XAPP288 (1.0) October 19, 2001

Serial Digital Interface (SDI) Video Decoder

Author: John F. Snow

Summary

The ANSI/SMPTE 259M-1997 standard specifies a serial digital interface (SDI) for digital video equipment operating at either the 525-line, 60 Hz video standard or the 625-line, 50 Hz video standard.[1] The SDI standard describes how to transport both composite and component digital video over standard video coax. SDI is widely accepted and often forms the video transportation "backbone" of television studios and broadcast centers.

The SDI standard can be broken down into three main functions: the encoder, the physical layer, and the decoder. This is one in a series of application notes describing SDI implementation in Xilinx FPGAs. Figure 1 shows the correlation between the various application notes and the elements of the SDI link.

This application note focuses on the SDI decoder. The reference design includes several implementations of the SDI decoder optimized for use with the Virtex™-II family and other Xilinx family features. Both serial (bit-rate) and parallel (word-rate) implementations of the SDI decoder are presented. Design examples are included to illustrate alternative solutions for standard SDI decoder devices, the National CLC011 and the Cypress CY7C9335, by using the decoder implementations developed in this application note.

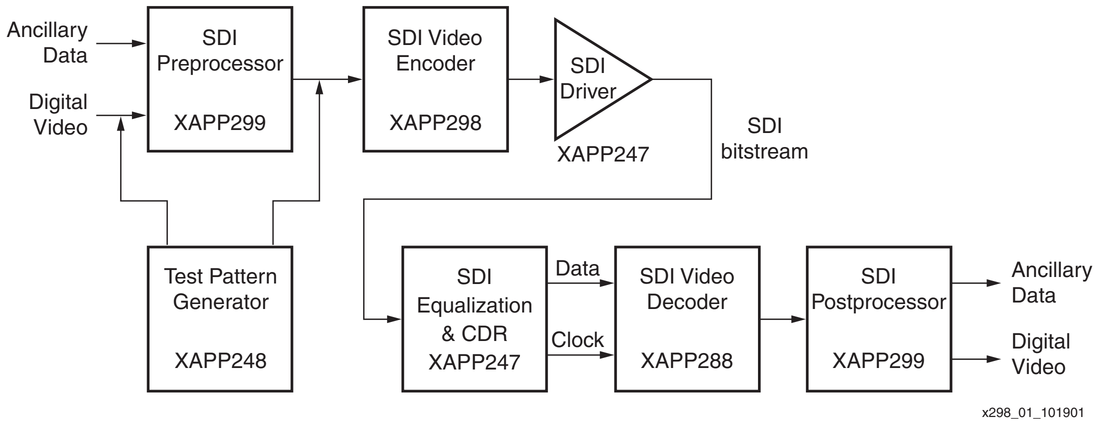

Figure 1: **SDI Block Diagram and Application Notes**

SDI Introduction

Digital Video Formats

The SDI standard describes how to transport standard definition digital video serially over a video coax cable. This standard describes the encoding and decoding processes performed on the video bitstream for transportation across the physical layer. The standard also describes the electrical and mechanical characteristics of the physical layer. However, it does not define the actual format of the digital video data. Refer to the following additional standards for the definition of SDI compatible digital video formats:

- ANSI/SMPTE 125M, ANSI/SMPTE 267M, and ITU-R BT.601-5 for 4 x 3 and 16 x 9 aspect ratio 4:2:2 component digital video [1][2]
- ANSI/SMPTE 244M for composite NTSC digital video [1]
- IEC 1179 (now called IEC 61179) for composite PAL digital video [3]

The SDI standard does not cover high-definition digital video. Another standard, SMPTE 292M, defines a serial digital interface standard for high-definition digital video, commonly called HD-SDI. The bandwidth requirements for high-definition video are significantly higher than for standard definition video. Also, the HD-SDI standard differs from the SDI standard in the way that the video components are interleaved. Because of these higher bandwidth requirements and format differences, the implementation of a HD-SDI decoder has different considerations than that of a standard definition SDI decoder and is not covered in this application note.

Any of the digital video formats supported by the SDI standard use either eight bits or ten bits per data word. The SDI standard is natively a ten-bit format, but allows eight-bit video to be transported across the interface.

Encoding and Decoding

Prior to sending digital video serially across the physical layer, a SDI transmitter must encode the video in accordance with the SDI standard. This encoding process is designed to insure that sufficient level transitions occur in the serial bitstream to allow the receiver to recover the clock and data. After the receiver captures the serial data, the decoder must reverse the encoding process to recover the original video data.

