

# FPGA-based Direct Torque Control using Vivado HLS

TN133 | Posted on April 2, 2021 | Updated on May 23, 2025



**Benoît STEINMANN**

Software Team Leader

imperix • in

---

## Table of Contents

- [Suggested prerequisites](#)
- [Software resources](#)
- [Design choices](#)
- [HLS C code implementation](#)
- [Vivado HLS testbench](#)
- [Observing comparator inputs](#)
- [Verifying PWM signals](#)
- [Deployment of the Vivado HLS code on the B-Box RCP](#)
  - [Synthesis result](#)
  - [Integrating the Vivado HLS design in the FPGA firmware](#)
- [CPU implementation \(using Simulink blockset\)](#)
- [Experimental results of the Vivado HLS Direct Torque Control](#)

This technical note presents an FPGA-based Direct Torque Control (DTC) of a PMSM motor using Vivado HLS, coupled with the possibility to customize the FPGA firmware of a B-Box. This approach increases the responsiveness of the DTC implementation presented in [AN004](#) by porting part of the control logic to the FPGA.

Xilinx [Vivado High-Level Synthesis](#) (HLS) is a tool that transforms C code into an RTL implementation that can be synthesized for an FPGA. The two main benefits are:

- It greatly facilitates the implementation of complex algorithms, as the designer can work at a higher level of abstraction (C/C++ code)
- It provides a higher system performance by offloading parts of the computations from the CPU to the FPGA and leverages the parallel architecture of the FPGA

Another example of high-level synthesis is presented in [TN121](#), which addresses automated HDL code generation using Matlab HDL Coder.

This example has been written before the release of the newest [FPGA control template](#), as such it does not implement the latest recommendations such as the use of AXI4-Stream interfaces.

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

## Suggested prerequisites

- Information on how to set up the toolchain for the FPGA programming is available in the [Vivado Design Suite installation](#) page.
- Quick-start information on how to use the *sandbox* is provided in the [getting started with FPGA](#) page.

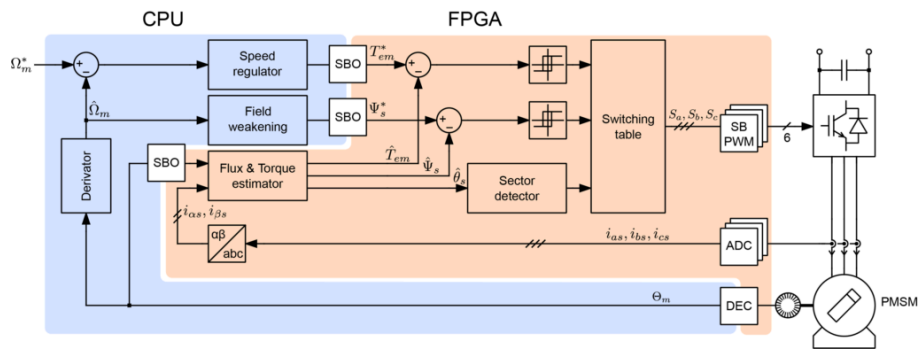
## Software resources

## Design choices

The DTC algorithm has been split into two parts:

- A *fast* part, implemented in the FPGA. This part requires a fast action to keep the torque and flux values within the hysteresis bounds. This corresponds to all the computations and logic resulting from the current measurements, which can be sampled at a high rate (typically 400 kHz).
- A *slow* part, implemented in the CPU and executed at the interrupt frequency (typically 40 kHz). This includes mainly the generation of the torque and flux references, which don't require to be updated as fast as the sampling of the currents, given that the mechanical dynamics are much slower than the electrical ones.

