P-3043

DESIGN AND IMPLEMENTATION OF HDLC PROCEDURES BASED ON FPGA

A PROJECT REPORT

Submitted by

S.KAVITHA (71206106023)

P.SARANYA (71206106044)

D.VANIKIRUTHIKA (71206106056)

in partial fulfillment for the award of the degree

of

BACHELOR OF ENGINEERING

In

ELECTRONICS AND COMMUNICATION ENGINEERING

KUMARAGURU COLLEGE OF TECHNOLOGY, COIMBATORE

ANNA UNIVERSITY:: CHENNAI 600 025

APRIL 2010

ANNA UNIVERSITY: CHENNAI 600 025

BONAFIDE CERTIFICATE

Certified that this project report "**DESIGN AND IMPLEMENTATION OF HDLC PROCEDURES BASED ON FPGA**" is the bonafide work of "**KAVITHA.S, SARANYA.P, VANIKIRUTHIKA.D**" who carried out the project under my supervision.

[illegible]

SIGNATURE

Dr.Rajeswari Mariappan

HEAD OF THE DEPARTMENT

Electronics and
Communication Engineering,
Kumaraguru College
of Technology,
Coimbatore-641006.

[illegible]

SIGNATURE

Dr.Rajeswari Mariappan

PROJECT GUIDE,

HEAD OF THE DEPARTMENT

Electronics and
Communication Engineering,
Kumaraguru College
of Technology,
Coimbatore-641006.

The candidates with University Register Nos. 71206106023, 71206106044,71206106056 was examined by us in the project viva-voce examination held on 15.04.2010.

[illegible]
15.04.2010

INTERNAL EXAMINER

[illegible]

EXTERNAL EXAMINER

ACKNOWLEDGEMENT

1.INTRODUCTION

CHAPTER 1

1. INTRODUCTION

1.1 OBJECTIVE

This project illustrates how to generate Frame Check Sequence (FCS) of HDLC-Cyclic Redundancy Check (CRC) in FPGA. Verification is done by downloading the HDLC modules designed in VHDL into FPGA.

1.2 HDLC OVERVIEW

HDLC (High-level Data Link Control) is a group of protocols or rules for transmitting data between network points (sometimes called nodes). In HDLC, data is organized into a unit called a frame and sent across a network to a destination that verifies its successful arrival. The HDLC protocol also manages the flow or pacing at which data is sent. HDLC is one of the most commonly-used protocols in what is data link layer of the industry communication reference model called Open Systems Interconnection (OSI).

Physical layer is the detailed physical level that involves actually generating and receiving the electronic signals. Network layer is the higher level that has knowledge about the network, including access to router tables that indicate where to forward or send data. On sending, programming in network layer creates a frame that usually contains source and destination network addresses. HDLC encapsulates the network layer frame, adding data link control information to a new, larger frame.

HDLC is a protocol developed by the International Organization for Standardization (ISO). It falls under the ISO standards ISO 3309 and ISO 4335. It has found itself being used throughout the world. It has been so widely implemented because it supports both half duplex and full duplex communication lines, point to point(peer to peer) and multi-point networks, and switched or non-switched channels. The procedures outlined in HDLC are designed to permit synchronous, code-transparent data transmission. Other benefits of HDLC are that the control information is always in the same position, and specific bit patterns used for control differ dramatically from those in representing data, which reduces the chance of errors.

1.3 ORGANISATION OF THE REPORT

This report describes about the implementation of HDLC procedures

Chapter 1: This provides an overview of HDLC. It includes the scope and objective of the project

Chapter 2: It provides an understanding of working mechanism of the encoding and decoding modules

Chapter 3: It gives details about the HDLC frame structure, commands and responses. It also includes modes of operation.

Chapter 4: It provides an understanding of frame check sequence (cyclic redundancy check).

Chapter 5: It gives details about the conversion of data from parallel to serial in encoding module and serial to parallel conversion in decoding module.

Chapter 6: It explains the basics of VHDL hardware description language.

Chapter 7: Coding part is included in this chapter.

Chapter 8: It includes the simulation results.

Chapter 9 :It gives the advantage and future enhacement of this project.

Chapter 10: It outlines the conclusion of the project.

2.PROJECT OVERVIEW

CHAPTER 2

2.PROJECT OVERVIEW

High Level Data Link Control, also known as HDLC, is a bit oriented, switched and non-switched protocol. It is a data link control protocol, and falls within Data Link Layer (DLL) of the Open System Interface (OSI) model

HDLC procedures design contain two module,

1. Encoding-and-sending module.
2. Receiving-and-decoding module.

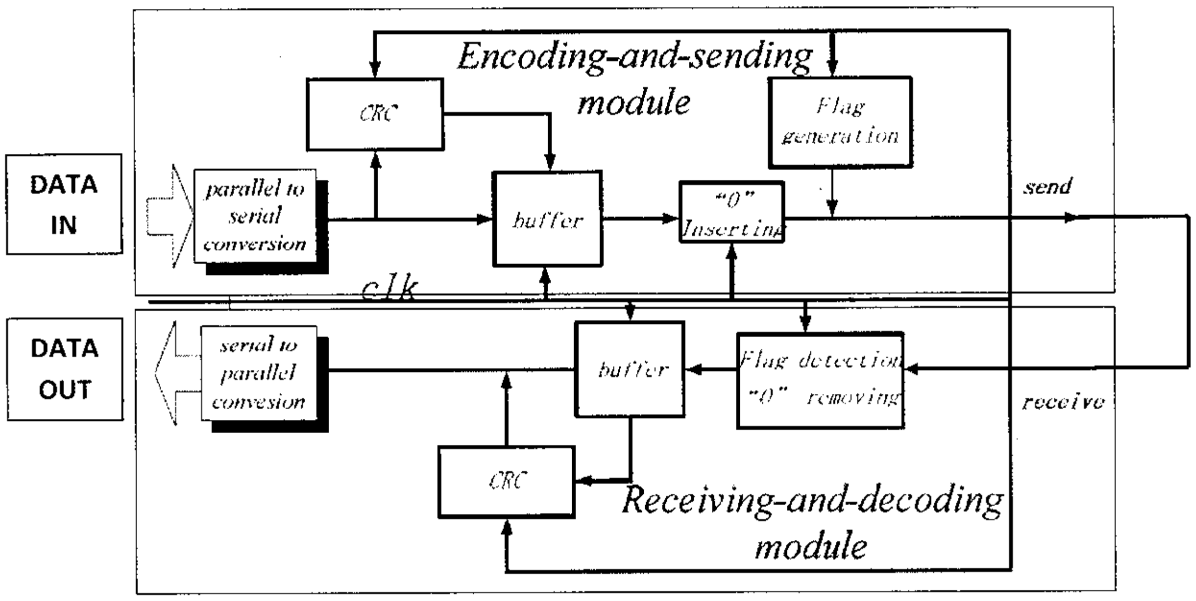
1. Encoding-and-sending module

Encoding-and-sending module includes data (from RAM) parallel-to-serial conversion unit, CRC unit, data buffer unit, "0" inserting unit and flag generation unit.

2. Receiving-and-decoding module

Receiving-and decoding module includes flag detection unit, "0" removing unit, data buffer unit, CRC unit and data (to RAM) serial-to parallel conversion unit. All units work at clock (clk) rising edges synchronously.

2.1 OVERALL BLOCK DIAGRAM



DATA IN
DATA OUT
Encoding-and-sending module
CRC
Flag generation
parallel to serial conversion
buffer
"0" Inserting
send
clk
serial to parallel convesion
buffer
Flag detection "0" removing
receive
CRC
Receiving-and-decoding module

2.2 WORKING

2.2.1 ENCODING-AND-SENDING MODULE

Encoding-and-sending module adopts Finite State Machine (FSM) in VHDL, which can be divided into five states: Idle state, Frame header sending state, User data sending state, CRC codes sending state and Frame end sending state. After being reset or powered up, FSM enters Idle state; once "data sending" command is received, FSM jumps into Frame header sending state, then User data sending state, CRC codes sending state and finally ends with Frame end sending state. After all states are finished, FSM is back to Idle state. When sending data (except for flag codes), a counter in the module counts the number of continuous "1"s. Once the counter's value is up to "5", this module outputs a "0", and the counter is cleared to zeros.

Encoding-and-sending module achieves "0" inserting function. When sending data, we usually send frame header flag not once, but three times. This is to ensure that the receiving-and-decoding module could detect data packet header efficiently, not lose or misjudge it.

2.2.2 RECEIVING-AND-DECODING MODULE

Receiving-and-decoding module also adopts FSM . The FSM includes Frame header detecting state, User data processing state, CRC verifying and comparing state, Storage flag generation state and Data storing state. In Frame header detecting state, FSM detects the frame header flag "7E" in the bit stream with an 8-bit left shift register; after frame header flag "7E" is detected and user data show up, FSM jumps into User data processing state, which includes "0" removing function, CRC secondary verification at the receiver, acquirement of user data and original CRC results from the bit stream and so on. At the time frame end flag is detected, FSM jumps into *CRC* verifying and comparing state. FSM compares the CRC secondary verifying results with the original CRC results, and checks whether they are equal or not to determine data transmission accuracy. Finally, user data and CRC results are stored into RAM.

3.HDLC

CHAPTER 3

3.HDLC

High Level Data Link Control, also known as HDLC, is a bi t oriented, switched and non-switched protocol. It is a data link control protocol, and falls within Data Link Layer (DLL) of the Open System Interface (OSI) model

3.1 HDLC Frame Structure

HDLC uses the term "frame" to indicate an entity of data(or a protocol data unit) transmitted from one station to another. Figure 3.1 below is a graphical representation of a HDLC frame with an information field.

Figure 3.1 An HDLC frame with an information field

Field Name	Size(in bits)
Flag Field(F)	8 bits
Address Field(A)	8 bits
Control Field(C)	8 or 16 bits
Information Field(I)	Variable; Not used in some frames
Frame Check Sequence(FCS)	16 or 32 bits
Closing Flag Field(F)zzz	8 bits

TABLE 3.1 HDLC FIELDS

3.1.1 FLAG FIELD

Every frame on the link must begin and end with a flag sequence field (F). Stations attached to the data link must continually listen for a flag sequence. The flag sequence is an octet looking like 01111110. Flags are continuously transmitted on the link between frames to keep the link active. Two other bit sequences are used in HDLC as signals for the stations on the link. These two bit sequences are:

- Seven 1's, but less than 15 signals an abort signal. The stations on the link know there is a problem on the link.
- 15 or more 1's indicate that the channel is in an idle state.

The time between the transmission of actual frames is called the interframe time fill. The interframe time fill is accomplished by transmitting continuous flags between frames. The flags may be in 8 bit multiples.

HDLC is a code-transparent protocol. It does not rely on a specific code for interpretation of line control. This means that if a bit at position N in an octet has a specific meaning, regardless of the other bits in the same octet. If an octet has a bit sequence of 01111110, but is not a flag field, HLDC uses a technique called **bit-stuffing** to differentiate this bit sequence from a flag field. Once the transmitter detects that it is sending 5 consecutive 1's, in inserts a 0 bit to prevent a "phony" flag.

