Chapter 2
FPGA Application Design

In wired or wireless communication systems, the information that needs to be transmitted is not only required to reach the destination but it should be error free and should make efficient use of the channel bandwidth available. Various DSP based encoding/decoding algorithms, data compression and noise filtering techniques have been developed to achieve effective and efficient data transmission with the help of FPGAs for hardware implementation. FPGA based implementations provide the flexibility of re-programming and quick delivery of the product to the market.

This chapter demonstrates the design of a simple DS-SS system including the basic building blocks such as, PN sequence generator, BPSK modulator/demodulator, BOOTH multiplier, Low Pass Filter and convolutional coding. The system is designed using Verilog HDL, simulation and functional verification of the design is performed using ModelSim® XE III 6.0d, and synthesis using Xilinx® ISE. The design is implemented and tested on Xilinx® Spartan 2E FPGA.

This chapter also demonstrates some of the algorithms and techniques used to accomplish data integrity and channel bandwidth efficiency in a communication system such as, Low Pass FIR filter using efficient Distributed Arithmetic (DA) architecture, Discrete Cosine Transform (DCT) using Scaled DCT architecture and Convolution coding and Viterbi decoding techniques. The Low Pass-Finite Impulse Response (LP-FIR) filter coefficients are calculated using MatLab FDA tool based on the given specification of the filter. The systems are designed using Verilog HDL, simulation and functional verification of the design is done using ModelSim® XE II 6.0d and synthesis using Xilinx® ISE. The designs are implemented on Xilinx® Spartan 2E FPGA.

The prerequisites for approaching this chapter would be an adequate background of basic digital communication system.

V.A. Chandrasetty, *VLSI Design: A Practical Guide for FPGA and ASIC Implementations*, SpringerBriefs in Electrical and Computer Engineering,
DOI 10.1007/978-1-4614-1120-8_2, © Springer Science+Business Media, LLC 2011

2.1 Design of Direct Sequence-Spread Spectrum System

Direct Sequence-Spread Spectrum (DS-SS) is a transmission technique in which a pseudo-noise code, independent of the information data is employed as a modulation waveform to "spread" the signal energy over a bandwidth much greater than the signal information bandwidth. At the receiver the signal is *de-spread* using a synchronized replica of the pseudo-noise code. The spreading sequence in DS-SS is often called as PN sequence.

In this section, the spread signal is modulated using Binary Phase Shift keying (BPSK) modulation technique in the transmitter and on the receiver side the modulated signal is recovered using BPSK demodulation technique.

The basic building blocks of DS-SS system are shown in Fig. 2.1 [1].

2.1.1 PN Sequence Generator

2.1.1.1 Overview of PN Sequence Generator

A Pseudo-random Noise (PN) sequence/code is a binary sequence that exhibits randomness properties but has a finite length and is therefore deterministic. PN generators are heart of every spread spectrum systems. Each symbol or bit in the sequence is called as *Chip* [2].

PN generators are based on Linear Feedback Shift Registers (LFSR). The contents of the registers are shifted right by one position at each clock cycle. The feedback from predetermined registers or taps to the left most register are XNOR-ed together.

LFSRs have several variables:

- The number of stages in the shift registers
- The number of taps in the feedback path
- The position of each tap in the shift registers stage
- The initial starting condition of the shift register often referred to as the "FILL" state

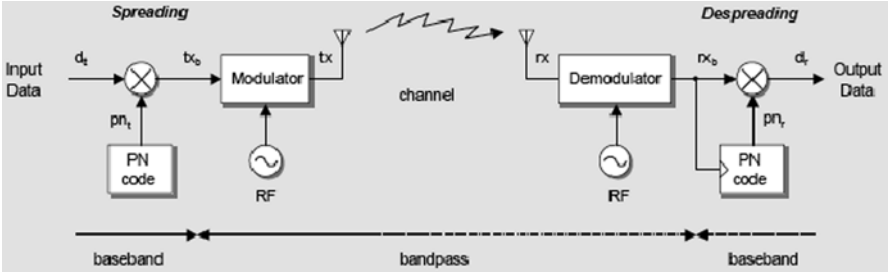

Fig. 2.1 Basic building blocks of DS-SS system

The longer the number of stages of shift registers in the PN generator, longer the duration of the PN sequence before it repeats. For a shift register of fixed length N, the number and duration of the sequences that it can generate are determined by the number and position of taps used to generate the parity feedback bit.

A maximum length sequence (L) for a shift register of length N is referred to as m-sequence and is defined as [3]:

$$L = 2^N - 1,$$

E.g. an eight stage LFSR will have a set of m-sequences of length 255.

Some of the most popular types of PN Sequence generators are:

- m-sequence codes
- Barker codes
- Gold codes

2.1.1.2 Design of PN Sequence Generator

Design

Specifications:

- Clock frequency for PN sequence generator system, $F_{pn} = 100$ KHz.
- LFSR length, $N = 4$.
- LFSRs are of D-FF type.
- X-NOR gate is used for linear parity feedback to the system.
- FPGA board clock frequency, $F_b = 50$ MHz (assumption)

Procedure:

- A clock frequency of 100 KHz for PN Sequence generator is designed using a divider of 500 clock cycles of F_b.
  Clock divider $= F_b/F_{pn} = 50$ MHz/100 KHz $= 500$
- Maximum length sequence, $N = 4$ corresponds to 4 D-FF to realize LFSRs of the PN generator system.
  Since $N = 4$, the maximum sequence length $L = 2^4 - 1 = 15$.
  Hence the sequence repeats every 15 clock cycles.
- The Chip rate for the PN sequence generator system is calculated as follows:
  Chip period, $T_c = 1/100$ KHz $= 10$ μs
  Chip rate, $F_c = 100$ KHz
- The bit period for the input data signal is calculated as follows:
  Data bit period, $T_d =$ Max. sequence Length (L) × Chip period (Tc)
  For the system, $T_d = 15 \times 10$ μs
  Hence, the input data bit period for the system is, $T_d \geq 150$ μs.

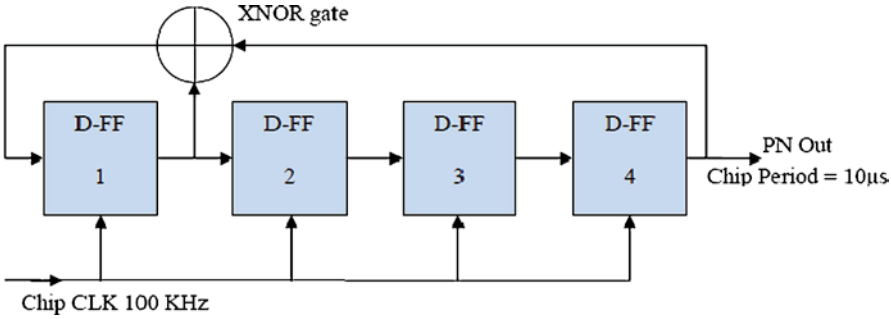

Fig. 2.2 Block diagram of a PN sequence generator

Block Diagram

The block diagram of a PN sequence generator for the design specification is shown in Fig. 2.2.

2.1.1.3 Properties of PN Sequence

Merits of using PN sequence [4]*:*

1. *Balance property*: In each period of the sequence the number of binary ones differ from the number of binary zeros by at most one digit (when LFSR stage length is odd)

$$\text{Pn} = +1+1+1-1-1+1-1 = +1$$

2. *Run-length Distribution*: A run is a sequence of a single type of binary digits. Among the sequence of ones and zeros in each period it is desirable that one-half the runs of each type are of length 1, about one-fourth are of length 2, one-eight are of length 3 and so on.
3. *Autocorrelation*: The origin of the name pseudo-noise is that the digital signal has an autocorrelation function which is very similar to that of a white noise signal. For PN sequences the autocorrelation has a large peaked maximum for perfect synchronization of two identical sequences (like white noise). The synchronization of receiver is based on this property.
4. *Cross-correlation*: Cross-correlation is the measure of agreement between two different codes pn_1 and pn_2. When Cross-correlation is zero the codes are called Orthogonal. In CDMA multiple users occupy the same RF bandwidth and transmit simultaneously. When the user codes are orthogonal, there is no

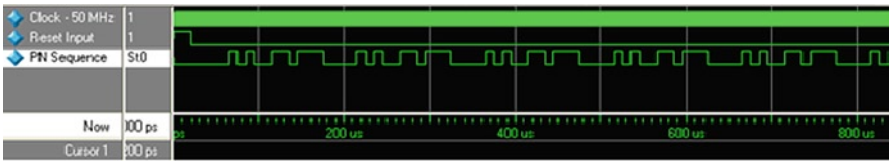

Fig. 2.3 Simulation results for PN sequence generator

interference between the users after dispreading and the privacy of the communication of each user is protected.

