



THE IMAGINATION UNIVERSITY PROGRAMME

RVfpga Lab 6

Introdução às E/S

1. INTRODUÇÃO

Nos Labs 6-10, aprenderá como utilizar e expandir o sistema de Entrada/Saída (E/S) do RVfpga para permitir ao processador RISC-V interagir com dispositivos periféricos. Abaixo encontra-se uma visão geral dos tópicos abordados nestes laboratórios:

- **Lab 6:** Aprender a utilizar os pinos de entrada/saída de uso geral (GPIO) ligados aos LEDs, interruptores e botões na placa Nexys A7
- **Lab 7:** Aprender a utilizar os mostradores de 7 segmentos disponíveis na placa
- **Lab 8:** Aprender a utilizar temporizadores
- **Lab 9:** Aprender a utilizar interrupções para fazer interface com dispositivos externos
- **Lab 10:** Aprender a ligar o Sistema RVfpga com o acelerómetro SPI da placa

Neste laboratório, descrevemos primeiro as principais características de um sistema de I/O de uso geral e o utilizado no Sistema RVfpga (Secção 2). Descrevemos depois uma versão teórica simplificada de um controlador GPIO genérico (Secção 3). Finalmente, concentramo-nos no controlador GPIO utilizado no SweRVofX SoC: analisamos primeiro a sua especificação de alto nível e introduzimos exercícios fundamentais (Secções 4 e 5). Concluimos o laboratório analisando a sua implementação de baixo nível, simulando RVfpgaSim no Verilator, e introduzindo exercícios avançados (Secções 6 e 7).

Utilizamos esta mesma estrutura geral nos laboratórios 7-10. Nas secções iniciais, descrevemos a especificação de alto nível do controlador E/S (as suas principais características, registos e o seu funcionamento, e o mapa da memória) e depois introduzimos exercícios fundamentais para a prática utilizando o periférico. Nas secções avançadas, descrevemos a implementação de baixo nível do controlador e fornecemos exercícios para a sua modificação e, em seguida, escrevemos programas que testam a modificação.

Nota para os instrutores: pode escolher a complexidade dos exercícios de acordo com o seu nível de curso. Por exemplo, num curso de primeiro/segundo ano (como o Computer Fundamentals ou Computer Organization), os exercícios fundamentais - neste laboratório, Secção 5 - seriam adequados. Contudo, num curso mais avançado (tal como Arquitetura de Computadores ou Projeto de Sistemas Embebidos), tanto os exercícios fundamentais como avançados - neste laboratório, as secções 5 a 7 - poderiam ser utilizados.

2. ARQUITECTURA DE ENTRADA/SAÍDA

A Figura 1 ilustra a estrutura da Arquitetura Von Neumann, que é composta por três blocos principais: a CPU, a Memória, e o Sistema I/O. Nos laboratórios 6-10, concentramo-nos na interação do CPU com dispositivos de entrada/saída (E/S). Os dispositivos de E/S são também referidos como periféricos ou simplesmente dispositivos. Aqui, apresentamos o papel de cada unidade principal:

- **CPU:** o CPU é o iniciador de todas as operações de E/S. É o controlador (historicamente chamado "mestre", mas esse termo é depreciado) de qualquer transação de E/S. Um controlador de acesso direto à memória (DMA) (DMAC) também poderia atuar como controlador, mas não está incluído neste laboratório.
- **Controlador de dispositivos:** O *controlador de dispositivos* espera por pedidos de leitura/escrita de um *controlador* para executar qualquer ação. Os controladores de dispositivos comportam-se como *periféricos* (anteriormente chamados "escravos", mas esse termo é depreciado) no sistema I/O. Conceptualmente, um controlador de

dispositivos consiste numa série de *registos* que são acessíveis a partir do *controlador*. Os valores destes registos instruem o *periférico* sobre que ação a realizar.

- **A interligação/interconnect** (barramento, crossbar, etc.) estabelece um caminho entre o *controlador* e os *periféricos*. A interligação é normalmente implementada com várias camadas ligadas através de uma *ponte* que impede que certos dispositivos atrasem todo o sistema.

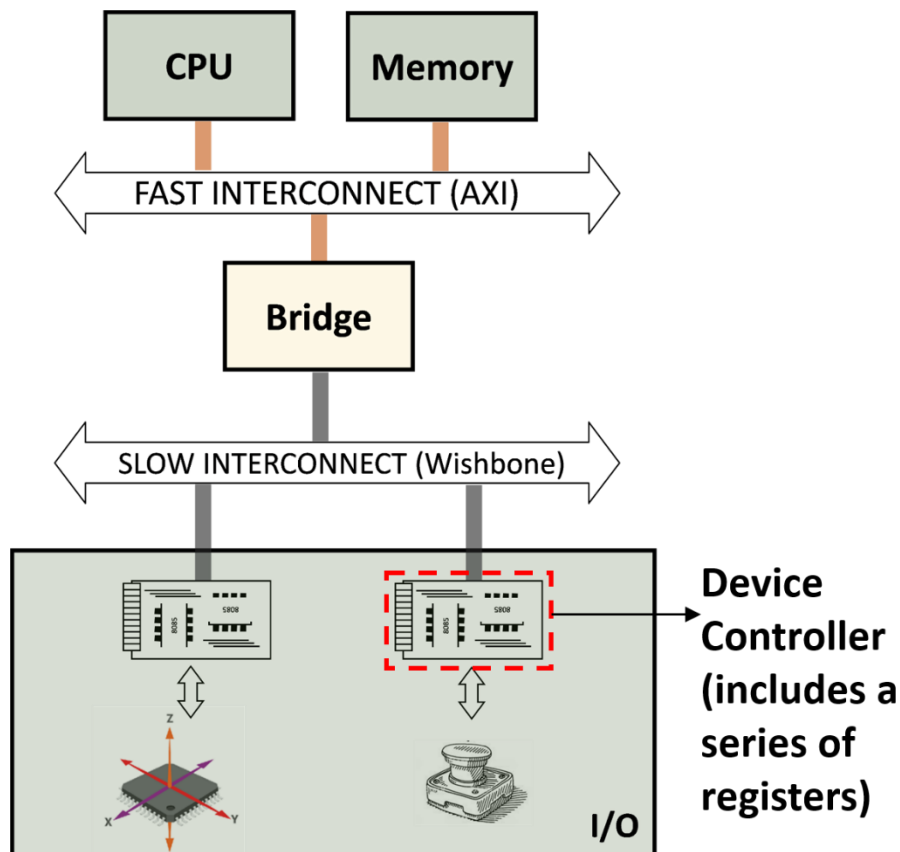


Figura 1 Sistema Computacional Genérico

A Figura 2 mostra o sub-sistema de E/S do RVfpga. Inclui os seguintes sete periféricos:

- LEDs e Interruptores (considerados um único periférico), ligados ao módulo GPIO1
- Mostradores de 7-segmentos, ligados ao módulo System Controller
- Memória Flash, ligado ao módulo SPI1
- Acelerómetro, ligado ao módulo SPI2
- Temporizador
- UART
- ROM de arranque (Boot)

Um multiplexador seleciona um periférico entre as sete possibilidades e liga-o ao CPU. Note-se que é necessário um Wishbone to AXI Bridge porque os periféricos utilizam um Wishbone bus (cor cinza) enquanto o SweRV EH1 Core utiliza uma ponte AXI (cor laranja).

