Chapter 7

Example Design: I2S Versus SPDIF

The SPDIF (Sony/Philips Digital Interface Format) and I2S (Inter-IC Sound) standards have been developed and used by many consumer electronics manufacturers to provide a means for transmitting digital audio information between ICs and to eliminate the need to transmit analog signals between devices. By keeping the signal digital until the conversion to analog can be localized, it will be less susceptible to noise and signal degradation.

The objective of this chapter is to describe architectures for both I2S and SPDIF receivers and to analyze the method for recovery of the asynchronous signals and resynchronization of the audio data.

7.1 I2S

The I2S format is designed to transmit audio data up to sampling rates of 192 kHz in a source-synchronous fashion. By "source-synchronous" we are referring to the scenario where a clock is transmitted along with the data. With a source-synchronous signal, it is not necessary to share a system clock between the transmitting and receiving device. The sample size of the data can be 16 bits to 24 bits and is normalized to full-scale amplitude regardless of sample size. Unlike the SPDIF format, words of different lengths cannot be interchanged without defining the new size in the receiver.

The main design issue related to I2S is passing the samples between the source clock domain to the local clock domain. Because the signal is transmitted along with the source clock, the data can be easily reconstructed using the source clock and subsequently resynchronized.

Advanced FPGA Design. By Steve Kilts

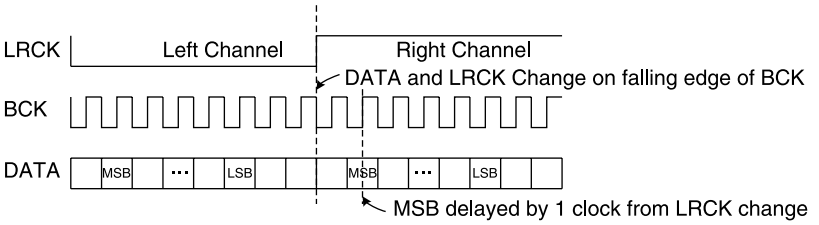

Figure 7.1 I2S timing.

7.1.1 Protocol

I2S has a very simple three-wire synchronous protocol. The three signals are defined as follows:

- LRCK (left/right channel select): When LRCK is low, the data belongs to the left channel, and when LRCK is high, the data belongs to the right channel.
- BCK (bit clock): This is the source-synchronous clock.
- DATA (serial audio data): This provides raw sample bits from the audio codes. The bits are synchronous with BCK.

The timing is illustrated with the waveforms shown in Figure 7.1.

As can be seen from these waveforms, LRCK defines the channel (low = left, high = right), and BCK clocks in the logic value on the DATA line. All transitions of the LRCK and DATA take place on the falling edge of the clock, which allows for a small amount of skew in either direction without violating setup and hold times. The length from the MSB to the LSB is defined by the word size, which is predefined in some manner depending on the application. Note that many I2S receivers have multiple modes outside of the "true" I2S format that are also considered a part of the protocol. These other formats include right and left justification mode, but here we will only consider the I2S format described above. Additionally, we will fix the data word size to 16 bits.

7.1.2 Hardware Architecture

The hardware architecture for an I2S module is very simple as shown in Figure 7.2.

On every rising edge of BCK, the logic value on DATA is clocked into the shift register. When a transition on LRCK is detected, the data word in the shift register is loaded into an output register determined by the polarity of LRCK. The entire I2S circuit uses BCK as the system clock to create a fully synchronous receiver. The data, once latched in the output register, must be passed to the local

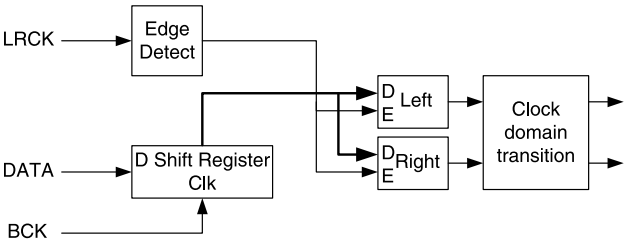

Figure 7.2 I2S architecture.

