High-Level Synthesis for FPGA developments

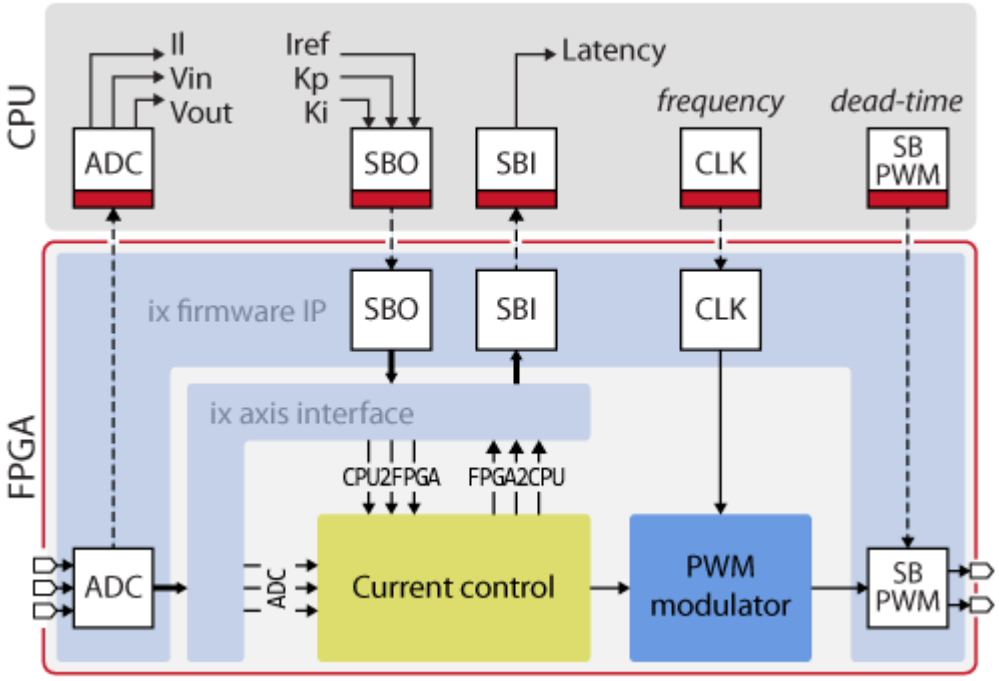
TN142 | Posted on June 15, 2021 | Updated on May 7, 2025

Benoît STEINMANN
Software Team Leader
imperix

Table of Contents

High-level synthesis (HLS) tools greatly facilitate the implementation of complex power electronics controller algorithms in FPGA. Indeed HLS tools allow the user to work at a higher level of abstraction. For instance, the user can use Xilinx Vitis HLS to develop FPGA modules using **C/C++** or the Model Composer plug-in for **Simulink** to use graphical programming instead.

This page shows how IPs generated using high-level synthesis tools can be integrated into the FPGA of an imperix power controller. To this end, the example of a **PI-based current controller for a buck converter** is used to illustrate all the required steps.



Power converter control FPGA

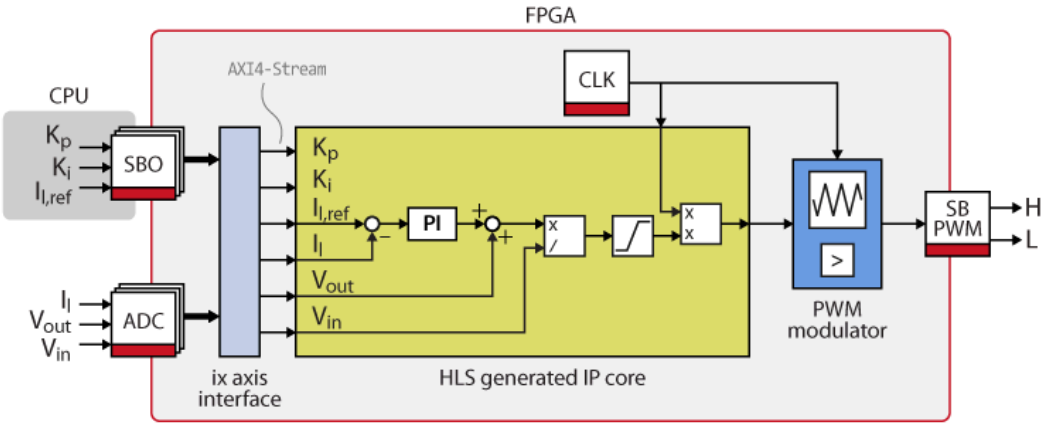
To find all FPGA-related notes, you can visit FPGA development homepage.