The SDI standard uses two generator polynomials, normally expressed as linear feedback shift registers (LFSR), to implement two separate encoding stages. First, the video bitstream is scrambled using the generator polynomial:

$$G_1(x) = x^9 + x^4 + 1$$

The output of this first encoding stage is referred to as the scrambled non-return-to-zero (NRZ) bitstream.

The second encoding stage uses the generator polynomial:

$$G_2(x) = x + 1$$

to convert the scrambled NRZ bitstream to a polarity-free scrambled NRZ-inverted (NRZI) bitstream. NRZI is DC balanced for transmission across the physical layer. If the bitstream is inverted between the transmitter and the receiver, then the polarity-free nature of the SDI bitstream allows the decoder to properly recover the original data.

The SDI decoder reverses the encoding process by using the same generator polynomials in reverse order: G_2 to convert from NRZI to NRZ and then G_1 to descramble the bitstream.

Figure 2 illustrates the encoding and decoding processes when implemented in LFSRs. Using standard LFSR notation, the circles with plus symbols inside are exclusive-OR gates. The boxes represent individual flip-flops. The LSB of a data word is sent first.

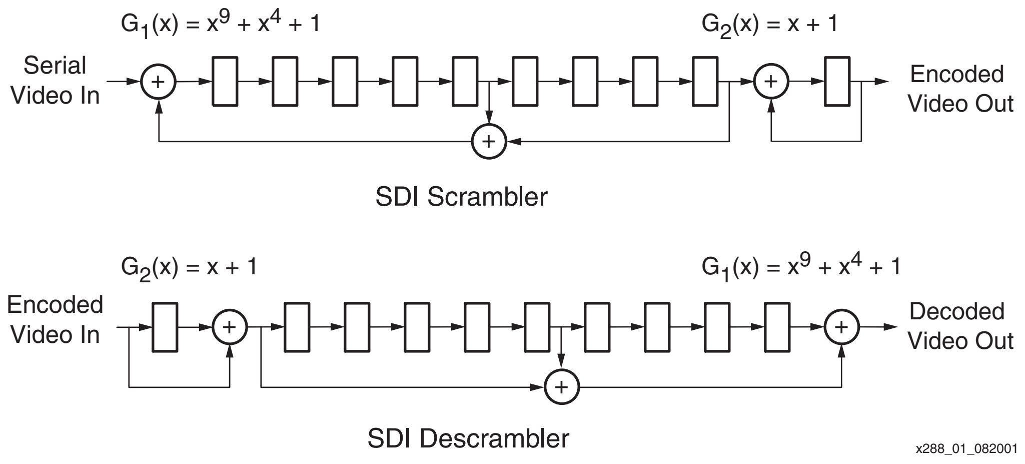

Figure 2: **Example SDI Encoder and Decoder Processes**

Framing

After decoding the video bitstream, the receiver determines where individual ten-bit words begin and end in the serial bitstream for data-word extraction. This process is called framing. In order to frame the bitstream, a unique and recognizable pattern must be sent periodically for the framer to use as a framing reference.

All of the digital-video formats supported by SDI share similar definitions for the timing reference signal (TRS) symbols. TRS symbols delineate between the active and inactive portions of the video. There are two TRS symbols sent per line of video: one at the start of active video called SAV, and one at the end of active video called EAV. A TRS symbol is sent as four consecutive words, formatted as:

```
3ff 000 000 XYZ
```

The first word transmitted is hexadecimal `3ff` and the last word is `XYZ`. The first three words of the TRS symbol, called the preamble, form a unique bit sequence in the bitstream. The fourth word, called `XYZ`, varies depending on the specific digital video format being transported.

Since the TRS preamble is common across all the supported digital video formats, is sent on a regular basis, and is unique in the bitstream, it is used as the framing reference. Upon detecting a sequence of ten consecutive ones and twenty consecutive zeros, the framer can determine the proper boundaries of all subsequent data words in the bitstream.

SDI Bit Rates

The bit rates supported by SDI range from 143 Mb/s to 360 Mb/s, depending on the digital video format being transported. The SDI standard defines four different bit rates and identifies their "support levels" as shown in Table 1. SDI compliant equipment is not required to support all bit rates. A piece of equipment supporting bit rates up to 270 Mb/s is said to conform to ANSI/SMPTE 259M-ABC, meaning it supports levels A, B, and C.

