THE IMAGINATION UNIVERSITY PROGRAMME

RVfpga Lab 7
7-Segment Displays

1. INTRODUCTION

This lab describes how the RVfpga System was extended to work with 7-segment displays and shows how to modify the 7-segment display controller. The Nexys A7 FPGA board has eight 7-segment displays. We first describe how they work (Section 2) and then analyse the high-level specification of the 8-digit 7-segment display controller included in the RVfpga System and provide some fundamental exercises (Sections 3 and 4). Finally, we analyse the low-level implementation of this controller, perform a Verilator simulation and provide additional exercises where you will modify and experiment with the controller implementation (Sections 5 and 6).

2. 7-SEGMENT DISPLAYS ON THE NEXYS A7 BOARD

The Nexys A7 board contains two 4-digit common-anode 7-segment LED displays[1], configured to behave as a single 8-digit 7-segment display (see Figure 1). Each of the eight digits is composed of seven segments arranged in a "figure 8" pattern (see Figure 2), with an LED for each segment. Each of these segments can be switched on or off, so any one of 128 patterns can be displayed on a digit by illuminating certain LED segments and leaving the others dark; specifically, among these 128 patterns, the decimal digits can be displayed as shown in Figure 2.

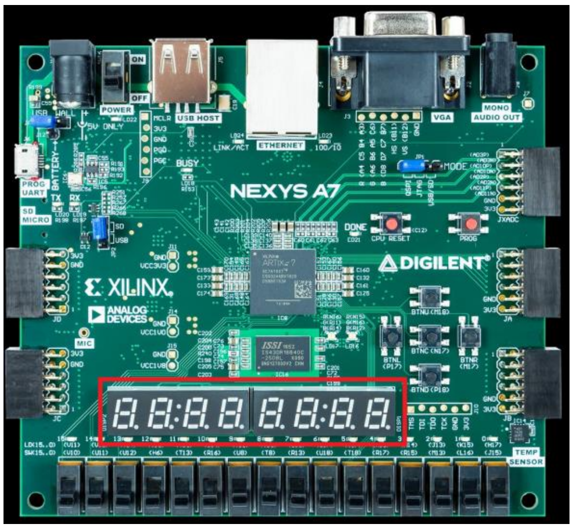


Figure 1. 8-digit 7-segment displays on the Nexys A7

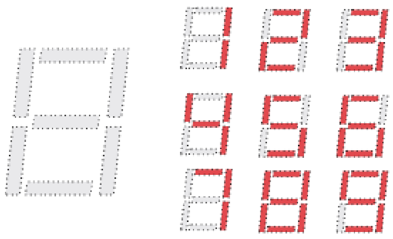


Figure 2. Patterns corresponding to decimal digits

[1] The information in this section is described in: https://reference.digilentinc.com/reference/programmable-logic/nexys-a7/reference-manual

The LED segments of a single digit are labelled *A*-*G*, as shown on the right of Figure 3. The anodes of the seven LEDs for a single digit are tied together into one “common anode” circuit node, but the LED cathodes remain separate (see Figure 3). The eight common anode signals, one for each digit (*AN0*-*AN7*), act as a “digit enable”. The cathodes of the same segment on all eight digits are connected into seven signals, *CA*-*CG* (see Figure 3). (Note that an eighth signal exists for the decimal point, *DP*, but we will not use it in this lab.) For example, the cathode of segment *D* from the eight digits are grouped together into a single circuit node called *CD*. This signal connection scheme creates a multiplexed display, where the cathode signals are common to all digits, but they can only illuminate the segments of the digit whose corresponding anode signal is asserted. All these signals are driven low when active; thus, to illuminate a segment, for example, segment *D* on digit *2*, both the anode *AN2* and the cathode *CD* must be driven low.

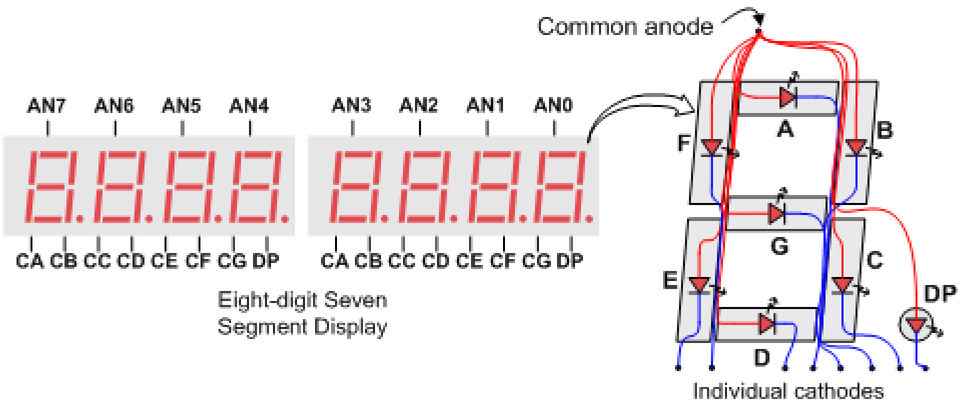


Figure 3. Connection of the 8-digit 7-segment Display on the Nexys A7

A scanning display controller circuit can be used to show an 8-digit number on the 8-digit 7-segment displays. This circuit drives the cathodes with the pattern of each digit in a repeating continuous succession at an update rate that is faster than the human eye can detect; at the same time the circuit drives the anodes one at a time. Thus, each digit is illuminated just one-eighth of the time, but, because the eye cannot perceive the darkening of a digit before it is illuminated again, the digit appears to be continuously illuminated.

