XAPP687 (v1.0) November 21, 2003

64B/66B Encoder/Decoder

Author: Nick McKay and Matt DiPaolo

Summary

This application note describes the encoding and decoding blocks of the 64B/66B encoding scheme. This application allows designs to use the RocketIO™ transceiver of the Virtex-II Pro™ device or an external SerDes with either Virtex-II™ or Virtex-II Pro devices.

Introduction

Serial standards use an embedded clock in the data stream. To ensure the Clock Data Recovery (CDR) can track the embedded clock; a transition rich data stream is required. In the past, 8B/10B encoding provided the transition rich data along with a DC balanced data stream for many standards, including 1X and 2X Fibre Channel and Gigabit Ethernet. However, as serial rates increase to 3.125 Gbits/s and greater, the overhead of 8B/10B encoding becomes unacceptable in standards such as 10G Fibre Channel and 10G Gigabit Ethernet.

To reduce the overhead, 64B/66B encoding was created. This scheme encodes 64 bits of data into 66 bits, providing a 3% overhead compared to the 20% overhead of 8B/10B encoding. Figure 1 shows how the 64B/66B encode/decode logic of this application fits in a complete system. This application note provides detailed information on the reference design indicated in grey. It does not cover the scrambler, descrambler, synchronization block, or gearbox. Their functions are briefly described after the figure.

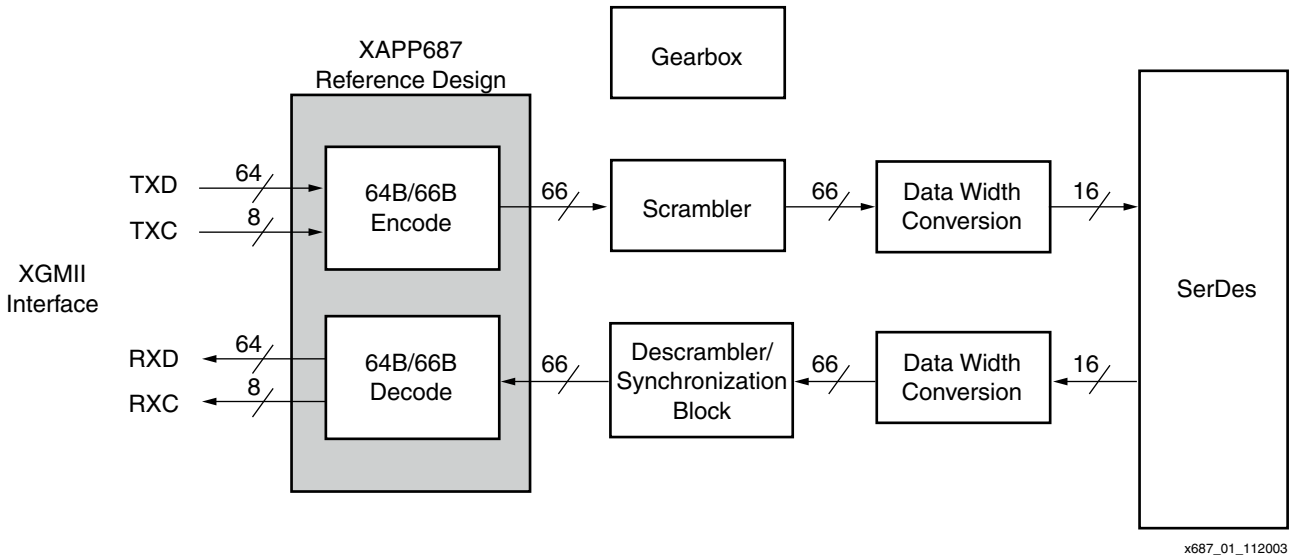


Figure 1: **64B/66B System Data Path**

The 64B/66B encoder takes XGMII signals consisting of a 64-bit data and a 8-bit control bus and converts them into 66-bit code words according to the scheme outlined in Section 49.2.4 of the IEEE 802.3ae specification. The decoder converts the 66-bit code words back to XGMII-compatible signals.

