

FFIT

User Manual



Universidad
Rey Juan Carlos

1 Document description

This manual describes the usage of FFIT (FPGA Fault Injection Tool) as well as the modifications in the source files that have to be done to support additional FPGA parts.

The current version of FFIT supports:

- AMD Kintex UltraScale FPGA KCU105 Evaluation Kit.
- Digilent Nexys 4 DDR / Nexys A7-100T.

Materials required:

- One of the FPGA boards mentioned before.
- Digilent USB to UART interface PMODs and USB cables [1].
- A computer running the FFIT suite and Xilinx Vivado.

Experimental set-up example:

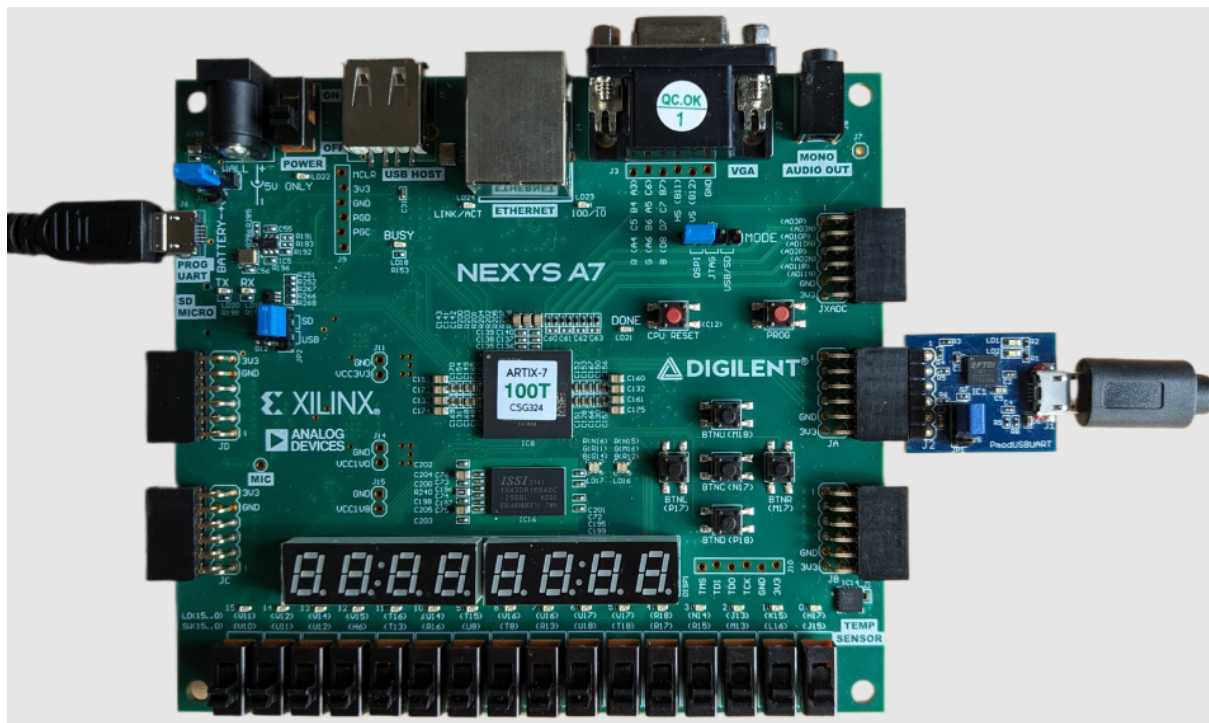


Figure 1: Experimental set-up example for the Nexys A7-100T showing the required connections between the FPGA and the host computer. The USB cable on the left is used to send the output of the design under test to the host computer. The USB on the right is connected through a Digilent PMOD to control the fault injection campaign.

2 FFIT suite usage

This section presents a beginner's step-by-step guide, from creating a Vivado project with the design under test to performing a fault injection campaign with the FFIT suite.

Creating a Vivado Project

Once Vivado is open, the first step is to click on *Create Project* to display the *New Project* wizard. In the wizard:

1. Click on *Next* and enter the name of the project and the project location. It is recommended to use a path without spaces. Click on *Next*.
2. In the *Project Type* step, select *RTL Project*. Click on *Next*.
3. In the *Add Sources* step, select the *Target language*. In this example, VHDL is selected, but it depends on the desired language for your design. Click on *Next*.
4. Leave the *Add Constraints* step unfilled. Click on *Next*.
5. Select *Nexys A7-100T* in the *Boards* tab. If the Nexys A7 board is not listed, you can download the board files from the Digilent repository [2] and paste all of the folders into the C:/Xilinx/Vivado (or /opt/Xilinx/Vivado/) data/boards/board_files directory. Relaunching Vivado after that should make the board appear.
6. Click on *Next*, then click on *Finish* to create an empty Vivado project.

Adding the Xilinx SEM IP to the Project

Once the Vivado project is created, the next step is to add the Xilinx Soft Error Mitigation (SEM) IP Controller [3] to the project.

The SEM IP core serves as the fault injection engine for the emulation process. Supported by Xilinx, this IP core is designed to detect and correct soft errors in the Configuration Memory of Xilinx FPGAs. It can read the configuration memory to identify errors and, upon discovering any altered bits, it flips them to correct the errors. This ability to read and write into the configuration memory is utilized to manage fault injection.

To add the SEM IP core to a Vivado project, click on the Vivado IP Catalog and search for "sem ip". Double-clicking on the *Soft Error Mitigation* IP will open the *Customize IP* window (see Fig. 2). In this window, enable the error injection and error correction options, select *Repair* as the correction method, and set the *Controller Clock Frequency* to 100 MHz. Click on *OK* and then *Generate* in *Generate the Output Products*.

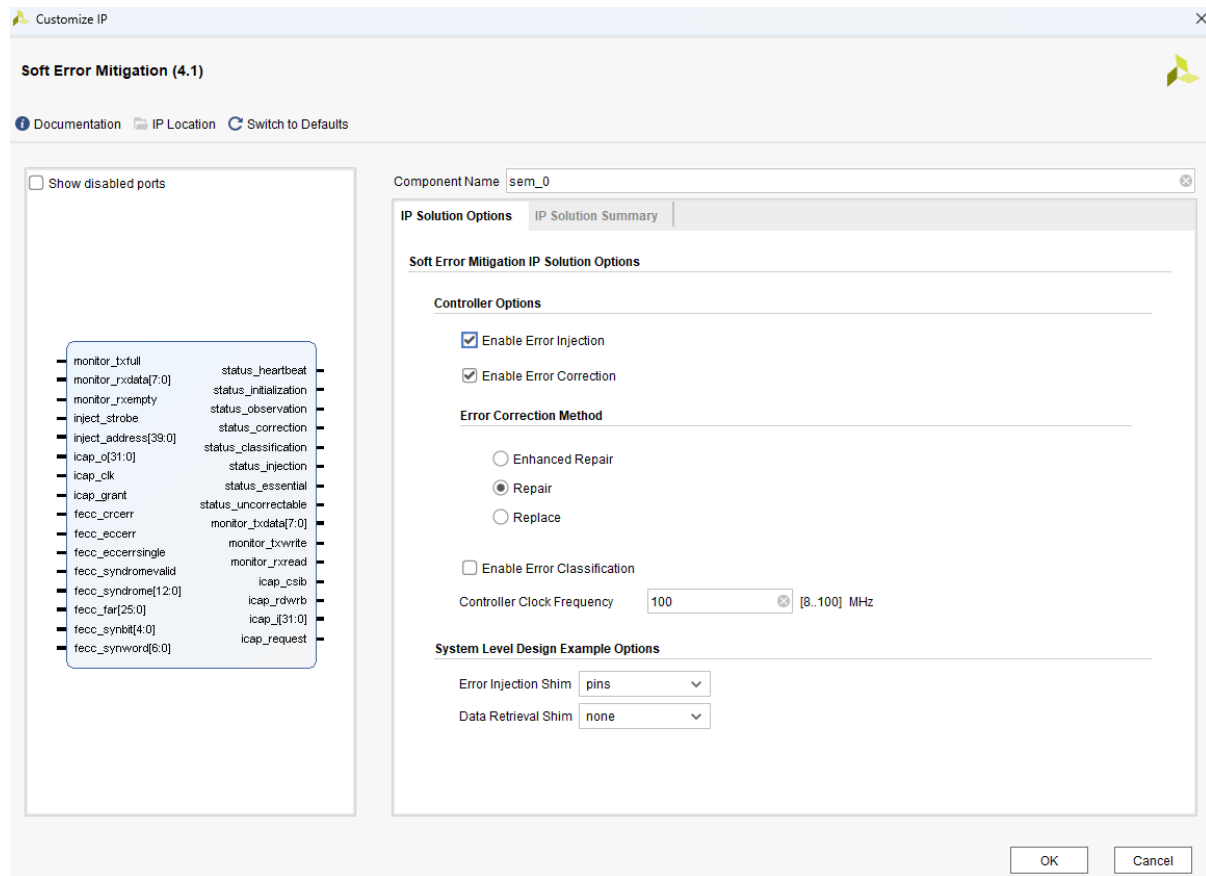


Figure 2: Customizing the Xilinx SEM IP Core

After that, the generated SEM IP VHDL files can be added to the project. To do this, in the *Project Manager* click on *Add Sources* > *Add or create design sources* > *Add files*, and add the files listed below. These files are located in */sem_0_ex/imports*. This project can be accessed by pressing right click on *sem_0* > *Open IP Example Design*.