system clock domain. Thus, the domain transition occurs at the very end of the I2S data recovery. The implementation is shown below.

```
module I2S(
  output reg        oStrobeL, oStrobeR,
  output reg [23:0] oDataL, oDataR,
  input             iBCK, // bit clock
  input             iSysClk, // local system clock
  input             iDataIn,
  input             iLRCK);
  reg               DataCapture;
  reg               rdatain;
  // registers to capture input data on rising and falling
  // edges of clock
  reg        [23:0] Capture;
  // strobes for valid data
  reg               StrobeL, StrobeR;
  reg        [2:0]  StrobeDelayL, StrobeDelayR;
  reg        [23:0] DataL, DataR;
  reg               LRCKPrev;
  reg        [4:0]  bitcounter;
  reg               triggerleft, triggerright;

  wire              LRCKRise, LRCKFall;
  wire       [23:0] DataMux;

  // detect edges of LRCK
  assign LRCKRise = iLRCK & !LRCKPrev;
  assign LRCKFall = !iLRCK & LRCKPrev;

  // assuming 16 bit data
  assign DataMux  = {Capture[15:0], 8'b0};

  always @(posedge iBCK) begin
    DataCapture      <= (bitcounter != 0);
    triggerleft      <= LRCKRise;
    triggerright     <= LRCKFall;
    rdatain          <= iDataIn;
    // for detecting edges of LRCK
    LRCKPrev         <= iLRCK;
```

```
    // capture data on rising edge, MSB first
    if(DataCapture)
      Capture[23:0]  <= {Capture[22:0], rdatain};

    // counter for left justified formats
    if(LRCKRise || LRCKFall)
      bitcounter     <= 16;
    else if(bitcounter != 0)
      bitcounter     <= bitcounter - 1;

    // Load data into register for resynchronization
    if(triggerleft) begin
      DataL[23:0]    <= DataMux;
      StrobeL        <= 1;
    end
    else if(triggerright) begin
      DataR[23:0]  <= DataMux;
      StrobeR       <= 1;
    end
    else begin
      StrobeL        <= 0;
      StrobeR        <= 0;
    end
  end

  // resynchronize to new clock domain
  always @(posedge iSysClk) begin
    // delay strobes relative to data
    StrobeDelayL     <= {StrobeDelayL[1:0], StrobeL};
    StrobeDelayR     <= {StrobeDelayR[1:0], StrobeR};

    // upon the rising edge of the delayed strobe
    // the data has settled
    if(StrobeDelayL[1] & !StrobeDelayL[2]) begin
      oDataL         <= DataL;  // load output
      oStrobeL       <= 1; // single cycle strobe in
                          new domain
    end
    else
      oStrobeL       <= 0;

    if(StrobeDelayR[1] & !StrobeDelayR[2]) begin
      oDataR         <= DataR; // load output
      oStrobeR       <= 1; // single cycle strobe in new
                          domain
    end
    else
      oStrobeR       <= 0;
  end
endmodule
```

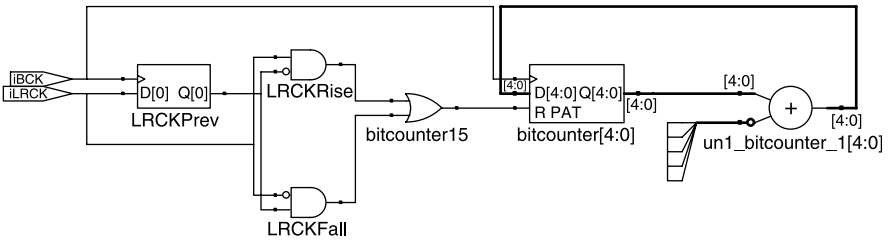

Figure 7.3 LRCK detection.