Demerits of using PN sequence [4]*:*

1. *Synchronization*: The most sensitive aspect of DS-SS system is the synchronization of the transmitter's PN sequence to that of the receiver where an offset of even one PN chip can result in noise rather than a de-spread symbol sequence.
2. *Increased Bandwidth*: As the data signal is spread using PN codes at higher frequency, there is an increase in bandwidth used in the process.
3. *Complexity*: There is an increased complexity and computational load both in the receiver and the transmitter to spread/de-spread the signal.

2.1.1.4 Simulation Results for PN Sequence Generator

The PN sequence generator is designed using Verilog HDL. Functional verification and simulation is performed using ModelSim.

The simulation results for PN sequence generator is shown in Fig. 2.3.

2.1.2 Transmitter for Direct Sequence-Spread Spectrum System

2.1.2.1 Overview of DS-SS Transmitter System

In DS-SS transmitter, the input data bits are spread by PN sequence generator. The spreading is actually done by multiplying the data bits with that of the PN sequence code generated. The frequency of PN sequence is higher than the Data signal. After spreading, the Data signal is modulated and transmitted. There are several schemes available for modulation, viz. BPSK, QPSK, M-QAM etc. The most widely used modulation scheme is the BPSK. In this design, BPSK modulation is used to modulate and transmit the spread signal.

The basic building blocks of a simple DS-SS transmitter system are shown in Fig. 2.4.

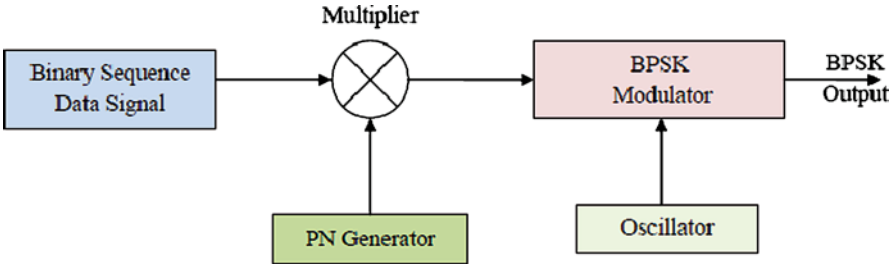

Fig. 2.4 Block diagram of a DS-SS transmitter system

Table 2.1 Truth table for the multiplier

m(t)	p(t)	s(t)
0	0	1
0	1	0
1	0	0
1	1	1

2.1.2.2 Design of DS-SS Transmitter

Multiplier Design

Specifications:

- PN sequence Chip rate, Tc = 10 μs.
- Data signal Bit rate, Tb ≥ 150 μs.

Let the data signal be m(t) and PN sequence p(t). The two signals are multiplied and the multiplied output is the spread signal. Truth table for the multiplier s(t) = m(t). p(t) is shown in Table 2.1.

From the truth table, it can be inferred that an XNOR gate can act as a multiplier to spread the data signal with the PN signal. Hence the block diagram for the multiplier is shown in Fig. 2.5.

Oscillator Design

Specification:

- PN sequence Chip rate, Tc = 10 μs.
- Carrier frequency, Fc ≥ 5 times Chip rate.

Design:

- The oscillator carrier sampling rate is designed
  Let the Sampling rate of sine wave be Fs = 25 MHz.

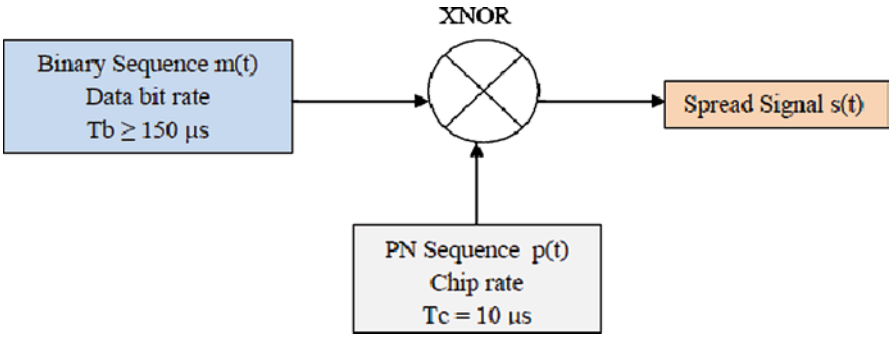

Fig. 2.5 Block diagram of a data and PN sequence multiplier

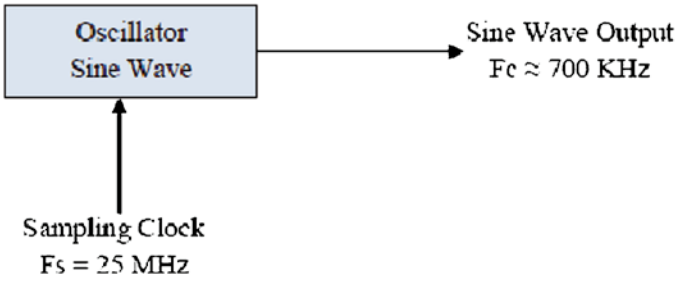

Fig. 2.6 Block diagram of an oscillator

- Number of samples for a full cycle of sine wave is designed
  Let the number of samples for a full cycle be N = 36.
- The oscillator is designed to generate sine wave of carrier frequency Fc
  $Fc \geq 5(1/T_C) = 5(1/10\mu s) = 500 KHz.$

For the above design with sampling rate 25 MHz and 36 samples per cycle, the carrier frequency, Fc = 25 MHz/36 ≈ 700 KHz. The oscillator is implemented using a Look-Up-Table (LUT) of nine samples and the logic is design in order to oscillate generating a sine wave.

The block diagram of the oscillator as per the design is shown in Fig. 2.6.

BPSK Modulator Design

Specification:

- Spread binary sequence is the input to the system
- Oscillator carrier sine wave of frequency, Fc ≈ 700 KHz

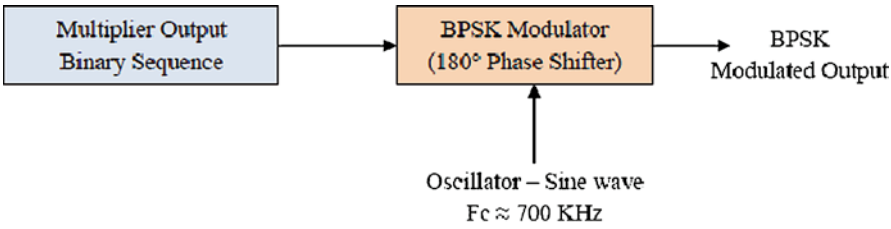

Fig. 2.7 Block diagram of BPSK modulator

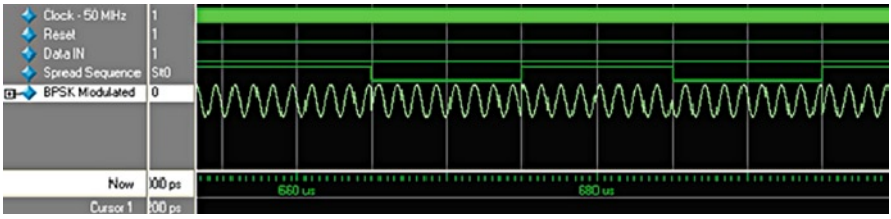

Fig. 2.8 Simulation results for DS-SS transmitter system

Design:

The BPSK modulator is designed using the spread binary sequence as the input to the system and the carrier frequency F_c. The logic is implemented in such a way that the phase of the sine wave is shifted by 180° whenever the input binary bit changes.

The block diagram of the BPSK Modulator as per the design is shown in Fig. 2.7.

2.1.2.3 Simulation Results for DS-SS Transmitter

The DS-SS transmitter is designed using Verilog HDL. Functional verification and simulation is done using ModelSim. The simulation results for DS-SS transmitter is shown in Fig. 2.8.

