

Getting started with FPGA control development

PN159 | Posted on June 2, 2021 | Updated on May 7, 2025



Benoît STEINMANN

Software Team Leader

imperix • in

Table of Contents

- [Presentation of the FPGA control starter template](#)
- [Creating the FPGA control starter template](#)
 - [Downloading the required sources](#)
 - [Starting a new imperix sandbox project](#)
 - [Adding the AXI4-Stream interface](#)
- [Using the SBIO_BUS](#)
 - [SBIO modules](#)
 - [SBIO interconnect](#)
- [How the AXI4-Stream interface operates](#)
 - [Retrieving analog measurement with the M_AXIS_ADC interfaces](#)
 - [Exchanging data using M_AXIS_CPU2FPGA and S_AXIS_FPGA2CPU](#)
 - [Getting the sample time Ts](#)
 - [Using reset signals](#)
- [Step-by-step “hello world” example](#)
 - [CPU-side implementation](#)
 - [FPGA-side implementation](#)
 - [Loading the bitstream into the imperix controller](#)
 - [Experimental validation](#)
- [Going further](#)
 - [Testing M_AXIS_Ts](#)
 - [Using the USR pins](#)
 - [Additional tutorials](#)

This note explains how to get started with the implementation of power converter control algorithms in the FPGA of [imperix power electronic controllers](#). The benefit of offloading all or parts of the computations from the CPU to the FPGA is that it often results in much faster closed-loop control systems.

First, the **FPGA control starter template** is presented and a tutorial on how to create this template is provided. Then, the reader will learn how to **retrieve ADC results** from the FPGA as well as to **exchange data with the CPU**. The page ends with a simple hello-world example illustrating all the keys steps of FPGA control implementation.

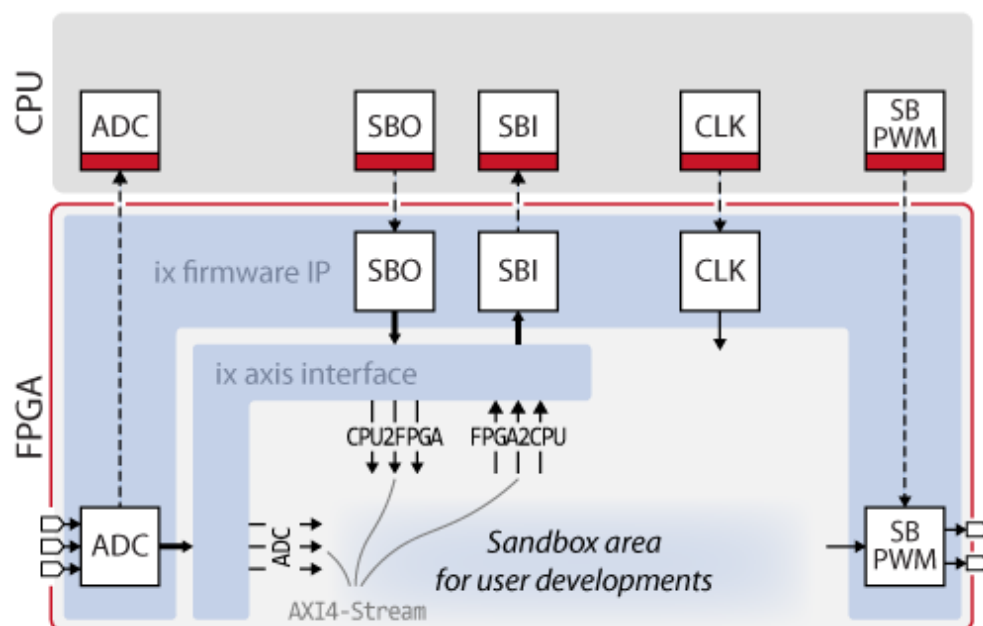
This page is the first of a 3-part tutorial explaining step-by-step how to implement the closed-loop control of a buck converter in FPGA without using VHDL or Verilog. The second note explains how to generate a [PWM modulator in FPGA](#) using the Simulink blockset Xilinx System Generator or MATLAB HDL Coder. The last note shows how to create the [PI-based current control using high-level synthesis](#) with Xilinx Model Composer (Simulink blockset) or Xilinx Vitis HLS (C++).

To find all FPGA-related notes, please visit [FPGA development homepage](#).

Presentation of the FPGA control starter template

The FPGA control starter template allows for easy integration of custom FPGA-based control algorithms in the **sandbox** area of the [B-Box RCP](#) or the [B-Board PRO](#). As shown in the image below, it consists of:

- the obfuscated “**imperix firmware IP**” which contains the FPGA logic required to operate imperix controllers (documented in [PN116](#)), and
- the “**ix axis interface**” module which provides easy-to-use AXI4-Stream interfaces to exchange data with the user logic.



The provided *AXI4-Stream interface* module connects to the data interfaces and timing signals of the *imperix firmware IP*, interprets these signals, and converts them into much more user-friendly **AXI4-Stream (AXIS)** interfaces (ADC, CPU2FPGA and FPGA2CPU). The AXI4-Stream protocol is a widely used standard to interconnect

components that exchange data. This means that the provided template can directly be connected to a wide range of [Xilinx-provided IPs](#) or to user-made algorithms developed using High-Level Synthesis (HLS) design tools such as [Vitis HLS](#) (C++) or [Model Composer](#) (Simulink).

This *AXI4-Stream interface module* is written in VHDL. It is provided as a starting point and will meet the need of most applications. However, if required, it can be edited by the user to add extra input/outputs, rename them, change their data sizes, etc.

Creating the FPGA control starter template

This tutorial requires Xilinx Vivado which is available at no-cost.

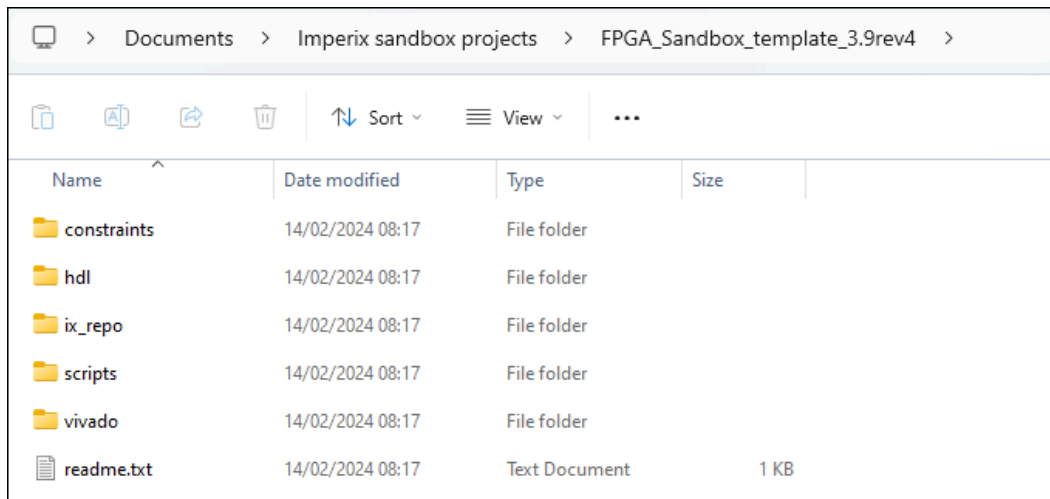
The [Vivado Design Suite installation](#) guide explains how to download and install Xilinx Vivado for free.

Downloading the required sources

All the required sources are packed into the **FPGA_Sandbox_template** archive which can be downloaded from the [Download and update imperix IP](#) page.

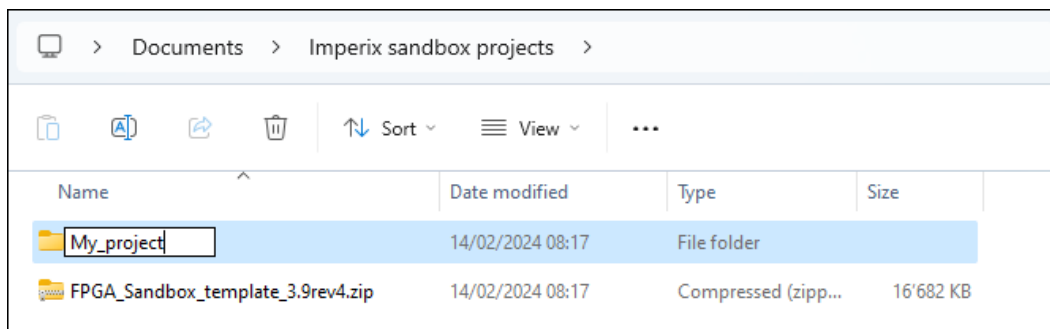
Since version 3.9, the template file structure is the following

- **constraints**
 - **sandbox_pins_*.xdc**: top-level ports to a physical package pin assignation
- **hdl**
 - **AXIS_interface.vhd**: AXI4-Stream interface module, presented on the next chapter
 - **user_cb_pwm.vhd**: simple carrier-based modulator, described in [TN141](#)
- **ix_repo**: Vivado IP Catalog repository. Contains the imperix firmware IP and its interfaces.
- **scripts**:
 - **create_project.bat**: launch Vivado and call create_project.tcl
 - **create_project.tcl**: contains the TCL commands that create and configure the sandbox Vivado project
- **vivado**: contains the Vivado projects generated by the create_project script

