Replace Your Am7968 TAXI™ Transmitter With a CY7B923 HOTLink™

Introduction

The TAXI™ family of data communications parts was one of the first to provide the benefits of high-speed serial transport of parallel information. Because of its flexibility and wide data-rate range, it has found usage in numerous commercial and military applications.

Time, however, has moved on and the original TAXI has in many cases been left behind. The Am7968 is a full bipolar design and consumes over 1W while newer components, like the Cypress HOTLink™, are capable of operating at twice the data rate and less than half the power. In addition, the military version of the Am7968 has been discontinued, leaving numerous designs in jeopardy.

Fortunately, a relatively simple replacement is available for the Am7968 that (in most cases) requires little or no change in surrounding system logic, *including* the Am7969 TAXI receiver. This simple replacement uses the Cypress CY7B923 HOTLink Transmitter, along with a small PLD, to form a logic and timing equivalent replacement. The use of such a replacement allows the continued use and manufacture of these legacy systems with minimal impact to the equipment and system interconnect.

Overview

The Am7968 TAXI transmitter, when operating in 8-bit mode, uses a 4B/5B encoding scheme to convert input data and commands into a form suitable for serial transmission and clock recovery. Communication with an existing Am7969 TAXI receiver requires the use of this same encoding scheme, presented in the same form and data-rate as that generated by the Am7968. By operating the CY7B923 HOTLink Transmitter in Bypass mode (unencoded 10-bit data path) mated to a small PLD, it is possible to exactly emulate the 4B/5B encoding used by the Am7968.

Am7968 Functionality

The Am7968 is both very similar to the HOTLink transmitter, and very different. Both parts communicate serially over a differential PECL (Positive ECL) link. Both parts employ a PLL clock multiplier to change a slow byte-rate clock into a fast bit-rate clock. However, most of the similarity ends here.

Data Encoding

Unlike HOTLink, which normally operates with an 8B/10B DC-balanced code, the Am7968 encodes its data stream using a 4B/5B algorithm standardized for use with the FDDI (Fiber Distributed Data Interface). This encoding converts four bits of parallel data into five bits of serial data. With such a small a code set to work with, it is not possible to maintain a DC-balance in the data stream. To improve this somewhat, the Am7968 also performs an NRZI (non-return-to-zero, invert on ones) encoding of the serial data.

4B/5B Encoding

The data is encoded to ensure a minimum density of transitions in the serial interface. These transitions are necessary to allow the receive end of the serial link to locate the boundaries of bits on the serial interface. Without this (or a similar) encoding, transmission of a long string of zeros or ones would turn into a DC level on the serial interface. Without any transitions to identify some of the bit boundaries, the receiver clock would eventually drift slightly in frequency and capture incorrect information from the serial interface.

The 4B/5B encoding used with the Am7968 allows all sixteen possible 4-bit data groupings to be represented by 5-bit patterns that all contain transitions. Since the complete 5-bit data space actually contains a total 32 possible combinations, only half of the available patterns are used to represent data. These data combinations are listed in *Table 1*.

Table 1. 4B/5B/NRZI Data Encoding

HEX Data	Binary Data	4B/5B Encoded	0-Carry NRZI	1-Carry NRZI
0	0000	11110	10100	01011
1	0001	01001	01110	10001
2	0010	10100	11000	00111
3	0011	10101	11001	00110
4	0100	01010	01100	10011
5	0101	01011	01101	10010
6	0110	01110	01011	10100
7	0111	01111	01010	01010
8	1000	10010	11100	00011
9	1001	10011	11101	00010
A	1010	10110	11011	00100
B	1011	10111	11010	00101
C	1100	11010	10011	01100
D	1101	11011	10010	01101
E	1110	11100	10111	01000
F	1111	11101	10110	01001

NRZI Encoding

In addition to converting the parallel 4-bit data into serial 5-bit data, a second level of encoding is added to improve its signaling characteristics. This encoding (called NRZI) removes the need to know if a transmitted bit was sent as a one or a zero. This is done by converting 1-bits into inversions in the serial stream, while 0-bits maintain the same HIGH or LOW signal level. Because all 1 and 0 information is now determined only by transitions (not by active level), the serial receiver can now correctly decode the serial data stream even if the differential inputs are swapped.

