Chapter 3

Motion Control with FPGA

Miguel Angel Martínez Prado,
Juvenal Rodríguez Reséndiz,
Diana Carolina Toledo Pérez,
Carlos Miguel Torres Hernández and
Gilberto Herrera Ruiz

Additional information is available at the end of the chapter

http://dx.doi.org/10.5772/67200

Abstract

The aim of this chapter is to provide an introduction to the field programmable gate array (FPGA)-based digital control system design for motion control. It is intended as a reference for the undergraduate students in science and engineering, professionals, and enthusiastic people who have a basic knowledge in discrete control theory and digital systems using reconfigurable logic. The scope of this chapter includes the analysis, simulation, and implementation of classic control algorithms. The presented topics serve as a foundation for the implementation of more complex systems. An experimental section is provided, which validates the proposed digital design.

Keywords: FPGA, motion-control, PID-control, VHDL, robotics

1. Introduction

The reconfigurable logic in industries opened countless opportunities especially in the field of control and automation. This technology facilitates the implementation of complex control algorithms with fast response.

Nowadays, the control system engineers require new tools for creating better electronic design for automation systems. Some modern tools that are available on the market allow the designer to create, simulate, and verify the desired hardware design. This can help in the evaluation of the complex system designs with fewer resources.

Among modern tools used by the controllers, the field programmable gate array (FPGA) provides a shorter processing time than the conventional methods like microprocessor- or

microcontroller-based designs. Furthermore, it has benefits such as improved accuracy and efficiency of the algorithms.

In the industry, the FPGA technology began to be used by the designers in areas like telecommunications, signal processing, image processing, and control systems such as robotic arms and assembly lines. Later, this technology began to be utilized in applications where the fast processing of information is desired, such as medical equipment, robotics, aeronautics, etc. [1].

Proportional integral derivative (PID) controller is one of the most commonly used design, due to its simple design and its robustness with respect to the parameter uncertainty [2–4]. They are usually used in the speed controlling applications of direct current or permanent magnet motors, through pulse-width modulation (PWM) pulses [5], output current, voltage, or frequency. It is possible to find out in the scientific literature PID implementations on hardware [6–8] whose authors have demonstrate the effectiveness of their designs; however, most of these implementations are not easy to develop and for some cases they are destined to be implemented only in some FPGA families.

In order to enhance the designer experience, the FPGA card manufacturers incorporate multicore processors equipped with flash memory into their designs for enhancing the computing capacity and data parallel processing. In this way, the controllers can implement functions that require fast processing in hardware and computationally intensive algorithms into the processor.

Implementing functions into the FPGA chip, the platforms that are available on the market works with HDL codes, which decreases hardware resource use and therefore, at the same time, reduces the cost and energy consumption of the system. Moreover, these platforms manage simulators for assessment of the design before its implementation.

Section 2 of this chapter is related to the digital controllers, which describes the PID and other controllers in discrete form. Section 3 provides the hardware description of the PID controller in VHDL language, and finally, the fourth section provides the simulation and experimental validation, which demonstrates how to perform numerical simulations using Simulink and Modelsim. Furthermore, an experimental validation on a DC motor system is also provided.

2. Discretization of classical controllers

The proportional-integral derivative (PID) controller is widely used in industry due to its high performance with most of the plants even if they are nonlinear [2]. Besides, its parameters can be tuned empirically and still achieve a good performance. Due to the complexity of the algorithm, its implementation has been limited to microcontrollers or digital signal processors [3], and furthermore, most of the researchers who are experts in control theory do not have a deep knowledge on reconfigurable logic [4, 5].

PID controller has the following form:

http://dx.doi.org/10.5772/67200

$$u(t) = K_p\left[e(t) + \frac{1}{T_i}\int_0^t e(\tau)d\tau + T_d\frac{de(t)}{dt}\right] \tag{1}$$

where $e(t)$ is the difference between the desired ($w(t)$) and measured ($y(t)$) response of the system, i.e.:

$$e(t) = w(t) - y(t) \tag{2}$$

and $u(t)$ is the control signal, used to control the actuator's operation to obtain a desired closed-loop performance. Finally, the parameters K_p, T_i, and T_d are the proportional gain, integral, and derivative time constants, respectively. A more popular form of Eq. (1) can be obtained by using the Laplace transform as follows:

$$U(s) = \left(K_p + \frac{K_i}{s} + K_d s\right)E(s), \tag{3}$$

where $K_i = \frac{K_p}{T_i}$ and $K_d = K_p T_d$.

Eqs. (2) and (3) are time-dependent functions; therefore, they cannot be implemented directly into a digital system. In that case, it is necessary to find out the discrete form of Eq. (1) by applying numerical methods.

The proportional part of the equation does not require any additional transformation because it involves a simple multiplication, but the integral and derivative require a numerical approximation. First, the integral of the error function can be considered as the sum of the area of small rectangles of base longitude of T_s (which is commonly termed as the sampling period), and height $e(k)$ at a given time instant, $t = kT_s$, i.e.:

$$\int_0^t e(\tau)d\tau \approx T_s\sum_{i=1}^{k} e(i). \tag{4}$$

Similarly, the derivative term can be approximated as:

$$\frac{de(t)}{dt} \approx \frac{e(k) - e(k-1)}{T_s} \tag{5}$$

for a given time $t = kT_s$. Now substituting Eqs. (4) and (5) into Eq. (1), it is possible to rewrite the PID controller in its discrete form as:

$$u(k) = K_p\left\{e(k) + \frac{T_s}{T_i}\sum_{i=1}^{k} e(i) + \frac{T_d}{T_s}[e(k) - e(k-1)]\right\} \tag{6}$$

Eq. (6) provides the storage of error samples from $t = 0$ until $t = kT_s$, which can be easily implemented by software on a microprocessor or DSP target. It is common to have kilobytes of memory in microprocessor platforms and such storage does not carry any problem; however,

when we deal with reconfigurable logic, it is of vital importance to save logic resources; therefore, a more suitable form of Eq. (6) is needed. Above is achieved by computing the differential term $\Delta u(k)$ instead of computing directly $u(k)$. Let us define the differential term $\Delta u(k)$ as:

$$\Delta u(k) = u(k) - u(k-1), \tag{7}$$

and

$$u(k-1) = K_p\left\{e(k-1) + \frac{T_s}{T_i}\sum_{i=1}^{k-1} e(i) + \frac{T_d}{T_s}[e(k-1) - e(k-2)]\right\} \tag{8}$$

Subtracting Eq. (8) from Eq. (6) yields:

$$\Delta u(k) = K_p\left\{e(k) - e(k-1) + \frac{T_s}{T_i}e(k) + \frac{T_d}{T_s}[e(k) - 2e(k-1) + e(k-2)]\right\} \tag{9}$$

From Eq. (7), it is possible to rewrite the control output $u(k)$ in terms of $u(k-1)$ and $\Delta u(k)$ as:

$$u(k) = \Delta u(k) + u(k-1). \tag{10}$$

It is worth to note that during the first iteration, i.e., for $t = kT_s = 0$, the term $u(k-1)$ becomes zero, while for subsequent iterations, this term holds the previously computed value of $u(k)$. Finally, substituting Eq. (9) in Eq. (10), the PID control law becomes

$$u(k) = K_p\left\{e(k) - e(k-1) + \frac{T_s}{T_i}e(k) + \frac{T_d}{T_s}[e(k) - 2e(k-1) + e(k-2)]\right\} + u(k-1) \tag{11}$$

