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

CHAPTER 7

Introduction to Digital Signal Processing

7.1 Introduction

The processing of analogue electrical signals and digital data from one form to another is fundamental to many electronic circuits and systems. Both analogue (voltage and current) signals and digital (logic value) data can be processed by many types of circuits, and the task of finding the right design is a sometimes confusing but normal part of the design process. It depends on identifying the benefits and limitations of the possible implementations to select the most appropriate solution for the particular scenario. Initial concerns are:

- Is the input analogue or digital?
- Is the output analogue or digital?
- Will signal processing use analogue or digital techniques?

This idea is shown in Figure 7.1, where signal processing uses either an analogue signal processor (ASP) or a digital signal processor (DSP). If an analogue signal is to be processed or output as digital data, then the analogue signal must be converted to digital using the analogue-to-digital converter (ADC). The operation of this circuit is discussed in Chapter 8. If a digital signal is to be processed or output as an analogue signal, then the digital data will be converted to analogue using the digital-to-analogue converter (DAC). The operation of this circuit is also discussed in Chapter 8.

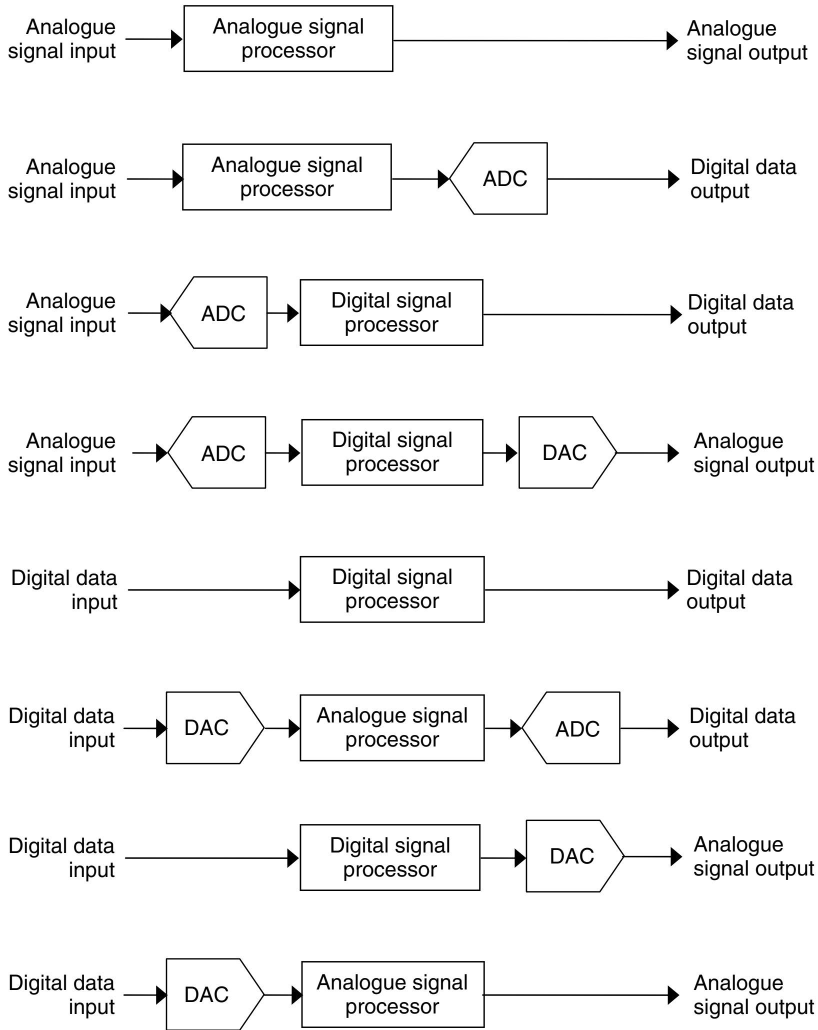


Figure 7.1: Processing of analogue signals and digital data

ASP and DSP each has its own advantages and disadvantages:

Analogue implementation:

Advantages:

- high bandwidth (from DC up to high signal frequencies)
- high resolution

- ease of design
- good approach for simpler design solutions

Disadvantages:

- component value change occurs with component aging
- component value change occurs with temperature variations
- behavior variance between manufactured circuits due to component tolerances
- difficult to change circuit operation

Digital implementation:

Advantages:

- programmable and configurable solution (either programmed in software on a processor or configured in hardware on a CPLD/FPGA)
- operation insensitive to temperature variations
- precise behavior (no behavior variance due to varying component tolerances)
- can implement algorithms that cannot be implemented in analogue
- ease of upgrading and modifying the design

Disadvantages:

- implementation issues due to issues related to numerical calculations
- requires high-performance digital processing
- design complexity
- higher cost

Increasingly, digital implementations are the preferred choice because of their advantages over analogue and because of the ability to implement advanced

algorithms that are only possible in the digital domain. In many cases where there are analogue signals and also a requirement for analogue circuitry, the analogue circuitry is kept to a minimum, and the majority of the work performed by the circuit uses digital techniques. The two main areas for digital signal processing considered in this text are digital filters [1–4] and digital control algorithms [5–7].

These can be implemented both in software on the microprocessor (μP), microcontroller (μC), or the digital signal processor (DSP) and in hardware on the complex programmable logic device (CPLD) or field programmable gate array (FPGA). The basis for all possible implementation approaches is a circuit design that will accept samples of digitized analogue signals or direct digital data, perform an algorithm that uses the current sampled value and previous sampled values, and output the digital data directly or in analogue form. The algorithm to be implemented is typically developed using the Z-transform. This algorithm is an equation (or set of equations) that defines a current output in terms of the sums and differences of a current input sample and previous input samples, along with weighting factors. However, to achieve a working implementation of the algorithm, a number of key steps are required:

- analysis of the signal to filter or system to control
- creation of the design specification
- design of the algorithm to fulfill the design requirements
- simulation of the operation of the algorithm
- analysis of the stability of the resulting system
- implementation of the algorithm in the final system
- testing of the final system

It is not the purpose of this text to provide a comprehensive introduction to the Z-transform, but rather to highlight its key points and how the algorithm can be implemented in hardware within a CPLD or FPGA.

Whether digital filtering or digital control is required, a typical system for undertaking DSP tasks is shown in Figure 7.2. Here, the digital system accepts an analogue input and outputs an analogue response. This is undertaken on one or more inputs and creates one or more outputs. In the view shown in Figure 7.2,

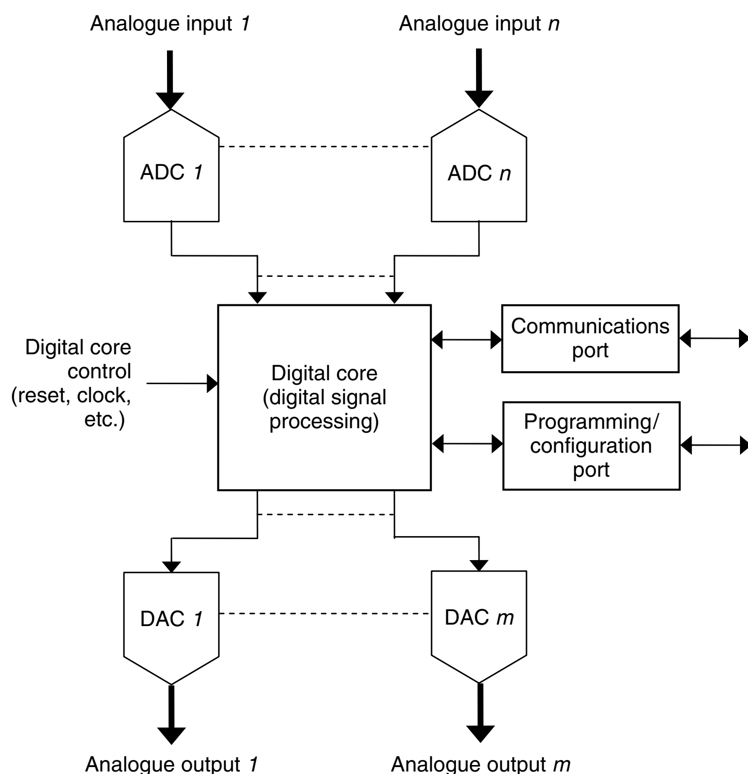
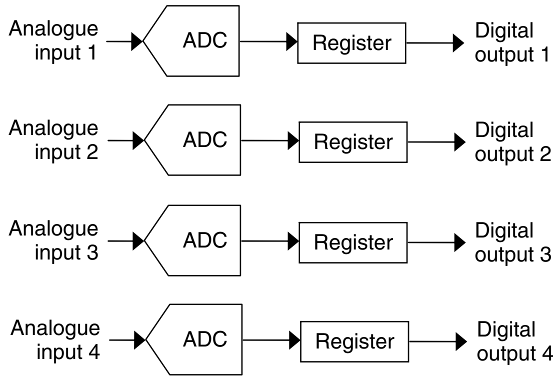


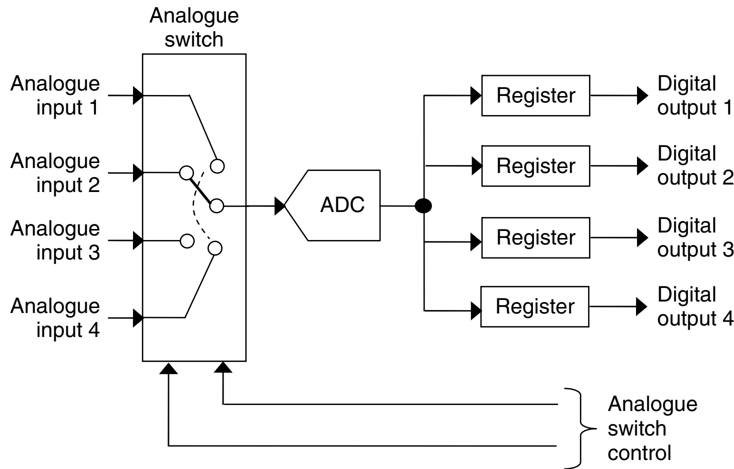
Figure 7.2: Generic digital signal processing arrangement (with analogue I/O)

a DSP core contains the algorithm to implement in addition to a control unit that creates the necessary control signals for ADC control, DAC control, communications port control, and the correct operation of the algorithm. Also shown is a programming/configuration port used to upload a software program (processor-based system) or a hardware configuration (FPGA- or CPLD-based system).

An alternative to using multiple ADCs to sample the analogue input is to use a single ADC, then switch the different analogue inputs to the ADC in turn. The system that utilizes individual ADCs for each analogue input has the capability to sample all analogue inputs in parallel. A system that uses a single switched ADC must sample each input in series (one after another). A parallel ADC arrangement provides for a short sampling period (compared to the serial arrangement, whose signal sampling period



(a) Parallel sampling of analogue inputs



(b) Serial sampling of analogue inputs

Figure 7.3: Parallel or serial sampling of an analogue input

equals the time taken to sample one analogue input multiplied by the number of inputs). However, the need for a parallel or serial arrangement depends on the system requirements and the signal sampling period required. Figure 7.3 shows this idea for a system with four analogue inputs and each digital output is stored in a register.

The choice of ADC architecture determines the number of control pins required by the ADCs and DACs and the conversion time (A/D and D/A). The choice of

digital code (e.g., unsigned straight binary or 2s complement signed binary) influences the amount of digital signal encoding and decoding required within the digital core.

It should now be noted that integral to the design of these circuits but not shown here are anti-aliasing filters at the system input (analogue input) to remove any high-frequency signals that would cause aliasing problems with the sampled data.

Example 1: Single-Input, Single-Output DSP Top-Level Description

The basic design architecture shown in Figure 7.2 can be coded in VHDL for a particular design requirement. Consider a custom digital signal processor design that is to sample a single analogue input via an eight-bit ADC, undertake a particular digital signal processing algorithm, and produce an analogue output via an eight-bit DAC. The digital design is to be implemented in hardware using a CPLD or FPGA. The timing of the digital design is to be controlled by a digital input master clock and an active low asynchronous reset. The basic architecture for this design is shown in Figure 7.4. Here, the DSP core:

- uses the AD7575 eight-bit LC^{2}MOS successive approximation ADC [8]
- uses the AD7524 eight-bit buffered multiplying DAC [9]
- incorporates a simple UART (universal asynchronous receiver transmitter) for communications between the DSP core and an external digital system, using only the *Tx* (transmit) and *Rx* (receive) serial data connections

The digital core contains the algorithm to implement, the necessary control unit that will create the ADC and DAC control signals, and the UART control and data signals. The data to pass to the UART transmitter and the data (or commands) to be received from the UART receiver are specified in the design requirements. The UART has a DR (data received) output used to inform the control unit that a byte has been received from the external digital system and a Transmit input that is used to instruct the UART to transmit a byte of data.

The set-up is shown in Figure 7.5. Here, a CPLD implements the digital actions and interfaces directly with the ADC and DAC. All devices are considered to operate on the same power supply voltage (e.g., +3.3 V) and use the same I/O standards. A suitable clock frequency must be chosen to ensure that all operations can be

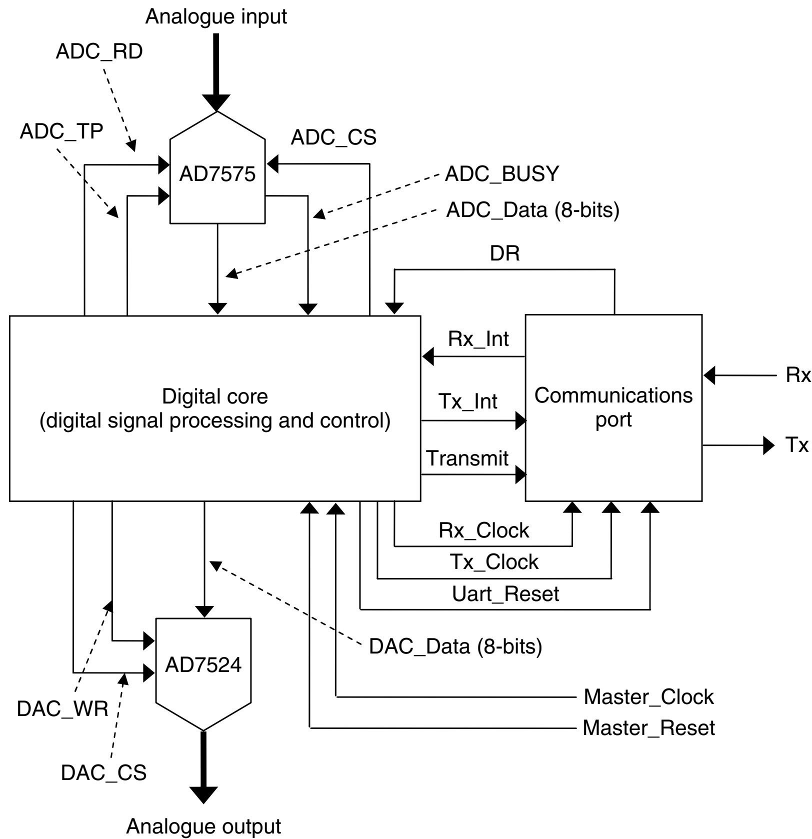


Figure 7.4: Custom DSP core architecture

undertaken within the CPLD (or FPGA) in the required time. The CPLD interfaces with an external system (here a PC) via the RS-232C interface. To enable this, the voltage levels created and accepted by the CPLD must be level-shifted to those required by the RS-232C standard.

The top-level design for the digital circuitry to be configured into the CPLD (or FPGA) can be coded in VHDL. The VHDL structural code (the name of the top-level design here is *top*) is shown in Figure 7.6. Here, the core within the CPLD or FPGA contains two main functional blocks: the first contains the digital core (Dsp_Core), and the second contains the UART (Uart).

The I/O pins for the design are detailed in Table 7.1.