The scrambler block encrypts the transmitted data, and the descrambler block decrypts the received data. These blocks are not required in all applications and are not included in the reference design.

The synchronization block aligns the incoming serial data on a 66-bit boundary to also align on the 64-bit boundary on the XGMII side of the 64B/66B encode/decode block. This block is not included in the reference design.

The gearbox creates the two clock domains to allow the 66 data bits to be converted into the correct SerDes interface width. This design function requires a shift register and data width converter. This block is not included in the reference design due to the numerous bus interfaces available.

Reference Design

The reference design, which is available in VHDL or Verilog, consists of three modules: encoder, decoder, and encoder_decoder_top. The encoder_decoder_top module instantiates the encoder and decoder blocks along with other logic including clock generation.

The VHDL reference design contains four files: `encode_decode_top.vhd`, `encoder.vhd`, `decoder.vhd`, and `pcs_util.vhd`. The Verilog reference design contains three files: `encode_decode_top.v`, `encoder.v`, and `decoder.v`. The design files can be found on the following website: http://www.xilinx.com/bvdocs/appnotes/xapp687.zip.

Encoder

Figure 2 shows a block diagram of the encoder.

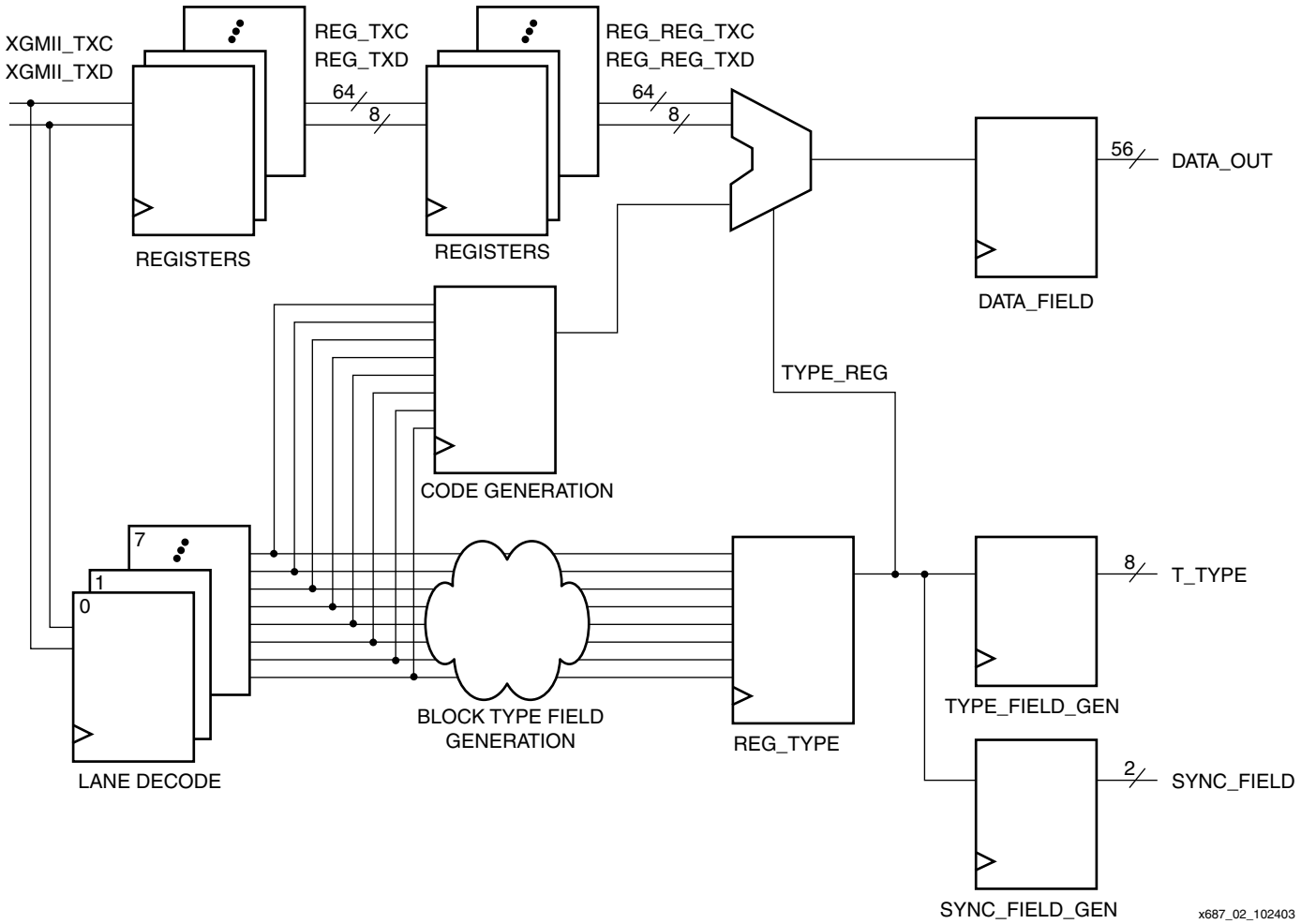


Figure 2: **Encoder Block Diagram**

Table 1 defines all encoder block signals.

Table 1: **Encoder Signals**

Port	I/O	Description
CLK	Input	Clocks all the encoder logic
DATA_OUT	Output [65:0]	Encoded data bus
