



Imagination大学计划

RVfpga实验11

SweRV EH1配置、结构和性能监视

1.简介

在前10个RVfpga实验（实验1-10）中，我们介绍了RISC-V架构以及如何使用各种外设与SweRV EH1内核通信。在接下来的10个实验（实验11-20）中，我们将深入到微架构级别并分析SweRV EH1处理器的内部运行方式以及缓存/存储器层级的工作方式。

SIGASI STUDIO: 在这些实验中，我们将处理扩展Verilog项目：SweRV EH1 Core RTL。分析各种模块和信号的一种方法是使用Sublime Text等典型编辑器（<https://www.sublimetext.com/>），这类编辑器提供了强大的功能，支持在项目中导航、检查文件、查找字符串等。不过，还有更合适的特定替代方案，例如我们强烈推荐的**Sigasi Studio**（<https://www.sigasi.com/>）。补充文档SweRVref.docx展示了如何安装和使用Sigasi Studio（SweRVref文档的第1部分）。

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。图1给出了SweRV EH1微架构的高级视图。SweRV EH1支持RISC-V的整数指令（I）、压缩指令（C）和整数乘法指令（M）扩展。它拥有极高的每MHz性能（4.9 CM/MHz），这得益于其包含多种微架构技术，从流水线和指令高速缓存等最基本、最常见的技术到超标量执行、非阻塞装载和除法、两个可在出现数据冒险的必要情况下重复算术逻辑指令的辅助ALU（有关详细信息，请参见实验15）、未对齐的装载和存储、指令和数据的暂存存储器以及高级分支预测等其他更具体、更先进的技术。上述所有技术都将在这些实验中进行广泛分析。

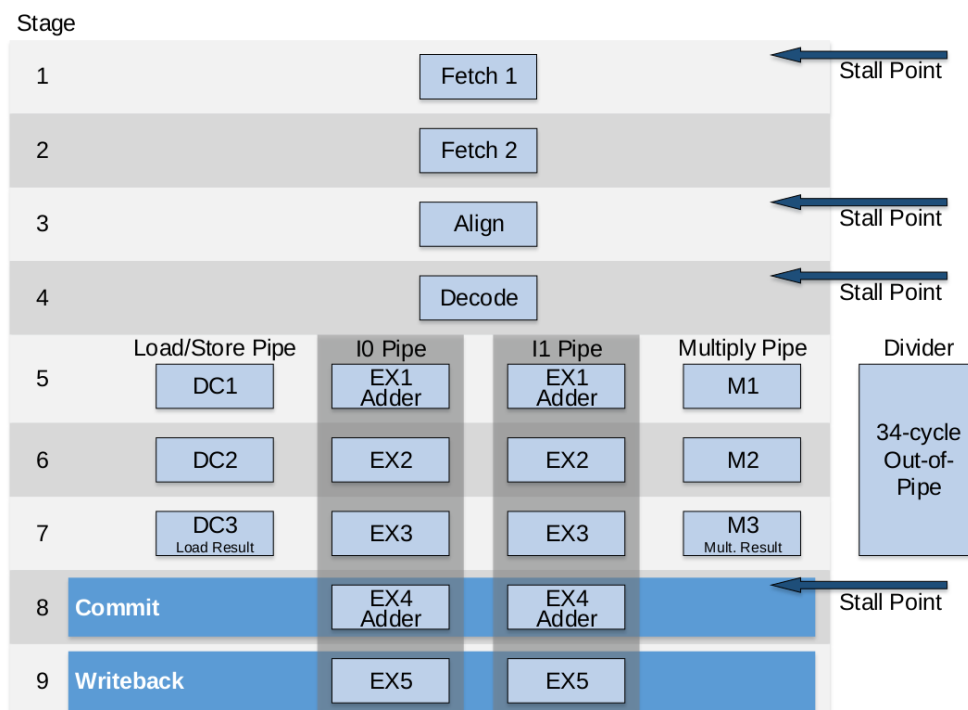


图1. SweRV EH1内核微架构

（图来自[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)）

注：在开始这组实验之前，我们建议您仔细阅读教材《数字设计和计算机体系结构：RISC-V 版本》的第7章和第8章，作者为S. Harris和D. Harris（Morgan Kaufmann © 2021）。这些实验的一些内容受这本书的启发。我们将这本书称为DDCARV。

大多数实验分为两部分：分别是基础部分和高级部分。此外，鉴于实际处理器（例如SweRV EH1）某些部分的复杂度较高，一些详细信息已移至给定实验的附录中。这样，用户便可选择只完成基础部分、完成基础部分和高级部分，甚至深入研究附录以了解处理器更复杂的部分。

实验11-20从概念的理论说明开始，随后使用图和示例程序的Verilator仿真来说明概念。这些示例程序仅用于说明概念。我们还提供相关练习以加深对所描述概念的理解和体验。

根据课程的目标和深度，学员有可能只完成实验的一小部分。以下实验涵盖流水线、存储器构成和高级微架构/存储器层级的概念：

- **流水线：**实验11、12、14、15和实验16的第一部分（分支指令）
- **存储器：**实验11、13和19
- **高级微架构和存储器层级：**实验17、18、20和实验16的第二部分（分支预测器）

在本实验（实验11）中，我们开始分析SweRV EH1处理器。具体来说：

- **第2部分**介绍Verilog RTL结构和每个流水线阶段的详细信息。
- **第3部分**展示如何使用性能计数器来分析处理器性能。

补充文档（**SwerRVref.docx**）介绍以下内容：

- **第1部分：**Sigasi Studio的应用。
- **第2部分：**SweRV EH1处理器的配置。
- **第3部分：**模块的RVfpga系统层级及其最相关的信号
- **第4部分：**用于对控制位进行分组的结构/类型
- **第5部分：**RISC-V压缩指令
- **第6部分：**实际基准

在此初始方法后，我们会在实验12-20中将此分析扩展到不同处理器单元。具体来说：

- **实验12**重点关注**算术逻辑**指令，实验期间将更深入地了解译码、EX1/EX2/EX3和回写阶段。
- **实验13**介绍**访存指令**（装载和存储），实验期间重点关注DC1/DC2/DC3阶段。
- **实验14**探讨了**结构冒险**，实验期间重点关注3周期流水线乘法指令和与非阻塞装载相关的特定情况。该实验还将分析附录中的34周期非流水线除法指令。
- **实验15**分析**数据冒险**，实验期间将介绍处理器的旁路路径。

- **实验16**介绍**控制冒险**、分支指令和分支预测器，将重点关注SweRV EH1处理器的取指1和取指2两个阶段。
- 在之前的实验中，大多数情况下只使用1路处理器，但**实验17**介绍2路超标量处理器，例如SweRV EH1。
- **实验18**是一个实践实验，期间将向SweRV EH1内核添加新指令和硬件计数器。
- **实验19**和**20**重点关注处理器中的各个低延时存储器：指令高速缓存（**IS**）以及紧密耦合的指令和数据存储器（**ICCM**和**DCCM**）。

2. SweRV EH1微架构的初始近似

DDCARV中介绍的处理器包含5个流水线阶段，分别称为**取指**、**译码**、**执行**、**访存**和**回写**阶段。相比之下，SweRV EH1流水线分为9级（图1）：**取指1**、**取指2**、**对齐**、**译码**、**EX1/DC1/M1**、**EX2/DC2/M2**、**EX3/DC3/M3**、**提交**和**回写**级。比较两个处理器时，有些阶段是等效的，例如译码和回写阶段。但SweRV EH1添加了并行路径（装载/存储、整数与乘法管道），将一些阶段分成多个阶段（**取指**为2个阶段，**执行**为3个阶段），并添加了一些阶段（**提交**和**对齐**阶段）。

本节的其余部分介绍Verilog RTL结构和每个流水线阶段的详细信息。A部分介绍SweRV EH1 Verilog模块的层级。B部分和C部分逐阶段讨论SweRV EH1的微架构。最后，D部分提供B部分和C部分中给出的理论说明的实际示例。

SWERV EH1处理器的配置： SweRV EH1处理器的许多结构和功能都可以配置或使能/禁止。补充文档SweRVref.docx在第2部分中解释了这些不同的选项，实验12-20中将经常使用这些选项。

A. SweRV EH1 Verilog模块的层级

图2给出了构成SweRV EH1处理器的主要Verilog模块（某些模块未包含在图中）的层级。该图扩展了GSG的图29，后者给出了构成RVfpga系统的Verilog模块的层级。这些模块位于同名文件中，位置如下：`[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex`目录。

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**IS**。**swerv**模块为整体CPU；其对构成SweRV EH1处理器的模块进行实例化：**取指单元（ifu）**、**译码单元（dec）**、**执行单元（exu）**、**装载/存储单元（lsu）**...

