A-PDF Split DEMO : Purchase from www.A-PDF.com to remove the watermark


```
105         BEGIN
106
107          Wait for 0 ns;     Command_ADC_BUSY <= '0';
108                             Feedback_ADC_BUSY <= '0';
109          Wait;
110
111   END PROCESS;
112
113
114   END ARCHITECTURE Behavioural;
```

Figure 10.22: (Continued)

VHDL code by a direct mapping for this small design reduced the design time. The multiplications were undertaken within the *Proportional_Gain* and *Integrator_Gain* blocks. The design can use either a full 16×16 multiplier design or a shift-and-add approach. Given that the multiplications are fixed and relatively simple, a full multiplier design can be expected to produce a larger hardware design than necessary.

An internal wordlength of 16 bits is used in this case study and must be considered in the calculations performed. Where the potential for number overflow existed, this was prevented either by ensuring that the internal values are never large enough to create an overflow, or if an overflow situation does occur, by saturating the output from a computation to the limits set by the wordlength. The internal multiplication within the integrator gain also produces a number with integer and fractional parts. Therefore, for a fixed-point calculation, the lower part of the 16-bit wordlength must be used to represent the fractional part, and the upper part must be used to represent the integer part. Placing the decimal point in the number is a design decision. If the finite wordlength creates errors in calculations, that information is fed back to the original simulation model for the control system and used to modify the controller.

10.4 Case Study 2: Digital Filter Design

10.4.1 Introduction

Digital filters perform the operations of addition, subtraction, multiplication, and division on sampled data. Among the types of digital filter are the infinite impulse response (IIR) filter, the finite impulse response (FIR) filter [15], and the

computationally efficient cascaded integrator comb (CIC) filter [16]. The CIC filter is widely used in decimation and interpolation in communications systems:

- *Decimation* is the process of sample rate reduction. Where a signal is sampled at a particular sampling rate, a decimator reduces the original sample rate (f_s) to a lower rate (f_s/M). For example, if a signal is sampled at 10 kHz and M = 5, a decimator outputs a value once every M samples and discards the other (M – 1) samples. When M = 5, the sample rate is reduced from 10 kHz to 2 kHz.
- *Interpolation* is the process of sample rate increase. Where a signal is sampled at a particular sampling rate, the interpolation process increases the sample rate (f_s) to a higher rate (Lf_s). For example, if a signal is sampled at 10 kHz and L = 5, an interpolator outputs a value at an increased rate of 50 kHz. The input sample is output once every L output values, and the interpolator will fill the remaining (L – 1) output values with a zero value.

This idea is shown in Figure 10.23.

Decimation and interpolation functions are used in communications systems and in circuits such as the digital signal conditioning circuitry within sigma-delta modulator architecture ADCs and DACs. For example, CIC filters are suited for digital anti-aliasing filtering prior to decimation; a typical arrangement is shown in Figure 10.24. Here, the input is applied to the CIC filter, and the output from the CIC filter is applied to an FIR filter.