For each of the 8 digits to appear bright and continuously illuminated, all eight digits should be driven once every 1 to 16 ms, and each digit would be illuminated for 1/8 of the refresh cycle (e.g., for a 16ms refresh cycle, each digit is illuminated for 2ms). As explained above, the controller must drive the cathodes of a digit low with the correct pattern while the corresponding anode signal is also driven low. However, since the Nexys A7 uses NPN transistors to drive enough current into the common anode point, the anode enables are inverted. Therefore, both the AN0...7 and the CA...G/DP signals are driven low when active.

To illustrate the process, assume that you want to show *71* on the two right-most digits. The controller circuit would drive *AN0*, *CB*, and *CC* low for the first 2ms, thus showing a *1* in the right-most digit. Then, for the next 2ms, the circuit would drive *AN1*, *CA*, *CB*, and *CC* low, thus showing a *7* in the next most significant digit. If the process is repeated indefinitely, the human eye will see number *71* in the two right-most digits.

3. HIGH-LEVEL SPECIFICATION OF THE 8-DIGIT 7-SEGMENT DISPLAY CONTROLLER

In this section, we first describe and analyse the high-level specification of the 8-digit 7-segment displays controller used in the RVfpga System, and then we provide exercises for using it.

A. High-level specification

The 8-digit 7-segment display controller used in this course has been custom-designed for the RVfpga System. It includes two registers, called *Enables_Reg* and *Digits_Reg*, that are mapped to addresses 0x80001038 and 0x8000103C respectively (note that these addresses are unused addresses within the address range reserved for the System Controller, which you can view at https://github.com/chipsalliance/Cores-SweRVolf).

TASK: Locate the declaration of registers *Enables_Reg* and *Digits_Reg*, as well as the place where they are assigned a value. The 8-digit 7-segment displays is implemented in file: *[RVfpgaPath]/RVfpga/src/SweRVolfSoC/Peripherals/SystemController/swervolf_syscon.v*.

Enables_Reg is an 8-bit register where each bit determines if the corresponding digit is *ON* (0) or *OFF* (1). *Digits_Reg* is a 32-bit register where each 4-bit group represents the hexadecimal value to show in the corresponding digit. For example, to show *71* on the two right-most digits, the programmer would assign the following values to the registers:

- *Enables_Reg = 0xFC* (two right-most digits enabled)
- *Digits_Reg = 0x00000071* (value = 71)

4. FUNDAMENTAL EXERCISES

Exercise 1. Write a RISC-V assembly program and a C program that shows the value of the switches on the four right-most digits of the 7-segment displays.

Exercise 2. Write a RISC-V assembly program and a C program that shows the string “0-1-2-3-4-5-6-7-8” moving from the right to the left of the 8-digit 7-segment displays. That is, 0 should show up on the right-most digit first. Then it should move to the left and 1 should show up on the right-most digit, and so on.

5. 8-DIGIT 7-SEGMENT DISPLAY CONTROLLER: LOW-LEVEL IMPLEMENTATION, SIMULATION

Up until this point, we have shown how to use the 8-digit 7-segment displays only. In this section, we describe their low-level implementation and analyse RVfpgaSim in simulation when executing a simple assembly example. Finally, we provide exercises for modifying the 8-digit 7-segment display controller.

A. Low-level implementation of the 8-digit 7-segment display controller

Similar to previous general-purpose I/O (GPIO) labs, we divide the analysis of the 8-digit 7-segment display controller into three phases:

1. Connection between the SoC and the I/O device on the board (left shadowed region in Figure 4);
2. Integration of the new controller, which is included inside the SweRVolfX System Controller contained in the SoC (middle shadowed region in Figure 4);
3. Connection between the new controller and the SweRV EH1 Core (right shadowed region in Figure 4).