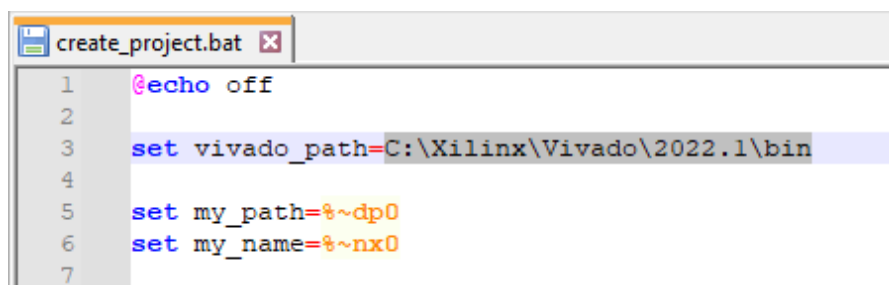


Starting a new imperix sandbox project

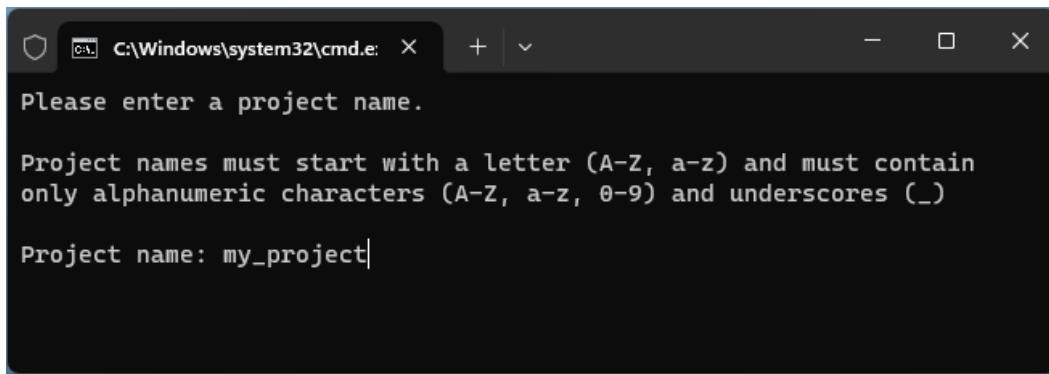
1. Download **FPGA_Sandbox_template_*.zip**
2. Unzip it and save the content somewhere on the PC
3. Rename the folder to something more explicit



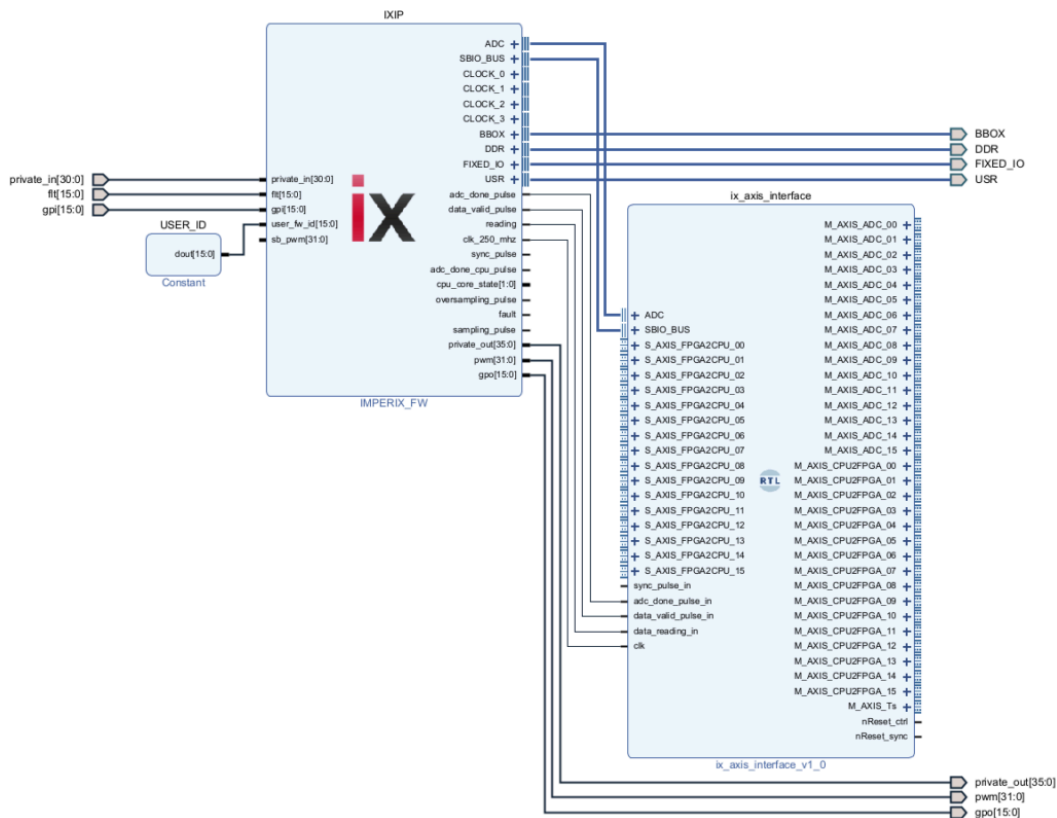
4. Open **scripts/create_project.bat** using a text editor
5. Set the vivado_path variable to match the Vivado version installed on the PC



6. Double click on **scripts/create_project.bat**
Windows Defender SmartScreen may display a warning pop-up. Simply click *More info* then *Run anyway*.
7. Enter a project name and click enter



The Vivado sandbox project will be created and configured, its block design is shown below.



imperix sandbox template Vivado block design

If, for some reason, the script is not working properly then the Vivado project can be created manually by following the procedure below:

Manually creating Vivado imperix sandbox project

1. Open **Vivado**.
2. Click **Create Project**.
3. Chose a name and a location.
4. Select project type **RTL Project** and check the box **Do not specify sources at this time**.
5. Select the part named **xc7z030fbg676-3**.
6. Hit **Finish**. The project should open.

7. Go to the **IP Catalog**, right-click on **Vivado Repository**, hit **Add repository...**
 Select <my_project>/ix_repo/
 The *IMPERIX_FW* IP, *clock_gen*, and *user_regs* interfaces should be found. Press **OK**.
8. Click on **Create block design**, name it “top” and click **OK**.
9. Open the freshly created block design, do a right-click in a blank area of the design, select **Add IP...** and search for “IMPERIX_FW” and hit ENTER.
10. Keep the [Ctrl] key pressed and select the IP
 pins flt, gpi, private_in, DDR, FIXED_IO, BBOS, USR,
 gpo, pwm and private_out. Hit [Ctrl+T] to create top-level ports.
11. By default Vivado adds “_0” after each port name. Remove the “_0” from every port. For that, click on each port and change the **name** property in the **Block Pin Properties** block (appearing on the left of the diagram by default). For instance, change “flt_0[15:0]” to “flt[15:0]”.
12. The user_fw_id input may be used to identify the firmware version. We recommend instantiating a *Constant* IP (Right-click, **Add IP...**, search for *Constant*) to give an identification number to the design. To change the constant width, double-click on the **Constant** block and set the **Const Width** to 16.
13. Go to the **Sources** tab, right-click on the block design file (top.bd) and select **Create HDL Wrapper...**
 In the dialog box choose **Let Vivado manage wrapper and auto-update** and hit **OK**.
14. Right-click on the **Design Sources** folder
 Choose **Add Sources...**
 Check **Add or create constraints**
 Click on **Add Files**
 Select <my_project>/sandbox_pins.xdc
 Uncheck **Copy constraints files into project**
 Hit **Finish**

From this point the project is synthesizable.