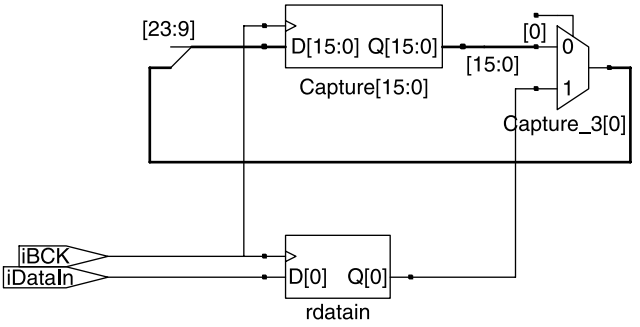

Figure 7.4 Bit capture.

The first step in the above implementation is to detect a transition on LRCK so we can clear the bit counter. This is implemented in a synchronous fashion as shown in Figure 7.3.

Next, we need to begin capturing bits into our shift register as shown in Figure 7.4.

Finally, we use the LRCK trigger to load the shift register into the output register and resynchronize the data with the local clock domain.

7.1.3 Analysis

When capturing and resynchronizing data from a source-synchronous data stream, there are a number of options available to the designer. The three options available with the I2S implementation are

1. Using a delayed validity bit to resynchronize the outputs
2. Double flopping the input stream
3. FIFO outputs

In the above implementation, we chose to use a delayed validity bit. Note that there are a number of design considerations when choosing a method for a particular implementation. The first consideration is speed. The advantage of the

above implementation is that it runs at the audio bit clock speed, which in the worst case (192 kHz) is about 12 MHz. If we were running this module at the system clock speed, we may have to meet timing at perhaps hundreds of megahertz. Clearly, timing compliance will be much easier at the slower clock speed, which will allow the designer flexibility to implement low-area design techniques and allow the synthesis tool to target a compact implementation. The disadvantage is the increased complexity of the clock distribution and timing analysis. The implementation results are shown for each topology at the end of this section.

The scenario where a FIFO would be required at the outputs would arise when the receiving system (located behind the I2S interface) cannot handle periodic bursts of data. If the hardware were a pure pipeline or was at least dedicated to the processing of the incoming audio data, this would not be a problem. However, if the device that is capturing the data accesses the module through a shared bus, the data cannot simply present itself as soon as it is available. In this case, a FIFO provides a clean transition to the new domain as long as the average data rate on the bus end is greater than the audio data rate as shown in Figure 7.5.

The implementation of Figure 7.5 will require dual-port RAM resources as well as some control logic to implement the FIFOs. The final implementation results for all topologies are shown in Table 7.1.

Clearly, there is a significant amount of overhead associated with the FIFO implementation and it would not be a desirable solution unless required by the system.

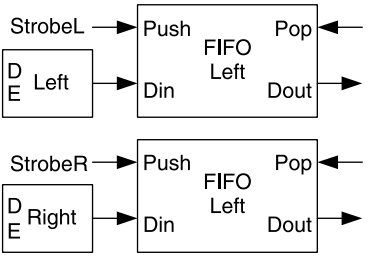

Figure 7.5 FIFO synchronization.

Table 7.1 Implementation Results for I2S Synchronization

Frequency	Double-flop outputs 197 MHz	Double-flop inputs 220 MHz	FIFO outputs 164 MHz
Flip-flops	62	72	130
LUTs	15	35	62
Clock buffers	2	1	2
Block RAMs	0	0	2

7.2 SPDIF

The SPDIF format is designed to transmit audio data up to sampling rates of 192 kHz (until recently, the maximum sampling frequency has been locked at 96 kHz, so many devices will not upsample beyond this prior to transmission). The sample size of the data can be 16 bits to 24 bits and is normalized to full-scale amplitude regardless of sample size. In other words, additional bits are automatically detected as additional bits of precision and not an increase in absolute amplitude. From an implementation perspective, a 16-bit word can be viewed as a 24-bit word with 8 bits of zeros appended to the least significant bits of precision. Thus, capturing the data word is the same regardless of word size (contrast this with I2S, which must have word size and format defined prior to capture).

The main design issue related to SPDIF is its asynchronous nature. Because the signal is transmitted via only one wire, there is no way to directly synchronize to the transmitting device and ultimately the audio signal. All of the information necessary to recover the clock is encoded into the serial stream and must be reconstructed before audio information can be extracted.