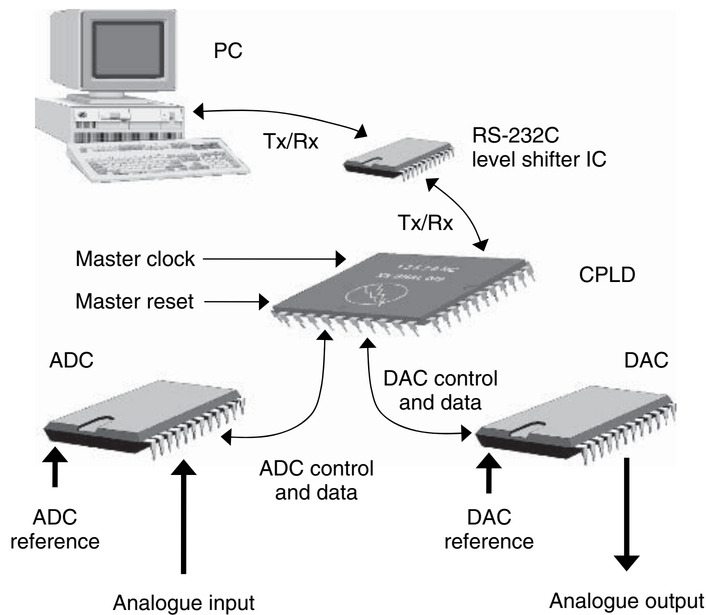


Figure 7.5: System set-up

The internal signals used within the design are detailed in Table 7.2.

The basic operation of the digital system is shown in the flow chart in Figure 7.7. At the start of the circuit operation, the circuit is in a reset state. It then follows a repetitive sequence—sample the analogue input, run the digital algorithm, and update the analogue output—that continues until the circuit is reset back to the reset state.

An example DSP core structure for this design is shown in Figure 7.8. The algorithm, control unit, and I/O register functions are placed in separate blocks. The VHDL code for this structure is shown in Figure 7.9, where the control unit is designed to create four control signals (algorithm control (3:0)) to control the movement and storage of data through the algorithm block. There will be as many control signals as required for the particular algorithm.

An example UART structure for this design is shown in Figure 7.10. The receiver and transmitter functions are placed in separate blocks. The VHDL code for this structure is shown in Figure 7.11.

```
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 Top IS
8       PORT ( ADC_BUSY     : IN   STD_LOGIC;
9              ADC_TP       : OUT  STD_LOGIC;
10             ADC_RD       : OUT  STD_LOGIC;
11             ADC_CS       : OUT  STD_LOGIC;
12             ADC_Data     : IN   STD_LOGIC_VECTOR (7 downto 0);
13             DAC_WR       : OUT  STD_LOGIC;
14             DAC_CS       : OUT  STD_LOGIC;
15             DAC_Data     : OUT  STD_LOGIC_VECTOR (7 downto 0);
16             Master_Clock : IN   STD_LOGIC;
17             Master_Reset : IN   STD_LOGIC;
18             Rx           : IN   STD_LOGIC;
19             Tx           : OUT  STD_LOGIC);
20  END ENTITY Top;
21
22
23  ARCHITECTURE Structural OF Top IS
24
25  SIGNAL Tx_Int      :  STD_LOGIC_VECTOR (7 downto 0);
26  SIGNAL Tx_Clock    :  STD_LOGIC;
27  SIGNAL Rx_Int      :  STD_LOGIC_VECTOR (7 downto 0);
28  SIGNAL Rx_Clock    :  STD_LOGIC;
29  SIGNAL Uart_Reset  :  STD_LOGIC;
30  SIGNAL DR          :  STD_LOGIC;
31  SIGNAL Transmit    :  STD_LOGIC;
32
33
34  COMPONENT Dsp_Core IS
35      PORT ( ADC_BUSY     : IN   STD_LOGIC;
36             ADC_TP       : OUT  STD_LOGIC;
37             ADC_RD       : OUT  STD_LOGIC;
38             ADC_CS       : OUT  STD_LOGIC;
39             ADC_Data     : IN   STD_LOGIC_VECTOR (7 downto 0);
40             DAC_WR       : OUT  STD_LOGIC;
41             DAC_CS       : OUT  STD_LOGIC;
42             DAC_Data     : OUT  STD_LOGIC_VECTOR (7 downto 0);
43             Master_Clock : IN   STD_LOGIC;
44             Master_Reset : IN   STD_LOGIC;
45             Rx           : IN   STD_LOGIC_VECTOR (7 downto 0);
46             Rx_Clock     : OUT  STD_LOGIC;
47             Tx           : OUT  STD_LOGIC_VECTOR (7 downto 0);
48             Tx_Clock     : OUT  STD_LOGIC;
49             Uart_Reset   : OUT  STD_LOGIC;
```

Figure 7.6: Top-level structural VHDL code

```
50            DR            : IN   STD_LOGIC;
51            Transmit      : OUT  STD_LOGIC);
52   END COMPONENT Dsp_Core;
53
54   COMPONENT Uart IS
55       PORT (  Uart_Reset   : IN   STD_LOGIC;
56               Rx_Clock     : IN   STD_LOGIC;
57               Tx_Clock     : IN   STD_LOGIC;
58               Rx_Int       : OUT  STD_LOGIC_VECTOR (7 downto 0);
59               Tx_Int       : IN   STD_LOGIC_VECTOR (7 downto 0);
60               Rx           : IN   STD_LOGIC;
61               Tx           : OUT  STD_LOGIC;
62               DR           : OUT  STD_LOGIC;
63               Transmit     : IN   STD_LOGIC);
64   END COMPONENT Uart;
65
66
67   BEGIN
68
69    I1 : Dsp_Core
70          PORT MAP( ADC_BUSY          =>  ADC_BUSY,
71                    ADC_TP            =>  ADC_TP,
72                    ADC_RD            =>  ADC_RD,
73                    ADC_CS            =>  ADC_CS,
74                    ADC_Data          =>  ADC_Data,
75                    DAC_WR            =>  DAC_WR,
76                    DAC_CS            =>  DAC_CS,
77                    DAC_Data          =>  DAC_Data,
78                    Master_Clock      =>  Master_Clock,
79                    Master_Reset      =>  Master_Reset,
80                    Rx                =>  Rx_Int,
81                    Rx_Clock          =>  Rx_Clock,
82                    Tx                =>  Tx_Int,
83                    Tx_Clock          =>  Tx_Clock,
84                    Uart_Reset        =>  Uart_Reset,
85                    DR                =>  DR,
86                    Transmit          =>  Transmit);
87
88    I2 : Uart
89         PORT MAP( Uart_Reset        =>  Uart_Reset,
90                   Rx_Clock          =>  Rx_Clock,
91                   Tx_Clock          =>  Tx_Clock,
92                   Rx_Int            =>  Rx_Int,
93                   Tx_Int            =>  Tx_Int,
94                   Rx                =>  Rx,
95                   Tx                =>  Tx,
96                   DR                =>  DR,
97                   Transmit          =>  Transmit);
98
99    END ARCHITECTURE Structural;
```

Figure 7.6: (Continued)

Table 7.1: Example I/O pins

Pin name	Direction	Purpose
ADC_BUSY	Input	ADC converts analogue input to digital
ADC_TP	Output	Connect to logic 1 in application (test use only)
ADC_RD	Output	ADC read (active low)
ADC_CS	Output	ADC chip select (active low)
ADC_Data	Input	8-bit data from ADC
DAC_WR	Output	DAC write (active low)
DAC_CS	Output	DAC chip select (active low)
DAC_Data	Output	8-bit data to DAC
Master_Clock	Input	Clock input
Master_Reset	Input	Reset control input (active low asynchronous reset)
Rx	Input	Serial data input to UART
Tx	Output	Serial data output from UART

Table 7.2: Example internal signals

Signal name	Purpose
Tx_Int	8-bit data (byte) to send out via the UART
Tx_Clock	UART transmitter clock (x16 baud rate)
Rx_Int	8-bit data (byte) received from the UART
Rx_Clock	UART receiver clock (x16 baud rate)
Uart_Reset	Reset control input (active low asynchronous reset)
DR	Byte of data received on UART Rx input
Transmit	Control signal to initiate the transmission of a byte of data on the UART Tx output

Example 2: Switched Analogue Input

Consider now a circuit that accepts two analogue inputs and produces a single analogue output. The basic architecture for this design is shown in Figure 7.12 where the DSP core:

- uses the AD7575 eight-bit LC^2MOS successive approximation ADC [8]
- uses the AD7524 eight-bit buffered multiplying DAC [9]
- incorporates a simple UART for communications between the DSP core and an external digital system, with just the Tx (transmit) and Rx (receive) serial data connections used

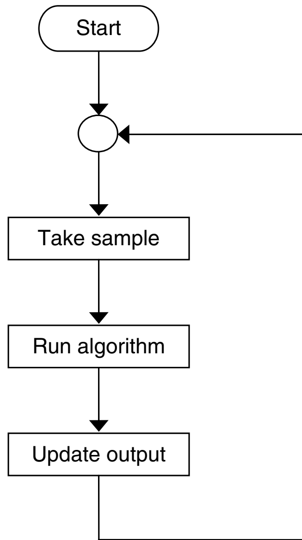
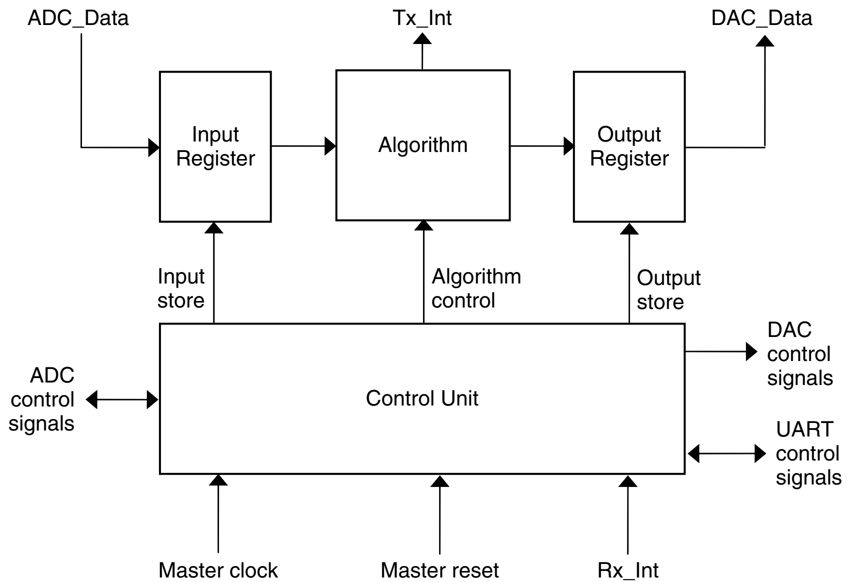


Figure 7.7: Overview of core operation (flow chart)



Note: All blocks have a common master reset input.

Figure 7.8: Example DSP core structure

```
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 Dsp_Core IS
8       PORT ( ADC_BUSY      :  IN   STD_LOGIC;
9              ADC_TP        :  OUT  STD_LOGIC;
10             ADC_RD        :  OUT  STD_LOGIC;
11             ADC_CS        :  OUT  STD_LOGIC;
12             ADC_Data      :  IN   STD_LOGIC_VECTOR (7 downto 0);
13             DAC_WR        :  OUT  STD_LOGIC;
14             DAC_CS        :  OUT  STD_LOGIC;
15             DAC_Data      :  OUT  STD_LOGIC_VECTOR (7 downto 0);
16             Master_Clock  :  IN   STD_LOGIC;
17             Master_Reset  :  IN   STD_LOGIC;
18             Rx            :  IN   STD_LOGIC_VECTOR (7 downto 0);
19             Rx_Clock      :  OUT  STD_LOGIC;
20             Tx            :  OUT  STD_LOGIC_VECTOR (7 downto 0);
21             Tx_Clock      :  OUT  STD_LOGIC;
22             Uart_Reset    :  OUT  STD_LOGIC;
23             DR            :  IN   STD_LOGIC;
24             Transmit      :  OUT  STD_LOGIC);
25  END ENTITY Dsp_Core;
26
27
28  ARCHITECTURE Structural OF Dsp_Core IS
29
30
31  SIGNAL    ADC_Data_Int          : STD_LOGIC_VECTOR (7 downto 0);
32  SIGNAL    DAC_Data_Int          : STD_LOGIC_VECTOR (7 downto 0);
33  SIGNAL    Algorithm_Control     : STD_LOGIC_VECTOR (3 downto 0);
34  SIGNAL    Input_Store           : STD_LOGIC;
35  SIGNAL    Output_Store          : STD_LOGIC;
36
37
38  COMPONENT Algorithm IS
39       PORT ( ADC_Data_In         : IN   STD_LOGIC_VECTOR(7 downto 0);
40              Reset               : IN   STD_LOGIC;
41              Algorithm_Control   : IN   STD_LOGIC_VECTOR(3 downto 0);
42              Tx                  : OUT  STD_LOGIC_VECTOR(7 downto 0);
43              DAC_Data_Out        : OUT  STD_LOGIC_VECTOR(7 downto 0));
44  END COMPONENT  Algorithm;
45
46
47  COMPONENT Register_8_Bit IS
48       PORT ( Store               : IN  STD_LOGIC;
49              Reset               : IN  STD_LOGIC;
50              Data_In             : IN  STD_LOGIC_VECTOR(7 downto 0);
51              Data_Out            : OUT STD_LOGIC_VECTOR(7 downto 0));
52  END COMPONENT Register_8_Bit;
53
54
55  COMPONENT Control_Unit IS
56       PORT ( Master_Clock        : IN   STD_LOGIC;
57              Master_Reset        : IN   STD_LOGIC;
```

Figure 7.9: Example DSP core structure VHDL code

```
58              Rx                   : IN   STD_LOGIC_VECTOR(7 downto 0);
59              Uart_Reset           : OUT  STD_LOGIC;
60              Rx_Clock             : OUT  STD_LOGIC;
61              Tx_Clock             : OUT  STD_LOGIC;
62              Transmit             : OUT  STD_LOGIC;
63              DR                   : IN   STD_LOGIC;
64              ADC_BUSY             : IN   STD_LOGIC;
65              ADC_TP               : OUT  STD_LOGIC;
66              ADC_RD               : OUT  STD_LOGIC;
67              ADC_CS               : OUT  STD_LOGIC;
68              DAC_WR               : OUT  STD_LOGIC;
69              DAC_CS               : OUT  STD_LOGIC;
70              Input_Store          : OUT  STD_LOGIC;
71              Output_Store         : OUT  STD_LOGIC);
72  END COMPONENT Control_Unit;
73
74
75  BEGIN
76
77
78  I_Algorithm : Algorithm
79       PORT MAP(  ADC_Data_In         =>   ADC_Data_Int,
80                  Reset               =>   Master_Reset,
81                  Algorithm_Control   =>   Algorithm_Control,
82                  Tx                  =>   Tx,
83                  DAC_Data_Out        =>   DAC_Data_Int);
84
85
86  I_ControlUnit : Control_Unit
87       PORT MAP ( Master_Clock        =>   Master_Clock,
88                  Master_Reset        =>   Master_Reset,
89                  Rx                  =>   Rx,
90                  Uart_Reset          =>   Uart_Reset,
91                  Rx_Clock            =>   Rx_Clock,
92                  Tx_Clock            =>   Tx_Clock,
93                  Transmit            =>   Transmit,
94                  DR                  =>   DR,
95                  ADC_BUSY            =>   ADC_BUSY,
96                  ADC_TP              =>   ADC_TP,
97                  ADC_RD              =>   ADC_RD,
98                  ADC_CS              =>   ADC_CS,
99                  DAC_WR              =>   DAC_WR,
100                 DAC_CS              =>   DAC_CS,
101                 Input_Store         =>   Input_Store,
102                 Output_Store        =>   Output_Store);
103
104
105 Input_Register : Register_8_Bit
106      PORT MAP ( Store               =>   Input_Store,
107                 Reset               =>   Master_Reset,
108                 Data_In             =>   DAC_Data_Int,
109                 Data_Out            =>   DAC_Data);
110
111 Outut_Register : Register_8_Bit
112      PORT MAP ( Store               =>   Output_Store,
113                 Reset               =>   Master_Reset,
114                 Data_In             =>   DAC_Data_Int,
115                 Data_Out            =>   DAC_Data);
116
117
118 END ARCHITECTURE Structural;
```