An example of an NRZI-encoded serial stream and encoder is shown in *Figure 1*. Two different output streams are shown in the figure. Which of the two streams is actually generated is determined by the state of the encoder flip-flop when the NRZI encoding of the current character is started. The two possible NRZI encodings of each 4B/5B data character are also listed in *Table 2*. Notice that these two columns are the exact inverse of each other.

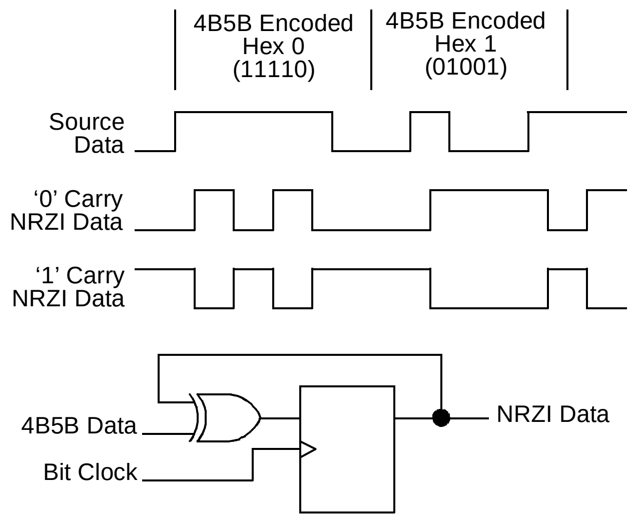

Figure 1. NRZI Encoder

Am7968 Commands

The 4B/5B code makes use of specific patterns from a 32-symbol space. Of these 32 possible symbols, sixteen are allocated to represent the hex data values x'0' through x'F'. This leaves sixteen additional 5-bit patterns that can be assigned meanings other than data.

For the Am7968, eight of the remaining sixteen patterns are used to define synchronization and in-band command codes that can be used for various interface control functions. These eight patterns are identified as other alphabetic letters, similar to the hexadecimal characters greater than 9. These control code names and their associated encodings are listed in *Table 2*.

Table 2. 4B/5B/NRZI Control Code Encoding

Control Code	4B/5B Encoded	0-Carry NRZI	1-Carry NRZI
H	00100	00111	11000
I	11111	10101	01010
J	11000	10000	01111
K	10001	11110	00001
Q	00000	00000	11111
R	00111	00101	11010
S	11001	10001	01110
T	01101	01001	10110

Unlike the data characters, which can be combined in any fashion to transmit bytes of information, the Control Codes are only defined for use in specific pair combinations. These control code pairings are generated when specific combinations of bits are present on the four command input lines to the Am7968. These command input groupings are listed in *Table 3*.

Table 3. Am7968 Command Codes

HEX Command	Binary Command	Control Code Pair
0	0000	Data
No STRB	No STRB	JK (8-bit Sync)
1	0001	II
2	0010	TT
3	0011	TS
4	0100	IH
5	0101	TR
6	0110	SR
7	0111	SS
8	1000	HH
9	1001	HI
A	1010	HQ
B	1011	RR
C	1100	RS
D	1101	QH
E	1110	QI
F	1111	QQ

Am7968 Control Signals

A block diagram of the Am7968 is shown in *Figure 2*. This figure shows the control signals and data/command buses used to control the part. Unlike the CY7B923 HOTLink transmitter (see *Figure 3*), the Am7968 has separate input buses for data and commands. The data input bus is eight bits in width while the command bus is only four bits wide.

