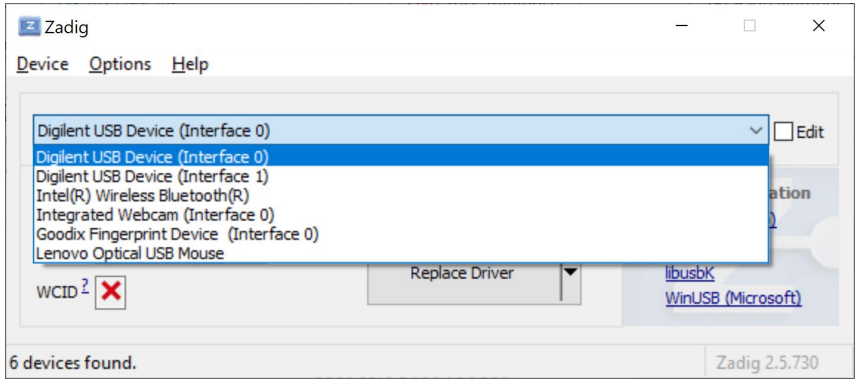
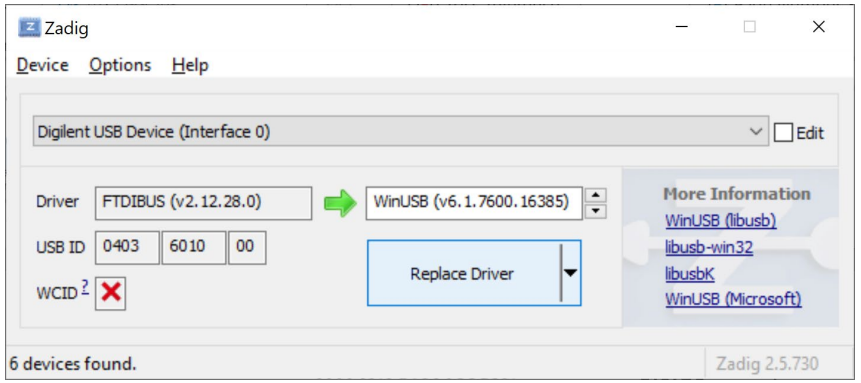
RVfpga

了解计算机架构的完整课程

安装软件工具

- 安装**VSCode**和**PlatformIO**
  - https://code.visualstudio.com/Download
  - 在VSCode中安装PlatformIO扩展功能
- 安装**GTKWave**
  - https://sourceforge.net/projects/gtkwave/files/
  - ***gtkwave-3.3.100-bin-win64.zip***
- 安装电路板驱动
  - Windows使用zadig软件
  - Linux使用提供*driversLinux_NexysA7*文件夹下脚本





Workshop日程安排

上午	
9:00-9:10	Imagination介绍
9:10-9:40	RVfpga课程介绍
9:40-10:30	实验1：RISC-V编程
10:30-10:50	茶歇
10:50-11:00	Digilent介绍
11:00-11:10	RIOS Lab介绍
11:10-12:00	实验2：VeeR EH1流水线

下午	
12:00-14:00	午餐休息
14:00-14:50	实验3：I/O简介
14:50-15:30	实验4：VeeR EH1数据冒险
15:30-15:50	茶歇
15:50-16:40	实验5：硬件计数器与基准测试
16:40-17:00	讨论 Q&A

目录

- RVfpga课程介绍
- 实验1：RISC-V编程
- 实验2：VeeR EH1流水线
- 实验3：I/O简介与中断驱动I/O
- 实验4：VeeR EH1数据冒险
- 实验5：硬件计数器与基准测试
- 讨论 Q&A

RVfpga简介

- RISC-V FPGA（**RVfpga**）是一套教学包，其中包含一整套说明、工具和实验，用于展示如何：
  - 编程RISC-V SoC
  - 向RISC-V SoC添加更多功能
  - 分析和修改RISC-V内核与存储结构
- 该教学包由**Imagination Technologies**与其学术和行业合作伙伴共同开发
- RVfpga围绕Chips Alliance的**VeeRwolf SoC**（基于Western Digital的RISC-V **VeeR EH1**内核）构建
- https://university.imgtec.com/teaching-download/

RVfpga内容

- 入门指南
  - RISC-V架构和RVfpga概述
  - 安装工具
  - 通过硬件和仿真运行RVfpga
- 实验
  - **1-10：**
    - RISC-V编程
    - Vivado项目
    - I/O系统
  - **11-20：**
    - RISC-V流水线
    - RISC-V存储系统
    - 基准测试

RVfpga既指课程名称，也指以FPGA为目标的RISC-V SoC。

所有实验都包括了使用和修改RVfpga系统的练习，通过动手设计帮助更好的理解，同时部分练习也提供了解答。

RVfpga实验

第1部分

编号	标题
1	C语言编程
2	RISC-V汇编语言
3	函数调用
4	图像处理
5	创建Vivado项目
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	VeeR EH1配置、结构和性能监视
12	算术/逻辑指令：add指令
13	存储指令：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）
Verilator和**GTKwave**

硬件*

Digilent的**Nexys A7**/Nexys 4 DDR FPGA电路板

*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**VeeR EH1**

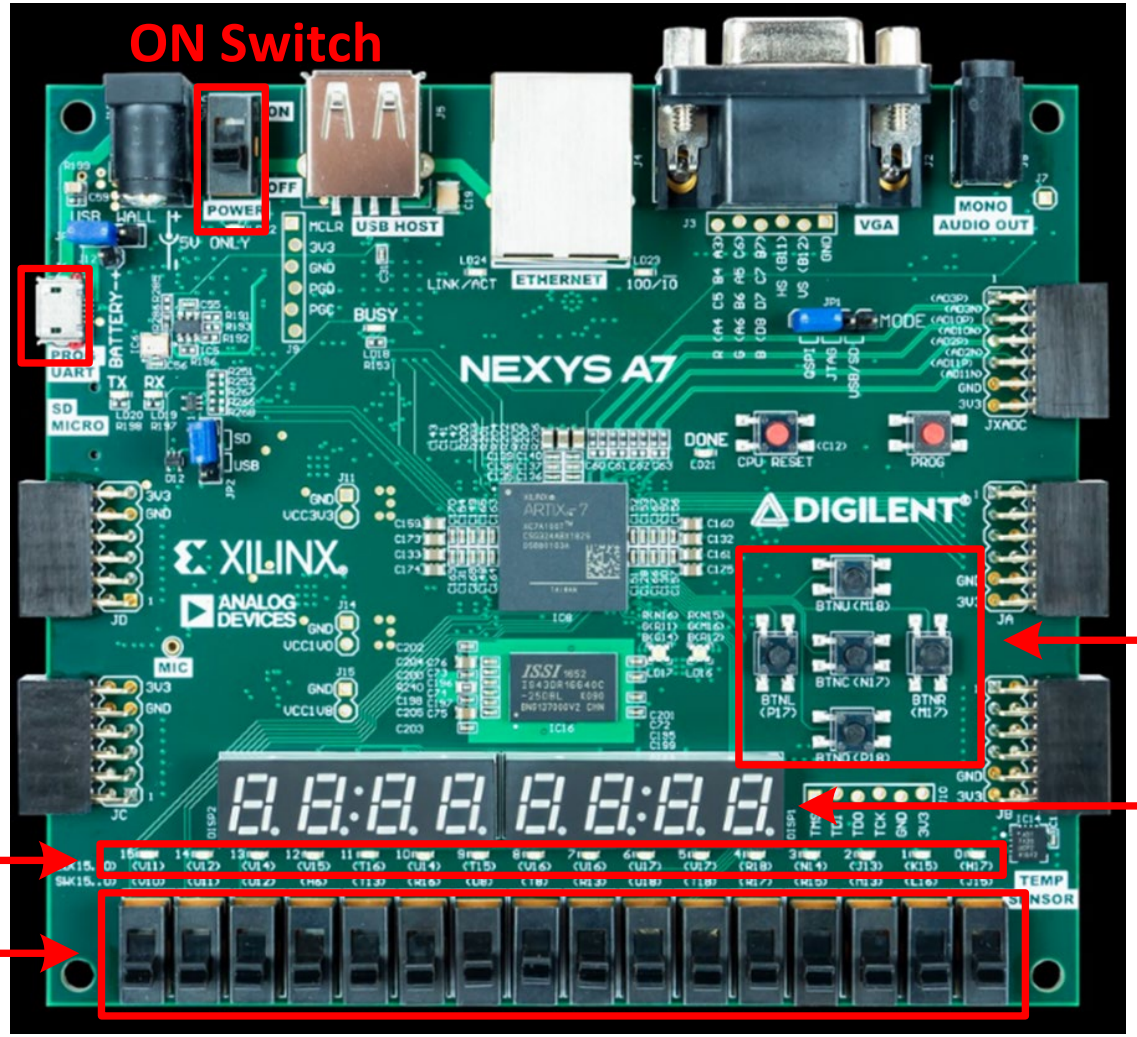
SoC： Chips Alliance的**VeeRwolf**

除FPGA电路板外，所有软硬件均可免费使用

RVfpga软件工具

- **Xilinx的Vivado IDE**
  - 查看RVfpga源文件（Verilog/SystemVerilog）和层级
  - 为以Nexys A7电路板为目标的RVfpga创建bit文件
- **PlatformIO** – Visual Studio Code（VSCode）的扩展
  - 将RVfpga下载到Nexys A7电路板
  - 在RVfpga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级别仿真RVfpga，以分析RVfpga的内部信号

RVfpga硬件电路板



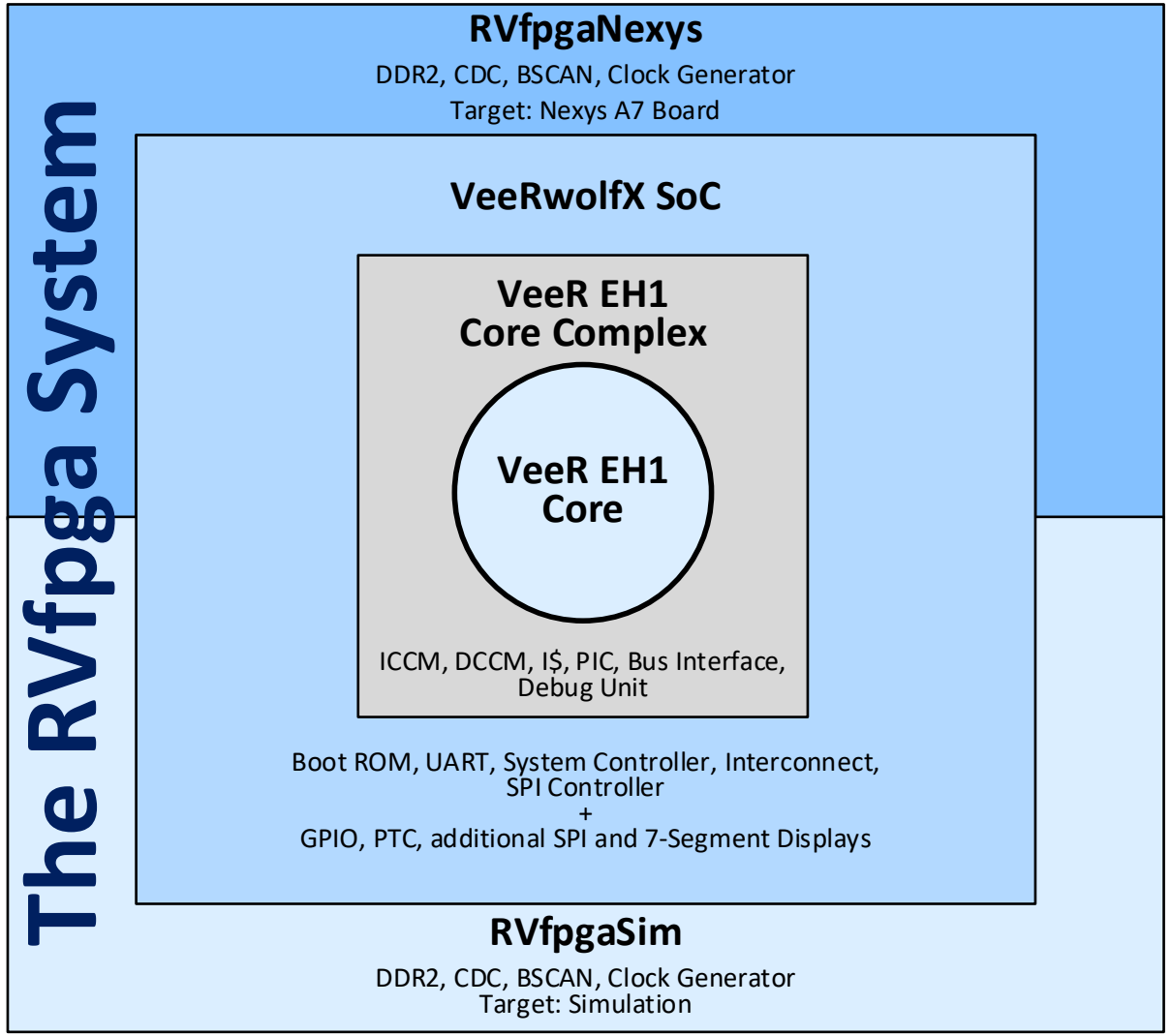
电路板图片来源：https://reference.digilentinc.com/

- **Nexys A7-100T FPGA**
  - 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）

支持的平台

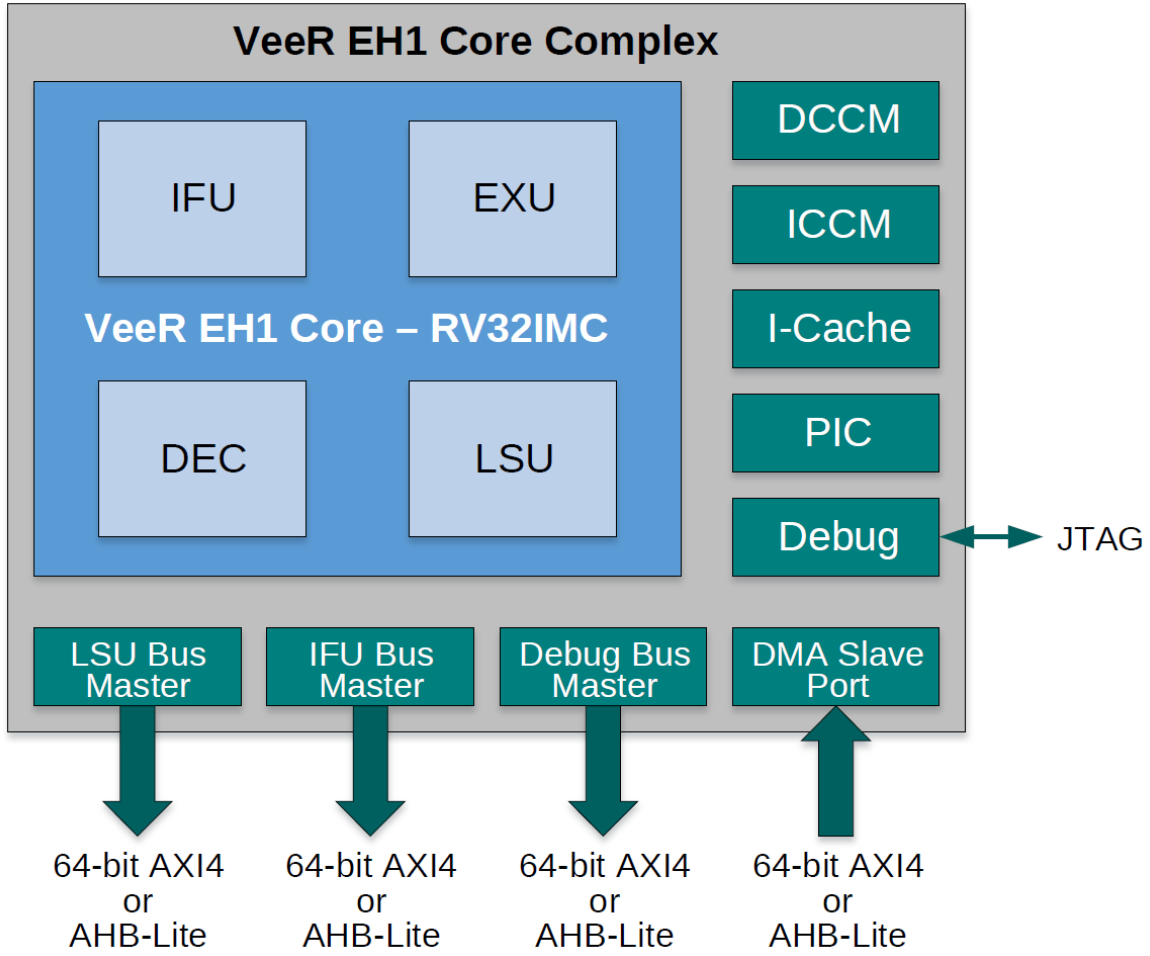
- 操作系统
  - Ubuntu 18.04和20.04
  - Windows 10
  - macOS

RVfpga模块层次



- **VeeR EH1 Core / Core Complex**
  - 包括内核、存储器、可编程中断控制器、总线接口、调试单元
- **VeeRwolfX SoC**
  - VeeRwolf的扩展，增加了新的外设
  - 前者包含引导ROM、UART、系统控制器、AXI/Wishbone总线、SPI控制器
- **RVfpga System**
  - **RVfpgaNexys:**以Nexys A7电路板及其外设为目标的SweRVolfX SoC，增加了DDR2、时钟域同步、BSCAN等
  - **RVfpgaSim:** 用于仿真的SweRVolfX SoC

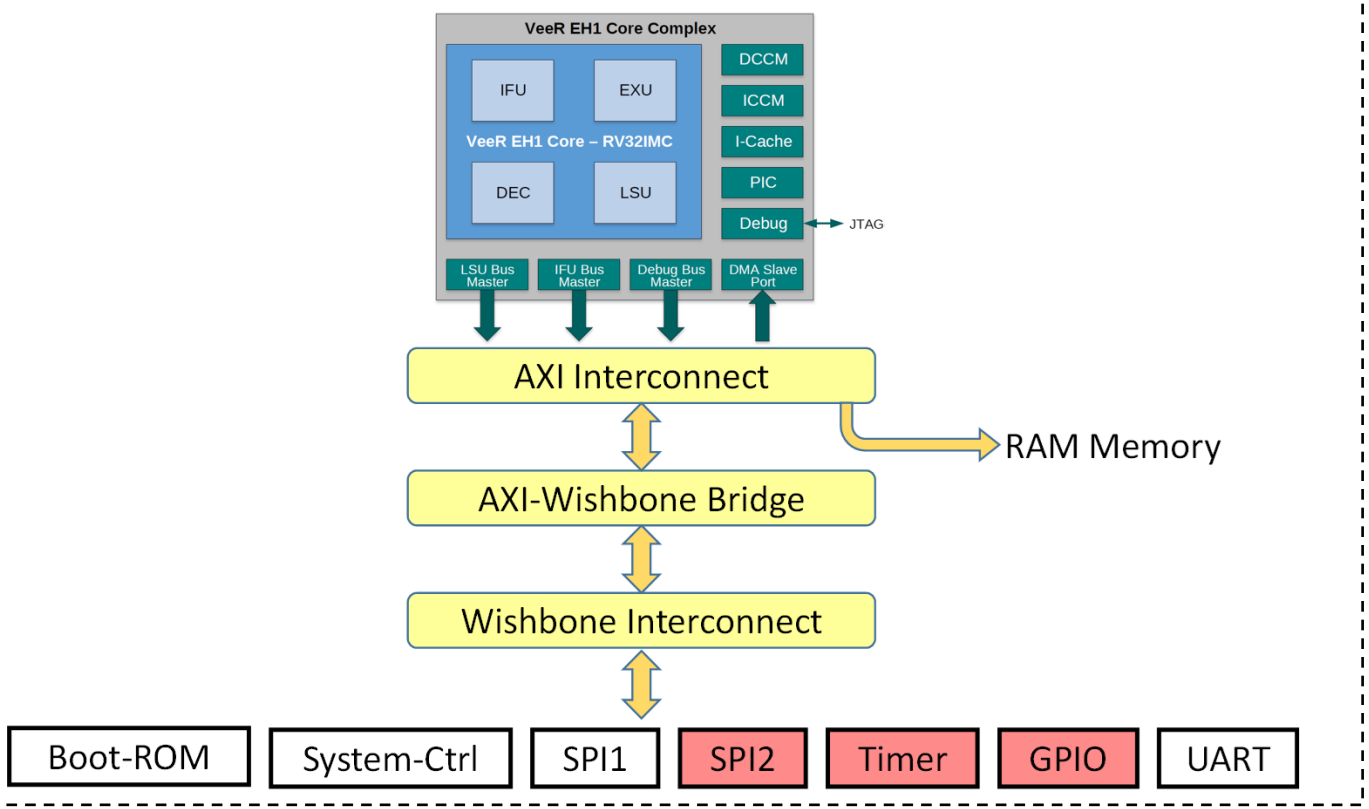
VeeR EH1内核



图片来源：https://github.com/chipsalliance/Cores-VeeR-EH1/blob/main/docs/RISC-V_VeeR_EH1_PRM.pdf

- Western Digital的开源内核
- 32位（RV32ICM）超标量内核，具有双发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