At the receiving end, the receiving station inspects the incoming frame. If it detects 5 consecutive 1's it looks at the next bit. If it is a 0, it pulls it out. If it is a 1, it looks at the 8th bit. If the 8th bit is a 0, it know an abort or idle signal has been sent. It then proceeds to inspect the following bits to determine appropriate action. This is the manner in which HDLC achieves code-transparency. HDLC is not concerned with any specific bit code inside the data stream. It is only concerned with keeping flags unique.

3.1.2 ADDRESS FIELD

The address field (A) identifies the primary or secondary stations involvement in the frame transmission or reception. Each station on the link has a unique address. In an unbalanced configuration, the A field in both commands and responses refers to the secondary station. In a balanced configuration, the command frame contains the destination station address and the response frame has the sending station's address.

3.1.3 CONTROL FIELD

HDLC uses the control field(C) to determine how to control the communications process. This field contains the commands, responses and sequences numbers used to maintain the data flow accountability of the link, defines the functions of the frame and initiates the logic to control the movement of traffic between sending and receiving stations. There three control field formats:

- **Information Transfer Format**

The frame is used to transmit end-user data between two devices.

- **Supervisory Format**

The control field performs control functions such as acknowledgment of frames, requests for re-transmission, and requests for temporary suspension of frames being transmitted. Its use depends on the operational mode being used.

- **Unnumbered Format**

This control field format is also used for control purposes. It is used to perform link initialization, link disconnection and other link control functions.

3.1.4 INFORMATION FIELD

This field is not always in a HDLC frame. It is only present when the Information Transfer Format is being used in the control field. The information field contains the actually data the sender is transmitting to the receiver.

3.1.5 FRAME CHECK SEQUENCE FIELD

This field contains a 16 bit, or 32 bit cyclic redundancy check. It is used for error detection.

3.2 HDLC STATIONS AND CONFIGURATIONS

HDLC specifies the following three types of stations for data link control:

- Primary Station
- Secondary Station
- Combined Station

3.2.1 PRIMARY STATION

Within a network using HDLC as it's data link protocol, if a configuration is used in which there is a primary station, it is used as the controlling station on the link. It has the responsibility of controlling all other stations on the link (usually secondary stations). Despite this important aspect of being on the link, the primary station is also responsible for the organization of data flow on the link. It also takes care of error recovery at the data link level.

3.2.2 SECONDARY STATION

If the data link protocol being used is HDLC, and a primary station is present, a secondary station must also be present on the data link. The secondary station is under the control of the primary station. It has no ability, or direct responsibility for controlling the link. It is only activated when requested by the primary station. It only responds to the primary station. The secondary station's frames are called responses. It can only send response frames when requested by the primary station.

3.2.3 COMBINED STATION

A combined station is a combination of a primary and secondary station. On the link, all combined stations are able to send and receive commands and responses without any permission from any other stations on the link. Each combined station is in full control of itself, and does not rely on any other stations on the link. No other stations can control any combined station.

3.3 HDLC CONFIGURATIONS

HDLC also defines three types of configurations for the three types of stations:

- Unbalanced Configuration
- Balanced Configuration
- Symmetrical Configuration

3.3.1 UNBALANCED CONFIGURATION

The unbalanced configuration in an HDLC link consists of a primary station and one or more secondary stations. The unbalanced occurs because one stations controls the other stations.

In a unbalanced configurations, any of the following can be used:

- Full - Duplex or Half - Duplex operation
- Point to Point or Multi-point networks

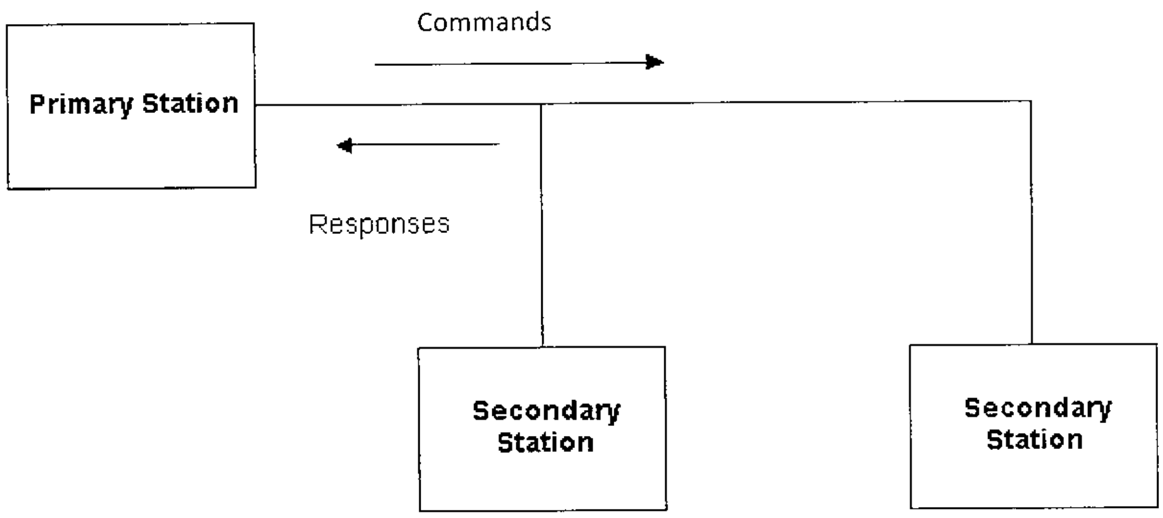


Figure 3.2 An unbalanced configuration

3.3.2 SYMMETRICAL CONFIGURATION

Symmetrical configuration is not widely in use today. It consists of two independent point to point, unbalanced station configurations. In this configurations, each station has a primary and secondary status. Each station is logically considered as two stations.

3.3.3 BALANCED CONFIGURATION

The balanced configuration in an HDLC link consists of two or more combined stations. Balanced configurations can used only the following:

- Full - Duplex or Half - Duplex operation
- Point to Point network.

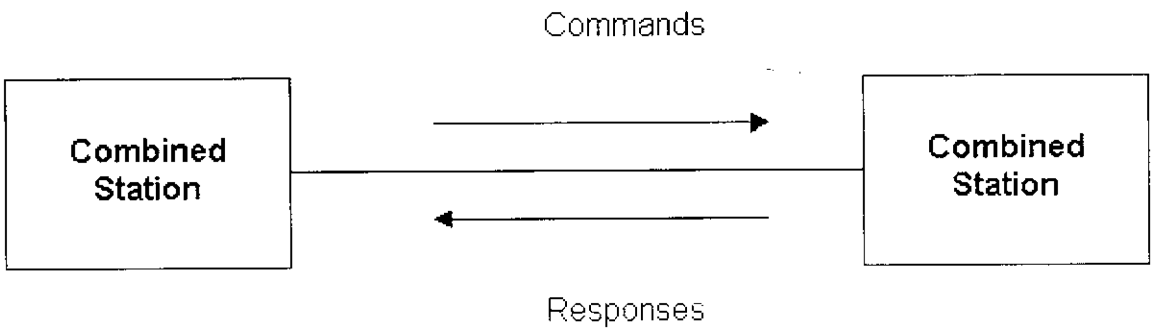


Figure 3.3 A balanced configuration

Each station can act as either primary or secondary. Each of the stations have equal and complimentary responsibility compared to each other.

3.4 HDLC OPERATIONAL MODES

HDLC offers three different modes of operation. The three modes of operations are:

- Normal Response Mode(NRM)
- Asynchronous Response Mode(ARM)
- Asynchronous Balanced Mode(ABM)

3.4.1 NORMAL RESPONSE MODE

This is the mode in which the primary station initiates transfers to the secondary station. The secondary station can only transmit a response when, and only when, it is instructed to do so by the primary station. In other words, the secondary station must receive explicit permission from the primary station to transfer a response. After receiving permission from the primary station, the secondary station initiates it's transmission. This transmission from the secondary station to the primary station may be much more than just an acknowledgment of a frame. It may in fact be more than one information frame. Once the last frame is transmitted by the secondary station, it must wait once again from explicit permission to transfer anything, from the primary station. Normal Response Mode is only used within an unbalanced configuration.

3.4.2 ASYNCHRONOUS RESPONSE MODE

In this mode, the primary station doesn't initiate transfers to the secondary station. In fact, the secondary station does not have to wait to receive explicit permission from the primary station to transfer any frames. The frames may be more than just acknowledgment frames. They may contain data, or control information regarding the status of the secondary station. This mode can reduce overhead on the link, as no frames need to be transferred in order to give the secondary station permission to initiate a transfer. However some limitations do exist. Due to the fact that this mode is Asynchronous, the secondary station must wait until it detects and idle channel before it can transfer any frames. This is when the ARM link is operating at half-duplex. If the ARM link is operating at full-duplex, the secondary station can transmit at any time. In this mode, the primary station still retains responsibility for error recovery, link setup, and link disconnection.

3.4.3 ASYNCHRONOUS BALANCED MODE

This mode uses combined stations. There is no need for permission on the part of any station in this mode. This is because combined stations do not require any sort of instructions to perform any task on the link.Normal Response Mode is used most frequently in multi-point lines, where the primary station controls the link. Asynchronous Response Mode is better for point to point links, as it reduces overhead. Asynchronous Balanced Mode is not used widely today.

The "asynchronous" in both ARM and ABM does not refer to the format of the data on the link. It refers to the fact that any given station can transfer frames without explicit permission or instruction from any other station.

3.4.4 HDLC NON-OPERATIONAL MODES

HDLC also defines three non-operational modes. The three non-operational modes are:

- Normal Disconnected Mode(NDM)
- Asynchronous Disconnected Mode(ADM)
- Initialization Mode(IM)

The two disconnected modes(NDM and ADM) differ from the operational modes in that the secondary station is logically disconnected from the link(note the secondary station is not physically disconnected from the link). The IM mode is different from the operations modes in that the secondary station's data link control program is in need of regeneration or it is in need of an exchange of parameters to be used in an operational mode.

4.FRAME CHECK SEQUENCE

CHAPTER4

4.FRAME CHECK SEQUENCE

The cyclic redundancy check, or CRC, is a technique for detecting errors in digital data, but not for making corrections when errors are detected. It is used primarily in data transmission. In the CRC method, a certain number of check bits, often called a checksum, are appended to the message being transmitted. The receiver can determine whether or not the check bits agree with the data, to ascertain with a certain degree of probability whether or not an error occurred in transmission. If an error occurred, the receiver sends a "negative acknowledgement" (NAK) back to the sender, requesting that the message be retransmitted.

4.1 ERROR CHECKING TECHNIQUES

There are several techniques for generating check bits that can be added to a message. Perhaps the simplest is to append a single bit, called the "parity bit," which makes the total number of 1-bits in the code vector (message with parity bit appended) even (or odd). If a single bit gets altered in transmission, this will change the parity from even to odd (or the reverse). The sender generates the parity bit by simply summing the message bits modulo 2 that is, by *exclusive or*'ing them together.