ENABLE	Input	Enables the encoder (active High)
INIT	Input	Asynchronous reset (active High)
T_TYPE	Output	Indicates the type of sequence transmitted
XGMII_TXC	Input [7:0]	XGMII transmit control bus
XGMII_TXD	Input [63:0]	XGMII transmit data

Figure 3 shows the encoding scheme, and Table 2 provides the valid control codes, taken from the IEEE 802.3ae specification.

Input Data	Sync	Block Payload							
Bit Position / Data Block Format	01	2							65
D_0 D_1 D_2 D_3 D_4 D_5 D_6 D_7	01	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
Control Block Formats		Block Type Field							
C_0 C_1 C_2 C_3 C_4 C_5 C_6 C_7	10	0x1e	C_0	C_1	C_2	C_3	C_4	C_5	C_6	C_7
C_0 C_1 C_2 C_3 O_4 D_5 D_6 D_7	10	0x2d	C_0	C_1	C_2	C_3	O_4	D_5	D_6	D_7
C_0 C_1 C_2 C_3 S_4 D_5 D_6 D_7	10	0x33	C_0	C_1	C_2	C_3		D_5	D_6	D_7
O_0 D_1 D_2 D_3 S_4 D_5 D_6 D_7	10	0x66	D_1	D_2	D_3	O_0		D_5	D_6	D_7
O_0 D_1 D_2 D_3 O_4 D_5 D_6 D_7	10	0x55	D_1	D_2	D_3	O_0	O_4	D_5	D_6	D_7
S_0 D_1 D_2 D_3 D_4 D_5 D_6 D_7	10	0x78	D_1	D_2	D_3	D_4		D_5	D_6	D_7
O_0 D_1 D_2 D_3 C_4 C_5 C_6 C_7	10	0x4b	D_1	D_2	D_3	O_0	C_4	C_5	C_6	C_7
T_0 C_1 C_2 C_3 C_4 C_5 C_6 C_7	10	0x87		C_1	C_2	C_3	C_4	C_5	C_6	C_7
D_0 T_1 C_2 C_3 C_4 C_5 C_6 C_7	10	0x99	D_0		C_2	C_3	C_4	C_5	C_6	C_7
D_0 D_1 T_2 C_3 C_4 C_5 C_6 C_7	10	0xaa	D_0	D_1		C_3	C_4	C_5	C_6	C_7
D_0 D_1 D_2 T_3 C_4 C_5 C_6 C_7	10	0xb4	D_0	D_1	D_2		C_4	C_5	C_6	C_7
D_0 D_1 D_2 D_3 T_4 C_5 C_6 C_7	10	0xcc	D_0	D_1	D_2	D_3		C_5	C_6	C_7
D_0 D_1 D_2 D_3 D_4 T_5 C_6 C_7	10	0xd2	D_0	D_1	D_2	D_3	D_4		C_6	C_7
D_0 D_1 D_2 D_3 D_4 D_5 T_6 C_7	10	0xe1	D_0	D_1	D_2	D_3	D_4	D_5		C_7
D_0 D_1 D_2 D_3 D_4 D_5 D_6 T_7	10	0xff	D_0	D_1	D_2	D_3	D_4	D_5	D_6	