扩展VeeRwolf SoC

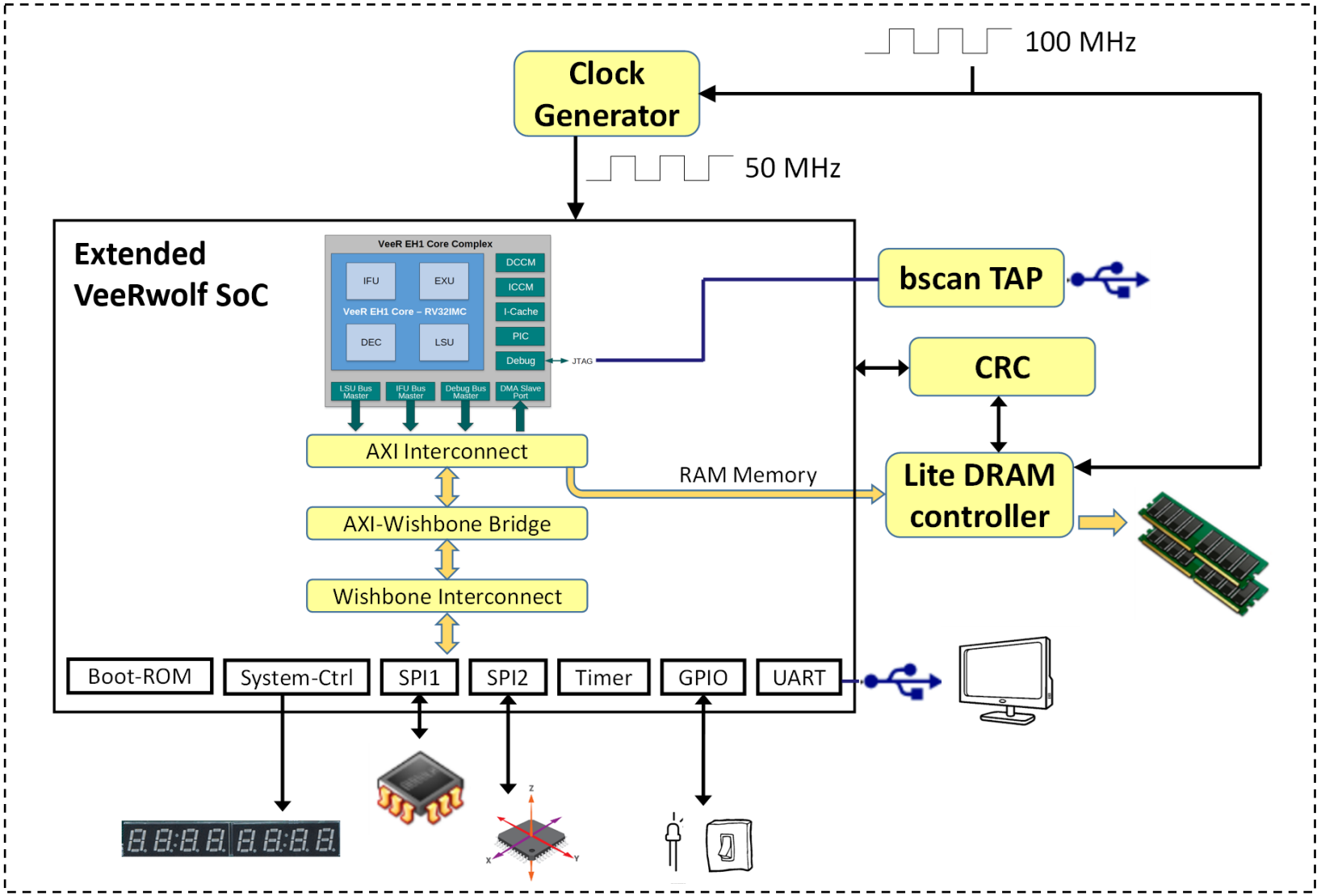


扩展VeeRwolf存储器映射

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SoC
- VeeRwolf使用VeeR EH1内核。VeeRwolf包括Boot-ROM、UART、系统控制器和SPI控制器（SPI1）
- RVfpga通过添加另一个SPI控制器（SPI2）、GPIO、8位7段显示屏和定时器扩展VeeRwolf。
- VeeR内核使用AXI总线，而外设使用Wishbone总线，因此SoC还具有AXI与Wishbone转换模块

RVfpgaNexys



- **RVfpga**：以Nexys A7 FPGA电路板为目标的扩展SweRVolf SoC（添加若干外设）：
  - **内核和系统**：
    - 扩展VeeRwolf 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电路板上使用的外设：
    - 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

RVfpgaSim

- **RVfpgaSim**在VeeRwolfX SoC外添加了一层testbench，用于HDL仿真

目录

- RVfpga课程介绍
- 实验1：RISC-V编程
- 实验2：VeeR EH1流水线
- 实验3：I/O简介与中断驱动I/O
- 实验4：VeeR EH1数据冒险
- 实验5：硬件计数器与基准测试
- 讨论 Q&A

实验1： RISC-V编程

- 将彩色图转换为灰度图的图像处理程序
- 有些函数用C语言编写，有些则用汇编语言编写

实验1：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

实验1：汇编函数

```
.globl ColourToGrey_Pixel
.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
```

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

```
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);
}
```

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

实验1： 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
- **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：函数调用过程中不用保留寄存器内容（即，函数调用前后寄存器值无需保持不变）

实验1： 保留/非保留寄存器

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

实验1：结构与数组

```
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])
                                      ...    }
```

实验1：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);

  // Initialize Uart
  uartInit();

  // Print message on the serial output
  printfNexys("Created Grey Image");
  ...
}
```

- 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**打开串口**：

实验1： Western Digital的BSP和PSP

- Western Digital提供：
  - **PSP**：处理器支持包
  - **BSP**：电路板支持包

- 这两个支持包为给定的处理器（VeeR EH1内核）和电路板（Nexys A7 FPGA电路板）提供了常用的函数。
  - 示例：`printfNexys`（类似C语言的`printf`函数）

实验1：结果可视化

- 启动调试器并执行到程序末尾循环
- 在调试控制台（debug console）执行以下命令

  cd AdditionalFiles

  dump value GreyImage.dat GreyImage
- 将dat文件转换为ppm文件
  - 在Linux下 ./dump2ppm GreyImage.dat GreyImage.ppm 128 128 1
  - 在windows下使用上面相同的参数执行dump2ppm.exe
- 使用GIMP打开ppm文件
  - https://www.gimp.org/downloads/

实验1：练习

- 练习1：对256*256输入图像执行程序。
- 练习2：统计灰度图像中接近白色（>235）和接近黑色（<20）的像素点数量，并在串口输出。
- 练习3：将ColourToGrey_Pixel汇编子程序转换为C函数，将C函数ColourToGrey转换为调用ColourToGrey_Pixel C函数的汇编子程序。
- 练习4：将“Blur Filter”（模糊滤镜）应用于彩色图像。

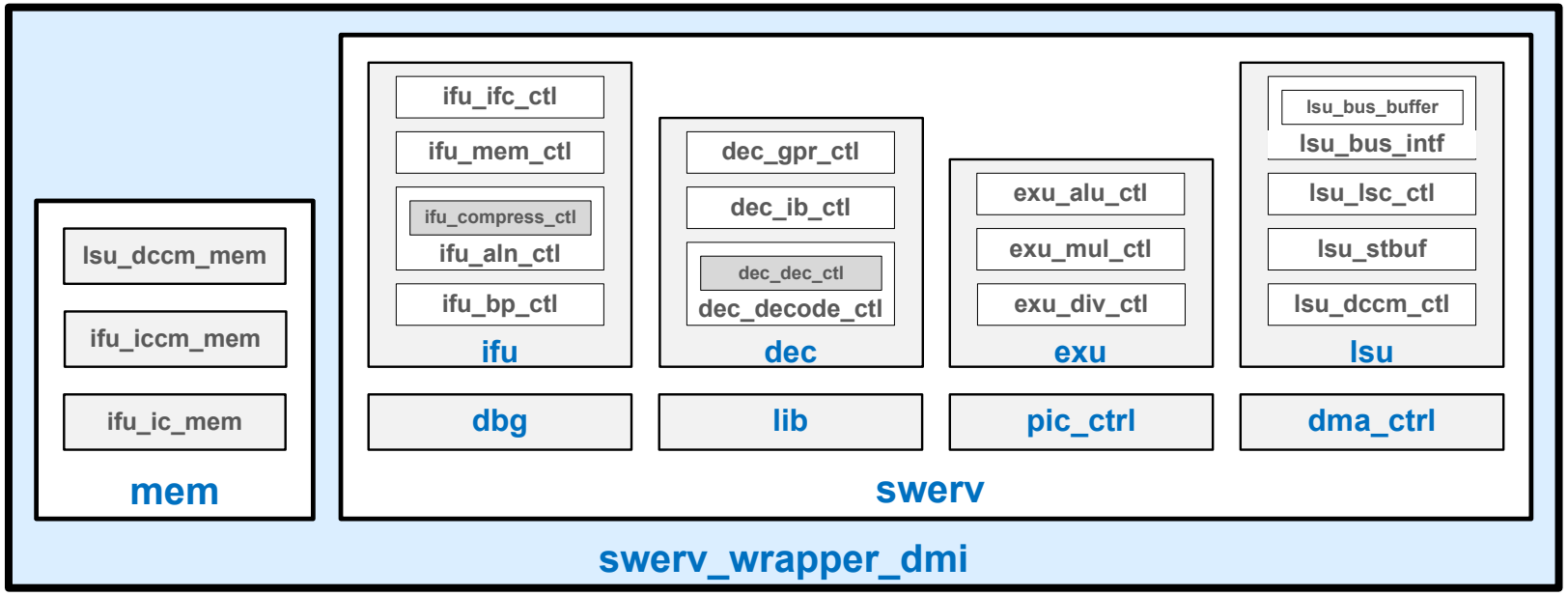
茶歇 10:30-10:50

目录

- RVfpga课程介绍
- 实验1：RISC-V编程
- 实验2：VeeR EH1流水线
- 实验3：I/O简介与中断驱动I/O
- 实验4：VeeR EH1数据冒险
- 实验5：硬件计数器与基准测试
- 讨论 Q&A

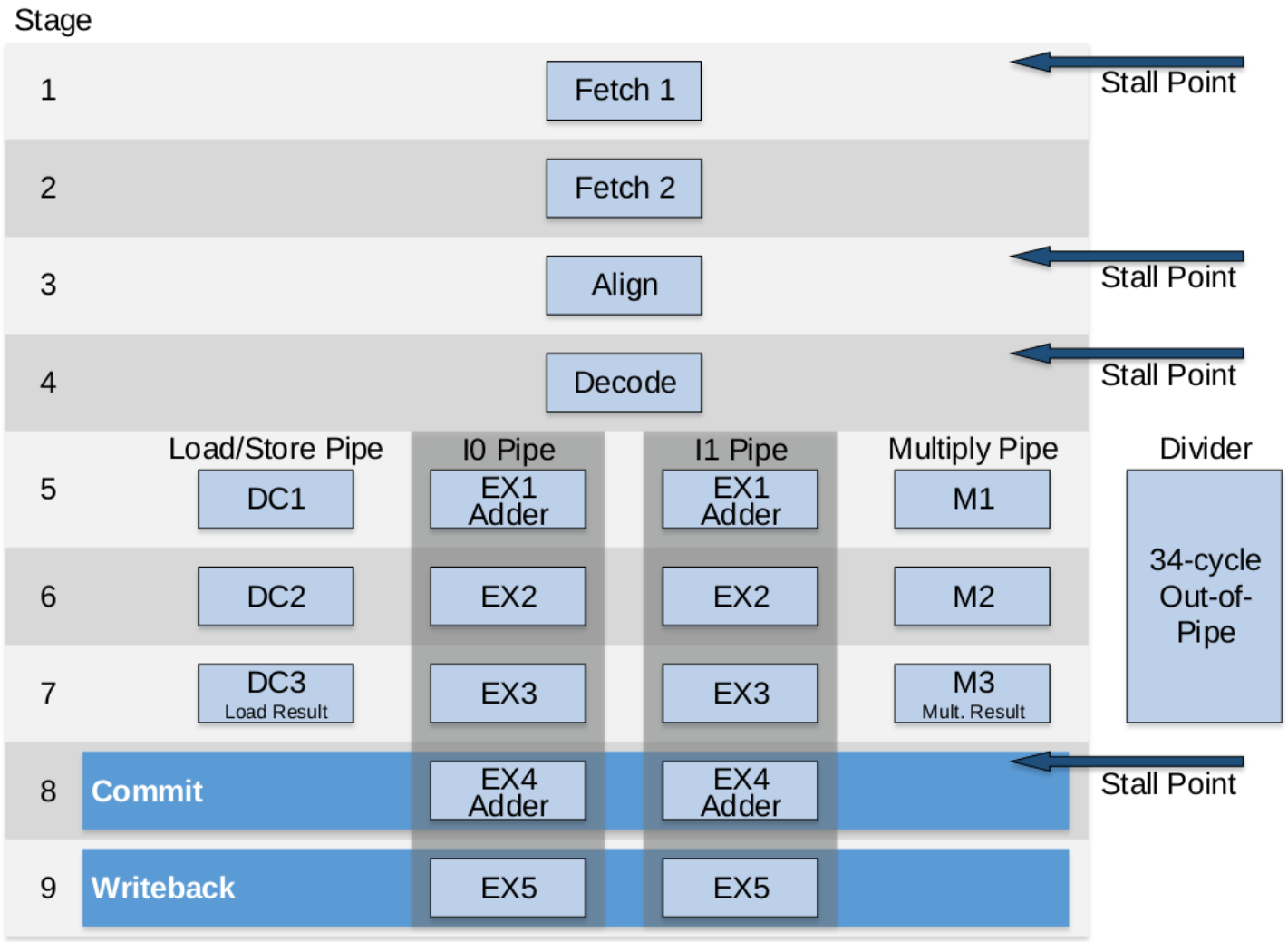
实验2： VeeR EH1模块

- 下图给出了主要Verilog模块的层级。
- **mem**模块对构成VeeR 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CPU实例，包含IFU、DEC、EXU、LSU四大模块和dbg、pic、dma等模块。



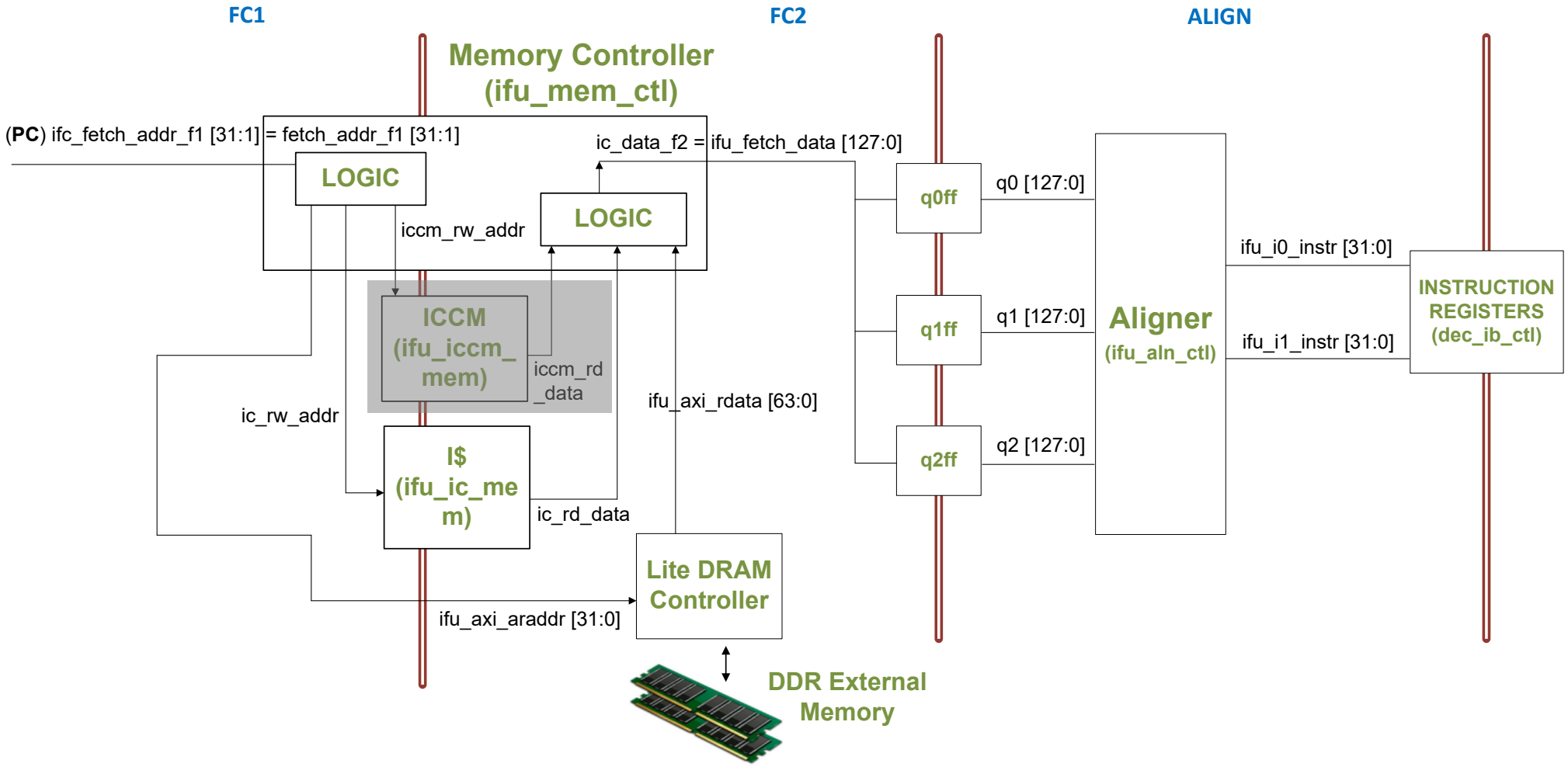
实验2： VeeR EH1流水线

- VeeR EH1是32位2路超标量9级流水线顺序处理器
- 实验11-20将对此进行详细分析



实验2：取指和对齐阶段

- VeeR 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

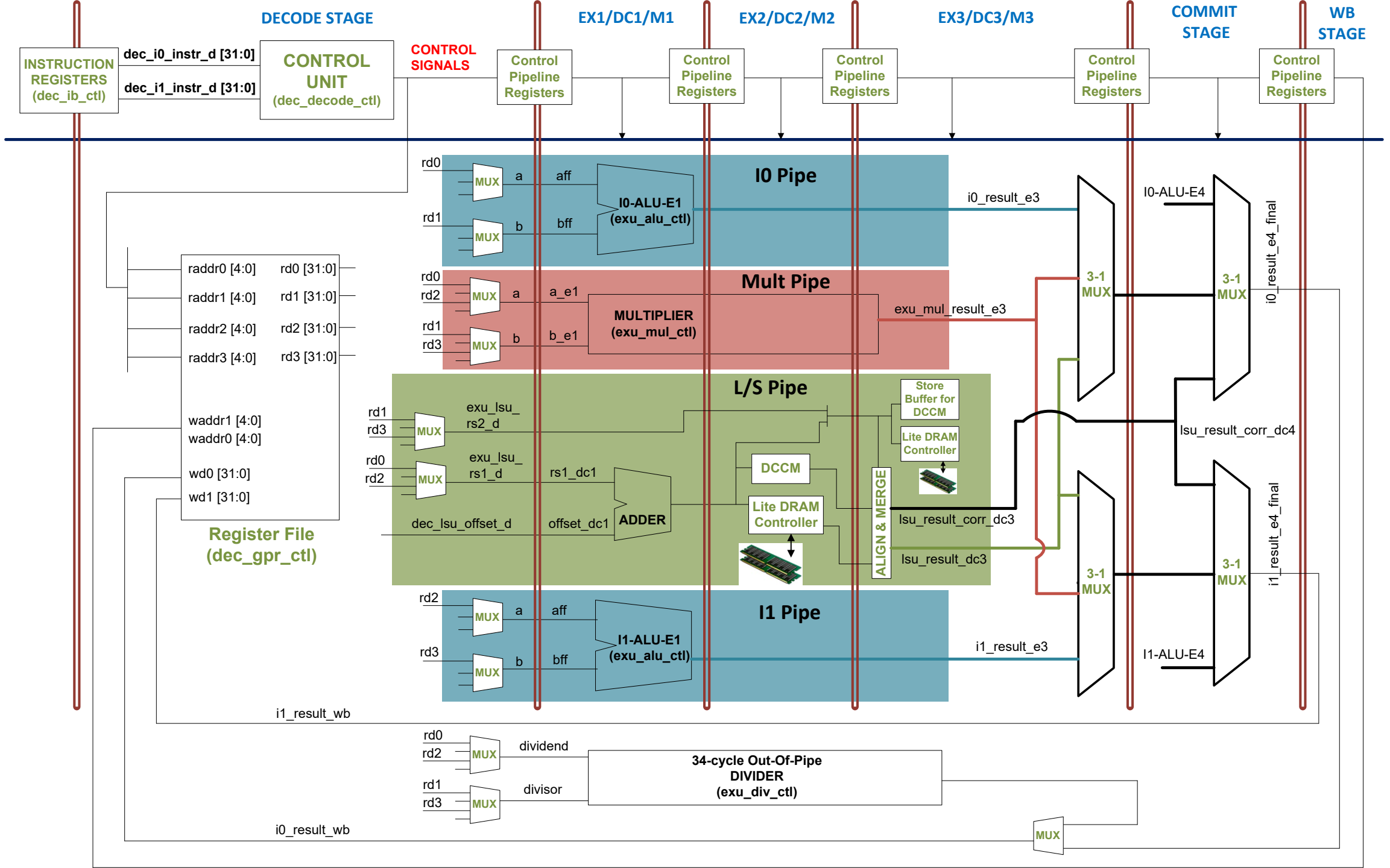


实验2：取指（FC1和FC2）阶段

- **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数据。）

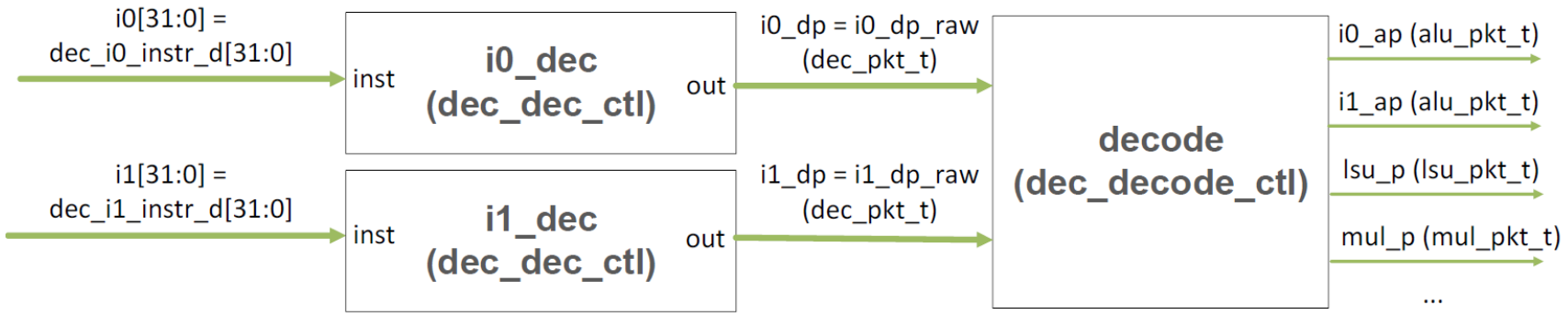
实验2：对齐（ALIGN）阶段

- 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VeeR 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。


DECODE STAGE
EX1/DC1/M1
EX2/DC2/M2
EX3/DC3/M3
COMMIT STAGE
WB STAGE
INSTRUCTION REGISTERS (dec_ib_ctl)
dec_i0_instr_d [31:0]
dec_i1_instr_d [31:0]
CONTROL UNIT (dec_decode_ctl)
CONTROL SIGNALS
Control Pipeline Registers
I0 Pipe
I0-ALU-E1 (exu_alu_ctl)
i0_result_e3
Mult Pipe
MULTIPLIER (exu_mul_ctl)
exu_mul_result_e3
L/S Pipe
Store Buffer for DCCM
Lite DRAM Controller
DCCM
ADDER
ALIGN & MERGE
lsu_result_corr_dc3
lsu_result_dc3
I1 Pipe
I1-ALU-E1 (exu_alu_ctl)
i1_result_e3
I0-ALU-E4
I1-ALU-E4
3-1 MUX
i0_result_e4_final
i1_result_e4_final
lsu_result_corr_dc4
Register File (dec_gpr_ctl)
raddr0 [4:0]
raddr1 [4:0]
raddr2 [4:0]
raddr3 [4:0]
rd0 [31:0]
rd1 [31:0]
rd2 [31:0]
rd3 [31:0]
waddr1 [4:0]
waddr0 [4:0]
wd0 [31:0]
wd1 [31:0]
i1_result_wb
i0_result_wb
34-cycle Out-Of-Pipe DIVIDER (exu_div_ctl)
dividend
divisor

实验2：译码（DEC）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**

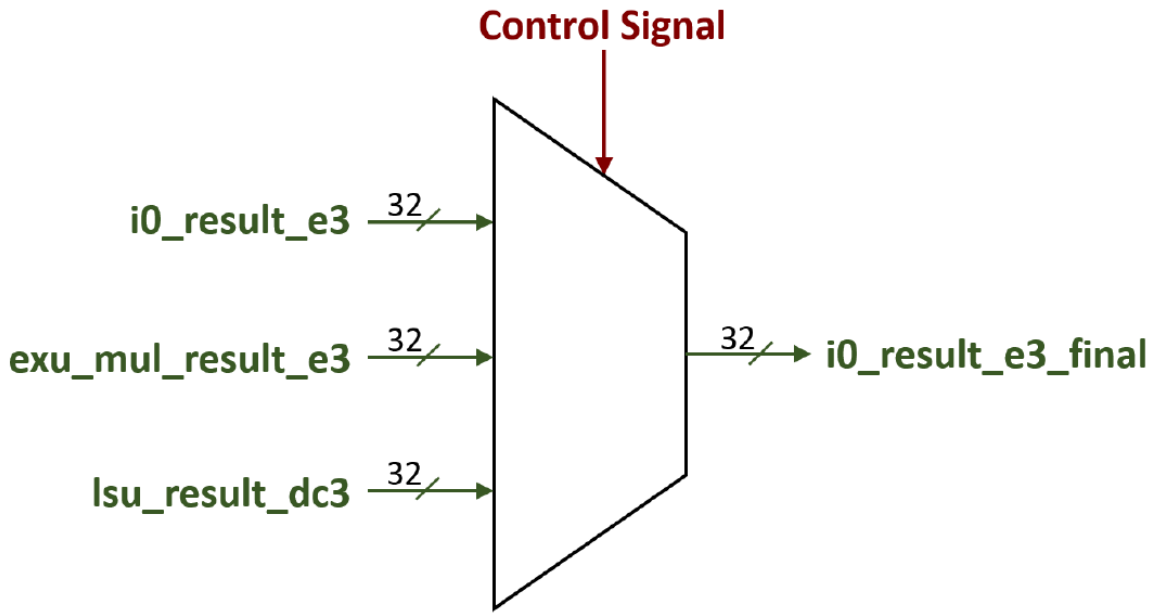


  - **将指令和操作数分发到适当的通道：**
    - 两个整数管道：I0和I1
    - 乘法管道
    - 装载/存储管道（L/S）
    - 34周期除法器
  - **多路开关选择的操作数来自：**
    - 转发逻辑
    - 立即数
    - 寄存器文件

实验2：执行（EX/DC/M）阶段

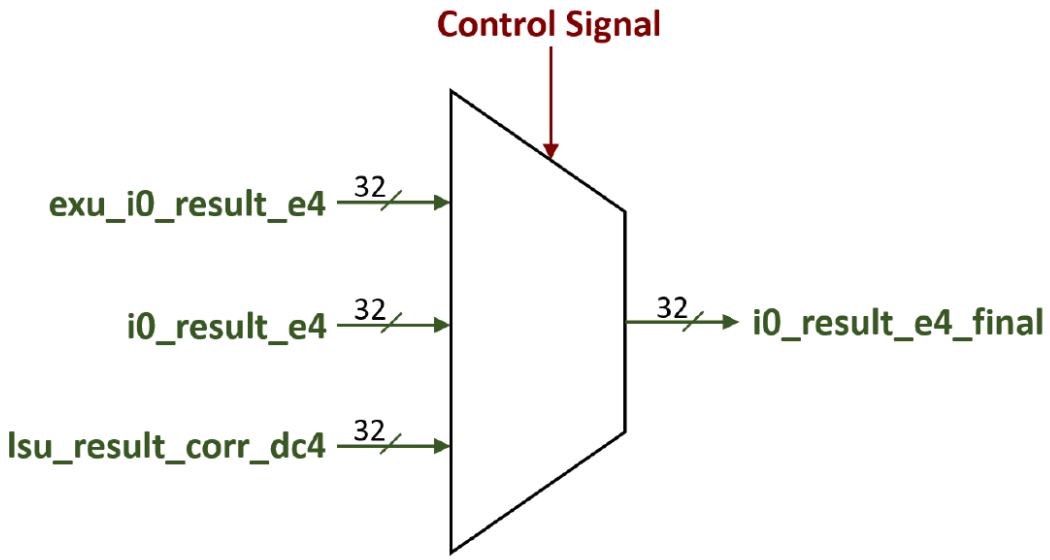
- **I0/I1管道**：两个整数管道具有三个阶段（EX1、EX2和EX3）。EX1执行ALU运算。
- **乘法管道**：乘法管道包含一个使用三个阶段（M1、M2和M3）的3周期整数乘法器。
- **装载/存储（L/S）管道**：L/S管道执行装载和存储指令。
- **除法器**：除法器是一个非流水34周期整数除法器。

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



实验2：提交（Commit）和回写（WB）阶段

- **提交阶段：**选择要回写到寄存器文件的结果。

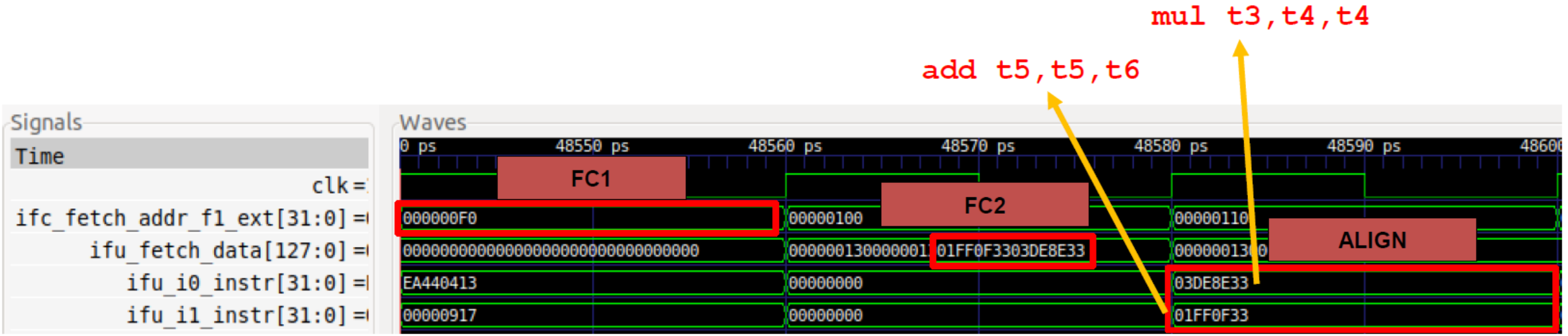


- **回写阶段：**使用写端口0和1将结果写入寄存器文件。控制流水线寄存器提供寄存器标识符和使能信号（在译码阶段生成）。

实验2：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
```