4.1.1 EXCLUSIVE OR OPERATION

The EX-OR operation appends the parity bit (or its complement) to the message. The receiver can check the message by summing all the message bits modulo 2 and checking that the sum agrees with the parity bit. Equivalently, the receiver can sum all the bits (message and parity) and check that the result is 0 (if even parity is being used). This simple parity technique is often said to detect 1-bit errors. Actually it detects errors in any odd number of bits (including the parity bit), but it is a small comfort to know you are detecting 3-bit errors if you are missing 2-bit errors.

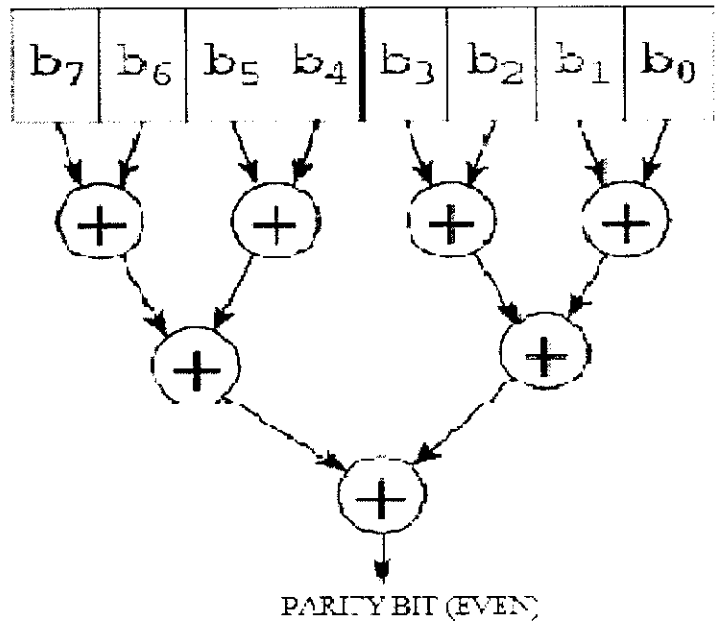


FIGURE 4.1 EX-OR TREE

For bit serial sending and receiving, the hardware to generate and check a single parity bit is very simple. It consists of a single *exclusive or* gate together with some control circuitry. For bit parallel transmission, an *exclusive or* tree may be used, as illustrated in Figure 4.1.

4.1.2 OTHER TECHNIQUES

Other techniques for computing a checksum are to form the *exclusive or* of all the bytes in the message, or to compute a sum with end-around carry of all the bytes. In the latter method the carry from each 8-bit sum is added into the least significant bit of the accumulator. It is believed that this is more likely to detect errors than the simple *exclusive or*, or the sum of the bytes with carry discarded. A technique that is believed to be quite good in terms of error detection, and which is easy to implement in hardware, is the cyclic redundancy check. This is another way to compute a checksum, usually eight, 16, or 32 bits in length, that is appended to the message.

4.2 CRC

The CRC is based on polynomial arithmetic, in particular, on computing the remainder of dividing one polynomial by another. A polynomial in a single variable x whose coefficients are 0 or 1 is considered.

4.2.1 ADDITION AND SUBTRACTION OF POLYNOMIALS

Addition and subtraction are done **modulo 2**—that is, they are both the same as the *exclusive or* operator. For example, the sum of the polynomials $x3 + x + 1$ and $x4 + x3 + x2 + x$ is as is their difference. These polynomials are not usually written with minus signs, but they could be, because a coefficient of -1 is equivalent to a coefficient of 1.

4.2.2 MULTIPLICATION AND DIVISION OF POLYNOMIALS

Multiplication of such polynomials is straightforward. The product of one coefficient by another is the same as their combination by the logical *and* operator, and the partial products are summed using *exclusive or*. Multiplication is not needed to compute the CRC checksum. Division of polynomials can be done in much the same way a long division of polynomials over the integers. Below is an example.

$$
\begin{array}{r|lllllll}
 & x^4 & +x^3 & & & +1 & & \\
\hline
x^3+x+1 & x^7 & +x^6 & +x^5 & + & & x^2 & +x \\
 & x^7 & + & x^5 & +x^4 & & & \\
\cline{2-5}
 & & x^6 & + & x^4 & & & \\
 & & x^6 & + & x^4 & +x^3 & & \\
\cline{3-6}
 & & & & & x^3 & +x^2 & +x \\
 & & & & & x^3 & + & x+1 \\
\cline{6-8}
 & & & & & & x^2 & +\ 1
\end{array}
$$

The quotient of multiplied by the divisor of plus the remainder of equals the dividend. For instance, the message 11001001, where the order of transmission is from left to right (11001001) is treated as a representation of the polynomial The sender and receiver agree on a certain fixed polynomial called the *generator* polynomial.

4.2.3 COMPUTATION OF CHECKSUM

To compute an *r*-bit CRC checksum, the generator polynomial must be of degree *r*. The sender appends *r* 0-bits to the *m*-bit message and divides the resulting polynomial of degree by the generator polynomial. This produces a remainder polynomial of degree (or less). The remainder polynomial has *r* coefficients, which are the checksum. The quotient polynomial is discarded. The data transmitted (the code vector) is the original *m*-bit message followed by the *r* bit checksum.

There are two ways for the receiver to assess the correctness of the transmission. It can compute the checksum from the first *m* bits of the received data, and verify that it agrees with the last *r* received bits. Alternatively, the receiver can divide all the received bits by the generator polynomial and check that the *r*-bit remainder is 0. To see that the remainder must be 0, let *M* be the polynomial representation of the message, and let *R* be the polynomial representation of the remainder that was computed by the sender. Then the transmitted data corresponds to the polynomial (or, equivalently,

Let *G* is the generator polynomial and *Q* is the quotient. Therefore the transmitted data, is equal to *QG*, which is clearly a multiple of *G*. If the receiver is built as nearly as possible just like the sender, the receiver will append *r* 0-bits to the received data as it computes the remainder *R*. But the received data with 0-bits appended is still a multiple of G, so the computed remainder is still 0.

Two simple observations: For an r-bit checksum, G should be of degree r, because otherwise the first bit of the checksum would always be 0,which wastes a bit of the checksum. Similarly, the last coefficient should be 1 (that is, G should not be divisible by x), because otherwise the last bit of the checksum would always be 0 (because if G is divisible by x, then R must be also).

4.2.4 GENERATOR POLYNOMIALS

The following are the facts about generator polynomials

• If G contains two or more terms, all single-bit errors are detected.

• If G is not divisible by x (that is, if the last term is 1), and e is the least positive integer such that G evenly divides then all double errors that are within a frame of e bits are detected.

• An r-bit CRC checksum detects all burst errors of length (A burst error of length r is a string of r bits in which the first and last are in error, and the intermediate bits may or may not be in error). The generator polynomial creates a checksum of length 1, which applies even parity to the message.

4.3 ILLUSTRATION

Table 4.1 shows the generator polynomials used by some common CRC standards.

The "Hex" column shows the hexadecimal representation of the generator polynomial; the most significant bit is omitted, as it is always 1.

Common Name	r	Generator	
		Polynomial	Hex
CRC-12	12	$x^{12}+x^{11}+x^3+x^2+x+1$	80F
CRC-16	16	$x^{16}+x^{15}+x^2+1$	8005
CRC-CCITT	16	$x^{16}+x^{12}+x^5+1$	1021
CRC-32	32	$x^{32}+x^{26}+x^{23}+x^{22}+x^{16}+x^{12}+x^{11}+x^{10}+x^8+x^7+x^5+x^4+x^2+x+1$	04C11DB7

TABLE 4.1 GENERATOR POLYNOMIALS FOR SOME CRC CODES

The CRC standards differ in ways other than the choice of generating polynomial. Most initialize by assuming that the message has been preceded by certain nonzero bits, others do no such initialization. Most transmit the bits within a byte least significant bit first, some most significant bit first. Most append the checksum least significant byte first, others most significant byte first. Some complement the checksum.

4.3.1 ERROR DETECTION

To detect the error of erroneous insertion or deletion of leading 0's, some protocols prepend one or more nonzero bits to the message. These don't actually get transmitted, they are simply used to initialize the key register (described below) used in the CRC calculation. A value of r 1-bits seems to be universally used. The receiver initializes its register in the same way. The problem of trailing 0's is a little more difficult. There would be no problem if the receiver operated by comparing the remainder based on just the message bits to the checksum received. But, it seems to be simpler for the receiver to calculate the remainder for all bits received (message and checksum) plus r appended 0-bits. The remainder should be 0. But, with a 0 remainder, if the message has trailing 0-bits inserted or deleted, the remainder will still be 0, so this error goes undetected.

The usual solution to this problem is for the sender to complement the checksum before appending it. Because this makes the remainder calculated by the receiver nonzero (usually), the remainder will change if trailing 0's are inserted or deleted. Using the "mod" notation for remainder, we know that

$$(Mx^r + R) \bmod G = 0.$$

Denoting the "complement" of the polynomial R by we have

$$\begin{aligned}(Mx^r + \overline{R}) \bmod G &= (Mx^r + (x^{r-1} + x^{r-2} + \ldots + 1 - R)) \bmod G \\ &= ((Mx^r + R) + x^{r-1} + x^{r-2} + \ldots + 1) \bmod G \\ &= (x^{r-1} + x^{r-2} + \ldots + 1) \bmod G.\end{aligned}$$

The checksum calculated by the receiver for an error-free transmission should be

$$(x^{r-1} + x^{r-2} + \ldots + 1) \bmod G.$$

This is a constant (for a given G).

4.4 DIVISION ALGORITHM

The division process might be described informally as follows:

1. Initialize the CRC register to all 0-bits.
2. Get first/next message bit m.
3. If the high-order bit of CRC is 1, shift CRC and m together left 1 position, and XOR the result with the low- order r bits of G.
4. Otherwise, Just shift CRC and m left 1 position.
5. If there are more message bits, go back to get the next one.

The subtraction should be done first, and then the shift. It would be done that way if the CRC register held the entire generator polynomial, which in bit form is bits. Instead, the CRC register holds only the low-order r bits of G, so the shift is done first, to align things properly.

Below is shown the contents of the CRC register for the generator

$$G = x^3 + x + 1$$

and the message $M = x^7 + x^6 + x^5 + x^2 + x$. .Expressed in binary, G =1011 and M= 11100110.

000 Initial CRC contents. High-order bit is 0, so just shift in first message bit.

001 High-order bit is 0, so just shift in second message bit, giving:

011 High-order bit is 0 again, so just shift in third message bit, giving:

111 High-order bit is 1, so shift and then XOR with 011, giving:

101 High-order bit is 1, so shift and then XOR with 011, giving:

001 High-order bit is 0, so just shift in fifth message bit, giving:

011 High-order bit is 0, so just shift in sixth message bit, giving:

111 High-order bit is 1, so shift and then XOR with 011, giving:

101 There are no more message bits, so this is the remainder.

These steps can be implemented with the (simplified) circuit shown in Figure 4.2, which is known as a *feedback shift register*.