UG035_ch3_23_102403

Note: The C's in the Input Data column are the XGMII (8-bit) versions whereas the C's in the Block Payload column are the 10GBASE-R (7-bit) versions.

Figure 3: **64B/66B Block Formats**

Table 2: **Control Codes**

Control Character	Notation	XGMII Control Code	10GBASE-R Control Code	10GBASE-R 0 Code	8B/10B Code
idle	/I/	0x07	0x00	-	K28.0, K28.3, or K28.5
start	/S/	0xFB	encoded by block type field	-	K27.7
terminate	/T/	0xFD	encoded by block type field	-	K29.7
error	/E/	0xFE	0x1E	-	K30.7
Sequence ordered_set	/Q/	0x9C	encoded by block type field plus O mode	0x0	K28.4
reserved0	/R/	0x1C	0x2D	-	K28.0
reserved1	-	0x3C	0x33	-	K28.1
reserved2	/N/	0x7C	0x4B	-	K28.3
reserved3	/K/	0xBC	0x55	-	K28.5
reserved4	-	0xDC	0x66	-	K28.6
reserved5	-	0xF7	0x78	-	K23.7
Signal ordered_set	/Fsig/	0x5C	encoded by block type field plus O mode	0xF	K28.2

The encoder is implemented in slices in the FPGA fabric. The reference design decodes the 64-bit XGMII data on a per 8-bit lane basis (all in parallel). Each lane can be considered as either data or control depending on the state of its associated control bit. XGMII_TXC[0] contains information on lane 0 (XGMII_TXD[7:0]), XGMIII_TXC[1] contains information on lane 1 (XGMII_TXD[15:8]), and so on up to XGMII_TXC[7] giving information on lane 7 (XGMII_TXD[63:56]).

Operation through the encoder is as follows. First XGMII_TXD and XGMII_TXC are pipelined (twice) for cases when XGMII data is not a control byte. XGMII_TXD and XGMII_TXC also feed the LANE DECODE block that first checks each data lane and indicate if the lane contains data or control. This block also checks for valid control codes:

- Idle
- Terminate
- Start (lane 0 or 4 only)
- Sequence ordered set (lane 0 or 4 only)
- Signal ordered set (lane 0 or 4 only)
- Error
- Five reserved code words

Internal signals, such as lanex_data, lanex_idle, and lanex_terminate, hold information on the type of data in each lane. These signals are passed on to the BLOCK TYPE FIELD GENERATION logic.

The BLOCK TYPE FIELD GENERATION logic determines the type for each byte, such as type_1e, type_2d, and type_33. These internal types are passed on to the REG_TYPE block where the TYPE_REG bus is created. The TYPE_REG bus is used by the TYPE_FIELD_GEN block, SYNC_FIELD_GEN block, and the multiplexer that feeds the DATA_FIELD block.

At the same time, the CODE GENERATION block creates the 10GBASE-R control code based on the type of byte passed. For example, if an idle character (shown in Table 2 as 0x07 in XGMII format) is indicated, the CODE GENERATION block outputs a `0x00` 10GBASE-R control code to the data field multiplexer.

Next the TYPE_FIELD_GEN block uses the TYPE_REG bus to determine the type field value to locate in bits [9:2] of the block payload shown in Figure 3. Also the SYNC_FIELD_GEN block reads the TYPE_REG bus to determine the sync field to send (shown as bits [1:0] in Figure 3).

