THE IMAGINATION UNIVERSITY PROGRAMME

RVfpga Lab 6
Introduction to I/O

1. INTRODUCTION

In Labs 6-10, you will learn how to use and expand RVfpga's Input/Output (I/O) system to enable the RISC-V processor to interact with peripheral devices. Below is an overview of the topics covered in these labs:

- **Lab 6:** Learn how to use the general-purpose input/output (GPIO) pins connected to the LEDs, switches, and pushbuttons on the Nexys A7 board
- **Lab 7:** Learn how to use the 7-segment displays available on the board
- **Lab 8:** Learn how to use timers
- **Lab 9:** Learn how to use interrupts to interface with external devices
- **Lab 10:** Learn how to interface the RVfpga System with the onboard SPI accelerometer

In this lab, we first describe the main features of a general-purpose I/O system and the one used in the RVfpga System (Section 2). We then describe a simplified theoretical version of a generic GPIO controller (Section 3). Finally, we focus on the GPIO controller used in the SweRVolfX SoC: we first analyse its high-level specification and introduce fundamental exercises (Sections 4 and 5). We conclude the lab by analysing its low-level implementation, simulating RVfpgaSim in Verilator, and introducing advanced exercises (Sections 6 and 7).

We use this same general structure in Labs 7-10. In the beginning sections, we describe the I/O controller's high-level specification (its main features, registers and their operation, and the memory map) and then introduce fundamental exercises for practice using the peripheral. In the advanced sections, we describe the controller's low-level implementation and provide exercises for modifying it and then writing programs that test the modification.

Note to instructors: you may choose the complexity of exercises according to your course level. For example, in a first/second year course (such as Computer Fundamentals or Computer Organization), the fundamental exercises – in this lab, Section 5 – would be suitable. However, in a more advanced course (such as Computer Architecture or Embedded System Design), both the fundamental and advanced exercises – in this lab, sections 5 to 7 – could be used.

2. INPUT/OUTPUT ARCHITECTURE

Figure 1 illustrates the structure of the Von Neumann Architecture, which is composed of three main blocks: the CPU, the Memory, and the I/O System. In Labs 6-10, we focus on the CPU's interaction with input/output (I/O) devices. I/O devices are also referred to as peripherals or simply devices. We overview the role of each main unit here:

- **CPU**: the CPU is the initiator of all I/O operations. It is the *controller* (historically called "master", but that term is deprecated) of any I/O transaction. A direct-memory-access (DMA) controller (DMAC) could also act as a controller, but it is not included in this lab.
- **Device Controller:** The *device controller* waits for read/write requests from a *controller* to perform any action. Device controllers behave as *peripherals* (formerly called "slaves," but that term is deprecated) in the I/O system. Conceptually, a device controller consists of a series of *registers* that are accessible from the *controller*. The values of these registers instruct the *peripheral* about what action to perform.
- **The interconnect** (bus, crossbar, etc.) establishes a path between the *controller* and the *peripherals.* Interconnect is usually implemented with several layers connected

through a *bridge* that prevents certain devices from slowing down the entire system.

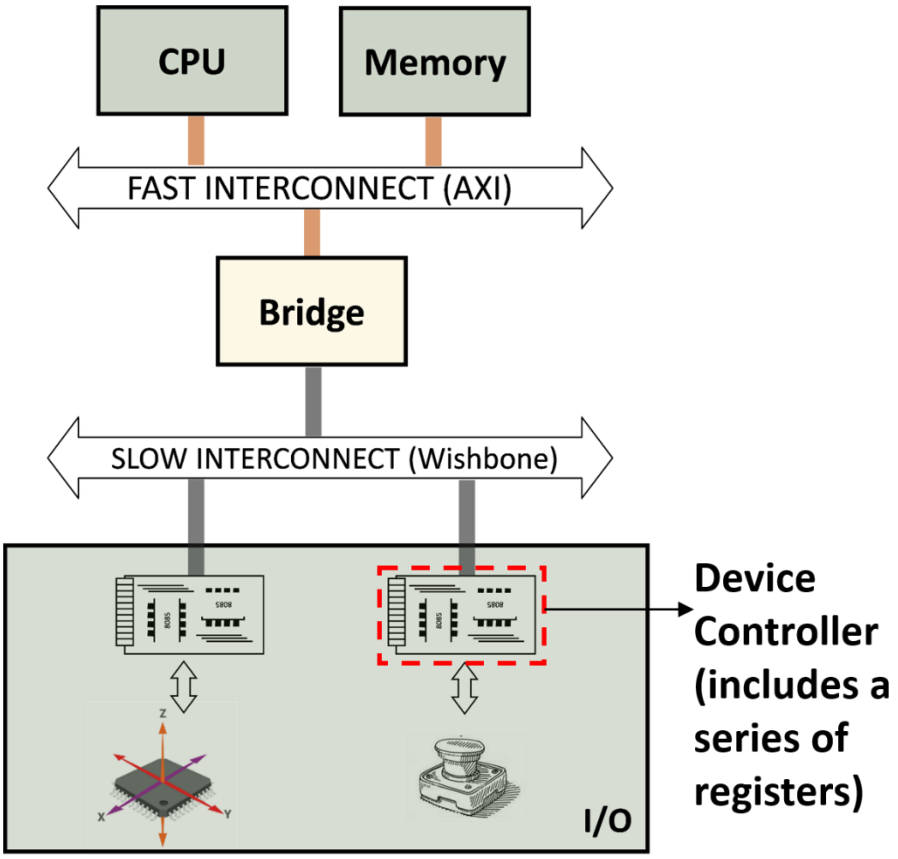

Figure 1 Generic Computing System

Figure 2 shows RVfpga's I/O system. It includes the following seven peripherals:

- LEDs and Switches (considered a single peripheral), connected to the GPIO1 module
- 7-segment displays, connected to the System Controller module
- Flash Memory, connected to the SPI1 module
- Accelerometer, connected to the SPI2 module
- Timer
- UART
- Boot ROM

A multiplexer selects one peripheral among the seven possibilities and connects it with the CPU. Note that a Wishbone to AXI Bridge is necessary because the peripherals use a Wishbone bus (grey colour) whereas the SweRV EH1 Core uses an AXI bridge (orange colour).

