10.3 Case Study 1: DC Motor Control

10.3.1 Introduction

In this case study, a digital controller is developed to control the speed of a DC electric motor. The overall control system model will be developed in MATLAB® [9] and its Simulink® [10] toolbox. The model of the control algorithm will then be manually converted to VHDL code using a set design translation flow for implementation as a digital controller using a CPLD. The design issues will be captured and presented in a way that allows the VHDL code to be generated automatically. The overall design flow is shown in Figure 10.7.

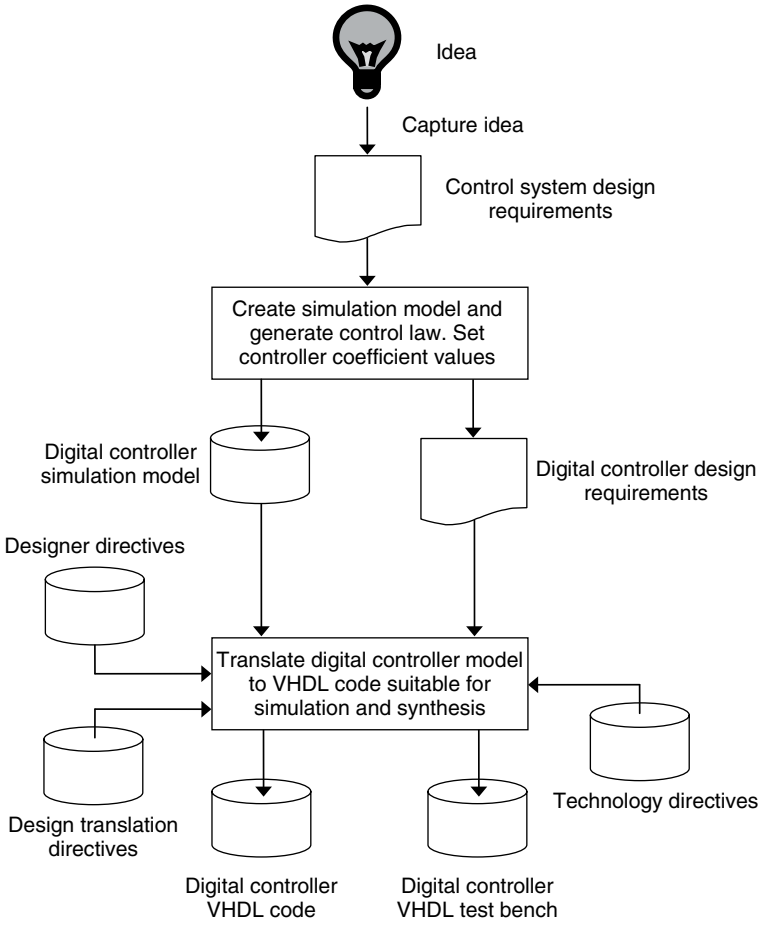

Figure 10.7: Motor control case study design flow

10.3.2 Motor Control System Overview

The control system is a closed-loop controller using PI (proportional plus integral) control [11, 12]. Other forms of control algorithm such as PID (proportional plus integral plus derivative) could be used, but the added complexity is unnecessary in this case. The particular control algorithm was chosen based on the requirements of the motor (the plant to control) and the required system response. As such, PI control provides zero steady-sate error in the motor speed (a motor speed steady-state error would exist if only proportional control was used) and a design simple to implement and easy to understand. The coefficients of each action within the PI control law are set to give a response that settles to a steady state in an adequately short time. The initial step in the design is to create the control system block diagram, shown in Figure 10.8.

The controller receives two analogue signals (voltages): first the command input that sets the required motor speed, then a feedback input that identifies the actual speed of the motor. In this control system model, then:

- The motor is modeled as a Laplace transform with the transfer function [1/(1 + 0.1s)].
- The analogue input range for the controller is 0 V to + 5.0 V, which indicates a speed in both directions of motor shaft rotation, where:
  - 0 V indicates a maximum motor shaft speed in an anticlockwise direction.
  - + 5.0 V indicates a maximum motor shaft speed in a clockwise direction.
  - + 2.5 V indicates that the motor shaft is stationary.

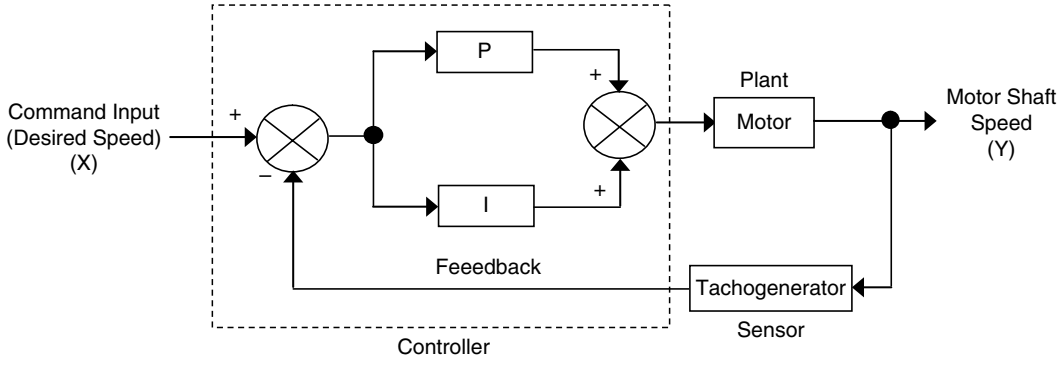

Figure 10.8: Motor control system example with PI control

- The proportional action (Kp) gain is +2.0, and the integral action gain (Ki) is +8.0 (not optimized).
- This is a high-level behavioral model (and a linear model of the system) that does not take into account nonlinear effects such as value limits, slew rate effects, and any existing motor dead zone.
- The motor model contains a tachogenerator (sensor) that produces an analogue voltage output in the range 0 V to +5.0 V.
- The command input (required speed) and actual motor speed outputs are considered to be voltages, and the motor shaft speed uses suitable units (e.g., rads/sec).
- The model uses the built-in Simulink® library continuous time blocks, and no design hierarchy has been developed.
- The digital controller is required to sample analogue signals and to undertake digital signal processing on the discrete time samples. The sampling frequency for this design is 100 Hz, a slow sampling frequency compared to many control systems, but adequate for this application.
- The model uses only continuous time blocks, so when the digital controller is created, the analogue model prototype must be converted to a digital approximation. Those parts of the controller to be mapped to a digital algorithm modeled in VHDL must therefore be identified.

The motor model used is a simple first-order Laplace transform that models the motor and tachogenerator as a single unit. This was created by monitoring the tachogenerator output voltage to a step change in motor speed command input voltage. This is reasonably representative of the motor reaction to larger step changes in command input, but does not model nonideal characteristics such as a motor dead zone around a null (zero) command input and the need to minimize the command input voltage required for the motor to react to a command input change.

A full analysis of the control system is undertaken to determine that the derived control algorithm is suitable for the application. This analysis is not, however, covered in this text.

