

# FPGA-based control of a grid-tied inverter

TN147 | Posted on June 2, 2021 | Updated on December 31, 2025



Shu WANG

Development Engineer

imperix • [in](#)

---

## Table of Contents

- [Grid-tied inverter control](#)
- [Overview of the FPGA-based inverter control task](#)
- [Performance analysis of the control task](#)
  - [Latency and control delay](#)
  - [Resource utilization](#)
- [Experimental validation](#)
  - [Testbench description](#)
  - [Experimental results](#)
- [Creation of the Vivado block design](#)
  - [Raw ADC data conversion](#)
  - [Sample time \( \$T\_s\$ \) conversion](#)
  - [Grid synchronization](#)
  - [DQ current control](#)
  - [Duty cycles computation](#)
  - [PWM generation](#)
  - [CPU-side implementation](#)
  - [Debugging and monitoring](#)

This note presents an FPGA control implementation of a **grid-tied current-controlled inverter**. It combines several control modules presented in different Technical Notes to form a complete converter control, executed entirely in the FPGA of a [B-Box RCP](#) controller.

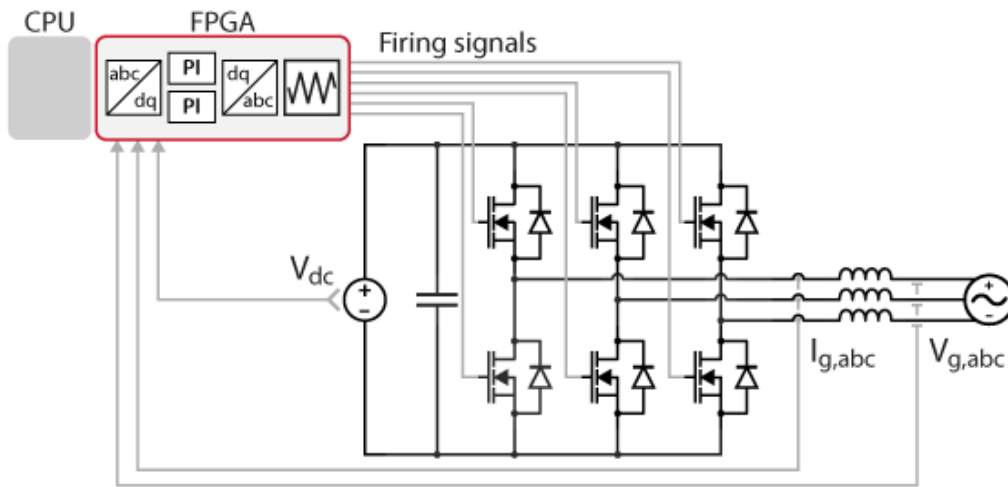
Thanks to the [FPGA programmability](#) of the B-Box controller, complex control algorithms can be effectively executed at high rates and with minimal latency. In particular, this example shows that a grid-oriented current control algorithm can be executed as fast as 650 kHz, whereas the equivalent CPU-based execution is “limited” to 210 kHz (which is already an industry-leading figure amongst prototyping controllers).

Besides, the fast switching frequency used in this example takes full advantage of the imperix [SiC phase leg module](#).

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

# Grid-tied inverter control

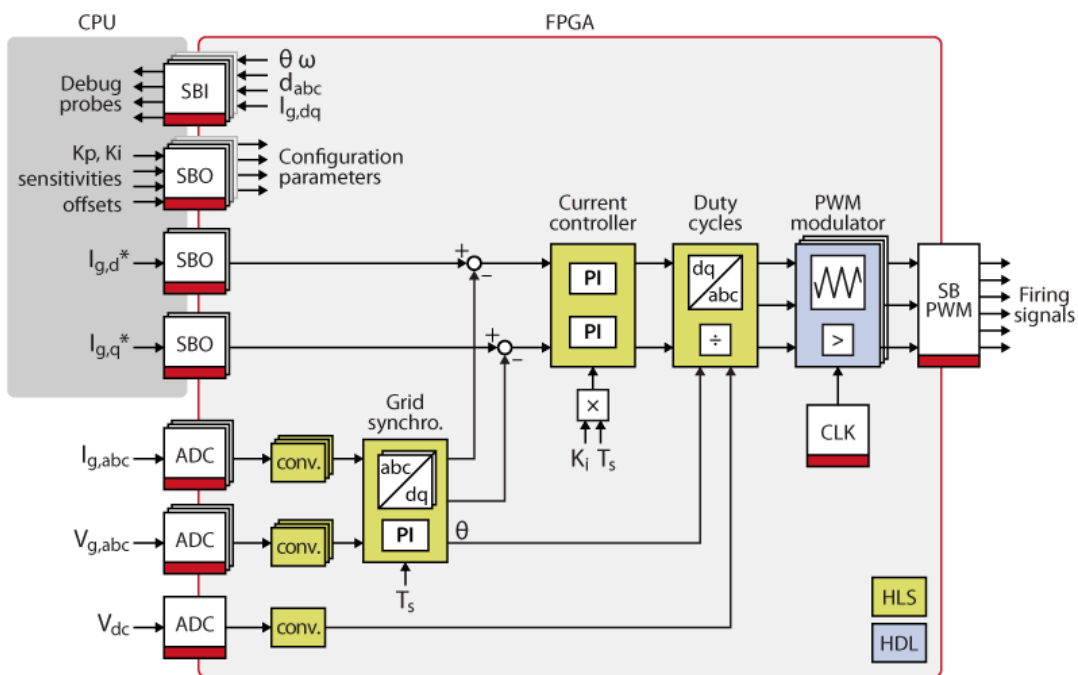
The controlled system is a standard current-controlled voltage-source inverter, connected to a 3-phase grid. This converter is built using imperix power modules in the [experimental validation](#) section.



Grid-tied voltage-source inverter

The control algorithm is entirely executed in the controller FPGA and implements a [three-phase PLL](#) for grid synchronization coupled to a standard [dq current control](#) in the grid-oriented reference frame. Based on user-defined current references, the controller computes the voltages that the inverter should produce in order to match the required current. These voltages are then modulated in PWM signals and fed to the gates of the inverter.

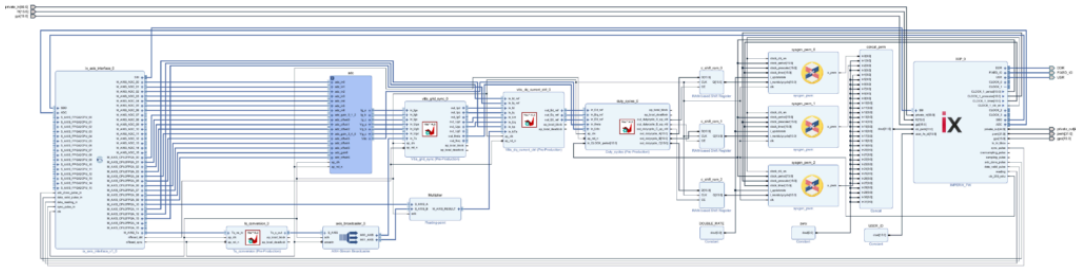
The overall inverter control algorithm is shown below, and each of the elementary control blocks is further described in the following sections.



Block diagram of the implemented FPGA-based control algorithm (simplified view)

# Overview of the FPGA-based inverter control task

Below is shown the implemented Vivado block design used to generate the FPGA bitstream. The [creation of the Vivado block design](#) section describes in more detail how to reproduce this block design. All the sources can be downloaded by clicking on the button below.



[Download TN147\\_block\\_design.pdf](#)

The design uses the following IPs

- **ADC conversion** module (Vitis HLS)
- **Ts conversion** module (Vitis HLS)
- **Grid synchronization** module (Vitis HLS or Model Composer, documented in [TN143](#))
- **Dq current controller** (Vitis HLS or Model Composer, documented in [TN144](#))
- **duty cycle computation** module (Vitis HLS)
- **Carrier-based PWM** module (System Generator or HDL Coder, documented in [TN141](#))

[Download TN147 FPGA Grid Tied Inverter.zip](#)

These IPs have been implemented using **High-Level Synthesis** tools such as [Vitis HLS](#) (free of cost, C++) and [Model Composer](#) (~500\$, requires MATLAB Simulink). These tools offer a simple yet powerful way of developing control algorithms in FPGA.

## Performance analysis of the control task

Without any special optimization, the latency of the presented inverter control algorithm is roughly  $1.5\mu\text{s}$  (see details below), which means that it can run above 650 kHz. Comparatively, the similar CPU-based algorithm presented in [TN106](#) can run at up to 210 kHz.