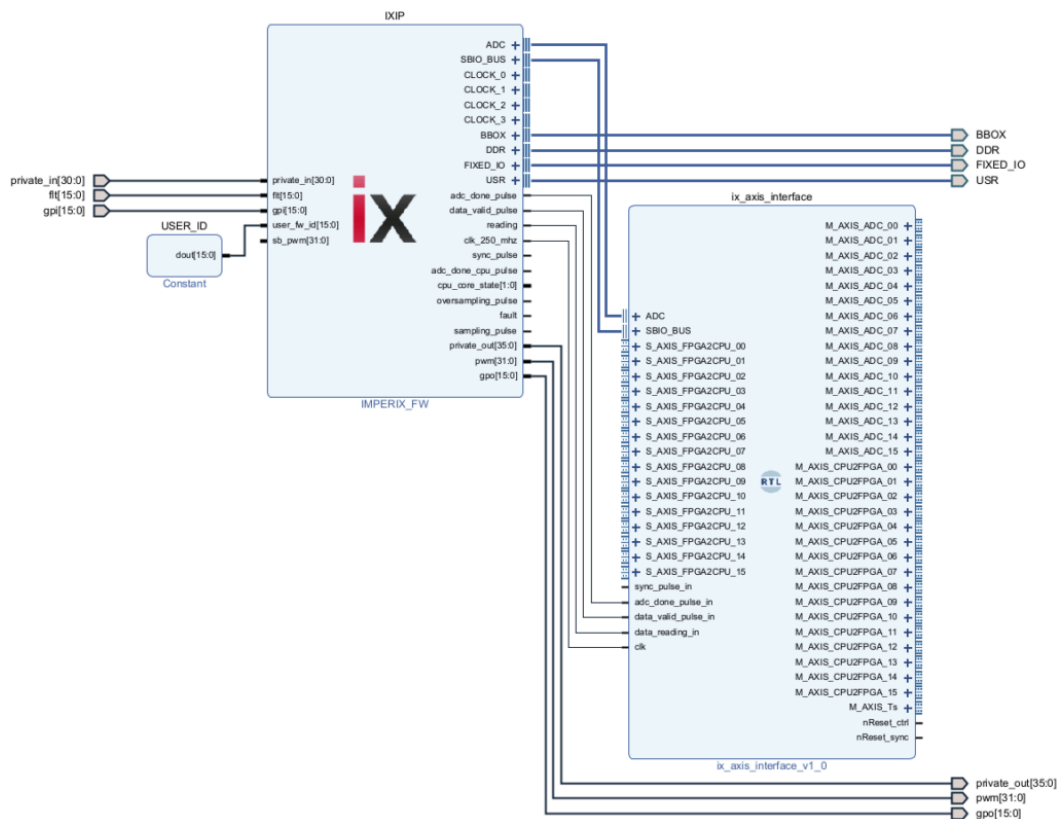
Adding the AXI4-Stream interface

Adding the *AXI4-Stream interface* is optional. It is useful if the FPGA control design uses AXI4-Stream interfaces.

1. Right-click on **Design Sources**
 Choose **Add Sources....**
 Check **Add or create design sources**.
2. Press **Add Files**. Go to your repository and
 select <my_project>/AXIS_interface.vhd.

We recommend unchecking “Copy sources into project” and working directly from the files in the folder <my_project>/hdl/ so the sources can be shared across multiple projects. Press **Finish** and wait for the update to finish.

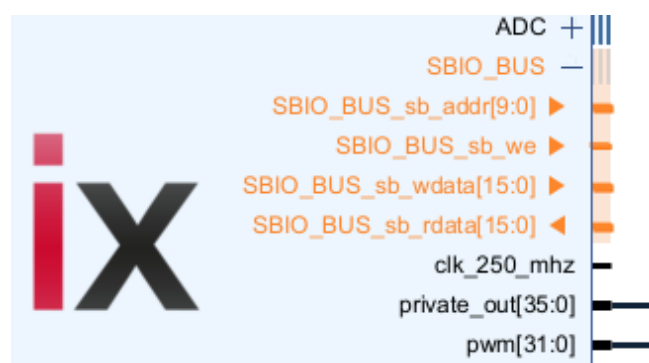
3. Right-click somewhere in the block design and choose **Add module...** and select the *ix_axis_interface* module. Alternatively, the file listed in the *Design Sources* can be drag and dropped it on the block diagram.
4. Connect the pins as follows (to get a clear layout, change “Default View” to “No Loops” in the top bar of the Diagram block, then right-click somewhere in the block design and press **Regenerate Layout**)



imperix sandbox template Vivado block design

Using the SBIO_BUS

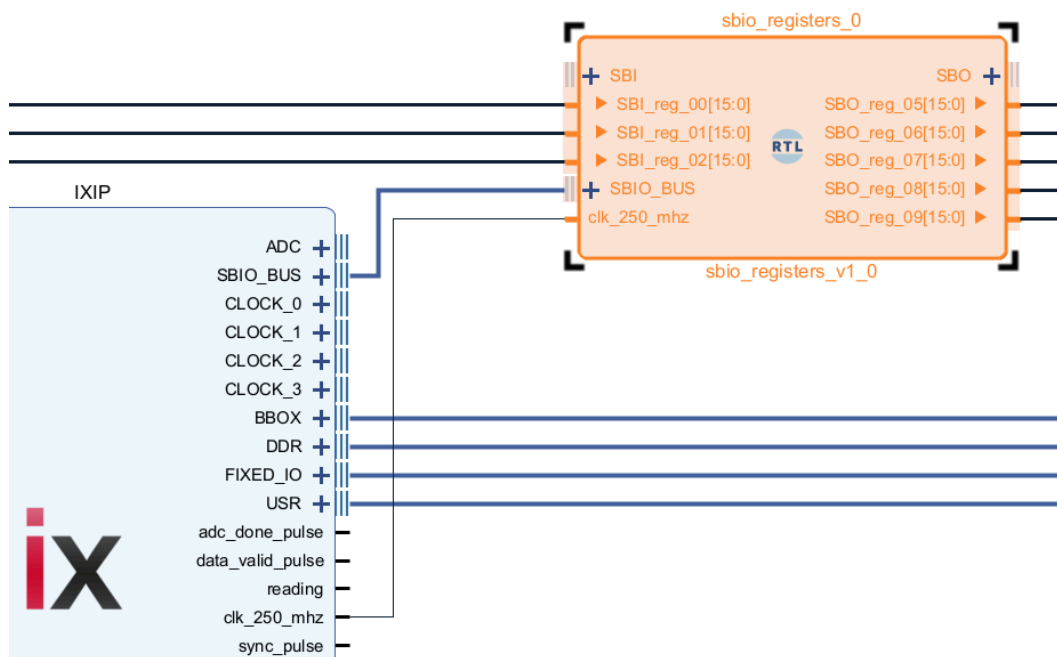
The **SBIO_BUS** (SandBox IO bus) is a 16-bit memory-mapped bus allowing the CPU to addressing up to 1024 register in the FPGA.



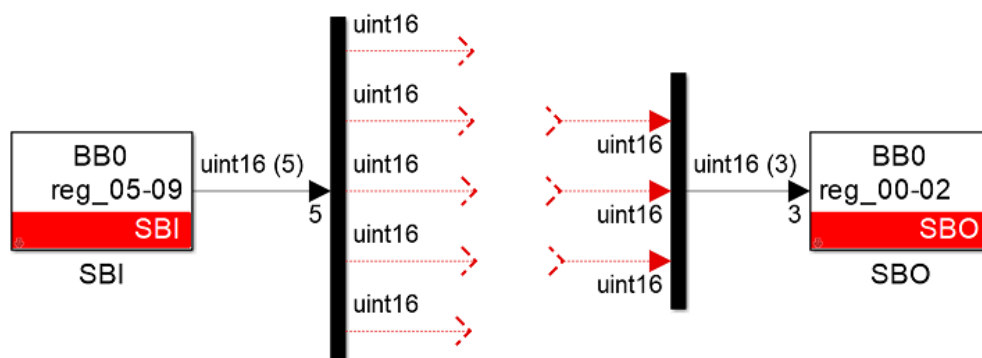
SBIO modules

Two **SBIO modules** are provided in the template:

- the **AXI4-Stream interface** (AXIS_interface.vhd) which is described in the next section,
- the **SBIO registers** (sbio_registers.vhd) shown below, which provides 16-bit registers that can easily be connected to user logic.