2.1.3 Receiver for Direct Sequence-Spread Spectrum System

2.1.3.1 Overview of DS-SS Receiver System

In DS-SS receiver, the input to the system is the BPSK modulated signal. This signal would have been affected by noise and other interference in the communication channel. The DS-SS receiver should be designed carefully to reproduce the data signal with least error.

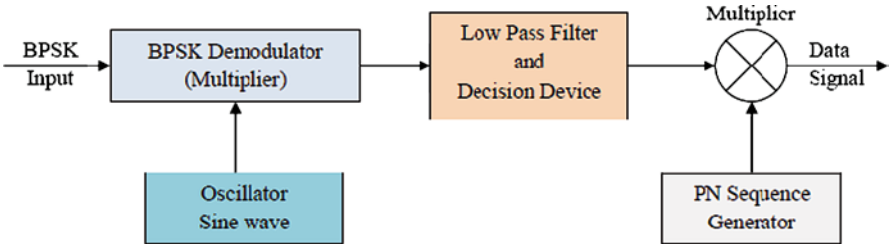

Fig. 2.9 Block diagram of a DS-SS receiver system

The BPSK modulated input signal is multiplied by the locally generated carrier wave by the oscillator. The multiplied signal is then passed through the low pass filter to get low frequency components only. A decision device is used to approximate the signal to binary sequence. This binary sequence is the spread sequence of the data signal.

The most sensitive part of the DS-SS receiver is the synchronization of the locally generated PN sequence and the sequence obtained from the decision device [3]. Even a single bit mismatch may lead to noise instead of the data signal. Suitable technique is used to achieve synchronization and multiply the local PN sequence code with that of the received PN code. The Data signal is obtained after the multiplication process.

In this design, since transmitter and receiver uses common clock on the same FPGA board, the delay in the receiver is considered and modeled appropriately. No specific synchronization technique is used.

The block diagram of a simple DS-SS receiver system is shown in Fig. 2.9.

2.1.3.2 Design of DS-SS Receiver

BPSK Demodulator Design

Specifications:

- BPSK modulated signal is the input to the system
- Oscillator carrier sine wave of frequency, Fc ≈ 700 KHz

The input BPSK signal is multiplied with the carrier sine wave generated from the local oscillator. The design and implementation of the signed BOOTH multiplier is discussed in the following section.

The multiplied output will have higher frequency components and channel noise as well. The high frequency components are eliminated using a suitable Low Pass Filter. Design of rectangular window Low-Pass FIR filter is also discussed in the following section.

The filtered low frequency component will have distortion in the signal. Hence a suitable 'Decision Device' is used to smoothen to binary sequence.

BOOTH Multiplier Design

The BPSK modulated input signal is multiplied with the carrier sine wave generated using the local oscillator. A signed multiplier is designed using BOOTH multiplier algorithm [5].

The BOOTH algorithm used to implement the signed multiplier is as follows:

- The multiplicand X and multiplier Y is loaded into a register. Bit adjustment is made with X and Y so that bits length of X and Y are equal. Bit '0' is padded in order to achieve it
- An accumulator is used to store the result. The length of the accumulator should be twice the length of multiplicand or multiplier. A = 2X or 2Y
- The multiplicand X is loaded into the accumulator from LSB
- A dummy bit of 0 is appended with the accumulator A at the LSB
- During the multiplication operation, the pair of LSB of the accumulator and the dummy bit is considered to follow further arithmetic operations
- Depending on the bit pair obtained in the previous step, following operations are performed:
  - "00" – Arithmetic shift right of the Accumulator.
  - "01" – Add multiplier Y to the Accumulator A (from MSB of A) and Arithmetic shift right of Accumulator.
  - "10" – Subtract multiplier Y from the Accumulator A (from MSB of A) and Arithmetic shift right of Accumulator.
  - "11" – Arithmetic shift right of the Accumulator.

  Shift operations are performed along with dummy bit.
- The above operations are continued till MSB of multiplicand X is shifted off from the accumulator A.

In this section, 5-bit signed BOOTH multiplier is designed and implemented.

Low Pass Filter and Decision Device Design

Specifications:

- The multiplied output from the BPSK demodulator is the input to this system
- A Low Pass Filter with cutoff frequency, f = 105 KHz
- Oscillator carrier wave sampling rate, Fs = 25 MHz

Design:

A Rectangular window FIR filter is designed with a cutoff frequency, f = 105 KHz. Let the length of impulse response for the filter, N = 2.
The desired response of the ideal Low-pass filter is given by,

$$H_d(e^{jw}) = 1, 0 \le f \le 105\ \text{KHz},\ \text{otherwise } 0$$

The normalized angular frequency, $\omega_c = 2\pi F/Fs = 8.4\pi \times 10^{-3}$

$$H_d(e^{jw}) = 1, 0 \le \omega \le \omega_c;\ 0, \omega_c \le \omega \le \pi$$

The filter coefficients are given by,

$$h_d(n) = \sin(8.4\pi \times 10^{-3}\ N)/(\pi N),\ \text{where } N \neq 0.$$

Therefore, the filter coefficients are,

$$h(0) = 8.40 \times 10^{-3} \text{ and } h(1) = 8.39 \times 10^{-3}$$

In this design, one sample of the signal is stored in a register and then it's added with the next sample. The filtered output samples obtained is then processed by the Decision Device. The output of the Decision Device is held High (1) when the output of the filter is non-negative otherwise it's made Low (0).

2.1.3.3 Noise Models and Synchronization

Noise models [1]*:*

- *Multi Path Channels*: In wireless channels there exists often multi path propagation. Since there are more than one path from the transmitter to the receiver. Such multi paths may be due to (a) atmospheric reflection or refraction (b) Reflections from ground, buildings or other objects. Corrective actions are taken to eliminate noise due to multi path channels using appropriate synchronization techniques.
- *Jamming*: The goal of the jammer is to disturb the communication of his adversary. Protection against jamming waveforms is provided by purposely making the information-beating signal occupy a bandwidth far in excess of the minimum bandwidth necessary to transmit it. This has the effect of making the transmitted signal assume a noise-like appearance so as to blend into background. The transmitted signal thus enabled to propagate through the channel undetected by anyone who may be listening. Spread spectrum is a method of "camouflaging" the information bearing signal.

In this design, the noise effect is not modeled as the transmitter and receiver is on the same FPGA board without any air interface.

Synchronization techniques [1]*:*

For proper operation of DS-SS system, the locally generated PN sequence in the receiver is synchronized to the PN sequence of the transmitter generator in both its

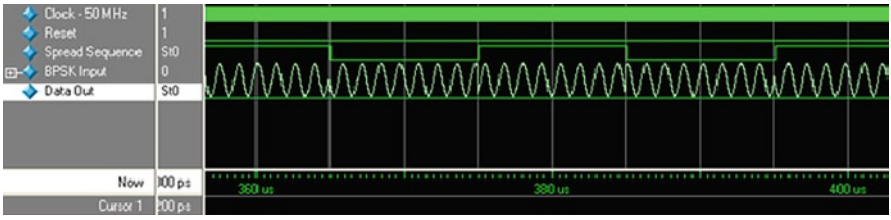

Fig. 2.10 Simulation results for DS-SS receiver system

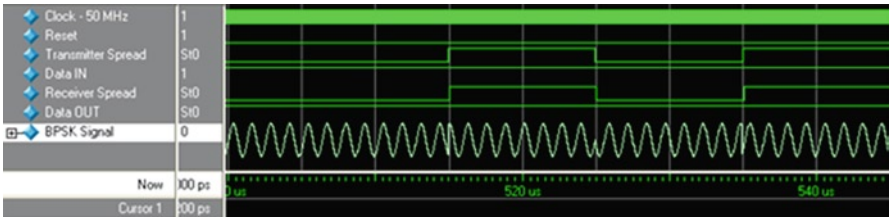

Fig. 2.11 Simulation results for DS-SS modem

rate and position. A slight misalignment in the sequence results in noise instead of data signal.

The process of synchronizing the locally generated PN sequence with the received PN sequence is usually accomplished in two steps. The first step called *acquisition* consists of bringing the two spreading signals into coarse alignment with one another. Once the received PN sequence has been acquired, the second step called *tracking* takes over and continuously maintains the best possible waveform fine alignment by means of a feedback loop. This is essential to achieve highest correlation power and thus highest processing gain (SNR) at the receiver.