## Latency and control delay

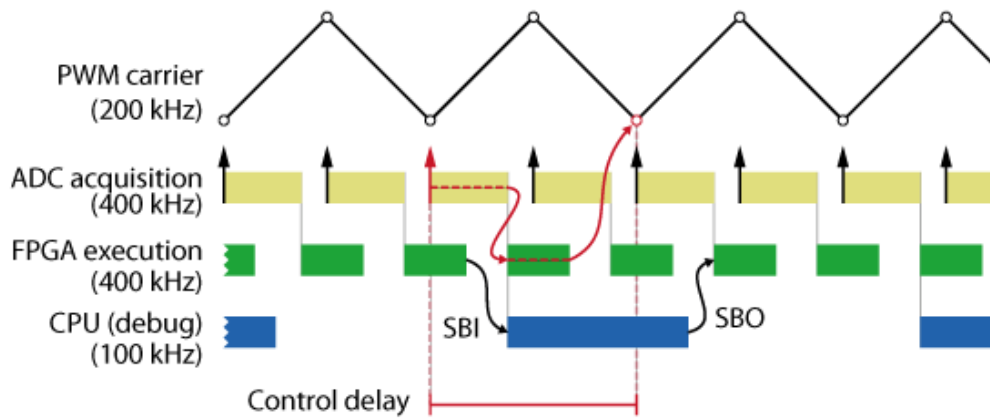
The total latency can be estimated by simply adding up the latency of each module, which are:

Module	Latency Vitis HLS	Latency Model Composer
ADC conversion	24 cycles	24 cycles
<b>Total latency</b>	379 cycles ( <b><math>1.52\mu\text{s}</math></b> )	363 cycles ( <b><math>1.45\mu\text{s}</math></b> )

Module	Latency Vitis HLS	Latency Model Composer
Grid synchronization	145 cycles	146 cycles
DQ current control	72 cycles	56 cycles
Duty cycles computation	138 cycles	137 cycles
<b>Total latency</b>	<b>379 cycles (1.52μs)</b>	<b>363 cycles (1.45μs)</b>

That estimated latency is comparable to the measured latency of 385 cycles (1.54μs) with the Vitis HLS implementation. The Model Composer approach achieves a slightly lower latency thanks to automatic optimization but at the expense of slightly higher resource utilization.

Considering a conversion time of the ADC chip of 2μs, the total control delay is 3.54μs, which is larger than one sampling period (chosen 2.5μs – 400kHz). Therefore, the execution of the control task is pipelined with the ADC acquisition, as shown in the figure below. The control delay to consider when tuning the current controller is therefore  $T_{d,ctrl} = 2T_s$ .



Further details on control delay identification can be found in the [PN142](#).

## Controller tuning

The gains of the PI controllers are tuned using the Magnitude Optimum, as introduced in [Vector current control](#). The total delay of the control loop is identified as:

- Control delay:  $T_{d,ctrl} = 2T_s = 5 \mu s$
- Modulator delay (double-rate update):  $T_{d,PWM} = T_{sw}/4 = T_s/2 = 1.25 \mu s$
- Sensing delay: (16 kHz filter)  $T_{d,sens} \approx 10 \mu s$
- Total loop delay:  $T_{d,tot} = T_{d,ctrl} + T_{d,PWM} + T_{d,sens} \approx 16.25 \mu s$

## Resource utilization

The resource utilization is estimated by Vivado after the implementation and is shown below, for the Vitis HLS approach.

Resource	Utilization (inverter control)	Utilization (imperix firmware 3.6)	Total utilization
LUT	18'733	24'220	42'953 (54.65%)
LUTRAM	115	798	913 (3.43%)
FF	30'713	50'732	81'445 (51.81%)
BRAM	2	37	39 (14.72%)
DSP	104	0	104 (26%)

FPGA resource utilization of the inverter control algorithm and the imperix firmware

Resource utilization and latency may vary depending on the software's versions, selected synthesis/placement optimizations, etc.

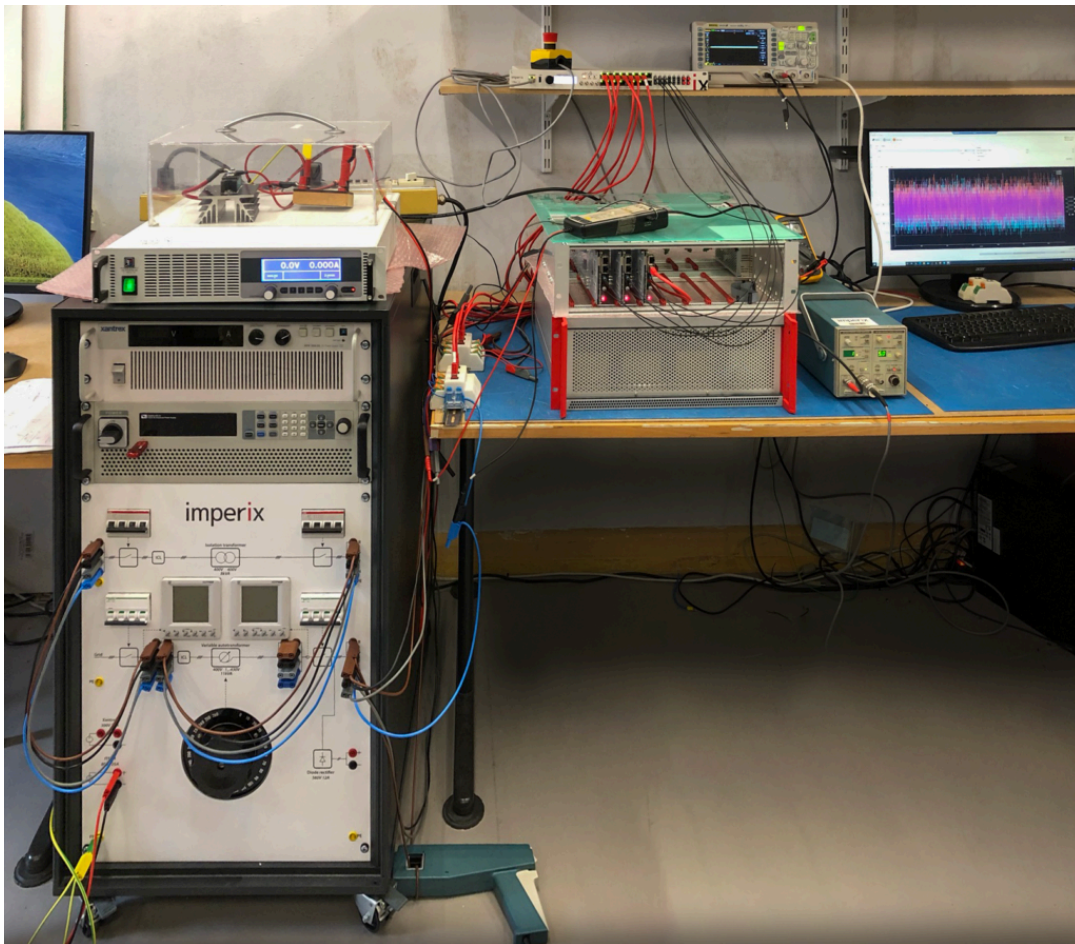
## Experimental validation

General **safety-related recommendations** for operating power converters in a laboratory environment are given in [TN181](#).

## Testbench description

A physical testbench is built to validate the developed control strategy experimentally, using the following equipment:

- Controller: [B-Box RCP](#) with [ACG SDK for Simulink](#)
- Inverter: 3x [PEB8024 phase leg module](#) with fast-switching SiC MOSFETs
- Grid inductors: from [passive filters box](#)



The main operating parameters are summarized in the tables below.

Parameter	Value
DC bus voltage	750 V
Grid voltage	380 V
Grid inductor (from <a href="#">filter box</a> )	2.36 mH

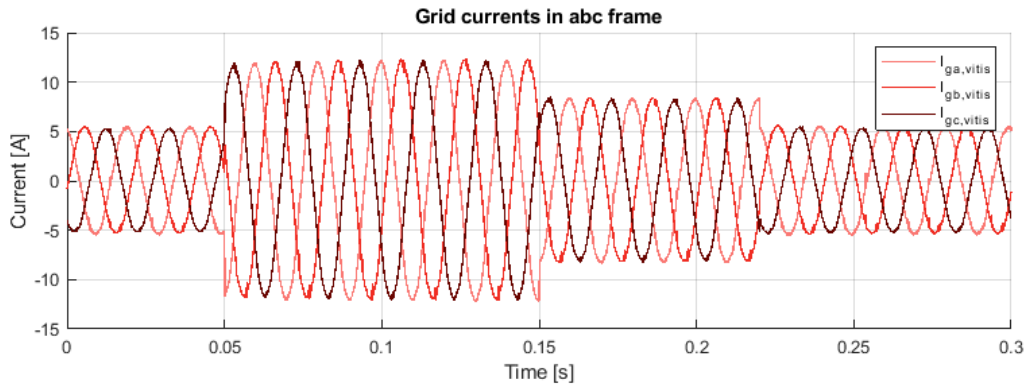
Parameter	Value
Sampling frequency	400 kHz
Control frequency (FPGA)	400 kHz
Switching frequency	200 kHz

Due to the high switching frequency of this application (200kHz), the PWM modulators are configured with a small deadtime (200ns). Therefore, only PEB8024 power modules, which support smaller deadtimes can be used for this testbench. Using other types of modules may result in irreversible damage to the semiconductors.