Figure 7.9: (Continued)

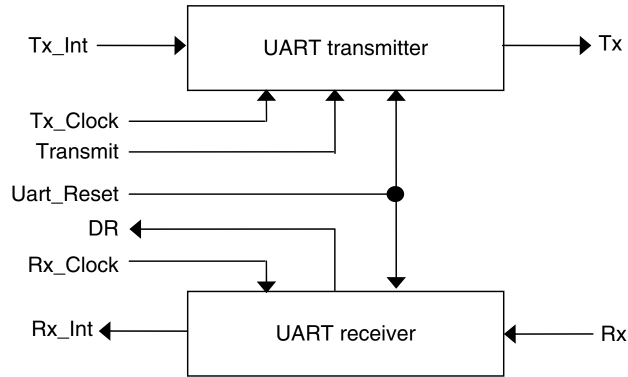


Figure 7.10: Example UART structure

```
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 Uart IS
8       PORT (  Uart_Reset    : IN    STD_LOGIC;
9               Rx_Clock      : IN    STD_LOGIC;
10              Tx_Clock      : IN    STD_LOGIC;
11              Rx_Int        : OUT   STD_LOGIC_VECTOR (7 downto 0);
12              Tx_Int        : IN    STD_LOGIC_VECTOR (7 downto 0);
13              Rx            : IN    STD_LOGIC;
14              Tx            : OUT   STD_LOGIC;
15              DR            : OUT   STD_LOGIC;
16              Transmit      : IN    STD_LOGIC);
17  END ENTITY Uart;
18
19
20  ARCHITECTURE Structural OF Uart IS
21
22
23  COMPONENT Transmitter IS
24       PORT ( Tx_Clock     : IN    STD_LOGIC;
25              Reset        : IN    STD_LOGIC;
```

Figure 7.11: Example UART structure VHDL code

```
26              Transmit    : IN   STD_LOGIC;
27              Tx_Int      : IN   STD_LOGIC_VECTOR(7 downto 0);
28              Tx          : OUT  STD_LOGIC);
29  END COMPONENT Transmitter;
30
31
32  COMPONENT Receiver IS
33       PORT ( Rx_Clock    : IN   STD_LOGIC;
34              Reset       : IN   STD_LOGIC;
35              Rx          : IN   STD_LOGIC;
36              DR          : OUT  STD_LOGIC;
37              Rx_Int      : OUT  STD_LOGIC_VECTOR(7 downto 0));
38  END COMPONENT Receiver;
39
40
41  BEGIN
42
43  I1: Transmitter
44           PORT MAP ( Tx_Clock =>  Tx_Clock,
45                      Reset    =>  Uart_Reset,
46                      Transmit =>  Transmit,
47                      Tx_Int   =>  Tx_Int,
48                      Tx       =>  Tx);
49
50  I2 : Receiver
51        PORT MAP ( Rx_Clock =>  Rx_Clock,
52                   Reset    =>  Uart_Reset,
53                   Rx       =>  Rx,
54                   DR       =>  DR,
55                   Rx_Int   =>  Rx_Int);
56
57  END ARCHITECTURE Structural;
```

Figure 7.11: (Continued)

The design is basically the same as that described in Example 1, plus an additional output (Input_Select) from the control unit that selects the analogue input using the analogue switch such that:

- When Input_Select = 0, then analogue input 1 is selected.
- When Input_Select = 1, then analogue input 2 is selected.

The basic operation of the digital system is shown in the flowchart in Figure 7.13. At the start of the circuit operation, the circuit is in a reset state. It then

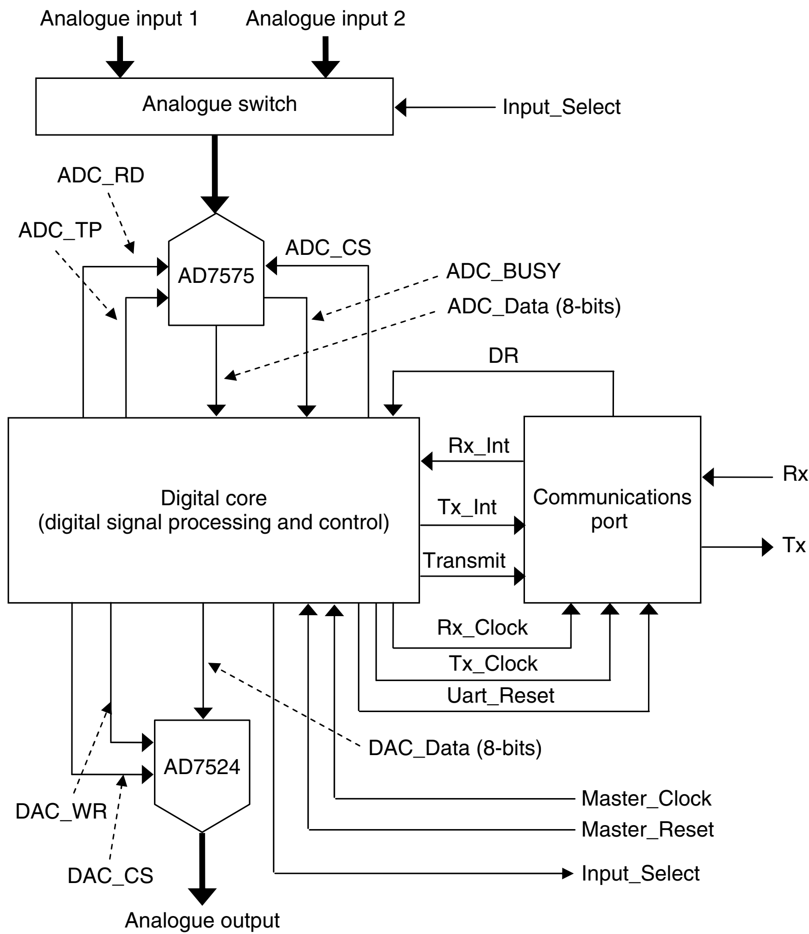


Figure 7.12: Custom DSP core architecture

follows a repetitive sequence—sample both analogue inputs, run the digital algorithm, and update the analogue output—until the circuit is reset back to the reset state.

The top-level design for the digital circuitry to be configured into the CPLD (or FPGA) can be coded in VHDL. The VHDL structural code (the name of the top-level

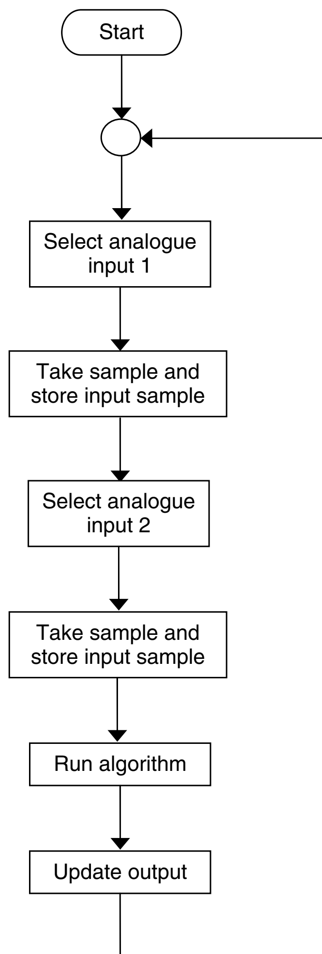


Figure 7.13: Overview of core operation (flowchart)

design here is *Top*) is shown in Figure 7.14. Here, the core within the CPLD or FPGA contains two main functional blocks: the first contains the digital core (Dsp_Core), and the second contains the UART (Uart).

The I/O pins for the design are detailed in Table 7.3.

```
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 Top IS
8       PORT ( ADC_BUSY     : IN   STD_LOGIC;
9              ADC_TP       : OUT  STD_LOGIC;
10             ADC_RD       : OUT  STD_LOGIC;
11             ADC_CS       : OUT  STD_LOGIC;
12             ADC_Data     : IN   STD_LOGIC_VECTOR (7 downto 0);
13             DAC_WR       : OUT  STD_LOGIC;
14             DAC_CS       : OUT  STD_LOGIC;
15             DAC_Data     : OUT  STD_LOGIC_VECTOR (7 downto 0);
16             Master_Clock : IN   STD_LOGIC;
17             Master_Reset : IN   STD_LOGIC;
18             Rx           : IN   STD_LOGIC;
19             Tx           : OUT  STD_LOGIC;
20             Input_Select : OUT  STD_LOGIC);
21  END ENTITY Top;
22
23
24  ARCHITECTURE Structural OF Top IS
25
26  SIGNAL Tx_Int      :  STD_LOGIC_VECTOR (7 downto 0);
27  SIGNAL Tx_Clock    :  STD_LOGIC;
28  SIGNAL Rx_Int      :  STD_LOGIC_VECTOR (7 downto 0);
29  SIGNAL Rx_Clock    :  STD_LOGIC;
30  SIGNAL Uart_Reset  :  STD_LOGIC;
31  SIGNAL DR          :  STD_LOGIC;
32  SIGNAL Transmit    :  STD_LOGIC;
33
34
35  COMPONENT Dsp_Core IS
36      PORT ( ADC_BUSY     : IN   STD_LOGIC;
37             ADC_TP       : OUT  STD_LOGIC;
38             ADC_RD       : OUT  STD_LOGIC;
39             ADC_CS       : OUT  STD_LOGIC;
40             ADC_Data     : IN   STD_LOGIC_VECTOR (7 downto 0);
41             DAC_WR       : OUT  STD_LOGIC;
42             DAC_CS       : OUT  STD_LOGIC;
43             DAC_Data     : OUT  STD_LOGIC_VECTOR (7 downto 0);
44             Master_Clock : IN   STD_LOGIC;
45             Master_Reset : IN   STD_LOGIC;
46             Rx           : IN   STD_LOGIC_VECTOR (7 downto 0);
47             Rx_Clock     : OUT  STD_LOGIC;
48             Tx           : OUT  STD_LOGIC_VECTOR (7 downto 0);
49             Tx_Clock     : OUT  STD_LOGIC;
50             Uart_Reset   : OUT  STD_LOGIC;
51             DR           : IN   STD_LOGIC;
52             Transmit     : OUT  STD_LOGIC;
53             Input_Select : OUT  STD_LOGIC);
```

Figure 7.14: Top-level structural VHDL code

```
54   END COMPONENT Dsp_Core;
55
56   COMPONENT Uart IS
57       PORT (  Uart_Reset    : IN   STD_LOGIC;
58               Rx_Clock      : IN   STD_LOGIC;
59               Tx_Clock      : IN   STD_LOGIC;
60               Rx_Int        : OUT  STD_LOGIC_VECTOR (7 downto 0);
61               Tx_Int        : IN   STD_LOGIC_VECTOR (7 downto 0);
62               Rx            : IN   STD_LOGIC;
63               Tx            : OUT  STD_LOGIC;
64               DR            : OUT  STD_LOGIC;
65               Transmit      : IN   STD_LOGIC);
66   END COMPONENT Uart;
67
68
69   BEGIN
70
71   I1 : Dsp_Core
72        PORT MAP( ADC_BUSY          =>  ADC_BUSY,
73                  ADC_TP            =>  ADC_TP,
74                  ADC_RD            =>  ADC_RD,
75                  ADC_CS            =>  ADC_CS,
76                  ADC_Data          =>  ADC_Data,
77                  DAC_WR            =>  DAC_WR,
78                  DAC_CS            =>  DAC_CS,
79                  DAC_Data          =>  DAC_Data,
80                  Master_Clock      =>  Master_Clock,
81                  Master_Reset      =>  Master_Reset,
82                  Rx                =>  Rx_Int,
83                  Rx_Clock          =>  Rx_Clock,
84                  Tx                =>  Tx_Int,
85                  Tx_Clock          =>  Tx_Clock,
86                  Uart_Reset        =>  Uart_Reset,
87                  DR                =>  DR,
88                  Transmit          =>  Transmit,
89                  Input_Select      =>  Input_Select);
90
91     I2 : Uart
92          PORT MAP( Uart_Reset        =>  Uart_Reset,
93                    Rx_Clock          =>  Rx_Clock,
94                    Tx_Clock          =>  Tx_Clock,
95                    Rx_Int            =>  Rx_Int,
96                    Tx_Int            =>  Tx_Int,
97                    Rx                =>  Rx,
98                    Tx                =>  Tx,
99                    DR                =>  DR,
100                   Transmit          =>  Transmit);
101
102    END ARCHITECTURE Structural;
```

Figure 7.14: (Continued)

Table 7.3: Example I/O pins

Pin name	Direction	Purpose
ADC_BUSY	Input	ADC converts analogue input to digital
ADC_TP	Output	Connect to logic 1 in application (test use only)
ADC_RD	Output	ADC read (active low)
ADC_CS	Output	ADC chip select (active low)
ADC_Data	Input	8-bit data from ADC
DAC_WR	Output	DAC write (active low)
DAC_CS	Output	DAC chip select (active low)
DAC_Data	Output	8-bit data to DAC
Master_Clock	Input	Clock input
Master_Reset	Input	Reset control input (active low asynchronous reset)
Rx	Input	Serial data input to UART
Tx	Output	Serial data output from UART
Input_Select	Output	Analogue switch control (0 = analogue input 1 selected, 1 = analogue input 2 selected)

7.2 Z-Transform

The Z-transform is used in the design and analysis of sampled data systems to describe the properties of a sampled data signal and/or a system. It is used in all aspects of digital signal processing as a way to:

- describe the properties of a sampled data signal and/or a system
- transform a continuous time system described using Laplace transforms into a discrete time equivalent
- mathematically analyze the signal and/or system
- view a sampled data signal and/or a system graphically as a block diagram

The Laplace transform is used in continuous time systems to describe a transfer function (the system input-output relationship) with a set of poles and zeros. A continuous time transfer function of a system is represented by the equation:

$$\frac{Y(s)}{X(s)} = G(s) = \frac{N(s)}{D(s)}$$

where:

Y(s) is the output signal from the system

X(s) is the input signal to the system

G(s) is the system transfer function

N(s) is the numerator of the equation

D(s) is the denominator of the equation

This equation is then expanded to become:

$$\frac{Y(s)}{X(s)} = \frac{b_0 + b_1 s + b_2 s^2 + \ldots + b_m . s^m}{a_0 + a_1 s + a_2 s^2 + \ldots + a_n . s^n}$$

The poles of the characteristic equation can be found by solving the denominator for:

$$D(s) = 0$$

The zeros of the characteristic equation can be found by solving the denominator for:

$$N(s) = 0$$

Analysis of the poles and zeros determines the performance of the system in both the time and frequency domains. These poles and zeros are complex numbers composed of real (Re(s)) and imaginary (Im(s)) parts. For a system to be stable, the poles of the system must lie to the left of the imaginary axis on the graph of the real and imaginary parts (the Argand diagram), as shown in Figure 7.15. Any pole to the right of the axis indicates an unstable system. A pole that appears on the imaginary axis corresponds to a marginally stable system. The available analysis techniques are described in many DSP, digital filter design, and digital control texts, so they will not be covered further in this text.