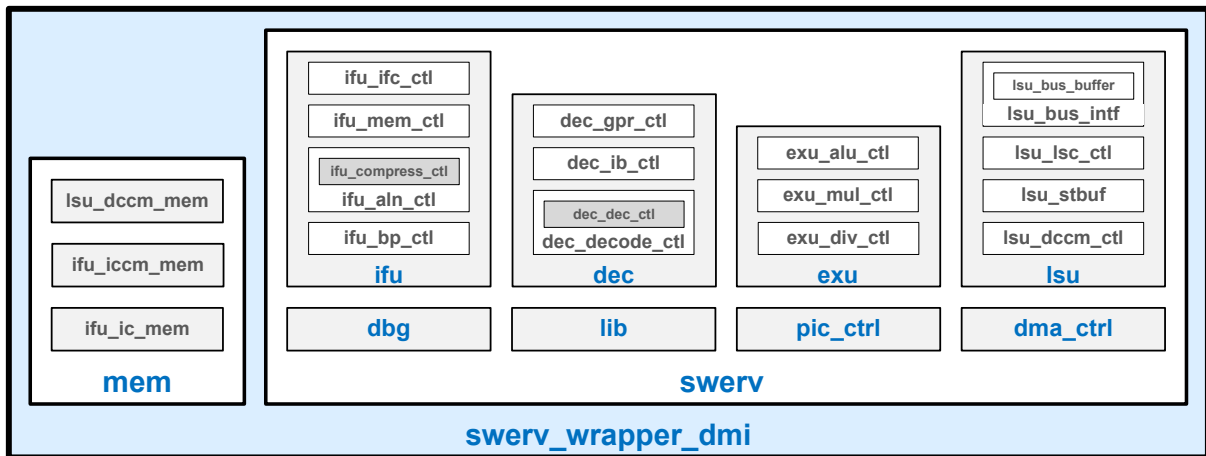


图2. SweRV EH1主要模块

SweRV EH1内核的主要信号： 补充文档SweRVref.docx在第3部分中提供SweRV EH1处理器各模块的主要输入/输出信号。进行实验11-20时，可将其作为参考。

B. 取指（FC1和FC2）和对齐阶段

在本部分中，我们将分析流水线的前三个阶段：SweRV EH1流水线的两个取指阶段（FC1和FC2）和对齐阶段。图3给出了这些阶段的高度简化视图。

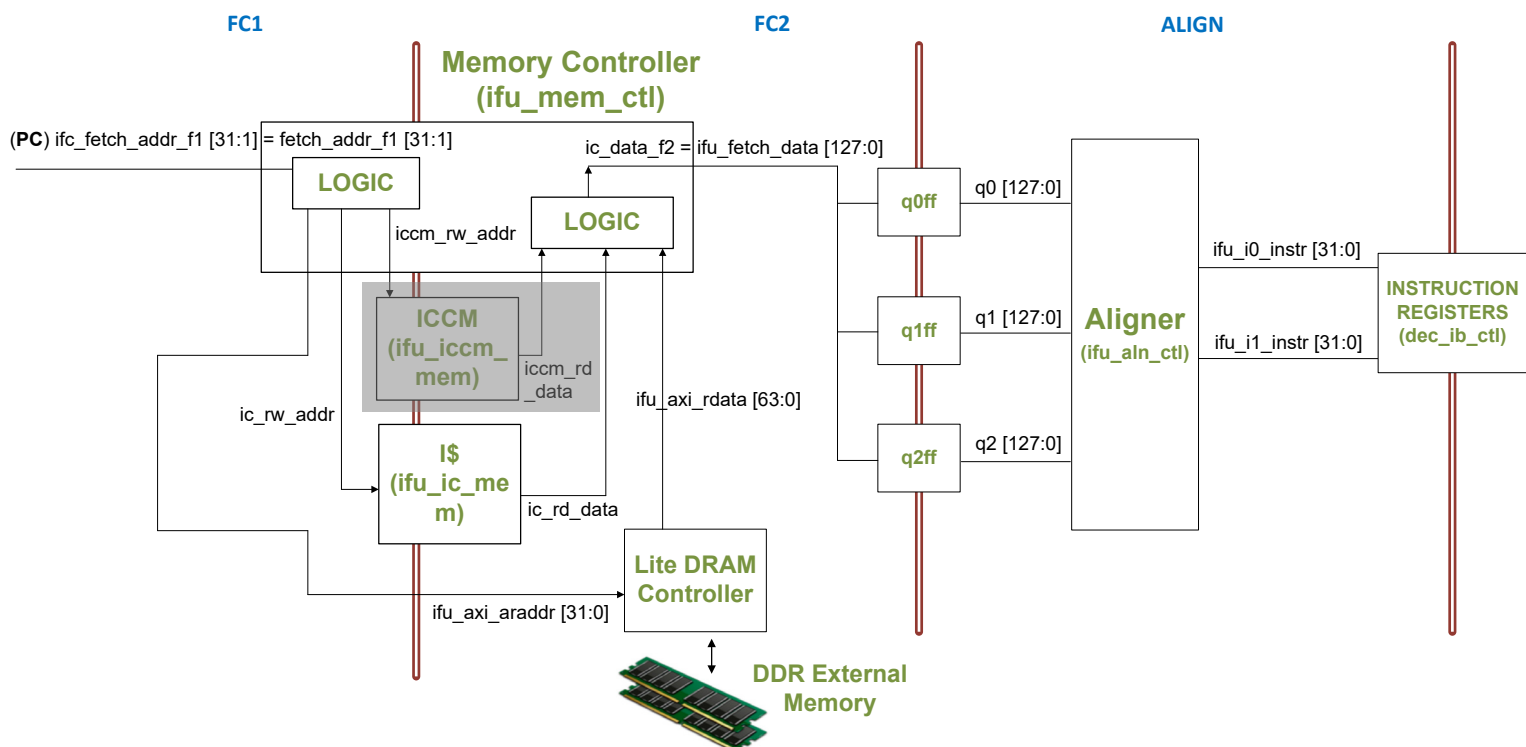


图3. FC1、FC2和对齐阶段的简化视图。
请注意，ICCM带有阴影，表示它在RVfpga系统中被禁止。

i. 取指阶段（FC1和FC2）

在每个循环中，取指阶段负责从指令存储器中读取指令。在我们的配置中，指令存储器由ICCM（在模块`ifu_iccm_mem`中实现）、指令高速缓存（`IS`，在模块`ifu_ic_mem`中实现）和DDR外部存储器组成。`IS`和ICCM都由一个统一的存储器控制器（`ifu_mem_ctl`）控制，而外部存储器由Lite DRAM控制器控制。在默认RVfpga系统中，ICCM被禁止，但可以按照实验20中的说明轻松将其包含在内。