Finally, the DATA_FIELD block registers the data from the data multiplexer whose output is a combination of data and 10GBASE-R control codes, also selected by the TYPE_REG bus.

The outputs of the DATA_FIELD, TYPE_FIELD_GEN, and SYNC_FIELD_GEN blocks combine to form the 66-bit encoded data bus (DATA_OUT) shown as the block payload in Figure 3.

The design is reset asynchronously when the INIT signal is asserted High. If the ENABLE signal is deasserted, then the DATA_OUT output is frozen at its last clocked value.

Decoder

Figure 4 shows a block diagram of the decoder.

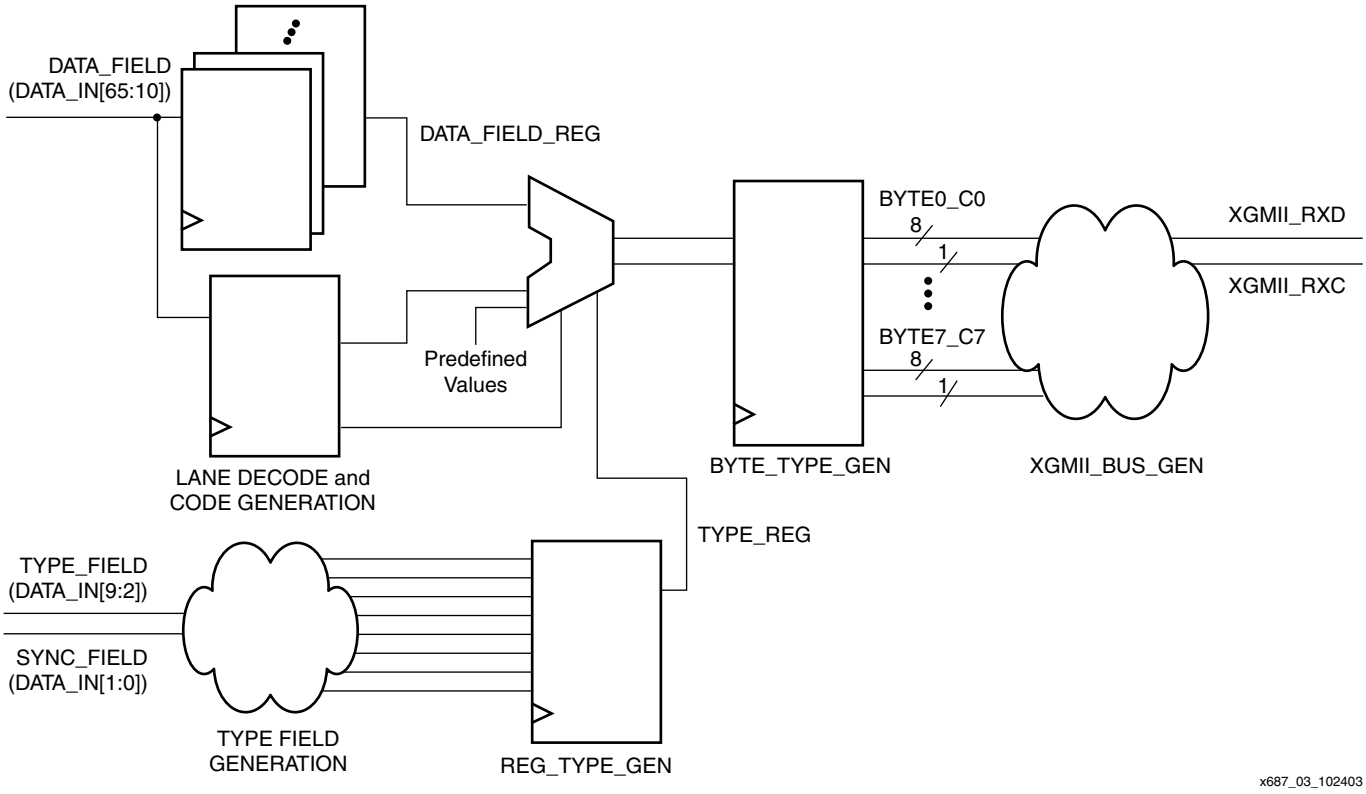


Figure 4: **Decoder Block Diagram**

Table 3 defines all decoder block signals.

