1/2/3、提交和WB阶段仿真



# RVfpga实验11：仿真分析

- **译码:** 从寄存器文件中读取指令的操作数，然后提供给乘法管道和I1管道。
- **EX1/2/3和提交:** 计算加法和乘法。
  - $i0\_result\_e4 = exu\_mul\_result\_e3 = 0x6A * 0x6A = 0x2BE4$
  - $i1\_result\_e4 = i1\_result\_e3 = 0x6C + 0x01 = 0x6D$
- **写回:** 将结果写回到寄存器文件中。
  - $waddr0 = 0x1C \quad wd0 = 0x2BE4$
  - $waddr1 = 0x1E \quad wd1 = 0x6D$

# RVfpga实验11：硬件计数器

- 硬件计数器是目前大多数处理器中包含的一组特殊用途寄存器，用于记录表中所示的指标。

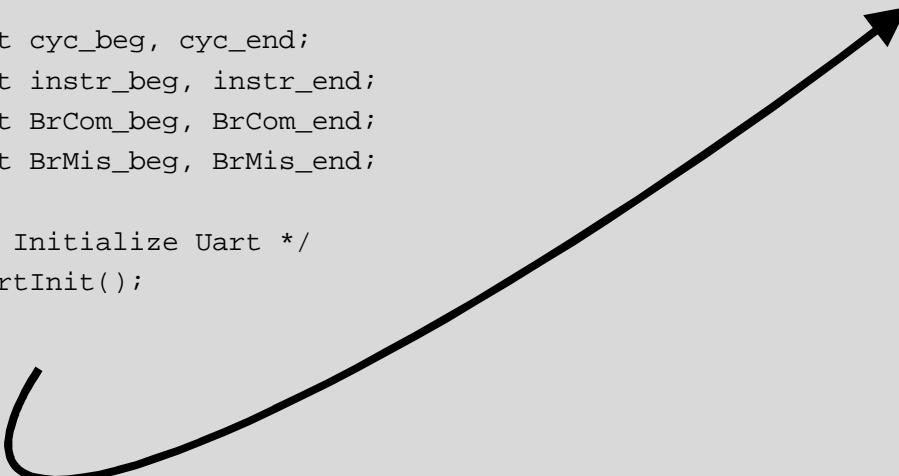
0	Reserved	17	CSR read/write	34	Cycles SB/WB stalled
1	Cycles clock active	18	CSR write rd==0	35	Cycles DMA DCCM transaction stalled
2	I-Cache hits	19	Ebreak	36	Cycles DMA ICCM transaction stalled
3	I-Cache misses	20	Ecall	37	Exceptions taken
4	Intrs commited	21	Fence	38	Timer interrupts taken
5	Intrs committed 16-b	22	Fence.i	39	Exteranal interrupts taken
6	Intrs committed 32-b	23	Mret	40	TLU flushes
7	Intrs aligned	24	Branches commited	41	Branch error flushes
8	Intrs 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实验11：通过Western Digital的PSP使用性能计数器

```
#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_COMMITED_ALL);
    pspPerformanceCounterSet(D_PSP_COUNTER2, E_BRANCHES_COMMITED);
    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);
}
```

# 实验12： 算术/逻辑指令： add指令

# RVfpga实验12：简介

- 本实验将分析SweRV EH1流水线中的算术和逻辑指令流，重点关注add指令。
- 分为两部分：
  - add指令的基本分析
  - add指令的高级分析
- 这两种分析使用相同的示例程序，如下一张幻灯片所示。

# RVfpga实验12：示例程序

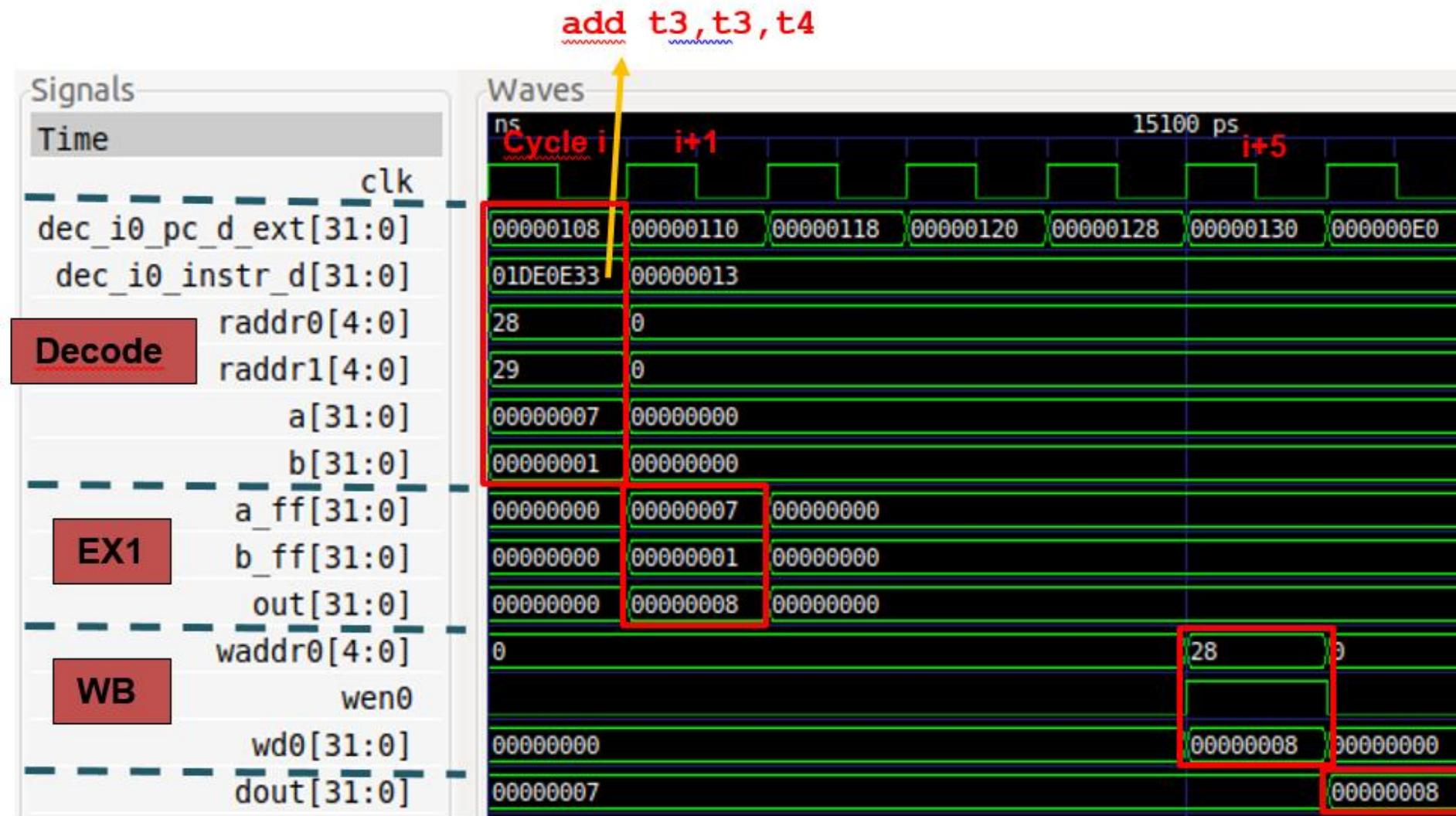
```
.globl main
main:

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

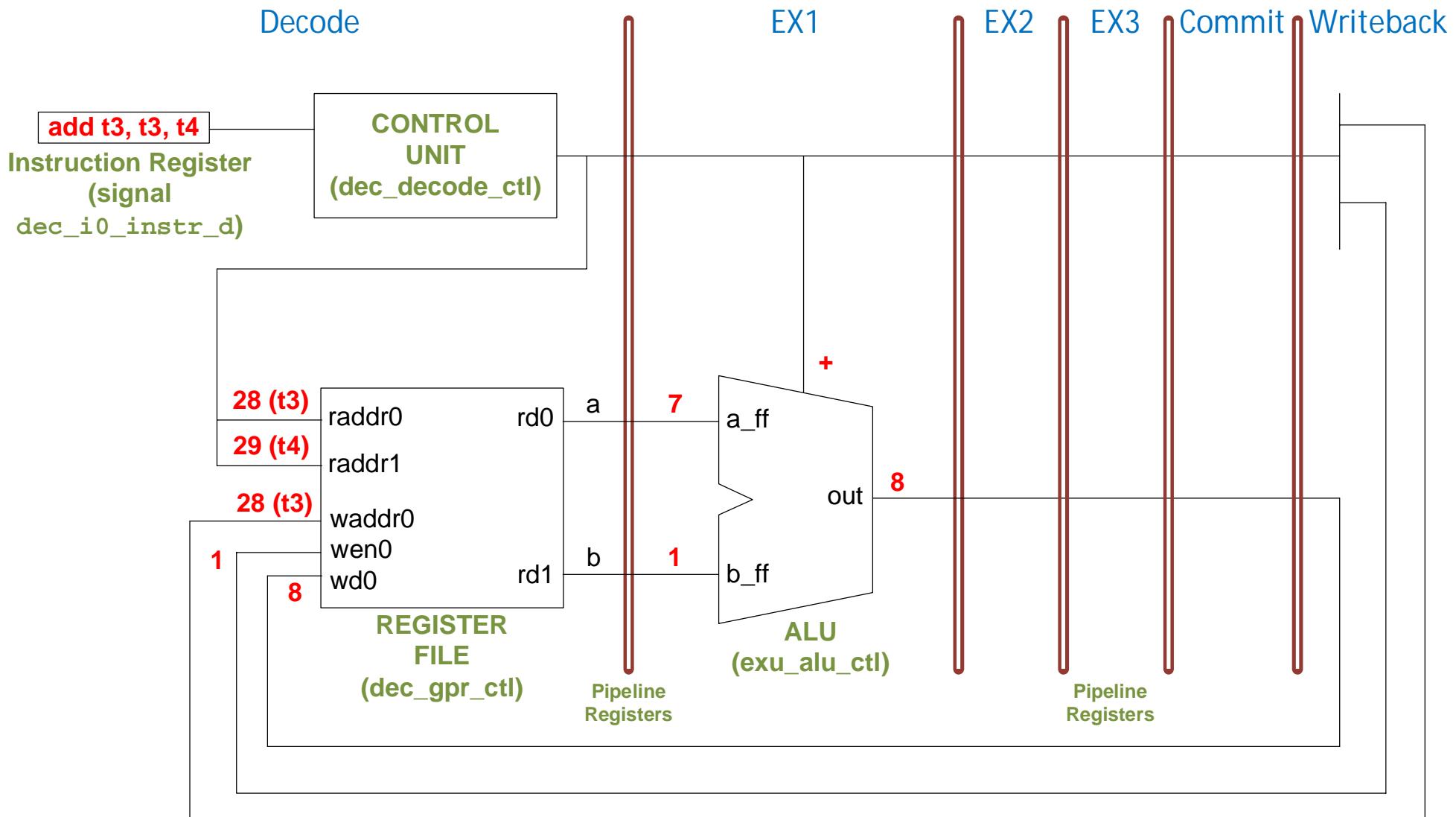
REPEAT:
    INSERT_NOPS_10
    add t3, t3, t4      # t3 = t3 + t4
    INSERT_NOPS_10
    beq zero, zero, REPEAT # Repeat the loop

.end
```

# RVfpga实验12：基本分析 - 仿真



# RVfpga实验12：基本分析 – SweRV EH1流水线



# RVfpga实验12：基本分析 - 仿真

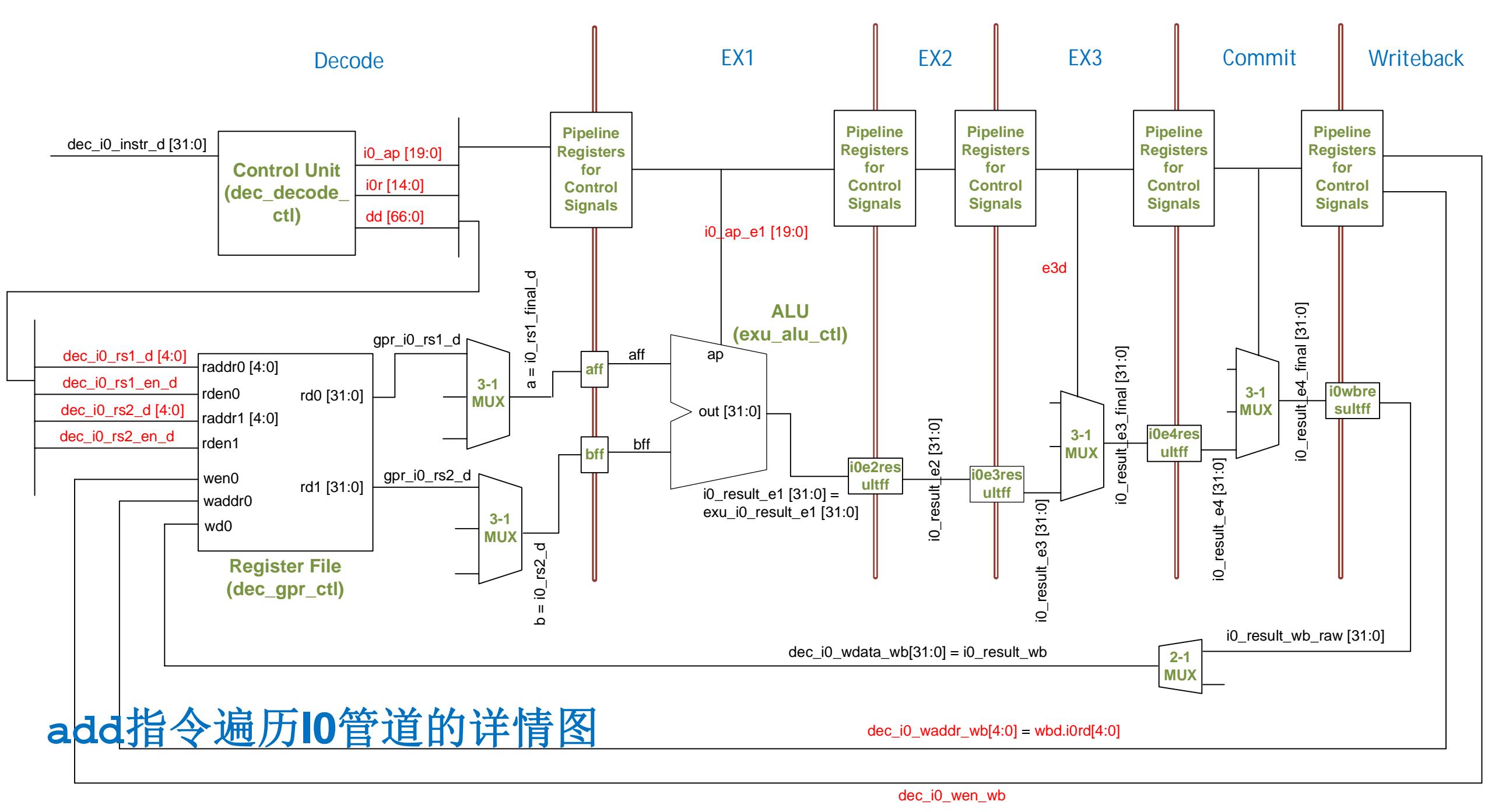
- **周期i:** 译码：信号dec\_i0\_instr\_d包含32位机器指令0x01DE0E33。在RISC-V中，add指令的字段为： 00 | rs1 | 000 | rd | 0110011

在这一阶段，将生成控制信号并读取寄存器文件。此外，操作数将传播到**I0**管道。

- **周期i+1:** **EX1:** 执行add指令。将加法的结果作为ALU的输出（信号out = 8）。
- **周期i+5:** 写回：将加法的结果写回到寄存器文件中：wd0 = 0x8、wen0 = 1且waddr0 = 0x28

# RVfpga实验12：高级分析

- 下一页的图为add指令遍历IO管道的详情图。



# 实验13： 访存指令： lw和sw指令

# RVfpga实验13：简介

- 实验13将分析存储器读写。
- 分为三部分：
  - 低延时加载：检查读取低延时DCCM（不暂停处理器）时的加载/存储管道。
  - 低延时存储：检查DCCM的存储。
  - 高延时加载和存储：在读/写Nexys A7开发板上的DDR主存储器时重复之前的分析。

# RVfpga实验13：加载 - 示例程序

```
.globl main

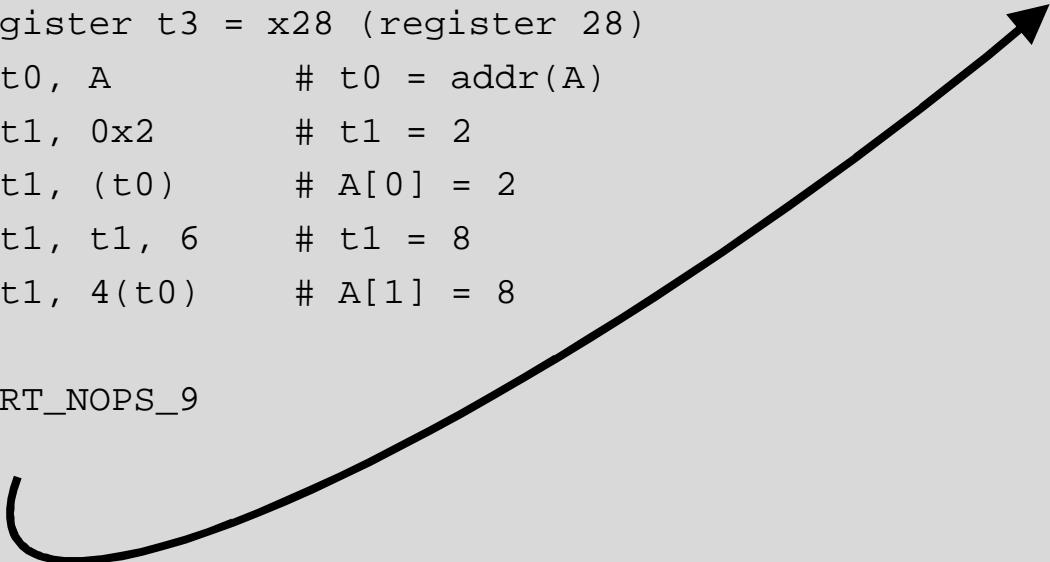
.section .midccm
A: .space 8

.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

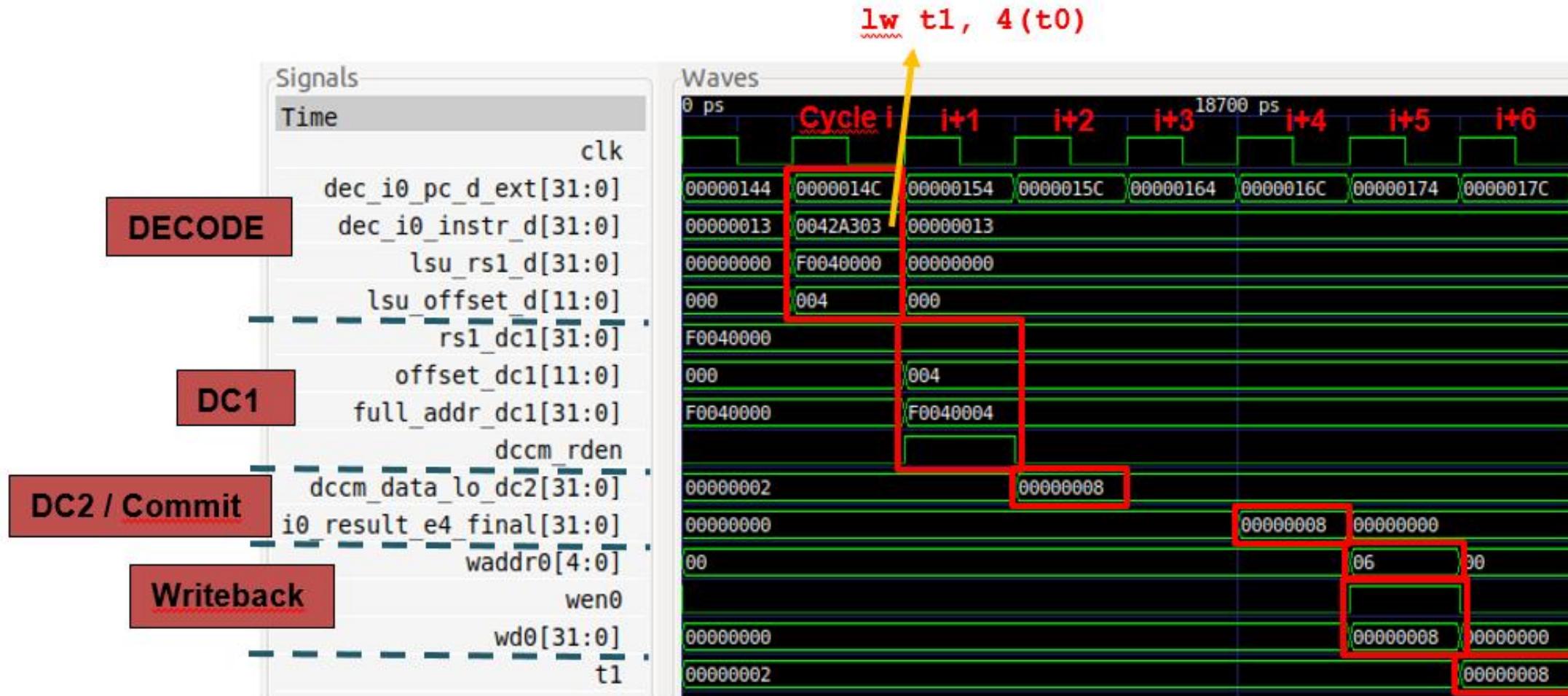
INSERT_NOPS_9
```