## Experimental results

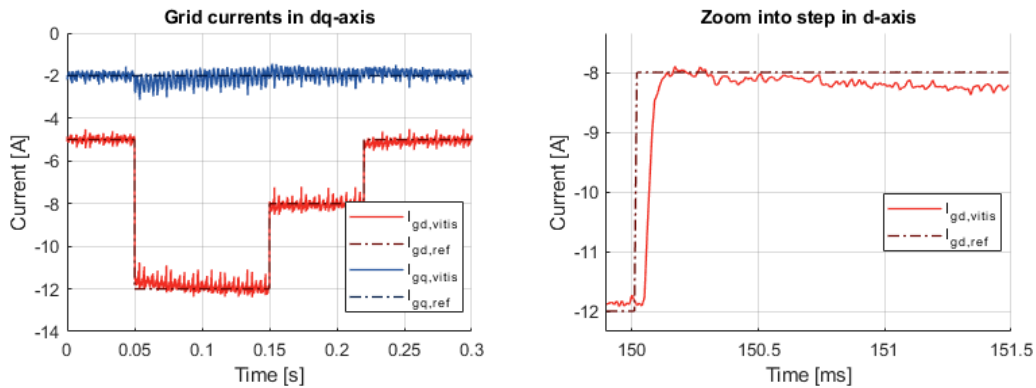
Various reference steps are performed on the d- and q-axis current references to validate the reference tracking and perturbation rejection abilities of the developed algorithm.

The graph below shows the measured grid currents when the d-axis current reference takes the values 5, 12, and 8 A.



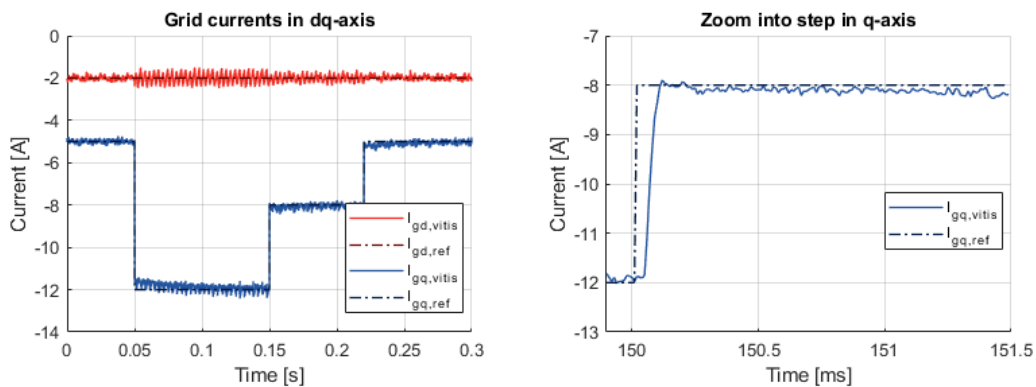
Measured grid currents when changing the d-axis current reference

When projected into the dq rotating reference frame, the measured grid currents give the following results. It can be seen that the current reference is successfully and rapidly tracked by the d-axis controllers and that the perturbation is well rejected on the q axis.



Measured grid currents in the dq reference frame when changing the d-axis current reference

If the same reference steps are performed on the q-axis, the same observations can be made.



Measured grid currents in the dq reference frame when changing the q-axis current reference

# Creation of the Vivado block design

This section provides a step-by-step explanation of how to re-create the Vivado project to generate the FPGA bitstream of the Grid-tied inverter control.

Information on how to create an FPGA control template is available on the [Getting started with FPGA control impl.](#)

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

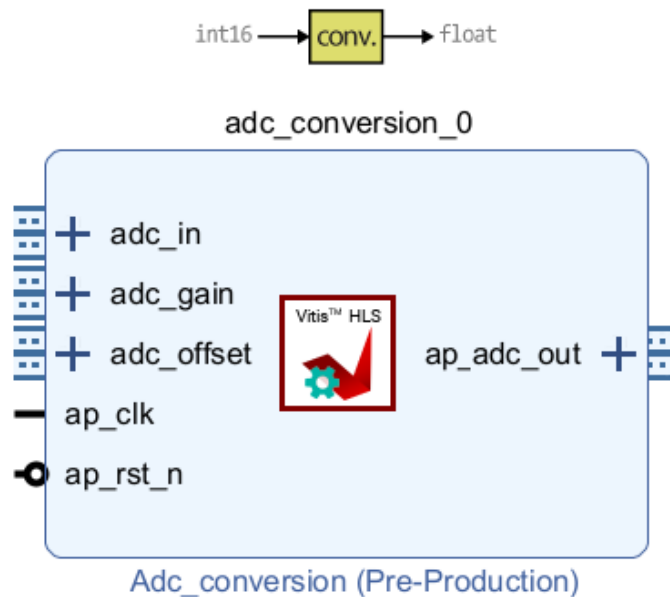
## Raw ADC data conversion

The ADC conversion results are available from the AXI4-Stream interfaces **M\_AXIS\_ADC** of the "ix axis interface" module. They return the raw **16-bit signed integer** result from the ADC chips.

The ADC conversion IP shown below serves to convert that raw data into the **actual measured quantities** in the float datatype. Knowing the sensitivity  $S$  of the sensor and the B-Box RCP front-end gain  $G$ , the formula below can be used:

$$\alpha[\text{bit/A}] = S \cdot G \cdot 32768/10$$

A numerical example of gain computation is available in the last section of the [B-Box analog frontend configuration](#) page.



Vitis HLS – ADC conversion C++ source

```
#include "ADC_conversion.h"

#include <hls_stream.h>
#include <stdint.h>

void adc_conversion(
    hls::stream<int16_t>& adc_in,
    hls::stream<float>& adc_gain,
    hls::stream<float>& adc_offset,
```