At this level of design abstraction (i.e., a simplified model of the system), none of the implementation issues have been considered and only a mathematical model of the system exists. But of course, ultimately, the system must be built using electronic circuits. The basic arrangement created for such a control system is shown in Figure 10.9. Here, the CPLD implements the digital control algorithm and interfaces

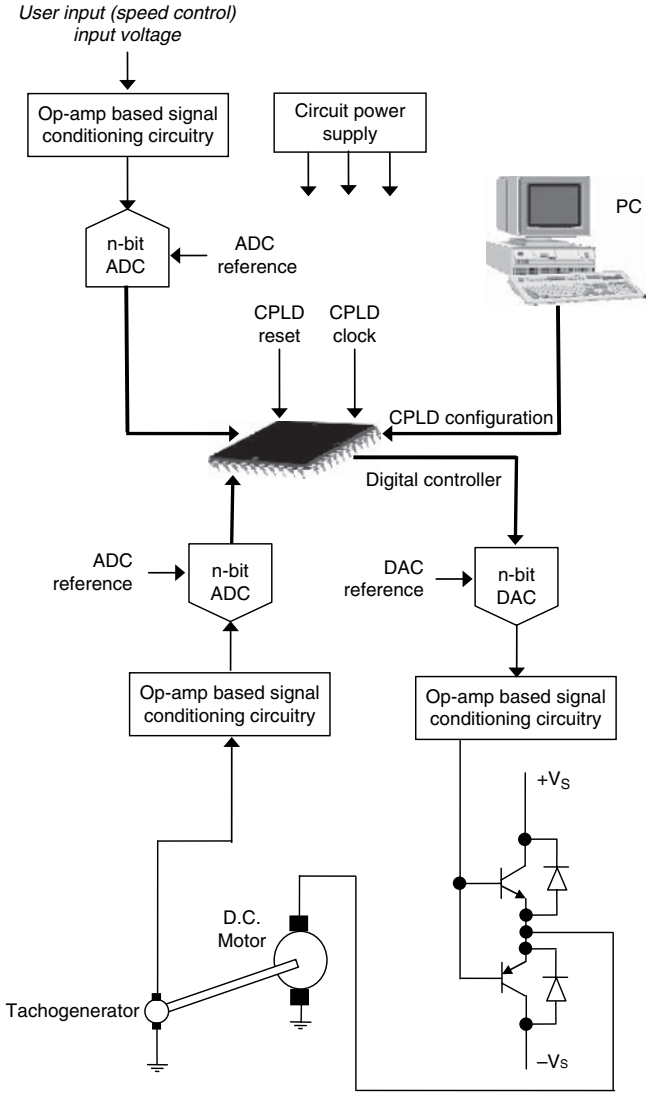

Figure 10.9: Motor speed control circuit arrangement

to two ADCs (analogue-to-digital converters, to sample the analogue input voltages for the command input and the feedback) and one DAC (digital-to-analogue converter, to output an analogue voltage to create the motor voltage). This DAC output voltage is applied to a transistor power amplifier (because the DAC would not be able to provide the necessary voltage and current levels required by the DC motor). Op-amp based analogue circuitry is used on the ADC inputs and DAC outputs as necessary to provide specific low-power analogue signal conditioning. A power supply unit provides the necessary voltage and current levels required by the overall circuit. Finally, a PC is used here to configure the CPLD.

10.3.3 MATLAB®/Simulink® Model Creation and Simulation

Before considering how controller is to be implemented, the control law (algorithm) must be developed and analyzed. An example Simulink® model for this system is shown in Figure 10.10.

The controller is placed within a single block (the controller block), and the motor (motor model) is modeled using as a first-order system a Laplace transform equation. The motor model also contains the tachogenerator output, so the output from the

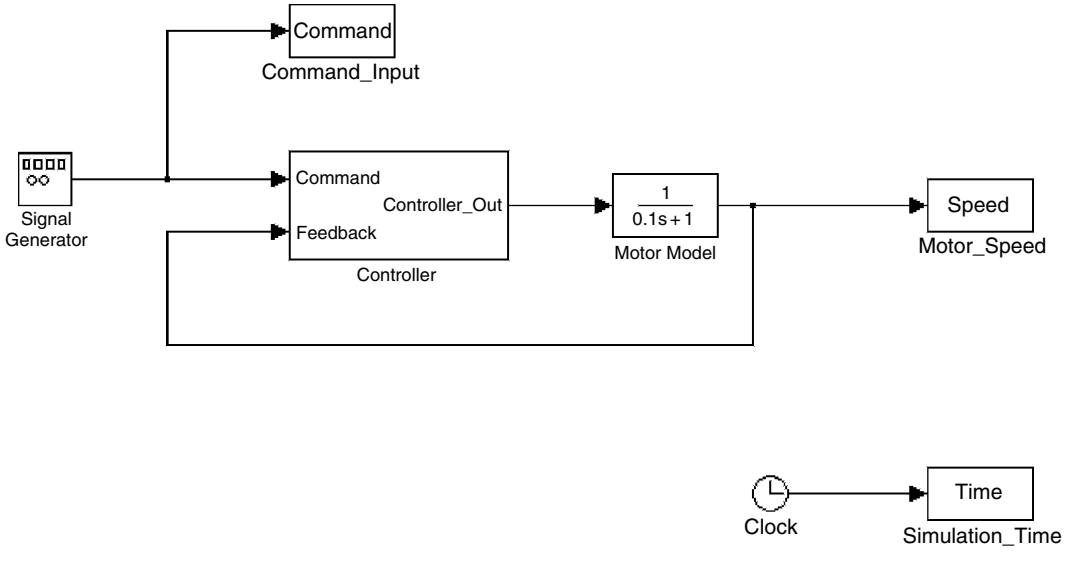

Figure 10.10: Simulink® model for the motor control system case study

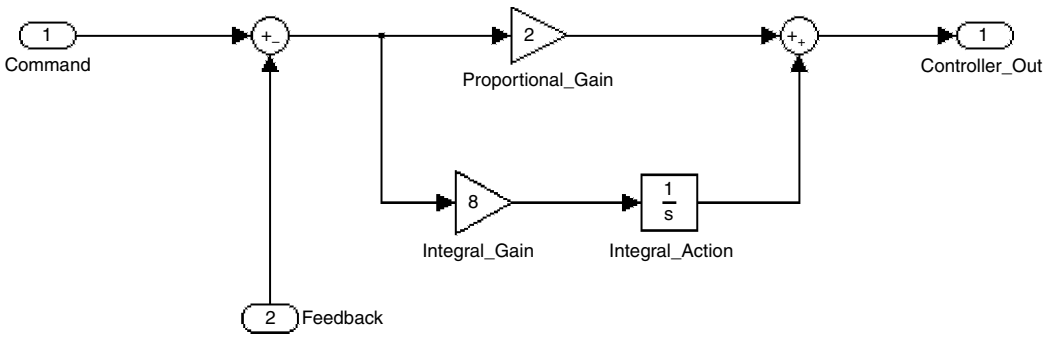

Figure 10.11: Simulink® model for the PI controller

system is modeled as the tachogenerator voltage (which represents the motor shaft speed). This equation was obtained from the motor itself by applying directly a step input voltage to the motor and observing the tachogenerator voltage. A signal generator (signal generator block) allows different signals to be applied to the system.