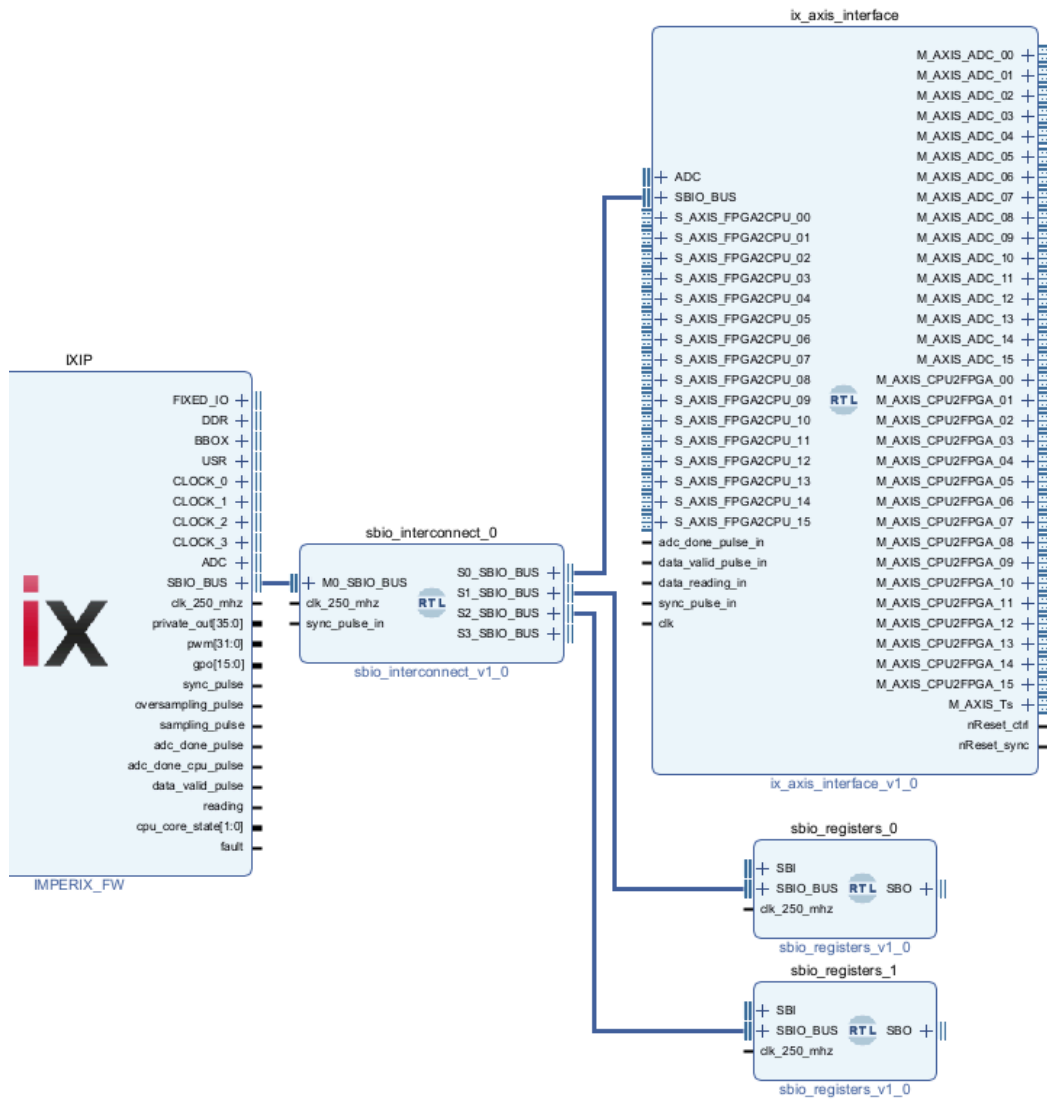


The user can read and write from this bus using the [SBO](#) and [SBI](#) blocks as shown below. During execution, SBIs are read before each CPU task executions and SBOs written at the end of each CPU task execution.



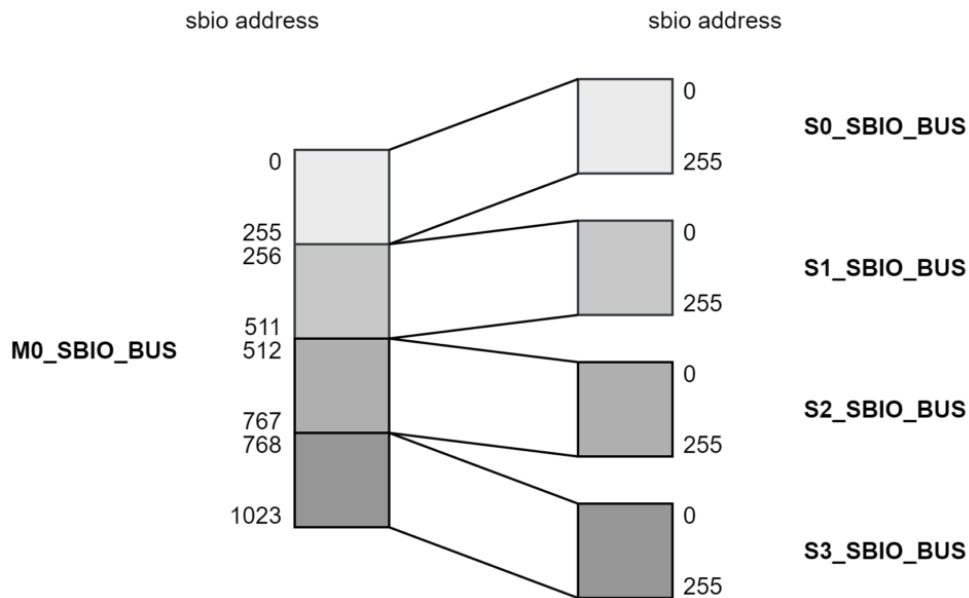
SBIO interconnect

The **SBIO interconnect** increases the number of SBIO_BUS interfaces, allowing to connect multiple SBIO modules as illustrated below.



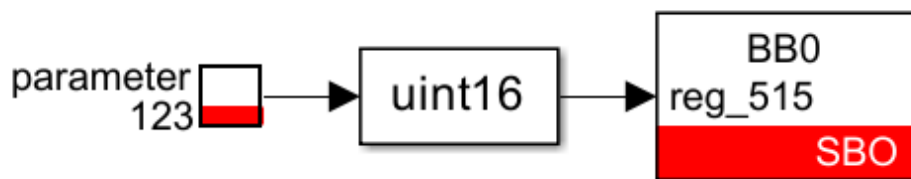
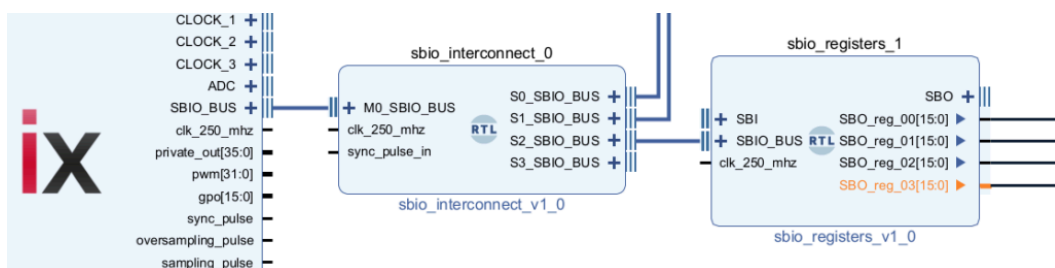
sbio_interconnect

The address mapping of the SBIO interconnect is shown below, it divides the SBIO addressable range in 4 smaller areas.



sbio_interconnect memory mapping

As an example, to write to **SBO_reg_03** of an sbio_registers block connected to **S2_SBio_Bus**, The user has to use an SBO block to register number $512+3=515$.



How the AXI4-Stream interface operates

This section focuses on the *AXI-Stream interface* module (ix_axis_interface). For further information on the *imperix firmware* IP (IMPERIX_FW) please refer to the [imperix firmware IP product guide](#).

If required, `AXIS_interface.vhd` can easily be edited to improve the readability, by renaming the interfaces and removing the unused ones for instance. In this example, the interface is kept as it is.

Retrieving analog measurement with the M_AXIS_ADC interfaces

The *Master AXI4-Stream* interfaces **M_AXIS_ADC_00** to **M_AXIS_ADC_15** correspond to the 16 analog inputs of the imperix device.

They return the raw **16-bit signed integer** result from the ADC each time conversion results are available. Consequently, users should manually perform the data-type conversion and apply correct gains in their FPGA projects, to transform the acquired value in its physical unit. To learn how to compute this gain, please refer to the last section of the [ADC](#) page.

The ADC sampling frequency can only be configured using the [CONFIG](#) block. ADC acquisition can not be triggered from within the FPGA.

Exchanging data using M_AXIS_CPU2FPGA and S_AXIS_FPGA2CPU

The *Master AXI4-Stream* interfaces **M_AXIS_CPU2FPGA** and the *Slave AXI4-Stream* interfaces **S_AXIS_FPGA2CPU** serve to exchange **32-bit** data between the CPU code and the FPGA.

To read/write values on the FPGA2CPU/CPU2FPGA ports, the user can download the Simulink model from the step-by-step hello world section below and re-use the following blocks



write a float value from the CPU to the FPGA

The provided template uses the following mapping between the 16-bit SBI/SBO registers and the 32-bit AXI4-Stream interfaces:

bit number	31	16	15	0	bit number	31	16	15	0
CPU2FPGA_00	SBO_01		SBO_00		FPGA2CPU_00	SBI_01		SBI_00	
CPU2FPGA_01	SBO_03		SBO_02		FPGA2CPU_01	SBI_03		SBI_02	
CPU2FPGA_02	SBO_05		SBO_04		FPGA2CPU_02	SBI_05		SBI_04	
CPU2FPGA_03	SBO_07		SBO_06		FPGA2CPU_03	SBI_07		SBI_06	
CPU2FPGA_04	SBO_09		SBO_08		FPGA2CPU_04	SBI_09		SBI_08	
CPU2FPGA_05	SBO_11		SBO_10		FPGA2CPU_05	SBI_11		SBI_10	
CPU2FPGA_06	SBO_13		SBO_12		FPGA2CPU_06	SBI_13		SBI_12	
CPU2FPGA_07	SBO_15		SBO_14		FPGA2CPU_07	SBI_15		SBI_14	
CPU2FPGA_08	SBO_17		SBO_16		FPGA2CPU_08	SBI_17		SBI_16	
CPU2FPGA_09	SBO_19		SBO_18		FPGA2CPU_09	SBI_19		SBI_18	
CPU2FPGA_10	SBO_21		SBO_20		FPGA2CPU_10	SBI_21		SBI_20	
CPU2FPGA_11	SBO_23		SBO_22		FPGA2CPU_11	SBI_23		SBI_22	
CPU2FPGA_12	SBO_25		SBO_24		FPGA2CPU_12	SBI_25		SBI_24	
CPU2FPGA_13	SBO_27		SBO_26		FPGA2CPU_13	SBI_27		SBI_26	
CPU2FPGA_14	SBO_29		SBO_28		FPGA2CPU_14	SBI_29		SBI_28	
CPU2FPGA_15	SBO_31		SBO_30		FPGA2CPU_15	SBI_31		SBI_30	

If the user chooses to write a single-precision floating-point data on the AXI4-Stream interfaces **M_AXIS_CPU2FPGA_00**, then he has to:

1. use a [MATLAB Function block](#) to transform a *single* value into two *uint16* values (see code below)
2. and then use the [SBO](#) block to send these two *uint16* values to SBO_reg_00 and SBO_reg_01 (CPU2FPGA_00).