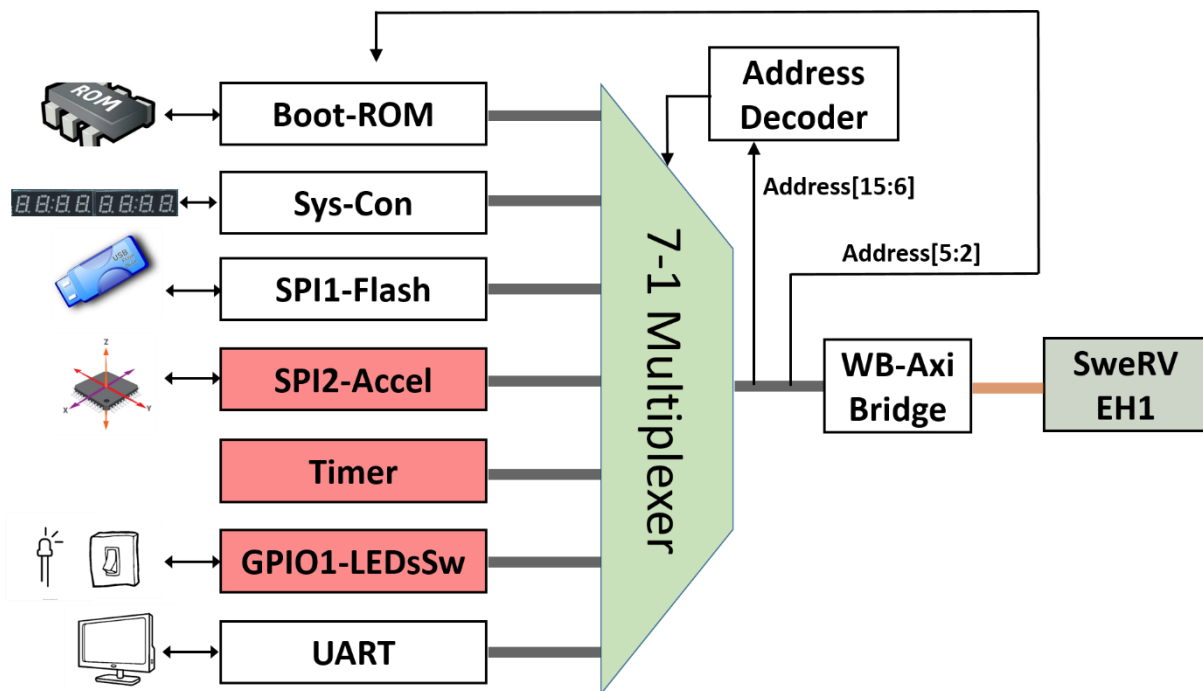


Figura 2. Sistema E/S no sistema RVfpga

TAREFA: Localize cada um dos elementos da Figura 2 no SoC. Terá de inspecionar os seguintes ficheiros e diretórios:

[RVfpgaPath]/RVfpga/src/SweRVolfSoC/swervolf_core.v (ficheiro principal, onde se encontram instanciados os elementos da Figura 2).

[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

Como descrito no Guia de Iniciação da RVfpga, o SweRVolf original (<https://github.com/chipsalliance/Cores-SweRVolf>) inclui apenas alguns dos periféricos mostrados na Figura 2: especificamente, a Boot ROM, o Controlador de Sistema (sem os mostradores de 7-Segmentos), a Memória Flash SPI e a UART (mostrada em branco em Figura 2). Lembre-se do GSG que o SweRVolfX SoC estende o SweRVolf SoC original com novos periféricos: um acelerómetro SPI, um temporizador, um módulo GPIO (mostrado a vermelho na Figura 2), e um controlador de mostrador de 7 segmentos (que estende o controlador de sistema existente da SweRVolf).

Cada periférico recebe valores do processador e/ou envia valores de volta para o processador. Os endereços de memória são reservados para valores de E/S e são chamados *registos*, *registos de E/S com memória mapeada*, ou *registos do controlador de dispositivos*. Para enviar um valor para um periférico, o CPU escreve um valor para um endereço de memória especificado (ou seja, registo com memória mapeada). Para ler um valor de um periférico, o CPU carrega um valor a partir de um endereço de memória especificado. Assim, uma simples operação *load/store* do CPU pode configurar o dispositivo, verificar o seu estado, ou ler/escrever dados nele/para ele.

O multiplexer na Figura 2 seleciona o controlador do dispositivo pretendido utilizando *Address[15:6]*. Os controladores de dispositivos utilizam *Address[5:2]* para selecionar entre vários registos utilizados para controlar o dispositivo.

3. ENTRADA/SAÍDA DE USO GERAL (GPIO)

Um controlador de E/S de uso geral (GPIO) expõe os pinos digitais externos ao programador. Em qualquer momento do programa, esses pinos podem ser configurados como entradas ou saídas. Essa designação é por pino e pode mudar ao longo do programa, se desejado. Os pinos GPIO podem ser ligados a dispositivos externos tais como LEDs, interruptores e botões de pressão.

A Figura 3 ilustra um diagrama simplificado para um módulo genérico GPIO que liga um pino externo ao CPU. O pino pode ser ligado a qualquer dispositivo de entrada/saída, tal como um LED, um interruptor, etc. O pino é ligado a um *tri-state buffer*, realçado a verde na figura. Este buffer permite ao programador configurar o pino como uma entrada ou saída. Se o *tri-state buffer* estiver ativo, o pino funciona como uma saída (por exemplo, para aceder um LED). Se o *tri-state buffer* estiver desativado, o pino atua como uma entrada (por exemplo, para a leitura dos valores de um interruptor).

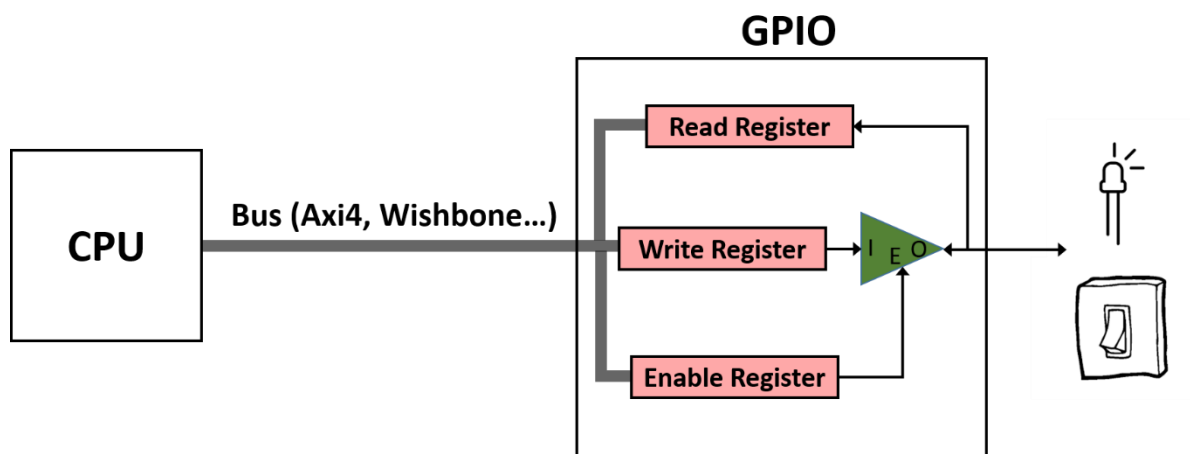


Figura 3. Circuito GPIO simplificado

Um *tri-state buffer* pode funcionar como um buffer regular (quando está ativado) ou ter uma saída flutuante (quando está desativado). O *tri-state buffer* tem duas entradas, E (ativar) e I (entrada), e uma saída, O, e a sua tabela de verdade é mostrada na Tabela 1. Quando E é 1, o *tri-state buffer* atua como um *buffer* regular, sendo a saída (O) igual à entrada (I). Quando E é 0, não existe qualquer ligação entre a entrada e a saída, logo a saída (O) não é activada, ou seja, O é flutuante. Na Figura 3, para configurar um pino como saída, E é 1, o que permite ao CPU ativar o pino. Quando um pino é configurado como uma entrada, E é 0, o que impede a CPU de ativar o pino, e permite que o periférico o ative.