Integrating HLS designs in the FPGA

Description of the design

The image below shows the example that will be implemented on this page. It is a PI-based current controller for a buck converter, based on the algorithm presented on the PI controller implementation for current control technical note. This example uses the following resources

- the **FPGA control starter template** from the getting started with FPGA guide
- the **PWM modulator IP** from the FPGA PWM modulator example
- the **high-level synthesis PI-based current control IP** from either
  - the C++ implementation presented in the Xilinx Vitis HLS guide
  - or the Simulink implemention presented in the Model Composer guide



The **axis interface** provides the inputs of the current control algorithm in form of **AXI4-Stream** ports. The following ports are used:

- `CPU2FPGA_00` for the current reference `Il_ref` (32-bit single-precision)
- `CPU2FPGA_01` for the parameter `Kp` (32-bit single-precision)
- `CPU2FPGA_02` for the parameter `Ki` (32-bit single-precision)
- `ADC_00` for the measured current `Il` (16-bit signed integer)
- `ADC_01` for the measured output voltage of the converter `Vout` (16-bit signed integer)
- `ADC_02` for the measured input voltage of the converter `Vint` (16-bit signed integer)
- `Ts` for the sampling period in nanoseconds (32-bit unsigned integer)

Aside from AXI4-Stream data, the current control IP also uses the ports:

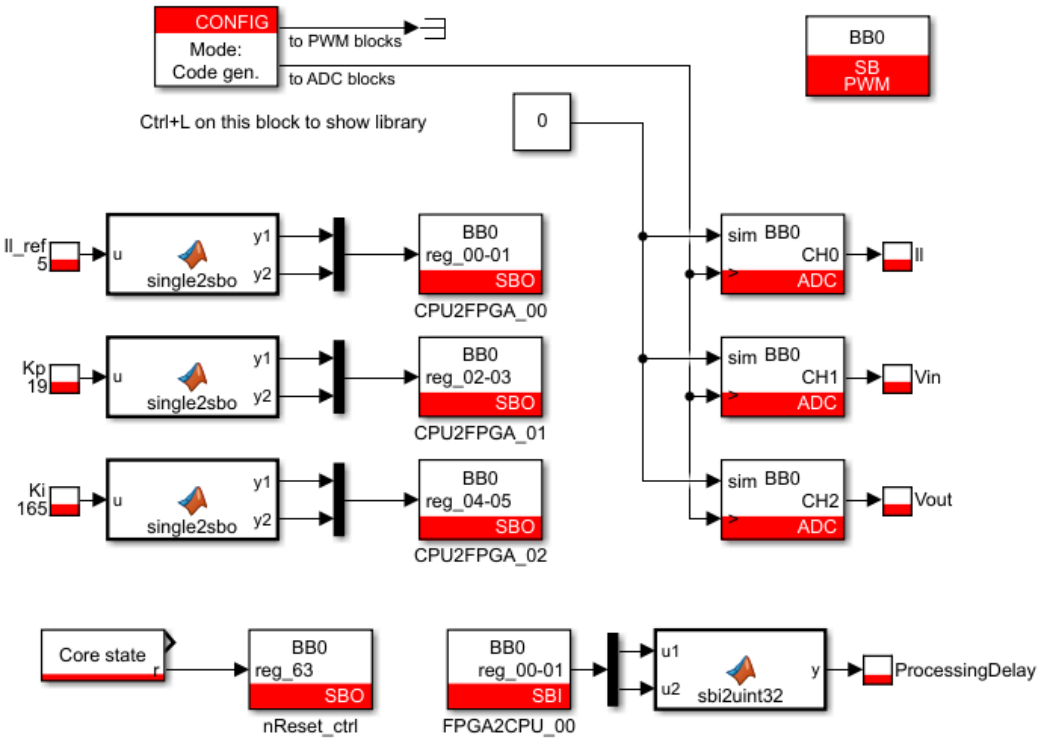
- `CLOCK_period` for the PWM period in ticks (16-bit unsigned)
- `nReset_ctrl` to reset the PI when the controller is not in OPERATING state

Using these signals, the HLS IP computes a 16-bit unsigned `duty_cycle_ticks` that is forwarded to the PWM IP. And finally, the PWM IP uses the `sb_pwm` driver to output the PWM signals to optical fibers of the B-Box RCP controller. The PWM IP and the SB-PWM driver are further documented on the FPGA PWM modulator page.

