



**Imagination**大学计划

# **SweRV EH1参考**

## **层级、模块、信号和类型**

本文档提供有关以下主题的额外说明：

- 第1部分： **Sigasi Studio**
- 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： **分组控制位的主要结构/类型**
- 第5部分： **RISC-V压缩指令**
- 第6部分： **实际基准**

## 1. SIGASI STUDIO

Sigasi Studio可帮助设计人员以最直观的方式编写、检查和修改数字电路设计，从而提高工作效率。该工具能够了解设计环境，并借助智能自动完成和代码重构等高级功能实现更为简单高效的VHDL、Verilog和SystemVerilog设计。

用户需要付费购买许可证，才能使用专业版本的Sigasi Studio。幸运的是，您可以访问以下链接轻松获取供教育用的免费许可证：<https://www.sigasi.com/try-form-edu/>。填写个人信息并获得许可证后，您将收到一封包含Sigasi Studio下载链接以及安装和使用说明的电子邮件（<https://www.sigasi.com/download/>，参见图1）。该软件适用于Windows、Linux和MacOS操作系统。



图1. Sigasi Studio的下载链接以及安装和使用说明

一旦在系统中安装了Sigasi Studio，就可以使用该软件来检查RVfpga。以下链接为Hendrik Eeckhaut两年前发表的文章，其中介绍了如何创建和配置SweRV EH1项目：[https://insights.sigasi.com/tech/swerv\\_riscv/](https://insights.sigasi.com/tech/swerv_riscv/)。接下来，我们将以这些信息为切入点，提供创建和配置RVfpga项目的完整说明。

1. 创建一个目录[RVfpgaPath]/RVfpga/src的副本，将其命名为[RVfpgaPath]/RVfpga/src\_SigasiStudio
2. 打开Sigasi Studio，进入下载目录中，双击文件*sigasi\_internal*（参见图2）。

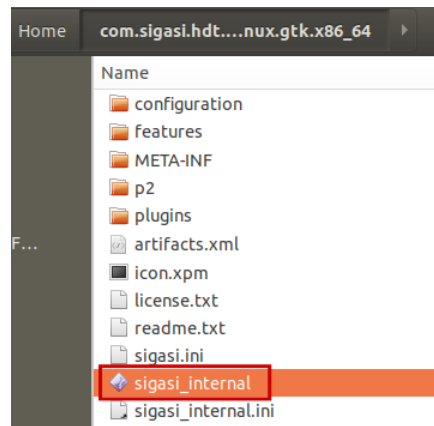


图2. 打开Sigasi Studio

3. 在Sigasi Studio窗口中，单击“File”（文件）→“Import”（导入），将打开一个新窗口，要求您选择要添加到系统中的项目类型。选择“Import a (System) Verilog project”（导入（System）Verilog项目），然后单击“Next”（下一步）（图3）。

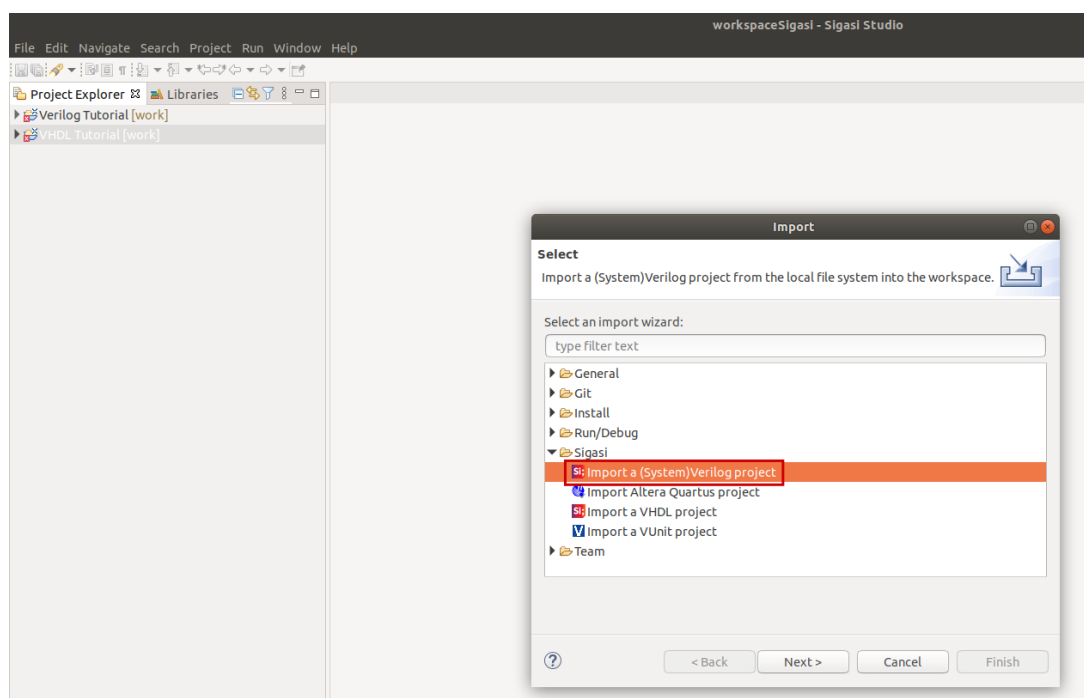
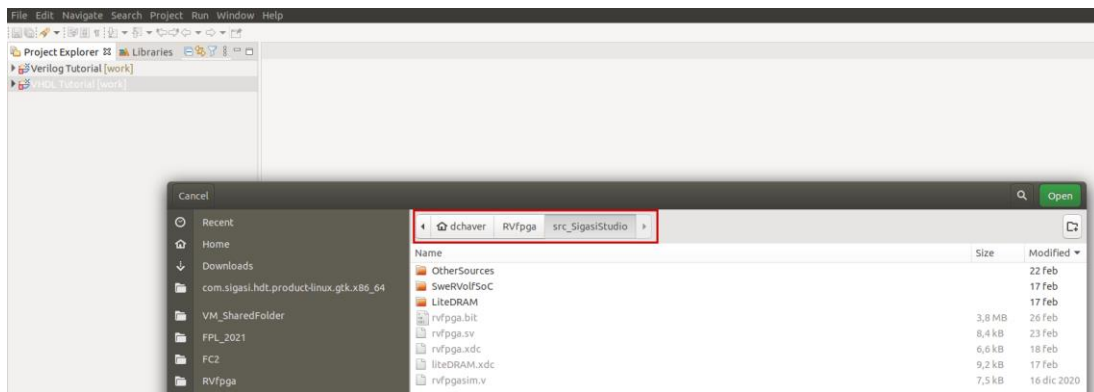


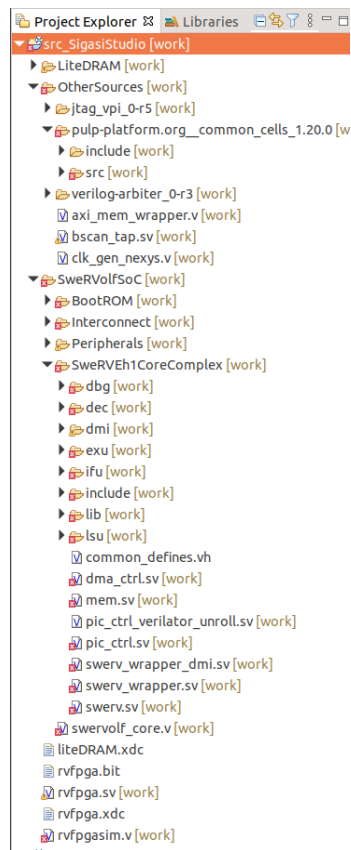
图3. 导入RVfpga项目

4. 现在单击“Browse...”（浏览...），导航到src\_SigasiStudio目录并选择该目录，单击“Open”（打开）（参见图4），然后单击“Finish”（完成）。



**图4. 打开RVfpga源目录**

5. 打开项目时会出现大量错误（参见图5），其中大多数错误是由于项目配置中缺少多个包含文件而引起的。



**图5. RVfpga Sigasi Studio项目的初始错误**

6. 在“Project Explorer”（项目资源管理器）中，右键单击`src_SigasiStudio`项目，打开“Properties”（属性）窗口（参见图6）。

