THE IMAGINATION UNIVERSITY PROGRAMME

RVfpgaEL2 Lab 6
Introduction to I/O
Boolean board
(Revisions by Roy Kravitz)

1. INTRODUCTION

In Labs 6-10, you will learn how to use and expand RVfpgaEL2'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 Boolean 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 RVfpgaEL2 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 RVfpgaEL2 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 VeeRwolf 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 RVfpgaEL2Sim 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.

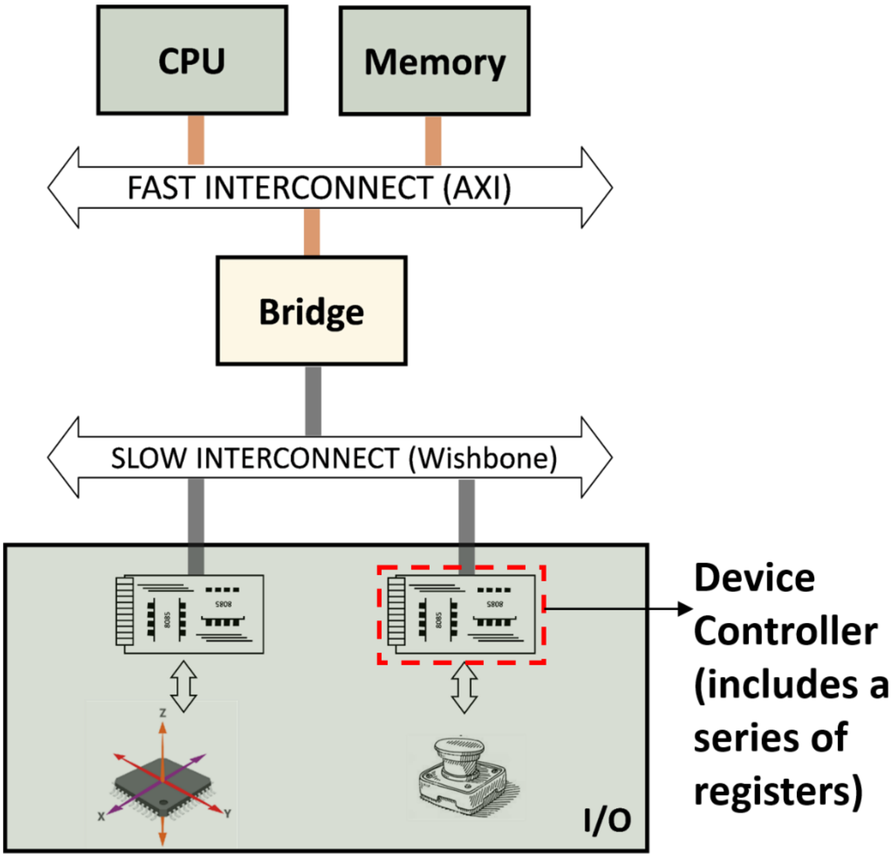

Figure 1 Generic Computing System

Figure 2 shows RVfpgaEL2's I/O system. It includes the following five peripherals:

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

A multiplexer selects one peripheral among the five 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 VeeR EL2 Core uses an AXI bridge (orange colour).