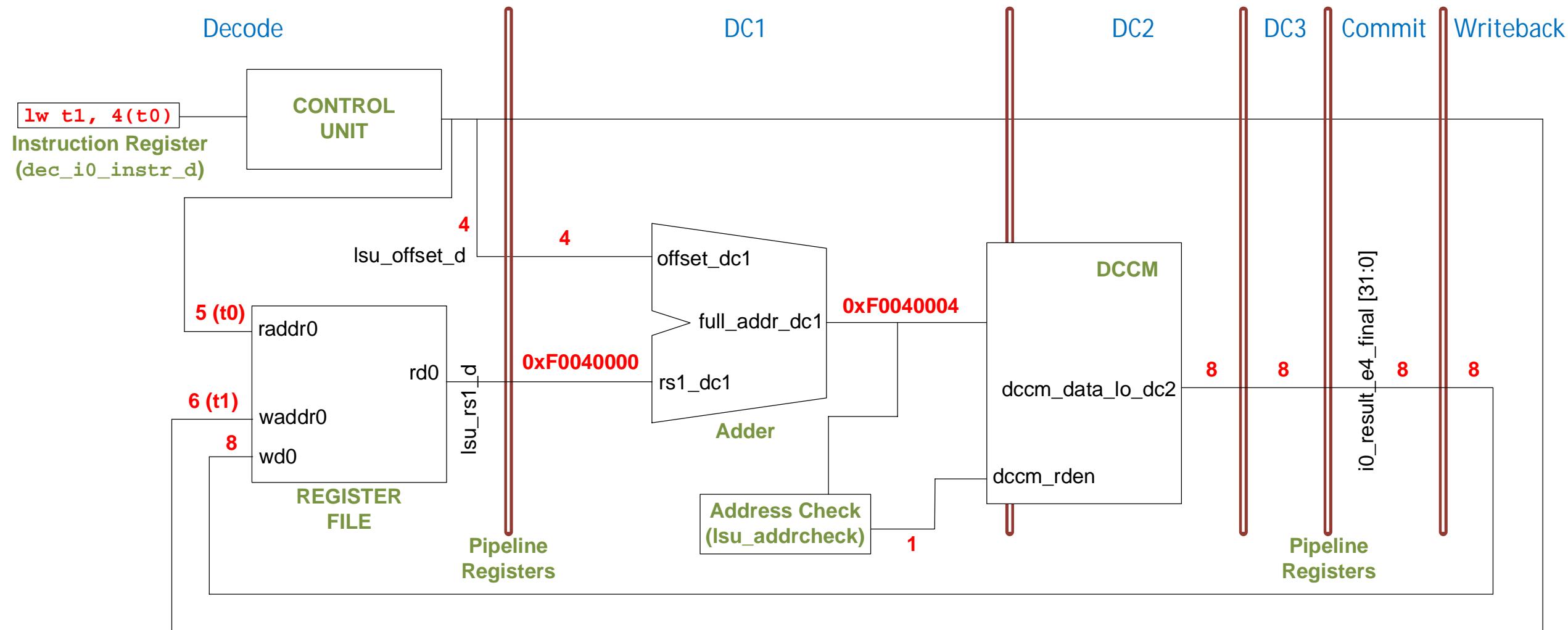
```
REPEAT:
    INSERT_NOPS_1
    lw  t1, (t0)
    INSERT_NOPS_9
    INSERT_NOPS_4
    lw  t1, 4(t0)
    INSERT_NOPS_10
    INSERT_NOPS_4
    beq zero, zero, REPEAT    # Repeat the loop
```

```
.end
```

# RVfpga实验13：低延时加载 - 仿真



# RVfpga实验13：低延时加载 - SweRV EH1流水线

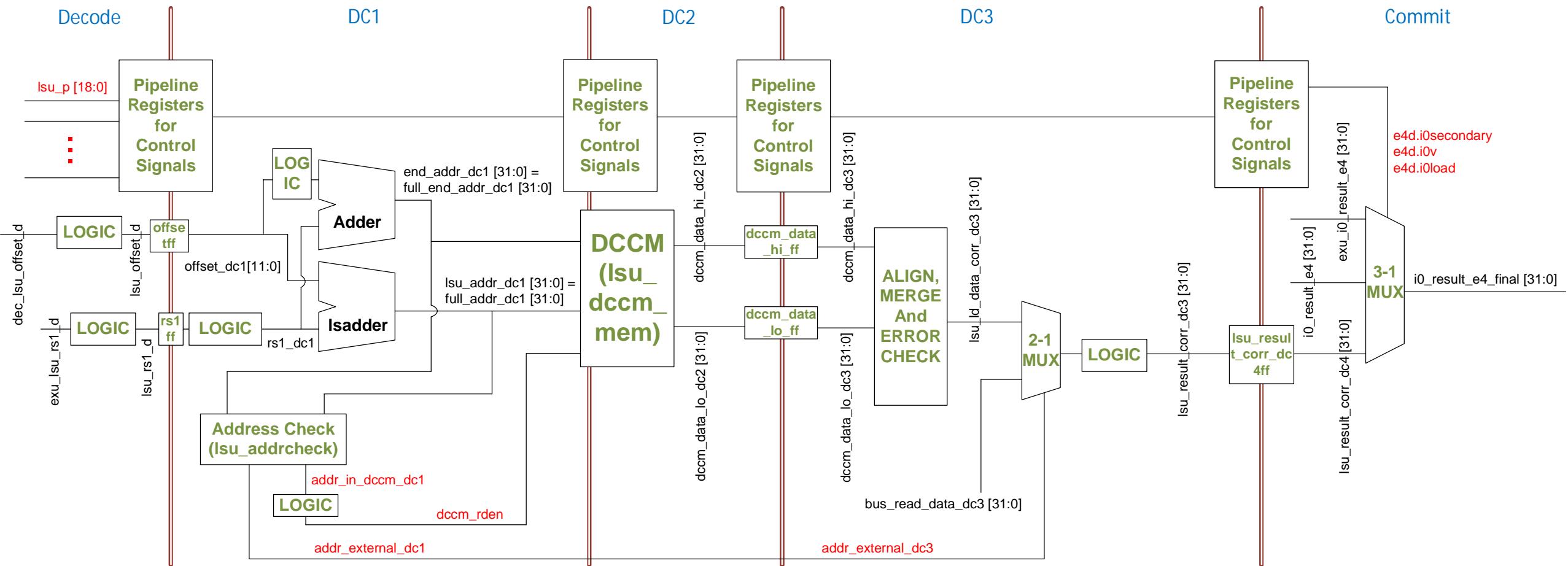


# RVfpga实验13：低延时加载 - 分析

- **周期i:** 译码: 生成控制信号并读取操作数:
  - t0 = 0xF0040000
  - Offset = 0x004
- **周期i+1: DC1:**
  - 计算地址: full\_addr\_dc1 = 0xF0040004
  - 查找访问的存储器区域 → dccm\_rden置为有效
- **周期i+2: DC2:** 读取DCCM → dccm\_data\_lo\_dc2 = 0x8
- **周期i+5:** 写回: 将从存储器读取的值写回到寄存器文件:
  - wd0 = 0x8
  - wen0 = 1
  - waddr0 = 0x6

# RVfpga实验13：高延时加载分析

- 下一张幻灯片上的图为lw指令通过IO管道执行期间遍历的主要元素的详情图。
- 实验11已给出相关说明，但下面的新图只关注LSU管道，提供与lw指令相关的详细信息。
- 实验13的文档提供下图中lw指令执行期间每个阶段的深入说明（此处不包含）。



1w遍历IO管道的详情图

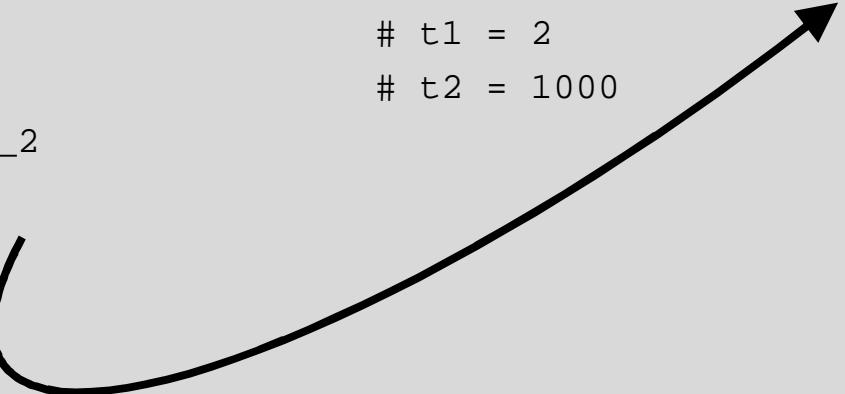
# RVfpga实验13: 存储 - 示例程序

```
.globl main

.section .midccm
A: .space 4000

.text

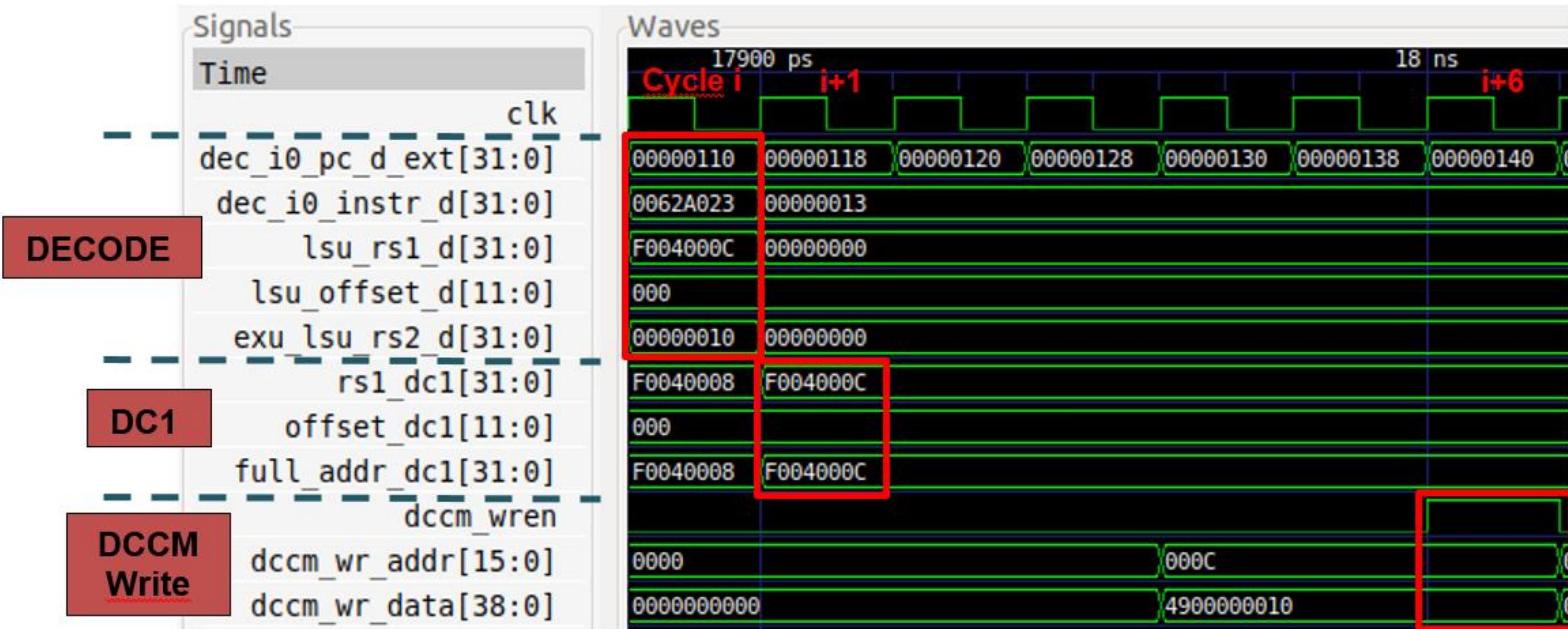
main:
la t0, A           # t0 = addr(A)
li t1, 0x2          # t1 = 2
li t2, 1000         # t2 = 1000
INSERT_NOPS_2


```

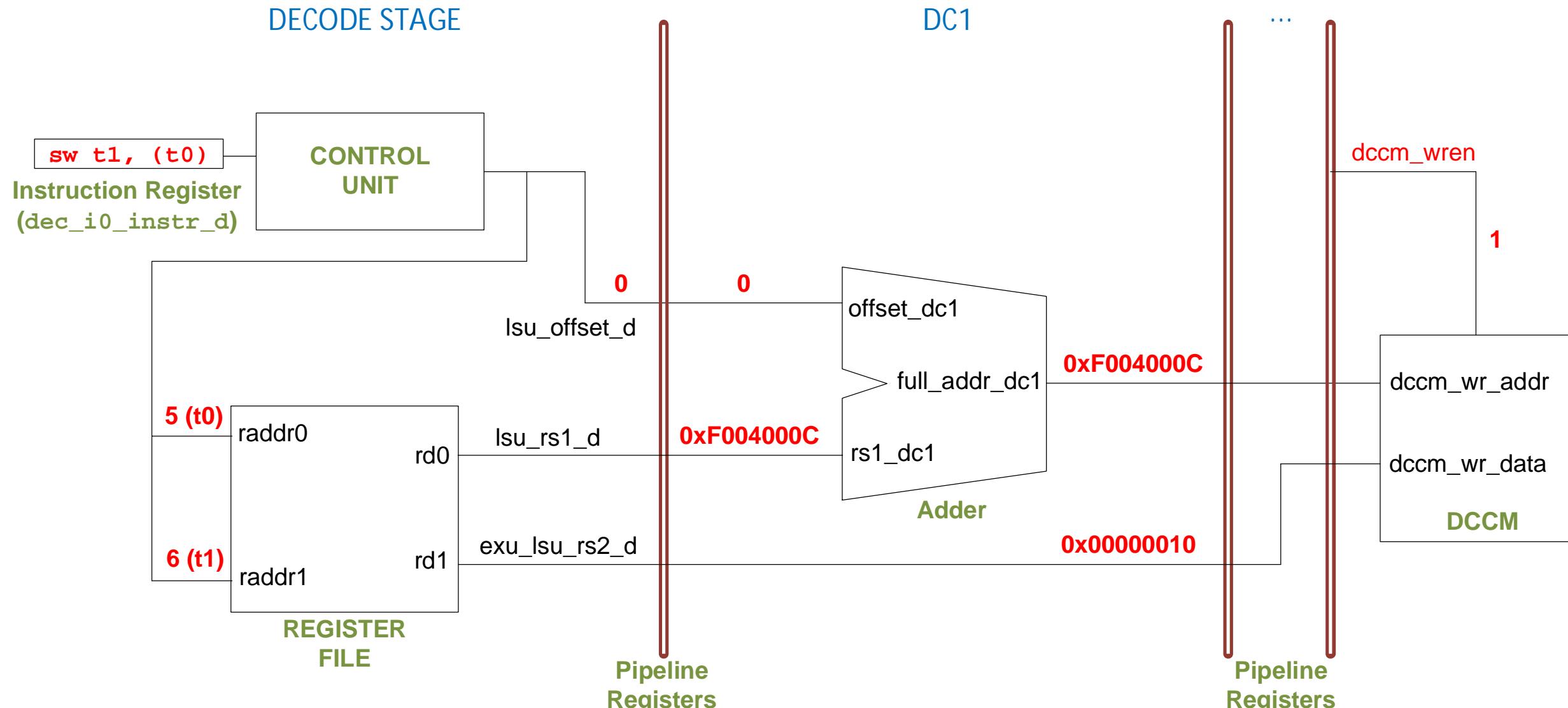
```
REPEAT:
    sw t1, (t0)
    INSERT_NOPS_10
    INSERT_NOPS_4
    lw t1, (t0)
    INSERT_NOPS_10
    add t1,t1,t1
    add t0,t0,0x04
    add t2,t2,-1
    INSERT_NOPS_10
    bne t2, zero, REPEAT    # Repeat the loop
    nop
    nop

.end
```

# RVfpga实验13：低延时存储 – 仿真



# RVfpga实验13：低延时存储 – SweRV EH1流水线



# RVfpga实验13：存储基本分析 - 仿真

- **周期i:** 译码：生成控制信号并读取操作数：

- $t_0 = 0xF004000C$
  - Offset = 0x000
  - $t_1 = 0x10$

- **周期i+1: DC1:**

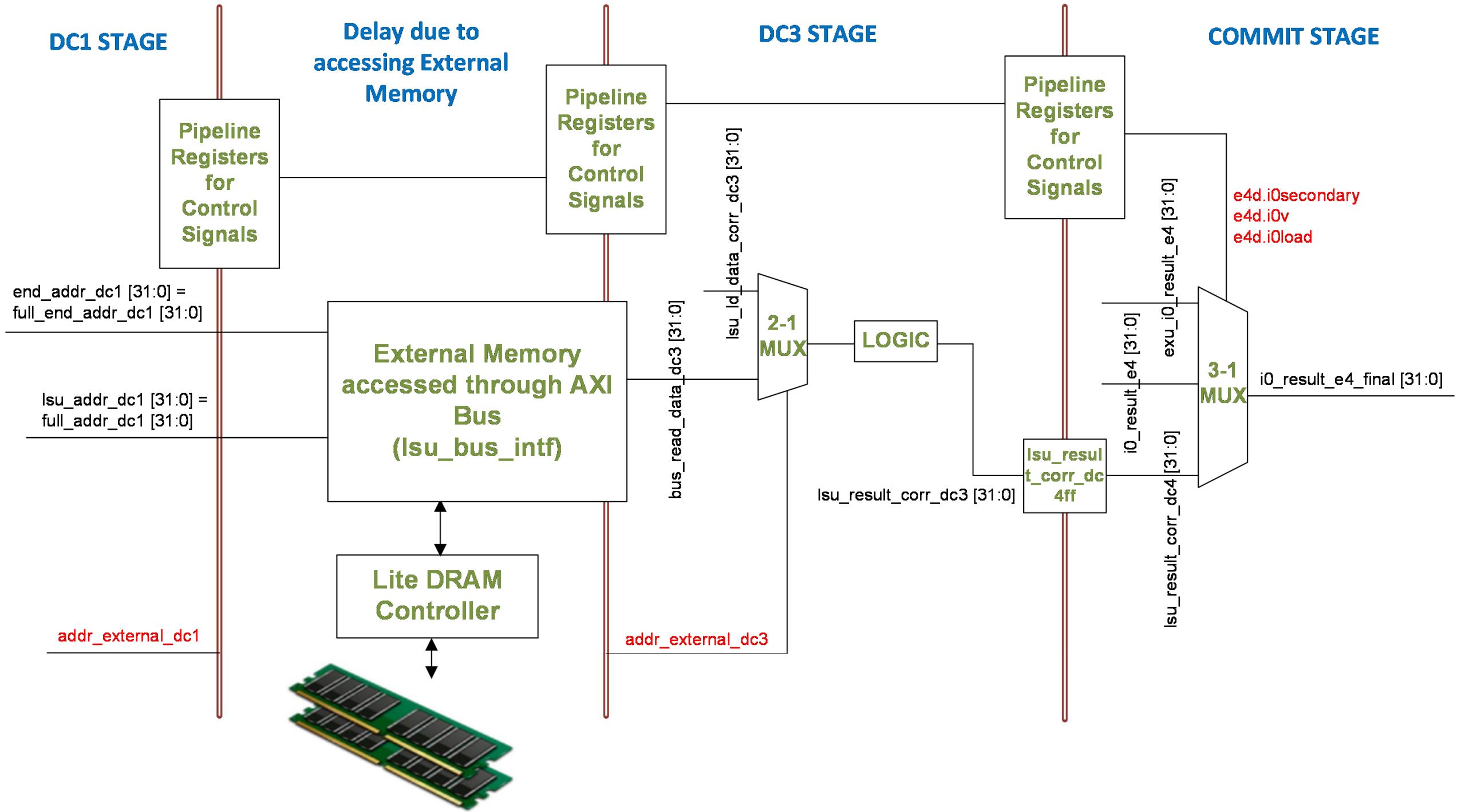
- 计算地址：full\_addr\_dc1 = 0xF004000C

- **周期i+6: DCCM写入:**

- dccm\_wr\_addr = 0x000C
  - dccm\_wr\_data = 0x10

# RVfpga实验13：外部存储器加载

- 下一张幻灯片上的图给出了lw指令为读取主存储器而遍历的主要路径。
- 处理器必须暂停来等待来自外部存储器的数据。
- 外部存储器通过AXI总线访问，该总线为Lite DRAM控制器提供地址，然后在几个周期后将请求的数据对齐并发送到DC3阶段。
- DC3阶段的2选1多路复用开关选择来自外部存储器的数据，而不是来自DCCM的数据。



# RVfpga实验13: 外部存储器 - 示例程序

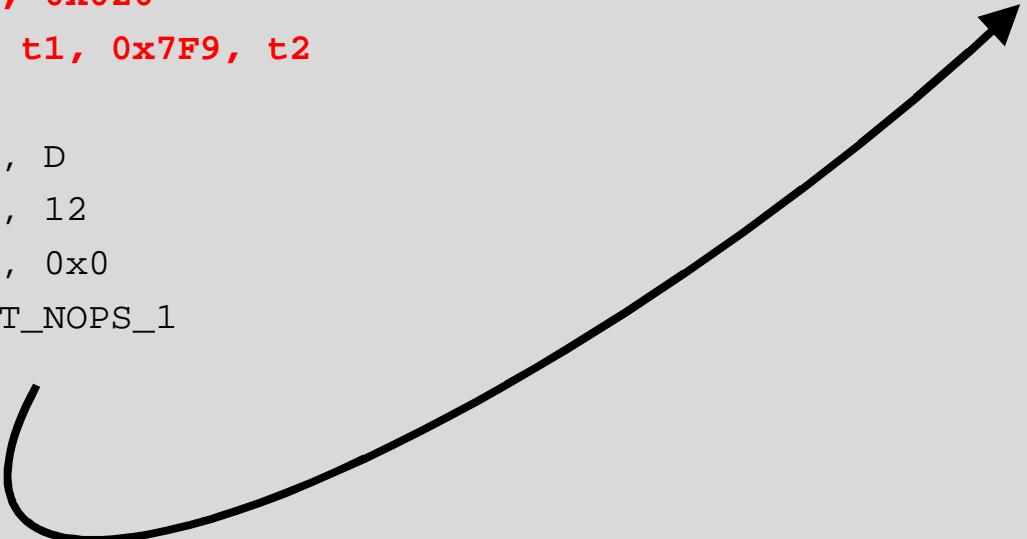
```
.globl main

.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
```

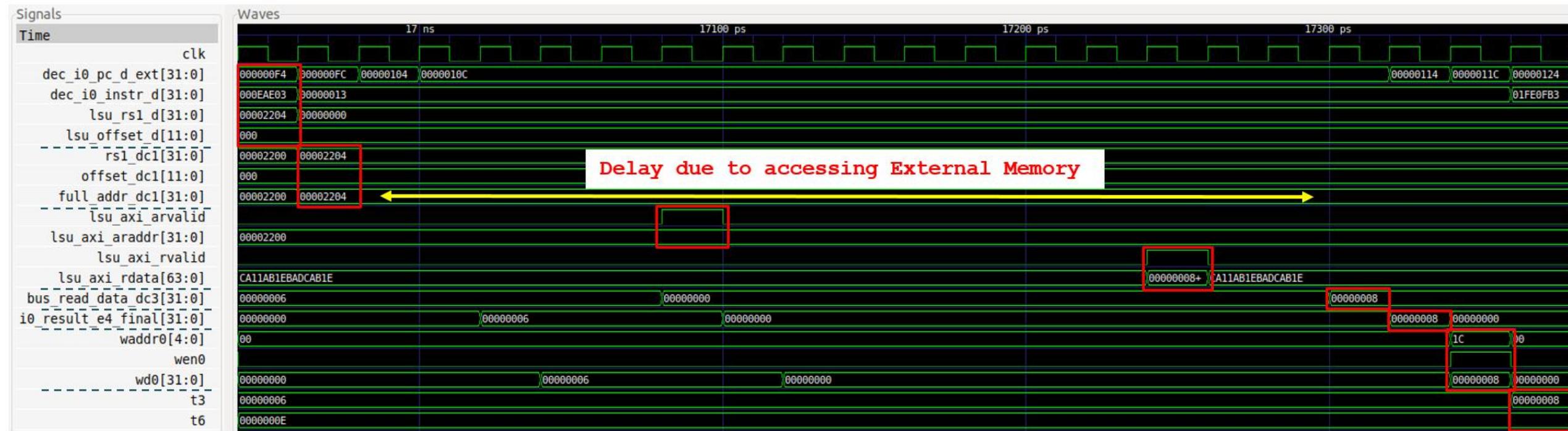


```
REPEAT:
    lw t3, (t4)
    add t5, t5, -1
    INSERT_NOPS_10
    add t6, t3, t6
    add t4, t4, 4
    INSERT_NOPS_9
    bne t5, zero, REPEAT      # Repeat the loop

    INSERT_NOPS_4

.end
```