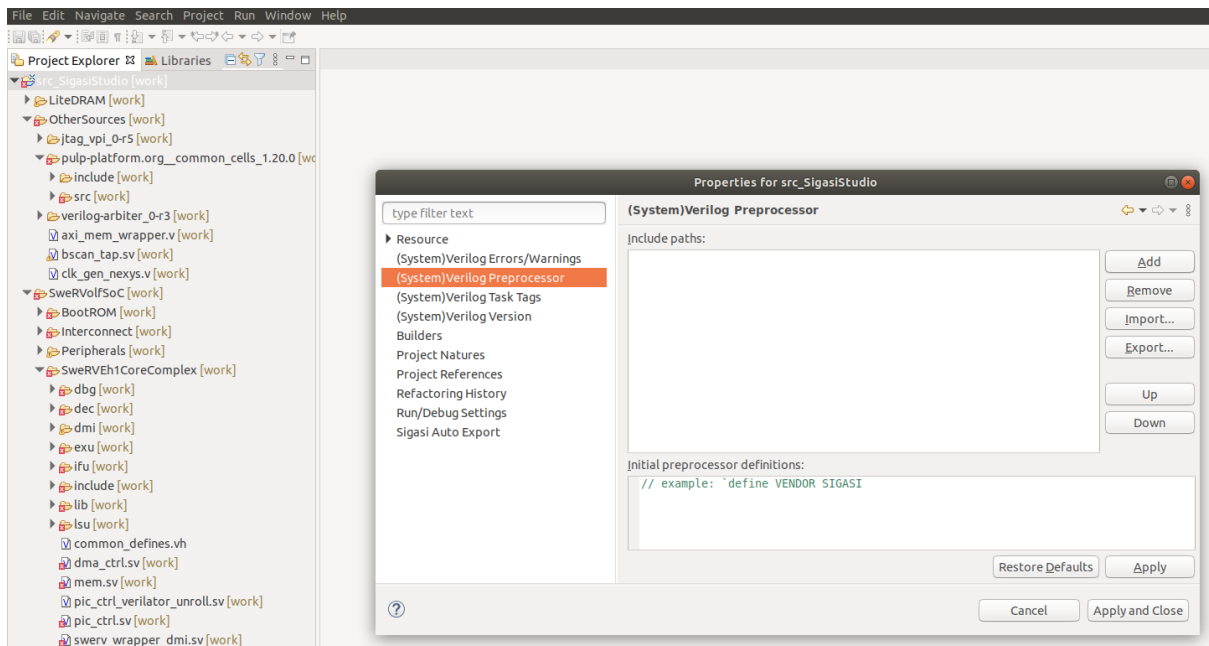


图6. 项目属性

7. 在“Properties”（属性）窗口（图6）中选择“(System)Verilog Preprocessor”（（System）Verilog预处理器），并添加以下包含路径（方法为单击右侧的“Add”（添加）按钮）：

- *[RVfpgaPath]/RVfpga/src\_SigasiStudio/SweRVolfSoC/SweRVEh1CoreComplex/include*
- *[RVfpgaPath]/RVfpga/src\_SigasiStudio/OtherSources/pulp-platform.org\_\_common\_cells\_1.20.0/include*
- *[RVfpgaPath]/RVfpga/src\_SigasiStudio/SweRVolfSoC/Interconnect/AxiInterconnect/pulp-platform.org\_\_axi\_0.25.0/include*
- *[RVfpgaPath]/RVfpga/src\_SigasiStudio/SweRVolfSoC/Interconnect/AxiInterconnect*
- *[RVfpgaPath]/RVfpga/src\_SigasiStudio/SweRVolfSoC/Interconnect/WishboneInterconnect*

添加上述五个目录后，单击“Apply”（应用）按钮。

然后，在同一窗口的底部框（初始预处理器定义）中，输入以下行：``include "common_defines.vh"`。单击“Apply”（应用）和“Close”（关闭）按钮。