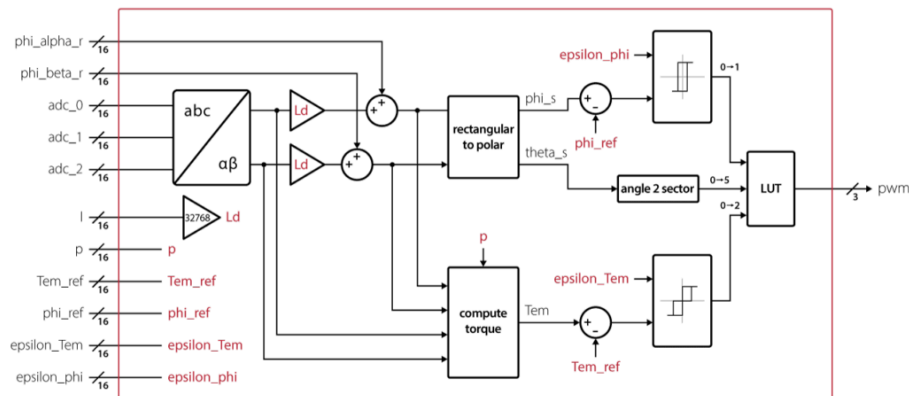
The two parts are illustrated below:



## HLS C code implementation

The logic ported to the FPGA is illustrated in the figure below. The ports are intended to be interfaced to the imperix firmware IP as follow:

- `adc_0, adc_1, adc_2`: connected to the ADC interface
- `phi_alpha_r, phi_beta_r, Tem_ref, phi_ref`: connected to SBO registers (real-time registers)
- `l, p, epsilon_Tem, epsilon_phi`: connected to SBO registers (configuration registers)
- `pwm`: connected to `sb_pwm`



The logic above has been translated into HLS C code. To derive an efficient hardware implementation, the following choices have been made:

- The inputs are 16-bit wide to be compatible with the SBO and ADC interfaces of the imperix IP
- The algorithm uses the ADC 16-bit results without applying any gain. This imposes to divide the setpoints coming from the CPU ( $T_{em\_ref}$ ,  $\phi_{ref}$ ,  $\epsilon_{Tem}$  and  $\epsilon_{phi}$ ) by the gain that would have been applied to the ADC before sending them to the FPGA.

- The internal logic uses fixed-point arithmetic to be fast enough to handle the 250 MHz clock of the imperix IP and avoid the need to perform clock-domain crossing (CDC).
- Ld ranges between 0.001 and 0.1. To convey its value to the FPGA using a 16-bit integer, the original value is multiplied by  $2^{15}$  from the CPU and re-divided by the same amount in the HLS implementation.

HLS implementation source code:

