

**THE IMAGINATION UNIVERSITY PROGRAMME**

**RVfpgaEL2 Lab 6**

**Introduction to I/O**

# 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 Basys 3 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:** Lab 10 is not present for the Basys 3 Board, because that board does not have an 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, and introducing advanced exercises (Sections 6 and 7).

We use this same general structure in Labs 7-9. 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 peripherals. 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.

RISC-V INSTRUCTI

ISC-V INSTRUCTION SET

# 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 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
* SPI Flash

A multiplexer selects one peripheral among the six 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).

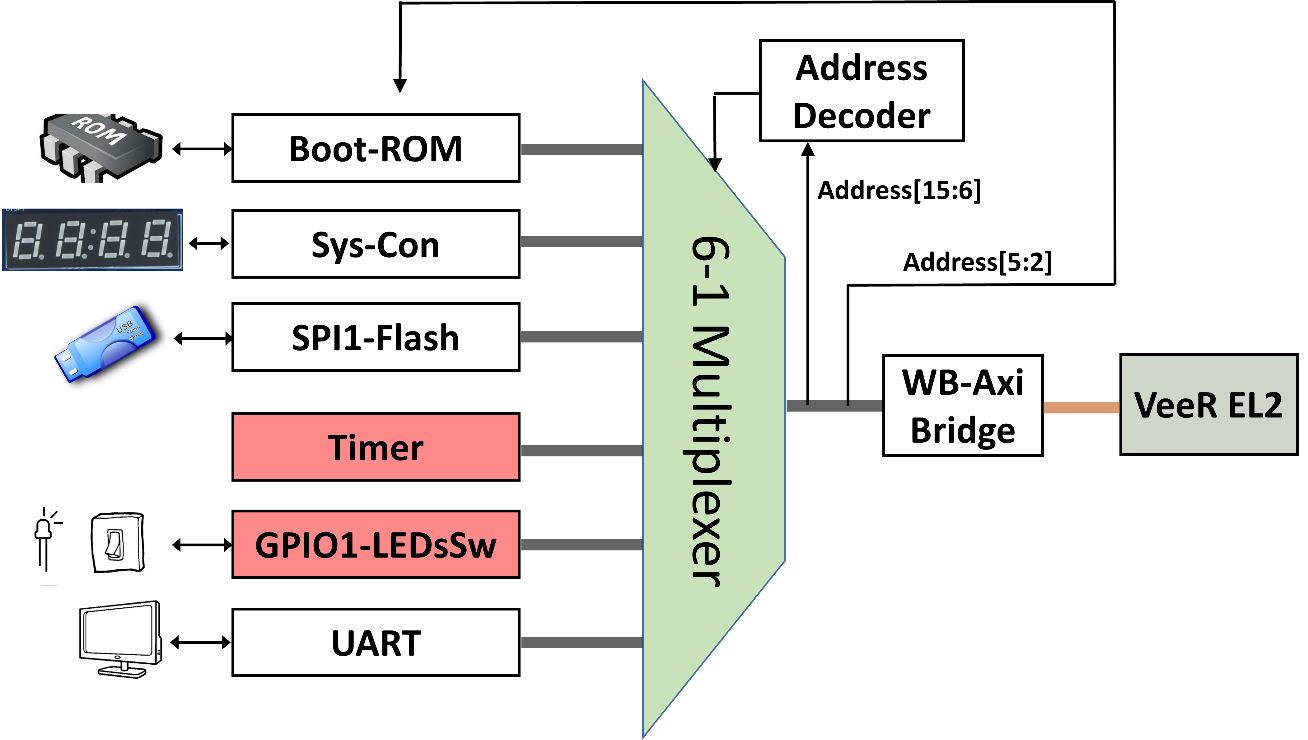


Figure 2. RVfpgaEL2’s I/O system

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

*[RVfpgaBasysPath]/src/VeeRwolf/veerwolf\_core.v* (main file, where the elements from Figure 2 are instantiated).