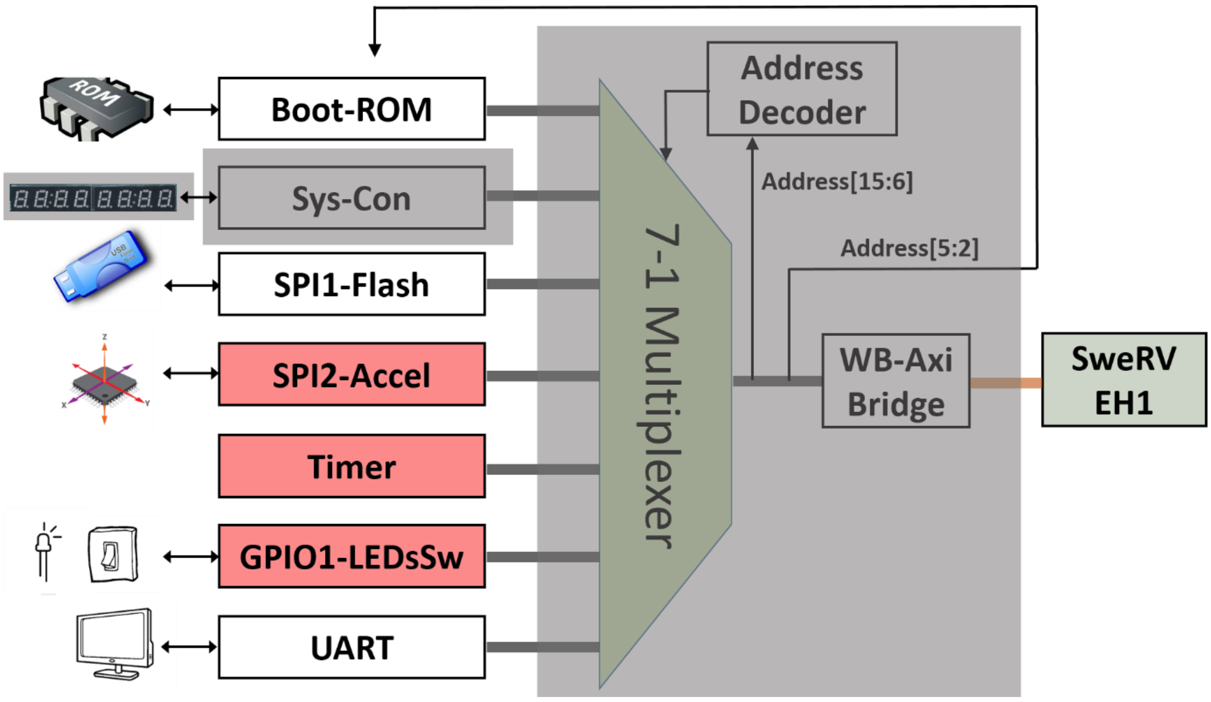
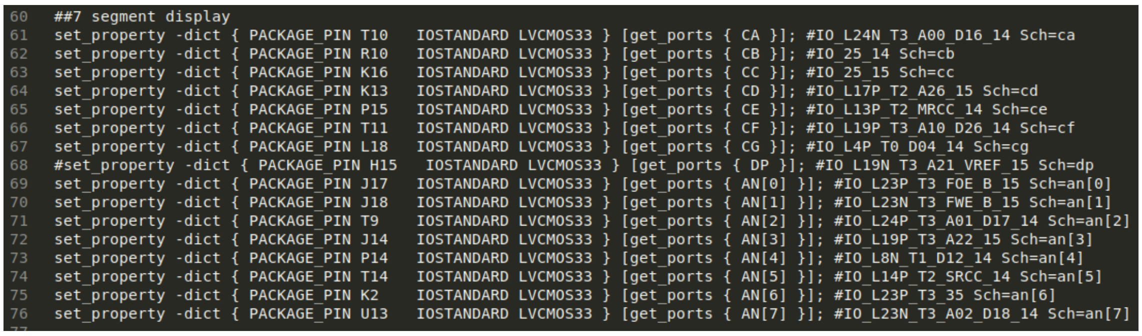


Figure 4. 8-digit 7-segment displays controller analysis in 3 phases

1. Connection of the LEDs/Switches to the SoC

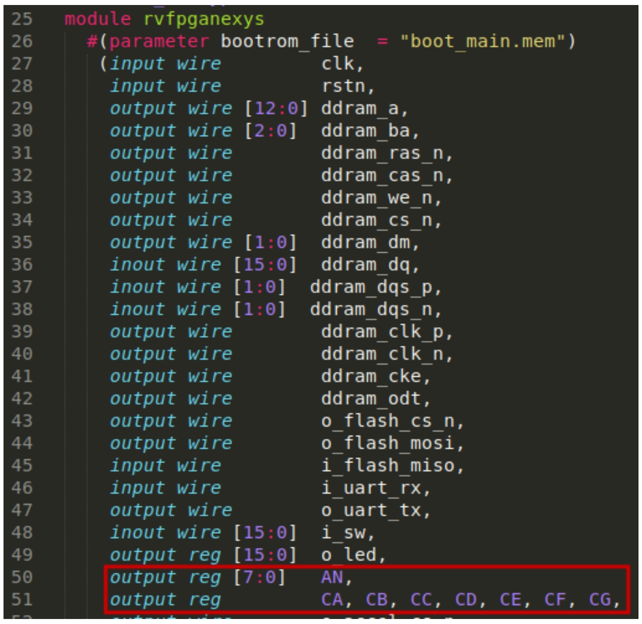
The constraints file of the project (*[RVfpgaPath]/RVfpga/src/rvfpganexys.xdc*) defines the connection between the input/output SoC signals and the board. Each I/O device on the Nexys A7 FPGA board is connected to a specific FPGA pin. The signal that connects the eight anodes (see Figure 3) is called *AN[i]* (with *i* ranging from 0-7), and the signals that connect the cathodes of similar segments on all 8 digits (see Figure 3) are called *CA, CB, CC, CD, CE, CF* and *CG*. Figure 5 shows the snippet of the constraints file where these connections are defined.