The Z-transform is used in discrete time systems to create a discrete time transfer function of the system with a set of poles and zeros. It is a formal transformation for

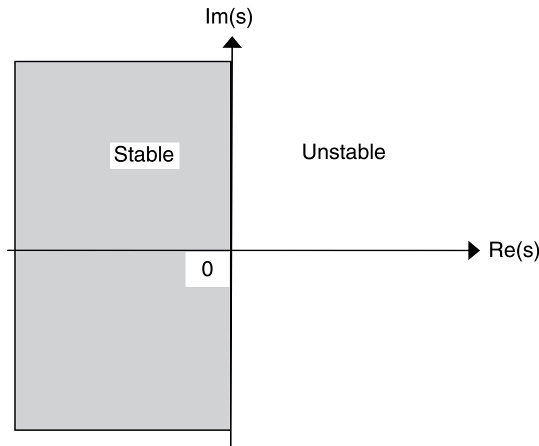


Figure 7.15: Argand diagram to analyze the stability of a continuous-time system

discrete time signals (signals described in terms of their samples) to a new complex variable called z. For a discrete time signal x(n), then:

$$x(n) = x(0), x(1), x(2), \ldots, \text{etc.}$$

Parentheses indicate the signal sample number. The Z-transform for this is written as an infinite power series in terms of the complex variable z as:

$$Z\{x(n)\} = x(0) + x(1)z^{-1} + x(2)z^{-2} + \ldots$$

This could be also written as:

$$Z\{x(n)\} = X(z) = \sum x(n)z^{-1}$$

The pulse transfer function of a system is now defined as the Z-transform of the output divided by the Z-transform of the input and is written as:

$$G(z) = \frac{Z\{y(n)\}}{Z\{x(n)\}} = \frac{Y(z)}{X(z)}$$

where:

- Y(z), is the output signal from the system
- X(z), is the input signal to the system
- G(z), is the pulse transfer function

for a general discrete time transfer function written as:

$$G(z) = \frac{Y(z)}{X(z)} = \frac{N(z)}{D(z)}$$

where:

- Y(z), is the output signal from the system
- X(z), is the input signal to the system
- G(z), is the system transfer function
- N(z), is the numerator of the general discrete time transfer function
- D(z), is the denominator of the general discrete time transfer function

This is then expanded to become:

$$\frac{Y(z)}{X(z)} = \frac{b_0 + b_1 z + b_2 z^2 + ... + b_m.z^m}{a_0 + a_1 z + a_2 z^2 + ... + a_n.z^n}$$

The poles of the characteristic equation can be found by solving the denominator for:

$$D(z) = 0$$

The zeros of the characteristic equation can be found by solving the denominator for:

$$N(z) = 0$$

Analysis of the poles and zeros determines the performance of the system in both the time and frequency domains. These poles and zeros are complex numbers

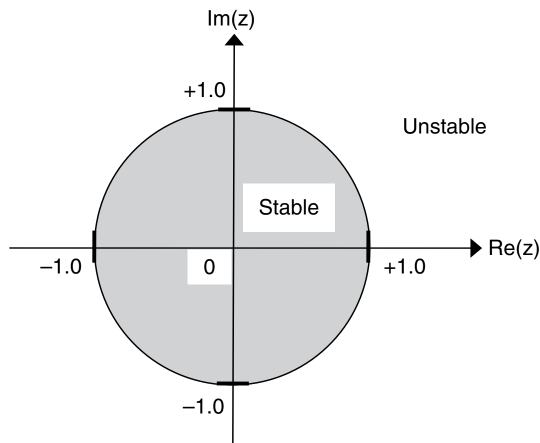


Figure 7.16: Argand diagram showing the unit circle to analyze the stability of a discrete-time system

composed of real (Re(z)) and imaginary (Im(z)) parts. For a system to be stable, the poles of the system must lie within the unit circle on the graph of the real and imaginary parts (the Argand diagram), as shown in Figure 7.16. Any pole outside the unit circle indicates an unstable system. A pole that appears on the unit circle corresponds to a marginally stable system. The available analysis techniques are described in many DSP, digital filter design, and digital control texts, so they will not be covered further in this text.

Comparing systems defined using the Laplace transform and the Z-transform, a continuous time system with a pole at s will have the same dynamic characteristics as a discrete time system with a pole at:

$$z = e^{sT}$$

Here, T is the sampling period of the signal sampling. This allows a discrete-time system to be designed initially as a continuous-time system, then to be translated to a discrete-time implementation. The discrete-time implementation uses signal samples (the current sample and delayed [previous] samples).

However, care must be taken in the implementation of the discrete-time system to account for implementation limitations and for the effect of frequency warping, which occurs when an analogue prototype system is translated to a discrete-time implementation. These aspects are discussed in the next section, on digital control.

The effect of delaying a signal by n samples is to multiply its Z-transform by z^{-n}. This effect is used to implement a discrete-time transfer function either in software or in hardware by sampling and delaying signals. A delay by one sample (Z^{-1}) is shown in Figure 7.17,

where:

$$(\text{Data Output}(z)) = (\text{Data input}(z))z^{-1}$$

Here, D-type flip-flops with asynchronous active low resets store the input data. The Store input is the clock input to each of the flip-flops (all flip-flops are considered to have a common clock input) provides the control for the storage of the data input.

A delay element design used to store a value and delay by one sample is a register. An eight-bit data delay element design in VHDL is shown in Figure 7.18.

Figure 7.19 provides an example VHDL test bench for the delay element.

The individual delay elements can be cascaded to provide a delay-by-m output where m is an integer number that identifies how many clock control signals are required before the input signal becomes an output.

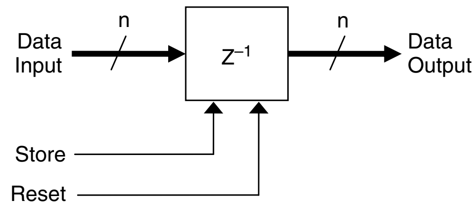


Figure 7.17: Delay element (n-bit register)

```
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   ENTITY Delay IS
7       PORT ( Data_In      : IN   STD_LOGIC_VECTOR(7 downto 0);
8              Store        : IN   STD_LOGIC;
9              Reset        : IN   STD_LOGIC;
10             Data_Out     : OUT  STD_LOGIC_VECTOR(7 downto 0));
11  END ENTITY Delay;
12
13  ARCHITECTURE Behavioural OF Delay IS
14
15  BEGIN
16
17  Store_Process: PROCESS(Store, Data_In, Reset)
18
19  BEGIN
20
21          IF (Reset = '0') THEN
22
23                  Data_Out(7 downto 0) <= "00000000";
24
25          ELSIF (Store'EVENT AND Store = '1') THEN
26
27                  Data_Out(7 downto 0) <= Data_In(7 downto 0);
28
29          END IF;
30
31  END PROCESS Store_Process;
32
33  END ARCHITECTURE Behavioural;
```

Figure 7.18: Delay element (eight-bit register)

Example 3: Delay-by-3 Circuit

To illustrate the delay-by-m circuit, consider a delay-by-3 circuit using three delay elements as shown in Figure 7.20, where:

$$\begin{aligned}
(\text{No_Delay}(z)) &= (\text{Data input}(z)) \\
(\text{Delay_By_One}(z)) &= (\text{Data input}(z)z^{-1} \\
(\text{Delay_By_Two}(z)) &= (\text{Data input}(z))z^{-2} \\
(\text{Delay_By_Three}(z)) &= (\text{Data input}(z))z^{-3}
\end{aligned}$$

```
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 Test_Delay_vhd IS
8   END Test_Delay_vhd;
9
10
11  ARCHITECTURE Behavioural OF Test_Delay_vhd IS
12
13  COMPONENT Delay
14  PORT(
15               Data_In   : IN  STD_LOGIC_VECTOR(7 downto 0);
16               Store     : IN  STD_LOGIC;
17               Reset     : IN  STD_LOGIC;
18               Data_Out  : OUT STD_LOGIC_VECTOR(7 downto 0));
19  END COMPONENT;
20
21  SIGNAL Store   :   STD_LOGIC := '0';
22  SIGNAL Reset   :   STD_LOGIC := '0';
23  SIGNAL Data_In :   STD_LOGIC_VECTOR(7 downto 0) := (others=>'0');
24
25  SIGNAL Data_Out :  STD_LOGIC_VECTOR(7 downto 0);
26
27  BEGIN
28
29  uut: Delay PORT MAP(
30               Data_In   =>  Data_In,
31               Store     =>  Store,
32               Reset     =>  Reset,
33               Data_Out  =>  Data_Out);
34
35
36  Reset_Process : PROCESS
37
38  BEGIN
39
40               Wait for 0 ns;  Reset <= '0';
41               Wait for 5 ns;  Reset <= '1';
42               Wait;
43
44  END PROCESS Reset_Process;
45
46
47  Store_Process : PROCESS
48
49  BEGIN
50
51               Wait for 0 ns;   Store  <= '0';
52               Wait for 10 ns;  Store  <= '1';
53               Wait for 10 ns;  Store  <= '0';
54
55  END PROCESS Store_Process;
56
57
58  DataIn_Process : PROCESS
59
60  BEGIN
61
62               Wait for 0 ns;   Data_In  <= "00000000";
63               Wait for 60 ns;  Data_In  <= "11111111";
64               Wait for 20 ns;  Data_In  <= "00000000";
65               Wait for 20 ns;  Data_In  <= "11111111";
66               Wait for 20 ns;  Data_In  <= "00000000";
67
68               Wait for 20 ns;
69
70  END PROCESS DataIn_Process;
71
72  END ARCHITECTURE Behavioural;
```

Figure 7.19: VHDL test bench for delay element

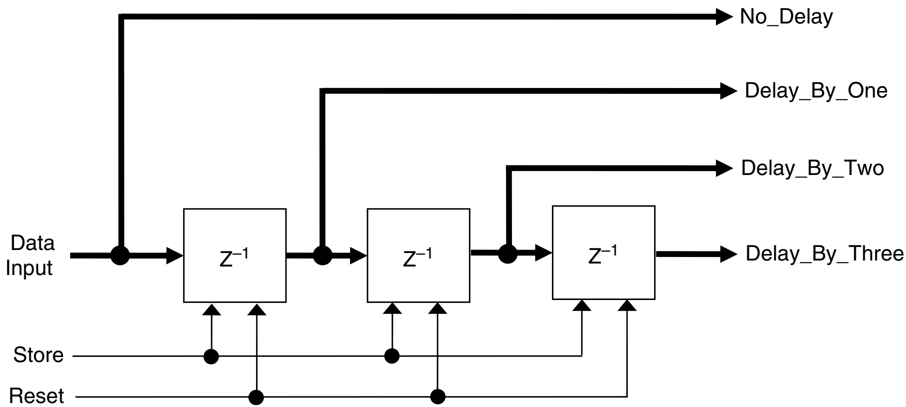


Figure 7.20: Delay-by-3 circuit schematic

Here, the input data and each of the delay element outputs is also available for monitoring signal progression through the circuit.

Such a circuit could be coded in VHDL using a dataflow, behavioral, or structural description. Figure 7.21 shows a behavioral description for this design using two processes. The first process is created to store the input signal in three eight-bit registers, the outputs of which are internal signals. The second process takes the internal signals and provides these as outputs. In the structure illustrated here, the internal signals can be read by another process within the design if this delay-by-3 circuit is modified within a larger design.

Figure 7.22 provides an example VHDL test bench for the delay-by-3 behavioral description.

Using the delay element shown in Figure 7.18, then a structural VHDL description for the delay-by-3 circuit can be created. An example of this is shown in Figure 7.23.

In this design, the outputs from the delay elements are now buffered using an eight-bit buffer (`Buffer_Cell`). The VHDL code for this buffer design is shown in Figure 7.24.

```
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 Delay_By_3_Behavioural IS
8       PORT (  Data_In          : IN   STD_LOGIC_VECTOR(7 downto 0);
9               Store            : IN   STD_LOGIC;
10              Reset            : IN   STD_LOGIC;
11              No_Delay         : OUT  STD_LOGIC_VECTOR(7 downto 0);
12              Delay_By_One     : OUT  STD_LOGIC_VECTOR(7 downto 0);
13              Delay_By_Two     : OUT  STD_LOGIC_VECTOR(7 downto 0);
14              Delay_By_Three   : OUT  STD_LOGIC_VECTOR(7 downto 0));
15  END ENTITY Delay_By_3_Behavioural;
16
17
18  ARCHITECTURE Behavioural OF Delay_By_3_Behavioural IS
19
20  SIGNAL Internal_1 : STD_LOGIC_VECTOR(7 downto 0);
21  SIGNAL Internal_2 : STD_LOGIC_VECTOR(7 downto 0);
22  SIGNAL Internal_3 : STD_LOGIC_VECTOR(7 downto 0);
23
24  BEGIN
25
26  Store_Process : PROCESS(Store, Data_In, Internal_1, Internal_2, Internal_3, Reset)
27
28  BEGIN
29
30         IF (Reset = '0') THEN
31
32                Internal_1 (7 downto 0)   <=  "00000000";
33                Internal_2 (7 downto 0)   <=  "00000000";
34                Internal_3 (7 downto 0)   <=  "00000000";
35
36         ELSIF (Store'EVENT AND Store = '1') THEN
37
38                Internal_1(7 downto 0)  <=  Data_In(7 downto 0);
39                Internal_2(7 downto 0)  <=  Internal_1(7 downto 0);
40                Internal_3(7 downto 0)  <=  Internal_2(7 downto 0);
41
42         END IF;
43
44  END PROCESS Store_Process;
45
46
47  Update_Outputs: PROCESS(Data_In, Internal_1, Internal_2, Internal_3)
48
49  BEGIN
50
51  No_Delay(7 downto 0)        <=  Data_In(7 downto 0);
52  Delay_By_One(7 downto 0)    <=  Internal_1(7 downto 0);
53  Delay_By_Two(7 downto 0)    <=  Internal_2(7 downto 0);
54  Delay_By_Three(7 downto 0)  <=  Internal_3(7 downto 0);
55
56  END PROCESS Update_Outputs;
57
58
59  END ARCHITECTURE Behavioural;
```

Figure 7.21: Delay-by-3 circuit behavioral VHDL description

```
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 Test_Delay_By_3_Behavioural_vhd IS
8   END Test_Delay_By_3_Behavioural_vhd;
9
10
11  ARCHITECTURE Behavioural OF Test_Delay_By_3_Behavioural_vhd IS
12
13  COMPONENT Delay_By_3_Behavioural
14   PORT (  Data_In         : IN   STD_LOGIC_VECTOR(7 downto 0);
15           Store           : IN   STD_LOGIC;
16           Reset           : IN   STD_LOGIC;
17           No_Delay        : OUT  STD_LOGIC_VECTOR(7 downto 0);
18           Delay_By_One    : OUT  STD_LOGIC_VECTOR(7 downto 0);
19           Delay_By_Two    : OUT  STD_LOGIC_VECTOR(7 downto 0);
20           Delay_By_Three  : OUT  STD_LOGIC_VECTOR(7 downto 0));
21  END COMPONENT;
22
23  SIGNAL Store   : STD_LOGIC:= '0';
24  SIGNAL Reset   : STD_LOGIC := '0';
25  SIGNAL Data_In : STD_LOGIC_VECTOR(7 downto 0) := (others=>'0');
26
27  SIGNAL  Data_Out        :  STD_LOGIC_VECTOR(7 downto 0);
28  SIGNAL  No_Delay        :  STD_LOGIC_VECTOR(7 downto 0);
29  SIGNAL  Delay_By_One    :  STD_LOGIC_VECTOR(7 downto 0);
30  SIGNAL  Delay_By_Two    :  STD_LOGIC_VECTOR(7 downto 0);
31  SIGNAL  Delay_By_Three  :  STD_LOGIC_VECTOR(7 downto 0);
32
33
34  BEGIN
35
36  uut: Delay_By_3_Behavioural PORT MAP(
37                Data_In        =>  Data_In,
38                Store          =>  Store,
39                Reset          =>  Reset,
40                No_Delay       =>  No_Delay,
41                Delay_By_One   =>  Delay_By_One,
42                Delay_By_Two   =>  Delay_By_Two,
43                Delay_By_Three =>  Delay_By_Three);
44
45
```