如图3所示，指令地址（称为取指地址`ifc_fetch_addr_f1`）在第一个取指阶段（**FC1**）计算，实验16中将对它进行进一步讨论。该地址提供给指令存储器控制器（在模块`ifu_mem_ctl`中实现）：`fetch_addr_f1 = ifu_fetch_addr_f1`。

信号通常具有与其所属单元相对应的前缀。例如，“`ifu`”代表取指单元。信号将附加与其关联的阶段。例如，“`f1`”表示FC1阶段。

指令在第二个取指阶段（**FC2**）从主存储器（即DDR外部存储器）或ICCM读取。如果指令地址处于主存储器地址范围内，则`IS`提供指令。在`IS`未命中时，流水线必须暂停，直到外部存储器通过AXI总线提供指令，这需要几个周期。如果指令地址处于ICCM地址范围内，则ICCM通过`ifu_ic_mem`模块内部实现的多路开关提供低延时指令。

RVfpga系统的指令存储器配置如下（此配置可以修改，我们将在以后的实验中展示）：

- 16 KiB指令高速缓存
- 512 KiB ICCM（禁止）：地址范围：0xEE000000 – 0xEE07FFFF
- 128 MiB外部存储器：地址范围：0x00000000 – 0x07FFFFFF

如果程序没有暂停（即没有控制、数据或结构冒险，没有`IS`未命中等），则每两个周期读取4条32位指令（总共128位）：请参见信号`ifu_fetch_data[127:0]`。这足以使2路超标量流水线以每周期2条指令的最大吞吐量工作。三个缓冲区（`q0ff`、`q1ff`和`q2ff`）最多可以存储三个此类128位指令束。

ii. 对齐阶段

对齐阶段在两个取指阶段之后（参见图3），在模块`ifu_aln_ctl`中实现。对齐阶段负责执行两个主要任务：

- **每个周期向译码阶段提供两条32位指令：**对齐阶段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，它们临时存储在缓冲区`q0ff`、`q1ff`和`q2ff`中。这两条指令通过信号`ifu_i0_instr[31:0]`（通路0）和`ifu_i1_instr[31:0]`（通路1）分配给SweRV EH1中的每一路（共两路），然后存储在模块`dec_ib_ctl`中实现的两个指令寄存器（Instruction Register, IR）中。

- **解压指令：**RISC-V的压缩指令扩展（RVC）通过减少控制、立即数和寄存器字段的大小并利用冗余或隐含寄存器将常见整数和浮点指令的大小减小到16位。指令大小减小后可降低成本、功耗和所需的存储空间（参见DDCARV的第6.6.5节）。对齐阶段将在必要时解压这些16位指令，然后将其传送到仅译码32位指令的译码阶段。此过程由if_compress_ctl模块执行，该模块在对齐器（模块ifu_aln_ctl）内部实例化。

压缩指令：补充文档SweRVref.docx在第5部分中解释了SweRV EH1中压缩指令的执行，并提出了一些新任务。

C. 译码、执行、提交和回写阶段

在本部分中，我们将分析SweRV EH1流水线的译码、执行、提交和回写阶段。图4给出了这些阶段的简化视图，我们将在以后的实验中进行扩展。

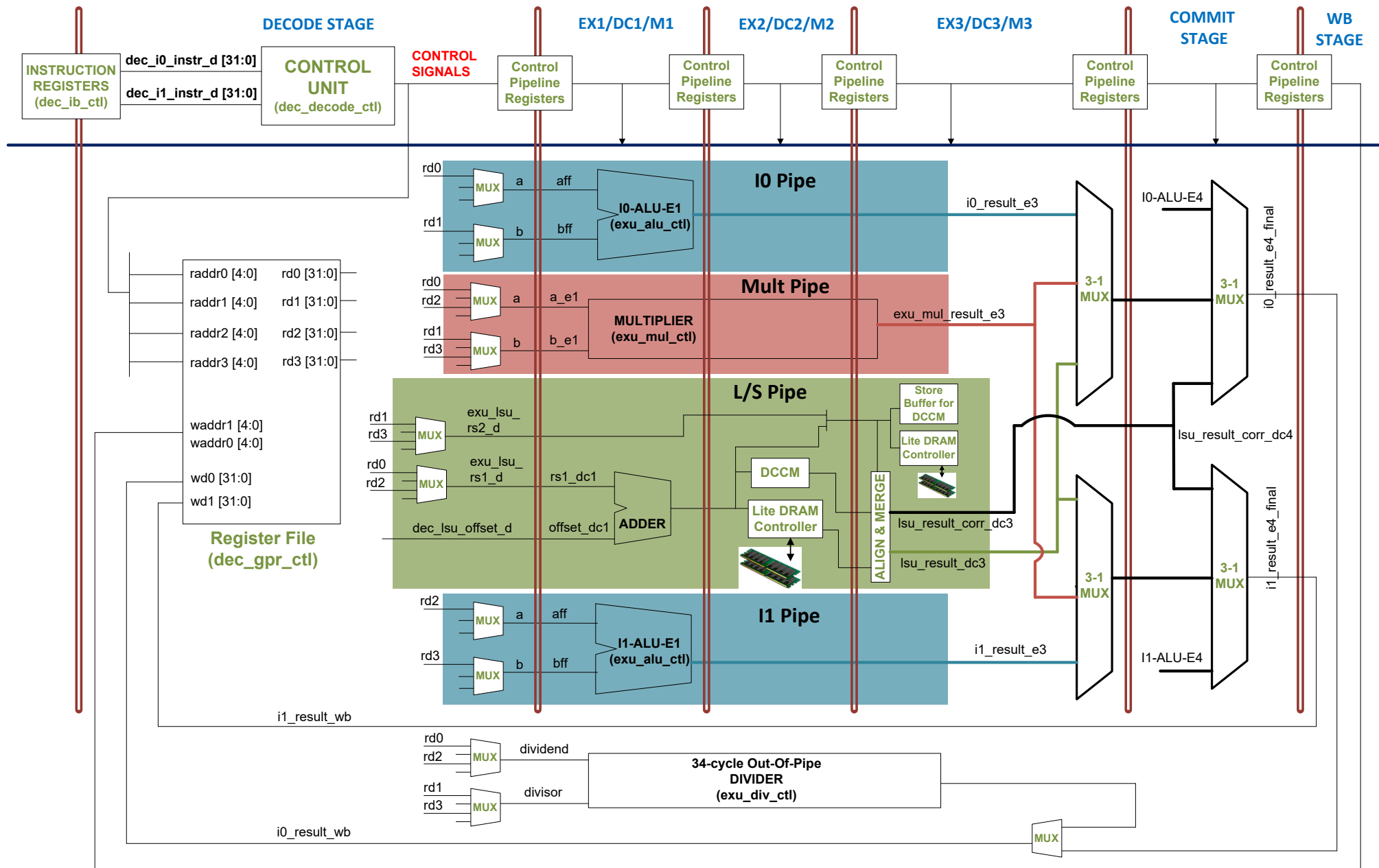


图4. 译码、执行、提交和回写阶段的简化视图

i. 译码阶段

此阶段的Verilog模块位于文件夹

`[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/dec`中。在每个周期中，译码阶段负责两个主要任务：

- **对指令进行译码并产生控制信号：**控制信号分为几种类型，具体在 `[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/include/swerv_types.sv` 中定义。每个结构/类型都与给定的单元相关：ALU（`alu_pkt_t`）、乘法单元（`mul_pkt_t`）、除法单元（`div_pkt_t`）和寄存器（`reg_pkt_t`）等。