The common terms in Eq. (11) can be grouped so that the control law takes the form of a digital filter, i.e.:

$$u(k) = q_0 e(k) + q_1 e(k-1) + q_2 e(k-2) + u(k-1) \tag{12}$$

where

$$q_0 = K_p\left(1 + \frac{T_s}{T_i} + \frac{T_d}{T_s}\right)$$

$$q_1 = -K_p\left(1 + 2\frac{T_d}{T_s}\right)$$

$$q_2 = K_p\frac{T_d}{T_s}$$

Proceeding with the same analysis, the reader could easily derive the formulas for a proportional-integral (PI) digital controller, which has the form:

$$u(t) = q_0 e(k) + q_1 e(k-1) + u(k-1) \tag{13}$$

where $q_0 = K_p\left(1 + {}^{T_s}/_{T_i}\right)$ and $q_1 = -K_p$. Similarly, the proportional-derivative (PD) controller may be written as:

http://dx.doi.org/10.5772/67200

$$u(t) = q_0 e(k) + q_1 e(k-1) + q_2 e(k-2) + u(k-1) \tag{14}$$

where

$$q_0 = K_p\left(1 + \frac{T_d}{T_s}\right)$$

$$q_1 = -K_p\left(1 + 2\frac{T_d}{T_s}\right)$$

$$q_2 = K_p\frac{T_d}{T_s}$$

Other controllers represented in the Laplace domain can be discretized by using approximations, e.g., the Tustin formulae:

$$s = \frac{2(z-1)}{T_s(z+1)} \tag{15}$$

For example, let us consider the following lead compensator:

$$\frac{U(s)}{E(s)} = k\frac{s+\omega_1}{s+\omega_2} \tag{16}$$

Substituting Eq. (15) in Eq. (16), we obtain:

$$\frac{U(z)}{E(z)} = k\frac{\frac{2(z-1)}{T_s(z+1)} + \omega_1}{\frac{2(z-1)}{T_s(z+1)} + \omega_2}$$

$$= k\frac{2(z-1) + \omega_1 T_s(z+1)}{2(z-1) + \omega_2 T_s(z+1)}$$

$$= k\frac{(\omega_1 T_s + 2)z + \omega_1 T_s - 2}{(\omega_2 T_s + 2)z + \omega_2 T_s - 2}$$

$$= k\frac{(\omega_1 T_s + 2)z + \omega_1 T_s - 2}{z + \frac{\omega_2 T_s - 2}{\omega_2 T_s + 2}}$$

$$= k\left(\frac{\omega_1 T_s + 2}{\omega_2 T_s + 2}\right)\frac{z + \frac{\omega_1 T_s - 2}{\omega_1 T_s + 2}}{z + \frac{\omega_2 T_s - 2}{\omega_2 T_s + 2}}$$

The above equation can be rewritten as:

$$\frac{U(z)}{E(z)} = K\frac{z+A}{z+B} \tag{17}$$

where

$$K = k\frac{\omega_1 T_s + 2}{\omega_2 T_s + 2}$$

$$A = \frac{\omega_1 T_s - 2}{\omega_1 T_s + 2}$$

$$B = \frac{\omega_2 T_s - 2}{\omega_2 T_s + 2}$$

From a digital point of view, Eq. (17) is still inconvenient. In order to obtain a suitable digital representation, it is necessary to represent this equation as a difference equation. This can be performed by multiplying the numerator and the denominator of the right-hand side of Eq. (17) by z^{-1}. This is equivalent to the shifting operation in the time domain, where the signal is delayed by one sample. Thus, the lead compensator takes the following form:

$$\frac{U(z)}{E(z)} = K\frac{1+Az^{-1}}{1+Bz^{-1}}$$

Further simplification yields

$$U(z)(1+Bz^{-1}) = KE(z)(1+Az^{-1})$$

Expanding terms:

$$U(z) + BU(z)z^{-1} = KE(z) + KAE(z)z^{-1}$$

Solving the above equation for $U(z)$ we have:

$$U(z) = KE(z) + KAE(z)z^{-1} - BU(z)z^{-1}$$

It is well known that:

$$E(z) = e(k) \tag{18}$$

and

$$E(z)z^{-1} = e(k-1) \tag{19}$$

therefore, the discrete lead compensator filter can be expressed as:

$$u(k) = Ke(k) + KAe(k-1) - Bu(k-1) \tag{20}$$

which is quite similar to Eq. (14).

http://dx.doi.org/10.5772/67200

3. Hardware description

There are important features that the reader must consider before starting the description process. First, the nature of the feedback signal should be considered. If the sensor which measures the variable to be controlled has an analogue nature, it is necessary to use an analogue to digital converter (ADC) which has an output with a fixed bit width. In order to avoid performing arithmetic operations between signals of different bit width, it is strongly suggested that the setpoint or reference has the same bit width as the measured variable. Additionally, if the error signal has a wide bus width, let us say wider than 16 bits, this signal can be saturated in order to avoid wider bus widths in preceding computations.

The second aspect to consider is the number and characteristics of the embedded multipliers or DSP slices that the target device possesses. Most of the FPGAs available in the market have multipliers with a fixed bus width, 18 × 18 bits for instance. For any other bus-width, the synthesis tool shall use logic resources to build a customized multiplier instead of using those available in the hardware. This bad practice leads to major resource utilization.

In summary, the error signal and the controller gains shall have a bus width, which matches the bus width of the available embedded multipliers. For example, the available multipliers have an 18 × 18 bus width, the error signal has 16 bits width, and the controller gains are required to have a fixed point format 16.16. Then a possible solution that does not imply the usage of a customized multiplier is to expand the bus width of error and gains to 18 and 36 bits, respectively, being in the latter signal, 18 bits for the integer part and the remaining 18 bits for the fractional part. Thus, the synthesis tool would use two 18 × 18 embedded multipliers. Above is possible whenever the bus width of error signal and gains be the multiple of the bus width of the embedded multipliers and while the extension of the signals preserves their signs.

Finally, the controller output must be congruent with the nature of the actuators. If we deal with an analogue actuator, it is necessary to include a digital to analogue converter (DAC) which, as the case of the ADC, has a fixed bus width; therefore, the controller output must agree such width. However, depending on the polynomial degree of the filter and the bus width of the used multipliers, the controller output eventually could have a wider bus width; so, it would be necessary to saturate and truncate decimal data from the controller output signal.

In this section, we shall describe the design of a PID digital controller; however, the reader could easily modify the proposed design for most of the control above laws. The resultant design is implemented in VHDL; it is validated in a cosimulation environment, and finally, it is tested in a real-life application to control the position of a brushed DC servo motor.

The PID digital filter seen as a black-box module is depicted in **Figure 1**. The ERR signal represents the error signal, which is the difference between the setpoint and the feedback data. Similarly, the signals Q0, Q1, and Q2 are the controller gains, and their value depends on K_p, T_i, T_d, and T_s as it is explained in previous sections. On the other side, the UOUT signal is the filter output, which is feed forwardly to the actuator. CLK and RST are the master clock and master reset signals, respectively.