Figure 7.22: VHDL test bench for delay-by-3 circuit behavioral VHDL description

```
46   Reset_Process : PROCESS
47
48   BEGIN
49
50                 Wait for 0 ns;  Reset <= '0';
51                 Wait for 5 ns;  Reset <= '1';
52                 Wait;
53
54   END PROCESS Reset_Process;
55
56
57   Store_Process : PROCESS
58
59   BEGIN
60
61                 Wait for 0 ns;   Store  <= '0';
62                 Wait for 10 ns;  Store  <= '1';
63                 Wait for 10 ns;  Store  <= '0';
64
65   END PROCESS Store_Process;
66
67
68   DataIn_Process : PROCESS
69
70   BEGIN
71
72                 Wait for 0 ns;   Data_In  <= "00000000";
73                 Wait for 60 ns;  Data_In  <= "11111111";
74                 Wait for 20 ns;  Data_In  <= "00000000";
75                 Wait for 20 ns;  Data_In  <= "11111111";
76                 Wait for 20 ns;  Data_In  <= "00000000";
77
78         Wait for 20 ns;
79
80   END PROCESS DataIn_Process;
81
82
83   END ARCHITECTURE Behavioural;
```

Figure 7.22: (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   ENTITY Delay_By_3_Structural IS
7          PORT ( Data_In          : IN   STD_LOGIC_VECTOR(7 downto 0);
8                 Store            : IN   STD_LOGIC;
9                 Reset            : IN   STD_LOGIC;
10                No_Delay         : OUT  STD_LOGIC_VECTOR(7 downto 0);
11                Delay_By_One     : OUT  STD_LOGIC_VECTOR(7 downto 0);
12                Delay_By_Two     : OUT  STD_LOGIC_VECTOR(7 downto 0);
13                Delay_By_Three   : OUT  STD_LOGIC_VECTOR(7 downto 0));
14  END ENTITY Delay_By_3_Structural;
15
16  ARCHITECTURE Structural OF Delay_By_3_Structural IS
17
18  SIGNAL Internal_1 : STD_LOGIC_VECTOR(7 downto 0);
19  SIGNAL Internal_2 : STD_LOGIC_VECTOR(7 downto 0);
20  SIGNAL Internal_3 : STD_LOGIC_VECTOR(7 downto 0);
21
22  COMPONENT Delay IS
23      PORT ( Data_In      : IN   STD_LOGIC_VECTOR(7 downto 0);
24             Store        : IN   STD_LOGIC;
25             Reset        : IN   STD_LOGIC;
26             Data_Out     : OUT  STD_LOGIC_VECTOR(7 downto 0));
27  END COMPONENT Delay;
28
29  COMPONENT Buffer_Cell IS
30      PORT ( Data_In      : IN   STD_LOGIC_VECTOR(7 downto 0);
31             Data_Out     : OUT  STD_LOGIC_VECTOR(7 downto 0));
32  END COMPONENT Buffer_Cell;
33
34  BEGIN
35  I_Delay1 : Delay
36           PORT MAP( Data_In  => Data_In,
37                     Store    => Store,
38                     Reset    => Reset,
39                     Data_Out => Internal_1);
40
41  I_Delay2 : Delay
42           PORT MAP( Data_In  => Internal_1,
43                     Store    => Store,
44                     Reset    => Reset,
45                     Data_Out => Internal_2);
46
47  I_Delay3 : Delay
48           PORT MAP( Data_In  => Internal_2,
49                     Store    => Store,
50                     Reset    => Reset,
51                     Data_Out => Internal_3);
52
53  I_Buffer1 : Buffer_Cell
54            PORT MAP( Data_In  => Data_In,
55                      Data_Out => No_Delay);
56
57  I_Buffer2 : Buffer_Cell
58            PORT MAP( Data_In  => Internal_1,
59                      Data_Out => Delay_By_One);
60
61  I_Buffer3 : Buffer_Cell
62            PORT MAP( Data_In  => Internal_2,
63                      Data_Out => Delay_By_Two);
64
65  I_Buffer4 : Buffer_Cell
66            PORT MAP( Data_In  => Internal_3,
67                      Data_Out => Delay_By_Three);
68
69  END ARCHITECTURE Structural;
```

Figure 7.23: Delay-by-3 circuit structural VHDL description

```
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 Buffer_Cell IS
8       Port ( Data_In  : IN   STD_LOGIC_VECTOR (7 downto 0);
9              Data_Out : OUT  STD_LOGIC_VECTOR (7 downto 0));
10  END ENTITY Buffer_Cell;
11
12
13  ARCHITECTURE Behavioural OF Buffer_Cell IS
14
15  BEGIN
16
17  Buffer_Process: PROCESS(Data_In)
18
19  BEGIN
20
21          Data_Out(7 downto 0) <= Data_In(7 downto 0);
22
23  END PROCESS Buffer_Process;
24
25  END ARCHITECTURE Behavioural;
```

Figure 7.24: Eight-bit buffer VHDL description

7.3 Digital Control

A control system is composed of two subsystems, a plant and a controller. The plant is the object controlled by the controller. The plant and controller can be either analogue or digital, although digital control algorithms have become more popular because they can be quickly and cost-effectively implemented. In many cases, digital algorithms are implemented using a software program running on a suitable processor within a PC or processor-based embedded system, so the implementer need not have the skills and/or tools to design controllers in hardware on FPGAs and CPLDs. The fundamental algorithm design is however the same, whether the implementation is in hardware or software, and a hardware implementation using an FPGA or CPLD might in some situations be the preferred option. A custom digital controller in hardware has several benefits over processor-based implementation:

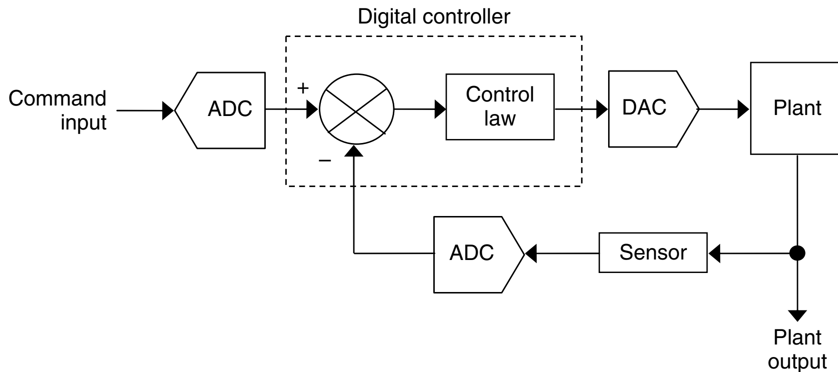


Figure 7.25: Basic computer-based control system

- Custom hardware can be optimized for the application.
- Any processor features not required in the application are not included in the design.
- A software program to run on the target hardware need not be developed.

As an example, Figure 7.25 shows a basic computer-based control system with two analogue inputs and an analogue output. The user sets the required plant output by applying a suitable command input signal. The controller responds to the command input and creates a plant control signal based on the difference between the command input and a feedback signal from the plant. The control law chosen determines how the controller and plant respond to the command input.

In Figure 7.25, then:

- This is an automatic control system in that once a user has set the command input, the system will automatically perform to the requirements of the command input (i.e., it will automatically set the plant to the value set by the command input).
- Using a digital controller, this is also referred to as direct digital control (DDC).
- The first analogue input is a DC voltage (here rather than a current), which sets the value required for the plant (the output load to be controlled). In a motor speed control system, for example, the DC voltage represents the required motor speed. This is the command input. Increasing the command

input in a positive direction increases the motor shaft speed in one direction of motor shaft rotation. Increasing the command input in a negative direction increases the motor shaft speed in the opposite direction of motor shaft rotation. A command input of zero indicates a the motor shaft speed of zero.

- The second analogue input is a feedback voltage whose value indicates the value attained by the plant. In a motor speed control system, for example, the DC feedback voltage represents the actual motor shaft speed.
- The analogue output is a signal that is applied to the plant. In a motor speed control system, for example, this is the voltage applied to the motor terminals.

This is an example of a closed-loop control system in that the feedback signal applied to the controller is subtracted from the command input to form an error signal. This error signal is applied to the control law (the algorithm to act on the current sampled input and previous sampled inputs). In general, there can be one or more inputs and one or more outputs. The plant is a continuous time plant, and the inputs to and output from the digital controller are analogue signals.

In general, this leads to the following nine possible arrangements:

1. The control system is either an open-loop system (no feedback) or a closed-loop system (feedback).
2. The command input can be either analogue or digital.
3. The feedback can be either analogue or digital.
4. The controller output can be either analogue or digital.
5. There can be one or more command inputs.
6. There can be one or more feedback signals.
7. There can be one or more plant control signals (outputs from the controller).
8. The controller can implement one or more control algorithms.
9. The digital control algorithm can be designed directly in digital, or it can be created by first creating an analogue prototype, then converting the analogue control law to a digital control law.

The digital controller (or filter) is designed to undertake the required operations using a particular circuit architecture. This architecture is chosen to enable the required

operations in the required time using the minimal amount of circuitry (or size of software program) and effectively using the available resources provided by the target technology. The architecture might use a predefined standard computer architecture or a custom architecture. A custom architecture either is based on a processor architecture, or it implements the algorithm exactly as represented by the control law or filter equation.

Standard computer architecture is based on either the Von Neumann or Harvard computer architecture, shown in Figure 7.26. In Von Neumann architecture, the data and instructions share memory and buses, meaning that both cannot be read at the same time. In some applications, this sequential access of data and instructions limits the speed of operation. The Harvard architecture separates the data and instructions storage and buses, thereby providing higher speed of operation than a Von Neumann computer architecture but at the price of increased design complexity.

The processor used within the computer architecture is based on CISC (complex instruction set computer) or RISC (reduced instruction set computer) architecture. The CISC is designed to complete a task in as few lines of processor

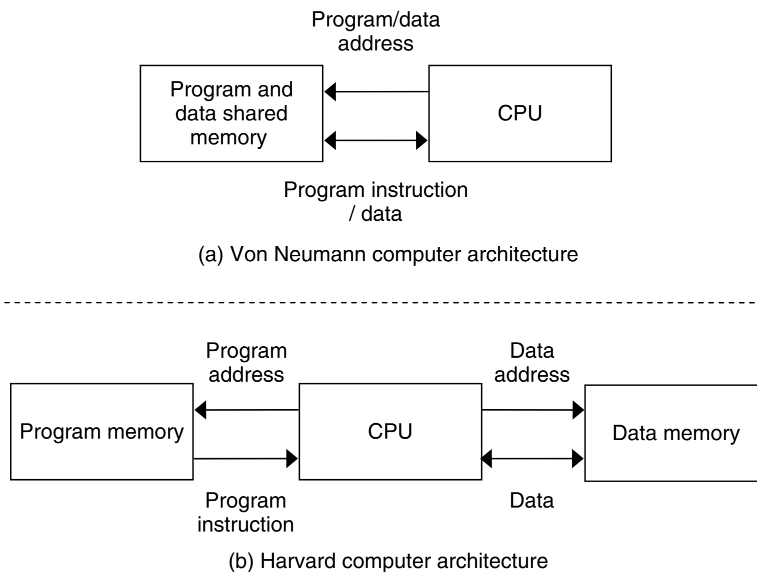


Figure 7.26: Von Neumann and Harvard computer architectures

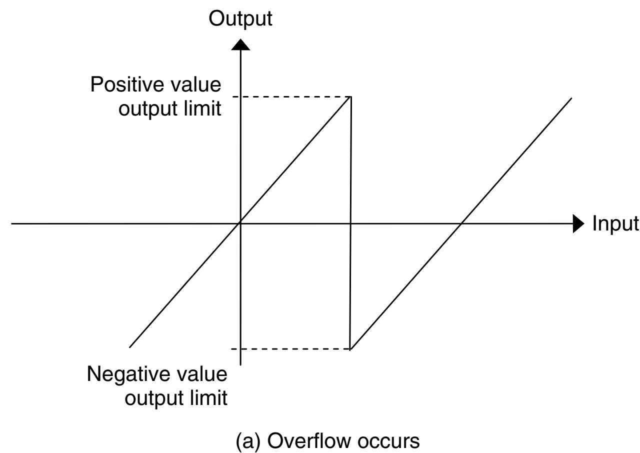
assembly code as possible, which it achieves by incorporating hardware into the processor that can understand and execute a sequence of operations. The RISC architecture, on the other hand, uses a set of simple instructions that are executed quickly; to perform a complex operation, those simple instructions are combined to form the overall complex operation. Although the RISC approach requires more lines of processor assembly code, it enables smaller and faster processors to be designed. RISC processors are incorporated into many embedded systems.

In a digital control or digital filtering application, a number of operations that need to be performed are common to all applications, and the choice of which operations to incorporate and in which order depends on the application. Table 7.4 identifies the types of operation required.

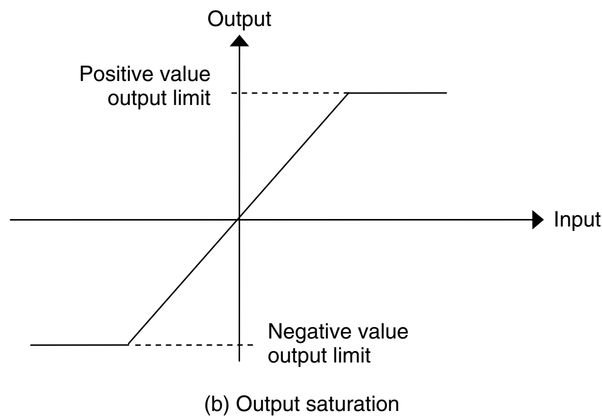
The overflow prevention operation is required to prevent a value from exceeding its positive and negative limits for correct operation. For example, a four-bit, 2s complement signed number has a range from -8_{10} (1000_2) to $+7_{10}$ (0111_2). If the number is at a value of $+7_{10}$ (0111_2) and one is added to it, the resulting binary code

Table 7.4: Basic operations for digital control and digital filtering

Type of operation	Description
Arithmetic	Perform the basic operations of addition, subtraction, multiplication, and division.
Value store	Store a value in a register for use at a later time.
Wordlength increase/decrease	Increase/Decrease the wordlength of a value to account for the value increasing/decreasing as an arithmetic operation is performed on it.
Overflow prevention	Prevent a value from exceeding a predefined limit (both positive and negative values).
Value truncation	Limit the wordlength of a value by truncation.
Value rounding	Limit the wordlength of a value by rounding.
Conversion	Convert values from one form to another (e.g., unsigned binary to 2s complement signed binary and vice versa).
Sample input control	Control the sampling of the analogue signal(s) to use as the input(s) to the digital controller or filter.
Update output control	Control the output of the analogue signal(s) result(s) as the output(s) from the digital controller or filter.
External communications	Communicate with external systems.



(a) Overflow occurs



(b) Output saturation

Figure 7.27: Overflow and saturation

would be 1000_2. This is -8_{10} in the number system, even though the number should be $+8_{10}$. This effect is referred to as overflow and must be prevented, either by designing circuitry to detect the possibility of overflow and preventing it, or by ensuring that the situation would never occur in the normal operation of the design. Figure 7.27 shows the effect of saturation on an adder that adds 2s complement numbers where (a) there is no overflow prevention, and (b) the output of the adder is designed to saturate rather than overflow. The detection circuitry and saturation can be coded in VHDL. An example schematic for such a circuit is shown in Figure 7.28. Here, the 2s

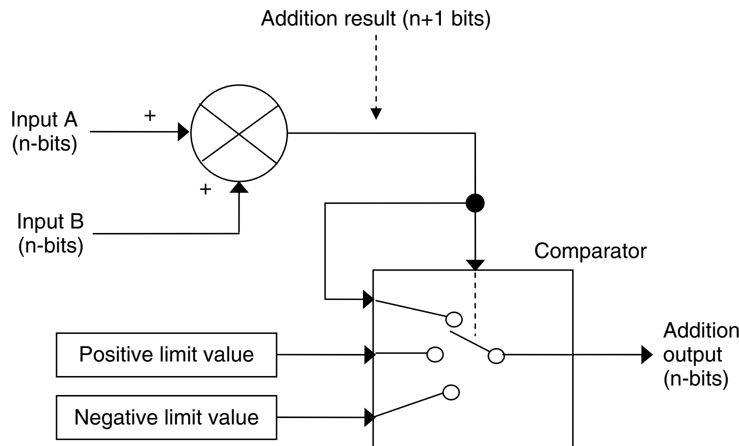


Figure 7.28: 2s complement adder with overflow prevention

complement adder receives two n-bit words and performs an n + 1 addition. The result of this addition is then compared to value limits (positive and negative), and depending on the result of the comparison, the circuit will produce one of three outputs:

1. the result of the addition (n-bits of the n + 1 bit number)
2. the positive limit value (n-bits)
3. the negative limit value (n-bits)

This occurs in a situation where the result of an n-bit arithmetic operation remains n-bits in size. However, in a custom architecture, the potential exists for the range of values to increase or decrease in wordlength as it passes through the arithmetic operations. The designer has this choice.

The choice of wordlength and the truncation or rounding of values as they pass through a digital filter or digital control algorithm affects the result; specifically, how closely the digital result in the implementation represents the result of the calculation if truncation or rounding had not occurred. Additionally, the examples considered in this text apply to fixed-point arithmetic. Designs can also accommodate floating point arithmetic.

The digital control algorithm can be designed using any of a number of possible methods. In many cases, the proportional plus integral plus derivative (PID) controller is used, and the implementation of this algorithm in digital will be

considered in this text. When an analogue controller is to be used as a prototype for the digital controller, and the analogue controller is to be developed using Laplace transforms, then the transformation between the analogue and digital will be undertaken in three phrases:

1. Develop the initial Laplace transform equation (using the variable *s*).
2. Replace the variable *s* with one of the available approximations, so that now the equation is in terms of the variable z; that is, create the pulse transfer function G(z).
3. Implement the equation either using digital logic (hardware) or in software.

The pulse transfer function G(z) is created using one of the following:

- Forward difference or Euler's method:

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

- Backward difference method:

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

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

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

Here, T is the signal sampling period. These methods are readily applied by hand to transform from *s* to *z*.

Example 4: Proportional (P) Control

Consider a digital controller that is to perform proportional control. The controller will accept two inputs, the command input and feedback signals, and will output a single controller output, the controller effort signal. The two inputs are initially subtracted and multiplied by a gain value (the proportional gain Kp is set here to +7). This gain value is held in a ROM. The arrangement for this controller is shown in

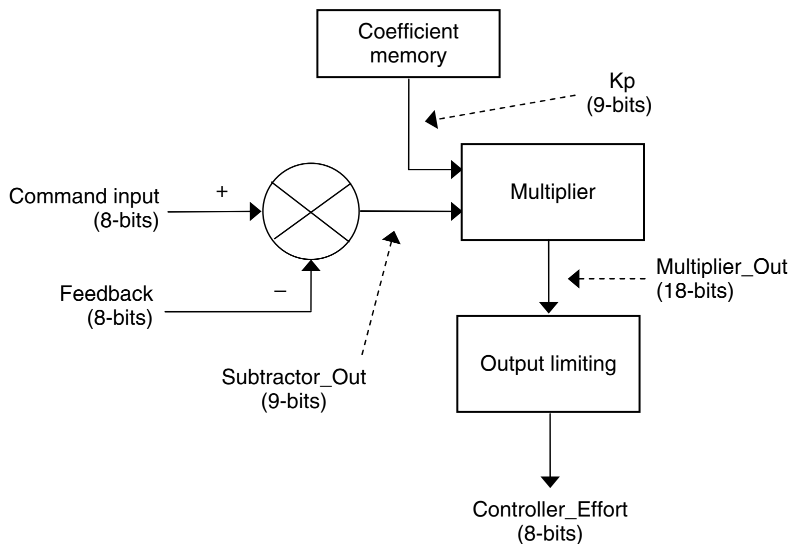


Figure 7.29: Digital proportional gain

Figure 7.29, and here, the internal wordlength increases as the values pass through the arithmetic operations, but finally will be limited to eight bits at the controller output (the inputs are also eight bits). The multiplication in this example is undertaken using a digital multiplier. Figure 7.30 provides the VHDL code for the structure of this design. In this implementation, each block will be coded as a unique entity-architecture pair, although this might not necessarily be the best solution. The design here is purely combinational logic and as such includes no clock or reset inputs.

Figure 7.31 shows the schematic for the synthesized VHDL code using the Xilinx® ISE™ tools. When a digital multiplier is required and the coefficient is fixed, then an alternative to using a digital multiplier is to use a shift-and-add operation. For example, multiplying a value by 2 is a shift-left operation by one bit—simple and easy to do in digital logic and avoids the need for a large digital multiplier.

Example 5: Discrete-Time Integrator

In many situations, then integral action is added to the proportional action in order to achieve the required response from the plant. The integral action can be represented