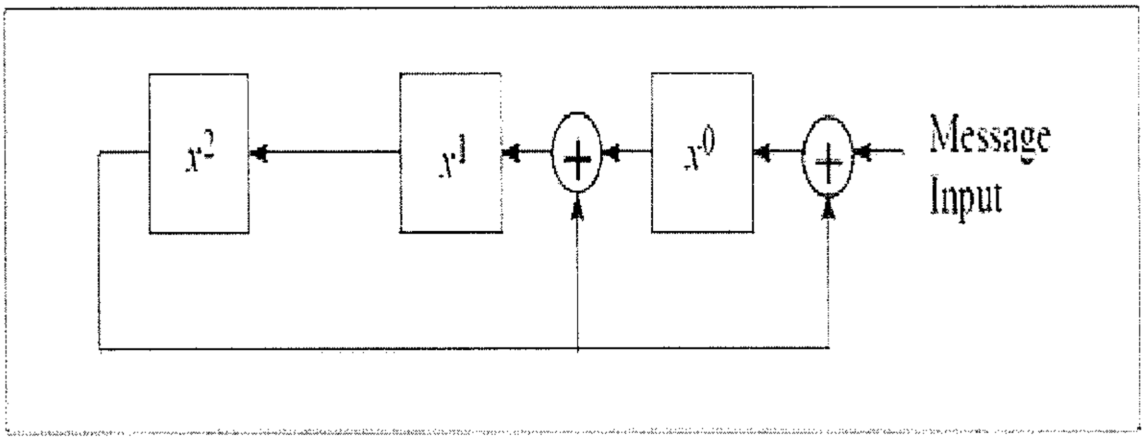


FIGURE 4-2. Polynomial division circuit for $G = x^3 + x + 1$.

The three boxes in the figure represent the three bits of the CRC register. When a message bit comes in, if the high-order bit (*x*2 box) is 0, simultaneously the message bit is shifted into the *x*0 box, the bit in *x*0 is shifted to *x*1, the bit in *x*1 is shifted to *x*2, and the bit in *x*2 is discarded. If the high-order bit of the CRC register is 1,then a 1 is present at the lower input of each of the two *exclusive or* gates. When a message bit comes in, the same shifting takes place but the three bits that wind up in the CRC register have been *exclusive or*'ed with binary 011.

When all the message bits have been processed, the CRC holds M mod G. If the circuit of Figure 4.2 were used for the CRC calculation, then after processing the message, r (in this case 3) 0-bits would have to be fed in. Then the CRC register would have the desired checksum.

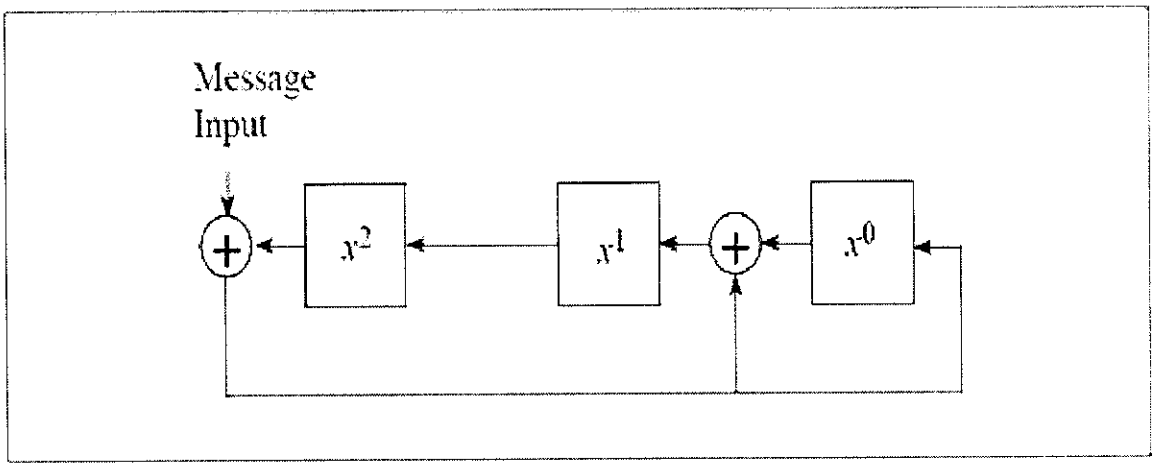


FIGURE 4-3. CRC circuit for $G = x^3 + x - 1$.

Instead of feeding the message in at the right end, feed it in at the left end, r steps away, as shown in Figure 4.3. This has the effect of premultiplying the input message M by xr. But premultiplying and postmultiplying are the same for polynomials. Therefore, as each message bit comes in, the CRC register contents are the remainder for the portion of the message processed.

5.DATA CONVERSION

CHAPTER 5

5.DATA CONVERSION

HDLC is a bit oriented protocol. Data conversion is done in order to support the operations involved. Types of data conversions used in HDLC are

Parallel to serial conversion (encoding module)

Serial to parallel conversion (decoding module)

5.1 PARALLEL TO SERIAL SHIFT REGISTER

In parallel-in serial-out register (or parallel-to-serial shift register, or shift-out register) each flip-flop must be able to accept data from either a serial or a parallel source. A small two-input multiplexer is required in front of each input. An extra input line selects between serial and parallel input signals, and as usual the flip-flops are loaded in accordance with a common clock signal.

The parallel to serial conversion circuit introduces a new input button i.e a mode control. The button labelled "S" indicates that the shift-out register is currently in serial mode. Thus, input signals present at the serial input just above the "S" button will be shifted into the register one by one with each clock pulse. If you click on the "S" button, it will change state as expected, but will also change to a "P" to indicate that the register now operates in parallel mode. This enables you to load the entire register at once from the parallel inputs just below the multiplexers.

The circuit shown in figure 5.1 has both parallel and serial inputs and outputs, it can serve as either a shift-in register or a shift-out register. This capability can have advantages in many cases. The least significant bit (LSB) is always available first at the serial output

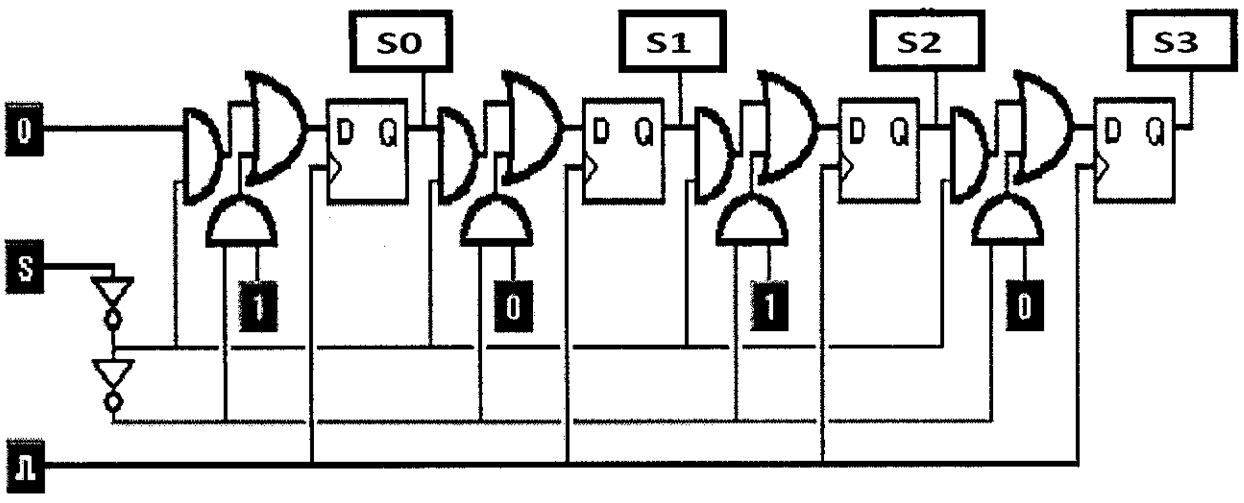
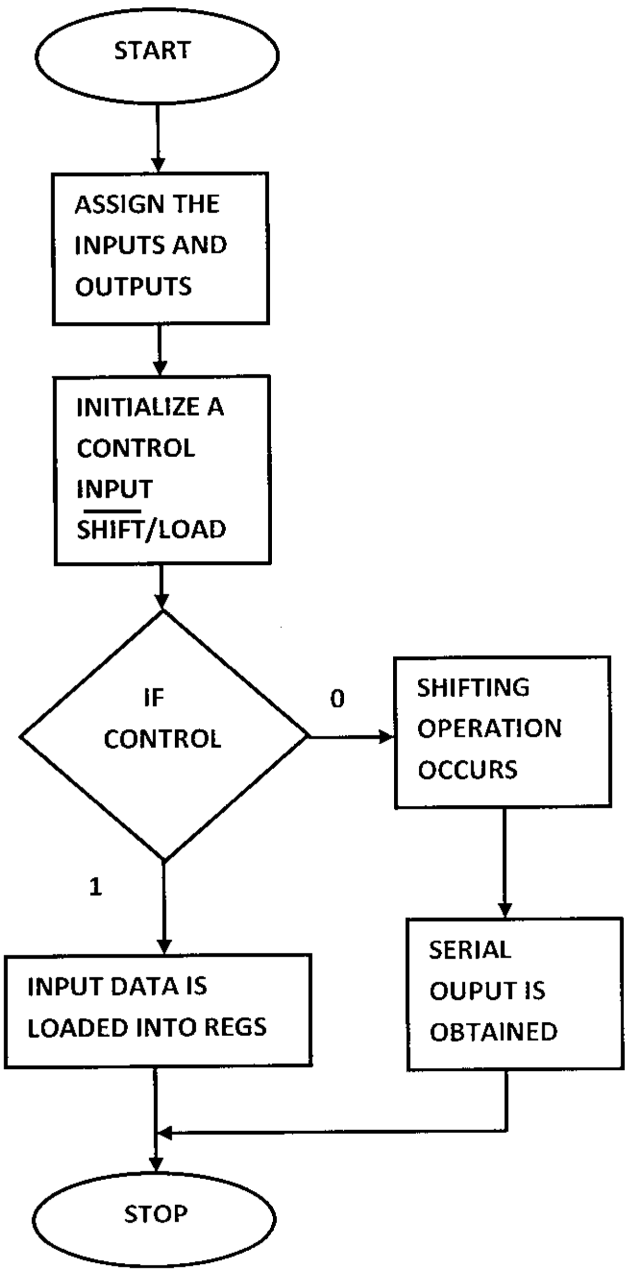


FIGURE 5.1 PARALLEL TO SERIAL CONVERSION

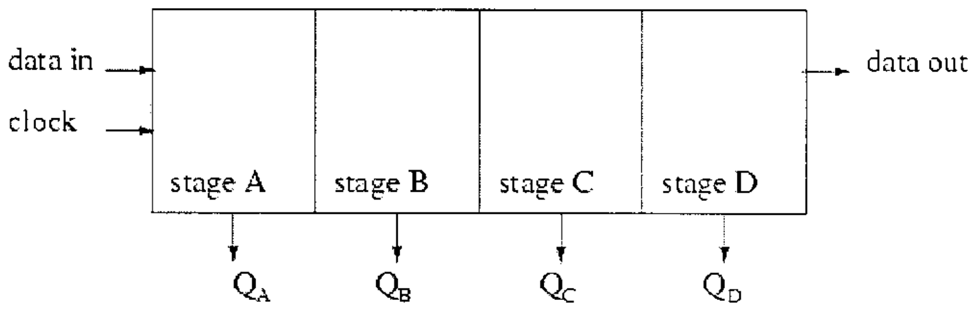
The inclusion of a serial input makes it possible to cascade multiple circuits of this type in order to increase the number of bits in the total register. This is common practice in real-world circuits.