# RVfpga实验13：外部存储器 - 仿真



# RVfpga实验13：外部存储器 – 分析

- 在译码阶段，示例的第四次迭代中的地址为0x00002204。
- 然后，地址通过AXI总线发送到外部存储器：
  - lsu\_axi\_arvalid = 1
  - lsu\_axi\_araddr = 0x00002200
- 几个周期后，外部存储器返回通过AXI总线读取的**64位**数据
  - lsu\_axi\_rdata = 0x0000000800000006
  - lsu\_axi\_rvalid = 1
- 最后，从64位数据中提取所请求的32位数据，插入到主流水线路径中，并写入寄存器文件。

# 实验14： 结构冒险

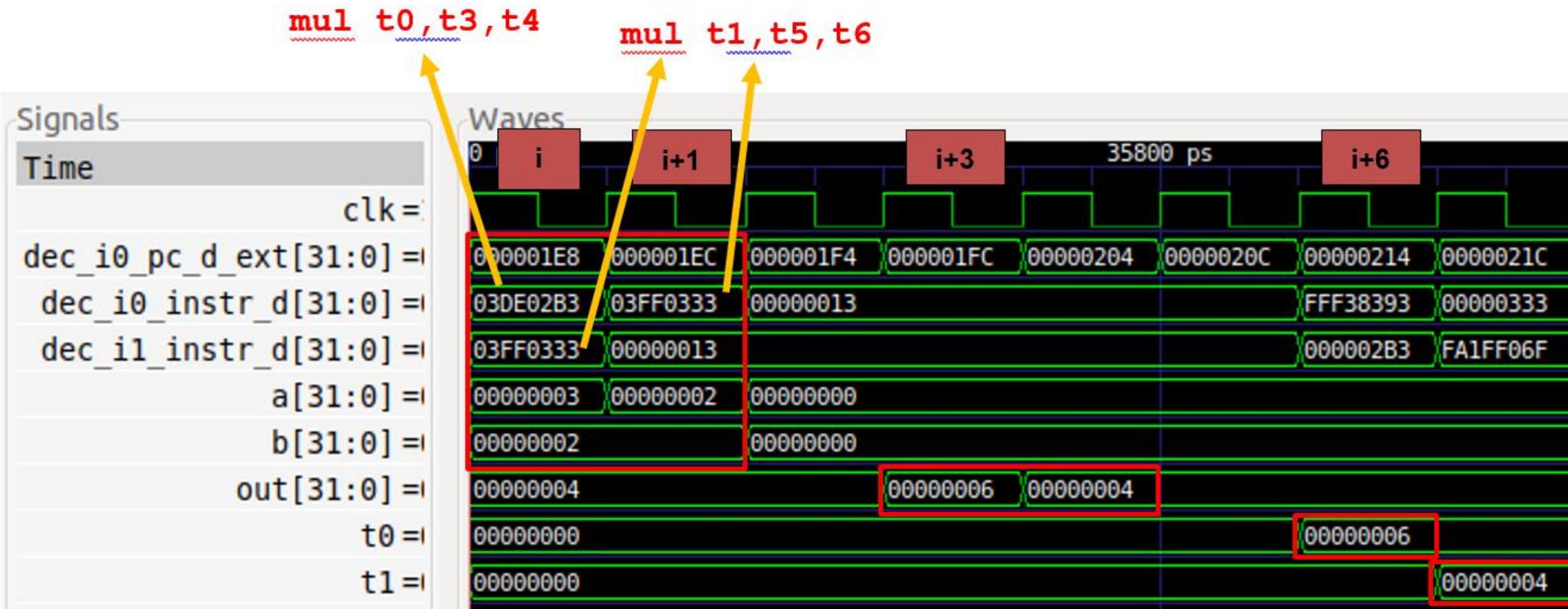
# RVfpga实验14：简介

- 实验14将说明两种结构冒险（它们具有不同的性能-成本权衡）。
  - **单元冲突：**二条mul指令在同一周期到达译码阶段。乘法器是流水线的，因此第二条mul指令仅延迟一个周期。硬件成本和性能下降（仅一个周期）较低。
  - **寄存器文件写端口冲突：**三条指令在同一周期内到达写回阶段，其中之一是在几个周期之前执行的非阻塞加载。**SweRV EH1**具有三个（而不是两个）写端口。可避免结构冒险（不会导致性能损失），但由于额外的寄存器文件端口，其硬件成本较高。
  - 请注意，div指令也可能导致冒险（在实验的附录中进行讨论）。

# RVfpga实验14：2条mul指令 - 示例程序

```
.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
```

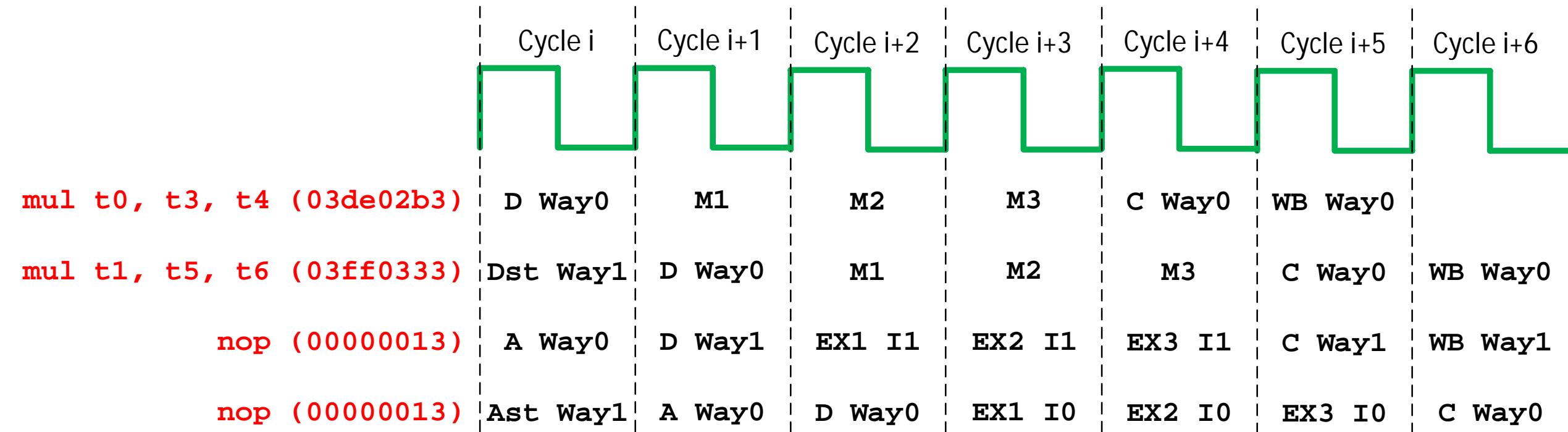
# RVfpga实验14：2条mul指令 - 仿真



# RVfpga实验14：2条mul指令 - 分析

- **周期i:** 两条mul指令在同一周期到达译码阶段。结构冒险将阻止第二条mul指令进行。
- **周期i+1:** 第一条mul指令在流水线乘法器的第一阶段（M1）执行，而第二条mul指令在译码阶段等待。
- **周期i+2:** 第一条mul指令在流水线乘法器的第二阶段（M2）执行，而第二条mul指令在第一阶段（M1）执行。
- **周期i+3:** 第一条mul指令获取结果：out = 0x6。
- **周期i+4:** 第二条mul指令获取结果：out = 0x4。
- **周期i+6:** 寄存器文件使用第一条mul指令的结果（t0 = 0x6）更新。
- **周期i+7:** 寄存器文件使用第二条mul指令的结果（t1 = 0x4）更新。

# RVfpga实验14：2条mul指令 - 图



# RVfpga实验14：3个同步写操作 - 示例程序

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



Figure 6. Verilator simulation for the example from Figure 4

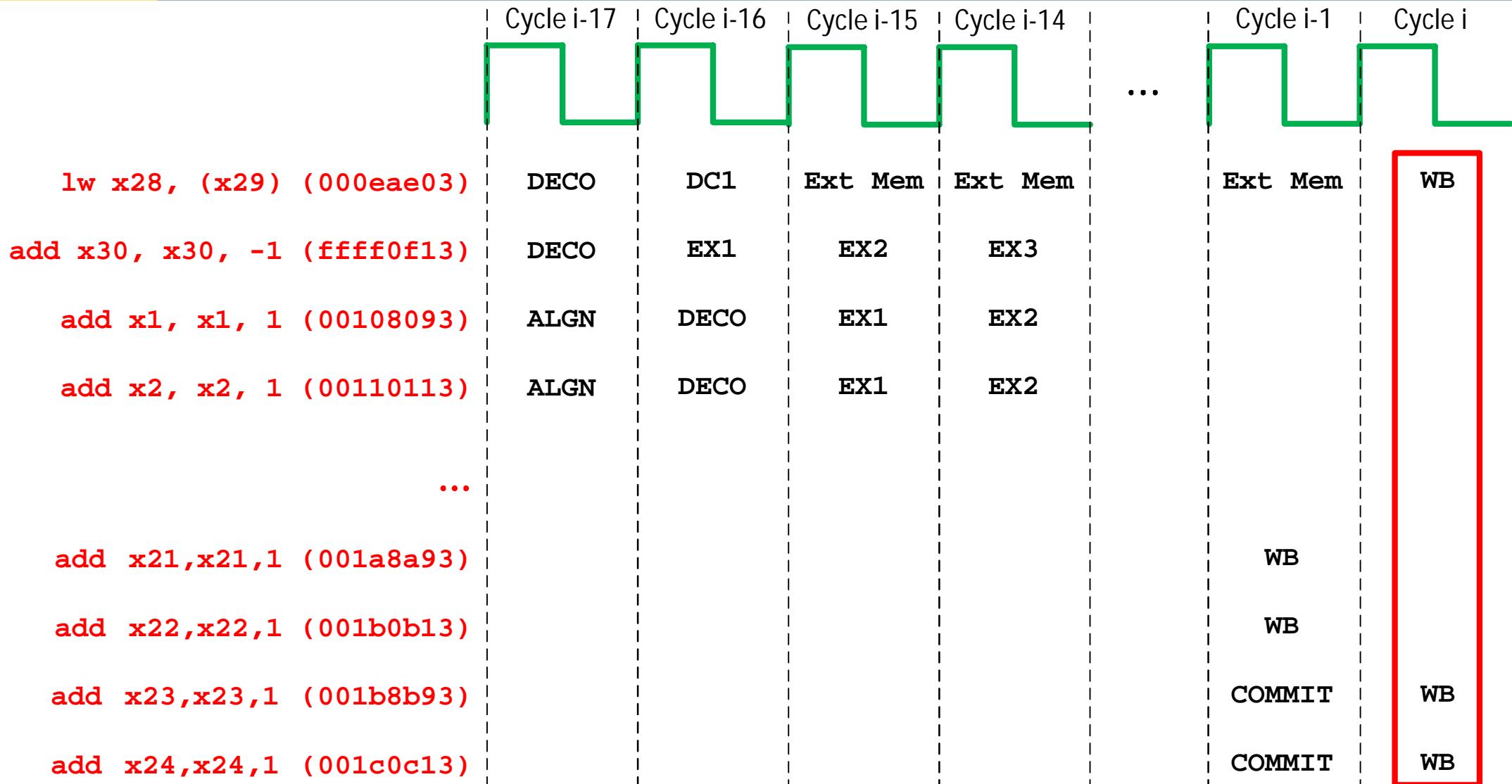
Three simultaneous writes to the Register File:

- `lw` writes register x28 (0x1C)
- `add` writes register x23 (0x17)
- `add` writes register x24 (0x18)

# RVfpga实验14：3个同步写操作 – 分析

- **周期i-17:** lw指令处于译码阶段。
- **周期i-16:** 计算有效存储器地址并通过AXI总线发送到外部存储器。load指令等待几个周期，以便外部存储器提供数据。
- **周期i-5:** 对两条冲突的add指令进行译码。
- **周期i:** lw指令和两条冲突的add指令进入写回阶段（写入寄存器文件），由于寄存器文件具有三个写端口，因此可以实现。

# RVfpga实验14：3个同步写操作 - 图



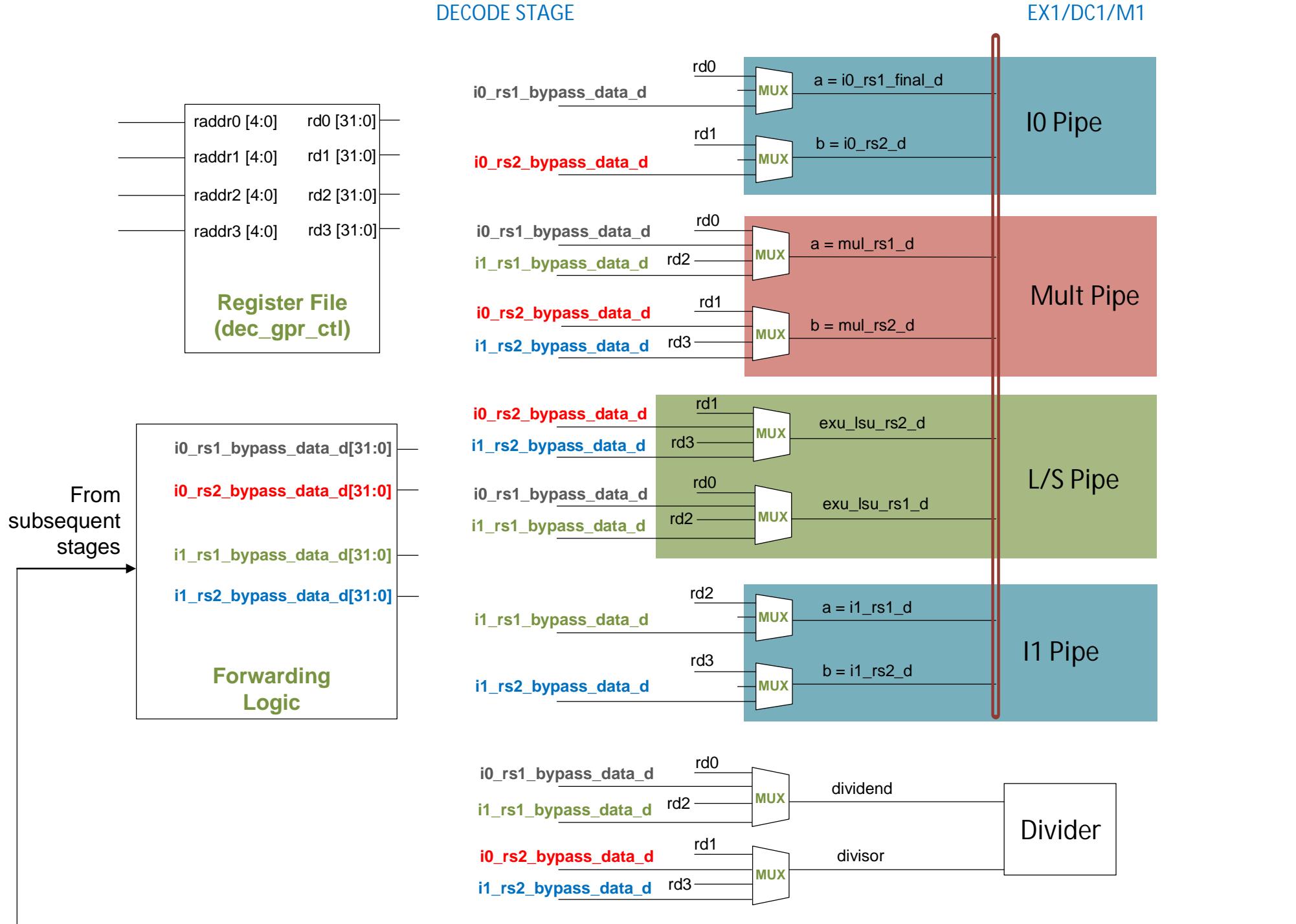
# 实验15： 数据冒险

# RVfpga实验15：简介

- 实验15将分析如何解除**RAW**数据冒险。
- 可通过以下方式解除**RAW**数据冒险：暂停处理器或转发（也称为旁路）来自后面阶段执行的指令的值。
- 两种情况分析：
  - 通过转发到译码阶段（使用几个新的多路开关）解除**RAW**数据冒险
  - 使用两个额外的**ALU**在提交阶段解除**RAW**数据冒险

# RVfpga实验15：通过转发解除数据冒险

- 转发到译码阶段需要在功能单元（ALU、乘法器、计算DC1中的有效地址的加法器等）前面添加多路复用开关，以从寄存器文件或后续阶段选择操作数。
- 下一张幻灯片上的图显示了译码阶段的转发值。转发逻辑为每个通路中的两个源操作数中的每一个生成旁路



# RVfpga实验15：通过转发解除数据冒险 - 示例

```
.globl Test_Assembly
.text

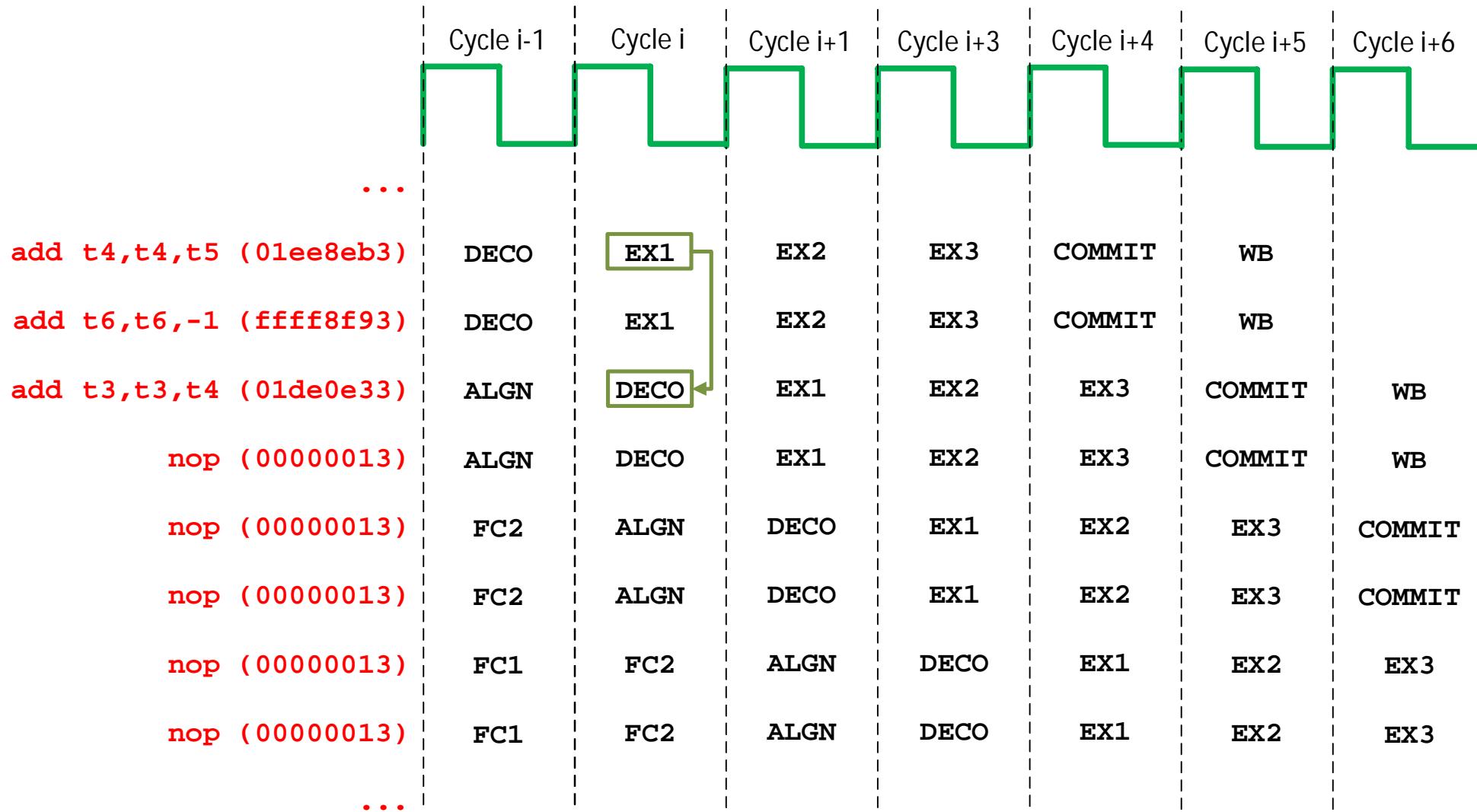
Test_Assembly:

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

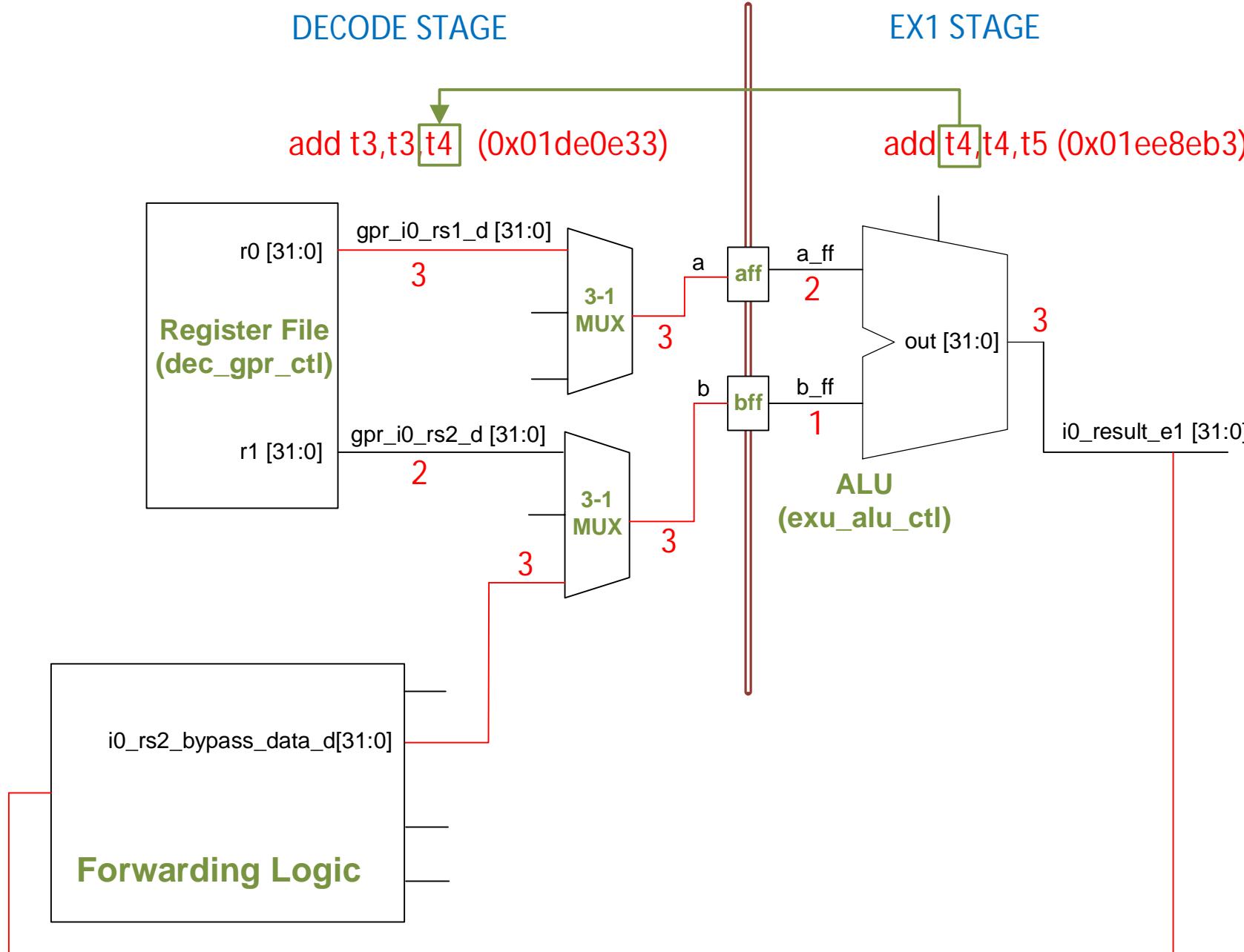
REPEAT:
    INSERT_NOPS_8
    add t4, t4, t5          # t4 = t4 + t5 (t4 = 2 + 1)
    add t6, t6, -1
    add t3, t3, t4          # t3 = t3 + t4 (t3 = 3 + 3)
    INSERT_NOPS_9
    li t3, 0x3
    li t4, 0x2    li t5, 0x1    bne t6, zero, REPEAT      # Repeat the loop

.end
```