Tabela 1. Tabela da verdade da porta *tri-state buffer*

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

1	1	1
---	---	---

O Sistema RVfpga utiliza E/S com memória mapeada para ler/escrever os valores armazenados nestes registos. Por exemplo, assumir que o pino da Figura 3 está ligado a um interruptor e que os três registos do GPIO são mapeados da seguinte forma:

- Registo de Leitura = Endereço 0x80001400
- Registo de Escrita = Endereço 0x80001404
- Registo de Ativação = Endereço 0x80001408

Para ler o estado do interruptor, fazemos o seguinte:

1. Configurar o pino como uma entrada, escrevendo um 0 no Registo de Ativação/*Enable* (ou seja, executando um *store* de 0 no endereço 0x80001408).
2. Leia o Registo de Leitura executando uma instrução *load* no endereço 0x80001400.

4. ESPECIFICAÇÃO DE ALTO NÍVEL DO GPIO

Nesta secção, analisamos primeiro a especificação de alto nível do GPIO da SweRVolfX e depois propomos um exercício que utiliza este periférico.

A. Especificação De Alto Nível Do GPIO

O módulo GPIO utilizado no SweRVolfX é do OpenCores (<https://opencores.org/projects/gpio>). O documento `gpio_spec.pdf` fornecido com o download do módulo GPIO do OpenCore descreve a especificação de alto nível do módulo. Está disponível aqui:

[RVfpgaPath]/RVfpga/src/SweRVolfSoC/Peripherals/gpio/docs/gpio_spec.pdf. Resumimos as funcionalidades e principais operações do módulo GPIO neste laboratório. No entanto, é possível obter as especificações completas em *gpio_spec.pdf*.

O módulo GPIO tem as seguintes características principais:

- Utiliza uma ligação Wishbone.
- Funciona apenas como dispositivo periférico.
- O utilizador pode utilizar 1-32 pinos GPIO.
- - Vários módulos GPIO (também chamados *cores* GPIO) podem ser utilizados em paralelo para aceder a mais de 32 pinos GPIO.
- - Todos os pinos GPIO podem ser:
 - bidirecional (neste caso, são necessárias células externas de E/S bi-direcionais).
 - *tri-state* ou ativação *open-drain* (neste caso, são necessárias células externas de E/S *tri-state* ou *open-drain*).
- - Pinos GPIO que são programados como entradas:
 - podem ser registados.
 - pode causar um pedido de interrupção ao CPU.

A secção 4 da especificação principal GPIO descreve os registos de controlo e estado disponíveis dentro do módulo GPIO. Cada um destes registos é atribuído a um endereço diferente, conforme indicado na Tabela 2. O endereço de base para os registos GPIO é **0x80001400**.

Tabela 2. Registos GPIO

Nome	Endereço	Largura	Acesso	Descrição
RGPIO_IN	0x80001400	1-32	R	Dados de entrada GPIO

RGPIO_OUT	0x80001404	1-32	R/W	Dados de saída GPIO
RGPIO_OE	0x80001408	1-32	R/W	Ativação do driver de saída GPIO
RGPIO_INTE	0x8000140C	1-32	R/W	Habilitação de Interrupção
RGPIO_PTRIG	0x80001410	1-32	R/W	Tipo de evento que desencadeia uma interrupção
RGPIO_AUX	0x80001414	1-32	R/W	Entradas auxiliares multiplexadas para saídas GPIO
RGPIO_CTRL	0x80001418	2	R/W	Registo de Controlo
RGPIO_INTS	0x8000141C	1-32	R/W	Estado de Interrupção
RGPIO_ECLK	0x80001420	1-32	R/W	Permitir gpio_eclk para registar RGPIO_IN
RGPIO_NEC	0x80001424	1-32	R/W	Selecionar a transição ativa de gpio_eclk

Embora o módulo GPIO do OpenCore seja mais complexo do que a versão simplificada ilustrada na Figura 3, ainda podemos identificar os três registos a partir da Figura 3: *Leitura / Read* (entrada/*input*), *Escrita / Write* (saída/*output*), e *Ativação / Enable*. No módulo GPIO da OpenCore's, estes registos são chamados, respectivamente: RGPIO_IN, RGPIO_OUT e RGPIO_OE e são mapeados para os endereços 0x80001400, 0x80001404, e 0x80001408 respectivamente.

TAREFA: Localizar a declaração de registos RGPIO_IN, RGPIO_OUT e RGPIO_OE no módulo GPIO, bem como a definição dos seus endereços. O módulo GPIO está aqui: `[RVfpgaPath]/RVfpga/src/SweRVolfSoC/Peripherals/gpio/gpio_top.v`.

O RGPIO_IN regista entradas de uso geral. O registo RGPIO_OUT ativa as saídas de uso geral. RGPIO_OE configura cada pino E/S como uma entrada ou saída. Quando o bit de *enable* (dentro do RGPIO_OE) é definido, o driver de saída de uso geral correspondente é activado, e assim o pino pode ser ligado a um periférico de saída, tal como um LED. Quando o bit de *enable* é desativado, o driver de saída passa a funcionar em modo open-drain, também chamado modo *tri-state* ou de alta impedância, e assim o pino pode ser ligado a um periférico de entrada, tal como um interruptor ou um botão de pressão.

No RVfpgaNexys, os primeiros 16 pinos GPIO, pinos 15:0, do módulo GPIO estão ligados aos 16 LEDs da placa Nexys A7. Os últimos 16 pinos GPIO, pinos 31:16, do controlador GPIO, são ligados aos 16 interruptores da placa.

5. EXERCÍCIOS BÁSICOS

Exercício 1. Escrever um programa Assembly RISC-V e um programa C que mostra um bloco de quatro LEDs acesos que se deslocam repetidamente de um lado dos 16 LEDs disponíveis no tabuleiro para o outro. Inclui também dois interruptores que controlam a velocidade e a direção. Switch[0] muda a velocidade e Switch[1] muda a direção da seguinte forma:

- Se Switch[0] estiver ligado (alto), os LEDs acesos devem mover-se rapidamente. Caso contrário, os LEDs acesos devem mover-se lentamente. Pode definir o que significa "rápido" e "lento", mas qualquer uma das velocidades deve ser visível, e deve ser capaz de detectar uma diferença na velocidade só de olhar para ela.
- Se Switch[1] estiver ligado (alto), os LEDs acesos devem mover-se repetidamente da direita para a esquerda (começam à direita quando atingem o LED mais à esquerda). Caso contrário, os LEDs acesos devem mover-se repetidamente da esquerda para a direita.

A Figura 4 abaixo mostra a placa Nexys A7 com os LEDs e interruptores destacados.

