DQ current control using FPGA-based PI controllers

TN144 | Posted on July 14, 2021 | Updated on May 7, 2025

Shu WANG
Development Engineer
imperix • in

Table of Contents

Control algorithms for power electronics converters often rely on PI controllers executed on the CPU of the controller. That's the technique used in most of the application notes on this knowledge. However, in some situations, it could be desired to run the control loop on an FPGA (e.g. to offload the CPU, or achieve much faster control rates). In that case, it is required to develop an FPGA-based PI controller.

The implementation of an **FPGA-based PI controller** is given as an example in the introductory notes to Vitis HLS and Model Composer. These are the two recommended approaches to develop FPGA-based control routines. In most realizations, however, the control of three-phase AC variables requires two PI controllers running in a synchronous reference frame (dq), as introduced in the note about CPU-based vector current control.

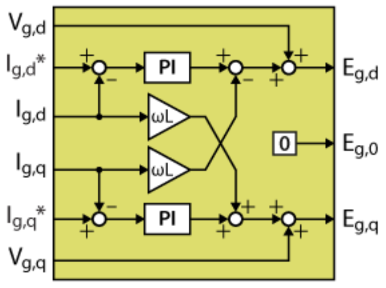
This page gives an example of an **FPGA-based dq PI controller for current control of a grid-tied three-phase inverter** using both Vitis HLS and Model Composer approaches. Testbenches are also presented and help to validate that the implemented code or model behaves as expected, before even executing it on a real-time controller.

For traditional CPU-based dq current control, please refer to vector current control.

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

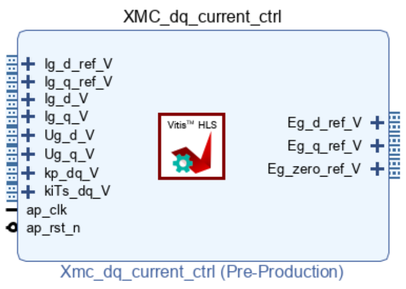
Overview of the dq current control module

The implemented algorithm is a traditional vector (dq) current control, including a PI controller on each axis and a decoupling network. The measured grid voltages are feedforwarded right after the current controllers, giving voltage quantities $E_{g,dq0}$ that the converter should generate to induce the desired current. The zero-sequence voltage is here set to zero but can be used for instance for third-harmonic injection, as done in the TN146.



Traditional dq current control algorithm

The following sections detail how to pack that algorithm into the FPGA IP shown below, using both High-Level Synthesis tools Vitis HLS and Model Composer. That IP uses *AXI4-Stream* inputs and outputs to be compatible with other IPs developed on other pages, as well as with Xilinx IP cores for Vivado. In particular, this IP is meant to be connected to the grid synchronization IP developed in FPGA-based grid synchronization.



Generated IP implementing the dq current control

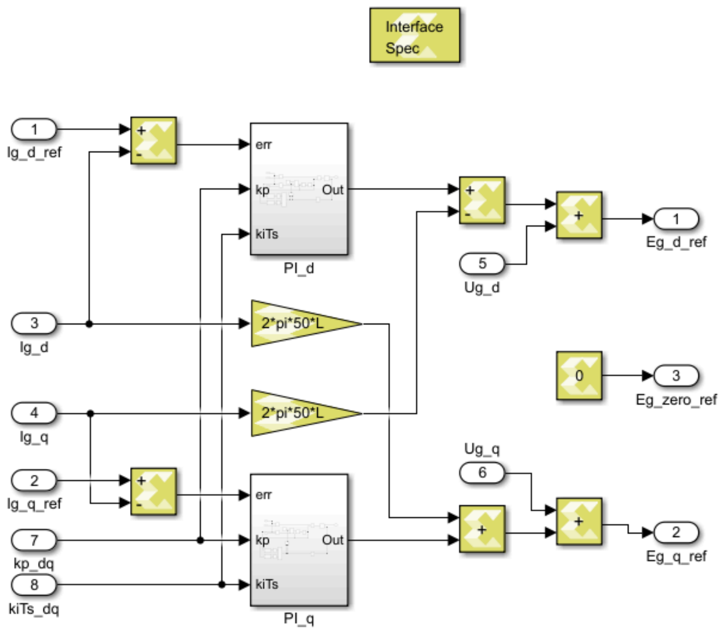
Further details on integrating a Vivado IP in the FPGA are presented in [High-Level Synthesis integration into the FPGA](). A complete converter control algorithm that uses that IP is presented in [FPGA-based converter control]().

