



Imagination大学计划

RVfpga实验6

I/O简介

1. 简介

在实验6-10中，您将学习如何使用和扩展RVfpga的输入/输出（Input/Output，I/O）系统，以实现RISC-V处理器与外设的交互。下文概述了这些实验涵盖的主题：

- **实验6：**了解如何使用连接到Nexys A7开发板上的LED、开关和按钮的通用输入/输出（General-Purpose Input/Output，GPIO）引脚
- **实验7：**了解如何使用板上7段显示屏
- **实验8：**了解如何使用定时器
- **实验9：**了解如何使用中断与外部器件连接
- **实验10：**了解如何将RVfpga系统与板上SPI加速计连接

在本实验中，我们首先将介绍通用I/O系统的主要特性以及RVfpga系统中用到的特性（第2部分），然后介绍通用GPIO控制器的简化理论版本（第3部分）。最后，我们将重点关注SweRVolfX SoC中使用的GPIO控制器：首先分析其高级规范并介绍基本练习（第4部分和第5部分）。我们将通过分析其底层实现、在Verilator中仿真RVfpgaSim并介绍高级练习来总结实验（第6部分和第7部分）。

我们在实验7-10中使用相同的通用结构。在开始部分中，我们将先介绍I/O控制器的高级规范（其主要特性、寄存器及其操作以及存储器映射），然后介绍一些基本练习来实践如何使用外设。在高级部分中，我们将介绍控制器的底层实现，并通过练习帮助您了解如何修改底层实现以及编写测试修改的程序。

讲师注意事项：您可以根据课程级别选择练习的复杂度。例如，在第一年/第二年的课程（如计算机基础或计算机组成原理）中，基本练习（本实验的第5部分）十分合适。但是，在更高级的课程（如计算机体系结构或嵌入式系统设计）中，可以使用基础练习和高级练习（本实验的第5部分至第7部分）。

2. 输入/输出架构

图1阐述了冯·诺依曼架构的结构，这种架构由CPU、存储器和I/O系统三个主要模块组成。在实验6-10中，我们将重点介绍CPU与输入/输出（I/O）器件的交互。I/O器件也称为外设或简称为设备。下面我们将概述每个主要单元的作用：

- **CPU：**CPU是所有I/O操作的发起者，也是所有I/O事务的**控制器**（过去称为“主器件”，但该术语已被弃用）。直接存储器访问（Direct-Memory-Access，DMA）控制器（DMAC）也可以充当控制器，但本实验中不涉及。
- **设备控制器：***设备控制器*等待来自**控制器**的读/写请求来执行操作。设备控制器在I/O系统中充当**外设**（以前称为“从设备”，但该术语已被弃用）。从概念上讲，设备控制器由一系列可从**控制器**访问的**寄存器**组成。这些寄存器的值指示**外设**应当执行哪些操作。
- **互连**（总线和交叉等）在**控制器**和**外设**之间建立了一条路径。互连通常通过**桥接器**连接的多个层实现，作用是防止某些器件降低整个系统的速度。

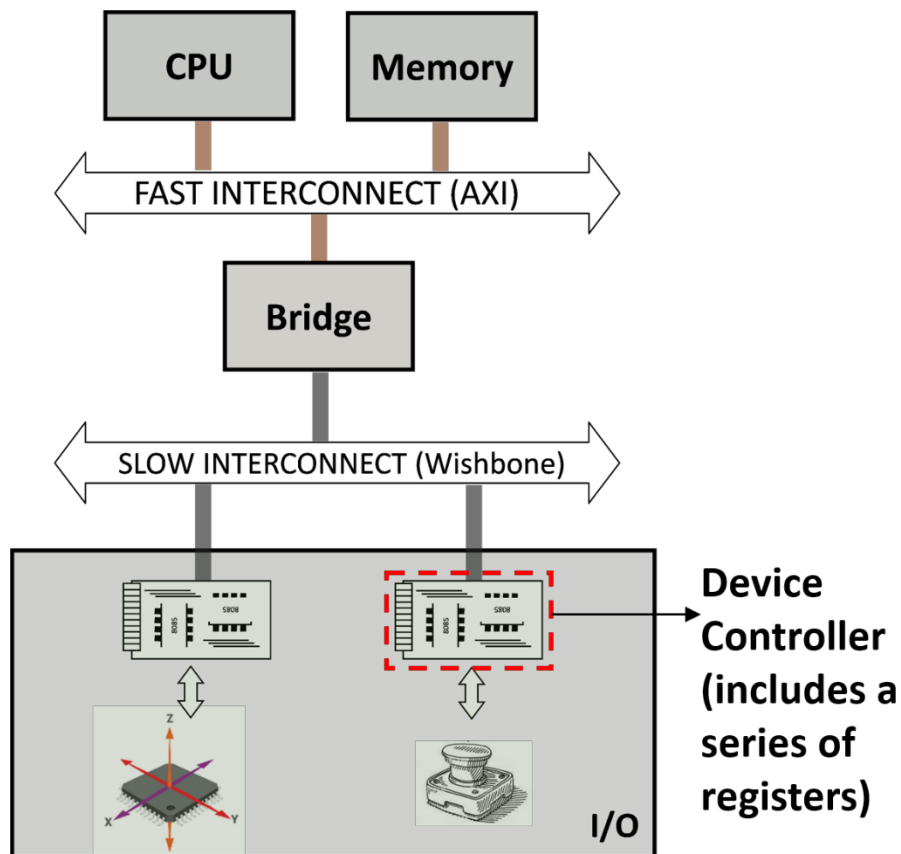


图1通用计算系统

图2显示了RVfpga的I/O系统。系统包括以下七个外设：

- 连接到GPIO1模块的LED和开关（被视为单个外设）
- 连接到系统控制器模块的7段显示屏
- 连接到SPI1模块的闪存
- 连接到SPI2模块的加速计
- 定时器
- UART
- 引导ROM

多路开关在七种可能性中选择一个外设并将其与CPU连接。请注意，必须采用Wishbone-AXI桥，因为外设使用Wishbone总线（灰色），而SweRV EH1内核使用AXI桥（橙色）。