用于控制位的结构：补充文档SweRVref.docx在第4部分中扩展了SweRV EH1处理器中用于对控制信号进行分组的主要结构/类型的说明，并提出了一些新任务。在后面的实验中，我们将重点关注与所讨论单元相关的类型。

模块**dec_decode_ctl**中实现的控制单元接收在前几个阶段取指、解压、对齐及分配到每一路的两条32位指令（通路0为信号`dec_i0_instr_d[31:0]`，通路1为信号`dec_i1_instr_d[31:0]`）并对其进行译码，为每条指令生成控制信号。图5给出了控制单元（模块**dec_decode_ctl**）的高级视图，控制单元分两个阶段生成控制信号：前两个模块（**i0_dec**和**i1_dec**）使用指令（`i0`和`i1`）生成总体控制信号（`i0_dp`和`i1_dp`，其类型均为`dec_pkt_t`），然后第二个单元（**译码**）使用这些信号为每个流水线路径生成控制信号，也称为“管道”（`i0_ap`、`i1_ap`、`lsu_p`和`mul_p`等）。

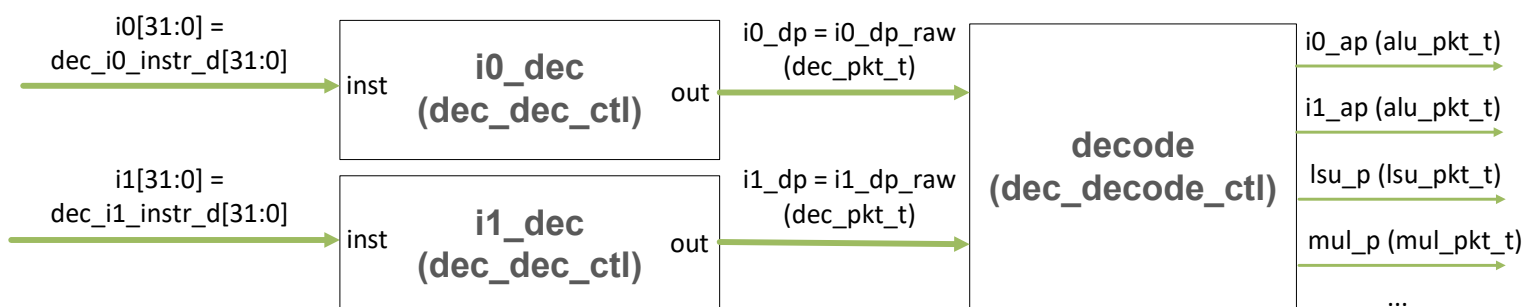


图5. 控制单元

控制单元使用流水线寄存器（在图4中标记为**控制流水线寄存器**）将这些控制信号传播到后续流水线阶段，这些流水线寄存器处于各流水线阶段之间。

- **将指令分发到适当的管道并提供操作数：**如图4所示，SweRV EH1包括两个整数管道（`I0`和`I1`）、一个乘法管道和一个装载/存储管道（`L/S`），此外，还包括一个位于流水线外部的34周期除法器。每条指令译码完成后，处理器将其发送到四个单独的流水线之一：
 - 算术逻辑和分支指令在`I0/I1`管道中执行。

- 装载和存储在L/S管道中执行。
- 乘法指令通过乘法管道执行。
- 将执行的指令划分到除法管道。

鉴于每个周期最多对两条指令进行译码（一条在通路0中，另一条在通路1中），两条均在可能时调度执行。例如，一些可能的组合包括：

- 两条独立的算术逻辑指令发送到I0和I1管道。
- 一条算术逻辑指令和一条乘法（mul）指令分别发送到I0（或I1）和乘法管道。
- 访存（装载或存储）指令在L/S管道中执行，乘法指令在乘法管道中执行。

遗憾的是，当一条或两条指令必须暂停时，存在某些情况（例如冒险，我们将在实验14-16中分析这些情况）。这些情况也会在译码阶段确定。例如：

- 如果两条mul指令在同一个周期内译码，则通过将第二条mul指令延迟一个周期来消除结构冒险（实验14中将对对此进行详细分析）。
- 如果两个相关的算术逻辑（Arithmetic-Logic, A-L）指令在同一周期内进行译码，则通过将第二条A-L指令延迟一个周期来消除写后读数据冒险（实验15中将对对此进行详细分析）。

除了调度指令外，还必须为管道提供相应的操作数。为此，几个3:1和4:1多路开关（参见图4）会在可能的操作数中进行选择，并使用流水线寄存器将其传播到下个阶段。这些多路开关在模块exu的第279-328行中实现（即使多路开关位于exu模块内部，它们也在译码阶段工作）。其输入操作数的来源可能如下：

- **旁路逻辑：**大多数数据相关性在译码阶段通过旁路来避免，我们将在实验15中进行分析。来自旁路逻辑的输入未在图4的3:1和4:1多路开关中进行标记，为简单起见，仅显示空白线。
- **立即寻址：**一些RISC-V指令使用立即寻址模式，其中操作数直接从指令位提供。来自立即寻址的输入未在图4的3:1和4:1多路开关中进行显示，——仅显示空白输入线）。
- **寄存器文件：**SweRV EH1处理器中可用的寄存器文件（图6）有4个读端口和3个写端口（请注意，第三个写端口在图4中包含的寄存器文件中被忽略，因为它仅用于特定情况，我们将在以后的实验中进行分析）。这些读/写端口可在每个周期执行两条指令。来自寄存器文件的输入仅使用信号的名称在图4的3:1和4:1多路开关中进行显示。为简单起见，未显示与寄存器文件的连接。

每个读/写端口都有一个5位地址（raddr0 ... raddr3和waddr0 ... waddr2）以及一个1位使能信号（rden0 ... rden3和wen0 ... wen2），后者未在图4中显示。写端口也有一个32位写数据输入（wd0 ... wd2），读端口有一个32位读数据输出（0 ... rd3）。寄存器文件包含32个32位寄存器（称为x0-x31），其中x0硬接线到0。