Table 1 summarizes the signals properties of the PID_Digital_Controller module. It is important to clear that the error signal and the controller gains have 16 and 32 bits of width,

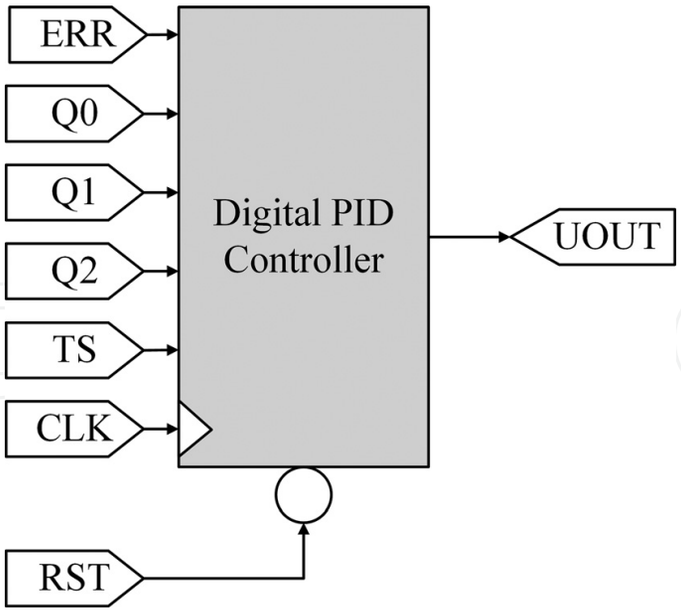

Figure 1. Black-box module of PID digital filter.

Signal name	Direction	Bus width	Description
RST	Input	1	Active low master reset
CLK	Input	1	Master clock
TS	Input	1	Sampling signal
ERR	Input	16	Two's complement error signal
Q0	Input	32	Filter coefficient q0
Q1	Input	32	Filter coefficient q1
Q2	Input	32	Filter coefficient q2
UOUT	Output	16	Controller output

Table 1. List of signals properties of PID digital controller.

respectively; above to match the standard data size of most of the common programming languages; however, these signals are internally expanded to 18 and 36 bits in order to use two 18 × 18 embedded multipliers as previously mentioned. Additionally, the controller gains are given in a fixed point format 16.16, i.e., the 16 most significant bits represent the integer part while the 16 less significant represent the fractional part.

Figure 2 illustrates a block diagram of the digital PID controller. This module comprises five standard load registers, two multiplexors, a multiplier, an adder, a saturator, and a finite state machine (FSM). White blocks represent pure combinational processes, whereas gray ones represent sequential and synchronous processes.

The data path starts at the input registers. At this point, the multiplexors bypass the corresponding signal error and controller gains selected by the SEL signal, which is driven by the FSM, the multiplier and adder accumulate this product with the previous result and so on

http://dx.doi.org/10.5772/67200

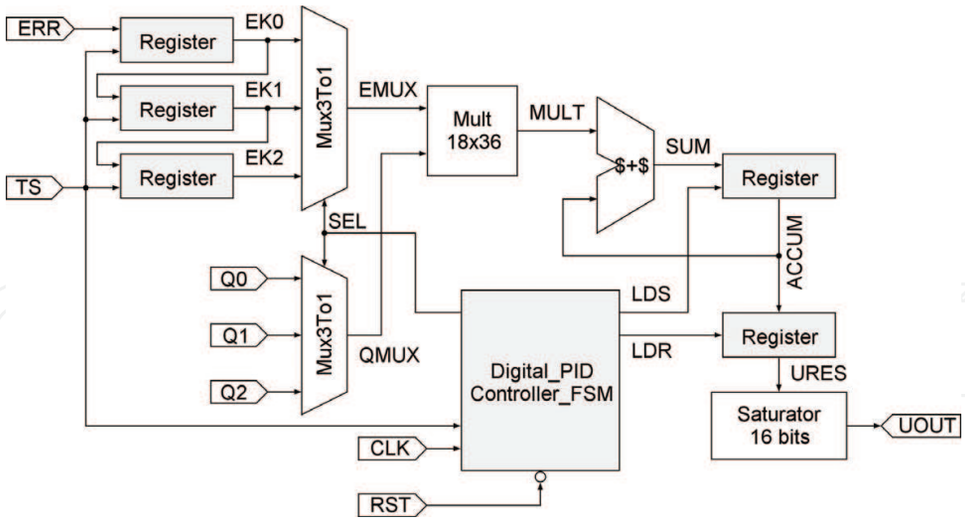

Figure 2. Block diagram of digital PID controller.

the next terms. At the final stage, a saturator trims the bus width of the controller output and saturates its value to 16 bits.

Signals EK0, EK1, and EK2 have 16 bits of width; however, at the multiplexor output, their sign is extended two bits, i.e., the EMUX signal has 18 bits of width. Similarly, the signals Q0, Q1, and Q2 have 32 bits, and at the multiplexor output, QMUX, these signals are extended to 36 bits. The product of error signal as per its corresponding coefficient has 54 bits; nevertheless, this signal is extended again in order to avoid a possible overflow because of the recurrent addition with previous results. Thus, given that there are three sums involved in the solution of the control algorithm, the signal MULT is extended three bits more to obtain a bus width of 57 bits finally. Signals ACCUM and URES also have a 57 bits bus width.

The pipelined structure of registers at the top-left corner, depicted in **Figure 2**, is planned to latch the error signals $e(k)$, $e(k-1)$, and $e(k-2)$ when signal TS is asserted. On the other hand, the register located at the top-right corner together with the adder perform the accumulation process through the assertion of signal LDS. And finally, the last register in the data path serves only as a holder for the final result of the algorithm. This latter register loads data when LDR is asserted.

Figure 3 illustrates the operation of the Digital_PID_Controller_FSM; it includes five states. The first state is an idle state, which waits for the assertion of the sampling signal TS. Second, third, and fourth states perform the multiplication and accumulation of the filter terms, and such partial results are added to the previous final result. The fifth state only asserts the signal LDR to latch the final result and jumps directly to the first state in order to repeat the whole process.

The following source code corresponds to the top-level entity of the design. Its architecture is structural since it is composed of many other components as mentioned above, in total there are 12 instances and 3 concurrent assignations.