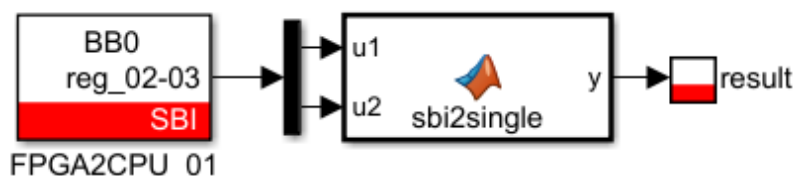


```
function [y1,y2] = single2sbo(u)
    temp = typecast(single(u),'uint16');
    y1 = temp(1);
    y2 = temp(2);
```

Code language: Matlab (matlab)

And if he wishes to read a result from the FPGA to the CPU (still in single-precision floating-point format) using **S_AXIS_FPGA2CPU_01**, then he has to:

1. use the [SBI](#) block to retrieve the two *uint16* values from SBO_reg_02 and SBO_reg_03 (FPGA2CPU_01)
2. and then use a [MATLAB Function block](#) to transform these two *uint16* values into a *single* value (see code below).



```
function y = sbi2single(u1,u2)
    y = single(0); % fix simulink bug: force compiled size of output
    y = typecast([uint16(u1) uint16(u2)], 'single');
```

Code language: Matlab (matlab)

Getting the sample time Ts

The **M_AXIS_Ts** interface provides the sample period in nanoseconds in a 32-bit unsigned integer format. This signal may be used, for instance, by the integrators of PI controllers. This value is measured by counting the time difference between two *adc_done_pulse*.

Using reset signals

The AXI4-Stream module also provides two reset signals:

- **nReset_sync**: this reset signal is activated each time the user code is loaded through Cockpit. It can be used as a standard reset signal.
- **nReset_ctrl**: this reset is triggered from the CPU through *SBO_reg_63* using a [core state](#) block. Its intended use is, for instance, to reset the PI controller integrator when the converter is not operating (when the PWM outputs are disabled).

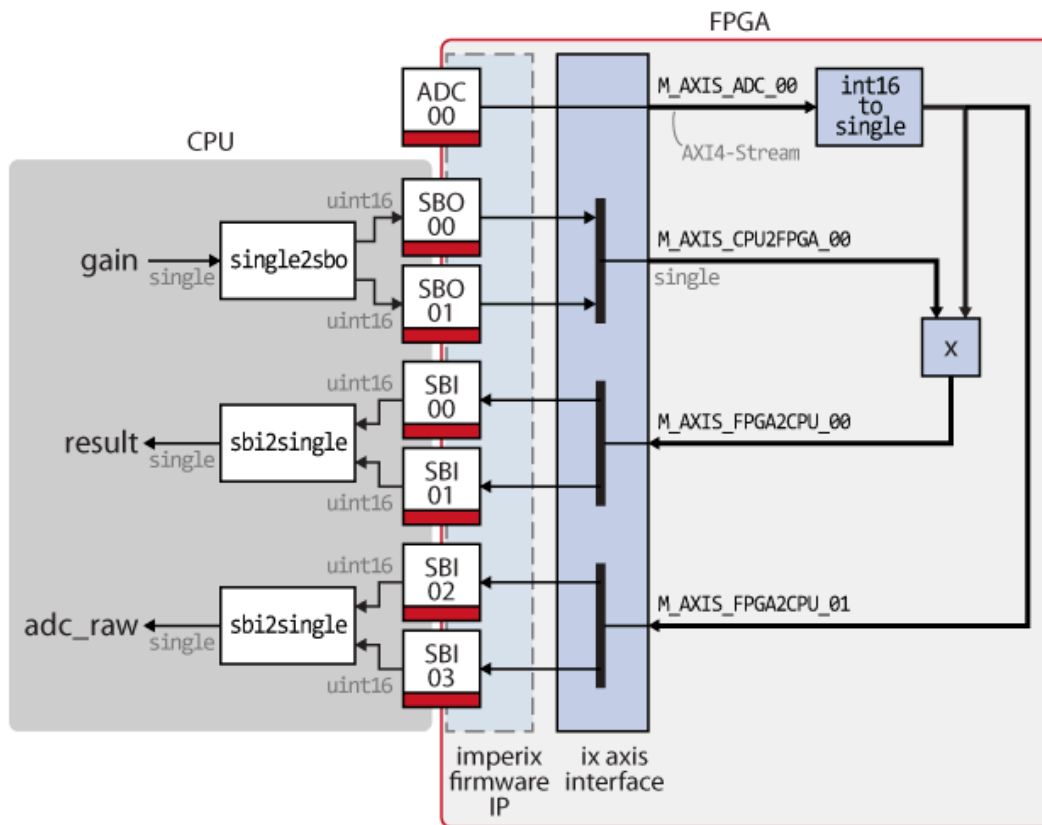
Both signals are active-low and activated for 4 periods of *clk_250_mhz*.

Step-by-step “hello world” example

This simple “hello world” example serves to showcase the complete FPGA development workflow on Vivado.

As illustrated on the image below, it does the following:

1. From the CPU, a *gain* value (single-precision) is transferred to the FPGA using the **CPU2FPGA_00** interface (SBO_00 and SBO_01).
2. In the FPGA, the data coming from the **ADC_00** interface is converted into a *single* value.
3. The ADC value is then multiplied by the *gain*.
4. Finally, the multiplication result is sent back to the CPU through **FPGA2CPU_00**.
5. The *raw value* of the ADC is also sent to the CPU using **FPGA2CPU_01**.



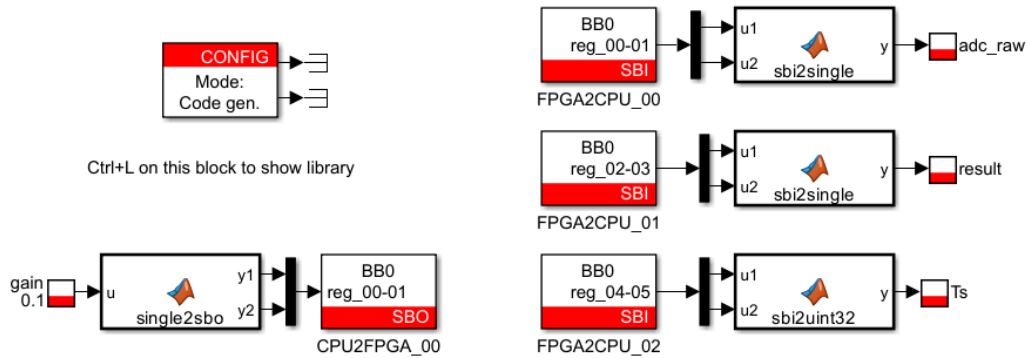
The FPGA logic will be implemented using exclusively readily available Xilinx Vivado IP, namely:

- **Floating-point IP** configured for *Fixed-to-float* operation to convert an int16 to a single-precision floating-point value.
- **Floating-point IP** configured for the *Multiply* operation to multiply two single-precision floating-point values.
- **AXI4-Stream Broadcast IP** to duplicate the output of the *int16 to single* block.

Other useful Xilinx Vivado IPs are listed in the [AXI4-Stream IPs from Xilinx](#) page. The user can also implement his own IP blocks, either using directly VHDL or Verilog, or high-level design tools such as [Vitis HLS](#) (C++) or [Model Composer](#) (Simulink).