7.2.1 Protocol

Each sample of audio data is packetized into a 32-bit frame that includes additional information such as parity, validity, and user-definable bits (the user bits and even the validity bits are often ignored in many general-purpose devices). For stereo applications, two frames must be transmitted for each sample period. Thus, the bit rate must be $32*2*F_s$ (2.8224 MHz for 44.1 kHz, 6.144 MHz for 96 kHz, etc). The 32-bit packet format is defined in Table 7.2.

In the implementation described in this chapter, we will only decode the audio data and preamble.

To enable the SPDIF receiver to identify distinct bits as well as to resynchronize the packets, a special one-wire encoding is used called Biphase Mark Code (BMC). With this form of encoding, the data signal transitions on every bit regardless of whether it is encoded as a 1 or a 0. The difference between these

Table 7.2 SPDIF Frame Definition

Bits	Field
31	Parity (not including the preamble)
30	Channel status information
29	Subcode data
28	Validity (0 = valid)
27 : 4	Audio sample (MSB at bit 27)
3 : 0	Preamble

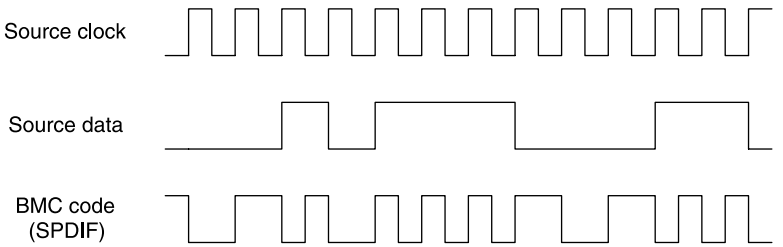

Figure 7.6 Example BMC encoding.

Table 7.3 SPDIF preambles

Preamble	SPDIF signal if last level = 0	SPDIF signal if last level = 1
Left channel at the start of a data block	11101000	00010111
Left channel not at the start of a data block	11100010	00011101
Right channel	11100100	00011011

bits is that the SPDIF signal will transition once per bit for a logic-0 and twice per bit for a logic-1. An example encoding is shown in Figure 7.6.

The first two waveforms shown in Figure 7.6 are the clock and data seen by the transmitter. In a synchronous transmission medium such as I2S, this clock as well as the synchronized data are passed to the receiver making the data recovery trivial. When only one wire is available, the data is encoded in BMC format as shown in the third waveform. As can be seen from this waveform, the clock is encoded into the data stream with the requirement of at least one transition for every bit. Note that the clock that sources the SPDIF stream must be twice the frequency of the audio clock to provide two transitions for every logic-1.

Due to the fact that the encoding of a data bit must transition once per bit, SPDIF provides a means to synchronize each frame by violating this condition once per frame. This is performed in the preamble as shown in Table 7.3.

As can be seen from these bit sequences, each preamble violates the transition rule by allowing a sequence of three consecutive clock periods of the same level. Detecting these preambles allows the receiver to synchronize the audio data to the appropriate channel. For a hardware implementation, a clock with a sufficient frequency must be used to be able to not only distinguish the difference between a logic-0 and a logic-1 (a 2× difference in pulse widths) but also a difference between a logic-0 and a preamble (a 1.5× difference in pulse widths).

7.2.2 Hardware Architecture

The basic architecture for the SPDIF receiver is shown in Figure 7.7.

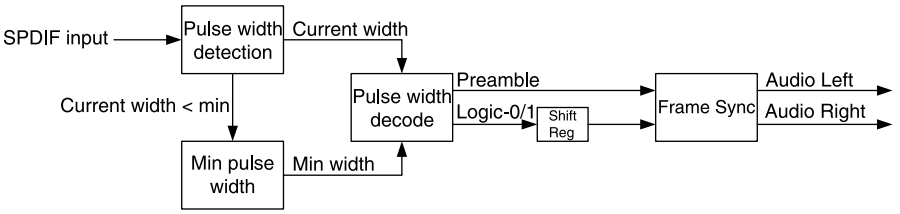

Figure 7.7 SPDIF architecture.