*[RVfpgaBasysPath]/src/VeeRwolf/Peripherals*

*[RVfpgaBasysPath]/src/VeeRwolf/Interconnect/WishboneInterconnect*

*[RVfpgaBasysPath]/src/VeeRwolf/Peripherals/SystemController/veerwolf\_syscon.v*

*[RVfpgaBasysPath]/src/VeeRwolf/Interconnect/WishboneInterconnect/wb\_intercon.v*

*[RVfpgaBasysPath]/src/VeeRwolf/Interconnect/WishboneInterconnect/wb\_intercon.vh*

As described in the RVfpgaEL2 Getting Started Guide (GSG), the original VeeRwolf(<https://github.com/chipsalliance/VeeRwolf>) system-on-chip (SoC) includes only some of the peripherals shown in Figure 2: specifically, the Boot ROM, System Controller (with no 7-Segment Displays), SPI Flash and UART (shown in white in Figure 2). Remember from the GSG that VeeRwolfX SoC extends the original VeeRwolf SoC with new peripherals: a Timer, a GPIO module (shown in red in Figure 2), an SPI Accelerometer 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. Particular memory addresses are assigned to hold values in registers that are then used by the peripherals. These addresses 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.

# 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.

# 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.

1. **GPIO Module**

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: *[RVfpgaBasysPath]/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 read the complete specifications in *gpio\_spec.pdf*.

The GPIO module has the following main features:

* It uses a Wishbone Interconnect.
* 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 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: *[RVfpgaBasysPath]/src/VeeRwolf/Peripherals/gpio/gpio\_top.v*.

R

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-Basys3, the lower 16 GPIO pins, pins 15:0, of the GPIO module are connected to the 16 LEDs on the Basys 3 board. The upper 16 GPIO pins, pins 31:16, of the GPIO controller are connected to the 16 on-board switches.

# Fundamental Exercises

# Exercise 1. Write either a RISC-V assembly program or 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.

Recall that you can run the same program both in RVfpgaEL2-Basys3 and RVfpgaEL2-ViDBo. For example, this figure shows the program running on RVfpgaEL2-ViDBo:

A close-up of a circuit board

Description automatically generated

# 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.

1. **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-Basys3’ 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).

A screenshot of a computer

Description automatically generated

Figure 4. GPIO analysis in 3 phases

1. **Connection of the LEDs/Switches with the SoC**

The project’s constraints file (*[RVfpgaBasysPath]/src/rvfpgabasys3.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 V17.

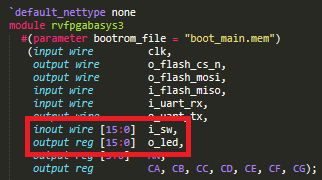
The Basys 3 board includes 16 LEDs and 16 Switches. The signal that connects the 16 LEDs with the top-module of the SoC (called rvfpgabasys3, available inside file *[RVfpgaBasysPath]/src/rvfpgabasys3.sv*) is called o\_led[15:0], and the signal that connects the 16 Switches with top-module is called i\_sw[15:0]. Figure 5 shows the section of the Xilinx design constraint (xdc) file, *rvfpgabasys3.xdc* (available in *[RVfpgaBasysPath]/src*) where these 32 connections between the signal and FPGA pin are defined.

A screenshot of a computer program

Description automatically generated

Figure 5. Connection of i\_sw[15:0] with the on-board switches and o\_led[15:0] with the on-board LEDs (file *rvfpgabasys3.xdc*).

The top-module (**rvfpgabasys3**) 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 240), 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 245).



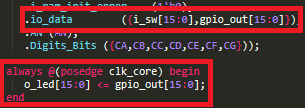


Figure 6. Connection of the LEDs and the switches with the top-level module (rvfpgabasys3.sv)

**TASKS:** 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:

*[RVfpgaBasysPath]/src/rvfpgabasys3.xdc*

*[RVfpgaBasysPath]/src/rvfpgabasys3.sv*

*[RVfpgaBasysPath]/src/VeeRwolf/veerwolf\_core.v*

In the previous section we said that in RVfpgaEL2-Basys3 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?

1. **Integration of the GPIO module in the SoC**

In the **veerwolf\_core** module (*[RVfpgaBasysPath]/src/VeeRwolf/veerwolf\_core.v*), the GPIO module is instantiated and integrated into the SoC (see Figure 7).