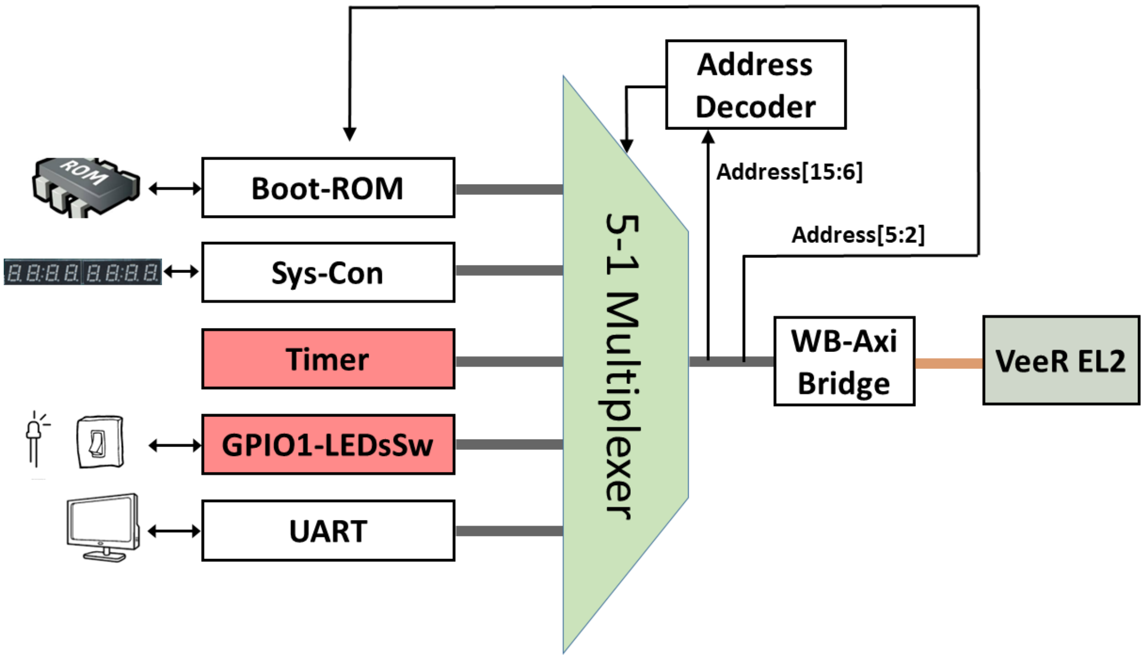

Figure 2. I/O System in the RVfpgaEL2 System

TASK: Locate each of the elements of Figure 2 in the SoC. You will need to inspect the following files and directories: *[RVfpgaBooleanPath]/src/VeeRwolf/veerwolf_core.v* (main file, where the elements from Figure 2 are instantiated). *[RVfpgaBooleanPath]/src/VeeRwolf/Peripherals* *[RVfpgaBooleanPath]/src/VeeRwolf/Interconnect/WishboneInterconnect* *[RVfpgaBooleanPath]/src/VeeRwolf/Peripherals/SystemController/veerwolf_syscon.v* *[RVfpgaBooleanPath]/src/VeeRwolf/Interconnect/WishboneInterconnect/wb_intercon.v* *[RVfpgaBooleanPath]/src/VeeRwolf/Interconnect/WishboneInterconnect/wb_intercon.vh*

As described in the RVfpgaEL2 Getting Started Guide, the original VeeRwolf (https://github.com/chipsalliance/VeeRwolf) includes only some of the peripherals shown in Figure 2: specifically, the Boot ROM, System Controller (with no 7-Segment Displays) and UART (shown in white in Figure 2). Remember from the GSG that VeeRwolf SoC extends the original VeeRwolf SoC with new peripherals: a Timer, a GPIO module (shown in red in Figure 2), and a 7-segment display controller (that extends VeeRwolf'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).

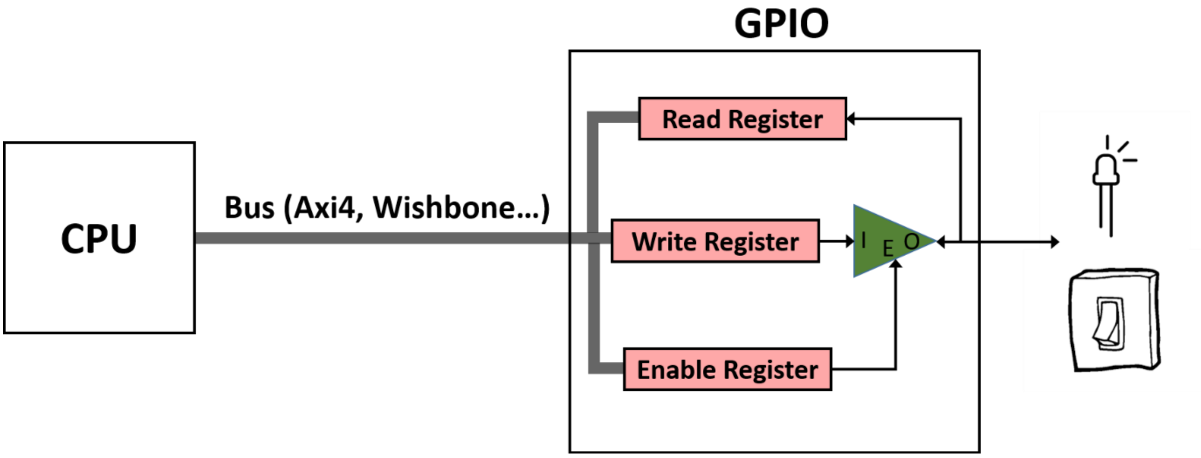

Figure 3. GPIO simplified circuit

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 RVfpgaEL2 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 VeeRwolf's GPIO and then we propose one exercise that uses this peripheral.