图7所示为最终的状态。

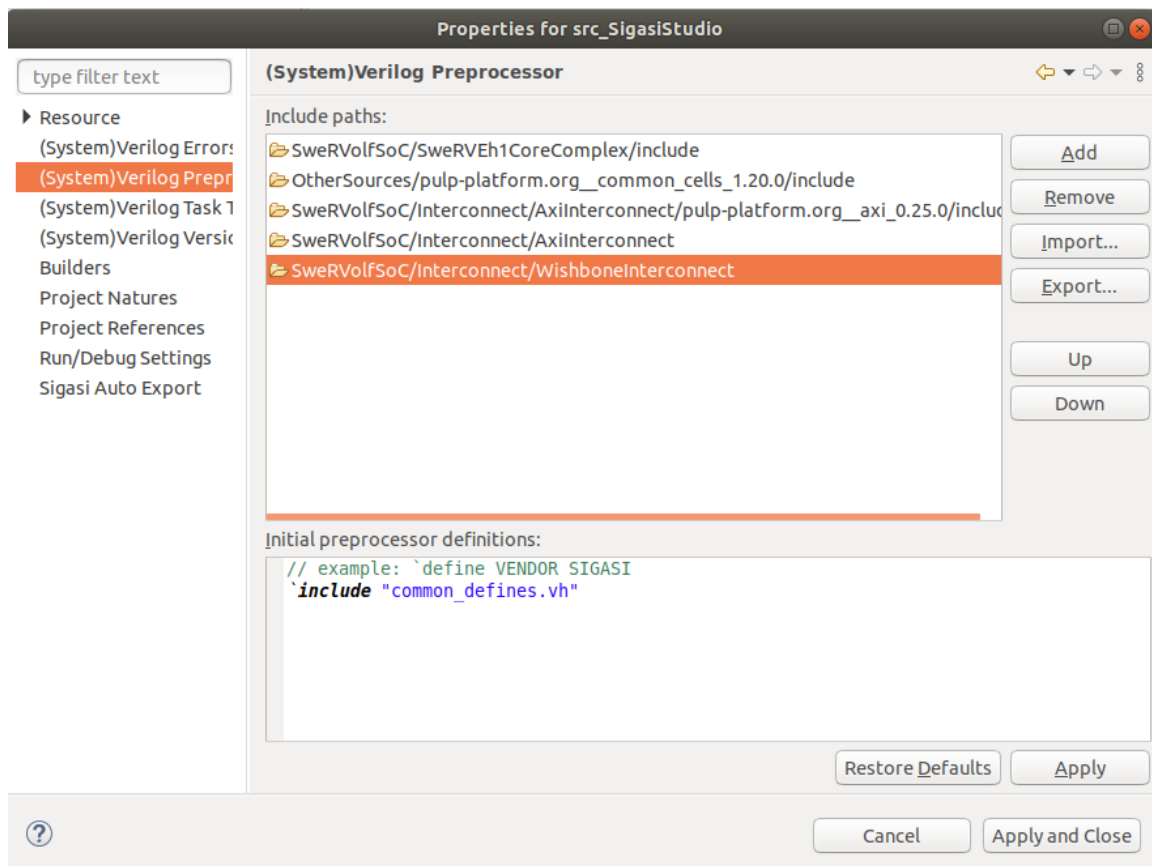


图7. 包含目录和文件

#### 8. 最后，删除文件

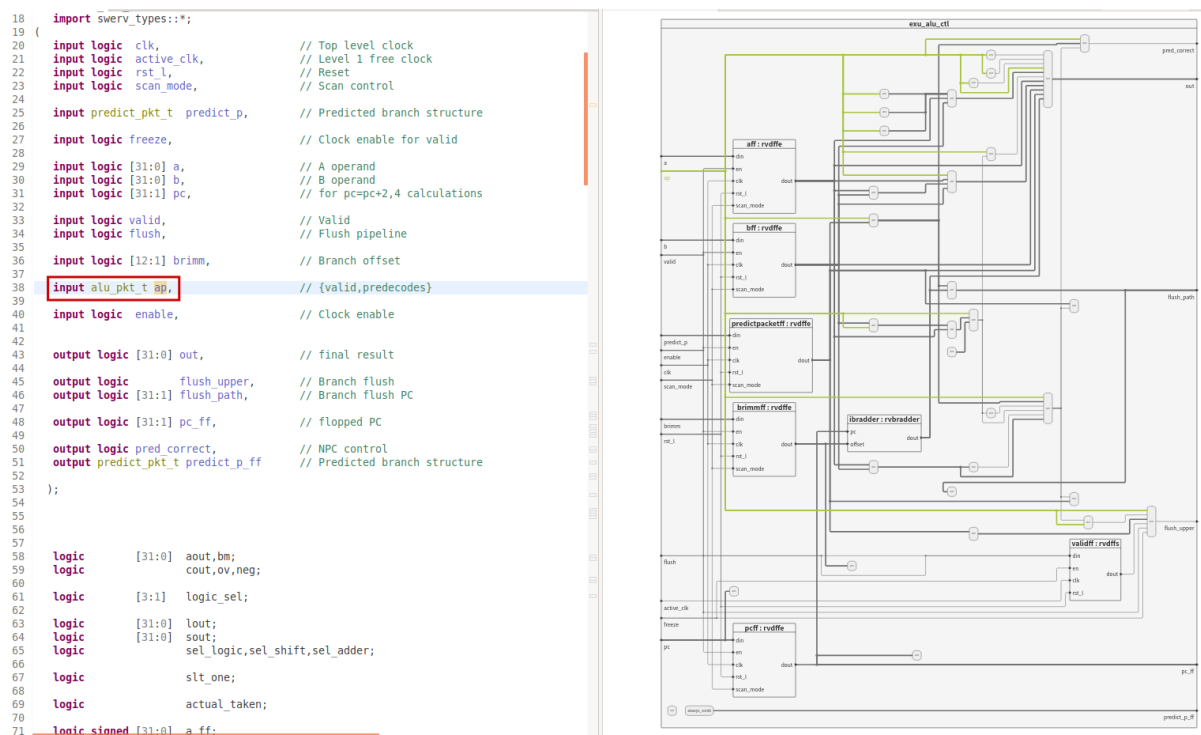
*[RVfpgaPath]/RVfpga/src\_SigasiStudio/SweRVolfSoC/BootROM/sw/boot\_main.vh*, 我们的项目无需使用该文件，并且保留该文件会导致一些错误。可以在文件资源管理器中或Sigasi Studio内部删除该文件。

完成上述步骤后，错误应会全部消失，只剩下一些警告，您可以忽略这些警告。

此时便可开始使用Sigasi Studio检查RVfpga SoC。我们接下来将对该工具的一些功能进行测试：

1. 在顶部菜单中，打开“Window”（窗口）→“Show View”（显示视图）→“Block Diagram”（框图），工具右侧将打开一个新窗口，以便您以图形方式在模块中导航。
2. 在本实验中，我们将分析算术指令和逻辑指令。这些指令在ALU中执行，ALU则在模块**exu\_alu\_ctl**内部实现。在“Project Explorer”（项目资源管理器）窗口中双击该模块将其打开。将显示图8所示的界面。

3. 通过在Verilog代码中右键单击某个信号并选择“Show In”（显示方式）→“Block Diagram”（框图），即可在框图中突出显示该信号。与突出显示的信号相关的线路也将在框图窗口中突出显示，如图9所示，其中突出显示的数据包为ap。



Imagination大学计划 – RVfpga SweRVref  
版本2.0 – 2021年11月30日  
© Copyright Imagination Technologies



4. 还可以通过双击框图中的模块，在Verilog代码中实现组合模块。例如，图10中显示了生成信号out的模块。

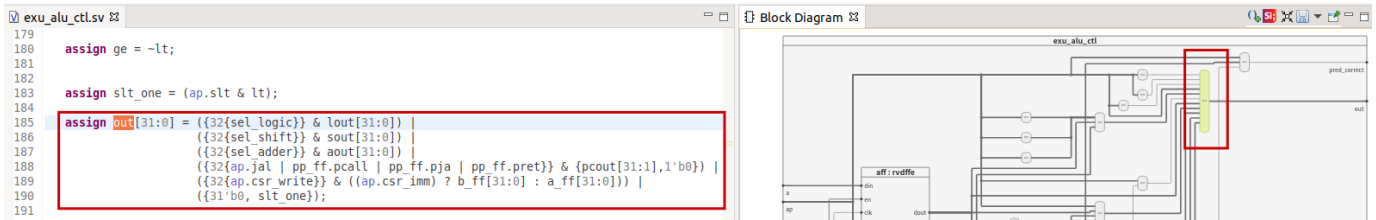


图10. 突出显示生成信号out的组合模块的Verilog代码

5. 最后，我们通过右键单击Verilog代码中的模块实例并选择“Open Declaration”（打开声明），在框图上打开一个模块声明。图11所示为在文件beh\_lib.sv中实现的模块rvdffe。

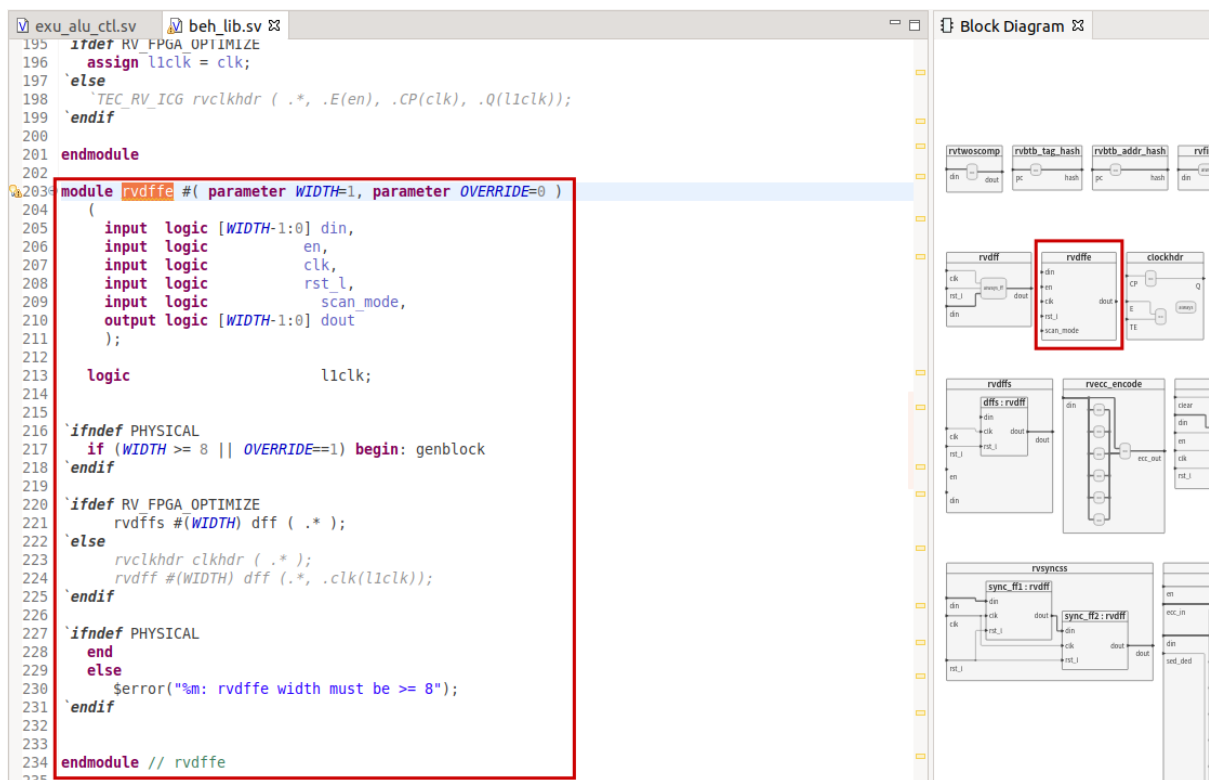


图11. 模块rvdffe

## 2. SWERV EH1处理器的配置

### A. 配置内核结构

[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/include/common\_defines.vh 允许用户配置内核的众多结构，如指令高速缓存、ICCM/DCCM、分支预测器等。RVfpga系统提供默认配置，您可以通过两种不同的方式更改配置：

- 可以在文件 `common_defines.vh` 中手动编辑参数。
- 可以使用Western Digital随SweRV EH1软件包一同提供的 `swerv.config` 脚本。有关该脚本的使用方法，请参见<https://github.com/chipsalliance/Cores-SweRV/tree/branch1.8> 在RVfpga中，可访问以下位置获取 `swerv.config` 脚本：  
[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/include/

如实验室1中所述，生成新的配置文件后，可以在Vivado中重新合成SoC，并获取新的RVfpga系统比特流。

### B. 禁止使用压缩指令

在某些情况下，我们可能希望禁止使用压缩指令。为此，必须对我们的PlatformIO进行两项更改：

- 在文件 `platformio.ini` 中包含以下新行：  

```
build_unflags = -Wa,-march=rv32imac -march=rv32imac
build_flags = -Wa,-march=rv32ima -march=rv32ima
extra_scripts = extra_script.py
```
- 将文件 `extra_script.py` 添加到项目的源代码中。该文件包含以下代码行：  

```
Import("env")
env.Append(
    LINKFLAGS=[
        "-Wa,-march=rv32ima",
        "-march=rv32ima"
    ]
)
```