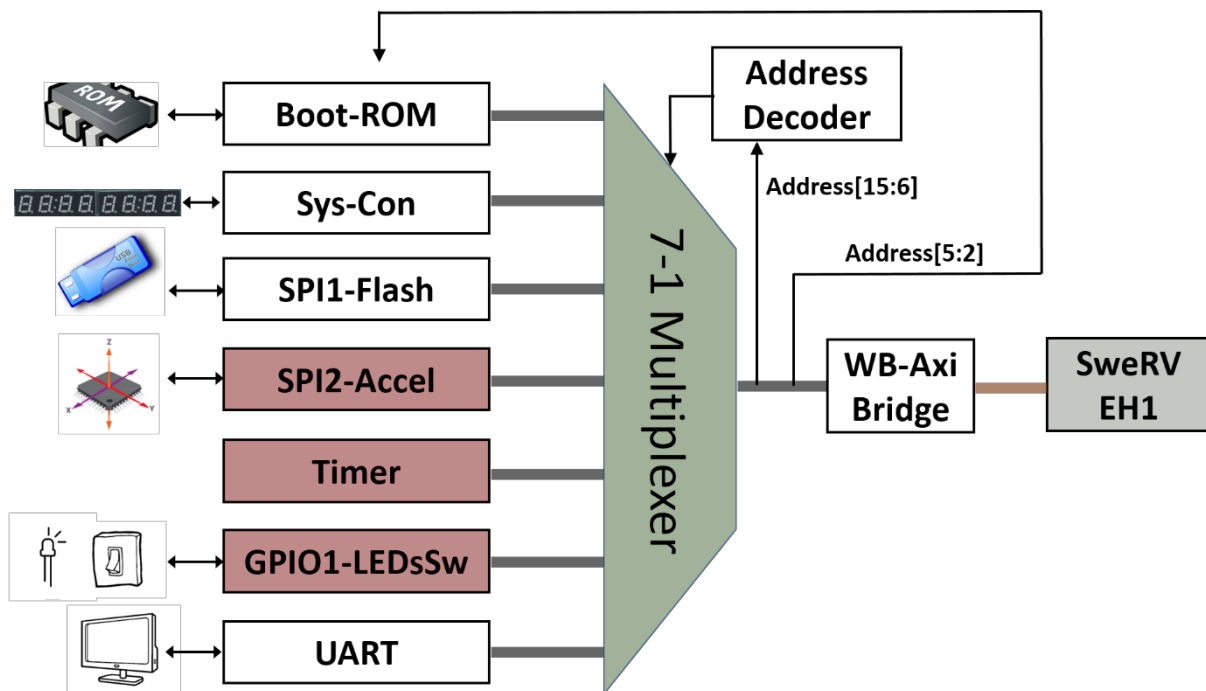


图2. RVfpga系统中的I/O系统

任务： 在SoC中找到图2的每个元件。您将需要检查以下文件和目录：

[RVfpgaPath]/RVfpga/src/SweRVolfSoC/swervolf_core.v (主文件，其中图2的元件已实例化)。
 [RVfpgaPath]/RVfpga/src/SweRVolfSoC/Peripherals
 [RVfpgaPath]/RVfpga/src/SweRVolfSoC/Interconnect/WishboneInterconnect
 [RVfpgaPath]/RVfpga/src/SweRVolfSoC/Peripherals/SystemController/swervolf_syscon.v
 [RVfpgaPath]/RVfpga/src/SweRVolfSoC/Interconnect/WishboneInterconnect/wb_intercon.v
 [RVfpgaPath]/RVfpga/src/SweRVolfSoC/Interconnect/WishboneInterconnect/wb_intercon.vh

如《RVfpga入门指南》中所述，原始SweRVolf (<https://github.com/chipsalliance/Cores-SweRVolf>) 仅包含图2所示的一些外设：具体包括引导ROM、系统控制器（无7段显示屏）、SPI闪存和UART（在图2中显示为白色）。根据本GSG，SweRVolfX SoC使用全新外设扩展原始SweRVolf SoC：SPI加速计、定时器、GPIO模块（在图2中显示为红色）和7段显示控制器（扩展SweRVolf的现有系统控制器）。

每个外设从处理器接收值和/或将值发送回处理器。存储器地址保留用于I/O值，分别称为寄存器、存储器映射I/O寄存器和设备控制器寄存器。要向外设发送值，CPU会将值存储到指定的存储器地址（即，存储器映射寄存器）。要从外设读取值，CPU从指定的存储器地址装载值。这样一来，CPU只需通过简单的装载/存储操作即可配置设备、检查设备状态或对设备执行数据读/写操作。

图2中的多路开关使用地址[15:6]选择请求的设备控制器。设备控制器使用地址[5:2]从几个寄存器中选择用来控制设备的寄存器。

3. 通用输入/输出（GPIO）

通用I/O（General-Purpose I/O，GPIO）控制器将外部数字引脚提供给程序员。在程序中的任何给定时间，这些引脚都可以配置为输入或输出。该名称是按引脚分配的，如有需要，可以在整个程序中更改。GPIO引脚可以连接到外部器件，例如LED、开关和按钮。

图3给出了将一个外部引脚连接到CPU的通用GPIO模块的简化图。该引脚可以连接到任何输入/输出器件，例如LED和开关等。在这张图中，该引脚连接到三态缓冲器（在图中以绿色突出显示）。该缓冲器允许程序员将引脚配置为输入或输出。如果使能三态缓冲器，则该引脚充当输出（例如，用于驱动LED）。如果禁止三态缓冲器，则该引脚充当输入（例如，用于读取开关值）。

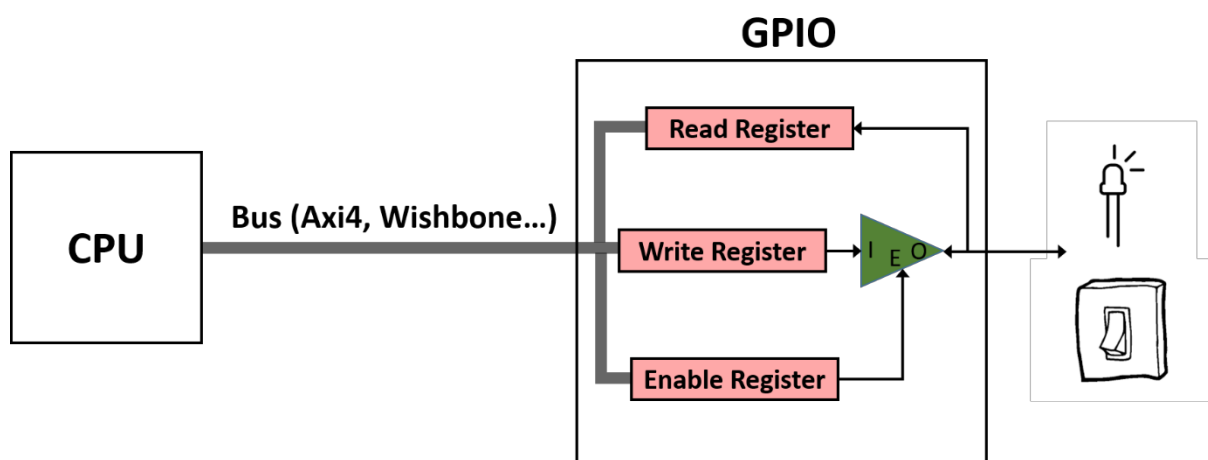


图3. GPIO简化电路

三态缓冲器既可以充当常规缓冲器（使能时），也可以具有悬空的输出（禁止时）。三态缓冲器具有两个输入（E（使能）和I（输入））和一个输出（O），其真值表如表1所示。当E为1时，三态缓冲器充当常规缓冲器，输出（O）和输入（I）相同。当E为0时，输入和输出之间不存在连接且输出（O）未被驱动；O处于悬空状态。在图3中，要将引脚配置为输出，E应为1，这样CPU便可驱动引脚。当引脚配置为输入时，E为0，这样CPU便无法驱动该引脚，此时可由外设驱动引脚。

表1. 三态真值表

E	I	O
0	0	高阻态
0	1	高阻态
1	0	0
1	1	1

RVfpga系统使用存储器映射I/O读写存储在这些寄存器中的值。例如，假设图3中的引脚连接到开关并且GPIO中的三个寄存器的映射如下：

- 读寄存器 = 地址0x80001400
- 写寄存器 = 地址0x80001404
- 使能寄存器 = 地址0x80001408

要读取开关的状态，请执行以下操作：

1. 通过向使能寄存器写入0（即，向地址0x80001408执行 *存储0* 的指令）来将引脚配置为输入。
2. 通过向地址0x80001400执行 *装载* 指令读取读寄存器。

4. GPIO高级规范

在本部分中，我们首先分析SweRVofX GPIO的高级规范，然后提供一个使用该外设的练习。

A. GPIO高级规范

可从OpenCores（<https://opencores.org/projects/gpio>）获取SweRVofX中使用的GPIO模块。OpenCore GPIO模块下载文件随附的gpio_spec.pdf文档介绍了模块的高级规范，这篇文档的下载地址为：*[RVfpgaPath]/RVfpga/src/SweRVofXSoC/Peripherals/gpio/docs/gpio_spec.pdf*。在本实验中，我们总结了GPIO模块的主要操作和特性。不过，您也可以在*gpio_spec.pdf*中获取完整的规范。

GPIO模块的主要特性如下：

- 使用Wishbone互连。
- 仅用作外设。
- 用户可以使用1-32个GPIO引脚。
- 可以并行使用多个GPIO模块（也称为GPIO内核）来访问超过32个GPIO引脚。
- 所有GPIO引脚都可以：
 - 用作双向引脚（这种情况下需要外部双向I/O单元）。
 - 使能三态或漏极开路（这种情况下需要外部三态或漏极开路I/O单元）。
- 编程为输入的GPIO引脚：
 - 可通过寄存器进行操作。
 - 可以向CPU发出中断请求。

GPIO内核规范的第4部分介绍了GPIO模块内部可用的控制和状态寄存器。其中每个寄存器都分配给一个不同的地址，如表2所示。GPIO寄存器的基址为0x80001400。