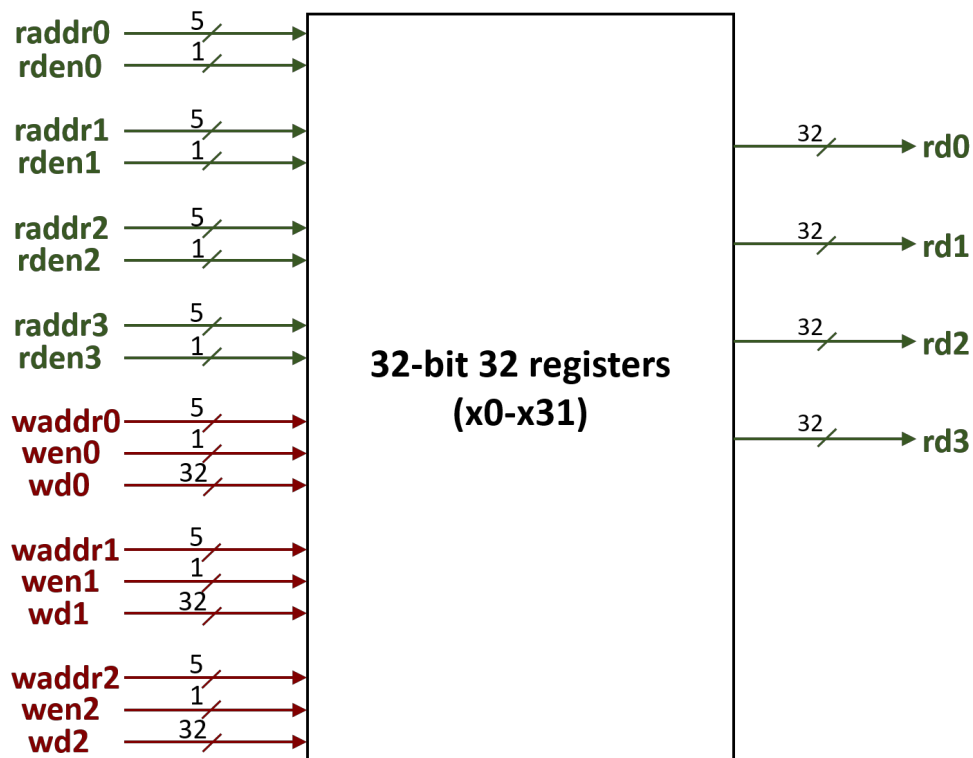


图6. SweRV EH1中可用的寄存器文件

任务：在模块`dec_gpr_ctl`中实现寄存器文件，并在模块`dec`中将其实例化（参见图7）。分析模块`dec_gpr_ctl`的Verilog代码及主要信号的仿真（位于文件`[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/dec/dec_gpr_ctl.sv`中），以了解其工作方式。请注意，SweRV EH1处理器允许包含多个寄存器文件，但RVfpga系统中使用的配置仅使用一个寄存器文件（参见文件`dec.sv`的第402行：`localparam GPR_BANKS = 1;`）。

```

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

```

图7. 模块`dec`内部的寄存器文件实例化

ii. 执行阶段

在本小节中，我们将分析SweRV EH1中可用管道的简化版本：两个**整数管道**（**I0管道**和**I1管道**）、一个**乘法管道**、一个**装载/存储管道**和一个非流水线34周期除法器。

I0/I1管道：两个整数管道在图4中以蓝色显示。它们分为三个阶段，分别称为**EX1**、**EX2**和**EX3**。这两个管道的**EX1**阶段均包含一个1周期延时的ALU，能够执行算术运算（例如**加法**或**减法**）以及逻辑运算（例如**逻辑与**或**逻辑或**）。**EX2**和**EX3**阶段执行少量任务，但必须通过这两个阶段将**A-L**指令与其他需要三个周期来计算运算的指令类型（例如**装载**、**存储**和**乘法**等）进行同步。在实验12中，我们将更详细地分析I0/I1管道。

乘法管道：乘法管道在图4中以红色显示。它分为三个阶段：**M1**、**M2**和**M3**。该管道包括一个能够执行整数乘法的3周期乘法器。在实验14中，我们将更详细地分析乘法管道。

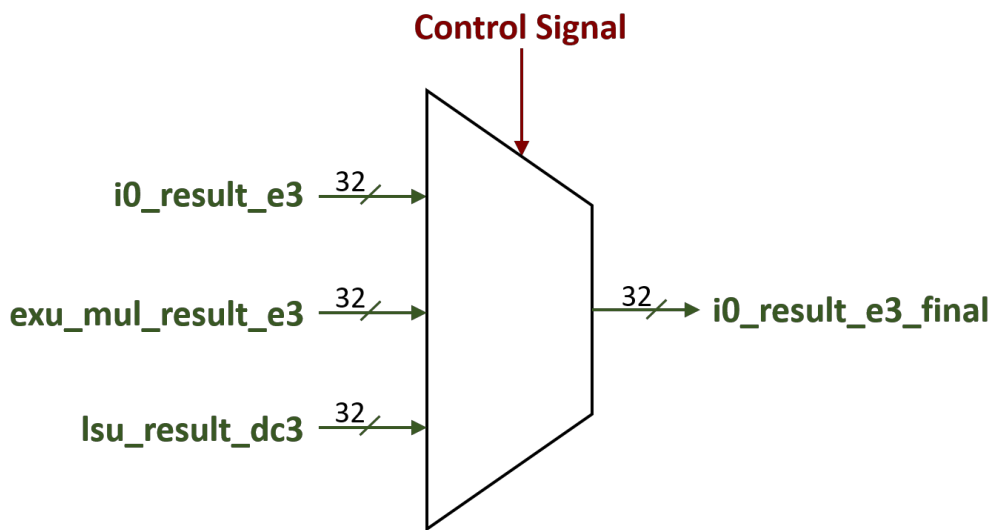
装载/存储（L/S）管道：L/S管道在图4中以绿色显示。在实验13中，我们将深入探索这条流线路径。装载和存储指令均通过L/S管道执行。它包括3个阶段：

- **DC1：**在第一阶段，加法器单元通过将寄存器基地址和立即偏移量相加来计算地址。
- **DC2：**在第二阶段，装载指令使用DC1中计算的地址读取存储器。如果地址映射到DCCM，则访问延时仅为1个周期，流水线将继续运行，不会暂停。但是，如果访问映射到主存储器，则流水线可能需要暂停几个周期，具体取决于阻塞/非阻塞装载如何使用以及相关性的存在，我们将在后面的实验中进一步分析。
- **DC3：**在第三阶段，数据进行对齐和合并（例如，如果之前存储到同一地址的存储操作仍在执行，则可能需要将来自该存储操作的数据转发到装载阶段）。在此阶段，存储指令开始写入存储器，过程持续几个周期。如果写操作映射到DCCM，则数据和地址在发送到DCCM之前均缓存在存储缓冲区中（正如我们在实验13中分析的那样）；如果写操作映射到主存储器，则数据和地址均通过AXI总线发送到外部存储器（Lite DRAM控制器管理对该存储器的访问）。

除法器：除法器在图4中以白色显示。它是一个非流水线单元，需要最多34个周期来计算其结果。实验14将更详细地分析除法器。

两个3:1多路开关：在第三个执行阶段（EX3/DC3/M3）结束时（如图4所示），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。这两个多路开关位于**dec_decode_ctl**模块中。与通路0相关联的上方多路开关如图8所示。该多路开关的三个输入包括：

1. **I0管道结果：**i0_result_e3。实验12将分析该路径。
2. **L/S管道结果：**lsu_result_dc3。实验13将分析该路径。
3. **乘法管道结果：**exu_mul_result_e3。实验14将分析该路径。



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

图8. 通过3:1多路开关选择EX3结果：图和Verilog

任务：根据图8分析多路开关的控制位。请注意，控制位在信号e3d中，该信号由信号dd经流水线处理得到，后一个信号由控制单元在译码阶段生成（有关控制位的说明，请参见SweRVref.docx）。

iii. 提交阶段

在提交阶段，两个3:1多路开关（每路一个）选择要回写到寄存器文件的结果（参见图4）。与通路0相关联的上方多路开关如图9所示。它有三个输入：

1. **EX3结果：** i0_result_e4。（EX3的3:1多路开关的输出）。
2. **更正后的读取数据：** lsu_result_corr_dc4。实验13将分析该路径。
3. **辅助ALU结果：** exu_i0_result_e4。为了简单起见，这些ALU未在图4中进行显示。如前文所述，这些ALU可在出现数据冒险的必要情况下重复算术逻辑指令（有关详细信息，请参见实验15）。