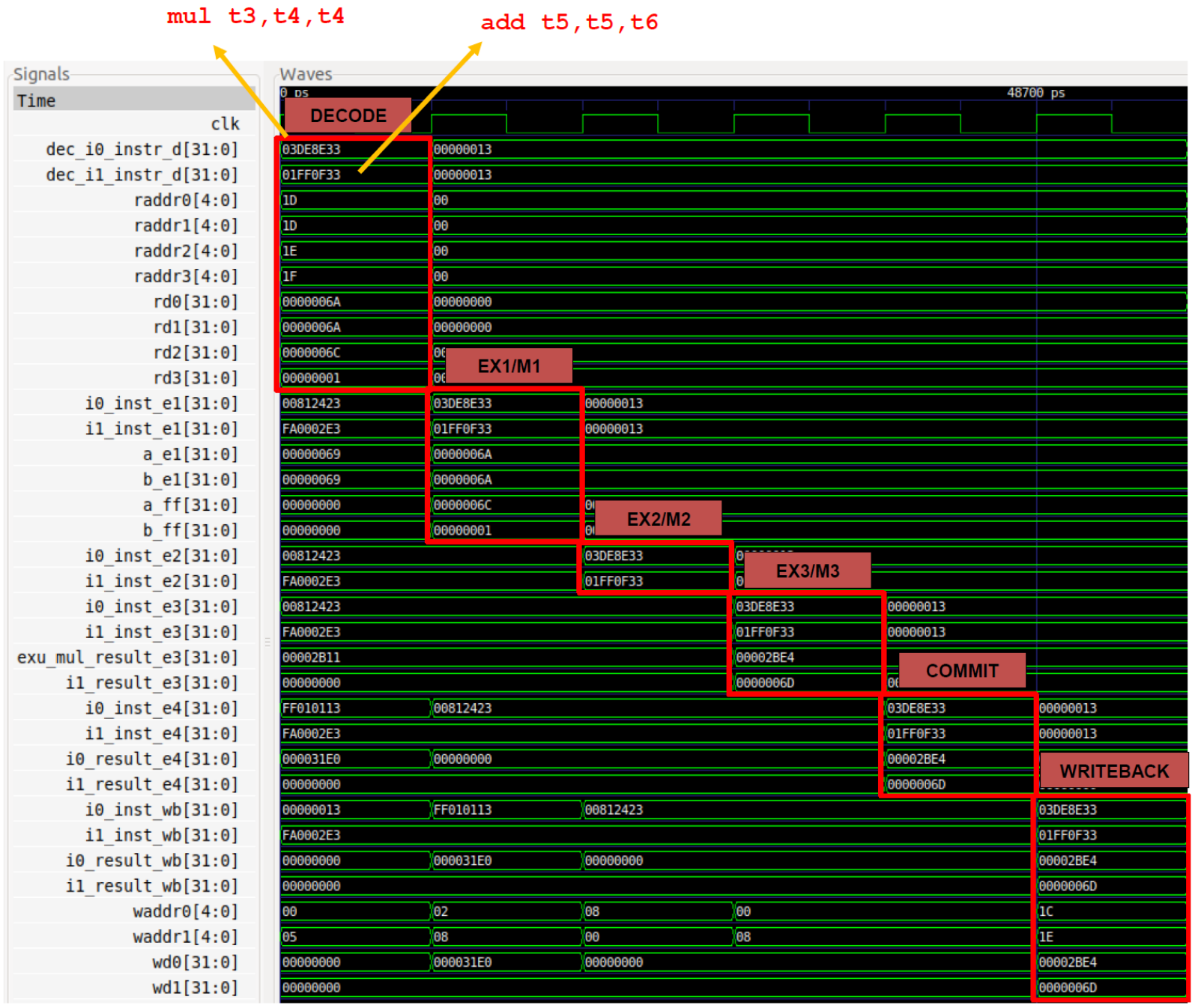
实验2：仿真分析



- **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 = 0x0000001300000013**01FF0F3303DE8E33**
- **对齐**：提取两条指令并分发到VeeR 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）

实验2：仿真分析

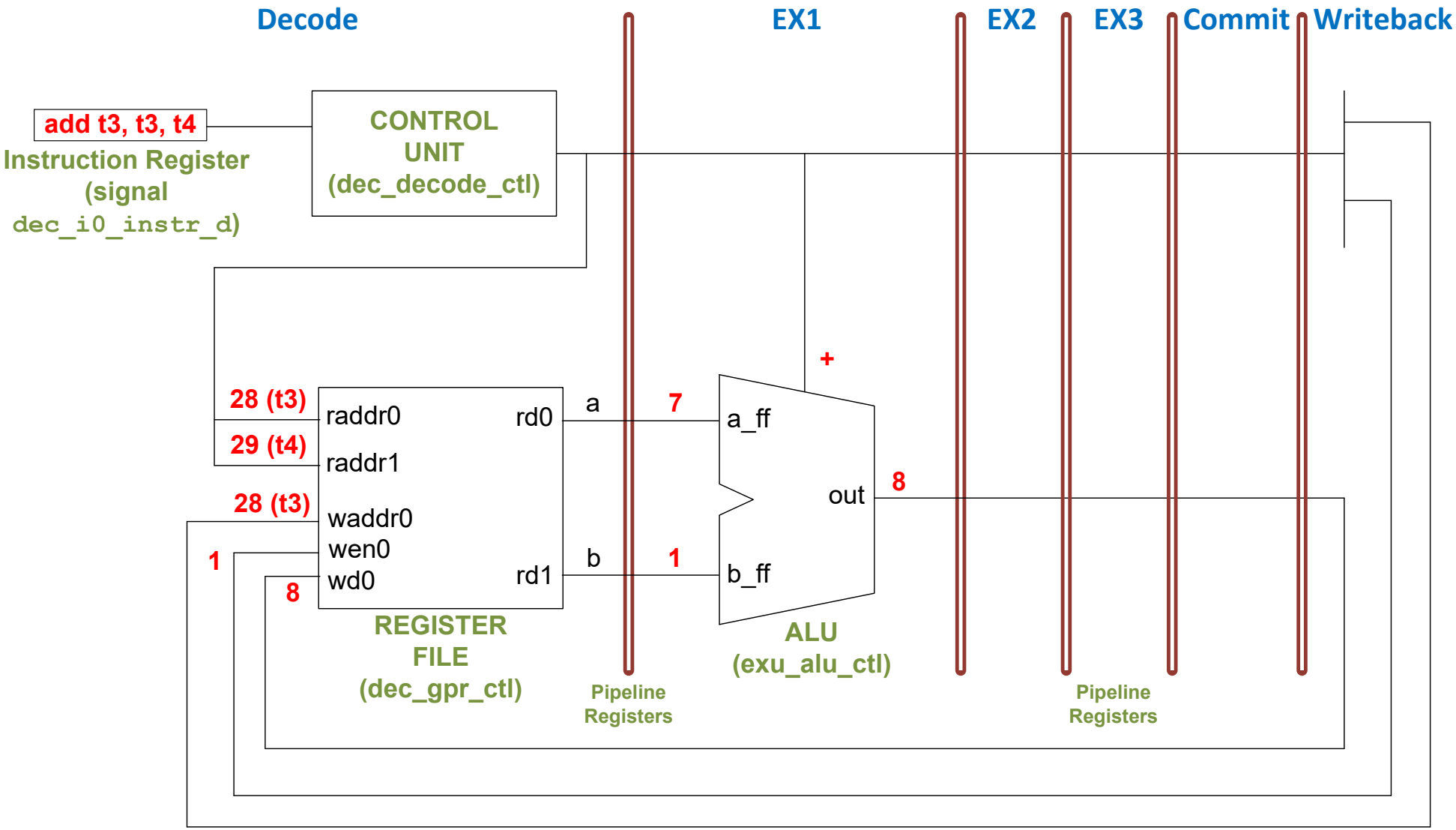
- **译码**：从寄存器文件中读取指令的操作数，然后提供给乘法管道和I1管道。
- **EX1/2/3和提交**：计算加法和乘法。
  - `i0_result_e4` = `exu_mul_result_e3` = 0x6A * 0x6A = 0x2BE4
  - `i1_result_e4` = `i1_result_e3` = 0x6C + 0x01 = 0x6D
- **回写**：将结果回写到寄存器文件中。
  - `waddr0` = 0x1C　　`wd0` = 0x2BE4
  - `waddr1` = 0x1E　　`wd1` = 0x6D



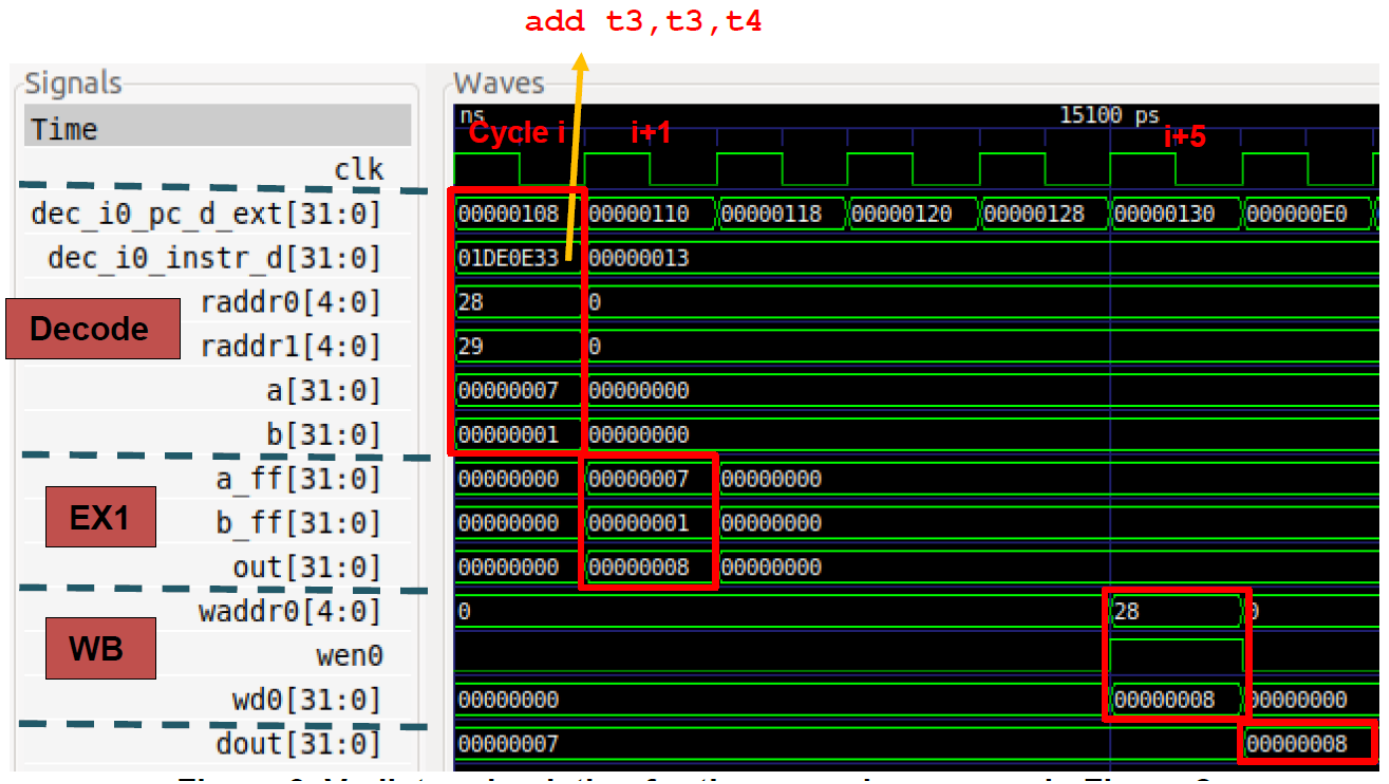
实验2：add指令进一步分析

```
.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
```

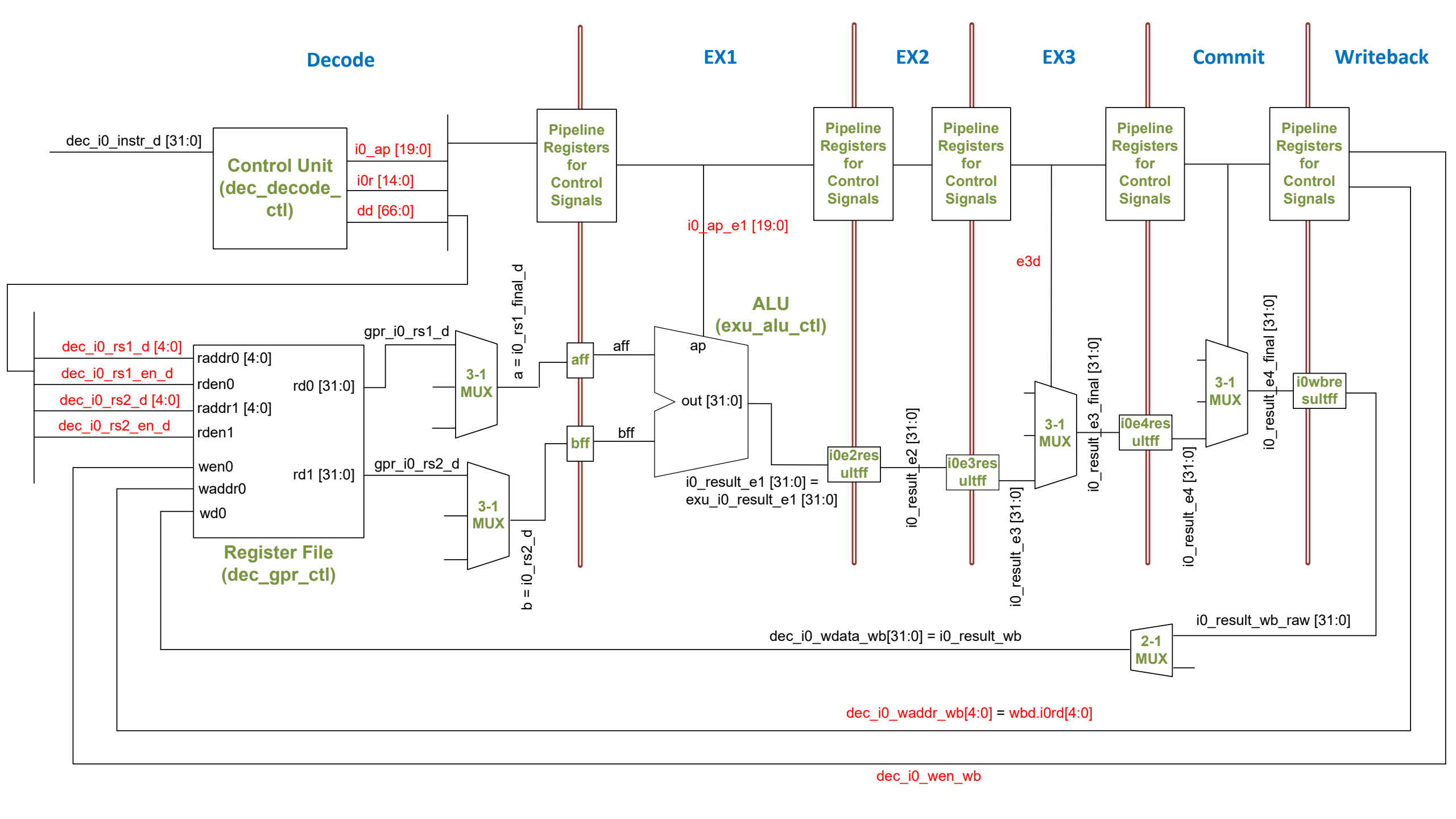
实验2： add流水线


Decode
EX1
EX2
EX3
Commit
Writeback
add t3, t3, t4
Instruction Register
(signal
dec_i0_instr_d)
CONTROL
UNIT
(dec_decode_ctl)
28 (t3)
29 (t4)
28 (t3)
1
8
raddr0
raddr1
waddr0
wen0
wd0
rd0
rd1
REGISTER
FILE
(dec_gpr_ctl)
a
b
7
1
+
a_ff
b_ff
out
8
ALU
(exu_alu_ctl)
Pipeline
Registers
Pipeline
Registers