In this design, synchronization technique is not modeled since the same clock and PN sequence for receiver and transmitter is implemented on the same FPGA board. A delay of one clock pulse is modeled while multiplying the PN code in the receiver to compensate the filtering delay of one sample.

2.1.3.4 Simulation Results for DS-SS Receiver

The DS-SS receiver is designed using Verilog HDL [6]. Functional verification and simulation is done using ModelSim.

The simulation results for DS-SS receiver is shown in Fig. 2.10.

The simulation results for DS-SS modem is shown in Fig. 2.11. The synthesis report obtained from Xilinx ISE is also shown in Fig. 2.12. The modem can operate at a maximum frequency of 64 MHz on Xilinx Spartan 2E FPGA.

```
=========================================================
HDL Synthesis Report

Macro Statistics
# Adders/Subtractors                                   : 17
 10-bit adder                                          : 1
 11-bit adder                                          : 10
 32-bit adder                                          : 4
 5-bit adder                                           : 2
# Counters                                             : 5
 32-bit up counter                                     : 3
 32-bit updown counter                                 : 2
# Registers                                            : 33
 1-bit register                                        : 29
 10-bit register                                       : 2
 5-bit register                                        : 2
# Comparators                                          : 3
 33-bit comparator greatequal                          : 3
# Multiplexers                                         : 5
 11-bit 4-to-1 multiplexer                             : 5
# Xors                                                 : 8
 1-bit xor2                                            : 8

=========================================================
```

Fig. 2.12 Synthesis report for DS-SS modem

2.2 FIR Filter Design

2.2.1 Concepts of FIR Filter

A discrete-time filter produces a discrete-time output sequence for the discrete-time input sequence. In the Finite Impulsive Response (FIR) system, the impulse response sequence is of finite duration, i.e. it has a finite number of non-zero terms and hence the filter coefficients are also constant. The response of the FIR filter depends only on the present and past input samples (a causal system). Thus making the system always stable.

The difference equation for length 'M' FIR filter is given by [4],

$$y(n) = b_0 \times (n) + b_1 \times (n-1) + b_2 \times (n-2) + b_3 \times (n-3) + \ldots . b_{M-1} \times (n-M+1)$$

$$Y(n) = \sum_{K=0}^{M-1} b_k \times (n\text{-}K)$$

where, $[b_k]$ is the set of filter coefficients.

Some of the important characteristics of FIR digital filter are as follows [4]:

- They can have an exact linear phase
- They are always stable
- The design methods are generally linear
- They can be realized efficiently in hardware
- The filter start-up transients have finite duration
- The filter coefficients are constant for the given order of the filter

Table 2.2 Filter coefficients for LP FIR filter with order 16

Transfer function	Coefficients	Transfer function	Coefficients
h(0)	0.0328	h(8)	0.5763
h(1)	0.0816	h(9)	–0.0550
h(2)	–0.0065	h(10)	–0.0694
h(3)	–0.0047	h(11)	0.0847
h(4)	0.0847	h(12)	–0.0047
h(5)	–0.0694	h(13)	–0.0065
h(6)	–0.0550	h(14)	0.0816
h(7)	0.5763	h(15)	0.0328

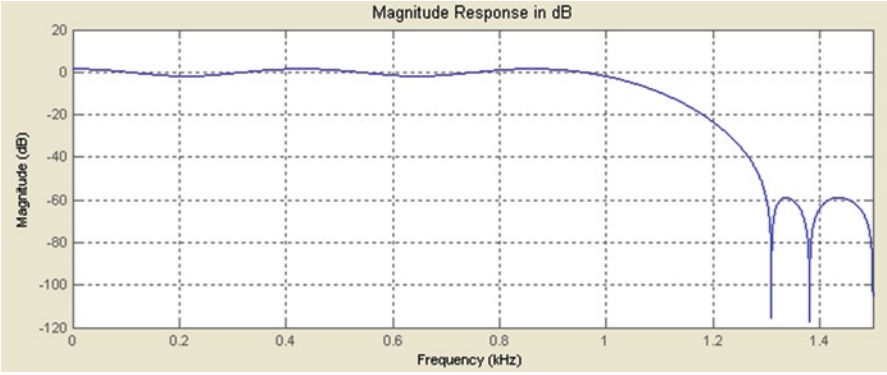

Fig. 2.13 Frequency response (Magnitude) for the designed LP FIR filter

In this section a Low-Pass FIR filter is designed using MatLab FDA tool for the given specifications. Simulated using ModelSim® and implemented using Xilinx® 2E FPGA.

2.2.2 Low Pass FIR Filter Design

The Low Pass FIR (LPF) specifications given in the assignment are,

- $F_{pass} = 1$ KHz, $F_{stop} = 1.3$ KHz
- Pass band ripple = 3 dB, Stop band ripple = 60 dB

Assuming,

- Sampling frequency of the input signal, $F_s = 3$ KHz.
- FIR Filter design method: Equiripple with density factor 16.

The filter coefficients are obtained using MatLab FDA tool for the given specification. The order of the filter, N = 16. The filter coefficients h(n) are as shown in Table 2.2. The frequency response for the given filter specification is shown in Fig. 2.13

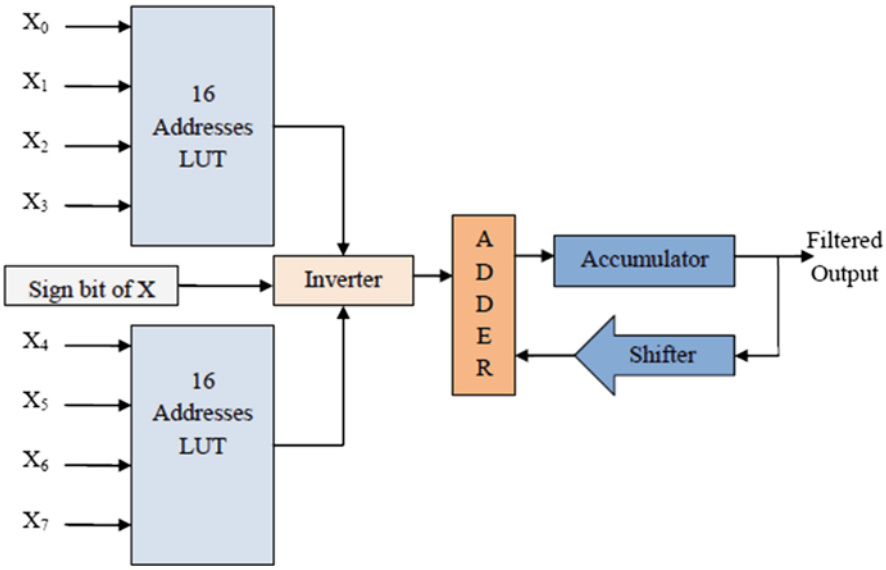

Fig. 2.14 Block diagram to illustrate the functional operation of DA architecture

2.2.3 Distributed Arithmetic Architecture

Distributed Arithmetic (DA) is an important technique to implement digital signal processing functions in FPGAs. DA provides an approach for multiplier-less implementation of DSP systems. It is an algorithm that can perform multiplication with Look-Up Table (LUT) based schemes. DA specifically targets the sum of products (also referred to as the vector dot product) computation that is found in many of the important DSP filtering and frequency transforming functions [7].

In this section, LP FIR filter is designed and implemented using DA architecture. By observing the filter coefficients in Table 2.2, the second half (8–15) of filter coefficients are mirror image of the first half (0–7). Hence the SOP for second half can be accessed from the first half by re-ordering the input bits appropriately. The first half (0–7) coefficients can be broken into two parts and SOP can be calculated and stored in two different blocks. Hence, two LUTs of length 16 are sufficient to store the SOP for the obtained filter coefficients.

The basic functional operation of DA architecture is shown in Fig. 2.14.