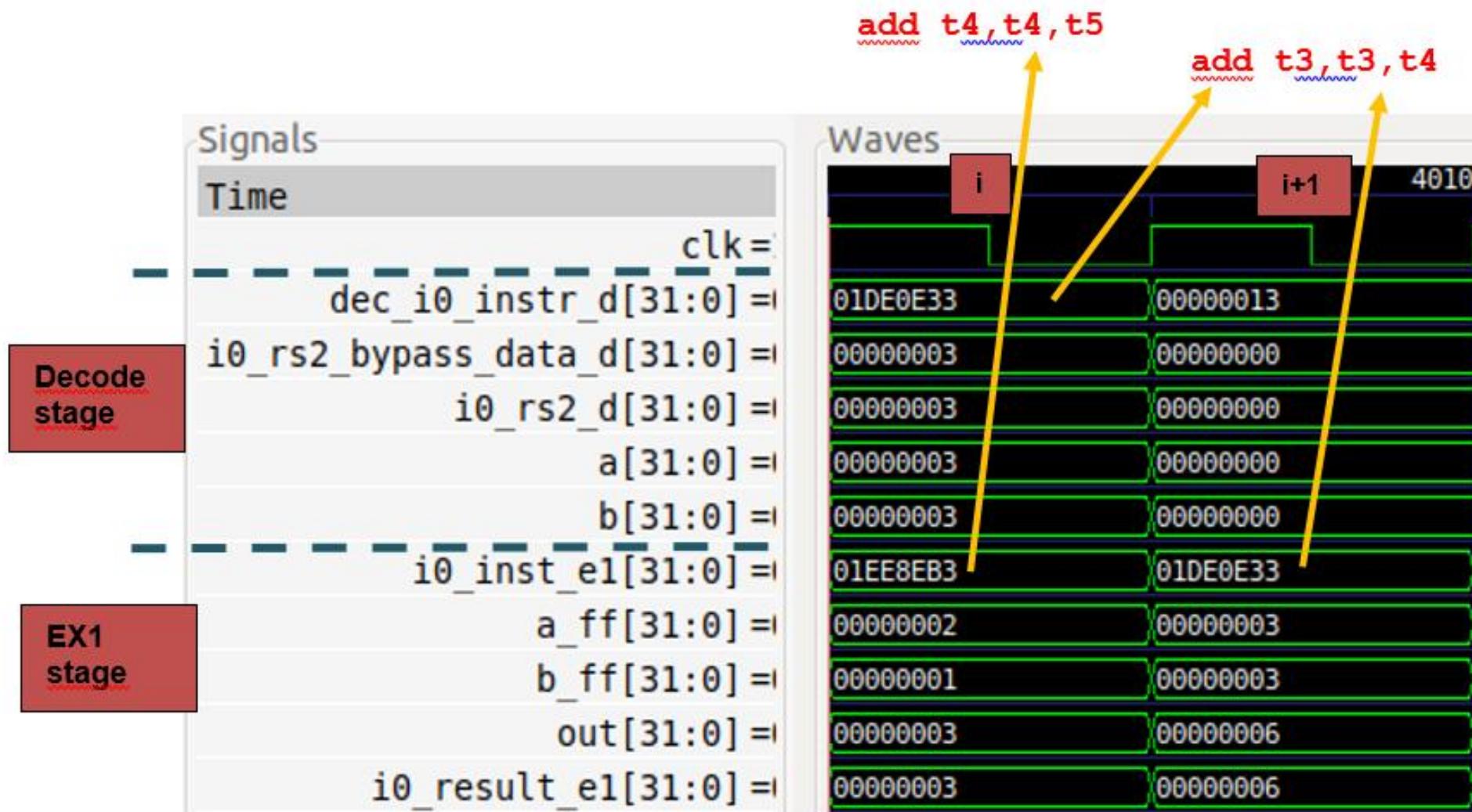
# RVfpga实验15：通过转发解除数据冒险 – 图



# RVfpga实验15：通过转发解除数据冒险 - 流水线

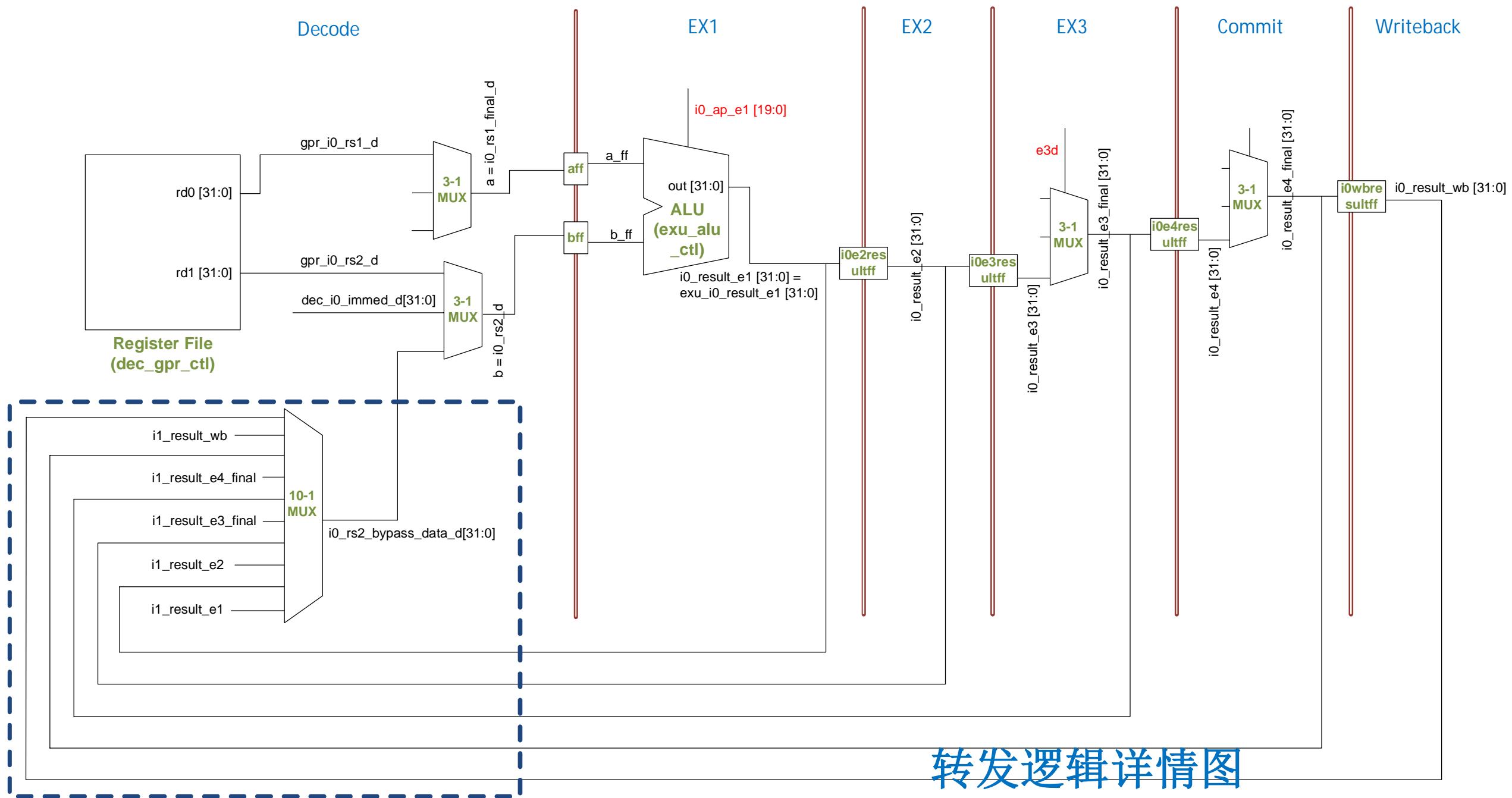


# RVfpga实验15：通过转发解除数据冒险 - 仿真



# RVfpga实验15：通过转发解除数据冒险 - 分析

- 指令add t4, t4, t5 (0x01ee8eb3)：
  - 周期*i*:** 此add指令处于I0管道的EX1阶段 (`i0_inst_e1 = 0x01ee8eb3`)。它在ALU中计算以下加法：
    - $a_{ff}(2) + b_{ff}(1) = out(3)$
  - 结果在译码阶段发送到转发逻辑。
- 指令add t3, t3, t4 (0x01de0e33)：
  - 周期*i*:** 此add指令处于通路0的译码阶段 (`dec_i0_instr_d = 0x01de0e33`)。转发逻辑将EX1阶段的结果 (`i0_result_e1`) 转发到译码阶段 (`i0_rs2_bypass_data_d`)。两个3选1多路复用开关生成操作数，具体如下：
    - 操作数  $a = 3$  (来自寄存器文件)
    - 操作数  $b = 3$  (来自I0管道EX1阶段的ALU输出，经过转发逻辑)
  - 周期*i+1*:** 此add指令处于I0管道的EX1阶段 (`i0_inst_e1 = 0x01de0e33`)。它在ALU中计算正确的加法：
    - $a_{ff}(3) + b_{ff}(3) = out(6)$

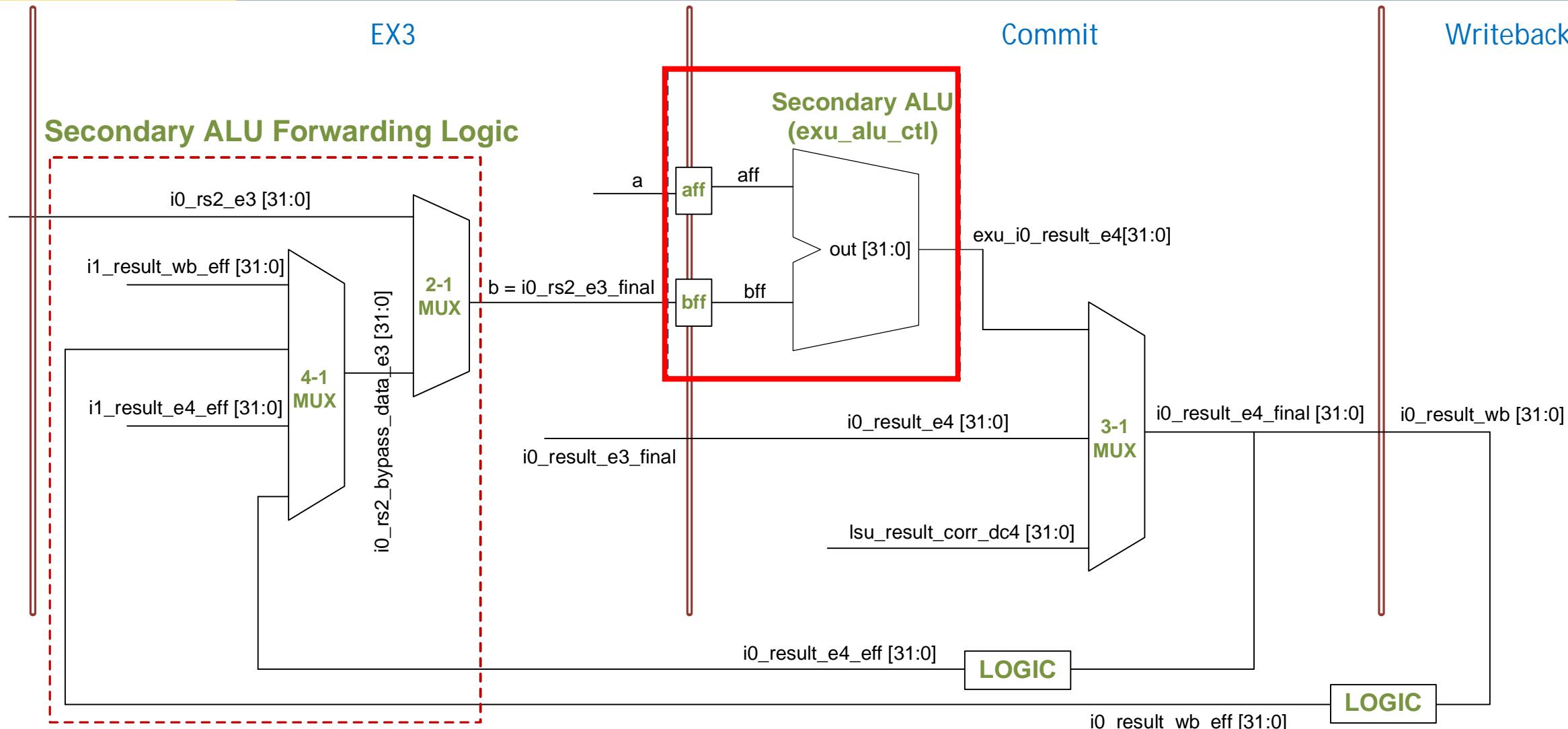


# 转发逻辑详情图

# RVfpga实验15：通过在提交阶段转发解除数据冒险

- 需要多个周期才能获得结果的指令（即多周期操作，例如lw、mul和div）无法转发到译码阶段。
- 但SweRV EH1在每路的提交阶段都添加一个额外的ALU（辅助ALU）。必要时，此ALU使用适当的输入重新计算算术逻辑运算。
- 因此，不会因暂停而丢失周期 – 但代价是添加了两个额外的ALU（每路一个）以及控制信号和逻辑。

# RVfpga实验15：通过在提交阶段转发解除数据冒险 – 流水线



# RVfpga实验15：通过在提交阶段转发解除数据冒险 – 示例

```
.globl Test_Assembly
```

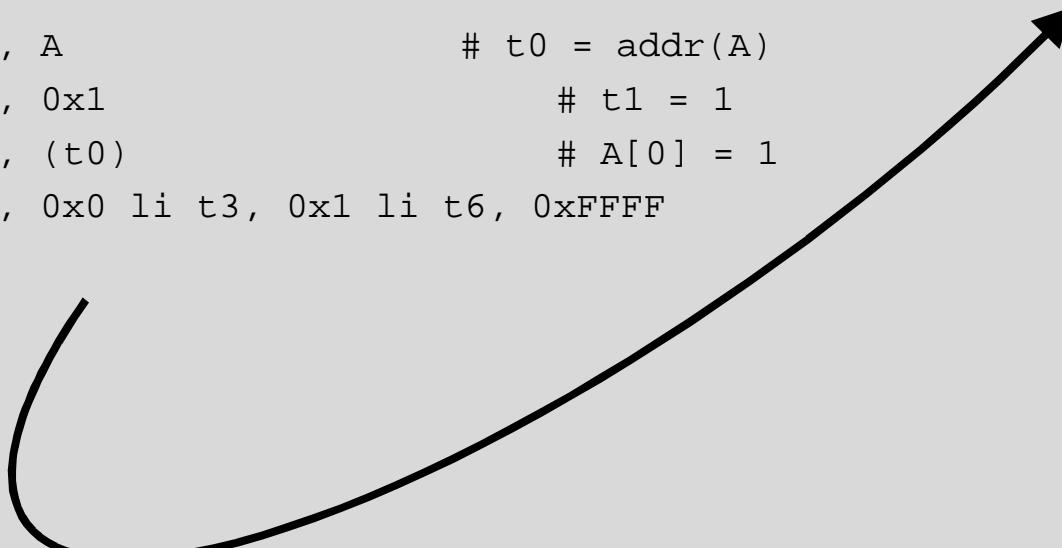
```
.section .midccm
```

```
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
```

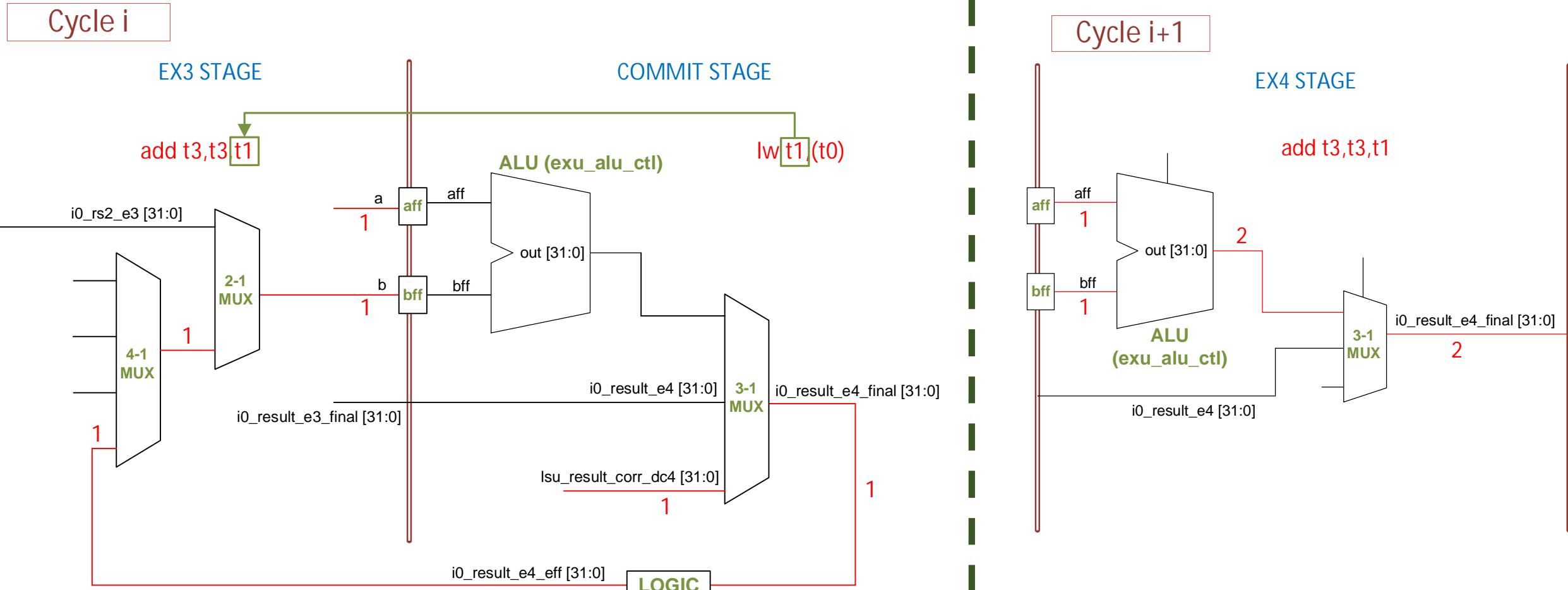
```
add t5, t5, 0x1
```

```
j REPEAT
```