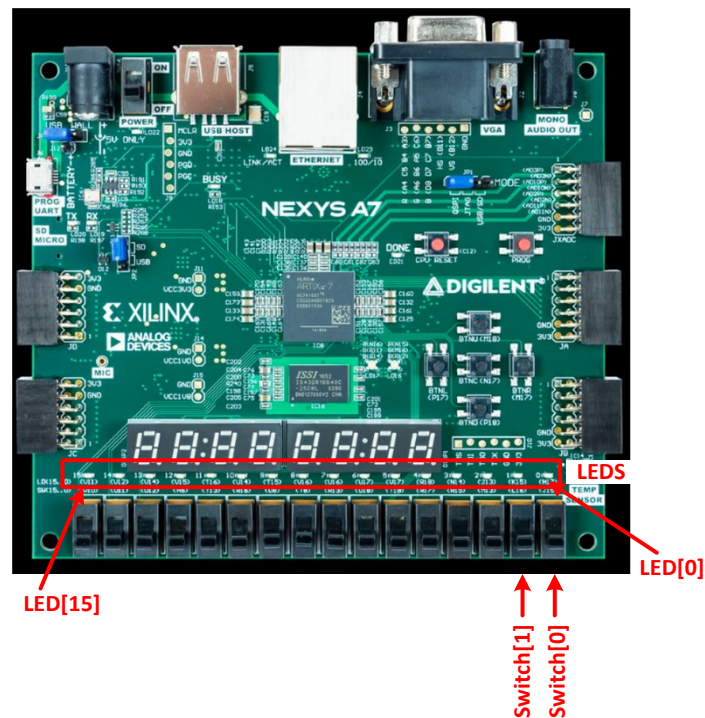


Figura 4. Placa FPGA Nexys A7: LEDs e Interruptores

Dica: Lembrar que os interruptores estão ligados aos pinos 31:16 dos registos de E/S com memória mapeada. Assim, para ler o Switch[0], é necessário escrever 0 em RGPIO_OE[16] para depois ler o valor de RGPIO_IN[16]. Terá de configurar RGPIO_OE de forma apropriada para aceder aos outros LEDs e interruptores.

6. GPIO IMPLEMENTAÇÃO DE BAIXO NÍVEL, SIMULAÇÃO

Nesta secção, descrevemos os detalhes de baixo nível do GPIO utilizado no SweRVolfX. Em seguida, modificamos o RVfpgaSim e executamos uma simulação de exemplo no Verilator para um simples exemplo de Assembly. Finalmente, propomos alguns exercícios onde irá primeiro simular RVfpgaSim, em seguida, modificá-lo para adicionar um novo periférico GPIO e finalmente escrever um programa que usa este novo periférico.

A. Implementação de baixo nível GPIO

Agora que já teve alguma experiência em aceder aos pinos GPIO usando a E/S mapeada em memória, vamos mergulhar nos detalhes de baixo nível do GPIO. O GPIO pode ser dividido em três partes principais, tal como ilustrado em Figura 5: (1) Ligação externa do RVfpgaNexys aos LEDs/Switches da placa (região sombreada esquerda na Figura 5); (2) Integração do módulo GPIO no SweRVolfX SoC (região sombreada ao centro na Figura 5); (3) Ligação entre o GPIO e o Núcleo SweRV EH1 (região sombreada à direita na Figura 5).

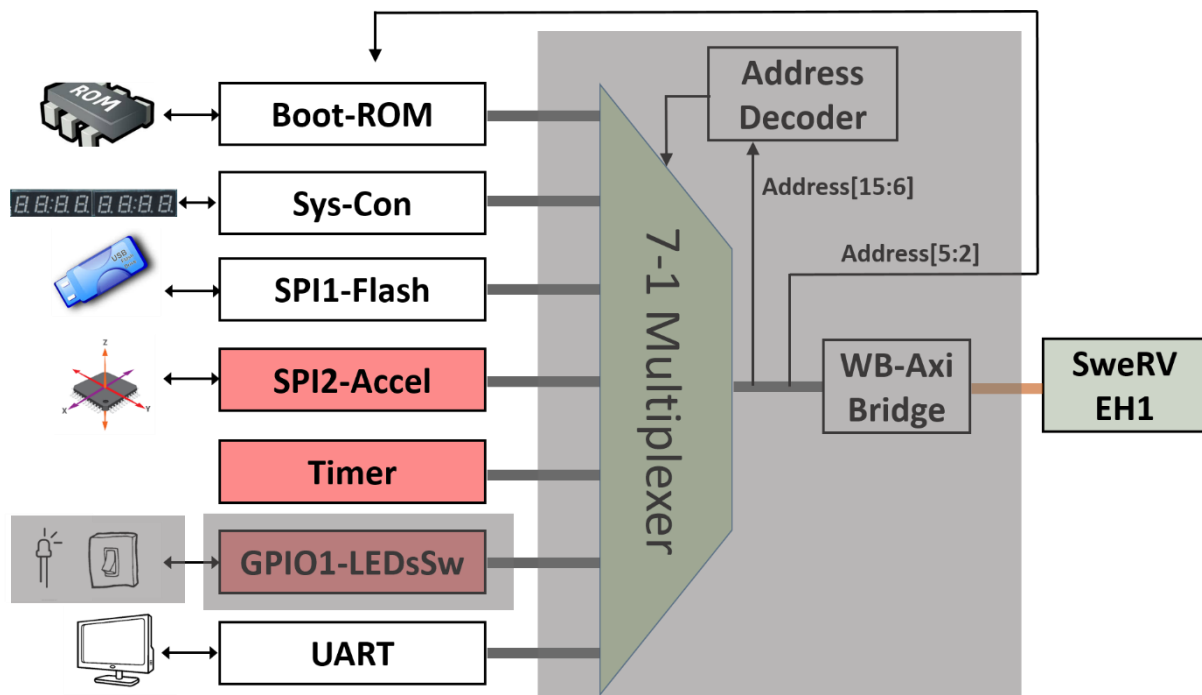


Figura 5. Análise do GPIO em 3 fases

i. Ligação dos LEDs/Switches com o SoC

O ficheiro de *constraints* do projecto (*[RVfpgaPath]/RVfpga/src/rvfpganexys.xdc*) define a ligação entre os sinais SoC de entrada/saída e os dispositivos da placa. Cada dispositivo de placa está associado a um determinado pino da FPGA. Por exemplo, Switch[0], o interruptor mais à direita na placa, é ligado através de uma pista da placa de circuito impresso (PCB) ao pino FPGA J15.

A placa Nexys A7 inclui 16 LEDs e 16 interruptores. O sinal que liga os 16 LEDs ao módulo superior do SoC (chamado *rvfpganexys*, disponível no interior do ficheiro *[RVfpgaPath]/RVfpga/src/rvfpganexys.sv*) é chamado *o_led[15:0]*, e o sinal que liga os 16 Switches com o módulo principal (top) é chamado *i_sw[15:0]*. A Figura 6 mostra a secção do ficheiro de *constraints* no projeto da Xilinx (xdc), *rvfpganexys.xdc* (disponível em *[RVfpgaPath]/RVfpga/src*) onde estas 32 ligações entre o sinal e o pino da FPGA são definidas.

```

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] }]

```

Figura 6. Ligação de *i_sw[15:0]* com os interruptores da placa e *o_led[15:0]* com os LEDs da placa (ficheiro *rvfpganexys.xdc*).

As linhas 48-49 do módulo principal (*rvfpganexys*) mostram estes dois sinais ligados ao SoC (parte esquerda da Figura 7), e o fim desse módulo mostra a sua ligação com o módulo *swervolf_core* (parte direita da Figura 7). Note que os sinais *i_sw* e *o_led* são fundidos no sinal *io_data* (linha 257), um sinal de entrada/saída de 32 bits ligado ao GPIO do módulo *swervolf_core* (como será mostrado mais tarde, na Figura 8). Além disso, note que o sinal *o_led* é registado através de um sinal intermédio, *gpio_out* (linha 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,

```

Figura 7. Ligação dos LEDs e dos Switches com o módulo principal (*rvfpganexys.sv*)

TAREFAS: Siga estes dois sinais (*i_sw* e *o_led*) desde o ficheiro de *constraints* ao módulo SoC SweRVolf (onde são fundidos em *io_data*). Terá de inspecionar os seguintes ficheiros:

```
[RVfpgaPath]/RVfpga/src/rvfpganexys.xdc
[RVfpgaPath]/RVfpga/src/rvfpganexys.sv
[RVfpgaPath]/RVfpga/src/SweRVolfSoC/swervolf_core.v
```

Na secção anterior dissemos que no RVfpgaNexys os 16 primeiros pinos GPIO (15 a 0) do módulo GPIO são ligados aos 16 LEDs da placa, enquanto os 16 últimos pinos GPIO (31 a 16) do controlador GPIO são ligados aos 16 interruptores da placa. Isto corresponde à implementação descrita nesta secção e em Figura 8?

ii. Integração do módulo GPIO no SoC

nas linhas 299-354 do módulo **swervolf core**

([RVfpgaPath]/RVfpga/src/SweRVolfSoC/swervolf_core.v), o módulo GPIO é instanciado e integrado no SoC (ver Figura 8).