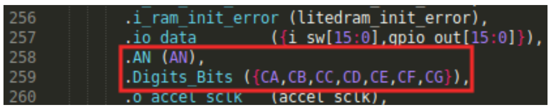
```
##7 segment display
set_property -dict { PACKAGE_PIN T10   IOSTANDARD LVCMOS33 } [get_ports { CA }]; #IO_L24N_T3_A00_D16_14 Sch=ca
set_property -dict { PACKAGE_PIN R10   IOSTANDARD LVCMOS33 } [get_ports { CB }]; #IO_25_14 Sch=cb
set_property -dict { PACKAGE_PIN K16   IOSTANDARD LVCMOS33 } [get_ports { CC }]; #IO_25_15 Sch=cc
set_property -dict { PACKAGE_PIN K13   IOSTANDARD LVCMOS33 } [get_ports { CD }]; #IO_L17P_T2_A26_15 Sch=cd
set_property -dict { PACKAGE_PIN P15   IOSTANDARD LVCMOS33 } [get_ports { CE }]; #IO_L13P_T2_MRCC_14 Sch=ce
set_property -dict { PACKAGE_PIN T11   IOSTANDARD LVCMOS33 } [get_ports { CF }]; #IO_L19P_T3_A10_D26_14 Sch=cf
set_property -dict { PACKAGE_PIN L18   IOSTANDARD LVCMOS33 } [get_ports { CG }]; #IO_L4P_T0_D04_14 Sch=cg
#set_property -dict { PACKAGE_PIN H15   IOSTANDARD LVCMOS33 } [get_ports { DP }]; #IO_L19N_T3_A21_VREF_15 Sch=dp
set_property -dict { PACKAGE_PIN J17   IOSTANDARD LVCMOS33 } [get_ports { AN[0] }]; #IO_L23P_T3_FOE_B_15 Sch=an[0]
set_property -dict { PACKAGE_PIN J18   IOSTANDARD LVCMOS33 } [get_ports { AN[1] }]; #IO_L23N_T3_FWE_B_15 Sch=an[1]
set_property -dict { PACKAGE_PIN T9    IOSTANDARD LVCMOS33 } [get_ports { AN[2] }]; #IO_L24P_T3_A01_D17_14 Sch=an[2]
set_property -dict { PACKAGE_PIN J14   IOSTANDARD LVCMOS33 } [get_ports { AN[3] }]; #IO_L19P_T3_A22_15 Sch=an[3]
set_property -dict { PACKAGE_PIN P14   IOSTANDARD LVCMOS33 } [get_ports { AN[4] }]; #IO_L8N_T1_D12_14 Sch=an[4]
set_property -dict { PACKAGE_PIN T14   IOSTANDARD LVCMOS33 } [get_ports { AN[5] }]; #IO_L14P_T2_SRCC_14 Sch=an[5]
set_property -dict { PACKAGE_PIN K2    IOSTANDARD LVCMOS33 } [get_ports { AN[6] }]; #IO_L23P_T3_35 Sch=an[6]
set_property -dict { PACKAGE_PIN U13   IOSTANDARD LVCMOS33 } [get_ports { AN[7] }]; #IO_L23N_T3_A02_D18_14 Sch=an[7]
```

Figure 5. Connection of the 8-digit 7-segment displays inputs (file *rvfpganexys.xdc*).

In lines 50-51 of the top-module of our system (module **rvfpganexys**, implemented in file *[RVfpgaPath]/RVfpga/src/rvfpganexys.sv*) you can find the 8-digit 7-segment displays input signals connected to the SoC, *AN[i]* and *CA…CG* (left part of Figure 6), and at the end of that module (right part of Figure 6) you can find their connection to the **swervolf_core** module (note that the *CA…CG* signals are renamed in that module as *Digits_Bits[6:0]*).



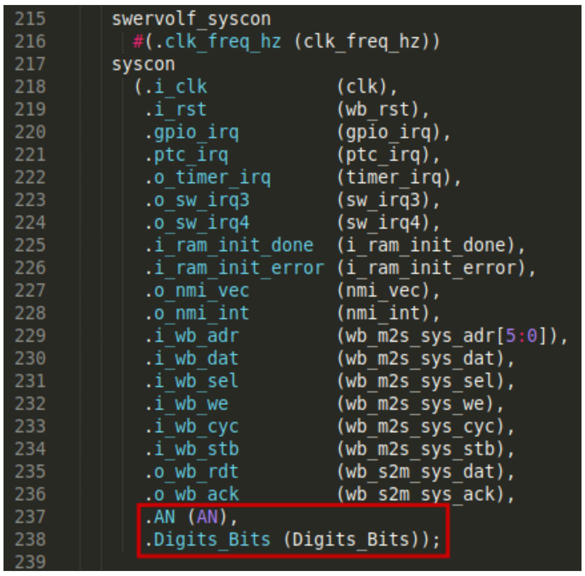
```
25  module rvfpganexys
26    #(parameter bootrom_file  = "boot_main.mem")
27     (input wire          clk,
28      input wire          rstn,
29      output wire [12:0]  ddram_a,
30      output wire [2:0]   ddram_ba,
31      output wire         ddram_ras_n,
32      output wire         ddram_cas_n,
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,
50      output reg [7:0]    AN,
51      output reg          CA, CB, CC, CD, CE, CF, CG,
```



```
256        .i_ram_init_error (litedram_init_error),
257        .io_data        ({i_sw[15:0],gpio_out[15:0]}),
258        .AN (AN),
259        .Digits_Bits ({CA,CB,CC,CD,CE,CF,CG}),
260        .o_accel_sclk   (accel_sclk),
```

Figure 6. Connection of the 8-digit 7-segment displays to the SoC (file: *rvfpganexys.sv*).