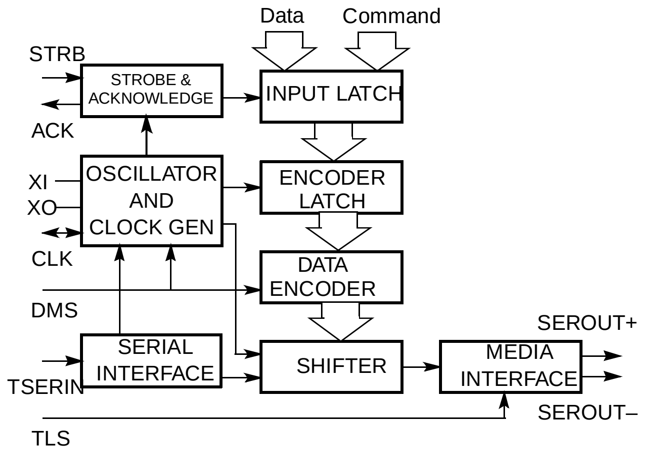

Figure 2. Am7968 Logic Diagram

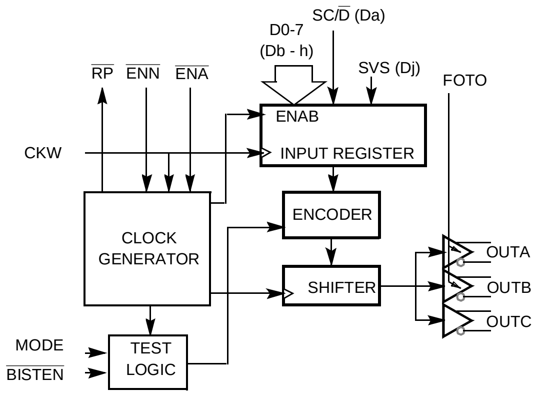

Figure 3. CY7B923 Transmitter Logic Diagram

Loading of data into the Am7968 is also handled differently. This is performed using the STRB input to clock the information present in the data and command buses into the Am7968. This STRB signal may be semi-asynchronous to the normal transmitter reference clock on the X1 input.

To operate the Am7968 at or near its reference clock byte rate it is necessary to strobe data into the part with much more care than when operating at slower rates. There is, in effect, a “stayout” area around the falling edge of the reference clock where data and commands should not be strobed into the part.

HOTLink Emulation of Am7968

To create a drop-in replacement for a part, it is necessary to present an interface to the host system that contains the same signals, clocks, and timing as the logic element being replaced. In the case of the Am7968, the critical signals used for operation are

- DI[7:0]-eight-bit data bus
- CI[3:0]-four-bit command bus
- STRB-data strobe
- ACK-data strobe acknowledge
- ±SEROUT-differential PECL serial data
- X1-external byte reference clock

While there are other signals present on the Am7968, they are primarily static signals used for configuration.

Emulator Block Diagram

The emulator is built from two components, as shown in *Figure 4*: a CY7C343 EPLD that performs the 4B/5B and NRZI encoding, and a CY7B923 HOTLink transmitter to sequence the bits and drive the serial PECL interface. This two-chip design assumes that double frequency byte clock is present in the system to clock both the EPLD and the HOTLink transmitter. For those systems that only have the byte-rate clock present, it is possible to generate the 2x clock using a single Cypress CY7B991 RoboClock Programmable Skew Clock Buffer.

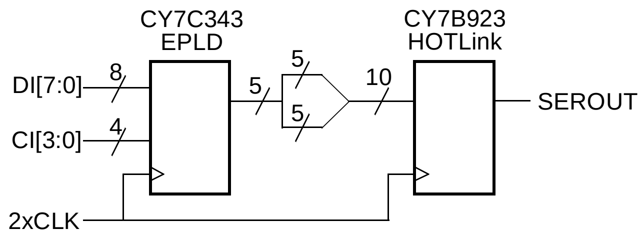

Figure 4. Am7968 Emulator Block Diagram

The 2x clock is necessary in the system because the HOTLink transmitter is normally only capable of sequencing bits with the data rate range of 160 to 330 Mbits/second. This is significantly faster than the maximum 125-Mbit/second data rate of the Am7968 transmitter. To allow the HOTLink transmitter to generate a serial stream that is data-rate compatible with an attached Am7969 receiver requires sequencing out bits in pairs. This effectively cuts the data rate of the transmitter in half. This timing relationship is shown in *Figure 5.*

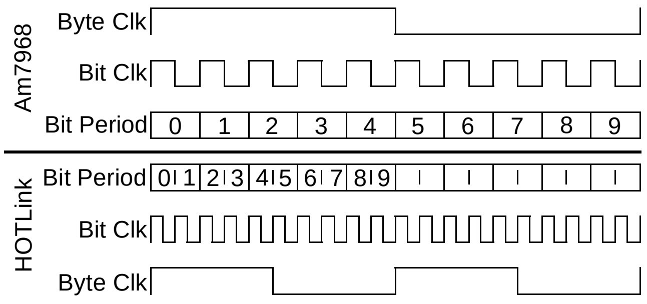

Figure 5. Am7968 vs. CY7B923 Bit Timing

This bit timing is accomplished by having the encoder EPLD generate only five NRZI bits on each 2x clock cycle. Each of these five bits is attached to two adjacent bit-inputs on the HOTLink transmitter. For example, encoder output bit 0 would be wired to HOTLink transmitter bits 0 and 1.