```
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 Proportional_Gain IS
8       PORT ( Command_Input       : IN   STD_LOGIC_VECTOR (7 downto 0);
9              Feedback            : IN   STD_LOGIC_VECTOR (7 downto 0);
10             Controller_Effort   : OUT  STD_LOGIC_VECTOR (7 downto 0));
11  END ENTITY Proportional_Gain;
12
13
14  ARCHITECTURE Structural OF Proportional_Gain IS
15
16
17  SIGNAL  Subtractor_Out  :  STD_LOGIC_VECTOR(8 downto 0);
18  SIGNAL  Kp              :  STD_LOGIC_VECTOR(8 downto 0);
19  SIGNAL  Multiplier_Out  :  STD_LOGIC_VECTOR(17 downto 0);
20
21
22  COMPONENT Subtractor IS
23      PORT ( Data_In_1     : IN   STD_LOGIC_VECTOR (7 downto 0);
24             Data_In_2     : IN   STD_LOGIC_VECTOR (7 downto 0);
25             Data_Out      : OUT  STD_LOGIC_VECTOR (8 downto 0));
26  END COMPONENT Subtractor;
27
28
29  COMPONENT Coefficient_Memory IS
30      PORT ( Data_Out    : OUT  STD_LOGIC_VECTOR (8 downto 0));
31  END COMPONENT Coefficient_Memory;
32
33
34  COMPONENT Multiplier IS
35      PORT ( Data_In      : IN   STD_LOGIC_VECTOR (8 downto 0);
36             Coefficient  : IN   STD_LOGIC_VECTOR (8 downto 0);
37             Data_Out     : OUT  STD_LOGIC_VECTOR (17 downto 0));
38  END COMPONENT Multiplier;
39
40
41  COMPONENT Output_Limit IS
42      PORT ( Data_In     : IN   STD_LOGIC_VECTOR (17 downto 0);
43             Data_Out    : OUT  STD_LOGIC_VECTOR (7 downto 0));
44  END COMPONENT Output_Limit;
45
46
47  BEGIN
48
49  I1 : Subtractor
50       PORT MAP ( Data_In_1    =>  Command_Input,
51                  Data_In_2    =>  Feedback,
52                  Data_Out     =>  Subtractor_Out);
53
54  I2 : Coefficient_Memory
55       PORT MAP ( Data_Out     =>  Kp);
56
57  I3 : Multiplier
58       PORT MAP ( Data_In      =>  Subtractor_Out,
59                  Coefficient  =>  Kp,
60                  Data_Out     =>  Multiplier_Out);
61
62  I4 : Output_Limit
63       PORT MAP ( Data_In      =>  Multiplier_Out,
64                  Data_Out     =>  Controller_Effort);
65
66
67  END ARCHITECTURE Structural;
```

Figure 7.30: Digital proportional gain VHDL structure code

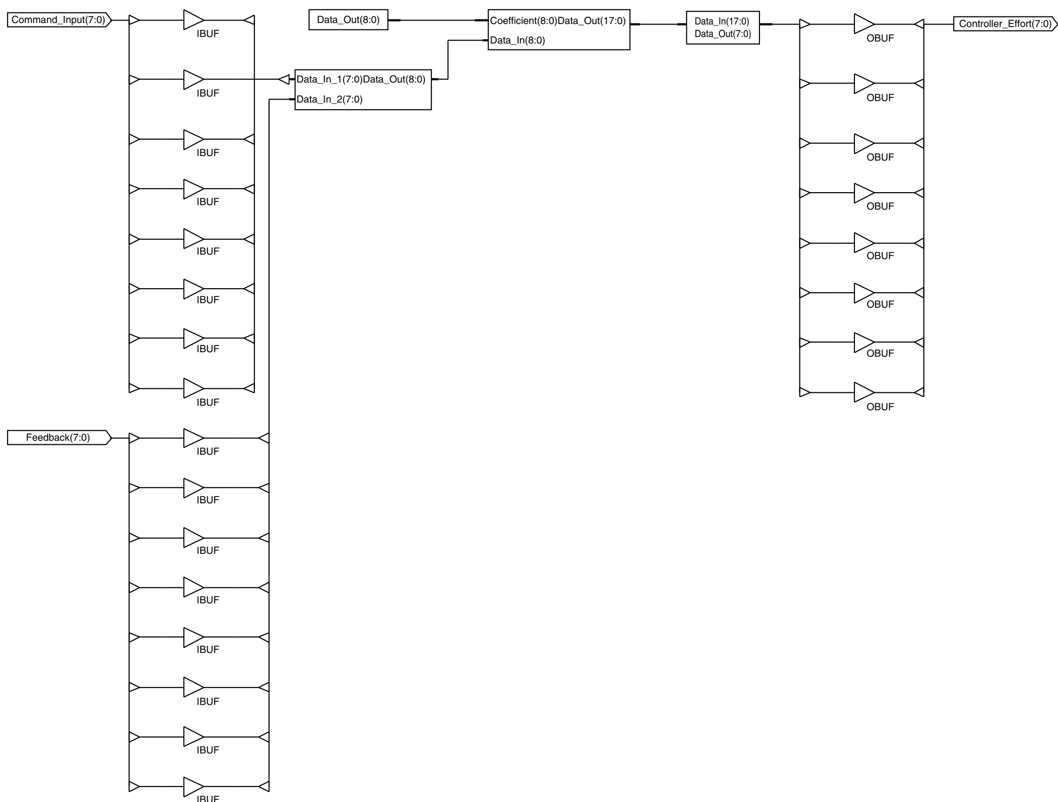


Figure 7.31: Digital proportional gain schematic for the synthesized VHDL code

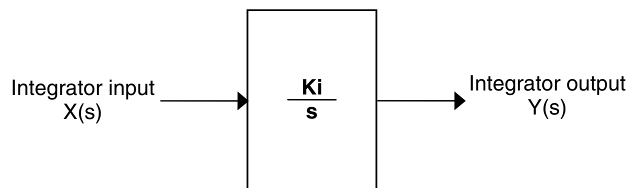


Figure 7.32: Integral action (Laplace transform)

using Z-transforms. Taking an integral action represented initially using a Laplace transform as shown in Figure 7.32, this can be translated to a Z-transform by one of a number of transforms.

Here, Ki is the integral action gain. The (Ki/s) equation can be transformed using Tustin's approximation, giving:

$$Y(z) = \left(\frac{Ki}{\left(\frac{2}{T}\right)\left(\frac{z-1}{z+1}\right)} \right) X(z)$$

This can be manipulated to create:

$$Y(z)(z-1) = X(z)\left(\frac{KiT}{2}\right)(z+1)$$

Finally, manipulating this further gives the equation in terms of the current sample and previous (delayed) samples with the equation in terms of z^{-n}:

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

This can be represented by the block diagram shown in Figure 7.33. Here, each of the operations is identified and can be implemented in hardware using any of three methods:

1. multiplication by (KiT/2)
2. two addition of two values
3. two value delays by one sample (z^{-1})

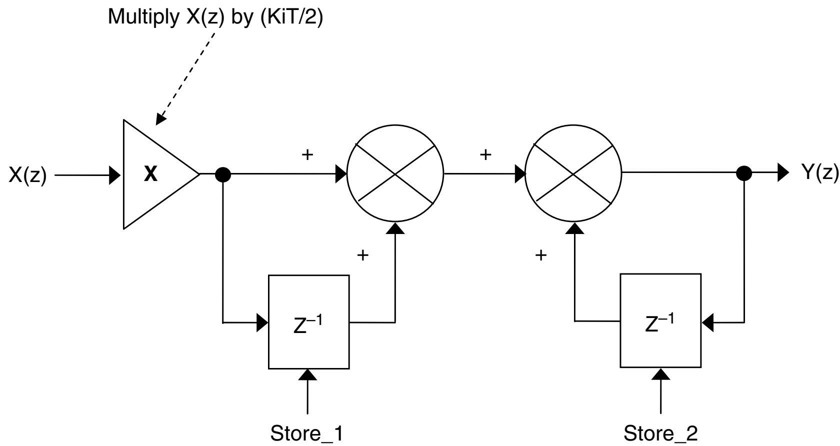


Figure 7.33: Discrete-time integral action

This uses the same basic building blocks as previous examples and can be implemented in VHDL as a structural description (using discrete designs for each of the functional blocks) or as a behavioral or dataflow description.

The multiplication is positioned before the first addition operation. However, the multiplication can be placed after the addition, and if necessary, values can be scaled within the design to address the potential problem of ever-increasing wordlengths due to the range of values that could be encountered in the design.

A modification to the integrator design shown in Figure 7.33 would include an antiwindup circuit. Integrator windup can occur when an input is of a size and polarity that, over time, causes the integrator output to become larger and larger. It can take a substantial amount of time for the integrator output to reduce when the input signal reverses polarity. Additionally, as the values within the integrator become larger, the potential for overflow occurs, which must be taken into account in the design of the circuit.

Example 6: Discrete-Time Differentiator

In addition to the proportional and integral actions, derivative action (a differentiator) can be added to achieve the required response from the plant. The derivative action can be represented using Z-transforms. A derivative action represented initially using a Laplace transform, as shown in Figure 7.34, can be translated to a Z-transform by any of a number of transforms.

Here, Kd is the derivative action gain. The (Kd s) equation can be transformed using Tustin's approximation. This then gives:

$$Y(z) = (Kd)\left(\left(\frac{2}{T}\right)\left(\frac{z-1}{z+1}\right)\right)X(z)$$

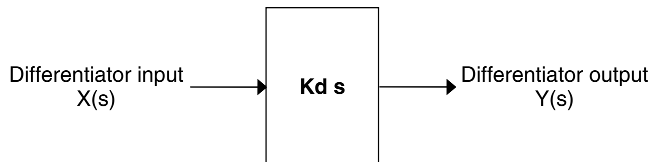


Figure 7.34: Differential action (Laplace transform)

This can be manipulated to create:

$$Y(z)(z+1) = X(z)\left(\frac{2Kd}{T}\right)(z-1)$$

Finally, manipulating further gives the equation in terms of the current sample and previous (delayed) samples with the equation in terms of z^{-n}:

$$Y(z) = \left(\left(\frac{2Kd}{T}\right)\left(x(z) - x(z)z^{-1}\right)\right) - Y(z)z^{-1}$$

This can be represented by the block diagram shown in Figure 7.35. Here, each of the operations is identified and this can be implemented in hardware using any of three methods:

1. multiplication by (2Kd/T)
2. two subtraction of two values
3. two value delays by one sample (z^{-1})

This uses the same basic building blocks as previous examples and can be implemented in VHDL as a structural description (using discrete designs for each of the functional blocks) or as a behavioral or dataflow description.

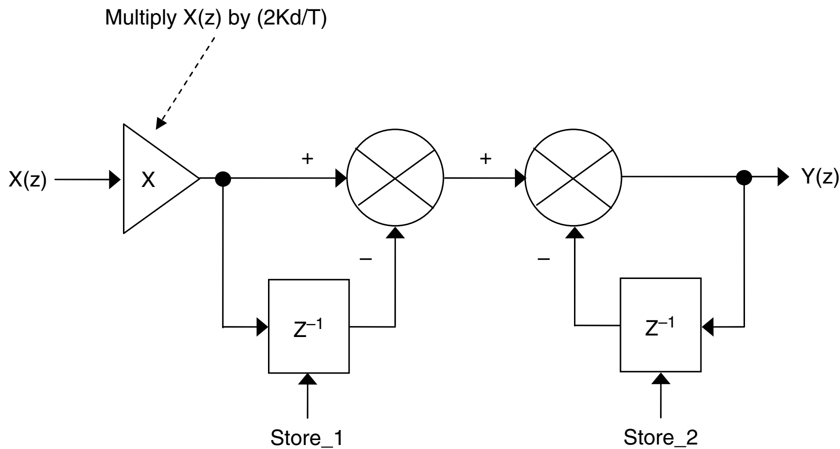


Figure 7.35: Discrete-time derivative action

The multiplication is positioned before the first subtraction operation. However, the multiplication could be placed after the subtraction, and if necessary, values can be scaled within the design to address the potential problem of ever-increasing wordlengths due to the range of values that could be encountered in the design.

Although this structure is similar to the discrete-time integrator, it would not suffer from windup because the feedback signal to the second subtractor is subtracted from the internal signal rather than added.

Example 7: PID Controller

The proportional, integral, and derivative control actions can be brought together to create a PID controller. Figure 7.36 shows an example of how this can be created. As the design increases in complexity, the need for more additions/subtractions and multiplications/divisions increases. This highlights the need to develop an architecture that uses hardware efficiently and can operate within the time constraints of the design. The arithmetic operations to be undertaken either can be designed to be either separate actions (each action requiring its own dedicated hardware) or can be shared (each addition, subtraction, multiplication, or division has a single common block, as is typical in the architecture of an arithmetic and logic unit, ALU). Hence, design speed of operation can be considered against the size of the hardware circuit required for a given architecture.

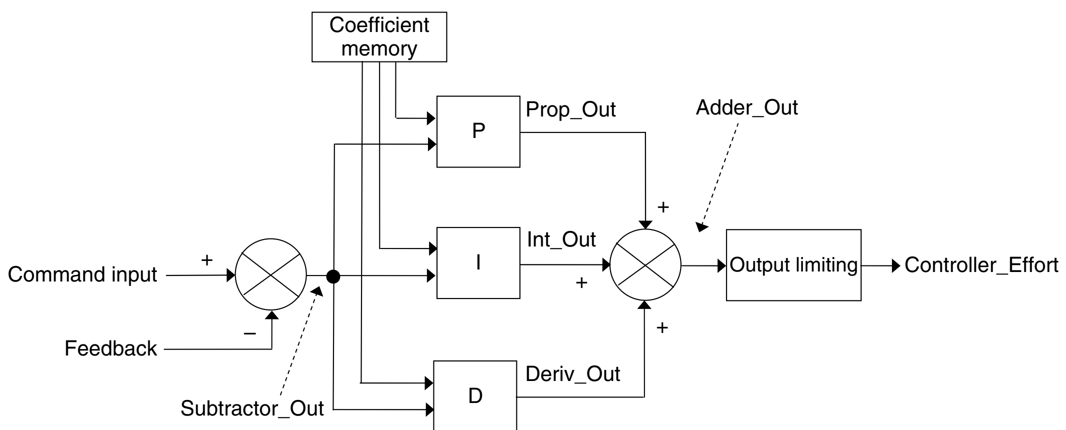


Figure 7.36: Digital PID controller

With the design shown in Figure 7.36, the actions can be implemented such that one of the following two scenarios exists:

1. Each action identified in the block diagram can be created using its own dedicated hardware.
2. Resources can be shared. Table 7.5 provides an example flow of actions for an implementation using shared resources.

Table 7.5: Shared resources for the digital PID controller

Action number	Action description
1	Subtract the feedback input from the command input
2	Store result (Subtractor_Out)
3	Read Subtractor_Out and apply to proportional action
4	Store result (Prop_Out)
5	Read Subtractor_Out and apply to integral action
6	Store result (Int_Out)
7	Read Subtractor_Out and apply to derivative action
8	Store result (Deriv_Out)
9	Read Prop_Out, Int_Out, and Deriv_Out
10	Add Prop_Out, Int_Out, and Deriv_Out
11	Store result (Adder_Out)
12	Read Adder_Out
13	Apply output limiting
14	Store result (Controller_Effort)

7.4 Digital Filtering

7.4.1 Introduction

A filter is a circuit that performs some type of signal processing on a frequency-dependent basis. These filters can be realized in both analogue and digital circuits. Digital filters receive one or more discrete time signals (signal samples) and modify these signals to produce one or more outputs, and filters will pass or reject frequencies based on their required operation:

1. *Low-pass* filters will pass low-frequency signals but reject high-frequency signals.
2. *High-pass* filters will pass high-frequency signals but reject low-frequency signals.

3. *Band-pass* filters will pass a band of signal frequencies but will reject frequencies lower than or higher than the pass range.
4. *Band-reject* or *notch* filters will reject a band of signal frequencies but will pass frequencies lower than or higher than the pass range.

The idealized response for each of the filters is shown in Figure 7.37. On each plot, the X-axis is the frequency (f), and the Y-axis is the magnitude (|H|) of the filter output signal at a particular frequency. The response of an actual filter will deviate from this idealized response. Additionally, although only the filter signal output magnitude is shown, both the signal magnitude and phase response would need to be considered.

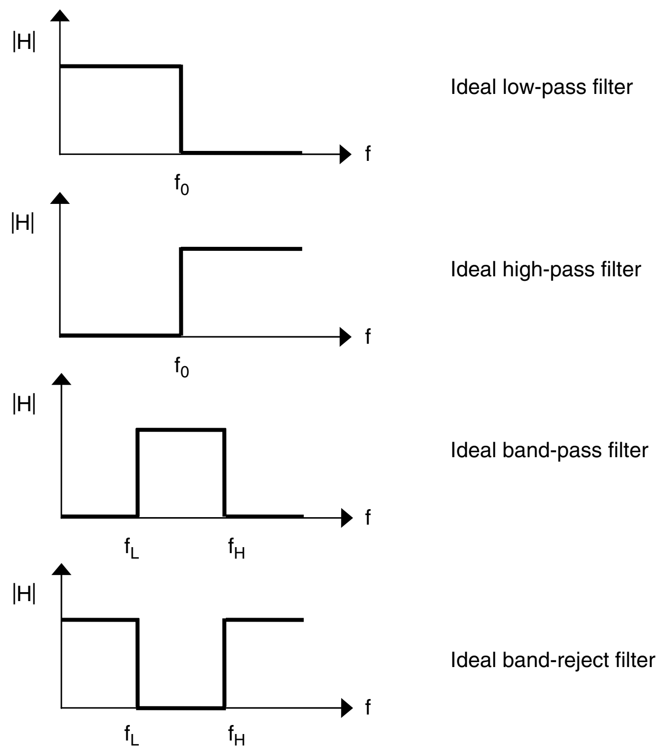


Figure 7.37: Idealized filter response

There are four types of filter design [10]:

1. Bessel filter
2. Butterworth filter
3. Chebyshev filter
4. elliptic filter

The ideal filter response is also referred to as a brick-wall response due to its shape. In low-pass and high-pass filters, the cut-off frequency is f_0. For the band-pass filter, two cut-off frequencies exist, lower (f_L) and upper (f_H), and signals are passed between them. The center frequency is in the center of the pass-band. The frequency range between the lower and upper cut-off frequencies is the bandwidth of the filter. The band-reject filter is the complement of the band-pass filter. To the four responses identified in Figure 7.37 is added a fifth, the all-pass filter. With this, all signal frequencies are passed.

Analogue filters are either passive filters (containing resistors, capacitors, and inductors) or active filters (using active devices such as a transistor or operational amplifier).

Digital filters use DSP techniques on either software- or hardware-based systems. The general structure for a digital filter, shown in Figure 7.38, is similar to the digital controller previously discussed, but the architecture here is presented in a slightly different arrangement. Only one analogue input signal is to be sampled, and the output is a single analogue signal.

The following components are identified in Figure 7.38:

- The *digital filter core* contains three main blocks:
  - The *digital filter algorithm* is responsible for implementing the algorithm operations (add, subtract, multiply, divide, store).
  - The *filter coefficient memory* stores the coefficients used by the digital filter algorithm for multiplications and divisions.
  - The *control unit* provides the necessary timing for actions to occur (ADC input sampling, DAC output updating, filter coefficient memory access, and digital filter algorithm operation).
- The *communications port* allows the filter to communicate with an external digital system.

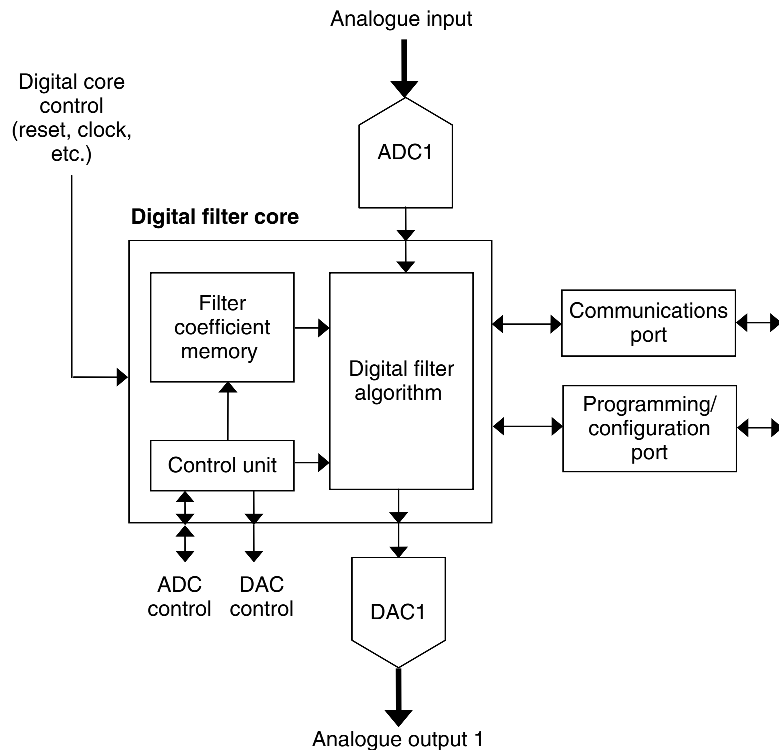


Figure 7.38: General digital filter architecture (with analogue I/O)

- The *programming/configuration port* uploads a software program (in a processor-based system) or a hardware configuration (in an FPGA- or CPLD-based system).

The complete system is controlled by external control signals such as a clock and reset.

This architecture can be modified to provide for different scenarios and specific implementation requirements. In general, the choice of architecture must consider a range of design and implementation issues that include:

1. whether to use standard processor type architecture or to develop a custom architecture
2. available hardware resources

3. functionality possible with the target technology
4. design performance requirements
5. ability to modify and/or upgrade the design
6. power consumption of the circuit implementation
7. circuit power supply requirements
8. peripheral integration—the ability to connect peripheral devices as and when necessary
9. cost
10. availability of suitable design tools
11. availability of a suitable design flow
12. support of **DfX**:
    - **DfA**, design for assembly
    - **DfD**, design for debug
    - **DfM**, design for manufacturability
    - **DfR**, design for reliability
    - **DfT**, design for testability
    - **DfY**, design for yield

Example 8: Digital Filter Structure

Consider the filter architecture shown in Figure 7.38. This can be coded for in VHDL as a structural description. Consider the case where the digital filter algorithm requires four control signals and eight fixed 16-bit coefficients. The filter coefficients are stored in a ROM within the design and are set when the CPLD is configured. A structural description for each of the main blocks within the digital filter core is shown in Figure 7.39.

The filter coefficient memory has three address lines and sixteen data lines. There are no memory control signals, so when an address is applied to the memory, the data stored in that address is applied to the digital filter algorithm.

The ADC used is the AD7575, and the DAC used is the AD7524.