The pulse width detection logic contains a free-running counter that resets whenever the input from the BMC code toggles. In addition to the counter reset, the current width is decoded against the running minimum pulse width. If the current width is greater than 2.5× the minimum pulse width, the pulse is decoded as a BMC violation and part of the preamble. If the width is greater than 1.5× the min pulse width, the pulse is decoded as a logic-0. If the width is less than the running minimum pulse width, it overrides the minimum pulse width, and the audio data is assumed to be invalid due to the absence of a lock. Otherwise, the pulse is decoded as half of a logic-1.

If a logic-1 or logic-0 is detected, this bit is shifted into a 24-bit shift register in preparation for synchronization with the preamble. When a preamble is detected, the previous frame has completed and can now be decoded based on the mapping of the various fields. The implementation is shown in the following code.

```
module spdif(
  output reg        oDatavalidL, oDatavalidR,
  output reg [23:0] oDataL, oDataR,
  input             iClk, // main system clock used to
                          sample spdif data
  input             iSPDIFin);
  reg        [2:0]  inputsr; // input shift register
  reg               datatoggle; // register pulses high
                                   when data toggles
  // counts the width between data transitions
  reg        [9:0]  pulsewidthcnt;
  // register to hold width between transitions
  reg        [9:0]  pulsewidth;
  reg        [9:0]  onebitwidth; // 1-bit width reference
  // signals that pulsewidth has just become valid
  reg               pulsewidthvalid;
  reg               bitonedet; // detect logic-1 capture
  reg               newbitreg; // new data registered
  reg        [27:0] framecapture; // captured frame
  reg               preambledetect;
  reg               preamblesyncen;
  reg               channelsel;  // select channel based
                                    on preamble
```

```
reg        [5:0]  bitnum;
reg        [10:0] onebitwidth1p5;

reg               onebitload; // load 1-bit reference
                                width
reg               onebitupdown; // 1: reference width
                                  should increment
// width used for comparison against reference
reg        [9:0]  pulsewidthcomp;
reg               onebitgood; // reference is equal to
                                input width
reg               preamblesync; // flags preamble in
                                  spdif stream
reg               shiftnewdat; // ok to capture
// load data into output buffer
reg               outputload, outputloadprev;
reg               pulsewidthsmall, pulsewidthlarge;
reg        [11:0] onebitwidth2p5;
wire              trigviolation;
wire              newbit;  // raw data decoded from stream

// flag a violation in BMC code
assign trigviolation   = {1'b0, pulsewidth[9:0], 1'b0} >
                         onebitwidth2p5;

// if width is small, data is 1.  Otherwise data is 0
assign newbit          = ({pulsewidth[9:0],1'b0} <
                           onebitwidth1p5[10:0]);

always @(posedge iClk) begin
  inputsr              <= {inputsr[1:0], iSPDIFin};
// shift data in
// trigger on change in data
  datatoggle           <= inputsr[2] ^ inputsr[1];

// counter for pulse width
  if(datatoggle) begin
// counter resets when input toggles
    pulsewidth[9:0]    <= pulsewidthcnt[9:0];