在实验11-20使用的大多数示例中，为了简单起见，我们将禁止使用压缩指令。

### C. 使能/禁止内核功能

《SweRV EH1编程器参考手册》（[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)）中的表10-1列出了 `mfdc` 寄存器（CSR 0x7F9）位。该寄存器包含底层核心控制位，可禁止特定功能，如流水线或双发射执行、分支预测器等。表1所示为该寄存器可控制的九个内核功能。要使能或禁止各项内核功能，只需将寄存器的相应位清零或置1。例如，您可以在汇编程序中加入以下两条汇编指令，以禁止双发射执行、辅助ALU和流水线执行：

```
li t2, 0x481
csrrs t1, 0x7F9, t2
```

**表1. 功能禁止控制寄存器（*mfdc*: CSR 0x7F9）**

31-11	保留	7	0: 使能辅助ALU 1: 禁止辅助ALU	3	0: 使能分支预测和返回地址堆栈 1: 禁止分支预测和返回地址堆栈
10	0: 双发射执行 1: 单发射执行	6	0: 副作用存储采用流水线形式 1: 副作用存储阻止所有后续的总线事务，直至 存储发出收到默认值的响应	2	0: 使能写缓冲区合并 1: 禁止写缓冲区合并
9	保留	5	0: 使能非阻塞装载/除法 1: 禁止非阻塞装载/除法	1	保留
8	0: 使能ICCM/DCCM ECC检查 1: 禁止ICCM/DCCM ECC检查	4	0: 使能快速除法 1: 禁止快速除法	0	0: 流水线执行 1: 单指令执行

我们将在实验11-20中使用不同的配置，以便比较SweRV EH1在使能/禁止不同内核功能时的性能、I\$命中/未命中数和分支预测器中/未命中数等。

### 3. SweRV EH1内核的主要模块和信号

RVfpga系统在Nexys A7电路板的Artix-7 FPGA上运行，如图12所示。该图详细介绍了系统的层级，包括Verilog模块和子模块的名称。RVfpga系统由SweRVolf内核（**swervolf\_core**）、DRAM控制器（**litedram\_top**）、时钟生成模块（**clk\_gen\_nexys**）和一些接口模块组成。SweRVolf内核则由SweRV EH1处理器（**swerv\_wrapper\_dmi**）和其他接口模块（**wb\_intercon**、**axi\_intercon**和**uart\_top**等）组成。SweRV EH1处理器的顶层模块**swerv\_wrapper\_dmi**会对内核的两个主要模块**mem**和**swerv**进行实例化。在本文档的后续部分，我们将列出这两个模块的子模块和主要信号。请注意，您可以在模块接口处找到每个模块的其余信号。在实验11-20中，我们将专门研究这些信号，以分析处理器不同部分的操作。