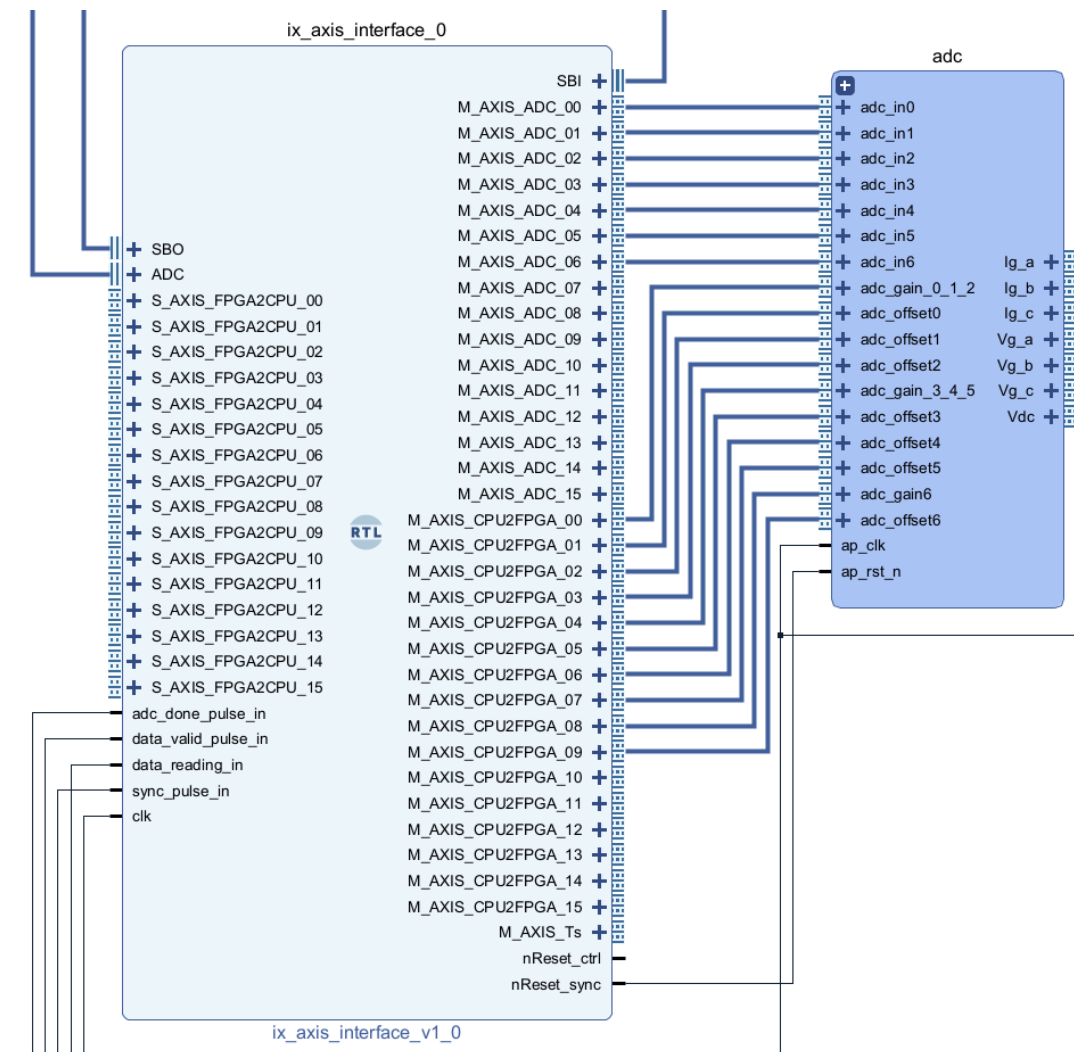
```

hls::stream<float>& adc_out)
{
// see https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/pragma-HLS-interface
// "both" means registers are placed on TDATA, TVALID, and TREADY
#pragma HLS INTERFACE axis port=adc_in register_mode=both register
#pragma HLS INTERFACE axis port=adc_gain register_mode=both register
#pragma HLS INTERFACE axis port=adc_offset register_mode=both register
#pragma HLS INTERFACE axis port=adc_out register_mode=both register
// turns off block-level I/O protocols
#pragma HLS interface ap_ctrl_none port=return

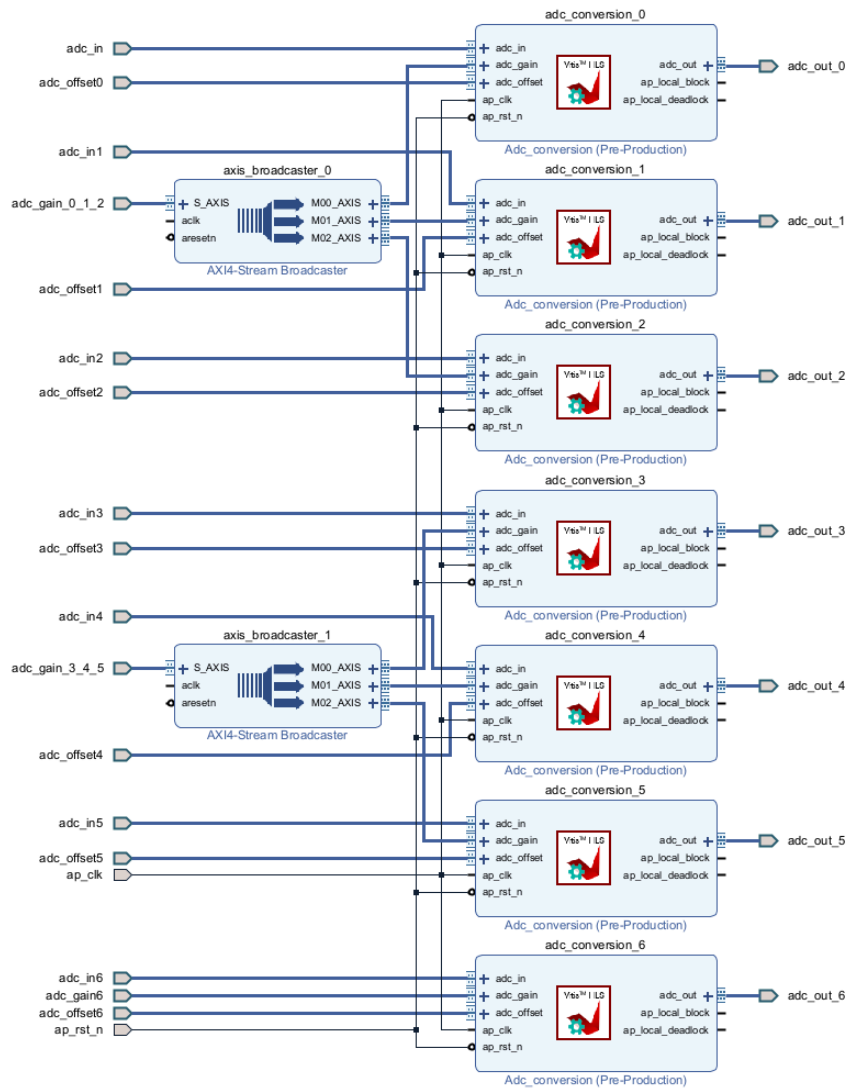
int16_t adc = adc_in.read();
float gain = adc_gain.read();
float offset = adc_offset.read();
float result = (float)adc * gain - offset;
adc_out.write(result);
}Code language: C++ (cpp)

```

This ADC conversion module is connected as shown in the two screenshots below. The **AXI4-Stream Broadcaster** IP (included with Vivado) serves to propagate an AXI4-Stream to multiple ports.



The "adc" hierarchy contains the conversion logic

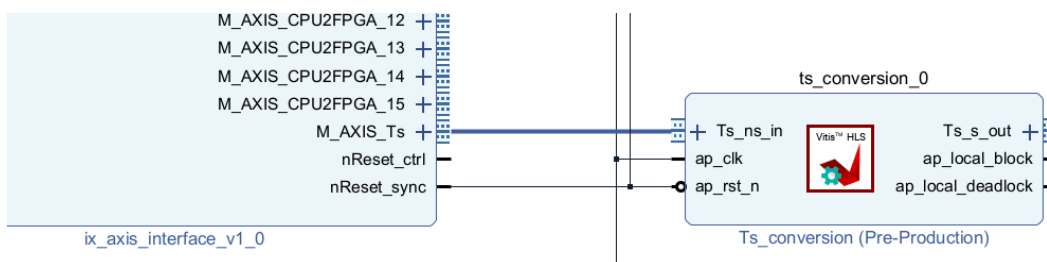


Content of the “adc” hierarchy

## Sample time ( $T_s$ ) conversion

The **M\_AXIS\_Ts** port of the “ix axis interface” provides the sample period  $T_s$  in **nanoseconds** in a **32-bit unsigned integer** format. This signal is the time distance between two consecutive samples.

The **Ts conversion module** shown below converts this signal into a **floating-point** value in **seconds**. The result will be used by the PI controllers of the Grid synchronization module and the dq current control module.



```

#include "Ts_conversion.h"

#include <hls_stream.h>
#include <stdint.h>

void ts_conversion(
    hls::stream<uint32_t>& Ts_ns_in,
    hls::stream<float>& Ts_s_out)
{

// see https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/pragma-HLS-interface
// "both" means registers are placed on TDATA, TVALID, and TREADY
#pragma HLS INTERFACE axis port=Ts_ns_in register_mode=both register
#pragma HLS INTERFACE axis port=Ts_s_out register_mode=both register
// turns off block-level I/O protocols
#pragma HLS interface ap_ctrl_none port=return

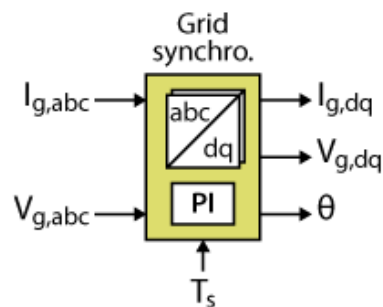
    uint32_t Ts_ns = Ts_ns_in.read();
    // convert from nanoseconds to seconds
    float Ts_s = (float)Ts_ns * 0.000000001f;
    Ts_s_out.write(Ts_s);
}Code language: C++ (cpp)

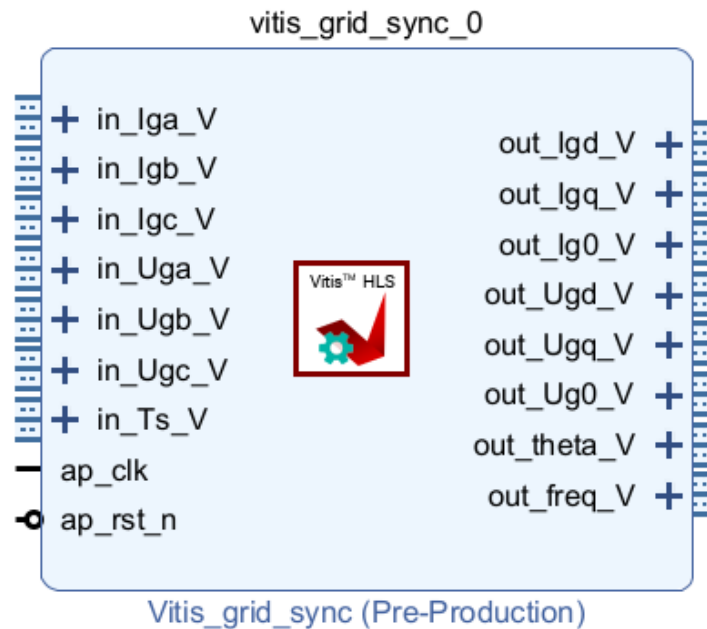
```