Table 1: **SDI Standard Bit Rates**

Support Level	Bit Rate	Video Format	Standard
Level A	143 Mb/s	NTSC composite	ANSI/SMPTE 244M-1995
Level B	177 Mb/s	PAL composite	IEC 61179
Level C	270 Mb/s	4 x 3 4:2:2 component	ANSI/SMPTE 125M-1995 and ITU-R BT.601-5
Level D	360 Mb/s	16 x 9 4:2:2 component	ANSI/SMPTE 267M-1995 and ITU-R BT.601-5

Error Detection

The SDI standard does not mandate the use of an error detection mechanism. Some of the digital video standards, SMPTE 125M for example, specify error detection bits in the XYZ word of the TRS symbol to determine the validity of the TRS symbol. However, the SDI standard highly recommends embedding error detection checkwords into the SDI video stream as described in SMPTE RP 165-1994.

At the receiving end of the SDI link, checksums are generated for the incoming bitstream and compared to the embedded checkwords. Error detection can only be done after the bitstream is descrambled and framed. An optional error detection module can simply be bolted onto the output of any of the framer modules described in this document. SDI error detection modules are described in XAPP299: Serial Digital Interface (SDI) Ancillary Data and EDH Processors.[6]

Reference Design

In the reference design available at ftp://ftp.xilinx.com/pub/applications/xapp/xapp288.zip, the descrambler and framer functions are implemented as separate modules. Serial and parallel implementations of both modules are provided.

Serial Descrambler Implementation: ser_descrambler.*

The descrambling process involves "division" of the incoming bitstream by the generator polynomials. This can be implemented very simply by a LFSR configured as shown in the SDI descrambler diagram in Figure 2, page 2. A serial implementation results in a very small amount of hardware. However, a serial implementation of the descrambler must run at the full bit rate of the SDI interface, up to 360 MHz. A serial implementation also requires the availability of a bit-rate clock for the FPGA. By using Virtex-II devices, a serial descrambler can be implemented to support SDI bit-rates up to 360 Mb/s.

The HDL files *ser_descrambler.** contain direct implementations of the descrambler LFSR described in Figure 2. As shown in **Reference Design Results**, this implementation occupies eleven flip-flops and two LUTs for both Virtex-II and Spartan-II devices. In Virtex-II devices, the serial descrambler runs fast enough to support the highest bit rate supported by the SDI standard. In a Spartan-II device, it supports the 270 Mb/s rate.

It is also be possible to implement the LFSR using the SRL16 feature found in the Virtex architecture. An SRL16-based implementation is smaller than the implementation presented here, potentially about half the size. However, it is more difficult to get an SRL16-based implementation to run at the highest SDI bit-rates.

The *ser_descrambler* module contains two control inputs, *NRZI* and *DESC*, to enable the NRZI-to-NRZ conversion and the descrambler, respectively. These control signals allow the two functions to be bypassed if the incoming data is non-SDI compliant. In normal SDI operation, both inputs should be tied High.

Parallel Descrambler Implementation: par_descrambler.*

The descrambler function can also be implemented in a parallel manner, processing one ten-bit word every clock cycle.This obviously requires more hardware but only needs to run at one-tenth the bit rate of the SDI link.

In some situations it is advantageous to use a larger parallel descrambler implementation. The FPGA could receive parallel data from some external SDI receiver device performing a serial-to-parallel conversion of the data. It may also be more economical to use a parallel implementation. Since the parallel descrambler only has to run at the word rate, lower performance FPGAs can be used to support the highest SDI bit rates. An optimized parallel descrambler is actually only about twice as much hardware as the serial descrambler described in the previous section when implemented in a Xilinx FPGA.

A block diagram of the module described in the *par_descrambler.** files is shown in Figure 3. This module accepts a ten-bit input word and generates a ten-bit output word every clock cycle. There is a one-clock cycle latency through the descrambler, caused by the output register.

The NRZI-to-NRZ converter is implemented as ten 2-input gates that XOR each bit with the bit that preceded it in the bitstream. This requires the availability of eleven NRZI bits to generate ten NRZ bits. The eleventh bit comes from storing the MSB of the input-data word in a register making it available to be XORed with the LSB of the data word received in the next clock cycle.

Ten 3-input XOR gates form the SDI descrambler. These gates generate the ten descrambled output bits by combining 19 bits from the NRZI-to-NRZ converter. Ten of the input bits are from the current output of the NRZI-to-NRZ converter. The other nine-input bits were generated by the NRZI-to-NRZ converter during the previous clock cycle and are stored in the *desc_in* register.

As shown in **Reference Design Results**, this parallel descrambler turns out to be fairly small, surprisingly only about twice the size of the serial descrambler, and runs well above the 36 MHz clock rate required to support the highest SDI bit rate.