实验2：仿真分析



- 周期i：信号dec_i0_instr_d包含32位机器指令0x01DE0E33。add指令的字段为：00 | rs1 | 000 | rd | 0110011。这一阶段将生成**控制信号**并**读取寄存器文件**。
- 周期i+1：**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：将加法结果**回写**到寄存器文件中：wd0 = 8、wen0 = 1且waddr0 = 28


Decode
EX1
EX2
EX3
Commit
Writeback
dec_i0_instr_d [31:0]
Control Unit (dec_decode_ctl)
i0_ap [19:0]
i0r [14:0]
dd [66:0]
Pipeline Registers for Control Signals
Pipeline Registers for Control Signals
Pipeline Registers for Control Signals
Pipeline Registers for Control Signals
Pipeline Registers for Control Signals
i0_ap_e1 [19:0]
e3d
ALU (exu_alu_ctl)
dec_i0_rs1_d [4:0]
dec_i0_rs1_en_d
dec_i0_rs2_d [4:0]
dec_i0_rs2_en_d
raddr0 [4:0]
rden0
raddr1 [4:0]
rden1
wen0
waddr0
wd0
rd0 [31:0]
rd1 [31:0]
Register File (dec_gpr_ctl)
gpr_i0_rs1_d
gpr_i0_rs2_d
3-1 MUX
3-1 MUX
a = i0_rs1_final_d
b = i0_rs2_d
aff
bff
aff
bff
ap
out [31:0]
i0_result_e1 [31:0] = exu_i0_result_e1 [31:0]
i0e2res ultff
i0e3res ultff
i0e4res ultff
i0wbre sultff
i0_result_e2 [31:0]
i0_result_e3 [31:0]
i0_result_e4 [31:0]
i0_result_e3_final [31:0]
i0_result_e4_final [31:0]
3-1 MUX
3-1 MUX
2-1 MUX
i0_result_wb_raw [31:0]
dec_i0_wdata_wb[31:0] = i0_result_wb
dec_i0_waddr_wb[4:0] = wbd.i0rd[4:0]
dec_i0_wen_wb

实验2：练习

- 1. 对以下逻辑指令执行与本实验中提供的分析类似的分析：and、or和xor。
- 2. 在Verilator仿真中以及直接在Verilog代码中分析RV32I基本整数指令集中提供的左移/右移指令：srl、sra和sll。
- 3. 在Verilator仿真中以及直接在Verilog代码中分析RV32I基本整数指令集中提供的小于置位指令：slt和sltu。
- 4. 在Verilator仿真中以及直接在Verilog代码中分析RV32I基本整数指令集中提供的立即数指令：addi、andi、ori、xori、srli、srai、slli、slti和sltui。
- 5. 上述流水线中没有讨论立即数，流水线中需要哪些额外的逻辑来支持支持I类型指令？以addi指令为例进行分析

Imagination

GPU | CPU | AI

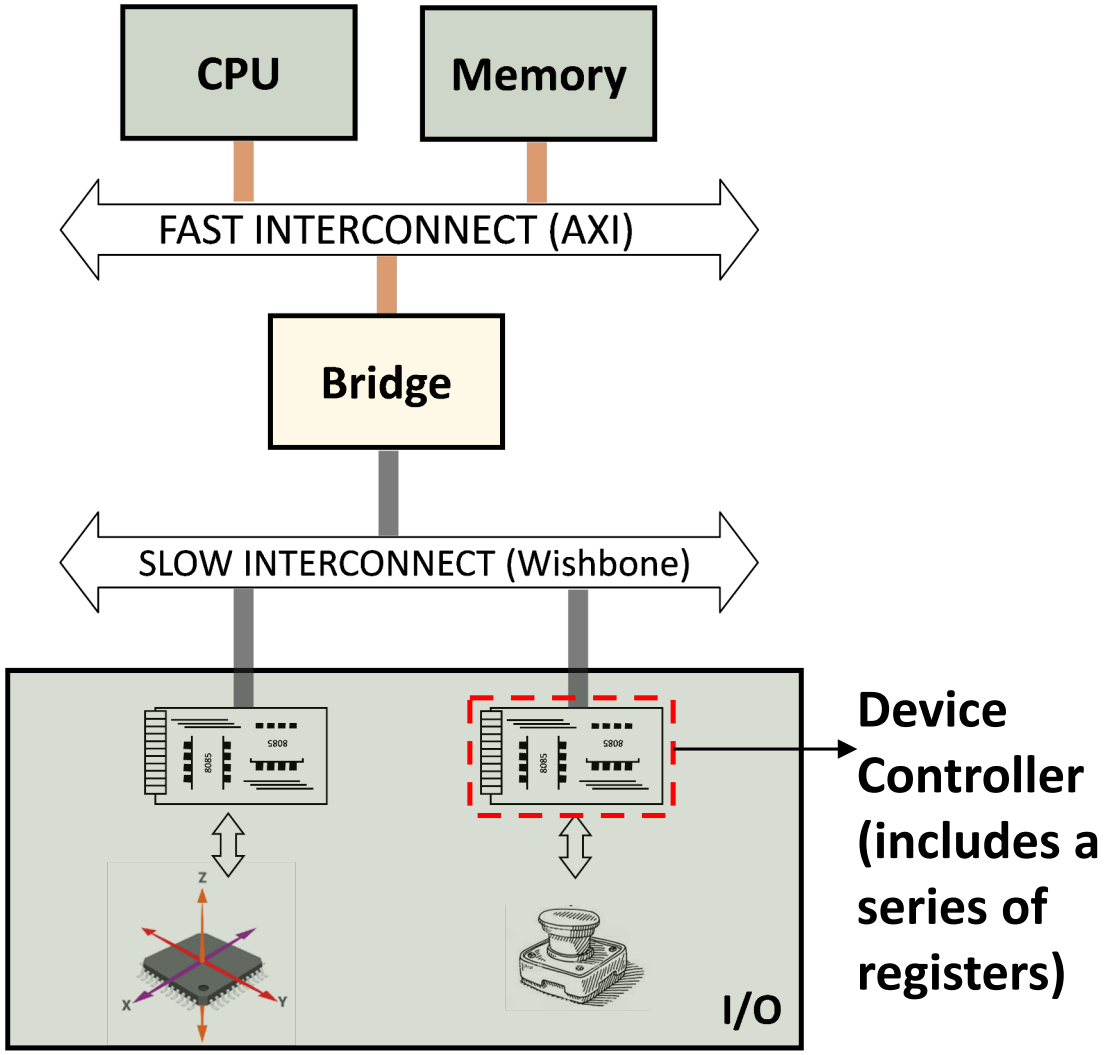
群聊: IM G&浙大—Rvfpga workshop

目录

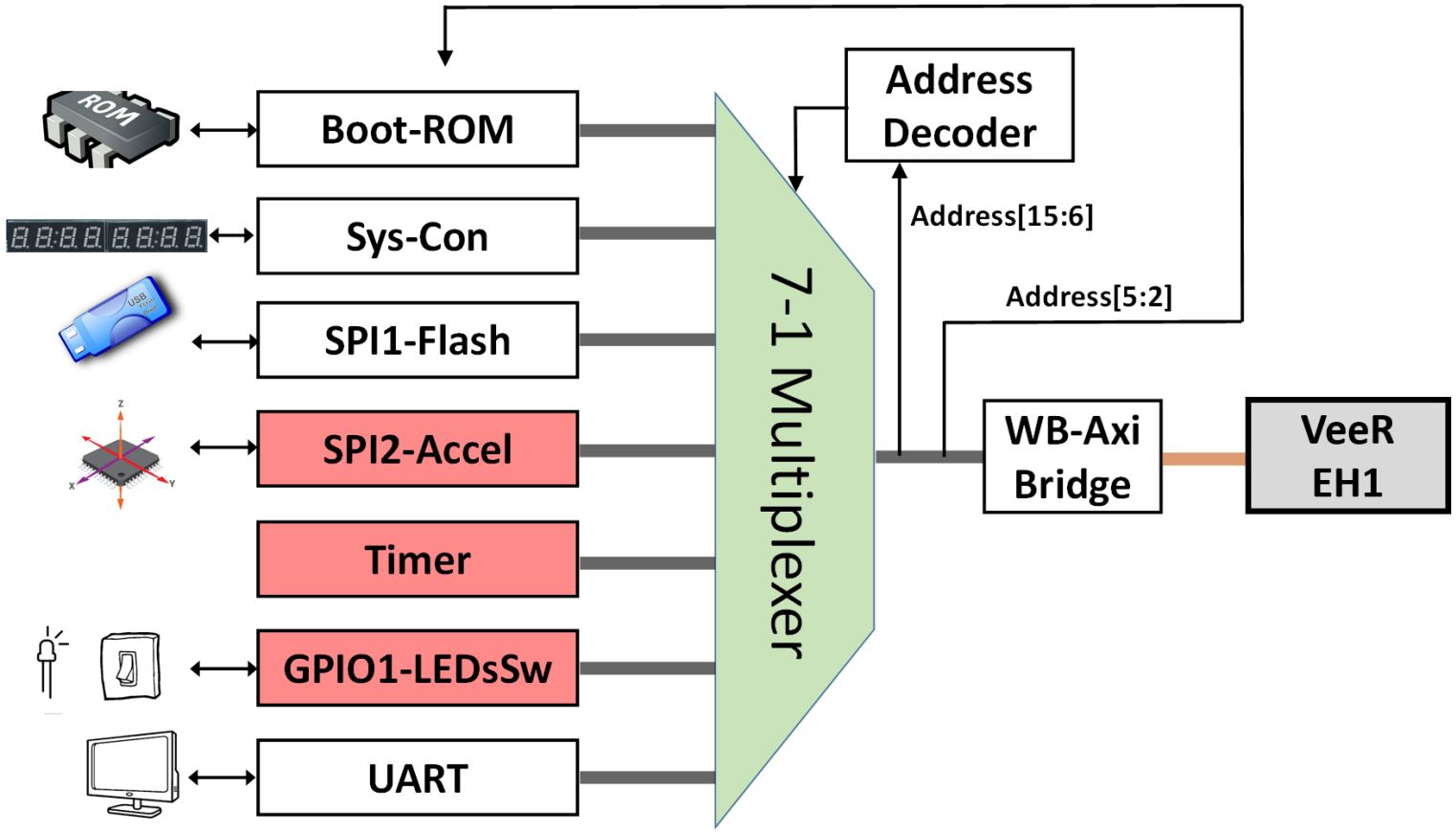
- RVfpga课程介绍
- 实验1：RISC-V编程
- 实验2：VeeR EH1流水线
- 实验3：I/O简介与中断驱动I/O
- 实验4：VeeR EH1数据冒险
- 实验5：硬件计数器与基准测试
- 讨论 Q&A

实验3：I/O与中断

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



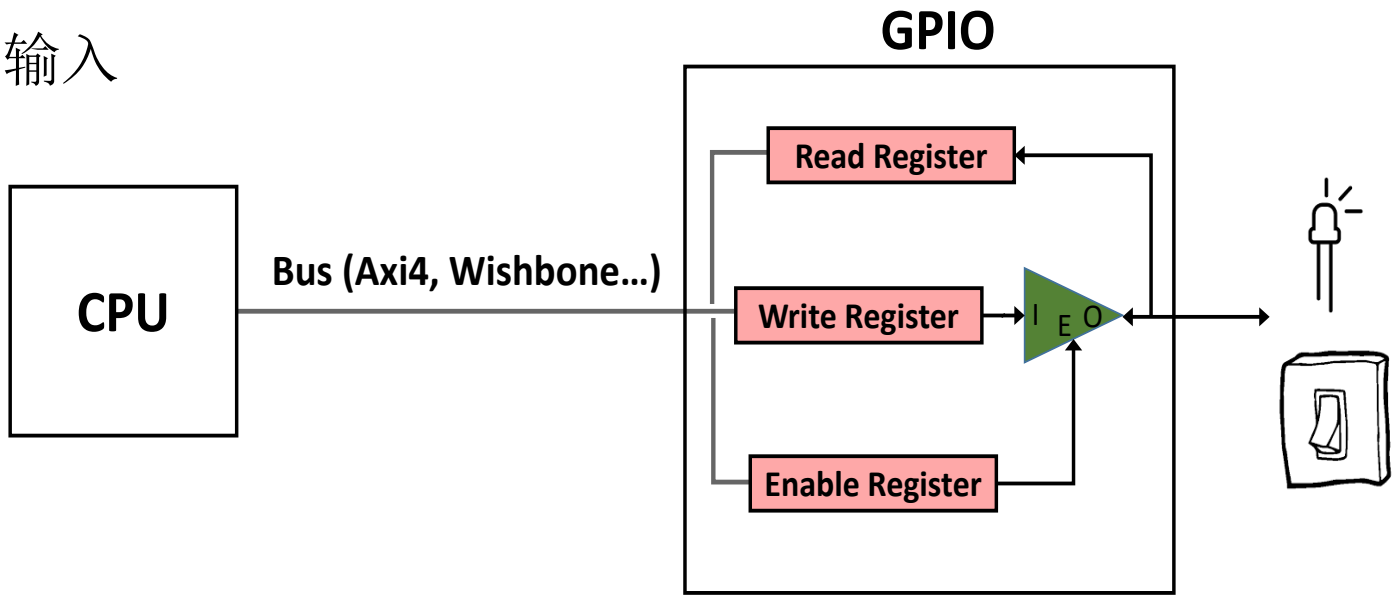
实验3： 具有I/O的处理器



- **VeeRwolf外设：**
  - 引导ROM
  - 系统控制器
  - SPI1闪存
  - UART
- **RVfpga添加了以下外设：**
  - 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）

实验3：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



实验3： 寄存器映射存储地址

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
```

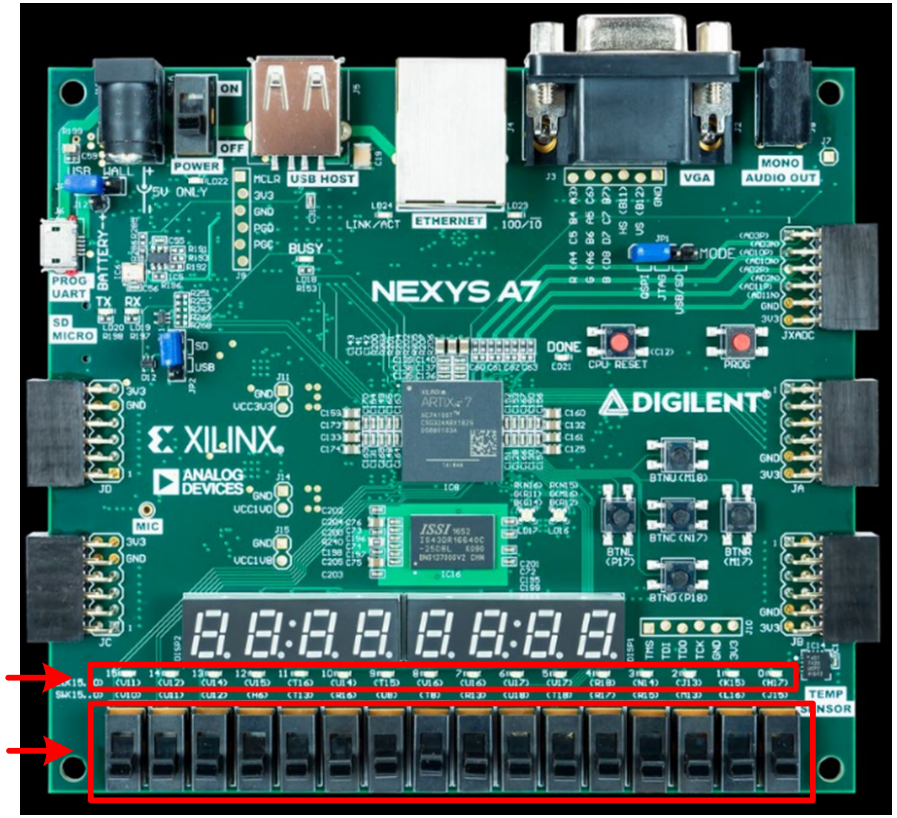
实验3： RVfpga GPIO模块

- **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
- **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）

实验3： 写入LED与读取开关



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]15:0
```

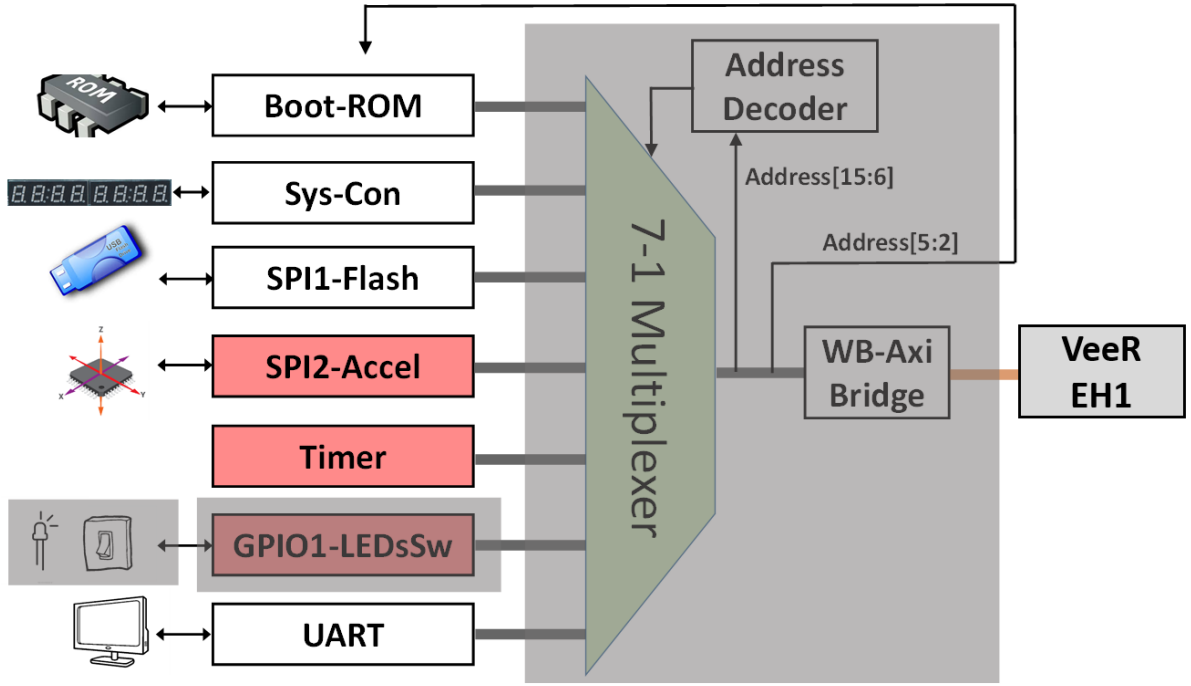
读取开关：

- 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]31:16 = switch values
srli t5, t5, 16 # [t5]15:0 = switch values
```

实验3： GPIO实现

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



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

实验3： 外部连接与顶层连接

文件**rvfpga.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

文件**rvfpga.sv**：定义i_sw和o_led的合并信号io_data[31:0]与swervolf_core连接

```
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] }]
```

```
256         .i_ram_init_error (litedram_init_error),
257         .io_data        ({i_sw[15:0],gpio_out[15:0]}),
258         .AN (AN),
259         .Digits_Bits ({CA,CB,CC,CD,CE,CF,CG}),
260         .o_accel_sclk   (accel_sclk),
261         .o_accel_cs_n   (o_accel_cs_n),
262         .o_accel_mosi   (o_accel_mosi),
263         .i_accel_miso   (i_accel_miso));
264
265      always @(posedge clk_core) begin
266         o_led[15:0] <= gpio_out[15:0];
267      end
268
```

实验3： 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));
```

实验3： 与VeeR 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'hffffffc0, 32'hffffffc0, 32'hffffffc0, 32'hffffffc0, 32'hffffffc0, 32'hfffff000}))
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

实验3：中断驱动I/O

- **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，不断轮询一个值（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。