FIGURE 5.2 FLOWCHART FOR PARALLEL TO SERIAL CONVERSION


START
ASSIGN THE INPUTS AND OUTPUTS
INITIALIZE A CONTROL INPUT $\overline{\text{SHIFT}}$/LOAD
IF CONTROL
0
SHIFTING OPERATION OCCURS
1
INPUT DATA IS LOADED INTO REGS
SERIAL OUPUT IS OBTAINED
STOP

5.2 SERIAL-IN PARALLEL OUT SHIFT REGISTER

A serial-in/parallel-out shift register shifts data into internal storage elements and shifts data out at the serial-out, data-out, pin. It is different in that it makes all the internal stages available as outputs. Therefore, a serial-in/parallel out shift register converts data from serial format to parallel format. If four data bits are shifted in by four clock pulses via a single wire at data-in, below, the data becomes available simultaneously on the four Outputs Q_A to Q_D after the fourth clock pulse.



5.3 Serial-in, parallel-out shift register with 4-stages

The practical application of the serial-in/parallel-out shift register is to convert data from serial format on a single wire to parallel format on multiple wires. Perhaps, we will illuminate four LEDs (Light Emitting Diodes) with the four outputs (**Q_A Q_B Q_C Q_D**).

6.VHDL

CHAPTER 6

6.VHDL

6.1 INTRODUCTION TO VHDL

VHDL is an acronym for VHSIC Hardware Description Language (VHSIC is an acronym for Very High Speed Integrated circuits).It is a Hardware description language that can be used to model a digital system at many levels of abstraction ranging from the algorithmic level to the gate level. The complexity of the digital system being modeled could vary from that of a simple gate to a complete digital electronic system, or anything in between. The digital system can also be described hierarchically. Timing can also be explicitly modeled in the same description.

The VHDL language can be regarded as an integrated amalgamation of the following languages:

Sequential language

Concurrent language

Net-list language

Timing specifications

Waveform generation language=>VHDL

Therefore, the language has constructs that enable you to express the concurrent or sequential behavior of a digital system with or without timing. It also allows you to model the system as an interconnection of components. Test waveforms can also be generated using the same constructs. All the above constructs may be combined to provide a comprehensive description of the system in a single model.

The language not only defines the syntax but also defines very clear simulation semantics for each language construct. Therefore, models written in this language can be verified using a VHDL simulator. It is strongly typed language and is often verbose to write. It inherits man of its features, especially the sequential language part, from the Ada programming language. Because VHDL provides an extensive range of modeling capabilities, it is often difficult to understand. Fortunately, it is possible to quickly assimilate a core subset of the language that is both easy and simple to understand without learning the more complex features. This subset is usually sufficient to model most applications. Th complete language, however, has sufficient power to capture the descriptions of the most complex chips to a complete electronic system.

6.2 FEATURES OF VHDL

The following are the major capabilities that the language provides along with the features that differentiate it from other hardware description languages.

- The language supports hierarchy, that is, a digital system can be modeled as a set of interconnected components; each component, in turn, can be modeled as a set of interconnected subcomponents.
- The language supports flexible design methodologies: top-down, bottom-up, or mixed.

- The language is not technology-specific, but is capable of supporting technology-specific features. It can also support various hardware technologies, for example, you may define new logic types and new components, and you may also specify technology-specific attributes.
- By being technology independent, the same behavior model can be synthesized into different vendor libraries.
- It supports both synchronous and asynchronous timing models.
- Various digital modeling techniques such as finite-state machine descriptions, algorithmic descriptions, and Boolean equations can be modeled using the language.
- The language is publicly available, human readable, machine readable, and above all, it is not proprietary.
- The capability of defining new data types provides the power to describe and simulate a new design technique at a very high level of abstraction without any concern about the implementation details.

6.3 DIFFERENT LEVEL OF ABSTRACTION

A digital system can be represented at different levels of abstraction [1]. This keeps the description and design of complex systems manageable.In VHDL, the different levels of abstraction are

Behavioral level modeling

Structural level modeling

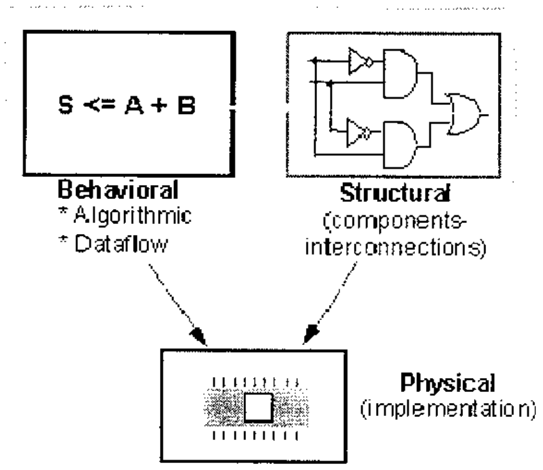


Figure 6.1: Levels of abstraction: Behavioral, Structural

6.3.1 BEHAVIORAL LEVEL MODELING

The highest level of abstraction is the behavioral level that describes a system in terms of what it does (or how it behaves) rather than in terms of its components and interconnection between them. A behavioral description specifies the relationship between the input and output signals. This could be a Boolean expression or a more abstract description such as the Register Transfer

Sor Algorithmic level. As an example, let us consider a simple circuit that warns car passengers when the door is open or the seatbelt is not used whenever the car key is inserted in the ignition lock At the behavioral level this could be expressed as,

Warning = Ignition_on AND (Door_open OR Seatbelt_off)

6.3.2 TYPES IN BEHAVIORAL LEVEL MODELING

The behavioral level can be further divided into two kinds of styles:

1.**Data flow**

2.**Algorithmic**.

1.DATAFLOW

The dataflow representation describes how data moves through the system. This is typically done in terms of data flow between registers (Register Transfer level). The data flow model makes use of concurrent statements that are executed in parallel as soon as data arrives at the input.

2.ALGORITHMIC

In algorithmic, sequential statements are executed in the sequence that they are specified. VHDL allows both concurrent and sequential signal assignments that will determine the manner in which they are executed.

6.3.3 STRUCTURAL LEVEL MODELING

The structural level describes a system as a collection of gates and components that are interconnected to perform a desired function. A structural description could be compared to a schematic of interconnected logic gates.

It is a representation that is usually closer to the physical realization of a system. For the example above, the structural representation is shown in Figure 2 below.

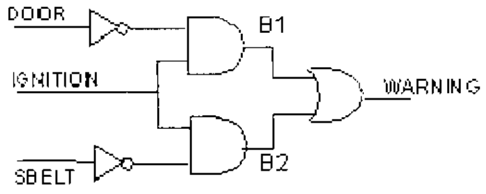


Figure 6.2: Structural representation of a “buzzer” circuit.

VHDL allows one to describe a digital system at the structural or the behavioral level.

7.CODING

CHAPTER 7

7. CODING

7.1 HDLC MAIN CODING

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity hdlc is
port(  clk   :          in std_logic;
      rst   : in std_logic;
      frame  :  out std_logic_vector(63 downto 0)
         );
end hdlc;
architecture str of hdlc is
component encoding is
  port(
       clk    : in std_logic;
       rst    : in std_logic;
       txd    : out std_logic
    );
    end component;
component decoding is
port(
       clk    : in std_logic;
       rst    : in std_logic;
       rxd    : in  std_logic;
      frame   : out std_logic_vector(63 downto 0)
    );
  end component;
signal w1  : std_logic;


begin
u0:encoding port map(clk,rst,w1);
u1:decoding port map(clk,rst,w1,frame);
end;
```

7.2 CONVERT PARALLEL INPUT TO SERIAL DATA

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity par_ser is
     port
  (
    clock    : in  Std_Logic;
    reset    : in  Std_Logic;
    data_in  : in  Std_Logic_Vector(7 downto 0);
    txd      : out Std_Logic
    );
end par_ser;
architecture only of par_ser is
 signal transmit_shift_register   : Std_Logic_Vector(7 downto 0);
 signal address                   : Std_Logic_Vector(3 downto 0);
 begin
 process(clock,reset,data_in,transmit_shift_register)
  constant counter    : Std_Logic_Vector(3 downto 0):="0001";
  begin
  if reset = '0' the
            transmit_shift_register <= "00000000";
            txd <= '1'
       address <= "0000";
      elsif (clock'event and clock='1') then
      transmit_shift_register  <=data_in;
       txd <= '1'
      if  transmit_shift_register/="UUUUUUUU"   THEN
      case address is
      when "0000" =>
      txd <= '0';
       address <= address + counter;
      when "0001" | "0010" | "0011" | "0100" |
      "0101" | "0110" | "0111" | "1000"  =>
       txd <= transmit_shift_register(0);
         transmit_shift_register <= '1' & transmit_shift_register(7 downto 1);
       address <= address + counter
        when "1001" =>
       txd <= '1';
       transmit_shift_register <= '1' & transmit_shift_register(7 downto 1);
       address <= "0000";
        when others =>
               null;
            end case;
     end if;
    end if;
end process;
end only;
```

7.3 CRC CODING

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity poly1 is
   Port ( input:in std_logic_vector(31 downto 0);
          clk : in std_logic;
          output :out std_logic_vector(15 downto 0) );
end poly1;

architecture Behavioral of  poly1 is

constant poly : std_logic_vector(16 downto 0):="10001000000100001";

function  gfmul(op1  :  in  std_logic_vector(31  downto  0);op2  :  in
std_logic_vector(16 downto 0))return std_logic_vector is
  variable  temp  :  std_logic_vector(31  downto
0):="00000000000000000000000000000000";
  variable output : std_logic_vector(15 downto 0):="0000000000000000";
  --variable poly : std_logic_vector(8 downto 0):="100011011";
  begin
  --for i in 0 to 7 loop
  --  if(op1(7-i)='1')then
  --     temp((14-i)downto(7-i)):=temp((14-i)downto(7-i)) xor op2;
  --  end if;
```