The design is analyzed using both hand calculations and the Simulink® simulator, with typical analogue input signals (step, sine wave, DC, triangle, and ramp) as part of the overall system analysis routine. A frequency response could be undertaken by generating a model for frequency analysis in MATLAB®.

The PI controller is shown in Figure 10.11.

The control system is simulated, and the gain values for the proportional and integral actions are set so that the required response is obtained: a stable system with a transient response that matches the requirements of the design specification. For a proportional gain of +2.0 and an integral gain of +8.0 (not optimized), the system response (i.e., motor shaft speed) produces an overdamped response to a step input as shown in Figure 10.12.

10.3.4 Translating the Design to VHDL

After the analysis of the system has been completed, the digital controller model is translated to VHDL code suitable for simulation and synthesis. This requires that the VHDL code be generated according to a set design translation flow in the following eight steps:

1. Translation preparation (according to the nine steps below).
2. Set the architecture details (according to the six steps below).

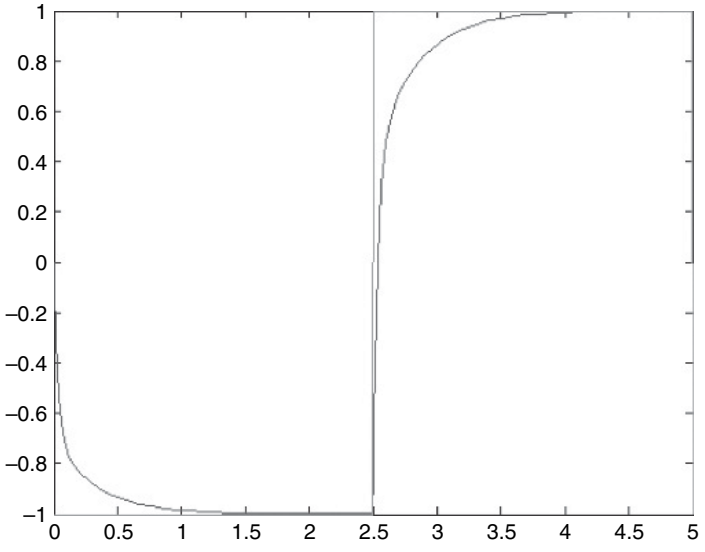

Figure 10.12: Simulation results for step change

3. Translation from Simulink® model to VHDL code by reading the Simulink® model, extracting the necessary design information, and generating the VHDL code.
4. Generate VHDL test bench.
5. Simulate the VHDL code and check for correct operation to validate the operation of the generated VHDL code.
6. Synthesize the VHDL code and resimulate the design to generate a structural design based on the particular target technology.
7. Configure the CPLD and validate the operation of the design.
8. Use the controller.

The nine steps of translation preparation are:

1. Identify the parts to be translated into digital (the controller).
2. Remove any unnecessary information, leaving only the controller model.
3. Identify the digital controller interfacing.

4. Identify the clock and reset inputs, along with any other control signals.
5. Identify any external communications required.
6. Set up the support necessary to include the translation directives (see architecture details below).
7. Identify the technology directives (any requirements for the target technology, such as CPLD) and the synthesis tool to be used.
8. Identify any designer directives.
9. Determine what test circuitry is to be inserted into the design and at what stage in the design process.

The six steps to set the architecture details are:

1. Identify the particular architecture to use.
2. Identify the internal wordlength within the digital signal processing part of the digital core.
3. Identify any specific circuits to avoid (e.g., specific VHDL code constructs).
4. Identify the control signals required by the I/O.
5. Identify the number system to use (e.g., 2s complement) in the arithmetic operations.
6. Identify any number scaling requirements to limit the required wordlength within the design.

The model translation must initially consider the architecture to use either a processor-based architecture running a software application (standard fixed architecture processor or a configurable processor) or a custom hardware architecture based directly on the model. This idea is shown in Figure 10.13.

If the translation is to be performed manually, this can be undertaken by visual reference to the graphical representation of the model (i.e., the block diagram). If the translation is to be performed automatically (by a software application), the translation can be performed using the underlying text based model (i.e., with the Simulink®*.mdl* file).

A fixed architecture processor is based on an existing CISC or RISC architecture, and its translation either will generate the hardware design (in HDL) and the processor

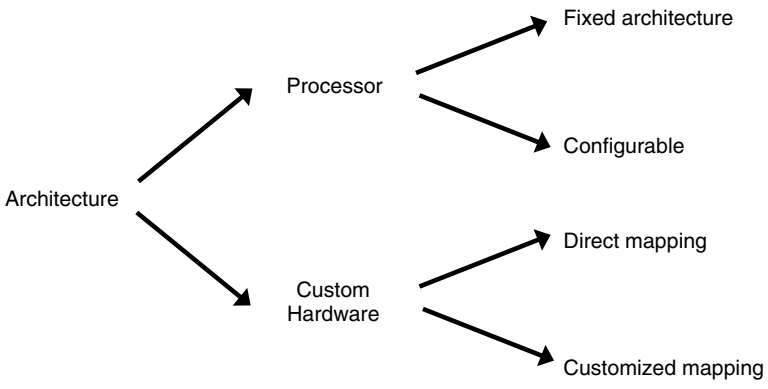

Figure 10.13: Controller architecture decisions

microcode together, or will generate only the processor microcode using an existing processor design. The configurable processor is a processor design that dynamically changes specific aspects of the architecture based on the particular application.

Direct mapping starts with the model as presented and directly translates its functions to a custom hardware HDL code equivalent. Customized mapping uses custom architecture based on the model, but then determines the most appropriate way to implement its functions (e.g., by using multiple multiplication blocks or a single multiplexed multiplier block) based on the application.

No matter what particular architecture is chosen, in addition to generating the required digital signal processing algorithm hardware (as identified in the system block diagram), then there would be the need to also generate the necessary interfacing signals for external circuitry such as ADCs and DACs, and the internal timing signals for the control of the signal processing operations, along with the storage and movement of data signals within the design. These interfacing and internal timing signals would need to be created by an additional circuit creating the functions of a *control unit* particular to the design.

In this case study, direct mapping of model functions will be considered, so the controller shown in Figure 10.11 will be translated. This requires the use of the following main functional blocks:

- one subtraction block
- one addition block

- one proportional action
- one integral action (shown as the integral gain and integrator action blocks)

One complication with this model is that it was created using continuous time blocks as an analogue prototype of the digital controller. The Simulink® model uses Laplace transforms, which much be approximated to a pulse transfer function for discrete time implementation. The pulse transfer function G(z) is created from the Laplace (s) transform form using one of the following methods where T is the signal sampling period:

1. Forward difference or Euler's method:

$$s = \frac{z - 1}{T}$$

2. Backward difference method:

$$s = \frac{z - 1}{zT}$$

3. Tustin's approximation (also referred to as the bilinear transform):

$$s = \frac{2}{T} \cdot \frac{z - 1}{z + 1}$$

These methods are readily applied by hand to transform from s to z.