表2. GPIO寄存器

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

尽管OpenCore的GPIO模块比图3中所示的简化版本更为复杂，但我们仍可从图3识别出以下三个寄存器：读（输入）、写（输出）和使能。在OpenCore的GPIO模块中，这些寄存器分别称为：RGPIO_IN、RGPIO_OUT和RGPIO_OE，分别映射到0x80001400、0x80001404和0x80001408。

任务：在GPIO模块中找到寄存器RGPIO_IN、RGPIO_OUT和RGPIO_OE的声明及其地址的定义。GPIO模块位于：
`[RVfpgaPath]/RVfpga/src/SweRVolfSoC/Peripherals/gpio/gpio_top.v.`

RGPIO_IN寄存器锁存通用输入。RGPIO_OUT寄存器驱动通用输出。RGPIO_OE将每个I/O引脚配置为输入或输出。当使能位（位于RGPIO_OE内部）置1时，相应的通用输出驱动器将被使能，因此可将引脚连接到输出外设，例如LED。当使能位清零时，输出驱动器将在漏极开路（也称为三态或高阻态）模式下工作，因此可将引脚连接到输入外设，例如开关或按钮。

在RVfpgaNexys中，GPIO模块的前16个GPIO引脚（引脚15:0）连接到Nexys A7开发板上的16个LED。GPIO控制器的后16个GPIO引脚（引脚31:16）连接到16个板上开关。

5. 基本练习

练习1. 编写一个RISC-V汇编程序和一个C程序，此程序会显示一个由四个点亮的LED组成的模块重复地从板上16个LED的一侧移动到另一侧。另外，还包括两个控制速度和方向的开关。Switch[0]用于更改速度，而Switch[1]用于更改方向，如下所示：

- 如果Switch[0]为ON（高电平），则点亮的LED应快速移动。否则，点亮的LED应缓慢移动。可以定义“快”和“慢”的含义，但任何一种速度都必须明显可见，并且必须能够仅通过观察便察觉到速度的不同。
- 如果Switch[1]为ON（高电平），则点亮的LED应重复从右向左移动（当它们到达最左侧的LED时，将从右侧重新开始）。否则，点亮的LED应重复地从左向右移动。

下面的图4给出了Nexys A7开发板，其中突出显示了LED和开关。

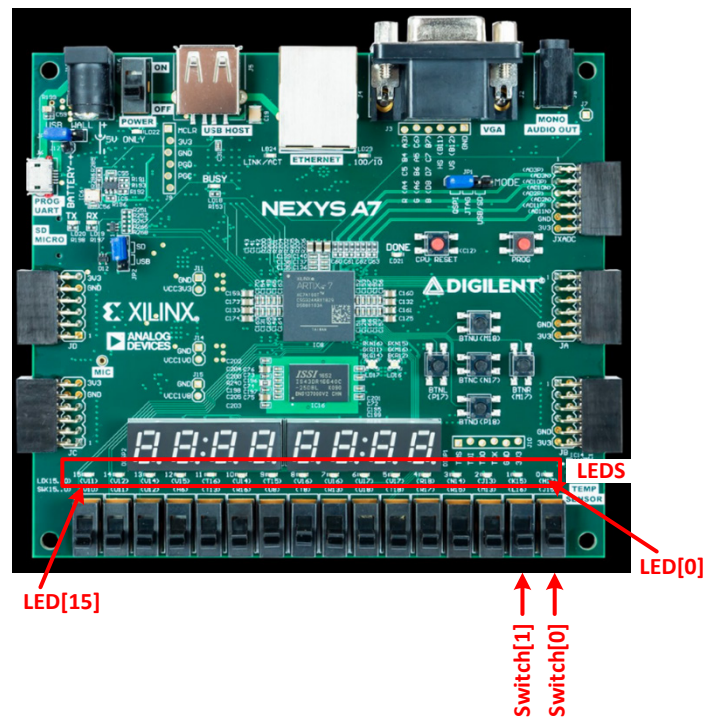


图4. Nexys A7 FPGA开发板：LED和开关

提示：回顾一下可知，开关已连接到寄存器映射I/O寄存器的引脚31:16。因此，要读取开关[0]，您需要向RGPIO_OE[16]写入0，然后读取RGPIO_IN[16]的值。您将需要适当地配置RGPIO_OE来访问其他LED和开关。

6. GPIO底层实现和仿真

在本部分中，我们将介绍SweRVolfX中使用的GPIO的底层细节。随后，我们会修改RVfpgaSim并在Verilator中执行一个示例仿真，以获得简单的汇编示例。最后，我们提供了一些练习，您需要先仿真RVfpgaSim，然后对其进行修改以添加新的GPIO外设，最后编写一个使用该新外设的程序。

A. GPIO底层实现

现在，您已经熟悉如何使用存储器映射I/O访问GPIO引脚，我们接下来深入了解GPIO的底层细节。GPIO可以分为三个主要部分，如图5所示：(1) RVfpgaNexys与板上LED/开关的外部连接（图5中的左侧阴影区域）；(2) GPIO模块到SweRVolfX SoC的集成（图5中的中间阴影区域）；(3) GPIO与SweRV EH1内核之间的连接（图5中的右侧阴影部分）。

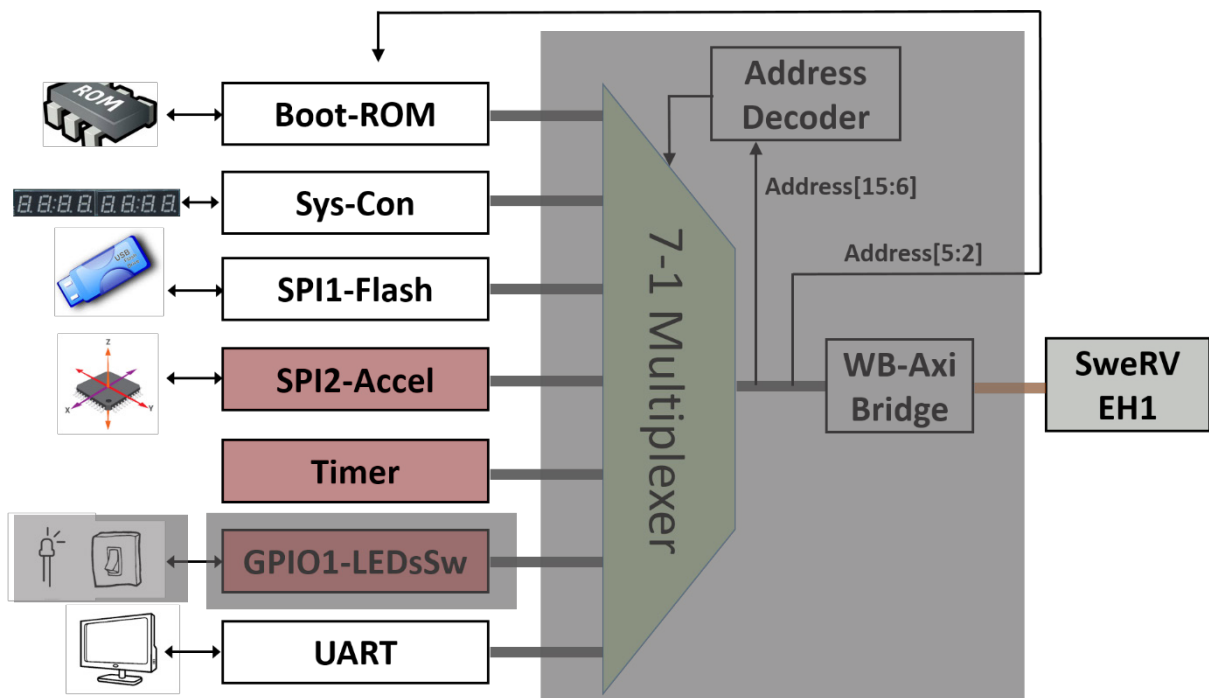


图5. GPIO分析（3个阶段）

i. LED/开关与SoC的连接

项目的约束文件（`[RVfpgaPath]/RVfpga/src/rvfpganexys.xdc`）定义了输入/输出SoC信号与开发板器件之间的连接。每个开发板器件都与给定的FPGA引脚关联。例如，板上最右边的开关Switch[0]通过印刷开发板（Printed Circuit Board, PCB）走线连接到FPGA引脚J15。