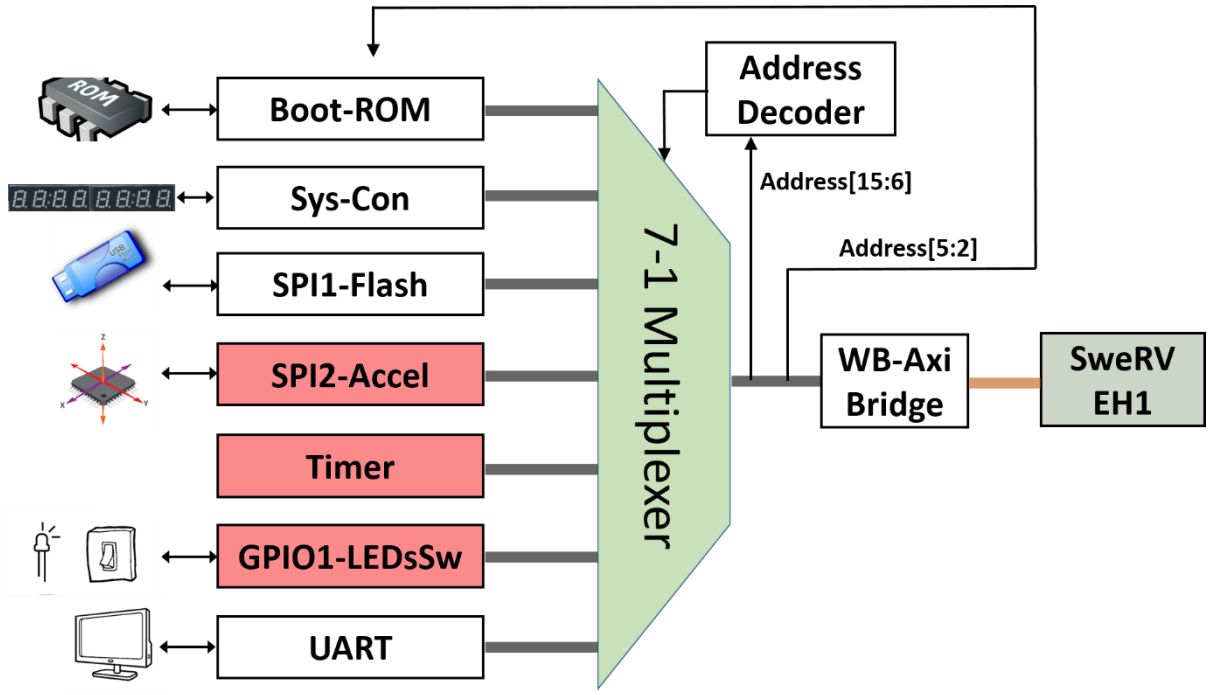

Figure 2. I/O System in the RVfpga System

TASK: Locate each of the elements of Figure 2 in the SoC. You will need to inspect the following files and directories:

[RVfpgaPath]/RVfpga/src/SweRVolfSoC/swervolf_core.v (main file, where the elements from Figure 2 are instantiated).

[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

As described in the RVfpga Getting Started Guide, the original SweRVolf (https://github.com/chipsalliance/Cores-SweRVolf) includes only some of the peripherals shown in Figure 2: specifically, the Boot ROM, System Controller (with no 7-Segment Displays), SPI Flash Memory and UART (shown in white in Figure 2). Remember from the GSG that SweRVolfX SoC extends the original SweRVolf SoC with new peripherals: an SPI Accelerometer, a Timer, a GPIO module (shown in red in Figure 2), and a 7-segment display controller (that extends SweRVolf's existing System Controller).

Each peripheral receives values from the processor and/or sends values back to the processor. Memory addresses are reserved for I/O values and are called *registers*, *memory-mapped I/O registers*, or *device controller registers*. To send a value to a peripheral, the CPU stores a value to a specified memory address (i.e., memory-mapped register). To read a value from a peripheral, the CPU loads a value from a specified memory address. Thus, a simple *load/store* operation from the CPU may configure a device, check its status, or read/write data from/onto it.

The multiplexer in Figure 2 selects the requested device controller using *Address[15:6]*. The device controllers use *Address[5:2]* to select among several registers used to control the device.

3. GENERAL PURPOSE INPUT/OUTPUT (GPIO)

A general-purpose I/O (GPIO) controller exposes external digital pins to the programmer. At any given time in the program, those pins can be configured as either inputs or outputs. That designation is per pin and can change throughout the program, if desired. GPIO pins can be connected to external devices such as LEDs, switches, and pushbuttons.

Figure 3 illustrates a simplified diagram for a generic GPIO module connecting one external pin to the CPU. The pin can be connected to any input/output device, such as an LED, a switch, etc. The pin is connected to a tri-state buffer, highlighted in green in the figure. This buffer allows the programmer to configure the pin as either an input or output. If the tri-state buffer is enabled, the pin acts as an output (for example, for driving an LED). If the tri-state buffer is disabled, the pin acts as an input (for example, for reading from switch values).

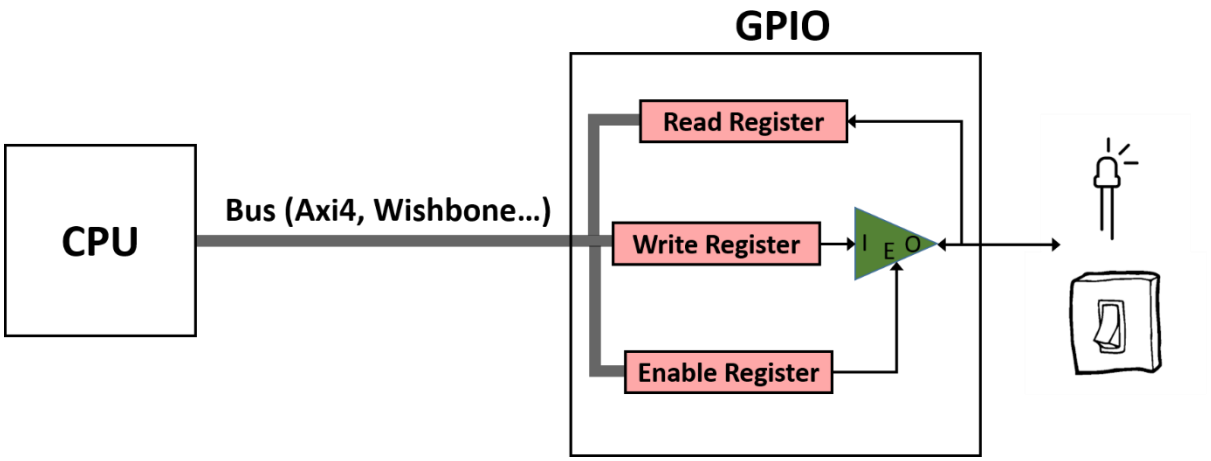

Figure 3. GPIO simplified circuit

A tri-state buffer can either act as a regular buffer (when it is enabled) or have a floating output (when it is disabled). The tri-state buffer has two inputs, E (enable) and I (input), and one output, O, and its truth table is shown in Table 1. When E is 1, the tri-state acts as a regular buffer with the output (O) and input (I) being the same. When E is 0, no connection exists between the input and output and the output (O) is not driven; O is floating. In Figure 3, to configure a pin as an output, E is 1, which allows the CPU to drive the pin. When a pin is configured as an input, E is 0, which keeps the CPU from driving the pin and allows the peripheral to drive it.