In this case study, Tustin's approximation is used. It applies only to the integral action since the proportional action is simply a multiplication on the sampled data.
The proportional action (using Z-transforms) is:

$$P(z) = Kp.X(z)$$

The integral action (using Z-transforms) is:

$$I(z) = \left(\left(\frac{KiT}{2}\right)\left(x(z) + x(z)z^{-1}\right)\right) + I(z)z^{-1}$$

The PI controller block diagram can be remodeled using Z-transforms, as shown in Figure 10.14. The two storage (z^{-1}) blocks have a common clock signal that controls when the inputs to the blocks are stored. This control signal must be created. The

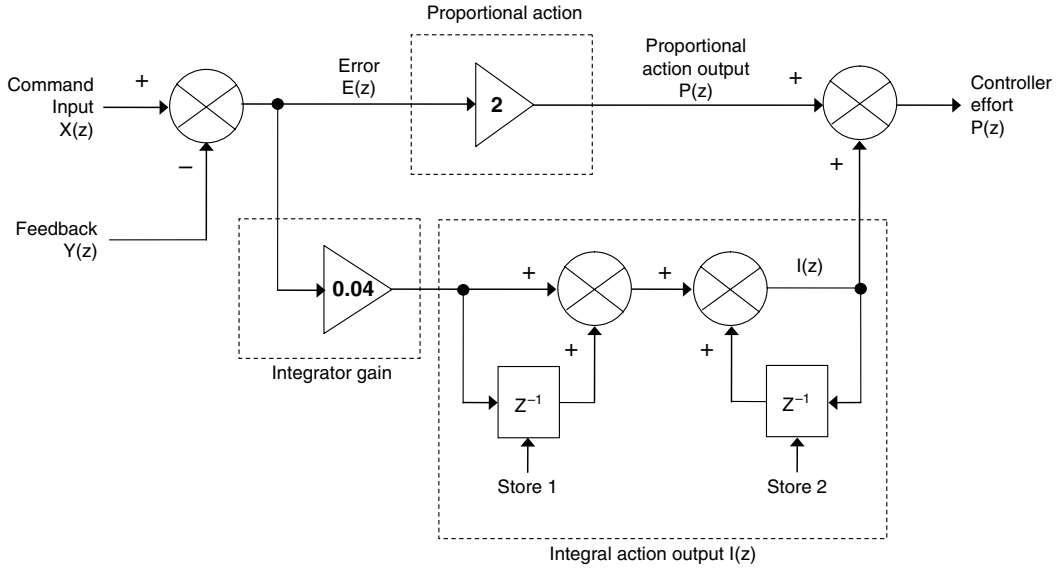

Figure 10.14: Discrete time PI controller

controller block shown in Figure 10.14 forms the digital signal processing core of the overall controller design. Figure 10.15 shows this core along with the necessary control unit that generates the internal control signals based on the timing requirements of the controller. The inputs to the controller are sampled at a sampling frequency of 100 Hz; this timing is generated from a master input clock. After the algorithm has been run on the current input signal (and previous inputs along with previous outputs), the current output is updated. Because these actions are performed in less time than the 100 Hz sampling frequency allows, the design must wait until the next sample is required. This idea is shown in Figure 10.16.

Signed arithmetic is used inside the control algorithm hardware (2s complement in this case study). To achieve this, and given that the input is straight binary, the sampled value must be stored (in a register) and converted to a 2s complement number, as shown in Table 10.3.

Finally, the interconnects between the main functional blocks must be considered. The inputs are analogue inputs sampled using two AD7575 eight-bit LC^2MOS (leadless chip carrier metal oxide semiconductor) successive approximation ADCs [13]. The output is an analogue signal created using a single AD7524 eight-bit buffered multiplying DAC [14]. The internal wordlength is 16 bits, so the eight-bit input and analogue output is transformed from 16-bit input and output. The eight-bit