Finally, the two signals are inserted from the **swervolf_core** module into the System Controller module (**swervolf_syscon**) (see Figure 7), where the 8-digit 7-segment display controller is implemented.



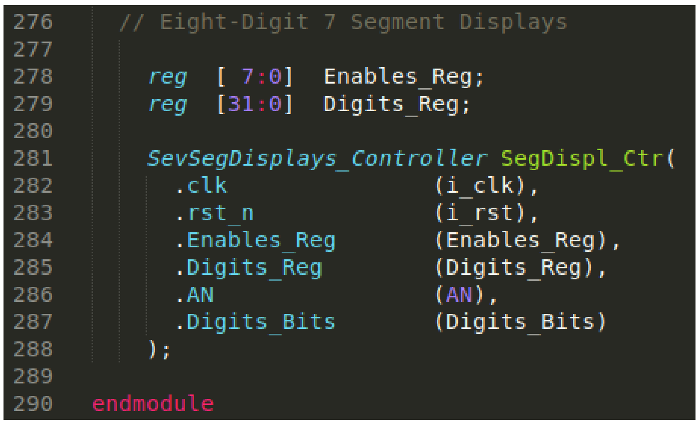
```
   swervolf_syscon
     #(.clk_freq_hz (clk_freq_hz))
   syscon
     (.i_clk            (clk),
      .i_rst            (wb_rst),
      .gpio_irq         (gpio_irq),
      .ptc_irq          (ptc_irq),
      .o_timer_irq      (timer_irq),
      .o_sw_irq3        (sw_irq3),
      .o_sw_irq4        (sw_irq4),
      .i_ram_init_done  (i_ram_init_done),
      .i_ram_init_error (i_ram_init_error),
      .o_nmi_vec        (nmi_vec),
      .o_nmi_int        (nmi_int),
      .i_wb_adr         (wb_m2s_sys_adr[5:0]),
      .i_wb_dat         (wb_m2s_sys_dat),
      .i_wb_sel         (wb_m2s_sys_sel),
      .i_wb_we          (wb_m2s_sys_we),
      .i_wb_cyc         (wb_m2s_sys_cyc),
      .i_wb_stb         (wb_m2s_sys_stb),
      .o_wb_rdt         (wb_s2m_sys_dat),
      .o_wb_ack         (wb_s2m_sys_ack),
      .AN (AN),
      .Digits_Bits (Digits_Bits));
```

Figure 7. Connection of the 8-digit 7-segment displays to the System Controller (file: *swervolf_core.v*).

TASK: Follow these signals (*CA-CG* and *AN*) from the constraints file to the System Controller module (where *CA-CG* are merged into array *Digits_Bits*). You will need to inspect the following files:
[RVfpgaPath]/RVfpga/src/rvfpganexys.xdc
[RVfpgaPath]/RVfpga/src/rvfpganexys.sv
[RVfpgaPath]/RVfpga/src/SweRVolfSoC/swervolf_core.v
[RVfpgaPath]/RVfpga/src/SweRVolfSoC/Peripherals/SystemController/swervolf_syscon.v

2. Integration of the 8-digit 7-segment display controller into the SoC

In lines 276-288 of module **swervolf_syscon** (*[RVfpgaPath]/RVfpga/src/SweRVolfSoC/Peripherals/SystemController/swervolf_syscon.v*) the 8-digit 7-segment display controller is instantiated and integrated in the SoC (see Figure 8).



```
276    // Eight-Digit 7 Segment Displays
277
278      reg  [ 7:0]  Enables_Reg;
279      reg  [31:0]  Digits_Reg;
280
281      SevSegDisplays_Controller SegDispl_Ctr(
282        .clk               (i_clk),
283        .rst_n             (i_rst),
284        .Enables_Reg       (Enables_Reg),
285        .Digits_Reg        (Digits_Reg),
286        .AN                (AN),
287        .Digits_Bits       (Digits_Bits)
288      );
289
290  endmodule
```

Figure 8. 8-digit 7-segment displays controller instantiation (file: *swervolf_syscon.v*).