Table 1. Tri-state truth table

E	I	O
0	0	Hi-Z
0	1	Hi-Z
1	0	0
1	1	1

The RVfpga System uses memory-mapped I/O to read/write the values stored in these registers. For example, assume that the pin from Figure 3 is connected to a switch and that the three registers in the GPIO are mapped as follows:

- Read Register = Address 0x80001400
- Write Register = Address 0x80001404
- Enable Register = Address 0x80001408

To read the state of the switch, we do the following:

1. Configure the pin as an input by writing a 0 to the Enable Register (i.e., by executing a *store* of 0 to address 0x80001408).
2. Read the Read Register by executing a *load* instruction to address 0x80001400.

4. GPIO HIGH-LEVEL SPECIFICATION

In this section, we first analyse the high-level specification of SweRVolfX's GPIO and then we propose one exercise that uses this peripheral.

A. GPIO high-level specification

The GPIO module used in SweRVolfX is from OpenCores (https://opencores.org/projects/gpio). The gpio_spec.pdf document provided with the OpenCore's GPIO module download describes the module's high-level specification. It is available here: *[RVfpgaPath]/RVfpga/src/SweRVolfSoC/Peripherals/gpio/docs/gpio_spec.pdf*. We summarize the main operation and features of the GPIO module in this lab. However, you can obtain the complete specifications in *gpio_spec.pdf*.

The GPIO module has the following main features:

- It uses a Wishbone Interconnection.
- It operates as a peripheral device only.
- The user may use 1-32 GPIO pins.
- Multiple GPIO modules (also called GPIO cores) can be used in parallel to access more than 32 GPIO pins.
- All GPIO pins can be:
    - bi-directional (external bi-directional I/O cells are required in this case).
    - tri-state or open-drain enabled (external tri-state or open-drain I/O cells are required in this case).
- GPIO pins that are programmed as inputs:
    - can be registered.
    - can cause an interrupt request to the CPU.

Section 4 of the GPIO core specification describes the control and status registers available inside the GPIO module. Each of these registers is assigned to a different address as shown in Table 2. The base address for the GPIO registers is **0x80001400**.

Table 2. GPIO Registers

Name	Address	Width	Access	Description
RGPIO_IN	0x80001400	1-32	R	GPIO input data
RGPIO_OUT	0x80001404	1-32	R/W	GPIO output data
RGPIO_OE	0x80001408	1-32	R/W	GPIO output driver enable
RGPIO_INTE	0x8000140C	1-32	R/W	Interrupt enable
RGPIO_PTRIG	0x80001410	1-32	R/W	Type of event that triggers an interrupt
RGPIO_AUX	0x80001414	1-32	R/W	Multiplex auxiliary inputs to GPIO outputs
RGPIO_CTRL	0x80001418	2	R/W	Control register

RGPIO_INTS	0x8000141C	1-32	R/W	Interrupt status
RGPIO_ECLK	0x80001420	1-32	R/W	Enable gpio_eclk to latch RGPIO_IN
RGPIO_NEC	0x80001424	1-32	R/W	Select active edge of gpio_eclk

Although the OpenCore's GPIO module is more complex than the simplified version illustrated in Figure 3, we can still identify the three registers from Figure 3: Read (input), Write (output), and Enable. In the OpenCore's GPIO module, these registers are called, respectively: RGPIO_IN, RGPIO_OUT and RGPIO_OE and are mapped to addresses 0x80001400, 0x80001404, and 0x80001408 respectively.

TASK: Locate the declaration of registers RGPIO_IN, RGPIO_OUT and RGPIO_OE in the GPIO module, as well as the definition of their addresses. The GPIO module is here: *[RVfpgaPath]/RVfpga/src/SweRVolfSoC/Peripherals/gpio/gpio_top.v*.

The RGPIO_IN register latches general-purpose inputs. The RGPIO_OUT register drives general-purpose outputs. RGPIO_OE configures each I/O pin as an input or output. When the enable bit (within RGPIO_OE) is set, the corresponding general-purpose output driver is enabled, and thus the pin can be connected to an output peripheral, such as an LED. When the enable bit is cleared, the output driver is operating in open-drain, also called tri-state or high impedance, mode, and thus the pin can be connected to an input peripheral, such as a switch or pushbutton.

In RVfpgaNexys, the first 16 GPIO pins, pins 15:0, of the GPIO module are connected to the 16 LEDs on the Nexys A7 board. The last 16 GPIO pins, pins 31:16, of the GPIO controller are connected to the 16 on-board switches.

5. FUNDAMENTAL EXERCISES

Exercise 1. Write a RISC-V assembly program and a C program that shows a block of four lit LEDs that repeatedly moves from one side of the 16 LEDs available on the board to the other. Also include two switches that control the speed and direction. Switch[0] changes the speed and Switch[1] changes the direction as follows:

- If Switch[0] is ON (high), the lit LEDs should move quickly. Otherwise, the lit LEDs should move slowly. You may define what "quickly" and "slowly" mean, but either speed must be visible, and you must be able to detect a difference in speed just by looking at it.
- If Switch[1] is ON (high), the lit LEDs should repeatedly move from right-to-left (they start back at the right when they reach the left-most LED). Otherwise, the lit LEDs should repeatedly move from left-to-right.

Figure 4 below shows the Nexys A7 board with the LEDs and switches highlighted.

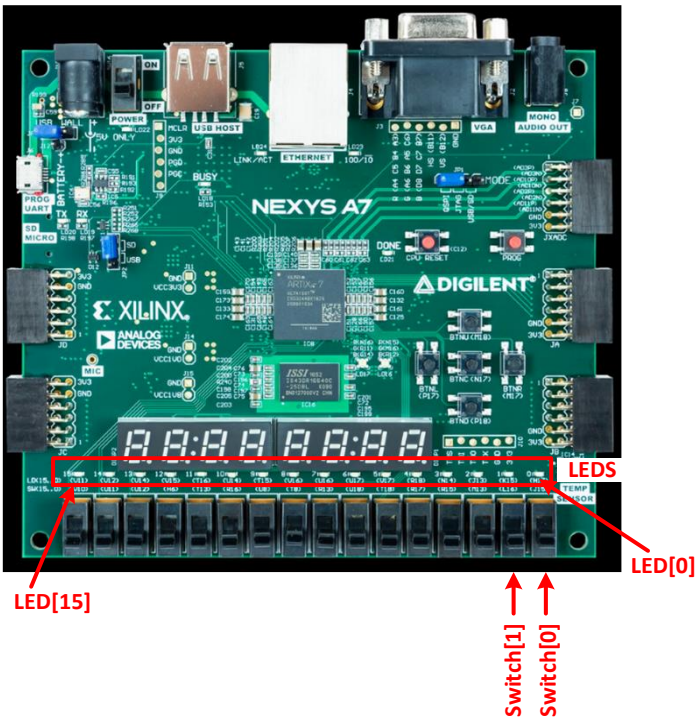

Figure 4. Nexys A7 FPGA Board: LEDs and Switches

Hint: Recall that the switches are connected to pins 31:16 of the memory-mapped I/O registers. So, to read Switch[0], you would need to write 0 to RGPIO_OE[16] and then read the value of RGPIO_IN[16]. You will need to configure RGPIO_OE appropriately to access the other LEDs and switches.