A. GPIO high-level specification

The GPIO module used in VeeRwolf 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: *[RVfpgaBooleanPath]/src/VeeRwolf/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: *[RVfpgaBooleanPath]/src/VeeRwolf/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 RVfpgaEL2-Boolean, the first 16 GPIO pins, pins 15:0, of the GPIO module are connected to the 16 LEDs on the Boolean 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.

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

In this section, we describe the low-level details of the GPIO used in VeeRwolf. We then propose some exercises where you will first modify the SoC to add a new GPIO peripheral and then 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 4: (1) RVfpgaEL2-Boolean' external connection to the on-board LEDs/Switches (left shaded region in Figure 4); (2) Integration of the GPIO module into the VeeRwolf SoC (middle shaded region in Figure 4); (3) Connection between the GPIO and the VeeR EL2 Core (right shaded region in Figure 4).

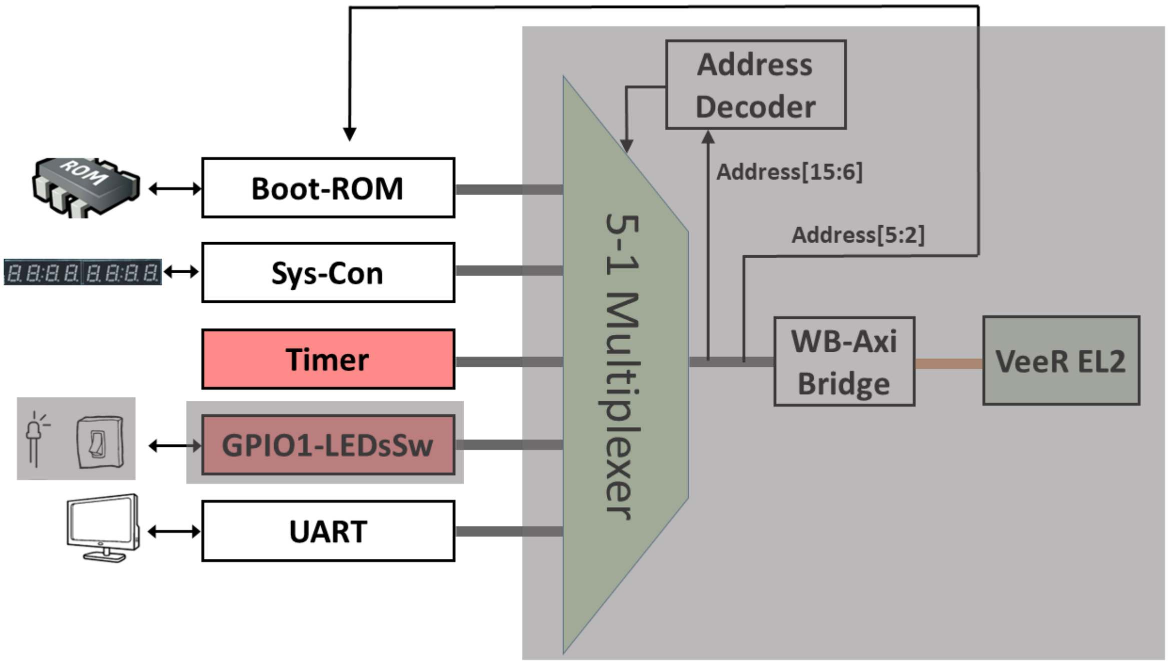

Figure 4. GPIO analysis in 3 phases

i. Connection of the LEDs/Switches with the SoC

The constraints file of the project (*[RVfpgaBooleanPath]/src/rvfpgaboolean.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 V2.