FPGA-based dq current control using Model Composer

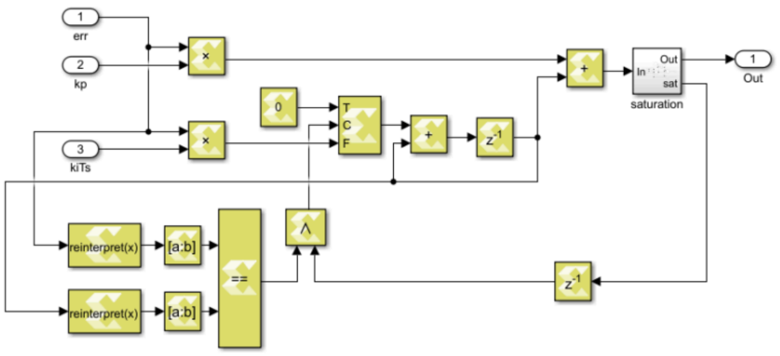
The Model Composer implementation of the FPGA-based vector current controller is shown below.

For more details on how to generate an IP and how to integrate it to the FPGA of imperix controllers, please refer to the [introductory note to Model Composer]().

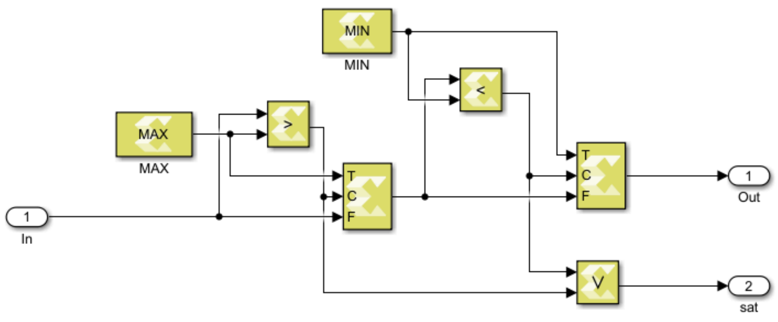
[Download **TN144_FPGA_DQ_Current_Control_Model_Composer.zip**]()