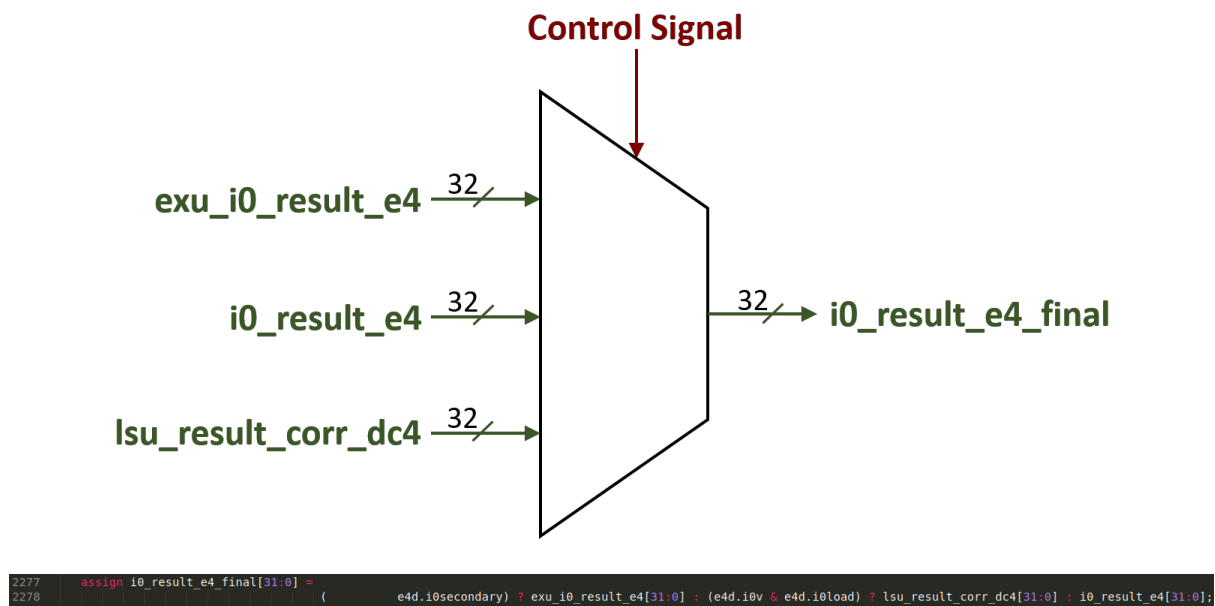


图9. 通过3:1多路开关选择最终结果：图和Verilog


任务：根据图9分析多路开关的控制位，这些控制位位于模块**dec_decode_ctl**中。

iv. 回写阶段

最后一个阶段（回写阶段）使用如图6所示的前两个写端口（0和1）将结果写入寄存器文件（在实验14中，我们将看到何时使用第三个写端口（2））。并非所有周期均写入两个结果：有些指令不写入寄存器（即分支指令、存储指令...），也并非所有周期均执行两条指令。寄存器标识符和使能信号在译码阶段生成并由控制流水线寄存器提供。

D. Verilator中的仿真示例

在本部分中，我们将演示在SweRV EH1流水线中并行执行的两条指令的仿真过程，展示前几部分中介绍的信号。后面的实验还将使用Verilator仿真来可视化处理器的内部信号并通过图示阐述理论说明。

接下来将执行图10所示的示例代码，重点关注mul和add指令（以红色突出显示），它们是无限循环的一部分。文件夹[RVfpgaPath]/RVfpga/Labs/Lab11/ExampleProgram提供了PlatformIO项目，这样便可根据需要分析、仿真和更改程序。在PlatformIO中打开项目并进行编译（根据《入门指南》，可以通过单击VSCode底部的按钮编译项目）。反汇编文件（位于[RVfpgaPath]/RVfpga/Labs/Lab11/ExampleProgram/.pio/build/swervolf_nexys/firmware.dis）会显示地址和机器码。请注意，两条指令分别位于地址0x000000F0和0x000000F4处：


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

这两条指令前后存在几条nop（无操作）指令以将其与其他指令隔离，这样便能更好地进行分析。nop指令不会改变系统的状态。在RISC-V中，nop转换为addi x0,x0,0，后者编码为值为0x00000013的32位机器指令。这段代码中定义了几个宏以将一些nop指令（1到10）插入其中（为简单起见，宏定义不包含在图10中，但可以在PlatformIO项目中看到）。

为清楚起见，我们将按照SweRVref文档的第2部分所述的过程禁止分支预测器和压缩指令。

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

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

图10. 循环中包含mul和add指令的示例程序

图11和12所示为执行图10中的程序时处理器信号的Verilator波形。图11所示为来自前三个流水线阶段（FC1、FC2和对齐 – 参见图3）的信号。图12所示为来自其余阶段的信号（参见图4）。为了与图3和图4保持一致，我们将结果拆分为两个图，但请记住，这两条指令来自对齐阶段（图11的右侧）到译码阶段（图12的左侧）。

图中包含以下信号，用于在指令通过流水线时跟踪指令（ifu跟踪对齐阶段的指令，dec跟踪译码阶段的指令，ex跟踪X（X = 第一、第二和第三）执行阶段的指令，e4跟踪提交阶段的指令，wb跟踪回写阶段的指令）以及获知为指令分配的通路（为i0分配通路0，为i1分配通路1）。

- | | |
|---------------------------------|------------|
| • ifu_i0_instr和ifu_i1_instr | → 对齐阶段的指令 |
| • dec_i0_instr_d和dec_i1_instr_d | → 译码阶段的指令 |
| • i0_inst_e1和i1_inst_e1 | → EX1阶段的指令 |
| • i0_inst_e2和i1_inst_e2 | → EX2阶段的指令 |
| • i0_inst_e3和i1_inst_e3 | → EX3阶段的指令 |
| • i0_inst_e4和i1_inst_e4 | → 提交阶段的指令 |
| • i0_inst_wb和i1_inst_wb | → 回写阶段的指令 |