The **SevSegdisplays_Controller** module receives, in addition to the clock signal (*i_clk*, renamed as *clk*) and the reset signal (*i_rst*, renamed as *rst_n*), two input signals (*Enables_Reg* and *Digits_Reg*), which are the two memory-mapped control registers already described. This module outputs two signals, *AN* and *Digits_Bits*, which are connected to the 7-segment displays on the board. For the example showing *71* on the two right-most digits, the **SevSegdisplays_Controller** would assign the following values to signals *AN* and *Digits_Bits*:

- From 0 to 2ms: Signal *AN[0]* is low to enable digit 0 (the right-most digit) to display. Signals *Digits_Bits[5]* and *Digits_Bits[4]* (that correspond to *CB* and *CC*) are also low to display "1" on digit 0 (the right-most digit). All other signals are high.
- From 2 to 4ms: Signal *AN[1]* is low to enable digit 1 to display. *Digits_Bits[6], Digits_Bits[5]* and *Digits_Bits[4]* (that correspond to *CA, CB,* and *CC*) are high to display "*7"* on digit 1. All other signals are high.
- From 4 to 16ms: *AN[2]*…*AN[7]* are high in 2 ms intervals so that they do not display values. The segments are also high for the remaining digits, digits 2-7.

The **SevSegdisplays_Controller** module is implemented in lines 295-366 of file *[RVfpgaPath]/RVfpga/src/SweRVolfSoC/Peripherals/SystemController/swervolf_syscon.v*. It contains the following subunits:

- Two multiplexers select the value to send to the *AN* and *Digits_Bits* signals every 2ms. The multiplexer is implemented inside module **SevSegMux**.
- For creating the 2ms period, we use a **counter** module provided in files

counter.sv and *delta_counter.sv*, both included in folder *[RVfpgaPath]/RVfpga/src/OtherSources/pulp-platform.org__common_cells_1.20.0/src*. The counter is configured to count from 0 to 2^{19}, and the 3 most significant bits, which change approximately every 2ms, are used as the select signals for the two multiplexers described above.

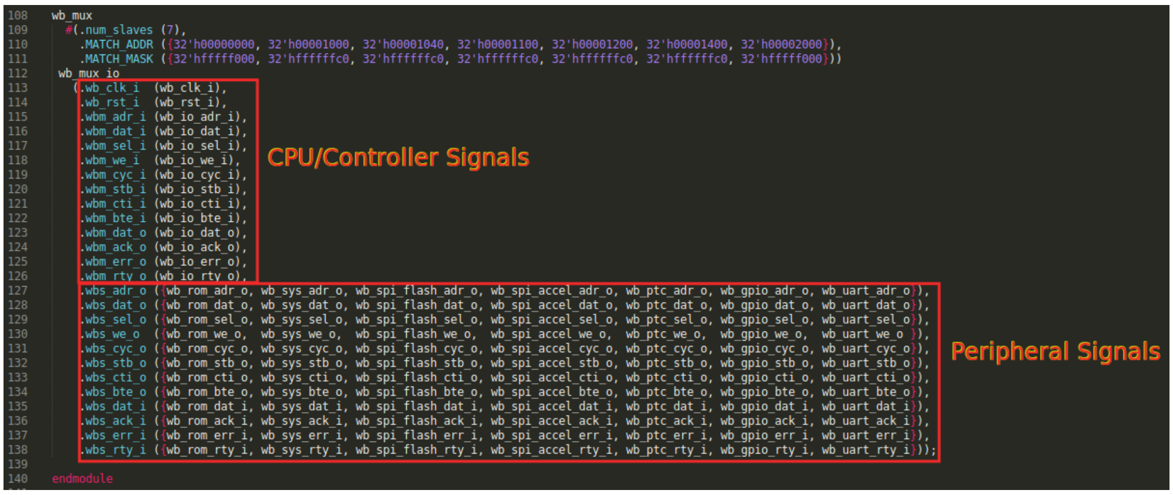
- A decoder is implemented in module **SevenSegDecoder**, which outputs the segment values for a given 4-bit hexadecimal value.

TASKS: Analyse the **SevSegdisplays_Controller** module in detail. The simulation performed in the next section can help you on this task. You can also extend the simulation with new signals if necessary.

3. Connection between the 8-digit 7-segment displays controller and the SweRV EH1 Core

As described in Lab 6, the device controllers are connected to the SweRV EH1 Core using a multiplexer (see Figure 4). Remember that the 7:1 multiplexer (Figure 9) is instantiated in file *[RVfpgaPath]/RVfpga/src/SweRVolfSoC/Interconnect/WishboneInterconnect/wb_intercon.v*. Then, the **wb_intercon** module is instantiated in lines 104-205 of file *[RVfpgaPath]/RVfpga/src/SweRVolfSoC/Interconnect/WishboneInterconnect/wb_intercon.vh*. This latter file is included in line 145 of the **swervolf_core** module located here: *[RVfpgaPath]/RVfpga/src/SweRVolfSoC/swervolf_core.v*.

The multiplexer selects which peripheral to read or write, connecting the CPU (*wb_io_** signals – lines 115-126 of Figure 9) with the Wishbone Bus of one peripheral (lines 127-138 of Figure 9), depending on the address (lines 110-111). For example, if the address generated by the CPU is in the range 0x80001000-0x8000103F, the System Controller is selected, and thus signals *wb_io_** will be connected with signals *wb_sys_**.