The Boolean board includes 16 LEDs and 16 Switches. The signal that connects the 16 LEDs with the top-module of the SoC (called rvfpgaboolean, available inside file *[RVfpgaBooleanPath]/src/rvfpgaboolean.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 5 shows the section of the Xilinx design constraint (xdc) file, rvfpgaboolean.xdc (available in *[RVfpgaBooleanPath]/src*) where these 32 connections between the signal and FPGA pin are defined.

```
# On-board Slide Switches
set_property -dict {PACKAGE_PIN V2 IOSTANDARD LVCMOS33} [get_ports {i_sw[0]}]
set_property -dict {PACKAGE_PIN U2 IOSTANDARD LVCMOS33} [get_ports {i_sw[1]}]
set_property -dict {PACKAGE_PIN U1 IOSTANDARD LVCMOS33} [get_ports {i_sw[2]}]
set_property -dict {PACKAGE_PIN T2 IOSTANDARD LVCMOS33} [get_ports {i_sw[3]}]
set_property -dict {PACKAGE_PIN T1 IOSTANDARD LVCMOS33} [get_ports {i_sw[4]}]
set_property -dict {PACKAGE_PIN R2 IOSTANDARD LVCMOS33} [get_ports {i_sw[5]}]
set_property -dict {PACKAGE_PIN R1 IOSTANDARD LVCMOS33} [get_ports {i_sw[6]}]
set_property -dict {PACKAGE_PIN P2 IOSTANDARD LVCMOS33} [get_ports {i_sw[7]}]
set_property -dict {PACKAGE_PIN P1 IOSTANDARD LVCMOS33} [get_ports {i_sw[8]}]
set_property -dict {PACKAGE_PIN N2 IOSTANDARD LVCMOS33} [get_ports {i_sw[9]}]
set_property -dict {PACKAGE_PIN N1 IOSTANDARD LVCMOS33} [get_ports {i_sw[10]}]
set_property -dict {PACKAGE_PIN M2 IOSTANDARD LVCMOS33} [get_ports {i_sw[11]}]
set_property -dict {PACKAGE_PIN M1 IOSTANDARD LVCMOS33} [get_ports {i_sw[12]}]
set_property -dict {PACKAGE_PIN L1 IOSTANDARD LVCMOS33} [get_ports {i_sw[13]}]
set_property -dict {PACKAGE_PIN K2 IOSTANDARD LVCMOS33} [get_ports {i_sw[14]}]
set_property -dict {PACKAGE_PIN K1 IOSTANDARD LVCMOS33} [get_ports {i_sw[15]}]

# On-board LEDs
set_property -dict {PACKAGE_PIN G1 IOSTANDARD LVCMOS33} [get_ports {o_led[0]}]
set_property -dict {PACKAGE_PIN G2 IOSTANDARD LVCMOS33} [get_ports {o_led[1]}]
set_property -dict {PACKAGE_PIN F1 IOSTANDARD LVCMOS33} [get_ports {o_led[2]}]
set_property -dict {PACKAGE_PIN F2 IOSTANDARD LVCMOS33} [get_ports {o_led[3]}]
set_property -dict {PACKAGE_PIN E1 IOSTANDARD LVCMOS33} [get_ports {o_led[4]}]
set_property -dict {PACKAGE_PIN E2 IOSTANDARD LVCMOS33} [get_ports {o_led[5]}]
set_property -dict {PACKAGE_PIN E3 IOSTANDARD LVCMOS33} [get_ports {o_led[6]}]
set_property -dict {PACKAGE_PIN E5 IOSTANDARD LVCMOS33} [get_ports {o_led[7]}]
set_property -dict {PACKAGE_PIN E6 IOSTANDARD LVCMOS33} [get_ports {o_led[8]}]
set_property -dict {PACKAGE_PIN C3 IOSTANDARD LVCMOS33} [get_ports {o_led[9]}]
set_property -dict {PACKAGE_PIN B2 IOSTANDARD LVCMOS33} [get_ports {o_led[10]}]
set_property -dict {PACKAGE_PIN A2 IOSTANDARD LVCMOS33} [get_ports {o_led[11]}]
set_property -dict {PACKAGE_PIN B3 IOSTANDARD LVCMOS33} [get_ports {o_led[12]}]
set_property -dict {PACKAGE_PIN A3 IOSTANDARD LVCMOS33} [get_ports {o_led[13]}]
set_property -dict {PACKAGE_PIN B4 IOSTANDARD LVCMOS33} [get_ports {o_led[14]}]
set_property -dict {PACKAGE_PIN A4 IOSTANDARD LVCMOS33} [get_ports {o_led[15]}]
```

Figure 5. Connection of i_sw[15:0] with the on-board switches and o_led[15:0] with the on-board LEDs (file *rvfpgaboolean.xdc*).