```
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 Filter_Core IS
8       PORT ( Master_Clock    :  IN   STD_LOGIC;
9              Master_Reset    :  IN   STD_LOGIC;
10             ADC_Data_In     :  IN   STD_LOGIC_VECTOR(7 downto 0);
11             ADC_BUSY        :  IN   STD_LOGIC;
12             ADC_TP          :  OUT  STD_LOGIC;
13             ADC_RD          :  OUT  STD_LOGIC;
14             ADC_CS          :  OUT  STD_LOGIC;
15             DAC_Data_Out    :  OUT  STD_LOGIC_VECTOR(7 downto 0);
16             DAC_WR          :  OUT  STD_LOGIC;
17             DAC_CS          :  OUT  STD_LOGIC);
18  END ENTITY Filter_Core;
19
20
21  ARCHITECTURE Structural OF Filter_Core IS
22
23
24  SIGNAL Coefficient_Internal      : STD_LOGIC_VECTOR(15 downto 0);
25  SIGNAL Control_Internal          : STD_LOGIC_VECTOR(3 downto 0);
26  SIGNAL Memory_Address_Internal   : STD_LOGIC_VECTOR(2 downto 0);
27
28
29  COMPONENT Algorithm IS
30      PORT ( Filter_In        : IN   STD_LOGIC_VECTOR (7 downto 0);
31             Reset            : IN   STD_LOGIC;
32             Coefficient      : IN   STD_LOGIC_VECTOR (15 downto 0);
33             Filter_Control   : IN   STD_LOGIC_VECTOR (3 downto 0);
34             Filter_Out       : OUT  STD_LOGIC_VECTOR (7 downto 0));
35  END COMPONENT Algorithm;
36
37
38  COMPONENT Coefficient_Memory IS
39      PORT ( Address          : IN   STD_LOGIC_VECTOR (2 downto 0);
40             Data             : OUT  STD_LOGIC_VECTOR (15 downto 0));
41  END COMPONENT Coefficient_Memory;
42
43
44  COMPONENT Control_Unit IS
45      PORT ( Master_Clock     :  IN   STD_LOGIC;
46             Master_Reset     :  IN   STD_LOGIC;
47             Filter_Control   :  OUT  STD_LOGIC_VECTOR (3 downto 0);
48             Memory_Address   :  OUT  STD_LOGIC_VECTOR (2 downto 0);
49             ADC_BUSY         :  IN   STD_LOGIC;
50             ADC_TP           :  OUT  STD_LOGIC;
51             ADC_RD           :  OUT  STD_LOGIC;
52             ADC_CS           :  OUT  STD_LOGIC;
53             DAC_WR           :  OUT  STD_LOGIC;
54             DAC_CS           :  OUT  STD_LOGIC);
55  END COMPONENT Control_Unit;
```

Figure 7.39: Digital filter core example