```

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

```

Figura 8. Integração do módulo GPIO (ficheiro *swervolf_core.v*).

A interface do módulo pode ser dividida em dois blocos: Sinais Wishbone (Tabela 3), que permitem ao SweRV EH1 Core comunicar com a GPIO usando um modelo de controlador/periférico, e sinais externos de E/S (Tabela 4).

Tabela 3. Sinais Wishbone

Porto	Largura	Direção	Descrição
wb_cyc_i	1	Entradas	Indica um ciclo de barramento válido (seleção do núcleo)
wb_adr_i	15	Entradas	Entradas de endereços
wb_dat_i	32	Entradas	Entradas de dados
wb_dat_o	32	Saídas	Saídas de dados
wb_sel_i	4	Entradas	Indica bytes válidos no barramento de dados (durante o ciclo válido deve ser 0xf)
wb_ack_o	1	Saída	Saída de confirmação (indica o término normal da transação)

wb_err_o	1	Saída	Saída de aviso de erro (indica um cancelamento anormal da transacção)
wb_rty_o	1	Saída	Não usado
wb_we_i	1	Entrada	Transacção de escrita quando for nível alto
wb_stb_i	1	Entrada	Indica ciclo de transferência de dados válido
wb_inta_o	1	Saída	Saída de interrupção

Tabela 4. Sinais externos de E/S

Porto	Largura	Direção	Descrição
in_pad_i	1-32	Entradas	Entradas GPIO
out_pad_o	1-32	Saídas	Saídas GPIO
oen_padoen_o	1-32	Saídas	Ativadores das saídas GPIO (para <i>drivers tri-state</i> ou <i>open-drain</i>)

Como mostrado na linha 342 da Figura 8, bits 5:2 do endereço fornecido pelo *core* no sinal do barramento Wishbone *wb_m2s_gpio_adr[5:2]* são utilizados para selecionar um entre os 10 registos mapeados em memória disponíveis. Estes quatro bits são fornecidos ao *core* GPIO através do sinal *wb_adr_i* (também mostrado na Figura 8).

A entrada *ext_pad_i* liga diretamente com o GPIO Read Register (RGPIO_IN). Da mesma forma, a saída *ext_pad_o* liga diretamente com o Registo de Escrita GPIO (RGPIO_OUT). Estes dois sinais estão ligados aos LEDs e Switches (*i_gpio*, *o_gpio*, *io_data*) através de 32 módulos *tri-state buffer* (Figura 8, linhas 305-336). Desta forma, todos os 32 pinos podem ser configurados como entradas ou saídas. No nosso caso, os 16 pinos inferiores, pinos 15:0, estão ligados aos LEDs (Figura 7) e, portanto, devem ser configurados como saídas; os 16 pinos superiores, 31:16, estão ligados aos interruptores (Figura 7) e, portanto, devem ser configurados como entradas. Implementamos estes 32 *tri-state buffer*, incluindo o seguinte módulo no final do módulo **swervolf_core** (linhas 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
```

TAREFAS: Os pinos GPIO (*io_data*) estão ligados ao módulo GPIO através de *tri-state buffer* (ver Figura 8). Analisar o *tri-state buffer* para os dois estados possíveis do sinal de ativação/*enable* (*oe*=0 and *oe*=1).

Tendo em conta a ligação entre o módulo GPIO e os LEDs/Switches da placa, quais os valores a que o programador atribui a *en_gpio*?

iii. Ligação entre o GPIO e o Core SweRV EH1

Como mostrado na Figura 2, os controladores do dispositivo estão ligados ao Núcleo SweRV EH1 através de um multiplexer e de uma ponte. O multiplexer seleciona um entre os N periféricos possíveis (no nosso caso, N=7), dependendo do endereço gerado pelo CPU. A ponte traduz os sinais Wishbone utilizados pelos controladores do dispositivo para os sinais AXI4 utilizados pelo Núcleo SweRV e vice-versa (implementados no ficheiro *[RVfpgaPath]/RVfpga/src/SweRVolfSoC/Interconnect/AxiToWb/axi2wb.v*).

O multiplexer 7:1 (Figura 9) é implementado no ficheiro `[RVfpgaPath]/RVfpga/src/SweRVolfSoC/Interconnect/WishboneInterconnect/wb_intercon.v`, que é instanciado nas linhas 104-205 do ficheiro `[RVfpgaPath]/RVfpga/src/SweRVolfSoC/Interconnect/WishboneInterconnect/wb_intercon.vh`. Este último ficheiro está incluído na linha 168 do módulo **swervolf_core** localizado aqui: `[RVfpgaPath]/RVfpga/src/SweRVolfSoC/swervolf_core.v`.

```

108 wb_mux
109 #(.num_slaves (7),
110 .MATCH_ADDR ({32'h00000000, 32'h00001000, 32'h00001040, 32'h00001100, 32'h00001200, 32'h00001400, 32'h00002000}),
111 .MATCH_MASK ({32'hfffff000, 32'hfffffc0, 32'hfffffc0, 32'hfffffc0, 32'hfffffc0, 32'hfffffc0, 32'hffff000}))
112 wb_mux_io
113 (.wb_clk_i (wb_clk_i),
114  .wb_rst_i (wb_rst_i),
115  .wbm_addr_i (wb_io_addr_i),
116  .wbm_dat_i (wb_io_dat_i),
117  .wbm_sel_i (wb_io_sel_i),
118  .wbm_we_i (wb_io_we_i),
119  .wbm_cyc_i (wb_io_cyc_i),
120  .wbm_stb_i (wb_io_stb_i),
121  .wbm_cti_i (wb_io_cti_i),
122  .wbm_bte_i (wb_io_bte_i),
123  .wbm_dat_o (wb_io_dat_o),
124  .wbm_ack_o (wb_io_ack_o),
125  .wbm_err_o (wb_io_err_o),
126  .wbm_rty_o (wb_io_rty_o),
127  .wbs_addr_o ({wb_rom_addr_o, wb_sys_addr_o, wb_spi_flash_addr_o, wb_spi_accel_addr_o, wb_ptc_addr_o, wb_gpio_addr_o, wb_uart_addr_o}),
128  .wbs_dat_o ({wb_rom_dat_o, wb_sys_dat_o, wb_spi_flash_dat_o, wb_spi_accel_dat_o, wb_ptc_dat_o, wb_gpio_dat_o, wb_uart_dat_o}),
129  .wbs_sel_o ({wb_rom_sel_o, wb_sys_sel_o, wb_spi_flash_sel_o, wb_spi_accel_sel_o, wb_ptc_sel_o, wb_gpio_sel_o, wb_uart_sel_o}),
130  .wbs_we_o ({wb_rom_we_o, wb_sys_we_o, wb_spi_flash_we_o, wb_spi_accel_we_o, wb_ptc_we_o, wb_gpio_we_o, wb_uart_we_o}),
131  .wbs_cyc_o ({wb_rom_cyc_o, wb_sys_cyc_o, wb_spi_flash_cyc_o, wb_spi_accel_cyc_o, wb_ptc_cyc_o, wb_gpio_cyc_o, wb_uart_cyc_o}),
132  .wbs_stb_o ({wb_rom_stb_o, wb_sys_stb_o, wb_spi_flash_stb_o, wb_spi_accel_stb_o, wb_ptc_stb_o, wb_gpio_stb_o, wb_uart_stb_o}),
133  .wbs_cti_o ({wb_rom_cti_o, wb_sys_cti_o, wb_spi_flash_cti_o, wb_spi_accel_cti_o, wb_ptc_cti_o, wb_gpio_cti_o, wb_uart_cti_o}),
134  .wbs_bte_o ({wb_rom_bte_o, wb_sys_bte_o, wb_spi_flash_bte_o, wb_spi_accel_bte_o, wb_ptc_bte_o, wb_gpio_bte_o, wb_uart_bte_o}),
135  .wbs_dat_i ({wb_rom_dat_i, wb_sys_dat_i, wb_spi_flash_dat_i, wb_spi_accel_dat_i, wb_ptc_dat_i, wb_gpio_dat_i, wb_uart_dat_i}),
136  .wbs_ack_i ({wb_rom_ack_i, wb_sys_ack_i, wb_spi_flash_ack_i, wb_spi_accel_ack_i, wb_ptc_ack_i, wb_gpio_ack_i, wb_uart_ack_i}),
137  .wbs_err_i ({wb_rom_err_i, wb_sys_err_i, wb_spi_flash_err_i, wb_spi_accel_err_i, wb_ptc_err_i, wb_gpio_err_i, wb_uart_err_i}),
138  .wbs_rty_i ({wb_rom_rty_i, wb_sys_rty_i, wb_spi_flash_rty_i, wb_spi_accel_rty_i, wb_ptc_rty_i, wb_gpio_rty_i, wb_uart_rty_i}));
139
140 endmodule