Table 3: **Decoder Signals**

Port	I/O	Description
CLK	Input	Clocks all the decoder logic
DATA_IN	Input [65:0]	Encoded data bus (consists of the DATA_FIELD, TYPE_FIELD, and SYNC_FIELD bits)
IDLE_BUS	Output [7:0]	Indicates if an idle is contained in that lane
INIT	Input	Asynchronous reset (active High)
R_TYPE	Output	Indicates what type of sequence is received
SYNC_LOCK	Input	Indicates when data is synchronized (active High)
XGMII_RXC	Output [7:0]	XGMII received control bus
XGMII_RXD	Output [63:0]	XGMII received data

The decoder recovers the XGMII data and control signals (XGMII_RXD and XGMII_RXC) from the 66-bit code words that are input on the DATA_IN port. This data consists of a SYNC_FIELD (DATA_IN [1:0]), a TYPE_FIELD (DATA_IN [9:2]), and a DATA_FIELD (DATA_IN [65:10]). As in the encoder, the decoder works on a lane by lane basis (all in parallel). The data field is pipelined before being input into the multiplexer.

The data field is also input into the LANE DECODE and CODE GENERATION block (see Figure 4). This block checks for the 7-bit 10GBASE-R control code in each lane. If a 10GBASE-R control character is present, the control_x signals are set to the 8-bit XGMII control code. For example if the block received the idle character in lane 1 (`0x00`), the control_1 signal is set to `0x07` as shown in Table 2.

The TYPE FIELD generation logic decodes the type of data on the input bus by looking at the TYPE_FIELD bits of the DATA_IN [9:2] and SYNC_FIELD bits DATA_IN [1:0] shown in Figure 4. This logic creates signals such as type_66, type_55, and type_4b, which are used by the REG_TYPE_GEN block.

The REG_TYPE_GEN block creates the TYPE_REG bus, which selects whether DATA (from the DATA_FIELD_REG bus) or XGMII control codes (created from the LANE DECODE and CODE GENERATION block) is passed on for each lane into the XGMII_BUS_GEN logic. The XGMII_BUS_GEN logic creates the XGMII_RXD and XGMII_RXC buses.

The design is reset asynchronously when the INIT signal is asserted High or the SYNC_LOCK port is Low. The full 10GBASE-R design uses this reset capability to prevent the decoder from operating on data that is not synchronized.

Timing Diagrams

Figure 5 shows the timing relationship between unencoded data into the encoder and encoded data out of the encoder block.