Code 1. Digital_PID_Controller.vhd.

```
Library IEEE;

use IEEE.std_logic_1164.all;
```

```
Entity Digital_PID_Filter is
   port(
   RST : in std_logic;
   CLK : in std_logic;
   TS : in std_logic;
   ERR: in std_logic_vector(15 downto 0);
   Q0 : in std_logic_vector(31 downto 0);
   Q1 : in std_logic_vector(31 downto 0);
   Q2 : in std_logic_vector(31 downto 0);
   UOUT : out std_logic_vector(15 downto 0)
   );
end Digital_PID_Controller;
Architecture Structural of Digital_PID_Controller is
--Components declaration-----------------------------------------------------
Component Digital_PID_Controller_FSM is port(
   RST : in std_logic;
   CLK : in std_logic;
   TS : in std_logic;
   LDS : out std_logic;
   LDR : out std_logic;
   SEL : out std_logic_vector(1 downto 0));
end Component;
Component LoadRegister is generic(n : integer := 8);
port(
RST : in std_logic;
CLK : in std_logic;
LDR : in std_logic;
DIN : in std_logic_vector(n − 1 downto 0);
DOUT : out std_logic_vector(n − 1 downto 0));
end Component;
```

http://dx.doi.org/10.5772/67200

```
Component Multiplexor3To1 is generic(n : integer := 8);
   port(
   DIN0 : in std_logic_vector(n - 1 downto 0);
   DIN1 : in std_logic_vector(n - 1 downto 0);
   DIN2 : in std_logic_vector(n - 1 downto 0);
   SEL : in std_logic_vector(1 downto 0);
   DOUT : out std_logic_vector(n - 1 downto 0));
end Component;
Component Multiplier is generic(m, n : integer := 9);
   port(
   OPA : in std_logic_vector(m - 1 downto 0);
   OPB : in std_logic_vector(n - 1 downto 0);
   RES : out std_logic_vector((m + n - 1) downto 0));
   end Component;
Component Adder is generic(n : integer := 8);
   port(
   OPA : in std_logic_vector(n - 1 downto 0);
   OPB : in std_logic_vector(n - 1 downto 0);
   RES : out std_logic_vector(n - 1 downto 0));
end Component;
Component Saturator57To16 is port(
   DIN : in std_logic_vector(56 downto 0);
   DOUT : out std_logic_vector(15 downto 0));
end Component;
--Signals declaration-----------------------------------------------------
signal LDS : std_logic;
signal LDR : std_logic;
signal SEL : std_logic_vector(1 downto 0);
signal EK0 : std_logic_vector(15 downto 0);
signal EK1 : std_logic_vector(15 downto 0);
```

```
signal EK2 : std_logic_vector(15 downto 0);
signal EAUX : std_logic_vector(15 downto 0);
signal EMUX : std_logic_vector(17 downto 0);
signal QAUX : std_logic_vector(31 downto 0);
signal QMUX : std_logic_vector(35 downto 0);
signal MULT : std_logic_vector(53 downto 0);
signal MULE : std_logic_vector(56 downto 0);
signal USUM : std_logic_vector(56 downto 0);
signal ACCU : std_logic_vector(56 downto 0);
signal URES : std_logic_vector(56 downto 0);
begin
--Concurrent assignations-----------------------------------------------
   EMUX <= EAUX(15) & EAUX(15) & EAUX;
   QMUX <= QAUX(31) & QAUX(31) & QAUX(31) & QAUX(31) & QAUX;
   MULE <= MULT(53) & MULT(53) & MULT(53) & MULT;
--Component instances--------------------------------------------------
   U01 : Digital_PID_Controller_FSM port map(RST, CLK, TS, LDS, LDR, SEL);
   U02 : LoadRegister generic map(16) port map(RST, CLK, TS, ERR, EK0);
   U03 : LoadRegister generic map(16) port map(RST, CLK, TS, EK0, EK1);
   U04 : LoadRegister generic map(16) port map(RST, CLK, TS, EK1, EK2);
   U05 : Multiplexor3To1 generic map(16) port map(EK0, EK1, EK2, SEL, EAUX);
   U06 : Multiplexor3To1 generic map(32) port map(Q0, Q1, Q2, SEL, QAUX);
   U07 : Multiplier generic map(18, 36) port map(EMUX, QMUX, MULT);
   U08 : Adder generic map(57) port map(MULE, ACCU, USUM);
   U09 : LoadRegister generic map(57) port map(RST, CLK, LDS, USUM, ACCU);
   U10 : LoadRegister generic map(57) port map(RST, CLK, LDR, ACCU, URES);
   U11 : Saturator57To16 port map(URES, UOUT);
-------------------------------------------------------------------------
end Structural;
```

http://dx.doi.org/10.5772/67200

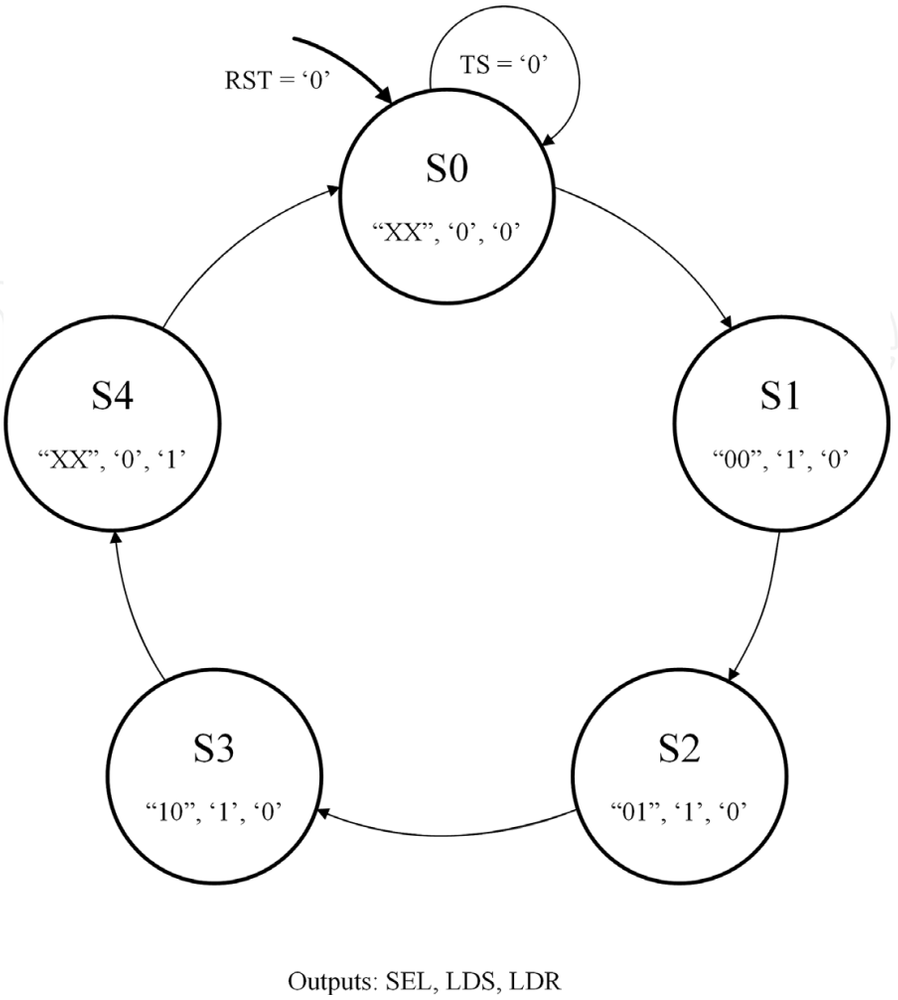

Figure 3. Finite state machine of the digital PID filter.

The following code corresponds to the implementation of the FSM depicted in **Figure 3**. It has a behavioral architecture since it is composed by a couple of processes; the first one includes the combinational logic, which performs the state transitions and sets the output logic. The second process emulates the behavior of a D-type flip-flop, which updates the data with each rising-edge of the master clock signal.