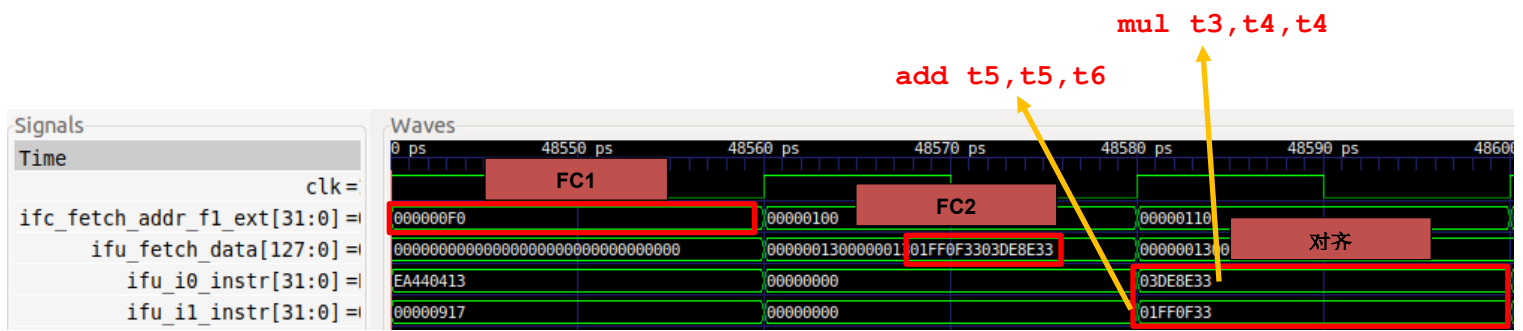


图11. 前三个流水线阶段的仿真：FC1、FC2和对齐

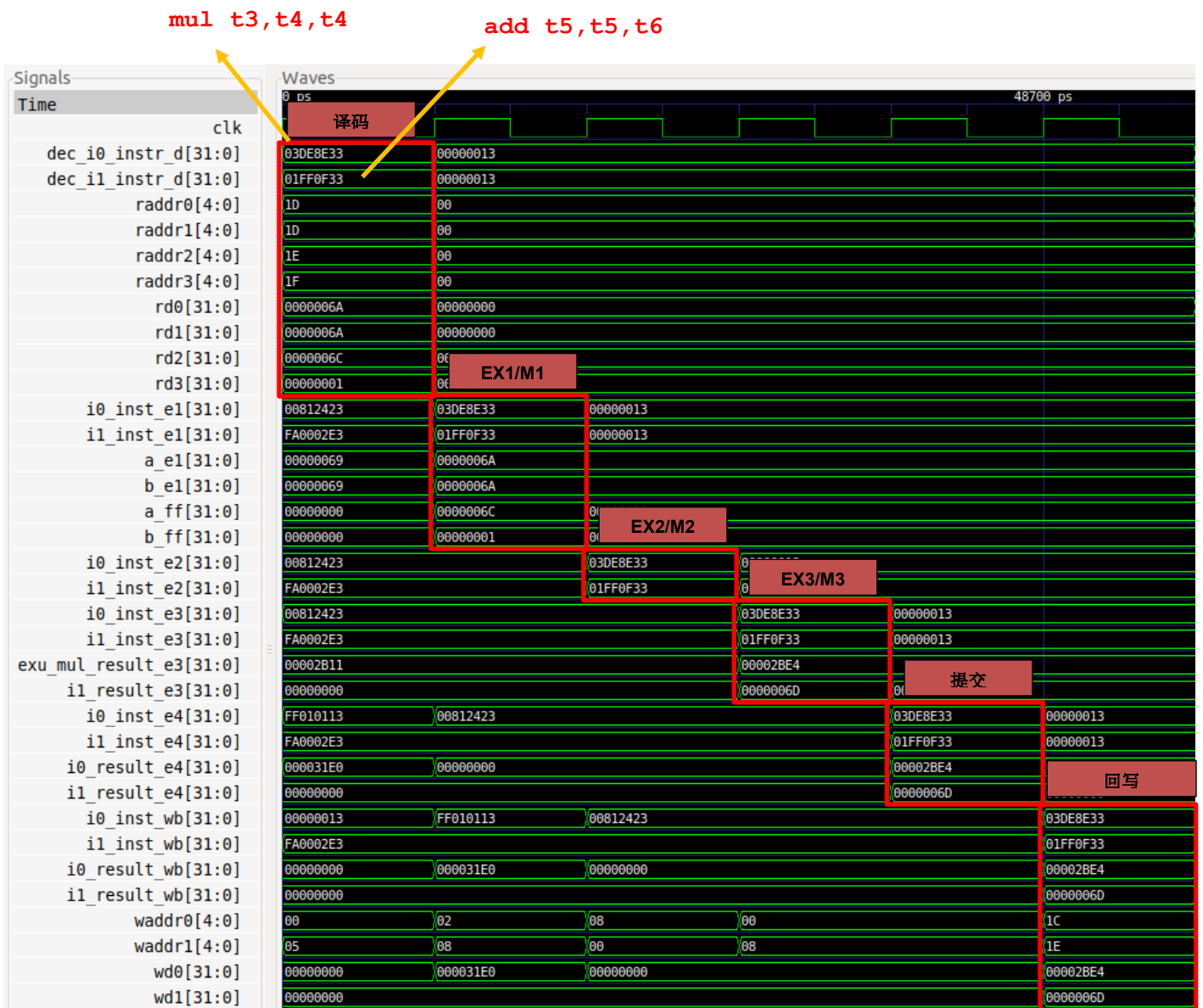



图12. 最后阶段的仿真：译码、EX1/M1、EX2/M2、EX3/M3、提交和回写

任务：按照以下步骤（如GSG第7部分中详述）在自己的计算机上重复图11和图12中的仿真过程：

- 必要时生成仿真二进制文件（*Vrvfpgasim*）。
- 在PlatformIO中，打开在以下位置提供的项目：
[RVfpgaPath]/RVfpga/Labs/Lab11/ExampleProgram。
- 在文件*platformio.ini*中建立到RVfpga仿真二进制文件（*Vrvfpgasim*）的正确路径。
- 使用Verilator生成仿真轨迹（生成轨迹）。
- 使用GTKWave打开轨迹。
- 使用文件*test_1.tcl*和*test_2.tcl*（在*[RVfpgaPath]/RVfpga/Labs/Lab11/ExampleProgram*中提供）打开与图11和图12所示信号相同的信号。为此，在GTKWave上，单击“**File** → **Read Tcl Script File**”（文件 → 读取Tcl脚本文件），然后选择*test_1.tcl*或*test_2.tcl*文件。
- 单击几次“**Zoom In**”（放大）（）移动至48600 ps（或循环的任何其他迭代，第一次迭代除外）。

同时分析图11和图12中的波形以及图3和图4中的图。图中包括与每个流水线阶段相关联的一些信号。以红色突出显示的值对应于两条经过流水线的指令（mul和add）。

- **FC1：**在图11的第一个周期中，信号*ifc_fetch_addr_f1_ext[31:0]*（程序计数器，提供给指令存储器）包含mul指令的地址（即，*指向*）（*ifc_fetch_addr_f1_ext* = 0x000000F0）。
- **FC2：**在图11的第二个周期中，指令存储器提供一个新的128位信号，其中包括我们在示例中分析的两条指令（mul以绿色显示，add以红色显示）：

```
ifu_fetch_data = 0x000000130000001301FF0F3303DE8E33
```

- **对齐：**在图11的最后一个周期中，从新的128位信号中提取两条指令并分发给SweRV EH1包含的两个通路。

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

- **译码：**在图12的第一个周期中，对两条指令进行译码——即从寄存器文件中读取指令的寄存器值，并生成控制位（图中未显示，但可以按照SweRVref.docx所述添加其中一些控制位）。操作数（寄存器值）置于rd0、rd1、rd2和rd3中。

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

- **EX1/M1、EX2/M2、EX3/M3和提交：**在图12接下来的三个周期中，执行加法和乘法。在EX3/M3结束时，使用两个3:1多路开关选择结果，然后传播到提交阶段。

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

- **回写：**在图12的最后一个周期中，将结果回写到寄存器文件中。

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

3. SweRV EH1中的硬件计数器

接下来展示如何使用性能计数器来分析处理器性能。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，例如执行的指令数、执行的周期数、每条指令的平均时钟周期数（CPI）、指令高速缓存命中/未命中次数、正确/错误预测分支数等。

在实验12-20中，我们将定期使用SweRV EH1中的性能计数器来测量和比较不同的幅值。

实际基准：在文件夹[RVfpgaPath]/RVfpga/Labs/Lab20/RealBenchmarks中，我们提供了三个实际应用（CoreMark、Dhrystone和图像处理），实验20中将使用它们来测试SweRV EH1处理器的不同功能。补充文档SweRVref.docx在第6部分中简要描述了这些应用，实验20将展开描述并提出几个任务。

A. SweRV EH1中的性能计数器

RISC-V SweRV EH1程序员参考手册（https://github.com/chipsalliance/Cores-SweRV/blob/master/docs/RISC-V_SweRV_EH1_PRM.pdf）介绍了RISC-V处理器的基本硬件性能监视功能。必须实现以下性能计数器，这些计数器也是控制和状态寄存器（CSR）：

- **mcycle：**自过去任意时间以来，hart（硬件线程）已执行的时钟周期数。
- **minstret：**自过去任意时间以来，hart已停用的指令数。
- **mhpmcounter3–mhpmcounter31：**29个其他事件计数器。事件选择器CSR（*mhpmevent3–mhpmevent31*）为WARL（写入任何值，读取合法值）寄存器，用于控制哪个事件导致相应计数器递增。这些事件的含义由平台定义，但事件0保留，表示“无事件”。

并非所有计数器都需要实现。将计数器及其相应的事件选择器硬接线到0是一种合法实现。具体来说，在SweRV EH1中，只有事件计数器3到6（*mhpmcounter3–mhpmcounter6*）及其相应的事件选择器（*mhpmevent3–mhpmevent6*）正常工作，而事件计数器7到31（*mhpmcounter7–mhpmcounter31*）及其相应的事件选择器（*mhpmevent7–mhpmevent31*）硬接线到0。这些计数器的使能由mgpmc寄存器的位0（0 = 禁止，1 = 使能）控制。