## Grid synchronization

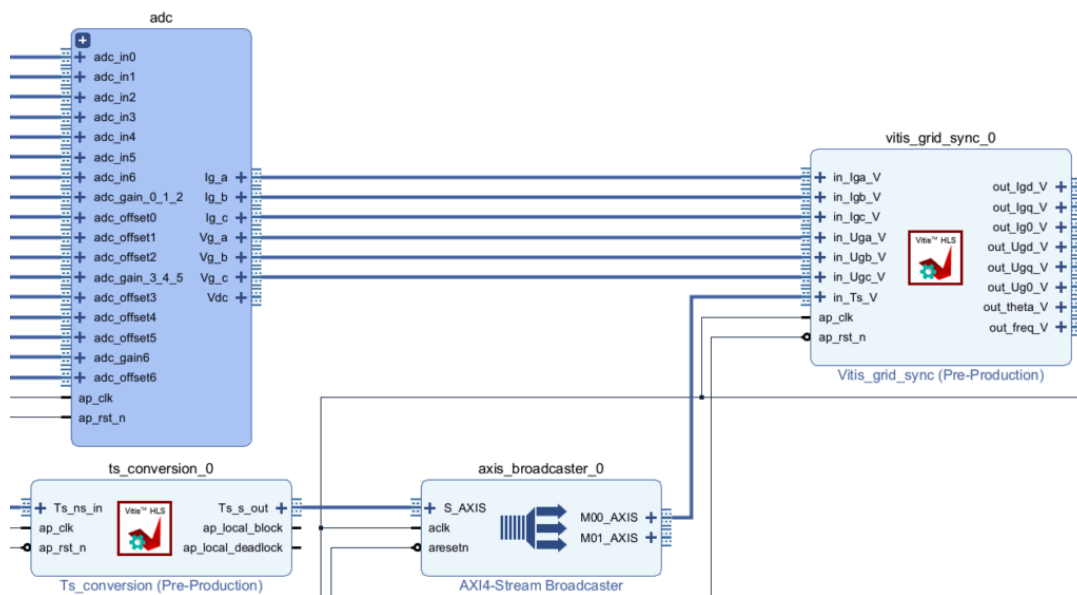
The grid synchronization is done using the **dq-type PLL** which transforms both the grid voltages and currents into dq components, that are used in the dq current controller.

The grid synchronization IP shown below is documented in the [FPGA impl. of a PLL for grid sync](#) page.



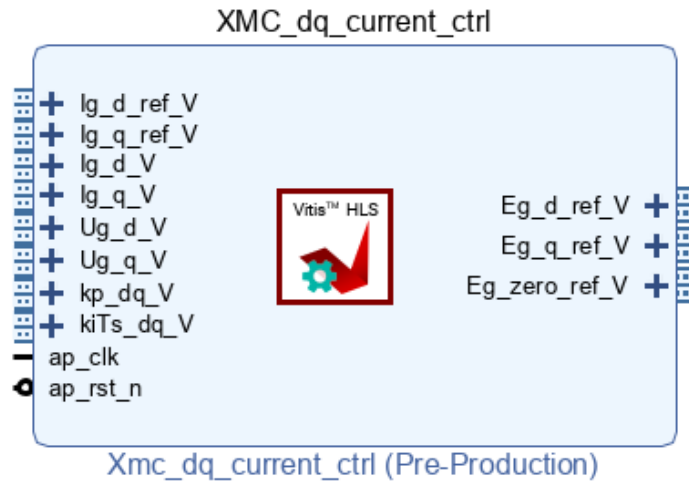
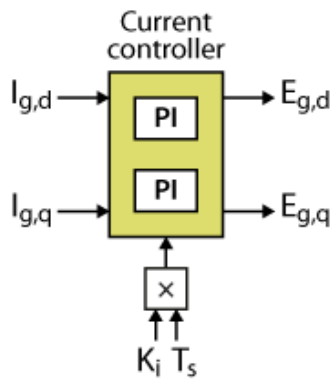


In the Vivado project, the IP is connected as follows. An **AXI4-Stream Broadcaster** is used because the sample time (Ts) signal is also connected to the dq current controller as shown in the next section.

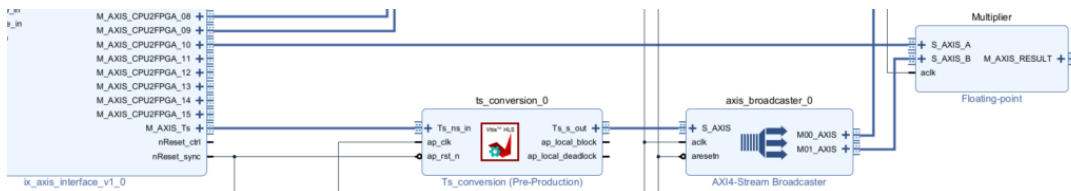


## DQ current control

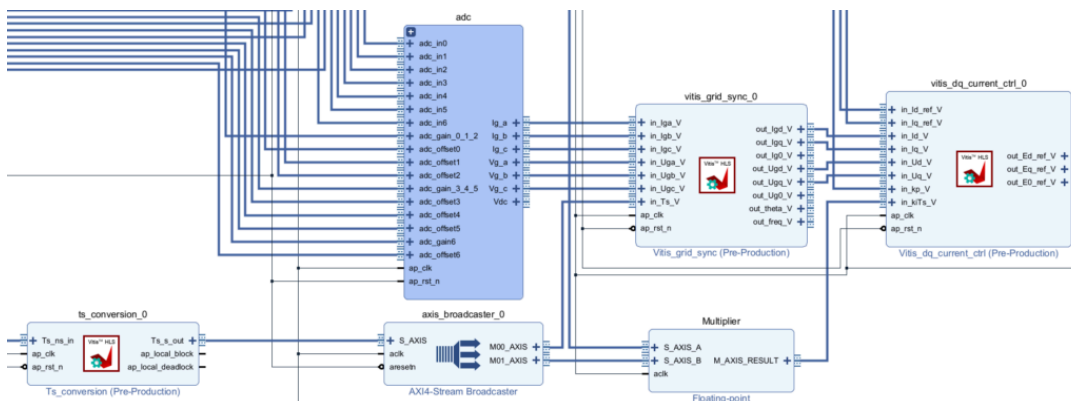
The implementation of the dq current controller is detailed in [FPGA-based PI for dq current control](#). It consists of two identical PI controllers with a decoupling network for independent control of the d- and q-components of the grid current.



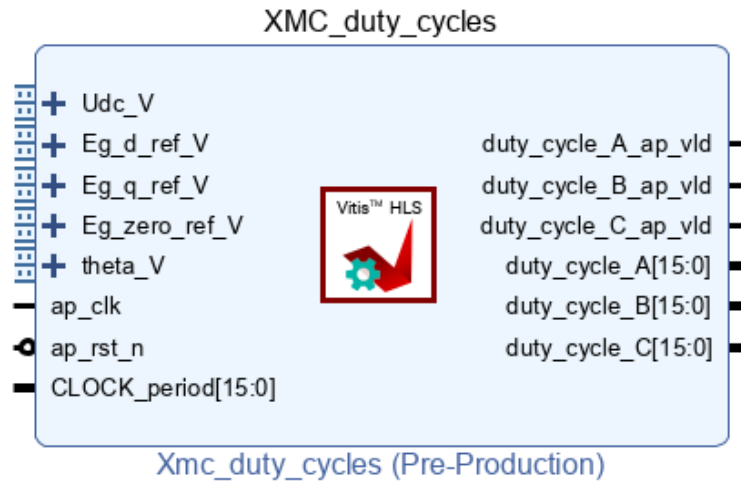
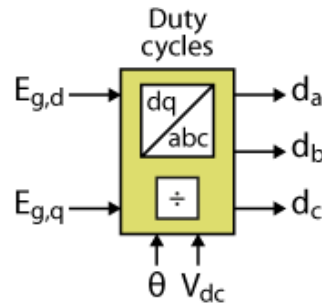
The **kiTs\_dq** port takes as input the results of **Ts** multiplied by **Ki** (Ki is a parameter set from the CPU, using the port **CPU2FPGA\_10**). The multiplication is performed using a **Floating-Point IP** as shown below.