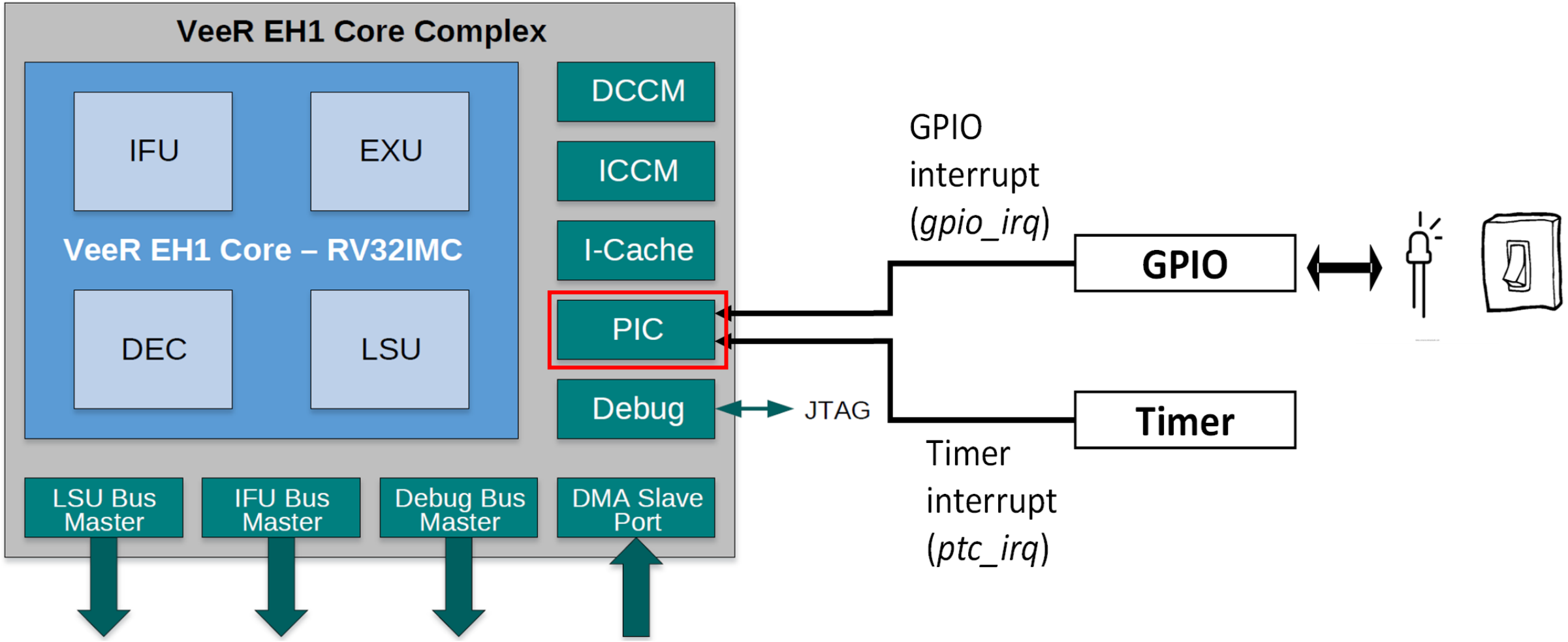
实验3：中断处理

- 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**
- VeeR EH1内核按照RISC-V的PLIC（平台级中断控制器）规范处理中断。该内核包含一个可编程中断控制器（**PIC**）。它具有：
  - 255个中断源
  - 15个优先级

实验3： PIC主要功能

- 允许/禁止：PIC能够允许/禁止外部中断
- 配置：可以将PIC配置为监听具有不同极性（高电平有效/低电平有效）或类型（边沿触发/电平触发）的外部中断。PIC还允许将ISR分配给不同的存储器地址
- 过滤和优先级分配：PIC允许为中断分配优先级。当主程序运行时，PIC选择已允许的优先级最高的触发中断。
- 通知：PIC选择了优先级最高的中断后，它将通知内核停止执行主程序，以便跳转到为所选中断服务的程序。
- 抢占：如果允许嵌套中断，则可以抢占由另一个具有更高优先级的中断服务的中断。

实验3： 中断硬件结构


VeeR EH1 Core Complex
IFU
EXU
VeeR EH1 Core – RV32IMC
DEC
LSU
DCCM
ICCM
I-Cache
PIC
Debug
JTAG
LSU Bus Master
IFU Bus Master
Debug Bus Master
DMA Slave Port
GPIO interrupt (gpio_irq)
GPIO
Timer
Timer interrupt (ptc_irq)

实验3：配置中断

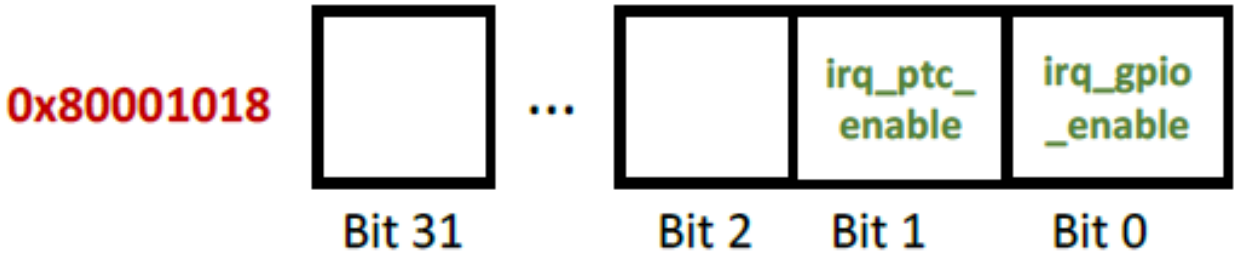
- 使用WD的PSP/BSP：
  - 使用WD的PSP/BSP初始化中断
  - 初始化255个中断中的一个或多个并提供ISR
  - 将应触发中断的外设信号与中断引脚相连
  - 允许所有中断
  - 允许外部中断

实验3：中断示例

- 使用中断读取Switch[0]的值 - 仅在上升沿（0跳变为→1）
- 初始化中断系统
  - 默认初始化（配置向量表、初始化触发IRQ寄存器、清除外部中断）
  - 设置特定优先级阈值
- 初始化外部中断线路IRQ4
- 初始化外设
- 允许全局中断

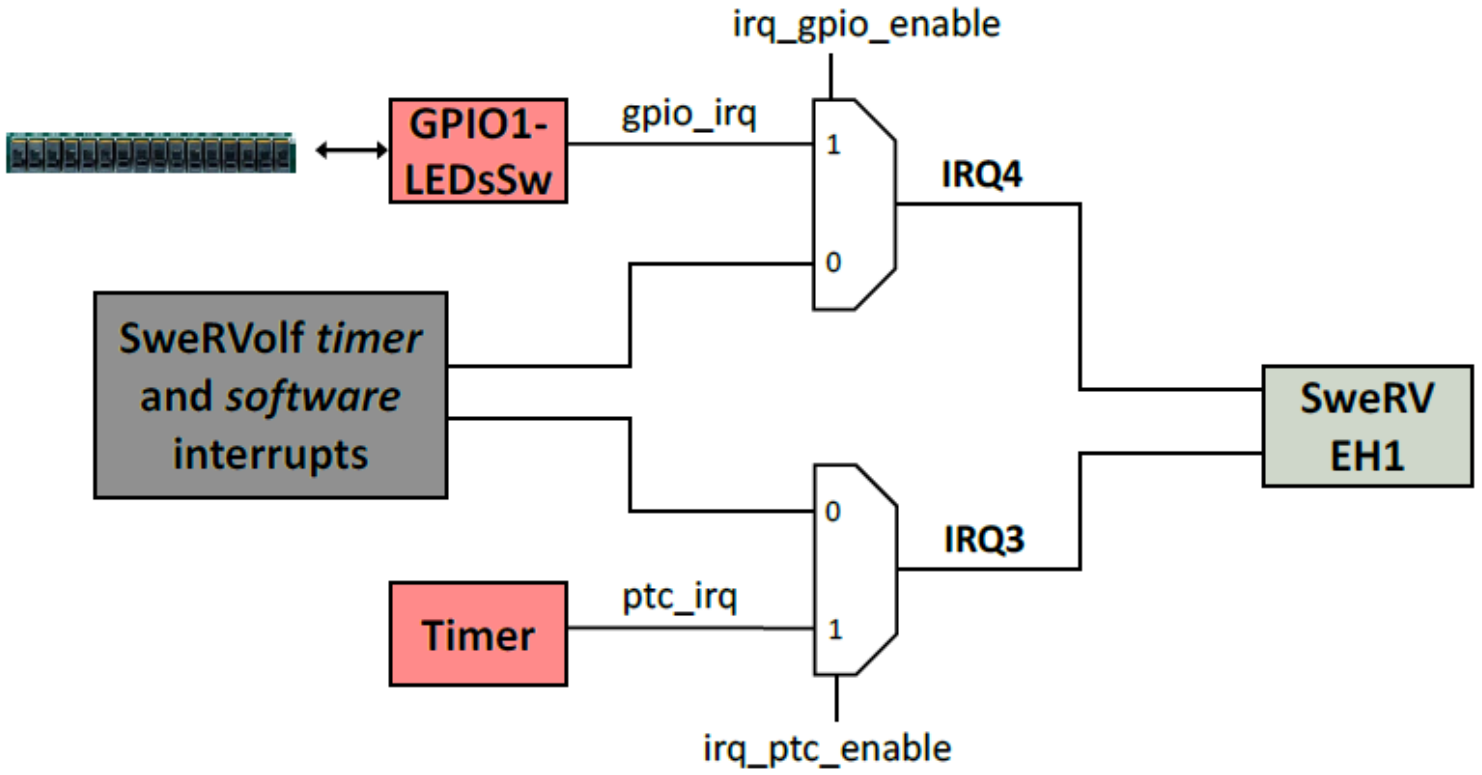
实验3：初始化外部中断线路IRQ4

- 配置IRQ4中断类型，清除潜在待处理中断
- 设置IRQ4优先级
- 在PIC中允许IRQ4中断
- 将GPIO中断服务程序（GPIO_ISR）放入向量表
- 将IRQ4与GPIO中断线路连接



实验3： RVfpga中断逻辑电路

- 添加外部中断源
  - 可以将中断连接到未使用的外部中断源，目前仅使用了255条中断线路中的两条，但需要修改WD的库。
  - 可以将中断连接到IRQ4，与GPIO使用单向量中断模式，尽管在某些情况下多向量模式更为可取，但这种方法的优点是可以重复使用BSP。



实验3： GPIO中断服务程序

- 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);
}
```

实验3：初始化外设

- 设置用于中断的GPIO寄存器：
  - RGPIO_INTE = 0x10000（允许Switch[0]的中断）
  - RGPIO_PTRIG = 0x10000（在Switch[0]的上升沿触发中断）
  - RGPIO_INTS = 0x0（清除所有中断）
  - RGPIO_CTRL = 0x1（允许GPIO中断）

名称	地址	宽度	访问	说明
RGPIO_IN	0x80001400	1-32	R	GPIO输入数据
RGPIO_OUT	0x80001404	1-32	R/W	GPIO输出数据
RGPIO_OE	0x80001408	1-32	R/W	GPIO输出驱动器使能
RGPIO_INTE	0x8000140C	1-32	R/W	**中断允许**
RGPIO_PTRIG	0x80001410	1-32	R/W	**触发中断的事件类型**
RGPIO_AUX	0x80001414	1-32	R/W	将辅助输入与GPIO输出复用
RGPIO_CTRL	0x80001418	2	R/W	**控制寄存器**
RGPIO_INTS	0x8000141C	1-32	R/W	**中断状态**
RGPIO_ECLK	0x80001420	1-32	R/W	使能gpio_eclk以锁存RGPIO_IN
RGPIO_NEC	0x80001424	1-32	R/W	选择gpio_eclk的有效边沿

实验3：练习

- 使用中断读取Switch[0]的值
- 修改程序以包括定时器中断源
  - 通过设置字0x80001018的位1（irq_ptc_enable）可以将定时器中断连接到IRQ3
  - 创建一个初始化PTC中断的函数
  - 创建PTC_ISR，由IRQ3调用
- 可以通过设置优先级阈值与在执行时更改优先级，实现在7段显示屏上显示计数到10后停止计数

目录

- RVfpga课程介绍
- 实验1：RISC-V编程
- 实验2：VeeR EH1流水线
- 实验3：I/O简介与中断驱动I/O
- 实验4：VeeR EH1数据冒险
- 实验5：硬件计数器与基准测试
- 讨论 Q&A

实验4：简介

- 实验4将分析如何解除**RAW数据冒险**。

- RAW(Read After Write)数据冒险：
  - 假设指令i在程序中位于指令j之前，且两条指令均使用寄存器x。
  - 当指令j先读取寄存器x，指令i后写入寄存器x时，将发生RAW情况。

实验4：简介

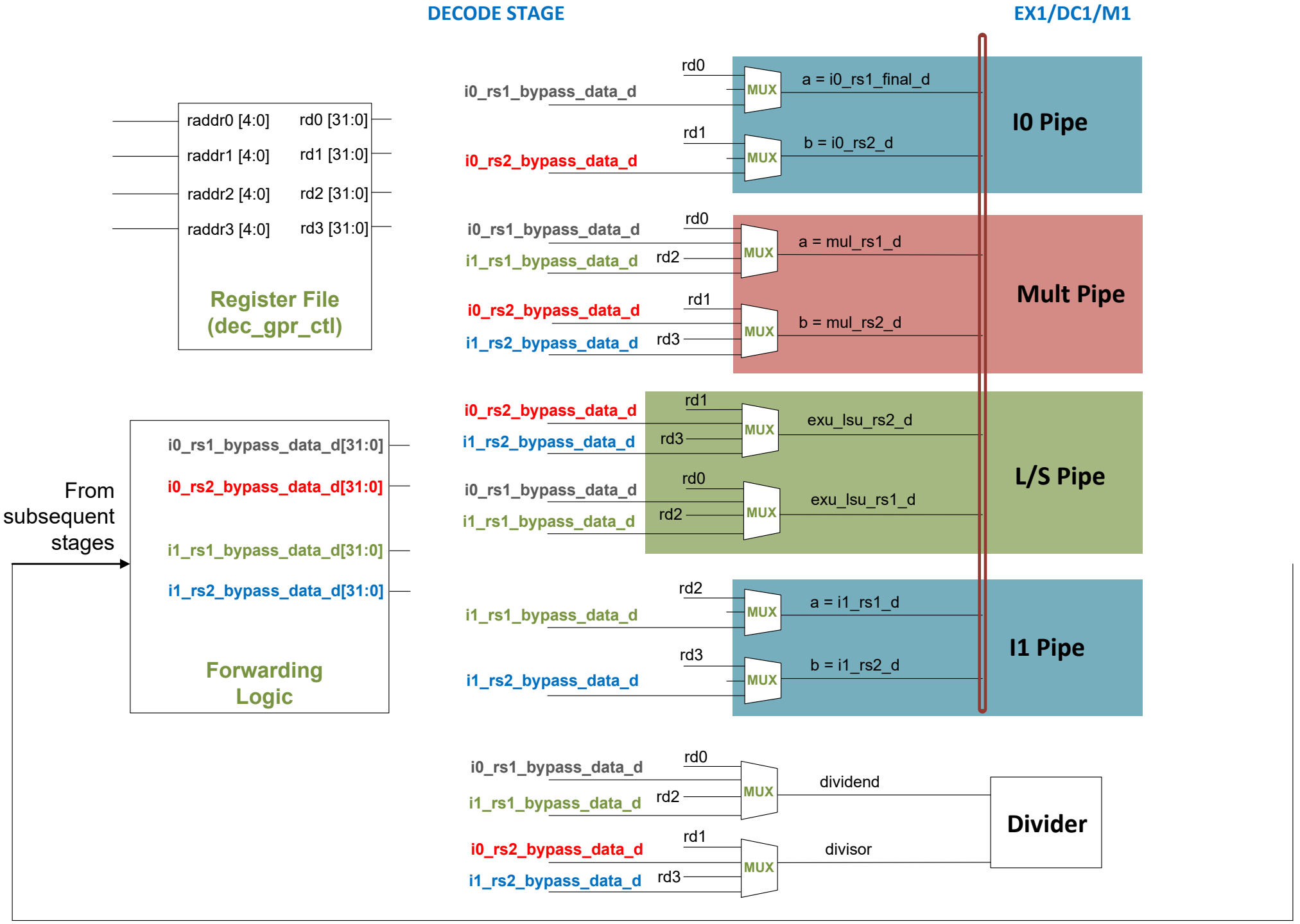
- 解除RAW数据冒险：
  - **暂停**处理器
  - **转发**（也称为旁路）来自后面阶段执行的指令的值

- **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

实验4：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（即转发）。


DECODE STAGE
EX1/DC1/M1
raddr0 [4:0]
raddr1 [4:0]
raddr2 [4:0]
raddr3 [4:0]
rd0 [31:0]
rd1 [31:0]
rd2 [31:0]
rd3 [31:0]
Register File
(dec_gpr_ctl)
From
subsequent
stages
i0_rs1_bypass_data_d[31:0]
i0_rs2_bypass_data_d[31:0]
i1_rs1_bypass_data_d[31:0]
i1_rs2_bypass_data_d[31:0]
Forwarding
Logic
i0_rs1_bypass_data_d
i0_rs2_bypass_data_d
i1_rs1_bypass_data_d
i1_rs2_bypass_data_d
rd0
rd1
rd2
rd3
MUX
a = i0_rs1_final_d
b = i0_rs2_d
I0 Pipe
a = mul_rs1_d
b = mul_rs2_d
Mult Pipe
exu_lsu_rs2_d
exu_lsu_rs1_d
L/S Pipe
a = i1_rs1_d
b = i1_rs2_d
I1 Pipe
dividend
divisor
Divider

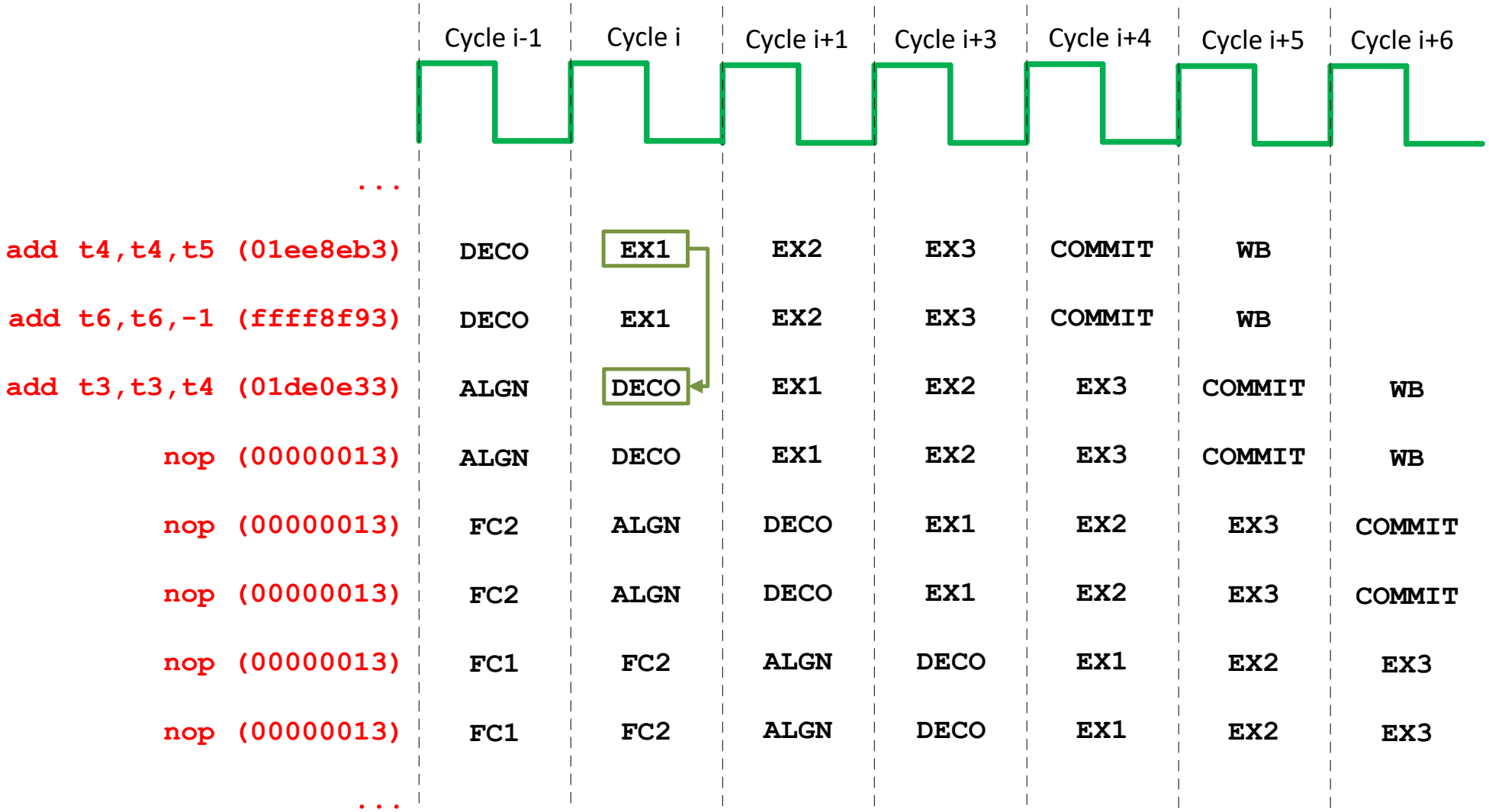
实验4：通过转发解除数据冒险 – 示例