HLS C code of the DTC algorithm

```
#include "dtc.h"

#define ONE_OVER_SQRT_3 (d_t)0.577350269
#define ONE_OVER_THREE (d_t)0.333333333
#define TWO_OVER_THREE (d_t)0.666666666

void ComputeAbcToAlphaBeta(d_t a, d_t b, d_t c, d_t &alpha, d_t &beta);

void ComputeRectangularToPolar(d_t x, d_t y, d_t &r, d_t &theta);

d_t ComputeTorque(d_t a, d_t b, d_t c, d_t d, d_t p);

ap_uint<2> ComputeTemHystState(ap_uint<2> previous_state, d_t Tem_s, d_t ref_Tem, d_t epsilon_Tem);

ap_uint<1> ComputePhiHystState(ap_uint<1> previous_state, d_t phi_s, d_t ref_phi, d_t epsilon_phi);

ap_uint<3> ComputeSector(d_t theta);

static ap_uint<3> pwm_lut[2][3][6] =
{
    {
        {0b100, 0b101, 0b001, 0b011, 0b010, 0b110},
        {0b000, 0b111, 0b000, 0b111, 0b000, 0b111},
        {0b010, 0b110, 0b100, 0b101, 0b001, 0b011}
    },
    {
        {0b101, 0b001, 0b011, 0b010, 0b110, 0b100},
        {0b111, 0b000, 0b111, 0b000, 0b111, 0b000},
        {0b011, 0b010, 0b110, 0b100, 0b101, 0b001}
    }
};

#ifdef TESTBENCH
ap_uint<3> dtc(inputs &ins, outputs &outs)
#else
ap_uint<3> dtc(inputs &ins)
#endif
{
    d_t adc0 = ins.adc0 + ins.adc_offset0;
    d_t adc1 = ins.adc1 + ins.adc_offset1;
    d_t adc2 = ins.adc2 + ins.adc_offset2;

    // -----

    d_t i_alpha, i_beta;

    ComputeAbcToAlphaBeta(ins.adc0, ins.adc1, ins.adc2, i_alpha, i_beta);

    // -----

    d_t Ld = ((ap_fixed<W,I>)ins.l / 32768);

    d_t i_alpha_times_l = i_alpha * Ld;
    d_t i_beta_times_l = i_beta * Ld;

    d_t psi_alpha = i_alpha_times_l + ins.phi_alpha_r;
    d_t psi_beta = i_beta_times_l + ins.phi_beta_r;

    // -----
```

```

d_t phi_s;
d_t theta_s;

ComputeRectangularToPolar(psi_alpha, psi_beta, phi_s, theta_s);

// -----

d_t Tem_s;

Tem_s = ComputeTorque(psi_alpha, psi_beta, i_alpha, i_beta, ins.p);

// -----

static ap_uint<2> Tem_hyst_state = 0;

Tem_hyst_state = ComputeTemHystState(Tem_hyst_state, Tem_s, ins.Tem_ref, ins.epsilon_Tem);

// -----

static ap_uint<1> phi_hyst_state = 0;

phi_hyst_state = ComputePhiHystState(phi_hyst_state, phi_s, ins.phi_ref, ins.epsilon_phi);

// -----

ap_uint<3> sector = ComputeSector(theta_s);

// -----

#ifdef TESTBENCH

outs.debug_i_alpha = i_alpha;
outs.debug_i_beta = i_beta;
outs.debug_psi_alpha = psi_alpha;
outs.debug_psi_beta = psi_beta;
outs.debug_phi_s = phi_s;
outs.debug_theta_s = theta_s * 8;
outs.debug_Tem_s = Tem_s;
outs.debug_TemHystState = Tem_hyst_state;
outs.debug_PhiHystState = phi_hyst_state;
outs.debug_sector = sector;

#endif

// -----

return pwm_lut[phi_hyst_state][Tem_hyst_state][sector];
}

void ComputeAbcToAlphaBeta(d_t a, d_t b, d_t c, d_t &alpha, d_t &beta)
{
    alpha = a*ONE_OVER_THREE - b*TWO_OVER_THREE - c*TWO_OVER_THREE;
    beta = (b - c)*ONE_OVER_SQRT_3;
}

void ComputeRectangularToPolar(d_t x, d_t y, d_t &r, d_t &theta)
{
    ap_fixed<W*2, I*2, AP_TRN, AP_SAT> x_squared = x*x;
    ap_fixed<W*2, I*2, AP_TRN, AP_SAT> y_squared = y*y;

    r = hls::sqrt((ap_fixed<64, 32>)(x_squared + y_squared));

    #pragma HLS PIPELINE
    theta = hls::atan2((ap_fixed<W, I>)y, (ap_fixed<W, I>)x);
}

d_t ComputeTorque(d_t a, d_t b, d_t c, d_t d, d_t p)
{
    ap_fixed<W*2, I*2> a_times_d = a*d;
    ap_fixed<W*2, I*2> b_times_c = b*c;