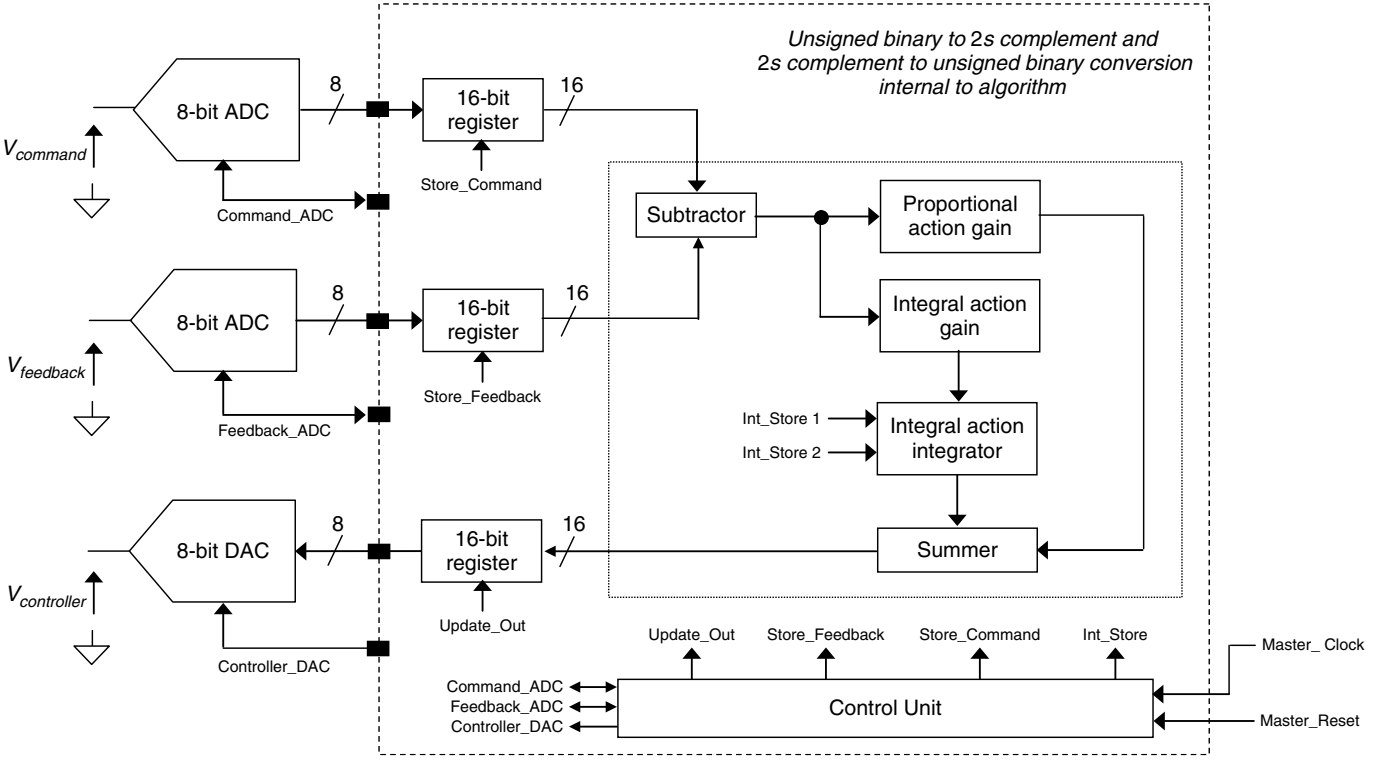

Figure 10.15: Electronic controller circuit block diagram

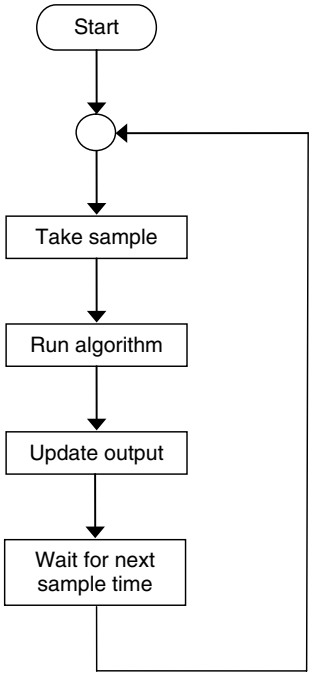

Figure 10.16: PI controller operation flowchart

output circuitry must also include value limiting because the 16-bit internal value exceeds the value limits set by the eight-bit output.

The Simulink® model for the overall control system must be reviewed and should contain:

- information for translation to VHDL
- information *not* for translation to VHDL

Table 10.3: Binary I/O to internal value mapping

Digital I/O code, decimal	Digital I/O code (8-bits), binary		Internal code, decimal	Internal code (8-bits), binary
0	00000000	→	–128	10000000
127	01111111	→	–1	11111111
128	10000000	→	0	00000000
255	11111111	→	+127	01111111

The information *not* for translation to VHDL includes information such as visual attributes and software version information and must be stripped from the representation of the model used for translation to VHDL. The Simulink® model code for the controller only is shown in Figure 10.17. This is the text description of the model shown in Figure 10.11. It consists of the blocks used, their attributes, and the interconnect between the blocks (lines). Interpreting this model requires knowledge of its syntax and how the values that can be modified by the user are represented. The syntax is readable, and the names used are identifiable by comparison with the block diagram view.

This model for the controller can be remodeled in VHDL, shown in Figure 10.18 as a structural description for the control algorithm. Detailed operation of each block is defined in separate entity-architecture pairs.

The Xilinx ISE™ RTL schematic for the synthesized controller design is shown in Figure 10.19.

This control algorithm is placed in the overall VHDL structural description of the controller, as shown in Figure 10.20.

The Xilinx ISE™ RTL schematic for the synthesized controller design is shown in Figure 10.21.

The final step is to generate and simulate a VHDL test bench for the controller. An example VHDL test bench is shown in Figure 10.22.

10.3.5 Concluding Remarks

This case study design was for a simple digital control algorithm, but it also shows the main operations required for typical digital control algorithms. The VHDL code to implement the design within a CPLD was created by mapping the original Simulink® block diagram to a VHDL code equivalent in which each of the main functional blocks was presented as a unique entity-architecture pair. The structural design of the controller top level and the control algorithm were presented, although the details of the individual operations are left for the reader to implement.

The block diagram was mapped directly to VHDL to implement a custom hardware design. In many cases, this would result in a large design, particularly when multiple multiplications are necessary. However here, the ease and rapid development of the

```
System {
    Name                    "Controller"

    Block {
      BlockType             Inport
      Name                  "Command"
      Position              [20, 73, 50, 87]
    }
    Block {
      BlockType             Inport
      Name                  "Feedback"
      Position              [140, 225, 170, 240]
      Orientation           "up"
      Port                  "2"
    }
    Block {
       BlockType            Integrator
      Name                  "Integral_Action"
      Ports                 [1, 1]
      Position              [355, 155, 385, 185]
    }
    Block {
      BlockType             Gain
      Name                  "Integral_Gain"
      Position              [260, 155, 290, 185]
      Gain                  "8"
    }
    Block {
      BlockType             Gain
      Name                  "Proportional_Gain"
      Position              [285, 65, 315, 95]
      Gain                  "2"
    }
    Block {
      BlockType             Sum
      Name                  "Sum"
      Ports                 [2, 1]
      Position              [145, 70, 165, 90]
      ShowName              off
      IconShape                   "round"
      Inputs                "|+-"
      InputSameDT           off
      OutDataTypeMode       "Inherit via internal rule"
    }
    Block {
      BlockType             Sum
      Name                  "Sum1"
      Ports                 [2, 1]
      Position              [430, 70, 450, 90]
      ShowName              off
      IconShape                   "round"
      Inputs                "|++"
      InputSameDT           off
      OutDataTypeMode       "Inherit via internal rule"
    }
```

Figure 10.17: Simulink® model for the PI controller

```
    Block {
      BlockType               Outport
      Name                    "Controller_Out"
      Position                [525, 73, 555, 87]
    }
    Line {
      SrcBlock                "Sum"
      SrcPort                 1
      Points                  [0, 0; 45, 0]
      Branch {
        DstBlock               "Proportional_Gain"
        DstPort                1
      }
      Branch {
        Points                 [0, 90]
        DstBlock               "Integral_Gain"
        DstPort                1
      }
    }
    Line {
      SrcBlock                "Integral_Gain"
      SrcPort                 1
      DstBlock                "Integral_Action"
      DstPort                 1
    }
    Line {
      SrcBlock                "Proportional_Gain"
      SrcPort                 1
      DstBlock                "Sum1"
      DstPort                 1
    }
    Line {
      SrcBlock                "Integral_Action"
      SrcPort                 1
      Points                  [50, 0]
      DstBlock                "Sum1"
      DstPort                 2
    }
    Line {
      SrcBlock                "Command"
      SrcPort                 1
      DstBlock                "Sum"
      DstPort                 1
    }
    Line {
      SrcBlock                "Feedback"
      SrcPort                 1
      DstBlock                "Sum"
      DstPort                 2
    }
    Line {
      SrcBlock                "Sum1"
      SrcPort                 1
      DstBlock                "Controller_Out"
      DstPort                 1
    }
  }
```