- sem_0_sem_cfg.vhd
- sem_0_sem_example.vhd
- sem_0_sem_mon.vhd
- sem_0_sem_mon_fifo.vhd
- sem_0_sem_mon_piso.vhd
- sem_0_sem_mon_sipo.vhd

To communicate with the SEM IP, serial communication through a UART interface will be used. The communication is established at a default baud rate. To change this value, you can modify the *V_ENABLETIME* constant definition in line 182 of the *sem_0_sem_mon.vhd* file. The new value can be calculated based on the desired baud rate using the *V_ENABLETIME* calculator in the FFIT GUI or by applying the formula:

$$V_ENABLETIME = \text{round to integer} \left(\frac{\text{input clock frequency}}{16 \times \text{nominal bitrate}} \right) - 1$$

A rounding error as great as ± 0.5 can result from the computation of $V_ENABLETIME$. This error produces a bit rate that is slightly different from the nominal bit rate. A difference of approximately 2% is considered acceptable.

Example: The input clock is 100 MHz, and the desired bit rate is 115,200 baud.

$$V_ENABLETIME = \text{round to integer} \left(\frac{100,000,000}{16 \times 115,200} \right) - 1 = 53$$

With this value, the actual bit rate is approximately 115741 baud, which deviates by 0.47% from the nominal bit rate of 115200 baud. Therefore, this value is acceptable because the difference is within $\pm 1\%$. Replace the $V_ENABLETIME$ constant definition in line 182 of the `sem_0_sem_mon.vhd` file by:

```
constant V_ENABLETIME : integer := 53;
```

Once the baud rate is adjusted, the next step is to make external the `icap_clk` signal. This clock signal allows the SEM IP to synchronize with sequential designs. To do so, open the `sem_0_sem_example.vhd` file, declare the `icap_clk_out` signal in the entity, and connect it to the internal `icap_clk` signal. The VHDL code should look like this:

```
entity sem_0_sem_example is
port ( clk                : in    std_logic;
      status_heartbeat    : out   std_logic;
      status_initialization : out  std_logic;
      status_observation   : out   std_logic;
      status_correction    : out   std_logic;
      status_classification : out   std_logic;
      status_injection     : out   std_logic;
      status_essential     : out   std_logic;
      status_uncorrectable : out   std_logic;
      inject_strobe        : in    std_logic;
      inject_address       : in    std_logic_vector(39 downto 0);
      monitor_tx           : out   std_logic;
      monitor_rx           : in    std_logic;
      icap_clk_out         : out   std_logic
);
end entity sem_0_sem_example;
```

```

architecture xilinx of sem_0_sem_example is
...
begin
...
icap_grant <= '1';
status_heartbeat <= status_heartbeat_internal;
status_initialization <= status_initialization_internal;
status_observation <= status_observation_internal;
status_correction <= status_correction_internal;
status_classification <= status_classification_internal;
status_injection <= status_injection_internal;
status_essential <= status_essential_internal;
status_uncorrectable <= status_uncorrectable_internal;
icap_clk_out <= icap_clk;
...
end architecture xilinx;

```

After these modifications, the SEM IP is ready to be instantiated in a wrapper file together with the design under test. The design under test will be added in the next step. First, let's create a VHDL source file named `top` and edit it as follows to instantiate the SEM IP:

```

-- Library declaration
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Entity definition
entity top is
    Port ( clk                : in    std_logic; -- To the FPGA clock
          status_initialization : out  std_logic; -- Status signals
          status_observation   : out  std_logic; -- will be connected
          status_correction    : out  std_logic; -- to LEDs
          status_uncorrectable : out  std_logic;
          monitor_tx           : out  std_logic; -- To the UART port
          monitor_rx           : in   std_logic; -- To the UART port
    );
end top;

```

```
architecture Behavioral of top is
  component sem_0_sem_example is
    port ( clk
          : in    std_logic;
          status_heartbeat
          : out   std_logic;
          status_initialization
          : out   std_logic;
          status_observation
          : out   std_logic;
          status_correction
          : out   std_logic;
          status_classification
          : out   std_logic;
          status_injection
          : out   std_logic;
          status_essential
          : out   std_logic;
          status_uncorrectable
          : out   std_logic;
          inject_strobe
          : in    std_logic;
          inject_address
          : in    std_logic_vector(39 downto 0);
          monitor_tx
          : out   std_logic;
          monitor_rx
          : in    std_logic;
          icap_clk_out
          : out   std_logic
        );
  end component;

  signal icap_clk_out, injection_done : std_logic;
begin
  SEM : sem_0_sem_example port map(
    clk          => clk,
    status_heartbeat => open,          -- not useful
    status_initialization => status_initialization,
    status_observation => status_observation,
    status_correction => status_correction,
    status_classification => open,      -- not useful
    status_injection => injection_done,
    status_essential => open,          -- not useful
    status_uncorrectable => status_uncorrectable,
    inject_strobe => '0',             -- unused
    inject_address => (others => '0'), -- unused
    monitor_tx => monitor_tx,
    monitor_rx => monitor_rx,
    icap_clk_out => icap_clk_out
  );
end Behavioral;
```

Finally, let's include and edit the constraints file to map entity ports. Click *Add Sources* > *Add or create constraints* > *Add files* and select `sem_0_sem_example.xdc` from the `sem_0` project. Open the file in the *Constraints* folder of the *Sources* window to edit it.

```
#####
## Controller: Internal Timing constraints
#####

set_max_delay 9.0 -from [get_pins SEM/example_cfg/example_frame_ecc/*]
-quiet -datapath_only
set_max_delay 20.0 -from [get_pins SEM/example_cfg/example_frame_ecc/*]
-to [all_outputs] -quiet -datapath_only

#####
## Example Design: Master Clock
#####

create_clock -name clk -period 10.0 [get_ports clk]
set_property IOSTANDARD LVCMOS25 [get_ports clk]

#####
## Example Design: Status Pins
#####

set_property DRIVE 8 [get_ports status_initialization]
set_property SLEW FAST [get_ports status_initialization]
set_property IOSTANDARD LVCMOS25 [get_ports status_initialization]

set_property DRIVE 8 [get_ports status_observation]
set_property SLEW FAST [get_ports status_observation]
set_property IOSTANDARD LVCMOS25 [get_ports status_observation]

set_property DRIVE 8 [get_ports status_correction]
set_property SLEW FAST [get_ports status_correction]
set_property IOSTANDARD LVCMOS25 [get_ports status_correction]

set_property DRIVE 8 [get_ports status_uncorrectable]
set_property SLEW FAST [get_ports status_uncorrectable]
set_property IOSTANDARD LVCMOS25 [get_ports status_uncorrectable]
```



```
set_output_delay -clock clk -10.0 [get_ports [list status_observation
status_correction status_uncorrectable status_initialization]] -max
set_output_delay -clock clk 0 [get_ports [list status_observation
status_correction status_uncorrectable status_initialization]] -min
```

```
#####
```

```
## Example Design: MON Shim and Pins
```

```
#####
```

```
set_property DRIVE 8 [get_ports monitor_tx]
set_property SLEW FAST [get_ports monitor_tx]
set_property IOSTANDARD LVCMOS25 [get_ports monitor_tx]
set_property IOSTANDARD LVCMOS25 [get_ports monitor_rx]
```

```
set_input_delay -clock clk -max -10.0 [get_ports monitor_rx]
set_input_delay -clock clk -min 20.0 [get_ports monitor_rx]
set_output_delay -clock clk -10.0 [get_ports monitor_tx] -max
set_output_delay -clock clk 0 [get_ports monitor_tx] -min
```

```
#####
```

```
## Example Design: Logic Placement
```

```
#####
```

```
create_pblock SEM_CONTROLLER
resize_pblock -pblock SEM_CONTROLLER -add RAMB18_X1Y30:RAMB18_X1Y39
resize_pblock -pblock SEM_CONTROLLER -add RAMB36_X1Y15:RAMB36_X1Y19
resize_pblock -pblock SEM_CONTROLLER -add SLICE_X28Y75:SLICE_X35Y99
add_cells_to_pblock -pblock SEM_CONTROLLER -cells [get_cells SEM/example_controller]
add_cells_to_pblock -pblock SEM_CONTROLLER -cells [get_cells SEM/example_mon/*]
```

```
## Force ICAP to the required (top) site in the device.
```

```
## Force FRAME_ECC to the required (only) site in the device.
```

```
set_property LOC FRAME_ECC_X0Y0 [get_cells SEM/example_cfg/example_frame_ecc]
set_property LOC ICAP_X0Y1 [get_cells SEM/example_cfg/example_icap]
```

```
#####
## Example Design: I/O Placement
#####
```

```
set_property LOC E3 [get_ports clk]
set_property LOC H17 [get_ports status_initialization]
set_property LOC K15 [get_ports status_observation]
set_property LOC J13 [get_ports status_correction]
set_property LOC D14 [get_ports status_uncorrectable]
set_property LOC D18 [get_ports monitor_tx]
set_property LOC E18 [get_ports monitor_rx]
```

At this point, the SEM IP design is fully functional, and the *Generate Bitstream* step can be executed to program the FPGA and verify its correct behavior. To perform this sanity check, click on *Generate Bitstream*, connect the FPGA to the computer as shown in Fig. 1, open a serial receiver application such as TeraTerm, set the baud rate to 115,200, configure new-line reception to AUTO, and then follow these steps in Vivado: *Open Hardware Manager > Open Target > Auto Connect > Program device > Program*. If everything works correctly, the following message should appear in the terminal:

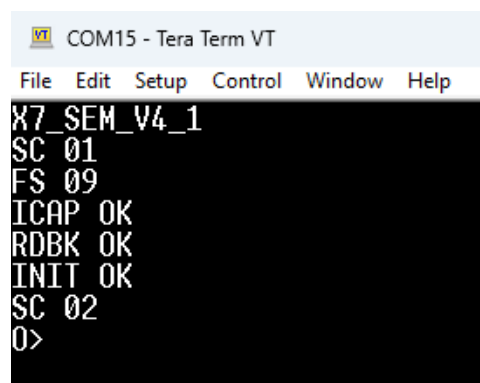


Figure 3: SEM IP messages received in the Tera Term terminal. The SEM IP initializes correctly and waits in observation mode.