```
.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
```

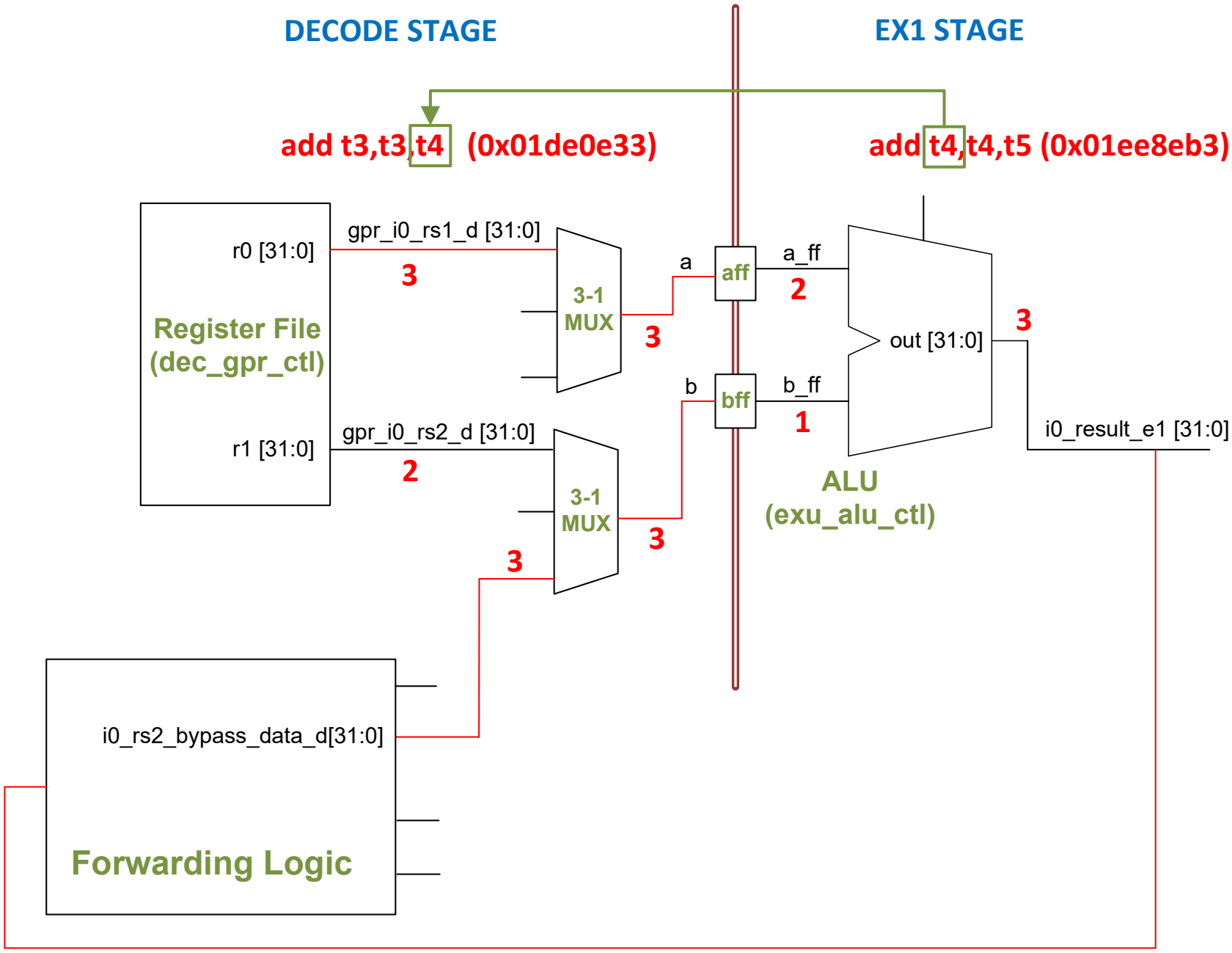
[RVfpgaPath]/RVfpga/Labs/Lab4/DataHazards_AL-AL/.pio/build/swervolf_nexys/firmware.dis

```
00000180 <REPEAT>:
 180:   00000013                nop
 184:   00000013                nop
 188:   00000013                nop
 18c:   00000013                nop
 190:   00000013                nop
 194:   00000013                nop
 198:   00000013                nop
 19c:   00000013                nop
 1a0:   01ee8eb3                add     t4,t4,t5
 1a4:   ffff8f93                addi    t6,t6,-1
 1a8:   01de0e33                add     t3,t3,t4
 1ac:   00000013                nop
 1b0:   00000013                nop
 1b4:   00000013                nop
 1b8:   00000013                nop
 1bc:   00000013                nop
 1c0:   00000013                nop
 1c4:   00000013                nop
 1c8:   00000013                nop
 1cc:   00000013                nop
 1d0:   00300e13                li      t3,3
 1d4:   00200e93                li      t4,2
 1d8:   00100f13                li      t5,1
 1dc:   fa0f92e3                bnez    t6,180 <REPEAT>
```

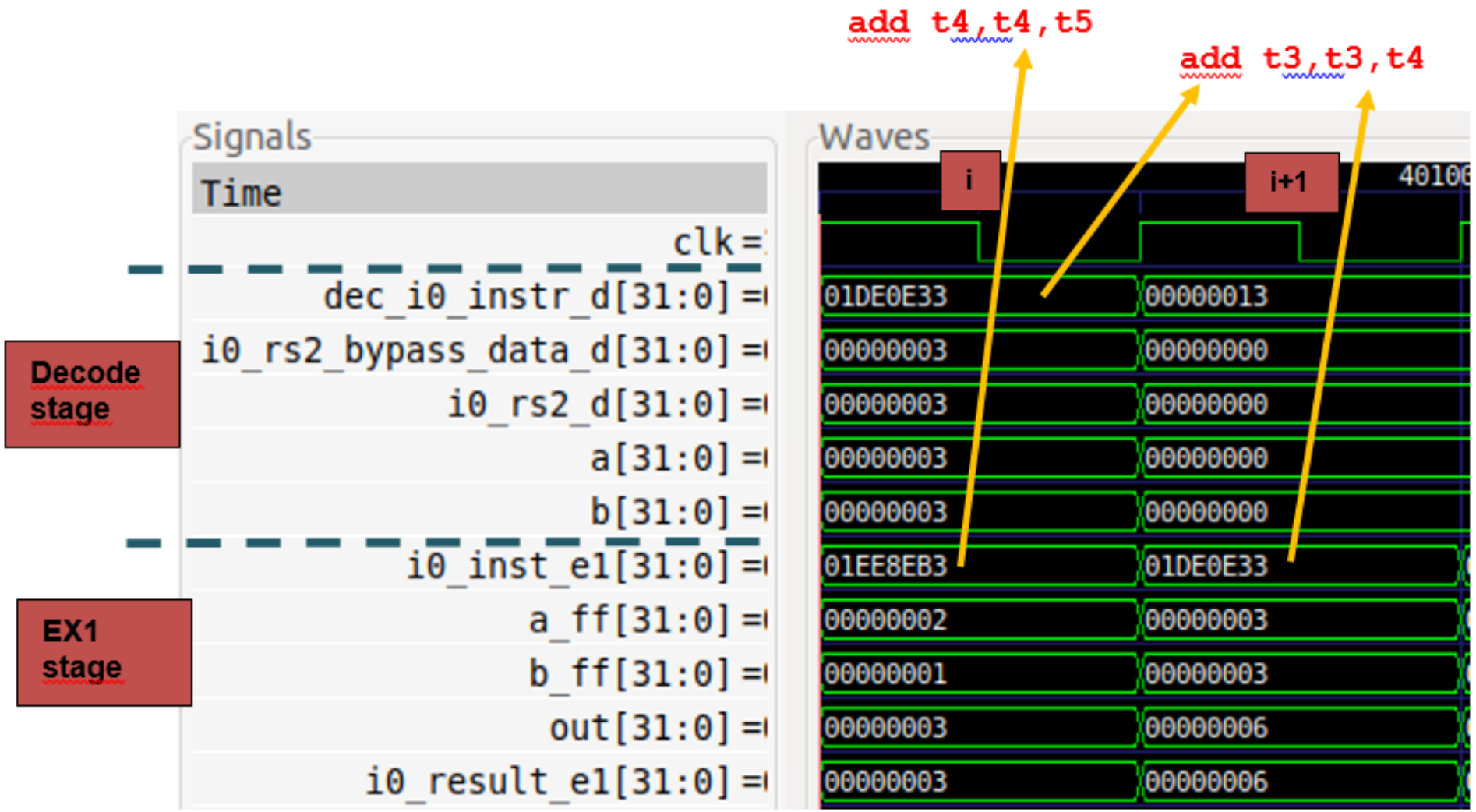
实验4：通过转发解除数据冒险 – 图



实验4：通过转发解除数据冒险 – 流水线

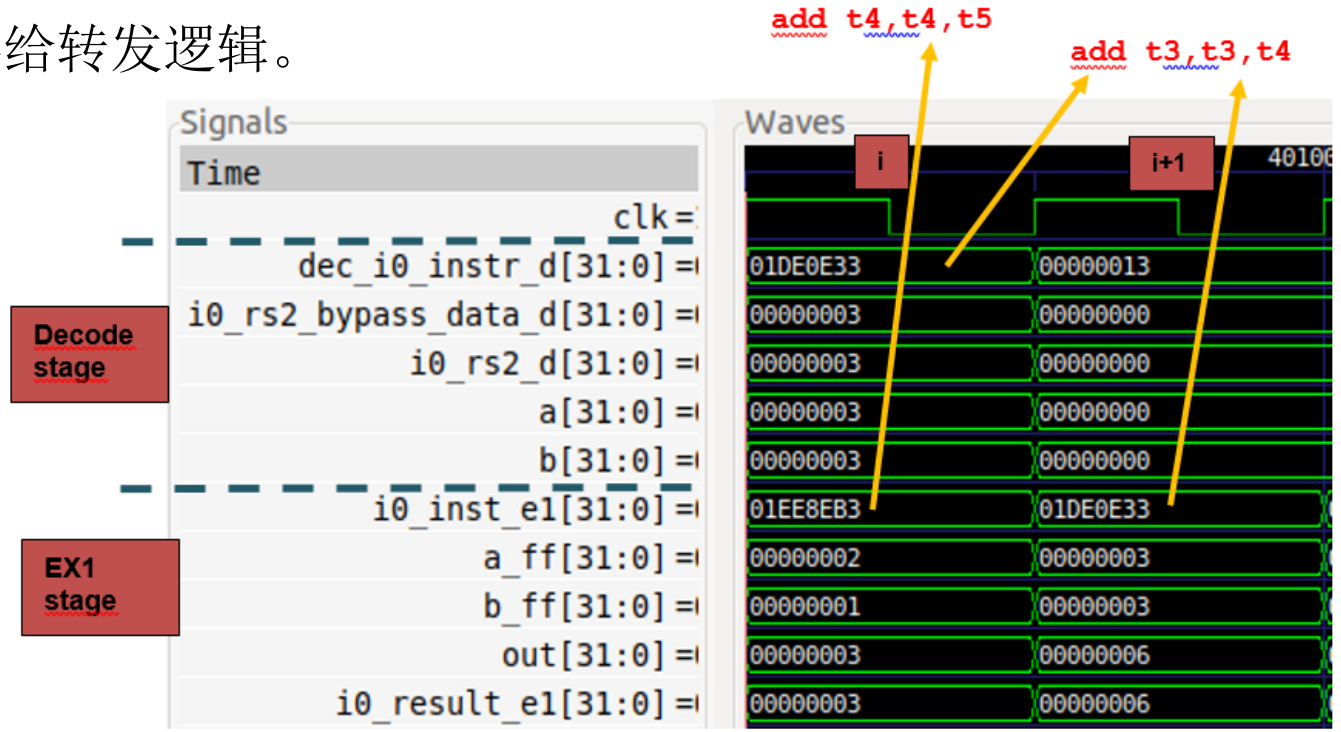


实验4：通过转发解除数据冒险 – 仿真



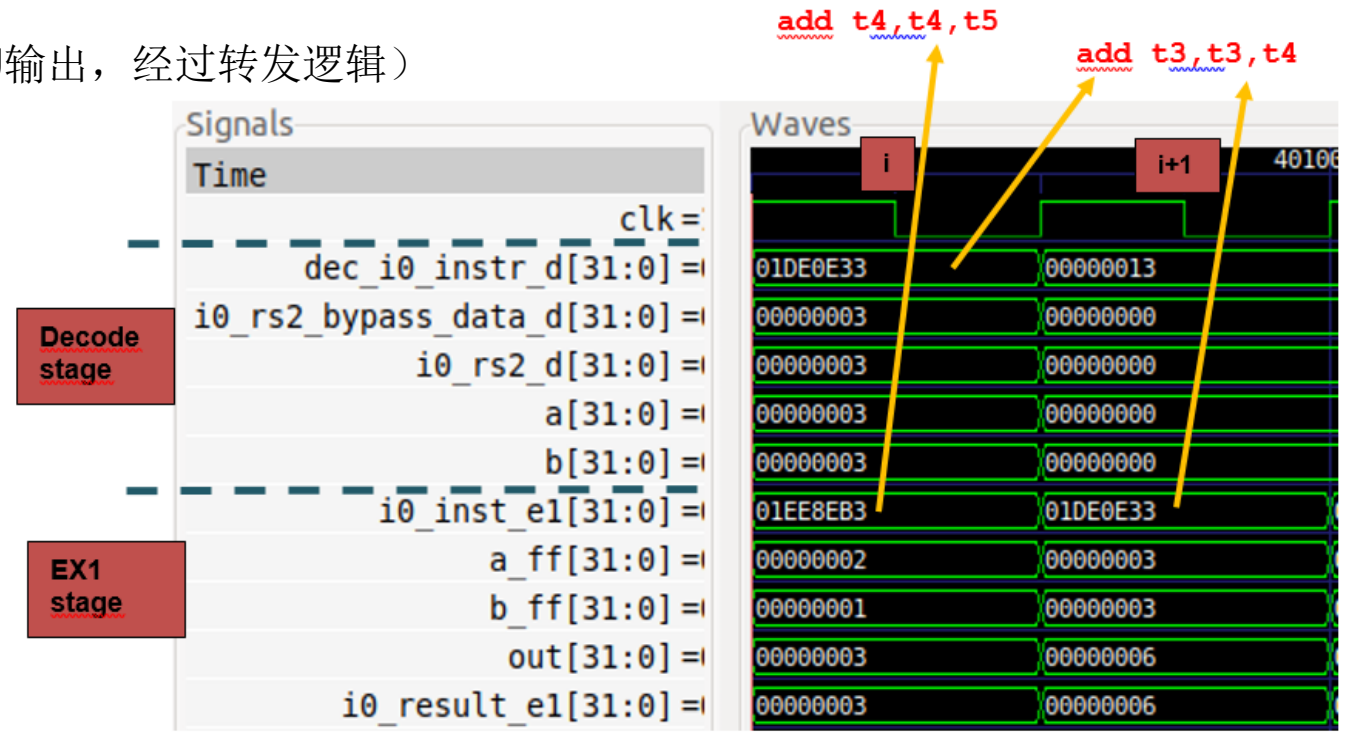
实验4：通过转发解除数据冒险 – 分析

- 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)
  - 加法的结果在译码阶段作为输入提供给转发逻辑。



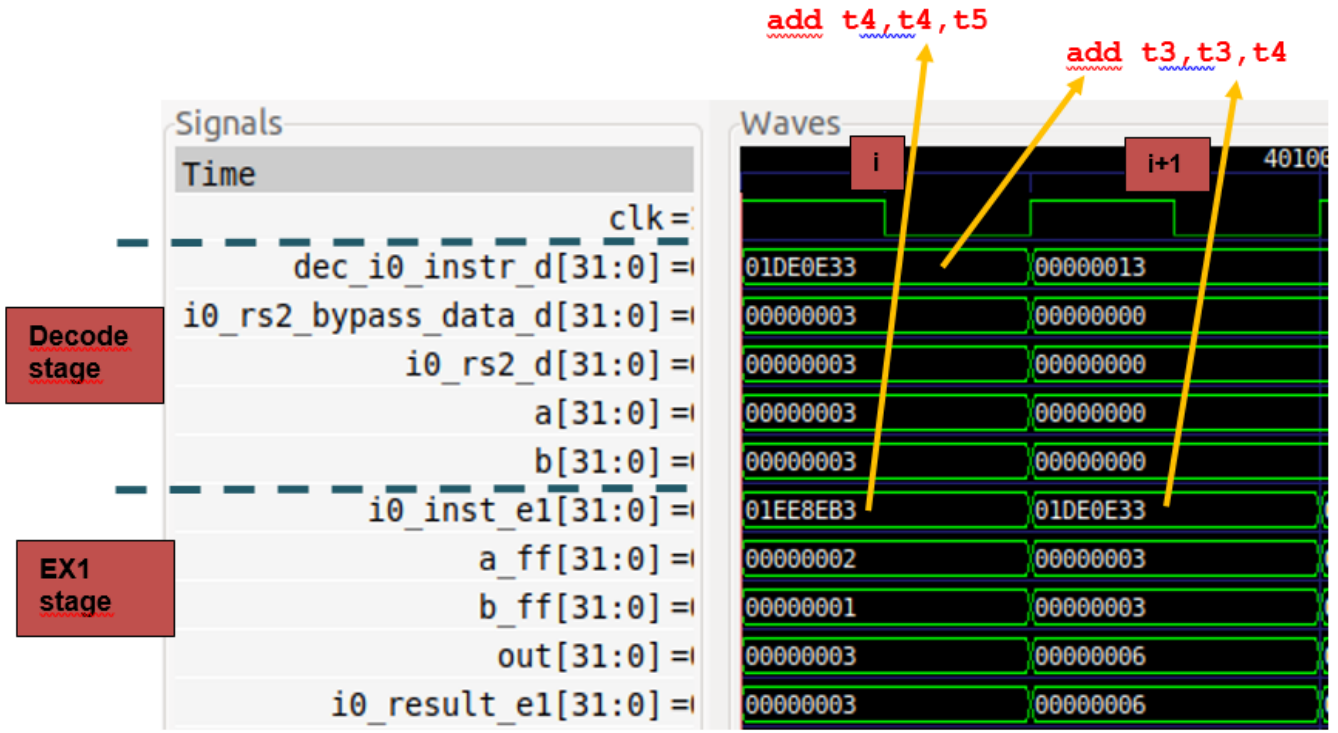
实验4：通过转发解除数据冒险 – 分析

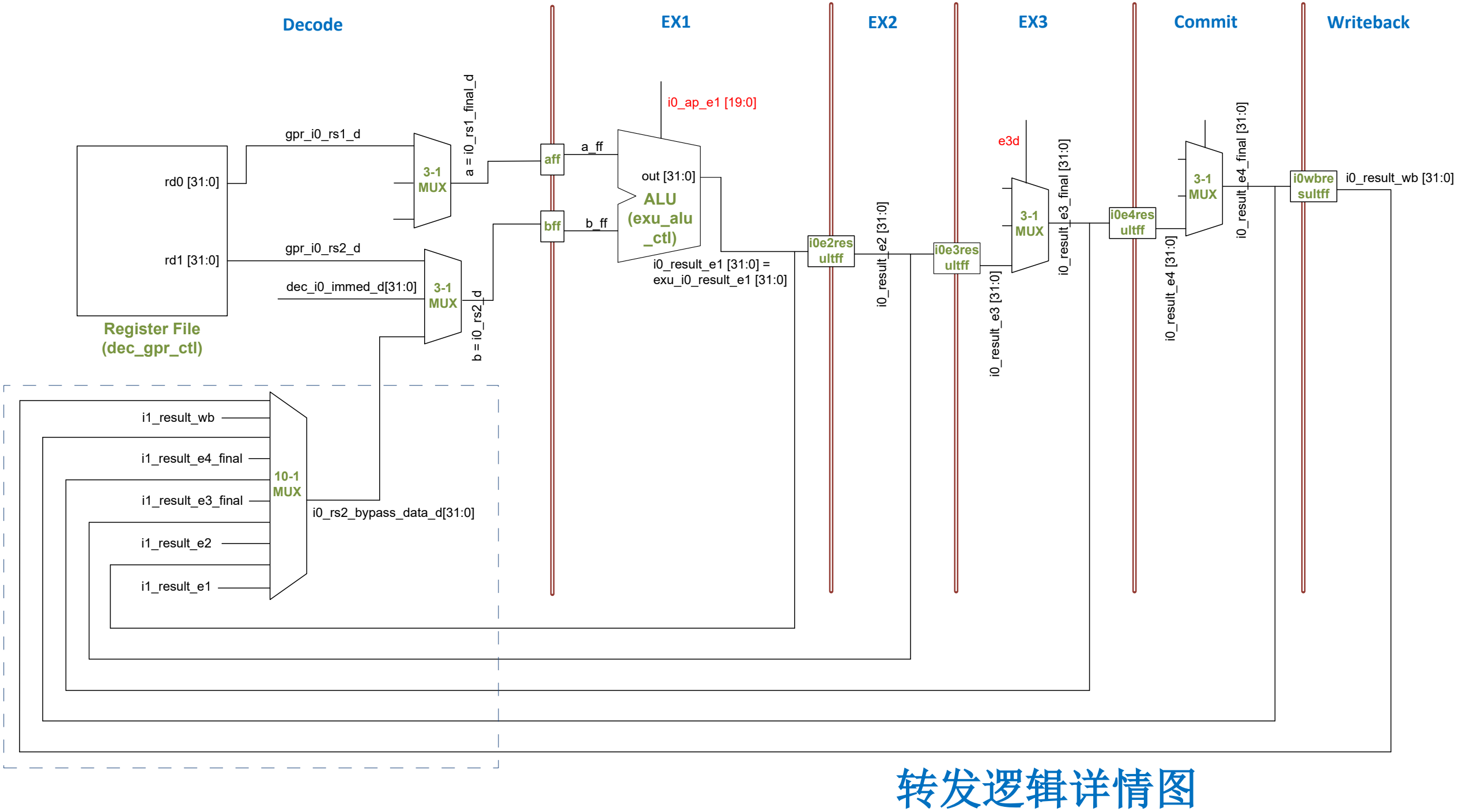
- 指令`add t3,t3,t4`（0x01de0e33）：
  - **周期*i*：**此add指令处于通路0的译码阶段（`dec_i0_instr_d` = 0x01de0e33）。转发逻辑将EX1阶段的结果（`i0_result_e1`）转发到译码阶段（`i0_rs2_bypass_data_d`）。两个3:1多路开关生成操作数，具体如下：
    - 操作数a = 3（来自寄存器文件）
    - 操作数b = 3（来自I0管道EX1阶段的ALU输出，经过转发逻辑）



实验4：通过转发解除数据冒险 – 分析

- 指令`add t3,t3,t4`（0x01de0e33）：
  - **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)




Decode
EX1
EX2
EX3
Commit
Writeback
rd0 [31:0]
rd1 [31:0]
Register File
(dec_gpr_ctl)
gpr_i0_rs1_d
gpr_i0_rs2_d
dec_i0_immed_d[31:0]
3-1
MUX
a = i0_rs1_final_d
b = i0_rs2_d
aff
bff
a_ff
b_ff
i0_ap_e1 [19:0]
out [31:0]
ALU
(exu_alu
_ctl)
i0_result_e1 [31:0] =
exu_i0_result_e1 [31:0]
i0e2res
ultff
i0_result_e2 [31:0]
i0e3res
ultff
e3d
i0_result_e3 [31:0]
i0_result_e3_final [31:0]
i0e4res
ultff
i0_result_e4 [31:0]
i0_result_e4_final [31:0]
i0wbre
sultff
i0_result_wb [31:0]
i1_result_wb
i1_result_e4_final
i1_result_e3_final
i1_result_e2
i1_result_e1
10-1
MUX
i0_rs2_bypass_data_d[31:0]

转发逻辑详情图

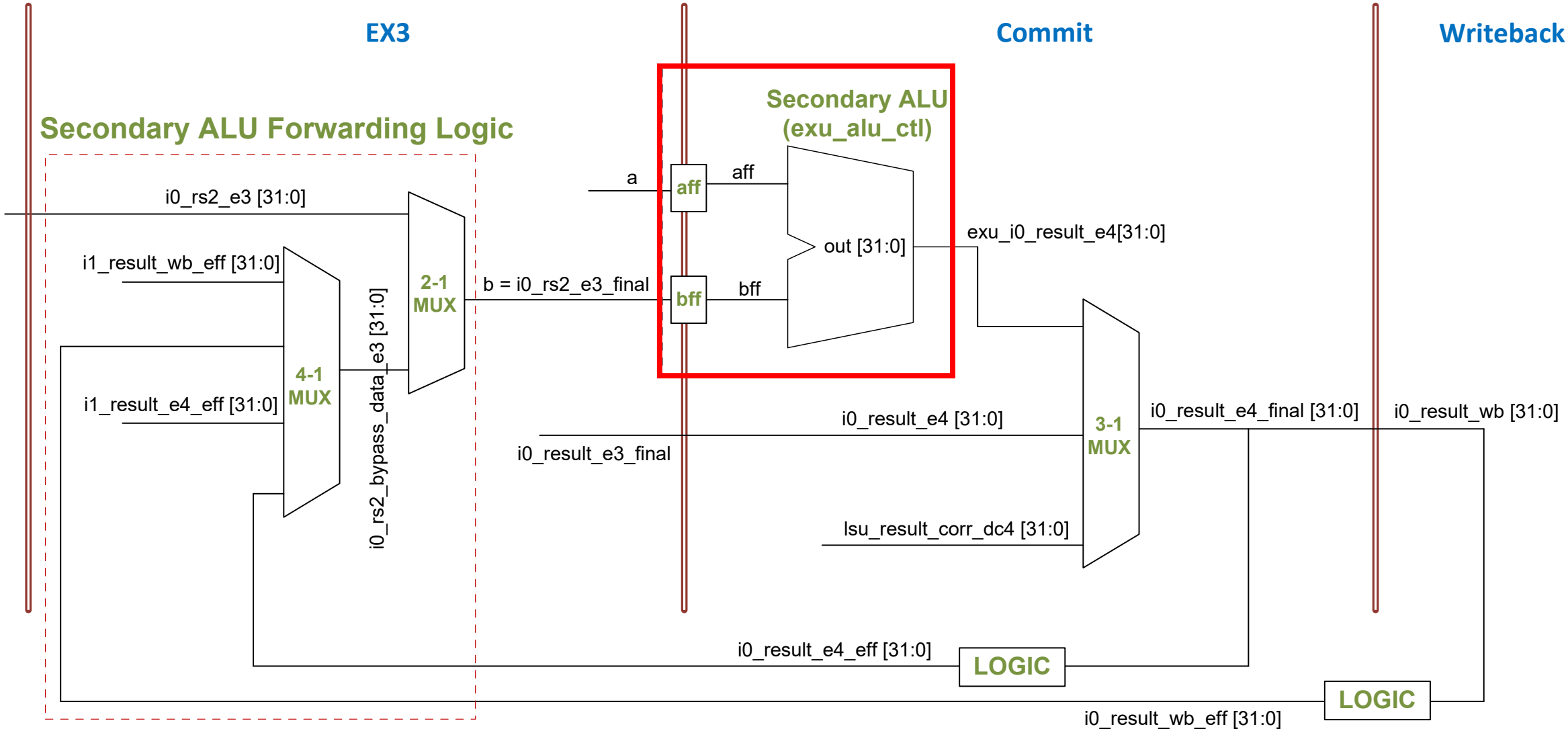
实验4：通过在提交阶段转发解除数据冒险

- 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VeeR 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。

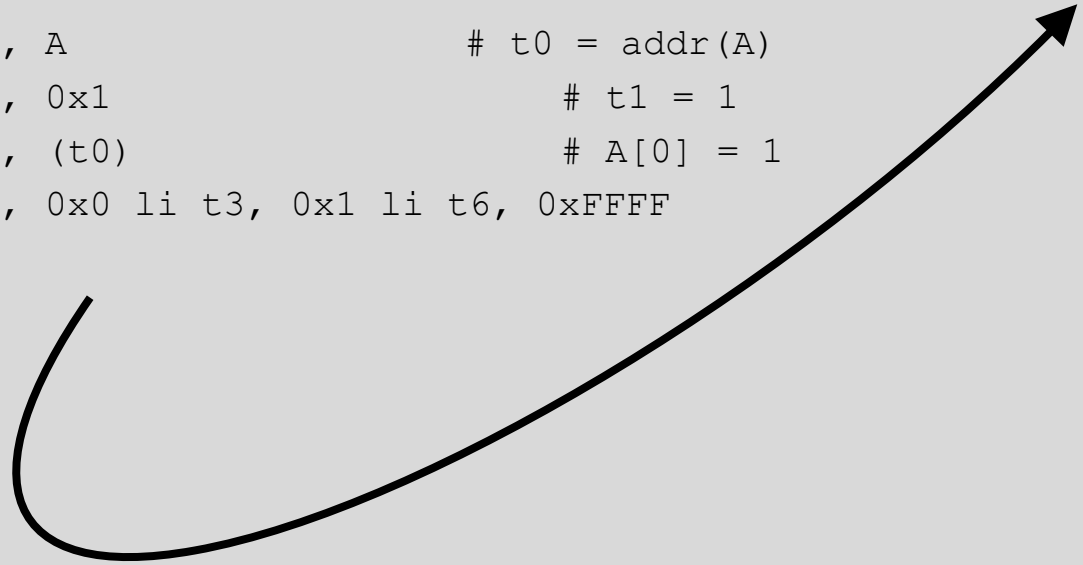
- 因此，**不会**因暂停而丢失周期 – 但代价是添加了**两个额外的ALU**（每路一个）以及控制信号和逻辑。

实验4：通过在提交阶段转发解除数据冒险 – 流水线



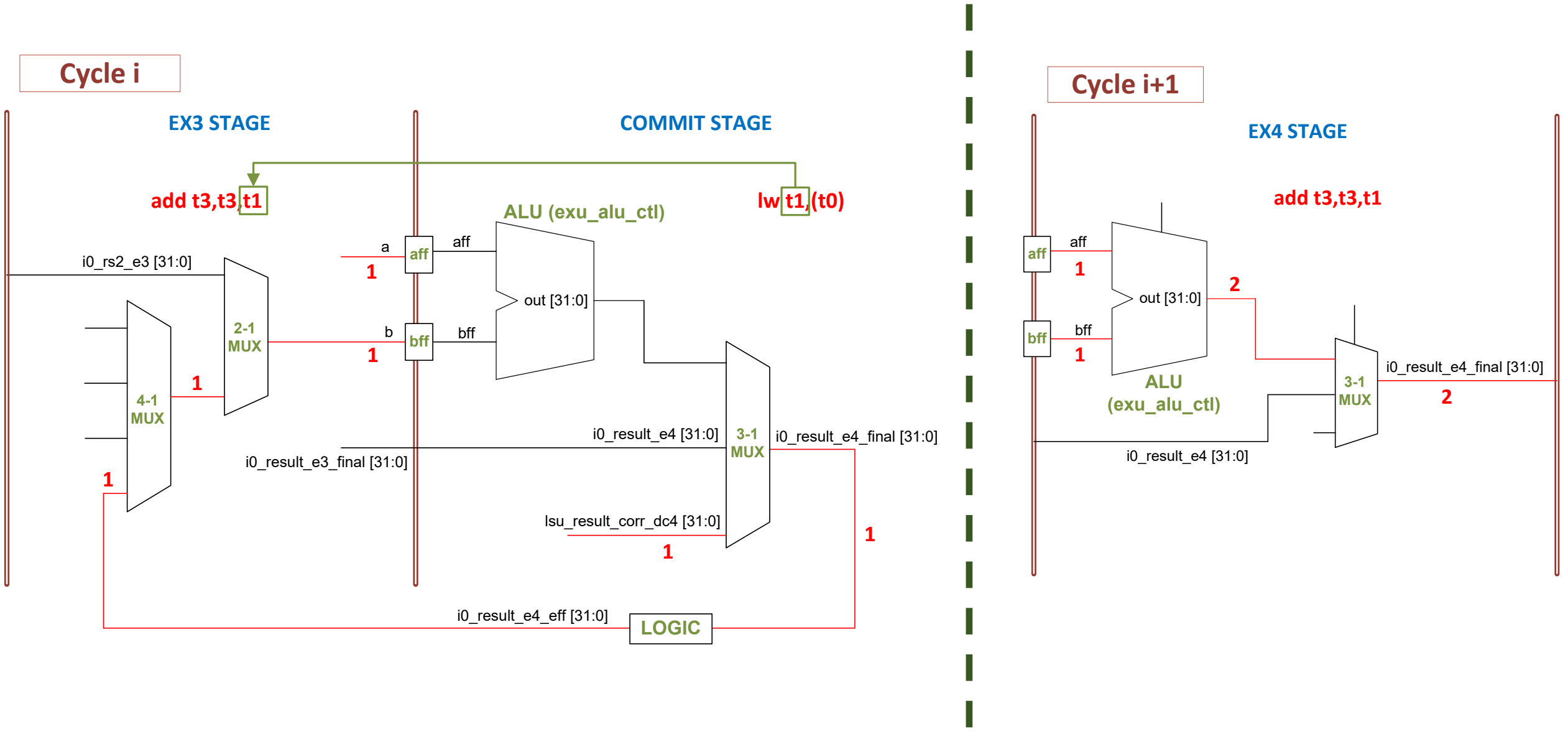
实验4：通过在提交阶段转发解除数据冒险 – 示例