The top-module (**rvfpgaboolean**) shows these two signals connected to the SoC (top of Figure 6), and the end of that module shows their connection with the **veerwolf_core** module (bottom of Figure 6). 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 **veerwolf_core** module (as will be shown later, in Figure 7). Moreover, note that the *o_led* signal is latched through an intermediate signal, *gpio_out* (line 266).

```
module rvfpgaboolean
  #(parameter bootrom_file = "boot_main.mem")
   (input  wire         clk,
    input  wire         i_uart_rx,
    output wire         o_uart_tx,
    inout   wire [15:0]  i_sw,
    output  reg  [15:0]  o_led
```

```
    .io_data          ({i_sw[15:0],gpio_out[15:0]})
    .AN (AN),
    .Digits_Bits ({CA,CB,CC,CD,CE,CF,CG}));

always @(posedge clk_core) begin
    o_led[15:0] <= gpio_out[15:0];
end
```

Figure 6. Connection of the LEDs and the Switches with the top-module (*rvfpgaboolean.sv*)

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

[RVfpgaBooleanPath]/src/rvfpgaboolean.xdc
[RVfpgaBooleanPath]/src/rvfpgaboolean.sv
[RVfpgaBooleanPath]/src/VeeRwolf/veerwolf_core.v

In the previous section we said that in RVfpgaEL2-Boolean 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 7?