As mentioned, the design is fully functional and the SEM IP can be controlled by sending input commands. For more information about manual use of the SEM IP, please refer to the official Xilinx documentation [4].

In the next subsection, a design under test will be added to the SEM IP Vivado project explained in this section to perform error injection campaigns.

Adding the Design Under Test to the Project

To illustrate the fault injection process, a 10-tap finite impulse response (FIR) filter is used as the design under test. The VHDL code for the filter is available in the GitHub repository and Appendix A.

The test setup consists of an input ROM, two FIR filters (one of which will be subjected to fault injection), a comparator, and a UART interface. Both FIR filters process the same inputs from the ROM, and their outputs are compared to detect errors. Since faults are injected only into one of them, any discrepancy in the outputs indicates that an error has caused a fault. The comparator's output is then transmitted to the computer via the UART module. Figure 4 illustrates this setup.

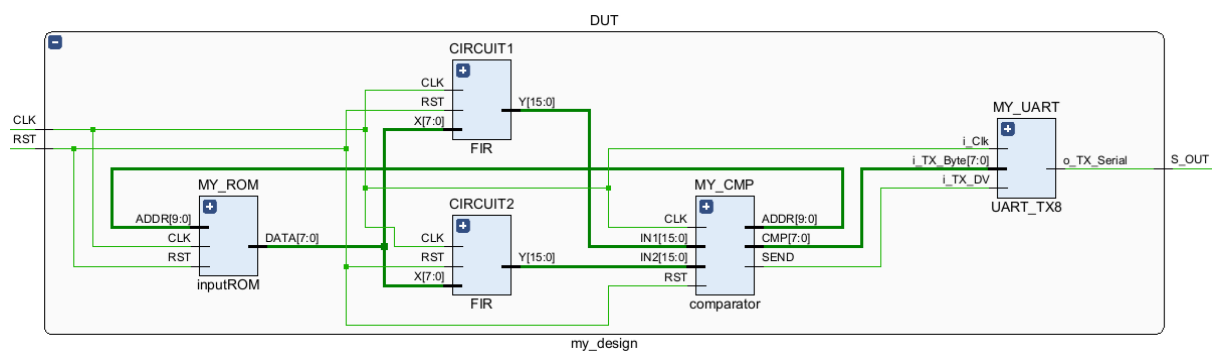


Figure 4: Design under test setup comprising two FIR filters, an input ROM, a comparator, and a UART interface. Errors are injected exclusively into CIRCUIT1, while CIRCUIT2 is used to detect discrepancies in the Y output signal.

Duplicating the FIR filter increases FPGA resource usage. If the circuit is too large to be duplicated, an alternative approach is to use a ROM with pre-recorded golden outputs. In this case, the circuit must be simulated beforehand using a testbench.

To set up the project, add the following source files, as listed in Appendix A:

- flipflopD.vhd
- FIR.vhd
- inputROM.vhd
- comparator.vhd
- uart_tx.vhd
- my_design.vhd

Then, modify the existing top.vhd file to instantiate the my_design entity. The final Vivado project hierarchy is shown in Fig. 5. The complete top.vhd file is also available in Appendix A.

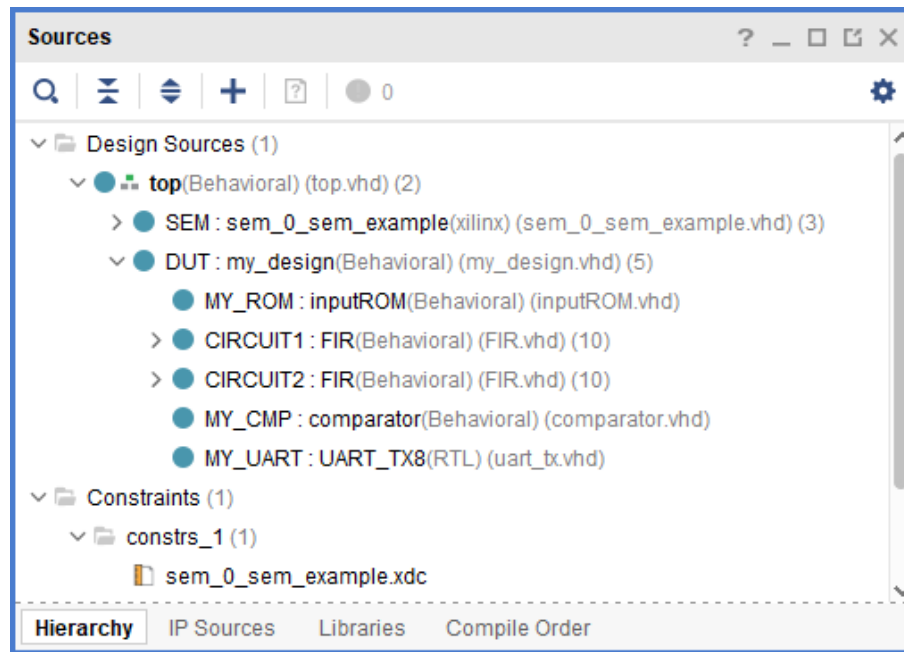


Figure 5: Vivado hierarchy of the complete project, including the SEM IP and the design under test.

Constraining the Design and Obtaining Coordinates

Create a new constraints file named `design_constraints.xdc` and copy the content below. Then click on *Generate Bitstream*.

```
#####
## DUT Logic Placement
#####
set_property -dict { PACKAGE_PIN D4    IOSTANDARD LVCMOS25 } [get_ports { S_OUT }]

create_pblock MY_DUT
resize_pblock -pblock MY_DUT -add SLICE_X32Y2:SLICE_X44Y10
add_cells_to_pblock -pblock MY_DUT -cells [get_cells DUT/CIRCUIT1/*]
set_property EXCLUDE_PLACEMENT true      [get_pblocks MY_DUT]
set_property DONT_TOUCH true             [get_cells DUT/CIRCUIT1/*]
set_property DONT_TOUCH true             [get_cells DUT/CIRCUIT2/*]

# Generate EBD File
set_property BITSTREAM.SEU.ESSENTIALBITS yes [current_design]

# For Flash Memory Programming
set_property BITSTREAM.CONFIG.SPI_BUSWIDTH 4 [current_design]
set_property CONFIG_MODE SPIx4               [current_design]
```

This new constraints file is required to connect the serial output of the design under test (S_OUT) to the UART port of the Nexys A7-100T board. In addition, the file is used to create a pBlock that encapsulates the design block intended for fault injection (denoted as CIRCUIT1). Two important properties are set in this file:

- A DONT_TOUCH property is applied to both CIRCUIT1 and CIRCUIT2. This ensures that Vivado does not optimize either of them away during synthesis. In particular, CIRCUIT2 is preserved despite being in parallel with CIRCUIT1, which might otherwise lead Vivado to consider it redundant and remove it.
- The pBlock is marked with the property EXCLUDE_PLACEMENT so that no other design cells are placed within its region.

It is important at this stage to obtain the coordinates of the pBlock (namely, X Lo, X Hi, Y Lo, and Y Hi). To do so, after generating the bitstream, click on *Open Implemented Design*, select the pBlock, and navigate to the Rectangles tab (see Fig. 6). These coordinates will later be used by FFIT to generate the fault injection addresses, so it is crucial to record them.

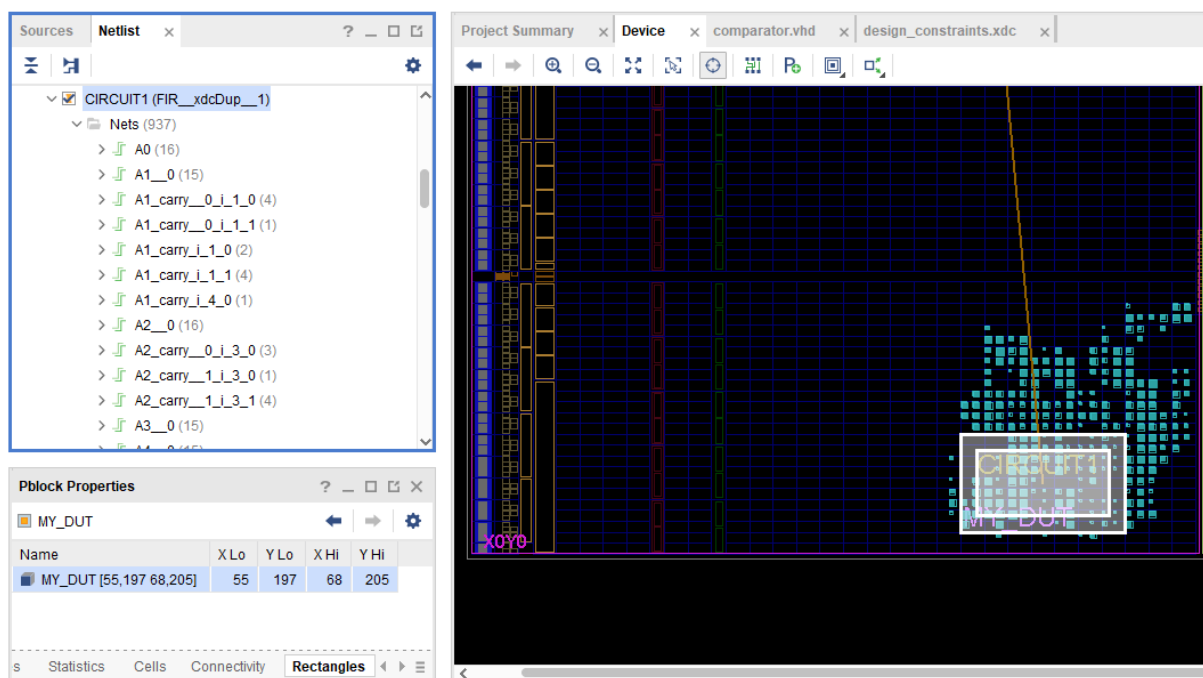


Figure 6: Vivado's Rectangles tab showing the pBlock coordinates X Lo = 55, Y Lo = 197, X Hi = 68, and Y Hi = 205

Two additional commands are added in the constraints file:

1. The command

```
set_property BITSTREAM.SEU.ESSENTIALBITS yes [current_design]
```

enables the generation of an Essential Bits File (EBD). This file, which contains only the essential configuration bits, will be used together with the pBlock coordinates to generate the injection addresses.