Nexys A7开发板包括16个LED和16个开关。将16个LED与SoC（称为rvfpganexys，在内部文件`[RVfpgaPath]/RVfpga/src/rvfpganexys.sv`中提供）的顶层模块相连的信号名称为`o_led[15:0]`，将16个开关与顶层模块相连的信号名称为`i_sw[15:0]`。图6给出了Xilinx设计约束（xdc）文件rvfpganexys.xdc部分（位于`[RVfpgaPath]/RVfpga/src`），其中定义了信号与FPGA引脚之间的这32个连接。

```

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] }]

```

图6. `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`rvfpganexys.xdc`)

顶层模块（`rvfpganexys`）的第48-49行给出了连接到SoC的这两个信号（图7的左侧部分），该模块的末尾给出了其与`swervolf_core`模块的连接（图7的右侧部分）。请注意，`i_sw`和`o_led`信号合并到信号`io_data`中（第257行），这是与`swervolf_core`模块中的GPIO连接的32位输入/输出信号（如后面的图8所示）。此外，请注意`o_led`信号通过中间信号`gpio_out`（第266行）锁存。

```

25 module rvfpganexys
26     #(parameter bootrom_file = "boot_main.mem")
27     input wire      clk,
28     input wire      rstn,
29     output wire [12:0] ddram_a,
30     output wire [2:0] ddram_ba,
31     output wire      ddram_ras_n,
32     output wire      ddram_cas_n,
33     output wire      ddram_we_n,
34     output wire      ddram_cs_n,
35     output wire [1:0] ddram_dm,
36     inout wire [15:0] ddram_dq,
37     inout wire [1:0] ddram_dqs_p,
38     inout wire [1:0] ddram_dqs_n,
39     output wire      ddram_clk_p,
40     output wire      ddram_clk_n,
41     output wire      ddram_cke,
42     output wire      ddram_odt,
43     output wire      o_flash_cs_n,
44     output wire      o_flash_mosi,
45     input wire       i_flash_miso,
46     input wire       i_uart_rx,
47     output wire      o_uart_tx,
48     inout wire [15:0] i_sw,
49     output reg [15:0] o_led,

```

```

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

```

图7. LED与开关和顶层模块的连接（`rvfpganexys.sv`）

任务：跟踪从约束文件到SweRVolf SoC模块的这两个信号（`i_sw`和`o_led`），它们在SweRVolf SoC模块中合并为`io_data`。您将需要检查以下文件：

`[RVfpgaPath]/RVfpga/src/rvfpganexys.xdc`
`[RVfpgaPath]/RVfpga/src/rvfpganexys.sv`

[RVfpgaPath]/RVfpga/src/SweRVolfSoC/swervolf_core.v

在上一部分中我们提到过，在RVfpgaNexys中，GPIO模块的前16个GPIO引脚（15到0）连接到16个板上LED，而GPIO控制器的后16个GPIO引脚（31到16）连接到16个板上开关。这是否符合本部分和图8中描述的实现？

ii. GPIO模块到SoC的集成

在swervolf_core模块（[RVfpgaPath]/RVfpga/src/SweRVolfSoC/swervolf_core.v）的第299-354行中，GPIO模块经过实例化并集成到SoC中（参见图8）。

```

299 // GPIO - Leds and Switches
300 wire [31:0] en_gpio;
301 wire      gpio_irq;
302 wire [31:0] i_gpio;
303 wire [31:0] o_gpio;
304
305 bidirec gpio0 (.oe(en_gpio[0]), .inp(o_gpio[0]), .outp(i_gpio[0]), .bidir(io_data[0]));
306 bidirec gpio1 (.oe(en_gpio[1]), .inp(o_gpio[1]), .outp(i_gpio[1]), .bidir(io_data[1]));
307 bidirec gpio2 (.oe(en_gpio[2]), .inp(o_gpio[2]), .outp(i_gpio[2]), .bidir(io_data[2]));
308 bidirec gpio3 (.oe(en_gpio[3]), .inp(o_gpio[3]), .outp(i_gpio[3]), .bidir(io_data[3]));
309 bidirec gpio4 (.oe(en_gpio[4]), .inp(o_gpio[4]), .outp(i_gpio[4]), .bidir(io_data[4]));
310 bidirec gpio5 (.oe(en_gpio[5]), .inp(o_gpio[5]), .outp(i_gpio[5]), .bidir(io_data[5]));
311 bidirec gpio6 (.oe(en_gpio[6]), .inp(o_gpio[6]), .outp(i_gpio[6]), .bidir(io_data[6]));
312 bidirec gpio7 (.oe(en_gpio[7]), .inp(o_gpio[7]), .outp(i_gpio[7]), .bidir(io_data[7]));
313 bidirec gpio8 (.oe(en_gpio[8]), .inp(o_gpio[8]), .outp(i_gpio[8]), .bidir(io_data[8]));
314 bidirec gpio9 (.oe(en_gpio[9]), .inp(o_gpio[9]), .outp(i_gpio[9]), .bidir(io_data[9]));
315 bidirec gpio10 (.oe(en_gpio[10]), .inp(o_gpio[10]), .outp(i_gpio[10]), .bidir(io_data[10]));
316 bidirec gpio11 (.oe(en_gpio[11]), .inp(o_gpio[11]), .outp(i_gpio[11]), .bidir(io_data[11]));
317 bidirec gpio12 (.oe(en_gpio[12]), .inp(o_gpio[12]), .outp(i_gpio[12]), .bidir(io_data[12]));
318 bidirec gpio13 (.oe(en_gpio[13]), .inp(o_gpio[13]), .outp(i_gpio[13]), .bidir(io_data[13]));
319 bidirec gpio14 (.oe(en_gpio[14]), .inp(o_gpio[14]), .outp(i_gpio[14]), .bidir(io_data[14]));
320 bidirec gpio15 (.oe(en_gpio[15]), .inp(o_gpio[15]), .outp(i_gpio[15]), .bidir(io_data[15]));
321 bidirec gpio16 (.oe(en_gpio[16]), .inp(o_gpio[16]), .outp(i_gpio[16]), .bidir(io_data[16]));
322 bidirec gpio17 (.oe(en_gpio[17]), .inp(o_gpio[17]), .outp(i_gpio[17]), .bidir(io_data[17]));
323 bidirec gpio18 (.oe(en_gpio[18]), .inp(o_gpio[18]), .outp(i_gpio[18]), .bidir(io_data[18]));
324 bidirec gpio19 (.oe(en_gpio[19]), .inp(o_gpio[19]), .outp(i_gpio[19]), .bidir(io_data[19]));
325 bidirec gpio20 (.oe(en_gpio[20]), .inp(o_gpio[20]), .outp(i_gpio[20]), .bidir(io_data[20]));
326 bidirec gpio21 (.oe(en_gpio[21]), .inp(o_gpio[21]), .outp(i_gpio[21]), .bidir(io_data[21]));
327 bidirec gpio22 (.oe(en_gpio[22]), .inp(o_gpio[22]), .outp(i_gpio[22]), .bidir(io_data[22]));
328 bidirec gpio23 (.oe(en_gpio[23]), .inp(o_gpio[23]), .outp(i_gpio[23]), .bidir(io_data[23]));
329 bidirec gpio24 (.oe(en_gpio[24]), .inp(o_gpio[24]), .outp(i_gpio[24]), .bidir(io_data[24]));
330 bidirec gpio25 (.oe(en_gpio[25]), .inp(o_gpio[25]), .outp(i_gpio[25]), .bidir(io_data[25]));
331 bidirec gpio26 (.oe(en_gpio[26]), .inp(o_gpio[26]), .outp(i_gpio[26]), .bidir(io_data[26]));
332 bidirec gpio27 (.oe(en_gpio[27]), .inp(o_gpio[27]), .outp(i_gpio[27]), .bidir(io_data[27]));
333 bidirec gpio28 (.oe(en_gpio[28]), .inp(o_gpio[28]), .outp(i_gpio[28]), .bidir(io_data[28]));
334 bidirec gpio29 (.oe(en_gpio[29]), .inp(o_gpio[29]), .outp(i_gpio[29]), .bidir(io_data[29]));
335 bidirec gpio30 (.oe(en_gpio[30]), .inp(o_gpio[30]), .outp(i_gpio[30]), .bidir(io_data[30]));
336 bidirec gpio31 (.oe(en_gpio[31]), .inp(o_gpio[31]), .outp(i_gpio[31]), .bidir(io_data[31]));
337
338 gpio_top gpio_module(
339     .wb_clk_i      (clk),
340     .wb_rst_i      (wb_rst),
341     .wb_cyc_i      (wb_m2s_gpio_cyc),
342     .wb_adr_i      ({2'b0,wb_m2s_gpio_adr[5:2],2'b0}),
343     .wb_dat_i      (wb_m2s_gpio_dat),
344     .wb_sel_i      (4'b1111),
345     .wb_we_i      (wb_m2s_gpio_we),
346     .wb_stb_i      (wb_m2s_gpio_stb),
347     .wb_dat_o      (wb_s2m_gpio_dat),
348     .wb_ack_o      (wb_s2m_gpio_ack),
349     .wb_err_o      (wb_s2m_gpio_err),
350     .wb_inta_o     (gpio_irq),
351     // External GPIO Interface
352     .ext_pad_i     (i_gpio[31:0]),
353     .ext_pad_o     (o_gpio[31:0]),
354     .ext_padoe_o   (en_gpio));
355