```

CPU/Controller Signals

Peripheral Signals

Figura 9. Multiplexer 7-1 seleciona o periférico para se ligar ao CPU (`wb_intercon.v`).

O multiplexer seleciona qual periférico para ler ou escrever, ligando o CPU (sinais `wb_io_*` – linhas 115-126 da Figura 9) com o barramento Wishbone de um periférico (linhas 127-138 da Figura 9), dependendo do endereço (linhas 110-111). Por exemplo, se o endereço gerado pelo CPU estiver no intervalo 0x80001400-0x8000143F, o gpio periférico é selecionado, e assim os sinais `wb_io_*` estarão ligados com sinais `wb_gpio_*`.

A Figura 10 mostra a implementação Verilog do multiplexer (disponível no ficheiro `[RVfpgaPath]/RVfpga/src/SweRVolfSoC/Interconnect/WishboneInterconnect/wb_intercon_1.2.2-r1/wb_mux.v`).

TAREFA: Analise em detalhe a implementação do multiplexer. Pode focar-se nos sinais relacionados com o GPIO (`wb_gpio_*`). Terá de inspecionar os seguintes ficheiros:

`[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`

Compreender esta parte do SoC é importante não só para este laboratório, mas também para laboratórios futuros. A simulação realizada na secção seguinte pode ajudá-lo a compreendê-la se prolongar a simulação adicionando novos sinais relacionados com o 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] ffl;
105   input [num_slaves-1:0] in;
106   integer i;
107
108   begin
109     ffl = 0;
110     for (i = num_slaves-1; i >= 0; i=i-1) begin
111       if (in[i])
112         ffl = i;
113     end
114   end
115 endfunction
116
117 assign slave_sel = ffl(match);
118
119 always @(posedge wb_clk_i)
120   wbm_err <= wbm_cyc_i & !(match);
121
122 assign wbs_adr_o = {num_slaves{wbm_adr_i}};
123 assign wbs_dat_o = {num_slaves{wbm_dat_i}};
124 assign wbs_sel_o = {num_slaves{wbm_sel_i}};
125 assign wbs_we_o = {num_slaves{wbm_we_i}};
126
127 /* verilator lint_off WIDTH */
128 assign wbs_cyc_o = match & (wbm_cyc_i << slave_sel);
129 /* verilator lint_on WIDTH */
130 assign wbs_stb_o = {num_slaves{wbm_stb_i}};
131
132 assign wbs_cti_o = {num_slaves{wbm_cti_i}};
133 assign wbs_bte_o = {num_slaves{wbm_bte_i}};
134
135 assign wbm_dat_o = wbs_dat_i[slave_sel*dw+:dw];
136 assign wbm_ack_o = wbs_ack_i[slave_sel];
137 assign wbm_err_o = wbs_err_i[slave_sel] | wbm_err;
138 assign wbm_rty_o = wbs_rty_i[slave_sel];
139
140 endmodule

```

Figura 10. Multiplexer Wishbone (ficheiro *wb_mux.v*).

B. Simulação Com o Verilator

Nesta secção, modificamos primeiro o simulador RVfpgaSim adicionando um novo sinal de entrada. Em seguida, recompilamos o RVfpgaSim usando o Verilator e analisamos este novo sinal quando o simulador executa um programa simples.

i. Modificar e Recompilar RVfpgaSim

Em simulação, não temos LEDs ou interruptores reais. Assim, no *testbench* (*[RVfpgaPath]/RVfpga/src/rvfpgasim.v*), simulamos o acionamento dos interruptores atribuindo ao sinal (*i_sw*) um valor constante de 0xFE34 (parte esquerda da Figura 11). Os interruptores são então fornecidos como uma entrada para o SoC SweRVolfX (parte direita da Figura 11).

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

Figura 11. O sinal `i_sw` é atribuído e passado para o SweRVolfX SoC no `rvfpgasim.v`.

Lembre-se do Guia de Introdução que o *testbench* (`rvfpgasim.v`) recebe os sinais de entrada (`clk`, `rst`, etc.) para o RVfpgaSim (parte esquerda da Figura 12) e instancia o módulo `swervolf_core` (parte direita da Figura 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),
```

Figura 12. Sinais de entrada para instanciação no RVfpgaSim e SweRVolfX (ficheiro `rvfpgasim.v`).

Nalgumas situações, poderá querer adicionar um novo sinal de entrada/saída ao simulador. Como exemplo, explicamos a seguir como se pode incluir um sinal de entrada ao RVfpgaSim, chamado `i_sw0`, que fornece um valor para o interruptor mais à direita.

Siga os próximos passos:

1. Modificar o ficheiro `[RVfpgaPath]/RVfpga/src/rvfpgasim.v`:
 - a. Inclua um novo sinal de entrada de 1 bit chamado `i_sw0`. Ver Figura 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)
```

Figura 13. Novo sinal de entrada `i_sw0`.

- b. Fornecer este sinal como o interruptor mais à direita. Atribuir os restantes valores do interruptor como 0xFE34 – exceto para o bit 0 – como antes). Ver Figura 14.

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

Figura 14. `i_sw0` é o interruptor mais à direita.

2. Modificar o ficheiro `[RVfpgaPath]/RVfpga/verilatorSIM/tb.cpp`: este é o ficheiro principal C++ para o Verilator. No final deste ficheiro, pode encontrar um ciclo `while` (mostrado na Figura 15) em que cada iteração constitui um pulso de relógio. Note-se que o sinal do relógio para o SoC é gerado dentro deste ciclo (linha 175), invertendo o seu valor binário em cada iteração ($1 \rightarrow 0$ ou $0 \rightarrow 1$). Além disso, o tempo de simulação é calculado na variável `main_time` (linha 176) e mede-se em nanossegundos (o ciclo do relógio é de 20 ns e, portanto, o pulso de relógio é de 10 ns). Finalmente, note-se que a simulação termina quando o tempo de simulação atinge o valor `timeout` (linhas 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 }

```

Figura 15. Ciclo While para a simulação.