2. The commands

```
set_property BITSTREAM.CONFIG.SPI_BUSWIDTH 4 [current_design]  
set_property CONFIG_MODE SPIx4 [current_design]
```

are required for programming the FPGA's flash memory. With these properties set, the design can automatically be loaded from flash during a reconfiguration, eliminating the need to reload the bitstream from Vivado. For more information about SPI flash memory programming, please refer to Appendix B.

To verify the complete design, generate the bitstream, connect the PMOD and USB cables as shown in Fig. 1, then open two Tera Term terminals configured at 115,200 baud. In the SEM IP Tera Term terminal, send a capital I to set the SEM IP into idle mode. Next, type N C000000000 to inject an error into a dummy frame. Since this error does not affect the functionality of the design, the design under test Tera Term terminal should receive a 0. This 0 is the output of the comparator, indicating that both FIR filter designs are producing the same results. However, if an error is injected into a critical bit, this value is expected to change, and a 1 will be received instead. The injection addresses of these critical bits are extracted using FFIT, which also automates the fault injection campaign.

FFIT Graphical User Interface (GUI)

FFIT GUI is a Python-based tool designed to facilitate and simplify the fault injection process in FPGAs. It covers both fault injection execution and the generation of injection addresses following different strategies. Upon execution, FFIT displays a graphical user interface, as shown in Fig. 7. This interface is divided into three tabs to organize different functions and enhance usability. The first tab that opens by default when launching the application is dedicated to the generation of injection addresses, as this is the first step in the process.

Fig. 8 provides a detailed view of the *Address Generation* tab. This tab allows users to load the EBD file obtained in the previous step. To obtain the addresses of the EBD file correctly, it is necessary to select an FPGA from among those characterized (currently Nexys 4 and KCU105) and specify the circuit coordinates (X Lo, X Hi, Y Lo, and Y Hi), obtained as detailed in the previous section. Currently, the only available injection strategy is exhaustive injection, though additional strategies will be included in future updates. Once the necessary data has been entered, clicking the *Generate*

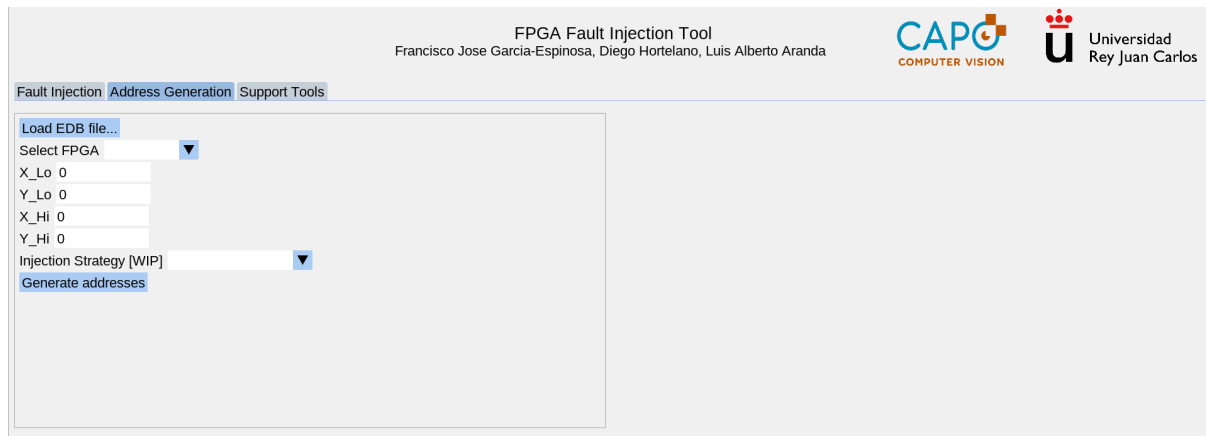


Figure 7: FFIT application: Address Generation tab

addresses button will create a file named “*injection_addresses.txt*” containing all the injection addresses.

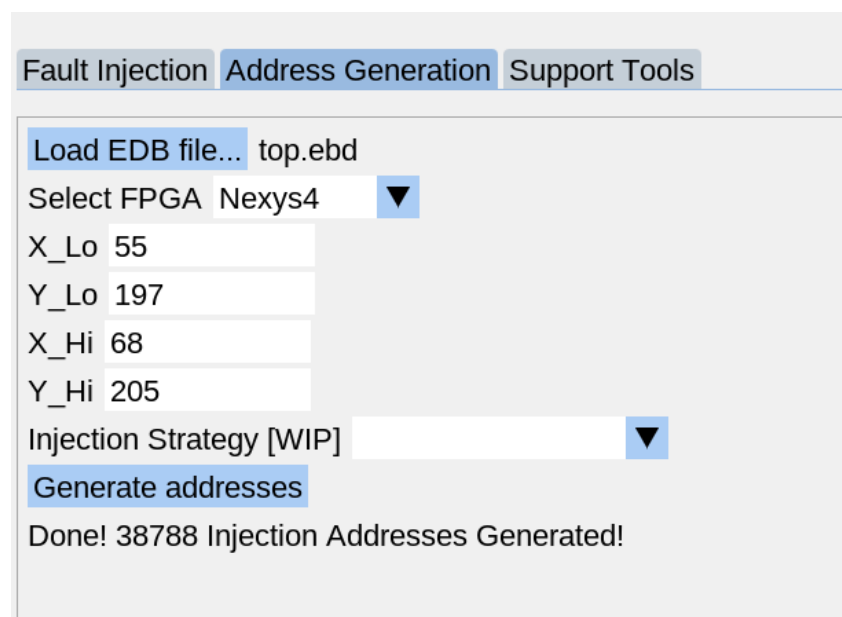


Figure 8: FFIT application: injection addresses generated

Once the injection addresses have been generated, the next step is to inject faults. The functions related to this process are located in the *Fault Injection* tab, which is shown in Fig. 9. This tab is divided into two parts: the left side focuses on application configuration, while the right side is dedicated to displaying the fault injection results.

The left part of this tab is detailed in Fig. 10. Here, users can select the previously generated address file and specify the COM ports for both the SEM IP and the designed circuit. If the FPGA was not connected before launching the application, it may be necessary to click the “*Update ports*” button to refresh the available port list. After selecting the ports and their baud rates from the available options, we can click the

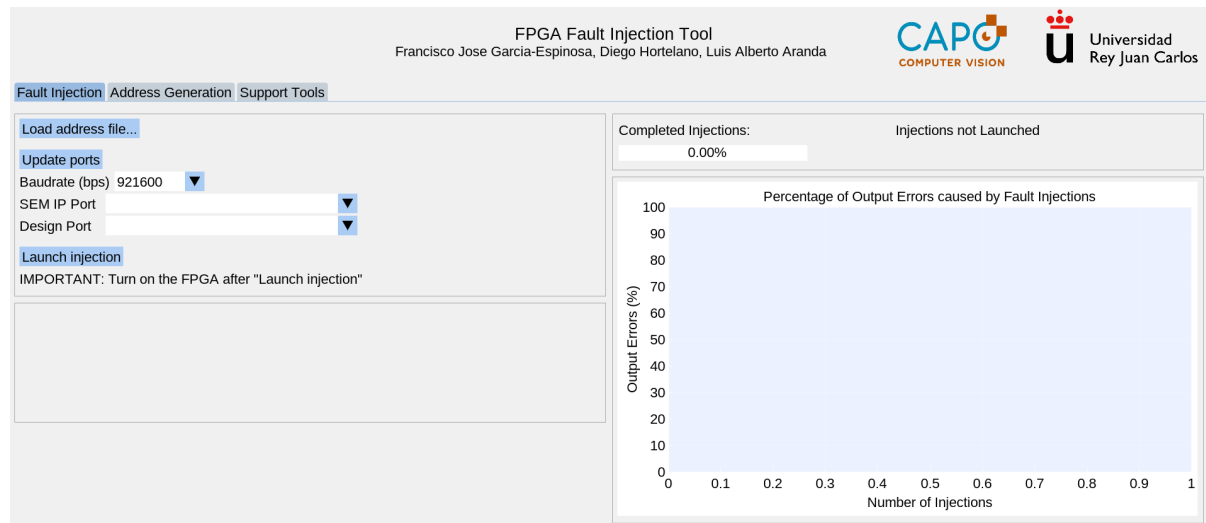


Figure 9: FFIT application: Fault Injection tab

“*Launch injection*” button and power on the FPGA. It is crucial to turn on the FPGA after clicking the button, as the application verifies the information displayed by the FPGA during startup. Once the fault injection process is running, the interface displays information on the current injection, the corresponding address, and whether an output error has occurred (1) or not (0). Due to the speed at which injections are performed, this information changes very quickly, so it is also stored in an “*out.txt*” file for further processing. Furthermore, it is important to check this information, because in case of an uncorrectable (non-recoverable) error, the application will display this information here, interrupting its execution until the FPGA is restarted.