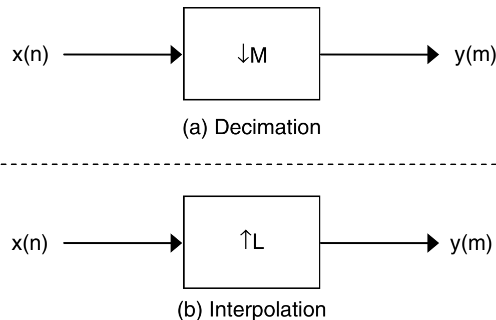

Figure 10.23: Decimation and interpolation

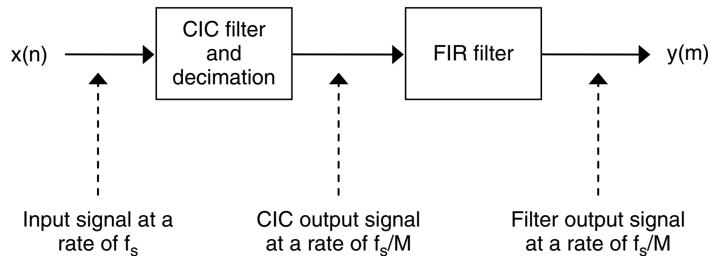

Figure 10.24: CIC filter in decimation

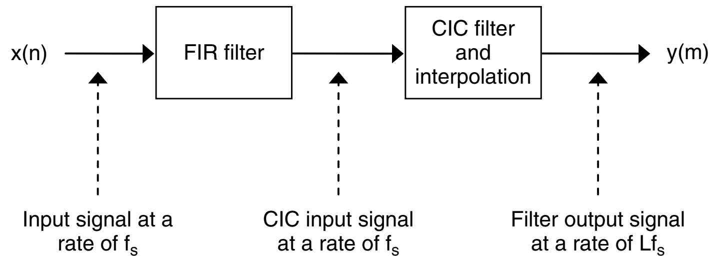

Figure 10.25: CIC filter in interpolation

A CIC filter used in interpolation is shown in Figure 10.25. Here, the input is applied to the FIR filter, and the output from the FIR filter is applied to an CIC filter.

In this case study, a third-order digital CIC filter will be developed to filter a single-bit bitstream pattern. The overall filter model will be developed in MATLAB® [9] and its Simulink® toolbox [10]. The model of the filter algorithm will then be manually converted to VHDL code using a set design translation flow for implementation as a digital filter 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 10.26.

10.4.2 Filter Overview

The CIC filter consists of an integrator section and a comb section. The integrator implements integration of the signal, and the comb implements differentiation on the signal. The operation of the CIC filter is well explained in many texts, so it is not considered further here. For use in decimation, the CIC filter has the form shown in Figure 10.27. This design is for a third-order CIC filter with three integrator and three