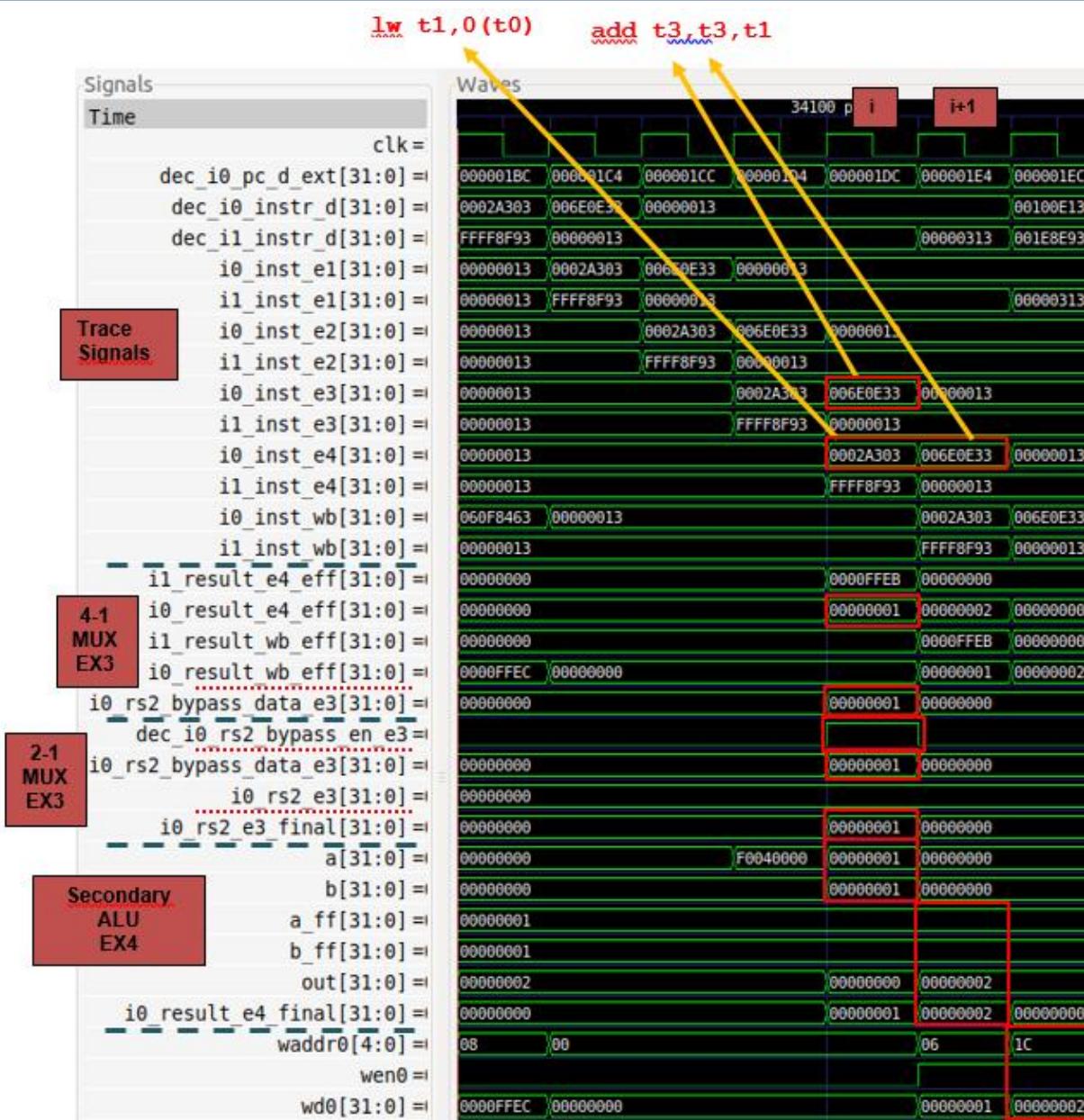
```
OUT:
```

```
.end
```

# RVfpga实验15：通过在提交阶段转发解除数据冒险 – 流水线



# RVfpga实验15：通过在提交阶段转发解除数据冒险 - 仿真



# RVfpga实验15：通过在提交阶段转发解除数据冒险 – 仿真

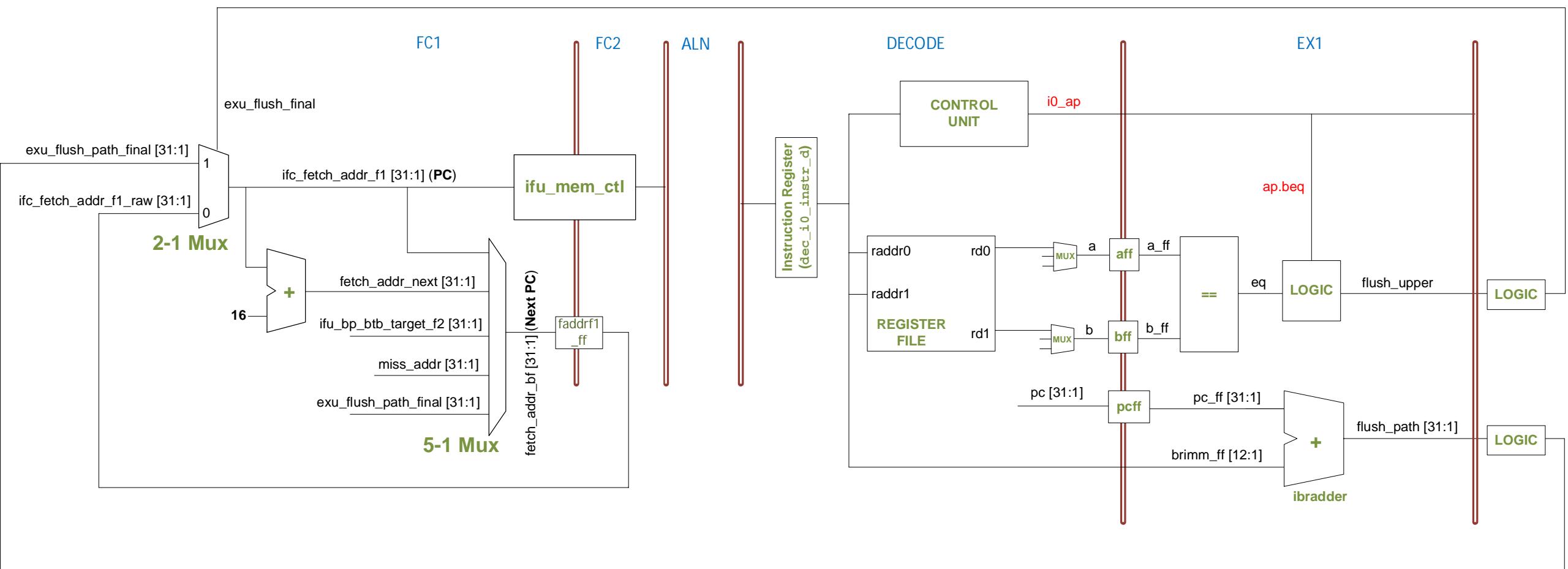
- 跟踪信号
  - 周期*i*: add指令处于通路0的EX3阶段 ( $i_0\_inst\_e3 = 0x006E0E33$ )，lw指令处于I0管道的提交阶段 ( $i_0\_inst\_e4 = 0x0002A303$ )。
  - 周期*i+1*: add指令处于通路0的提交阶段 ( $i_0\_inst\_e4 = 0x006E0E33$ )。
- 4选1多路开关
  - 周期*i*: 选择lw指令的结果（在提交阶段）：  
 $i_0\_rs2\_bypass\_data\_e3 = i_0\_result\_e4\_eff = 0x00000001$
- 2选1多路开关
  - 周期*i*: 由于lw和add之间的相关性，选择旁路值：  
 $i_0\_rs2\_e3\_final = i_0\_rs2\_bypass\_data\_e3 = 0x00000001$
- 提交阶段ALU
  - 周期*i+1*: 使用正确的值重新计算add运算：  
 $out = a\_ff + b\_ff = 0x00000001 + 0x00000001 = 0x00000002$
- 3选1多路开关
  - 周期*i+1*: 选择辅助ALU的输出 ( $exu\_i0\_result\_e4$ )。（当不存在相关性时，选择*i0\_result\_e4*。）

# 实验16： 控制冒险 和分支指令

# RVfpga实验16：简介

- 实验16将重点关注分支和跳转指令引起的**控制冒险**。
- 这些指令必须计算下一条指令的地址（对于非跳转/分支指令，下一条指令的地址是  $PC + 4$ ）。
- 控制冒险可能：
  - 暂停流水线，直到计算出下一条指令的地址，或
  - 预测是否会发生分支，并从预测的路径中取指。分支确定后，如果与预测相反，处理器可以清除取指的指令，如果与预测相同，则继续执行取指的指令。
- SweRV EH1有两个可能的分支预测器（Branch Predictor, BP）
  - 简单分支预测器：总是预测为不发生分支。性能较差，但没有硬件成本。
  - Gshare分支预测器：性能较高，需要花费额外的硬件成本。
- 本实验将使用naïve和Gshare BP分析beq指令的执行。

# RVfpga实验16: beq指令执行和PC计算

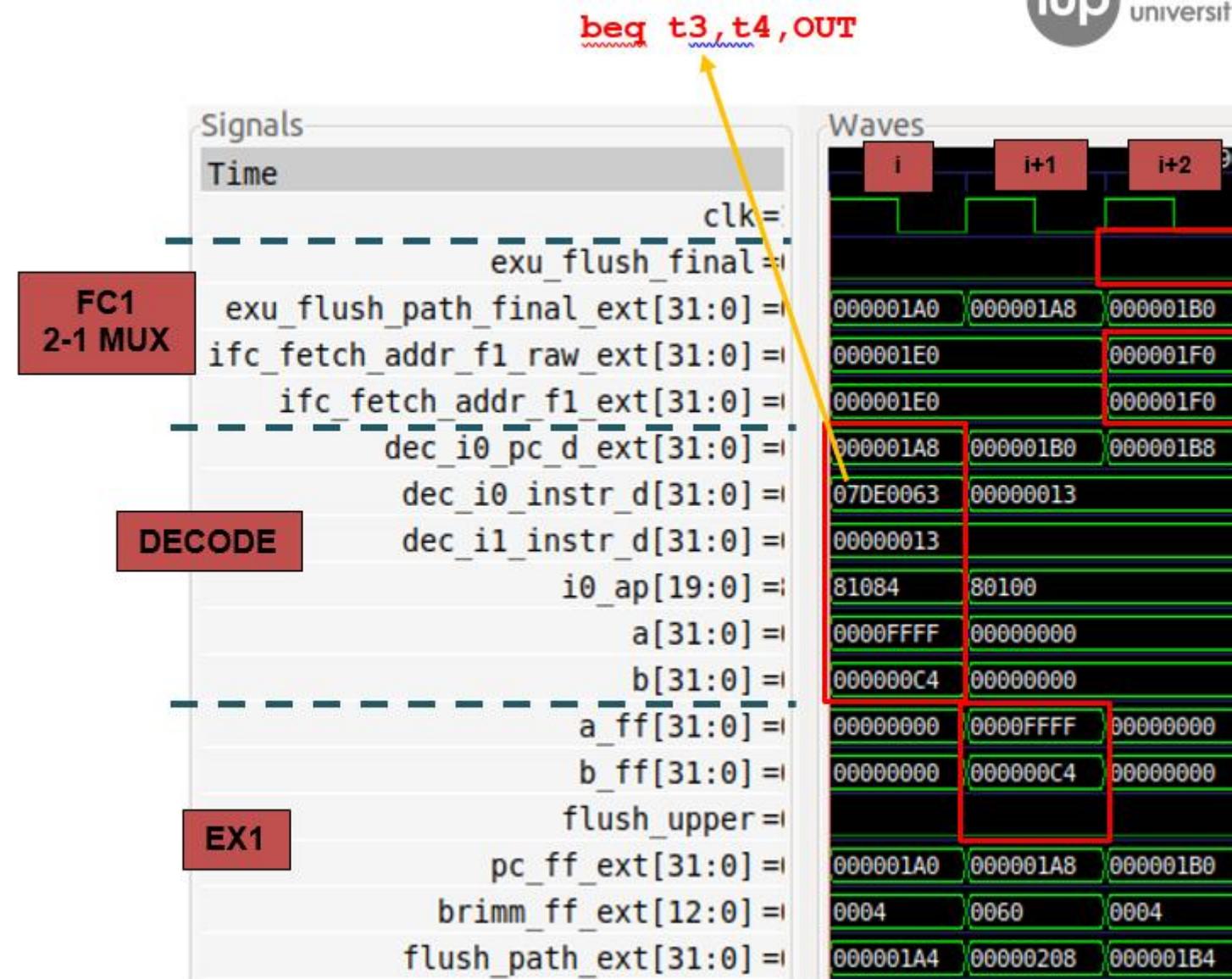


# RVfpga实验16: beq指令执行和PC计算 – 示例

```
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实验16：第一条beq指令的执行 - 仿真

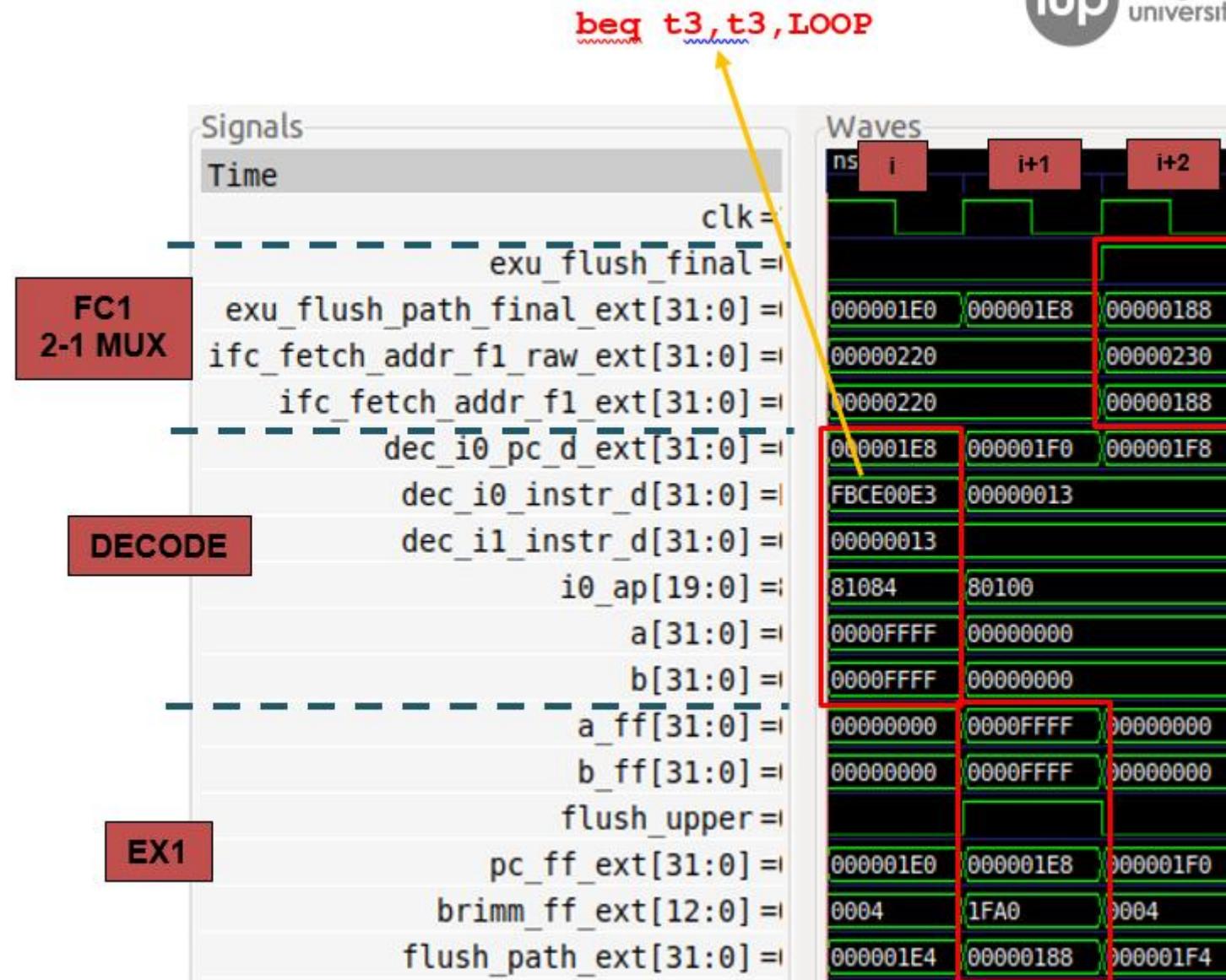
TOP university p



# RVfpga实验16：第一条beq指令的执行 – 分析

- **周期*i* - beq指令的译码阶段：**在通路0中对第一条beq指令（0x07DE0063）进行译码。生成控制信号、读取寄存器文件并将分支指令传送至I0管道。信号a和b（本例中分别为0xFFFF和0xC4）中包含下一阶段所使用的比较器的输入。
- **周期*i+1* - beq指令的EX1阶段：**执行beq指令。比较信号a\_ff和b\_ff。两个数（0xFFFF和0xC4）不同，因此不发生分支。在本例中，Gshare预测器被禁止，因此所有分支的预测结果均为不发生（i0\_ap.predict\_nt = 1）。因此，分支预测结果正确，流水线不清除（flush\_upper = 0）。
- **周期*i+2* - FC1阶段：**假设已预测分支并确定不发生分支，则正常按顺序取指。应注意exu\_flush\_final = 0, ifc\_fetch\_addr\_f1\_ext[31:0] = ifc\_fetch\_addr\_f1\_raw\_ext[31:0] = 0x000001F0。该地址指向下一个连续的128位指令束。

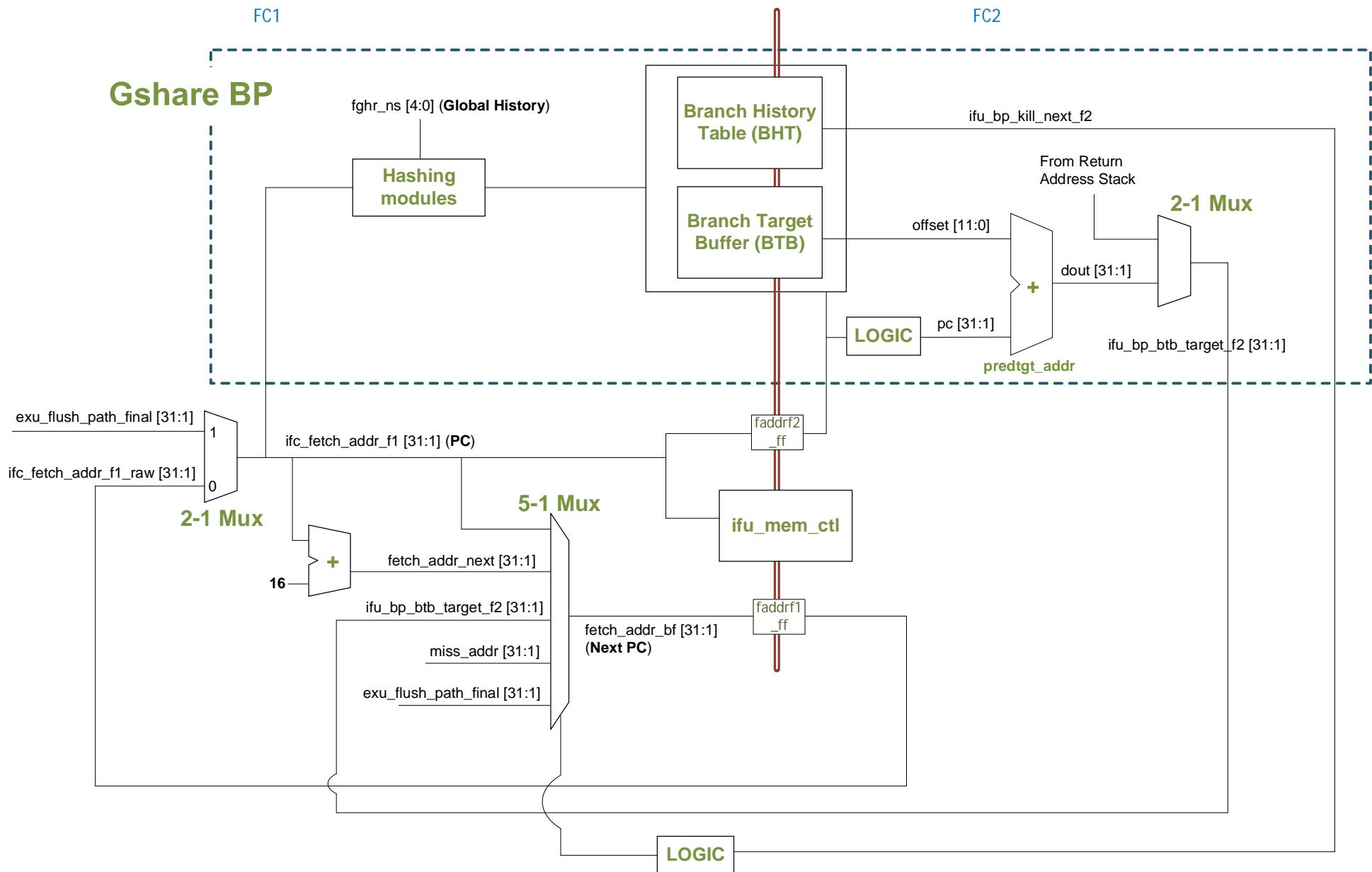
# RVfpga实验16：第二条beq指令的执行 - 仿真



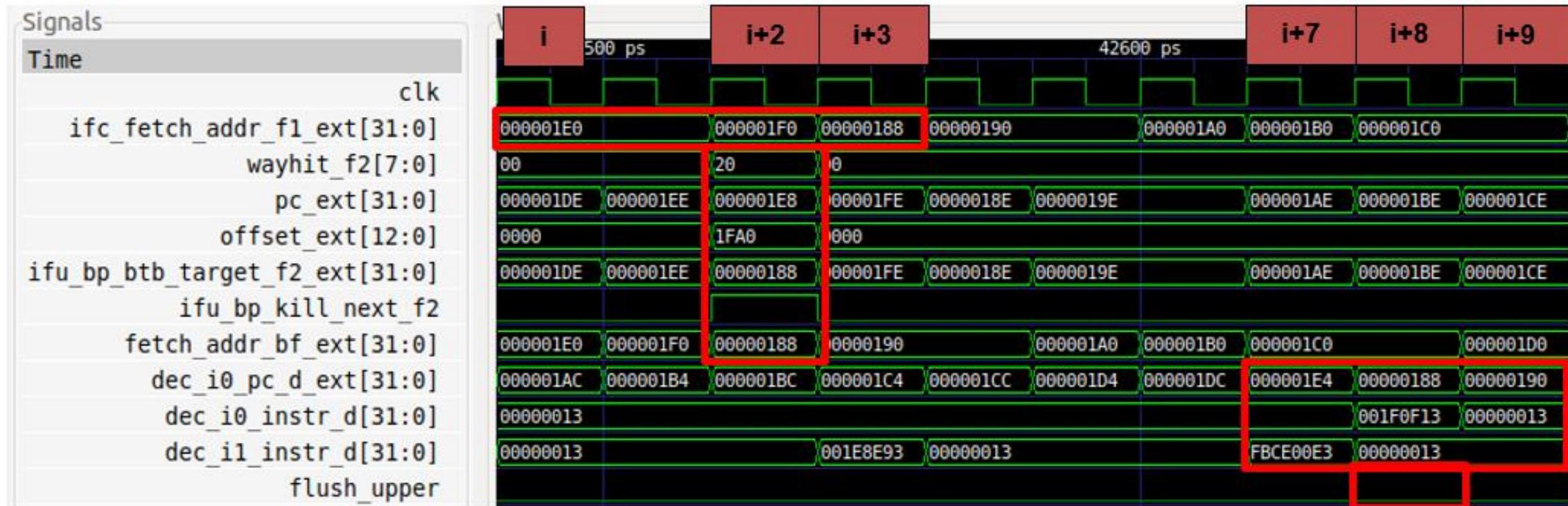
# RVfpga实验16：第二条beq指令的执行 – 分析

- **周期*i* - beq指令的译码阶段：**在通路0中对第二条beq指令（0xFBCE00E3）进行译码。生成流水线控制信号、读取寄存器文件并将分支指令传送至I<sub>0</sub>管道。信号a和b（本例中均为0xFFFF）中包含下一阶段所使用的比较器的输入。
- **周期*i+1* - beq指令的EX1阶段：**执行beq指令。比较信号a\_ff和b\_ff。两个数相等，因此发生分支。但是，简单BP将所有分支预测为不发生（i0\_ap.predict\_nt = 1）。因此，分支预测结果错误，必须清除取指的指令（flush\_upper = 1）。
- **周期*i+2* - FC1阶段：**指令必须在分支目标地址处继续执行。exu\_flush\_final = 1 and ifc\_fetch\_addr\_f1\_ext = exu\_flush\_path\_final\_ext = 0x00000188.该地址对应于分支目标地址，即循环的第一条指令的地址。

# RVfpga实验16: Gshare分支预测器



# RVfpga实验16：第二条beq指令的Gshare分支预测器



# RVfpga实验16：第二条beq指令的Gshare分支预测器

- **周期*i*:** 包含第二条分支的指令束的地址将提供给指令高速缓存: `ifc_fetch_addr_f1_ext = 0x000001E0`。系统使用该地址读取分支目标缓冲区（**BTB**）。
- **周期*i+2*:** 在**BTB**中发生了命中: `wayhit_f2 = 0x20`。将分支地址 (`pc_ext = 0x000001E8`) 与**BTB**提供的偏移量 (`offset_ext = 0x1FA0`, 该值为负值) 相加, 即可得到预测目标地址 (`ifu_bp_btb_target_f2_ext = 0x00000188`)。如果**BHT**预测将发生分支 (`ifu_bp_kill_next_f2 = 1`), 则预测目标地址将用作下次取指PC (`fetch_addr_bf_ext = 0x00000188`)。
- **周期*i+3*:** 取指地址为上一周期中计算的分支预测目标地址: `ifc_fetch_addr_f1_ext = 0x00000188`。
- **周期*i+7*:** 在通路1中对分支进行译码 (`dec_i1_instr_d = 0xFBCE00E3`)。
- **周期*i+8*:** 执行分支。预测正确, 因此无需触发清除操作 (`flush_upper = 0`)。
- **周期*i+9*:** 如果预测正确, 将通过分支目标地址正常继续执行分支。

# 实验17： 超标量执行

# RVfpga实验17：简介

- Western Digital的SweRV EH1处理器是32位内核，采用9级流水线和双路超标量设计。
- 超标量处理器包含数据路径硬件的多个副本，可同时执行多条指令。
- 执行单条指令的延时与标量处理器相同，但处理器可以在每个周期执行和提交更多的指令，从而提高吞吐量。

# RVfpga实验17：简介

- SweRV EH1是一个双路超标量处理器，
  - 每个周期最多可以取指、执行和提交两条指令。
  - 多端口寄存器文件最多读取四个源操作数，并在每个周期中写回两个值（加上一个来自非阻塞加载的值，如实验15的分析所示）。
  - 每个通路包含独立管道：两条整数管道、一条乘法管道、一条加载-存储管道和一个非流水线除法器。
- 理想情况下，双路超标量处理器的吞吐量（IPC）应为单发射处理器的两倍。遗憾的是，在实际的程序中，从单路处理器升级为双路处理器时，性能会提高到1.3至1.5倍；但添加第二条通路需要使用更多的硬件。
- 在本实验中，我们将分析两个简单的程序，比较使用SweRV EH1单发射和双发射配置时的行为。

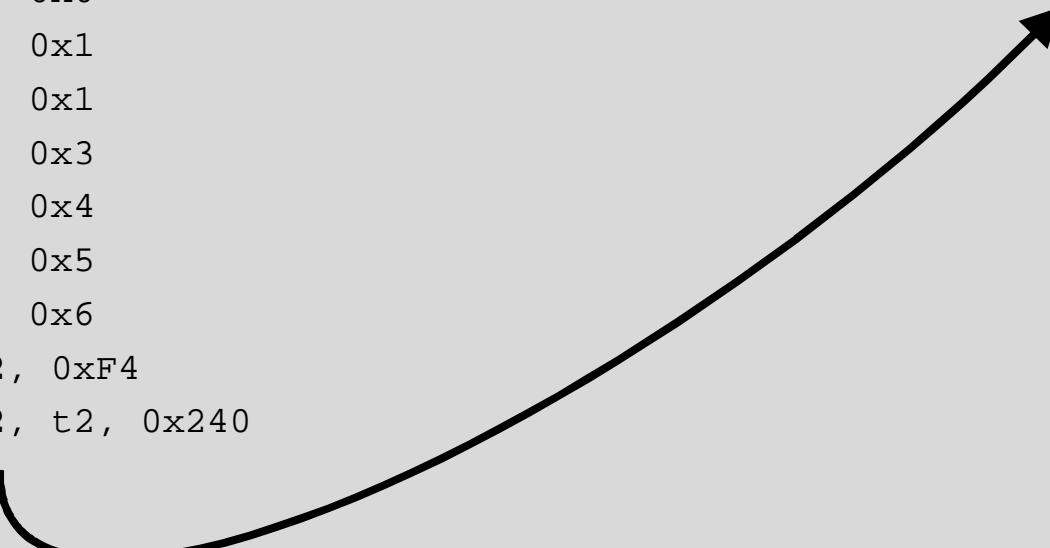
# RVfpga实验17：四条独立的A-L指令 - 示例 - 单发