Emulator PLD Block Diagram

The majority of the emulator signals are on the parallel TTL-compatible side of the design. These parallel signals (all except the PECL ±SEROUT signals) all tie into the CY7C343 control EPLD. This EPLD performs all the data capture, 4B/5B encoding, NRZI encoding, and byte timing for the emulator. A block diagram of the internal functions of the EPLD is shown in *Figure 6.*

The logic is effectively split into five major sections. These sections control the data capture, holding register, 4B/5B/NRZI encoding, NRZI carry encoding, and clocking.

Control EPLD Operation

Data Capture Register

Data is loaded into the 12-bit Data Capture register on the rising edge of any STRB pulse. Once latched, the contents of the CI[3:0] bits determine what data is fed to the Merged Data/Command register. If any of the CI[3:0] bits are HIGH the CI bus is fed to both the upper and lower halves of the register. If all CI bits are LOW, the DI[7:0] data bus is fed to the register instead.

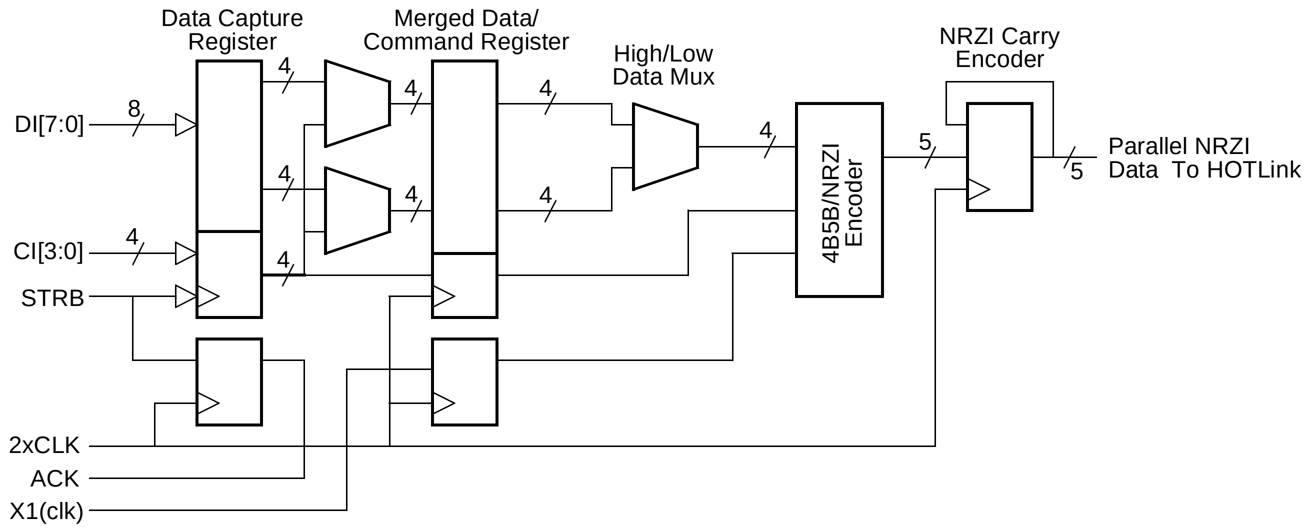

Figure 6. 4B/5B/NRZI Encoder PLD Block Diagram

Merged Data/Command Register

The Merged Data/Command register is a 9-bit register that is loaded every other cycle of the 2xCLK. The upper eight bits of this register are loaded with the output of the multiplexer from the data Capture register. The lowest bit identifies if the data in the register is a command or data.