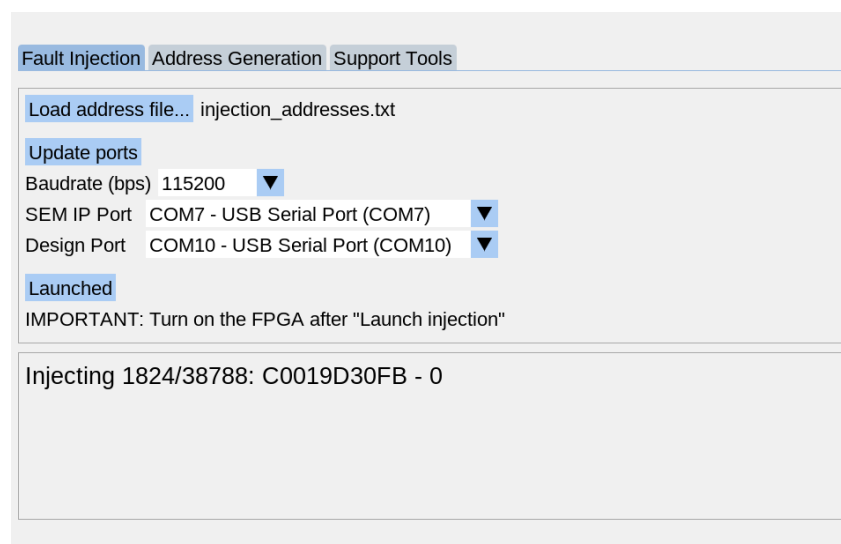


Figure 10: FFIT application: configuration of fault injections

Alternatively, Fig. 11 shows the right section of the “*Fault Injection*” tab. This section, focused on monitoring the progress and results of the fault injection campaign, pro-

vides information on the percentage of completed injections, the elapsed time since the campaign started, and the current number of output errors, as well as the percentage they represent over the number of current injections performed. This information is also represented graphically, with real-time updates after each injection, allowing users to observe trends in long injection campaigns (see 11(a)). Once the injection campaign is finished, the interface displays information regarding the completion of the campaign, showing the total time required and the number of total output errors (see 11(b)). In addition, a short summary of the fault injection campaign conducted is included at the end of the “*out.txt*” file.

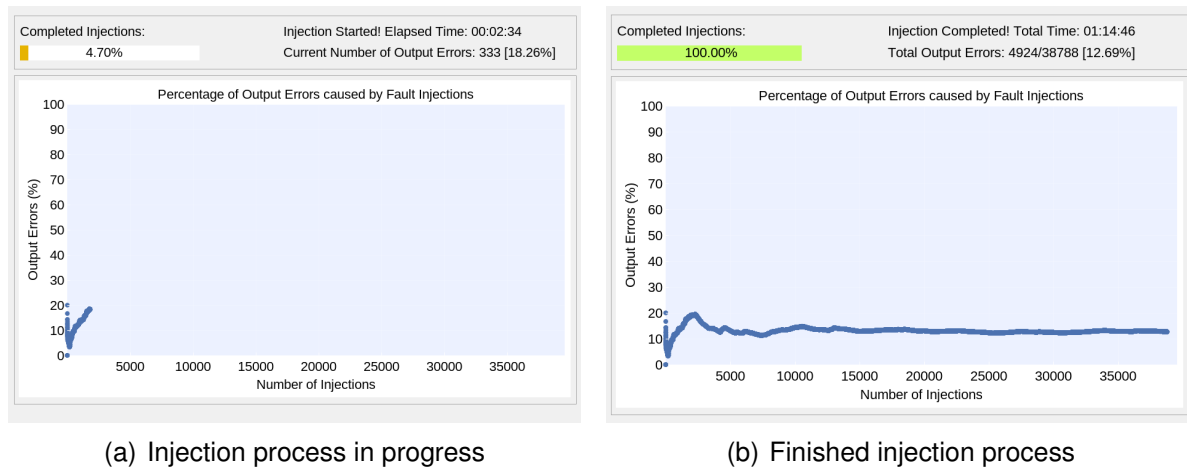


Figure 11: FFIT application: visualization of the injection process

Finally, the third tab “*Support Tools*” includes tools or utilities related to fault injection. Currently it only allows to obtain the `V_ENABLETIME` described in the previous section, although new functionalities will be added in future updates.

3 Extending FFIT to other FPGA devices

By default, FFIT supports only the KCU105 and Nexys A7 devices. However, the tool can be upgraded to include support for additional FPGA models. The steps to characterize the Nexys A7-100T are outlined below for reference.

Creating a Simple Design in Vivado

To characterize a board, you must create a simple project in the Xilinx Vivado Design Suite. In this step-by-step example, a NOT gate is used due to its simplicity and minimal resource requirements, which help reduce the number of ones in the generated EBD file. The VHDL code for the NOT gate is shown below.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity notGate is
    Port (X : in  STD_LOGIC;
          Y : out STD_LOGIC
    );
end notGate;

architecture Dataflow of notGate is
begin
    Y <= not(X);
end Dataflow;
```

To place this design within a pBlock, a top-level VHDL file must be created. This file includes the instantiation of the NOT gate. The VHDL for the top file is shown below:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity top is
    Port (X : in  STD_LOGIC;
          Y : out STD_LOGIC
    );
end top;

architecture Structural of top is
    component notGate is
```

```

    Port (X : in  STD_LOGIC;
          Y : out STD_LOGIC
    );
end component;
begin
    MY_GATE : notGate port map(X => X, Y => Y);
end Structural;

```

Next, click *Run Implementation* without applying any specific constraints. This step is essential for determining the name of the LUT cell assigned by Vivado to the NOT gate and for selecting the most suitable I/O physical pins for the board. The name of the LUT cell can be found by opening the implemented design (see Fig. 12):

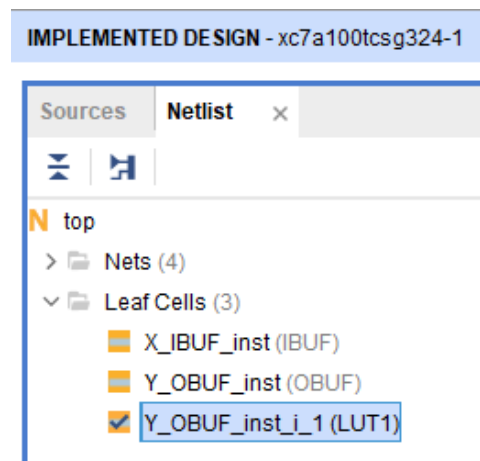


Figure 12: Obtaining LUT cell name for the NOT gate

The most suitable I/O physical pins depend on the specific board. However, since EBD files are organized from left to right within a clock region, it is crucial to select two physical pins located on the left side of the board (see Fig. 13 for reference). This approach ensures that the routing is primarily confined to the left side of the pBlock. Consequently, the last “ones” written in the EBD file are more likely to correspond to the essential bits associated with the NOT gate.

With all this information, we can now create an XDC file with the required constraints. This file defines a pBlock containing the NOT gate cell (`Y_OBUF_inst_i_1` in Fig. 12), assigns the previously determined I/O physical pins (J15 and H17), and instructs Vivado to generate the EBD file needed to characterize the board. In this example, the pBlock is positioned at the [0,0] coordinates. The content of the XDC file is shown below:

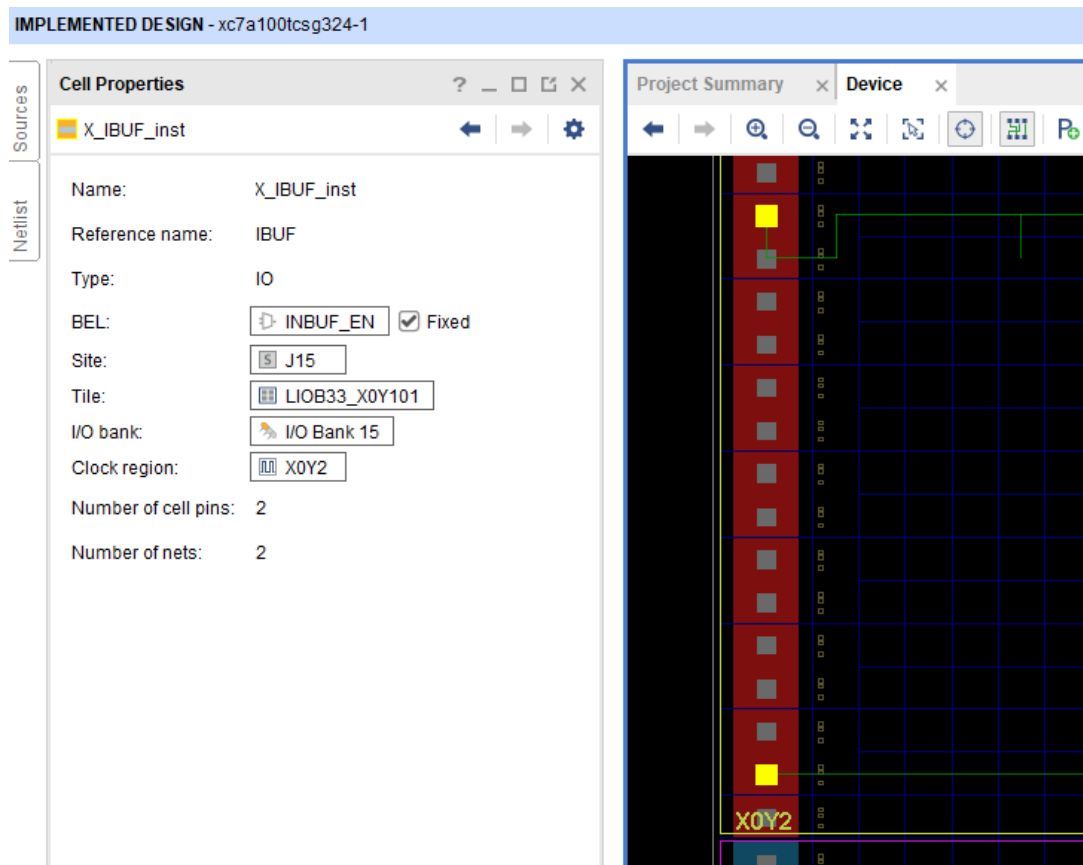


Figure 13: Selecting I/O physical pins in the Nexys

```
# Place the pBlock with the NOT gate
create_pblock MY_DUT_1
resize_pblock -pblock MY_DUT_1 -add SLICE_X0Y0:SLICE_X0Y0
add_cells_to_pblock -pblock MY_DUT_1 -cells [get_cells Y_OBUF_inst_i_1]

# Input/output mapping
set_property -dict {PACKAGE_PIN J15 IOSTANDARD LVCMOS33} [get_ports {X}]
set_property -dict {PACKAGE_PIN H17 IOSTANDARD LVCMOS33} [get_ports {Y}]

# Generate EBD File
set_property BITSTREAM.SEU.ESSENTIALBITS yes [current_design]
```

Obtaining Fx


To obtain the Fx value used in the FFIT suite, which is the amount of frames per horizontal tile, the NOT gate must be placed at both the beginning and the end of each clock region within the FPGA. For each placement, the design should be synthesized and the bitstream generated, allowing Vivado to produce the corresponding EBD file.