CPU-side implementation

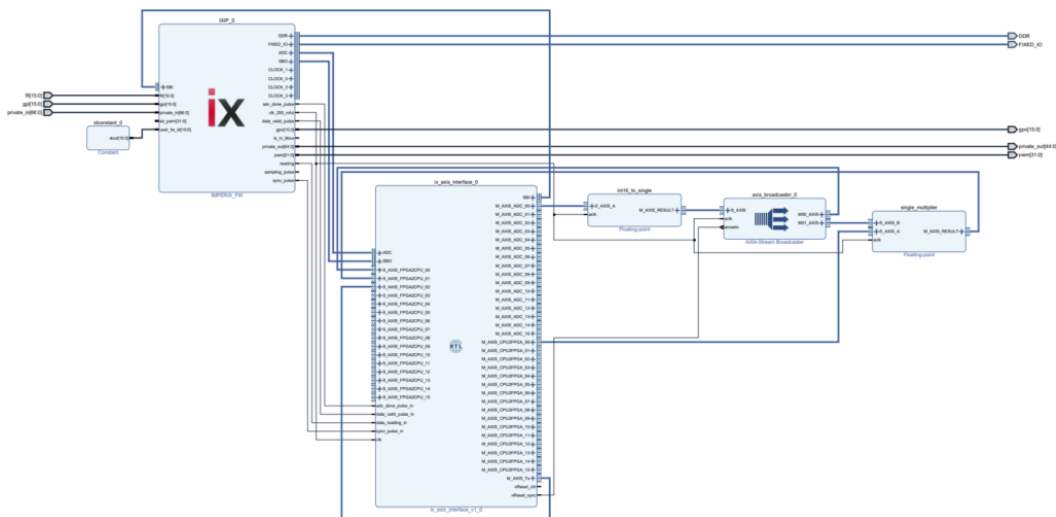
The CPU-side code provided below has been implemented using Simulink and the imperix [ACG SDK](#). To make these variables available during run-time, the *gain* is set using a [tunable parameter](#) and the *adc_raw* and *result* are read using [probes](#). These 3 values are encoded as **single** (32-bit single-precision floating-point). The MATLAB Function blocks *single2sbo* and *sbi2single* allow to easily map *single* values to [SBI](#) and [SBO](#) blocks.



[Click to download PN159_Getting_Started_With_FPGA.slx](#)

FPGA-side implementation

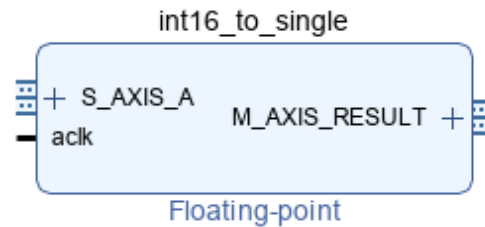
As mentioned earlier, only standard Xilinx Vivado IP blocks are used to implement the algorithm in this example. The **PN159_vivado_design.pdf** file below shows the full Vivado FPGA design. Here are the step-by-step instructions to reproduce it.



[Click to download PN159_vivado_design.pdf](#)

1. Add a block to convert the int16 data of ADC_00 to a single-precision floating-point

1. Right-click somewhere in the block design and choose **Add IP...**
2. Search for the *Floating-point* IP, drag-and-drop it on the diagram.
3. Rename it as *int16_to_single*.
4. Double-click on the *int16_to_single* block. In the pop-up window, select the **Fixed-to-float** operation, change Auto to Manual and set the precision type to Custom. Then, set the integer width to 16, and select **Single** as precision for the result.



Operation Selection Precision of Inputs Precision of Result

A Precision Type Custom

Integer Width 16 [1 - 64]

Fraction Width 0 [0 - 48]

Total Width : 16

Operation Selection Precision of Inputs Precision of Result

Result Precision Type

Please select floating-point precision

☒ Single ☐ Double ☐ Custom

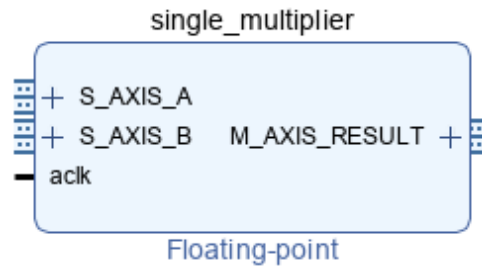
Exponent Width 8 [0 - 64]

Fraction Width 24 [0 - 64]

Total Width : 32

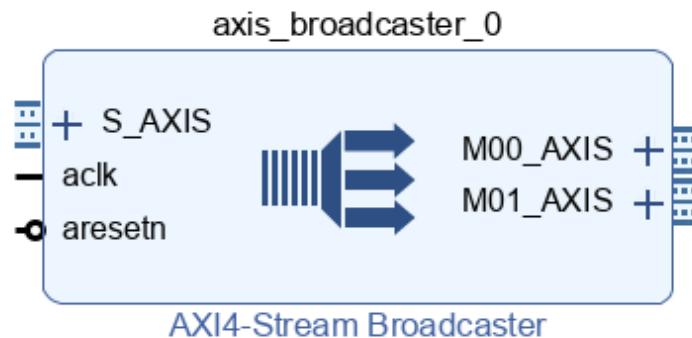
2. Add the multiplier block

1. Add another *Floating-point* IP and rename it as *single_multiplier*.
2. Double-click on the block.
3. Select the **Multiply** operation and **Single** input.



3. Broadcast one stream to two streams

1. Right-click somewhere in the block design and choose **Add IP...**
2. Search for the *AXI4-Stream Broadcaster*, drag it and drop it on the diagram.
3. Keep all options to *auto*.



4. Connect all the blocks as follow:

- **M_AXIS_ADC_00** to **S_AXIS_A** of *int16_to_single*
- **M_AXIS_RESULT** of *int16_to_single* to **S_AXIS** of *axis_boradcaster_0*
- **M00_AXIS** of *axis_broadcaster_0* to **S_AXIS_FPGA2CPU_00**
- **M01_AXIS** of *axis_broadcaster_0* to **S_AXIS_B** of *single_multiplier*
- **M_AXIS_CPU2FPGA_00** to **S_AXIS_A** of *single_multiplier*
- **M_AXIS_RESULT** of *single_multiplier* to **S_AXIS_FPGA2CPU_01**
- **clk_250_mhz** to all **aclk** inputs
- **nReset_sync** to **aresetn** of *axis_broadcaster_0*

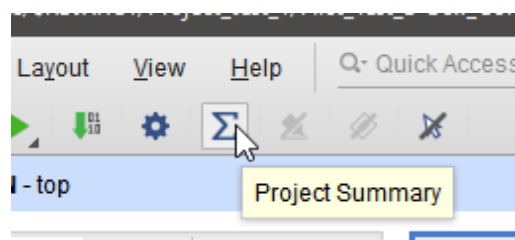
5. Finally, the design can be synthesized and the bitstream generated.

Click **Generate bitstream**. It will launch the synthesis, implementation and bitstream generation.

6. Always make sure that your design meets the timing requirements!

This information is available from the Project Summary

To learn more please visit the Xilinx documentation on [Timing Closure](#).

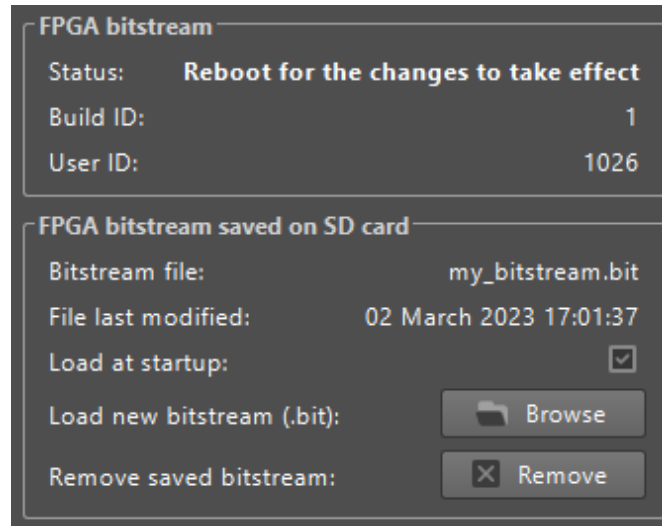


Opening the Vivado Project Summary

7. If the timing requirement are met, click on **File** → **Export** → **Export Bitstream File...** to save the bitstream somewhere on the computer.

Loading the bitstream into the imperix controller

Using [imperix Cockpit](#), the bitstream is loaded into the imperix controller device from the **target configuration** window.

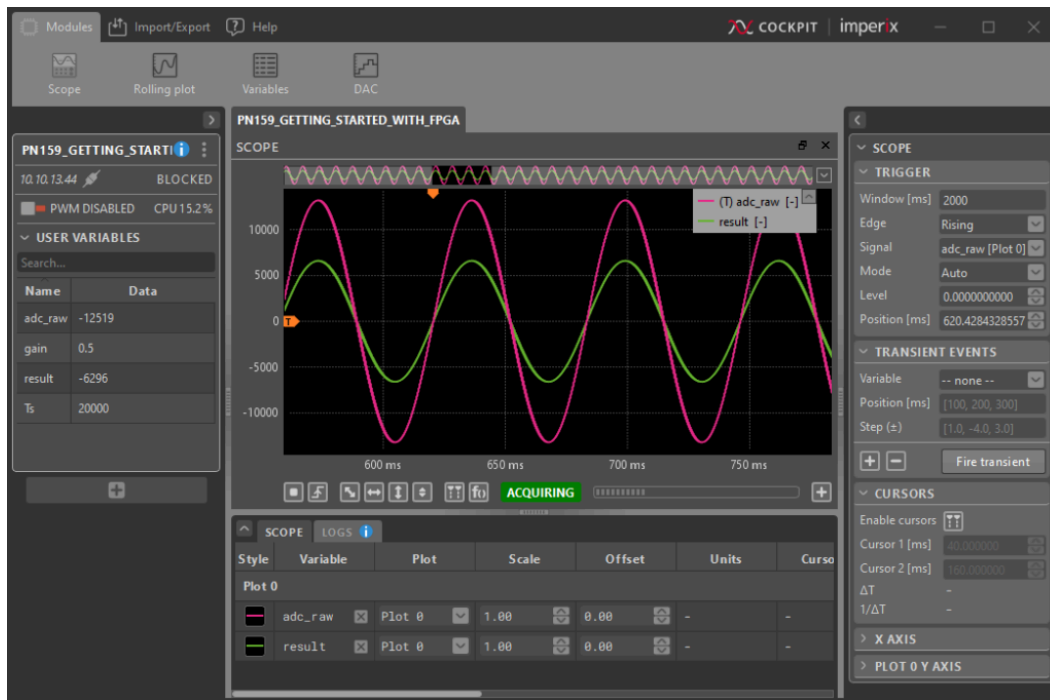


Loading an FPGA bitstream in the controller

Experimental validation

Finally, the CPU code is generated from the Simulink model and loaded into the device, as explained in the [programming and operating imperix controllers](#) page.