2.2.4 Simulation and Synthesis Results

The LP FIR filter is designed using Verilog HDL. The design is simulated using ModelSim®. The impulse response for the LP FIR filter system is shown in Fig. 2.15. In this design, fixed point representations of real numbers are used. Filtered output

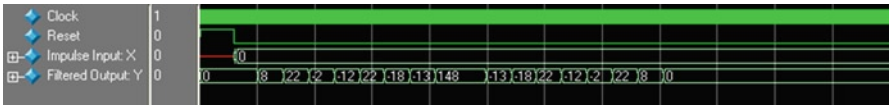

Fig. 2.15 Simulation results for impulse response for the LP FIR filter system

```
=========================================================================
HDL Synthesis Report

Macro Statistics
# ROMs                                          : 4
 16x13-bit ROM                                  : 4
# Adders/Subtractors                            : 4
 13-bit adder                                   : 3
 22-bit adder carry in                          : 1
# Counters                                      : 1
 4-bit up counter                               : 1
# Registers                                     : 20
 1-bit register                                 : 1
 10-bit register                                : 16
 13-bit register                                : 1
 22-bit register                                : 2
# Multiplexers                                  : 16
 1-bit 10-to-1 multiplexer                      : 16
# Logic shifters                                : 2
 22-bit shifter logical left                    : 1
 32-bit shifter logical left                    : 1

=========================================================================
```

Fig. 2.16 HDL synthesis report for LP FIR filter design

values have lower 8 bits representing decimal part. Hence the exact filtered output values from the simulation results are calculated as follows:

$$Y = \left(8, 22, -2, -12, 22, -18, -13, 148, 148, -13, -18, 22, -12, -2, 22, 8\right) / 2^8$$

$$Y = (0.0312, 0.8593, -0.0078, -0.0468, 0.8593, -0.0703, -0.0507, 0.5781,$$
$$0.5781, -0.0507, -0.0703, 0.8593, -0.0468, -0.0078, 0.8593, 0.0312)$$

The design is synthesized and implemented on Xilinx® Spartan 2E FPGA. The HDL synthesis report is shown in Fig. 2.16.

2.3 Discrete Cosine Transform Algorithms

2.3.1 Concepts of DCT

The Discrete Cosine Transform (DCT) is a technique that converts a spatial domain waveform into its constituent frequency components as represented by a set

of coefficients. The process of reconstructing a set of spatial domain samples is called the Inverse Discrete Cosine Transform (IDCT). The equation for 1-D N-point DCT is given by [8],

$$X(k) = \alpha(k)\sum_{n=0}^{N-1} x(n)\cos[\frac{\pi(2n+1)k}{2N}] \qquad 0 \le k \le N-1$$

where,

$$\alpha(0) = \sqrt{\frac{1}{N}},\ \alpha(k) = \sqrt{\frac{2}{N}}\ for\ \ 1 \le k \le N-1$$

One-Dimensional DCT has most often been used in two-dimensional DCT by employing the row-column decomposition which makes it suitable for hardware implementation. Typically the DCT coefficients produced have most of the block's energy in a few frequency domain elements and hence quantization and coding is applied after DCT to provide lossless as well as lossy actual compression [8].

For data compression of image/video frames, usually a block of data is converted from spatial domain samples to another domain (usually frequency domain) which offers more compact representation. DCT technique is used in a wide range of signal and image processing applications. Some of the most popular applications are [8],

- JPEG and JPEG2000 image compression standards
- MPEG digital video standards
- H.261 and H.263 video conferencing standards
- Progressive Image Transmission (PIT) systems: teleconferencing, medical diagnostic imaging and security services

2.3.2 DCT Architectures on FPGA

The DCT can be implemented on FPGA using various architectures. Some of the popular one's reported in [9] are discussed below:

- *Distributed Arithmetic*: The N-points DCT can be considered as N parallel filters. The DCT on the array requires N shift registers for parallel-to-serial conversion, N LUT memories and N shift-accumulators. All the N memories receive the same address. One shift-register and a shift-accumulator are each mapped to an add-shift cluster, while the LUT is mapped to a part of a memory cluster.

  Area usage: 8 shift registers + 8 ROMs + 8 Accumulators
- *Mixed ROM*: The 8-point 1D-DCT can be expressed as the product of an 8 × 8 matrix by an eight element column vector. Through algebraic manipulations, this matrix can be reduced to 4 × 4 matrix. Hence, the number of words per ROM is reduced to only 16 but some overhead has been incurred in the form of adders to calculate the address of the ROMs.

  Area usage: 4 adders + 4 subtractions + 8 shift registers + 8 accumulators + 8 ROMs

- *CORDIC Rotator based:* The DCT computation is done using CORDIC rotator [10]. Since the memory is an integral part of the DA, and ROM size increases exponentially with respect to vector size N. Many techniques have been developed for reducing the size of ROM. The CORDIC algorithm reformulates the 1-D DCT so that the ROM size is reduced to a fix size of four words, independent of the bandwidth of the input data. The DA functionality is implemented by converting parallel data to serial through shift registers and using this data to formulate the address of the memories. This implementation requires 6-CORDIC and 16 butterfly adders for an 8-point 1-D DCT. The CORDIC rotators are implemented through ROM and shift accumulators, while butterfly adders are implemented through add-shift clusters [11].

  Area usage: 8 adders + 8 subtractions + 8 shift registers + 12 accumulators + 12 ROMs
- *Skew circular convolution*: This technique starts with re-ordering the input sequences. Then skew circular convolutions are performed on the reordered inputs, which give odd-indexed transformed sequence. The transformed sequences are re-ordered for the proper output sequences.

  Area usage: 4 adders + 4 subtractions + 8 shift registers + 8 accumulators + 8 ROMs

2.3.3 Scaled 1-D 8-Point DCT Architecture

Since using LUTs results in a very efficient and regular structure suitable for VLSI implementation, especially on the FPGAs, there has been great interest in developing similar kind of LUT based DCT architecture. The Scaled DCT architecture is also a LUT based design. The architecture is primarily designed by making mathematical and trigonometric manipulation using 1-D 8-point DCT equation on eight input data samples. In this design, LUT based Distributed Arithmetic architecture is used. The basic building blocks of this architecture are [9]:

- 20 butterfly adders
- 12 shift registers
- 10 LUTs

The constant scale factor (Y0 and Y4) is not considered in this implementation as that can be combined with the quantization constants without requiring any additional hardware such as LUTs. The simplified 1-D 8-point DCT equations are as shown below:

$$Y_0 = \left[\sqrt{2} \times (X_0 + X_1 + X_2 + X_3 + X_4 + X_5 + X_6 + X_7)\right] / 4$$

$$Y_1 = \left[(X_0 - X_7) \times A + (X_1 - X_6) \times B + (X_2 - X_5) \times C + (X_3 - X_4) \times D\right] / 2$$

$$Y_2 = \left[(X_0 + X_7 - X_3 - X_4) \times E + (X_1 + X_6 - X_2 - X_5) \times F\right] / 2$$

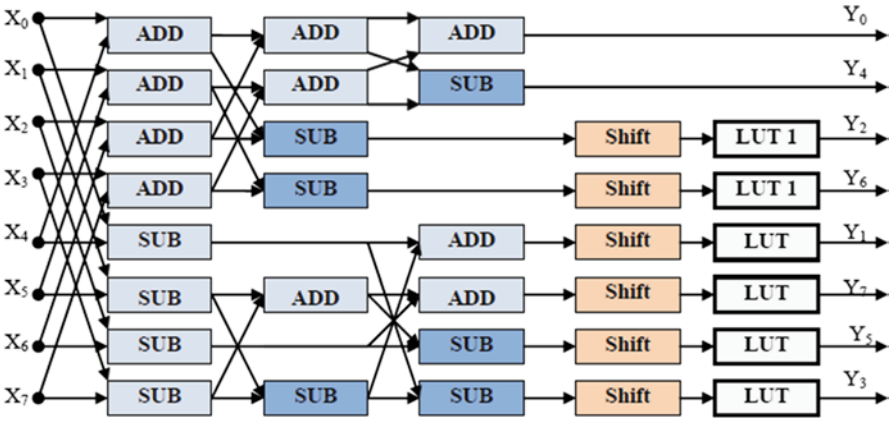

Fig. 2.17 Block diagram of scaled DCT architecture

$$Y_3 = \left[(X_0 - X_7)\times B + (X_6 - X_1)\times D + (X_5 - X_2)\times A + (X_4 - X_3)\times C\right]/2$$

$$Y_4 = \left[\sqrt{2}\times(X_0 - X_1 - X_2 + X_3 + X_4 - X_5 - X_6 + X_7)\right]/2$$

$$Y_5 = \left[(X_0 - X_7)\times C + (X_6 - X_1)\times A + (X_2 - X_5)\times D + (X_3 - X_4)\times B\right]/2$$

$$Y_6 = \left[(X_0 + X_7 - X_3 - X_4)\times F + (X_2 + X_5 - X_1 - X_6)\times E\right]/2$$

$$Y_7 = \left[(X_0 - X_7)\times D + (X_6 - X_1)\times C + (X_2 - X_5)\times B + (X_4 - X_3)\times A\right]/2$$

For N = 8,

$A = \cos(\pi/16)$
$B = \cos(3\pi/16)$
$C = \cos(5\pi/16)$
$D = \cos(7\pi/16)$
$E = \cos(\pi/8)$
$F = \cos(3\pi/8)$

The constant values A, B, C, D, E and F that is required to be multiplied with input X is performed by LUT based Distributed Arithmetic architecture. The block diagram of Scaled DCT architecture for 1-D 8-point samples is shown in Fig. 2.17.