The ADC values provided by the starter template are the raw result from the ADC chips. They are multiplied by a *gain* inside the HLS IP to obtain physical values. An example of *gain* computation is available on the ADC block help page.

CPU-side implementation using Simulink

The CPU-side model is quite simple, as the control algorithm runs entirely in the FPGA. The CPU code provides the current reference and Kp/Ki parameters, operates the PI reset signal, and configures the PWM outputs.



The *single2sbo* MATLAB Function blocks are used to map the current reference **Il_ref** and the **Kp**, **Ki** parameter to the CPU2FPGA ports.

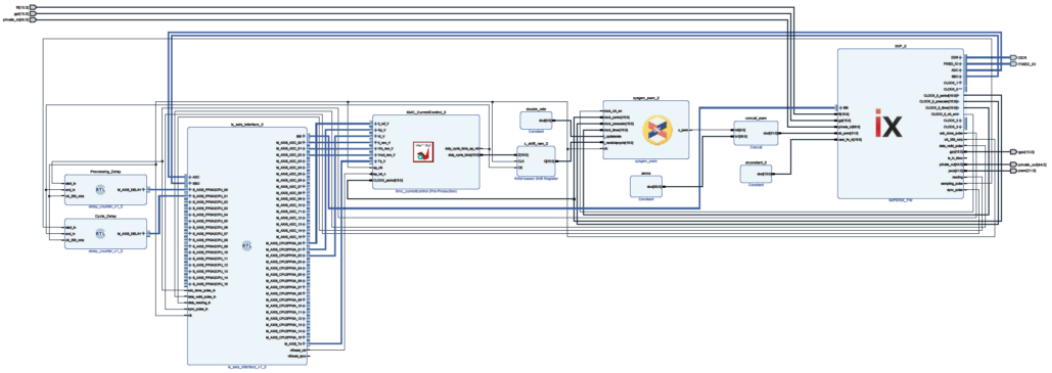
This **nReset_ctrl** signal is used to keep the PI integrator at reset when the controller is not in OPERATING state. As documented in Getting started with FPGA, this reset signal is controlled using *SBO_63*. To obtain the desired behavior, we'll simply connect the reset output of a Core state block to *SBO_63*.

And finally, the SB-PWM block is used to activate the output PWM **channel 0 (CH0)** (lane #0 and lane #1). The output is configured as **Dual (PWM_H + PWM_L)** with a **deadtime** of 1 µs. This configuration expects a PWM signal coming to **sb_pwm[0]** input of the *imperix firmware* IP and will automatically generate the complementary signals with the configured deadtime.