6. GPIO LOW-LEVEL IMPLEMENTATION, SIMULATION

In this section, we describe the low-level details of the GPIO used in SweRVolfX. We then modify RVfpgaSim and perform an example simulation in Verilator for a simple assembly example. Finally, we propose some exercises where you will first simulate RVfpgaSim, then modify it to add a new GPIO peripheral and finally write a program that uses this new peripheral.

A. GPIO low-level implementation

Now that you have had some experience with accessing the GPIO pins using memory-mapped I/O, let's dive into the low-level details of the GPIO. The GPIO can be divided into three main parts, as shown in Figure 5: (1) RVfpgaNexys' external connection to the on-board LEDs/Switches (left shaded region in Figure 5); (2) Integration of the GPIO module into the SweRVolfX SoC (middle shaded region in Figure 5); (3) Connection between the GPIO and the SweRV EH1 Core (right shaded region in Figure 5).

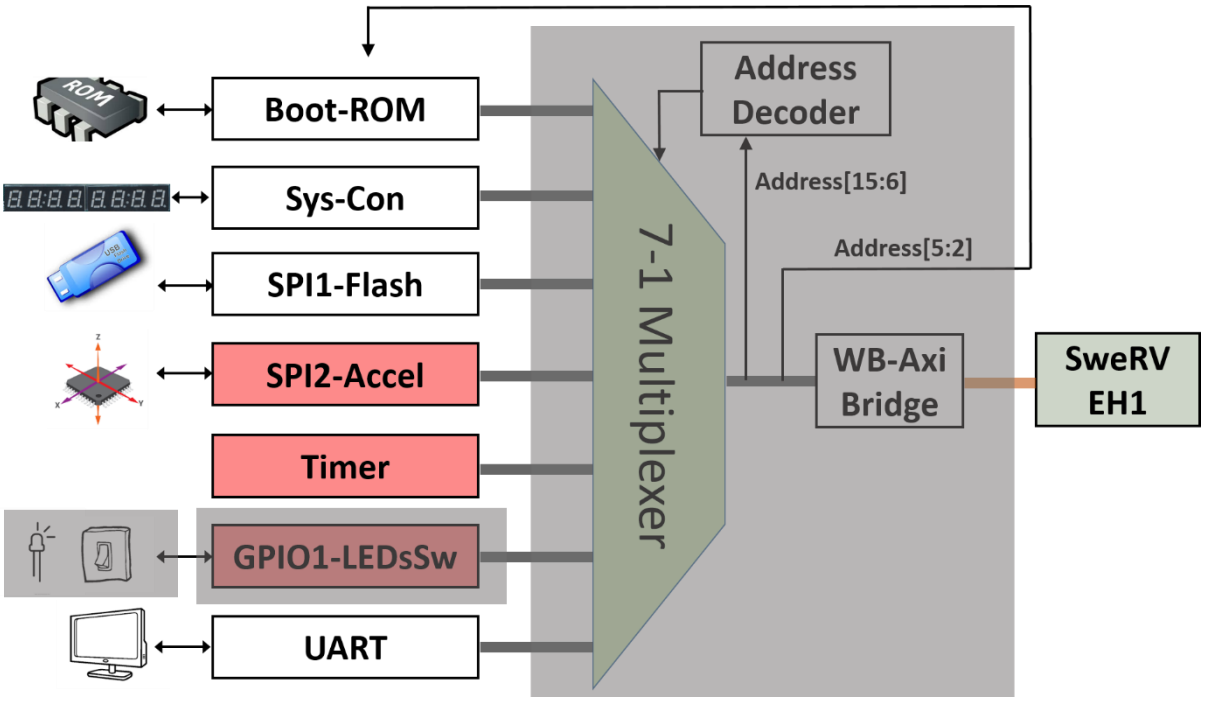

Figure 5. GPIO analysis in 3 phases

i. Connection of the LEDs/Switches with the SoC

The constraints file of the project (*[RVfpgaPath]/RVfpga/src/rvfpganexys.xdc*) defines the connection between the input/output SoC signals and the board devices. Each board device is associated with a given FPGA pin. For example, Switch[0], the right-most switch on the board, is connected through a printed circuit board (PCB) trace to FPGA pin J15.

The Nexys A7 board includes 16 LEDs and 16 Switches. The signal that connects the 16 LEDs with the top-module of the SoC (called rvfpganexys, available inside file *[RVfpgaPath]/RVfpga/src/rvfpganexys.sv*) is called *o_led[15:0]*, and the signal that connects the 16 Switches with top-module is called *i_sw[15:0]*. Figure 6 shows the section of the Xilinx design constraint (xdc) file, rvfpganexys.xdc (available in *[RVfpgaPath]/RVfpga/src*) where these 32 connections between the signal and FPGA pin are defined.

```
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] }]
```

Figure 6. Connection of i_sw[15:0] with the on-board switches and o_led[15:0] with the on-board LEDs (file *rvfpganexys.xdc*).

Lines 48-49 of the top-module (**rvfpganexys**) show these two signals connected to the SoC (left part of Figure 7), and the end of that module shows their connection with the **swervolf_core** module (right part of Figure 7). Note that the *i_sw* and *o_led* signals are merged in signal *io_data* (line 257), a 32-bit input/output signal connected with the GPIO in the **swervolf_core** module (as will be shown later, in Figure 8). Moreover, note that the *o_led* signal is latched through an intermediate signal, *gpio_out* (line 266).

```
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
```

Figure 7. Connection of the LEDs and the Switches with the top-module (*rvfpganexys.sv*)

TASKS: Follow these two signals (*i_sw* and *o_led*) from the constraints file to the SweRVolf SoC module (where they are merged in *io_data*). You will need to inspect the following files:

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

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

In the previous section we said that in RVfpgaNexys the 16 first GPIO pins (15 to 0) of the GPIO module are connected to the 16 on-board LEDs, whereas the 16 last GPIO pins (31 to 16) of the GPIO controller are connected with the 16 on-board switches. Does this correspond with the implementation described in this section and in Figure 8?