Implementation of the FPGA-based PI controller using Model Composer



PI_d and PI_q subsystems

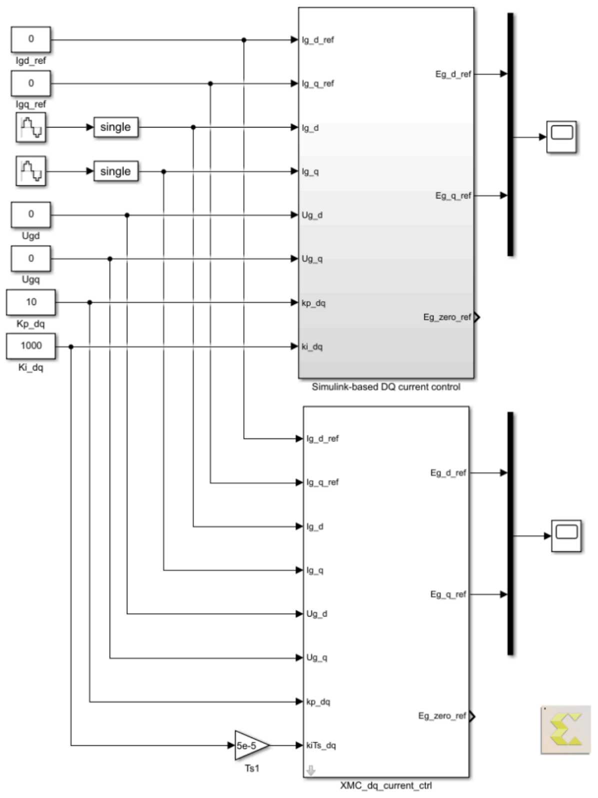


saturation subsystem

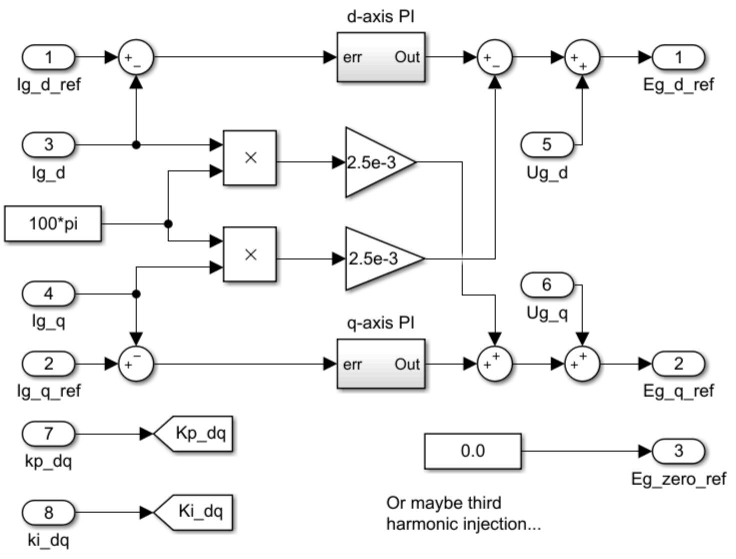
Testbench of the Model Composer implementation

Model Composer allows the user to test his implementation directly from within Simulink in an offline simulation. For comparison purposes with the Vitis HLS approach, the same inputs as the Vitis testbench are fed to the Model Composer model.

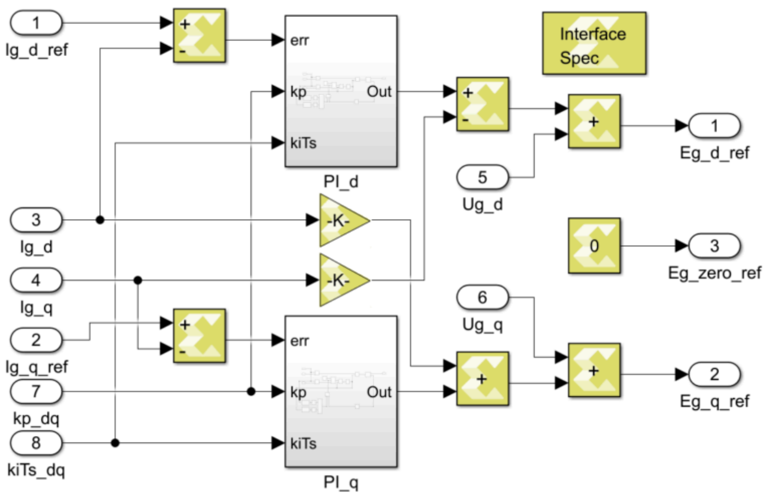
A previously-validated Simulink implementation of the vector current controller serves as a reference for both Vitis HLS and Model Composer testbenches.



Simulink testbench with reference model (top) and Model Composer implementation (bottom)



Reference model



Model under test (Model Composer)

FPGA-based dq current control using Xilinx Vitis HLS

The complete Vitis HLS sources of the FPGA-based vector current controller can be downloaded below:

Download **TN144_FPGA_DQ_Current_Control_Vitis_HLS.zip**

Below is the C++ code of the algorithm used in this example, for reference. From there, this code can be used to generate an IP and integrate it into the FPGA of an imperix controller, following the instructions of the introductory note to Vitis HLS.