Once the order of the clock regions is known, the next step is to determine F_x . To do so, the following equation must be used,

$$F_x = \frac{EBDline_{max} - EBD_{header} - EBD_{dummy}}{(X_{max} - X_{min} - N_{BRAM} - N_{DSP}) \cdot W_f} \quad (1)$$

where:

$EBDline_{max}$ is the last line annotated for the current clock region. EBD_{header} is the amount of header lines in the EBD file (which is equal to 8 in the 7 Series family). EBD_{dummy} is the amount of lines of the first padding frame (which is equal to 101 in the 7 Series family [5]). X_{max} and X_{min} are the maximum and minimum X coordinates of the current clock region, N_{BRAM} and N_{DSP} the number of BRAM and DSP columns in that clock region, and W_f the number of words per frame for the FPGA family selected.

In the Nexys example, the first clock region is X0Y2 (see Fig. 14). From the corresponding EBD file, the experimentally determined last line is $EBDline_{max} = 110602$. In Vivado, the X_{max} and X_{min} coordinates for the X0Y2 clock region can be identified in the *Device* tab when the implemented design is opened (see Fig. 15). To correctly locate the first column containing logic resources, it is important to enable the *Routing Resources* view by clicking the  button.

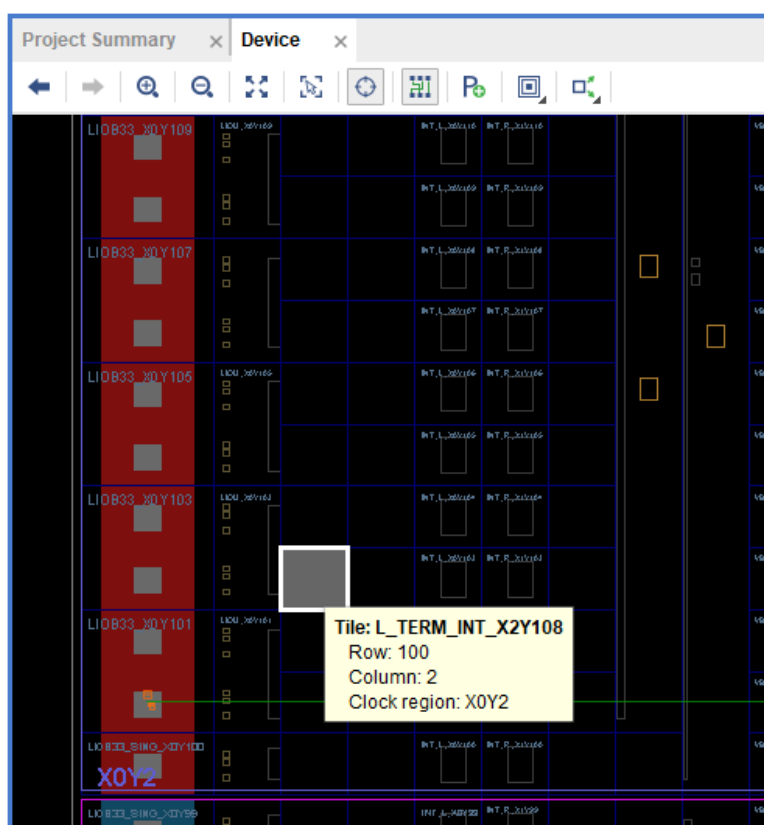


Figure 15: Obtaining X_{min} for clock region X0Y2 using the *Routing Resources* view

It is also important to notice that the column shown in Fig. 15 is the number 2, but the first column in the FPGA starts with 0, so X_{min} for this clock region is, in fact $X_{min} = 3$. Similarly, the X_{max} coordinate is the X_{min} coordinate for the next clock region, so $X_{max} = 79$ for clock region X0Y2. The values for the rest of clock regions of the Nexys are shown in Fig. 14. Similarly, by using the *Device* tab when the implemented design is opened, the number and position of the BRAM and DSP columns within each clock region can be determined. These columns are also depicted in Fig. 14 as vertical red dashed lines. The number of BRAM and DSP columns in the X0Y2 clock region are $N_{BRAM} = 1$ and $N_{DSP} = 1$. Finally, the number of words per frame (W_f) is an FPGA family-dependent parameter. For the 7 Series family, this value is $W_f = 101$ [5]. For the sake of completeness, the W_f values for both the UltraScale and the UltraScale+ are $W_f = 123$ [6] and $W_f = 93$ [7] respectively.

Now that we have obtained all this information, the F_x value can be estimated:

$$F_x = \frac{EBDline_{max} - EBD_{header} - EBD_{dummy}}{(X_{max} - X_{min} - N_{BRAM} - N_{DSP}) \cdot W_f} = \frac{110602 - 8 - 101}{(79 - 3 - 1 - 1) \cdot 101} = 14.78 \approx 15$$

Similarly, a second test to check the calculated F_x value can be performed for the X1Y2 clock region:

$$F_x = \frac{EBDline_{max} - EBD_{header} - EBD_{dummy}}{(X_{max} - X_{min} - N_{BRAM} - N_{DSP}) \cdot W_f} = \frac{196856 - 8 - 101}{(146 - 3 - 4 - 3) \cdot 101} = 14.32 \approx 15$$

Therefore, it can be concluded that, for the Nexys A7 board, $F_x = 15$.

Obtaining EBD File Ranges

Now that the F_x value for the FPGA device under study has been obtained, the EBD line associated to a specific X coordinate can be determined by using the following equation:

$$EBDline = (X_{adj} - X_{min}) \cdot F_x \cdot W_f + F_{yi} \quad (2)$$

where X_{adj} is the X coordinate of the cell minus the number of BRAM and DSP columns to the left of the cell. Therefore, an "adjustment" in X has to be performed depending on the coordinate. Naturally, this procedure must be automated in the software code by defining different X ranges and the corresponding number of columns to subtract

based on the X value. For example, the X coordinate of the leftmost BRAM column is 19. If the X value of the cell is less than 19, no subtraction is needed. However, if the X value is, for instance, 21, the BRAM column must be subtracted, resulting in an effective value of 20.

Once the input X coordinate is adjusted by subtracting the appropriate BRAM or DSP columns, equation (2) is almost ready to be used. The last parameter to calculate is F_{yi} , which is an offset in the vertical axis. This offset is the amount of EBD lines already covered in the horizontal axis by the previous clock regions, and only depends on the vertical movement among clock regions. For example, since the EBD line counting starts with regions X0Y2 and X1Y2, in these clock regions:

$$F_{y2} = 0$$

However, when moving to X0Y3, which is the third clock region, the amount of EBD lines already counted in the X0Y2–X1Y2 range must be added:

$$F_{y3} = (X_{adj} - X_{min}) \cdot F_x \cdot W_f + F_{y2} = ((145 - 3 - 4) - 3) \cdot 15 \cdot 101 + 0 = 204525$$

Similarly, for the Y1 clock regions, the EBD lines counted in the X0Y2–X1Y2 range (204525) plus the EBD lines counted in the X0Y3–X1Y3 range have to be added. Therefore,

$$F_{y1} = (X_{adj} - X_{min}) \cdot F_x \cdot W_f + F_{y3} = ((129 - 3 - 3) - 3) \cdot 15 \cdot 101 + 204525 = 181800 + 204525 = 386325.$$

Finally, for the Y0 clock regions, the EBD lines counted in the three previous ranges must be added to calculate F_{y0} :

$$F_{y0} = (X_{adj} - X_{min}) \cdot F_x \cdot W_f + F_{y1} = ((145 - 3 - 4) - 3) \cdot 15 \cdot 101 + 386325 = 204525 + 386325 = 590850.$$

Now that F_{yi} is calculated. Equation (2) can be used to determine any EBD file line. A useful test involves calculating the EBD file ranges for each clock region. These ranges are valuable for verifying that the experimental values recorded during the F_x calculation fall within the EBD range, confirming the correctness of F_x . Table 1 summarizes these values.

| Clock region | Experimental ranges | Calculated ranges |
|--------------|---------------------|-------------------|
| (1) X0Y2 | 3–110,602 | 1–112,110 |
| (2) X1Y2 | 117,060–196,856 | 112,111–204,525 |
| (3) X0Y3 | 204,829–316,232 | 204,526–316,635 |
| (4) X1Y3 | 322,898–383,597 | 316,636–386,325 |
| (5) X0Y1 | 397,840–497,525 | 386,326–498,436 |
| (6) X1Y1 | 504,092–583,680 | 498,437–590,850 |
| (7) X0Y0 | 591,263–701,747 | 590,851–702,960 |
| (8) X1Y0 | 708,314–770,629 | 702,961–772,650 |

Table 1: EBD file line ranges for each clock region of the Nexys

Including New FPGA Devices in the FFIT Code

With the reverse engineering completed and verified. The final step is including the previous experimental parameters in the code. As a summary, the important values to collect are listed below:

1. W_f : given by Xilinx for each family [5, 6, 7].
2. F_x : obtained experimentally by placing the NOT gate in several clock region X coordinates.
3. F_{yi} : it is an offset calculated when the other parameters are known. It is the number of EBD lines to sum depending on the Y coordinate of the cell.
4. X_{adj} : the X coordinate of the cell minus the number of BRAM and DSP columns to the left of the cell. It is calculated in the code by defining different X coordinate ranges associated to an amount of columns to subtract.
5. X_{min} : obtained by looking at the first column with logic resources in the *Device* tab of the implemented design in Vivado.