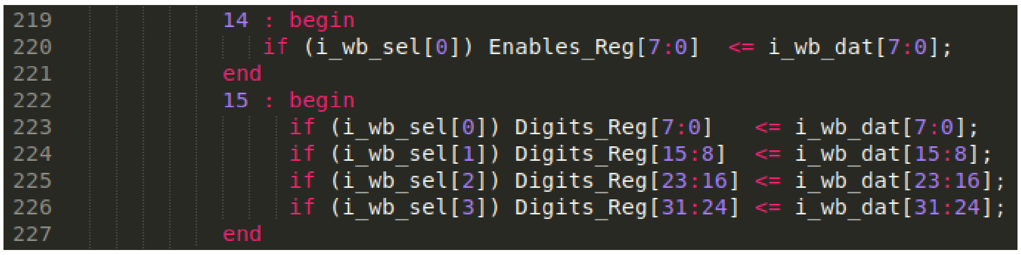


```
108   wb_mux
109     #(.num_slaves (7),
110       .MATCH_ADDR ({32'h00000000, 32'h00001000, 32'h00001040, 32'h00001100, 32'h00001200, 32'h00001400, 32'h00002000}),
111       .MATCH_MASK ({32'hfffff000, 32'hffffffc0, 32'hffffffc0, 32'hffffffc0, 32'hffffffc0, 32'hffffffc0, 32'hfffff000}))
112    wb_mux_io
113      (.wb_clk_i  (wb_clk_i),
114       .wb_rst_i  (wb_rst_i),
115       .wbm_adr_i (wb_io_adr_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_adr_o ({wb_rom_adr_o, wb_sys_adr_o, wb_spi_flash_adr_o, wb_spi_accel_adr_o, wb_ptc_adr_o, wb_gpio_adr_o, wb_uart_adr_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
```

Figure 9. 7-1 multiplexer that selects the peripheral connected with the CPU (file: *wb_intercon.v*).

The registers included in the System Controller are written from the CPU by directly connecting them to the data signal of the Wishbone Bus (*i_wb_dat*), based on the address (*i_wb_adr*) generated by the CPU (lines 162-228 of module **swervolf_syscon**).

<u>TASK</u>: Inspect lines 162-228 of module **swervolf_syscon** in order to understand how addresses are mapped in the System Controller. Focus on lines 219 to 227 (Figure 10), which refer to registers *Enables_Reg* and *Digits_Reg* (as we mentioned before, the addresses assigned to these two registers are 0x80001038 and 0x8000103C respectively).



```
219                 14 : begin
220                    if (i_wb_sel[0]) Enables_Reg[7:0]  <= i_wb_dat[7:0];
221                 end
222                 15 : begin
223                      if (i_wb_sel[0]) Digits_Reg[7:0]   <= i_wb_dat[7:0];
224                      if (i_wb_sel[1]) Digits_Reg[15:8]  <= i_wb_dat[15:8];
225                      if (i_wb_sel[2]) Digits_Reg[23:16] <= i_wb_dat[23:16];
226                      if (i_wb_sel[3]) Digits_Reg[31:24] <= i_wb_dat[31:24];
227                 end
```

Figure 10. Connection between the 8-digit 7-segment displays and the Core (file *swervolf_syscon.v*).

B. Verilator Simulation

In this section, we use **RVfpgaSim** for inspecting the main signals of the 8-digit 7-segment displays controller when the processor executes a simple example that drives this peripheral. In simulation, we analyse signals *AN* and *Digits_Bits* while we execute the example from Figure 11, which writes 71 onto the two right-most digits. You can find this program at: `[RVfpgaPath]/RVfpga/Labs/Lab7/71_7SegDispl` (you can also find the C version at: `[RVfpgaPath]/RVfpga/Labs/Lab7/71_7SegDispl_C-Lang`).

```
#define SegEn_ADDR     0x80001038
#define SegDig_ADDR    0x8000103C

.globl main
main:

    li t1, SegEn_ADDR
    li t6, 0xFC
    sb t6, 0(t1)                                # Enable the 7SegDisplays

    li t1, SegDig_ADDR
    li t6, 0x71
    sw t6, 0(t1)                                # Write the 7SegDisplays

next:  beq zero, zero, next

.end
```

Figure 11. 71_7SegDispl.S example

Figure 12 shows the disassembly version of the 71_7SegDispl.elf program, which, after compilation in PlatformIO, you can find at:
`[RVfpgaPath]/RVfpga/Labs/Lab7/71_7SegDispl/.pio/build/swervolf_nexys/firmware.dis`

```
00000090 <main>:
  90:  80001337            lui    t1,0x80001
  94:  03830313            addi   t1,t1,56 # 80001038
  98:  0fc00f93            li     t6,252
  9c:  01f30023            sb     t6,0(t1)
  a0:  80001337              lui    t1,0x80001
  a4:  03c30313              addi   t1,t1,60 # 8000103c
  a8:  07100f93              li     t6,113
  ac:  01f32023              sw     t6,0(t1)

000000b0 <next>:
  b0:  00000063              beqz   zero,b0 <next>
```

Figure 12. Disassembly version of the 71_7SegDispl.S example

Follow the next steps for running the simulation. (If you prefer not to run the simulation, you can directly go to step 7.)