```

图8. GPIO模块的集成（文件swervolf_core.v）

模块的接口可以分为两个部分：允许SweRV EH1内核使用控制器/外设模型与GPIO通信的Wishbone信号（表3）以及外部I/O信号（表4）。

表3. Wishbone信号

端口	宽度	方向	说明
wb_cyc_i	1	输入	指示有效的总线周期（内核选择）
wb_adr_i	15	输入	地址输入
wb_dat_i	32	输入	数据输入
wb_dat_o	32	输出	数据输出
wb_sel_i	4	输入	指示数据总线上的有效字节（在有效周期内，必须为0xf）
wb_ack_o	1	输出	应答输出（指示正常事务终止）
wb_err_o	1	输出	错误应答输出（指示异常事务终止）
wb_rty_o	1	输出	未使用
wb_we_i	1	输入	置为高电平时写事务
wb_stb_i	1	输入	指示有效的数据传输周期
wb_inta_o	1	输出	中断输出

表4. 外部I/O信号

端口	宽度	方向	说明
in_pad_i	1-32	输入	GPIO输入
out_pad_o	1-32	输出	GPIO输出
oen_padoen_o	1-32	输出	GPIO输出驱动器使能（用于三态或漏极开路驱动器）

如图8的第342行所示，Wishbone总线信号中由内核提供的地址的位5:2

（`wb_m2s_gpio_adr[5:2]`）用于从10个可用的存储器映射寄存器中选择1个。这四位通过`wb_adr_i`信号提供给GPIO内核（如图8所示）。

输入`ext_pad_i`直接与GPIO读寄存器（`RGPIO_IN`）连接。同样，输出`ext_pad_o`直接与GPIO写寄存器（`RGPIO_OUT`）连接。这两个信号通过32个三态缓冲器模块连接到LED和开关（`i_gpio`、`o_gpio`和`io_data`）（图8的第305-336行）。这样，全部32个引脚都可以配置为输入或输出。本例中的低16个引脚（引脚15:0）连接到LED（图7），因此必须配置为输出；高16个引脚（31:16）连接到开关（图7），因此必须配置为输入。我们通过在`swervolf_core`模块的末尾（第634-640行）包含以下模块来实现这32个三态缓冲器：

```
module bidirec (input wire oe, input wire inp, output wire outp, inout wire bidir);
    assign bidir = oe ? inp : 1'bZ ;
    assign outp = bidir;
endmodule
```

任务：GPIO引脚（`io_data`）通过三态缓冲器连接到GPIO模块（见图8）。分析三态缓冲器的使能信号的两种可能状态（`oe = 0`和`oe = 1`）。

考虑到GPIO模块和板上LED/开关之间的连接，编程器应将哪些值分配给`en_gpio`？

iii. GPIO与SweRV EH1内核的连接

如图2所示，设备控制器通过多路开关和桥接器连接到SweRV EH1内核。多路开关根据CPU生成的地址在N个可能的外设中选择一个（本例中N = 7）。桥接器将设备控制器使用的Wishbone信号转换为SweRV内核使用的AXI4信号，反之亦然（在文件 `[RVfpgaPath]/RVfpga/src/SweRVolfSoC/Interconnect/AxiToWb/axi2wb.v` 中实现）。

7:1多路开关（图9）在文件

`[RVfpgaPath]/RVfpga/src/SweRVolfSoC/Interconnect/WishboneInterconnect/wb_intercon.v` 中实现，该文件在

`[RVfpgaPath]/RVfpga/src/SweRVolfSoC/Interconnect/WishboneInterconnect/wb_intercon.vh`

文件的第104-205行实例化。后一个文件包含在`swervolf_core`模块的第168行，该模块位于：

`[RVfpgaPath]/RVfpga/src/SweRVolfSoC/swervolf_core.v`。

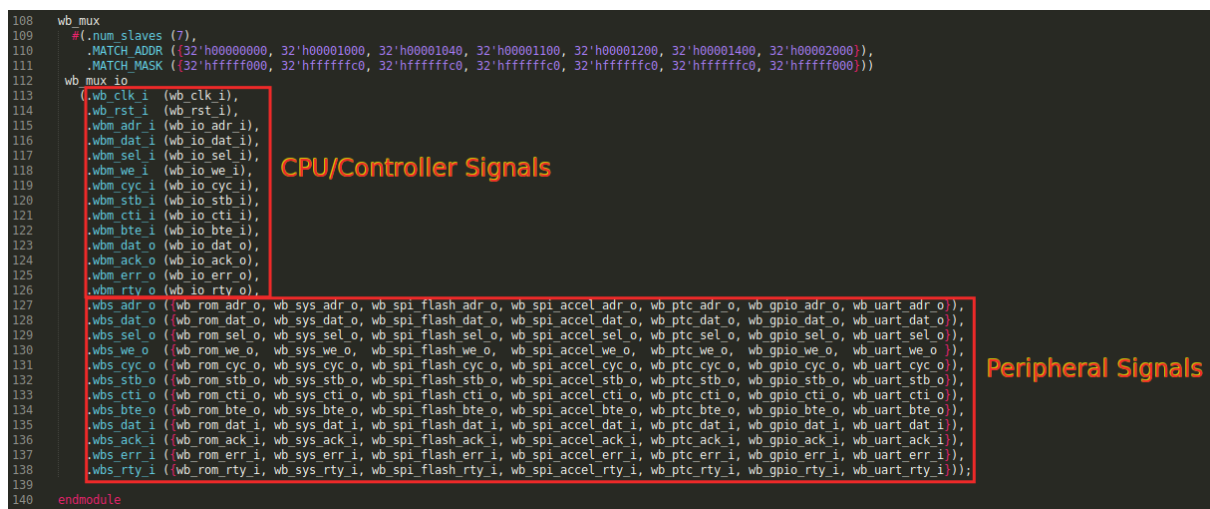


图9. 7-1多路开关选择与CPU连接的外设（`wb_intercon.v`）

多路开关选择要读取或写入哪个外设，根据地址（第110-111行）将CPU（`wb_io_*`信号 – 图9的第115-126行）与一个外设的Wishbone总线（图9的第127-138行）连接。例如，如果CPU生成的地址在0x80001400-0x8000143F范围内，则选择GPIO外设，从而将信号`wb_io_*`与信号`wb_gpio_*`连接。

图10给出了多路开关的Verilog实现（位于文件

`[RVfpgaPath]/RVfpga/src/SweRVolfSoC/Interconnect/WishboneInterconnect/wb_intercon_1.2.2-r1/wb_mux.v`中）。