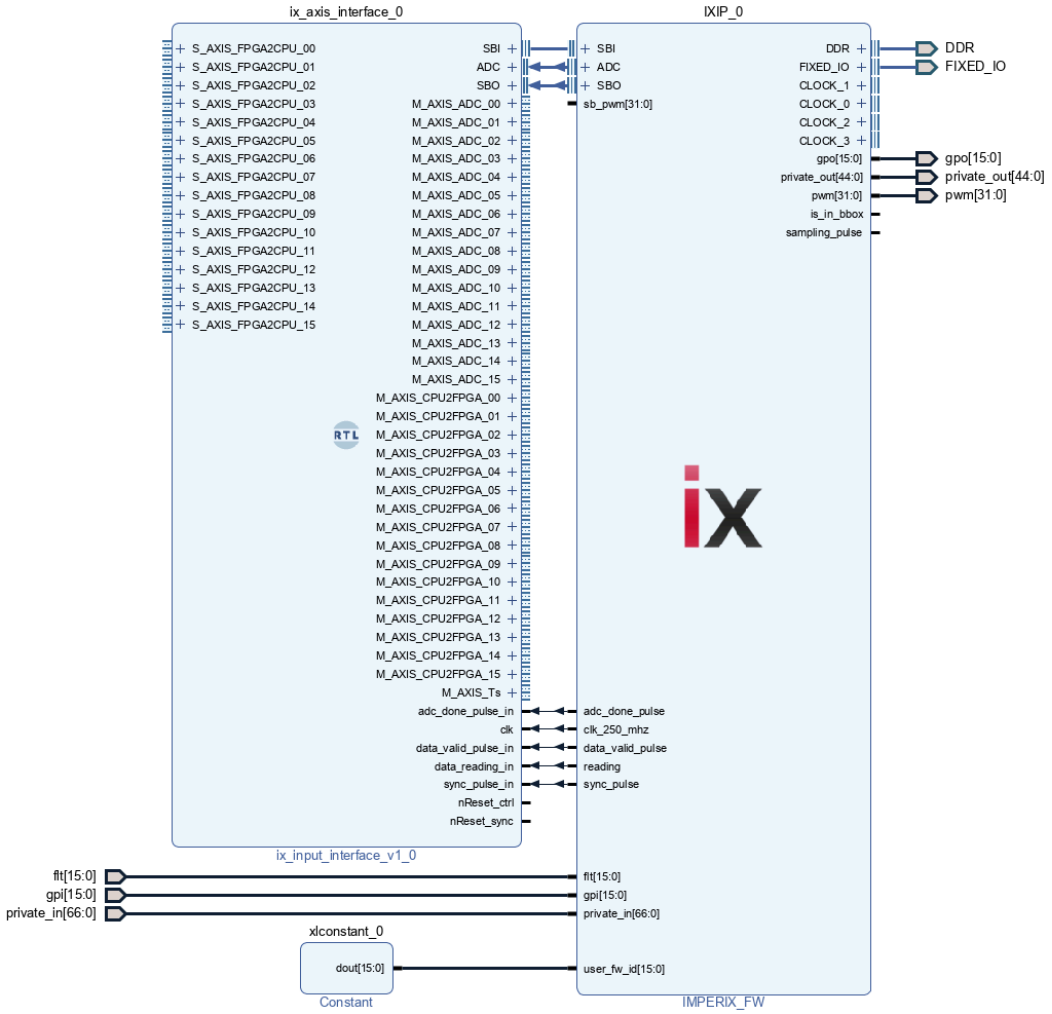
The ADC blocks are only used to retrieve the analog input signals at the CPU level for real-time monitoring. They do not affect the closed-loop control behavior.

FPGA-side implementation using Vivado

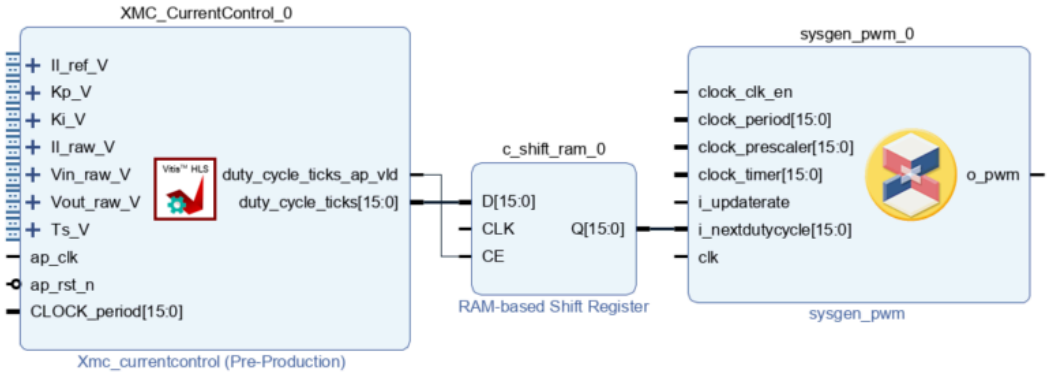
The **TN142_vivado_design.pdf** file below shows the full Vivado FPGA design. Here are the step-by-step instructions to reproduce it.