```
 --end loop;
 temp:=op1;
 for i in 0 to 15 loop
  if(temp(31-i)='1')then
     temp((31-i)downto (15-i)):=temp((31-i)downto (15-i)) xor poly;
  end if;

 end loop;


 output:=temp(15 downto 0);
 return output;
 end  gfmul;
begin
process(clk,input)



begin


   output<=gfmul(input,poly);
end process;
end Behavioral;
```

7.4 ENCODING

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
```

```
entity encoding is

port(
     clk   : in std_logic;
     rst   : in std_logic;
     txd   : out std_logic
  );


end encoding;

architecture beh of encoding is

constant poly : std_logic_vector(16 downto 0):="10001000000100001";

function gfmul(op1 : in std_logic_vector(31 downto 0);op2 : in
std_logic_vector(16 downto 0))return std_logic_vector is

  variable temp : std_logic_vector(31 downto
0):="00000000000000000000000000000000";
  variable output : std_logic_vector(15 downto 0):="0000000000000000";

begin

  temp:=op1;
  for i in 0 to 15 loop
   if(temp(31-i)='1')then
      temp((31-i)downto (15-i)):=temp((31-i)downto (15-i)) xor poly;
     end if;
    end loop;
    output:=temp(15 downto 0);
   return output;
 end  gfmul;
  signal frame_syn : std_logic_vector(7 downto 0):="01111110";
 signal add_field : std_logic_vector(7 downto 0):="10111111";
 signal con_field : std_logic_vector(7 downto 0):="00111111";
 signal data      : std_logic_vector(15 downto 0):="1011110101110110";
 signal crc       : std_logic_vector(15 downto 0);
 signal address   : Std_Logic_Vector(4 downto 0):="00000";
 signal count     : Std_Logic_Vector(4 downto 0):="00001";
 signal sh_reg    : std_logic_vector(63 downto 0);
 signal counter   : Std_Logic_Vector(3 downto 0):="0000";
 signal bit_count : Std_Logic_Vector(5 downto 0):="000000";
 begin
 process(clk,rst,address)
      begin
    if rst='0' then
    crc<=gfmul((data & "0000000000000000"),poly);
 sh_reg<="00000000000000000000000000000000000000000000000000000000
 00000000";
 elsif (clk' event and clk='1') then
  case address is
  when "00000" =>
  sh_reg<=frame_syn & crc & data & con_field & add_field & frame_syn;
  address <= address + count;
  when "00001" | "00010" | "00011" | "00100" |
 "00101" | "00110" | "00111" | "01000" =>
  txd <= sh_reg(0);
 sh_reg <= '1' & sh_reg(63 downto 1);
address <= address + count;
when "01010" | "01011" | "01100" | "01101" |
"01110" | "01111" | "10000" | "10001"=>
        txd <= sh_reg(0);
        sh_reg <= '1' & sh_reg(63 downto 1);
address <= address + count;
 when "10010" =>
 counter<="1111";
 when others =>
 if bit_count="110000" then
 address<="01010";
  end if;
 if counter<"0101" then
     if sh_reg(0)='1' then
         txd <= sh_reg(0);
         sh_reg <= '1' & sh_reg(63 downto 1);
         counter<=counter+1;
         bit_count<=bit_count+1;
       else
        txd <= sh_reg(0);
         sh_reg <= '1' & sh_reg(63 downto 1);
         counter<="0000";
         bit_count<=bit_count+1;
   end if;
else
```

```
    txd <= '0';
    counter<="0000";
end if;
end case;
end if;
end process;
end beh;
```

7.5 DECODING

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity decoding is
 port(
       clk    : in std_logic;
       rst   : in std_logic;
       rxd    : in  std_logic;
      frame   : out std_logic_vector(63 downto 0)
  );
end decoding;
architecture beh of decoding is
constant poly : std_logic_vector(16 downto 0):="10001000000100001";
function gfmul(op1 : in std_logic_vector(31 downto 0);op2 : in std_logic_vector(16 downto 0))return std_logic_vector is
  variable temp : std_logic_vector(31 downto 0):="00000000000000000000000000000000";
  variable output : std_logic_vector(15 downto 0):="0000000000000000";
begin
 temp:=op1;
 for i in 0 to 15 loop
 if(temp(31-i)='1')then
      temp((31-i)downto (15-i)):=temp((31-i)downto (15-i)) xor poly;
 end if;
end loop;
output:=temp(15 downto 0);


return output;
end  gfmul;



--signal crc        : std_logic_vector(15 downto 0);
signal address   : Std_Logic_Vector(4 downto 0):="00000";
signal count     : Std_Logic_Vector(4 downto 0):="00001";
signal sh_reg    : std_logic_vector(63 downto 0);
signal counter   : Std_Logic_Vector(3 downto 0):="0000";
signal bit_count : Std_Logic_Vector(5 downto 0):="000000";
begin
process(clk,rst,address)
begin
if rst='0' then
sh_reg<="00000000000000000000000000000000000000000000000000000
00000000";
elsif (clk' event and clk='1') then
case address is
```

```
      when "00000" =>
      address <= address + count;
       when "00001" | "00010" | "00011" | "00100" |
       "00101" | "00110" | "00111" | "01000" =>
                    sh_reg <= rxd & sh_reg(63 downto 1);
                    address <= address + count;
       when "01010" | "01011" | "01100" | "01101" |
       "01110" | "01111" | "10000" | "10001"=>
                    sh_reg <= rxd & sh_reg(63 downto 1);
                    address <= address + count;

when "10010" =>
      frame<=sh_reg;
       counter<="1111";
when others =>
           if bit_count="110000" then
           address<="01010";
           end if;
          if counter<"0101" then
     if rxd='1' then
          sh_reg <= rxd & sh_reg(63 downto 1);
          counter<=counter+1;
          bit_count<=bit_count+1;
      else
          sh_reg <= rxd & sh_reg(63 downto 1);
          counter<="0000";
          bit_count<=bit_count+1;
    end if;
```

```
      else
        sh_reg <= sh_reg;
        counter<="0000";
end if;
end case;
end if;
end process;
end beh;
```

7.6 CONVERT SERIAL DATA TO PARALLEL DATA

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
entity ser_par is
  port
  (
    clock   : in  std_logic;
    reset   : in  std_logic;
    RxD     : in  std_logic;
    Data_out : out std_logic_vector(7 downto 0));
end ser_par;
architecture only of ser_par is
  signal address                  : std_logic_vector(3 downto 0);
  signal receiver_shift_register   : std_logic_vector(7 downto 0);
begin
process(clock,reset,RxD)
constant counter      : std_logic_vector(3 downto 0):="0001";
```

```
 begin
if reset = '0' then
      address <= "0000";
      receiver_shift_register <= "00000000";
 elsif (clock'event and clock='1') then
case address is
when "0000" =>
          address <= address + counter;
when "0001" | "0010" | "0011" | "0100" |
         "0101" | "0110" | "0111" | "1000" =>
          address <= address + counter;
          receiver_shift_register <= RxD & receiver_shift_register(7 downto 1);

  when "1001" =>
                 Data_out <= receiver_shift_register;
                 address <= "0000"
   when others =>
                 null;
        end case;
end if;
end process;
end only;
```

8.SIMULATION RESULT

CHAPTER 8

SIMULATION RESULT

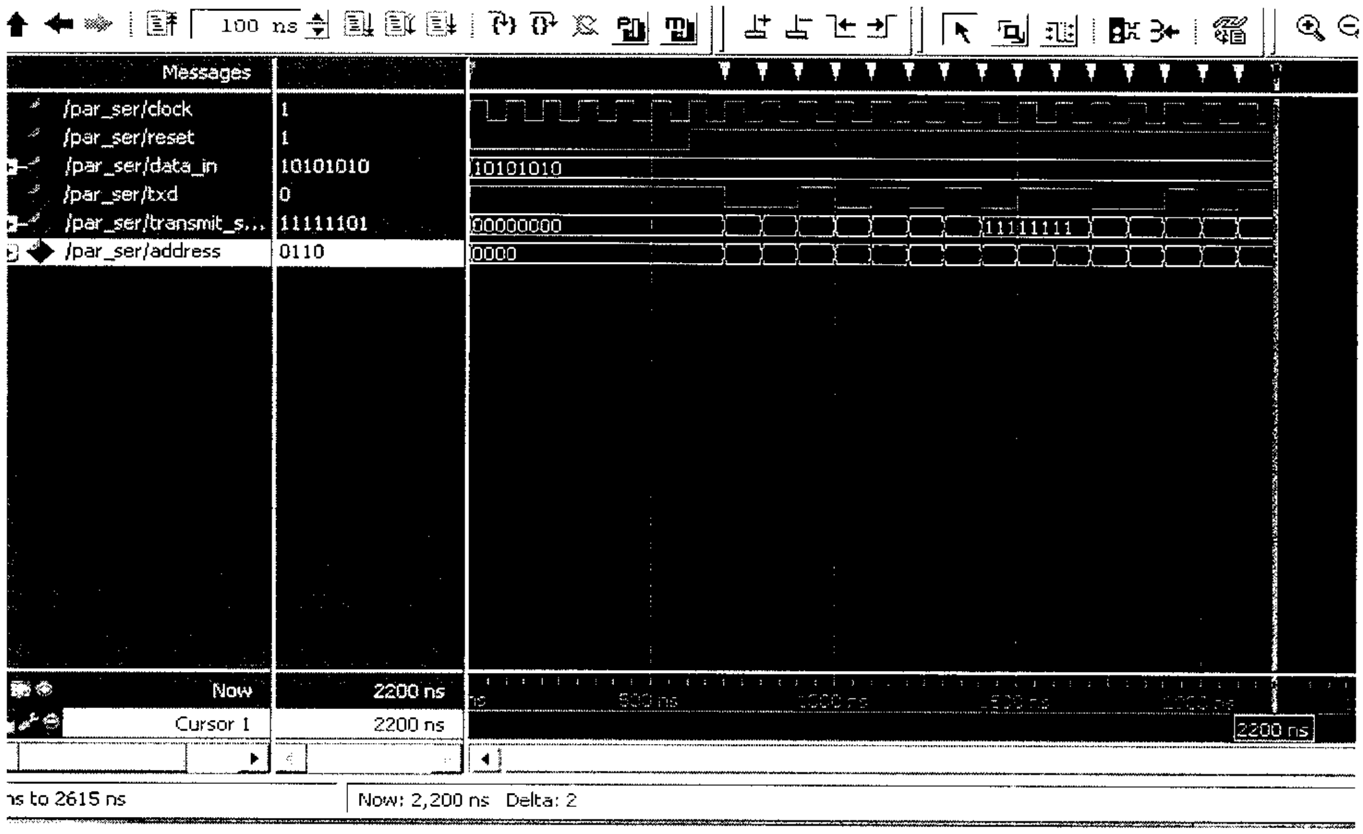
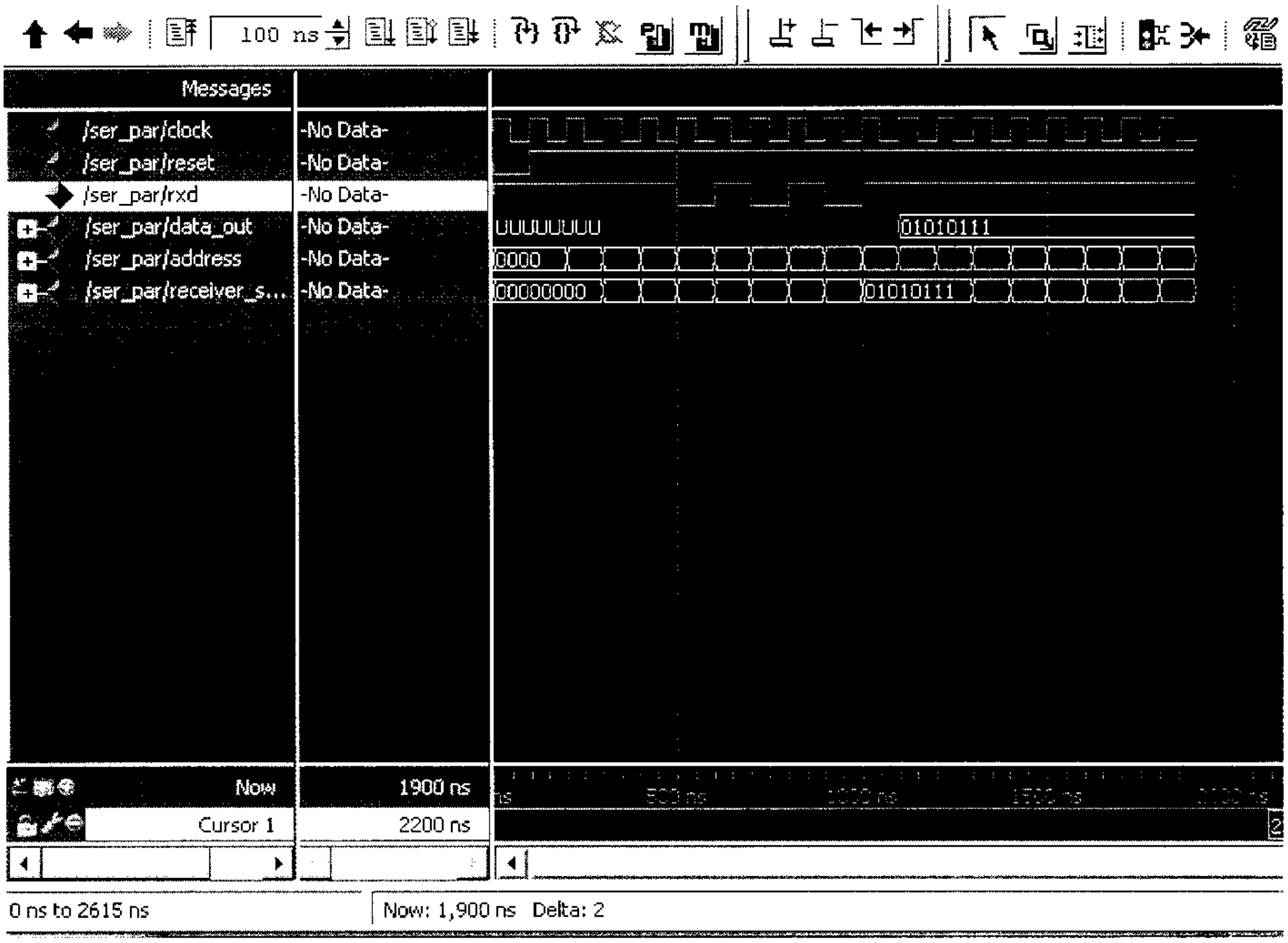