2.3.4 Simulation and Synthesis Results

In this section, 1-D 8-point DCT is designed using Scaled DCT architecture and coded in Verilog HDL. The design is simulated using ModelSim®. The DCT for the input samples, X = (4, 2, 8, 4, 4, 6, 6, 6) is as shown in Fig. 2.18.

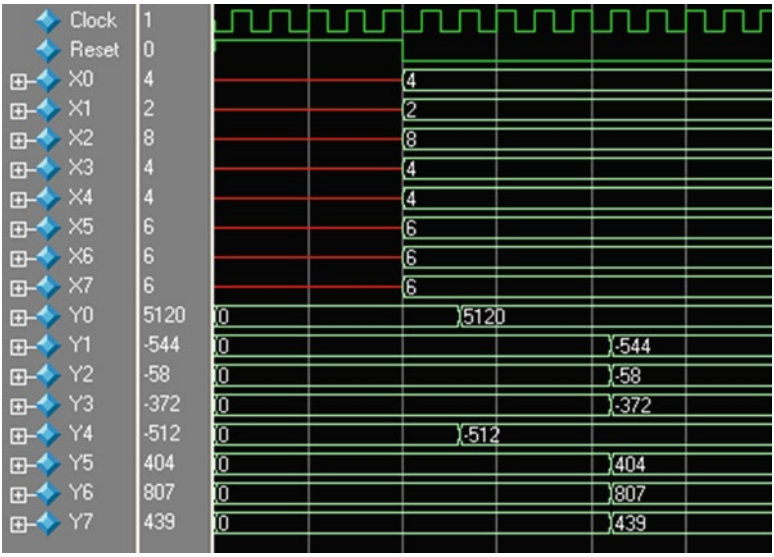

Fig. 2.18 Simulation results for 1-D 8-point DCT

$$Y = (5120/\sqrt{2}, -544, -58, -372, -512/\sqrt{2}, 404, 807, 439)$$

In this design, fixed point representations of real numbers are used. DCT output values have lower eight bits representing decimal part of DCT output. Hence the exact DCT output values from the simulation results are calculated as follows:

$$Y = (5120/\sqrt{2}, -544, -58, -372, -512/\sqrt{2}, 404, 807, 439)/2^8$$

$$Y = (14.1421, -2.0882, -0.2242, -1.4221, -1.4142, 1.6011, 3.1543, 1.7475)$$

This design is implemented on Xilinx® Spartan 2E FPGA. The HDL [13] synthesis report is shown in Fig. 2.19.

2.4 Convolution Codes and Viterbi Decoding

2.4.1 Concepts of Convolution Codes

Forward Error Correction (FEC) technique is used to improve the capacity of channel by adding some carefully designed redundant information to the data that is transmitted over the communication channel. The process of adding this redundant information is known as *channel coding*.

```
=========================================================================
HDL Synthesis Report

Macro Statistics
# ROMs                                                : 10
 4x16-bit ROM                                         : 10
# Adders/Subtractors                                  : 31
 16-bit adder                                         : 13
 4-bit adder carry out                                : 4
 5-bit adder                                          : 10
 5-bit adder carry out                                : 3
 6-bit adder carry out                                : 1
# Counters                                            : 1
 4-bit up counter                                     : 1
# Registers                                           : 113
 1-bit register                                       : 91
 16-bit register                                      : 8
 5-bit register                                       : 14
# Multiplexers                                        : 10
 1-bit 5-to-1 multiplexer                             : 10
# Logic shifters                                      : 12
 16-bit shifter logical left                          : 12

=========================================================================
```

Fig. 2.19 HDL synthesis report for 1-D 8-point DCT

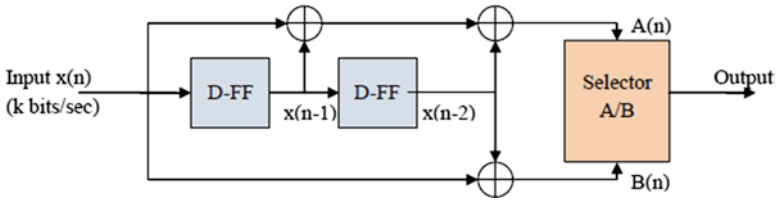

Fig. 2.20 Block diagram of convolutional encoder for a rate ½., constraint length K = 3

Convolutional coding and Block coding are the two major forms of channel coding. Convolutional codes operate on serial data, one or a few bits at a time. Block codes operate on relatively large message blocks. There are a variety of useful convolutional and block codes, and a variety of algorithms for decoding the received coded information sequences to recover the original data. Convolutional encoding with Viterbi decoding is a FEC technique that is particularly suited to a channel in which the transmitted signal is corrupted mainly by Additive White Gaussian Noise (AWGN) [12].

The technique of convolutional coding transforms a binary message into a sequence of symbols to be transmitted. Upon reception, the received information must be related back to the original message bits. If there are no errors the process of decoding is readily accomplished. In general, convolutional coding techniques are applied to very long messages, such as the continuous stream of data from a satellite television transmitter.

A convolutional encoder with two shift registers is shown in Fig. 2.20.

Table 2.3 State transition table for the convolutional encoder

Current state	Output symbols, if input = 0	Output symbols, if input = 1
00	00	11
01	11	00
10	10	01
11	01	10

The system block diagram can be expressed with the following equations:

$$A(n) = x(n) + x(n-1) + x(n-2)$$
$$B(n) = x(n) + x(n-2)$$

The basic building components of the convolutional encoder are flip-flops comprising the shift registers and Exclusive-OR gates comprising the associated Modulo-Two adders. The number of shift registers in the encoder generating the encoded sequence determines the capability of the decoder to detect and correct number of bit errors received on the receiver in the obtained encoded sequence of data.

In this encoder, data bits are provided at a rate of 'k' bits per second. Channel symbols are output at the rate of $n = 2k$ symbols per second. The constraint length $K = 3$ is the length of convolutional encoder, i.e., how many k-bit stages are available to feed the combinatorial logic that produces the output symbols. The input bit is stable during the encoder cycle. The encoder cycle starts when an input clock edge occurs. When the input clock edge occurs, the output of the left-hand flip-flop is clocked into the right-hand flip-flop, the previous input bit is clocked into the left-hand flip-flop and a new input bit becomes available. Then the outputs of the upper and lower modulo-two adders become stable. The output selector cycles through two states. In the first state, it selects and outputs the output of the upper modulo-two adder. In the second state, it selects and outputs the output of the lower modulo-two adder.

The state transition table that lists the channel output symbols, given the current state and the input data is shown in Table 2.3.

2.4.2 Viterbi Decoder

A Viterbi decoder uses the Viterbi algorithm for decoding bit stream that has been encoded using Convolutional codes. There are other algorithms for decoding a convolutional encoded stream (Ex: Fanon algorithm). The Viterbi algorithm is the most resource-consuming but it does the maximum likelihood decoding [12]. Viterbi decoding has the advantage that it has a fixed decoding time. It is well suited for hardware decoder implementation. But its computational requirements grow exponentially as a function of constraint length. So it is usually limited in practice to constraint lengths of $K \leq 10$.