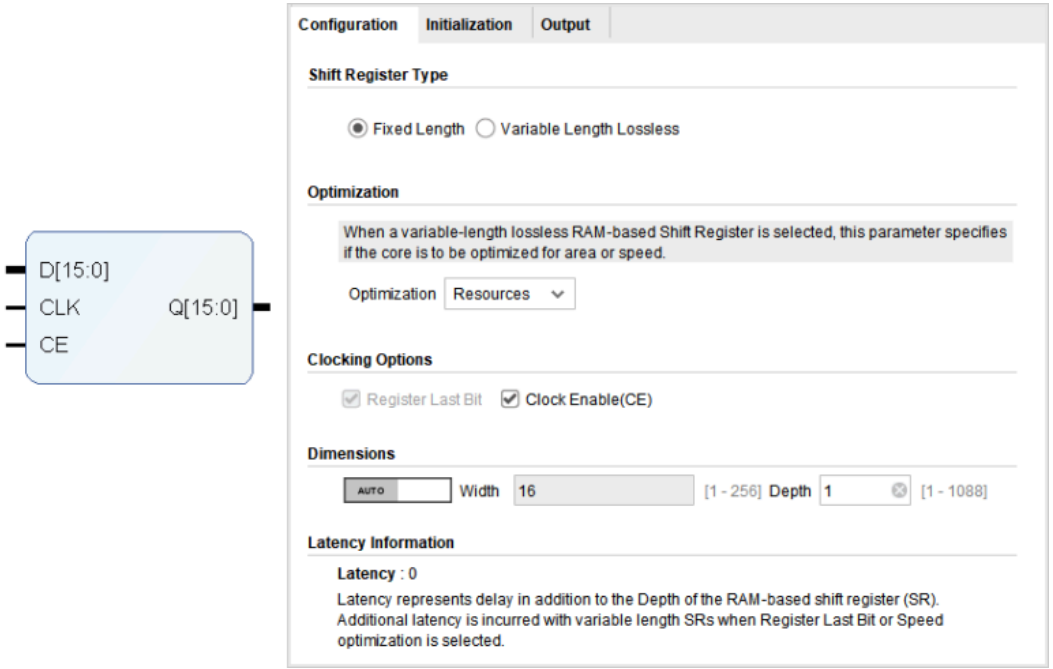


1. Create an FPGA control implementation starter template by following the [Getting started with FPGA control implementation].

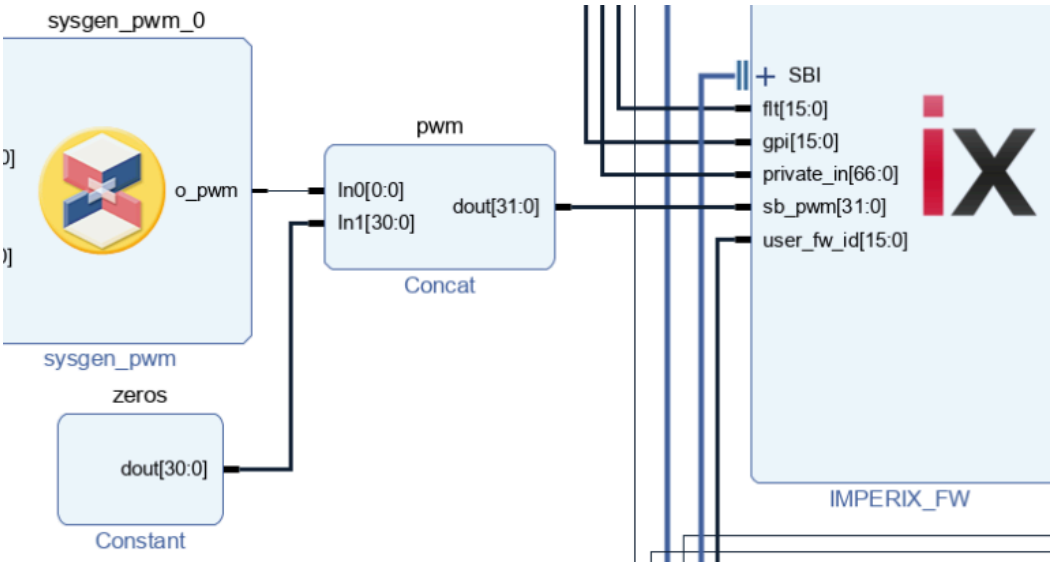


2. Add the *PWM IP* (from the [custom PWM in FPGA] page) and *current control IP* (from the [Xilinx Vitis HLS] guide or the [Model Composer] guide) into your Vivado project. In the screenshots of this example, we'll use the IPs generated using System Generator and Vitis HLS, respectively.
   To read the `duty_cycle_ticks` only when `duty_cycle_ticks_ap_vld` is '1', the *RAM-based Shift Register* IP is used. With the configuration shown in the screenshot below, this block adds one register stage that acts as a buffer. It keeps the last computed duty cycle until a new value has been computed. When a new value is available, it replaces the old one.