Code 2. Digital_PID_Controller_FSM.vhd.

```
Library IEEE;

use IEEE.std_logic_1164.all;

Entity Digital_PID_Controller_FSM is

   port(

   RST : in std_logic;

   CLK : in std_logic;
```

```
    TS : in std_logic;
    LDS : out std_logic;
    LDR : out std_logic;
    SEL : out std_logic_vector(1 downto 0)
    );
end Digital_PID_Controller_FSM;
Architecture Behavioral of Digital_PID_Controller_FSM is
signal Sp, Sn : std_logic_vector(2 downto 0);
begin
    combinational : process(Sp, TS)
    begin
case Sp is
when “000” =>
LDS <= '0';
LDR <= '0';
SEL <= “XX”;
if TS = '1' then
Sn <= “001”;
else
Sn <= Sp;
end if;
when “001” =>
LDS <= '1';
LDR <= '0';
SEL <= “00”;
Sn <= “010”;
when “010” =>
LDS <= '1';
LDR <= '0';
```

http://dx.doi.org/10.5772/67200

```
SEL <= “01”;
Sn <= “011”;
when “011” =>
LDS <= '1';
LDR <= '0';
SEL <= “10”;
Sn <= “100”;
when others =>
LDS <= '0';
LDR <= '1';
SEL <= “XX”;
Sn <= “000”;
end case;
   end process Combinational;
   Sequential : process(RST, CLK)
   begin
if RST = ‘0’ then
Sp <= “000”;
elsif CLK'event and CLK = '1' then
Sp <= Sn;
end if;
   end process Sequential;
end Behavioral;
```

The LoadRegister module is labeled as Register in **Figure 2**. The objective of this module is to store some particular value. While LDR is asserted, this module stores the value present at the input DIN, and the result is reflected in the output until the next clock event. When LDR is low, the register preserves the last latched value.

Code 3. LoadRegister.vhd.

```
Library IEEE;
use IEEE.std_logic_1164.all;
```

```
Entity LoadRegister is
generic(n : integer := 8);
port(
RST : in std_logic;
CLK : in std_logic;
LDR : in std_logic;
DIN : in std_logic_vector(n − 1 downto 0);
DOUT : out std_logic_vector(n − 1 downto 0)
);
end LoadRegister;
Architecture Behavioral of LoadRegister is
signal Qp, Qn : std_logic_vector(n − 1 downto 0);
begin
Combinational : process(Qp, LDR, DIN)
begin
if LDR = ‘1’ then
Qn <= DIN;
else
Qn <= Qp;
end if;
DOUT <= Qp;
end process Combinational;
Sequential : process(RST, CLK)
begin
if RST = ‘0’ then
Qp <= (others => '0');
elsif CLK'event and CLK = '1' then
Qp <= Qn;
end if;
```

http://dx.doi.org/10.5772/67200

```
end process Sequential;

end Behavioral;
```

The multiplexor in the following code allows directing each sample of the error signal with their corresponding coefficient.

Code 4. Multiplexor3To1.vhd.

```
Library IEEE;

use IEEE.std_logic_1164.all;

Entity Multiplexor3To1 is

generic(n : integer := 8);

port(

DIN0 : in std_logic_vector(n - 1 downto 0);

DIN1 : in std_logic_vector(n - 1 downto 0);

DIN2 : in std_logic_vector(n - 1 downto 0);

SEL : in std_logic_vector(1 downto 0);

DOUT : out std_logic_vector(n - 1 downto 0)

);

end Multiplexor3To1;

Architecture DataFlow of Multiplexor3To1 is

begin

With SEL Select DOUT <=

DIN0 when “00”, DIN1 when “01”, DIN2 when “10”, (others =>'0') when others;

end DataFlow;
```

The following module performs an arithmetic sum between two vectors. It is worth to note that this module does not depend on the clock.

Code 5. Adder.vhd.

```
Library IEEE;

use IEEE.std_logic_1164.all;

use IEEE.std_logic_unsigned.all;

Entity Adder is

generic(n : integer := 8);
```

```
port(
OPA : in std_logic_vector(n − 1 downto 0);
OPB : in std_logic_vector(n − 1 downto 0);
RES : out std_logic_vector(n − 1 downto 0)
);
end Adder;
Architecture DataFlow of Adder is
begin
RES <= OPA + OPB;
end DataFlow;
```

Similarly, the multiplier performs an arithmetic product between two vectors; however, it is important to preserve the sign of the result; therefore, it is included in the IEEE.std_logic_arith library.

Code 6. Multiplier.vhd.

```
Library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
Entity Multiplier is
generic(m, n : integer := 9);
port(
OPA : in std_logic_vector(m − 1 downto 0);
OPB : in std_logic_vector(n − 1 downto 0);
RES : out std_logic_vector((m + n − 1) downto 0)
);
end Multiplier;
Architecture DataFlow of Multiplier is
begin
RES <= signed(OPA) * signed(OPB);
end DataFlow;
```

http://dx.doi.org/10.5772/67200

The last module is provided in the Code 7, which is used to limit the output. For this particular case, the controller output has been adjusted to 16 bits in order to convert this value to an analogue signal using a DAC. Since the controller gains are given in a fixed point format 16.16, the less significant bits of the controller output are trimmed, i.e., only the integer part of the control output is considered during the digital to analogue conversion.

Code 7. Saturator57To16.vhd.

```
Library IEEE;

use IEEE.std_logic_1164.all;

use IEEE.std_logic_arith.all;

Entity Saturator57To16 is

port(

DIN : in std_logic_vector(56 downto 0);

DOUT : out std_logic_vector(15 downto 0)

);

end Saturator57To16;

Architecture Behavioral of Saturator57To16 is

constant UMAX : std_logic_vector(56 downto 0) := '0' & X"0000007FFF0000";

constant UMIN : std_logic_vector(56 downto 0) := '1' & X"FFFFFF80010000";

begin

process(DIN)

begin

if signed(DIN) > signed(UMAX) then

DOUT <= UMAX(31 downto 16);

elsif signed(DIN) < signed(UMIN) then

DOUT <= UMIN(31 downto 16);

else

DOUT <= DIN(31 downto 16);

end if;

end process;

end Behavioral;
```

4. Simulation and experimental results

In this section, the designed PID controller is tested using the software and then experimental studies are carried out for a motion control application of a DC brushed servo motor. The software simulation was performed using Matlab Simulink and ModelSim. Both software applications can run with shared memory in order to perform the cosimulation process.

The first study consists of two control loops as illustrated in **Figure 4**. Both control loops have the same input and plant to be controlled, but the first one is implemented using a Simulink PID block (software implementation), whereas the second one is obtained using the VHDL implementation as described in the previous section (hardware implementation). The aim of this study is to compare the performance between the software- and hardware-based implementations.