FIGURE 8.1 PARALLEL TO SERIAL CONVERSION



Messages	
/ser_par/clock	-No Data-
/ser_par/reset	-No Data-
/ser_par/rxd	-No Data-
/ser_par/data_out	-No Data-
/ser_par/address	-No Data-
/ser_par/receiver_s...	-No Data-
Now	1900 ns
Cursor 1	2200 ns

UUUUUUUU 01010111

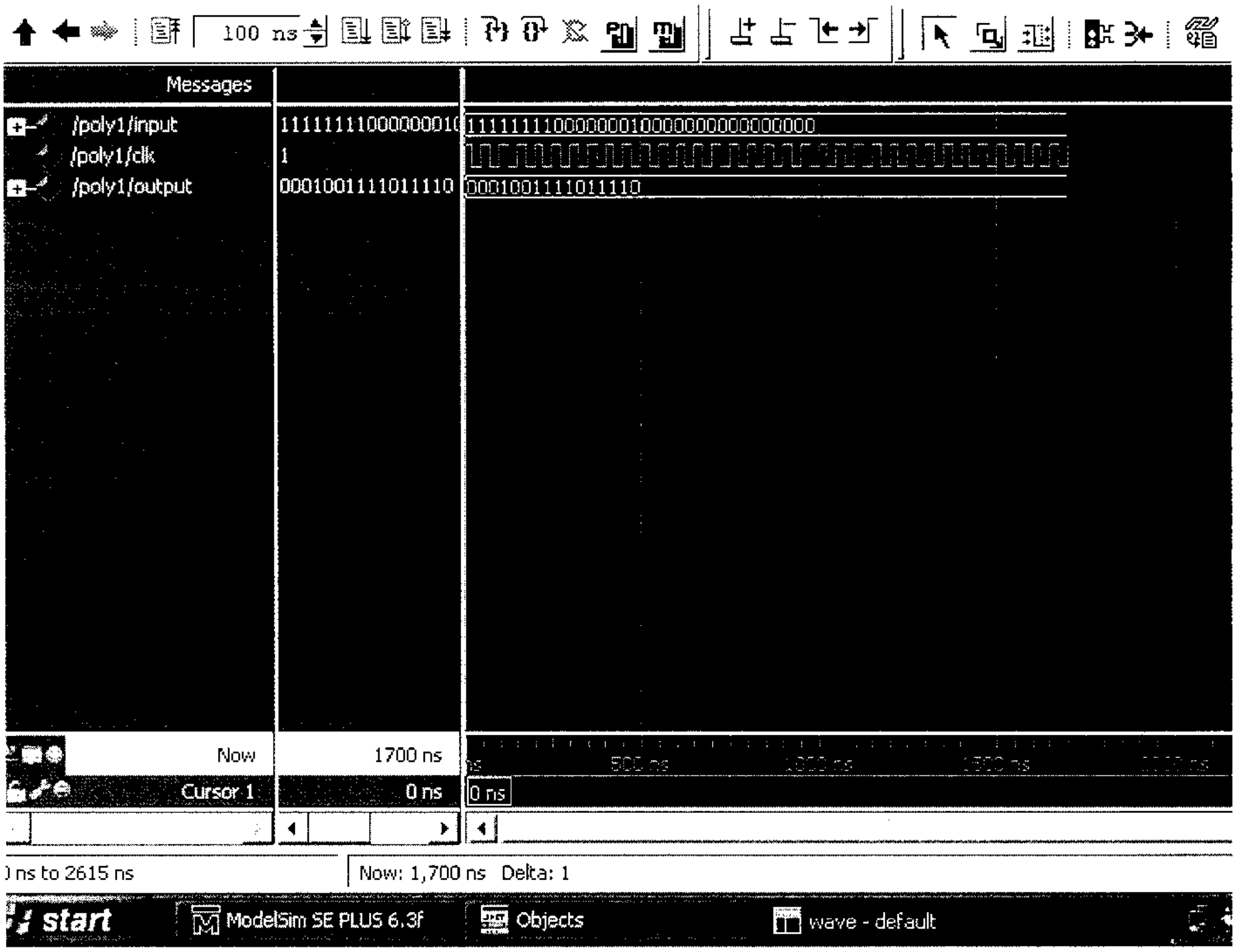
0000

00000000 01010111

0 ns to 2615 ns

Now: 1,900 ns Delta: 2

FIGURE 8.2 SERIAL TO PARALLEL CONVERSION



Messages

/poly1/input 11111111000000010 11111111000000001000000000000000

/poly1/clk 1

/poly1/output 0001001111011110 0001001111011110

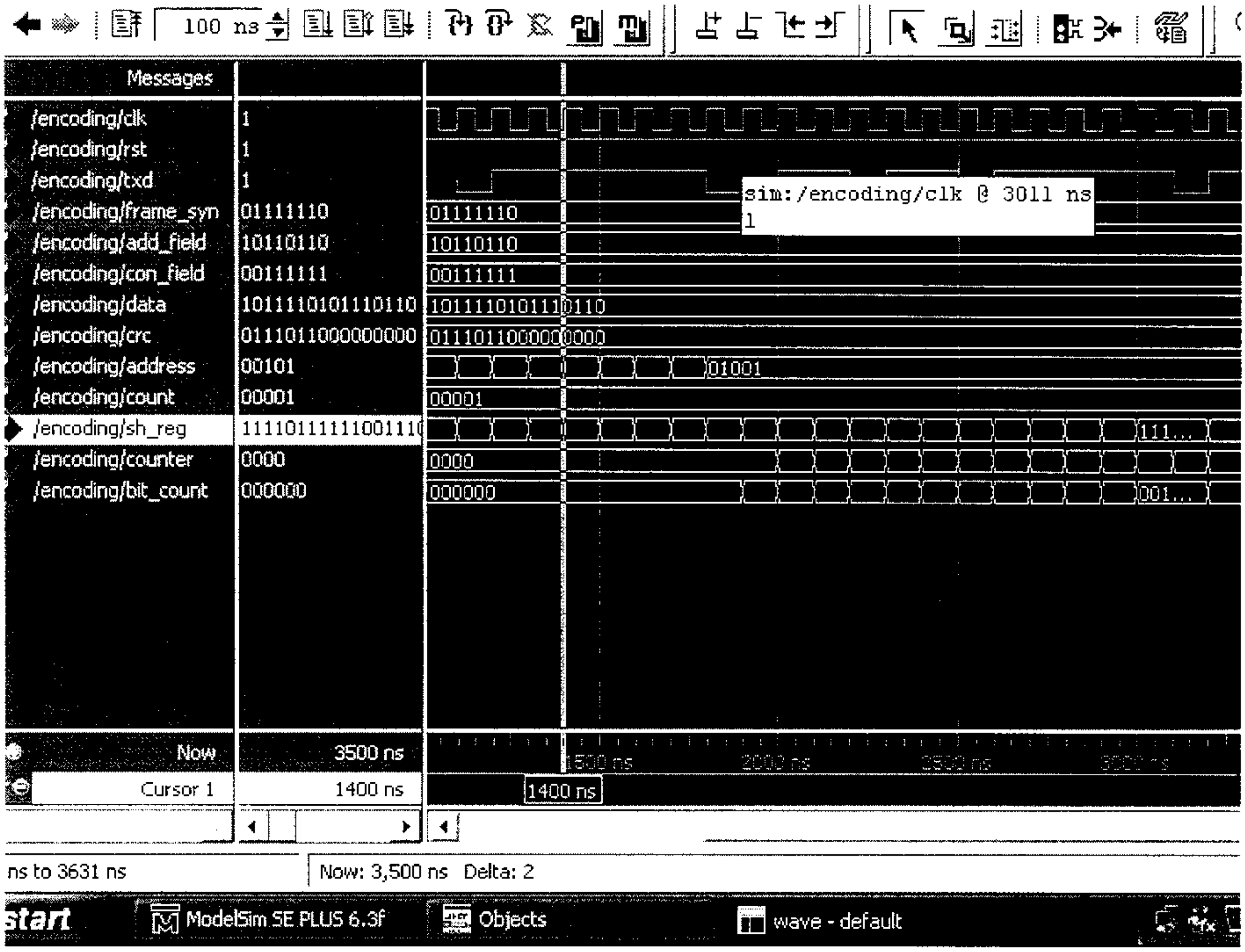
Now 1700 ns

Cursor 1 0 ns 0 ns

) ns to 2615 ns

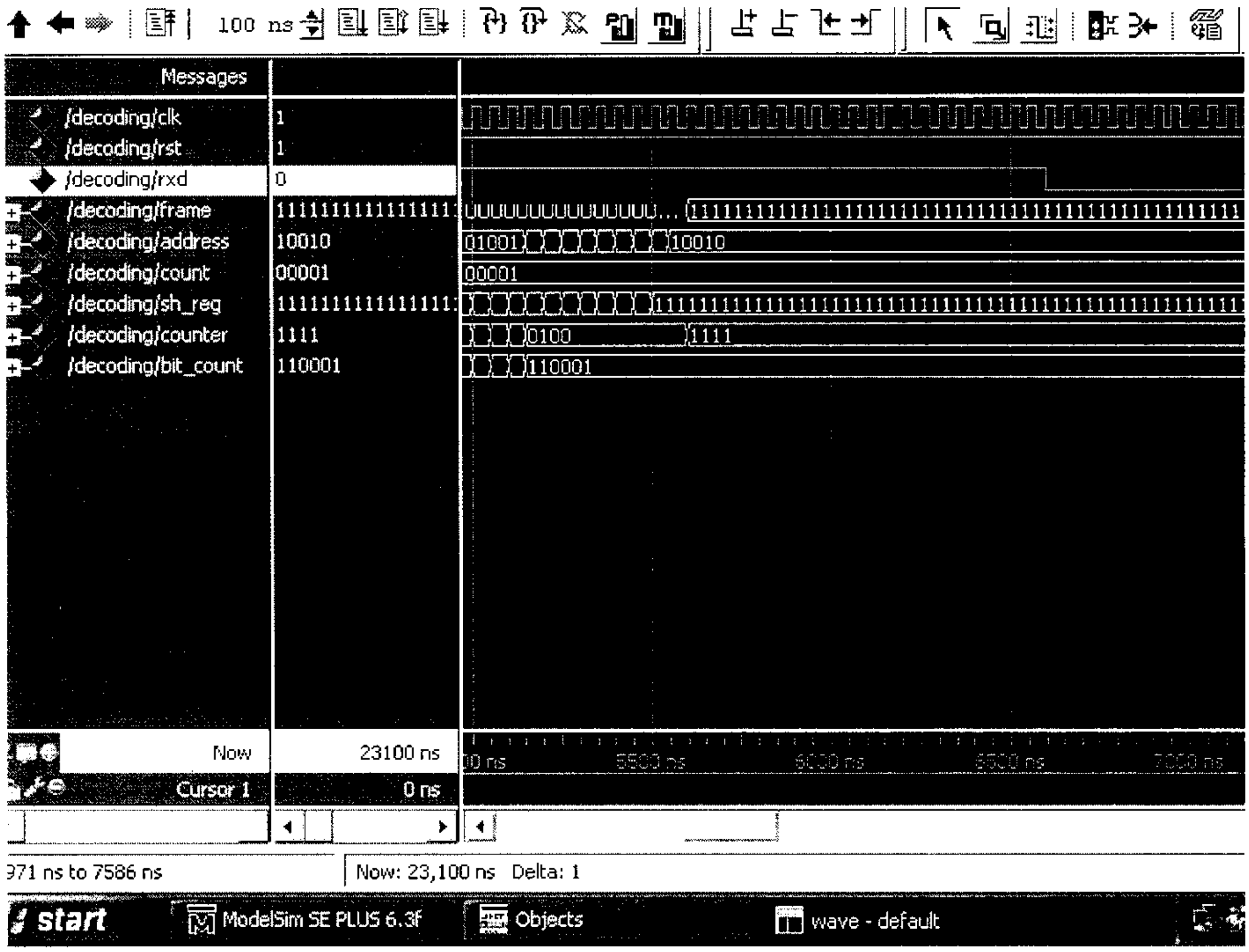
Now: 1,700 ns Delta: 1

FIGURE 8.3 CRC



```
/encoding/clk          1
/encoding/rst          1
/encoding/txd          1
/encoding/frame_syn    01111110
/encoding/add_field    10110110
/encoding/con_field    00111111
/encoding/data         1011110101110110
/encoding/crc          0111011000000000
/encoding/address      00101
/encoding/count        00001
/encoding/sh_reg       1111011111100111
/encoding/counter      0000
/encoding/bit_count    000000
```

FIGURE 8.4 ENCODING

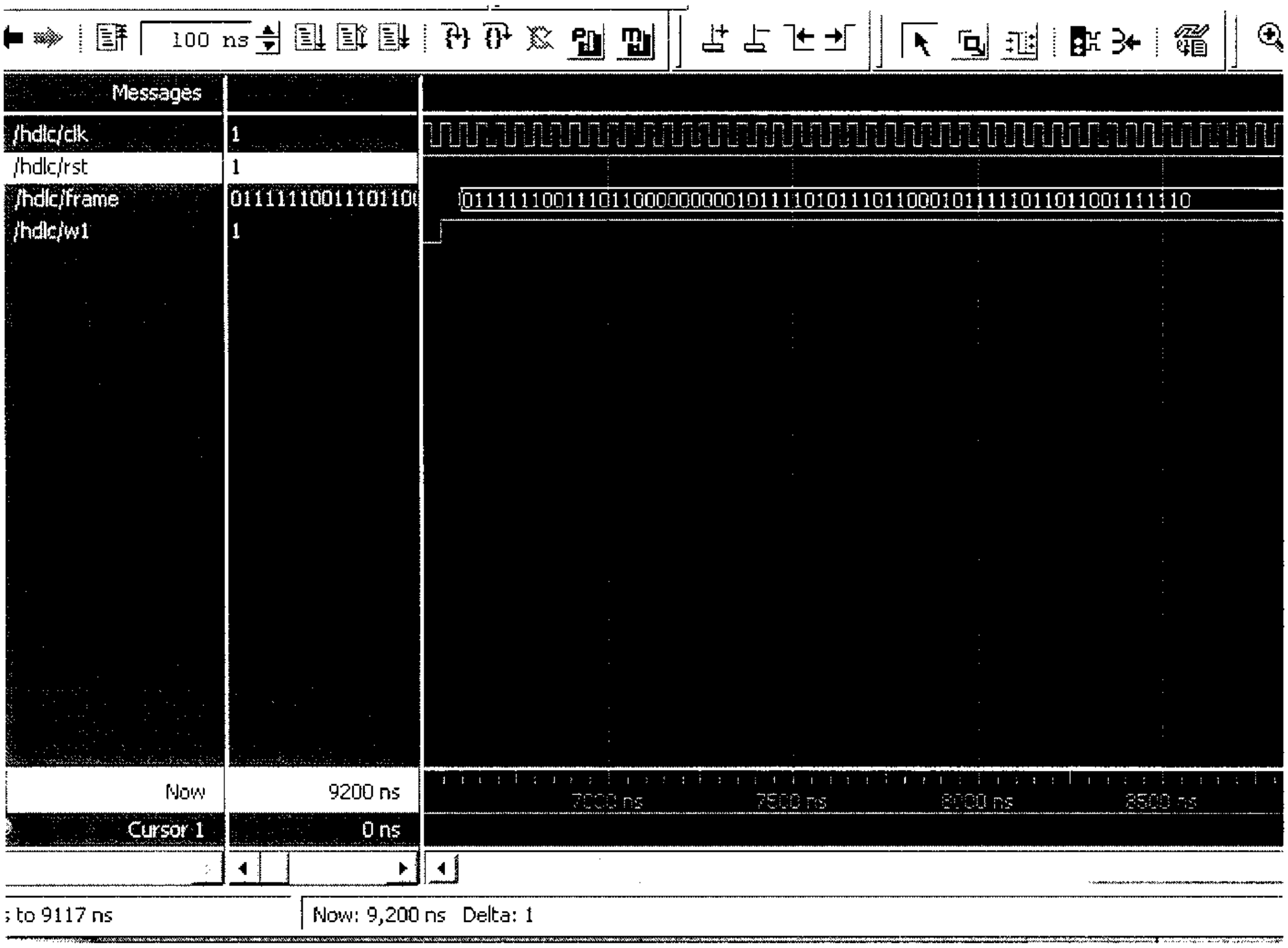


```
Messages
/decoding/clk        1
/decoding/rst        1
/decoding/rxd        0
/decoding/frame      11111111111111111   UUUUUUUUUUUUUUU... 1111111111111111111111111111111111111111111111
/decoding/address    10010               01001 ... 10010
/decoding/count      00001               00001
/decoding/sh_reg     11111111111111111   ... 1111111111111111111111111111111111111111111111111
/decoding/counter    1111                ... 0100 ... 1111
/decoding/bit_count  110001              ... 110001

Now       23100 ns
Cursor 1  0 ns

971 ns to 7586 ns     Now: 23,100 ns  Delta: 1
```

FIGURE 8.5 DECODING



```
/hdlc/clk      1
/hdlc/rst      1
/hdlc/frame    0111111001110110    01111110011101100000000001011110101110110001011111011011001111110
/hdlc/w1       1