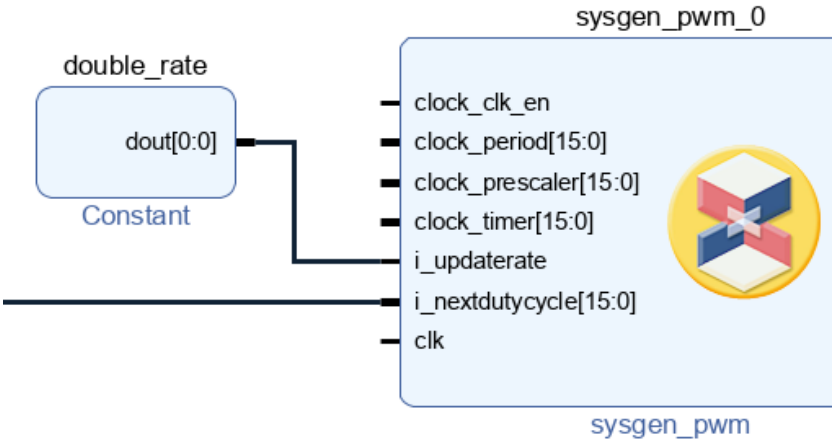




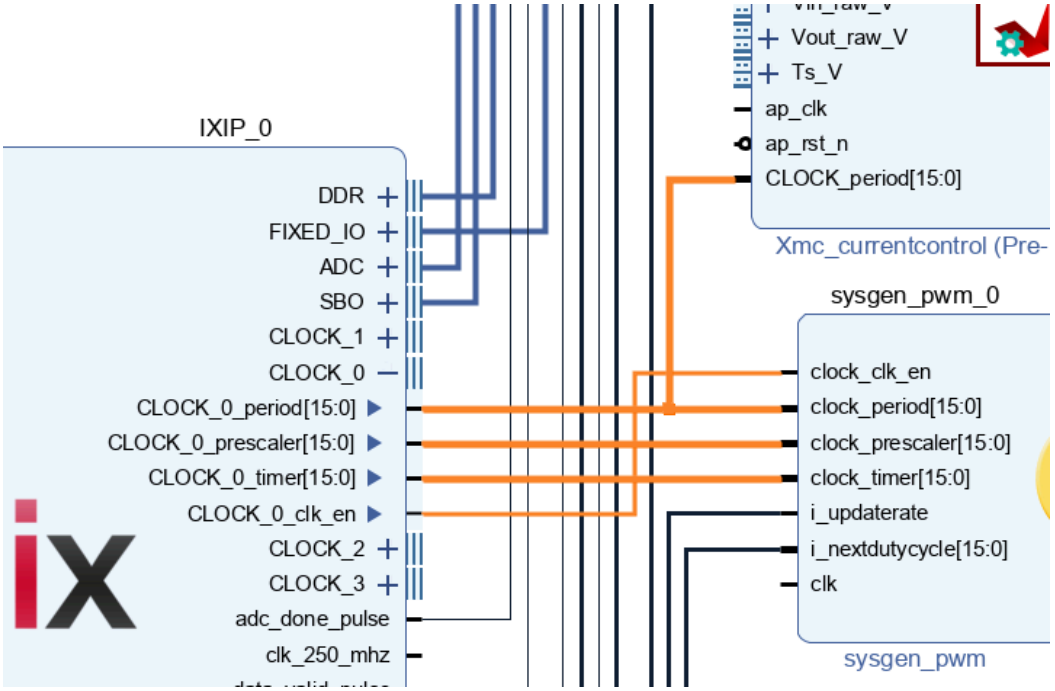
3. Add a **Constant** IP to set all the 31 unused sb_pwm outputs to '0'. Set its *Const Width* to 31 and its *Const Val* to 0.
4. Add a **Concat** IP. It will serve to concat the pwm output of the PWM IP with the zeros of the Constant IP.