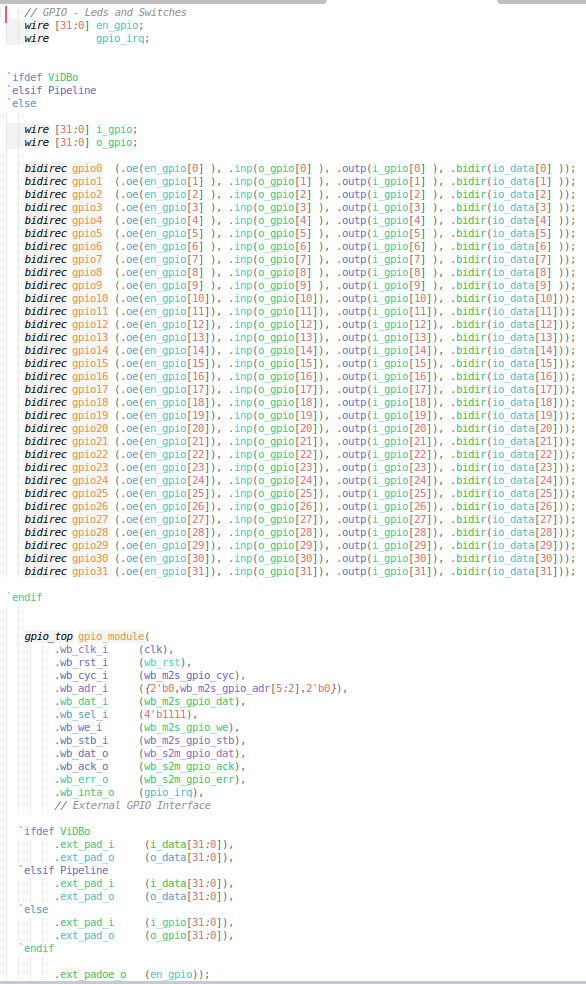


Figure 7. Integration of the GPIO module (file *veerwolf\_core.v*).

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** |
| ext\_pad\_i | 1-32 | Inputs | GPIO inputs |
| ext\_pad\_o | 1-32 | Outputs | GPIO outputs |
| ext\_padoe\_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 (determined by signal ext\_padoe\_o). 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 **veerwolf\_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.

**TASKS:** 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, which, as can be seen in Figure 7, is connected with ext\_padoe\_o?

1. **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* = 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 VeeR Core and vice versa (implemented in file *[RVfpgaBasysPath]/src/VeeRwolf/Interconnect/AxiToWb/axi2wb.v*).

The 6:1 multiplexer (Figure 8) is instantiated in file *[RVfpgaBasysPath]/src/VeeRwolf/Interconnect/WishboneInterconnect/wb\_intercon.v*. Then, the **wb\_intercon** module is instantiated in file *[RVfpgaBasysPath]/src/VeeRwolf/Interconnect/WishboneInterconnect/wb\_intercon.vh*. This latter file is included in the **veerwolf\_core** module located here: *[RVfpgaBasysPath]/src/VeeRwolf/veerwolf\_core.v*.

A screenshot of a computer program

Description automatically generated

Figure 8. 6:1 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 *[RVfpgaBasysPath]/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:

*[RVfpgaBasysPath]/src/VeeRwolf/Peripherals/SystemController/veerwolf\_syscon.v*

*[RVfpgaBasysPath]/src/VeeRwolf/Interconnect/WishboneInterconnect/wb\_intercon.v*

*[RVfpgaBasysPath]/src/VeeRwolf/Interconnect/WishboneInterconnect/wb\_intercon.vh*

*[RVfpgaBasysPath]/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.

RISC-V INSTRUCTION

# Advanced Exercises

# Exercise 2. Expand RVfpgaEL2-Basys3 and RVfpgaEL2-ViDBo to support the five on-board pushbuttons. The pushbuttons are shown in Figure 21. The five buttons are named according to their location: up, down, left, right, and center – BTNU, BTND, BTNL, BTNR, BTNC.