The parallel descrambler is implemented with a clock enable input called *ld* (Load). If *ld* is asserted for one clock cycle out of every ten, the parallel descrambler can be clocked by the bit-rate clock. If the parallel descrambler receives a word-rate clock, *ld* should be tied High.

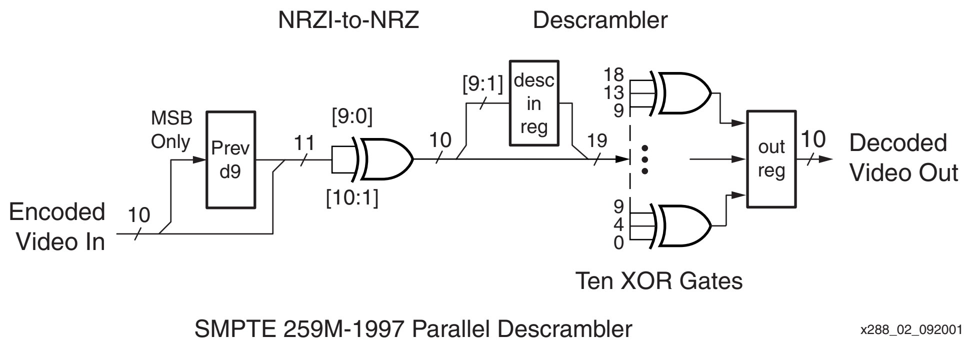

Figure 3: **Parallel Descrambler Block Diagram**

Framer Implementation

The data stream coming out of the descrambler is unframed. There are no indications of where the actual word boundaries occur in the data. If the framer is receiving ten-bit wide parallel data words from the descrambler, the actual video word boundaries do not necessarily correspond to the arbitrary word boundaries of the incoming data.

The framer must scan the unframed data stream for the 30-bit TRS preamble that consists of ten consecutive "1" bits followed by twenty consecutive "0" bits. When a TRS preamble is detected, the framer can resynchronize to the ten-bit word boundaries in the video stream in order to generate properly framed video data words.

Most commercially available SDI framers allow control over whether the framer should resynchronize if it receives a TRS at a new offset. Sometimes an error in the video bitstream will cause a false TRS to be detected. It is also possible that a non-SDI standard compliant data stream will occasionally be transmitted over a SDI link. In this case, the receiver must temporarily disable resynchronization because the non-SDI data may contain bit sequences falsely detected as TRS symbols. These false TRS symbols can be ignored or filtered if the framer has an input to selectively control when resynchronization occurs.

The framer modules each have an input called *frame_en* to control automatic resynchronization. If *frame_en* is Low, the framer will detect new TRS offsets but it will not resynchronize, therefore subsequent data words output by the framer are potentially not framed properly.

The framer module also has a new start position (*nsp*) output. This signal can also be interpreted as an indication of the presence of a framing error. It will be asserted High when a TRS is detected at an offset different from the current offset used by the framer. It will remain asserted until the offset error is corrected by either receiving another TRS matching the offset used by the framer, or by receiving another TRS when *frame_en* is asserted High.

There are several ways to use the *nsp* output and the *frame_en* input to control how and when the framer modules respond to new TRS offsets:

1. If *frame_en* is tied High, the framer will always resynchronize to new TRS offsets.
2. If *frame_en* is tied Low, the framer will not resynchronize. This is primarily useful when the receiver knows the data sent over the interface is non-SDI compliant data. This is generally not useful without control logic to enable and disable the resynchronization at the appropriate times.
3. If *frame_en* is tied to the *nsp* output of the framer, automatic filtering of TRS offsets will occur. When a TRS symbol is detected that is at a new offset, *nsp* will be asserted High but the framer will not change the offset until another TRS symbol is detected. This filters out one-time TRS offset errors.

4. More sophisticated TRS offset filtering algorithms can be implemented by designing a state machine to monitor the *nsp* output and control *frame_en*.

During the time that the four words of the TRS symbol are present on the output, the framer module generates an asserted High *trs* output. The *trs* output is asserted High even if automatic resynchronization has been disabled (*frame_en* Low) and the TRS is at a new offset. In this case, *trs* is asserted, but the TRS symbol coming out of the framer will not be properly framed. If it is desired that *trs* only be asserted when framed TRS symbols are output, it is a simple matter to use *nsp* to qualify the *trs* output.

The framer modules generate ten-bit words. If eight-bit video is being sent through the SDI link, then use only the eight MSBs of the framer module output port.