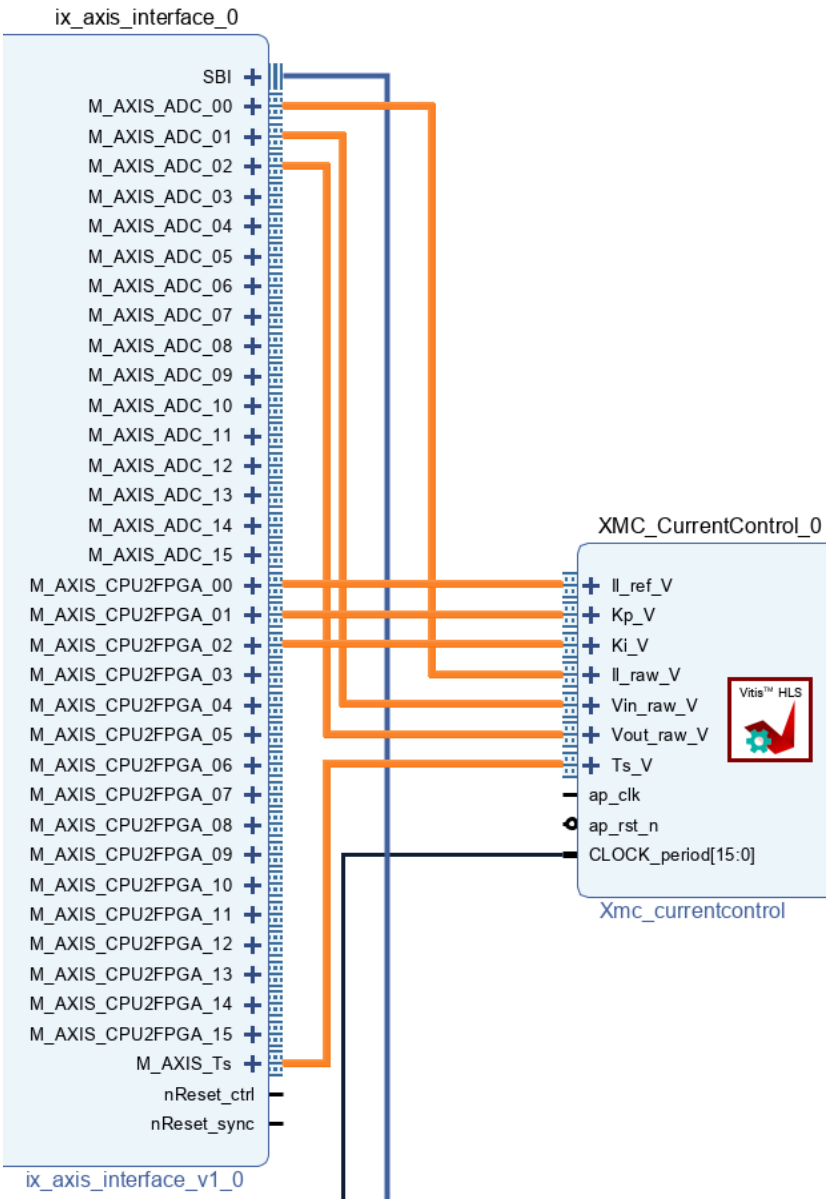
5. Add a **Constant** IP to set to set the update rate. '0' = single rate, '1' = double rate.