```
.globl Test_Assembly

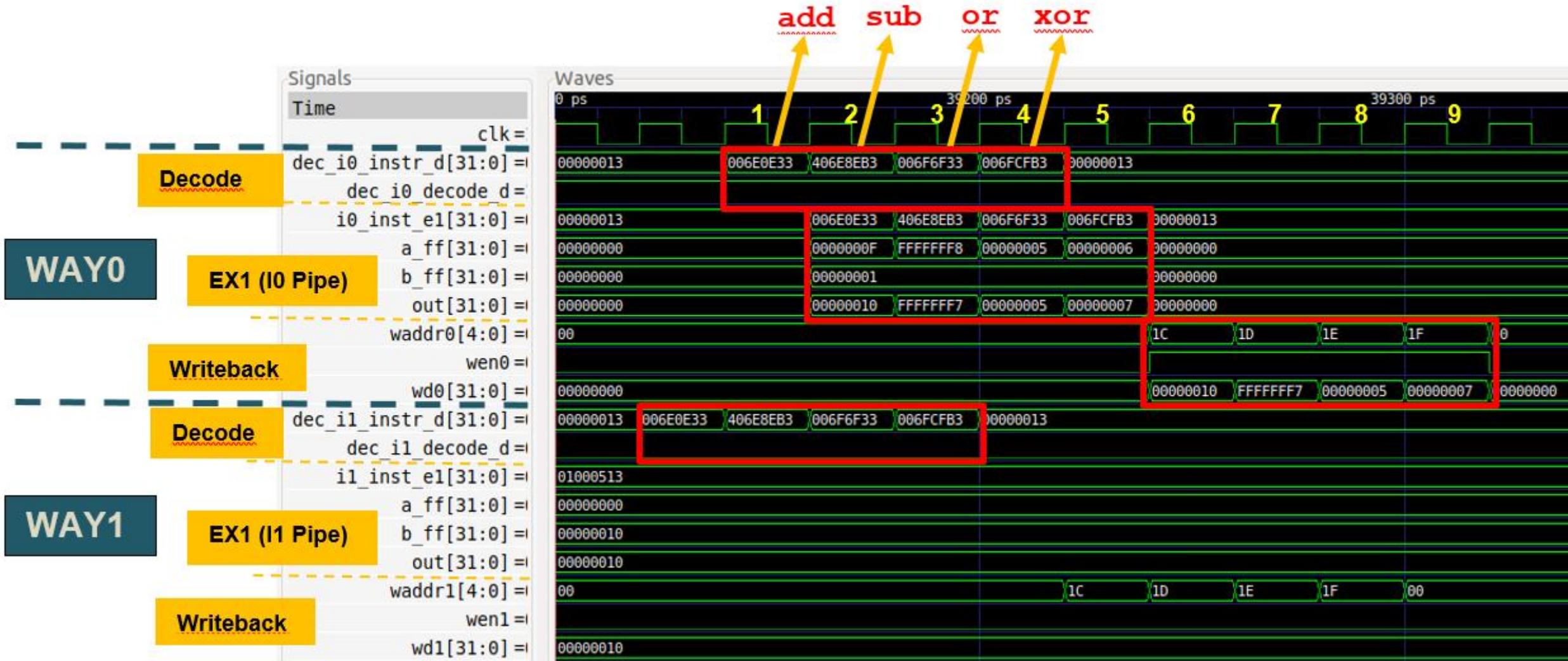
.text
Test_Assembly:

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
```

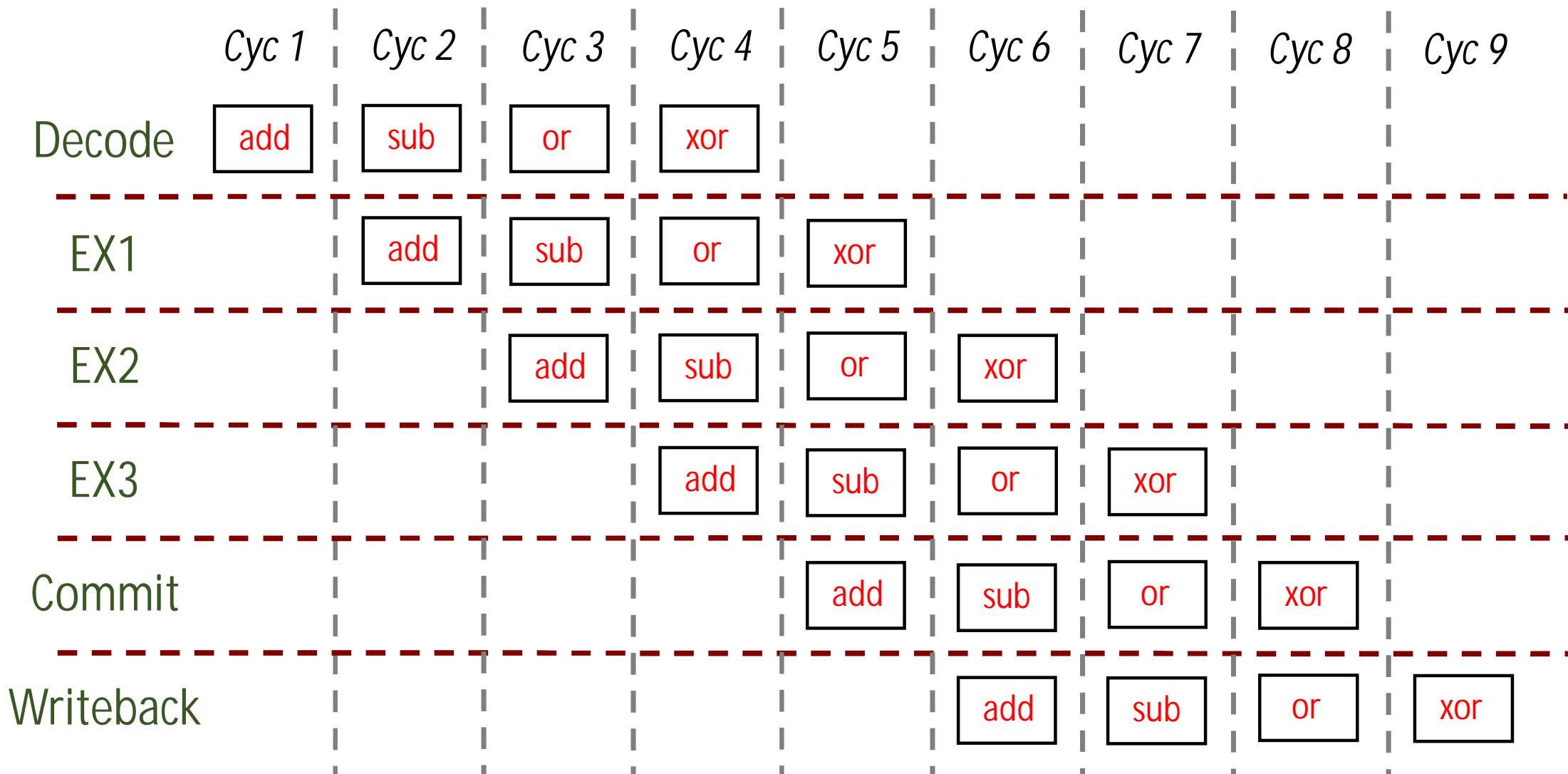
# RVfpga实验17：四条独立的A-L指令 - 仿真 - 单发



# RVfpga实验17：四条独立的A-L指令 - 仿真 - 单发

- 译码时，两条通路都会接收指令，但仅通路0会将指令发送至执行阶段，因为通路1的发送功能已禁用。
  - 通路0：
    - 在我们的示例中，信号dec\_i0\_decode\_d始终为1；具体地说，对于我们所分析的四条AL指令，其值均为1。
    - 译码阶段的指令会传播到I0管道（i0\_inst\_e1[31:0]）。
  - 通路1：
    - 在我们的示例中，信号dec\_i1\_decode\_d始终为0；具体地说，对于我们所分析的四条AL指令，其值均为0。
    - 译码阶段的指令不会传播（i1\_inst\_e1[31:0]）到执行阶段。
- 因此，系统仅使用来自I0管道的ALU（参见两条通路中的信号aff、bff和out），并且仅使用寄存器文件的写端口0（参见两条通路中的信号waddr、wen和wd）。

# RVfpga实验17：四条独立的A-L指令 - 图 - 单发



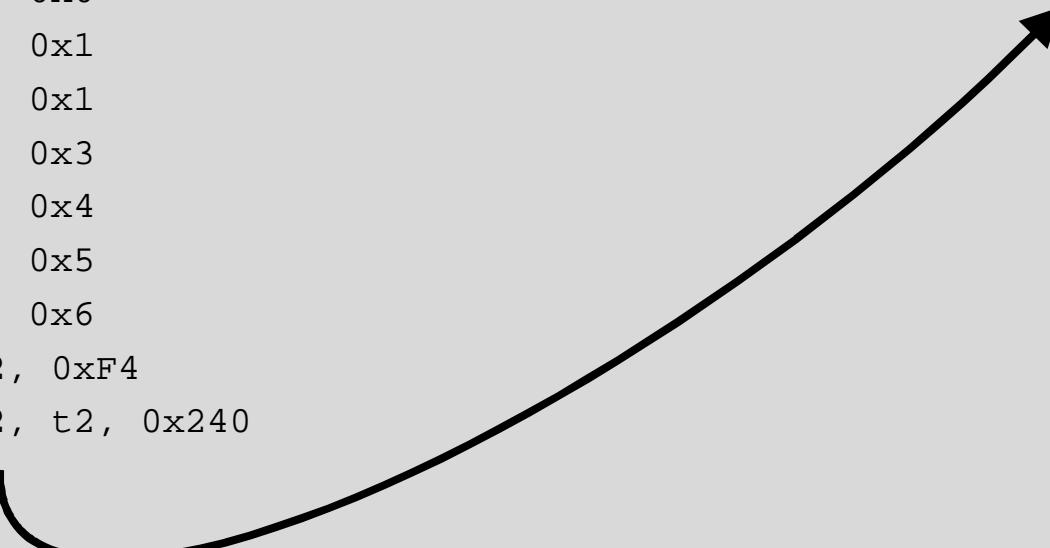
# RVfpga实验17：四条独立的A-L指令 - 示例 - 双发

```
.globl Test_Assembly

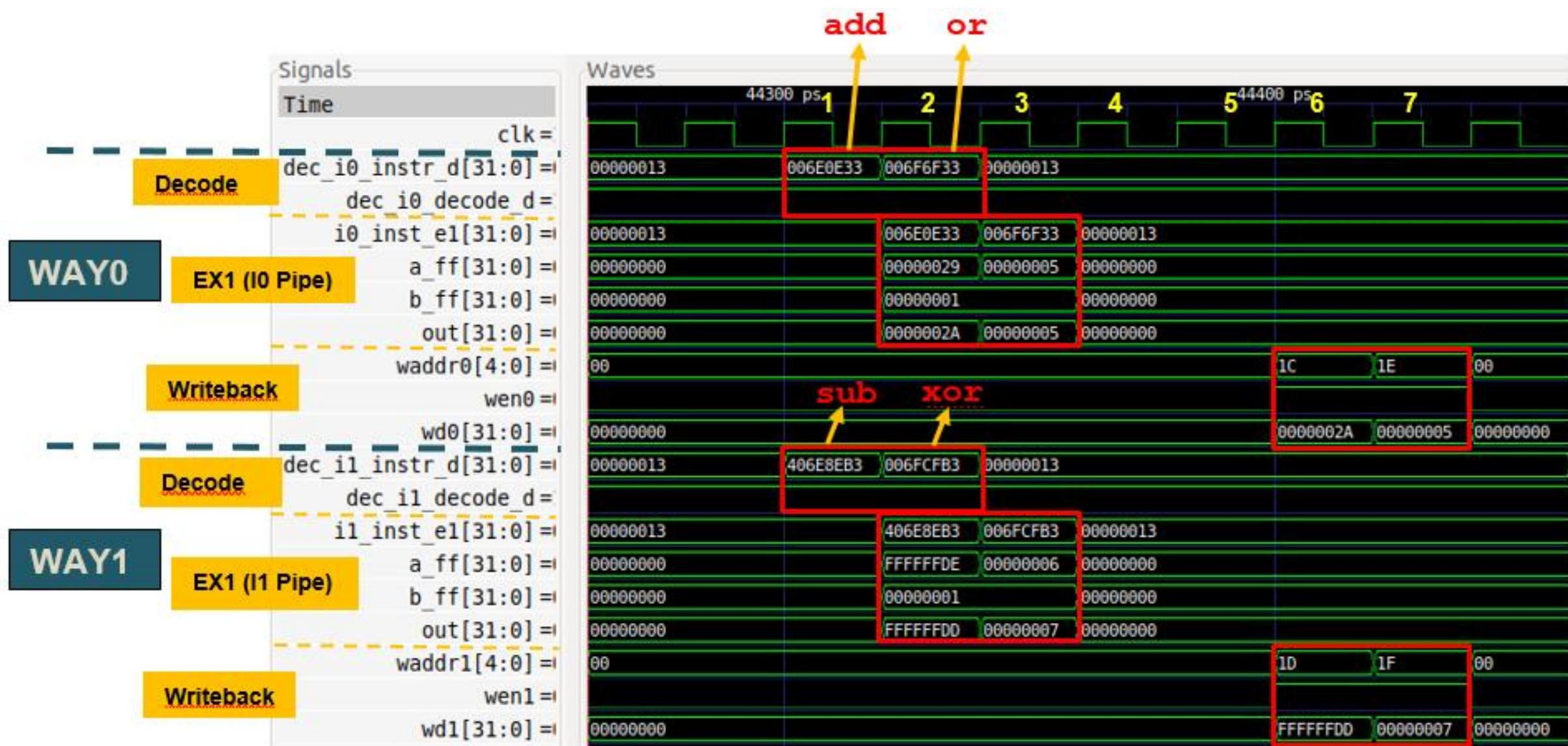
.text
Test_Assembly:

# 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
```

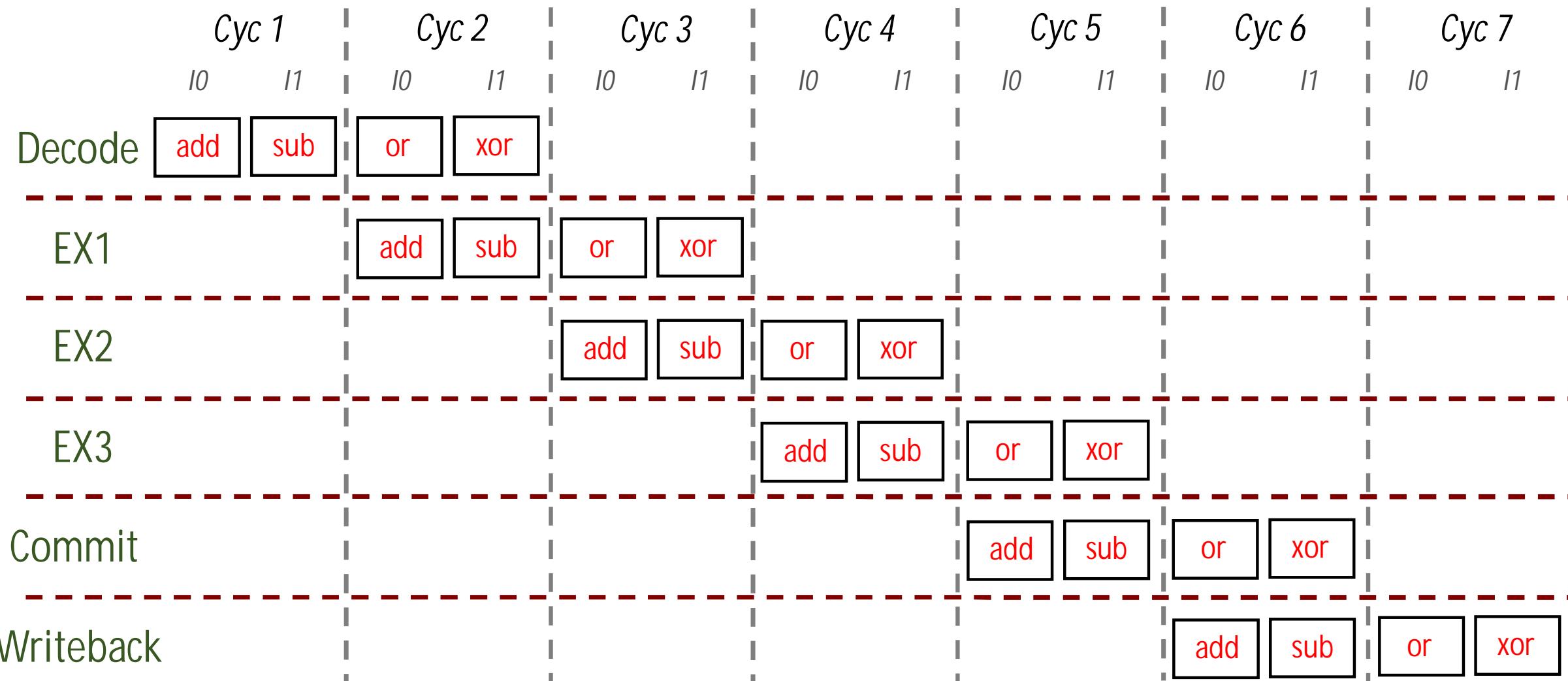
# RVfpga实验17：四条独立的A-L指令 - 仿真 - 双发



# RVfpga实验17：四条独立的A-L指令 – 分析 – 双发

- 在每个周期中，对两条指令进行译码（每个通路一条），并将两条指令发送到执行阶段（一条通过I0管道，另一条通过I1管道）。
  - 通路0：**
    - 对于示例中的四条A-L指令中的两条，信号dec\_i0\_decode\_d始终为1（另外两条在通路1中进行译码）。
    - 译码阶段的指令会传播到I0管道（i0\_inst\_e1[31:0]）。
  - 通路1：**
    - 对于示例中的四条A-L指令中的两条，信号dec\_i1\_decode\_d始终为1（另外两条在通路0中进行译码）。
    - 译码阶段的指令会传播到I1管道（i1\_inst\_e1[31:0]）。
- 因此，系统会使用两条管道（I0和I1）中的ALU（参见两条通路中的信号aff、bff和out），并且会使用两个寄存器文件写端口（参见两条通路中的信号waddr、wen和wd）。

# RVfpga实验17：四条独立的A-L指令 - 图 - 双发



# RVfpga实验17：互相交织的两条mul指令和两条A-L指令 – 示例 – 双发

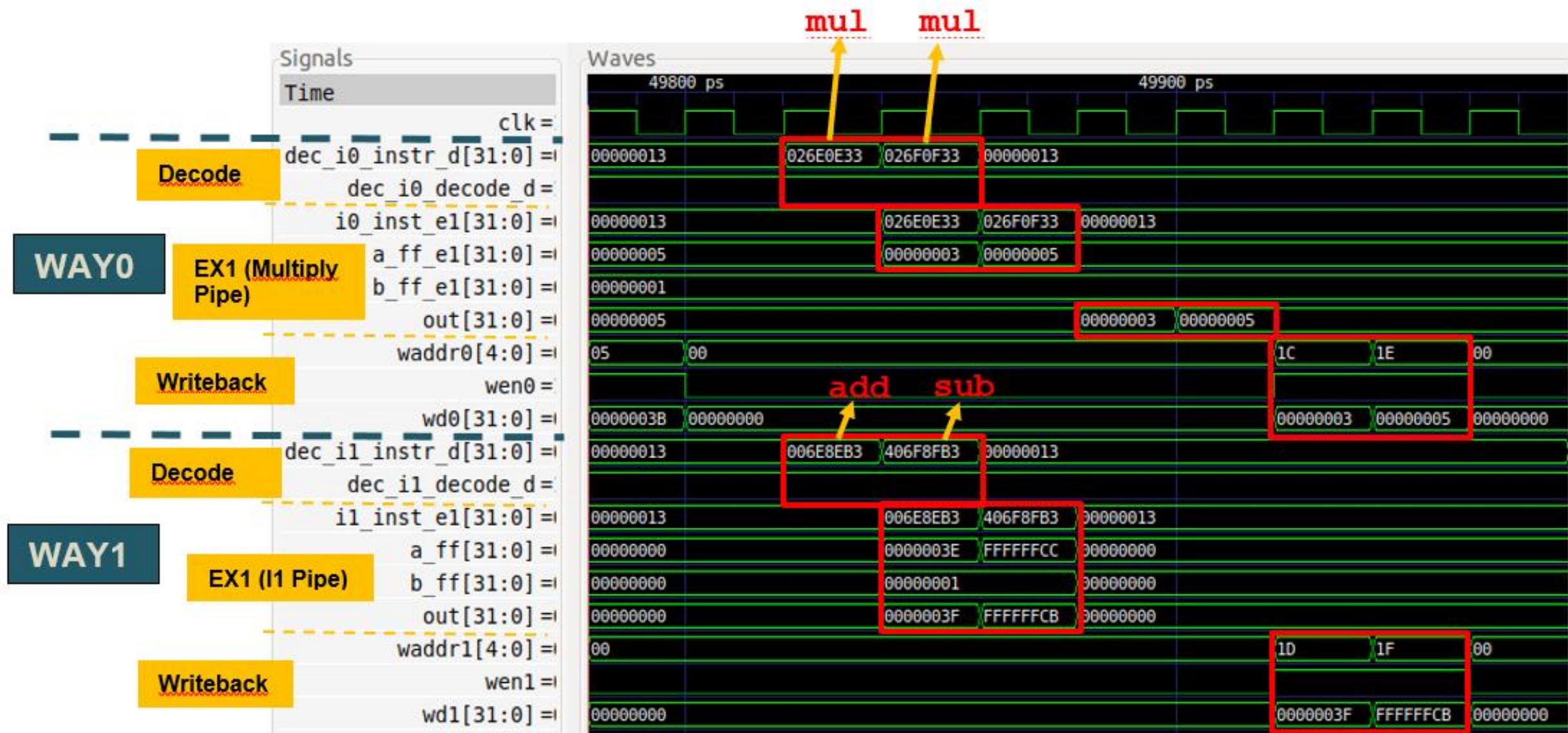
```
.globl Test_Assembly
.text
Test_Assembly:
# li t2, 0x400      # Disable Dual-Issue Execution
# csrrs t1, 0x7F9, t2