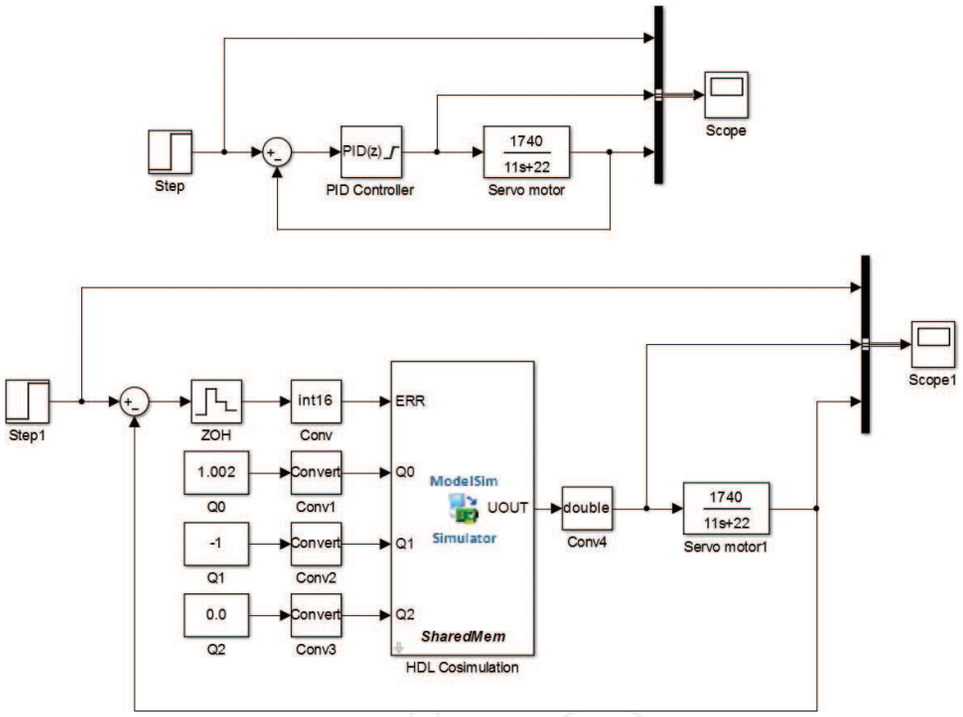

Figure 4. Simulink model for the test of the PID controller.

Before proceeding with the cosimulation, it is important to remark some differences between both PID implementations. The software implementation utilizes a floating point data type with double precision; also, it includes filtering algorithms to compute the derivative and integral term. On the other hand, the hardware implementation described above utilizes the backward rectangular method (BRM) to compute the integral and a simple two-point differentiation for the derivative term; furthermore, the data type utilized has a fixed point format.

The response of the tested control loops is depicted in **Figure 5**. There are three aspects to be considered from the output response. The first one is that the starting angle of the response for the case of the software implementation is higher than the hardware implementation, that is, the response of the software implementation control loop is faster. Second, the first plant reaches the setpoint, relatively faster than the second one, which proves the first assumption.

http://dx.doi.org/10.5772/67200

Finally, the control output of the second loop (hardware implementation) is noisy when compared with the software implementation.

Despite of the aforementioned difference, the performance of the hardware implementation could be acceptable for the control applications in industrial environments. Such kind of an application is described below.

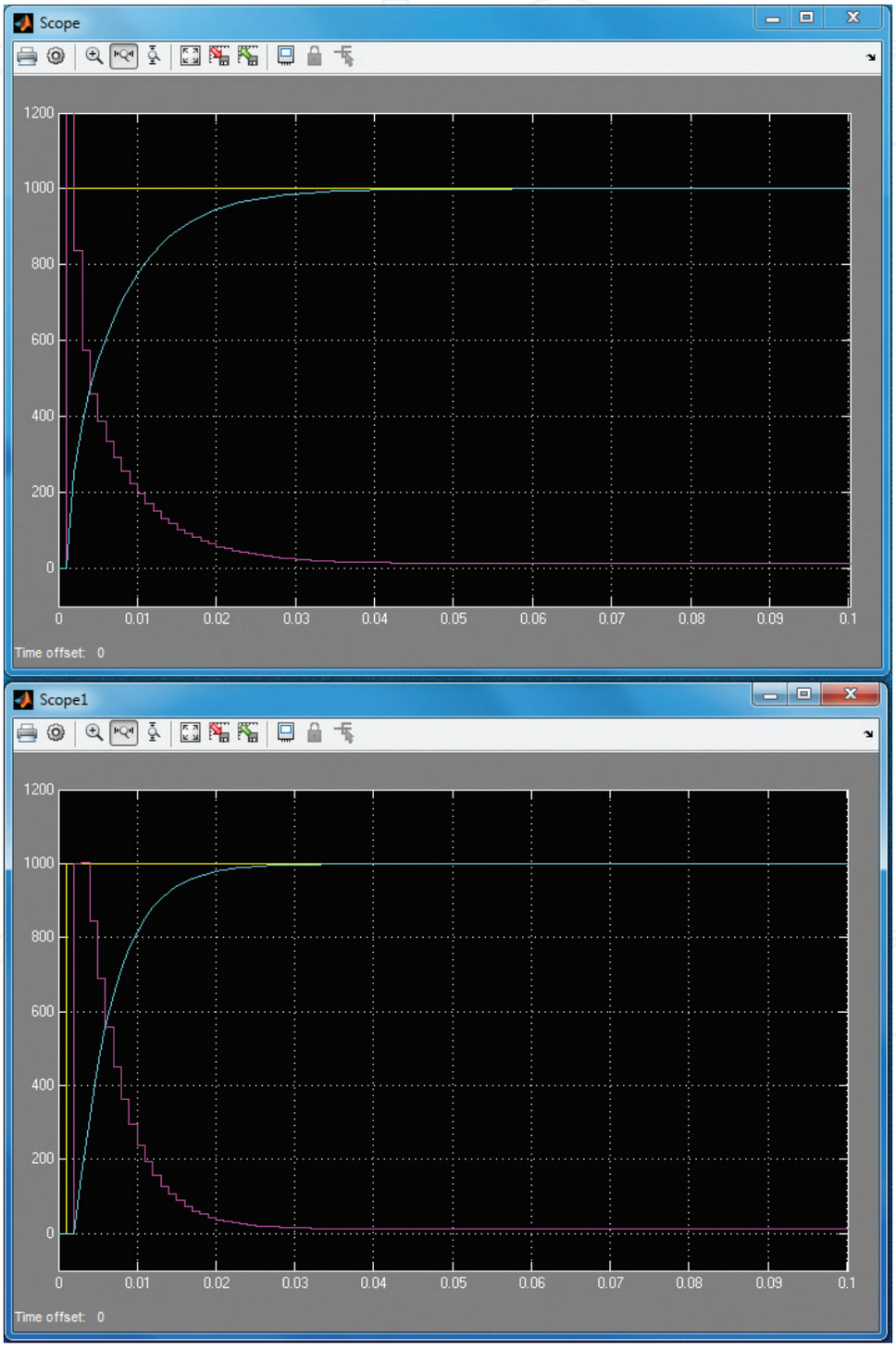

Figure 5. Simulation result of both PID implementations: Simulink (top) and VHDL (bottom).

The application was designed to perform the motion control of a three-degree of freedom robotic arm as illustrated in **Figure 6**. This robot is actuated by 12 V brushed DC motors combined with a 171.79:1 metal spur-gearbox and it has an integrated 48 counts per revolution (CPR) quadrature encoder on the motor shaft, which provides 8245.92 counts per revolution.

A power amplifier stage has been included, which consists of three Texas Instruments LMD18245 power amplifiers required to drive and control the current for the servo motors. These amplifiers can operate with an analogue current reference. For this reason, a digital to analogue converter (DAC) is required. For this application, the Analog Devices AD5668 is utilized, which has eight analogue outputs with a resolution of 16 bits.