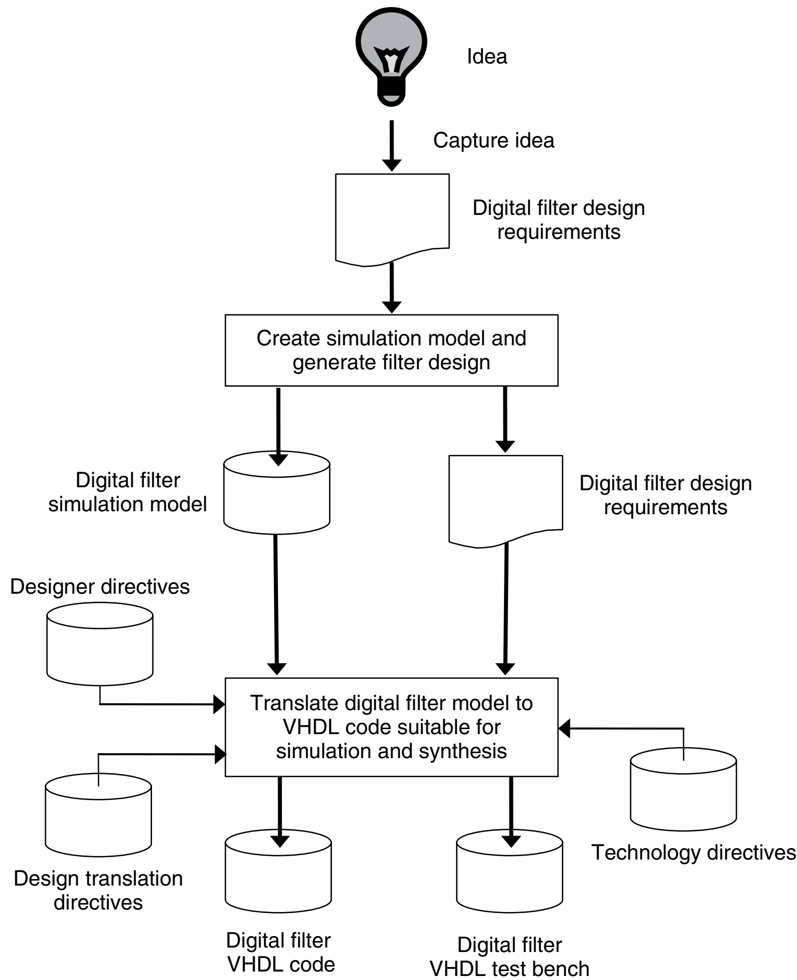

Figure 10.26: CIC filter case study design flow

comb circuits. (A fourth-order CIC filter would use four integrators and four comb circuits, etc.) Variations on this basic structure are possible. Note that the decimator is placed between the integrator and comb parts of the design.

The integrator is modeled using Z-transforms as:

$$\frac{\text{Output}(z)}{\text{Input}(z)} = \left(\frac{1}{1 - z^{-1}}\right)$$

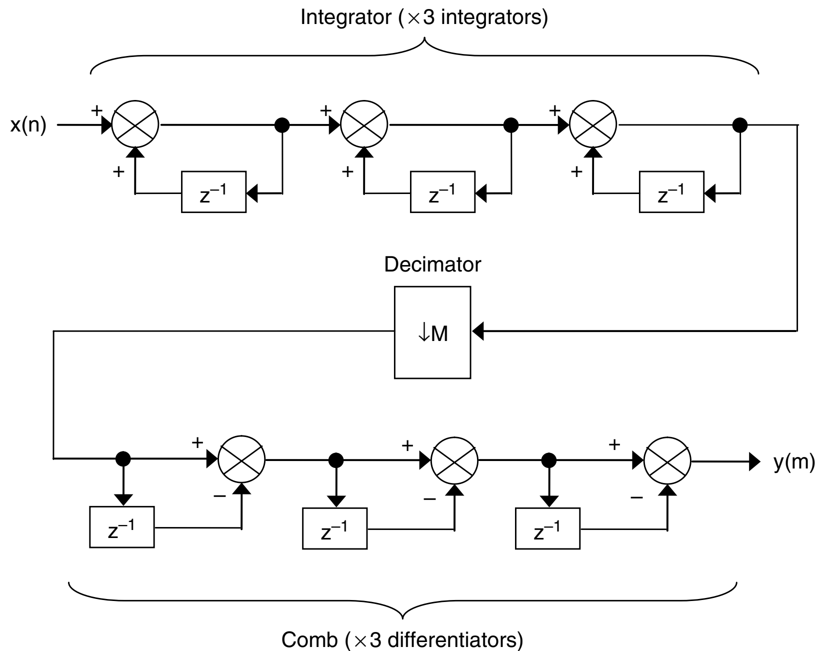

Figure 10.27: Third-order CIC filter in decimation

The differentiator is modeled using Z-transforms as:

$$\frac{\text{Output}(z)}{\text{Input}(z)} = \left(1 - z^{-1}\right)$$

Each delay block has a control signal to store the input to the delay. Note that these forms for the integrator and differentiator differ from those presented in Chapter 7.

10.4.3 MATLAB®/Simulink® Model Creation and Simulation

Before considering how the controller is to be implemented, the algorithm must be developed and analyzed. An example Simulink® model for this system is shown in Figure 10.28.

Here, the CIC filter is separated into the integrator and differentiator parts. Within the integrators, the inputs are sampled at a sampling rate of f_s. Within