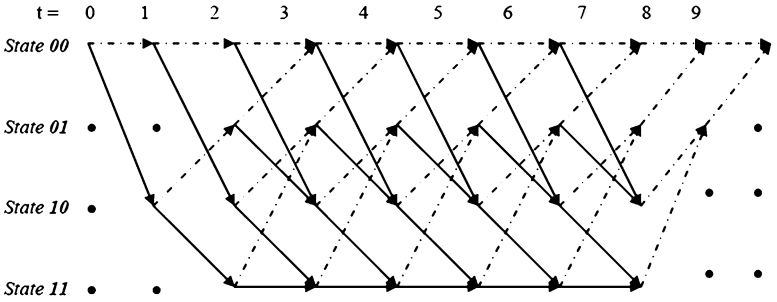

Fig. 2.21 Trellis diagram for Viterbi decoding with encoder rate ½ and K = 3

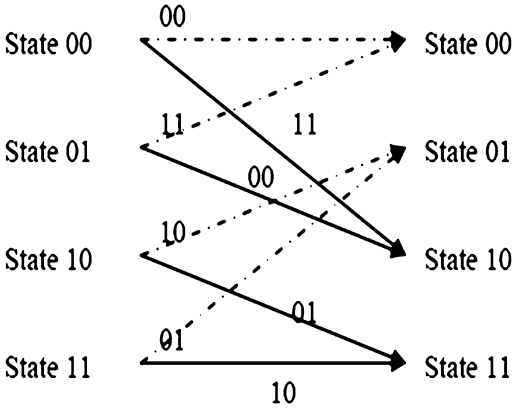

Fig. 2.22 State transitions from one state to the next state

The most important concept to aid in understanding the Viterbi algorithm is the Trellis diagram. The Trellis diagram for the convolutional encoder rate ½, constraint length K = 3 is shown in Fig. 2.21.

The four possible states of the encoder are depicted as four rows of horizontal dots. There is one column of four dots for the initial state of the encoder and one for each time instant during the message. For a 4-bit message with two encoder memory flushing bits, there are six time instants in addition to t = 0, which represents the initial condition of the encoder. The solid lines connecting dots in the diagram represent state transitions when the input bit is a one. The dotted lines represent state transitions when the input bit is a zero. The expanded version of the transition between one time instant to the next is shown in Fig. 2.22. Notice the correspondence between the arrows in the Trellis diagram and the state transition diagram. Since the initial condition of the encoder is State 00, and the two memory flushing bits are zeros, the arrows start out at State 00 and end up at the same state [12].

Each time when a pair of channel symbols is received, the metric- Hamming distance between the received channel symbol pair and the possible channel symbol pairs is calculated for each state. The Hamming distance is computed by simply counting how many bits are different between the received channel symbol pair and the possible

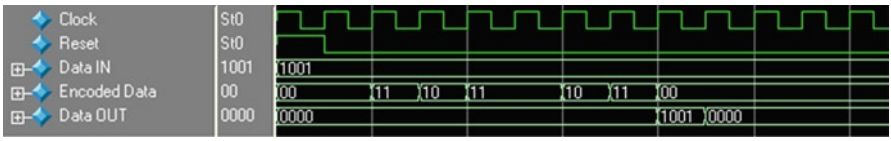

Fig. 2.23 Simulation results for Viterbi decoding with no error in received channel data

channel symbol pairs. The results can only be zero, one, or two. The metrics computed at each time instant for the paths between the states at the previous time instant and the sates at the current time instant are called *branch* metrics. For the first time instant, the results are stored as "accumulated error metric" values associated with the states. For the second time instant onwards, the accumulated error metrics will be computed by adding the previous accumulated error metrics to the current branch metrics. The process is continued for k + m symbols (for k bits message and m shift registers). The smallest accumulated error metric in the final state indicates how many channel symbol errors occurred. This survival path which has the least accumulated error metric is selected. Original message bits are recreated by interpreting the bits from the solid and dotted arrows from the survival path in the Trellis diagram. The two flushing bits at the end are discarded from the recreated message bits.

In this section, Viterbi decoder for 4-bit message is designed using Viterbi algorithm [12].

- Four registers of 6-bit width are used to store the survival path at each state transition.
- Four registers of 4-bit width are used to store the accumulated error metrics at each state.
- At the end of the last state, the survival path having the least accumulated error metrics is used to reproduce the estimated input message bits from the survival path register.

2.4.3 Simulation and Synthesis Results

In this section, Convolutional encoder is designed using two shift-registers and Viterbi decoder is designed using Accumulated Error Metrics algorithm. The design is simulated using ModelSim®.

Assuming the input data to the convolutional encoder is x = (1001), the encoded sequence is, e = (11 10 11 11 10 11). Following different cases are simulated to test the Viterbi decoder design:

1. No error in the received data from the channel. The simulation result for this case is shown in Fig. 2.23.
   Received data: 11 10 11 11 10 11
2. One bit error in the received data from the channel. The simulation result for this case is shown in Fig. 2.24.

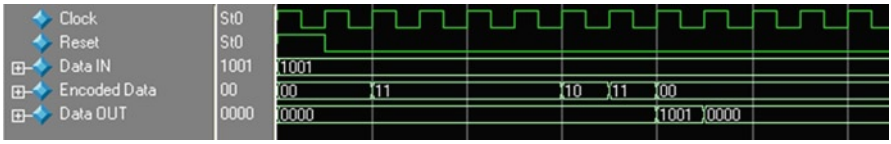

Fig. 2.24 Simulation results for Viterbi decoding with one bit error in received channel data

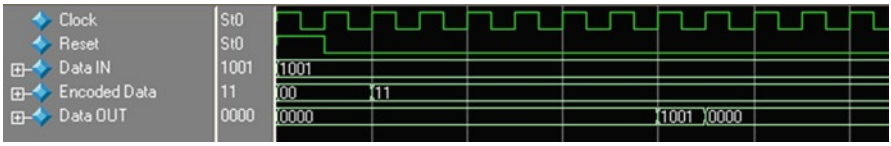

Figure 2.25 Simulation results for Viterbi decoding with two bits error in received channel data

```
================================================================================
HDL Synthesis Report

Macro Statistics
# ROMs                                                 : 4
 4x10-bit ROM                                         : 2
 4x12-bit ROM                                         : 2
# Adders/Subtractors                                   : 16
 3-bit adder                                          : 16
# Counters                                             : 1
 3-bit up counter                                     : 1
# Registers                                            : 17
 1-bit register                                       : 8
 3-bit register                                       : 4
 4-bit register                                       : 1
 6-bit register                                       : 4
# Comparators                                          : 13
 3-bit comparator greater                             : 4
 3-bit comparator lessequal                           : 9
# Multiplexers                                         : 22
 1-bit 4-to-1 multiplexer                             : 18
 3-bit 4-to-1 multiplexer                             : 4
# Xors                                                 : 2
 1-bit xor2                                           : 2

================================================================================
```

Fig. 2.26 HDL synthesis report for convolutional encoder and Viterbi decoder

Received data: 11 11 11 11 10 11

3. Two bits error in the received data from the channel. The simulation result for this case is shown in Fig. 2.25.
   Received data: 11 11 11 11 11 11

This design is implemented on Xilinx® Spartan 2E FPGA. The HDL synthesis report is shown in Fig. 2.26.

Appendix

A. Verilog HDL code for PN sequence generator

```
module pn_sequence(clk,rst,pnOUT);

input clk;
input rst;
output pnOUT;

// Instantiate PN sequence clock : 100 KHz
pn_clock pn_clock_seq(clk,rst,pnCLK);

// Generate PN Sequence : m = 4
reg pnOUT;
reg [3:0]shift4reg;

always @(posedge pnCLK, posedge rst)
begin
    if(rst)
    begin
        pnOUT <= 1'b0;
       shift4reg[3:0] <= 4'b0;
    end
    else
    begin
        shift4reg[0] <= ~(shift4reg[0] ^ shift4reg[3]);
        shift4reg[1] <= shift4reg[0];
        shift4reg[2] <= shift4reg[1];
        shift4reg[3] <= shift4reg[2];
        pnOUT <= shift4reg[3];
    end
end
endmodule

// PN Sequence Clock  : 100 KHz
module pn_clock(clk,rst,pnCLK);
input clk;
input rst;
output pnCLK;

reg pnCLK;
integer pnCLKCnt;

always @(posedge clk, posedge rst)
begin
    if(rst)
    begin
        pnCLK <= 1'b0;
        pnCLKCnt = 0;
    end
    else
    begin
        pnCLKCnt = pnCLKCnt + 1;
        if(pnCLKCnt >= 250)
        begin
            pnCLK <= !pnCLK;
            pnCLKCnt = 0;
        end
```

```
          else
              pnCLK <= pnCLK;
      end
  end
endmodule
```