As with the descrambler function, the framer function can be implemented both in a serial manner and in parallel. The same trade-offs apply. A serial framer will be smaller but must run at the bit rate. A parallel framer will be larger but only needs to run at one-tenth the bit rate.

Serial Framer: ser_framer.*

The block diagram of the serial framer implementation is shown in Figure 4. The incoming video bitstream passes through a series of shift registers that, when combined, delay the video bitstream by 32-clock cycles. These shift registers are the three registers located along the top of Figure 4. The 32-bit delay generated by these video shift registers allows the TRS detection logic to examine 30 consecutive bits for a TRS symbol and determine if the framer needs to be resynchronized before any of those bits appear on the output port.

To detect a TRS preamble, a ten-bit wide AND gate and a ten-bit wide NOR gate determine if the contents of the trs_detect register contains all zeros or all ones. The output of the AND gate (*all_ones*) is delayed through a 20-bit long shift register (*ones_reg*). The output of the NOR gate (*all_zeros*) is delayed through a ten-bit long shift register (*zeros_reg*). By ANDing together the output of the *ones_reg*, *zeros_reg* and the *all_zeros* signal, a *trs_detected* signal is generated to only be asserted when the TRS preamble is contained in the video shift registers.

The offset logic block contains a ten-bit ring counter called *bit_cntr*. This counter is reset to its starting count when the framer resynchronizes. Otherwise, it causes the *out_reg* to load the ten bits contained in the *des_reg* once every ten clock cycles.

The offset logic block generates the *trs*, *nsp*, and *out_rdy* outputs. The *out_rdy* signal is asserted for one clock cycle when the *out_reg* is reloaded. This signal is generated by assigning it to one of the bits of the *bit_cntr*. If downstream logic requires more setup time, the clock cycle when *out_rdy* is asserted can easily be changed by changing the *bit_cntr* bit assigned to *out_rdy*.

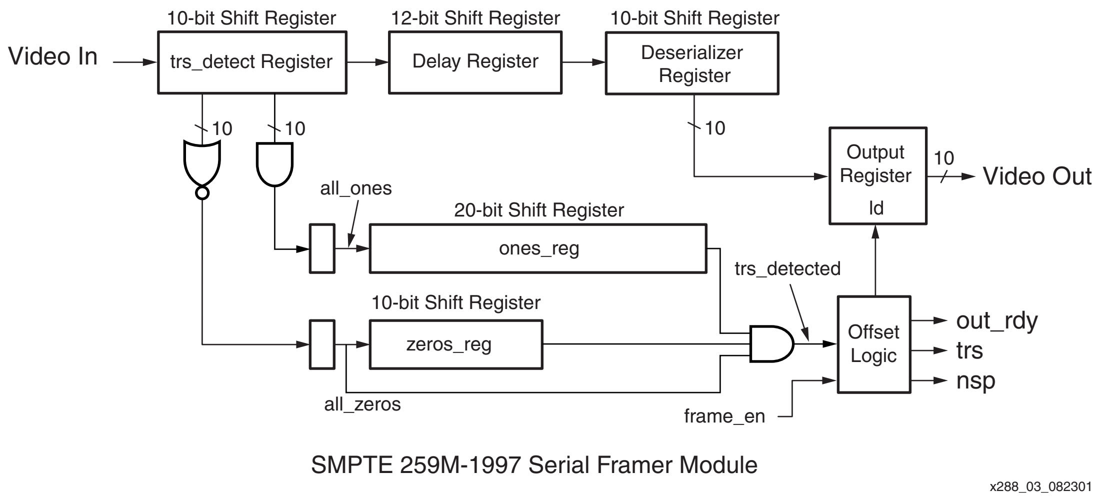

Figure 4: **Serial Framer Block Diagram**

Two slightly different implementations of this serial framer are provided. The *ser_framer.** files are coded to cause the synthesis tool to infer flip-flops for all the shift registers in the module. This results in the best performance and, as can be seen from **Reference Design Results**, it runs at over 400 MHz in a Virtex-II device, sufficient to support the highest bit rates of the SDI specification.

A more compact, but potentially lower performance, implementation is provided in the *ser_framer_srl16*.* files. This version codes the *delay_reg*, *ones_reg*, and *zeros_reg* as arrays, allowing most synthesis tools to infer SRL16 blocks for these registers. This results in a significant reduction in the size of the module (**Reference Design Results**), but may not always produce the fastest results.

Parallel Framer: par_framer.*

Figure 5 shows a block diagram of a parallel implementation of a framer. The *par_framer.** files contain the HDL descriptions of the parallel framer module.