任务： 详细分析多路开关的实现。您可以重点关注与GPIO相关的信号（`wb_gpio_*`）。您将需要检查以下文件：

`[RVfpgaPath]/RVfpga/src/SweRVolfSoC/Peripherals/SystemController/swervolf_syscon.v`
`[RVfpgaPath]/RVfpga/src/SweRVolfSoC/Interconnect/WishboneInterconnect/wb_intercon.v`
`[RVfpgaPath]/RVfpga/src/SweRVolfSoC/Interconnect/WishboneInterconnect/wb_intercon.vh`
`[RVfpgaPath]/RVfpga/src/SweRVolfSoC/Interconnect/WishboneInterconnect/wb_intercon_1.2.2-r1/wb_mux.v`

理解SoC的这一部分对本实验以及后面的实验都十分重要。如果您通过添加与多路开关相关的新信号来扩展仿真，则在下一部分中执行的仿真可以帮助您加深理解。

```

82 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
83 // Master/slave connection
84 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
85
86 //Use parameter instead of localparam to work around a bug in Xilinx ISE
87 parameter slave_sel_bits = num_slaves > 1 ? $clog2(num_slaves) : 1;
88
89 reg wbm_err;
90 wire [slave_sel_bits-1:0] slave_sel;
91 wire [num_slaves-1:0] match;
92
93 genvar idx;
94
95 generate
96     for(idx=0; idx<num_slaves ; idx=idx+1) begin : addr_match
97         assign match[idx] = (wbm_adr_i & MATCH_MASK[idx*aw+:aw]) == MATCH_ADDR[idx*aw+:aw];
98     end
99 endgenerate
100
101 //
102 // Find First 1 - Start from MSB and count downwards, returns 0 when no bit set
103 //
104 function [slave_sel_bits-1:0] ffl;
105     input [num_slaves-1:0] in;
106     integer i;
107
108     begin
109         ffl = 0;
110         for (i = num_slaves-1; i >= 0; i=i-1) begin
111             if (in[i])
112                 ffl = i;
113         end
114     end
115 endfunction
116
117 assign slave_sel = ffl(match);
118
119 always @(posedge wb_clk_i)
120     wbm_err <= wbm_cyc_i & !(match);
121
122 assign wbs_adr_o = {num_slaves{wbm_adr_i}};
123 assign wbs_dat_o = {num_slaves{wbm_dat_i}};
124 assign wbs_sel_o = {num_slaves{wbm_sel_i}};
125 assign wbs_we_o = {num_slaves{wbm_we_i}};
126
127 /* verilator lint_off WIDTH */
128 assign wbs_cyc_o = match & (wbm_cyc_i << slave_sel);
129 /* verilator lint_on WIDTH */
130 assign wbs_stb_o = {num_slaves{wbm_stb_i}};
131
132 assign wbs_cti_o = {num_slaves{wbm_cti_i}};
133 assign wbs_bte_o = {num_slaves{wbm_bte_i}};
134
135 assign wbm_dat_o = wbs_dat_i[slave_sel*dw+:dw];
136 assign wbm_ack_o = wbs_ack_i[slave_sel];
137 assign wbm_err_o = wbs_err_i[slave_sel] | wbm_err;
138 assign wbm_rty_o = wbs_rty_i[slave_sel];
139
140 endmodule

```

图10. Wishbone多路开关（文件**wb_mux.v**）

B. Verilator仿真

在本部分中，我们首先通过添加新的输入信号来修改RVfpgaSim仿真器。然后，我们使用Verilator重新编译RVfpgaSim，并在仿真器执行简单程序时分析该新信号。

i. 修改并重新编译RVfpgaSim

仿真过程中没有真正的LED或开关。因此，在测试平台（`[RVfpgaPath]/RVfpga/src/rvfpgasim.v`）中，我们将通过为该信号（`i_sw`）分配常量值0xFE34（图11的左侧部分）来仿真开关驱动。开关随后将作为输入提供给SweRVolfX SoC（图11的右侧部分）。


```
80    wire [15:0] i_sw;
81    assign i_sw = 16'hFE34; 248    .io_data      ({i_sw,16'bz}));
```

图11. 信号*i_sw*在*rvfpgasim.v*中分配并传递到SweRVolfX SoC。

根据《入门指南》，测试平台（*rvfpgasim.v*）接收RVfpgaSim的输入信号（*clk*和*rst*等）（图12的左侧部分）并实例化*swervolf_core*模块（图12的右侧部分）。

```
28    (input wire clk,
29    input wire rst,
30    input wire i_jtag_tck,
31    input wire i_jtag_tms,
32    input wire i_jtag_tdi,
33    input wire i_jtag_trst_n,
34    output wire o_jtag_tdo,
35    output wire o_uart_tx,
36    output wire o_gpio)

190    swervolf_core
191    #(.bootrom_file (bootrom_file))
192    swervolf
193    (.clk (clk),
194    .rstn (!rst),
195    .dmi_reg_rdata (dmi_reg_rdata),
```

图12. RVfpgaSim和SweRVolfX实例化的输入信号（文件*rvfpgasim.v*）

在某些情况下，您可能需要向仿真器添加新的输入/输出信号。在下面的示例中，我们将说明如何包含RVfpgaSim的输入信号（名称为*i_sw0*），此信号提供了最右侧开关的值。

请按下面的步骤操作：

1. 修改文件[RVfpgaPath]/RVfpga/src/rvfpgasim.v:

- a. 包含新的1位输入信号（名称为*i_sw0*）。请参见图13。

```
28    (input wire clk,
29    input wire rst,
30    input wire i_jtag_tck,
31    input wire i_jtag_tms,
32    input wire i_jtag_tdi,
33    input wire i_jtag_trst_n,
34    output wire o_jtag_tdo,
35    output wire o_uart_tx,
36    output wire o_gpio,
37    input wire i_sw0)
```

图13. 新的*i_sw0*输入信号

- b. 提供该信号作为最右侧的开关信号。如前文所述，为其余开关值分配0xFE34（位0除外）。请参见图14。

```
80
81    wire [15:0] i_sw;
82    // assign i_sw = 16'hFE34;
83    assign i_sw = {15'b111111100011010,i_sw0};
84
```

图14. 提供*i_sw0*作为最右侧的开关信号

- 修改文件[RVfpgaPath]/RVfpga/verilatorSIM/tb.cpp：这是Verilator的C++主文件。该文件的末尾有一个while循环（如图15所示），其中每次迭代都构成一个时钟脉冲。请注意，SoC的时钟信号是在该循环（第175行）内生成的，具体方法是在每次迭代中取反其二进制值（1→0或0→1）。此外，仿真时间在变量main_time（第176行）中计算，测量单位为ns（时钟周期为20 ns，因此时钟脉冲为10 ns）。最后请注意，当仿真时间达到值timeout（第171-174行）时，仿真结束。

```

136 top->clk = 1;
137 top->rst = 1;
138 while (!(done || Verilated::gotFinish())) {
139     if (main_time == 100) {
140         printf("Releasing reset\n");
141         top->rst = 0;
142     }
143     if (main_time == 200)
144         top->i_jtag_trst_n = true;
145
146     top->eval();
147     if (tftp)
148         tftp->dump(main_time);
149     if (baud_rate) do_uart(&uart_context, top->o_uart_tx);
150     if (jtag && (main_time > 300)) {
151         int ret = jtag->doJTAG(main_time/20, //doJtag requires t to only increment by one
152             &top->i_jtag_tms,
153             &top->i_jtag_tdi,
154             &top->i_jtag_tck,
155             top->o_jtag_tdo);
156         if (ret != VerilatorJtagServer::SUCCESS) {
157             if (ret == VerilatorJtagServer::CLIENT_DISCONNECTED) {
158                 printf("Ending simulation. Reason: jtag_vpi client disconnected.\n");
159                 done = true;
160             }
161             else {
162                 printf("Ending simulation. Reason: jtag_vpi error encountered.\n");
163                 done = true;
164             }
165         }
166     }
167     if (gpio0 != top->o_gpio) {
168         printf("%lu: gpio0 is %s\n", main_time, top->o_gpio ? "on" : "off");
169         gpio0 = top->o_gpio;
170     }
171     if (timeout && (main_time >= timeout)) {
172         printf("Timeout: Exiting at time %lu\n", main_time);
173         done = true;
174     }
175     top->clk = !top->clk;
176     main_time+=10;
177 }