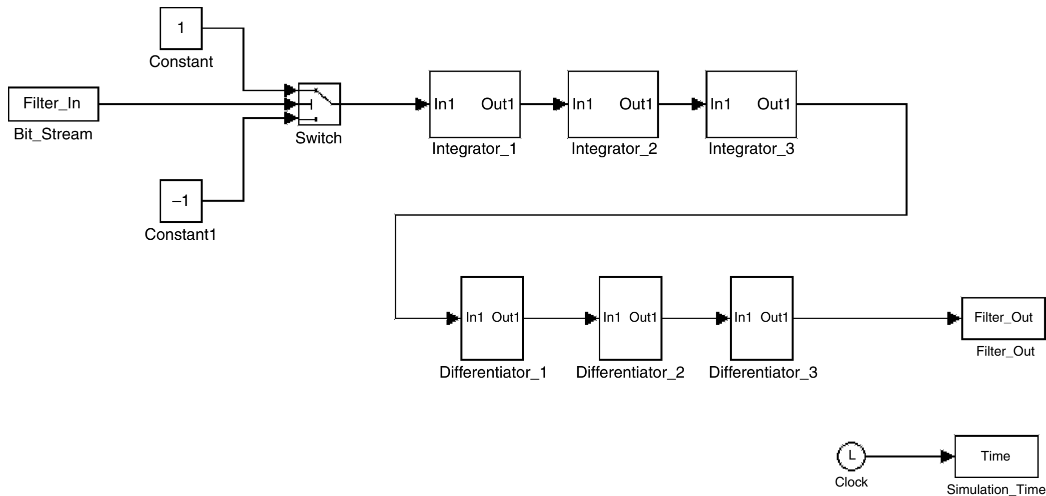

Figure 10.28: Simulink® model for the CIC filter

the differentiators, the inputs are sampled at a sampling rate of (f_s/M). In this model, the CIC filter is intended for use in the digital signal conditioning circuitry within a sigma-delta ADC design. A single-bit bitstream pattern (Filter_In) is applied to the filter input, and a 16-bit output from the CIC filter (Filter_Out) is created. This is achieved by a switch block at the input of the filter such that:

- When the input is a logic 0, then using 2s complement arithmetic, a value of -1_{10} is applied to the filter input.
- When the input is a logic 1, then using 2s complement arithmetic, a value of $+1_{10}$ is applied to the filter input.

The integrator design is shown in Figure 10.29, and the differentiator design is shown in Figure 10.30.

The design is analyzed using both hand calculations and the Simulink® simulator, with typical bitstream patterns representing different signal frequencies as part of the overall system analysis routine. A frequency response could be undertaken by generating a model for frequency analysis in MATLAB®.

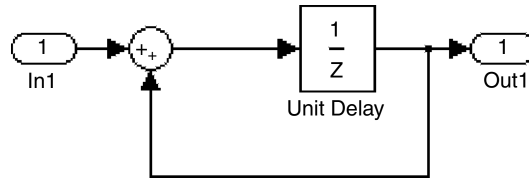

Figure 10.29: Simulink® model for the integrator

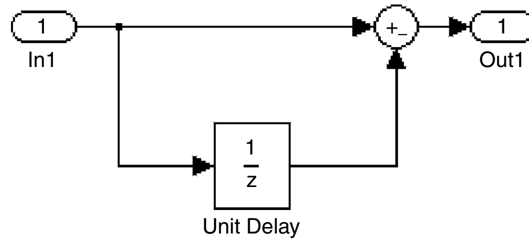

Figure 10.30: Simulink® model for the differentiator

10.4.4 Translating the Design to VHDL

After system analysis of the system has been completed, the digital filter 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 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).
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 filter.

The nine steps of translation preparation are:

1. Identify the parts to be translated into digital (the filter).
2. Remove any unnecessary information, leaving only the filter model.
3. Identify the digital filter interfacing.
4. Identify the clock and reset inputs, along with any other filter 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 which architecture to use, either a processor-based architecture running a software application (standard fixed

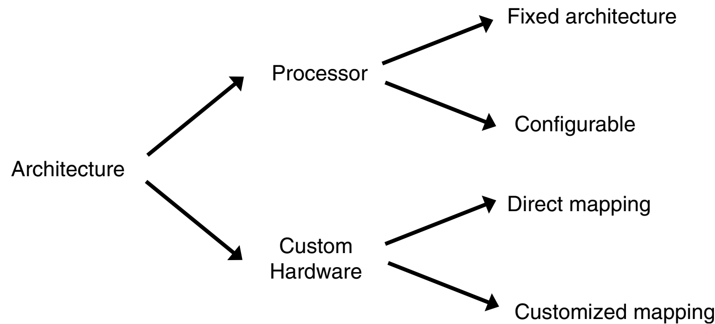

Figure 10.31: Filter architecture decisions

architecture processor or a configurable processor) or a custom hardware architecture based directly on the model. This idea is shown in Figure 10.31.