The dq current controller is connected as shown below. The inputs **Kp**, **Id\_ref**, and **Iq\_ref** are connected to **CPU2FPGA** ports **11**, **12**, and **13**.



## Duty cycles computation

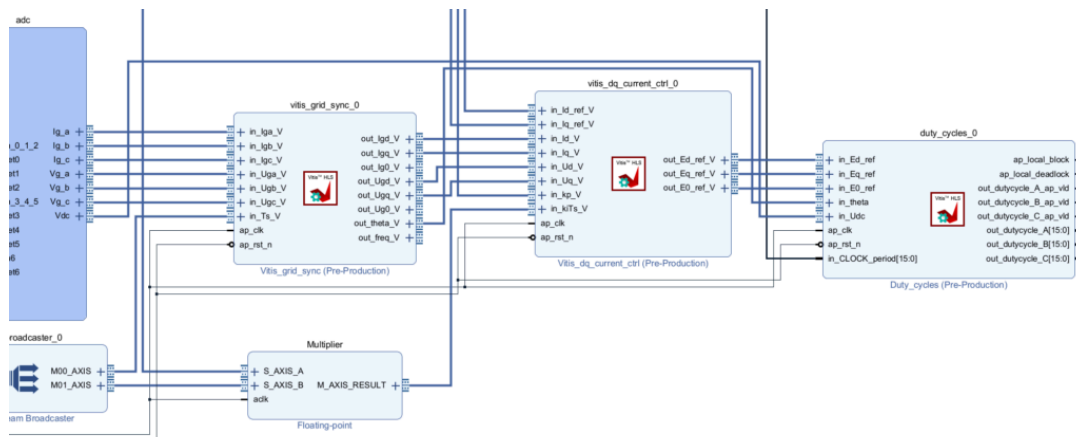


This block converts the voltage references computed by the dq current controller to  $abc$  quantities  $E_{g,abc}$ , and computes the corresponding duty cycles (in the range [0,1]) according to

$$d_{abc} = \left( \frac{E_{g,abc}}{V_{dc}} + 0.5 \right) \cdot T_{clk},$$

where  $T_{clk}$  is the period of the clock used in the PWM modulator, expressed in ticks (1 tick = 4 ns). The duty cycles are converted into uint16 numbers with a unit of ticks to be compatible with the PWM modulator.

It is connected as shown below. **in\_CLOCK\_period** is connected to **CLOCK\_1\_period** of the IMPERIX\_FW IP.



**Vitis HLS – Duty cycles computation**

```
#include "duty_cycles.h"

#include <hls_stream.h>
#include <stdint.h>
```

```

#include "ap_fixed.h"
#include "hls_math.h"

void dq02abc(float d, float q, float zero, float wt, float& A, float& B, float& C)
{
#pragma HLS inline

    const ap_fixed<16,2> sqrt3_2 = 0.86602540378444;// sqrt(3)/2

    ap_fixed<32,16> d_fix    = (ap_fixed<32,16>)d;
    ap_fixed<32,16> q_fix    = (ap_fixed<32,16>)q;
    ap_fixed<32,16> zero_fix = (ap_fixed<32,16>)zero;
    ap_fixed<16,4> wt_fix    = (ap_fixed<16,4>)wt;

    ap_fixed<16, 2> cos_wt = hls::cos(wt_fix);
    ap_fixed<16, 2> sin_wt = hls::sin(wt_fix);

    ap_fixed<32,16> alpha_fix = d_fix*cos_wt - q_fix*sin_wt;
    ap_fixed<32,16> beta_fix  = d_fix*sin_wt + q_fix*cos_wt;

    ap_fixed<32,16> A_fix = alpha_fix + zero_fix;
    ap_fixed<32,16> B_fix = zero_fix - alpha_fix/2 + sqrt3_2*beta_fix;
    ap_fixed<32,16> C_fix = zero_fix - alpha_fix/2 - sqrt3_2*beta_fix;

    A = (float)A_fix;
    B = (float)B_fix;
    C = (float)C_fix;
}

float sat(float input, float max_sat, float min_sat)
{
#pragma HLS inline

    if(input > max_sat) {
        return max_sat;
    } else if(input < min_sat) {
        return min_sat;
    } else {
        return input;
    }
}

void vitis_duty_cycles( hls::stream<float>& in_Udc,
    hls::stream<float>& in_Ed_ref,
    hls::stream<float>& in_Eq_ref,
    hls::stream<float>& in_E0_ref,
    hls::stream<float>& in_theta,
    uint16_t in_CLOCK_period,
    uint16_t& out_dutycycle_A,
    uint16_t& out_dutycycle_B,
    uint16_t& out_dutycycle_C)
{
#pragma HLS INTERFACE axis port=in_Udc register_mode=both register
#pragma HLS INTERFACE axis port=in_Ed_ref register_mode=both register
#pragma HLS INTERFACE axis port=in_Eq_ref register_mode=both register
#pragma HLS INTERFACE axis port=in_E0_ref register_mode=both register
#pragma HLS INTERFACE axis port=in_theta register_mode=both register
#pragma HLS interface ap_ctrl_none port=return

```

```

float Udc = in_Udc.read();
float Ed_ref = in_Ed_ref.read();
float Eq_ref = in_Eq_ref.read();
float E0_ref = in_E0_ref.read();
float theta = in_theta.read();

float A,B,C;
dq02abc(Ed_ref, Eq_ref, E0_ref, theta, A, B, C);

float next_d_A = A/Udc + 0.5;
float next_d_B = B/Udc + 0.5;
float next_d_C = C/Udc + 0.5;

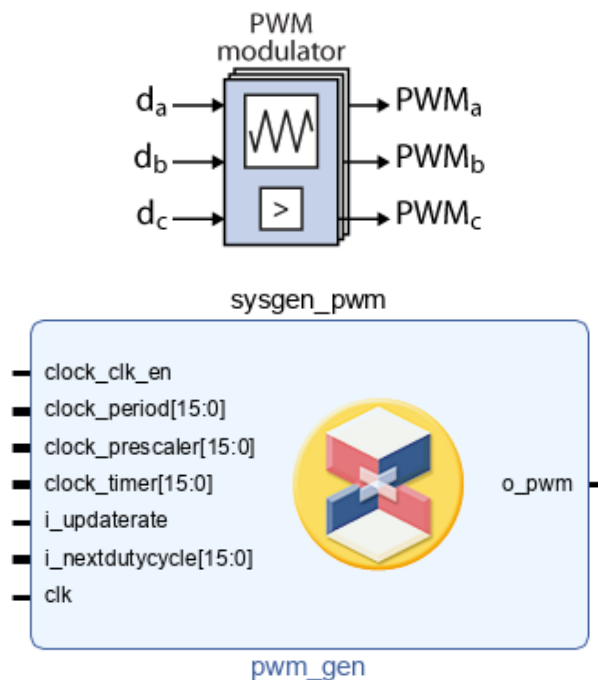
float d_A = sat(next_d_A, 1.0, 0.0);
float d_B = sat(next_d_B, 1.0, 0.0);
float d_C = sat(next_d_C, 1.0, 0.0);

ap_fixed<16,2> d_A_fix = (ap_fixed<16,2>)d_A;
ap_fixed<16,2> d_B_fix = (ap_fixed<16,2>)d_B;
ap_fixed<16,2> d_C_fix = (ap_fixed<16,2>)d_C;

out_dutycycle_A = (uint16_t)(d_A_fix * in_CLOCK_period);
out_dutycycle_B = (uint16_t)(d_B_fix * in_CLOCK_period);
out_dutycycle_C = (uint16_t)(d_C_fix * in_CLOCK_period);
}
Code language: C++ (cpp)