Now            9200 ns        7000 ns    7500 ns    8000 ns    8500 ns
Cursor 1       0 ns

; to 9117 ns        Now: 9,200 ns   Delta: 1
```

FIGURE 8.6 HDLC

9.CONCLUSION

CHAPTER 9

CONCLUSION

Thus the Frame Check Sequence(FCS) is generated and HDLC protocol is implemented in TKbase kit using FPGA. It is verified by downloading the HDLC modules designed in VHDL.

In the design of small or medium-sized bulk of communication products, using FPGAs to replace ASIC devices for HDLC procedures implementation is a proper choice, which implies a better application prospect.

FUTURE ENHANCEMET

In the future work HDLC protocol can be designed to eliminate bit insertion/deletion and convert it into a byte-oriented protocol.Conventional serial bit stream protocols can take advantage of the aspect of the ISDN standard to simplify the hardware necessary to implement the protocols. The modified version provides the same function as zero bit insertion/deletion in HDLC in a more efficient manner. Specifically, a technique which does byte insertion instead of bit insertion to insure data transparency can be implemented.

10.BIBLIOGRAPHY

CHAPTER 10

BIBLIORAPHY

PAPERS

1.[Black] Black, Richard. Web site www.cl.cam.ac.uk/Research/SRG/bluebook/ 21/crc/crc.html. University of Cambridge Computer Laboratory Systems Research Group, February 1994.

2. [PeBr] Peterson, W. W. and Brown, D.T. "Cyclic Codes for Error Detection." In*Proceedings of the IRE*, January 1961, 228–235.

3. [Tanen] Tanenbaum, Andrew S. *Computer Networks*, Second Edition. Prentice Hall, 1988.

4. Jun Hang , Wenhao Zhang , "Design and implementation of HDLC Procedures " , Beihang University Press, pp 72-78 , Aug 2007

5. Henriksson T ,Dake Liu, "Implementation of cyclic calculation ", Design Automation conference , pp 563-564,jan 2003

BOOKS

1. Data Communications And Networking -Behrouz A Forouzan

2. Computer Networks - Uyless D Black

3.Designer's guide to VHDL –Ahendon

4.VHDL programming by examples- Perry