If the translation were performed manually, this could be accomplished by visual reference to the graphical representation of the model (i.e., the block diagram). If the translation were performed automatically (by a software application), it could be accomplished 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 the translation either will generate the hardware design (in HDL) and the processor microcode together, or will use an existing processor design and only generate the processor microcode. 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 filter shown in Figure 10.28 will be translated. This requires the use of the following main functional blocks:

- three integrator blocks
- three differentiator blocks
- one switch block
- two constant values

The input is a single-bit bitstream pattern, and the output is a 16-bit pattern. 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

The information *not* for translation to VHDL includes information such as visual attributes and software version information, which must be stripped from the representation of the model used for translation to VHDL.
The Simulink® model code for the filter only is shown in Figure 10.32.
This is the text description of the model shown in Figure 10.28. It consists of the blocks used, their attributes, and the interconnect between the blocks (lines). Interpreting this model requires knowledge of its model syntax and how the values that can be modified by the user are represented in the model. The syntax is readable, and the names used can be identified by comparison with the block diagram view.

To create a digital design to implement the filter, a control unit is needed within the design to generate the necessary timing signals to control the operation of the filter parts from master clock and reset inputs. The basic structure for this is shown in Figure 10.33.

```
1   System {
2      Name                      "comb_filter_1"
3      Block {
4        BlockType                       FromWorkspace
5        Name                    "Bit_Stream"
6        Position                [25, 78, 90, 102]
7        VariableName            "Filter_In"
8        SampleTime              "0"
9      }
10     Block {
11       BlockType                       Constant
12       Name                    "Constant"
13       Position                [135, 20, 165, 50]
14     }
15     Block {
16       BlockType                       Constant
17       Name                    "Constant1"
18       Position                [135, 145, 165, 175]
19       Value                   "-1"
20     }
21     Block {
22       BlockType                       SubSystem
23       Name                    "Differentiator_1"
24       Ports                   [1, 1]
25       Position                [355, 215, 395, 275]
26       TreatAsAtomicUnit               off
27     }
28     Block {
29       BlockType                       SubSystem
30       Name                    "Differentiator_2"
31       Ports                   [1, 1]
32       Position                [455, 215, 495, 275]
33       TreatAsAtomicUnit               off
34     }
35     Block {
36       BlockType                       SubSystem
37       Name                    "Differentiator_3"
38       Ports                   [1, 1]
39       Position                [550, 215, 590, 275]
40       TreatAsAtomicUnit               off
41     }
42     Block {
43       BlockType                       ToWorkspace
44       Name                    "Filter_Out"
45       Position                [715, 230, 775, 260]
46       VariableName            "Filter_Out"
47       MaxDataPoints           "inf"
48       SampleTime              "-1"
49       SaveFormat              "Structure"
50     }
51     Block {
52       BlockType                       SubSystem
53       Name                    "Integrator_1"
54       Ports                   [1, 1]
55       Position                [330, 66, 395, 114]
56       TreatAsAtomicUnit               off
57     }
```