The parallel framer accepts ten-bit unframed data words. It looks for 30-bit TRS preambles that can begin at any of the ten bits in the input word and can span from the first word through the next two or three words. The TRS detection logic needs to look across a total of 39 bits to determine if a TRS symbol is present and to determine its offset.

The incoming data is pipelined through three cascaded registers called *in1_reg*, *in2_reg*, and *in3_reg*. The 30 bits from these three registers plus the nine LSBs from the input port form the 39-bit wide vector that the TRS detection logic examines.

A series of ten-bit wide AND and NOR gates examine the 39-bit input vector to determine if a TRS symbol is present. If so, an internal *trs_detected* signal is asserted, and the offset of the TRS symbol is determined. The offset encoder produces a numerical offset value indicating the starting bit position of the TRS symbol. The output of the offset encoder is compared to the current offset value stored in the offset register to determine if the newly detected TRS symbol is at a different offset position. The nsp logic uses the output of the comparator to generate the *nsp* signal and to load the offset register from the output of the offset encoder when resynchronization occurs. The offset register controls a barrel shifter that extracts the ten-bit output word from a 19-bit wide piece of the input video stream.

As shown in **Reference Design Results**, the parallel framer is fast enough to support all bit rates of the SDI standard.

It is tempting to try to reduce the number of 10-bit wide NOR gates from twenty to ten by using one set of ten gates sequentially. This technique was explored and found to produce about the same size result. With some synthesis tools it actually produced a much larger implementation.

The parallel framer uses a barrel shifter to extract the framed data from the bitstream. In the original Verilog code, this was implemented with a simple assignment statement using the right shift operator. This produced widely varied results with different synthesis tools. The barrel shifter was subsequently re-coded with two levels of multiplexers. This produces good results in all synthesis tools and in both Verilog and VHDL.

When using Virtex-II technology, it is possible to use an embedded 18 x 18 multiplier to implement most of the barrel shifter.[4] The files *par_framer_mult.** use one 18 x 18 embedded multiplier to generate the nine LSBs of the multiplier. A 10-to-1 MUX to is used to generate the tenth bit of the barrel shifter. If it is a Virtex-II design and a free multiplier is available, then using the embedded multiplier results in significant savings in the number of LUTs required to implement the parallel framer module.

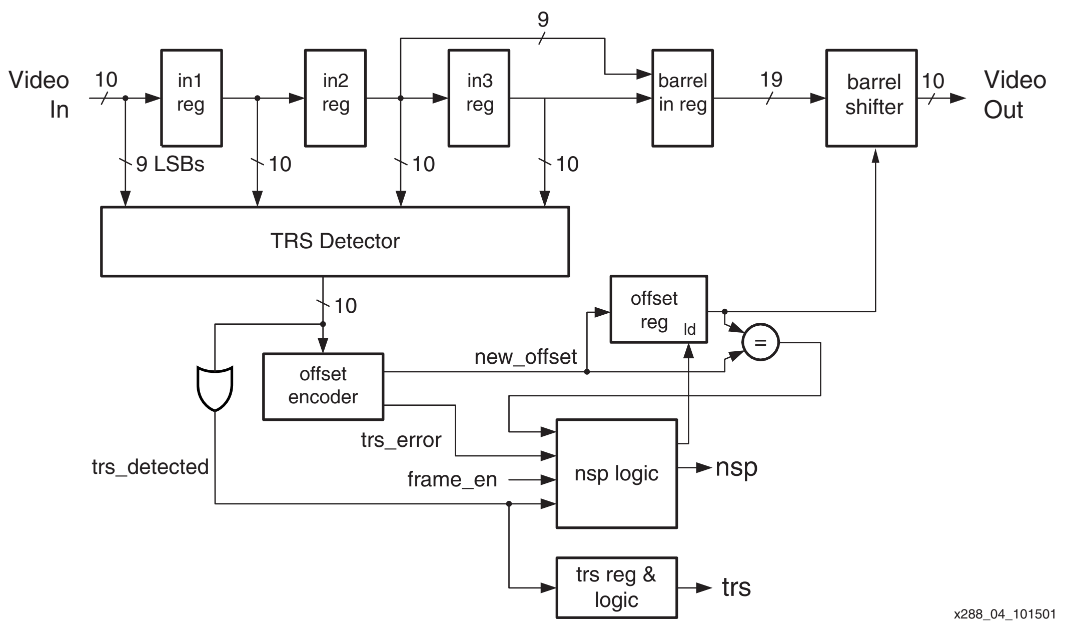

Figure 5: **SMPTE 259M-1997 Parallel Framer Block Diagram**

CLC011 Example: An Alternative Solution

The National CLC011 Serial Digital Video Decoder is a serial implementation of an SDI decoder. It accepts a serial bitstream and a bit-rate clock and produces 10-bit-wide data words on its output.