Figure 10.17: (Continued)

```
1  LIBRARY ieee;
2  USE ieee.std_logic_1164.all;
3  USE ieee.std_logic_arith.all;
4  USE ieee.std_logic_unsigned.all;
5
6
7  ENTITY Control_Algorithm IS
8   PORT ( Command              : IN   STD_LOGIC_VECTOR (15 downto 0);
9          Feedback             : IN   STD_LOGIC_VECTOR (15 downto 0);
10         Controller_Out       : OUT  STD_LOGIC_VECTOR (15 downto 0);
11         Integrator_Store_1   : IN   STD_LOGIC;
12         Integrator_Store_2   : IN   STD_LOGIC;
13         Reset                : IN   STD_LOGIC);
14 END ENTITY Control_Algorithm;
15
16
17 ARCHITECTURE Structural OF Control_Algorithm IS
18
19
20 SIGNAL   Error          :  STD_LOGIC_VECTOR (15 downto 0);
21 SIGNAL   Proportional   :  STD_LOGIC_VECTOR (15 downto 0);
22 SIGNAL   Int_1          :  STD_LOGIC_VECTOR (15 downto 0);
23 SIGNAL   Int_2          :  STD_LOGIC_VECTOR (15 downto 0);
24 SIGNAL   Int_3          :  STD_LOGIC_VECTOR (15 downto 0);
25 SIGNAL   Int_4          :  STD_LOGIC_VECTOR (15 downto 0);
26 SIGNAL   Integral       :  STD_LOGIC_VECTOR (15 downto 0);
27
28
29 COMPONENT Adder IS
30  PORT ( Data_In_1 : IN   STD_LOGIC_VECTOR (15 downto 0);
31         Data_In_2 : IN   STD_LOGIC_VECTOR (15 downto 0);
32         Data_Out  : OUT  STD_LOGIC_VECTOR (15 downto 0));
33 END COMPONENT Adder;
34
35 COMPONENT Subtractor IS
36  PORT ( Data_In_1 : IN   STD_LOGIC_VECTOR (15 downto 0);
37         Data_In_2 : IN   STD_LOGIC_VECTOR (15 downto 0);
38         Data_Out  : OUT  STD_LOGIC_VECTOR (15 downto 0));
39 END COMPONENT Subtractor;
40
41 COMPONENT Integral_Gain IS
42  PORT ( Data_In  : IN   STD_LOGIC_VECTOR (15 downto 0);
43         Data_Out : OUT  STD_LOGIC_VECTOR (15 downto 0));
44 END COMPONENT Integral_Gain;
45
46 COMPONENT Proportional_Gain IS
47  PORT ( Data_In  : IN   STD_LOGIC_VECTOR (15 downto 0);
48         Data_Out : OUT  STD_LOGIC_VECTOR (15 downto 0));
49 END COMPONENT Proportional_Gain;
50
```

Figure 10.18: VHDL model for the control algorithm

```
51  COMPONENT Delay IS
52   PORT ( Data_In   :  IN   STD_LOGIC_VECTOR (15 downto 0);
53          Data_Out  :  OUT  STD_LOGIC_VECTOR (15 downto 0);
54          Store     :  IN   STD_LOGIC;
55          Reset     :  IN   STD_LOGIC);
56  END COMPONENT Delay;
57
58
59  BEGIN
60
61
62  I1 : Subtractor
63       PORT MAP  ( Data_In_1  =>  Command,
64                   Data_In_2  =>  Feedback,
65                   Data_Out   =>  Error);
66
67  I2 : Proportional_Gain
68       PORT MAP  ( Data_In    =>  Error,
69                   Data_Out   =>  Proportional);
70  I3 : Adder
71       PORT MAP  ( Data_In_1  =>  Proportional,
72                   Data_In_2  =>  Integral,
73                   Data_Out   =>  Controller_Out);
74
75  I4 : Integral_Gain
76       PORT MAP  ( Data_In    =>  Error,
77                   Data_Out   =>  Int_1);
78  I5 : Delay
79       PORT MAP  ( Data_In    =>  Int_1,
80                   Data_Out   =>  Int_2,
81                   Reset      =>  Reset,
82                   Store      =>  Integrator_Store_1);
83
84  I6 : Adder
85       PORT MAP  ( Data_In_1  =>  Int_1,
86                   Data_In_2  =>  Int_2,
87                   Data_Out   =>  Int_3);
88
89  I7 : Adder
90       PORT MAP  ( Data_In_1  =>  Int_3,
91                   Data_In_2  =>  Int_4,
92                   Data_Out   =>  Integral);
93
94  I8 : Delay
95       PORT MAP  ( Data_In    =>  Integral,
96                   Data_Out   =>  Int_4,
97                   Reset      =>  Reset,
98                   Store      =>  Integrator_Store_2);
99
100
101 END ARCHITECTURE Structural;
```

Figure 10.18: (Continued)

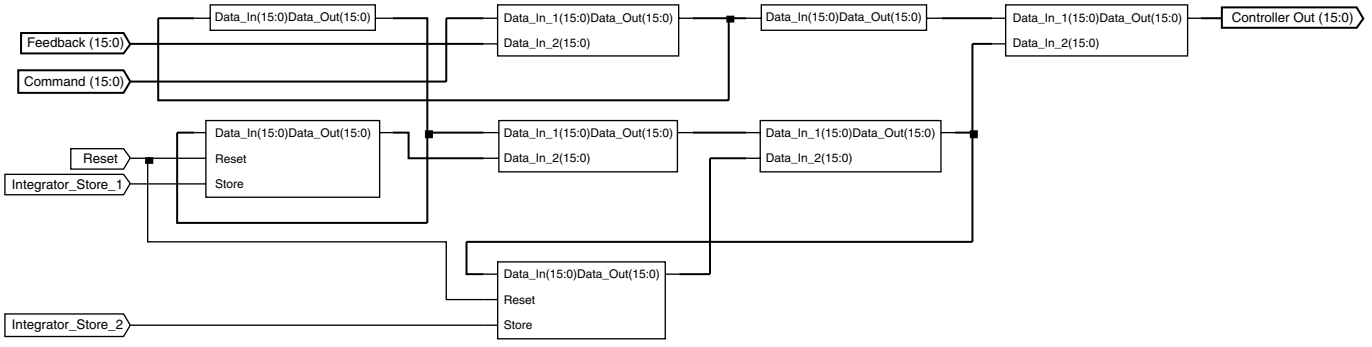

Figure 10.19: Digital control algorithm synthesis results (Coolrunner™-II CPLD)

```
 1  LIBRARY ieee;
 2  USE ieee.std_logic_1164.all;
 3  USE ieee.std_logic_arith.all;
 4  USE ieee.std_logic_unsigned.all;
 5
 6
 7  ENTITY Controller IS
 8   PORT ( Command_ADC_BUSY  : IN   STD_LOGIC;
 9          Command_ADC_TP    : OUT  STD_LOGIC;
10          Command_ADC_RD    : OUT  STD_LOGIC;
11          Command_ADC_CS    : OUT  STD_LOGIC;
12          Command_ADC_Data  : IN   STD_LOGIC_VECTOR (7 downto 0);
13          Feedback_ADC_BUSY : IN   STD_LOGIC;
14          Feedback_ADC_TP   : OUT  STD_LOGIC;
15          Feedback_ADC_RD   : OUT  STD_LOGIC;
16          Feedback_ADC_CS   : OUT  STD_LOGIC;
17          Feedback_ADC_Data : IN   STD_LOGIC_VECTOR(7 downto 0);
18          Controller_DAC_WR : OUT  STD_LOGIC;
19          Controller_DAC_CS : OUT  STD_LOGIC;
20          Controller_Out    : OUT  STD_LOGIC_VECTOR(7 downto 0);
21          Master_Clock      : IN   STD_LOGIC;
22          Master_Reset      : IN   STD_LOGIC);
23  END ENTITY Controller;
24
25
26  ARCHITECTURE Structural OF Controller IS
27
28
29  SIGNAL Command_Int          :  STD_LOGIC_VECTOR (15 downto 0);
30  SIGNAL Feedback_Int         :  STD_LOGIC_VECTOR (15 downto 0);
31  SIGNAL Controller_Out_Int   :  STD_LOGIC_VECTOR (15 downto 0);
32  SIGNAL Store_Command        :  STD_LOGIC;
33  SIGNAL Store_Feedback       :  STD_LOGIC;
34  SIGNAL Update_Out           :  STD_LOGIC;
35  SIGNAL Integrator_Store_1   :  STD_LOGIC;
36  SIGNAL Integrator_Store_2   :  STD_LOGIC;
37
38
39  COMPONENT Control_Algorithm IS
40   PORT ( Command             : IN  STD_LOGIC_VECTOR(15 downto 0);
41          Feedback            : IN  STD_LOGIC_VECTOR(15 downto 0);
42          Controller_Out      : OUT STD_LOGIC_VECTOR(15 downto 0);
43          Integrator_Store_1  : IN  STD_LOGIC;
44          Integrator_Store_2  : IN  STD_LOGIC;
45          Reset               : IN  STD_LOGIC);
46  END COMPONENT Control_Algorithm;
```