If a STRB has occurred to load data into the Data Capture register during the previous cycle, that information is clocked into the Merged Data/Command register. If a STRB has not occurred, then a x'00' command is forced into the Merged Data/Command register.

4B/5B/NRZI Encoder

The data in the Merged Data/Command register is sequenced through the 4B/5B/NRZI encoder in two four-bit groups. The first group encodes the upper four bits of the command or data byte, while the second group encodes the lower four bits. In addition to the data bits, the encoder also needs to know if the bits represent a command or data, and (for commands) if the information is the upper or lower half-byte.

The NRZI output of the encoder assumes a zero for the starting or carry-in state of the NRZI encode operation. By pre-encoding the NRZI information, a large number of XOR gates can be removed from the design.

NRZI Carry Encoder

To generate the correct NRZI sequence it is necessary to track the state of the previous bit in the output sequence. This is done by feeding the most significant bit of the output register back to the input of the register, and XORing it with the next five bits of information. This effectively performs a selective inversion of the pre-encoded NRZI data. This inversion allows the data output to follow the NRZI encoding listed in *Tables 1* and 2.

Clocking

In the implementation documented here, this design uses two independent clocks: one for the STRB signal and the 2xCLK for the remainder of the logic. In addition to these two clocks, the EPLD monitors the X1 clock to determine which phase of the 2xCLK to capture and mux the internal data.

Conclusion

This design implements a two-chip drop-in replacement for the Am7968 TAXI transmitter. The design makes use of programmable logic to implement an external encoder that mimics the interface and timing of the Am7968.

The control EPLD was implemented using a CY7C343 EPLD. This PLD was designed and coded with VHDL (VHSIC Hardware Description Language), and compiled and simulated using the Cypress *Warp3*® tool. The full source code for the design is present in Appendix A of this application note, and is available from the Cypress electronic Bulletin Board System (BBS).

For those Am7968-based systems that are truly synchronous in nature, this design may be modified to operate with a single clock, and allow usage of the FLASH370™ family of CPLDs in addition to the CY7C34x series.

Because of the modularity and reusability of VHDL code, it is possible to incorporate the code in Appendix A with additional functionality in larger or more complex CPLDs or FPGAs, thereby reducing the hardware impact of this emulation to a reprogrammed logic part and a simple replacement of the Am7968 with the more capable CY7B923. Such a system would then be able to support a much faster data rate in the future with the simple reprogramming of the controlling PLD.

References

1. Cypress Semiconductor, CY7B923/CY7B933 HOTLink Transmitter/Receiver Data Sheet, Cypress Semiconductor *Data Book*, May, 1995.
2. Cypress Semiconductor, *HOTLink User's Guide*, 3rd Edition, April 1999.
3. Advanced Micro Devices, TAXIchip Integrated Circuits Transparent Asynchronous Transmitter/Receiver Interface Am7968/Am7969-125 Am7968/Am7969-175 Data Sheet and Technical Manual, 1992.

Appendix A. 4B/5B Encoder PLD

```
-- TAXI8SM.VHD

-- This design describes the operation of a PLD used to convert a
-- standard HOTLink transmitter (CY7B923) into a part set equivalent
-- to the older AMD TAXI-125. This PLD only emulates the TAXI
-- in 8-bit mode (dual 4B/5B encoders).

-- This design only operates in the standard synchronous mode
-- of the TAXI, as it does not contain any FIFO stages. It does
-- correctly generate all 16 TAXI command codes present. It does
-- not support cascade mode.
ENTITY taxi8top IS PORT (
   -- TAXI Parallel-side pins
   clk: IN BIT;                        -- PLD Clock, 2X multiple of
                                       --  standard TAXI clock
   sys_clk: IN BIT;                    -- standard TAXI clock, sampled
                                       --  by the PLD for phase alignment
   strobe: IN BIT;                     -- TAXI data load clock, used
                                       -- to control loading of the
                                       -- input register. Needs to
                                       -- be interruptible to force
                                       -- generation of SYNC codes
   D_In: IN BIT_VECTOR(0 TO 7);        -- data input bus
   CL: IN BIT_VECTOR(0 TO 3);          -- command input bus
   -- HOTLink parallel-side pins
   D_Out: OUT BIT_VECTOR(0 TO 4)       -- HOTLink data inputs, two/pin
   );
ATTRIBUTE part_name OF taxi8top:ENTITY IS “C343";
END taxi8top;

USE work.cypress.all;
USE work.table_bv.all;
USE work.rtlpkg.all;
USE work.memorypkg.all;

ARCHITECTURE struct OF taxi8top IS
-- add internal signals
SIGNAL outreg : BIT_VECTOR(0 TO 4);    -- output data register
SIGNAL encode : BIT_VECTOR(0 TO 4);    -- 4B/5B/NRZI  encoder output
SIGNAL xreg   : BIT_VECTOR(0 TO 4);    -- output XOR register
SIGNAL in_reg : BIT_VECTOR(0 TO 11);   -- 12-bit input register
SIGNAL hld_reg: BIT_VECTOR(0 TO 8);    -- data input hold register
SIGNAL in_data: BIT_VECTOR(0 TO 5);    -- encoder input
SIGNAL strb_in: BIT;                   -- strobe received flag
SIGNAL strb_n:  BIT;                   -- inverted strobe
SIGNAL phase1:  BIT;                   -- hold enable for STROBE in
```