ii. Integration of the GPIO module in the SoC

In the **swervolf_core** module (*[RVfpgaBooleanPath]/src/VeeRwolf/veerwolf_core.v*), the GPIO module is instantiated and integrated into the SoC (see Figure 7).

```verilog
// GPIO - Leds and Switches
wire [31:0] en_gpio;
wire        gpio_irq;


`ifdef ViDBo
`elsif Pipeline
`else

  wire [31:0] i_gpio;
  wire [31:0] o_gpio;

  bidirec gpio0  (.oe(en_gpio[0] ), .inp(o_gpio[0] ), .outp(i_gpio[0] ), .bidir(io_data[0] ));
  bidirec gpio1  (.oe(en_gpio[1] ), .inp(o_gpio[1] ), .outp(i_gpio[1] ), .bidir(io_data[1] ));
  bidirec gpio2  (.oe(en_gpio[2] ), .inp(o_gpio[2] ), .outp(i_gpio[2] ), .bidir(io_data[2] ));
  bidirec gpio3  (.oe(en_gpio[3] ), .inp(o_gpio[3] ), .outp(i_gpio[3] ), .bidir(io_data[3] ));
  bidirec gpio4  (.oe(en_gpio[4] ), .inp(o_gpio[4] ), .outp(i_gpio[4] ), .bidir(io_data[4] ));
  bidirec gpio5  (.oe(en_gpio[5] ), .inp(o_gpio[5] ), .outp(i_gpio[5] ), .bidir(io_data[5] ));
  bidirec gpio6  (.oe(en_gpio[6] ), .inp(o_gpio[6] ), .outp(i_gpio[6] ), .bidir(io_data[6] ));
  bidirec gpio7  (.oe(en_gpio[7] ), .inp(o_gpio[7] ), .outp(i_gpio[7] ), .bidir(io_data[7] ));
  bidirec gpio8  (.oe(en_gpio[8] ), .inp(o_gpio[8] ), .outp(i_gpio[8] ), .bidir(io_data[8] ));
  bidirec gpio9  (.oe(en_gpio[9] ), .inp(o_gpio[9] ), .outp(i_gpio[9] ), .bidir(io_data[9] ));
  bidirec gpio10 (.oe(en_gpio[10]), .inp(o_gpio[10]), .outp(i_gpio[10]), .bidir(io_data[10]));
  bidirec gpio11 (.oe(en_gpio[11]), .inp(o_gpio[11]), .outp(i_gpio[11]), .bidir(io_data[11]));
  bidirec gpio12 (.oe(en_gpio[12]), .inp(o_gpio[12]), .outp(i_gpio[12]), .bidir(io_data[12]));
  bidirec gpio13 (.oe(en_gpio[13]), .inp(o_gpio[13]), .outp(i_gpio[13]), .bidir(io_data[13]));
  bidirec gpio14 (.oe(en_gpio[14]), .inp(o_gpio[14]), .outp(i_gpio[14]), .bidir(io_data[14]));
  bidirec gpio15 (.oe(en_gpio[15]), .inp(o_gpio[15]), .outp(i_gpio[15]), .bidir(io_data[15]));
  bidirec gpio16 (.oe(en_gpio[16]), .inp(o_gpio[16]), .outp(i_gpio[16]), .bidir(io_data[16]));
  bidirec gpio17 (.oe(en_gpio[17]), .inp(o_gpio[17]), .outp(i_gpio[17]), .bidir(io_data[17]));
  bidirec gpio18 (.oe(en_gpio[18]), .inp(o_gpio[18]), .outp(i_gpio[18]), .bidir(io_data[18]));
  bidirec gpio19 (.oe(en_gpio[19]), .inp(o_gpio[19]), .outp(i_gpio[19]), .bidir(io_data[19]));
  bidirec gpio20 (.oe(en_gpio[20]), .inp(o_gpio[20]), .outp(i_gpio[20]), .bidir(io_data[20]));
  bidirec gpio21 (.oe(en_gpio[21]), .inp(o_gpio[21]), .outp(i_gpio[21]), .bidir(io_data[21]));
  bidirec gpio22 (.oe(en_gpio[22]), .inp(o_gpio[22]), .outp(i_gpio[22]), .bidir(io_data[22]));
  bidirec gpio23 (.oe(en_gpio[23]), .inp(o_gpio[23]), .outp(i_gpio[23]), .bidir(io_data[23]));
  bidirec gpio24 (.oe(en_gpio[24]), .inp(o_gpio[24]), .outp(i_gpio[24]), .bidir(io_data[24]));
  bidirec gpio25 (.oe(en_gpio[25]), .inp(o_gpio[25]), .outp(i_gpio[25]), .bidir(io_data[25]));
  bidirec gpio26 (.oe(en_gpio[26]), .inp(o_gpio[26]), .outp(i_gpio[26]), .bidir(io_data[26]));
  bidirec gpio27 (.oe(en_gpio[27]), .inp(o_gpio[27]), .outp(i_gpio[27]), .bidir(io_data[27]));
  bidirec gpio28 (.oe(en_gpio[28]), .inp(o_gpio[28]), .outp(i_gpio[28]), .bidir(io_data[28]));
  bidirec gpio29 (.oe(en_gpio[29]), .inp(o_gpio[29]), .outp(i_gpio[29]), .bidir(io_data[29]));
  bidirec gpio30 (.oe(en_gpio[30]), .inp(o_gpio[30]), .outp(i_gpio[30]), .bidir(io_data[30]));
  bidirec gpio31 (.oe(en_gpio[31]), .inp(o_gpio[31]), .outp(i_gpio[31]), .bidir(io_data[31]));

`endif


  gpio_top gpio_module(
      .wb_clk_i      (clk),
      .wb_rst_i      (wb_rst),
      .wb_cyc_i      (wb_m2s_gpio_cyc),
      .wb_adr_i      ({2'b0,wb_m2s_gpio_adr[5:2],2'b0}),
      .wb_dat_i      (wb_m2s_gpio_dat),
      .wb_sel_i      (4'b1111),
      .wb_we_i       (wb_m2s_gpio_we),
      .wb_stb_i      (wb_m2s_gpio_stb),
      .wb_dat_o      (wb_s2m_gpio_dat),
      .wb_ack_o      (wb_s2m_gpio_ack),
      .wb_err_o      (wb_s2m_gpio_err),
      .wb_inta_o     (gpio_irq),
      // External GPIO Interface

  `ifdef ViDBo
      .ext_pad_i      (i_data[31:0]),
      .ext_pad_o      (o_data[31:0]),
  `elsif Pipeline
      .ext_pad_i      (i_data[31:0]),
      .ext_pad_o      (o_data[31:0]),
  `else
      .ext_pad_i      (i_gpio[31:0]),
      .ext_pad_o      (o_gpio[31:0]),
  `endif

      .ext_padoe_o    (en_gpio));
```

Figure 7. Integration of the GPIO module (file *veerwolf_core.v*).

The interface of the module can be divided into two blocks: Wishbone signals (Table 3), which allow the VeeR EL2 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 Figure 7, 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 7).

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 board LEDs and Switches (*i_gpio*, *o_gpio*, *io_data*) through 32 tri-state buffer modules. 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 6) and thus they must be configured as outputs; the upper 16 pins, 31:16, are connected to the switches (Figure 6) 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:

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

Note that the simulators do not use the tristate buffers.

<u>TASKS</u>: The board GPIO pins (*io_data*) are connected to the GPIO module through tri-state buffers (see Figure 7). 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 VeeR EL2 Core

As shown in Figure 2, the device controllers are connected to the VeeR EL2 Core through a multiplexer and a bridge. The multiplexer selects one among the N possible peripherals (in our case, N=5), 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 VeeR Core and vice versa (implemented in file *[RVfpgaBooleanPath]/src/VeeRwolf/Interconnect/AxiToWb/axi2wb.v*).