Figure 10.20: VHDL model for the controller

```
47
48  COMPONENT Control_Unit IS
49      PORT ( Master_Clock          : IN   STD_LOGIC;
50             Master_Reset          : IN   STD_LOGIC;
51             Command_ADC_BUSY      : IN   STD_LOGIC;
52             Command_ADC_TP        : OUT  STD_LOGIC;
53             Command_ADC_RD        : OUT  STD_LOGIC;
54             Command_ADC_CS        : OUT  STD_LOGIC;
55             Feedback_ADC_BUSY     : IN   STD_LOGIC;
56             Feedback_ADC_TP       : OUT  STD_LOGIC;
57             Feedback_ADC_RD       : OUT  STD_LOGIC;
58             Feedback_ADC_CS       : OUT  STD_LOGIC;
59             Controller_DAC_WR     : OUT  STD_LOGIC;
60             Controller_DAC_CS     : OUT  STD_LOGIC;
61             Store_Command         : OUT  STD_LOGIC;
62             Store_Feedback        : OUT  STD_LOGIC;
63             Update_Out            : OUT  STD_LOGIC;
64             Integrator_Store_1    : OUT  STD_LOGIC;
65             Integrator_Store_2    : OUT  STD_LOGIC);
66  END COMPONENT Control_Unit;
67
68
69  COMPONENT Input_Register IS
70      PORT ( Data_In   :  IN   STD_LOGIC_VECTOR (7 downto 0);
71             Data_Out  :  OUT  STD_LOGIC_VECTOR (15 downto 0);
72             Store     :  IN   STD_LOGIC;
73             Reset     :  IN   STD_LOGIC);
74  END COMPONENT Input_Register;
75
76
77  COMPONENT Output_Register IS
78      PORT ( Data_In   :  IN   STD_LOGIC_VECTOR (15 downto 0);
79             Data_Out  :  OUT  STD_LOGIC_VECTOR (7 downto 0);
80             Store     :  IN   STD_LOGIC;
81             Reset     :  IN   STD_LOGIC);
82  END COMPONENT Output_Register;
83
84
85  BEGIN
86
87  I1 : Control_Algorithm
88       PORT MAP(  Command              =>  Command_Int,
89                  Feedback             =>  Feedback_Int,
90                  Controller_Out       =>  Controller_Out_Int,
91                  Integrator_Store_1   =>  Integrator_Store_1,
92                  Integrator_Store_2   =>  Integrator_Store_2,
```

Figure 10.20: (Continued)

```
93                  Reset                 =>  Master_Reset);
94
95   I2 : Control_Unit
96        PORT MAP(  Master_Clock          =>  Master_Clock,
97                   Master_Reset          =>  Master_Reset,
98                   Command_ADC_BUSY      =>  Command_ADC_BUSY,
99                   Command_ADC_TP        =>  Command_ADC_TP,
100                  Command_ADC_RD        =>  Command_ADC_RD,
101                  Command_ADC_CS        =>  Command_ADC_CS,
102                  Feedback_ADC_BUSY     =>  Feedback_ADC_BUSY,
103                  Feedback_ADC_TP       =>  Feedback_ADC_TP,
104                  Feedback_ADC_RD       =>  Feedback_ADC_RD,
105                  Feedback_ADC_CS       =>  Feedback_ADC_CS,
106                  Controller_DAC_WR     =>  Controller_DAC_WR,
107                  Controller_DAC_CS     =>  Controller_DAC_CS,
108                  Store_Command         =>  Store_Command,
109                  Store_Feedback        =>  Store_Feedback,
110                  Update_Out            =>  Update_Out,
111                  Integrator_Store_1    =>  Integrator_Store_1,
112                  Integrator_Store_2    =>  Integrator_Store_2);
113
114  I3 : Input_Register
115       PORT MAP ( Data_In               =>  Command_ADC_Data,
116                  Data_Out              =>  Command_Int,
117                  Store                 =>  Store_Command,
118                  Reset                 =>  Master_Reset);
119
120
121  I4 : Input_Register
122       PORT MAP ( Data_In               =>  Feedback_ADC_Data,
123                  Data_Out              =>  Feedback_Int,
124                  Store                 =>  Store_Feedback,
125                  Reset                 =>  Master_Reset);
126
127  I5 : Output_Register
128       PORT MAP ( Data_In               =>  Controller_Out_Int,
129                  Data_Out              =>  Controller_Out,
130                  Store                 =>  Update_Out,
131                  Reset                 =>  Master_Reset);
132
133
134  END ARCHITECTURE Structural;
```

Figure 10.20: (Continued)