```

图15. 用于仿真的While循环

在进入循环之前为新的i_sw0信号分配二进制值0（图16的左侧部分），然后在进入循环30 μs时将其更改为1（请参见图16的右侧部分）。

<pre> 136 top->clk = 1; 137 top->rst = 1; 138 139 top->i_sw0 = 0; 140 141 while (!(done Verilated::gotFinish())) { </pre>	<pre> 178 top->clk = !top->clk; 179 main_time+=10; 180 181 if (main_time == 30000) { 182 top->i_sw0 = 1; 183 } 184 185 } </pre>
---	--

图16. 为信号i_sw0分配值

- 完成所有这些更改后，通过执行以下命令重新编译RVfpgaSim（GSG中对此进行了说明）：

```
cd [RVfpgaPath]/RVfpga/verilatorSIM
```

```
make clean
make
```

新文件 *Vrvfpgasim*（RVfpgaSim 仿真二进制文件）应在目录 *[RVfpgaPath]/RVfpga/verilatorSIM* 内生成。

WINDOWS: 必须在 Cygwin 终端内执行最后一步（步骤4）（有关详细说明，请参见《入门指南》中的第6部分和附录C）。请注意，C: Windows 文件夹在 Cygwin 内的位置为: */cygdrive/c*。

MacOS: 有关详细说明，请参见《入门指南》的附录D。

ii. 分析程序 *LedsSwitches.S* 的仿真

在本部分中，我们将仿真《RVfpga入门指南》中的 *LedsSwitches.S* 示例程序（图17）。该程序读取开关上的值，并将读取到的值写入 Nexys A7 开发板上的 LED。请注意，我们需要配置使能寄存器，以便根据其连接将 32 个输入/输出引脚配置为输入或输出。具体来说，GPIO 的低 16 个引脚连接到 LED，这样它们便是相对于 CPU 的输出引脚（使能 = 1）。GPIO 的高 16 个引脚连接到开关，它们是相对于 CPU 的输入引脚（使能 = 0）。由于开关占用读寄存器的高 16 位，因此在将它们的值写入 LED 之前，必须先将其右移。

```
#define GPIO_SWs    0x80001400
#define GPIO_LEDs   0x80001404
#define GPIO_INOUT  0x80001408

.globl main
main:

li x28, 0xFFFF
li x29, GPIO_INOUT
sw x28, 0(x29)                # Write the Enable Register

next:
    li a1, GPIO_SWs          # Read the Switches
    lw t0, 0(a1)

    li a0, GPIO_LEDs
    srl t0, t0, 16
    sw t0, 0(a0)              # Write the LEDs

    beq zero, zero, next
.end
```

图17. 用于在 SweRVofX SoC 中运行的 *LedsSwitches.s*

请按照以下步骤运行仿真。

1. 在计算机上打开 VSCode/PlatformIO。
2. 在顶部栏上，单击“*File*”（文件）→“*Open Folder...*”（打开文件夹...）（图18），然后导航至目录 *[RVfpgaPath]/RVfpga/examples/*

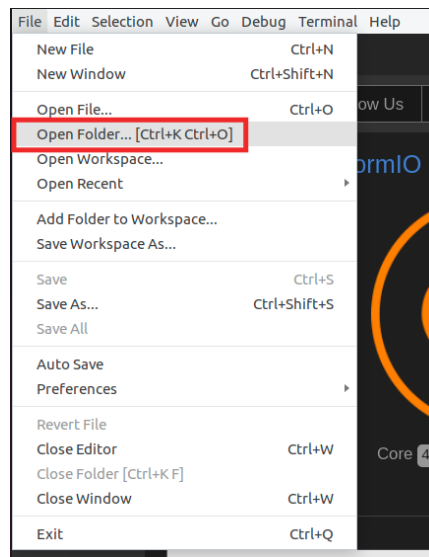


图18. 打开LedsSwitches.S示例

3. 选择目录**LedsSwitches**（不要打开，只需将其选中），然后单击“OK”（确定）。该示例随即将在PlatformIO中打开。
4. 打开文件**platformio.ini**，检查上面生成的RVfpgaSim仿真二进制文件图19的路径（上一部分的步骤3）是否正确。根据入门指南，其形式应如下所示：

```
board_debug.verilator.binary =  
[RVfpgaPath]/RVfpga/verilatorSIM/Vrvfpgasim
```

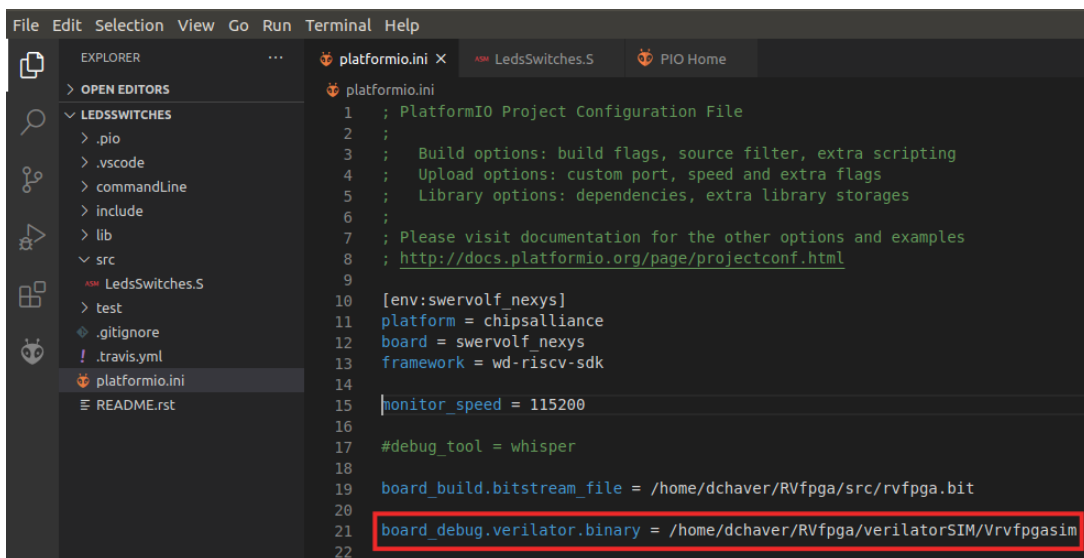


图19. Platformio初始化文件：platformio.ini

Windows: RVfpgaSim仿真可执行文件的名称为Vrvfpgasim.exe。因此：

```
board_debug.verilator.binary = [RVfpgaPath]\RVfpga\verilatorSIM\Vrvfpgasim.exe
```

- 单击左侧功能区菜单中的PlatformIO图标运行仿真，然后展开“Project Tasks”（项目任务）→ `env:swervolf_nexys` → “Platform”（平台），并单击“Generate Trace”（生成跟踪）。

文件`trace.vcd`应当已在

`[RVfpgaPath]/RVfpga/examples/LedsSwitches/.pio/build/swervolf_nexys`内部生成，随后通过在PlatformIO终端中输入以下命令，可使用`GTKWave`将其打开。

```
gtkwave [RVfpgaPath]/RVfpga/examples/LedsSwitches/.pio/build/swervolf_nexys/trace.vcd
```

WINDOWS: 已下载的文件夹`gtkwave64`包括一个称为`gtkwave.exe`的应用程序，该应用程序位于`bin`文件夹内。双击该应用程序启动GTKWave。在应用程序的顶部，单击“**File**”（文件）– “**Open New Tab**”（打开新选项卡），然后打开在文件夹`[RVfpgaPath]/RVfpga/examples/LedsSwitches/.pio/build/swervolf_nexys`中生成的`trace.vcd`文件。

- 在轨迹中包含以下信号（转到模块`rvfpgasim - swervolf`，找到下面各个信号）：
 - 添加时钟信号：`clk`
 - 添加GPIO输入信号：`i_gpio`
 - 添加GPIO输出信号：`o_gpio`

在图（图20）中，您将看到16个开关的值（信号`i_gpio`的高16位）已复制到16个LED（信号`o_gpio`的低16位）中，但存在一些延时。此外，最右侧的开关将在30 μ s时发生更改（0→1），这会使最右侧的LED在一段时间后也发生更改。