Figure 10.32: Simulink® model for the CIC filter

```
58      Block {
59        BlockType                   SubSystem
60        Name                  "Integrator_2"
61        Ports                 [1, 1]
62        Position              [430, 66, 495, 114]
63        TreatAsAtomicUnit           off
64      }
65      Block {
66        BlockType                   SubSystem
67        Name                  "Integrator_3"
68        Ports                 [1, 1]
69        Position              [530, 66, 595, 114]
70        TreatAsAtomicUnit           off
71      }
72      Block {
73        BlockType                   Switch
74        Name                  "Switch"
75        Position              [235, 75, 265, 105]
76        InputSameDT           off
77      }
78      Line {
79        SrcBlock              "Bit_Stream"
80        SrcPort               1
81        DstBlock              "Switch"
82        DstPort               2
83      }
84      Line {
85        SrcBlock              "Constant"
86        SrcPort               1
87        Points                [25, 0; 0, 45]
88        DstBlock              "Switch"
89        DstPort               1
90      }
91      Line {
92        SrcBlock              "Constant1"
93        SrcPort               1
94        Points                [25, 0; 0, -60]
95        DstBlock              "Switch"
96        DstPort               3
97      }
98      Line {
99        SrcBlock              "Switch"
100       SrcPort               1
101       DstBlock              "Integrator_1"
102       DstPort               1
103     }
104     Line {
105       SrcBlock              "Integrator_1"
106       SrcPort               1
107       DstBlock              "Integrator_2"
108       DstPort               1
109     }
110     Line {
111       SrcBlock              "Integrator_2"
112       SrcPort               1
113       DstBlock              "Integrator_3"
114       DstPort               1
115     }
```

Figure 10.32: (Continued)

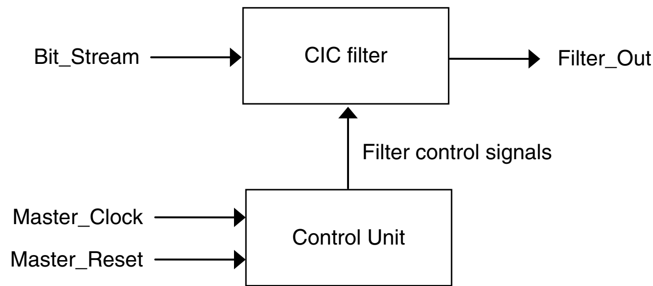

Figure 10.33: Digital filter control

The CIC filter can be remodeled in VHDL, shown in Figure 10.34 as a structural description for the filter. Detailed operation of each of the blocks is defined in separate entity-architecture pairs.

10.4.5 Concluding Remarks

In this case study, a third-order CIC digital filter was developed as a Simulink® block diagram and translated to a VHDL model for implementation within a CPLD. The structural VHDL description for the CIC filter section of a digital core was developed. The following VHDL code is also needed to configure the CPLD:

- top-level design containing the CIC filter and the control unit
- switch block details
- integrator details
- differentiator details

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 where multiple repeated operations are needed. However, the ease and rapid development of the VHDL code by direct mapping for this small design reduced design time. This design included no multiplications, so the multiplier implementation required in other digital filter designs was not needed.

An internal wordlength of 16 bits was required for this design, which has to be accommodated in the calculations. When number overflow was possible in the

```
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 CIC_Filter IS
8       Port ( Bit_Stream    : IN   STD_LOGIC;
9              Master_Clock  : IN   STD_LOGIC;
10             Master_Reset  : IN   STD_LOGIC;
11             Filter_Out    : OUT  STD_LOGIC_VECTOR (15 downto 0));
12  END ENTITY CIC_Filter;
13
14
15
16  ARCHITECTURE Structural OF CIC_Filter IS
17
18
19  SIGNAL Internal_1               :  STD_LOGIC_VECTOR(15 downto 0);
20  SIGNAL Internal_2               :  STD_LOGIC_VECTOR(15 downto 0);
21  SIGNAL Internal_3               :  STD_LOGIC_VECTOR(15 downto 0);
22  SIGNAL Internal_4               :  STD_LOGIC_VECTOR(15 downto 0);
23  SIGNAL Internal_5               :  STD_LOGIC_VECTOR(15 downto 0);
24  SIGNAL Internal_6               :  STD_LOGIC_VECTOR(15 downto 0);
25  SIGNAL Internal_7               :  STD_LOGIC_VECTOR(15 downto 0);
26  SIGNAL Internal_8               :  STD_LOGIC_VECTOR(15 downto 0);
27  SIGNAL Integrator_1_Store       :  STD_LOGIC;
28  SIGNAL Integrator_2_Store       :  STD_LOGIC;
29  SIGNAL Integrator_3_Store       :  STD_LOGIC;
30  SIGNAL Differentiator_1_Store   :  STD_LOGIC;
31  SIGNAL Differentiator_2_Store   :  STD_LOGIC;
32  SIGNAL Differentiator_3_Store   :  STD_LOGIC;
33
34
35  COMPONENT Switch IS
36      PORT ( Bit_Stream     :  IN   STD_LOGIC;
37             Data_In_1      :  IN   STD_LOGIC_VECTOR (15 downto 0);
38             Data_In_2      :  IN   STD_LOGIC_VECTOR (15 downto 0);
39             Data_Out       :  OUT  STD_LOGIC_VECTOR (15 downto 0));
40  END COMPONENT Switch;
41
42
43  COMPONENT Plus_One IS
44      PORT ( Data_Out   : OUT  STD_LOGIC_VECTOR (15 downto 0));
45  END COMPONENT Plus_One;
46
47
```