The Servo_Controller module is mainly composed of a point of sum, a Digital_PID_Controller, a DAC-Driver, and an Encoder_Quadrature_Interface as shown in **Figure 7**. The reference position (REF) signal is treated as a 32-bit register whose value can be written directly from the PS. On the other hand, the motor position is measured using the Encoder_Quadrature_Interface module. This module uses the signals CHA and CHB from the encoder as the input, and it generates an output signal POS which has a width of 32 bits. Both signals REF and POS

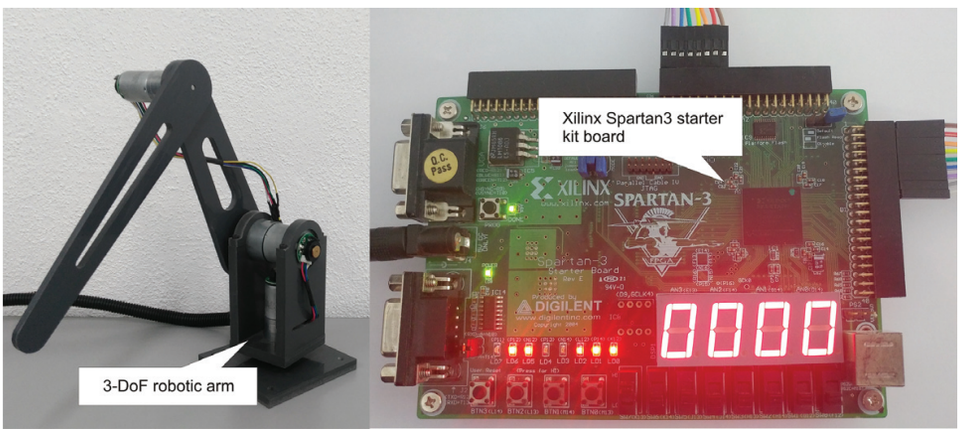

Figure 6. Experimental setup.

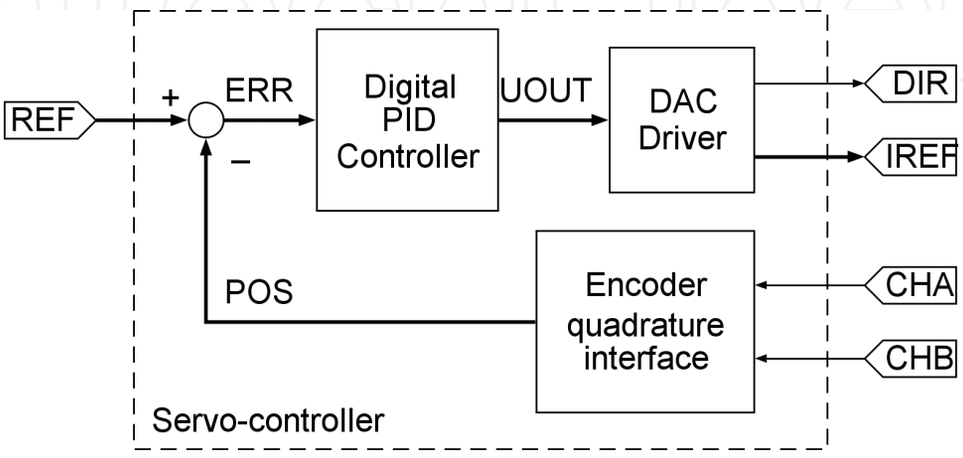

Figure 7. Servo controller block diagram.

http://dx.doi.org/10.5772/67200

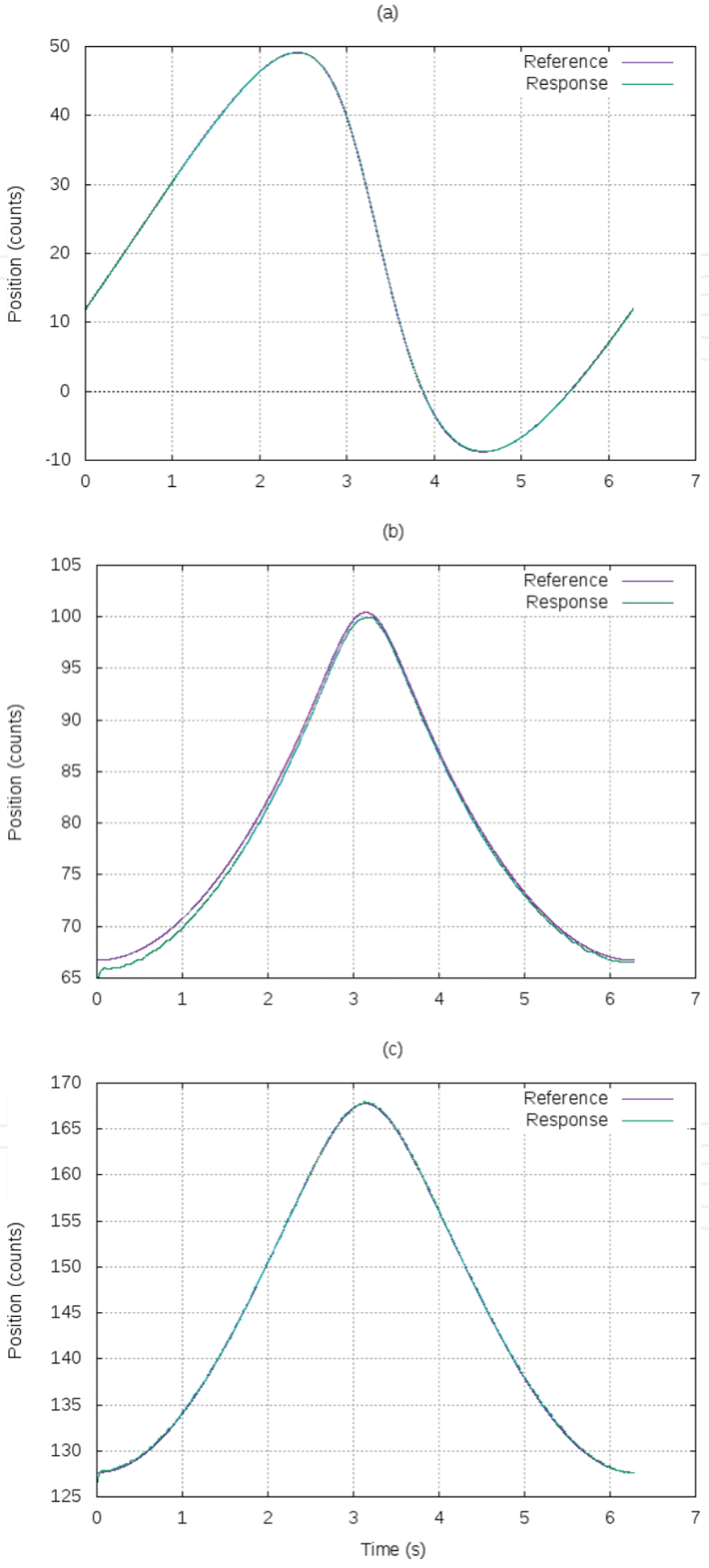

Figure 8. Response of the system to the trajectory commanded: (a) joint 1, (b) joint 2, and (c) joint 3.

are subtracted and trimmed to 16 bits to avoid any saturation in further computations. In **Figure 7**, this signal is labeled as ERR, which serves as the input to the Digital_PID_Controller.