ii. Integration of the GPIO module in the SoC

In lines 299-354 of the **swervolf_core** module (*[RVfpgaPath]/RVfpga/src/SweRVolfSoC/swervolf_core.v*), the GPIO module is instantiated and integrated into the SoC (see Figure 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));
```

Figure 8. Integration of the GPIO module (file *swervolf_core.v*).

The interface of the module can be divided into two blocks: Wishbone signals (Table 3), which allow the SweRV EH1 Core to communicate with the GPIO using a controller/peripheral model, and external I/O signals (Table 4).

Table 3. Wishbone Signals

Port	Width	Direction	Description
wb_cyc_i	1	Inputs	Indicates valid bus cycle (core select)
wb_adr_i	15	Inputs	Address inputs
wb_dat_i	32	Inputs	Data inputs
wb_dat_o	32	Outputs	Data outputs
wb_sel_i	4	Inputs	Indicates valid bytes on data bus (during valid cycle it must be 0xf)
wb_ack_o	1	Output	Acknowledgment output (indicates normal transaction termination)
wb_err_o	1	Output	Error acknowledgment output (indicates an abnormal transaction termination)
wb_rty_o	1	Output	Not used
wb_we_i	1	Input	Write transaction when asserted high
wb_stb_i	1	Input	Indicates valid data transfer cycle
wb_inta_o	1	Output	Interrupt output

Table 4. External I/O Signals

Port	Width	Direction	Description
in_pad_i	1-32	Inputs	GPIO inputs
out_pad_o	1-32	Outputs	GPIO outputs
oen_padoen_o	1-32	Outputs	GPIO output drivers enables (for three-state or open-drain drivers)

As shown in line 342 of Figure 8, bits 5:2 of the address provided by the core in the Wishbone bus signal *wb_m2s_gpio_adr[5:2]* are used for selecting one among the 10 available memory-mapped registers. These four bits are provided to the GPIO Core through the *wb_adr_i* signal (also shown in Figure 8).

Input *ext_pad_i* connects directly with the GPIO Read Register (RGPIO_IN). Similarly, output *ext_pad_o* connects directly with the GPIO Write Register (RGPIO_OUT). These two signals are connected to the LEDs and Switches (*i_gpio*, *o_gpio*, *io_data*) through 32 tri-state buffer modules (Figure 8, lines 305-336). That way, all 32 pins can be configured as inputs or outputs. In our case, the lower 16 pins, pins 15:0, are connected to the LEDs (Figure 7) and thus they must be configured as outputs; the upper 16 pins, 31:16, are connected to the switches (Figure 7) and thus they must be configured as inputs. We implement these 32 tristate buffers by including the following module at the end of the **swervolf_core** module (lines 634-640):

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

TASKS: The GPIO pins (*io_data*) are connected to the GPIO module through tri-state buffers (see Figure 8). Analyse the tri-state buffer for the two possible states of the enable signal (*oe*=0 and *oe*=1).

Taking into account the connection between the GPIO module and the on-board LEDs/Switches, what values should the programmer assign to *en_gpio*?

iii. Connection between the GPIO and the SweRV EH1 Core

As shown in Figure 2, the device controllers are connected to the SweRV EH1 Core through a multiplexer and a bridge. The multiplexer selects one among the N possible peripherals (in our case, N=7), depending on the address generated by the CPU. The bridge translates the Wishbone signals used by the device controllers to the AXI4 signals used by the SweRV Core and vice versa (implemented in file *[RVfpgaPath]/RVfpga/src/SweRVolfSoC/Interconnect/AxiToWb/axi2wb.v*).

The 7:1 multiplexer (Figure 9) is implemented in file *[RVfpgaPath]/RVfpga/src/SweRVolfSoC/Interconnect/WishboneInterconnect/wb_intercon.v*, which is instantiated in lines 104-205 of file *[RVfpgaPath]/RVfpga/src/SweRVolfSoC/Interconnect/WishboneInterconnect/wb_intercon.vh* . This latter file is included in line 168 of the **swervolf_core** module located here: *[RVfpgaPath]/RVfpga/src/SweRVolfSoC/swervolf_core.v*.

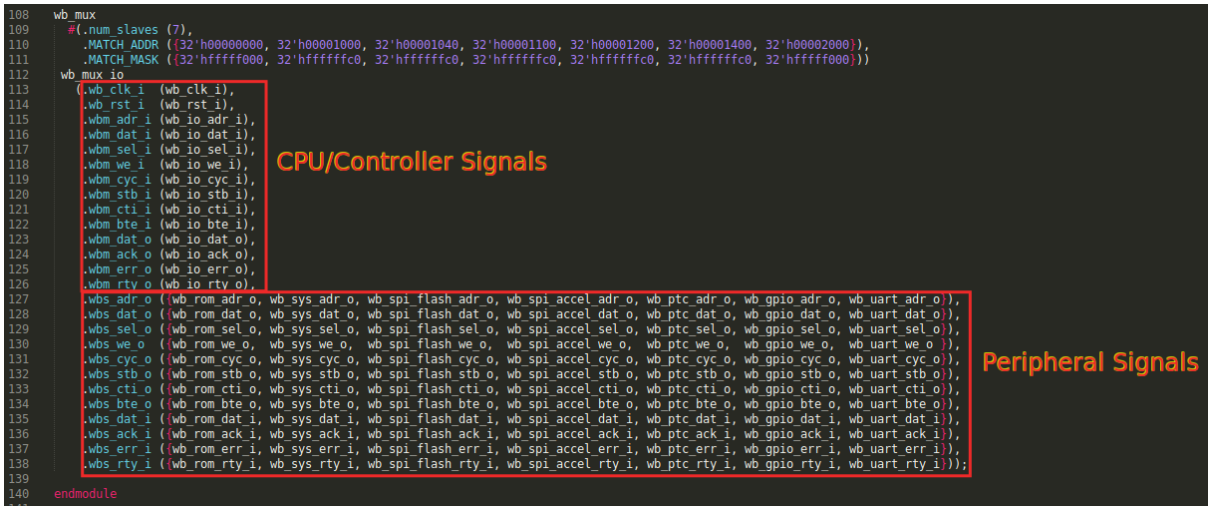

Figure 9. 7-1 multiplexer selects the peripheral to connect to the CPU (*wb_intercon.v*).

The multiplexer selects which peripheral to read or write, connecting the CPU (*wb_io_** signals – lines 115-126 of Figure 9) with the Wishbone Bus of one peripheral (lines 127-138 of Figure 9), depending on the address (lines 110-111). For example, if the address generated by the CPU is in the range 0x80001400-0x8000143F, the GPIO peripheral is selected, and thus signals *wb_io_** will be connected with signals *wb_gpio_**.

Figure 10 shows the Verilog implementation of the multiplexer (available in file *[RVfpgaPath]/RVfpga/src/SweRVolfSoC/Interconnect/WishboneInterconnect/wb_intercon_1.2.2-r1/wb_mux.v*).

TASK: Analyse in detail the implementation of the multiplexer. You can focus on the GPIO-related signals (*wb_gpio_**). You will need to inspect the following files:

[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

Understanding this part of the SoC is important not only for this lab but also for future labs. The simulation performed in the next section can help you in understanding it if you extend

the simulation by adding new signals related with the multiplexer.