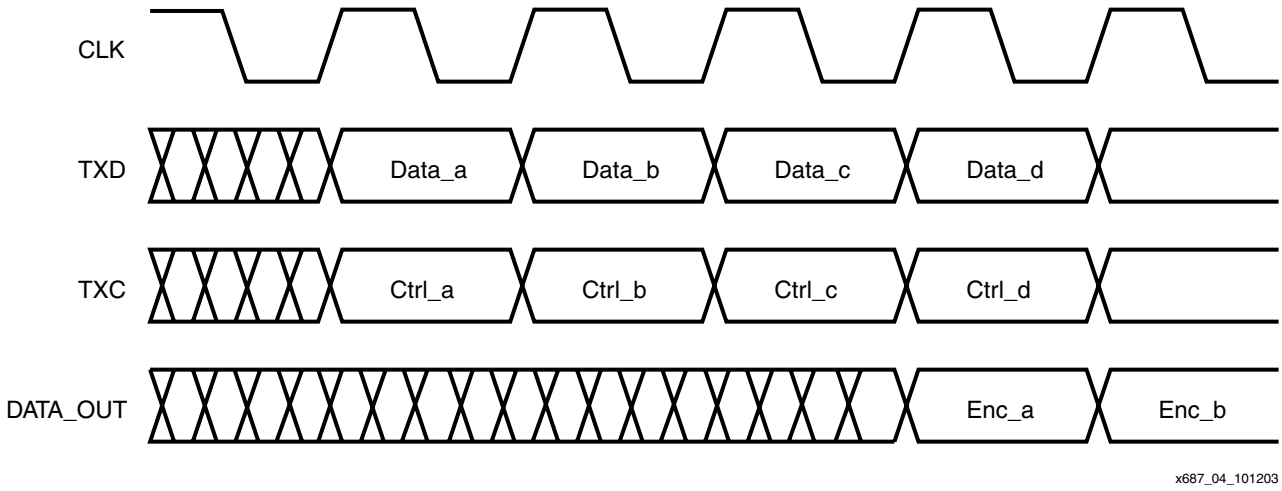


Figure 5: **Encoder Timing Waveforms**

Figure 6 shows the timing relationship between encoded data into the decoder and unencoded data out of the decoder block

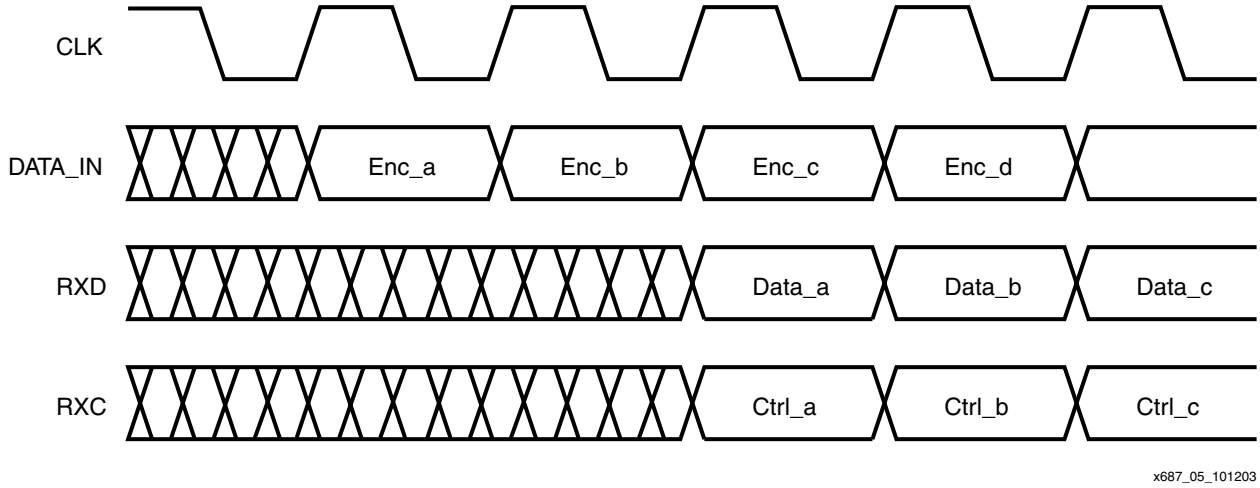


Figure 6: **Decoder Timing Waveforms**

Reference Design Simulation

Mentor Graphics Corporation ModelSim SE, version 5.7d, was used to simulate both Verilog and VHDL reference designs. The SE version supports mixed language simulation. Four testbenches are provided with this reference design: two for the Verilog design (one functional, one timing) and two for the VHDL design (one functional, one timing). These testbenches are located in the `design/testbench` directory.