A circular trajectory is consideredin order to evaluate the controller performance. The center of the circumference with a radius of 50 mm is located at (110.0, 0.0, 70.0) being the Z-coordinate constant through the whole movement, i.e., the entire movement is carried out only in the X-Y plane. First, the interpolation process generates each point along the circumference. This point is used to solve the inverse kinematics. Thus, the resulting angle set is converted to encoder counts and written to the setpoint registers of each servo controller. The sampling time for the generation of each point is chosen as $T_s = 0.001$ s.

Figure 8 shows the response of each servo controller to the generated path. It can be clearly seen that the first and the third joints closely follow the reference trajectory, whereas the second joint shows a larger variation from the commanded trajectory. This could be due to the influence of nonlinear dynamics of the servo motor or due to the gravity force. However, the Cartesian position of the robot remains very close to the commanded trajectory as can be seen in **Figure 9**.

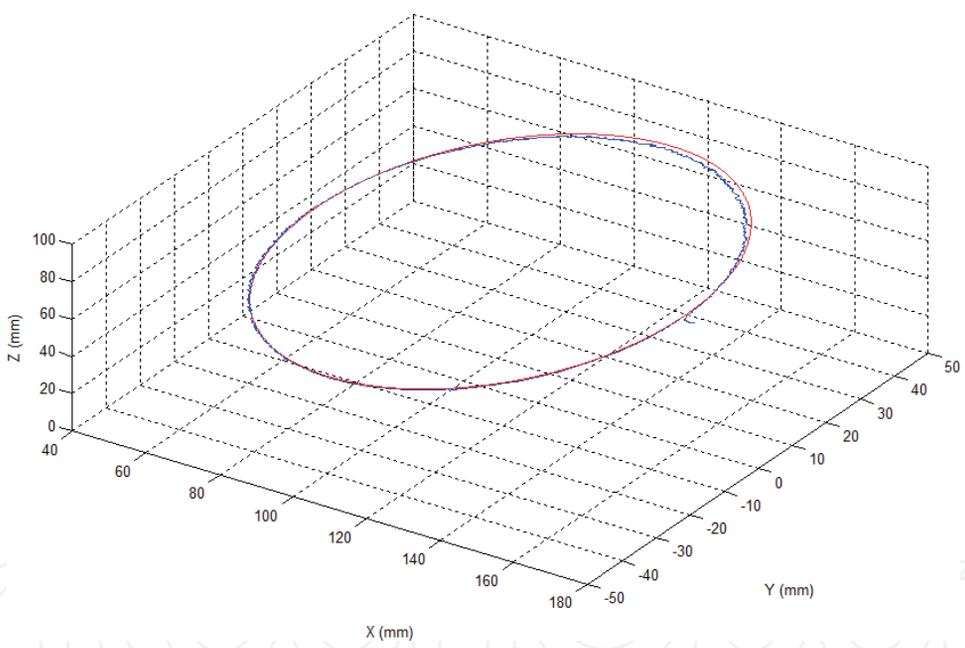

Figure 9. Cartesian response of the system.

5. Conclusions

The endeavor of this chapter is how to deal with theoretical and practical issues regarding the control systems. The readers who are not familiar with motion control systems can get a basic knowledge into reconfigurable logic circuit-based digital design. Generally, it is necessary to combine classroom and practical concepts; for that reasons, these sections are not only aimed for the students or professors but also for professionals who want to obtain a basic understanding about the closed-loop control design.

For educators, many concepts can be applied in courses as servo systems, programming, classical control, and digital control, to mention a few. Since FPGA technology is almost available in all engineering schools, there is no restriction to apply the code shown in this manuscript. In addition, sequential devices might be used too. It is due to the facility to translate HDL code to C code. In this sense, microcontrollers, digital signal processors, and digital signal controllers get a good approach to make a motion control task.

It is recommended to use standard compilers and hardware tools that do not demand high computational resources. It is because of the synthesis stage, which is often the hard part of the developing work.

Author details

Miguel Angel Martínez Prado, Juvenal Rodríguez Reséndiz*, Diana Carolina Toledo Pérez, Carlos Miguel Torres Hernández and Gilberto Herrera Ruiz

*Address all correspondence to: juvenal@uaq.edu.mx

Facultad de Ingeniería, Universidad Autónoma de Querétaro, Cerro de las Campanas SN, Col. Las Campanas, Querétaro, Qro, México

References

[1] Monmasson E, Cirstea M N. FPGA design methodology for industrial control systems - A review. IEEE Transactions on Industrial Electronics. 2007;**54**:1824–1842. DOI: 10.1109/TIE.2007.898281

[2] Zhao W, Kim B H, Larson A C, Voyles R M. FPGA implementation of closed-loop control system for small-scale robot. In: ICAR '05. Proceedings, 12th International Conference on Advanced Robotics, 2005; IEEE Xplore; July 2005;. pp. 70–77. DOI: 10.1109/ICAR.2005.1507393

[3] Ghosh S, Barai R K, Bhattarcharya S, Bhattarcharya P, Rudra S, Dutta A, Pyne R. An FPGA based implementation of a flexible digital PID controller for a motion control system. In: 2013 International Conference on Computer Communication and Informatics (ICCCI); 04 Jan–06 Jan 2013; Coimbatore, Tamil Nadu, India. IEEE Xplore; 2013; pp. 1–6. DOI: 10.1109/ICCCI.2013.6466277

[4] Xu Y, Zhao J, Huang J. Multiple linear motor control system based on FPGA. In: 2014 17th International Conference on Electrical Machines and Systems (ICEMS); 22 Oct–25 Oct 2014; Hangzhou, China. IEEE Xplore; 2014. pp. 2327–2331. DOI: 10.1109/ICEMS.2014.7013875

[5] Nandayapa M, Mitsantisuk C, Ohishi K. Improving bilateral control feedback by using novel velocity and acceleration estimation methods in FPGA. In: 2012 12th IEEE

International Workshop on Advanced Motion Control (AMC); 25 Mar–27 Mar 2012; Sarajevo, Bosnia and Herzegovina: IEEE Xplore; 2012. pp. 1–6. DOI: 10.1109/AMC.2012.6197024

[6] Bagni D, Mackay D. Floating-point PID controller design with Vivado HLS and system generator for DSP [Internet]. 2013. Available from: http://www.xilinx.com/support/documentation/application_notes/xapp1163.pdf [Accessed: 2016-02-25]

[7] Aboelaze M, Shehata MG. Implementation of multiple PID controllers on FPGA. In: 2015 IEEE International Conference on Electronics, Circuits, and Systems (ICECS); 7–9 Dec 2015; Egypt. IEEE Xplore; 2015. pp. 446–449. DOI: 10.1109/ICECS.2015.7440344

[8] Uzunović T, Žunić E, Badnjević A, Mioković I, Konjicija S. Implementation of digital PID controller. In: 2010 Proceedings of the 33rd International Convention MIPRO; 24–28 May 2010; Opatija, Croatia. IEEE Xplore; 2010. pp. 1357–1361.