```
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] ff1;
105       input [num_slaves-1:0] in;
106       integer          i;
107
108       begin
109    ff1 = 0;
110    for (i = num_slaves-1; i >= 0; i=i-1) begin
111       if (in[i])
112 /* verilator lint_off WIDTH */
113          ff1 = i;
114 /* verilator lint_on WIDTH */
115    end
116       end
117    endfunction
118
119    assign slave_sel = ff1(match);
120
121    always @(posedge wb_clk_i)
122      wbm_err <= wbm_cyc_i & !(|match);
123
124    assign wbs_adr_o = {num_slaves{wbm_adr_i}};
125    assign wbs_dat_o = {num_slaves{wbm_dat_i}};
126    assign wbs_sel_o = {num_slaves{wbm_sel_i}};
127    assign wbs_we_o  = {num_slaves{wbm_we_i}};
128 /* verilator lint_off WIDTH */
129
130    assign wbs_cyc_o = match & (wbm_cyc_i << slave_sel);
131 /* verilator lint_on WIDTH */
132    assign wbs_stb_o = {num_slaves{wbm_stb_i}};
133
134    assign wbs_cti_o = {num_slaves{wbm_cti_i}};
135    assign wbs_bte_o = {num_slaves{wbm_bte_i}};
136
137    assign wbm_dat_o = wbs_dat_i[slave_sel*dw+:dw];
138    assign wbm_ack_o = wbs_ack_i[slave_sel];
139    assign wbm_err_o = wbs_err_i[slave_sel] | wbm_err;
140    assign wbm_rty_o = wbs_rty_i[slave_sel];
141
142 endmodule
```

Figure 10. Wishbone multiplexer (file *wb_mux.v*).

B. Verilator Simulation

In this section, we first modify RVfpgaSim simulator by adding a new input signal. We then recompile RVfpgaSim using Verilator and analyse this new signal when the simulator executes a simple program.

i. Modify and Recompile RVfpgaSim

In simulation, we do not have real LEDs or switches. Thus, in the testbench (*[RVfpgaPath]/RVfpga/src/rvfpgasim.v*), we simulate driving the switches by assigning that signal (`i_sw`) a constant value of 0xFE34 (left part of Figure 11). The switches are then provided as an input to the SweRVolfX SoC (right part of Figure 11).

```
80      wire [15:0]  i_sw;
81      assign  i_sw = 16'hFE34;
```

```
248         .io_data            ({i_sw,16'bz}));
```

Figure 11. Signal `i_sw` assigned and passed to the SweRVolfX SoC in rvfpgasim.v.

Remember from the Getting Started Guide that the testbench (*rvfpgasim.v*) receives the input signals (*clk*, *rst*, etc.) for RVfpgaSim (left part of Figure 12) and instantiates the **swervolf_core** module (right part of Figure 12).

```
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),
```

Figure 12. Input signals for RVfpgaSim and SweRVolfX instantiation (file *rvfpgasim.v*).

In some situations, you may want to add a new input/output signal to the simulator. As an example, we next explain how you can include an input signal to RVfpgaSim, called `i_sw0`, which provides a value for the right-most switch.

Follow the next steps:

1. Modify file *[RVfpgaPath]/RVfpga/src/rvfpgasim.v*:

    a. Include a new 1-bit input signal called `i_sw0`. See Figure 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)
```

Figure 13. New `i_sw0` input signal.

    b. Provide this signal as the right-most switch. Assign the remaining switch values to be 0xFE34 – except for bit 0 – as before). See Figure 14.

```
80
81      wire [15:0]  i_sw;
82    //  assign  i_sw = 16'hFE34;
83      assign i_sw = {15'b1111111100011010,i_sw0};
84
```

Figure 14. Provide `i_sw0` as the right-most switch.

2. Modify file *[RVfpgaPath]/RVfpga/verilatorSIM/tb.cpp*: this is the C++ main file for Verilator. At the end of this file, you can find a `while` loop (shown in Figure 15) in which each iteration constitutes a clock pulse. Note that the clock signal for the SoC is generated within this loop (line 175), by inverting its binary value in each iteration (1→0 or 0→1). In addition, the simulation time is computed in variable `main_time` (line 176) and it is measured in nanoseconds (the clock cycle is 20 ns and thus the clock pulse 10 ns). Finally, note that the simulation finishes when the simulation time reaches value `timeout` (lines 171-174).

```
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 (tfp)
148         tfp->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     }
```

Figure 15. While loop for the simulation.

Assign a binary value of **0** to the new `i_sw0` signal before entering the loop (left part of Figure 16), and change it to **1** at time **30 us** inside the loop (see the right part of Figure 16).

```
136     top->clk = 1;
137     top->rst = 1;
138
139     top->i_sw0 = 0;
140
141     while (!(done || Verilated::gotFinish())) {
```

```
178     top->clk = !top->clk;
179     main_time+=10;
180
181     if (main_time == 30000) {
182       top->i_sw0 = 1;
183     }
184
185   }
```

Figure 16. Assign value to signal `i_sw0`.

3. Once you have performed all these changes, recompile RVfpgaSim by executing the following commands (this was explained in the GSG):

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

```
make clean
make
```

A new file *Vrvfpgasim* (the RVfpgaSim simulation binary), should be generated inside directory *[RVfpgaPath]/RVfpga/verilatorSIM*.

WINDOWS: You have to do this last step (step 4) inside the Cygwin terminal (refer to Section 6 and Appendix C in the Getting Started Guide for the detailed instructions). Note that the *C:* Windows folder can be found inside Cygwin at: */cygdrive/c*.

MacOS: Refer to Appendix D of the Getting Started Guide for the detailed instructions.

ii. Analyse the simulation of program *LedsSwitches.S*

In this section, we simulate the *LedsSwitches.S* example program (Figure 17) from the RVfpga Getting Started Guide. This program reads the values on the switches and writes that value to the LEDs on the Nexys A7 board. Note that we need to configure the enable register, so that the 32 input/output pins are configured as inputs or outputs, according to their connections. Specifically, the lower 16 pins of the GPIO are connected to the LEDs, so they are output pins with respect to the CPU (Enable=1). The upper 16 pins of the GPIO are connected to the switches, which are input pins with respect to the CPU (Enable=0). Because the switches occupy the upper 16 bits of the read register, they must be shifted to the right before writing their value to the LEDs.

```
#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
```

Figure 17. LedsSwitches.s for running in the SweRVolfX SoC

Follow the next steps for running the simulation.