The testbenches send the block formats defined in Figure 3 with the sequences shown in Table 2. Then the encoded data is sent to the decoder block where it is decoded and verified. The following sample display shows properly encoded and decoded data:

```
# ******************Start sending data to encoder
# -------------------------->
#  *****************IDLE received
#  Time =               920256
# -------------------------->
#  *****************START received
# -------------------------->
#  *****************DATA received
# -------------------------->
#  *****************TERMINATION received
# Sending error -->start in wrong octet
# ******************************************
# *************Completed sending sets**********
# ******************************************
# -------------------------->
#  *****************RESERVED received
# -------------------------->
#  *****************LAST RESERVED received
# -------------------------->
#  **************ERROR DATA set received
# -------------------------->
#  **************All DATA received simulation complete
```

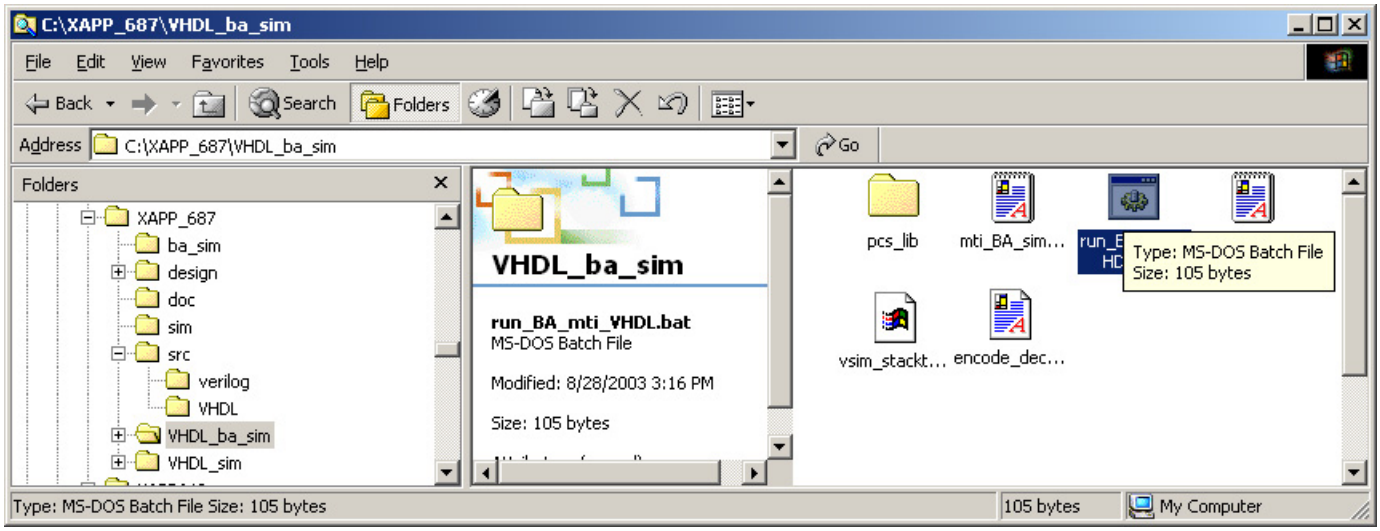
There are four directories for running the simulations. The two Verilog directories are:

- `sim` (functional)
- `ba_sim` (timing)

The two VHDL directories are:

- `VHDL_sim` (functional)
- `VHDL_ba_sim` (timing)

For example, to begin VHDL timing simulation, double-click on the batch file icon as shown in Figure 7. The directory structure also is shown.



x687_06_101203

Figure 7: **VHDL Timing Simulation Batch File**

Each batch file compiles all the necessary files and runs the simulation for the time needed to complete the data checking.

Design Utilization

Table 4 shows the design resource utilization. The reference design was implemented using Synplify 7.2/XST and ISE 6.1 SP1.

Note: Numbers may vary depending on synthesis tool and tool settings.

Table 4: **Design Resource Utilization**

	FFs	LUTs	Slices	Period (ns)
Encoder/Decoder	511	1114	687	6.3
XC2VP7 Utilization	5%	11%	13%	-

References

The following documents provide additional material related to this application note:

- Virtex-II Pro Platform FPGA User Guide
  http://www.xilinx.com/bvdocs/userguides/ug012.pdf
- IEEE Std 802.3ae-2002 section 49
  http://standards.ieee.org/

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
11/21/03	1.0	Initial Xilinx release.