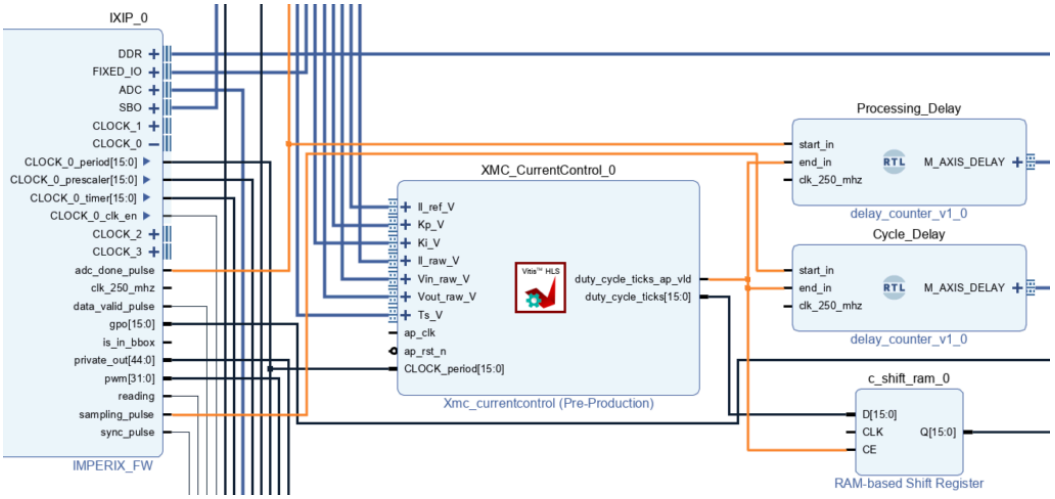
6. Connect the clock signals as below:



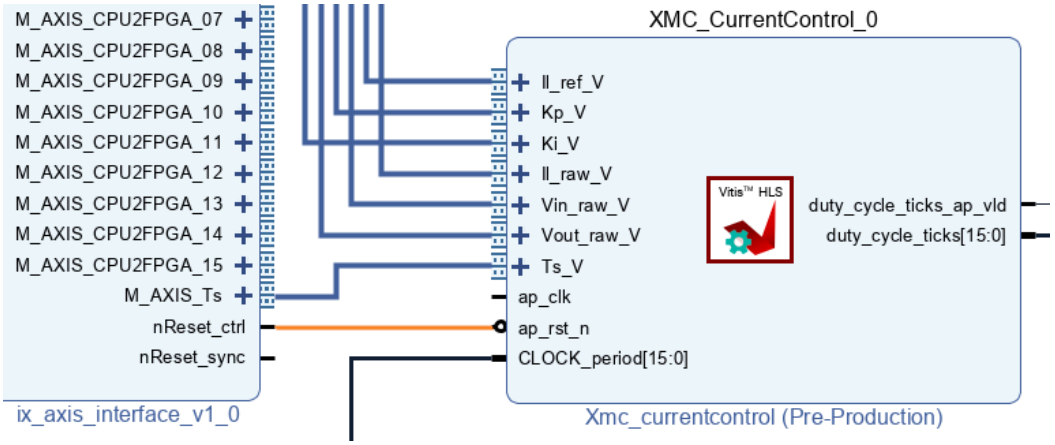
7. Connect the AXI4-Streams
   - **`M_AXIS_CPU2FPGA_00`** to **`Il_ref_V`**
   - **`M_AXIS_CPU2FPGA_01`** to **`Kp_V`**
   - **`M_AXIS_CPU2FPGA_02`** to **`Ki_V`**
   - **`M_AXIS_ADC_00`** to **`Il_raw_v`**
   - **`M_AXIS_ADC_01`** to voltage **`Vout_raw_V`**
   - **`M_AXIS_ADC_02`** to **`Vint_raw_V`**
   - **`M_AXIS_Ts`** to **`Ts_V`**



8. The provided *Delay Counter* VHDL module (delay_counter.vhd) measures the elapsed time between two signals and outputs a time in nanoseconds, encoded as a uint32.
   In this design, the *delay counter* modules are used purely for debugging purposes. As shown in the image below, one is used to measure the *FPGA processing delay*, which is the delay between the `adc_done_pulse` and the `duty_cycle_ticks_ap_vld`. Another module is used to measure the *FPGA cycle delay* by measuring the delay between the `sampling_pulse` and the `duty_cycle_ticks_ap_vld`. More information on what these delays represent are available on the discrete control delay product node.



9. Connect the `nReset_ctrl` signal to `ap_rst_n`.



10. And finally connect the clk signals to `clk_250_mhz`.
11. Click **Generate bitstream**. It will launch the synthesis, implementation, and bitstream generation
12. Once the bitstream generation is completed, click on **File → Export → Export Bitstream File...** to save the bitstream somewhere on your computer.

Back to FPGA development homepage