Appendix A. 4B/5B Encoder PLD (continued)

```
-- 4B/5B encoder data constants
-- data half-bytes
CONSTANT DI_0: x01_VECTOR(0 TO 4) := “00000";
CONSTANT DI_1: x01_VECTOR(0 TO 4) := “00001";
CONSTANT DI_2: x01_VECTOR(0 TO 4) := “00010";
CONSTANT DI_3: x01_VECTOR(0 TO 4) := “00011";
CONSTANT DI_4: x01_VECTOR(0 TO 4) := “00100";
CONSTANT DI_5: x01_VECTOR(0 TO 4) := “00101";
CONSTANT DI_6: x01_VECTOR(0 TO 4) := “00110";
CONSTANT DI_7: x01_VECTOR(0 TO 4) := “00111";
CONSTANT DI_8: x01_VECTOR(0 TO 4) := “01000";
CONSTANT DI_9: x01_VECTOR(0 TO 4) := “01001";
CONSTANT DI_A: x01_VECTOR(0 TO 4) := “01010";
CONSTANT DI_B: x01_VECTOR(0 TO 4) := “01011";
CONSTANT DI_C: x01_VECTOR(0 TO 4) := “01100";
CONSTANT DI_D: x01_VECTOR(0 TO 4) := “01101";
CONSTANT DI_E: x01_VECTOR(0 TO 4) := “01110";
CONSTANT DI_F: x01_VECTOR(0 TO 4) := “01111";

-- command constants
CONSTANT CI_0: x01_VECTOR(0 TO 4) := “10000";
CONSTANT CI_1: x01_VECTOR(0 TO 4) := “10001";
CONSTANT CI_2: x01_VECTOR(0 TO 4) := “10010";
CONSTANT CI_3: x01_VECTOR(0 TO 4) := “10011";
CONSTANT CI_4: x01_VECTOR(0 TO 4) := “10100";
CONSTANT CI_5: x01_VECTOR(0 TO 4) := “10101";
CONSTANT CI_6: x01_VECTOR(0 TO 4) := “10110";
CONSTANT CI_7: x01_VECTOR(0 TO 4) := “10111";
CONSTANT CI_8: x01_VECTOR(0 TO 4) := “11000";
CONSTANT CI_9: x01_VECTOR(0 TO 4) := “11001";
CONSTANT CI_A: x01_VECTOR(0 TO 4) := “11010";
CONSTANT CI_B: x01_VECTOR(0 TO 4) := “11011";
CONSTANT CI_C: x01_VECTOR(0 TO 4) := “11100";
CONSTANT CI_D: x01_VECTOR(0 TO 4) := “11101";
CONSTANT CI_E: x01_VECTOR(0 TO 4) := “11110";
CONSTANT CI_F: x01_VECTOR(0 TO 4) := “11111";

-- data output constants
-- zero carry-in, NRZI encoded
CONSTANT DO_0: x01_VECTOR(0 TO 4) := “10100";   -- 11110 4B/5B
CONSTANT DO_1: x01_VECTOR(0 TO 4) := “01110";   -- 01001 4B/5B
CONSTANT DO_2: x01_VECTOR(0 TO 4) := “11000";   -- 10100 4B/5B
CONSTANT DO_3: x01_VECTOR(0 TO 4) := “11001";   -- 10101 4B/5B
CONSTANT DO_4: x01_VECTOR(0 TO 4) := “01100";   -- 01010 4B/5B
CONSTANT DO_5: x01_VECTOR(0 TO 4) := “01101";   -- 01011 4B/5B
CONSTANT DO_6: x01_VECTOR(0 TO 4) := “01011";   -- 01110 4B/5B
CONSTANT DO_7: x01_VECTOR(0 TO 4) := “01010";   -- 01111 4B/5B
```