    pulsewidthcnt      <= 2;
  end
  else
    pulsewidthcnt      <= pulsewidthcnt + 2;

  // width register will be valid 1 clock after the data
     toggles
  pulsewidthvalid      <= datatoggle;

  // onebitload checks to see if input period is out of
     bounds
  // current width is 1/2 1-bit width
  pulsewidthsmall      <= ({1'b0, onebitwidth[9:1]} >
                           pulsewidth[9:0]);
```

```
// current width is 4x 1-bit width
pulsewidthlarge      <= ({2'b0, pulsewidth[9:2]} >
                         onebitwidth);
// load new reference if out of bounds
onebitload           <= pulsewidthlarge || pulse
                        widthsmall;

// register width comparison value
if(!newbit)
  pulsewidthcomp     <= {1'b0, pulsewidth[9:1]};
else
  pulsewidthcomp     <= pulsewidth[9:0];

// checks to see if reference is equal to input width
onebitgood           <= (pulsewidthcomp == onebit
                         width);
// increment reference if input width is greater than
   reference
onebitupdown         <= (pulsewidthcomp > onebitwidth);

// keep track of 1-bit width
// load reference if input width is out of bounds
if(onebitload)
  onebitwidth        <= pulsewidth[9:0];
else if(!onebitgood && pulsewidthvalid) begin
  // adjust reference
  if(onebitupdown)
    onebitwidth      <= onebitwidth+1;
  else
    onebitwidth      <= onebitwidth-1;
end

// set onebitwidth*1.5 and onebitwidth*2.5
onebitwidth1p5       <= ({onebitwidth[9:0], 1'b0} +
 {1'b0, onebitwidth[9:0]});
onebitwidth2p5       <= ({onebitwidth[9:0], 2'b0} +
 {2'b0, onebitwidth[9:0]});
// preamblesync is valid only when last frame has
  completed
preamblesyncen       <= (bitnum == 0) && datatoggle;
// trigger on preamble in spdif header if input width
   > 2.5*reference
preamblesync         <= preamblesyncen && trigviolation;

// capture preamble
if(preamblesync)
  preambledetect     <= 1;
else if(preambledetect && pulsewidthvalid)
  preambledetect     <= 0;

// set channel
if(preambledetect && pulsewidthvalid)
```

```
      channelsel          <= !trigviolation;
    else if(trigviolation && pulsewidthvalid)
      channelsel          <= 0;

    newbitreg             <= newbit;
    // only trigger on a bit-1 capture every other transition
    if(!newbitreg)
      bitonedet           <= 0;
    else if(newbit && datatoggle)
      bitonedet           <= !bitonedet;

    // set flag to capture data when bit-0 or bit-1 is valid
    shiftnewdat           <= pulsewidthvalid && (!newbit ||
                             bitonedet);

    // shift register for capture data
    if(shiftnewdat)
      framecapture[27:0] <= {newbit, framecapture[27:1]};

    // increment bit counter when new bit is valid
    // reset bit counter when previous frame has finished
    if(outputload)
      bitnum              <= 0;
    else if(preamblesync)
      bitnum              <= 1;
    else if(shiftnewdat && (bitnum != 0))
      bitnum              <= bitnum + 1;

    // data for current frame is ready
    outputload            <= (bitnum == 31);
    outputloadprev        <= outputload;

    // load captured data into output register
    if(outputload & !outputloadprev) begin
      if(channelsel) begin
        oDataR            <= framecapture[23:0];
        oDatavalidR       <= 1;
      end
      else begin
        oDataL            <= framecapture[23:0];
        oDatavalidL       <= 1;
      end
    end
    else begin
      oDatavalidR         <= 0;
      oDatavalidL         <= 0;
    end
  end
endmodule
```