SweRV EH1程序员参考手册的第7章（https://github.com/chipsalliance/Cores-SweRV/blob/master/docs/RISC-V_SweRV_EH1_PRM.pdf）详细介绍了SweRV EH1中的四个性能计数器的特性和操作：

- 四个标准64位宽事件计数器
- 每个计数器具有标准单独事件选择功能
- 标准选择性计数使能/禁止可控性
- 同步计数器使能/禁止可控性
- 标准周期计数器
- 标准停用指令计数器
- 支持基于标准SoC的机器定时器寄存器

该文档中的表7-2列出了SweRV EH1中的50个可计数事件，这些事件在表1中进行了汇总。

表1. SweRV EH1中的可计数事件列表

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

B. 通过 Western Digital 的处理器支持包（Processor Support Package, PSP）使用性能计数器

在寄存器级别使用性能监视系统有点复杂；幸运的是，WD PSP

（<https://github.com/westerndigitalcorporation/riscv-fw-infrastructure>）中的一些函数提供了一种更简单的性能监视方法。如果已按照GSG中的说明安装了PlatformIO，Ubuntu系统中应存在以下两个文件：

- `~/platformio/packages/framework-wd-riscv-sdk/psp/psp_performance_monitor_eh1.c`
- `~/platformio/packages/framework-wd-riscv-sdk/psp/api_inc/psp_performance_monitor_eh1.h`

Windows: `.platformio`文件夹位于用户文件夹（`C:\Users\<USER>`）内。请注意，可能需要启用系统管理才能查看隐藏的文件/文件夹。

macOS: 与在Linux中一样，`.platformio`文件夹位于主文件夹（`~/platformio`）内。

凭借.c文件（`psp_performance_monitor_eh1.c`）实现的函数，可以执行诸如使能/禁止性能监视器（`pspEnableAllPerformanceMonitor`）、将计数器与事件配对

(`pspPerformanceCounterSet`) 或获取计数器值 (`pspPerformanceCounterGet`) 之类的操作。

.h 文件 (`psp_performance_monitor_eh1.h`) 在 `typedef enum pspPerformanceMonitorEvents` 中提供表 1 中的每个事件的名称。

[RVfpgaPath]/RVfpga/Labs/Lab11/HwCounters_Example 中提供的以下示例 (图 13) 说明了如何使用 SweRV EH1 中的四个硬件计数器来测量：周期数、指令数、已提交的分支和预测错误的分支。Main 函数：

- 初始化 UART (`uartInit()`)
- 使能硬件计数器 (`pspEnableAllPerformanceMonitor(1)`)
- 为每个计数器 (`D_PSP_COUNTER0 - D_PSP_COUNTER3`) 分配要测量的事件 (周期数、指令数、已提交的分支和预测错误的分支)
- 读取计数器 (`pspPerformanceCounterGet(D_PSP_COUNTER0)`)
- 调用一个简单的汇编程序 (`Test_Assembly()`) 并再次读取计数器
- 使用函数 `printfNexys` 打印每个计数器的值。

完成一些寄存器的初始化之后，`Test_Assembly()` 函数将循环重复 1,000,000 次；该循环包含五个算术逻辑 (AL) 指令和一个条件分支。反汇编文件也显示在图 13 的末尾，以方便您了解构成循环主体的 32 位机器指令的值。

Test.C 文件

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

#include <psp_api.h>

extern void Test_Assembly(void);

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

    /* Initialize Uart */

    uartInit();

    pspEnableAllPerformanceMonitor(1);

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

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

```

Test_Assembly();

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

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

while(1);
}

```

Test_Assembly.S文件

```

.globl Test_Assembly

.text

Test_Assembly:

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

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

.end

```

firmware.dis文件

```

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

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

```

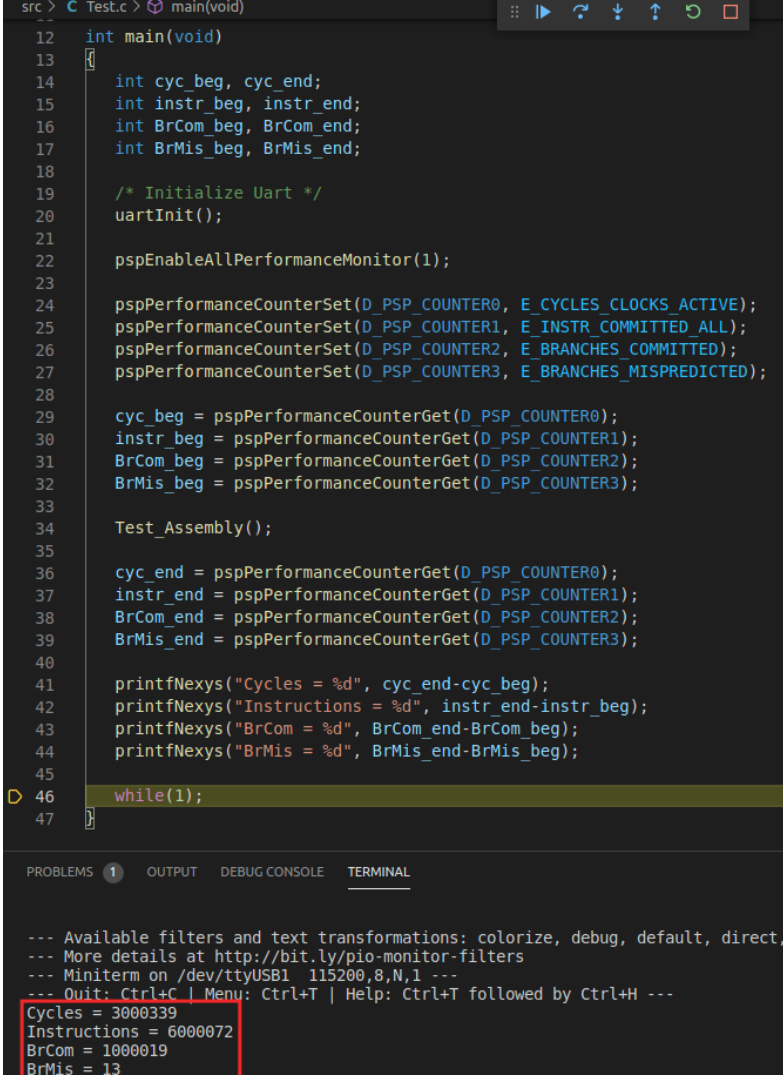
```

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

```

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

任务：按照GSG所述在Nexys A7板上执行图13中的程序。对于测量的四个事件，应得到图14所示的结果。解释并证明结果。



```

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

```

```

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

```

图14. 执行Test.C

任务：在图13所示程序的硬件计数器中测量其他事件。为此，必须使用 pspPerformanceCounterSet 函数在 Test.c 文件中更改待测量事件的配置。请注意，可以使用WD PSP文件中定义的宏配置不同的事件（如表1所示）：
.platformio/packages/framework-wd-riscv-sdk/psp/api_inc/psp_performance_monitor_eh1.h。
例如，如果要测量I\$未命中数而不是分支未命中数，则必须在文件中将Test.c行：
pspPerformanceCounterSet(D_PSP_COUNTER3, E_BRANCHES_MISPREDICTED);
替换为行： pspPerformanceCounterSet(D_PSP_COUNTER3, E_I_CACHE_MISSES);

任务：在Test_Assembly函数中提供其他程序并检查不同的事件是否提供了预期的结果。可以尝试其他指令，例如装载、存储、乘法、除法...以及引发流水线暂停的冒险。