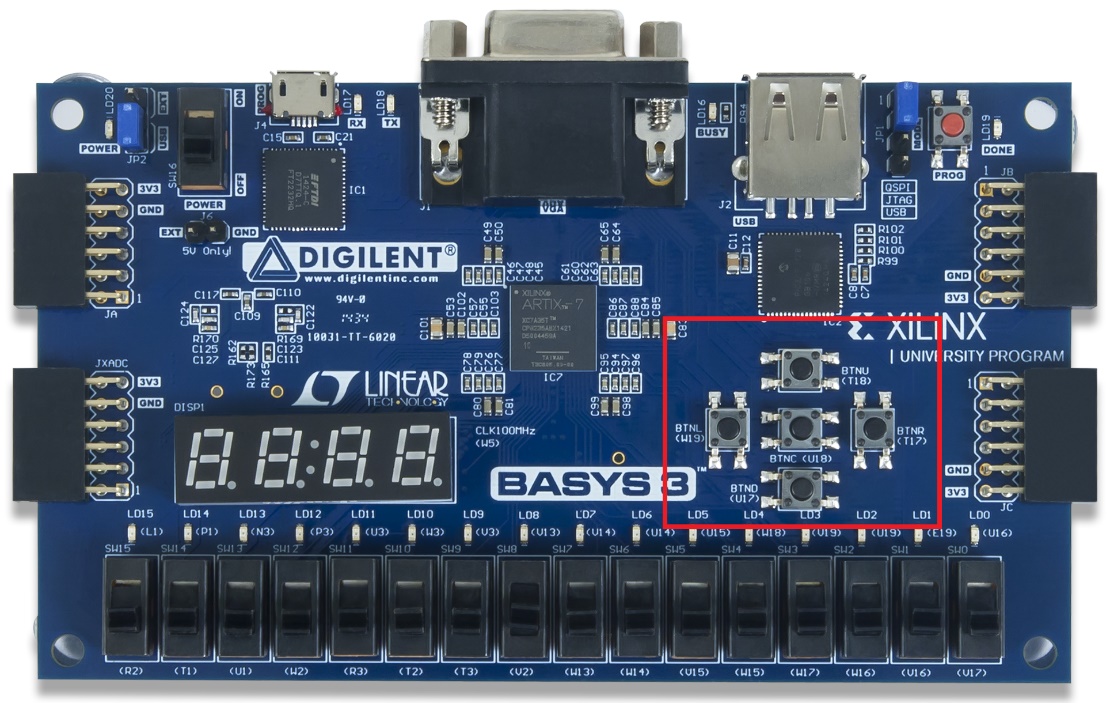


Figure 21. Pushbuttons on Basys 3 FPGA Board

* 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 instance of the GPIO module in VeeRwolf, as well as 5 new tri-state buffers and all the necessary signals.
* Extend the multiplexer instantiation (explained in detail in the previous section, see Figure 8) for connecting the new peripheral with the core (files *wb\_intercon.v* and *wb\_intercon.vh*). You must create and connect the signals for the new gpio instantiation and assign addresses to its registers. Use the addresses starting at 0x80001800, which are free, for mapping the registers exposed by the new GPIO controller (remember that the 16 most significant bits in parameter MATCH\_ADDR are set to 0).

**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, ptc, gpio, gpio2, uart]  
 slaves:  
 rom:  
 offset : 0x00000000  
 size : 0x00001000  
 sys:  
 offset : 0x00001000  
 size : 0x00000040  
 spi\_flash:  
 offset : 0x00001040  
 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.

* When you use **RVfpgaEL2-ViDBo**, in file *[RVfpgaBasysPath]/Simulators/SimulationSources/rvfpgasim.v*, a 5-bit input is provided, called i\_pb, that carries the values provided from the virtual board pushbuttons. This signal must be connected with the new GPIO module. Remember that the SweRVolf SoC is instantiated from this module (rvfpgasim).

If you use Windows, do not forget to reference the *libwebsockets* library correctly as explained in the GSG.

* When you use **RVfpgaEL2-Basys3**, you must also modify the constraints file (*[RVfpgaBasysPath]/src/rvfpgabasys3.xdc*), to take into account that the four pushbuttons are connected to the following FPGA pins:
  + btn[0] (labelled btnc on the board) is connected to PIN U18
  + btn[1] (labelled btnu on the board) is connected to PIN T18
  + btn[2] (labelled btnl on the board) is connected to PIN W19
  + btn[3] (labelled btnr on the board) is connected to PIN T17
  + btn[4] (labelled btnd on the board) is connected to PIN U17

# Exercise 3. Write a RISC-V assembly program and a C program that displays a binary number on the LEDs that starts at the value 1 and increments by 1. Include an empty loop for waiting between displaying each incremented value so that the values are viewable by the human eye. Read BTN0 and BTN1 through the peripheral implemented in Exercise 2. Use BTN0 to change the speed of the count and BTN1 to restart the count whenever it is pressed.

Recall that you can run the same program on either RVfpgaEL2-Basys3 or RVfpgaEL2-ViDBo.