```

```

    ap_fixed<W, I> temp = (ap_fixed<W*2, I*2>)(a_times_d - b_times_c) / 4096;

    return (d_t)(1.5) * p * temp;
}

ap_uint<2> ComputeTemHystState(ap_uint<2> previous_state, d_t Tem_s, d_t ref_Tem, d_t epsilon_Tem)
{
    d_t ref_Tem_minus_Tem_s = ref_Tem - Tem_s;

    if(previous_state == 0)
    {
        if(ref_Tem_minus_Tem_s > epsilon_Tem)
        {
            return 2;
        }
        else if(ref_Tem_minus_Tem_s > 0)
        {
            return 1;
        }
    }
    else if(previous_state == 1)
    {
        if(ref_Tem_minus_Tem_s > epsilon_Tem)
        {
            return 2;
        }
        else if(ref_Tem_minus_Tem_s < - epsilon_Tem)
        {
            return 0;
        }
    }
    else
    {
        if(ref_Tem_minus_Tem_s < - epsilon_Tem)
        {
            return 0;
        }
        else if(ref_Tem_minus_Tem_s < 0)
        {
            return 1;
        }
    }

    return previous_state;
}

ap_uint<1> ComputePhiHystState(ap_uint<1> previous_state, d_t phi_s, d_t ref_phi, d_t epsilon_phi)
{
    d_t ref_phi_minus_phi_s = ref_phi - phi_s;

    if(previous_state == 0)
    {
        if(ref_phi_minus_phi_s > epsilon_phi)
        {
            return 1;
        }
    }
    else
    {
        if(ref_phi_minus_phi_s < - epsilon_phi)
        {
            return 0;
        }
    }

    return previous_state;
}

ap_uint<3> ComputeSector(d_t theta)
{

```

```

if(theta < 0)
    theta += (d_t)PI_FIXED * (d_t)2;

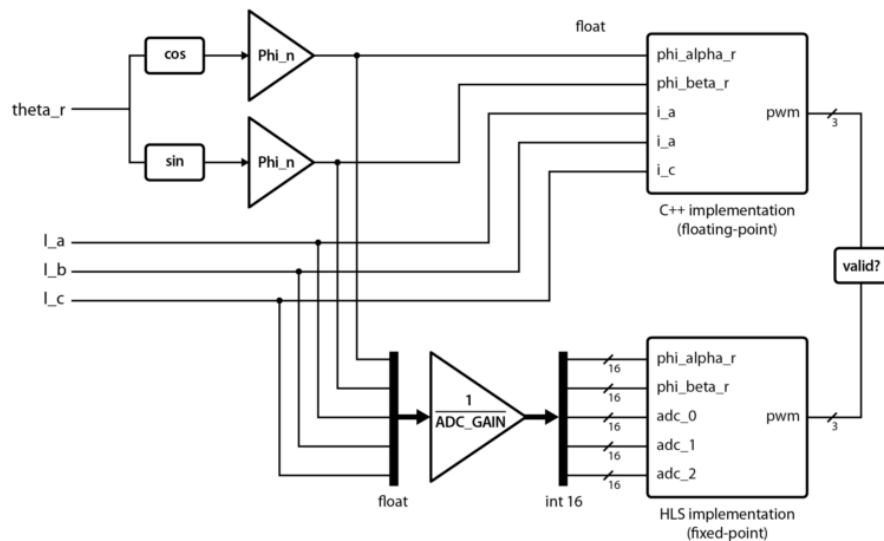
if(theta < (d_t)PI_FIXED/(d_t)6)
    return 0;
else if(theta < (d_t)PI_FIXED/(d_t)2)
    return 1;
else if(theta < (d_t)5*(d_t)PI_FIXED/(d_t)6)
    return 2;
else if(theta < (d_t)7*(d_t)PI_FIXED/(d_t)6)
    return 3;
else if(theta < (d_t)3*(d_t)PI_FIXED/(d_t)2)
    return 4;
else if(theta < (d_t)11*(d_t)PI_FIXED/(d_t)6)
    return 5;
else if(theta < (d_t)2*(d_t)PI_FIXED)
    return 0;

return 0;
}Code language: C++ (cpp)

```

## Vivado HLS testbench

Vivado HLS provides a C/C++ simulator to validate designs. As illustrated in the figure below, a test bench has been developed to compare the PWM signals of the HLS fixed-point implementation against a floating-point model which is algorithmically equivalent to the Simulink implementation presented in [AN004](#).