```

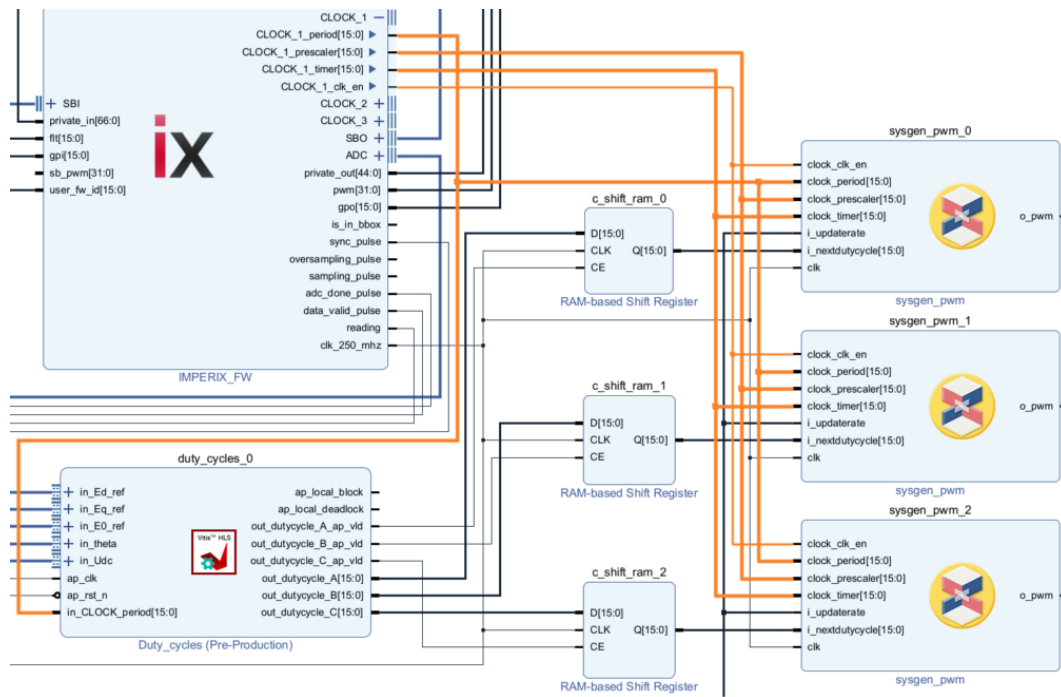
## PWM generation



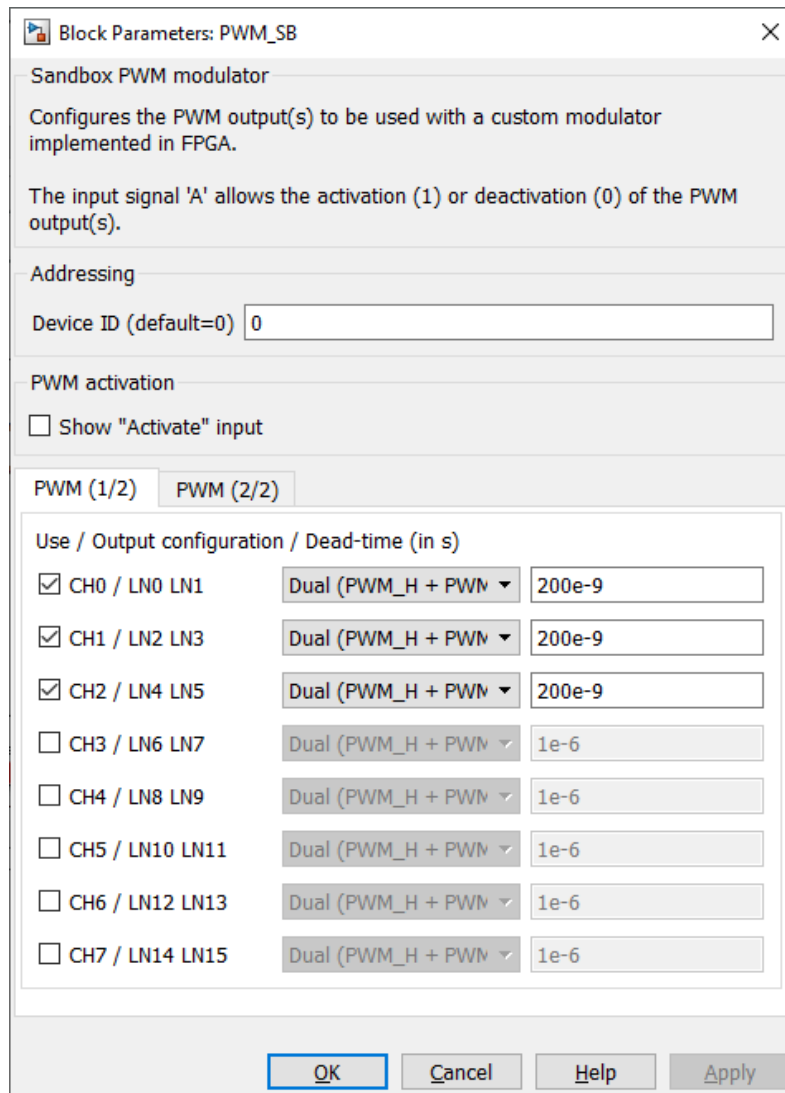
Finally, the duty cycles are transformed into PWM signals in the PWM modulator block. The implementation details are presented in [PWM modulator implementation in FPGA](#).

The PWM block uses the clock signal `CLOCK_1` as a reference to generate the PWM triangular carrier signal. The switching frequency is therefore equal to the frequency of `CLOCK_1`, which

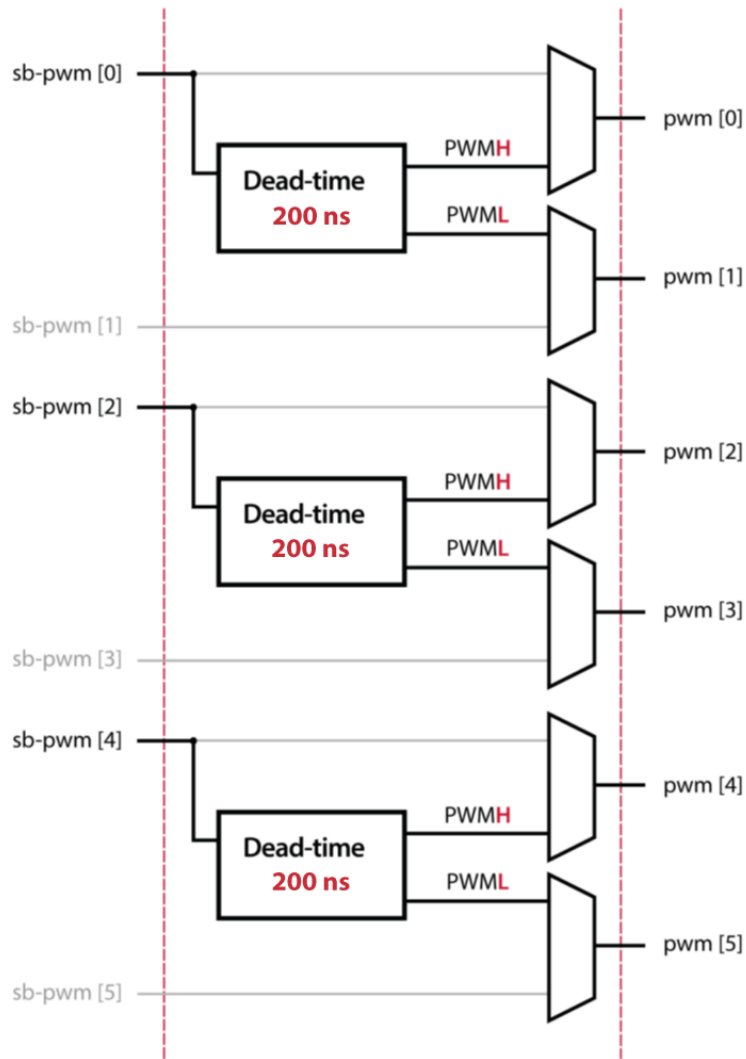
can be configured using a [CLK](#) block.



The high-side switching signals are obtained by comparing the 3 duty cycles  $d_{abc}$  with the triangular carrier. The generation of the low-side switching signals is done by the [SB-PWM](#) driver incorporated into the imperix IP and the dead time duration is specified in the CPU block [Sandbox PWM configurator](#).