Using the *ser_descrambler* and *ser_framer* modules, it is easy to implement an alternative to the CLC011 using Xilinx FPGAs. The inputs and outputs of these two modules are very similar in function to the corresponding signals on the CLC011. However, the CLC011 provides two output signals that are not generated by the *ser_framer* module: *PCLK* and *EAV*.

PCLK is used to indicate to downstream logic that data is valid on the PD outputs. The *ser_framer* generates an *out_rdy* signal that is similar to *PCLK* but does not have a 50% duty cycle. The *out_rdy* signal is better suited if the downstream logic is in the same FPGA as *ser_framer* since it can be used as a clock enable signal. If a true *PCLK* signal is required, it is quite simple to modify the *ser_framer* to generate a *PCLK* signal using the *bit_cntr*.

The *EAV* (end of active video) output is asserted Low during the time that the fourth word of a TRS symbol is present on the outputs and bit six (the H bit) of that word is Low.

The *framer_X011 module* generates both the *PCLK* and *EAV* outputs. The combination of the *ser_descrambler* module and the *framer_X011* module completes an alternative implementation of a CLC011.The file *X011.** contains the top level HDL descriptions.

CY7C9335 Example: An Alternative Solution

The Cypress CY7C9335 SMPTE 259M/DVB-ASI Descrambler/Framer-Controller is a parallel implementation of a SDI decoder. It accepts a ten-bit scrambled video word and generates a ten-bit descrambled and framed video word every clock cycle. The *par_descrambler* and *par_framer* modules plus a small amount of extra logic supplies most of the functionality of the CY7C9335. The *X7C9335.** files contain HDL descriptions of the X7C9335 design shown in Figure 6.

The CY7C9335 is designed to operate in two modes, an SDI compliant mode and a DVB-ASI mode when used in conjunction with a Cypress CY7B9334 receiver. The CY7B9334 receiver takes in the serial bitstream, does clock and data recovery, and deserializes the bit-stream before sending it to the CY7C9335 decoder.

In the DVB-ASI mode, the SDI decoder (both the descrambler and the framer functions) are effectively bypassed and the CY7C9335 simply passes the incoming data straight through to its outputs. In this case, the CY7B9334 implements 8B/10B decoding as defined in the DVB-ASI standard. An input signal called *DVB_EN* puts the CY7C9335 into DVB-ASI mode. The CY7C9335 will also bypass the SDI decoding if the *BYPASS* input is High and *DVB_EN* is Low. A multiplexer prior to the output register implements the bypass functionality controlled by *DVB_EN* and *BYPASS*.

The input and output signals of the *X7C9335* are similar in function to the signals of the CY7C9335. The following paragraphs describe these signals.

The *X7C9335* has an input to control how framer resynchronization occurs. This signal, called *sync_en*, will force the *X7C9335* to always resynchronize (*sync_en* Low) or to filter out single erroneous TRS symbols (*sync_en* High).

The CY7C9335 has an output called *RF* that is designed to connect to the CY7B9334. This output is the inverted and registered *DVB_EN* input. The *X7C9335* replicates this signal with its *rf* output.

The *X7C9335* has a horizontal sync output signal called *h_sync*. This signal toggles states every time a TRS symbol is detected by the framer. If sync filtering is enabled (*sync_en* High), *h_sync* still toggles even if the detected TRS symbol is at a new offset position. If the *dvb_en* input is low, *h_sync* will not toggle.

The *X7C9335* has a synchronization error output signal called *sync_err*. If TRS filtering is enabled (*sync_en* High), this signal will pulse High for one (word-rate) clock cycle when the framer filters out a TRS symbol that is offset from the current framer reference.

The CY7C9335 contains a DVB-ASI mode state machine. This state machine generates an output called *A/B* used to control the CY7B9335 device. The purpose of this signal is to cause the CY7B9335 to invert the DVB-ASI data stream if too many errors occur. If DVB-ASI data streams are routed through SDI switches or repeaters, they can become inverted and can not be decoded by the CY7B9335's 8B/10B decoder. By examining the data stream for errors, the state machine will toggle the *A/B* signal if too many errors are detected in order to try and compensate for an inversion in the data stream. This *A/B* output and state machine have not been implemented in the *X7C9335* design.