Atribuir um valor binário de **0** para o novo sinal de `i_sw0` antes de entrar no ciclo (parte esquerda da Figura 16), e mudá-lo para **1** no momento **30 us** dentro do ciclo (parte direita de Figura 16).

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

Figura 16. Atribuir valor ao sinal `i_sw0`.

3. Depois de ter efetuado todas estas alterações, recompila o RVfpgaSim executando os seguintes comandos (isto foi explicado no GSG):

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

Um novo file *Vrvfpgasim* (o binário de simulação RVfpgaSim), deve ser gerado dentro do diretório *[RVfpgaPath]/RVfpga/verilatorSIM*.

WINDOWS: Tem de fazer este último passo (passo 4) dentro do terminal Cygwin (consulte a Secção 6 e o Apêndice C no Guia de Introdução para obter as instruções detalhadas). Note que a pasta C: Windows pode ser encontrada dentro de Cygwin em: */cygdrive/c*.

MacOS: Consulte o Apêndice D do Guia de Introdução para obter as instruções detalhadas.

ii. Analisar a simulação do programa *LedsSwitches.S*

Nesta secção, simulamos o programa de exemplo *LedsSwitches.S* (Figura 17) a partir do Guia de Introdução RVfpga. Este programa lê os valores nos interruptores e escreve esse valor para os LEDs na placa Nexys A7. Note que precisamos de configurar o registo de ativação, de modo que os 32 pinos de entrada/saída sejam configurados como entradas ou saídas, de acordo com as suas ligações. Especificamente, os 16 pinos inferiores do GPIO estão ligados aos LEDs, pelo que são pinos de saída em relação ao CPU (Enable=1). Os 16 pinos superiores do GPIO estão ligados aos interruptores, que são pinos de entrada em relação ao CPU (Ativar=0). Como os interruptores ocupam os 16 bits superiores do registo de leitura, devem ser deslocados para a direita antes de escreverem o seu valor para os 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)           # Escrever o registo de ativação

next:
    li a1, GPIO_SWs      # Ler os interruptores
    lw t0, 0(a1)

    li a0, GPIO_LEDs
    srl t0, t0, 16
    sw t0, 0(a0)         # Escrever nos LEDs

    beq zero, zero, next
.end
```

Figura 17. LedsSwitches.s para correr no SweRVolfX SoC

Siga os próximos passos para executar a simulação.

1. Abra o VSCode/PlatformIO no seu computador.
2. Na barra superior, clique em *File→Open Folder...* (Figura 18), e navegue para o diretório *[RVfpgaPath]/RVfpga/examples/*

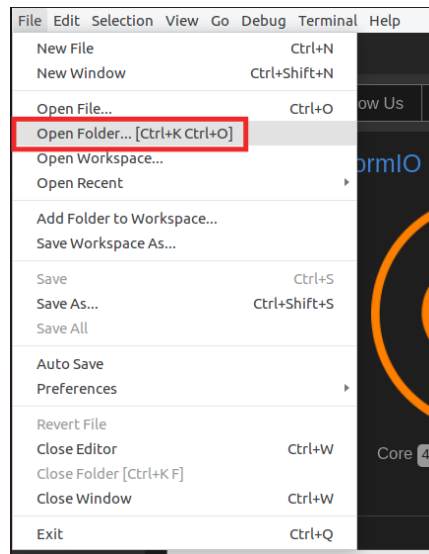


Figura 18. Abra o exemplo do LedsSwitches.S

3. Selecione diretório *LedsSwitches* (não abra, mas apenas selecione-o) e clique em OK. O exemplo será aberto no PlatformIO.
4. Abra o ficheiro *platformio.ini* e verifique se o caminho para o binário de simulação RVfpgaSim (Figura 19) gerado acima (passo 3 na secção anterior) está correto. Lembre-se do GSG que deve parecer:

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

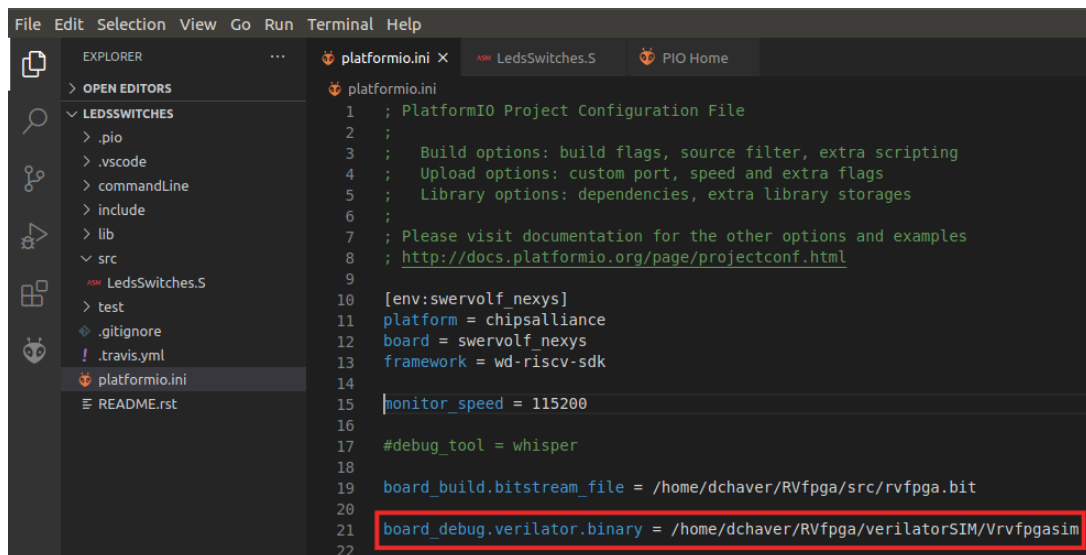



Figura 19. Ficheiro de inicialização do PlatformIO: platformio.ini

Windows: O executável de simulação RVfpgaSim chama-se *Vrvfpgasim.exe*. Assim:

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

5. Executar a simulação clicando no ícone PlatformIO na faixa do menu esquerdo , em seguida, expandir As Tarefas do Projeto → env:swervolf_nexys → Plataforma e clicar em Gerar Traço.

O ficheiro *trace.vcd* deveria ter sido gerado dentro *[RVfpgaPath]/RVfpga/examples/LedsSwitches/.pio/build/swervolf_nexys*, e pode abri-lo com *GTKWave* escrevendo o seguinte comando no terminal PlatformIO.

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

WINDOWS: a pasta *gtkwave64* que descarregou, inclui uma aplicação chamada *gtkwave.exe* dentro da pasta *bin*. Lance GTKWave clicando duas vezes nessa aplicação. Na parte superior da aplicação, clique em **Ficheiro – Abra o Separador Novo** e abra o *traço.vcd* ficheiro gerado na pasta *[RVfpgaPath]/RVfpga/examples/LedsSwitches/.pio/build/swervolf_nexys*.

6. Incluir no traço de simulação os seguintes sinais (ir para o módulo *rvfpgasim-swervolf* para encontrar cada um destes sinais):
 - Adicione o sinal do relógio: **clk**
 - Adicione o sinal de entrada GPIO : **i_gpio**
 - Adicione o sinal de saída GPIO: **o_gpio**

No gráfico (Figura 20), verá que o valor dos 16 interruptores (16 bits de sinal mais significativos **i_gpio**) é copiado para os 16 LEDs (16 bits de sinal menos significativos **o_gpio**) com algum atraso. Além disso, o interruptor mais à direita muda (01) no tempo →30us, o que faz com que o LED mais à direita também mude algum tempo depois.

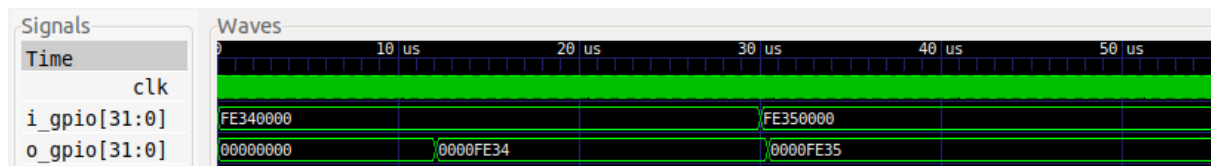


Figura 20. Simulação do programa LedsSwitches

7. EXERCÍCIOS AVANÇADOS

Exercício 2. Analisar a simulação da secção anterior com mais detalhe. A Figura 21 mostra a versão “desassemblada” do programa .elf *LedsSwitches* (Figura 17), com as três instruções que acedem aos registos GPIO (Ativar/Enable, Ler e Escrever) destacadas. Lembre-se do Guia de Introdução que pode facilmente visualizar no PlatformIO a versão de desmontagem do programa .elf abrindo o ficheiro *firmware.dis*, que foi gerado no momento da compilação dentro da pasta:

[RVfpgaPath]/RVfpga/examples/LedsSwitches/.pio/build/swervolf-nexys/ (ver Figura 21).