References

- [1] Digilent. Pmod usbuart: Usb to uart interface, 2025. URL: <https://digilent.com/shop/pmod-usbuart-usb-to-uart-interface/>.
- [2] Digilent. Vivado board files, 2025. Accessed: 2025-02-10. URL: https://github.com/Digilent/vivado-boards/tree/master/new/board_files.
- [3] Xilinx. Soft error mitigation (sem) core, 2025. Accessed: 2025-02-10. URL: <https://www.xilinx.com/products/intellectual-property/sem.html>.
- [4] AMD. Soft error mitigation (sem) ip core, 2025. Accessed: 2025-02-11. URL: https://docs.amd.com/r/en-US/pg036_sem.
- [5] Xilinx. 7 series - sem ip - how to use the sem ip error report to look up bit error locations using essential bit data in an ebd file?, 2016. Accessed: 2024-11-29. URL: https://adaptivesupport.amd.com/s/article/67337?language=en_US.
- [6] Xilinx. Ultrascale - sem ip - how to use the sem ip error report to look up bit error locations using essential bit data in the ebd file, 2021. Accessed: 2024-11-29. URL: https://adaptivesupport.amd.com/s/article/67086?language=en_US.
- [7] Xilinx. Ultrascale+ - sem ip - how to use the sem ip error report to look up essential bit error locations using the ebd (essential bit data) file?, 2021. Accessed: 2024-11-29. URL: https://adaptivesupport.amd.com/s/article/70684?language=en_US.

A VHDL Codes

This appendix contains the VHDL codes used in the Nexys A7-100T example project.

flipflopD.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

-- D flip-flop entity
entity flipflopD is
    port( CLK : in  std_logic;
          RST : in  std_logic;
          D   : in  signed(15 downto 0);
          Q   : out signed(15 downto 0)
    );
end flipflopD;

-- Architecture with active-high synchronous reset
architecture Behavioral of flipflopD is
begin
    process(CLK) begin
        if(rising_edge(CLK)) then
            if(RST = '1') then
                Q <= (others => '0');
            else
                Q <= D;
            end if;
        end if;
    end process;
end Behavioral;
```

FIR.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity FIR is
    Port ( CLK : in  std_logic;
          RST : in  std_logic;
          X   : in  std_logic_vector(7 downto 0);  -- FIR input
          Y   : out std_logic_vector(15 downto 0)  -- FIR output
    );
end FIR;

architecture Behavioral of FIR is
    component flipflopD is
        port( CLK : in  std_logic;
              RST : in  std_logic;
              D   : in  signed(15 downto 0);
              Q   : out signed(15 downto 0)
        );
    end component;

    signal MCM0, MCM1, MCM2, MCM3, MCM4 : signed(15 downto 0);
    signal MCM5, MCM6, MCM7, MCM8, MCM9, MCM10 : signed(15 downto 0);
    signal A0, A1, A2, A3, A4, A5, A6, A7, A8, A9 : signed(15 downto 0);
    signal Q0, Q1, Q2, Q3, Q4, Q5, Q6, Q7, Q8, Q9 : signed(15 downto 0);

    -- Filter Coefficients
    constant H0 : signed(7 downto 0) := "11111111";
    constant H1 : signed(7 downto 0) := "11011010";
    constant H2 : signed(7 downto 0) := "11100101";
    constant H3 : signed(7 downto 0) := "11100010";
    constant H4 : signed(7 downto 0) := "11100110";
    constant H5 : signed(7 downto 0) := "11101111";
    constant H6 : signed(7 downto 0) := "11111010";
    constant H7 : signed(7 downto 0) := "00000101";
    constant H8 : signed(7 downto 0) := "00001100";
    constant H9 : signed(7 downto 0) := "00010001";
```

```
constant H10 : signed(7 downto 0) := "00011111";
begin
  -- Multiple Constant Multiplications
  MCM0 <= H0 * signed(X);
  MCM1 <= H1 * signed(X);
  MCM2 <= H2 * signed(X);
  MCM3 <= H3 * signed(X);
  MCM4 <= H4 * signed(X);
  MCM5 <= H5 * signed(X);
  MCM6 <= H6 * signed(X);
  MCM7 <= H7 * signed(X);
  MCM8 <= H8 * signed(X);
  MCM9 <= H9 * signed(X);
  MCM10 <= H10* signed(X);

  -- Adders
  A0 <= Q0 + MCM0;
  A1 <= Q1 + MCM1;
  A2 <= Q2 + MCM2;
  A3 <= Q3 + MCM3;
  A4 <= Q4 + MCM4;
  A5 <= Q5 + MCM5;
  A6 <= Q6 + MCM6;
  A7 <= Q7 + MCM7;
  A8 <= Q8 + MCM8;
  A9 <= Q9 + MCM9;

  -- Delay Flip-Flops
  FF0 : flipflopD port map(CLK, RST, A1, Q0);
  FF1 : flipflopD port map(CLK, RST, A2, Q1);
  FF2 : flipflopD port map(CLK, RST, A3, Q2);
  FF3 : flipflopD port map(CLK, RST, A4, Q3);
  FF4 : flipflopD port map(CLK, RST, A5, Q4);
  FF5 : flipflopD port map(CLK, RST, A6, Q5);
  FF6 : flipflopD port map(CLK, RST, A7, Q6);
  FF7 : flipflopD port map(CLK, RST, A8, Q7);
  FF8 : flipflopD port map(CLK, RST, A9, Q8);
  FF9 : flipflopD port map(CLK, RST, MCM10, Q9);
```

```
process(CLK, RST) begin
    if (rising_edge(CLK)) then
        if (RST = '1') then
            Y <= (others => '0');
        else
            Y <= std_logic_vector(A0);
        end if;
    end if;
end process;
end Behavioral;
```

inputROM.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity inputROM is
    port( CLK      : in  std_logic;
          RST      : in  std_logic;
          ADDR     : in  integer range 0 to 999;          -- ROM address
          DATA    : out std_logic_vector(7 downto 0)    -- ROM content
    );
end inputROM;

architecture Behavioral of inputROM is
    type rom_type is array (0 to 999) of std_logic_vector(7 downto 0);
    constant fir_rom : rom_type :=(x"01",x"02",x"03",x"04" ...);
    -- Some values are omitted for brevity
    -- Full VHDL code is available in the GitHub repository
begin
    process(CLK) begin
        if (rising_edge(CLK)) then
            if (RST = '1') then
                DATA <= (others => '0');
            else
                DATA <= fir_rom(ADDR);
            end if;
        end if;
    end process;
end Behavioral;
```

comparator.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity comparator is
    Port ( CLK      : in  std_logic;
          RST       : in  std_logic;
          IN1       : in  std_logic_vector(15 downto 0); -- CIRCUIT1 output
          IN2       : in  std_logic_vector(15 downto 0); -- CIRCUIT2 output
          ADDR      : out integer range 0 to 999;         -- ROM Address
          SEND      : out std_logic;                     -- Ready to send via UART
          CMP       : out std_logic_vector(7 downto 0)    -- Comparison result
    );
end comparator;

architecture Behavioral of comparator is
    signal count : integer range 0 to 999 := 999;
    signal error_flag : std_logic := '0';
begin
    process(CLK) begin
        if (rising_edge(CLK)) then
            if (RST = '1') then
                count <= 0;
                SEND <= '0';
                error_flag <= '0';
            else
                if (count < 998) then
                    count <= count + 1;
                    SEND <= '0';
                elsif (count = 998) then
                    count <= count + 1;
                    SEND <= '1';
                elsif (count = 999) then
                    SEND <= '0';
                end if;
            end if;
        end if;
    end process;
end;
```



```
        if (error_flag = '0') then
            if (IN1 /= IN2) then
                error_flag <= '1';
            end if;
        end if;
    end if;
end if;
end process;

-- Output assignment
-- 30 and 31 in hex are 0 and 1 in ASCII respectively
CMP <= x"31" when error_flag = '1' else x"30";
ADDR <= count;
end Behavioral;
```

uart_tx.vhd

```

-----
-- File Downloaded from http://www.nandland.com
-----

-- This file contains the UART Transmitter. This transmitter is able
-- to transmit 8 bits of serial data, one start bit, one stop bit,
-- and no parity bit. When transmit is complete o_TX_Done will be
-- driven high for one clock cycle.
--
-- Set Generic g_CLKS_PER_BIT as follows:
-- g_CLKS_PER_BIT = (Frequency of i_Clk)/(Frequency of UART)
-- Example: 100 MHz Clock, 115200 baud UART
-- (100000000)/(115200) = 868
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity UART_TX8 is
  generic (
    g_CLKS_PER_BIT : integer := 868      -- 115200 baud
  );
  port (
    i_Clk          : in  std_logic;
    i_TX_DV        : in  std_logic;
    i_TX_Byte      : in  std_logic_vector(7 downto 0);
    o_TX_Active    : out std_logic;
    o_TX_Serial    : out std_logic;
    o_TX_Done      : out std_logic
  );
end UART_TX8;

architecture RTL of UART_TX8 is
  type t_SM_Main is (s_Idle, s_TX_Start_Bit, s_TX_Data_Bits,
                    s_TX_Stop_Bit, s_Cleanup);
  signal r_SM_Main : t_SM_Main := s_Idle;
  signal r_Clk_Count : integer range 0 to g_CLKS_PER_BIT-1 := 0;
  signal r_Bit_Index : integer range 0 to 7 := 0;  -- 8 Bits Total
  signal r_TX_Data   : std_logic_vector(7 downto 0) := (others => '0');