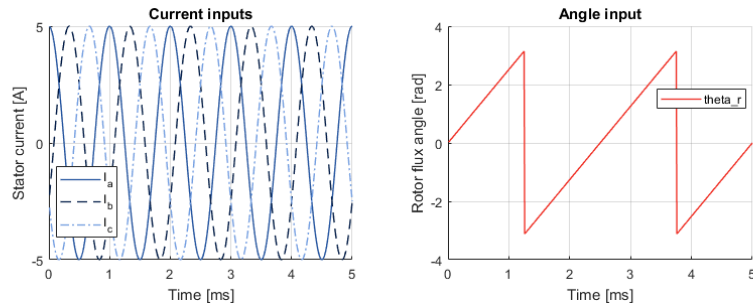
The test bench uses the following input signals:

- Three-phase currents  $I_a$ ,  $I_b$  and  $I_c$ : sinusoidal signals with a frequency of 1 kHz and an amplitude of 5 A. The signals are divided by ADC\_GAIN to obtain 16-bit values representing the results of the ADCs.
- Rotor flux angle  $\theta_r$ : a sawtooth with a frequency of 400 Hz and an amplitude of  $\pi$ .

The estimated torque (proportional to the sinus of the machine load angle



) will be sinusoidal, with a frequency of 1 kHz – 400 Hz = 600 Hz, as verified later.

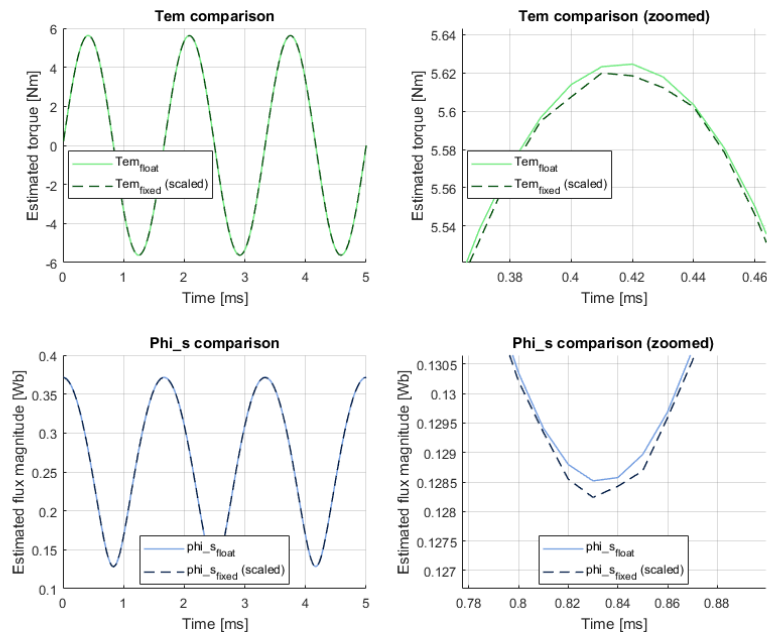


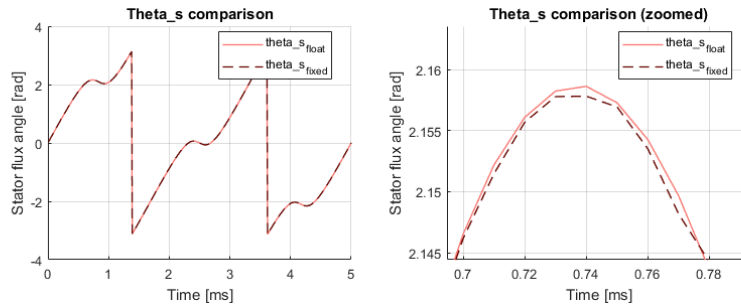
The following values are set to the other inputs:

input	C implementation	HLS implementation
Ld	0.0243	$0.0243 * 32768$
p	3	3
Tem_ref	1	$1 / \text{ADC\_GAIN}$
phi_ref	0.3	$0.3 / \text{ADC\_GAIN}$
epsilon_Tem	0.095	$0.095 / \text{ADC\_GAIN}$
epsilon_phi	0.005	$0.005 / \text{ADC\_GAIN}$

## Observing comparator inputs

Various intermediate signals are extracted and saved in CSV files. This allows for the easy plotting of the simulation results for visual verification, which greatly helps during the design phase. The estimator outputs (torque, flux and flux angle) are shown below. The zoomed graphs show the approximation stemming from the use of fixed-point arithmetic. These small differences sometimes lead to a hysteresis state difference between the two implementations when a signal is close to a comparator limit.