The first step in the above architecture is to resynchronize the incoming data stream to the local system clock. A double-flop technique is used as described in previous chapters for passing a single bit across domains. This is shown in Figure 7.8.

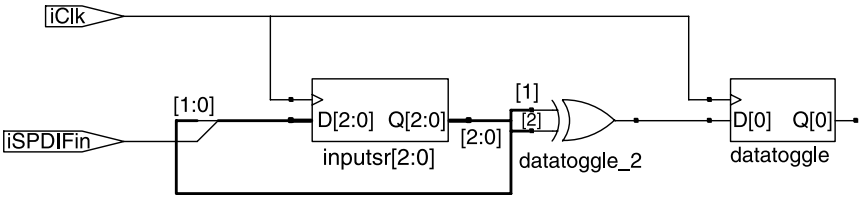

Figure 7.8 Resynchronizing the SPDIF input.

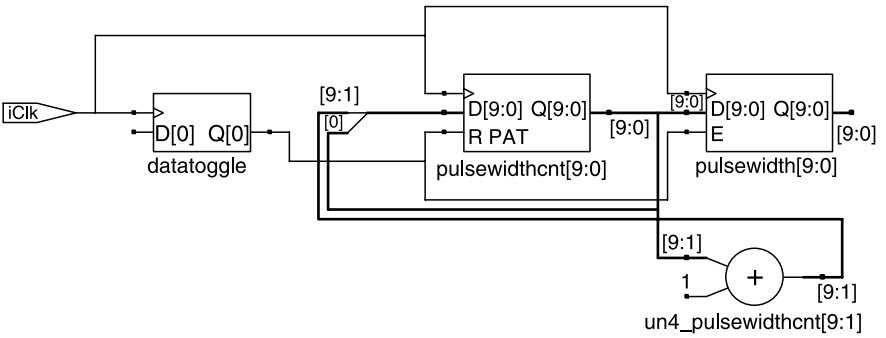

Figure 7.9 SPDIF pulse width counter.

Note that the bits that are used for edge detection are bits 2 and 1. Bits 0 and 1 in the shift register are used for clock synchronization only, and bit 2 is used for the detection of a transition. The synchronized toggle flag in datatoggle is used to reset the counter for the pulse width as shown in Figure 7.9. Notice how the synthesis tool was able to utilize the reset and clock enable pins of the flip-flop elements and eliminate any muxing. This was described in Chapter 2.

The next step is to determine if the pulse width is out of acceptable bounds and whether we need to reset the running value for a 1-bit width. The logic shown in Figure 7.10 performs the boundary condition checks and sets a bit to reload the reference width.

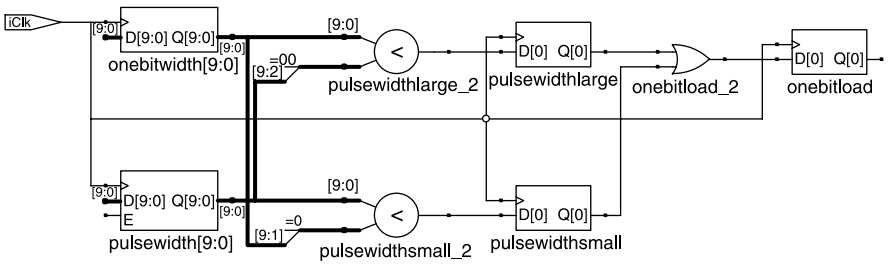

Figure 7.10 Pulse width reference.

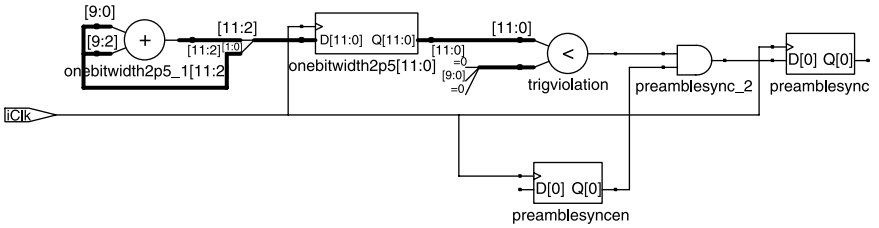

Figure 7.11 Preamble detection.

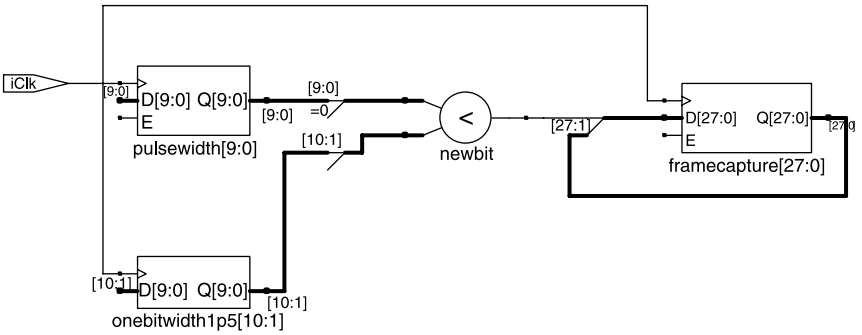

Figure 7.12 Bit detection.

The next block of logic is to detect a preamble. Figure 7.11 shows the scaling of the reference width by 2.5 and performing of the frame synchronization.