```

```

    signal r_TX_Done    : std_logic := '0';
begin
    p_UART_TX : process (i_Clk)
    begin
        if rising_edge(i_Clk) then
            case r_SM_Main is
                when s_Idle =>
                    o_TX_Active <= '0';
                    o_TX_Serial <= '1';           -- Drive Line High for Idle
                    r_TX_Done    <= '0';
                    r_Clk_Count <= 0;
                    r_Bit_Index <= 0;
                    if i_TX_DV = '1' then
                        r_TX_Data <= i_TX_Byte;
                        r_SM_Main <= s_TX_Start_Bit;
                    else
                        r_SM_Main <= s_Idle;
                    end if;
                    -- Send out Start Bit. Start bit = 0
                when s_TX_Start_Bit =>
                    o_TX_Active <= '1';
                    o_TX_Serial <= '0';
                    -- Wait g_CLKS_PER_BIT-1 clock cycles for start bit to finish
                    if r_Clk_Count < g_CLKS_PER_BIT-1 then
                        r_Clk_Count <= r_Clk_Count + 1;
                        r_SM_Main    <= s_TX_Start_Bit;
                    else
                        r_Clk_Count <= 0;
                        r_SM_Main    <= s_TX_Data_Bits;
                    end if;
                    -- Wait g_CLKS_PER_BIT-1 clock cycles for data bits to finish
                when s_TX_Data_Bits =>
                    o_TX_Serial <= r_TX_Data(r_Bit_Index);
                    if r_Clk_Count < g_CLKS_PER_BIT-1 then
                        r_Clk_Count <= r_Clk_Count + 1;
                        r_SM_Main    <= s_TX_Data_Bits;
                    else
                        r_Clk_Count <= 0;

```

```

        -- Check if we have sent out all bits
        if r_Bit_Index < 7 then
            r_Bit_Index <= r_Bit_Index + 1;
            r_SM_Main    <= s_TX_Data_Bits;
        else
            r_Bit_Index <= 0;
            r_SM_Main    <= s_TX_Stop_Bit;
        end if;
    end if;

    -- Send out Stop bit.  Stop bit = 1
    when s_TX_Stop_Bit =>
        o_TX_Serial <= '1';
        -- Wait g_CLKS_PER_BIT-1 clock cycles for Stop bit to finish
        if r_Clk_Count < g_CLKS_PER_BIT-1 then
            r_Clk_Count <= r_Clk_Count + 1;
            r_SM_Main    <= s_TX_Stop_Bit;
        else
            r_TX_Done    <= '1';
            r_Clk_Count <= 0;
            r_SM_Main    <= s_Cleanup;
        end if;
        -- Stay here 1 clock
        when s_Cleanup =>
            o_TX_Active <= '0';
            r_TX_Done    <= '1';
            r_SM_Main    <= s_Idle;
        when others =>
            r_SM_Main <= s_Idle;
        end case;
    end if;
end process p_UART_TX;

o_TX_Done <= r_TX_Done;
end RTL;

```

my_design.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity my_design is
    Port ( CLK      : in  std_logic;
          RST      : in  std_logic;
          S_OUT    : out std_logic  -- Serial output
        );
end my_design;

architecture Behavioral of my_design is
    component inputROM is
        port( CLK      : in  std_logic;          -- Clock
              RST      : in  std_logic;          -- Reset
              ADDR     : in  integer range 0 to 999; -- Address
              DATA    : out std_logic_vector(7 downto 0) -- Output Data
        );
    end component;

    component FIR is
        Port ( CLK : in  std_logic;
              RST : in  std_logic;
              X   : in  std_logic_vector(7 downto 0); -- FIR input
              Y   : out std_logic_vector(15 downto 0) -- FIR output
        );
    end component;

    component comparator is
        Port ( CLK : in  std_logic;
              RST : in  std_logic;
              IN1  : in  std_logic_vector(15 downto 0); -- CIRCUIT1 output
              IN2  : in  std_logic_vector(15 downto 0); -- CIRCUIT2 output
              ADDR : out integer range 0 to 999;         -- ROM Address
              SEND : out std_logic;                      -- Ready to send via UART
              CMP  : out std_logic_vector(7 downto 0)    -- Comparison result
        );
    end component;
```

```
component UART_TX8 is
  generic (
    g_CLKS_PER_BIT : integer := 868      -- 115200 baud
  );
  port (
    i_Clk      : in  std_logic;
    i_TX_DV    : in  std_logic;
    i_TX_Byte  : in  std_logic_vector(7 downto 0);
    o_TX_Active : out std_logic;
    o_TX_Serial : out std_logic;
    o_TX_Done   : out std_logic
  );
end component;

signal ADDR : integer range 0 to 999;
signal ROM_OUT, CMP : std_logic_vector(7 downto 0);
signal FIR1_OUT, FIR2_OUT : std_logic_vector(15 downto 0);
signal SEND : std_logic;
begin
  MY_ROM : inputROM port map(
    CLK => CLK,
    RST => RST,
    ADDR => ADDR,
    DATA => ROM_OUT
  );
  CIRCUIT1 : FIR port map(
    CLK => CLK,
    RST => RST,
    X  => ROM_OUT,
    Y  => FIR1_OUT
  );
  CIRCUIT2 : FIR port map(
    CLK => CLK,
    RST => RST,
    X  => ROM_OUT,
    Y  => FIR2_OUT
  );
```

```
MY_CMP : comparator port map(  
    CLK  => CLK,  
    RST  => RST,  
    IN1  => FIR1_OUT,  
    IN2  => FIR2_OUT,  
    ADDR => ADDR,  
    SEND => SEND,  
    CMP  => CMP  
);  
MY_UART : UART_TX8 generic map(g_CLKS_PER_BIT => 868)  
port map(  
    i_Clk      => CLK,  
    i_TX_DV    => SEND,  
    i_TX_Byte  => CMP,  
    o_TX_Active => open,  
    o_TX_Serial => S_OUT,  
    o_TX_Done  => open  
);  
end Behavioral;
```

top.vhd

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity top is
    Port ( clk                : in    std_logic;
          status_initialization : out  std_logic;
          status_observation    : out  std_logic;
          status_correction     : out  std_logic;
          status_uncorrectable  : out  std_logic;
          monitor_tx            : out  std_logic;
          monitor_rx            : in    std_logic;
          S_OUT                 : out  std_logic    -- DUT serial output
        );
end top;

architecture Behavioral of top is
    component sem_0_sem_example is
        port ( clk                : in    std_logic;
              status_heartbeat    : out  std_logic;
              status_initialization : out  std_logic;
              status_observation   : out  std_logic;
              status_correction    : out  std_logic;
              status_classification : out  std_logic;
              status_injection     : out  std_logic;
              status_essential     : out  std_logic;
              status_uncorrectable : out  std_logic;
              inject_strobe        : in    std_logic;
              inject_address       : in    std_logic_vector(39 downto 0);
              monitor_tx          : out  std_logic;
              monitor_rx          : in    std_logic;
              icap_clk_out        : out  std_logic
            );
    end component;
end component;

```



```
component my_design is
  Port ( CLK      : in  std_logic;
        RST      : in  std_logic;
        S_OUT    : out std_logic  -- Serial output
      );
end component;

signal icap_clk_out, injection_done : std_logic;
begin
  SEM : sem_0_sem_example port map(
    clk                => clk,
    status_heartbeat    => open,
    status_initialization => status_initialization,
    status_observation  => status_observation,
    status_correction   => status_correction,
    status_classification => open,
    status_injection    => injection_done,
    status_essential    => open,
    status_uncorrectable => status_uncorrectable,
    inject_strobe       => '0',
    inject_address      => (others => '0'),
    monitor_tx          => monitor_tx,
    monitor_rx          => monitor_rx,
    icap_clk_out        => icap_clk_out
  );

  DUT : my_design port map(
    CLK  => icap_clk_out,
    RST  => injection_done,
    S_OUT => S_OUT
  );
end Behavioral;
```

B SPI Flash Memory Programming

This appendix explains how to program the Nexys A7-100T SPI flash memory.

1. **Configure Vivado project:** add the following lines in the `design_constraints.xdc` file and generate the bitstream.

```
set_property BITSTREAM.CONFIG.SPI_BUSWIDTH 4 [current_design]
set_property CONFIG_MODE SPIx4 [current_design]
```

2. **Convert the bitstream into PROM format:** open *TCL Console* in Vivado and type the following TCL commands.

- For Linux users:

```
write_cfgmem -format mcs -interface spix4 -size 16 -loadbit "up 0x0
./your_path/project_1.runs/impl_1/top.bit" -file main.mcs
```

- For Windows users:

```
write_cfgmem -format mcs -interface spix4 -size 16 -loadbit "up 0x0
C:/your_path/project_1.runs/impl_1/top.bit" -file C:/your_path/main.mcs
```

Note: change “your path” by your Vivado project path.

3. **Programming the SPI flash memory:**

- (a) Click in *Open Hardware Manager* > *Open Target* > *Auto Connect*
- (b) Right click on the target (i.e. xc7a100t_0) and select *Add Configuration Memory Device*
- (c) Select s25fl128sxxxxxx0 for the Nexys A7-100T
- (d) Right click in the configuration memory device and select *Program Configuration Memory Device*
- (e) Select the configuration file `main.mcs` generated in step 2
- (f) Choose the option *Entire Configuration Memory Device, Erase, Program, and Verify*, and then click *OK*
- (g) Set the JP1 jumper in the Nexys A7-100T board in QSPI position
- (h) Reset the FPGA to load the design from flash memory every time a power-cycle or a reconfiguration is performed