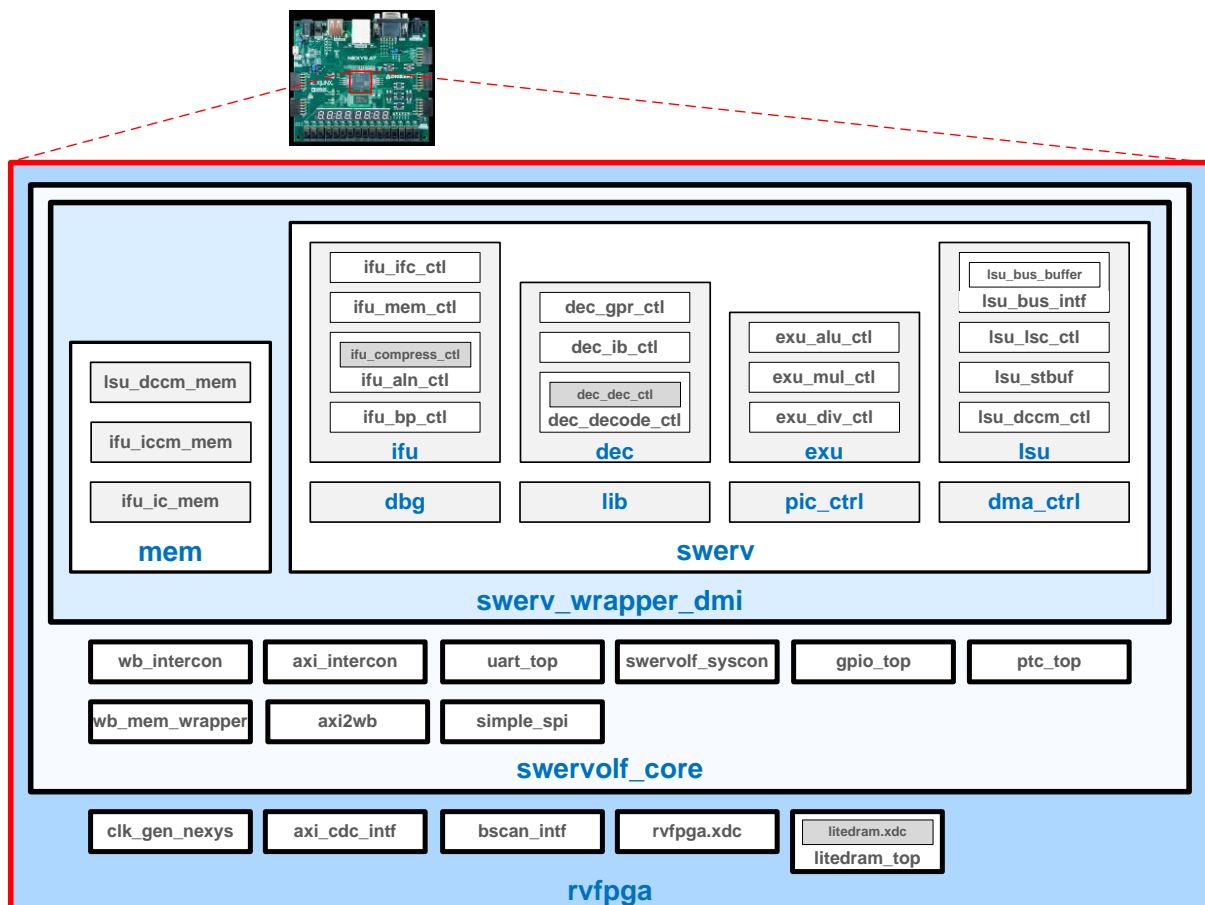


图12. RVfpga系统的层级

#### 模块: **mem**

**功能:** 此模块会对SweRV中提供的三个内部存储器进行实例化：ICCM、DCCM和I\$。表2列出了**mem**的子模块及其接口信号。

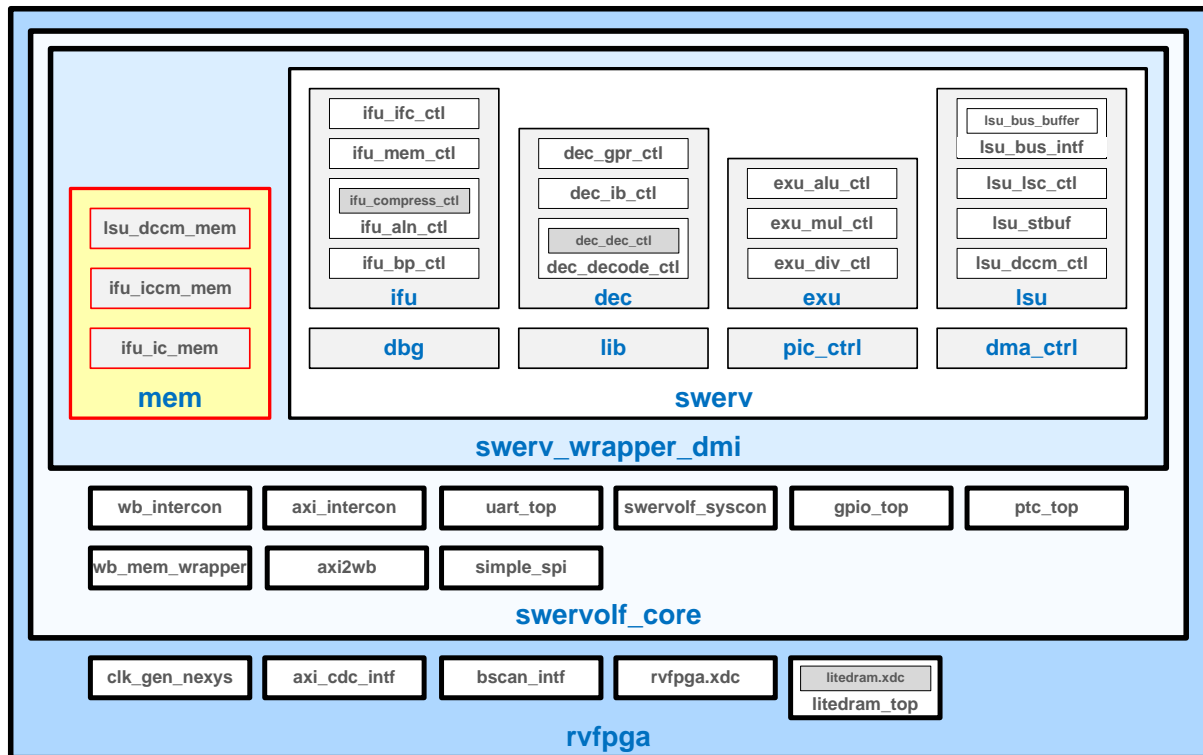


图13. 模块mem及其子模块

表2. Mem子模块和I/O

单元	I/O	名称	说明
ICCM: <b>ifu_iccm_mem</b> (其中包含ICCM 模块包装程序)	输入	iccm_wren	写使能
		iccm_rden	读使能
		[`RV_ICCM_BITS-1:2] iccm_rw_addr	读/写地址
		[77:0] iccm_wr_data	写数据
	输出	[155:0] iccm_rd_data	读数据
IS: <b>ifu_ic_mem</b> (其中包含指令高 速缓存数据和标记 模块包装程序)	输入	[3:0] ic_wr_en	写使能
		ic_rd_en	读使能
		[31:2] ic_rw_addr	读/写地址
		[67:0] ic_wr_data	用于填充指令高速缓存的数据。具有奇偶校验。
	输出	[135:0] ic_rd_data	从指令高速缓存读取的数据。F2阶段具有奇偶校验。
		[3:0] ic_rd_hit	每条通路中的命中/未命中
DCCM: <b>lsu_dccm_mem</b> (其中包含 DCCM模块包装 程序)	输入	dccm_wren	写使能
		dccm_rden	读使能
		[`RV_DCCM_BITS-1:0] dccm_wr_addr	写地址
		[`RV_DCCM_BITS-1:0] dccm_rd_addr_lo	读地址
		[`RV_DCCM_BITS-1:0] dccm_rd_addr_hi	进行未对齐访问时高位存储区的读地址

		[`RV_DCCM_FDATA_WIDTH-1:0] dccm_wr_data	写数据
	输出	[`RV_DCCM_FDATA_WIDTH-1:0] dccm_rd_data_lo	读数据低位存储区
		[`RV_DCCM_FDATA_WIDTH-1:0] dccm_rd_data_hi	读数据高位存储区

## 模块： **swerv**

**功能：**如图14所示，**swerv**是SweRV EH1内核的顶层模块。该模块会对内核的主要模块进行实例化，其中最重要的模块包括：**ifu**、**dec**、**exu**和**lsu**。表3至表6列出了这些单元的子模块和接口信号。**Swerv**模块通过SweRV包装程序（**swerv\_wrapper\_dmi**）与**mem**模块通信。

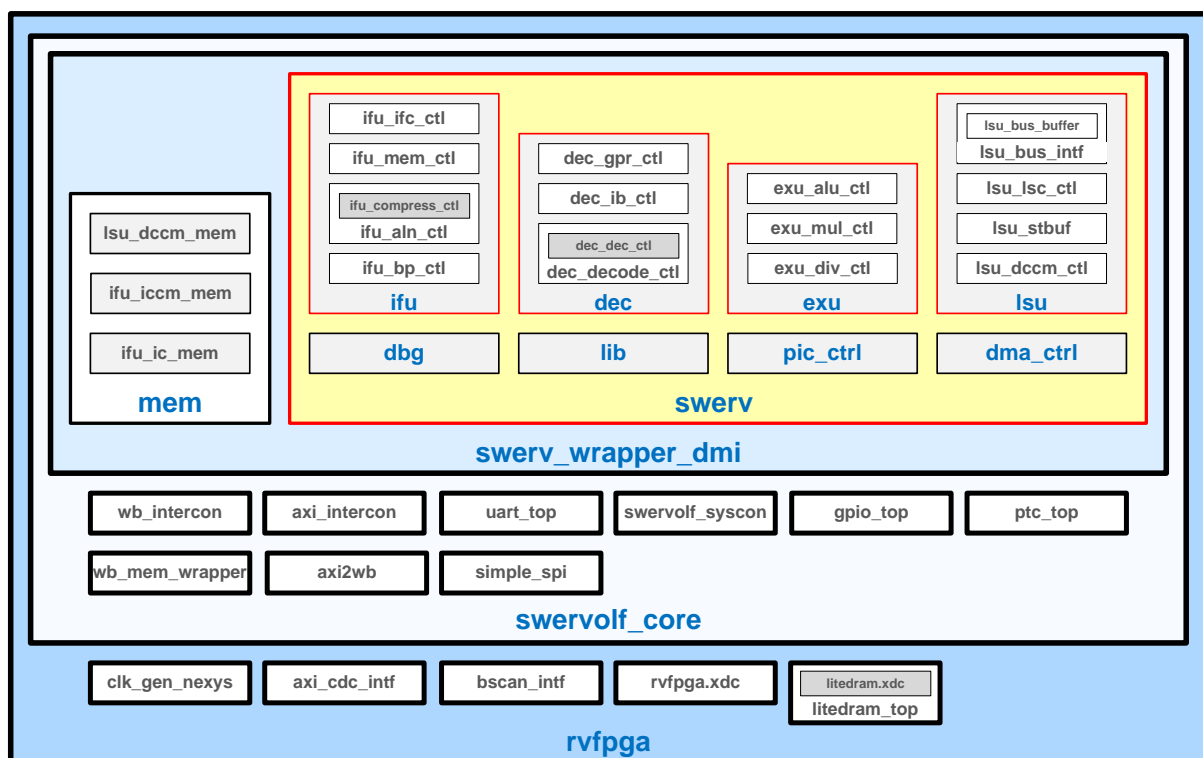


图14. **Swerv**及其子模块

表3. **Ifu**（取指单元）的I/O和子模块（包括子模块的I/O）

单元	I/O	名称	说明
取指单元： <b>ifu</b> （此模块是用于取指、分支预测器和对齐）	输入/输出	多个信号	用于 <b>mem</b> 模块输入/输出的ICCM端口
		多个信号	用于 <b>mem</b> 模块输入/输出的IS\$端口
		多个信号	IFU AXI端口
	输入	exu_flush_final	清除流水线
		[31:1] exu_flush_path_final	清除取指地址

器的顶层模块)	输出	[31:0] ifu_i0_instr	指令0。从对齐阶段到译码阶段
		[31:0] ifu_i1_instr	指令1。从对齐阶段到译码阶段
		[31:1] ifu_i0_pc	指令0 PC (程序计数器)。从对齐阶段到译码阶段
		[31:1] ifu_i1_pc	指令1 PC。从对齐阶段到译码阶段
取指控制: <b>ifu ifc_ctl</b> (此模块可实现取指管道控制, 生成从指令存储器中取指的下一个地址。)	输入	exu_flush_final	清除流水线
		[31:1] ifu_bp_btb_target_f2	预测的目标PC
		[31:1] exu_flush_path_final	清除路径
	输出	output logic [31:1] ifc_fetch_addr_f1	FC1阶段的取指地址
	内部	logic [31:1] fetch_addr_next	序列地址
指令存储器 (I\$和ICCM) 控制: <b>ifu_mem_ctl</b> (指令存储器控制 – lcache和ICCM –)	输入	[31:1] fetch_addr_f1	FC1阶段的取指地址 (重命名 ifc_fetch_addr_f1)
	输出	[127:0] ic_data_f2	I\$或ICCM在FC2阶段读取的数据, 将发送至对齐阶段
对齐控制: <b>ifu_aln_ctl</b> (指令对齐器)	输入	[127:0] ifu_fetch_data	来自取指阶段的128位取指数据
	内部	logic [127:0] q2, q1, q0	3个缓冲区
	输出	[31:0] ifu_i0_instr	指令通路0
		[31:0] ifu_i1_instr	指令通路1
		[31:1] ifu_i0_pc	指令通路0 PC
		[31:1] ifu_i1_pc	指令通路1 PC
分支预测器: <b>ifu_bp_ctl</b>	输入	[31:1] ifc_fetch_addr_f1	FC1阶段的取指地址
	输出	[31:1] ifu_bp_btb_target_f2	预测的目标PC
		ifu_bp_kill_next_f2	采取/未采取的分支