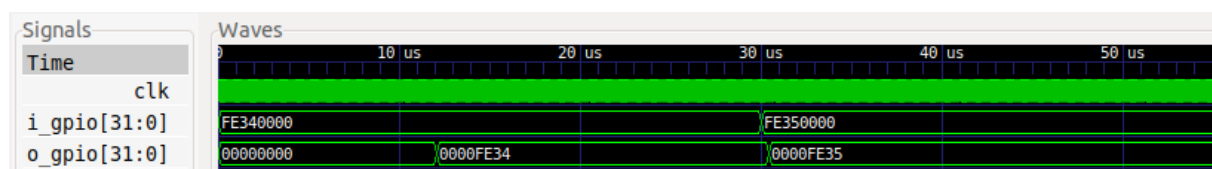


图20. LedsSwitches程序的仿真

7. 高级练习

练习2. 更详细地分析上一部分中的仿真。图21 给出了`.elf LedsSwitches`程序（图17）的反汇编版本，其中突出显示了用于访问GPIO寄存器的三条指令（使能、读和写）。根据本《入门指南》，您可以通过在PlatformIO中打开文件`firmware.dis`轻松查看`.elf`程序的反汇编版本，该文件于编译时在以下文件夹内生成：

`[RVfpgaPath]/RVfpga/examples/LedsSwitches/.pio/build/swervolf-nexys/`（见图21）。

```

> PSP
> src
  ≡ .sconsign36.dblite
  ≡ firmware.bin
  ≡ firmware.dis
  ≡ firmware.elf
  ≡ LedsSwitches.map
  ≡ libBoardBSP.a
  ≡ libPSP.a
  ≡ project.checksum
> libdeps
> .vscode
> commandLine
> include
> lib
  ≡ src

```

```

65 Disassembly of section .text:
66
67 00000090 <main>:
68 90: 00010e37      lui t3,0x10
69 94: fffe0e13      addi t3,t3,-1 # ffff <_sp+0xcebf>
70 98: 80001eb7      lui t4,0x80001
71 9c: 408e8e93      addi t4,t4,1032 # 80001408 <OVERLAY_END_OF_OVERLAYS+0xa0001408>
72 a0: 01cea023      sw t3,0(t4)
73
74 000000a4 <next>:
75 a4: 800015b7      lui a1,0x80001
76 a8: 40058593      addi a1,a1,1024 # 80001400 <OVERLAY_END_OF_OVERLAYS+0xa0001400>
77 ac: 0005a283      lw t0,0(a1)
78 b0: 80001537      lui a0,0x80001
79 b4: 40450513      addi a0,a0,1028 # 80001404 <OVERLAY_END_OF_OVERLAYS+0xa0001404>
80 b8: 0102d293      srli t0,t0,0x10
81 bc: 00552023      sw t0,0(a0)
82 c0: fe0002e3      beqz zero,a4 <next>

```

图21. LedsSwitches.S程序的反汇编版本

在RVfpgaSim中仿真该程序，并在执行图21中以红色突出显示的三条访存指令（sw、lw和sw）期间分析GPIO信号。这将帮助您了解A部分中说明的GPIO底层实现。

您可以从B部分的仿真开始，添加并分析以下信号的值（转到参考模块找到每个信号）：

- rvfpgasim → swervolf → swerv_eh1 → swerv → ifu
 - 时钟： **clk**。
 - 已取指令： **ifu_i0_instr**和**ifu_i1_instr**。
- rvfpgasim – swervolf
 - 32位输入/输出引脚： **i_gpio**和**o_gpio**。
 - CPU提供的地址： **wb_m2s_io_adr**。
- rvfpgasim – swervolf – gpio_module
 - GPIO外部接口： **ext_pad_i**、**ext_pad_o**和**ext_padoe_o**。
- rvfpgasim – swervolf – wb_intercon0
 - 图2的多路开关的输出地址和数据信号： **wb_io_adr_i**、**wb_io_dat_i**和**wb_io_dat_o**。
 - 图2的多路开关的输入GPIO数据信号： **wb_gpio_adr_i**、**wb_gpio_dat_i**和**wb_gpio_dat_o**。
 - 图2的多路开关的选择信号： **wb_*_cyc_o**。
- rvfpgasim – swervolf – wb_intercon0 – wb_mux_io
 - 图2的多路开关的匹配信号： **match**。
- rvfpgasim – swervolf – swerv_eh1 – swerv – dec – arf – gpr_banks(0) – gpr(5) – gprff
 - t0的寄存器值： **dout**。

练习3. 扩展RVfpgaNexys以支持五个板上按钮。按钮如图22所示。这五个按钮根据其位置命名：上、下、左、右和中 – BTNU、BTND、BTNL、BTNR、BTNC。

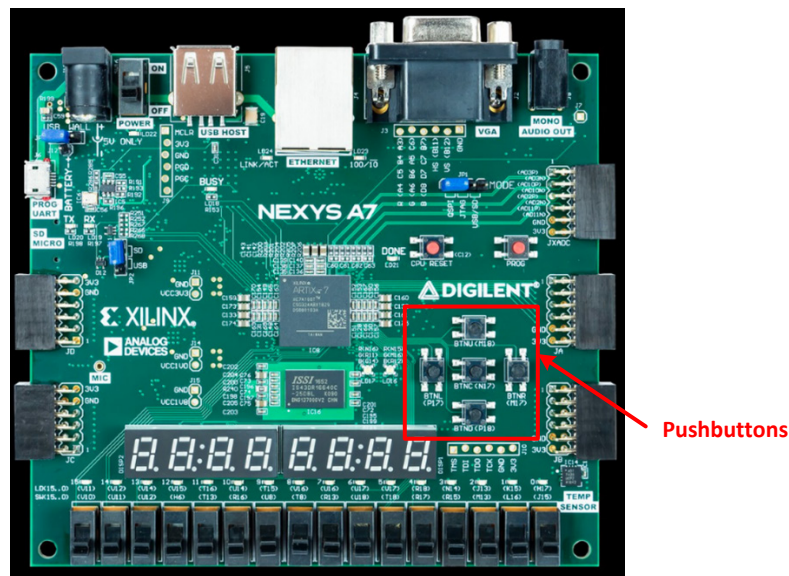


图22. Nexys A7 FPGA开发板上的按钮

- 假设我们正在使用的GPIO模块（`gpio_top`）的最大大小为32，即I/O引脚数为32（16个LED + 16个开关），则需要在SweRVolfX中包含GPIO模块的另一个实例，以及5个新的三态缓冲器和所有必要的信号。
- 使用从0x80001800开始的地址（可用）来映射新GPIO控制器对应的寄存器。请注意，必须修改多路开关（图9）以包含新外设。
- 在修改约束文件时，还必须考虑到5个按钮连接到以下FPGA引脚：
 - BTNC连接到引脚N17
 - BTNU连接到引脚M18
 - BTNL连接到引脚P17
 - BTNR连接到引脚M17
 - BTND连接到引脚P18

练习4. 在RVfpgaNexys中为5个板上按钮设计另一个控制器。

- 与练习3相比，在此示例中，必须根据图3中说明的方案在Verilog或SystemVerilog中实现自己的GPIO控制器。实际上，甚至可以简化相应电路，只包含一个读寄存器（即，无需包含三态缓冲器或写寄存器）。
- 无需从上一个练习中删除控制器，因为按钮可以映射到该GPIO控制器未使用的地址。
- 将新控制器包含在系统控制器外设中。可以使用未使用的地址范围0x8000101C-0x8000101F。请注意，系统控制器中包含的寄存器基于CPU生成的地址（`i_wb_adr`）直接连接到Wishbone总线（`o_wb_rdt`）的数据信号，以这种方式读入到CPU中。检查模块`swervolf_syscon`（`[RVfpgaPath]/RVfpga/src/SweRVolfSoC/Peripherals/SystemController/swervolf_syscon.v`）的第234-266行，以帮助了解如何继续操作。

练习5. 编写一个RISC-V汇编程序和一个C程序，以便从1开始在LED上显示递增的二进制计数。程序中包含一个用于在各个递增值的显示之间留出等待时间的空循环，以便人眼可以看到这些值。通过练习3中实现的OpenCores外设读取BTNC并使用它来更改计数速度，然后通过练习4中实现的专用外设读取BTNU并在每次按下时使用它重启计数。