Figure 10.34: VHDL model for the CIC filter

```
48  COMPONENT Minus_One IS
49      PORT ( Data_Out   : OUT  STD_LOGIC_VECTOR (15 downto 0));
50  END COMPONENT Minus_One;
51
52
53  COMPONENT Integrator IS
54      PORT ( Data_In    : IN   STD_LOGIC_VECTOR (15 downto 0);
55             Data_Out   : OUT  STD_LOGIC_VECTOR (15 downto 0);
56             Store      : IN   STD_LOGIC;
57             Reset      : IN   STD_LOGIC);
58  END COMPONENT Integrator;
59
60
61  COMPONENT Differentiator IS
62      PORT ( Data_In    : IN   STD_LOGIC_VECTOR (15 downto 0);
63             Data_Out   : OUT  STD_LOGIC_VECTOR (15 downto 0);
64             Store      : IN   STD_LOGIC;
65             Reset      : IN   STD_LOGIC);
66  END COMPONENT Differentiator;
67
68
69  COMPONENT Control_Unit is
70      PORT ( Master_Clock             : IN   STD_LOGIC;
71             Master_Reset             : IN   STD_LOGIC;
72             Integrator_1_Store       : OUT  STD_LOGIC;
73             Integrator_2_Store       : OUT  STD_LOGIC;
74             Integrator_3_Store       : OUT  STD_LOGIC;
75             Differentiator_1_Store   : OUT  STD_LOGIC;
76             Differentiator_2_Store   : OUT  STD_LOGIC;
77             Differentiator_3_Store   : OUT  STD_LOGIC);
78  END COMPONENT Control_Unit;
79
80
81  BEGIN
82
83  I1: Switch
84      PORT MAP ( Bit_Stream             =>  Bit_Stream,
85                 Data_In_1              =>  Internal_1,
86                 Data_In_2              =>  Internal_2,
87                 Data_Out               =>  Internal_3);
88
89  I2 : Plus_One
90       PORT MAP ( Data_Out              =>  Internal_1);
91
92  I3 : Minus_One
93       PORT MAP ( Data_Out              =>  Internal_2);
94
```