Appendix A. 4B/5B Encoder PLD (continued)

```
CONSTANT DO_8: x01_VECTOR(0 TO 4) := “11100";   -- 10010 4B/5B
CONSTANT DO_9: x01_VECTOR(0 TO 4) := “11101";   -- 10011 4B/5B
CONSTANT DO_A: x01_VECTOR(0 TO 4) := “11011";   -- 10110 4B/5B
CONSTANT DO_B: x01_VECTOR(0 TO 4) := “11010";   -- 10111 4B/5B
CONSTANT DO_C: x01_VECTOR(0 TO 4) := “10011";   -- 11010 4B/5B
CONSTANT DO_D: x01_VECTOR(0 TO 4) := “10010";   -- 11011 4B/5B
CONSTANT DO_E: x01_VECTOR(0 TO 4) := “10111";   -- 11100 4B/5B
CONSTANT DO_F: x01_VECTOR(0 TO 4) := “10110";   -- 11101 4B/5B
CONSTANT DO_H: x01_VECTOR(0 TO 4) := “00111";   -- 00100 4B/5B
CONSTANT DO_I: x01_VECTOR(0 TO 4) := “10101";   -- 11111 4B/5B
CONSTANT DO_J: x01_VECTOR(0 TO 4) := “10000";   -- 11000 4B/5B
CONSTANT DO_K: x01_VECTOR(0 TO 4) := “11110";   -- 10001 4B/5B
CONSTANT DO_Q: x01_VECTOR(0 TO 4) := “00000";   -- 00000 4B/5B
CONSTANT DO_R: x01_VECTOR(0 TO 4) := “00101";   -- 00111 4B/5B
CONSTANT DO_S: x01_VECTOR(0 TO 4) := “10001";   -- 11001 4B/5B
CONSTANT DO_T: x01_VECTOR(0 TO 4) := “01001";   -- 01101 4B/5B

-- generate decoder table
CONSTANT table: x01_TABLE(0 TO 41, 0 TO 10) := (
-- data mappings
--
--Input   HI_LO   Output
-------   -----   ------
   DI_0  &  ’x’  &  DO_0,
   DI_1  &  ’x’  &  DO_1,
   DI_2  &  ’x’  &  DO_2,
   DI_3  &  ’x’  &  DO_3,
   DI_4  &  ’x’  &  DO_4,
   DI_5  &  ’x’  &  DO_5,
   DI_6  &  ’x’  &  DO_6,
   DI_7  &  ’x’  &  DO_7,
   DI_8  &  ’x’  &  DO_8,
   DI_9  &  ’x’  &  DO_9,
   DI_A  &  ’x’  &  DO_A,
   DI_B  &  ’x’  &  DO_B,
   DI_C  &  ’x’  &  DO_C,
   DI_D  &  ’x’  &  DO_D,
   DI_E  &  ’x’  &  DO_E,
   DI_F  &  ’x’  &  DO_F,

   CI_0  &  ’1’  &  DO_J,
   CI_0  &  ’0’  &  DO_K,
   CI_1  &  ’x’  &  DO_I,
   CI_2  &  ’x’  &  DO_T,
   CI_3  &  ’1’  &  DO_T,
   CI_3  &  ’0’  &  DO_S,
   CI_4  &  ’1’  &  DO_I,
```