## Verifying PWM signals

Due to the approximation mentioned in the last section, we expect and tolerate that the PWM signals have a one-period difference. With this in mind, the following self-test mechanism has been implemented:

```
for (int i = 0; i < ITERATIONS; i++)
{
    // ... some code

    int pwm_float = dtc_float(tb_ins_float, tb_outs_float);

    // ... some code

    int pwm_dut = dtc(tb_ins, tb_outs);

    if((pwm_dut != pwm_float))
    {
        diffs++;
        printf("diff #%d at i = %d\n", diffs, i);

        // error if PWM are different for two iterations
        if(last_diff){
            errors++;
            printf("error #%d at i = %d\n", errors, i);
        }
        last_diff = 1;
    }
    else
    {
        last_diff = 0;
    }
}

if (errors > 0)
    printf("----- Test failed -----\n");
else
    printf("----- Test passed -----\n");

printf("Iterations: %d\n", ITERATIONS);
printf("Differences: %d\n", diffs);
printf("Errors: %d\n", errors); Code language: C++ (cpp)
```

The resulting output below confirms that the C++ and HLS implementation produce identical PWM outputs:

```
----- Test passed -----
Iterations: 1000000
Differences: 1376
Errors: 0 Code language: VHDL (vhdl)
```

## Deployment of the Vivado HLS code on the B-Box RCP

### Synthesis result



The HLS synthesis report shown below indicates that the latency of the module is 0.436  $\mu$ s, which is more than **13 times faster than the CPU-based implementation**. It also predicts that the design can run at a clock frequency of 250 MHz.

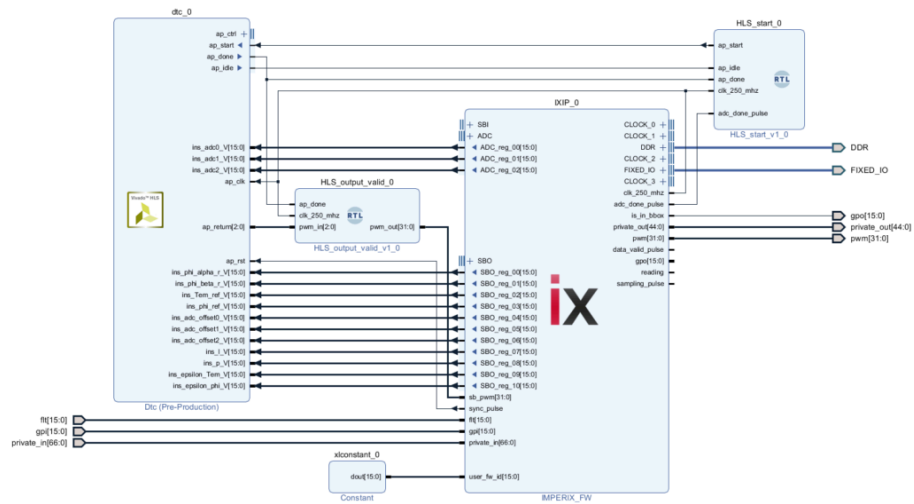
Performance Estimates			
Timing			
Summary			
Clock	Target	Estimated	Uncertainty
ap_clk	4.00 ns	3.489 ns	0.50 ns
Latency			
Summary			
Latency (cycles)		Latency (absolute)	
min	max	min	max
109	109	0.436 us	0.436 us
Interval (cycles)		min	max
		109	109
		Type	none