The 5:1 multiplexer (Figure 8) is instantiated in file *[RVfpgaBooleanPath]/src/VeeRwolf/Interconnect/WishboneInterconnect/wb_intercon.v*. Then, the **wb_intercon** module is instantiated in file *[RVfpgaBooleanPath]/src/VeeRwolf/Interconnect/WishboneInterconnect/wb_intercon.vh*. This latter file is included in the **veerwolf_core** module located here: *[RVfpgaBooleanPath]/src/VeeRwolf/veerwolf_core.v*.

```
wb_mux
  #(.num_slaves (5),
    .MATCH_ADDR ({32'h00000000, 32'h00001000, 32'h00001200, 32'h00001400, 32'h00002000}),
    .MATCH_MASK ({32'hfffff000, 32'hffffffc0, 32'hffffffc0, 32'hffffffc0, 32'hfffff000}))
 wb_mux_io
   (.wb_clk_i  (wb_clk_i),
    .wb_rst_i  (wb_rst_i),
    .wbm_adr_i (wb_io_adr_i),
    .wbm_dat_i (wb_io_dat_i),
    .wbm_sel_i (wb_io_sel_i),
    .wbm_we_i  (wb_io_we_i),
    .wbm_cyc_i (wb_io_cyc_i),
    .wbm_stb_i (wb_io_stb_i),
    .wbm_cti_i (wb_io_cti_i),
    .wbm_bte_i (wb_io_bte_i),
    .wbm_dat_o (wb_io_dat_o),
    .wbm_ack_o (wb_io_ack_o),
    .wbm_err_o (wb_io_err_o),
    .wbm_rty_o (wb_io_rty_o),
    .wbs_adr_o ({wb_rom_adr_o, wb_sys_adr_o, wb_ptc_adr_o, wb_gpio_adr_o, wb_uart_adr_o}),
    .wbs_dat_o ({wb_rom_dat_o, wb_sys_dat_o, wb_ptc_dat_o, wb_gpio_dat_o, wb_uart_dat_o}),
    .wbs_sel_o ({wb_rom_sel_o, wb_sys_sel_o, wb_ptc_sel_o, wb_gpio_sel_o, wb_uart_sel_o}),
    .wbs_we_o  ({wb_rom_we_o,  wb_sys_we_o,  wb_ptc_we_o,  wb_gpio_we_o,  wb_uart_we_o}),
    .wbs_cyc_o ({wb_rom_cyc_o, wb_sys_cyc_o, wb_ptc_cyc_o, wb_gpio_cyc_o, wb_uart_cyc_o}),
    .wbs_stb_o ({wb_rom_stb_o, wb_sys_stb_o, wb_ptc_stb_o, wb_gpio_stb_o, wb_uart_stb_o}),
    .wbs_cti_o ({wb_rom_cti_o, wb_sys_cti_o, wb_ptc_cti_o, wb_gpio_cti_o, wb_uart_cti_o}),
    .wbs_bte_o ({wb_rom_bte_o, wb_sys_bte_o, wb_ptc_bte_o, wb_gpio_bte_o, wb_uart_bte_o}),
    .wbs_dat_i ({wb_rom_dat_i, wb_sys_dat_i, wb_ptc_dat_i, wb_gpio_dat_i, wb_uart_dat_i}),
    .wbs_ack_i ({wb_rom_ack_i, wb_sys_ack_i, wb_ptc_ack_i, wb_gpio_ack_i, wb_uart_ack_i}),
    .wbs_err_i ({wb_rom_err_i, wb_sys_err_i, wb_ptc_err_i, wb_gpio_err_i, wb_uart_err_i}),
    .wbs_rty_i ({wb_rom_rty_i, wb_sys_rty_i, wb_ptc_rty_i, wb_gpio_rty_i, wb_uart_rty_i}));
```

Figure 8. 5-1 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) with the Wishbone Bus of one peripheral, depending on the address. 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_**.