1. Open VSCode/PlatformIO on your computer.

2. On the top bar, click on *File→Open Folder...* (Figure 18), and browse into directory *[RVfpgaPath]/RVfpga/examples/*

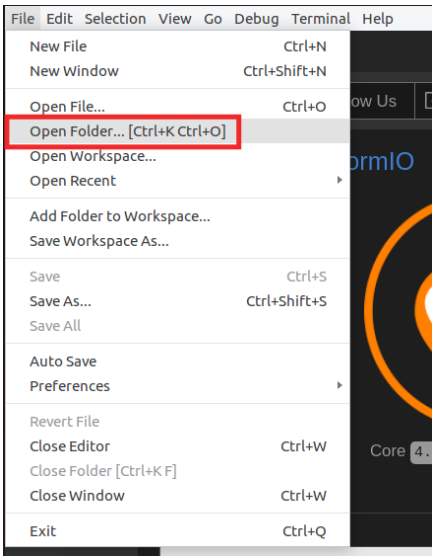

Figure 18. Open the LedsSwitches.S example

3. Select directory *LedsSwitches* (do not open it, but just select it) and click OK. The example will open in PlatformIO.

4. Open file *platformio.ini* and check if the path to the RVfpgaSim simulation binary (Figure 19) generated above (step 3 in the previous section) is correct. Remember from the GSG that it should look like:

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

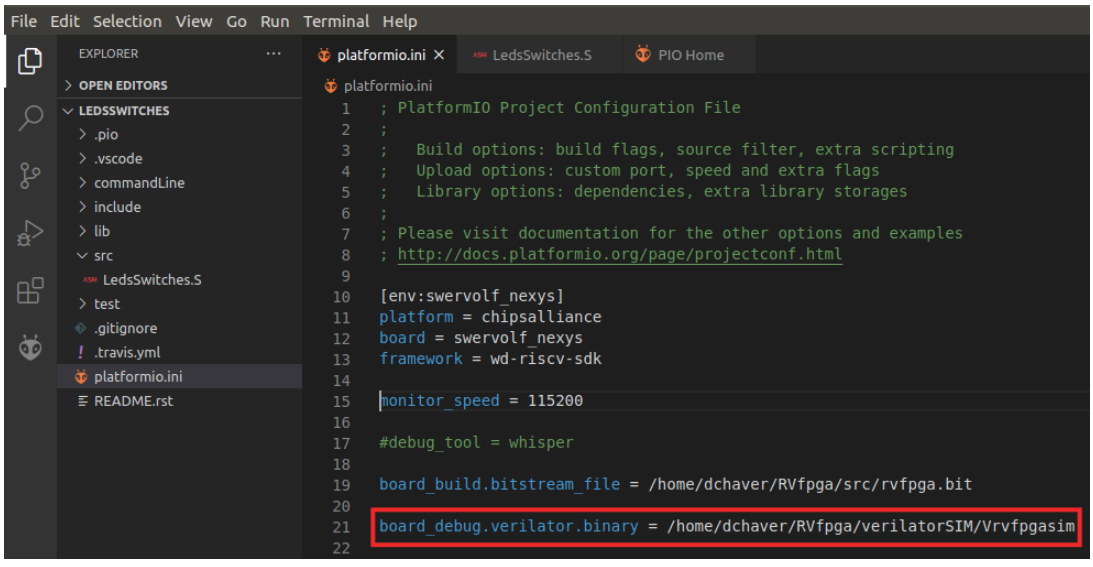

Figure 19. Platformio initialization file: platformio.ini

Windows: The RVfpgaSim simulation executable is called `Vrvfpgasim.exe`. Thus:

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

5. Run the simulation by clicking on the PlatformIO icon in the left menu ribbon  , then expand Project Tasks → env:swervolf_nexys → Platform and click on Generate Trace.

   File *trace.vcd* should have been generated inside *[RVfpgaPath]/RVfpga/examples/LedsSwitches/.pio/build/swervolf_nexys*, and you can open it with *GTKWave* by typing the following command into the PlatformIO terminal.

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

WINDOWS: folder *gtkwave64* that you downloaded, includes an application called *gtkwave.exe* inside the *bin* folder. Launch GTKWave by double clicking on that application. On the top part of the application, click on **File – Open New Tab**, and open the trace.vcd file generated in folder *[RVfpgaPath]/RVfpga/examples/LedsSwitches/.pio/build/swervolf_nexys*.

6. Include in the trace the following signals (go into module *rvfpgasim–swervolf* for finding each of these signals):
   - Add the clock signal: ***clk***
   - Add the GPIO input signal: ***i_gpio***
   - Add the GPIO output signal: ***o_gpio***

   In the graph (Figure 20), you will see that the value of the 16 switches (16 most significant bits of signal ***i_gpio***) is copied to the 16 LEDs (16 least significant bits of signal ***o_gpio***) with some delay. Moreover, the right-most switch changes (0→1) at time 30us, and this makes the right-most LED also change some time later.

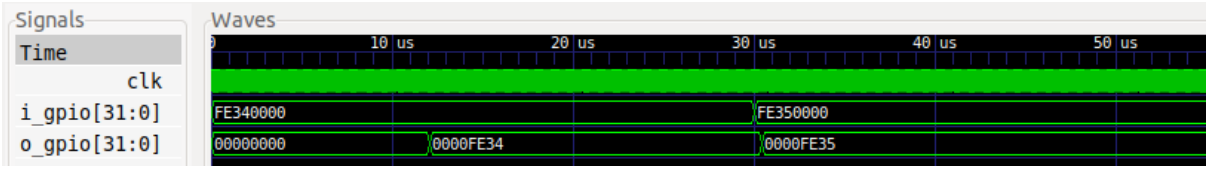

Figure 20. Simulation of the LedsSwitches program

7. ADVANCED EXERCISES

Exercise 2. Analyse the simulation from the previous section in more detail. Figure 21 shows the disassembly version of the .elf *LedsSwitches* program (Figure 17), with the three instructions that access the GPIO registers (Enable, Read and Write) highlighted. Remember from the Getting Started Guide that you can easily view in PlatformIO the disassembly version of the .elf program by opening file *firmware.dis*, which is generated at compilation time inside folder: `[RVfpgaPath]/RVfpga/examples/LedsSwitches/.pio/build/swervolf-nexys/` (see Figure 21).