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实验17：互相交织的两条mul指令和两条A-L指令 – 仿真 – 双发



- 译码时，两条通路都会接收指令并将指令发送至执行阶段。

## — 通路0：

- 对于示例中所分析的四条指令中的两条，信号dec\_i0\_decode\_d始终为1（另外两条在通路1中进行译码）。
- 译码阶段的指令会发送到乘法管道（i0\_inst\_e1[31:0]）。

## — 通路1：

- 对于示例中所分析的四条指令中的两条，信号dec\_i1\_decode\_d始终为1（另外两条在通路1中进行译码）。
- 译码阶段的指令（dec\_i1\_instr\_d[31:0]）会传播到I1管道（i1\_inst\_e1[31:0]）
- 因此，系统会使用I1管道和乘法器中的ALU（参见信号a\_ff\_e1、b\_ff\_e1和out，以及信号a\_ff、b\_ff和out），并且会使用两个寄存器文件写端口（参见两条通路中的信号waddr、wen和wd）。

# 实验18： 添加新功能： 指令和计数器

# RVfpga实验18：简介

- 本实验将应用在先前实验中获得的知识来修改SweRV EH1处理器，向其添加以下新功能：
  - 添加**A-L**指令：通过RISC-V架构中提供的全新位操作扩展来添加算术逻辑指令。
  - 添加浮点指令：添加三条浮点指令：加、乘、除。然后使用这些指令进行二分法计算。
  - 添加硬件计数器：添加一个新的硬件计数器，用于计算执行的I型指令数量。
- 在一些练习中，我们将指导您完成修改内核的过程，在其他练习中，您必须自行确定所需的操作。您可以在实验文档中找到所有详细信息。

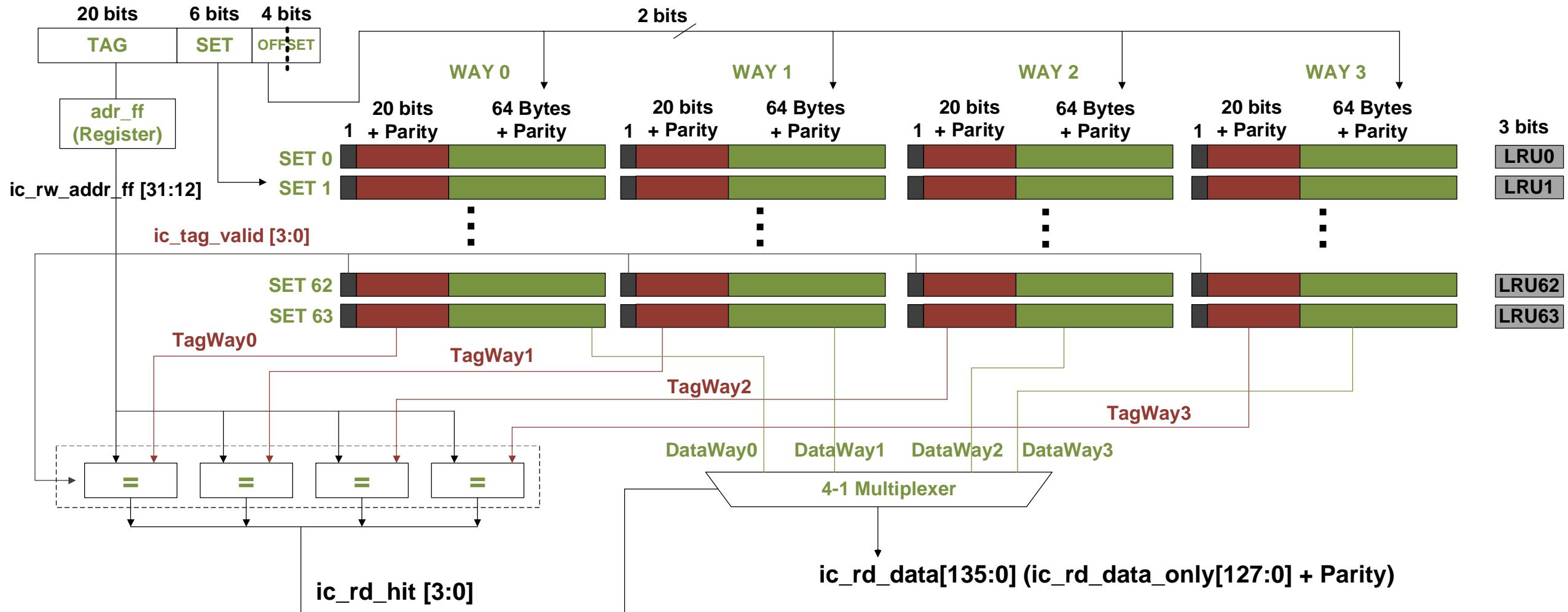
# 实验19： 指令高速缓存

# RVfpga实验19：简介

- 本实验将介绍并探究RVfpga系统的存储器系统。Rvfga的存储器系统具有以下元素：
  - 外部DDR主存储器
  - 指令高速缓存（I\$）
  - 两个便笺式存储器（也称为紧密耦合存储器），一个用于存储数据（DCCM），另一个用于存储指令（ICCM）。默认系统中会禁止ICCM。
- 本实验主要分析指令高速缓存（I\$）的操作。本实验将介绍I\$的配置方式，研究高速缓存未命中和命中的处理方式，并分析I\$替换策略。

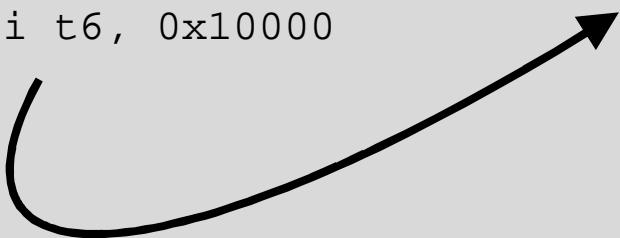
# RVfpga实验19: I\$配置和操作

ic\_rw\_addr [31:2] (Fetch Address)



# RVfpga实验19: I\$失效和命中管理 - 示例

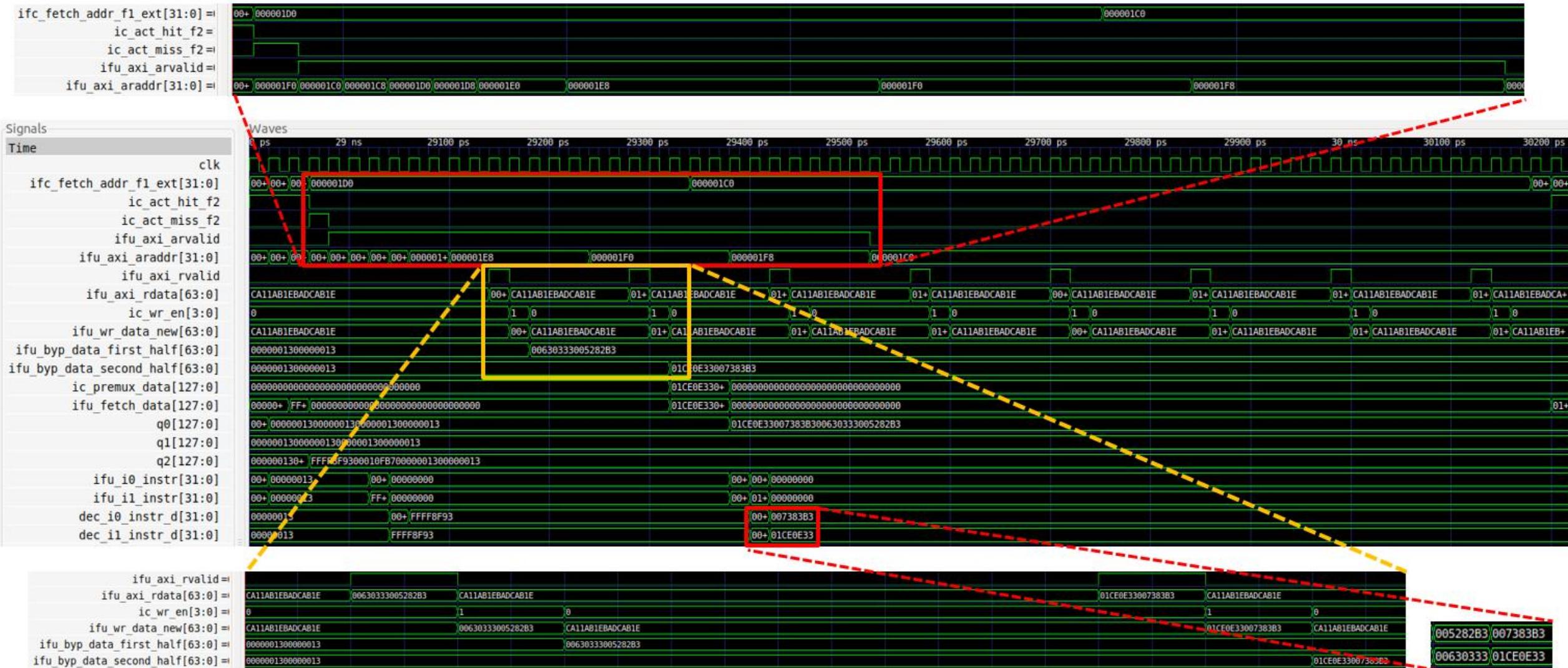
```
Test_Assembly:  
    INSERT_NOPS_3  
    INSERT_NOPS_8  
    INSERT_NOPS_8  
    li t6, 0x10000
```



```
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实验19: I\$失效管理 - 仿真



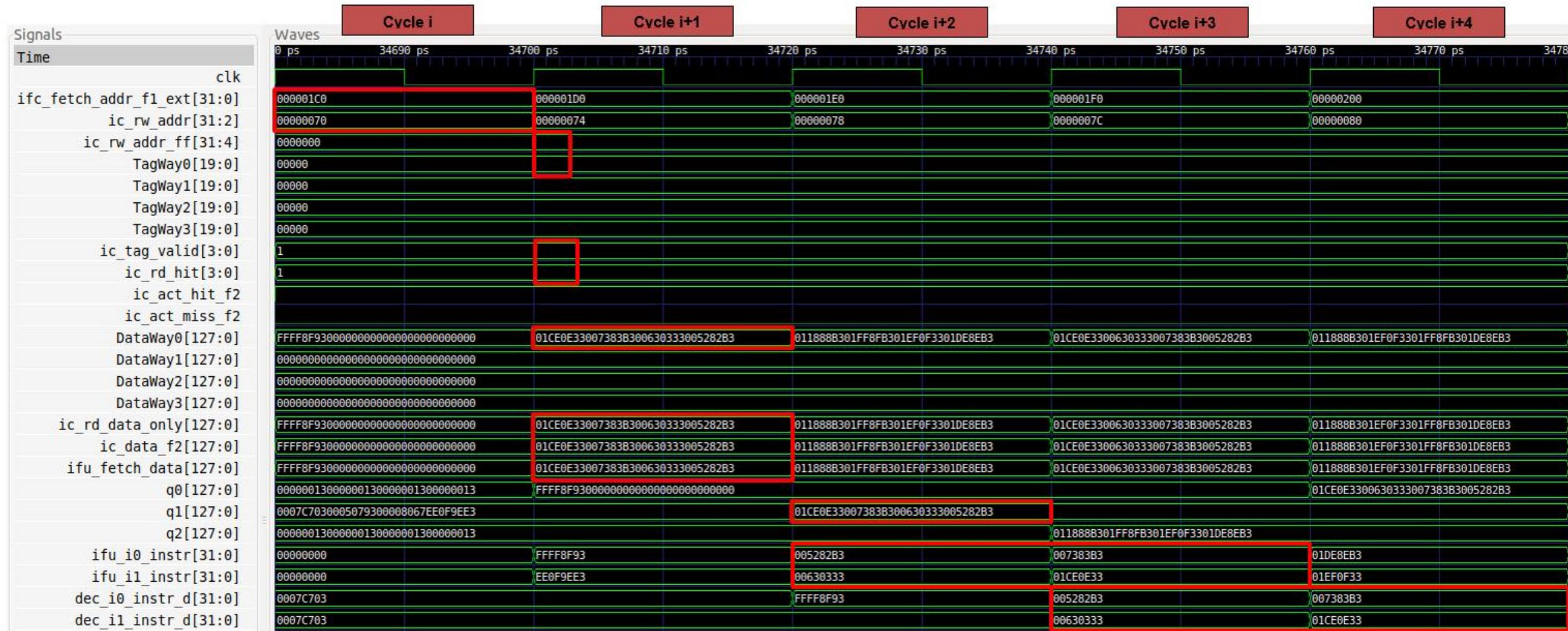
# RVfpga实验19: I\$失效管理 - 分析

- 仿真展示了首次执行16条add指令时的取指操作。如果这些指令不在I\$中，则I\$中将触发失效，必须将指令从DDR外部存储器复制到I\$中。
  - 每隔约29 ns便会出现一次I\$失效 (`ic_act_miss_f2 = 1`)，该事件将触发通过AXI总线发起的块请求 (`ifu_axi_arvalid = 1`)。
  - 系统将通过AXI总线依次请求构成目标块的八个64位块。
    - 信号`ifu_axi_arvalid`连续27个周期保持高电平。该信号表示通道正在发送有效的读地址和控制信息。
    - 在`ifu_axi_arvalid = 1`的这27个周期内，信号`ifu_axi_araddr`将通过AXI总线依次提供8个64位块的初始地址，这8个地址必须从DDR存储器读取。

# RVfpga实验19: I\$失效管理 - 分析

- 中间的图则展示了八个64位数据块通过信号ifu\_axi\_rdata中的AXI总线依次到达处理器。
  - 信号ifu\_axi\_rvalid用于指示通道正在发送所需的读数据，该信号每经7个周期便会保持一个周期的高电平。
  - 八个64位块（每个块包含两条指令）均由信号ifu\_axi\_rdata提供。
- 下面的两幅图显示，八个64位块在到达高速缓存控制器后均会立即写入I\$。
- 最终，可以看到上述四条指令从I\$控制器旁路到流水线，以便在I\$失效后尽快重新启动执行。几个周期后，四条指令将到达译码阶段。

# RVfpga实验19: I\$命中管理 - 仿真



# RVfpga实验19: I\$命中管理 - 分析

- 在之前的仿真中，可以看到I\$中出现命中。
  - 周期i:** 第一条add指令 (add t0,t0,t0) 的地址由信号ifc\_fetch\_addr\_f1\_ext提供。该信号将传递到I\$, 但需要去掉两个最低有效位，因为指令以4字节（32位）边界对齐。因此，ic\_rw\_addr = 0x00000070。标记数组和数据数组使用取指地址的子集。
  - 周期i+1:** 四个标记（每个高速缓存通路一个）位于信号TagWay0-TagWay3中。这些标记将与取指地址的TAG字段进行比较。在本例中，所有标记均与TAG字段相同，但只有一条通路（通路0）有效 (ic\_tag\_valid = 0001)，因此将在通路0中发出命中消息：ic\_rd\_hit = 0001。四个128位指令束同样位于信号DataWay0-DataWay3中：ic\_rd\_data\_only = 0x01ce0e33007383b300630333005282b3
  - 周期i+2:** 在对齐阶段，从缓冲区q1中提取第一条和第二条add指令：ifu\_i0\_instr = 0x005282b3且 ifu\_i1\_instr = 0x00630333
  - 周期i+3:** 在对齐阶段提取第三条和第四条add指令，同时对第一条和第二条add指令进行译码：ifu\_i0\_instr = 0x007383b3、ifu\_i1\_instr = 0x01ce0e33、dec\_i0\_instr\_d = 0x005282b3且 dec\_i1\_instr\_d = 0x00630333
  - 周期i+4:** 最后，对第三条和第四条add指令进行译码：dec\_i0\_instr\_d = 0x007383b3且 dec\_i1\_instr\_d = 0x01ce0e33

# RVfpga实验19：I\$替换策略

- 大多数组相连高速缓存采用最近最少使用（Least Recently Used, LRU）的替换策略。然而，跟踪最近最少使用的通路较为复杂，因此一般使用简化的LRU策略（通常称为伪LRU），该策略已足够满足实际需求。
- SweRV EH1使用名为二进制树伪**LRU**的简化策略。
  - 要实现该策略，N路组相连高速缓存中的每组需要有N-1个位（称为LRU状态）。在SweRV EH1的I\$中则为每组3位。

块替换

LRU状态	替换的通路
x00	通路0
x10	通路1
0x1	通路2
1x1	通路3

LRU状态更新

写入的通路	下一LRU状态
通路0	-11
通路1	-01
通路2	1-0
通路3	0-0

# RVfpga实验19: I\$替换策略 - 示例

- 下面的示例在一个无限循环中访问五个不同的I\$块，这五个块均映射到同一I\$组：**SET = 8**。
- 无限循环包含五条j（跳转）指令，其中每对j指令由1023个nop分隔。j指令与nop共占用4 KiB空间，等同于I\$中每条通路的大小。