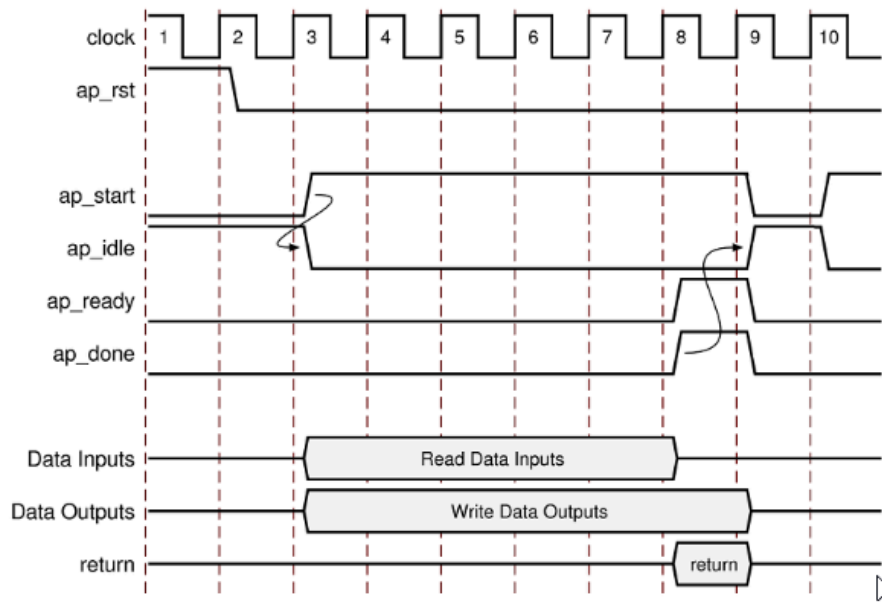
Utilization Estimates			
Summary			
Name	BRAM_18K	DSP48E	FF
DSP	-	-	-
Expression	-	-	0
FIFO	-	-	-
Instance	12	34	24988
Memory	0	-	3
Multiplexer	-	-	494
Register	-	-	1369
Total	12	34	26360
Available	530	40	157200
Utilization (%)	2	8	16

## Integrating the Vivado HLS design in the FPGA firmware

The IP generated from Vivado HLS is instantiated in a sandbox environment. Details on how to edit the firmware of the B-Box are given in [PN116](#). Instructions on how to set up the development environment are given in [PN120](#).



Two simple VHDL modules have been created for the design to comply with the block-level interface protocol defined in Vivado Design Suite User Guide [UG902](#).



The `HLS_start.vhd` module asserts the `ap_start` signal when an `adc_done_pulse` is detected and clear it when `ap_done` is asserted.

```
MY_PROCESS : process(clk_250_mhz)
begin
    if rising_edge(clk_250_mhz) then
        if adc_done_pulse = '1' and ap_idle = '1' then
            i_reg_ap_start <= '1';
        end if;
        if ap_done = '1' then
            i_reg_ap_start <= '0';
        end if;
    end if;
end process MY_PROCESS;
```

`ap_start <= i_reg_ap_start;`Code language: VHDL (vhd1)

The `HLS_output.vhd` module samples the HLS IP `pwm` output when `ap_done` is asserted and apply them to the appropriate `sb_pwm` input.

```
MY_PROCESS : process(clk_250_mhz)
begin
    if rising_edge(clk_250_mhz) then
        if ap_done = '1' then
            i_reg_pwm(0) <= pwm_in(0);
            i_reg_pwm(2) <= pwm_in(1);
            i_reg_pwm(4) <= pwm_in(2);
        end if;
    end if;
end process MY_PROCESS;
```


`pwm_out <= i_reg_pwm;`Code language: VHDL (vhd1)

## CPU implementation (using Simulink blockset)

The following configuration is used:

- SBO register 00: `phi_alpha_r` (real-time register)
- SBO register 01: `phi_beta_r` (real-time register)
- SBO register 02: `Tem_ref` (real-time register)
- SBO register 03: `phi_ref` (real-time register)
- SBO register 04: 1 (configuration register)
- SBO register 05 p (configuration register)
- SBO register 06: `epsilon_Tem` (configuration register)