Vitis HLS source of the dq current controller (C++)

```
#include "dq_current_ctrl.h"

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


float saturation(float in, bool& sat)
{
  #pragma HLS INLINE

  if(in > MAX_SAT) {
    sat = 1;
    return MAX_SAT;
  } else if(in < MIN_SAT) {
    sat = 1;
    return MIN_SAT;
  } else {
    sat = 0;
    return in;
    }
}

void pi_d(float kp, float kiTs, float Id, float Id_ref, float& pi_out_d)
{
  #pragma HLS INLINE

  static float accum_d = 0;
  static bool sat_d = 0;
  #pragma HLS RESET variable=accum_d
  #pragma HLS RESET variable=sat_d

  float err_d,P_d,I_d,next_out_d;
  bool clamp_d;

  err_d = Id_ref - Id;
  P_d = kp*err_d;
  I_d = kiTs*err_d;

  clamp_d = (hls::signbit(accum_d) == hls::signbit(err_d)) & sat_d;
  if(!clamp_d) {
    accum_d += I_d;
  }

  next_out_d = accum_d + P_d;
  pi_out_d = saturation(next_out_d,sat_d);
}

void pi_q(float kp, float kiTs, float Iq, float Iq_ref, float& pi_out_q)
{
#pragma HLS INLINE

  static float accum_q = 0;
  static bool sat_q = 0;
  #pragma HLS RESET variable=accum_q
  #pragma HLS RESET variable=sat_q

  float err_q = Iq_ref - Iq;
  float P_q = kp*err_q;
  float I_q = kiTs*err_q;

  bool clamp_q = (hls::signbit(accum_q) == hls::signbit(err_q)) & sat_q;
  if(!clamp_q) {
    accum_q += I_q;
  }

  float next_out_q = accum_q + P_q;
  pi_out_q = saturation(next_out_q,sat_q);
}

void vitis_dq_current_ctrl(
  hls::stream<float>& in_Id_ref,
  hls::stream<float>& in_Iq_ref,
  hls::stream<float>& in_Id,
  hls::stream<float>& in_Iq,
  hls::stream<float>& in_Ud,
  hls::stream<float>& in_Uq,
  hls::stream<float>& in_kp,
  hls::stream<float>& in_kiTs,
  hls::stream<float>& out_Ed_ref,
  hls::stream<float>& out_Eq_ref,
  hls::stream<float>& out_E0_ref)
{
  #pragma HLS INTERFACE axis port=in_Id_ref
  #pragma HLS INTERFACE axis port=in_Iq_ref
  #pragma HLS INTERFACE axis port=in_Id
  #pragma HLS INTERFACE axis port=in_Iq
  #pragma HLS INTERFACE axis port=in_Ud
  #pragma HLS INTERFACE axis port=in_Uq
  #pragma HLS INTERFACE axis port=in_kp
  #pragma HLS INTERFACE axis port=in_kiTs
  #pragma HLS INTERFACE axis port=out_Ed_ref
  #pragma HLS INTERFACE axis port=out_Eq_ref
  #pragma HLS INTERFACE axis port=out_E0_ref
  #pragma HLS interface ap_ctrl_none port=return

  float Id_ref = in_Id_ref.read();
  float Iq_ref = in_Iq_ref.read();
  float Id = in_Id.read();
  float Iq = in_Iq.read();
  float Ud = in_Ud.read();
  float Uq = in_Uq.read();
  float kp = in_kp.read();
  float kiTs = in_kiTs.read();

  float pi_out_d, pi_out_q,Ed_ref,Eq_ref,E0_ref;

  pi_d(kp, kiTs, Id, Id_ref, pi_out_d);
  pi_q(kp, kiTs, Iq, Iq_ref, pi_out_q);

  Ed_ref = -Iq*wL + Ud + pi_out_d;
  Eq_ref = Id*wL + Uq + pi_out_q;
  E0_ref = 0;

  out_Ed_ref.write(Ed_ref);
  out_Eq_ref.write(Eq_ref);
  out_E0_ref.write(E0_ref);
}
```
Code language: C++ (cpp)

Note that the integrators are represented as accumulators, using static variables. The directive `#pragma HLS RESET` is used to control the reset of integrators (e.g. when the control is not active, the integrator should be kept at reset). Since the polarity of the reset signal will be automatically set to active low when using AXI4, it is not necessary to specify it explicitly.

Note also that in the codes above, the integral gain is $K_i T_s$ instead of K_i, which means that in the Vivado project, a floating-point multiplier is needed to compute $K_i \cdot T_s$ before this module. The reason to do this is not technical, but just to be consistent with the Model Composer implementation (see below), which is limited to 8 input/output ports.

Testbench of the Vitis HLS implementation

Simulating the closed-loop control behavior of the developed current controller in Vitis HLS is uneasy since it would require a model of the control process (i.e. model of the PWM modulators, of the power converter, and of the acquisition of the measurements). Instead, we can run the Vitis HLS controller using the same input signals as an already-validated Simulink model, and validate the implementation by comparing the results.