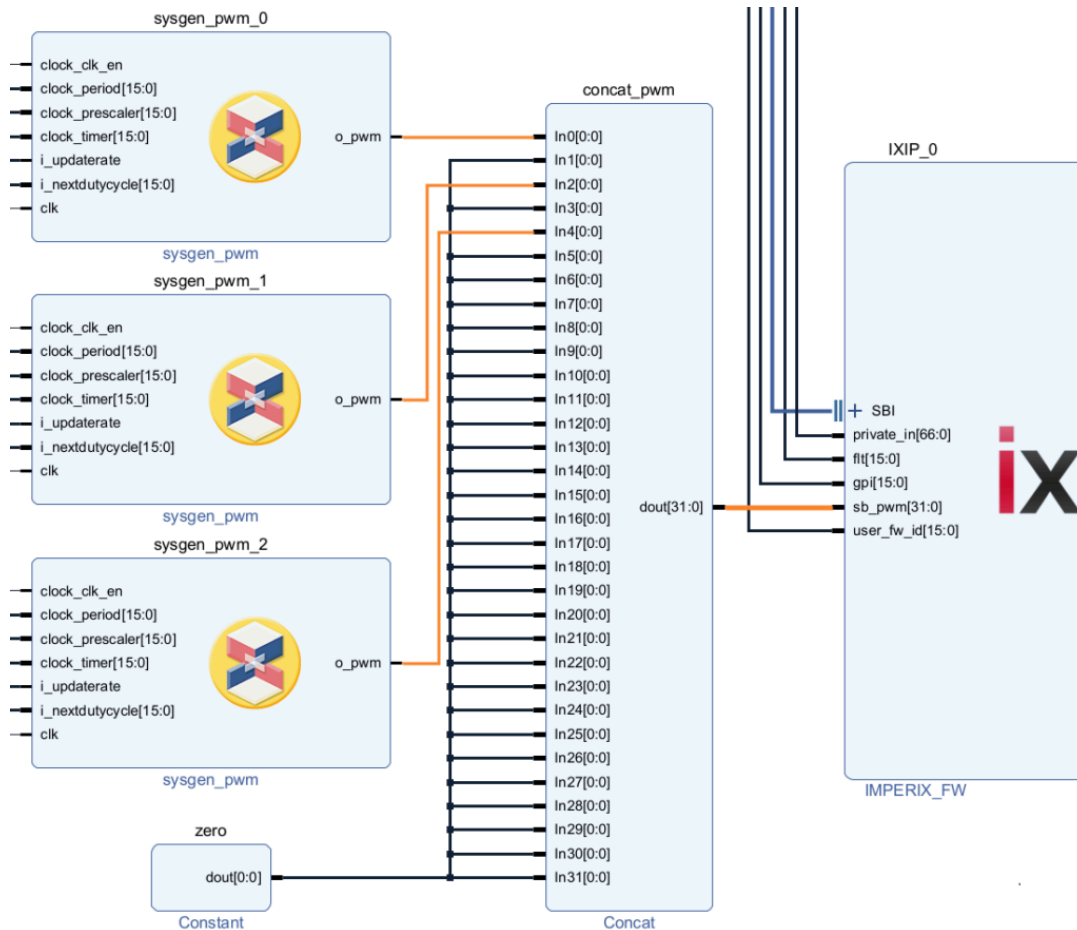


Configuration of the SB-PWM block in Simulink



Mapping between sb\_pwm and pwm ports of the imperix IP in Vivado

Thanks to the SB-PWM driver, the safety mechanism of the B-Box RCP is also available for custom-made PWM modulators. In case an over-value is detected during operation, the PWM outputs are immediately blocked and the operation is safely stopped.



## CPU-side implementation

The CPU model shown below is only used for the following tasks:

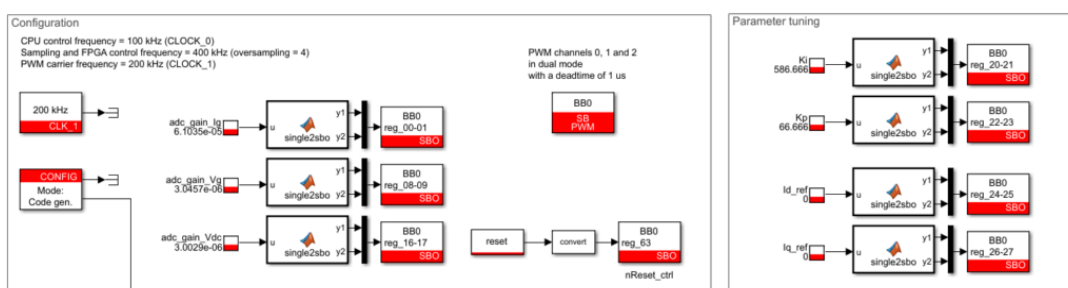
### Configuration:

- Configure the sampling frequency using the [Configuration block](#) (CLOCK\_0 = 100 kHz, oversampling = 4)
- Configure the modulator clocks using the [CLK block](#) (CLOCK\_1 = 200 kHz)
- Configure the ADC conversion parameters (gain and offset)

### Parameter tuning:

- Tune the controller gains (Kp and Ki)
- Transfer the dq current references  $I_{g,d}^*$  and  $I_{g,q}^*$  to the FPGA

The mapping between SBI/SBO registers and CPU2FPGA/FPGA2CPU port is explained on the [Getting started](#) page.

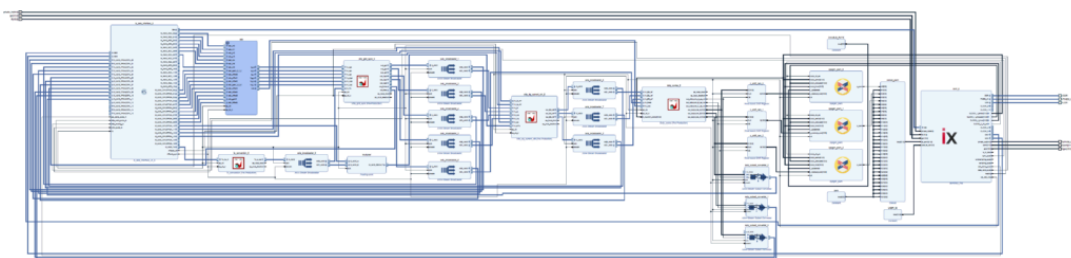


# Debugging and monitoring

For debugging and monitoring purposes, internal signals of the FPGA inverter control model can be split using **AXI4-Stream Broadcaster** IPs and routed to **FPGA2CPU** ports of the imperix IP. This way, the signals can be accessed from the CPU, connected to [probe variable](#) blocks, and observed using [Cockpit](#).

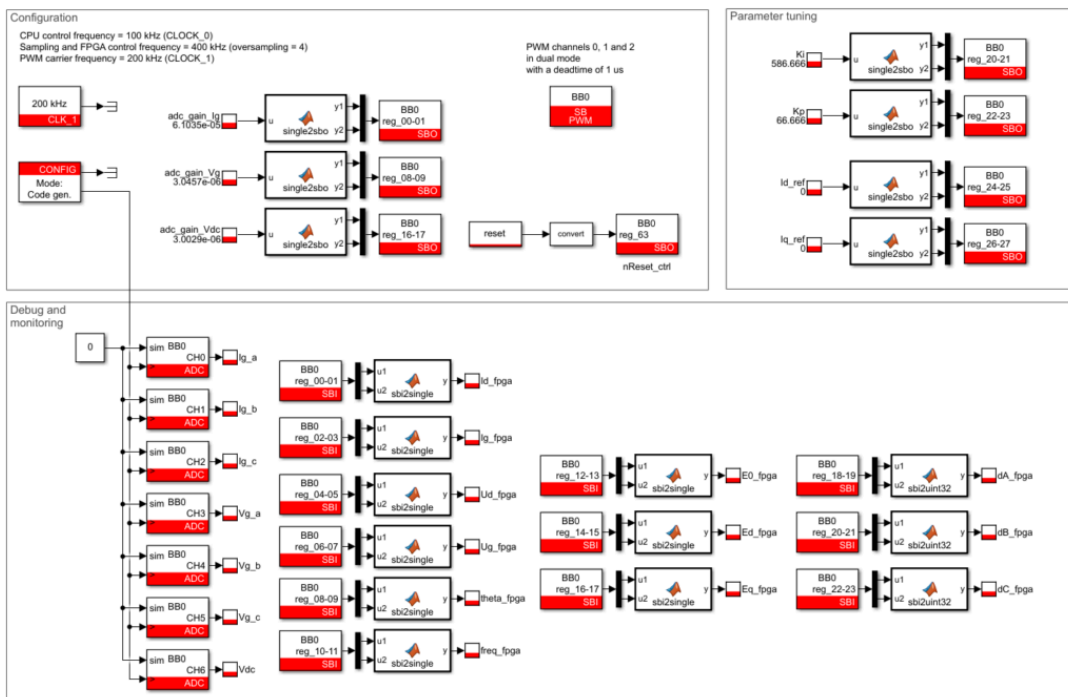
Additionally, ADC blocks can still be used to observe the analog input signals. For accurate readings, make sure the sensitivities of the ADC blocks match the gain parameter sent to the FPGA!

Please note that if the FPGA control is running faster than the CPU, then the CPU will only see a downsampled version of the observed signals.



Vivado block design with debug probes

[Download TN147\\_block\\_design\\_with\\_debug\\_probes.pdf](#)



Complete CPU model with debug probes and ADC blocks

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