Appendix A. 4B/5B Encoder PLD (continued)

```
   CI_4  &  ’0’  &  DO_H,
   CI_5  &  ’1’  &  DO_T,
   CI_5  &  ’0’  &  DO_R,
   CI_6  &  ’1’  &  DO_S,
   CI_6  &  ’0’  &  DO_R,
   CI_7  &  ’x’  &  DO_S,
   CI_8  &  ’x’  &  DO_H,
   CI_9  &  ’1’  &  DO_H,
   CI_9  &  ’0’  &  DO_I,
   CI_A  &  ’1’  &  DO_H,
   CI_A  &  ’0’  &  DO_Q,
   CI_B  &  ’x’  &  DO_R,
   CI_C  &  ’1’  &  DO_R,
   CI_C  &  ’0’  &  DO_S,
   CI_D  &  ’1’  &  DO_Q,
   CI_D  &  ’0’  &  DO_H,
   CI_E  &  ’1’  &  DO_Q,
   CI_E  &  ’0’  &  DO_I,
   CI_F  &  ’x’  &  DO_Q);

BEGIN
-- declare input register.  Data is clocked by the external STROBE
-- signal.  This same strobe signal is used to synchronize the internal
-- two-state machine.

p1: PROCESS BEGIN
   WAIT UNTIL (strobe=’1’);
     in_reg(0 TO 7) <= D_In(0 TO 7);
     in_reg(8 TO 11) <= CL(0 TO 3);
END PROCESS p1;-- capture strobe event

-- async set when strobe is present
-- use synchronous clear from clk when part is set and sys_clk present
phase1 <= strb_in AND sys_clk;
st1: DSRFF PORT MAP  (phase1, strobe, zero, clk, strb_in);

-- setup input data hold register
p2: PROCESS BEGIN
   WAIT UNTIL (clk=’1’);
     IF sys_clk = ’0’ THEN       -- hold data
       hld_reg <= hld_reg;
     ELSIF strb_in=’0’ THEN       -- no data, load a SYNC command
       hld_reg <= “000000001";
     ELSIF (in_reg(8 TO 11) /= “0000") THEN  -- check for a command
       hld_reg(0 TO 3) <= in_reg(8 TO 11);
       hld_reg(4 TO 7) <= in_reg(8 TO 11);
       hld_reg(8) <= ’1’; -- set as a command
     ELSE
```

Appendix A. 4B/5B Encoder PLD (continued)

```
      hld_reg(0 TO 7) <= in_reg(0 TO 7);
      hld_reg(8) <= ’0’; -- set as data
   END IF;
END PROCESS p2;

-- declare data mux select for input to the 4B/5B encoder
p3: PROCESS (hld_reg, sys_clk)
   BEGIN
   in_data(5) <= NOT sys_clk;      -- hi/low nibble select
   in_data(0) <= hld_reg(8);
   IF sys_clk = ’0’ THEN           -- enable high nibble first
     in_data(1 TO 4) <= hld_reg(4 TO 7);
   ELSE
     in_data(1 TO 4) <= hld_reg(0 TO 3);
   END IF;
END PROCESS p3;

-- declare 4B/5B encoder
p4: PROCESS (in_data)
BEGIN
   encode <= ttf(table,(in_data));
END PROCESS p4;

-- declare output register

dr0: DFF PORT MAP (encode(0), clk, outreg(0));
dr1: DFF PORT MAP (encode(1), clk, outreg(1));
dr2: DFF PORT MAP (encode(2), clk, outreg(2));
dr3: DFF PORT MAP (encode(3), clk, outreg(3));
dr4: DFF PORT MAP (encode(4), clk, outreg(4));

dx0: XDFF PORT MAP (outreg(0), xreg(4), clk, xreg(0));
dx1: XDFF PORT MAP (outreg(1), xreg(4), clk, xreg(1));
dx2: XDFF PORT MAP (outreg(2), xreg(4), clk, xreg(2));
dx3: XDFF PORT MAP (outreg(3), xreg(4), clk, xreg(3));
dx4: XDFF PORT MAP (outreg(4), xreg(4), clk, xreg(4));

-- assign output register to outputs
D_Out <= xreg;
END struct;     -- end of top level design
```

AMD, TAXI, and TAXIchip are trademarks of Advanced Micro Devices.
FLASH370 and HOTLink are trademarks and *Warp3* is a registered trademark of Cypress Semiconductor.