To test the design, a sinusoidal signal is fed to the analog input of the B-Box. Then, using Cockpit's scope, the $result = 0.5 * adc_raw$ is observed:

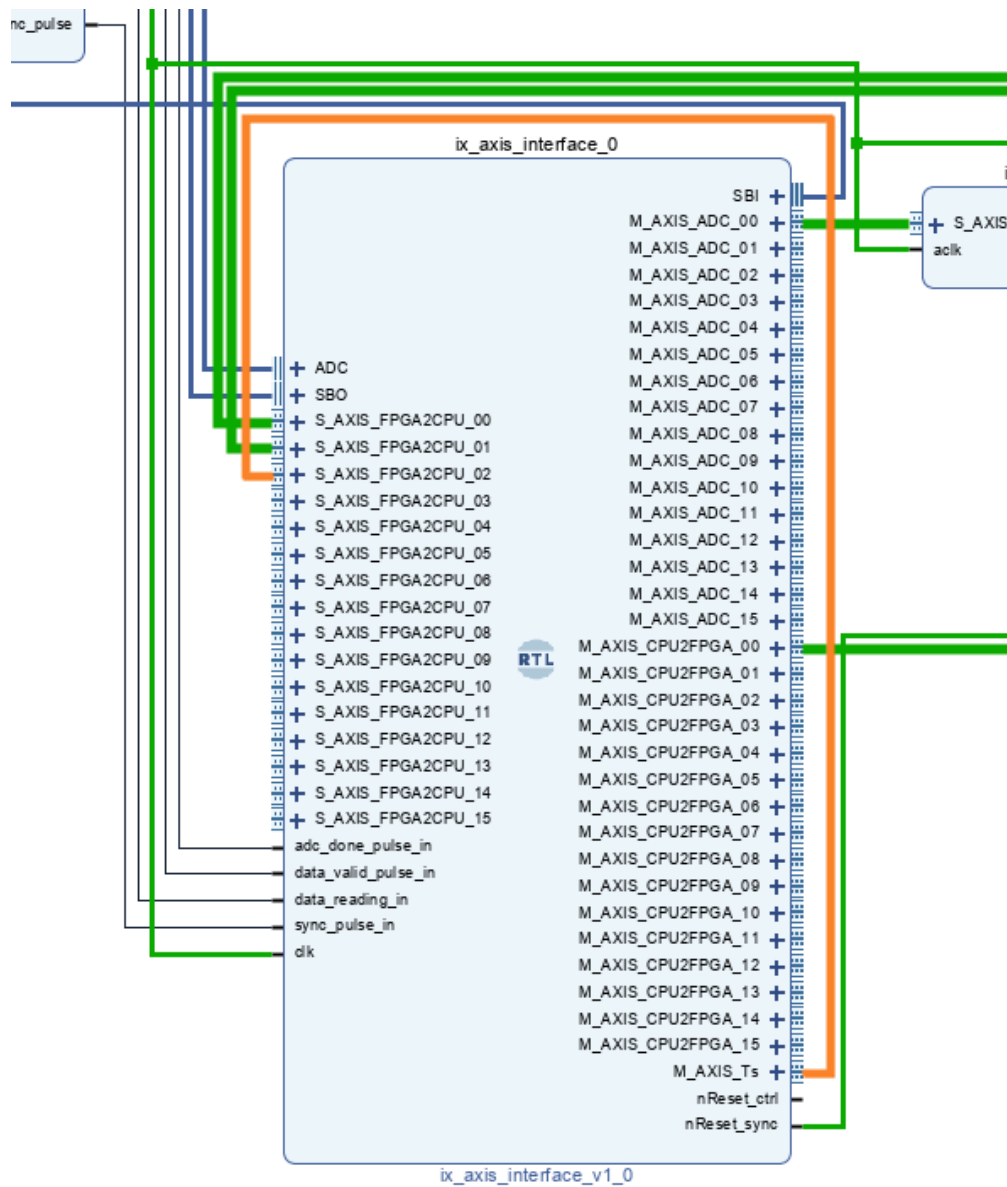


Going further

Testing M_AXIS_Ts

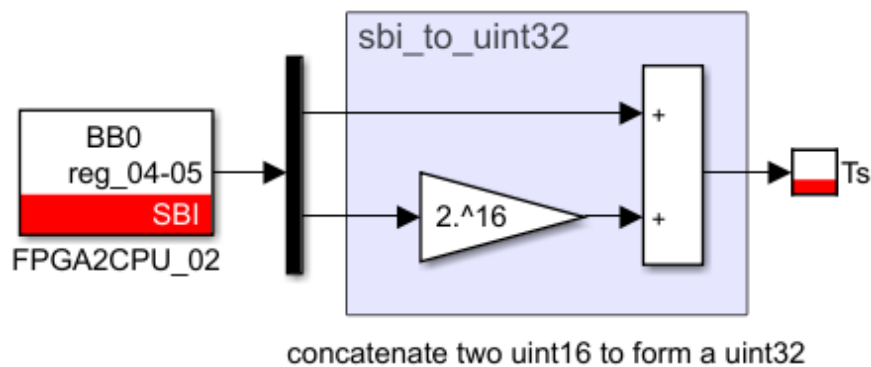
This section goes a bit further in the demonstration of the provided AXI4-Stream M_AXIS_Ts to help understand the **M_AXIS_Ts interface**, and show how to **transfer an uint32** value from the FPGA to the CPU.

The modification of the FPGA bitstream is quite simple: simply connect **M_AXIS_Ts** to **S_AXIS_FPGA2CPU_02** as shown in orange on the image below. This allows reading the sample time Ts value from the CPU.

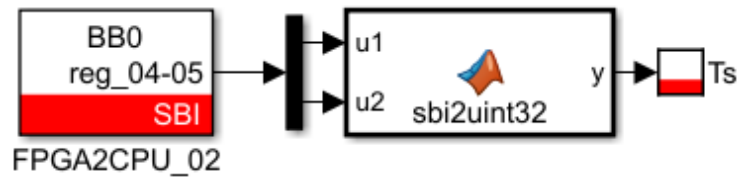


On the CPU, to retrieve the Ts value, the value is read from **S_AXIS_FPGA2CPU_02** (SBI_04 and SBI_05). Because the value is a *32-bit unsigned integer*, the transformation is different from before as shown below.

Option 1



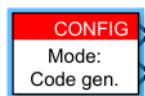
Option 2



```
function y = sbi2uint32(u1,u2)
    y = uint32(0); % fix compiled size of output
    y = typecast([uint16(u1) uint16(u2)], 'uint32');
```

Code language: Matlab (matlab)

Finally, using Cockpit, the result can be observed. If the control task frequency (CLOCK_0) is set to 50 kHz, then **M_AXIS_Ts** will return 20'000 ns.



Block Parameters: Configuration1

Simulink configuration

Performs the basic configuration of the model. Configures the main time base of the model (control task) and sampling signal using CLOCK_0.

- The first output signal sets the frequency of the connected PWM blocks.
- The second output triggers the sampling of the connected ADC blocks.

Model execution purpose

☐ Simulation

☒ Automated Code Generation

Draw sampling and control task information

Parameters initialization file

☐ Execute script file at init (.m) File name param.m

Control task Advanced sampling Simulation

CLOCK_0 frequency (Hz) 20e3

Sampling phase (0 to 1) 0.5

Postscaler 0

Additional clocks are available by using a 'Clock' block.
[Add a 'Clock' block to the model.](#)

OK Cancel Help Apply

PN159_GETTING_STARTI ⓘ ⋮	
10.10.13.44 🖱	BLOCKED
■ PWM DISABLED	CPU 15.6%
▼ USER VARIABLES	
Search...	
Name	Data
adc_raw	7067
gain	0.5
result	3600
Ts	20000

If CLOCK_0 is kept at 50 kHz and the oversampling is activated with an oversampling ratio of 20, then **M_AXIS_Ts** will return 1000 ns, which corresponds to the actual sampling period.

CONFIG
Mode:
Code gen.

Block Parameters: Configuration1

Simulink configuration
Performs the basic configuration of the model. Configures the main time base of the model (control task) and sampling signal using CLOCK_0.