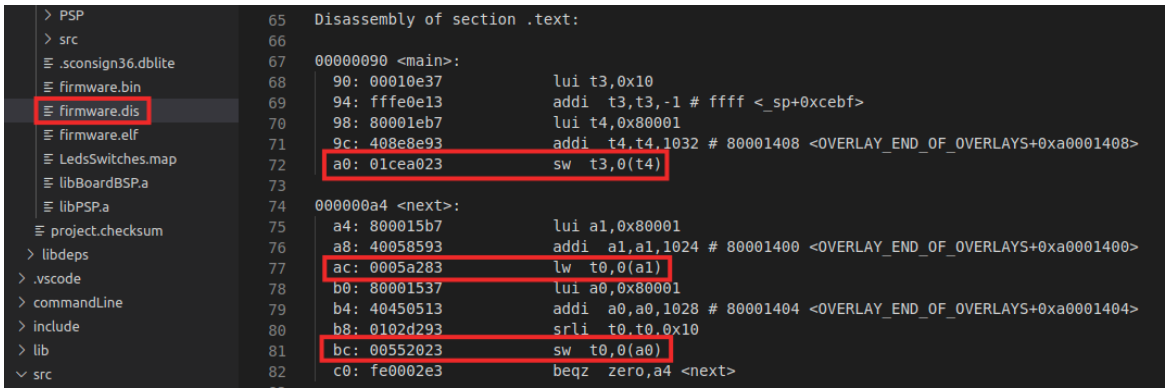

Figure 21. Disassembly version of program LedsSwitches.S

Simulate this program in RVfpgaSim and analyse the GPIO signals during the execution of each of the three memory instructions highlighted in red in Figure 21 (`sw`, `lw`, and `sw`). This will help you understand the GPIO low-level implementation explained in Section A.

You can start from the simulation from Section B and add and analyse the values for the following signals (go into the referred modules for locating each signal):

- rvfpgasim → swervolf → swerv_eh1 → swerv → ifu
  - Clock: ***clk***.
  - Fetched instructions: ***ifu_i0_instr*** and ***ifu_i1_instr***.
- rvfpgasim – swervolf
  - 32-bit input/output pins: ***i_gpio*** and ***o_gpio***.
  - Address provided by the CPU: ***wb_m2s_io_adr***.
- rvfpgasim – swervolf – gpio_module
  - GPIO External Interface: ***ext_pad_i***, ***ext_pad_o*** and ***ext_padoe_o***.
- rvfpgasim – swervolf – wb_intercon0
  - Output address and data signals for the multiplexer of Figure 2: ***wb_io_adr_i***, ***wb_io_dat_i***, ***wb_io_dat_o***.
  - Input GPIO data signals for the multiplexer of Figure 2: ***wb_gpio_adr_i***, ***wb_gpio_dat_i***, ***wb_gpio_dat_o***.
  - Selection signals for the multiplexer of Figure 2: ***wb_*_cyc_o***.
- rvfpgasim – swervolf – wb_intercon0 – wb_mux_io
  - Match signal for the multiplexer of Figure 2: ***match***.
- rvfpgasim – swervolf – swerv_eh1 – swerv – dec – arf – gpr_banks(0) – gpr(5) – gprff
  - Register value for t0: ***dout***.

Exercise 3. Expand **RVfpgaNexys** to support the five on-board pushbuttons. The pushbuttons are shown in Figure 22. The five buttons are named according to their location: up, down, left, right, and center – BTNU, BTND, BTNL, BTNR, BTNC.

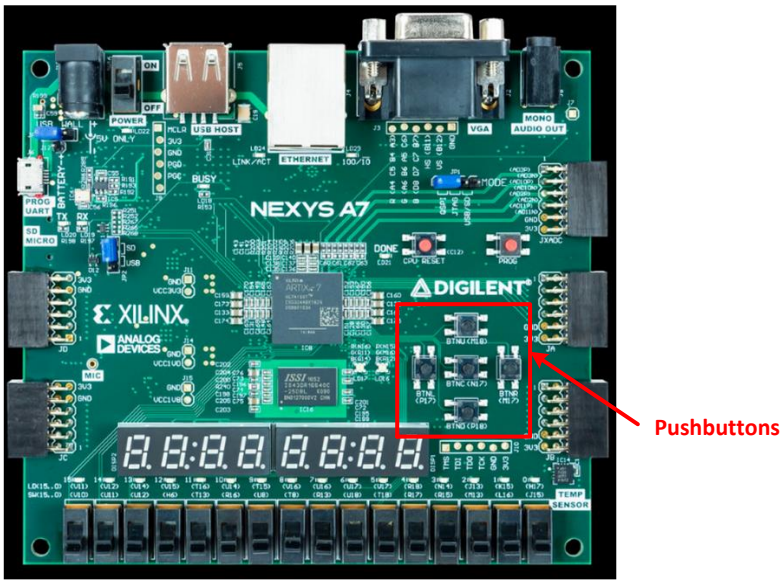

Figure 22. Pushbuttons on Nexys A7 FPGA Board

a. Given that the maximum size of the GPIO module that we are using (`gpio_top`) is 32, which is the number of I/O pins that we have (16 LEDs + 16 Switches), you need to include another instantiation of the GPIO module in SweRVolfX, as well as 5 new tri-state buffers and all the necessary signals.
b. Use the addresses starting at 0x80001800 (which are available) for mapping the registers exposed by the new GPIO controller. Note that you must modify the multiplexer (Figure 9) for including the new peripheral.
c. You must also modify the constraints file taking into account that the five pushbuttons are connected to the following FPGA pins:
    i. BTNC is connected to PIN N17
    ii. BTNU is connected to PIN M18
    iii. BTNL is connected to PIN P17
    iv. BTNR is connected to PIN M17
    v. BTND is connected to PIN P18

Exercise 4. Design another controller in **RVfpgaNexys** for the five on-board pushbuttons.

a. In contrast to Exercise 3, in this case you must implement your own GPIO controller in Verilog or SystemVerilog based on the scheme illustrated in Figure 3. In fact, you can even simplify that circuit and only include a **Read Register** (i.e. you do not need to include the tri-state buffers nor the **Write Register**).
b. You do not need to remove the controller from the previous exercise because the pushbuttons can be mapped to addresses not used by that GPIO controller.
c. Include the new controller inside the System Controller peripheral. You can use the address range 0x8000101C-0x8000101F, which is unused. Note that the registers included in the System Controller are read into the CPU by directly connecting them to the data signal of the Wishbone Bus (o_wb_rdt) based on the address (i_wb_adr) generated by the CPU. Inspect lines 234-266 of module **swervolf_syscon** (*[RVfpgaPath]/RVfpga/src/SweRVolfSoC/Peripherals/SystemController/swervolf_syscon.v*) to help you understand how to proceed.

Exercise 5. Write a RISC-V assembly program and a C program that displays an increasingly incrementing binary count on the LEDs, starting at 1. Include an empty loop for waiting between displaying each incremented value so that the values are viewable by the human eye. Read BTNC through the OpenCores peripheral implemented in Exercise 3 and use it to change the speed of the count, and read BTNU through the ad-hoc peripheral implemented in Exercise 4 and use it to restart the count whenever it is pressed.