- 
- The block diagram illustrates the motor control system architecture. It starts with configuration blocks (CONFIG, BB0, SB, PWM, BB0 config, SBO) that provide parameters to the main control blocks. The system includes a Vdc block for braking chopper control, a CB PWM block for duty cycle generation, and several ADC blocks for measuring Vdc, Ia, Ib, and Ic. These measurements are processed by a Derivator block to calculate speed. The speed is then used in a Field weakening block to generate a flux reference. The speed reference is also used in a Speed regulation block to generate a torque reference. The torque reference is then used in a Conversion3 block to generate a final output. The system also includes a Conversion1 block for angle measurement and a Conversion2 block for flux reference generation.


Block Parameters: PWM\_SB1

Sandbox PWM modulator

Configure the output PWM lanes or channels to be used with a custom modulator implemented in FPGA.

Addressing

B-Box number (default=0)

0

PWM activation

☐ Show "Activate" input

PWM (1/2)

PWM (2/2)

Use / Output configuration / Dead-time (in s)

<input checked="" type="checkbox"/>	CH0 / LN0 LN1	Dual (PWM_H + PW	1e-6	
<input checked="" type="checkbox"/>	CH1 / LN2 LN3	Dual (PWM_H + PW	1e-6	
<input checked="" type="checkbox"/>	CH2 / LN4 LN5	Dual (PWM_H + PW	1e-6	
<input type="checkbox"/>	CH3 / LN6 LN7	Dual (PWM_H + PW	1e-6	
<input type="checkbox"/>	CH4 / LN8 LN9	Dual (PWM_H + PW	1e-6	
<input type="checkbox"/>	CH5 / LN10 LN11	Dual (PWM_H + PW	1e-6	
<input type="checkbox"/>	CH6 / LN12 LN13	Dual (PWM_H + PW	1e-6	
<input type="checkbox"/>	CH7 / LN14 LN15	Dual (PWM_H + PW	1e-6	

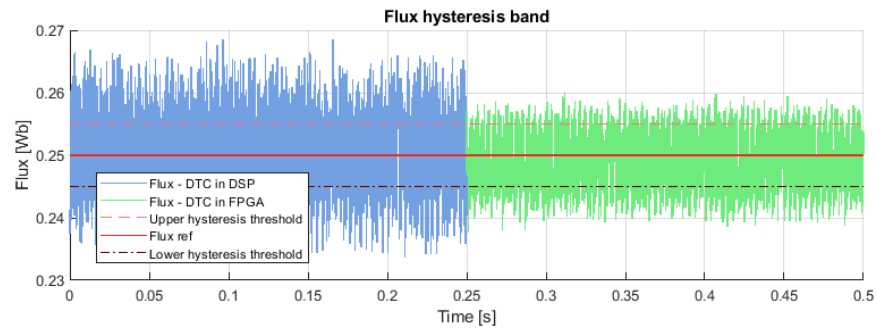
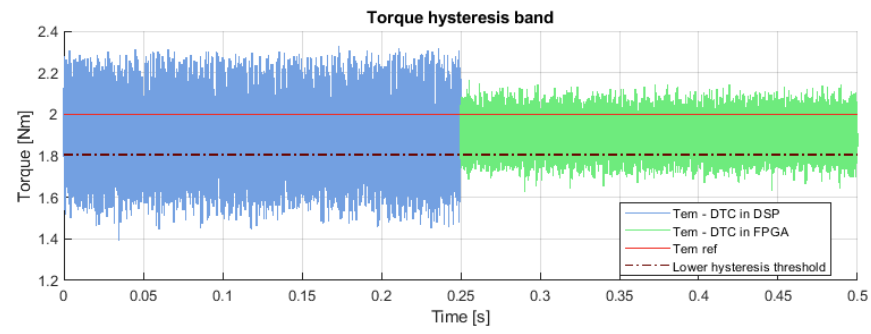
OK

Cancel

Help

Apply

The following graphs show a significant reduction of the torque and flux ripples thanks to the shorter control delay enabled by the FPGA-based implementation.



Back to [FPGA development homepage](#)