**表4. Dec（译码单元）的I/O和子模块（包括子模块的I/O）**

单元	I/O	名称	说明
译码单元: <b>dec</b> （此模块是用于指令译码、依赖性记分牌和寄存器文件访问的顶层模块）	输入	exu_flush_final	当信号值为1时，清除流水线
		[31:0] ifu_i0_instr, [31:1] ifu_i1_instr	来自对齐阶段的指令
		[31:1] ifu_i0_pc [31:1] ifu_i1_pc	来自对齐阶段的PC
	输出	alu_pkt_t i0_ap alu_pkt_t i1_ap	ALU控制信号
		lsu_pkt_t lsu_p	LSU控制信号
		mul_pkt_t mul_p	MUL控制信号
		div_pkt_t div_p	DIV控制信号
		predict_pkt_t i0_predict_p_d i1_predict_p_d	发送至ALU的预测信号
		[31:1] dec_i0_pc_d [31:1] dec_i1_pc_d	译码阶段的指令地址
		[31:0] gpr_i0_rs1_d [31:0] gpr_i0_rs2_d [31:0] gpr_i1_rs1_d [31:0] gpr_i1_rs2_d	来自寄存器文件的I0/I1 rs1/rs2数据
		[31:0] dec_i0_immed_d [31:0] dec_i1_immed_d	立即数值
		[12:1] dec_i0_br_immed_d [12:1] dec_i1_br_immed_d	分支偏移
		[31:0] i0_rs1_bypass_data_d [31:0] i0_rs2_bypass_data_d [31:0] i0_rs1_bypass_data_e2 [31:0] i0_rs2_bypass_data_e2 [31:0] i0_rs1_bypass_data_e3 [31:0] i0_rs2_bypass_data_e3	I0 rs1/rs2旁路数据
		[31:0] i1_rs1_bypass_data_d [31:0] i1_rs2_bypass_data_d [31:0] i1_rs1_bypass_data_e2 [31:0] i1_rs2_bypass_data_e2 [31:0] i1_rs1_bypass_data_e3 [31:0] i1_rs2_bypass_data_e3	I1 rs1/rs2旁路数据
		[31:0] dec_i0_instr_d [31:0] dec_i1_instr_d	译码阶段的指令
	内部		



		[31:0] dec_i0_rs1_d [31:0] dec_i0_rs2_d [31:0] dec_i1_rs1_d [31:0] dec_i1_rs2_d	rs1/rs2数据
从对齐阶段发送至译码阶段的指令/PC <b>dec_ib_ctl</b> (用于将指令和PC从对齐阶段传播到译码阶段的缓冲区)	输入	[31:0] ifu_i0_instr [31:0] ifu_i1_instr	对齐阶段的I0/I1指令
		[31:1] ifu_i0_pc [31:1] ifu_i1_pc	对齐阶段的I0/I1 PC
	输出	[31:0] dec_i0_instr_d [31:0] dec_i1_instr_d	译码阶段的I0/I1指令
		[31:1] dec_i0_pc_d [31:1] dec_i1_pc_d	译码阶段的I0/I1 PC
对指令进行译码并计算旁路值: <b>dec_decode_ctl</b> (对2条指令进行译码并计算旁路值)	输入	[31:1] dec_i0_pc_d [31:1] dec_i1_pc_d [31:0] exu_i0_result_e1	I0/I1 PC
		[31:0] dec_i0_instr_d, [31:0] dec_i1_instr_d	译码阶段的指令
	输出	alu_pkt_t i0_a alu_pkt_t i1_ap	ALU控制信号
		lsu_pkt_t lsu_p	LSU控制信号
		mul_pkt_t mul_p	MUL控制信号
		div_pkt_t div_p	DIV控制信号
		predict_pkt_t i0_predict_p_d i1_predict_p_d	发送至ALU的预测信号
		[4:0] dec_i0_rs1_d [4:0] dec_i0_rs2_d [4:0] dec_i1_rs1_d [4:0] dec_i1_rs2_d	I0/I1 rs1/rs2索引
		[31:0] dec_i0_immed_d [31:0] dec_i1_immed_d	立即数值
		[12:1] dec_i0_br_immed_d [12:1] dec_i1_br_immed_d	分支偏移
		[31:0] i0_rs1_bypass_data_d [31:0] i0_rs2_bypass_data_d [31:0] i0_rs1_bypass_data_e2 [31:0] i0_rs2_bypass_data_e2 [31:0] i0_rs1_bypass_data_e3 [31:0] i0_rs2_bypass_data_e3	I0 rs1/rs2旁路数据
		[31:0] i1_rs1_bypass_data_d [31:0] i1_rs2_bypass_data_d [31:0] i1_rs1_bypass_data_e2	I1 rs1/rs2旁路数据

		[31:0] i1_rs2_bypass_data_e2 [31:0] i1_rs1_bypass_data_e3 [31:0] i1_rs2_bypass_data_e3	
寄存器文件: <b>dec_gpr_ctl</b> (寄存器文件)	输入	[4:0] raddr0, raddr1	读地址
		[4:0] raddr2, raddr3	
		[4:0] waddr0, waddr1	写地址
		[4:0] waddr2	
		[31:0] wd0, wd1, wd2	写数据
		rden0, rden1, rden2, rden3	读使能
		wen0, wen1, wen2	写使能
	输出	[31:0] rd0, rd1, rd2, rd3	读数据

**表5. exu (执行单元) 的I/O和子模块 (包括子模块的I/O)**

单元	I/O	名称	说明
执行单元: <b>exu</b> (该模块为用于执行A-L指令的顶层模块)	输入	alu_pkt_t i0_ap, alu_pkt_t i1_ap	ALU控制
		mul_pkt_t mul_p	MUL控制
		div_pkt_t div_p	DIV控制
		[31:1] dec_i0_pc_d, dec_i1_pc_d	译码阶段的PC
		[31:0] gpr_i0_rs1_d	I0/I1 rs1/rs2
		[31:0] gpr_i0_rs2_d	
		[31:0] gpr_i1_rs1_d	
		[31:0] gpr_i1_rs2_d	
		[31:0] dec_i0_immed_d	立即数值
		[31:0] dec_i1_immed_d	
		[12:1] dec_i0_br_immed_d	分支偏移
		[12:1] dec_i1_br_immed_d	
		[31:0] i0_rs1_bypass_data_d	I0 rs1/rs2旁路数据
		[31:0] i0_rs2_bypass_data_d	
		[31:0] i0_rs1_bypass_data_e2	
		[31:0] i0_rs2_bypass_data_e2	
		[31:0] i0_rs1_bypass_data_e3	
		[31:0] i0_rs2_bypass_data_e3	
		[31:0] i1_rs1_bypass_data_d	I1 rs1/rs2旁路数据
		[31:0] i1_rs2_bypass_data_d	
		[31:0] i1_rs1_bypass_data_e2	
		[31:0] i1_rs2_bypass_data_e2	
		[31:0] i1_rs1_bypass_data_e3	
		[31:0] i1_rs2_bypass_data_e3	
	输出	exu_flush_final	当信号值为1时清除流水线
		[31:0] exu_i0_result_e1	主ALU结果
		[31:0] exu_i1_result_e1	
		[31:0] exu_i0_result_e4	辅助ALU结果
		[31:0] exu_i1_result_e4	
		[31:0] exu_mul_result_e3	MUL结果
		[31:0] exu_div_result	DIV结果
		[31:0] exu_lsu_rs1_d	装载/存储地址