- The first output signal sets the frequency of the connected PWM blocks.
- The second output triggers the sampling of the connected ADC blocks.

Model execution purpose

☐ Simulation

☒ Automated Code Generation

Draw sampling and control task information

Parameters initialization file

☐ Execute script file at init (.m) File name param.m

Control task Advanced sampling Simulation

Oversampling configuration

Oversampling

Evenly distributed sampling events

Total number of sampling events per period

20

Acquisition parameter

ADC acquisition delay

500 ns (B-Box Micro, B-Board PRO or TPI)

OK Cancel Help Apply

PN159_GETTING_START

10.10.13.44 BLOCKED

PWM DISABLED CPU 14.9%

USER VARIABLES

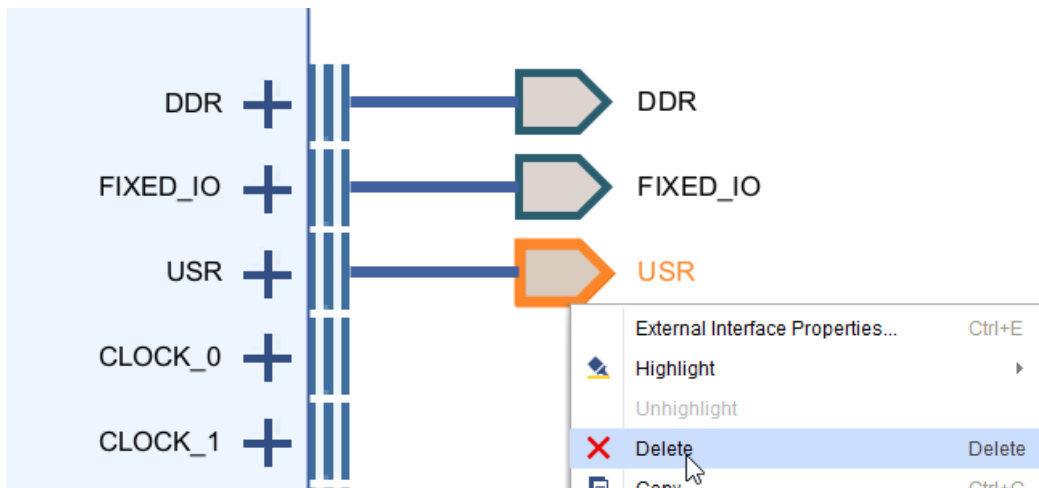
Search...

Name	Data
adc_raw	-13153
gain	0.5
result	-6585.5
Ts	1000

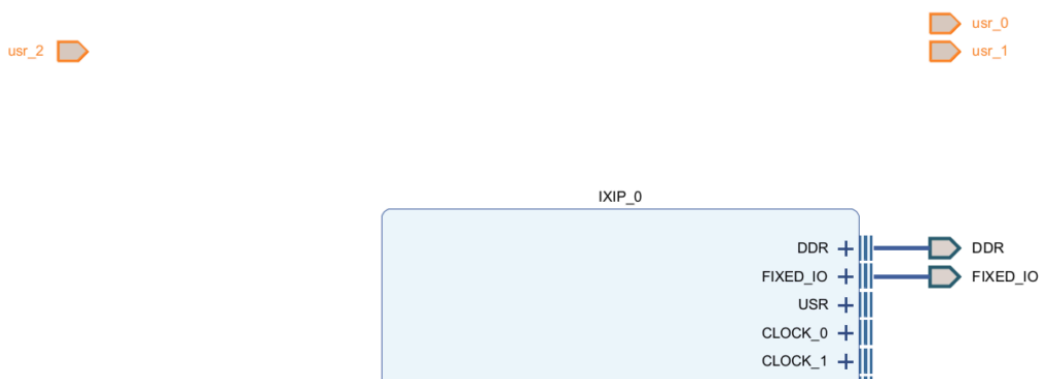
Using the USR pins

The imperix controllers feature 36 user-configurable 3.3V I/Os (the **USR** pins) that are directly accessible from the FPGA. Their physical locations are available in the [B-Board PRO datasheet](#) and [B-Box RCP datasheet](#).

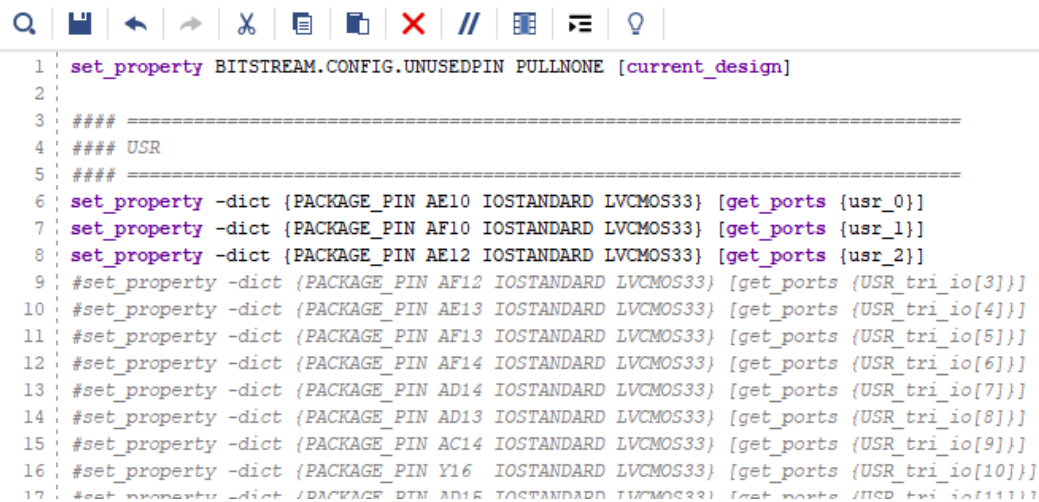
By default, the USR pins are connect to the imperix IP. Currently they are only used to communicate with the with the [motor interface](#). If the motor interface is not used, then the USR port can safely be deleted.



These pins are then available for other use. The screenshot show an example where the USR pins 0, 1, and 2 are used.



The constraints\sandbox_pins.xdc must be edited accordingly. As shown below, we recommend commenting (#) the unused pins to avoid generated unnecessary warning the Vivado. For more information on constraints in Xilinx FPGA please refer to the [using constraints in Vivado Design Suite](#) user guide.



```

1  set_property BITSTREAM.CONFIG.UNUSEDPIN PULLNONE [current_design]
2
3  ### =====
4  ###  USR
5  ###  =====
6  set_property -dict {PACKAGE_PIN AE10 IOSTANDARD LVCMOS33} [get_ports {usr_0}]
7  set_property -dict {PACKAGE_PIN AF10 IOSTANDARD LVCMOS33} [get_ports {usr_1}]
8  set_property -dict {PACKAGE_PIN AE12 IOSTANDARD LVCMOS33} [get_ports {usr_2}]
9  #set_property -dict {PACKAGE_PIN AF12 IOSTANDARD LVCMOS33} [get_ports {USR_tri_io[3]}]
10 #set_property -dict {PACKAGE_PIN AE13 IOSTANDARD LVCMOS33} [get_ports {USR_tri_io[4]}]
11 #set_property -dict {PACKAGE_PIN AF13 IOSTANDARD LVCMOS33} [get_ports {USR_tri_io[5]}]
12 #set_property -dict {PACKAGE_PIN AF14 IOSTANDARD LVCMOS33} [get_ports {USR_tri_io[6]}]
13 #set_property -dict {PACKAGE_PIN AD14 IOSTANDARD LVCMOS33} [get_ports {USR_tri_io[7]}]
14 #set_property -dict {PACKAGE_PIN AD13 IOSTANDARD LVCMOS33} [get_ports {USR_tri_io[8]}]
15 #set_property -dict {PACKAGE_PIN AC14 IOSTANDARD LVCMOS33} [get_ports {USR_tri_io[9]}]
16 #set_property -dict {PACKAGE_PIN Y16 IOSTANDARD LVCMOS33} [get_ports {USR_tri_io[10]}]
17 #set_property -dict {PACKAGE_PIN AD15 IOSTANDARD LVCMOS33} [get_ports {USR_tri_io[11]}]

```

When top level ports are modified, the block design wrapper must be updated accordingly to reflect the changes. To make sure `top_wrapper.vhd` is up-to-date, the recommended procedure is to remove the current one and generate a new one:

- Right click on `top_wrapper` -> Remove File from Project...
- Check “Also delete the project file from disk” -> OK
- Right click on `top` -> Create HDL Wrapper

The [FPGA-based SPI communication IP for ADC](#) page shows an example where USR pins are used to drive an external ADC using the SPI protocol.

Additional tutorials

The page [custom FPGA PWM modulator](#) explains how to **drive PWM outputs** or to use the **CLOCK interfaces** through a simple example. The PWM modulator sources are provided as VHDL, as a Xilinx System Generator model, and as a MATLAB HDL Coder model.

The page [high-level synthesis for FPGA](#) developments shows how to integrate **HLS-generated IPs** in an FPGA control implementation using a **PI-based current control** of a buck power converter as an example.

Back to [FPGA development homepage](#)