The multiplexer is implemented in file *[RVfpgaBooleanPath]/src/VeeRwolf/Interconnect/WishboneInterconnect/wb_intercon_1.2.2-r1/rtl/verilog/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:
[RVfpgaBooleanPath]/src/VeeRwolf/Peripherals/SystemController/veerwolf_syscon.v
[RVfpgaBooleanPath]/src/VeeRwolf/Interconnect/WishboneInterconnect/wb_intercon.v
[RVfpgaBooleanPath]/src/VeeRwolf/Interconnect/WishboneInterconnect/wb_intercon.vh
[RVfpgaBooleanPath]/src/VeeRwolf/Interconnect/WishboneInterconnect/wb_intercon_1.2.2-r1/rtl/verilog/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.

7. ADVANCED EXERCISES

Exercise 2. Expand **RVfpgaEL2-Boolean** to support the four on-board pushbuttons. The pushbuttons are shown in Figure 9. The four buttons are named BTN0(upper left), BTN1 (upper right), BTN2 (lower left), BTN3 (lower right).

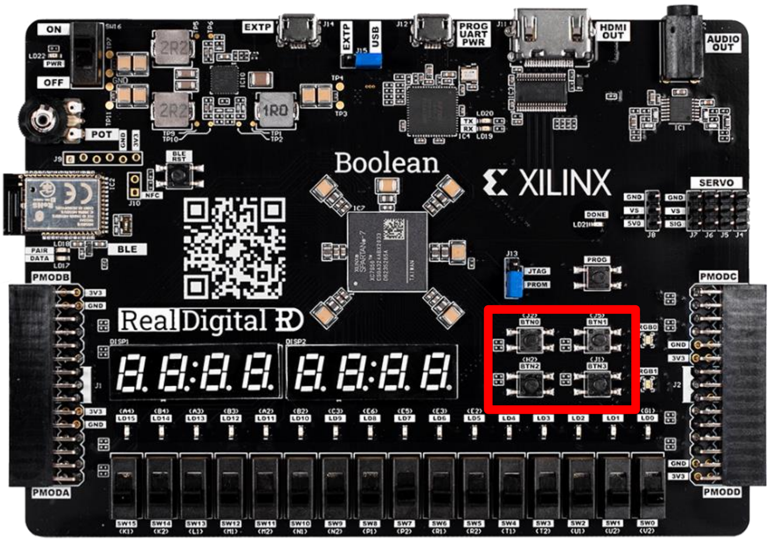

Figure 9. Pushbuttons on Boolean FPGA Board

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 VeeRwolf, as well as 4 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 8) for including the new peripheral.

   NOTE: You can automate the process of extending the multiplexer with the help of a script instead of doing it by hand. For that purpose, follow the next steps:
   - Download the Wishbone Interconnect Utilities from:
     https://github.com/olofk/wb_intercon/tree/1250154467e4a5658043f4be3945fc15a7808551
   - Unzip the downloaded file and go into the **sw** folder.
   - Run the following command, with a **config.yml** file that specifies the number of peripherals and their mapping: **python3 wb_intercon_gen2.py config.yml**

For example, a config.yml file that creates a multiplexer for the default peripherals plus the pushbuttons would look like this:

```
files_root: .
vlnv: ::wb_intercon:0
parameters:
  masters:
    io:
      slaves : [rom, sys, spi_flash, spi_accel, ptc, gpio, gpio2, uart]
  slaves:
    rom:
      offset : 0x00000000
      size : 0x00001000
    sys:
      offset : 0x00001000
      size : 0x00000040
    spi_flash:
      offset : 0x00001040
      size : 0x00000040
    spi_accel:
      offset : 0x00001100
      size : 0x00000040
    ptc:
      offset : 0x00001200
      size : 0x00000040
    gpio:
      offset : 0x00001400
      size : 0x00000040
    gpio2:
      offset : 0x00001800
      size : 0x00000040
    uart:
      offset : 0x00002000
      size : 0x00001000
```

You will obtain the two files that implement the multiplexer, **wb_intercon.v** and **wb_intercon.vh** which you can then use in your extended SoC.

Note (RK): You may get syntax errors in `wb_intercon.v`. This is because the inputs and outputs are not given type (like wire). Add the line:

```
`default_nettype wire
```

before the module declaration.

c. You must also modify the constraints file considering that the four pushbuttons are connected to the following FPGA pins:
   i. BTN0 is connected to PIN J2
   ii. BTN1is connected to PIN J5
   iii. BTN2 is connected to PIN H2
   iv. BTN3 is connected to PIN J1

Exercise 3. 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 BTN0 through the GPIO peripheral you added in Exercise 2 and use it to change the speed of the count. Read BTN1 and use it to set the count to 1 whenever it is pressed.