		[31:0] exu_lsu_rs2_d	存储数据
ALU: <b>exu_alu_ctl</b> (算术逻辑单元)	输入	[31:0] a	A操作数
		[31:0] b	B操作数
		[31:1] pc	用于计算下一PC (即PC+2或PC+4)
		[12:1] brimm	分支偏移
		alu_pkt_t ap	ALU控制
	输出	[31:0] out	ALU结果
		flush_upper	分支清除
		[31:1] flush_path	目标PC
		[31:1] pc_ff	
乘法器: <b>exu_mul_ctl</b>	输入	[31:0] a	A操作数
		[31:0] b	B操作数
		mul_pkt_t mp	MUL控制
	输出	[31:0] out	MUL结果
除法器: <b>exu_div_ctl</b>	输入	[31:0] dividend	分子
		[31:0] divisor	分母
		div_pkt_t dp	DIV控制
	输出	[31:0] out	DIV结果

表6. *lsu* (装载/存储单元) 的I/O和子模块 (包括子模块的I/O)

单元	I/O	名称	说明
装载/存储单元: <b>lsu</b> (该模块为用于指令装载/存储单元的顶层模块)	输入/输出	多个信号	用于mem模块输入/输出的DCCM端口
		多个信号	DMA从端口
		多个信号	LSU AXI端口
	输入	[31:0] exu_lsu_rs1_d	装载/存储地址
		[31:0] exu_lsu_rs2_d	存储数据
		[11:0] dec_lsu_offset_d	地址偏移
		lsu_pkt_t lsu_p	LSU控制
	输出	[31:0] lsu_result_dc3	LSU读数据
地址计算: <b>lsu_lsc_ctl</b> (LSU控制并计算装载/存储地址)	输入	[31:0] exu_lsu_rs1_d	装载/存储地址
		[31:0] exu_lsu_rs2_d	存储数据
		[11:0] dec_lsu_offset_d	地址偏移
		lsu_pkt_t lsu_p	LSU控制
	输出	[31:0] lsu_addr_dc1 [31:0] end_addr_dc1	初始/最终地址
DCCM控制: <b>lsu_dccm_ctl</b> (DCCM控制)	输入	[`RV_DCCM_FDATA_WIDTH-1:0] dccm_rd_data_lo	读数据 (低位存储区)
		[`RV_DCCM_FDATA_WIDTH-1:0] dccm_rd_data_hi	读数据 (高位存储区)
	输出	dccm_wren	写使能
		dccm_rden	读使能
		[`RV_DCCM_BITS-1:0] dccm_wr_addr	写地址

		[`RV_DCCM_BITS-1:0] dccm_rd_addr_lo	读地址（低位存储区）
		[`RV_DCCM_BITS-1:0] dccm_rd_addr_hi	读地址（高位存储区）：未对齐装载需使用此地址
		[`RV_DCCM_FDATA_WIDTH-1:0] dccm_wr_data	写数据
存储缓冲区： <b>lsu_stbuf</b> （存储缓冲区）	输入	lsu_addr_dc3	地址
		[`RV_DCCM_DATA_WIDTH-1:0] store_ecc_datafn_hi_dc3	写数据（高位存储区）
		[`RV_DCCM_DATA_WIDTH-1:0] store_ecc_datafn_lo_dc3	写数据（低位存储区）
	输出	[`RV_LSU_SB_BITS-1:0] stbuf_addr_any	存储缓冲区地址
		[`RV_DCCM_DATA_WIDTH-1:0] stbuf_data_any	存储缓冲区数据

## 4. 用于对控制位进行分组的结构/类型

下文总结了文件

`[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/include/swerv_types.sv`中定义的、用于在SweRV EH1处理器中对控制信号进行分组的主要结构/类型。

- **dec\_pkt\_t**: 此类型为主要控制结构类型，其中包含alu（如果执行算术逻辑指令，则信号值为1，否则为0）、load（如果执行load指令，则信号值为1，否则为0）、legal（如果指令合法，则信号值为1，如果不合法，则为0）、rs1（如果指令从寄存器文件获得第一个输入操作数，则信号值为1，否则为0）、imm12（如果指令使用12位立即数作为输入操作数，则信号值为1，否则为0）等处理器主要控制信号。

此结构类型用于模块**dec\_decode\_ctl**内部，可生成许多其他控制信号。声明此类型的四个信号（通路0: i0\_dp\_raw, i0\_dp。通路1: i1\_dp\_raw, i1\_dp）并使用这些信号生成文件**swerv\_types.sv**定义的其他结构的控制位。

这些位在模块**dec\_dec\_ctl**内部分配，该模块可通过文件

`[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/dec/dec_decode_ctl.sv`末尾提供的开源工具（**coredecode**和**espresso**）自动生成。

- **alu\_pkt\_t**: 此结构类型包含与ALU操作相关的控制信号，例如valid（如果执行算术逻辑指令，则信号值为1，否则为0）、add（如果执行add指令，则信号值为1，否则为0）、beq（如果执行beq指令，则信号值为1，否则为0）等。此类型的两个信号称为i0\_ap和i1\_ap，在模块**dec\_decode\_ctl**中定义。

这些位在模块**dec\_decode\_ctl**（实现路径:

`[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/dec/dec_decode_ctl.sv`）中基于结构**dec\_pkt\_t**的位进行分配（参见**dec\_decode\_ctl**的第711-770行）。

- **reg\_pkt\_t**: 此结构类型包含两个源寄存器（字段rs1和rs2）和一个目标寄存器（字段rd）的标识符。此类型的两个信号称为i0r和i1r，在模块**dec\_decode\_ctl**内部定义。这些信号通过模块**dec\_decode\_ctl**内部指令寄存器的适当字段分配（参见本模块第1121-1127行）。
- **dest\_pkt\_t**: 此结构类型包含回写阶段使用的控制位，我们将在后续部分进行分析。此类型的一个信号称为dd，在模块**dec\_decode\_ctl**内部定义。
- **rets\_pkt\_t**、**br\_pkt\_t**、**br\_tlu\_pkt\_t**和**predict\_pkt\_t**: 这些结构类型与分支指令和分支预测器相关。
- **lsu\_pkt\_t**: 此结构类型包含与装载/存储单元相关的控制信号，例如half（如果读/写半字，则信号值为1，否则为0）、load（如果执行load指令，则信号值为1，否则为0）、valid（如果指令有效，则信号值为1，否则为0）等。此类型的一个信号称为lsu\_p，在模块**dec\_decode\_ctl**中定义。

- **mul\_pkt\_t**: 此结构类型包含与乘法单元相关的控制信号，例如rs1\_sign和rs2\_sign（用于确定输入操作数被视为有符号还是无符号）以及valid（如果指令有效，则信号值为1，否则为0）等。此类型的一个信号称为mul\_p，在模块**dec\_decode\_ctl**内部定义。
- **div\_pkt\_t**: 此结构类型包含与除法单元相关的控制信号，例如unsign（如果运算为无符号运算，则信号值为1，否则为0）和valid（如果指令有效，则信号值为1，否则为0）等。此类型的一个信号称为div\_p，在模块**dec\_decode\_ctl**内部定义。

**任务：** 打开文件

*[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/include/swerv\_types.sv*, 并在后续介绍用于对控制位进行分组的结构类型时分析该文件。

**任务：** 快速查看模块**dec\_decode\_ctl**和**dec\_dec\_ctl**，了解如何根据指令的32位来分配控制信号字段。这两个模块用途广泛且结构复杂，因此我们不打算对其进行详细分析。此外，查看模块**dec\_dec\_ctl**的自动创建过程，如**dec\_decode\_ctl.sv**的第2482-2495行所述。

## 5. 压缩指令

在我们的大多数实验中，为了简单起见，我们禁止使用了压缩指令，但在本部分中，我们描述并分析了RISC-V的压缩指令扩展（RVC）以及在SweRV EH1中执行压缩指令的情况。显然，您可以随意在实验中使能压缩指令，自行扩展分析。

**注：**开始本实验前，建议您先阅读S.Harris和D.Harris所著的《数字设计和计算机体系结构》（RISC-V版本，Morgan Kaufmann出版，简称[DDCARV]）的第6.6.5节。本部分的一些内容受这本书的启发。

RVC扩展通过减少控制、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。对于手持和移动应用来说，这些改进至关重要。由于SweRV EH1包含RVC，我们的汇编程序可以使用压缩指令和32位指令的组合。