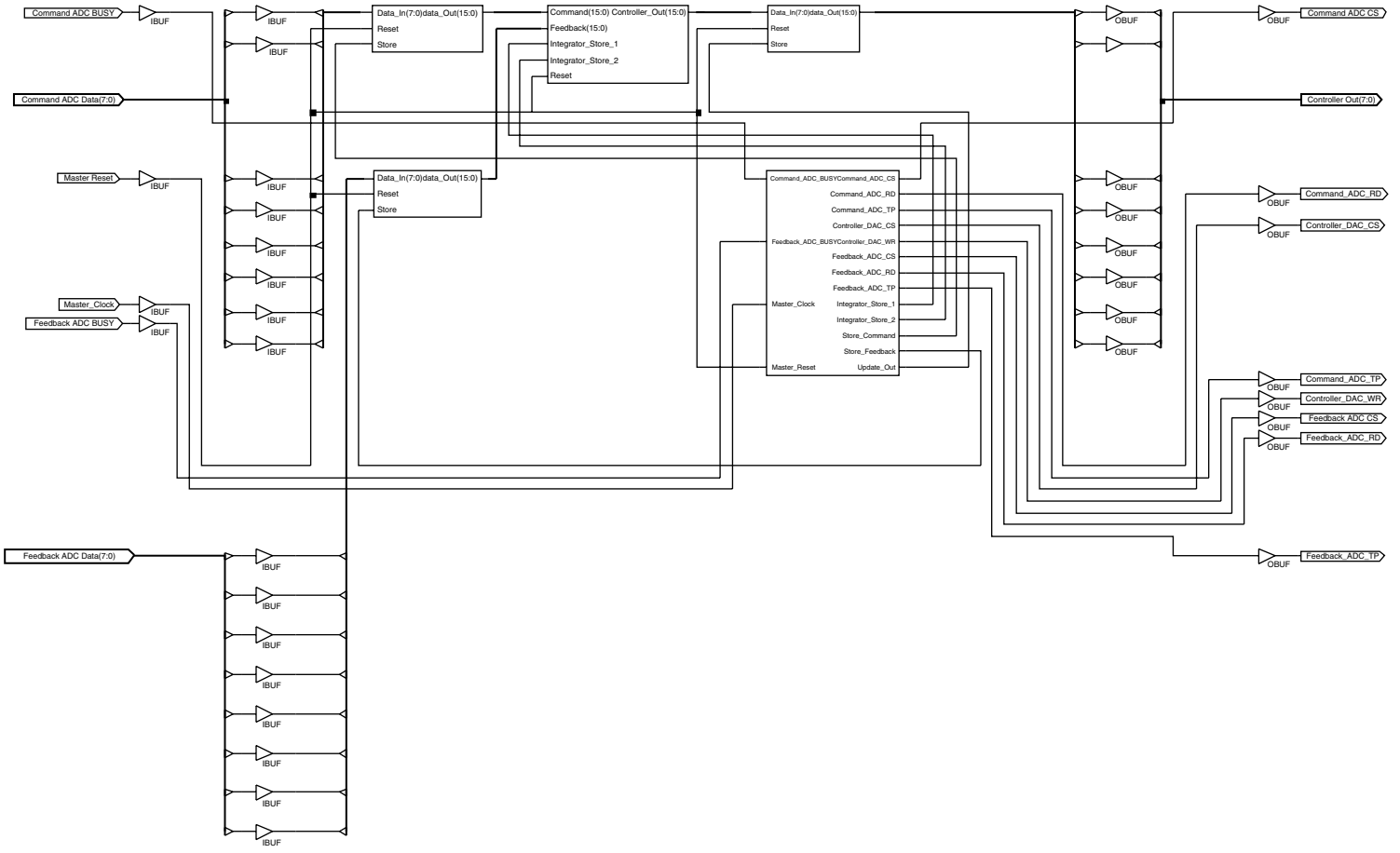

Figure 10.21: Digital controller synthesis results (Coolrunner™-II CPLD)

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
USE ieee.std_logic_unsigned.all;


ENTITY Test_Controller_vhd IS
END Test_Controller_vhd;


ARCHITECTURE Behavioural OF Test_Controller_vhd IS


COMPONENT Controller
PORT(
               Command_ADC_BUSY     : IN  STD_LOGIC;
               Command_ADC_Data     : IN  STD_LOGIC_VECTOR(7 downto 0);
               Feedback_ADC_BUSY    : IN  STD_LOGIC;
               Feedback_ADC_Data    : IN  STD_LOGIC_VECTOR(7 downto 0);
               Master_Clock         : IN  STD_LOGIC;
               Master_Reset         : IN  STD_LOGIC;
               Command_ADC_TP       : OUT STD_LOGIC;
               Command_ADC_RD       : OUT STD_LOGIC;
               Command_ADC_CS       : OUT STD_LOGIC;
               Feedback_ADC_TP      : OUT STD_LOGIC;
               Feedback_ADC_RD      : OUT STD_LOGIC;
               Feedback_ADC_CS      : OUT STD_LOGIC;
               Controller_DAC_WR    : OUT STD_LOGIC;
               Controller_DAC_CS    : OUT STD_LOGIC;
               Controller_Out       : OUT STD_LOGIC_VECTOR(7 downto 0));
END COMPONENT;


SIGNAL Command_ADC_BUSY    :  STD_LOGIC := '0';
SIGNAL Feedback_ADC_BUSY   :  STD_LOGIC := '0';
SIGNAL Master_Clock        :  STD_LOGIC := '0';
SIGNAL Master_Reset        :  STD_LOGIC := '0';
SIGNAL Command_ADC_Data    :  STD_LOGIC_VECTOR(7 downto 0) := (others=>'0');
SIGNAL Feedback_ADC_Data   :  STD_LOGIC_VECTOR(7 downto 0) := (others=>'0');


SIGNAL Command_ADC_TP      :  STD_LOGIC;
SIGNAL Command_ADC_RD      :  STD_LOGIC;
SIGNAL Command_ADC_CS      :  STD_LOGIC;
SIGNAL Feedback_ADC_TP     :  STD_LOGIC;
SIGNAL Feedback_ADC_RD     :  STD_LOGIC;
SIGNAL Feedback_ADC_CS     :  STD_LOGIC;
SIGNAL Controller_DAC_WR   :  STD_LOGIC;
SIGNAL Controller_DAC_CS   :  STD_LOGIC;
SIGNAL Controller_Out      :  STD_LOGIC_VECTOR(7 downto 0);


```

Figure 10.22: VHDL test bench for the controller

```
53  BEGIN
54
55
56  uut: Controller PORT MAP(
57                  Command_ADC_BUSY   =>  Command_ADC_BUSY,
58                  Command_ADC_TP     =>  Command_ADC_TP,
59                  Command_ADC_RD     =>  Command_ADC_RD,
60                  Command_ADC_CS     =>  Command_ADC_CS,
61                  Command_ADC_Data   =>  Command_ADC_Data,
62                  Feedback_ADC_BUSY  =>  Feedback_ADC_BUSY,
63                  Feedback_ADC_TP    =>  Feedback_ADC_TP,
64                  Feedback_ADC_RD    =>  Feedback_ADC_RD,
65                  Feedback_ADC_CS    =>  Feedback_ADC_CS,
66                  Feedback_ADC_Data  =>  Feedback_ADC_Data,
67                  Controller_DAC_WR  =>  Controller_DAC_WR,
68                  Controller_DAC_CS  =>  Controller_DAC_CS,
69                  Controller_Out     =>  Controller_Out,
70                  Master_Clock       =>  Master_Clock,
71                  Master_Reset       =>  Master_Reset);
72
73
74  Reset_Process : PROCESS
75          BEGIN
76
77          Wait for 0 ns;  Master_Reset <= '0';
78          Wait for 5 ns;  Master_Reset <= '1';
79          Wait;
80
81  END PROCESS;
82
83
84  Clock_Process : PROCESS
85          BEGIN
86
87          Wait for 0 ns;  Master_Clock <= '0';
88          Wait for 10 ns; Master_Clock <= '1';
89          Wait for 10 ns; Master_Clock <= '0';
90
91  END PROCESS;
92
93
94  ADC_Data_Process : PROCESS
95          BEGIN
96
97           Wait for 0 ns;    Command_ADC_Data <= "00000000";
98                             Feedback_ADC_Data <= "00000000";
99          Wait;
100
101 END PROCESS;
102
103
104 ADC_Busy_Process : PROCESS
```

Figure 10.22: (Continued)