B. Verilog HDL code for LP FIR filter using Distributed Arithmetic Architecture

```
  module fir_filter(clk,rst,y,x0);

  input clk;
  input rst;
  input [9:0]x0;
  output [21:0]y;
  reg [21:0]y;
  reg [9:0]r1,r2,r3,r4,r5,r6,r7,r8,r9,r10,r11,r12,r13,r14,r15,r16;

  reg [12:0]temp;
  reg [21:0]calc;
  reg [3:0]count;
  reg load;


  always @(posedge clk, posedge rst)
  begin
     if(rst)
     begin
      r1<=0; r2<=0; r3<=0; r4<=0; r5<=0; r6<=0; r7<=0; r8<=0;
      r9<=0; r10<=0; r11<=0; r12<=0; r13<=0; r14<=0; r15<=0; r16<=0;
      temp = 0;
      calc = 0;
      count = 0; load = 1;
      y = 0;
     end
     else
     begin
        if(load)
        begin
            r1 <= x0;r2 <= r1;r3 <= r2;r4 <= r3;
           r5 <= r4;r6 <= r5;r7 <= r6;r8 <= r7;
            r9 <= r8;r10<= r9;r11<= r10;r12<= r11;
            r13<= r12;r14<= r13;r15<= r14;r16<= r15;
            load = 0; count = 0;
        end
     else
       begin
temp = block1Value( r1[count], r2[count], r3[count], r4[count]) +
       block2Value( r5[count], r6[count], r7[count], r8[count]) +
       block2Value(r12[count],r11[count],r10[count], r9[count]) +
       block1Value(r16[count],r15[count],r14[count],r13[count]);
```

```
       if((count == 4'b1001) &&
           (r1[count]  ||  r2[count]  ||  r3[count]  ||  r4[count]  ||
            r5[count] || r6[count] || r7[count] || r8[count] ||
            r12[count]|| r11[count]|| r10[count]|| r9[count] ||
            r16[count]|| r15[count]|| r14[count]|| r13[count]))

          // For negative numbers
          calc = calc + (~(temp << count) + 1);
       else
          calc = calc + (temp << count);
            if(calc[12] == 1'b1)
                calc[21:13] = 9'b111111111;
            else
                calc[21:13] = 9'b000000000;

                if(count == 4'b1001)
                begin
                    y = calc;
                    calc = 0;
                    temp = 0;
                    load = 1;
                end
                else
                 count = count + 1;
       end
   end
end


function [12:0]block1Value; // LUT_1
input a1,a2,a3,a4;
begin
    case({a1,a2,a3,a4})
        4'b0000 : block1Value = 13'b0000000000000;
        4'b0001 : block1Value = 13'b1111111110100;
        4'b0010 : block1Value = 13'b1111111111110;
        4'b0011 : block1Value = 13'b1111111110010;
        4'b0100 : block1Value = 13'b0000000010110;
        4'b0101 : block1Value = 13'b0000000001001;
        4'b0110 : block1Value = 13'b0000000010011;
        4'b0111 : block1Value = 13'b0000000000111;
        4'b1000 : block1Value = 13'b0000000001000;
        4'b1001 : block1Value = 13'b1111111111100;
        4'b1010 : block1Value = 13'b0000000000111;
        4'b1011 : block1Value = 13'b1111111111011;
        4'b1100 : block1Value = 13'b0000000011100;
        4'b1101 : block1Value = 13'b0000000010001;
        4'b1110 : block1Value = 13'b0000000011010;
        4'b1111 : block1Value = 13'b0000000010000;
        default : block1Value = 13'b0000000000000;
    endcase

    //$display("blockValue 1 : %b\n",block1Value);
end
endfunction
```

```
    function [12:0]block2Value; // LUT_2
    input b1,b2,b3,b4;
    begin
        case({b1,b2,b3,b4})
            4'b0000 : block2Value = 13'b0000000000000;
            4'b0001 : block2Value = 13'b0000010010100;
            4'b0010 : block2Value = 13'b1111111110011;
            4'b0011 : block2Value = 13'b0000010111000;
            4'b0100 : block2Value = 13'b1111111101110;
            4'b0101 : block2Value = 13'b0000010000010;
            4'b0110 : block2Value = 13'b1111110000000;
            4'b0111 : block2Value = 13'b0000001110100;
            4'b1000 : block2Value = 13'b0000000010110;
            4'b1001 : block2Value = 13'b0000011001001;
            4'b1010 : block2Value = 13'b0000000001000;
            4'b1011 : block2Value = 13'b0000010011000;
            4'b1100 : block2Value = 13'b0000000000100;
            4'b1101 : block2Value = 13'b0000010010110;
            4'b1110 : block2Value = 13'b1111111110100;
            4'b1111 : block2Value = 13'b0000010001010;
            default : block2Value = 13'b0000000000000;
        endcase
            //$display("blockValue 2 : %b\n",block2Value);
    end
    endfunction
endmodule
```

C. Verilog HDL code for Convolutional encoder

```
    module Conv_Encoder(clk, rst, dataIn, dataOut);

    input clk;
    input rst;
    input [3:0]dataIn;
    output [1:0]dataOut;

    reg [1:0]dataOut;
    reg [3:0]inBit;
    reg ff1,ff2;


    always@(posedge clk, posedge rst)
    begin
        if(rst)
        begin
            dataOut <= 2'b0;
            ff1 <= 1'b0;
            ff2 <= 1'b0;
            inBit <= dataIn;
        end
        else
        begin
            ff1 <= inBit[0];
            ff2 <= ff1;
            dataOut[0] <= (inBit[0]^ff2);
            dataOut[1] <= (inBit[0]^ff1)^ff2;
            inBit <= inBit >> 1;
        end
    end
    endmodule
```

References

1. Meel J (1999) Introduction to spread spectrum, Cirius Communications, Belgium
2. Miller A, Gulotta M (2004) PN generators (XAPP211), Xilinx Inc
3. An Introduction to Direct Sequence – Spread Spectrum (2003), Maxim Integrated Products Inc
4. Proakis JG, Manolakis DK (1995) Digital signal processing: principles, algorithm and application, 3rd edn. Prentice Hall, Englewood Cliffs
5. Booth's Algorithm: Multiplication and Division (2010) http://www.scribd.com/doc/3132888/Booths-Algorithm-Multiplication-Division. Accessed Oct 2010
6. Palinitkar S (2003) Verilog HDL: a guide to digital design and synthesis, 2nd edn. Prentice Hall, Palo Alto
7. Grover RS, Shang W, Li Q (2002) A faster distributed arithmetic architecture for FPGAs. In: ACM/SIGDA 10th International symposium on field-programmable gate arrays, Monterey, CA, USA, 24–26 Feb 2002, pp 31–39
8. Marshall D (2001) The discrete cosine transform. Cardiff Schoo of Computer Science & Informatics. http://www.cs.cf.ac.uk/Dave/Multimedia/node231.html. Accessed 10 October 2006
9. Khawan S, Baloch S, Pai A, Ahmed I, Aydin N, Arslan T, Westall F (2004) Efficient implementation of mobile video computations on domain-specific reconfigurable arrays. In: Conference on design, automation and test in Europe, vol 2, Paris, 16–20 Feb 2004, p 21230
10. Meyer-Baese U (2006) Digital signal processing with field programmable gate arrays, 2nd edn. Springer, Berlin/New York
11. Andraka Consulting Group, Inc. (2007) The CORDIC algorithm. http://www.andraka.com/cordic.htm. Accessed 2 April 2007
12. Fleming C (2006) A tutorial on convolutional coding with Viterbi decoding. Spectrum applications. http://home.netcom.com/%7Echip.f/viterbi/tutorial.html. Accessed 10 April 2006
13. Vahid F, Lysecky R (2007) Verilog for digital design. Wiley, Hoboken

http://www.springer.com/978-1-4614-1119-2

VLSI Design
A Practical Guide for FPGA and ASIC Implementations
Chandrasetty, V.A.
2011, XIII, 106 p. 93 illus., Softcover
ISBN: 978-1-4614-1119-2