在SweRV EH1中，有一个专用于解压指令的模块：**ifu\_compress\_ctl**。该模块会接收16位压缩指令，并输出相应的32位未压缩指令。在图15中，我们比实验11更详细地展示了对齐阶段的情况（我们仍然留下了一些黑框，供您自行分析）。三个**ifu\_compress\_ctl**模块在模块**ifu\_aln\_ctl**中实例化，从信号aligndata[63:0]处接收压缩指令，并在信号uncompress0[31:0]、uncompress1[31:0]和uncompress2[31:0]中返回相应的未压缩指令。如果指令已经为未压缩格式，则直接由aligndata[63:0]信号提供指令。

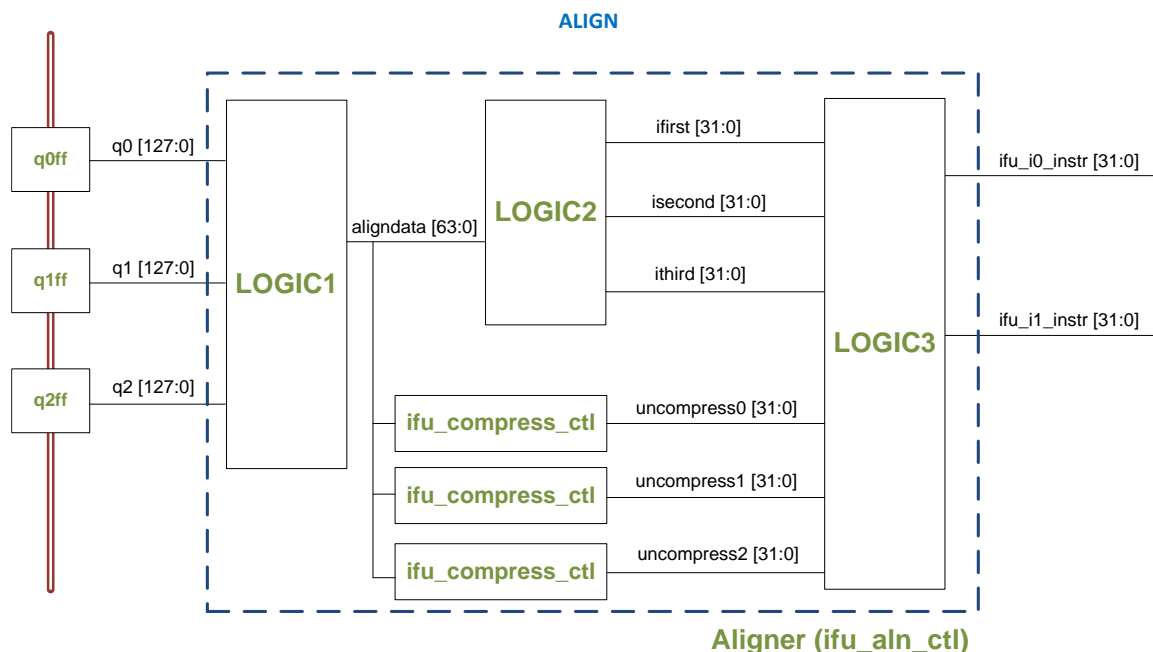


图15. 对齐阶段

图16上半部分所示的代码是DDCARV第6章示例31中的简单C程序。图16下半部分所示的代码是在使能RVC扩展的情况下，在PlatformIO中编译该C程序时生成的汇编代码（请注意，此汇编代码与[DDCARV]中所示的略有不同）。我们用红色突出显示构成循环主体的指令，其中同时包含16位和32位指令。

```

int scores[200];
int main(void) {
    int i;
    for (i = 0; i < 200; i = i + 1){
        scores[i] = scores[i] + 10;
    }
    return(0);
}

```

```

00000088 <main>:
88: 6789          lui      a5,0x2
8a: 12078793      addi     a5,a5,288 # 2120 <scores>
8e: 32078693      addi     a3,a5,800
92: 4398         lw      a4,0(a5)
94: 0791         addi    a5,a5,4
96: 0729         addi    a4,a4,10
98: fee7ae23     sw      a4,-4(a5)
9c: fed79be3     bne     a5,a3,92 <main+0xa>
a0: 4501          li      a0,0
a2: 8082          ret

```

图16. 压缩指令示例

图17所示为图16中循环的一次完整迭代的Verilator仿真结果。请注意，当指令addi a5,a5,4处于对齐阶段（在图中的第一个周期中以红色突出显示）时，将从64位束（aligndata[63:0]）中提取该指令，并将其从16位指令（0x0791）解压缩为32位指令（0x00478793）。（可访问[RVfpgaPath]/RVfpga/Labs/Lab11/Compressed\_C-Example获取代码，以便自行执行Verilator仿真。）

- 在RISC-V中，16位c.addi指令的操作码如下（参见[DDCARV]的附录B）：  
000 | imm(1-bit) | rd/rs1 | imm(5-bits) | 01

因此，可以轻松验证0x0791（0000011110010001）是否对应于c.addi a5,4（请记住，a5 = x15）。

- Imm = 000100
- rd = rs1 = 01111 (x15)

- 在RISC-V中，32位addi指令的操作码如下（参见[DDCARV]的附录B）：  
imm(12-bits) | rs1 | 000 | rd | 0010011

因此，可以轻松验证0x00478793（00000000010001111000011110010011）是否对应于addi a5,a5,4（请记住，a5 = x15）。

- Imm = 000000000100
- rs1 = 01111 (x15)
- rd = 01111 (x15)

在图17所示的第二个周期中，将对齐sw指令。由于该指令在RISC-V架构中缺乏相应的压缩版本，因此不需要解压缩，只需直接从aligndata[63:0]信号中选择指令并传播到译码阶段。



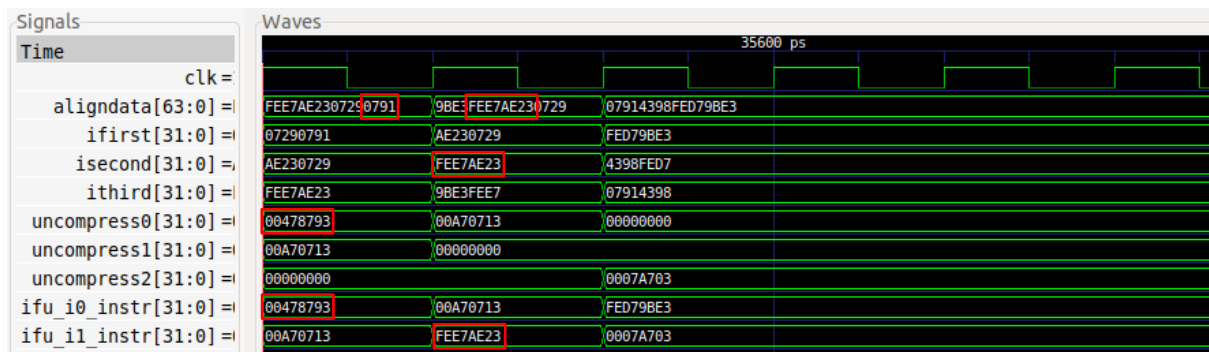


图17. 图16所示代码的仿真结果

**任务：**以压缩/未压缩指令为分类依据，分析循环主体中的其余指令。

**任务：**观察模块ifu\_compress\_ctl的内部结构，分析其工作原理。

## 6. 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，实验20中将使用它们来测试SweRV EH1处理器的不同功能。该实验将进一步介绍这三种基准，以及我们为每种基准提供的不同应用版本。

- **CoreMark:** 文件夹  
`[RVfpgaPath]/RVfpga/Labs/Lab20/RealBenchmarks/CoreMark_HwCounters`中提供了一个PlatformIO项目，其中包含在RVfpgaNexys上运行程序时的CoreMark基准。我们使用<https://github.com/chipsalliance/Cores-SweRV>提供的源代码对该基准进行了修改，使其能够适用于RVfpga系统。
- **Dhrystone:** 文件夹  
`[RVfpgaPath]/RVfpga/Labs/Lab20/RealBenchmarks/Dhrystone_HwCounters`中提供了一个PlatformIO项目，其中包含在RVfpgaNexys上运行程序时的Dhrystone基准。我们使用<https://github.com/chipsalliance/Cores-SweRV>提供的源代码对该基准进行了修改，使其能够适用于RVfpga系统。
- **图像处理:** 文件夹  
`[RVfpgaPath]/RVfpga/Labs/Lab20/RealBenchmarks/ImageProcessing_HwCounters`中提供了一个PlatformIO项目，其中包含我们在实验5中将RGB图像转换为灰度图像时所使用的应用程序。