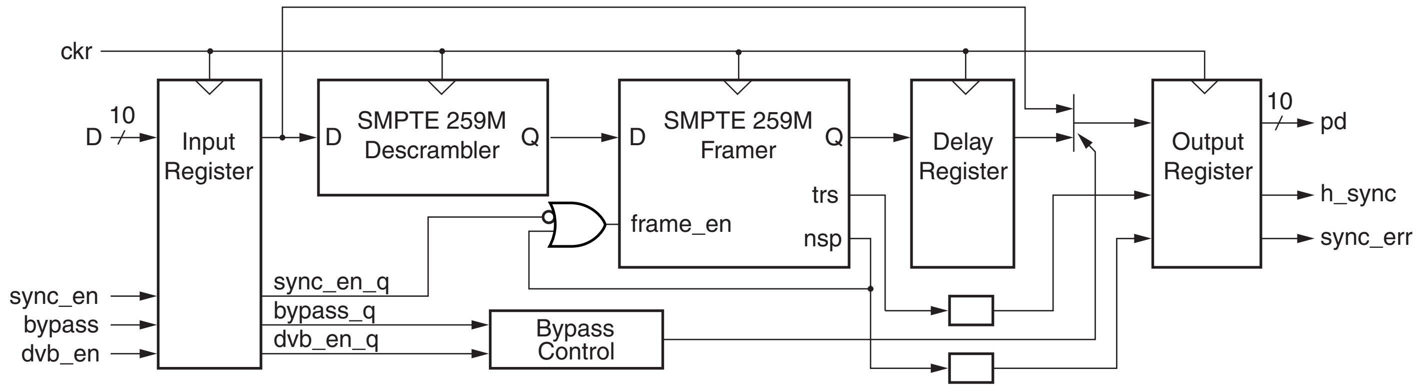

Figure 6: **X7C9335 Block Diagram**

Reference Design Results

The Table 2 shows the results after place and route of the various modules implemented in this application note. All results were obtained using the Verilog versions with Xilinx ISE version 3.3i. Results using the VHDL files are not shown but are essentially identical. Virtex-II results are for a -5 speed grade device. Spartan-II results are for a -6 speed grade device.

Table 2: **Design Results**

File	XST		
	Size in LUTs/FFs	**Virtex-II Speed**	**Spartan-II Speed**
ser_descrambler.v	2/11	490 MHz	300 MHz
par_descrambler.v	19/20	440 MHz	260 MHz
ser_framer.v	19/91	380 MHz	300 MHz
ser_framer_srl16.v	23/53	355 MHz	300 MHz
par_framer.v	84/49	100 MHz	80 MHz
par_framer_mult.v	51/55	85 Mhz	NA[1]
X011.v	24/103	390 MHz	300 MHz
X7C9335.v	115/106	100 MHz	75 MHz

Notes:

1. par_framer_mult is applicable only to Virtex-II devices since it uses an embedded 18 x 18 multiplier.

Testing

The best way to test the SDI decoder modules is in a test bench, connecting to an SDI encoder module being driven by a video generator. The Xilinx application note XAPP298: Serial Digital Interface (SDI) Video Encoder and Test Generator [5]describes not only the design of a SDI encoder module, but also details the implementation of the test bench to test the decoder modules implemented in this application note.

Conclusion

Xilinx FPGAs can implement an SDI decoder function thus replacing costly external components. The Virtex-II devices are fast enough to implement an SDI standard decoder in a serial fashion thus producing a very compact implementation running at the full 360 Mb/s rate. Parallel implementations of the SDI decoder are also possible. These parallel implementations will be somewhat larger, but only need to run at one-tenth the bit rate of the SDI link.

References

1. All the SMPTE standards referenced in this application note are available from The Society of Motion Picture and Television Engineers. These standards can be purchased at the SMPTE web site: http://www.smpte.org.
2. The ITU-R BT.601-5 standard can be purchased from the International Telecommunication Union at http://www.itu.int/itudoc/itu-r/rec/bt/.
3. The IEC 1179 standard is now called the IEC 61179 standard and can be purchased from the International Electrotechnical Commission at http://www.iec.ch/webstore.
4. Xilinx application note XAPP195: Implementing Barrel Shifters Using Multipliers by Paul Glover.
5. Xilinx application note XAPP298: MicroBlaze and Multimedia Development Board: Serial Digital Interface (SDI) Video Encoder and Test Generator by John F. Snow.
6. Xilinx application note XAPP299: MicroBlaze and Multimedia Development Board: Serial Digital Interface (SDI) Ancillary Data and EDH Processors by John F. Snow.

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
10/19/01	1.0	Initial Xilinx release.