```
56
57
58  BEGIN
59
60  I1 : Algorithm
61      PORT MAP( Filter_In        =>  ADC_Data_In,
62                Reset            =>  Master_Reset,
63                Coefficient      =>  Coefficient_Internal,
64                Filter_Control   =>  Control_Internal,
65                Filter_Out       =>  DAC_Data_Out);
66
67  I2 : Coefficient_Memory
68       PORT MAP( Address          =>  Memory_Address_Internal,
69                 Data             =>  Coefficient_Internal);
70
71  I3 : Control_Unit
72       PORT MAP( Master_Clock     =>  Master_Clock,
73                 Master_Reset     =>  Master_Reset,
74                 Filter_Control   =>  Control_Internal,
75                 Memory_Address   =>  Memory_Address_Internal,
76                 ADC_BUSY         =>  ADC_BUSY,
77                 ADC_TP           =>  ADC_TP,
78                 ADC_RD           =>  ADC_RD,
79                 ADC_CS           =>  ADC_CS,
80                 DAC_WR           =>  DAC_WR,
81                 DAC_CS           =>  DAC_CS);
82
83  END ARCHITECTURE Structural;
```

Figure 7.39: (Continued)

The control unit identifies the control signals for the memory, algorithm, ADC, and DAC, and does not include any control signals for the communications interface.

Figure 7.40 shows the schematic for the synthesized VHDL code using the Xilinx® ISE™ tools.

VHDL entity-architecture pairs can then be created to complete the design by adding the required detail to the algorithm, memory, and control unit blocks.

Example 9: Multiply by Two

Although a digital implementation could be created to solve a given problem, it is not always suitable. Consider the need to amplify an analogue voltage by two. This could be implemented in analogue or digital, and Figure 7.41 shows a possible implementation of both. The analogue circuit uses a noninverting operational amplifier (op-amp). The digital circuit is rather more complex.

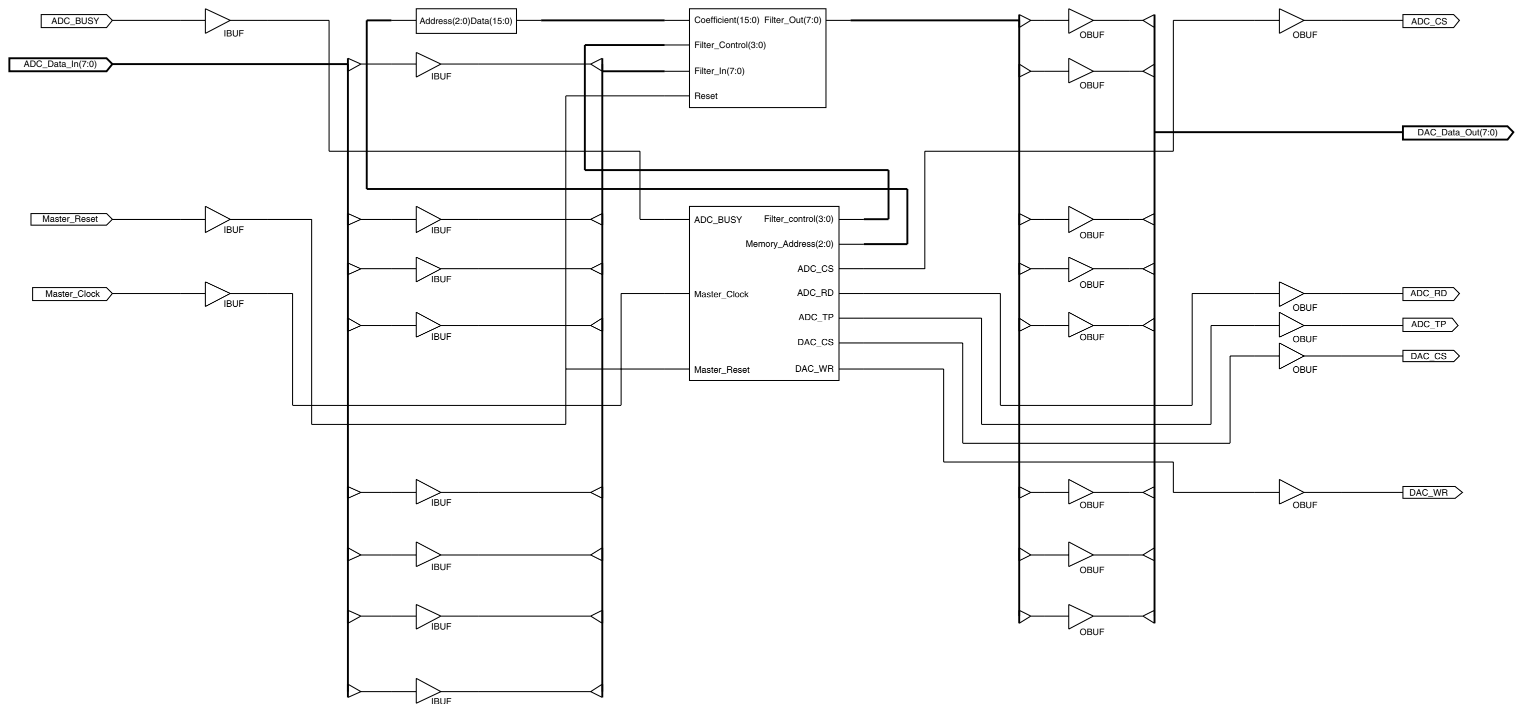


Figure 7.40: Digital filter example schematic for the synthesized VHDL code

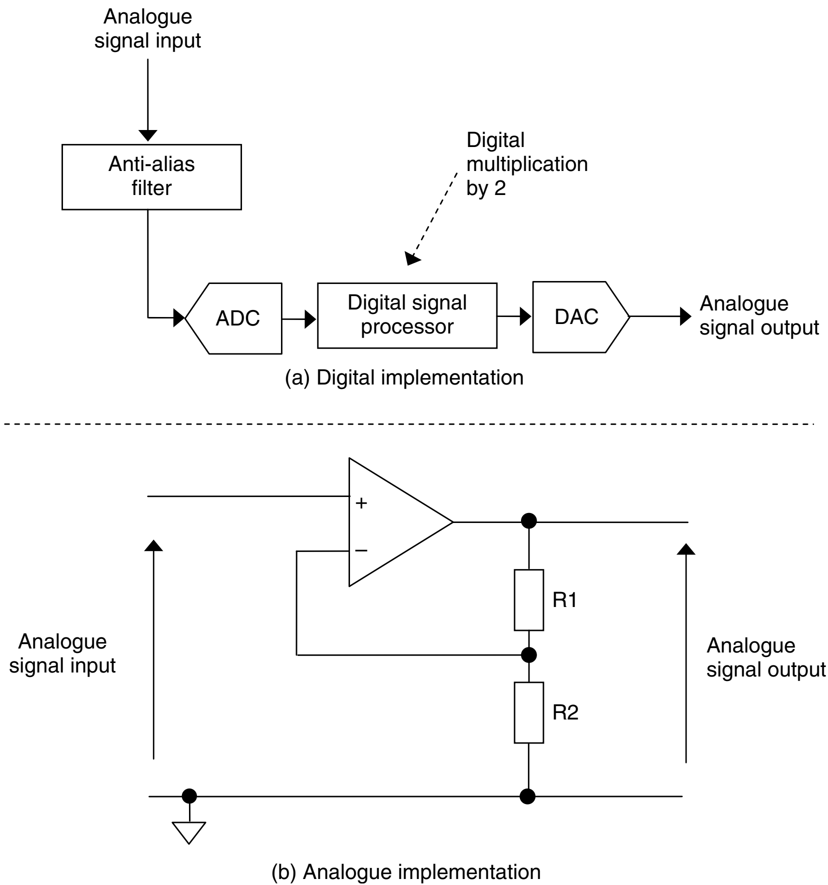


Figure 7.41: Amplifier implementation

Which implementation would be better?

Filters are of two types: infinite impulse response (IIR) and finite impulse response (FIR). The type of filter chosen determines the architecture of the filter and what values are to be used in the calculations. The basic filter structures are identified below.

7.4.2 Infinite Impulse Response Filters

The infinite impulse response (IIR) filter is a recursive filter in that the output from the filter is computed by using the current and previous inputs and previous outputs.

Because the filter uses previous values of the output, there is feedback of the output in the filter structure. The design of the IIR filter is based on identifying the pulse transfer function G(z) that satisfies the requirements of the filter specification. This can be undertaken either by developing an analogue prototype and then transforming it to the pulse transfer function, or by designing directly in digital. Figure 7.42 shows typical IIR filter architecture.

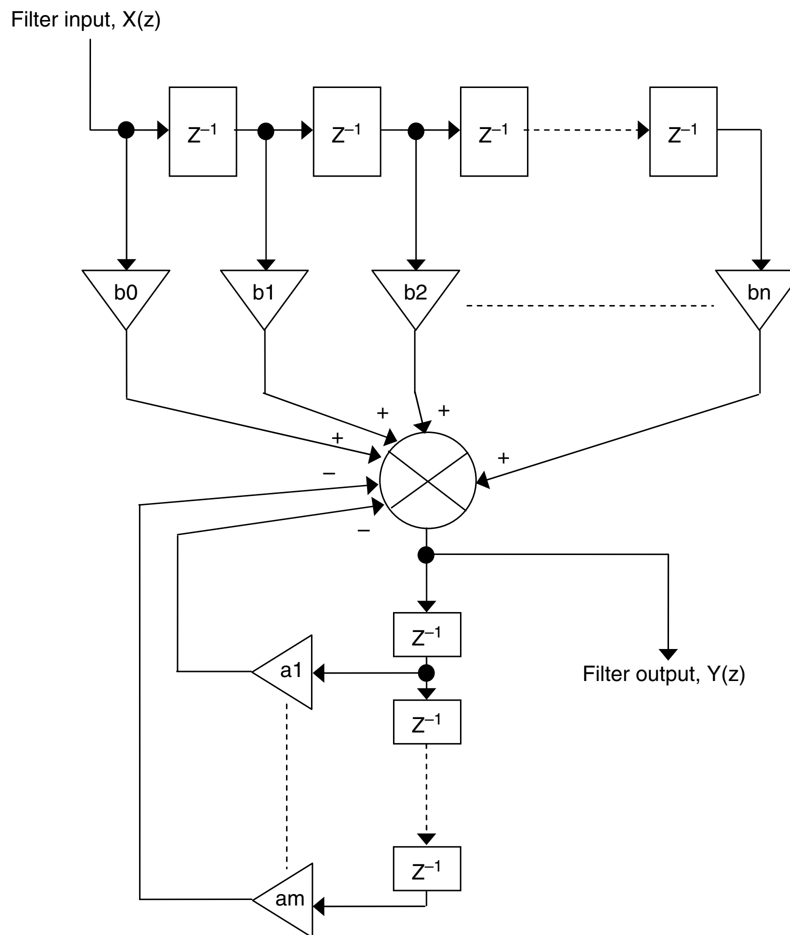


Figure 7.42: Typical architecture of an IIR filter

7.4.3 Finite Impulse Response Filters

The finite impulse response (FIR) filter is a nonrecursive filter in that the output from the filter is computed by using the current and previous inputs. It does not use previous values of the output, so there is no feedback in the filter structure. The design of the FIR filter is based on identifying the pulse transfer function G(z) that satisfies the requirements of the filter specification. This can be undertaken either by developing an analogue prototype and then transforming this to the pulse transfer function, or by designing directly in digital. A nonrecursive filter is always stable, and the amplitude and phase characteristics can be arbitrarily specified. However, a nonrecursive filter generally requires more memory and arithmetic operations than a recursive filter equivalent. Figure 7.43 shows typical FIR filter architecture.

Here, the filter input is applied to a sequence of sample delays (z^{-1}), and the outputs from each delay (and the input itself) are applied to the inputs of multipliers. Each multiplier has a coefficient set by the filter requirements. The outputs from each multiplier are then applied to the inputs of an adder, and the filter output is then taken from the output of the adder.

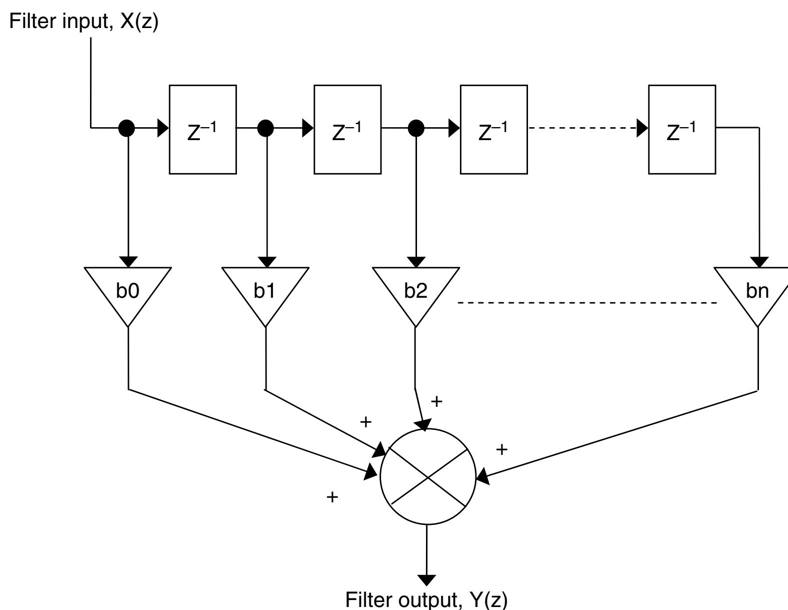


Figure 7.43: Typical architecture of an FIR filter

References

[1] Terrell, T. J., *Introduction to Digital Filters*, The MacMillan Press Ltd., 1980, ISBN 0-333-24671-3.

[2] Kamen, E. W., and Heck, B. S., *Fundamentals of Signals and Systems Using the Web and MATLAB®*, Pearson Education Ltd., 2007, ISBN 0-13-168737-9.

[3] Ifeachor, E. C., and Jervis, B. W., *Digital Signal Processing: A Practical Approach*, Pearson Education Ltd., 2002, ISBN 0-201-59619-9.

[4] Meade, M. L., and Dillon, C. R., *Signals and Systems Models and Behaviour*, Chapman & Hall, 1991, ISBN 0-412-40110-x.

[5] Hanselman, D., and Littlefield, B., *Mastering MATLAB® 6—A Comprehensive Tutorial and Reference*, Prentice Hall Inc., 2001, ISBN 0-13-019468-9.

[6] Golten, J., and Verwer, A., *Control System Design and Simulation*, McGraw-Hill, 1991, ISBN 0-07-707412-2.

[7] Astrom, K. J., and Wittenmark, B., *Computer-Controlled Systems Theory and Design*, Second Edition, Prentice Hall International, 1990, ISBN 0-13-172784-2.

[8] Analog Devices Inc., *AD7575 LC^{2}MOS Successive Approximation ADC* datasheet.

[9] Analog Devices Inc., *AD7524 CMOS 8-Bit Buffered Multiplying DAC* datasheet.

[10] Schaumann, R., and Van Valkenburg, M., *Design and Analog Filters*, Oxford University Press, 2001, ISBN 0-19-511877-4.

Student Exercises

7.1 Develop the VHDL code for a design that will perform the following three functions:

- Sample an analogue signal from a 12-bit ADC. (Choose an ADC and obtain the required control signals from the device data sheet.)
- Multiply the signal by 0.76 with an error of no more than 5 percent.
- Output the result to a 12-bit DAC. (Choose an DAC and obtain the required control signals from the device data sheet.)

7.2 Modify the design in Exercise 7.1 so that the sample is multiplied by a value set from a PC via a simple UART receiver (using the integer value of the byte received from the UART).

7.3 From analysis of the data sheets from available ADCs, create the VHDL code that will control the sampling from the following ADCs:

- 8-bit
- 10-bit
- 12-bit
- 14-bit
- 16-bit
- 18-bit

7.4 From analysis of the data sheets from available DACs, create the VHDL code that will control the output of data to the following DACs:

- 8-bit
- 10-bit
- 12-bit
- 14-bit
- 16-bit
- 18-bit

7.5 Develop the VHDL code for a PID controller where each of the actions in the controller is defined in its own entity-architecture pair. The coefficients for each of the control actions are to be stored in a ROM. What assumptions are made in the implementation?

7.6 Develop the VHDL code for a PID controller where the additions/subtractions and multiplications/divisions are shared by all of the control actions. What assumptions are made in the implementation?