The input signals of this testbench are the following two sine signals (the other inputs are kept constant for simplification):

$$\begin{aligned}&I_d = 5\sin(100\pi \cdot nT_s)\\&I_q = 5\sin(100\pi \cdot nT_s + \frac{\pi}{2})\\&n = 0, 1, 2, \ldots, 999\\&T_s = 5 \times 10^{-5}\end{aligned}$$

The simulation results are stored in a CSV file and plotted with MATLAB on top of the results of the Simulink (and Model Composer) models for comparison. The comparison is done at the bottom of this page.

The testbench is executed using the following code:

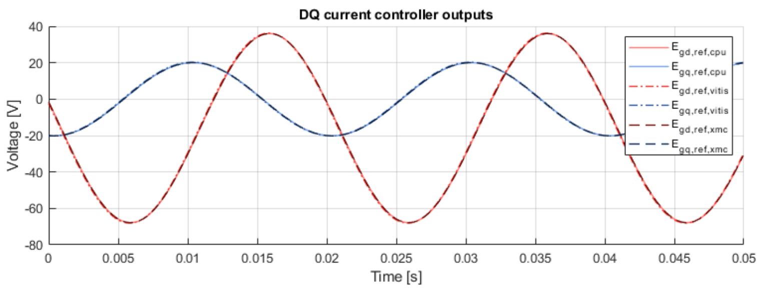
Vitis HLS dq current controller testbench

```
#include "dq_current_ctrl.h"

#include "ap_fixed.h"
#include "math.h"
#include <hls_stream.h>
#include <stdint.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

#define ITERATIONS 1000
#define two_pi 6.283185307f

int main()
{
        /* Test signals */
        hls::stream<float> tb_in_Id_ref;
        hls::stream<float> tb_in_Iq_ref;
        hls::stream<float> tb_in_Id;
        hls::stream<float> tb_in_Iq;
        hls::stream<float> tb_in_Ud;
        hls::stream<float> tb_in_Uq;
        hls::stream<float> tb_in_kp;
        hls::stream<float> tb_in_kiTs;
```

```cpp
        hls::stream<float> tb_out_Ed_ref;
        hls::stream<float> tb_out_Eq_ref;
        hls::stream<float> tb_out_E0_ref;

        /* Test inputs*/
        float in_Id_ref = 0;
        float in_Iq_ref = 0;
        float in_Id;
        float in_Iq;
        float in_Ud = 0;
        float in_Uq = 0;
        float in_kp = 10;
        float in_kiTs = 0.05;

        float angle = 0;
        float Ts = 0.00005;             // fs = 20kHz
        float wt = two_pi*50*Ts;        // fgrid = 50Hz

        /* Test outputs */
        float out_Ed_ref;
        float out_Eq_ref;

        /* Create csv file to store simulation data*/
        FILE *outputs;
        outputs = fopen ("outputs.csv", "w+");
        fprintf(outputs,"%s","Ed_ref,Eq_ref\n");

        /* Run model */
        for (int i = 0; i < ITERATIONS; i++)
        {
                angle = remainder((angle+wt),two_pi);
                in_Id = 5*sin(angle);
                in_Iq = 2*sin(angle+two_pi/4);

                tb_in_Id_ref.write(in_Id_ref);
                tb_in_Iq_ref.write(in_Iq_ref);
                tb_in_Id.write(in_Id);
                tb_in_Iq.write(in_Iq);
                tb_in_Ud.write(in_Ud);
                tb_in_Uq.write(in_Uq);
                tb_in_kp.write(in_kp);
                tb_in_kiTs.write(in_kiTs);

                vitis_dq_current_ctrl(tb_in_Id_ref,tb_in_Iq_ref,tb_in_Id,tb_in_Iq,tb_in_Ud,tb_in_Uq,tb_in_kp,tb_in

                out_Ed_ref = tb_out_Ed_ref.read();
                out_Eq_ref = tb_out_Eq_ref.read();

                fprintf(outputs,"%f,%f %s",out_Ed_ref,out_Eq_ref,"\n");
        }

        fclose(outputs);

        printf("------ Test Passed ------\n");

        /* End */
        return 0;
}
```
Code language: C++ (cpp)

Testbench results

The graph below shows that all three implementations have the same behavior, which validates that the Vitis HLS and Model Composer implementations are correct.



Testbench results with reference model (cpu), Vitis HLS (vitis), and Model Composer (xmc)

Further readings

A similar approach is used in the implementation of a grid synchronization module for the FPGA. Both these IPs are used in FPGA-based converter control.