Figure 10.34: (Continued)

```
 95  I4 : Integrator
 96       PORT MAP ( Data_In                 =>  Internal_3,
 97                  Data_Out                =>  Internal_4,
 98                  Store                   =>  Integrator_1_Store,
 99                  Reset                   =>  Master_Reset);
100
101  I5 : Integrator
102       PORT MAP ( Data_In                 =>  Internal_4,
103                  Data_Out                =>  Internal_5,
104                  Store                   =>  Integrator_2_Store,
105                  Reset                   =>  Master_Reset);
106
107  I6 : Integrator
108       PORT MAP ( Data_In                 =>  Internal_5,
109                  Data_Out                =>  Internal_6,
110                  Store                   =>  Integrator_3_Store,
111                  Reset                   =>  Master_Reset);
112
113  I7 : Differentiator
114       PORT MAP ( Data_In                 =>  Internal_6,
115                  Data_Out                =>  Internal_7,
116                  Store                   =>  Differentiator_1_Store,
117                  Reset                   =>  Master_Reset);
118
119  I8 : Differentiator
120       PORT MAP ( Data_In                 =>  Internal_7,
121                  Data_Out                =>  Internal_8,
122                  Store                   =>  Differentiator_2_Store,
123                  Reset                   =>  Master_Reset);
124
125  I9 : Differentiator
126       PORT MAP ( Data_In                 =>  Internal_8,
127                  Data_Out                  => Filter_Out,
128                  Store                     => Differentiator_3_Store,
129                  Reset                     => Master_Reset);
130
131  I10 : Control_Unit
132        PORT MAP ( Master_Clock             => Master_Clock,
133                   Master_Reset             => Master_Reset,
134                   Integrator_1_Store       => Integrator_1_Store,
135                   Integrator_2_Store       => Integrator_2_Store,
136                   Integrator_3_Store       => Integrator_3_Store,
137                   Differentiator_1_Store => Differentiator_1_Store,
138                   Differentiator_2_Store => Differentiator_2_Store,
139                   Differentiator_3_Store => Differentiator_3_Store);
140
141   END ARCHITECTURE Structural;
```

Figure 10.34: (Continued)

integrators, it was prevented either by ensuring that the internal values encountered are never large enough to create an overflow situation, or if an overflow situation could occur, saturating the output from a computation to the limits set by the wordlength.

10.5 Automating the Translation

The two case studies presented provide a snapshot of two possible target applications for the automatic generation of VHDL code from a system-level simulation model. A number of design implementation issues were raised and solved for these two scenarios. However, for automating the translation process into VHDL, the translation steps must be adaptable to a more generic application. Any possible approach to automating model translation, however, must:

1. be capable of being manually undertaken (i.e., by hand) if required
2. allow the designer to enter specific requirements for the particular application
3. be presented to the designer in a way that is familiar to his or her particular engineering domain and technical language
4. not intentionally restrict the designer to such an extent that the translation tool cannot be used
5. be aware that different versions of the software can vary the syntax of the underlying text file containing the model description, so a translation tool written for one version of the simulation software must be validated for a different version
6. select a software programming language appropriate to the end use of the application
7. select an architecture appropriate to the required operation and coding styles in VHDL
8. be developed in a modular manner so that the translation tool can be readily modified and enhanced
9. consider timing issues in the underlying digital logic
10. consider testability issues for the designs to be implemented

11. Effectively and efficiently deal with design hierarchy

12. Consider the circuit functions that are required to support the algorithm modelled at the high level of description, but which are not modelled at this level. For example, in digital designs there would be the need to include some form of circuit *control*. This would be required to perform synchronisation of signals around the circuit and ensure that the correct data flow is provided. In addition, specific control signals for signal sampling (e.g. through and ADC) and output updating (e.g. through a DAC) would be required

13. Include the capability for automatic documentation creation as part of the translation process. This can be in the form of document formats such as plain text, postscript, portable document format (PDF) and hypertext markup language (HTML)

10.6 Future Directions

The area of ESL design is still emerging, and various activities are undertaken in defining the direction for ESL design. However, there is a basic need to combine into a single and robust design methodology multiple design methods, EDA tools, and implementation technologies. With the area of ESL design dynamically changing, designers must be aware of the technologies, ESL design methodologies, and EDA tools that are becoming available to provide the right approach for the types of complex electronic systems being developed. This will come from the collaboration between the developers and the design community. Initially, a number of different approaches will be adopted; those showing the most promise will ultimately become industry standards, adopted and formally developed by one or more of the professional bodies.

Alongside the systems-level design methods and EDA tools being developed to solve the complex problems encountered today and expected in the future, there is still the need for electronic circuit and computer software designers who work at the most detailed level of design. Advances at this detailed level allow more complex systems to be developed of smaller size, in less time, and at lower cost. No matter how complex a system becomes, the devil will always remain in the details, so the need for effective communication and collaboration among the designers working at all levels of abstraction will always exist.