Note in the implementation of Figure 7.11 that the factor of 2.5 was optimally implemented by a simple shift and add of the original signal. Similarly, we need to determine if the pulse width is indicating a bit-0 or a bit-1 (assuming the pulse width is not indicating a preamble).

In the circuit shown in Figure 7.12, the data that is shifted into the framecapture shift register is dependent on the width of the current pulse. In other words, if the current pulse width is less than 1.5× the pulse width of the value of a single bit width, the data shifted in is a logic-1. Otherwise, the data is a logic-0.

Finally, a transition on the output load is detected (dependent on the bit counter), the channel is selected, and the frame data is loaded into the appropriate output register as shown in Figure 7.13.

7.2.3 Analysis

When resynchronizing a signal with an encoding such as BMC, there is no choice but to sample this signal at the front end and map it into the local clock domain. No processing can take place until this initial resynchronization occurs. Additionally, the system clock that is used to sample the SPDIF stream must be sufficiently faster than the minimum pulse width of the SPDIF stream itself to provide

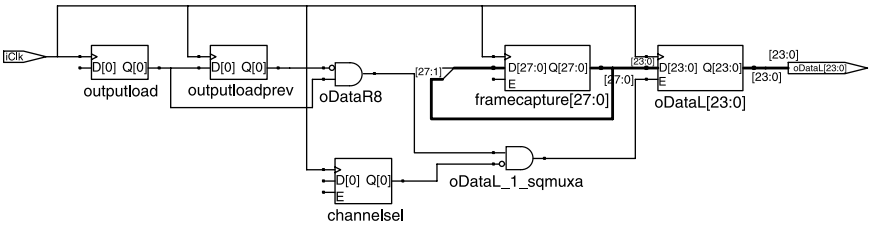

Figure 7.13 SPDIF output Synchronization

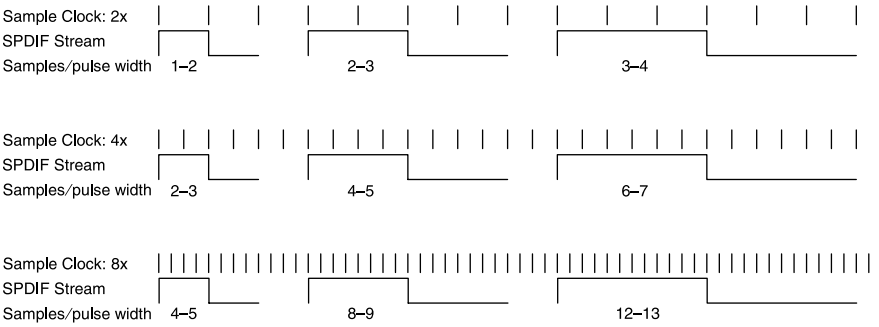

Figure 7.14 SPDIF sampling rates.

enough resolution when detecting thresholds in the pulse width. Specifically, under all relative phases of the sampling clock to the SPDIF stream, we require the following:

- The pulse width of a logic-0 is between 1.5× and 3× of the minimum pulse width (logic-1).
- The pulse width of a preamble violation is between 2.5× and 4× of the minimum pulse width.
- There are at least two clock periods of margin in the thresholds to account for jitter of either the input stream or the system clock.

Figure 7.14 illustrates the various sampling rates.

As can be seen from this diagram, the criteria for reliable signal recovery is when we have a sampling rate of at least 8× the maximum clock frequency (full

Table 7.4 Implementation Results in a Xilinx Spartan-3 XC3S50

Frequency	130 MHz
FFs	161
LUTs	153

period for a logic-1). For a 192-kHz sampling rate, this corresponds with a worst-case timing of: 192 kHz-64*8 = 98.304 MHz. If we target this at a Xilinx Spartan-3 device with a 10 ns period (allowing for about 100 ps of jitter), we obtain the results shown in Table 7.4.

Although we can easily achieve the desired frequency, the logic required to implement the signal recovery is large relative to a source-synchronous system such as I2S.