```

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

```

```

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

```

Figura 21. Desassemblagem do programa LedsSwitches.S

Simular este programa em RVfpgaSim e analisar os sinais GPIO durante a execução de cada uma das três instruções de memória destacadas a vermelho na Figura 21 (sw, lw e sw). Isto irá ajudá-lo a entender a implementação de baixo nível do GPIO explicada na Secção A.

Pode partir da simulação da Secção B e adicionar e analisar os valores dos seguintes sinais (vá dentro dos módulos referidos para localizar cada sinal):

- rvfpgasim → swervolf → swerv_eh1 → swerv → ifu
 - Relógio: **clk**.
 - Instruções carregadas: **ifu_i0_instr** e **ifu_i1_instr**.
- rvfpgasim – swervolf
 - Pinos de entrada/saída 32-bit: **i_gpio** e **o_gpio**.
 - Endereço fornecido pelo CPU: **wb_m2s_io_adr**.
- rvfpgasim – swervolf – gpio_module
 - Interface Externa GPIO: **ext_pad_i**, **ext_pad_o** e **ext_padoe_o**.
- rvfpgasim – swervolf – wb_intercon0
 - Endereço de saída e sinais de dados para o multiplexer da Figura 2: **wb_io_adr_i**, **wb_io_dat_i**, **wb_io_dat_o**.
 - Sinais de dados GPIO de entrada para o multiplexer da Figura 2: **wb_gpio_adr_i**, **wb_gpio_dat_i**, **wb_gpio_dat_o**.
 - Sinais de seleção para o multiplexer da Figura 2: **wb_*_cyc_o**.
- rvfpgasim – swervolf – wb_intercon0 – wb_mux_io
 - Sinal de correspondência para o multiplexer da Figura 2: **match**.
- rvfpgasim – swervolf – swerv_eh1 – swerv – dec – arf – gpr_banks(0) – gpr(5) – gprff
 - Valor de registo para t0: **dout**.

Exercício 3. Expandir o **RVfpgaNexys** para suportar os cinco botões de pressão incorporados na placa. Os botões de pressão são mostrados na Figura 22. Os cinco botões são nomeados de acordo com a sua localização: cima, baixo, esquerda, direita e centro - BTNU, BTND, BTNL, BTNR, BTNC.

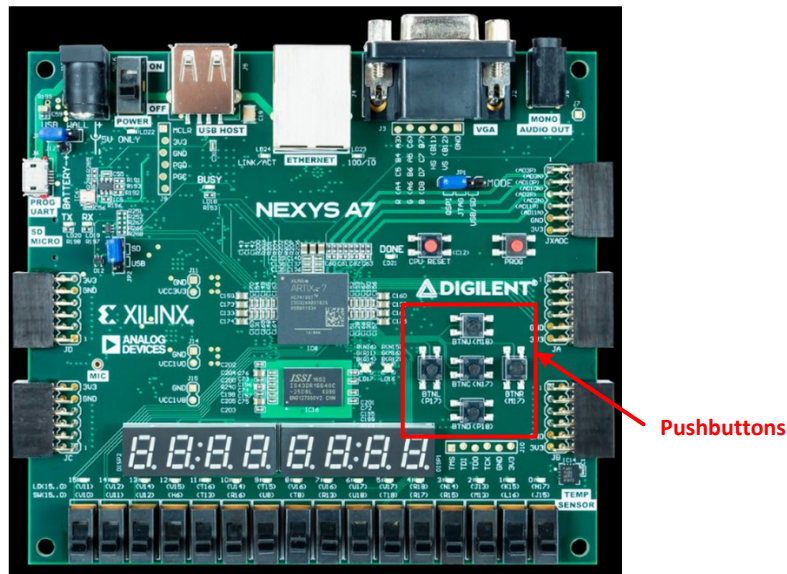


Figura 22. Botões de pressão na placa Nexys A7 FPGA

- a. Dado que o tamanho máximo do módulo GPIO que estamos a utilizar (`gpio_top`) ser 32, que é o número de pinos de E/S que temos (16 LEDs + 16 Interruptores), é necessário incluir outra instanciação do módulo GPIO no SwERVolfX, bem como 5 novos *buffers tri-state* e todos os sinais necessários.
- b. Utilizar os endereços a partir de 0x80001800 (que estão disponíveis) para mapear os registos expostos pelo novo controlador GPIO. Note-se que deve modificar o multiplexador (Figura 9) por incluir o novo periférico.
- c. Deve também modificar o ficheiro *constraints* tendo em conta que os cinco botões de pressão estão ligados aos seguintes pinos da FPGA:
 - i. BTNC está ligado a PIN N17
 - ii. BTNU está ligado a PIN M18
 - iii. BTNL está ligado a PIN P17
 - iv. BTNR está ligado a PIN M17
 - v. BTND está ligado a PIN P18

Exercício 4. Criar outro controlador na **RVfpgaNexys** para os cinco botões de pressão da placa.

- a. Em contraste com o Exercício 3, neste caso deve implementar o seu próprio controlador GPIO em Verilog ou SystemVerilog com base no esquema ilustrado na Figura 3. Na verdade, pode até simplificar esse circuito e incluir apenas um registo de leitura, ou **Read Register** (i.e. não é necessário incluir os *buffers tri-state* nem os registos de escrita ou **Write Register**).
- b. Não é necessário retirar o controlador do exercício anterior porque os botões podem ser mapeados para endereços não utilizados por esse controlador GPIO.
- c. Incluir o novo controlador dentro do periférico System Controller. Pode utilizar a gama de endereços 0x8000101C-0x8000101F, que não é utilizada. Note que os registos incluídos no Controlador do Sistema são lidos na CPU ligando-os diretamente ao sinal de dados do barramento Wishbone (o_wb_rdt) com base no endereço (i_wb_adr) gerado pelo CPU. Inspeccionar as linhas 234-266 do módulo **swervolf_syscon** (*[RVfpgaPath]/RVfpga/src/SweRVolfSoC/Peripherals/SystemController/swervolf_syscon.v*) para o ajudar a compreender como proceder.

Exercício 5. Escrever um programa Assembly RISC-V e um programa C que exibe uma contagem binária crescente nos LEDs, começando em 1. Incluir um ciclo vazio para espera entre a exibição de cada valor incrementado para que os valores sejam visíveis pelo olho humano. Ler o BTNC através do periférico da OpenCores implementado no Exercício 3 e usá-lo para alterar a velocidade da contagem, e ler BTNU através do periférico ad-hoc implementado no Exercício 4 e usá-lo para reiniciar a contagem sempre que for pressionado.