```
.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
```

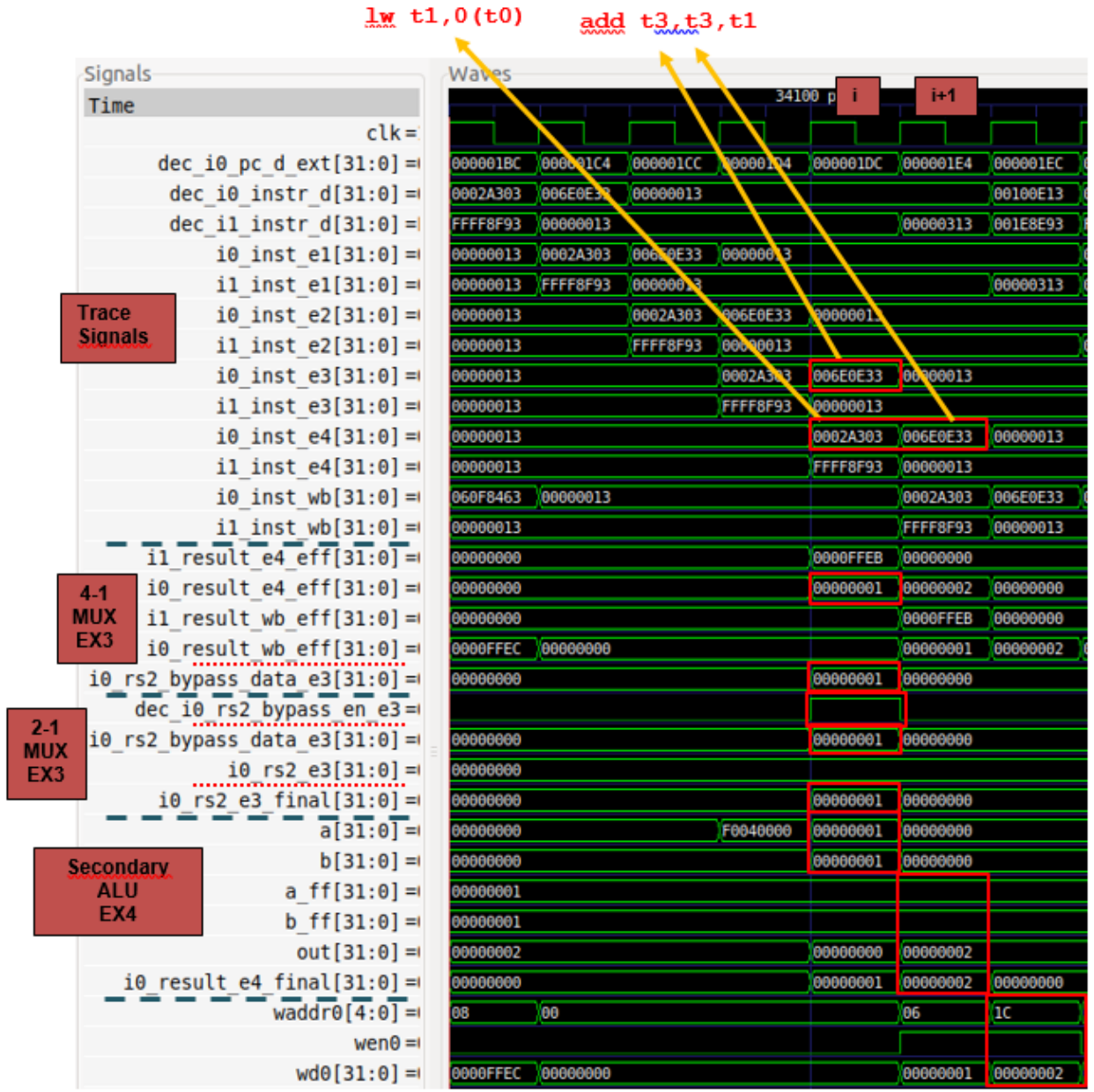


```
REPEAT:
   beq t6, zero, OUT       # Stay in the loop?
   INSERT_NOPS_9
   lw t1, (t0)
   add t6, t6, -1
   add t3, t3, t1              # t3 = t3 + t1
   INSERT_NOPS_8
   li t1, 0x0
   li t3, 0x1
   add t4, t4, 0x1
   add t5, t5, 0x1
j REPEAT

OUT:

.end
```

实验4：通过在提交阶段转发解除数据冒险 – 流水线



实验4：通过在提交阶段转发解除数据冒险 – 仿真



实验4：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（i0_inst_e3 = 0x006E0E33），lw指令处于I0管道的提交阶段（i0_inst_e4 = 0x0002A303）。
  - 周期*i+1*：add指令处于通路0的提交阶段（i0_inst_e4 = 0x006E0E33）。
- 4:1多路开关
  - 周期*i*：选择lw指令的结果（在提交阶段）：

    i0_rs2_bypass_data_e3 = i0_result_e4_eff = 0x00000001
- 2:1多路开关
  - 周期*i*：由于lw和add之间的相关性，选择旁路值：

    i0_rs2_e3_final = i0_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
- 3:1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4：练习 – 示例

- **练习 1.** 对于第一种情况（**通过转发解除数据冒险**）中的程序，针对两条相互依赖的指令彼此之间距离不同的情况执行与图8中相同的分析。可以通过更改两条相关add指令之间的nop数量来控制距离。
- 此外，需创建第一个输入操作数接收转发数据的其他示例。
- 还可以创建两条add指令通过I1管道执行的其他示例，但需确认行为相同。
- 最后，将相关add指令（add t3,t3,t4）替换为通过其他管道执行的其他相关指令并分析仿真结果。例如，可以包含以下指令之一来代替第二条add指令：
  - lw t3, (t4)（强制读取值来自DCCM）
  - mul t3, t3, t4
  - div t3, t3, t4

- **练习 2.** 在自己的计算机上重复第二种情况（**通过在提交阶段转发解除数据冒险**）的仿真过程。可以使用以下位置提供的.tcl文件：
- [RVfpgaPath]/Rvfpga/Labs/Lab4/Solutions/DataHazards_Close-LW-AL/scriptLoad.tcl。

- **练习 3.** 分析lw指令后跟与装载读取的值相关的mul指令这种情况。在第二种情况的程序中，只需将相关add指令替换为mul指令。

茶歇 15:30-15:50

目录

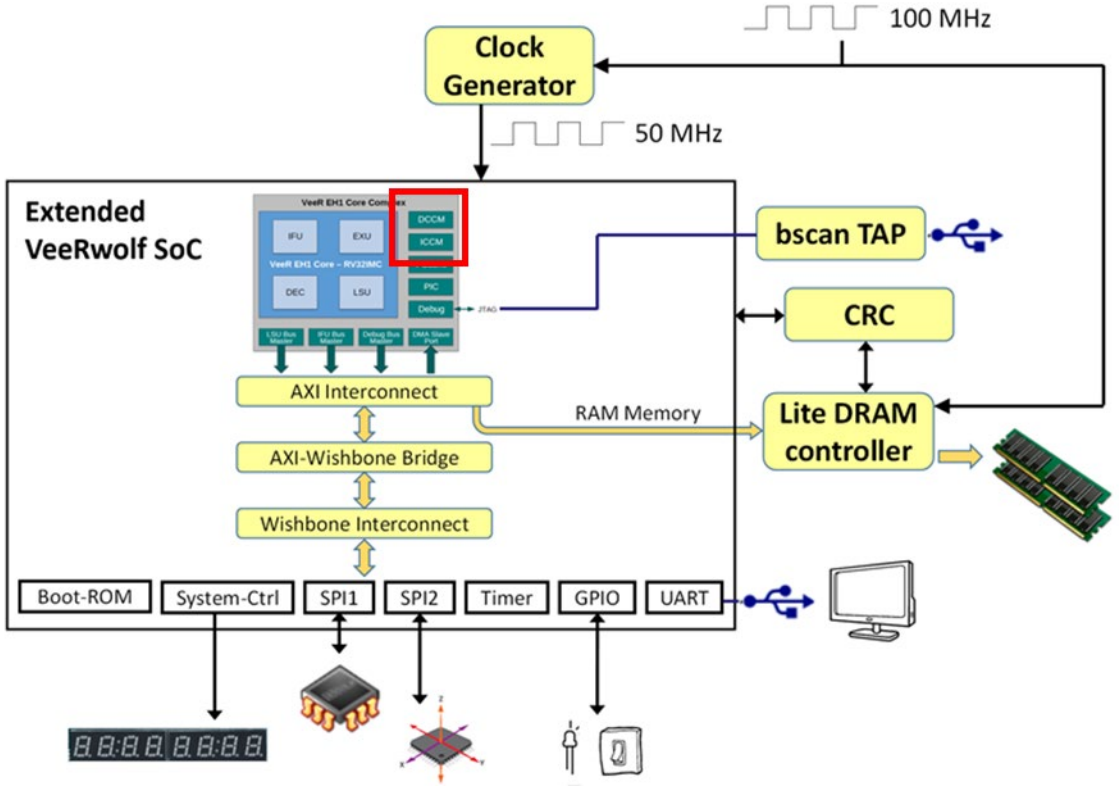
- RVfpga课程介绍
- 实验1：RISC-V编程
- 实验2：VeeR EH1流水线
- 实验3：I/O简介与中断驱动I/O
- 实验4：VeeR EH1数据冒险
- 实验5：硬件计数器与基准测试
- 讨论 Q&A

实验5：实验概述

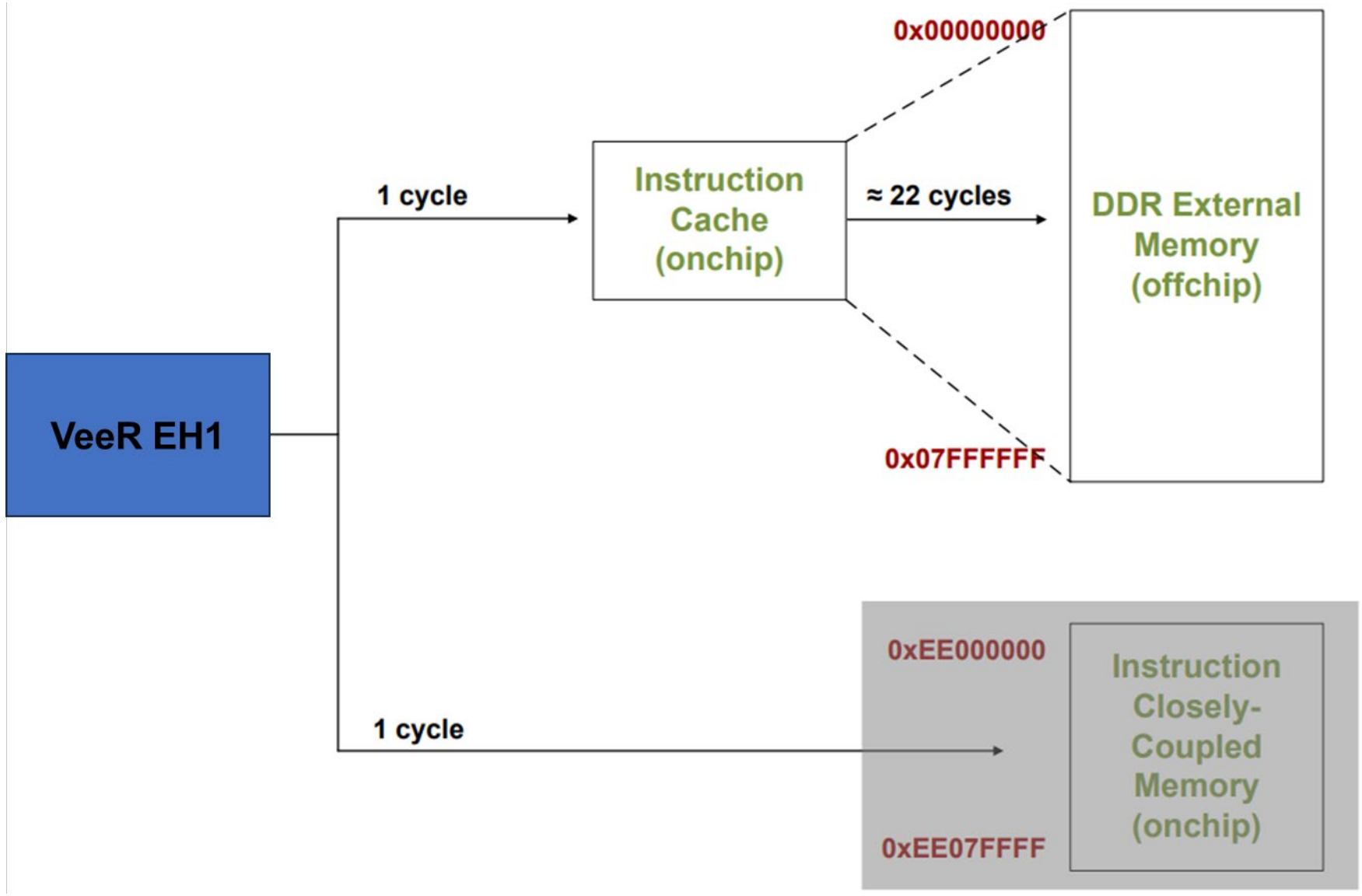
- 分析 EH1 处理器中提供的便笺式存储器（ICCM和DCCM）
- 使用硬件计数器测量处理器事件，分析处理器性能
- 利用基准测试 **CoreMark** 和 **Dhrystone** 测定处理器性能

实验5：便笺式存储器

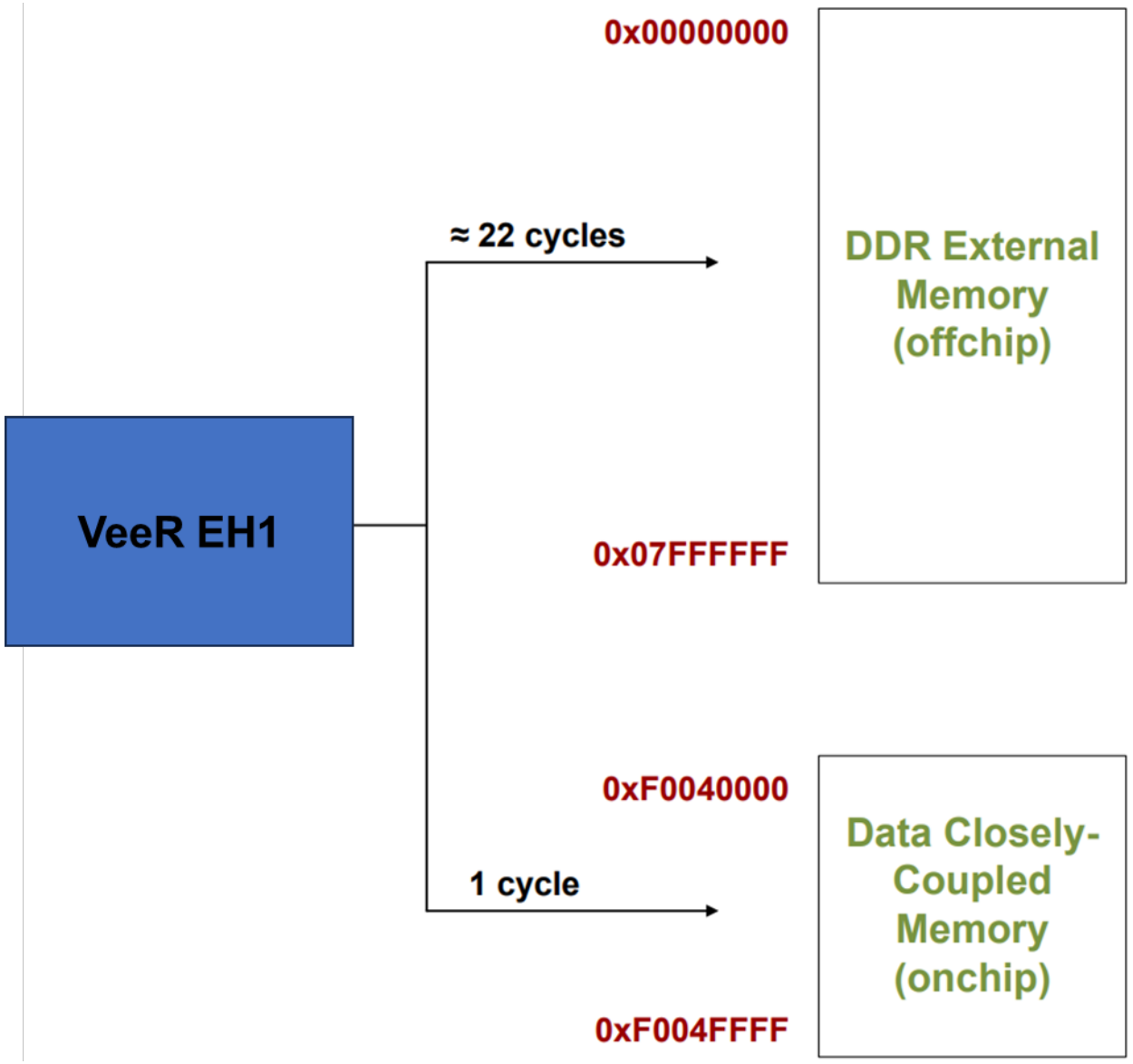
- RVfpga系统包括两个便笺式存储器
  - 一个用于存储数据，称为数据紧密耦合存储器（Data Closely-Coupled Memory，DCCM）
  - 一种用于存储指令，称为指令紧密耦合存储器（Instruction Closely-Coupled Memory，ICCM）



实验5：指令存储器 – 地址空间



实验5：数据存储器 – 地址空间



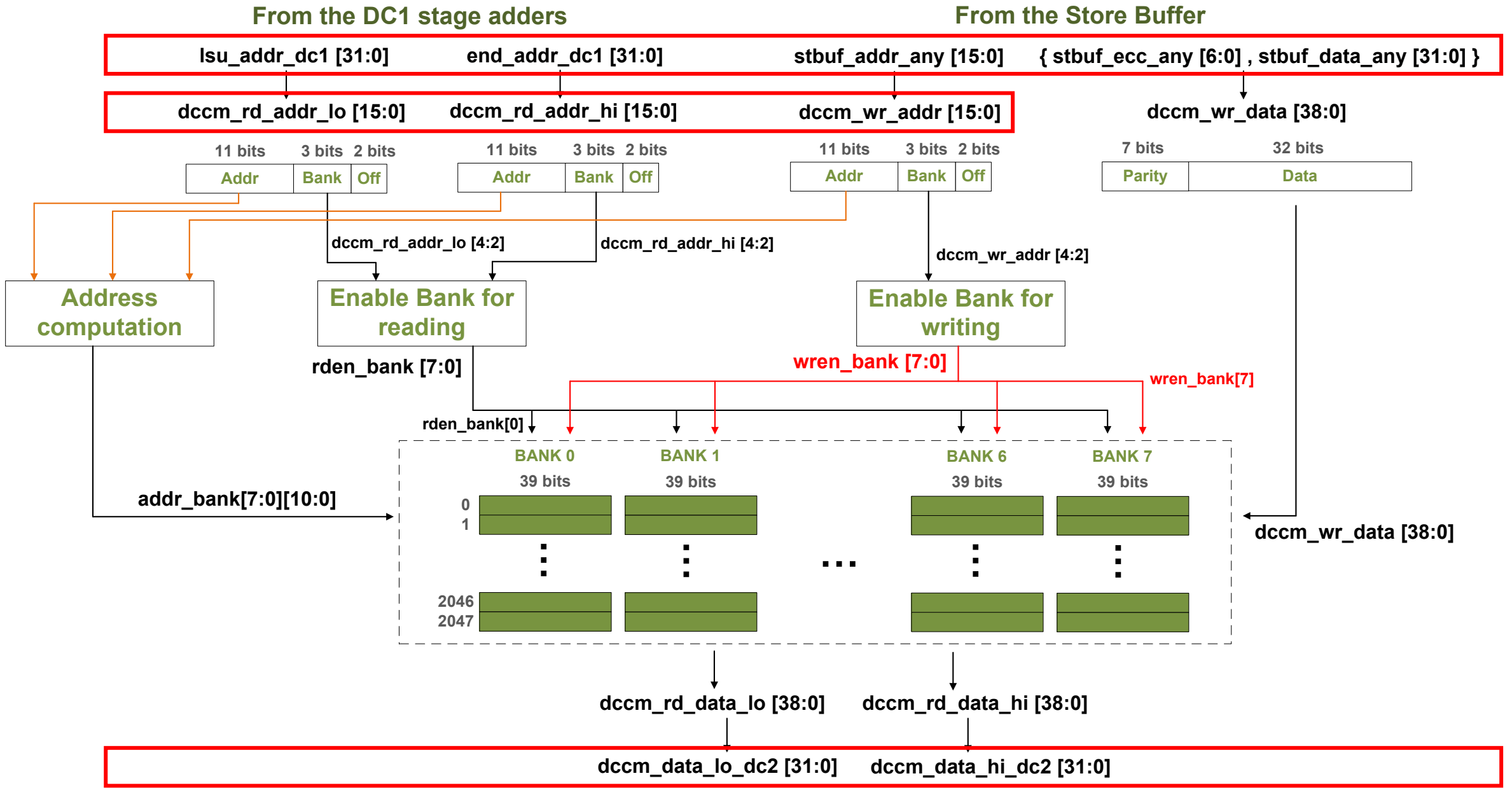
实验5：DCCM配置和操作

- RVfpga系统的DCCM和ICCM可使用文件[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/include/common_defines.vh中定义的一系列参数进行多种配置

- 下图总结了RVfpga系统中的ICCM和DCCM配置

特性	值
DCCM	
使能位	1
地址空间	0xF0040000 – 0xF004FFFF
大小	64 KiB
存储区数量	8
存储区大小	2048x39位（有7位为奇偶校验位）
ICCM	
使能位	0

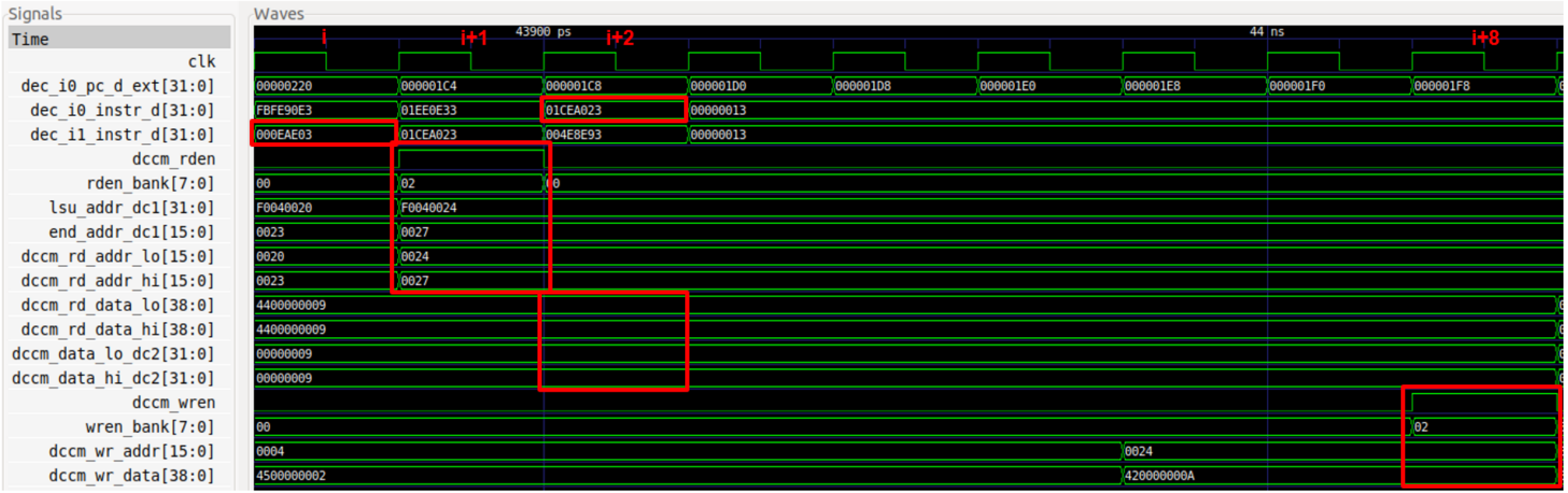
实验5：DCCM内部设计



实验5：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
```

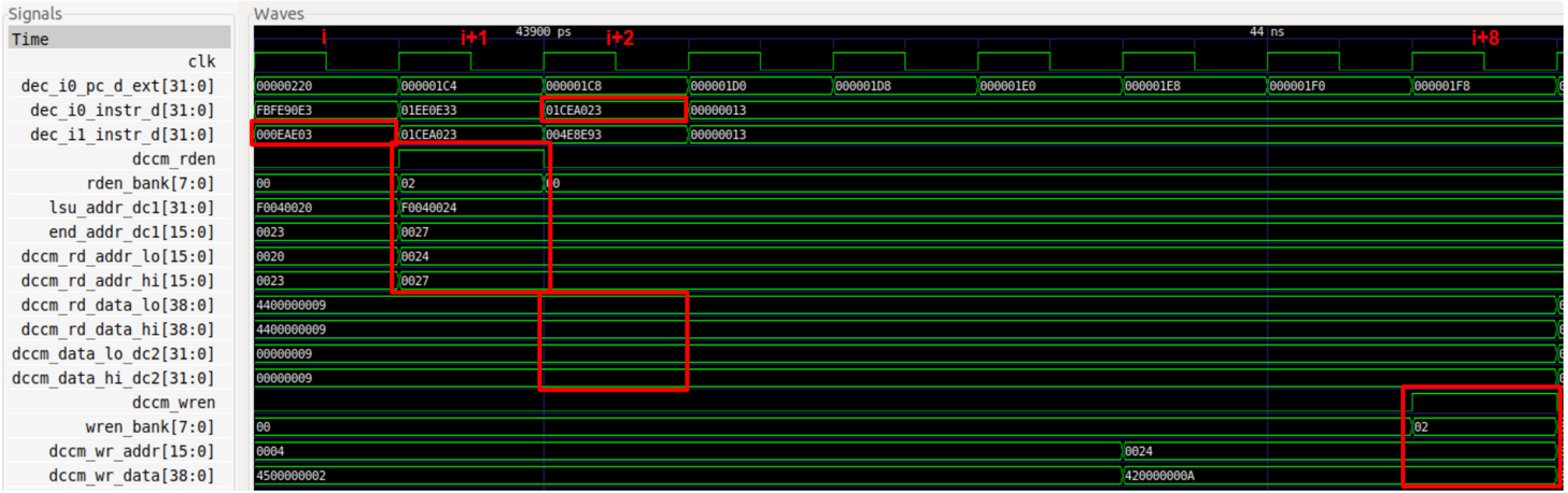
实验5：DCCM – 仿真与分析



- **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

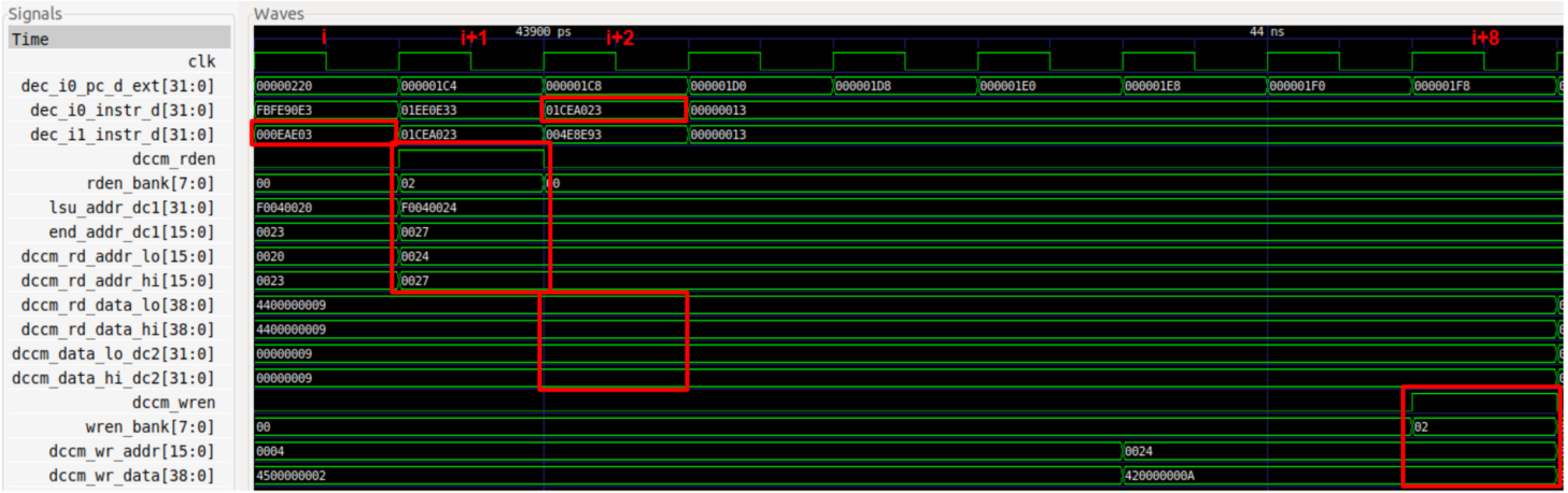
  完成地址检查后，将使能DCCM的读操作：dccm_rden = 1。由于访问是字对齐的，因此仅读取第二个存储区：rden_bank = 0x02（二进制值为00000010）。

实验5：DCCM – 仿真与分析



- **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

实验5：DCCM – 仿真与分析



- **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

实验5：基准测试

- **基准测试**
  - 运行程序（或程序集）测定处理器性能
  - 比较不同处理器性能
- **常用基准测试**
  - CoreMark
  - Dhrystone
- **针对 RVfpga 适配**
  - 硬件计数器：测量各种处理器事件
  - 使用DCCM/ICCM和编译器优化的支持

实验5：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，用于记录各种指标

0	保留	17	CSR读/写	34	SB/WB周期暂停
1	时钟周期有效	18	CSR写/读==0	35	DMA DCCM事务周期暂停
2	指令高速缓存命中	19	Ebreak	36	DMA ICCM事务周期暂停
3	指令高速缓存未命中	20	Ecall	37	发生异常
4	指令已提交	21	Fence	38	发生定时器中断
5	16位指令已提交	22	Fence.i	39	发生外部中断
6	32位指令已提交	23	Mret	40	TLU清除
7	指令已对齐	24	分支已提交	41	分支错误清除
8	指令已译码	25	分支预测错误	42	指令总线事务 – 指令
9	乘法已提交	26	进行分支	43	数据总线事务 – ld/st
10	除法已提交	27	分支不可预测	44	数据总线事务未对齐
11	装载已提交	28	取指周期暂停	45	指令总线错误
12	存储已提交	29	对齐器周期暂停	46	数据总线错误
13	装载未对齐	30	译码周期暂停	47	由于指令总线繁忙，周期暂停
14	存储未对齐	31	后同步周期暂停	48	由于数据总线繁忙，周期暂停
15	ALU已提交	32	预同步周期暂停	49	中断周期已禁止
16	CSR读取	33	周期冻结	50	禁止时中断周期暂停

实验5：使用硬件计数器

- 调用 PSP (Platform Support Package)
  - #include <psp_api.h>
- 使能计数器
  - pspEnableAllPerformanceMonitor(1);
- 设置计数器
  - pspPerformanceCounterSet(D_PSP_COUNTER0, E_CYCLES_CLOCKS_ACTIVE);
  - pspPerformanceCounterSet(D_PSP_COUNTER1, E_INSTR_COMMITTED_ALL);
- 读取计数器
  - cyc_end = pspPerformanceCounterGet(D_PSP_COUNTER0);
  - instr_end = pspPerformanceCounterGet(D_PSP_COUNTER1);
- 输出数据
  - printfNexys("Cycles = %d", cyc_end - cyc_beg);
  - printfNexys("Instructions = %d", instr_end - instr_beg);

实验5：CoreMark 硬件计数器

- 在本次基准测试中，均使用硬件计数器来测量各种处理器事件
- 需要配置用于测量以下四个事件的硬件计数器：周期数、指令数、指令总线事务（指令）和数据总线事务（ld/st指令）
  - pspPerformanceCounterSet(D_PSP_COUNTER0, E_CYCLES_CLOCKS_ACTIVE);
  - pspPerformanceCounterSet(D_PSP_COUNTER1, E_INSTR_COMMITTED_ALL);
  - pspPerformanceCounterSet(D_PSP_COUNTER1, E_D_BUD_TRANSACTIONS_LD_ST);
  - pspPerformanceCounterSet(D_PSP_COUNTER1, E_D_BUS_TRANSACTIONS_INSTR);

实验5：指标

- **CoreMark指标**
  - CoreMark会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）
- 由于 EH1为双路超标量处理器，因此其**理想IPC (Instructions per Cycle)**为**2**

实验5：演示

- 情景**1**：无编译器优化或**DCCM/ICCM**
- 情景**2**：使用**DCCM**
- 情景**3**：使用**DCCM**和编译器优化

实验5：各种条件下 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 （DCCM）	0 （DCCM）
指令总线事务	392 （指令缓存）	392 （指令缓存）	504 （指令缓存）

实验5：硬件计数器

- 无编译器优化
  - Cycles = 529930
  - Instructions = 496678
  - Inst Miss = 49
  - Inst Bus Transactions = 392
- O2优化
  - Cycles = 288340
  - Instructions = 309637
  - Inst Miss = 62
  - Inst Bus Transactions = 504
- O3 编译器优化
  - Cycles = 268869
  - Instructions = 292421
  - Inst Miss = 219
  - Inst Bus Transactions = 1760

实验5：练习

- 练习 1) 使用Dhrystone基准替代CoreMark基准，执行相同的分析
  - Dhrystone基准项目地址：*RealBenchmarks/Dhrystone_HwCounter*
- 练习 2) 使用图像处理应用程序替代CoreMark，执行相同的分析
  - 应用程序将RGB图像转换为灰度图像
  - 图像处理应用项目地址：*RealBenchmarks/ImageProcessing_HwCounters*
  - 默认RVfpga系统的DCCM没有足够的空间来存储图像，因此需使用具有128 KiB DCCM 的 Rvfpga系统（*RealBenchmarks/Bitstreams/rvfpganexys_DCCM-128.bit*）
- 练习 3) 使用ICCM执行CoreMark基准测试，并与演示结果对比
  - 在文件 *platformio.ini* 中，使用使能ICCM的链接器脚本（*RealBenchmarks/CoreMark_HwCounters/ld/link_DCCM-ICCM.ld* ）
  - 默认RVfpga系统不包括ICCM ，在文件 *platformio.ini* 中，使用使能ICCM的RVfpga系统（*RealBenchmarks/Bitstreams/rvfpganexys_DCCM-ICCM.bit*）

Q&A