1. In this case, for the simulation only, you must reduce the clock period by changing COUNT_MAX (see line 295 of file *[RVfpgaPath]/RVfpga/src/SweRVolfSoC/Peripherals/SystemController/swervolf_syscon.v*) from 20 to 5; otherwise, it would take too much time to see the results. Modify the value of COUNT_MAX and then recompile RVfpgaSim by executing the following commands (this was explained in the GSG):

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

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

WINDOWS: in case you are using Windows, you have to execute these commands inside the Cygwin terminal (refer to Section 6 and Appendix C in the Getting Started Guide for the detailed instructions). Note that the *C:* Windows folder can be found inside Cygwin at: */cygdrive/c*.

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

2. Open VSCode/PlatformIO on your computer.

3. On the top bar, click on *File* - *Open Folder...*, and browse into directory *[RVfpgaPath]/RVfpga/Labs/Lab7*

4. Select directory *71_7SegDispl* (do not open it, but just select it) and click OK. The example will open in PlatformIO.

5. Open file *platformio.ini* and check if the path to the RVfpgaSim simulation binary is correct. Remember from the GSG that it should look like:
   ```
   board_debug.verilator.binary =
   [RVfpgaPath]/RVfpga/verilatorSIM/Vrvfpgasim
   ```

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

   File *trace.vcd* should have been generated inside *[RVfpgaPath]/RVfpga/Labs/Lab7/71_7SegDispl/.pio/build/swervolf_nexys*, and you can open it with *GTKWave* by executing the following command:

```
gtkwave [RVfpgaPath]/RVfpga/Labs/Lab7/71_7SegDispl/.pio/build/swervolf_nexys/trace.vcd
```

WINDOWS: folder *gtkwave64* that you downloaded, includes an application called *gtkwave.exe* inside the *bin* folder. Launch GTKWave by double clicking on that application. On the top part of the application, click on **File – Open New Tab**, and open the trace.vcd file generated in folder *[RVfpgaPath]/RVfpga/Labs/Lab7/71_7SegDispl/.pio/build/swervolf_nexys*.

7. Include the following signals in the simulation (go into the referred modules for locating each of the signals):
   - rvfpgasim – swervolf – syscon – SegDispl_Ctr
     - ✓ Input signals: ***Enables_Reg*** and ***Digits_Reg***.
     - ✓ Output signals: ***AN*** and ***Digits_Bits***.

8. Analyse the simulation shown in Figure 13. Initially, the values shown on the eight 7-segment displays are all 0 (initially all the digits are enabled as *Enables_Reg*=0). We then disable the six left-most digits by writing *0xFC* to the *Enables_Reg* (`sb` instruction in Figure 12) and write *71* into the two right-most digits by writing *0x71* to the *Digits_Reg* (`sw` instruction in Figure 12). The effect on the output signals is as follows (as shown in Figure 13):
   - In the first period: *AN=0xFE* and *Digits_Bits=0x4F*, thus displaying *1* on the right-most digit, digit 0.
   - In the second period: *AN=0xFD* and *Digits_Bits=0x0F*, thus displaying *7* on the next digit, digit 1.
   - In the next six periods: *AN=0xFF* and *Digits_Bits=0x01*, thus switching off the six left-most digits.
   - This process then repeats.

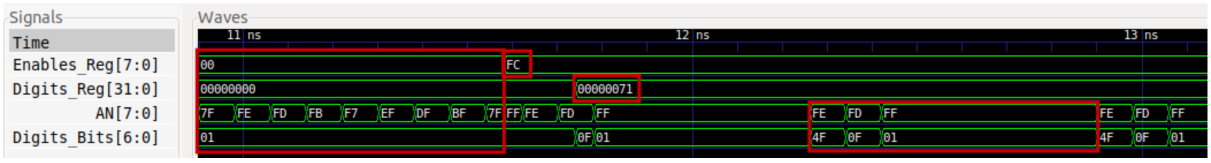


Figure 13. Write value 71 on the two right-most digits of the 8-digits 7-segment displays

9. Before continuing, do not forget to restore the value of COUNT_MAX to its original value (COUNT_MAX=20).

6. ADVANCED EXERCISES

Exercise 3. Modify the controller described in this lab so that the 8-digit 7-segment displays can show any combination of ON/OFF LEDs.

- You do not need an enable register now. Instead, you need eight 7-bit registers. Call them: Segments_Digit0 – Segments_Digit7, one for each of the eight 7-segment displays. In each of these registers, each bit indicates if the corresponding segment is ON (0) or OFF (1). For example, if all the bits of the first register (Segments_Digit0) are 0, all segments in the right-most digit will be ON, whereas if all the bits of the first register are 1, all segments of the right-most digit will be OFF.

- You can map these two new registers to the same addresses that we used before (first remove the two previous registers *Enables_Reg* and *Digits_Reg*):
    - Segments_Digit0 ←→ Address 0x80001038
    - Segments_Digit1 ←→ Address 0x80001039
    - …
    - Segments_Digit7 ←→ Address 0x8000103F
- Note that you do not need the 4-7 decoder anymore (module **SevenSegDecoder**), as the information provided by the program is already decoded.

Exercise 4. Use the new controller for printing the following on the 8-digit 7-segment displays: “I SAY HI”. As usual, implement both RISC-V assembly and C versions of the program.