```
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
```

## RVfpga实验19: L\$替换策略 - L\$组8的变化

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

Valid	Tag	Data	
1	00000000000000000000000000000000	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	00000000000000000000000000000000	j_Set8_Block2   nop   ...   nop	WAY 0
1	00000000000000000000000000000001	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	00000000000000000000000000000000	j_Set8_Block2   nop   ...   nop	WAY 0
1	00000000000000000000000000000001	j_Set8_Block3   nop   ...   nop	WAY 1
1	00000000000000000000000000000010	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	00000000000000000000000000000000	j_Set8_Block2   nop   ...   nop	WAY 0
1	00000000000000000000000000000001	j_Set8_Block3   nop   ...   nop	WAY 1
1	00000000000000000000000000000010	j_Set8_Block4   nop   ...   nop	WAY 2
1	00000000000000000000000000000011	j_Set8_Block5   nop   ...   nop	WAY 3

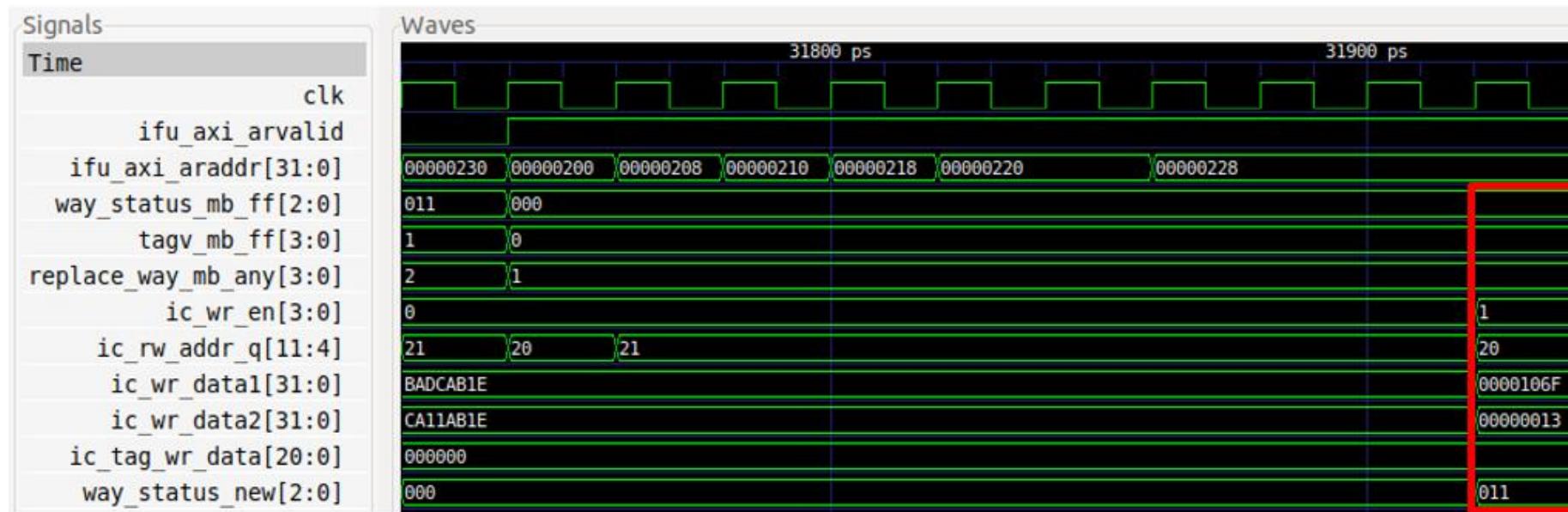
LRU STATE = 000

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

1	00000000000000000000000000000010	j_Set8_Block1   nop   ...   nop	WAY 0
1	00000000000000000000000000000001	j_Set8_Block3   nop   ...   nop	WAY 1
1	00000000000000000000000000000010	j_Set8_Block4   nop   ...   nop	WAY 2
1	00000000000000000000000000000011	j_Set8_Block5   nop   ...   nop	WAY 3

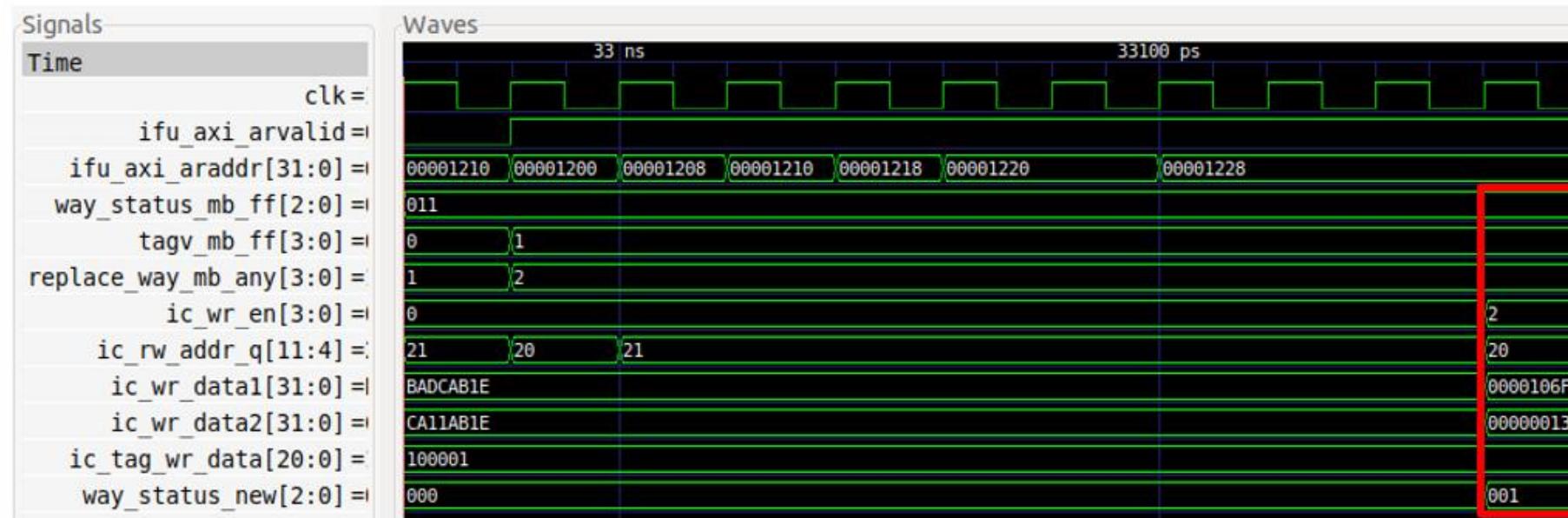
LRU STATE = 011

# RVfpga实验19: I\$替换策略 – 第1次跳转



- 第一次跳转的地址（0x200）映射到I\$的组8。该组的初始状态为空，因此必须将新块写入通路0。 $\text{replace\_way\_mb\_any} = \text{ic\_wr\_en} = 0001$ 。组8的LRU状态更新如下： $\text{way\_status\_new} = 011$ 。
- I\$块从DDR存储器中读取，并以64位块的形式写入I\$。上图展示了将新块的标记和前两条指令写入组8的过程：
  - $\text{ic\_rw\_addr\_q[11:4]} = 00100000$  (组8)
  - $\text{ic\_tag\_wr\_data[19:0]} = 0x0$
  - $\text{ic\_wr\_data1[31:0]} = 0x0000106F$  ( $j \text{ Set8_Block2}$ )
  - $\text{ic\_wr\_data2[31:0]} = 0x00000013$  (nop)

# RVfpga实验19: I\$替换策略 – 第2次跳转



- 第二次跳转的地址（0x1200）同样映射到I\$的组8。在该组中，仅通路0有效:  $\text{tagv\_mb\_ff} = 0001$ 。因此，必须将新块写入通路1:  $\text{replace\_way\_mb\_any} = \text{ic\_wr\_en} = 0010$ .组8的LRU状态更新如下:  $\text{way\_status\_new} = 001$ 。
- I\$块从DDR存储器中读取，并以64位块的形式写入I\$。上图展示了将新块的标记和前两条指令写入组8的过程：
  - $\text{ic\_rw\_addr\_q[11:4]} = 00100000$  (组8)
  - $\text{ic\_tag\_wr\_data[19:0]} = 0x1$
  - $\text{ic\_wr\_data1[31:0]} = 0x0000106F$  (j Set8\_Block3)
  - $\text{ic\_wr\_data2[31:0]} = 0x00000013$  (nop)

# RVfpga实验19: I\$替换策略 – 第5次跳转



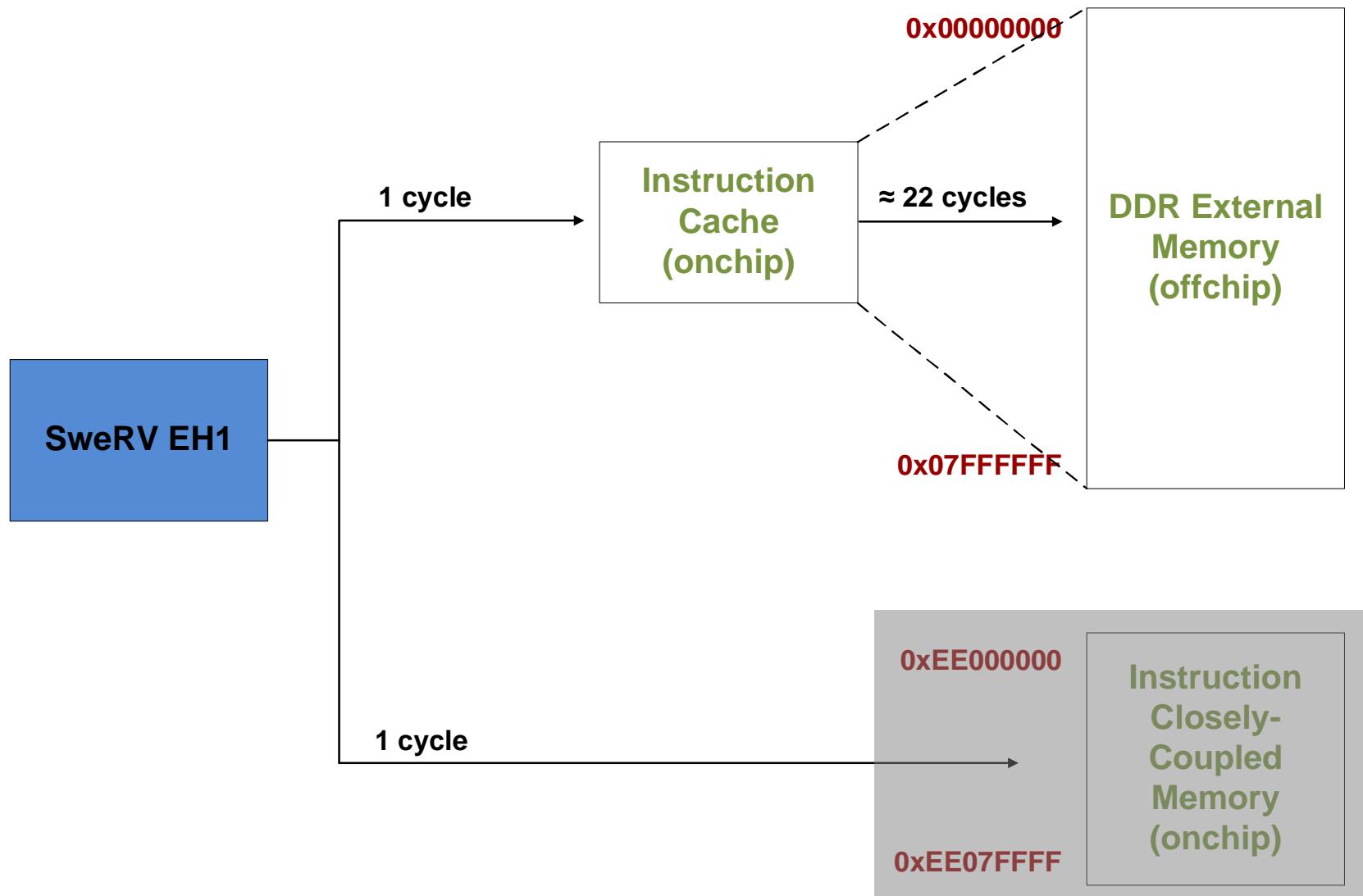
- 第五次跳转的地址（0x4200）同样映射到I\$的组8。但是，此时组已满:  $\text{tagv\_mb\_ff} = 1111$ 。因此，必须将新块写入通路1:  $\text{replace\_way\_mb\_any} = \text{ic\_wr\_en} = 0001$ 。组8的LRU状态更新如下:  $\text{way\_status\_new} = 011$ 。
- I\$块从DDR存储器中读取，并以64位块的形式写入I\$。上图展示了将新块的标记和前两条指令写入组8的过程：
  - $\text{ic\_rw\_addr\_q[11:4]} = 00100000$  (组8)
  - $\text{ic\_tag\_wr\_data[19:0]} = 0x4$
  - $\text{ic\_wr\_data1[31:0]} = 0x800fc06f$  (j Set8\_Block1)
  - $\text{ic\_wr\_data2[31:0]} = 0x00008067$  (ret)

# 实验20： ICCM、DCCM和 基准测试

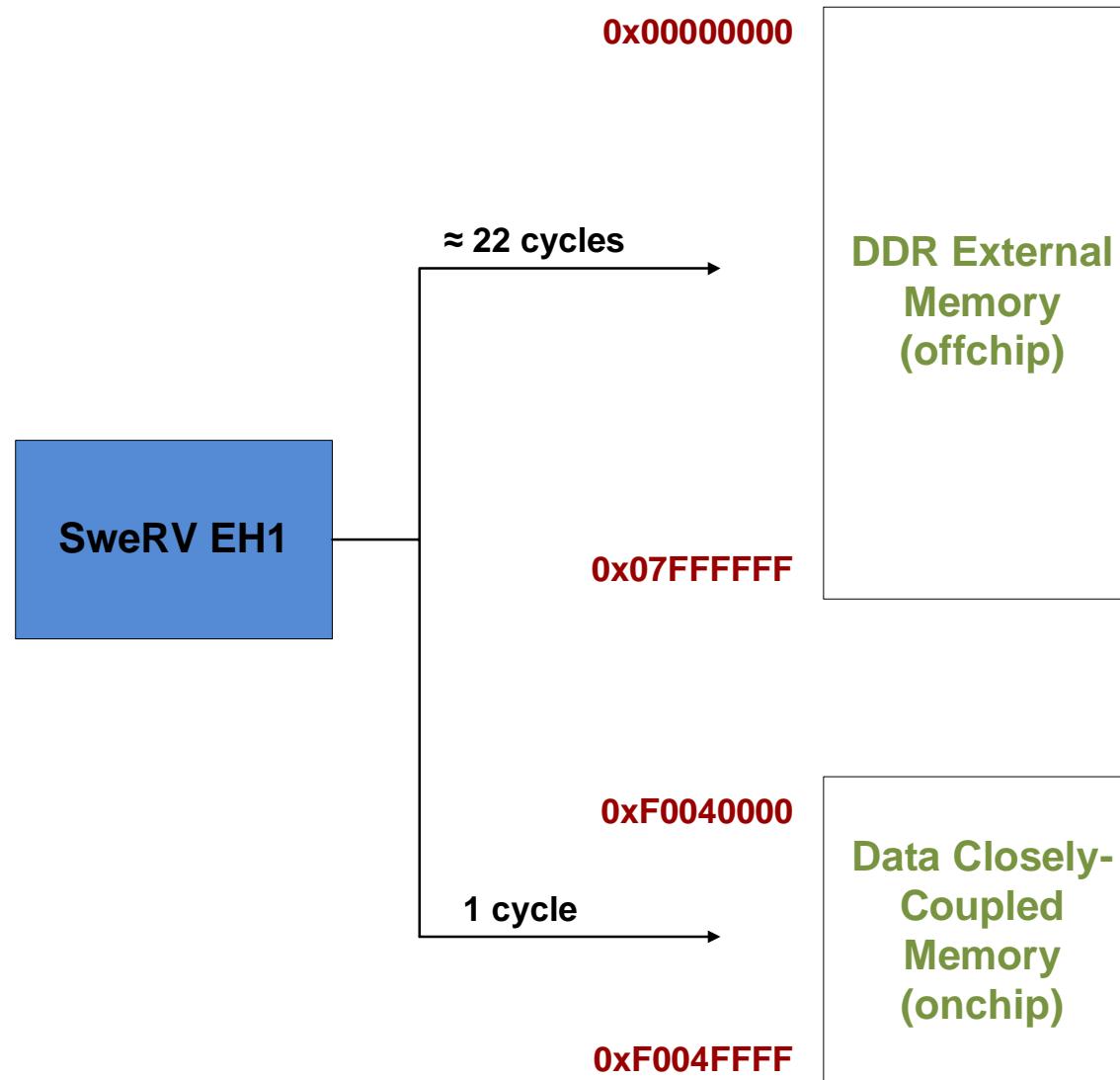
# RVfpga实验20：简介

- 本实验将分析SweRV EH1处理器中提供的便笺式存储器（ICCM和DCCM），然后会提供几个基准测试示例和练习，以演示实验11-20中的一些概念。
- 回顾一下，RVfpga系统包括两个便笺式存储器：
  - 一个用于存储数据，称为数据紧密耦合存储器（Data Closely-Coupled Memory, DCCM）
  - 一种用于存储指令，称为指令紧密耦合存储器（Instruction Closely-Coupled Memory, ICCM）

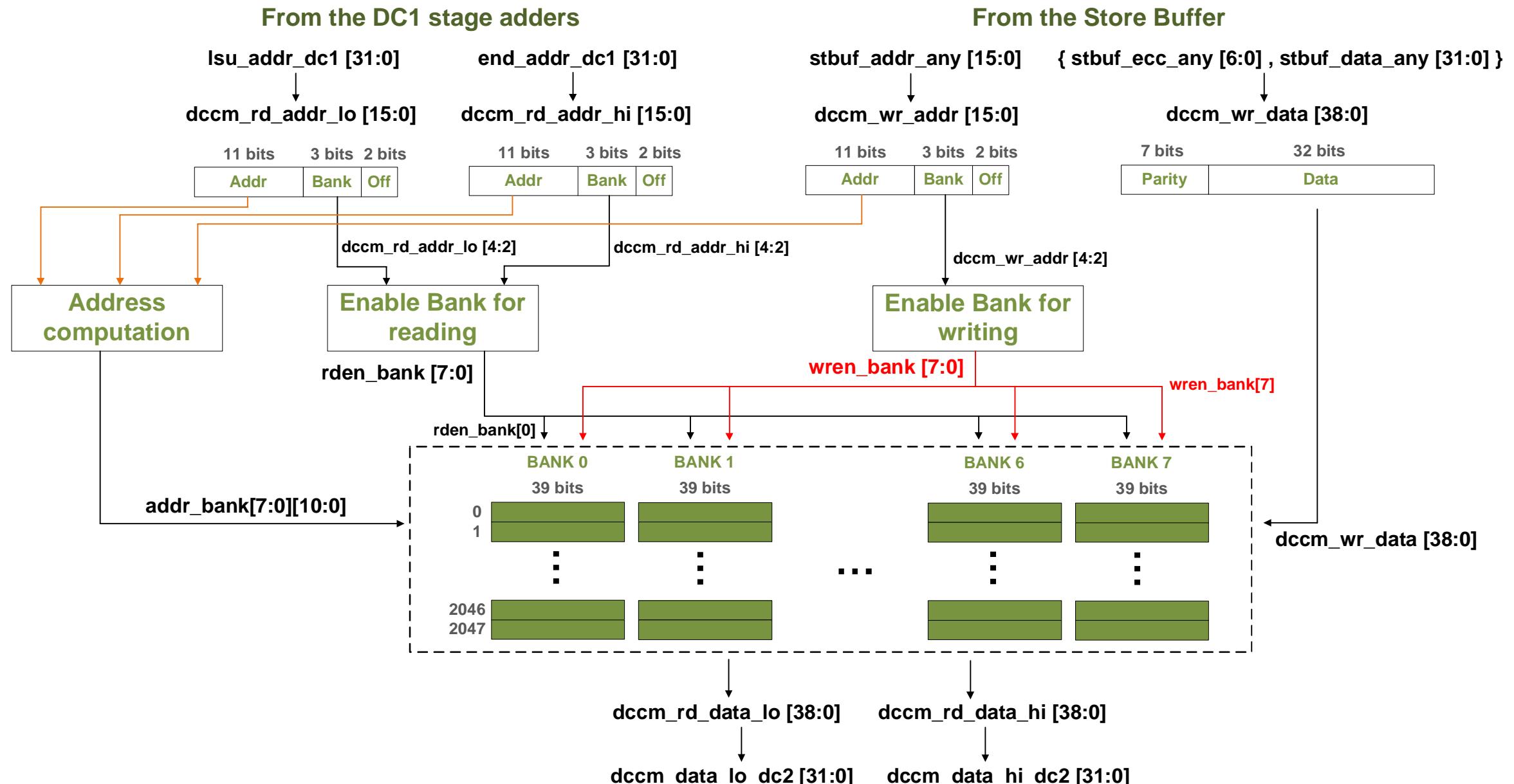
# RVfpga实验20：指令存储器 – 地址空间



# RVfpga实验20：数据存储器 – 地址空间



# RVfpga实验20: DCCM配置和操作

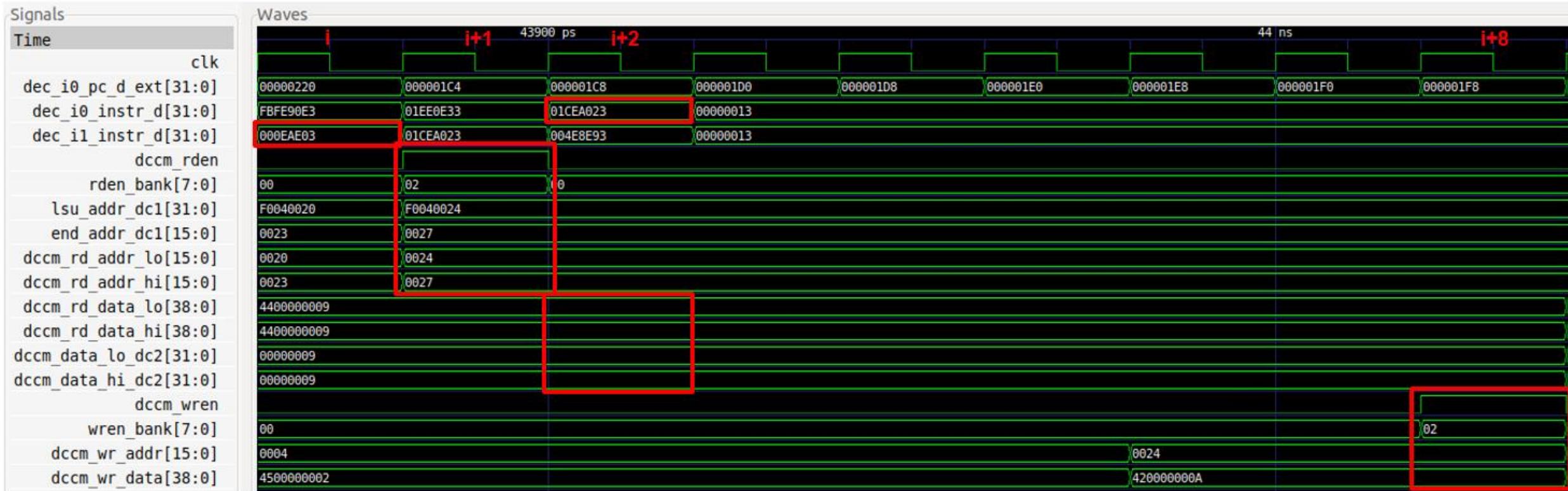


# RVfpga实验20: DCCM – 示例

```
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实验20: DCCM – 仿真



# RVfpga实验20: DCCM – 分析

- **周期i:** 在通路1中对lw指令进行译码: `dec_i1_instr_d = 0x000eae03`。
- **周期i+1:** 在DC1阶段生成地址, 然后将地址提供给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`
- 完成地址检查后, 将使能DCCM的读操作: `dccm_rden = 1`。由于访问是字对齐的, 因此仅读取第二个存储区: `rden_bank = 0x02` (二进制值为00000010)。
- **周期i+2:** 从DCCM获取读数据, 并将其提供给内核:
  - `dccm_rd_data_lo = 0x4400000009` → `dccm_data_lo_dc2 = 0x00000009`
  - `dccm_rd_data_hi = 0x4400000009` → `dccm_data_hi_dc2 = 0x00000009`
- **周期i+8:** 将读取值加1的结果 ( $0x00000009 + 1 = 0x0000000A$ ) 写入DCCM:
  - `dccm_wren = 1`
  - `wren_bank = 0x02` (二进制值为00000010; 即第二个存储区)
  - `dccm_wr_addr = 0x0024`
  - `dccm_wr_data = 0x420000000A`

# RVfpga实验20：基准测试

- 如需对处理器进行基准测试，应运行程序（或程序集）并测定处理器性能。通过在多个处理器上运行相同的基准，可对这些处理器进行比较。
- 本实验引入了两种常用的基准：**CoreMark**和**Dhystone**。我们使用Chips Alliance提供的源代码（<https://github.com/chipsalliance/Cores-SweRV>）对它们进行了修改，使其能够适用于RVfpga系统。在练习中，我们还将使用实验5中的图像处理应用程序作为基准。
- 对于任何基准，硬件计数器都将测量各种处理器事件。除了修改基准以便使用RISC-V硬件计数器外，我们还添加了一些对使用DCCM/ICCM和编译器优化的支持。
- 接下来，我们将介绍改变存储器配置和编译器优化时的**CoreMark**性能。

# RVfpga实验20：指标

- **CoreMark指标**
  - CoreMark运行循环的多次迭代。
  - **CoreMark分数（CM）**：每秒钟完成的迭代次数（即，迭代数/秒）。
  - **CM/MHz**： CM除以单位为MHz的时钟频率（也称为Iterat/Sec/MHz，即迭代数/秒/MHz）。
- 回顾一下，由于Swerv EH1为双路超标量处理器，因此其**理想IPC为2**。

# RVfpga实验20：各种条件下的CoreMark

	编译器 = 调试 外部存储器	编译器 = 调试 DCCM	编译器 = 优化 DCCM
CM/MHz	0.47	1.88	3.47
指令数	约50万	约50万	30.9万
周期数	约200万	约50万	28.8万
IPC (指令数/周期)	0.25	约为1	约为1
数据总线事务	约133,000 (全部转到外部 存储器)	0 (由于DCCM)	0 (由于DCCM)
指令总线事务	392 (由于I\$)	392 (由于I\$)	392 (由于I\$)